開發人員指南

AWS Lambda

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS Lambda: 開發人員指南

Copyright © 2026 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon 的商標和商業外觀不得用於任何非 Amazon 的產品或服務，也不能以任何可能造成客戶混淆、任何貶低或使 Amazon 名譽受損的方式使用 Amazon 的商標和商業外觀。所有其他非 Amazon 擁有的商標均為其各自擁有者的財產，這些擁有者可能附屬於 Amazon，或與 Amazon 有合作關係，亦或受到 Amazon 贊助。

Table of Contents

什麼是 AWS Lambda？ 1
Lambda 的運作方式 1
主要功能 2
相關資訊 3
運作方式 3
Lambda 函式與函式處理常式 3
Lambda 執行環境與執行時期 4
事件與觸發程序 4
Lambda 許可與角色 5
執行程式碼 7
建立事件驅動型架構 21
設計應用程式 31
建立第一個函數 38
先決條件 38
建立函數 40
調用函數 45
清除 49
後續步驟 49
範例應用程式與模式 51
檔案處理 51
資料庫整合 51
排程任務 51
長時間執行的工作流程 52
其他資源 52
檔案處理應用程式 52
建立原始程式碼檔案 53
部署應用程式 56
測試應用程式 66
後續步驟 73
已排定維護應用程式 74
先決條件 74
下載範例應用程式檔案 75
建立範例 DynamoDB 資料表並填入資料 84
建立排定維護應用程式 87

測試應用程式 91
後續步驟 92
使用 Lambda 耐久函數建立訂單處理系統 92
先決條件 92
建立來源碼檔案 92
部署應用程式 93
測試應用程式 94
後續步驟 95
開發工具 97
本機開發工具 97
基礎結構即程式碼 (IaC) 工具 98
GitHub Actions 工具 98
AWS Lambda 的 Powertools 99
工作流程與事件管理工具 99
本機開發 99
本機開發的主要優勢 99
必要條件 100
身分驗證與存取控制 100
從主控台遷移至本機開發 103
在本機使用函式 103
將函式轉換為 AWS SAM 範本並使用 IaC 工具 106
後續步驟 106
GitHub Actions 106
範例工作流程 107
其他資源 108
基礎設施即程式碼 (IaC) 108
用於 Lambda 的 IaC 工具 108
使用 AWS SAM 與 Infrastructure Composer 109
使用 AWS CDK 120
Powertools for AWS Lambda 129
Powertools for AWS 的關鍵優勢 130
將 Powertools 與 AWS 整合 130
後續步驟 106
工作流程與事件 131
具有耐用函數的程式碼優先協同運作 131
使用 Step Functions 協同運作工作流程 131

使用 EventBridge 與 EventBridge 排程器管理事件 133
Lambda 耐用函數 134
主要優點 134
使用耐用函數的時機 134
與 Step Functions 比較的耐久性函數 135
運作方式 135
後續步驟 136
基本概念 137
持久性執行 137
DurableContext 137
步驟 138
等待狀態 139
叫用其他 函數 140
耐用的函數組態 141
另請參閱 142
建立耐用的函數 142
先決條件 143
建立耐用的 函數 145
叫用耐用的 函數 151
清除 153
後續步驟 154
使用 CLI 部署 154
使用基礎設施做為程式碼 161
耐用函數或 Step Functions 171
何時使用耐用函數 171
何時使用 Step Functions 171
決策架構 172
功能比較 172
混合架構 173
遷移考量事項 173
相關資源 173
範例 173
短期容錯程序 173
長時間執行的程序 178
進階模式 185
後續步驟 197

安全與許可 197
執行角色許可 198
狀態加密 199
CloudTrail 日誌記錄 199
跨帳戶考量事項 200
繼承的 Lambda 安全功能 200
耐用的執行 SDK 200
DurableContext 201
開發套件的功能 202
檢查點的運作方式 202
重播行為 203
可用的耐用操作 203
如何計量耐久性操作 212
支援的執行時期 214
叫用耐用函數 218
調用方法 218
合格 ARNs需求 219
了解執行生命週期 220
從應用程式碼叫用 221
鏈結調用 222
事件來源映射 223
重試 230
冪等性 237
測試耐用函數 241
本機測試 241
雲端測試 244
測試項目 245
測試策略 246
偵錯失敗 246
監控耐用的 函數 246
CloudWatch 指標 246
EventBridge 事件 248
AWS X-Ray 追蹤 250
最佳實務 251
寫入確定性程式碼 251
等冪性的設計 255

有效率地管理狀態 257
設計有效的步驟 258
有效率地使用等待操作 259
其他考量 260
其他資源 260
Lambda 受管執行個體 262
關鍵功能 262
何時使用 Lambda 受管執行個體 262
運作方式 262
並行模型 263
租用和隔離 263
了解受管執行個體 263
定價 264
Lambda 受管執行個體與 Lambda （預設） 運算類型有何不同 264
後續步驟 265
開始使用 265
建立 Lambda 受管執行個體函數 （主控台） 265
建立 Lambda 受管執行個體函數 (AWS CLI) 266
核心概念 272
容量提供者 272
擴展 276
安全 279
Operator 角色 280
執行環境 283
版本發佈 285
執行時間 285
支援的語言 286
語言特定考量事項 286
後續步驟 287
Java 執行時間 287
Node.js 執行時間 291
Python 執行時間 296
.NET 執行時間 298
Rust 302
聯網 304
連線選項 305

具有網際網路閘道的公有子網路 305
VPC 端點 306
具有 NAT 閘道的私有子網路 306
選擇連線選項 307
後續步驟 308
監控 308
可用的指標 308
指標頻率和保留 309
在 CloudWatch 中檢視指標 309
使用指標來最佳化效能 310
後續步驟 310
配額 310
Lambda API 請求配額 310
Lambda 受管執行個體資源配額 311
事件來源映射配額 311
請求提高配額 312
後續步驟 312
最佳實務 312
容量提供者組態 312
執行個體類型選取 313
函數組態 313
擴展組態 313
安全 314
成本最佳化 314
監控與可觀測性 314
語言特定考量事項 315
後續步驟 315
疑難排解 315
調節和擴展問題 315
並行問題 316
效能問題 318
容量提供者問題 319
監控和可觀測性問題 320
取得其他協助 322
後續步驟 322
Lambda 執行時間 323

支援的執行時期 323
新執行期版本 326
執行期淘汰政策 327
共同責任模型 327
棄用後的執行時期使用情況 328
接收執行期棄用通知 330
已取代的執行階段 331
執行時期版本更新 334
回溯相容性 335
執行時期更新模式 336
兩階段執行階段版本推展 336
設定執行時期管理 337
執行時期版本復原 338
執行時期版本更新 339
共同責任模式 341
許可 342
依執行時期取得關於函數的資料 344
列出使用特定執行時期的函數版本 344
識別最常調用和最近調用的函數 346
執行階段修改 350
特定語言的環境變數 350
包裝函式指令碼 350
Runtime API 354
下次調用 355
調用回應 356
初始化錯誤 356
調用錯誤 357
僅限作業系統的執行期 360
建置自訂執行時間 361
自訂執行期教學 365
開放原始碼儲存庫 373
執行時期介面用戶端 373
事件程式庫 373
容器基礎映像 374
開發工具 374
專案範例 375

設定函數 376
.zip 封存檔 378
建立函數 378
使用主控台程式碼編輯器 380
更新函數程式碼 380
變更執行階段 380
變更架構 381
使用 Lambda API 381
下載函式程式碼 382
CloudFormation 382
加密 383
容器映像 389
要求 390
使用 AWS 基礎映像 391
使用僅限 AWS 作業系統的基礎映像 392
使用非AWS 基礎映像 392
執行時期介面用戶端 393
Amazon ECR 許可 393
函數生命週期 396
記憶體 397
何時增加記憶體 397
使用主控台 397
使用 AWS CLI 398
使用 AWS SAM 398
接受函數記憶體建議 (主控台) 399
暫時性儲存 400
使用案例 400
使用主控台 400
使用 AWS CLI 401
使用 AWS SAM 401
指令集 (ARM/x86) 403
使用 arm64 架構的優點 403
遷移到 arm64 架構的要求 404
函數程式碼與 arm64 架構的相容性 404
如何遷移到 arm64 架構 404
設定指令集架構 405

Timeout (逾時) 406
何時增加逾時 406
使用主控台 406
使用 AWS CLI 407
使用 AWS SAM 407
設定耐用的函數 408
啟用持久性執行 408
耐用函數的 IAM 許可 409
設定最佳實務 410
環境變數 412
建立環境變數 412
環境變數的範例案例 415
擷取環境變數 417
定義執行時間環境變數 418
保護環境變數 420
將函數附接至 VPC 424
所需的 IAM 許可 424
將 Lambda 函數連接至 中的 Amazon VPC AWS 帳戶 426
連接到 VPC 時的網際網路存取 429
IPv6 支援 429
搭配使用 Lambda 與 Amazon VPC 的最佳實務 430
了解 Hyperplane Elastic Network Interfaces (ENI) 432
使用 IAM 條件金鑰進行 VPC 設定 432
VPC 教學課程 437
將函數附接至另一個帳戶中的資源 438
先決條件 438
在函數帳戶中建立 Amazon VPC 438
為函數的執行角色授予 VPC 許可 439
建立 VPC 對等互連請求 440
準備資源的帳戶 440
更新函數帳戶中的 VPC 組態 442
測試 函數 443
Lambda 函數的網際網路存取 444
傳入網路 468
Lambda 介面端點的考量 468
為 Lambda 建立介面端點 469

為 Lambda 建立介面端點政策 470
檔案系統 472
支援的地區 472
執行角色和使用者許可 472
設定檔案系統和存取點 473
連線至檔案系統 (主控台) 474
別名 475
使用別名 477
加權別名 477
版本 482
建立函數版本 482
使用版本 483
授予許可 485
Tags (標籤) 486
使用標籤所需的許可 486
搭配使用標籤與主控台 486
搭配 使用標籤 AWS CLI 487
回應串流 490
回應串流的頻寬限制 491
VPC 與回應串流的相容性 491
撰寫函數 491
調用函數 493
教學課程：建立具有函數 URL 的回應串流函數 494
中繼資料端點 500
開始使用 500
了解可用區域 IDs 503
API 參考 503
調用函數 505
同步調用函數 506
非同步調用 509
錯誤處理 510
組態 510
保留記錄 512
調用 521
同步調用限制 521
長時間執行工作流程的非同步調用 521

執行管理 APIs ... 522
事件來源映射 ... 524
事件來源映射和觸發條件 ... 524
批次處理行為 ... 525
佈建模式 ... 528
事件來源映射 API ... 529
事件來源映射標籤 ... 530
事件篩選 ... 534
了解事件篩選基本知識 ... 535
處理不符合篩選條件標準的記錄 ... 537
篩選條件規則語法 ... 537
將篩選條件標準連接至事件來源映射 (主控台) ... 539
將篩選條件標準連接至事件來源映射 (AWS CLI) ... 540
將篩選條件標準連接至事件來源映射 (AWS SAM) ... 541
篩選條件加密 ... 542
使用具有不同 的篩選條件 AWS 服務 ... 548
在主控台中測試 ... 549
使用測試事件調用函數 ... 549
建立私有測試事件 ... 549
建立可共用測試事件 ... 550
刪除可共用測試事件結構描述 ... 551
函數狀態 ... 552
更新期間的函式狀態 ... 553
重試 ... 555
遞迴迴圈偵測 ... 556
了解遞迴迴圈偵測 ... 556
支援的 AWS 服務 和 SDKs ... 557
遞迴迴圈通知 ... 559
回應遞迴迴圈偵測通知 ... 560
允許 Lambda 函數在遞迴迴圈中執行 ... 561
支援 Lambda 遞迴迴圈偵測功能的區域 ... 563
函數 URL ... 564
建立函數 URL (主控台) ... 565
建立函數 URL (AWS CLI) ... 567
將函數 URL 新增至 CloudFormation 範本 ... 567
跨來源資源共享 (CORS) ... 569

調節函數 URL 570
停用函數 URL 571
刪除函數 URL 571
存取控制 571
呼叫函數 URL 581
監控函數 URL 592
函式 URL 與 Amazon API Gateway 比較 593
教學課程：建立 Webhook 端點 598
函數擴展 612
了解和視覺化並行 612
計算函數的並行 616
了解預留並行和佈建並行 617
預留並行 618
佈建並行 619
Lambda 如何配置佈建並行 623
比較預留並行和佈建並行 623
了解並行和每秒請求數 624
並行配額 626
設定預留並行 628
設定預留並行 628
準確估計函數所需的預留並行 630
設定佈建並行 632
設定佈建並行 633
準確估算函數所需的佈建並行 634
使用佈建並行時最佳化函數程式碼 635
使用環境變數來檢視和控制佈建並行行為 636
了解使用佈建並行的記錄和計費行為 636
使用 Application Auto Scaling 自動化佈建並行管理 637
擴展行為 641
並行擴展率 641
監控並行 642
一般並行指標 642
佈建並行指標 642
使用 ClaimedAccountConcurrency 指標 644
使用 Node.js 進行建置 647
包含執行時期的 SDK 版本 648

使用保持連線 649
CA 憑證載入 649
實驗 Node.js 功能 650
處理常式 651
設定專案 651
範例函數 652
CommonJS 和 ES 模組 654
Node.js 初始化 655
處理常式命名慣例 655
輸入事件物件 656
有效的處理常式模式 657
使用適用於 JavaScript 的 開發套件 658
存取環境變數 659
使用全域狀態 660
最佳實務 660
部署 .zip 封存檔 662
Node.js 中的執行期相依項 662
建立不含相依項的 .zip 部署套件 663
建立含相依項的 .zip 部署套件 663
為相依項建立 Node.js 層 664
相依項搜尋路徑和含執行期程式庫 665
使用 .zip 檔案建立及更新 Node.js Lambda 函數 666
部署容器映像 672
AWS Node.js 的基礎映像 672
使用 AWS 基礎映像 673
使用非AWS 基礎映像 679
層 689
封裝層內容 689
在 Lambda 中建立層 694
將層新增至函式 694
應用程式範例 695
Context 696
日誌 698
建立傳回日誌的函數 698
搭配 Node.js 使用 Lambda 進階日誌控制項 700
在 Lambda 主控台檢視日誌 706

在 CloudWatch 主控台中檢視 記錄 706
使用 AWS Command Line Interface (AWS CLI) 檢視日誌 706
刪除日誌 709
追蹤 710
使用 ADOT 來檢測您的 Node.js 函數 711
使用 X-Ray SDK 來檢測 Node.js 函數 711
透過 Lambda 主控台來啟用追蹤 712
透過 Lambda API 啟用追蹤 713
使用 啟用追蹤 CloudFormation 713
解讀 X-Ray 追蹤 714
將執行時間相依項存放存在層中 (X-Ray SDK) 716
使用 TypeScript 建置 718
開發環境 719
Lambda 的類型定義 720
處理常式 721
設定專案 721
範例函數 722
CommonJS 和 ES 模組 724
Node.js 初始化 725
處理常式命名慣例 725
輸入事件物件 725
有效的處理常式模式 726
使用適用於 JavaScript 的 開發套件 729
存取環境變數 729
使用全域狀態 730
最佳實務 730
部署 .zip 封存檔 732
使用 AWS SAM 732
使用 AWS CDK 733
使用 AWS CLI 和 esbuild 736
部署容器映像 739
使用 Node.js 基礎映像來建置和封裝 TypeScript 函數程式碼 739
Context 746
日誌 748
工具與程式庫 748
使用 Powertools for AWS Lambda (TypeScript) 和 AWS SAM 進行結構化記錄 748

使用 Powertools for AWS Lambda (TypeScript) 和 AWS CDK 進行結構化記錄 751
在 Lambda 主控台檢視日誌 755
在 CloudWatch 主控台中檢視 記錄 755
追蹤 756
使用 Powertools for AWS Lambda (TypeScript) 和 AWS SAM 進行追蹤 756
使用 Powertools for AWS Lambda (TypeScript) 和 AWS CDK 進行追蹤 758
解讀 X-Ray 追蹤 762
使用 Python 建置 764
包含執行時期的 SDK 版本 765
停用的 Python 功能 766
回應格式 766
延伸模組正常關機 766
處理常式 768
Python Lambda 函式程式碼範例 768
處理常式命名慣例 770
使用 Lambda 事件物件 771
存取和使用 Lambda 內容物件 772
Python 處理常式的有效處理常式簽章 772
傳回值 773
在處理常式中使用 適用於 Python (Boto3) 的 AWS SDK 774
存取環境變數 774
Python Lambda 函數的程式碼最佳實務 774
部署 .zip 封存檔 777
Python 中的執行期相依項 777
建立不含相依項的 .zip 部署套件 778
建立含相依項的 .zip 部署套件 778
相依項搜尋路徑和含執行期程式庫 781
使用 __pycache__ 資料夾 782
建立含原生程式庫的 .zip 部署套件 782
使用 .zip 檔案建立及更新 Python Lambda 函數 783
部署容器映像 790
AWS Python 的基礎映像 790
使用 AWS 基礎映像 792
使用非AWS 基礎映像 798
層 808
封裝層內容 808

在 Lambda 中建立層 694
將層新增至函式 813
應用程式範例 814
Context 815
日誌 817
列印至日誌 817
使用記錄程式庫 818
搭配 Python 使用 Lambda 進階日誌控制項 819
在 Lambda 主控台檢視日誌 824
在 CloudWatch 主控台中檢視日誌 824
使用 檢視日誌 AWS CLI 824
刪除日誌 827
工具與程式庫 828
使用 Powertools for AWS Lambda (Python) 和 AWS SAM 進行結構化記錄 828
使用 Powertools for AWS Lambda (Python) 和 AWS CDK 進行結構化記錄 832
測試 839
測試無伺服器應用程式 840
追蹤 842
使用 Powertools for AWS Lambda (Python) 和 AWS SAM 進行追蹤 843
使用 Powertools for AWS Lambda (Python) 和 AWS CDK 進行追蹤 845
使用 ADOT 來檢測您的 Python 函數 850
使用 X-Ray SDK 來檢測 Python 功能 850
透過 Lambda 主控台來啟用追蹤 851
透過 Lambda API 啟用追蹤 851
使用 啟用追蹤 CloudFormation 852
解讀 X-Ray 追蹤 852
將執行時間相依項存放存在層中 (X-Ray SDK) 855
使用 Ruby 建置 856
包含執行時期的 SDK 版本 857
啟用 Yet Another Ruby JIT (YJIT) 858
處理常式 859
Ruby 處理常式基本概念 859
Ruby Lambda 函數的程式碼最佳實務 860
部署 .zip 封存檔 862
Ruby 中的相依項 862
建立不含相依項的 .zip 部署套件 863

建立含相依項的 .zip 部署套件 863
為相依項建立 Ruby 層 864
建立含原生程式庫的 .zip 部署套件 865
使用 .zip 檔案建立及更新 Ruby Lambda 函數 866
部署容器映像 872
AWS Ruby 的基礎映像 872
使用 AWS 基礎映像 873
使用非AWS 基礎映像 879
層 889
封裝層內容 889
在 Lambda 中建立層 694
在函式中使用來自層的 gem 896
將層新增至函式 896
應用程式範例 897
Context 898
日誌 899
建立傳回日誌的函數 899
在 Lambda 主控台檢視日誌 900
在 CloudWatch 主控台中檢視 記錄 900
使用 AWS Command Line Interface (AWS CLI) 檢視日誌 901
刪除日誌 904
使用 Ruby 記錄程式庫 904
追蹤 906
透過 Lambda API 啟用主動追蹤 911
使用 啟用主動追蹤 CloudFormation 911
將執行時間相依性儲存在圖層中 912
使用 Java 建置 914
處理常式 918
設定 Java 處理常式專案 918
Java Lambda 函式程式碼範例 919
Java 處理常式的有效類別定義 924
處理常式命名慣例 925
定義和存取輸入事件物件 925
存取和使用 Lambda 內容物件 926
在處理常式中使用適用於 Java v2 的 AWS SDK 927
存取環境變數 928

使用全域狀態 929
Java Lambda 函數的程式碼最佳實務 929
部署 .zip 封存檔 931
先決條件 931
工具與程式庫 931
使用 Gradle 建立部署套件 933
使用層來管理相依項 934
使用 Maven 建立部署套件 934
使用 Lambda 主控台上傳部署套件 936
使用 上傳部署套件 AWS CLI 938
使用 上傳部署套件 AWS SAM 939
部署容器映像 941
AWS Java 的基礎映像 941
使用 AWS 基礎映像 943
使用非AWS 基礎映像 951
層 962
封裝層內容 962
在 Lambda 中建立層 694
將層新增至函式 965
自訂序列化 967
何時使用自訂序列化 967
實作自訂序列化 968
測試自訂序列化 968
自訂啟動行為 970
了解 JAVA_TOOL_OPTIONS 環境變數 970
Log4Shell 的 Log4j 修補程式 Log4Shell 972
Ahead-of-Time(AOT) 和 CDS 快取 972
Context 973
範例應用程式中的內容 975
日誌 976
建立傳回日誌的函數 976
搭配 Java 使用 Lambda 進階日誌控制項 978
使用 Log4j2 和 SLF4J 實作進階日誌記錄 980
工具與程式庫 984
使用 Powertools for AWS Lambda (Java) 和 AWS SAM 進行結構化記錄 984
在 Lambda 主控台檢視日誌 988

在 CloudWatch 主控台中檢視 記錄 988
使用 AWS Command Line Interface (AWS CLI) 檢視日誌 989
刪除日誌 992
日誌記錄程式碼範例 992
追蹤 993
使用 Powertools for AWS Lambda (Java) 和 AWS SAM 進行追蹤 994
使用 Powertools for AWS Lambda (Java) 和 AWS CDK 進行追蹤 996
使用 ADOT 來檢測您的 Java 函數 1007
使用 X-Ray SDK 來檢測 Java 功能 1008
透過 Lambda 主控台來啟用追蹤 1008
透過 Lambda API 啟用追蹤 1009
使用 啟用追蹤 CloudFormation 1009
解讀 X-Ray 追蹤 1010
將執行時間相依項存放存在層中 (X-Ray SDK) 1012
樣本應用程式中的 X-Ray 追蹤 (X-Ray SDK) 1013
範例應用程式 1015
使用 Go 建置 1016
Go 執行期支援 1016
工具與程式庫 1017
處理常式 1018
設定 Go 處理常式專案 1018
範例 Go Lambda 函數程式碼 1019
處理常式命名慣例 1021
定義和存取輸入事件物件 1021
存取和使用 Lambda 內容物件 1022
Go 處理常式的有效處理常式簽章 1023
在處理常式中使用 適用於 Go 的 AWS SDK v2 1024
存取環境變數 1025
使用全域狀態 1026
Go Lambda 函數的程式碼最佳實務 1026
Context 1028
內容物件中支援的變數、方法和屬性 1028
存取叫用內容資訊 1029
在 AWS SDK 用戶端初始化和呼叫中使用內容 1030
部署 .zip 封存檔 1032
在 macOS 和 Linux 上建立 .zip 檔案 1032

在 Windows 上建立 .zip 檔案 1034
使用 .zip 檔案建立及更新 Go Lambda 函數 1036
部署容器映像 1043
AWS 部署 Go 函數的基礎映像 1043
Go 執行期介面用戶端 1044
使用僅限 AWS 作業系統的基礎映像 1044
使用非AWS 基礎映像 1050
層 1059
日誌 1060
建立傳回日誌的函數 1060
在 Lambda 主控台檢視日誌 1062
在 CloudWatch 主控台中檢視 記錄 1062
使用 AWS Command Line Interface (AWS CLI) 檢視日誌 1062
刪除日誌 1065
追蹤 1066
使用 ADOT 來檢測您的 Go 函數 1066
使用 X-Ray SDK 來檢測 Go 函數 1067
透過 Lambda 主控台來啟用追蹤 1067
透過 Lambda API 啟用追蹤 1068
使用 啟用追蹤 CloudFormation 1068
解讀 X-Ray 追蹤 1069
使用 C# 建置 1072
開發環境 1072
安裝 .NET 專案範本 1072
安裝和更新 CLI 工具 1072
處理常式 1074
設定 C# 處理常式專案 1074
C# Lambda 函式程式碼範例 1075
類別庫處理常式 1079
可執行組件處理常式 1079
C# 函式的有效處理常式簽章 1080
處理常式命名慣例 1081
C# Lambda 函式中的序列化 1081
以檔案為基礎的函數 1084
存取和使用 Lambda 內容物件 1084
在處理常式中使用 適用於 .NET 的 SDK v3 1085

存取環境變數 1086
使用全域狀態 1086
使用 Lambda Annotations 架構簡化函數程式碼 1086
C# Lambda 函數的程式碼最佳實務 1088
部署套件 1090
.NET Lambda Global CLI 1090
AWS SAM 1096
AWS CDK 1099
ASP.NET 1103
層 1108
部署容器映像 1109
AWS .NET 的基礎映像 1109
使用 AWS 基礎映像 1110
使用非AWS 基礎映像 1112
原生 AOT 編譯 1117
Lambda 執行時間 1117
必要條件 1117
開始使用 1118
序列化 1121
裁剪 1121
疑難排解 1122
Context 1123
日誌 1125
建立傳回日誌的函數 1125
搭配 .NET 使用 Lambda 進階日誌控制項 1126
工具與程式庫 1132
使用 Powertools for AWS Lambda (.NET) 和 AWS SAM 用於結構化記錄 1133
在 Lambda 主控台檢視日誌 1136
在 CloudWatch 主控台中檢視 記錄 1136
使用 AWS Command Line Interface (AWS CLI) 檢視日誌 1136
刪除日誌 1139
追蹤 1140
使用 Powertools for AWS Lambda (.NET) 和 AWS SAM 用於追蹤 1141
使用 X-Ray SDK 來檢測 .NET 函數 1143
透過 Lambda 主控台來啟用追蹤 1145
透過 Lambda API 啟用追蹤 1145

使用 啟用追蹤 CloudFormation 1145
解讀 X-Ray 追蹤 1146
測試 1150
測試無伺服器應用程式 1151
使用 PowerShell 建置 1154
開發環境 1155
部署套件 1156
建立 Lambda 函數 1156
處理常式 1158
傳回資料 1158
Context 1160
日誌 1161
建立傳回日誌的函數 1161
在 Lambda 主控台檢視日誌 1163
在 CloudWatch 主控台中檢視 記錄 1163
使用 AWS Command Line Interface (AWS CLI) 檢視日誌 1163
刪除日誌 1166
使用 Rust 建置 1167
處理常式 1169
設定 Rust 處理常式專案 1169
Lambda 函式程式碼範例 1170
Rust 處理常式的有效類別定義 1172
處理常式命名慣例 1174
定義和存取輸入事件物件 1174
存取和使用 Lambda 內容物件 1175
在處理常式 適用於 Rust 的 AWS SDK 中使用 1176
存取環境變數 1176
使用共用狀態 1177
Rust Lambda 函數的程式碼最佳實務 1177
Context 1179
存取叫用內容資訊 1179
HTTP 事件 1180
部署 .zip 封存檔 1182
先決條件 1182
建置函數 1182
部署函數 1183

叫用函數 1185
層 1186
日誌 1187
建立編寫日誌的函數 1187
實作帶有追蹤套件的進階日誌記錄 1187
最佳實務 1190
函數程式碼 1190
函數組態 1191
函數可擴展性 1192
指標與警示 1193
使用串流 1193
安全最佳實務 1194
測試無伺服器函數 1196
目標業務成果 1197
測試項目 1197
如何測試無伺服器 1198
測試技術 1198
在雲端進行測試 1199
透過模擬物件進行測試 1201
使用 AWS SAM CLI 進行本機測試 1202
透過模擬進行測試 1203
最佳實務 1203
排定雲端測試的優先順序 1204
構建程式碼以實現可測試性 1204
加快回饋迴圈的開發速度 1204
全力進行整合測試 1204
建立獨立的測試環境 1205
將模擬物件用於隔離的商業邏輯 1206
請謹慎使用模擬器 1206
在本機進行測試的難題 1206
範例：Lambda 函數建立 S3 儲存貯體 1207
範例：Lambda 函數處理來自 Amazon SQS 佇列的訊息 1207
常見問答集 1208
後續步驟和資源 1208
Lambda SnapStart 1210
使用案例 1211

支援的功能和限制 1211
支援的區域 1212
相容性考量 1212
定價 1212
啟動 SnapStart 1214
啟動 SnapStart (主控台) 1214
啟動 SnapStart (AWS CLI) 1214
啟動 SnapStart (API) 1217
函數狀態 1217
更新快照 1218
使用 SnapStart AWS SDKs 1218
搭配 CloudFormation AWS SAM、 和 使用 SnapStart AWS CDK 1218
刪除快照 1219
處理唯一性 1220
避免儲存狀態 1220
使用 CSPRNGs 1222
掃描工具 (Java) 1224
執行階段掛鉤 1225
Java 1225
Python 1228
.NET 1230
監控 1233
CloudWatch Logs 1233
AWS X-Ray 1234
遙測 API 1234
API Gateway 和函數 URL 指標 1235
安全模型 1236
最佳實務 1237
效能調校 1237
網路最佳實務 1240
疑難排解 1242
SnapStartNotReadyException 1242
SnapStartTimeoutException 1242
500 內部服務錯誤 1243
401 (未經授權) 1243
UnknownHostException (Java) 1243

快照建立失敗 1244
快照建立延遲 1244
租戶隔離 1245
何時使用租戶隔離模式 1245
支援的功能和限制 1246
支援的 AWS 區域 1246
考量事項 1246
定價 1246
隔離模式 1247
啟用租戶隔離 1248
主控台 1248
AWS CLI 1248
API 1249
CloudFormation 1249
使用租用戶隔離叫用 函數 1251
AWS CLI 1251
API 1251
API Gateway 1252
SDK 1253
在程式碼中存取 1255
存取 1255
用量模式 1256
監控 1257
監控 1259
了解租戶隔離模式的記錄 1259
疑難排解 1261
InvalidParameterValueException 1261
TooManyRequestsException 1261
整合其他服務 1262
建立觸發條件 1262
服務清單 1263
Apache Kafka 1265
MSK 1266
自我管理的 Apache Kafka 1310
事件輪詢器擴展 1327
輪詢與串流位置 1333

取用者群組 ID 1334
事件篩選 1335
結構描述登錄檔與事件來源 1340
低延遲 Apache Kafka 1367
重試組態 1369
保留失敗的調用 1374
Kafka 失敗時目的地 1381
Kafka ESM 記錄 1386
疑難排解 1391
API Gateway 1397
選擇 API 類型 1397
將端點新增至您的 Lambda 函數 1399
Proxy 整合 1399
事件格式 1400
回應格式 1400
許可 1401
範例應用程式 1403
Powertools for AWS Lambda 的事件處理常式 1403
教學課程 1404
錯誤 1420
API Gateway 與函數 URL 比較 1421
Infrastructure Composer 1425
將 Lambda 函數匯出至 Infrastructure Composer 1425
其他資源 1427
CloudFormation 1428
Amazon DocumentDB 1431
Amazon DocumentDB 事件範例 1432
先決條件和許可 1433
設定網路安全 1434
建立 Amazon DocumentDB 事件來源映射 (主控台) 1437
建立 Amazon DocumentDB 事件來源映射 (SDK 或 CLI) 1438
輪詢和串流開始位置 1441
監控 Amazon DocumentDB 事件來源 1441
教學課程 1441
DynamoDB 1467
輪詢和批次處理串流 1467

輪詢和串流開始位置 1468
同時讀取 1469
範例事件 1469
建立映射 1470
批次項目失敗 1473
錯誤處理 1489
有狀態處理 1495
Parameters 1500
事件篩選 1502
教學課程 1510
EC2 1526
將許可授予 EventBridge (CloudWatch Events) 1526
Elastic Load Balancing (Application Load Balancer) 1528
Powertools for AWS Lambda 的事件處理常式 1529
使用 EventBridge 排程器調用 1531
設定執行角色 1531
建立排程 1531
相關資源 1535
IoT 1536
Kinesis Data Streams 1538
輪詢和批次處理串流 1538
範例事件 1540
建立映射 1541
批次項目失敗 1545
錯誤處理 1564
有狀態處理 1569
參數 1572
事件篩選 1574
教學課程 1578
Kubernetes 1595
Kubernetes 專用 AWS 控制器 (ACK) 1595
Crossplane 1595
MQ 1597
了解 Amazon MQ 的 Lambda 取用者群組 1599
設定事件來源 1602
參數 1608

事件篩選 1609
疑難排解 1615
RDS 1617
設定函數以使用 RDS 資源 1617
連線至 Lambda 函數中的 Amazon RDS 資料庫 1623
處理來自 Amazon RDS 的事件通知 1642
完成 Lambda 和 Amazon RDS 教學課程 1643
Amazon RDS 與 DynamoDB 比較 1643
S3 1647
教學課程：使用 S3 觸發條件 1648
教學課程：使用 Amazon S3 觸發條件建立縮圖 1673
Secrets Manager 1700
選擇合適方案 1700
何時使用 Secrets Manager 1700
使用 AWS 參數和秘密 Lambda 延伸模組 1701
從 Powertools for 使用參數公用程式 AWS Lambda 1711
SQS 1717
了解 Amazon SQS 事件來源映射的輪詢和批次處理行為 1718
搭配 Amazon SQS 事件來源映射使用佈建模式 1718
設定 Amazon SQS 事件來源映射的佈建模式 1719
標準佇列訊息事件範例 1721
FIFO 佇列訊息事件範例 1722
建立映射 1723
擴展行為 1726
錯誤處理 1729
Parameters 1743
事件篩選 1745
教學課程 1749
SQS 跨帳戶教學課程 1767
步驟函數 1774
何時使用 Step Functions 1774
不適用 Step Functions 的時機 1780
S3 批次 1782
從 Amazon S3 批次操作叫用 Lambda 函數 1783
SNS 1784
Powertools for AWS Lambda 的冪等性公用程式 1784

使用主控台為 Lambda 函數新增 Amazon SNS 主題觸發條件 1784
手動新增 Lambda 函數的 Amazon SNS 主題觸發條件 1785
範例 SNS 事件形狀 1786
教學課程 1787
Lambda 許可 1806
執行角色 (函數存取其他資源的許可) 1808
在 IAM 主控台中建立執行角色 1808
使用 建立和管理角色 AWS CLI 1809
為 Lambda 執行角色授予最低權限存取權 1810
更新執行角色 1810
AWS 受管政策 1812
來源函數 ARN 1814
存取許可 (其他實體存取函數的許可) 1819
身分型政策 1819
資源型政策 1825
屬性型存取控制 1833
資源與條件 1840
安全、控管與合規 1847
資料保護 1847
傳輸中加密 1848
靜態加密 1849
使用服務連結角色 1854
Lambda 的服務連結角色許可 1854
為 Lambda 建立服務連結角色 1854
編輯 Lambda 的服務連結角色 1855
刪除 Lambda 的服務連結角色 1855
Lambda 服務連結角色支援的區域 1855
身分和存取權管理 1857
目標對象 1857
使用身分驗證 1858
使用政策管理存取權 1859
AWS Lambda 如何使用 IAM 1860
身分型政策範例 1865
AWS 受管政策 1867
疑難排解 1874
控管 1876

使用 Guard 的主動式控制項 1878
使用 的主動控制 AWS Config 1882
使用 偵測控制項 AWS Config 1889
程式碼簽署 1893
程式碼掃描 1895
可觀測性 1899
法規遵循驗證 1906
恢復能力 1906
基礎架構安全 1907
使用公有端點保護工作負載 1907
身分驗證和授權 1907
保護 API 端點 1908
程式碼簽署 1909
簽署驗證 1909
建立組態 1910
許可 1911
程式碼簽署組態標籤 1913
監控與偵錯函式 1917
定價 1917
函數指標 1918
檢視函數指標 1918
指標類型 1919
函數日誌 1928
選擇用於傳送日誌的服務目的地 1928
設定日誌目的地 1929
設定 Lambda 函數的進階日誌記錄控制項 1929
日誌格式 1929
日誌層級篩選 1935
使用 CloudWatch Logs 記錄日誌 1940
使用 Firehose 記錄日誌 1957
使用 Amazon S3 記錄日誌 1959
CloudTrail 日誌 1964
CloudTrail 中的 Lambda 資料事件 1965
CloudTrail 中的 Lambda 管理事件 1966
使用 CloudTrail 疑難排解已停用的 Lambda 事件來源 1968
Lambda 事件範例 1969

AWS X-Ray 1972
了解 X-Ray 追蹤 1973
Lambda 中的預設追蹤行為 1977
執行角色許可 1977
透過 Lambda API 啟用 Active 追蹤 1978
使用 啟用Active追蹤 CloudFormation 1978
函式深入解析 1980
運作方式 1980
定價 1980
支援的執行期 1981
在主控台中啟用 Lambda Insights 1981
以程式設計方式啟用 Lambda Insights 1981
使用 Lambda Insights 儀表板 1981
偵測函式異常 1983
故障排除函式 1985
後續步驟？ 1986
檢視應用程式指標 1987
Application Signals 1989
Application Signals 與 Lambda 的整合方式 1989
定價 1990
支援的執行期 1990
在 Lambda 主控台中啟用 Application Signals 1990
使用 Application Signals 儀表板 1991
使用 VS Code 偵錯 1993
支援的執行期 1993
安全性與遠端偵錯 1993
必要條件 1994
遠端偵錯 Lambda 函式 1994
停用遠端偵錯 1995
其他資訊 1995
Lambda 層 1997
如何使用層 1999
層和層的版本 1999
封裝層 2000
每個 Lambda 執行時間的層路徑 2000
建立和刪除層 2003

建立圖層 2003
刪除圖層版本 2004
新增層 2005
尋找圖層資訊 2006
具有 CloudFormation 的層 2009
具有 AWS SAM 的層 2010
Lambda 延伸 2011
執行環境 2011
對效能和資源的影響 2012
許可 2013
設定延伸項目 2014
設定延伸項目 (.zip 封存檔) 2014
在容器映像中使用延伸項目 2014
後續步驟 2015
延伸合作夥伴 2016
AWS 受管延伸 2016
Extensions API 2018
Lambda 執行環境生命週期 2019
Extensions API 參考 2027
遙測 API 2033
使用遙測 API 建立延伸項目 2034
註冊延伸項目 2036
建立遙測接聽程式 2036
指定目的地通訊協定 2037
設定記憶體使用量和緩衝 2038
將訂閱請求傳送至遙測 API 2040
輸入遙測 API 訊息 2041
API 參考 2044
Event 結構描述參考 2048
將事件轉換為 OTel 跨度 2069
Logs API 2075
疑難排解 2087
組態 2087
記憶體組態 2088
CPU 受限的組態 2088
逾時 2088

調用之間的記憶體外洩 2089
非同步結果傳回至稍後的調用 2091
部署 2095
一般：許可遭拒/無法載入此類檔案 2095
一般：呼叫 UpdateFunctionCode 時發生錯誤 2096
Amazon S3：錯誤代碼 PermanentRedirect。 2096
一般：找不到、無法載入、無法匯入、找不到類別、沒有此類檔案或目錄 2097
一般：未定義的方法處理常式 2097
一般：超過 Lambda 程式碼儲存限制 2098
Lambda：分層轉換失敗 2098
Lambda：InvalidParameterValueException 或 RequestEntityTooLargeException 2099
Lambda：InvalidParameterValueException 2099
Lambda：並行和記憶體配額 2100
Lambda：佈建並行的別名組態無效 2100
調用 2101
Lambda：函數在初始化階段逾時 (Sandbox.Timedout) 2102
IAM：lambda:InvokeFunction 未獲授權 2102
Lambda：找不到有效的引導程序 (Runtime.InvalidEntrypoint) 2103
Lambda：無法執行操作 ResourceConflictException 2103
Lambda：函數卡在待定狀態 2103
Lambda：一個函數正在使用所有並行 2103
一般：無法使用其他帳戶或服務調用函數 2104
一般：函數調用正在循環 2104
Lambda：具有佈建並行的別名路由 2104
Lambda：使用佈建並行的冷啟動 2104
Lambda：新版本的冷啟動 2105
Lambda：未預期的 Node.js 在執行時間結束 (Runtime.NodejsExit) 2105
EFS：函數無法掛載 EFS 檔案系統 2106
EFS：函數無法連線到 EFS 檔案系統 2106
EFS：因為逾時，函數無法掛載 EFS 檔案系統 2106
Lambda：Lambda 偵測到耗時太久的 IO 程序 2106
容器：CodeArtifactUserException 錯誤 2107
容器：InvalidEntrypoint 錯誤 2107
執行 2107
Lambda：使用 Visual Studio Code 進行遠端偵錯 2108
Lambda：執行時間太長 2108

Lambda：非預期的事件承載 2109
Lambda：出乎意料的大型承載大小 2109
Lambda：JSON 編碼與解碼錯誤 2110
Lambda：日誌或追蹤沒有出現 2110
Lambda：並非所有函數的日誌都會顯示 2111
Lambda：該函數在執行完成之前傳回 2112
Lambda：執行非預期的函式版本或別名 2112
Lambda：偵測無限迴圈 2112
一般：下游服務無法使用 2113
AWS SDK：版本和更新 2114
Python：程式庫的載入不正確 2115
Java：從 Java 11 更新至 Java 17 後，函數需要更長的時間來處理事件 2115
Kafka：處理和重試組態問題時發生錯誤 2115
事件來源映射 2116
識別和管理限流 2116
處理函數中的錯誤 2117
識別和處理背壓 2120
聯網 2121
VPC：函數會失去網際網路存取或逾時 2121
VPC：TCP 或 UDP 連線間歇性失敗 2122
VPC：函數需要存取 而不 AWS 服務 使用網際網路 2122
VPC：已達到彈性網絡介面限制 2122
EC2：具有「lambda」類型的彈性網路介面 2122
DNS：遇到 UNKNOWNHOSTEXCEPTION，無法連線至主機 2123
範例應用程式 2124
使用 AWS SDKs 2127
程式碼範例 2129
基本概念 2131
Hello Lambda 2132
了解基本概念 2142
動作 2278
案例 2412
使用 Lambda 函數自動確認已知使用者 2412
使用 Lambda 函數自動遷移已知使用者 2453
建立 REST API 以追蹤 COVID-19 資料 2476
建立出借圖書館 REST API 2477

建立傳訊應用程式 2478
建立無伺服器應用程式來管理相片 2478
建立 websocket 聊天應用程式 2482
建立應用程式以分析客戶意見回饋 2483
從瀏覽器調用 Lambda 函式 2489
使用 S3 Object Lambda 轉換資料 2490
使用 API Gateway 來調用 Lambda 函式 2490
使用 Step Functions 調用 Lambda 函式 2492
使用排程事件來調用 Lambda 函式 2493
使用 Neptune API 查詢圖形資料 2494
進行 Amazon Cognito 使用者身分驗證後，可使用 Lambda 函數撰寫自訂活動資料 2495
無伺服器範例 2518
連線至 Lambda 函數中的 Amazon RDS 資料庫 2518
使用 Kinesis 觸發條件調用 Lambda 函數 2537
使用 DynamoDB 觸發條件調用 Lambda 函式 2548
使用 Amazon DocumentDB 觸發條件調用 Lambda 函數 2557
使用 Amazon MSK 觸發條件調用 Lambda 函數 2569
使用 Amazon S3 觸發條件調用 Lambda 函數 2579
使用 Amazon SNS 觸發條件調用 Lambda 函數 2590
使用 Amazon SQS 觸發條件調用 Lambda 函數 2600
使用 Kinesis 觸發條件報告 Lambda 函數的批次項目失敗 2609
使用 DynamoDB 觸發條件報告 Lambda 函式的批次項目失敗 2622
使用 Amazon SQS 觸發條件報告 Lambda 函數的批次項目失敗 2633
AWS 社群貢獻 2643
建置和測試無伺服器應用程式 2643
Lambda 配額 2646
運算與儲存 2647
函數組態、部署和執行 2648
Lambda API 請求 2651
其他服務 2651
文件歷史紀錄 2653
舊版更新 2672
..... mmdclxxviii

什麼是 AWS Lambda？

> **Tip**
>
> 參加 Serverless 專家的免費實作研討會，了解如何使用最佳實務建置 Serverless 應用程式。按一下此處註冊。

AWS Lambda 是一種運算服務，無需管理伺服器即可執行程式碼。您的程式碼會自動執行，並依據負載向上擴展或縮減規模，且採用按使用量付費的計價方式。若要開始使用，請參閱建立第一個函數。

您可以將 Lambda 用於：

- 檔案處理：在檔案上傳至 Amazon Simple Storage Service 時自動處理檔案。如需詳細資訊，請參閱檔案處理範例。
- 長時間執行的工作流程：使用耐用的 Lambda 函數來建置狀態良好的多步驟工作流程，最多可執行一年。非常適合訂單處理、核准工作流程、human-in-the-loop，以及需要記住進度的複雜資料管道。
- 資料庫操作與整合範例：回應資料庫變更並自動執行資料工作流程。如需詳細資訊，請參閱資料庫範例。
- 排程與定期任務：使用 EventBridge 定期執行自動化操作。如需詳細資訊，請參閱排程任務範例。
- 串流處理：處理即時資料串流，用於分析與監控用途。如需詳細資訊，請參閱 Kinesis Data Streams。
- Web 應用程式：建置可擴展的 Web 應用程式，能依據需求自動調整規模。
- 行動後端：建立安全的 API 後端，供行動裝置與 Web 應用程式使用。
- IoT 後端：處理 Web、行裝置動、IoT 與第三方 API 請求。如需詳細資訊，請參閱 IoT。

如需定價資訊，請參閱 AWS Lambda 定價。

Lambda 的運作方式

使用 Lambda 時，您只需負責程式碼的相關操作。Lambda 在高可用性運算基礎結構上執行您的程式碼，並管理所有運算資源，包括伺服器與作業系統維護、容量佈建、自動擴展以及日誌記錄。

由於 Lambda 是事件驅動型無伺服器運算服務，其採用的程式設計典範有別於傳統 Web 應用程式。下列模型說明了 Lambda 的運作方式：

1. 您可以在用於建立 Lambda 應用程式的基本建置區塊 – Lambda 函式中編寫並組織程式碼。
2. 您可以透過 Lambda 許可控制安全性和存取權，使用執行角色來管理函數可以與哪些 AWS 服務互動，以及哪些資源政策可以與程式碼互動。
3. 事件來源 AWS 和服務會觸發您的 Lambda 函數，以 JSON 格式傳遞事件資料，您的函數會處理這些資料（包括事件來源映射）。
4. Lambda 會透過語言專屬執行時期 (如 Node.js 與 Python) 在執行環境中執行程式碼，這些執行環境已封裝您的執行時期、層級與延伸模組。

> **Tip**
>
> 若要了解如何建置無伺服器解決方案，請參閱無伺服器開發人員指南。

主要功能

設定、控制與部署安全應用程式：

- 環境變數無需部署新的程式碼即可修改應用程式行為。
- 版本可以安全地測試新功能，同時維持穩定的生產環境。
- Lambda 層透過跨多個函式共用通用元件，最佳化程式碼的重複使用性與可維護性。
- 程式碼簽署透過確保僅有經核准的程式碼能進入生產環境，強制執行安全合規要求。

可靠擴展與執行：

- 並行與擴展控制可在流量激增期間精準管理應用程式的回應速度與資源使用率。
- Lambda SnapStart 可大幅縮短冷啟動時間。Lambda SnapStart 可提供低至一秒的啟動效能，通常不會變更函數程式碼。
- 回應串流透過逐步傳送大型承載資料來實現即時處理，從而最佳化函式效能。
- 容器映像運用容器工作流程，將具有複雜相依性的函式進行封裝。

無縫連線與整合：

- VPC 網路可防護敏感資源與內部服務的安全。
- 檔案系統整合能跨函式調用共用持續性資料，並管理具狀態作業。

- 函數 URL 無需額外服務即可建立公開面向的 API 與端點。
- Lambda 延伸可透過監控、安全與營運工具增強函式效能。

相關資訊

- 如需 Lambda 運作方式的詳細資訊，請參閱 Lambda 的運作方式。
- 若要開始使用 Lambda，請參閱建立第一個 Lambda 函數。
- 如需範例應用程式清單，請參閱透過範例應用程式與模式快速入門。

Lambda 的運作方式

Lambda 函式是用來建置 Lambda 應用程式的基本建置區塊。若要編寫函式，必須了解組成 Lambda 程式設計模型的核心概念和元件。本節將引導您了解開始使用 Lambda 建置無伺服器應用程式所需的基本元素。

- Lambda 函式與函式處理常式 - Lambda 函數是一小段程式碼，用於回應事件。函數可以是標準（最多 15 分鐘）或耐用（最多一年）。函數是您用來建置應用程式的基本建置區塊。處理常式是 Lambda 函式程式碼處理事件物件的進入點。
- Lambda 執行環境與執行時期 – Lambda 執行環境會管理執行函式所需的資源。對於耐用的函數，執行環境包含自動狀態管理和檢查點功能。執行時間是您函數執行的特定語言環境。
- 事件與觸發程序 - 其他 AWS 服務 可以調用函數以回應特定事件。對於耐用的函數，事件也可以觸發暫停工作流程的恢復。
- Lambda 許可與角色 - 控制誰可以存取您的函式，以及 AWS 服務 您的函式可以與哪些其他互動。耐用的函數需要額外的許可，才能進行狀態管理和延伸執行。

> **Tip**
>
> 若要更全面地了解無伺服器開發，請參閱 AWS Serverless Developer Guide 中的 Understanding the difference between traditional and serverless development。

Lambda 函式與函式處理常式

在 Lambda 中，函式是您用來建立應用程式的基本建置區塊。Lambda 函式是為回應事件而執行的一段程式碼，例如使用者按一下網站上的按鈕，或檔案上傳至 Amazon Simple Storage Service (Amazon

S3) 儲存貯體。透過耐用的 函數，您的程式碼可以在步驟之間暫停執行，自動維護狀態，使其非常適合長時間執行的工作流程，例如訂單處理或內容管制。您可以將函式視為一種具有下列特性的獨立程式。

Lambda 函式處理常式是函式程式碼中處理事件的方法。當函式因回應事件而執行時，Lambda 會執行函式處理常式。導致函式執行的事件相關資料，會直接傳遞至處理常式。雖然 Lambda 函式中的程式碼可以包含多個方法或函式，但 Lambda 函式只能有一個處理常式。

若要建立 Lambda 函式，需將函式程式碼及其相依項共同封裝為部署套件。Lambda 支援兩種類型的部署套件：.zip 壓縮檔與容器映像。

- 一個函式有一個特定的任務或用途
- 僅在需要回應特定事件時執行
- 會在任務完成後自動停止執行

Lambda 執行環境與執行時期

Lambda 函式會在 Lambda 為您管理的安全隔離執行環境中執行。對於耐用的函數，執行環境包含用於狀態管理和工作流程協調的其他元件。執行環境會管理執行函數所需的程序和資源。當函式首次被調用時，Lambda 會為該函式建立一個新的執行環境以在其中執行。函式執行完成後，Lambda 不會立即停止該執行環境；如果函式被再次調用，Lambda 可以重複使用現有的執行環境。

Lambda 執行環境中還包含執行時期，這是一種與特定程式語言相關的環境，負責在 Lambda 與您的函式之間傳遞事件資訊與回應。Lambda 提供許多適用於最熱門程式設計語言的受管執行時期，但您也可以自行建立執行時期。

對於受管執行時期，Lambda 會自動對使用該執行時期的函式套用安全性更新與修補程式。

事件與觸發程序

您也可以透過 Lambda 主控台、AWS CLI 或任一 AWS 軟體開發套件 (SDK) 直接調用 Lambda 函式。在生產應用程式中，另一個 會調用您的函數 AWS 服務 以回應特定事件，這是較常見的做法。例如，您可能希望每當 Amazon DynamoDB 資料表新增項目時，就執行某個函式。

若要讓函式回應事件，您可以設定觸發程序。觸發程序會將函式連線至事件來源，且函式可以有多個觸發程序。當事件發生時，Lambda 會以 JSON 文件的形式接收事件資料，並將其轉換為程式碼能夠處理的物件。您可以為事件定義下列 JSON 格式，Lambda 執行時期會將此 JSON 文件轉換為物件，然後再將其傳遞至函式的處理常式。

Example自訂 Lambda 事件

```
{
  "Location": "SEA",
  "WeatherData":{
    "TemperaturesF":{
      "MinTempF": 22,
      "MaxTempF": 78
    },
    "PressuresHPa":{
      "MinPressureHPa": 1015,
      "MaxPressureHPa": 1027
    }
  }
}
```

Amazon Kinesis 或 Amazon SQS 等串流和佇列服務會使用事件來源映射，而非標準觸發。事件來源映射會輪詢來源以取得新資料、批次記錄，然後使用批次事件調用函式。如需詳細資訊，請參閱事件來源映射與直接觸發條件有何不同。

若要了解觸發程序的運作方式，建議先完成使用 Amazon S3 觸發程序教學課程。若需觸發程序使用總覽，或透過 Lambda 主控台建立觸發程序的操作步驟，請參閱整合其他服務。

Lambda 許可與角色

對於 Lambda，您需要設定兩種主要的許可類型：

- 您的函數存取其他 所需的許可 AWS 服務
- 其他使用者和 存取函數 AWS 服務 所需的許可

下列各節介紹了這兩種許可類型，並討論了套用最低權限許可的最佳實務。

函數存取其他 AWS 資源的許可

Lambda 函數通常需要存取其他 AWS 資源並對其執行動作。例如，函式可能會讀取 DynamoDB 資料表中項目、將物件儲存在 S3 儲存貯體中，或向 Amazon SQS 佇列寫入資料。若要為函式提供執行這些動作所需的許可，您可以使用執行角色。

Lambda 執行角色是一種特殊的 AWS Identity and Access Management (IAM) 角色，您在帳戶中建立的身分，具有政策中定義與其相關聯的特定許可。

每個 Lambda 函式都必須擁有執行角色，而且單一角色可供多個函式共同使用。當函式被調用時，Lambda 會擔任函式的執行角色，取得執行該角色政策中定義之動作的許可。

若您透過 Lambda 主控台建立函式，Lambda 會自動為您的函式建立執行角色。該角色的政策會授予函式基本許可，可將日誌輸出寫入 Amazon CloudWatch Logs。若要授予函數在其他 AWS 資源上執行動作的許可，您需要編輯角色以新增額外的許可。新增許可的最簡單方法是使用 AWS 受管政策。受管政策是由 建立和管理 AWS ，並提供許多常見使用案例的許可。例如，若函式對 DynamoDB 資料表執行 CRUD 操作，您可以將 AmazonDynamoDBFullAccess 政策新增至角色。

其他使用者與資源存取函式所需的許可

若要授予其他 AWS 服務 許可來存取您的 Lambda 函數，您可以使用以資源為基礎的政策。在 IAM 中，資源型政策會連接至資源 (本案例中即您的 Lambda 函式)，並定義可存取該資源的對象，以及允許對象執行哪些動作。

若要讓另一個 AWS 服務 透過觸發叫用函數，函數的資源型政策必須授予該服務使用該`lambda:InvokeFunction`動作的許可。若您透過主控台建立觸發程序，Lambda 會自動為您新增此許可。

若要授予其他 AWS 使用者存取函數的許可，您可以在函數的資源型政策中定義此項目，方式與另一個 AWS 服務 或 資源完全相同。您也可以使用與使用者相關聯的身分型政策。

Lambda 許可的最佳實務

使用 IAM 政策設定許可時，安全最佳實務是僅授予執行任務所需的許可。這稱為最低權限原則。若要開始授予函數的許可，您可以選擇使用 AWS 受管政策。受管政策是授予執行任務所需許可最快速且最簡單的方式，但其中可能包含您不需要的其他許可。從早期開發到進入測試及生產階段的過程中，建議透過間建立自訂客戶管理政策，將許可精簡至僅保留實際所需的項目。

在使用資源型政策授予存取函式的許可時，此原則同樣適用。例如，如果您想要授予 Amazon S3 叫用函數的許可，最佳實務是限制對個別儲存貯體或儲存貯體的存取 AWS 帳戶，而不是授予 S3 服務的空白許可。

透過 Lambda 執行程式碼

當您編寫 Lambda 函式時，其實是在建立將在獨特的無伺服器環境中執行的程式碼。了解 Lambda 實際執行程式碼的運作原理，需掌握兩個核心面向：定義程式碼與 Lambda 互動方式的程式設計模型，以及決定 Lambda 如何管理程式碼執行時期環境的執行環境生命週期。

Lambda 程式設計模型

無論您是以 Python、Java 或任何其他支援的語言撰寫，程式設計模型都會做為 Lambda 如何使用程式碼的常見規則集。程式設計模型包含您的執行時期與處理常式。

對於標準函數：

1. Lambda 收到事件。
2. Lambda 會使用執行時間，以您的程式碼可以使用的格式準備事件。
3. 執行時期將格式化的事件傳送至您的處理常式。
4. 您的處理常式會使用您撰寫的程式碼來處理事件。

對於耐用函數：

1. Lambda 收到事件
2. 執行時間會準備事件和 DurableContext
3. 您的處理常式可以：
   - 使用自動檢查點處理步驟
   - 暫停執行而不耗用資源
   - 從最後一個成功的檢查點繼續
   - 在步驟之間維持狀態

此模型的關鍵是處理常式，這是 Lambda 將事件傳送給您的程式碼進行處理的入口。您可以將其視為程式碼的進入點。當 Lambda 收到事件時，會將此事件與部分內容資訊傳遞給您的處理常式。然後，處理常式會執行您的程式碼來處理這些事件。例如，其可能會在有檔案上傳至 Amazon S3 時讀取該檔案、分析映像或更新資料庫。當程式碼完成事件處理後，處理常式即準備就緒，可繼續處理下一個事件。

Lambda 執行模型

雖然程式設計模型會定義 Lambda 與您的程式碼互動的方式，但執行環境是 Lambda 實際執行函數的位置，這是專為您的函數建立的安全隔離運算空間。

每個環境遵循的生命週期會因標準和耐用函數而異：

標準函數（最多 15 分鐘）：

1. 初始化：環境設定和程式碼載入
2. 調用：函數程式碼的單一執行
3. 關閉：環境清除

耐用函數（最長 1 年）：

1. 初始化：環境和持久狀態設定
2. 調用：具有自動檢查點的多個步驟
3. 等待狀態：暫停執行而不耗用資源
4. 繼續：從最後一個檢查點重新啟動
5. 關閉：清除持久狀態

此環境會處理執行函式所涉及的重要層面。其會為函式提供記憶體與 `/tmp` 目錄用於暫時儲存。對於持久性函數，它也會管理：

- 步驟之間的自動狀態持久性
- 檢查點儲存和復原
- 等待狀態協調
- 長時間執行執行的進度追蹤

了解 Lambda 程式設計模型

Lambda 提供兩種程式設計模型：執行最多 15 分鐘的標準函數，以及執行最多一年的耐久函數。雖然兩者都共用核心概念，但耐用函數為長時間執行且具狀態的工作流程新增了功能。

Lambda 提供的程式設計模型對於所有執行時間通用。程式設計模型會定義程式碼與 Lambda 系統之間的介面。透過在函數組態中定義一個處理常式，將函數的進入點告知 Lambda。執行時間會將包含呼叫事件和內容的物件傳入至處理常式，例如函數名稱和請求 ID。

對於耐用函數，處理常式也會收到 DurableContext 物件，提供：

- 透過 step() 的檢查點功能
- 透過 wait() 和 waitForCallback() 等待狀態管理
- 叫用之間的自動狀態持續性

當處理常式完成處理第一個事件時，執行時間就會傳送至另一個。對於耐用函數，處理常式可以暫停步驟之間的執行，而且 Lambda 將在函數恢復時自動儲存和還原狀態。函數的類別會保留在記憶體中，因此可以重複使用在初始化程式碼的處理常式方法外宣告的用戶端和變數。為了節省後續事件的處理時間，請建立可重複使用的資源，例如初始化期間的 AWS SDK 用戶端。初始化後，函式的每個執行個體都可以處理數千個請求。

您的函數也可以存取 `/tmp` 目錄中的本機儲存體，此暫時性快取可用於多個調用。如需詳細資訊，請參閱執行環境。

啟用AWS X-Ray 追蹤時，執行時間會記錄初始化和執行的個別子區段。

執行時間會從您的函數擷取記錄輸出，並將其傳送至 Amazon CloudWatch Logs。除了記錄函式的輸出之外，執行階段也會記錄函式叫用開始和結束的項目。這包含有要求 ID、帳單期、初始化持續時間和其他詳細資料的記錄。如果函數拋出錯誤，執行時間會將該錯誤傳回給叫用者。

> **Note**
>
> 記錄受 CloudWatch Logs 配額限制。日誌資料可能會因節流而遺失，或在某些情況下，因函數的執行個體停止而遺失。

持久性函數的主要差異：

- 狀態會在步驟之間自動保留
- 函數可以暫停執行而不耗用資源
- 步驟會在失敗時自動重試
- 透過檢查點追蹤進度

Lambda 會隨著需求增加執行額外的執行個體，以及隨需求降低停止執行個體，藉以擴展您的函數。此模型會導致應用程式結構的變化，例如：

- 除非另有說明，否則可能會不按順序或同時處理傳入的請求。

- 不依賴函數長期存留的執行個體，而是將應用程式的狀態存放在其他服務中。
- 使用本機儲存和類別層級物件來提高效能，但將部署套裝服務的大小和傳輸至執行環境的資料量降至最低。

如需使用慣用程式設計語言的程式設計模型實作簡介，請參閱以下章節。

- 使用 Node.js 建置 Lambda 函數
- 使用 Python 建置 Lambda 函數
- 使用 Ruby 建置 Lambda 函數
- 使用 Java 建置 Lambda 函數
- 使用 Go 建置 Lambda 函數
- 使用 C# 建置 Lambda 函數
- 使用 PowerShell 建置 Lambda 函數

了解 Lambda 執行環境生命週期

Lambda 執行環境支援標準函數（最多 15 分鐘）和耐用函數（最多一年）。雖然兩者共用相同的基本生命週期，但耐用函數會為長時間執行的工作流程新增狀態管理功能。

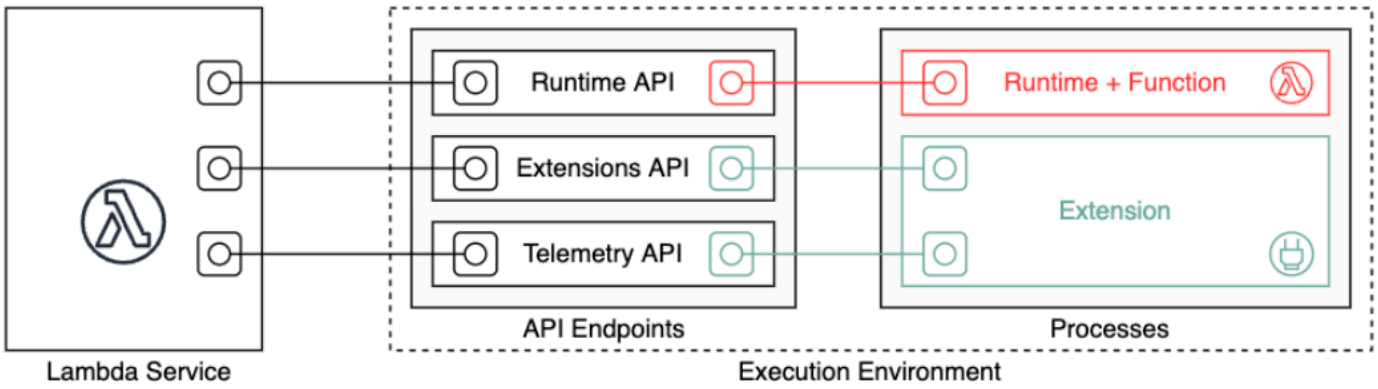
Lambda 會在執行環境中叫用您的函數，該環境可提供安全且隔離的執行時間環境。執行環境會管理執行函式所需的資源。執行環境也會提供函式執行階段的生命週期支援，以及與函式相關聯的任何外部延伸項目。

對於耐用函數，執行環境包含以下項目的其他元件：

- 步驟之間的狀態持久性
- 檢查點管理
- 等待狀態協調
- 進度追蹤

> **Lambda 受管執行個體執行環境**
>
> 如果您使用的是 Lambda 受管執行個體，執行環境與 Lambda（預設）函數有重要的差異。受管執行個體支援並行調用、使用不同的生命週期模型，以及在客戶擁有的基礎設施上執行。如需受管執行個體執行環境的詳細資訊，請參閱 了解 Lambda 受管執行個體執行環境。

函數的執行時間會使用 Runtime API 與 Lambda 進行通訊。延伸項目會使用 Extensions API 與 Lambda 進行通訊。延伸項目還可以透過使用遙測 API 來接收函數的日誌訊息和其他遙測項目。



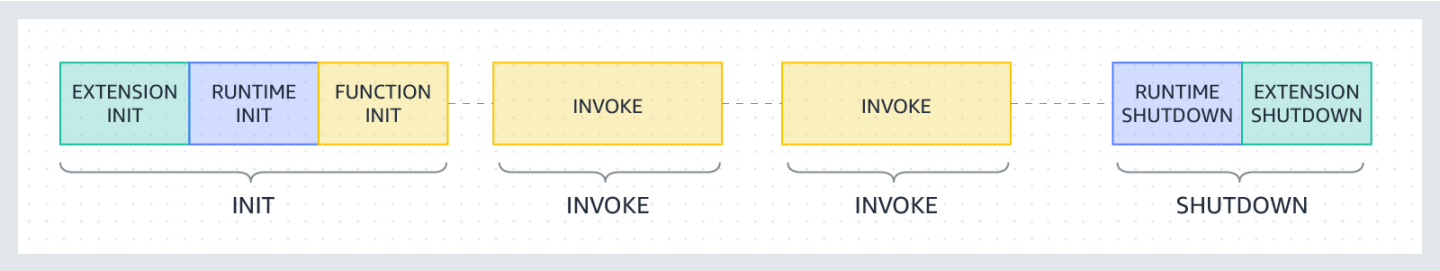
當您建立 Lambda 函數時，您將指定組態資訊，例如您的函數允許的記憶體數量與執行時間上限。Lambda 會使用此資訊來設定執行環境。

函式的執行階段和每個外部延伸項目都是在執行環境中執行的程序。許可、資源、認證和環境變數會在函式和延伸項目之間共用。

主題

- Lambda 執行環境生命週期
- 冷啟動和延遲
- 使用佈建並行減少冷啟動
- 最佳化靜態初始化

Lambda 執行環境生命週期



每個階段都以 Lambda 傳送到執行階段和所有已註冊延伸項目的事件開始。執行時間和每個已註冊延伸項目都會透過傳送 `Next` API 請求來表示已完成。當執行時間和每個延伸項目已完成且沒有擱置的事件時，Lambda 凍結執行環境。

耐用函數的生命週期階段包括：

- 初始化：標準初始化加上持久狀態設定
- 調用：可以包含具有自動檢查點的多個步驟執行
- 等待：函數可以暫停執行，而不會耗用資源
- 繼續：從最後一個檢查點重新啟動函數
- 關閉：清除耐用狀態和資源

主題

- 初始化階段
- 初始化階段期間出現的失敗
- 還原階段 (僅限 Lambda SnapStart)
- 調用階段
- 調用階段期間出現的故障
- 關閉階段

初始化階段

在 `Init` 階段中，Lambda 會執行三項任務：

- 啟動所有延伸項目 (`Extension init`)
- Bootstrap 執行時間 (`Runtime init`)
- 執行該函式的靜態代碼 (`Function init`)
- 執行任何檢查點前的執行時期勾點 (僅限 Lambda SnapStart)

當執行階段和所有延伸項目透過傳送 `Next` API 請求發出訊號表示它們已準備就緒時，`Init` 階段便會結束。`Init` 階段限制為 10 秒。如果所有三項任務都未在 10 秒內完成，Lambda 會在第一次函數調用時以設定的函數逾時重試 `Init` 階段。

在 Lambda SnapStart 啟動的情況下，發佈函數版本時會發生 `Init` 階段。Lambda 會儲存初始化執行環境的記憶體和磁碟狀態快照、保留加密的快照，並快取以進行低延遲存取。如果您擁有檢查點前的執行時期勾點，則程式碼會在 `Init` 階段結束時執行。

> **Note**
>
> 10 秒逾時不適用於使用佈建並行、SnapStart 或 Lambda 受管執行個體的函數。對於佈建並行、SnapStart 和受管執行個體函數，初始化程式碼最多可執行 15 分鐘。時間限制為 130 秒或設定的函數逾時 (最長 900 秒)，以較長者為準。

使用佈建並行時，當您設定函數的電腦設定，Lambda 會初始化執行環境。Lambda 也可確保初始化的執行環境在調用之前隨時可用。您會發現函數調用和初始化階段之間出現差距。根據函數的執行期和記憶體組態，您也會在初始化的執行環境上第一次調用時看到變數延遲。

對於使用隨需並行的函數，Lambda 偶爾會在調用請求之前初始化執行環境。發生這種情況時，您也會注意到函數初始化和調用階段之間出現時間差。建議您不要依賴此行為。

初始化階段期間出現的失敗

如果在 `Init` 階段期間函數當機或出現逾時，Lambda 會在 `INIT_REPORT` 日誌檔中發出錯誤資訊。

Example— 逾時的 INIT_REPORT 日誌

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: timeout
```

Example— 延伸失敗的 INIT_REPORT 日誌

```
INIT_REPORT Init Duration: 1236.04 ms Phase: init Status: error Error Type:
 Extension.Crash
```

如果 `Init` 階段成功，除非已啟用 SnapStart 或佈建並行，否則 Lambda 不會發出 `INIT_REPORT` 日誌。SnapStart 和佈建並行函數一律會發出 `INIT_REPORT`。如需詳細資訊，請參閱監控 Lambda SnapStart。

還原階段 (僅限 Lambda SnapStart)

第一次調用 SnapStart 函數且函數擴展時，Lambda 會從保留的快照繼續新的執行環境，而不是從頭開始初始化函數。如果您設定了還原後執行時期勾點，其程式碼會在 `Restore` 階段結束時執行。您需支付還原後執行時期勾點期間的費用。執行時期必須完成載入，且還原後執行時期勾點必須在逾時限制 (10 秒) 內完成。否則，您將收到 SnapStartTimeoutException 訊息。`Restore` 階段完成時，Lambda 會調用函數處理常式 (調用階段)。

還原階段期間出現的失敗

如果 Restore 階段失敗，Lambda 會在 RESTORE_REPORT 日誌檔中發出錯誤資訊。

Example— 逾時的 RESTORE_REPORT 日誌

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: timeout
```

Example— 執行期勾點失敗的 RESTORE_REPORT 日誌

```
RESTORE_REPORT Restore Duration: 1236.04 ms Status: error Error Type: Runtime.ExitError
```

如需RESTORE_REPORT 日誌的詳細資訊，請參閱 監控 Lambda SnapStart。

調用階段

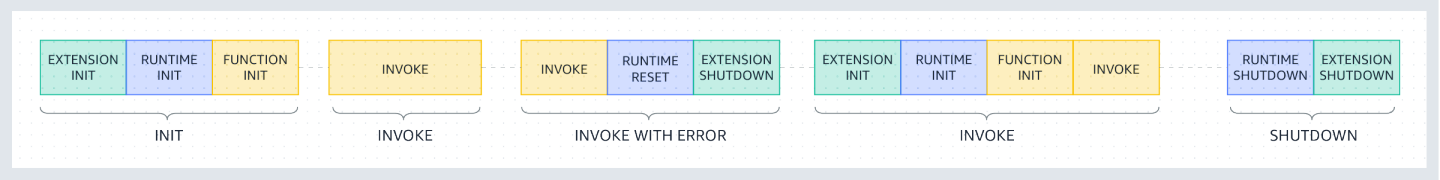
調用 Lambda 函數以回應 Next API 請求時，Lambda 會將 Invoke 事件傳送至執行時間和每個延伸項目。

該函式的逾時設定會限制整個 Invoke 階段的持續時間。例如，如果您將函式逾時設定為 360 秒，則函式和所有延伸項目都需要在 360 秒內完成。請注意，沒有獨立的調用後階段。持續時間是所有調用時間 (執行階段 + 延伸項目) 的總和，直到函式和所有延伸項目完成執行後才會計算。

調用階段會在執行階段後結束，所有延伸項目訊號都透過傳送 Next API 請求完成。

調用階段期間出現的故障

如果 Lambda 函數當機或在 Invoke 階段逾時，Lambda 會重設執行環境。下圖會說明發生調用故障時 Lambda 執行環境的行為：



在前一張示意圖中：

- 第一階段是 INIT 階段，執行期間未發生錯誤。
- 第二階段是 INVOKE 階段，執行期間未發生錯誤。
- 假設在某個時間點，函式遇到調用失敗 (常見原因包括函式逾時、執行時期錯誤、記憶體耗盡、VPC 連線問題、許可錯誤、並行限制以及各種組態問題)。如需可能調用失敗的完整清單，請參閱the section called “調用”。第三階段 (標記為「INVOKE WITH ERROR」) 會說明此狀況。發生此狀況

時，Lambda 服務會進行重設。重設的行為會與 Shutdown 事件一樣。首先，Lambda 會關閉執行階段，然後將 Shutdown 事件傳送給每個已註冊的外部延伸項目。事件會包括關閉的原因。如果將此環境用於新的調用，Lambda 便會在下一次調用時重新初始化延伸項目和執行階段。

請注意，在下一個初始化階段之前，Lambda 重設不會清除 /tmp 目錄內容。這種行為與一般關機階段一致。

> **Note**
>
> AWS 目前正在對 Lambda 服務實作變更。由於這些變更，您可能會看到系統日誌訊息的結構和內容，與 AWS 帳戶中不同 Lambda 函數發出的追蹤區段之間存在細微差異。
> 如果函數的系統日誌組態設為純文字，則此變更會影響函數發生調用失敗時在 CloudWatch Logs 中擷取的日誌訊息。下列範例展示了舊格式和新格式的日誌輸出。
> 這些變化將在未來幾週內實作，除中國和 GovCloud 區域以外，所有 AWS 區域 中的所有函數都會轉換至使用新格式的日誌訊息和追蹤區段。

Example CloudWatch Logs 日誌輸出 (執行時期或延伸項目損毀) - 舊式

```
START RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Version: $LATEST
RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Error: Runtime exited without
 providing a reason
Runtime.ExitError
END RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1
REPORT RequestId: c3252230-c73d-49f6-8844-968c01d1e2e1 Duration: 933.59 ms Billed
 Duration: 934 ms Memory Size: 128 MB Max Memory Used: 9 MB
```

Example CloudWatch Logs 日誌輸出 (函數逾時) - 舊式

```
START RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21 Version: $LATEST
2024-03-04T17:22:38.033Z b70435cc-261c-4438-b9b6-efe4c8f04b21 Task timed out after
 3.00 seconds
END RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21
REPORT RequestId: b70435cc-261c-4438-b9b6-efe4c8f04b21 Duration: 3004.92 ms Billed
 Duration: 3117 ms Memory Size: 128 MB Max Memory Used: 33 MB Init Duration: 111.23
 ms
```

CloudWatch Logs 的新格式包括 REPORT 行中的額外 `status` 欄位。在執行時期或延伸項目損毀的情況下，該 REPORT 行也包含欄位 `ErrorType`。

Example CloudWatch Logs 日誌輸出 (執行時期或延伸項目損毀) - 新式

```
START RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd Version: $LATEST
END RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd
REPORT RequestId: 5b866fb1-7154-4af6-8078-6ef6ca4c2ddd Duration: 133.61 ms Billed
 Duration: 214 ms Memory Size: 128 MB Max Memory Used: 31 MB Init Duration: 80.00
 ms Status: error Error Type: Runtime.ExitError
```

Example CloudWatch Logs 日誌輸出 (函數逾時) - 新式

```
START RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda Version: $LATEST
END RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda
REPORT RequestId: 527cb862-4f5e-49a9-9ae4-a7edc90f0fda Duration: 3016.78 ms Billed
 Duration: 3101 ms Memory Size: 128 MB Max Memory Used: 31 MB Init Duration: 84.00
 ms Status: timeout
```

- 第四階段指的是出現調用故障後立即進入的 INVOKE 階段。在此階段中，Lambda 會透過重新執行 INIT 階段來再次初始化環境。(我們將其稱為隱藏的初始化。) 發生隱藏的初始化時，Lambda 不會在 CloudWatch Logs 中明確回報其他 INIT 階段。相反地，您可能會注意到 REPORT 行中的持續時間包含其他的 INIT 持續時間以及 INVOKE 持續時間。舉例來說，假設您在 CloudWatch 中看到以下日誌：

  ```
  2022-12-20T01:00:00.000-08:00 START RequestId: XXX Version: $LATEST
  2022-12-20T01:00:02.500-08:00 END RequestId: XXX
  2022-12-20T01:00:02.500-08:00 REPORT RequestId: XXX Duration: 3022.91 ms
  Billed Duration: 3000 ms Memory Size: 512 MB Max Memory Used: 157 MB
  ```

  在這個例子中，REPORT 和 START 時間戳記之間的差距是 2.5 秒。這與回報的 3022.91 毫秒持續時間不相符，因為它不會將 Lambda 執行的額外 INIT（隱藏的初始化）納入考量。在這個例子中，您可以推斷實際的 INVOKE 階段花費了 2.5 秒的時間。

  若要深入了解此行為，您可以使用 使用遙測 API 即時存取延伸功能的遙測資料. 調用階段發生隱藏的初始化時，遙測 API 便會透過 `phase=invoke` 發送 `INIT_START`、`INIT_RUNTIME_DONE` 及 `INIT_REPORT` 事件。

- 第五階段指的是 SHUTDOWN 階段，執行期間未發生錯誤。

關閉階段

當 Lambda 即將關閉執行時間，它會將 Shutdown 事件傳送至每個已註冊外部延伸。延伸項目可以使用此時間進行最終清理工作。Shutdown 事件是對 Next API 請求的回應。

持續時間限制：Shutdown 階段的最長持續時間視已註冊延伸項目的組態而定：

- 0 毫秒 - 沒有註冊延伸的函數
- 500 毫秒 - 具有已註冊內部延伸項目的函數
- 2,000 毫秒 - 具有一個或多個已註冊外部延伸項目的函數

如果執行時間或延伸項目未在限制內回應 Shutdown 事件，Lambda 會使用 SIGKILL 訊號結束程序。

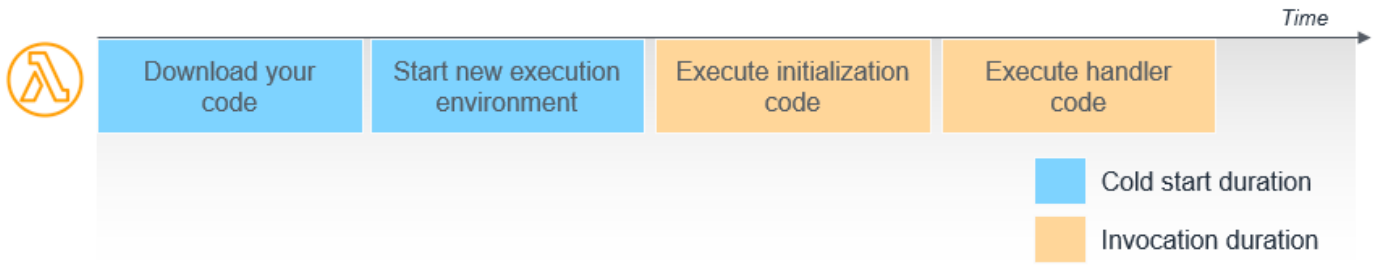
在函數和所有延伸項目完成之後，Lambda 會維護執行環境一段時間，並預期另一個函數調用。不過，Lambda 每隔幾小時終止一次執行環境，以允許執行時期更新和維護，即使對於持續調用的函數也是如此。您不應該假設執行環境將無限期持續運作。如需詳細資訊，請參閱在函式中實作無狀態。

再次叫用函數時，Lambda 會解凍環境以重複使用。重複使用執行環境具有下列含義：

- 在函數處理常式方法外宣告的物件會保持初始化，於再次呼叫函數時提供額外的最佳化。例如，假設您的 Lambda 函數建立資料庫連線，而不是重建連線，那麼在後續呼叫時便會使用原始連線。建議您在程式碼中新增邏輯，在建立連線前先確認是否存在既有連線。
- 每個執行環境都會在 /tmp 目錄中提供 512 MB 到 10,240 MB 的磁碟空間，增量為 1 MB。執行內容凍結時，目錄環境會凍結，所提供的暫時性可用於多重調用。您可以新增額外的程式碼，確認快取是否具有您已儲存的資料。如需部署大小限制的詳細資訊，請參閱Lambda 配額。
- 如果 Lambda 重複使用執行環境，則會恢復由 Lambda 函數啟動且在函數結束時沒有完成的背景程序或回呼。請確定程式碼中的任何背景程序或回呼在程式碼存在前已完成。

冷啟動和延遲

當 Lambda 收到透過 Lambda API 執行函式的請求時，此服務會先準備執行環境。在此初始化階段，服務會下載程式碼、啟動環境，並執行主要處理常式外部的任何初始化程式碼。最後，Lambda 才會執行處理常式程式碼。



在此圖表中，下載程式碼和設定環境的前兩個步驟通常稱為「冷啟動」。這段時間會產生費用，且會增加調用的整體延遲時間。

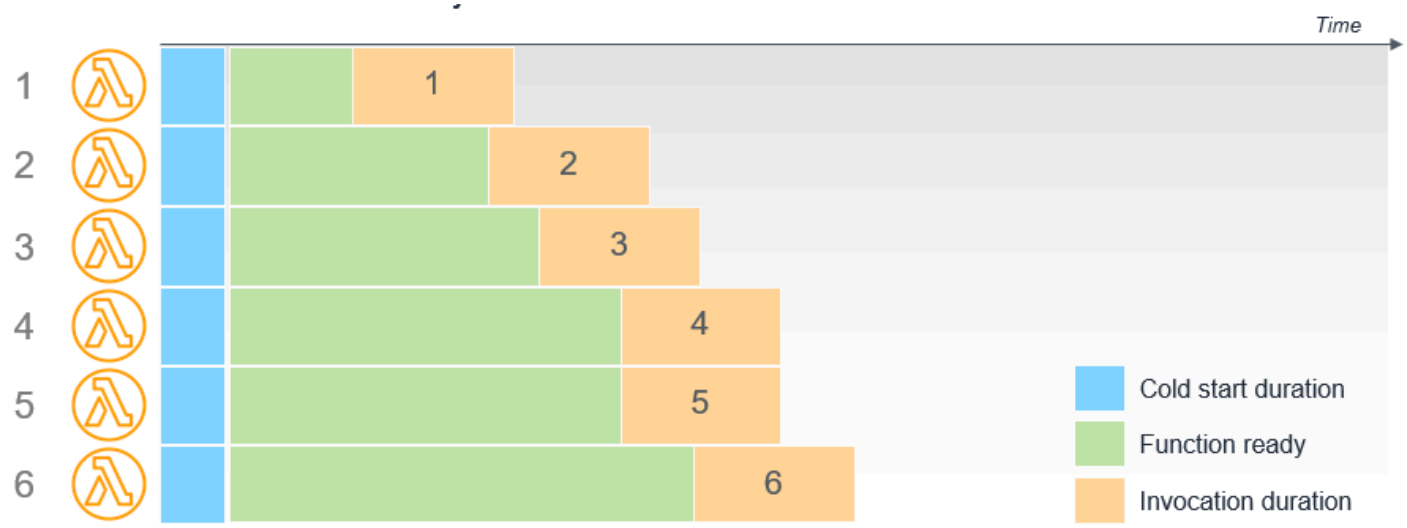
調用完成之後，執行環境會凍結。為了改善資源管理和效能，Lambda 會將執行環境保留一段時間。在此期間，如果收到另一個針對相同函式的請求，Lambda 可以重複使用該環境。第二個請求通常會更快完成，因為執行環境已完全設定。這就是所謂的「暖啟動」。

冷啟動通常發生在不到 1% 的調用中。冷啟動的持續時間從低於 100 毫秒至超過 1 秒不等。一般而言，冷啟動在開發和測試函式中通常比在生產工作負載更為常見。這是因為開發和測試函數的調用頻率通常較低。

使用佈建並行減少冷啟動

如果工作負載需要可預測的函式啟動時間，建議使用佈建並行解決方案，確保盡可能降低延遲。此功能會預先初始化執行環境，從而減少冷啟動時間。

例如，若函式的佈建並行為 6，表示具有 6 個預先暖啟動的執行環境。



最佳化靜態初始化

靜態初始化在處理常式程式碼開始在函數中執行之前進行。這是您在主要處理常式外部提供的初始化程式碼。此程式碼通常用於匯入程式庫和相依項、設定組態，以及初始化與其他服務的連線。

下列 Python 範例示範了如何在調用期間執行 `lambda_handler` 函式之前，在初始化階段匯入和設定模組以及建立 Amazon S3 用戶端。

```
import os
import json
import cv2
import logging
import boto3

s3 = boto3.client('s3')
logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):

  # Handler logic...
```

函式執行前的延遲，主要來自初始化程式碼的執行時間。首次建立新的執行環境，即會執行此程式碼。如果調用使用暖執行環境，則不會再次執行初始化程式碼。影響初始化程式碼延遲的因素包括：

- 函數套件的大小，包括匯入的程式庫和相依項以及 Lambda 層。
- 程式碼和初始化工作的數量。
- 程式庫和其他服務在設定連線及其他資源方面的表現。

開發人員可以採取多個步驟來最佳化靜態初始化延遲。如果一個函數具有許多物件和連線，您可以將單一函數重新架構為多個專用函數。這些函式各自較小，並且每個函式的初始化程式碼也更少。

重要的是，函數僅匯入所需的程式庫和相依項。例如，如果您只在 AWS SDK 中使用 Amazon DynamoDB，您可以要求個別服務，而不是整個 SDK。比較下列三個範例：

```
// Instead of const AWS = require('aws-sdk'), use:
const DynamoDB = require('aws-sdk/clients/dynamodb')

// Instead of const AWSXRay = require('aws-xray-sdk'), use:
const AWSXRay = require('aws-xray-sdk-core')
```

```
// Instead of const AWS = AWSXRay.captureAWS(require('aws-sdk')), use:
const dynamodb = new DynamoDB.DocumentClient()
AWSXRay.captureAWSClient(dynamodb.service)
```

靜態初始化通常也是開啟資料庫連線的最佳位置，可允許函式在對同一執行環境的多次調用中重複使用連線。但是，您可能具有大量僅在函數的某些執行路徑中使用的物件。在這種情況下，您可以在全域範圍內延遲載入變數，以縮短靜態初始化持續時間。

避免使用全域變數來取得特定內容的資訊。如果您的函數具有僅在單次調用的生命週期內使用並在下次調用時重設的全域變數，請使用處理常式本機的變數範圍。這不僅可以防止調用間的全域變數洩漏，還可以改善靜態初始化效能。

透過 Lambda 建立事件驅動型架構

事件是指能夠觸發 Lambda 函式執行的任何事物。事件可以透過兩種方式觸發 Lambda 函數：透過直接調用 （ 推送 ） 和事件來源映射 （ 提取 ）。

許多 AWS 服務可以直接叫用您的 Lambda 函數。這些服務會將事件推送至您的 Lambda 函式。觸發函數的事件幾乎可以是任何事件，從透過 API Gateway 的 HTTP 請求、由 EventBridge 規則管理的排程、 AWS IoT 事件或 Amazon S3 事件。透過事件來源映射，Lambda 會主動從佇列或串流擷取 (或拉取) 事件。您僅需設定 Lambda 檢查來自支援服務的事件，Lambda 便會自動處理輪詢與函式調用作業。

事件在傳遞至函式時，會以 JSON 格式結構化呈現。JSON 結構會因事件來源服務與類型而異。雖然標準 Lambda 函數調用最多可持續 15 分鐘 （ 或使用耐久函數最長一年 ），但 Lambda 最適合持續 1 秒或更短的短調用。此特性在事件驅動型架構中尤為明顯：在這種架構中，每個 Lambda 函式都被視為一個微服務，負責執行一組精簡的特定指令。

> **Note**
>
> 事件驅動型架構使用網路跨不同系統進行通訊，這會引入可變延遲。對於需要極低延遲的工作負載，例如即時交易系統，此設計可能不是最佳選擇。但是，對於高度可擴展和可用的工作負載，或是具有不可預測流量模式的工作負載，事件驅動型架構可以提供有效的方式來符合這些需求。

主題

- 事件驅動型架構的優勢
- 事件驅動型架構的權衡
- 基於 Lambda 的事件驅動型應用程式反模式

事件驅動型架構的優勢

在事件驅動型架構中，Lambda 支援兩種調用方法：

1. 直接調用 （ 推送方法 ）： AWS 服務會直接觸發 Lambda 函數。例如：
   - Amazon S3 於檔案上傳時觸發函式
   - API Gateway 於接收 HTTP 請求時觸發函式

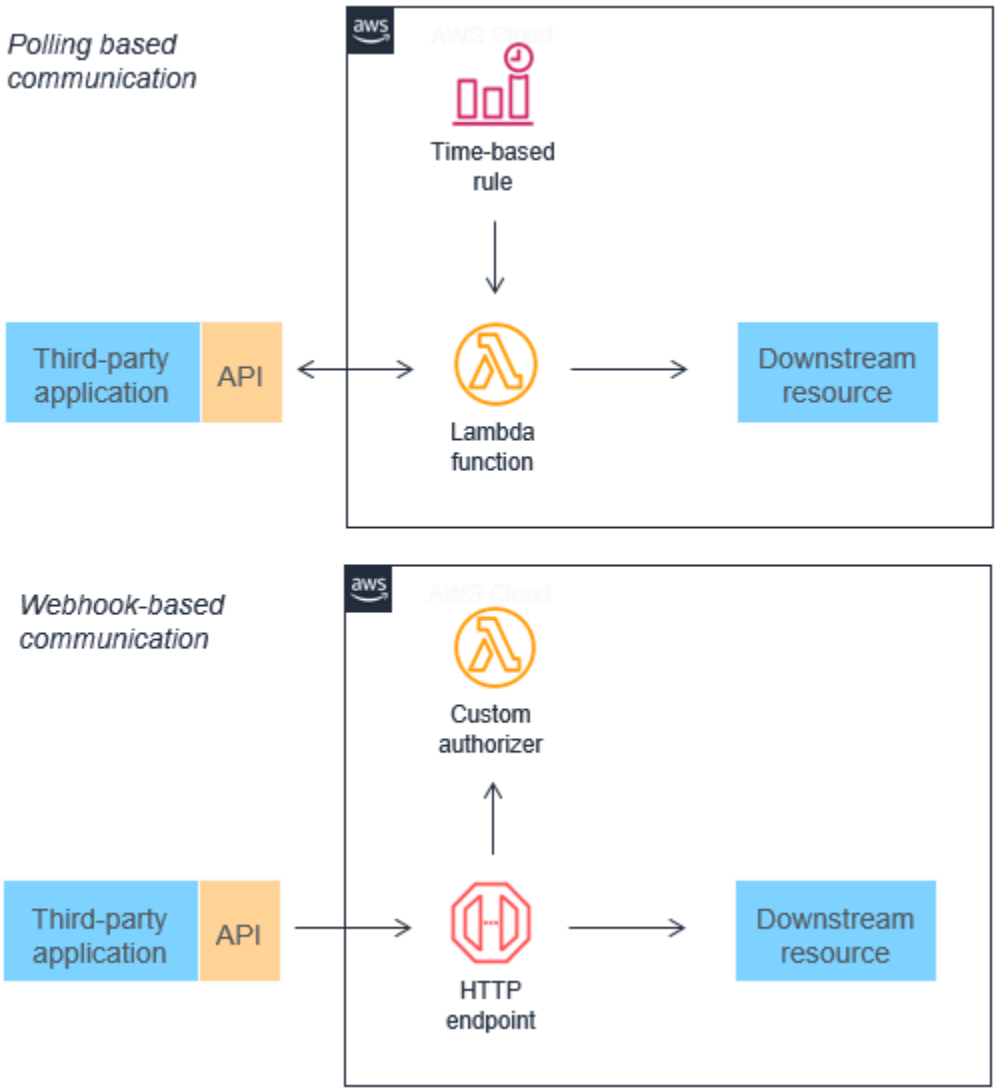
2. 事件來源映射 (拉取方法)：Lambda 會擷取事件並調用函式。例如：

- Lambda 從 Amazon SQS 佇列擷取訊息後調用函式
- Lambda 從 DynamoDB 串流讀取記錄後調用函式

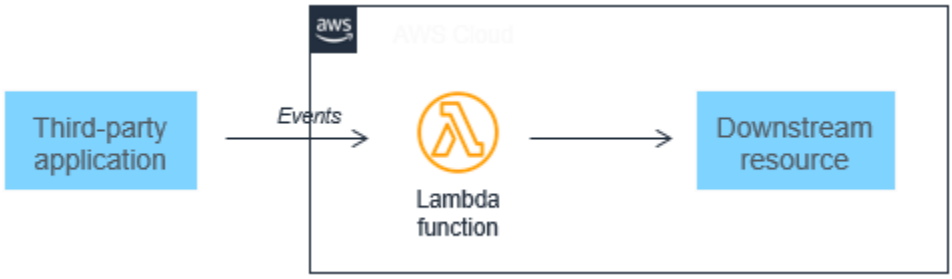
如下文所述，這兩種方法共同實現了事件驅動型架構的優勢。

將輪詢和 Webhook 取代為事件

許多傳統架構會透過輪詢與 Webhook 機制，在不同元件之間傳達狀態資訊。輪詢在擷取更新時的效率可能非常低，因為新資料變得可用和與下游服務同步之間存在延遲。您想要整合的其他微服務並非總是支援 Webhook。它們也可能需要自訂授權和身分驗證組態。在這兩種情況下，如果沒有開發團隊的額外工作，這些整合方法難以隨需擴展。



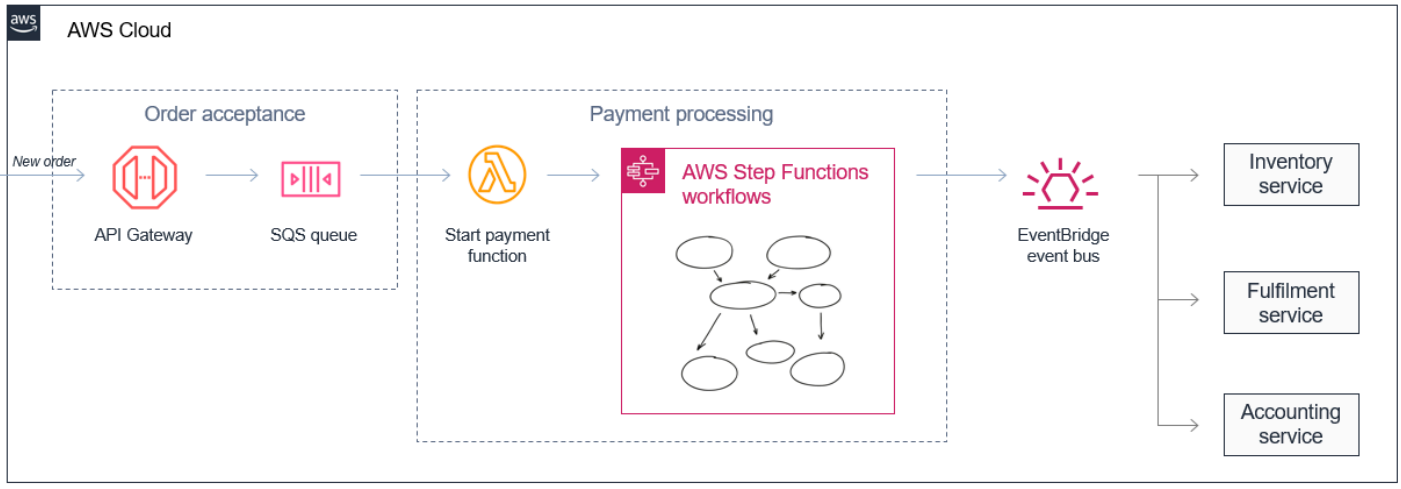
這兩種機制都可以取代為事件，這些事件可以篩選、路由並推送至下游使用微服務。此方法可能會導致頻寬耗用減少、CPU 使用率降低，並可能降低成本。這些架構還可以降低複雜性，因為每個功能單元都較小，而且程式碼通常較少。



事件驅動型架構也可以更輕鬆地設計近乎即時的系統，協助組織遠離批次型處理。事件會在應用程式中的狀態變更時產生，因此微服務的自訂程式碼應設計為處理單一事件。由於擴展由 Lambda 服務處理，因此該架構無需變更自訂程式碼，即可應對流量的顯著增加。隨著事件向上擴展，處理事件的運算層也隨之擴展。

降低複雜性

微服務可讓開發人員和架構師簡化複雜的工作流程。例如，電子商務整體可以分為訂單接受和付款程序，使用單獨的庫存、履行和會計服務。在整體中管理和協調可能很複雜，會成為一系列與事件非同步通訊的解耦服務。



此方法也可讓您組合以不同速率處理資料的服務。在此情況下，訂單接受微服務可以透過緩衝 Amazon SQS 佇列中的訊息來存放大量傳入訂單。

由於處理付款的複雜性，付款處理服務通常較慢，因此可以從 Amazon SQS 佇列取得穩定的訊息串流。它可以使用 協調複雜的重試和錯誤處理邏輯 AWS Step Functions，並為數十萬個訂單協調作用中的付款工作流程。

替代方法：對於使用標準程式設計語言的協同運作，您可以使用 Lambda 耐久函數。耐用的函數可讓您使用自動檢查點和重試，以程式碼撰寫訂單接受、付款處理和通知邏輯。當工作流程主要涉及 Lambda 函數，而且您偏好在程式碼中保留協同運作邏輯時，此方法運作良好。

改善可擴展性和可擴充性

微服務產生的事件通常會發布至 Amazon SNS 和 Amazon SQS 等傳訊服務。這些行為就像微服務之間的彈性緩衝，有助於在流量增加時處理擴展。然後，Amazon EventBridge 之類的服務可以根據事件的內容篩選和路由訊息，如規則中所定義。因此，事件型應用程式可能比單體式應用程式更具可擴展性，並提供更大的備援。

此系統也具有高度可擴展性，允許其他團隊擴展特性並新增功能，而不會影響訂單處理和付款處理微服務。透過使用 EventBridge 發布事件，此應用程式可與庫存微服務等現有系統整合，但也可讓任何未來應用程式以事件取用者身分整合。事件的生產者不了解事件取用者，這有助於簡化微服務邏輯。

事件驅動型架構的權衡

可變延遲

與單一裝置上可能處理相同記憶體空間內所有內容的單體應用程式不同，事件驅動型應用程式會跨網路進行通訊。此設計引入了可變延遲。雖然可以設計應用程式以將延遲降至最低，但單體式應用程式幾乎總是可以最佳化來降低延遲，而犧牲可擴展性和可用性。

需要穩定低延遲效能的工作負載 (例如銀行中的高頻交易應用程式，或倉儲中低於一毫秒的機器人自動化)，並非事件驅動型架構的理想候選項目。

最終一致性

事件代表狀態變更，許多事件在任何指定時間點流經架構中的不同服務，此類工作負載通常最終會一致。這使得處理交易、處理重複項目或確定系統的確切整體狀態更複雜。

某些工作負載包含最終一致 (例如，目前小時內的總訂單數) 或高度一致 (例如，目前庫存) 的要求組合。對於那些需要高度資料一致性的工作負載，有架構模式可支援此作業。例如：

- DynamoDB 可以提供高度一致性讀取，有時延遲較高，會耗用比預設模式更高的輸送量。DynamoDB 也能支援交易，可協助維持資料一致性。
- 您可以使用 Amazon RDS 來實現需要 ACID 屬性的功能，但關聯式資料庫的可擴展性一般低於 DynamoDB 等 NoSQL 資料庫。Amazon RDS Proxy 可協助管理 Lambda 函數等暫時性取用者的連線集區和擴展。

事件型架構通常以個別事件為基礎設計，而非大批次資料。通常，工作流程旨在管理個別事件或執行流程的步驟，而不是同時在多個事件上操作。在無伺服器架構中，即時事件處理優先於批次處理：應以多次小規模增量更新取代批次作業。雖然這可以提高工作負載的可用性和可擴展性，但也讓事件更難以感知其他事件。

將值傳回給呼叫者

在許多情況下，事件型應用程式是非同步的。這意味著呼叫者服務不會等待來自其他服務的請求，然後再繼續其他工作。這是事件驅動型架構的基本特性，可實現可擴展性和彈性。這意味著傳遞傳回值或工作流程的結果會比同步執行流程更複雜。

生產系統中的大多數 Lambda 調用都是非同步的，可回應來自 Amazon S3 或 Amazon SQS 等服務的事件。在這些情況下，處理事件的成功或失敗通常比傳回值更重要。在 Lambda 中提供無效字母佇列 (DLQ) 等功能，以確保您可以識別和重試失敗的事件，而無需通知呼叫者。

跨服務和函數進行偵錯

與單體式應用程式相比，偵錯事件驅動型系統的方式也有所不同。由於傳遞事件的系統和服務不同，因此無法在發生錯誤時記錄和重現多個服務的確切狀態。由於每個服務和函數調用都具有單獨的日誌檔案，因此確定導致錯誤的特定事件發生的情況可能更複雜。

在事件驅動型系統中成功建置偵錯方法有三個重要要求。首先，強大的記錄系統至關重要，這由 Amazon CloudWatch 跨 AWS 服務提供，並內嵌在 Lambda 函數中。其次，在這些系統中，請務必確保每個事件都具有一個交易識別碼，該識別碼在整個交易的每個步驟中都會記錄下來，以在搜尋日誌時提供協助。

最後，強烈建議使用偵錯和監控服務 (例如 AWS X-Ray) 來自動化日誌的剖析和分析。這可以跨多個 Lambda 調用和服務取用日誌，讓您更輕鬆地找出問題的根本原因。如需有關使用 X-Ray 進行疑難排解的深度資訊，請參閱疑難排解演練。

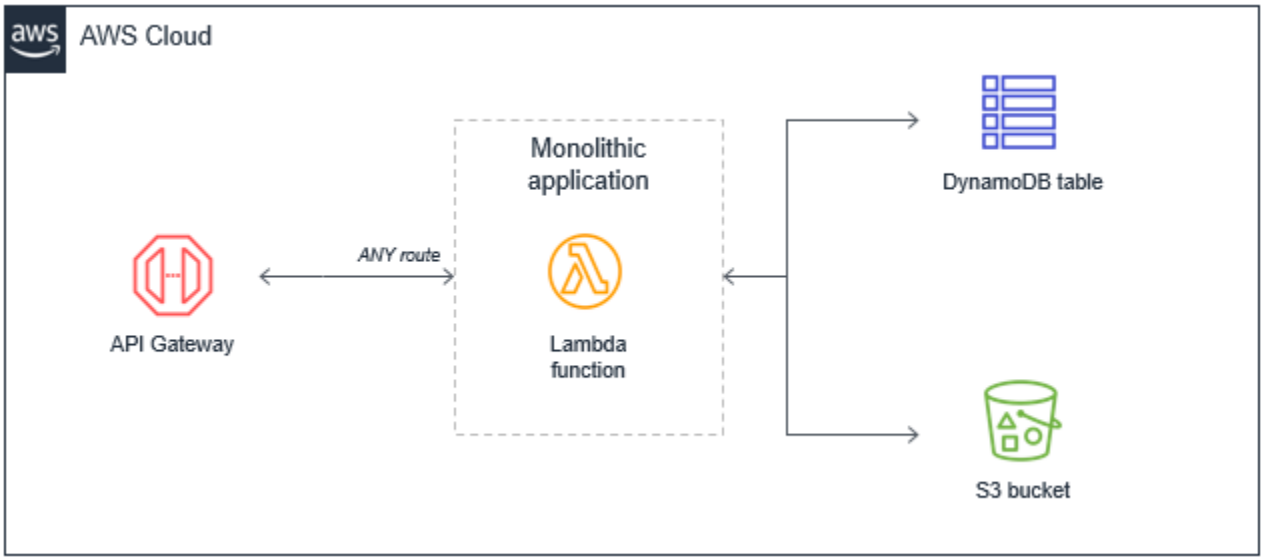
基於 Lambda 的事件驅動型應用程式反模式

使用 Lambda 建置事件驅動型架構時，請避免下列常見的反模式。這些模式有效，但可能會增加成本和複雜性。

Lambda 單體

在許多從傳統伺服器 (例如 Amazon EC2 執行個體或 Elastic Beanstalk 應用程式) 遷移而來的應用程式中，開發人員通常會採用「直接移轉」的方式沿用現有程式碼。通常，這會產生單一 Lambda 函數，

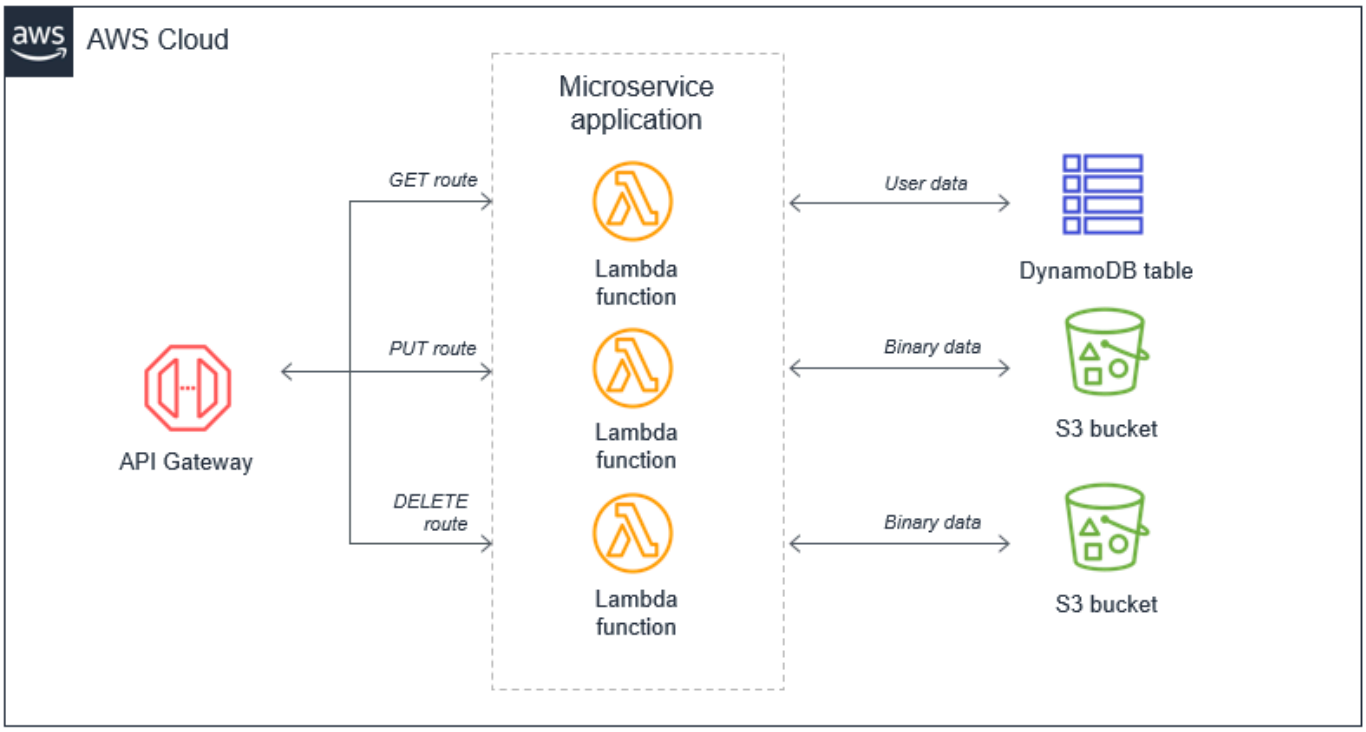
其中包含針對所有事件觸發的所有應用程式邏輯。對於基本 Web 應用程式，單體式 Lambda 函數會處理所有 API Gateway 路由，並與所有必要的下游資源整合。



此方法有數個缺點：

- 套件大小 – Lambda 函數可能較大，因為它包含所有路徑的所有可能程式碼，這使得 Lambda 服務執行速度變慢。
- 難以強制執行最低權限：函式的執行角色必須允許所有路徑所需的所有資源的許可，導致許可範圍非常廣泛。這是一個安全隱患題。功能單體中的許多路徑不需要所有已授予的許可。
- 更難升級：在生產系統中，對單一函式的任何升級風險都較大，可能導致整個應用程式中斷運作。升級 Lambda 函數中的單一路徑就是對整個函數的升級。
- 更難維護：由於程式碼儲存庫為單體架構，多位開發人員協同處理服務時會面臨更多困難。它也會增加開發人員的認知負擔，並使為程式碼建立適當的測試涵蓋範圍變得更加困難。
- 更難重複使用程式碼：更加難以將可重複使用的程式庫與單體分隔，導致程式碼重複使用變得更加困難。當您開發和支援更多專案時，這可能會讓支援程式碼和擴展團隊速度變得更加困難。
- 更難測試：隨著程式碼行數的增加，對程式碼庫中所有可能的輸入和進入點組合進行單元測試會更加困難。對於程式碼較少的小型服務，實作單元測試通常更容易。

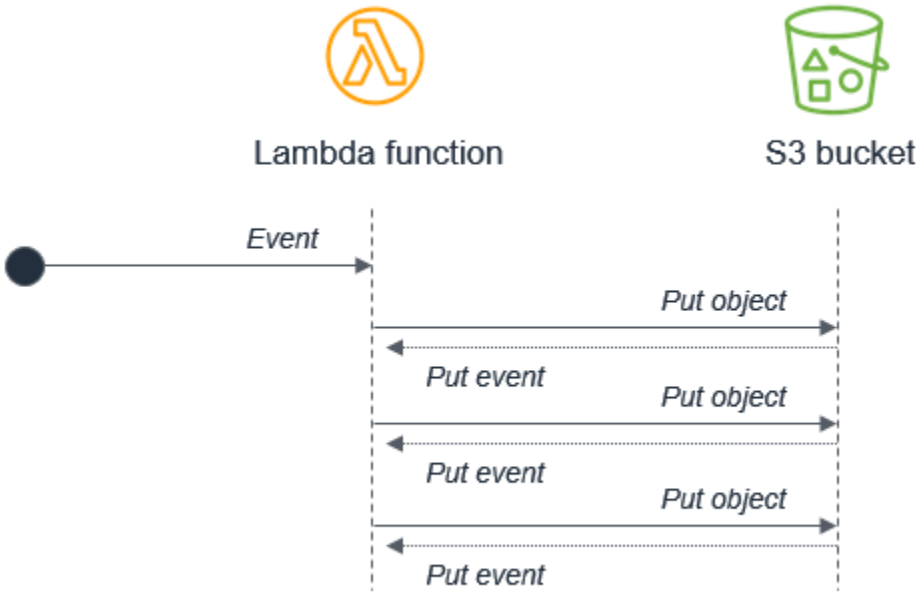
首選的替代方案是將單體 Lambda 函數分解為個別微服務，將單一 Lambda 函數映射至單一定義良好的任務。在這個具有幾個 API 端點的簡易 Web 應用程式中，產生的微服務型架構可以基於 API Gateway 路由。



導致 Lambda 函數失控的遞迴模式

AWS 服務會產生叫用 Lambda 函數的事件，而 Lambda 函數可以將訊息傳送到 AWS 服務。通常，調用 Lambda 函數的服務或資源應與函數輸出至的服務或資源不同。如果不進行管理，可能會導致無限迴圈。

例如，Lambda 函式會將物件寫入 Amazon S3 物件，進而透過 put 事件調用同一個 Lambda 函式。調用會導致第二個物件寫入儲存貯體，這會調用相同的 Lambda 函數：



雖然大多數程式設計語言都存在無限迴圈的可能性，但此反模式有可能在無伺服器應用程式中取用更多資源。Lambda 和 Amazon S3 都會根據流量自動擴展，因此迴圈可能會導致 Lambda 擴展以使用所有可用的並行，Amazon S3 將繼續寫入物件並為 Lambda 產生更多事件。

此範例以 S3 為例，但 Amazon SNS、Amazon SQS、DynamoDB 及其他服務中也存在遞迴迴圈的風險。您可以使用遞迴迴圈偵測功能，找出並避免此反模式。

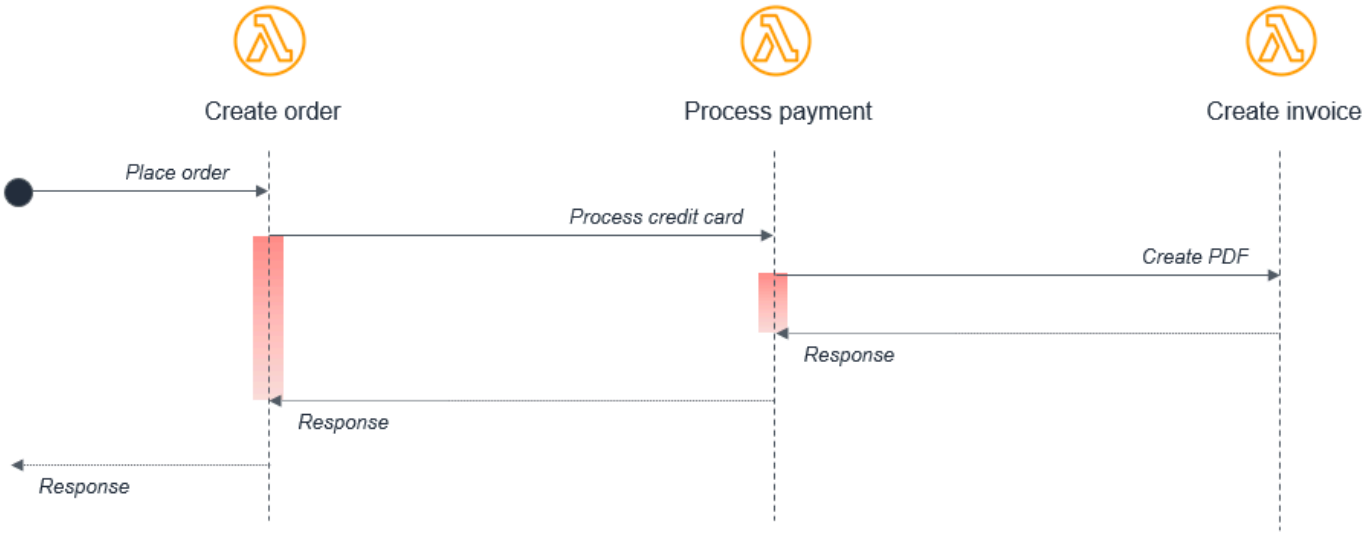
呼叫 Lambda 函數的 Lambda 函數

函數支援封裝和程式碼重複使用。大多數程式設計語言都支援程式碼同步呼叫程式碼庫中的函數的概念。在此情況下，呼叫者會等到函數傳回回應。

> **Note**
>
> 雖然 Lambda 函數因為成本和複雜性問題而直接呼叫其他 Lambda 函數通常是反模式，但這不適用於耐用函數，其專門設計來調用其他函數來協調多步驟工作流程。

當傳統伺服器或虛擬執行個體發生這種情況時，作業系統排程器會切換至其他可用的工作。CPU 以 0% 還是 100% 執行並不影響應用程式的整體成本，因為您要支付擁有和營運伺服器的固定成本。

此模型通常無法很好地適應無伺服器開發。例如，請考慮一個由三個處理訂單的 Lambda 函數組成的簡易電子商務應用程式：



在此案例中，Create order 函數會呼叫 Process payment 函數，進而呼叫 Create invoice 函數。雖然此同步流程可能在伺服器上的單一應用程式內運作，但在分散式無伺服器架構中引入了幾個可避免的問題：

- 成本：使用 Lambda 時，您需要支付調用期間的費用。在此範例中，當 Create invoice 函式執行時，其他兩個函式也會在等待狀態下執行，在圖表上以紅色顯示。
- 錯誤處理：在巢狀調用中，錯誤處理可能會變得更複雜。例如，建立發票中的錯誤可能需要程序付款函數來撤銷費用，或者可能改為重試建立發票程序。
- 緊耦合：處理付款通常比建立發票需要更長時間。在此模型中，整個工作流程的可用性受最慢函數限制。
- 擴展：所有三個函式的並行必須相等。在忙碌的系統中，這會使用比其他方式需要更多的並行。

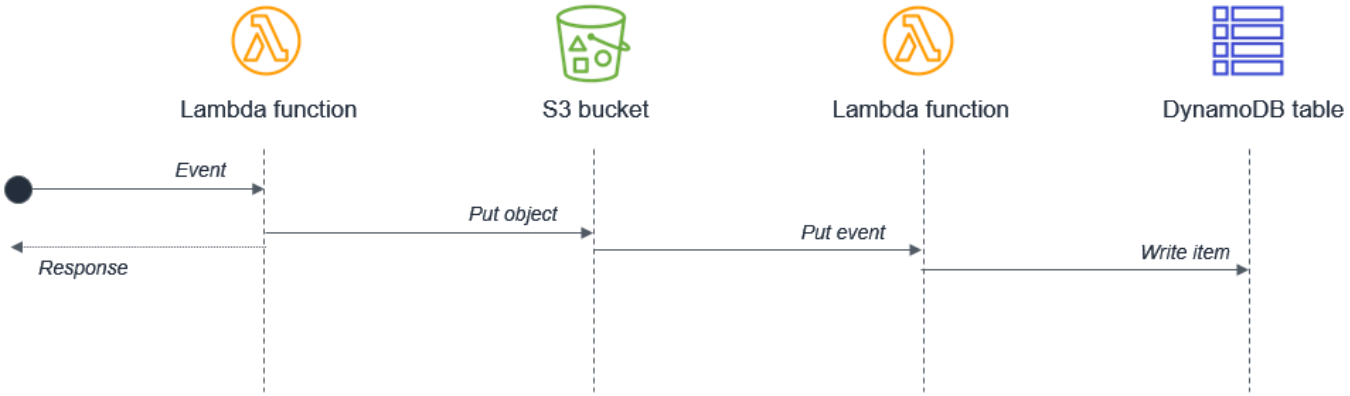
在無伺服器應用程式中，有兩種常見方法可以避免此模式。首先，在 Lambda 函式之間使用 Amazon SQS 佇列。如果下游程序比上游程序更慢，佇列會長期保留訊息，並解耦這兩個函數。在此範例中，建立訂單函數會將訊息發佈至 Amazon SQS 佇列，而程序付款函數會使用來自佇列的訊息。

第二個方法是使用 AWS Step Functions。對於具有多種類型的故障和重試邏輯的複雜程序，Step Functions 有助於減少協調工作流程所需的自訂程式碼數量。因此，Step Functions 會協調工作並穩健地處理錯誤和重試，而 Lambda 函數僅包含商業邏輯。

在單一 Lambda 函數內的同步等待

請確定未在單一 Lambda 函數中同步排程任何可能並行的活動。例如，Lambda 函數可能會寫入 S3 儲存貯體，然後寫入 DynamoDB 資料表：

在此設計中，由於各項互動採取循序執行方式，等待時間會逐漸累積放大。若第二個任務需依賴第一個任務完成，您可透過兩個獨立的 Lambda 函式來減少總等待時間與執行成本：



在此設計中，第一個 Lambda 函式會在將物件放入 Amazon S3 儲存貯體之後立即回應。S3 服務會調用第二個 Lambda 函數，然後將資料寫入至 DynamoDB 資料表。此方法可將 Lambda 函數執行的總等待時間降至最低。

設計 Lambda 應用程式

架構良好的事件驅動應用程式使用 AWS 服務和自訂程式碼的組合來處理和管理請求和資料。本章著重介紹應用程式設計中與 Lambda 相關的主題。在為忙碌的生產系統設計應用程式時，無伺服器架構師有許多重要的考量。

許多適用於軟體開發和分散式系統的最佳實務，也適用於無伺服器應用程式開發。整體目標是開發符合下列條件的工作負載：

- 可靠 – 為您的最終使用者提供高水準的可用性。 AWS 無伺服器服務是可靠的，因為其亦專為解決故障而設計。
- 耐久 – 提供符合工作負載耐久性需求的儲存選項。
- 安全 – 遵循最佳實務，並使用提供的工具來保護對工作負載的存取，並在發生任何問題時限制影響範圍。
- 高效能 – 有效率地使用運算資源，並滿足最終使用者的效能需求。
- 具成本效益 – 設計可避免不必要的成本的架構，這些架構可以在不超支的情況下進行擴展及除役，而不會產生大量開銷。

下列設計原則可協助您建置符合這些目標的工作負載。並非每一項原則都適用於所有架構，但這些原則應能為您的總體架構決策提供指引。

主題

- 使用服務而非自訂程式碼
- 了解 Lambda 抽象層級
- 在函式中實作無狀態
- 最小化耦合度
- 針對隨選資料而非批次處理進行建置
- 選擇複雜工作流程的協同運作選項
- 實作冪等性
- 使用多個 AWS 帳戶來管理配額

使用服務而非自訂程式碼

無伺服器應用程式通常包含數個 AWS 服務，與 Lambda 函數中執行的自訂程式碼整合。雖然 Lambda 可以與大多數 AWS 服務整合，但無伺服器應用程式中最常使用的服務如下：

Category	AWS 服務
運算	AWS Lambda
資料儲存	Amazon S3 Amazon DynamoDB Amazon RDS
API	Amazon API Gateway
應用程式整合	Amazon EventBridge Amazon SNS Amazon SQS
協調	Lambda 耐用函數 AWS Step Functions
串流資料和分析	Amazon Data Firehose

> **Note**
>
> 許多無伺服器服務皆提供資料複寫功能與多區域支援，包括 DynamoDB 與 Amazon S3。Lambda 函數可以作為部署管道的一部分部署在多個區域中，並且可以將 API Gateway 設定為支援此組態。請參閱此範例架構，其示範如何實現這一點。

分散式架構中有許多建立良好的常見模式，您可以使用 AWS 服務自行建置或實作。對於大多數客戶，投資時間從頭開始開發這些模式幾乎沒有商業價值。當您的應用程式需要其中一個模式時，請使用對應的 AWS 服務：

模式	AWS 服務
佇列	Amazon SQS

模式	AWS 服務
事件匯流排	Amazon EventBridge
發布/訂閱 (扇出)	Amazon SNS
協調	Lambda 耐用函數 AWS Step Functions
API	Amazon API Gateway
事件串流	Amazon Kinesis

這些服務旨在與 Lambda 整合，您可以使用基礎設施即程式碼 (IaC) 來建立和捨棄服務中的資源。您可以透過 AWS SDK 使用其中任何服務，而無需安裝應用程式或設定伺服器。熟悉透過 Lambda 函數中的程式碼使用這些服務，是生產設計良好的無伺服器應用程式的重要步驟。

了解 Lambda 抽象層級

Lambda 服務會限制您對執行 Lambda 函數和基礎作業系統、Hypervisor 和硬體的存取。此服務會持續改善和變更基礎設施，以新增功能、降低成本，並提高服務效能。您的程式碼應假設不了解 Lambda 的架構方式，且假設沒有硬體親和性。

同樣地，Lambda 與其他 服務的整合是由 管理 AWS，只有少量的組態選項會公開給您。例如，API Gateway 與 Lambda 互動時無需考慮負載平衡，因為這項功能已完全由服務本身管理。您也無法直接控制服務在任何時間點調用函式時所使用的可用區域，亦無法控制 Lambda 如何決定執行環境數量的向上擴展或縮減時機。

此抽象概念可協助您專注於應用程式的整合層面、資料流程，以及工作負載為最終使用者提供價值的商業邏輯。允許服務管理基礎機制可協助您更快速地開發應用程式，同時減少要維護的自訂程式碼。

在函式中實作無狀態

對於標準 Lambda 函數，您應該假設環境僅存在於單一調用中。函式應在首次啟動時初始化任何必要的狀態。例如，函式可能需要從 DynamoDB 資料表擷取資料。在結束前，函式應將所有永久性資料變更提交至 Amazon S3、DynamoDB 或 Amazon SQS 等耐久存放區。其不應依賴任何現有的資料結構或暫存檔案，或任何需要多次調用才能管理的內部狀態。

使用耐用函數時，狀態會在叫用之間自動保留，無需手動將狀態保留到外部儲存體。不過，對於未透過 DurableContext 明確管理的任何資料，您仍應遵循無狀態原則。

若要初始化資料庫連線與程式庫，或是載入狀態資訊，您可以使用靜態初始化功能。由於執行環境會盡可能重複使用以改善效能，因此您可以在多次調用中分攤初始化這些資源所花費的時間。但是，您不應在此全域範圍內儲存函數中使用的任何變數或資料。

最小化耦合度

大多數架構應偏好許多、較短的函數，而不是較少、較大的函數。每個函數的用途應該是處理傳遞到函數的事件，而無需了解或預期整個工作流程或交易量。這使得此函數與事件來源無關，且與其他服務的耦合最少。

任何不常變更的全域範圍常數都應作為環境變數實作，以允許在不部署的情況下進行更新。任何機密或敏感資訊都應儲存在 AWS Systems Manager Parameter Store 或 AWS Secrets Manager 中，並由函數載入。由於這些資源具有帳戶專屬特性，您可建立跨多個帳戶的建置管道。管道根據環境載入適當的機密，而無需向開發人員公開這些機密，也無需變更任何程式碼。

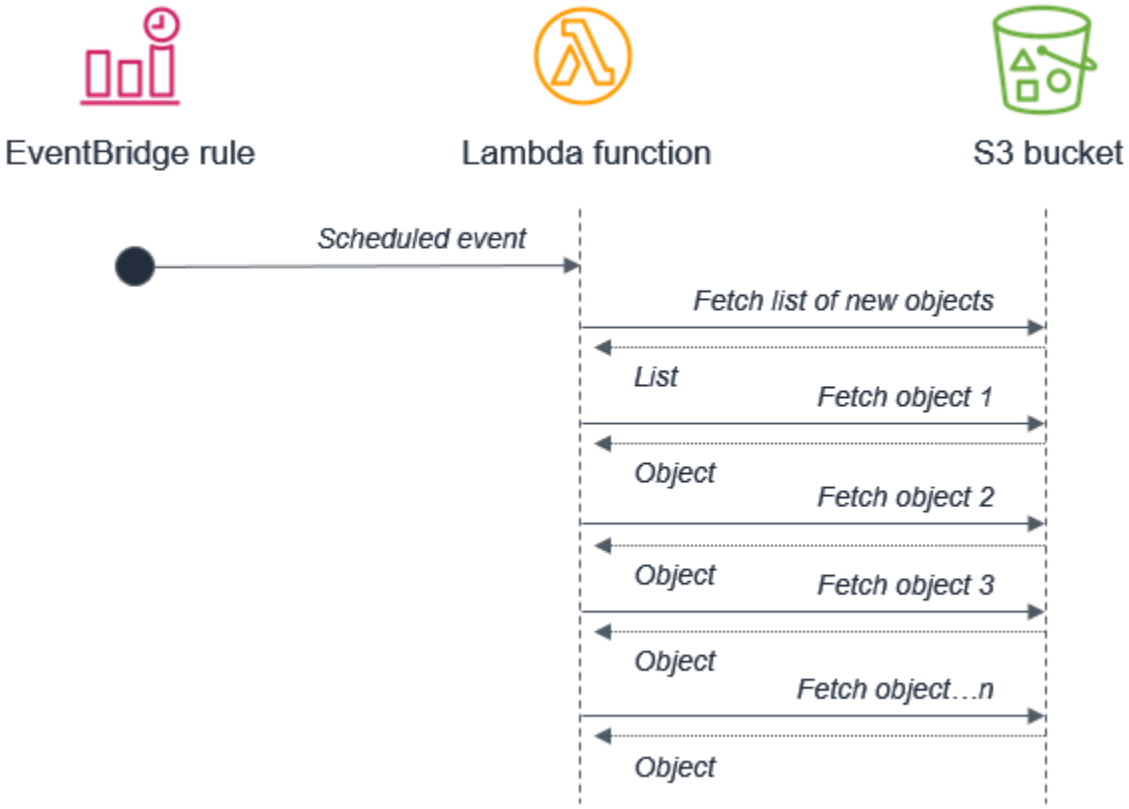
針對隨選資料而非批次處理進行建置

許多傳統系統旨在定期執行並處理隨著時間累積的交易批次。例如，銀行應用程式可能每小時執行一次，以處理中央分類帳中的 ATM 交易。在 Lambda 型應用程式中，每個事件都應觸發自訂處理，從而允許服務視需要向上擴展並行，以提供近乎即時的交易處理。

雖然標準 Lambda 函數的執行時間限制為 15 分鐘，但耐用函數最多可執行一年，因此適合長時間執行的處理需求。不過，您仍然應該盡可能偏好事件驅動型處理，而不是批次處理。

雖然您可以透過使用 Amazon EventBridge 中規則的排程表達式，在無伺服器應用程式中執行 cron 任務，但應謹慎使用或作為最後的手段。在任何採用批次處理模式的排程任務中，交易量都有可能增長到超出 Lambda 15 分鐘時間限制內可以處理的數量。如果外部系統的限制迫使您使用排程器，您通常應排程最短的合理重複時段。

例如，使用批次程序觸發 Lambda 函式來擷取新的 Amazon S3 物件清單並非最佳實務。這是因為服務在批次之間接收到的新物件可能多於 15 分鐘 Lambda 函數內可以處理的物件。



相反，Amazon S3 儲存貯體應在每次有新物件放入其中時都調用 Lambda 函式。此方法的可擴展性顯著提高，並且可以近乎即時地進行處理。

選擇複雜工作流程的協同運作選項

涉及分支邏輯、不同類型故障模型與重試邏輯的工作流程，通常會使用協調器來追蹤整體執行的狀態。請勿在標準 Lambda 函數中建置臨機操作協同運作。這會導致緊密耦合、複雜的路由代碼，而且沒有自動狀態復原。

請改用下列其中一個專用協同運作選項：

- Lambda 耐久函數：以應用程式為中心的協同運作，使用標準程式設計語言搭配自動檢查點、內建重試和執行復原。非常適合偏好將工作流程邏輯與 Lambda 內的商業邏輯一起保存在程式碼中的開發人員。

- AWS Step Functions：視覺化工作流程協同運作與 220 多種 AWS 服務的原生整合。非常適合用於多服務協調、零維護基礎設施和視覺化工作流程設計。

如需在這些選項之間進行選擇的指引，請參閱耐用函數或 Step Functions。

透過 Step Functions，您可以使用狀態機器來管理協同運作。這會從您的程式碼擷取錯誤處理、路由和分支邏輯，並將其取代為使用 JSON 宣告的狀態機器。除了提升工作流程的穩健性與可觀測性之外，您還可以為工作流程新增版本控制功能，並將狀態機器轉化成可新增至程式碼儲存庫的編碼資源。

Lambda 函式中的簡易工作流程，往往會隨著時間推移逐漸變得更加複雜。操作生產無伺服器應用程式時，請務必識別何時發生這種情況，以便將此邏輯遷移至狀態機器或耐用函數。

實作冪等性

AWS 無伺服器服務，包括 Lambda，具有容錯能力，旨在處理故障。例如，若某項服務調用 Lambda 函式時發生服務中斷，Lambda 會自動在另一個可用區域中調用您的函式。如果您的函式擲回錯誤，Lambda 會重試此調用請求。

由於同一事件可能會被多次接收，因此函數應設計為等冪。這意味著多次接收同一事件不會在第一次收到事件之後變更結果。

您可在 Lambda 函式中實作冪等性：透過 DynamoDB 資料表追蹤近期處理過的識別碼，藉此判斷該交易先前是否已處理完畢。DynamoDB 資料表通常實作存留時間 (TTL) 值來使項目過期，以限制所使用的儲存空間。

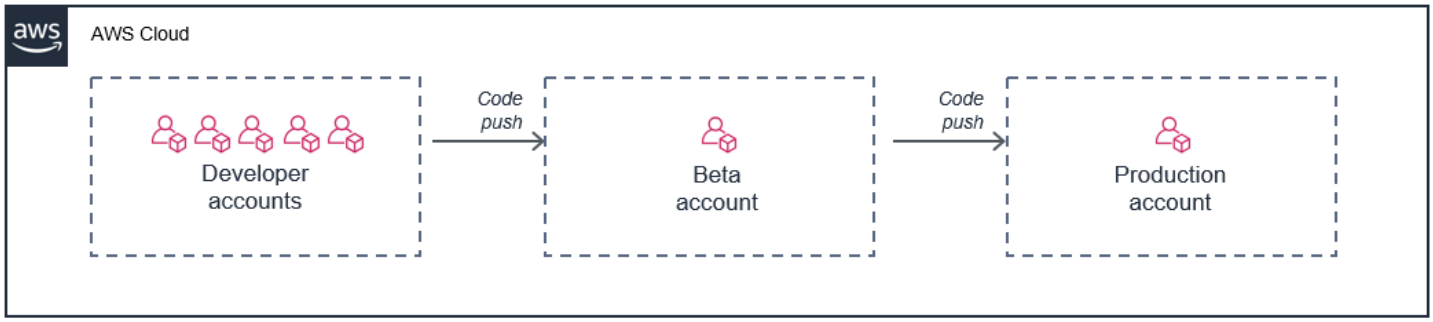
使用多個 AWS 帳戶來管理配額

中的許多服務配額 AWS 是在帳戶層級設定。這表示隨著您新增更多工作負載，可能會快速用盡配額上限。

解決此問題的有效方法是使用多個 AWS 帳戶，將每個工作負載分配到自己的 帳戶。這樣可防止與其他工作負載或非生產資源共用配額。

此外，您可以透過 AWS Organizations 集中管理這些帳戶的帳單、合規性與安全性。您可以將政策連接到帳戶群組，避免自訂指令碼和手動程序。

一種常見的方法是為每個開發人員提供 AWS 帳戶，然後針對 Beta 部署階段和生產環境使用不同的帳戶：



在此模型中，每位開發人員的帳戶都具備獨立的配額限制，因此其資源使用行為不會影響您的生產環境。此做法還能讓開發人員在本機開發環境中測試 Lambda 函式，並直接對接其個人帳戶中的真實雲端資源。

建立第一個 Lambda 函數

若要開始使用 Lambda，請使用 Lambda 主控台來建立函數。您可以在幾分鐘內建立和部署函數，並在主控台中加以測試。

進行教學課程時，您會學到一些基本 Lambda 概念，例如如何使用 Lambda「事件物件」，將引數傳遞給函數。您也會學到如何從函數傳回日誌輸出，以及如何在 Amazon CloudWatch Logs 中查看函數的調用日誌。

為了簡化，您可以使用 Python 或 Node.js 執行期建立函數。您可以使用這些轉譯語言，直接在主控台的內建程式碼編輯器中編輯函數程式碼。使用 Java 和 C# 等編譯語言，您必須在本機建置機器上建立部署套件，並將其上傳至 Lambda。若要了解如何使用其他執行期，將函數部署至 Lambda，請參閱the section called “後續步驟”一節中的連結。

> **Tip**
>
> 若要了解如何建置無伺服器解決方案，請參閱無伺服器開發人員指南。

先決條件

註冊 AWS 帳戶

如果您沒有 AWS 帳戶，請完成下列步驟來建立一個。

註冊 AWS 帳戶

1. 開啟 https://portal.aws.amazon.com/billing/signup。
2. 請遵循線上指示進行。

   部分註冊程序需接收來電或簡訊，並在電話鍵盤輸入驗證碼。

   當您註冊 時 AWS 帳戶，AWS 帳戶根使用者會建立 。根使用者有權存取該帳戶中的所有 AWS 服務 和資源。作為安全最佳實務，請將管理存取權指派給使用者，並且僅使用根使用者來執行需要根使用者存取權的任務。

AWS 會在註冊程序完成後傳送確認電子郵件給您。您可以隨時登錄 https://aws.amazon.com/ 並選擇我的帳戶，以檢視您目前的帳戶活動並管理帳戶。

建立具有管理存取權的使用者

註冊 後 AWS 帳戶，請保護 AWS 帳戶根使用者、啟用 AWS IAM Identity Center和建立管理使用者，以免將根使用者用於日常任務。

保護您的 AWS 帳戶根使用者

1. 選擇根使用者並輸入 AWS 帳戶 您的電子郵件地址，以帳戶擁有者AWS 管理主控台身分登入 。在下一頁中，輸入您的密碼。

   如需使用根使用者登入的說明，請參閱 AWS 登入 使用者指南中的以根使用者身分登入。

2. 若要在您的根使用者帳戶上啟用多重要素驗證 (MFA)。

   如需說明，請參閱《IAM 使用者指南》中的為您的 AWS 帳戶 根使用者 （主控台） 啟用虛擬 MFA 裝置。

建立具有管理存取權的使用者

1. 啟用 IAM Identity Center。

   如需指示，請參閱《AWS IAM Identity Center 使用者指南》中的啟用 AWS IAM Identity Center。

2. 在 IAM Identity Center 中，將管理存取權授予使用者。

   如需使用 IAM Identity Center 目錄 做為身分來源的教學課程，請參閱AWS IAM Identity Center 《 使用者指南》中的使用預設值設定使用者存取 IAM Identity Center 目錄。

以具有管理存取權的使用者身分登入

- 若要使用您的 IAM Identity Center 使用者簽署，請使用建立 IAM Identity Center 使用者時傳送至您電子郵件地址的簽署 URL。

  如需使用 IAM Identity Center 使用者登入的說明，請參閱AWS 登入 《 使用者指南》中的登入 AWS 存取入口網站。

指派存取權給其他使用者

1. 在 IAM Identity Center 中，建立一個許可集來遵循套用最低權限的最佳實務。

   如需指示，請參閱《AWS IAM Identity Center 使用者指南》中的建立許可集。

2. 將使用者指派至群組，然後對該群組指派單一登入存取權。

   如需指示，請參閱《AWS IAM Identity Center 使用者指南》中的新增群組。

使用主控台建立一個 Lambda 函數

在此範例中，您的函數會使用 JSON 物件，其中包含兩個標示 `"length"` 和 `"width"` 的整數值。函數會將這些值相乘以計算區域，並以 JSON 字串傳回。

您的函數也會列印計算出的區域，以及其 CloudWatch 日誌群組的名稱。稍後在本教學課程中，您會學到如何使用 CloudWatch Logs 查看函數調用的記錄。

若要使用主控台建立 Hello world Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇建立函數。
3. 選取從頭開始撰寫。
4. 在基本資訊窗格中，為函數名稱輸入 **`myLambdaFunction`**。
5. 針對執行期，選擇 Node.js 24 或 Python 3.14。
6. 將架構設定為 x86_64，然後選擇建立函數。

除了可傳回 `Hello from Lambda!` 訊息的簡單函數之外，Lambda 還可為函數建立執行角色。執行角色是授予 Lambda 函數存取 AWS 服務 和資源許可的 AWS Identity and Access Management (IAM) 角色。對於您的函數，Lambda 建立的角色會授予寫入 CloudWatch Logs 的基本許可。

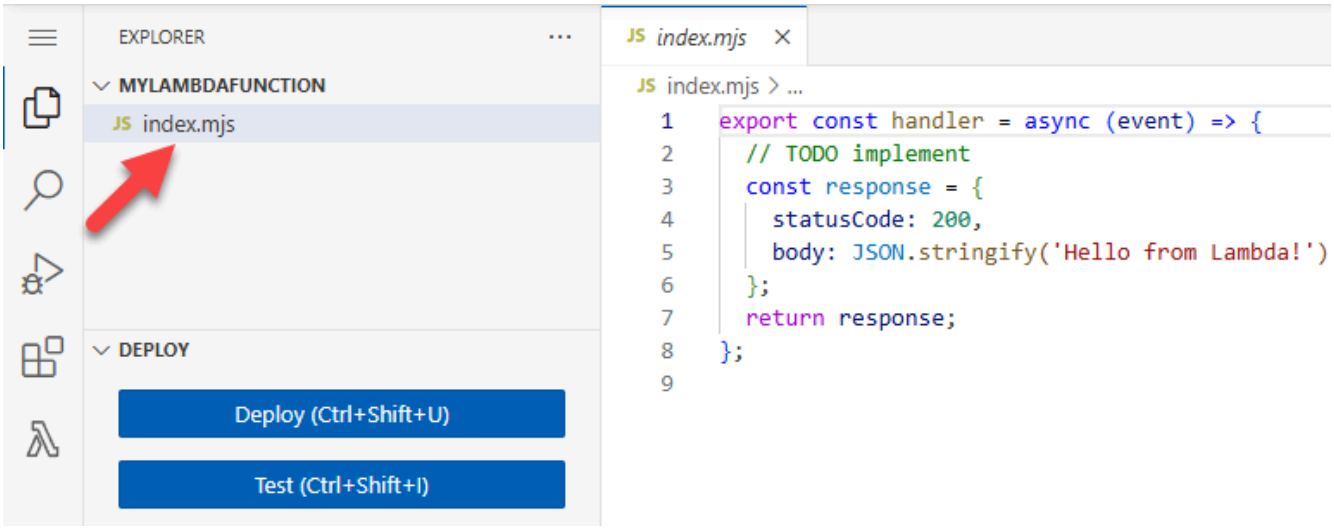
使用主控台的內建程式碼編輯器，將 Lambda 建立的 Hello world 程式碼取代為您自己的函數程式碼。

Node.js

若要在主控台中修改程式碼

1. 選擇 程式碼 標籤。

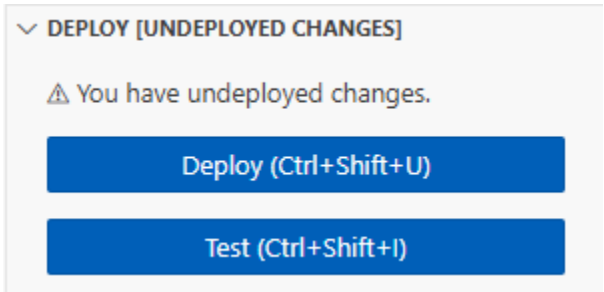
   在主控台的內建程式碼編輯器中，您應該會看到 Lambda 建立的函數程式碼。如果您在程式碼編輯器中沒看到 index.mjs 標籤，請在檔案總管中選取 index.mjs，如下圖所示。



2. 將以下程式碼貼到 index.mjs 標籤中，替換 Lambda 建立的程式碼。

```
export const handler = async (event, context) => {

  const length = event.length;
  const width = event.width;
  let area = calculateArea(length, width);
  console.log(`The area is ${area}`);

  console.log('CloudWatch log group: ', context.logGroupName);

  let data = {
    "area": area,
  };
    return JSON.stringify(data);

  function calculateArea(length, width) {
    return length * width;
  }
};
```

3. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



了解函數程式碼

在進行下一步之前，讓我們花一點時間看看函數程式碼，並了解一些重要 Lambda 概念。

- Lambda 處理常式：

  您的 Lambda 函數包含 Node.js 函數 `handler`。以 Node.js 編寫的 Lambda 函數可以包含多個 Node.js 函數，但「handler」函數始終是程式碼的進入點。當有人調用您的函數時，Lambda 會執行此方法。

  使用主控台建立 Hello world 函數時，Lambda 會自動將函數的處理常式方法名稱設定為 `handler`。請勿編輯此 Node.js 函數的名稱。如果這麼做，調用函數時，Lambda 將無法執行您的程式碼。

  若要進一步了解以 Node.js 編寫的 Lambda 處理常式，請參閱 the section called “處理常式”。

- Lambda 事件物件：

  `handler` 函數會使用兩個引數，`event` 和 `context`。Lambda 中的「事件」是一種 JSON 格式的文件，它包含供函數處理的資料。

  如果您的函數是由另一個函數叫用 AWS 服務，則事件物件會包含導致叫用之事件的相關資訊。舉例來說，如果在物件上傳到 Amazon Simple Storage Service (Amazon S3) 儲存貯體時調用函數，則該事件會包含儲存貯體的名稱和物件索引鍵。

  在此範例中，您會進入有兩個索引鍵/值組的 JSON 格式文件，在主控台中建立事件。

- Lambda 內容物件：

  函數使用的第二個引數為 `context`。Lambda 會自動將「內容物件」傳遞至您的函數。內容物件包含有關函數調用及執行環境的資訊。

  您可以使用內容物件，基於監控目的，輸出函數調用的資訊。在此範例中，您的函數使用 `logGroupName` 參數，輸出其 CloudWatch 日誌群組的名稱。

  若要進一步了解以 Node.js 編寫的 Lambda 內容物件，請參閱 the section called “Context”。

- 在 Lambda 中記錄：

  您可以透過 Node.js，使用 `console.log` 和 `console.error` 等主控台方法，將資訊傳送到函數的日誌。範例程式碼使用 `console.log` 陳述式，輸出計算出的區域及函數的 CloudWatch Logs 群組名稱。您也可以使用任何寫入 `stdout` 或 `stderr` 的記錄程式庫。

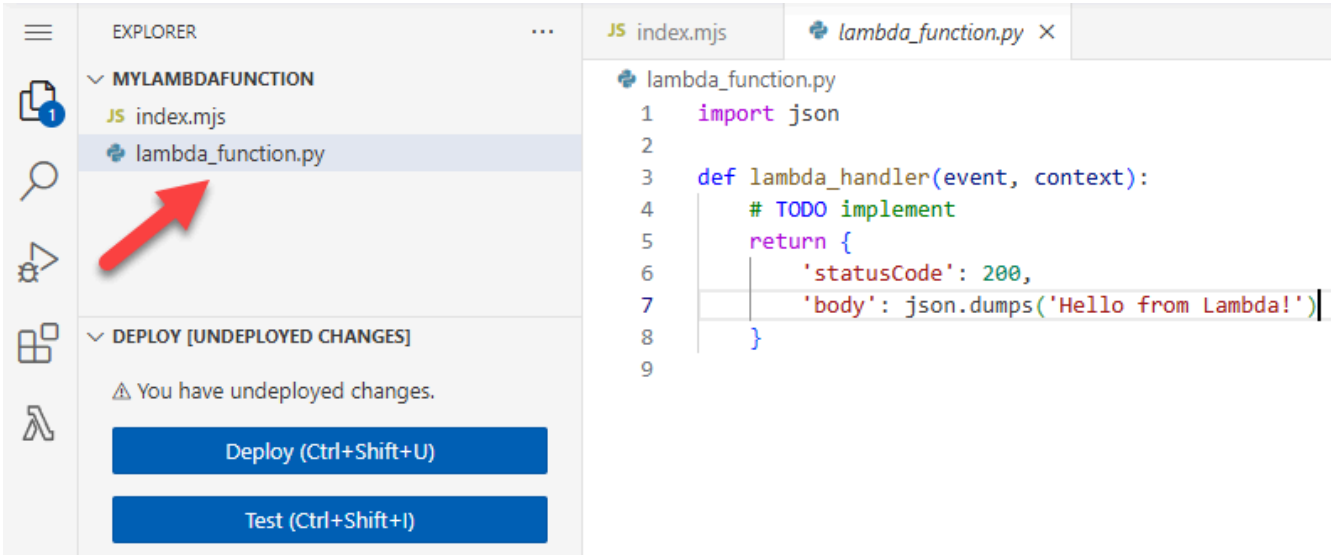
如需詳細資訊，請參閱 the section called “日誌”。若要了解如何在其他執行期中記錄，請參閱您有興趣之執行期的「建置方式」頁面。

Python

若要在主控台中修改程式碼

1. 選擇 程式碼 標籤。

   在主控台的內建程式碼編輯器中，您應該會看到 Lambda 建立的函數程式碼。如果您在程式碼編輯器中沒看到 lambda_function.py，請在檔案總管中選取 lambda_function.py，如下圖所示。

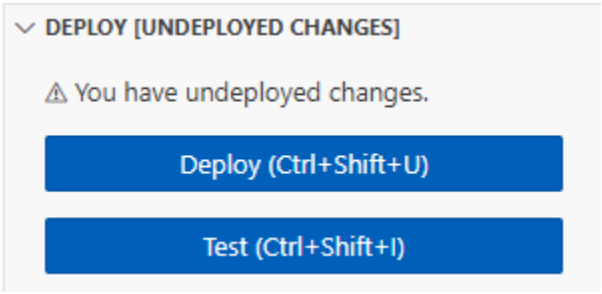


2. 將以下程式碼貼到 lambda_function.py 標籤中，替換 Lambda 建立的程式碼。

```
import json
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):

    # Get the length and width parameters from the event object. The
    # runtime converts the event object to a Python dictionary
    length = event['length']
    width = event['width']
```

```
    area = calculate_area(length, width)
    print(f"The area is {area}")

    logger.info(f"CloudWatch logs group: {context.log_group_name}")

    # return the calculated area as a JSON string
    data = {"area": area}
    return json.dumps(data)

def calculate_area(length, width):
    return length*width
```

3. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



了解函數程式碼

在進行下一步之前，讓我們花一點時間看看函數程式碼，並了解一些重要 Lambda 概念。

- Lambda 處理常式：

  Lambda 函數包含 Python 函數 `lambda_handler`。以 Python 編寫的 Lambda 函數可以包含多個 Python 函數，但「handler」函數始終是程式碼的進入點。當有人調用您的函數時，Lambda 會執行此方法。

  使用主控台建立 Hello world 函數時，Lambda 會自動將函數的處理常式方法名稱設定為 `lambda_handler`。請勿編輯此 Python 函數的名稱。如果這麼做，調用函數時，Lambda 將無法執行您的程式碼。

  若要進一步了解以 Python 編寫的 Lambda 處理常式，請參閱the section called “處理常式”。

- Lambda 事件物件：

  `lambda_handler` 函數會使用兩個引數，`event` 和 `context`。Lambda 中的「事件」是一種 JSON 格式的文件，它包含供函數處理的資料。

如果您的函數是由另一個函數叫用 AWS 服務，則事件物件會包含導致叫用之事件的相關資訊。舉例來說，如果在物件上傳到 Amazon Simple Storage Service (Amazon S3) 儲存貯體時調用函數，則該事件會包含儲存貯體的名稱和物件索引鍵。

在此範例中，您會進入有兩個索引鍵/值組的 JSON 格式文件，在主控台中建立事件。

- Lambda 內容物件：

  函數使用的第二個引數為 `context`。Lambda 會自動將「內容物件」傳遞至您的函數。內容物件包含有關函數調用及執行環境的資訊。

  您可以使用內容物件，基於監控目的，輸出函數調用的資訊。在此範例中，您的函數使用 `log_group_name` 參數，輸出其 CloudWatch 日誌群組的名稱。

  若要進一步了解以 Python 編寫的 Lambda 內容物件，請參閱the section called “Context”。

- 在 Lambda 中記錄：

  透過 Python，您可以使用 `print` 陳述式或 Python 記錄程式庫，將資訊傳送到函數的日誌。為了說明擷取內容的差異，範例程式碼會使用這兩種方法。在生產應用程式中，建議您使用記錄程式庫。

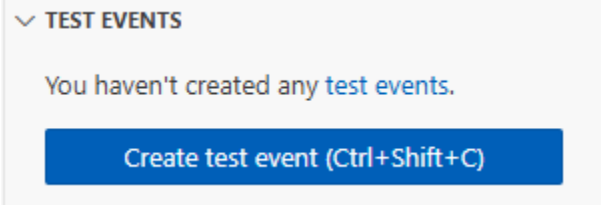
  如需詳細資訊，請參閱 the section called “日誌”。若要了解如何在其他執行期中記錄，請參閱您有興趣之執行期的「建置方式」頁面。

使用主控台程式碼編輯器調用 Lambda 函數

若要使用 Lambda 主控台程式碼編輯器來調用函數，請先建立要傳送至函數的測試事件。事件是一種 JSON 格式文件，其中包含兩個索引鍵/值組，索引鍵 `"length"` 和 `"width"`。

若要建立測試事件

1. 在主控台程式碼編輯器的 TEST EVENTS 區段中，選擇建立測試事件。



2. Event Name (事件名稱) 輸入 **`myTestEvent`**。

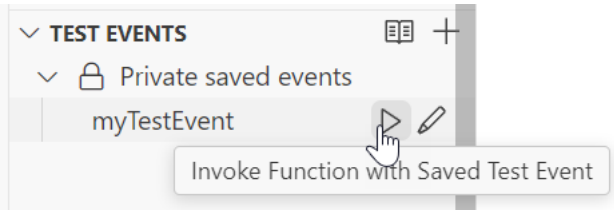
3. 在事件 JSON 區段中，將預設 JSON 取代為下列項目：

```
{
  "length": 6,
  "width": 7
}
```

4. 選擇儲存。

若要測試函數並檢視調用記錄

在主控台程式碼編輯器的 TEST EVENTS 區段中，選擇測試事件旁邊的執行圖示：



當函數完成執行時，回應和函數日誌會顯示在 OUTPUT 索引標籤中。您應該會看到類似下列的結果：

Node.js

```
Status: Succeeded
Test Event Name: myTestEvent

Response
"{\"area\":42}"

Function Logs
START RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Version: $LATEST
2024-08-31T23:39:45.313Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO The area is 42
2024-08-31T23:39:45.331Z 5c012b0a-18f7-4805-b2f6-40912935034a INFO CloudWatch log
 group:  /aws/lambda/myLambdaFunction
END RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a
REPORT RequestId: 5c012b0a-18f7-4805-b2f6-40912935034a Duration: 20.67 ms Billed
 Duration: 21 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration: 163.87 ms

Request ID
5c012b0a-18f7-4805-b2f6-40912935034a
```

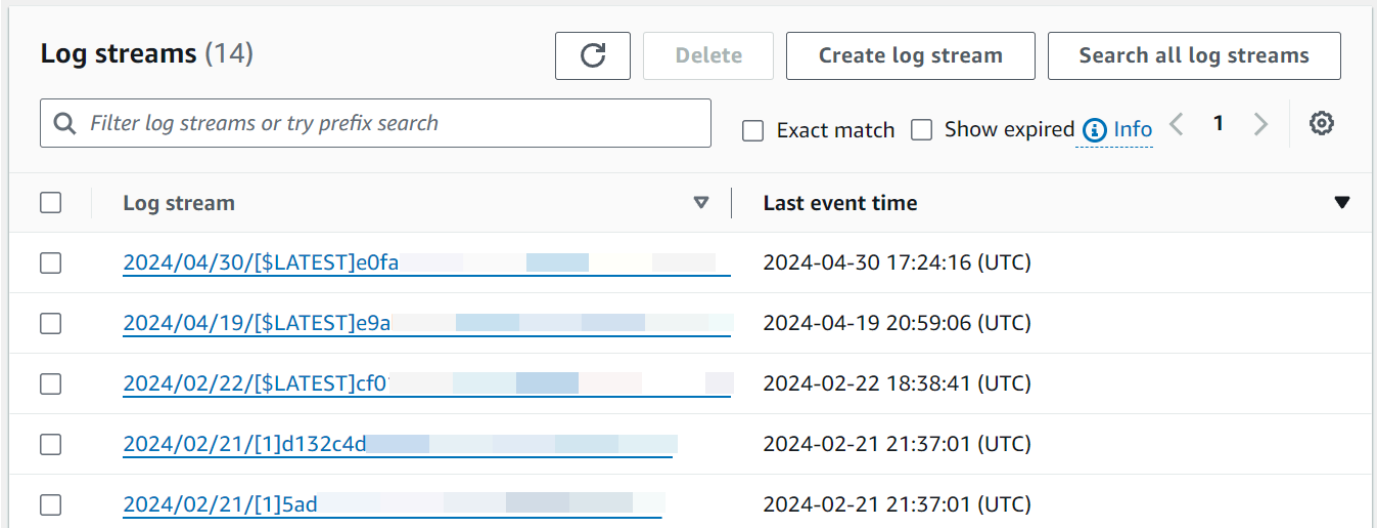
Python

```
Status: Succeeded
Test Event Name: myTestEvent

Response
"{\"area\": 42}"

Function Logs
START RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Version: $LATEST
The area is 42
[INFO] 2024-08-31T23:43:26.428Z 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b CloudWatch logs
 group: /aws/lambda/myLambdaFunction
END RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
REPORT RequestId: 2d0b1579-46fb-4bf7-a6e1-8e08840eae5b Duration: 1.42 ms Billed
 Duration: 2 ms Memory Size: 128 MB Max Memory Used: 39 MB Init Duration: 123.74 ms

Request ID
2d0b1579-46fb-4bf7-a6e1-8e08840eae5b
```

當您在 Lambda 主控台之外調用函數時，必須使用 CloudWatch Logs 來檢視函數的執行結果。

若要在 CloudWatch Logs 中檢視函數的調用記錄

1. 開啟 CloudWatch 主控台的 日誌群組 頁面。
2. 為函數 (`/aws/lambda/myLambdaFunction`) 選擇日誌群組名稱。這是您的函數列印至主控台的日誌群組名稱。
3. 向下捲動並選擇要查看的函數調用日誌串流。



您應該會看到類似下列的輸出：

Node.js

```
INIT_START Runtime Version: nodejs:22.v13    Runtime Version ARN:
 arn:aws:lambda:us-
west-2::runtime:e3aaabf6b92ef8755eaae2f4bfdcb7eb8c4536a5e044900570a42bdba7b869d9
START RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20 Version: $LATEST
2024-08-23T22:04:15.809Z    5c012b0a-18f7-4805-b2f6-40912935034a  INFO The area
 is 42
2024-08-23T22:04:15.810Z    aba6c0fc-cf99-49d7-a77d-26d805dacd20  INFO
 CloudWatch log group:  /aws/lambda/myLambdaFunction
END RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20
REPORT RequestId: aba6c0fc-cf99-49d7-a77d-26d805dacd20    Duration: 17.77 ms
 Billed Duration: 18 ms    Memory Size: 128 MB    Max Memory Used: 67 MB    Init
 Duration: 178.85 ms
```

Python

```
INIT_START Runtime Version: python:3.13.v16    Runtime Version ARN:
 arn:aws:lambda:us-
west-2::runtime:ca202755c87b9ec2b58856efb7374b4f7b655a0ea3deb1d5acc9aee9e297b072
START RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e Version: $LATEST
The area is 42
[INFO] 2024-09-01T00:05:22.464Z 9315ab6b-354a-486e-884a-2fb2972b7d84 CloudWatch
 logs group: /aws/lambda/myLambdaFunction
END RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e
```

```
REPORT RequestId: 9d4096ee-acb3-4c25-be10-8a210f0a9d8e    Duration: 1.15 ms
 Billed Duration: 2 ms    Memory Size: 128 MB    Max Memory Used: 40 MB
```

清除

使用完範例函數時，請加以刪除。您還可以刪除存放函數日誌的日誌群組，以及主控台建立的執行角色。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **confirm**，然後選擇刪除。

刪除日誌群組

1. 開啟 CloudWatch 主控台的 日誌群組 頁面。
2. 選取函數的日誌群組 (`/aws/lambda/myLambdaFunction`)。
3. 選擇 動作 、刪除日誌群組 。
4. 在 刪除日誌群組 對話方塊中，選擇 刪除 。

刪除執行角色

1. 開啟 AWS Identity and Access Management (IAM) 主控台的角色頁面。
2. 選取函數的執行角色，(例如 `myLambdaFunction-role-31exxmpl`)。
3. 選擇 刪除。
4. 在刪除角色對話方塊中輸入角色名稱，然後選擇刪除。

其他資源和後續步驟

既然您已使用主控台建立並測試簡單的 Lambda 函數，接著請採取下列後續步驟：

- 了解如何將相依項新增至函數，並使用 .zip 部署套件加以部署。從以下連結中選擇您偏好的語言。

Node.js

the section called “部署 .zip 封存檔”

Typescript

the section called “部署 .zip 封存檔”

Python

the section called “部署 .zip 封存檔”

Ruby

the section called “部署 .zip 封存檔”

Java

the section called “部署 .zip 封存檔”

Go

the section called “部署 .zip 封存檔”

C#

the section called “部署套件”

- 若要了解如何使用另一個函數叫用 Lambda 函數 AWS 服務，請參閱 教學課程：使用 Amazon S3 觸發條件調用 Lambda 函數。
- 選擇下列任一教學課程，獲得搭配使用 Lambda 與其他 AWS 服務的更複雜範例。
  - 教學課程：搭配使用 Lambda 與 API Gateway：建立用於調用 Lambda 函數的 Amazon API Gateway REST API。
  - 使用 Lambda 函數存取 Amazon RDS 資料庫：使用 Lambda 函數，透過 RDS 代理將資料寫入 Amazon Relational Database Service (Amazon RDS) 資料庫。
  - 使用 Amazon S3 觸發條件建立縮圖影像：每次將影像檔案上傳到 Amazon S3 儲存貯體時，使用 Lambda 函數建立縮圖。

透過範例應用程式與模式快速入門

下列資源可用於快速建立並部署能實作部分常見 Lambda 使用案例的無伺服器應用程式。對於每個範例應用程式，我們提供使用 手動建立和設定資源的說明 AWS 管理主控台，或使用 AWS Serverless Application Model 來使用 IaC 部署資源。遵循主控台說明，進一步了解如何為每個應用程式設定個別 AWS 資源，或使用 AWS SAM 快速部署資源，就像您在生產環境中一樣。

檔案處理

- PDF 加密應用程式：建立無伺服器應用程式，能在 PDF 檔案上傳至 Amazon Simple Storage Service 儲存貯體時進行加密，並將加密後的檔案儲存至另一個儲存貯體。此類應用程式對於在上傳檔案時保護敏感文件非常有用。

- 映像分析應用程式：建立無伺服器應用程式，能透過 Amazon Rekognition 從映像中擷取文字。此類應用程式對於文件處理、內容審核與自動化映像分析非常有用。

資料庫整合

- 佇列到資料庫應用程式：建立無伺服器應用程式，能將佇列訊息寫入 Amazon RDS 資料庫。此類應用程式對於處理使用者註冊與處理訂單提交非常有用。

- 資料庫事件處理常式：建立無伺服器應用程式，能回應 Amazon DynamoDB 資料表變更。此類應用程式對於稽核日誌、資料複寫與工作流程自動化非常有用。

排程任務

- 資料庫維護應用程式：建立無伺服器應用程式，能使用 Cron 排程自動刪除 Amazon DynamoDB 資料表中超過 12 個月的項目。此類應用程式對於自動化資料庫維護與資料生命週期管理非常有用。
- 為 Lambda 函式建立 EventBridge 排程規則：在 EventBridge 中為規則套用排程表達式，依據定時排程觸發 Lambda 函式。此格式使用 cron 語法，並且可以設定一分鐘精細度。

長時間執行的工作流程

- 訂單處理應用程式：使用處理複雜訂單履行的耐用函數建立無伺服器應用程式，包括付款處理、庫存檢查和運送協調。此範例示範如何建置可在維護狀態的同時長時間執行的工作流程。

其他資源

使用下列資源進一步探索 Lambda 與無伺服器應用程式部署：

- Serverless Land：用於建構無伺服器應用程式的現成模式庫。它可協助開發人員使用 Lambda、API Gateway 和 EventBridge 等 AWS 服務更快地建立應用程式。該網站提供預先建置的解決方案與最佳實務，讓開發無伺服器系統變得更加輕鬆。
- Lambda 範例應用程式：本指南 GitHub 儲存庫中提供的應用程式。這些範例示範使用各種語言 AWS 和服務。每個範例應用程式都包含可用於輕鬆部署和清理的指令碼和支援資源。
- Lambda AWS SDKs的程式碼範例：示範如何搭配 AWS 軟體開發套件 (SDKs) 使用 Lambda 的範例。這些範例包括基本概念、動作、案例和 AWS 社群貢獻。範例涵蓋基本操作、個別服務函數，以及使用多個函數 AWS 或服務的特定任務。

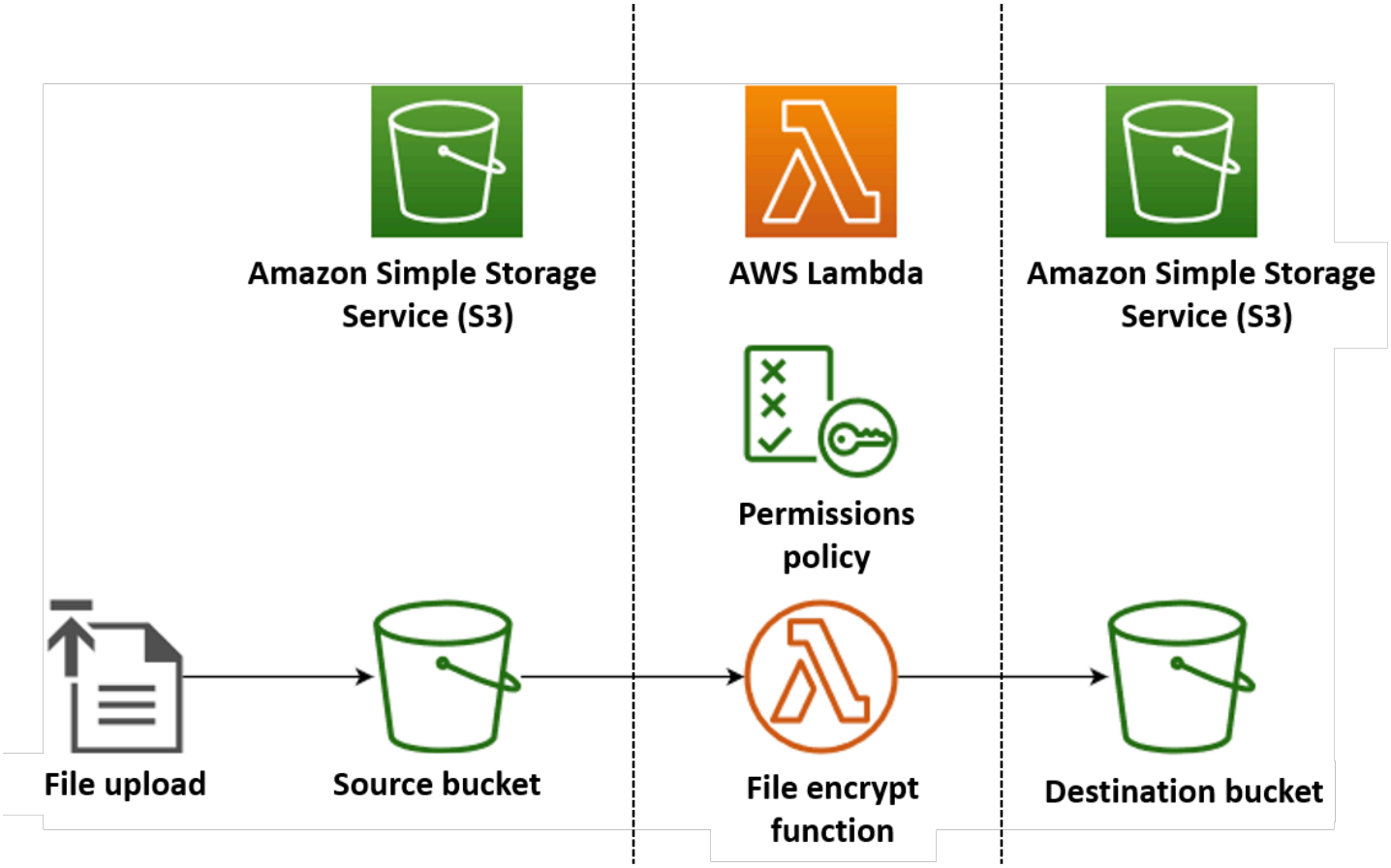
建立無伺服器檔案處理應用程式

Lambda 最常見的使用案例之一是執行檔案處理任務。例如，您可以使用 Lambda 函數從 HTML 檔案或影像自動建立 PDF 檔案，或在使用者上傳影像時建立縮圖。

在此範例中，您會建立這樣的應用程式：當 PDF 檔案上傳至 Amazon Simple Storage Service (Amazon S3) 儲存貯體時，該應用程式會自動加密 PDF 檔案。若要實作此應用程式，您需要建立下列資源：

- 供使用者上傳 PDF 檔案 S3 儲存貯體
- 使用 Python 編寫的 Lambda 函數，用於讀取上傳的檔案，並建立其受密碼保護的加密版本
- 供 Lambda 儲存加密檔案的第二個 S3 儲存貯體

您也可以建立 AWS Identity and Access Management (IAM) 政策，授予 Lambda 函數在 S3 儲存貯體上執行讀取和寫入操作的許可。



> **Tip**
>
> 如果您是 Lambda 新手，建議在建立此範例應用程式之前，先學習「建立第一個函數」教學課程。

您可以使用 AWS 管理主控台 或 AWS Command Line Interface () 建立和設定資源，以手動部署您的應用程式AWS CLI。您也可以使用 AWS Serverless Application Model (AWS SAM) 來部署應用程式。AWS SAM 是基礎設施即程式碼 (IaC) 工具。藉助 IaC，您無需手動建立資源，而是在程式碼中定義資源，然後便可自動部署資源。

如果您想要在部署此範例應用程式之前，進一步了解如何將 Lambda 與 IaC 搭配使用，請參閱the section called “基礎設施即程式碼 (IaC)”。

建立 Lambda 函式原始程式碼檔案

在專案目錄中建立下列檔案：

- `lambda_function.py`：執行檔案加密之 Lambda 函數的 Python 函數程式碼
- `requirements.txt`：資訊清單檔案，用於定義 Python 函數程式碼所需的相依項

展開下列各區段即可檢視程式碼，並進一步了解每個檔案的作用。若要在本機電腦上建立檔案，請複製並貼上下面的程式碼，或從 aws-lambda-developer-guide GitHub 儲存庫下載檔案。

Python 函數程式碼

複製以下程式碼並貼到名稱為 `lambda_function.py` 的檔案中。

```
from pypdf import PdfReader, PdfWriter
import uuid
import os
from urllib.parse import unquote_plus
import boto3

# Create the S3 client to download and upload objects from S3
s3_client = boto3.client('s3')

def lambda_handler(event, context):
    # Iterate over the S3 event object and get the key for all uploaded files
    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        key = unquote_plus(record['s3']['object']['key']) # Decode the S3 object key to
 remove any URL-encoded characters
        download_path = f'/tmp/{uuid.uuid4()}.pdf' # Create a path in the Lambda tmp
 directory to save the file to
        upload_path = f'/tmp/converted-{uuid.uuid4()}.pdf' # Create another path to
 save the encrypted file to

        # If the file is a PDF, encrypt it and upload it to the destination S3 bucket
        if key.lower().endswith('.pdf'):
            s3_client.download_file(bucket, key, download_path)
            encrypt_pdf(download_path, upload_path)
            encrypted_key = add_encrypted_suffix(key)
            s3_client.upload_file(upload_path, f'{bucket}-encrypted', encrypted_key)

# Define the function to encrypt the PDF file with a password
def encrypt_pdf(file_path, encrypted_file_path):
    reader = PdfReader(file_path)
    writer = PdfWriter()
```

```
    for page in reader.pages:
        writer.add_page(page)

    # Add a password to the new PDF
    writer.encrypt("my-secret-password")

    # Save the new PDF to a file
    with open(encrypted_file_path, "wb") as file:
        writer.write(file)

# Define a function to add a suffix to the original filename after encryption
def add_encrypted_suffix(original_key):
    filename, extension = original_key.rsplit('.', 1)
    return f'{filename}_encrypted.{extension}'
```

> **Note**
>
> 在此範例程式碼中，用於加密檔案的密碼 (`my-secret-password`) 寫死到函數程式碼中。但在生產應用程式中，請勿在函數程式碼中包含密碼等敏感資訊。反之，請建立 AWS Secrets Manager 秘密，然後使用 AWS 參數和秘密 Lambda 延伸來擷取 Lambda 函數中的登入資料。

Python 函數程式碼包含三個函數：調用 Lambda 函數時執行的處理常式函數，以及該處理常式調用以執行 PDF 加密的其他兩個函數，分別為 `encrypt_pdf` 和 `add_encrypted_suffix`。

當該函數被 Amazon S3 調用時，Lambda 會將 JSON 格式的事件引數傳遞給該函數，其中包含導致調用的事件詳細資訊。在此例中，這些資訊包括 S3 儲存貯體的名稱和上傳檔案的物件索引鍵。若要進一步了解 Amazon S3 的事件物件格式，請參閱the section called “S3”。

然後，您的函數會使用 適用於 Python (Boto3) 的 AWS SDK 將事件物件中指定的 PDF 檔案下載至其本機暫時儲存目錄，再使用 pypdf程式庫進行加密。

最後，該函數會使用 Boto3 SDK 將經過加密的檔案存放在 S3 目的地儲存貯體中。

`requirements.txt` 資訊清單檔案

複製以下程式碼並貼到名稱為 `requirements.txt` 的檔案中。

```
boto3
pypdf
```

在此範例中，函數程式碼只包含兩個不屬於標準 Python 程式庫的相依項，即適用於 Python 的 SDK (Boto3) 和函數用來執行 PDF 加密的 `pypdf` 套件。

> **Note**
>
> Lambda 執行時期包含適用於 Python 的 SDK (Boto3) 版本，因此程式碼無需將 Boto3 新增至函數的部署套件即可執行。不過，為了維持對函數相依項的完整控制，並避免版本不一致可能造成的問題，Python 的最佳實務是在函數的部署套件中包含所有函數相依項。如需進一步了解，請參閱the section called “Python 中的執行期相依項”。

部署應用程式

您可以手動或使用 來建立和部署此範例應用程式的資源 AWS SAM。在生產環境中，我們建議您使用類似 的 IaC 工具 AWS SAM ，快速且重複地部署整個無伺服器應用程式，而無需使用手動程序。

手動部署資源

若要手動部署應用程式：

- 建立來源和目的地 Amazon S3 儲存貯體
- 建立一個 Lambda 函數來加密 PDF 檔案，並將經過加密的版本儲存至 S3 儲存貯體
- 設定一個 Lambda 觸發條件，在物件上傳至來源儲存貯體時調用函數

在開始之前，先確保建置機器上已安裝 Python。

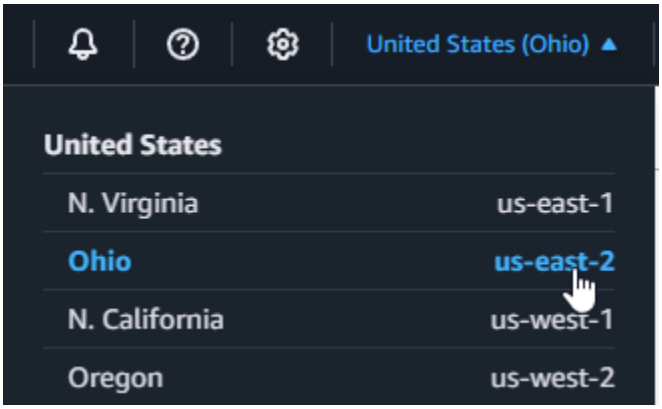
建立兩個 S3 儲存貯體

首先建立兩個 S3 儲存貯體。第一個儲存貯體是將接收 PDF 檔案上傳的來源儲存貯體。當您調用函數時，Lambda 會使用第二個儲存貯體來儲存經過加密的檔案。

Console

若要建立 S3 儲存貯體 (主控台)

1. 開啟 Amazon S3 主控台的一般用途儲存貯體頁面。
2. 選取最 AWS 區域 接近您地理位置的 。可使用螢幕頂端的下拉式清單來變更區域。



3. 選擇建立儲存貯體。
4. 在 General configuration (一般組態) 下，執行下列動作：
   a. 在儲存貯體類型欄位中，選取一般用途。
   b. 在儲存貯體名稱欄位中，輸入符合 Amazon S3 儲存貯體命名規則的全域唯一名稱。儲存貯體名稱只能包含小寫字母、數字、句點 (.) 和連字號 (-)。
5. 其他所有選項維持設為預設值，然後選擇建立儲存貯體。
6. 重複步驟 1 到 4 以建立目的地儲存貯體。對於儲存貯體名稱，輸入 **amzn-s3-demo-bucket-encrypted**，其中 **amzn-s3-demo-bucket** 是您剛才建立的來源儲存貯體名稱。

AWS CLI

在開始之前，先確保建置機器上已安裝AWS CLI。

建立 Amazon S3 儲存貯體 (AWS CLI)

1. 執行下列 CLI 命令以建立來源儲存貯體。您為儲存貯體選擇的名稱必須具有全域唯一性，且遵循 Amazon S3 儲存貯體命名規則。名稱只能包含小寫字母、數字、句點 (.) 和連字號 (-)。對於 `region` 和 `LocationConstraint`，請選擇最接近您地理位置的 AWS 區域。

```
aws s3api create-bucket --bucket amzn-s3-demo-bucket --region us-east-2 \
--create-bucket-configuration LocationConstraint=us-east-2
```

   稍後在教學課程中，您必須在與來源儲存貯體 AWS 區域 相同的 中建立 Lambda 函數，因此請記下您選擇的區域。
2. 執行下列命令以建立目的地儲存貯體。對於儲存貯體名稱，必須使用 **amzn-s3-demo-bucket-encrypted**，其中 **amzn-s3-demo-bucket** 是您在步驟 1 中建立的來源儲存貯體

名稱。針對 region和 LocationConstraint，選擇 AWS 區域 您用來建立來源儲存貯體的相同 。

```
aws s3api create-bucket --bucket amzn-s3-demo-bucket-encrypted --region us-
east-2 \
--create-bucket-configuration LocationConstraint=us-east-2
```

建立執行角色

執行角色是授予 Lambda 函數存取 AWS 服務 和資源許可的 IAM 角色。若要向函數提供 Amazon S3 的讀取和寫入權限，請連接 AWS 受管政策 AmazonS3FullAccess。

Console

若要建立執行角色並連接 **AmazonS3FullAccess** 受管政策 (主控台)

1. 在 IAM 主控台中開啟 角色頁面。
2. 選擇建立角色。
3. 在信任的實體類型欄位中，選取 AWS 服務；在使用案例欄位中，選取 Lambda。
4. 選擇下一步。
5. 執行下列操作，新增 AmazonS3FullAccess 受管政策：
   a. 在許可政策區段，於搜尋列輸入 **AmazonS3FullAccess**。
   b. 勾選政策旁的核取方塊。
   c. 選擇下一步。
6. 在角色詳細資訊區段，於角色名稱欄位輸入 **LambdaS3Role**。
7. 選擇建立角色。

AWS CLI

若要建立執行角色並連接 **AmazonS3FullAccess** 受管政策 (AWS CLI)

1. 將下面的 JSON 儲存在名為 trust-policy.json 的檔案中。此信任政策允許 Lambda 透過授予服務主體呼叫 AWS Security Token Service (AWS STS) AssumeRole動作的許可來使用角色的lambda.amazonaws.com許可。

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

2. 在儲存 JSON 信任政策文件的目錄中，執行下列 CLI 命令，建立執行角色。

```
aws iam create-role --role-name LambdaS3Role --assume-role-policy-document
 file://trust-policy.json
```

3. 若要執行下列 CLI 命令，以連接 AmazonS3FullAccess 受管政策。

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::aws:policy/AmazonS3FullAccess
```

建立函數部署套件

要建立函數，需建立包含函數程式碼和其相依項的部署套件。對於此應用程式，函數程式碼使用單獨的程式庫進行 PDF 加密。

建立部署套件

1. 導覽至專案目錄，其中包含您先前從 GitHub 建立或下載的檔案 lambda_function.py 和 requirements.txt，然後建立新的目錄，並命名為 package。
2. 執行以下命令，在 package 目錄中安裝 requirements.txt 檔案中指定的相依項。

```
pip install -r requirements.txt --target ./package/
```

3. 建立包含應用程式程式碼及其相依項的 .zip 檔案。在 Linux 或 MacOS 中，使用命令列界面執行下列命令。

```
cd package
zip -r ../lambda_function.zip .
cd ..
zip lambda_function.zip lambda_function.py
```

在 Windows 中，使用您偏好的 zip 工具建立 `lambda_function.zip` 檔案。確保包含相依項的 `lambda_function.py` 檔案和資料夾全部都在 .zip 檔案的根目錄中。

您也可以使用 Python 虛擬環境建立部署套件。請參閱 使用 .zip 封存檔部署 Python Lambda 函數

建立 Lambda 函式

現在使用上一個步驟建立的部署套件來部署 Lambda 函數。

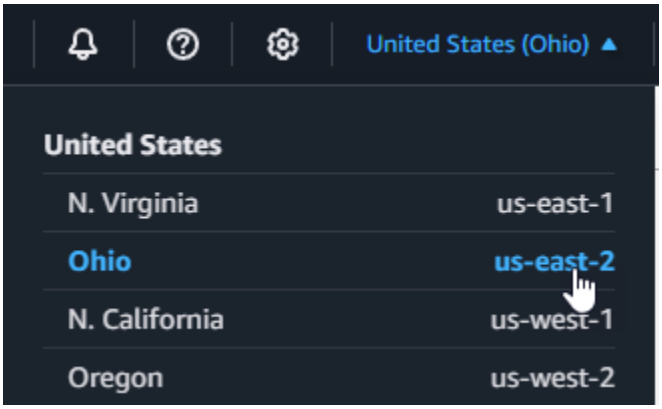
Console

建立函數的方式 (主控台)

要使用主控台建立 Lambda 函數，首先建立包含一些 'Hello world' 程式碼的基本函數。然後，透過上傳您在上一個步驟中建立的 .zip 檔案，將此程式碼取代為您自己的函數程式碼。

若要確保函數在加密大型 PDF 檔案時不會逾時，您需要設定函數的記憶體和逾時設定。另外，您還需要將函數的日誌格式設定為 JSON。使用提供的測試指令碼時，必須設定 JSON 格式日誌，以便從 CloudWatch Logs 讀取函數的調用狀態，以確認調用成功。

1. 開啟 Lambda 主控台中的函數頁面。
2. 請確定您在 AWS 區域 建立 S3 儲存貯體的相同 中作業。可使用螢幕頂端的下拉式清單來變更區域。



3. 選擇 Create function (建立函數)。

4. 選擇 Author from scratch (從頭開始撰寫)。
5. 在基本資訊下，請執行下列動作：
   a. 針對函數名稱，請輸入 **`EncryptPDF`**。
   b. 針對執行時期，選擇 Python 3.12。
   c. 對於 Architecture (架構)，選擇 x86_64。
6. 執行下列步驟，連接在上一個步驟建立的執行角色：
   a. 展開 Change default execution role (變更預設執行角色) 區段。
   b. 選取使用現有角色。
   c. 在現有角色欄位中，選取您的角色 (`LambdaS3Role`)。
7. 選擇建立函數。

上傳函數程式碼 (主控台)

1. 在程式碼來源窗格中選擇上傳來源。
2. 選擇 .zip 檔案。
3. 選擇上傳。
4. 在檔案選擇器中，選取 .zip 檔案，並選擇開啟。
5. 選擇儲存。

若要設定函數記憶體和逾時 (主控台)

1. 選取函數的組態索引標籤。
2. 在一般組態窗格中，選擇編輯。
3. 將記憶體設定為 256 MB，並將逾時設定為 15 秒。
4. 選擇儲存。

若要設定日誌格式 (主控台)

1. 選取函數的組態索引標籤。
2. 選取監控和操作工具。
3. 在日誌組態窗格中，選擇編輯。

4. 針對記錄組態，選取 JSON。
5. 選擇儲存。

AWS CLI

建立函數 (AWS CLI)

- 從包含 `lambda_function.zip` 檔案的目錄執行以下命令。針對 `region` 參數，請將 `us-east-2` 取代為您建立 S3 儲存貯體的區域。

```
aws lambda create-function --function-name EncryptPDF \
--zip-file fileb://lambda_function.zip --handler lambda_function.lambda_handler
 \
--runtime python3.12 --timeout 15 --memory-size 256 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-2 \
--logging-config LogFormat=JSON
```

設定 Amazon S3 觸發條件以調用函數

若要在將檔案上傳至來源儲存貯體時執行 Lambda 函數，您需要設定函數的觸發條件。可以使用主控台或 AWS CLI來設定 Amazon S3 觸發條件。

> **Important**
>
> 此程序會將 S3 儲存貯體設定為每次在儲存貯體中建立物件時即會調用您的函數。請務必僅在來源儲存貯體上進行設定。如果 Lambda 函數在進行調用的同一個儲存貯體中建立物件，則可以在一個迴圈中連續調用函數。這可能會導致未預期的費用向您的 收費 AWS 帳戶。

Console

設定 Amazon S3 觸發條件 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇您的函數 (`EncryptPDF`)。
2. 選擇 Add trigger (新增觸發條件)。
3. 選取 S3。
4. 在儲存貯體下，選取您的來源儲存貯體。
5. 在事件類型下，選取所有物件建立事件。

6. 在遞迴調用下，選取核取方塊，確認您了解不建議使用相同的 S3 儲存貯體進行輸入和輸出作業。您可以閱讀無伺服器園地中 導致 Lambda 函數失控的遞迴模式，進一步了解 Lambda 中的遞迴調用模式。
7. 選擇新增。

   當您使用 Lambda 主控台建立觸發條件時，Lambda 會自動建立資源型政策，為您選取的服務授予調用函數的許可。

AWS CLI

設定 Amazon S3 觸發條件 (AWS CLI)

1. 為函式新增資源型政策，允許 Amazon S3 來源儲存貯體在您新增檔案時調用函式。資源型政策陳述式提供叫用 函數的其他 AWS 服務 許可。若要授予 Amazon S3 調用函數的許可，請執行下列 CLI 命令。請務必以您自己的 AWS 帳戶 ID 取代 `source-account` 參數，並使用您自己的來源儲存貯體名稱。

```
aws lambda add-permission --function-name EncryptPDF \
--principal s3.amazonaws.com --statement-id s3invoke --action
 "lambda:InvokeFunction" \
--source-arn arn:aws:s3:::amzn-s3-demo-bucket \
--source-account 123456789012
```

   使用此命令定義的政策允許 Amazon S3 僅在來源儲存貯體上發生動作時調用函數。

> **Note**
>
> 雖然 S3 儲存貯體名稱全域唯一，但是在使用資源型政策時，最佳實務是指定儲存貯體必須屬於您的帳戶。這是因為如果您刪除儲存貯體，另一個 可以使用相同的 Amazon Resource Name (ARN) AWS 帳戶 建立儲存貯體。

2. 將下面的 JSON 儲存在名為 `notification.json` 的檔案中。套用至來源儲存貯體時，此 JSON 會設定儲存貯體，以便在每次新增新物件時傳送通知至 Lambda 函數。將 AWS 帳戶 Lambda 函數 ARN AWS 區域 中的數字 和 取代為您自己的帳號和區域。

```
{
"LambdaFunctionConfigurations": [
    {
      "Id": "EncryptPDFEventConfiguration",
```

```
      "LambdaFunctionArn": "arn:aws:lambda:us-
east-2:123456789012:function:EncryptPDF",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. 執行下列 CLI 命令，將您建立的 JSON 檔案中的通知設定套用至來源儲存貯體。用您自己的來源儲存貯體名稱取代 amzn-s3-demo-bucket。

```
aws s3api put-bucket-notification-configuration --bucket amzn-s3-demo-bucket \
--notification-configuration file://notification.json
```

若要深入了解 put-bucket-notification-configuration 命令和 notification-configuration 選項，請參閱 AWS CLI 命令參考中的 put-bucket-notification-configuration。

使用 部署資源 AWS SAM

在開始之前，先確保建置機器上已安裝 Docker 與最新版本的 AWS SAMCLI。

1. 在專案目錄中，將下列程式碼複製並貼到名為 template.yaml 的檔案中。取代預留位置儲存貯體名稱：

   - 對於來源儲存貯體，請以符合 S3 儲存貯體命名規則的任何名稱取代 amzn-s3-demo-bucket。
   - 對於目的地儲存貯體，請將取代 amzn-s3-demo-bucket-encrypted 為 <source-bucket-name>-encrypted，其中 <source-bucket> 是您為來源儲存貯體選擇的名稱。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Resources:
  EncryptPDFFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: EncryptPDF
      Architectures: [x86_64]
      CodeUri: ./
      Handler: lambda_function.lambda_handler
```

```
      Runtime: python3.12
      Timeout: 15
      MemorySize: 256
      LoggingConfig:
        LogFormat: JSON
      Policies:
        - AmazonS3FullAccess
      Events:
        S3Event:
          Type: S3
          Properties:
            Bucket: !Ref PDFSourceBucket
            Events: s3:ObjectCreated:*

  PDFSourceBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: amzn-s3-demo-bucket

  EncryptedPDFBucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: amzn-s3-demo-bucket-encrypted
```

AWS SAM 範本會定義您為應用程式建立的資源。在此範例中，範本使用 `AWS::Serverless::Function` 類型定義 Lambda 函數，並使用 `AWS::S3::Bucket` 類型定義兩個 S3 儲存貯體。範本中指定的儲存貯體名稱是預留位置。使用 部署應用程式之前 AWS SAM，您需要編輯範本，以使用符合 S3 儲存貯體命名規則的全域唯一名稱來重新命名儲存貯體。如需進一步了解，請參閱the section called “使用 部署資源 AWS SAM”。

該 Lambda 函數資源的定義使用 `S3Event` 事件屬性設定函數的觸發條件。此觸發條件會在來源儲存貯體中有物件建立時調用該函數。

函數定義也會指定要連接到函數執行角色的 AWS Identity and Access Management (IAM) 政策。AWS 受管政策 `AmazonS3FullAccess` 為該函數提供讀取和寫入物件至 Amazon S3 所需的許可。

2. 在您儲存 `template.yaml`、`lambda_function.py` 和 `requirements.txt` 檔案的目錄中執行以下命令。

```
sam build --use-container
```

此命令會收集應用程式的建置成品，並將它們放置在適當的格式和位置以進行部署。指定 `--use-container` 選項可在類似 Lambda 的 Docker 容器中建置函數。之所以要使用它，是為了讓您不必在本機機器上安裝 Python 3.12，即可進行建置。

在建置過程中， 會在您以 範本中的 `CodeUri` 屬性指定的位置 AWS SAM 尋找 Lambda 函數程式碼。在此例中，我們將該位置設定為目前的目錄 (`./`)。

如果`requirements.txt`檔案存在， AWS SAM 會使用它來收集指定的相依性。根據預設， 會使用函數程式碼和相依性 AWS SAM 建立 .zip 部署套件。您也可以選擇使用 PackageType 屬性，將函數部署為容器映像。

3. 若要部署應用程式並建立 AWS SAM 範本中指定的 Lambda 和 Amazon S3 資源，請執行下列命令。

```
sam deploy --guided
```

使用 `--guided`旗標表示 AWS SAM 會顯示提示，引導您完成部署程序。對於此部署，請按 Enter 接受預設選項。

在部署過程中， 會在您的 中 AWS SAM 建立下列資源 AWS 帳戶：

- 名為 的 An CloudFormation stack `sam-app`
- 名稱為 `EncryptPDF` 的 Lambda 函數
- 兩個 S3 儲存貯體，其中包含您在編輯`template.yaml` AWS SAM 範本檔案時選擇的名稱
- 具有名稱格式為 `sam-app-EncryptPDFFunctionRole-`*`2qGaapHFWOQ8`* 的函數 IAM 執行角色

AWS SAM 完成建立資源後，您應該會看到下列訊息：

```
Successfully created/updated stack - sam-app in us-east-2
```

測試應用程式

若要測試應用程式，可將 PDF 檔案上傳至來源儲存貯體，並確認 Lambda 在目的地儲存貯體中建立加密的檔案版本。在此範例中，您可以使用 主控台或 手動測試， AWS CLI或使用提供的測試指令碼。

對於生產應用程式，您可以使用單元測試等傳統測試方法和技術，確認 Lambda 函數程式碼是否正常運作。最佳實務也是執行提供的測試指令碼中執行的測試，即執行使用真實雲端資源的整合測試。雲端

的整合測試可確認基礎設施已正確部署，且事件會如預期在不同服務之間流動。如需詳細資訊，請參閱測試無伺服器函數。

手動測試應用程式

您可以將 PDF 檔案新增至 Amazon S3 來源儲存貯體，藉此以手動方式測試該函數。在將檔案新增至來源儲存貯體後，Lambda 函數應該能夠被自動調用，並且會產生經過加密的檔案版本並存放在目標儲存貯體中。

Console

若要透過上傳檔案來測試應用程式 (主控台)

1. 若要將 PDF 檔案上傳到 S3 儲存貯體，請執行以下操作：
   a. 開啟 Amazon S3 主控台的儲存貯體頁面，並選擇來源儲存貯體。
   b. 選擇上傳。
   c. 選擇新增檔案，然後使用檔案選擇器選擇您要上傳的 PDF 檔案。
   d. 選擇開啟，然後選擇上傳。
2. 透過執行下列操作，確認 Lambda 已在目標儲存貯體中儲存了經過加密的 PDF 檔案版本：
   a. 導覽回 Amazon S3 主控台的儲存貯體頁面，然後選擇目的地儲存貯體。
   b. 在物件窗格中，您現在應該會看到名稱格式為 `filename_encrypted.pdf` 的檔案 (其中 `filename.pdf` 是您上傳到來源儲存貯體的檔案名稱)。若要下載經過加密的 PDF 檔案，請選取檔案，然後選擇下載。
   c. 確認您可以使用 Lambda 函數用於保護下載之檔案的密碼 (`my-secret-password`) 開啟文檔。

AWS CLI

若要透過上傳檔案 (AWS CLI) 測試應用程式

1. 從包含要上傳之 PDF 檔案的目錄中，執行下列 CLI 命令。將 `--bucket` 參數取代為來源儲存貯體名稱。對於 `--key` 和 `--body` 參數，請使用測試檔案的檔案名稱。

```
aws s3api put-object --bucket amzn-s3-demo-bucket --key test.pdf --body ./
test.pdf
```

2. 確認函數已建立檔案的加密版本，並將其儲存到目標 S3 儲存貯體。執行以下 CLI 命令，將 amzn-s3-demo-bucket-encrypted 取代為您自己的目的地儲存貯體的名稱。

```
aws s3api list-objects-v2 --bucket amzn-s3-demo-bucket-encrypted
```

如果函數成功運作，則您會看到類似以下內容的輸出。您的目標儲存貯體應包含名稱格式為 <your_test_file>_encrypted.pdf 的檔案，其中 <your_test_file> 是您上傳的檔案名稱。

```
{
    "Contents": [
        {
            "Key": "test_encrypted.pdf",
            "LastModified": "2023-06-07T00:15:50+00:00",
            "ETag": "\"7781a43e765a8301713f533d70968a1e\"",
            "Size": 2763,
            "StorageClass": "STANDARD"
        }
    ]
}
```

3. 若要下載 Lambda 儲存在目的地儲存貯體中的檔案，請執行下面的 CLI 命令。用目的地儲存貯體名稱取代 --bucket。針對 --key 參數，請使用檔案名稱 <your_test_file>_encrypted.pdf，其中 <your_test_file> 是上傳的測試檔案名稱。

```
aws s3api get-object --bucket amzn-s3-demo-bucket-encrypted --
key test_encrypted.pdf my_encrypted_file.pdf
```

此命令會將檔案下載到您目前的目錄，並將其儲存為 my_encrypted_file.pdf。

4. 確認您可以使用 Lambda 函數用於保護下載之檔案的密碼 (my-secret-password) 開啟文檔。

使用自動化指令碼測試應用程式

在專案目錄中建立下列檔案：

- test_pdf_encrypt.py：用於自動化測試應用程式的測試指令碼
- pytest.ini：測試指令碼的組態檔案

展開下列各區段即可檢視程式碼，並進一步了解每個檔案的作用。

自動化測試指令碼

複製以下程式碼並貼到名稱為 test_pdf_encrypt.py 的檔案中。請務必取代預留位置儲存貯體名稱：

- 在 test_source_bucket_available 函數中，將 amzn-s3-demo-bucket 取代為來源儲存貯體名稱。
- 在 test_encrypted_file_in_bucket 函數中，以 *source-bucket*-encrypted 取代 amzn-s3-demo-bucket-encrypted，其中 source-bucket> 是來源儲存貯體的名稱。
- 在 cleanup 函式中，以來源儲存貯體的名稱取代 amzn-s3-demo-bucket，並以目的地儲存貯體取代 amzn-s3-demo-bucket-encrypted。

```
import boto3
import json
import pytest
import time
import os

@pytest.fixture
def lambda_client():
    return boto3.client('lambda')

@pytest.fixture
def s3_client():
    return boto3.client('s3')

@pytest.fixture
def logs_client():
    return boto3.client('logs')

@pytest.fixture(scope='session')
def cleanup():
    # Create a new S3 client for cleanup
    s3_client = boto3.client('s3')

    yield
    # Cleanup code will be executed after all tests have finished

    # Delete test.pdf from the source bucket
```

```
    source_bucket = 'amzn-s3-demo-bucket'
    source_file_key = 'test.pdf'
    s3_client.delete_object(Bucket=source_bucket, Key=source_file_key)
    print(f"\nDeleted {source_file_key} from {source_bucket}")

    # Delete test_encrypted.pdf from the destination bucket
    destination_bucket = 'amzn-s3-demo-bucket-encrypted'
    destination_file_key = 'test_encrypted.pdf'
    s3_client.delete_object(Bucket=destination_bucket, Key=destination_file_key)
    print(f"Deleted {destination_file_key} from {destination_bucket}")


@pytest.mark.order(1)
def test_source_bucket_available(s3_client):
    s3_bucket_name = 'amzn-s3-demo-bucket'
    file_name = 'test.pdf'
    file_path = os.path.join(os.path.dirname(__file__), file_name)

    file_uploaded = False
    try:
        s3_client.upload_file(file_path, s3_bucket_name, file_name)
        file_uploaded = True
    except:
        print("Error: couldn't upload file")

    assert file_uploaded, "Could not upload file to S3 bucket"



@pytest.mark.order(2)
def test_lambda_invoked(logs_client):

    # Wait for a few seconds to make sure the logs are available
    time.sleep(5)

    # Get the latest log stream for the specified log group
    log_streams = logs_client.describe_log_streams(
        logGroupName='/aws/lambda/EncryptPDF',
        orderBy='LastEventTime',
        descending=True,
        limit=1
    )

    latest_log_stream_name = log_streams['logStreams'][0]['logStreamName']
```

```
    # Retrieve the log events from the latest log stream
    log_events = logs_client.get_log_events(
        logGroupName='/aws/lambda/EncryptPDF',
        logStreamName=latest_log_stream_name
    )

    success_found = False
    for event in log_events['events']:
        message = json.loads(event['message'])
        status = message.get('record', {}).get('status')
        if status == 'success':
            success_found = True
            break

    assert success_found, "Lambda function execution did not report 'success' status in
 logs."

@pytest.mark.order(3)
def test_encrypted_file_in_bucket(s3_client):
    # Specify the destination S3 bucket and the expected converted file key
    destination_bucket = 'amzn-s3-demo-bucket-encrypted'
    converted_file_key = 'test_encrypted.pdf'

    try:
        # Attempt to retrieve the metadata of the converted file from the destination
 S3 bucket
        s3_client.head_object(Bucket=destination_bucket, Key=converted_file_key)
    except s3_client.exceptions.ClientError as e:
        # If the file is not found, the test will fail
        pytest.fail(f"Converted file '{converted_file_key}' not found in the
 destination bucket: {str(e)}")

def test_cleanup(cleanup):
    # This test uses the cleanup fixture and will be executed last
    pass
```

自動化測試指令碼會執行三個測試函數，以確認應用程式的正確運作：

- `test_source_bucket_available` 測試透過將測試 PDF 檔案上傳至儲存貯體，確認來源儲存貯體已成功建立。
- 測試 `test_lambda_invoked` 會查詢函數的最新 CloudWatch Logs 日誌串流，以確認在上傳測試檔案後，該 Lambda 函數執行並報告成功。

- 測試 test_encrypted_file_in_bucket 會確認目的地儲存貯體包含經過加密的檔案 test_encrypted.pdf。

執行所有這些測試後，指令碼會執行額外的清除步驟，以刪除來源和目的地儲存貯體中的 test.pdf 和 test_encrypted.pdf 檔案。

如同 AWS SAM 範本，此檔案中指定的儲存貯體名稱是預留位置。在執行測試之前，您需要編輯此檔案，修改為應用程式的真實儲存貯體名稱。如需進一步了解，請參閱the section called “使用自動化指令碼測試應用程式”

測試指令碼組態檔案

複製以下程式碼並貼到名稱為 pytest.ini 的檔案中。

```
[pytest]
markers =
    order: specify test execution order
```

這用於指定 test_pdf_encrypt.py 指令碼中測試的執行順序。

若要執行測試，請執行下列動作：

1. 確保本機環境中已安裝 pytest 模組。您可以透過執行以下命令安裝 pytest：

   ```
   pip install pytest
   ```

2. 在包含檔案 test.pdf 和 test_pdf_encrypt.py 的目錄中儲存名為 pytest.ini 的 PDF 檔案。
3. 開啟終端機或 Shell 程式，並從包含測試檔案的目錄執行以下命令。

   ```
   pytest -s -v
   ```

   測試完成後，輸出應如以下所示：

   ```
   ============================================================ test session starts
    ============================================================
   platform linux -- Python 3.12.2, pytest-7.2.2, pluggy-1.0.0 -- /usr/bin/python3
   cachedir: .pytest_cache
   ```

```
hypothesis profile 'default' -> database=DirectoryBasedExampleDatabase('/home/
pdf_encrypt_app/.hypothesis/examples')
Test order randomisation NOT enabled. Enable with --random-order or --random-order-
bucket=<bucket_type>
rootdir: /home/pdf_encrypt_app, configfile: pytest.ini
plugins: anyio-3.7.1, hypothesis-6.70.0, localserver-0.7.1, random-order-1.1.0
collected 4 items

test_pdf_encrypt.py::test_source_bucket_available PASSED
test_pdf_encrypt.py::test_lambda_invoked PASSED
test_pdf_encrypt.py::test_encrypted_file_in_bucket PASSED
test_pdf_encrypt.py::test_cleanup PASSED
Deleted test.pdf from amzn-s3-demo-bucket
Deleted test_encrypted.pdf from amzn-s3-demo-bucket-encrypted


============================================================== 4 passed in 7.32s
 ===============================================================
```

後續步驟

現在您已完成此範例應用程式的建立。您可以使用提供的程式碼做為基礎，建立其他類型的檔案處理應用程式。您可以根據自己使用案例的檔案處理邏輯，修改 `lambda_function.py` 檔案中的程式碼。

許多典型的檔案處理使用案例都涉及影像處理。使用 Python 時，最常用的影像處理程式庫 (例如 pillow)，通常包含 C 或 C++ 元件。為了確保函數的部署套件與 Lambda 執行環境相容，請務必使用正確的來源分佈二進位檔。

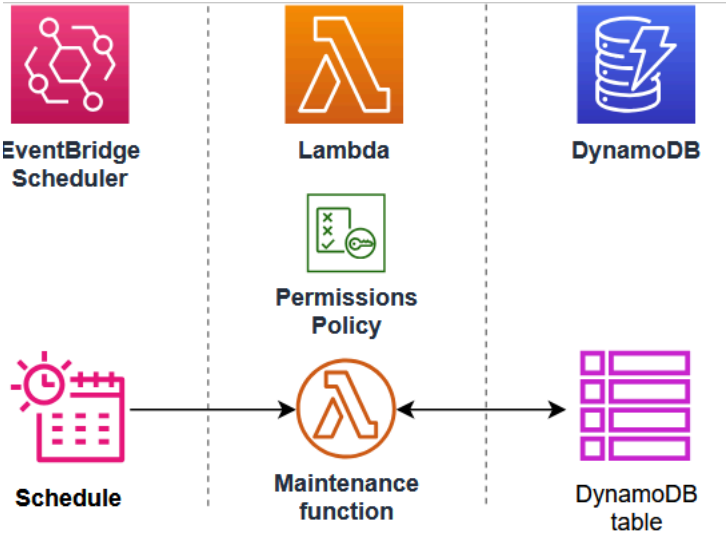
使用 部署資源時 AWS SAM，您需要採取一些額外的步驟，在部署套件中包含正確的來源分佈。由於 AWS SAM 不會為與建置機器不同的平台安裝相依性，因此如果您的建置機器使用與 Lambda 執行環境不同的作業系統或架構，在 `requirements.txt` 檔案中指定正確的來源分佈 (`.whl` 檔案） 將無法運作。您應該執行下列其中一項操作：

- 執行 `sam build` 時使用 `--use-container` 選項。當您指定此選項時， AWS SAM 會下載與 Lambda 執行環境相容的容器基礎映像，並使用該映像在 Docker 容器中建置函數的部署套件。若要進一步了解，請參閱 Building a Lambda function inside of a provided container 一節。
- 使用正確的來源分佈二進位檔自行建置函數的 .zip 部署套件，並將 .zip 檔案儲存在範本中指定為 `CodeUri`的目錄中 AWS SAM 。若要進一步了解如何使用二進位分佈為 Python 建置 .zip 部署套件，請參閱the section called “建立含相依項的 .zip 部署套件”和the section called “建立含原生程式庫的 .zip 部署套件”。

建立應用程式以執行排定的資料庫維護

您可以使用 AWS Lambda 取代排程程序，例如自動化系統備份、檔案轉換和維護任務。在此範例中，您會建立一個無伺服器應用程式，用於透過刪除舊項目，對 DynamoDB 資料表執行定期的排定維護。該應用程式使用 EventBridge 排程器，對 Cron 排程調用 Lambda 函數。調用時，函數會查詢資料表中是否有超過一年的項目，並刪除它們。函數會在 CloudWatch Logs 中記錄刪除的每個項目。

若要實作此範例，首先需要建立 DynamoDB 資料表，並將其填入一些測試資料，以供函數查詢。然後，建立具有 EventBridge 排程器觸發條件的 Python Lambda 函數，並為該函數建立 IAM 執行角色，讓其能夠從資料表讀取和刪除項目。



> **Tip**
>
> 如果您是 Lambda 的新手，建議您在建立此範例應用程式之前完成教學課程「建立第一個函數」。

您可以使用 建立和設定資源，以手動部署您的應用程式 AWS 管理主控台。您也可以使用 AWS Serverless Application Model (AWS SAM) 來部署應用程式。 AWS SAM 是基礎設施即程式碼 (IaC) 工具。藉助 IaC，您無需手動建立資源，而是在程式碼中定義資源，然後便可自動部署資源。

如果您想要在部署此範例應用程式之前，進一步了解如何將 Lambda 與 IaC 搭配使用，請參閱the section called “基礎設施即程式碼 (IaC)”。

先決條件

在建立範例應用程式之前，請確保已安裝必要的命令列工具和程式。

- Python

  為向用於測試應用程式的 DynamoDB 資料表填入資料，此範例使用 Python 指令碼和 CSV 檔案。請確定您的機器已安裝 Python 3.8 版或更新版本。

- AWS SAM CLI

  如果您想要使用 建立 DynamoDB 資料表並部署範例應用程式 AWS SAM，則需要安裝 AWS SAM CLI。如需了解如何安裝，請參閱《AWS SAM 使用者指南》中的安裝指示。

- AWS CLI

  若要使用提供的 Python 指令碼填入測試資料表，則需要安裝並設定 AWS CLI。這是因為指令碼使用 適用於 Python (Boto3) 的 AWS SDK，需要存取您的 AWS Identity and Access Management (IAM) 登入資料。您也需要 AWS CLI 安裝 ，才能使用 來部署資源 AWS SAM。若要安裝 CLI，請參閱《AWS Command Line Interface 使用者指南》中的安裝指示。

- Docker

  若要使用 部署應用程式 AWS SAM，您的建置機器也必須安裝 Docker。請遵循 Docker 文件網站上 Install Docker Engine 一節的指示。

下載範例應用程式檔案

若要建立範例資料庫和排定維護應用程式，您需要在專案目錄中建立下列檔案：

範例資料庫檔案

- `template.yaml` - 可用來建立 DynamoDB 資料表的 AWS SAM 範本
- `sample_data.csv`：CSV 檔案，其中包含要載入資料表的範例資料
- `load_sample_data.py`：Python 指令碼，用於將 CSV 檔案中的資料寫入資料表

排定維護應用程式檔案

- `lambda_function.py`：執行資料庫維護之 Lambda 函數的 Python 函數程式碼
- `requirements.txt`：資訊清單檔案，用於定義 Python 函數程式碼所需的相依項
- `template.yaml` - 可用來部署應用程式的 AWS SAM 範本

測試檔案

- `test_app.py`：Python 指令碼，用於掃描資料表，以確認函數的成功運作，並輸出已存在超過一年的所有記錄

展開以下區段以檢視程式碼，並進一步了解每個檔案在建立和測試應用程式過程中的作用。若要在本機機器上建立這些檔案，請複製並貼上下面的程式碼。

AWS SAM 範本（範例 DynamoDB 資料表）

複製以下程式碼並貼到名稱為 `template.yaml` 的檔案中。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: SAM Template for DynamoDB Table with Order_number as Partition Key and
 Date as Sort Key

Resources:
  MyDynamoDBTable:
    Type: AWS::DynamoDB::Table
    DeletionPolicy: Retain
    UpdateReplacePolicy: Retain
    Properties:
      TableName: MyOrderTable
      BillingMode: PAY_PER_REQUEST
      AttributeDefinitions:
        - AttributeName: Order_number
          AttributeType: S
        - AttributeName: Date
          AttributeType: S
      KeySchema:
        - AttributeName: Order_number
          KeyType: HASH
        - AttributeName: Date
          KeyType: RANGE
      SSESpecification:
        SSEEnabled: true
      GlobalSecondaryIndexes:
        - IndexName: Date-index
          KeySchema:
            - AttributeName: Date
              KeyType: HASH
          Projection:
            ProjectionType: ALL
      PointInTimeRecoverySpecification:
```

```
        PointInTimeRecoveryEnabled: true

Outputs:
  TableName:
    Description: DynamoDB Table Name
    Value: !Ref MyDynamoDBTable
  TableArn:
    Description: DynamoDB Table ARN
    Value: !GetAtt MyDynamoDBTable.Arn
```

> **Note**
>
> AWS SAM 範本使用 的標準命名慣例`template.yaml`。此範例有兩個範本檔案：一個用於建立範例資料庫，另一個用於建立應用程式本身。將它們儲存在專案資料夾的單獨子目錄中。

此 AWS SAM 範本會定義您為測試應用程式所建立的 DynamoDB 資料表資源。該資料表的主索引鍵為 `Order_number`，排序索引鍵為 `Date`。為了讓該 Lambda 函數直接依日期尋找項目，我們還定義了名為 `Date-index` 的全域次要索引。

若要進一步了解如何使用 `AWS::DynamoDB::Table` 資源建立和設定 DynamoDB 資料表，請參閱《AWS CloudFormation 使用者指南》中的 AWS::DynamoDB::Table 一節。

範例資料庫資料檔案

複製以下程式碼並貼到名稱為 `sample_data.csv` 的檔案中。

```
Date,Order_number,CustomerName,ProductID,Quantity,TotalAmount
2023-09-01,ORD001,Alejandro Rosalez,PROD123,2,199.98
2023-09-01,ORD002,Akua Mansa,PROD456,1,49.99
2023-09-02,ORD003,Ana Carolina Silva,PROD789,3,149.97
2023-09-03,ORD004,Arnav Desai,PROD123,1,99.99
2023-10-01,ORD005,Carlos Salazar,PROD456,2,99.98
2023-10-02,ORD006,Diego Ramirez,PROD789,1,49.99
2023-10-03,ORD007,Efua Owusu,PROD123,4,399.96
2023-10-04,ORD008,John Stiles,PROD456,2,99.98
2023-10-05,ORD009,Jorge Souza,PROD789,3,149.97
2023-10-06,ORD010,Kwaku Mensah,PROD123,1,99.99
2023-11-01,ORD011,Li Juan,PROD456,5,249.95
2023-11-02,ORD012,Marcia Oliveria,PROD789,2,99.98
2023-11-03,ORD013,Maria Garcia,PROD123,3,299.97
2023-11-04,ORD014,Martha Rivera,PROD456,1,49.99
```

```
2023-11-05,ORD015,Mary Major,PROD789,4,199.96
2023-12-01,ORD016,Mateo Jackson,PROD123,2,199.99
2023-12-02,ORD017,Nikki Wolf,PROD456,3,149.97
2023-12-03,ORD018,Pat Candella,PROD789,1,49.99
2023-12-04,ORD019,Paulo Santos,PROD123,5,499.95
2023-12-05,ORD020,Richard Roe,PROD456,2,99.98
2024-01-01,ORD021,Saanvi Sarkar,PROD789,3,149.97
2024-01-02,ORD022,Shirley Rodriguez,PROD123,1,99.99
2024-01-03,ORD023,Sofia Martinez,PROD456,4,199.96
2024-01-04,ORD024,Terry Whitlock,PROD789,2,99.98
2024-01-05,ORD025,Wang Xiulan,PROD123,3,299.97
```

此檔案包含一些範例測試資料，為標準逗號分隔值 (CSV) 格式，用於填入 DynamoDB 資料表。

用於載入範例資料的 Python 指令碼

複製以下程式碼並貼到名稱為 `load_sample_data.py` 的檔案中。

```
import boto3
import csv
from decimal import Decimal

# Initialize the DynamoDB client
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table('MyOrderTable')
print("DDB client initialized.")

def load_data_from_csv(filename):
    with open(filename, 'r') as file:
        csv_reader = csv.DictReader(file)
        for row in csv_reader:
            item = {
                'Order_number': row['Order_number'],
                'Date': row['Date'],
                'CustomerName': row['CustomerName'],
                'ProductID': row['ProductID'],
                'Quantity': int(row['Quantity']),
                'TotalAmount': Decimal(str(row['TotalAmount']))
            }
            table.put_item(Item=item)
            print(f"Added item: {item['Order_number']} - {item['Date']}")

if __name__ == "__main__":
    load_data_from_csv('sample_data.csv')
```

```
    print("Data loading completed.")
```

此 Python 指令碼會先使用 適用於 Python (Boto3) 的 AWS SDK 來建立 DynamoDB 資料表的連線。然後，它會逐一查看 CSV 檔案 example-data 中的每一列，從該列建立項目，並使用 boto3 SDK 將項目寫入 DynamoDB 資料表。

Python 函數程式碼

複製以下程式碼並貼到名稱為 `lambda_function.py` 的檔案中。

```
import boto3
from datetime import datetime, timedelta
from boto3.dynamodb.conditions import Key, Attr
import logging

logger = logging.getLogger()
logger.setLevel("INFO")

def lambda_handler(event, context):
    # Initialize the DynamoDB client
    dynamodb = boto3.resource('dynamodb')

    # Specify the table name
    table_name = 'MyOrderTable'
    table = dynamodb.Table(table_name)

    # Get today's date
    today = datetime.now()

    # Calculate the date one year ago
    one_year_ago = (today - timedelta(days=365)).strftime('%Y-%m-%d')

    # Scan the table using a global secondary index
    response = table.scan(
        IndexName='Date-index',
        FilterExpression='#date < :one_year_ago',
        ExpressionAttributeNames={
            '#date': 'Date'
        },
        ExpressionAttributeValues={
            ':one_year_ago': one_year_ago
        }
    )
```

```
     # Delete old items
    with table.batch_writer() as batch:
        for item in response['Items']:
            Order_number = item['Order_number']
            batch.delete_item(
                Key={
                    'Order_number': Order_number,
                    'Date': item['Date']
                }
            )
            logger.info(f'deleted order number {Order_number}')

    # Check if there are more items to scan
    while 'LastEvaluatedKey' in response:
        response = table.scan(
            IndexName='DateIndex',
            FilterExpression='#date < :one_year_ago',
            ExpressionAttributeNames={
                '#date': 'Date'
            },
            ExpressionAttributeValues={
                ':one_year_ago': one_year_ago
            },
            ExclusiveStartKey=response['LastEvaluatedKey']
        )

        # Delete old items
        with table.batch_writer() as batch:
            for item in response['Items']:
                batch.delete_item(
                    Key={
                        'Order_number': item['Order_number'],
                        'Date': item['Date']
                    }
                )

    return {
        'statusCode': 200,
        'body': 'Cleanup completed successfully'
    }
```

該 Python 函數的程式碼包含 Lambda 在調用該函數時執行的處理常式函數 (`lambda_handler`)。

當 EventBridge 排程器調用函數時，它會使用 適用於 Python (Boto3) 的 AWS SDK 來建立 DynamoDB 資料表的連線，以執行排定的維護任務。然後，它會使用 Python 程式庫 `datetime` 來計算距離當日一年的日期，然後再掃描資料表中是否有早於此日期的項目並刪除它們。

請注意，DynamoDB 查詢和掃描操作的回應大小上限為 1 MB。如果回應大於 1 MB，DynamoDB 會對資料進行分頁，並在回應中傳回 `LastEvaluatedKey` 元素。為了確保函數處理資料表中的所有記錄，我們會檢查此索引鍵是否存在，然後繼續從上次評估的位置執行資料表掃描，直到掃描完整個資料表為止。

`requirements.txt` 資訊清單檔案

複製以下程式碼並貼到名稱為 `requirements.txt` 的檔案中。

```
boto3
```

在此範例中，函數程式碼只包含一個不屬於標準 Python 程式庫的相依項，即函數用於掃描 DynamoDB 資料表和刪除項目的適用於 Python 的 SDK (Boto3)。

> **Note**
>
> Lambda 執行時期包含適用於 Python 的 SDK (Boto3) 版本，因此程式碼無需將 Boto3 新增至函數的部署套件即可執行。不過，為了維持對函數相依項的完整控制，並避免版本不一致可能造成的問題，Python 的最佳實務是在函數的部署套件中包含所有函數相依項。如需進一步了解，請參閱the section called “Python 中的執行期相依項”。

AWS SAM 範本 （排程維護應用程式）

複製以下程式碼並貼到名稱為 `template.yaml` 的檔案中。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: SAM Template for Lambda function and EventBridge Scheduler rule

Resources:
  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: ScheduledDBMaintenance
      CodeUri: ./
```

```
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      Events:
        ScheduleEvent:
          Type: ScheduleV2
          Properties:
            ScheduleExpression: cron(0 3 1 * ? *)
            Description: Run on the first day of every month at 03:00 AM
      Policies:
        - CloudWatchLogsFullAccess
        - Statement:
            - Effect: Allow
              Action:
                - dynamodb:Scan
                - dynamodb:BatchWriteItem
              Resource: !Sub 'arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/
MyOrderTable'

  LambdaLogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub /aws/lambda/${MyLambdaFunction}
      RetentionInDays: 30

Outputs:
  LambdaFunctionName:
    Description: Lambda Function Name
    Value: !Ref MyLambdaFunction
  LambdaFunctionArn:
    Description: Lambda Function ARN
    Value: !GetAtt MyLambdaFunction.Arn
```

> **Note**
>
> AWS SAM 範本使用 的標準命名慣例`template.yaml`。此範例有兩個範本檔案：一個用於建立範例資料庫，另一個用於建立應用程式本身。將它們儲存在專案資料夾的單獨子目錄中。

此 AWS SAM 範本會定義您應用程式的資源。我們使用 `AWS::Serverless::Function` 資源定義 Lambda 函數。EventBridge 排程器排程和調用 Lambda 函數的觸發條件乃使用類型為 `ScheduleV2`

之此資源的 Events 屬性建立。若要進一步了解如何在 AWS SAM 範本中定義 EventBridge 排程器排程，請參閱《 AWS Serverless Application Model 開發人員指南》中的 ScheduleV2。

除了 Lambda 函數和 EventBridge 排程器排程外，我們還需要為該函數定義一個 CloudWatch 日誌群組，以將已刪除項目的記錄傳送至其中。

測試指令碼

複製以下程式碼並貼到名稱為 test_app.py 的檔案中。

```
import boto3
from datetime import datetime, timedelta
import json

# Initialize the DynamoDB client
dynamodb = boto3.resource('dynamodb')

# Specify your table name
table_name = 'YourTableName'
table = dynamodb.Table(table_name)

# Get the current date
current_date = datetime.now()

# Calculate the date one year ago
one_year_ago = current_date - timedelta(days=365)

# Convert the date to string format (assuming the date in DynamoDB is stored as a
 string)
one_year_ago_str = one_year_ago.strftime('%Y-%m-%d')

# Scan the table
response = table.scan(
    FilterExpression='#date < :one_year_ago',
    ExpressionAttributeNames={
        '#date': 'Date'
    },
    ExpressionAttributeValues={
        ':one_year_ago': one_year_ago_str
    }
)

# Process the results
old_records = response['Items']
```

```
# Continue scanning if we have more items (pagination)
while 'LastEvaluatedKey' in response:
    response = table.scan(
        FilterExpression='#date < :one_year_ago',
        ExpressionAttributeNames={
            '#date': 'Date'
        },
        ExpressionAttributeValues={
            ':one_year_ago': one_year_ago_str
        },
        ExclusiveStartKey=response['LastEvaluatedKey']
    )
    old_records.extend(response['Items'])

for record in old_records:
    print(json.dumps(record))

# The total number of old records should be zero.
print(f"Total number of old records: {len(old_records)}")
```

此測試指令碼使用 適用於 Python (Boto3) 的 AWS SDK 建立 DynamoDB 資料表的連線，並掃描超過一年的項目。為了讓您能夠確認該 Lambda 函數是否已成功執行，在測試結束時，該函數會列印資料表中存在時間超過一年之記錄的數目。如果 Lambda 函數成功執行，則資料表中的舊記錄數目應為零。

建立範例 DynamoDB 資料表並填入資料

若要測試該排定維護應用程式，請先建立 DynamoDB 資料表，並填入一些範例資料。您可以使用 AWS 管理主控台 手動或使用 AWS SAM自動建立資料表。建議您使用 AWS SAM ，使用幾個 AWS CLI 命令快速建立和設定資料表。

Console

若要建立 DynamoDB 資料表

1. 開啟 DynamoDB 主控台的資料表頁面。
2. 選擇建立資料表。
3. 執行下列動作來建立資料表：
    a. 在資料表詳細資訊下，對於資料表名稱，輸入 **MyOrderTable**。

b. 對於分割區索引鍵，輸入 **Order_number** 並保持類型設定為字串。

c. 對於排序索引鍵，輸入 **Date** 並保持類型設定為字串。

d. 將資料表設定保持為預設設定，然後選擇建立資料表。

4. 當資料表完成建立且其狀態顯示為作用中時，請執行下列動作來建立全域次要索引 (GSI)。該應用程式將使用此 GSI 直接依日期搜尋項目，以決定要刪除的項目。

a. 從資料表清單中選擇 MyOrderTable。

b. 選擇索引索引標籤。

c. 在全域次要索引下，選擇建立索引。

d. 在索引詳細資訊下，輸入 **Date** 作為分割區索引鍵，並保持資料類型設定為字串。

e. 對於 Index name (索引名稱)，輸入 **Date-index**。

f. 保持所有其他參數的預設設定，捲動至頁面底部，然後選擇建立索引。

AWS SAM

若要建立 DynamoDB 資料表

1. 導覽至儲存 DynamoDB 資料表 `template.yaml` 檔案的資料夾。請注意，此範例使用兩個 `template.yaml` 檔案。請確定它們儲存在不同的子資料夾中，而且您位於包含範本的資料夾中，這樣才能建立 DynamoDB 資料表。

2. 執行下列命令。

```
sam build
```

此命令會收集要部署之資源的建置成品，並將它們放置在適當的格式和位置以進行部署。

3. 執行以下命令，以建立 `template.yaml` 檔案中指定的 DynamoDB 資源。

```
sam deploy --guided
```

使用 `--guided`旗標表示 AWS SAM 會顯示提示，引導您完成部署程序。在此部署中，輸入 **cron-app-test-db** 的 `Stack name`，並使用 Enter 鍵接受所有其他選項的預設值。

當 AWS SAM 完成建立 DynamoDB 資源時，您應該會看到下列訊息。

```
Successfully created/updated stack - cron-app-test-db in us-west-2
```

4. 此外，您可以透過開啟 DynamoDB 主控台的資料表頁面，進一步確認該 DynamoDB 資料表已成功建立。您應該會看到名為 `MyOrderTable` 的資料表。

建立資料表之後，您接下來需要新增一些範例資料來測試應用程式。您先前下載的 CSV 檔案 `sample_data.csv` 包含一些範例項目，其中包括訂單編號、日期，以及客戶和訂單資訊。使用提供的 python 指令碼 `load_sample_data.py`，將這些資料新增至資料表。

若要將範例資料新增至資料表

1. 導覽至包含檔案 `sample_data.csv` 和 `load_sample_data.py` 的目錄。如果這兩個檔案位於不同的目錄中，請將它們移動至同一位置。
2. 透過執行以下命令，建立用於執行該指令碼的 Python 虛擬環境。建議您使用虛擬環境，因為在後續步驟中，您需要安裝 適用於 Python (Boto3) 的 AWS SDK。

   ```
   python -m venv venv
   ```

3. 執行以下命令以啟用該虛擬環境：

   ```
   source venv/bin/activate
   ```

4. 執行以下命令，在虛擬環境中安裝適用於 Python 的 SDK (Boto3)。該指令碼使用此程式庫連線到 DynamoDB 資料表和新增項目。

   ```
   pip install boto3
   ```

5. 執行以下命令，以執行指令碼來填入資料表。

   ```
   python load_sample_data.py
   ```

   如果指令碼執行成功，則它會在載入每個項目時將其列印到主控台並報告 `Data loading completed`。

6. 執行以下命令以停用該虛擬環境：

   ```
   deactivate
   ```

7. 您可以透過進行以下操作，以確認資料是否已載入 DynamoDB 資料表：

   a. 開啟 DynamoDB 主控台的探索項目頁面，然後選擇相應資料表 (`MyOrderTable`)。

b. 在傳回的項目窗格中，您應該會看到指令碼新增至資料表的 CSV 檔案中的 25 個項目。

建立排定維護應用程式

您可以使用 AWS 管理主控台 或使用 逐步建立和部署此範例應用程式的資源 AWS SAM。在生產環境中，我們建議您使用 Infrustracture-as-Code (IaC) 工具 AWS SAM ，例如 ，在不使用手動程序的情況下重複部署無伺服器應用程式。

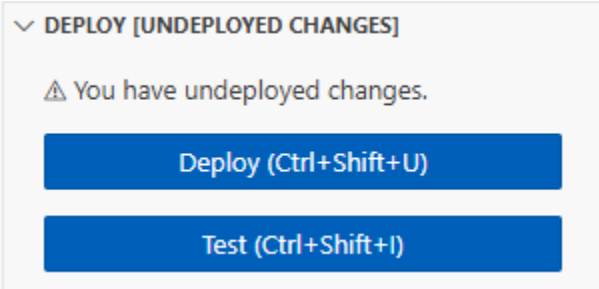
在此範例中，請依照主控台指示了解如何分別設定每個 AWS 資源，或依照 AWS SAM 指示使用 AWS CLI 命令快速部署應用程式。

Console

使用 建立 函數 AWS 管理主控台

首先，建立包含基本入門程式碼的函數。然後，您可以直接在 Lambda 程式碼編輯器中複製和貼上的程式碼，或以 `.zip` 套件的形式上傳程式碼，從而以自己的函數程式碼取代此程式碼。針對此任務，建議複製並貼上程式碼。

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇建立函數。
3. 選擇 Author from scratch (從頭開始撰寫)。
4. 在基本資訊下，請執行下列動作：
   a. 針對函數名稱，請輸入 **`ScheduledDBMaintenance`**。
   b. 針對執行時期，選擇最新的 Python 版本。
   c. 對於 Architecture (架構)，選擇 x86_64。
5. 選擇建立函數。
6. 建立函數後，您可以使用提供的函數程式碼來設定函數。
   a. 在程式碼來源窗格中，以您先前儲存的 `lambda_function.py` 檔案中的 Python 函數程式碼取代 Lambda 建立的 Hello world 程式碼。
   b. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



若要設定函數記憶體和逾時 (主控台)

1. 選取函數的組態索引標籤。
2. 在一般組態窗格中，選擇編輯。
3. 將記憶體設定為 256 MB，並將逾時設定為 15 秒。如果您處理的是具有許多記錄的大型資料表，例如生產環境中的資料表，那麼可以考慮將逾時設定為較大的數字。這可讓函數有更多時間掃描和清理資料庫。
4. 選擇儲存。

若要設定日誌格式 (主控台)

您可以設定 Lambda 函數，以非結構化文字或 JSON 格式輸出日誌。建議您使用 JSON 格式的日誌，這樣搜尋和篩選日誌資料可以更輕鬆。若要進一步了解 Lambda 日誌組態選項，請參閱the section called “設定 Lambda 函數的進階日誌記錄控制項”。

1. 選取函數的組態索引標籤。
2. 選取監控和操作工具。
3. 在日誌組態窗格中，選擇編輯。
4. 針對記錄組態，選取 JSON。
5. 選擇儲存。

若要設定 IAM 許可

若要向該函數提供讀取和刪除 DynamoDB 項目所需的許可，您需要向該函數的執行角色新增政策，以定義必要的許可。

1. 開啟組態索引標籤，然後從左側導覽列中選擇許可。
2. 在執行角色下面，選擇角色名稱。

3. 在 IAM 主控台中，選擇新增許可，然後選擇建立內嵌政策。
4. 使用 JSON 編輯器並輸入以下政策：

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:Scan",
                "dynamodb:DeleteItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/MyOrderTable"
        }
    ]
}
```

5. 將政策命名為 **DynamoDBCleanupPolicy**，然後建立它。

若要將 EventBridge 排程器設定為觸發條件 (主控台)

1. 開啟 EventBridge 主控台。
2. 在左側導覽窗格中，選擇排程器區段下的排程器。
3. 選擇建立排程。
4. 執行下列操作以配置排程：
   a. 在排程名稱下，輸入排程的名稱 (例如 **DynamoDBCleanupSchedule**)。
   b. 在排程模式下，選擇週期性排程。
   c. 對於排程類型，保留預設設定以 Cron 為基礎的排程，然後輸入以下排程詳細資訊：
      - 分鐘：**0**
      - 小時：**3**
      - 月曆日：**1**
      - 月：*****
      - 一週當中的天：**?**
      - 年：*****

評估時，此 Cron 表達式會在每個月第一天的上午 03:00 執行。

d. 對於彈性時段，選擇關閉。

5. 選擇下一步。
6. 執行以下操作以設定 Lambda 函數的觸發條件：

a. 在目標詳細資訊窗格中，將目標 API 設定為範本目標，然後選取 AWS Lambda 調用。

b. 在調用下，從下拉式清單中選取 Lambda 函數 (`ScheduledDBMaintenance`)。

c. 讓承載保留空白，然後選擇下一步。

d. 向下捲動至許可，然後選取為此排程建立新角色。當您使用主控台建立新的 EventBridge 排程器排程時，EventBridge 排程器會建立新的政策，其中包含該排程調用函數所需的必要許可。如需有關管理排程許可的詳細資訊，請參閱《EventBridge 排程器使用者指南》中的 Cron-based schedules 一節。

e. 選擇下一步。

7. 檢閱設定，然後選擇建立排程，以完成排程和 Lambda 觸發條件的建立。

AWS SAM

使用 部署應用程式 AWS SAM

1. 導覽至儲存應用程式 `template.yaml` 檔案的資料夾。請注意，此範例使用兩個 `template.yaml` 檔案。請確定它們儲存在不同的子資料夾中，而且您位於包含範本的資料夾中，這樣才能建立應用程式。
2. 將您先前下載的檔案 `lambda_function.py` 和 `requirements.txt` 複製到相同的資料夾。 AWS SAM 範本中指定的程式碼位置為 `./`，表示目前位置。當您嘗試部署應用程式時，AWS SAM 將在此資料夾中搜尋 Lambda 函數程式碼。
3. 執行下列命令。

```
sam build --use-container
```

此命令會收集要部署之資源的建置成品，並將它們放置在適當的格式和位置以進行部署。指定 `--use-container` 選項可在類似 Lambda 的 Docker 容器中建置函數。之所以要使用它，是為了讓您不必在本機機器上安裝 Python 3.12，即可進行建置。

4. 執行以下命令，以建立 `template.yaml` 檔案中指定的 Lambda 和 EventBridge 排程器資源。

```
sam deploy --guided
```

使用 `--guided`旗標表示 AWS SAM 會顯示提示，引導您完成部署程序。在此部署中，輸入 **`cron-maintenance-app`** 的 `Stack name`，並使用 Enter 鍵接受所有其他選項的預設值。

當 AWS SAM 完成建立 Lambda 和 EventBridge 排程器資源時，您應該會看到下列訊息。

```
Successfully created/updated stack - cron-maintenance-app in us-west-2
```

5. 此外，您可以透過開啟 Lambda 主控台的函數頁面，進一步確認該 Lambda 函數已成功建立。您應該會看到名為 `ScheduledDBMaintenance` 的函數。

測試應用程式

若要測試排程是否能夠正確觸發函數，以及函數是否能夠正確清除資料庫中的記錄，您可以暫時修改排程，以在特定時間執行一次。然後，您可以再次執行 `sam deploy`，將週期性排程重設為每月執行一次。

使用 執行應用程式 AWS 管理主控台

1. 導覽回 EventBridge 排程器主控台頁面。
2. 選擇相應排程，然後選擇編輯。
3. 在排程模式區段的週期下，選擇一次性排程。
4. 將調用時間設定為幾分鐘後並檢閱設定，然後選擇儲存。

在排程執行並調用其目標之後，您可以執行 `test_app.py` 指令碼，以確認函數已成功從 DynamoDB 資料表移除所有舊記錄。

若要使用 Python 指令碼驗證舊記錄是否已被刪除

1. 在命令列中，導覽至儲存 `test_app.py` 的資料夾。
2. 執行指令碼。

```
python test_app.py
```

如果成功，您就會看到以下輸出。

```
Total number of old records: 0
```

後續步驟

您現在可以根據自己特定的應用程式需求，修改 EventBridge 排程器排程。EventBridge Sch排程器支援以下排程表達式：cron、rate 和 one-time schedules。

如需有關 EventBridge 排程器排程表達式的詳細資訊，請參閱 EventBridge Scheduler User Guide 中的 Schedule types。IAM User Guide 中的 Access Management

使用 Lambda 耐久函數建立訂單處理系統

> **Note**
>
> NEED：新增顯示 API Gateway、耐用函數工作流程和支援服務的架構圖 (DynamoDB、EventBridge)

先決條件

- AWS CLI 已安裝和設定
- 需求：特定耐用函數需求

建立來源碼檔案

在專案目錄中建立下列檔案：

- `lambda_function.py` - 函數程式碼
- `requirements.txt` - 相依性資訊清單

函式程式碼

```
# NEED: Verify correct imports
import boto3
import json
```

```
def lambda_handler(event, context):
    # NEED: Verify DurableContext syntax
    durable = context.durable

    try:
        # Validate and store order
        order = await durable.step('validate', async () => {
            return validate_order(event['order'])
        })

        # Process payment
        # NEED: Verify wait syntax
        await durable.wait(/* wait configuration */)

        # Additional steps
        # NEED: Complete implementation

    except Exception as e:
        # NEED: Error handling patterns
        raise e

def validate_order(order_data):
    # NEED: Implementation
    pass
```

要求檔案

```
# NEED: List of required packages
```

部署應用程式

為訂單建立 DynamoDB 資料表

1. 請在 https://console.aws.amazon.com/dynamodb/ 開啟 DynamoDB 主控台。
2. 選擇建立資料表
3. 針對資料表名稱，輸入 **`Orders`**
4. 針對分割區索引鍵，輸入 **`orderId`**
5. 將其他設定保留為預設值
6. 選擇建立資料表

建立 Lambda 函數

1. 在 https：//https://console.aws.amazon.com/lambda/ 開啟 Lambda 主控台
2. 選擇 Create function (建立函數)
3. 從頭選取作者
4. 在函數名稱輸入 **`ProcessOrder`**
5. 針對執行期，選擇您偏好的執行期
6. NEED：新增耐用函數特定的組態
7. 選擇 Create function (建立函數)

建立 API Gateway 端點

1. 在 https：//https://console.aws.amazon.com/apigateway/ 開啟 API Gateway 主控台
2. 選擇建立 API
3. 選取 HTTP API
4. 選擇建置
5. 新增與您的 Lambda 函數的整合
6. 設定訂單處理的路由
7. 部署 API

測試應用程式

提交測試順序：

```
{
    "orderId": "12345",
    "items": [
        {
            "productId": "ABC123",
            "quantity": 1
        }
    ]
}
```

NEED：新增持久性函數的特定監控指示

後續步驟

新增商業邏輯

實作庫存管理：

```
async def check_inventory(order):
    # Add inventory check logic
    pass
```

新增價格計算：

```
async def calculate_total(order):
    # Add pricing logic
    pass
```

改善錯誤處理

新增補償邏輯：

```
async def reverse_payment(order):
    # Add payment reversal logic
    pass
```

處理訂單取消：

```
async def cancel_order(order):
    # Add cancellation logic
    pass
```

整合外部系統

```
async def notify_shipping_provider(order):
    # Add shipping integration
    pass

async def send_customer_notification(order):
    # Add notification logic
    pass
```

增強監控

- 建立 CloudWatch 儀表板
- 設定訂單處理時間的指標
- 設定延遲訂單的提醒

Lambda 的開發工具

您可以存取各種工具，在整個開發生命週期中提高生產力和易用性。本節提供了可協助許多 Lambda 客戶設計、開發和管理其應用程式之工具的資訊。從在 IDE 中進行本機開發，到部署和管理複雜的無伺服器應用程式，這些工具可協助您簡化工作流程、提升程式碼品質，以及加速開發強大的以 Lambda 為基礎之解決方案。

- 本機開發：在偏好的開發環境中更快速地編寫和測試 Lambda 函式。 AWS Toolkit for VS Code 使用 Lambda 的直接部署功能，啟用本機函數開發、偵錯和測試。
- 基礎設施即程式碼 (IaC) - 從本機測試到生產 AWS SAM環境，一致地部署和管理無伺服器應用程式。 AWS CDK CloudFormation 可讓您透過程式碼定義和管理無伺服器基礎設施，以實現一致的版本控制部署。
- GitHub Actions：直接從程式碼儲存庫實現 Lambda 自動化部署。GitHub Actions 可讓您設定工作流程，在您推送程式碼或組態變更時自動部署 Lambda 函式，從而簡化 CI/CD 管道。
- Powertools for AWS Lambda - 使用較少的自訂程式碼建置生產就緒型無伺服器應用程式。Powertools for AWS Lambda （ 也稱為 Powertools for AWS) 是一種開放原始碼開發人員工具組，可簡化在 Python、TypeScript、Java 和 .NET 之間實作無伺服器最佳實務，例如可觀測性、參數擷取和冪等。
- 工作流程和事件 - 協調 Lambda 函數與 AWS 服務、APIs和外部系統。Lambda 提供兩種協同運作選項：Lambda 耐用函數適用於使用 Lambda 內標準程式設計語言的應用程式中心協同運作，AWS Step Functions以及具有跨多個服務的視覺化設計的工作流程中心協同運作。Amazon EventBridge 為事件驅動型架構提供事件管理功能。如需選擇協同運作方法的協助，請參閱耐用函數或 Step Functions。

本機開發工具

本機開發環境讓您能夠離線工作，並在快速迭代 Lambda 函式的同時利用進階 IDE 功能。這些工具可協助您對複雜函式進行偵錯，並在連線受限的環境中進行開發。此外，這些工具也支援團隊協作，能與版本控制系統整合。

如需有關在本機開發 Lambda 函式的詳細資訊，請參閱使用 VS Code 在本機開發 Lambda 函式。此頁面說明如何將 Lambda 函數開發從 AWS 主控台移至 Visual Studio Code，它提供豐富的開發環境，具有偵錯和程式碼完成等功能。若要進行轉換，您需要設定 AWS Toolkit for Visual Studio Code 和 登入資料，之後您可以在 VS Code 中使用進階功能，同時維持直接部署到 的功能 AWS。

Lambda 的本機開發提供數種關鍵功能：

- 使用 Visual Studio Code 與 Lambda 主控台的整合功能
- 設定本機 Lambda 開發環境
- 在本機偵錯並測試函式
- 套用本機函式管理的最佳實務

如需詳細資訊，請參閱使用 VS Code 在本機開發 Lambda 函式。

基礎結構即程式碼 (IaC) 工具

透過基礎結構即程式碼 (IaC) 工具，您可以使用程式碼定義並管理無伺服器架構。這種方法有助於維持跨環境的一致性，讓您控制基礎結構版本，並促進 DevOps 實務。IaC 對於自動化部署、確保一致的環境和管理多區域部署尤其重要。

Lambda 的關鍵 IaC 工具與概念，包括用於範本建立、部署管理及無伺服器基礎結構最佳實務的架構：

- 適用於 Lambda 開發的 IaC 核心原則
- CloudFormation AWS SAM、 和 AWS CDK 功能
- 工具選取條件與比較
- Lambda IaC 實作的最佳實務

無論您是獨立處理小型專案，還是作為大型團隊的一員管理企業級規模的無伺服器應用程式，這些開發和部署工具都可協助您更有效地編寫、部署並管理 Lambda 函式。

如需詳細資訊，請參閱將 Lambda 搭配基礎設施即程式碼 (IaC)。

GitHub Actions 工具

GitHub Actions 提供直接從程式碼儲存庫為 Lambda 函式自動部署的能力。透過在儲存庫中建立工作流程檔案，您可以在推送程式碼或組態變更時自動部署 Lambda 函式，從而簡化持續整合和持續部署 (CI/CD) 管道。部署 Lambda 函數動作提供宣告式 YAML 介面，可簡化部署程序、透過 OpenID Connect (OIDC) 處理 AWS 登入資料，並支援各種部署案例，包括程式碼更新、組態變更和乾執行驗證。這種整合能力使團隊能夠利用現有 GitHub 工作流程，同時維持一致且自動化的部署流程。

如需詳細資訊，請參閱使用 GitHub Actions 部署 Lambda 函式。

AWS Lambda 的 Powertools

Powertools for AWS 是一種開放原始碼開發人員工具組，可協助您以最少的自訂程式碼實作無伺服器最佳實務。該工具組支援 Python、TypeScript/Node.js、Java 與 .NET，提供公用程式函式、裝飾項目和中介軟體，可簡化常見的 Lambda 開發任務。該工具組包含內建的可觀測性功能，例如結構化日誌記錄、追蹤和指標收集，以及用於參數擷取、秘密管理和冪等性模式的公用程式。這些工具符合架構 AWS 良好的最佳實務，並協助開發人員更有效率地建置生產就緒的無伺服器應用程式。透過減少樣板程式碼並標準化常見模式，適用於 的 Powertools AWS 可讓團隊專注於商業邏輯，同時在整個應用程式中維持無伺服器最佳實務的一致實作。

如需詳細資訊，請參閱Powertools for AWS Lambda。

工作流程與事件管理工具

Lambda 應用程式可用於協調複雜的工作流程和處理各種事件。 AWS 提供專用工具，協助您管理無伺服器開發的這些層面。了解 AWS Step Functions 工作流程協同運作的 和事件管理的 Amazon EventBridge，以及如何將其與您的 Lambda 函數整合。這些工具可以提供強大的狀態管理和事件驅動型架構，進而顯著增強無伺服器應用程式的可擴展性和可靠性。透過利用這些服務，您可以建置更複雜且更具彈性的以 Lambda 為基礎之解決方案，從而處理複雜的業務流程，因應各種系統和應用程式事件。

如需詳細資訊，請參閱管理 Lambda 工作流程與事件。

使用 VS Code 在本機開發 Lambda 函式

您可以將 Lambda 函式從 Lambda 主控台遷移至 Visual Studio Code，後者不僅提供完整的開發環境，還支援您使用 AWS SAM與 AWS CDK 等其他本機開發選項。

本機開發的主要優勢

雖然 Lambda 主控台提供了一種快速編輯與測試函式的方式，但本機開發能帶來更進階的功能：

- 進階 IDE 功能：偵錯、程式碼自動完成與重構工具
- 離線開發：在部署至雲端前，先於本機進行程式碼開發與測試
- 基礎結構即程式碼整合：與 AWS SAM、AWS CDK 和 Infrastructure Composer 無縫搭配使用
- 相依項管理：完全控制函式相依項

必要條件

在 VS Code 中本機開發 Lambda 函式之前，您必須已滿足以下條件：

- VS Code：如需安裝說明，請參閱 Download VS Code。
- AWS Toolkit for Visual Studio Code：如需安裝說明，請參閱 Setting up the AWS Toolkit for Visual Studio Code。如需概觀，請參閱 AWS Toolkit for Visual Studio Code。
- AWS 憑證：如需有關設定憑證的資訊，請參閱 Setting up your AWS credentials。
- AWS SAM CLI：如需安裝說明，請參閱 Installing the AWS SAM CLI。
- 已安裝 Docker (選用項，但在本機測試時為必要項)：如需安裝說明，請參閱 Get Docker。

> **Note**
>
> 如果您已在本機設定了 AWS 帳戶和設定檔，請確保已將 AdministratorAccess 受管政策新增至您設定的 AWS 設定檔。

身分驗證與存取控制

與在雲端開發一樣，若要在本機開發 Lambda 函式，您需要 AWS 憑證來代表您安全地存取和管理 AWS 資源。AWS Toolkit for VS Code 支援下列身分驗證方法：

AWS Toolkit for VS Code 支援下列身分驗證方法：

- IAM 使用者長期憑證
- 來自擔任角色的臨時憑證
- 聯合身分
- AWS 帳戶根使用者憑證 (不建議使用)

本節會引導您使用 IAM 使用者長期憑證，完成這些憑證的取得與設定流程。

取得 IAM 憑證

如果您已有具備存取金鑰的 IAM 使用者，請備妥存取金鑰 ID 與私密存取金鑰，供下一節使用。若無所述金鑰，請依照下列步驟建立金鑰：

> **Note**
>
> 您必須一起使用存取金鑰 ID 和私密存取金鑰來驗證您的請求。

若要建立 IAM 使用者與存取金鑰：

1. 前往 https://console.aws.amazon.com/iam/ 開啟 IAM 主控台
2. 在導覽窗格中，選擇使用者 。
3. 選擇 Create user (建立使用者)。
4. 在使用者名稱欄位中輸入名稱，然後選擇下一步。
5. 在設定許可欄位中，選擇直接連接政策。
6. 選取 AdministratorAccess，然後選擇下一步。
7. 選擇 Create user (建立使用者)。
8. 在成功橫幅中，選擇檢視使用者。
9. 選擇 Create access key (建立新的存取金鑰)。
10. 在使用案例欄位中，選擇本機程式碼。
11. 選取確認核取方塊，然後選擇下一步。
12. (選用) 輸入描述標籤值。
13. 選擇 Create access key (建立新的存取金鑰)。
14. 立即複製您的存取金鑰和私密存取金鑰。一旦離開此頁面，您將無法再次存取該私密存取金鑰。

> **Important**
>
> 切勿共用秘密金鑰，亦勿將其提交至原始碼控制系統。請妥善儲存這些金鑰，並在不再需要時立即刪除。

> **Note**
>
> 如需詳細資訊，請參閱 IAM User Guide 中的 Create an IAM user in your AWS account 和 Manage access keys for IAM users。

使用 AWS Toolkit 設定 AWS 憑證

下表總結了您將在後續步驟中完成的憑證設定流程。

操作步驟	為什麼？
開啟「登入」面板	開始驗證
使用命令面板搜尋「AWS 新增連線」	存取登入 UI
選擇「IAM 憑證」	使用存取金鑰以程式設計方式進行存取
輸入設定檔名稱、存取金鑰、秘密金鑰	提供連線所需的憑證
檢視 AWS Explorer 更新	確認連線已成功建立

請依照如下步驟完成 AWS 帳戶身分驗證：

1. 在 VS Code 中開啟「登入」面板：
   a. 若要開始身分驗證流程，請選取左側導覽窗格中的 AWS 圖示，或開啟「命令面板」(在 Mac 上使用 Cmd+Shift+P，在 Windows/Linux 上使用 Ctrl+Shift+P)，然後搜尋並選取「AWS 新增連線」。
2. 在「登入」面板中，選擇 IAM 憑證，然後選取繼續。

> **Note**
>
> 若要繼續，須允許 VS Code 的 AWS IDE 延伸模組存取您的資料。

3. 輸入設定檔名稱、存取金鑰 ID 與私密存取金鑰，然後選取繼續。
4. 透過檢查 VS Code 中的 AWS Explorer，驗證您的 AWS 服務與資源的連線狀態。

如需有關使用長期憑證設定身分驗證的資訊，請參閱 Using long-term credentials to authenticate AWS SDKs and tools。

如需有關設定身分驗證的資訊，請參閱 AWS Toolkit for Visual Studio Code User Guide 中的 AWS IAM credentials。

從主控台遷移至本機開發

> **Note**
>
> 若您已在主控台進行變更，請在轉換至本機開發前，確認所有變更皆已部署完成。

若要將 Lambda 函式從 Lambda 主控台遷移至 VS Code，請完成下列步驟：

1. 開啟 Lambda 主控台。
2. 選擇 函數的名稱。
3. 選取程式碼來源索引標籤。
4. 選擇在 Visual Studio Code 中開啟。

   > **Note**
   >
   > 在 Visual Studio Code 中開啟按鈕僅在 AWS Toolkit 3.69.0 版及更新版本中可用。若已安裝舊版 AWS Toolkit，則可能會在 VS Code 中看到一條訊息：`Cannot open the handler`。若要解決此問題，請將 AWS Toolkit 更新至最新版本。

5. 出現提示時，同意瀏覽器開啟 VS Code。

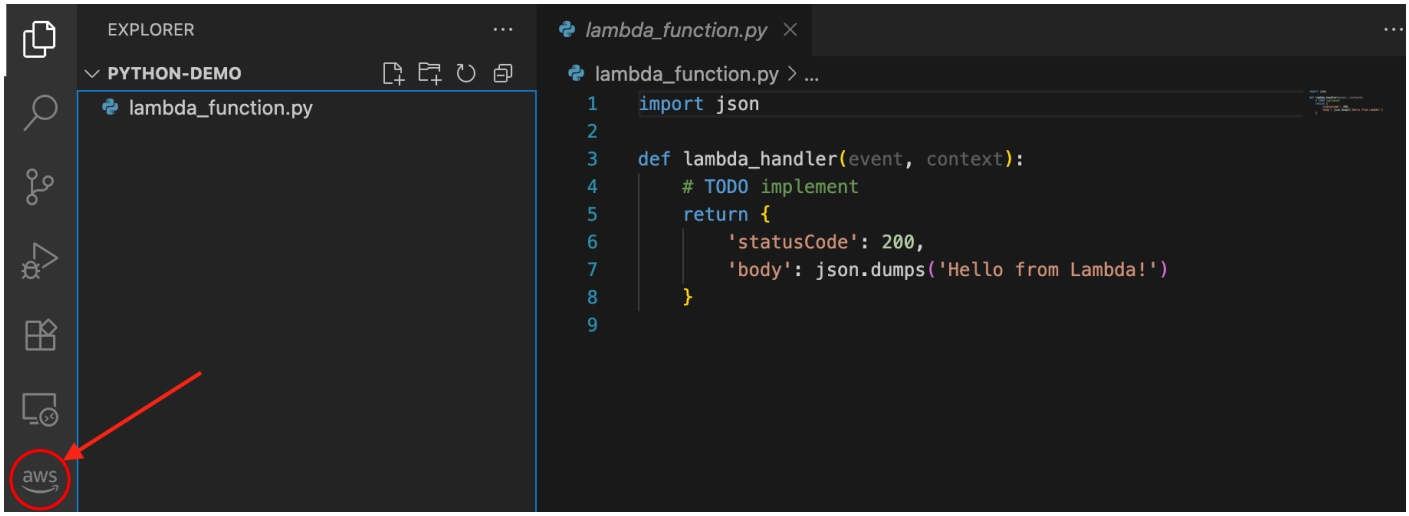
當您在 VS Code 中開啟函式時，Lambda 會在專為快速測試與部署而設計的臨時位置，使用函式程式碼建立一個本機專案。該專案包含函式程式碼、相依項以及可用於本機開發的基本專案結構。

如需有關在 VS Code 中使用 AWS 的詳細資訊，請參閱 AWS Toolkit for Visual Studio Code User Guide。

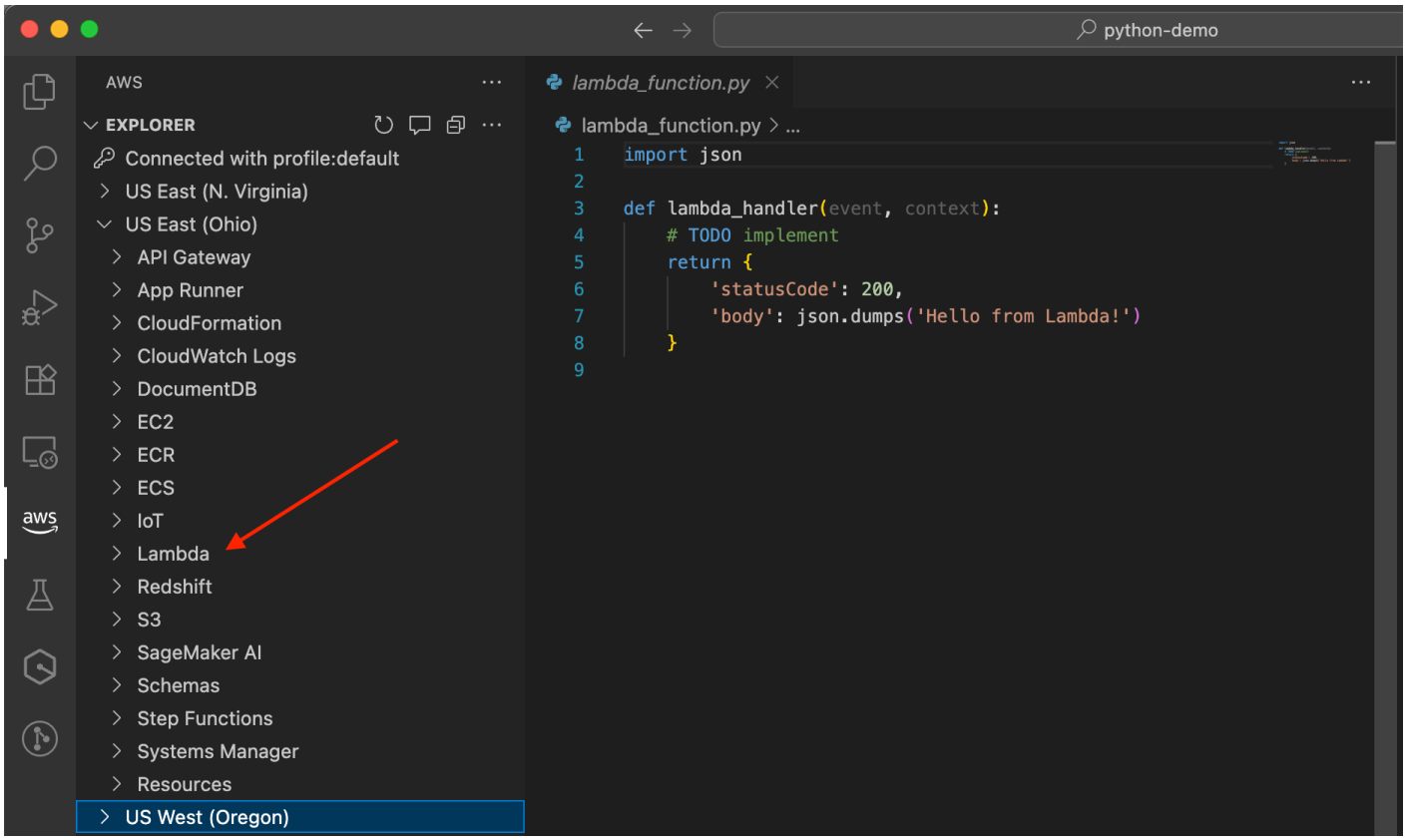
在本機使用函式

在 VS Code 中開啟函式後，請依照下列步驟存取和管理函式：

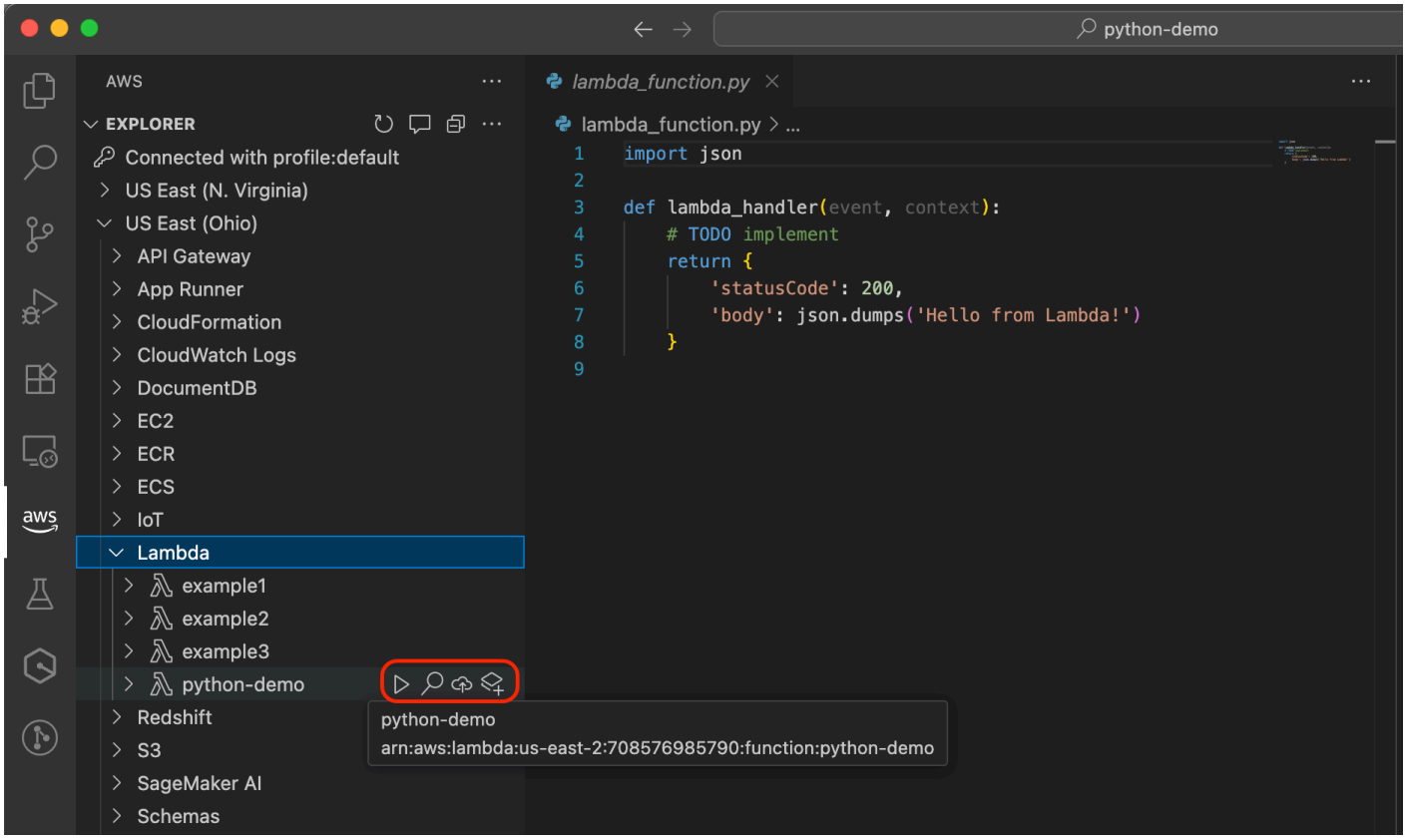
1. 選取側邊欄中的 AWS 圖示，開啟 AWS Explorer：



2. 在 AWS Explorer 中，選取包含 Lambda 函式的區域：



3. 在所選的區域下，展開 Lambda 區段，即可檢視並管理函式：



在 VS Code 中開啟函式後，您可以：

- 使用完整的語言支援與程式碼自動完成功能來編輯函式程式碼。
- 透過 VS Code 的 LocalStack 整合功能，測試會在執行期間對其他 AWS 服務發起 API 呼叫的 Lambda 函式，例如從 DynamoDB 資料表讀取資料或寫入 Amazon S3 儲存貯體。LocalStack 是一種雲端服務模擬器，提供完整的本機開發環境來測試服務間的整合運作。您也可以使用 AWS SAM CLI 在本機容器中測試函式。如果函式對其他 AWS 服務發起 API 呼叫，這些呼叫會到達真正的 AWS 資源，而非模擬資源。
- 透過中斷點與變數檢查功能進行函式偵錯。如需詳細資訊，請參閱 AWS Toolkit for Visual Studio Code User Guide 中的 Running and debugging Lambda functions directly from code。
- 使用雲端圖示將更新後的函式部署回 AWS。
- 安裝並管理函式的相依項。

如需詳細資訊，請參閱 AWS Toolkit for Visual Studio Code User Guide 中的 Working with AWS Lambda functions。

將函式轉換為 AWS SAM 範本並使用 IaC 工具

在 VS Code 中，您可以選擇 Lambda 函式旁邊的轉換為 AWS SAM 應用程式圖示，將 Lambda 函式轉換為 AWS SAM 範本。系統會提示您選取 AWS SAM 專案位置。選取位置後，Lambda 函式會被轉換為儲存在新的 AWS SAM 專案中的 `template.yaml` 檔案。

將函式轉換為 AWS SAM 範本後，您可以：

- 對基礎結構進行版本控制
- 自動化部署流程
- 遠端偵錯函式
- 將其他 AWS 資源新增至應用程式
- 在整個開發生命週期中維護一致的環境
- 透過 Infrastructure Composer 視覺化編輯 AWS SAM 範本

如需有關使用 IaC 工具的詳細資訊，請參閱下列指南：

- AWS Serverless Application Model Developer Guide
- AWS Cloud Development Kit (AWS CDK) Developer Guide
- Infrastructure Composer Developer Guide
- AWS CloudFormation User Guide

這些工具提供其他定義、測試和部署無伺服器應用程式的功能。

後續步驟

若要進一步了解如何在 VS Code 中使用 Lambda 函式，請參閱下列資源：

- AWS Toolkit for VS Code User Guide 中的 Working with AWS Lambda functions
- AWS Toolkit for VS Code User Guide 中的 Working with serverless applications
- 《Lambda 開發人員指南》中的基礎結構即程式碼

使用 GitHub Actions 部署 Lambda 函式

在您將程式碼或組態變更推送至程式碼庫時，可透過 GitHub Actions 自動部署 Lambda 函式。部署 Lambda 函式動作提供簡潔的宣告式 YAML 介面，可消除手動部署步驟的複雜性。

範例工作流程

若要設定自動化 Lambda 函式部署，請在儲存庫的 `.github/workflows/` 目錄中建立工作流程檔案：

Example用於部署 Lambda 的 GitHub Actions 工作流程

```
name: Deploy AWS Lambda

on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
    permissions:
      id-token: write # Required for OIDC authentication
      contents: read  # Required to check out the repository
    steps:
      - uses: actions/checkout@v4

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume: arn:aws:iam::123456789012:role/GitHubActionRole
          aws-region: us-east-1

      - name: Deploy Lambda Function
        uses: aws-actions/aws-lambda-deploy@v1
        with:
          function-name: my-lambda-function
          code-artifacts-dir: ./dist
```

此工作流程會在您將變更推送至 `main` 分支時執行。其會檢查您的儲存庫、使用 OpenID Connect (OIDC) 設定 AWS 憑證，並使用 `./dist` 目錄中的程式碼部署函式。

如需有關更新函式組態、透過 S3 儲存貯體部署及試執行驗證的更多範例，請參閱 Deploy Lambda Function README。

其他資源

- Configure AWS Credentials GitHub Action
- Configuring OpenID Connect in AWS

將 Lambda 搭配基礎設施即程式碼 (IaC)

Lambda 函數很少單獨執行。相反，它們通常是具有其他資源 (例如資料庫、佇列和儲存體) 的無伺服器應用程式的一部分。使用 基礎設施即程式碼 (IaC)，您可以自動化部署程序，以快速且重複地部署並更新涉及許多不同 AWS 資源的整個無伺服器應用程式。這種方法可加快您的開發週期，使組態管理更加輕鬆，並確保您的資源每次都以相同的方式部署。

用於 Lambda 的 IaC 工具

CloudFormation

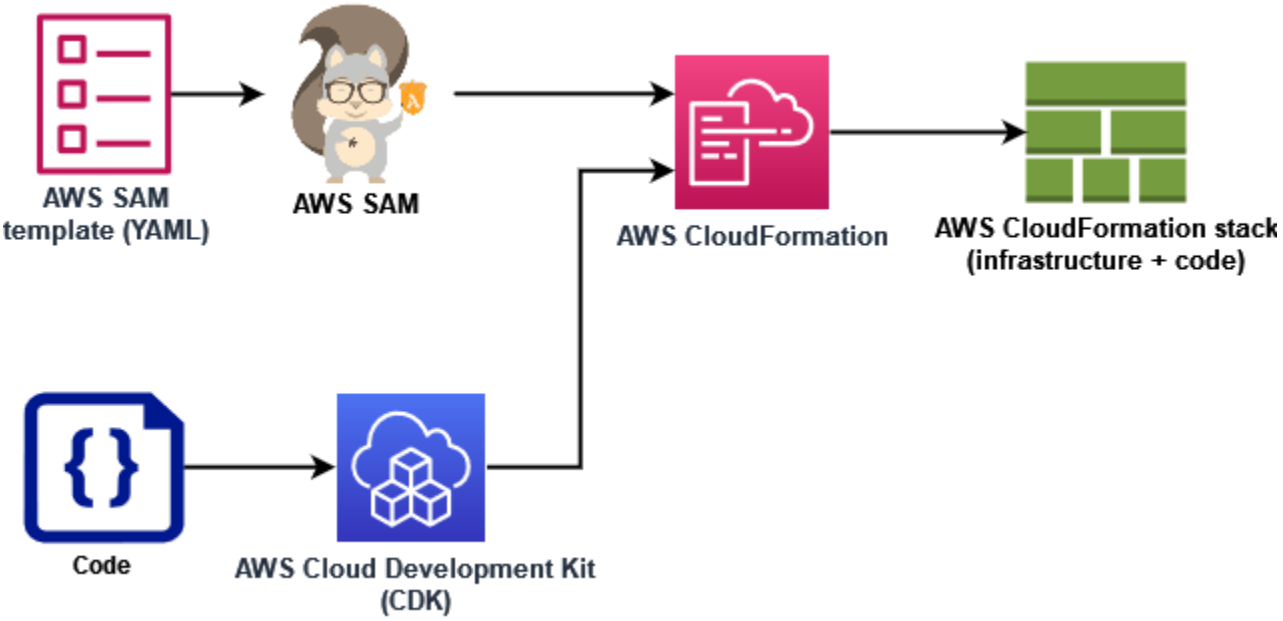
CloudFormation 是來自 AWS 的基礎 IaC 服務。可以使用 YAML 或 JSON 範本來建模和佈建整個 AWS 基礎設施，包括 Lambda 函數。CloudFormation 可處理建立、更新和刪除 AWS 資源的複雜性。

AWS Serverless Application Model (AWS SAM)

AWS SAM 是在 CloudFormation 上建置的開放原始碼架構。它提供可定義無伺服器應用程式的簡化語法。使用 AWS SAM 範本，只需幾行 YAML 即可快速佈建 Lambda 函數、API、資料庫和事件來源。

AWS Cloud Development Kit (AWS CDK)

CDK 是 IaC 的程式碼優先方法。可以使用 TypeScript、JavaScript、Python、Java、C#、.Net 或 Go 來定義基於 Lambda 的架構。選擇您偏好的語言，並使用程式設計元素，例如參數、條件、迴圈、合成和繼承，來定義基礎設施的所需結果。然後，CDK 會產生基礎 CloudFormation 範本以進行部署。如需搭配使用 Lambda 與 CDK 的範例，請參閱 使用 部署 Lambda 函數 AWS CDK。



AWS 還提供了一種稱為 AWS Infrastructure Composer、使用簡單的圖形介面開發 IaC 範本的服務。使用 Infrastructure Composer，您可以透過在視覺化畫布中拖曳、分組和連線 AWS 服務來設計應用程式架構。然後，Infrastructure Composer 會從您的設計中建立 AWS SAM 範本或 CloudFormation 範本，供您用來部署應用程式。

在下面的 the section called “使用 AWS SAM 與 Infrastructure Composer” 章節中，您可以使用 Infrastructure Composer，根據現有的 Lambda 函數為無伺服器應用程式開發範本。

在 AWS SAM 與 Infrastructure Composer 中使用 Lambda 函式

在本教學課程中，您可以透過從現有的 Lambda 函數建立 AWS SAM 範本，然後新增其他 AWS 資源，在 Infrastructure Composer 中建置無伺服器應用程式，開始將 IaC 與 Lambda 搭配使用。

當您執行此教學課程時，您將學習到一些基本概念，例如如何在 AWS SAM 中指定 AWS 資源。您也將學習如何使用 Infrastructure Composer 來建置可使用 AWS SAM 或 CloudFormation 部署的無伺服器應用程式。

請執行下列步驟以完成本教學課程：

- 建立範例 Lambda 函數
- 使用 Lambda 主控台可檢視函數的 AWS SAM 範本
- 將函數的配置匯出到 AWS Infrastructure Composer 並根據函數的組態設計一個簡單的無伺服器應用程式
- 儲存可用來部署無伺服器應用程式的更新 AWS SAM 範本

必要條件

在本教學課程中，您會使用 Infrastructure Composer 的本機同步處理功能，將範本和程式碼檔案儲存到本機建置機器。要使用此功能，您需要一個支援檔案系統存取 API 的瀏覽器，該瀏覽器允許 Web 應用程式在本機檔案系統中讀取、寫入和儲存文件。我們建議使用 Google Chrome 或 Microsoft Edge。如需檔案系統存取 API 的詳細資訊，請參閱什麼是檔案系統存取 API？

建立 Lambda 函數

在此第一步驟中，將會建立 Lambda 函數，可用於完成本教學課程的其餘部分。為了簡化事情，您可以使用 Lambda 主控台，使用 Python 3.11 執行期來建立基本的「Hello world」函數。

若要使用主控台建立「Hello world」Lambda 函數

1. 開啟 Lambda 主控台。
2. 選擇 Create function (建立函數)。
3. 保持選取從頭開始撰寫，然後在基本資訊之下的函數名稱中輸入 **LambdaIaCDemo**。
4. 針對執行期，選取 Python 3.11。
5. 選擇建立函數。

檢視函數的 AWS SAM 範本

在您將函數組態匯出至 Infrastructure Composer 之前，請使用 Lambda 主控台以 AWS SAM 範本的方式檢視函數目前的組態。按照本章節中的步驟操作，您將瞭解 AWS SAM 範本的剖析，以及如何定義 Lambda 函數等資源以開始指定無伺服器應用程式。

檢視函數的 AWS SAM 範本

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您剛建立的函數 (`LambdaIaCDemo`)。
3. 在函數概觀窗格中，選擇範本。

   代替表示函數配置的圖表，您將看到一個函數的 AWS SAM 模板。範本看起來應該如下所示。

```
# This AWS SAM template has been generated from your function's
# configuration. If your function has one or more triggers, note
# that the AWS resources associated with these triggers aren't fully
# specified in this template and include placeholder values.Open this template
# in AWS Application Composer or your favorite IDE and modify
```

```
# it to specify a serverless application with other AWS resources.
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        Statement:
          - Effect: Allow
            Action:
              - logs:CreateLogGroup
            Resource: arn:aws:logs:us-east-1:123456789012:*
          - Effect: Allow
            Action:
              - logs:CreateLogStream
              - logs:PutLogEvents
            Resource:
              - >-
                arn:aws:logs:us-east-1:123456789012:log-group:/aws/lambda/
LambdaIaCDemo:*
```

讓我們花一點時間看看函數的 YAML 範本，並了解一些重要概念。

範本以宣告 `Transform: AWS::Serverless-2016-10-31` 開始。此聲明是必需的，因為在幕後，AWS SAM 模板是透過 CloudFormation 而部署。使用 `Transform` 陳述式將範本識別為 AWS SAM 範本檔案。

在 `Transform` 聲明之後伴隨的 `Resources` 部分。這是您要使用 AWS SAM 範本部署的 AWS 資源的定義位置。AWS SAM 範本可以包含 AWS SAM 資源和 CloudFormation 資源的組合。這是因為在部署期間，AWS SAM 範本會展開為 CloudFormation 範本，因此任何有效的 CloudFormation 語法都可以新增至 AWS SAM 範本。

目前，範本 `Resources` 區段中只定義了一個資源，即您的 Lambda 函數 `LambdaIaCDemo`。若要將 Lambda 函數新增至 AWS SAM 範本，請使用 `AWS::Serverless::Function` 資源類型。Lambda 函數資源的 `Properties` 定義函數的執行期、函數處理常式和其他組態選項。此處也定義了函數原始碼的路徑，AWS SAM 應該用於部署該函數。若要進一步了解 AWS SAM 中的 Lambda 函數資源，請參閱 AWS SAM 開發人員指南 中的 AWS::Serverless::Function。

除了函數屬性和組態外，範本還會為您的函數指定 AWS Identity and Access Management (IAM) 政策。此政策授予您函數將日誌寫入 Amazon CloudWatch Logs 的許可。當您在 Lambda 主控台中建立函數時，Lambda 會自動將此政策附加至您的函數。若要進一步瞭解如何為 AWS SAM 範本中的函數指定 IAM 政策，請參閱《AWS SAM 開發人員指南》中 AWS::Serverless::Function 頁面上的 `policies` 屬性。

若要進一步瞭解 AWS SAM 範本的結構，請參閱AWS SAM 範本剖析。

使用 AWS Infrastructure Composer 來設計無伺服器應用程式

若要開始使用函數的 AWS SAM 範本做為起點來建立簡單的無伺服器應用程式，請將函數組態匯出至 Infrastructure Composer，然後啟動 Infrastructure Composer 的本機同步處理模式。本機同步會自動將函數的程式碼和 AWS SAM 範本保存到本機構置機器，並在您在 Infrastructure Composer 中新增其他 AWS 資源時保持已儲存的範本保持同步。

若要將您的函數匯出至 Infrastructure Composer

1. 在函數概觀窗格中，選擇匯出至應用程式編寫器。

   若要將函數的組態和程式碼匯出至 Infrastructure Composer，Lambda 會在您的帳戶中建立 Amazon S3 儲存貯體來暫時存放此資料。
2. 在對話方塊中，選擇確認並建立專案以接受此儲存貯體的預設名稱，並將函數的設定和程式碼匯出至 Infrastructure Composer。
3. (選擇性) 若要為 Lambda 建立的 Amazon S3 儲存貯體選擇其他名稱，請輸入新名稱，然後選擇確認並建立專案。Amazon S3 儲存貯體的名稱必須是全域唯一的，並遵循儲存貯體命名規則。

選取確認並建立專案會開啟 Infrastructure Composer 主控台。在畫布上，您將看到您 Lambda 函數。

4. 從選單下拉式清單中選擇啟用本機同步。
5. 在開啟的對話方塊中，選擇選取資料夾，然後選取本機建置機器上的資料夾。
6. 選擇啟用以啟用本機同步。

若要將您的函數匯出至 Infrastructure Composer，您需要有使用某些 API 動作的許可。如果您無法匯出函數，請見 the section called “所需的許可” 並確認您有所需的許可。

> **Note**
>
> 標準 Amazon S3 定價適用於 Lambda 在您將函數匯出至 Infrastructure Composer 時所建立的儲存貯體。Lambda 放入儲存貯體的物件會在 10 天後自動刪除，但 Lambda 不會刪除儲存貯體本身。
> 若要避免額外費用新增至您的 AWS 帳戶，請在將函數匯出至 Infrastructure Composer 之後，依照刪除儲存貯體中的指示執行。如需 Lambda 所建立 Amazon S3 儲存貯體的詳細資訊，請參閱 the section called “Infrastructure Composer”。

若要在 Infrastructure Composer 中設計無伺服器應用程式

啟用本機同步之後，您在 Infrastructure Composer 中所做的變更會反映在儲存於本機建置機器上的 AWS SAM 範本中。您現在可以將其他 AWS 資源拖放到 Infrastructure Composer 畫布上，以建置您的應用程式。在此範例中，您將 Amazon SQS 簡單佇列新增為 Lambda 函數的觸發程序，以及新增 DynamoDB 資料表供函數寫入資料。

1. 執行下列動作，將 Amazon SQS 觸發條件新增至您的 Lambda 函數：
   a. 在資源面板的搜尋欄位中，輸入 **SQS**。
   b. 將 SQS 佇列資源拖曳到畫布上，並將其放置在 Lambda 函數的左側。
   c. 選擇詳細資訊，然後為邏輯 ID 輸入 **LambdaIaCQueue**。
   d. 選擇儲存。
   e. 按一下 SQS 佇列卡上的訂閱連接埠，然後將它拖曳至 Lambda 函數卡上的左側連接埠，即可連接您的 Amazon SQS 和 Lambda 資源。兩個資源之間出現一條線表示連線成功。Infrastructure Composer 也會在畫布底部顯示訊息，指示出兩個資源已成功連線。
2. 執行下列動作，為您的 Lambda 函數新增 Amazon DynamoDB 資料表，以便將資料寫入：

a. 在資源面板的搜尋欄位中，輸入 **DynamoDB**。

b. 將 DynamoDB 資料表資源拖曳到畫布上，並將其放置在 Lambda 函數的右側。

c. 選擇詳細資訊，然後為邏輯 ID 輸入 **LambdaIaCTable**。

d. 選擇儲存。

e. 按一下 Lambda 函數卡的右側連接埠，然後將其拖曳至 DynamoDB 卡上的左側連接埠，藉此將 DynamoDB 資料表連接至 Lambda 函數。

現在您已新增這些額外資源，讓我們來看看 Infrastructure Composer 已建立的更新 AWS SAM 範本。

若要檢視更新的 AWS SAM 範本

- 在 Infrastructure Composer 畫布上，選擇範本以從畫布檢視切換至範本檢視。

您的 AWS SAM 範本現在應該包含下列其他資源和屬性：

- 識別碼為 `LambdaIaCQueue` 的 Amazon SQS 佇列

```
LambdaIaCQueue:
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
```

  當您使用 Infrastructure Composer 新增 Amazon SQS 佇列時，Infrastructure Composer 會設定 `MessageRetentionPeriod` 屬性。您也可以選取 SQS 佇列卡上的詳細資訊，然後核取或取消核取 Fifo 佇列來設定 `FifoQueue` 屬性。

  若要設定佇列的其他屬性，您可以手動編輯範本以新增它們。若要進一步瞭解 `AWS::SQS::Queue` 資源及其可用的屬性，請參閱《CloudFormation 使用者指南》中的AWS::SQS::Queue。

- Lambda 函數定義中的 `Events` 屬性，可將 Amazon SQS 佇列指定為函數的觸發程序

```
Events:
  LambdaIaCQueue:
    Type: SQS
    Properties:
      Queue: !GetAtt LambdaIaCQueue.Arn
      BatchSize: 1
```

此 Events 屬性由事件類型和一組依賴於類型的屬性組成。若要瞭解不同 AWS 服務，您可以設定以觸發 Lambda 函數和您可以設定的屬性，請參閱《AWS SAM 開發人員指南》中的 EventSource。

- 具有識別碼 LambdaIaCTable 的 DynamoDB 資料表

```
LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

使用 Infrastructure Composer 新增 DynamoDB 資料表時，您可以選擇 DynamoDB 資料表卡片上的詳細資料並編輯索引鍵值來設定資料表的索引鍵。Infrastructure Composer 也會設定許多其他屬性的預設值，包括 BillingMode 和 StreamViewType。

若要進一步瞭解這些屬性和您可以新增至 AWS SAM 範本的其他屬性，請參閱《CloudFormation 使用者指南》中的 AWS::DynamoDB::Table。

- 一項新的 IAM 政策，可讓您在新增的 DynamoDB 資料表上執行 CRUD 操作的函數許可。

```
Policies:
...
  - DynamoDBCrudPolicy:
    TableName: !Ref LambdaIaCTable
```

完整的最終 AWS SAM 範本看起來應該如下所示。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Specification template describing your function.
Resources:
  LambdaIaCDemo:
    Type: AWS::Serverless::Function
    Properties:
```

```
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 3
      Handler: lambda_function.lambda_handler
      Runtime: python3.11
      Architectures:
        - x86_64
      EventInvokeConfig:
        MaximumEventAgeInSeconds: 21600
        MaximumRetryAttempts: 2
      EphemeralStorage:
        Size: 512
      RuntimeManagementConfig:
        UpdateRuntimeOn: Auto
      SnapStart:
        ApplyOn: None
      PackageType: Zip
      Policies:
        - Statement:
            - Effect: Allow
              Action:
                - logs:CreateLogGroup
              Resource: arn:aws:logs:us-east-1:594035263019:*
            - Effect: Allow
              Action:
                - logs:CreateLogStream
                - logs:PutLogEvents
              Resource:
                - arn:aws:logs:us-east-1:594035263019:log-group:/aws/lambda/
LambdaIaCDemo:*
        - DynamoDBCrudPolicy:
            TableName: !Ref LambdaIaCTable
      Events:
        LambdaIaCQueue:
          Type: SQS
          Properties:
            Queue: !GetAtt LambdaIaCQueue.Arn
            BatchSize: 1
      Environment:
        Variables:
          LAMBDAIACTABLE_TABLE_NAME: !Ref LambdaIaCTable
          LAMBDAIACTABLE_TABLE_ARN: !GetAtt LambdaIaCTable.Arn
  LambdaIaCQueue:
```

```
    Type: AWS::SQS::Queue
    Properties:
      MessageRetentionPeriod: 345600
  LambdaIaCTable:
    Type: AWS::DynamoDB::Table
    Properties:
      AttributeDefinitions:
        - AttributeName: id
          AttributeType: S
      BillingMode: PAY_PER_REQUEST
      KeySchema:
        - AttributeName: id
          KeyType: HASH
      StreamSpecification:
        StreamViewType: NEW_AND_OLD_IMAGES
```

使用 AWS SAM (選擇性) 部署您的無伺服器應用程式

如果您使用剛剛在 Infrastructure Composer 中建立的範本，想要用 AWS SAM 來部署無伺服器應用程式，則必須先安裝 AWS SAM CLI。若要執行此操作，請遵循安裝 AWS SAM CLI 中的說明。

在部署應用程式之前，您也需要更新 Infrastructure Composer 與範本一起儲存的函數程式碼。目前，Infrastructure Composer 儲存的 `lambda_function.py` 檔案只包含 Lambda 在建立函數時提供的基本 'Hello world' 程式碼。

若要更新函數程式碼，請複製下列程式碼，並將其貼到 Infrastructure Composer 儲存至本機建置機器的 `lambda_function.py` 檔案中。當您啟動本機同步模式時，已指定 Infrastructure Composer 要儲存此檔案的目錄。

此程式碼接受來自您在 Infrastructure Composer 中建立之 Amazon SQS 佇列的訊息中的鍵值對。如果索引鍵和值都是字串，則程式碼會使用它們將項目寫入範本中定義的 DynamoDB 資料表。

更新 Python 函數程式碼

```
import boto3
import os
import json

# define the DynamoDB table that Lambda will connect to
tablename = os.environ['LAMBDAIACTABLE_TABLE_NAME']

# create the DynamoDB resource
dynamo = boto3.client('dynamodb')
```

```
def lambda_handler(event, context):
    # get the message out of the SQS event
    message = event['Records'][0]['body']
    data = json.loads(message)
    # write event data to DDB table
    if check_message_format(data):
        key = next(iter(data))
        value = data[key]
        dynamo.put_item(
            TableName=tablename,
            Item={
                'id': {'S': key},
                'Value': {'S': value}
            }
        )
    else:
        raise ValueError("Input data not in the correct format")

# check that the event object contains a single key value
# pair that can be written to the database
def check_message_format(message):
    if len(message) != 1:
        return False

    key, value = next(iter(message.items()))

    if not (isinstance(key, str) and isinstance(value, str)):
        return False

    else:
        return True
```

若要部署您的無伺服器應用程式

若要使用 AWS SAM CLI 部署您的應用程式，請執行下列步驟。若要讓您的函數正確建置和部署，Python 第 3.11 版本必須安裝在您的建置機器和 PATH。

1. 從 Infrastructure Composer 儲存 `template.yaml` 和 `lambda_function.py` 檔案的目錄中執行下列命令。

   ```
   sam build
   ```

此命令會收集應用程式的建置成品，並將它們放置在適當的格式和位置以進行部署。

2. 若要部署應用程式並建立 AWS SAM 範本中指定的 Lambda、Amazon SQS 和 DynamoDB 資源，請執行下列命令。

```
sam deploy --guided
```

使用此 `--guided` 標記意味著 AWS SAM 將顯現提示，以引導您完成部署過程。對於此部署，請按 Enter 接受預設選項。

在部署程序期間，AWS SAM 會在您的 AWS 帳戶 中建立下列資源：

- 一個名為 `sam-app` 的 CloudFormation 堆疊
- 名稱格式為 `sam-app-LambdaIaCDemo-99VXPpYQVv1M` 的 Lambda 函數
- 名稱格式為 `sam-app-LambdaIaCQueue-xL87VeKsGiIo` 的 Amazon SQS 佇列
- 名稱格式為 `sam-app-LambdaIaCTable-CN0S66C0VLNV` 的 DynamoDB 資料表

AWS SAM 還會建立必要的 IAM 角色和政策，讓 Lambda 函數可以讀取來自 Amazon SQS 佇列的訊息，並在 DynamoDB 資料表上執行 CRUD 操作。

測試已部署的應用程式 (選擇性)

若要確認您的無伺服器應用程式是否正確部署，請將訊息傳送至包含索引鍵值組的 Amazon SQS 佇列，並檢查 Lambda 是否使用這些值將項目寫入 DynamoDB 資料表。

測試您的無伺服器應用程式

1. 開啟 Amazon SQS 主控台的佇列頁面，然後選取從範本所建立 AWS SAM 的佇列。名稱具有格式 `sam-app-LambdaIaCQueue-xL87VeKsGiIo`。
2. 選擇傳送和接收訊息，然後將以下 JSON 貼到傳送訊息區段裡的訊息內文中。

```
{
    "myKey": "myValue"
}
```

3. 選擇傳送訊息。

將訊息傳送至佇列會導致 Lambda 透過範本 AWS SAM 中定義的事件來源映射調用您的函數。若要確認 Lambda 已如預期調用您的函數，請確認項目已新增至 DynamoDB 資料表中。

4. 開啟 DynamoDB 主控台的 資料表 頁面，然後選擇資料表。名稱具有格式 `sam-app-LambdaIaCTable-CN0S66C0VLNV`。
5. 選擇探索資料表項目。在 Items returned 窗格中，應該會看到一個包含 id `myKey` 和 數值 `myValue` 的項目。

使用 部署 Lambda 函數 AWS CDK

AWS Cloud Development Kit (AWS CDK) 是基礎設施即程式碼 (IaC) 架構，您可以使用您選擇的程式設計語言來定義 AWS 雲端基礎設施。若要定義您自己的雲端基礎設施，要先編寫包含一個或更多堆疊的應用程式 (使用 CDK 支援的其中一種語言)。然後，您將它合成到 CloudFormation 範本，並將您的資源部署到 AWS 帳戶。按照本主題中的步驟部署一個 Lambda 函數，從 Amazon API Gateway 端點傳回事件。

隨附於 CDK 的 AWS 建構程式庫提供模組，可讓您用來建立 AWS 服務 所提供資源的模型。針對熱門服務，程式庫會提供具有智能預設和最佳實務的彙整建構。您可以使用 aws_lambda 模組來定義函數和相關資源，幾行程式碼即可完成。

先決條件

開始本教學課程之前，請執行下列命令 AWS CDK 來安裝 。

```
npm install -g aws-cdk
```

步驟 1：設定您的 AWS CDK 專案

為您的新 AWS CDK 應用程式建立目錄並初始化專案。

JavaScript

```
mkdir hello-lambda
cd hello-lambda
cdk init --language javascript
```

TypeScript

```
mkdir hello-lambda
```

```
cd hello-lambda
cdk init --language typescript
```

Python

```
mkdir hello-lambda
cd hello-lambda
cdk init --language python
```

專案開始後，啟用專案的虛擬環境，並安裝 AWS CDK的基準相依項。

```
source .venv/bin/activate
python -m pip install -r requirements.txt
```

Java

```
mkdir hello-lambda
cd hello-lambda
cdk init --language java
```

將此 Maven 專案匯入 Java 整合式開發環境 (IDE)。例如，在 Eclipse 中，依次選擇檔案 > 匯入 > Maven > 現有的 Maven 專案。

C#

```
mkdir hello-lambda
cd hello-lambda
cdk init --language csharp
```

> **Note**
>
> AWS CDK 應用程式範本使用專案目錄的名稱來產生來源檔案和類別的名稱。在此範例中，目錄名為 `hello-lambda`。若使用不同的專案目錄名稱，您的應用程式將與這些說明不相符。

AWS CDK v2 包含名為 的單一套件 AWS 服務 中所有 的穩定建構`aws-cdk-lib`。在初始化專案時，此套件會安裝為相依性套件。使用某些程式設計語言時，套件會在您第一次建置專案時安裝。

步驟 2：定義 AWS CDK 堆疊

CDK 堆疊是一或多個建構的集合，可定義 AWS 資源。每個 CDK 堆疊代表您 CDK 應用程式中的 CloudFormation 堆疊。

若要定義 CDK 堆疊，請遵循您偏好之程式設計語言的指示。此堆疊定義下列項目：

- 函數的邏輯名稱：`MyFunction`
- 函數程式碼的位置 (在 `code` 屬性中指定)。如需詳細資訊，請參閱 AWS Cloud Development Kit (AWS CDK) API Reference 中的 Handler code 一節。
- REST API 的邏輯名稱：`HelloApi`
- API Gateway 端點的邏輯名稱：`ApiGwEndpoint`

請注意，本教學課程中的所有 CDK 堆疊都會使用 Lambda 函數的 Node.js 執行時期。您可以針對此 CDK 堆疊和 Lambda 函數使用不同的程式設計語言，以利用每種語言的優勢。例如，您可以針對 CDK 堆疊使用 TypeScript，以利用基礎設施程式碼靜態輸入的優勢。您可以針對 Lambda 函數使用 JavaScript，以利用動態類型語言的彈性和快速開發優勢。

JavaScript

開啟 `lib/hello-lambda-stack.js` 檔案並將內容替換如下：

```
const { Stack } = require('aws-cdk-lib');
const lambda = require('aws-cdk-lib/aws-lambda');
const apigw = require('aws-cdk-lib/aws-apigateway');

class HelloLambdaStack extends Stack {
  /**
   *
   * @param {Construct} scope
   * @param {string} id
   * @param {StackProps=} props
   */
  constructor(scope, id, props) {
    super(scope, id, props);
    const fn = new lambda.Function(this, 'MyFunction', {
      code: lambda.Code.fromAsset('lib/lambda-handler'),
      runtime: lambda.Runtime.NODEJS_LATEST,
      handler: 'index.handler'
    });
```

```
    const endpoint = new apigw.LambdaRestApi(this, 'MyEndpoint', {
      handler: fn,
      restApiName: "HelloApi"
    });

  }
}

module.exports = { HelloLambdaStack }
```

TypeScript

開啟 lib/hello-lambda-stack.ts 檔案並將內容替換如下：

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
import * as apigw from "aws-cdk-lib/aws-apigateway";
import * as lambda from "aws-cdk-lib/aws-lambda";
import * as path from 'node:path';

export class HelloLambdaStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps){
    super(scope, id, props)
    const fn = new lambda.Function(this, 'MyFunction', {
      runtime: lambda.Runtime.NODEJS_LATEST,
      handler: 'index.handler',
      code: lambda.Code.fromAsset(path.join(__dirname, 'lambda-handler')),
    });

    const endpoint = new apigw.LambdaRestApi(this, `ApiGwEndpoint`, {
      handler: fn,
      restApiName: `HelloApi`,
    });

  }
}
```

Python

開啟 /hello-lambda/hello_lambda/hello_lambda_stack.py 檔案並將內容替換如下：

```
from aws_cdk import (
    Stack,
```

```
    aws_apigateway as apigw,
    aws_lambda as _lambda
)
from constructs import Construct

class HelloLambdaStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        fn = _lambda.Function(
            self,
            "MyFunction",
            runtime=_lambda.Runtime.NODEJS_LATEST,
            handler="index.handler",
            code=_lambda.Code.from_asset("lib/lambda-handler")
        )

        endpoint = apigw.LambdaRestApi(
            self,
            "ApiGwEndpoint",
            handler=fn,
            rest_api_name="HelloApi"
        )
```

Java

開啟 /hello-lambda/src/main/java/com/myorg/HelloLambdaStack.java 檔案並將內容替換如下：

```
package com.myorg;

import software.constructs.Construct;
import software.amazon.awscdk.Stack;
import software.amazon.awscdk.StackProps;
import software.amazon.awscdk.services.apigateway.LambdaRestApi;
import software.amazon.awscdk.services.lambda.Function;

public class HelloLambdaStack extends Stack {
    public HelloLambdaStack(final Construct scope, final String id) {
        this(scope, id, null);
    }
```

```
    public HelloLambdaStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        Function hello = Function.Builder.create(this, "MyFunction")

 .runtime(software.amazon.awscdk.services.lambda.Runtime.NODEJS_LATEST)

 .code(software.amazon.awscdk.services.lambda.Code.fromAsset("lib/lambda-handler"))
                            .handler("index.handler")
                            .build();

        LambdaRestApi api = LambdaRestApi.Builder.create(this, "ApiGwEndpoint")
                                .restApiName("HelloApi")
                                .handler(hello)
                                .build();

    }
}
```

C#

開啟 src/HelloLambda/HelloLambdaStack.cs 檔案並將內容替換如下：

```
using Amazon.CDK;
using Amazon.CDK.AWS.APIGateway;
using Amazon.CDK.AWS.Lambda;
using Constructs;

namespace HelloLambda
{
    public class HelloLambdaStack : Stack
    {
        internal HelloLambdaStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var fn = new Function(this, "MyFunction", new FunctionProps
            {
                Runtime = Runtime.NODEJS_LATEST,
                Code = Code.FromAsset("lib/lambda-handler"),
                Handler = "index.handler"
            });
```

```
            var api = new LambdaRestApi(this, "ApiGwEndpoint", new
 LambdaRestApiProps
            {
                Handler = fn
            });
        }
    }
}
```

步驟 3：建立 Lambda 函數程式碼

1. 從專案 (hello-lambda) 的根目錄建立 Lambda 函數程式碼的 /lib/lambda-handler 目錄。此目錄是在 AWS CDK 堆疊的 code 屬性中指定。
2. 在 /lib/lambda-handler 目錄中，建立名為 index.js 的新檔案。將以下程式碼貼到檔案。該函數會從 API 請求擷取特定屬性，並將其傳回為 JSON 回應。

```
exports.handler = async (event) => {
  // Extract specific properties from the event object
  const { resource, path, httpMethod, headers, queryStringParameters, body } =
 event;
  const response = {
    resource,
    path,
    httpMethod,
    headers,
    queryStringParameters,
    body,
  };
  return {
    body: JSON.stringify(response, null, 2),
    statusCode: 200,
  };
};
```

步驟 4：部署 AWS CDK 堆疊

1. 從專案根目錄執行 cdk synth 命令：

```
cdk synth
```

此命令會從 CDK 堆疊合成 AWS CloudFormation 範本。該範本是大約為 400 行的 YAML 檔案，與下面的內容相似：

> **Note**
>
> 如果您遇到以下錯誤，請確定您位於專案目錄的根目錄。
>
> ```
> --app is required either in command-line, in cdk.json or in ~/.cdk.json
> ```

Example CloudFormation 範本

```
Resources:
  MyFunctionServiceRole3C357FF2:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Statement:
          - Action: sts:AssumeRole
            Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
        Version: "2012-10-17"
      ManagedPolicyArns:
        - Fn::Join:
            - ""
            - - "arn:"
              - Ref: AWS::Partition
              - :iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
    Metadata:
      aws:cdk:path: HelloLambdaStack/MyFunction/ServiceRole/Resource
  MyFunction1BAA52E7:
    Type: AWS::Lambda::Function
    Properties:
      Code:
        S3Bucket:
          Fn::Sub: cdk-hnb659fds-assets-${AWS::AccountId}-${AWS::Region}
        S3Key:
 ab1111111cd32708dc4b83e81a21c296d607ff2cdef00f1d7f48338782f92l3901.zip
      Handler: index.handler
      Role:
```

```
        Fn::GetAtt:
          - MyFunctionServiceRole3C357FF2
          - Arn
      Runtime: nodejs24.x
      ...
```

2. 執行 cdk deploy 命令：

```
cdk deploy
```

等待資源建立完畢。最終輸出包含 API Gateway 端點的 URL。範例：

```
Outputs:
HelloLambdaStack.ApiGwEndpoint77F417B1 = https://abcd1234.execute-api.us-
east-1.amazonaws.com/prod/
```

步驟 5：測試函數

若要調用該 Lambda 函數，請複製 API Gateway 端點並將其貼到 Web 瀏覽器或執行 `curl` 命令：

```
curl -s https://abcd1234.execute-api.us-east-1.amazonaws.com/prod/
```

回應是從原始事件物件所選取屬性的 JSON 表示，其中包含對 API Gateway 端點提出之請求的相關資訊。範例：

```
{
  "resource": "/",
  "path": "/",
  "httpMethod": "GET",
  "headers": {
    "Accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/
webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7",
    "Accept-Encoding": "gzip, deflate, br, zstd",
    "Accept-Language": "en-US,en;q=0.9",
    "CloudFront-Forwarded-Proto": "https",
    "CloudFront-Is-Desktop-Viewer": "true",
    "CloudFront-Is-Mobile-Viewer": "false",
    "CloudFront-Is-SmartTV-Viewer": "false",
    "CloudFront-Is-Tablet-Viewer": "false",
    "CloudFront-Viewer-ASN": "16509",
    "CloudFront-Viewer-Country": "US",
```

```
    "Host": "abcd1234.execute-api.us-east-1.amazonaws.com",
     ...
```

步驟 6：清除您的資源

API Gateway 端點可公開存取。為了避免意外產生費用，請執行 cdk destroy 命令來刪除堆疊和所有相關聯的資源。

```
cdk destroy
```

後續步驟

如需以您選擇的語言撰寫 AWS CDK 應用程式的資訊，請參閱下列內容：

TypeScript

- 以 TypeScript 使用 AWS CDK

JavaScript

- 以 JavaScript 使用 AWS CDK

Python

- 在 Python AWS CDK 中使用

Java

- 在 Java AWS CDK 中使用

C#

- 在 C# AWS CDK 中使用

Go

- 在 Go AWS CDK 中使用

Powertools for AWS Lambda

Powertools for AWS Lambda (也稱為 Powertools for AWS) 提供公用程式函式、裝飾項目和中介軟體，可處理常見的 Lambda 任務，例如結構化日誌記錄、追蹤、指標收集和輸入驗證。使用

Powertools for AWS Lambda 能夠實作無伺服器最佳實務，並加速開發多個 Lambda 函式。這樣做可以簡化 Lambda 函式中的常見開發任務。

Powertools for AWS 的關鍵優勢

儘管沒有 Powertools for AWS 也能進行 Lambda 開發，但使用該工具組能帶來多項優勢：

- 內建可觀測性：結構化日誌記錄、追蹤和自訂指標
- 秘密管理：參數擷取、秘密處理和冪等性
- 漸進式增強：選擇最契合需求的公用程式
- 加速開發：事件剖析、驗證和批次處理
- 最佳實務：實作 AWS Well-Architected 無伺服器模式

將 Powertools 與 AWS 整合

Powertools for AWS 可協助您以更少的自訂程式碼建置生產就緒型無伺服器應用程式。Powertools for AWS 提供 Python、TypeScript/Node.js、.NET 和 Java 版本，您可以透過 Lambda 層或使用語言套件管理工具將其納入。每種語言實作版本均提供結構化日誌記錄、追蹤、指標收集會事件處理等核心功能，同時保留各程式設計語言原生的慣用寫法。這些實作還輔以用於 AWS 服務整合的專用元件，支援參數擷取、批次處理和 API 處理，以及關聯 ID 傳播、錯誤處理和冪等性模式等最佳實務。這些功能相輔相成，能讓開發人員建置強大且可維護的無伺服器應用程式，同時減少自訂程式碼的負擔。

- Powertools for AWS Lambda (Python)
- Powertools for AWS Lambda (TypeScript)
- Powertools for AWS Lambda (Java)
- Powertools for AWS Lambda (.NET)

後續步驟

若要進一步了解如何使用 Powertools for AWS，請參閱下列資源：

- Powertools for AWS Lambda workshop
- Serverless patterns that use Powertools for AWS
- AWS well-architected serverless lens
- Building Serverless APIs with Powertools for AWS Lambda

管理 Lambda 工作流程與事件

使用 Lambda 建置無伺服器應用程式時，您通常需要協調函數執行和處理事件的方法。 AWS 提供數種方法來協調 Lambda 函數：

- Lambda 中用於程式碼優先工作流程協同運作的 Lambda 耐久函數
- AWS Step Functions 用於跨多個 服務的視覺化工作流程協同運作
- 適用於事件驅動型架構和排程的 Amazon EventBridge 排程器和 Amazon EventBridge

您也可以將這些方法整合在一起。例如，您可以使用 EventBridge 排程器在特定事件發生時觸發持久函數或 Step Functions 工作流程，或設定工作流程以在定義的執行點將事件發佈至 EventBridge 排程器。本節中的下列主題提供如何使用這些協同運作選項的詳細資訊。

具有耐用函數的程式碼優先協同運作

Lambda 耐用函數提供工作流程協同運作的程式碼優先方法，可讓您直接在 Lambda 函數內建置具狀態且長時間執行的工作流程。與外部協同運作服務不同，耐用的 函數會將工作流程邏輯保留在程式碼中，讓您更輕鬆地搭配業務邏輯進行版本化、測試和維護。

當您需要時，耐用的函數是理想的選擇：

- 使用標準程式設計語言：使用熟悉的程式設計語言定義工作流程，例如 JavaScript 和 Python
- 長時間執行的程序：執行工作流程，最多可執行一年，遠超過標準 Lambda 函數的 15 分鐘限制
- 簡化開發：將工作流程邏輯和商業邏輯保留在相同的程式碼庫中，以便於維護和測試
- 經濟實惠的等待：在等待狀態期間暫停執行，而不耗用運算資源
- 內建狀態管理：自動檢查點和狀態持續性，無需外部儲存組態

如需在耐用函數與 Step Functions 之間進行選擇的說明，請參閱耐用函數或 Step Functions。

如需耐用函數的詳細資訊，請參閱 Lambda 耐用函數。

使用 Step Functions 協同運作工作流程

AWS Step Functions 是一種工作流程協同運作服務，可協助您將多個 Lambda 函數和其他 AWS 服務協調為結構化工作流程。這些工作流程可以維護狀態、使用複雜的重試機制處理錯誤，以及大規模處理資料。

Step Functions 提供兩種類型的工作流程，可滿足不同的協同運作需求：

標準工作流程

適用於需具備「恰好一次」執行語義、執行時間較長且可稽核的工作流程。標準工作流程最長可以執行一年，提供詳細的執行歷史記錄，並支援視覺化偵錯功能。此類型適用於訂單履行、資料處理管道或多步驟分析任務等流程。

快速工作流程

專為具備「至少一次」執行語義、高事件量且執行時間短的工作負載而設計。快速工作流程最長可執行五分鐘，非常適合大量事件處理、串流資料轉換或 IoT 資料擷取案例。相較於標準工作流程，此類型提供的輸送量更高，成本可能更低。

> **Note**
>
> 如需有關 Step Functions 工作流程類型的詳細資訊，請參閱 Choosing workflow type in Step Functions。

在此類工作流程中，Step Functions 提供兩種類型的映射狀態，用於實現平行處理：

內嵌映射

在父工作流程的執行歷史記錄中處理來自 JSON 陣列的項目。內嵌映射最多支援 40 個並行迭代，適用於較小的資料集或需要將所有處理保持在單次執行內的情況。如需詳細資訊，請參閱 Using Map state in Inline mode。

分散式映射

透過迭代處理超過 256 KiB 或需要超過 40 個並行迭代的資料集，實現對大規模平行工作負載的處理。分散式映射可支援最多 10,000 個平行子工作流程執行，擅長處理儲存在 Amazon S3 中的半結構化資料 (例如 JSON 或 CSV 檔案)，因此非常適合批次處理與 ETL 操作。如需詳細資訊，請參閱 Using Map state in Distributed mode。

透過結合這些工作流程類型與映射狀態，Step Functions 提供了靈活且強大的工具集，能夠從小型操作到大型資料處理管道，全面協同運作複雜的無伺服器應用程式。

若要開始將 Lambda 與 Step Functions 搭配使用，請參閱使用 Step Functions 協同運作 Lambda 函式。

使用 EventBridge 與 EventBridge 排程器管理事件

Amazon EventBridge 是一項事件匯流排服務，可協助建置事件驅動型架構。它會路由 AWS 服務、整合應用程式和軟體即服務 (SaaS) 應用程式之間的事件。EventBridge 排程器是一種無伺服器排程器，可讓您從一個中央服務建立、執行並管理任務，允許您使用 Cron 和 Rate 表達式依排程調用 Lambda 函式，或設定一次性調用函式。

Amazon EventBridge 和 EventBridge 排程器可協助使用 Lambda 建置事件驅動型架構。EventBridge 會路由 AWS 服務、整合應用程式和 SaaS 應用程式之間的事件，而 EventBridge 排程器則提供特定排程功能，以定期或一次性叫用 Lambda 函數。

這些服務為使用 Lambda 函式提供數種關鍵功能：

- 使用 EventBridge 建立規則來比對事件，並將其路由至 Lambda 函式
- 使用 EventBridge 排程器設定以 Cron 與 Rate 表達式驅動的週期性函式調用
- 在特定日期和時間設定一次性函式調用
- 為排程調用定義彈性時段與重試政策

如需詳細資訊，請參閱依排程調用 Lambda 函數。

Lambda 耐用函數

Lambda 耐用函數可讓您建置彈性的多步驟應用程式和 AI 工作流程，可執行長達一年，同時在中斷的情況下保持可靠的進度。當持久性函數執行時，這個完整的生命週期稱為持久性執行，它使用檢查點來追蹤進度，並透過重新執行自動從失敗中復原，從頭開始重新執行，同時略過完成的工作。

在每個函數中，您會使用耐用的操作做為基本的建置區塊。步驟會使用內建重試和進度追蹤來執行商業邏輯，同時等待暫停執行而不會產生運算費用，因此非常適合長時間執行的程序human-in-the-loop工作流程或輪詢外部相依性。無論您是在處理訂單、協調微服務或協調代理式 AI 應用程式，當您使用熟悉的程式設計語言編寫程式碼時，耐用的函數會自動維持狀態並從失敗中復原。

主要優點

自然撰寫彈性程式碼：使用熟悉的程式設計建構，您可以撰寫自動處理失敗的程式碼。內建檢查點、透明重試和自動復原意味著您的商業邏輯保持乾淨且專注。

只需為您使用的項目付費：在等待操作期間，您的函數會暫停，而不會產生運算費用。對於等待數小時或數天的長時間執行工作流程，您只需支付實際處理時間，而非閒置等待。

操作簡單：使用 Lambda 的無伺服器模型，您可以自動擴展，包括scale-to-zero，而無需管理基礎設施。耐用的函數會自動處理狀態管理、重試邏輯和故障復原，從而降低營運開銷。

使用耐用函數的時機

短期協調：透過自動轉返失敗，協調跨多項服務的付款、庫存和運送。透過驗證、付款授權、庫存分配和履行來處理訂單，並保證完成。

安心處理付款：建立彈性付款流程，透過失敗維持交易狀態，並自動處理重試。協調跨付款供應商的多步驟授權、詐騙檢查和和解，並跨步驟進行完整稽核。

建置可靠的 AI 工作流程：建立多步驟 AI 工作流程，以鏈結模型呼叫、整合人工意見回饋，並在失敗期間果斷地處理長時間執行的任務。停用後自動繼續，且只需支付作用中執行時間的費用。

協調複雜的訂單履行：使用內建彈性協調跨庫存、付款、運送和通知系統的訂單處理。自動處理部分故障、保留中斷時的順序狀態，並有效率地等待外部事件，而不會耗用運算資源。

自動化多步驟業務工作流程：為員工加入、貸款核准和跨越數天或數週的合規程序建立可靠的工作流程。維護跨人工核准、系統整合和排程任務的工作流程狀態，同時提供程序狀態和歷史記錄的完整可見性。

與 Step Functions 比較的耐久性函數

耐用的函數和 Step Functions 都提供具有自動狀態管理的工作流程協同運作。主要差異在於它們的執行位置，以及您如何定義工作流程：

- 耐用函數：在 Lambda 內執行、使用標準程式設計語言、在 Lambda 環境中管理
- Step Functions：單機服務、圖形式 DSL 或視覺化設計工具，完全受管，無需維護

耐用的函數非常適合在 Lambda 中應用程式開發，其中工作流程與商業邏輯緊密結合。Step Functions 擅長跨您需要視覺化設計、原生整合超過 220 個服務以及零維護基礎設施 AWS 的服務進行工作流程協同運作。

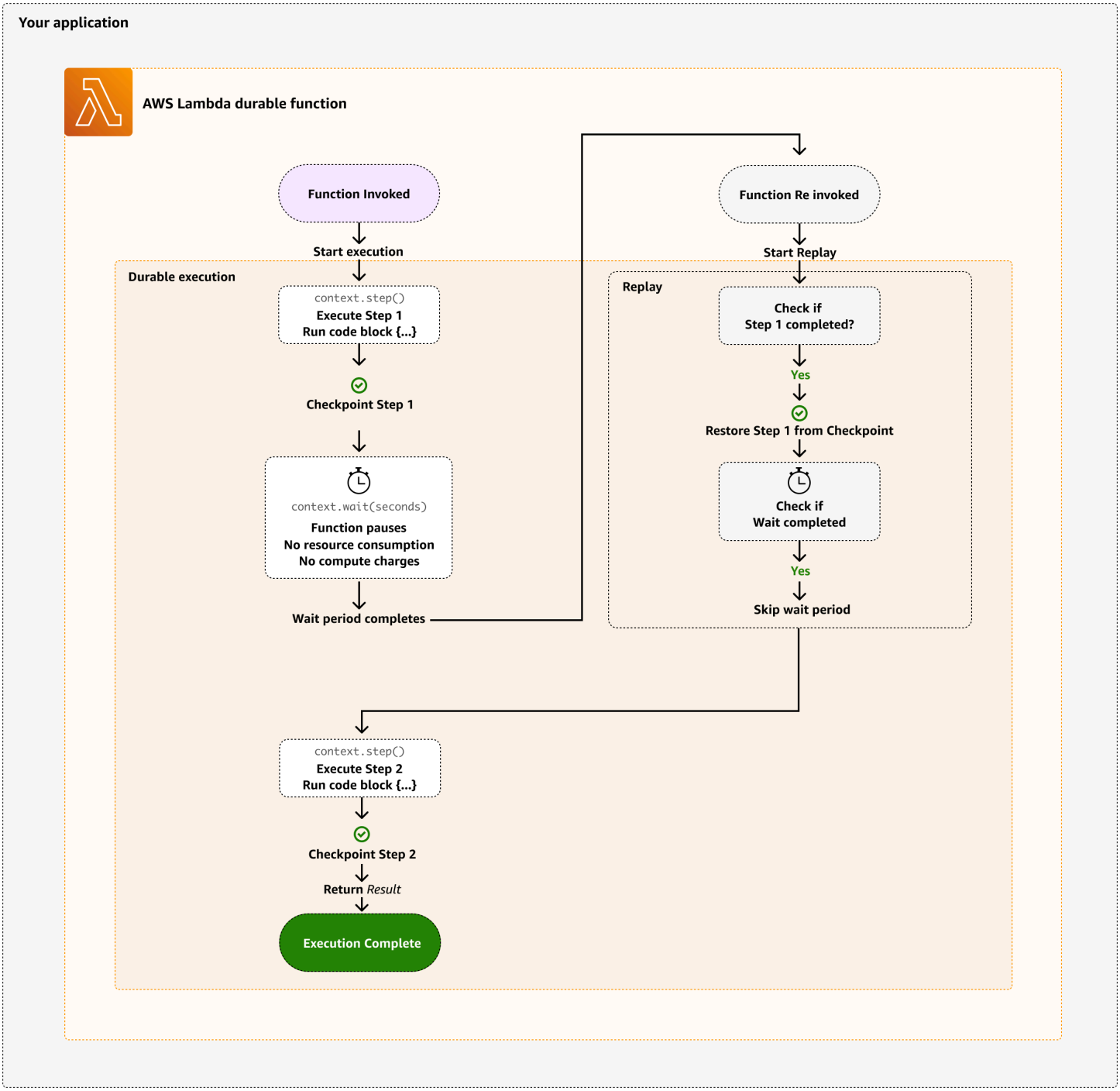
如需詳細比較，請參閱耐用函數或 Step Functions。

運作方式

在幕後，耐用的函數是一般的 Lambda 函數，使用檢查點/重播機制來追蹤進度，並透過使用者定義的暫停點支援長時間執行的操作，通常稱為耐用執行。函數從暫停或中斷恢復後，系統會執行重播。在重播期間，您的程式碼會從頭開始執行，但會略過已完成的檢查點，使用儲存的結果，而不是重新執行已完成的操作。此重播機制可確保一致性，同時啟用長時間執行的執行。

為了在您的應用程式中利用此checkpoint-and-replay機制，Lambda 提供耐用的執行 SDK。開發套件可消除管理檢查點和重播的複雜性，公開您在程式碼中使用的稱為耐久操作的簡單基本概念。開發套件適用於 JavaScript、TypeScript、Python 和 Java（預覽版），可與您現有的 Lambda 開發工作流程無縫整合。

使用 SDK，您可以包裝 Lambda 事件處理常式，然後提供 DurableContext 與您的事件並行。此內容可讓您存取耐用的操作，例如步驟和等待。您可以將函數邏輯寫入為一般循序程式碼，但不會直接呼叫服務，而是將這些呼叫包裝為自動檢查點和重試的步驟。當您需要暫停執行時，您可以新增暫停函數的等待，而不會產生費用。開發套件可處理所有複雜的狀態管理和幕後重播，讓您的程式碼保持乾淨且可讀取。



後續步驟

- 開始使用耐用的 函數
- 探索耐用的執行 SDK
- 耐用函數或 Step Functions

- 監控和偵錯耐用函數
- 檢閱安全性和許可
- 遵循最佳實務

基本概念

Lambda 提供適用於 JavaScript、TypeScript 和 Python SDKs。這些 SDKs是建置耐用函數的基礎，提供您檢查點進度、處理重試和管理執行流程所需的基本概念。如需完整的 SDK 文件和範例，請參閱 GitHub 上的 JavaScript/TypeScript SDK 和 Python SDK。

持久性執行

持久性執行代表 Lambda 持久性函數的完整生命週期，使用檢查點和重播機制來追蹤業務邏輯進度、暫停執行並從失敗中復原。當函數在暫停或中斷後恢復時，會重新播放先前完成的檢查點，且函數會繼續執行。

生命週期可能包含 Lambda 函數的多次調用以完成執行，特別是在暫停或故障復原之後。此方法可讓您的函數長時間執行（長達一年），同時在中斷的情況下維持可靠的進度。

重播的運作方式

Lambda 會在函數執行時保留所有耐久操作（步驟、等待和其他操作）的執行中日誌。當您的函數需要暫停或遇到中斷時，Lambda 會儲存此檢查點日誌並停止執行。恢復時，Lambda 會從頭再次叫用您的函數，並重播檢查點日誌，以儲存的值取代已完成的操作。這表示您的程式碼會再次執行，但先前完成的步驟不會重新執行。而是改用其儲存的結果。

此重播機制對於了解持久功能至關重要。您的程式碼在重播期間必須是確定性的，這表示它在相同的輸入下會產生相同的結果。避免在步驟外出現副作用（例如產生隨機數字或取得目前時間）的操作，因為這些操作可能會在重播期間產生不同的值，並導致非確定性的行為。

DurableContext

DurableContext 是您耐用函數接收的內容物件。它提供持久性操作的方法，例如建立檢查點和管理執行流程的步驟和等待。

您的耐用函數會收到 `DurableContext`而非預設 Lambda 內容：

TypeScript

```
import {
```

```
  DurableContext,
  withDurableExecution,
} from "@aws/durable-execution-sdk-js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const result = await context.step(async () => {
      return "step completed";
    });
    return result;
  },
);
```

Python

```
from aws_durable_execution_sdk_python import (
    DurableContext,
    durable_execution,
    durable_step,
)

@durable_step
def my_step(step_context, data):
    # Your business logic
    return result

@durable_execution
def handler(event, context: DurableContext):
    result = context.step(my_step(event["data"]))
    return result
```

適用於耐用函數的 Python SDK 使用同步方法，不支援 `await`。TypeScript SDK 使用 `async/await`。

步驟

步驟會使用內建重試和自動檢查點來執行商業邏輯。每個步驟都會儲存其結果，確保您的函數可以在中斷後從任何已完成的步驟恢復。

TypeScript

```
// Each step is automatically checkpointed
```

```
const order = await context.step(async () => processOrder(event));
const payment = await context.step(async () => processPayment(order));
const result = await context.step(async () => completeOrder(payment));
```

Python

```
# Each step is automatically checkpointed
order = context.step(lambda: process_order(event))
payment = context.step(lambda: process_payment(order))
result = context.step(lambda: complete_order(payment))
```

等待狀態

等待狀態是您的函數停止執行（並停止充電）的計劃暫停，直到繼續為止。使用它們來等待期間、外部回呼或特定條件。

TypeScript

```
// Wait for 1 hour without consuming resources
await context.wait({ seconds:3600 });

// Wait for external callback
const approval = await context.waitForCallback(
  async (callbackId) => sendApprovalRequest(callbackId)
);
```

Python

```
# Wait for 1 hour without consuming resources
context.wait(3600)

# Wait for external callback
approval = context.wait_for_callback(
    lambda callback_id: send_approval_request(callback_id)
)
```

當您的函數遇到等待或需要暫停時，Lambda 會儲存檢查點日誌並停止執行。恢復時，Lambda 會再次叫用您的函數，並重播檢查點日誌，以儲存的值取代已完成的操作。

對於更複雜的工作流程，耐用的 Lambda 函數也隨附進階操作，例如`parallel()`用於並行執行、`map()`用於處理陣列、`runInChildContext()`用於巢狀操作，以及`waitForCondition()`用於輪詢。如需何時使用每項操作的詳細範例和指引，請參閱範例。

叫用其他 函數

調用允許耐用的函數呼叫其他 Lambda 函數，並等待其結果。呼叫函數會在調用函數執行時暫停，建立保留結果的檢查點。這可讓您建置模組化工作流程，讓專門的 函數處理特定任務。

使用 `context.invoke()`從您的耐用函數中呼叫其他函數。調用是檢查點的，因此如果您的函數在調用函數完成後中斷，它會繼續以儲存的結果繼續，而不會重新調用函數。

TypeScript

```
// Invoke another function and wait for result
const customerData = await context.invoke(
  'validate-customer',
  'arn:aws:lambda:us-east-1:123456789012:function:customer-service:1',
  { customerId: event.customerId }
);

// Use the result in subsequent steps
const order = await context.step(async () => {
  return processOrder(customerData);
});
```

Python

```
# Invoke another function and wait for result
customer_data = context.invoke(
    'arn:aws:lambda:us-east-1:123456789012:function:customer-service:1',
    {'customerId': event['customerId']},
    name='validate-customer'
)

# Use the result in subsequent steps
order = context.step(
    lambda: process_order(customer_data),
    name='process-order'
)
```

調用函數可以是耐用或標準 Lambda 函數。如果您叫用持久性函數，呼叫函數會等待完全持久性執行完成。此模式在微服務架構中很常見，其中每個函數都會處理特定網域，讓您從專業、可重複使用的函數編寫複雜的工作流程。

> **Note**
>
> 不支援跨帳戶調用。調用函數必須與呼叫函數位於相同的 AWS 帳戶中。

耐用的函數組態

耐用函數具有特定的組態設定，可控制執行行為和資料保留。這些設定與標準 Lambda 函數組態分開，並套用至整個耐久執行生命週期。

DurableConfig 物件會定義耐用函數的組態：

```
{
  "ExecutionTimeout": Integer,
  "RetentionPeriodInDays": Integer
}
```

Execution timeout (執行逾時)

執行逾時可控制持久性執行從開始到完成的時間長度。這與 Lambda 函數逾時不同，它控制單一函數調用可以執行的時間長度。

當 Lambda 函數通過檢查點、等待和重播時，持久的執行可以跨越多個 Lambda 函數調用。執行逾時適用於持久性執行的總經過時間，不適用於個別函數叫用。

了解差異

Lambda 函數逾時（最長 15 分鐘）會限制函數的每個個別調用。持久的執行逾時（最長 1 年）會限制從執行開始到完成、失敗或逾時的總時間。在此期間，您的函數可能會在處理步驟、等待並從失敗中復原時被叫用多次。

例如，如果您將持久的執行逾時設定為 24 小時，Lambda 函數逾時設定為 5 分鐘：

- 每個函數調用必須在 5 分鐘內完成
- 整個耐久執行最多可執行 24 小時
- 在這 24 小時內，您的函數可以被叫用多次

- 等待操作不會計入 Lambda 函數逾時，但會計入執行逾時

您可以使用 Lambda 主控台或 建立耐用函數時 AWS CLI，設定執行逾時 AWS SAM。在 Lambda 主控台中，選擇您的函數，然後選擇組態、持久性執行。以秒為單位設定執行逾時值 （預設值：86400 秒 / 24 小時，最小值：60 秒，最大值：31536000 秒 / 1 年）。

> **Note**
>
> 執行逾時和 Lambda 函數逾時的設定不同。Lambda 函數逾時控制每個個別調用可以執行的時間長度 （最長 15 分鐘）。執行逾時控制整個持久性執行的總經過時間 （最長 1 年）。

保留期間

保留期間控制 Lambda 在持久性執行完成後保留執行歷史記錄和檢查點資料的時間長度。此資料包含步驟結果、執行狀態和完整的檢查點日誌。

保留期間到期後，Lambda 會刪除執行歷史記錄和檢查點資料。您無法再擷取執行詳細資訊或重播執行。保留期會在執行達到結束狀態 (SUCCEEDED、FAILED、STOPPED 或 TIMED_OUT) 時開始。

您可以使用 Lambda 主控台 AWS CLI或 來設定建立耐用函數時的保留期間 AWS SAM。在 Lambda 主控台中，選擇您的函數，然後選擇組態、持久性執行。以天為單位設定保留期間值 （預設值：14 天，下限：1 天，上限：90 天）。

根據您的合規要求、除錯需求和成本考量，選擇保留期。較長的保留期可提供更多偵錯和稽核的時間，但會增加儲存成本。

另請參閱

- 耐用函數或 Step Functions – 比較耐用函數與 Step Functions，以了解每個方法最有效的時機。

建立 Lambda 耐用函數

若要開始使用 Lambda 耐用函數，請使用 Lambda 主控台建立耐用函數。在幾分鐘內，您可以建立和部署耐久的 函數，該函數使用步驟 並等待示範檢查點型執行。

執行教學課程時，您將學習基本的耐用函數概念，例如如何使用 `DurableContext` 物件、使用步驟建立檢查點，以及使用等待暫停執行。您也將了解當您的函數在等待後恢復時，重播的運作方式。

為了保持簡單，本教學課程說明如何使用 Python 或 Node.js 執行時間來建立函數。您可以使用這些轉譯語言，直接在主控台的內建程式碼編輯器中編輯函數程式碼。

Java （預覽版）中的耐用函數目前只能透過容器映像部署。如需從容器映像建立持久性函數的詳細資訊，請參閱持久性函數支援的執行時間，或使用基礎設施即程式碼部署 Lambda 持久性函數。

> **Note**
>
> 耐用函數目前支援 Python 和 Node.js (JavaScript/TypeScript) 執行期和容器映像 (OCI)，例如 Java。如需支援執行時間版本和容器映像選項的完整清單，請參閱耐用函數支援的執行時間。如需搭配 Lambda 使用容器映像的詳細資訊，請參閱《Lambda 開發人員指南》中的建立 Lambda 容器映像。

> **Tip**
>
> 若要了解如何建置無伺服器解決方案，請參閱無伺服器開發人員指南。

先決條件

註冊 AWS 帳戶

如果您沒有 AWS 帳戶，請完成下列步驟來建立一個。

註冊 AWS 帳戶

1. 開啟 https://portal.aws.amazon.com/billing/signup。
2. 請遵循線上指示進行。

   部分註冊程序需接收來電或簡訊，並在電話鍵盤輸入驗證碼。

   當您註冊 時 AWS 帳戶，AWS 帳戶根使用者會建立 。根使用者有權存取該帳戶中的所有 AWS 服務 和資源。作為安全最佳實務，請將管理存取權指派給使用者，並且僅使用根使用者來執行需要根使用者存取權的任務。

AWS 會在註冊程序完成後傳送確認電子郵件給您。您可以隨時登錄 https://aws.amazon.com/ 並選擇我的帳戶，以檢視您目前的帳戶活動並管理帳戶。

建立具有管理存取權的使用者

註冊 後 AWS 帳戶，請保護 AWS 帳戶根使用者、啟用 AWS IAM Identity Center和建立管理使用者，以免將根使用者用於日常任務。

保護您的 AWS 帳戶根使用者

1. 選擇根使用者並輸入 AWS 帳戶 您的電子郵件地址，以帳戶擁有者AWS 管理主控台身分登入 。在下一頁中，輸入您的密碼。

   如需使用根使用者登入的說明，請參閱 AWS 登入 使用者指南中的以根使用者身分登入。

2. 若要在您的根使用者帳戶上啟用多重要素驗證 (MFA)。

   如需說明，請參閱《IAM 使用者指南》中的為您的 AWS 帳戶 根使用者 （主控台） 啟用虛擬 MFA 裝置。

建立具有管理存取權的使用者

1. 啟用 IAM Identity Center。

   如需指示，請參閱《AWS IAM Identity Center 使用者指南》中的啟用 AWS IAM Identity Center。

2. 在 IAM Identity Center 中，將管理存取權授予使用者。

   如需使用 IAM Identity Center 目錄 做為身分來源的教學課程，請參閱AWS IAM Identity Center《 使用者指南》中的使用預設值設定使用者存取 IAM Identity Center 目錄。

以具有管理存取權的使用者身分登入

- 若要使用您的 IAM Identity Center 使用者簽署，請使用建立 IAM Identity Center 使用者時傳送至您電子郵件地址的簽署 URL。

  如需使用 IAM Identity Center 使用者登入的說明，請參閱AWS 登入 《 使用者指南》中的登入 AWS 存取入口網站。

指派存取權給其他使用者

1. 在 IAM Identity Center 中，建立一個許可集來遵循套用最低權限的最佳實務。

   如需指示，請參閱《AWS IAM Identity Center 使用者指南》中的建立許可集。

2. 將使用者指派至群組，然後對該群組指派單一登入存取權。

   如需指示，請參閱《AWS IAM Identity Center 使用者指南》中的新增群組。

使用主控台建立 Lambda 耐用函數

在此範例中，您的耐用函數會使用自動檢查點處理多個步驟的訂單。函數會取得包含訂單 ID 的 JSON 物件、驗證訂單、處理付款，以及確認訂單。每個步驟都會自動檢查點，因此如果函數中斷，則會從上次完成的步驟繼續執行。

您的函數也會示範等待操作，暫停執行一小段時間以模擬等待外部確認。

使用主控台建立耐用的函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇建立函數。
3. 選取從頭開始撰寫。
4. 在基本資訊窗格中，為函數名稱輸入 **myDurableFunction**。
5. 針對執行期，選擇 Node.js 24 或 Python 3.14。
6. 選取啟用持久性執行。

Lambda 會使用包含檢查點操作 (`lambda:CheckpointDurableExecutions` 和) 許可的執行角色來建立您的耐用函數`lambda:GetDurableExecutionState`。

> **Note**
>
> Lambda 執行時間包含持久性執行 SDK，因此您可以測試持久性函數，而無需封裝相依性。不過，我們建議您在部署套件中包含 開發套件以供生產使用。這可確保版本一致性，並避免可能影響函數的潛在執行時間更新。

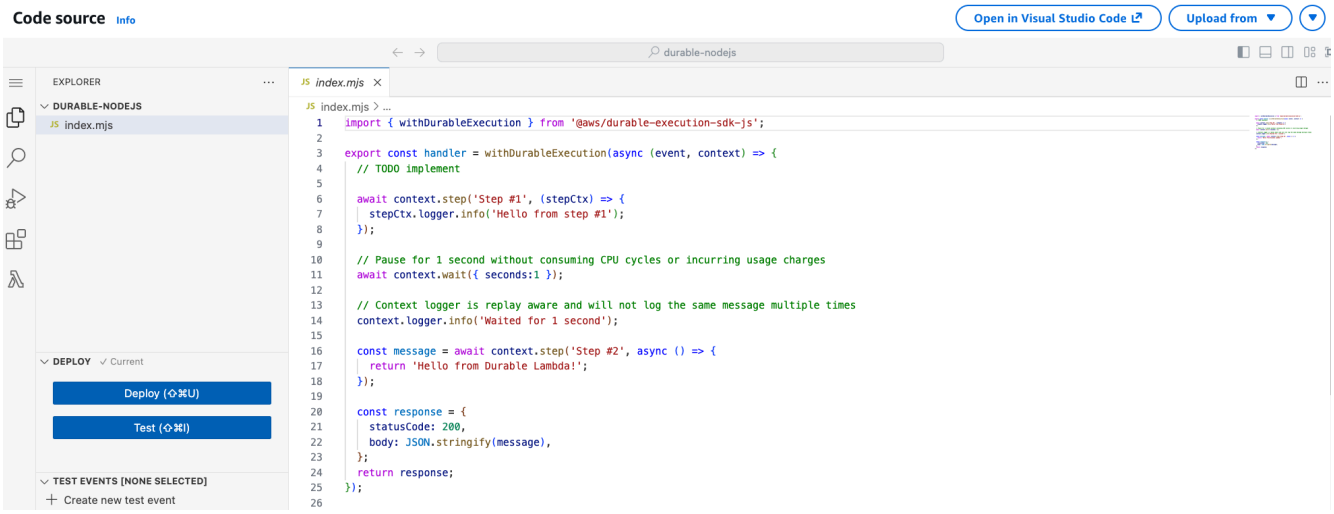
使用主控台的內建程式碼編輯器來新增您的耐用函數程式碼。

Node.js

若要在主控台中修改程式碼

1. 選擇 程式碼 標籤。

在主控台的內建程式碼編輯器中，您應該會看到 Lambda 建立的函數程式碼。如果您在程式碼編輯器中沒看到 index.mjs 標籤，請在檔案總管中選取 index.mjs，如下圖所示。

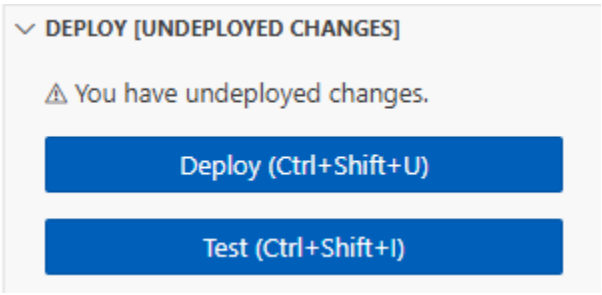


2. 將以下程式碼貼到 index.mjs 標籤中，替換 Lambda 建立的程式碼。

```
import {
  DurableContext,
  withDurableExecution,
} from "@aws/durable-execution-sdk-js";

export const handler = withDurableExecution(
  async (event, context) => {
    const orderId = event.orderId;

    // Step 1: Validate order
    const validationResult = await context.step(async (stepContext) => {
      stepContext.logger.info(`Validating order ${orderId}`);
      return { orderId, status: "validated" };
    });

    // Step 2: Process payment
    const paymentResult = await context.step(async (stepContext) => {
      stepContext.logger.info(`Processing payment for order ${orderId}`);
      return { orderId, status: "paid", amount: 99.99 };
    });

    // Wait for 10 seconds to simulate external confirmation
    await context.wait({ seconds: 10 });

    // Step 3: Confirm order
```

```
    const confirmationResult = await context.step(async (stepContext) => {
      stepContext.logger.info(`Confirming order ${orderId}`);
      return { orderId, status: "confirmed" };
    });

    return {
      orderId: orderId,
      status: "completed",
      steps: [validationResult, paymentResult, confirmationResult]
    };
  }
);
```

3. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



了解您的耐用函數程式碼

在進行下一個步驟之前，讓我們先查看函數程式碼，並了解關鍵的耐久函數概念。

- `withDurableExecution` 包裝函式：

  您的耐用函數會包裝在 中`withDurableExecution`。此包裝函式提供 `DurableContext` 物件和管理檢查點操作，以啟用持久的執行。

- `DurableContext` 物件：

  您的函數會收到 ，而不是標準 Lambda 內容`DurableContext`。此物件提供建立檢查點等耐久操作的方法`wait()`，例如 `step()`和 。

- 步驟和檢查點：

  每個`context.step()`呼叫都會在執行前後建立檢查點。如果您的函數中斷，則會從上次完成的檢查點繼續執行。函數不會重新執行完成的步驟。它會改用其儲存的結果。

- 等待操作：

`context.wait()` 呼叫會暫停執行，而不會耗用運算資源。當等待完成時，Lambda 會再次叫用您的函數，並重播檢查點日誌，以儲存的值取代已完成的步驟。

- 重播機制：

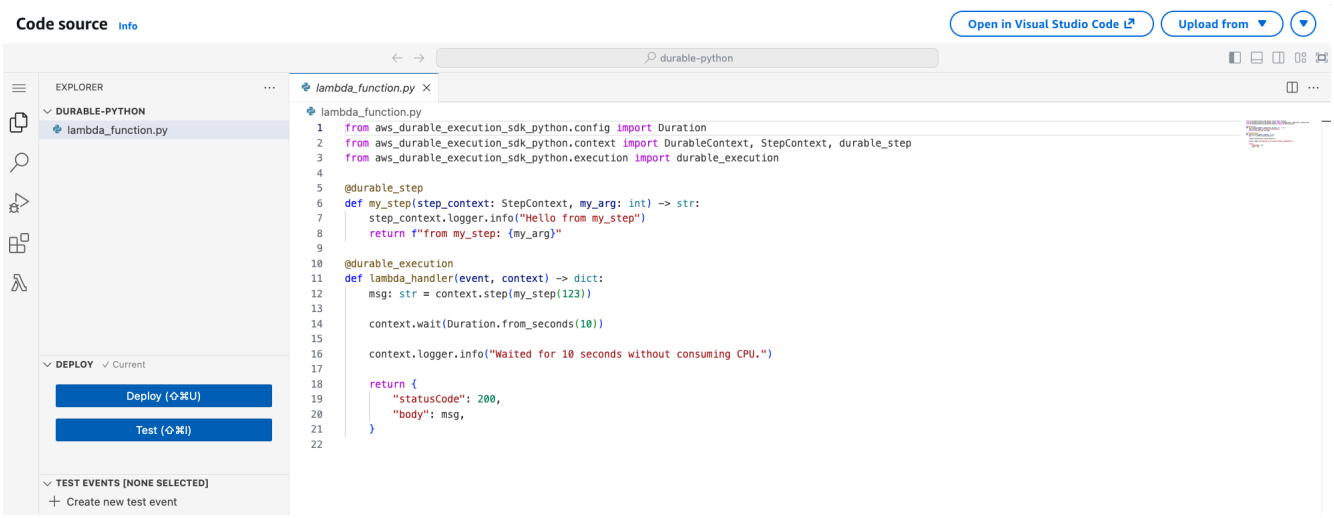
  當您的函數在等待或中斷後繼續時，Lambda 會從頭開始執行您的程式碼。不過，已完成的步驟不會重新執行。Lambda 會從檢查點日誌重播其結果。這就是您的程式碼必須具有決定性的原因。

Python

若要在主控台中修改程式碼

1. 選擇 程式碼 標籤。

   在主控台的內建程式碼編輯器中，您應該會看到 Lambda 建立的函數程式碼。如果您在程式碼編輯器中沒看到 lambda_function.py，請在檔案總管中選取 lambda_function.py，如下圖所示。

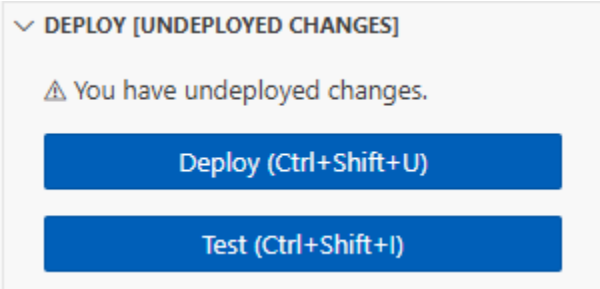


2. 將以下程式碼貼到 lambda_function.py 標籤中，替換 Lambda 建立的程式碼。

```
from aws_durable_execution_sdk_python import (
    DurableContext,
    durable_execution,
    durable_step,
)
from aws_durable_execution_sdk_python.config import Duration

@durable_step
```

```
def validate_order(step_context, order_id):
    step_context.logger.info(f"Validating order {order_id}")
    return {"orderId": order_id, "status": "validated"}

@durable_step
def process_payment(step_context, order_id):
    step_context.logger.info(f"Processing payment for order {order_id}")
    return {"orderId": order_id, "status": "paid", "amount": 99.99}

@durable_step
def confirm_order(step_context, order_id):
    step_context.logger.info(f"Confirming order {order_id}")
    return {"orderId": order_id, "status": "confirmed"}

@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']

    # Step 1: Validate order
    validation_result = context.step(validate_order(order_id))

    # Step 2: Process payment
    payment_result = context.step(process_payment(order_id))

    # Wait for 10 seconds to simulate external confirmation
    context.wait(Duration.from_seconds(10))

    # Step 3: Confirm order
    confirmation_result = context.step(confirm_order(order_id))

    return {
        "orderId": order_id,
        "status": "completed",
        "steps": [validation_result, payment_result, confirmation_result]
    }
```

3. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



了解您的耐用函數程式碼

在進行下一個步驟之前，讓我們先查看函數程式碼，並了解關鍵的耐久函數概念。

- `@durable_execution` 裝飾項目：

  您的處理常式函數以 裝飾`@durable_execution`。此裝飾項目透過提供 `DurableContext` 物件和管理檢查點操作來實現持久的執行。

- `@durable_step` 裝飾項目：

  每個步驟函數都以 裝飾`@durable_step`。此裝飾項目會將函數標記為建立檢查點的耐久步驟。

- `DurableContext` 物件：

  您的函數會收到 ，而不是標準 Lambda 內容`DurableContext`。此物件提供建立檢查點等耐久操作的方法`wait()`，例如 `step()`和 。

- 步驟和檢查點：

  每個`context.step()`呼叫都會在執行前後建立檢查點。如果您的函數中斷，則會從上次完成的檢查點繼續執行。函數不會重新執行完成的步驟。它會改用其儲存的結果。

- 等待操作：

  `context.wait()` 呼叫會暫停執行，而不會耗用運算資源。當等待完成時，Lambda 會再次叫用您的函數，並重播檢查點日誌，以儲存的值取代已完成的步驟。

- Python SDK 是同步的：

  請注意，Python SDK 不使用 `await`。所有耐用的操作都是同步方法呼叫。

使用主控台程式碼編輯器叫用耐用函數

未指定明確版本（或發佈）時，主控台會使用$LATEST版本限定詞叫用耐久函數。不過，為了決定性執行程式碼，您必須一律使用指向穩定版本的合格 ARN。

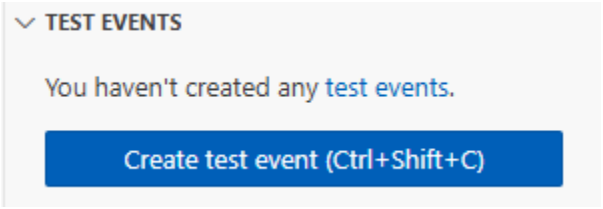
發佈函數的版本

1. 選擇版本索引標籤。
2. 選擇 Publish new version (發佈新版本)。
3. 針對版本描述，輸入 **`Initial version`**（選用）。
4. 選擇發布。
5. Lambda 會建立函數的第 1 版。請注意，函數 ARN 現在:1會在結尾包含 ，表示這是第 1 版。

現在建立測試事件以傳送至您的 函數。事件是包含訂單 ID 的 JSON 格式文件。

若要建立測試事件

1. 在主控台程式碼編輯器的 TEST EVENTS 區段中，選擇建立測試事件。



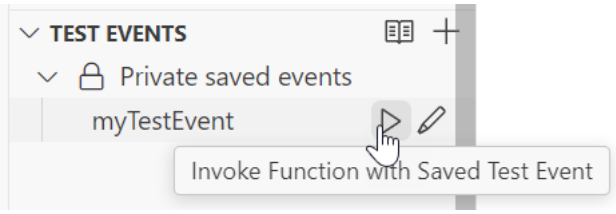
2. Event Name (事件名稱) 輸入 **`myTestEvent`**。
3. 在事件 JSON 區段中，將預設 JSON 取代為下列項目：

```
{
  "orderId": "order-12345"
}
```

4. 選擇儲存。

測試您的耐用函數並檢視執行

在主控台程式碼編輯器的 TEST EVENTS 區段中，選擇測試事件旁邊的執行圖示：



您的耐用函數會開始執行。因為它包含 10 秒的等待時間，初始調用會快速完成，而且函數會在等待期間之後繼續。您可以在持久性執行索引標籤中檢視執行進度。

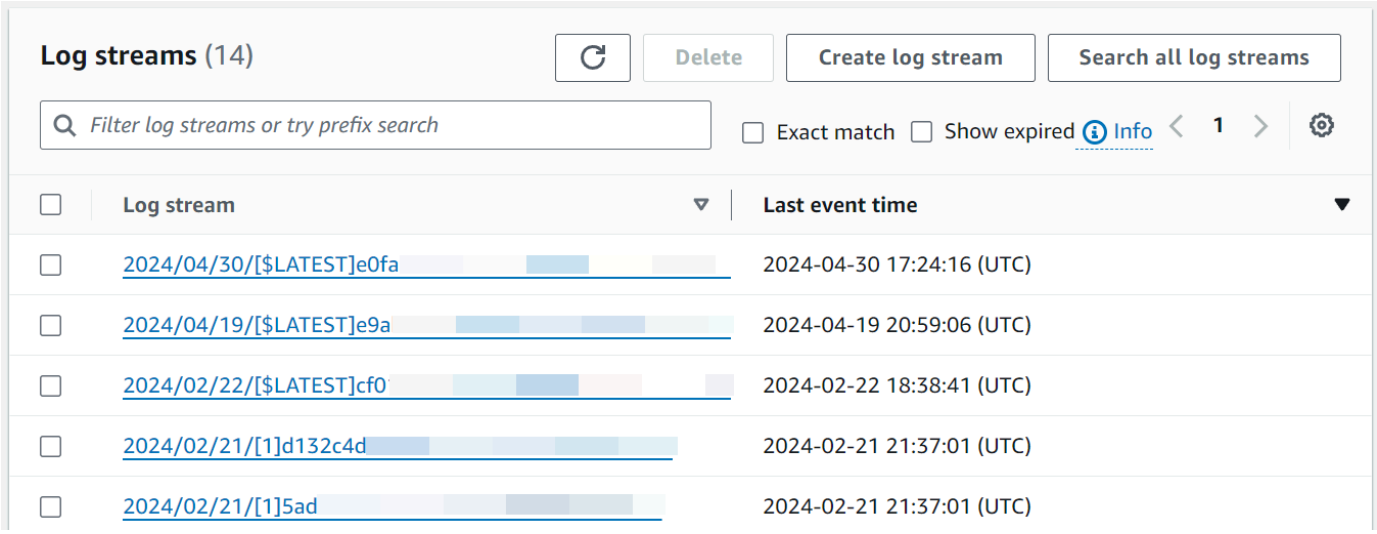
檢視您的耐久函數執行

1. 選擇持久性執行索引標籤。
2. 在清單中尋找您的執行。執行會顯示目前狀態（執行中、成功或失敗）。
3. 選擇執行 ID 以檢視詳細資訊，包括：
   - 顯示每個步驟完成時間的執行時間表
   - 檢查點歷史記錄
   - 等待期間
   - 步驟結果

您也可以在 CloudWatch Logs 中檢視函數的日誌，以查看每個步驟的主控台輸出。

若要在 CloudWatch Logs 中檢視函數的調用記錄

1. 開啟 CloudWatch 主控台的 日誌群組 頁面。
2. 為函數 (`/aws/lambda/myDurableFunction`) 選擇日誌群組名稱。
3. 向下捲動並選擇要查看的函數調用日誌串流。



您應該會看到每次叫用函數的日誌項目，包括初始執行和等待後的重播。

> **Note**
>
> 當您從 DurableContext（例如 context.logger或 stepContext.logger）使用記錄器時，日誌也會出現在 Lambda 主控台的耐久執行和步驟檢視中。這些日誌可能需要一些時間才能載入。

清除

當您完成使用範例耐用函數時，請將其刪除。您還可以刪除存放函數日誌的日誌群組，以及主控台建立的執行角色。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **confirm**，然後選擇刪除。

刪除日誌群組

1. 開啟 CloudWatch 主控台的 日誌群組 頁面。

2. 選取函數的日誌群組 (/aws/lambda/myDurableFunction)。
3. 選擇 動作 、刪除日誌群組 。
4. 在 刪除日誌群組 對話方塊中，選擇 刪除 。

刪除執行角色

1. 開啟 AWS Identity and Access Management (IAM) 主控台的角色頁面。
2. 選取函數的執行角色，(例如 myDurableFunction-role-*31exxmpl*)。
3. 選擇 刪除。
4. 在刪除角色對話方塊中輸入角色名稱，然後選擇刪除。

其他資源和後續步驟

現在您已使用 主控台建立並測試簡單的耐用函數，請採取下列後續步驟：

- 了解耐用函數的常見使用案例，包括分散式交易、訂單處理和人工審核工作流程。請參閱範例。
- 了解如何使用 CloudWatch 指標和執行歷史記錄監控持久的函數執行。請參閱監控和偵錯。
- 了解如何同步和非同步叫用耐久函數，以及管理長時間執行的執行。請參閱叫用耐用函數。
- 遵循撰寫確定性程式碼、管理檢查點大小和最佳化成本的最佳實務。請參閱最佳實務。
- 了解如何在本機和雲端測試耐用的函數。請參閱測試耐用函數。
- 比較耐用的函數與 Step Functions，以了解每個方法最有效的時機。請參閱耐用函數或 Step Functions。

使用 部署和叫用 Lambda 耐用函數 AWS CLI

使用 AWS CLI 以使用命令建立和部署 Lambda 耐用函數。此方法可讓您直接控制部署程序的每個步驟。

先決條件

- 安裝及設定 AWS CLI。如需說明，請參閱安裝 AWS CLI。
- 使用函數程式碼和耐用的執行 SDK 建立部署套件。
- 建立具有檢查點許可的 IAM 執行角色。

建立執行角色

建立具有基本 Lambda 執行和檢查點操作許可的 IAM 角色。

建立執行角色

1. 建立允許 Lambda 擔任角色的信任政策文件。將此儲存為 `trust-policy.json`：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

2. 建立角色：

```
aws iam create-role \
  --role-name durable-function-role \
  --assume-role-policy-document file://trust-policy.json
```

3. 連接檢查點操作和基本執行的耐久執行政策：

```
aws iam attach-role-policy \
  --role-name durable-function-role \
  --policy-arn arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicDurableExecutionRolePolicy
```

`AWSLambdaBasicDurableExecutionRolePolicy` 受管政策包含檢查點操作 (`lambda:CheckpointDurableExecutions` 和 `lambda:GetDurableExecutionState`) 和基本 Lambda 執行所需的許可。

建立耐用的 函數

使用 `--durable-config` 參數建立您的耐用函數。

建立耐用的函數

1. 將具有相依性的函數程式碼封裝至 .zip 檔案：

```
zip -r function.zip index.mjs node_modules/
```

2. 建立已啟用持久性執行的函數：

```
aws lambda create-function \
  --function-name myDurableFunction \
  --runtime nodejs22.x \
  --role arn:aws:iam::123456789012:role/durable-function-role \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --durable-config '{"ExecutionTimeout": 3600, "RetentionPeriodInDays": 7}'
```

> **Note**
>
> 您只能在建立函數時啟用持久性執行。您無法在現有函數上啟用它。

> **Note**
>
> 目前 Java（預覽）中的耐用函數只能透過容器映像建立。如需從容器映像建立持久性函數的詳細資訊，請參閱持久性函數支援的執行時間。

發佈版本

雖然可以使用$LATEST版本限定詞叫用耐久的函數，但您必須一律使用指向穩定版本的合格 ARN，以確保程式碼的確定性執行。

```
aws lambda publish-version \
  --function-name myDurableFunction \
  --description "Initial version"
```

命令會傳回版本 ARN。請注意 ARN 結尾的版本編號（例如 :1)。

或者，建立指向 版本的別名：

```
aws lambda create-alias \
  --function-name myDurableFunction \
  --name prod \
  --function-version 1
```

叫用耐用的 函數

使用合格的 ARN （版本或別名） 叫用您的耐久函數。

> **Note**
>
> 等冪叫用：若要防止重試失敗叫用時重複執行，您可以提供執行名稱，確保at-most-once。如需詳細資訊，請參閱等冪性。

同步調用

對於在 15 分鐘內完成的執行，請使用同步調用：

```
aws lambda invoke \
  --function-name myDurableFunction:1 \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

或使用別名：

```
aws lambda invoke \
  --function-name myDurableFunction:prod \
  --payload '{"orderId": "order-12345"}' \
  --cli-binary-format raw-in-base64-out \
  response.json
```

非同步調用

對於長時間執行的執行，請使用非同步調用：

```
aws lambda invoke \
  --function-name myDurableFunction:prod \
  --invocation-type Event \
  --payload '{"orderId": "order-12345"}' \
```

```
  --cli-binary-format raw-in-base64-out \
  response.json
```

使用非同步調用時，Lambda 會立即傳回 。函數會繼續在背景執行。

> **Note**
>
> 您可以在 主控台中使用 $LATEST進行原型設計和測試。對於生產工作負載，請使用發佈的版本或別名。

管理持久的執行

使用下列命令來管理和監控耐用的函數執行。

列出執行

列出耐用函數的所有執行：

```
aws lambda list-durable-executions-by-function \
  --function-name myDurableFunction
```

取得執行詳細資訊

取得特定執行的詳細資訊：

```
aws lambda get-durable-execution \
  --durable-execution-arn arn:aws:lambda:us-
east-1:123456789012:function:myDurableFunction:my-function-version/durable-execution/
my-execution-name/my-execution-id
```

取得執行歷史記錄

檢視執行的檢查點歷史記錄：

```
aws lambda get-durable-execution-history \
  --durable-execution-arn arn:aws:lambda:us-
east-1:123456789012:function:myDurableFunction:my-function-version/durable-execution/
my-execution-name/my-execution-id
```

停止執行

停止執行中的耐久執行：

```
aws lambda stop-durable-execution \
  --durable-execution-arn arn:aws:lambda:us-
east-1:123456789012:function:myDurableFunction:my-function-version/durable-execution/
my-execution-name/my-execution-id
```

更新函數程式碼

更新您的耐用函數程式碼並發佈新版本：

更新和發佈新版本

1. 更新函數程式碼：

   ```
   aws lambda update-function-code \
     --function-name myDurableFunction \
     --zip-file fileb://function.zip
   ```

2. 等待更新完成：

   ```
   aws lambda wait function-updated \
     --function-name myDurableFunction
   ```

3. 發佈新版本：

   ```
   aws lambda publish-version \
     --function-name myDurableFunction \
     --description "Updated order processing logic"
   ```

4. 更新別名以指向新版本：

   ```
   aws lambda update-alias \
     --function-name myDurableFunction \
     --name prod \
     --function-version 2
   ```

> **Important**
>
> 執行中的執行會繼續使用其啟動的版本。新的調用使用更新的別名版本。

檢視函數日誌

在 CloudWatch Logs 中檢視耐用函數的日誌：

```
aws logs tail /aws/lambda/myDurableFunction --follow
```

篩選特定執行的日誌：

```
aws logs filter-log-events \
  --log-group-name /aws/lambda/myDurableFunction \
  --filter-pattern "exec-abc123"
```

清除資源

刪除您的耐用函數和相關聯的資源：

```
# Delete the function
aws lambda delete-function --function-name myDurableFunction

# Delete the IAM role policies
aws iam detach-role-policy \
  --role-name durable-function-role \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole

aws iam detach-role-policy \
  --role-name durable-function-role \
  --policy-arn arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicDurableExecutionRolePolicy

# Delete the role
aws iam delete-role --role-name durable-function-role
```

後續步驟

使用 部署您的耐用函數之後 AWS CLI：

- 使用 `list-durable-executions`和 `get-durable-execution`命令監控執行
- 檢視 AWS CloudTrail 資料事件中的檢查點操作
- 針對執行失敗或長時間執行的執行設定 CloudWatch 警示

- 使用 shell 指令碼或 CI/CD 管道自動化部署

如需 Lambda AWS CLI 命令的詳細資訊，請參閱 AWS CLI 命令參考。

使用 Infrastructure as Code 部署 Lambda 耐用函數

您可以使用 Infrastructure as Code (IaC) 工具部署 Lambda 耐用函數 AWS CloudFormation，例如 AWS CDK AWS Serverless Application Model， 或 Terraform。這些工具可讓您在程式碼中定義函數、執行角色和許可，讓部署可重複且受版本控制。

這三個工具都需要您：

- 在函數上啟用持久性執行
- 將檢查點許可授予執行角色
- 發佈版本或建立別名 （ 持久性函數需要合格的 ARNs)

ZIP 中的耐用函數

AWS CloudFormation

使用 在範本中 CloudFormation 定義您的耐用函數。下列範例會建立具有所需許可的耐用函數。

```
AWSTemplateFormatVersion: '2010-09-09'
Description: Lambda durable function example

Resources:
  DurableFunctionRole:
    Type: AWS::IAM::Role
    Properties:
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service: lambda.amazonaws.com
            Action: sts:AssumeRole
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicDurableExecutionRolePolicy

  DurableFunction:
```

```
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: myDurableFunction
      Runtime: nodejs22.x
      Handler: index.handler
      Role: !GetAtt DurableFunctionRole.Arn
      Code:
        ZipFile: |
          // Your durable function code here
          export const handler = async (event, context) => {
            return { statusCode: 200 };
          };
      DurableConfig:
        ExecutionTimeout: 3600
        RetentionPeriodInDays: 7

  DurableFunctionVersion:
    Type: AWS::Lambda::Version
    Properties:
      FunctionName: !Ref DurableFunction
      Description: Initial version

  DurableFunctionAlias:
    Type: AWS::Lambda::Alias
    Properties:
      FunctionName: !Ref DurableFunction
      FunctionVersion: !GetAtt DurableFunctionVersion.Version
      Name: prod

Outputs:
  FunctionArn:
    Description: Durable function ARN
    Value: !GetAtt DurableFunction.Arn
  AliasArn:
    Description: Function alias ARN (use this for invocations)
    Value: !Ref DurableFunctionAlias
```

部署範本

```
aws cloudformation deploy \
  --template-file template.yaml \
  --stack-name my-durable-function-stack \
  --capabilities CAPABILITY_IAM
```

AWS CDK

AWS CDK 可讓您使用程式設計語言定義基礎設施。下列範例示範如何使用 TypeScript 和 Python 建立耐用的函數。

TypeScript

```
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as iam from 'aws-cdk-lib/aws-iam';
import { Construct } from 'constructs';

export class DurableFunctionStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create the durable function
    const durableFunction = new lambda.Function(this, 'DurableFunction', {
      runtime: lambda.Runtime.NODEJS_22_X,
      handler: 'index.handler',
      code: lambda.Code.fromAsset('lambda'),
      functionName: 'myDurableFunction',
      durableConfig: { executionTimeout: Duration.hours(1), retentionPeriod:
 Duration.days(30) },
    });

    // Create version and alias
    const version = durableFunction.currentVersion;
    const alias = new lambda.Alias(this, 'ProdAlias', {
      aliasName: 'prod',
      version: version,
    });

    // Output the alias ARN
    new cdk.CfnOutput(this, 'FunctionAliasArn', {
      value: alias.functionArn,
      description: 'Use this ARN to invoke the durable function',
    });
  }
}
```

Python

```
from aws_cdk import (
    Stack,
    aws_lambda as lambda_,
    aws_iam as iam,
    CfnOutput,
)
from constructs import Construct

class DurableFunctionStack(Stack):
    def __init__(self, scope: Construct, id: str, **kwargs):
        super().__init__(scope, id, **kwargs)

        # Create the durable function
        durable_function = lambda_.Function(
            self, 'DurableFunction',
            runtime=lambda_.Runtime.NODEJS_22_X,
            handler='index.handler',
            code=lambda_.Code.from_asset('lambda'),
            function_name='myDurableFunction',
            durable_execution={execution_timeout: Duration.hours(1),
 retention_period: Duration.days(30)}
        )

        # Add durable execution managed policy for checkpoint permissions
        durable_function.role.add_managed_policy(
            iam.ManagedPolicy.from_aws_managed_policy_name('service-role/
AWSLambdaBasicDurableExecutionRolePolicy')
        )

        # Create version and alias
        version = durable_function.current_version
        alias = lambda_.Alias(
            self, 'ProdAlias',
            alias_name='prod',
            version=version
        )

        # Output the alias ARN
        CfnOutput(
            self, 'FunctionAliasArn',
            value=alias.function_arn,
            description='Use this ARN to invoke the durable function'
```

```
        )
```

部署 CDK 堆疊

```
cdk deploy
```

AWS Serverless Application Model

AWS SAM 簡化無伺服器應用程式的 CloudFormation 範本。下列範本使用 建立耐久的 函數 AWS SAM。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Lambda durable function with SAM

Resources:
  DurableFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: myDurableFunction
      Runtime: nodejs22.x
      Handler: index.handler
      CodeUri: ./src
      DurableConfig:
        ExecutionTimeout: 3600
        RetentionPeriodInDays: 7
      Policies:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicDurableExecutionRolePolicy
      AutoPublishAlias: prod

Outputs:
  FunctionArn:
    Description: Durable function ARN
    Value: !GetAtt DurableFunction.Arn
  AliasArn:
    Description: Function alias ARN (use this for invocations)
    Value: !Ref DurableFunction.Alias
```

部署 SAM 範本

```
sam build
```

```
sam deploy --guided
```

Terraform

Terraform 是支援 AWS 資源的熱門開放原始碼 IaC 工具。下列範例使用 AWS 提供者版本 6.25.0 或更新版本，使用 Terraform 建立耐久的函數。

```
terraform {
  required_version = ">= 1.0"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = ">= 6.25.0"
    }
  }
}

provider "aws" {
  region = "us-east-2"
}

# IAM Role for Lambda Function
resource "aws_iam_role" "lambda_role" {
  name = "durable-function-role"

  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action = "sts:AssumeRole"
      Effect = "Allow"
      Principal = {
        Service = "lambda.amazonaws.com"
      }
    }]
  })
}

# Attach durable execution policy for checkpoint operations
resource "aws_iam_role_policy_attachment" "lambda_durable" {
  policy_arn = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicDurableExecutionRolePolicy"
  role       = aws_iam_role.lambda_role.name
}
```

```
# Lambda Function with Durable Execution enabled
resource "aws_lambda_function" "durable_function" {
  filename      = "function.zip"
  function_name = "myDurableFunction"
  role          = aws_iam_role.lambda_role.arn
  handler       = "index.handler"
  runtime       = "nodejs22.x"
  timeout       = 30
  memory_size   = 512

  durable_config {
    execution_timeout = 900
    retention_period  = 7
  }
}

# Publish a version
resource "aws_lambda_alias" "prod" {
  name             = "prod"
  function_name    = aws_lambda_function.durable_function.function_name
  function_version = aws_lambda_function.durable_function.version
}

output "function_arn" {
  description = "ARN of the Lambda function"
  value       = aws_lambda_function.durable_function.arn
}

output "alias_arn" {
  description = "ARN of the function alias (use this for invocations)"
  value       = aws_lambda_alias.prod.arn
}
```

使用 Terraform 部署

```
terraform init
terraform plan
terraform apply
```

> **Note**
>
> Lambda 耐久函數的 Terraform 支援需要 AWS 提供者版本 6.25.0 或更新版本。如果您使用的是舊版，請更新您的提供者版本。

來自 OCI 容器映像的耐用函數

您也可以根據容器映像建立耐久函數。如需如何建置容器映像的指示，請參閱耐用函數支援的執行時間。

AWS CDK

AWS CDK 可讓您使用程式設計語言定義基礎設施。下列範例示範如何從容器映像使用 TypeScript 建立耐用的函數。

```
import * as cdk from 'aws-cdk-lib';
import * as lambda from 'aws-cdk-lib/aws-lambda';
import * as iam from 'aws-cdk-lib/aws-iam';
import { Construct } from 'constructs';

export class DurableFunctionStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Create the durable function
    const durableFunction = new lambda.DockerImageFunction(this, 'DurableFunction', {
      code: lambda.DockerImageCode.fromImageAsset('./lambda', {
        platform: cdk.aws_ecr_assets.Platform.LINUX_AMD64,
      }),
      functionName: 'myDurableFunction',
      memorySize: 512,
      timeout: cdk.Duration.seconds(30),
      durableConfig: { executionTimeout: cdk.Duration.hours(1), retentionPeriod:
 cdk.Duration.days(30) },
    });

    // Create version and alias
    const version = durableFunction.currentVersion;
    const alias = new lambda.Alias(this, 'ProdAlias', {
      aliasName: 'prod',
      version: version,
```

```
    });

    // Output the alias ARN
    new cdk.CfnOutput(this, 'FunctionAliasArn', {
      value: alias.functionArn,
      description: 'Use this ARN to invoke the durable function',
    });
  }
}
```

部署 CDK 堆疊

```
cdk deploy
```

AWS Serverless Application Model

AWS SAM 簡化無伺服器應用程式的 CloudFormation 範本。下列範本使用 建立耐久的 函數 AWS SAM。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: Lambda durable function with SAM

Resources:
  DurableFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: myDurableFunction
      PackageType: Image
      ImageUri: ./src
      DurableConfig:
        ExecutionTimeout: 3600
        RetentionPeriodInDays: 7
      Policies:
        - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicDurableExecutionRolePolicy
      AutoPublishAlias: prod
    Metadata:
      DockerTag: latest
      DockerContext: ./src
      Dockerfile: Dockerfile

Outputs:
  FunctionArn:
```

```
    Description: Durable function ARN
    Value: !GetAtt DurableFunction.Arn
  AliasArn:
    Description: Function alias ARN (use this for invocations)
    Value: !Ref DurableFunction.Alias
```

部署 SAM 範本

```
sam build
sam deploy --guided
```

常見的組態模式

無論您使用哪種 IaC 工具，請遵循這些模式以取得耐用的函數：

啟用持久性執行

在函數上設定 `DurableConfig` 屬性以啟用持久性執行。此屬性僅在建立函數時可用。您無法在現有函數上啟用持久性執行。

授予檢查點許可

將 `AWSLambdaBasicDurableExecutionRolePolicy`受管政策連接至執行角色。此政策包含必要的 `lambda:CheckpointDurableExecutions`和 `lambda:GetDurableExecutionState`許可。

使用合格的 ARNs

為您的函數建立版本或別名。耐用的函數需要合格的 ARNs（具有版本或別名）才能叫用。在 `AutoPublishAlias`中使用 AWS SAM 或在 CloudFormation AWS CDK和 Terraform 中建立明確版本。

套件相依性

在您的部署套件中包含耐用的執行 SDK。對於 Node.js，安裝 `@aws/durable-execution-sdk-js`。針對 Python，安裝 `aws-durable-execution-sdk-python`。

後續步驟

部署您的耐用函數之後：

- 使用合格的 ARN （版本或別名） 測試您的函數

- 在持久性執行索引標籤下的 Lambda 主控台中監控執行進度
- 檢視 AWS CloudTrail 資料事件中的檢查點操作
- 檢閱 CloudWatch Logs 以取得函數輸出和重播行為

如需使用 IaC 工具部署 Lambda 函數的詳細資訊，請參閱：

- CloudFormation AWS::Lambda::Function 參考
- AWS CDK Lambda 模組文件
- AWS SAM 開發人員指南

耐用函數或 Step Functions

Lambda 耐用函數和 透過自動狀態管理和故障復原 AWS Step Functions 啟用可靠的工作流程協同運作。它們提供不同的開發人員偏好設定和架構模式。耐用函數已針對 Lambda 內的應用程式開發進行最佳化，而 Step Functions 則是針對跨 AWS 服務的工作流程協同運作而建置。

何時使用耐用函數

在以下情況下使用耐用的函數：

- 您的團隊偏好標準程式設計語言和熟悉的開發工具
- 您的應用程式邏輯主要位於 Lambda 函數內
- 您想要精細控制程式碼中的執行狀態
- 您正在建置以 Lambda 為中心的應用程式，在工作流程和商業邏輯之間緊密結合
- 您想要快速反覆運算，而無需在程式碼和視覺化/JSON 設計工具之間切換

何時使用 Step Functions

在下列情況下使用 Step Functions：

- 您需要視覺化工作流程呈現才能跨團隊可見性
- 您正在協調多個 AWS 服務，並希望在沒有自訂 SDK 程式碼的情況下進行原生整合
- 您需要零維護基礎設施 （無需修補、執行時間更新）
- 非技術利益相關者需要了解和驗證工作流程邏輯

決策架構

使用下列問題來判斷哪些服務符合您的使用案例：

- 您的主要重點是什麼？ Lambda 中的應用程式開發 → 耐用函數。跨 AWS → Step Functions 的工作流程協同運作。
- 您偏好的程式設計模型是什麼？ 標準程式設計語言 → 耐用的函數。以圖形為基礎的 DSL 或視覺化設計工具 → Step Functions。
- 涉及多少 AWS 服務？ 主要是 Lambda → 耐用函數。多項 AWS 服務 → Step Functions。
- 您使用哪些開發工具？ Lambda 開發人員體驗、IDE 搭配 LLM 代理程式、程式設計語言特定單元測試架構 AWS SAM、 AWS CDK AWS Toolkit → 耐用函數。視覺化工作流程建置器， AWS CDK 以建立工作流程模型 → Step Functions。
- 誰管理基礎設施？ 需要 Lambda 內的彈性 → 耐用函數。想要完全受管、零維護 → Step Functions。

功能比較

下表比較 Step Functions 和 Lambda 耐用函數之間的主要功能：

功能	AWS Step Functions	Lambda 耐用函數
主要焦點	跨 的工作流程協同運作 AWS	Lambda 中的應用程式開發
服務類型	獨立的專用工作流程服務	在 Lambda 中執行
程式設計模型	以圖形為基礎的 Amazon States 語言 DSL 或 AWS CDK	標準程式設計語言 (JavaScript/TypeScript、Python)
開發工具	主控台 / AWS Toolkit IDE 延伸中的視覺化建置器， AWS CDK	IDE 和 LLM 代理程式內的 Lambda DX、單元測試架構 AWS SAM、AWS Toolkit IDE 延伸模組
整合	220+ AWS 服務、16k APIs	Lambda 事件驅動的程式設計模型延伸 （事件來源）
管理	全受管、執行期無關、零維護 （無修補、執行期更新）	在 Lambda 環境中受管

功能	AWS Step Functions	Lambda 耐用函數
最適合	業務流程和 IT 自動化、資料處理、AI 工作流程	分散式交易、具狀態應用程式邏輯、函數協同運作、資料處理、AI 工作流程

混合架構

許多應用程式都受益於使用這兩種服務。常見的模式是對 Lambda 內的應用程式層級邏輯使用耐久的函數，而 Step Functions 則協調 Lambda 函數以外跨多個 AWS 服務的高階工作流程。

遷移考量事項

啟動簡單、不斷發展的複雜：從 Lambda 工作流程的耐久函數開始。當您需要多服務協同運作或視覺化工作流程設計時，請新增 Step Functions。

現有的 Step Functions 使用者：為已建立的跨服務工作流程保留 Step Functions。對於需要可靠性的新 Lambda 應用程式邏輯，請考慮使用耐用的函數。

相關資源

- Lambda 耐用函數
- 使用 Step Functions 協調 Lambda 函數
- 耐用函數入門

範例和使用案例

Lambda 耐用函數可讓您使用步驟和等待等耐用操作來建置容錯的多步驟應用程式。透過自動檢查點和檢查點重播模型。 執行會在失敗後從頭開始重新啟動，但會略過已完成的檢查點，您的函數可以從失敗中復原並繼續執行，而不會遺失進度。

短期容錯程序

使用耐用的 函數來建置通常在幾分鐘內完成的可靠操作。雖然這些程序比長時間執行的工作流程短，但它們仍然受益於分散式系統的自動檢查點和容錯能力。耐用的 函數有助於確保您的多步驟程序即使在個別服務呼叫失敗時也能順利完成，而不需要複雜的錯誤處理或狀態管理程式碼。

常見案例包括飯店預訂系統、餐廳預訂平台、共乘行程請求、事件票證購買和 SaaS 訂閱升級。這些案例有共同的特性：多個必須一起完成的服務呼叫、在暫時性故障時自動重試的需求，以及跨分散式系統維持一致狀態的需求。

跨微型服務的分散式交易

使用自動轉返失敗，協調跨多項服務的付款、庫存和運送。每個服務操作都會包裝在步驟中，確保交易可以在服務失敗時從任何時間點復原。

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { orderId, amount, items } = event;

    // Reserve inventory across multiple warehouses
    const inventory = await context.step("reserve-inventory", async () => {
      return await inventoryService.reserve(items);
    });

    // Process payment
    const payment = await context.step("process-payment", async () => {
      return await paymentService.charge(amount);
    });

    // Create shipment
    const shipment = await context.step("create-shipment", async () => {
      return await shippingService.createShipment(orderId, inventory);
    });

    return { orderId, status: 'completed', shipment };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution
```

```
@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']
    amount = event['amount']
    items = event['items']

    # Reserve inventory across multiple warehouses
    inventory = context.step(
        lambda _: inventory_service.reserve(items),
        name='reserve-inventory'
    )

    # Process payment
    payment = context.step(
        lambda _: payment_service.charge(amount),
        name='process-payment'
    )

    # Create shipment
    shipment = context.step(
        lambda _: shipping_service.create_shipment(order_id, inventory),
        name='create-shipment'
    )

    return {'orderId': order_id, 'status': 'completed', 'shipment': shipment}
```

如果任何步驟失敗，函數會自動從最後一個成功的檢查點重試。即使付款處理暫時失敗，庫存保留仍會持續存在。當函數重試時，它會略過已完成的庫存步驟，並直接進行付款處理。這可消除重複的保留，並確保分散式系統的狀態一致。

使用多個步驟進行訂單處理

透過驗證、付款授權、庫存分配和自動重試和復原的履行來處理訂單。每個步驟都會經過檢查點，確保即使個別步驟失敗並重試，訂單仍會繼續。

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";
```

```
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { orderId, customerId, items } = event;

    // Validate order details
    const validation = await context.step("validate-order", async () => {
      const customer = await customerService.validate(customerId);
      const itemsValid = await inventoryService.validateItems(items);
      return { customer, itemsValid };
    });

    if (!validation.itemsValid) {
      return { orderId, status: 'rejected', reason: 'invalid_items' };
    }

    // Authorize payment
    const authorization = await context.step("authorize-payment", async () => {
      return await paymentService.authorize(
        validation.customer.paymentMethod,
        calculateTotal(items)
      );
    });

    // Allocate inventory
    const allocation = await context.step("allocate-inventory", async () => {
      return await inventoryService.allocate(items);
    });

    // Fulfill order
    const fulfillment = await context.step("fulfill-order", async () => {
      return await fulfillmentService.createShipment({
        orderId,
        items: allocation.allocatedItems,
        address: validation.customer.shippingAddress
      });
    });

    return {
      orderId,
      status: 'completed',
      trackingNumber: fulfillment.trackingNumber
    };
  }
```

```
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution

@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']
    customer_id = event['customerId']
    items = event['items']

    # Validate order details
    def validate_order(_):
        customer = customer_service.validate(customer_id)
        items_valid = inventory_service.validate_items(items)
        return {'customer': customer, 'itemsValid': items_valid}

    validation = context.step(validate_order, name='validate-order')

    if not validation['itemsValid']:
        return {'orderId': order_id, 'status': 'rejected', 'reason':
 'invalid_items'}

    # Authorize payment
    authorization = context.step(
        lambda _: payment_service.authorize(
            validation['customer']['paymentMethod'],
            calculate_total(items)
        ),
        name='authorize-payment'
    )

    # Allocate inventory
    allocation = context.step(
        lambda _: inventory_service.allocate(items),
        name='allocate-inventory'
    )

    # Fulfill order
    fulfillment = context.step(
```

```
        lambda _: fulfillment_service.create_shipment({
            'orderId': order_id,
            'items': allocation['allocatedItems'],
            'address': validation['customer']['shippingAddress']
        }),
        name='fulfill-order'
    )

    return {
        'orderId': order_id,
        'status': 'completed',
        'trackingNumber': fulfillment['trackingNumber']
    }
```

此模式可確保訂單永遠不會卡在中繼狀態。如果驗證失敗，則會在付款授權之前拒絕訂單。如果付款授權失敗，則不會配置庫存。每個步驟都以上一個步驟為基礎，並具有自動重試和復原。

注意

條件式檢查`if (!validation.itemsValid)`位於步驟之外，並在重播期間重新執行。這是安全的，因為它是確定性的，它永遠會在相同的驗證物件下產生相同的結果。

長時間執行的程序

對跨越數小時、數天或數週的程序使用耐久的函數。等待操作會暫停執行，而不會產生運算費用，讓長時間執行的程序符合成本效益。在等待期間，您的函數會停止執行，Lambda 會回收執行環境。恢復時，Lambda 會再次叫用您的函數，並從最後一個檢查點重播。

此執行模型讓耐用的函數非常適合需要長時間暫停的程序，無論是等待人工決策、外部系統回應、排程處理時段還是以時間為基礎的延遲。您只需支付作用中的運算時間，而非等待。

常見案例包括文件核准程序、排程批次處理、多日加入程序、訂閱試用程序，以及延遲通知系統。這些案例有共同的特性：以小時或天為單位測量的延長等待期、在這些等待期間維持執行狀態的需求，以及支付閒置運算時間的成本敏感要求是禁止的。

Human-in-the-loop核准

暫停執行文件檢閱、核准或決策，同時維持執行狀態。函數會等待外部回呼而不耗用資源，在收到核准時自動繼續。

此模式對於需要人工判斷或外部驗證的程序至關重要。函數會在回呼點暫停，等待時不會產生運算費用。當有人透過 API 提交其決策時，Lambda 會再次調用您的函數，並從檢查點重播，繼續執行核准結果。

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { documentId, reviewers } = event;

    // Step 1: Prepare document for review
    const prepared = await context.step("prepare-document", async () => {
      return await documentService.prepare(documentId);
    });

    // Step 2: Request approval with callback
    const approval = await context.waitForCallback(
      "approval-callback",
      async (callbackId) => {
        await notificationService.sendApprovalRequest({
          documentId,
          reviewers,
          callbackId,
          expiresIn: 86400
        });
      },
      {
        timeout: { seconds: 86400 }
      }
    );

    // Function resumes here when approval is received
    if (approval?.approved) {
      const finalized = await context.step("finalize-document", async () => {
        return await documentService.finalize(documentId, approval.comments);
      });

      return {
        status: 'approved',
```

```
        documentId,
        finalizedAt: finalized.timestamp
      };
    }

    // Handle rejection
    await context.step("archive-rejected", async () => {
      await documentService.archive(documentId, approval?.reason);
    });

    return {
      status: 'rejected',
      documentId,
      reason: approval?.reason
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution,
 WaitConfig

@durable_execution
def lambda_handler(event, context: DurableContext):
    document_id = event['documentId']
    reviewers = event['reviewers']

    # Step 1: Prepare document for review
    prepared = context.step(
        lambda _: document_service.prepare(document_id),
        name='prepare-document'
    )

    # Step 2: Request approval with callback
    def send_approval_request(callback_id):
        notification_service.send_approval_request({
            'documentId': document_id,
            'reviewers': reviewers,
            'callbackId': callback_id,
            'expiresIn': 86400
```

```
        })

    approval = context.wait_for_callback(
        send_approval_request,
        name='approval-callback',
        config=WaitConfig(timeout=86400)
    )

    # Function resumes here when approval is received
    if approval and approval.get('approved'):
        finalized = context.step(
            lambda _: document_service.finalize(document_id,
approval.get('comments')),
            name='finalize-document'
        )

        return {
            'status': 'approved',
            'documentId': document_id,
            'finalizedAt': finalized['timestamp']
        }

    # Handle rejection
    context.step(
        lambda _: document_service.archive(document_id, approval.get('reason') if
approval else None),
        name='archive-rejected'
    )

    return {
        'status': 'rejected',
        'documentId': document_id,
        'reason': approval.get('reason') if approval else None
    }
```

收到回呼且您的函數繼續時，它會從頭開始重播。prepare-document 步驟會立即傳回其檢查點結果。waitForCallback 操作也會立即傳回預存核准結果，而不是再次等待。接著執行會繼續完成或封存步驟。

多階段資料管道

透過擷取、轉換和載入階段處理大型資料集，並在階段之間使用檢查點。每個階段可能需要數小時才能完成，而檢查點可讓管道在中斷時從任何階段繼續。

此模式非常適合 ETL 工作流程、資料遷移或批次處理任務，而您需要在其中使用復原點分階段處理資料。如果階段失敗，管道會從上次完成的階段繼續，而不是從頭開始重新啟動。您也可以使用等待操作在階段之間暫停；遵守速率限制、等待下游系統就緒，或在離峰時間排程處理。

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { datasetId, batchSize } = event;

    // Stage 1: Extract data from source
    const extracted = await context.step("extract-data", async () => {
      const records = await sourceDatabase.extractRecords(datasetId);
      return { recordCount: records.length, records };
    });

    // Wait 5 minutes to respect source system rate limits
    await context.wait({ seconds: 300 });

    // Stage 2: Transform data in batches
    const transformed = await context.step("transform-data", async () => {
      const batches = chunkArray(extracted.records, batchSize);
      const results = [];

      for (const batch of batches) {
        const transformed = await transformService.processBatch(batch);
        results.push(transformed);
      }

      return { batchCount: batches.length, results };
    });

    // Wait until off-peak hours (e.g., 2 AM)
    const now = new Date();
```

```
    const targetHour = 2;
    const msUntilTarget = calculateMsUntilHour(now, targetHour);
    await context.wait({ seconds: Math.floor(msUntilTarget / 1000) });

    // Stage 3: Load data to destination
    const loaded = await context.step("load-data", async () => {
      let loadedCount = 0;

      for (const result of transformed.results) {
        await destinationDatabase.loadBatch(result);
        loadedCount += result.length;
      }

      return { loadedCount };
    });

    // Stage 4: Verify and finalize
    const verified = await context.step("verify-pipeline", async () => {
      const verification = await destinationDatabase.verifyRecords(datasetId);
      await pipelineService.markComplete(datasetId, verification);
      return verification;
    });

    return {
      datasetId,
      recordsProcessed: extracted.recordCount,
      batchesProcessed: transformed.batchCount,
      recordsLoaded: loaded.loadedCount,
      verified: verified.success
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution
from datetime import datetime

@durable_execution
def lambda_handler(event, context: DurableContext):
    dataset_id = event['datasetId']
```

```
    batch_size = event['batchSize']

    # Stage 1: Extract data from source
    def extract_data(_):
        records = source_database.extract_records(dataset_id)
        return {'recordCount': len(records), 'records': records}

    extracted = context.step(extract_data, name='extract-data')

    # Wait 5 minutes to respect source system rate limits
    context.wait(300)

    # Stage 2: Transform data in batches
    def transform_data(_):
        batches = chunk_array(extracted['records'], batch_size)
        results = []

        for batch in batches:
            transformed = transform_service.process_batch(batch)
            results.append(transformed)

        return {'batchCount': len(batches), 'results': results}

    transformed = context.step(transform_data, name='transform-data')

    # Wait until off-peak hours (e.g., 2 AM)
    now = datetime.now()
    target_hour = 2
    ms_until_target = calculate_ms_until_hour(now, target_hour)
    context.wait(ms_until_target // 1000)

    # Stage 3: Load data to destination
    def load_data(_):
        loaded_count = 0

        for result in transformed['results']:
            destination_database.load_batch(result)
            loaded_count += len(result)

        return {'loadedCount': loaded_count}

    loaded = context.step(load_data, name='load-data')

    # Stage 4: Verify and finalize
```

```
    def verify_pipeline(_):
        verification = destination_database.verify_records(dataset_id)
        pipeline_service.mark_complete(dataset_id, verification)
        return verification

    verified = context.step(verify_pipeline, name='verify-pipeline')

    return {
        'datasetId': dataset_id,
        'recordsProcessed': extracted['recordCount'],
        'batchesProcessed': transformed['batchCount'],
        'recordsLoaded': loaded['loadedCount'],
        'verified': verified['success']
    }
```

每個階段都會包裝在步驟中，建立檢查點，允許管道在中斷時從任何階段繼續。擷取和轉換之間的 5 分鐘等待會遵守來源系統速率限制，而不會耗用運算資源，而等到上午 2 點才會在離峰時間排程昂貴的負載操作。

注意

new Date() 呼叫和calculateMsUntilHour()函數位於步驟之外，並在重播期間重新執行。對於必須在重播之間保持一致的時間型操作，請計算步驟內的時間戳記，或僅將其用於等待持續時間 （檢查點）。

進階模式

使用持久性函數來建置複雜的多步驟應用程式，結合多個持久性操作、平行執行、陣列處理、條件式邏輯和輪詢。這些模式可讓您建置複雜的應用程式來協調許多任務，同時維持容錯能力和自動復原。

進階模式超過簡單的循序步驟。您可以與 同時執行操作parallel()、使用 處理陣列map()、使用 等待外部條件waitForCondition()，並結合這些基本概念來建置可靠的應用程式。每個耐用的操作都會建立自己的檢查點，因此您的應用程式可以在中斷時從任何時間點復原。

使用者加入程序

引導使用者完成註冊、電子郵件驗證、設定檔設定，以及具有重試處理的初始組態。此範例結合了循序步驟、回呼和條件式邏輯，以建立完整的加入程序。

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { userId, email } = event;

    // Step 1: Create user account
    const user = await context.step("create-account", async () => {
      return await userService.createAccount(userId, email);
    });

    // Step 2: Send verification email
    await context.step("send-verification", async () => {
      return await emailService.sendVerification(email);
    });

    // Step 3: Wait for email verification (up to 48 hours)
    const verified = await context.waitForCallback(
      "email-verification",
      async (callbackId) => {
        await notificationService.sendVerificationLink({
          email,
          callbackId,
          expiresIn: 172800
        });
      },
      {
        timeout: { seconds: 172800 }
      }
    );

    if (!verified) {
      await context.step("send-reminder", async () => {
        await emailService.sendReminder(email);
      });

      return {
        status: "verification_timeout",
        userId,
        message: "Email verification not completed within 48 hours"
```

```
      };
    }

    // Step 4: Initialize user profile in parallel
    const setupResults = await context.parallel("profile-setup", [
      async (ctx: DurableContext) => {
        return await ctx.step("create-preferences", async () => {
          return await preferencesService.createDefaults(userId);
        });
      },

      async (ctx: DurableContext) => {
        return await ctx.step("setup-notifications", async () => {
          return await notificationService.setupDefaults(userId);
        });
      },

      async (ctx: DurableContext) => {
        return await ctx.step("create-welcome-content", async () => {
          return await contentService.createWelcome(userId);
        });
      }
    ]);

    // Step 5: Send welcome email
    await context.step("send-welcome", async () => {
      const [preferences, notifications, content] = setupResults.getResults();
      return await emailService.sendWelcome({
        email,
        preferences,
        notifications,
        content
      });
    });

    return {
      status: "onboarding_complete",
      userId,
      completedAt: new Date().toISOString()
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution,
 WaitConfig
from datetime import datetime

@durable_execution
def lambda_handler(event, context: DurableContext):
    user_id = event['userId']
    email = event['email']

    # Step 1: Create user account
    user = context.step(
        lambda _: user_service.create_account(user_id, email),
        name='create-account'
    )

    # Step 2: Send verification email
    context.step(
        lambda _: email_service.send_verification(email),
        name='send-verification'
    )

    # Step 3: Wait for email verification (up to 48 hours)
    def send_verification_link(callback_id):
        notification_service.send_verification_link({
            'email': email,
            'callbackId': callback_id,
            'expiresIn': 172800
        })

    verified = context.wait_for_callback(
        send_verification_link,
        name='email-verification',
        config=WaitConfig(timeout=172800)
    )

    if not verified:
        context.step(
            lambda _: email_service.send_reminder(email),
            name='send-reminder'
        )
```

```
        return {
            'status': 'verification_timeout',
            'userId': user_id,
            'message': 'Email verification not completed within 48 hours'
        }

    # Step 4: Initialize user profile in parallel
    def create_preferences(ctx: DurableContext):
        return ctx.step(
            lambda _: preferences_service.create_defaults(user_id),
            name='create-preferences'
        )

    def setup_notifications(ctx: DurableContext):
        return ctx.step(
            lambda _: notification_service.setup_defaults(user_id),
            name='setup-notifications'
        )

    def create_welcome_content(ctx: DurableContext):
        return ctx.step(
            lambda _: content_service.create_welcome(user_id),
            name='create-welcome-content'
        )

    setup_results = context.parallel(
        [create_preferences, setup_notifications, create_welcome_content],
        name='profile-setup'
    )

    # Step 5: Send welcome email
    def send_welcome(_):
        results = setup_results.get_results()
        preferences, notifications, content = results[0], results[1], results[2]
        return email_service.send_welcome({
            'email': email,
            'preferences': preferences,
            'notifications': notifications,
            'content': content
        })

    context.step(send_welcome, name='send-welcome')

    return {
```

```
        'status': 'onboarding_complete',
        'userId': user_id,
        'completedAt': datetime.now().isoformat()
    }
```

此程序將循序步驟與檢查點結合，以建立帳戶和傳送電子郵件，然後暫停最多 48 小時等待電子郵件驗證，而不會耗用資源。條件式邏輯會根據驗證是否完成或逾時來處理不同的路徑。設定檔設定任務會使用平行操作同時執行，以減少總執行時間，而且每個步驟都會在暫時性失敗時自動重試，以協助確保上線可靠地完成。

跨 函數的鏈結調用

使用 從耐用函數內叫用其他 Lambda 函數`context.invoke()`。呼叫函數會在等待調用函數完成時暫停，建立保留結果的檢查點。如果呼叫函數在調用函數完成後中斷，它會以儲存的結果繼續呼叫函數。

當您有處理特定網域（客戶驗證、付款處理、庫存管理）且需要在工作流程中協調它們的特殊 函數時，請使用此模式。每個函數都會維護自己的邏輯，並可由多個協調器函數叫用，避免程式碼重複。

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";

// Main orchestrator function
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { orderId, customerId } = event;

    // Step 1: Validate customer by invoking customer service function
    const customer = await context.invoke(
      "validate-customer",
      "arn:aws:lambda:us-east-1:123456789012:function:customer-service:1",
      { customerId }
    );

    if (!customer.isValid) {
      return { orderId, status: "rejected", reason: "invalid_customer" };
    }
```

```
    // Step 2: Check inventory by invoking inventory service function
    const inventory = await context.invoke(
      "check-inventory",
      "arn:aws:lambda:us-east-1:123456789012:function:inventory-service:1",
      { orderId, items: event.items }
    );

    if (!inventory.available) {
      return { orderId, status: "rejected", reason: "insufficient_inventory" };
    }

    // Step 3: Process payment by invoking payment service function
    const payment = await context.invoke(
      "process-payment",
      "arn:aws:lambda:us-east-1:123456789012:function:payment-service:1",
      {
        customerId,
        amount: inventory.totalAmount,
        paymentMethod: customer.paymentMethod
      }
    );

    // Step 4: Create shipment by invoking fulfillment service function
    const shipment = await context.invoke(
      "create-shipment",
      "arn:aws:lambda:us-east-1:123456789012:function:fulfillment-service:1",
      {
        orderId,
        items: inventory.allocatedItems,
        address: customer.shippingAddress
      }
    );

    return {
      orderId,
      status: "completed",
      trackingNumber: shipment.trackingNumber,
      estimatedDelivery: shipment.estimatedDelivery
    };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution

# Main orchestrator function
@durable_execution
def lambda_handler(event, context: DurableContext):
    order_id = event['orderId']
    customer_id = event['customerId']

    # Step 1: Validate customer by invoking customer service function
    customer = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:customer-service:1',
        {'customerId': customer_id},
        name='validate-customer'
    )

    if not customer['isValid']:
        return {'orderId': order_id, 'status': 'rejected', 'reason':
 'invalid_customer'}

    # Step 2: Check inventory by invoking inventory service function
    inventory = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:inventory-service:1',
        {'orderId': order_id, 'items': event['items']},
        name='check-inventory'
    )

    if not inventory['available']:
        return {'orderId': order_id, 'status': 'rejected', 'reason':
 'insufficient_inventory'}

    # Step 3: Process payment by invoking payment service function
    payment = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:payment-service:1',
        {
            'customerId': customer_id,
            'amount': inventory['totalAmount'],
            'paymentMethod': customer['paymentMethod']
        },
        name='process-payment'
    )
```

```
    # Step 4: Create shipment by invoking fulfillment service function
    shipment = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:fulfillment-service:1',
        {
            'orderId': order_id,
            'items': inventory['allocatedItems'],
            'address': customer['shippingAddress']
        },
        name='create-shipment'
    )

    return {
        'orderId': order_id,
        'status': 'completed',
        'trackingNumber': shipment['trackingNumber'],
        'estimatedDelivery': shipment['estimatedDelivery']
    }
```

每次調用都會在協調器函數中建立檢查點。如果協調器在客戶驗證完成後中斷，它會使用儲存的客戶資料從該檢查點繼續，略過驗證調用。這可防止對下游服務的重複呼叫，並確保中斷之間的執行一致。

調用函數可以是耐用的函數或標準 Lambda 函數。如果您叫用耐用的 函數，它可以擁有自己的多步驟工作流程，其中包含等待和檢查點。協調器只會等待完整耐久執行完成，並收到最終結果。

> **Note**
>
> 不支援跨帳戶調用。所有調用函數必須與呼叫函數位於相同的 AWS 帳戶中。

使用檢查點進行批次處理

處理數百萬筆記錄，並在失敗後從最後一個成功的檢查點自動復原。此範例示範 函數如何將`map()`操作與區塊化和速率限制結合，以處理大規模的資料處理。

TypeScript

```
import { DurableContext, withDurableExecution } from "@aws/durable-execution-sdk-
js";
```

```
interface Batch {
  batchIndex: number;
  recordIds: string[];
}

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    const { datasetId, batchSize = 1000 } = event;

    // Step 1: Get all record IDs to process
    const recordIds = await context.step("fetch-record-ids", async () => {
      return await dataService.getRecordIds(datasetId);
    });

    // Step 2: Split into batches
    const batches: Batch[] = [];
    for (let i = 0; i < recordIds.length; i += batchSize) {
      batches.push({
        batchIndex: Math.floor(i / batchSize),
        recordIds: recordIds.slice(i, i + batchSize)
      });
    }

    // Step 3: Process batches with controlled concurrency
    const batchResults = await context.map(
      "process-batches",
      batches,
      async (ctx: DurableContext, batch: Batch, index: number) => {
        const processed = await ctx.step(`batch-${batch.batchIndex}`, async () => {
          const results = [];
          for (const recordId of batch.recordIds) {
            const result = await recordService.process(recordId);
            results.push(result);
          }
          return results;
        });

        const validated = await ctx.step(`validate-${batch.batchIndex}`, async () =>
 {
          return await validationService.validateBatch(processed);
        });

        return {
          batchIndex: batch.batchIndex,
```

```
          recordCount: batch.recordIds.length,
          successCount: validated.successCount,
          failureCount: validated.failureCount
        };
      },
      {
        maxConcurrency: 5
      }
    );

    // Step 4: Aggregate results
    const summary = await context.step("aggregate-results", async () => {
      const results = batchResults.getResults();
      const totalSuccess = results.reduce((sum, r) => sum + r.successCount, 0);
      const totalFailure = results.reduce((sum, r) => sum + r.failureCount, 0);

      return {
        datasetId,
        totalRecords: recordIds.length,
        batchesProcessed: batches.length,
        successCount: totalSuccess,
        failureCount: totalFailure,
        completedAt: new Date().toISOString()
      };
    });

    return summary;
  }
);
```

Python

```
from aws_durable_execution_sdk_python import DurableContext, durable_execution,
 MapConfig
from datetime import datetime
from typing import List, Dict

@durable_execution
def lambda_handler(event, context: DurableContext):
    dataset_id = event['datasetId']
    batch_size = event.get('batchSize', 1000)
```

```
    # Step 1: Get all record IDs to process
    record_ids = context.step(
        lambda _: data_service.get_record_ids(dataset_id),
        name='fetch-record-ids'
    )

    # Step 2: Split into batches
    batches = []
    for i in range(0, len(record_ids), batch_size):
        batches.append({
            'batchIndex': i // batch_size,
            'recordIds': record_ids[i:i + batch_size]
        })

    # Step 3: Process batches with controlled concurrency
    def process_batch(ctx: DurableContext, batch: Dict, index: int):
        batch_index = batch['batchIndex']

        def process_records(_):
            results = []
            for record_id in batch['recordIds']:
                result = record_service.process(record_id)
                results.append(result)
            return results

        processed = ctx.step(process_records, name=f'batch-{batch_index}')

        validated = ctx.step(
            lambda _: validation_service.validate_batch(processed),
            name=f'validate-{batch_index}'
        )

        return {
            'batchIndex': batch_index,
            'recordCount': len(batch['recordIds']),
            'successCount': validated['successCount'],
            'failureCount': validated['failureCount']
        }

    batch_results = context.map(
        process_batch,
        batches,
        name='process-batches',
```

```
        config=MapConfig(max_concurrency=5)
    )

    # Step 4: Aggregate results
    def aggregate_results(_):
        results = batch_results.get_results()
        total_success = sum(r['successCount'] for r in results)
        total_failure = sum(r['failureCount'] for r in results)

        return {
            'datasetId': dataset_id,
            'totalRecords': len(record_ids),
            'batchesProcessed': len(batches),
            'successCount': total_success,
            'failureCount': total_failure,
            'completedAt': datetime.now().isoformat()
        }

    summary = context.step(aggregate_results, name='aggregate-results')

    return summary
```

記錄會分割為可管理的批次，以避免記憶體或下游服務過載，然後多個批次程序同時maxConcurrency控制平行處理。每個批次都有自己的檢查點，因此失敗只會重試失敗的批次，而不是重新處理所有記錄。此模式非常適合處理可能需要數小時的 ETL 任務、資料遷移或大量操作。

後續步驟

- 探索基本概念以了解 DurableContext、步驟和等待
- 檢閱撰寫確定性程式碼和最佳化效能的最佳實務
- 了解如何在本機和雲端測試耐用的函數
- 比較耐用的函數與 Step Functions，以了解每個方法最有效的時機。請參閱耐用函數或 Step Functions。

Lambda 耐用函數的安全性和許可

Lambda 耐用函數需要特定的 IAM 許可才能管理檢查點操作。僅授予函數所需的許可，以遵循最低權限原則。

執行角色許可

您耐用函數的執行角色需要建立檢查點和擷取執行狀態的許可。下列政策顯示最低必要許可：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:CheckpointDurableExecution",
                "lambda:GetDurableExecutionState"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:function-name:*"
        }
    ]
}
```

當您使用主控台建立耐用的函數時，Lambda 會自動將這些許可新增至執行角色。如果您使用 AWS CLI 或 建立函數 AWS CloudFormation，請將這些許可新增至執行角色。

最低權限原則

將 Resource元素範圍限定為特定函數 ARNs，而不是使用萬用字元。這只會將執行角色限制為需要它們的函數的檢查點操作。

範例：多個函數的範圍許可

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:CheckpointDurableExecution",
                "lambda:GetDurableExecutionState"
            ],
            "Resource": [
                "arn:aws:lambda:us-east-1:123456789012:function:orderProcessor:*",
                "arn:aws:lambda:us-east-1:123456789012:function:paymentHandler:*"
            ]
        }
    ]
```

```
}
```

或者，您可以使用 AWS 受管政策`AWSLambdaBasicDurableExecutionRolePolicy`，其中包含必要的持久執行許可，以及 Amazon CloudWatch Logs 的基本 Lambda 執行許可。

狀態加密

Lambda 耐用函數使用 AWS 擁有的金鑰自動啟用靜態加密，無需付費。每個函數執行都會維持其他執行無法存取的隔離狀態。不支援客戶受管金鑰 (CMK)。

檢查點資料包括：

- 步驟結果和傳回值
- 執行進度和時間軸
- 等待狀態資訊

當 Lambda 讀取或寫入檢查點資料時，所有資料都會使用 TLS 在傳輸中加密。

使用自訂序列化程式和還原序列化程式進行自訂加密

對於關鍵安全需求，您可以使用自訂序列化程式和使用耐用 SDK 的還原序列化程式 (SerDer) 來實作自己的加密和解密機制。此方法可讓您完全控制用來保護檢查點資料的加密金鑰和演算法。

> **Important**
>
> 當您使用自訂加密時，您會失去 Lambda 主控台和 API 回應中操作結果的可見性。檢查點資料會在執行歷史記錄中顯示加密，且無法在未解密的情況下進行檢查。

函數的執行角色對自訂 SerDer 實作中使用的 AWS KMS 金鑰的需求`kms:Encrypt`和`kms:Decrypt`許可。

CloudTrail 日誌記錄

Lambda 會將檢查點操作記錄為其中的資料事件 AWS CloudTrail。您可以使用 CloudTrail 來稽核檢查點的建立時間、追蹤執行狀態變更，以及監控對持久執行資料的存取。

檢查點操作會顯示在具有下列事件名稱的 CloudTrail 日誌中：

- `CheckpointDurableExecution` - 步驟完成並建立檢查點時記錄

- `GetDurableExecutionState` - 當 Lambda 在重播期間擷取執行狀態時記錄

若要啟用耐用函數的資料事件記錄，請設定 CloudTrail 追蹤來記錄 Lambda 資料事件。如需詳細資訊，請參閱《CloudTrail 使用者指南》中的記錄資料事件。

範例：檢查點操作的 CloudTrail 日誌項目

```
{
    "eventVersion": "1.08",
    "eventTime": "2024-11-16T10:30:45Z",
    "eventName": "CheckpointDurableExecution",
    "eventSource": "lambda.amazonaws.com",
    "requestParameters": {
        "functionName": "myDurableFunction",
        "executionId": "exec-abc123",
        "stepId": "step-1"
    },
    "responseElements": null,
    "eventType": "AwsApiCall"
}
```

跨帳戶考量事項

如果您跨 AWS 帳戶叫用耐久的函數，呼叫帳戶需要 `lambda:InvokeFunction` 許可，但檢查點操作一律使用函數帳戶中的執行角色。呼叫帳戶無法直接存取檢查點資料或執行狀態。

此隔離可確保檢查點資料在函數的帳戶內保持安全，即使從外部帳戶調用也一樣。

繼承的 Lambda 安全功能

耐用的函數繼承 Lambda 的所有安全、控管和合規功能，包括 VPC 連線、環境變數加密、無效字母佇列、預留並行、函數 URLs、程式碼簽署和合規認證 (SOC、PCI DSS、HIPAA 等）。

如需 Lambda 安全性功能的詳細資訊，請參閱《Lambda 開發人員指南》中的 中的安全性 AWS Lambda。持久功能的唯一額外安全考量是本指南中記錄的檢查點許可。

耐用的執行 SDK

耐用執行 SDK 是建置耐用函數的基礎。它提供檢查點進度、處理重試和管理執行流程所需的基本概念。SDK 可抽象化檢查點管理和重播的複雜性，讓您撰寫可自動容錯的循序程式碼。

開發套件適用於 JavaScript、TypeScript、Python 和 Java （預覽版）。如需完整的 API 文件和範例，請參閱 GitHub 上的 JavaScript/TypeScript SDK、Python SDK 和 Java SDK。

DurableContext

開發套件為您的函數提供公開所有耐久操作的DurableContext物件。此內容會取代標準 Lambda 內容，並提供建立檢查點、管理執行流程以及與外部系統協調的方法。

若要使用 SDK，請使用耐用的執行包裝函式來包裝 Lambda 處理常式：

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Your function receives DurableContext instead of Lambda context
    // Use context.step(), context.wait(), etc.
    return result;
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext

@durable_execution
def handler(event: dict, context: DurableContext):
    # Your function receives DurableContext
    # Use context.step(), context.wait(), etc.
    return result
```

Java (Preview)

```
import software.amazon.lambda.durable.DurableContext;
import software.amazon.lambda.durable.DurableHandler;
```

```
public class Handler extends DurableHandler<Object, String> {
    @Override
    public String handleRequest(Object input, DurableContext context) {
        // Your function receives DurableContext
        // Use context.step(), context.wait(), etc.
        return result;
    }
}
```

包裝函式會攔截函數叫用、載入任何現有的檢查點日誌，並提供DurableContext管理重播和檢查點的 。

開發套件的功能

開發套件會處理三個關鍵責任，以實現持久的執行：

檢查點管理：軟體開發套件會在函數執行耐久操作時自動建立檢查點。每個檢查點都會記錄操作類型、輸入和結果。函數完成步驟時，SDK 會保留檢查點再繼續。這可確保函數可以在中斷時從任何已完成的操作恢復。

重播協調：當函數在暫停或中斷後恢復時，軟體開發套件會執行重播。它會從頭開始執行程式碼，但會略過已完成的操作，使用儲存的檢查點結果，而不是重新執行它們。SDK 可確保重播是確定性的，因為提供相同的輸入和檢查點日誌，您的函數會產生相同的結果。

狀態隔離：SDK 會與您的業務邏輯分開維護執行狀態。每個持久性執行都有自己的檢查點日誌，其他執行無法存取。SDK 會加密靜態檢查點資料，並確保狀態在重播之間保持一致。

檢查點的運作方式

當您呼叫耐用操作時，軟體開發套件會遵循以下順序：

1. 檢查現有檢查點：軟體開發套件會檢查此操作是否已在先前的調用中完成。如果檢查點存在，軟體開發套件會傳回儲存的結果，而無需重新執行操作。
2. 執行操作：如果沒有檢查點，則 SDK 會執行您的操作程式碼。對於步驟，這表示呼叫您的 函數。對於等待，這表示排程恢復。
3. 建立檢查點：操作完成後，開發套件會序列化結果並建立檢查點。檢查點包含操作類型、名稱、輸入、結果和時間戳記。
4. 持久性檢查點： SDK 會呼叫 Lambda 檢查點 API 來持久性檢查點。這可確保檢查點在繼續執行之前是耐用的。

5. 傳回結果：軟體開發套件會將操作結果傳回至您的程式碼，繼續下一個操作。

此序列可確保一旦操作完成，就會安全地存放其結果。如果您的函數在任何時間點中斷，開發套件可以重新執行到最後完成的檢查點。

重播行為

當您的函數在暫停或中斷後繼續時，軟體開發套件會執行重播：

1. 載入檢查點日誌： SDK 會從 Lambda 擷取此執行的檢查點日誌。
2. 從頭開始執行：軟體開發套件會從頭調用您的處理常式函數，而不是從暫停的位置調用。
3. 略過已完成的持久性操作：當程式碼呼叫持久性操作時，軟體開發套件會根據檢查點日誌檢查每個操作。對於已完成的耐久操作，軟體開發套件會傳回預存結果，而不執行操作程式碼。

> **Note**
>
> 如果子內容的結果大於檢查點大小上限 (256 KB)，則在重播期間會再次執行內容的程式碼。這可讓您從在內容中執行的耐久操作建構大型結果，這些操作將從檢查點日誌中查詢。因此，只能在內容本身中執行確定性程式碼。使用具有大型結果的子內容時，最佳實務是在步驟內執行長時間執行或非確定性工作，並且只執行將結果合併到內容本身的短期執行任務。

4. 在中斷點繼續：當開發套件在沒有檢查點的情況下達到 操作時，它會正常執行，並在耐久操作完成時建立新的檢查點。

此重播機制需要您的程式碼具有決定性。假設輸入和檢查點日誌相同，您的函數必須進行相同序列的耐久操作呼叫。SDK 透過驗證操作名稱和類型在重播期間符合檢查點日誌來強制執行此操作。

可用的耐用操作

`DurableContext` 提供不同協調模式的操作。每個耐用的操作都會自動建立檢查點，確保您的函數可以隨時恢復。

步驟

使用自動檢查點和重試來執行商業邏輯。對呼叫外部服務、執行計算或執行任何應檢查點邏輯的操作使用步驟。軟體開發套件會在步驟前後建立檢查點，存放結果以進行重播。

TypeScript

```
const result = await context.step('process-payment', async () => {
  return await paymentService.charge(amount);
});
```

Python

```
result = context.step(
    lambda _: payment_service.charge(amount),
    name='process-payment'
)
```

Java (Preview)

```
var result = context.step("process-payment", Payment.class,
    () -> paymentService.charge(amount)
);
```

步驟支援可設定的重試策略、執行語意 (at-most-once或at-least-once和自訂序列化。

等待

在指定的持續時間內暫停執行，而不耗用運算資源。SDK 會建立檢查點、終止函數叫用，以及排程恢復。當等待完成時，Lambda 會再次叫用您的函數，而 SDK 會在繼續之前重播至等待點。

TypeScript

```
// Wait 1 hour without charges
await context.wait({ seconds: 3600 });
```

Python

```
# Wait 1 hour without charges
context.wait(3600)
```

Java (Preview)

```
// Wait 1 hour without charges
context.wait(Duration.ofHours(1));
```

回呼

回呼可讓函數暫停並等待外部系統提供輸入。當您建立回呼時，開發套件會產生唯一的回呼 ID 並建立檢查點。然後，您的函數會暫停（終止調用），而不會產生運算費用。外部系統使用 SendDurableExecutionCallbackSuccess或 SendDurableExecutionCallbackFailure Lambda APIs提交回呼結果。提交回呼時，Lambda 會再次叫用您的函數，開發套件會重播至回呼點，而函數會繼續回呼結果。

開發套件提供兩種使用回呼的方法：

createCallback：建立回呼，並同時傳回 promise 和回呼 ID。您可以將回呼 ID 傳送至外部系統，該系統會使用 Lambda API 提交結果。

TypeScript

```
const [promise, callbackId] = await context.createCallback('approval', {
  timeout: { hours: 24 }
});

await sendApprovalRequest(callbackId, requestData);
const approval = await promise;
```

Python

```
callback = context.create_callback(
    name='approval',
    config=CallbackConfig(timeout_seconds=86400)
)

context.step(
    lambda _: send_approval_request(callback.callback_id),
    name='send_request'
)

approval = callback.result()
```

Java (Preview)

```
var config = CallbackConfig.builder(Duration.ofHours(24)).timeout()

var callback = context.createCallback("approval", String.class, config);

context.step("send-request", String.class, () -> {
    notificationService.sendApprovalRequest(callback.callbackId(), requestData);
    return "request-sent";
});

// Blocks until the callback finishes or times out
String approval = callback.get();
```

waitForCallback：將回呼建立和提交結合在一個操作中，簡化回呼處理。開發套件會建立回呼、使用回呼 ID 執行您的提交者函數，並等待結果。

TypeScript

```
const result = await context.waitForCallback(
  'external-api',
  async (callbackId, ctx) => {
    await submitToExternalAPI(callbackId, requestData);
  },
  { timeout: { minutes: 30 } }
);
```

Python

```
result = context.wait_for_callback(
    lambda callback_id: submit_to_external_api(callback_id, request_data),
    name='external-api',
    config=WaitForCallbackConfig(timeout_seconds=1800)
)
```

Java (Preview)

waitForCallback 仍在開發 Java。

設定逾時以防止函數無限期等待。如果回呼逾時，開發套件會擲回 `CallbackError`，而您的函數可以處理逾時案例。針對長時間執行的回呼使用活動訊號逾時，以偵測外部系統何時停止回應。

將回呼用於human-in-the-loop工作流程、外部系統整合、Webhook 回應或任何執行必須針對外部輸入暫停的情況。

平行執行

與選用並行控制同時執行多個操作。SDK 會管理平行執行、為每個操作建立檢查點，以及根據您的完成政策處理失敗。

TypeScript

```
const results = await context.parallel([
  async (ctx) => ctx.step('task1', async () => processTask1()),
  async (ctx) => ctx.step('task2', async () => processTask2()),
  async (ctx) => ctx.step('task3', async () => processTask3())
]);
```

Python

```
results = context.parallel(
    lambda ctx: ctx.step(lambda _: process_task1(), name='task1'),
    lambda ctx: ctx.step(lambda _: process_task2(), name='task2'),
    lambda ctx: ctx.step(lambda _: process_task3(), name='task3')
)
```

Java (Preview)

平行處理仍在開發 Java。

使用 `parallel` 可同時執行獨立操作。

Map

使用選用並行控制，同時對陣列中的每個項目執行 操作。SDK 會管理並行執行、為每個操作建立檢查點，以及根據您的完成政策處理失敗。

TypeScript

```
const results = await context.map(itemArray, async (ctx, item, index) =>
  ctx.step('task', async () => processItem(item, index))
);
```

Python

```
results = context.map(
    item_array,
    lambda ctx, item, index: ctx.step(
        lambda _: process_item(item, index),
        name='task'
    )
)
```

Java (Preview)

地圖仍在開發 Java。

使用 map 處理具有並行控制的陣列。

子內容

建立用於分組操作的隔離執行內容。子內容有自己的檢查點日誌，可包含多個步驟、等待和其他操作。SDK 會將整個子內容視為單一單位，以供重試和復原。

使用子內容來組織複雜的工作流程、實作子工作流程，或隔離應同時重試的操作。

TypeScript

```
const result = await context.runInChildContext(
  'batch-processing',
  async (childCtx) => {
    return await processBatch(childCtx, items);
  }
);
```

Python

```
result = context.run_in_child_context(
    lambda child_ctx: process_batch(child_ctx, items),
    name='batch-processing'
)
```

Java (Preview)

```
var result = context.runInChildContext(
    "batch-processing",
    String.class,
    childCtx -> process_batch(childCtx, items)
);
```

重播機制要求以決定性順序進行持久性操作。使用多個子內容，您可以同時執行多個工作串流，而且決定性會分別套用在每個內容中。這可讓您建置高效能函數，以有效率地利用多個 CPU 核心。

例如，假設我們啟動兩個子內容 A 和 B。在初始調用時，內容中的步驟會依此順序執行，而 'A' 步驟會與 'B' 步驟同時執行：A1, B1, B2, A2, A3。重新執行時，從檢查點日誌擷取結果時，時間會快得多，而步驟的發生順序也不同：B1, A1, A2, B2, A3。由於「A」步驟的順序正確 (A1、A2, A3)，且「B」步驟的順序正確 (B1、B2)，因此正確滿足確定性的需求。

條件式等待

輪詢在嘗試之間具有自動檢查點的條件。SDK 會執行您的檢查函數、建立具有結果的檢查點、根據您的策略等待，並重複直到滿足條件為止。

TypeScript

```
const result = await context.waitForCondition(
  async (state, ctx) => {
    const status = await checkJobStatus(state.jobId);
    return { ...state, status };
  },
  {
    initialState: { jobId: 'job-123', status: 'pending' },
    waitStrategy: (state) =>
      state.status === 'completed'
        ? { shouldContinue: false }
        : { shouldContinue: true, delay: { seconds: 30 } }
  }
);
```

Python

```
result = context.wait_for_condition(
    lambda state, ctx: check_job_status(state['jobId']),
    config=WaitForConditionConfig(
        initial_state={'jobId': 'job-123', 'status': 'pending'},
        wait_strategy=lambda state, attempt:
            {'should_continue': False} if state['status'] == 'completed'
            else {'should_continue': True, 'delay': 30}
    )
)
```

Java (Preview)

waitForCondition 仍在開發 Java。

使用 `waitForCondition`輪詢外部系統、等待資源準備就緒，或使用退避實作重試。

函數調用

叫用另一個 Lambda 函數並等待其結果。SDK 會建立檢查點、叫用目標函數，並在叫用完成時繼續您的函數。這可啟用函數合成和工作流程分解。

TypeScript

```
const result = await context.invoke(
  'invoke-processor',
  'arn:aws:lambda:us-east-1:123456789012:function:processor:1',
  { data: inputData }
);
```

Python

```
result = context.invoke(
    'arn:aws:lambda:us-east-1:123456789012:function:processor:1',
    {'data': input_data},
    name='invoke-processor'
)
```

Java (Preview)

```
var result = context.invoke(
    "invoke-processor",
    "arn:aws:lambda:us-east-1:123456789012:function:processor:1",
    inputData,
    Result.class,
    InvokeConfig.builder().build()
);
```

如何計量耐久性操作

您透過 呼叫的每個耐用操作都會DurableContext建立檢查點，以追蹤執行進度並存放狀態資料。這些操作會根據其用量產生費用，而檢查點可能包含對您的資料寫入和保留成本有所貢獻的資料。儲存的資料包括調用事件資料、步驟傳回的承載，以及完成回呼時傳遞的資料。了解耐用操作的計量方式可協助您預估執行成本並最佳化工作流程。如需定價的詳細資訊，請參閱 Lambda 定價頁面。

承載大小是指持久性操作持續的序列化資料大小。資料是以位元組為單位測量，大小可能會因 操作使用的序列化程式而有所不同。操作的承載可能是成功完成的結果本身，如果操作失敗，可能是序列化錯誤物件。

基本操作

基本操作是耐用函數的基本建置區塊：

作業	檢查點計時	操作數量	資料持續存在
執行	已開始	1	輸入承載大小
執行	已完成 Succeeded/Failed/Stopped)	0	輸出承載大小
Step (步驟)	Retry/Succeeded/Failed	1 + N 次重試	每次嘗試傳回的承載大小
等候	已開始	1	N/A
WaitForCondition	每次輪詢嘗試	1 + N 輪詢	每次輪詢嘗試傳回的承載大小
調用層級重試	已開始	1	錯誤物件的承載

回呼操作

回呼操作可讓您的函數暫停並等待外部系統提供輸入。這些操作會在建立回呼和完成時建立檢查點：

作業	檢查點計時	操作數量	資料持續存在
CreateCallback	已開始	1	N/A

作業	檢查點計時	操作數量	資料持續存在
透過 API 呼叫完成回呼	已完成	0	回呼承載
WaitForCallback	已開始	3 + N 次重試（內容 + 回呼 + 步驟）	提交者步驟嘗試傳回的承載，加上兩個回呼承載副本

複合操作

複合操作結合多個耐久操作來處理複雜的協調模式，例如平行執行、陣列處理和巢狀內容：

作業	檢查點計時	操作數量	資料持續存在
平行	已開始	1 + N 個分支 (1 個父內容 + N 個子內容）	每個分支最多兩個傳回承載大小的副本，以及每個分支的狀態
Map	已開始	1 + N 個分支 (1 個父內容 + N 個子內容）	每次反覆運算傳回的承載大小最多兩個副本，加上每次反覆運算的狀態
Promise 協助程式	已完成	1	從 promise 傳回的承載大小
RunInChildContext	成功/失敗	1	從子內容傳回的承載大小

對於內容，例如來自 `runInChildContext`或由複合操作內部使用，小於 256 KB 的結果會直接檢查點。不會儲存較大的結果，而是透過重新處理內容的操作，在重播期間重建結果。

支援耐用函數的執行時間

耐用的函數可用於選取的受管執行期和 OCI 容器映像，以提高執行期版本的靈活性。您可以直接在主控台中使用受管執行時間，或透過infrastructure-as-code以程式設計方式建立 Node.js 和 Python 的耐用函數。Java （預覽版） 中的耐用函數目前只能透過容器映像部署。

Lambda 受管執行時間

當您在 Lambda 主控台中或使用 AWS CLI 搭配 `--durable-config '{"ExecutionTimeout": 3600, "RetentionPeriodInDays": 7}'` 參數建立函數時，下列受管執行期支援耐久函數。如需 Lambda 執行時間的完整資訊，請參閱 Lambda 執行時間。

Language	執行時期
Node.js	nodejs22.x
Node.js	nodejs24.x
Python	python3.13
Python	python3.14

> **Note**
>
> Lambda 執行時間包括用於測試和開發的耐久執行 SDK。不過，我們建議您在部署套件中包含 開發套件以供生產使用。這可確保版本一致性，並避免可能影響函數行為的潛在執行時間更新。

Node.js

在 Node.js 專案中安裝 SDK：

```
npm install @aws/durable-execution-sdk-js
```

SDK 支援 JavaScript 和 TypeScript。對於 TypeScript 專案，開發套件包含類型定義。

Python

在 Python 專案中安裝 SDK：

```
pip install aws-durable-execution-sdk-python
```

Python SDK 使用同步方法，不需要 `async/await`。

Java（預覽版）

將相依性新增至 `pom.xml`：

```
<dependency>
    <groupId>software.amazon.lambda.durable</groupId>
    <artifactId>aws-durable-execution-sdk-java</artifactId>
    <version>VERSION</version>
</dependency>
```

在 Java 專案中安裝 SDK：

```
mvn install
```

Java 開發套件的預覽版本可供使用。waitForCondition、 waitForCallback、平行和映射操作仍在開發中。

容器映像

您可以搭配容器映像使用耐用的函數，以支援其他執行時間版本或自訂執行時間組態。容器映像可讓您使用不可做為受管執行時間的執行時間版本，或自訂執行時間環境。

若要使用容器映像建立耐用的函數：

1. 根據 Lambda 基礎映像建立 Dockerfile
2. 在容器中安裝耐用的執行 SDK

3. 建置容器映像並將其推送至 Amazon Elastic Container Registry
4. 從已啟用持久性執行的容器映像建立 Lambda 函數

容器範例

建立 Dockerfile：

Python

建立適用於 Python 3.11 的 Dockerfile：

```
FROM public.ecr.aws/lambda/python:3.11

# Copy requirements file
COPY requirements.txt ${LAMBDA_TASK_ROOT}/

# Install dependencies including durable SDK
RUN pip install -r requirements.txt

# Copy function code
COPY lambda_function.py ${LAMBDA_TASK_ROOT}/

# Set the handler
CMD [ "lambda_function.handler" ]
```

建立`requirements.txt`檔案：

```
aws-durable-execution-sdk-python
```

Java (Preview)

為 Java 25 建立 Dockerfile：

```
FROM --platform=linux/amd64 public.ecr.aws/lambda/java:25

# Install Maven
```

```
RUN dnf install -y maven

WORKDIR /var/task

# Copy Maven configuration and source code
COPY pom.xml .
COPY src ./src

# Build
RUN mvn clean package -DskipTests

# Move JAR to lib directory
RUN mv target/*.jar lib/

# Set the handler
CMD ["src.path.to.lambdaFunction::handler"]
```

建置並推送映像：

```
# Build the image
docker build -t my-durable-function .

# Tag for ECR
docker tag my-durable-function:latest 123456789012.dkr.ecr.us-east-1.amazonaws.com/my-
durable-function:latest

# Push to ECR
docker push 123456789012.dkr.ecr.us-east-1.amazonaws.com/my-durable-function:latest
```

建立已啟用持久性執行的函數：

```
aws lambda create-function \
  --function-name myDurableFunction \
  --package-type Image \
  --code ImageUri=123456789012.dkr.ecr.us-east-1.amazonaws.com/my-durable-
function:latest \
  --role arn:aws:iam::123456789012:role/lambda-execution-role \
  --durable-config '{"ExecutionTimeout": 3600, "RetentionPeriodInDays": 7}'
```

如需搭配 Lambda 使用容器映像的詳細資訊，請參閱《Lambda 開發人員指南》中的建立 Lambda 容器映像。

執行時期考量

SDK 版本管理：在您的部署套件或容器映像中包含耐久的執行 SDK。這可確保函數使用特定的 SDK 版本，並且不受執行時間更新的影響。在 `package.json`或 中釘選 SDK 版本`requirements.txt`，以控制升級時間。

執行期更新： AWS 更新受管執行期，以包含安全性修補程式和錯誤修正。這些更新可能包含新的 SDK 版本。為了避免意外行為，請在部署套件中包含 開發套件，並在部署至生產環境之前進行徹底測試。

容器映像大小：容器映像的最大未壓縮大小為 10 GB。耐用的執行 SDK 為您的映像新增最小大小。使用多階段建置並移除不必要的相依性，以最佳化您的容器。

冷啟動效能：容器映像的冷啟動時間可能比受管執行時間更長。耐用的執行 SDK 對冷啟動效能的影響最小。如果冷啟動延遲對您的應用程式至關重要，請使用佈建並行。

叫用耐用的 Lambda 函數

耐用的 Lambda 函數支援與標準 Lambda 函數相同的調用方法。您可以同步、非同步或透過事件來源映射叫用耐久函數。叫用程序與標準 函數相同，但耐用的 函數為長時間執行和自動狀態管理提供額外的功能。

調用方法

同步叫用：叫用耐久的 函數並等待回應。同步調用受到 Lambda 限制為 15 分鐘 （或更少，取決於設定的函數和執行逾時）。當您需要立即結果或與預期回應APIs 和服務整合時，請使用同步調用。您可以使用等待操作進行有效率的運算，而不會中斷發起人 — 調用會等待整個耐久執行完成。對於等冪性執行啟動，請使用等冪性中所述的執行名稱參數。

```
aws lambda invoke \
  --function-name my-durable-function:1 \
  --cli-binary-format raw-in-base64-out \
  --payload '{"orderId": "12345"}' \
  response.json
```

非同步調用：將事件排入佇列以進行處理，而無需等待回應。Lambda 會將事件置於佇列中並立即傳回。非同步調用支援長達 1 年的執行持續時間。將非同步叫用用於fire-and-forget的情況，或在背景中可能發生處理時。對於等冪性執行開始，請使用等冪性中所述的執行名稱參數。

```
aws lambda invoke \
  --function-name my-durable-function:1 \
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
  --payload '{"orderId": "12345"}' \
  response.json
```

事件來源映射：將 Lambda 設定為在 Amazon SQS、Kinesis 或 DynamoDB 等串流或佇列型服務提供記錄時，自動叫用您的耐久函數。事件來源映射會輪詢事件來源，並使用批次記錄叫用函數。如需搭配耐久函數使用事件來源映射的詳細資訊，包括執行持續時間限制，請參閱搭配持久函數的事件來源映射。

如需每個調用方法的完整詳細資訊，請參閱同步調用和非同步調用。

> **Note**
>
> 耐用的函數支援無效字母佇列 (DLQs) 來處理錯誤，但不支援 Lambda 目的地。設定 DLQ 以從失敗的調用中擷取記錄。

合格 ARNs需求

耐用的函數需要合格的識別符才能叫用。您必須使用版本編號、別名或 叫用耐久函數$LATEST。您可以搭配版本/別名尾碼使用完整的合格 ARN 或函數名稱。您無法使用不合格的識別符 （沒有版本或別名尾碼）。

有效的調用：

```
# Using full ARN with version number
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:1

# Using full ARN with alias
```

```
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:prod

# Using full ARN with $LATEST
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:$LATEST

# Using function name with version number
my-durable-function:1

# Using function name with alias
my-durable-function:prod
```

無效的叫用：

```
# Unqualified ARN (not allowed)
arn:aws:lambda:us-east-1:123456789012:function:my-durable-function

# Unqualified function name (not allowed)
my-durable-function
```

此要求可確保持久的執行在整個生命週期中保持一致。當持久性執行開始時，它會固定到特定的函數版本。如果您的函數暫停並在數小時或數天後恢復，Lambda 會叫用開始執行的相同版本，以確保整個工作流程的程式碼一致性。

最佳實務

將編號版本或別名用於生產耐久函數，而非 $LATEST。編號版本不可變，並支援確定性重播。或者，別名提供穩定的參考，您可以更新以指向新版本，而無需變更調用程式碼。當您更新別名時，新的執行會使用新版本，而進行中的執行會繼續其原始版本。您可以使用 $LATEST進行原型設計，或縮短開發期間的部署時間，了解如果基礎程式碼在執行期間變更，執行可能無法正確重播（甚至失敗）。

了解執行生命週期

當您叫用耐久的函數時，Lambda 會建立持久的執行，可以跨越多個函數叫用：

1. 初始調用：您的調用請求會建立新的耐久執行。Lambda 會指派唯一的執行 ID 並開始處理。
2. 執行和檢查點：當函數執行持久性操作時，開發套件會建立檢查點來追蹤進度。
3. 暫停（如果需要）：如果您的函數使用持久的等待，例如 `wait`或 `waitForCallback`，或自動步驟重試，Lambda 會暫停執行，並在運算時間內停止收費。

4. 恢復：當需要繼續時（包括重試後），Lambda 會再次叫用您的函數。SDK 會重播檢查點日誌，並從執行暫停的位置繼續。
5. 完成：當您的函數傳回最終結果或擲出未處理的錯誤時，持久性執行就會完成。

對於同步調用，發起人會等待整個耐久執行完成，包括任何等待操作。如果執行超過呼叫逾時 (15 分鐘或更短），則呼叫會逾時。對於非同步調用，Lambda 會立即傳回 ，且執行會獨立繼續。使用耐久 APIs 來追蹤執行狀態並擷取最終結果。

從應用程式碼叫用

使用 AWS SDKs從您的應用程式程式碼叫用耐久的函數。叫用程序與標準函數相同：

TypeScript

```
import { LambdaClient, InvokeCommand } from '@aws-sdk/client-lambda';

const client = new LambdaClient({});

// Synchronous invocation
const response = await client.send(new InvokeCommand({
  FunctionName: 'arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
  Payload: JSON.stringify({ orderId: '12345' })
}));

const result = JSON.parse(Buffer.from(response.Payload!).toString());

// Asynchronous invocation
await client.send(new InvokeCommand({
  FunctionName: 'arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
  InvocationType: 'Event',
  Payload: JSON.stringify({ orderId: '12345' })
}));
```

Python

```
import boto3
```

```
import json

client = boto3.client('lambda')

# Synchronous invocation
response = client.invoke(
    FunctionName='arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
    Payload=json.dumps({'orderId': '12345'})
)

result = json.loads(response['Payload'].read())

# Asynchronous invocation
client.invoke(
    FunctionName='arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
    InvocationType='Event',
    Payload=json.dumps({'orderId': '12345'})
)
```

鏈結調用

耐用的 函數可以使用 的 invoke操作叫用其他耐用和非耐用的 函數DurableContext。這會建立鏈結叫用，呼叫函數會等待（暫停）叫用函數完成：

TypeScript

```
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Invoke another durable function and wait for result
    const result = await context.invoke(
      'process-order',
      'arn:aws:lambda:us-east-1:123456789012:function:order-processor:1',
      { orderId: event.orderId }
    );

    return { statusCode: 200, body: JSON.stringify(result) };
  }
);
```

Python

```
@durable_execution
def handler(event, context: DurableContext):
    # Invoke another durable function and wait for result
    result = context.invoke(
        'arn:aws:lambda:us-east-1:123456789012:function:order-processor:1',
        {'orderId': event['orderId']},
        name='process-order'
    )

    return {'statusCode': 200, 'body': json.dumps(result)}
```

鏈結調用會在呼叫函數中建立檢查點。如果呼叫函數中斷，它會使用調用函數的結果從檢查點繼續，而無需重新調用函數。

> **Note**
>
> 不支援跨帳戶鏈結調用。調用函數必須與呼叫函數位於相同的 AWS 帳戶中。

具有耐用函數的事件來源映射

耐用函數適用於所有 Lambda 事件來源映射。設定耐用函數的事件來源映射，與設定標準函數的方式相同。事件來源映射會自動輪詢事件來源，例如 Amazon SQS、Kinesis 和 DynamoDB Streams，並使用批次記錄叫用函數。

事件來源映射適用於使用複雜多步驟工作流程處理串流或佇列的耐久函數。例如，您可以建立耐久的函數，透過重試、外部 API 呼叫和人工核准來處理 Amazon SQS 訊息。

事件來源映射如何叫用耐久函數

事件來源映射會同步叫用持久性函數，等待完全持久性執行完成，然後再處理下一個批次或將記錄標記為已處理。如果總耐久執行時間超過 15 分鐘，則執行會逾時並失敗。事件來源映射會收到逾時例外狀況，並根據其重試組態進行處理。

15 分鐘執行限制

當事件來源映射調用持久性函數時，總持久性執行持續時間不能超過 15 分鐘。此限制適用於從開始到完成的整個持久性執行，而不只是個別函數叫用。

此 15 分鐘限制與 Lambda 函數逾時分開（最長 15 分鐘）。函數逾時控制每個個別調用可以執行的時間長度，而持久的執行逾時控制從執行開始到完成的總經過時間。

範例案例：

- 有效：耐用的函數會處理具有三個步驟的 Amazon SQS 訊息，每個步驟需要 2 分鐘，然後等待 5 分鐘，再完成最後一個步驟。總執行時間：11 分鐘。這是因為總計少於 15 分鐘。
- 無效：耐用函數會處理 Amazon SQS 訊息、在 2 分鐘內完成初始處理，然後等待 20 分鐘進行外部回呼，然後再完成。總執行時間：22 分鐘。這超過 15 分鐘的限制，將會失敗。
- 無效：耐用函數會處理 Kinesis 記錄，其中多個等待操作在步驟之間總計為 30 分鐘。即使每個個別調用都快速完成，總執行時間仍超過 15 分鐘。

> **Important**
>
> 使用事件來源映射時，將您的持久執行逾時設定為 15 分鐘或更少，否則建立事件來源映射將會失敗。如果您的工作流程需要較長的執行時間，請使用如下所述的中介函數模式。

設定事件來源映射

使用 Lambda 主控台 AWS CLI或 AWS SDKs 設定耐用函數的事件來源映射。所有標準事件來源映射屬性都適用於耐用函數：

```
aws lambda create-event-source-mapping \
  --function-name arn:aws:lambda:us-east-1:123456789012:function:my-durable-function:1
 \
  --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
  --batch-size 10 \
  --maximum-batching-window-in-seconds 5
```

為耐用函數設定事件來源映射時，請記得使用合格的 ARN（具有版本編號或別名）。

使用事件來源映射處理錯誤

事件來源映射提供內建錯誤處理，可搭配耐用的 函數運作：

- 重試行為：如果初始調用失敗，事件來源映射會根據其重試組態重試。根據您的需求設定重試嘗試次數上限和重試間隔。
- 無效字母佇列：設定無效字母佇列以擷取所有重試後失敗的記錄。這可防止訊息遺失，並啟用失敗記錄的手動檢查。
- 部分批次失敗：對於 Amazon SQS 和 Kinesis，請使用部分批次失敗報告個別處理記錄，並且只重試失敗的記錄。
- 偏差錯誤：對於 Kinesis 和 DynamoDB 串流，啟用偏差錯誤以分割失敗的批次並隔離有問題的記錄。

> **Note**
>
> 耐用的函數支援無效字母佇列 (DLQs) 來處理錯誤，但不支援 Lambda 目的地。設定 DLQ 以從失敗的調用中擷取記錄。

如需事件來源映射錯誤處理的完整資訊，請參閱事件來源映射。

將中介函數用於長時間執行的工作流程

如果您的工作流程需要超過 15 分鐘才能完成，請在事件來源映射和耐用函數之間使用中介標準 Lambda 函數。中介函數會從事件來源映射接收事件，並以非同步方式叫用耐久函數，移除 15 分鐘的執行限制。

此模式會將事件來源映射的同步調用模型與耐久函數長時間執行的執行模型分離。事件來源映射會叫用中介函數，該函數會在啟動耐久執行後快速傳回。然後，耐用函數會視需要獨立執行（最長 1 年）。

Architecture

中介函數模式使用三個元件：

1. 事件來源映射：輪詢事件來源 (Amazon SQS、Kinesis、DynamoDB Streams)，並使用批次記錄同步叫用中介函數。
2. 中介函數：從事件來源映射接收事件的標準 Lambda 函數，會視需要驗證和轉換資料，並以非同步方式叫用持久函數。此函數會快速完成（通常不到 1 秒），並將控制項傳回至事件來源映射。

3. 耐用函數：使用可長時間執行的複雜多步驟邏輯來處理事件。非同步叫用，因此不受 15 分鐘限制。

實作

中介函數會從事件來源映射接收整個事件，並以非同步方式叫用耐久函數。使用執行名稱參數來確保等冪執行開始，防止事件來源映射重試時重複處理：

TypeScript

```
import { LambdaClient, InvokeCommand } from '@aws-sdk/client-lambda';
import { SQSEvent } from 'aws-lambda';
import { createHash } from 'crypto';

const lambda = new LambdaClient({});

export const handler = async (event: SQSEvent) => {
  // Invoke durable function asynchronously with execution name
  await lambda.send(new InvokeCommand({
    FunctionName: 'arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
    InvocationType: 'Event',
    Payload: JSON.stringify({
      executionName: event.Name,
      event: event
    })
  }));

  return { statusCode: 200 };
};
```

Python

```
import boto3
import json
import hashlib

lambda_client = boto3.client('lambda')

def handler(event, context):
```

```
    # Invoke durable function asynchronously with execution name
    lambda_client.invoke(
        FunctionName='arn:aws:lambda:us-east-1:123456789012:function:my-durable-
function:1',
        InvocationType='Event',
        Payload=json.dumps({
            'executionName': execution_name,
            'event': event["name"]
        })
    )

    return {'statusCode': 200}
```

對於中介函數本身的冪等性，如果事件來源映射重試中介函數，請使用適用於 的 Powertools AWS Lambda 來防止重複叫用持久函數。

耐用函數會接收具有執行名稱的承載，並使用長時間執行的邏輯處理所有記錄：

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';

export const handler = withDurableExecution(
  async (payload: any, context: DurableContext) => {
    const sqsEvent = payload.event;

    // Process each record with complex, multi-step logic
    const results = await context.map(
      sqsEvent.Records,
      async (ctx, record) => {
        const validated = await ctx.step('validate', async () => {
          return validateOrder(JSON.parse(record.body));
        });

        // Wait for external approval (could take hours or days)
        const approval = await ctx.waitForCallback(
          'approval',
          async (callbackId) => {
            await requestApproval(callbackId, validated);
```

```
          },
          { timeout: { hours: 48 } }
        );

        // Complete processing
        return await ctx.step('complete', async () => {
          return completeOrder(validated, approval);
        });
      }
    );

    return { statusCode: 200, processed: results.getResults().length };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
from aws_durable_execution_sdk_python.config import Duration, WaitForCallbackConfig
from collections.abc import Sequence
import json

def validate_order(order_data: dict) -> dict:
    """Validate order data - always passes."""
    return order_data

def request_approval(callback_id: str, validated_order: dict) -> None:
    """Request approval for the order - always passes."""
    pass

def complete_order(validated_order: dict, approval_result: str) -> dict:
    """Complete the order processing - always passes."""
    return validated_order

@durable_execution
def lambda_handler(payload, context: DurableContext):
    sqs_event = payload['event']

    def process_record(
        ctx: DurableContext,
        record: dict,
```

```
        index: int,
        items: Sequence[dict]
    ) -> dict:
        validated = ctx.step(
            lambda _: validate_order(json.loads(record['body'])),
            name=f'validate-{index}'
        )

        approval = ctx.wait_for_callback(
            submitter=lambda callback_id, wait_ctx: request_approval(callback_id,
validated),
            name=f'approval-{index}',
            config=WaitForCallbackConfig(timeout=Duration.from_seconds(172800))
        )

        return ctx.step(
            lambda _: complete_order(validated, approval),
            name=f'complete-{index}'
        )

    results = context.map(
        inputs=sqs_event['Records'],
        func=process_record,
        name='process-records'
    )

    return {
        'statusCode': 200,
        'started': results.started_count,
        'completed': results.success_count,
        'failed': results.failure_count,
        'total': results.total_count
    }
```

關鍵考量

此模式會取消事件來源映射與持久性執行的耦合，以移除 15 分鐘執行限制。中介函數會在啟動耐久執行後立即傳回，允許事件來源映射繼續處理。然後，耐用函數會視需要獨立執行。

中介函數會在叫用持久性函數時成功，而不是在持久性執行完成時成功。如果稍後持久性執行失敗，則事件來源映射不會重試，因為它已成功處理批次。在耐用函數中實作錯誤處理，並為失敗的執行設定無效字母佇列。

使用執行名稱參數來確保等冪執行開始。如果事件來源映射重試中介函數，則耐久函數不會啟動重複的執行，因為執行名稱已存在。

支援的事件來源

耐用函數支援使用事件來源映射的所有 Lambda 事件來源：

- Amazon SQS 佇列 （ 標準和 FIFO)
- Kinesis 串流
- DynamoDB Streams
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- 自我管理的 Apache Kafka
- Amazon MQ (ActiveMQ 和 RabbitMQ)
- Amazon DocumentDB 變更串流

呼叫耐用函數時，所有事件來源類型都受限於 15 分鐘的耐久執行限制。

重試 Lambda 耐用函數

耐用的 函數提供自動重試功能，讓您的應用程式能夠適應暫時性故障。開發套件處理兩個層級的重試：商業邏輯故障的步驟重試，以及基礎設施故障的後端重試。

步驟重試

當步驟中發生未攔截的例外狀況時，軟體開發套件會根據設定的重試策略自動重試步驟。步驟重試是檢查點操作，可讓 SDK 暫停執行並在稍後繼續，而不會遺失進度。

步驟重試行為

下表說明 SDK 在步驟中如何處理例外狀況：

案例	發生的情況	計量影響
步驟中剩餘重試嘗試的例外狀況	SDK 會建立重試的檢查點，並暫停函數。在下次調用時，步驟會重試已設定的退避延遲。	1 操作 + 錯誤承載大小
步驟中的例外狀況，沒有剩餘的重試嘗試	步驟失敗並擲回例外狀況。如果您的處理常式程式碼未擷取此例外狀況，則整個執行會失敗。	1 操作 + 錯誤承載大小

當步驟需要重試時，開發套件會檢查重試狀態，並在沒有其他工作執行時結束 Lambda 調用。這可讓 SDK 實作退避延遲，而不會耗用運算資源。函數會在退避期間後自動恢復。

設定步驟重試策略

設定重試策略以控制步驟處理失敗的方式。您可以指定重試的嘗試次數上限、退避間隔和條件。

最大嘗試次數的指數退避：

TypeScript

```
const result = await context.step('call-api', async () => {
  const response = await fetch('https://api.example.com/data');
  if (!response.ok) throw new Error(`API error: ${response.status}`);
  return await response.json();
}, {
  retryStrategy: (error, attemptCount) => {
    if (attemptCount >= 5) {
      return { shouldRetry: false };
    }
    // Exponential backoff: 2s, 4s, 8s, 16s, 32s (capped at 300s)
    const delay = Math.min(2 * Math.pow(2, attemptCount - 1), 300);
    return { shouldRetry: true, delay: { seconds: delay } };
  }
});
```

Python

```
def retry_strategy(error, attempt_count):
    if attempt_count >= 5:
        return {'should_retry': False}
    # Exponential backoff: 2s, 4s, 8s, 16s, 32s (capped at 300s)
    delay = min(2 * (2 ** (attempt_count - 1)), 300)
    return {'should_retry': True, 'delay': delay}

result = context.step(
    lambda _: call_external_api(),
    name='call-api',
    config=StepConfig(retry_strategy=retry_strategy)
)
```

固定間隔退避：

TypeScript

```
const orders = await context.step('query-orders', async () => {
  return await queryDatabase(event.userId);
}, {
  retryStrategy: (error, attemptCount) => {
    if (attemptCount >= 3) {
      return { shouldRetry: false };
    }
    return { shouldRetry: true, delay: { seconds: 5 } };
  }
});
```

Python

```
def retry_strategy(error, attempt_count):
    if attempt_count >= 3:
        return {'should_retry': False}
    return {'should_retry': True, 'delay': 5}
```

```
orders = context.step(
    lambda _: query_database(event['userId']),
    name='query-orders',
    config=StepConfig(retry_strategy=retry_strategy)
)
```

條件式重試（僅重試特定錯誤）：

TypeScript

```
const result = await context.step('call-rate-limited-api', async () => {
  const response = await fetch('https://api.example.com/data');

  if (response.status === 429) throw new Error('RATE_LIMIT');
  if (response.status === 504) throw new Error('TIMEOUT');
  if (!response.ok) throw new Error(`API_ERROR_${response.status}`);

  return await response.json();
}, {
  retryStrategy: (error, attemptCount) => {
    // Only retry rate limits and timeouts
    const isRetryable = error.message === 'RATE_LIMIT' || error.message ===
 'TIMEOUT';

    if (!isRetryable || attemptCount >= 3) {
      return { shouldRetry: false };
    }

    // Exponential backoff: 1s, 2s, 4s (capped at 30s)
    const delay = Math.min(Math.pow(2, attemptCount - 1), 30);
    return { shouldRetry: true, delay: { seconds: delay } };
  }
});
```

Python

```
def retry_strategy(error, attempt_count):
    # Only retry rate limits and timeouts
```

```
    is_retryable = str(error) in ['RATE_LIMIT', 'TIMEOUT']

    if not is_retryable or attempt_count >= 3:
        return {'should_retry': False}

    # Exponential backoff: 1s, 2s, 4s (capped at 30s)
    delay = min(2 ** (attempt_count - 1), 30)
    return {'should_retry': True, 'delay': delay}

result = context.step(
    lambda _: call_rate_limited_api(),
    name='call-rate-limited-api',
    config=StepConfig(retry_strategy=retry_strategy)
)
```

停用重試：

TypeScript

```
const isDuplicate = await context.step('check-duplicate', async () => {
  return await checkIfOrderExists(event.orderId);
}, {
  retryStrategy: () => ({ shouldRetry: false })
});
```

Python

```
is_duplicate = context.step(
    lambda _: check_if_order_exists(event['orderId']),
    name='check-duplicate',
    config=StepConfig(
        retry_strategy=lambda error, attempt: {'should_retry': False}
    )
)
```

當重試策略傳回 時`shouldRetry: false`，步驟會立即失敗，而不會重試。將此用於不應重試的操作，例如等冪等性檢查或具有無法安全重複之副作用的操作。

步驟外部的例外狀況

當您的處理常式程式碼中發生未攔截的例外狀況，但在任何步驟之外，軟體開發套件會將執行標記為失敗。這可確保正確擷取和報告應用程式邏輯中的錯誤。

案例	發生的情況	計量影響
任何步驟外處理常式程式碼的例外狀況	SDK 會將執行標記為 FAILED，並傳回錯誤。系統不會自動重試例外狀況。	錯誤承載大小

若要啟用易出錯程式碼的自動重試，請使用重試策略將其包裝在步驟中。步驟提供可設定退避的自動重試，而步驟外的程式碼會立即失敗。

後端重試

當 Lambda 遇到基礎設施故障、執行時間錯誤，或開發套件無法與持久的執行服務通訊時，就會發生後端重試。Lambda 會自動重試這些失敗，以協助您的耐用函數從暫時性基礎設施問題中復原。

後端重試案例

當函數遇到下列情況時，Lambda 會自動重試函數：

- 內部服務錯誤 - 當 Lambda 或持久性執行服務傳回 5xx 錯誤時，表示暫時服務問題。
- 限流 - 當函數因並行限制或服務配額而限流時。
- 逾時 - 當 SDK 在逾時期間內無法達到持久的執行服務時。
- 沙盒初始化失敗 - 當 Lambda 無法初始化執行環境時。
- 執行時間錯誤 - 當 Lambda 執行時間遇到函數程式碼以外的錯誤，例如out-of-memory錯誤或程序當機。
- 無效的檢查點字符錯誤 - 當檢查點字符不再有效時，通常是由於服務端狀態變更。

下表說明 SDK 如何處理這些案例：

案例	發生的情況	計量影響
耐用處理常式外的執行時間錯誤 (OOM、逾時、當機)	Lambda 會自動重試調用。開發套件會從最後一個檢查點重播，略過完成的步驟。	錯誤承載大小 + 每次重試 1 個操作
呼叫 / GetDurableExecutionState APIs時發生服務錯誤 (5xx) CheckpointDurableExecution 或逾時	Lambda 會自動重試調用。開發套件會從最後一個檢查點重播。	錯誤承載大小 + 每次重試 1 個操作
呼叫 / GetDurableExecutionState APIs 時調節 (429) CheckpointDurableExecution 或無效的檢查點字符	Lambda 會自動重試具有指數退避的調用。開發套件會從最後一個檢查點重播。	錯誤承載大小 + 每次重試 1 個操作
當 / GetDurableExecutionState APIs時，用戶端錯誤 CheckpointDurableExecution (4xx，429 和無效的字符除外)	SDK 會將執行標記為 FAILED。不會自動重試，因為錯誤表示永久問題。	錯誤承載大小

後端重試會使用指數退避並繼續，直到函數成功或達到執行逾時為止。在重播期間，軟體開發套件會略過已完成的檢查點，並繼續從上次成功的操作執行，確保您的函數不會重新執行已完成的工作。

重試最佳實務

設定重試策略時，請遵循下列最佳實務：

- 設定明確的重試策略 - 不要依賴生產環境中的預設重試行為。為您的使用案例使用適當的最大嘗試次數和退避間隔來設定明確的重試策略。

- 使用條件式重試 - `shouldRetry` 實作邏輯僅重試暫時性錯誤（速率限制、逾時），並在永久錯誤時快速失敗（驗證失敗，找不到）。
- 設定適當的最大嘗試次數 - 在彈性和執行時間之間取得平衡。太多重試可能會延遲故障偵測，而太少會導致不必要的故障。
- 使用指數退避 - 指數退避可減少下游服務的負載，並提高從暫時性故障中復原的可能性。
- 在步驟中包裝容易出錯的程式碼 - 無法自動重試步驟外部的程式碼。使用重試策略在步驟中包裝外部 API 呼叫、資料庫查詢和其他容易出錯的操作。
- 監控重試指標 - 在 Amazon CloudWatch 中追蹤步驟重試操作和執行失敗，以識別模式並最佳化重試策略。

冪等性

耐用的 函數提供內建的冪等性，以便從執行名稱開始執行。當您提供執行名稱時，Lambda 會使用它來防止重複的執行，並啟用呼叫請求的安全重試。根據預設at-least-once執行語意 - 在重播期間，軟體開發套件會傳回檢查點結果，而不會重新執行已完成的步驟，但您的商業邏輯必須等冪，才能在完成之前處理潛在的重試。

> **Note**
>
> Lambda 事件來源映射 (ESM) 不支援啟動時的冪等性。因此，每次叫用（包括重試）都會啟動新的耐久執行。若要使用事件來源映射確保等冪性執行，請在函數程式碼中實作等冪性邏輯，例如使用 Powertools for AWS Lambda或使用一般 Lambda 函數做為代理（分派器），以使用等冪性金鑰（執行名稱參數）叫用持久性函數。

執行名稱

您可以在叫用耐用函數時提供執行名稱。執行名稱做為等冪性金鑰，可讓您安全地重試調用請求，而無需建立重複的執行。如果您未提供名稱，Lambda 會自動產生唯一的執行 ID。

執行名稱在您的帳戶和區域中必須是唯一的。當您使用已存在的執行名稱叫用函數時，Lambda 行為取決於現有執行的狀態以及承載是否相符。

冪等行為

下表說明 Lambda 如何處理調用請求，取決於您是否提供執行名稱、現有的執行狀態，以及承載是否相符：

案例	提供的名稱？	現有執行狀態	承載是否相同？	Behavior (行為)
1	否	N/A	N/A	新的執行已開始：Lambda 會產生唯一的執行 ID，並啟動新的執行
2	是	從未存在或保留過期	N/A	新的執行已開始：Lambda 使用提供的名稱啟動新的執行
3	是	執行中	是	等冪啟動：Lambda 會傳回現有的執行資訊，而不會啟動重複的執行資訊。對於同步調用，這可做為重新連接至執行中的執行
4	是	執行中	否	錯誤：Lambda 傳回`DurableExecutionAlreadyExists` 錯誤，因為具有此名稱的執行已使用不同的承載執行
5	是	關閉（成功、失敗、停止或逾時）	是	等冪啟動：Lambda 會傳回現有的執行資訊，而不啟動新的執行。傳回已關閉的執行結果
6	是	關閉（成功、失敗、停止或逾時）	否	錯誤：Lambda 傳回`DurableExecutionAlreadyExists` 錯

案例	提供的名稱？	現有執行狀態	承載是否相同？	Behavior (行為)
				誤，因為具有此名稱的執行已在不同的承載下完成

注意

案例 3 和 5 示範等冪行為，其中 Lambda 透過傳回現有的執行資訊而非建立重複項目，安全地處理重複的調用請求。

步驟冪等性

根據預設at-least-once執行語意。當您的函數在等待、回呼或失敗後重播時，軟體開發套件會根據檢查點日誌檢查每個步驟。對於已完成的步驟，開發套件會傳回檢查點結果，而不會重新執行步驟邏輯。不過，如果步驟失敗或函數在步驟完成之前中斷，則該步驟可能會執行多次。

您在步驟中包裝的業務邏輯必須等冪，才能處理潛在的重試。使用等冪性金鑰，以確保付款或資料庫寫入等操作只會執行一次，即使步驟重試也一樣。

範例：在步驟中使用等冪索引鍵

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';
import { randomUUID } from 'crypto';

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Generate idempotency key once
    const idempotencyKey = await context.step('generate-key', async () => {
      return randomUUID();
    });

    // Use idempotency key in payment API to prevent duplicate charges
    const payment = await context.step('process-payment', async () => {
      return paymentAPI.charge({
        amount: event.amount,
        idempotencyKey: idempotencyKey
```

```
      });
    });

    return { statusCode: 200, payment };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
import uuid

@durable_execution
def handler(event, context: DurableContext):
    # Generate idempotency key once
    idempotency_key = context.step(
        lambda _: str(uuid.uuid4()),
        name='generate-key'
    )

    # Use idempotency key in payment API to prevent duplicate charges
    payment = context.step(
        lambda _: payment_api.charge(
            amount=event['amount'],
            idempotency_key=idempotency_key
        ),
        name='process-payment'
    )

    return {'statusCode': 200, 'payment': payment}
```

您可以將執行模式設定為 ，將步驟設定為at-most-once的執行語意AT_MOST_ONCE_PER_RETRY。這可確保每次重試嘗試時，步驟最多執行一次，但如果函數在步驟完成之前中斷，則可能完全無法執行。

SDK 透過驗證該步驟名稱和順序在重播期間符合檢查點日誌，強制執行確定性重播。如果您的程式碼嘗試以不同的順序或以不同的名稱執行步驟，開發套件會擲回NonDeterministicExecutionError。

重播如何搭配已完成的步驟運作：

1. 第一次叫用：函數執行步驟 A、建立檢查點，然後等待
2. 第二次調用（等待後）：函數從頭開始重播，步驟 A 會立即傳回檢查點結果，而無需重新執行，然後繼續步驟 B
3. 第三次叫用（在另一個等待之後）：函數從頭開始重播，步驟 A 和 B 會立即傳回檢查點結果，然後繼續步驟 C

此重播機制可確保已完成的步驟不會重新執行，但您的商業邏輯仍必須等冪才能在完成之前處理重試。

測試 Lambda 耐用函數

AWS 為耐用函數提供專用測試 SDKs，可讓您在本機和雲端執行和檢查執行。為您的語言安裝測試 SDK：

TypeScript

```
npm install --save-dev @aws/aws-durable-execution-sdk-js-testing
```

如需完整的文件和範例，請參閱 GitHub 上的 TypeScript 測試 SDK。

Python

```
pip install aws-durable-execution-sdk-python-testing
```

如需完整的文件和範例，請參閱 GitHub 上的 Python 測試 SDK。

測試 SDK 提供兩種測試模式：快速單元測試的本機測試，以及針對已部署函數的整合測試的雲端測試。

本機測試

本機測試會在開發環境中執行您的耐久函數，而不需要部署的資源。測試執行器會直接執行函數程式碼，並擷取所有要檢查的操作。

針對單元測試、測試驅動的開發和 CI/CD 管道使用本機測試。測試會在本機執行，無需網路延遲或額外費用。

測試範例：

TypeScript

```
import { withDurableExecution } from '@aws/aws-durable-execution-sdk-js';
import { DurableFunctionTestRunner } from '@aws/aws-durable-execution-sdk-js-
testing';

const handler = withDurableExecution(async (event, context) => {
  const result = await context.step('calculate', async () => {
    return event.a + event.b;
  });
  return result;
});

test('addition works correctly', async () => {
  const runner = new DurableFunctionTestRunner({ handler });
  const result = await runner.run({ a: 5, b: 3 });

  expect(result.status).toBe('SUCCEEDED');
  expect(result.result).toBe(8);

  const step = result.getStep('calculate');
  expect(step.result).toBe(8);
});
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
from aws_durable_execution_sdk_python_testing import DurableFunctionTestRunner
from aws_durable_execution_sdk_python.execution import InvocationStatus

@durable_execution
def handler(event: dict, context: DurableContext) -> int:
    result = context.step(lambda _: event["a"] + event["b"], name="calculate")
    return result

def test_addition():
    runner = DurableFunctionTestRunner(handler=handler)
    with runner:
```

```
        result = runner.run(input={"a": 5, "b": 3}, timeout=10)

    assert result.status is InvocationStatus.SUCCEEDED
    assert result.result == 8

    step = result.get_step("calculate")
    assert step.result == 8
```

測試執行器會擷取執行狀態，包括最終結果、個別步驟結果、等待操作、回呼和任何錯誤。您可以依名稱檢查操作，或逐一查看所有操作，以驗證執行行為。

執行存放區

測試 SDK 使用執行存放區來保留測試執行資料。根據預設，測試會使用快速且不需要清除的記憶體內存放區。若要偵錯或分析執行歷史記錄，您可以使用將執行儲存為 JSON 檔案的檔案系統存放區。

記憶體內存放區（預設）：

記憶體內存放區會在測試執行期間將執行資料保留在記憶體中。當測試完成時，資料會遺失，因此非常適合標準單元測試和 CI/CD 管道，您不需要在測試完成後檢查執行。

檔案系統存放區：

檔案系統存放區會將執行資料保留為 JSON 檔案至磁碟。每個執行都會儲存在個別的檔案中，以便在測試完成後輕鬆檢查執行歷史記錄。在偵錯複雜測試失敗或分析執行模式時，請使用 檔案系統存放區。

使用環境變數設定存放區：

```
# Use filesystem store
export AWS_DEX_STORE_TYPE=filesystem
export AWS_DEX_STORE_PATH=./test-executions

# Run tests
pytest tests/
```

執行檔案會以已淨化的名稱存放，並包含完整的執行狀態，包括操作、檢查點和結果。如果儲存目錄不存在，檔案系統存放區會自動建立儲存目錄。

雲端測試

雲端測試會叫用 中部署的耐用函數 AWS ，並使用 Lambda API 擷取其執行歷史記錄。使用雲端測試，透過真正的 AWS 服務和組態來驗證類似生產環境中的行為。

雲端測試需要已部署的函數和 AWS 登入資料，具有叫用函數和讀取執行歷史記錄的許可：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction",
                "lambda:GetDurableExecution",
                "lambda:GetDurableExecutionHistory"
            ],
            "Resource": "arn:aws:lambda:region:account-id:function:function-name"
        }
    ]
}
```

雲端測試範例：

TypeScript

```
import { DurableFunctionCloudTestRunner } from '@aws/aws-durable-execution-sdk-js-
testing';

test('deployed function processes orders', async () => {
  const runner = new DurableFunctionCloudTestRunner({
    functionName: 'order-processor',
    region: 'us-east-1'
  });

  const result = await runner.run({ orderId: 'order-123' });

  expect(result.status).toBe('SUCCEEDED');
  expect(result.result.status).toBe('completed');
});
```

Python

```
from aws_durable_execution_sdk_python_testing import (
    DurableFunctionCloudTestRunner,
    DurableFunctionCloudTestRunnerConfig
)

def test_deployed_function():
    config = DurableFunctionCloudTestRunnerConfig(
        function_name="order-processor",
        region="us-east-1"
    )
    runner = DurableFunctionCloudTestRunner(config=config)

    result = runner.run(input={"orderId": "order-123"})

    assert result.status is InvocationStatus.SUCCEEDED
    assert result.result["status"] == "completed"
```

雲端測試會叫用實際部署的函數，並從中擷取執行歷史記錄 AWS。這可讓您驗證與其他 AWS 服務的整合、驗證效能特性，以及使用類似生產的資料和組態進行測試。

測試項目

透過驗證執行結果、操作行為和錯誤處理來測試耐用的函數。專注於商業邏輯正確性，而不是實作詳細資訊。

驗證執行結果：檢查函數是否傳回指定輸入的預期值。測試成功執行和錯誤案例，以確保函數適當地處理無效的輸入。

檢查操作執行：確認步驟、等待和回呼如預期般執行。檢查步驟結果，以確保中繼操作產生正確的值。驗證等待操作是否設定適當的逾時，以及回呼是否使用正確的設定建立。

測試錯誤處理：當指定無效的輸入時，使用描述性錯誤訊息來驗證函數是否正確失敗。模擬暫時性失敗並確認操作重試正確，以測試重試行為。檢查永久故障是否不會觸發不必要的重試。

驗證工作流程：對於多步驟工作流程，驗證操作以正確的順序執行。測試條件式分支，以確保不同的執行路徑正常運作。驗證平行操作同時執行並產生預期結果。

SDK 文件儲存庫包含廣泛的測試模式範例，包括多步驟工作流程、錯誤案例、逾時處理和輪詢模式。

測試策略

在開發期間和 CI/CD 管道中使用單元測試的本機測試。本機測試會快速執行，不需要 AWS 登入資料，並提供程式碼變更的立即意見回饋。撰寫本機測試，以驗證商業邏輯、錯誤處理和操作行為。

在部署到生產環境之前，使用雲端測試進行整合測試。雲端測試會驗證具有真實 AWS 服務和組態的行為、驗證效能特性，以及測試end-to-end工作流程。在預備環境中執行雲端測試，以便在整合問題到達生產環境之前加以發現。

在本機測試中模擬外部相依性，以隔離函數邏輯並保持快速測試。使用雲端測試來驗證與資料庫、APIs和其他 服務等外部 AWS 服務的實際整合。

撰寫重點測試來驗證一個特定行為。使用描述性測試名稱來說明正在測試的內容。將相關測試分組在一起，並針對常見的設定程式碼使用測試固定裝置。保持簡單的測試，並避免難以理解的複雜測試邏輯。

偵錯失敗

當測試失敗時，請檢查執行結果以了解發生錯誤。檢查執行狀態，以查看函數是否成功、失敗或逾時。讀取錯誤訊息以了解失敗原因。

檢查個別操作結果，找出行為偏離預期的位置。檢查步驟結果以查看產生了哪些值。驗證操作排序，以確認依預期順序執行的操作。計數操作以確保建立正確數量的步驟、等待和回呼。

常見問題包括可在重播時產生不同結果的非確定性程式碼、透過重播期間中斷的全域變數共用狀態，以及因條件式邏輯錯誤而缺少操作。使用標準除錯器和記錄來逐步執行函數程式碼和追蹤執行流程。

對於雲端測試，請檢查 CloudWatch Logs 中的執行歷史記錄，以查看詳細的操作日誌。使用追蹤來追蹤跨 服務的執行流程，並識別瓶頸。

監控耐用的 函數

您可以使用 CloudWatch 指標、CloudWatch Logs 和追蹤來監控耐用的函數。由於耐用的函數可以長時間執行並跨越多個函數叫用，因此監控它們需要了解其獨特的執行模式，包括檢查點、狀態轉換和重播行為。

CloudWatch 指標

Lambda 會自動將指標發佈至 CloudWatch，無需額外費用。耐用函數提供超出標準 Lambda 指標的額外指標，協助您監控長時間執行的工作流程、狀態管理和資源使用率。

耐用的執行指標

Lambda 會針對持久性執行發出下列指標：

指標	Description
ApproximateRunningDurableExecutions	處於 RUNNING 狀態的耐久執行數量
ApproximateRunningDurableExecutionsUtilization	目前使用中您帳戶的最大執行中耐久執行配額的百分比
DurableExecutionDuration	持久執行保持為 RUNNING 狀態的經過時間，以毫秒為單位
DurableExecutionStarted	啟動的持久性執行數量
DurableExecutionStopped	使用 StopDurableExecution API 停止的耐久執行數目
DurableExecutionSucceeded	成功完成的持久性執行數量
DurableExecutionFailed	因失敗而完成的耐久執行數量
DurableExecutionTimedOut	超過其設定執行逾時的持久執行數量
DurableExecutionOperations	在持久性執行中執行的累計操作數 （上限：3，000)
DurableExecutionStorageWrittenBytes	持久執行持續的累積資料量，以位元組為單位 （上限：100 MB)

CloudWatch 指標

Lambda 會發出耐用函數的標準調用、效能和並行指標。由於耐久的執行可能會在通過檢查點和重播時跨越多個函數叫用，因此這些指標的行為與標準函數不同：

- 調用：計算每個函數調用，包括重播。單一耐久執行可以產生多個調用資料點。

- 持續時間：分別測量每個函數叫用。DurableExecutionDuration 用於單一持久性執行所花費的總時間。
- 錯誤：追蹤函數叫用失敗。DurableExecutionFailed 用於執行層級失敗。

如需標準 Lambda 指標的完整清單，請參閱 Lambda 函數的指標類型。

建立 CloudWatch 警示

建立 CloudWatch 警示，以在指標超過閾值時通知您。常見警示包括：

- ApproximateRunningDurableExecutionsUtilization 超過配額的 80%
- DurableExecutionFailed 增加超過閾值
- DurableExecutionTimedOut 表示執行逾時
- DurableExecutionStorageWrittenBytes 接近儲存限制

如需詳細資訊，請參閱使用 CloudWatch 警示。

EventBridge 事件

Lambda 會將持久的執行狀態變更事件發佈至 EventBridge。您可以使用這些事件來觸發工作流程、傳送通知，或追蹤耐久函數的執行生命週期變更。

持久性執行狀態變更事件

每當持久性執行變更狀態時，Lambda 都會向 EventBridge 發出事件。這些事件具有下列特性：

- 來源：aws.lambda
- 詳細資訊類型： Durable Execution Status Change

狀態變更事件會針對下列執行狀態發佈：

- RUNNING - 執行已開始
- SUCCEEDED - 執行成功完成
- STOPPED - 使用 StopDurableExecution API 停止執行
- FAILED - 執行失敗並發生錯誤
- TIMED_OUT - 執行超過設定的逾時

下列範例顯示持久的執行狀態變更事件：

```
{
  "version": "0",
  "id": "d019b03c-a8a3-9d58-85de-241e96206538",
  "detail-type": "Durable Execution Status Change",
  "source": "aws.lambda",
  "account": "123456789012",
  "time": "2025-11-20T13:08:22Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
    "durableExecutionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST/durable-execution/090c4189-b18b-4296-9d0c-cfd01dc3a122/9f7d84c9-ea3d-3ffc-
b3e5-5ec51c34ffc9",
    "durableExecutionName": "order-123",
    "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:2",
    "status": "RUNNING",
    "startTimestamp": "2025-11-20T13:08:22.345Z"
  }
}
```

對於終端機狀態 (SUCCEEDED、STOPPED、FAILED、TIMED_OUT)，事件包含指出執行完成時間endTimestamp的欄位。

建立 EventBridge 規則

建立規則，將持久的執行狀態變更事件路由到 Amazon Simple Notification Service、Amazon Simple Queue Service 或其他 Lambda 函數等目標。

下列範例會建立符合所有持久執行狀態變更的規則：

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"]
}
```

下列範例會建立僅符合失敗執行的規則：

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"],
  "detail": {
    "status": ["FAILED"]
  }
}
```

下列範例會建立符合特定函數狀態變更的規則：

```
{
  "source": ["aws.lambda"],
  "detail-type": ["Durable Execution Status Change"],
  "detail": {
    "functionArn": [{
      "prefix": "arn:aws:lambda:us-east-1:123456789012:function:my-function"
    }]
  }
}
```

如需建立規則的詳細資訊，請參閱《EventBridge 使用者指南》中的 Amazon EventBridge 教學課程。EventBridge

AWS X-Ray 追蹤

您可以在耐用的函數上啟用 X-Ray 追蹤。Lambda 會將 X-Ray 追蹤標頭傳遞至持久性執行，讓您追蹤整個工作流程的請求。

若要使用 Lambda 主控台啟用 X-Ray； 追蹤，請選擇您的函數，然後選擇組態、監控和操作工具，然後在 X-Ray 下開啟主動追蹤。

若要使用 啟用 X-Ray 追蹤 AWS CLI：

```
aws lambda update-function-configuration \
    --function-name my-durable-function \
    --tracing-config Mode=Active
```

若要使用 啟用 AWS X-Ray 追蹤 AWS SAM：

```
Resources:
  MyDurableFunction:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      DurableConfig:
        ExecutionTimeout: 3600
```

如需 X-Ray 的詳細資訊，請參閱 AWS X-Ray 開發人員指南。

Lambda 耐用函數的最佳實務

耐用函數使用重播型執行模型，需要與傳統 Lambda 函數不同的模式。遵循這些最佳實務來建置可靠且符合成本效益的工作流程。

寫入確定性程式碼

在重播期間，您的函數會從頭開始執行，並且必須遵循與原始執行相同的執行路徑。持久性操作之外的程式碼必須具有決定性，在相同的輸入下產生相同的結果。

以步驟包裝非確定性操作：

- 隨機數產生和 UUIDs
- 目前時間或時間戳記
- 外部 API 呼叫和資料庫查詢
- 檔案系統操作

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';
import { randomUUID } from 'crypto';
```

```
export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Generate transaction ID inside a step
    const transactionId = await context.step('generate-transaction-id', async () =>
 {
      return randomUUID();
    });

    // Use the same ID throughout execution, even during replay
    const payment = await context.step('process-payment', async () => {
      return processPayment(event.amount, transactionId);
    });

    return { statusCode: 200, transactionId, payment };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
import uuid

@durable_execution
def handler(event, context: DurableContext):
    # Generate transaction ID inside a step
    transaction_id = context.step(
        lambda _: str(uuid.uuid4()),
        name='generate-transaction-id'
    )

    # Use the same ID throughout execution, even during replay
    payment = context.step(
        lambda _: process_payment(event['amount'], transaction_id),
        name='process-payment'
    )

    return {'statusCode': 200, 'transactionId': transaction_id, 'payment': payment}
```

Important (重要)

請勿使用全域變數或關閉在步驟之間共用狀態。透過傳回值傳遞資料。重播期間全域狀態中斷，因為步驟會傳回快取的結果，但全域變數會重設。

避免關閉變動：在關閉中擷取的變數可能會在重播期間遺失變動。步驟會傳回快取的結果，但步驟外的變數更新不會重播。

TypeScript

```
// # WRONG: Mutations lost on replay
export const handler = withDurableExecution(async (event, context) => {
  let total = 0;

  for (const item of items) {
    await context.step(async () => {
      total += item.price; // ## Mutation lost on replay!
      return saveItem(item);
    });
  }

  return { total }; // Inconsistent value!
});

// # CORRECT: Accumulate with return values
export const handler = withDurableExecution(async (event, context) => {
  let total = 0;

  for (const item of items) {
    total = await context.step(async () => {
      const newTotal = total + item.price;
      await saveItem(item);
      return newTotal; // Return updated value
    });
  }

  return { total }; // Consistent!
});

// # EVEN BETTER: Use map for parallel processing
export const handler = withDurableExecution(async (event, context) => {
  const results = await context.map(
    items,
    async (ctx, item) => {
```

```
      await ctx.step(async () => saveItem(item));
      return item.price;
    }
  );

  const total = results.getResults().reduce((sum, price) => sum + price, 0);
  return { total };
});
```

Python

```
# # WRONG: Mutations lost on replay
@durable_execution
def handler(event, context: DurableContext):
    total = 0

    for item in items:
        context.step(
            lambda _: save_item_and_mutate(item, total),  # ## Mutation lost on
 replay!
            name=f'save-item-{item["id"]}'
        )

    return {'total': total}  # Inconsistent value!

# # CORRECT: Accumulate with return values
@durable_execution
def handler(event, context: DurableContext):
    total = 0

    for item in items:
        total = context.step(
            lambda _: save_item_and_return_total(item, total),
            name=f'save-item-{item["id"]}'
        )

    return {'total': total}  # Consistent!

# # EVEN BETTER: Use map for parallel processing
@durable_execution
def handler(event, context: DurableContext):
```

```
    def process_item(ctx, item):
        ctx.step(lambda _: save_item(item))
        return item['price']

    results = context.map(items, process_item)
    total = sum(results.get_results())

    return {'total': total}
```

等冪性的設計

由於重試或重播，操作可能會執行多次。非等冪性操作會導致重複的副作用，例如向客戶收費兩次或傳送多封電子郵件。

使用等冪符記：在步驟中產生符記，並將其包含在外部 API 呼叫中，以防止重複操作。

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Generate idempotency token once
    const idempotencyToken = await context.step('generate-idempotency-token', async
 () => {
      return crypto.randomUUID();
    });

    // Use token to prevent duplicate charges
    const charge = await context.step('charge-payment', async () => {
      return paymentService.charge({
        amount: event.amount,
        cardToken: event.cardToken,
        idempotencyKey: idempotencyToken
      });
    });

    return { statusCode: 200, charge };
  }
```

```
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext
import uuid

@durable_execution
def handler(event, context: DurableContext):
    # Generate idempotency token once
    idempotency_token = context.step(
        lambda _: str(uuid.uuid4()),
        name='generate-idempotency-token'
    )

    # Use token to prevent duplicate charges
    def charge_payment(_):
        return payment_service.charge(
            amount=event['amount'],
            card_token=event['cardToken'],
            idempotency_key=idempotency_token
        )

    charge = context.step(charge_payment, name='charge-payment')

    return {'statusCode': 200, 'charge': charge}
```

使用at-most-once語意：對於絕不能重複的關鍵操作（金融交易、庫存扣除），請設定at-most-once的執行模式。

TypeScript

```
// Critical operation that must not duplicate
await context.step('deduct-inventory', async () => {
  return inventoryService.deduct(event.productId, event.quantity);
}, {
  executionMode: 'AT_MOST_ONCE_PER_RETRY'
});
```

Python

```
# Critical operation that must not duplicate
context.step(
    lambda _: inventory_service.deduct(event['productId'], event['quantity']),
    name='deduct-inventory',
    config=StepConfig(execution_mode='AT_MOST_ONCE_PER_RETRY')
)
```

資料庫冪等性：使用check-before-write模式、條件更新或 upsert 操作，以防止重複記錄。

有效率地管理狀態

每個檢查點都會將狀態儲存為持久性儲存。大型狀態物件會增加成本、緩慢檢查點和影響效能。僅儲存必要的工作流程協調資料。

保持最小狀態：

- 存放 IDs和參考，而非完整物件
- 視需要在步驟內擷取詳細資訊
- 針對大型資料使用 Amazon S3 或 DynamoDB，以 狀態傳遞參考
- 避免在步驟之間傳遞大型承載

TypeScript

```
import { withDurableExecution, DurableContext } from '@aws/durable-execution-sdk-
js';

export const handler = withDurableExecution(
  async (event: any, context: DurableContext) => {
    // Store only the order ID, not the full order object
    const orderId = event.orderId;

    // Fetch data within each step as needed
```

```
    await context.step('validate-order', async () => {
      const order = await orderService.getOrder(orderId);
      return validateOrder(order);
    });

    await context.step('process-payment', async () => {
      const order = await orderService.getOrder(orderId);
      return processPayment(order);
    });

    return { statusCode: 200, orderId };
  }
);
```

Python

```
from aws_durable_execution_sdk_python import durable_execution, DurableContext

@durable_execution
def handler(event, context: DurableContext):
    # Store only the order ID, not the full order object
    order_id = event['orderId']

    # Fetch data within each step as needed
    context.step(
        lambda _: validate_order(order_service.get_order(order_id)),
        name='validate-order'
    )

    context.step(
        lambda _: process_payment(order_service.get_order(order_id)),
        name='process-payment'
    )

    return {'statusCode': 200, 'orderId': order_id}
```

設計有效的步驟

步驟是耐用函數中的基本工作單位。精心設計的步驟可讓工作流程更易於理解、偵錯和維護。

步驟設計原則：

- 使用描述性名稱 - 類似 的名稱，validate-order而不是step1讓日誌和錯誤更容易理解
- 將名稱保持靜態 - 請勿將動態名稱與時間戳記或隨機值搭配使用。步驟名稱必須具有決定性才能重播
- 平衡精細程度 - 將複雜的操作分解為重點步驟，但避免增加檢查點負荷的過小步驟
- 群組相關操作 - 應該一起成功或失敗的操作屬於同一步驟

有效率地使用等待操作

等待操作會暫停執行，而不會耗用資源或產生成本。使用它們，而不是讓 Lambda 保持執行狀態。

以時間為基礎的等待：使用 context.wait() 處理延遲，而非 setTimeout或 sleep。

外部回呼：等待外部系統context.waitForCallback()時使用 。一律設定逾時以防止無限期等待。

輪詢：使用 context.waitForCondition()搭配指數退避來輪詢外部服務，而不會讓它們過多。

TypeScript

```
// Wait 24 hours without cost
await context.wait({ seconds: 86400 });

// Wait for external callback with timeout
const result = await context.waitForCallback(
  'external-job',
  async (callbackId) => {
    await externalService.submitJob({
      data: event.data,
      webhookUrl: `https://api.example.com/callbacks/${callbackId}`
    });
  },
  { timeout: { seconds: 3600 } }
);
```

Python

```
# Wait 24 hours without cost
context.wait(86400)

# Wait for external callback with timeout
result = context.wait_for_callback(
    lambda callback_id: external_service.submit_job(
        data=event['data'],
        webhook_url=f'https://api.example.com/callbacks/{callback_id}'
    ),
    name='external-job',
    config=WaitForCallbackConfig(timeout_seconds=3600)
)
```

其他考量

錯誤處理：重試暫時性故障，例如網路逾時和速率限制。請勿重試永久失敗，例如無效的輸入或身分驗證錯誤。使用適當的最大嘗試次數和退避率來設定重試策略。如需詳細範例，請參閱錯誤處理和重試。

效能：透過儲存參考而非完整承載，將檢查點大小降至最低。使用 `context.parallel()`和 `context.map()` 同時執行獨立操作。批次相關操作，以減少檢查點額外負荷。

版本控制：使用版本編號或別名叫用函數，將執行釘選到特定的程式碼版本。確保新的程式碼版本可以處理較舊版本的狀態。請勿重新命名步驟或以中斷重播的方式變更其行為。

序列化：將 JSON 相容類型用於操作輸入和結果。將日期轉換為 ISO 字串，並將自訂物件轉換為純物件，然後再將其傳遞至持久的操作。

監控：啟用具有執行 IDs結構化記錄。針對錯誤率和執行持續時間設定 CloudWatch 警示。使用追蹤來識別瓶頸。如需詳細指引，請參閱監控和偵錯。

測試：測試快樂路徑、錯誤處理和重播行為。回呼和等待的測試逾時案例。使用本機測試來縮短反覆運算時間。如需詳細指引，請參閱測試耐用函數。

要避免的常見錯誤：不要巢狀`context.step()`化呼叫，請改用子內容。在步驟中包裝非確定性操作。一律設定回呼的逾時。平衡步驟精細程度與檢查點額外負荷。儲存參考，而不是處於 狀態的大型物件。

其他資源

- Python SDK 文件 - 完整的 API 參考、測試模式和進階範例

- TypeScript SDK 文件 - 完整的 API 參考、測試模式和進階範例

Lambda 受管執行個體

Lambda 受管執行個體可讓您在目前世代的 Amazon EC2 執行個體上執行 Lambda 函數，包括 Graviton4、網路最佳化執行個體和其他特殊運算選項，而無需管理執行個體生命週期、作業系統和語言執行時間修補、路由、負載平衡或擴展政策。使用 Lambda 受管執行個體，您可以受益於 EC2 定價優勢，包括 EC2 Savings Plans 和預留執行個體。

如需支援的執行個體類型清單，請前往 AWS Lambda 定價頁面，然後選取您的 AWS 區域。

關鍵功能

Lambda 受管執行個體提供下列功能：

- 選擇合適的執行個體 - 根據效能和成本需求選擇適當的執行個體，包括存取最新的 CPUs，例如 Graviton4、可設定的記憶體 CPU 比率和高頻寬聯網。
- 自動佈建 AWS 自動佈建適當的執行個體，並啟動函數執行環境。
- 動態擴展 - 執行個體會根據函數流量模式動態擴展。
- 全受管體驗 - AWS 使用您熟悉的相同廣泛事件來源整合來處理基礎設施管理、擴展、修補和路由。

何時使用 Lambda 受管執行個體

針對下列使用案例考慮 Lambda 受管執行個體：

- 高磁碟區可預測工作負載 - 非常適合穩定狀態的工作負載，而不會發生意外流量尖峰。Lambda 受管執行個體預設會在五分鐘內擴展以處理加倍的流量。
- 效能關鍵型應用程式 - 存取最新的 CPUs、不同的記憶體 CPU 比率和高網路輸送量
- 法規要求 - 精細控管需求，可控制 VPC 和執行個體放置
- 各種應用程式 - 事件驅動型應用程式、媒體/資料處理、Web 應用程式和舊版工作負載遷移至無伺服器

運作方式

Lambda 受管執行個體使用容量提供者做為執行函數的基礎：

1. 建立容量提供者 - 透過指定 VPC 組態和選用的執行個體需求和擴展組態，定義函數執行的位置

2. 建立函數 - 照常建立 Lambda 函數，並將其連接至容量提供者

3. 發佈函數版本 - 發佈後，函數版本會在容量提供者執行個體上變成作用中

當您使用容量提供者發佈函數版本時，Lambda 會在您的帳戶中啟動受管執行個體。根據預設，它會針對 AZ 彈性啟動三個執行個體，並在標記函數版本 ACTIVE 之前啟動三個執行環境。如果您將函數連接至已執行其他函數的現有容量提供者，且可用執行個體已有容納新函數執行環境的容量，則 Lambda 可能不會啟動新執行個體。

並行模型

Lambda 受管執行個體支援多並行調用，其中一個執行環境可以同時處理多個調用。這與 Lambda（預設）運算類型不同，它提供單一並行模型，其中一個執行環境一次最多可以執行一個調用。多並行可提高基礎 EC2 執行個體的使用率，對 Web 服務或批次任務等 IO 密集型應用程式特別有益。此執行模型變更表示執行緒安全性、狀態管理和內容隔離必須根據執行時間以不同方式處理。

租用和隔離

Lambda（預設）運算類型是多租戶，利用 Firecracker microVM 技術在共用 Lambda 機群上執行的執行環境之間提供隔離。Lambda 受管執行個體會在您的帳戶中執行，提供最新的 EC2 硬體和定價選項。受管執行個體使用 EC2 Nitro 執行個體上執行的容器來提供隔離，而不是 Firecracker。容量提供者可做為 Lambda 函數的安全界限。函數會在執行個體內的容器中執行。

了解受管執行個體

Lambda 受管執行個體函數會在您帳戶中的 EC2 受管執行個體上執行。這些執行個體完全由 Lambda 管理，這表示相較於標準 EC2 執行個體，您對這些執行個體具有有限的許可。您可以透過以下方式識別帳戶中的 Lambda 受管執行個體：

- EC2 `DescribeInstances`輸出中 `Operator` 欄位的存在
- 執行個體上的`aws:lambda:capacity-provider`標籤

您無法直接在這些執行個體上執行標準 EC2 操作，例如手動終止它們。若要銷毀受管執行個體，請刪除相關聯的容量提供者。然後，Lambda 會在容量提供者刪除程序中終止執行個體。

定價

Lambda 受管執行個體使用以 EC2-based定價，除了 EC2 執行個體成本之外，還需支付 15% 的管理費。此定價模式支援 EC2 Savings Plans、預留執行個體，以及套用至 EC2 用量的任何其他定價折扣。如需其他詳細資訊，請參閱定價頁面：https：//https://aws.amazon.com/lambda/pricing/

重要：EC2 定價折扣僅適用於基礎 EC2 運算，不適用於管理費。

Lambda 受管執行個體與 Lambda （預設） 運算類型有何不同

相較於 Lambda （預設），Lambda 受管執行個體會變更 Lambda 處理請求的方式。

主要差異：

	Lambda （預設）	Lambda 受管執行個體
並行模型	單一並行模型，其中一個執行環境一次最多可支援一個調用	多並行調用，其中一個執行環境可以同時處理多個調用，提高輸送量，特別是對於 IO 密集型應用程式
租用和隔離	多租戶，使用 Firecracker microVM 技術在共用 Lambda 機群上執行的執行環境之間提供隔離	在您的帳戶中執行 ，使用 EC2 Nitro 提供隔離。容量提供者做為安全界限，在執行個體內的容器中執行 函數
定價方式	每個請求持續時間定價	使用 EC2 定價模型的執行個體型定價，包括隨需執行個體和預留執行個體，以及 Compute Savings Plans 等節省選項
擴展行為	當沒有免費的執行環境來處理傳入呼叫（冷啟動）時擴展。無流量擴展至零	僅根據 CPU 資源使用率以非同步方式擴展，無需冷啟動。擴展到沒有流量設定的最小執行環境
最適合	具有高載流量的函數，可以處理一些冷啟動時間，或沒有持續負載的應用程式，受益於擴展到零	當您想要 EC2 的彈性、定價計劃和硬體選項時，大量可預測流量函數

後續步驟

- 了解 Lambda 受管執行個體的容量提供者
- 了解 Lambda 受管執行個體的擴展
- 檢閱 Java、Node.js 和 Python 的執行時間特定指南
- 為您的容量提供者設定 VPC 連線
- 了解 Lambda 受管執行個體的安全性和許可

Lambda 受管執行個體入門

建立 Lambda 受管執行個體函數 （主控台）

您可以使用 Lambda 主控台來建立在容量提供者管理的 Amazon EC2 執行個體上執行的受管執行個體函數。

重要：在建立受管執行個體函數之前，您必須先建立容量提供者。這些函數需要容量提供者來定義將執行函數的 Amazon EC2 基礎設施。

建立 Lambda 受管執行個體函數 （主控台）

1. 開啟 Lambda 主控台。
2. 從左側導覽窗格中選擇容量提供者。
3. 選擇建立容量提供者。
4. 在容量提供者設定區段中，輸入容量提供者的名稱。
5. 選取容量提供者的 VPC 和許可。您可以使用現有的 或建立新的 。如需建立必要運算子角色的資訊，請參閱 Lambda 受管執行個體的 Lambda 運算子角色。
6. 展開 Advanced settings (進階設定)。
7. 選擇處理器架構和執行個體類型來定義您的執行個體需求。
8. 在自動擴展下，指定容量提供者的 EC2 vCPUs數量上限。您也可以選擇手動執行個體擴展模式來設定自己的擴展值，以進行精確控制。
9. 選擇建立容量提供者以建立新的容量提供者。
10. 接下來，選擇建立函數。
11. 選取從頭開始撰寫。
12. 在基本資訊窗格中，提供函數名稱。

13針對執行期，選擇任何支援的執行期。

14選擇函數的架構（與您為容量提供者選取的架構相同）。根據預設，x86_64。

15在許可下，確保您具有所選執行角色的許可。否則，您可以建立新的角色。

16在其他組態下，選擇運算類型做為 Lambda 受管執行個體。

17您在先前步驟中建立的容量提供者 ARN 應預先選取。

18選擇每個 vCPU 比率的記憶體大小和執行環境記憶體 (GiB)。 GiB

19選擇 Create function (建立函數)。

您的 Lambda 受管執行個體函數已建立，並將在您指定的容量提供者上佈建容量。函數建立通常需要幾分鐘的時間。完成後，您可以編輯函數程式碼並執行第一個測試。

建立 Lambda 受管執行個體函數 (AWS CLI)

先決條件

開始前，請確定您具有下列項目：

- AWS CLI – 安裝和設定 AWS CLI。如需詳細資訊，請參閱安裝或更新最新版本的 AWS CLI。
- IAM 許可 – 您的 IAM 使用者或角色必須具有建立 Lambda 函數、容量提供者和傳遞 IAM 角色的許可。請注意，`iam:CreateServiceLinkedRole`如果您是第一次在帳戶中建立容量提供者，或是已刪除服務連結角色 (SLR)，您也需要 。

步驟 1：建立所需的 IAM 角色

Lambda 受管執行個體需要兩個 IAM 角色：函數的執行角色和容量提供者的運算子角色。運算子角色允許 Lambda 代表您啟動、終止和監控 Amazon EC2 執行個體。函數執行角色會授予函數存取其他 AWS 服務和資源的許可。

建立 Lambda 執行角色

1. 建立信任政策文件，允許 Lambda 擔任該角色：

```
cat > lambda-trust-policy.json << 'EOF'
{
  "Version": "2012-10-17",
  "Statement": [
    {
```

```
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF
```

2. 建立執行角色：

```
aws iam create-role \
  --role-name MyLambdaExecutionRole \
  --assume-role-policy-document file://lambda-trust-policy.json
```

3. 連接基本執行政策：

```
aws iam attach-role-policy \
  --role-name MyLambdaExecutionRole \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

建立容量提供者運算子角色

1. 建立信任政策文件，允許 Lambda 擔任運算子角色：

```
cat > operator-trust-policy.json << 'EOF'
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF
```

2. 建立運算子角色：

```
aws iam create-role \
  --role-name MyCapacityProviderOperatorRole \
  --assume-role-policy-document file://operator-trust-policy.json
```

3. 連接必要的 EC2 許可政策：

```
aws iam attach-role-policy \
  --role-name MyCapacityProviderOperatorRole \
  --policy-arn arn:aws:iam::aws:policy/AWSLambdaManagedEC2ResourceOperator
```

步驟 2：設定 VPC 資源

Lambda 受管執行個體會在您的 VPC 中執行，且需要子網路和安全群組。

建立 VPC 資源

1. 建立 VPC：

```
VPC_ID=$(aws ec2 create-vpc \
  --cidr-block 10.0.0.0/16 \
  --query 'Vpc.VpcId' \
  --output text)
```

2. 建立子網路：

```
SUBNET_ID=$(aws ec2 create-subnet \
  --vpc-id $VPC_ID \
  --cidr-block 10.0.1.0/24 \
  --query 'Subnet.SubnetId' \
  --output text)
```

3. 建立安全群組：

```
SECURITY_GROUP_ID=$(aws ec2 create-security-group \
  --group-name my-capacity-provider-sg \
  --description "Security group for Lambda Managed Instances" \
  --vpc-id $VPC_ID \
  --query 'GroupId' \
  --output text)
```

注意：您的 Lambda 受管執行個體函數需要 VPC 組態才能存取 VPC 外部的資源，以及將遙測資料傳輸到 CloudWatch Logs 和 X-Ray。如需組態詳細資訊，請參閱 Lambda 受管執行個體的聯網。

步驟 3：建立容量提供者

容量提供者會管理執行 Lambda 函數的 EC2 執行個體。

建立容量提供者

```
ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)

aws lambda create-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --vpc-config SubnetIds=[$SUBNET_ID],SecurityGroupIds=[$SECURITY_GROUP_ID] \
  --permissions-config CapacityProviderOperatorRoleArn=arn:aws:iam::${ACCOUNT_ID}:role/
MyCapacityProviderOperatorRole \
  --instance-requirements Architectures=[x86_64] \
  --capacity-provider-scaling-config MaxVCpuCount=30
```

此命令會使用下列組態建立容量提供者：

- VPC 組態 – 指定 EC2 執行個體的子網路和安全群組
- 許可 – 定義 Lambda 用來管理 EC2 執行個體的 IAM 角色
- 執行個體需求 – 指定 x86_64 架構
- 擴展組態 – 為容量提供者設定最多 30 vCPUs

步驟 4：使用內嵌程式碼建立 Lambda 函數

使用內嵌程式碼建立函數

1. 首先，建立簡單的 Python 函數並將其內嵌封裝：

```
# Create a temporary directory for the function code
mkdir -p /tmp/my-lambda-function
cd /tmp/my-lambda-function

# Create a simple Python handler
cat > lambda_function.py << 'EOF'
import json
```

```
def lambda_handler(event, context):
    return {
        'statusCode': 200,
        'body': json.dumps({
            'message': 'Hello from Lambda Managed Instances!',
            'event': event
        })
    }
EOF

# Create a ZIP file
zip function.zip lambda_function.py
```

2. 使用內嵌 ZIP 檔案建立 Lambda 函數：

```
ACCOUNT_ID=$(aws sts get-caller-identity --query Account --output text)
REGION=$(aws configure get region)

aws lambda create-function \
  --function-name my-managed-instance-function \
  --package-type Zip \
  --runtime python3.13 \
  --handler lambda_function.lambda_handler \
  --zip-file fileb:///tmp/my-lambda-function/function.zip \
  --role arn:aws:iam::${ACCOUNT_ID}:role/MyLambdaExecutionRole \
  --architectures x86_64 \
  --memory-size 2048 \
  --ephemeral-storage Size=512 \
  --capacity-provider-config
 LambdaManagedInstancesCapacityProviderConfig={CapacityProviderArn=arn:aws:lambda:
${REGION}:${ACCOUNT_ID}:capacity-provider:my-capacity-provider}
```

函數的建立方式如下：

- 執行時間 – Python 3.13
- 處理常式 – 中的 `lambda_handler`函數 `lambda_function.py`
- 記憶體 – 2048 MB
- 暫時性儲存 – 512 MB
- 容量提供者 – 連結至您建立的容量提供者

步驟 5：發佈函數版本

若要在 Lambda 受管執行個體上執行函數，您必須發佈版本。

發佈函數版本

```
aws lambda publish-version \
  --function-name my-managed-instance-function
```

此命令會發佈函數的第 1 版，並將其部署到容量提供者。

步驟 6：叫用您的 函數

發佈之後，您可以叫用 函數。

叫用您的 函數

```
aws lambda invoke \
  --function-name my-managed-instance-function:1 \
  --payload '{"name": "World"}' \
  response.json

# View the response
cat response.json
```

函數會在容量提供者管理的 EC2 執行個體上執行，並傳回回應。

清除

若要避免產生費用，請刪除您建立的資源：

1. 刪除 函數：

   ```
   aws lambda delete-function --function-name my-managed-instance-function
   ```

2. 刪除容量提供者：

   ```
   aws lambda delete-capacity-provider --capacity-provider-name my-capacity-provider
   ```

3. 刪除 VPC 資源：

   ```
   aws ec2 delete-security-group --group-id $SECURITY_GROUP_ID
   aws ec2 delete-subnet --subnet-id $SUBNET_ID
   ```

```
aws ec2 delete-vpc --vpc-id $VPC_ID
```

4. 刪除 IAM 角色：

```
aws iam detach-role-policy \
  --role-name MyLambdaExecutionRole \
  --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
aws iam detach-role-policy \
  --role-name MyCapacityProviderOperatorRole \
  --policy-arn arn:aws:iam::aws:policy/AWSLambdaManagedEC2ResourceOperator

aws iam delete-role --role-name MyLambdaExecutionRole
aws iam delete-role --role-name MyCapacityProviderOperatorRole
```

核心概念

Lambda 受管執行個體引進數個與傳統 Lambda 函數不同的核心概念。了解這些概念對於在 EC2 基礎設施上有效部署和管理函數至關重要。

容量提供者是 Lambda 受管執行個體的基礎。容量提供者會定義函數執行所在的運算基礎設施，包括 VPC 組態、執行個體需求和擴展政策。容量提供者也會做為函數的安全界限，這表示指派給相同容量提供者的所有函數都必須相互信任。

擴展行為與傳統 Lambda 函數明顯不同。受管執行個體不會在呼叫到達時隨需擴展，而是根據 CPU 資源使用率以非同步方式擴展。這種方法可消除冷啟動，但需要規劃流量成長。如果您的流量在 5 分鐘內增加一倍以上，您可能會遇到調節，因為 Lambda 會擴展容量以滿足需求。

安全性和許可需要仔細考慮。您需要操作員角色許可，才能允許 Lambda 在容量提供者中管理 EC2 資源。此外，使用者需要 `lambda:PassCapacityProvider`許可，才能將函數指派給容量提供者，做為安全閘道來控制哪些函數可在特定基礎設施上執行。

多並行執行是受管執行個體的主要特性。每個執行環境可以同時處理多個調用，最大限度地提高 IO 密集型應用程式的資源使用率。這與傳統 Lambda 不同，每個環境一次處理一個請求。此執行模型需要注意執行緒安全性、狀態管理和內容隔離，視您的執行時間而定。

以下各節提供有關每個核心概念的詳細資訊。

容量提供者

容量提供者是執行 Lambda 受管執行個體的基礎。它可做為函數的安全界限，並定義 Lambda 將代表您佈建和管理的運算資源。

當您建立容量提供者時，您可以指定：

- VPC 組態 - 執行個體將執行的子網路和安全群組
- 許可 - Lambda 管理 EC2 資源的 IAM 角色
- 執行個體需求 （選用） - 架構和執行個體類型偏好設定
- 擴展組態 （選用） - Lambda 如何擴展您的執行個體

了解容量提供者做為安全界限

容量提供者可做為 VPC 內 Lambda 函數的安全界限，取代 Firecracker 型隔離。函數在執行個體內的容器中執行，但與 Firecracker microVMs 不同，容器在函數之間不提供強大的安全隔離。

關鍵安全概念：

- 容量提供者：定義 Lambda 函數信任層級的安全界限
- 容器隔離：容器不是安全提供者 - 不要依賴它們來確保不受信任工作負載之間的安全
- 信任分離：使用不同的容量提供者來分離非相互信任的工作負載

建立容量提供者

您可以使用 AWS CLI、 AWS 管理主控台或 AWS SDKs建立容量提供者。

使用 AWS CLI：

```
aws lambda create-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --vpc-config
 SubnetIds=subnet-12345,subnet-67890,subnet-11111,SecurityGroupIds=sg-12345 \
  --permissions-config CapacityProviderOperatorRoleArn=arn:aws:iam::123456789012:role/
MyOperatorRole \
  --instance-requirements Architectures=x86_64 \
  --capacity-provider-scaling-config ScalingMode=Auto
```

必要參數

CapacityProviderName

- 容量提供者的唯一名稱

- 在您的 AWS 帳戶中必須是唯一的

VpcConfig

- SubnetIds（必要）：至少一個子網路，最多 16 個。跨多個可用區域使用子網路以實現彈性
- SecurityGroupIds（選用）：執行個體的安全群組。如果未指定，則預設為 VPC 預設安全群組

PermissionsConfig

- CapacityProviderOperatorRoleArn（必要）：允許 Lambda 管理容量提供者中 EC2 資源的 IAM 角色

選用的參數

InstanceRequirements

為您的容量提供者指定架構和執行個體類型：

- 架構：選擇 `x86_64`或 `arm64`。預設值為 `x86_64`
- AllowedInstanceTypes：指定允許的執行個體類型。範例：`m5.8xlarge`
- ExcludedInstanceTypes：使用萬用字元指定排除的執行個體類型。您只能指定其中一個 AllowedInstanceTypes 或 ExcludedInstanceTypes

根據預設，Lambda 會為您的工作負載選擇最佳執行個體類型。我們建議您讓 Lambda 受管執行個體為您選擇執行個體類型，因為限制可能的執行個體類型數量可能會導致可用性降低。

CapacityProviderScalingConfig

設定 Lambda 如何擴展您的執行個體：

- ScalingMode：將 設定為 `Auto` 以進行自動擴展或`Manual`手動控制。預設值為 `Auto`
- MaxVCpuCount：容量提供者的 vCPUs數目上限。預設為 400。
- ScalingPolicies：定義 CPU 和記憶體使用率的目標追蹤擴展政策

KmsKeyArn

指定 EBS 加密的 AWS KMS 金鑰。如果未指定，則預設為 AWS 受管金鑰。

Tags (標籤)

新增標籤以組織和管理容量提供者。

管理容量提供者

更新容量提供者

您可以使用 UpdateCapacityProvider API 更新容量提供者的特定屬性。

```
aws lambda update-capacity-provider \
  --capacity-provider-name my-capacity-provider \
  --capacity-provider-scaling-config ScalingMode=Auto
```

刪除容量提供者

您可以使用 DeleteCapacityProvider API 刪除不再需要的容量提供者。

```
aws lambda delete-capacity-provider \
  --capacity-provider-name my-capacity-provider
```

注意：您無法刪除已連接函數版本的容量提供者。

檢視容量提供者詳細資訊

使用 GetCapacityProvider API 擷取容量提供者的相關資訊。

```
aws lambda get-capacity-provider \
  --capacity-provider-name my-capacity-provider
```

容量提供者狀態

容量提供者可以處於下列其中一種狀態：

- 待定：正在建立容量提供者
- 作用中：容量提供者已就緒可供使用
- 失敗：容量提供者建立失敗
- 刪除：正在刪除容量提供者

配額

- 每個帳戶的容量提供者上限：1，000
- 每個容量提供者的函數版本上限：100（無法增加）

最佳實務

1. 依信任層級分隔：為具有不同安全需求的工作負載建立不同的容量提供者
2. 使用描述性名稱：命名容量提供者，以清楚指出其預期用途和信任層級（例如 `production-trusted`、`dev-sandbox`)
3. 使用多個可用區域：指定跨多個AZs子網路以獲得高可用性
4. 讓 Lambda 選擇執行個體類型：除非您有特定的硬體需求，否則允許 Lambda 為可用性選取最佳的執行個體類型
5. 監控用量：使用 AWS CloudTrail 監控容量提供者指派和存取模式

後續步驟

- 了解擴展 Lambda 受管執行個體
- 了解 Lambda 受管執行個體的安全性和許可
- 為您的容量提供者設定 VPC 連線
- 檢閱 Java、Node.js 和 Python 的執行時間特定指南

擴展 Lambda 受管執行個體

當呼叫到達時，Lambda 受管執行個體不會擴展，也不支援冷啟動。反之，它會使用資源消耗訊號以非同步方式擴展。受管執行個體目前會根據 CPU 資源使用率和多並行飽和度進行擴展。

主要差異：

- Lambda（預設）：在沒有免費執行環境來處理傳入調用時擴展（冷啟動）
- Lambda 受管執行個體：根據執行環境的 CPU 資源使用率和多並行飽和，以非同步方式擴展

如果您的流量在 5 分鐘內增加一倍以上，您可能會在 Lambda 擴展執行個體和執行環境以滿足需求時看到調節。

擴展生命週期

Lambda 受管執行個體使用分散式架構來管理擴展：

元件：

- 受管執行個體 - 在您提供的子網路中於您的帳戶中執行
- Router and Scaler - 路由呼叫和管理擴展的共用 Lambda 元件
- Lambda 代理程式 - 在每個受管執行個體上執行，以管理執行環境生命週期並監控資源消耗

運作方式：

1. 當您使用容量提供者發佈函數版本時，Lambda 會在您的帳戶中啟動受管執行個體。根據預設，它會針對 AZ 彈性啟動三個 ，並在標記函數版本 ACTIVE 之前啟動三個執行環境。
2. 每個受管執行個體都可以針對映射至相同容量提供者的多個函數執行環境。
3. 隨著流量流入您的應用程式，執行環境會耗用 資源。Lambda 代理程式會通知 Scaler，以決定是否擴展新的執行環境或受管執行個體。
4. 如果 Router 嘗試將呼叫傳送至資源消耗較高的執行環境，則該執行個體上的 Lambda 代理程式會通知它在另一個執行個體上重試。
5. 隨著流量減少，Lambda 代理程式會通知 Scaler，以決定縮減執行環境並縮減受管執行個體。

調整擴展行為

您可以透過四個控制項自訂受管執行個體的擴展行為：

函數層級控制項

1. 函數記憶體和 vCPUs

選擇函數的記憶體大小和 vCPU 配置。支援的最小函數大小為 2GB 和 1vCPU。

考量：

- 挑選支援函數多並行執行的記憶體和 vCPU 設定
- 您無法設定少於 1 個 vCPU 的函數，因為在受管執行個體上執行的函數應支援多並行工作負載
- 您無法選擇小於 2GB，因為這符合 c 執行個體的 2 比 1 記憶體與 vCPU 比率，其比率最低
- 對於 Python 應用程式，您可能需要選擇較高比率的記憶體與 vCPUs，例如 4 比 1 或 8 比 1，因為 Python 處理多並行的方式

- 如果您執行 CPU 密集型操作或執行少量 IO，您應該選擇多個 vCPU

2. 並行上限

設定每個執行環境的並行上限。

預設行為：Lambda 選擇合理的預設值，以平衡適用於各種應用程式的資源消耗和輸送量。

調整準則：

- 增加並行：如果您的函數調用使用極少的 CPU，您可以將並行上限提高到每個 vCPU 最多 64 個
- 減少並行：如果您的應用程式耗用大量記憶體和極少 CPU，您可以減少並行上限

重要：由於 Lambda 受管執行個體適用於多並行應用程式，因此在擴展時，並行非常低的執行環境可能會遇到調節。

容量提供者層級控制

3. 目標資源使用率

選擇您自己的 CPU 使用率消耗目標。

預設行為：Lambda 維持足夠的空間，讓您的流量在 5 分鐘內加倍，無需調節。

最佳化選項：

- 如果您的工作負載非常穩定，或者您的應用程式對調節不敏感，您可以將目標設定為高層級，以實現更高的使用率和更低的成本
- 如果您想要維持高載流量的標頭空間，您可以將資源目標設定為低階，這將需要更多容量

4. 執行個體類型選取

設定允許或排除的執行個體類型。

預設行為：Lambda 會為您的工作負載選擇最佳的執行個體類型。我們建議您讓 Lambda 受管執行個體為您選擇執行個體類型，因為限制可能的執行個體類型數量可能會導致可用性降低。

自訂組態：

- 特定硬體需求：將允許的執行個體類型設定為相容的執行個體清單。例如，如果您的應用程式需要高網路頻寬，您可以選取多個 n 個執行個體類型

- 成本最佳化：對於測試或開發環境，您可以選擇較小的執行個體類型，例如 m7a.large 執行個體類型

後續步驟

- 了解 Lambda 受管執行個體的容量提供者
- 檢閱處理多並行的執行時間特定指南
- 為您的容量提供者設定 VPC 連線能力
- 監控擴展指標以最佳化擴展行為

安全與許可

Lambda 受管執行個體使用容量提供者做為信任界限。函數會在這些執行個體內的容器中執行，但容器不提供工作負載之間的安全隔離。指派給相同容量提供者的所有函數都必須相互信任。

關鍵安全概念

- 容量提供者：定義 Lambda 函數信任層級的安全界限
- 容器隔離：容器不是安全界限 - 不要依賴它們來確保不受信任工作負載之間的安全
- 信任分離：使用不同的容量提供者來分隔不受互信的工作負載

所需的許可

PassCapacityProvider 動作

使用者需要 `lambda:PassCapacityProvider`許可，才能將函數指派給容量提供者。此許可充當安全閘道，確保只有授權使用者才能在特定容量提供者中放置函數。

帳戶管理員可透過 IAM `lambda:PassCapacityProvider` 動作控制哪些函數可以使用特定容量提供者。以下情況需要此動作：

- 建立使用 Lambda 受管執行個體的函數
- 更新函數組態以使用容量提供者
- 透過基礎設施將函數部署為程式碼

IAM 政策範例

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "lambda:PassCapacityProvider",
      "Resource": "arn:aws:lambda:*:*:capacity-provider:trusted-workloads-*"
    }
  ]
}
```

服務連結角色

AWS Lambda 使用AWSServiceRoleForLambda服務連結角色來管理容量提供者中的 Lambda 受管執行個體 ec2 資源。

最佳實務

1. 依信任層級分隔：為具有不同安全需求的工作負載建立不同的容量提供者
2. 使用描述性名稱：命名容量提供者，以清楚指出其預期用途和信任層級（例如 `production-trusted`、`dev-sandbox`)
3. 套用最低權限：僅授予必要容量提供者的PassCapacityProvider許可
4. 監控用量：使用 AWS CloudTrail 監控容量提供者指派和存取模式

後續步驟

- 了解 Lambda 受管執行個體的容量提供者
- 了解 Lambda 受管執行個體的擴展
- 為您的容量提供者設定 VPC 連線
- 檢閱 Java、Node.js 和 Python 的執行時間特定指南

Lambda 受管執行個體的 Lambda 運算子角色

當您使用 Lambda 受管執行個體時，Lambda 需要許可才能管理帳戶中的運算容量。運算子角色透過 IAM 政策提供這些許可，允許 Lambda 管理容量提供者中的 EC2 執行個體。

Lambda 在執行這些管理操作時擔任 運算子角色，類似於 Lambda 在函數執行時擔任 執行角色的方式。

建立運算子角色

您可以在 IAM 主控台或使用 CLI AWS 建立運算子角色。角色必須包含：

- 許可政策 – 授予管理容量提供者和相關資源的許可
- 信任政策 – 允許 Lambda 服務 (`lambda.amazonaws.com`) 擔任該角色

許可政策

運算子角色需要管理容量提供者和基礎運算資源的許可。該角色至少需要 AWSLambdaManagedEC2ResourceOperator 受管政策中的許可，目前為：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances",
        "ec2:CreateTags",
        "ec2:AttachNetworkInterface"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:instance/*",
        "arn:aws:ec2:*:*:network-interface/*",
        "arn:aws:ec2:*:*:volume/*"
      ],
      "Condition": {
        "StringEquals": {
          "ec2:ManagedResourceOperator": "scaler.lambda.amazonaws.com"
        }
      }
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:DescribeAvailabilityZones",
        "ec2:DescribeCapacityReservations",
        "ec2:DescribeInstances",
        "ec2:DescribeInstanceStatus",
        "ec2:DescribeInstanceTypeOfferings",
        "ec2:DescribeInstanceTypes",
```

```
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances",
        "ec2:CreateNetworkInterface"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:subnet/*",
        "arn:aws:ec2:*:*:security-group/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "ec2:RunInstances"
      ],
      "Resource": [
        "arn:aws:ec2:*:*:image/*"
      ],
      "Condition": {
        "StringEquals": {
          "ec2:Owner": "amazon"
        }
      }
    }
  ]
}
```

信任政策

信任政策允許 Lambda 擔任運算子角色：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
```

```
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Lambda 受管執行個體的服務連結角色

若要以負責的方式管理 Lambda 受管執行個體的生命週期，Lambda 需要持續存取才能終止您帳戶中的受管執行個體。Lambda 使用 AWS Identity and Access Management (IAM) 服務連結角色 (SLR) 來執行這些操作。

自動建立：服務連結角色會在您第一次建立容量提供者時自動建立。建立第一個容量提供者的使用者必須具有`lambda.amazonaws.com`委託人的`iam:CreateServiceLinkedRole`許可。

許可：服務連結角色授予 Lambda 受管執行個體的下列許可：

- `ec2:TerminateInstances` – 在執行個體生命週期結束時終止執行個體
- `ec2:DescribeInstances` – 列舉受管執行個體

刪除：只有在刪除帳戶中的所有 Lambda 受管執行個體容量提供者之後，才能刪除此服務連結角色。

如需服務連結角色的詳細資訊，請參閱the section called “使用服務連結角色”。

了解 Lambda 受管執行個體執行環境

Lambda 受管執行個體提供替代部署模型，可在客戶擁有的 Amazon EC2 執行個體上執行函數程式碼，同時 Lambda 管理操作層面。受管執行個體的執行環境與 Lambda （預設） 函數有幾項重要的差異，特別是在處理並行調用和管理容器生命週期方面。

注意：如需 Lambda （預設） 執行環境的相關資訊，請參閱了解 Lambda 執行環境生命週期。

執行環境生命週期

Lambda 受管執行個體函數執行環境的生命週期與 Lambda （預設） 有幾個關鍵方面不同：

初始化階段

在初始化階段，Lambda 會執行下列步驟：

- 初始化並註冊所有延伸模組

- 引導執行時間進入點。執行時間會產生設定的執行時間工作者數量（實作取決於執行時間）
- 執行函數初始化程式碼（處理常式外部的程式碼）
- 呼叫 ，等待至少一個執行階段工作者發出整備訊號 `/runtime/invocation/next`

當延伸項目已初始化且至少有一個執行時間工作者已呼叫 時，初始化階段會被視為完成`/runtime/invocation/next`。然後，該函數已準備好處理調用。

> **Note**
>
> 對於 Lambda 受管執行個體函數，初始化最多可能需要 15 分鐘。逾時是最多 130 秒或設定的函數逾時（最多 900 秒）。

調用階段

Lambda 受管執行個體函數的調用階段具有數種獨特特性：

持續操作。與 Lambda（預設）不同，執行環境會持續作用中，並在呼叫到達時處理呼叫，而不會在呼叫之間凍結。

平行處理。多個調用可以在相同的執行環境中同時執行，每個調用都由不同的執行時間工作者處理。

獨立逾時。函數設定的逾時會套用至每個個別叫用。當調用逾時時，Lambda 會將該特定調用標記為失敗，但不會中斷其他執行中的調用或終止執行環境。

背壓處理。如果所有執行期工作者都忙於處理調用，則會拒絕新的調用請求，直到工作者可用為止。

錯誤處理和復原

Lambda 受管執行個體函數執行環境中的錯誤處理與 Lambda 不同（預設）：

叫用逾時。當個別調用逾時時，Lambda 會傳回該調用的逾時錯誤。不過，Lambda 受管執行個體不會強制執行逾時，您的程式碼會繼續執行。身為函數開發人員，您必須負責偵測和處理逾時。內容物件會公開呼叫的剩餘時間，其中零或負值表示逾時。執行環境中的其他並行調用會繼續正常處理。

執行期工作者失敗。如果執行期工作者程序當機，則執行環境會繼續與其餘運作狀態良好的工作者一起運作。

延伸模組當機。如果延伸程序在初始化或操作期間當機，整個執行環境會標示為運作狀態不佳並終止。Lambda 會建立新的執行環境來取代它。

不重設/修復。與 Lambda （預設） 不同，受管執行個體不會嘗試在發生錯誤後重設並重新初始化執行環境。反之，運作狀態不佳的容器會終止，並以新的容器取代。

Lambda 受管執行個體中的 $LATEST.PUBLISHED 版本

Lambda 受管執行個體函數支援與 Lambda 相同的編號版本控制工作流程 （預設）。如果您不想維護編號版本，Lambda 受管執行個體會引入新的版本類型：$LATEST.PUBLISHED。此版本可讓您視需要使用更新的程式碼或組態建立或重新發佈最新發佈的版本，而無需管理編號版本。

與 $LATEST 的主要差異：當您使用不合格的 ARN 叫用 Lambda 受管執行個體函數時，Lambda 會隱含叫用 $LATEST.PUBLISHED版本，而不是未發佈的 $LATEST 版本。

下列 AWS CLI 命令會建立或重新發佈$LATEST.PUBLISHED版本。

```
aws lambda publish-version --function-name my-function --publish-to LATEST_PUBLISHED
```

您應該會看到下列輸出：

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:
$LATEST.PUBLISHED",
  "Version": "$LATEST.PUBLISHED",
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
  "Handler": "function.handler",
  "Runtime": "nodejs24.x",
  ...
}
```

> **Note**
>
> 如果您使用 AWS CloudFormation 或 Lambda 主控台來建立 Lambda 受管執行個體函數，Lambda 會自動建立$LATEST.PUBLISHED版本。

Lambda 受管執行個體執行時間

使用 Lambda 受管執行個體時，Lambda 會以不同的方式處理請求。Lambda 受管執行個體不會在每個執行環境中依序處理請求，而是在每個執行環境中同時處理多個請求。此執行模型變更表示使

用 Lambda 受管執行個體的函數需要考慮執行緒安全性、狀態管理和內容隔離，這些問題不會出現在 Lambda （預設） 單一並行模型中。此外，多並行實作會因執行時間而有所不同。

支援的語言

Lambda 受管執行個體可與下列程式設計語言和執行時間搭配使用：

- Java：Java 21 和更新版本。
- Python：Python 3.13 和更新版本。
- Node.js：Node.js 22 和更新版本。
- .NET：.NET 8 及更新版本。
- Rust：支援使用僅限作業系統的執行時間 `provided.al2023`和更新版本。

語言特定考量事項

每種程式設計語言都會以不同的方式實作多並行。您需要了解如何在您選擇的程式設計語言中實作多並行，以套用適當的並行最佳實務。

Java

使用單一程序搭配並行的作業系統執行緒。多個執行緒同時執行處理常式方法，需要安全處理狀態和共用資源的執行緒。

Python

使用多個 Python 程序，其中每個並行請求在個別程序中執行。這可避免大多數並行問題，但 `/tmp`目錄等共用資源需要小心。

Node.js

使用具有非同步執行的工作者執行緒。並行請求會分散到工作者執行緒，而且每個工作者執行緒也可以非同步處理並行請求，需要安全處理狀態和共用資源。

.NET

使用 .NET 任務來非同步處理多個並行請求。需要安全處理狀態和共用資源。

Rust

使用單一程序搭配由 Tokio 提供支援的非同步任務。處理常式必須是 `Clone + Send`。

後續步驟

如需每個執行時間的詳細資訊，請參閱下列主題：

- Lambda 受管執行個體的 Java 執行期
- Lambda 受管執行個體的 Node.js 執行時間
- Lambda 受管執行個體的 Python 執行期
- Lambda 受管執行個體的 .NET 執行時間
- Rust 支援 Lambda 受管執行個體

Lambda 受管執行個體的 Java 執行期

對於 Java 執行期，Lambda 受管執行個體會使用作業系統執行緒來並行。Lambda 會在初始化期間為每個執行環境載入您的處理常式物件一次，然後建立多個執行緒。這些執行緒會平行執行，並且需要安全處理狀態和共用資源的執行緒。每個執行緒共用相同的處理常式物件和任何靜態欄位。

並行組態

Lambda 傳送至每個執行環境的並行請求數目上限是由函數組態中的 `PerExecutionEnvironmentMaxConcurrency`設定所控制。這是選用設定，預設值會根據執行時間而有所不同。對於 Java 執行時間，預設值為每個 vCPU 32 個並行請求，或者您可以設定自己的值。此值也會決定 Java 執行時間使用的執行緒數目。Lambda 會根據每個執行環境的容量，自動調整並行請求的數量，直到設定的最大值為止，以吸收這些請求。

為多並行建置函數

使用 Lambda 受管執行個體時，您應該套用與任何其他多執行緒環境中相同的執行緒安全實務。由於處理常式物件會跨所有執行時間工作者執行緒共用，因此任何變動狀態都必須是執行緒安全。這包括集合、資料庫連線，以及在請求處理期間修改的任何靜態物件。

AWS SDK 用戶端是安全的執行緒，不需要特殊處理。

範例：資料庫連線集區

下列程式碼使用在執行緒之間共用的靜態資料庫連線物件。視使用的連線程式庫而定，這可能不安全執行緒。

```
public class DBQueryHandler implements RequestHandler<Object, String> {
    // Single connection shared across all threads - NOT SAFE
    private static Connection connection;

    public DBQueryHandler() {
        this.connection = DriverManager.getConnection(jdbcUrl, username, password);
    }

    @Override
    public String handleRequest(Object input, Context context) {
        PreparedStatement stmt = connection.prepareStatement(query);
        ResultSet rs = stmt.executeQuery();
        // Multiple threads using same connection causes issues
        return result.toString();
    }
}
```

安全執行緒的方法是使用連線集區。在下列範例中，函數處理常式會從集區擷取連線。連線僅用於單一請求的內容。

```
public class DBQueryHandler implements RequestHandler<Object, String> {

    private static HikariDataSource dataSource;

    public DBQueryHandler() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl("jdbc:mysql://localhost:3306/your_database");
        dataSource = new HikariDataSource(config); // Create pool once per Lambda
 container
    }

    @Override
    public String handleRequest(Object input, Context context) {
        String query = "SELECT column_name FROM your_table LIMIT 10";
        StringBuilder result = new StringBuilder("Data:\n");

        // try-with-resources automatically calls close() on the connection,
        // which returns it to the HikariCP pool (does NOT close the physical DB
 connection)
        try (Connection connection = dataSource.getConnection();
             PreparedStatement stmt = connection.prepareStatement(query);
             ResultSet rs = stmt.executeQuery()) {
```

```
            while (rs.next()) {
                result.append(rs.getString("column_name")).append("\n");
            }

        } catch (Exception e) {
            context.getLogger().log("Error: " + e.getMessage());
            return "Error";
        }

        return result.toString();
    }
}
```

範例：集合

標準 Java 集合不安全執行緒：

```
public class Handler implements RequestHandler<Object, String> {
    private static List<String> items = new ArrayList<>();
    private static Map<String, Object> cache = new HashMap<>();

    @Override
    public String handleRequest(Object input, Context context) {
        items.add("list item");  // Not thread-safe
        cache.put("key", input); // Not thread-safe
        return "Success";
    }
}
```

請改用安全執行緒集合：

```
public class Handler implements RequestHandler<Object, String> {
    private static final List<String> items =
        Collections.synchronizedList(new ArrayList<>());
    private static final ConcurrentHashMap<String, Object> cache =
        new ConcurrentHashMap<>();

    @Override
    public String handleRequest(Object input, Context context) {
        items.add("list item");  // Thread-safe
        cache.put("key", input); // Thread-safe
        return "Success";
    }
```

```
}
```

共用 /tmp 目錄

/tmp 目錄會跨執行環境中的所有並行請求共用。同時寫入相同檔案可能會導致資料損毀，例如，如果另一個程序覆寫檔案。若要解決此問題，請實作共用檔案的檔案鎖定，或針對每個執行緒或每個請求使用唯一的檔案名稱，以避免衝突。請記得清除不需要的檔案，以避免耗盡可用空間。

日誌

日誌交錯（來自在日誌中交錯之不同請求的日誌項目）在多並行系統中是正常的。

使用 Lambda 受管執行個體的函數一律使用進階記錄控制所引進的結構化 JSON 日誌格式。此格式包含 requestId，允許日誌項目與單一請求相關聯。當您使用來自 context.getLogger() 的LambdaLogger物件requestId時， 會自動包含在每個日誌項目中。如需詳細資訊，請參閱 the section called “搭配 Java 使用 Lambda 進階日誌控制項”。

請求內容

context 物件繫結至請求執行緒。使用 context.getAwsRequestId()可安全執行緒存取目前請求的請求 ID。

使用 context.getXrayTraceId() 存取 X-Ray 追蹤 ID。這可讓執行緒安全存取目前請求的追蹤 ID。Lambda 不支援具有 Lambda 受管執行個體_X_AMZN_TRACE_ID的環境變數。使用 AWS SDK 時，會自動傳播 X-Ray 追蹤 ID。

使用 com.amazonaws.services.lambda.runtime.Context.getRemainingTimeInMillis() 偵測逾時。如需詳細資訊，請參閱the section called “錯誤處理和復原”。

如果您在程式中使用虛擬執行緒或在初始化期間建立執行緒，您將需要將任何必要的請求內容傳遞至這些執行緒。

初始化和關閉

每個執行環境會發生一次函數初始化。在初始化期間建立的物件會跨執行緒共用。

對於具有擴充功能的 Lambda 函數，執行環境會在關閉期間發出 SIGTERM 訊號。延伸項目使用此訊號來觸發清除任務，例如排清緩衝區。您可以訂閱 SIGTERM 事件來觸發函數清除任務，例如關閉資料庫連線。若要詳細了解執行環境生命週期，請參閱 the section called “執行環境”。

相依性版本

Lambda 受管執行個體需要下列最低套件版本：

- AWS 適用於 Java 的 SDK 2.0：2.34.0 版或更新版本
- AWS 適用於 Java 的 X-Ray 開發套件：2.20.0 版或更新版本
- AWS Distro for OpenTelemetry - 適用於 Java 的檢測：2.20.0 版或更新版本
- 適用於 AWS Lambda (Java) 的 Powertools：2.8.0 版或更新版本

Lambda 的 Powertools AWS (Java)

Powertools for AWS Lambda (Java) 與 Lambda 受管執行個體相容，並提供用於記錄、追蹤、指標等的公用程式。如需詳細資訊，請參閱 Powertools for AWS Lambda (Java)。

後續步驟

- 檢閱 Lambda 受管執行個體的 Node.js 執行期
- 檢閱 Lambda 受管執行個體的 Python 執行時間
- 檢閱 Lambda 受管執行個體的 .NET 執行時間
- 了解擴展 Lambda 受管執行個體

Lambda 受管執行個體的 Node.js 執行時間

對於 Node.js 執行期，Lambda 受管執行個體使用具有 `async/ await`型執行的工作者執行緒來處理並行請求。每個工作者執行緒會發生一次函數初始化。並行調用由兩個維度處理：工作者執行緒提供跨 vCPUs平行處理，而非同步執行則在每個執行緒內提供並行處理。相同工作者執行緒處理的每個並行請求都會共用相同的處理常式物件和全域狀態，需要在多個並行請求下安全處理。

並行上限

Lambda 傳送至每個執行環境的並行請求數目上限是由函數組態中的 `PerExecutionEnvironmentMaxConcurrency`設定所控制。這是選用設定，預設值會根據執行時間而有所不同。對於 Node.js 執行時間，預設值為每個 vCPU 64 個並行請求，或者您可以設定自己的值。Lambda 會根據每個執行環境的容量，自動調整並行請求的數量，直到設定的最大值為止，以吸收這些請求。

對於 Node.js，每個執行環境可以處理的並行請求數量取決於工作者執行緒的數量，以及每個工作者執行緒的非同步處理並行請求的容量。預設的工作者執行緒數量取決於可用的 vCPUs 數量，或者您可以

透過設定AWS_LAMBDA_NODEJS_WORKER_COUNT環境變數來設定工作者執行緒數量。我們建議您使用非同步函數處理常式，因為這允許處理每個工作者執行緒的多個請求。如果您的函數處理常式是同步的，則每個工作者執行緒一次只能處理單一請求。

為多並行建置函數

使用非同步函數處理常式，每個執行階段工作者會同時處理多個請求。全域物件將在多個並行請求之間共用。對於可變物件，請避免使用全域狀態或使用 AsyncLocalStorage。

AWS SDK 用戶端是非同步安全的，不需要特殊處理。

範例：全域狀態

下列程式碼使用在函數處理常式內變動的全域物件。這不是非同步安全。

```
let state = {
    currentUser: null,
    requestData: null
};

export const handler = async (event, context) => {
    state.currentUser = event.userId;
    state.requestData = event.data;

    await processData(state.requestData);

    // state.currentUser might now belong to a different request
    return { user: state.currentUser };
};
```

在函數處理常式內初始化state物件可避免共用全域狀態。

```
export const handler = async (event, context) => {
    let state = {
        currentUser: event.userId,
        requestData: event.data
    };

    await processData(state.requestData);

    return { user: state.currentUser };
};
```

範例：資料庫連線

下列程式碼使用在多個調用之間共用的共用用戶端物件。視使用的連線程式庫而定，這可能不是並行安全。

```
const { Client } = require('pg');

// Single connection created at init time
const client = new Client({
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD
});

// Connect once during cold start
client.connect();

exports.handler = async (event) => {
  // Multiple parallel invocations share this single connection = BAD
  // With multi-concurrent Lambda, queries will collide
  const result = await client.query('SELECT * FROM users WHERE id = $1',
 [event.userId]);

  return {
    statusCode: 200,
    body: JSON.stringify(result.rows[0])
  };
};
```

並行安全方法是使用連線集區。集區會針對每個並行資料庫查詢使用個別的連線。

```
const { Pool } = require('pg');

// Connection pool created at init time
const pool = new Pool({
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  user: process.env.DB_USER,
  password: process.env.DB_PASSWORD,
  max: 20,  // Max connections in pool
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 2000
```

```
});

exports.handler = async (event) => {
  // Pool gives each parallel invocation its own connection
  const result = await pool.query('SELECT * FROM users WHERE id = $1', [event.userId]);

  return {
    statusCode: 200,
    body: JSON.stringify(result.rows[0])
  };
};
```

Node.js 22 回呼型處理常式

使用 Node.js 22 時，您無法搭配 Lambda 受管執行個體使用以回呼為基礎的函數處理常式。只有 Lambda （預設） 函數支援以回呼為基礎的處理常式。對於 Node.js 24 和更新版本的執行時間，Lambda （預設） 和 Lambda 受管執行個體的回呼型函數處理常式都已棄用。

反之，使用 Lambda 受管執行個體時，請使用 `async`函數處理常式。如需詳細資訊，請參閱在 Node.js 中定義 Lambda 函數處理常式。

共用 /tmp 目錄

`/tmp` 目錄會跨執行環境中的所有並行請求共用。同時寫入相同檔案可能會導致資料損毀，例如，如果另一個程序覆寫檔案。若要解決此問題，請實作共用檔案的檔案鎖定，或針對每個請求使用唯一的檔案名稱以避免衝突。請記得清除不需要的檔案，以避免耗盡可用空間。

日誌

日誌交錯 （來自在日誌中交錯之不同請求的日誌項目） 在多並行系統中是正常的。使用 Lambda 受管執行個體的函數一律使用進階記錄控制所引進的結構化 JSON 日誌格式。此格式包含 `requestId`，允許日誌項目與單一請求相關聯。當您使用`console`記錄器時， `requestId` 會自動包含在每個日誌項目中。如需詳細資訊，請參閱 the section called “搭配 Node.js 使用 Lambda 進階日誌控制項”。

熱門的第三方日誌程式庫，例如 Winston，通常包含使用 主控台進行日誌輸出的支援。

請求內容

使用 `context.awsRequestId` 可讓您以非同步安全的方式存取目前請求的請求 ID。

使用 `context.xRayTraceId` 存取 X-Ray 追蹤 ID。這可提供目前請求之追蹤 ID 的並行安全存取。Lambda 不支援具有 Lambda 受管執行個體_X_AMZN_TRACE_ID的環境變數。使用 AWS SDK 時，會自動傳播 X-Ray 追蹤 ID。

使用 `context.getRemainingTimeInMillis()` 偵測逾時。如需詳細資訊，請參閱the section called “錯誤處理和復原”。

初始化和關閉

每個工作者執行緒會發生一次函數初始化。如果您的函數在初始化期間發出日誌，您可能會看到重複的日誌項目。

對於具有擴充功能的 Lambda 函數，執行環境會在關閉期間發出 SIGTERM 訊號。延伸項目使用此訊號來觸發清除任務，例如排清緩衝區。具有擴充功能的 Lambda （預設） 函數也可以使用訂閱 SIGTERM 訊號`process.on()`。使用 Lambda 受管執行個體的函數不支援此功能，因為`process.on()`無法與工作者執行緒搭配使用。若要詳細了解執行環境生命週期，請參閱 the section called “執行環境”。

相依性版本

Lambda 受管執行個體需要下列最低套件版本：

- AWS 適用於 JavaScript 的 SDK v3：3.933.0 版或更新版本
- AWS 適用於 Node.js 的 X-Ray 開發套件：3.12.0 版或更新版本
- AWS Distro for OpenTelemetry - JavaScript 檢測：0.8.0 版或更新版本
- Powertools for AWS Lambda (TypeScript)：2.29.0 版或更新版本

Lambda 的 Powertools AWS (TypeScript)

Powertools for AWS Lambda (TypeScript) 與 Lambda 受管執行個體相容，並提供用於記錄、追蹤、指標等的公用程式。如需詳細資訊，請參閱 Powertools for AWS Lambda (TypeScript)。

後續步驟

- 檢閱 Lambda 受管執行個體的 Java 執行期
- 檢閱 Lambda 受管執行個體的 Python 執行時間
- 檢閱 Lambda 受管執行個體的 .NET 執行時間

- 了解擴展 Lambda 受管執行個體

Lambda 受管執行個體的 Python 執行期

Lambda 執行時間使用多個 Python 程序來處理並行請求。每個並行請求會在單獨的程序中執行，具有自己的記憶體空間和初始化。每個程序一次可同步處理一個請求。程序不會直接共用記憶體，因此全域變數、模組層級快取和單頓物件會在並行請求之間隔離。

並行組態

Lambda 傳送至每個執行環境的並行請求數目上限是由函數組態中的`PerExecutionEnvironmentMaxConcurrency`設定所控制。這是選用設定，預設值會根據執行時間而有所不同。對於 Python 執行時間，預設值為每個 vCPU 16 個並行請求，或者您可以設定自己的值。此值也會決定 Python 執行時間所使用的程序數目。Lambda 會根據每個執行環境的容量，自動調整並行請求的數量，直到設定的最大值為止，以吸收這些請求。

> **Important**
>
> 使用程序型並行表示每個執行階段工作者程序都會執行自己的初始化。總記憶體用量等於每個程序的記憶體乘以並行程序的數量。如果您要載入大型程式庫或資料集並具有高並行性，則記憶體使用量會很大。根據您的工作負載，您可能需要調整 CPU-to-memory的比率，或使用較低的並行設定，以避免超過可用的記憶體。您可以使用 CloudWatch 中的 `MemoryUtilization` 指標來追蹤記憶體耗用量。

為多並行建置函數

由於程序型多並行模型，使用 Python 執行時間的 Lambda 受管執行個體函數不會同時從多個調用存取記憶體內資源。您不需要為記憶體內並行安全性套用編碼實務。

共用 /tmp 目錄

`/tmp` 目錄會跨執行環境中的所有並行請求共用。同時寫入相同檔案可能會導致資料損毀，例如，如果另一個程序覆寫檔案。若要解決此問題，請實作共用檔案的檔案鎖定，或針對每個程序或每個請求使用唯一的檔案名稱，以避免衝突。請記得清除不需要的檔案，以避免耗盡可用空間。

日誌

日誌交錯（來自在日誌中交錯之不同請求的日誌項目）在多並行系統中是正常的。

使用 Lambda 受管執行個體的函數一律使用進階記錄控制所引進的結構化 JSON 日誌格式。此格式包含 `requestId`，允許日誌項目與單一請求相關聯。當您從 Lambda 中的 Python 標準程式庫使用`logging`模組時，`requestId` 會自動包含在每個日誌項目中。如需詳細資訊，請參閱搭配 Python 使用 Lambda 進階記錄控制。

請求內容

使用 `context.aws_request_id` 存取目前請求的請求 ID。

透過 Python 執行時間，您可以使用 `_X_AMZN_TRACE_ID`環境變數來存取具有 Lambda 受管執行個體的 X-Ray 追蹤 ID。使用 AWS SDK 時，會自動傳播 X-Ray 追蹤 ID。

使用 `context.get_remaining_time_in_millis()` 偵測逾時。如需詳細資訊，請參閱the section called “錯誤處理和復原”。

初始化和關閉

每個程序會初始化一次函數。如果您的函數在初始化期間發出日誌，您可能會看到重複的日誌項目。

對於具有擴充功能的 Lambda 函數，執行環境會在關閉期間發出 SIGTERM 訊號。延伸項目使用此訊號來觸發清除任務，例如排清緩衝區。您可以訂閱 SIGTERM 事件來觸發函數清除任務，例如關閉資料庫連線。若要詳細了解執行環境生命週期，請參閱 the section called “執行環境”。

相依性版本

Lambda 受管執行個體需要下列最低套件版本：

- 適用於 AWS Lambda (Python) 的 Powertools：3.23.0 版或更新版本

Lambda AWS 的 Powertools (Python)

Powertools for AWS Lambda (Python) 與 Lambda 受管執行個體相容，並提供用於記錄、追蹤、指標等的公用程式。如需詳細資訊，請參閱 Powertools for AWS Lambda (Python)。

後續步驟

- 檢閱 Lambda 受管執行個體的 Java 執行時間
- 檢閱 Lambda 受管執行個體的 Node.js 執行時間
- 檢閱 Lambda 受管執行個體的 .NET 執行時間

- 了解擴展 Lambda 受管執行個體

Lambda 受管執行個體的 .NET 執行時間

對於 .NET 執行時間，Lambda 受管執行個體在每個執行環境中使用單一 .NET 程序。使用 .NET 任務處理多個並行請求。

並行組態

Lambda 傳送至每個執行環境的並行請求數目上限是由函數組態中的 `PerExecutionEnvironmentMaxConcurrency`設定所控制。這是選用設定，預設值會根據執行時間而有所不同。對於 .NET 執行時間，預設值為每個 vCPU 32 個並行請求，或者您可以設定自己的值。Lambda 會根據每個執行環境的容量，自動調整並行請求的數量，直到設定的最大值為止，以吸收這些請求。

為多並行建置函數

使用 Lambda 受管執行個體時，您應該套用與任何其他多並行環境中相同的並行安全實務。由於處理常式物件會跨所有任務共用，因此任何變動狀態都必須是執行緒安全。這包括集合、資料庫連線，以及在請求處理期間修改的任何靜態物件。

AWS SDK 用戶端可安全執行緒，不需要特殊處理。

範例：資料庫連線集區

下列程式碼使用在並行請求之間共用的靜態資料庫連線物件。`SqlConnection` 物件執行緒不安全。

```
public class DBQueryHandler
{
    // Single connection shared across threads - NOT SAFE
    private SqlConnection connection;

    public DBQueryHandler()
    {
        connection = new SqlConnection("your-connection-string-here");
        connection.Open();
    }

    public string Handle(object input, ILambdaContext context)
    {
```

```
        using var cmd = connection.CreateCommand();
        cmd.CommandText = "SELECT ..."; // your query

        using var reader = cmd.ExecuteReader();

        ...
    }
}
```

若要解決此問題，請針對從連線集區繪製的每個請求使用個別的連線。ADO.NET 等供應商會在連線物件開啟時`Microsoft.Data.SqlClient`自動支援連線集區。

```
public class DBQueryHandler
{
    public DBQueryHandler()
    {
    }

    public string Handle(object input, ILambdaContext context)
    {
        using var connection = new SqlConnection("your-connection-string-here");
        connection.Open();
        using var cmd = connection.CreateCommand();
        cmd.CommandText = "SELECT ..."; // your query

        using var reader = cmd.ExecuteReader();

        ...
    }
}
```

範例：集合

標準 .NET 集合不安全執行緒：

```
public class Handler
{
    private static List<string> items = new List<string>();
    private static Dictionary<string, object> cache = new Dictionary<string, object>();

    public string FunctionHandler(object input, ILambdaContext context)
    {
```

```
        items.Add(context.AwsRequestId);
        cache["key"] = input;

        return "Success";
    }
}
```

使用 System.Collections.Concurrent命名空間的集合以確保並行安全：

```
public class Handler
{
    private static ConcurrentBag<string> items = new ConcurrentBag<string>();
    private static ConcurrentDictionary<string, object> cache = new
 ConcurrentDictionary<string, object>();

    public string FunctionHandler(object input, ILambdaContext context)
    {
        items.Add(context.AwsRequestId);
        cache["key"] = input;

        return "Success";
    }
}
```

共用 /tmp 目錄

/tmp 目錄會跨執行環境中的所有並行請求共用。對相同檔案的並行寫入可能會導致資料損毀，例如，如果另一個請求覆寫檔案。若要解決此問題，請實作共用檔案的檔案鎖定，或針對每個請求使用唯一的檔案名稱以避免衝突。請記得清除不需要的檔案，以避免耗盡可用空間。

日誌

日誌交錯（來自在日誌中交錯之不同請求的日誌項目）在多並行系統中是正常的。使用 Lambda 受管執行個體的函數一律使用進階記錄控制所引進的結構化 JSON 日誌格式。此格式包含 requestId，允許日誌項目與單一請求相關聯。當您使用 context.Logger 物件產生日誌時， requestId 會自動包含在每個日誌項目中。如需詳細資訊，請參閱 the section called “搭配 .NET 使用 Lambda 進階日誌控制項”。

請求內容

使用 context.AwsRequestId 屬性來存取目前請求的請求 ID。

使用 `context.TraceId` 屬性存取 X-Ray 追蹤 ID。這可提供目前請求之追蹤 ID 的並行安全存取。Lambda 不支援具有 Lambda 受管執行個體`_X_AMZN_TRACE_ID`的環境變數。使用 AWS SDK 時，會自動傳播 X-Ray 追蹤 ID。

使用 `ILambdaContext.RemainingTime` 偵測逾時。如需詳細資訊，請參閱the section called “錯誤處理和復原”。

初始化和關閉

每個執行環境都會初始化一次函數。在初始化期間建立的物件會跨請求共用。

對於具有擴充功能的 Lambda 函數，執行環境會在關閉期間發出 SIGTERM 訊號。延伸項目使用此訊號來觸發清除任務，例如排清緩衝區。您可以訂閱 SIGTERM 事件來觸發函數清除任務，例如關閉資料庫連線。若要詳細了解執行環境生命週期，請參閱 the section called “執行環境”。

相依性版本

Lambda 受管執行個體需要下列最低套件版本：

- Amazon.Lambda.Core：2.7.1 版或更新版本
- Amazon.Lambda.RuntimeSupport：1.14.1 版或更新版本
- OpenTelemetry.Instrumentation.AWSLambda:1.14.0 版或更新版本
- AWSXRayRecorder.Core:2.16.0 版或更新版本
- AWSSDK.Core：4.0.0.32 版或更新版本

適用於 AWS Lambda 的 Powertools (.NET)

Powertools for AWS Lambda (.NET) 和 AWS Distro for OpenTelemetry - Instrumentation for DotNet 目前不支援 Lambda 受管執行個體。

後續步驟

- 檢閱 Lambda 受管執行個體的 Java 執行時間
- 檢閱 Lambda 受管執行個體的 Node.js 執行時間
- 檢閱 Lambda 受管執行個體的 Python 執行時間
- 了解擴展 Lambda 受管執行個體

Rust 支援 Lambda 受管執行個體

並行組態

Lambda 傳送至每個執行環境的並行請求數目上限是由函數組態中的 `PerExecutionEnvironmentMaxConcurrency`設定所控制。這是選用設定，而 Rust 的預設值為每個 vCPU 8 個並行請求，或者您可以設定自己的值。此值會決定執行時間產生的 Tokio 任務數量，並在執行環境的生命週期內為靜態。每個工作者一次只處理一個傳輸中請求，每個工作者都沒有多工。Lambda 會根據每個執行環境的容量，自動調整並行請求的數量，直到設定的最大值為止，以吸收這些請求。

為多並行建置函數

使用 Lambda 受管執行個體時，您應該套用與任何其他多執行緒環境中相同的執行緒安全實務。由於處理常式物件會跨所有工作者執行緒共用，因此任何變動狀態都必須是執行緒安全。這包括集合、資料庫連線，以及在請求處理期間修改的任何靜態物件。

若要啟用並行請求處理，請將`concurrency-tokio`功能旗標新增至 `Cargo.toml` 檔案。

```
[dependencies]
lambda_runtime = { version = "1", features = ["concurrency-tokio"] }
```

`lambda_runtime::run_concurrent(…)` 進入點必須在 Tokio 執行時間內從 呼叫，通常由主要函數上的 `#[tokio::main]` 屬性提供。您的處理常式關閉必須實作 `Clone` + `Send`。這可讓架構安全地跨多個非同步任務共用您的處理常式。如果不符合這些邊界，您的程式碼將無法編譯。

當您需要跨調用 （資料庫集區、組態結構） 的共用狀態時，請將它包裝在 中，`Arc`並將 `Arc` 複製到每個調用中。

所有適用於 Rust 的 AWS SDK 用戶端都具有並行安全性，不需要特殊處理。

範例： AWS SDK 用戶端

下列範例使用 S3 用戶端在每次叫用時上傳物件。在沒有 的情況下，用戶端會直接複製到關閉中`Arc`：

```
let config = aws_config::load_defaults(BehaviorVersion::latest()).await;
let s3_client = aws_sdk_s3::Client::new(&config);

run_concurrent(service_fn(move |event: LambdaEvent<Request>| {
```

```
    let s3_client = s3_client.clone(); // cheap clone, no Arc needed
    async move {
        s3_client.put_object()
            .bucket(&event.payload.bucket)
            .key(&event.payload.key)
            .body(event.payload.body.into_bytes().into())
            .send()
            .await?;
        Ok(Response { message: "uploaded".into() })
    }
}))
.await
```

範例：資料庫連線集區

當您的處理常式需要存取共用狀態，例如用戶端和組態時，請將它包裝在 中，Arc並將 Arc 複製到每個調用中：

```
#[derive(Debug)]
struct AppState {
    dynamodb_client: DynamoDbClient,
    table_name: String,
    cache_ttl: Duration,
}

let config = aws_config::load_defaults(BehaviorVersion::latest()).await;
let state = Arc::new(AppState {
    dynamodb_client: DynamoDbClient::new(&config),
    table_name: std::env::var("TABLE_NAME").expect("TABLE_NAME must be set"),
    cache_ttl: Duration::from_secs(300),
});

run_concurrent(service_fn(move |event: LambdaEvent<Request>| {
    let state = state.clone();
    async move { handle(event, state).await }
}))
.await
```

共用 /tmp 目錄

/tmp 目錄會在相同執行環境中的所有並行調用之間共用。每次叫用使用唯一的檔案名稱 （例如包含請求 ID) 或實作明確檔案鎖定以避免資料損毀。

日誌

日誌交錯（來自在日誌中交錯之不同請求的日誌項目）在多並行系統中是正常的。使用 Lambda 受管執行個體的函數透過 Lambda 的進階記錄控制支援結構化 JSON 日誌格式。此格式包含 `requestId`，允許日誌項目與單一請求相關聯。如需詳細資訊，請參閱 the section called “實作帶有追蹤套件的進階日誌記錄”。

請求內容

`Context` 物件會直接傳遞至每個處理常式調用。使用 `event.context.request_id` 存取目前請求的請求 ID。

使用 `event.context.xray_trace_id` 存取 X-Ray 追蹤 ID。Lambda 不支援具有 Lambda 受管執行個體_X_AMZN_TRACE_ID的環境變數。使用適用於 Rust 的 AWS SDK 時，會自動傳播 X-Ray 追蹤 ID。

使用 `event.context.deadline` 來偵測逾時 — 它包含以毫秒為單位的調用截止日期。

初始化和關閉

每個執行環境會發生一次函數初始化。在初始化期間建立的物件會跨請求共用。

對於具有擴充功能的 Lambda 函數，執行環境會在關閉期間發出 SIGTERM 訊號。延伸項目使用此訊號來觸發清除任務，例如排清緩衝區。 `lambda_runtime`提供協助程式，以簡化設定正常的關閉訊號處理`spawn_graceful_shutdown_handler()`。若要詳細了解執行環境生命週期，請參閱 the section called “執行環境”。

相依性版本

Lambda 受管執行個體需要下列最低套件版本：

- `lambda_runtime`：1.1.1 版或更新版本，並啟用 `concurrency-tokio`功能
- 支援的 Rust 版本 (MSRV) 下限為 1.84.0。

Lambda 受管執行個體的網路

執行 Lambda 受管執行個體函數時，您需要設定網路連線，讓函數能夠存取 VPC 外部的資源。這包括 Amazon S3 和 DynamoDB 等 AWS 服務。傳輸遙測資料到 CloudWatch Logs 和 X-Ray 也需要連線。

連線選項

設定 VPC 連線有三種主要方法，每個方法在成本、安全性和複雜性方面都有不同的權衡。

具有網際網路閘道的公有子網路

此選項使用透過網際網路閘道直接存取網際網路的公有子網路。您可以選擇 IPv4 和 IPv6 組態。

IPv4 搭配網際網路閘道

使用網際網路閘道設定 IPv4 連線

1. 建立或使用具有 IPv4 CIDR 區塊的現有公有子網路。
2. 將網際網路閘道連接至 VPC。
3. 更新路由表，將`0.0.0.0/0`流量路由到網際網路閘道。
4. 確保資源已指派公有 IPv4 地址或彈性 IP 地址。
5. 設定安全群組以允許所需連接埠上的傳出流量。

此組態提供雙向連線，允許來自函數的傳出連線和來自網際網路的傳入連線。

IPv6 搭配網際網路閘道

使用網際網路閘道設定 IPv6 連線

1. 在 VPC 上啟用 IPv6。
2. 建立或使用已指派 IPv6 CIDR 區塊的現有公有子網路。
3. 將網際網路閘道連接到您的 VPC （相同的網際網路閘道可以同時處理 IPv4 和 IPv6)。
4. 更新路由表，將`::/0`流量路由到網際網路閘道。
5. 驗證您需要存取 AWS 的服務是否支援您區域中的 IPv6。
6. 設定安全群組以允許所需連接埠上的傳出流量。

此組態使用 IPv6 定址提供雙向連線。

IPv6 搭配輸出限定網際網路閘道

使用輸出限定網際網路閘道設定 IPv6 連線

1. 在 VPC 上啟用 IPv6。
2. 建立或使用已指派 IPv6 CIDR 區塊的現有公有子網路。
3. 將輸出限定網際網路閘道連接至 VPC。
4. 更新路由表，將`::/0`流量路由到僅限輸出的網際網路閘道。
5. 驗證您需要存取 AWS 的服務是否支援您區域中的 IPv6。
6. 設定安全群組以允許所需連接埠上的傳出流量。

此組態提供僅限傳出連線，防止來自網際網路的傳入連線，同時允許函數啟動傳出連線。

VPC 端點

VPC 端點可讓您將 VPC 私下連線至支援的 AWS 服務，而不需要網際網路閘道、NAT 裝置、VPN 連線或 AWS Direct Connect 連線。VPC 與 AWS 服務之間的流量不會離開 Amazon 網路。

設定 VPC 端點

1. 開啟位於 https：//console.aws.amazon.com/vpc/ 的 Amazon VPC 主控台。
2. 在導覽窗格中選擇端點。
3. 選擇建立端點。
4. 在 Service category (服務類別) 中，選擇 AWS services。
5. 針對服務名稱，選取您需要的服務端點 （ 例如，`com.amazonaws.region.s3`針對 Amazon S3)。
6. 針對 VPC，選取您的 VPC。
7. 針對子網路，選取您要建立端點網路介面的子網路。如需高可用性，請選取多個可用區域中的子網路。
8. 對於 Security group (安全群組)，選取要與端點網路介面建立關聯的安全群組。安全群組必須允許所需連接埠上來自函數安全群組的傳入流量。
9. 選擇建立端點。

針對函數需要存取的每個 AWS 服務重複這些步驟。

具有 NAT 閘道的私有子網路

此選項使用 NAT 閘道為私有子網路中的資源提供網際網路存取，同時保持資源的私密性。

使用 NAT 閘道設定私有子網路

1. 使用 CIDR 區塊建立公有子網路（如果尚未存在）。
2. 將網際網路閘道連接至 VPC。
3. 在公有子網路中建立 NAT 閘道並指派彈性 IP 地址。
4. 更新公有子網路路由表以新增路由：→ `0.0.0.0/0` 網際網路閘道。
5. 建立或使用具有 CIDR 區塊的現有私有子網路。
6. 更新私有子網路路由表以新增路由：→ NAT `0.0.0.0/0` 閘道。
7. 設定安全群組以允許所需連接埠上的傳出流量。

若要取得高可用性，請在每個可用區域中部署一個 NAT 閘道，並將每個可用區域的路由表設定為使用本機 NAT 閘道。這可避免跨可用區域資料傳輸費用，並改善彈性。

選擇連線選項

選擇連線選項時，請考慮下列因素：

具有網際網路閘道的公有子網路

- 成本最低的最簡單組態
- 適合開發和測試環境
- 資源可以從網際網路接收傳入連線（安全考量）
- 同時支援 IPv4 和 IPv6

VPC 端點

- 最高安全性，流量會保留在 AWS 網路內
- 與網際網路路由相比，延遲更低
- 建議用於具有嚴格安全需求的生產環境
- 每個端點、每個可用區域和每個處理 GB 的成本較高
- 需要每個可用區域中的端點以獲得高可用性

具有 NAT 閘道的私有子網路

- 資源保持私有，沒有傳入網際網路存取

- 標準企業架構模式
- 支援所有 IPv4 網際網路流量
- NAT 閘道每小時和資料處理費用的成本適中
- 僅支援 IPv4

後續步驟

- 了解 Lambda 受管執行個體的容量提供者
- 了解 Lambda 受管執行個體的擴展
- 檢閱 Java、Node.js 和 Python 的執行時間特定指南
- 了解 Lambda 受管執行個體的安全性和許可

監控 Lambda 受管執行個體

您可以使用 CloudWatch 指標監控 Lambda 受管執行個體。Lambda 會自動將指標發佈至 CloudWatch，以協助您監控資源使用率、追蹤成本和最佳化效能。

可用的指標

Lambda 受管執行個體提供兩個層級的指標：容量提供者層級和執行環境層級。

容量提供者層級指標

容量提供者層級指標可讓您了解執行個體的整體資源使用率。這些指標使用以下維度：

- CapacityProviderName - 容量提供者的名稱
- InstanceType - EC2 執行個體類型

資源使用率指標：

- CPUUtilization - 容量提供者中跨執行個體的 CPU 使用率百分比
- MemoryUtilization - 容量提供者中跨執行個體的記憶體使用率百分比

容量指標：

- vCPUAvailable - 執行個體上可用於配置的 vCPU 數量（計數）
- MemoryAvailable - 執行個體上可用於配置的記憶體數量（以位元組為單位）
- vCPUAllocated - 在執行個體上為執行環境配置的 vCPU 數量（計數）
- MemoryAllocated - 在執行個體上為執行環境配置的記憶體量（以位元組為單位）

執行環境層級指標

執行環境層級指標可讓您了解個別函數的資源使用率和並行。這些指標使用以下維度：

- CapacityProviderName - 容量提供者的名稱
- FunctionName - Lambda 函數的名稱
- 資源 - 依資源，檢視特定版本函數的指標。

> **Note**
>
> 對於 Lambda 受管執行個體 (LMI)，`Resource`維度僅支援函數版本。格式是 `<FunctionName>:<FunctionVersion>`。

可用的執行環境指標：

- ExecutionEnvironmentConcurrency - 5 分鐘取樣期間的並行上限
- ExecutionEnvironmentConcurrencyLimit - 每個執行環境的最大並行限制
- ExecutionEnvironmentCPUUtilization - 函數執行環境的 CPU 使用率百分比
- ExecutionEnvironmentMemoryUtilization - 函數執行環境的記憶體使用率百分比

指標頻率和保留

Lambda 受管執行個體指標會以 5 分鐘為間隔發佈，並保留 15 個月。

在 CloudWatch 中檢視指標

在 CloudWatch 主控台中檢視 Lambda 受管執行個體指標

1. 在 https：//console.aws.amazon.com/cloudwatch/ 開啟 CloudWatch 主控台。

2. 在導覽窗格中，選擇 指標。
3. 在所有指標索引標籤中，選擇 AWS/Lambda。
4. 選擇您要檢視的指標維度：
   - 對於容量提供者層級指標，依 CapacityProviderName 和 InstanceType 篩選
   - 對於執行環境層級指標，依 CapacityProviderName、FunctionName 和資源篩選
5. 選取您要監控的指標。

使用指標來最佳化效能

監控 CPU 和記憶體使用率，以了解您的函數大小是否正確。高使用率可能表示需要較大的執行個體類型或增加函數記憶體配置。追蹤並行指標以了解擴展行為並識別潛在的限流。

監控容量指標，以確認工作負載有足夠的資源可用。vCPUAvailable 和 MemoryAvailable 指標可協助您了解執行個體上的剩餘容量。

後續步驟

- 了解擴展 Lambda 受管執行個體
- 檢閱 Java、Node.js 和 Python 的執行時間特定指南
- 為您的容量提供者設定 VPC 連線
- 了解 Lambda 受管執行個體的安全性和許可

Lambda 受管執行個體配額

此頁面說明 AWS Lambda 受管執行個體的服務配額。這些配額與 AWS Lambda （預設） 配額不同。某些配額可應要求增加。

Lambda API 請求配額

這些配額控制您可以進行 API 呼叫以管理 Lambda 受管執行個體容量提供者的速率。讀取和寫入 API 速率限制適用於所有容量提供者操作，包括建立、更新、描述和刪除容量提供者。

資源	配額
所有容量提供者讀取 APIs 的最大合併速率 （每秒請求數）	每秒 15 個請求。無法提高。

資源	配額
所有容量提供者寫入 APIs 的最大合併速率（每秒請求數）	每秒 1 個請求。無法提高。

Lambda 受管執行個體資源配額

這些配額定義您 AWS 帳戶中核心 Lambda 受管執行個體資源的限制。它們管理您可以建立的容量提供者數量，以及可以與每個容量提供者相關聯的函數版本數量。

資源	配額
容量提供者	1,000 個。在帳戶中建立的容量提供者數量上限。
每個容量提供者的函數版本	100 每個容量提供者的函數版本數量上限。無法提高。

事件來源映射配額

這些配額控制處理 Lambda 受管執行個體上各種 AWS 服務之事件的輸送量和組態限制。輸送量限制可確保可預測的效能，而映射計數限制有助於維持服務穩定性。Lambda 受管執行個體上的事件來源映射支援 Amazon SQS、DynamoDB Streams、Amazon Kinesis Data Streams、Amazon MSK 和自我管理 Apache Kafka 作為事件來源。

資源	配額
Lambda 受管執行個體上的標準 SQS 事件來源映射輸送量	每秒 5 MB。無法提高。
Lambda 受管執行個體上的標準 Kafka 事件來源映射輸送量	每秒 1 MB。無法提高。
Lambda 受管執行個體上的標準 Kafka 事件來源映射	100 個事件來源映射。無法提高。
Lambda 受管執行個體上的 Kinesis 事件來源映射輸送量	每秒 25 MB。可以增加。
Lambda 受管執行個體上的 DynamoDB 事件來源映射輸送量	每秒 10 MB。可以增加。

資源	配額
在 Lambda 受管執行個體上調用非同步調用的請求輸送量	每秒 5 MB。可以增加。

請求提高配額

對於可以提高的配額，您可以透過 Service Quotas 主控台請求提高配額。

請求提高配額

1. 在 https：//console.aws.amazon.com/servicequotas/ 開啟 Service Quotas 主控台。
2. 在導覽窗格中，選擇 AWS services (AWS 服務)。
3. 選擇 AWS Lambda。
4. 選取您要增加的配額。
5. 選擇 Request quota increase (請求增加配額)。
6. 輸入新的配額值，並提供增加的理由。
7. 選擇請求。

後續步驟

- 了解 Lambda 受管執行個體的容量提供者
- 了解 Lambda 受管執行個體的擴展
- 檢閱 Java、Node.js 和 Python 的執行時間特定指南
- 為您的容量提供者設定 VPC 連線

Lambda 受管執行個體的最佳實務

容量提供者組態

依信任層級分隔容量提供者。為具有不同安全需求的工作負載建立不同的容量提供者。指派給相同容量提供者的所有函數都必須相互信任，因為容量提供者可做為安全界限。

使用描述性名稱。命名容量提供者，以清楚指出其預期用途和信任層級（例如 `production-trusted`、)`dev-sandbox`。這有助於團隊了解每個容量提供者的目的和安全性狀態。

使用多個可用區域。在建立容量提供者時，指定跨多個可用區域的子網路。Lambda 預設會針對 AZ 彈性啟動三個執行個體，以確保函數的高可用性。

執行個體類型選取

讓 Lambda 選擇執行個體類型。根據預設，Lambda 會為您的工作負載選擇最佳的執行個體類型。我們建議您讓 Lambda 受管執行個體為您選擇執行個體類型，因為限制可能的執行個體類型數量可能會導致可用性降低。

指定特定需求的執行個體類型。如果您有特定的硬體需求，請將允許的執行個體類型設定為相容的執行個體清單。例如：

- 對於需要高網路頻寬的應用程式，請選取多個 n 個執行個體類型
- 針對具有成本限制的測試或開發環境，請選擇較小的執行個體類型，例如 m7a.large

函數組態

選擇適當的記憶體和 vCPU 設定。選取支援函數多並行執行的記憶體和 vCPU 組態。支援的函數大小下限為 2GB 和 1 個 vCPU。

- 對於 Python 應用程式，選擇較高比率的記憶體與 vCPUs（例如 4 比 1 或 8 比 1），因為 Python 處理多並行的方式
- 對於執行少量 IO 的 CPU 密集型操作或函數，請選擇多個 vCPU
- 對於 IO 密集型應用程式，例如 Web 服務或批次任務，多並行可提供最大效益

適當設定並行上限。Lambda 會針對平衡資源消耗和輸送量的最大並行選擇合理的預設值。根據函數的資源用量調整此設定：

- 如果您的函數調用使用極少的 CPU，請增加並行上限（每個 vCPU 最多 64 個）
- 如果您的應用程式使用大量記憶體和極少 CPU，請減少並行上限

請注意，並行極低的執行環境可能會遇到調節和擴展困難。

擴展組態

設定適當的目標資源使用率。根據預設，Lambda 會維持足夠的空間，讓您的流量在 5 分鐘內加倍，無需調節。根據您的工作負載特性進行調整：

- 對於對調節不敏感的非常穩定工作負載或應用程式，請將目標設定為高層級，以實現更高的使用率和更低的成本
- 對於具有潛在流量暴增的工作負載，請將資源目標設定為低層級，以維持額外的空間

規劃流量成長。如果您的流量在 5 分鐘內增加一倍以上，您可能會在 Lambda 擴展執行個體和執行環境時看到調節。設計您的應用程式，以在快速擴展期間處理潛在的限流。

安全

套用 PassCapacityProvider 許可的最低權限。僅授予必要容量提供者的`lambda:PassCapacityProvider`許可。使用資源層級許可來限制使用者可以指派給函數的容量提供者。

監控容量提供者用量。Use AWS CloudTrail 可監控容量提供者指派和存取模式。這有助於識別未經授權的存取嘗試，並確保符合安全政策。

分隔不受信任的工作負載。請勿在不受信任的工作負載之間依賴容器進行安全隔離。使用不同的容量提供者來分隔不受互信的工作負載。

成本最佳化

利用 EC2 定價選項。利用 EC2 Savings Plans 和預留執行個體來降低成本。這些定價選項適用於基礎 EC2 運算 (15% 的管理費不會折扣）。

針對穩定狀態工作負載進行最佳化。Lambda 受管執行個體最適合具有可預測大量流量的穩定狀態函數。對於高載流量模式，Lambda （預設） 可能更具成本效益。

監控資源使用率。追蹤 CloudWatch 指標以了解 CPU 和記憶體使用率。根據實際用量模式調整函數記憶體配置和執行個體類型選擇，以最佳化成本。

監控與可觀測性

監控容量提供者指標。追蹤容量提供者層級指標，包括 CPUUtilization、MemoryUtilization、vCPUAvailable 和 MemoryAvailable，以確認工作負載有足夠的資源可用。

監控執行環境指標。追蹤執行環境層級指標，包括 ExecutionEnvironmentConcurrency 和 ExecutionEnvironmentConcurrencyLimit，以了解擴展行為並識別潛在的限流。

設定 CloudWatch 警示。為關鍵指標建立 CloudWatch 警示，以主動識別問題：

- CPU 或記憶體使用率高
- 可用容量低
- 接近並行限制

語言特定考量事項

遵循特定語言的最佳實務。每種程式設計語言處理多並行的方式都不同。檢閱語言特定指南以取得詳細建議：

- Java：將執行緒安全集合、 `AtomicInteger`和 `ThreadLocal`用於請求特定狀態
- Node.js：針對所有請求特定的狀態使用 InvokeStore，並避免全域變數
- Python：在 中使用唯一的檔案名稱`/tmp`搭配請求 IDs並考慮以程序為基礎的記憶體隔離
- Rust：使用 `run_concurrent`而非 `run`，並啟用 `concurrency-tokio`功能。處理常式必須是 `Clone` + `Send`。

測試執行緒安全和並行問題。在部署到生產環境之前，請徹底測試您的 函數是否有執行緒安全問題、競爭條件，以及在並行負載下的適當狀態隔離。

後續步驟

- 了解 Lambda 受管執行個體的容量提供者
- 了解 Lambda 受管執行個體的擴展
- 檢閱 Java、Node.js 和 Python 的執行時間特定指南
- 為您的容量提供者設定 VPC 連線
- 使用 CloudWatch 指標監控 Lambda 受管執行個體

故障診斷 Lambda 受管執行個體

調節和擴展問題

向上擴展期間的高錯誤率

問題：當流量快速增加時，您遇到限流錯誤 (HTTP 429)。

原因：Lambda 受管執行個體會根據 CPU 資源使用率和多並行飽和度以非同步方式擴展。如果您的流量在 5 分鐘內增加一倍以上，您可能會在 Lambda 擴展執行個體和執行環境以滿足需求時看到調節。

解決方案：

- 調整目標資源使用率：如果您的工作負載具有可預測的流量模式，請設定較低的目標資源使用率，以維持額外的流量暴增空間。
- 暖機前容量：對於計劃的流量增加，在較長的時間內逐漸增加流量，以允許擴展保持步調。
- 監控擴展指標：追蹤調節錯誤指標，以了解調節和容量擴展問題的原因。
- 檢閱函數組態：確保您的函數記憶體和 vCPU 設定支援多並行執行。視需要增加函數記憶體或 vCPU 配置。

緩慢縮減規模

問題：執行個體在流量減少後需要很長的時間才能縮減規模。

原因：Lambda 受管執行個體會逐漸縮減規模，以維持可用性，並避免可能影響效能的快速容量變更。

解決方案：

這是預期的行為。Lambda 會保守地縮減執行個體，以確保穩定性。監控您的 CloudWatch 指標，以追蹤執行中的執行個體數量。

並行問題

具有低並行體驗調節的執行環境

問題：儘管有可用的容量，您的函數仍會遇到限流。

原因：並行上限極低的執行環境可能無法有效地擴展。Lambda 受管執行個體專為多並行應用程式而設計。

解決方案：

- 增加並行上限：如果您的函數調用使用極少 CPU，請將並行上限設定為每個 vCPU 64。
- 最佳化函數程式碼：檢閱函數程式碼以減少每次呼叫的 CPU 使用量，從而實現更高的並行。
- 調整函數記憶體和 vCPU：確保您的函數有足夠的資源來處理多個並行調用。

執行緒安全問題 (Java 執行時間)

問題：您的 Java 函數會產生不正確的結果，或在載入時遇到競爭條件。

原因：多個執行緒同時執行處理常式方法，共用狀態不安全。

解決方案：

- 將 `AtomicInteger`或 `AtomicLong`用於計數器，而非基本類型
- 將 取代`HashMap`為 `ConcurrentHashMap`
- 使用 `Collections.synchronizedList()` 包裝 `ArrayList`
- 將 `ThreadLocal`用於請求特定狀態
- 從 Lambda 內容物件存取追蹤 IDs，而不是環境變數

如需詳細指引，請參閱 Lambda 受管執行個體的 Java 執行期文件。

狀態隔離問題 (Node.js 執行時間)

問題：您的 Node.js 函數會傳回來自不同請求的資料或發生資料損毀。

原因：全域變數會在相同工作者執行緒上的並行調用之間共用。當非同步操作產生控制時，其他調用可以修改共用狀態。

解決方案：

- 針對所有請求特定的狀態`@aws/lambda-invoke-store`安裝和使用
- 使用 `InvokeStore.set()`和 取代全域變數 `InvokeStore.get()`
- 在 中使用唯一的檔案名稱`/tmp`搭配請求 IDs
- 使用 `InvokeStore.getXRayTraceId()`而非環境變數存取追蹤 IDs

如需詳細指引，請參閱 Lambda 受管執行個體的 Node.js 執行期文件。

檔案衝突 (Python 執行時間)

問題：您的 Python 函數會從 中的檔案讀取不正確的資料`/tmp`。

原因：多個程序共用 `/tmp`目錄。同時寫入至相同檔案可能會導致資料損毀。

解決方案：

- 使用具有請求 IDs的唯一檔案名稱： `/tmp/request_{context.request_id}.txt`
- 將檔案鎖定搭配 `fcntl.flock()` 用於共用檔案
- 使用`os.remove()`後使用 清除暫存檔案

如需詳細指引，請參閱 Lambda 受管執行個體的 Python 執行期文件。

效能問題

高記憶體使用率

問題：您的函數遇到高記憶體使用率或out-of-memory錯誤。

原因：Python 中的每個並行請求會在具有其記憶體空間的個別程序中執行。總記憶體用量等於每個程序記憶體乘以並行程序。

解決方案：

- 在 CloudWatch `MemoryUtilization` 中監控指標
- 如果記憶體用量接近函數的記憶體限制，請減少`MaxConcurrency`設定
- 增加函數記憶體配置以支援更高的並行
- 透過隨需載入資料來最佳化記憶體用量，而不是在初始化期間

效能不一致

問題：函數效能在叫用之間有很大差異。

原因：Lambda 可能會根據可用性選擇不同的執行個體類型，或者函數可能會在資源可用性不同的執行個體上執行。

解決方案：

- 指定允許的執行個體類型：如果您有特定的效能需求，請在容量提供者中設定允許的執行個體類型，以限制 Lambda 可以選擇的執行個體類型。
- 監控執行個體層級指標：`MemoryUtilization`在容量提供者層級追蹤 `CPUUtilization`和 ，以識別資源限制。

- 檢閱容量指標：檢查 vCPUAvailable和 MemoryAvailable ，以確保執行個體上有足夠的資源可用。

容量提供者問題

函數版本無法成為 ACTIVE

問題：發佈後，您的函數版本會保持待定狀態。

原因：Lambda 正在啟動受管執行個體並啟動執行環境。此程序需要時間，尤其是新容量提供者的第一個函數版本。

解決方案：

等待 Lambda 完成初始化程序。Lambda 預設會針對 AZ 彈性啟動三個執行個體，並在標記函數版本 ACTIVE 之前啟動三個執行環境。這通常需要幾分鐘的時間。

無法刪除容量提供者

問題：嘗試刪除容量提供者時發生錯誤。

原因：您無法刪除已連接函數版本的容量提供者。

解決方案：

1. 使用容量提供者搭配 ListFunctionVersionsByCapacityProvider API 識別所有函數版本。
2. 刪除或更新這些函數版本，以移除容量提供者關聯。
3. 重試刪除容量提供者。

函數發佈期間的一般錯誤訊息

問題：發佈函數時，您遇到一般錯誤訊息，例如「發佈期間發生內部錯誤」。

解決方案：

- 檢查 IAM 許可：確定您具有嘗試使用的容量提供者lambda:PassCapacityProvider許可。
- 驗證容量提供者組態：使用 GetCapacityProvider API 確認您的容量提供者處於 ACTIVE 狀態。
- 檢閱 VPC 組態：確保容量提供者中指定的子網路和安全群組已正確設定並可存取。

- Check AWS CloudTrail 日誌：檢閱 CloudTrail 日誌，以取得有關失敗操作的詳細錯誤資訊。

監控和可觀測性問題

缺少 CloudWatch 指標

問題：您在 CloudWatch 中看不到容量提供者或函數的預期指標。

原因：指標每隔 5 分鐘發佈一次。新的容量提供者或函數可能沒有立即可用的指標。

解決方案：

在發佈函數版本後等待至少 5-10 分鐘，然後預期指標會出現在 CloudWatch 中。確認您查看的是正確的命名空間 (`AWS/Lambda`) 和維度 (`CapacityProviderName`、 `FunctionName`或 `InstanceType`)。

找不到 CloudWatch 日誌

問題：您的函數已成功執行，但無法在 CloudWatch Logs 中找到日誌。

原因：Lambda 受管執行個體會在您的 VPC 中執行，且需要網路連線才能將日誌傳送至 CloudWatch Logs。如果沒有適當的 VPC 連線組態，您的函數就無法連線到 CloudWatch Logs 服務端點。

解決方案：

設定 VPC 連線，讓您的函數將日誌傳送至 CloudWatch Logs。您有三種選項：

選項 1：CloudWatch Logs 的 VPC 端點 （建議用於生產）

1. 開啟位於 https：//console.aws.amazon.com/vpc/ 的 Amazon VPC 主控台。
2. 在導覽窗格中選擇端點。
3. 選擇建立端點。
4. 在 Service category (服務類別) 中，選擇 AWS services。
5. 針對服務名稱，選取 `com.amazonaws.region.logs`(`region`以您的 AWS 區域取代）。
6. 針對 VPC，選取容量提供者使用的 VPC。
7. 針對子網路，選取您要建立端點網路介面的子網路。如需高可用性，請選取多個可用區域中的子網路。
8. 針對安全群組，選取允許從函數安全群組傳入 HTTPS 流量 （連接埠 443) 的安全群組。

9. 為端點啟用私有 DNS。
10選擇建立端點。

選項 2：具有網際網路閘道的公有子網路

如果您的容量提供者使用公有子網路，請確保：

1. 網際網路閘道已連接至您的 VPC
2. 路由表會將`0.0.0.0/0`流量路由至網際網路閘道
3. 安全群組允許連接埠 443 上的傳出 HTTPS 流量

選項 3：具有 NAT 閘道的私有子網路

如果您的容量提供者使用私有子網路，請確保：

1. NAT 閘道存在於公有子網路中
2. 私有子網路路由表會將`0.0.0.0/0`流量路由至 NAT 閘道
3. 公有子網路路由表會將`0.0.0.0/0`流量路由至網際網路閘道
4. 安全群組允許連接埠 443 上的傳出 HTTPS 流量

如需 VPC 連線選項的詳細指引，請參閱 Lambda 受管執行個體的 VPC 連線。

從並行請求關聯日誌有困難

問題：來自不同請求的日誌會交錯，因此難以追蹤個別請求。

原因：預期日誌交錯，以及多並行系統中的標準行為。

解決方案：

- 使用 JSON 格式的結構化記錄：在所有日誌陳述式中包含請求 ID
- Java：搭配 使用 Log4j `ThreadContext` 來自動包含請求 ID
- Node.js：`console.log()`搭配 JSON 格式使用，並包含 `InvokeStore.getRequestId()`
- Python：使用具有 JSON 格式的標準記錄模組，並包含 `context.request_id`

如需詳細指引，請參閱執行時間特定的文件頁面。

取得其他協助

如果您在嘗試這些解決方案後持續遇到問題：

1. 檢閱 CloudWatch 指標：檢查容量提供者和執行環境指標，以識別資源限制或擴展問題。
2. Check AWS CloudTrail 日誌：檢閱 CloudTrail 日誌，以取得 API 呼叫和錯誤的詳細資訊。
3. 聯絡 AWS 支援：如果您無法解決問題，請聯絡 AWS 支援，提供有關容量提供者組態、函數組態以及您遇到的特定錯誤訊息的詳細資訊。

後續步驟

- 了解 Lambda 受管執行個體的容量提供者
- 了解 Lambda 受管執行個體的擴展
- 檢閱 Java、Node.js 和 Python 的執行時間特定指南
- 使用 CloudWatch 指標監控 Lambda 受管執行個體
- 檢閱 Lambda 受管執行個體的最佳實務

Lambda 執行期

Lambda 透過使用執行期支援多種語言。執行期會提供特定於語言的環境，其可轉傳調用事件、內容資訊以及 Lambda 與函數之間的回應。您可以使用由 Lambda 提供的執行期或是自行建置。

Lambda 與您選擇的執行時期無關。對於簡單的函數，Python 和 Node.js 等解釋型語言可提供最快的效能。對於運算較複雜的函數，像 Java 這樣的編譯語言通常初始化速度較慢，但在 Lambda 處理程序中的執行速度很快。執行時期的選擇也受開發人員偏好和語言熟悉程度的影響。

每個主要的程式設計語言版本都有獨立的執行階段，且具有唯一的執行階段識別符，例如 `nodejs24.x` 或 `python3.14`。若要設定函數以使用新的主要語言版本，您需變更執行期識別符。由於 AWS Lambda 無法保證主要版本之間的回溯相容性，因此這是客戶驅動的操作。

針對 定義為容器映像的函數，您可以在建立容器映像時選擇執行期和 Linux 發行版。若要變更執行期，請建立新的容器映像。

若要為部署套件使用 .zip 封存檔，您可以在建立函數時選擇執行期。若要變更執行期，您可以更新函數的組態。執行期與其中一個 Amazon Linux 發行版配對。基礎執行環境會提供其他程式庫以及環境變數，讓您可以從函數程式碼中存取。

Lambda 在執行環境中調用您的函數。執行環境提供安全且隔離的執行期環境，它會管理執行您的函數所需的資源。Lambda 會從之前的調用 (若有) 中重新使用執行環境，或者它會建立一個新的執行環境。

若要在 Lambda 中使用其他語言，例如 Go 或 Rust，請使用僅限作業系統的執行期。Lambda 執行環境提供了執行期介面，用於取得調用事件及傳送回應。您可以透過實作自訂執行期和您的函數程式碼，或在一個層中部署其他語言。

支援的執行時期

下表列出受支援的 Lambda 執行期和預計的棄用日期。在執行期被棄用後，您依然能在限定時間內建立與更新函數。如需詳細資訊，請參閱the section called “棄用後的執行時期使用情況”。根據我們的 the section called “執行期淘汰政策”，該表格提供了目前預測的執行時期的棄用日期。這些日期提供用於規劃目的，且可能發生變更。

> **Important**
>
> Amazon Linux 2 的生命週期將於 2026 年 6 月 30 日結束。除了語言執行期修補程式之外，適用於 Java 8 (AL2)、Java 11、Java 17、Python 3.10、Python 3.11 和 provided.al2 的

Lambda 執行期和容器基礎映像將繼續接收重大和所選重要 Amazon Linux 2 安全問題的修補程式，直到下表中顯示的棄用日期為止。
我們建議客戶盡快升級至以 Amazon Linux 2023 為基礎的執行時間。對於升級至 Java 21 或 Java 25 的客戶，您可以使用AWS Transform 自訂來協助進行這些升級。對於無法升級其 Java 版本的客戶，我們計劃在 2026 年第 Q2結束前發行 Java 8、Java 11 和 Java 17 的 Amazon Linux 2023 型執行期。

Name	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Node.js 24	`nodejs24.x`	Amazon Linux 2023	2028 年 4 月 30 日	2028 年 6 月 1 日	2028 年 7 月 1 日
Node.js 22	`nodejs22.x`	Amazon Linux 2023	2027 年 4 月 30 日	2027 年 6 月 1 日	2027 年 7 月 1 日
Node.js 20	`nodejs20.x`	Amazon Linux 2023	2026 年 4 月 30 日	2026 年 8 月 31 日	2026 年 9 月 30 日
Python 3.14	`python3.14`	Amazon Linux 2023	2029 年 6 月 30 日	2029 年 7 月 31 日	2029 年 8 月 31 日
Python 3.13	`python3.13`	Amazon Linux 2023	2029 年 6 月 30 日	2029 年 7 月 31 日	2029 年 8 月 31 日
Python 3.12	`python3.12`	Amazon Linux 2023	2028 年 10 月 31 日	2028 年 11 月 30 日	2029 年 1 月 10 日
Python 3.11	`python3.11`	Amazon Linux 2	2027 年 6 月 30 日	2027 年 7 月 31 日	2027 年 8 月 31 日
Python 3.10	`python3.10`	Amazon Linux 2	2026 年 10 月 31 日	2026 年 11 月 30 日	2027 年 1 月 15 日
Java 25	`java25`	Amazon Linux 2023	2029 年 6 月 30 日	2029 年 7 月 31 日	2029 年 8 月 31 日
Java 21	`java21`	Amazon Linux 2023	2029 年 6 月 30 日	2029 年 7 月 31 日	2029 年 8 月 31 日

Name	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Java 17	`java17`	Amazon Linux 2	2027 年 6 月 30 日	2027 年 7 月 31 日	2027 年 8 月 31 日
Java 11	`java11`	Amazon Linux 2	2027 年 6 月 30 日	2027 年 7 月 31 日	2027 年 8 月 31 日
Java 8	`java8.al2`	Amazon Linux 2	2027 年 6 月 30 日	2027 年 7 月 31 日	2027 年 8 月 31 日
.NET 10	`dotnet10`	Amazon Linux 2023	2028 年 11 月 14 日	2028 年 12 月 14 日	2029 年 1 月 15 日
.NET 9（僅限容器）	`dotnet9`	Amazon Linux 2023	2026 年 11 月 10 日	未排程	未排程
.NET 8	`dotnet8`	Amazon Linux 2023	2026 年 11 月 10 日	2026 年 12 月 10 日	2027 年 1 月 11 日
Ruby 3.4	`ruby3.4`	Amazon Linux 2023	2028 年 3 月 31 日	2028 年 4 月 30 日	2028 年 5 月 31 日
Ruby 3.3	`ruby3.3`	Amazon Linux 2023	2027 年 3 月 31 日	2027 年 4 月 30 日	2027 年 5 月 31 日
Ruby 3.2	`ruby3.2`	Amazon Linux 2	2026 年 3 月 31 日	2026 年 8 月 31 日	2026 年 9 月 30 日
僅限作業系統的執行期	`provided.al2023`	Amazon Linux 2023	2029 年 6 月 30 日	2029 年 7 月 31 日	2029 年 8 月 31 日
僅限作業系統的執行期	`provided.al2`	Amazon Linux 2	2026 年 7 月 31 日	2026 年 8 月 31 日	2026 年 9 月 30 日

> **Note**
>
> 對於新區域，Lambda 將不支援設定為在未來 6 個月內棄用的執行階段。

Lambda 透過修補程式和次要版本發布的支援，讓受管執行期和其對應的容器基礎映像檔保持在最新狀態。如需詳細資訊，請參閱 Lambda 執行期更新。

若要以程式設計方式與 Lambda 函數中的其他 AWS 服務 和資源互動，您可以使用其中一個 AWS SDKs。Node.js、Python 和 Ruby 執行時間包含 AWS SDK 的版本。不過，若要維持對相依項的完全控制權，並在執行時期自動更新期間最大化回溯相容性，建議一律將程式碼使用的 SDK 模組 (以及任何相依項) 納入函式的部署套件或 Lambda 層。

建議僅在部署中無法納入額外套件時，才使用執行時期內含的 SDK 版本。例如，當您使用 Lambda 主控台程式碼編輯器或在 CloudFormation 範本中使用內嵌函數程式碼建立函數時。

Lambda 會定期更新 Node.js、Python 和 Ruby 執行時間中包含的 AWS SDKs 版本。若要判斷目前所用執行時期內含的 AWS SDK 版本，請參閱下列章節：

- 執行時期內含 SDK 版本 (Node.js)
- 執行時期內含 SDK 版本 (Python)
- 執行時期內含 SDK 版本 (Ruby)

棄用 Go 1.x 執行期後，Lambda 將繼續支援 Go 程式設計語言。如需詳細資訊，請參閱AWS 運算部落格上的將 AWS Lambda 函數從 Go1.x 執行時間遷移至 Amazon Linux 2 上的自訂執行時間。

所有支援的 Lambda 執行時期都支援 x86_64 和 arm64 架構。

新執行期版本

Lambda 僅在發行版本到達語言發布週期的長期支援 (LTS) 階段時提供新語言版本的受管理執行期。例如，對於 Node.js 發布週期，即當發行版本達到 Active LTS 階段時。

在發行版本達到長期支援階段以前，它將停留於開發中，並且仍可能有重大變更。Lambda 依預設會自動套用執行期更新，因此對執行期版本的重大變更可能使您的函數無法如預期執行。

Lambda 不會為未排定 LTS 發行版本的語言版本提供受管理執行期。

下列清單顯示即將到來的 Lambda 執行期目標啟動月份。這些日期僅為象徵性，並可能有所變更。

- Ruby 3.5 - 2026 年 3 月
- AL2023 上的 Java 8、11 和 17 - 2026 年第 2 Q2
- Node.js 26 - 2026 年 11 月
- Python 3.15 - 2026 年 11 月

執行期淘汰政策

用於 .zip 壓縮檔的 Lambda 執行時期，均須以接受維護與安全性更新的作業系統、程式設計語言及軟體程式庫的組合為基礎建置。Lambda 的標準淘汰政策是在執行時期的任何主要元件到達社群長期支援 (LTS) 結束，且不再提供安全性更新時，淘汰執行時期。大多數情況下，這是語言執行期，雖然在某些情況下，執行期可以被棄用，因為操作系統 (OS) 到達 LTS 的結束。

取代執行時間之後， AWS 可能不再將安全修補程式或更新套用至該執行時間，使用該執行時間的函數將不再符合技術支援的資格。這類已棄用的執行時期會依現狀提供，無需任何保證，且可能包含問題、錯誤、瑕疵或其他漏洞。

若要進一步了解如何管理執行期升級和棄用，請參閱運算AWS 部落格上的下列章節和管理AWS Lambda 執行期升級。

> **Important**
>
> Lambda 偶爾會在執行期支援的語言版本的支援日期結束後，在有限期間內延遲棄用 Lambda 執行期。在此期間，Lambda 僅將安全性修補程式套用至執行期作業系統。Lambda 不會在到達支援結束日期之後，將安全性修補程式套用到程式設計語言執行期。

共同責任模型

Lambda 負責為所有受支援的受管執行時期與容器映像策劃和發布安全性更新。預設情況下，Lambda 會將這些更新自動套用至使用受管執行時期的函數。若預設自動執行時期更新設定已變更，請參閱執行時期管理控制共同責任模型。對於使用容器映像部署的函數，您需負責從最新的基礎映像重建函數的容器映像，並重新部署容器映像。

當執行時期已棄用時，Lambda 更新受管執行時期和容器基礎映像的責任會停止。您有責任升級函數以使用支援的執行時期或基礎映像。

在所有情況下，您都必須負責將更新套用至函數程式碼，包括其相依項。您在共同責任模型下的責任摘要如下表所示。

執行時期生命週期階段	Lambda 的責任	您的責任
支援的受管執行時期	使用安全修補程式和其他更新來提供定期執行時期更新。	更新您的函數程式碼，包括相依項，以解決任何安全漏洞。

執行時期生命週期階段	Lambda 的責任	您的責任
	根據預設自動套用執行時期更新 (請參閱 the section called "執行時期更新模式"，了解非預設行為)。	
支援的容器映像	使用安全修補程式和其他更新，為容器基礎映像提供定期更新。	更新您的函數程式碼，包括相依項，以解決任何安全漏洞。 使用最新的基礎映像，定期重建和重新部署容器映像。
即將棄用的受管執行時期	透過文件、 Health 儀板表電子郵件和 在執行時間棄用之前通知客戶 Trusted Advisor。 執行時期更新的責任會在棄用時結束。	監控 Lambda 文件 Health 儀板表、電子郵件或 Trusted Advisor 以取得執行時間棄用資訊。 在棄用先前的執行時期之前，將函數升級至受支援的執行時期。
即將棄用的容器映像	棄用通知不適用於使用容器映像的函數。 容器基礎映像更新的責任會在棄用時結束。	在棄用上一個映像之前，請注意棄用排程，並將函數升級至受支援的基礎映像。

棄用後的執行時期使用情況

取代執行時間之後， AWS 可能不再將安全修補程式或更新套用至該執行時間，使用該執行時間的函數將不再符合技術支援的資格。雖然您可以無限期地繼續叫用函數，但 AWS 強烈建議您遷移到支援的執行時間。已取代的執行時期「依現狀」提供，無任何保證，且可能包含問題、錯誤、瑕疵或其他漏洞。使用已棄用執行時期的函數也可能會遇到效能降低或其他問題，例如憑證過期，這可能會導致它們停止正常運作。

執行時期被取代後，您隨時可以將函式更新為使用較新版本的受支援執行時期。建議先使用新的執行時期測試函式，再在生產環境中套用變更。函式更新被封鎖後，將無法還原至已取代的執行時期。建議使用函數版本與別名功能，以啟用具有復原功能的安全部署。

下列時間軸說明了執行時期被取代後會發生的情況：

執行時期生命週期階段	當	內容
棄用通知期間	棄用前至少 180 天	• AWS 會透過電子郵件和 傳送通知 Health 儀板表 給在其$LATEST版本中使用此執行時間的 帳戶。 • 受影響的函式也會列於 Health 儀板表 的「排定的變更」索引標籤中，以及 AWS Trusted Advisor 已取代的執行時間檢查中。
棄用	取代日期	• AWS 可能不再套用安全性更新或其他更新。 • 函式不再符合技術支援資格。 • 您無法再於 Lambda 主控台透過已取代的執行時期建立或更新函式。您可以透過 AWS CLI AWS SAM或 繼續建立和更新函數 CloudFormation。
封鎖函數建立	棄用後至少 30 天	• Lambda 開始阻擋建立新函式。 • 您可以透過 AWS CLI AWS SAM或 繼續更新現有函數的程式碼和組態 CloudFormation。
封鎖函數更新	棄用後至少 60 天	• Lambda 開始阻擋更新現有函式的程式碼與組態。 • 您仍可將函式組態升級至支援的執行時期。不過，復原至已取代的執行時期可能會遭到阻擋。

> **Note**
>
> 對於某些執行時間， AWS 會將 block-function-create 和 block-function-update 日期延遲到棄用後 30 和 60 天。 AWS 已進行此變更以回應客戶意見回饋，讓您有更多時間升級函數。請參閱 the section called “支援的執行時期” 和 the section called “已取代的執行階段” 中的資料表，並查看執行時期的日期。此表格所列日期前，Lambda 不會開始阻擋建立或更新函式。

接收執行期棄用通知

當執行時間接近其棄用日期時，如果您 AWS 帳戶 使用該執行時間中的任何函數，Lambda 會傳送電子郵件提醒給您。通知也會顯示在 Health 儀板表 和 中 AWS Trusted Advisor。

- 接收電子郵件通知：

  執行期被棄用前至少 180 天，Lambda 會向您傳送電子郵件提醒。此電子郵件將列出所有使用執行期之函數的 $LATEST 版本。若要查看受影響函數版本的完整清單，請使用 Trusted Advisor 或參閱 the section called “依執行時期取得關於函數的資料”。

  Lambda AWS 帳戶會傳送電子郵件通知給您的主要帳戶聯絡人。如需有關檢視或更新帳戶中電子郵件地址的資訊，請參閱《AWS 一般參考》中的更新聯絡資訊。

- 透過 接收通知 Health 儀板表：

  會在執行時間棄用前至少 180 天 Health 儀板表 顯示通知。通知顯示在其他通知下方的您的帳戶運作狀態頁面上。通知的受影響資源索引標籤會列出所有使用執行期之函數的 $LATEST 版本。

  > **Note**
  >
  > 若要查看受影響函數版本的完整和up-to-date清單，請使用 Trusted Advisor 或參閱 the section called “依執行時期取得關於函數的資料”。

  Health 儀板表 通知會在受影響的執行時間棄用後 90 天過期。

- 使用 AWS Trusted Advisor

  Trusted Advisor 會在執行時間棄用前至少 180 天顯示通知。通知顯示在安全性頁面上。受影響函數的清單會顯示在使用已棄用執行期的AWS Lambda 函數的下方。此函數清單同時顯示 $LATEST 和已發佈版本，並且會自動更新以反映您的函數的最新狀態。

您可以在 Trusted Advisor 主控台的偏好設定頁面 Trusted Advisor 中，從 開啟每週電子郵件通知。

已取代的執行階段

下列執行階段已達到終止支援：

Name	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Python 3.9	python3.9	Amazon Linux 2	2025 年 12 月 15 日	2026 年 8 月 31 日	2026 年 9 月 30 日
Node.js 18	nodejs18.x	Amazon Linux 2	2025 年 9 月 1 日	2026 年 8 月 31 日	2026 年 9 月 30 日
.NET 6	dotnet6	Amazon Linux 2	2024 年 12 月 20 日	2026 年 8 月 31 日	2026 年 9 月 30 日
Python 3.8	python3.8	Amazon Linux 2	2024 年 10 月 14 日	2026 年 8 月 31 日	2026 年 9 月 30 日
Node.js 16	nodejs16.x	Amazon Linux 2	2024 年 6 月 12 日	2026 年 8 月 31 日	2026 年 9 月 30 日
.NET 7 (僅限容器)	dotnet7	Amazon Linux 2	2024 年 5 月 14 日	N/A	N/A
Java 8	java8	Amazon Linux	2024 年 1 月 8 日	2024 年 2 月 8 日	2026 年 9 月 30 日
Go 1.x	go1.x	Amazon Linux	2024 年 1 月 8 日	2024 年 2 月 8 日	2026 年 9 月 30 日
僅限作業系統的執行期	provided	Amazon Linux	2024 年 1 月 8 日	2024 年 2 月 8 日	2026 年 9 月 30 日
Ruby 2.7	ruby2.7	Amazon Linux 2	2023 年 12 月 7 日	2024 年 1 月 9 日	2026 年 9 月 30 日

Name	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Node.js 14	nodejs14.x	Amazon Linux 2	2023 年 12 月 4 日	2024 年 1 月 9 日	2026 年 9 月 30 日
Python 3.7	python3.7	Amazon Linux	2023 年 12 月 4 日	2024 年 1 月 9 日	2026 年 9 月 30 日
.NET Core 3.1	dotnetcore3.1	Amazon Linux 2	2023 年 4 月 3 日	2023 年 4 月 3 日	2023 年 5 月 3 日
Node.js 12	nodejs12.x	Amazon Linux 2	2023 年 3 月 31 日	2023 年 3 月 31 日	2023 年 4 月 30 日
Python 3.6	python3.6	Amazon Linux	2022 年 7 月 18 日	2022 年 7 月 18 日	2022 年 8 月 29 日
.NET 5 (僅限容器)	dotnet5.0	Amazon Linux 2	2022 年 5 月 10 日	N/A	N/A
.NET Core 2.1	dotnetcore2.1	Amazon Linux	2022 年 1 月 5 日	2022 年 1 月 5 日	2022 年 4 月 13 日
Node.js 10	nodejs10.x	Amazon Linux 2	2021 年 7 月 30 日	2021 年 7 月 30 日	2022 年 2 月 14 日
Ruby 2.5	ruby2.5	Amazon Linux	2021 年 7 月 30 日	2021 年 7 月 30 日	2022 年 3 月 31 日
Python 2.7	python2.7	Amazon Linux	2021 年 7 月 15 日	2021 年 7 月 15 日	2022 年 5 月 30 日
Node.js 8.10	nodejs8.10	Amazon Linux	2020 年 3 月 6 日	2020 年 2 月 4 日	2020 年 3 月 6 日
Node.js 4.3	nodejs4.3	Amazon Linux	2020 年 3 月 5 日	2020 年 2 月 3 日	2020 年 3 月 5 日
Node.js 4.3 邊緣	nodejs4.3-edge	Amazon Linux	2020 年 3 月 5 日	2019 年 3 月 31 日	2019 年 4 月 30 日

Name	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Node.js 6.10	`nodejs6.10`	Amazon Linux	2019 年 8 月 12 日	2019 年 7 月 12 日	2019 年 8 月 12 日
.NET Core 1.0	`dotnetcore1.0`	Amazon Linux	2019 年 6 月 27 日	2019 年 6 月 30 日	2019 年 7 月 30 日
.NET Core 2.0	`dotnetcore2.0`	Amazon Linux	2019 年 5 月 30 日	2019 年 4 月 30 日	2019 年 5 月 30 日
Node.js 0.10	`nodejs`	Amazon Linux	2016 年 8 月 30 日	2016 年 9 月 30 日	2016 年 10 月 31 日

在幾乎大多數情況下，都會預先知道語言版本或作業系統的生命週期結束日期。以下連結提供 Lambda 以受管執行期形式支援的每種語言之生命週期結束排程。

語言和框架支援政策

- Node.js – github.com
- Python – devguide.python.org
- Ruby – www.ruby-lang.org
- Java – www.oracle.com 和 Corretto 常見問答集
- Go – golang.org
- .NET – dotnet.microsoft.com

了解 Lambda 如何管理執行時期版本更新

Lambda 提供安全性更新、錯誤修正、新功能、效能增強功能，以及次要版本支援，讓每個受管理執行階段都保持最新狀態。這些執行階段更新會發佈為執行階段版本。Lambda 會將函數從舊的執行階段版本遷移至新的執行階段版本，藉此將執行階段更新套用至函數。

預設情況下，對於使用受管執行階段的函數，Lambda 會自動套用執行階段更新。若使用自動執行階段更新，Lambda 需承受修補執行階段版本的運作負擔。對大多數客戶而言，自動更新是正確的選擇。您可以透過設定執行時期管理設定來變更此預設行為。

Lambda 也會將每個新的執行階段版本發佈為容器映像。若要更新容器型函數的執行階段版本，您必須從更新的基礎映像建立新的容器映像，然後重新部署函數。

每個執行階段版本都與一個版本編號和一個 ARN (Amazon Resource Name) 相關聯。執行階段版本編號使用 Lambda 定義的編號結構描述，與程式設計語言使用的版本編號無關。執行時期版本編號並非總是連續的。例如，版本 42 之後可能會是版本 45。執行階段版本 ARN 是每個執行階段版本的唯一識別符。您可以在 Lambda 主控台或函數日誌的 `INIT_START` 行中，檢視函數目前執行時期版本的 ARN。

執行階段版本不應與執行階段識別符混淆。每個執行階段都有唯一的執行階段識別符，例如 `python3.14` 或 `nodejs24.x`。這些會對應到每個主要的程式設計語言版本。執行階段版本會描述個別執行階段的修補程式版本。

> **Note**
>
> 相同執行時間版本編號的 ARN 可能因 AWS 區域 和 CPU 架構而異。

主題

- 回溯相容性
- 執行時期更新模式
- 兩階段執行階段版本推展
- 設定 Lambda 執行時期管理設定
- 復原 Lambda 執行時期版本
- 識別 Lambda 執行時期版本變更
- 了解 Lambda 執行時期管理的共同責任模式
- 控制高合規應用程式的 Lambda 執行時期更新許可

回溯相容性

Lambda 致力於提供與現有函數回溯相容的執行階段更新。但是與軟體修補一樣，在極少數情況下，執行階段更新可能會對現有函數產生負面影響。例如，安全性修補程式可能會暴露現有函數的潛在問題，取決於先前的不安全行為。

在建置與部署函式時，務必掌握相依項的管理方法，避免執行時期未來更新時發生相容性問題。舉例而言，若函式相依於套件 A，而套件 A 又相依於套件 B，這兩個套件已內建於 Lambda 執行時期 (例如，二者可能是 SDK 或其相依項的一部分，或是執行時期系統程式庫的一部分)。

請考量下列情況：

部署	修補相容性	Reason
• 套件 A：使用執行時期內建版本 • 套件 B：使用執行時期內建版本	是	未來對套件 A 與 B 進行的執行時期更新將保持回溯相容性。
• 套件 A：封裝於部署套件中 • 套件 B：封裝於部署套件中	是	您的部署具有優先性，因此未來對套件 A 與 B 進行的執行時期更新不會產生影響。
• 套件 A：封裝於部署套件中 • 套件 B：使用執行時期內建版本	是*	未來對套件 B 進行的執行時期更新將保持回溯相容性。 *如果 A 與 B 緊密耦合，可能引發相容性問題。例如，適用於 Python 的 AWS SDK 中的 `boto3`和 `botocore`套件應該一起部署。
• 套件 A：使用執行時期內建版本 • 套件 B：封裝於部署套件中	否	未來對套件 A 進行的執行時期更新，可能需要相應更新套件 B 的版本。然而，已部署的套件 B 版本具有優先性，且可能與套件 A 的更新版本不具備向前相容性。

若要與未來執行時期更新保持相容性，請遵循下列最佳實務：

- 盡可能封裝所有相依性：在您的部署套件中包含所有必要的程式庫，包括 AWS SDK 及其相依性。此舉能確保元件組合的穩定性與相容性。
- 謹慎使用執行時間提供的 SDKs：只有在您無法包含其他套件時（例如，使用 Lambda 主控台程式碼編輯器或在 AWS CloudFormation 範本中使用內嵌程式碼時），才依賴執行時間提供的 SDK。
- 避免覆寫系統程式庫：請勿部署可能與未來執行時期更新衝突的自訂作業系統程式庫。

執行時期更新模式

Lambda 致力於提供與現有函數回溯相容的執行階段更新。但是與軟體修補一樣，在極少數情況下，執行階段更新可能會對現有函數產生負面影響。例如，安全性修補程式可能會暴露現有函數的潛在問題，取決於先前的不安全行為。在罕見的執行階段版本不相容情況下，Lambda 執行階段管理控制有助於降低對工作負載造成影響的風險。針對每個函數版本 ($LATEST 或已發佈版本)，您可以選擇下列其中一種執行階段更新模式：

- Auto (default) (自動 (預設)) - 使用 兩階段執行階段版本推展 自動更新為最新且安全的執行階段版本。我們建議大多數客戶使用此模式，便能隨時受益於執行階段更新。
- 函數更新：當您更新函數時，更新為最新且安全的執行階段版本。當您更新函數時，Lambda 會將函數的執行階段更新為最新且安全的執行階段版本。此做法可同步執行階段更新與函數部署，讓您控制 Lambda 何時套用執行階段更新。使用此模式，您可以及早偵測並減輕罕見的執行階段更新不相容情況。使用此模式時，您必須定期更新函數，讓執行階段保持最新狀態。
- 手動：手動更新您的執行階段版本。您可以在函數組態中指定執行階段版本。該函數將無限期使用此執行階段版本。在罕見情況下，新的執行階段版本會與現有函數不相容，您可以使用此模式，讓函數復原至較舊的執行階段版本。建議您不要使用 Manual (手動) 模式來嘗試達到跨部署的執行階段一致性。如需詳細資訊，請參閱復原 Lambda 執行時期版本。

將執行階段更新套用至函數的責任分配，會根據您選擇的執行階段更新模式而有所不同。如需詳細資訊，請參閱了解 Lambda 執行時期管理的共同責任模式。

兩階段執行階段版本推展

Lambda 會依照以下順序引入新的執行階段版本：

1. 在第一階段，每當您建立或更新函數，Lambda 都會套用新的執行階段版本。當您呼叫 UpdateFunctionCode 或 UpdateFunctionConfiguration API 操作，函數便會更新。

2. 在第二階段，Lambda 會更新任何使用 Auto (自動) 執行階段更新模式且尚未更新為新執行階段版本的函數。

推展程序的整體持續時間會因許多因素而有所不同，包括執行階段更新中所包含任何安全性修補程式的嚴重性。

如果您正在開發和部署函數，很可能會在第一階段選擇新的執行階段版本。這會將執行階段更新與函數更新同步。在極少數情況下，最新的執行階段版本會對您的應用程式造成負面影響，而此做法可讓您立即採取更正動作。非開發中的函數仍然會在第二階段獲得自動執行階段更新的運作效益。

此方法不會影響設為 Function update (函數更新) 或 Manual (手動) 模式的函數。使用 Function update (函數更新) 模式的函數，只會在您建立或更新函數時收到最新的執行階段更新。使用 Manual (手動) 模式的函數不會收到執行階段更新。

Lambda 以漸進、滾動方式於 AWS 區域發佈新的執行階段版本。如果您的函數設為 Auto (自動) 或 Function update (函數更新) 模式，那麼同時部署至不同區域或同一地區在不同時間部署的函數，可能會採用不同的執行階段版本。需要在不同環境中保證執行期版本一致性的客戶，應 使用容器映像來部署 Lambda 函數。手動 模式設計為暫時緩解措施，可在發生執行階段版本不相容函數的罕見情況下啟用執行階段版本復原。

設定 Lambda 執行時期管理設定

您可以使用 Lambda 主控台或 AWS Command Line Interface (AWS CLI) 來配置執行階段管理設定。

> **Note**
>
> 您可以針對每個函數版本分別配置執行階段管理設定。

設定 Lambda 如何更新執行階段版本 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 在 Code (程式碼) 索引標籤的 Runtime settings (執行階段設定) 底下，選擇 Edit runtime management configuration (編輯執行階段組態)。
4. 在 Runtime management configuration (執行階段管理組態) 底下，選擇下列其中一項：
   - 若要讓函數自動更新為最新的執行階段版本，請選擇 Auto (自動)。
   - 若要在變更函數時將函數更新為最新的執行階段版本，請選擇 Function update (函數更新)。

- 若要讓您的函數只在變更執行階段版本 ARN 時才更新為最新的執行階段版本，請選擇 Manual (手動)。您可以在 Runtime management configuration (執行階段管理組態) 下找到執行階段版本 ARN。您也可以在函數日誌的 `INIT_START` 行中找到 ARN。

如需關於這些選項的詳細資訊，請參閱執行時期更新模式。

5. 選擇儲存。

設定 Lambda 如何更新執行階段版本 (AWS CLI)

若要設定函數的執行時期管理，請執行 put-runtime-management-config AWS CLI 命令。使用 `Manual` 模式時，您還必須提供執行階段版本 ARN。

```
aws lambda put-runtime-management-config \
  --function-name my-function \
  --update-runtime-on Manual \
  --runtime-version-arn arn:aws:lambda:us-
east-2::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1
```

您應該會看到類似下列的輸出：

```
{
  "UpdateRuntimeOn": "Manual",
  "FunctionArn": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
  "RuntimeVersionArn": "arn:aws:lambda:us-
east-2::runtime:8eeff65f6809a3ce81507fe733fe09b835899b99481ba22fd75b5a7338290ec1"
}
```

復原 Lambda 執行時期版本

在罕見情況下，新的執行階段版本會與現有函數不相容，您可以將執行階段版本復原為較舊的版本。這可讓您的應用程式保持正常運作並將中斷時間縮到最短，這樣便有時間在改回最新執行階段版本之前修正不相容情況。

Lambda 不會對您可以使用任何特定執行階段版本的時間長度施加時間限制。不過，我們強烈建議您盡快更新至最新的執行階段版本，以便享有最新的安全性修補程式、效能改善項目和功能。Lambda 提供復原為較舊執行階段版本的選項，只能在發生執行階段更新相容性問題的罕見情況中當成暫時緩解的手段。長時間使用舊版執行階段版本的函數，最終可能會遇到效能降低或發生問題 (例如憑證到期)，這可能會導致它們無法正常運作。

您可以透過下列方式復原執行階段版本：

- 使用手動執行時期更新模式
- 使用已發佈的函數版本

如需詳細資訊，請參閱AWS Compute 部落格上的 AWS Lambda 執行階段管理控制介紹。

使用 Manual (手動) 執行階段更新模式復原執行階段版本

如果您使用的是 Auto (自動) 執行階段版本更新模式，或是使用 $LATEST 執行階段版本，您可以使用 Manual (手動) 模式復原執行階段版本。針對您要復原的函數版本，將執行階段版本更新模式變更為 Manual (手動)，並指定上一個執行階段版本的 ARN。如需查找舊版執行階段 ARN 的詳細資訊，請參閱：識別 Lambda 執行時期版本變更。

> **Note**
>
> 如果您的函數版本 $LATEST 設定為使用 Manual (手動) 模式，則無法變更函數使用的 CPU 架構或執行階段版本。若要進行這些變更，您必須變更為 Auto (自動) 或 Function update (函數更新) 模式。

使用已發布的函數版本復原執行階段版本

已發佈的函數版本是您建立 $LATEST 函數程式碼和組態時的不可變快照。在 Auto (自動) 模式中，Lambda 會在執行階段版本推展的第二階段期間，自動更新已發佈函數版本的執行階段版本。在 Function update (函數更新) 模式中，Lambda 不會更新已發佈函數版本的執行階段版本。

因此，使用 Function update (函數更新) 模式發佈的函數版本，會建立函數程式碼、組態和執行階段版本的靜態快照。透過將 Function update (函數更新) 模式與函數版本搭配使用，您可以將執行階段更新與部署同步。您也可以將流量重新導向至先前發佈的函數版本，協調程式碼、組態和執行階段版本的復原。您可以將此做法整合到持續整合和持續交付 (CI/CD) 中，以便在發生執行階段更新不相容的罕見情況下進行全自動復原。使用此做法時，您必須定期更新函數，並發佈新的函數版本以取得最新的執行階段更新。如需更多詳細資訊，請參閱 了解 Lambda 執行時期管理的共同責任模式。

識別 Lambda 執行時期版本變更

執行時期版本編號和 ARN 會記錄在 `INIT_START` 日誌行中，而 Lambda 每次建立新的執行環境時都會向 CloudWatch Logs 發送該日誌行。由於執行環境對所有函數調用都是使用相同的執行期版本，因

此 Lambda 只會在執行初始化階段時發送 INIT_START 日誌行。Lambda 不會為每個函數調用發出此日誌行。Lambda 會將此日誌行發送到 CloudWatch Logs，但不會顯示在主控台中。

> **Note**
>
> 執行時期版本編號並非總是連續的。例如，版本 42 之後可能會是版本 45。

Example INIT_START 日誌行

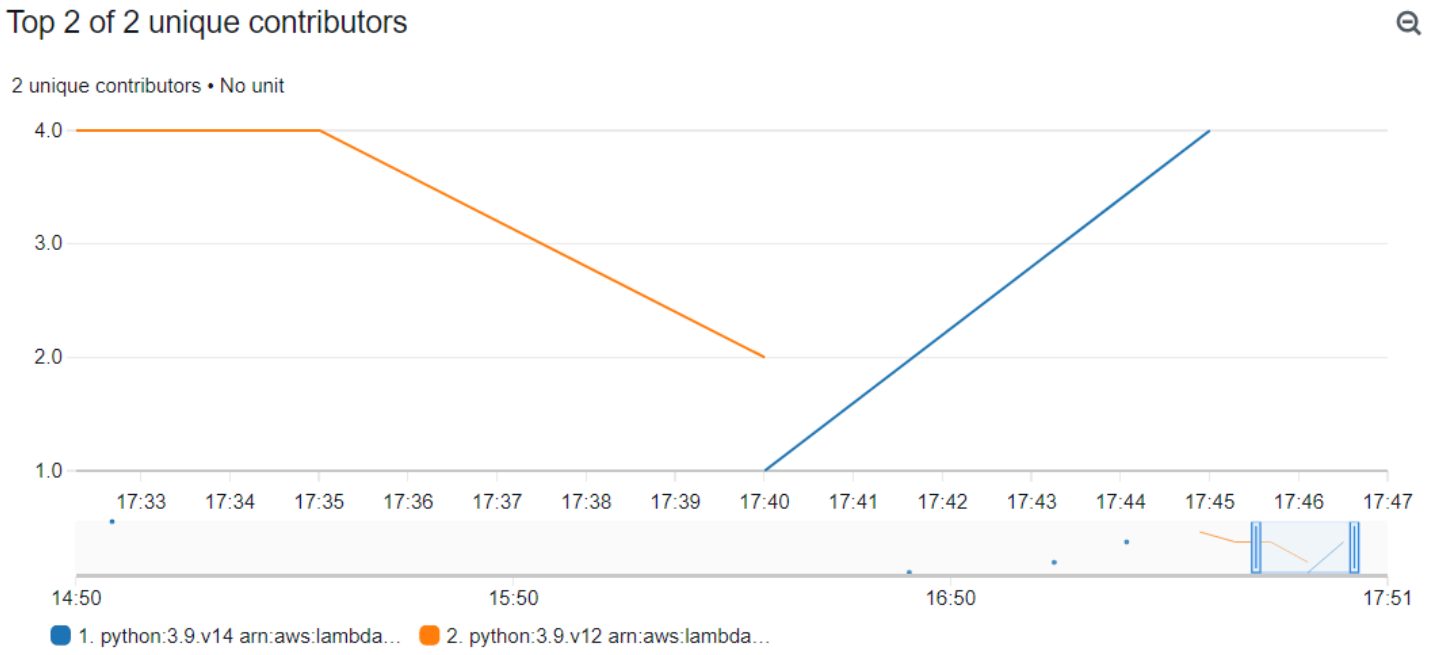
```
INIT_START Runtime Version: python:3.13.v14    Runtime Version ARN: arn:aws:lambda:eu-
south-1::runtime:7b620fc2e66107a1046b140b9d320295811af3ad5d4c6a011fad1fa65127e9e6I
```

您可以使用 Amazon CloudWatch Contributor Insights 來識別執行階段版本之間的轉換情形，而不是直接使用日誌。下列規則會計算每個 INIT_START 日誌行中不同的執行階段版本。若要使用此規則，請將範例日誌群組名稱 /aws/lambda/* 取代為函數或函數群組的適當前置詞。

```
{
  "Schema": {
    "Name": "CloudWatchLogRule",
    "Version": 1
  },
  "AggregateOn": "Count",
  "Contribution": {
    "Filters": [
      {
        "Match": "eventType",
        "In": [
          "INIT_START"
        ]
      }
    ],
    "Keys": [
      "runtimeVersion",
      "runtimeVersionArn"
    ]
  },
  "LogFormat": "CLF",
  "LogGroupNames": [
    "/aws/lambda/*"
  ],
  "Fields": {
```

```
    "1": "eventType",
    "4": "runtimeVersion",
    "8": "runtimeVersionArn"
  }
}
```

下列 CloudWatch Contributor Insights 報告顯示上述規則擷取的執行階段版本轉換範例。橘線顯示舊執行階段版本 (python:3.13.v12) 的執行環境初始化，藍線顯示新執行階段版本 (python:3.13.v14) 的執行環境初始化。



了解 Lambda 執行時期管理的共同責任模式

Lambda 負責為所有受支援的受管執行階段與容器映像策劃和發佈安全性更新。更新現有函數以使用最新執行階段版本的責任分配，視您使用的執行階段更新模式而有所不同。

Lambda 負責將執行階段更新套用至設定為使用 Auto (自動) 執行階段更新模式的所有函數。

對於使用 Function update (函數更新) 執行階段更新模式設定的函數，您必須負責定期更新函數。當您進行這些更新，Lambda 會負責套用執行階段更新。如果您沒有更新函數，Lambda 就不會更新執行階段。如果您沒有定期更新函數，強烈建議您設定為自動執行階段更新，以便繼續收到安全性更新。

對於設定為使用 Manual (手動) 執行階段更新模式的函數，您必須負責將函數更新為使用最新的執行階段版本。強烈建議您使用此模式的目的，只是在發生執行階段更新不相容的罕見情況下，將執行階段版本復原當成暫時緩和措施。我們也建議您盡快變更為 Auto (自動) 模式，以縮短函數處於未修補狀態的時間。

如果您使用容器映像來部署函數，則 Lambda 負責發佈更新的基礎映像。在這種情況下，您需負責從最新的基礎映像重建函數的容器映像，並重新部署容器映像。

以下表進行總結：

部署模式	Lambda 的責任	客户的責任
受管執行階段，Auto (自動) 模式	發佈包含最新修補程式的新執行階段版本。 將執行階段修補程式套用到現有函數	在發生執行階段更新相容性問題的罕見情況下，還原至先前的執行階段版本。請遵循回溯相容性的最佳實務。
受管執行階段，Function update (函數更新) 模式	發佈包含最新修補程式的新執行階段版本。	定期更新函數以取得最新的執行階段版本。 若您沒有定期更新函數，請將函數切換至Auto (自動) 模式。 在發生執行階段更新相容性問題的罕見情況下，還原至先前的執行階段版本。請遵循回溯相容性的最佳實務。
受管執行階段，Manual (手動) 模式	發佈包含最新修補程式的新執行階段版本。	此模式僅適用於發生執行階段更新相容性問題的罕見情況時，暫時復原執行階段。 請盡快將函數切換至 Auto (自動) 或 Function update (函數更新) 模式及最新的執行階段版本。
容器映像	發佈包含最新修補程式的新容器映像。	使用最新的容器基礎映像定期重新部署函數，以取得最新的修補程式。

如需與 AWS 的共同責任的詳細資訊，請參閱共同責任模式。

控制高合規應用程式的 Lambda 執行時期更新許可

為了滿足修補需求，Lambda 客戶通常仰賴自動執行階段更新。如果您的應用程式對修補更新狀態有嚴格要求，您可能需要限制使用舊版執行階段。您可以使用 AWS Identity and Access Management (IAM) 來限制 Lambda 的執行階段管理控制，以拒絕 AWS 帳戶中的使用者存取

PutRuntimeManagementConfig API 操作。此操作用於選擇函數的執行階段更新模式。拒絕存取此操作會導致所有函數均預設為 Auto (自動) 模式。您可以使用服務控制政策 (SCP) 在對整個組織套用此限制。如果必須將函數復原至較舊的執行時期版本，您可以依個別情況授予政策例外狀況。

擷取使用已棄用執行時期之 Lambda 函數的資料

當 Lambda 執行時間即將棄用時，Lambda 會透過電子郵件提醒您，並在 Health 儀板表 和 中提供通知 Trusted Advisor。這些電子郵件和通知會列出使用執行時期的 $LATEST 函數版本。若要列出使用特定執行時間的所有函數版本，您可以使用 AWS Command Line Interface (AWS CLI) 或其中一個 AWS SDKs。

如果您有大量函數使用由於已棄用而產生的執行時間，您也可以使用 AWS CLI AWS SDKs 來協助您排定最常調用函數的更新優先順序。

請參閱下列各節，了解如何使用 AWS CLI 和 AWS SDKs來收集使用特定執行時間之函數的資料。

列出使用特定執行時期的函數版本

若要使用 AWS CLI 列出所有使用特定執行時間的函數版本，請執行下列命令。`RUNTIME_IDENTIFIER` 將 取代為已棄用之執行時間的名稱，然後選擇您自己的執行時間 AWS 區域。如僅需列出 $LATEST 函數版本，請省略命令中的 `--function-version ALL`。

```
aws lambda list-functions --function-version ALL --region us-east-1 --output text --
query "Functions[?Runtime=='RUNTIME_IDENTIFIER'].FunctionArn"
```

> **Tip**
>
> 範例命令會列出特定 `us-east-1` 區域中的函數 AWS 帳戶 。您將需要針對帳戶具有函數的每個區域和您的每個區域重複此命令 AWS 帳戶。

您也可以使用其中一個 AWS SDKs 列出使用特定執行期的函數。下列範例程式碼使用 V3 適用於 JavaScript 的 AWS SDK 和 適用於 Python (Boto3) 的 AWS SDK 來傳回使用特定執行時期之函數的函數 ARN 清單。範例程式碼也會傳回每個所列函數的 CloudWatch 日誌群組。您可以使用此日誌群組來尋找函數的上次調用日期。如需詳細資訊，請參閱下一節the section called “識別最常調用和最近調用的函數”。

Node.js

Example列出使用特定執行時期之函數的 JavaScript 程式碼

```
import { LambdaClient, ListFunctionsCommand } from "@aws-sdk/client-lambda";
const lambdaClient = new LambdaClient();
```

```
const command = new ListFunctionsCommand({
    FunctionVersion: "ALL",
    MaxItems: 50
});
const response = await lambdaClient.send(command);

for (const f of response.Functions){
    if (f.Runtime == '<your_runtime>'){ // Use the runtime id, e.g. 'nodejs24.x' or
 'python3.14'
        console.log(f.FunctionArn);
        // get the CloudWatch log group of the function to
        // use later for finding the last invocation date
        console.log(f.LoggingConfig.LogGroup);
    }
}
// If your account has more functions than the specified
// MaxItems, use the returned pagination token in the
// next request with the 'Marker' parameter
if ('NextMarker' in response){
    let paginationToken = response.NextMarker;
  }
```

Python

Example列出使用特定執行時期之函數的 Python 程式碼

```
import boto3
from botocore.exceptions import ClientError

def list_lambda_functions(target_runtime):

    lambda_client = boto3.client('lambda')

    response = lambda_client.list_functions(
        FunctionVersion='ALL',
        MaxItems=50
    )
    if not response['Functions']:
            print("No Lambda functions found")
    else:
        for function in response['Functions']:
            if function['PackageType']=='Zip' and function['Runtime'] ==
 target_runtime:
```

```
                print(function['FunctionArn'])
                # Print the CloudWatch log group of the function
                # to use later for finding last invocation date
                print(function['LoggingConfig']['LogGroup'])

    if 'NextMarker' in response:
       pagination_token = response['NextMarker']

if __name__ == "__main__":
    # Replace python3.12 with the appropriate runtime ID for your Lambda functions
    list_lambda_functions('python3.12')
```

若要進一步了解如何使用 AWS SDK 來列出使用 ListFunctions 動作的函數，請參閱您偏好程式設計語言的 SDK 文件。

您也可以使用 AWS Config 進階查詢功能，列出所有使用受影響執行時間的函數。此查詢只會傳回函數 $LATEST 版本，但您可以使用 AWS 帳戶 單一命令彙總查詢以列出所有區域和多個區域的函數。若要進一步了解，請參閱《 AWS Config 開發人員指南》中的查詢 AWS Auto Scaling 資源的目前組態狀態。

識別最常調用和最近調用的函數

如果您的 AWS 帳戶 包含使用由於已棄用之執行時間的函數，建議您優先更新經常調用的函數或最近調用的函數。

如果您只有幾個函數，您可以使用 CloudWatch Logs 主控台查看函數的日誌串流以收集此資訊。如需詳細資訊，請參閱檢視傳送至 CloudWatch Logs 的日誌資料。

若要查看最近的函數調用次數，您也可以使用 Lambda 主控台中顯示的 CloudWatch 指標資訊。若要檢視此資訊，請執行下列步驟：

1. 開啟 Lambda 主控台中的函數頁面。
2. 選取您要查看調用統計資料的函數。
3. 選擇 Monitor (監控) 索引標籤。
4. 使用日期範圍選擇器設定您希望檢視統計資料的期間。最近調用會顯示在調用窗格中。

對於具有較多函數的帳戶，使用 DescribeLogStreams 和 GetMetricStatistics API 動作，以程式設計方式使用 AWS CLI 或其中一個 AWS SDKs 收集此資料會更有效率。

下列範例提供使用 V3 適用於 JavaScript 的 AWS SDK 和 的程式碼片段 適用於 Python (Boto3) 的 AWS SDK ，以識別特定函數的上次調用日期，並判斷過去 14 天內特定函數的調用次數。

Node.js

Example用於尋找函數上次調用時間的 JavaScript 程式碼

```
import { CloudWatchLogsClient, DescribeLogStreamsCommand } from "@aws-sdk/client-
cloudwatch-logs";
const cloudWatchLogsClient = new CloudWatchLogsClient();
const command = new DescribeLogStreamsCommand({
    logGroupName: '<your_log_group_name>',
    orderBy: 'LastEventTime',
    descending: true,
    limit: 1
});
try {
    const response = await cloudWatchLogsClient.send(command);
    const lastEventTimestamp = response.logStreams.length > 0 ?
        response.logStreams[0].lastEventTimestamp : null;
    // Convert the UNIX timestamp to a human-readable format for display
    const date = new Date(lastEventTimestamp).toLocaleDateString();
    const time = new Date(lastEventTimestamp).toLocaleTimeString();
    console.log(`${date} ${time}`);

} catch (e){
    console.error('Log group not found.')
}
```

Python

Example用於尋找函數上次調用時間的 Python 程式碼

```
import boto3
from datetime import datetime

cloudwatch_logs_client  = boto3.client('logs')

response = cloudwatch_logs_client.describe_log_streams(
    logGroupName='<your_log_group_name>',
    orderBy='LastEventTime',
    descending=True,
```

```
    limit=1
)

try:
    if len(response['logStreams']) > 0:
        last_event_timestamp = response['logStreams'][0]['lastEventTimestamp']
        print(datetime.fromtimestamp(last_event_timestamp/1000)) # Convert timestamp
 from ms to seconds
    else:
        last_event_timestamp = None
except:
    print('Log group not found')
```

> **Tip**
>
> 您可以使用 ListFunctions API 作業來尋找函數的日誌群組名稱。如需如何執行此作業的範例，請參閱the section called “列出使用特定執行時期的函數版本”。

Node.js

Example用於尋找過去 14 天內調用次數的 JavaScript 程式碼

```
import { CloudWatchClient, GetMetricStatisticsCommand } from "@aws-sdk/client-
cloudwatch";
const cloudWatchClient = new CloudWatchClient();
const command = new GetMetricStatisticsCommand({
    Namespace: 'AWS/Lambda',
    MetricName: 'Invocations',
    StartTime: new Date(Date.now()-86400*1000*14), // 14 days ago
    EndTime: new Date(Date.now()),
    Period: 86400 * 14, // 14 days.
    Statistics: ['Sum'],
    Dimensions: [{
        Name: 'FunctionName',
        Value: '<your_function_name>'
    }]
});
const response = await cloudWatchClient.send(command);
const invokesInLast14Days = response.Datapoints.length > 0 ?
    response.Datapoints[0].Sum : 0;
```

```
console.log('Number of invocations: ' + invokesInLast14Days);
```

Python

Example用於尋找過去 14 天內調用次數的 Python 程式碼

```
import boto3
from datetime import datetime, timedelta

cloudwatch_client = boto3.client('cloudwatch')

response = cloudwatch_client.get_metric_statistics(
    Namespace='AWS/Lambda',
    MetricName='Invocations',
    Dimensions=[
        {
            'Name': 'FunctionName',
            'Value': '<your_function_name>'
        },
    ],
    StartTime=datetime.now() - timedelta(days=14),
    EndTime=datetime.now(),
    Period=86400 * 14, # 14 days
    Statistics=[
        'Sum'
    ]
)

if len(response['Datapoints']) > 0:
    invokes_in_last_14_days = int(response['Datapoints'][0]['Sum'])
else:
    invokes_in_last_14_days = 0

print(f'Number of invocations: {invokes_in_last_14_days}')
```

修改執行階段環境

您可以使用內部延伸項目來修改執行階段程序。內部延伸項目不是單獨程序，它們會作為執行時程序的一部分執行。

Lambda 提供特定語言環境變數，您可以進行設定，以便將選項和工具新增至執行時間。Lambda 還提供包裝函數指令碼，它允許 Lambda 將執行時間啟動委派給您的指令碼。您可以建立包裝程式指令碼來自訂執行階段啟動行為。

特定語言的環境變數

Lambda 支援僅限組態的方式，可透過下列特定語言的環境變數，在函數初始化期間預先載入程式碼：

- `JAVA_TOOL_OPTIONS` - 在 Java 中，Lambda 支援此環境變數，以在 Lambda 中設定其他命令列變數。此環境變數可讓您指定工具的初始化，特別是在您使用 `agentlib` 或 `javaagent` 選項啟動原生或 Java 程式設計語言代理程式時。如需詳細資訊，請參閱 `JAVA_TOOL_OPTIONS` 環境變數。
- `NODE_OPTIONS` - 可在 Node.js 執行期中使用。
- `DOTNET_STARTUP_HOOKS` - 在 .NET Core 3.1 及更高版本中，此環境變數指定了 Lambda 可以使用的組件 (dll) 的路徑。

使用特定語言的環境變數是設定啟動屬性的慣用方式。

包裝函式指令碼

您可以建立包裝函數指令碼來自訂 Lambda 函數的執行時間啟動行為。包裝函式指令碼可讓您設定無法透過特定語言環境變數來設定的組態參數。

> **Note**
>
> 如果包裝函式指令碼未成功啟動執行階段程序，調用可能會失敗。

所有原生 Lambda 執行期均支援包裝程式指令碼。僅限作業系統的執行期 (`provided` 執行期系列) 不支援包裝程式指令碼。

當您為函數使用包裝函數指令碼時，Lambda 會使用您的指令碼啟動執行時間。Lambda 會向您的指令碼傳送解譯器的路徑，以及標準執行時間啟動的所有原始引數。您的指令碼可以延伸或轉換程式的啟動行為。例如，指令碼可以插入和更改引數、設定環境變數，或擷取指標、錯誤和其他診斷資訊。

您可以透過將 AWS_LAMBDA_EXEC_WRAPPER 環境變數的值設定為可執行二進位檔案或指令碼的檔案系統路徑來指定指令碼。

範例：建立包裝函數指令碼並用作 Lambda 層

在下面的例子中，您建立一個包裝函式指令碼來啟動與 -X importtime 選項的 Python 解譯器。當您執行函數時，Lambda 會產生日誌項目，以顯示每個匯入的匯入持續時間。

若要建立包裝函數指令碼並用作層

1. 建立層的目錄：

```
mkdir -p python-wrapper-layer/bin
cd python-wrapper-layer/bin
```

2. 在 bin 目錄中，將下列程式碼貼到名為 importtime_wrapper 的新檔案中。這是包裝函數指令碼。

```
#!/bin/bash

# the path to the interpreter and all of the originally intended arguments
args=("$@")

# the extra options to pass to the interpreter
extra_args=("-X" "importtime")

# insert the extra options
args=("${args[@]:0:$#-1}" "${extra_args[@]}" "${args[@]: -1}")

# start the runtime with the extra options
exec "${args[@]}"
```

3. 給予指令碼可執行的許可：

```
chmod +x importtime_wrapper
```

4. 為層建立 .zip 檔案：

```
cd ..
zip -r ../python-wrapper-layer.zip .
```

5. 確認 .zip 檔案具有下列目錄結構：

```
python-wrapper-layer.zip
# bin
    # importtime_wrapper
```

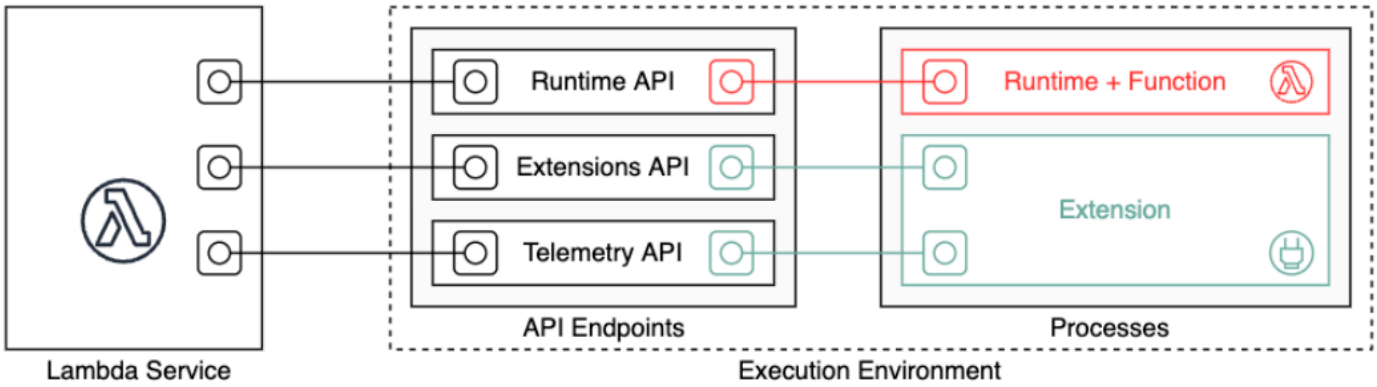
6. 使用 .zip 套件建立層。
7. 使用 Lambda 主控台建立函數。
   a. 開啟 Lambda 主控台。
   b. 選擇 Create function (建立函數)。
   c. 在 函數名稱 中輸入文字。
   d. 針對執行時期，選擇最新支援的 Python 執行時期。
   e. 選擇 Create function (建立函數)。
8. 將圖層新增到您的函式中。
   a. 選擇您的函數，然後選擇程式碼索引標籤 (如果尚未選取)。
   b. 向下捲動至層區段，然後選擇新增層。
   c. 針對層來源，選取自訂層，然後從自訂層下拉式清單中選擇您的層。
   d. 對於版本，選擇 1。
   e. 選擇新增。
9. 新增包裝函數環境變數。
   a. 選擇組態索引標籤，然後選擇環境變數。
   b. 在 Environment variables (環境變數) 下，選擇 Edit (編輯)。
   c. 選擇 Add environment variable (新增環境變數)。
   d. 在 Key (索引鍵) 欄位，輸入 `AWS_LAMBDA_EXEC_WRAPPER`。
   e. 針對值，輸入 `/opt/bin/importtime_wrapper` (`/opt/` + .zip 層的資料夾結構)。
   f. 選擇儲存。
10. 測試包裝函數指令碼。
    a. 選擇 測試 標籤。
    b. 在測試事件下，選擇測試。您不需要建立測試事件 - 預設事件會正常運作。
    c. 向下捲動至日誌輸出。因為您的包裝函式指令碼使用 `-X importtime` 選項啟動 Python 解譯器，所以日誌會顯示每次導入所需的時間。例如：

```
532 |           collections
import time:        63 |         63 |           _functools
import time:      1053 |       3646 |         functools
import time:      2163 |       7499 |       enum
import time:       100 |        100 |         _sre
import time:       446 |        446 |           re._constants
import time:       691 |       1136 |         re._parser
import time:       378 |        378 |         re._casefix
import time:       670 |       2283 |       re._compiler
import time:       416 |        416 |       copyreg
```

針對自訂執行時期使用 Lambda 執行時期 API

AWS Lambda 為自訂執行時間提供 HTTP API，以從 Lambda 接收呼叫事件，並在 Lambda 執行環境中傳送回應資料。本節包含 Lambda 執行時期 API 的 API 參考。

> **Lambda 受管執行個體支援並行請求**
>
> Lambda 受管執行個體使用與 Lambda （預設） 函數相同的執行時間 API。關鍵差異在於受管執行個體可以接受並行請求`/next`，並且`/response`請求達到設定的`AWS_LAMBDA_MAX_CONCURRENCY`限制。這可讓在單一執行環境中同時處理多個調用。如需受管執行個體的詳細資訊，請參閱 了解 Lambda 受管執行個體執行環境。



執行時間 API 版本 2018-06-01 的 OpenAPI 規格可從 runtime-api.zip 中獲得

若要建立 API 請求 URL，執行時間會從 `AWS_LAMBDA_RUNTIME_API` 環境變數中取得 API 端點，新增 API 版本，以及新增所需的資源路徑。

Example請求

```
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next"
```

API 方法

- 下次調用
- 調用回應
- 初始化錯誤
- 調用錯誤

下次調用

路徑 – `/runtime/invocation/next`

方法 – GET

執行時間會傳送此訊息至 Lambda 來請求調用事件。回應內文包含該次調用的承載，此 JSON 文件含有取自函式觸發條件的事件資料。回應標頭包含有關該次調用的額外資料。

回應標頭

- `Lambda-Runtime-Aws-Request-Id` - 請求 ID，它將識別觸發函數調用的請求。

  例如 `8476a536-e9f4-11e8-9739-2dfe598c3fcd`。
- `Lambda-Runtime-Deadline-Ms` - 函數逾時的日期，以 Unix 時間毫秒為單位。

  例如 `1542409706888`。
- `Lambda-Runtime-Invoked-Function-Arn` - 在調用中指定的 Lambda 函數、版本或別名的 ARN。

  例如 `arn:aws:lambda:us-east-2:123456789012:function:custom-runtime`。
- `Lambda-Runtime-Trace-Id` - AWS X-Ray 追蹤標頭。

  例如 `Root=1-5bef4de7-ad49b0e87f6ef6c87fc2e700;Parent=9a9197af755a6419;Sampled=1`。
- `Lambda-Runtime-Client-Context` – 對於來自 AWS Mobile SDK 的調用，則為用戶端應用程式和裝置的資料。
- `Lambda-Runtime-Cognito-Identity` – 對於來自 AWS Mobile SDK 的調用，是有關 Amazon Cognito 身分提供者的資料。

請勿在 GET 請求上設定逾時，因為回應可能會延遲。在 Lambda 引導執行時間與執行時間有可傳回的事件之間，執行時間處理可能會凍結幾秒鐘。

請求 ID 將追蹤 Lambda 內的調用。您可以在傳送回應時使用此值指定調用。

追蹤標頭包含追蹤 ID、父系 ID 和抽樣決策。如果對請求進行抽樣，請求將由 Lambda 或上游服務進行抽樣。執行時間應設定 `_X_AMZN_TRACE_ID` 為標頭的值。X-Ray 開發套件將讀取此項以取得 ID 並決定是否要追蹤請求。

調用回應

路徑 – `/runtime/invocation/AwsRequestId/response`

方法 – POST

在函數執行到完成之後，執行時間會傳送調用回應至 Lambda。若為同步調用，Lambda 會將回應傳送給用戶端。

Example成功請求

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "SUCCESS"
```

初始化錯誤

如果函數傳回錯誤或執行時間在初始化期間遇到錯誤，執行時間會使用此方法將錯誤報告給 Lambda。

路徑 – `/runtime/init/error`

方法 – POST

標頭

`Lambda-Runtime-Function-Error-Type` - 執行時間遇到的錯誤類型。必要：否。

此標頭包含一個字串值。Lambda 可接受任何字串，但我們建議使用格式 <category.reason>。例如：

- Runtime.NoSuchHandler
- Runtime.APIKeyNotFound
- Runtime.ConfigInvalid
- Runtime.UnknownReason

主體參數

`ErrorRequest` - 關於錯誤的資訊。必要：否。

此欄位是具有下列結構的 JSON 物件：

```
{
```

```
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

請注意，Lambda 接受 errorType 的任何值。

下列範例顯示 Lambda 函數錯誤訊息，其中函數無法剖析調用中提供的事件資料。

Example函數錯誤

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

回應內文參數

- StatusResponse – 字串. 狀態信息，隨 202 回應代碼一起傳送。
- ErrorResponse - 其他錯誤資訊，與錯誤回應代碼一起傳送。ErrorResponse 包含錯誤類型和錯誤訊息。

回應代碼

- 202 - 已接受
- 403 - 禁止
- 500 - 容器錯誤。不可復原的狀態。執行時間應立即退出。

Example初始化錯誤請求

```
ERROR="{\"errorMessage\" : \"Failed to load function.\", \"errorType\" :
 \"InvalidFunctionException\"}"
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/init/error" -d "$ERROR" --
header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

調用錯誤

如果函數傳回錯誤或執行時間遇到錯誤，執行時間會使用此方法將錯誤報告給 Lambda。

路徑 – `/runtime/invocation/AwsRequestId/error`

方法 – POST

標頭

`Lambda-Runtime-Function-Error-Type` - 執行時間遇到的錯誤類型。必要：否。

此標頭包含一個字串值。Lambda 可接受任何字串，但我們建議使用格式 <category.reason>。例如：

- Runtime.NoSuchHandler
- Runtime.APIKeyNotFound
- Runtime.ConfigInvalid
- Runtime.UnknownReason

主體參數

`ErrorRequest` - 關於錯誤的資訊。必要：否。

此欄位是具有下列結構的 JSON 物件：

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

請注意，Lambda 接受 `errorType` 的任何值。

下列範例顯示 Lambda 函數錯誤訊息，其中函數無法剖析調用中提供的事件資料。

Example函數錯誤

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

回應內文參數

- `StatusResponse` – 字串. 狀態信息，隨 202 回應代碼一起傳送。

- `ErrorResponse` - 其他錯誤資訊，與錯誤回應代碼一起傳送。ErrorResponse 包含錯誤類型和錯誤訊息。

回應代碼

- 202 - 已接受
- 400 - 錯誤請求
- 403 - 禁止
- 500 - 容器錯誤。不可復原的狀態。執行時間應立即退出。

Example錯誤請求

```
REQUEST_ID=156cb537-e2d4-11e8-9b34-d36013741fb9
ERROR="{\"errorMessage\" : \"Error parsing event data.\", \"errorType\" :
 \"InvalidEventDataException\"}"
curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/error"
 -d "$ERROR" --header "Lambda-Runtime-Function-Error-Type: Unhandled"
```

何時使用 Lambda 的僅限作業系統執行時期

Lambda 為 Java、Python、Node.js、.NET 和 Ruby 提供受管理執行期。若要使用未提供受管理執行期的程式設計語言建立 Lambda 函數，請使用僅限作業系統的執行期 (provided 執行期系列)。僅限作業系統的執行期有三種主要的使用案例：

- 原生ahead-of-time (AOT) 編譯：Go、Rust、Swift 和 C++ 等語言會以原生方式編譯至可執行的二進位檔，不需要專用語言執行時間。這些語言僅需要可在其中執行編譯二進位檔的作業系統環境。您也可以使用僅限 Lambda 作業系統的執行時間來部署以 .NET Native AOT 和 Java GraalVM Native Image 編譯的二進位檔。

  您必須在二進位中包含執行期介面用戶端。執行期介面用戶端會呼叫 針對自訂執行時期使用 Lambda 執行時期 API 來擷取函數調用，然後呼叫您的函數處理常式。Lambda 為 Rust、Go、.NET Native AOT、Swift （實驗性） 和 C++ （實驗性） 提供執行期介面用戶端。

  您必須將二進位檔編譯為適用於 Linux 環境，以及您計劃用於函數的同一指令集架構 (x86_64 或 arm64)。
- 第三方執行時間：您可以使用off-the-shelf執行時間執行 Lambda 函數，例如適用於 PHP 的 Bref。
- 自訂執行期：您可以為 Lambda 不提供受管理執行期的語言或語言版本建置自己的執行期，例如 Node.js 19。如需詳細資訊，請參閱建置 的自訂執行時間 AWS Lambda。這是僅限作業系統的執行期最不常見的使用案例。

Lambda 支援以下僅限作業系統的執行期：

Name	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
僅限作業系統的執行期	`provided.al2023`	Amazon Linux 2023	2029 年 6 月 30 日	2029 年 7 月 31 日	2029 年 8 月 31 日
僅限作業系統的執行期	`provided.al2`	Amazon Linux 2	2026 年 7 月 31 日	2026 年 8 月 31 日	2026 年 9 月 30 日

與 Amazon Linux 2 相比，Amazon Linux 2023 (`provided.al2023`) 執行期具有多項優點，包括更小的部署足跡和更新版本的程式庫，如 `glibc`。

provided.al2023 執行期使用 dnf 而非 yum 做為套件管理工具，後者是 Amazon Linux 2 中的預設套件管理工具。如需 provided.al2023和 之間差異的詳細資訊provided.al2，請參閱 AWS 運算部落格上的 的 Amazon Linux 2023 執行時間簡介 AWS Lambda。

建置 的自訂執行時間 AWS Lambda

您可以使用任何程式設計語言實作 AWS Lambda 執行時間。執行時間是一款程式，它會在調用 Lambda 函數時執行該函數的處理常式方法。您可將執行期納入函數的部署套件或是在層中分發。建立 Lambda 函數時，選擇僅限作業系統的執行期 (provided 執行期系列)。

> **Note**
>
> 建立自訂執行期是一種進階使用案例。如果您想要了解有關編譯成本機二進位檔或使用第三方現成執行期的資訊，請參閱 何時使用 Lambda 的僅限作業系統執行時期。

如需了解自訂執行期部署過程的演練，請參閱 教學課程：建置自訂執行期。

主題

- 要求
- 在自訂執行階段中實作回應串流
- 為 Lambda 受管執行個體建置自訂執行期

要求

自訂執行期時必須完成特定初始化和處理任務。執行期會執行函數的設定程式碼、從環境變數中讀取處理常式名稱，並從 Lambda 執行期 API 中讀取叫用事件。執行時間會將事件資料傳遞至函數處理常式，並將處理常式的回應發佈回 Lambda。

初始化任務

初始化任務是按照函式的每一執行個體各執行一次，以備妥用於處理調用的環境。

- 擷取設定 - 讀取環境變數以取得關於函數和環境的詳細資訊。
  - _HANDLER - 處理常式所在位置，取自函數的組態。標準格式為 *file*.*method*，其中 file 是不含副檔名的檔案名稱，而 method 則是定義於該檔案內的方法或函式的名稱。
  - LAMBDA_TASK_ROOT - 包含函數程式碼的目錄。

- AWS_LAMBDA_RUNTIME_API - 執行時間 API 的主機和連接埠。

  如需可用變數的完整清單，請參閱 定義執行時間環境變數。

- 初始化函數 - 載入處理常式檔案並執行其所包含的任何全域或靜態程式碼。函式應只建立一次靜態資源 (如開發套件用戶端和資料庫連線)，以供多次調用時重複使用。
- 處理錯誤 - 如果發生錯誤，則呼叫初始化錯誤 API 並立即退出。

初始化會計入計費執行時間和逾時。當執行觸發新函數執行個體的初始化時，您可以在記錄和 AWS X-Ray 追蹤中查看初始化時間。

Example log

```
REPORT RequestId: f8ac1208... Init Duration: 48.26 ms   Duration: 237.17 ms   Billed
 Duration: 300 ms   Memory Size: 128 MB   Max Memory Used: 26 MB
```

處理任務

經執行後，執行時間會使用 Lambda 執行時間介面來管理傳入的事件並報告錯誤。初始化任務完成後，執行時間將以迴圈處理傳入的事件。在您的執行時間程式碼中，依序執行下列步驟。

- 取得事件 - 呼叫下次調用 API 以取得下一個事件。回應內文包含事件資料。回應標頭包含請求 ID 及其他資訊。
- 傳播追蹤標頭 - 從 API 回應中的 Lambda-Runtime-Trace-Id 標頭取得 X-Ray 追蹤標頭。使用相同的值在本機設定 _X_AMZN_TRACE_ID 環境變數。X-Ray 開發套件使用這個值來連接服務之間的追蹤資料。
- 建立內容物件 - 建立含有內容資訊的物件，其資訊取自 API 回應中的環境變數和各標頭。
- 調用函數處理常式 - 將事件與內容物件傳遞至處理常式。
- 處理回應 - 呼叫調用回應 API 以發佈處理常式的回應。
- 處理錯誤 - 如果發生錯誤，則呼叫調用錯誤 API。
- 清理 - 釋放未使用的資源、傳送資料至其他服務，或是執行額外的任務後再取得下一個事件。

進入點

自訂執行時間的進入點是名為 bootstrap 的可執行檔。引導檔案可做為執行時間，也可以調用另一個建立執行時間的檔案。如果部署套件的根目錄不包含名為 bootstrap 的檔案，Lambda

會在函數的層中尋找該檔案。若 bootstrap 檔案不存在或不是可執行檔，函數會在調用時傳回 Runtime.InvalidEntrypoint 錯誤。

以下範例 bootstrap 檔案使用隨附的 Node.js 版本，在單獨的檔案 runtime.js 中執行 JavaScript 執行期。

Example引導

```
#!/bin/sh
    cd $LAMBDA_TASK_ROOT
    ./node-v11.1.0-linux-x64/bin/node runtime.js
```

在自訂執行階段中實作回應串流

針對回應串流函數，response 和 error 端點稍微修改了行為，允許執行期將部分回應串流至用戶端，並以區塊形式傳回承載。如需特定行為的詳細資訊，請參閱以下內容：

- /runtime/invocation/AwsRequestId/response - 傳播執行期的 Content-Type 標頭以傳送至用戶端。Lambda 透過 HTTP/1.1 區塊傳輸編碼，以區塊為單位傳回回應承載。若要將回應串流至 Lambda，執行期必須：
  - 將Lambda-Runtime-Function-Response-Mode HTTP 標頭設為 streaming。
  - 將Transfer-Encoding 標頭設為 chunked。
  - 編寫符合 HTTP/1.1 區塊傳輸編碼規範的回應。
  - 成功編寫回應後會關閉基礎連線。
- /runtime/invocation/AwsRequestId/error - 執行期可以使用此端點向 Lambda 報告函數或執行期錯誤，Lambda 也接受 Transfer-Encoding 標頭。只能在執行階段開始傳送叫用回應之前呼叫此端點。
- 在 /runtime/invocation/AwsRequestId/response 中使用錯誤尾端報告串流中間錯誤 - 若要報告在執行期開始編寫叫用回應後出現的錯誤，執行期可選擇連接名為 Lambda-Runtime-Function-Error-Type 和 Lambda-Runtime-Function-Error-Body 的 HTTP 尾端標頭。Lambda 會將此視為成功回應，並將執行期提供的錯誤中繼資料轉送給用戶端。

> **Note**
>
> 若要附加尾端標頭，執行階段必須在 HTTP 要求的開頭設定標頭值 Trailer。這是 HTTP/1.1 區塊傳輸編碼規範的要求。

- `Lambda-Runtime-Function-Error-Type` - 執行期遇到的錯誤類型。此標頭包含一個字串值。Lambda 可接受任何字串，但我們建議使用格式 *<category.reason>*。例如 `Runtime.APIKeyNotFound`。
- `Lambda-Runtime-Function-Error-Body` - 有關錯誤的 Base64 編碼資訊。

為 Lambda 受管執行個體建置自訂執行期

Lambda 受管執行個體使用與 Lambda （預設） 函數相同的執行時間 API。不過，實作自訂執行時間以支援受管執行個體並行執行模型的方式有重大差異。

並行請求處理

為受管執行個體建立自訂執行期的主要差異是支援並行調用。與執行時間一次處理一個調用的 Lambda （預設） 函數不同，受管執行個體可以在單一執行環境中同時處理多個調用。

您的自訂執行時間必須：

- 支援並行**/next**請求 – 執行時間可以對下一個調用 API 進行多個同時呼叫，直到`AWS_LAMBDA_MAX_CONCURRENCY`環境變數指定的限制為止。
- 處理並行**/response**請求 – 多個調用可以同時呼叫調用回應 API。
- 實作執行緒安全請求處理 – 透過正確管理共用資源和狀態，確保並行調用不會互相干擾。
- 使用唯一的請求 IDs – 使用 `Lambda-Runtime-Aws-Request-Id`標頭分別追蹤每個叫用，以比對回應與其對應的請求。

實作模式

受管執行個體執行時間的典型實作模式包括建立工作者執行緒或程序來處理並行調用：

1. 讀取並行限制 – 在初始化時，讀取`AWS_LAMBDA_MAX_CONCURRENCY`環境變數以判斷要支援的並行調用數量。
2. 建立工作者集區 – 初始化等於並行限制的工作者集區 （執行緒、程序或非同步任務）。
3. 工作者處理迴圈 – 每個工作者獨立：
   - 呼叫 `/runtime/invocation/next` 以取得呼叫事件
   - 使用事件資料調用函數處理常式
   - 將回應發佈至 `/runtime/invocation/AwsRequestId/response`

- 重複迴圈

其他考量

- 記錄格式 – 受管執行個體僅支援 JSON 日誌格式。確保您的執行時間遵守AWS_LAMBDA_LOG_FORMAT環境變數，且僅使用 JSON 格式。如需詳細資訊，請參閱設定 JSON 和純文字日誌格式。
- 共用資源 – 使用具有並行調用的 /tmp目錄等共用資源時請小心。實作適當的鎖定機制，以防止競爭條件。

如需 Lambda 受管執行個體執行環境的詳細資訊，請參閱 了解 Lambda 受管執行個體執行環境。

教學課程：建置自訂執行期

在本教學課程中，您將建立具有自訂執行時間的 Lambda 函數。首先，您要將執行時間納入函式的部署套件中。接著再將其遷移到與函式分開而單獨管理的 Layer。最後，您要透過更新該執行時間 Layer 以資源為基礎的許可政策，將其與世界各地的人共享。

必要條件

此教學課程假設您具備基本的 Lambda 操作知識並了解 Lambda 主控台。若您尚未了解，請遵循 使用主控台建立一個 Lambda 函數 中的指示，建立您的第一個 Lambda 函數。

若要完成下列步驟，您需要 AWS CLI 版本 2。命令和預期的輸出會列在不同的區塊中：

```
aws --version
```

您應該會看到下列輸出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

對於長命令，逸出字元 (\) 用於將命令分割為多行。

在 Linux 和 macOS 上，使用您偏好的 shell 和套件軟體管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 zip)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的

Windows 子系統。本指南中的 CLI 命令範例使用 Linux 格式。如果您使用的是 Windows CLI，必須重新格式化包含內嵌 JSON 文件的命令。

您需要 IAM 角色來建立 Lambda 函數。該角色需要許可，才能將日誌傳送至 CloudWatch Logs，並存取您的函數使用的 AWS 服務。如果您還沒有函數開發角色，請立即建立一個。

若要建立執行角色

1. 在 IAM 主控台中開啟角色頁面。
2. 選擇建立角色。
3. 建立具備下列屬性的角色。
   - 信任實體 - Lambda。
   - 許可 - AWSLambdaBasicExecutionRole。
   - 角色名稱 - **lambda-role**。

   AWSLambdaBasicExecutionRole 政策具備函數將日誌寫入到 CloudWatch Logs 時所需的許可。

建立函數

建立具有自訂執行時間的 Lambda 函數。本範例包括兩個檔案：執行期 `bootstrap` 檔案以及函數處理常式。兩個檔案都是以 Bash 實作。

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir runtime-tutorial
cd runtime-tutorial
```

2. 建立稱為 `bootstrap` 的新檔案。這是自訂執行期。

   Example引導

```
#!/bin/sh

set -euo pipefail

# Initialization - load function handler
source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"
```

```
# Processing
while true
do
  HEADERS="$(mktemp)"
  # Get an event. The HTTP request will block until one is received
  EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

  # Extract request ID by scraping response headers received above
  REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Run the handler function from the script
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

  # Send the response
  curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE"
done
```

執行時間將從部署套件載入函式指令碼。其使用了兩個變數以找出指令碼。LAMBDA_TASK_ROOT 將告知套件解壓縮的位置，而 _HANDLER 則包含指令碼的名稱。

在執行期載入函數指令碼之後，它將使用執行期 API 從 Lambda 中擷取調用事件、傳遞事件至處理常式，並將回應發布回 Lambda。為了取得請求 ID，執行時間將 API 回應中的各標頭儲存至暫時檔案，然後從該檔案讀取 Lambda-Runtime-Aws-Request-Id 標頭。

> **Note**
>
> 執行時間另還負責其他任務，包括錯誤處理以及向處理常式提供內容資訊。如需詳細資訊，請參閱 要求。

3. 為函數建立指令碼。以下範例指令碼定義了一個接受事件資料的處理常式函數，會將該資料記錄到 stderr 並予以傳回。

   Example function.sh

```
function handler () {
  EVENT_DATA=$1
  echo "$EVENT_DATA" 1>&2;
```

```
  RESPONSE="Echoing request: '$EVENT_DATA'"

  echo $RESPONSE
}
```

runtime-tutorial 目錄現在應該看起來像這樣：

```
runtime-tutorial
# bootstrap
# function.sh
```

4. 將檔案轉成可執行檔並加入至 .zip 封存檔。這是部署套件。

```
chmod 755 function.sh bootstrap
zip function.zip function.sh bootstrap
```

5. 建立名為的函數 bash-runtime。對於 --role，輸入 Lambda 執行角色的 ARN。

```
aws lambda create-function --function-name bash-runtime \
--zip-file fileb://function.zip --handler function.handler --runtime
 provided.al2023 \
--role arn:aws:iam::123456789012:role/lambda-role
```

6. 調用函數。

```
aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
 response.txt --cli-binary-format raw-in-base64-out
```

如果您使用 AWS CLI 第 2 版，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該看到如下回應：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

7. 確認回應。

```
cat response.txt
```

您應該看到如下回應：

```
Echoing request: '{"text":"Hello"}'
```

建立 Layer

為了將執行時間程式碼與函式程式碼分開，您要建立一個僅包含執行時間的 Layer。Layer 讓您能夠單獨開發函式的依存項目，且若搭配多個函式使用同一 Layer 還可減少儲存空間用量。如需更多詳細資訊，請參閱 使用層管理 Lambda 相依項。

1. 建立包含 `bootstrap` 檔案的 .zip 檔案。

```
zip runtime.zip bootstrap
```

2. 使用 publish-layer-version 命令來建立 layer。

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

如此即建立了 Layer 的第一個版本。

更新函數

若要搭配函式使用執行期 Layer，請將函式設定成使用 Layer，並且移除函式中的執行期程式碼。

1. 更新函式組態以提取 Layer。

```
aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:1
```

這會將執行時間新增至 `/opt` 目錄中的函數。若要確保 Lambda 使用層中的執行期，您必須將 `boostrap` 從函數的部署套件中移除，如接下來兩個步驟所示。

2. 建立包含函數程式碼的 .zip 檔案。

```
zip function-only.zip function.sh
```

3. 更新函式程式碼，使之僅包含處理常式指令碼。

```
aws lambda update-function-code --function-name bash-runtime --zip-file fileb://
function-only.zip
```

4. 調用函數以確認其是否可搭配執行期 layer 運作。

```
aws lambda invoke --function-name bash-runtime --payload '{"text":"Hello"}'
 response.txt --cli-binary-format raw-in-base64-out
```

如果您使用 AWS CLI 第 2 版，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該看到如下回應：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
```

5. 確認回應。

```
cat response.txt
```

您應該看到如下回應：

```
Echoing request: '{"text":"Hello"}'
```

更新執行時間

1. 若要記錄執行環境的相關資訊，請更新執行時間指令碼使其輸出環境變數。

Example引導

```
#!/bin/sh
```

```
set -euo pipefail

# Configure runtime to output environment variables
echo "##  Environment variables:"
env

# Load function handler
source $LAMBDA_TASK_ROOT/"$(echo $_HANDLER | cut -d. -f1).sh"

# Processing
while true
do
  HEADERS="$(mktemp)"
  # Get an event. The HTTP request will block until one is received
  EVENT_DATA=$(curl -sS -LD "$HEADERS" "http://
${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/next")

  # Extract request ID by scraping response headers received above
  REQUEST_ID=$(grep -Fi Lambda-Runtime-Aws-Request-Id "$HEADERS" | tr -d
 '[:space:]' | cut -d: -f2)

  # Run the handler function from the script
  RESPONSE=$($(echo "$_HANDLER" | cut -d. -f2) "$EVENT_DATA")

  # Send the response
  curl "http://${AWS_LAMBDA_RUNTIME_API}/2018-06-01/runtime/invocation/$REQUEST_ID/
response"  -d "$RESPONSE"
done
```

2. 建立包含新版本 `bootstrap` 檔案的 .zip 檔案。

```
zip runtime.zip bootstrap
```

3. 建立 `bash-runtime` 層的新版本。

```
aws lambda publish-layer-version --layer-name bash-runtime --zip-file fileb://
runtime.zip
```

4. 設定函式以使用新版本的 Layer。

```
aws lambda update-function-configuration --function-name bash-runtime \
--layers arn:aws:lambda:us-east-1:123456789012:layer:bash-runtime:2
```

共享 Layer

若要與其他 AWS 帳戶共用層，請將跨帳戶許可陳述式新增至層的資源型政策。執行 add-layer-version-permission 命令，並將帳戶 ID 指定為 `principal`。在各陳述式中，您可將許可授予單一帳戶、所有帳戶或 AWS Organizations 中的某個組織。

以下列範例會授予帳戶 111122223333 存取 `bash-runtime` layer 第 2 版的許可。

```
aws lambda add-layer-version-permission \
  --layer-name bash-runtime \
  --version-number 2 \
  --statement-id xaccount \
  --action lambda:GetLayerVersion \
  --principal 111122223333 \
  --output text
```

您應該會看到類似下列的輸出：

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

許可僅適用於單一層版本。每次建立新的層版本時均需重複此程序。

清除

刪除各個版本的 Layer。

```
aws lambda delete-layer-version --layer-name bash-runtime --version-number 1
aws lambda delete-layer-version --layer-name bash-runtime --version-number 2
```

由於函數持有對該層的第 2 版的參考，所以它仍存在於 Lambda 中。本函式將繼續運作，但無法再設定各函式使用已刪除的版本。若您修改了函數的 layer 清單，則必須指定新的版本或略去已刪除的 layer。

使用 delete-function 命令刪除函數。

```
aws lambda delete-function --function-name bash-runtime
```

開放原始碼儲存庫

AWS Lambda 提供各種開放原始碼工具、程式庫和元件，協助您建置、自訂和最佳化無伺服器應用程式。這些資源包括執行時間界面用戶端、事件程式庫、容器基礎映像、開發工具和範例專案，這些專案由 GitHub 維護 AWS 和提供。透過利用這些開放原始碼儲存庫，您可以擴展 Lambda 的功能、建立自訂執行時間、處理來自各種 AWS 服務的事件，以及深入了解函數的效能。本頁概要介紹支援 Lambda 開發的關鍵開放原始碼專案。

執行時期介面用戶端

Lambda 執行時期介面用戶端 (RIC) 是開放原始碼程式庫，可實作執行時期 API 並管理函式程式碼與 Lambda 服務之間的互動。這些用戶端負責接收調用事件、傳遞內容資訊以及報告錯誤。

Lambda 受管執行時期和容器基礎映像所使用的執行時期介面用戶端，會以開放原始碼形式發布。在建置自訂執行時期或擴展現有的執行時期時，可使用這些開放原始碼程式庫來簡化實作。下列開放原始碼 GitHub 儲存庫包含 Lambda RIC 的原始程式碼：

- Node.js Runtime Interface Client
- Python Runtime Interface Client
- Java Runtime Interface Client
- Ruby Runtime Interface Client
- .NET Runtime Interface Client
- Rust 執行期界面用戶端
- Go Runtime Interface Client
- Swift 執行期界面用戶端（實驗性）
- C++ 執行期界面用戶端（實驗性）
- Lambda Base Images

如需有關使用這些用戶端建置自訂執行時期的詳細資訊，請參閱the section called “建置自訂執行時間”。

事件程式庫

Lambda 事件程式庫提供類型定義和協助程式公用程式，用於處理來自各種 AWS 服務的事件。這些程式庫能協助您以類型安全的方式剖析與處理事件資料，讓您更輕鬆地處理來自 Amazon S3、Amazon DynamoDB 及 Amazon API Gateway 等服務的事件。

對於編譯的語言， AWS 提供下列事件程式庫：

- Java Event Library
- .NET Event Libraries
- Go Event Library
- Rust Event Library

對於 Node.js、Python 和 Ruby 等直譯式語言，事件可以直接剖析為 JSON 物件，無需額外的獨立程式庫。不過，使用 Node.js 和 Python 的開發人員可以利用 的 powertools AWS Lambda，為提供類型提示、資料驗證和功能 AWS 的事件提供內建結構描述，類似於編譯語言程式庫提供的功能。

- Powertools for TypeScript
- Powertools for Python

容器基礎映像

AWS 提供開放原始碼容器基礎映像，您可以用來做為建置 Lambda 函數容器映像的起點。這些基礎映像包括執行時期介面用戶端，以及在 Lambda 執行環境中執行函式所需的其他元件。

如需有關可用基礎映像及其使用方式的詳細資訊，請參閱 AWS Lambda Base Images 儲存庫和the section called “容器映像”。

開發工具

AWS 提供額外的開放原始碼開發工具，協助您建置和最佳化 Lambda 函數：

的 Powertools AWS Lambda

的 Powertools AWS Lambda 使用基本公用程式簡化無伺服器開發，以防止重複處理，以及多記錄處理和 Kafka 消費者程式庫的批次處理。這些功能可協助您將程式碼複雜度與營運負擔降至最低。

您也可以利用內建事件結構描述驗證、結構化記錄和追蹤，以及參數存放區整合，這些整合旨在加速建立生產就緒的 Lambda 函數，同時遵循架構 AWS 良好的最佳實務。

GitHub 儲存庫：

- Python
- TypeScript

- Java
- .NET

Java 開發工具

- Java Profiler (實驗階段)：用於分析 Java Lambda 函式的工具。
- Java 程式庫：此儲存庫彙整了用於 Lambda 開發的完整 Java 程式庫與工具集合，包含 JUnit 測試實用工具、效能分析工具等核心專案。
- 無伺服器 Java 容器：此程式庫能讓您在 Lambda 上執行現有的 Java 應用程式，所需修改極少。

.NET 開發工具

AWS Lambda .NET 儲存庫提供 Lambda 開發的 .NET 程式庫和工具，包括 等關鍵專案，適用於 .NET CLI 和 .NET Core 伺服器用於託管 .NET Core 應用程式 AWS Lambda 的工具。

專案範例

在 Serverless Land 儲存庫中探索 Lambda 範例專案和應用程式的完整集合。這些範例示範了各種 Lambda 使用案例、整合模式與最佳實務，能協助您開始使用無伺服器應用程式。

設定 AWS Lambda 函數

了解如何使用 Lambda API 或主控台來設定 Lambda 函數的核心功能和選項。

[.zip 檔案封存]

當您想要包含相依性、自訂執行時間層或函數程式碼以外的任何檔案時，請建立 Lambda 函數部署套件。部署套件是包含函數程式碼和相依性的 .zip 檔案封存。

[容器映像]

當您需要更多對建置程序的控制，或您的函數需要自訂執行時間組態時，請使用容器映像來封裝函數程式碼和相依性。您可以使用 Docker CLI 等工具，建置、測試和部署 Lambda 函數做為容器映像。

[記憶體]

了解如何以及何時增加函數記憶體。

[暫時性儲存]

了解如何以及何時增加函數的臨時儲存容量。

[Timeout (逾時)]

了解如何以及何時增加函數的逾時值。

[耐用的組態]

讓您的 Lambda 函數在叫用之間維持狀態的同時，可以長時間執行，最長可達一年。設定長時間執行、具狀態工作流程的執行逾時、狀態保留和版本控制行為。耐用的函數可讓您建置複雜的多步驟程序，以暫停、繼續和自動處理中斷。

[環境變數]

您可以讓函數程式碼具備可攜性，並使用環境變量來將它們儲存在函數的組態中，以便保存在程式碼之外。

[傳出網路]

您可以使用 Lambda 函數搭配 Amazon VPC 中的 AWS 資源。將函數連線到 VPC 可讓您在關聯式資料庫和快取等私有子網路中存取資源。

[傳入網路]

您可以使用界面 VPC 端點來叫用您的 Lambda 函數，而不需要透過公有網際網路。

檔案系統

您可以使用 Lambda 函數來將 Amazon EFS 掛載至本機目錄。檔案系統可讓您的函數程式碼安全存取和修改共用資源，並發揮高度並行效能。

別名

您可以將用戶端設定為使用別名來調用特定的 Lambda 函數版本，而非更新用戶端。

版本

發佈您的函數版本，即可將程式碼和組態儲存為無法變更的獨立資源。

Tags (標籤)

使用標籤來啟用屬性型存取控制 (ABAC)、組織 Lambda 函數，以及使用 AWS Cost Explorer 或 AWS Billing and Cost Management 服務篩選和產生函數的報告。

回應串流

您可以設定 Lambda 函數 URL，將回應承載串流回用戶端。透過提高第一個位元組時間 (TTFB) 效能，回應串流有益於延遲敏感應用程式。這是因為您可以在部分回應可用時將其傳回給用戶端。此外，您可以使用回應串流來建置可傳回更大承載的函數。

中繼資料端點

使用 Lambda 中繼資料端點來探索函數正在執行的可用區域，讓您能夠透過路由至相同可用區域資源來最佳化延遲，並實作可用區域感知彈性模式。

以 .zip 封存檔形式部署 Lambda 函數

當您建立 Lambda 函數時，可以將函數程式碼封裝到部署套件中。Lambda 支援兩種類型的部署套件：容器映像和 .zip 封存檔。建立函數的工作流程取決於部署套件類型。若要設定一個定義為容器映像的函數，請參閱the section called “容器映像”。

可使用 Lambda 主控台和 Lambda API 來建立以 .zip 封存檔定義的函數。也可以上傳更新的 .zip 檔案來變更函數程式碼。

> **Note**
>
> 不能變更現有函數的部署套件類型 (.zip 或容器映像)。例如，您不能轉換容器映像函數以使用 .zip 封存檔。您必須建立新的函數。

主題

- 建立函數
- 使用主控台程式碼編輯器
- 更新函數程式碼
- 變更執行階段
- 變更架構
- 使用 Lambda API
- 下載函式程式碼
- CloudFormation
- 加密 Lambda .zip 部署套件

建立函數

當您建立以 .zip 封存檔定義的函數時，您可以選擇函數的程式碼範本、語言版本和執行角色。Lambda 建立函數後，您可以新增函數程式碼。

建立函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇建立函數。

3. 選擇 Author from scratch (從頭開始編寫) 或 Use a blueprint (使用藍圖) 來建立函數。
4. 在 基本資訊 下，請執行下列動作：
   a. 針對 函數名稱，輸入函數名稱。函數名稱的長度限制為 64 個字元。
   b. 對於執行時間，選擇函數要使用的語言版本。
   c. (選用) 對於 Architecture (架構)，選擇要用於函數的指令集架構。預設架構值為 x86_64。為您的函數建置部署套件時，確定它與此指令集架構相容。
5. (選用) 在許可下，展開變更預設執行角色。您可建立新的執行角色，或使用現有的角色。
6. (選用) 展開 Advanced settings (進階設定)。您可對函數選擇程式碼簽署組態。您也可以為函數設定 (Amazon VPC) 進行存取。
7. 選擇建立函數。

Lambda 建立新函數。您現在可以使用主控台來新增函數程式碼，並設定其他函數參數與功能。如需程式碼部署指示，請參閱函數所用執行時間的處理常式頁面。

Node.js

使用 .zip 封存檔部署 Node.js Lambda 函數

Python

使用 .zip 封存檔部署 Python Lambda 函數

Ruby

使用 .zip 封存檔部署 Ruby Lambda 函數

Java

使用 .zip 或 JAR 封存檔部署 Java Lambda 函數

Go

使用 .zip 封存檔部署 Go Lambda 函數

C#

使用 .zip 封存檔建置和部署 C# Lambda 函數

PowerShell

使用 .zip 封存檔部署 PowerShell Lambda 函數

使用主控台程式碼編輯器

主控台將建立具有單一來源檔案的 Lambda 函數。對於指令碼語言，您可以使用內建的程式碼編輯器編輯該檔案並新增更多檔案。選擇 Save (儲存) 以儲存變更。然後，若要執行程式碼，請選擇 Test (測試)。

當您儲存函數程式碼時，Lambda 主控台會建立 .zip 封存檔部署套件。當您在主控台之外開發函數程式碼 (使用 IDE) 時，您需要建立部署套件將您的程式碼上傳到 Lambda 函數。

更新函數程式碼

對於指令碼語言 (Node.js、Python 和 Ruby)，您可以在內嵌的程式碼編輯器中編輯函數程式碼。如果程式碼大於 3MB，或如果您需要新增程式庫，或對於編輯器不支援的語言 (Java、Go、C#)，必須將函數程式碼上傳為 .zip 封存。如果 .zip 封存檔小於 50 MB，您可以從本機電腦上傳 .zip 封存檔。如果此檔案大於 50 MB，請將該檔案從 Amazon S3 儲存貯體上傳至函數。

若要將函數程式碼上傳為 .zip 封存

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要更新的函數並選擇 Code (程式碼) 索引標籤。
3. 在程式碼來源下，選擇上傳自。
4. 選擇 .zip file (.zip 檔案)，然後選擇 Upload (上傳)。
   - 在檔案選擇器中，選取新的映像版本、選擇 Open (開啟)，然後選擇 Save (儲存)。
5. (替代步驟 4) 選擇 Amazon S3 location (Amazon S3 位置)。
   - 在文字方塊中，輸入 .zip 封存檔的 S3 連結 URL，然後選擇 Save (儲存)。

變更執行階段

如果您將函數組態更新為使用新的執行階段，則可能需要更新函數程式碼，才能與新執行階段相容。如果您將函數組態更新為使用不同的執行時間，則必須提供與執行時間和架構相容的新函數程式碼。如需如何為函數程式碼建立部署套件的指示，請參閱函數所使用之執行時間的處理常式頁面。

Node.js 20、Python 3.12、Java 21、.NET 8、Ruby 3.3 和更新的基礎映像是以 Amazon Linux 2023 最小容器映像為基礎。舊版基礎映像使用 Amazon Linux 2。與 Amazon Linux 2 相比，AL2023 具有多項優點，包括更小的部署足跡和更新版本的程式庫，如 `glibc`。如需詳細資訊，請參閱 AWS 運算部落格上的隆重介紹適用於 AWS Lambda的 Amazon Linux 2023 執行期。

變更執行階段

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要更新的函數並選擇 Code (程式碼) 索引標籤。
3. 向下捲動到 Runtime settings (執行時間設定) 區段 (在程式碼編輯器下)。
4. 選擇編輯。
   a. 請在 Runtime (執行階段) 選取執行階段識別符。
   b. 對於 Handler (處理常式)，指定函數的處理常式。
   c. 對於 Architecture (架構)，選擇要用於函數的指令集架構。
5. 選擇儲存。

變更架構

在可以變更指令集架構之前，您需要確保函數的程式碼與目標架構相容。

如果您使用 Node.js、Python 或 Ruby，並在內嵌的編輯器中編輯您的函數程式碼，則現有的程式碼可能無需修改即可執行。

不過，如果您使用 .zip 封存檔部署套件來提供函數程式碼，則必須準備新的 .zip 封存檔，該封存檔會針對目標執行時間和指令集架構正確地進行編譯和建置。如需指示，請參閱函數執行時間的處理常式頁面。

變更指令集架構

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要更新的函數並選擇 Code (程式碼) 索引標籤。
3. 在 Runtime settings (執行時間設定) 中，選擇 Edit (編輯)。
4. 對於 Architecture (架構)，選擇要用於函數的指令集架構。
5. 選擇儲存。

使用 Lambda API

若要建立及設定使用 .zip 封存檔的函數，請使用下列 API 操作：

- CreateFunction

- UpdateFunctionCode
- UpdateFunctionConfiguration

下載函式程式碼

您可以透過 Lambda 主控台下載函式程式碼 .zip 的目前未發布 ($LATEST) 版本。若要執行此操作，請先確定您具有下列 IAM 許可：

- `iam:GetPolicy`
- `iam:GetPolicyVersion`
- `iam:GetRole`
- `iam:GetRolePolicy`
- `iam:ListAttachedRolePolicies`
- `iam:ListRolePolicies`
- `iam:ListRoles`

若要下載函式程式碼 .zip

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要下載其函式程式碼 .zip 的函式。
3. 在函式概觀頁面中，點選下載按鈕，接著選擇下載函式程式碼 .zip。
    - 或者，根據函數的組態，選擇下載 AWS SAM 檔案以產生和下載 SAM 範本。您也可以選擇下載兩者，同時下載 .zip 與 SAM 範本。

CloudFormation

您可以使用 CloudFormation 建立使用 .zip 檔案封存的 Lambda 函數。在您的 CloudFormation 範本中，`AWS::Lambda::Function`資源會指定 Lambda 函數。如需 `AWS::Lambda::Function` 資源中的屬性說明，請參閱 AWS CloudFormation 使用者指南中的 AWS::Lambda::Function。

在 `AWS::Lambda::Function` 資源中，設定下列屬性，以建立定義為 .zip 封存檔的函數：

- AWS::Lambda::Function
    - PackageType - 設定為 `Zip`。

- 程式碼 — 在 S3Bucket 和 S3Key 欄位中輸入 Amazon S3 儲存貯體名稱和 .zip 檔案名稱。對於 Node.js 或 Python，您可以提供 Lambda 函數的內嵌原始碼。
- 執行時間 — 設定執行時間值。
- 架構 – 將架構值設定為 arm64 以使用 AWS Graviton2 處理器。依預設，架構值為 x86_64。

加密 Lambda .zip 部署套件

Lambda 一般使用 AWS KMS key為 .zip 部署套件和函數組態詳細資訊提供靜態伺服器端加密。預設情況下，Lambda 使用 AWS 擁有的金鑰。如果此預設行為符合您的工作流程，您不需要設定任何其他項目。 AWS 不會向您收取使用此金鑰的費用。

如果您願意，您可以改為提供 AWS KMS 客戶受管金鑰。您可以這樣做以控制 KMS 金鑰的輪換或滿足您的組織對管理 KMS 金鑰的請求。當您使用客戶受管的金鑰時，只有您帳戶中具有 KMS 金鑰存取權的使用者才能檢視或管理函數的程式碼或組態。

客戶受管金鑰會產生標準 AWS KMS 費用。如需詳細資訊，請參閱 AWS Key Management Service 定價。

建立客戶自管金鑰

您可以使用 AWS 管理主控台或 AWS KMS APIs 來建立對稱客戶受管金鑰。

建立對稱客戶受管金鑰

請依照《AWS Key Management Service 開發人員指南》中的建立對稱加密，建立對稱 KMS 金鑰步驟執行。

許可

金鑰政策

金鑰政策會控制客戶受管金鑰的存取權限。每個客戶受管金鑰都必須只有一個金鑰政策，其中包含決定誰可以使用金鑰及其使用方式的陳述式。如需詳細資訊，請參閱《AWS Key Management Service 開發人員指南》中的如何變更金鑰政策。

使用客戶受管金鑰來加密 .zip 部署套件時，Lambda 不會為金鑰新增授權。相反地，您的 AWS KMS 金鑰政策必須允許 Lambda 代表您呼叫下列 AWS KMS API 操作：

- kms:GenerateDataKey
- kms:Decrypt

下列範例金鑰政策允許帳戶 111122223333 中的所有 Lambda 函數呼叫指定客戶受管金鑰的必要 AWS KMS 操作：

Example AWS KMS 金鑰政策

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.amazonaws.com"
            },
            "Action": [
                "kms:GenerateDataKey",
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:us-east-1:111122223333:key/key-id",
            "Condition": {
                "StringLike": {
                "kms:EncryptionContext:aws:lambda:FunctionArn":
 "arn:aws:lambda:us-east-1:111122223333:function:*"
                }
            }
        }
    ]
}
```

如需有關故障診斷金鑰存取的詳細資訊，請參閱《AWS Key Management Service 開發人員指南》。

主體許可

當您使用客戶受管金鑰來加密 .zip 部署套件時，只有可存取該金鑰的主體才能存取 .zip 部署套件。例如，無法存取客戶受管金鑰的主體無法使用包含在 GetFunction 回應中的預先簽章 S3 URL 下載 .zip 套件。回應的 `Code` 區段中會傳回 `AccessDeniedException`。

Example AWS KMS AccessDeniedException

```
{
```

```
    "Code": {
        "RepositoryType": "S3",
        "Error": {
            "ErrorCode": "AccessDeniedException",
            "Message": "KMS access is denied. Check your KMS permissions. KMS
 Exception: AccessDeniedException KMS Message: User: arn:aws:sts::111122223333:assumed-
role/LambdaTestRole/session is not authorized to perform: kms:Decrypt on resource:
 arn:aws:kms:us-east-1:111122223333:key/key-id with an explicit deny in a resource-
based policy"
        },
        "SourceKMSKeyArn": "arn:aws:kms:us-east-1:111122223333:key/key-id"
    },
 ...
```

如需 AWS KMS 金鑰許可的詳細資訊，請參閱 的身分驗證和存取控制 AWS KMS。

將客戶受管金鑰用於 .zip 部署套件

使用下列 API 參數來設定 .zip 部署套件的客戶受管金鑰：

- SourceKMSKeyArn：加密來源 .zip 部署套件 (您上傳的檔案)。
- KMSKeyArn：加密環境變數和 Lambda SnapStart 快照。

指定 SourceKMSKeyArn 和 KMSKeyArn 時，Lambda 會使用 KMSKeyArn 金鑰來加密 Lambda 用來調用函數的解壓縮套件版本。當已指定 SourceKMSKeyArn 但未指定 KMSKeyArn 時，Lambda 會使用 AWS 受管金鑰 來加密套件的解壓縮版本。

Lambda console

若要在建立函數時新增客戶受管金鑰加密

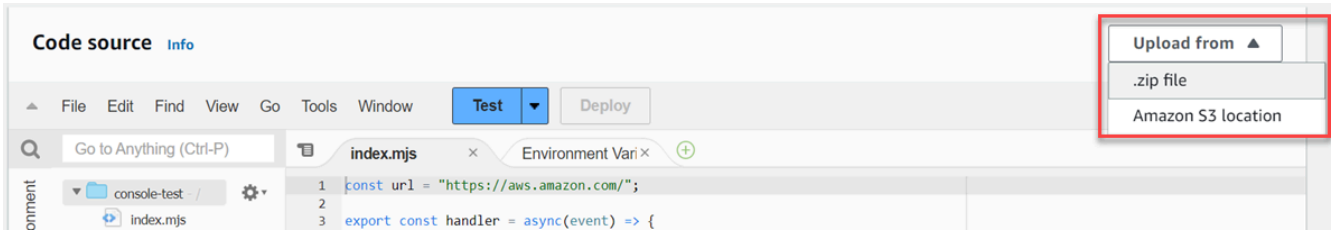
1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇建立函數。
3. 選擇 Author from scratch (從頭開始撰寫) 或 Container image (容器映像)。
4. 在 基本資訊 下，請執行下列動作：
   a. 針對 函數名稱，輸入函數名稱。
   b. 對於執行時間，選擇函數要使用的語言版本。
5. 展開進階設定，然後選取使用 AWS KMS 客戶受管金鑰啟用加密。

6. 選擇客戶受管金鑰。
7. 選擇建立函數。

若要移除客戶受管金鑰加密，或使用不同的金鑰，必須再次上傳 .zip 部署套件。

若要將客戶受管金鑰加密新增至現有函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 .zip 檔案或 Amazon S3 位置。



5. 上傳檔案或輸入 Amazon S3 位置。
6. 選擇使用 AWS KMS 客戶受管金鑰啟用加密。
7. 選擇客戶受管金鑰。
8. 選擇儲存。

AWS CLI

若要在建立函數時新增客戶受管金鑰加密

在下列 create-function 範例中：

- `--code`：指定 .zip 部署套件的本機路徑 (`ZipFile`) 和客戶受管金鑰來加密它 (`SourceKMSKeyArn`)。
- `--kms-key-arn`：指定客戶受管金鑰，以加密環境變數和部署套件的解壓縮版本。

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
```

```
  --code ZipFile=fileb://myFunction.zip,SourceKMSKeyArn=arn:aws:kms:us-
east-1:111122223333:key/key-id \
  --kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key2-id
```

在下列 create-function 範例中：

- `--code`：指定 Amazon S3 儲存貯體 (S3Bucket、S3Key、S3ObjectVersion) 中 .zip 檔案的位置，以及用於加密檔案的客戶受管金鑰 ()SourceKMSKeyArn。
- `--kms-key-arn`：指定客戶受管金鑰，以加密環境變數和部署套件的解壓縮版本。

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x --handler index.handler \
  --role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
  --code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion,SourceKMSKeyArn=arn:aws:kms:us-
east-1:111122223333:key/key-id \
  --kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key2-id
```

若要將客戶受管金鑰加密新增至現有函數

在下列 update-function-code 範例中：

- `--zip-file`：指定 .zip 部署套件的本機路徑。
- `--source-kms-key-arn`：指定客戶受管金鑰，以加密部署套件的壓縮版本。Lambda 使用 AWS 擁有的金鑰來加密用於函式調用的解壓縮套件。如果您想要使用客戶受管金鑰來加密套件的解壓縮版本，請使用 `--kms-key-arn` 選項執行 update-function-configuration 命令。

```
aws lambda update-function-code \
  --function-name myFunction \
  --zip-file fileb://myFunction.zip \
  --source-kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key-id
```

在下列 update-function-code 範例中：

- `--s3-bucket`：指定 Amazon S3 儲存貯體中 .zip 檔案的位置。
- `--s3-key`：指定部署套件的 Amazon S3 金鑰。
- `--s3-object-version`：對於版本控制的物件，要使用的部署套件物件版本。

- `--source-kms-key-arn`：指定客戶受管金鑰，以加密部署套件的壓縮版本。Lambda 使用 AWS 擁有的金鑰來加密用於函式調用的解壓縮套件。如果您想要使用客戶受管金鑰來加密套件的解壓縮版本，請使用 `--kms-key-arn` 選項執行 update-function-configuration 命令。

```
aws lambda update-function-code \
  --function-name myFunction \
  --s3-bucket amzn-s3-demo-bucket \
  --s3-key myFileName.zip \
  --s3-object-version myObject Version
  --source-kms-key-arn arn:aws:kms:us-east-1:111122223333:key/key-id
```

若要從現有函數中移除客戶受管金鑰加密

在下列 update-function-code 範例中，`--zip-file`指定 .zip 部署套件的本機路徑。當您在沒有`--source-kms-key-arn`選項的情況下執行此命令時，Lambda 會使用 AWS 擁有的金鑰來加密部署套件的壓縮版本。

```
aws lambda update-function-code \
  --function-name myFunction \
  --zip-file fileb://myFunction.zip
```

使用容器映像建立 Lambda 函數

AWS Lambda 函數的程式碼包含指令碼或編譯的程式及其相依性。使用部署套件將函數程式碼部署到 Lambda。Lambda 支援兩種類型的部署套件：容器映像和 .zip 封存檔。

您可以透過三種方式為 Lambda 函數建置容器映像：

- 使用 Lambda AWS 的基礎映像

  AWS 基礎映像會預先載入語言執行期、用來管理 Lambda 與函數程式碼之間互動的執行期界面用戶端，以及用於本機測試的執行期界面模擬器。

- 使用僅限 AWS 作業系統的基礎映像

  AWS 僅限作業系統的基礎映像包含 Amazon Linux 發行版本和執行時間界面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像檔與 Lambda 相容，您必須在映像中加入您語言的 執行期介面用戶端。

- 使用非AWS 基礎映像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像檔與 Lambda 相容，您必須在映像中加入您語言的 執行期介面用戶端。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

若要使用容器映像建立 Lambda 函數，請在本機建置映像，然後將映像上傳到 Amazon Elastic Container Registry (Amazon ECR) 儲存庫。若您使用 AWS Marketplace 賣方提供的容器映像，需先將映像複製至私有 Amazon ECR 儲存庫。接著，請在建立函數時指定儲存庫 URI。Amazon ECR 儲存庫必須與 Lambda 函數位於相同的 AWS 區域 中。只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像來建立函數。如需詳細資訊，請參閱Amazon ECR 跨帳戶許可。

> **Note**
>
> Lambda 的容器映像不支援 Amazon ECR FIPS 端點。如果儲存庫 URI 包含 `ecr-fips`，則表示您正在使用 FIPS 端點。範例：`111122223333.dkr.ecr-fips.us-east-1.amazonaws.com`。

本頁面會說明建立 Lambda 相容容器映像檔的基礎映像類型和要求。

> **Note**
>
> 不能變更現有函數的部署套件類型 (.zip 或容器映像)。例如，您不能轉換容器映像函數以使用 .zip 封存檔。您必須建立新的函數。

主題

- 要求
- 使用 Lambda AWS 的基礎映像
- 使用僅限 AWS 作業系統的基礎映像
- 使用非AWS 基礎映像
- 執行時期介面用戶端
- Amazon ECR 許可
- 函數生命週期

要求

安裝 AWS CLI 版本 2 和 Docker CLI。此外也請注意下列請求：

- 該容器映像必須實作 針對自訂執行時期使用 Lambda 執行時期 API。 AWS 開放原始碼 執行期界面用戶端會實作 API。您可以將執行期介面用戶端新增至您的偏好基礎映像中，以使其與 Lambda 相容。
- 容器映像必須能夠在僅唯讀檔案系統上執行。您的函數程式碼可以存取具有 512 MB 和 10,240 MB 儲存空間的可寫入 `/tmp` 目錄，增量為 1 MB。
- 預設 Lambda 使用者必須能夠讀取執行函數程式碼所需的所有檔案。Lambda 透過定義具有最低權限許可的預設 Linux 使用者來遵守安全最佳實務。這表示您不需要在 Dockerfile 中指定 USER。確認您的應用程式的程式碼不依賴其他 Linux 使用者無法執行的檔案。

- Lambda 僅支援 Linux 容器映像。
- Lambda 會提供多架構基礎映像。不過，您為函數建置的映像必須只以其中一個架構為目標。Lambda 不支援使用多架構容器映像的函數。

使用 Lambda AWS 的基礎映像

您可以使用 Lambda 的其中一個 AWS 基礎映像 來建置函數程式碼的容器映像。基礎映像會預先載入語言執行期，以及在 Lambda 上執行容器映像所需的其他元件。您可以將函數程式碼和相依項新增至基礎映像，然後將其封裝為容器映像。

AWS 會定期更新 Lambda AWS 的基礎映像。如果您的 Dockerfile 包含 FROM 屬性的映像名稱，則您的 Docker 用戶端會從 Amazon ECR 儲存庫 提取最新版本的映像。若要使用更新的基礎映像，必須重建容器映像並更新函數程式碼。

Node.js 20、Python 3.12、Java 21、.NET 8、Ruby 3.3 及更新版本的基礎映像，均以 Amazon Linux 2023 最小容器映像為基礎。舊版基礎映像使用 Amazon Linux 2。與 Amazon Linux 2 相比，AL2023 具有多項優點，包括更小的部署足跡和更新版本的程式庫，如 `glibc`。

以 AL2023 為基礎的映像使用 `microdnf` (符號連結為 `dnf`) 而不是 `yum` 作為套件管理工具，後者是 Amazon Linux 2 中的預設套件管理工具。`microdnf` 是 `dnf` 的獨立實作。對於以 AL2023 為基礎的映像中包含的套件清單，請參閱 Comparing packages installed on Amazon Linux 2023 Container Images 中的 Minimal Container 欄。如需 AL2023 和 Amazon Linux 2 之間差異的詳細資訊，請參閱 AWS 運算部落格上的 Introducing the Amazon Linux 2023 runtime for AWS Lambda。

> **Note**
>
> 若要在本機執行AL2023-based映像，包括搭配 AWS Serverless Application Model (AWS SAM)，您必須使用 Docker 20.10.10 版或更新版本。

若要使用 AWS 基礎映像建置容器映像，請選擇您慣用語言的說明：

- Node.js
- TypeScript(使用 Node.js 的基礎映像)
- Python
- Java
- Go

- .NET
- Ruby

使用僅限 AWS 作業系統的基礎映像

AWS 僅限作業系統的基礎映像包含 Amazon Linux 發行版本和執行時間界面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像檔與 Lambda 相容，您必須在映像中加入您語言的 執行期介面用戶端。

Tags (標籤)	執行期	作業系統	Dockerfile	棄用
al2023	僅限作業系統的執行期	Amazon Linux 2023	GitHub 上僅限作業系統之執行期的 Dockerfile	2029 年 6 月 30 日
al2	僅限作業系統的執行期	Amazon Linux 2	GitHub 上僅限作業系統之執行期的 Dockerfile	2026 年 7 月 31 日

Amazon Elastic Container Registry 公有資源庫：gallery.ecr.aws/lambda/provided

使用非AWS 基礎映像

Lambda 支援符合下列其中一種映像資訊清單格式的任何映像：

- Docker 映像資訊清單 V2，結構描述 2 (需搭配 1.10 或更新版本的 Docker 使用)
- 開放容器計劃 (OCI) 規範 (v1.0.0 及以上版本)

Lambda 支援的未壓縮影像大小上限為 10 GB，包括所有圖層。

> **Note**
>
> - 若要使映像檔與 Lambda 相容，您必須在映像中加入您語言的 執行期介面用戶端。
> - 為取得最佳效能，建議將映像資訊清單大小保持在 25,400 位元組以下。若要縮小映像資訊清單大小，建議盡可能減少映像中的層數量與註釋內容。

執行時期介面用戶端

如果您使用僅限作業系統的基礎映像或替代的基礎映像，則必須在映像中加入執行期介面用戶端。執行時間界面用戶端必須擴展 針對自訂執行時期使用 Lambda 執行時期 API，以管理 Lambda 與函數程式碼之間的互動。 為下列語言 AWS 提供開放原始碼執行時間界面用戶端：

- Node.js
- Python
- Java
- .NET
- Go
- Ruby
- Rust –

如果您使用的語言沒有 AWS提供的執行時間界面用戶端，您必須建立自己的 。

Amazon ECR 許可

使用容器映像建立 Lambda 函數之前，您必須先在本機建置映像，並將其上傳至 Amazon ECR 儲存庫。建立函數時，請指定 Amazon ECR 儲存庫 URI。

確定建立函數之使用者或角色的許可包含 `GetRepositoryPolicy`、`BatchGetImage`、`SetRepositoryPolicy`和 `GetDownloadUrlForLayer`。

例如，使用 IAM 主控台建立具有下列政策的角色：

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "VisualEditor0",
      "Effect": "Allow",
      "Action": [
        "ecr:SetRepositoryPolicy",
        "ecr:GetRepositoryPolicy",
        "ecr:BatchGetImage",
```

```
        "ecr:GetDownloadUrlForLayer"
      ],
      "Resource": "arn:aws:ecr:us-east-1:111122223333:repository/hello-world"
    }
  ]
}
```

Amazon ECR 儲存庫政策

若為與 Amazon ECR 中的容器映像位於相同帳戶中的函數，您可以將 `ecr:BatchGetImage` 和 `ecr:GetDownloadUrlForLayer` 許可新增至 Amazon ECR 儲存庫政策。以下範例顯示最低政策：

```
{
        "Sid": "LambdaECRImageRetrievalPolicy",
        "Effect": "Allow",
        "Principal": {
          "Service": "lambda.amazonaws.com"
        },
        "Action": [
          "ecr:BatchGetImage",
          "ecr:GetDownloadUrlForLayer"
        ]
    }
```

如需 Amazon ECR 儲存庫許可的詳細資訊，請參閱《Amazon Elastic Container Registry 使用者指南》中的私有儲存庫政策。

如果 Amazon ECR 儲存庫不包含這些許可，Lambda 會嘗試自動新增這些許可。只有當呼叫 Lambda 的主體擁有 `ecr:getRepositoryPolicy` 和 `ecr:setRepositoryPolicy` 許可時，Lambda 才會新增許可。

若要檢視或編輯 Amazon ECR 儲存庫許可，請遵循《Amazon Elastic Container Registry 使用指南》中設定私有儲存庫政策聲明中的指示。

Amazon ECR 跨帳戶許可

相同區域中的不同帳戶可以建立使用您帳戶擁有的容器映像之函數。在下列範例中，您的 Amazon ECR 儲存庫許可政策需要下列陳述式才能為帳戶編號 123456789012 授予存取權。

- CrossAccountPermission – 允許帳戶 123456789012 建立和更新使用此 ECR 儲存庫中映像的 Lambda 函數。

- LambdaECRImageCrossAccountRetrievalPolicy – 如果很長一段時間內未被調用，Lambda 最終會將函數的狀態設定為非作用中。必須提供此陳述式，讓 Lambda 可以擷取容器映像來最佳化，並代表 123456789012 所擁有的函數進行快取。

Example- 將跨帳戶許可新增至儲存庫

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "CrossAccountPermission",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "AWS": "arn:aws:iam::123456789012:root"
      },
      "Resource": "arn:aws:ecr:us-east-1:123456789012:repository/example-lambda-repository"
    },
    {
      "Sid": "LambdaECRImageCrossAccountRetrievalPolicy",
      "Effect": "Allow",
      "Action": [
        "ecr:BatchGetImage",
        "ecr:GetDownloadUrlForLayer"
      ],
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Condition": {
        "ArnLike": {
          "aws:sourceARN": "arn:aws:lambda:us-east-1:123456789012:function:*"
        }
      },
      "Resource": "arn:aws:ecr:us-east-1:123456789012:repository/example-lambda-repository"
    }
```

```
  ]
}
```

若要授與多個帳戶的存取權，您需將帳戶 ID 新增至 `CrossAccountPermission` 政策中的主體清單，也要新增至 `LambdaECRImageCrossAccountRetrievalPolicy` 中的條件評估清單。

如果您在 AWS 組織中使用多個帳戶，建議您列舉 ECR 許可政策中的每個帳戶 ID。此方法符合在 IAM 政策中設定窄許可 AWS 的安全性最佳實務。

除了 Lambda 許可之外，建立函數的使用者或角色還必須具有 `BatchGetImage` 和 `GetDownloadUrlForLayer` 許可。

函數生命週期

上傳新增或更新的容器映像之後，Lambda 會先最佳化該映像，函數才能處理呼叫。最佳化程序可能需要幾秒鐘。該函數會保持`Pending`狀態，直至程序完成，而後狀態會轉變為`Active`。在函數達到`Active`狀態之前，您無法調用函數。

如果函數在多個星期未被叫用，Lambda 會回收其最佳化版本，並將函數轉換為 `Inactive` 狀態。若要重新啟用函數，您必須叫用它。Lambda 拒絕第一次叫用，並且該函數進入 `Pending` 狀態，直到 Lambda 重新最佳化映像。函數隨後會傳回 `Active` 狀態。

Lambda 會定期從 Amazon ECR 儲存庫中擷取關聯的容器映像。如果對應的容器映像不再存在於 Amazon ECR 或是已撤銷許可，該函數會進入 `Failed` 狀態，並且針對任何函數叫用 Lambda 都會傳回失敗。

您可以使用 Lambda API 來取得函數狀態的相關資訊。如需詳細資訊，請參閱Lambda 函數狀態。

設定 Lambda 函數記憶體

Lambda 會按設定的記憶體數量比例來配置 CPU 功率。記憶體是可供 Lambda 函數在執行時間使用的記憶體數量。可使用記憶體 設定來增加或減少配置給函數的記憶體和 CPU 功率。您可以設定介於 128 MB 到 10,240 MB 之間的記憶體，增量為 1 MB。在 1,769 MB，函數等於一個完整 vCPU (每秒一個 vCPU 秒的額度)。

此頁面說明如何及何時更新 Lambda 函數的記憶體設定。

章節

- 判斷 Lambda 函數的適當記憶體設定
- 設定函數記憶體 (主控台)
- 設定函數記憶體 (AWS CLI)
- 設定函數記憶體 (AWS SAM)
- 接受函數記憶體建議 (主控台)

判斷 Lambda 函數的適當記憶體設定

記憶體是控制函數效能的主要手段。預設設定為 128 MB，這是可能的最低設定。建議僅將 128 MB 用於簡單的 Lambda 函數，例如轉換事件並將其路由至其他 AWS 服務的函數。更高的記憶體配置可以提高使用匯入程式庫、Lambda 層、Amazon Simple Storage Service (Amazon S3) 或 Amazon Elastic File System (Amazon EFS) 的函數效能。按比例增加更多記憶體會增加 CPU 容量，從而提高可用的整體運算能力。如果函數受到 CPU、網路或記憶體限制，則增加記憶體設定可以大幅改善其效能。

若要找到合適的記憶體組態，請透過 Amazon CloudWatch 監控您的函式，並在記憶體耗用量即將達到設定上限時設定警示。這有助於識別記憶體受限函數。對於 CPU 密集型與 IO 密集型函式，監控持續時間也可以提供有價值的洞察。在這些情況下，增加記憶體有助於解決運算或網路瓶頸。

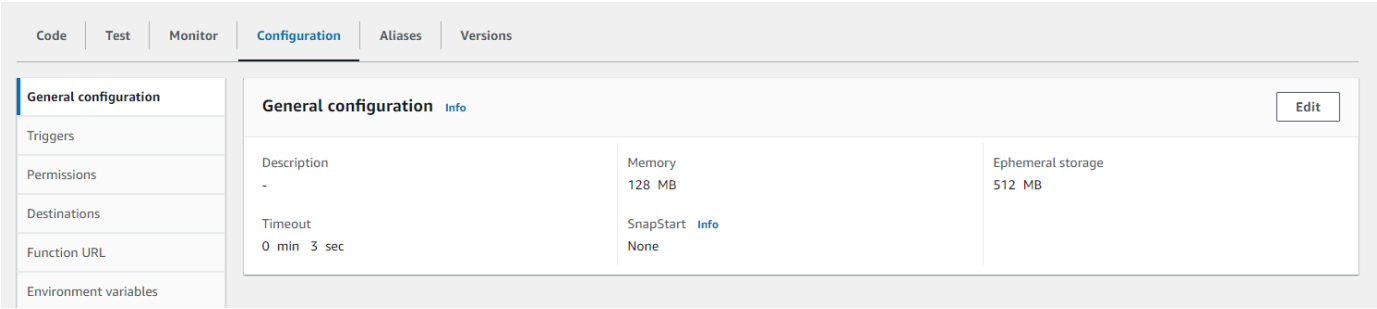
您也可以考慮使用開放原始碼 AWS Lambda Power Tuning 工具。此工具使用 AWS Step Functions 以不同的記憶體配置執行 Lambda 函數的多個並行版本，並測量效能。輸入函數會在您的帳戶中執行 AWS ，執行即時 HTTP 呼叫和 SDK 互動，以測量即時生產案例中可能的效能。也可以實作 CI/CD 流程，使用此工具自動測量您部署的新函數效能。

設定函數記憶體 (主控台)

您可以在 Lambda 主控台中設定函數的記憶體。

更新函數記憶體

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態索引標簽，然後選擇一般組態。



4. 在一般組態中，選擇編輯。
5. 對於記憶體，設定從 128 MB 到 10,240 MB 的值。
6. 選擇儲存。

設定函數記憶體 (AWS CLI)

可以使用 update-function-configuration 命令來設定函數的記憶體。

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --memory-size 1024
```

設定函數記憶體 (AWS SAM)

可以使用 AWS Serverless Application Model 來設定函數的記憶體。更新 `template.yaml` 檔案中的 MemorySize 屬性，然後執行 sam 部署。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
```

```
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 1024
      # Other function properties...
```

接受函數記憶體建議 (主控台)

如果您在 AWS Identity and Access Management (IAM) 中有管理員許可，您可以選擇接收來自 的 Lambda 函數記憶體設定建議 AWS Compute Optimizer。如需有關針對您的帳戶或組織選擇加入記憶體建議的指示，請參閱 AWS Compute Optimizer 使用者指南中的選擇加入您的帳戶。

> **Note**
>
> Compute Optimizer 只支援使用 x86_64 架構的函數。

當您已選擇加入且 Lambda 函數符合 Compute Optimizer 需求時，您可以在一般組態中檢視並接受來自 Lambda 主控台 Compute Optimizer 的函數記憶體建議。

設定 Lambda 函數的暫時性儲存

Lambda 在 /tmp 目錄中為函數提供暫時性儲存。此儲存體是臨時的且對每個執行環境都是唯一的。可以使用暫時性儲存設定來控制分配給函數的暫時性儲存量。您可以設定介於 512 MB 到 10,240 MB 之間的暫時性儲存，以 1 MB 為單位。存放在 /tmp 中的所有資料都會使用 AWS管理的金鑰進行靜態加密。

此頁面說明常見使用案例，以及如何更新 Lambda 函數的暫時性儲存。

章節

- 增加暫時性儲存的常見使用案例
- 配置暫時性儲存（主控台）
- 設定暫時性儲存 (AWS CLI)
- 設定暫時性儲存 (AWS SAM)

增加暫時性儲存的常見使用案例

以下是受益於增加的暫時性儲存的幾個常見使用案例：

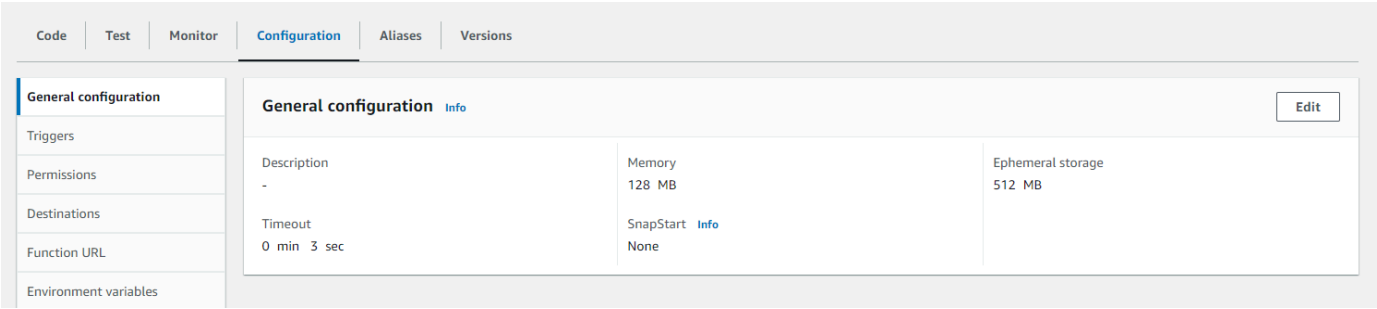
- Extract-transform-load (ETL) 任務：當程式碼執行中繼運算或下載其他資源以完成處理時，增加暫時性儲存。更多臨時空間可讓更複雜的 ETL 任務在 Lambda 函數中執行。
- 機器學習 (ML) 推論：許多推論任務依賴大型參考資料檔案，包括程式庫和模型。有了更多暫時性儲存，可以將更大的模型從 Amazon Simple Storage Service (Amazon S3) 下載到 /tmp，並在處理中使用它們。
- 資料處理：對於從 Amazon S3 下載物件以回應 S3 事件的工作負載，更多的 /tmp 空間可讓您處理較大的物件，而無需使用記憶體內處理。建立 PDF 或程序媒體的工作負載也會受益於更多暫時性儲存。
- 圖形處理：映像處理是基於 Lambda 的應用程式的常見使用案例。對於處理大型 TIFF 檔案或衛星影像的工作負載，更多的暫時性儲存可讓您更輕鬆地使用程式庫，並在 Lambda 中執行運算。

配置暫時性儲存（主控台）

可以在 Lambda 主控台中設定暫時性儲存。

若要修改函數的暫時性儲存

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態索引標簽，然後選擇一般組態。



4. 在一般組態中，選擇編輯。
5. 對於暫時性儲存，設定介於 512 MB 到 10,240 MB 之間的值，以 1 MB 為單位。
6. 選擇儲存。

設定暫時性儲存 (AWS CLI)

可以使用 update-function-configuration 命令來設定暫時性儲存。

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --ephemeral-storage '{"Size": 1024}'
```

設定暫時性儲存 (AWS SAM)

可以使用 AWS Serverless Application Model 來設定函數的暫時性儲存。更新 `template.yaml` 檔案中的 EphemeralStorage 屬性，然後執行 sam 部署。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
```

```
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs22.x
      Architectures:
        - x86_64
      EphemeralStorage:
        Size: 10240
      # Other function properties...
```

選取和設定 Lambda 函數的指令集架構

Lambda 函數的指令集架構會決定 Lambda 用來執行函數的電腦處理器類型。Lambda 提供了指令集架構的選擇：

- arm64 – AWS Graviton2 處理器的 64 位元 ARM 架構。
- x86_64 - 64 位元 x86 架構，適用於 x86 處理器。

> **Note**
>
> arm64 架構在大多數情況下都可用 AWS 區域。如需詳細資訊，請參閱AWS Lambda 定價。在記憶體價格表格中，選擇 Arm Price 索引標籤，然後開啟區域下拉式清單，以查看哪個 AWS 區域 支援搭配 Lambda 的 arm64。
> 如需如何使用 arm64 架構建立函數的範例，請參閱 AWS Lambda 函數 Powered by AWS Graviton2 Processor。

主題

- 使用 arm64 架構的優點
- 遷移到 arm64 架構的要求
- 函數程式碼與 arm64 架構的相容性
- 如何遷移到 arm64 架構
- 設定指令集架構

使用 arm64 架構的優點

與在 x86_64 架構上執行的同等函數相比，使用 arm64 架構 (AWS Graviton2 處理器） 的 Lambda 函數可以實現更高的價格和效能。考慮將 arm64 用於運算密集的應用程式，例如高效能運算、視訊編碼和模擬工作負載。

Graviton2 CPU 使用 Neoverse N1 核心，並支援 Armv8.2 (包括 CRC 和加密延伸模組)，加上數個其他架構延伸模組。

Graviton2 透過每個 vCPU 提供更大的 L2 快取來減少記憶體讀取時間，進而改善 Web 和行動後端、微服務和資料處理系統的延遲效能。Graviton2 也提供改善的加密效能，並支援指令集，其可改善 CPU 型機器學習推論的延遲。

如需 AWS Graviton2 的詳細資訊，請參閱 AWS Graviton 處理器。

遷移到 arm64 架構的要求

當您選取要遷移至 arm64 架構的 Lambda 函數時，為了確保順利遷移，請確定您的函數符合下列需求：

- 部署套件只包含開放原始碼元件和您控制的原始碼，以便您可以針對遷移進行任何必要的更新。
- 如果函數程式碼包含第三方相依性，則每個程式庫或套件都會提供 arm64 版本。

函數程式碼與 arm64 架構的相容性

您的 Lambda 函數程式碼必須與函數的指令集架構相容。將函數遷移到 arm64 架構之前，請注意下列有關目前函數程式碼的幾點：

- 如果您使用內嵌的程式碼編輯器新增函數程式碼，則您的程式碼可能無需修改即可在任一架構上執行。
- 如果上傳了您的函數程式碼，則您必須上傳與目標架構相容的新程式碼。
- 如果您的函數使用層，則必須檢查每一層以確定其與新架構相容。如果層不相容，請編輯函數，將目前的層版本取代為相容的層版本。
- 如果您的函數使用 Lambda 延伸模組，則必須檢查每個延伸模組，以確定其與新架構相容。
- 如果您的函數使用容器映像部署套件類型，則必須建立與函數架構相容的新容器映像。

如何遷移到 arm64 架構

若要將 Lambda 函數遷移至 arm64 架構，建議您遵循下列步驟：

1. 為您的應用程式或工作負載建置相依性清單。常見相依性包括：
   - 函數使用的所有程式庫和套件。
   - 用來建置、部署和測試函數的工具，例如編譯器、測試套件、持續交付和持續整合 (CI/CD) 管道、佈建工具，以及指令碼。
   - Lambda 延伸模組和第三方工具，用來在生產環境中監控函數。
2. 對於每個相依性，請檢查版本，然後檢查 arm64 版本是否可用。
3. 建置環境來遷移應用程式。
4. 引導應用程式。

5. 測試和除錯應用程式。
6. 測試 arm64 函數的效能。將效能與 x86_64 版本進行比較。
7. 更新您的基礎設施管道以支援 arm64 Lambda 函數。
8. 將部署暫存至生產環境。

   例如，使用別名路由組態來分割函數 x86 和 arm64 版本之間的流量，並比較效能和延遲。

如需如何為 arm64 架構建立程式碼環境的詳細資訊，包括 Java、Go、.NET 和 Python 的語言特定資訊，請參閱 Graviton GitHub AWS GitHub 儲存庫入門。

設定指令集架構

您可以使用 Lambda 主控台、 AWS SDKs、 AWS Command Line Interface (AWS CLI) 或 為新的和現有的 Lambda 函數設定指令集架構 CloudFormation。依照這些步驟操作，在主控台中為現有的 Lambda 函數變更指令集架構。

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數名稱以為其設定指令集架構。
3. 在程式碼主索引標籤上的執行期設定區段選擇編輯。
4. 在 架構 選擇要用於函數的指令集架構。
5. 選擇儲存。

設定 Lambda 函數逾時

Lambda 會在逾時之前執行程式碼一段設定的時間。逾時為 Lambda 函數可以執行的最長時間，以秒為單位。此設定的預設值為 3 秒，但您可以調整此值，以 1 秒為單位，最大值為 900 秒 (15 分鐘)。

此頁面說明如何及何時更新 Lambda 函數的逾時設定。

章節

- 確定 Lambda 函數的適當逾時值
- 設定逾時 (主控台)
- 設定逾時 (AWS CLI)
- 設定逾時 (AWS SAM)

確定 Lambda 函數的適當逾時值

如果逾時值接近函數的平均持續時間，則函數意外逾時的風險更高。函數的持續時間會有所不同，這取決於資料傳輸和處理量，以及函數與之互動的任何服務的延遲。逾時的一些常見原因包括：

- Amazon Simple Storage Service (Amazon S3) 中的下載量比平均水平更大或耗時更長。
- 函數會向另一個服務提出請求，這需要更長的時間才能回應。
- 提供給函數的參數要求函數具有更高的運算複雜性，這會導致調用時間更長。

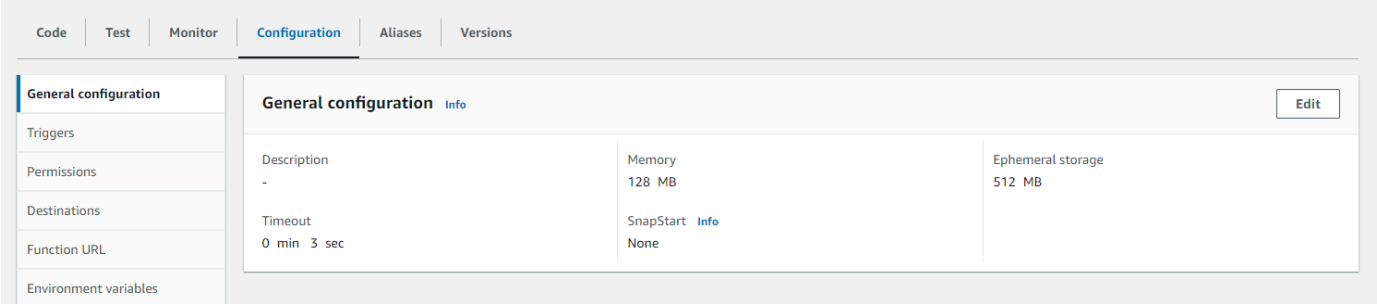
測試應用程式時，請確保測試可準確反映資料的大小和數量以及真實的參數值。為了方便起見，測試通常會使用小型範例，但您應該使用工作負載合理預期上限的資料集。

設定逾時 (主控台)

可以在 Lambda 主控台中設定函數逾時。

若要修改函數的逾時

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態索引標籤，然後選擇一般組態。



4. 在一般組態中，選擇編輯。
5. 對於逾時，設定介於 1 到 900 秒 (15 分鐘) 之間的值。
6. 選擇儲存。

設定逾時 (AWS CLI)

可以使用 update-function-configuration 命令，以秒為單位設定逾時值。下列範例命令會將函數逾時增加到 120 秒 (2 分鐘)。

Example

```
aws lambda update-function-configuration \
  --function-name my-function \
  --timeout 120
```

設定逾時 (AWS SAM)

可以使用 AWS Serverless Application Model 來設定函數的逾時值。更新 `template.yaml` 檔案中的 Timeout 屬性，然後執行 sam 部署。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
```

```
    Timeout: 120
    # Other function properties...
```

設定 Lambda 耐用函數

若要為 Lambda 函數啟用持久性執行，您需要設定特定設定，以控制函數的執行時間、保留狀態資料的時間，以及需要哪些許可。

啟用持久性執行

若要為 Lambda 函數啟用持久性執行，請在函數組態DurableConfig中設定 。此設定控制執行逾時、狀態保留和版本控制行為。

AWS CLI

```
aws lambda update-function-configuration \
  --function-name my-durable-function \
  --durable-config '{
    "ExecutionTimeout": 3600,
    "RetentionPeriodInDays": 30,
    "AllowInvokeLatest": true
  }'
```

CloudFormation

```
Resources:
  MyDurableFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-durable-function
      Runtime: nodejs18.x
      Handler: index.handler
      Code:
        ZipFile: |
          // Your durable function code
      DurableConfig:
        ExecutionTimeout: 3600
        RetentionPeriodInDays: 30
        AllowInvokeLatest: true
```

組態參數：

- `ExecutionTimeout` - 最長執行時間，以秒為單位（一年最多 31，536，000 個）
- `RetentionPeriodInDays` - 保留執行狀態和歷史記錄的時間長度 (1-365 天）
- `AllowInvokeLatest` - 是否允許調用 $LATEST 版本以進行持久性執行

耐用函數的 IAM 許可

除了標準 Lambda 執行角色之外，耐用函數需要額外的 IAM 許可。函數的執行角色必須包含狀態管理和持久執行 APIs許可。

最低必要許可：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction",
        "lambda:GetFunction",
        "lambda:ManageDurableState",
        "lambda:GetDurableExecution",
        "lambda:ListDurableExecutions"
      ],
      "Resource": "arn:aws:lambda:*:*:function:*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": "arn:aws:logs:*:*:*"
    }
  ]
```

```
}
```

CloudFormation 執行角色範例：

```
DurableFunctionRole:
  Type: AWS::IAM::Role
  Properties:
    AssumeRolePolicyDocument:
      Version: '2012-10-17'
      Statement:
        - Effect: Allow
          Principal:
            Service: lambda.amazonaws.com
          Action: sts:AssumeRole
    ManagedPolicyArns:
      - arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
    Policies:
      - PolicyName: DurableFunctionPolicy
        PolicyDocument:
          Version: '2012-10-17'
          Statement:
            - Effect: Allow
              Action:
                - lambda:ManageDurableState
                - lambda:GetDurableExecution
                - lambda:ListDurableExecutions
              Resource: '*'
```

設定最佳實務

設定生產用途的耐用函數時，請遵循下列最佳實務：

- 設定適當的執行逾時 - ExecutionTimeout 根據工作流程的預期持續時間上限來設定 。請勿設定不必要的長時間逾時，因為它們會影響成本和資源配置。
- 平衡保留與儲存成本 - RetentionPeriodInDays根據您的偵錯和稽核需求進行設定。較長的保留期會增加儲存成本。
- 在生產環境中使用版本控制 - false 在生產環境中將 AllowInvokeLatest設定為 ，並使用特定函數版本或別名進行持久性執行。

- 監控狀態大小 - 大型狀態物件會增加儲存成本，並可能影響效能。將狀態降至最低，並針對大型資料使用外部儲存。
- 設定適當的記錄 - 啟用詳細記錄，以對長時間執行的工作流程進行故障診斷，但請注意日誌磁碟區和成本。

生產組態範例：

```
{
  "ExecutionTimeout": 86400,
  "RetentionPeriodInDays": 7,
  "AllowInvokeLatest": false
}
```

使用 Lambda 環境變數

您可以使用環境變數來調整函數的行為，而無需更新程式碼。環境變數是存放在函數特定版本組態中的一對字串。Lambda 執行時間可讓程式碼使用環境變數，並設定其他環境變數，這些變數包含函數和調用請求的相關資訊。

> **Note**
>
> 為了提高安全性，我們建議您使用 AWS Secrets Manager 而非環境變數來存放資料庫登入資料和其他敏感資訊，例如 API 金鑰或授權字符。如需詳細資訊，請參閱在 Lambda 函式中使用 Secrets Manager 秘密。

在函數調用之前，不會評估環境變數。您定義的任何值都會視為文字字串，而且不會展開。在函數程式碼中執行變數評估。

建立 Lambda 環境變數

您可以使用 Lambda 主控台、 AWS Command Line Interface (AWS CLI)、 AWS Serverless Application Model (AWS SAM) 或使用 AWS SDK 在 Lambda 中設定環境變數。

Console

您可以在函數的未發佈版本上定義環境變數。發布版本時，該版本的環境變數與其他版本特定組態設定會被鎖定。

您可以透過定義索引鍵和值，為函數建立環境變數。函數使用索引鍵的名稱來擷取環境變數的值。

在 Lambda 主控台中設定環境變數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態索引標籤，然後選擇環境變數。
4. 在 Environment variables (環境變數) 下，選擇 Edit (編輯)。
5. 選擇 Add environment variable (新增環境變數)。
6. 輸入索引鍵和值。

請求

- 索引鍵需以英文字母為開頭，且至少有兩個字元。
- 索引鍵僅包含字母、數字和底線字元 (_)。
- 索引鍵不是由 Lambda 預留。
- 所有環境變數的大小總計不超過 4 KB。

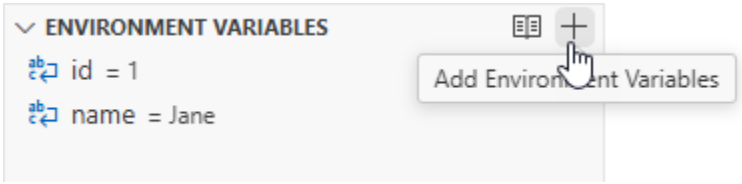
7. 選擇儲存。

在主控台程式碼編輯器中產生環境變數清單

可在 Lambda 程式碼編輯器中產生環境變數清單。這是在編碼時參考環境變數的快速方法。

1. 選擇 程式碼 標籤。
2. 向下捲動至程式碼編輯器的 ENVIRONMENT VARIABLES 區段。下面列出了現有環境變數：

3. 若要建立新的環境變數，請選擇加號 (+)：



在主控台程式碼編輯器中列出環境變數時，會保持加密狀態。如果在傳輸過程中啟用加密輔助工具進行加密，則這些設定會維持不變。如需詳細資訊，請參閱保護 Lambda 環境變數。

環境變數清單為唯讀狀態，且只能在 Lambda 主控台中使用。下載函數的 .zip 封存檔時，不會包含此檔案，且您無法透過上傳此檔案來新增環境變數。

AWS CLI

下列範例會在名為 `my-function` 的函數上設定兩個環境變數。

```
aws lambda update-function-configuration \
  --function-name my-function \
```

```
  --environment "Variables={BUCKET=amzn-s3-demo-bucket,KEY=file.txt}"
```

當您使用 update-function-configuration 命令套用環境變數時，會取代 Variables 結構的整個內容。若要在新增環境變數時保留現有的環境變數，請在請求中包含所有現有值。

若要取得目前的組態，請使用 get-function-configuration 命令。

```
aws lambda get-function-configuration \
  --function-name my-function
```

您應該會看到下列輸出：

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:111122223333:function:my-function",
    "Runtime": "nodejs24.x",
    "Role": "arn:aws:iam::111122223333:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "amzn-s3-demo-bucket",
            "KEY": "file.txt"
        }
    },
    "RevisionId": "0894d3c1-2a3d-4d48-bf7f-abade99f3c15",
    ...
}
```

可以從 get-function-configuration 的輸出中將修訂 ID 作為參數傳遞給 update-function-configuration。這樣可確保在讀取組態和更新組態期間，值不會變更。

若要設定函數的加密金鑰，請設定 KMSKeyARN 選項。

```
aws lambda update-function-configuration \
  --function-name my-function \
  --kms-key-arn arn:aws:kms:us-east-2:111122223333:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

AWS SAM

可以使用 AWS Serverless Application Model 來設定函數的環境變數。更新 template.yaml 檔案中的環境和變數屬性，然後執行 sam 部署。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: An AWS Serverless Application Model template describing your function.
Resources:
  my-function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: .
      Description: ''
      MemorySize: 128
      Timeout: 120
      Handler: index.handler
      Runtime: nodejs24.x
      Architectures:
        - x86_64
      EphemeralStorage:
        Size: 10240
      Environment:
        Variables:
          BUCKET: amzn-s3-demo-bucket
          KEY: file.txt
      # Other function properties...
```

AWS SDKs

若要使用 AWS SDK 管理環境變數，請使用下列 API 操作。

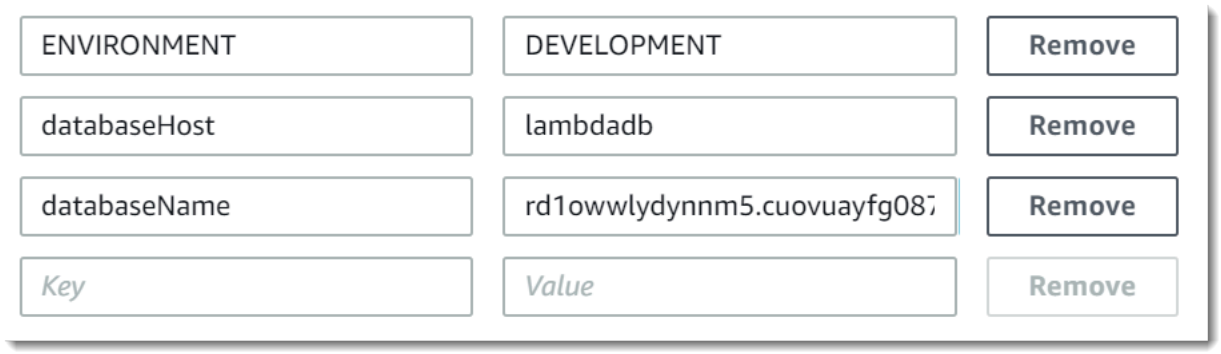
- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

若要進一步了解，請參閱您偏好的程式設計語言的 AWS SDK 文件。

環境變數的範例案例

您可以使用環境變數，自訂測試環境和生產環境中的函數行為。例如，您可以建立兩個具備相同程式碼，但不同組態的函數。一個函數連接到測試資料庫，另一個函數連接到生產資料庫。在此情況下，可以使用環境變數將資料庫的主機名稱和其他連線詳細資訊傳遞給函數。

以下範例顯示如何將資料庫主機和資料庫名稱定義為環境變數。



如果您希望測試環境產生比實際執行環境更多的偵錯資訊，您可以設定環境變數，以將測試環境設定為使用更詳細的記錄或更詳細的追蹤。

例如，在測試環境中，您可以使用 LOG_LEVEL 索引鍵與表示偵錯或追蹤日誌層級的值來設定環境變數。在 Lambda 函式的程式碼中，您可以使用此環境變數來設定日誌層級。

下列 Python 與 Node.js 語言程式碼範例示範可如何完成此操作。這些範例假設環境變數在 Python 中的值為 DEBUG，在 Node.js 中的值為 debug。

Python

Example設定日誌層級的 Python 程式碼

```
import os
import logging

# Initialize the logger
logger = logging.getLogger()

# Get the log level from the environment variable and default to INFO if not set
log_level = os.environ.get('LOG_LEVEL', 'INFO')

# Set the log level
logger.setLevel(log_level)

def lambda_handler(event, context):
    # Produce some example log outputs
    logger.debug('This is a log with detailed debug information - shown only in test
 environment')
    logger.info('This is a log with standard information - shown in production and
 test environments')
```

Node.js (ES module format)

Example設定日誌層級的 Node.js 程式碼

此範例使用 `winston` 記錄程式庫。使用 npm 將此程式庫新增至函式的部署套件。如需詳細資訊，請參閱the section called “建立含相依項的 .zip 部署套件”。

```
import winston from 'winston';

// Initialize the logger using the log level from environment variables, defaulting
 to INFO if not set
const logger = winston.createLogger({
   level: process.env.LOG_LEVEL || 'info',
   format: winston.format.json(),
   transports: [new winston.transports.Console()]
});

export const handler = async (event) => {
   // Produce some example log outputs
   logger.debug('This is a log with detailed debug information - shown only in test
 environment');
   logger.info('This is a log with standard information - shown in production and
 test environment');

};
```

擷取 Lambda 環境變數

若要在函數程式碼中擷取環境變數，請使用程式設計語言的標準方法。

Node.js

```
let region = process.env.AWS_REGION
```

Python

```
import os
```

```
region = os.environ['AWS_REGION']
```

> **Note**
>
> 在某些情況下，您可能需要使用下列格式：
>
> ```
> region = os.environ.get('AWS_REGION')
> ```

Ruby

```
region = ENV["AWS_REGION"]
```

Java

```
String region = System.getenv("AWS_REGION");
```

Go

```
var region = os.Getenv("AWS_REGION")
```

C#

```
string region = Environment.GetEnvironmentVariable("AWS_REGION");
```

PowerShell

```
$region = $env:AWS_REGION
```

Lambda 透過靜態加密環境變數來安全地存放它們。您可以設定 Lambda 使用不同的加密金鑰、在用戶端加密環境變數值，或在具有 的 CloudFormation 範本中設定環境變數 AWS Secrets Manager。

定義執行時間環境變數

Lambda 執行時間會在初始化期間設定數個環境變數。大多數的環境變數都會提供函數或執行時間的資訊。這些環境變數的索引鍵都已進行預留，無法在您的函數組態中設定。

預留環境變數

- _HANDLER - 在函數中設定的處理常式位置。
- _X_AMZN_TRACE_ID - X-Ray 追蹤標頭。此環境變數會隨著每次調用而變更。
  - 未針對僅限作業系統的執行期 (provided 執行期系列) 定義此環境變數。您可以使用 下次調用 中的 Lambda-Runtime-Trace-Id 回應標頭為自訂執行階段設定 _X_AMZN_TRACE_ID。
  - 對於 Java 執行期版本 17 及更高版本，不使用此環境變數。相反地，Lambda 會將追蹤資訊儲存在 com.amazonaws.xray.traceHeader 系統屬性中。
- AWS_DEFAULT_REGION – 執行 Lambda 函數的預設 AWS 區域 位置。
- AWS_REGION – 執行 Lambda 函數 AWS 區域 的 。若已完成定義，此值便會覆寫 AWS_DEFAULT_REGION。
  - 如需搭配 AWS SDKs 使用 AWS 區域 環境變數的詳細資訊，請參閱 AWS SDKs和工具參考指南中的AWS 區域。
- AWS_EXECUTION_ENV – 執行時間識別符，字首為 AWS_Lambda_ (例如，AWS_Lambda_java8)。未針對僅限作業系統的執行期 (provided 執行期系列) 定義此環境變數。
- AWS_LAMBDA_FUNCTION_NAME - 函數的名稱。
- AWS_LAMBDA_FUNCTION_MEMORY_SIZE - 可供函數使用的記憶體量 (MB)。
- AWS_LAMBDA_FUNCTION_VERSION - 正在執行的函數版本。
- AWS_LAMBDA_INITIALIZATION_TYPE – 函數的初始化類型，即 on-demand、snap-start、provisioned-concurrency或 lambda-managed-instances。如需詳細資訊，請參閱設定佈建並行使用 Lambda SnapStart 改善啟動效能、 或 Lambda 受管執行個體。
- AWS_LAMBDA_LOG_GROUP_NAME、AWS_LAMBDA_LOG_STREAM_NAME - Amazon CloudWatch Logs 群組和函數串流的名稱。AWS_LAMBDA_LOG_GROUP_NAME 和 AWS_LAMBDA_LOG_STREAM_NAME 環境變數無法在 Lambda SnapStart 函數中使用。
- AWS_ACCESS_KEY、AWS_ACCESS_KEY_ID、AWS_SECRET_ACCESS_KEY、AWS_SESSION_TOKEN - 從函數的執行角色中取得的存取金鑰。
- AWS_LAMBDA_RUNTIME_API - (自訂執行時間) 執行時間 API 的主機和連接埠。
- LAMBDA_TASK_ROOT - 指向您 Lambda 函數程式碼的路徑。
- LAMBDA_RUNTIME_DIR - 指向執行時間程式庫的路徑。
- AWS_LAMBDA_MAX_CONCURRENCY – （僅限 Lambda 受管執行個體） Lambda 將傳送至一個執行環境的並行調用數目上限。
- AWS_LAMBDA_METADATA_API – 格式的中繼資料端點伺服器地址 {ipv4_address}:{port}（例如 169.254.100.1:9001)。

- AWS_LAMBDA_METADATA_TOKEN – 目前執行環境的唯一身分驗證字符，用於對中繼資料端點的請求進行身分驗證。Lambda 會在初始化時自動產生此權杖。

下列其他環境變數並未進行預留，可以在您的函數組態中進行擴充。

未預留的環境變數

- LANG – 執行時間的地區設定 (en_US.UTF-8)。
- PATH – 執行路徑 (/usr/local/bin:/usr/bin/:/bin:/opt/bin)。
- LD_LIBRARY_PATH – 系統程式庫路徑 (/var/lang/lib:/lib64:/usr/lib64:$LAMBDA_RUNTIME_DIR:$LAMBDA_RUNTIME_DIR/lib:$LAMBDA_TASK_ROOT:$LAMBDA_TASK_ROOT/lib:/opt/lib)。
- NODE_PATH - (Node.js) Node.js 程式庫路徑 (/opt/nodejs/node12/node_modules/:/opt/nodejs/node_modules:$LAMBDA_RUNTIME_DIR/node_modules)。
- NODE_OPTIONS – (Node.js) 對於 Node.js 執行時期，您可以使用 NODE_OPTIONS 來重新啟用 Lambda 預設停用的實驗性功能。
- PYTHONPATH – (Python) Python 程式庫路徑 ($LAMBDA_RUNTIME_DIR)。
- GEM_PATH - (Ruby) Ruby 程式庫路徑 ($LAMBDA_TASK_ROOT/vendor/bundle/ruby/3.3.0:/opt/ruby/gems/3.3.0)。
- AWS_XRAY_CONTEXT_MISSING - 對於 X-Ray 追蹤，Lambda 將它設定為 LOG_ERROR，以避免從 X-Ray 開發套件中擲回執行時間錯誤。
- AWS_XRAY_DAEMON_ADDRESS - 對於 X-Ray 追蹤，為 X-Ray 常駐程式的 IP 地址和連接埠。
- AWS_LAMBDA_DOTNET_PREJIT – (.NET) 設定此變數，以啟用或停用 .NET 特定執行時期最佳化。值包含 always、never 和 provisioned-concurrency。如需詳細資訊，請參閱設定函數的佈建並行。
- TZ – 環境的時區 (:UTC)。執行環境使用 NTP 來同步系統時鐘。

顯示的範例值會反映最新的執行時間。特定變數或其值是否存在，可能會因先前的執行時間而有所不同。

保護 Lambda 環境變數

若要保護您的環境變數，可以使用伺服器端加密來保護您的靜態資料，並使用用戶端加密來保護傳輸中的資料。

> **Note**
>
> 為了提高資料庫安全性，我們建議您使用 AWS Secrets Manager 而非環境變數來存放資料庫登入資料。如需詳細資訊，請參閱在 Lambda 函式中使用 Secrets Manager 秘密。

靜態安全

Lambda 永遠使用 AWS KMS key提供靜態伺服器端加密。預設情況下，Lambda 使用 AWS 受管金鑰。如果此預設行為符合您的工作流程，您便不需要設定其他項目。Lambda 會在您的帳戶 AWS 受管金鑰 中建立 ，並為您管理許可。 AWS 不會向您收取使用此金鑰的費用。

如果您願意，您可以改為提供 AWS KMS 客戶受管金鑰。您可以這樣做以控制 KMS 金鑰的輪換或滿足您的組織對管理 KMS 金鑰的請求。當您使用客戶受管的金鑰時，只有您帳戶中具有 KMS 金鑰存取權的使用者才能檢視或管理函數上的環境變數。

客戶受管金鑰會產生標準 AWS KMS 費用。如需詳細資訊，請參閱 AWS Key Management Service 定價。

傳輸中安全

基於額外的安全性考量，您可啟用傳輸中加密輔助工具，此類工具能確保環境變數在傳輸過程中透過用戶端加密得到加密保護。

若要為環境變數配置加密

1. 使用 AWS Key Management Service (AWS KMS) 為 Lambda 建立任何客戶受管金鑰，以用於伺服器端和用戶端加密。如需詳細資訊，請參閱AWS Key Management Service 開發人員指南中的建立金鑰。
2. 使用 Lambda 主控台，導覽至 Edit environment variables (編輯環境變數) 頁面。
   a. 開啟 Lambda 主控台中的 函數頁面。
   b. 選擇一個函數。
   c. 選擇 Configuration (組態)，然後從左側導覽列選擇 Environment variables (環境變數)。
   d. 在 Environment variables (環境變數) 區段中，選擇 Edit (編輯)。
   e. 展開 Encryption configuration (加密組態)。
3. (選用) 啟用主控台加密輔助工具，以透過用戶端加密保護傳輸中的資料。
   a. 在傳輸中加密下，選擇啟用傳輸中加密輔助工具。

b. 對於每個要啟用主控台加密輔助工具的環境變數，請選擇該環境變數旁的加密選項。

c. 在 AWS KMS key 下加密傳輸中，選擇您在此程序開始時建立的客戶受管金鑰。

d. 選擇 Execution role policy (執行角色政策)，然後複製政策。此政策授予函數的執行角色解密環境變數的許可。

   儲存此政策，以便在此程序的最後一個步驟中使用。

e. 向函數中新增解密環境變數的程式碼。若要查看範例，請選擇解密秘密程式碼片段。

4. (選用) 指定用於靜態加密的客戶受管金鑰。

   a. 選擇 Use a customer master key (使用客戶主金鑰)。

   b. 選擇在此程序開始時建立的客戶受管金鑰。

5. 選擇 Save (儲存)。

6. 設定許可。

   如果搭配使用客戶受管金鑰和伺服器端加密，請向您希望其能檢視或管理該函數環境變數的任何使用者或角色授予許可。如需詳細資訊，請參閱管理伺服器端加密 KMS 金鑰的許可。

   如果您要啟用用戶端加密來增強傳輸中安全，您的函數需要許可來呼叫 `kms:Decrypt` API 操作。將您先前在此程序中儲存的政策新增至函數的執行角色。

管理伺服器端加密 KMS 金鑰的許可

您的使用者或函數的執行角色不需要任何 AWS KMS 許可，即可使用預設加密金鑰。若要使用客戶受管金鑰，您需要使用金鑰的許可。Lambda 會使用您的許可，在金鑰上建立授權。這麼做可讓 Lambda 使用它來進行加密。

- `kms:ListAliases` – 在 Lambda 主控台中檢視金鑰。
- `kms:CreateGrant`、`kms:Encrypt` - 在函數上設定客戶受管金鑰。
- `kms:Decrypt` - 檢視及管理使用客戶受管金鑰加密的環境變數。

您可以從 AWS 帳戶 或從金鑰的資源型許可政策取得這些許可。 `ListAliases` 是由 Lambda 的 受管政策提供。金鑰政策會將其餘許可授予 Key users (金鑰使用者) 群組中的使用者。

沒有 `Decrypt` 許可的使用者仍然可以管理函數，但是無法在 Lambda 主控台中檢視或管理環境變數。如要防止使用者檢視環境變數，請將拒絕存取預設金鑰、客戶受管金鑰，或是所有金鑰的陳述式新增到使用者的許可。

Example IAM 政策 - 透過金鑰 ARN 拒絕存取

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Deny",
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:us-east-2:111122223333:key/3be10e2d-
xmpl-4be4-bc9d-0405a71945cc"
        }
    ]
}
```

如需有關管理金鑰許可的詳細資訊，請參閱《AWS Key Management Service 開發人員指南》中的AWS KMS中的金鑰政策。

讓 Lambda 函數存取 Amazon VPC 中的資源

使用 Amazon Virtual Private Cloud (Amazon VPC)，您可以在 中建立私有網路 AWS 帳戶 以託管 資源，例如 Amazon Elastic Compute Cloud (Amazon EC2) 執行個體、Amazon Relational Database Service (Amazon RDS) 執行個體和 Amazon ElastiCache 執行個體。可以透過包含資源的私有子網路，將函數連接至 VPC，讓 Lambda 函數存取 Amazon VPC 中託管的資源。請依照下列各節中的指示，使用 Lambda 主控台、 AWS Command Line Interface (AWS CLI) 或 將 Lambda 函數連接至 Amazon VPC AWS SAM。

> **Note**
>
> 每個 Lambda 函數都會在 Lambda 服務擁有和管理的 VPC 內執行。這些 VPC 由 Lambda 自動維護，客戶看不到。將函數設定為存取 Amazon VPC 中的其他 AWS 資源，不會影響函數在其中執行的 Lambda 受管 VPC。

章節

- 所需的 IAM 許可
- 將 Lambda 函數連接至 中的 Amazon VPC AWS 帳戶
- 連接到 VPC 時的網際網路存取
- IPv6 支援
- 搭配使用 Lambda 與 Amazon VPC 的最佳實務
- 了解 Hyperplane Elastic Network Interfaces (ENI)
- 使用 IAM 條件金鑰進行 VPC 設定
- VPC 教學課程

所需的 IAM 許可

若要將 Lambda 函數連接至 中的 Amazon VPC AWS 帳戶，Lambda 需要許可來建立和管理其用來讓函數存取 VPC 中的資源的網路界面。

Lambda 建立的網路介面稱為 Hyperplane Elastic Network Interfaces 或 Hyperplane ENI。若要進一步了解這些網路介面，請參閱 the section called “了解 Hyperplane Elastic Network Interfaces (ENI)”。

您可以將 AWS 受管政策 AWSLambdaVPCAccessExecutionRole 連接至函數的執行角色，為函數提供所需的許可。當您在 Lambda 主控台中建立新函數並將其連接至 VPC 時，Lambda 會自動為您新增此許可政策。

如果想要建立自己的 IAM 許可政策，請務必新增下列所有許可並允許這些許可適用於所有資源 (`"Resource": "*"`)：

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2：DescribeSubnets
- ec2:DeleteNetworkInterface
- ec2:AssignPrivateIpAddresses
- ec2:UnassignPrivateIpAddresses

請注意，函數的角色只需要這些許可來建立網路介面，而不是調用函數。即使從函數的執行角色中移除這些許可，仍可在函數連接至 Amazon VPC 時成功調用函數。

若要將函數連接至 VPC，Lambda 還需要使用您的 IAM 使用者角色來驗證網路資源。確保使用者角色具有下列 IAM 許可：

- ec2:DescribeSecurityGroups
- ec2:DescribeSubnets
- ec2:DescribeVpcs
- ec2：GetSecurityGroupsForVpc

> **Note**
>
> Lambda 服務會使用您授予給函數執行角色的 Amazon EC2 許可，將函數連接至 VPC。不過，您也會隱含地將這些許可授予給函數的程式碼。這意味著您的函數程式碼能夠進行這些 Amazon EC2 API 呼叫。如需有關遵循安全最佳實務的建議，請參閱 the section called “安全最佳實務”。

將 Lambda 函數連接至 中的 Amazon VPC AWS 帳戶

AWS 帳戶 使用 Lambda 主控台、 AWS CLI 或 將函數連接至 中的 Amazon VPC AWS SAM。如果您使用 AWS CLI 或 AWS SAM，或使用 Lambda 主控台將現有函數連接至 VPC，請確定函數的執行角色具有上一節中列出的必要許可。

Lambda 函數無法透過專用執行個體租用直接連線到 VPC。若要連線到專用 VPC 中的資源，請透過預設租用將它對等連線到第二個 VPC。

Lambda console

若要在建立時將函數連接至 Amazon VPC

1. 開啟 Lambda 主控台的函數頁面，然後選擇 建立函數。
2. 在 Basic information (基本資訊) 下，對於 Function name (函數名稱)，為您的函數輸入名稱。
3. 執行下列操作以配置函數的 VPC 設定：
   a. 展開 Advanced settings (進階設定)。
   b. 選取啟用 VPC，然後選擇要與函數連接的 VPC。
   c. (選擇性) 若要允許傳出 IPv6 流量，請選取允許雙堆疊子網路的 IPv6 流量。
   d. 選擇要為其建立網路介面的子網路和安全群組。如果您選取允許雙堆疊子網路的 IPv6 流量，則所有選取的子網路均必須具有 IPv4 CIDR 區塊和 IPv6 CIDR 區塊。

   > **Note**
   >
   > 若要存取私有資源，請將您的函數連線到私有子網路。如果函數需要網際網路存取，請參閱 the section called “Lambda 函數的網際網路存取”。將函數連接到公有子網路並不會提供網際網路存取或公有 IP 位址給該函數。

4. 選擇建立函數。

若要將現有函數連接至 Amazon VPC

1. 開啟 Lambda 主控台的函數頁面，然後選取您的函數。
2. 選擇組態索引標籤，然後選擇 VPC。
3. 選擇編輯。
4. 在 VPC 下，選取要連接函數的 Amazon VPC。

5. (選擇性) 若要允許傳出 IPv6 流量，請選取允許雙堆疊子網路的 IPv6 流量。
6. 選擇要為其建立網路介面的子網路和安全群組。如果您選取允許雙堆疊子網路的 IPv6 流量，則所有選取的子網路均必須具有 IPv4 CIDR 區塊和 IPv6 CIDR 區塊。

> **Note**
>
> 若要存取私有資源，請將您的函數連線到私有子網路。如果函數需要網際網路存取，請參閱 the section called “Lambda 函數的網際網路存取”。將函數連接到公有子網路並不會提供網際網路存取或公有 IP 位址給該函數。

7. 選擇儲存。

AWS CLI

若要在建立時將函數連接至 Amazon VPC

- 若要建立 Lambda 函數並將其連接至 VPC，請執行下列 CLI `create-function` 命令。

```
aws lambda create-function --function-name my-function \
--runtime nodejs24.x --handler index.js --zip-file fileb://function.zip \
--role arn:aws:iam::123456789012:role/lambda-role \
--vpc-config
 Ipv6AllowedForDualStack=true,SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a03
```

指定您自己的子網路和安全群組，並根據使用案例將 `Ipv6AllowedForDualStack` 設定為 `true` 或 `false`。

若要將現有函數連接至 Amazon VPC

- 若要將現有函數連接至 VPC，請執行下列 CLI `update-function-configuration` 命令。

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config Ipv6AllowedForDualStack=true,
 SubnetIds=subnet-071f712345678e7c8,subnet-07fd123456788a036,SecurityGroupIds=sg-085912
```

若要從 VPC 中斷開函數的連接

- 若要從 VPC 中斷開函數的連接，請使用 VPC 子網路和安全群組的空清單來執行下列 `update-function-configuration` CLI 命令。

```
aws lambda update-function-configuration --function-name my-function \
--vpc-config SubnetIds=[],SecurityGroupIds=[]
```

AWS SAM

若要將函數連接至 VPC

- 若要將 Lambda 函數連接至 Amazon VPC，請將 `VpcConfig` 屬性新增至函數定義，如下列範例範本所示。如需此屬性的詳細資訊，請參閱CloudFormation 《 使用者指南》中的 AWS：：Lambda：：Function VpcConfig（屬性 AWS SAM `VpcConfig`會直接傳遞至 資源的 `VpcConfig` CloudFormation `AWS::Lambda::Function` 屬性）。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31

Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./lambda_function/
      Handler: lambda_function.handler
      Runtime: python3.12
      VpcConfig:
        SecurityGroupIds:
          - !Ref MySecurityGroup
        SubnetIds:
          - !Ref MySubnet1
          - !Ref MySubnet2
      Policies:
        - AWSLambdaVPCAccessExecutionRole

  MySecurityGroup:
    Type: AWS::EC2::SecurityGroup
    Properties:
      GroupDescription: Security group for Lambda function
      VpcId: !Ref MyVPC
```

```
MySubnet1:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref MyVPC
    CidrBlock: 10.0.1.0/24

MySubnet2:
  Type: AWS::EC2::Subnet
  Properties:
    VpcId: !Ref MyVPC
    CidrBlock: 10.0.2.0/24

MyVPC:
  Type: AWS::EC2::VPC
  Properties:
    CidrBlock: 10.0.0.0/16
```

如需在 中設定 VPC 的詳細資訊 AWS SAM，請參閱CloudFormation 《 使用者指南》中的 AWS：：EC2：：VPC。

連接到 VPC 時的網際網路存取

依預設，Lambda 函數可以存取公有網際網路。將函數連接至 VPC 時，它只能存取該 VPC 中可用的資源。若要讓您的函數存取網際網路，也需要設定 VPC 以存取網際網路。如需詳細資訊，請參閱 the section called “Lambda 函數的網際網路存取”。

IPv6 支援

您的函數可透過 IPv6 連線至雙堆疊 VPC 子網路中的資源。此選項預設為關閉。若要允許傳出 IPv6 流量，請使用主控台或者具有 create-function 或 update-function-configuration 命令的 `--vpc-config Ipv6AllowedForDualStack=true` 選項。

> **Note**
>
> 若要允許 VPC 中的傳出 IPv6 流量，連線到該函數的所有子網路都必須是雙堆疊子網路。Lambda 不支援 VPC 中純 IPv6 子網路的傳出 IPv6 連線，亦不支援未連接 VPC 之函式的傳出 IPv6 連線。

您可以更新函數程式碼來透過 IPv6 明確連線至子網路資源。下列 Python 範例開啟了通訊端，並連接到 IPv6 伺服器。

Example— 連線至 IPv6 伺服器

```
def connect_to_server(event, context):
    server_address = event['host']
    server_port = event['port']
    message = event['message']
    run_connect_to_server(server_address, server_port, message)

def run_connect_to_server(server_address, server_port, message):
    sock = socket.socket(socket.AF_INET6, socket.SOCK_STREAM, 0)
    try:
        # Send data
        sock.connect((server_address, int(server_port), 0, 0))
        sock.sendall(message.encode())
        BUFF_SIZE = 4096
        data = b''
        while True:
            segment = sock.recv(BUFF_SIZE)
            data += segment
            # Either 0 or end of data
            if len(segment) < BUFF_SIZE:
                break
        return data
    finally:
        sock.close()
```

搭配使用 Lambda 與 Amazon VPC 的最佳實務

若要確保 Lambda VPC 組態符合最佳實務準則，請遵循以下章節的建議。

安全最佳實務

若要將 Lambda 函數連接至 VPC，需要為函數的執行角色提供一些 Amazon EC2 許可。需要這些許可才能建立函數用來存取 VPC 中資源的網路介面。不過，這些許可也會隱含授予給函數的程式碼。這意味著函數程式碼擁有進行這些 Amazon EC2 API 呼叫的許可。

若要遵循最低權限存取的原則，請將拒絕政策新增至函數的執行角色，如下列範例所示。此政策可防止函式程式碼呼叫 Amazon EC2 API，同時仍允許 Lambda 服務代表您管理 VPC 資源。此政策使用

`lambda:SourceFunctionArn` 條件索引鍵，該索引鍵僅適用於函式程式碼在執行期間進行的 API 呼叫。如需詳細資訊，請參閱the section called “來源函數 ARN”。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action": [
                 "ec2:CreateNetworkInterface",
                 "ec2:DeleteNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
                 "ec2:DescribeSubnets",
                 "ec2:DetachNetworkInterface",
                 "ec2:AssignPrivateIpAddresses",
                 "ec2:UnassignPrivateIpAddresses"
            ],
            "Resource": [ "*" ],
            "Condition": {
                "ArnEquals": {
                    "lambda:SourceFunctionArn": [
                        "arn:aws:lambda:us-
west-2:123456789012:function:my_function"
                    ]
                }
            }
        }
    ]
}
```

AWS 提供安全群組和網路存取控制清單 (ACLs)，以提高 VPC 的安全性。安全群組控制資源的傳入與傳出流量，網路 ACL 則是控制子網的傳入與傳出流量。安全群組可為大多數子網路提供足夠的存取控制。如果您想讓 VPC 多一層安全，可以使用網路 ACL。如需使用 Amazon VPC 時安全最佳實務的一般準則，請參閱《Amazon Virtual Private Cloud 使用者指南》中的 VPC 安全最佳實務。

效能最佳實務

將函數連接至 VPC 時，Lambda 會檢查是否有可用於連線的可用網路資源 (Hyperplane ENI)。Hyperplane ENI 與安全群組和 VPC 子網路的特定組合相關聯。如果已將一個函數連接至

VPC，在連接另一個函數時指定相同的子網路和安全群組意味著 Lambda 可以共用網路資源，並避免建立新的 Hyperplane ENI。如需 Hyperplane ENI 及其生命週期的詳細資訊，請參閱 the section called “了解 Hyperplane Elastic Network Interfaces (ENI)”。

了解 Hyperplane Elastic Network Interfaces (ENI)

Hyperplane ENI 是一種受管資源，可作為 Lambda 函數與您希望函數連線的資源之間的網路介面。將函數連接至 VPC 時，Lambda 服務會自動建立和管理這些 ENI。

Hyperplane ENI 不會直接顯示，而且您不需要設定或管理它們。不過，了解它們的運作方式可協助您了解函數在連接到 VPC 時的行為。

當您第一次使用特定子網路和安全群組組合將函數連接至 VPC 時，Lambda 會建立 Hyperplane ENI。帳戶中其他使用相同子網路和安全群組組合的函數也可以使用此 ENI。Lambda 會盡可能重複使用現有的 ENI，以最佳化資源使用率，並盡量減少建立新的 ENI。每個 Hyperplane ENI 均支援多達 65,000 個連線/連接埠。如果連線數量超過此限制，Lambda 會根據網路流量和並行需求自動擴展 ENI 數量。

對於新函數，當 Lambda 正在建立 Hyperplane ENI 時，函數會保持待定狀態，您無法調用它。只有在 Hyperplane ENI 就緒時，函數才會轉換為作用中狀態，這可能需要幾分鐘的時間。對於現有函數，您無法執行針對函數的其他操作，例如建立版本或更新函數的程式碼，但您可以繼續調用函數的先前版本。

在管理 ENI 生命週期的過程中，Lambda 可能會刪除並重新建立 ENI，以此平衡跨多個 ENI 的網路流量負載，或解決 ENI 運作狀態檢查中發現的問題。此外，如果 Lambda 函式連續保持閒置狀態 14 天，Lambda 會回收未使用的所有 Hyperplane ENI，並將函式狀態設定為 `Inactive`。下一個調用嘗試將失敗，而且函數會重新進入擱置狀態，直到 Lambda 完成 Hyperplane ENI 的建立或配置為止。建議設計不要依賴 ENI 的持續性。

當您更新函數以移除其 VPC 組態時，Lambda 最多需要 20 分鐘才能刪除連接的 Hyperplane ENI。Lambda 只會在沒有其他函數 (或已發佈的函數版本) 使用該 Hyperplane ENI 時刪除 ENI。

Lambda 需要函數執行角色中的許可才能刪除 Hyperplane ENI。如果您在 Lambda 刪除 Hyperplane ENI 之前刪除執行角色，Lambda 將無法刪除 Hyperplane ENI。您可以手動執行刪除作業。

使用 IAM 條件金鑰進行 VPC 設定

您可以針對 VPC 設定使用 Lambda 特定條件金鑰，為您的 Lambda 函數提供額外的許可控制。例如，您可以請求組織中的所有功能都連線到 VPC。您也可以指定函數的使用者可以和不可以使用的子網路和安全性群組。

Lambda 支援 IAM 政策中的以下條件金鑰：

- lambda:VpcIds - 允許或拒絕一個或多個 VPC。
- lambda:SubnetIds - 允許或拒絕一個或多個子網路。
- lambda:SecurityGroupIds - 允許或拒絕一個或多個安全群組。

Lambda API 操作 CreateFunction 和 UpdateFunctionConfiguration 支援這些條件金鑰。如需有關在 IAM 政策中使用條件金鑰的詳細資訊，請參閱 IAM 使用者指南中的 IAM JSON 政策元素：條件。

> **Tip**
>
> 如果您的函數已經包含來自先前 API 請求的 VPC 組態，則可以在沒有 VPC 組態的情況下傳送 `UpdateFunctionConfiguration` 請求。

具有 VPC 設定條件金鑰的範例政策

下列範例示範如何使用條件金鑰進行 VPC 設定。建立具有所需限制的政策陳述式之後，請附加目標 使用者或角色的政策陳述式。

確保使用者僅部署與 VPC 連線的函數

若要確保所有使用者僅部署與 VPC 連線的函數，您可以拒絕不含有效 VPC ID 的函數建立和更新操作。

請注意，VPC ID 不是 `CreateFunction` 或 `UpdateFunctionConfiguration` 請求的輸入參數。Lambda 會根據子網路和安全群組參數擷取 VPC ID 值。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "EnforceVPCFunction",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
```

```
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "Null": {
           "lambda:VpcIds": "true"
        }
      }
    }
  ]
}
```

拒絕使用者存取特定 VPC、子網路或安全群組

若要拒絕使用者存取特定 VPC，請使用 `StringEquals` 來檢查 `lambda:VpcIds` 條件的值。下列範例會拒絕使用者存取 `vpc-1` 和 `vpc-2`。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "EnforceOutOfVPC",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Effect": "Deny",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "lambda:VpcIds": [
                        "vpc-1",
                        "vpc-2"
                    ]
                }
            }
        }
    ]
}
```

若要拒絕使用者存取特定子網路，請使用 StringEquals 來檢查 lambda:SubnetIds 條件的值。下列範例會拒絕使用者存取 subnet-1 和 subnet-2。

```
{
      "Sid": "EnforceOutOfSubnet",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
    }
```

若要拒絕使用者存取特定安全性群組，請使用 StringEquals 來檢查 lambda:SecurityGroupIds 條件的值。下列範例會拒絕使用者存取 sg-1 和 sg-2。

```
{
      "Sid": "EnforceOutOfSecurityGroups",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Deny",
      "Resource": "*",
      "Condition": {
        "ForAnyValue:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

允許使用者使用特定 VPC 設定建立和更新功能

若要允許使用者存取特定 VPC，請使用 StringEquals 來檢查 lambda:VpcIds 條件的值。下列範例可讓使用者存取 vpc-1 和 vpc-2。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "EnforceStayInSpecificVpc",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Effect": "Allow",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "lambda:VpcIds": [
                        "vpc-1",
                        "vpc-2"
                    ]
                }
            }
        }
    ]
}
```

若要允許使用者存取特定子網路，請使用 StringEquals 來檢查 lambda:SubnetIds 條件的值。下列範例可讓使用者存取 subnet-1 和 subnet-2。

```
{
      "Sid": "EnforceStayInSpecificSubnets",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
```

```
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SubnetIds": ["subnet-1", "subnet-2"]
        }
      }
    }
```

若要允許使用者存取特定安全性群組，請使用 `StringEquals` 來檢查 `lambda:SecurityGroupIds` 條件的值。下列範例可讓使用者存取 `sg-1` 和 `sg-2`。

```
{
      "Sid": "EnforceStayInSpecificSecurityGroup",
      "Action": [
          "lambda:CreateFunction",
          "lambda:UpdateFunctionConfiguration"
       ],
      "Effect": "Allow",
      "Resource": "*",
      "Condition": {
        "ForAllValues:StringEquals": {
            "lambda:SecurityGroupIds": ["sg-1", "sg-2"]
        }
      }
    }
  ]
}
```

VPC 教學課程

在以下教學中，您會將 Lambda 函數連線至 VPC 中的資源。

- 教學課程：使用 Lambda 函數來存取 Amazon VPC 中的 Amazon RDS
- 教學課程：設定 Lambda 函數以在 Amazon VPC 中存取 Amazon ElastiCache

讓 Lambda 函數存取另一個帳戶中 Amazon VPC 中的資源

您可以讓 AWS Lambda 函數存取由另一個帳戶管理的 Amazon Virtual Private Cloud 中的 Amazon VPC 中的資源，而無需將任一 VPC 暴露到網際網路。此存取模式可讓您使用 與其他組織共用資料 AWS。使用此存取模式，可以在 VPC 之間共用資料，其安全性和效能高於網際網路。將 Lambda 函數設定為使用 Amazon VPC 對等互連來存取這些資源。

> **Warning**
>
> 當您允許帳戶或 VPC 之間的存取時，請檢查您的計劃是否符合管理這些帳戶之各個組織的安全要求。遵循本文件中的指示將影響資源的安全狀態。

在本教學課程中，使用 IPv4 透過對等互連將兩個帳戶連線在一起。可以設定尚未連線至 Amazon VPC 的 Lambda 函數。可以設定 DNS 解析，將函數連線到不提供靜態 IP 的資源。若要調整這些說明以適應其他對等互連案例，請參閱 VPC 對等互連指南。

先決條件

若要讓 Lambda 函數存取另一個帳戶中的資源，您必須擁有：

- Lambda 函數，設定為對資源進行身分驗證，然後從資源中讀取。
- 另一個帳戶中的資源，例如 Amazon RDS 叢集，可透過 Amazon VPC 提供。
- Lambda 函數帳戶和資源帳戶的憑證。如果您無權使用資源的帳戶，請聯絡授權使用者以準備該帳戶。
- 建立和更新 VPC (並支援 Amazon VPC 資源) 以與 Lambda 函數建立關聯的許可。
- 更新 Lambda 函數執行角色和 VPC 組態的許可。
- 在 Lambda 函數帳戶中建立 VPC 對等互連的許可。
- 在資源帳戶中接受 VPC 對等互連的許可。
- 更新資源 VPC (並支援 Amazon VPC 資源) 組態的許可。
- 調用 Lambda 函數的許可。

在函數帳戶中建立 Amazon VPC

在 Lambda 函數帳戶中建立 Amazon VPC、子網路、路由表和安全群組。

如何使用主控台建立 VPC、子網路和其他 VPC 資源

1. 在 https://console.aws.amazon.com/vpc/ 開啟 Amazon VPC 主控台。
2. 在儀表板上，選擇建立 VPC。
3. 對於 IPv4 CIDR 區塊，請提供私有 CIDR 區塊。CIDR 區塊不得與資源 VPC 中使用的區塊重疊。請勿挑選資源 VPC 用來將 IP 指派給資源的區塊，或已在資源 VPC 的路由表中定義的區塊。如需有關定義適當 CIDR 區塊的詳細資訊，請參閱 VPC CIDR 區塊。
4. 選擇自訂可用區域。
5. 選取與資源相同的可用區域。
6. 針對公有子網路數量，選擇 0。
7. 對於 VPC endpoints (VPC 端點)，選擇 None (無)。
8. 選擇建立 VPC。

為函數的執行角色授予 VPC 許可

將 AWSLambdaVPCAccessExecutionRole 附接至函數的執行角色，以允許它連線到 VPC。

若要為函數的執行角色授予 VPC 許可

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇 函數的名稱。
3. 選擇 Configuration (組態)。
4. 選擇許可。
5. 在角色名稱中，選擇執行角色。
6. 在許可政策區段中，選擇新增許可。
7. 在下拉式清單中，選擇附接政策。
8. 在搜尋方塊中，輸入 `AWSLambdaVPCAccessExecutionRole`。
9. 若要在政策名稱的左側，選擇核取方塊。
10. 選擇新增許可。

若要將函數連接至 Amazon VPC

1. 開啟 Lambda 主控台中的函數頁面。

2. 選擇 函數的名稱。
3. 選擇組態索引標籤，然後選擇 VPC。
4. 選擇編輯。
5. 在 VPC 中，選取您的 VPC
6. 在子網路中，選擇子網路。
7. 在安全群組中，選擇 VPC 的預設安全群組。
8. 選擇儲存。

建立 VPC 對等互連請求

建立從函數的 VPC (請求者 VPC) 到資源的 VPC (接受者 VPC) 的 VPC 對等互連請求。

若要從函數的 VPC 請求 VPC 對等互連

1. 開啟 https://console.aws.amazon.com/vpc/。
2. 在導覽窗格中，選擇 Peering connections (對等互連)。
3. 關閉 Create peering connection (建立對等互連)。
4. 對於 VPC ID (請求者)，選取函數的 VPC。
5. 對於帳戶 ID，輸入資源帳戶的 ID。
6. 對於 VPC ID (接受者)，輸入資源的 VPC。

準備資源的帳戶

若要建立對等互連並準備資源的 VPC 以使用連線，請使用具有先決條件中所列許可的角色來登入資源的帳戶。登入步驟可能會根據帳戶的保護措施而有所不同。如需如何登入 AWS 帳戶的詳細資訊，請參閱 AWS 登入使用者指南。在資源的帳戶中，執行下列程序。

若要接受 VPC 對等互連請求

1. 開啟 https://console.aws.amazon.com/vpc/。
2. 在導覽窗格中，選擇 Peering connections (對等互連)。
3. 選取待定 VPC 對等互連 (狀態為待接受)。
4. 選擇 動作
5. 從下拉式清單中，選擇接受請求。

6. 出現確認提示時，請選擇接受請求。
7. 選擇立即修改我的路由表，將路由新增至 VPC 的主路由表，以便透過對等互連傳送和接收流量。

檢查資源 VPC 的路由表。根據資源 VPC 的設定方式，Amazon VPC 產生的路由可能無法建立連線。檢查新路由與 VPC 現有組態之間的衝突。如需有關疑難排解的詳細資訊，請參閱《Amazon 虛擬私有雲端 VPC 對等互連指南》中的 VPC 對等互連疑難排解。

若要更新資源的安全群組

1. 開啟 https://console.aws.amazon.com/vpc/。
2. 在導覽窗格中，選擇安全群組。
3. 選取資源的安全群組。
4. 選擇動作。
5. 從下拉式清單中，選擇編輯傳入規則。
6. 選擇新增規則。
7. 針對來源輸入函數的帳戶 ID 和安全群組 ID，以斜線分隔 (例如 111122223333/sg-1a2b3c4d)。
8. 選擇編輯傳出規則。
9. 檢查是否限制傳出流量。預設 VPC 設定允許所有傳出流量。如果傳出流量受限，請繼續下一個步驟。
10. 選擇新增規則。
11. 針對目的地輸入函數的帳戶 ID 和安全群組 ID，以斜線分隔 (例如 111122223333/sg-1a2b3c4d)。
12. 選擇儲存規則。

若要啟用對等互連的 DNS 解析

1. 開啟 https://console.aws.amazon.com/vpc/。
2. 在導覽窗格中，選擇 Peering connections (對等互連)。
3. 選取您的對等互連。
4. 選擇動作。
5. 選擇編輯 DNS 設定。
6. 在接受者 DNS 解析下，選取允許請求者 VPC 將接受者 VPC 主機的 DNS 解析為私有 IP。
7. 選擇儲存變更。

更新函數帳戶中的 VPC 組態

登入函數的帳戶，然後更新 VPC 組態。

若要新增 VPC 對等互連的路由

1. 開啟 https://console.aws.amazon.com/vpc/。
2. 在導覽窗格中，選擇 Route tables (路由表)。
3. 選取與函數相關聯子網路的路由表名稱旁邊的核取方塊。
4. 選擇動作。
5. 選擇 Edit routes (編輯路由)。
6. 選擇 Add route (新增路由)。
7. 對於目的地，輸入資源 VPC 的 CIDR 區塊。
8. 對於目標，選取 VPC 對等互連。
9. 選擇儲存變更。

如需有關更新路由表時可能遇到之考量的詳細資訊，請參閱更新 VPC 對等互連的路由表。

若要更新 Lambda 函數的安全群組

1. 開啟 https://console.aws.amazon.com/vpc/。
2. 在導覽窗格中，選擇安全群組。
3. 選擇動作。
4. 選擇 Edit inbound Rules (編輯傳入規則)。
5. 選擇新增規則。
6. 對於來源，輸入資源的帳戶 ID 和安全群組 ID，以正斜線分隔 (例如 111122223333/sg-1a2b3c4d)。
7. 選擇儲存規則。

若要啟用對等互連的 DNS 解析

1. 開啟 https://console.aws.amazon.com/vpc/。
2. 在導覽窗格中，選擇 Peering connections (對等互連)。
3. 選取您的對等互連。

4. 選擇動作。
5. 選擇編輯 DNS 設定。
6. 在請求者 DNS 解析下，選取允許接受者 VPC 將請求者 VPC 主機的 DNS 解析為私有 IP。
7. 選擇儲存變更。

測試 函數

若要建立測試事件並檢查函數的輸出

1. 在程式碼來源窗格中選擇測試。
2. 選取建立新事件。
3. 在事件 JSON 面板中，將預設值取代為 Lambda 函數適用的輸入。
4. 選擇調用。
5. 在執行結果索引標籤中，確認回應包含預期輸出。

此外，可以檢查函數的日誌，確認日誌如您所預期。

若要在 CloudWatch Logs 中檢視函數的調用記錄

1. 選擇 Monitor (監控) 索引標籤。
2. 選擇檢視 CloudWatch 日誌。
3. 在日誌串流標籤上，為函數調用選擇日誌串流。
4. 確認日誌如您所預期。

啟用 VPC 連線的 Lambda 函數的網際網路存取

根據預設，會在具有網際網路存取權的 Lambda 受管 VPC 中執行 Lambda 函數。若要存取帳戶中 VPC 的資源，可以將 VPC 組態新增至函數。這會將函數限制在該 VPC 內的資源，除非 VPC 具有網際網路存取權。此頁面說明如何提供 VPC 所連接 Lambda 函數的網際網路存取。

我還沒有 VPC

建立 VPC

建立 VPC 工作流程會建立 Lambda 函數所需的所有 VPC 資源，以從私有子網路中存取公有網際網路，包括子網路、NAT 閘道、網際網路閘道和路由表項目。

若要建立 VPC

1. 在 https://console.aws.amazon.com/vpc/ 開啟 Amazon VPC 主控台。
2. 在儀表板上，選擇建立 VPC。
3. 針對 Resources to create (建立資源)，選擇 VPC and more (VPC 等)。
4. 設定 VPC
   a. 針對自動產生名稱標籤，輸入 VPC 的名稱。
   b. 對於 IPv4 CIDR 區塊，您可以保留預設建議，或者您也可以輸入應用程式或網路所需的 CIDR 區塊。
   c. 如果您的應用程式使用 IPv6 地址進行通訊，請選擇 IPv6 CIDR 區塊 > Amazon 提供的 IPv6 CIDR 區塊。
5. 設定子網路
   a. 針對可用區域數量，選擇 2。建議至少使用兩個可用區域，以實現高可用性。
   b. 針對公用子網路數量，選擇 2。
   c. 在 Number of private subnet (私有子網路數量) 中，選擇 2。
   d. 您可以保留公有子網路的預設 CIDR 區塊，或者您也可以展開自訂子網路 CIDR 區塊並輸入 CIDR 區塊。如需詳細資訊，請參閱子網路 CIDR 區塊。
6. 若為 NAT 閘道，請選擇每個 AZ 1 個以提高彈性。
7. 對於僅限輸出的網際網路閘道，如果選擇包含 IPv6 CIDR 區塊，請選擇是。
8. 若為 VPC 端點，請保留預設值 (S3 閘道)。此選項沒有任何費用。如需詳細資訊，請參閱適用於 Amazon S3 的 VPC 端點類型。

9. 對於 DNS 選項，保留預設的設定。
10. 選擇建立 VPC。

設定 Lambda 函數

若要在建立函數時設定 VPC

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇建立函數。
3. 在 Basic information (基本資訊) 下，對於 Function name (函數名稱)，為您的函數輸入名稱。
4. 展開 Advanced settings (進階設定)。
5. 選取啟用 VPC，然後選擇一個 VPC。
6. (選擇性) 若要允許傳出 IPv6 流量，請選取允許雙堆疊子網路的 IPv6 流量。
7. 對於子網路，選取所有私有子網路。私有子網路可透過 NAT 閘道存取網際網路。將函數連線到公有子網路並不會為其提供網際網路存取。

> **Note**
>
> 如果您選取允許雙堆疊子網路的 IPv6 流量，則所有選取的子網路均必須具有 IPv4 CIDR 區塊和 IPv6 CIDR 區塊。

8. 對於安全群組，選取允許傳出流量的安全群組。
9. 選擇建立函數。

Lambda 會使用 AWSLambdaVPCAccessExecutionRole AWS 受管政策自動建立執行角色。此政策中的許可僅適用於為 VPC 組態建立彈性網路介面，而非調用函數。若要套用最低權限許可，可以在建立函數和 VPC 組態後，從執行角色中移除 AWSLambdaVPCAccessExecutionRole 政策。如需詳細資訊，請參閱所需的 IAM 許可。

若要為現有函數設定 VPC

若要將 VPC 組態新增至現有函數，函數的執行角色必須擁有建立和管理彈性網路介面的許可。AWSLambdaVPCAccessExecutionRole AWS 受管政策包含必要的許可。若要套用最低權限許可，可以在建立 VPC 組態後，從執行角色中移除 AWSLambdaVPCAccessExecutionRole 政策。

1. 開啟 Lambda 主控台中的 函數頁面。

2. 選擇一個函數。
3. 選擇組態索引標籤，然後選擇 VPC。
4. 在 VPC 下，選擇 Edit (編輯)。
5. 選取 VPC。
6. (選擇性) 若要允許傳出 IPv6 流量，請選取允許雙堆疊子網路的 IPv6 流量。
7. 對於子網路，選取所有私有子網路。私有子網路可透過 NAT 閘道存取網際網路。將函數連線到公有子網路並不會為其提供網際網路存取。

> **Note**
>
> 如果您選取允許雙堆疊子網路的 IPv6 流量，則所有選取的子網路均必須具有 IPv4 CIDR 區塊和 IPv6 CIDR 區塊。

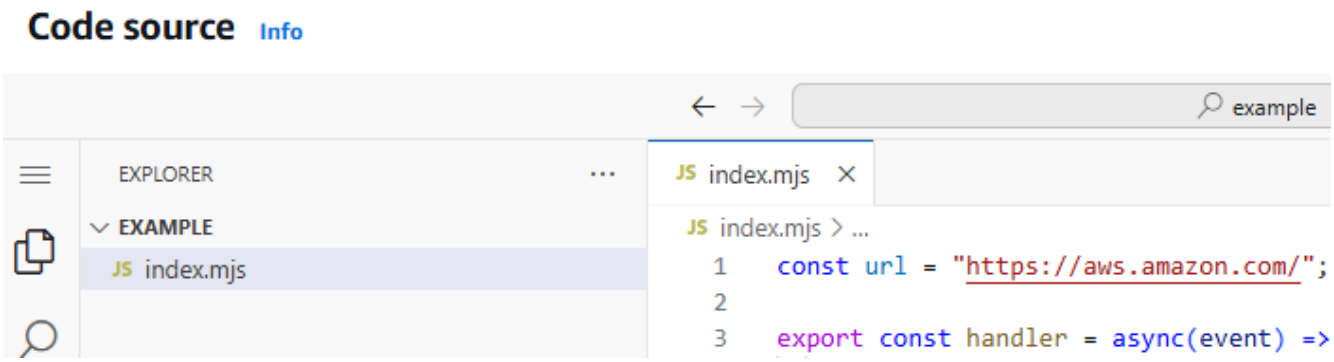
8. 對於安全群組，選取允許傳出流量的安全群組。
9. 選擇儲存。

測試函數

使用下列範例程式碼，確認 VPC 連線的函數可以連線到公有網際網路。如果成功，程式碼會傳回 200 狀態碼。如果失敗，函數會逾時。

Node.js

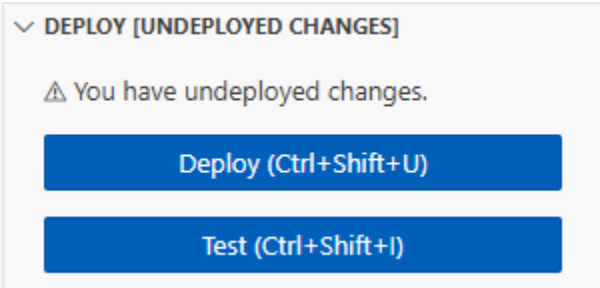
1. 在 Lambda 主控台的程式碼來源窗格中，將下列程式碼貼到 index.mjs 檔案中。函數會對公有端點發出 HTTP GET 請求，並傳回 HTTP 回應程式碼，以測試函數是否可存取公有網際網路。



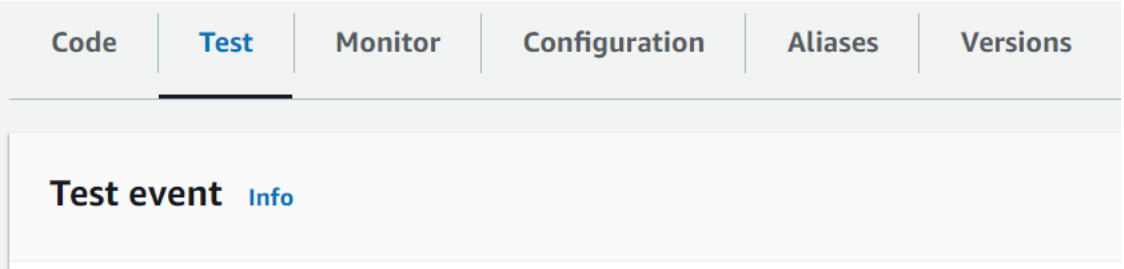
Example— 具有 async/await 的 HTTP 請求

```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

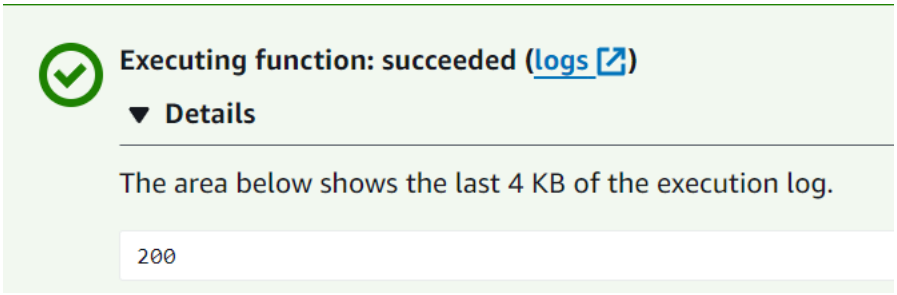
2. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



3. 選擇測試標籤。



4. 選擇測試。
5. 函數會傳回 200 狀態碼。這表示函數擁有傳出網際網路存取權。

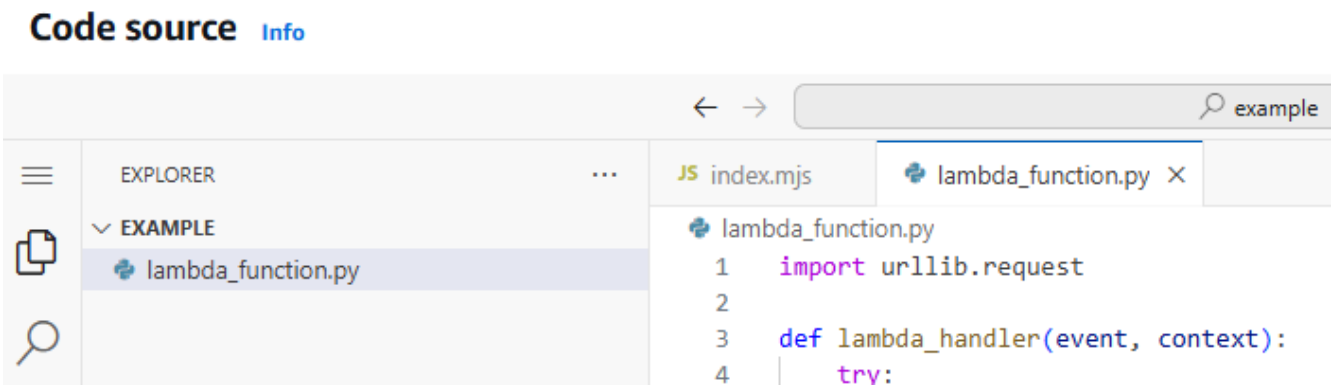


如果函數無法連線到公有網際網路，會收到類似以下的錯誤訊息：

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

Python

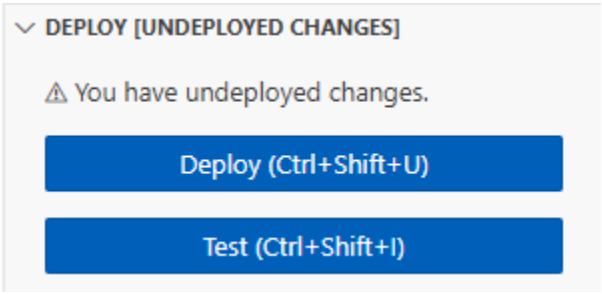
1. 在 Lambda 主控台的程式碼來源窗格中，將下列程式碼貼到 lambda_function.py 檔案中。函數會對公有端點發出 HTTP GET 請求，並傳回 HTTP 回應程式碼，以測試函數是否可存取公有網際網路。



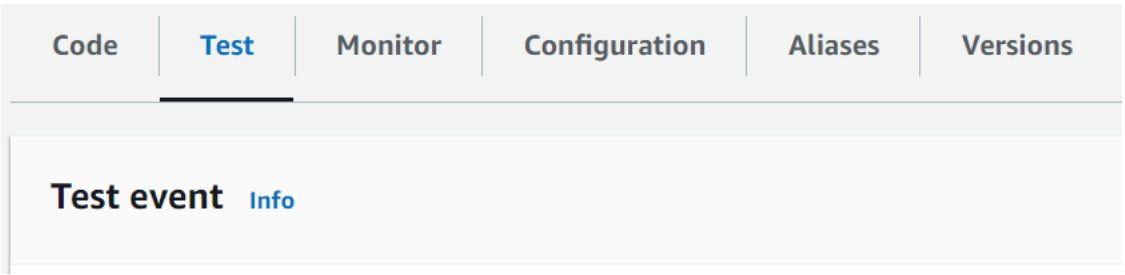

```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
```

```
        print('Error:', e)
        raise e
```

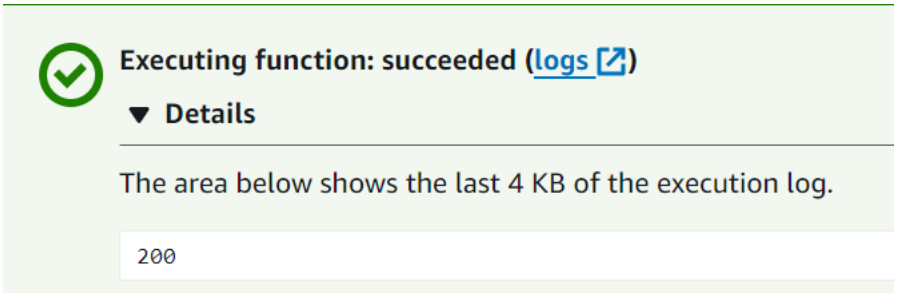
2. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



3. 選擇測試標籤。



4. 選擇測試。
5. 函數會傳回 200 狀態碼。這表示函數擁有傳出網際網路存取權。



如果函數無法連線到公有網際網路，會收到類似以下的錯誤訊息：

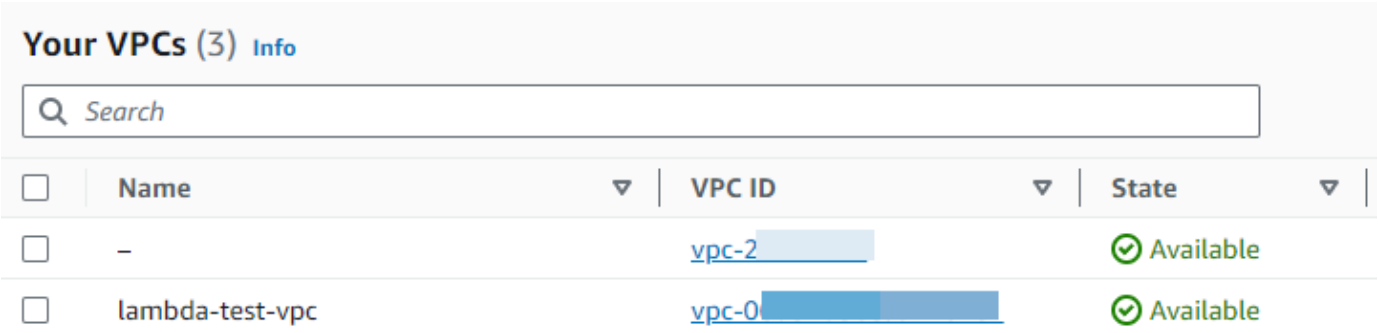
```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

我已擁有 VPC

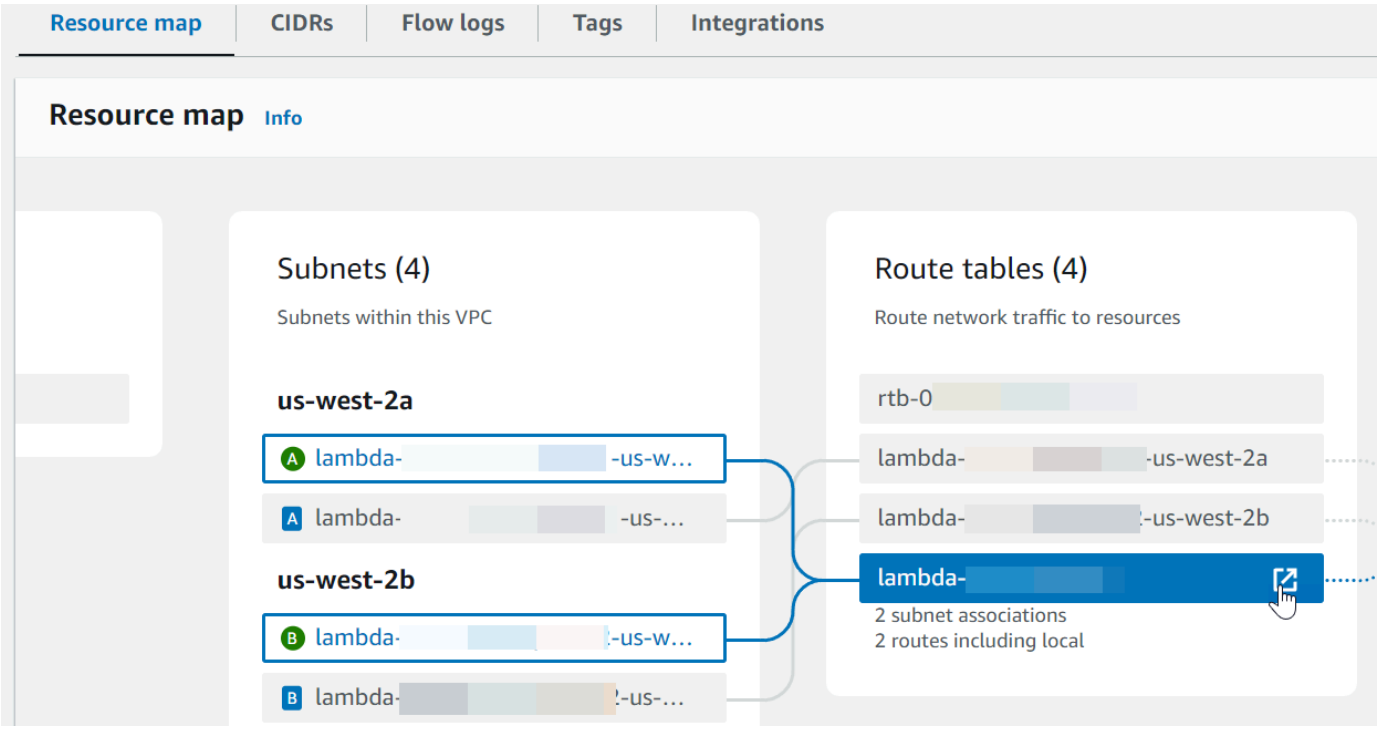
如果您已擁有 VPC，但需要設定 Lambda 函數的公有網際網路存取，請遵循下列步驟。此程序假設您的 VPC 擁有至少兩個子網路。如果沒有兩個子網路，請參閱《Amazon VPC 使用者指南》中的建立子網路。

驗證路由表組態

1. 在 https://console.aws.amazon.com/vpc/ 開啟 Amazon VPC 主控台。
2. 選擇 VPC ID。



3. 向下捲動至資源映射區段。請注意路由表映射。開啟映射至子網路的每個路由表。

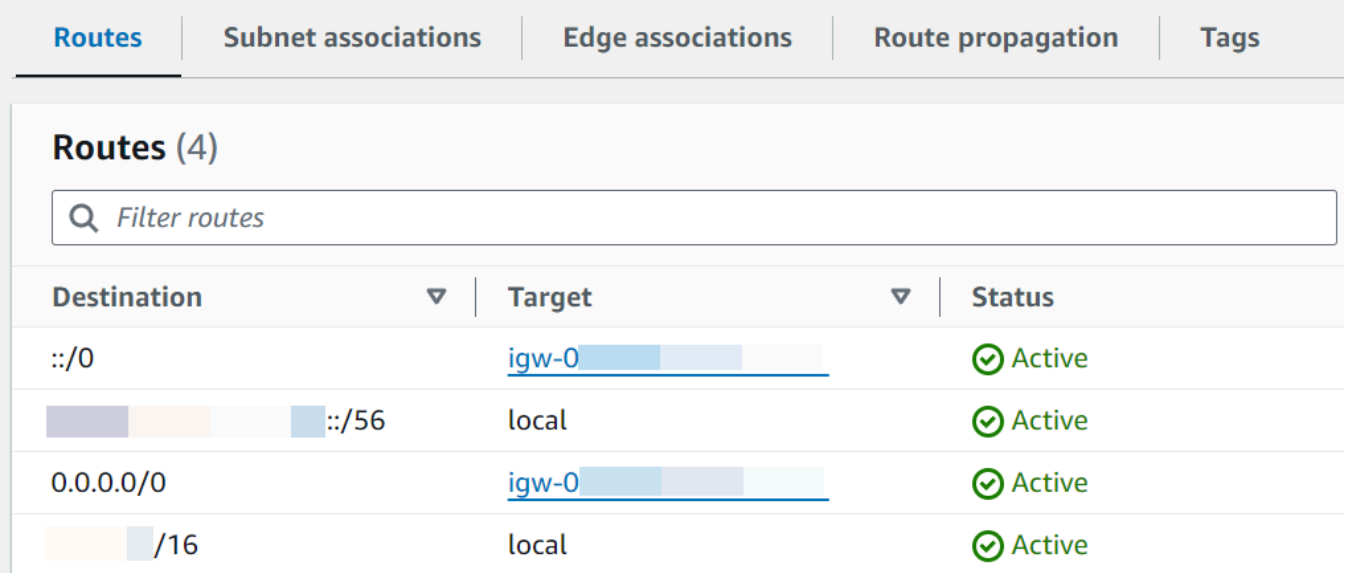


4. 向下捲動至路由索引標籤。檢閱路由設定，判斷 VPC 是否具有下列兩個路由表。每個需求都必須透過單獨的路由表來滿足。

- 網際網路流量 (IPv4 為 `0.0.0.0/0`、IPv6 為 `::/0`) 會路由至網際網路閘道 (`igw-xxxxxxxxxx`)。這意味著與路由表相關聯的子網路是公有子網路。

> **Note**
>
> 如果子網路沒有 IPv6 CIDR 區塊，則只能看到 IPv4 路由 (`0.0.0.0/0`)。

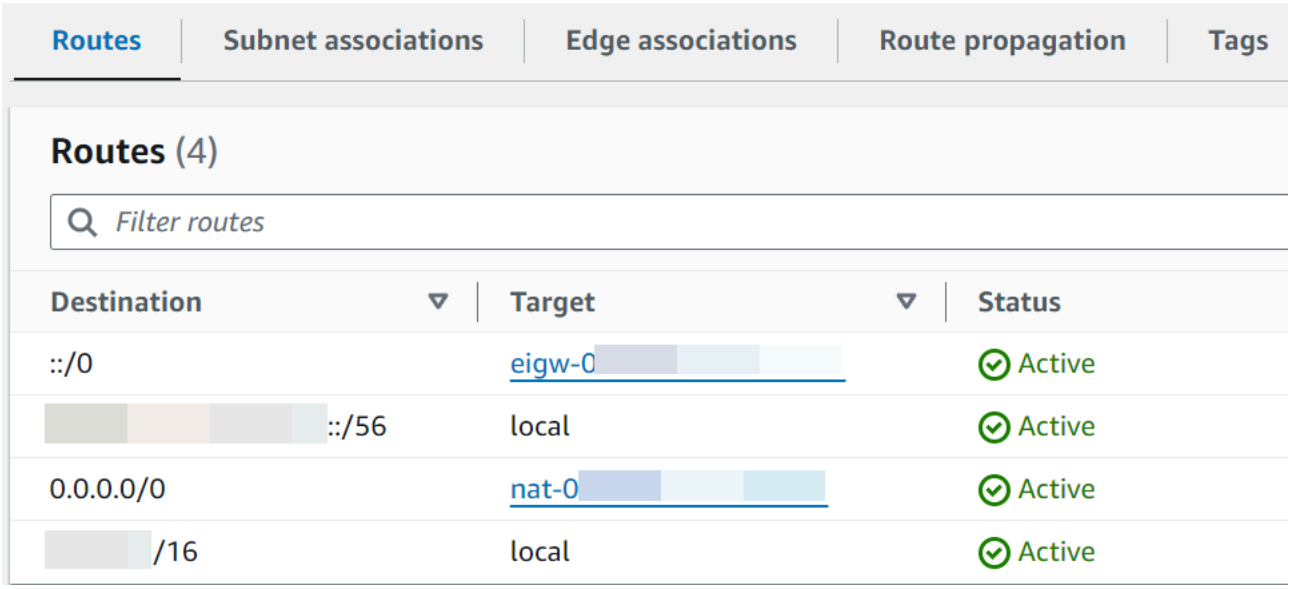
Example公有子網路路由表



- IPv4 (`0.0.0.0/0`) 的網際網路流量會路由至與公有子網路相關聯的 NAT 閘道 (`nat-xxxxxxxxxx`)。這意味著子網路是私有子網路，可透過 NAT 閘道存取網際網路。

> **Note**
>
> 如果子網路具有 IPv6 CIDR 區塊，路由表也必須將網際網路 IPv6 流量 (`::/0`) 路由至僅限輸出的網際網路閘道 (`eigw-xxxxxxxxxx`)。如果子網路沒有 IPv6 CIDR 區塊，則只能看到 IPv4 路由 (`0.0.0.0/0`)。

Example私有子網路路由表



5. 重複上述步驟，直到您已檢閱 VPC 中與子網路關聯的每個路由表，並確認您有一個具有網際網路閘道的路由表，以及一個具有 NAT 閘道的路由表。

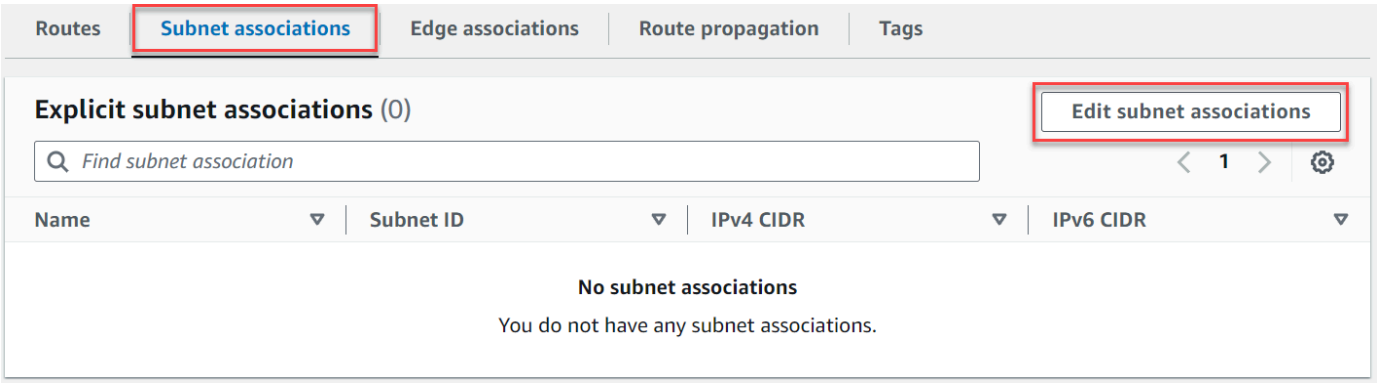
   如果沒有兩個路由表，其中一個可路由到網際網路閘道，另一個可路由到 NAT 閘道，請依照下列步驟建立缺少的資源和路由表項目。

建立路由表

請依照下列步驟建立路由表，並將其與子網路關聯。

若要使用 Amazon VPC 主控台建立自訂路由表

1. 在 https://console.aws.amazon.com/vpc/ 開啟 Amazon VPC 主控台。
2. 在導覽窗格中，選擇 Route tables (路由表)。
3. 選擇 Create route table (建立路由表)。
4. (選用) 針對 Name (名稱)，輸入路由表的名稱。
5. 在 VPC 中，選擇您的 VPC。
6. (選用) 若要新增標籤，請選擇 Add new tag (新增標籤)，然後輸入標籤鍵和標籤值。
7. 選擇 Create route table (建立路由表)。
8. 在 Subnet associations (子網關聯) 標籤上，選擇 Edit subnet associations (編輯子網關聯)。



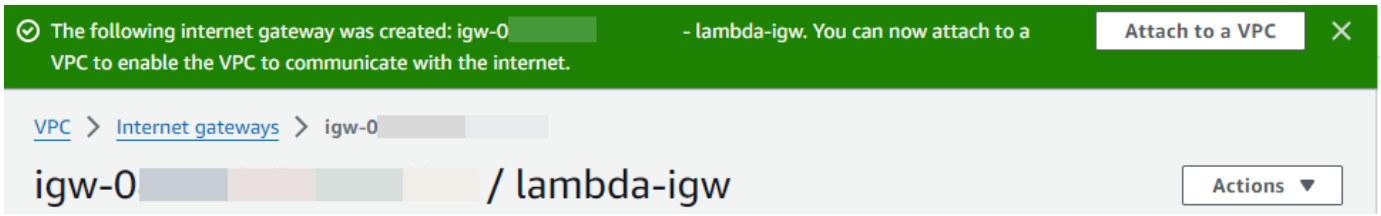
9. 選取子網路的核取方塊以和路由表建立關聯。
10. 選擇 Save associations (儲存關聯)。

建立網際網路閘道

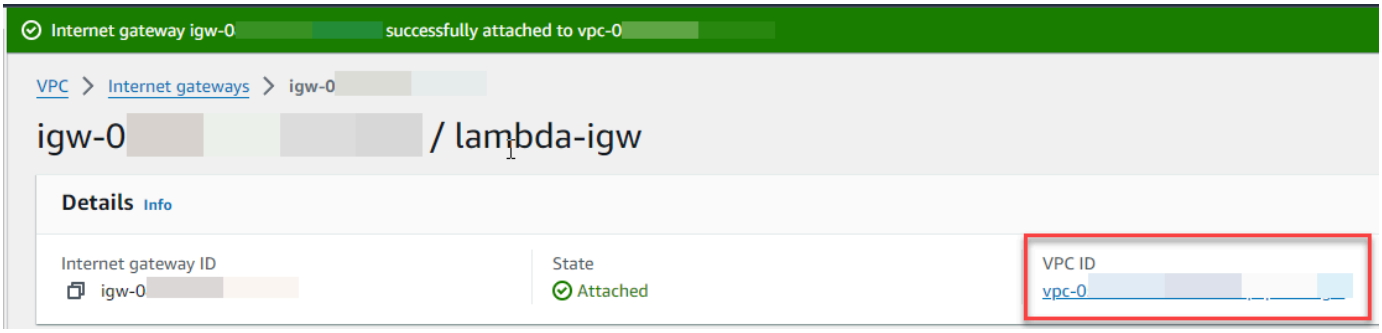
請依照下列步驟建立網際網路閘道，將其連接至 VPC，並將其新增至公有子網路的路由表。

建立網際網路閘道

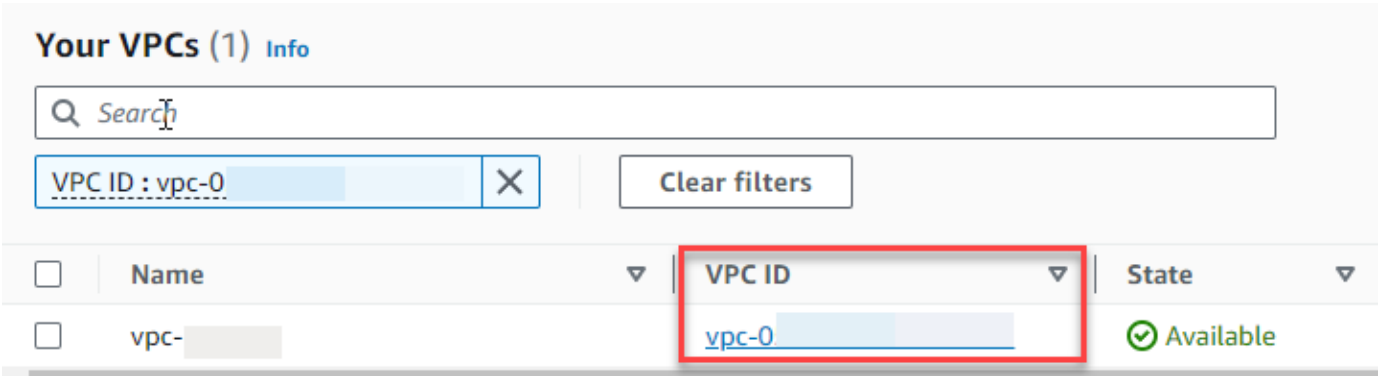
1. 在 https://console.aws.amazon.com/vpc/ 開啟 Amazon VPC 主控台。
2. 在導覽窗格中，選擇 Internet gateways (網際網路閘道)。
3. 選擇建立網際網路閘道。
4. (可選) 輸入網際網路閘道的名稱。
5. (選用) 若要新增標籤，請選擇 Add new tag (新增標籤)，然後輸入標籤金鑰和值。
6. 選擇建立網際網路閘道。
7. 從畫面頂端的橫幅中選擇附接至 VPC，選取可用的 VPC，然後選擇附接網際網路閘道。



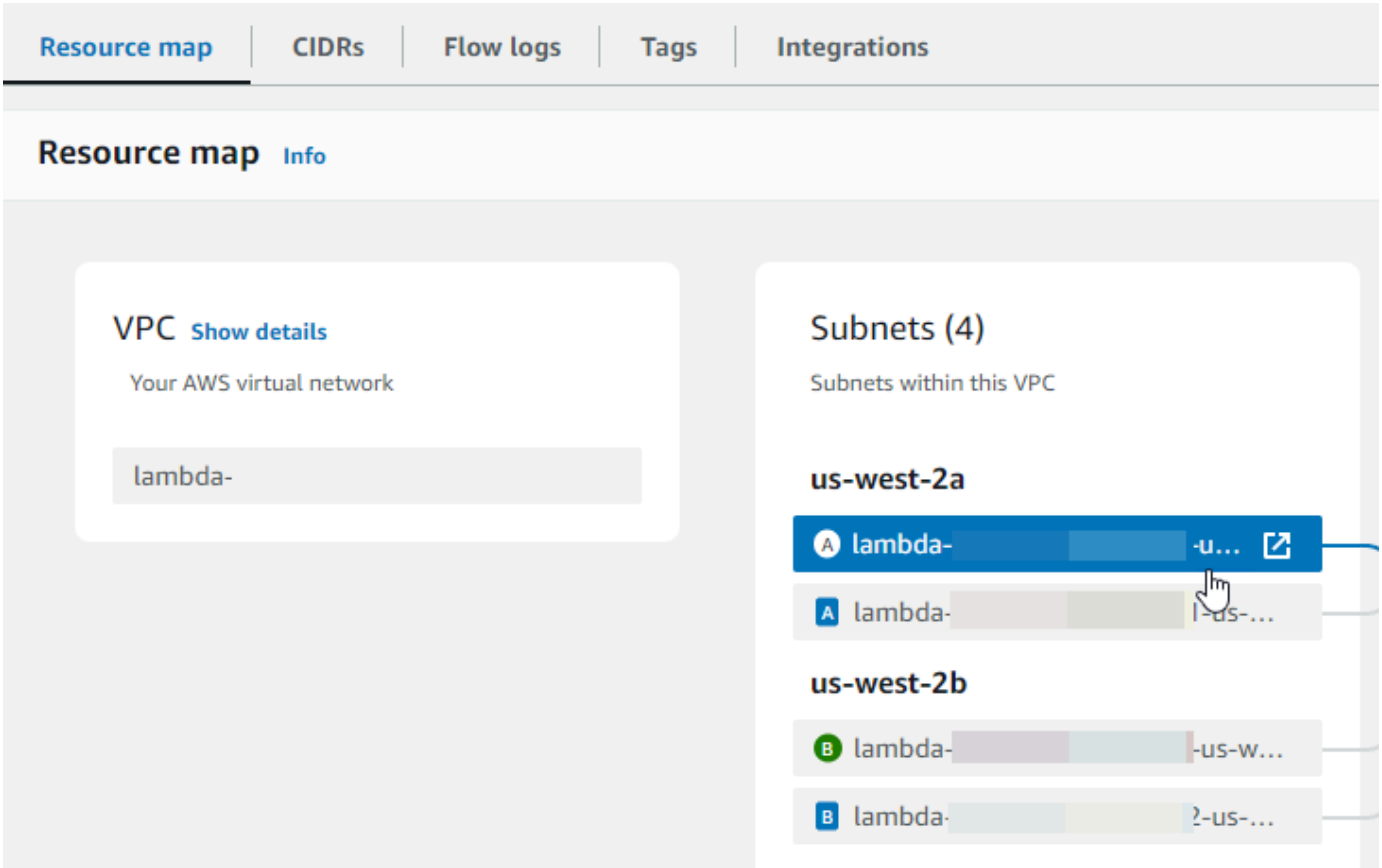
8. 選擇 VPC ID。



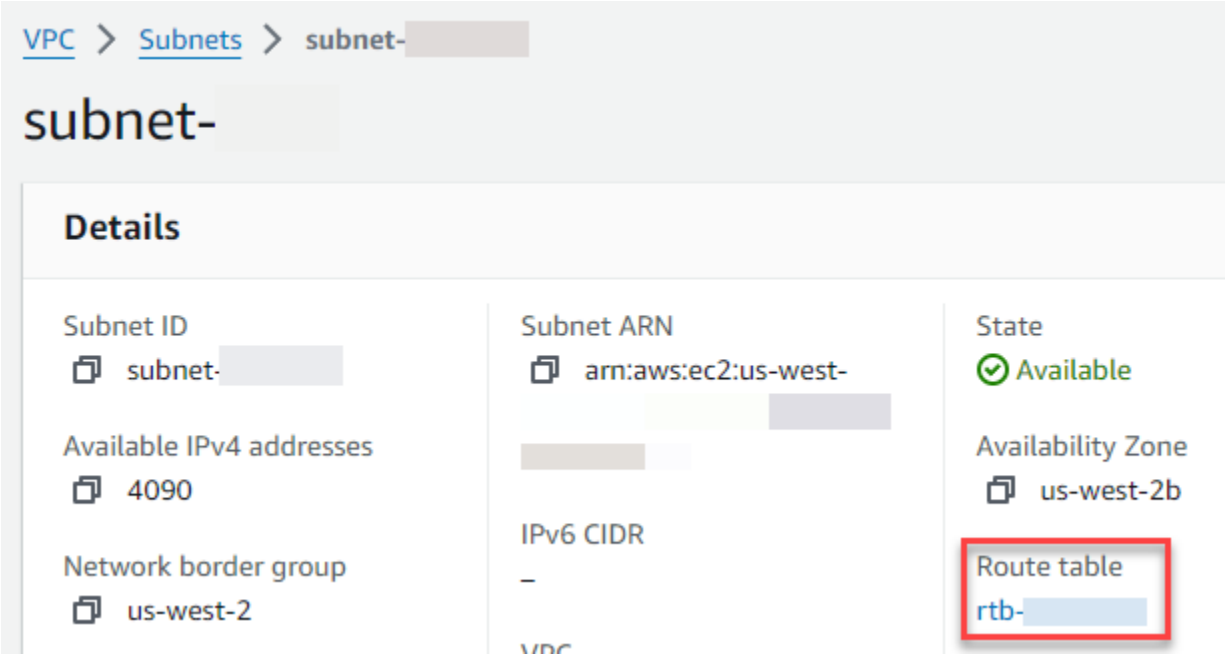
9. 再次選擇 VPC ID 以開啟詳細資訊頁面。



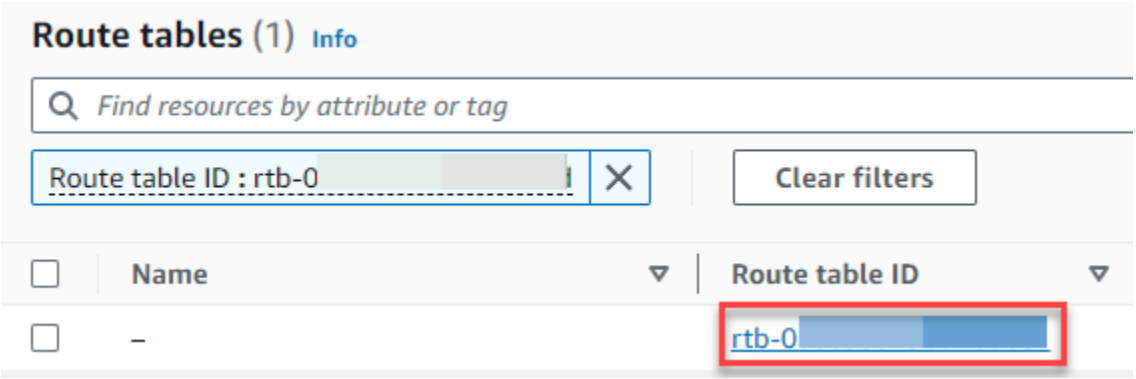
10. 向下捲動至資源映射區段，然後選擇子網路。子網路詳細資訊會顯示在新索引標籤中。



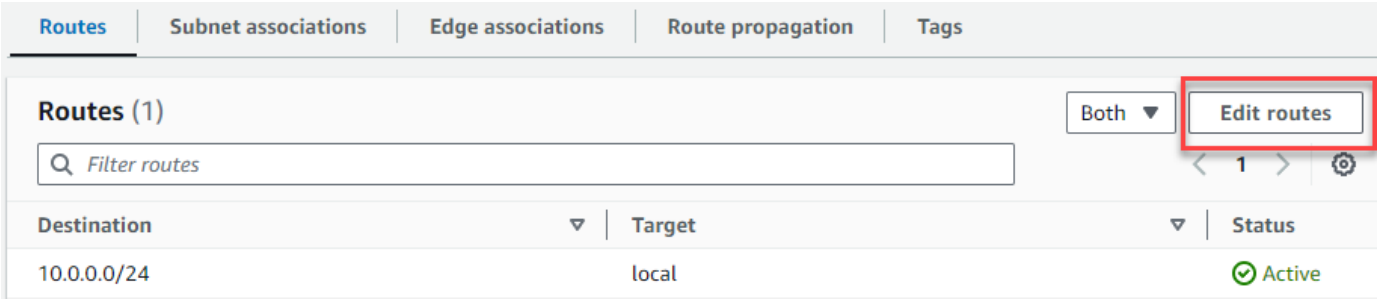
11. 選擇路由表下方的連結。



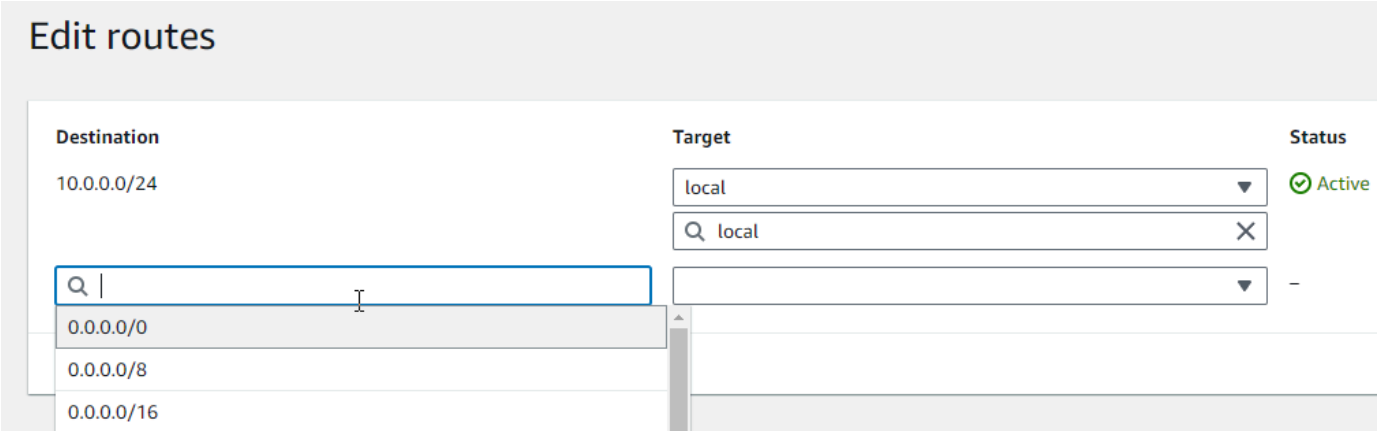
12. 選擇路由表 ID 以開啟路由表詳細資訊頁面。



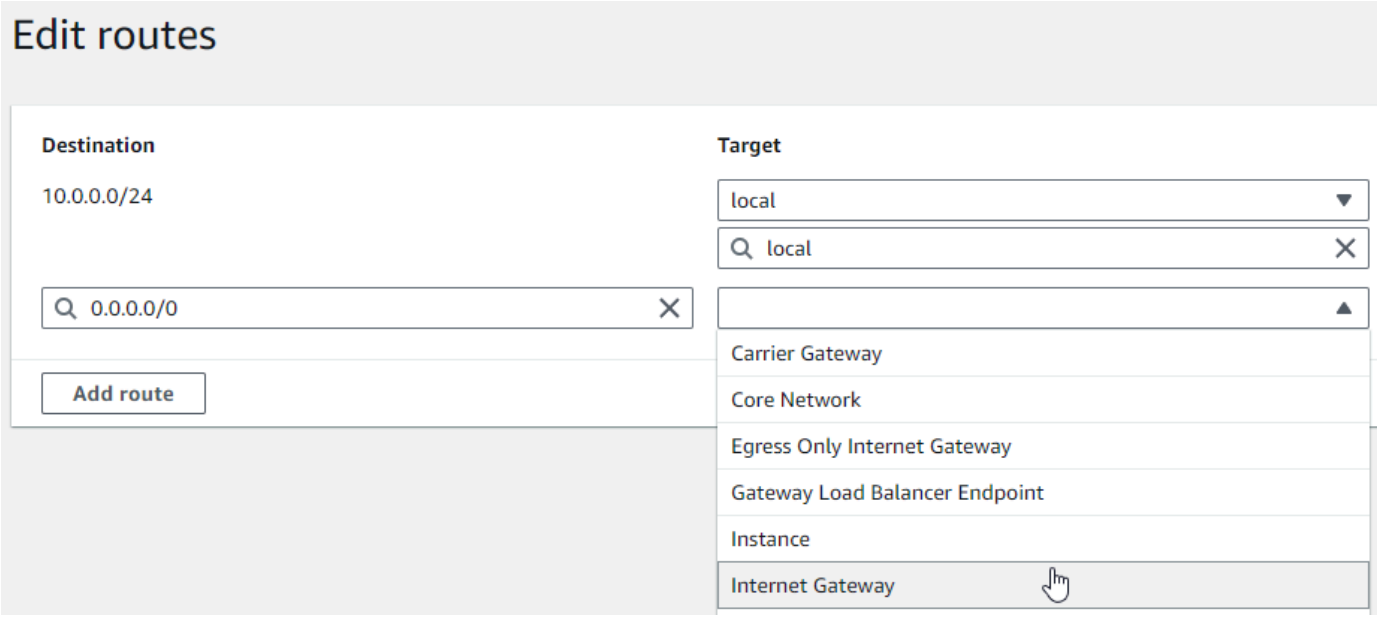
13. 在路由下，選擇編輯路由。



14. 選擇新增路由，然後在目的地方塊中輸入 `0.0.0.0/0`。



15. 對於目標，選取網際網路閘道，然後選擇先前建立的網際網路閘道。如果子網路具有 IPv6 CIDR 區塊，還必須將 `::/0` 的路由新增至相同的網際網路閘道。



16. 選擇儲存變更。

建立 NAT 閘道

請依照下列步驟建立 NAT 閘道，將其與公有子網路建立關聯，然後將其新增至私有子網路的路由表。

若要建立 NAT 閘道並將其與公有子網路建立關聯

1. 在導覽窗格中，選擇 NAT 閘道。
2. 選擇建立 NAT 閘道。
3. (可選) 輸入 NAT 閘道的名稱。

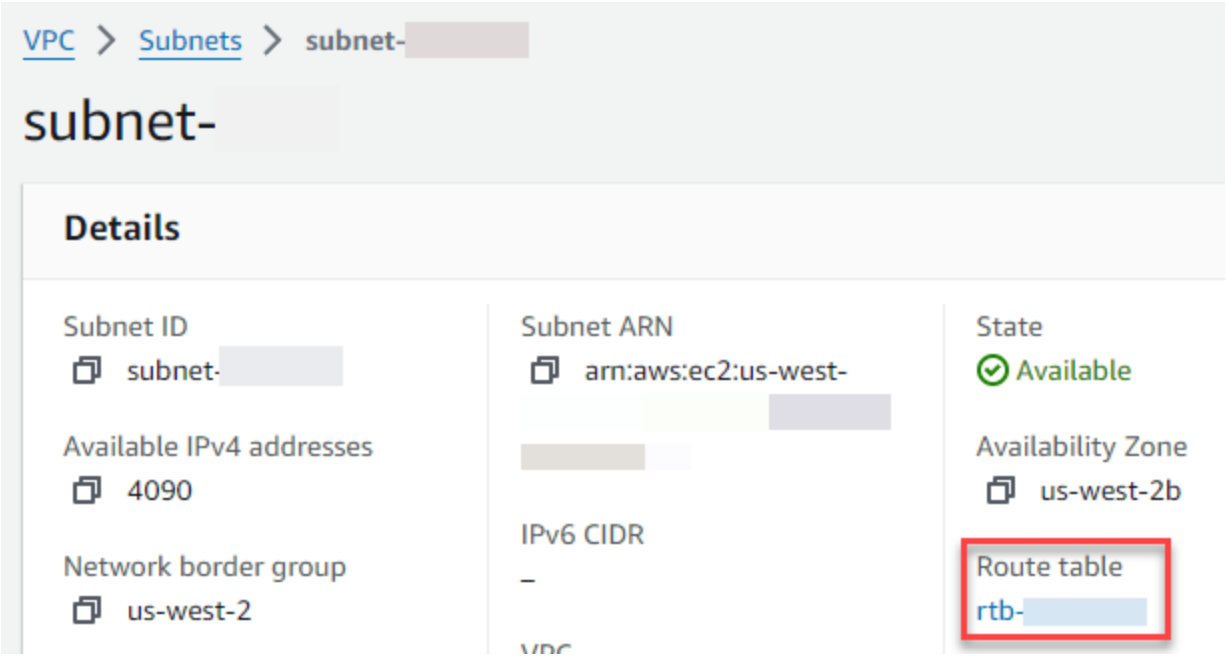
4. 對於子網路，選取 VPC 中的公有子網路。(公有子網路是指可直接路由至其路由表中網際網路閘道的子網路。)

> **Note**
>
> NAT 閘道與公有子網路關聯，但路由表項目位於私有子網路中。

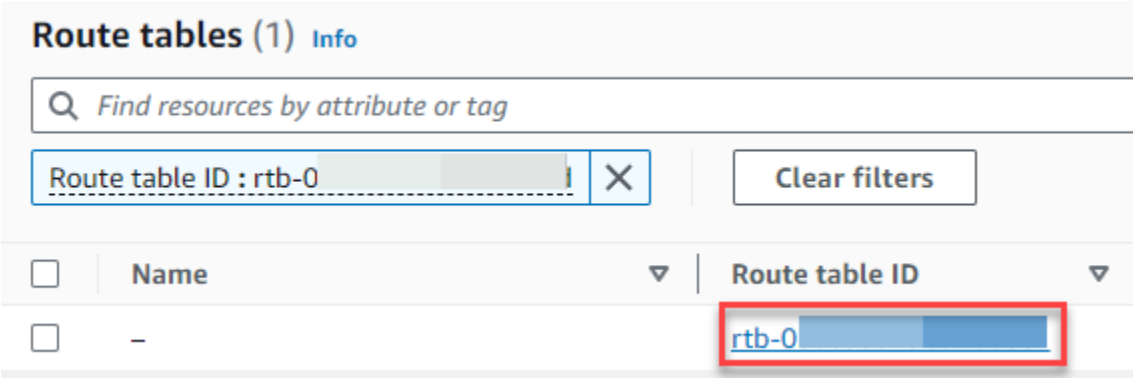
5. 對於彈性 IP 配置 ID，選取彈性 IP 位址，或選擇配置彈性 IP。
6. 選擇建立 NAT 閘道。

若要將路由新增至私有子網路路由表中的 NAT 閘道

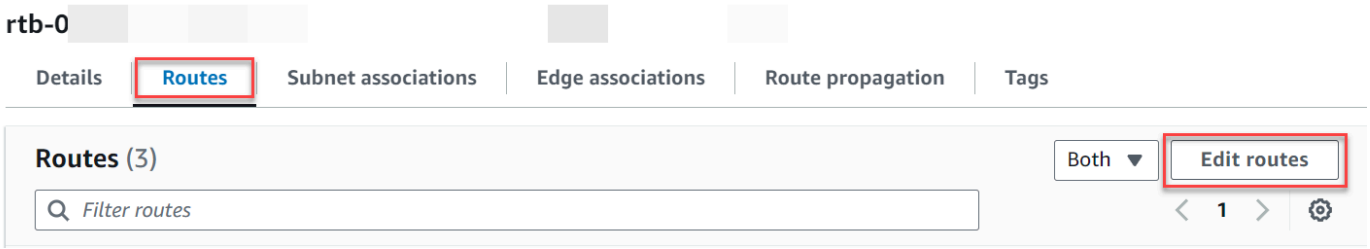
1. 在導覽窗格中，選擇 Subnets (子網)。
2. 選取 VPC 中的私有子網路。(私有子網路是指不可路由至其路由表中網際網路閘道的子網路。)
3. 選擇路由表下方的連結。



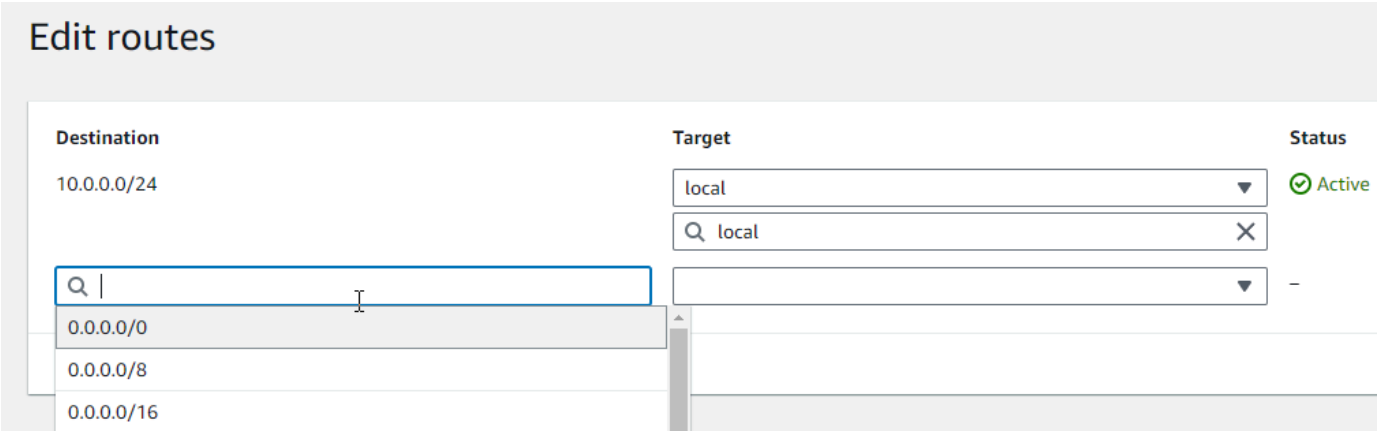
4. 選擇路由表 ID 以開啟路由表詳細資訊頁面。



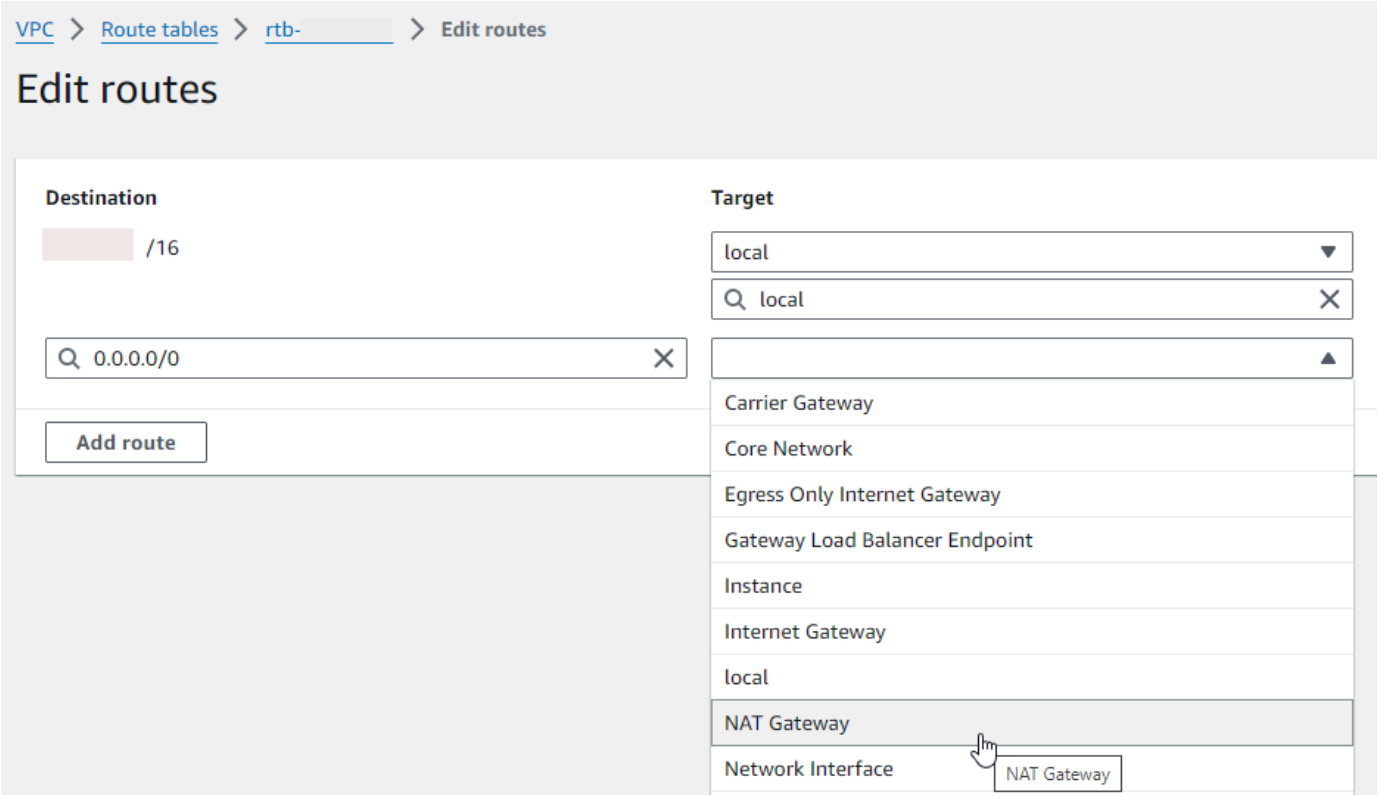
5. 向下捲動並選擇路由索引標籤，然後選擇編輯路由



6. 選擇新增路由，然後在目的地方塊中輸入 `0.0.0.0/0`。



7. 對於目標，選取 NAT 閘道，然後選擇先前建立的 NAT 閘道。



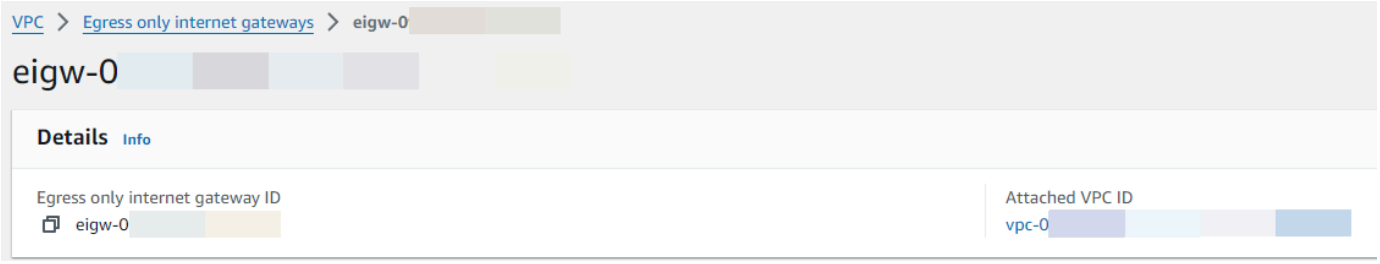
8. 選擇儲存變更。

建立僅限輸出的網際網路閘道 (僅限 IPv6)

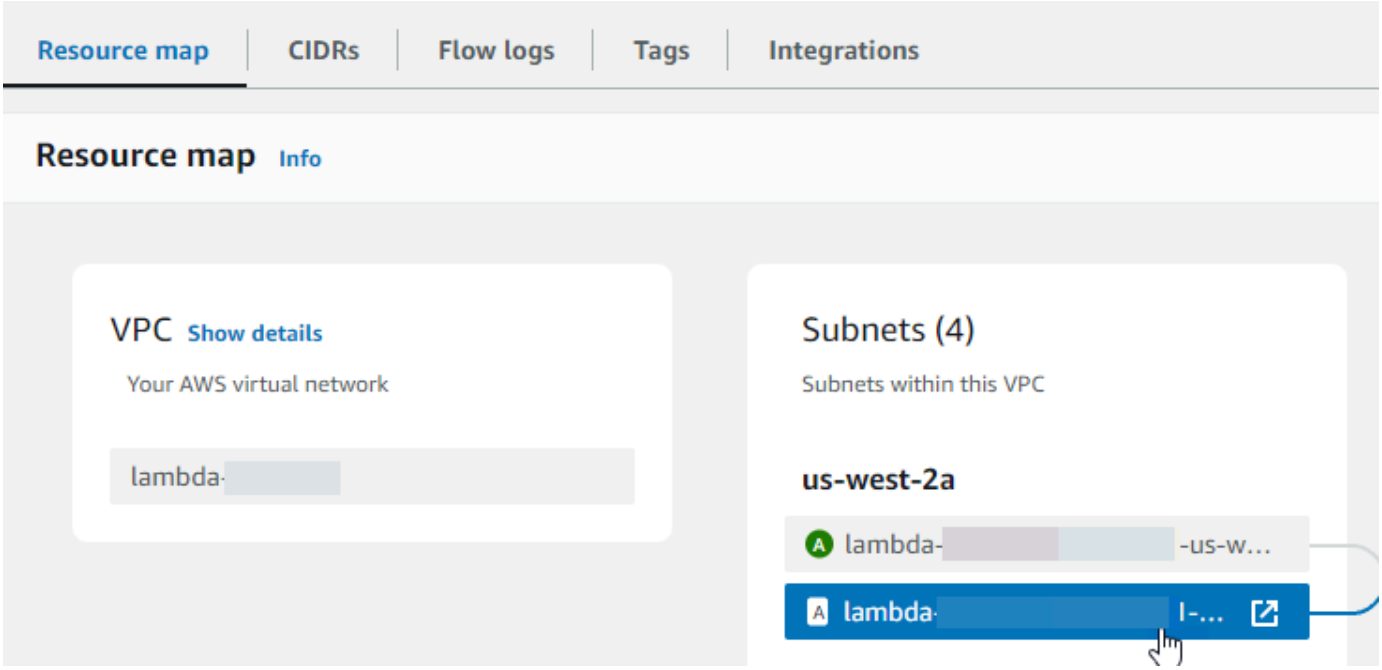
請依照下列步驟建立僅限輸出的網際網路閘道，並將其新增至私有子網路的路由表。

建立輸出限定網際網路閘道

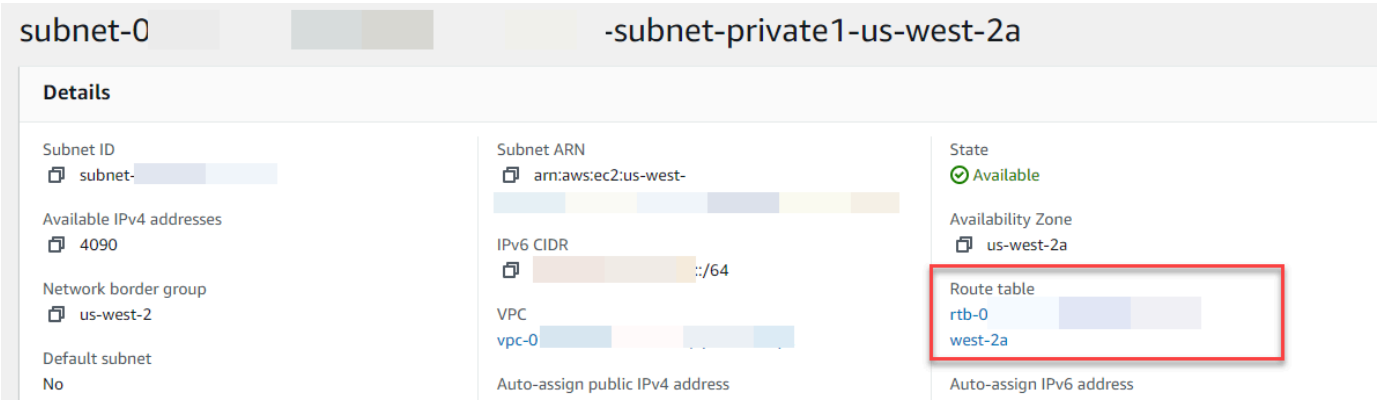
1. 在導覽窗格中，選擇輸出限定網際網路閘道。
2. 選擇建立輸出限定網際網路閘道。
3. (選用) 輸入名稱。
4. 選取要在其中建立輸出限定網際網路閘道的 VPC。
5. 選擇建立輸出限定網際網路閘道。
6. 選擇已附接 VPC ID 下方的連結。



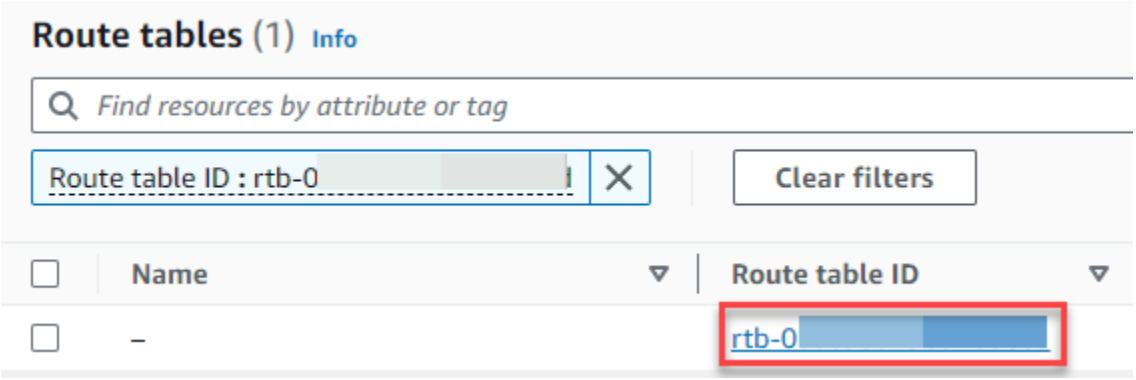
7. 選擇 VPC ID 下方的連結，以開啟 VPC 詳細資訊頁面。
8. 向下捲動至資源映射區段，然後選擇私有子網路。(私有子網路是指不可路由至其路由表中網際網路閘道的子網路。) 子網路詳細資訊會顯示在新索引標籤中。



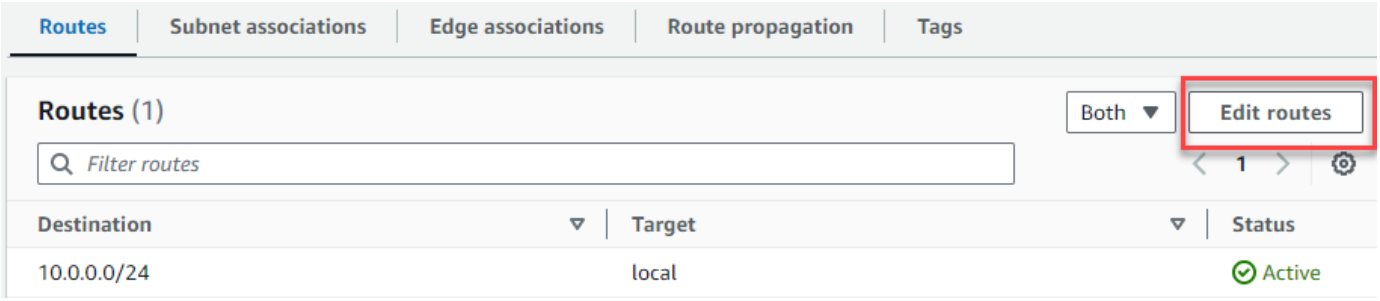
9. 選擇路由表下方的連結。



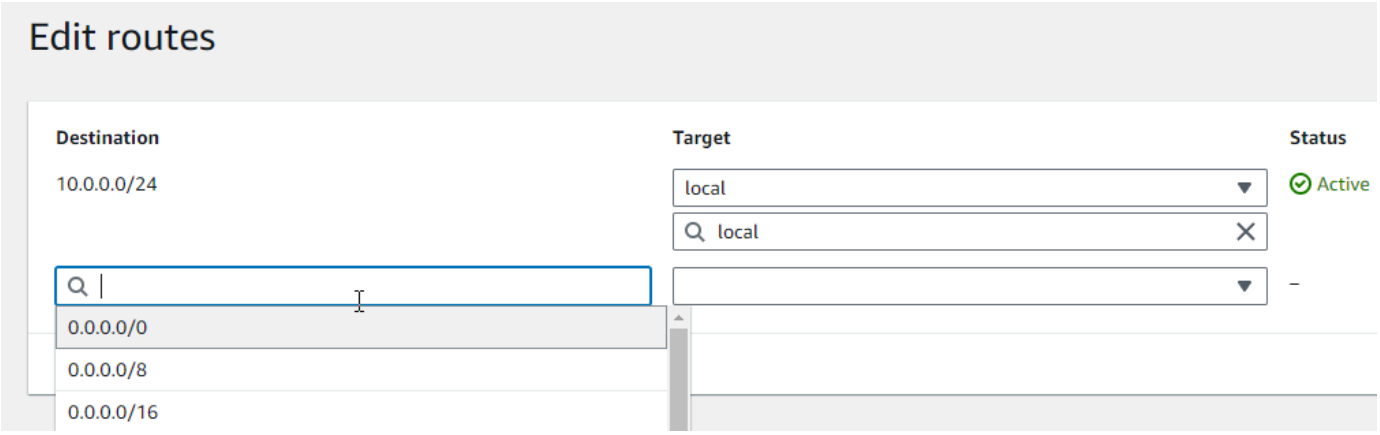
10. 選擇路由表 ID 以開啟路由表詳細資訊頁面。



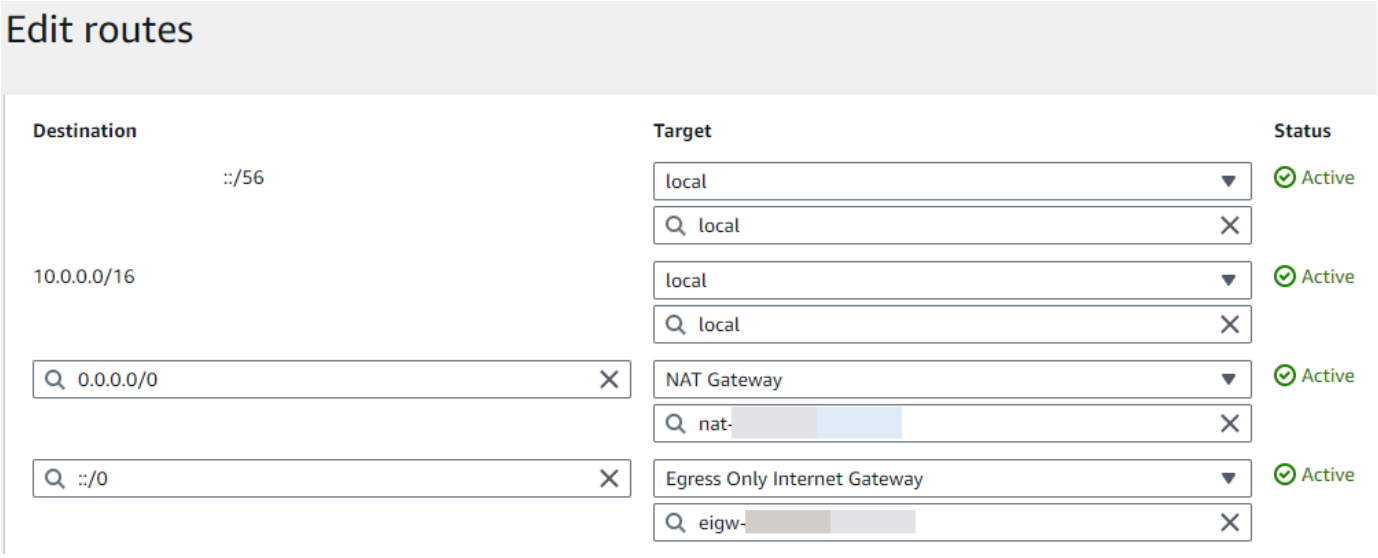
11. 在路由下，選擇編輯路由。



12. 選擇新增路由，然後在目的地方塊中輸入 `::/0`。



13. 對於目標，選取僅限輸出的網際網路閘道，然後選擇先前建立的閘道。



14. 選擇儲存變更。

設定 Lambda 函數

若要在建立函數時設定 VPC

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇建立函數。
3. 在 Basic information (基本資訊) 下，對於 Function name (函數名稱)，為您的函數輸入名稱。
4. 展開 Advanced settings (進階設定)。
5. 選取啟用 VPC，然後選擇一個 VPC。
6. (選擇性) 若要允許傳出 IPv6 流量，請選取允許雙堆疊子網路的 IPv6 流量。
7. 對於子網路，選取所有私有子網路。私有子網路可透過 NAT 閘道存取網際網路。將函數連線到公有子網路並不會為其提供網際網路存取。

> **Note**
>
> 如果您選取允許雙堆疊子網路的 IPv6 流量，則所有選取的子網路均必須具有 IPv4 CIDR 區塊和 IPv6 CIDR 區塊。

8. 對於安全群組，選取允許傳出流量的安全群組。
9. 選擇建立函數。

Lambda 會使用 AWSLambdaVPCAccessExecutionRole AWS 受管政策自動建立執行角色。此政策中的許可僅適用於為 VPC 組態建立彈性網路介面，而非調用函數。若要套用最低權限許可，可以在建立函數和 VPC 組態後，從執行角色中移除 AWSLambdaVPCAccessExecutionRole 政策。如需詳細資訊，請參閱所需的 IAM 許可。

若要為現有函數設定 VPC

若要將 VPC 組態新增至現有函數，函數的執行角色必須擁有建立和管理彈性網路介面的許可。AWSLambdaVPCAccessExecutionRole AWS 受管政策包含必要的許可。若要套用最低權限許可，可以在建立 VPC 組態後，從執行角色中移除 AWSLambdaVPCAccessExecutionRole 政策。

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態索引標籤，然後選擇 VPC。
4. 在 VPC 下，選擇 Edit (編輯)。
5. 選取 VPC。
6. (選擇性) 若要允許傳出 IPv6 流量，請選取允許雙堆疊子網路的 IPv6 流量。
7. 對於子網路，選取所有私有子網路。私有子網路可透過 NAT 閘道存取網際網路。將函數連線到公有子網路並不會為其提供網際網路存取。

> **Note**
>
> 如果您選取允許雙堆疊子網路的 IPv6 流量，則所有選取的子網路均必須具有 IPv4 CIDR 區塊和 IPv6 CIDR 區塊。

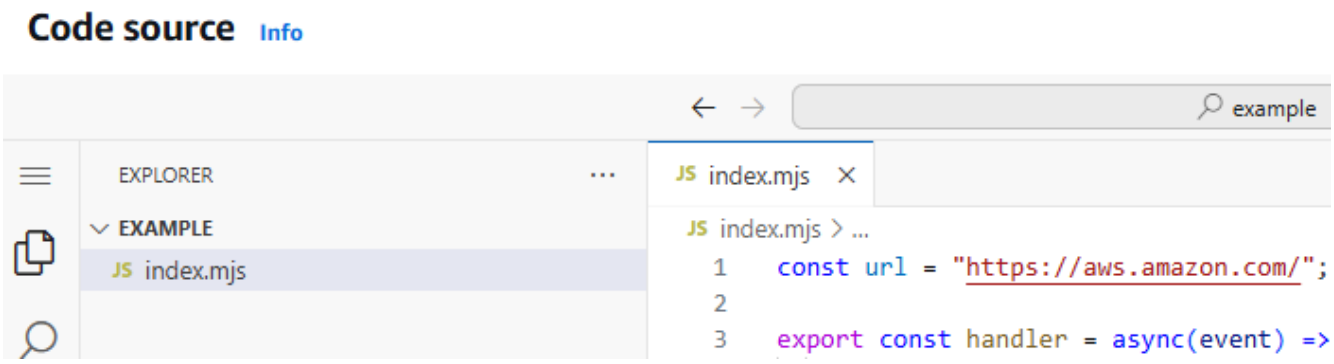
8. 對於安全群組，選取允許傳出流量的安全群組。
9. 選擇儲存。

測試函數

使用下列範例程式碼，確認 VPC 連線的函數可以連線到公有網際網路。如果成功，程式碼會傳回 200 狀態碼。如果失敗，函數會逾時。

Node.js

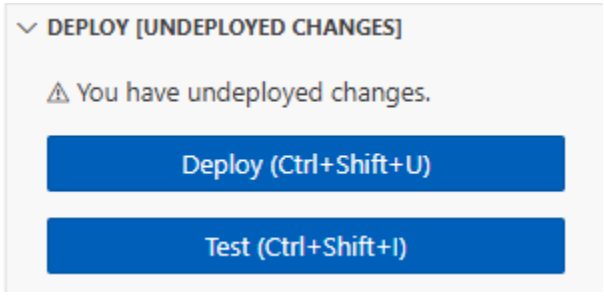
1. 在 Lambda 主控台的程式碼來源窗格中，將下列程式碼貼到 index.mjs 檔案中。函數會對公有端點發出 HTTP GET 請求，並傳回 HTTP 回應程式碼，以測試函數是否可存取公有網際網路。



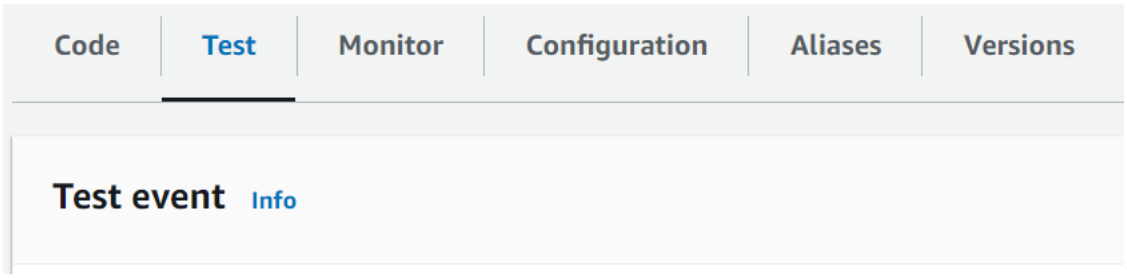
Example— 具有 async/await 的 HTTP 請求

```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

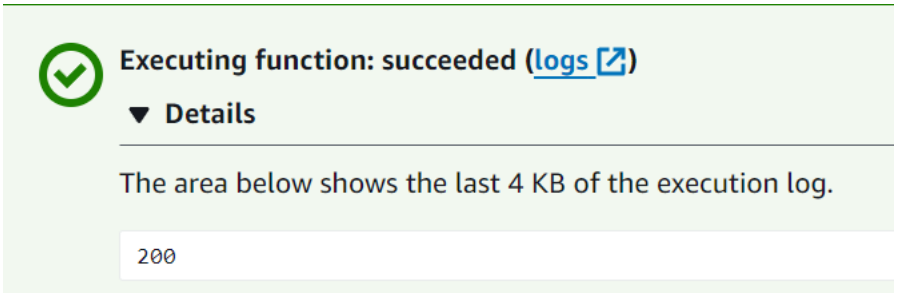
2. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



3. 選擇測試標籤。



4. 選擇測試。
5. 函數會傳回 200 狀態碼。這表示函數擁有傳出網際網路存取權。

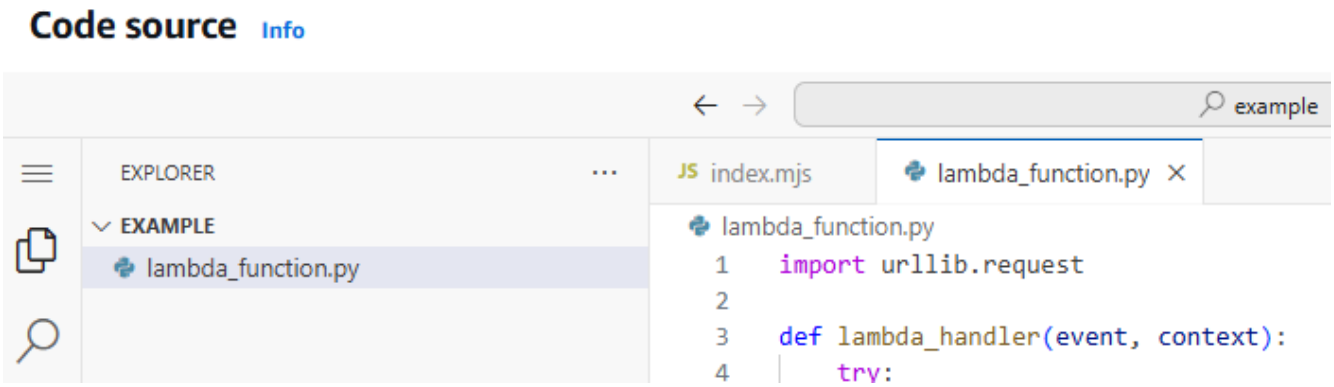


如果函數無法連線到公有網際網路，會收到類似以下的錯誤訊息：

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

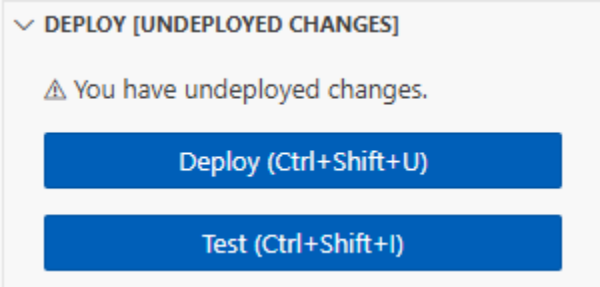
Python

1. 在 Lambda 主控台的程式碼來源窗格中，將下列程式碼貼到 lambda_function.py 檔案中。函數會對公有端點發出 HTTP GET 請求，並傳回 HTTP 回應程式碼，以測試函數是否可存取公有網際網路。

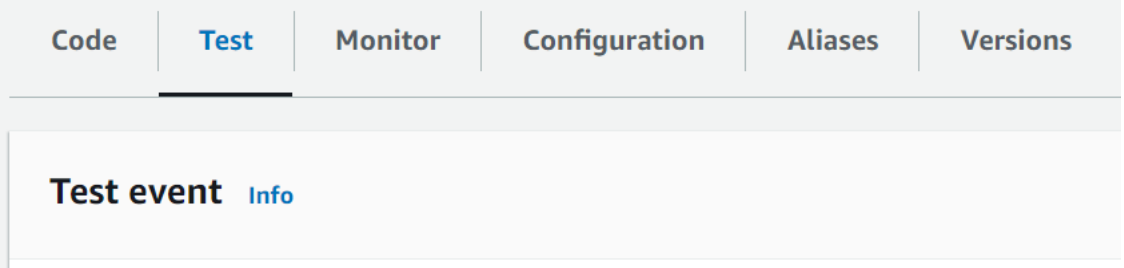


```
import urllib.request

def lambda_handler(event, context):
    try:
        response = urllib.request.urlopen('https://aws.amazon.com')
        status_code = response.getcode()
        print('Response Code:', status_code)
        return status_code
    except Exception as e:
        print('Error:', e)
        raise e
```

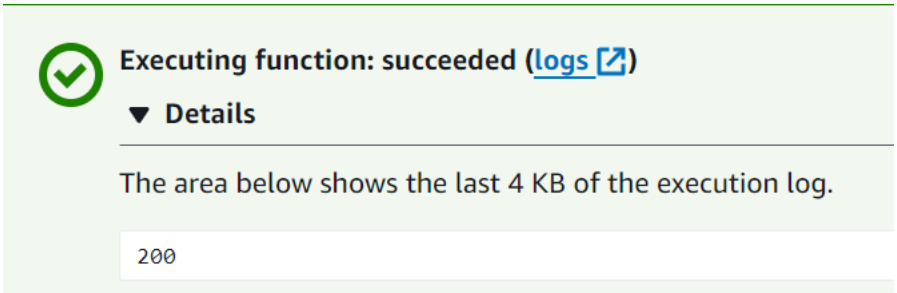
2. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



3. 選擇測試標籤。



4. 選擇測試。
5. 函數會傳回 200 狀態碼。這表示函數擁有傳出網際網路存取權。



如果函數無法連線到公有網際網路，會收到類似以下的錯誤訊息：

```
{
  "errorMessage": "2024-04-11T17:22:20.857Z abe12jlc-640a-8157-0249-9be825c2y110
 Task timed out after 3.01 seconds"
}
```

連線 Lambda 的傳入介面 VPC 端點

如果您使用 Amazon Virtual Private Cloud (Amazon VPC) 來託管您的 AWS 資源，可以在您的 VPC 與 Lambda 之間建立連線。您可以使用此連線來叫用您的 Lambda 函數，而不需要透過公有網際網路。

若要在 VPC 與 Lambda 之間建立私有連線，請建立介面 VPC 端點。介面端點由 AWS PrivateLink 支援，讓您不需要網際網路閘道、NAT 裝置、VPN 連線或 AWS Direct Connect 連線即可私有存取 Lambda API。VPC 中的執行個體不需要公有 IP 地址，即能與 Lambda API 通訊。您的 VPC 與 Lambda 之間的網路流量都不會離開 AWS 網路範圍。

每個介面端點都由子網路中的一個或多個彈性網路介面表示。網路介面會提供一個私有 IP 地址，以作為連至 Lambda 的流量進入點。

章節

- Lambda 介面端點的考量
- 為 Lambda 建立介面端點
- 為 Lambda 建立介面端點政策

Lambda 介面端點的考量

在設定 Lambda 的介面端點之前，請務必檢閱 Amazon VPC 使用者指南中的介面端點屬性和限制。

您可以從 VPC 呼叫任何 Lambda API 操作。例如，您可以從 VPC 內呼叫 `Invoke` API 來叫用 Lambda 函數。如需完整的 Lambda API 清單，請參閱 Lambda API 參考中的動作。

`use1-az3` 是 Lambda VPC 函數的有限容量區域。您不應將此可用區域中的子網路與 Lambda 函數搭配使用，因為這可能會在出現中斷狀況時導致區域備援減少。

用於持續連線的 keep-alive (保持啟用)

Lambda 一段時間後會清除閒置連線，因此您必須使用實時指示來維持持續連線。叫用函式時，嘗試重複使用閒置連線會導致連線錯誤。若要維護持續連線，請使用與執行階段相關聯的 keep-alive (保持啟用) 指令。如需範例，請參閱《適用於 JavaScript 的 AWS SDK 開發人員指南》中的在 Node.js 中重複使用 Keep-Alive 的連線。

計費考量

透過介面端點來存取 Lambda 函數無需額外成本。如需 Lambda 定價的詳細資訊，請參閱 AWS Lambda 定價。

AWS PrivateLink 的標準定價適用於 Lambda 的介面端點。對於在每個可用區域中佈建的介面端點，以及透過介面端點處理的資料，都會按每小時向您的 AWS 帳戶收費。如需介面端點定價的詳細資訊，請參閱 AWS PrivateLink 定價。

VPC 對等互連考量

您可以使用 VPC 對等互連，將其他 VPC 連線到具有介面端點的 VPC。VPC 對等互連是兩個 VPC 之間的網路連線。您可以在自己的 VPC 之間建立 VPC 對等互連的連線，或與其他 AWS 帳戶的 VPC 建立對等互連的連線。VPC 也可以位於兩個不同的 AWS 區域。

對等互連的 VPC 之間的流量會保留在 AWS 網路上，且不會周遊公用網際網路。VPC 對等互連後，兩個 VPC 中的 Amazon Elastic Compute Cloud (Amazon EC2) 執行個體、Amazon Relational Database Service (Amazon RDS) 執行個體或啟用 VPC 功能的 Lambda 函數等這類資源就可以透過在其中一個 VPC 中建立的介面端點來存取 Lambda API。

為 Lambda 建立介面端點

您可使用 Amazon VPC 主控台或 AWS Command Line Interface (AWS CLI) 來為 Lambda 建立介面端點。如需詳細資訊，請參閱《Amazon VPC 使用者指南》中的建立介面端點。

若要為 Lambda 建立介面端點 (主控台)

1. 開啟 Amazon VPC 主控台的 Endpoints (端點) 頁面。
2. 選擇建立端點。
3. 對於服務類別，請確定已選擇 AWS 服務。
4. 在服務名稱中，選擇 com.amazonaws.***region***.lambda。確認類型為介面。
5. 建立 VPC 和子網路
6. 若要啟用介面端點的私有 DNS，請選取 Enable DNS Name (啟用 DNS 名稱) 核取方塊。建議您為 AWS 服務 的 VPC 端點啟用私有 DNS 名稱。這可確保使用公有服務端點的請求 (例如透過 AWS SDK 提出的請求) 可解析至您的 VPC 端點。
7. 對於安全群組，選擇一或多個安全群組。
8. 選擇建立端點。

若要使用私有 DNS 選項，您必須設定 VPC 的 `enableDnsHostnames` 和 `enableDnsSupportattributes`。如需詳細資訊，請參閱 Amazon VPC 使用者指南中的檢視並更新 VPC 的 DNS 支援。如果您為該介面端點啟用私有 DNS，您可以使用其區域的預設 DNS 名稱 (例

如 lambda.us-east-1.amazonaws.com)，向 Lambda 發出 API 請求。如需更多服務端點，請參閱《AWS 一般參考》中的服務端點和配額。

如需詳細資訊，請參閱《Amazon VPC 使用者指南》中的透過介面端點存取服務。

如需使用 CloudFormation 建立和設定端點的詳細資訊，請參閱 AWS CloudFormation 使用者指南中的 AWS::EC2::VPCEndpoint 資源。

若要為 Lambda 建立介面端點 (AWS CLI)

使用 create-vpc-endpoint 命令，並指定 VPC ID、VPC 端點 (介面) 的類型、服務名稱、要使用端點的子網路，以及要與端點網路介面建立關聯的安全群組。例如：

```
aws ec2 create-vpc-endpoint
  --vpc-id vpc-ec43eb89
  --vpc-endpoint-type Interface
  --service-name com.amazonaws.us-east-1.lambda
  --subnet-id subnet-abababab
  --security-group-id sg-1a2b3c4d
```

為 Lambda 建立介面端點政策

若要控制誰可以使用您的介面端點，以及使用者可以存取哪些 Lambda 函數，您可以將端點政策連接至您的端點。此政策會指定下列資訊：

- 可執行動作的主體。
- 委託人可以執行的動作。
- 委託人可以對其執行動作的資源。

如需詳細資訊，請參閱《Amazon VPC 使用者指南》中的使用 VPC 端點控制對服務的存取。

範例：Lambda 動作的介面端點政策

以下是 Lambda 端點政策的範例。當連接到端點時，此政策允許使用者 MyUser 叫用函式 my-function。

> **Note**
>
> 您需要在資源中同時包含合格和不合格的函數 ARN。

```
{
   "Statement":[
      {
         "Principal":
         {
             "AWS": "arn:aws:iam::111122223333:user/MyUser"
         },
         "Effect":"Allow",
         "Action":[
            "lambda:InvokeFunction"
         ],
         "Resource": [
               "arn:aws:lambda:us-east-2:123456789012:function:my-function",
               "arn:aws:lambda:us-east-2:123456789012:function:my-function:*"
            ]
      }
   ]
}
```

設定 Lambda 函數的檔案系統存取權

您可以設定函數，將 Amazon Elastic File System (Amazon EFS) 檔案系統掛載到本機目錄。使用 Amazon EFS，您的函數程式碼可安全地存取和修改共用資源，並發揮高度並行效能。

支援的地區

Amazon EFS for Lambda 已在所有商業區域推出，但以下區域除外：亞太區域 (紐西蘭)、亞太區域 (台北)、亞太區域 (馬來西亞)、墨西哥 (中部)、亞太區域 (泰國)、加拿大西部 (卡加利)。

章節

- 執行角色和使用者許可
- 設定檔案系統和存取點
- 連線至檔案系統 (主控台)

執行角色和使用者許可

如果檔案系統沒有使用者設定的 AWS Identity and Access Management (IAM) 政策，EFS 將使用預設政策，只要是可以使用檔案系統掛載目標連線至檔案系統的用戶端，該政策都會授予完全存取權。如果檔案系統具有使用者設定的 IAM 政策，則函數的執行角色必須具有正確的 `elasticfilesystem` 許可。

執行角色許可

- elasticfilesystem:ClientMount
- elasticfilesystem:ClientWrite (唯讀連線不需要)

這些許可包含在 AmazonElasticFileSystemClientReadWriteAccess 受管政策中。此外，您的執行角色必須具有連線至檔案系統的 VPC 所需的許可。

設定檔案系統時，Lambda 會使用您的許可來驗證掛載目標。若要設定函數以連線至檔案系統，您的使用者需要下列許可：

使用者許可

- elasticfilesystem:DescribeMountTargets

設定檔案系統和存取點

在函數所連線的每個可用區域中，在具有掛載目標的 Amazon EFS 中建立檔案系統。為了提高效能和彈性，請至少使用兩個可用區域。例如，在簡單的組態中，您可以在不同的可用區域中擁有包含兩個私有子網路的 VPC。此函數連線到兩個子網路，每個子網路都有可用的掛載目標。確定函數和掛載目標所使用的安全群組允許 NFS 流量 (連接埠 2049)。

> **Note**
>
> 建立檔案系統時，您選擇稍後無法變更的效能模式。General purpose (一般用途) 模式具有較低的延遲，而 Max I/O (IO 上限) 模式支援較高的輸送量上限和 IOPS。如需協助選擇，請參閱 Amazon Elastic File System 使用者指南中的 Amazon EFS 效能。

存取點會將函數的每個執行個體連線至可用區域連線到的正確掛載目標。為了獲得最佳效能，請使用非根路徑建立存取點，並限制您在每個目錄中建立的檔案數目。下列範例會在檔案系統上建立名為 `my-function` 的目錄，並將擁有者 ID 設為 1001，具有標準目錄許可 (755)。

Example存取點組態

- 名稱 – `files`
- 使用者 ID – `1001`
- 群組 ID – `1001`
- 路徑 – `/my-function`
- 許可 – `755`
- 擁有者使用者 ID – `1001`
- 群組使用者 ID – `1001`

當函數使用存取點時，會提供使用者 ID 1001，並具有目錄的完整存取權。

如需詳細資訊，請參閱 Amazon Elastic File System 使用者指南中的下列主題。

- 為 Amazon EFS 建立資源
- 處理使用者、群組和許可

連線至檔案系統 (主控台)

此函數會透過 VPC 中的本機網路連線至檔案系統。您的函數連線到的子網路可能是包含檔案系統掛載點的子網路，或同一個可用區域中的子網路，其可將 NFS 流量 (連接埠 2049) 路由至檔案系統。

> **Note**
>
> 如果您的函數尚未連線至 VPC，請參閱讓 Lambda 函數存取 Amazon VPC 中的資源。

設定檔案系統存取權

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 組態 ，然後選擇 檔案系統 。
4. 在 檔案系統 中，選擇 新增檔案系統。
5. 設定下列屬性：
   - EFS file system (EFS 檔案系統) - 相同 VPC 中的檔案系統存取點。
   - Local mount path (本機掛載路徑) - 檔案系統掛載於 Lambda 函數 (以 `/mnt/` 開頭) 的位置。

> **定價**
>
> Amazon EFS 會針對儲存和輸送量收取費用，費率依儲存類別而異。如需詳細資訊，請參閱 Amazon EFS 定價。
> Lambda 會針對 VPC 之間的資料傳輸收取費用。這僅適用於您的函數的 VPC 對等連線到帶有檔案系統的另一個 VPC 的情況。費率與相同區域內 VPC 之間的 Amazon EC2 資料傳輸費用相同。如需詳細資訊，請參閱 Lambda 定價。

建立 Lambda 函數的別名

您可以為您的 Lambda 函數建立別名。Lambda 別名是您可以更新的函數版本的指標。函數的使用者可以使用別名 Amazon Resource Name (ARN) 來存取函數版本。部署新版本時，您可以更新別名以使用新版本，或分割兩個版本之間的流量。

Console

若要使用主控台建立別名

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 Aliases (別名)，然後選擇 Create alias (建立別名)。
4. 在 Create alias (建立別名) 頁面，執行下列動作：
   a. 輸入別名的 Name (名稱)。
   b. (選用) 輸入別名的 Description (描述)。
   c. 在 Version (版本)中，選擇要別名指向的函數版本。
   d. (選用) 若要在別名上設定路由，請展開 Weighted alias (加權別名)。如需更多詳細資訊，請參閱 使用加權別名實作 Lambda Canary 部署。
   e. 選擇 Save (儲存)。

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 建立別名，請使用 create-alias 命令。

```
aws lambda create-alias \
  --function-name my-function \
  --name alias-name \
  --function-version version-number \
  --description " "
```

若要變更別名以指向新版本的函數，請使用 update-alias 命令。

```
aws lambda update-alias \
  --function-name my-function \
  --name alias-name \
  --function-version version-number
```

若要刪除別名，請使用 delete-alias 命令。

```
aws lambda delete-alias \
  --function-name my-function \
  --name alias-name
```

上述步驟中的 AWS CLI 命令會對應到下列 Lambda API 操作：

- CreateAlias
- UpdateAlias
- DeleteAlias

在事件來源和許可政策中使用 Lambda 別名

每個別名都有唯一的 ARN。別名只能指向函數版本，而非另一個別名。您可以更新別名以指向新版本的函數。

諸如 Amazon Simple Storage Service (Amazon S3) 等事件來源會叫用您的 Lambda 函數。當事件發生時，這些事件來源會維護一個映射，以識別發生事件時要叫用的函數。如果您在映射組態中指定了 Lambda 函數別名，則不需要在函數版本變更時更新映射。如需更多詳細資訊，請參閱 Lambda 如何處理來自串流和佇列式事件來源的記錄。

在資源政策中，您可以授予事件來源使用 Lambda 函數的許可。如果您在政策中指定了別名 ARN，則不需要在函數版本變更時更新政策。

資源政策

您可以使用以資源為基礎的政策，將服務、資源或帳號存取權授予您的函數。該許可的範圍取決於您是將其套用至別名、版本或整個函數。例如，如果您使用別名名稱 (例如 `helloworld:PROD`)，許可可讓您使用別名 ARN (`helloworld`) 來叫用 `helloworld:PROD` 函數。

如果您嘗試在沒有別名或特定版本的情況下叫用該函數，則會出現許可錯誤。即使您嘗試直接叫用與別名相關聯的函數版本，仍會發生此許可錯誤。

例如，當 Amazon S3 代表 `amzn-s3-demo-bucket` 時，下列 AWS CLI 命令會授予 Amazon S3 叫用 `helloworld` 函數之 PROD 別名的許可。

```
aws lambda add-permission \
  --function-name helloworld \
  --qualifier PROD \
  --statement-id 1 \
  --principal s3.amazonaws.com \
  --action lambda:InvokeFunction \
  --source-arn arn:aws:s3:::amzn-s3-demo-bucket \
  --source-account 123456789012
```

如需在政策中使用資源名稱的詳細資訊，請參閱微調政策的資源和條件區段。

使用加權別名實作 Lambda Canary 部署

可以使用加權別名來分割相同函數的兩個不同版本之間的流量。透過此方法，可以用少量流量測試新版本的函數，並視需要快速復原。這稱為 Canary 部署。Canary 部署與藍/綠部署的不同之處在於只將新版本暴露於一部分請求，而不是一次切換所有流量。

一個別名可以指向最多兩個 Lambda 函數版本。版本必須符合下列條件：

- 兩個版本必須擁有相同的執行角色。
- 這兩個版本都必須具有相同無效字元佇列組態，或是沒有無效字元佇列組態。
- 兩個版本都必須發佈。別名不能指向 $LATEST。

> **Note**
>
> Lambda 使用簡單的概率模型來在兩個函數版本之間分配流量。在流量較低時，您可能會看到每個版本已設定流量百分比與實際流量百分比之間，存在很大差異。如果您的函數使用佈建並行，透過在別名路由作用期間設定較高數目的已佈建並行執行個體，則可以避免溢出調用。

建立加權別名

Console

若要使用主控台在別名上設定路由

> **Note**
>
> 確認函數至少有兩個已發佈的版本。若要建立其他版本，請遵循建立函數版本中的指示。

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 Aliases (別名)，然後選擇 Create alias (建立別名)。
4. 在 Create alias (建立別名) 頁面，執行下列動作：
   a. 輸入別名的 Name (名稱)。
   b. (選用) 輸入別名的 Description (描述)。
   c. 在 Version (版本)中，選擇要別名指向的第一個函數版本。
   d. 展開 Weighted alias (加權別名)。
   e. 在 Additional version (其他版本)中，選擇要別名指向的第二個函數版本。

f. 為 Weight (%) (權重 (%))函數輸入一個權重值。權數是當別名被叫用時，指派至該版本的流量百分比。第一版收到剩餘權數。例如，若指定 10% 至 Additional version (其他版本)，第一版會自動指派 90%。

g. 選擇儲存。

AWS CLI

使用 create-alias 和 update-alias AWS CLI 命令來設定兩個函數版本之間的流量權重。當您建立或更新別名時，請在 `routing-config` 參數中指定流量權重。

以下範例會建立一個名為 routing-alias 的 Lambda 函數別名，此別名指向函數的版本 1。函數的版本 2 會接收 3% 的流量。剩餘 97% 的流量會路由至版本 1。

```
aws lambda create-alias \
  --name routing-alias \
  --function-name my-function \
  --function-version 1  \
  --routing-config AdditionalVersionWeights={"2"=0.03}
```

使用 `update-alias` 命令可增加連入流量至版本 2 的百分比。在下列範例中，您會將流量增加到 5%。

```
aws lambda update-alias \
  --name routing-alias \
  --function-name my-function \
  --routing-config AdditionalVersionWeights={"2"=0.05}
```

若要將所有流量路由傳送至版本 2，請使用 `update-alias` 命令，將 `function-version` 屬性變更為將別名指向版本 2。此命令也會重設路由組態。

```
aws lambda update-alias \
  --name routing-alias \
  --function-name my-function  \
  --function-version 2 \
  --routing-config AdditionalVersionWeights={}
```

上述步驟中的 AWS CLI 命令對應至下列 Lambda API 操作：

- CreateAlias

- UpdateAlias

判斷調用哪個版本

當您設定兩個函數版本之間的流量權重時，有兩種方法可判斷已叫用的 Lambda 函數版本：

- CloudWatch Logs - 對於每個函數調用，Lambda 會自動發出含有已調用版本 ID 的 START 記錄。範例：

  ```
  START RequestId: 1dh194d3759ed-4v8b-a7b4-1e541f60235f Version: 2
  ```

  對於別名叫用，Lambda 使用 ExecutedVersion 維度，依照已叫用的版本來篩選指標資料。如需詳細資訊，請參閱檢視 Lambda 函數的指標。
- 回應承載 (同步呼叫) – 回應至同步函式呼叫包含 x-amz-executed-version 標題，以顯示已呼叫哪個函式版本。

使用加權別名建立滾動部署

使用 AWS CodeDeploy 和 AWS Serverless Application Model (AWS SAM) 建立滾動部署，自動偵測函數程式碼的變更、部署新版本的函數，並逐漸增加流向新版本的流量。流量和增加速率是您可以設定的參數。

在滾動部署中， 會 AWS SAM 執行下列任務：

- 配置您的 Lambda 函數並建立別名。加權別名路由組態是實作滾動部署的基本功能。
- 建立 CodeDeploy 應用程式和部署群組。部署群組會管理滾動部署和轉返 (如有需要)。
- 在您建立 Lambda 函數的新版本時進行偵測。
- 觸發 CodeDeploy 開始新版本的部署。

範本範例 AWS SAM

下列範例顯示 AWS SAM 範本，以進行簡單的滾動部署。

```
AWSTemplateFormatVersion : '2010-09-09'
Transform: AWS::Serverless-2016-10-31
Description: A sample SAM template for deploying Lambda functions

Resources:
```

```
# Details about the myDateTimeFunction Lambda function
  myDateTimeFunction:
    Type: AWS::Serverless::Function
    Properties:
      Handler: myDateTimeFunction.handler
      Runtime: nodejs24.x
# Creates an alias named "live" for the function, and automatically publishes when you
 update the function.
      AutoPublishAlias: live
      DeploymentPreference:
# Specifies the deployment configuration
          Type: Linear10PercentEvery2Minutes
```

此範本會定義名為 myDateTimeFunction 的 Lambda 函數，且具備下列屬性。

AutoPublishAlias

AutoPublishAlias 屬性會建立名為 live 的別名。此外，當您為該函數儲存新代碼時， AWS SAM 架構會自動偵測。之後，架構會發佈一個新的函數版本並更新 live 別名以指向新版本。

DeploymentPreference

DeploymentPreference 屬性會決定 CodeDeploy 應用程式將流量從 Lambda 函數的原始版本轉移到新版本的速率。值 Linear10PercentEvery2Minutes 每隔兩分鐘會將額外 10% 的流量轉移到新版本。

如需預先定義的部署組態清單，請參閱部署組態。

如需如何使用 CodeDeploy 建立滾動部署的詳細資訊 AWS SAM，請參閱以下內容：

- 教學課程：使用 CodeDeploy 和 部署更新的 Lambda 函數 AWS Serverless Application Model
- 使用 逐步部署無伺服器應用程式 AWS SAM

管理 Lambda 函數版本

您可以使用版本來管理 函數的部署。例如，您可以發佈新版本的函式來測試 Beta 版，而不會影響穩定生產版本的使用者。每次發佈函數時，Lambda 都會建立新版本的函數。新版本是此函數未發佈版本的副本。未發布的版本名稱為 $LATEST。

重要的是，每當您部署函式程式碼時，都會覆寫 $LATEST 中的目前程式碼。若要儲存 $LATEST 的目前迭代，需建立新的函式版本。如果 $LATEST 與先前發布的版本相同，則必須先將變更部署到 $LATEST，才能建立新版本。這些變更可能包括更新程式碼或修改函式組態設定。

發布函數版本後，其程式碼、執行時期、架構、記憶體、層以及大多數其他組態設定都是不可變的。這意味著如果未從 $LATEST 發布新版本，就無法變更這些設定。您可以為已發佈的函數版本設定下列項目：

- 觸發
- 目的地
- 佈建並行
- 非同步叫用
- 資料庫連線和代理

> **Note**
>
> 搭配使用執行時期管理控制項與自動模式時，函數版本所使用的執行時期版本會自動更新。使用 Function update (函數更新) 或 Manual (手動) 模式時，不會更新執行階段版本。如需詳細資訊，請參閱the section called “執行時期版本更新”。

章節

- 建立函數版本
- 使用版本
- 授予許可

建立函數版本

您只能在未發佈的函數版本上變更函數代碼和設定。當您發佈版本時，Lambda 會鎖定程式碼和大多數設定以為該版本的使用者保持一致的使用體驗。

您可以使用 Lambda 主控台建立函數版本。

新建函數版本

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇一個函數，然後選擇 Versions 索引標簽。
3. 在版本組態頁面上，選擇 Publish new version (發佈新版本)。
4. (選用) 輸入版本描述。
5. 選擇發布。

或者，您可以使用 PublishVersion API 操作發佈函數的版本。

下列 AWS CLI 命令會發佈新版本的 函數。回應會傳回關於新版本的組態資訊，包含版本號碼，以及含有版本尾碼的函式 ARN。

```
aws lambda publish-version --function-name my-function
```

您應該會看到下列輸出：

```
{
  "FunctionName": "my-function",
  "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function:1",
  "Version": "1",
  "Role": "arn:aws:iam::123456789012:role/lambda-role",
  "Handler": "function.handler",
  "Runtime": "nodejs24.x",
  ...
}
```

> **Note**
>
> Lambda 會為版本控制指派依序遞增的序號。即使刪除並重新建立函數，Lambda 也不會重複使用版本號碼。

使用版本

您可以使用合格的 ARN 或不合格的 ARN 來參考您的 Lambda 函數。

- 合格的 ARN - 帶有版本尾碼的函數 ARN。下列範例指的是 helloworld 函數的版本 42。

```
arn:aws:lambda:aws-region:acct-id:function:helloworld:42
```

- 不合格的 ARN - 沒有版本尾碼的函數 ARN。

```
arn:aws:lambda:aws-region:acct-id:function:helloworld
```

您可以在所有相關 API 操作中使用合格或不合格的 ARN。但是，您無法使用不合格的 ARN 來建立別名。

如果您決定不發佈函數版本，則可以在事件來源映射中使用合格或不合格的 ARN 來調用函數。當您使用不合格的 ARN 調用函數時，Lambda 會隱含調用 $LATEST。

每個 Lambda 函數版本的合格 ARN 是唯一的。發佈版本之後，您就無法變更 ARN 或函數程式碼。

只有當從未發布過程式碼或從上次發布版本後程式碼已變更，Lambda 才會發布新的函數版本。如果沒有變更，函數版本會保持在最新發佈的版本。

當您發布一個版本時，Lambda 會建立函式程式碼與組態的不可變快照。並非所有組態變更都會觸發新版本的發布。下列組態變更符合函式發布版本的條件：

- 函數程式碼
- 環境變數
- 執行時期
- 處理常式
- 層
- 記憶體大小
- Timeout (逾時)
- VPC 組態
- 無效字母佇列 (DLQ) 組態
- IAM 角色
- Description
- 架構 (x86_64 或 arm64)
- 暫時性儲存大小
- 套件類型

- 記錄組態
- 檔案系統組態
- SnapStart
- 追蹤組態

預留並行等操作設定的變更不會觸發新版本的發布。

授予許可

您可以使用以資源為基礎的政策或以身分為基礎的政策來授予您函數的存取權。許可的範圍取決於您是將政策套用至函數或函數的一個版本。如需政策中函數資源名稱的詳細資訊，請參閱微調政策的資源和條件區段。

您可以使用 函數別名，簡化事件來源和 AWS Identity and Access Management (IAM) 政策的管理。如需詳細資訊，請參閱建立 Lambda 函數的別名。

在 Lambda 函數上使用標籤

可以標記函數來組織和管理資源。標籤是與跨 AWS 服務支援的資源相關聯的自由格式索引鍵值對。如需標籤使用案例的詳細資訊，請參閱《標記 AWS 資源和標籤編輯器指南》中的常見標記策略。

標籤適用於函數層級，而不適用於版本或別名。標籤不是在您發佈版本時 AWS Lambda 建立 快照的版本特定組態的一部分。可以使用 Lambda API 來檢視和更新標籤。也可以在 Lambda 主控台中管理特定函數時檢視和更新標籤。

章節

- 使用標籤所需的許可
- 搭配使用標籤與 Lambda 主控台
- 搭配 使用標籤 AWS CLI

使用標籤所需的許可

若要允許 AWS Identity and Access Management (IAM) 身分 (使用者、群組或角色) 讀取或設定資源上的標籤，請為其授予相應許可：

- lambda:ListTags：當資源具有標籤時，將此許可授予給需要對其呼叫 `ListTags` 的任何人員。對於已標記函數，`GetFunction` 也需要此許可。
- lambda:TagResource：將此許可授予給需要呼叫 `TagResource` 或在建立時執行標記的任何人員。

或者，也可以考慮授予 lambda：UntagResource 許可，以允許對資源進行 `UntagResource` 呼叫。

如需詳細資訊，請參閱Lambda 適用的身分型 IAM 政策。

搭配使用標籤與 Lambda 主控台

您可以使用 Lambda 主控台建立具有標籤的函數、將標籤新增至現有函數，以及依您新增的標籤篩選函數。

在建立函數時新增標籤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇建立函數。
3. 選擇 Author from scratch (從頭開始撰寫) 或 Container image (容器映像)。

4. 在基本資訊中，設定您的函數。如需有關設定函數的詳細資訊，請參閱 設定函數。
5. 展開 Advanced settings (進階設定)，然後選取 Enable tags (啟用標籤)。
6. 選擇 Add new tag (新增標籤)，然後輸入 Key (索引鍵) 和選用的 Value (值)。若要新增更多標籤，請重複此步驟。
7. 選擇 Create function (建立函數)。

為現有函數新增標籤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 選擇 Configuration (組態)，然後選擇 Tags (標籤)。
4. 在 Tags (標籤) 下，選擇 Manage tags (管理標籤)。
5. 選擇 Add new tag (新增標籤)，然後輸入 Key (索引鍵) 和選用的 Value (值)。若要新增更多標籤，請重複此步驟。
6. 選擇儲存。

使用標籤篩選函數

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇搜尋方塊以查看函數屬性和標籤索引鍵的清單。
3. 選擇標籤索引鍵以查看目前 AWS 區域中正在使用的值清單。
4. 選取使用：「tag-name」以查看所有標記有此索引鍵的函數，或選擇運算子以進一步依值篩選。
5. 選取標籤值，依標籤索引鍵和值的組合進行篩選。

搜尋列也支援搜尋標籤鍵。輸入 `tag` 以僅查看標籤索引鍵清單，或輸入索引鍵名稱以在清單中尋找。

搭配 使用標籤 AWS CLI

可以使用 Lambda API 在現有的 Lambda 資源 (包括函數) 上新增和移除標籤。也可以在建立函數時新增標籤，這可讓您在整個生命週期中標記資源。

使用 Lambda 標籤 API 來更新標籤

可以透過 TagResource 和 UntagResource API 操作來新增和移除受支援 Lambda 資源的標籤。

可使用 AWS CLI來呼叫這些操作。若要將標籤新增至現有資源，請使用 `tag-resource` 命令。此範例會新增兩個標籤，一個包含索引鍵 *Department*，另一個包含索引鍵 *CostCenter*。

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

若要移除標籤，請使用 `untag-resource` 命令。此範例會移除具有索引鍵 *Department* 的標籤。

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
--tag-keys Department
```

在建立函數時新增標籤

若要建立帶有標籤的新 Lambda 函數，請使用 CreateFunction API 操作。指定 `Tags` 參數。可以使用 `create-function` CLI 命令和 --tags 選項來呼叫此操作。將標籤參數與 `CreateFunction` 搭配使用之前，請確保您的角色除了此操作所需的一般許可外，還擁有標記資源的許可。如需有關標記許可的詳細資訊，請參閱the section called “使用標籤所需的許可”。此範例會新增兩個標籤，一個包含索引鍵 *Department*，另一個包含索引鍵 *CostCenter*。

```
aws lambda create-function --function-name my-function
--handler index.js --runtime nodejs24.x \
--role arn:aws:iam::123456789012:role/lambda-role \
--tags Department=Marketing,CostCenter=1234ABCD
```

檢視函數上的標籤

若要檢視套用至特定 Lambda 資源的標籤，請使用 `ListTags` API 操作。如需詳細資訊，請參閱 ListTags。

您可以使用 `list-tags` AWS CLI 命令呼叫此操作，方法是提供 ARN (Amazon Resource Name)。

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier
```

可以使用 GetFunction API 操作檢視套用至特定資源的標籤。可比較的功能不可用於其他資源類型。

可以使用 `get-function` CLI 命令呼叫此操作：

```
aws lambda get-function --function-name my-function
```

依標籤篩選資源

您可以使用 AWS Resource Groups Tagging API GetResources API 操作，依標籤篩選資源。GetResources 操作可接收最多 10 個篩選條件，每個篩選條件皆包含標籤索引鍵與最多 10 個標籤值。為 GetResources 提供一個 ResourceType，即可依特定資源類型進行篩選。

您可以使用 get-resources AWS CLI 命令呼叫此操作。如需使用 get-resources 的範例，請參閱《AWS CLI 命令參考》中的 get-resources。

Lambda 函數的回應串流

Lambda 函數可以透過 Lambda 函數 URLs 或使用 InvokeWithResponseStream API （透過 AWS SDK 或直接 API 呼叫），將回應承載原生串流回用戶端。您的 Lambda 函數也可以透過 Amazon API Gateway 代理整合串流回應承載，該整合使用 InvokeWithResponseStream API 來叫用您的函數。透過提高第一個位元組時間 (TTFB) 效能，回應串流有益於延遲敏感應用程式。這是因為您可以在部分回應可用時將其傳回給用戶端。此外，相較於緩衝回應的 6 MB 上限，回應串流函式最多可傳回 200 MB 的承載。串流回應也意味著您的函數不需要將整個回應放在記憶體裡。若是非常大的回應，這有助於減少您需要為函數設定的記憶體容量。

> **Note**
>
> Lambda 回應串流尚未全面提供 AWS 區域。請參閱依AWS 區域劃分的 Builder Center 功能，以取得依區域劃分的功能可用性。

Lambda 串流回應的速度取決於回應的大小。函式回應的串流速度在前 6 MB 不受限制。若回應大於 6 MB，則其餘的回應會受到頻寬上限的限制。如需串流頻寬的詳細資訊，請參閱回應串流的頻寬限制。

串流回應會產生成本，而且當叫用用戶端連線中斷時，不會中斷或停止串流回應。客戶需支付完整函數持續時間的費用，因此客戶在設定長函數逾時時應小心。

Lambda 支援 Node.js 受管執行期的回應串流。對於其他語言，包括 Python，您可以使用自訂執行期搭配自訂執行期 API 整合來串流回應或使用 Lambda Web Adapter。

> **Note**
>
> 透過 Lambda 主控台測試函數時，您一律會看到緩衝的回應。

主題

- 回應串流的頻寬限制
- VPC 與回應串流的相容性
- 撰寫啟用回應串流的 Lambda 函數
- 使用 Lambda 函數 URL 調用啟用回應串流的函數
- 教學課程：建立具有函數 URL 的回應串流 Lambda 函數

回應串流的頻寬限制

函式回應承載的前 6 MB 不受頻寬限制。在最初的高頻寬流量後，Lambda 會以最高 2 MBps 的速率串流您的回應。如果函式回應一直都沒超過 6 MB，則永遠不會適用此頻寬限制。

> **Note**
>
> 頻寬限制僅適用於函數的回應有效負載，不適用於函數的網路存取。

無頻寬上限的速率取決於諸多因素 (包括函數的處理速度)。一般來說，您可以預期函式回應前 6 MB 的速率會高於 2 Mbps。如果您的函數正在將回應串流到 以外的目的地 AWS，則串流速率也取決於外部網際網路連線的速度。

VPC 與回應串流的相容性

在 VPC 環境中使用 Lambda 函式時，回應串流需注意下列重要考量事項：

- Lambda 函式 URL 不支援 VPC 環境中的回應串流。
- 您可以使用 `InvokeWithResponseStream` API 透過 AWS SDK 叫用 Lambda 函數，在 VPC 內使用回應串流。這需要為 Lambda 設定適當的 VPC 端點。
- 對於 VPC 環境，您需要為 Lambda 建立介面 VPC 端點，用以啟用 VPC 環境中資源與 Lambda 服務之間的通訊。

VPC 環境中回應串流的典型架構可能包括：

```
Client in VPC -> Interface VPC endpoint for Lambda -> Lambda function -> Response
 streaming back through the same path
```

撰寫啟用回應串流的 Lambda 函數

撰寫回應串流函數的處理常式不同於典型的處理常式模式。撰寫串流函數時，請務必執行下列動作：

- 使用 `awslambda.streamifyResponse()` 裝飾項目包裝函式。`awslambda` 全域物件由 Lambda 的 Node.js 執行時期環境提供。
- 從容地結束串流，以確保所有資料處理都完成。

設定處理常式函數以串流回應

為了向執行期指示 Lambda 應該串流函數的回應，您必須使用 `streamifyResponse()` 裝飾項目包裝函數。這會通知執行期使用適當的邏輯路徑來串流回應，並讓函數串流回應。

`streamifyResponse()` 裝飾項目接受的函數可接受以下參數：

- `event` – 提供函數 URL 調用事件的相關資訊，例如 HTTP 方法、查詢參數和請求內文。
- `responseStream` – 提供可寫入的串流。
- `context` – 提供方法和屬性，以及有關調用、函數以及執行環境的資訊。

`responseStream` 物件是 Node.js `writableStream`。與任何此類串流一樣，您應該使用 `pipeline()` 方法。

> **Note**
>
> `awslambda` 全域物件由 Lambda 的 Node.js 執行時期自動提供，不需要匯入。

Example啟用回應串流的處理常式

```
import { pipeline } from 'node:stream/promises';
import { Readable } from 'node:stream';

export const echo = awslambda.streamifyResponse(async (event, responseStream, _context)
 => {
  // As an example, convert event to a readable stream.
  const requestStream = Readable.from(Buffer.from(JSON.stringify(event)));

  await pipeline(requestStream, responseStream);
});
```

雖然 `responseStream` 提供寫入串流的 `write()` 方法，但仍建議您盡可能使用 `pipeline()`。使用 `pipeline()` 可確保可寫入的串流不會被更快的可讀串流所淹沒。

結束串流

請確保在處理常式傳回之前正確結束串流。`pipeline()` 方法會自動處理這種情形。

對於其他使用案例，請呼叫 `responseStream.end()` 方法以正確結束串流。此方法發出訊號，表示不應向串流寫入更多資料。如果您使用 `pipeline()` 或 `pipe()` 寫入串流，則不需要此方法。

從 Node.js 24 開始，Lambda 不會在處理常式傳回或回應串流結束之後，再等待未解決的承諾完成。如果您的函數依賴其他非同步操作，例如計時器或擷取，您應該在處理常式中`await`執行這些操作。

Example使用 pipeline() 結束串流的範例

```
import { pipeline } from 'node:stream/promises';

export const handler = awslambda.streamifyResponse(async (event, responseStream,
 _context) => {
  await pipeline(requestStream, responseStream);
});
```

Example未使用 pipeline() 結束串流的範例

```
export const handler = awslambda.streamifyResponse(async (event, responseStream,
 _context) => {
  responseStream.write("Hello ");
  responseStream.write("world ");
  responseStream.write("from ");
  responseStream.write("Lambda!");
  responseStream.end();
});
```

使用 Lambda 函數 URL 調用啟用回應串流的函數

> **Note**
>
> 您的 Lambda 函數現在可以透過 Amazon API Gateway 代理整合串流回應承載。

可以透過變更函數 URL 的調用模式來調用已啟用回應串流的函數。調用模式決定了 Lambda 用來調用函數的 API 操作。可用的調用模式如下：

- `BUFFERED` – 此為預設選項。Lambda 會使用 `Invoke` API 操作調用您的函數。承載完成時，即可使用調用結果。承載大小上限為 6 MB。
- `RESPONSE_STREAM` – 啟用您的函數，當承載結果變得可用時串流它們。Lambda 會使用 `InvokeWithResponseStream` API 操作調用您的函數。回應承載大小上限為 200 MB。

您仍然可以透過直接呼叫 Invoke API 操作來調用函數而無需回應串流。不過，Lambda 會串流透過函數 URL 調用的所有回應承載，直到您將調用模式變更為 BUFFERED。

Console

設定函數 URL 的調用模式 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您要為其設定調用模式的函數名稱。
3. 選擇 Configuration (組態) 標籤，然後選擇 Function URL (函數 URL)。
4. 選擇編輯，然後選擇其他設定。
5. 在調用模式下，選擇所需的調用模式。
6. 選擇儲存。

AWS CLI

若要設定函數 URL 的調用模式 (AWS CLI)

```
aws lambda update-function-url-config \
  --function-name my-function \
  --invoke-mode RESPONSE_STREAM
```

CloudFormation

若要設定函數 URL 的調用模式 (CloudFormation)

```
MyFunctionUrl:
  Type: AWS::Lambda::Url
  Properties:
    AuthType: AWS_IAM
    InvokeMode: RESPONSE_STREAM
```

如需設定函數 URL 的詳細資訊，請參閱 Lambda 函數 URL。

教學課程：建立具有函數 URL 的回應串流 Lambda 函數

在本教學課程中，您會建立格式為 .zip 封存檔的 Lambda 函數，其包含的函數 URL 端點會傳回回應串流。如需設定函數 URL 的詳細資訊，請參閱 函數 URL。

先決條件

此教學課程假設您具備基本的 Lambda 操作知識並了解 Lambda 主控台。若您尚未了解，請遵循 使用主控台建立一個 Lambda 函數 中的指示，建立您的第一個 Lambda 函數。

若要完成下列步驟，您需要 AWS CLI 版本 2。命令和預期的輸出會列在不同的區塊中：

```
aws --version
```

您應該會看到下列輸出：

```
aws-cli/2.13.27 Python/3.11.6 Linux/4.14.328-248.540.amzn2.x86_64 exe/x86_64.amzn.2
```

對於長命令，逸出字元 (\) 用於將命令分割為多行。

在 Linux 和 macOS 上，使用您偏好的 shell 和套件軟體管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。本指南中的 CLI 命令範例使用 Linux 格式。如果您使用的是 Windows CLI，必須重新格式化包含內嵌 JSON 文件的命令。

建立執行角色

建立執行角色，授予您的 Lambda 函數存取 AWS 資源的許可。

若要建立執行角色

1. 開啟 AWS Identity and Access Management (IAM) 主控台的 角色 頁面。
2. 選擇建立角色。
3. 建立具備下列屬性的角色：
   - 信任的實體類型：AWS 服務
   - 使用案例：Lambda
   - 許可 – AWSLambdaBasicExecutionRole

- 角色名稱 - **response-streaming-role**。

AWSLambdaBasicExecutionRole 政策具有函數將日誌寫入 Amazon CloudWatch Logs 所需的許可。建立角色後，請記下其 Amazon Resource Name (ARN)。下一個步驟將需要此值。

建立回應串流函數 (AWS CLI)

使用 AWS Command Line Interface (AWS CLI) 建立具有函數 URL 端點的回應串流 Lambda 函數。

建立可串流回應的函數

1. 將下列程式碼範例複製至名為 `index.js` 的檔案中。此函式會以 1 秒的間隔串流三個回應。

```
exports.handler = awslambda.streamifyResponse(
 async (event, responseStream, _context) => {
  // Metadata is a JSON serializable JS object. Its shape is not defined here.
  const metadata = {
  statusCode: 200,
  headers: {
   "Content-Type": "application/json",
   "CustomHeader": "outerspace"
  }
  };

  // Assign to the responseStream parameter to prevent accidental reuse of the non-
wrapped stream.
  responseStream = awslambda.HttpResponseStream.from(responseStream, metadata);

  responseStream.write("Streaming with Helper \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 0 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 1 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.write("Hello 2 \n");
  await new Promise(r => setTimeout(r, 1000));
  responseStream.end();
  await responseStream.finished();
 }
  );
```

2. 建立部署套件。

```
zip function.zip index.js
```

3. 使用 `create-function` 命令建立一個 Lambda 函數。使用上一個步驟的角色 ARN 取代 `--role` 的值。此命令會將函式逾時設定為 10 秒，允許函式串流三個回應。

```
aws lambda create-function \
  --function-name my-streaming-function \
  --runtime nodejs24.x \
  --zip-file fileb://function.zip \
  --handler index.handler \
  --timeout 10 \
  --role arn:aws:iam::123456789012:role/response-streaming-role
```

建立函數 URL

1. 為函式新增可授予 `lambda:InvokeFunctionUrl` 及 `lambda:InvokeFunction` 許可的資源型政策。每個陳述式都必須在個別命令中進行新增。將 的值取代`--principal`為您的 AWS 帳戶 ID。

```
aws lambda add-permission \
  --function-name my-streaming-function \
  --action lambda:InvokeFunctionUrl \
  --statement-id UrlPolicyInvokeURL \
  --principal 123456789012 \
  --function-url-auth-type AWS_IAM
```

```
aws lambda add-permission \
    --function-name my-streaming-function \
    --action lambda:InvokeFunction \
    --statement-id UrlPolicyInvokeFunction \
    --principal 123456789012
```

2. 使用 `create-function-url-config` 命令為函數建立 URL 端點。

```
aws lambda create-function-url-config \
  --function-name my-streaming-function \
  --auth-type AWS_IAM \
  --invoke-mode RESPONSE_STREAM
```

> **Note**
>
> 如果收到有關 `--invoke-mode` 的錯誤，可能需要升級至較新版本的 AWS CLI。

測試函數 URL 端點

透過調用函數來測試整合。可以在瀏覽器中開啟函數 URL，也可以使用 curl。

```
curl --request GET "https://abcdefghijklm7nop7qrs740abcd.lambda-url.us-east-1.on.aws/"
 --user "AKIAIOSFODNN7EXAMPLE" --aws-sigv4 "aws:amz:us-east-1:lambda" --no-buffer
```

我們的函數 URL 使用 IAM_AUTH 驗證類型。這表示您必須同時使用 AWS 存取金鑰與私密金鑰來簽署請求。在上一個命令中，將 取代AKIAIOSFODNN7EXAMPLE為 AWS 存取金鑰 ID。出現提示時輸入您的 AWS 私密金鑰。如果您沒有 AWS 私密金鑰，您可以改為使用臨時 AWS 登入資料。

您應該看到如下回應：

```
Streaming with Helper
Hello 0
Hello 1
Hello 2
```

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。透過刪除您不再使用 AWS 的資源，您可以避免不必要的 費用 AWS 帳戶。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇刪除。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。

2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **`confirm`**，然後選擇 刪除 。

使用 Lambda 中繼資料端點

Lambda 中繼資料端點可讓您的函數探索他們在哪個可用區域 (AZ) 中執行，讓您能夠透過路由至 Amazon ElastiCache 和 Amazon RDS 端點等相同可用區域資源來最佳化延遲，並實作可用區域感知彈性模式。

端點會透過執行環境中的 localhost HTTP API 以簡單的 JSON 格式傳回中繼資料，並且可供執行時間和延伸存取。

章節

- 開始使用
- 了解可用區域 IDs
- API 參考

開始使用

Powertools for AWS Lambda 提供公用程式，以存取 Python、TypeScript、Java 和 .NET 中的 Lambda 中繼資料端點。公用程式會在第一次呼叫後快取回應，並自動處理 SnapStart 快取失效。

使用 Powertools for AWS Lambda 中繼資料公用程式或直接呼叫中繼資料端點

Python

安裝 Powertools 套件：

```
pip install "aws-lambda-powertools"
```

在處理常式中使用中繼資料公用程式：

Example使用 Powertools 擷取 AZ ID (Python)

```
from aws_lambda_powertools.utilities.lambda_metadata import get_lambda_metadata

def handler(event, context):
    metadata = get_lambda_metadata()
    az_id = metadata.availability_zone_id  # e.g., "use1-az1"

    return {"az_id": az_id}
```

TypeScript

安裝 Powertools 套件：

```
npm install @aws-lambda-powertools/commons
```

在處理常式中使用中繼資料公用程式：

Example使用 Powertools 擷取 AZ ID (TypeScript)

```
import { getMetadata } from '@aws-lambda-powertools/commons/utils/metadata';

const metadata = await getMetadata();

export const handler = async () => {
  const { AvailabilityZoneID: azId } = metadata;
  return azId;
};
```

Java

將 Powertools 相依性新增至您的 pom.xml：

```
<dependencies>
    <dependency>
        <groupId>software.amazon.lambda</groupId>
        <artifactId>powertools-lambda-metadata</artifactId>
        <version>2.10.0</version>
    </dependency>
</dependencies>
```

在處理常式中使用中繼資料用戶端：

Example使用 Powertools 擷取 AZ ID (Java)

```
import software.amazon.lambda.powertools.metadata.LambdaMetadata;
import software.amazon.lambda.powertools.metadata.LambdaMetadataClient;

public class App implements RequestHandler<Object, String> {

    @Override
    public String handleRequest(Object input, Context context) {
```

```
        LambdaMetadata metadata = LambdaMetadataClient.get();
        String azId = metadata.getAvailabilityZoneId(); // e.g., "use1-az1"

        return "{\"azId\": \"" + azId + "\"}";
    }
}
```

.NET

安裝 Powertools 套件：

```
dotnet add package AWS.Lambda.Powertools.Metadata
```

在處理常式中使用中繼資料類別：

Example使用 Powertools 擷取 AZ ID (.NET)

```
using AWS.Lambda.Powertools.Metadata;

public class Function
{
    public string Handler(object input, ILambdaContext context)
    {
        var azId = LambdaMetadata.AvailabilityZoneId;
        return $"Running in AZ: {azId}";
    }
}
```

All Runtimes

所有 Lambda 執行期都支援中繼資料端點，包括自訂執行期和容器映像。使用以下範例，使用 Lambda 在執行環境中自動設定的環境變數，直接從您的函數存取中繼資料 API。

Example直接存取中繼資料端點

```
# Variables are automatically set by Lambda
METADATA_ENDPOINT="http://${AWS_LAMBDA_METADATA_API}/2026-01-15/metadata/execution-
environment"

# Make the request
RESPONSE=$(curl -s -H "Authorization: Bearer ${AWS_LAMBDA_METADATA_TOKEN}"
 "$METADATA_ENDPOINT")
```

```
# Parse the AZ ID
AZ_ID=$(echo "$RESPONSE" | jq -r '.AvailabilityZoneID')

echo "Function is running in AZ ID: $AZ_ID"
```

了解可用區域 IDs

AZ IDs（例如，use1-az1) 一律在所有 AWS 帳戶中參考相同的實體位置，而 AZ 名稱（例如，us-east-1a) 可能會映射到特定區域中每個 AWS 帳戶中的不同實體基礎設施。如需詳細資訊，請參閱 AZ IDs以取得跨帳戶一致性。

將 AZ ID 轉換為 AZ 名稱：

若要將 AZ ID 轉換為 AZ 名稱，請使用 Amazon EC2 DescribeAvailabilityZones API。若要使用此 API，請將 ec2:DescribeAvailabilityZones許可新增至函數的執行角色。

API 參考

環境變數

Lambda 會在每個執行環境中自動設定下列環境變數：

- AWS_LAMBDA_METADATA_API – 格式的中繼資料伺服器地址 {ipv4_address}:{port}（例如 169.254.100.1:9001)。
- AWS_LAMBDA_METADATA_TOKEN – 目前執行環境的唯一身分驗證字符。Lambda 會在初始化時自動產生此權杖。將其包含在所有中繼資料 API 請求中。

Endpoint

GET http://${AWS_LAMBDA_METADATA_API}/2026-01-15/metadata/execution-environment

請求

必要標頭：

- Authorization – 來自具有 Bearer 結構描述之AWS_LAMBDA_METADATA_TOKEN環境變數的字符值：Bearer <token>。此字符型身分驗證可針對伺服器端請求偽造 (SSRF) 漏洞提供深度防禦。每個執行環境都會在初始化時收到唯一的隨機產生字符。

回應

狀態： `200 OK`

Content-Type： `application/json`

快取控制： `private, max-age=43200, immutable`

回應在執行環境中是不可變的。用戶端應快取回應並遵循 `Cache-Control` TTL。對於 SnapStart 函數，TTL 會在初始化期間減少，因此當執行環境可能位於不同的可用區域中時，用戶端會在還原後重新整理中繼資料。如果您使用 Powertools，會自動處理快取和 SnapStart 無效。

本文：

```
{
  "AvailabilityZoneID": "use1-az1"
}
```

`AvailabilityZoneID` 欄位包含執行環境執行所在可用區域的唯一識別符。

> **Note**
>
> 其他欄位可能會新增至未來更新的回應。用戶端應該忽略未知欄位，如果出現新欄位，則不會失敗。

錯誤回應

- 401 未授權 – `Authorization`標頭遺失或包含無效的字符。驗證您正在傳遞 `Bearer ${AWS_LAMBDA_METADATA_TOKEN}`。
- 405 不允許的方法 – 請求方法不是 `GET`。
- 500 內部伺服器錯誤 – 伺服器端處理錯誤。

了解 Lambda 函數調用方法

部署 Lambda 函數之後，可以透過多種方式調用它：

- Lambda 主控台：使用 Lambda 主控台快速建立測試事件以調用函數。
- SDK AWS – 使用 AWS SDK 以程式設計方式叫用您的 函數。
- 調用 API：使用 Lambda 調用 API 以直接調用您的 函數。
- AWS Command Line Interface (AWS CLI) – 使用 `aws lambda invoke` AWS CLI 命令直接從命令列叫用函數。
- 函數 URL HTTP(S) 端點：使用函數 URL 建立專用 HTTP(S) 端點，可用來調用函數。

所有這些方法都是調用函數的直接方式。在 Lambda 中，常見的使用案例是根據應用程式中其他位置發生的事件調用函數。有些服務可以調用 Lambda 函數來處理每個新事件。這稱為觸發條件。對於串流和佇列型服務，Lambda 會調用函數來處理批次記錄。這稱為事件來源映射。

當您調用函式時，您可以選擇以同步或非同步方式進行調用。使用同步調用，您會等待函式處理事件並傳回回應。使用非同步調用，Lambda 會將事件排入佇列以進行處理，並立即傳回回應。調用 API 中的 `InvocationType` 請求參數決定 Lambda 如何調用函數。值 `RequestResponse` 表示同步調用，值 `Event` 表示非同步調用。

若要透過 IPv6 調用您的函數，請使用 Lambda 的公有雙堆疊端點。雙堆疊端點可同時支援 IPv4 和 IPv6。Lambda 雙堆疊端點使用下列語法：

```
protocol://lambda.us-east-1.api.aws
```

也可以使用 Lambda 函數 URL 透過 IPv6 來調用函數。函數 URL 端點的格式如下：

```
https://url-id.lambda-url.us-east-1.on.aws
```

如果函數調用導致錯誤，則對於同步調用，請檢視回應中的錯誤訊息，並手動重試調用。對於非同步調用，Lambda 會自動處理重試，並可將調用記錄傳送至目的地。

同步調用 Lambda 函數

當您以同步方式調用函數時，Lambda 會執行函數並等候回應。當函數執行完成時，Lambda 會從函數的程式碼傳回回應，其中包含調用的函數版本等額外資料。若要透過 AWS CLI 以同步方式調用函式，請使用 `invoke` 命令。

```
aws lambda invoke --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "key": "value" }' response.json
```

如果您使用 AWS CLI 第 2 版，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

下圖顯示以同步方式調用 Lambda 函數的用戶端。Lambda 會將事件直接傳送到函數，並將函數的回應傳送回調用者。

`payload` 是包含 JSON 格式事件的字串。AWS CLI 從該函數寫入回覆的檔案名稱是 `response.json`。如果函數傳回物件或錯誤，則回應內文是 JSON 格式的物件或錯誤。如果函數結束時沒有錯誤，則回應內文為 `null`。

> **Note**
>
> 在傳送回應之前，Lambda 不會等待外部擴充功能完成。外部延伸項目會在執行環境中做為獨立的處理序執行，並在函數調用完成之後繼續執行。如需更多詳細資訊，請參閱 使用 Lambda 擴充功能來增強 Lambda 函數。

命令的輸出會顯示於終端機，包括來自 Lambda 的回應中標頭中的資訊。這包括處理事件的版本 (當您使用別名時很實用)，以及 Lambda 傳回的狀態碼。如果 Lambda 能夠執行函數，則狀態碼為 200，即使函數傳回了錯誤。

> **Note**
>
> 對於逾時很久的函式，您的用戶端可能在等待回應的同時，在同步調用期間中斷連線。設定您的 HTTP 用戶端、SDK、防火牆、Proxy 或作業系統，以透過逾時或持續作用設定允許長時間連線。

如果 Lambda 無法執行函數，錯誤則會顯示在輸出中。

```
aws lambda invoke --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload value response.json
```

您應該會看到下列輸出：

```
An error occurred (InvalidRequestContentException) when calling the Invoke operation:
 Could not parse request body into json: Unrecognized token 'value': was expecting
 ('true', 'false' or 'null')
 at [Source: (byte[])"value"; line: 1, column: 11]
```

AWS CLI 是開放原始碼工具，可讓您在命令列 shell 中使用命令來與 AWS 服務互動。若要完成本節中的步驟，您必須擁有 AWS CLI 版本 2。

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用 AWS CLI 第 2 版，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

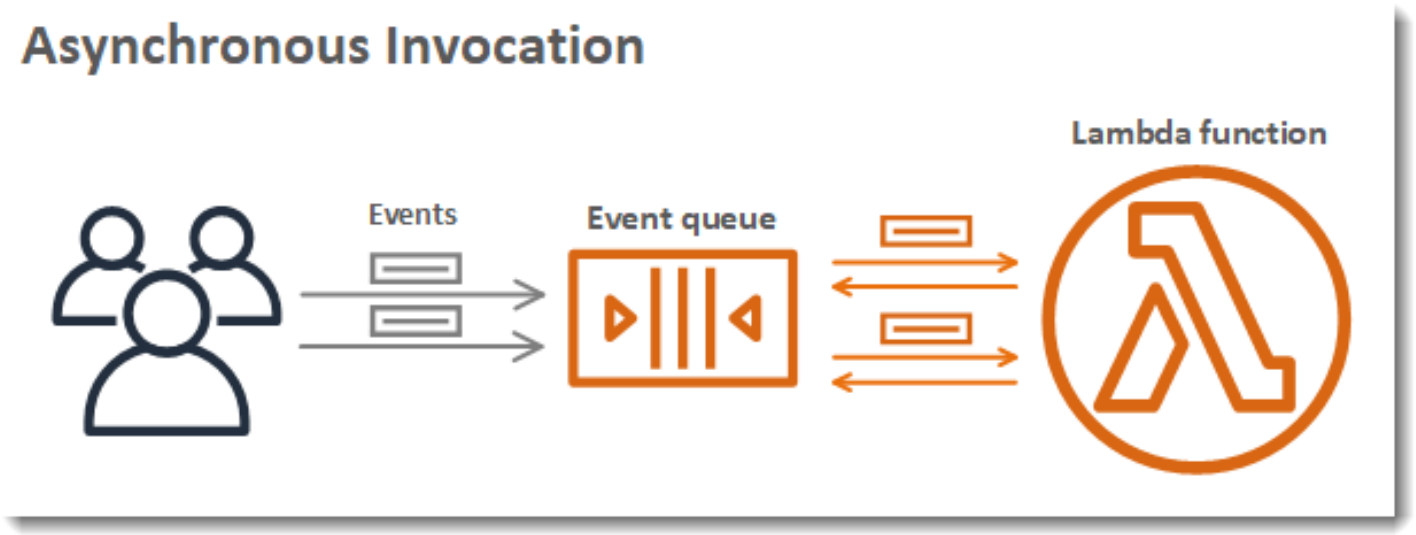
如需有關 Invoke API 的詳細資訊 (包括參數、標頭和錯誤的完整清單)，請參閱調用。

當您直接調用函式時，您可以檢查回應中是否有錯誤並重試。AWS CLI 和 AWS SDK 也會在用戶端逾時、調節和服務錯誤時自動重試。如需更多詳細資訊，請參閱 了解 Lambda 中的重試行為。

非同步調用 Lambda 函數

數個 AWS 服務 (例如 Amazon Simple Storage Service (Amazon S3) 和 Amazon Simple Notification Service (Amazon SNS)) 會以非同步方式調用函數以處理事件。您也可以使用 AWS Command Line Interface (AWS CLI) 或其中一個 AWS SDK 以非同步方式調用 Lambda 函數。當您以非同步方式呼叫函數時，您不需要等待來自函數程式碼的回應。您可以將事件傳遞給 Lambda，而 Lambda 會處理其餘的工作。您可以設定 Lambda 處理錯誤的方式，且可將調用記錄傳送至 Amazon Simple Queue Service (Amazon SQS) 或 Amazon EventBridge (EventBridge) 這類下游資源，以便將您應用程式的元件鏈結在一起。

下圖顯示以非同步方式來調用 Lambda 函數的用戶端。Lambda 會先將事件排入佇列，再將事件傳送到函數。



針對非同步調用，Lambda 會將事件置放在佇列中，並傳回成功回應，其中不包含其他資訊。單獨的程序會從佇列讀取事件，並將事件傳送到您的函數。

若要使用 AWS Command Line Interface (AWS CLI) 或其中一個 AWS SDK 以非同步方式調用 Lambda 函數，須將 InvocationType 參數設定為 `Event`。以下是使用 AWS CLI 命令調用函數的範例。

```
aws lambda invoke \
  --function-name my-function  \
  --invocation-type Event \
  --cli-binary-format raw-in-base64-out \
  --payload '{ "key": "value" }' response.json
```

您應該會看到下列輸出：

```
{
    "StatusCode": 202
}
```

如果您使用 AWS CLI 第 2 版，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

輸出檔 (`response.json`) 不包含任何資訊，但仍會在您執行此命令時建立。如果 Lambda 無法將事件新增到佇列，錯誤訊息就會顯示在命令輸出中。

Lambda 如何處理非同步調用的錯誤和重試

Lambda 會管理函數的非同步事件佇列，並在發生錯誤時嘗試重試。依預設，如果函數傳回錯誤，則 Lambda 會嘗試多執行函數兩次，且兩次嘗試之間等候一分鐘，而第二次和第三次嘗試之間等候兩分鐘。函式錯誤包含函式程式碼所傳回的錯誤，以及函式執行時間所傳回的錯誤，例如逾時。

如果函式沒有足夠的並行可用來處理所有事件，則額外請求會遭到調節。依預設，對於調節錯誤 (429) 和系統錯誤 (500 序列)，Lambda 會將事件傳回到佇列，並嘗試再次執行函數長達 6 小時。重試間隔從第一次嘗試後的 1 秒呈指數增加到最多 5 分鐘。如果佇列包含許多項目，Lambda 會增加重試間隔，並降低從佇列讀取事件的速率。

即使您的函數並未傳回錯誤，它也可以從 Lambda 收到相同的事件很多次，因為佇列本身最終一致。如果函式來不及處理傳入事件，也可以從佇列中刪除事件，而不需傳送到函式。確保您的函式程式碼可從容地處理重複的事件，而且您有足夠的並行可用來處理所有調用。

當佇列很長時，新的事件可能會在 Lambda 有機會將其傳送到函數前就已過期。當事件過期時或所有處理嘗試皆失敗時，Lambda 便會捨棄該事件。您可以設定函數的錯誤處理，以減少 Lambda 執行的重試次數，或更快地捨棄未處理的事件。若要擷取已捨棄的事件，請為函式設定無效字母佇列。若要擷取調用失敗的記錄 (例如逾時或執行時期錯誤)，請建立失敗時目的地。

設定 Lambda 非同步調用的錯誤處理

使用以下設定，設定 Lambda 在非同步調用函數時如何處理錯誤和重試：

- MaximumEventAgeInSeconds：Lambda 在捨棄事件之前，將事件保留在非同步事件佇列中的時間上限，以秒為單位。
- MaximumRetryAttempts：當函數傳回錯誤時，Lambda 重試的次數上限。

使用 Lambda 主控台或 AWS CLI 設定函數、版本或別名的錯誤處理設定。

Console

設定錯誤處理

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態，然後選擇非同步調用。
4. 在 非同步調用 下方，選擇 編輯。
5. 進行下列設定。
   - Maximum age of event (事件存留期上限) - Lambda 在非同步事件佇列中保留事件的時間上限，最多 6 小時。
   - Retry attempts (重試嘗試) - 當函數傳回錯誤時，Lambda 重試的次數上限 (介於 0 到 2)。
6. 選擇儲存。

AWS CLI

若要使用 AWS CLI 設定非同步調用，請使用 put-function-event-invoke-config 命令。下列範例會設定事件存留期為 1 小時且不重試的函數。

```
aws lambda put-function-event-invoke-config \
  --function-name error \
  --maximum-event-age-in-seconds 3600 \
  --maximum-retry-attempts 0
```

此 `put-function-event-invoke-config` 命令會覆寫任何在函數、版本或別名上的現有組態。若要設定某個選項而不重設其他選項，請使用 update-function-event-invoke-config。下列範例會將 Lambda 設定為在無法處理事件時，將記錄傳送至名為 `destination` 的標準 SQS 佇列。

```
aws lambda update-function-event-invoke-config \
  --function-name my-function \
  --destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-1:123456789012:destination"}}'
```

您應該會看到下列輸出：

```
{
    "LastModified": 1573686021.479,
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
    "MaximumRetryAttempts": 0,
    "MaximumEventAgeInSeconds": 3600,
    "DestinationConfig": {
        "OnSuccess": {},
        "OnFailure": {}
    }
}
```

當調用事件超過最大存留期，或所有重試嘗試都失敗時，Lambda 會將其捨棄。若要保留已捨棄事件的副本，請設定失敗事件的目的地。

擷取 Lambda 非同步調用的記錄

Lambda 可以將非同步調用的記錄傳送到下列其中一項 AWS 服務。

- Amazon SQS：標準 SQS 佇列
- Amazon SNS：標準 SNS 主題
- Amazon S3：Amazon S3 儲存貯體 (僅限失敗時)
- AWS Lambda：Lambda 函數
- Amazon EventBridge：EventBridge 事件匯流排

呼叫記錄包含請求和回應 (JSON 格式) 的詳細資訊。您可以為成功處理的事件及所有處理嘗試皆失敗的事件設定個別目標。或者，您可以將標準 Amazon SQS 佇列或標準 Amazon SNS 主題設為捨棄事件的無效字母佇列。針對無效字母佇列，Lambda 只會傳送事件的內容，而不包含回應的詳細資訊。

如果 Lambda 無法將記錄傳送到您設定的目的地，便會將 `DestinationDeliveryFailures` 指標傳送至 Amazon CloudWatch。如果您的組態中包含不受支援的目的地類型 (例如 Amazon SQS FIFO 佇列或 Amazon SNS FIFO 主題)，就可能發生這種情形。傳遞錯誤也可能因許可錯誤和大小限制而發生。如需 Lambda 調用指標的詳細資訊，請參閱：the section called “呼叫指標”

> **Note**
>
> 為了防止函數觸發，您可以將函數的預留並行設為零。當您將非同步調用函數的預留並行設定為零時，Lambda 會開始將新事件傳送至設定的無效字母佇列或失敗時的事件目的地，不會進

行任何重試。若要處理在預留並行設定為零時傳送的事件，您必須使用來自無效字母佇列或失敗時的事件目的地之事件。

新增目的地

若要保留非同步調用的記錄，請將目的地新增至您的函數。您可以選擇將成功或失敗的調用傳送至目的地。每個函數都可以有多個目的地，因此您可以為成功和失敗的事件配置單獨的目的地。傳送至目的地的每筆記錄都是包含調用之詳細資訊的 JSON 文件。與錯誤處理設定相似，您可以設定函數、函數版本或是別名的目標。

> **Tip**
>
> 您也可以保留下列事件來源映射類型的呼叫失敗記錄：Amazon Kinesis、Amazon DynamoDB 和 Apache Kafka (Amazon MSK 和自我管理 Apache Kafka)。

以下資料表列出針對非同步調用記錄支援的目的地。若要讓 Lambda 成功將記錄傳送至您選擇的目的地，請確定函數的執行角色也包含相關許可。此資料表也說明每個目的地類型如何接收 JSON 調用記錄。

目的地類型	所需的許可	目的地特定 JSON 格式
Amazon SQS 佇列	sqs:SendMessage	Lambda 會將調用記錄做為 `Message` 傳遞至目的地。
Amazon SNS 主題	sns:Publish	Lambda 會將調用記錄做為 `Message` 傳遞至目的地。
Amazon S3 儲存貯體 (僅限失敗時)	s3:PutObject s3:ListBucket	• Lambda 會以 JSON 物件的形式在目的地儲存貯體中儲存調用記錄。 • S3 物件使用以下命名慣例： `aws/lambda/async/<function-name>/YYYY/MM/DD/YYYY-MM-DD`

目的地類型	所需的許可	目的地特定 JSON 格式
		`THH.MM.SS-<Random UUID>`
Lambda 函式	lambda:InvokeFunction	Lambda 傳遞調用記錄會以承載形式傳遞給函數。
EventBridge	events:PutEvents	• Lambda 會在 PutEvents 呼叫中以 `detail` 形式來傳遞調用記錄。 • `source` 事件欄位的值是 `lambda`。 • `detail-type` 事件欄位的值是 Lambda Function Invocation Result - Success (Lambda 函數調用結果 - 成功) 或 Lambda Function Invocation Result - Failure (Lambda 函數調用結果 - 失敗)。 • `resource` 事件欄位包含函數和目的地 Amazon Resource Names (ARN)。 • 如需其他事件欄位，請參閱 Amazon EventBridge 事件。

> **Note**
>
> 對於 Amazon S3 目的地，如果您已使用 KMS 金鑰在相應儲存貯體上啟用加密，則函數也需要 kms:GenerateDataKey 許可。

> **Important**
>
> 若以 Amazon SNS 為目的地，注意 Amazon SNS 的訊息大小上限為 256 KB。若非同步調用承載接近 1 MB，則調用記錄 (包含原始承載與額外中繼資料) 可能會超過 Amazon SNS 的上限，導致傳送失敗。建議將 Amazon SQS 或 Amazon S3 用作較大承載的目的地。

以下步驟說明如何使用 Lambda 主控台和 AWS CLI為函式設定目的地。

Console

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在函數概觀下，選擇新增目的地。
4. 針對來源，選擇非同步調用。
5. 如為條件，請從下列選項中選擇：
   - On failure (失敗時) - 當事件的所有處理嘗試都失敗，或超過存留期上限時，傳送記錄。
   - 成功時 - 當函數成功處理非同步調用時傳送記錄。
6. 對於目的地類型，請選擇接收調用記錄的資源類型。
7. 對於目的地，請選擇一個資源。
8. 選擇儲存。

AWS CLI

若要使用 設定目的地 AWS CLI，請執行 update-function-event-invoke-config 命令。下列範例會將 Lambda 設定為在無法處理事件時，將記錄傳送至名為 `destination` 的標準 SQS 佇列。

```
aws lambda update-function-event-invoke-config \
  --function-name my-function \
  --destination-config '{"OnFailure":{"Destination": "arn:aws:sqs:us-
east-1:123456789012:destination"}}'
```

Amazon S3 目的地的安全最佳實務

刪除已設定為目的地的 S3 儲存貯體而不從函數的組態中移除目的地，可能會產生安全風險。如果其他使用者知道目的地儲存貯體的名稱，他們可以在其 AWS 帳戶中重新建立儲存貯體。失敗調用的記錄會被傳送到其儲存貯體，可能公開來自您函數的資料。

> **Warning**
>
> 為了確保無法將函數的調用記錄傳送到另一個 中的 S3 儲存貯體 AWS 帳戶，請將條件新增至函數的執行角色，以限制您帳戶中儲存貯體的`s3:PutObject`許可。

下列範例顯示的 IAM 政策，將函數的 `s3:PutObject` 許可限制為帳戶中的儲存貯體。此政策也為 Lambda 提供了使用 S3 儲存貯體做為目的地所需的 `s3:ListBucket` 許可。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
}
```

若要使用 AWS 管理主控台 或 將許可政策新增至函數的執行角色 AWS CLI，請參閱下列程序中的指示：

Console

若要將許可政策新增至函數的執行角色 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選取您要修改其執行角色的 Lambda 函數。
3. 在組態索引標籤中，選擇許可。
4. 在執行角色索引標籤中，選取函數的角色名稱，以開啟角色的 IAM 主控台頁面。
5. 透過下列步驟將許可政策新增至角色：

   a. 在許可政策窗格中，選擇新增許可 ，然後選取建立內嵌政策。
   b. 在政策編輯器中，選取 JSON。
   c. 將您要新增的政策貼入編輯器 (取代現有的 JSON)，然後選擇下一步。
   d. 在政策詳細資訊下，輸入政策名稱。
   e. 選擇建立政策。

AWS CLI

若要將許可政策新增至函數的執行角色 (CLI)

1. 建立具有所需許可的 JSON 政策文件，並將其儲存在本機目錄中。
2. 使用 IAM `put-role-policy` CLI 命令，將許可新增至函數的執行角色。從您儲存 JSON 政策文件的目錄執行下列命令，並將角色名稱、政策名稱和政策文件取代為您自己的值。

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

調用記錄範例

當調用與條件相符時，Lambda 會將包含調用之詳細資訊的 JSON 文件傳送到目的地。下列範例顯示因函式錯誤而導致處理失敗三次事件的呼叫記錄。

Example

```
{
```

```
    "version": "1.0",
    "timestamp": "2019-11-14T18:16:05.568Z",
    "requestContext": {
        "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
        "condition": "RetriesExhausted",
        "approximateInvokeCount": 3
    },
    "requestPayload": {
        "ORDER_IDS": [
            "9e07af03-ce31-4ff3-xmpl-36dce652cb4f",
            "637de236-e7b2-464e-xmpl-baf57f86bb53",
            "a81ddca6-2c35-45c7-xmpl-c3a03a31ed15"
        ]
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "responsePayload": {
        "errorMessage": "RequestId: e4b46cbf-b738-xmpl-8880-a18cdf61200e Process exited
 before completing request"
    }
}
```

呼叫記錄包含事件、回應以及記錄傳送原因的詳細資訊。

追蹤傳至目的地的請求

您可以使用 AWS X-Ray 查看每個請求排入佇列、由 Lambda 函數處理並傳遞至目的地服務的連接檢視。為調用函數的函數或服務啟用 X-Ray 追蹤時，Lambda 會將 X-Ray 標頭新增至請求，並將標頭傳送至目的地服務。來自上游服務的追蹤會自動連結至來自下游 Lambda 函數和目的地服務的追蹤，進而建立整個應用程式的端對端檢視。如需追蹤的詳細資訊，請參閱：使用 視覺化 Lambda 函數叫用 AWS X-Ray。

新增無效字母佇列

做為失敗目標的替代項目，您可以設定您的函數，使其具備一個無效字母佇列來儲存捨棄的事件，以供後續處理。無效字母佇列的運作方式與失敗目標相同，會在事件的所有處理嘗試失敗，或是在沒有處理的情況下過期時使用。不過，您只能在函數層級新增或移除無效字母佇列。函數版本使用與未發布版本

($LATEST) 相同的無效字母佇列設定。失敗目標也支援其他目標，並會在呼叫記錄中包含函數回應的詳細資訊。

若要重新處理無效字母佇列中的事件，請將它設定為 Lambda 函數的事件來源。或者，您可以手動擷取事件。

您可以為無效字母佇列選擇 Amazon SQS 標準佇列或 Amazon SNS 標準主題。不支援 SQS FIFO 佇列和 Amazon SNS FIFO 主題。

- Amazon SQS 佇列 - 佇列會保留失敗的事件，直到其遭到擷取為止。如果您希望單一實體 (例如 Lambda 函數或 CloudWatch 警示) 處理失敗的事件，請選擇 Amazon SQS 標準佇列。如需詳細資訊，請參閱搭配 Amazon SQS 使用 Lambda。
- Amazon SNS 主題 - 主題會將失敗的事件轉送到一個或多個目的地。如果您希望多個實體對失敗的事件採取動作，請選擇 Amazon SNS 標準主題。例如，您可以設定一個主題，以將事件傳送到電子郵件地址、Lambda 函數及/或 HTTP 端點。如需詳細資訊，請參閱使用 Amazon SNS 通知調用 Lambda 函數。

若要將事件傳送到佇列或主題，您的函式需要額外許可。將具有所需許可的政策新增到您函數的執行角色。如果目標佇列或主題使用客戶受管 AWS KMS 金鑰加密，請確定函數的執行角色和金鑰的資源型政策都包含相關許可。

在建立目標及更新函式的執行角色之後，將無效字母佇列新增到您的函式。您可以設定多個函式來將事件傳送到相同的目標。

Console

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態，然後選擇非同步調用。
4. 在非同步調用下方，選擇編輯。
5. 將無效字母佇列服務設定為 Amazon SQS 或 Amazon SNS。
6. 選擇目標佇列或主題。
7. 選擇儲存。

AWS CLI

若要使用 設定無效字母佇列 AWS CLI，請使用 update-function-configuration 命令。

```
aws lambda update-function-configuration \
  --function-name my-function \
  --dead-letter-config TargetArn=arn:aws:sns:us-east-1:123456789012:my-topic
```

Lambda 會依現狀將事件傳送到無效字母佇列，其屬性中有額外資訊。您可以使用此資訊來識別該函式所傳回的錯誤，或建立事件與日誌或 AWS X-Ray 追蹤的關聯性。

無效字母佇列訊息屬性

- RequestID (字串) - 調用請求的 ID。請求 ID 會出現於函式日誌中。您也可以使用 X-Ray SDK 在追蹤的屬性中記錄請求 ID。然後，您可以在 X-Ray 主控台中依請求 ID 搜尋追蹤。
- ErrorCode (數字) - HTTP 狀態碼。
- ErrorMessage (字串) - 第 1 KB 的錯誤訊息。

如果 Lambda 無法將訊息傳送到無效字母佇列，則會刪除事件並發出 DeadLetterErrors 指標。這可能由於缺乏許可，或訊息總大小超過目標佇列或主題的限制，而發生此狀況。例如，假設一則內文大小接近 1 MB 的 Amazon SNS 通知觸發了某個函式，導致函式執行出錯。在這種情況下，Amazon SNS 新增的事件資料與 Lambda 新增的屬性結合後，可能導致訊息超過無效字母佇列中允許的大小上限。

如果您使用 Amazon SQS 作為事件來源，請對 Amazon SQS 佇列本身 (而非 Lambda 函數) 設定無效字母佇列。如需詳細資訊，請參閱搭配 Amazon SQS 使用 Lambda。

叫用耐用的 Lambda 函數

持久性 Lambda 函數可以使用與預設 Lambda 函數相同的方法叫用，但對於長時間執行的執行有重要的考量。本節涵蓋叫用模式、執行管理和耐久函數的最佳實務。

同步調用限制

耐用 Lambda 函數的同步調用限制為 15 分鐘，與預設 Lambda 函數相同。如果您的耐用函數需要執行超過 15 分鐘，則必須以非同步方式叫用它。

何時使用同步調用：用於在 15 分鐘內完成的耐久函數，以及當您需要立即結果時，例如快速核准工作流程或短資料處理任務。

長時間執行工作流程的非同步調用

對於執行時間可能超過 15 分鐘的耐用函數，請使用非同步調用。這可讓函數在用戶端收到立即確認時繼續執行。

TypeScript

```
import { LambdaClient, InvokeCommand } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

// Asynchronous invocation
const command = new InvokeCommand({
  FunctionName: "my-durable-function",
  InvocationType: "Event", // Asynchronous
  Payload: JSON.stringify({ orderId: "12345" })
});

await client.send(command);
```

Python

```
import boto3
import json
```

```
client = boto3.client('lambda')

# Asynchronous invocation
response = client.invoke(
    FunctionName='my-durable-function',
    InvocationType='Event',  # Asynchronous
    Payload=json.dumps({'order_id': '12345'})
)
```

執行管理 APIs

Lambda APIs 來管理和監控持久的函數執行，包括列出執行、取得執行狀態和停止執行。

TypeScript

```
// Get execution status
const statusCommand = new InvokeCommand({
  FunctionName: "my-durable-function",
  InvocationType: "RequestResponse",
  Payload: JSON.stringify({
    action: "getStatus",
    executionId: "exec-123"
  })
});

const result = await client.send(statusCommand);
```

Python

```
# Get execution status
response = client.invoke(
    FunctionName='my-durable-function',
    InvocationType='RequestResponse',
    Payload=json.dumps({
        'action': 'get_status',
        'execution_id': 'exec-123'
    })
)
```

Lambda 如何處理來自串流和佇列式事件來源的記錄

事件來源映射是 Lambda 資源，可從串流和佇列式服務讀取項目，並使用成批的記錄來調用函數。在事件來源映射中，名為事件輪詢器的資源會主動輪詢新訊息並調用函數。依預設，Lambda 會自動擴展事件輪詢器，但對於某些事件來源類型，可以使用佈建模式來控制專用於事件來源映射的事件輪詢器數量下限和上限。

下列服務使用事件來源映射調用 Lambda 函數：

- Amazon DocumentDB (with MongoDB compatibility) (Amazon DocumentDB)
- Amazon DynamoDB
- Amazon Kinesis
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- 自我管理的 Apache Kafka
- Amazon Simple Queue Service (Amazon SQS)

> **Warning**
>
> Lambda 事件來源映射至少會處理每個事件一次，而且可能會重複處理記錄。為避免與重複事件相關的潛在問題，強烈建議您讓函數程式碼具有等冪性。若要進一步了解，請參閱 AWS 知識中心中的如何使 Lambda 函數成為等冪。

事件來源映射與直接觸發條件有何不同

有些 AWS 服務 可以使用觸發直接叫用 Lambda 函數。這些服務會將事件推送至 Lambda，並在發生指定事件時立即調用函數。觸發條件適用於離散事件和即時處理。當您使用 Lambda 主控台建立觸發時，主控台會與對應的 AWS 服務互動，以在該服務上設定事件通知。觸發條件實際上由產生事件的服務 (而非 Lambda) 儲存和管理。以下是一些使用觸發條件調用 Lambda 函數的服務範例：

- Amazon Simple Storage Service (Amazon S3)：在儲存貯體中建立、刪除或修改物件時調用函數。如需詳細資訊，請參閱教學課程：使用 Amazon S3 觸發條件調用 Lambda 函數。
- Amazon Simple Notification Service (Amazon SNS)：當有訊息發布至 SNS 主題時調用函數。如需詳細資訊，請參閱教學課程： AWS Lambda 搭配 Amazon Simple Notification Service 使用。

- Amazon API Gateway：在對特定端點提出 API 請求時調用函數。如需詳細資訊，請參閱使用 Amazon API Gateway 端點調用 Lambda 函數。

事件來源映射是在 Lambda 服務內建立和管理的 Lambda 資源。事件來源映射旨在處理來自佇列的大量串流資料或訊息。批次處理來自串流或佇列的記錄比個別處理記錄更有效率。

批次處理行為

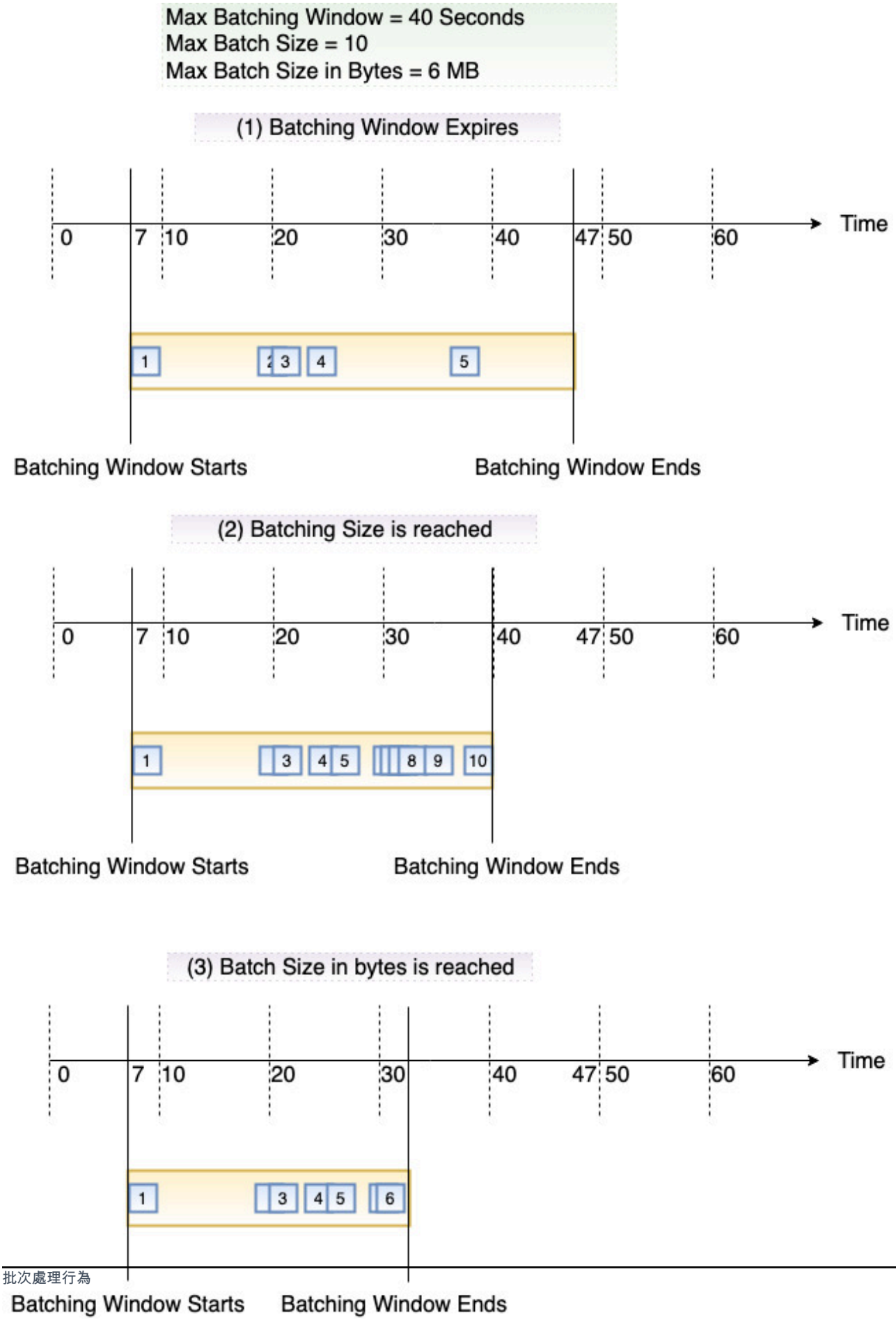
根據預設，事件來源映射會將記錄批次處理到 Lambda 傳送給函數的單個承載中。若要微調批次處理行為，可以設定一個批次間隔 (MaximumBatchingWindowInSeconds) 和批次大小 (BatchSize)。批次間隔是將記錄收集到單一承載的最長時間。批次大小是單一批次中記錄的最大數目。當下列三個條件之一成立時，Lambda 會調用您的函數：

- 批次間隔達到其最大值。預設批次間隔行為會有所不同，這取決於特定的事件來源。
  - 對於 Kinesis、DynamoDB 和 Amazon SQS 事件來源：預設批次間隔為 0 秒。這表示 Lambda 會在記錄可用時立即調用函數。若要設定批次處理間隔，請設定 `MaximumBatchingWindowInSeconds`。可以將此參數設定為從 0 秒到 300 秒之間的任意值，增量為 1 秒。如果設定批次處理間隔，則在上一個函數調用完成時開始下一個批次處理間隔。
  - 如果事件來源是 Amazon MSK、自主管理 Apache Kafka 、Amazon MQ 以及 Amazon DocumentDB：預設批次處理時段為 500 毫秒。您可以將 `MaximumBatchingWindowInSeconds` 設定為從 0 秒到 300 秒之間的任意值，增量為秒。在 Kafka 事件來源映射的佈建模式下，如果設定了批次處理時段，則下一個時段會在上一個批次完成後立即開始。在 Kafka 事件來源映射的非佈建模式下，如果設定了批次處理時段，則下一個時段會在上一個函式調用完成後立即開始。若要最大限度降低在佈建模式下使用 Kafka 事件來源映射時的延遲，請將 `MaximumBatchingWindowInSeconds` 設定為 0。此設定可確保 Lambda 在完成目前的函式調用後立即開始處理下一個批次。如需有關低延遲處理的更多資訊，請參閱低延遲 Apache Kafka。
  - 對於 Amazon MQ 和 Amazon DocumentDB 事件來源：預設批次處理時段為 500 毫秒。您可以將 `MaximumBatchingWindowInSeconds` 設定為從 0 秒到 300 秒之間的任意值，增量為秒。一旦第一條記錄到達，批次間隔就會開始。

> **Note**
>
> 因為您只能以秒為增量變更 `MaximumBatchingWindowInSeconds`，所以在變更之後無法恢復到 500 毫秒的預設批次處理間隔。要恢復預設批次間隔，必須建立新的事件來源映射。

- 已滿足批次大小。批次大小下限為 1。預設和最大批次大小取決於事件來源。如需這些值的詳細資訊，請參閱 `CreateEventSourceMapping` API 操作的 BatchSize 規格。
- 承載大小達到 6 MB。您無法修改此限制。

下圖說明了這三種情況。假設批次間隔從第 `t = 7` 秒開始。在第一種情況下，批次間隔在累積 5 條記錄後在第 `t = 47` 秒達到其最大值 40 秒。在第二種情況下，批次大小在批次間隔到期之前達到 10，因此批次間隔提前結束。在第三種情況下，承載大小在批次間隔到期之前達到最大值，因此批次間隔提前結束。


Max Batching Window = 40 Seconds
Max Batch Size = 10
Max Batch Size in Bytes = 6 MB
(1) Batching Window Expires
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
Batching Window Starts
Batching Window Ends
(2) Batching Size is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
8
9
10
Batching Window Starts
Batching Window Ends
(3) Batch Size in bytes is reached
0
7
10
20
30
40
47
50
60
Time
1
3
4
5
6
Batching Window Starts
Batching Window Ends

建議使用不同的批次與記錄大小測試，讓每個事件來源的輪詢頻率調整為函數可以多快完成作業。CreateEventSourceMapping `BatchSize` 參數會控制每次調用時可傳送至函數的記錄數目上限。更大的批次大小通常會更有效吸收更大集合的記錄的呼叫成本，增加您的傳輸量。

Lambda 在傳送下一批進行處理前不會等待任何已設定的擴充完成。換句話說，當 Lambda 處理下一批記錄時，您的擴充功能可能會繼續執行。如果您違反任何帳戶的 並行 設定或限制，便可能會產生限流的問題。若要偵測此是否為潛在問題，請監控您的函數，並確認您看到的 並行指標 是否高於事件來源映射的預期值。由於兩次調用之間的時間很短，Lambda 可能會短暫報告比碎片數目更高的並行用量。即使對於沒有延伸項目的 Lambda 函數也可能如此。

根據預設，如果您的函數傳回錯誤，則事件來源映射會重新處理整個批次，直到函數成功，或批次中的項目過期為止。若要確保依序處理，事件來源映射會暫停處理受影響的碎片，直到錯誤解決為止。針對串流來源 (DynamoDB 和 Kinesis)，您可以設定函數傳回錯誤時 Lambda 重試的次數上限。導致批次未抵達函數的服務錯誤或限流不會計入重試嘗試次數。您也可以設定事件來源映射，在捨棄事件批次時將調用記錄傳送至目的地。

佈建模式

Lambda 事件來源映射使用事件輪詢器來輪詢您的事件來源，以取得新訊息。根據預設，Lambda 會根據訊息磁碟區管理這些輪詢器的自動擴展。當訊息流量增加時，Lambda 會自動增加事件輪詢器的數量以處理負載，並在流量減少時減少輪詢器。

在佈建模式中，您可以透過定義專用輪詢資源的最小和最大限制來微調事件來源映射的輸送量，這些資源仍然準備好處理預期的流量模式。這些資源的自動擴展速度快 3 倍，可處理事件流量突然峰值，並提供 16 倍的容量來處理數百萬個事件。這可協助您建置具有嚴格效能要求的高度回應事件驅動型工作負載。

在 Lambda 中，事件輪詢器是一種運算單位，具有因事件來源類型而異的輸送量功能。對於 Amazon MSK 和自我管理的 Apache Kafka，每個事件輪詢器最多可以處理 5 MB/秒的輸送量或最多 5 個並行調用。例如，如果您的事件來源產生平均承載 1 MB，且函數的平均持續時間為 1 秒，則單一 Kafka 事件輪詢器可以支援 5 MB/秒輸送量和 5 個並行 Lambda 調用 （假設沒有承載轉換）。對於 Amazon SQS，每個事件輪詢器最多可以處理每秒 1 MB 的輸送量或最多 10 個並行調用。使用佈建模式會根據您的事件輪詢器用量產生額外費用。如需定價詳細資訊，請參閱 AWS Lambda 定價。

佈建模式適用於 Amazon MSK、自我管理 Apache Kafka 和 Amazon SQS 事件來源。雖然並行設定可讓您控制函數的擴展，但佈建的模式可讓您控制事件來源映射的輸送量。若要確保最大效能，您可能需要獨立調整這兩個設定。

佈建模式非常適合需要一致事件處理延遲的即時應用程式，例如處理市場資料饋送的金融服務公司、提供即時個人化建議的電子商務平台，以及管理即時玩家互動的遊戲公司。

每個事件輪詢器都支援不同的輸送量容量：

- 對於 Amazon MSK 和自我管理的 Apache Kafka：高達 5 MB/秒的輸送量或高達 5 個並行調用
- 對於 Amazon SQS：高達 1 MB/秒的輸送量或高達 10 個並行調用或每秒高達 10 個 SQS 輪詢 API 調用。

對於 Amazon SQS 事件來源映射，您可以將輪詢器數目下限設定為 2 到 200，預設值為 2，上限為 2 到 2，000，預設值為 200。Lambda 會在設定的最小值和最大值之間擴展事件輪詢器的數量，每分鐘快速增加最多 1，000 個並行，以提供事件的一致、低延遲處理。

對於 Kafka 事件來源映射，您可以將輪詢器的最小數量設定為 1 到 200，預設值為 1，最大值為 1 到 2，000，預設值為 200。Lambda 會根據主題中的事件待處理項目，在設定的最小值和最大值之間擴展事件輪詢器的數量，以提供事件的低延遲處理。

請注意，對於 Amazon SQS 事件來源，最大並行設定不能與佈建模式搭配使用。使用佈建模式時，您可以透過最大事件輪詢器設定來控制並行。

如需設定佈建模式的詳細資訊，請參閱下列章節：

- 設定 Amazon MSK 事件來源映射的佈建模式
- 為自我管理的 Apache Kafka 事件來源映射設定佈建模式
- 搭配 Amazon SQS 事件來源映射使用佈建模式

若要將佈建模式中的延遲降至最低，請將 `MaximumBatchingWindowInSeconds`設為 0。此設定可確保 Lambda 在完成目前的函式調用後立即開始處理下一個批次。如需有關低延遲處理的更多資訊，請參閱低延遲 Apache Kafka。

設定佈建模式後，可以透過監控 `ProvisionedPollers` 指標來觀察工作負載的事件輪詢器使用情況。如需詳細資訊，請參閱the section called “事件來源映射指標”。

事件來源映射 API

若要透過 AWS Command Line Interface (AWS CLI) 或 AWS SDK 來管理事件來源，可以使用下列 API 操作：

- CreateEventSourceMapping
- ListEventSourceMappings

- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

在事件來源映射上使用標籤

可以標記事件來源映射來組織和管理資源。標籤是與跨 AWS 服務支援的資源相關聯的自由格式索引鍵值對。如需標籤使用案例的詳細資訊，請參閱《標記 AWS 資源和標籤編輯器指南》中的常見標記策略。

事件來源映射與函數相關聯，而函數可以有自己的標籤。事件來源映射不會自動從函數繼承標籤。您可以使用 AWS Lambda API 來檢視和更新標籤。您也可以在 Lambda 主控台中管理特定事件來源映射時檢視和更新標籤，包括使用 Amazon SQS 佈建模式的對應。

使用標籤所需的許可

若要允許 AWS Identity and Access Management (IAM) 身分 (使用者、群組或角色) 讀取或設定資源上的標籤，請為其授予相應許可：

- lambda:ListTags：當資源具有標籤時，將此許可授予給需要對其呼叫 `ListTags` 的任何人員。對於已標記函數，`GetFunction` 也需要此許可。
- lambda:TagResource：將此許可授予給需要呼叫 `TagResource` 或在建立時執行標記的任何人員。

或者，也可以考慮授予 lambda：UntagResource 許可，以允許對資源進行 `UntagResource` 呼叫。

如需詳細資訊，請參閱Lambda 適用的身分型 IAM 政策。

搭配使用標籤與 Lambda 主控台

您可以使用 Lambda 主控台建立具有標籤的事件來源映射、將標籤新增至現有事件來源映射，以及依標籤篩選事件來源映射，包括在 Amazon SQS 的佈建模式中設定的事件來源映射。

當使用 Lambda 主控台為支援的串流和佇列型服務新增觸發條件時，Lambda 會自動建立事件來源映射。如需這些事件來源的詳細資訊，請參閱the section called “事件來源映射”。若要在主控台中建立事件來源映射，需要滿足下列先決條件：

- 一個函數。
- 來自受影響服務的事件來源。

可以新增標籤，做為用於建立或更新觸發條件的相同使用者介面的一部分。

若要在建立事件來源映射時新增標籤

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇 函數的名稱。
3. 在函數概觀下，選擇新增觸發條件。
4. 在觸發條件組態下面的下拉式清單中，選擇事件來源所來自的服務名稱。
5. 為事件來源提供核心組態。如需設定事件來源的詳細資訊，請查閱整合其他服務中相關服務的章節。
6. 在事件來源映射組態下，選擇其他設定。
7. 在標籤下，選擇新增標籤
8. 在索引鍵欄位中，輸入標籤索引鍵。如需有關標記限制的資訊，請參閱《 AWS 標記資源和標籤編輯器指南》中的標記命名限制和要求。
9. 選擇新增。

若要將標籤新增至現有的事件來源映射

1. 在 Lambda 主控台中開啟事件來源映射。
2. 從資源清單中，選擇與您的函數和事件來源 ARN 對應的事件來源映射的 UUID。
3. 從一般組態窗格下的索引標籤清單中，選擇標籤。
4. 選擇管理標籤。
5. 選擇 Add new tag (新增標籤)。
6. 在索引鍵欄位中，輸入標籤索引鍵。如需有關標記限制的資訊，請參閱《 AWS 標記資源和標籤編輯器指南》中的標記命名限制和要求。
7. 選擇儲存。

若要依標籤篩選事件來源映射

1. 在 Lambda 主控台中開啟事件來源映射。
2. 選取搜尋方塊。
3. 從下拉式清單中的標籤子標題下方，選取您的標籤索引鍵。
4. 選取使用：「tag-name」以查看所有標記有此索引鍵的事件來源映射，或選擇運算子以進一步依值篩選。

5. 選取標籤值，依標籤索引鍵和值的組合進行篩選。

搜尋方塊也支援搜尋標籤索引鍵。輸入索引鍵的名稱，以便在清單中尋找該索引鍵。

搭配 使用標籤 AWS CLI

可以使用 Lambda API 在現有的 Lambda 資源 (包括事件來源映射) 上新增和移除標籤。也可以在建立事件來源映射時新增標籤，這可讓您在整個生命週期中標記資源。

使用 Lambda 標籤 API 來更新標籤

可以透過 TagResource 和 UntagResource API 操作來新增和移除受支援 Lambda 資源的標籤。

可使用 AWS CLI來呼叫這些操作。若要將標籤新增至現有資源，請使用 `tag-resource` 命令。此範例會新增兩個標籤，一個包含索引鍵 *`Department`*，另一個包含索引鍵 *`CostCenter`*。

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

若要移除標籤，請使用 `untag-resource` 命令。此範例會移除具有索引鍵 *`Department`* 的標籤。

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
--tag-keys Department
```

在建立事件來源映射時新增標籤

若要建立具有標籤的新 Lambda 事件來源映射，請使用 CreateEventSourceMapping API 操作。指定 `Tags` 參數。您可以使用 `create-event-source-mapping` AWS CLI 命令和 `--tags`選項呼叫此操作。如需關於 CLI 命令的詳細資訊，請參閱《AWS CLI 命令參考》中的 create-event-source-mapping。

將 `Tags` 參數與 `CreateEventSourceMapping` 搭配使用之前，請確保您的角色除了此操作所需的一般許可外，還擁有標記資源的許可。如需有關標記許可的詳細資訊，請參閱the section called “使用標籤所需的許可”。

使用 Lambda 標籤 API 來檢視標籤

若要檢視套用至特定 Lambda 資源的標籤，請使用 `ListTags` API 操作。如需詳細資訊，請參閱 ListTags。

您可以使用 `list-tags` AWS CLI 命令呼叫此操作，方法是提供 ARN (Amazon Resource Name)。

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier
```

依標籤篩選資源

您可以使用 AWS Resource Groups Tagging API GetResources API 操作，依標籤篩選資源。`GetResources` 操作可接收最多 10 個篩選條件，每個篩選條件皆包含標籤索引鍵與最多 10 個標籤值。為 `GetResources` 提供一個 `ResourceType`，即可依特定資源類型進行篩選。

您可以使用 `get-resources` AWS CLI 命令呼叫此操作。如需使用 `get-resources` 的範例，請參閱《AWS CLI 命令參考》中的 get-resources。

控制 Lambda 將哪些事件傳送至您的函數

您可以使用事件篩選來控制 Lambda 將哪些記錄從串流或佇列中傳送至函數。例如，您可以新增篩選條件，讓函數僅處理包含特定資料參數的 Amazon SQS 訊息。事件篩選僅對特定事件來源映射有效。您可以將篩選條件新增至下列事件來源映射 AWS 服務：

- Amazon DynamoDB
- Amazon Kinesis Data Streams
- Amazon MQ
- Amazon Managed Streaming for Apache Kafka (Amazon MSK)
- 自我管理的 Apache Kafka
- Amazon Simple Queue Service (Amazon SQS)

如需使用特定事件來源進行篩選的特定資訊，請參閱the section called “使用具有不同 的篩選條件 AWS 服務”。Lambda 不支援 Amazon DocumentDB 的事件篩選。

依預設，最多可以為單一事件來源映射定義五種不同的篩選條件。篩選條件透過 OR 運算邏輯關聯。如果事件來源的記錄滿足一個或多個篩選條件，則 Lambda 會在它傳送至函數的下一個事件中包含該記錄。如果全部篩選條件都不滿足，則 Lambda 會捨棄該記錄。

> **Note**
>
> 如果需要為一個事件來源定義五個以上的篩選條件，則可以為每個事件來源請求增加最多 10 個篩選條件的配額。如果您嘗試新增超過目前配額許可的篩選條件，Lambda 將在您嘗試建立事件來源時傳回錯誤。

主題

- 了解事件篩選基本知識
- 處理不符合篩選條件標準的記錄
- 篩選條件規則語法
- 將篩選條件標準連接至事件來源映射 (主控台)
- 將篩選條件標準連接至事件來源映射 (AWS CLI)
- 將篩選條件標準連接至事件來源映射 (AWS SAM)
- 篩選條件加密

- 使用具有不同 的篩選條件 AWS 服務

了解事件篩選基本知識

篩選條件標準 (FilterCriteria) 物件是由篩選條件清單 (Filters) 組成的結構。每個篩選條件均是定義事件篩選模式 (Pattern) 的結構。模式是 JSON 篩選條件規則的字串表示法。FilterCriteria 物件的結構如下所示。

```
{
   "Filters": [
        {
            "Pattern": "{ \"Metadata1\": [ rule1 ], \"data\": { \"Data1\":
 [ rule2 ] }}"
        }
    ]
}
```

補充說明，此處是篩選條件的 Pattern 在純文字 JSON 中擴展的值。

```
{
    "Metadata1": [ rule1 ],
    "data": {
        "Data1": [ rule2 ]
    }
}
```

篩選條件模式可以包含中繼資料屬性、資料屬性或兩者。可用的中繼資料參數和資料參數的格式會根據作為事件來源的 AWS 服務 而變化。例如，假設您的事件來源映射從 Amazon SQS 佇列中收到以下記錄：

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "{\\n \"City\": \"Seattle\",\\n \"State\": \"WA\",\\n \"Temperature\":
 \"46\"\\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
    },
```

```
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
    "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
    "awsRegion": "us-east-2"
}
```

- 中繼資料屬性是包含有關建立該記錄之事件資訊的欄位。在 Amazon SQS 記錄範例中，中繼資料屬性包括 `messageID`、`eventSourceArn` 和 `awsRegion` 等欄位。
- 資料屬性是包含串流或佇列資料的記錄欄位。在 Amazon SQS 事件範例中，資料欄位的索引鍵為 `body`，而資料屬性為 `City`、`State` 和 `Temperature` 欄位。

不同類型的事件來源對其資料欄位使用不同的索引鍵值。若要篩選資料屬性，請務必在篩選條件模式中使用正確的索引鍵。如需資料篩選金鑰的清單，以及查看每個支援之篩選模式的範例 AWS 服務，請參閱 使用具有不同 的篩選條件 AWS 服務。

事件篩選可處理多級 JSON 篩選。例如，考慮 DynamoDB 串流中的下列記錄片段：

```
"dynamodb": {
    "Keys": {
        "ID": {
            "S": "ABCD"
        }
        "Number": {
            "N": "1234"
    },
    ...
}
```

假設您只想處理排序索引鍵 Number 值為 4567 的記錄。在這種情況下，您的 `FilterCriteria` 物件看起來像這樣：

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\": { \"Keys\": { \"Number\": { \"N\":
 [ "4567" ] } } } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 `Pattern` 在純文字 JSON 中擴展的值。

```
{
    "dynamodb": {
        "Keys": {
            "Number": {
                "N": [ "4567" ]
                }
            }
        }
}
```

處理不符合篩選條件標準的記錄

Lambda 如何處理不符合篩選條件的記錄，取決於事件來源。

- 針對 Amazon SQS，如果訊息不符合您的篩選條件標準，Lambda 會自動從佇列中刪除該訊息。您不需要在 Amazon SQS 中手動刪除這些訊息。
- 針對 Kinesis 和 DynamoDB，在您的篩選條件評估一筆記錄後，串流迭代器會向前移動通過此記錄。如果記錄不滿足篩選條件標準，則無需從事件來源中手動刪除記錄。保留期之後，Kinesis 和 DynamoDB 會自動刪除這些舊記錄。如果希望更快地刪除記錄，請參閱變更資料保留期間。
- 對於 Amazon MSK、自我管理的 Apache Kafka 和 Amazon MQ 訊息，Lambda 會捨棄不符合篩選條件中包含的所有欄位的訊息。針對 Amazon MSK 和自我管理 Apache Kafka，Lambda 會在成功調用函數之後，提交相符和不相符訊息的偏移量。若為 Amazon MQ，Lambda 會在成功調用函數後確認相符的訊息，並在篩選時確認不相符的訊息。

篩選條件規則語法

針對篩選條件規則，Lambda 支援 Amazon EventBridge 規則並使用與 EventBridge 相同的語法。如需詳細資訊，請參閱《Amazon EventBridge 使用者指南》中的 Amazon EventBridge 事件模式。

以下是可用於 Lambda 事件篩選的所有比較運算子摘要。

比較運算子	範例	Rule syntax (規則語法)
Null	UserID 為 Null (空值)	"UserID": [null]
空白	LastName 為空白	"LastName": [""]

比較運算子	範例	Rule syntax (規則語法)
Equals	名稱為「Alice」	"Name": ["Alice"]
等於 (忽略大小寫)	名稱為「Alice」	"Name": [{ "equals-ignore-case": "alice" }]
及	位置為「紐約」，日期是「週一」	"Location": ["New York"], "Day": ["Monday"]
或	PaymentType 為「信用卡」或「轉帳卡」	"PaymentType": ["Credit", "Debit"]
或 (多個欄位)	位置為「紐約」，或日期是「週一」。	"$or": [{ "Location": ["New York"] }, { "Day": ["Monday"] }]
Not	天氣是「下雨」以外的任何天氣	"Weather": [{ "anything-but": ["Raining"] }]
數字 (等於)	價格為 100	"Price": [{ "numeric": ["=", 100] }]
數字 (範圍)	價格大於 10，且小於或等於 20	"Price": [{ "numeric": [">", 10, "<=", 20] }]
存在	ProductName 已經存在	"ProductName": [{ "exists": true }]
不存在	ProductName 不存在	"ProductName": [{ "exists": false }]
開頭為	區域位於美國	"Region": [{"prefix": "us-" }]
Ends with	檔案名稱以 .png 副檔名做結尾。	"FileName": [{ "suffix": ".png" }]

> **Note**
>
> 正如 EventBridge 一般，就字串而言，Lambda 會在沒有大小寫折疊或任何其他字串標準化的情況下，使用精確字元比對。針對數字，Lambda 也會使用字串表示法。例如，300、300.0 和 3.0e2 不會被視為相等。

請注意，存在 (Exists) 運算子僅適用於事件來源 JSON 中的分葉節點。它與中繼節點不相符。例如，使用下列 JSON，篩選條件模式 `{ "person": { "address": [ { "exists": true } ] } }` 找不到相符項目，因為 `"address"` 是中繼節點。

```
{
  "person": {
    "name": "John Doe",
    "age": 30,
    "address": {
      "street": "123 Main St",
      "city": "Anytown",
      "country": "USA"
    }
  }
}
```

將篩選條件標準連接至事件來源映射 (主控台)

依照下列步驟使用 Lambda 主控台來建立具有篩選條件標準的新事件來源映射。

若要使用篩選條件標準 (主控台) 建立新事件來源映射

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要建立事件來源映射的函數名稱。
3. 在函數概觀下，選擇新增觸發條件。
4. 若為 Trigger configuration (觸發程序組態)，選擇支援事件篩選的觸發程序類型。如需支援的服務清單，請參閱本頁開頭的清單。
5. 展開 Additional settings (其他設定)。
6. 在 Filter criteria (篩選條件標準) 下，選擇 Add (新增)，然後定義並輸入您的篩選條件。例如，您可以輸入下列指令。

```
{ "Metadata" : [ 1, 2 ] }
```

這會指示 Lambda 僅處理欄位 Metadata 等於 1 或 2 的記錄。您可以繼續選取新增，新增更多篩選條件，直至達到允許的數目上限為止。

7. 完成新增篩選條件時，請選擇儲存。

使用主控台輸入篩選條件標準時，僅需輸入篩選模式，而不需要提供 Pattern 索引鍵或逸出引號。在上述指示的步驟 6 中，{ "Metadata" : [1, 2] } 會對應下列 FilterCriteria。

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

在主控台中建立事件來源映射之後，您可以在觸發程序詳細資訊中看見格式化的 FilterCriteria。如需有關使用主控台建立事件篩選條件的更多範例，請參閱 使用具有不同 的篩選條件 AWS 服務。

將篩選條件標準連接至事件來源映射 (AWS CLI)

假設您想要事件來源映射具有下列 FilterCriteria：

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

若要使用 AWS Command Line Interface (AWS CLI) 使用這些篩選條件建立新的事件來源映射，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
```

```
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

此 create-event-source-mapping 命令會使用指定的 FilterCriteria 為函數 my-function 建立新的 Amazon SQS 事件來源映射。

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"Metadata\" : [ 1, 2 ]}"}]}'
```

請注意，若要更新事件來源映射，您需要其 UUID。您可以從 list-event-source-mappings 呼叫中取得 UUID。Lambda 也會在 create-event-source-mapping CLI 回應中傳回 UUID。

若要從事件來源移除篩選條件，可以使用空白的 FilterCriteria 物件執行下列 update-event-source-mapping 命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria "{}"
```

如需使用 建立事件篩選條件的更多範例 AWS CLI，請參閱 使用具有不同 的篩選條件 AWS 服務。

將篩選條件標準連接至事件來源映射 (AWS SAM)

假設您想要在 中設定事件來源 AWS SAM ，以使用下列篩選條件：

```
{
   "Filters": [
      {
          "Pattern": "{ \"Metadata\" : [ 1, 2 ] }"
      }
   ]
}
```

若要將這些篩選條件標準新增到事件來源映射，請將下列程式碼片段插入事件來源的 YAML 範本中。

```
FilterCriteria:
  Filters:
    - Pattern: '{"Metadata": [1, 2]}'
```

如需建立和設定事件來源映射 AWS SAM 範本的詳細資訊，請參閱《 AWS SAM 開發人員指南》中的 EventSource 一節。如需使用 AWS SAM 範本建立事件篩選條件的更多範例，請參閱 使用具有不同 的篩選條件 AWS 服務。

篩選條件加密

依預設，Lambda 不會加密篩選條件物件。對於可以在篩選條件物件中包含敏感資訊的使用案例，您可以使用自己的 KMS 金鑰來加密它。

加密篩選條件物件後，可以使用 GetEventSourceMapping API 呼叫檢視其純文字版本。必須具有 `kms:Decrypt` 許可，才能成功以純文字檢視篩選條件。

> **Note**
>
> 如果篩選條件物件已加密，Lambda 會在 ListEventSourceMappings 呼叫回應中修訂 `FilterCriteria` 欄位的值。反之，此欄位會顯示為 `null`。若要查看真實的 `FilterCriteria` 值，請使用 GetEventSourceMapping API。
> 若要在主控台中檢視解密的 `FilterCriteria` 值，請確定您的 IAM 角色包含 GetEventSourceMapping 的許可。

您可以透過主控台、API/CLI 或 CloudFormation指定自己的 KMS 金鑰。

若要使用客戶擁有的 KMS 金鑰加密篩選條件 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇 Add trigger (新增觸發條件)。如果已有觸發條件，請選擇組態標籤，然後選擇觸發條件。選取現有觸發條件，然後選擇編輯。
3. 選取使用客戶管理的 KMS 金鑰加密旁邊的核取方塊。
4. 針對選擇客戶管理的 KMS 加密金鑰，選取現有的已啟用金鑰或建立新的金鑰。視操作而定，您需要以下部分或全部許可：`kms:DescribeKey`、`kms:GenerateDataKey` 和 `kms:Decrypt`。使用 KMS 金鑰政策來授予這些許可。

如果您使用自己的 KMS 金鑰，必須在金鑰政策中允許下列 API 操作：

- `kms:Decrypt` – 必須授予區域 Lambda 服務主體 (`lambda.`*`AWS_region`*`.amazonaws.com`)。這將允許 Lambda 使用此 KMS 金鑰解密資料。

  - 為了防止跨服務混淆代理人問題，金鑰政策會使用 aws:SourceArn 全域條件金鑰。aws:SourceArn 金鑰鍵的正確值是事件來源映射資源的 ARN，因此您只有在知道其 ARN 之後，才能將此值新增至政策。在向 KMS 提出解密請求時，Lambda 也會在加密內容中轉送 aws:lambda:FunctionArn 和 aws:lambda:EventSourceArn 金鑰及其個別值。這些值必須符合金鑰政策中指定的條件，解密請求才會成功。不需要包含 EventSourceArn for Self-managed Kafka 事件來源，因為它們沒有 EventSourceArn。
- kms:Decrypt – 還必須授予想要使用金鑰在 GetEventSourceMapping 或 DeleteEventSourceMapping API 呼叫中檢視純文字篩選條件的主體。
- kms:DescribeKey — 提供客戶管理之金鑰的詳細資訊，以便指定主體可以使用金鑰。
- kms:GenerateDataKey – 提供許可，讓 Lambda 代表指定的主體 (封套加密) 產生資料金鑰來加密篩選條件。

您可以使用 AWS CloudTrail 來追蹤 Lambda 代表您提出的 AWS KMS 請求。如需 CloudTrail 事件範例，請參閱 ???。

我們也建議使用 kms:ViaService 條件金鑰，將 KMS 金鑰的使用限制為僅限來自 Lambda 的請求。此金鑰的值是區域 Lambda 服務主體 (lambda.*AWS_region*.amazonaws.com)。以下是授予所有相關許可的金鑰政策範例：

Example AWS KMS 金鑰政策

JSON

```
{
    "Version":"2012-10-17",
    "Id": "example-key-policy-1",
    "Statement": [
        {
            "Sid": "Allow Lambda to decrypt using the key",
            "Effect": "Allow",
            "Principal": {
                "Service": "lambda.us-east-1.amazonaws.com"
            },
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": "*",
            "Condition": {
                "ArnEquals" : {
```

```
                    "aws:SourceArn": [
                        "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:<esm_uuid>"
                    ]
                },
                "StringEquals": {

 "kms:EncryptionContext:aws:lambda:FunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:test-function",

 "kms:EncryptionContext:aws:lambda:EventSourceArn": "arn:aws:sqs:us-
east-1:123456789012:test-queue"
                }
            }
        },
        {
            "Sid": "Allow actions by an AWS account on the key",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:root"
            },
            "Action": "kms:*",
            "Resource": "*"
        },
        {
            "Sid": "Allow use of the key to specific roles",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::123456789012:role/ExampleRole"
            },
            "Action": [
                "kms:Decrypt",
                "kms:DescribeKey",
                "kms:GenerateDataKey"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals" : {
                    "kms:ViaService": "lambda.us-east-1.amazonaws.com"
                }
            }
        }
    ]
```

```
}
```

若要使用您自己的 KMS 金鑰來加密篩選條件，您也可以使用下列 CreateEventSourceMapping AWS CLI 命令。使用 `--kms-key-arn` 旗標指定 KMS 金鑰 ARN。

```
aws lambda create-event-source-mapping --function-name my-function \
    --maximum-batching-window-in-seconds 60 \
    --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
    --filter-criteria "{\"filters\": [{\"pattern\": \"{\"a\": [\"1\", \"2\"]}\" }]}" \
    --kms-key-arn arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

如果已有事件來源映射，請改用 UpdateEventSourceMapping AWS CLI 命令。使用 `--kms-key-arn` 旗標指定 KMS 金鑰 ARN。

```
aws lambda update-event-source-mapping --function-name my-function \
    --maximum-batching-window-in-seconds 60 \
    --event-source-arn arn:aws:sqs:us-east-1:123456789012:my-queue \
    --filter-criteria "{\"filters\": [{\"pattern\": \"{\"a\": [\"1\", \"2\"]}\" }]}" \
    --kms-key-arn arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599
```

此操作會覆寫先前指定的 KMS 金鑰。如果您指定 `--kms-key-arn` 旗標和空引數，Lambda 會停止使用 KMS 金鑰來加密篩選條件。反之，Lambda 會預設回使用 Amazon 擁有的金鑰。

若要在 CloudFormation 範本中指定您自己的 KMS 金鑰，請使用 `AWS::Lambda::EventSourceMapping` 資源類型的 `KMSKeyArn` 屬性。例如，可以將下列程式碼片段插入事件來源的 YAML 範本。

```
MyEventSourceMapping:
  Type: AWS::Lambda::EventSourceMapping
  Properties:
    ...
    FilterCriteria:
      Filters:
        - Pattern: '{"a": [1, 2]}'
    KMSKeyArn: "arn:aws:kms:us-east-1:123456789012:key/055efbb4-xmpl-4336-
ba9c-538c7d31f599"
    ...
```

若要能夠在 GetEventSourceMapping 或 DeleteEventSourceMapping API 呼叫中以純文字檢視加密的篩選條件，必須具備 kms:Decrypt 許可。

自 2024 年 8 月 6 日起，如果您的函式未使用事件篩選功能，則 CreateEventSourceMapping、UpdateEventSourceMapping 與 DeleteEventSourceMapping 等 API 呼叫的 AWS CloudTrail 日誌中，將不再顯示 FilterCriteria 欄位。如果您的函數使用事件篩選，FilterCriteria 欄位會顯示為空白 ({})。如果您具有正確 KMS 金鑰的 kms:Decrypt 許可，仍然可以在 GetEventSourceMapping API 呼叫的回應中以純文字檢視篩選條件。

Create/Update/DeleteEventSourceMapping 呼叫的 CloudTrail 日誌項目範例

在 CreateEventSourceMapping 呼叫的下列 AWS CloudTrail 範例日誌項目中， FilterCriteria 會顯示為空白 ({})，因為函數使用事件篩選。即使 FilterCriteria 物件包含函數正在使用的有效篩選條件，也是如此。如果函數不使用事件篩選，CloudTrail 絕對不會在日誌項目中顯示 FilterCriteria 欄位。


```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:userid1",
        "arn": "arn:aws:sts::123456789012:assumed-role/Example/example-role",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA987654321EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/User1",
                "accountId": "123456789012",
                "userName": "User1"
            },
            "webIdFederationData": {},
            "attributes": {
                "creationDate": "2024-05-09T20:35:01Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "AWS Internal"
    },
    "eventTime": "2024-05-09T21:05:41Z",
    "eventSource": "lambda.amazonaws.com",
```

```
    "eventName": "CreateEventSourceMapping20150331",
    "awsRegion": "us-east-2",
    "sourceIPAddress": "AWS Internal",
    "userAgent": "AWS Internal",
    "requestParameters": {
        "eventSourceArn": "arn:aws:sqs:us-east-2:123456789012:example-queue",
        "functionName": "example-function",
        "enabled": true,
        "batchSize": 10,
        "filterCriteria": {},
        "kMSKeyArn": "arn:aws:kms:us-east-2:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "scalingConfig": {},
        "maximumBatchingWindowInSeconds": 0,
        "sourceAccessConfigurations": []
    },
    "responseElements": {
        "uUID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
        "batchSize": 10,
        "maximumBatchingWindowInSeconds": 0,
        "eventSourceArn": "arn:aws:sqs:us-east-2:123456789012:example-queue",
        "filterCriteria": {},
        "kMSKeyArn": "arn:aws:kms:us-east-2:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:example-
function",
        "lastModified": "May 9, 2024, 9:05:41 PM",
        "state": "Creating",
        "stateTransitionReason": "USER_INITIATED",
        "functionResponseTypes": [],
        "eventSourceMappingArn": "arn:aws:lambda:us-east-2:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb"
    },
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE33333",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "sessionCredentialFromConsole": "true"
}
```

使用具有不同 的篩選條件 AWS 服務

不同類型的事件來源對其資料欄位使用不同的索引鍵值。若要篩選資料屬性，請務必在篩選條件模式中使用正確的索引鍵。下表提供每個支援的篩選金鑰 AWS 服務。

AWS 服務	篩選索引鍵
DynamoDB	`dynamodb`
Kinesis	`data`
Amazon MQ	`data`
Amazon MSK	`value`
自我管理的 Apache Kafka	`value`
Amazon SQS	`body`

下列各節提供不同類型事件來源的篩選模式範例。其還為每個支援之服務提供支援的傳入資料格式和篩選條件模式內文格式的定義。

- the section called “事件篩選”
- the section called “事件篩選”
- the section called “事件篩選”
- the section called “事件篩選”
- the section called “事件篩選”

在主控台中，測試 Lambda 函數。

您可以藉由使用測試事件調用函數，在主控台中測試您的 Lambda 函數。一個測試事件是函數的 JSON 輸入。如果您的函數不需要輸入，則該事件可以是空白的文件 (`{}`)。

當您在主控台執行測試時，Lambda 會與測試事件同步調用函數。函數執行期會將 JSON 事件轉換為物件，並將其傳遞給程式碼的處理常式方法以進行處理。

> **建立測試事件**
>
> 您必須先建立私有或可共用的測試事件，才能在主控台中測試。

使用測試事件調用函數

若要測試函數

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要測試的函數名稱。
3. 選擇測試標籤。
4. 在測試事件下，選擇建立新事件或編輯已儲存事件，然後選擇您要使用的已儲存事件。
5. (選擇性) - 選擇事件 JSON 的範本。
6. 選擇測試。
7. 若要檢閱測試結果，在 Execution result (執行結果) 下，展開 Details (詳細資訊)。

若要在不儲存測試事件的情況下調用您的函數，選擇測試，然後再儲存。這會建立一個未儲存的測試事件，而 Lambda 僅會在工作階段期間保留該事件。

對於 Node.js、Python 和 Ruby 執行時期，您也可以在程式碼索引標籤上存取現有的已儲存和未儲存的測試事件。使用測試事件區段來建立、編輯和執行測試。

建立私有測試事件

只有事件建立者才能使用私有測試事件，且其不需要額外的許可即可使用。每個函數您可以建立並儲存最多 10 個私有測試事件。

若要建立私有測試事件

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要測試的函數名稱。
3. 選擇測試標籤。
4. 在 Test event (測試事件) 下，執行以下動作：
   a. 選擇 Template (範本)。
   b. 輸入該測試的 Name (名稱)。
   c. 在文字輸入方塊中，輸入 JSON 測試事件。
   d. 在 Event sharing settings (事件共用設定) 下，選擇 Private (私有)。
5. 選擇儲存變更。

對於 Node.js、Python 和 Ruby 執行時期，您也可以在程式碼索引標籤上建立測試事件。使用測試事件區段來建立、編輯和執行測試。

建立可共用測試事件

可共用的測試事件是您可以與相同 AWS 帳戶中的其他使用者共用的測試事件。您可以編輯其他使用者的可共用測試事件，並使用它們來調用您的函數。

Lambda 將可共用測試事件以結構描述的形式，儲存在名為 `lambda-testevent-schemas` 的 Amazon EventBridge (CloudWatch Events) 結構描述登錄檔中。由於 Lambda 利用此登錄檔來存放和呼叫您建立的可共用測試事件，我們建議您不要編輯此登錄檔或使用 `lambda-testevent-schemas` 名稱建立登錄檔。

若要查看、共用和編輯可共用測試事件，您必須具有以下所有EventBridge (CloudWatch Events) 結構描述登錄檔 API 操作的許可：

- `schemas.CreateRegistry`
- `schemas.CreateSchema`
- `schemas.DeleteSchema`
- `schemas.DeleteSchemaVersion`
- `schemas.DescribeRegistry`
- `schemas.DescribeSchema`
- `schemas.GetDiscoveredSchema`

- schemas.ListSchemaVersions
- schemas.UpdateSchema

請注意，儲存對可共用測試事件所做的編輯將覆蓋該事件。

如果您無法建立、編輯或查看可共用測試事件，請檢查您的帳戶是否具有執行這些操作所需要的許可。如果您擁有所需要的許可，但仍然無法存取可共用測試事件，請檢查任何可能會限制對 EventBridge (CloudWatch Events) 登錄檔的存取的資源型政策。

若要建立可共用測試事件

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要測試的函數名稱。
3. 選擇測試標籤。
4. 在 Test event (測試事件) 下，執行以下動作：
   a. 選擇 Template (範本)。
   b. 輸入該測試的 Name (名稱)。
   c. 在文字輸入方塊中，輸入 JSON 測試事件。
   d. 在 Event sharing settings (事件共用設定) 下，選擇 Shareable (可共用)。
5. 選擇儲存變更。

> 搭配 使用可共用的測試事件 AWS Serverless Application Model。
>
> 您可以使用 AWS SAM 叫用可共用的測試事件。請參閱《AWS Serverless Application Model 開發人員指南》中的 sam remote test-event。

刪除可共用測試事件結構描述

當您刪除可共用測試事件時，Lambda 會將其從 lambda-testevent-schemas 登錄檔中移除。如果您是從登錄檔移除最後一個可共用測試事件，則 Lambda 會刪除登錄檔。

如果刪除函數，Lambda 不會刪除任何關聯的可共用測試事件結構描述。您必須從 EventBridge (CloudWatch Events) 主控台手動清理這些資源。

Lambda 函數狀態

所有 Lambda 函式的函式組態中都包含一個狀態欄位，用於指示函式何時準備就緒可供調用。State 會提供函式目前狀態的相關資訊，包括是否可以成功調用函式。函數狀態不會變更函數叫用的行為或函數執行程式碼的方式。

> **Note**
>
> SnapStart 函式的函式狀態定義略有不同。如需詳細資訊，請參閱Lambda SnapStart 和函數狀態。

在許多情況下，DynamoDB 資料表是在調用間隔期間保留狀態的理想方式，因為其提供低延遲資料存取，並且可以隨著 Lambda 服務擴展。如果您使用 Amazon EFS for Lambda，也可以將資料儲存在此服務中，這樣也可以實現對檔案系統儲存的低延遲存取。

函數狀態包括：

- `Pending` – Lambda 建立函數之後，會將狀態設定為待定中。處於待定狀態時，Lambda 會嘗試建立或設定函數的資源，例如 VPC 或 EFS 資源。Lambda 不會叫用待定狀態期間的函數。在函數上操作的任何叫用或其他 API 操作都會失敗。
- `Active` – Lambda 完成資源組態和佈建之後，您的函數會轉換為啟用中狀態。只有啟用中的函數才能成功叫用。
- `Failed` – 表示資源配置或佈建發生錯誤。當函數建立失敗時，Lambda 會將函數狀態設定為失敗，而且您必須刪除並重新建立函數。
- `Inactive` – 當函數空閒足夠長的時間，以至於 Lambda 回收為其配置的外部資源時，該函數將變為非啟用狀態。當您嘗試叫用非啟用中的函數，叫用會失敗而 Lambda 會將函數設定為待定狀態，直到重新建立函數資源為止。如果 Lambda 無法重新建立資源，則函數會返回到非啟用狀態。您可能需要解決錯誤後再重新部署函式，將其還原為作用中狀態。

如果您是使用 SDK 型自動化工作流程或是直接呼叫 Lambda 的服務 API，請務必在調用之前檢查函數的狀態，以驗證其是否處於作用中狀態。您可以使用 Lambda API 動作 GetFunction 來完成此動作，或者透過使用 AWS SDK for Java 2.0 配置等候程序。

```
aws lambda get-function --function-name my-function --query 'Configuration.[State,
 LastUpdateStatus]'
```

您應該會看到下列輸出：

```
[
 "Active",
 "Successful"
]
```

當函數的建立處於待定狀態時，下列作業會失敗：

- Invoke
- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

更新期間的函式狀態

Lambda 提供兩種更新函式的操作：

- UpdateFunctionCode：更新函式的部署套件
- UpdateFunctionConfiguration：更新函式的組態

Lambda 使用 LastUpdateStatus 屬性來追蹤這些更新操作的進度。當更新正在進行時 (即 `"LastUpdateStatus": "InProgress"`)：

- 函式的狀態會保持 `Active`。
- 在更新完成之前，調用仍會使用函式的舊版程式碼與組態。
- 下列操作會失敗：
  - UpdateFunctionCode
  - UpdateFunctionConfiguration
  - PublishVersion
  - TagResource

當更新失敗時（當 時`"LastUpdateStatus": "Failed"`)：

- 函式的狀態會保持 `Active`。

- 調用會繼續使用函數先前的程式碼和組態。

Example GetFunctionConfiguration 回應

下列範例展示正在更新的函式執行 GetFunctionConfiguration 請求所傳回的結果。

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
    "Runtime": "nodejs24.x",
    "VpcConfig": {
        "SubnetIds": [
            "subnet-071f712345678e7c8",
            "subnet-07fd123456788a036",
            "subnet-0804f77612345cacf"
        ],
        "SecurityGroupIds": [
            "sg-085912345678492fb"
        ],
        "VpcId": "vpc-08e1234569e011e83"
    },
    "State": "Active",
    "LastUpdateStatus": "InProgress",
    ...
}
```

了解 Lambda 中的重試行為

直接叫用函數時，您可以決定處理函數程式碼相關錯誤的策略。Lambda 不會自動代您重試這類錯誤。若要進行重試，您可以手動重新叫用函數、將失敗的事件傳送到佇列進行偵錯，或忽略錯誤。您函式的程式碼可能已完全執行、部分執行或完全未執行。如果您重試，請確保您函式的程式碼可處理相同的事件很多次，而不會導致重複的交易或其他不想要的副作用。

當您間接叫用函式時，您需要留意叫用端的重試行為及任何遭遇請求的服務。這包括以下案例。

- Asynchronous invocation (非同步叫用) - Lambda 會重試函數錯誤兩次。如果函數沒有足夠的容量來處理所有傳入的請求，事件可能在佇列中等待數小時才會傳送到函數。您可以在函式上設定無效信件佇列，以擷取未成功處理的事件。如需更多詳細資訊，請參閱 the section called “無效信件佇列”。
- Event source mappings (事件來源映射) - 從串流中讀取的事件來源映射會重試整批項目。在錯誤解決或項目過期前，重複的錯誤會阻礙受影響碎片的處理。若要偵測已停滯的碎片，您可以監控反覆運算器年齡指標。

  對於從佇列讀取的事件來源映射，您可藉由設定來源佇列的可見性逾時和再驅動政策，決定重試之間的時間長度和失敗事件的目的地。如需詳細資訊，請參閱 Lambda 如何處理來自串流和佇列式事件來源的記錄 和 使用來自其他服務的事件叫用 Lambda AWS 下的服務特定主題。
- AWS 服務 – AWS 服務可以同步或非同步地調用您的函式。對於同步叫用，服務會決定是否要重試。例如，如果 Lambda 函數傳回 `TemporaryFailure` 回應代碼，則 Amazon S3 批次操作會重試該操作。代理來自上游使用者或用戶端之請求的服務可能有重試策略，或可能會將錯誤回應轉送回請求者。例如，API Gateway 一律會將錯誤回應轉送回請求者。

  對於非同步調用，無論調用來源為何，重試邏輯都相同。依預設，Lambda 最多會重試失敗的非同步調用兩次。如需更多詳細資訊，請參閱 Lambda 如何處理非同步調用的錯誤和重試。
- Other accounts and clients (其他帳戶和用戶端) - 當您授與其他帳戶的存取權時，您可使用以資源為基礎的政策來限制其可設定的服務或資源，以叫用您的函數。為了保護您的函數以免過載，請考慮透過 Amazon API Gateway 在您的函數前面放置 API 層。

為了協助您處理 Lambda 應用程式中的錯誤，Lambda 會與 Amazon CloudWatch 和 AWS X-Ray 等服務整合。您可以使用日誌、指標、警示和追蹤的組合，快速偵測及識別您的函式程式碼、API 或其他支援您應用程式的資源中的問題。如需更多詳細資訊，請參閱 監控、偵錯與疑難排解 Lambda 函式。

使用 Lambda 遞迴迴圈偵測功能防止無限迴圈

當您將 Lambda 函數設定為輸出至調用該函數的相同服務或資源時，就可以建立無限遞迴迴圈。例如，Lambda 函數可能會將訊息寫入 Amazon Simple Queue Service (Amazon SQS) 佇列，然後調用相同的函數。此調用會導致函數將另一則訊息寫入佇列，進而再次調用函數。

意外的遞迴迴圈可能會導致意外費用向您的 收費 AWS 帳戶。迴圈也可能導致 Lambda 擴展和使用您帳戶的所有可用並行處理。為了減少意外迴圈的影響，Lambda 可在特定類型的遞迴迴圈發生後不久偵測到它們。當 Lambda 偵測到遞迴迴圈時，預設會停止調用函數並通知您。如果您的設計刻意使用遞迴模式，您可以變更函數的預設組態，允許以遞迴方式調用它。如需詳細資訊，請參閱the section called “允許 Lambda 函數在遞迴迴圈中執行”。

章節

- 了解遞迴迴圈偵測
- 支援的 AWS 服務 和 SDKs
- 遞迴迴圈通知
- 回應遞迴迴圈偵測通知
- 允許 Lambda 函數在遞迴迴圈中執行
- 支援 Lambda 遞迴迴圈偵測功能的區域

了解遞迴迴圈偵測

Lambda 中的遞迴迴圈偵測透過追蹤事件來運作。Lambda 是事件驅動型運算服務，可在特定事件發生時執行函數程式碼。例如，將項目新增至 Amazon SQS 佇列或 Amazon Simple Notification Service (Amazon SNS) 主題時。Lambda 會將事件以 JSON 物件的形式傳遞至函數，其中包含系統狀態中的變更資訊。當事件導致函數執行時，這稱為調用。

若要偵測遞迴迴圈，Lambda 會使用 AWS X-Ray 追蹤標頭。當支援遞迴迴圈偵測的AWS 服務 將事件傳送至 Lambda 時，這些事件會自動使用中繼資料進行註解。當您的 Lambda 函數 AWS 服務 使用支援的 AWS SDK 版本將其中一個事件寫入到另一個支援的事件時，它會更新此中繼資料。更新後的中繼資料包含事件調用函數的次數計數。

> **Note**
>
> 您不需要啟用 X-Ray 作用中追蹤，即可使用此功能。依預設，所有 AWS 客戶都會開啟遞迴迴路偵測。使用此功能無需支付任何費用。

請求鏈是由相同觸發事件引起的一系列 Lambda 調用。例如，假設 Amazon SQS 佇列調用 Lambda 函數。然後，Lambda 函數會將已處理的事件傳回相同的 Amazon SQS 佇列，然後再次調用函數。在此範例中，函數的每次調用都屬於相同的請求鏈。

如果您的函數在相同的請求鏈中被調用大約 16 次，則 Lambda 會自動停止該請求鏈中的下一個函數調用並通知您。如果函數設有多個觸發條件，來自其他觸發條件的調用不會受影響。

> **Note**
>
> 即使來源佇列再驅動政策的 `maxReceiveCount` 設定高於 16，Lambda 遞迴保護也不會阻止 Amazon SQS 在偵測到遞迴迴圈並終止後重試訊息。當 Lambda 偵測到遞迴迴圈並捨棄後續調用時，會將 `RecursiveInvocationException` 傳回至事件來源映射。這會增加訊息上的 `receiveCount` 值。Lambda 會繼續重試訊息，並繼續封鎖函數調用，直至 Amazon SQS 判斷超過 `maxReceiveCount`，並將訊息傳送至設定的無效字母佇列。

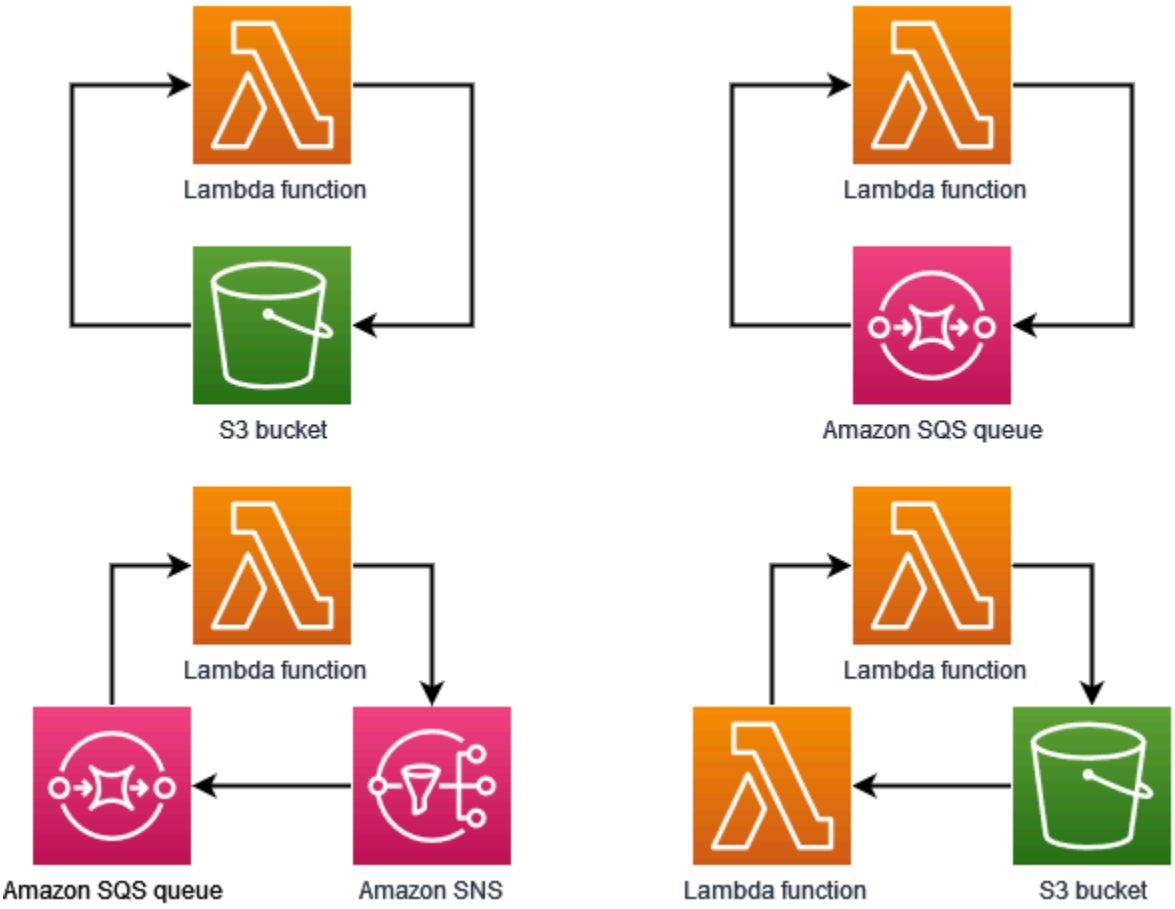
如果您為函數設定了故障時的目的地或無效字母佇列，則 Lambda 也會將事件從已停止的調用中傳送至目的地或無效字母佇列。為函式設定目的地或無效字母佇列時，請勿使用函式也在使用的事件觸發程序或事件來源映射。若將事件傳送至調用函式的同一個資源，則可能會建立另一個遞迴迴圈，且此迴圈同樣會被終止。若關閉遞迴迴圈偵測功能，此類迴圈不會被終止。

支援的 AWS 服務 和 SDKs

Lambda 只能偵測包含特定支援的遞迴迴圈 AWS 服務。若要偵測遞迴迴圈，您的函數也必須使用其中一個支援的 AWS SDKs。

支援的 AWS 服務

Lambda 目前可偵測函數、Amazon SQS、Amazon S3 和 Amazon SNS 之間的遞迴迴圈。Lambda 也會偵測僅由 Lambda 函數組成的迴圈，這些函數可以同步或非同步相互調用。下圖顯示 Lambda 可以偵測的一些迴圈範例：



當 Amazon DynamoDB AWS 服務 等其他 形成迴圈的一部分時，Lambda 目前無法偵測並停止它。

由於 Lambda 目前僅偵測涉及 Amazon SQS、Amazon S3 和 Amazon SNS 的遞迴迴圈，因此涉及其他 的迴圈仍可能導致 Lambda 函數 AWS 服務 意外使用。

為了防止向您的 收取意外費用 AWS 帳戶，建議您設定 Amazon CloudWatch 警示，以提醒您不尋常的使用模式。例如，可以設定 CloudWatch，以便在 Lambda 函數並行處理或調用出現峰值時通知您。也可以設定帳單警示，當帳戶中的支出超過指定的閾值時通知您。或者，可以使用 AWS Cost Anomaly Detection 來提醒您不尋常的帳單模式。

支援的 AWS SDKs

若要讓 Lambda 偵測遞迴迴圈，函數必須使用下列其中一個 SDK 版本或更高版本：

執行時期	最低必要 AWS SDK 版本
Node.js	2.1147.0 (SDK 版本 2)

執行時期	最低必要 AWS SDK 版本
	3.105.0 (SDK 版本 3)
Python	1.24.46 (boto3) 1.27.46 (botocore)
Java 8 和 Java 11	2.17.135
Java 17	2.20.81
Java 21	2.21.24
.NET	3.7.293.0
Ruby	3.134.0
PHP	3.232.0
Go	V2 SDK 1.57.0

Python 和 Node.js 等某些 Lambda 執行時間包含 AWS SDK 的版本。如果函數執行期中包含的 SDK 版本低於所需的最低版本，則可以將支援的 SDK 版本新增到函數的部署套件。您也可以使用 Lambda 層將支援的 SDK 版本新增至函數。如需每個 Lambda 執行期包含的 SDK 清單，請參閱 Lambda 執行期。

遞迴迴圈通知

當 Lambda 停止遞迴迴圈時，您會透過 Health 儀板表 和電子郵件接收通知。也可以使用 CloudWatch 指標來監控 Lambda 已停止的遞迴調用次數。

Health 儀板表 通知

當 Lambda 停止遞迴調用時， 會在您的帳戶運作狀態頁面的開啟和最近問題下 Health 儀板表 顯示通知。請注意，在 Lambda 停止遞迴調用後，可能需要 3.5 小時才會顯示此通知。如需有關在 中檢視帳戶事件的詳細資訊 Health 儀板表，請參閱《AWS 運作狀態使用者指南》中的運作狀態儀表板 – 您的帳戶運作狀態入門。 AWS

電子郵件提醒

當 Lambda 第一次停止函數的遞迴調用時，會傳送電子郵件提醒給您。對於 AWS 帳戶中的每個函數，Lambda 每 24 小時最多傳送一封電子郵件。Lambda 傳送電子郵件通知後，即使 Lambda 停止函數的進一步遞迴調用，在接下來的 24 小時也不會再收到該函數的任何電子郵件。請注意，在 Lambda 停止遞迴調用後，可能需要 3.5 小時才會收到此電子郵件提醒。

Lambda 會傳送遞迴迴圈電子郵件提醒給您 AWS 帳戶的主要帳戶聯絡人和替代操作聯絡人。如需有關檢視或更新帳戶中電子郵件地址的資訊，請參閱《AWS 一般參考》中的更新聯絡資訊。

Amazon CloudWatch 指標

CloudWatch 指標`RecursiveInvocationsDropped`會記錄 Lambda 已停止的函數調用次數，因為您的函數已在單一請求鏈中調用大約 16 次。Lambda 會在停止遞迴調用時立即發出此指標。若要檢視此指標，請按照在 CloudWatch 主控台上檢視指標的說明進行操作，並選擇指標 `RecursiveInvocationsDropped`。

回應遞迴迴圈偵測通知

當函數被同一觸發事件調用大約 16 次時，Lambda 會停止該事件的下一個函數調用，以中斷遞迴迴圈。若要防止 Lambda 已中斷的遞迴迴圈再次發生，請執行下列動作：

- 將函數的可用並行處理降為零，這會限制所有將來的調用。
- 移除或停用正在調用函數的觸發條件或事件來源映射。
- 識別並修正程式碼瑕疵，將事件寫回叫用函數 AWS 的資源。當您使用變數來定義函數的事件來源和目標時，就會出現常見的缺陷來源。檢查兩個變數沒有使用相同的值。

此外，如果 Lambda 函數的事件來源是 Amazon SQS 佇列，則請考慮在來源佇列上設定無效字母佇列。

> **Note**
>
> 請確定在來源佇列上設定無效字母佇列，而不是在 Lambda 函數上。您在函數上設定的無效字母佇列用於佇列的非同步調用函數，而不是事件來源佇列。

如果事件來源是 Amazon SNS 主題，則請考慮為函數新增故障時的目的地。

將函數的可用並行處理降為零 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇 函數的名稱。
3. 選擇調節。
4. 在調節函數對話方塊中，選擇確認。

若要移除函數的觸發條件或事件來源映射 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇 函數的名稱。
3. 選擇組態索引標籤，然後觸發條件。
4. 在觸發條件下，選取要刪除的觸發條件或事件來源映射，然後選擇刪除。
5. 在刪除觸發條件對話方塊中，選擇刪除。

若要停用函數的事件來源映射 (AWS CLI)

1. 若要尋找您要停用之事件來源映射的 UUID，請執行 AWS Command Line Interface (AWS CLI) list-event-source-mappings 命令。

```
aws lambda list-event-source-mappings
```

2. 若要停用事件來源映射，請執行 following AWS CLI update-event-source-mapping 命令。

```
aws lambda update-event-source-mapping --function-name MyFunction \
--uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 --no-enabled
```

允許 Lambda 函數在遞迴迴圈中執行

如果您的設計刻意使用遞迴迴圈，您可以設定 Lambda 函數，以允許以遞迴方式調用它。我們建議您避免在設計中使用遞迴迴圈。實作錯誤可能會導致使用您所有 AWS 帳戶可用的並行進行遞迴調用，以及向您的 帳戶收取意外費用。

> **Important**
>
> 如果您使用遞迴迴圈，請謹慎處理。實作最佳實務防護措施，將實作錯誤的風險降至最低。若要進一步了解使用遞迴模式的最佳實務，請參閱 Serverless Land 中的 Recursive patterns that cause run-away Lambda functions。

您可以設定函數，以允許使用 Lambda 主控台、 AWS Command Line Interface (AWS CLI) 和 PutFunctionRecursionConfig API 進行遞迴迴圈。您也可以在 AWS SAM 和 中設定函數的遞迴迴圈偵測設定 CloudFormation。

依預設，Lambda 會偵測並終止遞迴迴圈。除非設計刻意使用遞迴迴圈，否則建議不要變更函數的預設組態。

請注意，當您設定函數以允許遞迴迴圈時，不會發出 CloudWatch 指標 `RecursiveInvocationsDropped`。

Console

若要允許函數在遞迴迴圈中執行 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函數的名稱以開啟函數詳細資訊頁面。
3. 選擇組態標籤，然後選擇並行和遞迴偵測。
4. 除了遞迴迴圈偵測，選擇編輯。
5. 選取允許遞迴迴圈。
6. 選擇儲存。

AWS CLI

您可以使用 PutFunctionRecursionConfig API 來允許在遞迴迴圈中調用函數。為遞迴迴圈參數指定 `Allow`。例如，您可以使用 `put-function-recursion-config` AWS CLI 命令呼叫此 API：

```
aws lambda put-function-recursion-config --function-name yourFunctionName --
recursive-loop Allow
```

您可以將函數的組態變更回預設設定，以便 Lambda 在偵測到遞迴迴圈時將其終止。使用 Lambda 主控台或 AWS CLI編輯函數的組態。

Console

若要設定函數以終止遞迴迴圈 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函數的名稱以開啟函數詳細資訊頁面。
3. 選擇組態標籤，然後選擇並行和遞迴偵測。
4. 除了遞迴迴圈偵測，選擇編輯。
5. 選取終止遞迴迴圈。
6. 選擇儲存。

AWS CLI

您可以使用 PutFunctionRecursionConfig API 來設定函數，以便 Lambda 在偵測到遞迴迴圈時將其終止。為遞迴迴圈參數指定 `Terminate`。例如，您可以使用 `put-function-recursion-config` AWS CLI 命令呼叫此 API：

```
aws lambda put-function-recursion-config --function-name yourFunctionName --
recursive-loop Terminate
```

支援 Lambda 遞迴迴圈偵測功能的區域

所有商業區域都支援 Lambda 遞迴迴圈偵測，但墨西哥 (中部) 與亞太區域 (紐西蘭) 除外。

建立及管理 Lambda 函數 URL

函數 URL 是 Lambda 函數專用的 HTTP(S) 端點。您可以透過 Lambda 主控台或 Lambda API 建立及設定函數 URL。

> **Tip**
>
> Lambda 提供兩種透過 HTTP 端點調用函數的方式：函數 URL 與 Amazon API Gateway。如果不確定哪種方法最適合自己的使用案例，請參閱the section called “函式 URL 與 Amazon API Gateway 比較”。

當您建立函數 URL 時，Lambda 會自動為您產生不重複的 URL 端點。函數 URL 一旦建立，其 URL 端點便永遠不會變更。函數 URL 端點的格式如下：

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> 下列 不支援函數 URLs AWS 區域：亞太區域 （ 海德拉巴 ） (ap-south-2)、亞太區域 （ 墨爾本 ） (ap-southeast-4)、亞太區域 （ 馬來西亞 ） (ap-southeast-5)、亞太區域 （ 紐西蘭 ） (ap-southeast-6)、亞太區域 （ 泰國 ） (ap-southeast-7)、亞太區域 （ 台北 ） (ap-east-2)、加拿大西部 （ 卡加利 ） (ca-west-1)、歐洲 （ 西班牙 ） (eu-south-2)、歐洲 （ 蘇黎世 ） (eu-central-2)、以色列 （ 特拉維夫 ） (il-central-1) 和中東 （ 阿拉伯聯合大公國 ） ()me-central-1)。

函數 URL 可支援雙堆疊，能同時支援 IPv4 和 IPv6。為函數設定函數 URL 後，您可以利用 Web 瀏覽器、curl、Postman 或任何 HTTP 用戶端，透過 HTTP(S) 端點呼叫函數。

> **Note**
>
> 您只能透過公有網際網路存取您的函數 URL。雖然 Lambda 函數確實支援 AWS PrivateLink，但函數 URL 不支援。

Lambda 函數 URL 使用以資源為基礎的政策，妥善控管安全性和存取權。函數 URL 也支援跨來源資源共用 (CORS) 組態選項。

您可以將函數 URL 套用至任何函數別名或未發佈的$LATEST函數版本。函數 URL 無法新增至其他任何函數版本。

下一節說明如何使用 Lambda 主控台和 CloudFormation 範本建立 AWS CLI和管理函數 URL

主題

- 建立函數 URL (主控台)
- 建立函數 URL (AWS CLI)
- 將函數 URL 新增至 CloudFormation 範本
- 跨來源資源共享 (CORS)
- 調節函數 URL
- 停用函數 URL
- 刪除函數 URL
- 控制對 Lambda 函數 URL 的存取
- 呼叫 Lambda 函數 URL
- 監控 Lambda 函數 URL
- 選取一種使用 HTTP 請求調用 Lambda 函數的方法
- 教學課程：使用 Lambda 函式 URL 建立 Webhook 端點

建立函數 URL (主控台)

請遵循下列步驟，使用主控台建立函數 URL。

為現有函數建立函數 URL

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您要為其建立函數 URL 的函數名稱。
3. 選擇 Configuration (組態) 標籤，然後選擇 Function URL (函數 URL)。
4. 選擇 Create function URL (建立函數 URL)。
5. 為 Auth type (驗證類型) 選擇 AWS_IAM 或 NONE (無)。如需函數 URL 身分驗證的詳細資訊，請參閱存取控制。
6. (選用) 選取 Configure cross-origin resource sharing (CORS) (設定跨來源資源共享 (CORS))，然後完成函數 URL 的 CORS 設定。如需 CORS 的詳細資訊，請參閱「跨來源資源共享 (CORS)」。

7. 選擇儲存。

這樣即可為 $LATEST 未發佈版本的函數建立函數 URL。函數 URL 會顯示於主控台的 Function overview (函數概觀) 區段。

為現有別名建立函數 URL

1. 開啟 Lambda 主控台中的函數頁面。
2. 找到您要為其建立函數 URL 的函數別名，選擇其函數名稱。
3. 選擇 Aliases (別名) 標籤，接著找到您要為其建立函數 URL 的函數別名，選擇該別名的名稱。
4. 選擇 Configuration (組態) 標籤，然後選擇 Function URL (函數 URL)。
5. 選擇 Create function URL (建立函數 URL)。
6. 為 Auth type (驗證類型) 選擇 AWS_IAM 或 NONE (無)。如需函數 URL 身分驗證的詳細資訊，請參閱存取控制。
7. (選用) 選取 Configure cross-origin resource sharing (CORS) (設定跨來源資源共享 (CORS))，然後完成函數 URL 的 CORS 設定。如需 CORS 的詳細資訊，請參閱「跨來源資源共享 (CORS)」。
8. 選擇儲存。

這樣即可為函數別名建立函數 URL。函數 URL 會顯示於主控台中別名的 Function overview (函數概觀) 區段。

使用函數 URL 建立新函數

建立具有函數 URL 的新函數 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇建立函數。
3. 在基本資訊下，請執行下列動作：
   a. 為 Function name (函數名稱) 輸入您函數的名稱，例如 **my-function**。
   b. 針對執行時間，選擇您偏好的語言執行時間，例如 Node.js 24。
   c. 為 Architecture (架構) 選擇 x86_64 或 arm64。
   d. 展開 Permissions (許可)，接著選擇建立新的執行角色或使用現有角色。
4. 展開 Advanced settings (進階設定)，然後選取 Function URL (函數 URL)。

5. 為 Auth type (驗證類型) 選擇 AWS_IAM 或 NONE (無)。如需函數 URL 身分驗證的詳細資訊，請參閱存取控制。
6. (選用) 選取 Configure cross-origin resource sharing (CORS) (設定跨來源資源共享 (CORS))。只要您在建立函數的過程中選取此選項，在預設情形下，您的函數 URL 就能允許所有來源的請求。建立函數後，您可以編輯函數 URL 的 CORS 設定。如需 CORS 的詳細資訊，請參閱「跨來源資源共享 (CORS)」。
7. 選擇建立函數。

這樣即可為 $LATEST 未發佈版本的函數建立具有函數 URL 的新函數。函數 URL 會顯示於主控台的 Function overview (函數概觀) 區段。

建立函數 URL (AWS CLI)

若要使用 AWS Command Line Interface (AWS CLI) 為現有的 Lambda 函數建立函數 URL，請執行下列命令：

```
aws lambda create-function-url-config \
    --function-name my-function \
    --qualifier prod \ // optional
    --auth-type AWS_IAM
    --cors-config {AllowOrigins="https://example.com"} // optional
```

這會將函數 URL 新增至函數 **my-function** 的 **prod** 限定詞。如需這些組態參數的詳細資訊，請參閱 API 參考資料中的 CreateFunctionUrlConfig。

> **Note**
>
> 若要透過 建立函數 URL AWS CLI，函數必須已存在。

將函數 URL 新增至 CloudFormation 範本

若要將 `AWS::Lambda::Url` 資源新增至 CloudFormation 範本，請使用下列語法：

JSON

```
{
  "Type" : "AWS::Lambda::Url",
```

```
  "Properties" : {
      "AuthType" : String,
      "Cors" : Cors,
      "Qualifier" : String,
      "TargetFunctionArn" : String
    }
}
```

YAML

```
Type: AWS::Lambda::Url
Properties:
  AuthType: String
  Cors:
    Cors
  Qualifier: String
  TargetFunctionArn: String
```

Parameters

- (必要) `AuthType` – 定義函數 URL 的身分驗證類型。可能的值為 `AWS_IAM` 或 `NONE`。如果您希望只讓完成身分驗證的使用者存取，請設為 `AWS_IAM`。如要繞過 IAM 身分驗證，並允許任何使用者向您的函數提出請求，請設為 `NONE`。
- (選用) `Cors` – 定義函數 URL 的 CORS 設定。如要將 `Cors` 新增至 CloudFormation 中的 `AWS::Lambda::Url` 資源，請使用下列語法。

  Example AWS::Lambda::Url.Cors (JSON)

  ```
  {
    "AllowCredentials" : Boolean,
    "AllowHeaders" : [ String, ... ],
    "AllowMethods" : [ String, ... ],
    "AllowOrigins" : [ String, ... ],
    "ExposeHeaders" : [ String, ... ],
    "MaxAge" : Integer
  }
  ```

  Example AWS::Lambda::Url.Cors (YAML)

  ```
    AllowCredentials: Boolean
  ```

```
  AllowHeaders:
    - String
  AllowMethods:
    - String
  AllowOrigins:
    - String
  ExposeHeaders:
    - String
  MaxAge: Integer
```

- (選用) `Qualifier` – 別名名稱。
- (必要) `TargetFunctionArn` - Lambda 函數的名稱或 Amazon Resource Name (ARN)。有效名稱的格式包括：
  - 函數名稱 – `my-function`
  - 函數 ARN – `arn:aws:lambda:us-west-2:123456789012:function:my-function`
  - 部分 ARN – `123456789012:function:my-function`

跨來源資源共享 (CORS)

如要定義不同來源存取您函數 URL 的方式，請使用跨來源資源共享 (CORS)。如果您打算從其他網域呼叫函數 URL，建議您設定 CORS。Lambda 為函數 URL 支援以下 CORS 標頭。

CORS 標頭	CORS 組態屬性	範例值

Access-Control-Allow-Origin	`AllowOrigins`	`*` (允許所有來源)
		`https://www.example.com`
		`http://localhost:60905`

Access-Control-Allow-Methods	`AllowMethods`	`GET`, `POST`, `DELETE`, `*`
Access-Control-Allow-Headers	`AllowHeaders`	`Date`, `Keep-Alive` , `X-Custom-Header`
Access-Control-Expose-Headers	`ExposeHeaders`	`Date`, `Keep-Alive` , `X-Custom-Header`

CORS 標頭	CORS 組態屬性	範例值
Access-Control-Allow-Credentials	AllowCredentials	TRUE
Access-Control-Max-Age	MaxAge	5 (預設)、300

當您使用 Lambda 主控台或 設定函數 URL 的 CORS 時 AWS CLI，Lambda 會自動透過函數 URL 將 CORS 標頭新增至所有回應。或者，您也可以手動將 CORS 標頭新增至函數回應中。如果有衝突的標頭，預期的行為取決於請求的類型：

- 對於 OPTIONS 請求等預檢請求，函數 URL 上設定的 CORS 標頭優先。Lambda 只會在回應中傳回這些 CORS 標頭。
- 對於 GET 或 POST 請求等非預檢請求，Lambda 會傳回函數 URL 上設定的 CORS 標頭，以及函數傳回的 CORS 標頭。這可能會導致回應中出現重複的 CORS 標頭。您可能會看到類似如下的錯誤：`The 'Access-Control-Allow-Origin' header contains multiple values '*, *', but only one is allowed`。

一般而言，我們建議在函數 URL 上設定所有 CORS 設定，而不是在函數回應中手動傳送 CORS 標頭。

調節函數 URL

調節作業會限制函數處理請求的速度。這在許多情況下都相當實用，例如防止函數多載下游資源，或處理突然激增的請求數。

您可以設定預留並行，調節 Lambda 函數透過函數 URL 處理請求的速度。預留並行可限制函數的並行呼叫次數上限。函數的每秒請求率 (RPS) 上限相當於所設定預留並行數的 10 倍。例如，如果您將函數的預留並行值設為 100，則 RPS 最高為 1,000。

每當函數並行數量超過預留的並行數，函數 URL 就會傳回 HTTP 429 狀態碼。如果函數收到的請求超過所設定預留並行數 10 倍的 RPS 最大值，您也會收到 HTTP 429 錯誤。如需預留並行的詳細資訊，請參閱「設定函數的預留並行」。

停用函數 URL

在緊急情況下，您可能會希望拒絕傳入函數 URL 的所有流量。若要停用函數 URL，請將預留並行設為零。這樣就能調節函數 URL 收到的所有請求，進而使 URL 傳回 HTTP 429 狀態回應。如要重新啟動函數 URL，請刪除預留並行的組態，或將組態設為大於零的數量。

刪除函數 URL

當您刪除函數 URL，您就無法復原。建立新函數 URL 會產生不同的 URL 地址。

> **Note**
>
> 如果您刪除具有驗證類型 NONE 的函數 URL，Lambda 不會自動刪除關聯的資源型政策。如果您想要刪除此政策，則必須手動執行。

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函數的名稱。
3. 選擇 Configuration (組態) 標籤，然後選擇 Function URL (函數 URL)。
4. 選擇 刪除。
5. 將 delete 一詞輸入欄位以確認刪除。
6. 選擇 刪除。

> **Note**
>
> 當您刪除具有函數 URL 的函數時，Lambda 會非同步刪除相應的函數 URL。如果您立即在相同帳戶中建立具有相同名稱的新函數，則原來的函數 URL 可能會映射到新的函數，而不會被刪除。

控制對 Lambda 函數 URL 的存取

> **Note**
>
> 自 2025 年 10 月起，新的函數 URL 將同時需要 `lambda:InvokeFunctionUrl` 和 `lambda:InvokeFunction` 許可。

您可以透過 AuthType 參數及連接至特定函數的資源型政策，控制 Lambda 函數 URL 的存取權。這兩個元件的組態能決定誰可以對函數 URL 呼叫或執行其他管理動作。

AuthType 參數決定 Lambda 如何對函數 URL 的請求執行身分驗證或授權。設定函數 URL 時，您必須指定以下任一 AuthType 選項：

- AWS_IAM – Lambda 使用 AWS Identity and Access Management (IAM) 根據 IAM 主體的身分政策和函數的資源型政策來驗證和授權請求。如果您希望僅允許已進行身分驗證的使用者和角色透過函數 URL 來調用函數，請選擇此選項。

- NONE – Lambda 不會在呼叫函數前執行任何身分驗證。然而，函數的資源型政策永遠有效，而且您必須授予公有存取權，您的函數 URL 才能接收請求。選擇此選項，即可允許使用者公開存取函數 URL，而且不必完成身分驗證。

如需安全性的其他洞見，您可以使用 AWS Identity and Access Management Access Analyzer 來取得對函數 URL 的外部存取的完整分析。IAM Access Analyzer 也能監控您 Lambda 函數新增或更新的許可，以協助您識別授予公有和跨帳戶存取權的許可。您可以免費使用 IAM Access Analyzer。若要開始使用 IAM Access Analyzer，請參閱使用 AWS IAM Access Analyzer。

本頁提供了針對兩種身分驗證類型的資源型政策範例，也說明了如何使用 AddPermission API 操作或 Lambda 主控台來建立這些政策。如需有關設定許可後如何調用函數 URL 的資訊，請參閱呼叫 Lambda 函數 URL。

主題

- 使用 AWS_IAM 驗證類型
- 使用 NONE 驗證類型
- 控管和存取權控制

使用 AWS_IAM 驗證類型

如果您選擇採用 AWS_IAM 身分驗證類型，則使用者必須擁有 lambda:InvokeFunctionUrl 和 lambda:InvokeFunction 許可才能調用 Lambda 函數 URL。視提出調用請求的使用者而定，您可能需要透過資源型政策才能授予此許可。

如果提出請求的委託人與函數 URL AWS 帳戶 位於相同位置，則委託人必須在其身分型政策中具有 lambda:InvokeFunctionUrl和 lambda:InvokeFunction 許可，或在函數的資源型政策中授予許可。換言之，若使用者已經透過資源型政策取得 lambda:InvokeFunctionUrl 和

lambda:InvokeFunction 許可，即可自行決定是否使用身分型政策。政策評估需遵循 Policy evaluation logic 中概述的規則。

如果提出請求的主體位於不同帳戶，則主體必須同時透過身分型政策取得 lambda:InvokeFunctionUrl 和 lambda:InvokeFunction 許可，並且針對其嘗試調用的函式，透過資源型政策取得許可。政策評估需遵循 Determining whether a cross-account request is allowed 中概述的規則。

下列資源型政策允許 中example AWS 帳戶 444455556666的角色叫用與函數 相關聯的函數 URLmy-function。lambda：InvokedViaFunctionUrl 內容索引鍵會將 lambda:InvokeFunction 動作限制為函數 URL 呼叫。這意味著主體必須使用函數 URL 來調用函數。如未包含 lambda:InvokedViaFunctionUrl，則主體除了函數 URL 外，還可以透過其他調用方法來調用函數。

Example– 跨帳戶資源型政策

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::444455556666:role/example"
      },
      "Action": "lambda:InvokeFunctionUrl",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
      "Condition": {
        "StringEquals": {
          "lambda:FunctionUrlAuthType": "AWS_IAM"
        }
      }
    },
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::444455556666:role/example"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-1:123456789012:function:my-function",
      "Condition": {
```

```
        "Bool": {
          "lambda:InvokedViaFunctionUrl": "true"
        }
      }
    }
  ]
}
```

您可以依照下列步驟，透過主控台建立此資源型政策：

將 URL 呼叫許可授予其他帳戶 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您要授予 URL 呼叫許可的函數名稱。
3. 依序選擇 Configuration (組態) 索引標籤和 Permissions (許可)。
4. 在 Resource-based policy (資源型政策) 底下，選擇 Add permissions (新增許可)。
5. 選擇 Function URL (函數 URL)。
6. 針對 Auth type (驗證類型) 選擇 AW_IAM。
7. 輸入政策陳述式的陳述式 ID。
8. 依據要向其授予許可的使用者或角色，在主體中輸入其帳戶 ID 或 Amazon Resource Name (ARN)。例如：**444455556666**。
9. 選擇儲存。

或者，您可以使用下列 add-permission AWS Command Line Interface (AWS CLI) 命令建立此政策。使用 時 AWS CLI，您必須分別新增 `lambda:InvokeFunctionUrl`和 `lambda:InvokeFunction`陳述式。例如：

```
aws lambda add-permission --function-name my-function \
  --statement-id UrlPolicyInvokeURL \
  --action lambda:InvokeFunctionUrl \
  --principal 444455556666 \
  --function-url-auth-type AWS_IAM
```

```
aws lambda add-permission --function-name my-function \
  --statement-id UrlPolicyInvokeFunction \
  --action lambda:InvokeFunction \
```

```
  --principal 444455556666 \
  --invoked-via-function-url
```

使用 NONE 驗證類型

> **Important**
>
> 當函數 URL 身分驗證類型為 NONE，並且透過資源型政策授予公有存取權時，任何未經身分驗證的使用者只要取得函數 URL，即可調用對應函數。

在部分情況下，您可能需要將函數 URL 設定為公開存取。例如，您可能希望為直接透過 Web 瀏覽器提出的請求提供服務。如要允許使用者公開存取您的函數 URL，請選擇 NONE 驗證類型。

如果您選擇 NONE 驗證類型，Lambda 就不會使用 IAM 對存取函數 URL 的請求執行身分驗證。但是，函式必須具有允許 `lambda:InvokeFunctionUrl` 和 `lambda:InvokeFunction` 的資源型政策。當您NONE使用主控台或 AWS Serverless Application Model (AWS SAM) 建立具有身分驗證類型的函數 URL 時，Lambda 會自動為您建立以資源為基礎的政策。如果您直接使用 AWS CloudFormation AWS CLI或 Lambda API，則必須自行新增政策。

建議在使用 NONE 身分驗證類型時，在資源型政策中包含 lambda:InvokedViaFunctionUrl 內容索引鍵。此內容索引鍵可確保函數只能透過函數 URL 調用，而不能透過其他調用方法調用。

請注意有關此政策的下列資訊：

- 所有實體都可以呼叫 `lambda:InvokeFunctionUrl` 和 `lambda:InvokeFunction`。這意味著任何擁有函數 URL 的使用者都可以調用函數。
- `lambda:FunctionUrlAuthType` 條件索引鍵值為 NONE。這意味著，唯有當函數 URL 的身分驗證類型也是 NONE 時，此政策陳述式才會允許您存取函數 URL。
- `lambda:InvokedViaFunctionUrl` 條件可確保函數只能透過函數 URL 調用，而不能透過其他調用方法調用。

Example– NONE 身分驗證類型的預設資源型政策

JSON

```
{
  "Version":"2012-10-17",
```

```
  "Statement": [
    {
      "Sid": "FunctionURLAllowPublicAccess",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "lambda:InvokeFunctionUrl",
      "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
      "Condition": {
        "StringEquals": {
          "lambda:FunctionUrlAuthType": "NONE"
        }
      }
    },
    {
      "Sid": "FunctionURLInvokeAllowPublicAccess",
      "Effect": "Allow",
      "Principal": "*",
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
      "Condition": {
        "Bool": {
          "lambda:InvokedViaFunctionUrl": "true"
        }
      }
    }
  ]
}
```

使用 建立資源型政策 AWS CLI

除非您使用 主控台或 AWS SAM 建立身分驗證類型為 的函數 URL`NONE`，否則您必須自行新增資源型政策。使用下列命令為 `lambda:InvokeFunctionUrl` 和 `lambda:InvokeFunction` 許可建立陳述式。每個陳述式都必須在個別命令中進行新增。

```
aws lambda add-permission \
  --function-name UrlTestFunction \
  --statement-id UrlPolicyInvokeURL \
  --action lambda:InvokeFunctionUrl \
  --principal * \
  --function-url-auth-type NONE
```

```
aws lambda add-permission \
```

```
  --function-name UrlTestFunction \
  --statement-id UrlPolicyInvokeFunction \
  --action lambda:InvokeFunction \
  --principal * \
  --invoked-via-function-url
```

> **Note**
>
> 如果您刪除具有驗證類型 NONE 的函數 URL，Lambda 不會自動刪除關聯的資源型政策。如果您想要刪除此政策，則必須手動執行。

如果函數資源型政策並未授予 `lambda:invokeFunctionUrl` 和 `lambda:InvokeFunction` 許可，則當使用者嘗試調用函數 URL 時，將收到 403 Forbidden 錯誤代碼。即使函數 URL 使用 NONE 身分驗證類型，也會發生這種情況。

控管和存取權控制

除了函數 URL 呼叫許可之外，您也可以控制函數 URL 設定動作的存取權。Lambda 支援以下適用於函數 URL 的 IAM 政策動作：

- `lambda:InvokeFunctionUrl` – 使用函數 URL 呼叫 Lambda 函數。
- `lambda:CreateFunctionUrlConfig` – 建立函數 URL 並設定其 AuthType。
- `lambda:UpdateFunctionUrlConfig` – 更新函數 URL 組態及其 AuthType。
- `lambda:GetFunctionUrlConfig` – 檢視函數 URL 的詳細資訊。
- `lambda:ListFunctionUrlConfigs` – 列出函數 URL 組態。
- `lambda:DeleteFunctionUrlConfig` – 刪除函數 URL。

若要允許或拒絕其他 AWS 實體的函數 URL 存取，請在 IAM 政策中包含這些動作。例如，下列政策授予 `example`角色 AWS 帳戶 更新**`my-function`**帳戶 中函數的函數 URL 的`444455556666`許可`123456789012`。

Example跨帳戶函數 URL 政策

JSON

```
{
    "Version":"2012-10-17",
```

```
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action": "lambda:UpdateFunctionUrlConfig",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:my-
function"
        }
    ]
}
```

條件索引鍵

如要精細控制函數 URL 的存取權，請使用條件內容索引鍵。Lambda 為函數 URL 支援下列內容索引鍵：

- `lambda:FunctionUrlAuthType`：定義描述函數 URL 所用身分驗證類型的列舉值。此值可以是 `AWS_IAM` 或 `NONE`。
- `lambda:InvokedViaFunctionUrl`：將 `lambda:InvokeFunction` 動作限制為透過函數 URL 進行的呼叫。這可確保該函數只能使用函數 URL 調用，而不能透過其他調用方法調用。如需使用 `lambda:InvokedViaFunctionUrl` 內容索引鍵的資源型政策範例，請參閱使用 AWS_IAM 驗證類型與使用 NONE 驗證類型中的範例。

您可以在與函式相關聯的政策中使用這些內容索引鍵。例如，您可能希望限制哪些人可以變更函數 URL 的組態。若要針對 URL 驗證類型為 `NONE` 的所有函數拒絕所有 `UpdateFunctionUrlConfig` 請求，您可以定義以下政策：

Example明確拒絕請求的函數 URL 政策

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Principal": "*",
```

```
            "Action":[
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "NONE"
                }
            }
        }
    ]
}
```

若要授予 example角色 AWS 帳戶 444455556666許可，以對 URL 驗證類型為 的函數進行 CreateFunctionUrlConfig和 UpdateFunctionUrlConfig請求AWS_IAM，您可以定義下列政策：

Example明確允許請求的函數 URL 政策

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::444455556666:role/example"
            },
            "Action":[
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:us-east-1:123456789012:function:*",
            "Condition": {
                "StringEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
```

```
}
```

您也可以在服務控制政策(SCP) 中使用此條件索引鍵。在 AWS Organizations中使用 SCP 管理整個組織的許可。例如，若要拒絕使用者建立或更新使用 AWS_IAM 以外驗證類型的函數 URL，請使用以下服務控制政策：

Example明確拒絕請求的函數 URL SCP

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Deny",
            "Action":[
                "lambda:CreateFunctionUrlConfig",
                "lambda:UpdateFunctionUrlConfig"
            ],
            "Resource": "arn:aws:lambda:*:123456789012:function:*",
            "Condition": {
                "StringNotEquals": {
                    "lambda:FunctionUrlAuthType": "AWS_IAM"
                }
            }
        }
    ]
}
```

呼叫 Lambda 函數 URL

函數 URL 是 Lambda 函數專用的 HTTP(S) 端點。您可以透過 Lambda 主控台或 Lambda API 建立及設定函數 URL。

> **Tip**
>
> Lambda 提供兩種透過 HTTP 端點調用函式的方式：函式 URL 與 Amazon API Gateway。如果不確定哪種方法最適合自己的使用案例，請參閱the section called “函式 URL 與 Amazon API Gateway 比較”。

當您建立函數 URL 時，Lambda 會自動為您產生不重複的 URL 端點。函數 URL 一旦建立，其 URL 端點便永遠不會變更。函數 URL 端點的格式如下：

```
https://<url-id>.lambda-url.<region>.on.aws
```

> **Note**
>
> 下列 AWS 區域不支援函式 URL：亞太區域 (海德拉巴) (ap-south-2)、亞太區域 (墨爾本) (ap-southeast-4)、亞太區域 (馬來西亞) (ap-southeast-5)、亞太區域 (紐西蘭) (ap-southeast-6)、亞太區域 (泰國) (ap-southeast-7)、亞太區域 (台北) (ap-east-2)、加拿大西部 (卡加利)(ca-west-1)、歐洲 (西班牙) (eu-south-2)、歐洲 (蘇黎世) (eu-central-2)、以色列 (特拉維夫) (il-central-1) 以及中東 (阿拉伯聯合大公國) (me-central-1)。

函數 URL 可支援雙堆疊，能同時支援 IPv4 和 IPv6。設定函數 URL 後，您可以利用 Web 瀏覽器、curl、Postman 或任何 HTTP 用戶端，透過 HTTP(S) 端點呼叫函數。若要調用函式 URL，您必須擁有 `lambda:InvokeFunctionUrl` 和 `lambda:InvokeFunction` 許可。如需更多詳細資訊，請參閱 存取控制。

主題

- 函數 URL 呼叫基礎知識
- 請求和回應承載

函數 URL 呼叫基礎知識

如果您的函數 URL 使用 AWS_IAM 驗證類型，您必須使用 AWS Signature 第 4 版 (SigV4) 簽署各個 HTTP 請求。Postman 等工具提供使用 SigV4 簽署請求的內建功能。

如果您不使用工具簽署函數 URL 的 HTTP 請求，則必須使用 SigV4 手動簽署各個請求。您的函數 URL 收到請求時，Lambda 也會計算 SigV4 簽章。唯有簽章相符，Lambda 才會處理請求。如需使用 SigV4 手動簽署請求的相關說明，請參閱《Amazon Web Services 一般參考 指南》中的使用 Signature 第 4 版簽署 AWS 請求。

如果您的函數 URL 使用 NONE 驗證類型，您不必使用 SigV4 簽署請求。您可以使用 Web 瀏覽器、curl、Postman 或任何 HTTP 用戶端來呼叫您的函數。

如要測試函數的簡易 GET 請求，請使用 Web 瀏覽器。例如，如果函數 URL 為 `https://abcdefg.lambda-url.us-east-1.on.aws`，而且接受字串參數 `message`，則請求 URL 可能如下所示：

```
https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld
```

如要測試其他 HTTP 請求 (例如 POST 請求)，您可以使用 curl 等工具。例如，如果您希望在函數 URL 的 POST 請求中納入某些 JSON 資料，您可以使用下列 curl 命令：

```
curl -v 'https://abcdefg.lambda-url.us-east-1.on.aws/?message=HelloWorld' \
-H 'content-type: application/json' \
-d '{ "example": "test" }'
```

請求和回應承載

用戶端呼叫您的函數 URL 時，Lambda 會先將請求映射至事件物件，再將請求傳遞給函數。接著，函數的回應會映射至 Lambda 透過函數 URL 回傳給用戶端的 HTTP 回應。

請求和回應事件格式會採取與 Amazon API Gateway 承載格式 2.0 版一樣的結構描述。

請求承載格式

請求承載的結構如下：

```
{
  "version": "2.0",
  "routeKey": "$default",
  "rawPath": "/my/path",
  "rawQueryString": "parameter1=value1&parameter1=value2&parameter2=value",
```

```
  "cookies": [
    "cookie1",
    "cookie2"
  ],
  "headers": {
    "header1": "value1",
    "header2": "value1,value2"
  },
  "queryStringParameters": {
    "parameter1": "value1,value2",
    "parameter2": "value"
  },
  "requestContext": {
    "accountId": "123456789012",
    "apiId": "<urlid>",
    "authentication": null,
    "authorizer": {
        "iam": {
                "accessKey": "AKIA...",
                "accountId": "111122223333",
                "callerId": "AIDA...",
                "cognitoIdentity": null,
                "principalOrgId": null,
                "userArn": "arn:aws:iam::111122223333:user/example-user",
                "userId": "AIDA..."
        }
    },
    "domainName": "<url-id>.lambda-url.us-west-2.on.aws",
    "domainPrefix": "<url-id>",
    "http": {
      "method": "POST",
      "path": "/my/path",
      "protocol": "HTTP/1.1",
      "sourceIp": "123.123.123.123",
      "userAgent": "agent"
    },
    "requestId": "id",
    "routeKey": "$default",
    "stage": "$default",
    "time": "12/Mar/2020:19:03:58 +0000",
    "timeEpoch": 1583348638390
  },
  "body": "Hello from client!",
  "pathParameters": null,
```

```
  "isBase64Encoded": false,
  "stageVariables": null
}
```

參數	描述	範例
version	此事件的承載格式版本。Lambda 函數 URL 目前支援承載格式 2.0 版。	2.0
routeKey	函數 URL 不使用此參數。Lambda 將此設為 $default 作為預留位置使用。	$default
rawPath	請求路徑。例如，如果請求 URL 為 https://{url-id}.lambda-url.{region}.on.aws/example/test/demo ，則原始路徑值為 /example/test/demo 。	/example/test/demo
rawQueryString	內含請求查詢字串參數的原始字串。支援的字元包含 a-z、A-Z、0-9、.、_、-、%、&、=，以及 +。	"?parameter1=value1&parameter2=value2"
cookies	內含隨請求一併傳送之所有 Cookie 的陣列。	["Cookie_1=Value_1", "Cookie_2=Value_2"]
headers	以鍵值對形式呈現的請求標頭清單。	{"header1": "value1", "header2": "value2"}
queryStringParameters	請求的查詢參數。例如，如果請求 URL 為 https://{url-id}.lambda-url.{region}.on.aws/e	{"name": "Jane"}

參數	描述	範例
	xample?name=Jane ，則 queryStringParameters 值為具備索引鍵 name 和值 Jane 的 JSON 物件。	
requestContext	內含請求其他資訊的物件，例如 requestId 、請求的時間，以及呼叫者的身分 (如果透過 AWS Identity and Access Management (IAM) 取得授權的話)。	
requestContext.accountId	函數擁有者的 AWS 帳戶 ID。	"123456789012"
requestContext.apiId	函數 URL 的 ID。	"33anwqw8fj"
requestContext.authentication	函數 URL 不使用此參數。Lambda 將此設為 null。	null
requestContext.authorizer	內含呼叫者身分相關資訊的物件 (如果函數 URL 使用 AWS_IAM 驗證類型的話)，否則 Lambda 會將此設為 null。	
requestContext.authorizer.iam.accessKey	呼叫者身分的存取金鑰。	"AKIAIOSFODNN7EXAMPLE"
requestContext.authorizer.iam.accountId	呼叫者身分的 AWS 帳戶 ID。	"111122223333"
requestContext.authorizer.iam.callerId	呼叫者的 ID (使用者 ID)。	"AIDACKCEVSQ6C2EXAMPLE"

參數	描述	範例
requestContext.authorizer.iam.cognitoIdentity	函數 URL 不使用此參數。Lambda 將此設為 null，或將此從 JSON 排除。	null
requestContext.authorizer.iam.principalOrgId	與呼叫者身分相關聯的委託人組織 ID。	"AIDACKCEVSQORGEXAMPLE"
requestContext.authorizer.iam.userArn	呼叫者身分的使用者 Amazon Resource Name (ARN)。	"arn:aws:iam::111122223333:user/example-user"
requestContext.authorizer.iam.userId	呼叫者身分的使用者 ID。	"AIDACOSFODNN7EXAMPLE2"
requestContext.domainName	函數 URL 的網域名稱。	"<url-id>.lambda-url.us-west-2.on.aws"
requestContext.domainPrefix	函數 URL 的網域前綴。	"<url-id>"
requestContext.http	內含 HTTP 請求詳細資訊的物件。	
requestContext.http.method	此請求所採用的 HTTP 方法。有效值包括 GET、POST、PUT、HEAD、OPT 和 DELETE。	GET

參數	描述	範例
requestContext.http.path	請求路徑。例如，如果請求 URL 為 https://{url-id}.lambda-url.{region}.on.aws/example/test/demo ，則路徑值為 /example/test/demo 。	/example/test/demo
requestContext.http.protocol	請求的通訊協定。	HTTP/1.1
requestContext.http.sourceIp	提出請求之即時 TCP 連線的來源 IP 地址。	123.123.123.123
requestContext.http.userAgent	使用者代理程式請求標頭的值。	Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) Gecko/20100101 Firefox/42.0
requestContext.requestId	呼叫請求的 ID。您可以使用此 ID 追蹤與函數相關的呼叫日誌。	e1506fd5-9e7b-434f-bd42-4f8fa224b599
requestContext.routeKey	函數 URL 不使用此參數。Lambda 將此設為 $default 作為預留位置使用。	$default
requestContext.stage	函數 URL 不使用此參數。Lambda 將此設為 $default 作為預留位置使用。	$default
requestContext.time	請求的時間戳記。	"07/Sep/2021:22:50:22 +0000"
requestContext.timeEpoch	Unix epoch 時間格式的請求時間戳記。	"1631055022677"

參數	描述	範例
body	請求的本文。如果請求的內容屬於二進位類型，則本文會採用 base64 編碼。	{"key1": "value1", "key2": "value2"}
pathParameters	函數 URL 不使用此參數。Lambda 將此設為 null，或將此從 JSON 排除。	null
isBase64Encoded	如果本文為二進位承載並採用 base64 編碼，此值為 TRUE，否則為 FALSE。	FALSE
stageVariables	函數 URL 不使用此參數。Lambda 將此設為 null，或將此從 JSON 排除。	null

回應承載格式

函數傳回回應時，Lambda 會剖析回應內容，並將其轉換為 HTTP 回應。函數回應承載的格式如下：

```
{
   "statusCode": 201,
    "headers": {
        "Content-Type": "application/json",
        "My-Custom-Header": "Custom Value"
    },
    "body": "{ \"message\": \"Hello, world!\" }",
    "cookies": [
        "Cookie_1=Value1; Expires=21 Oct 2021 07:48 GMT",
        "Cookie_2=Value2; Max-Age=78000"
    ],
    "isBase64Encoded": false
}
```

Lambda 會為您推斷回應格式。如果您的函數傳回有效的 JSON，但未傳回 statusCode，則 Lambda 會採取下列假設：

- statusCodeis(200)

> **Note**
>
> 有效 statusCode 的範圍為 100 至 599。

- content-typeis(application/json)
- body 是函數的回應。
- isBase64Encodedis(false)

以下範例顯示 Lambda 函數輸出與回應承載之間的映射情形，以及回應承載與最終 HTTP 回應之間的映射情形。用戶端呼叫您的函數 URL 時，會看見 HTTP 回應。

字串回應的輸出範例

Lambda 函數輸出	轉譯後的回應輸出	HTTP 回應 (用戶端看見的內容)
`"Hello, world!"`	`{ "statusCode": 200, "body": "Hello, world!", "headers": { "content-type": "application/json" }, "isBase64Encoded": false }`	`HTTP/2 200 date: Wed, 08 Sep 2021 18:02:24 GMT content-type: application/json content-length: 15 "Hello, world!"`

JSON 回應的輸出範例

Lambda 函數輸出	轉譯後的回應輸出	HTTP 回應 (用戶端看見的內容)
`{ "message": "Hello, world!"`	`{ "statusCode": 200, "body": {`	`HTTP/2 200 date: Wed, 08 Sep 2021 18:02:24 GMT`

Lambda 函數輸出	轉譯後的回應輸出	HTTP 回應 (用戶端看見的內容)

}
 	
	"message": "Hello, world!"
	},
	"headers": {
	"content-type": "application/json"
	},
	"isBase64Encoded": false
	}
		content-type: application/json
		content-length: 34
		
		{
		"message": "Hello, world!"
		}

自訂回應的輸出範例

Lambda 函數輸出	轉譯後的回應輸出	HTTP 回應 (用戶端看見的內容)

{
"statusCode": 201,
"headers": {
"Content-Type": "application/json",
"My-Custom-Header": "Custom Value"
},
"body": JSON.stringify({
"message": "Hello, world!"
}),
"isBase64Encoded": false
}
	{
	"statusCode": 201,
	"headers": {
	"Content-Type": "application/json",
	"My-Custom-Header": "Custom Value"
	},
	"body": JSON.stringify({
	"message": "Hello, world!"
	}),
	"isBase64Encoded": false
	}
		HTTP/2 201
		date: Wed, 08 Sep 2021 18:02:24 GMT
		content-type: application/json
		content-length: 27
		my-custom-header: Custom Value
		
		{
		"message": "Hello, world!"
		}

Cookie

如要從函數傳回 Cookie，請勿手動新增 `set-cookie` 標頭。您應將 Cookie 加入回應承載物件中。Lambda 會自動轉譯 Cookie，並以 `set-cookie` 標頭形式新增至 HTTP 回應，如下所示。

Lambda 函數輸出	HTTP 回應 (用戶端看見的內容)

<pre>{
   "statusCode": 201,
    "headers": {
        "Content-Type": "application/
json",
        "My-Custom-Header": "Custom
 Value"
    },
    "body": JSON.stringify({
        "message": "Hello, world!"
    }),
    "cookies": [
        "Cookie_1=Value1; Expires=21
 Oct 2021 07:48 GMT",
        "Cookie_2=Value2; Max-Age=7
8000"
    ],
    "isBase64Encoded": false
}</pre>
	<pre>HTTP/2 201
	date: Wed, 08 Sep 2021 18:02:24 GMT
	content-type: application/json
	content-length: 27
	my-custom-header: Custom Value
	set-cookie: Cookie_1=Value2;
	 Expires=21 Oct 2021 07:48 GMT
	set-cookie: Cookie_2=Value2; Max-
	Age=78000
	
	{
	  "message": "Hello, world!"
	}</pre>

監控 Lambda 函數 URL

您可以使用 AWS CloudTrail 和 Amazon CloudWatch 監控函數 URL。

主題

- 使用 CloudTrail 監控函數 URL
- 函數 URL 的 CloudWatch 指標

使用 CloudTrail 監控函數 URL

針對函數 URL，Lambda 能將下列 API 操作記錄為 CloudTrail 日誌檔案事件：

- CreateFunctionUrlConfig
- UpdateFunctionUrlConfig
- DeleteFunctionUrlConfig
- GetFunctionUrlConfig
- ListFunctionUrlConfigs

每個日誌項目都包含呼叫者身分、提出請求的時間，以及其他詳細資訊等相關資訊。您可以檢視 CloudTrail 事件歷史記錄，查看過去 90 天內的所有事件。如要保留 90 天前的記錄，您可以建立線索。

預設情況下，CloudTrail 不會記錄 `InvokeFunctionUrl` 請求，系統會將這些請求視為資料事件。不過，您可以在 CloudTrail 中開啟資料事件記錄功能。如需詳細資訊，請參閱《AWS CloudTrail使用者指南》中的記錄資料事件。

函數 URL 的 CloudWatch 指標

Lambda 會將函數 URL 請求的彙總指標傳送到 CloudWatch。透過這些指標，您可以在 CloudWatch 主控台中監控函數 URL、建置儀錶板及設定警示。

函數 URL 支援以下呼叫指標。建議您搭配 `Sum` 統計數字一併檢視這些指標。

- `UrlRequestCount` – 對此函數提出的請求數量。
- `Url4xxCount` – 傳回 4XX HTTP 狀態碼的請求數量。收到 4XX 系列代碼表示用戶端發生錯誤，例如請求錯誤。

- `Url5xxCount` – 傳回 5XX HTTP 狀態碼的請求數量。收到 5XX 系列代碼表示伺服器端發生錯誤，例如函數錯誤和逾時。

函數 URL 也支援以下效能指標。建議您搭配 `Average` 或 `Max` 統計數字一併檢視這些指標。

- `UrlRequestLatency` – 函數 URL 從收到請求到傳回回應所經過的時間。

這些呼叫和效能指標均支援以下維度：

- `FunctionName` – 針對指派給函數`$LATEST`未發佈版本或任何函數別名的函數 URL，查看其彙總指標，例如 `hello-world-function`。
- `Resource` – 檢視特定函數 URL 的指標。您可使用函數名稱，並搭配函數未發佈的`$LATEST`版本或任一函數別名來加以定義，例如 `hello-world-function:$LATEST`。
- `ExecutedVersion` – 根據所執行的版本檢視特定函數 URL 的指標。此維度的主要功能是追蹤指派給`$LATEST`未發佈版本的函數 URL。

選取一種使用 HTTP 請求調用 Lambda 函數的方法

Lambda 的許多常見使用案例涉及使用 HTTP 請求調用您的函數。例如，您可能希望 Web 應用程式透過瀏覽器請求調用您的函式。Lambda 函式也可用於建立完整的 REST API、處理來自行動應用程式的使用者互動、透過 HTTP 呼叫處理來自外部服務的資料，或建立自訂 Webhook。

以下各節說明透過 HTTP 調用 Lambda 的選擇，並提供資訊以協助您針對特定使用案例做出正確決策。

選取 HTTP 調用方法時，您有哪些選擇？

Lambda 提供兩種主要方法來使用 HTTP 請求調用函數 - 函數 URL 和 API Gateway。這兩種選項的主要差異如下所示：

- Lambda 函數 URL 可為 Lambda 函數提供簡單、直接的 HTTP 端點。已針對簡單性和成本效益對其進行最佳化，並提供透過 HTTP 公開 Lambda 函數的最快路徑。
- API Gateway 是一種更進階的服務，用於建置功能完整的 API。API Gateway 已針對大規模建置和管理生產 API 進行最佳化，並提供完整的安全、監控和流量管理工具。

您已知道自己需求時的建議

如果您已經清楚自己的需求，以下是基本建議：

建議函數 URL 用於簡單的應用程式或原型設計，其中您只需要基本的身分驗證方法和請求/回應處理，並想要將成本和複雜性降至最低。

API Gateway 是大規模生產應用程式或需要更進階功能的情況的更佳選擇，例如 OpenAPI Description 支援、身分驗證選項、自訂網域名稱或豐富的請求/回應處理，包括限流、快取和請求/回應轉換。

選擇調用 Lambda 函數的方法時應考慮的事項

在函數 URL 和 API Gateway 之間選取時，需要考慮下列因素：

- 您的身分驗證需要，例如是否需要 OAuth 或 Amazon Cognito 來驗證使用者
- 您的擴展需求以及您要實作之 API 的複雜性
- 您是否需要進階功能，例如請求驗證和請求/回應格式化
- 您的監控需求
- 您的成本目標

透過了解這些因素，可以選擇最能平衡您的安全性、複雜性和成本需求的選項。

下列資訊摘要說明兩個選項之間的主要差異。

身分驗證

- 函數 URLs提供基本身分驗證選項。 AWS Identity and Access Management 可以將端點設定為公有 (無身分驗證) 或需要 IAM 身分驗證。透過 IAM 身分驗證，您可以使用標準 AWS 登入資料或 IAM 角色來控制存取。雖然設定簡單，但相較於其他驗證方法，此方法提供的選項有限。
- API Gateway 可存取更全面的身分驗證選項。除了 IAM 身分驗證之外，也可以使用 Lambda 授權工具 (自訂身分驗證邏輯)、Amazon Cognito 使用者集區和 OAuth2.0 流程。此彈性可讓您實作複雜的身分驗證機制，包括第三方身分驗證提供者、權杖型身分驗證和多重要素身分驗證。

請求/回應處理

- 函數 URL 提供基本的 HTTP 請求和回應處理。它們支援標準 HTTP 方法，並包含內建的跨來源資源共用 (CORS) 支援。雖然他們可以自然處理 JSON 承載和查詢參數，但它們不提供請求轉換或驗證功能。回應處理同樣簡單 – 用戶端接收來自 Lambda 函數的回應，就像 Lambda 傳回它一樣。

- API Gateway 可提供複雜的請求和回應處理功能。您可以定義請求驗證器，使用映射範本轉換請求和回應，設定請求/回應標頭，以及實作回應快取。API Gateway 也支援二進位承載和自訂網域名稱，並且可以在回應到達用戶端之前對其進行修改。可以使用 JSON 結構描述來設定請求/回應驗證和轉換的模型。

擴展

- 函數 URL 會根據 Lambda 函數的並行限制直接擴展，並透過將函數擴展到其設定的並行限制上限來處理流量尖峰。達到該限制後，Lambda 會使用 HTTP 429 回應來回應其他請求。沒有內建佇列機制，因此處理擴展完全取決於 Lambda 函數的組態。根據預設，Lambda 函數每個 有 1，000 個並行執行的限制 AWS 區域。
- 除了 Lambda 自己的擴展之外，API Gateway 還提供其他擴展功能。它包含內建的請求佇列和限流控制，可讓您更輕鬆地管理流量尖峰。根據預設，API Gateway 每秒最多可以處理 10,000 個請求，高載容量為每秒 5,000 個請求。它也提供在不同層級調節請求的工具 (API、階段或方法)，以保護後端。

監控

- 函數 URL 透過 Amazon CloudWatch 指標提供基本監控，包括請求計數、延遲和錯誤率。可以存取標準 Lambda 指標和日誌，它們會顯示傳入函數的原始請求。雖然這可提供基本的操作可見性，但指標主要著重於函數執行。
- API Gateway 提供全面的監控功能，包括詳細的指標、記錄和追蹤選項。可以透過 CloudWatch 來監控 API 呼叫、延遲、錯誤率和快取命中率/遺失率。API Gateway 也會整合 與 AWS X-Ray 以進行分散式追蹤，並提供可自訂的記錄格式。

Cost

- 函數 URL 遵循標準 Lambda 定價模型 – 您只需支付函數調用和運算時間的費用。URL 端點本身不收取額外費用。如果您不需要 API Gateway 的其他功能，這對於簡單的 API 或低流量應用程式而言是一個具成本效益的選擇。
- API Gateway 提供免費方案，其中包含針對 REST API 收到的一百萬個 API 呼叫，以及針對 HTTP API 收到的一百萬個 API 呼叫。之後，API Gateway 會針對 API 呼叫、資料傳輸和快取 (如果啟用) 收取費用。請參閱 API Gateway 定價頁面，了解您自己的使用案例費用。

其他功能

- 函數 URL 旨在實現簡便性和直接 Lambda 整合。它們支援 HTTP 和 HTTPS 端點，提供內建 CORS 支援，並提供雙堆疊 (IPv4 和 IPv6) 端點。雖然它們缺乏進階功能，但它們在您需要快速、直接地透過 HTTP 公開 Lambda 函數時表現卓越。
- API Gateway 包含許多其他功能，例如 API 版本控制、階段管理、用於用量計劃的 API 金鑰、透過 Swagger/OpenAPI 的 API 文件、WebSocket API、VPC 中的私有 API 以及 WAF 整合，以提供額外的安全性。它還支援 Canary 部署、用於測試的模擬整合，以及與 Lambda AWS 服務 以外的其他整合。

選取調用 Lambda 函數的方法

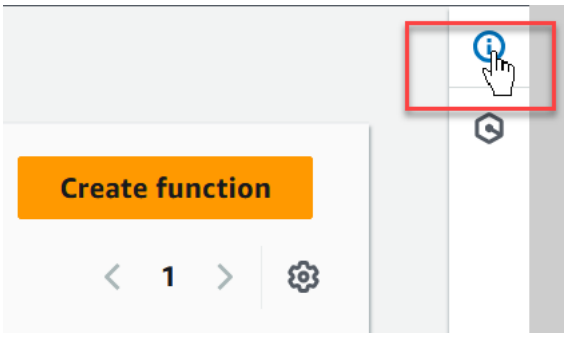
現在您已了解在 Lambda 函數 URL 和 API Gateway 之間進行選擇的條件，以及它們之間的主要差異，您可以選擇最符合您需求的選項，並使用下列資源來協助您開始使用。

Function URLs

使用下列資源開始使用函數 URL

- 遵循教學課程建立具有函數 URL 的 Lambda 函數
- 在本指南的 the section called “函數 URL” 章節中進一步了解 函數 URL
- 透過執行以下操作，嘗試主控台內的引導式教學課程建立簡單的 Web 應用程式：

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇畫面右上角的圖示以開啟說明面板。



3. 選取教學課程。
4. 在建立簡單的 Web 應用程式中，選擇開始教學課程。

API Gateway

使用下列資源開始使用 Lambda 和 API Gateway

- 依照教學課程搭配使用 Lambda 與 API Gateway 來建立與後端 Lambda 函數整合的 REST API。
- 在 Amazon API Gateway 開發人員指南的下列章節中，進一步了解 API Gateway 提供的不同類型 API：
  - API Gateway REST API
  - API Gateway HTTP API
  - API Gateway WebSocket API
- 請嘗試 Amazon API Gateway 開發人員指南的教學課程和研討會一節中的一個或多個範例。

教學課程：使用 Lambda 函式 URL 建立 Webhook 端點

在本教學課程中，您將建立 Lambda 函式 URL 來實作 Webhook 端點。Webhook 是一種輕量型的事件驅動型通訊機制，能透過 HTTP 自動在應用程式間傳輸資料。您可以使用 Webhook 來接收有關其他系統中發生事件的即時更新，例如當新客戶在網站上註冊、處理付款或上傳檔案時。

使用 Lambda 時，可以透過 Lambda 函式 URL 或 API Gateway 來實作 Webhook。對於不需要進階授權或請求驗證等功能的簡單 Webhook，函式 URL 是很好的選擇。

> **Tip**
>
> 如果您不確定哪種解決方案最適合特定使用案例，請參閱the section called “函式 URL 與 Amazon API Gateway 比較”。

先決條件

要完成本教學課程，必須在本機電腦上安裝 Python (3.8 版或更新版本) 或 Node.js (18 版或更新版本)。

為了使用 HTTP 請求測試端點，本教學課程使用了 curl。這是一個命令列工具，能透過多種網路通訊協定來傳輸資料。若尚未安裝此工具，請參閱 curl 文件了解安裝方法。

建立 Lambda 函式

首先建立當 HTTP 請求傳送至 Webhook 端點時執行的 Lambda 函式。在此範例中，傳送端應用程式會在付款提交時傳送更新，並在 HTTP 請求內文中指出付款是否成功。Lambda 函式會剖析請求，再依據付款狀態採取動作。在此範例中，程式碼僅會列印付款的訂單 ID，但在實際應用程式中，您可以將訂單新增至資料庫或傳送通知。

函式還實作了 Webhook 最常用的身分驗證方法，即雜湊訊息驗證碼 (HMAC)。使用此方法時，傳送端與接收端應用程式皆會共用一個秘密金鑰。傳送端應用程式使用雜湊演算法，利用此金鑰和訊息內容產生唯一簽章，並將該簽章作為 HTTP 標頭包含在 Webhook 請求中。然後，接收端應用程式會重複此步驟，使用秘密金鑰產生簽章，並將結果值與請求標頭中傳送的簽章進行比較。如果結果相符，請求會視為合法。

使用 Lambda 主控台並選擇 Python 或 Node.js 執行時期來建立函式。

Python

建立 Lambda 函式

1. 開啟 Lambda 主控台中的函數頁面。
2. 執行下列動作，建立基本的 'Hello world' 函式：
   a. 選擇 Create function (建立函數)。
   b. 選取從頭開始撰寫。
   c. 針對函數名稱，請輸入 **myLambdaWebhook**。
   d. 針對執行期，選取 python3.14。
   e. 選擇 Create function (建立函數)。
3. 在程式碼來源窗格中，複製並貼上以下項目來取代現有程式碼：

```
import json
import hmac
import hashlib
import os

def lambda_handler(event, context):

    # Get the webhook secret from environment variables
    webhook_secret = os.environ['WEBHOOK_SECRET']

    # Verify the webhook signature
    if not verify_signature(event, webhook_secret):
        return {
            'statusCode': 401,
            'body': json.dumps({'error': 'Invalid signature'})
        }

    try:
        # Parse the webhook payload
        payload = json.loads(event['body'])

        # Handle different event types
        event_type = payload.get('type')

        if event_type == 'payment.success':
            # Handle successful payment
            order_id = payload.get('orderId')
```

```
            print(f"Processing successful payment for order {order_id}")

            # Add your business logic here
            # For example, update database, send notifications, etc.

        elif event_type == 'payment.failed':
            # Handle failed payment
            order_id = payload.get('orderId')
            print(f"Processing failed payment for order {order_id}")

            # Add your business logic here

        else:
            print(f"Received unhandled event type: {event_type}")

        # Return success response
        return {
            'statusCode': 200,
            'body': json.dumps({'received': True})
        }

    except json.JSONDecodeError:
        return {
            'statusCode': 400,
            'body': json.dumps({'error': 'Invalid JSON payload'})
        }
    except Exception as e:
        print(f"Error processing webhook: {e}")
        return {
            'statusCode': 500,
            'body': json.dumps({'error': 'Internal server error'})
        }

def verify_signature(event, webhook_secret):
    """
    Verify the webhook signature using HMAC
    """
    try:
        # Get the signature from headers
        signature = event['headers'].get('x-webhook-signature')

        if not signature:
            print("Error: Missing webhook signature in headers")
            return False
```

```
        # Get the raw body (return an empty string if the body key doesn't
exist)
        body = event.get('body', '')

        # Create HMAC using the secret key
        expected_signature = hmac.new(
            webhook_secret.encode('utf-8'),
            body.encode('utf-8'),
            hashlib.sha256
        ).hexdigest()

        # Compare the expected signature with the received signature to
authenticate the message
        is_valid = hmac.compare_digest(signature, expected_signature)
        if not is_valid:
            print(f"Error: Invalid signature. Received: {signature}, Expected:
{expected_signature}")
            return False

        return True
    except Exception as e:
        print(f"Error verifying signature: {e}")
        return False
```

4. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼。

Node.js

建立 Lambda 函式

1. 開啟 Lambda 主控台中的函數頁面。
2. 執行下列動作，建立基本的 'Hello world' 函式：
   a. 選擇 Create function (建立函數)。
   b. 選取從頭開始撰寫。
   c. 針對函數名稱，請輸入 **myLambdaWebhook**。
   d. 針對執行期，選取 nodejs24.x。
   e. 選擇 Create function (建立函數)。
3. 在程式碼來源窗格中，複製並貼上以下項目來取代現有程式碼：

```
import crypto from 'crypto';

export const handler = async (event, context) => {
    // Get the webhook secret from environment variables
    const webhookSecret = process.env.WEBHOOK_SECRET;

    // Verify the webhook signature
    if (!verifySignature(event, webhookSecret)) {
        return {
            statusCode: 401,
            body: JSON.stringify({ error: 'Invalid signature' })
        };
    }

    try {
        // Parse the webhook payload
        const payload = JSON.parse(event.body);

        // Handle different event types
        const eventType = payload.type;

        switch (eventType) {
            case 'payment.success': {
                // Handle successful payment
                const orderId = payload.orderId;
                console.log(`Processing successful payment for order
 ${orderId}`);

                // Add your business logic here
                // For example, update database, send notifications, etc.
                break;
            }

            case 'payment.failed': {
                // Handle failed payment
                const orderId = payload.orderId;
                console.log(`Processing failed payment for order ${orderId}`);

                // Add your business logic here
                break;
            }

            default:
```

```
                console.log(`Received unhandled event type: ${eventType}`);
        }

        // Return success response
        return {
            statusCode: 200,
            body: JSON.stringify({ received: true })
        };

    } catch (error) {
        if (error instanceof SyntaxError) {
            // Handle JSON parsing errors
            return {
                statusCode: 400,
                body: JSON.stringify({ error: 'Invalid JSON payload' })
            };
        }

        // Handle all other errors
        console.error('Error processing webhook:', error);
        return {
            statusCode: 500,
            body: JSON.stringify({ error: 'Internal server error' })
        };
    }
};

// Verify the webhook signature using HMAC

const verifySignature = (event, webhookSecret) => {
    try {
        // Get the signature from headers
        const signature = event.headers['x-webhook-signature'];

        if (!signature) {
            console.log('No signature found in headers:', event.headers);
            return false;
        }

        // Get the raw body (return an empty string if the body key doesn't
 exist)
        const body = event.body || '';

        // Create HMAC using the secret key
```

```
        const hmac = crypto.createHmac('sha256', webhookSecret);
        const expectedSignature = hmac.update(body).digest('hex');

        // Compare expected and received signatures
        const isValid = signature === expectedSignature;
        if (!isValid) {
            console.log(`Invalid signature. Received: ${signature}, Expected:
${expectedSignature}`);
            return false;
        }

        return true;
    } catch (error) {
        console.error('Error during signature verification:', error);
        return false;
    }
 };
```

4. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼。

建立秘密金鑰

Lambda 函式若要驗證 Webhook 請求，會使用一個與呼叫應用程式共用的秘密金鑰。在此範例中，金鑰存放在環境變數中。但在生產應用程式中，請勿在函數程式碼中包含密碼等敏感資訊。反之，請建立 AWS Secrets Manager 秘密，然後使用 AWS 參數和秘密 Lambda 延伸來擷取 Lambda 函數中的登入資料。

建立並儲存 Webhook 秘密金鑰

1. 使用密碼編譯安全的隨機數字產生器，產生一個長的隨機字串。您可以使用下列以 Python 或 Node.js 編寫的程式碼片段來生成並列印 32 字元的秘密，也可以使用自己慣用的方法。

   Python

   Example產生秘密的程式碼

   ```
   import secrets
   webhook_secret = secrets.token_urlsafe(32)
   print(webhook_secret)
   ```

Node.js

Example產生秘密的程式碼 (ES 模組格式)

```
import crypto from 'crypto';
let webhookSecret = crypto.randomBytes(32).toString('base64');
console.log(webhookSecret)
```

2. 執行下列動作，將產生的字串儲存為函式的環境變數：

   a. 在函式的組態索引標籤中，選取環境變數。

   b. 選擇編輯。

   c. 選擇 Add environment variable (新增環境變數)。

   d. 在金鑰欄位中，輸入 **WEBHOOK_SECRET**；接著在值欄位中，輸入上一個步驟中產生的秘密。

   e. 選擇儲存。

本教學課程後續將再次用到此金鑰來測試函式，因此請立即記錄下來。

建立函式 URL 端點

使用 Lambda 函式 URL 為 Webhook 建立端點。由於您使用身分驗證類型 NONE 來建立具有公有存取權的端點，因此任何擁有 URL 的使用者都可以調用函式。若要進一步了解如何控制函式 URL 的存取權，請參閱the section called “存取控制”。如果 Webhook 需要更進階的身分驗證選項，建議採用 API Gateway。

建立函式 URL 端點

1. 在函式的組態索引標籤中，選取函式 URL。
2. 選擇 Create function URL (建立函數 URL)。
3. 在身分驗證類型欄位中，選取 NONE。
4. 選擇儲存。

剛剛建立的函式 URL 端點會顯示在函式 URL 窗格中。複製端點，在教學課程後續部分使用。

在主控台中測試函式

在透過 URL 端點使用 HTTP 請求調用函式之前，請先在主控台中測試，確認程式碼如預期運作。

若要在主控台中驗證函式，需先使用之前在教學課程中產生的秘密，搭配以下測試用 JSON 承載資料計算 Webhook 簽章：

```
{
    "type": "payment.success",
    "orderId": "1234",
    "amount": "99.99"
}
```

利用下列其中一個以 Python 或 Node.js 編寫的程式碼範例，並使用您自己的秘密來計算 Webhook 簽章。

Python

計算 Webhook 簽章

1. 將下列程式碼儲存為名為 `calculate_signature.py` 的檔案。將程式碼中的 Webhook 秘密取代為您自己的值。

```
import secrets
import hmac
import json
import hashlib

webhook_secret = "arlbSDCP86n_1H90s0fL_Qb2NAHBIBQOyGI0X4Zay4M"

body = json.dumps({"type": "payment.success", "orderId": "1234", "amount":
 "99.99"})

signature = hmac.new(
            webhook_secret.encode('utf-8'),
            body.encode('utf-8'),
            hashlib.sha256
        ).hexdigest()

print(signature)
```

2. 從儲存程式碼的同一目錄中執行下列命令來計算簽章。複製程式碼輸出的簽章。

```
python calculate_signature.py
```

Node.js

計算 Webhook 簽章

1. 將下列程式碼儲存為名為 `calculate_signature.mjs` 的檔案。將程式碼中的 Webhook 秘密取代為您自己的值。

```
import crypto from 'crypto';

const webhookSecret = "arlbSDCP86n_1H90s0fL_Qb2NAHBIBQOyGI0X4Zay4M"
const body = "{\"type\": \"payment.success\", \"orderId\": \"1234\", \"amount\":
 \"99.99\"}";

let hmac = crypto.createHmac('sha256', webhookSecret);
let signature = hmac.update(body).digest('hex');

console.log(signature);
```

2. 從儲存程式碼的同一目錄中執行下列命令來計算簽章。複製程式碼輸出的簽章。

```
node calculate_signature.mjs
```

現在，您可以在主控台中使用測試 HTTP 請求來測試函式程式碼。

在主控台中測試函式

1. 選取函式的程式碼索引標籤。
2. 在測試事件區段中，選擇建立新測試事件
3. Event Name (事件名稱) 輸入 **myEvent**。
4. 將下列項目複製並貼入事件 JSON 窗格中，取代現有的 JSON。將 Webhook 簽章取代為在上一個步驟中計算得出的值。

```
{
  "headers": {
    "Content-Type": "application/json",
    "x-webhook-signature":
 "2d672e7a0423fab740fbc040e801d1241f2df32d2ffd8989617a599486553e2a"
  },
  "body": "{\"type\": \"payment.success\", \"orderId\": \"1234\", \"amount\":
 \"99.99\"}"
```

```
}
```

5. 選擇儲存。
6. 選擇調用。

您應該會看到類似下列的輸出：

Python

```
Status: Succeeded
Test Event Name: myEvent

Response:
{
  "statusCode": 200,
  "body": "{\"received\": true}"
}

Function Logs:
START RequestId: 50cc0788-d70e-453a-9a22-ceaa210e8ac6 Version: $LATEST
Processing successful payment for order 1234
END RequestId: 50cc0788-d70e-453a-9a22-ceaa210e8ac6
REPORT RequestId: 50cc0788-d70e-453a-9a22-ceaa210e8ac6 Duration: 1.55 ms Billed
 Duration: 2 ms Memory Size: 128 MB Max Memory Used: 36 MB Init Duration: 136.32
 ms
```

Node.js

```
Status: Succeeded
Test Event Name: myEvent

Response:
{
  "statusCode": 200,
  "body": "{\"received\":true}"
}

Function Logs:
START RequestId: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 Version: $LATEST
2025-01-10T18:05:42.062Z e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 INFO Processing
 successful payment for order 1234
END RequestId: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4
```

```
REPORT RequestId: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 Duration: 60.10 ms Billed
 Duration: 61 ms Memory Size: 128 MB Max Memory Used: 72 MB Init Duration:
 174.46 ms

Request ID: e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4
```

使用 HTTP 請求測試函式

使用 curl 命令列工具來測試 Webhook 端點。

使用 HTTP 請求測試函式

1. 在終端或 Shell 程式中，執行下列 curl 命令。將 URL 取代為您自己的函式 URL 端點的值，並將 Webhook 簽章取代為您使用自己的秘密金鑰計算得出的簽章。

   ```
   curl -X POST https://ryqgmbx5xjzxahif6frvzikpre0bpvpf.lambda-url.us-west-2.on.aws/
    \
   -H "Content-Type: application/json" \
   -H "x-webhook-
   signature: d5f52b76ffba65ff60ea73da67bdf1fc5825d4db56b5d3ffa0b64b7cb85ef48b" \
   -d '{"type": "payment.success", "orderId": "1234", "amount": "99.99"}'
   ```

   您應該會看到下列輸出：

   ```
   {"received": true}
   ```

2. 執行下列動作，檢查函式的 CloudWatch 日誌，確認其正確剖析了承載：

   a. 在 Amazon CloudWatch 主控台中，開啟日誌群組頁面。

   b. 選取函式的日誌群組 (/aws/lambda/myLambdaWebhook)。

   c. 選取最新的日誌串流。

      您應該會在函式日誌中看到類似如下的輸出：

      Python

      ```
      Processing successful payment for order 1234
      ```

Node.js

```
2025-01-10T18:05:42.062Z e54fe6c7-1df9-4f05-a4c4-0f71cacd64f4 INFO
 Processing successful payment for order 1234
```

3. 執行下列 curl 命令，確認程式碼偵測到無效的簽章。將 URL 取代為您自己的函式 URL 端點。

```
curl -X POST https://ryqgmbx5xjzxahif6frvzikpre0bpvpf.lambda-url.us-west-2.on.aws/
 \
-H "Content-Type: application/json" \
-H "x-webhook-signature: abcdefg" \
-d '{"type": "payment.success", "orderId": "1234", "amount": "99.99"}'
```

您應該會看到下列輸出：

```
{"error": "Invalid signature"}
```

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。透過刪除您不再使用 AWS 的資源，您可以避免不必要的 費用 AWS 帳戶。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **confirm**，然後選擇刪除。

當您在主控台中建立 Lambda 函式時，Lambda 也會為函式建立執行角色。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取 Lambda 建立的執行角色。角色名稱具有格式 myLambdaWebhook-role-<random string>。
3. 選擇 刪除。

4. 在文字輸入欄位中輸入角色的名稱，然後選擇刪除。

了解 Lambda 函數擴展

並行是 AWS Lambda 函數同時處理的傳輸中請求數量。Lambda 會針對每個並行請求佈建個別的執行環境執行個體。函數收到更多請求時，Lambda 會自動處理執行環境的擴展數量，直到達到帳戶的並行上限為止。根據預設，Lambda 會為您的帳戶提供一個 AWS 區域中所有函數的總並行上限 1,000 個並行執行數。為了支援您的特定帳戶需求，您可以請求提高配額，並設定函數層級並行控制項，讓您的重要函數不會發生限流。

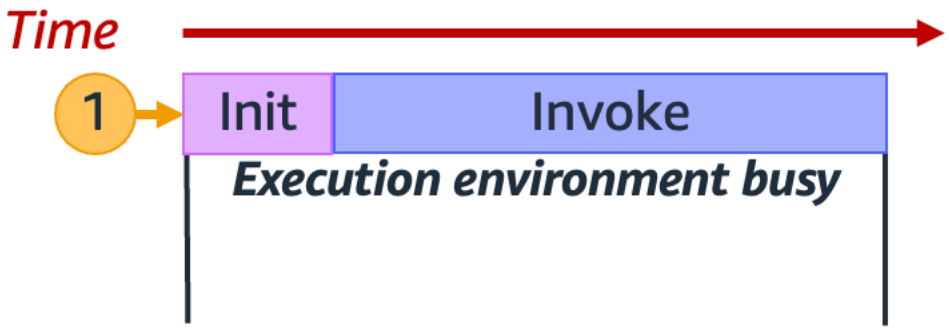
本主題說明 Lambda 中的並行和函數擴展。本主題結束時，您將能了解如何計算並行、視覺化兩個主要並行控制選項 (預留和佈建)、預估適當的並行控制設定，以及檢視指標以進一步最佳化。

章節

- 了解和視覺化並行
- 計算函數的並行
- 了解預留並行和佈建並行
- 了解並行和每秒請求數
- 並行配額
- 設定函數的預留並行
- 設定函數的佈建並行
- Lambda 擴展行為
- 監控並行

了解和視覺化並行

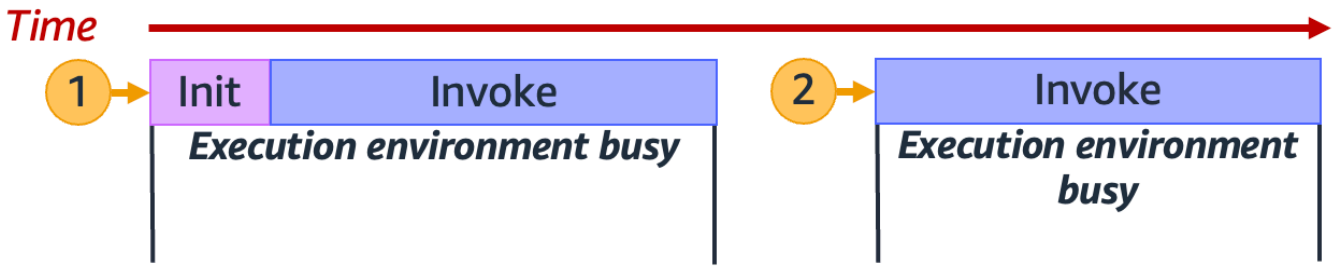
Lambda 會在安全且隔離的執行環境中調用函數。為了處理請求，Lambda 必須先初始化執行環境 (初始化階段)，然後才能用它來調用函數 (調用階段)：



> **Note**
>
> 實際初始化和調用持續時間可能因許多因素而異，例如您選擇的執行階段和 Lambda 函數程式碼。上圖表的用意並非表示初始化和調用階段持續時間的確切比例。

上圖使用矩形來代表單一執行環境。當函數收到第一個請求 (以帶有 1 標籤的黃色圓圈表示)，Lambda 會建立一個新的執行環境，並在初始化階段於主處理常式之外執行程式碼。接著 Lambda 會在調用階段執行函數的主要處理常式程式碼。在整個過程中，此執行環境會處於忙碌狀態，無法處理其他請求。

Lambda 完成處理第一個請求後，此執行環境就可以處理同一個函數的其他請求。Lambda 不需要為後續請求重新初始化環境。



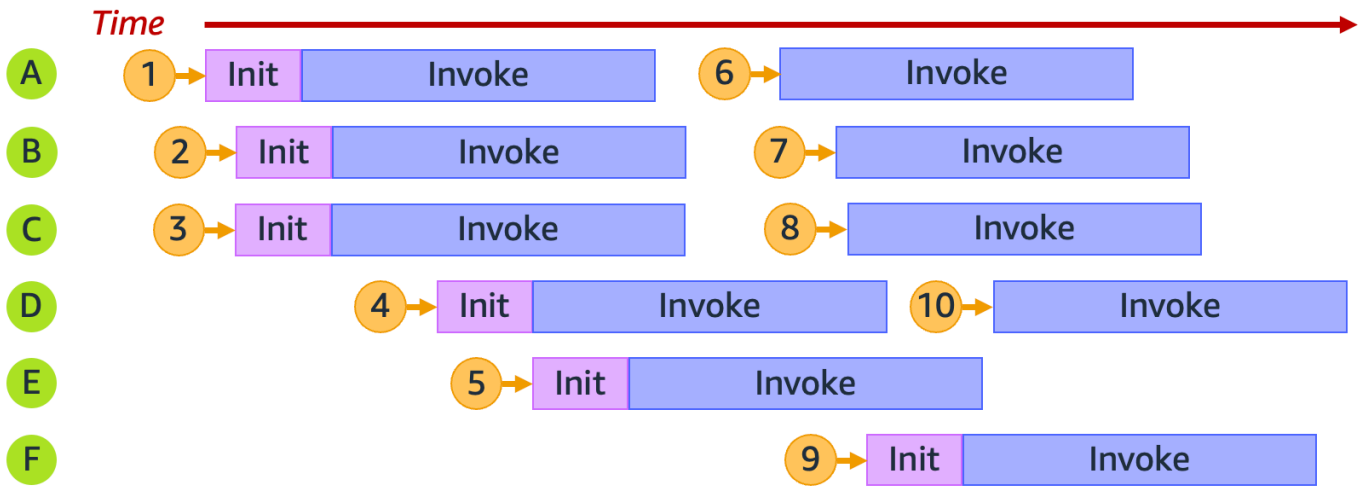
在上圖中，Lambda 重複使用執行環境來處理第二個請求 (以帶 2 標籤的黃色圓圈表示)。

目前為止，我們只將注意力放在執行環境的單一執行個體 (即並行 1)。實際上，Lambda 可能需要平行佈建多個執行環境執行個體，以便處理所有傳入的請求。當函數收到新請求，可能會發生的情況有以下兩種：

- 如果預先初始化的執行環境執行個體可用，Lambda 會用它來處理請求。

- 若不可用，Lambda 便會建立新的執行環境執行個體來處理請求。

例如，我們來看看當函數收到 10 個請求時會發生什麼情形：

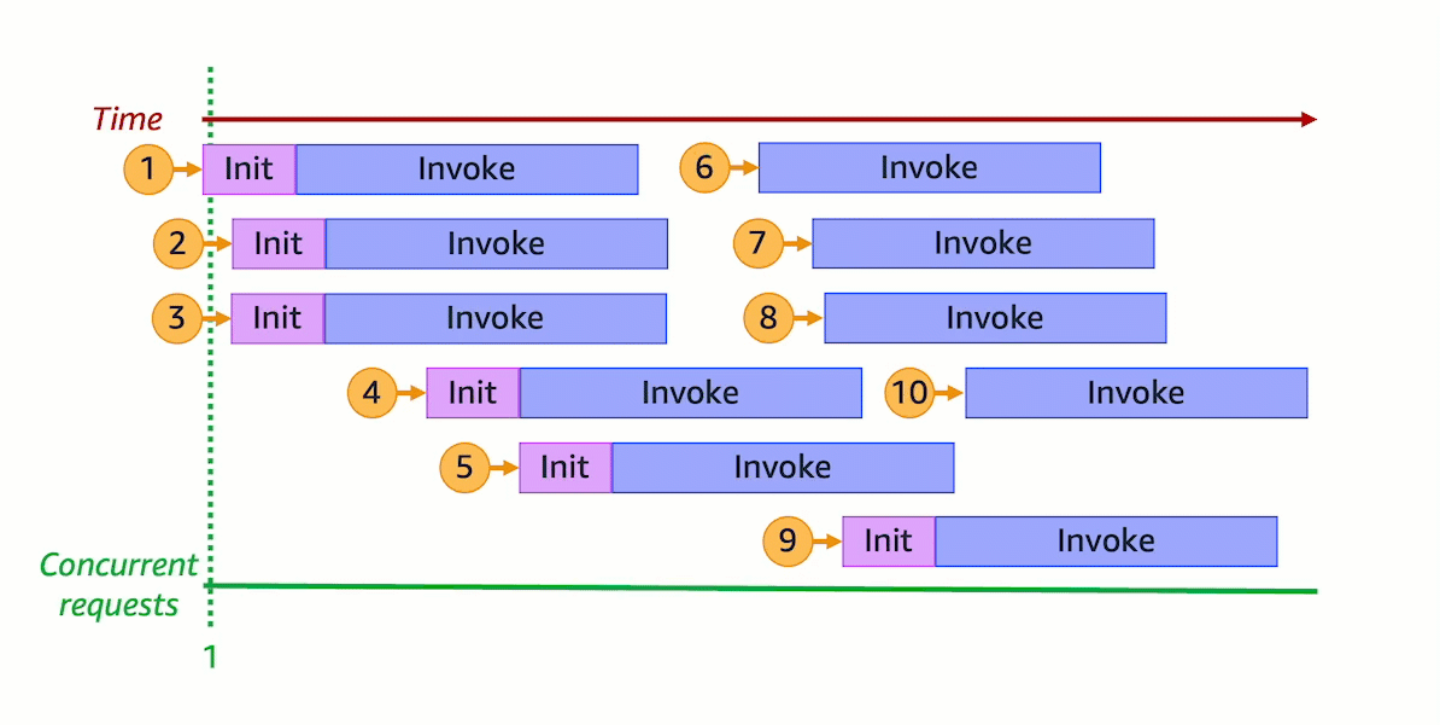


在上圖中，每個水平平面都代表單一執行環境執行個體 (以 A 至 F 標記)。以下是 Lambda 處理每個請求的方式：

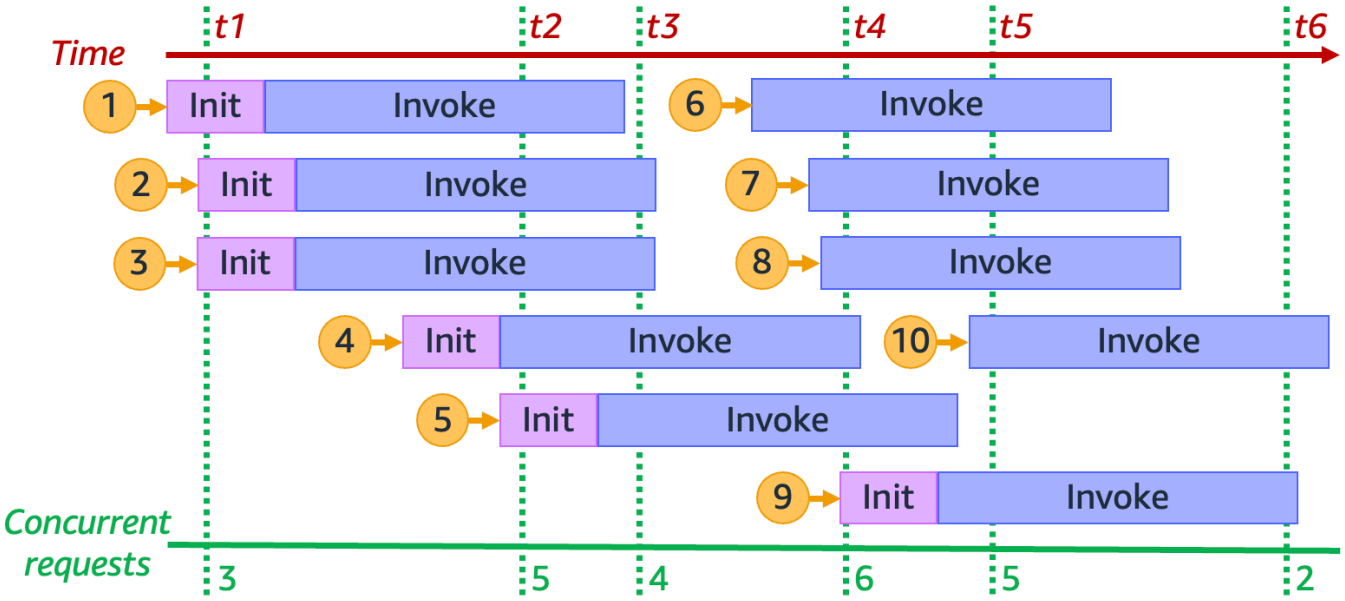
請求	Lambda 行為	推理
1	佈建新環境 A	這是第一個請求；沒有可用的執行環境執行個體。
2	佈建新環境 B	現有的執行環境執行個體 A 忙碌中。
3	佈建新環境 C	現有的執行環境執行個體 A 和 B 都忙碌中。
4	佈建新環境 D	現有的執行環境執行個體 A、B 和 C 都忙碌中。
5	佈建新環境 E	現有的執行環境執行個體 A、B、C 和 D 都忙碌中。
6	重複使用環境 A	執行環境執行個體 A 已完成處理請求 1，現在可供使用。

請求	Lambda 行為	推理
7	重複使用環境 B	執行環境執行個體 B 已完成處理請求 2，現在可供使用。
8	重複使用環境 C	執行環境執行個體 C 已完成處理請求 3，現在可供使用。
9	佈建新環境 F	現有的執行環境執行個體 A、B、C、D 和 E 都忙碌中。
10	重複使用環境 D	執行環境執行個體 D 已完成處理請求 4，現在可供使用。

當函數收到更多並行請求時，Lambda 會擴展執行環境執行個體的數量做為回應。以下動畫追蹤一段時間內並行請求的數目：



將上方動畫凍結在六個不同的時間點，我們可以得到下圖：



在上圖中，我們可以在任何時間點繪製一條垂直線，並計算與此線相交的環境數量。這可以得出該時間點並行請求的數量。例如，在 `t1` 時間點，有三個作用中環境可以處理三個並行請求。在 `t4` 時間點有六個作用中環境可處理六個並行請求，達到此模擬中的並行請求數上限。

總之，函數的並行就是代表函數同時處理的請求數目。為了回應函數並行的增加，Lambda 會佈建更多執行環境執行個體以滿足請求的需求。

計算函數的並行

在一般情況下，一個系統的並行是同時處理多個任務的能力。在 Lambda 中，並行是函數可同時處理的傳輸中請求數目。測量 Lambda 函數並行的快速實用方法是使用以下公式：

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

並行與每秒請求數不同。例如，假設函數平均每秒收到 100個請求。如果平均請求持續時間為一秒，那麼並行也會是 100：

```
Concurrency = (100 requests/second) * (1 second/request) = 100
```

但是，如果平均請求持續時間為 500 毫秒，則並行為 50：

```
Concurrency = (100 requests/second) * (0.5 second/request) = 50
```

在實務中並行 50 代表什麼意思？ 如果平均請求持續時間為 500 毫秒，則可以將函數的一個執行個體視為每秒能處理兩個請求。如此一來，函數需要 50 個執行個體來處理每秒 100 個請求的負載。並行 50 表示 Lambda 必須佈建 50 個執行環境執行個體，才能有效率地處理此工作負載，而不需要進行限流。底下以方程式的形式表示：

```
Concurrency = (100 requests/second) / (2 requests/second) = 50
```

如果函數收到的請求數量是兩倍 (每秒 200 個請求)，但處理每個請求只需要一半的時間 (250 毫秒)，則並行仍然是 50：

```
Concurrency = (200 requests/second) * (0.25 second/request) = 50
```

測驗您對並行的理解

假設您有一個函數，執行時間平均需要 200 毫秒。在峰值負載期間，您觀察到每秒有 5,000 個請求。峰值負載期間函數的並行為何？

答案

平均函數持續時間為 200 毫秒或 0.2 秒。使用並行公式，您可以插入數字來得出並行為 1,000：

```
Concurrency = (5,000 requests/second) * (0.2 seconds/request) = 1,000
```

或者，平均函數持續時間為 200 毫秒表示函數每秒可以處理 5 個請求。若要處理每秒 5,000 個請求的工作負載，您需要 1,000 個執行環境執行個體。因此並行為 1,000：

```
Concurrency = (5,000 requests/second) / (5 requests/second) = 1,000
```

了解預留並行和佈建並行

根據預設，您帳戶設有一個區域中所有函數的並行執行數上限 1,000。您的函數會隨需共用此 1,000 並行的集區。如果您用完可用的並行，您的函數會遇到限流（也就是，它們開始捨棄請求）。

某些函數的重要性可能高於其他函數。因此，您可能會想配置並行設定，確保重要函數可獲得所需的並行。並行控制項有兩種：預留並行和佈建並行。

- 使用預留並行即可設定並行執行個體數量上限與下限，為函式保留帳戶中的一部分並行配額。如果您不希望其他函數占用所有可用的未預留並行，此功能非常有用。當某個函數具有預留並行時，其他函數都無法使用該並行。

- 使用佈建並行為函數預先初始化多個環境執行個體。這對於縮短冷啟動延遲很有幫助。

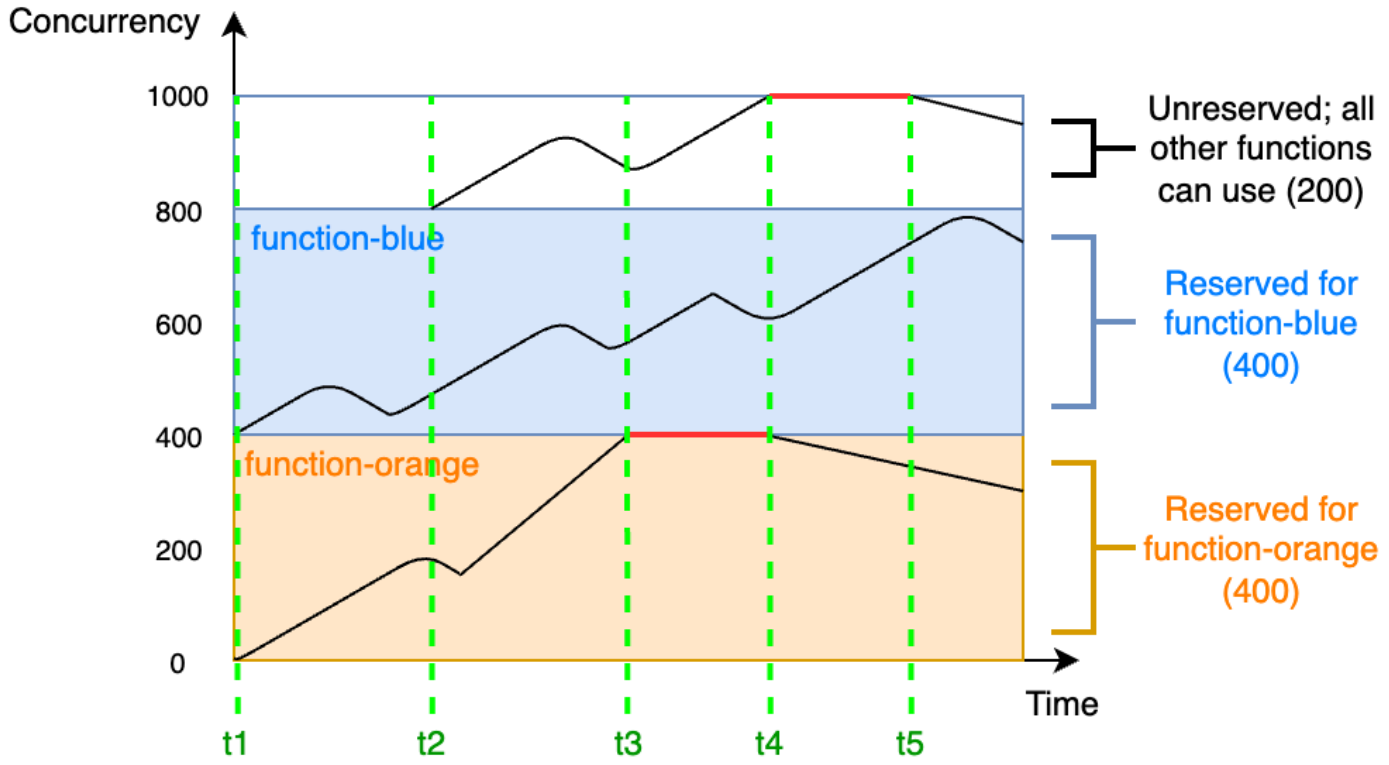
預留並行

如果您想要保證函數隨時可以使用一定數量的並行，請使用預留並行。

預留並行會設定要配置給函式的並行執行個體數量上限與下限。將預留並行專門留給某個函數時，其他函數都無法使用該並行。換句話說，設定預留並行可能會影響其他函數可用的並行集區。沒有預留並行的函數會共用未預留剩餘的並行集區。

設定預留並行會計入您的整體帳戶並行上限。設定函數的預留並行不收費。

為了更清楚理解預留並行，請參考下圖：



在本圖中，您的帳戶在此區域中所有函數的並行上限為預設值 1,000。假設您有兩個重要函數 `function-blue` 和 `function-orange`，且經常預期出現高調用量。您決定將 400個單位的預留並行提供給 `function-blue`，並為 `function-orange` 提供 400 單位的預留並行。在此範例中，您帳戶中的所有其他函數必須共用剩餘 200 單位的未預留並行。

本圖有五個特點：

- 在 t1，function-orange 和 function-blue 都開始接收請求。每個函數開始使用自己分配到的預留並行單元。
- 在 t2，function-orange 和 function-blue 穩定接收更多的請求。同時，您部署了一些其他 Lambda 函數，且這些函數開始接收請求。您沒有將預留並行分配給這些函數。這些函數開始使用剩餘 200 單位的未預留並行。
- 在 t3，function-orange 達到並行上限 400。雖然您帳戶中的其他地方存在未使用的並行，但 function-orange 無法存取。紅線表示 function-orange 正在進行限流，且 Lambda 可能會捨棄請求。
- 在 t4，function-orange 接收的請求開始變少，而且不再限流。但是，其他函數出現流量尖峰並開始限流。雖然您帳戶中的其他地方存在未使用的並行，但這些其他函數無法存取。紅線表示其他函數正在進行限流。
- 在 t5，其他函數接收的請求開始變少，而且不再限流。

在此範例中，請注意預留並行產生了下列影響：

- 函數可以獨立於帳戶中的其他函數進行擴展。您的帳戶下相同區域中沒有預留並行的所有函數，都會共用未預留的並行集區。如果沒有預留並行，其他函數可能會用盡所有可用的並行。這可以視需要防止您的重要數擴展。
- 您的函數無法無止盡擴展。預留並行會對函式的並行上下限設定限制。這表示函數不能使用為其他函數預留的並行，也不能使用未預留集區中的並行。此外，預留並行兼具下限與上限作用：會預留指定的容量專用於您的函式，也會防止容量擴展超過上限。您可以預留並行以防止函數用完帳戶中的所有可用並行，或過載下游資源。
- 您可能無法使用帳戶可用的所有並行。預留並行會計入您帳戶的並行上限，但這也表示其他函數無法使用該部分預留並行。如果函數沒有用盡您為它預留的所有並行，那麼實際上就浪費了這些並行。除非浪費的並行可讓您帳戶中的其他函數獲益，這才不會成為問題。

若要管理函數的預留並行設定，請參閱：設定函數的預留並行。

佈建並行

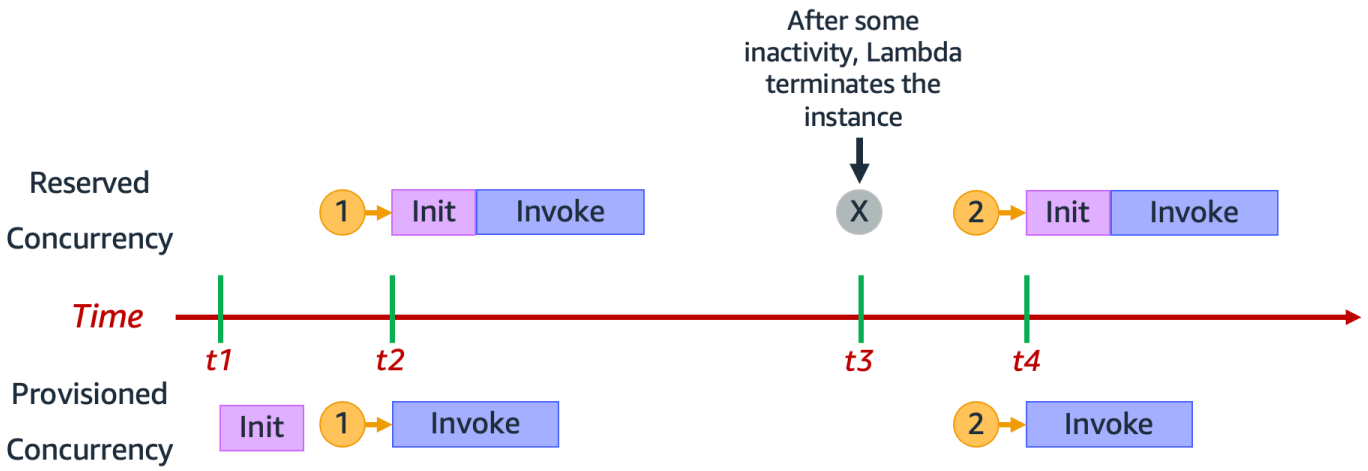
您可以使用預留並行來定義為 Lambda 函數預留的執行環境數目上限。不過，這些環境都不會預先初始化。因此，函數調用可能需要花更長的時間，因為 Lambda 必須先初始化新環境，才能用來調用函數。當 Lambda 必須初始化新環境才能執行調用時，就稱為冷啟動。若要緩解冷啟動情形，您可以使用佈建並行。

佈建並行是您要分配給函數的預先初始化執行環境數。如果您為函數設定了佈建並行，Lambda 便會初始化該數量的執行環境，以便立即回應函數請求。

> **Note**
>
> 設定佈建並行時，您的帳戶會產生額外費用。如果您使用 Java 11 或 Java 17 執行期，您也可以使用 Lambda SnapStart 來解決冷啟動問題，而無需額外付費。SnapStart 會使用執行環境的快取快照來顯著改善啟動效能。您無法對相同的函數版本同時使用 SnapStart 和佈建並行。如需有關 SnapStart 功能、限制和支援區域的詳細資訊，請參閱 使用 Lambda SnapStart 改善啟動效能。

使用佈建並行時，Lambda 仍會在背景回收執行環境。例如，調用失敗後可能會發生這種情況。但是在任何給定時間，Lambda 總是會確保預先初始化的環境數量等於函數佈建並行設定的值。重要的是，即使您使用的是佈建並行，如果 Lambda 必須重設執行環境，您仍然可能會遇到冷啟動延遲。

與之相反，如果使用的是預留並行，Lambda 可能會在閒置一段時間後完全終止環境。下圖說明這一點，比較為函數設定預留並行和佈建並行的情況下，單一執行環境的生命週期有何差異。

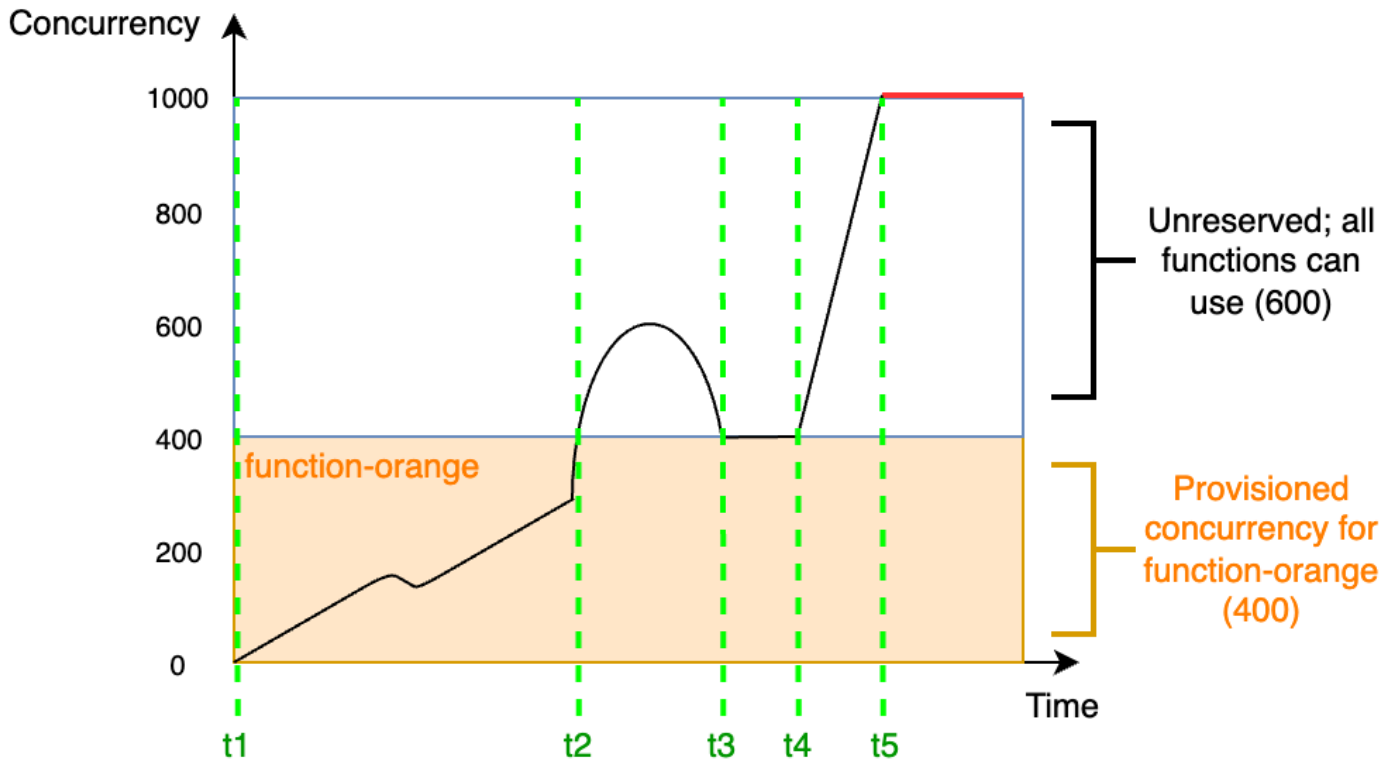


本圖有四個特點：

時間	預留並行	佈建並行
t1	什麼都沒發生。	Lambda 會預先初始化執行環境執行個體。

時間	預留並行	佈建並行
t2	請求 1 傳入。Lambda 必須初始化新的執行環境執行個體。	請求 1 傳入。Lambda 使用預先初始化的環境執行個體。
t3	閒置一段時間後，Lambda 會終止作用中環境執行個體。	什麼都沒發生。
t4	請求 2 傳入。Lambda 必須初始化新的執行環境執行個體。	請求 2 傳入。Lambda 使用預先初始化的環境執行個體。

為了更清楚理解佈建並行，請參考下圖：



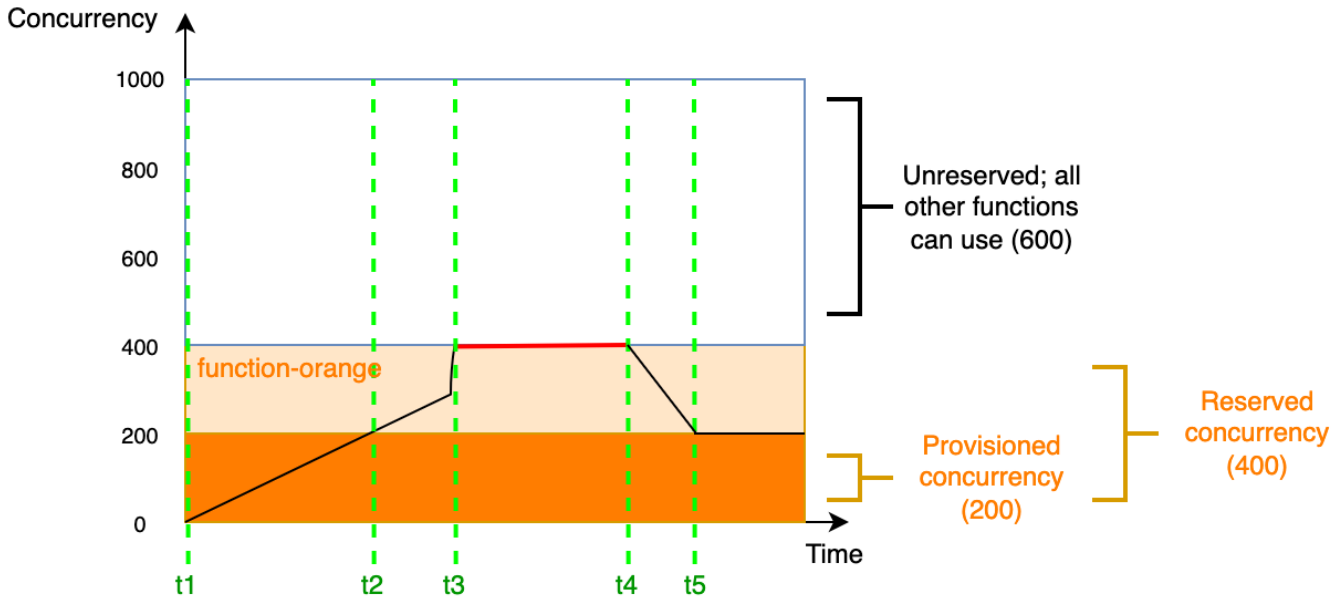
在此圖中，您的帳戶並行上限為 1,000。您決定將 400 單位的佈建並行提供給 `function-orange`。您帳戶中包括 `function-orange` 在內的所有函數，都可以使用剩餘 600 單位的未預留並行。

本圖有五個特點：

- 在 `t1`，`function-orange` 開始接收請求。由於 Lambda 已預先初始化 400 個執行環境執行個體，因此 `function-orange` 已準備好立即進行調用。

- 在 t2，function-orange 達到 400 個並行請求。因此，function-orange 用盡了佈建並行。但是由於仍有未預留並行可用，Lambda 還是可以用來處理 function-orange 的其他請求 (沒有限流)。Lambda 必須建立新的執行個體來處理這些請求，而且函數可能會發生冷啟動延遲。
- 在 t3，流量短暫達到峰值後 function-orange 回到 400 個並行請求。Lambda 再度能在沒有冷啟動延遲的情況下處理所有請求。
- 在 t4,帳戶中的函數發生流量暴增情形。此暴增可能來自 function-orange 或帳戶中的任何其他函數。Lambda 使用未預留並行處理這些請求。
- 在 t5，帳戶中的函數達到最大並行上限 1,000，並且發生限流。

上一個範例只考慮了佈建並行。實際上，您可以對函數同時設定佈建並行和預留並行。如果您有一個函數在工作日期間負責處理調用的一致性負載，但週末期間經常會遇到流量峰值，便可以這麼做。在這種情況下，您可以使用佈建並行設定環境的基準數量來處理平日的請求，並使用預留並行處理週末峰值流量。請思考下圖情形：



在此圖中，假設您為 function-orange 設定了 200 單位的佈建並行，以及 400 單位的預留並行。因為您已設定預留並行，因此 function-orange 完全無法使用 600 單位的未預留並行。

此圖有 5 個特點：

- 在 t1，function-orange 開始接收請求。由於 Lambda 已預先初始化 200 個執行環境執行個體，因此 function-orange 已準備好立即進行調用。

- 在 t2，function-orange 用盡了所有佈建並行。function-orange 可以使用預留並行繼續處理請求，但這些請求可能會遇到冷啟動延遲。
- 在 t3，function-orange 達到 400 個並行請求。因此，function-orange 用盡了所有預留並行。由於 function-orange 不能使用未預留並行，因此請求開始限流。
- 在 t4，function-orange 接收的請求開始變少，而且不再限流。
- 在 t5，function-orange 下降到 200 個並行請求，因此所有請求都能再次使用佈建並行 (即無冷啟動延遲)。

預留並行和佈建並行都會計入您的帳戶並行上限和區域配額中。換句話說，分配預留和佈建並行可能會影響其他函數可用的並行集區。設定佈建並行會產生 費用 AWS 帳戶。

> **Note**
>
> 如果在函數版本與別名功能上佈建的並行數量加到函數的預留並行，則所有呼叫都會在佈建的並行上執行。此組態也有對未發佈版本函數 ($LATEST) 進行節流的效果，以防止其執行。您無法配置多於函數預留並行的佈建並行。

若要管理函數的佈建並行設定，請參閱 設定函數的佈建並行。若要根據排程或應用程式使用率自動佈建並行擴展，請參閱 使用 Application Auto Scaling 自動化佈建並行管理。

Lambda 如何配置佈建並行

佈建並行不會在設定後立即上線。Lambda 會在準備一兩分鐘後，開始配置佈建的並行。對於每個函數，Lambda 每分鐘最多可以佈建 6，000 個執行環境，無論如何 AWS 區域。這與函數的並行擴展率完全相同。

當您提交配置佈建並行的請求時，在 Lambda 完全配置完成之前，您無法存取任何這些環境。例如，如果您請求 5,000 個佈建並行，在 Lambda 完全完成分配 5,000 個執行環境之前，您的請求都無法使用佈建並行。

比較預留並行和佈建並行

下表總結並比較預留和佈建並行。

主題	預留並行	佈建並行
定義	函數的執行環境執行個體數目上限。	為函數設定預先佈建的執行環境執行個體數目。
佈建行為	Lambda 會隨需佈建新的執行個體。	Lambda 會預先佈建執行個體 (即在函數開始接收請求之前)。
冷啟動行為	由於 Lambda 必須隨需建立新執行個體，因此可能會發生冷啟動延遲。	由於 Lambda 不需要隨需建立執行個體，因此不可能會冷啟動延遲。
限流行為	達到預留並行上限時，會對函數限流。	如果沒有設定預留並行：達到佈建並行上限時，函數會使用未預留的並行。 如果有設定預留並行：達到預留並行上限時，會對函數限流。
未設定情況下的預設行為	函數會使用帳戶中可用的未預留並行。	Lambda 不會預先佈建任何執行個體。反之，如果沒有設定預留並行：函數會使用帳戶中可用的未預留並行。 如果有設定預留並行：函數會使用預留並行。
定價	不收取額外費用。	會產生額外費用。

了解並行和每秒請求數

如前一節所述，並行與每秒請求數不同。這是使用平均請求持續時間小於 100 毫秒的函數時特別重要的區別。

對於帳戶中的所有函數，Lambda 會強制執行每秒請求數限制，等於帳戶並行的 10 倍。例如，由於預設帳戶並行限制為 1,000，因此您帳戶中的函數每秒最多可處理 10,000 個請求。

例如，假設有一個平均請求持續時間為 50 毫秒的函數。在每秒 20,000 個請求時，此函數的並行如下：

```
Concurrency = (20,000 requests/second) * (0.05 second/request) = 1,000
```

根據此結果，您可能會預期帳戶並行限制為 1,000 就足以處理此負載。不過，由於每秒 10,000 個請求的限制，您的函數每秒只能處理 20,000 個請求總數中的 10,000 個。此函數遇到了限流。

因此，在為函數配置並行設定時，您必須同時考慮並行和每秒請求數。這種情況下，您需要請求將帳戶並行限制提高到 2,000，因為這會將每秒請求總數限制提高到 20,000。

> **Note**
>
> 根據此每秒請求數限制，假設每個 Lambda 執行環境每秒最多只能處理 10 個請求是不正確的。計算配額時，Lambda 不會觀察任何個別執行環境的負載，而只會考慮整體並行情況和每秒的整體請求數。

測試您對並行的理解 (低於 100 毫秒函數)

假設您有一個函數，執行時間平均需要 20 毫秒。在峰值負載期間，您觀察到每秒有 30,000 個請求。峰值負載期間函數的並行為何？

答案

平均函數持續時間為 20 毫秒或 0.02 秒。使用並行公式，您可以插入數字來得出並行為 600：

```
Concurrency = (30,000 requests/second) * (0.02 seconds/request) = 600
```

依預設，1,000 的帳戶並行限制似乎足以處理此負載。不過，每秒 10,000 個請求的限制不足以應對每秒 30,000 個傳入請求的狀況。若要完全容納 30,000 個請求，您需要請求將帳戶並行限制提高到 3,000 或更高。

每秒請求數限制適用於 Lambda 中涉及並行的所有配額。換句話說，它適用於同步隨需函數、使用佈建並行的函數，以及並行擴展行為。例如，在以下幾種情況中，您必須仔細考慮並行和每秒請求數限制：

- 使用隨需並行的函數每 10 秒的並行可能爆增 500，或每 10 秒內的每秒請求數增加 5,000，以先發生者為準。

- 假設您有一個函數，其佈建並行分配為 10。此函數會在 10 個並行或每秒 100 個請求後 (以先發生者為準) 溢出到隨需並行。

並行配額

Lambda 會針對區域中所有函數可用的並行總量設定配額。這些配額存在於兩個層級：

- 帳戶層級，預設情況下，函數最多可有 1,000 單位的並行。若要提升配額，請參閱《Service Quotas 使用者指南》中的請求提升配額。
- 函數層級，預設情況下，您可在所有函數間保留最多 900 單位的並行。無論您的總帳戶並行限制為何，Lambda 一律會為未明確保留並行保留的函數保留 100 個並行單位。例如，如果您將帳戶並行上限增加到 2,000，就可以在函數層級預留最多 1,900 單位的並行。
- 在帳戶層級和函數層級，Lambda 也會強制執行每秒 10 倍於對應並行配額的請求限制。例如，這適用於帳戶層級並行、使用隨需並行的函數、使用佈建並行的函數，以及並行擴展行為。如需詳細資訊，請參閱the section called “了解並行和每秒請求數”。

若要檢查目前的帳戶層級並行配額，請使用 AWS Command Line Interface (AWS CLI) 執行下列命令：

```
aws lambda get-account-settings
```

您應該會看到類似以下的輸出：

```
{
    "AccountLimit": {
        "TotalCodeSize": 80530636800,
        "CodeSizeUnzipped": 262144000,
        "CodeSizeZipped": 52428800,
        "ConcurrentExecutions": 1000,
        "UnreservedConcurrentExecutions": 900
    },
    "AccountUsage": {
        "TotalCodeSize": 410759889,
        "FunctionCount": 8
    }
}
```

`ConcurrentExecutions` 是您的帳戶層級並行配額總計。`UnreservedConcurrentExecutions` 是您仍可配置給函數的保留並行數量。

函數收到更多請求時，Lambda 會自動提高執行環境的數量來處理這些請求，直到您的帳戶達到並行配額為止。但是，為了防止因應突然爆發的流量而過度擴展，Lambda 限制了函數擴展的速度。此並行擴展速率是您帳戶中的函數可擴展以因應增加的請求的最高速率。(也就是說，Lambda 建立新執行環境的速度可以有多快。) 並行擴展速率與帳戶層級並行限制不同，後者是您的函數可用的並行總量。

在每個 AWS 區域和每個函數中，您的並行擴展速率為每 10 秒 1，000 個執行環境執行個體 （或每 10 秒每秒 10，000 個請求）。換句話說，Lambda 每隔 10 秒就可以為每個函數配置最多 1,000 個額外執行環境的執行個體，或者每秒容納 10,000 個額外請求。

通常情況下，您不需要擔心此限制。對於大多數使用案例，Lambda 的擴展速率已足夠。

重要的是，並行擴展速率是函數層級限制。這意味著帳戶中的每個函數都可以獨立於其他函數進行擴展。

如需擴展行為的詳細資訊，請參閱 Lambda 擴展行為。

設定函數的預留並行

在 Lambda 中，並行是函數目前正在處理的傳輸中請求數目。有兩種類型的並行控制項可用：

- 預留並行 – 此控制項設定配置給函式的並行執行個體數量上限與下限。當某個函數具有預留並行時，其他函數都無法使用該並行。預留並行有助於確保最重要的函數始終有足夠的並行數量來處理傳入請求。此外，預留並行可用於限制並行量，防止資料庫連線等下游資源不堪重負。預留並行兼具下限與上限作用 – 會預留指定的容量專用於您的函式，也會防止容量擴展超過上限。設定函數的預留並行不會產生額外費用。
- 佈建並行 – 它是您分配給函數的預先初始化執行環境數目。這些執行環境已準備好立即回應傳入的函數請求。佈建並行對於降低函式的冷啟動延遲非常有用，旨在使函式的回應時間降低到兩位數的毫秒級。一般而言，互動式工作負載能從這項功能中獲益最多。這些是使用者啟動請求的應用程式 (例如 Web 和行動應用程式)，對延遲最敏感。非同步工作負載 (如資料處理管道) 通常對延遲較不敏感，因此通常不需要佈建並行。設定佈建並行會對您的 AWS 帳戶 帳戶產生額外費用。

本主題詳細說明如何管理及設定預留並行。如需這兩種並行控制項的概念性概觀，請參閱預留並行與佈建並行。如需有關設定佈建並行的資訊，請參閱 the section called “設定佈建並行”。

> **Note**
>
> 連結至 Amazon MQ 事件來源映射的 Lambda 函數具有預設並行上限。若使用 Apache Active MQ，並行執行個體數量上限為 5 個。若使用 ARabbit MQ，並行執行個體數量上限為 1 個。為函數設定保留或佈建並行不會改變這些上限。若要在使用 Amazon MQ 時要求增加預設的並行上限，請聯絡 支援。

章節

- 設定預留並行
- 準確估計函數所需的預留並行

設定預留並行

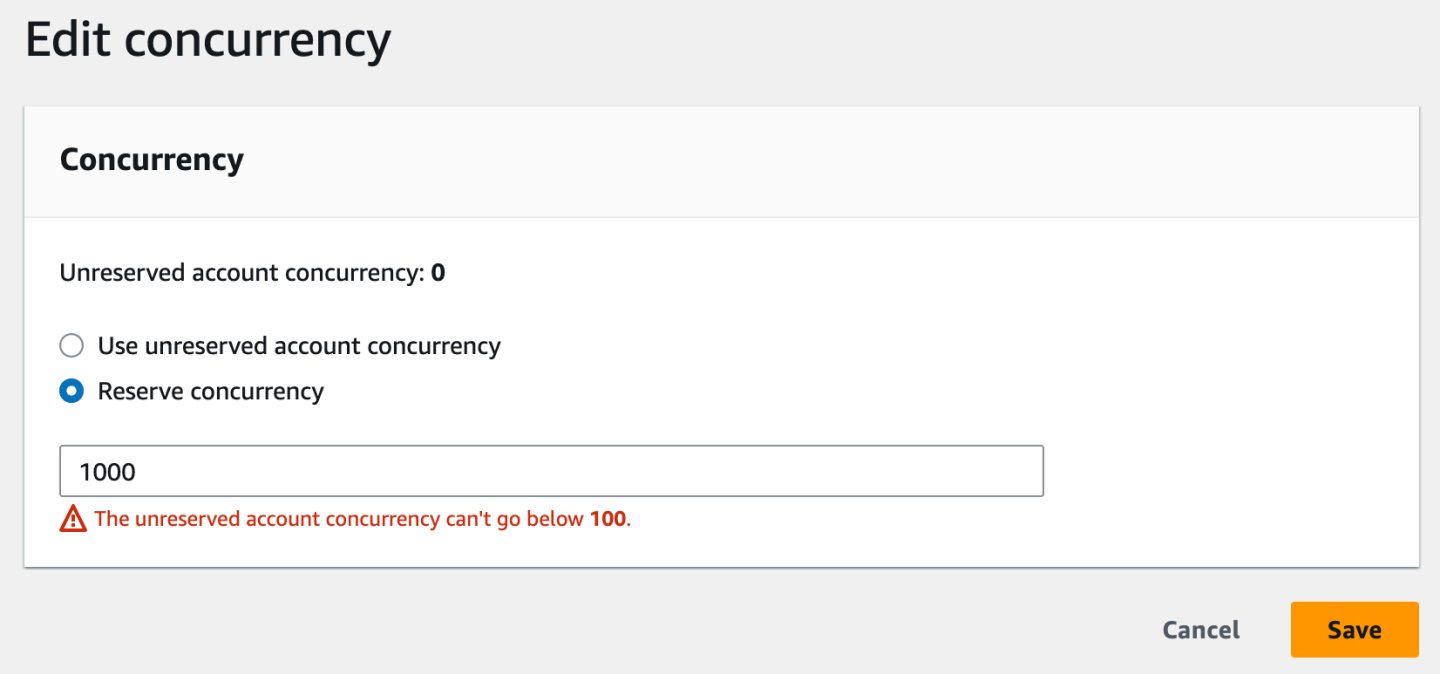
您可以使用 Lambda 主控台或 Lambda API 設定函數的預留並行設定。

預留函數並行的方式 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。

2. 選擇要預留並行的函數。
3. 選擇 Configuration (組態)，然後選擇 Concurrency (並行)。
4. 在 Concurrency (並行) 下，選擇 Edit (編輯)。
5. 選擇 Reserve concurrency (預留並行)。輸入要為函式預留的並行數量。
6. 選擇儲存。

您最多可以預留的數量為未預留帳戶並行數量減去 100 的值。剩餘的 100 個並行單位會用於未使用預留並行的函數。例如，如果您帳戶的並行上限為 1,000，則您無法為單一函數預留全部 1,000 個並行單位。



為函數預留並行會影響其他函數可用的並行集區。例如，如果您為 `function-a` 預留 100 個並行單位，則帳戶中的其他函數必須共用剩餘的 900 個並行單位 (即使 `function-a` 未使用全部 100 個預留並行單位也一樣)。

若要刻意節流函數，請將其預留並行設為 0。此動作會防止函數處理任何事件，直到您移除限制為止。

若要透過 Lambda API 設定預留並行，請使用下列 API 操作。

- PutFunctionConcurrency
- GetFunctionConcurrency
- DeleteFunctionConcurrency

例如，若要透過 AWS Command Line Interface (CLI) 來設定預留並行，請使用 `put-function-concurrency` 命令。下列命令會為名為 `my-function` 的函數預留 100 個並行單位：

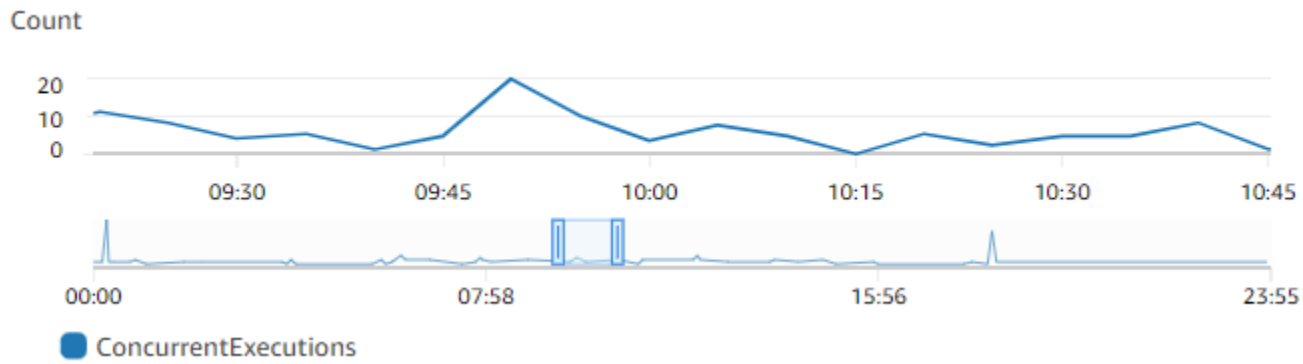
```
aws lambda put-function-concurrency --function-name my-function \
    --reserved-concurrent-executions 100
```

您應該會看到類似以下的輸出：

```
{
    "ReservedConcurrentExecutions": 100
}
```

準確估計函數所需的預留並行

如果函數目前正在為處理流量，您可以使用 CloudWatch 指標輕鬆檢視其並行指標。具體而言，`ConcurrentExecutions` 指標會顯示帳戶中每個函數的並行調用次數。



前一張圖表顯示此函數一天平均可處理 5 到 10 個並行請求，峰值時段則為 20 個請求。假設帳戶中還有很多其他函數。如果此函數對您的應用程式很重要，且您不想要捨棄任何請求，則請將預留並行設為 20 (或以上)。

或者，回想一下您也可以使用以下公式計算並行：

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

將每秒平均請求乘以平均請求持續時間 (以秒為單位)，可讓您粗略估計需要預留多少並行。您可以使用 `Invocation` 指標來預估每秒平均請求數，並使用 `Duration` 指標預估平均請求持續時間 (以秒為單位)。如需詳細資訊，請參閱將 CloudWatch 指標與 Lambda 搭配使用。

您應熟悉您的上游和下游輸送量限制。雖然 Lambda 函數可隨負載進行無縫擴展，但上游和下游相依性可能不具有相同的輸送量能力。如果您需要限制函數可擴展的高度，可以在函數中設定預留並行。

設定函數的佈建並行

在 Lambda 中，並行是函數目前正在處理的傳輸中請求數目。有兩種類型的並行控制項可用：

- 預留並行 – 此控制項設定配置給函式的並行執行個體數量上限與下限。當某個函數具有預留並行時，其他函數都無法使用該並行。預留並行有助於確保最重要的函數始終有足夠的並行數量來處理傳入請求。此外，預留並行可用於限制並行量，防止資料庫連線等下游資源不堪重負。預留並行兼具下限與上限作用 – 會預留指定的容量專用於您的函式，也會防止容量擴展超過上限。設定函數的預留並行不會產生額外費用。
- 佈建並行 – 它是您分配給函數的預先初始化執行環境數目。這些執行環境已準備好立即回應傳入的函數請求。佈建並行對於降低函式的冷啟動延遲非常有用，旨在使函式的回應時間降低到兩位數的毫秒級。一般而言，互動式工作負載能從這項功能中獲益最多。這些是使用者啟動請求的應用程式 (例如 Web 和行動應用程式)，對延遲最敏感。非同步工作負載 (如資料處理管道) 通常對延遲較不敏感，因此通常不需要佈建並行。設定佈建並行會對您的 AWS 帳戶 帳戶產生額外費用。

本主題詳細說明如何管理及設定佈建並行。如需這兩種並行控制項的概念性概觀，請參閱預留並行與佈建並行。如需有關設定預留並行的詳細資訊，請參閱the section called “設定預留並行”。

> **Note**
>
> 連結至 Amazon MQ 事件來源映射的 Lambda 函數具有預設並行上限。若使用 Apache Active MQ，並行執行個體數量上限為 5 個。若使用 ARabbit MQ，並行執行個體數量上限為 1 個。為函數設定保留或佈建並行不會改變這些上限。若要在使用 Amazon MQ 時要求增加預設的並行上限，請聯絡 支援。

章節

- 設定佈建並行
- 準確估算函數所需的佈建並行
- 使用佈建並行時最佳化函數程式碼
- 使用環境變數來檢視和控制佈建並行行為
- 了解使用佈建並行的記錄和計費行為
- 使用 Application Auto Scaling 自動化佈建並行管理

設定佈建並行

您可以使用 Lambda 主控台或 Lambda API 設定函數的佈建並行設定。

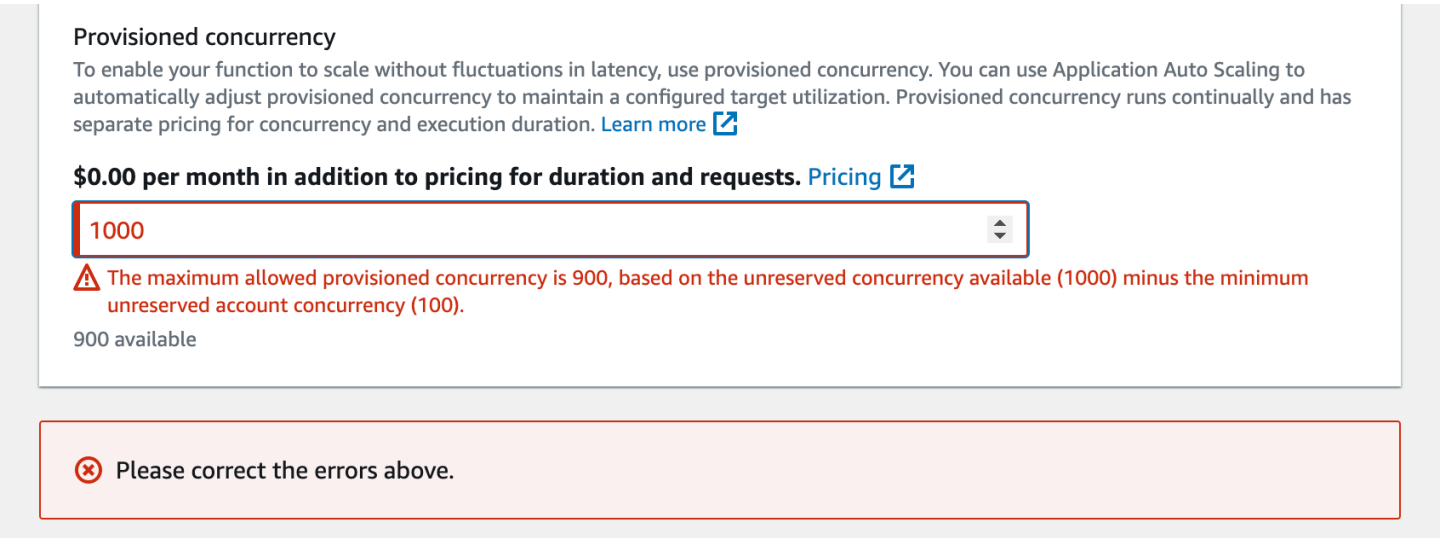
若要為函數配置佈建並行 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您要配置佈建並行的函數。
3. 選擇 Configuration (組態)，然後選擇 Concurrency (並行)。
4. 在 Provisioned concurrency configurations (佈建並行組態) 下方，選擇 Add configuration (新增組態)。
5. 選擇限定詞類型，以及別名或版本。

> **Note**
>
> 您無法將佈建並行與任何函數的 $LATEST 版本搭配使用。
> 如果您的函數有事件來源，請確定事件來源指向正確的函數別名或版本。否則，您的函數將不會使用佈建並行環境。

6. 在佈建並行下輸入一個數字。
7. 選擇儲存。

您最多可以在帳戶中設定的數量為未預留帳戶並行數量減去 100 的值。剩餘的 100 個並行單位會用於未使用預留並行的函數。例如，如果帳戶的並行限制為 1,000，而且您尚未將任何預留或佈建並行指派給任何其他函數，則單一函數最多可以設定 900 個佈建並行單位。



為函數設定佈建並行會影響其他函數可用的並行集區。例如，如果您為 function-a 設定 100 個佈建並行單位，則帳戶中的其他函數必須共用剩餘的 900 個並行單位。即使 function-a 沒有使用所有 100 個單位也是如此。

您可以為同一函數同時配置預留並行和佈建並行。在這種情況下，佈建並行不能超過預留並行。

此限制也適用於不同函數版本。您可以指派給特定函數版本的佈建並行上限等於函數的預留並行減去其他函數版本上的佈建並行。

若要透過 Lambda API 設定佈建並行，請使用下列 API 操作。

- PutProvisionedConcurrencyConfig
- GetProvisionedConcurrencyConfig
- ListProvisionedConcurrencyConfigs
- DeleteProvisionedConcurrencyConfig

例如，若要透過 AWS Command Line Interface (CLI) 來設定佈建並行，請使用 put-provisioned-concurrency-config 命令。下列命令會為名為 my-function 之函數的 BLUE 別名配置 100 個佈建並行單位：

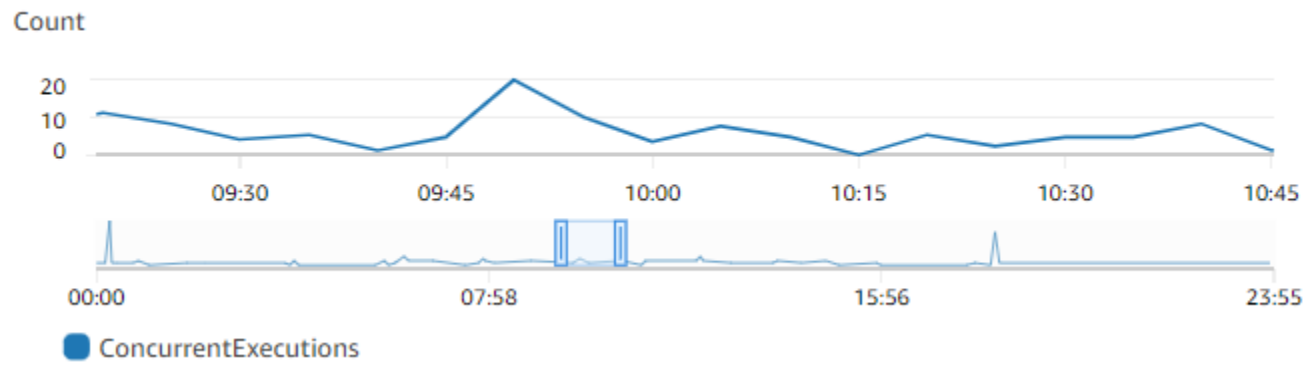
```
aws lambda put-provisioned-concurrency-config --function-name my-function \
  --qualifier BLUE \
  --provisioned-concurrent-executions 100
```

您應該會看到類似以下的輸出：

```
{
  "Requested ProvisionedConcurrentExecutions": 100,
  "Allocated ProvisionedConcurrentExecutions": 0,
  "Status": "IN_PROGRESS",
  "LastModified": "2023-01-21T11:30:00+0000"
}
```

準確估算函數所需的佈建並行

您可以使用 CloudWatch 指標來檢視任何作用中函數的並行指標。具體而言，ConcurrentExecutions 指標會顯示帳戶中函數的並行調用次數。



前一張圖表顯示此函數在任何特定時間平均可處理 5 到 10 個並行請求，峰值時段則為 20 個請求。假設帳戶中還有很多其他函數。如果此函數對您的應用程式很重要，而且每次調用都需要低延遲回應，請將佈建並行設為至少 20 個單位。

回想一下，您也可以使用以下公式計算並行：

```
Concurrency = (average requests per second) * (average request duration in seconds)
```

若要預估您需要多少並行，將每秒平均請求乘以平均請求持續時間 (以秒為單位)。您可以使用 `Invocation` 指標來預估每秒平均請求數，並使用 `Duration` 指標預估平均請求持續時間 (以秒為單位)。

設定佈建並行時，Lambda 建議在函數通常需要的並行量上額外加上 10% 的緩衝。例如，如果函數通常在 200 個並行請求達到峰值，可將佈建並行設定為 220 (200 個並行請求 + 10% = 220 佈建並行)。

使用佈建並行時最佳化函數程式碼

如果您使用的是佈建並行，請考慮調整函數程式碼結構，以最佳化低延遲。對於使用佈建並行的函數，Lambda 會在配置時執行任何初始化程式碼，例如載入程式庫和執行個體化用戶端。因此，建議將盡可能多的初始化移至主要函數處理常式以外，以避免在實際調用函數時影響延遲。相反地，在主要處理常式程式碼中初始化程式庫或執行個體化用戶端意味著您的函數必須在每次調用時執行此程式碼 (無論您是否使用佈建並行皆如此)。

對於隨需調用，每次函數冷啟動時，Lambda 可能需要重新執行初始化程式碼。針對此類函數，您可以選擇延遲特定功能的初始化，直至您的函數需要該功能。例如，請參閱下列 Lambda 處理常式的控制流程：

```
def handler(event, context):
    ...
```

```
    if ( some_condition ):
        // Initialize CLIENT_A to perform a task
    else:
        // Do nothing
```

在前面的範例中，開發人員在 `if` 陳述式中初始化 `CLIENT_A`，而不是在主要處理常式之外進行初始化。如此一來，Lambda 只會在滿足 `some_condition` 的情況下執行此程式碼。如果您在主要處理常式之外初始化 `CLIENT_A`，Lambda 會在每次冷啟動時執行該程式碼。這可能會增加整體延遲。

您可以新增 X-Ray 監控到函數，以測量 Lambda 向上擴展時的冷啟動。使用佈建並行的函式不會顯示冷啟動行為，因為執行環境在調用之前已準備妥當。不過，佈建並行必須套用至函式的特定版本或別名，而不是 $LATEST 版本。如果持續看到冷啟動行為，請確保調用的是已設定佈建並行的別名版本。

使用環境變數來檢視和控制佈建並行行為

您的函數可能會耗盡本身的所有佈建並行。Lambda 使用隨需執行個體來處理任何超出流量。為了判斷 Lambda 用於特定環境的初始化類型，請檢查 `AWS_LAMBDA_INITIALIZATION_TYPE` 環境變數的值。此變數有兩個可能的值：`provisioned-concurrency` 或 `on-demand`。`AWS_LAMBDA_INITIALIZATION_TYPE` 的值不可變，並且在整個環境的生命週期中保持不變。若要檢查函數程式碼中環境變數的值，請參閱擷取 Lambda 環境變數。

若您使用 .NET 8 執行時期，即使函式未使用佈建並行，仍可透過設定 `AWS_LAMBDA_DOTNET_PREJIT` 環境變數來改善函式的延遲問題。.NET 執行期會對第一次呼叫的每個程式庫採用延遲編譯和初始化。因此，Lambda 函數的第一次調用可能需要比後續調用更長的時間。若要減輕此問題，您可以針對 `AWS_LAMBDA_DOTNET_PREJIT` 選擇以下三個值之一：

- `ProvisionedConcurrency`：Lambda 會使用佈建並行，針對所有環境執行預先 JIT 編譯。這是預設值。
- `Always`：Lambda 會針對每個環境執行預先 JIT 編譯，即使函數未使用佈建並行。
- `Never`：Lambda 會針對所有環境停用預先 JIT 編譯。

了解使用佈建並行的記錄和計費行為

針對佈建並行環境，函數的初始化程式碼會在配置期間執行，且定期執行一次，因為 Lambda 會回收環境的執行個體。不過，即使環境執行個體從未處理請求，Lambda 也會針對初始化計費。佈建並行會持續執行，並與初始化和調用成本分開計費。如需詳細資訊，請參閱 AWS Lambda 定價。

當您使用佈建並行設定 Lambda 函式時，Lambda 會預先初始化該執行環境，使其在函式調用請求之前可供使用。每當環境初始化時，Lambda 都會以 JSON 日誌格式在 platform-initReport 日誌事件中記

錄函式的初始化持續時間欄位。若要查看此日誌事件，至少將 JSON 日誌層級設定為 INFO。您也可以使用遙測 API 來取用報告初始化持續時間欄位的平台事件。

使用 Application Auto Scaling 自動化佈建並行管理

Application Auto Scaling 可讓您依據排程或根據使用率來管理佈建並行。如果您的函數接收到可預測的流量模式，請使用排程擴展。如果您想要讓函數維持特定的使用率百分比，請使用目標追蹤擴展政策。

> **Note**
>
> 如果您使用 Application Auto Scaling 來管理函數的佈建並行，請確保先設定初始佈建並行值。如果您的函數沒有初始佈建並行值，Application Auto Scaling 可能無法正確處理函數擴展。

排程擴展

Application Auto Scaling 可讓您根據可預測的負載變化來設定自己的擴展排程。如需詳細資訊和範例，請參閱 Application Auto Scaling 使用指南中的 Application Auto Scaling 的排程擴展，以及 AWS 運算部落上的為週期性尖峰使用量排程 AWS Lambda 佈建並行。

目標追蹤

使用目標追蹤時，Application Auto Scaling 會會根據您定義擴展政策的方式，建立並管理一組 CloudWatch 警示。這些警示啟動時，Application Auto Scaling 會自動調整使用佈建並行配置的環境數量。對沒有可預測流量模式的應用程式使用目標追蹤。

若要使用目標追蹤擴展佈建並行，請使用 `RegisterScalableTarget` 和 `PutScalingPolicy` Application Auto Scaling API 作業。例如，如果您使用 AWS Command Line Interface (CLI)，請依照下列步驟執行：

1. 請將函數的別名註冊為擴展目標。以下範例會為函數 `my-function` 註冊別名 BLUE：

```
aws application-autoscaling register-scalable-target --service-namespace lambda \
    --resource-id function:my-function:BLUE --min-capacity 1 --max-capacity 100 \
    --scalable-dimension lambda:function:ProvisionedConcurrency
```

2. 將擴展政策套用至目標。以下範例會設定 Application Auto Scaling 來調整別名的佈建並行組態，將使用率維持在接近 70%，但您可以套用 10% 至 90% 之間的任意值。

```
aws application-autoscaling put-scaling-policy \
    --service-namespace lambda \
    --scalable-dimension lambda:function:ProvisionedConcurrency \
    --resource-id function:my-function:BLUE \
    --policy-name my-policy \
    --policy-type TargetTrackingScaling \
    --target-tracking-scaling-policy-configuration '{ "TargetValue":
 0.7, "PredefinedMetricSpecification": { "PredefinedMetricType":
 "LambdaProvisionedConcurrencyUtilization" }}'
```

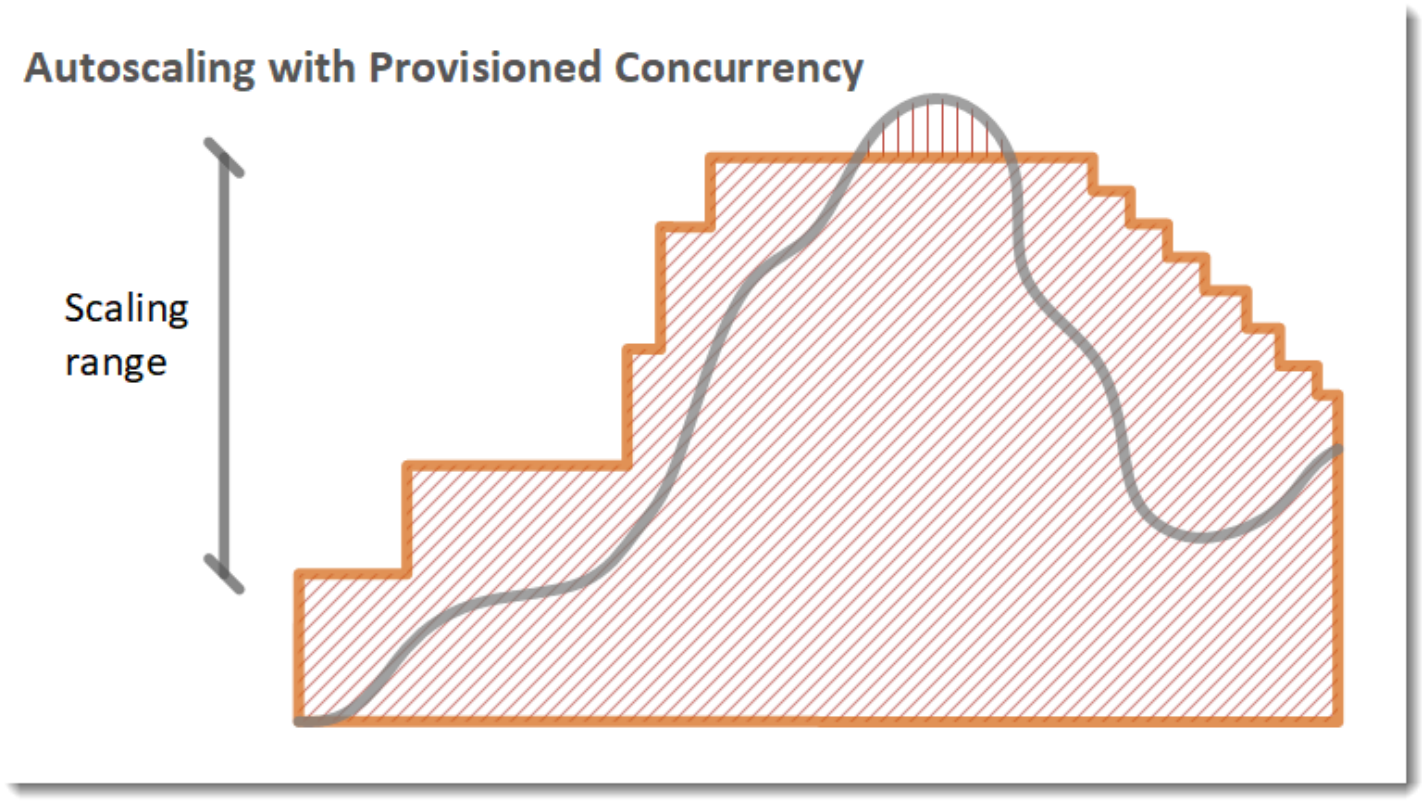
您應該會看到類似下面的輸出：

```
{
    "PolicyARN": "arn:aws:autoscaling:us-
east-2:123456789012:scalingPolicy:12266dbb-1524-xmpl-a64e-9a0a34b996fa:resource/lambda/
function:my-function:BLUE:policyName/my-policy",
    "Alarms": [
        {
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmHigh-aed0e274-
xmpl-40fe-8cba-2e78f000c0a7"
        },
        {
            "AlarmName": "TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66",
            "AlarmARN": "arn:aws:cloudwatch:us-
east-2:123456789012:alarm:TargetTracking-function:my-function:BLUE-AlarmLow-7e1a928e-
xmpl-4d2b-8c01-782321bc6f66"
        }
    ]
}
```

Application Auto Scaling 會在 CloudWatch 中建立兩個警示。第一個警示會在佈建並行的使用率不斷超過 70% 時觸發。發生這種情況時，Application Auto Scaling 會配置更多佈建並行來減少使用率。第二個警示會在使用率不斷低於 63% (70% 目標的 90%) 時觸發。發生這種情況時，Application Auto Scaling 會減少別名的佈建並行。

> **Note**
>
> Lambda 僅在函式處於作用中狀態且接收請求時，才會發出 `ProvisionedConcurrencyUtilization` 指標。在非作用期間，不會發出任何指標，而且自動擴展警示將進入 `INSUFFICIENT_DATA` 狀態。因此，應用程式自動擴展將無法調整函式的佈建並行。這可能導致非預期的計費。

在下列範例中，函數會根據使用率在佈建並行的最小和最大數量之間進行擴展。



圖例

- 函數執行個體
- 開啟請求
- 佈建並行

- ||||||||||||||||
  標準並行

當開啟的請求數量增加時，Application Auto Scaling 會大幅增加佈建並行，直到達到設定的最大值為止。在此之後，如果您尚未達到帳戶並行限制，則函數可以繼續按標準、未預留並行的方式擴展。當利用率下降且保持偏低時，Application Auto Scaling 會定期小幅減少佈建並行。

依預設，兩個 Application Auto Scaling 警示都會使用平均統計資料。經歷快速暴增流量的函數可能不會觸發這些警示。例如，假設您的 Lambda 函數執行速度很快 (即 20-100 毫秒)，而您的流量會快速暴增。在此情況下，請求數量將在暴增期間超過配置的佈建並行。不過，Application Auto Scaling 需要暴增負載維持至少 3 分鐘，才能佈建其他環境。或者，兩個 CloudWatch 警示都需要達到目標平均值的 3 個資料點，才能啟用自動擴展政策。如果您的函數遇到快速的流量爆增，使用最大統計資料而非平均統計資料，可以更有效地擴展佈建並行，將冷啟動降至最低。

如需目標追蹤擴展政策的詳細資訊，請參閱 Application Auto Scaling 的目標追蹤擴展政策。

Lambda 擴展行為

函數收到更多請求時，Lambda 會自動提高執行環境的數量來處理這些請求，直到您的帳戶達到並行配額為止。但是，為了防止因應突然爆發的流量而過度擴展，Lambda 限制了函數擴展的速度。此並行擴展速率是您帳戶中的函數可擴展以因應增加的請求的最高速率。(也就是說，Lambda 建立新執行環境的速度可以有多快。) 並行擴展速率與帳戶層級並行限制不同，後者是您的函數可用的並行總量。

並行擴展率

在每個 AWS 區域和每個函數中，您的並行擴展速率為每 10 秒 1,000 個執行環境執行個體 (或每隔 10 秒每秒 10,000 個請求)。換句話說，Lambda 每隔 10 秒就可以為每個函數配置最多 1,000 個額外執行環境的執行個體，或者每秒容納 10,000 個額外請求。

通常情況下，您不需要擔心此限制。對於大多數使用案例，Lambda 的擴展速率已足夠。

重要的是，並行擴展速率是函數層級限制。這意味著帳戶中的每個函數都可以獨立於其他函數進行擴展。

> **Note**
>
> 實際上，Lambda 會進行最佳嘗試在一段時間內持續重新填滿並行擴展速率，而不是每 10 秒重新填滿 1,000 個單位。

Lambda 不會累積並行擴展比率的未使用部分。這意味著在任何時刻，您的擴展速率最大始終為 1,000 個並行單位。例如，如果您沒有在 10 秒的間隔內使用任何可用的 1,000 個並行單位，則不會在接下來的 10 秒間隔內額外累積 1,000 個單位。在接下來的 10 秒間隔內，您的並行擴展速率仍然是 1,000。

只要您的函數持續接收到越來越多的請求，Lambda 就會以您可用的最快速率進行擴展，最高可達您帳戶的並行限制。您可以透過設定保留並行來限制個別函數可以使用的並行數量。當請求傳入的速度超過您函數可擴展的速度，或當您的函數達到最大並行時，額外請求會因為限流錯誤而請求失敗 (狀態碼為 429)。

監控並行

Lambda 會發出 Amazon CloudWatch 指標，協助您監控函數的並行。本主題將說明這些指標及其解譯方式。

章節

- 一般並行指標
- 佈建並行指標
- 使用 ClaimedAccountConcurrency 指標

一般並行指標

使用下列指標來監控 Lambda 函數的並行。每個指標的精細程度為 1 分鐘。

- `ConcurrentExecutions` – 在指定時間點的作用中並行調用數。Lambda 會針對所有函數、版本和別名發出此指標。對於 Lambda 主控台中的任何函數，Lambda 會在監控索引標籤的指標下以原生方式顯示 `ConcurrentExecutions` 的圖表。使用 MAX 檢視此指標。
- `UnreservedConcurrentExecutions` – 使用未預留並行的作用中並行調用數。Lambda 會針對區域中的所有函數發出此指標。使用 MAX 檢視此指標。
- `ClaimedAccountConcurrency` – 無法用於隨需調用的並行數量。`ClaimedAccountConcurrency` 等於 `UnreservedConcurrentExecutions` 加上配置並行的數量 (亦即預留並行總數加上佈建並行總數)。如果 `ClaimedAccountConcurrency` 超過帳戶並行限制，您可以請求提高帳戶並行上限。使用 MAX 檢視此指標。如需詳細資訊，請參閱使用 `ClaimedAccountConcurrency` 指標。

佈建並行指標

使用下列指標來監控使用佈建並行的 Lambda 函數。每個指標的精細程度為 1 分鐘。

- `ProvisionedConcurrentExecutions` – 目前使用佈建並行處理調用的執行環境執行個體數。Lambda 會針對已設定佈建並行的每個函數版本和別名發出此指標。使用 MAX 檢視此指標。

`ProvisionedConcurrentExecutions` 與您配置的佈建並行總數不同。例如，假設您將 100 個佈建並行單位配置給函數版本。在任何指定的分鐘內，如果在這 100 個執行環境中最多 50 個同時處理調用，則 MAX(`ProvisionedConcurrentExecutions`) 的值為 50。

- ProvisionedConcurrencyInvocations - Lambda 使用佈建並行調用函數程式碼的次數。Lambda 會針對已設定佈建並行的每個函數版本和別名發出此指標。使用 SUM 檢視此指標。

ProvisionedConcurrencyInvocations 與 ProvisionedConcurrentExecutions 不同，ProvisionedConcurrencyInvocations 計算調用總數，ProvisionedConcurrentExecutions 則計算作用中環境數。若要了解這種區別，請考慮以下案例：

在此範例中，假設您每分鐘收到 1 次調用，而且每次調用都需要 2 分鐘才能完成。每個橘色水平長條代表一個請求。假設您將 10 個佈建並行單位配置給此函數，這樣每個請求都會在佈建並行上執行。

在分鐘 0 和 1 之間，收到 Request 1。在分鐘 1 時，MAX(ProvisionedConcurrentExecutions) 的值為 1，因為在過去一分鐘內最多有 1 個執行環境處於作用中狀態。SUM(ProvisionedConcurrencyInvocations) 的值也是 1，因為在過去一分鐘內收到 1 個新請求。

在分鐘 1 和 2 之間，收到 Request 2，且 Request 1 繼續執行。在分鐘 2 時，MAX(ProvisionedConcurrentExecutions) 的值為 2，因為在過去一分鐘內最多有 2 個執行環境處於作用中狀態。不過，SUM(ProvisionedConcurrencyInvocations) 的值為 1，因為在過去一分鐘內只收到 1 個新請求。此指標行為會一直持續到範例結束為止。

- ProvisionedConcurrencySpilloverInvocations – 當所有佈建並行都在使用中時，Lambda 使用標準 (預留或未預留) 並行調用函數的次數。Lambda 會針對已設定佈建並行的每個函數版本和別名發出此指標。使用 SUM 檢視此指標。ProvisionedConcurrencyInvocations + ProvisionedConcurrencySpilloverInvocatio 值應等於函數調用總數 (即 Invocations 指標)。

ProvisionedConcurrencyUtilization – 使用中佈建並行百分比 (即 ProvisionedConcurrentExecutions 的值除以配置的佈建並行總數)。Lambda 會針對已設定佈建並行的每個函數版本和別名發出此指標。使用 MAX 檢視此指標。

例如，假設您將 100 個佈建並行單位佈建給函數版本。在任何指定的分鐘內，如果在這 100 個執行環境中最多 60 個同時處理調用，則 MAX(ProvisionedConcurrentExecutions) 的值為 60，MAX(ProvisionedConcurrencyUtilization) 的值為 0.6。

ProvisionedConcurrencySpilloverInvocations 的值偏高可能表示您需要為函數配置額外的佈建並行。或者，您可以設定 Application Auto Scaling，以根據預先定義的閾值來處理佈建並行的自動擴展。

相反地，ProvisionedConcurrencyUtilization 的值持續偏低可能表示您為函數配置過多佈建並行。

使用 ClaimedAccountConcurrency 指標

Lambda 會使用 ClaimedAccountConcurrency 指標來判斷您的帳戶可用於隨需調用的並行數量。Lambda 會使用以下公式計算 ClaimedAccountConcurrency：

```
ClaimedAccountConcurrency = UnreservedConcurrentExecutions + (allocated concurrency)
```

UnreservedConcurrentExecutions 是使用未預留並行的作用中並行調用數目。配置的並行是下列兩個部分的總和 (以「預留並行」取代 RC，以「佈建並行」取代 PC)：

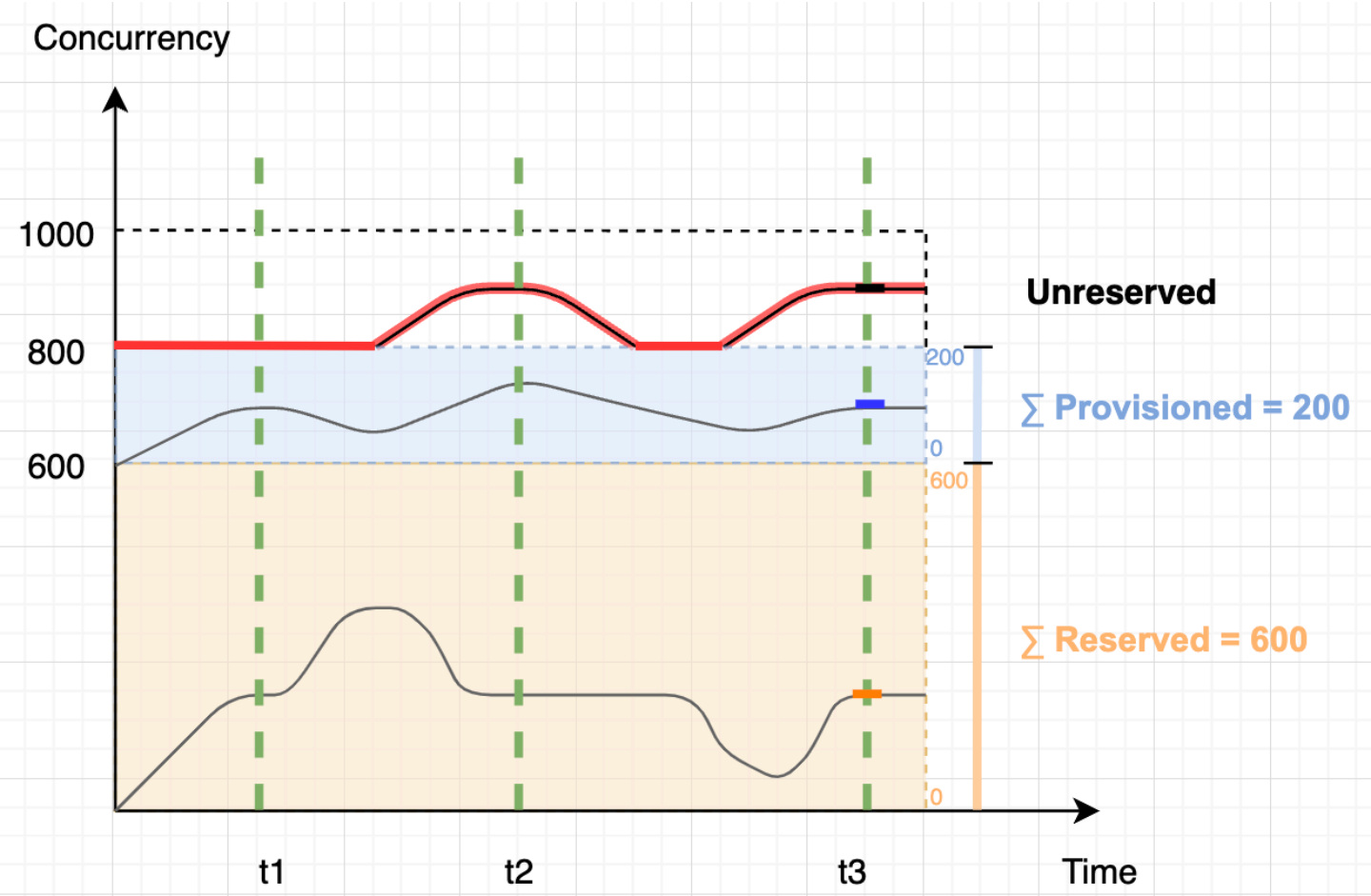
- 區域中所有函數的 RC 總數。
- 區域中使用 PC 的所有函數的 PC 總數，使用 RC 的函數除外。

> **Note**
>
> 您不能為一個函數配置多於 RC 的 PC。因此，一個函數的 RC 總是大於或等於它的 PC。對於同時具有 PC 和 RC 的函數，Lambda 在計算這些函數的配置並行比重時僅考慮 RC，即兩者當中的較大值。

Lambda 會使用 ClaimedAccountConcurrency 指標來判斷可用於隨需調用的並行數量，而不使用 ConcurrentExecutions。雖然 ConcurrentExecutions 指標對於追蹤作用中並行調用的數量很

有用，但並不總是反映您真正的並行可用性。這是因為 Lambda 也會考慮預留並行和佈建並行來確定可用性。

為了說明 ClaimedAccountConcurrency，請考慮一種情況，即您可以跨函數設定大量預留並行和佈建並行，但其中有很多未被使用。在下列範例中，假設您的帳戶並行限制為 1,000，而您的帳戶中有兩種主要的函數：function-orange 和 function-blue。您為 function-orange 配置 600 個單位的預留並行。您為 function-blue 配置 200 個單位的佈建並行。假設隨著時間推移，您要部署額外的函數並觀測以下流量模式：



在上一張圖表中，黑線表示隨時間推移的實際並行使用，紅線表示隨時間推移的 ClaimedAccountConcurrency 值。在這種情況下，儘管函數的實際並行利用率較低，但 ClaimedAccountConcurrency 始終至少為 800。這是因為您為 function-orange 和 function-blue 配置了 800 個單位總數的並行。從 Lambda 的角度來看，您已經「聲明」這些並行供使用，因此對於其他函數，您實際上只剩下 200 個單位的並行。

針對這個情況，在 ClaimedAccountConcurrency 公式中配置的並行是 800。然後，我們可以在圖表的不同點衍生 ClaimedAccountConcurrency 值。

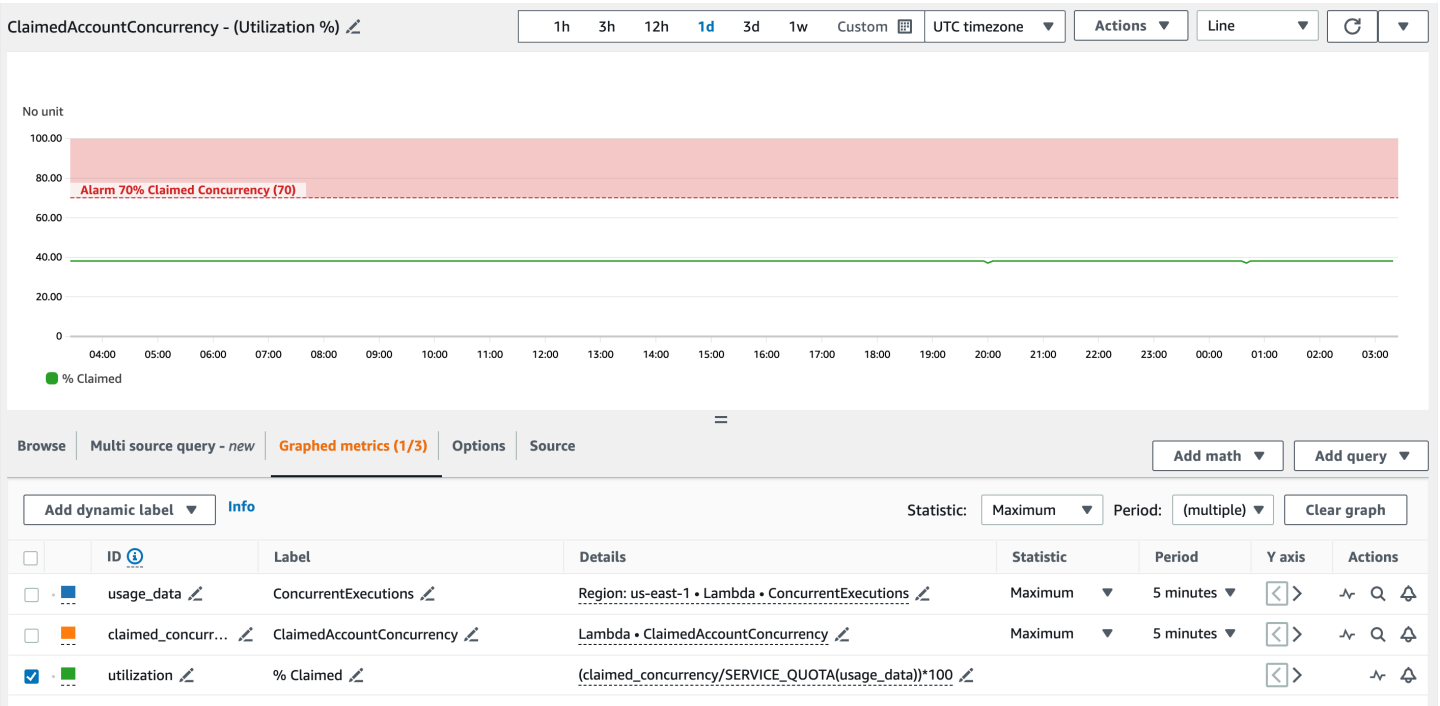
- 在 t1，ClaimedAccountConcurrency 為 800 (800 + 0 UnreservedConcurrentExecutions)。
- 在 t2，ClaimedAccountConcurrency 為 900 (800 + 100 UnreservedConcurrentExecutions)。
- 在 t3，ClaimedAccountConcurrency 還是 900 (800 + 100 UnreservedConcurrentExecutions)。

在 CloudWatch 中設定 **ClaimedAccountConcurrency** 指標

Lambda 會在 CloudWatch 中發出 ClaimedAccountConcurrency 指標。使用此指標和 SERVICE_QUOTA(ConcurrentExecutions) 的值取得有關您的帳戶的並行利用率百分比，如下方公式所示：

```
Utilization = (ClaimedAccountConcurrency/SERVICE_QUOTA(ConcurrentExecutions)) * 100%
```

下列螢幕截取畫面說明如何在 CloudWatch 中繪製此公式的圖形。綠色 claim_utilization 線代表此帳戶中的並行利用率，約為 40%：



先前的螢幕截取畫面還包含 CloudWatch 警示，該警示會在並行利用率超過 70% 時進入 ALARM 狀態。您可以使用 ClaimedAccountConcurrency 指標和類似的警示來主動確定您何時可能需要請求更高的帳戶並行上限。

使用 Node.js 建置 Lambda 函數

您可以在 AWS Lambda中執行具有 Node.js 的 JavaScript 程式碼。Lambda 提供用於執行程式碼來處理事件的 Node.js 執行期。您的程式碼會在包含 的環境中執行 適用於 JavaScript 的 AWS SDK，其中包含您管理之 AWS Identity and Access Management (IAM) 角色的登入資料。若要進一步了解 Node.js 執行時期隨附的 SDK 版本，請參閱 the section called “包含執行時期的 SDK 版本”。

Lambda 支援以下 Node.js 執行期。

Name	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Node.js 24	`nodejs24.x`	Amazon Linux 2023	2028 年 4 月 30 日	2028 年 6 月 1 日	2028 年 7 月 1 日
Node.js 22	`nodejs22.x`	Amazon Linux 2023	2027 年 4 月 30 日	2027 年 6 月 1 日	2027 年 7 月 1 日
Node.js 20	`nodejs20.x`	Amazon Linux 2023	2026 年 4 月 30 日	2026 年 8 月 31 日	2026 年 9 月 30 日

若要建立 Node.js 函數

1. 開啟 Lambda 主控台。
2. 選擇建立函數。
3. 進行下列設定：
   - 函數名稱：輸入函數名稱。
   - 執行時間：選擇 Node.js 24.x。
4. 選擇建立函數。

主控台將建立一個 Lambda 函數，其具有名為 `index.mjs` 的單一來源檔案。您可以使用內建的程式碼編輯器編輯該檔案並加入更多檔案。在 DEPLOY 區段中，選擇部署以更新函數的程式碼。然後，若要執行程式碼，請在測試事件區段中選擇建立測試事件。

index.mjs 檔案匯出名為 handler 的函數，它接受事件物件與內容物件。這就是在叫用函數時，Lambda 呼叫的處理常式函數。Node.js 函數執行期會從 Lambda 中取得調用事件並將它們傳遞至處理常式。在函式組態中，處理常式值為 index.handler。

當您儲存函數程式碼時，Lambda 主控台會建立 .zip 封存檔部署套件。當您在主控台之外開發函數程式碼 (使用 IDE) 時，您需要建立部署套件將您的程式碼上傳到 Lambda 函數。

除了傳遞調用事件外，函式執行期還會傳遞內容物件至處理常式。內容物件包含了有關調用、函式以及執行環境的額外資訊。更多詳細資訊將另由環境變數提供。

Lambda 函數隨附有 CloudWatch Logs 記錄群組。函數執行期會將每次調用的詳細資訊傳送至 CloudWatch Logs。它在調用期間會轉送您的函數輸出的任何記錄。如果您的函數傳回錯誤，Lambda 會對該錯誤進行格式化之後傳回給調用端。

主題

- 包含執行時期的 SDK 版本
- 使用保持連線保持 TCP 連線
- CA 憑證載入
- 實驗 Node.js 功能
- 在 Node.js 中定義 Lambda 函數處理常式
- 使用 .zip 封存檔部署 Node.js Lambda 函數
- 使用容器映像部署 Node.js Lambda 函數
- 對 Node.js Lambda 函數使用層
- 使用 Lambda 內容物件擷取 Node.js 函數資訊
- 記錄和監控 Node.js Lambda 函數
- 在 中檢測 Node.js 程式碼 AWS Lambda

包含執行時期的 SDK 版本

所有支援的 Lambda Node.js 執行時間都包含 適用於 JavaScript 的 AWS SDK v3 的特定次要版本，而非最新版本。執行時期中包含的特定次要版本取決於執行時期版本和您的 AWS 區域。若要查詢您目前所用執行時期中包含的 SDK 特定版本，請建立一個包含以下程式碼的 Lambda 函式。

Example index.mjs

```
import packageJson from '@aws-sdk/client-s3/package.json' with { type: 'json' };
```

```
export const handler = async () => ({ version: packageJson.version });
```

這會傳回下列格式的回應：

```
{
  "version": "3.632.0"
}
```

如需詳細資訊，請參閱在處理常式中使用適用於 JavaScript v3 的 SDK。

使用保持連線保持 TCP 連線

預設 Node.js HTTP/HTTPS 代理程式會為每個新的請求建立新的 TCP 連線。為避免建立新連線所產生的成本，所有支援的 Node.js 執行時期均預設啟用保持連線功能。保持連線可以減少使用 SDK 進行多次 API 呼叫的 Lambda 函數的請求次數。

若要停用保持連線，請參閱《AWS SDK for JavaScript 3.x 版本開發人員指南》中的在 Node.js 中透過保持連線重複使用連線。如需使用保持連線的詳細資訊，請參閱《 AWS 開發人員工具部落格》上的適用於 JavaScript 的模組化 AWS SDK 中的 HTTP 保持連線預設為開啟。

CA 憑證載入

對於 Node.js 18 以下的 Node.js 執行時期版本，Lambda 會自動載入 Amazon 專屬的 CA (憑證授權單位) 憑證，讓您更輕鬆地建立與其他 AWS 服務互動的函數。例如，Lambda 包含在 Amazon RDS 資料庫上安裝的驗證伺服器身分憑證所需的 Amazon RDS 憑證。此行為可能會在冷啟動期間產生效能影響。

從 Node.js 20 開始，在預設情況下，Lambda 不再載入額外的 CA 憑證。Node.js 20 執行期包含一個憑證檔案，其中包含所有 Amazon CA 憑證位於 `/var/runtime/ca-cert.pem`。若要從 Node.js 18 及更早版本的執行期還原相同的行為，請將 `NODE_EXTRA_CA_CERTS`環境變數設定為 `/var/runtime/ca-cert.pem`。

為了獲得最佳效能，我們建議您只將需要的憑證與部署套件搭配，並透過 `NODE_EXTRA_CA_CERTS` 環境變數載入憑證。憑證檔案應包含一或多個 PEM 格式的受信任根憑證或中繼 CA 憑證。例如，對於 RDS，請在程式碼旁包含所需的憑證做為 `certificates/rds.pem`。然後，藉由將 `NODE_EXTRA_CA_CERTS` 設定為 `/var/task/certificates/rds.pem` 載入憑證。

實驗 Node.js 功能

上游 Node.js 語言版本預設啟用部分實驗性功能。為確保執行時期穩定性與效能一致性，Lambda 會停用這些功能。下表列出了 Lambda 停用的實驗性功能。

實驗性功能	支援的 Node.js 版本	Lambda 套用的 Node.js 旗標	要重新啟用的 Lambda 旗標
支援在 ES 模組中使用 require 匯入模組	Node.js 20、Node.js 22	`--no-experimental-require-module`	`--experimental-require-module`
支援自動偵測 ES 模組與 CommonJS 模組的功能	Node.js 22	`--no-experimental-detect-module`	`--experimental-detect-module`

若要啟用已停用的實驗性功能，請在 `NODE_OPTIONS` 環境變數中設定重新啟用旗標。例如，若要啟用 ES 模組的 require 支援功能，請將 `NODE_OPTIONS` 設定為 `--experimental-require-module`。Lambda 偵測到此覆寫並移除了對應的停用旗標。

> **Important**
>
> 使用實驗性功能可能會導致不穩定性與效能問題。這些功能可能會在未來的 Node.js 版本中變更或移除。使用實驗性功能的函式不符合 Lambda 服務水準協議 (SLA) 或 AWS 支援的資格。

在 Node.js 中定義 Lambda 函數處理常式

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

本頁介紹了如何在 Node.js 中使用 Lambda 函式處理常式，包括專案設定選項、命名慣例及最佳實務。本頁還提供了一個 Node.js Lambda 函式範例，該函式會取得訂單的相關資訊、產生文字檔案收據，並將該檔案放入 Amazon Simple Storage Service (Amazon S3) 儲存貯體。如需編寫函數後如何部署函數的詳細資訊，請參閱the section called “部署 .zip 封存檔”或the section called “部署容器映像”。

主題

- 設定 Node.js 處理常式專案
- Node.js Lambda 函式程式碼範例
- CommonJS 和 ES 模組
- Node.js 初始化
- 處理常式命名慣例
- 定義和存取輸入事件物件
- Node.js 函式的有效處理常式模式
- 在處理常式中使用適用於 JavaScript v3 的 SDK
- 存取環境變數
- 使用全域狀態
- Node.js Lambda 函數的程式碼最佳實務

設定 Node.js 處理常式專案

初始化 Node.js Lambda 專案的方法有多種。例如，您可以使用 npm 建立標準的 Node.js 專案，建立 AWS SAM 應用程式，或建立 AWS CDK 應用程式。

使用 npm 建立專案：

```
npm init
```

此命令會初始化專案，並產生管理專案中繼資料和相依項的 `package.json` 檔案。

您的函式程式碼位於 `.js` 或 `.mjs` JavaScript 檔案中。在下列範例中，此檔案使用 ES 模組處理常式，因此我們將其命名為 `index.mjs`。Lambda 同時支援 ES 模組與 CommonJS 處理常式。如需詳細資訊，請參閱CommonJS 和 ES 模組。

一個典型的 Node.js Lambda 函式專案遵循以下一般結構：

```
/project-root
  ### index.mjs – Contains main handler
  ### package.json – Project metadata and dependencies
  ### package-lock.json – Dependency lock file
  ### node_modules/ – Installed dependencies
```

Node.js Lambda 函式程式碼範例

以下 Lambda 函式程式碼範例會取得訂單的相關資訊、產生文字檔案收據，並將該檔案放入 Amazon S3 儲存貯體。

Example index.mjs Lambda 函式

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';

// Initialize the S3 client outside the handler for reuse
const s3Client = new S3Client();

/**
 * Lambda handler for processing orders and storing receipts in S3.
 * @param {Object} event - Input event containing order details
 * @param {string} event.order_id - The unique identifier for the order
 * @param {number} event.amount - The order amount
 * @param {string} event.item - The item purchased
 * @returns {Promise<string>} Success message
 */
export const handler = async(event) => {
    try {
        // Access environment variables
        const bucketName = process.env.RECEIPT_BUCKET;
        if (!bucketName) {
            throw new Error('RECEIPT_BUCKET environment variable is not set');
        }

        // Create the receipt content and key destination
        const receiptContent = `OrderID: ${event.order_id}\nAmount: $
${event.amount.toFixed(2)}\nItem: ${event.item}`;
```

```
        const key = `receipts/${event.order_id}.txt`;

        // Upload the receipt to S3
        await uploadReceiptToS3(bucketName, key, receiptContent);

        console.log(`Successfully processed order ${event.order_id} and stored receipt
 in S3 bucket ${bucketName}`);
        return 'Success';
    } catch (error) {
        console.error(`Failed to process order: ${error.message}`);
        throw error;
    }
};

/**
 * Helper function to upload receipt to S3
 * @param {string} bucketName - The S3 bucket name
 * @param {string} key - The S3 object key
 * @param {string} receiptContent - The content to upload
 * @returns {Promise<void>}
 */
async function uploadReceiptToS3(bucketName, key, receiptContent) {
    try {
        const command = new PutObjectCommand({
            Bucket: bucketName,
            Key: key,
            Body: receiptContent
        });

        await s3Client.send(command);
    } catch (error) {
        throw new Error(`Failed to upload receipt to S3: ${error.message}`);
    }
}
```

此 `index.mjs` 檔案包含以下程式碼區段：

- `import` 區塊：此區塊用於納入 Lambda 函式所需的程式碼，例如 AWS SDK 用戶端。
- `const s3Client` 宣告：此宣告會在處理常式函式外部初始化 Amazon S3 用戶端。Lambda 會於初始化階段執行此程式碼，並且該用戶端會被保留下，在多次調用中重複使用。
- JSDoc 註解區塊：透過 JSDoc 註釋定義處理常式的輸入與輸出類型。

- `export const handler`：此為 Lambda 調用的主要處理常式函式。部署函式時，請為處理常式屬性指定 `index.handler`。`Handler` 屬性的值由檔案名稱與匯出的處理常式方法名稱組成，兩者以點號分隔。
- `uploadReceiptToS3` 函式：此為主要處理常式函式所參照的輔助函式。

若要讓此函式正常運作，其執行角色必須允許 `s3:PutObject` 動作。此外，請確保定義 `RECEIPT_BUCKET` 環境變數。成功調用後，Amazon S3 儲存貯體應包含收據檔案。

CommonJS 和 ES 模組

Node.js 支援兩個模組系統：CommonJS 和 ECMAScript 模組 (ES 模組）。Lambda 建議使用 ES 模組，因為它支援頂層等待，這可在執行環境初始化期間完成非同步任務。

Node.js 會將副檔名為 `.cjs` CommonJS 的檔案視為 CommonJS 模組，副`.mjs`檔名則表示 ES 模組。根據預設，Node.js 會將副檔名為 `.js` 的檔案視為 CommonJS 模組。您可以將 Node.js 設定為將`.js`檔案視為 ES 模組，方法是`module`在函數的 `package.json` 檔案中指定`type`為 。您可以在 Lambda 中設定 Node.js，透過將 `–experimental-detect-module`旗標新增至`NODE_OPTIONS`環境變數，自動偵測`.js`檔案是否應視為 CommonJS 或 ES 模組。如需詳細資訊，請參閱實驗 Node.js 功能。

下列範例顯示使用 ES 模組和 CommonJS 模組編寫的函數處理常式。本頁面上的其餘範例都使用 ES 模組。

ES module example

Example– ES 模組處理常式

```
const url = "https://aws.amazon.com/";

export const handler = async(event) => {
    try {
        const res = await fetch(url);
        console.info("status", res.status);
        return res.status;
    }
    catch (e) {
        console.error(e);
        return 500;
    }
};
```

CommonJS module example

Example– CommonJS 模組處理常式

```
const https = require("https");
let url = "https://aws.amazon.com/";

exports.handler = async function (event) {
  let statusCode;
  await new Promise(function (resolve, reject) {
    https.get(url, (res) => {
        statusCode = res.statusCode;
        resolve(statusCode);
      }).on("error", (e) => {
        reject(Error(e));
      });
  });
  console.log(statusCode);
  return statusCode;
};
```

Node.js 初始化

Node.js 使用非封鎖 I/O 模型，支援使用事件迴圈的高效率非同步操作。例如，如果 Node.js 進行網路呼叫，函數會繼續處理其他操作，而不會封鎖網路回應。收到網路回應時，會將其置於回呼佇列中。佇列中的任務會在目前任務完成時處理。

Lambda 建議使用頂層等待，以便在初始化期間完成執行環境初始化期間啟動的非同步任務。初始化期間未完成的非同步任務通常會在第一個函數叫用期間執行。這可能會導致非預期的行為或錯誤。例如，您的函數初始化可能會進行網路呼叫，從參數存放區擷取 AWS 參數。如果此任務未在初始化期間完成，則值在調用期間可能為 null。初始化和調用之間也可能有延遲，這可能會在時間敏感的操作中觸發錯誤。特別是，如果在初始化階段未完成呼叫， AWS 服務呼叫可能會依賴時間敏感的請求簽章，導致服務呼叫失敗。在初始化期間完成任務通常會改善冷啟動效能，並在使用佈建並行時先調用效能。如需詳細資訊，請參閱我們的部落格文章使用 Node.js ES 模組和頂層等待。 AWS Lambda

處理常式命名慣例

設定函式時，處理常式的設定值由檔案名稱與匯出的處理常式方法名稱組成，二者以點號分隔。主控台中建立的函數以及本指南中的範例，預設值都是 `index.handler`。這表示 `handler` 方法是透過 `index.js` 或 `index.mjs` 檔案匯出的。

如果要在主控台中使用不同檔案名稱或函數處理常式名稱建立函數，您必須編輯預設處理常式名稱。

變更函數處理常式名稱的方式 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇您的函數。
2. 選擇 程式碼 索引標籤。
3. 向下捲動至執行時間設定窗格，並選擇編輯。
4. 在處理常式中，輸入函數處理常式的新名稱。
5. 選擇儲存。

定義和存取輸入事件物件

JSON 是 Lambda 函數最常見的標準輸入格式。在此範例中，函數預期輸入類似以下內容：

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

在 Node.js 中開發 Lambda 函式時，可透過 JSDoc 註釋定義輸入事件的預期結構。在此範例中，我們在處理常式的 JSDoc 註解中定義輸入結構：

```
/**
 * Lambda handler for processing orders and storing receipts in S3.
 * @param {Object} event - Input event containing order details
 * @param {string} event.order_id - The unique identifier for the order
 * @param {number} event.amount - The order amount
 * @param {string} event.item - The item purchased
 * @returns {Promise<string>} Success message
 */
```

在 JSDoc 註解內定義這些類型之後，即可直接在程式碼中存取事件物件的欄位。例如，使用 `order_id` 可以從原始輸入擷取 `event.order_id` 的值。

Node.js 函式的有效處理常式模式

建議您使用 async/await 來宣告函式處理常式，而不是使用 callback。async/await 能以簡潔且易讀的方式編寫非同步程式碼，無需使用巢狀回呼或串連承諾。您可以透過非同步/等待模式撰寫讀起來像同步程式碼的程式碼，同時仍維持非同步和非封鎖的特性。

非同步函數處理常式 （ 建議 ）

async 關鍵字會將函數標記為非同步，且 await 關鍵字會暫停函數的執行，直到 Promise 獲得解決為止。處理常式接受下列引數：

- event：包含傳遞至函式的輸入資料。
- context：包含有關調用、函式以及執行環境的資訊。如需詳細資訊，請參閱使用 Lambda 內容物件擷取 Node.js 函數資訊。

以下所示為 async/await 模式的有效簽章：

```
export const handler = async (event) => { };
```

```
export const handler = async (event, context) => { };
```

同步函數處理常式

如果您的函數不執行任何非同步任務，您可以使用同步函數處理常式，使用以下其中一個函數簽章：

```
export const handler = (event) => { };
```

```
export const handler = (event, context) => { };
```

回應串流函數處理常式

Lambda 支援使用 Node.js 進行回應串流。回應串流函數處理常式使用awslambda.streamifyResponse()裝飾項目，並採用 3 個參數：responseStream、event和 context。函數簽章為：

```
export const handler = awslambda.streamifyResponse(async (event, responseStream,
 context) => { });
```

如需詳細資訊，請參閱Lambda 函數的回應串流。

以回呼為基礎的函數處理常式

> **Note**
>
> 以回呼為基礎的函數處理常式最多僅支援 Node.js 22。從 Node.js 24 開始，應使用非同步函數處理常式實作非同步任務。

以回呼為基礎的函數處理常式必須使用事件、內容和回呼引數。範例：

```
export const handler = (event, context, callback) => { };
```

回呼函式預期會接收一個 Error 與一個回應物件，其中回應物件必須可進行 JSON 序列化。該函式會一直執行，直至事件迴圈清空或函式逾時為止。直到完成所有的事件迴圈任務，回應才會傳送到叫用端。如果函式逾時，便會傳回錯誤。您可以透過設定 context.callbackWaitsForEmptyEventLoop 為「false (失敗)」來設定執行時間立即傳回回應。

Example- 具有 callback 的 HTTP 請求

以下範例函式檢查 URL 及傳回狀態碼給叫用端。

```
import https from "https";
let url = "https://aws.amazon.com/";

export const handler = (event, context, callback) => {
  https.get(url, (res) => {
    callback(null, res.statusCode);
  }).on("error", (e) => {
    callback(Error(e));
  });
};
```

在處理常式中使用適用於 JavaScript v3 的 SDK

通常，您將使用 Lambda 函數與其他 AWS 資源互動或進行更新。與這些資源互動的最簡單方法便是使用 適用於 JavaScript 的 AWS SDK。所有支援的 Lambda Node.js 執行時期都包含適用於 JavaScript 第 3 版的 SDK。不過，我們強烈建議您在部署套件中包含所需的 AWS SDK 用戶端。此舉能在日後

Lambda 執行時期更新時，最大限度實現回溯相容性。只有在您無法包含其他套件時（例如，在 AWS CloudFormation 範本中使用 Lambda 主控台程式碼編輯器或內嵌程式碼時），才依賴執行時間提供的 SDK。

若要將 SDK 相依項新增至函數，請針對您需要的特定 SDK 用戶端使用 `npm install` 命令。在範例程式碼中，我們使用的是 Amazon S3 用戶端。透過在包含 `package.json` 檔案的目錄中執行下列命令，即可新增這些相依項：

```
npm install @aws-sdk/client-s3
```

在函式程式碼中匯入所需的用戶端與命令，如範例函式所示：

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';
```

然後，初始化 Amazon S3 用戶端：

```
const s3Client = new S3Client();
```

在此範例中，我們在主要處理常式函式外部初始化 Amazon S3 用戶端，以免每次調用函式時都必須初始化該用戶端。初始化 SDK 用戶端之後，您就可以使用它來對該 AWS 服務進行 API 呼叫。此範例程式碼會呼叫 Amazon S3 PutObject API，如下所示：

```
const command = new PutObjectCommand({
    Bucket: bucketName,
    Key: key,
    Body: receiptContent
});
```

存取環境變數

在處理常式程式碼中，您可以使用 `process.env` 來引用任意環境變數。在此範例中，我們使用下列程式碼行來引用定義的 `RECEIPT_BUCKET` 環境變數：

```
// Access environment variables
const bucketName = process.env.RECEIPT_BUCKET;
if (!bucketName) {
    throw new Error('RECEIPT_BUCKET environment variable is not set');
}
```

使用全域狀態

在首次調用函式之前，Lambda 會在初始化階段執行靜態程式碼。在初始化期間建立的資源會在調用間隔期間保留在記憶體中，無需您在每次調用函式時都建立這些資源。

在範例程式碼中，S3 用戶端初始化程式碼位於主要處理常式外部。執行時期會在函式處理第一個事件之前初始化用戶端，且用戶端在所有調用中均可供重複使用。

Node.js Lambda 函數的程式碼最佳實務

建置 Lambda 函式時，請遵循下列準則：

- 區隔 Lambda 處理常式與您的核心邏輯。能允許您製作更多可測單位的函式。
- 控制函數部署套件內的相依性。 AWS Lambda 執行環境包含許多程式庫。對於 Node.js 和 Python 執行期，這些包括 AWS SDKs。若要啟用最新的一組功能與安全更新，Lambda 會定期更新這些程式庫。這些更新可能會為您的 Lambda 函數行為帶來細微的變更。若要完全掌控您函式所使用的相依性，請利用部署套件封裝您的所有相依性。
- 最小化依存項目的複雜性。偏好更簡易的框架，其可快速在執行環境啟動時載入。
- 將部署套件最小化至執行時間所必要的套件大小。這能減少您的部署套件被下載與呼叫前解壓縮的時間。

請利用執行環境重新使用來改看函式的效能。在函式處理常式之外初始化 SDK 用戶端和資料庫連線，並在本機快取 `/tmp` 目錄中的靜態資產。由您函式的相同執行個體處理的後續叫用可以重複使用這些資源。這可藉由減少函數執行時間來節省成本。

若要避免叫用間洩漏潛在資料，請不要使用執行環境來儲存使用者資料、事件，或其他牽涉安全性的資訊。如果您的函式依賴無法存放在處理常式內記憶體中的可變狀態，請考慮為每個使用者建立個別函式或個別函式版本。

使用 Keep-Alive 指令維持持續連線的狀態。Lambda 會隨著時間的推移清除閒置連線。叫用函數時嘗試重複使用閒置連線將導致連線錯誤。若要維護持續連線，請使用與執行階段相關聯的 keep-alive (保持啟用) 指令。如需範例，請參閱在 Node.js 中重複使用 Keep-Alive 的連線。

使用環境變數將操作參數傳遞給您的函數。例如，如果您正在寫入到 Amazon S3 儲存貯體，而非對您正在寫入的儲存貯體名稱進行硬式編碼，請將儲存貯體名稱設定為環境變數。

避免在 Lambda 函數中使用遞迴調用，其中函數會調用自己或啟動可能再次調用函數的程序。這會導致意外的函式呼叫量與升高的成本。若您看到意外的調用數量，當更新程式碼時，請立刻將函數的預留並行設為 `0`，以調節對函數的所有調用。

請勿在您的 Lambda 函數程式碼中使用未記錄的非公有 API。對於 AWS Lambda 受管執行期，Lambda 會定期將安全性和功能更新套用至 Lambda 的內部 APIs。這些內部 API 更新可能是向後不相容的，這會導致意外結果，例如若您的函數依賴於這些非公有 API，則叫用失敗。請參閱 API 參考查看公開可用 API 的清單。

撰寫等冪程式碼。為函數撰寫等冪程式碼可確保採用相同方式來處理重複事件。程式碼應正確驗證事件並正常處理重複的事件。如需詳細資訊，請參閱 How do I make my Lambda function idempotent? (如何讓 Lambda 函數等冪？)。

使用 .zip 封存檔部署 Node.js Lambda 函數

AWS Lambda 函數的程式碼包含 .js 或 .mjs 檔案，其中包含函數的處理常式程式碼，以及程式碼依賴的任何其他套件和模組。若要將此函數程式碼部署到 Lambda，您可以使用部署套件。此套件可以是 .zip 封存檔或容器映像。如需有關搭配使用容器映像與 Node.js 的詳細資訊，請參閱使用容器映像部署 Node.js Lambda 函數。

若要建立 .zip 封存檔的部署套件，您可以使用命令列工具的內建 .zip 封存檔公用程式，或任何其他 .zip 檔案公用程式 (例如 7zip)。以下各節顯示的範例假設您在 Linux 或 MacOS 環境中使用命令列 `zip` 工具。若要在 Windows 中使用相同命令，您可以安裝適用於 Linux 的 Windows 子系統，以取得 Ubuntu 和 Bash 的 Windows 整合版本。

請注意，Lambda 使用 POSIX 檔案許可，因此在建立 .zip 封存檔之前，您可能需要設定部署套件資料夾的許可。

主題

- Node.js 中的執行期相依項
- 建立不含相依項的 .zip 部署套件
- 建立含相依項的 .zip 部署套件
- 為相依項建立 Node.js 層
- 相依項搜尋路徑和含執行期程式庫
- 使用 .zip 檔案建立及更新 Node.js Lambda 函數

Node.js 中的執行期相依項

對於使用 Node.js 執行期的 Lambda 函數，相依項可以是任何 Node.js 模組。Node.js 執行期包含許多常見程式庫，以及 適用於 JavaScript 的 AWS SDK的某個版本。所有支援的 Node.js 執行時期皆包含第 3 版 SDK。若要使用第 2 版 SDK，須將該版 SDK 新增至 .zip 檔部署套件。若要查詢所用執行時期內含 SDK 的具體版本，請參閱the section called “包含執行時期的 SDK 版本”。

Lambda 會定期更新 Node.js 執行期中的 SDK 程式庫，以納入最新功能和安全升級。Lambda 也會將安全修補程式和更新套用至執行期中包含的其他程式庫。若要完全控制套件中的相依項，可以將任何包含執行期之相依項的偏好版本新增至部署套件。例如，如果想要針對 JavaScript 使用特定版本的 SDK，可以將其作為相依項包含在 .zip 檔案中。如需有關將包含執行期的相依項新增至 .zip 檔案的詳細資訊，請參閱 相依項搜尋路徑和含執行期程式庫。

在 AWS 共同責任模式下，您負責管理函數部署套件中的任何相依項。這包括套用更新和安全性修補程式。若要更新函數部署套件中的相依項，請先建立新的 .zip 檔案，然後將其上傳至 Lambda。如需詳細資訊，請參閱 建立含相依項的 .zip 部署套件 和 使用 .zip 檔案建立及更新 Node.js Lambda 函數。

建立不含相依項的 .zip 部署套件

如果除了 Lambda 執行期中包含的程式庫之外，函數程式碼沒有其他相依項，則 .zip 檔案只包含具有函數處理常式程式碼的 `index.js` 或 `index.mjs` 檔案。使用您慣用的 zip 公用程式建立 .zip 檔案，並將 `index.js` 或 `index.mjs` 檔案放在根目錄下。如果包含處理常式程式碼的檔案不在 .zip 檔案的根目錄下，則 Lambda 將無法執行程式碼。

若要了解如何部署 .zip 檔案以建立新的 Lambda 函數或更新現有函數，請參閱使用 .zip 檔案建立及更新 Node.js Lambda 函數。

建立含相依項的 .zip 部署套件

如果函數程式碼相依於 Lambda Node.js 執行期中不包含的套件或模組，則可以使用函數程式碼將這些相依項新增至 .zip 檔案，或使用 Lambda 層。本節中的指示說明如何在 .zip 部署套件中包含相依項。如需如何在層中包含相依項的指示，請參閱the section called “為相依項建立 Node.js 層”。

下列範例 CLI 命令會建立名為 `my_deployment_package.zip` 的 .zip 檔案，其中包含帶有函數處理常式程式碼及其相依項的 `index.js` 或 `index.mjs` 檔案。在此範例中，可以使用 npm 套件管理工具來安裝相依項。

建立部署套件

1. 導覽到包含 `index.js` 或 `index.mjs` 原始程式碼檔案的專案目錄。在此範例中，目錄名為 `my_function`。

   ```
   cd my_function
   ```

2. 使用 `npm install` 命令將函數的所需程式庫安裝在 `node_modules` 目錄中。在此範例中，將安裝 適用於 Node.js 的 AWS X-Ray SDK。

   ```
   npm install aws-xray-sdk
   ```

   這會建立如下資料夾結構：

   ```
   ~/my_function
   ```

```
### index.mjs
### node_modules
    ### async
    ### async-listener
    ### atomic-batcher
    ### aws-sdk
    ### aws-xray-sdk
    ### aws-xray-sdk-core
```

您也可以將自己建立的自訂模組新增至部署套件。在 node_modules 下建立一個帶有模組名稱的目錄，並將自訂編寫的套件儲存在此處。

3. 在根目錄建立包含專案資料夾內容的 .zip 檔案。使用 r (遞迴) 選項，以確保 zip 會壓縮子資料夾。

```
zip -r my_deployment_package.zip .
```

為相依項建立 Node.js 層

本節中的指示說明如何在層中包含相依項。如需如何在部署套件中包含相依項的指示，請參閱the section called “建立含相依項的 .zip 部署套件”。

將層新增至函數時，Lambda 會將層內容載入該執行環境的 /opt 目錄。在每一次 Lambda 執行期中，PATH 變數已包含 /opt 目錄中的特定資料夾路徑。為確保 Lambda 能夠擷取層內容，層 .zip 檔案應該將相依項置於以下任一資料夾路徑中：

- nodejs/node_modules
- nodejs/node18/node_modules (NODE_PATH)
- nodejs/node20/node_modules (NODE_PATH)
- nodejs/node22/node_modules (NODE_PATH)

例如，您的層 .zip 檔案結構可能如下所示：

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

此外，Lambda 會自動偵測 /opt/lib 目錄中的程式庫，以及 /opt/bin 目錄中的二進位檔案。若要確保 Lambda 正確找到您的層內容，您也可以建立結構如下的層：

```
custom-layer.zip
# lib
    | lib_1
    | lib_2
# bin
    | bin_1
    | bin_2
```

封裝層之後，請參閱the section called “建立和刪除層”及the section called “新增層”，完成層設定。

相依項搜尋路徑和含執行期程式庫

Node.js 執行期包含許多常見程式庫，以及 適用於 JavaScript 的 AWS SDK的某個版本。如果想要使用不同版本之包含執行期的程式庫，則可以透過將其與函數綁定在一起，或將其新增為部署套件中的相依項來執行此操作。例如，可以透過將其新增至 .zip 部署套件，來使用不同版本的 SDK。也可以將其包含在函數的 Lambda 層中。

當您在程式碼中使用 `import` 或 `require` 陳述式時，Node.js 執行期會在 `NODE_PATH` 路徑中搜尋目錄，直到找到模組為止。依預設，執行期搜尋的第一個位置是 .zip 部署套件解壓縮並掛載的目錄 (`/var/task`)。如果您在部署套件中納入含執行期程式庫的版本，則此版本的優先順序會高於執行期中包含的版本。部署套件中的相依項也優先於圖層中的相依項。

當您將相依項新增至圖層時，Lambda 會將其擷取到 `/opt/nodejs/nodexx/node_modules`，其中 `nodexx` 表示您所使用的執行期版本。在搜尋路徑中，此目錄的優先順序高於包含含執行期程式庫的目錄 (`/var/lang/lib/node_modules`)。因此，函數層中程式庫的優先順序高於執行期中包含的版本。

可以新增下列程式碼行，以查看 Lambda 函數的完整搜尋路徑。

```
console.log(process.env.NODE_PATH)
```

您也可以在 .zip 套件內的個別資料夾中新增相依項。例如，可以將自訂模組新增至名為 `common` 的 .zip 套件中的資料夾。解壓縮並掛載您的 .zip 套件時，此資料夾會放在 `/var/task` 目錄中。若要在程式碼中使用 .zip 部署套件中資料夾的相依性，請使用 `import { } from` 或 `const { } = require()` 陳述式，具體取決於您使用的是 CJS 還是 ESM 模組解析度。例如：

```
import { myModule } from './common'
```

如果將程式碼與 esbuild、rollup 或類似屬性綁定在一起，則函數使用的相依項會一起綁定在一個或多個檔案中。建議盡可能使用此方法來確定相依項。與將相依項新增至部署套件相比，綁定程式碼可提高效能，因為減少了 I/O 操作。

使用 .zip 檔案建立及更新 Node.js Lambda 函數

建立 .zip 部署套件後，您可以使用該套件建立新的 Lambda 函數或更新現有函數。您可以使用 Lambda 主控台、 AWS Command Line Interface和 Lambda API 部署 .zip 套件。您也可以使用 AWS Serverless Application Model (AWS SAM) 和 CloudFormation 建立並更新 Lambda 函數。

Lambda 的 .zip 部署套件大小上限為 250 MB (解壓縮)。請注意，此限制適用於您上傳的所有檔案 (包括任何 Lambda 層) 的大小總和。

Lambda 執行時間需有許可才能讀取部署套裝服務中的檔案。在 Linux 許可八進位標記法中，Lambda 需要 644 個許可 (rw-r--r--) 用於非可執行檔，以及 755 個許可 (rwxr-x) 用於目錄和可執行檔。

在 Linux 和 MacOS 中，使用 chmod 命令變更部署套件中檔案和目錄的檔案許可。例如，若要為非可執行檔提供正確的許可，請執行下列命令。

```
chmod 644 <filepath>
```

若要在 Windows 中變更檔案許可，請參閱 Microsoft Windows 文件的 Set, View, Change, or Remove Permissions on an Object。

> **Note**
>
> 如果您未授予 Lambda 存取部署套件中的目錄所需的許可，Lambda 會將這些目錄的許可設定為 755 (rwxr-xr-x)。

透過主控台使用 .zip 檔案建立及更新函數

若要建立新函數，您必須先在主控台中建立函數，然後上傳您的 .zip 封存檔。若要更新現有函數，請開啟函數的頁面，然後按照同樣的程序新增更新後的 .zip 檔案。

如果您的 .zip 檔案小於 50 MB，您可以透過直接從本機電腦上傳檔案來建立或更新函數。若 .zip 檔案大於 50 MB，您必須先將套件上傳至 Amazon S3 儲存貯體。如需如何使用 將檔案上傳至 Amazon S3 儲存貯體的說明 AWS 管理主控台，請參閱 Amazon S3 入門。若要使用 上傳檔案 AWS CLI，請參閱AWS CLI 《 使用者指南》中的移動物件。

> **Note**
>
> 不能變更現有函數的部署套件類型 (.zip 或容器映像)。例如，您不能轉換容器映像函數以使用 .zip 封存檔。您必須建立新的函數。

若要建立新的函數 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇建立函數。
2. 選擇 Author from scratch (從頭開始撰寫)。
3. 在基本資訊下，請執行下列動作：
   a. 在函數名稱中輸入函數名稱。
   b. 在執行期中選取要使用的執行期。
   c. (選用) 在架構中選擇要用於函數的指令集架構。預設架構值為 x86_64。請確定函數的 .zip 部署套件與您選取的指令集架構相容。
4. (選用) 在許可下，展開變更預設執行角色。您可建立新的執行角色，或使用現有的角色。
5. 選擇 Create function (建立函數)。Lambda 會使用您選擇的執行期建立一個基本的「Hello world」函數。

若要從本機電腦上傳 .zip 封存檔 (主控台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 .zip 檔案。
5. 若要上傳 .zip 檔案，請執行下列操作：
   a. 選擇上傳，然後在檔案選擇器中選取您的 .zip 檔案。
   b. 選擇 Open (開啟)。
   c. 選擇儲存。

若要從 Amazon S3 儲存貯體上傳 .zip 封存檔 (控制台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳新 .zip 檔案的函數。

2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 Amazon S3 位置。
5. 貼上 .zip 檔案的 Amazon S3 連結 URL，然後選擇儲存。

使用主控台程式碼編輯器更新 .zip 檔案函數

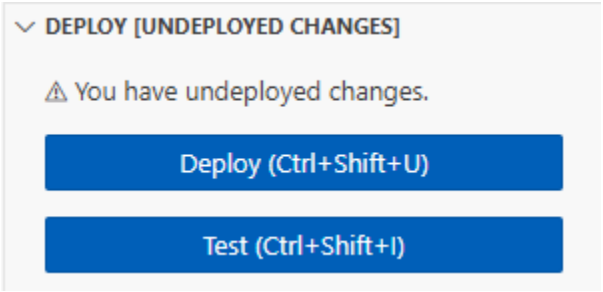
對於某些具有 .zip 部署套件的函數，您可以使用 Lambda 主控台的內建程式碼編輯器直接更新函數程式碼。若要使用此功能，您的函數必須符合下列條件：

- 您的函數必須使用其中一種轉譯語言執行期 (Python、Node.js 或 Ruby)
- 函數的部署套件必須小於 50 MB (未壓縮)。

具有容器映像部署套件之函數的函數程式碼無法直接在主控台中編輯。

若要使用主控台程式碼編輯器更新函數程式碼

1. 開啟 Lambda 主控台的函數頁面，然後選取您的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中，選取您的原始程式碼檔案，然後在整合式程式碼編輯器中加以編輯。
4. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



使用 建立和更新具有 .zip 檔案的函數 AWS CLI

您可以使用 AWS CLI 建立新函數，或使用 .zip 檔案更新現有函數。使用 create-function 和 update-function-code 命令來部署您的 .zip 套件。如果您的 .zip 檔案小於 50 MB，則可以從本機建置電腦的檔案位置上傳 .zip 套件。若檔案較大，則必須先從 Amazon S3 儲存貯體上傳 .zip 套件。如需如何使用將檔案上傳至 Amazon S3 儲存貯體的說明 AWS CLI，請參閱AWS CLI 《 使用者指南》中的移動物件。

> **Note**
>
> 如果您使用 從 Amazon S3 儲存貯體上傳 .zip 檔案 AWS CLI，儲存貯體必須與 AWS 區域 函數位於相同的 中。

若要搭配 使用 .zip 檔案建立新的函數 AWS CLI，您必須指定下列項目：

- 函數名稱 (`--function-name`)
- 函數的執行期 (`--runtime`)
- 函數執行角色的 Amazon Resource Name (ARN) (`--role`)
- 函數程式碼中處理常式方法的名稱 (`--handler`)

您也必須指定 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime nodejs24.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--code` 選項。您只需針對版本控制的物件使用 `S3ObjectVersion` 參數。

```
aws lambda create-function --function-name myFunction \
--runtime nodejs24.x --handler index.handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

若要使用 CLI 更新現有函數，您可以使用 `--function-name` 參數指定函數的名稱。您也必須指定要用來更新函數程式碼的 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--s3-bucket` 和 `--s3-key` 選項。您只需針對版本控制的物件使用 `--s3-object-version` 參數。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

透過 Lambda API 使用 .zip 檔案建立及更新函數

若要使用 .zip 封存檔建立及更新函數，請使用下列 API 操作：

- CreateFunction
- UpdateFunctionCode

使用 建立和更新具有 .zip 檔案的函數 AWS SAM

AWS Serverless Application Model (AWS SAM) 是一種工具組，可協助簡化建置和執行無伺服器應用程式的程序 AWS。您可以在 YAML 或 JSON 範本中定義應用程式的資源，並使用 AWS SAM 命令列界面 (AWS SAM CLI) 來建置、封裝和部署應用程式。當您從 AWS SAM 範本建置 Lambda 函數時，AWS SAM 會自動使用函數程式碼和您指定的任何相依性建立 .zip 部署套件或容器映像。若要進一步了解如何使用 AWS SAM 來建置和部署 Lambda 函數，請參閱《 AWS Serverless Application Model 開發人員指南》中的 入門 AWS SAM。

您也可以使用 AWS SAM 來建立使用現有 .zip 檔案封存的 Lambda 函數。若要使用 建立 Lambda 函數 AWS SAM，您可以將 .zip 檔案儲存在 Amazon S3 儲存貯體或建置機器的本機資料夾中。如需如何使用 將檔案上傳至 Amazon S3 儲存貯體的指示 AWS CLI，請參閱AWS CLI 《 使用者指南》中的移動物件。

在您的 AWS SAM 範本中，`AWS::Serverless::Function`資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `CodeUri`：設定為函數程式碼的 Amazon S3 URI、本機資料夾的路徑或 FunctionCode 物件
- `Runtime`：設定為所選執行期

使用 時 AWS SAM，如果您的 .zip 檔案大於 50MB，則不需要先將其上傳至 Amazon S3 儲存貯體。AWS SAM 可以從本機建置機器的位置上傳最大允許大小為 250MB （解壓縮） 的 .zip 套件。

若要進一步了解如何在 中使用 .zip 檔案部署函數 AWS SAM，請參閱《 AWS SAM 開發人員指南》中的 AWS::Serverless::Function。

使用 建立和更新具有 .zip 檔案的函數 CloudFormation

您可以使用 CloudFormation 建立使用 .zip 檔案封存的 Lambda 函數。若要使用 .zip 檔案建立 Lambda 函數，您必須先將檔案上傳至 Amazon S3 儲存貯體。如需如何使用 將檔案上傳至 Amazon S3 儲存貯體的說明 AWS CLI，請參閱《 使用者指南》中的移動物件。 AWS CLI

在您的 CloudFormation 範本中，`AWS::Lambda::Function`資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `Code`：在 `S3Bucket` 和 `S3Key` 欄位中輸入 Amazon S3 儲存貯體名稱和 .zip 檔案名稱。
- `Runtime`：設定為所選執行期

CloudFormation 產生的 .zip 檔案不能超過 4MB。若要進一步了解如何在 中使用 .zip 檔案部署函數 CloudFormation，請參閱CloudFormation 《 使用者指南》中的 AWS::Lambda::Function。

使用容器映像部署 Node.js Lambda 函數

您可以透過三種方式為 Node.js Lambda 函數建置容器映像：

- 使用 Node.js AWS 的基礎映像

  AWS 基礎映像會預先載入語言執行期、用來管理 Lambda 與函數程式碼之間互動的執行期界面用戶端，以及用於本機測試的執行期界面模擬器。

- 使用僅限 AWS 作業系統的基礎映像

  AWS 僅限作業系統的基礎映像包含 Amazon Linux 發行版本和執行時間界面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Node.js 的執行期介面用戶端。

- 使用非AWS 基礎映像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Node.js 的執行期介面用戶端。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

本頁面會說明如何為 Lambda 建置、測試和部署容器映像。

主題

- AWS Node.js 的基礎映像
- 使用 Node.js AWS 的基礎映像
- 透過執行期介面用戶端使用替代基礎映像

AWS Node.js 的基礎映像

AWS 為 Node.js 提供下列基礎映像：

Tags (標籤)	執行期	作業系統	Dockerfile	棄用
24	Node.js 24	Amazon Linux 2023	GitHub 上 Node.js 24 的 Dockerfile	2028 年 4 月 30 日
22	Node.js 22	Amazon Linux 2023	Dockerfile for Node.js 22 on GitHub	2027 年 4 月 30 日
20	Node.js 20	Amazon Linux 2023	Dockerfile for Node.js 20 on GitHub	2026 年 4 月 30 日

Amazon ECR 儲存庫：gallery.ecr.aws/lambda/nodejs

Node.js 20 和更新版本的基礎映像是以 Amazon Linux 2023 最小容器映像為基礎。舊版基礎映像使用 Amazon Linux 2。與 Amazon Linux 2 相比，AL2023 具有多項優點，包括更小的部署足跡和更新版本的程式庫，如 `glibc`。

以 AL2023 為基礎的映像使用 `microdnf` (符號連結為 `dnf`) 而不是 `yum` 作為套件管理工具，後者是 Amazon Linux 2 中的預設套件管理工具。`microdnf` 是 `dnf` 的獨立實作。對於以 AL2023 為基礎的映像中包含的套件清單，請參閱 Comparing packages installed on Amazon Linux 2023 Container Images 中的 Minimal Container 欄。如需 AL2023 和 Amazon Linux 2 之間差異的詳細資訊，請參閱 AWS 運算部落格上的 Introducing the Amazon Linux 2023 runtime for AWS Lambda。

> **Note**
>
> 若要在本機執行AL2023-based映像，包括搭配 AWS Serverless Application Model (AWS SAM)，您必須使用 Docker 20.10.10 版或更新版本。

使用 Node.js AWS 的基礎映像

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- Docker (最低版本 25.0.0)

- Docker buildx 外掛程式。
- Node.js

從基礎映像建立映像

從 Node.js AWS 的基礎映像建立容器映像

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir example
cd example
```

2. 使用 npm 建立新 Node.js 專案。若要接受互動體驗中提供的預設選項，請按下 Enter。

```
npm init
```

3. 建立稱為 index.js 的新檔案。您可以將下列範例函數程式碼新增至檔案進行測試，或使用您自己的函數程式碼。

   Example CommonJS 處理常式

```
exports.handler = async (event) => {
    const response = {
        statusCode: 200,
        body: JSON.stringify('Hello from Lambda!'),
    };
    return response;
};
```

4. 如果您的函數依賴 以外的程式庫 適用於 JavaScript 的 AWS SDK，請使用 npm 將它們新增至您的套件。
5. 建立包含下列組態的新 Dockerfile。
   - 將 FROM 屬性設定為基礎映像的 URI。
   - 使用 COPY 命令將函數程式碼和執行時期相依項複製到 {LAMBDA_TASK_ROOT}，一個 Lambda 定義的環境變數。
   - 將 CMD 引數設定為 Lambda 函數處理常式。

請注意，範例 Dockerfile 不包含 USER 指令。當您將容器映像部署到 Lambda 時，Lambda 會自動定義一個具有最低權限許可的預設 Linux 使用者。這與標準 Docker 行為不同，後者會在未提供 USER 指令時預設為 root 使用者。

Example Dockerfile

```
FROM public.ecr.aws/lambda/nodejs:22

# Copy function code
COPY index.js ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "index.handler" ]
```

6. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。若要使映像與 Lambda 相容，必須使用 --provenance=false 選項。

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

(選用) 在本機測試映像

1. 使用 docker run 命令啟動 Docker 影像。在此範例中，docker-image 為映像名稱，test 為標籤。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將`--platform linux/arm64`選項改用選項。`--platform linux/amd64`

2. 從新的終端機視窗，將事件張貼至本機端點。

   Linux/macOS

   在 Linux 或 macOS 中，執行下列 `curl` 命令：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   在 PowerShell 中，執行下列 `Invoke-WebRequest` 命令：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

3. 取得容器 ID。

   ```
   docker ps
   ```

4. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將 `--region`值設定為您要 AWS 區域 在其中建立 Amazon ECR 儲存庫的 。
   - `111122223333` 以您的 AWS 帳戶 ID 取代 。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於相同的 AWS 區域 中。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
```

```
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - `docker-image:test` 為 Docker 映像檔的名稱和標籤。這是您在 `docker build` 命令中指定的映像名稱和標籤。
   - 將 <ECRrepositoryUri> 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像來建立函數。如需詳細資訊，請參閱Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 會將映像標籤解析為特定映像摘要。這表示如果您將用來部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。

若要將新映像部署至相同的 Lambda 函數，必須使用 update-function-code 命令，即使 Amazon ECR 中的映像標籤保持不變亦如此。在以下範例中，`--publish` 選項會使用更新的容器映像來建立新的函數版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

透過執行期介面用戶端使用替代基礎映像

如果您使用僅限作業系統的基礎映像或替代的基礎映像，則必須在映像中加入執行期介面用戶端。執行期介面用戶端會讓您擴充 Runtime API，管理 Lambda 與函數程式碼之間的互動。

使用 npm 套件管理員安裝 Node.js 執行期界面用戶端：

```
npm install aws-lambda-ric
```

您還可以從 GitHub 下載 Node.js 執行時間界面用戶端。

下列範例示範如何使用非AWS 基礎映像建置 Node.js 的容器映像。範例 Dockerfile 使用 bookworm 基礎映像。Dockerfile 包含執行期界面用戶端。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- Docker (最低版本 25.0.0)
- Docker buildx 外掛程式。
- Node.js

使用替代基礎映像建立映像

從非AWS 基礎映像建立容器映像

1. 建立專案的目錄，然後切換至該目錄。

   ```
   mkdir example
   cd example
   ```

2. 使用 npm 建立新 Node.js 專案。若要接受互動體驗中提供的預設選項，請按下 Enter。

   ```
   npm init
   ```

3. 建立稱為 index.js 的新檔案。您可以將下列範例函數程式碼新增至檔案進行測試，或使用您自己的函數程式碼。

   Example CommonJS 處理常式

   ```
   exports.handler = async (event) => {
       const response = {
           statusCode: 200,
           body: JSON.stringify('Hello from Lambda!'),
       };
       return response;
   ```

```
};
```

4. 建立新的 Dockerfile。下列 Dockerfile 使用 bookworm 基礎映像，而非 AWS 基礎映像。Dockerfile 包含執行期介面用戶端，可讓映像與 Lambda 相容。Dockerfile 使用多階段建置。第一階段會建立組建映像，這是安裝函數相依項的標準 Node.js 環境。第二階段會建立更細微的映像，包含函數程式碼及其相依項。這會減少最終映像的大小。

   - 將 FROM 屬性設為基礎映像識別符。
   - 使用 COPY 命令複製函數程式碼和執行期相依項。
   - 將 ENTRYPOINT 設為您希望 Docker 容器在啟動時執行的模組。在此案例中，模組是執行期界面用戶端。
   - 將 CMD 引數設定為 Lambda 函數處理常式。

   請注意，範例 Dockerfile 不包含 USER 指令。當您將容器映像部署到 Lambda 時，Lambda 會自動定義一個具有最低權限許可的預設 Linux 使用者。這與標準 Docker 行為不同，後者會在未提供 USER 指令時預設為 root 使用者。

   Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM node:20-bookworm as build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Install build dependencies
RUN apt-get update && \
    apt-get install -y \
    g++ \
    make \
    cmake \
    unzip \
    libcurl4-openssl-dev

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}
```

```
WORKDIR ${FUNCTION_DIR}

# Install Node.js dependencies
RUN npm install

# Install the runtime interface client
RUN npm install aws-lambda-ric

# Grab a fresh slim copy of the image to reduce the final size
FROM node:20-bookworm-slim

# Required for Node runtimes which use npm@8.6.0+ because
# by default npm writes logs under /home/.npm and Lambda fs is read-only
ENV NPM_CONFIG_CACHE=/tmp/.npm

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT ["/usr/local/bin/npx", "aws-lambda-ric"]
# Pass the name of the function handler as an argument to the runtime
CMD ["index.handler"]
```

5. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 `docker-image` 並為其提供 `test` 標籤。若要使映像與 Lambda 相容，必須使用 `--provenance=false` 選項。

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> 此命令會指定 `--platform linux/amd64` 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 `--platform linux/arm64` 選項。

(選用) 在本機測試映像

使用 執行期界面模擬器 以在本機測試映像。您可以將模擬器建置到映像中，也可以使用以下步驟，將其安裝在本機電腦。

若要在本機電腦上安裝並執行執行期介面模擬器

1. 在您的專案目錄中執行以下命令，從 GitHub 下載執行期介面模擬器 (x86-64 架構)，並安裝在本機電腦上。

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   若要安裝 arm64 模擬器，請將上一個命令中的 GitHub 儲存庫 URL 替換為以下內容：

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   若要安裝 arm64 模擬器，請將 `$downloadLink` 更換為下列項目：

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

2. 使用 docker run 命令啟動 Docker 影像。注意下列事項：

- docker-image 是映像名稱，而 test 是標籤。
- /usr/local/bin/npx aws-lambda-ric index.handler 是 Dockerfile 中的 ENTRYPOINT，後面接著 CMD。

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /usr/local/bin/npx aws-lambda-ric index.handler
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/local/bin/npx aws-lambda-ric index.handler
```

此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將--platform linux/*arm64*選項改用選項。--platform linux/*amd64*

3. 將事件張貼至本機端點。

Linux/macOS

在 Linux 或 macOS 中，執行下列 curl 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在 PowerShell 中，執行下列 `Invoke-WebRequest` 命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 取得容器 ID。

```
docker ps
```

5. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。

   - 將 `--region`值設定為您要 AWS 區域 在其中建立 Amazon ECR 儲存庫的 。
   - 111122223333 以您的 AWS 帳戶 ID 取代 。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於相同的 AWS 區域 中。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - `docker-image:test` 為 Docker 映像檔的名稱和標籤。這是您在 `docker build` 命令中指定的映像名稱和標籤。

- 將 <ECRrepositoryUri> 替換為複製的 repositoryUri。確保在 URI 的末尾包含 :latest。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 :latest。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 ImageUri，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 :latest。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像來建立函數。如需詳細資訊，請參閱Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 會將映像標籤解析為特定映像摘要。這表示如果您將用來部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。

若要將新映像部署至相同的 Lambda 函數，必須使用 update-function-code 命令，即使 Amazon ECR 中的映像標籤保持不變亦如此。在以下範例中，`--publish` 選項會使用更新的容器映像來建立新的函數版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

對 Node.js Lambda 函數使用層

使用 Lambda 層封裝要在多個函式之間重複使用的程式碼與相依項。層通常具備程式庫相依性、自訂執行期或組態檔案。建立層包含三個一般步驟：

1. 封裝層內容。這表示建立 .zip 封存檔，其中包含您要在函數中使用的相依項。
2. 在 Lambda 中建立層。
3. 將層新增至函數中。

主題

- 封裝層內容
- 在 Lambda 中建立層
- 將層新增至函式
- 應用程式範例

封裝層內容

若要建立層，請將套件封裝成符合以下要求的 .zip 壓縮檔：

- 使用計劃用於 Lambda 函式的同一個 Node.js 版本來建置層。例如，如果您使用 Node.js 24 建置 layer，請使用 Node.js 24 執行時間做為函數。
- 層的 .zip 檔案必須使用下列任一目錄結構：
  - `nodejs/node_modules`
  - `nodejs/node`*X*`/node_modules` (其中 *X* 為 Node.js 版本，例如 `node22`)

  如需詳細資訊，請參閱每個 Lambda 執行時間的層路徑。
- 層中的套件必須與 Linux 相容。Lambda 函式會在 Amazon Linux 上執行。

您可以建立包含透過 npm 安裝的第三方 Node.js 程式庫 (如 `axios` 或 `lodash`) 的層，或您自訂的 JavaScript 模組的層。

第三方相依性

使用 npm 套件建立層

1. 建立必要的目錄結構，並將套件直接安裝至其中：

```
mkdir -p nodejs
npm install --prefix nodejs lodash axios
```

此命令會將套件直接安裝至 `nodejs/node_modules` 目錄，該目錄結構為 Lambda 所需的格式。

> **Note**
>
> 對於具有原生相依項或二進位元件 (例如 sharp 或 Bcrypt) 的套件，請確保其與 Lambda Linux 環境和函式的架構相容。您可能需要使用 `--platform` 旗標：
>
> ```
> npm install --prefix nodejs --platform=linux --arch=x64 sharp
> ```
>
> 對於更複雜的原生相依項，您可能需要在符合 Lambda 執行時期的 Linux 環境中進行編譯。在這種情況下，您可以使用 Docker。

2. 壓縮層內容：

Linux/macOS

```
zip -r layer.zip nodejs/
```

PowerShell

```
Compress-Archive -Path .\nodejs -DestinationPath .\layer.zip
```

.zip 檔案的目錄結構應如下所示：

```
nodejs/
### package.json
### package-lock.json
### node_modules/
    ### lodash/
    ### axios/
    ### (dependencies of the other packages)
```

> **Note**
>
> - 請確保 .zip 檔案在根層級包含 `nodejs` 目錄，且該目錄內含 `node_modules` 子目錄。此結構可確保 Lambda 能順利找到並匯入套件。
> - npm 將 `nodejs/` 目錄中的 `package.json` 和 `package-lock.json` 檔案用於相依項管理，但 Lambda 實現層功能時並不需要這些檔案。每個已安裝的套件皆包含其自身的 `package.json` 檔案，該檔案定義了 Lambda 匯入套件的方式。

自訂 JavaScript 模組

使用自訂的程式碼建立層

1. 建立層所需的目錄結構：

```
mkdir -p nodejs/node_modules/validator
cd nodejs/node_modules/validator
```

2. 為自訂模組建立 `package.json` 檔案，定義其匯入的方式：

   Example nodejs/node_modules/validator/package.json

```
{
  "name": "validator",
  "version": "1.0.0",
  "type": "module",
  "main": "index.mjs"
}
```

3. 建立 JavaScript 模組檔案：

   Example nodejs/node_modules/validator/index.mjs

```
export function validateOrder(orderData) {
  // Validates an order and returns formatted data
  const requiredFields = ['productId', 'quantity'];

  // Check required fields
  const missingFields = requiredFields.filter(field => !(field in orderData));
  if (missingFields.length > 0) {
```

```
    throw new Error(`Missing required fields: ${missingFields.join(', ')}`);
  }

  // Validate quantity
  const quantity = orderData.quantity;
  if (!Number.isInteger(quantity) || quantity < 1) {
    throw new Error('Quantity must be a positive integer');
  }

  // Format and return the validated data
  return {
    productId: String(orderData.productId),
    quantity: quantity,
    shippingPriority: orderData.priority || 'standard'
  };
}

export function formatResponse(statusCode, body) {
  // Formats the API response
  return {
    statusCode: statusCode,
    body: JSON.stringify(body)
  };
}
```

4. 壓縮層內容：

   Linux/macOS

   ```
   zip -r layer.zip nodejs/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\nodejs -DestinationPath .\layer.zip
   ```

   .zip 檔案的目錄結構應如下所示：

   ```
   nodejs/
   ### node_modules/
       ### validator/
           ### package.json
   ```

```
        ### index.mjs
```

5. 在您的函式中，匯入並使用模組。範例：

```
import { validateOrder, formatResponse } from 'validator';

export const handler = async (event) => {
  try {
    // Parse the order data from the event body
    const orderData = JSON.parse(event.body || '{}');

    // Validate and format the order
    const validatedOrder = validateOrder(orderData);

    return formatResponse(200, {
      message: 'Order validated successfully',
      order: validatedOrder
    });
  } catch (error) {
    if (error instanceof Error && error.message.includes('Missing required
 fields')) {
      return formatResponse(400, {
        error: error.message
      });
    }

    return formatResponse(500, {
      error: 'Internal server error'
    });
  }
};
```

您可以使用下列測試事件來調用函式：

```
{
    "body": "{\"productId\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

預期回應：

```
{
```

```
  "statusCode": 200,
  "body": "{\"message\":\"Order validated successfully\",\"order\":{\"productId\":
\"ABC123\",\"quantity\":2,\"shippingPriority\":\"express\"}}"
}
```

在 Lambda 中建立層

您可以使用 AWS CLI 或 Lambda 主控台發佈 layer。

AWS CLI

執行 publish-layer-version AWS CLI 命令來建立 Lambda layer：

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes nodejs24.x
```

相容的執行時期參數為選用參數。指定此參數後，Lambda 會據此在 Lambda 主控台中篩選層。

Console

建立圖層 (主控台)

1. 開啟 Lambda 主控台中的 層頁面。
2. 選擇 建立圖層。
3. 選擇上傳 .zip 檔案，然後上傳先前建立的 .zip 壓縮檔。
4. (選用) 在相容的執行時期欄位中，選擇與用於建置層的 Node.js 版本相對應的 Node.js 執行時期。
5. 選擇建立。

將層新增至函式

AWS CLI

若要將 layer 連接至函數，請執行 update-function-configuration AWS CLI 命令。對於 `--layers` 參數，請使用層 ARN。ARN 必須指定版本 (例如 `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`)。如需詳細資訊，請參閱層和層的版本。

```
aws lambda update-function-configuration --function-name my-function --cli-binary-
format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-
layer:1"
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

Console

將層新增至函式

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函式。
3. 向下捲動至層區段，然後選擇新增層。
4. 在選擇層欄位中，選取自訂層，然後選擇要使用的層。

> **Note**
>
> 如果未在建立層時新增相容的執行時期，此處不會列出層。您可以改為指定層 ARN。

5. 選擇新增。

應用程式範例

如需如何使用 Lambda 層的更多範例，請參閱《 AWS Lambda 開發人員指南 GitHub 儲存庫》中的 layer-nodejs 範例應用程式。此應用程式包含一個內含 lodash 程式庫的層。建立層之後，您可以部署並調用相應的函式，驗證層是否如預期那樣運作。

使用 Lambda 內容物件擷取 Node.js 函數資訊

當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。此物件提供的方法和各項屬性提供了有關調用、函式以及執行環境的資訊。

內容方法

- `getRemainingTimeInMillis()` - 傳回執行逾時前剩餘的毫秒數。

內容屬性

- `functionName` – Lambda 函數的名稱。
- `functionVersion` – 函數的版本。
- `invokedFunctionArn` - 用於調用此函數的 Amazon Resource Name (ARN)。指出調用者是否指定版本號或別名。
- `memoryLimitInMB` - 分配給函數的記憶體數量。
- `awsRequestId` - 調用請求的識別符。
- `logGroupName` - 函數的日誌群組。
- `logStreamName` - 函數執行個體的記錄串流。
- `identity` - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
  - `cognitoIdentityId` - 已驗證的 Amazon Cognito 身分。
  - `cognitoIdentityPoolId` - 授權調用的 Amazon Cognito 身分集區。
- `clientContext` - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `env.platform_version`
  - `env.platform`
  - `env.make`
  - `env.model`
  - `env.locale`

  - `custom` - 用戶端應用程式所設定的自訂值。
- `callbackWaitsForEmptyEventLoop` – 根據預設 (`true`)，在使用回呼型函數處理常式時，Lambda 會在回呼執行後等待事件迴圈為空，再結束函數叫用。將 設定為 `false`以傳送回應，並在回呼執行後立即結束調用，而不是等待事件迴圈為空。未處理的事件會在下次調用期間繼續執行。請注意，Lambda 僅支援 Node.js 22 和更早執行時間的回呼式函數處理常式。

以下範例函式紀錄內文資訊和傳回日誌的位置。

Example index.js 檔案

```
exports.handler = async function(event, context) {
  console.log('Remaining time: ', context.getRemainingTimeInMillis())
  console.log('Function name: ', context.functionName)
  return context.logStreamName
}
```

記錄和監控 Node.js Lambda 函數

AWS Lambda 會自動代表您監控 Lambda 函數，並將日誌傳送至 Amazon CloudWatch。您的 Lambda 函數隨附有 CloudWatch Logs 日誌群組，且函數的每一執行個體各有一個日誌串流。Lambda 執行期環境會將每次調用的詳細資訊傳送至日誌串流，並且轉傳來自函數程式碼的日誌及其他輸出。如需詳細資訊，請參閱將 Lambda 函式日誌傳送至 CloudWatch Logs。

此頁面說明如何從 Lambda 函數的程式碼產生日誌輸出，並使用 AWS Command Line Interface、Lambda 主控台或 CloudWatch 主控台存取日誌。

章節

- 建立傳回日誌的函數
- 搭配 Node.js 使用 Lambda 進階日誌控制項
- 在 Lambda 主控台檢視日誌
- 在 CloudWatch 主控台中檢視 記錄
- 使用 AWS Command Line Interface (AWS CLI) 檢視日誌
- 刪除日誌

建立傳回日誌的函數

若要由您的函式程式碼輸出日誌，您可以在主控台物件上使用方法，或任何能寫入 `stdout` 或 `stderr` 的記錄程式庫。下列範例會記錄環境變數和事件物件的值。

> **Note**
>
> 我們建議您在記錄輸入時使用輸入驗證和輸出編碼等技術。如果您直接記錄輸入資料，攻擊者可能會使用您的程式碼來使竄改難以被偵測、偽造日誌項目或繞過日誌監控。如需詳細資訊，請參閱常見弱點列舉中的 Improper Output Neutralization for Logs。

Example index.js 檔案 - 記錄

```
exports.handler = async function(event, context) {
  console.log("ENVIRONMENT VARIABLES\n" + JSON.stringify(process.env, null, 2))
  console.info("EVENT\n" + JSON.stringify(event, null, 2))
  console.warn("Event not processed.")
```

```
  return context.logStreamName
}
```

Example記錄格式

```
START RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Version: $LATEST
2019-06-07T19:11:20.562Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO ENVIRONMENT
 VARIABLES
{
  "AWS_LAMBDA_FUNCTION_VERSION": "$LATEST",
  "AWS_LAMBDA_LOG_GROUP_NAME": "/aws/lambda/my-function",
  "AWS_LAMBDA_LOG_STREAM_NAME": "2019/06/07/[$LATEST]e6f4a0c4241adcd70c262d34c0bbc85c",
  "AWS_EXECUTION_ENV": "AWS_Lambda_nodejs12.x",
  "AWS_LAMBDA_FUNCTION_NAME": "my-function",
  "PATH": "/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin",
  "NODE_PATH": "/opt/nodejs/node10/node_modules:/opt/nodejs/node_modules:/var/runtime/
node_modules",
  ...
}
2019-06-07T19:11:20.563Z c793869b-ee49-115b-a5b6-4fd21e8dedac INFO EVENT
{
  "key": "value"
}
2019-06-07T19:11:20.564Z c793869b-ee49-115b-a5b6-4fd21e8dedac WARN Event not processed.
END RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac
REPORT RequestId: c793869b-ee49-115b-a5b6-4fd21e8dedac Duration: 128.83 ms Billed
 Duration: 296 ms Memory Size: 128 MB Max Memory Used: 74 MB Init Duration: 166.62 ms
 XRAY TraceId: 1-5d9d007f-0a8c7fd02xmpl480aed55ef0 SegmentId: 3d752xmpl1bbe37e Sampled:
 true
```

Node.js 執行時間會記錄每次呼叫的 START、END 和 REPORT 行。它會將時間戳記、請求 ID 和日誌層級新增到函式所記錄的每個項目。報告明細行提供下列詳細資訊。

REPORT 行資料欄位

- RequestId - 進行調用的唯一請求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。當調用共用執行環境時，Lambda 會報告所有調用使用的記憶體上限。此行為可能會導致報告值高於預期值。

- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId - 對於追蹤的請求，這是 AWS X-Ray 追蹤 ID。
- SegmentId - 對於追蹤的請求，這是 X-Ray 區段 ID。
- 已取樣 - 對於追蹤的請求，這是取樣結果。

您可以在 Lambda 主控台、CloudWatch Logs 主控台或命令列中檢視日誌。

搭配 Node.js 使用 Lambda 進階日誌控制項

為了讓您更妥善地控制擷取、處理和使用函數日誌的方式，您可以針對支援的 Node.js 執行期設定下列記錄選項：

- 日誌格式 - 在純文字和結構化 JSON 格式之間為您的日誌進行選擇
- 日誌層級 - 對於 JSON 格式的日誌，請選擇 Lambda 傳送到 Amazon CloudWatch 的日誌之詳細等級，例如 ERROR、DEBUG 或 INFO
- 日誌群組 - 選擇您的函數將日誌傳送到的 CloudWatch 日誌群組

如需這些日誌選項的詳細資訊，以及如何設定函數以使用這些選項的說明，請參閱 the section called “設定 Lambda 函數的進階日誌記錄控制項”。

若要使用日誌格式和日誌層級選項與 Node.js Lambda 函數搭配使用，請參閱以下各章節中的指引。

搭配 Node.js 使用結構化的 JSON 日誌

如果您為函數的日誌格式選取 JSON，Lambda 會使用 `console.trace`、`console.debug`、`console.log`、`console.info`、`console.error` 和 `console.warn` 的主控台方法，將日誌輸出以結構化 JSON 形式傳送至 CloudWatch。每個 JSON 日誌物件都包含至少四個鍵值對，其中包含下列索引鍵：

- `"timestamp"` - 產生日誌訊息的時間
- `"level"` - 指派給訊息的日誌層級
- `"message"` - 日誌訊息的內容
- `"requestId"` - 進行調用的唯一請求 ID。

依據您的函數使用的記錄方法，此 JSON 物件還可能包含其他鍵值對。例如，如果您的函數使用 `console` 方法來記錄使用多個引數的錯誤物件，JSON 物件將包含具有索引鍵 `errorMessage`、`errorType` 和 `stackTrace` 的額外鍵值對。

如果您的程式碼已使用 Powertools 等其他日誌程式庫 AWS Lambda來產生 JSON 結構化日誌，則不需要進行任何變更。Lambda 不會對任何已經進行 JSON 編碼的日誌進行雙重編碼，因此您函數的應用程式日誌將繼續像以前一樣被擷取。

如需使用 Powertools AWS Lambda 記錄套件在 Node.js 執行時間中建立 JSON 結構化日誌的詳細資訊，請參閱 the section called “日誌”。

JSON 格式日誌輸出範例

下列範例顯示當您將函數的日誌格式設為 JSON 時，如何在 CloudWatch Logs 擷取使用單個和多個引數的 `console` 方法產生的各種日誌輸出。

第一個範例使用 `console.error` 方法輸出簡單字串。

Example Node.js 日誌程式碼

```
export const handler = async (event) => {
  console.error("This is a warning message");
  ...
}
```

Example JSON 日誌記錄

```
{
    "timestamp":"2025-11-01T00:21:51.358Z",
    "level":"ERROR",
    "message":"This is a warning message",
    "requestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

您還可以透過 `console` 方法使用單個或多個引數輸出結構更複雜的日誌訊息。在下一個範例中，您將使用 `console.log` 與單個引數輸出兩個鍵值對。請注意，在 Lambda 傳送至 CloudWatch Logs 的 JSON 物件中，`"message"` 欄位未被字串化。

Example Node.js 日誌程式碼

```
export const handler = async (event) => {
```

```
  console.log({data: 12.3, flag: false});
  ...
}
```

Example JSON 日誌記錄

```
{
    "timestamp": "2025-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "data": 12.3,
        "flag": false
    }
}
```

在下一個範例中，您要再次使用 console.log 方法建立日誌輸出。這一次，該方法使用兩個引數、一個包含兩個鍵值對的映射和一個識別字串。請注意，在這種情況下，由於您提供了兩個引數，Lambda 會對 "message" 欄位進行字串化。

Example Node.js 日誌程式碼

```
export const handler = async (event) => {
  console.log('Some object - ', {data: 12.3, flag: false});
  ...
}
```

Example JSON 日誌記錄

```
{
    "timestamp": "2025-12-08T23:21:04.664Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "Some object -  { data: 12.3, flag: false }"
}
```

Lambda 會指派使用 console.log 日誌層級 INFO 產生的輸出。

最後一個範例顯示如何使用 console 方法將錯誤物件輸出至 CloudWatch Logs。請注意，當您使用多個引數記錄錯誤物件時，Lambda 會新增 errorMessage、errorType 和 stackTrace 欄位至日誌輸出。

Example Node.js 日誌程式碼

```
export const handler = async (event) => {
  let e1 = new ReferenceError("some reference error");
  let e2 = new SyntaxError("some syntax error");
  console.log(e1);
  console.log("errors logged - ", e1, e2);
};
```

Example JSON 日誌記錄

```
{
    "timestamp": "2025-12-08T23:21:04.632Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": {
        "errorType": "ReferenceError",
        "errorMessage": "some reference error",
        "stackTrace": [
            "ReferenceError: some reference error",
            "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
            "    at Runtime.handleOnceNonStreaming (file:///var/runtime/
index.mjs:1173:29)"
        ]
    }
}

{
    "timestamp": "2025-12-08T23:21:04.646Z",
    "level": "INFO",
    "requestId": "405a4537-9226-4216-ac59-64381ec8654a",
    "message": "errors logged -  ReferenceError: some reference error
\n    at Runtime.handler (file:///var/task/index.mjs:3:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29) SyntaxError: some syntax
 error\n    at Runtime.handler (file:///var/task/index.mjs:4:12)\n    at
 Runtime.handleOnceNonStreaming
    (file:///var/runtime/index.mjs:1173:29)",
    "errorType": "ReferenceError",
    "errorMessage": "some reference error",
    "stackTrace": [
        "ReferenceError: some reference error",
        "    at Runtime.handler (file:///var/task/index.mjs:3:12)",
```

```
        "    at Runtime.handleOnceNonStreaming (file:///var/runtime/index.mjs:1173:29)"
    ]
}
```

記錄多個錯誤類型時，會從提供給 `console` 方法的第一個錯誤類型中擷取額外欄位 `errorMessage`、`errorType` 和 `stackTrace`。

搭配結構化 JSON 日誌使用內嵌指標格式 (EMF) 用戶端程式庫

AWS 提供 Node.js 的開放原始碼用戶端程式庫，可用來建立內嵌指標格式 (EMF) 日誌。如果您有使用這些程式庫的現有函數，並將函數的記錄格式變更為 JSON0，那麼CloudWatch 可能無法再識別程式碼所發出的指標。

如果您的程式碼目前直接使用 `console.log`或使用 Powertools for AWS Lambda (TypeScript) 發出 EMF 日誌，則如果您將函數的日誌格式變更為 JSON，CloudWatch 也無法剖析這些日誌。

> **Important**
>
> 為了確保 CloudWatch 能夠繼續正確剖析函數的 EMF 日誌檔，請將您的 EMF 和 Powertools for AWS Lambda 程式庫更新為最新版本。如果切換到 JSON 日誌格式，我們也建議您進行測試，以確保與函數的內嵌指標相容。如果您的程式碼直接使用 `console.log` 發出 EMF 記錄，請變更您的程式碼以將這些指標直接輸出至 `stdout`，如下列程式碼範例所示。

Example發出內嵌指標至 stdout 的程式碼

```
process.stdout.write(JSON.stringify(
    {
        "_aws": {
            "Timestamp": Date.now(),
            "CloudWatchMetrics": [{
                "Namespace": "lambda-function-metrics",
                "Dimensions": [["functionVersion"]],
                "Metrics": [{
                    "Name": "time",
                    "Unit": "Milliseconds",
                    "StorageResolution": 60
                }]
            }]
        },
        "functionVersion": "$LATEST",
```

```
        "time": 100,
        "requestId": context.awsRequestId
    }
) + "\n")
```

搭配 Node.js 使用日誌層級篩選

若要 AWS Lambda 讓 根據應用程式日誌層級篩選您的應用程式日誌，您的函數必須使用 JSON 格式的日誌。您可以透過兩種方式達成此操作：

- 使用標準主控台方法建立日誌輸出，並將函數設定為使用 JSON 日誌格式。 AWS Lambda 然後，會使用中所述的 JSON 物件中的「層級」鍵值對來篩選日誌輸出the section called “搭配 Node.js 使用結構化的 JSON 日誌”。若要瞭解如何設定函數的日誌格式，請參閱 the section called “設定 Lambda 函數的進階日誌記錄控制項”。
- 使用其他日誌程式庫或方法，在您的程式碼中建立 JSON 結構化日誌，其中包含定義日誌輸出層級的「層級」索引鍵值組。例如，您可以使用 Powertools for AWS Lambda 從程式碼產生 JSON 結構化日誌輸出。請參閱 the section called “日誌” 以瞭解有關在 Node.js 執行期使用 Powertools 的更多資訊。

  若要讓 Lambda 篩選函數的日誌，您還必須在 JSON 日誌輸出中包含 `"timestamp"` 索引鍵值組。必須以有效的 RFC 3339 時間戳記格式指定時間。如果您沒有提供有效的時間戳記，Lambda 會為日誌指派層級 INFO，並為您新增時間戳記。

當您將函數設定為使用日誌層級篩選時，請從下列選項中選取 AWS Lambda 您要傳送至 CloudWatch Logs 的日誌層級：

日誌層級	標準用量
TRACE (大多數詳細資訊)	用於追蹤程式碼執行路徑的最精細資訊
DEBUG	系統偵錯的詳細資訊
INFO	記錄函數正常操作的訊息
WARN	有關可能導致未解決意外行為的潛在錯誤的消息
ERROR	有關阻止程式碼按預期執行的問題的訊息
FATAL (最少詳細資訊)	有關導致應用程式停止運作的嚴重錯誤訊息

Lambda 會在選取的層級 (含) 和更低層級傳送日誌給 CloudWatch。例如，如果您設定 WARN 的日誌層級，Lambda 會傳送相對應於 WARN、ERROR 和 FATAL 層級的日誌檔。

在 Lambda 主控台檢視日誌

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

在 CloudWatch 主控台中檢視 記錄

您可以使用 Amazon CloudWatch 主控台來檢視所有 Lambda 函數調用的日誌。

若要在 CloudWatch 主控台上檢視日誌

1. 在 CloudWatch 主控台上開啟日誌群組頁面。
2. 選擇您的函數的日誌群組 (/aws/lambda/***your-function-name***)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數時，以及建立其他執行個體來處理並行調用時，就會顯示日誌串流。若要尋找特定調用的日誌，建議您使用 檢測函數 AWS X-Ray。X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

使用 AWS Command Line Interface (AWS CLI) 檢視日誌

AWS CLI 是一種開放原始碼工具，可讓您使用命令列 Shell 中的 命令與 AWS 服務互動。若要完成本節中的步驟，您必須擁有 AWS CLI 版本 2。

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
```

```
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 get-logs.sh。

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
```

```
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

在 中檢測 Node.js 程式碼 AWS Lambda

Lambda 與 整合 AWS X-Ray ，以協助您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追蹤請求，因為它會周遊您應用程式中的資源，其中可能包含 Lambda 函數和其他 AWS 服務。

若要將追蹤資料傳送至 X-Ray，您可以使用以下兩個 SDK 庫之一：

- AWS Distro for OpenTelemetry (ADOT) – OpenTelemetry (OTel) SDK 的安全、生產就緒、 AWS支援的分發。
- 適用於 Node.js 的 AWS X-Ray SDK – 用於生成追蹤資料並將其傳送至 X-Ray 的 SDK。

每個 SDK 均提供將遙測資料傳送至 X-Ray 服務的方法。然後，您可以使用 X-Ray 來檢視、篩選應用程式的效能指標並獲得洞察，從而識別問題和進行最佳化的機會。

> **Important**
>
> X-Ray 和 Powertools AWS Lambda SDKs是 提供的緊密整合檢測解決方案的一部分 AWS。ADOT Lambda Layers 是用於追蹤檢測之業界通用標準的一部分，這類檢測一般會收集更多資料，但可能不適用於所有使用案例。您可以使用任一解決方案在 X-Ray 中實作端對端追蹤。若要深入了解如何在兩者之間做選擇，請參閱在 AWS Distro for OpenTelemetry 和 X-Ray SDK 之間進行選擇。

章節

- 使用 ADOT 來檢測您的 Node.js 函數
- 使用 X-Ray SDK 來檢測 Node.js 函數
- 透過 Lambda 主控台來啟用追蹤
- 透過 Lambda API 啟用追蹤
- 使用 啟用追蹤 CloudFormation
- 解讀 X-Ray 追蹤
- 將執行時間相依項存放存在層中 (X-Ray SDK)

使用 ADOT 來檢測您的 Node.js 函數

ADOT 提供全受管 Lambda 層，包含使用 OTel SDK 收集遙測資料所需的一切內容。透過取用此層，您可以檢測 Lambda 函數，而無需修改任何函數程式碼。您還可以將層設定為對 OTel 進行自訂初始化。如需詳細資訊，請參閱 ADOT 文件中的針對 Lambda 上的 ADOT 收集器進行自訂組態設定。

針對 Python 執行階段，您可以新增適用於 ADOT Javascript 的AWS 受管 Lambda 層來自動檢測您的函數。如需有關如何新增此層的詳細資訊，請參閱 ADOT 文件中的 適用於 JavaScript 的AWS Distro for OpenTelemetry Lambda 支援。

使用 X-Ray SDK 來檢測 Node.js 函數

若要記錄 Lambda 函數對應用程式中其他資源所進行之呼叫的詳細資料，您也可以使用 適用於 Node.js 的 AWS X-Ray SDK。若要取得開發套件，請將 `aws-xray-sdk-core` 套件新增至應用程式的相依性。

Example blank-nodejs/package.json

```
{
  "name": "blank-nodejs",
  "version": "1.0.0",
  "private": true,
  "devDependencies": {
    "jest": "29.7.0"
  },
  "dependencies": {
    "@aws-sdk/client-lambda": "3.345.0",
    "aws-xray-sdk-core": "3.5.3"
  },
  "scripts": {
    "test": "jest"
  }
}
```

若要在 適用於 JavaScript 的 AWS SDK v3 中檢測 AWS SDK 用戶端，請使用 `captureAWSv3Client`方法包裝用戶端執行個體。

Example blank-nodejs/function/index.js – 追蹤 AWS SDK 用戶端

```
const AWSXRay = require('aws-xray-sdk-core');
```

```
const { LambdaClient, GetAccountSettingsCommand } = require('@aws-sdk/client-lambda');

// Create client outside of handler to reuse
const lambda = AWSXRay.captureAWSv3Client(new LambdaClient());

// Handler
exports.handler = async function(event, context) {
    event.Records.forEach(record => {
  ...
```

Lambda 執行階段會設定一些環境變數，以配置 X-Ray SDK。例如，Lambda 會將 AWS_XRAY_CONTEXT_MISSING 設定為 LOG_ERROR，以避免從 X-Ray SDK 中擲回執行階段錯誤。若要設置自訂內容遺失策略，請覆寫函式組態中的環境變數以使其不要有值，然後您可以透過程式設計方式設置內容遺失策略。

Example初始化程式碼範例

```
const AWSXRay = require('aws-xray-sdk-core');

// Configure the context missing strategy to do nothing
AWSXRay.setContextMissingStrategy(() => {});
```

如需詳細資訊，請參閱the section called “環境變數”。

新增正確的相依項並完成必要的程式碼變更後，請透過 Lambda 主控台或 API 在函數組態中啟用追蹤功能。

透過 Lambda 主控台來啟用追蹤

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態，然後選擇監控和操作工具。
4. 在其他監控工具欄位中，選擇編輯。
5. 在 CloudWatch Application Signals 和 AWS X-Ray 下，選擇啟用 Lambda 服務追蹤。
6. 選擇儲存。

透過 Lambda API 啟用追蹤

使用 AWS CLI 或 AWS SDK 在 Lambda 函數上設定追蹤，請使用下列 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

下列範例 AWS CLI 命令會在名為 my-function 的函數上啟用主動追蹤。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

使用 啟用追蹤 CloudFormation

若要在 CloudFormation 範本中的 `AWS::Lambda::Function` 資源上啟用追蹤，請使用 `TracingConfig` 屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用 `Tracing` 屬性。

Example template.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Serverless::Function
```

```
    Properties:
      Tracing: Active
      ...
```

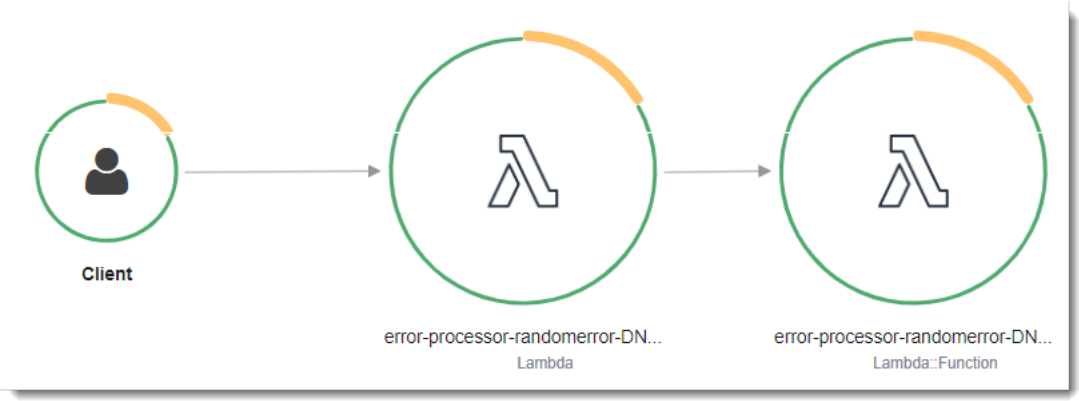
解讀 X-Ray 追蹤

您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將 AWSXRayDaemonWriteAccess 政策新增至執行角色。

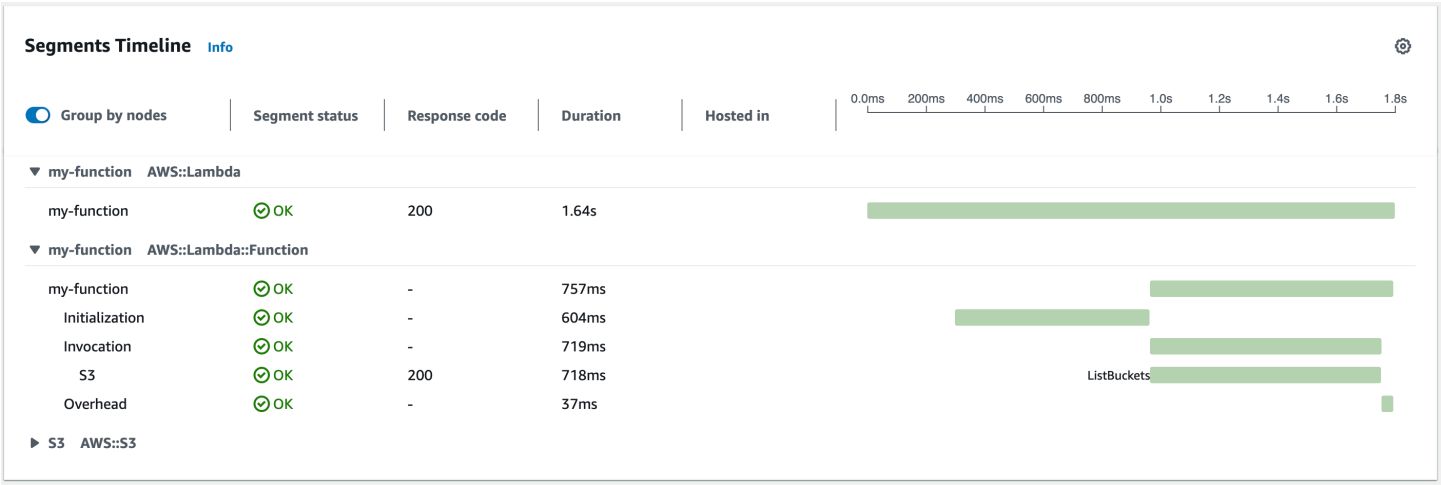
設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 服務圖顯示了有關應用程式及其所有元件的資訊。下列範例顯示了一個具有兩個函數的應用程式。主要函式會處理事件，有時會傳回錯誤。頂端的第二個函數會處理出現在第一個日誌群組中的錯誤，並使用 AWS SDK 呼叫 X-Ray、Amazon Simple Storage Service (Amazon S3) 和 Amazon CloudWatch Logs。

X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。不能針對函數設定 X-Ray 取樣率。

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會在每個追蹤上記錄 2 個區段，這會在服務圖表上建立兩個節點。下圖反白顯示了這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為 my-function，但其中之一的來源為 AWS::Lambda，而另一個的來源為 AWS::Lambda::Function。如果 AWS::Lambda 區段顯示錯誤，Lambda 服務就會出現問題。如果 AWS::Lambda::Function 區段顯示錯誤，表示您的函數出現了問題。



此範例會展開 AWS::Lambda::Function 區段以顯示其三個子區段：

> **Note**
>
> AWS 目前正在對 Lambda 服務實作變更。由於這些變更，您可能會看到系統日誌訊息的結構和內容，與 AWS 帳戶中不同 Lambda 函數發出的追蹤區段之間存在細微差異。
> 此處顯示的追蹤範例說明了舊式函數區段。下列段落說明了舊式和新式區段之間的差異。
> 這些變化將在未來幾週內實作，除中國和 GovCloud 區域以外，所有 AWS 區域 中的所有函數都會轉換至使用新格式的日誌訊息和追蹤區段。

舊式函數區段包含下列子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行期為做好準備以處理下一個事件所花費的時間。

新式函數區段不包含 `Invocation` 子區段。相反地，客戶子區段會直接連接至函數區段。如需舊式和新式函數區段結構的詳細資訊，請參閱the section called “了解 X-Ray 追蹤”。

您也可以檢測 HTTP 用戶端、記錄 SQL 查詢，以及建立具有註釋和中繼資料的自訂子區段。如需詳細資訊，請參閱《AWS X-Ray 開發人員指南》中的 適用於 Node.js 的 AWS X-Ray SDK 許可。

> **定價**
>
> 您可以在每月免費使用 X-Ray 追蹤，最高可達特定限制，作為 AWS 免費方案的一部分。達到閾值後，X-Ray 會收取追蹤儲存及擷取的費用。如需詳細資訊，請參閱 AWS X-Ray 定價。

將執行時間相依項存放存在層中 (X-Ray SDK)

如果您使用 X-Ray 開發套件來檢測函數程式碼的 AWS SDK 用戶端，您的部署套件可能會變得相當大。為了避免每次更新函數程式碼時上傳執行時間相依性，請將 X-Ray SDK 封裝在一個 Lambda 層中。

以下範例會顯示存放 適用於 Node.js 的 AWS X-Ray SDK的 `AWS::Serverless::LayerVersion` 資源。

Example template.yml - 相依性層

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
```

```
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-nodejs-lib
      Description: Dependencies for the blank sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - nodejs24.x
```

透過此組態，您只有在變更執行時間相依性時才會更新程式庫層。由於函數部署套件僅含有您的程式碼，因此有助於減少上傳時間。

為相依性建立圖層需要建置變更，才能在部署之前產生圖層封存。如需工作範例，請參閱 blank-nodejs 範例應用程式。

使用 TypeScript 建置 Lambda 函數

您可以使用 Node.js 執行期在 AWS Lambda中執行 TypeScript。由於 Node.js 不會在本機執行 TypeScript 程式碼，因此必須先將 TypeScript 程式碼轉譯為 JavaScript。然後，使用 JavaScript 檔案，將函數程式碼部署至 Lambda。您的程式碼會在包含適用於 JavaScript 的 AWS 開發套件的環境中執行，其中包含您管理之 AWS Identity and Access Management (IAM) 角色的登入資料。若要進一步了解 Node.js 執行時期隨附的 SDK 版本，請參閱 the section called “包含執行時期的 SDK 版本”。

Lambda 支援以下 Node.js 執行期。

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Node.js 24	`nodejs24.x`	Amazon Linux 2023	2028 年 4 月 30 日	2028 年 6 月 1 日	2028 年 7 月 1 日
Node.js 22	`nodejs22.x`	Amazon Linux 2023	2027 年 4 月 30 日	2027 年 6 月 1 日	2027 年 7 月 1 日
Node.js 20	`nodejs20.x`	Amazon Linux 2023	2026 年 4 月 30 日	2026 年 8 月 31 日	2026 年 9 月 30 日

主題

- 設定 TypeScript 開發環境
- Lambda 的類型定義
- 定義 TypeScript 格式的 Lambda 函數處理常式
- 使用 .zip 檔案封存，在 Lambda 中部署轉換的 TypeScript 程式碼
- 使用容器映像在 Lambda 中部署轉譯的 TypeScript 程式碼
- 使用 Lambda 內容物件擷取 TypeScript 函數資訊
- 記錄和監控 TypeScript Lambda 函數
- 在 中追蹤 TypeScript 程式碼 AWS Lambda

設定 TypeScript 開發環境

使用本機整合式開發環境 (IDE) 或文字編輯器來編寫 TypeScript 函式程式碼。無法在 Lambda 主控台上建立 TypeScript 程式碼。

您可以使用 esbuild 或 Microsoft 的 TypeScript 編譯器 (`tsc`)，將 TypeScript 程式碼轉譯為 JavaScript。AWS Serverless Application Model (AWS SAM) 與 AWS Cloud Development Kit (AWS CDK) 均採用 esbuild。

使用 esbuild 時，請考慮下列事項：

- 存在若干 TypeScript caveats。
- 您必須設定 TypeScript 轉譯設定，以便與計劃使用的 Node.js 執行時期相符。如需詳細資訊，請參閱 esbuild 文件中的目標。如需 tsconfig.json 檔案的範例，該檔案示範了如何以 Lambda 支援的特定 Node.js 版本為目標，請參閱 TypeScript GitHub 儲存庫。
- esbuild 不執行類型檢查。若要檢查類型，請使用 `tsc` 編譯器。執行 `tsc -noEmit` 或將 `"noEmit"` 參數新增至 tsconfig.json 檔案，如下列範例所示。這會將 `tsc` 設定為不發出 JavaScript 檔案。檢查類型後，使用 esbuild 將 TypeScript 檔案轉換為 JavaScript。

Example tsconfig.json

```
{
 "compilerOptions": {
   "target": "es2020",
   "strict": true,
   "preserveConstEnums": true,
   "noEmit": true,
   "sourceMap": false,
   "module":"commonjs",
   "moduleResolution":"node",
   "esModuleInterop": true,
   "skipLibCheck": true,
   "forceConsistentCasingInFileNames": true,
   "isolatedModules": true,
 },
 "exclude": ["node_modules", "**/*.test.ts"]
}
```

Lambda 的類型定義

@types/aws-lambda 套件提供 Lambda 函式的類型定義。若您的函式使用下列任意一項，請安裝此套件：

- 常見 AWS 事件來源，例如：
  - APIGatewayProxyEvent：適用於 Amazon API Gateway 代理整合
  - SNSEvent：適用於 Amazon Simple Notification Service 通知
  - SQSEvent：適用於 Amazon Simple Queue Service 訊息
  - S3Event：適用於 S3 觸發事件
  - DynamoDBStreamEvent：適用於 Amazon DynamoDB Streams
- Lambda 內容物件
- 回呼處理常式模式

若要將 Lambda 類型定義新增至函式，請將 @types/aws-lambda 安裝為開發相依項：

```
npm install -D @types/aws-lambda
```

然後，匯入 aws-lambda 中的類型：

```
import { Context, S3Event, APIGatewayProxyEvent } from 'aws-lambda';

export const handler = async (event: S3Event, context: Context) => {
    // Function code
};
```

陳述式 import ... from 'aws-lambda' 會匯入類型定義。不會匯入 aws-lambda npm 套件，該套件是一個不相關的第三方工具。如需詳細資訊，請參閱 DefinitelyTyped GitHub 儲存庫中的 aws-lambda。

> **Note**
>
> 若您使用自訂類型定義，則無需安裝 @types/aws-lambda 套件。如需可定義自訂事件物件類型的函式範例，請參閱 TypeScript Lambda 函式程式碼範例。

定義 TypeScript 格式的 Lambda 函數處理常式

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

本頁介紹了如何在 TypeScript 中使用 Lambda 函式處理常式，包括專案設定選項、命名慣例及最佳實務。本頁還提供了一個 TypeScript Lambda 函式範例，該函式會取得訂單的相關資訊、產生文字檔案收據，並將該檔案放入 Amazon Simple Storage Service (Amazon S3) 儲存貯體。如需編寫函數後如何部署函數的詳細資訊，請參閱the section called “部署 .zip 封存檔”或the section called “部署容器映像”。

主題

- 設定 TypeScript 專案
- TypeScript Lambda 函式程式碼範例
- CommonJS 和 ES 模組
- Node.js 初始化
- 處理常式命名慣例
- 定義和存取輸入事件物件
- TypeScript 函式的有效處理常式模式
- 在處理常式中使用適用於 JavaScript v3 的 SDK
- 存取環境變數
- 使用全域狀態
- TypeScript Lambda 函式的程式碼最佳實務

設定 TypeScript 專案

使用本機整合式開發環境 (IDE) 或文字編輯器來編寫 TypeScript 函式程式碼。無法在 Lambda 主控台上建立 TypeScript 程式碼。

初始化 TypeScript Lambda 專案的方法有多種。例如，您可以使用 npm 建立專案，建立 AWS SAM 應用程式，或建立 AWS CDK 應用程式。使用 npm 建立專案：

```
npm init
```

函式程式碼位於 `.ts` 檔案中，您在建置時需要將其轉譯為 JavaScript 檔案。您可以使用 esbuild 或 Microsoft 的 TypeScript 編譯器 (`tsc`)，將 TypeScript 程式碼轉換為 JavaScript。若要使用 esbuild，請將其新增為開發相依項：

```
npm install -D esbuild
```

一個典型的 TypeScript Lambda 函式專案遵循以下一般結構：

```
/project-root
  ### index.ts - Contains main handler
  ### dist/ - Contains compiled JavaScript
  ### package.json - Project metadata and dependencies
  ### package-lock.json - Dependency lock file
  ### tsconfig.json - TypeScript configuration
  ### node_modules/ - Installed dependencies
```

TypeScript Lambda 函式程式碼範例

以下 Lambda 函式程式碼範例會取得訂單的相關資訊、產生文字檔案收據，並將該檔案放入 Amazon S3 儲存貯體。此範例定義了自訂事件類型 (`OrderEvent`)。若要了解如何匯入 AWS 事件來源的類型定義，請參閱 Lambda 的類型定義。

> **Note**
>
> 此範例使用了 ES 模組處理常式。Lambda 同時支援 ES 模組與 CommonJS 處理常式。如需詳細資訊，請參閱CommonJS 和 ES 模組。

Example index.ts Lambda 函式

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';

// Initialize the S3 client outside the handler for reuse
const s3Client = new S3Client();

// Define the shape of the input event
type OrderEvent = {
    order_id: string;
    amount: number;
    item: string;
}
```

```
/**
 * Lambda handler for processing orders and storing receipts in S3.
 */
export const handler = async (event: OrderEvent): Promise<string> => {
    try {
        // Access environment variables
        const bucketName = process.env.RECEIPT_BUCKET;
        if (!bucketName) {
            throw new Error('RECEIPT_BUCKET environment variable is not set');
        }

        // Create the receipt content and key destination
        const receiptContent = `OrderID: ${event.order_id}\nAmount: $
${event.amount.toFixed(2)}\nItem: ${event.item}`;
        const key = `receipts/${event.order_id}.txt`;

        // Upload the receipt to S3
        await uploadReceiptToS3(bucketName, key, receiptContent);

        console.log(`Successfully processed order ${event.order_id} and stored receipt
 in S3 bucket ${bucketName}`);
        return 'Success';
    } catch (error) {
        console.error(`Failed to process order: ${error instanceof Error ?
 error.message : 'Unknown error'}`);
        throw error;
    }
};

/**
 * Helper function to upload receipt to S3
 */
async function uploadReceiptToS3(bucketName: string, key: string, receiptContent:
 string): Promise<void> {
    try {
        const command = new PutObjectCommand({
            Bucket: bucketName,
            Key: key,
            Body: receiptContent
        });

        await s3Client.send(command);
    } catch (error) {
```

```
        throw new Error(`Failed to upload receipt to S3: ${error instanceof Error ?
 error.message : 'Unknown error'}`);
    }
}
```

此 index.ts 檔案包含以下程式碼區段：

- import 區塊：此區塊用於納入 Lambda 函式所需的程式碼，例如 AWS SDK 用戶端。
- const s3Client 宣告：此宣告會在處理常式函式外部初始化 Amazon S3 用戶端。Lambda 會於初始化階段執行此程式碼，並且該用戶端會被保留下，在多次調用中重複使用。
- type OrderEvent：定義預期輸入事件的結構。
- export const handler：此為 Lambda 調用的主要處理常式函式。部署函式時，請為處理常式屬性指定 index.handler。Handler 屬性的值由檔案名稱與匯出的處理常式方法名稱組成，兩者以點號分隔。
- uploadReceiptToS3 函式：此為主要處理常式函式所參照的協助程式函式。

若要讓此函式正常運作，其執行角色必須允許 s3:PutObject 動作。此外，請確保定義 RECEIPT_BUCKET 環境變數。成功調用後，Amazon S3 儲存貯體應包含收據檔案。

CommonJS 和 ES 模組

Node.js 支援兩個模組系統：CommonJS 和 ECMAScript 模組 (ES 模組) 。Lambda 建議使用 ES 模組，因為它支援頂層等待，這可在執行環境初始化期間完成非同步任務。

Node.js 會將副檔名為 .cjs CommonJS 的檔案視為 CommonJS 模組，副.mjs檔名則表示 ES 模組。根據預設，Node.js 會將副檔名為 .js 的檔案視為 CommonJS 模組。您可以將 Node.js 設定為將.js檔案視為 ES 模組，方法是module在函數的 package.json 檔案中指定type為 。您可以在 Lambda 中設定 Node.js，透過將 –experimental-detect-module旗標新增至NODE_OPTIONS環境變數，自動偵測.js檔案是否應視為 CommonJS 或 ES 模組。如需詳細資訊，請參閱實驗 Node.js 功能。

下列範例顯示使用 ES 模組和 CommonJS 模組編寫的函數處理常式。本頁面上的其餘範例都使用 ES 模組。

Node.js 初始化

Node.js 使用非封鎖 I/O 模型，支援使用事件迴圈的高效率非同步操作。例如，如果 Node.js 進行網路呼叫，函數會繼續處理其他操作，而不會封鎖網路回應。收到網路回應時，會將其置於回呼佇列中。佇列中的任務會在目前任務完成時處理。

Lambda 建議使用頂層等待，以便在初始化期間完成執行環境初始化期間啟動的非同步任務。初始化期間未完成的非同步任務通常會在第一個函數叫用期間執行。這可能會導致非預期的行為或錯誤。例如，您的函數初始化可能會進行網路呼叫，以從參數存放區擷取 AWS 參數。如果此任務未在初始化期間完成，則值在調用期間可能為 null。初始化和調用之間也可能有延遲，這可能會在時間敏感的操作中觸發錯誤。特別是，如果在初始化階段未完成呼叫， AWS 服務呼叫可能會依賴時間敏感的請求簽章，導致服務呼叫失敗。在初始化期間完成任務通常會改善冷啟動效能，並在使用佈建並行時先調用效能。如需詳細資訊，請參閱我們的部落格文章使用 Node.js ES 模組和頂層等待。 AWS Lambda

處理常式命名慣例

設定函式時，處理常式的設定值由檔案名稱與匯出的處理常式方法名稱組成，二者以點號分隔。主控台中建立的函數以及本指南中的範例，預設值都是 `index.handler`。這表示 `handler` 方法是透過 `index.js` 或 `index.mjs` 檔案匯出的。

如果要在主控台中使用不同檔案名稱或函數處理常式名稱建立函數，您必須編輯預設處理常式名稱。

變更函數處理常式名稱的方式 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇您的函數。
2. 選擇 程式碼 索引標籤。
3. 向下捲動至執行時間設定窗格，並選擇編輯。
4. 在處理常式中，輸入函數處理常式的新名稱。
5. 選擇儲存。

定義和存取輸入事件物件

JSON 是 Lambda 函數最常見的標準輸入格式。在此範例中，函數預期輸入類似以下內容：

```
{
    "order_id": "12345",
    "amount": 199.99,
```

```
    "item": "Wireless Headphones"
}
```

使用 TypeScript 開發 Lambda 函式時，可透過類型或介面來定義輸入事件的結構。在此範例中，我們使用類型來定義事件結構：

```
type OrderEvent = {
    order_id: string;
    amount: number;
    item: string;
}
```

定義類型或介面之後，請在處理常式的簽章中加以使用，確保型別安全：

```
export const handler = async (event: OrderEvent): Promise<string> => {
```

在編譯期間，TypeScript 會驗證事件物件是否包含具有正確類型的必要欄位。例如，若嘗試將 `event.order_id` 用作數字或將 `event.amount` 用作字串，TypeScript 編譯器會報告錯誤。

TypeScript 函式的有效處理常式模式

建議您使用 async/await 來宣告函式處理常式，而不是使用 callback。async/await 能以簡潔且易讀的方式編寫非同步程式碼，無需使用巢狀回呼或串連承諾。您可以透過非同步/等待模式撰寫讀起來像同步程式碼的程式碼，同時仍維持非同步和非封鎖的特性。

本節中的範例使用了 `S3Event` 類型。不過，您可以在 @types/aws-lambda 套件中使用任何其他 AWS 事件類型，或定義您自己的事件類型。若要使用來自 @types/aws-lambda 的類型，請執行下列操作：

1. 將 @types/aws-lambda 套件新增為開發依賴項：

   ```
   npm install -D @types/aws-lambda
   ```

2. 匯入所需類型，例如 `Context`、`S3Event` 或 `Callback`。

非同步函數處理常式（建議）

`async` 關鍵字會將函數標記為非同步，且 `await` 關鍵字會暫停函數的執行，直到 `Promise` 獲得解決為止。處理常式接受下列引數：

- `event`：包含傳遞至函式的輸入資料。
- `context`：包含有關調用、函式以及執行環境的資訊。如需詳細資訊，請參閱使用 Lambda 內容物件擷取 TypeScript 函數資訊。

以下所示為 async/await 模式的有效簽章：

```
export const handler = async (event: S3Event): Promise<void> => { };
```

```
export const handler = async (event: S3Event, context: Context): Promise<void> => { };
```

> **Note**
>
> 以非同步方式處理項目陣列時，務必將 await 與 `Promise.all` 搭配使用，確保所有操作完成。`forEach` 等方法不會等待非同步回呼完成。如需更多資訊，請參閱 Mozilla 文件中的 Array.prototype.forEach()。

同步函數處理常式

如果您的函數不執行任何非同步任務，您可以使用同步函數處理常式，使用以下其中一個函數簽章：

```
export const handler = (event: S3Event): void => { };
```

```
export const handler = (event: S3Event, context: Context): void => { };
```

回應串流函數處理常式

Lambda 支援使用 Node.js 進行回應串流。回應串流函數處理常式使用 awslambda.streamifyResponse() 裝飾項目，並採用 3 個參數：事件、responseStream 和內容。函數簽章為：

```
export const handler = awslambda.streamifyResponse(async (event: APIGatewayProxyEvent,
 responseStream: NodeJS.WritableStream, context: Context) => { });
```

如需詳細資訊，請參閱 Lambda 函數的回應串流。

以回呼為基礎的函數處理常式

> **Note**
>
> 以回呼為基礎的函數處理常式最多僅支援 Node.js 22。從 Node.js 24 開始，應使用非同步函數處理常式實作非同步任務。

以回呼為基礎的函數處理常式可以使用事件、內容和回呼引數。回呼引數預期會接收一個 `Error` 與一個回應物件，其中回應物件必須可進行 JSON 序列化。

以下是回呼處理常式模式的有效簽章：

```
export const handler = (event: S3Event, context: Context, callback: Callback<void>):
 void => { };
```

該函式會一直執行，直至事件迴圈清空或函式逾時為止。直到完成所有的事件迴圈任務，回應才會傳送到叫用端。如果函式逾時，便會傳回錯誤。您可以透過設定 context.callbackWaitsForEmptyEventLoop 為「false (失敗)」來設定執行時間立即傳回回應。

Example帶回呼功能的 TypeScript 函數

下列範例使用了 `APIGatewayProxyCallback`，這是專用於 API Gateway 整合的指定回呼類型。大多數 AWS 事件來源使用上述簽章中顯示的一般`Callback`類型。

```
import { Context, APIGatewayProxyCallback, APIGatewayEvent } from 'aws-lambda';

export const lambdaHandler = (event: APIGatewayEvent, context: Context, callback:
 APIGatewayProxyCallback): void => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    callback(null, {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    });
};
```

在處理常式中使用適用於 JavaScript v3 的 SDK

通常，您將使用 Lambda 函數與其他 AWS 資源互動或進行更新。與這些資源互動的最簡單方法便是使用 適用於 JavaScript 的 AWS SDK。所有支援的 Lambda Node.js 執行時期都包含適用於 JavaScript 第 3 版的 SDK。不過，我們強烈建議您在部署套件中包含所需的 AWS SDK 用戶端。此舉能在日後 Lambda 執行時期更新時，最大限度實現回溯相容性。

若要將 SDK 相依項新增至函數，請針對您需要的特定 SDK 用戶端使用 `npm install` 命令。在範例程式碼中，我們使用的是 Amazon S3 用戶端。透過在包含 `package.json` 檔案的目錄中執行下列命令，即可新增這些相依項：

```
npm install @aws-sdk/client-s3
```

在函式程式碼中匯入所需的用戶端與命令，如範例函式所示：

```
import { S3Client, PutObjectCommand } from '@aws-sdk/client-s3';
```

然後，初始化 Amazon S3 用戶端：

```
const s3Client = new S3Client();
```

在此範例中，我們在主要處理常式函式外部初始化 Amazon S3 用戶端，以免每次調用函式時都必須初始化該用戶端。初始化 SDK 用戶端之後，您可以使用它來對該 AWS 服務進行 API 呼叫。此範例程式碼會呼叫 Amazon S3 PutObject API，如下所示：

```
const command = new PutObjectCommand({
    Bucket: bucketName,
    Key: key,
    Body: receiptContent
});
```

存取環境變數

在處理常式程式碼中，您可以使用 `process.env` 來引用任意環境變數。在此範例中，我們使用下列程式碼行來引用定義的 `RECEIPT_BUCKET` 環境變數：

```
// Access environment variables
const bucketName = process.env.RECEIPT_BUCKET;
if (!bucketName) {
```

```
    throw new Error('RECEIPT_BUCKET environment variable is not set');
}
```

使用全域狀態

在首次調用函式之前，Lambda 會在初始化階段執行靜態程式碼。在初始化期間建立的資源會在調用間隔期間保留在記憶體中，無需您在每次調用函式時都建立這些資源。

在範例程式碼中，S3 用戶端初始化程式碼位於主要處理常式外部。執行時期會在函式處理第一個事件之前初始化用戶端，且用戶端在所有調用中均可供重複使用。

TypeScript Lambda 函式的程式碼最佳實務

建置 Lambda 函式時，請遵循下列準則：

- 區隔 Lambda 處理常式與您的核心邏輯。能允許您製作更多可測單位的函式。
- 控制函數部署套件內的相依性。 AWS Lambda 執行環境包含許多程式庫。對於 Node.js 和 Python 執行期，這些包括 AWS SDKs。若要啟用最新的一組功能與安全更新，Lambda 會定期更新這些程式庫。這些更新可能會為您的 Lambda 函數行為帶來細微的變更。若要完全掌控您函式所使用的相依性，請利用部署套件封裝您的所有相依性。
- 最小化依存項目的複雜性。偏好更簡易的框架，其可快速在執行環境啟動時載入。
- 將部署套件最小化至執行時間所必要的套件大小。這能減少您的部署套件被下載與呼叫前解壓縮的時間。

請利用執行環境重新使用來改看函式的效能。在函式處理常式之外初始化 SDK 用戶端和資料庫連線，並在本機快取 `/tmp` 目錄中的靜態資產。由您函式的相同執行個體處理的後續叫用可以重複使用這些資源。這可藉由減少函數執行時間來節省成本。

若要避免叫用間洩漏潛在資料，請不要使用執行環境來儲存使用者資料、事件，或其他牽涉安全性的資訊。如果您的函式依賴無法存放在處理常式內記憶體中的可變狀態，請考慮為每個使用者建立個別函式或個別函式版本。

使用 Keep-Alive 指令維持持續連線的狀態。Lambda 會隨著時間的推移清除閒置連線。叫用函數時嘗試重複使用閒置連線將導致連線錯誤。若要維護持續連線，請使用與執行階段相關聯的 keep-alive (保持啟用) 指令。如需範例，請參閱在 Node.js 中重複使用 Keep-Alive 的連線。

使用環境變數將操作參數傳遞給您的函數。例如，如果您正在寫入到 Amazon S3 儲存貯體，而非對您正在寫入的儲存貯體名稱進行硬式編碼，請將儲存貯體名稱設定為環境變數。

避免在 Lambda 函數中使用遞迴調用，其中函數會調用自己或啟動可能再次調用函數的程序。這會導致意外的函式呼叫量與升高的成本。若您看到意外的調用數量，當更新程式碼時，請立刻將函數的預留並行設為 0，以調節對函數的所有調用。

請勿在您的 Lambda 函數程式碼中使用未記錄的非公有 API。對於 AWS Lambda 受管執行期，Lambda 會定期將安全性和功能更新套用至 Lambda 的內部 APIs。這些內部 API 更新可能是向後不相容的，這會導致意外結果，例如若您的函數依賴於這些非公有 API，則叫用失敗。請參閱 API 參考查看公開可用 API 的清單。

撰寫等冪程式碼。為函數撰寫等冪程式碼可確保採用相同方式來處理重複事件。程式碼應正確驗證事件並正常處理重複的事件。如需詳細資訊，請參閱 How do I make my Lambda function idempotent? (如何讓 Lambda 函數等冪？)。

使用 .zip 檔案封存，在 Lambda 中部署轉換的 TypeScript 程式碼

您必須先將 TypeScript 程式碼轉換為 JavaScript，才能將 TypeScript 程式碼部署到其中。 AWS Lambda JavaScript 本頁介紹了建置 TypeScript 程式碼，並透過 .zip 封存檔將其部署至 Lambda 的三種方法：

- 使用 AWS Serverless Application Model (AWS SAM)
- 使用 AWS Cloud Development Kit (AWS CDK)
- 使用 AWS Command Line Interface (AWS CLI) 和 esbuild

AWS SAM 並 AWS CDK 簡化 TypeScript 函數的建置和部署。AWS SAM 範本規範提供了一個簡單而清晰的語法，來描述構成無伺服器應用程式的 Lambda 函數、API、許可、組態和事件。藉助 AWS CDK，您可以在雲端建置可靠、可擴展且經濟高效的應用程式，並具有程式設計語言的強大表達能力。 AWS CDK 適用於中等至經驗豐富的 AWS 使用者。 AWS CDK 和 都 AWS SAM 使用 esbuild 將 TypeScript 程式碼轉換為 JavaScript。

使用 AWS SAM 將 TypeScript 程式碼部署至 Lambda

請遵循以下步驟，使用 AWS SAM下載、建置和部署範例 Hello World TypeScript 應用程式。此應用程式實作一個基本的 API 後端。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，會叫用 Lambda 函數。該函數會傳回 `hello world` 訊息。

> **Note**
>
> AWS SAM 使用 esbuild 從 TypeScript 程式碼建立 Node.js Lambda 函數。esbuild 支援目前為公開預覽。在公開預覽期間，esbuild 支援可能面臨向後相容變更。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- AWS SAM CLI 1.75 版或更新版本
- Node.js

部署範例 AWS SAM 應用程式

1. 使用 Hello World TypeScript 範本來初始化應用程式。

```
sam init --app-template hello-world-typescript --name sam-app --package-type Zip --runtime nodejs24.x
```

2. (選用) 範例應用程式包含常用工具的組態，如用於程式碼檢查的 ESLint 和用於單元測試的 Jest。執行檢查和測試命令：

```
cd sam-app/hello-world
npm install
npm run lint
npm run test
```

3. 建置應用程式。

```
cd sam-app
sam build
```

4. 部署應用程式。

```
sam deploy --guided
```

5. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請以 Enter 回應。
6. 輸出會顯示 REST API 的端點。在瀏覽器中開啟端點以測試函數。您應看到此回應：

```
{"message":"hello world"}
```

7. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

使用 AWS CDK 將 TypeScript 程式碼部署至 Lambda

請遵循以下步驟，使用 AWS CDK建置和部署範例 TypeScript 應用程式。此應用程式實作一個基本的 API 後端。其包含 API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，會叫用 Lambda 函數。該函數會傳回 `hello world` 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- AWS CDK 第 2 版
- Node.js
- Docker 或 esbuild

部署範例 AWS CDK 應用程式

1. 為您的新應用程式建立專案目錄。

```
mkdir hello-world
cd hello-world
```

2. 初始化應用程式。

```
cdk init app --language typescript
```

3. 新增 @types/aws-lambda 套件作為開發相依項。此套件包含 Lambda 的類型定義。

```
npm install -D @types/aws-lambda
```

4. 開啟 lib 目錄。您應看到名稱為 hello-world-stack.ts 的檔案。在此目錄中建立兩個新檔案：hello-world.function.ts 和 hello-world.ts。
5. 開啟 hello-world.function.ts，並將下列程式碼新增至檔案。這是 Lambda 函數的程式碼。

> **Note**
>
> `import` 陳述式會從 @types/aws-lambda 中匯入類型定義。不會匯入 `aws-lambda` NPM 套件，這是不相關的第三方工具。如需詳細資訊，請參閱 DefinitelyTyped GitHub 儲存庫中的 aws-lambda。

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
```

```
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. 開啟 hello-world.ts，然後將下列程式碼新增至檔案。這包含可建立 Lambda 函數的 NodejsFunction 建構，以及可建立 REST API 的 LambdaRestApi 建構。

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function');
    new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
  }
}
```

`NodejsFunction` 建構預設會假設以下內容：

- 您的函數處理常式被稱為 `handler`。
- 包含函數程式碼 (hello-world.function.ts) 的 .ts 檔案，與包含建構 (hello-world.ts) 的 .ts 檔案位於同一目錄中。該建構使用建構 ID ("hello-world") 和 Lambda 處理常式檔案的名稱 ("function") 來尋找函數程式碼。例如，如果您的函數程式碼位於名稱為 hello-world.my-function.ts 檔案，則 hello-world.ts 檔案必須引用如下所示函數程式碼：

```
const helloFunction = new NodejsFunction(this, 'my-function');
```

您可以變更此行為並設定其他 esbuild 參數。如需詳細資訊，請參閱 AWS CDK API 參考中的設定 esbuild。

7. 開啟 hello-world-stack.ts。這是定義 AWS CDK 堆疊的程式碼。將程式碼取代為以下內容：

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

8. 在包含 cdk.json 檔案的 hello-world 目錄中部署您的應用程式。

```
cdk deploy
```

9. 會使用 esbuild AWS CDK 建置和封裝 Lambda 函數，然後將函數部署至 Lambda 執行時間。輸出會顯示 REST API 的端點。在瀏覽器中開啟端點以測試函數。您應看到此回應：

```
{"message":"hello world"}
```

這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

使用 AWS CLI 和 esbuild 將 TypeScript 程式碼部署至 Lambda

下列範例示範如何使用 esbuild 和 將 TypeScript 程式碼傳輸和部署至 Lambda AWS CLI。esbuild 會產生一個 JavaScript 檔案，其中包含所有相依性。這是您需要新增至 .zip 封存的唯一檔案。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- Node.js
- Lambda 函數的執行角色
- 若您是 Windows 使用者，則為壓縮檔公用程式，如 7zip。

部署範例函數

1. 在您的本機電腦上，為新函數建立專案目錄。
2. 建立具有 npm 的新 Node.js 專案，或您選擇的套件管理器。

```
npm init
```

3. 新增 @types/aws-lambda 和 esbuild 套件作為開發相依項。@types/aws-lambda 套件包含 Lambda 的類型定義。

```
npm install -D @types/aws-lambda esbuild
```

4. 建立名稱為 index.ts 的新檔案。將下列程式碼新增至新檔案： 這是 Lambda 函數的程式碼。該函數會傳回 hello world 訊息。函數不會建立任何 API Gateway 資源。

> **Note**
>
> import 陳述式會從 @types/aws-lambda 中匯入類型定義。不會匯入 aws-lambda NPM 套件，這是不相關的第三方工具。如需詳細資訊，請參閱 DefinitelyTyped GitHub 儲存庫中的 aws-lambda。

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  console.log(`Event: ${JSON.stringify(event, null, 2)}`);
  console.log(`Context: ${JSON.stringify(context, null, 2)}`);
  return {
      statusCode: 200,
      body: JSON.stringify({
          message: 'hello world',
      }),
   };
};
```

5. 將建置指令碼新增至 package.json 檔案。這會將 esbuild 設定為自動建立 .zip 部署套件。如需詳細資訊，請參閱 esbuild 文件中的建置指令碼。

Linux and MacOS

```
"scripts": {
  "prebuild": "rm -rf dist",
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js",
  "postbuild": "cd dist && zip -r index.zip index.js*"
},
```

Windows

在此範例中，"postbuild" 命令會使用 7zip 公用程式來建立 .zip 檔案。使用您偏好的 Windows zip 公用程式，並視需要修改命令。

```
"scripts": {
  "prebuild": "del /q dist",
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js",
  "postbuild": "cd dist && 7z a -tzip index.zip index.js*"
},
```

6. 建置套件。

```
npm run build
```

7. 使用 .zip 部署套件來建立 Lambda 函數。用執行角色的 Amazon Resource Name (ARN) 取代反白顯示的文字。

```
aws lambda create-function --function-name hello-world --runtime "nodejs24.x" --
role arn:aws:iam::123456789012:role/lambda-ex --zip-file "fileb://dist/index.zip"
 --handler index.handler
```

8. 執行測試事件，以確認函數傳回下列回應。如果您想使用 API Gateway 叫用此函數，請建立並設定 REST API。

```
{
  "statusCode": 200,
  "body": "{\"message\":\"hello world\"}"
}
```

使用容器映像在 Lambda 中部署轉譯的 TypeScript 程式碼

您可以將 TypeScript 程式碼部署到 AWS Lambda 函數做為 Node.js 容器映像。 AWS 提供 Node.js 的基礎映像，以協助您建置容器映像。這些基礎映像會預先載入語言執行時間，以及在 Lambda 上執行映像所需的其他元件。 為每個基礎映像 AWS 提供 Dockerfile，以協助建置容器映像。

如果您使用社群或私有企業基礎映像，必須將 Node.js 執行時間介面用戶端 (RIC) 新增至基礎映像，以使其與 Lambda 相容。

Lambda 提供執行期介面模擬器，供您在本機測試函數。Node.js AWS 的基礎映像包含執行時間界面模擬器。如果您使用替代基礎映像 (例如 Alpine Linux 或 Debian 映像)，便可 將模擬器建置到映像中 或 將其安裝在本機電腦上。

使用 Node.js 基礎映像來建置和封裝 TypeScript 函數程式碼

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- Docker (最低版本 25.0.0)
- Docker buildx 外掛程式。
- Node.js 22.x

從基礎映像建立映像

從 Lambda AWS 的基礎映像建立映像

1. 在您的本機電腦上，為新函數建立專案目錄。
2. 建立具有 npm 的新 Node.js 專案，或您選擇的套件管理員。

```
npm init
```

3. 新增 @types/aws-lambda 和 esbuild 套件作為開發相依項。`@types/aws-lambda` 套件包含 Lambda 的類型定義。

```
npm install -D @types/aws-lambda esbuild
```

4. 將建置指令碼新增至 package.json 檔案。

```
  "scripts": {
  "build": "esbuild index.ts --bundle --minify --sourcemap --platform=node --
target=es2020 --outfile=dist/index.js"
}
```

5. 建立稱為 index.ts 的新檔案。將下列範本程式碼新增至新檔案。這是 Lambda 函數的程式碼。該函數會傳回 hello world 訊息。

> **Note**
>
> import 陳述式會從 @types/aws-lambda 中匯入類型定義。不會匯入 aws-lambda NPM 套件，這是不相關的第三方工具。如需詳細資訊，請參閱 DefinitelyTyped GitHub 儲存庫中的 aws-lambda。

```
import { Context, APIGatewayProxyResult, APIGatewayEvent } from 'aws-lambda';

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
    console.log(`Event: ${JSON.stringify(event, null, 2)}`);
    console.log(`Context: ${JSON.stringify(context, null, 2)}`);
    return {
        statusCode: 200,
        body: JSON.stringify({
            message: 'hello world',
        }),
    };
};
```

6. 建立包含下列組態的新 Dockerfile。

   - 將 FROM 屬性設定為基礎映像的 URI。
   - 設定 CMD 引數以指定 Lambda 函數處理常式。

   下列範例 Dockerfile 使用多階段建置。第一步將 TypeScript 程式碼轉譯為 JavaScript。第二步產生僅包含 JavaScript 檔案和生產相依項的容器映像。

請注意，範例 Dockerfile 不包含 USER 指令。當您將容器映像部署到 Lambda 時，Lambda 會自動定義一個具有最低權限許可的預設 Linux 使用者。這與標準 Docker 行為不同，後者會在未提供 USER 指令時預設為 root 使用者。

Example Dockerfile

```
FROM public.ecr.aws/lambda/nodejs:22 as builder
WORKDIR /usr/app
COPY package.json index.ts  ./
RUN npm install
RUN npm run build

FROM public.ecr.aws/lambda/nodejs:22
WORKDIR ${LAMBDA_TASK_ROOT}
COPY --from=builder /usr/app/dist/* ./
CMD ["index.handler"]
```

7. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。若要使映像與 Lambda 相容，必須使用 --provenance=false 選項。

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

(選用) 在本機測試映像

1. 使用 docker run 命令啟動 Docker 影像。在此範例中，docker-image 為映像名稱，test 為標籤。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將`--platform linux/arm64`選項改用選項。`--platform linux/amd64`

2. 從新的終端機視窗，將事件張貼至本機端點。

   Linux/macOS

   在 Linux 或 macOS 中，執行下列 `curl` 命令：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   在 PowerShell 中，執行下列 `Invoke-WebRequest` 命令：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

3. 取得容器 ID。

   ```
   docker ps
   ```

4. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將 `--region`值設定為您要 AWS 區域 在其中建立 Amazon ECR 儲存庫的 。
   - `111122223333` 以您的 AWS 帳戶 ID 取代 。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於相同的 AWS 區域 中。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
```

```
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - `docker-image:test` 為 Docker 映像檔的名稱和標籤。這是您在 `docker build` 命令中指定的映像名稱和標籤。
   - 將 `<ECRrepositoryUri>` 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像來建立函數。如需詳細資訊，請參閱Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 會將映像標籤解析為特定映像摘要。這表示如果您將用來部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。

若要將新映像部署至相同的 Lambda 函數，必須使用 update-function-code 命令，即使 Amazon ECR 中的映像標籤保持不變亦如此。在以下範例中，`--publish` 選項會使用更新的容器映像來建立新的函數版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

使用 Lambda 內容物件擷取 TypeScript 函數資訊

當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。此物件提供的方法和各項屬性提供了有關調用、函式以及執行環境的資訊。

若要啟用內容物件的類型檢查，必須將 @types/aws-lambda 套件新增為開發相依項，並匯入 `Context` 類型。如需詳細資訊，請參閱Lambda 的類型定義。

內容方法

- `getRemainingTimeInMillis()` - 傳回執行逾時前剩餘的毫秒數。

內容屬性

- `functionName` – Lambda 函數的名稱。
- `functionVersion` – 函數的版本。
- `invokedFunctionArn` - 用於調用此函數的 Amazon Resource Name (ARN)。指出調用者是否指定版本號或別名。
- `memoryLimitInMB` - 分配給函數的記憶體數量。
- `awsRequestId` - 調用請求的識別符。
- `logGroupName` - 函數的日誌群組。
- `logStreamName` - 函數執行個體的記錄串流。
- `identity` - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
  - `cognitoIdentityId` - 已驗證的 Amazon Cognito 身分。
  - `cognitoIdentityPoolId` - 授權調用的 Amazon Cognito 身分集區。
- `clientContext` - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `env.platform_version`
  - `env.platform`
  - `env.make`

  - `env.model`
  - `env.locale`
  - `Custom` - 用戶端應用程式所設定的自訂值。
- `callbackWaitsForEmptyEventLoop` – 根據預設 (`true`)，在使用回呼型函數處理常式時，Lambda 會在回呼執行後等待事件迴圈為空，再結束函數叫用。將 設定為 `false`以傳送回應，並在回呼執行後立即結束調用，而不是等待事件迴圈為空。未處理的事件會在下次調用期間繼續執行。請注意，Lambda 僅支援 Node.js 22 和更早執行時間的回呼式函數處理常式。

Example index.ts 檔案

以下範例函式紀錄內文資訊和傳回日誌的位置。

> **Note**
>
> 在 Lambda 函數中使用這段程式碼之前，您必須先新增 @types/aws-lambda 套件當作開發相依項。此套件包含 Lambda 的類型定義。如需詳細資訊，請參閱Lambda 的類型定義。

```
import { Context } from 'aws-lambda';
export const lambdaHandler = async (event: string, context: Context): Promise<string>
 => {
  console.log('Remaining time: ', context.getRemainingTimeInMillis());
  console.log('Function name: ', context.functionName);
  return context.logStreamName;
};
```

記錄和監控 TypeScript Lambda 函數

AWS Lambda 會自動監控 Lambda 函數，並將日誌項目傳送至 Amazon CloudWatch。您的 Lambda 函數隨附有 CloudWatch Logs 日誌群組，且函數的每一執行個體各有一個日誌串流。Lambda 執行期環境會將每次調用的詳細資訊和函數程式碼的其他輸出，傳送至日誌串流。如需 CloudWatch Logs 的詳細資訊，請參閱將 Lambda 函式日誌傳送至 CloudWatch Logs。

若要由您的函數程式碼輸出日誌，您可以在主控台物件上使用方法。若要更詳細記錄，您可以使用任何寫入 `stdout` 或 `stderr` 的記錄程式庫。

章節

- 使用日誌記錄工具和程式庫
- 使用 Powertools for AWS Lambda (TypeScript) 和 AWS SAM 進行結構化記錄
- 使用 Powertools for AWS Lambda (TypeScript) 和 AWS CDK 進行結構化記錄
- 在 Lambda 主控台檢視日誌
- 在 CloudWatch 主控台中檢視 記錄

使用日誌記錄工具和程式庫

Powertools for AWS Lambda (TypeScript) 是一種開發人員工具組，可實作無伺服器最佳實務並提高開發人員速度。Logger 公用程式提供 Lambda 優化記錄器，其中包含有關所有函數之函數內容的其他資訊，輸出結構為 JSON。使用此公用程式執行下列操作：

- 從 Lambda 內容、冷啟動和 JSON 形式的結構記錄輸出中擷取關鍵欄位
- 在收到指示時記錄 Lambda 調用事件 (預設為停用)
- 透過日誌採樣僅列印調用百分比的所有日誌 (預設為停用)
- 在任何時間點將其他金鑰附加至結構化日誌
- 使用自訂日誌格式化程式 (自帶格式化程式)，以與組織的日誌記錄 RFC 相容的結構輸出日誌。

使用 Powertools for AWS Lambda (TypeScript) 和 AWS SAM 進行結構化記錄

請依照下列步驟，使用 搭配整合式 Powertools for (TypeScript) 模組下載、建置和部署範例 Hello World TypeScript 應用程式 AWS SAM。 AWS Lambda TypeScript 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您

將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會調用、使用內嵌指標格式將日誌和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- Node.js 20 或更新版本
- AWS CLI 第 2 版
- AWS SAM CLI 1.75 版或更新版本。如果您有較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World TypeScript 範本來初始化應用程式。

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs24.x
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 `Enter`。

> **Note**
>
> 對於 HelloWorldFunction may not have authorization defined, Is this okay?，確保輸入 `y`。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要獲取該函數的日誌，請執行 sam 日誌。如需詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》 中的 使用日誌。

```
sam logs --stack-name sam-app
```

日誌輸出如下：

```
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.552000
 START RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Version: $LATEST
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.594000
 2025-08-31T09:33:10.557Z 70693159-7e94-4102-a2af-98a6343fb8fb
 INFO {"_aws":{"Timestamp":1661938390556,"CloudWatchMetrics":
[{"Namespace":"sam-app","Dimensions":[["service"]],"Metrics":
[{"Name":"ColdStart","Unit":"Count"}]}]},"service":"helloWorld","ColdStart":1}
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.595000
 2025-08-31T09:33:10.595Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
 {"level":"INFO","message":"This is an INFO log - sending HTTP 200 - hello world
 response","service":"helloWorld","timestamp":"2025-08-31T09:33:10.594Z"}
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.655000
 2025-08-31T09:33:10.655Z 70693159-7e94-4102-a2af-98a6343fb8fb INFO
 {"_aws":{"Timestamp":1661938390655,"CloudWatchMetrics":[{"Namespace":"sam-
app","Dimensions":[["service"]],"Metrics":[]}]},"service":"helloWorld"}
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.754000 END
 RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb
2025/01/31/[$LATEST]4d53e8d279824834a1ccd35511a4949c 2025-08-31T09:33:10.754000
 REPORT RequestId: 70693159-7e94-4102-a2af-98a6343fb8fb Duration: 201.55 ms Billed
 Duration: 202 ms Memory Size: 128 MB Max Memory Used: 66 MB Init Duration: 252.42
 ms
XRAY TraceId: 1-630f2ad5-1de22b6d29a658a466e7ecf5 SegmentId: 567c116658fbf11a
 Sampled: true
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

管理日誌保留

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，CloudWatch 會在該時間之後自動刪除日誌。若要設定日誌保留，請將下列項目新增至您的 AWS SAM 範本：

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
    # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

使用 Powertools for AWS Lambda (TypeScript) 和 AWS CDK 進行結構化記錄

請依照下列步驟，使用 下載、建置和部署範例 Hello World TypeScript 應用程式，搭配整合式 Powertools for AWS Lambda (TypeScript) 模組 AWS CDK。此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會調用、使用內嵌指標格式將日誌和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- Node.js 20 或更新版本
- AWS CLI 第 2 版
- AWS CDK 第 2 版
- AWS SAM CLI 1.75 版或更新版本。如果您有較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS CDK 應用程式

1. 為您的新應用程式建立專案目錄。

```
mkdir hello-world
cd hello-world
```

2. 初始化應用程式。

```
cdk init app --language typescript
```

3. 新增 @types/aws-lambda 套件作為開發相依項。

```
npm install -D @types/aws-lambda
```

4. 安裝 Powertools Logger 公用程式。

```
npm install @aws-lambda-powertools/logger
```

5. 開啟 lib 目錄。您應看到名稱為 hello-world-stack.ts 的檔案。在此目錄中建立兩個新檔案：hello-world.function.ts 和 hello-world.ts。
6. 開啟 hello-world.function.ts，並將下列程式碼新增至檔案。這是 Lambda 函數的程式碼。

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Logger } from '@aws-lambda-powertools/logger';
const logger = new Logger();

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  logger.info('This is an INFO log - sending HTTP 200 - hello world response');
  return {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
};
```

7. 開啟 hello-world.ts，然後將下列程式碼新增至檔案。其中包含 NodejsFunction 建構模組，它會建立 Lambda 函數、設定 Powertools 的環境變數，並將日誌保留設定為一週。另外也包括負責建立 REST API 的 LambdaRestApi 建構模組。

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { RetentionDays } from 'aws-cdk-lib/aws-logs';
import { CfnOutput } from 'aws-cdk-lib';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        Powertools_SERVICE_NAME: 'helloWorld',
        LOG_LEVEL: 'INFO',
      },
      logRetention: RetentionDays.ONE_WEEK,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
}
```

8. 開啟 hello-world-stack.ts。這是定義 AWS CDK 堆疊的程式碼。將程式碼取代為以下內容：

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. 返回專案目錄。

```
cd hello-world
```

10. 部署您的應用程式。

```
cdk deploy
```

11. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

12. 調用 API 端點：

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

13. 若要獲取該函數的日誌，請執行 sam 日誌。如需詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》 中的 使用日誌。

```
sam logs --stack-name HelloWorldStack
```

日誌輸出如下：

```
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.047000
 START RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Version: $LATEST
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.050000 {
"level": "INFO",
"message": "This is an INFO log - sending HTTP 200 - hello world response",
"service": "helloWorld",
"timestamp": "2025-08-31T14:48:37.048Z",
"xray_trace_id": "1-630f74c4-2b080cf77680a04f2362bcf2"
}
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.082000 END
 RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c
2025/01/31/[$LATEST]2ca67f180dcd4d3e88b5d68576740c8e 2025-08-31T14:48:37.082000
 REPORT RequestId: 19ad1007-ff67-40ce-9afe-0af0a9eb512c Duration: 34.60 ms Billed
 Duration: 35 ms Memory Size: 128 MB Max Memory Used: 57 MB Init Duration: 173.48
 ms
```

14. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
cdk destroy
```

在 Lambda 主控台檢視日誌

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

在 CloudWatch 主控台中檢視 記錄

您可以使用 Amazon CloudWatch 主控台來檢視所有 Lambda 函數調用的日誌。

若要在 CloudWatch 主控台上檢視日誌

1. 在 CloudWatch 主控台上開啟日誌群組頁面。
2. 選擇您的函數的日誌群組 (/aws/lambda/***your-function-name***)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立其他執行個體以處理並行調用時，就會顯示日誌串流。若要尋找特定調用的日誌，建議您使用 檢測函數 AWS X-Ray。X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

在 中追蹤 TypeScript 程式碼 AWS Lambda

Lambda 與 整合 AWS X-Ray ，以協助您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追蹤請求，因為它會周遊您應用程式中的資源，其中可能包含 Lambda 函數和其他 AWS 服務。

若要將追蹤資料傳送至 X-Ray，您可以使用以下三個 SDK 庫之一：

- AWS Distro for OpenTelemetry (ADOT) – OpenTelemetry (OTel) SDK 的安全、生產就緒、 AWS支援的分發。
- AWS X-Ray 適用於 Node.js 的開發套件 – 用於產生追蹤資料並將其傳送至 X-Ray 的開發套件。
- Powertools for AWS Lambda (TypeScript) – 開發人員工具組，用於實作無伺服器最佳實務並提高開發人員速度。

每個 SDK 均提供將遙測資料傳送至 X-Ray 服務的方法。然後，您可以使用 X-Ray 來檢視、篩選應用程式的效能指標並獲得洞察，從而識別問題和進行最佳化的機會。

> **Important**
>
> X-Ray 和 Powertools AWS Lambda SDKs是 提供的緊密整合檢測解決方案的一部分 AWS。ADOT Lambda Layers 是用於追蹤檢測之業界通用標準的一部分，這類檢測一般會收集更多資料，但可能不適用於所有使用案例。您可以使用任一解決方案在 X-Ray 中實作端對端追蹤。若要深入了解如何在兩者之間做選擇，請參閱在 AWS Distro for OpenTelemetry 和 X-Ray SDK 之間進行選擇。

章節

- 使用 Powertools for AWS Lambda (TypeScript) 和 AWS SAM 進行追蹤
- 使用 Powertools for AWS Lambda (TypeScript) 和 AWS CDK 進行追蹤
- 解讀 X-Ray 追蹤

使用 Powertools for AWS Lambda (TypeScript) 和 AWS SAM 進行追蹤

請依照下列步驟，使用 搭配整合式 Powertools for (TypeScript) 模組下載、建置和部署範例 Hello World TypeScript 應用程式 AWS SAM。 AWS Lambda TypeScript 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您

將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會調用、使用內嵌指標格式將日誌和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- Node.js
- AWS CLI 第 2 版
- AWS SAM CLI 1.75 版或更新版本。如果您有較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World TypeScript 範本來初始化應用程式。

```
sam init --app-template hello-world-powertools-typescript --name sam-app --package-type Zip --runtime nodejs24.x --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 `Enter`。

> **Note**
>
> 對於 HelloWorldFunction may not have authorization defined, Is this okay?，確保輸入 y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

```
XRay Event [revision 1] at (2023-01-31T11:29:40.527000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.483s)
  - 0.425s - sam-app/Prod [HTTP: 200]
    - 0.422s - Lambda [HTTP: 200]
  - 0.406s - sam-app-HelloWorldFunction-Xyzv11a1bcde [HTTP: 200]
  - 0.172s - sam-app-HelloWorldFunction-Xyzv11a1bcde
    - 0.179s - Initialization
    - 0.112s - Invocation
      - 0.052s - ## app.lambdaHandler
        - 0.001s - ### MySubSegment
    - 0.059s - Overhead
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。不能針對函數設定 X-Ray 取樣率。

使用 Powertools for AWS Lambda (TypeScript) 和 AWS CDK 進行追蹤

請依照下列步驟，使用 搭配整合式 Powertools for (TypeScript) 模組下載、建置和部署範例 Hello World TypeScript 應用程式 AWS CDK。 AWS Lambda TypeScript 此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會調用、使用內嵌指標格式將日誌和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- Node.js
- AWS CLI 第 2 版
- AWS CDK 第 2 版
- AWS SAM CLI 1.75 版或更新版本。如果您有較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS Cloud Development Kit (AWS CDK) 應用程式

1. 為您的新應用程式建立專案目錄。

```
mkdir hello-world
cd hello-world
```

2. 初始化應用程式。

```
cdk init app --language typescript
```

3. 新增 @types/aws-lambda 套件作為開發相依項。

```
npm install -D @types/aws-lambda
```

4. 安裝 Powertools Tracer 公用程式。

```
npm install @aws-lambda-powertools/tracer
```

5. 開啟 lib 目錄。您應看到名稱為 hello-world-stack.ts 的檔案。在此目錄中建立兩個新檔案：hello-world.function.ts 和 hello-world.ts。
6. 開啟 hello-world.function.ts，並將下列程式碼新增至檔案。這是 Lambda 函數的程式碼。

```
import { APIGatewayEvent, APIGatewayProxyResult, Context } from 'aws-lambda';
import { Tracer } from '@aws-lambda-powertools/tracer';
const tracer = new Tracer();

export const handler = async (event: APIGatewayEvent, context: Context):
 Promise<APIGatewayProxyResult> => {
  // Get facade segment created by Lambda
```

```
  const segment = tracer.getSegment();

  // Create subsegment for the function and set it as active
  const handlerSegment = segment.addNewSubsegment(`## ${process.env._HANDLER}`);
  tracer.setSegment(handlerSegment);

  // Annotate the subsegment with the cold start and serviceName
  tracer.annotateColdStart();
  tracer.addServiceNameAnnotation();

  // Add annotation for the awsRequestId
  tracer.putAnnotation('awsRequestId', context.awsRequestId);
  // Create another subsegment and set it as active
  const subsegment = handlerSegment.addNewSubsegment('### MySubSegment');
  tracer.setSegment(subsegment);
  let response: APIGatewayProxyResult = {
    statusCode: 200,
    body: JSON.stringify({
      message: 'hello world',
    }),
  };
  // Close subsegments (the Lambda one is closed automatically)
  subsegment.close(); // (### MySubSegment)
  handlerSegment.close(); // (## index.handler)

  // Set the facade segment as active again (the one created by Lambda)
  tracer.setSegment(segment);
  return response;
};
```

7. 開啟 hello-world.ts，然後將下列程式碼新增至檔案。其中包含 NodejsFunction 建構模組，它會建立 Lambda 函數、設定 Powertools 的環境變數，並將日誌保留設定為一週。另外也包括負責建立 REST API 的 LambdaRestApi 建構模組。

```
import { Construct } from 'constructs';
import { NodejsFunction } from 'aws-cdk-lib/aws-lambda-nodejs';
import { LambdaRestApi } from 'aws-cdk-lib/aws-apigateway';
import { CfnOutput } from 'aws-cdk-lib';
import { Tracing } from 'aws-cdk-lib/aws-lambda';

export class HelloWorld extends Construct {
  constructor(scope: Construct, id: string) {
    super(scope, id);
```

```
    const helloFunction = new NodejsFunction(this, 'function', {
      environment: {
        POWERTOOLS_SERVICE_NAME: 'helloWorld',
      },
      tracing: Tracing.ACTIVE,
    });
    const api = new LambdaRestApi(this, 'apigw', {
      handler: helloFunction,
    });
    new CfnOutput(this, 'apiUrl', {
      exportName: 'apiUrl',
      value: api.url,
    });
  }
}
```

8. 開啟 hello-world-stack.ts。這是定義 AWS CDK 堆疊的程式碼。將程式碼取代為以下內容：

```
import { Stack, StackProps } from 'aws-cdk-lib';
import { Construct } from 'constructs';
import { HelloWorld } from './hello-world';

export class HelloWorldStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);
    new HelloWorld(this, 'hello-world');
  }
}
```

9. 部署您的應用程式。

```
cd ..
cdk deploy
```

10. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?ExportName==`apiUrl`].OutputValue' --output text
```

11. 調用 API 端點：

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

12. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

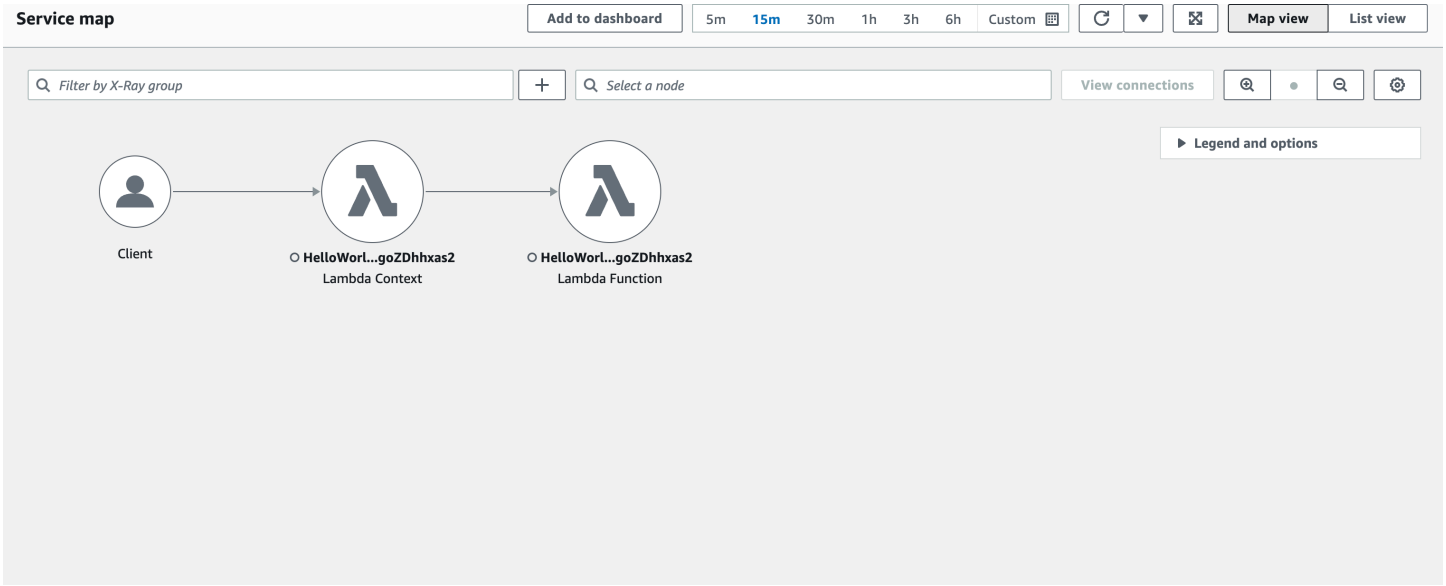
```
XRay Event [revision 1] at (2023-01-31T11:50:06.997000) with id
 (1-11a2222-111a222222cb33de3b95daf9) and duration (0.449s)
  - 0.350s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde [HTTP: 200]
  - 0.157s - HelloWorldStack-helloworldfunction111A2BCD-Xyzv11a1bcde
    - 0.169s - Initialization
    - 0.058s - Invocation
      - 0.055s - ## index.handler
        - 0.000s - ### MySubSegment
    - 0.099s - Overhead
```

13. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
cdk destroy
```

解讀 X-Ray 追蹤

設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 追蹤地圖提供了有關應用程式及其所有元件的資訊。下列範例顯示範例應用程式的追蹤：


Service map
Add to dashboard
5m
15m
30m
1h
3h
6h
Custom
Map view
List view
Filter by X-Ray group
Select a node
View connections
Legend and options
Client
HelloWorl...goZDhhxas2
Lambda Context
HelloWorl...goZDhhxas2
Lambda Function

使用 Python 建置 Lambda 函數

您可以在 AWS Lambda中執行 Python 程式碼。Lambda 提供用於執行程式碼來處理事件的 Python 執行期。您的程式碼會在包含 SDK for Python (Boto3) 的環境中執行，其中包含您管理之 AWS Identity and Access Management (IAM) 角色的登入資料。若要進一步了解 Python 執行時期隨附的 SDK 版本，請參閱 the section called “包含執行時期的 SDK 版本”。

Lambda 支援以下 Python 執行期。

Name	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Python 3.14	`python3.14`	Amazon Linux 2023	2029 年 6 月 30 日	2029 年 7 月 31 日	2029 年 8 月 31 日
Python 3.13	`python3.13`	Amazon Linux 2023	2029 年 6 月 30 日	2029 年 7 月 31 日	2029 年 8 月 31 日
Python 3.12	`python3.12`	Amazon Linux 2023	2028 年 10 月 31 日	2028 年 11 月 30 日	2029 年 1 月 10 日
Python 3.11	`python3.11`	Amazon Linux 2	2027 年 6 月 30 日	2027 年 7 月 31 日	2027 年 8 月 31 日
Python 3.10	`python3.10`	Amazon Linux 2	2026 年 10 月 31 日	2026 年 11 月 30 日	2027 年 1 月 15 日

若要建立 Python 函數

1. 開啟 Lambda 主控台。
2. 選擇建立函數。
3. 進行下列設定：
   - 函數名稱：輸入函數名稱。
   - 執行時間：選擇 Python 3.14。
4. 選擇建立函數。

主控台將建立一個 Lambda 函數，其具有名為 `lambda_function` 的單一來源檔案。您可以使用內建的程式碼編輯器編輯該檔案並加入更多檔案。在 DEPLOY 區段中，選擇部署以更新函數的程式碼。然後，若要執行程式碼，請在測試事件區段中選擇建立測試事件。

Lambda 函數隨附有 CloudWatch Logs 記錄群組。函數執行期會將每次調用的詳細資訊傳送至 CloudWatch Logs。它在調用期間會轉送您的函數輸出的任何記錄。如果您的函數傳回錯誤，Lambda 會對該錯誤進行格式化之後傳回給調用端。

主題

- 包含執行時期的 SDK 版本
- 停用的 Python 功能
- 回應格式
- 延伸模組正常關機
- 以 Python 定義 Lambda 函數處理常式
- 使用 .zip 封存檔部署 Python Lambda 函數
- 使用容器映像部署 Python Lambda 函數
- 針對 Python Lambda 函數使用層
- 使用 Lambda 內容物件擷取 Python 函數資訊
- 記錄和監控 Python Lambda 函數
- 以 Python 測試 AWS Lambda 函數
- 在 中檢測 Python 程式碼 AWS Lambda

包含執行時期的 SDK 版本

Python 執行時間中包含的 AWS SDK 版本取決於執行時間版本和您的 AWS 區域。若要尋找您使用的執行時期中包含的 SDK 版本，請使用下列程式碼建立 Lambda 函數。

```
import boto3
import botocore

def lambda_handler(event, context):
   print(f'boto3 version: {boto3.__version__}')
   print(f'botocore version: {botocore.__version__}')
```

停用的 Python 功能

下表列出在適用於 Python 的 Lambda 受管執行期和容器基礎映像中停用的 Python 功能。編譯 Python 執行時間可執行檔時，必須啟用這些功能，且無法使用執行時間旗標啟用。若要在 Lambda 中使用這些功能，您可以使用容器映像或自訂執行期，在啟用這些功能的情況下部署自己的 Python 執行期組建。

Python 功能	受影響的 Python 版本	狀態
Just-in-Time(JIT) 編譯器	Python 3.13 及更新版本	JIT 編譯器是實驗性的，不建議用於生產工作負載。因此，它會在 Lambda Python 執行時間中停用。
自由執行緒	Python 3.13 及更新版本	由於對單一執行緒程式碼的效能影響，Lambda Python 組建中會停用免費執行緒（停用全域解譯器鎖定的選項）。

回應格式

在 Python 3.12 及更高版本 Python 的執行期中，函數傳回的 JSON 回應包含 Unicode 字元。早期版本 Python 的執行期會在回應中傳回 Unicode 字元的逸出序列。例如，在 Python 3.11 中，如果您傳回 Unicode 字串，如 "こんにちは"，它將逸出 Unicode 字元並傳回 "\u3053\u3093\u306b\u3061\u306f"。Python 3.12 執行期會傳回原始的 "こんにちは"。

使用 Unicode 回應可使 Lambda 回應變小，因此能更容易地將較大的回應納入同步函數的 6 MB 最大承載大小。在之前的範例中，逸出版本為 32 位元組，相較之下，Unicode 字串為 17 位元組。

當您升級到 Python 3.12 或更高 Python 執行時期時，可能需要調整您的程式碼以適應新的回應格式。若呼叫者預期得到逸出 Unicode，您必須新增程式碼至傳回的函數以便手動逸出 Unicode，或調整呼叫者以處理 Unicode 傳回。

延伸模組正常關機

Python 3.12 及更高版本 Python 的執行期為具有外部延伸模組的函數提供正常關機功能。當 Lambda 關閉執行環境時，它會傳送 `SIGTERM` 訊號到執行期，然後傳送 `SHUTDOWN` 事件到每個註冊的外部延伸模組。您可以捕獲 Lambda 函數中的 `SIGTERM` 訊息並清理資源，例如由函數建立的資料庫連線等。

若要詳細了解執行環境生命週期，請參閱 了解 Lambda 執行環境生命週期。有關如何使用延伸模組正常關機的範例，請參閱 AWS 範例 GitHub 儲存庫。

以 Python 定義 Lambda 函數處理常式

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

本頁介紹了如何在 Python 中使用 Lambda 函式處理常式，包括命名慣例、有效的處理常式簽章及程式碼最佳實務。本頁還提供了一個 Python Lambda 函式範例，該函式會取得訂單的相關資訊、產生文字檔案收據，並將此檔案放入 Amazon Simple Storage Service (Amazon S3) 儲存貯體。

主題

- Python Lambda 函式程式碼範例
- 處理常式命名慣例
- 使用 Lambda 事件物件
- 存取和使用 Lambda 內容物件
- Python 處理常式的有效處理常式簽章
- 傳回值
- 在處理常式中使用 適用於 Python (Boto3) 的 AWS SDK
- 存取環境變數
- Python Lambda 函數的程式碼最佳實務

Python Lambda 函式程式碼範例

以下 Python Lambda 函式程式碼範例會取得訂單的相關資訊、產生文字檔案收據，並將此檔案放入 Amazon S3 儲存貯體：

Example Python Lambda 函式

```
import json
import os
import logging
import boto3

# Initialize the S3 client outside of the handler
s3_client = boto3.client('s3')

# Initialize the logger
logger = logging.getLogger()
logger.setLevel("INFO")
```

```
def upload_receipt_to_s3(bucket_name, key, receipt_content):
    """Helper function to upload receipt to S3"""

    try:
        s3_client.put_object(
            Bucket=bucket_name,
            Key=key,
            Body=receipt_content
        )
    except Exception as e:
        logger.error(f"Failed to upload receipt to S3: {str(e)}")
        raise

def lambda_handler(event, context):
    """
    Main Lambda handler function
    Parameters:
        event: Dict containing the Lambda function event data
        context: Lambda runtime context
    Returns:
        Dict containing status message
    """
    try:
        # Parse the input event
        order_id = event['Order_id']
        amount = event['Amount']
        item = event['Item']

        # Access environment variables
        bucket_name = os.environ.get('RECEIPT_BUCKET')
        if not bucket_name:
            raise ValueError("Missing required environment variable RECEIPT_BUCKET")

        # Create the receipt content and key destination
        receipt_content = (
            f"OrderID: {order_id}\n"
            f"Amount: ${amount}\n"
            f"Item: {item}"
        )
        key = f"receipts/{order_id}.txt"

        # Upload the receipt to S3
        upload_receipt_to_s3(bucket_name, key, receipt_content)
```

```
        logger.info(f"Successfully processed order {order_id} and stored receipt in S3
bucket {bucket_name}")

        return {
            "statusCode": 200,
            "message": "Receipt processed successfully"
        }

    except Exception as e:
        logger.error(f"Error processing order: {str(e)}")
        raise
```

此 檔案包含以下程式碼區段：

- `import` 區塊：用於納入 Lambda 函數所需的程式碼。
- SDK 用戶端和記錄器的全域初始化：將初始化程式碼包含在處理常式外部，可透過重複使用執行環境來提高函式效能。如需進一步了解，請參閱the section called “Python Lambda 函數的程式碼最佳實務”。
- `def upload_receipt_to_s3(bucket_name, key, receipt_content):` 此為主要 `lambda_handler` 函式呼叫的協助程式函式。
- `def lambda_handler(event, context):` 此為程式碼的主要處理常式函式，其中包含應用程式的主要邏輯。當 Lambda 調用函式處理常式時，Lambda 執行時期會傳遞兩個引數至該函式：其一為包含函式需處理之資料的事件物件，另一為包含函式調用相關資訊的內容物件。

處理常式命名慣例

建立 Lambda 函式時定義的 Lambda 函式處理常式名稱衍生自下列項目：

- Lambda 處理常式函數所在的檔案名稱。
- Python 處理常式函數的名稱。

在上述範例中，如果檔案名稱為 `lambda_function.py`，則處理常式會指定為 `lambda_function.lambda_handler`。這是使用 Lambda 主控台建立函式時，預設指定的處理常式名稱。

如果要在主控台中使用不同檔案名稱或函數處理常式名稱建立函數，您必須編輯預設處理常式名稱。

變更函數處理常式名稱的方式 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇您的函數。
2. 選擇 程式碼 索引標籤。
3. 向下捲動至執行時間設定窗格，並選擇編輯。
4. 在處理常式中，輸入函數處理常式的新名稱。
5. 選擇儲存。

使用 Lambda 事件物件

當 Lambda 調用函式時，會將事件物件引數傳遞至函式處理常式。JSON 物件是 Lambda 函式最常見的事件格式。在上一節的程式碼範例中，函式預期輸入格式如下：

```
{
    "Order_id": "12345",
    "Amount": 199.99,
    "Item": "Wireless Headphones"
}
```

如果函式由另一個 AWS 服務調用，則輸入事件也是 JSON 物件。事件物件的確切格式取決於調用函式的服務。若要檢視特定服務的事件格式，請參閱整合其他服務章節中的對應頁面。

如果輸入事件採用 JSON 物件的形式，Lambda 執行時期會將物件轉換為 Python 字典。若要將輸入 JSON 中的值指派給程式碼中的變數，請如程式碼範例所示，採用標準的 Python 字典方法。

您也可以將資料作為 JSON 陣列或任何其他有效的 JSON 資料類型傳遞至函式。下表定義了 Python 執行時期如何轉換這些 JSON 類型。

JSON 資料類型	Python 資料類型
object	字典 (`dict`)
陣列	清單 (`list`)
number	整數 (`int`) 或浮點數 (`float`)
string	字串 (`str`)
Boolean	布林值 (`bool`)

JSON 資料類型	Python 資料類型
null	空類型 (NoneType)

存取和使用 Lambda 內容物件

Lambda 內容物件包含有關函式調用和執行環境的資訊。Lambda 在調用時會自動將內容物件傳遞至函式。您可以使用內容物件，基於監控目的，輸出函數調用的資訊。

內容物件是在 Lambda 執行時期介面用戶端中定義的 Python 類別。若要傳回任何內容物件屬性的值，請使用內容物件的對應方法。例如，下列程式碼片段會將 `aws_request_id` 屬性 (調用請求的識別碼) 的值指派給名為 `request` 的變數。

```
request = context.aws_request_id
```

若要進一步了解如何使用 Lambda 內容物件，以及檢視可用方法與屬性的完整清單，請參閱the section called “Context”。

Python 處理常式的有效處理常式簽章

在 Python 中定義處理常式函式時，函式必須採用兩個引數。第一個引數是 Lambda 事件物件，第二個引數是 Lambda 內容物件。按照慣例，這些輸入引數通常命名為 `event` 和 `context`，但您可以隨意為其命名。如果您宣告的處理常式函式只有一個輸入引數，Lambda 會在嘗試執行函式時引發錯誤。在 Python 中宣告處理常式函式最常見的方式如下：

```
def lambda_handler(event, context):
```

您也可以在函式宣告中使用 Python 類型提示，如下例所示：

```
from typing import Dict, Any

def lambda_handler(event: Dict[str, Any], context: Any) -> Dict[str, Any]:
```

若要針對其他 AWS 服務產生的事件以及內容物件使用特定的 AWS 類型，請將 `aws-lambda-typing` 套件新增至函式的部署套件。您可以透過執行 **`pip install aws-lambda-typing`** 在開發環境中安裝此程式庫。下列程式碼片段說明如何了使用 AWS 特定的類型提示。在此範例中，預期事件為 Amazon S3 事件。

```
from aws_lambda_typing.events import S3Event
from aws_lambda_typing.context import Context
from typing import Dict, Any

def lambda_handler(event: S3Event, context: Context) -> Dict[str, Any]:
```

您無法將 Python `async` 函式類型用於處理常式函式。

傳回值

處理常式可以選擇性地傳回值，且該值必須可進行 JSON 序列化。常見的傳回類型包括 `dict`、`list`、`str`、`int`、`float` 和 `bool`。

傳回值的情況取決於調用該函數的調用類型和服務。例如：

- 如果使用 `RequestResponse` 調用類型同步調用 Lambda 函式，Lambda 會將 Python 函式呼叫結果傳回給調用 Lambda 函式的用戶端；該結果會出現在調用請求的 HTTP 回應中，且序列化為 JSON 格式。例如，AWS Lambda 主控台使用 `RequestResponse` 調用類型。因此，當您在主控台上調用函數時，主控台即會顯示傳回值。
- 如果處理常式傳回 `json.dumps` 無法序列化的物件，則執行時間會傳回錯誤。
- 如果處理常式傳回 None (如沒有 `return` 陳述式的 Python 函數隱含作業)，則執行時間會傳回 `null`。
- 如果使用 `Event` 調用類型 (非同步調用)，便會捨棄該值。

在範例程式碼中，處理常式會傳回下列 Python 字典：

```
{
  "statusCode": 200,
  "message": "Receipt processed successfully"
}
```

Lambda 執行時期會將此字典序列化，並將其作為 JSON 字串傳回給調用該函式的用戶端。

> **Note**
>
> 在 Python 3.9 及更高版本中，Lambda 會在錯誤回應中包含調用的請求 ID。

在處理常式中使用 適用於 Python (Boto3) 的 AWS SDK

通常，您會使用 Lambda 函式與其他 AWS 服務和資源互動。與這些資源互動的最簡單方法便是使用適用於 Python (Boto3) 的 AWS SDK。所有支援的 Lambda Python 執行時期都包含適用於 Python 的 SDK 版本。但是，如果程式碼需要使用 SDK，強烈建議在函式的部署套件中包含該 SDK。在部署套件中包含 SDK 可讓您完全控制相依項，並降低與其他程式庫發生版本不一致問題的風險。如需進一步了解，請參閱 the section called “Python 中的執行期相依項” 和 the section called “回溯相容性”。

若要在 Lambda 函式中使用適用於 Python 的 SDK，請將下列陳述式新增至函式程式碼開頭的 import 區塊：

```
import boto3
```

使用 `pip install` 命令將 `boto3` 程式庫新增至函式的部署套件。如需如何將相依項新增至 .zip 部署套件的詳細說明，請參閱the section called “建立含相依項的 .zip 部署套件”。若要進一步了解如何將相依項新增至部署為容器映像的 Lambda 函式，請參閱the section called “從基礎映像建立映像”或the section called “使用替代基礎映像建立映像”。

在程式碼中使用 `boto3` 時，無需提供任憑證即可初始化用戶端。例如，在範例程式碼中，我們使用下列程式碼行來初始化 Amazon S3 用戶端：

```
# Initialize the S3 client outside of the handler
s3_client = boto3.client('s3')
```

對於 Python，Lambda 會自動使用憑證來建立環境變數。`boto3` SDK 會在初始化期間檢查函式的環境變數是否包含這些憑證。

存取環境變數

在處理常式程式碼中，您可以使用 `os.environ.get` 方法來引用環境變數。在範例程式碼中，我們使用下列程式碼行來引用定義的 `RECEIPT_BUCKET` 環境變數：

```
# Access environment variables
bucket_name = os.environ.get('RECEIPT_BUCKET')
```

不要忘記在程式碼開頭的 import 區塊中包含 `import os` 陳述式。

Python Lambda 函數的程式碼最佳實務

請遵循下列清單中的準則，在建置 Lambda 函數時使用最佳編碼實務：

- 區隔 Lambda 處理常式與您的核心邏輯。能允許您製作更多可測單位的函式。例如，在 Python 中，這可能看起來像是：

```
def lambda_handler(event, context):
    foo = event['foo']
    bar = event['bar']
    result = my_lambda_function(foo, bar)

def my_lambda_function(foo, bar):
    // MyLambdaFunction logic here
```

- 控制函數部署套件內的相依性。AWS Lambda 執行環境包含多個程式庫。對於 Node.js 和 Python 執行時期，其中包括 AWS SDK。若要啟用最新的一組功能與安全更新，Lambda 會定期更新這些程式庫。這些更新可能會為您的 Lambda 函數行為帶來細微的變更。若要完全掌控您函式所使用的相依性，請利用部署套件封裝您的所有相依性。
- 最小化依存項目的複雜性。偏好更簡易的框架，其可快速在執行環境啟動時載入。
- 將部署套件最小化至執行時期所必要的套件大小。這能減少您的部署套件被下載與呼叫前解壓縮的時間。

請利用執行環境重新使用來改看函式的效能。在函式處理常式之外初始化 SDK 用戶端和資料庫連線，並在本機快取 `/tmp` 目錄中的靜態資產。由您函式的相同執行個體處理的後續叫用可以重複使用這些資源。這可藉由減少函數執行時間來節省成本。

若要避免叫用間洩漏潛在資料，請不要使用執行環境來儲存使用者資料、事件，或其他牽涉安全性的資訊。如果您的函式依賴無法存放在處理常式內記憶體中的可變狀態，請考慮為每個使用者建立個別函式或個別函式版本。

使用 Keep-Alive 指令維持持續連線的狀態。Lambda 會隨著時間的推移清除閒置連線。叫用函數時嘗試重複使用閒置連線將導致連線錯誤。若要維護持續連線，請使用與執行階段相關聯的 keep-alive (保持啟用) 指令。如需範例，請參閱在 Node.js 中重複使用 Keep-Alive 的連線。

使用環境變數將操作參數傳遞給您的函數。例如，如果您正在寫入到 Amazon S3 儲存貯體，而非對您正在寫入的儲存貯體名稱進行硬式編碼，請將儲存貯體名稱設定為環境變數。

避免在 Lambda 函數中使用遞迴調用，其中函數會調用自己或啟動可能再次調用函數的程序。這會導致意外的函式呼叫量與升高的成本。若您看到意外的調用數量，當更新程式碼時，請立刻將函數的預留並行設為 `0`，以調節對函數的所有調用。

請勿在您的 Lambda 函數程式碼中使用未記錄的非公有 API。對於 AWS Lambda 受管執行時間，Lambda 會定期將安全性和函數更新套用至 Lambda 的內部 API。這些內部 API 更新可能是向後

不相容的，這會導致意外結果，例如若您的函數依賴於這些非公有 API，則叫用失敗。請參閱 API 參考查看公開可用 API 的清單。

撰寫等冪程式碼。為函數撰寫等冪程式碼可確保採用相同方式來處理重複事件。程式碼應正確驗證事件並正常處理重複的事件。如需詳細資訊，請參閱 How do I make my Lambda function idempotent? (如何讓 Lambda 函數等冪？)。

使用 .zip 封存檔部署 Python Lambda 函數

AWS Lambda 函數的程式碼包含一個 .py 檔案，其中包含函數的處理常式程式碼，以及程式碼依賴的任何其他套件和模組。若要將此函數程式碼部署到 Lambda，您可以使用部署套件。此套件可以是 .zip 封存檔或容器映像。如需搭配 Python 使用容器映像檔的詳細資訊，請參閱使用容器映像部署 Python Lambda 函數。

若要建立 .zip 封存檔的部署套件，您可以使用命令列工具的內建 .zip 封存檔公用程式，或任何其他 .zip 檔案公用程式 (例如 7zip)。以下各節顯示的範例假設您在 Linux 或 MacOS 環境中使用命令列 `zip` 工具。若要在 Windows 中使用相同命令，您可以安裝適用於 Linux 的 Windows 子系統，以取得 Ubuntu 和 Bash 的 Windows 整合版本。

請注意，Lambda 使用 POSIX 檔案許可，因此在建立 .zip 封存檔之前，您可能需要設定部署套件資料夾的許可。

主題

- Python 中的執行期相依項
- 建立不含相依項的 .zip 部署套件
- 建立含相依項的 .zip 部署套件
- 相依項搜尋路徑和含執行期程式庫
- 使用 __pycache__ 資料夾
- 建立含原生程式庫的 .zip 部署套件
- 使用 .zip 檔案建立及更新 Python Lambda 函數

Python 中的執行期相依項

對於使用 Python 執行期的 Lambda 函數，相依項可以是任何 Python 套件或模組。使用 .zip 封存部署函數時，您可以使用函數程式碼將這些相依項新增至 .zip 檔案，或使用 Lambda 層。 圖層是單獨的 .zip 檔案，可以包含其他程式碼和內容。若要進一步了解如何使用以 Python 編寫的 Lambda 層，請參閱the section called “層”。

Lambda Python 執行時間包含 適用於 Python (Boto3) 的 AWS SDK 及其相依性。Lambda 會在執行期為您無法新增相依項的部署案例提供 SDK。這些案例包括使用內建程式碼編輯器在主控台中建立函數，或使用 AWS Serverless Application Model (AWS SAM) 或 CloudFormation 範本中的內嵌函數。

Lambda 會定期更新 Python 執行期中的程式庫，以納入最新的更新和安全性修補程式。如果您的函數使用執行階段中包含的 Boto3 SDK 版本，但您的部署套件包含 SDK 相依項，則可能會導致版本不相

符問題。例如，您的部署套件可能包含 SDK 相依項 urllib3。Lambda 在執行期更新 SDK 時，執行期新版本與部署套件中 urllib3 版本之間的相容性問題可能會導致函數失敗。

> **Important**
>
> 為了保持對相依項的完全控制，並避免可能發生的版本不相容問題，建議您將函數的所有相依項新增至部署套件，即使這些相依項的版本包含在 Lambda 執行期中。這包括 Boto3 SDK。

若要了解您使用的執行時期中包含哪個版本的 SDK for Python (Boto3)，請參閱the section called “包含執行時期的 SDK 版本”。

在 AWS 共同責任模式下，您負責管理函數部署套件中的任何相依項。這包括套用更新和安全性修補程式。若要更新函數部署套件中的相依項，請先建立新的 .zip 檔案，然後將其上傳至 Lambda。如需詳細資訊，請參閱 建立含相依項的 .zip 部署套件 和 使用 .zip 檔案建立及更新 Python Lambda 函數。

建立不含相依項的 .zip 部署套件

如果您的函數程式碼沒有相依項，則 .zip 檔案只會包含具有函數處理常式程式碼的 .py 檔案。使用您慣用的 zip 公用程式建立 .zip 檔案，並將 .py 檔案放在根目錄下。如果 .py 檔案不在 .zip 檔案的根目錄下，Lambda 將無法執行您的程式碼。

若要了解如何部署 .zip 檔案以建立新的 Lambda 函數或更新現有函數，請參閱使用 .zip 檔案建立及更新 Python Lambda 函數。

建立含相依項的 .zip 部署套件

如果您的函數程式碼相依於其他套件或模組，您可以使用函數程式碼將這些相依項新增至 .zip 檔案，或使用 Lambda 層。本節中的指示說明如何在 .zip 部署套件中包含相依項。若要讓 Lambda 執行程式碼，您必須將含有處理常式程式碼和所有函數相依性的 .py 檔案安裝在 .zip 檔案的根目錄。

假設您的函數代碼儲存在名為 `lambda_function.py` 的檔案中。下列範例 CLI 命令會建立名為 `my_deployment_package.zip` 的 .zip 檔案，其中包含函數程式碼及其相依項。您可以將相依項直接安裝到專案目錄中的資料夾，也可以使用 Python 虛擬環境。

若要建立部署套件 (專案目錄)

1. 導覽至包含 `lambda_function.py` 原始程式碼檔案的專案目錄。在此範例中，目錄名為 `my_function`。

```
cd my_function
```

2. 建立名為 package 的新目錄，您將在其中安裝相依項。

```
mkdir package
```

請注意，對於 .zip 部署套件，Lambda 預期您的原始程式碼及其相依項全部位於 .zip 檔案的根目錄。不過，直接在專案目錄中安裝相依項可能會產生大量新檔案和資料夾，並使導覽 IDE 變得困難。您可以在此處建立單獨的 package 目錄，將您的相依項與原始程式碼分開。

3. 在 package 目錄中安裝您的相依項。以下範例使用 pip 從 Python Package Index 安裝 Boto3 SDK。如果您的函數程式碼使用您自己建立的 Python 套件，請將其儲存在 package 目錄中。

```
pip install --target ./package boto3
```

4. 建立 .zip 檔案，將已安裝的程式庫放在根目錄下。

```
cd package
zip -r ../my_deployment_package.zip .
```

這會在專案目錄中產生 my_deployment_package.zip 檔案。

5. 將 lambda_function.py 檔案新增至 .zip 檔案的根目錄

```
cd ..
zip my_deployment_package.zip lambda_function.py
```

您的 .zip 檔案應具有扁平的目錄結構，並將函數的處理常式程式碼和所有相依項資料夾安裝在根目錄下，如下所示。

```
my_deployment_package.zip
|- bin
|  |-jp.py
|- boto3
|  |-compat.py
|  |-data
|  |-docs
...
|- lambda_function.py
```

如果包含函數處理常式程式碼的 .py 檔案不在 .zip 檔案的根目錄下，Lambda 將無法執行您的程式碼。

若要建立部署套件 (虛擬環境)

1. 在您的專案目錄中建立並啟用虛擬環境。在此範例中，專案目錄名為 my_function。

```
~$ cd my_function
~/my_function$ python3.14 -m venv my_virtual_env
~/my_function$ source ./my_virtual_env/bin/activate
```

2. 使用 pip 安裝所需的程式庫。以下範例會安裝 Boto3 SDK

```
(my_virtual_env) ~/my_function$ pip install boto3
```

3. 使用 pip show 在在虛擬環境中找出 pip 安裝相依項的位置。

```
(my_virtual_env) ~/my_function$ pip show <package_name>
```

pip 安裝程式庫的資料夾可以命名為 site-packages 或 dist-packages。此資料夾可以位於 lib/python3.x 或 lib64/python3.x 目錄下 (其中 python3.x 代表您所使用的 Python 版本)。

4. 停用虛擬環境

```
(my_virtual_env) ~/my_function$ deactivate
```

5. 導覽至包含您使用 pip 所安裝相依項的目錄，在專案目錄中建立 .zip 檔案，並將安裝的相依項放在根目錄下。在此範例中，pip 已經在 my_virtual_env/lib/python3.14/site-packages 目錄中安裝了您的相依項。

```
~/my_function$ cd my_virtual_env/lib/python3.14/site-packages
~/my_function/my_virtual_env/lib/python3.14/site-packages$ zip -r ../../../../
my_deployment_package.zip .
```

6. 導覽至包含處理常式程式碼的 .py 檔案所在專案目錄的根目錄，並將該檔案新增至 .zip 套件的根目錄。在此範例中，您的函數程式碼檔案名稱為 lambda_function.py。

```
~/my_function/my_virtual_env/lib/python3.14/site-packages$ cd ../../../../
```

```
~/my_function$ zip my_deployment_package.zip lambda_function.py
```

相依項搜尋路徑和含執行期程式庫

當您在程式碼中使用 `import` 陳述式時，Python 執行期會在其搜尋路徑中搜尋目錄，直到找到模組或套件為止。依預設，執行期搜尋的第一個位置是 .zip 部署套件解壓縮並掛載的目錄 (`/var/task`)。如果您在部署套件中納入含執行期程式庫的版本，則您的版本的優先順序會高於執行期中包含的版本。部署套件中的相依項也優先於圖層中的相依項。

當您將相依項新增至層時，Lambda 會將其擷取到 `/opt/python/lib/python3.x/site-packages`(其中 `python3.x` 表示您所使用的執行期版本) 或 `/opt/python`。在搜尋路徑中，這些目錄的優先順序會高於包含含執行期程式庫和使用 pip 安裝的程式庫的目錄 (`/var/runtime` 和 `/var/lang/lib/python3.x/site-packages`)。因此，函數層中程式庫的優先順序高於執行期中包含的版本。

> **Note**
>
> 在 Python 3.11 受管執行期和基礎映像中，軟體 AWS 開發套件及其相依性會安裝在 `/var/lang/lib/python3.11/site-packages`目錄中。

您可以新增下列程式碼片段，以查看 Lambda 函數的完整搜尋路徑。

```
import sys

search_path = sys.path
print(search_path)
```

> **Note**
>
> 由於部署套件或圖層中的相依項優先於含執行期程式庫，因此如果您在套件中包含 urllib3 這類 SDK 相依項而不同時包含 SDK 的話，則可能會造成版本不相容問題。如果您部署自己的 Boto3 相依項版本，則還必須在部署套件中部署 Boto3 作為相依項。我們建議您封裝函數的所有相依項，即使其版本包含在執行期中。

您也可以在 .zip 套件內的個別資料夾中新增相依項。例如，您可以將 Boto3 SDK 的版本新增至名為 `common` 的 .zip 套件中的資料夾。解壓縮並掛載您的 .zip 套件時，此資料夾會放在 `/var/task` 目

錄中。若要在程式碼中使用來自 .zip 部署套件中資料夾的相依項，請使用 `import from` 陳述式。例如，若要使用來自 .zip 套件中名為 common 的資料夾的 Boto3 版本，請使用下列陳述式。

```
from common import boto3
```

使用 __pycache__ 資料夾

我們建議您不要在函數的部署套件中包含 `__pycache__` 資料夾。在架構或作業系統不同的建置機器上編譯的 Python 位元組程式碼可能與 Lambda 執行環境不相容。

建立含原生程式庫的 .zip 部署套件

如果您的函數只使用純 Python 套件和模組，您可以使用 `pip install` 命令在任何本機建置機器上安裝相依項，並建立 .zip 檔案。許多主流 Python 程式庫 (包括 NumPy 和 Pandas) 都不是純 Python，且包含以 C 或 C++ 編寫的程式碼。將包含 C/C++ 程式碼的程式庫新增至部署套件時，必須正確建置套件，以確保套件與 Lambda 執行環境相容。

Python Package Index (PyPI)上提供的大多數套件都可以作為「wheel」(.whl 檔案)。.whl 檔案是一種 ZIP 檔案，其中的內建發佈包含針對特定作業系統和指令集架構預先編譯的二進位檔。若要讓您的部署套件與 Lambda 相容，請安裝適用於 Linux 作業系統和您函數指令集架構的 wheel。

某些套件可能只能作為原始檔發佈。對於這些套件，您需要自行編譯和建置 C/C++ 元件。

若要查看所需套件可用的發佈，請執行以下操作：

1. 在 Python Package Index 主頁上搜尋套件名稱。
2. 選擇您要使用的套件版本。
3. 選擇下載檔案。

使用內建發佈 (wheel)

若要下載與 Lambda 相容的 wheel，請使用 pip `--platform` 選項。

如果您的 Lambda 函數使用 x86_64 指令集架構，請執行下列 `pip install` 命令，在 `package` 目錄中安裝相容的 wheel。以您所使用的 Pytho 執行期版本取代 `--python 3.x`。

```
pip install \
--platform manylinux2014_x86_64 \
--target=package \
```

```
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

如果您的函數使用 arm64 指令集架構，請執行下列命令。以您所使用的 Pytho 執行期版本取代 `--python 3.x`。

```
pip install \
--platform manylinux2014_aarch64 \
--target=package \
--implementation cp \
--python-version 3.x \
--only-binary=:all: --upgrade \
<package_name>
```

使用原始檔發佈

如果您的套件只能作為原始檔發佈，則需要自行建置 C/C++ 程式庫。若要讓套件與 Lambda 執行環境相容，須在使用相同 Amazon Linux 作業系統的環境中建置套件。您可以在 Amazon Elastic Compute Cloud (Amazon EC2) Linux 執行個體中建置套件來執行此操作。

若要了解如何啟動並連線 Amazon EC2 Linux 執行個體，請參閱 Amazon EC2 User Guide 中的 Get started with Amazon EC2。

使用 .zip 檔案建立及更新 Python Lambda 函數

建立 .zip 部署套件後，您可以使用該套件建立新的 Lambda 函數或更新現有函數。您可以使用 Lambda 主控台、 AWS Command Line Interface和 Lambda API 部署 .zip 套件。您也可以使用 AWS Serverless Application Model (AWS SAM) 和 CloudFormation 建立並更新 Lambda 函數。

Lambda 的 .zip 部署套件大小上限為 250 MB (解壓縮)。請注意，此限制適用於您上傳的所有檔案 (包括任何 Lambda 層) 的大小總和。

Lambda 執行時間需有許可才能讀取部署套裝服務中的檔案。在 Linux 許可八進位標記法中，Lambda 需要 644 個許可 (rw-r--r--) 用於非可執行檔，以及 755 個許可 (rwxr-x) 用於目錄和可執行檔。

在 Linux 和 MacOS 中，使用 `chmod` 命令變更部署套件中檔案和目錄的檔案許可。例如，若要為非可執行檔提供正確的許可，請執行下列命令。

```
chmod 644 <filepath>
```

若要在 Windows 中變更檔案許可，請參閱 Microsoft Windows 文件的 Set, View, Change, or Remove Permissions on an Object。

> **Note**
>
> 如果您未授予 Lambda 存取部署套件中的目錄所需的許可，Lambda 會將這些目錄的許可設定為 755 (rwxr-xr-x)。

透過主控台使用 .zip 檔案建立及更新函數

若要建立新函數，您必須先在主控台中建立函數，然後上傳您的 .zip 封存檔。若要更新現有函數，請開啟函數的頁面，然後按照同樣的程序新增更新後的 .zip 檔案。

如果您的 .zip 檔案小於 50 MB，您可以透過直接從本機電腦上傳檔案來建立或更新函數。若 .zip 檔案大於 50 MB，您必須先將套件上傳至 Amazon S3 儲存貯體。如需如何使用 將檔案上傳至 Amazon S3 儲存貯體的說明 AWS 管理主控台，請參閱 Amazon S3 入門。若要使用 上傳檔案 AWS CLI，請參閱AWS CLI 《 使用者指南》中的移動物件。

> **Note**
>
> 不能變更現有函數的部署套件類型 (.zip 或容器映像)。例如，您不能轉換容器映像函數以使用 .zip 封存檔。您必須建立新的函數。

若要建立新的函數 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇建立函數。
2. 選擇 Author from scratch (從頭開始撰寫)。
3. 在基本資訊下，請執行下列動作：
   a. 在函數名稱中輸入函數名稱。
   b. 在執行期中選取要使用的執行期。
   c. (選用) 在架構中選擇要用於函數的指令集架構。預設架構值為 x86_64。請確定函數的 .zip 部署套件與您選取的指令集架構相容。
4. (選用) 在許可下，展開變更預設執行角色。您可建立新的執行角色，或使用現有的角色。
5. 選擇 Create function (建立函數)。Lambda 會使用您選擇的執行期建立一個基本的「Hello world」函數。

若要從本機電腦上傳 .zip 封存檔 (主控台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 .zip 檔案。
5. 若要上傳 .zip 檔案，請執行下列操作：
   a. 選擇上傳，然後在檔案選擇器中選取您的 .zip 檔案。
   b. 選擇 Open (開啟)。
   c. 選擇儲存。

若要從 Amazon S3 儲存貯體上傳 .zip 封存檔 (控制台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳新 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 Amazon S3 位置。
5. 貼上 .zip 檔案的 Amazon S3 連結 URL，然後選擇儲存。

使用主控台程式碼編輯器更新 .zip 檔案函數

對於某些具有 .zip 部署套件的函數，您可以使用 Lambda 主控台的內建程式碼編輯器直接更新函數程式碼。若要使用此功能，您的函數必須符合下列條件：

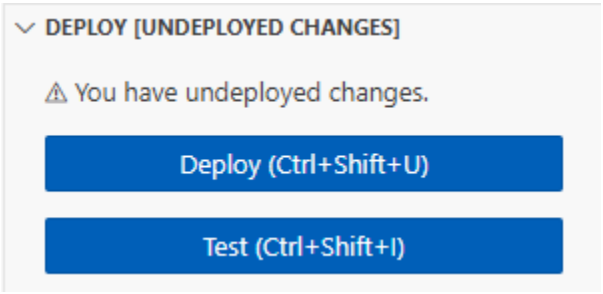
- 您的函數必須使用其中一種轉譯語言執行期 (Python、Node.js 或 Ruby)
- 函數的部署套件必須小於 50 MB (未壓縮)。

具有容器映像部署套件之函數的函數程式碼無法直接在主控台中編輯。

若要使用主控台程式碼編輯器更新函數程式碼

1. 開啟 Lambda 主控台的函數頁面，然後選取您的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中，選取您的原始程式碼檔案，然後在整合式程式碼編輯器中加以編輯。

4. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



使用 建立和更新具有 .zip 檔案的函數 AWS CLI

您可以使用 AWS CLI 建立新函數，或使用 .zip 檔案更新現有函數。使用 create-function 和 update-function-code 命令來部署您的 .zip 套件。如果您的 .zip 檔案小於 50 MB，則可以從本機建置電腦的檔案位置上傳 .zip 套件。若檔案較大，則必須先從 Amazon S3 儲存貯體上傳 .zip 套件。如需如何使用將檔案上傳至 Amazon S3 儲存貯體的說明 AWS CLI，請參閱AWS CLI 《 使用者指南》中的移動物件。

> **Note**
>
> 如果您使用 從 Amazon S3 儲存貯體上傳 .zip 檔案 AWS CLI，儲存貯體必須與 AWS 區域 函數位於相同的 中。

若要搭配 使用 .zip 檔案建立新的函數 AWS CLI，您必須指定下列項目：

- 函數名稱 (`--function-name`)
- 函數的執行期 (`--runtime`)
- 函數執行角色的 Amazon Resource Name (ARN) (`--role`)
- 函數程式碼中處理常式方法的名稱 (`--handler`)

您也必須指定 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime python3.14 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--code` 選項。您只需針對版本控制的物件使用 `S3ObjectVersion` 參數。

```
aws lambda create-function --function-name myFunction \
--runtime python3.14 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

若要使用 CLI 更新現有函數，您可以使用 `--function-name` 參數指定函數的名稱。您也必須指定要用來更新函數程式碼的 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--s3-bucket` 和 `--s3-key` 選項。您只需針對版本控制的物件使用 `--s3-object-version` 參數。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

透過 Lambda API 使用 .zip 檔案建立及更新函數

若要使用 .zip 封存檔建立及更新函數，請使用下列 API 操作：

- CreateFunction
- UpdateFunctionCode

使用 建立和更新具有 .zip 檔案的函數 AWS SAM

AWS Serverless Application Model (AWS SAM) 是一種工具組，可協助簡化建置和執行無伺服器應用程式的程序 AWS。您可以在 YAML 或 JSON 範本中定義應用程式的資源，並使用 AWS SAM 命令列界面 (AWS SAM CLI) 來建置、封裝和部署應用程式。當您從 AWS SAM 範本建置 Lambda 函數時，AWS SAM 會自動使用函數程式碼和您指定的任何相依性建立 .zip 部署套件或容器映像。若要進一步了解如何使用 AWS SAM 來建置和部署 Lambda 函數，請參閱《AWS Serverless Application Model 開發人員指南》中的 入門 AWS SAM。

您也可以使用 AWS SAM 來建立使用現有 .zip 檔案封存的 Lambda 函數。若要使用 建立 Lambda 函數 AWS SAM，您可以將 .zip 檔案儲存在 Amazon S3 儲存貯體或建置機器的本機資料夾中。如需如何使用 將檔案上傳至 Amazon S3 儲存貯體的說明 AWS CLI，請參閱AWS CLI 《 使用者指南》中的移動物件。

在您的 AWS SAM 範本中，`AWS::Serverless::Function`資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `CodeUri`：設定為函數程式碼的 Amazon S3 URI、本機資料夾的路徑或 FunctionCode 物件
- `Runtime`：設定為所選執行期

使用 時 AWS SAM，如果您的 .zip 檔案大於 50MB，則不需要先將其上傳至 Amazon S3 儲存貯體。AWS SAM 可以從本機建置機器的位置上傳最大允許大小為 250MB （ 解壓縮 ） 的 .zip 套件。

若要進一步了解如何在 中使用 .zip 檔案部署函數 AWS SAM，請參閱《 AWS SAM 開發人員指南》中的 AWS::Serverless::Function。

使用 建立和更新具有 .zip 檔案的函數 CloudFormation

您可以使用 CloudFormation 建立使用 .zip 檔案封存的 Lambda 函數。若要使用 .zip 檔案建立 Lambda 函數，您必須先將檔案上傳至 Amazon S3 儲存貯體。如需如何使用 將檔案上傳至 Amazon S3 儲存貯體的說明 AWS CLI，請參閱《 使用者指南》中的移動物件。 AWS CLI

對於 Node.js 和 Python 執行期，您也可以在 CloudFormation 範本中提供內嵌原始程式碼。CloudFormation 然後，當您建置函數時， 會建立包含程式碼的 .zip 檔案。

使用現有的 .zip 檔案

在您的 CloudFormation 範本中，`AWS::Lambda::Function`資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `Code`：在 `S3Bucket` 和 `S3Key` 欄位中輸入 Amazon S3 儲存貯體名稱和 .zip 檔案名稱。
- `Runtime`：設定為所選執行期

從內嵌程式碼建立 .zip 檔案

您可以在 CloudFormation 範本中宣告以 Python 或 Node.js 內嵌撰寫的簡單函數。由於程式碼內嵌在 YAML 或 JSON 中，因此您無法將任何外部相依項新增至部署套件。這表示您的函數必須使用 執行時

間中包含的 AWS SDK 版本。範本的需求 (例如必須逸出某些字元) 也會讓使用 IDE 的語法檢查和程式碼完成功能變得更加困難。也就是說，您的範本可能需要進行其他測試。由於這些限制，宣告函數內嵌程式碼最適合非常簡單且不會頻繁變更的程式碼。

若要從 Node.js 和 Python 執行期的內嵌程式碼建立 .zip 檔案，請在範本的 `AWS::Lambda::Function` 資源中設定下列屬性：

- `PackageType`：設定為 `Zip`
- `Code`：在 `ZipFile` 欄位中輸入您的函數程式碼
- `Runtime`：設定為所選執行期

CloudFormation 產生的 .zip 檔案不能超過 4MB。若要進一步了解如何在 中使用 .zip 檔案部署函數 CloudFormation，請參閱CloudFormation 《 使用者指南》中的 AWS::Lambda::Function。

使用容器映像部署 Python Lambda 函數

您可以透過三種方式為 Python Lambda 函數建置容器映像：

- 使用 Python AWS 的基礎映像

  AWS 基礎映像會預先載入語言執行期、用來管理 Lambda 與函數程式碼之間互動的執行期界面用戶端，以及用於本機測試的執行期界面模擬器。

- 使用僅限 AWS 作業系統的基礎映像

  AWS 僅限作業系統的基礎映像包含 Amazon Linux 發行版本和執行時間界面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Python 的執行期介面用戶端。

- 使用非AWS 基礎映像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Python 的執行期介面用戶端。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

本頁面會說明如何為 Lambda 建置、測試和部署容器映像。

主題

- AWS Python 的基礎映像
- 使用 Python AWS 的基礎映像
- 透過執行期介面用戶端使用替代基礎映像

AWS Python 的基礎映像

AWS 為 Python 提供下列基礎映像：

Tags (標籤)	執行期	作業系統	Dockerfile	棄用
3.14	Python 3.14	Amazon Linux 2023	GitHub 上適用於 Python 3.14 的 Dockerfile	2029 年 6 月 30 日
3.13	Python 3.13	Amazon Linux 2023	Dockerfile for Python 3.13 on GitHub	2029 年 6 月 30 日
3.12	Python 3.12	Amazon Linux 2023	Dockerfile for Python 3.12 on GitHub	2028 年 10 月 31 日
3.11	Python 3.11	Amazon Linux 2	Dockerfile for Python 3.11 on GitHub	2027 年 6 月 30 日
3.10	Python 3.10	Amazon Linux 2	Dockerfile for Python 3.10 on GitHub	2026 年 10 月 31 日

Amazon ECR 儲存庫：gallery.ecr.aws/lambda/python

Python 3.12 和更新版本的基礎映像以 Amazon Linux 2023 最小容器映像為基礎。Python 3.8-3.11 基礎映像以 Amazon Linux 2 映像為基礎。與 Amazon Linux 2 相比，以 AL2023 為基礎的映像具有多項優點，包括更小的部署足跡和更新版本的程式庫，如 `glibc`。

以 AL2023 為基礎的映像使用 `microdnf` (符號連結為 `dnf`) 而不是 `yum` 作為套件管理工具，後者是 Amazon Linux 2 中的預設套件管理工具。`microdnf` 是 `dnf` 的獨立實作。對於以 AL2023 為基礎的映像中包含的套件清單，請參閱 Comparing packages installed on Amazon Linux 2023 Container Images 中的 Minimal Container 欄。如需 AL2023 和 Amazon Linux 2 之間差異的詳細資訊，請參閱 AWS 運算部落格上的 Introducing the Amazon Linux 2023 runtime for AWS Lambda。

> **Note**
>
> 若要在本機執行AL2023-based映像，包括搭配 AWS Serverless Application Model (AWS SAM)，您必須使用 Docker 20.10.10 版或更新版本。

基礎映像中的相依性搜尋路徑

當您在程式碼中使用 `import` 陳述式時，Python 執行期會在其搜尋路徑中搜尋目錄，直到找到模組或套件為止。在預設情況下，執行期會先搜尋 `{LAMBDA_TASK_ROOT}` 目錄。如果您在映像中納入含執行期程式庫的版本，則此版本的優先順序會高於執行期中包含的版本。

搜尋路徑中包含的其他步驟取決於您使用的 Python Lambda 基礎映像版本：

- Python 3.11 及更高版本：含執行期的程式庫和使用 pip 安裝的程式庫已安裝在 `/var/lang/lib/python3.11/site-packages` 目錄。此目錄的優先順序會高於搜尋路徑中的 `/var/runtime`。您可以使用 pip 安裝更新的版本來覆寫 SDK。您可以使用 pip 來確認含執行期的 SDK 及其相依性是否與您安裝的任何套件相容。
- Python 3.8-3.10：含執行期的程式庫已安裝在 `/var/runtime` 目錄。使用 pip 安裝的程式庫已安裝在 `/var/lang/lib/python3.x/site-packages` 目錄。`/var/runtime` 目錄的優先順序會高於搜尋路徑中的 `/var/lang/lib/python3.x/site-packages`。

您可以新增下列程式碼片段，以查看 Lambda 函數的完整搜尋路徑。

```
import sys

search_path = sys.path
print(search_path)
```

使用 Python AWS 的基礎映像

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- Docker (最低版本 25.0.0)
- Docker buildx 外掛程式。
- Python

從基礎映像建立映像

從 Python AWS 的基礎映像建立容器映像

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir example
cd example
```

2. 建立稱為 `lambda_function.py` 的新檔案。您可以將下列範例函數程式碼新增至檔案進行測試，或使用您自己的函數程式碼。

   Example Python 函數

```
import sys
def handler(event, context):
    return 'Hello from AWS Lambda using Python' + sys.version + '!'
```

3. 建立稱為 `requirements.txt` 的新檔案。如果您使用上一個步驟的範例函數程式碼，請將檔案保留空白，因為沒有任何相依項。否則，請列出每個所需的程式庫。例如，如果您的函數使用 適用於 Python (Boto3) 的 AWS SDK，您的 `requirements.txt` 看起來應該像這樣：

   Example requirements.txt

```
boto3
```

4. 建立包含下列組態的新 Dockerfile。

   - 將 `FROM` 屬性設定為基礎映像的 URI。
   - 使用 `COPY` 命令將函數程式碼和執行時期相依項複製到 `{LAMBDA_TASK_ROOT}`，一個 Lambda 定義的環境變數。
   - 將 `CMD` 引數設定為 Lambda 函數處理常式。

   請注意，範例 Dockerfile 不包含 USER 指令。當您將容器映像部署到 Lambda 時，Lambda 會自動定義一個具有最低權限許可的預設 Linux 使用者。這與標準 Docker 行為不同，後者會在未提供 `USER` 指令時預設為 `root` 使用者。

   Example Dockerfile

```
FROM public.ecr.aws/lambda/python:3.12
```

```
# Copy requirements.txt
COPY requirements.txt ${LAMBDA_TASK_ROOT}

# Install the specified packages
RUN pip install -r requirements.txt

# Copy function code
COPY lambda_function.py ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.handler" ]
```

5. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 `docker-image` 並為其提供 `test` 標籤。若要使映像與 Lambda 相容，必須使用 `--provenance=false` 選項。

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> 此命令會指定 `--platform linux/amd64` 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 `--platform linux/arm64` 選項。

(選用) 在本機測試映像

1. 使用 docker run 命令啟動 Docker 影像。在此範例中，`docker-image` 為映像名稱，`test` 為標籤。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

此命令將映像作為容器執行，並在 `localhost:9000/2015-03-31/functions/function/invocations` 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將`--platform linux/arm64`選項改用選項。`--platform linux/amd64`

2. 從新的終端機視窗，將事件張貼至本機端點。

   Linux/macOS

   在 Linux 或 macOS 中，執行下列 `curl` 命令：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   在 PowerShell 中，執行下列 `Invoke-WebRequest` 命令：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

3. 取得容器 ID。

   ```
   docker ps
   ```

4. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將 `--region`值設定為您要 AWS 區域 在其中建立 Amazon ECR 儲存庫的 。
   - `111122223333` 以您的 AWS 帳戶 ID 取代 。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於相同的 AWS 區域 中。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
```

```
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - `docker-image:test` 為 Docker 映像檔的名稱和標籤。這是您在 `docker build` 命令中指定的映像名稱和標籤。
   - 將 <ECRrepositoryUri> 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像來建立函數。如需詳細資訊，請參閱Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 會將映像標籤解析為特定映像摘要。這表示如果您將用來部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。

若要將新映像部署至相同的 Lambda 函數，必須使用 update-function-code 命令，即使 Amazon ECR 中的映像標籤保持不變亦如此。在以下範例中，`--publish` 選項會使用更新的容器映像來建立新的函數版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

透過執行期介面用戶端使用替代基礎映像

如果您使用僅限作業系統的基礎映像或替代的基礎映像，則必須在映像中加入執行期介面用戶端。執行期介面用戶端會讓您擴充 Runtime API，管理 Lambda 與函數程式碼之間的互動。

使用 pip 套件管理員安裝 Python 執行期介面用戶端。

```
pip install awslambdaric
```

您還可以從 GitHub 下載 Python 執行時間介面用戶端。

下列範例示範如何使用非AWS 基礎映像建置 Python 的容器映像。範例 Dockerfile 使用官方 Python 基礎映像。Dockerfile 包含 Python 執行期界面用戶端。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- Docker (最低版本 25.0.0)
- Docker buildx 外掛程式。
- Python

使用替代基礎映像建立映像

從非AWS 基礎映像建立容器映像

1. 建立專案的目錄，然後切換至該目錄。

   ```
   mkdir example
   cd example
   ```

2. 建立稱為 `lambda_function.py` 的新檔案。您可以將下列範例函數程式碼新增至檔案進行測試，或使用您自己的函數程式碼。

   Example Python 函數

   ```
   import sys
   def handler(event, context):
       return 'Hello from AWS Lambda using Python' + sys.version + '!'
   ```

3. 建立稱為 `requirements.txt` 的新檔案。如果您使用上一個步驟的範例函數程式碼，請將檔案保留空白，因為沒有任何相依項。否則，請列出每個所需的程式庫。例如，如果您的函數使用 適用於 Python (Boto3) 的 AWS SDK，您的 `requirements.txt` 看起來應該像這樣：

Example requirements.txt

```
boto3
```

4. 建立新的 Dockerfile。下列 Dockerfile 使用官方 Python 基礎映像，而非 AWS 基礎映像。Dockerfile 包含執行期介面用戶端，可讓映像與 Lambda 相容。下列範例 Dockerfile 使用多階段建置。

   - 將 FROM 屬性設定為基礎映像。
   - 將 ENTRYPOINT 設為您希望 Docker 容器在啟動時執行的模組。在此案例中，模組是執行期界面用戶端。
   - 將 CMD 設定為 Lambda 函數處理常式。

   請注意，範例 Dockerfile 不包含 USER 指令。當您將容器映像部署到 Lambda 時，Lambda 會自動定義一個具有最低權限許可的預設 Linux 使用者。這與標準 Docker 行為不同，後者會在未提供 USER 指令時預設為 root 使用者。

   Example Dockerfile

```
# Define custom function directory
ARG FUNCTION_DIR="/function"

FROM python:3.12 AS build-image

# Include global arg in this stage of the build
ARG FUNCTION_DIR

# Copy function code
RUN mkdir -p ${FUNCTION_DIR}
COPY . ${FUNCTION_DIR}

# Install the function's dependencies
RUN pip install \
    --target ${FUNCTION_DIR} \
        awslambdaric

# Use a slim version of the base Python image to reduce the final image size
FROM python:3.12-slim

# Include global arg in this stage of the build
```

```
ARG FUNCTION_DIR
# Set working directory to function root directory
WORKDIR ${FUNCTION_DIR}

# Copy in the built dependencies
COPY --from=build-image ${FUNCTION_DIR} ${FUNCTION_DIR}

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/local/bin/python", "-m", "awslambdaric" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "lambda_function.handler" ]
```

5. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。若要使映像與 Lambda 相容，必須使用 --provenance=false 選項。

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

(選用) 在本機測試映像

使用 執行期界面模擬器 以在本機測試映像。您可以將模擬器建置到映像中，也可以使用以下步驟，將其安裝在本機電腦。

若要在本機電腦上安裝並執行執行期介面模擬器

1. 在您的專案目錄中執行以下命令，從 GitHub 下載執行期介面模擬器 (x86-64 架構)，並安裝在本機電腦上。

   Linux/macOS

```
mkdir -p ~/.aws-lambda-rie && \
    curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-
runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
```

```
    chmod +x ~/.aws-lambda-rie/aws-lambda-rie
```

若要安裝 arm64 模擬器，請將上一個命令中的 GitHub 儲存庫 URL 替換為以下內容：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

PowerShell

```
$dirPath = "$HOME\.aws-lambda-rie"
if (-not (Test-Path $dirPath)) {
    New-Item -Path $dirPath -ItemType Directory
}

$downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/
releases/latest/download/aws-lambda-rie"
$destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

若要安裝 arm64 模擬器，請將 `$downloadLink` 更換為下列項目：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. 使用 docker run 命令啟動 Docker 影像。注意下列事項：

   - `docker-image` 是映像名稱，而 `test` 是標籤。
   - `/usr/local/bin/python -m awslambdaric lambda_function.handler` 是 Dockerfile 中的 `ENTRYPOINT`，後面接著 `CMD`。

   Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        /usr/local/bin/python -m awslambdaric lambda_function.handler
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    /usr/local/bin/python -m awslambdaric lambda_function.handler
```

此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將--platform linux/*arm64*選項改用選項。--platform linux/*amd64*

3. 將事件張貼至本機端點。

Linux/macOS

在 Linux 或 macOS 中，執行下列 curl 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在 PowerShell 中，執行下列 Invoke-WebRequest 命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 取得容器 ID。

```
docker ps
```

5. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將 `--region`值設定為您要 AWS 區域 在其中建立 Amazon ECR 儲存庫的 。
   - 111122223333 以您的 AWS 帳戶 ID 取代 。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於相同的 AWS 區域 中。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - `docker-image:test` 為 Docker 映像檔的名稱和標籤。這是您在 `docker build` 命令中指定的映像名稱和標籤。
   - 將 `<ECRrepositoryUri>` 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像來建立函數。如需詳細資訊，請參閱Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 會將映像標籤解析為特定映像摘要。這表示如果您將用來部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。

若要將新映像部署至相同的 Lambda 函數，必須使用 update-function-code 命令，即使 Amazon ECR 中的映像標籤保持不變亦如此。在以下範例中，--publish 選項會使用更新的容器映像來建立新的函數版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

如需如何從 Alpine 基礎映像中建立 Python 映像的範例，請參閱 AWS 部落格上的Lambda 的容器映像支援。

針對 Python Lambda 函數使用層

使用 Lambda 層封裝要在多個函式之間重複使用的程式碼與相依項。層通常具備程式庫相依性、自訂執行期或組態檔案。建立層包含三個一般步驟：

1. 封裝層內容。這表示建立 .zip 封存檔，其中包含您要在函數中使用的相依項。
2. 在 Lambda 中建立層。
3. 將層新增至函數中。

主題

- 封裝層內容
- 在 Lambda 中建立層
- 將層新增至函式
- 應用程式範例

封裝層內容

若要建立層，請將套件封裝成符合以下要求的 .zip 壓縮檔：

- 使用計劃用於 Lambda 函式的相同 Python 版本來建置層。例如，如果您使用 Python 3.14 建置 layer，請使用適用於函數的 Python 3.14 執行時間。
- .zip 檔案必須在根層級包含 `python` 目錄。
- 層中的套件必須與 Linux 相容。Lambda 函式會在 Amazon Linux 上執行。

您可以建立包含透過 `pip` 安裝的第三方 Python 程式庫 (例如 `requests` 或 `pandas`) 的層，或您自訂的 Python 模組與套件的層。

第三方相依性

使用 pip 套件建立層

1. 選擇下列其中一種方法，將 `pip` 套件安裝到所需的頂層目錄 (`python/`)：

   pip install

   對於純 Python 套件 (如 requests 或 boto3)：

```
pip install requests -t python/
```

部分 Python 套件 (如 NumPy 與 Pandas) 包含經編譯的 C 元件。如果您在 macOS 或 Windows 上建置包含這些套件的層，可能需要使用此命令來安裝與 Linux 相容的 wheel：

```
pip install numpy --platform manylinux2014_x86_64 --only-binary=:all: -t python/
```

如需有關使用包含編譯元件之 Python 套件的詳細資訊，請參閱建立含原生程式庫的 .zip 部署套件。

requirements.txt

使用 `requirements.txt` 檔案可協助您管理套件版本，確保安裝的一致性。

Example requirements.txt

```
requests==2.31.0
boto3==1.37.34
numpy==1.26.4
```

如果 `requirements.txt` 檔案僅包含純 Python 套件 (如 requests 或 boto3)：

```
pip install -r requirements.txt -t python/
```

部分 Python 套件 (如 NumPy 與 Pandas) 包含經編譯的 C 元件。如果您在 macOS 或 Windows 上建置包含這些套件的層，可能需要使用此命令來安裝與 Linux 相容的 wheel：

```
pip install -r requirements.txt --platform manylinux2014_x86_64 --only-
binary=:all: -t python/
```

如需有關使用包含編譯元件之 Python 套件的詳細資訊，請參閱建立含原生程式庫的 .zip 部署套件。

2. 壓縮 `python` 目錄的內容。

Linux/macOS

```
zip -r layer.zip python/
```

PowerShell

```
Compress-Archive -Path .\python -DestinationPath .\layer.zip
```

.zip 檔案的目錄結構應如下所示：

```
python/              # Required top-level directory
### requests/
### boto3/
### numpy/
### (dependencies of the other packages)
```

> **Note**
>
> 如果使用 Python 虛擬環境 (venv) 來安裝套件，目錄結構將會有所不同 (例如 python/lib/python3.*x*/site-packages)。只要 .zip 檔案在根層級包含 python 目錄，Lambda 就能順利找到並匯入套件。

自訂 Python 模組

使用自訂的程式碼建立層

1. 為層建立所需的頂層目錄：

```
mkdir python
```

2. 在 python 目錄中建立 Python 模組。下列模組範例透過確認訂單包含所需資訊來驗證訂單。

Example自訂模組：validator.py

```
import json

def validate_order(order_data):
    """Validates an order and returns formatted data."""
    required_fields = ['product_id', 'quantity']

    # Check required fields
```

```
    missing_fields = [field for field in required_fields if field not in
 order_data]
    if missing_fields:
        raise ValueError(f"Missing required fields: {', '.join(missing_fields)}")

    # Validate quantity
    quantity = order_data['quantity']
    if not isinstance(quantity, int) or quantity < 1:
        raise ValueError("Quantity must be a positive integer")

    # Format and return the validated data
    return {
        'product_id': str(order_data['product_id']),
        'quantity': quantity,
        'shipping_priority': order_data.get('priority', 'standard')
    }

def format_response(status_code, body):
    """Formats the API response."""
    return {
        'statusCode': status_code,
        'body': json.dumps(body)
    }
```

3. 壓縮 python 目錄的內容。

   Linux/macOS

   ```
   zip -r layer.zip python/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\python -DestinationPath .\layer.zip
   ```

   .zip 檔案的目錄結構應如下所示：

   ```
   python/     # Required top-level directory
   ### validator.py
   ```

4. 在函式中，如同使用任何 Python 套件一樣，匯入並使用模組。範例：

```
from validator import validate_order, format_response
import json

def lambda_handler(event, context):
    try:
        # Parse the order data from the event body
        order_data = json.loads(event.get('body', '{}'))

        # Validate and format the order
        validated_order = validate_order(order_data)

        return format_response(200, {
            'message': 'Order validated successfully',
            'order': validated_order
        })
    except ValueError as e:
        return format_response(400, {
            'error': str(e)
        })
    except Exception as e:
        return format_response(500, {
            'error': 'Internal server error'
        })
```

您可以使用下列測試事件來調用函式：

```
{
    "body": "{\"product_id\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

預期回應：

```
{
  "statusCode": 200,
  "body": "{\"message\": \"Order validated successfully\", \"order\": {\"product_id
\": \"ABC123\", \"quantity\": 2, \"shipping_priority\": \"express\"}}"
}
```

在 Lambda 中建立層

您可以使用 AWS CLI 或 Lambda 主控台發佈 layer。

AWS CLI

執行 publish-layer-version AWS CLI 命令來建立 Lambda layer：

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes python3.14
```

相容的執行時期參數為選用參數。指定此參數後，Lambda 會據此在 Lambda 主控台中篩選層。

Console

建立圖層 (主控台)

1. 開啟 Lambda 主控台中的 層頁面。
2. 選擇 建立圖層。
3. 選擇上傳 .zip 檔案，然後上傳先前建立的 .zip 壓縮檔。
4. (選用) 在相容的執行時期欄位中，選擇與用於建置層的 Python 版本相對應的 Python 執行時期。
5. 選擇建立。

將層新增至函式

AWS CLI

若要將 layer 連接至函數，請執行 update-function-configuration AWS CLI 命令。對於 `--layers` 參數，請使用層 ARN。ARN 必須指定版本 (例如 `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`)。如需詳細資訊，請參閱層和層的版本。

```
aws lambda update-function-configuration --function-name my-function --cli-binary-
format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-
layer:1"
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資

訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

Console

將層新增至函式

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函式。
3. 向下捲動至層區段，然後選擇新增層。
4. 在選擇層欄位中，選取自訂層，然後選擇要使用的層。

> **Note**
>
> 如果未在建立層時新增相容的執行時期，此處不會列出層。您可以改為指定層 ARN。

5. 選擇新增。

應用程式範例

如需如何使用 Lambda 層的更多範例，請參閱《 AWS Lambda 開發人員指南 GitHub 儲存庫》中的 layer-python 範例應用程式。此應用程式包含兩個內含 Python 程式庫的層。建立層之後，您可以部署並調用對應的函式，驗證層是否如預期運作。

使用 Lambda 內容物件擷取 Python 函數資訊

當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。此物件提供的方法和各項屬性提供了有關調用、函式以及執行環境的資訊。如需如何將內容物件傳遞至函數處理常式的詳細資訊，請參閱以 Python 定義 Lambda 函數處理常式。

內容方法

- `get_remaining_time_in_millis` - 傳回執行逾時前剩餘的毫秒數。

內容屬性

- `function_name` – Lambda 函數的名稱。
- `function_version` – 函數的版本。
- `invoked_function_arn` - 用於叫用此函數的 Amazon Resource Name (ARN)。指出調用者是否指定版本號或別名。
- `memory_limit_in_mb` - 分配給函數的記憶體數量。
- `aws_request_id` - 調用請求的識別符。
- `log_group_name` - 函數的日誌群組。
- `log_stream_name` - 函數執行個體的記錄串流。
- `identity` - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
  - `cognito_identity_id` - 已驗證的 Amazon Cognito 身分。
  - `cognito_identity_pool_id` - 授權調用的 Amazon Cognito 身分集區。
- `client_context` - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。
  - `client.installation_id`
  - `client.app_title`
  - `client.app_version_name`
  - `client.app_version_code`
  - `client.app_package_name`
  - `custom` - 行動用戶端應用程式所設定的 `dict` 自訂值。
  - `env` - AWS 開發套件所提供的 `dict` 環境資訊。

Powertools for Lambda (Python) 提供了 Lambda 內容物件的介面定義。您可以使用介面定義作為類型提示，或進一步檢查 Lambda 內容物件的結構。如需介面定義，請參閱 GitHub 上的 powertools-lambda-python 儲存庫中的 lambda_context.py。

以下範例顯示記錄著內容資訊的處理常式函式。

Example handler.py

```
import time

def lambda_handler(event, context):
    print("Lambda function ARN:", context.invoked_function_arn)
    print("CloudWatch log stream name:", context.log_stream_name)
    print("CloudWatch log group name:",  context.log_group_name)
    print("Lambda Request ID:", context.aws_request_id)
    print("Lambda function memory limits in MB:", context.memory_limit_in_mb)
    # We have added a 1 second delay so you can see the time remaining in
 get_remaining_time_in_millis.
    time.sleep(1)
    print("Lambda time remaining in MS:", context.get_remaining_time_in_millis())
```

除了上面所列的選項，您也可以將 AWS X-Ray 開發套件用於 在 中檢測 Python 程式碼 AWS Lambda，識別重要的程式碼路徑，追蹤它們的效能，並且擷取要進行分析的資料。

記錄和監控 Python Lambda 函數

AWS Lambda 會自動監控 Lambda 函數，並將日誌項目傳送至 Amazon CloudWatch。您的 Lambda 函數隨附有 CloudWatch Logs 日誌群組，且函數的每一執行個體各有一個日誌串流。Lambda 執行期環境會將每次調用的詳細資訊和函數程式碼的其他輸出，傳送至日誌串流。如需 CloudWatch Logs 的詳細資訊，請參閱將 Lambda 函式日誌傳送至 CloudWatch Logs。

若要從函數程式碼輸出日誌，可以使用內建 `logging` 模組。如需更詳細的項目，您可以使用任何寫入 `stdout` 或 `stderr` 的記錄程式庫。

列印至日誌

若要將基本輸出傳送至日誌，請使用您函數中的 `print` 方法。下列範例會記錄 CloudWatch Logs 日誌群組和串流的值，以及事件物件。

請注意，如果您的函數使用 Python `print` 陳述式輸出日誌，Lambda 只能以純文字格式將日誌輸出傳送到 CloudWatch Logs 日誌。若要擷取結構化 JSON 中的記錄，您必須使用支援的記錄程式庫。如需詳細資訊，請參閱the section called “搭配 Python 使用 Lambda 進階日誌控制項”。

Example lambda_function.py

```
import os
def lambda_handler(event, context):
    print('## ENVIRONMENT VARIABLES')
    print(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    print(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    print('## EVENT')
    print(event)
```

Example記錄輸出

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
/aws/lambda/my-function
2025/08/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 147 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
```

```
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

Python 執行時間會記錄每次調用的 START、END 和 REPORT 行。REPORT 行包含以下資料：

REPORT 行資料欄位

- RequestId - 進行調用的唯一請求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。當調用共用執行環境時，Lambda 會報告所有調用使用的記憶體上限。此行為可能會導致報告值高於預期值。
- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId - 對於追蹤的請求，這是 AWS X-Ray 追蹤 ID。
- SegmentId - 對於追蹤的請求，這是 X-Ray 區段 ID。
- 已取樣 - 對於追蹤的請求，這是取樣結果。

使用記錄程式庫

如需更詳細的日誌，請使用標準程式庫中的 logging 模組，或任何寫入 stdout 或 stderr 的第三方記錄程式庫。

對於支援的 Python 執行期，您可以選擇是以純文字還是 JSON 擷取使用標準 logging 模組建立的日誌。如需詳細資訊，請參閱 the section called “搭配 Python 使用 Lambda 進階日誌控制項”。

目前，所有 Python 執行期的預設日誌格式都是純文字。下列範例顯示如何在 CloudWatch Logs 日誌中以純文字擷取使用標準logging 模組建立的日誌輸出。

```
import os
import logging
logger = logging.getLogger()
logger.setLevel("INFO")

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES')
```

```
    logger.info(os.environ['AWS_LAMBDA_LOG_GROUP_NAME'])
    logger.info(os.environ['AWS_LAMBDA_LOG_STREAM_NAME'])
    logger.info('## EVENT')
    logger.info(event)
```

來自 `logger` 的輸出包含記錄等級、時間戳記和請求 ID。

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2025-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES
[INFO]  2025-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    /aws/
lambda/my-function
[INFO]  2025-08-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    2025/01/31/
[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d
[INFO]  2025-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT
[INFO]  2025-08-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}
END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 117 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

> **Note**
>
> 當您的函數的日誌格式設定為純文字時，Python 執行期的預設日誌層級設定為 WARN。這意味著 Lambda 只會傳送 WARN 及更低層級的日誌輸出至 CloudWatch Logs。若要變更預設日誌層級，請使用 Python `logging` `setLevel()` 方法，如此範例程式碼所示。如果將函數的日誌格式設定為 JSON，我們建議您使用 Lambda 進階記錄控制項來設定函數的日誌層級，而不是在程式碼中設定日誌層級。如需進一步了解，請參閱 the section called “搭配 Python 使用日誌層級篩選”

搭配 Python 使用 Lambda 進階日誌控制項

為了讓您更妥善地控制擷取、處理和使用函數日誌的方式，您可以針對支援的 Lambda Python 執行期設定下列記錄選項：

- 日誌格式 - 在純文字和結構化 JSON 格式之間為您的日誌進行選擇

- 日誌層級 - 對於 JSON 格式的日誌，請選擇 Lambda 傳送到 Amazon CloudWatch 的日誌之詳細等級，例如 ERROR、DEBUG 或 INFO
- 日誌群組 - 選擇您的函數將日誌傳送到的 CloudWatch 日誌群組

如需這些日誌選項的詳細資訊，以及如何設定函數以使用這些選項的說明，請參閱 the section called “設定 Lambda 函數的進階日誌記錄控制項”。

若要進一步瞭解如何將日誌格式和日誌層級選項與 Python Lambda 函數搭配使用，請參閱以下各節中的指引。

搭配 Python 使用結構化的 JSON 日誌

如果您為函數的日誌格式選取 JSON，Lambda 會將 Python 標準記錄程式庫所輸出的日誌以結構化 JSON 形式傳送至 CloudWatch。每個 JSON 日誌物件都包含至少四個鍵值對，其中包含下列索引鍵：

- `"timestamp"` - 產生日誌訊息的時間
- `"level"` - 指派給訊息的日誌層級
- `"message"` - 日誌訊息的內容
- `"requestId"` - 進行調用的唯一請求 ID。

Python `logging` 程式庫還可以新增額外的鍵值對 (如 `"logger"`) 到這個 JSON 物件。

以下各章節中的範例顯示當您將函數的日誌格式設定為 JSON 時，如何在 CloudWatch Logs 日誌中擷取使用 Python `logging` 程式庫產生的記錄輸出。

請注意，如果您使用 `print` 方法產生基本的日誌輸出，如 the section called “列印至日誌” 中所描述，即使您將函數的日誌格式設定為 JSON，Lambda 仍會以純文字擷取這些輸出。

使用 Python 記錄程式庫進行標準 JSON 日誌輸出

下列範例程式碼片段和日誌輸出顯示當函數的日誌格式設為 JSON 時，如何在 CloudWatch Logs 中擷取使用 Python `logging` 程式庫產生的標準日誌輸出。

Example Python 日誌記錄程式碼

```
import logging
logger = logging.getLogger()
```

```
def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON 日誌記錄

```
{
    "timestamp":"2025-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"Inside the handler function",
    "logger": "root",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

在 JSON 中記錄額外的參數

將函數日誌格式設為 JSON 後，您也可以透過標準 Python `logging` 程式庫記錄額外參數，使用 `extra` 關鍵字將 Python 字典傳遞到日誌輸出。

Example Python 日誌記錄程式碼

```
import logging

def lambda_handler(event, context):
    logging.info(
        "extra parameters example",
        extra={"a":"b", "b": [3]},
    )
```

Example JSON 日誌記錄

```
{
  "timestamp": "2025-11-02T15:26:28Z",
  "level": "INFO",
  "message": "extra parameters example",
  "logger": "root",
  "requestId": "3dbd5759-65f6-45f8-8d7d-5bdc79a3bd01",
  "a": "b",
  "b": [
    3
  ]
```

```
}
```

在 JSON 中記錄例外狀況

下列程式碼片段顯示當您將日誌格式設定為 JSON 時，如何在函數的日誌輸出中擷取 Python 的例外狀況。請注意，使用 `logging.exception` 產生的日誌檔輸出會指派到日誌層級 ERROR。

Example Python 日誌記錄程式碼

```
import logging

def lambda_handler(event, context):
    try:
        raise Exception("exception")
    except:
        logging.exception("msg")
```

Example JSON 日誌記錄

```
{
  "timestamp": "2025-11-02T16:18:57Z",
  "level": "ERROR",
  "message": "msg",
  "logger": "root",
  "stackTrace": [
    "  File \"/var/task/lambda_function.py\", line 15, in lambda_handler\n    raise
 Exception(\"exception\")\n"
  ],
  "errorType": "Exception",
  "errorMessage": "exception",
  "requestId": "3f9d155c-0f09-46b7-bdf1-e91dab220855",
  "location": "/var/task/lambda_function.py:lambda_handler:17"
}
```

JSON 結構化日誌與其他記錄工具

如果您的程式碼已使用 Powertools 等其他日誌程式庫 AWS Lambda來產生 JSON 結構化日誌，則不需要進行任何變更。 AWS Lambda 不會對已編碼 JSON 的任何日誌進行雙重編碼。即使您將函數設定為使用 JSON 日誌格式，您的記錄輸出也會以您定義的 JSON 結構顯示於 CloudWatch 中。

下列範例顯示如何在 CloudWatch Logs 中擷取使用適用於 AWS Lambda 套件的 Powertools 產生的日誌輸出。無論函數的日誌組態設定為 JSON 還是 TEXT，此日誌輸出的格式都相同。如需使用

Powertools 的詳細資訊 AWS Lambda，請參閱 the section called “使用 Powertools for AWS Lambda (Python) 和 AWS SAM 進行結構化記錄” 和 the section called “使用 Powertools for AWS Lambda (Python) 和 AWS CDK 進行結構化記錄”

Example Python 記錄程式碼片段（使用 Powertools for AWS Lambda)

```
from aws_lambda_powertools import Logger

logger = Logger()

def lambda_handler(event, context):
    logger.info("Inside the handler function")
```

Example JSON 日誌記錄（使用 Powertools for AWS Lambda)

```
{
    "level": "INFO",
    "location": "lambda_handler:7",
    "message": "Inside the handler function",
    "timestamp": "2025-10-31 22:38:21,010+0000",
    "service": "service_undefined",
    "xray_trace_id": "1-654181dc-65c15d6b0fecbdd1531ecb30"
}
```

搭配 Python 使用日誌層級篩選

透過設定日誌層級篩選，您可以選擇只傳送特定日誌層級或更低層級的日誌至 CloudWatch Logs。若要瞭解如何設定函數的日誌層級篩選，請參閱 the section called “日誌層級篩選”。

若要 AWS Lambda 讓 根據應用程式日誌層級篩選您的應用程式日誌，您的函數必須使用 JSON 格式的日誌。您可以透過兩種方式達成此操作：

- 使用標準 Python `logging` 程式庫建立日誌輸出，並配置您的函數以使用 JSON 日誌格式。然後 AWS Lambda 使用 the section called “搭配 Python 使用結構化的 JSON 日誌” 中描述的 JSON 物件中的「層級」索引鍵值組篩選日誌輸出。若要瞭解如何設定函數的日誌格式，請參閱 the section called “設定 Lambda 函數的進階日誌記錄控制項”。
- 使用其他日誌程式庫或方法，在您的程式碼中建立 JSON 結構化日誌，其中包含定義日誌輸出層級的「層級」索引鍵值組。例如，您可以使用 Powertools for AWS Lambda 從程式碼產生 JSON 結構化日誌輸出。

您也可以使用列印陳述式輸出包含日誌層級識別碼的 JSON 物件。下列列印陳述式會產生 JSON 格式的輸出，其中日誌層級設定為 INFO。如果您函數的日誌層級設定為 INFO、DEBUG 或 TRACE，則 AWS Lambda 會將 JSON 物件傳送至 CloudWatch Logs。

```
print('{"msg":"My log message", "level":"info"}')
```

若要讓 Lambda 篩選函數的日誌，您還必須在 JSON 日誌輸出中包含 `"timestamp"` 索引鍵值組。必須以有效的 RFC 3339 時間戳記格式指定時間。如果您沒有提供有效的時間戳記，Lambda 會為日誌指派層級 INFO，並為您新增時間戳記。

在 Lambda 主控台檢視日誌

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

在 CloudWatch 主控台中檢視日誌

您可以使用 Amazon CloudWatch 主控台來檢視所有 Lambda 函數調用的日誌。

若要在 CloudWatch 主控台上檢視日誌

1. 在 CloudWatch 主控台上開啟日誌群組頁面。
2. 選擇您的函數的日誌群組 (/aws/lambda/***your-function-name***)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數時，以及建立其他執行個體來處理並行調用時，就會顯示日誌串流。若要尋找特定調用的日誌，建議您使用 檢測函數 AWS X-Ray。X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

使用 檢視日誌 AWS CLI

AWS CLI 是一種開放原始碼工具，可讓您使用命令列 Shell 中的 命令與 AWS 服務互動。若要完成本節中的步驟，您必須擁有 AWS CLI 版本 2。

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example擷取日誌 ID

下列範例顯示如何從名稱為 my-function 的函數的 LogResult 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 get-logs.sh。

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
```

```
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

使用其他記錄工具和程式庫

Powertools for AWS Lambda (Python) 是一種開發人員工具組，可實作 Serverless 最佳實務並提高開發人員速度。Logger 公用程式提供 Lambda 優化記錄器，其中包含有關所有函數之函數內容的其他資訊，輸出結構為 JSON。使用此公用程式執行下列操作：

- 從 Lambda 內容、冷啟動和 JSON 形式的結構記錄輸出中擷取關鍵欄位
- 在收到指示時記錄 Lambda 調用事件 (預設為停用)
- 透過日誌採樣僅列印調用百分比的所有日誌 (預設為停用)
- 在任何時間點將其他金鑰附加至結構化日誌
- 使用自訂日誌格式化程式 (自帶格式化程式)，以與組織的日誌記錄 RFC 相容的結構輸出日誌。

使用 Powertools for AWS Lambda (Python) 和 AWS SAM 進行結構化記錄

請依照以下步驟操作，使用 AWS SAM透過整合式 Powertools for Python 模組，下載、建置和部署範例 Hello World Python 應用程式。此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會調用、使用內嵌指標格式將日誌和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- Python 3.9
- AWS CLI 第 2 版
- AWS SAM CLI 1.75 版或更新版本。如果您有較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World Python 範本來初始化應用程式。

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.9 --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 Enter。

> **Note**
>
> 對於 HelloWorldFunction may not have authorization defined, Is this okay?，確保輸入 y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要獲取該函數的日誌，請執行 sam 日誌。如需詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》 中的 使用日誌。

```
sam logs --stack-name sam-app
```

日誌輸出如下：

```
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2025-02-03T14:59:50.371000 INIT_START Runtime Version:
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
```

```
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.114000 {
  "level": "INFO",
  "location": "hello:23",
  "message": "Hello world API - HTTP 200",
  "timestamp": "2025-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
```

```
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 756 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

管理日誌保留

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，CloudWatch 會在該時間之後自動刪除日誌。若要設定日誌保留，請將下列項目新增至您的 AWS SAM 範本：

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

使用 Powertools for AWS Lambda (Python) 和 AWS CDK 進行結構化記錄

請依照下列步驟，使用 下載、建置和部署範例 Hello World Python 應用程式，其中包含適用於 AWS Lambda (Python) 模組的整合式 Powertools AWS CDK。此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會調用、使用內嵌指標格式將日誌和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。函數會傳回 hello world 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- Python 3.9
- AWS CLI 第 2 版
- AWS CDK 第 2 版
- AWS SAM CLI 1.75 版或更新版本。如果您有較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS CDK 應用程式

1. 為您的新應用程式建立專案目錄。

   ```
   mkdir hello-world
   cd hello-world
   ```

2. 初始化應用程式。

```
cdk init app --language python
```

3. 安裝 Python 相依項。

```
pip install -r requirements.txt
```

4. 在根資料夾下建立 lambda_function 目錄。

```
mkdir lambda_function
cd lambda_function
```

5. 建立檔案 app.py，並將下列程式碼新增至檔案。這是 Lambda 函數的程式碼。

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.aws.amazon.com/powertools/python/latest//latest/core/
metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)

    # structured log
    # See: https://docs.aws.amazon.com/powertools/python/latest//latest/core/
logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
```

```
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.aws.amazon.com/powertools/python/latest//latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. 開啟 hello_world 目錄。您應看到名稱為 hello_world_stack.py 的檔案。

```
cd ..
cd hello_world
```

7. 開啟 hello_world_stack.py，並將下列程式碼新增至檔案。其中包含可建立 Lambda 函數、設定 Powertools 的環境變數並將日誌保留設定為一週的 Lambda 建構函數以及可建立 REST API 的 ApiGatewayv1 建構函數。

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
            # Using AWS Lambda Powertools via Lambda Layer
            # This imports the Powertools layer which provides observability
 features for Lambda functions
            # For available versions, see: https://docs.aws.amazon.com/powertools/
python/latest/#lambda-layer
        )
```

```
        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_9,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
            }
        )

        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
 deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
 proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. 部署您的應用程式。

```
cd ..
cdk deploy
```

9. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. 調用 API 端點：

```
curl GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

11. 若要獲取該函數的日誌，請執行 sam 日誌。如需詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》 中的 使用日誌。

```
sam logs --stack-name HelloWorldStack
```

日誌輸出如下：

```
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04
 2025-02-03T14:59:50.371000 INIT_START Runtime Version:
 python:3.9.v16    Runtime Version ARN: arn:aws:lambda:us-
east-1::runtime:07a48df201798d627f2b950f03bb227aab4a655a1d019c3296406f95937e2525
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.112000
 START RequestId: d455cfc4-7704-46df-901b-2a5cce9405be Version: $LATEST
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.114000 {
  "level": "INFO",
  "location": "hello:23",
    "message": "Hello world API - HTTP 200",
  "timestamp": "2025-02-03 14:59:51,113+0000",
  "service": "PowertoolsHelloWorld",
  "cold_start": true,
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "function_memory_size": "128",
  "function_arn": "arn:aws:lambda:us-east-1:111122223333:function:sam-app-
HelloWorldFunction-YBg8yfYtOc9j",
  "function_request_id": "d455cfc4-7704-46df-901b-2a5cce9405be",
  "correlation_id": "e73f8aef-5e07-436e-a30b-63e4b23f0047",
  "xray_trace_id": "1-63dd2166-434a12c22e1307ff2114f299"
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "function_name",
            "service"
          ]
```

```
        ],
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "function_name": "sam-app-HelloWorldFunction-YBg8yfYtOc9j",
  "service": "PowertoolsHelloWorld",
  "ColdStart": [
    1.0
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.126000 {
  "_aws": {
    "Timestamp": 1675436391126,
    "CloudWatchMetrics": [
      {
        "Namespace": "Powertools",
        "Dimensions": [
          [
            "service"
          ]
        ],
        "Metrics": [
          {
            "Name": "HelloWorldInvocations",
            "Unit": "Count"
          }
        ]
      }
    ]
  },
  "service": "PowertoolsHelloWorld",
  "HelloWorldInvocations": [
    1.0
  ]
}
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000 END
 RequestId: d455cfc4-7704-46df-901b-2a5cce9405be
```

```
2025/02/03/[$LATEST]ea9a64ec87294bf6bbc9026c05a01e04 2025-02-03T14:59:51.128000
 REPORT RequestId: d455cfc4-7704-46df-901b-2a5cce9405be    Duration: 16.33 ms
 Billed Duration: 756 ms    Memory Size: 128 MB    Max Memory Used: 64 MB    Init
 Duration: 739.46 ms
XRAY TraceId: 1-63dd2166-434a12c22e1307ff2114f299    SegmentId: 3c5d18d735a1ced0
 Sampled: true
```

12. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
cdk destroy
```

以 Python 測試 AWS Lambda 函數

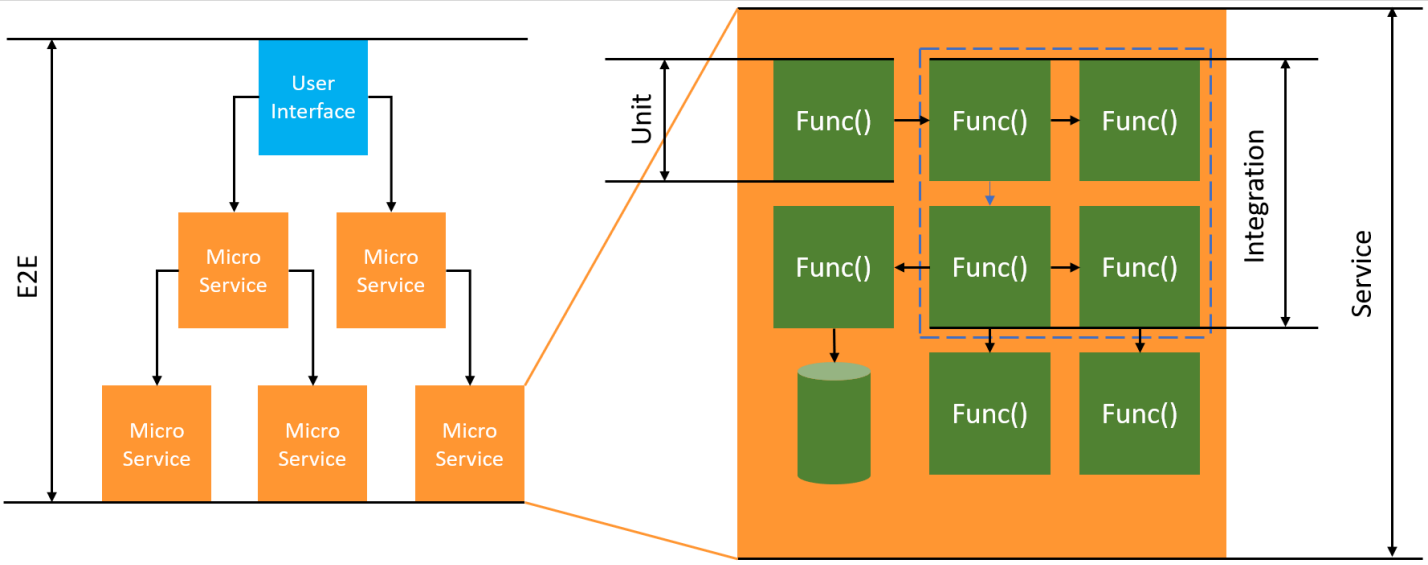
> **Note**
>
> 如需測試無伺服器解決方案之技術和最佳實務的完整介紹，請參閱測試函數章節。

測試無伺服器函數會使用傳統的測試類型和技術，但您也必須考慮測試整個無伺服器應用程式。以雲端為基礎的測試會為您的函數和無伺服器應用程式提供最準確的品質測量標準。

無伺服器應用程式架構包括透過 API 呼叫提供關鍵應用程式功能的受管服務。因此，您的開發週期應包括自動化測試，以便在函數和服務互動時驗證功能。

如果您未建立以雲端為基礎的測試，則可能會因本機環境與部署環境之間的差異而遇到問題。您的持續整合程序應先針對雲端佈建的一組資源進行測試，然後再將程式碼升級至下一個部署環境 (例如 QA、暫存或生產環境)。

繼續閱讀這份簡短指南，了解無伺服器應用程式的測試策略，或造訪無伺服器測試範例儲存庫，深入了解所選語言和執行期的特定實際範例。



若為無伺服器測試，您仍需要寫入單元、整合及端對端測試。

- 單元測試：針對一組隔離的程式碼區塊進行的測試。例如，驗證商業邏輯以計算指定的特定項目與目的地的運費。
- 整合測試：涉及到兩個以上元件或服務進行互動的測試 (通常在雲端環境)。例如，驗證函數是否有處理佇列中的事件。

- 端對端測試：驗證整個應用程式行為的測試。例如，確保基礎設施的設定正確無誤，以及事件如預期在服務之間流動，以記錄客戶的訂單。

測試無伺服器應用程式

通常會混合使用多種方法來測試無伺服器應用程式程式碼，包括在雲端進行測試、透過模擬物件進行測試，以及偶爾使用模擬器進行測試。

在雲端進行測試

在雲端進行測試對所有測試階段 (包括單元測試、整合測試和端對端測試) 來說都很有價值。您可以針對部署在雲端中的程式碼執行測試，並與雲端服務互動。這是最準確的程式碼品質測量方法。

您可以透過主控台使用測試事件，輕鬆在雲端對 Lambda 函數進行偵錯。一個測試事件是函數的 JSON 輸入。如果您的函數不需要輸入，該事件可以是空白的 JSON 文件 (`{}`)。主控台提供各種服務整合的範例事件。在主控台中建立事件後，您可以與團隊分享事件，讓測試變得更容易，結果更一致。

> **Note**
>
> 在控制台中測試函數是簡便快速的入門方式，而將測試週期自動化可確保應用程式的品質和開發速度。

測試工具

您可以透過一些工具和技術加快回饋迴圈的開發速度。例如，AWS SAM Accelerate 和 AWS CDK 監看模式都可以縮短更新雲端環境所需的時間。

Moto 是用於模擬 AWS 服務和資源的 Python 程式庫，您可以使用裝飾項目攔截和模擬回應，幾乎不用修改即可測試函數。

Powertools for AWS Lambda (Python) 的驗證功能提供裝飾項目，可用來驗證 Python 函數的輸入事件和輸出回應。

如需詳細資訊，請閱讀部落格文章：使用 Python 和 Mock AWS Services 對 Lambda 進行單元測試。

若要降低與雲端部署反覆運算相關的延遲，請參閱 AWS Serverless Application Model (AWS SAM) Accelerate、AWS Cloud Development Kit (AWS CDK) 監看模式。這些工具會監控您的基礎架構和程式碼是否變更。它們會自動建立增量更新並將其部署到您的雲端環境中，藉此回應這些變更。

如需這些工具的使用範例，請前往 Python 測試範例程式碼儲存庫。

在 中檢測 Python 程式碼 AWS Lambda

Lambda 與 整合 AWS X-Ray ，以協助您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追蹤請求，因為它會周遊您應用程式中的資源，其中可能包含 Lambda 函數和其他 AWS 服務。

若要將追蹤資料傳送至 X-Ray，您可以使用以下三個 SDK 庫之一：

- AWS Distro for OpenTelemetry (ADOT) – OpenTelemetry (OTel) SDK 的安全、生產就緒、 AWS支援的分發。
- 適用於 Python 的 AWS X-Ray SDK – 用於生成追蹤資料並將其傳送至 X-Ray 的 SDK。
- Powertools for AWS Lambda (Python) – 開發人員工具組，用於實作 Serverless 最佳實務並提高開發人員速度。

每個 SDK 均提供將遙測資料傳送至 X-Ray 服務的方法。然後，您可以使用 X-Ray 來檢視、篩選應用程式的效能指標並獲得洞察，從而識別問題和進行最佳化的機會。

> **Important**
>
> X-Ray 和 Powertools AWS Lambda SDKs是 提供的緊密整合檢測解決方案的一部分 AWS。ADOT Lambda Layers 是用於追蹤檢測之業界通用標準的一部分，這類檢測一般會收集更多資料，但可能不適用於所有使用案例。您可以使用任一解決方案在 X-Ray 中實作端對端追蹤。若要深入了解如何在兩者之間做選擇，請參閱在 AWS Distro for OpenTelemetry 和 X-Ray SDK 之間進行選擇。

章節

- 使用 Powertools for AWS Lambda (Python) 和 AWS SAM 進行追蹤
- 使用 Powertools for AWS Lambda (Python) 和 AWS CDK 進行追蹤
- 使用 ADOT 來檢測您的 Python 函數
- 使用 X-Ray SDK 來檢測 Python 功能
- 透過 Lambda 主控台來啟用追蹤
- 透過 Lambda API 啟用追蹤
- 使用 啟用追蹤 CloudFormation
- 解讀 X-Ray 追蹤
- 將執行時間相依項存放存在層中 (X-Ray SDK)

使用 Powertools for AWS Lambda (Python) 和 AWS SAM 進行追蹤

請依照下列步驟，使用 下載、建置和部署範例 Hello World Python 應用程式，其中包含適用於 AWS Lambda (Python) 模組的整合式 Powertools AWS SAM。此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會調用、使用內嵌指標格式將日誌和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。函數會傳回 hello world 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- Python 3.11
- AWS CLI 第 2 版
- AWS SAM CLI 1.75 版或更新版本。 如果您有較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World Python 範本來初始化應用程式。

```
sam init --app-template hello-world-powertools-python --name sam-app --package-type
 Zip --runtime python3.11 --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 Enter。

> **Note**
>
> 對於 HelloWorldFunction may not have authorization defined, Is this okay?，確保輸入 y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
```

```
      - fault count(5XX): 0
      - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。不能針對函數設定 X-Ray 取樣率。

使用 Powertools for AWS Lambda (Python) 和 AWS CDK 進行追蹤

請依照下列步驟，使用 下載、建置和部署範例 Hello World Python 應用程式，其中包含適用於 AWS Lambda (Python) 模組的整合式 Powertools AWS CDK。此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會調用、使用內嵌指標格式將日誌和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。函數會傳回 hello world 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- Python 3.11
- AWS CLI 第 2 版
- AWS CDK 第 2 版
- AWS SAM CLI 1.75 版或更新版本。 如果您有較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS CDK 應用程式

1. 為您的新應用程式建立專案目錄。

```
mkdir hello-world
cd hello-world
```

2. 初始化應用程式。

```
cdk init app --language python
```

3. 安裝 Python 相依項。

```
pip install -r requirements.txt
```

4. 在根資料夾下建立 lambda_function 目錄。

```
mkdir lambda_function
cd lambda_function
```

5. 建立檔案 app.py，並將下列程式碼新增至檔案。這是 Lambda 函數的程式碼。

```
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools import Logger
from aws_lambda_powertools import Tracer
from aws_lambda_powertools import Metrics
from aws_lambda_powertools.metrics import MetricUnit

app = APIGatewayRestResolver()
tracer = Tracer()
logger = Logger()
metrics = Metrics(namespace="PowertoolsSample")

@app.get("/hello")
@tracer.capture_method
def hello():
    # adding custom metrics
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/metrics/
    metrics.add_metric(name="HelloWorldInvocations", unit=MetricUnit.Count,
 value=1)
```

```
    # structured log
    # See: https://docs.powertools.aws.dev/lambda-python/latest/core/logger/
    logger.info("Hello world API - HTTP 200")
    return {"message": "hello world"}

# Enrich logging with contextual information from Lambda
@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
# Adding tracer
# See: https://docs.powertools.aws.dev/lambda-python/latest/core/tracer/
@tracer.capture_lambda_handler
# ensures metrics are flushed upon request completion/failure and capturing
 ColdStart metric
@metrics.log_metrics(capture_cold_start_metric=True)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

6. 開啟 hello_world 目錄。您應看到名稱為 hello_world_stack.py 的檔案。

```
cd ..
cd hello_world
```

7. 開啟 hello_world_stack.py，並將下列程式碼新增至檔案。其中包含可建立 Lambda 函數、設定 Powertools 的環境變數並將日誌保留設定為一週的 Lambda 建構函數以及可建立 REST API 的 ApiGatewayv1 建構函數。

```
from aws_cdk import (
    Stack,
    aws_apigateway as apigwv1,
    aws_lambda as lambda_,
    CfnOutput,
    Duration
)
from constructs import Construct

class HelloWorldStack(Stack):

    def __init__(self, scope: Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # Powertools Lambda Layer
        powertools_layer = lambda_.LayerVersion.from_layer_version_arn(
            self,
            id="lambda-powertools",
```

```
            # At the moment we wrote this example, the aws_lambda_python_alpha CDK
 constructor is in Alpha, o we use layer to make the example simpler
            # See https://docs.aws.amazon.com/cdk/api/v2/python/
aws_cdk.aws_lambda_python_alpha/README.html
            # Check all Powertools layers versions here: https://
docs.powertools.aws.dev/lambda-python/latest/#lambda-layer
            layer_version_arn=f"arn:aws:lambda:
{self.region}:017000801446:layer:AWSLambdaPowertoolsPythonV2:21"
        )

        function = lambda_.Function(self,
            'sample-app-lambda',
            runtime=lambda_.Runtime.PYTHON_3_11,
            layers=[powertools_layer],
            code = lambda_.Code.from_asset("./lambda_function/"),
            handler="app.lambda_handler",
            memory_size=128,
            timeout=Duration.seconds(3),
            architecture=lambda_.Architecture.X86_64,
            environment={
                "POWERTOOLS_SERVICE_NAME": "PowertoolsHelloWorld",
                "POWERTOOLS_METRICS_NAMESPACE": "PowertoolsSample",
                "LOG_LEVEL": "INFO"
            }
        )

        apigw = apigwv1.RestApi(self, "PowertoolsAPI",
 deploy_options=apigwv1.StageOptions(stage_name="dev"))

        hello_api = apigw.root.add_resource("hello")
        hello_api.add_method("GET", apigwv1.LambdaIntegration(function,
 proxy=True))

        CfnOutput(self, "apiUrl", value=f"{apigw.url}hello")
```

8. 部署您的應用程式。

```
cd ..
cdk deploy
```

9. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`apiUrl`].OutputValue' --output text
```

10. 調用 API 端點：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

11. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

```
New XRay Service Graph
  Start time: 2023-02-03 14:59:50+00:00
  End time: 2023-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
```

```
      - fault count(5XX): 0
      - total response time: 0

XRay Event [revision 1] at (2023-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

12. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
cdk destroy
```

使用 ADOT 來檢測您的 Python 函數

ADOT 提供全受管 Lambda 層，包含使用 OTel SDK 收集遙測資料所需的一切內容。透過取用此層，您可以檢測 Lambda 函數，而無需修改任何函數程式碼。您還可以將層設定為對 OTel 進行自訂初始化。如需詳細資訊，請參閱 ADOT 文件中的針對 Lambda 上的 ADOT 收集器進行自訂組態設定。

對於 Python 執行時間，您可以新增適用於 ADOT Python 的AWS 受管 Lambda 層來自動檢測您的函數。此層同時適用於 arm64 和 x86_64 架構。如需有關如何新增此層的詳細指示，請參閱 ADOT 文件中的適用於 Python 的AWS Distro for OpenTelemetry Lambda 支援。

使用 X-Ray SDK 來檢測 Python 功能

若要記錄 Lambda 函數對應用程式中其他資源所進行之呼叫的詳細資料，您也可以使用 適用於 Python 的 AWS X-Ray SDK。若要取得開發套件，請將 `aws-xray-sdk` 套件新增至應用程式的相依性。

Example requirements.txt

```
jsonpickle==1.3
aws-xray-sdk==2.4.3
```

在函數程式碼中，您可以使用 `aws_xray_sdk.core`模組修補程式`boto3`庫，以檢測 AWS SDK 用戶端。

Example函數 – 追蹤 AWS SDK 用戶端

```
import boto3
from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch_all

logger = logging.getLogger()
logger.setLevel(logging.INFO)
patch_all()

client = boto3.client('lambda')
client.get_account_settings()

def lambda_handler(event, context):
    logger.info('## ENVIRONMENT VARIABLES\r' + jsonpickle.encode(dict(**os.environ)))
  ...
```

新增正確的相依項並完成必要的程式碼變更後，請透過 Lambda 主控台或 API 在函數組態中啟用追蹤功能。

透過 Lambda 主控台來啟用追蹤

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態，然後選擇監控和操作工具。
4. 在其他監控工具欄位中，選擇編輯。
5. 在 CloudWatch Application Signals 和 AWS X-Ray 下，選擇啟用 Lambda 服務追蹤。
6. 選擇儲存。

透過 Lambda API 啟用追蹤

使用 AWS CLI 或 AWS SDK 在 Lambda 函數上設定追蹤，請使用下列 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration

- CreateFunction

下列範例 AWS CLI 命令會在名為 my-function 的函數上啟用主動追蹤。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

使用 啟用追蹤 CloudFormation

若要在 CloudFormation 範本中的 `AWS::Lambda::Function` 資源上啟用追蹤，請使用 `TracingConfig` 屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用 `Tracing` 屬性。

Example template.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

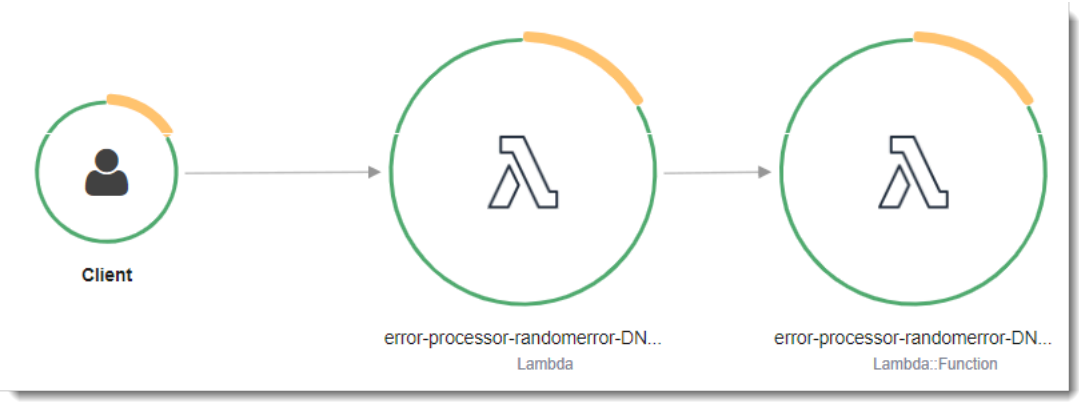
解讀 X-Ray 追蹤

您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將 AWSXRayDaemonWriteAccess 政策新增至執行角色。

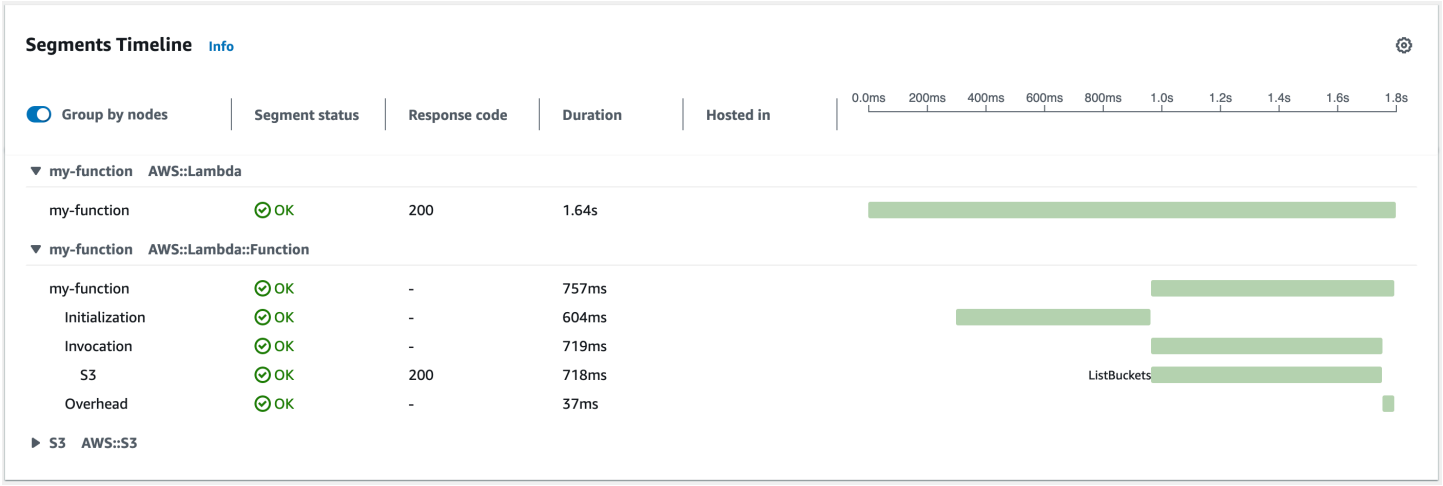
設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 服務圖顯示了有關應用程式及其所有元件的資訊。下列範例顯示了一個具有兩個函數的應用程式。主要函式會處理事件，有時會傳回錯誤。頂端的第二個函數會處理出現在第一個日誌群組中的錯誤，並使用 AWS SDK 呼叫 X-Ray、Amazon Simple Storage Service (Amazon S3) 和 Amazon CloudWatch Logs。

X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。不能針對函數設定 X-Ray 取樣率。

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會在每個追蹤上記錄 2 個區段，這會在服務圖表上建立兩個節點。下圖反白顯示了這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為 my-function，但其中之一的來源為 `AWS::Lambda`，而另一個的來源為 `AWS::Lambda::Function`。如果 `AWS::Lambda` 區段顯示錯誤，Lambda 服務就會出現問題。如果 `AWS::Lambda::Function` 區段顯示錯誤，表示您的函數出現了問題。



此範例會展開 `AWS::Lambda::Function` 區段以顯示其三個子區段：

> **Note**
>
> AWS 目前正在對 Lambda 服務實作變更。由於這些變更，您可能會看到系統日誌訊息的結構和內容，與 AWS 帳戶中不同 Lambda 函數發出的追蹤區段之間存在細微差異。
> 此處顯示的追蹤範例說明了舊式函數區段。下列段落說明了舊式和新式區段之間的差異。
> 這些變化將在未來幾週內實作，除中國和 GovCloud 區域以外，所有 AWS 區域 中的所有函數都會轉換至使用新格式的日誌訊息和追蹤區段。

舊式函數區段包含下列子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行期為做好準備以處理下一個事件所花費的時間。

新式函數區段不包含 `Invocation` 子區段。相反地，客戶子區段會直接連接至函數區段。如需舊式和新式函數區段結構的詳細資訊，請參閱the section called “了解 X-Ray 追蹤”。

您也可以檢測 HTTP 用戶端、記錄 SQL 查詢，以及建立具有註釋和中繼資料的自訂子區段。如需詳細資訊，請參閱《AWS X-Ray 開發人員指南》中的 適用於 Python 的 AWS X-Ray SDK 許可。

> **定價**
>
> 您可以在每月免費使用 X-Ray 追蹤，最高可達特定限制，作為 AWS 免費方案的一部分。達到閾值後，X-Ray 會收取追蹤儲存及擷取的費用。如需詳細資訊，請參閱 AWS X-Ray 定價。

將執行時間相依項存放存在層中 (X-Ray SDK)

如果您使用 X-Ray 開發套件來檢測函數程式碼的 AWS SDK 用戶端，您的部署套件可能會變得相當大。為了避免每次更新函數程式碼時上傳執行時間相依性，請將 X-Ray SDK 封裝在一個 Lambda 層中。

以下範例會顯示存放 適用於 Python 的 AWS X-Ray SDK的 `AWS::Serverless::LayerVersion` 資源。

Example template.yml - 相依性層

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-python-lib
      Description: Dependencies for the blank-python sample app.
      ContentUri: package/.
      CompatibleRuntimes:
        - python3.11
```

透過此組態，您只有在變更執行時間相依性時才會更新程式庫層。由於函數部署套件僅含有您的程式碼，因此有助於減少上傳時間。

為相依性建立圖層需要建置變更，才能在部署之前產生圖層封存。如需工作範例，請參閱 blank-python 範例應用程式。

使用 Ruby 建置 Lambda 函數

您可以在 AWS Lambda中執行 Ruby 程式碼。Lambda 提供用於執行程式碼來處理事件的 Ruby 執行期。您的程式碼會在包含 的環境中執行 適用於 Ruby 的 AWS SDK，其中包含您管理之 AWS Identity and Access Management (IAM) 角色的登入資料。若要進一步了解 Ruby 執行時期隨附的 SDK 版本，請參閱 the section called “包含執行時期的 SDK 版本”。

Lambda 支援以下 Ruby 執行期。

名稱	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Ruby 3.4	`ruby3.4`	Amazon Linux 2023	2028 年 3 月 31 日	2028 年 4 月 30 日	2028 年 5 月 31 日
Ruby 3.3	`ruby3.3`	Amazon Linux 2023	2027 年 3 月 31 日	2027 年 4 月 30 日	2027 年 5 月 31 日
Ruby 3.2	`ruby3.2`	Amazon Linux 2	2026 年 3 月 31 日	2026 年 8 月 31 日	2026 年 9 月 30 日

建立 Ruby 函式

1. 開啟 Lambda 主控台。
2. 選擇建立函數。
3. 進行下列設定：
   - 函數名稱：輸入函數名稱。
   - 執行時間：選擇 Ruby 3.4。
4. 選擇建立函數。

主控台將建立一個 Lambda 函數，其具有名為 `lambda_function.rb` 的單一來源檔案。您可以使用內建的程式碼編輯器編輯該檔案並加入更多檔案。在 DEPLOY 區段中，選擇部署以更新函數的程式碼。然後，若要執行程式碼，請在測試事件區段中選擇建立測試事件。

lambda_function.rb 檔案匯出名為 lambda_handler 的函數，它接受事件物件與內容物件。這就是在調用函數時，Lambda 呼叫的處理常式函數。Ruby 函數執行期會從 Lambda 中取得調用事件並其傳遞至處理常式。在函式組態中，處理常式值為 lambda_function.lambda_handler。

當您儲存函數程式碼時，Lambda 主控台會建立 .zip 封存檔部署套件。當您在主控台之外開發函數程式碼 (使用 IDE) 時，您需要建立部署套件將您的程式碼上傳到 Lambda 函數。

除了傳遞調用事件外，函式執行期還會傳遞內容物件至處理常式。內容物件包含了有關調用、函式以及執行環境的額外資訊。更多詳細資訊將另由環境變數提供。

Lambda 函數隨附有 CloudWatch Logs 記錄群組。函數執行期會將每次調用的詳細資訊傳送至 CloudWatch Logs。它在調用期間會轉送您的函數輸出的任何記錄。如果您的函數傳回錯誤，Lambda 會對該錯誤進行格式化之後傳回給調用端。

主題

- 包含執行時期的 SDK 版本
- 啟用 Yet Another Ruby JIT (YJIT)
- 以 Ruby 定義 Lambda 函數處理常式
- 使用 .zip 封存檔部署 Ruby Lambda 函數
- 使用容器映像部署 Ruby Lambda 函數
- 針對 Ruby Lambda 函數使用層
- 使用 Lambda 內容物件擷取 Ruby 函數資訊
- 記錄和監控 Ruby Lambda 函數
- 在 中檢測 Ruby 程式碼 AWS Lambda

包含執行時期的 SDK 版本

Ruby 執行時間中包含的 AWS SDK 版本取決於執行時間版本和您的 AWS 區域。適用於 Ruby 的 AWS SDK 設計為模組化，並以 分隔 AWS 服務。若要尋找您使用之執行時期中包含的特定服務 Gem 的版本編號，請使用下列格式的程式碼建立 Lambda 函數。將 aws-sdk-s3 和 Aws::S3 取代為程式碼使用的服務 Gem 名稱。

```
require 'aws-sdk-s3'

def lambda_handler(event:, context:)
  puts "Service gem version: #{Aws::S3::GEM_VERSION}"
  puts "Core version: #{Aws::CORE_GEM_VERSION}"
```

```
end
```

啟用 Yet Another Ruby JIT (YJIT)

Ruby 執行時間支援 YJIT，這是一種輕量、極簡的 Ruby JIT 編譯器。YJIT 提供了明顯更高的性能，但使用的記憶體也比 Ruby 解譯器更多。建議將 YJIT 用於 Ruby on Rails 工作負載。

依預設不會啟用 YJIT。若要為 Ruby 函數啟用 YJIT，請將RUBY_YJIT_ENABLE環境變數設定為 1。若要確認已啟用 YJIT，請列印 RubyVM::YJIT.enabled? 方法的結果。

Example– 確認已啟用 YJIT

```
puts(RubyVM::YJIT.enabled?())
# => true
```

以 Ruby 定義 Lambda 函數處理常式

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

主題

- Ruby 處理常式基本概念
- Ruby Lambda 函數的程式碼最佳實務

Ruby 處理常式基本概念

以下範例中，檔案 `function.rb` 定義了一個名為 `handler` 的處理常式方法。此處理常式函式接受兩個物件做為輸入並將傳回 JSON 文件。

Example function.rb

```
require 'json'

def handler(event:, context:)
    { event: JSON.generate(event), context: JSON.generate(context.inspect) }
end
```

在您的函數組態中，`handler` 設定會告知 Lambda 應至何處尋找處理常式。接續前述範例，此設定的正確值為 **`function.handler`**。其包括兩個以句點分隔的名稱：檔案的名稱和處理常式方法的名稱。

您也可以透過類別定義處理常式方法。以下範例在名為 `process` 的模組之下透過名為 `Handler` 的類別定義 `LambdaFunctions` 處理常式方法。

Example source.rb

```
module LambdaFunctions
  class Handler
    def self.process(event:,context:)
      "Hello!"
    end
  end
end
```

就本例而言，handler 設定是 **`source.LambdaFunctions::Handler.process`**。

處理常式接受的兩個物件分別為叫用事件和內容。事件是 Ruby 物件，其包含了由叫用端所提供的承載。如果承載是 JSON 文件，事件物件即為 Ruby 雜湊；否則將會是字串。內容物件具有方法和各項屬性，提供了有關叫用、函式以及執行環境的資訊。

每次叫用您的 Lambda 函數時都將執行函數處理常式。位於處理常式外部的靜態程式碼則是按照函式的每一執行個體各執行一次。如果您的處理常式使用了像是開發套件用戶端和資料庫連線之類的資源，您即可由處理常式方法外部建立該等資源，以供多次叫用時重複使用。

函式的每一執行個體均可處理多個叫用事件，但是一次僅處理一個事件。在任何特定時間內處理某一事件的執行個體數目稱為函式的並行數。如需 Lambda 執行環境的詳細資訊，請參閱 了解 Lambda 執行環境生命週期。

Ruby Lambda 函數的程式碼最佳實務

請遵循下列清單中的準則，在建置 Lambda 函數時使用最佳編碼實務：

- 區隔 Lambda 處理常式與您的核心邏輯。能允許您製作更多可測單位的函式。例如，在 Ruby 中，這可能看起來像是：

```
def lambda_handler(event:, context:)
    foo = event['foo']
    bar = event['bar']

    result = my_lambda_function(foo:, bar:)

end

def my_lambda_function(foo:, bar:)
    // MyLambdaFunction logic here

end
```

- 控制函數部署套件內的相依性。AWS Lambda 執行環境包含多個程式庫。對於 Ruby 執行時期，其中包含 AWS SDK。若要啟用最新的一組功能與安全更新，Lambda 會定期更新這些程式庫。這些更新可能會為您的 Lambda 函數行為帶來細微的變更。若要完全掌控您函式所使用的相依性，請利用部署套件封裝您的所有相依性。
- 最小化依存項目的複雜性。偏好更簡易的框架，其可快速在執行環境啟動時載入。
- 將部署套件最小化至執行時期所必要的套件大小。這能減少您的部署套件被下載與呼叫前解壓縮的時間。對於以 Ruby 撰寫的函數，請避免上傳整個 AWS SDK 程式庫做為部署套件的一部分。或者，選擇性倚賴取得您需要的 SDK 元件的 Gem 套件 (例如 DynamoDB 或 Amazon S3 Gem 套件)。

請利用執行環境重新使用來改看函式的效能。在函式處理常式之外初始化 SDK 用戶端和資料庫連線，並在本機快取 `/tmp` 目錄中的靜態資產。由您函式的相同執行個體處理的後續叫用可以重複使用這些資源。這可藉由減少函數執行時間來節省成本。

若要避免叫用間洩漏潛在資料，請不要使用執行環境來儲存使用者資料、事件，或其他牽涉安全性的資訊。如果您的函式依賴無法存放在處理常式內記憶體中的可變狀態，請考慮為每個使用者建立個別函式或個別函式版本。

使用 Keep-Alive 指令維持持續連線的狀態。Lambda 會隨著時間的推移清除閒置連線。叫用函數時嘗試重複使用閒置連線將導致連線錯誤。若要維護持續連線，請使用與執行階段相關聯的 keep-alive (保持啟用) 指令。如需範例，請參閱在 Node.js 中重複使用 Keep-Alive 的連線。

使用環境變數將操作參數傳遞給您的函數。例如，如果您正在寫入到 Amazon S3 儲存貯體，而非對您正在寫入的儲存貯體名稱進行硬式編碼，請將儲存貯體名稱設定為環境變數。

避免在 Lambda 函數中使用遞迴調用，其中函數會調用自己或啟動可能再次調用函數的程序。這會導致意外的函式呼叫量與升高的成本。若您看到意外的調用數量，當更新程式碼時，請立刻將函數的預留並行設為 `0`，以調節對函數的所有調用。

請勿在您的 Lambda 函數程式碼中使用未記錄的非公有 API。對於 AWS Lambda 受管執行時間，Lambda 會定期將安全性和函數更新套用至 Lambda 的內部 API。這些內部 API 更新可能是向後不相容的，這會導致意外結果，例如若您的函數依賴於這些非公有 API，則叫用失敗。請參閱 API 參考查看公開可用 API 的清單。

撰寫等冪程式碼。為函數撰寫等冪程式碼可確保採用相同方式來處理重複事件。程式碼應正確驗證事件並正常處理重複的事件。如需詳細資訊，請參閱 How do I make my Lambda function idempotent? (如何讓 Lambda 函數等冪？)。

使用 .zip 封存檔部署 Ruby Lambda 函數

AWS Lambda 函數的程式碼包含一個 .rb 檔案 (內含函數的處理常式程式碼)，以及您的程式碼所依賴的任何其他相依項 (Gem)。若要將此函數程式碼部署到 Lambda，您可以使用部署套件。此套件可以是 .zip 封存檔或容器映像。如需搭配 Ruby 使用容器映像的詳細資訊，請參閱使用容器映像部署 Ruby Lambda 函數。

若要建立 .zip 封存檔的部署套件，您可以使用命令列工具的內建 .zip 封存檔公用程式，或任何其他 .zip 檔案公用程式 (例如 7zip)。以下各節顯示的範例假設您在 Linux 或 MacOS 環境中使用命令列 `zip` 工具。若要在 Windows 中使用相同命令，您可以安裝適用於 Linux 的 Windows 子系統，以取得 Ubuntu 和 Bash 的 Windows 整合版本。

請注意，Lambda 使用 POSIX 檔案許可，因此在建立 .zip 封存檔之前，您可能需要設定部署套件資料夾的許可。

以下各節中的範例命令使用 Bundler 公用程式，將相依項新增至部署套件。若要安裝 bundler，請執行下列命令。

```
gem install bundler
```

章節

- Ruby 中的相依項
- 建立不含相依項的 .zip 部署套件
- 建立含相依項的 .zip 部署套件
- 為相依項建立 Ruby 層
- 建立含原生程式庫的 .zip 部署套件
- 使用 .zip 檔案建立及更新 Ruby Lambda 函數

Ruby 中的相依項

對於使用 Ruby 執行期的 Lambda 函數，相依項可以是任何 Ruby gem。使用 .zip 封存部署函數時，您可以使用函數程式碼將這些相依項新增至 .zip 檔案，或使用 Lambda 層。 圖層是單獨的 .zip 檔案，可以包含其他程式碼和內容。若要進一步了解如何使用 Lambda 層，請參閱 Lambda 層。

Ruby 執行時期包括 適用於 Ruby 的 AWS SDK。如果函數使用 SDK，則不需要將其與程式碼綁定在一起。但是，若要維持相依項的完全控制或使用特定版本的 SDK，可以將其新增到函數的部署套件。

您可以將 SDK 包含在 .zip 檔案中，也可以使用 Lambda 層進行新增。.zip 檔案或 Lambda 層中的相依項優先於執行期中包含的版本。若要查看適用於 Ruby 的哪個 SDK 版本包含在執行時期版本中，請參閱the section called “包含執行時期的 SDK 版本”。

在 AWS 共同責任模式下，您負責管理函數部署套件中的任何相依項。這包括套用更新和安全性修補程式。若要更新函數部署套件中的相依項，請先建立新的 .zip 檔案，然後將其上傳至 Lambda。如需詳細資訊，請參閱 建立含相依項的 .zip 部署套件 和 使用 .zip 檔案建立及更新 Ruby Lambda 函數。

建立不含相依項的 .zip 部署套件

如果您的函數程式碼沒有相依項，則 .zip 檔案只會包含具有函數處理常式程式碼的 .rb 檔案。使用您慣用的 zip 公用程式建立 .zip 檔案，並將 .rb 檔案放在根目錄下。如果 .rb 檔案不在 .zip 檔案的根目錄下，則 Lambda 將無法執行您的程式碼。

若要了解如何部署 .zip 檔案以建立新的 Lambda 函數或更新現有函數，請參閱使用 .zip 檔案建立及更新 Ruby Lambda 函數。

建立含相依項的 .zip 部署套件

如果您的函數程式碼相依於其他 Ruby gem，則您可以使用函數程式碼將這些相依項新增至 .zip 檔案，或使用 Lambda 層。本節中的指示說明如何在 .zip 部署套件中包含相依項。如需如何在層中包含相依項的指示，請參閱the section called “為相依項建立 Ruby 層”。

假設函數程式碼儲存在專案目錄中名為 `lambda_function.rb` 的檔案中。下列範例 CLI 命令會建立名為 `my_deployment_package.zip` 的 .zip 檔案，其中包含函數程式碼及其相依項。

建立部署套件

1. 在專案目錄中，建立 `Gemfile` 以在其中指定相依項。

   ```
   bundle init
   ```

2. 使用您偏好的文字編輯器，編輯 `Gemfile` 以指定函數的相依項。例如，若要使用 TZInfo gem，請編輯 `Gemfile`，如下所示。

   ```
   source "https://rubygems.org"
   gem "tzinfo"
   ```

3. 執行以下命令來安裝專案目錄中 `Gemfile` 指定的 gem。此命令將 `vendor/bundle` 設定為 gem 安裝的預設路徑。

```
bundle config set --local path 'vendor/bundle' && bundle install
```

您應該會看到類似下列的輸出。

```
Fetching gem metadata from https://rubygems.org/..........
Resolving dependencies...
Using bundler 2.4.13
Fetching tzinfo 2.0.6
Installing tzinfo 2.0.6
...
```

> **Note**
>
> 若稍後再次在全域安裝 gem，請執行下列命令。
>
> ```
> bundle config set --local system 'true'
> ```

4. 建立 .zip 封存檔，其中包含具有函數處理常式程式碼的 `lambda_function.rb` 檔案以及在上一個步驟中安裝的相依項。

```
zip -r my_deployment_package.zip lambda_function.rb vendor
```

您應該會看到類似下列的輸出。

```
adding: lambda_function.rb (deflated 37%)
  adding: vendor/ (stored 0%)
  adding: vendor/bundle/ (stored 0%)
  adding: vendor/bundle/ruby/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/build_info/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/cache/ (stored 0%)
  adding: vendor/bundle/ruby/3.2.0/cache/aws-eventstream-1.0.1.gem (deflated 36%)
...
```

為相依項建立 Ruby 層

若要了解如何將 Ruby 相依項封裝至 Lambda 層中，請參閱the section called “層”。

建立含原生程式庫的 .zip 部署套件

諸如 nokogiri、nio4r 和 mysql 等許多常見 Ruby gem 包含用 C 語言編寫的原生延伸模組。將包含 C 程式碼的程式庫新增至部署套件時，必須正確建置套件，以確保其與 Lambda 執行環境相容。

對於生產應用程式，建議您使用 AWS Serverless Application Model (AWS SAM) 來建置和部署程式碼。在 AWS SAM 中，使用 `sam build --use-container` 選項在類似 Lambda 的 Docker 容器中建置函數。若要進一步了解如何使用 AWS SAM 來部署函數程式碼，請參閱《AWS SAM 開發人員指南》中的建置應用程式。

若要在不使用 AWS SAM 的情況下建立包含 gem (帶有原生延伸模組) 的 .zip 部署套件，您也可以使用容器將相依項綁定在與 Lambda Ruby 執行期環境相同的環境中。若要完成這些步驟，必須在建置電腦上安裝 Docker。若要進一步了解如何安裝 Docker，請參閱 Install Docker Engine。

在 Docker 容器中建立 .zip 部署套件

1. 在本機建置電腦上建立一個資料夾，將容器儲存在其中。在該資料夾中，建立一個名為 `dockerfile` 的檔案並將以下程式碼貼到其中。

   ```
   FROM public.ecr.aws/sam/build-ruby3.2:latest-x86_64
   RUN gem update bundler
   CMD "/bin/bash"
   ```

2. 在建立 `dockerfile` 的資料夾中，執行以下命令以建立 Docker 容器。

   ```
   docker build -t awsruby32 .
   ```

3. 導覽到包含 `.rb` 檔案 (具有函數處理常式程式碼) 和 `Gemfile` (用於指定函數相依項) 的專案目錄。從該目錄內，執行下列命令以啟動 Lambda Ruby 容器。

   Linux/MacOS

   ```
   docker run --rm -it -v $PWD:/var/task -w /var/task awsruby32
   ```

   > **Note**
   >
   > 在 MacOS 中，您可能會看到警告，通知您所請求映像的平台與偵測到的主機平台不符。請忽略此警告。

Windows PowerShell

```
docker run --rm -it -v ${pwd}:var/task -w /var/task awsruby32
```

當容器啟動時，應能看到一個 bash 提示。

```
bash-4.2#
```

4. 設定套件公用程式，以安裝本機 `vendor/bundle` 目錄中 `Gemfile` 指定的 gem，並安裝相依項。

```
bash-4.2# bundle config set --local path 'vendor/bundle' && bundle install
```

5. 使用函數程式碼和其相依項建立 .zip 部署套件。在此範例中，包含函數處理常式程式碼的檔案命名為 `lambda_function.rb`。

```
bash-4.2# zip -r my_deployment_package.zip lambda_function.rb vendor
```

6. 結束容器並回到本機專案目錄。

```
bash-4.2# exit
```

現在可以使用 .zip 檔案部署套件來建立或更新 Lambda 函數。請參閱 使用 .zip 檔案建立及更新 Ruby Lambda 函數

使用 .zip 檔案建立及更新 Ruby Lambda 函數

建立 .zip 部署套件後，您可以使用該套件建立新的 Lambda 函數或更新現有函數。您可以使用 Lambda 主控台、AWS Command Line Interface 及 Lambda API 部署 .zip 套件。您也可以使用 AWS Serverless Application Model (AWS SAM) 和 CloudFormation 建立並更新 Lambda 函數。

Lambda 的 .zip 部署套件大小上限為 250 MB (解壓縮)。請注意，此限制適用於您上傳的所有檔案 (包括任何 Lambda 層) 的大小總和。

Lambda 執行時間需有許可才能讀取部署套裝服務中的檔案。在 Linux 許可八進位標記法中，Lambda 需要 644 個許可 (rw-r--r--) 用於非可執行檔，以及 755 個許可 (rwxr-x) 用於目錄和可執行檔。

在 Linux 和 MacOS 中，使用 chmod 命令變更部署套件中檔案和目錄的檔案許可。例如，若要為非可執行檔提供正確的許可，請執行下列命令。

```
chmod 644 <filepath>
```

若要在 Windows 中變更檔案許可，請參閱 Microsoft Windows 文件的 Set, View, Change, or Remove Permissions on an Object。

> **Note**
>
> 如果您未授予 Lambda 存取部署套件中的目錄所需的許可，Lambda 會將這些目錄的許可設定為 755 (rwxr-xr-x)。

透過主控台使用 .zip 檔案建立及更新函數

若要建立新函數，您必須先在主控台中建立函數，然後上傳您的 .zip 封存檔。若要更新現有函數，請開啟函數的頁面，然後按照同樣的程序新增更新後的 .zip 檔案。

如果您的 .zip 檔案小於 50 MB，您可以透過直接從本機電腦上傳檔案來建立或更新函數。若 .zip 檔案大於 50 MB，您必須先將套件上傳至 Amazon S3 儲存貯體。如需透過 AWS 管理主控台 將檔案上傳至 Amazon S3 儲存貯體的指示，請參閱《Amazon S3 入門》。若要使用 AWS CLI 上傳檔案，請參閱AWS CLI《使用者指南》 中的移動物件。

> **Note**
>
> 不能變更現有函數的部署套件類型 (.zip 或容器映像)。例如，您不能轉換容器映像函數以使用 .zip 封存檔。您必須建立新的函數。

若要建立新的函數 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇建立函數。
2. 選擇 Author from scratch (從頭開始撰寫)。
3. 在基本資訊下，請執行下列動作：
   a. 在函數名稱中輸入函數名稱。
   b. 在執行期中選取要使用的執行期。

c. (選用) 在架構中選擇要用於函數的指令集架構。預設架構值為 x86_64。請確定函數的 .zip 部署套件與您選取的指令集架構相容。

4. (選用) 在許可下，展開變更預設執行角色。您可建立新的執行角色，或使用現有的角色。
5. 選擇 Create function (建立函數)。Lambda 會使用您選擇的執行期建立一個基本的「Hello world」函數。

若要從本機電腦上傳 .zip 封存檔 (主控台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 .zip 檔案。
5. 若要上傳 .zip 檔案，請執行下列操作：

   a. 選擇上傳，然後在檔案選擇器中選取您的 .zip 檔案。
   b. 選擇 Open (開啟)。
   c. 選擇儲存。

若要從 Amazon S3 儲存貯體上傳 .zip 封存檔 (控制台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳新 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 Amazon S3 位置。
5. 貼上 .zip 檔案的 Amazon S3 連結 URL，然後選擇儲存。

使用主控台程式碼編輯器更新 .zip 檔案函數

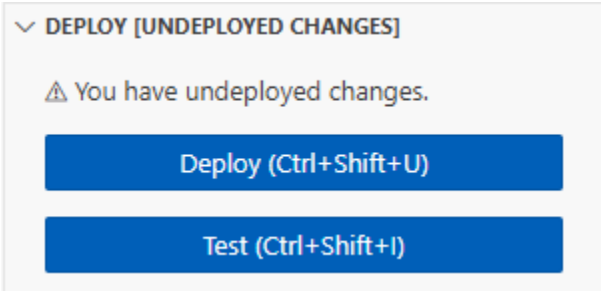
對於某些具有 .zip 部署套件的函數，您可以使用 Lambda 主控台的內建程式碼編輯器直接更新函數程式碼。若要使用此功能，您的函數必須符合下列條件：

- 您的函數必須使用其中一種轉譯語言執行期 (Python、Node.js 或 Ruby)
- 函數的部署套件必須小於 50 MB (未壓縮)。

具有容器映像部署套件之函數的函數程式碼無法直接在主控台中編輯。

若要使用主控台程式碼編輯器更新函數程式碼

1. 開啟 Lambda 主控台的函數頁面，然後選取您的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中，選取您的原始程式碼檔案，然後在整合式程式碼編輯器中加以編輯。
4. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



透過 AWS CLI 使用 .zip 檔案建立及更新函數

您可以使用 AWS CLI 建立新函數，或使用 .zip 檔案更新現有函數。使用 create-function 和 update-function-code 命令來部署您的 .zip 套件。如果您的 .zip 檔案小於 50 MB，則可以從本機建置電腦的檔案位置上傳 .zip 套件。若檔案較大，則必須先從 Amazon S3 儲存貯體上傳 .zip 套件。如需透過 AWS CLI 將檔案上傳至 Amazon S3 儲存貯體的指示，請參閱《AWS CLI 使用者指南》中的 移動物件。

> **Note**
>
> 如果您透過 AWS CLI 從 Amazon S3 儲存貯體上傳 .zip 檔案，則儲存貯體必須與函數位於相同的 AWS 區域。

若要使用 .zip 檔案與 AWS CLI 建立新函數，您必須指定下列項目：

- 函數名稱 (`--function-name`)
- 函數的執行期 (`--runtime`)
- 函數執行角色的 Amazon Resource Name (ARN) (`--role`)
- 函數程式碼中處理常式方法的名稱 (`--handler`)

您也必須指定 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--code` 選項。您只需針對版本控制的物件使用 `S3ObjectVersion` 參數。

```
aws lambda create-function --function-name myFunction \
--runtime ruby3.2 --handler lambda_function.lambda_handler \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

若要使用 CLI 更新現有函數，您可以使用 `--function-name` 參數指定函數的名稱。您也必須指定要用來更新函數程式碼的 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--s3-bucket` 和 `--s3-key` 選項。您只需針對版本控制的物件使用 `--s3-object-version` 參數。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

透過 Lambda API 使用 .zip 檔案建立及更新函數

若要使用 .zip 封存檔建立及更新函數，請使用下列 API 操作：

- CreateFunction
- UpdateFunctionCode

透過 AWS SAM 使用 .zip 檔案建立及更新函數

AWS Serverless Application Model(AWS SAM) 是一項工具組，可協助簡化在 AWS 上建置和執行無伺服器應用程式的程序。您可以在 YAML 或 JSON 範本中定義應用程式的資源，並使用 AWS SAM 命

令列界面 (AWS SAM CLI) 來建置、封裝及部署應用程式。使用 AWS SAM 範本建置 Lambda 函數時，AWS SAM 會使用函數程式碼和您指定的任何相依項目自動建立 .zip 部署套件或容器映像。若要進一步了解如何使用 AWS SAM 建置和部署 Lambda 函數，請參閱《AWS Serverless Application Model 開發人員指南》中的 AWS SAM入門。

您也可以使用 AWS SAM 來建立使用現有 .zip 封存檔的 Lambda 函數。若要使用 AWS SAM 建立 Lambda 函數，您可以將 .zip 檔案儲存在 Amazon S3 儲存貯體或建置電腦的本機資料夾中。如需透過 AWS CLI 將檔案上傳至 Amazon S3 儲存貯體的指示，請參閱《AWS CLI 使用者指南》中的 移動物件。

在您的 AWS SAM 範本中，`AWS::Serverless::Function` 資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `CodeUri`：設定為函數程式碼的 Amazon S3 URI、本機資料夾的路徑或 FunctionCode 物件
- `Runtime`：設定為所選執行期

在使用 AWS SAM 的情況下，如果您的 .zip 檔案大於 50 MB，則不需要先將檔案上傳至 Amazon S3 儲存貯體。AWS SAM 可以從本機建置電腦上的位置上傳大小上限為 250 MB (解壓縮) 的 .zip 套件。

若要進一步了解如何在 AWS SAM 中使用 .zip 檔案部署函數，請參閱《AWS SAM 開發人員指南》中的 AWS::Serverless::Function。

透過 CloudFormation 使用 .zip 檔案建立及更新函數

您可以使用 CloudFormation 來建立使用 .zip 封存檔的 Lambda 函數。若要使用 .zip 檔案建立 Lambda 函數，您必須先將檔案上傳至 Amazon S3 儲存貯體。如需透過 AWS CLI 將檔案上傳至 Amazon S3 儲存貯體的指示，請參閱《AWS CLI 使用者指南》中的 移動物件。

在您的 CloudFormation 範本中，`AWS::Lambda::Function` 資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `Code`：在 `S3Bucket` 和 `S3Key` 欄位中輸入 Amazon S3 儲存貯體名稱和 .zip 檔案名稱。
- `Runtime`：設定為所選執行期

CloudFormation 產生的 .zip 檔案不能超過 4 MB。若要進一步了解如何在 CloudFormation 中使用 .zip 檔案部署函數，請參閱《CloudFormation 使用者指南》中的 AWS::Lambda::Function。

使用容器映像部署 Ruby Lambda 函數

有三種方法可以為 Ruby Lambda 函數構建容器映像：

- 使用 Ruby AWS 的基礎映像

  AWS 基礎映像會預先載入語言執行期、用來管理 Lambda 與函數程式碼之間互動的執行期界面用戶端，以及用於本機測試的執行期界面模擬器。

- 使用僅限 AWS 作業系統的基礎映像

  AWS 僅限作業系統的基礎映像包含 Amazon Linux 發行版本和執行時間界面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Ruby 的執行期介面用戶端。

- 使用非AWS 基礎映像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Ruby 的執行期介面用戶端。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

本頁面會說明如何為 Lambda 建置、測試和部署容器映像。

主題

- AWS Ruby 的基礎映像
- 使用 Ruby AWS 的基礎映像
- 透過執行期介面用戶端使用替代基礎映像

AWS Ruby 的基礎映像

AWS 為 Ruby 提供下列基礎映像：

Tags (標籤)	執行期	作業系統	Dockerfile	棄用
3.4	Ruby 3.4	Amazon Linux 2023	GitHub 上適用於 Ruby 3.4 的 Dockerfile	2028 年 3 月 31 日
3.3	Ruby 3.3	Amazon Linux 2023	Dockerfile for Ruby 3.3 on GitHub	2027 年 3 月 31 日
3.2	Ruby 3.2	Amazon Linux 2	Dockerfile for Ruby 3.2 on GitHub	2026 年 3 月 31 日

Amazon ECR 儲存庫：gallery.ecr.aws/lambda/ruby

使用 Ruby AWS 的基礎映像

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- Docker (最低版本 25.0.0)
- Docker buildx 外掛程式。
- Ruby

從基礎映像建立映像

建立適用於 Ruby 的容器映像

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir example
cd example
```

2. 建立稱為 `Gemfile` 的新檔案。應用程式所需的 RubyGems 套件須列在這裡。可從 RubyGems 適用於 Ruby 的 AWS SDK 取得。您應該選擇要安裝的特定 AWS 服務 Gem 套件。例如，若要使用 Ruby Gem for Lambda，您的 Gemfile 看起來應該會像：

```
source 'https://rubygems.org'

gem 'aws-sdk-lambda'
```

或者，aws-sdk Gem 套件包含每個可用的 AWS 服務 Gem。這個 Gem 非常大，建議您只在依賴許多 AWS 服務時才使用它。

3. 使用套件安裝作業安裝 Gemfile 中指定的相依項。

```
bundle install
```

4. 建立稱為 `lambda_function.rb` 的新檔案。您可以將下列範例函數程式碼新增至檔案進行測試，或使用您自己的函數程式碼。

   Example Ruby 函數

```
module LambdaFunction
  class Handler
    def self.process(event:,context:)
      "Hello from Lambda!"
    end
  end
end
```

5. 建立新的 Dockerfile。下列範例 Dockerfile 使用 AWS 基礎映像。此 Dockerfile 使用下列組態：

   - 將 `FROM` 屬性設定為基礎映像的 URI。
   - 使用 `COPY` 命令將函數程式碼和執行時期相依項複製到 `{LAMBDA_TASK_ROOT}`，一個 Lambda 定義的環境變數。
   - 將 `CMD` 引數設定為 Lambda 函數處理常式。

   請注意，範例 Dockerfile 不包含 USER 指令。當您將容器映像部署到 Lambda 時，Lambda 會自動定義一個具有最低權限許可的預設 Linux 使用者。這與標準 Docker 行為不同，後者會在未提供 `USER` 指令時預設為 `root` 使用者。

   Example Dockerfile

```
FROM public.ecr.aws/lambda/ruby:3.4
```

```
# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。若要使映像與 Lambda 相容，必須使用 --provenance=false 選項。

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

(選用) 在本機測試映像

1. 使用 docker run 命令啟動 Docker 影像。在此範例中，docker-image 為映像名稱，test 為標籤。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將`--platform linux/arm64`選項改用選項。`--platform linux/amd64`

2. 從新的終端機視窗，將事件張貼至本機端點。

   Linux/macOS

   在 Linux 或 macOS 中，執行下列 `curl` 命令：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   在 PowerShell 中，執行下列 `Invoke-WebRequest` 命令：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

3. 取得容器 ID。

   ```
   docker ps
   ```

4. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將 `--region`值設定為您要 AWS 區域 在其中建立 Amazon ECR 儲存庫的 。
   - `111122223333` 以您的 AWS 帳戶 ID 取代 。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於相同的 AWS 區域 中。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
```

```
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - `docker-image:test` 為 Docker 映像檔的名稱和標籤。這是您在 `docker build` 命令中指定的映像名稱和標籤。
   - 將 `<ECRrepositoryUri>` 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像來建立函數。如需詳細資訊，請參閱Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 會將映像標籤解析為特定映像摘要。這表示如果您將用來部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。

若要將新映像部署至相同的 Lambda 函數，必須使用 update-function-code 命令，即使 Amazon ECR 中的映像標籤保持不變亦如此。在以下範例中，`--publish` 選項會使用更新的容器映像來建立新的函數版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

透過執行期介面用戶端使用替代基礎映像

如果您使用僅限作業系統的基礎映像或替代的基礎映像，則必須在映像中加入執行期介面用戶端。執行期介面用戶端會讓您擴充 Runtime API，管理 Lambda 與函數程式碼之間的互動。

使用 RubyGems.org 套件管理員安裝適用於 Ruby 的 Lambda 執行期介面用戶端：

```
gem install aws_lambda_ric
```

您還可以從 GitHub 下載 Ruby 執行時間介面用戶端。

下列範例示範如何使用非AWS 基礎映像為 Ruby 建置容器映像。範例 Dockerfile 使用官方 Ruby 基礎映像。Dockerfile 包含執行期界面用戶端。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- Docker (最低版本 25.0.0)
- Docker buildx 外掛程式。
- Ruby

使用替代基礎映像建立映像

使用替代基礎映像為 Ruby 建立容器映像

1. 建立專案的目錄，然後切換至該目錄。

   ```
   mkdir example
   cd example
   ```

2. 建立稱為 `Gemfile` 的新檔案。應用程式所需的 RubyGems 套件須列在這裡。可從 RubyGems 適用於 Ruby 的 AWS SDK 取得。您應該選擇要安裝的特定 AWS 服務 Gem 套件。例如，若要使用 Ruby Gem for Lambda，您的 Gemfile 看起來應該會像：

   ```
   source 'https://rubygems.org'

   gem 'aws-sdk-lambda'
   ```

   或者，aws-sdk Gem 套件包含每個可用的 AWS 服務 Gem。這個 Gem 非常大，建議您只在依賴許多 AWS 服務時才使用它。

3. 使用套件安裝作業安裝 Gemfile 中指定的相依項。

   ```
   bundle install
   ```

4. 建立稱為 lambda_function.rb 的新檔案。您可以將下列範例函數程式碼新增至檔案進行測試，或使用您自己的函數程式碼。

   Example Ruby 函數

   ```
   module LambdaFunction
     class Handler
       def self.process(event:,context:)
         "Hello from Lambda!"
       end
     end
   end
   ```

5. 建立新的 Dockerfile。下列 Dockerfile 使用 Ruby 基礎映像，而非 AWS 基礎映像。Dockerfile 包含 適用於 Ruby 的執行期介面用戶端，可讓映像與 Lambda 相容。或者，您可以將執行期界面用戶端新增到應用程式的 Gemfile 中。

   - 將 FROM 屬性設定為 Ruby 基礎映像。
   - 為函數程式碼建立一個目錄，以及指向該目錄的環境變數。在此範例中，目錄為 /var/task，它會鏡像 Lambda 執行環境。不過，您可以選擇函數程式碼的任何目錄，因為 Dockerfile 不使用 AWS 基礎映像。
   - 將 ENTRYPOINT 設為您希望 Docker 容器在啟動時執行的模組。在此案例中，模組是執行期界面用戶端。
   - 將 CMD 引數設定為 Lambda 函數處理常式。

   請注意，範例 Dockerfile 不包含 USER 指令。當您將容器映像部署到 Lambda 時，Lambda 會自動定義一個具有最低權限許可的預設 Linux 使用者。這與標準 Docker 行為不同，後者會在未提供 USER 指令時預設為 root 使用者。

   Example Dockerfile

   ```
   FROM ruby:2.7

   # Install the runtime interface client for Ruby
   RUN gem install aws_lambda_ric

   # Add the runtime interface client to the PATH
   ENV PATH="/usr/local/bundle/bin:${PATH}"
   ```

```
# Create a directory for the Lambda function
ENV LAMBDA_TASK_ROOT=/var/task
RUN mkdir -p ${LAMBDA_TASK_ROOT}
WORKDIR ${LAMBDA_TASK_ROOT}

# Copy Gemfile and Gemfile.lock
COPY Gemfile Gemfile.lock ${LAMBDA_TASK_ROOT}/

# Install Bundler and the specified gems
RUN gem install bundler:2.4.20 && \
    bundle config set --local path 'vendor/bundle' && \
    bundle install

# Copy function code
COPY lambda_function.rb ${LAMBDA_TASK_ROOT}/

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "aws_lambda_ric" ]

# Set the CMD to your handler (could also be done as a parameter override outside
 of the Dockerfile)
CMD [ "lambda_function.LambdaFunction::Handler.process" ]
```

6. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。若要使映像與 Lambda 相容，必須使用 --provenance=false 選項。

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

(選用) 在本機測試映像

使用 執行期界面模擬器 以在本機測試映像。您可以將模擬器建置到映像中，也可以使用以下步驟，將其安裝在本機電腦。

若要在本機電腦上安裝並執行執行期介面模擬器

1. 在您的專案目錄中執行以下命令，從 GitHub 下載執行期介面模擬器 (x86-64 架構)，並安裝在本機電腦上。

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   若要安裝 arm64 模擬器，請將上一個命令中的 GitHub 儲存庫 URL 替換為以下內容：

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
   ```

   若要安裝 arm64 模擬器，請將 `$downloadLink` 更換為下列項目：

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

2. 使用 docker run 命令啟動 Docker 影像。注意下列事項：

   - `docker-image` 是映像名稱，而 `test` 是標籤。
   - `aws_lambda_ric lambda_function.LambdaFunction::Handler.process` 是 Dockerfile 中的 `ENTRYPOINT`，後面接著 `CMD`。

Linux/macOS

```
docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
 9000:8080 \
    --entrypoint /aws-lambda/aws-lambda-rie \
    docker-image:test \
        aws_lambda_ric lambda_function.LambdaFunction::Handler.process
```

PowerShell

```
docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
 9000:8080 `
--entrypoint /aws-lambda/aws-lambda-rie `
docker-image:test `
    aws_lambda_ric lambda_function.LambdaFunction::Handler.process
```

此命令將映像作為容器執行，並在 `localhost:9000/2015-03-31/functions/function/invocations` 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將`--platform linux/arm64`選項改用選項。`--platform linux/amd64`

3. 將事件張貼至本機端點。

Linux/macOS

在 Linux 或 macOS 中，執行下列 `curl` 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在 PowerShell 中，執行下列 `Invoke-WebRequest` 命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 取得容器 ID。

```
docker ps
```

5. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將 `--region`值設定為您要 AWS 區域 在其中建立 Amazon ECR 儲存庫的 。
   - 111122223333 以您的 AWS 帳戶 ID 取代 。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於相同的 AWS 區域 中。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - `docker-image:test` 為 Docker 映像檔的名稱和標籤。這是您在 `docker build` 命令中指定的映像名稱和標籤。
   - 將 `<ECRrepositoryUri>` 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像來建立函數。如需詳細資訊，請參閱Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 會將映像標籤解析為特定映像摘要。這表示如果您將用來部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。

若要將新映像部署至相同的 Lambda 函數，必須使用 update-function-code 命令，即使 Amazon ECR 中的映像標籤保持不變亦如此。在以下範例中，`--publish` 選項會使用更新的容器映像來建立新的函數版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

針對 Ruby Lambda 函數使用層

使用 Lambda 層封裝要在多個函式之間重複使用的程式碼與相依項。層通常具備程式庫相依性、自訂執行期或組態檔案。建立層包含三個一般步驟：

1. 封裝層內容。這表示建立 .zip 封存檔，其中包含您要在函數中使用的相依項。
2. 在 Lambda 中建立層。
3. 將層新增至函數中。

主題

- 封裝層內容
- 在 Lambda 中建立層
- 在函式中使用來自層的 gem
- 將層新增至函式
- 應用程式範例

封裝層內容

若要建立層，請將套件封裝成符合以下要求的 .zip 壓縮檔：

- 使用計劃用於 Lambda 函式的相同 Ruby 版本來建立層。例如，若針對 Ruby 3.4 建立層，需為函式使用 Ruby 3.4 執行時期。
- 層的 .zip 檔案必須使用下列任一目錄結構：
  - `ruby/gems/x.x.x` (其中 *x.x.x* 表示 Ruby 版本，例如 `3.4.0`)
  - `ruby/lib`

  如需更多詳細資訊，請參閱 每個 Lambda 執行時間的層路徑。
- 層中的套件必須與 Linux 相容。Lambda 函式會在 Amazon Linux 上執行。

您可以建立包含第三方 Ruby gem 或您自訂 Ruby 模組與類別的層。許多熱門的 Ruby gem 都包含必須針對 Lambda Linux 環境編譯的原生延伸模組 (C 程式碼)。

純 Ruby gem

純 Ruby gem 僅包含 Ruby 程式碼，不需要編譯。這些 gem 更易於封裝，可在不同平台上運作。

使用純 Ruby gem 建立層

1. 建立 `Gemfile`，指定要包含在層中的純 Ruby gem：

   Example Gemfile

   ```
   source 'https://rubygems.org'

   gem 'tzinfo'
   ```

2. 使用封裝程式將 gem 安裝至 `vendor/bundle` 目錄：

   ```
   bundle config set --local path vendor/bundle
   bundle install
   ```

3. 將已安裝的 gem 複製至 Lambda 所需的目錄結構 (`ruby/gems/3.4.0`)：

   ```
   mkdir -p ruby/gems/3.4.0
   cp -r vendor/bundle/ruby/3.4.0*/* ruby/gems/3.4.0/
   ```

4. 壓縮層內容：

   Linux/macOS

   ```
   zip -r layer.zip ruby/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\ruby -DestinationPath .\layer.zip
   ```

   .zip 檔案的目錄結構應如下所示：

   ```
   ruby/
   ### gems/
       ### 3.4.0/
           ### gems/
           #   ### concurrent-ruby-1.3.5/
           #   ### tzinfo-2.0.6/
           ### specifications/
           ### cache/
           ### build_info/
   ```

```
        ### (other bundler directories)
```

> **Note**
>
> 必須在函式程式碼中個別要求每個 gem。不能使用 `bundler/setup` 或 `Bundler.require`。如需更多詳細資訊，請參閱 在函式中使用來自層的 gem。

帶有原生延伸模組的 gem

許多熱門的 Ruby gem 都包含必須針對目標平台編譯的原生延伸模組 (C 程式碼)。帶有原生延伸模組的熱門 gem 包括 nokogiri、pg、mysql2、sqlite3 和 ffi。這些 gem 必須在與 Lambda 執行時期相容的 Linux 環境中建置。

使用帶有原生延伸模組的 gem 建立層

1. 建立 `Gemfile`。

   Example Gemfile

   ```
   source 'https://rubygems.org'

   gem 'nokogiri'
   gem 'httparty'
   ```

2. 使用 Docker，在與 Lambda 相容的 Linux 環境中建置 gem。在 Dockerfile 中指定 AWS 基礎映像：

   Example適用於 Ruby 3.4 的 Dockerfile

   ```
   FROM public.ecr.aws/lambda/ruby:3.4

   # Copy Gemfile
   COPY Gemfile ./

   # Install system dependencies for native extensions
   RUN dnf update -y && \
       dnf install -y gcc gcc-c++ make

   # Configure bundler and install gems
   RUN bundle config set --local path vendor/bundle && \
   ```

```
    bundle install

# Create the layer structure
RUN mkdir -p ruby/gems/3.4.0 && \
    cp -r vendor/bundle/ruby/3.4.0*/* ruby/gems/3.4.0/

# Create the layer zip file
RUN zip -r layer.zip ruby/
```

3. 建置映像並擷取層：

```
docker build -t ruby-layer-builder .
docker run --rm -v $(pwd):/output --entrypoint cp ruby-layer-builder layer.zip /
output/
```

這會在正確的 Linux 環境中建置 gem，並將 `layer.zip` 檔案複製至本機目錄。.zip 檔案的目錄結構應如下所示：

```
ruby/
### gems/
    ### 3.4.0/
        ### gems/
        #   ### bigdecimal-3.2.2/
        #   ### csv-3.3.5/
        #   ### httparty-0.23.1/
        #   ### mini_mime-1.1.5/
        #   ### multi_xml-0.7.2/
        #   ### nokogiri-1.18.8-x86_64-linux-gnu/
        #   ### racc-1.8.1/
        ### build_info/
        ### cache/
        ### specifications/
        ### (other bundler directories)
```

> **Note**
>
> 必須在函式程式碼中個別要求每個 gem。不能使用 `bundler/setup` 或 `Bundler.require`。如需更多詳細資訊，請參閱 在函式中使用來自層的 gem。

自訂 Ruby 模組

使用自訂的程式碼建立層

1. 建立層所需的目錄結構：

```
mkdir -p ruby/lib
```

2. 在 `ruby/lib` 目錄中建立 Ruby 模組。下列模組範例透過確認訂單包含所需資訊來驗證訂單。

Example ruby/lib/order_validator.rb

```
require 'json'

module OrderValidator
  class ValidationError < StandardError; end

  def self.validate_order(order_data)
    # Validates an order and returns formatted data
    required_fields = %w[product_id quantity]

    # Check required fields
    missing_fields = required_fields.reject { |field| order_data.key?(field) }
    unless missing_fields.empty?
      raise ValidationError, "Missing required fields: #{missing_fields.join(',
 ')}"
    end

    # Validate quantity
    quantity = order_data['quantity']
    unless quantity.is_a?(Integer) && quantity > 0
      raise ValidationError, 'Quantity must be a positive integer'
    end

    # Format and return the validated data
    {
      'product_id' => order_data['product_id'].to_s,
      'quantity' => quantity,
      'shipping_priority' => order_data.fetch('priority', 'standard')
    }
  end

  def self.format_response(status_code, body)
```

```
    # Formats the API response
    {
      statusCode: status_code,
      body: JSON.generate(body)
    }
  end
end
```

3. 壓縮層內容：

   Linux/macOS

   ```
   zip -r layer.zip ruby/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\ruby -DestinationPath .\layer.zip
   ```

   .zip 檔案的目錄結構應如下所示：

   ```
   ruby/
   ### lib/
       ### order_validator.rb
   ```

4. 在函式中，要求並使用這些模組。必須在函式程式碼中個別要求每個 gem。不能使用 `bundler/setup` 或 `Bundler.require`。如需更多詳細資訊，請參閱 在函式中使用來自層的 gem。範例：

   ```
   require 'json'
   require 'order_validator'

   def lambda_handler(event:, context:)
     begin
       # Parse the order data from the event body
       order_data = JSON.parse(event['body'] || '{}')

       # Validate and format the order
       validated_order = OrderValidator.validate_order(order_data)

       OrderValidator.format_response(200, {
         message: 'Order validated successfully',
   ```

```
      order: validated_order
    })
  rescue OrderValidator::ValidationError => e
    OrderValidator.format_response(400, {
      error: e.message
    })
  rescue => e
    OrderValidator.format_response(500, {
      error: 'Internal server error'
    })
  end
end
```

您可以使用下列測試事件來調用函式：

```
{
    "body": "{\"product_id\": \"ABC123\", \"quantity\": 2, \"priority\": \"express
\"}"
}
```

預期回應：

```
{
  "statusCode": 200,
  "body": "{\"message\":\"Order validated successfully\",\"order\":{\"product_id\":
\"ABC123\",\"quantity\":2,\"shipping_priority\":\"express\"}}"
}
```

在 Lambda 中建立層

您可以使用 AWS CLI 或 Lambda 主控台來發布層。

AWS CLI

執行 publish-layer-version AWS CLI 命令來建立 Lambda 層：

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes ruby3.4
```

相容的執行時期參數為選用參數。指定此參數後，Lambda 會據此在 Lambda 主控台中篩選層。

Console

建立圖層 (主控台)

1. 開啟 Lambda 主控台中的 層頁面。
2. 選擇 建立圖層。
3. 選擇上傳 .zip 檔案，然後上傳先前建立的 .zip 壓縮檔。
4. (選用) 在相容的執行時期欄位中，選擇與用於建置層的 Ruby 版本相對應的 Ruby 執行時期。
5. 選擇建立。

在函式中使用來自層的 gem

在函式程式碼中，必須明確要求要使用的每個 gem。不支援 `bundler/setup` 和 `Bundler.require` 等封裝程式命令。以下是在 Lambda 函式中正確使用來自層的 gem 的執行方法：

```
# Correct: Use explicit requires for each gem
require 'nokogiri'
require 'httparty'

def lambda_handler(event:, context:)
  # Use the gems directly
  doc = Nokogiri::HTML(event['html'])
  response = HTTParty.get(event['url'])
  # ... rest of your function
end

# Incorrect: These Bundler commands will not work
# require 'bundler/setup'
# Bundler.require
```

將層新增至函式

AWS CLI

若要將層連接至函式，請執行 update-function-configuration AWS CLI 命令。對於 `--layers` 參數，請使用層 ARN。ARN 必須指定版本 (例如 `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`)。如需更多詳細資訊，請參閱 層和層的版本。

```
aws lambda update-function-configuration --function-name my-function --cli-binary-
format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-
layer:1"
```

如果您使用 AWS CLI 第 2 版，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

Console

將層新增至函式

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函式。
3. 向下捲動至層區段，然後選擇新增層。
4. 在選擇層欄位中，選取自訂層，然後選擇要使用的層。

> **Note**
>
> 如果未在建立層時新增相容的執行時期，此處不會列出層。您可以改為指定層 ARN。

5. 選擇新增。

應用程式範例

如需如何使用 Lambda 層的更多範例，請參閱 AWS Lambda Developer Guide GitHub repository 中的 layer-ruby 範例應用程式。此應用程式包含一個內含 tzinfo 程式庫的層。建立層之後，您可以部署並調用相應的函式，驗證層是否如預期那樣運作。

使用 Lambda 內容物件擷取 Ruby 函數資訊

當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。此物件提供的方法和各項屬性提供了有關調用、函式以及執行環境的資訊。

內容方法

- `get_remaining_time_in_millis` - 傳回執行逾時前剩餘的毫秒數。

內容屬性

- `function_name` – Lambda 函數的名稱。
- `function_version` – 函數的版本。
- `invoked_function_arn` - 用於叫用此函數的 Amazon Resource Name (ARN)。指出調用者是否指定版本號或別名。
- `memory_limit_in_mb` - 分配給函數的記憶體數量。
- `aws_request_id` - 調用請求的識別符。
- `log_group_name` - 函數的日誌群組。
- `log_stream_name` - 函數執行個體的記錄串流。
- `deadline_ms` - 執行逾時的日期，以 Unix 時間毫秒為單位。
- `identity` - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
- `client_context` - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。

記錄和監控 Ruby Lambda 函數

AWS Lambda 會自動代表您監控 Lambda 函數，並將日誌傳送至 Amazon CloudWatch。您的 Lambda 函數隨附有 CloudWatch Logs 日誌群組，且函數的每一執行個體各有一個日誌串流。Lambda 執行期環境會將每次調用的詳細資訊傳送至日誌串流，並且轉傳來自函數程式碼的日誌及其他輸出。如需詳細資訊，請參閱將 Lambda 函式日誌傳送至 CloudWatch Logs。

此頁面說明如何從 Lambda 函數的程式碼產生日誌輸出，並使用 AWS Command Line Interface、Lambda 主控台或 CloudWatch 主控台存取日誌。

章節

- 建立傳回日誌的函數
- 在 Lambda 主控台檢視日誌
- 在 CloudWatch 主控台中檢視 記錄
- 使用 AWS Command Line Interface (AWS CLI) 檢視日誌
- 刪除日誌
- 使用 Ruby 記錄程式庫

建立傳回日誌的函數

若要由您的函式程式碼輸出日誌，可使用 `puts` 陳述式或者任何能夠寫入 `stdout` 或 `stderr` 的記錄程式庫。下列範例會記錄環境變數和事件物件的值。

Example lambda_function.rb

```
# lambda_function.rb

def handler(event:, context:)
    puts "## ENVIRONMENT VARIABLES"
    puts ENV.to_a
    puts "## EVENT"
    puts event.to_a
end
```

Example記錄格式

```
START RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95 Version: $LATEST
## ENVIRONMENT VARIABLES
```

```
environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]3893xmpl7fac4485b47bb75b671a283c',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})
## EVENT
{'key': 'value'}
END RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95
REPORT RequestId: 8f507cfc-xmpl-4697-b07a-ac58fc914c95  Duration: 15.74 ms  Billed
 Duration: 147 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 130.49 ms
XRAY TraceId: 1-5e34a614-10bdxmplf1fb44f07bc535a1   SegmentId: 07f5xmpl2d1f6f85
 Sampled: true
```

Ruby 執行時間會記錄每次調用的 START、END 和 REPORT 行。報告明細行提供下列詳細資訊。

REPORT 行資料欄位

- RequestId - 進行調用的唯一請求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。當調用共用執行環境時，Lambda 會報告所有調用使用的記憶體上限。此行為可能會導致報告值高於預期值。
- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId - 對於追蹤的請求，這是 AWS X-Ray 追蹤 ID。
- SegmentId - 對於追蹤的請求，這是 X-Ray 區段 ID。
- 已取樣 - 對於追蹤的請求，這是取樣結果。

如需更詳細的日誌，請使用 the section called “使用 Ruby 記錄程式庫”。

在 Lambda 主控台檢視日誌

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

在 CloudWatch 主控台中檢視 記錄

您可以使用 Amazon CloudWatch 主控台來檢視所有 Lambda 函數調用的日誌。

若要在 CloudWatch 主控台上檢視日誌

1. 在 CloudWatch 主控台上開啟日誌群組頁面。
2. 選擇您的函數的日誌群組 (/aws/lambda/*your-function-name*)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立其他執行個體以處理並行調用時，就會顯示日誌串流。若要尋找特定調用的日誌，建議您使用 檢測函數 AWS X-Ray。X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

使用 AWS Command Line Interface (AWS CLI) 檢視日誌

是一種 AWS CLI 開放原始碼工具，可讓您使用命令列 Shell 中的 命令與 AWS 服務互動。若要完成本節中的步驟，您必須擁有 AWS CLI 版本 2。

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example解碼日誌

在相同的命令提示中，使用 `base64` 公用程式來解碼日誌。下列範例顯示如何擷取 `my-function` 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
```

```
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 `base64` 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 `base64 -D`。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 `sed` 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 `get-log-events` 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 `get-logs.sh`。

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
```

```
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

使用 Ruby 記錄程式庫

Ruby 記錄程式庫會傳回易於讀取的精簡日誌。使用記錄公用程式輸出與函數相關的詳細資訊、訊息和錯誤碼。

```
# lambda_function.rb

require 'logger'

def handler(event:, context:)
  logger = Logger.new($stdout)
  logger.info('## ENVIRONMENT VARIABLES')
  logger.info(ENV.to_a)
  logger.info('## EVENT')
  logger.info(event)
  event.to_a
end
```

來自 `logger` 的輸出包含記錄等級、時間戳記和請求 ID。

```
START RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125 Version: $LATEST
[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ##
 ENVIRONMENT VARIABLES

[INFO]  2020-01-31T22:12:58.534Z    1c8df7d3-xmpl-46da-9778-518e6eca8125
    environ({'AWS_LAMBDA_LOG_GROUP_NAME': '/aws/lambda/my-function',
```

```
 'AWS_LAMBDA_LOG_STREAM_NAME': '2020/01/31/[$LATEST]1bbe51xmplb34a2788dbaa7433b0aa4d',
 'AWS_LAMBDA_FUNCTION_NAME': 'my-function', ...})

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    ## EVENT

[INFO]  2020-01-31T22:12:58.535Z    1c8df7d3-xmpl-46da-9778-518e6eca8125    {'key':
 'value'}

END RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125
REPORT RequestId: 1c8df7d3-xmpl-46da-9778-518e6eca8125  Duration: 2.75 ms   Billed
 Duration: 117 ms Memory Size: 128 MB Max Memory Used: 56 MB  Init Duration: 113.51 ms
XRAY TraceId: 1-5e34a66a-474xmpl7c2534a87870b4370   SegmentId: 073cxmpl3e442861
 Sampled: true
```

在 中檢測 Ruby 程式碼 AWS Lambda

Lambda 與 整合 AWS X-Ray ，可讓您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追踪請求，因為它會周遊應用程式中的資源，從前端 API 到後端的存儲體和資料庫。只要將 X-Ray SDK 程式庫新增至建置組態，您就可以記錄函數對 AWS 服務所做的任何呼叫的錯誤和延遲。

設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 服務圖顯示了有關應用程式及其所有元件的資訊。下列範例顯示了一個具有兩個函數的應用程式。主要函式會處理事件，有時會傳回錯誤。頂端的第二個函數會處理出現在第一個日誌群組中的錯誤，並使用 AWS SDK 呼叫 X-Ray、Amazon Simple Storage Service (Amazon S3) 和 Amazon CloudWatch Logs。

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

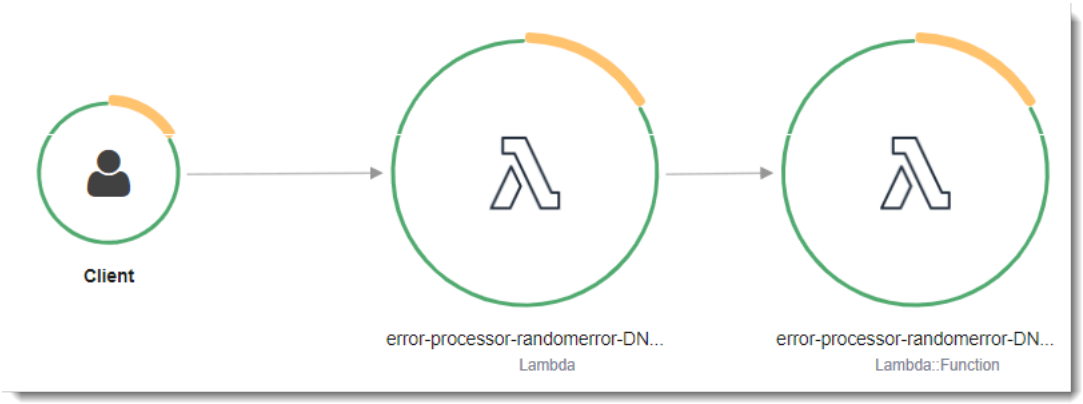
1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態，然後選擇監控和操作工具。
4. 在其他監控工具欄位中，選擇編輯。
5. 在 CloudWatch Application Signals 和 AWS X-Ray 下，選擇啟用 Lambda 服務追蹤。
6. 選擇儲存。

> **定價**
>
> 您可以在每月免費使用 X-Ray 追蹤，最高可達特定限制，作為 AWS 免費方案的一部分。達到閾值後，X-Ray 會收取追蹤儲存及擷取的費用。如需詳細資訊，請參閱 AWS X-Ray 定價。

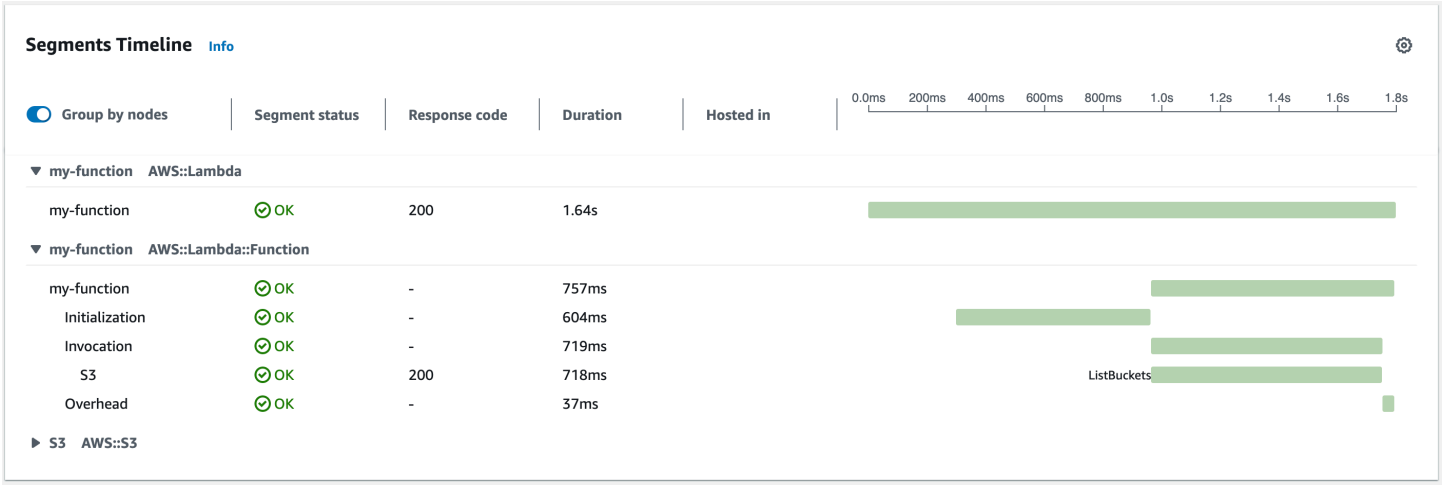
您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將 AWSXRayDaemonWriteAccess 政策新增至執行角色。

X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。不能針對函數設定 X-Ray 取樣率。

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會在每個追蹤上記錄 2 個區段，這會在服務圖表上建立兩個節點。下圖反白顯示了這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為 my-function，但其中之一的來源為 `AWS::Lambda`，而另一個的來源為 `AWS::Lambda::Function`。如果 `AWS::Lambda` 區段顯示錯誤，Lambda 服務就會出現問題。如果 `AWS::Lambda::Function` 區段顯示錯誤，表示您的函數出現了問題。



此範例會展開 `AWS::Lambda::Function` 區段以顯示其三個子區段：

> **Note**
>
> AWS 目前正在對 Lambda 服務實作變更。由於這些變更，您可能會看到系統日誌訊息的結構和內容，與 AWS 帳戶中不同 Lambda 函數發出的追蹤區段之間存在細微差異。
> 此處顯示的追蹤範例說明了舊式函數區段。下列段落說明了舊式和新式區段之間的差異。
> 這些變化將在未來幾週內實作，除中國和 GovCloud 區域以外，所有 AWS 區域 中的所有函數都會轉換至使用新格式的日誌訊息和追蹤區段。

舊式函數區段包含下列子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行期為做好準備以處理下一個事件所花費的時間。

新式函數區段不包含 `Invocation` 子區段。相反地，客戶子區段會直接連接至函數區段。如需舊式和新式函數區段結構的詳細資訊，請參閱the section called “了解 X-Ray 追蹤”。

您可以測試處理常式程式碼，以記錄中繼資料並追蹤下游呼叫。若要記錄處理常式對其他資源和服務所進行之呼叫的詳細資料，請使用適用於 Ruby 的 X-Ray 開發套件。若要取得開發套件，請將 `aws-xray-sdk` 套件新增至應用程式的相依性。

Example blank-ruby/function/Gemfile

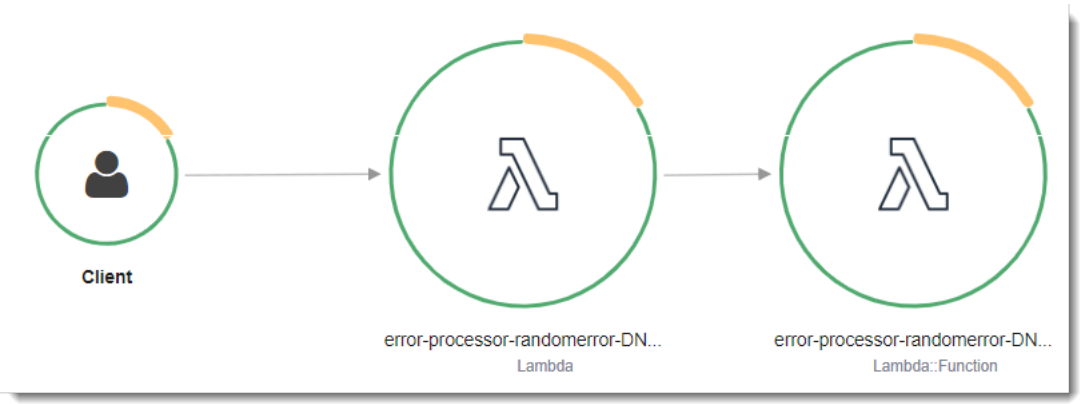
```
# Gemfile
source 'https://rubygems.org'

gem 'aws-xray-sdk', '0.11.4'
gem 'aws-sdk-lambda', '1.39.0'
gem 'test-unit', '3.3.5'
```

若要檢測 AWS SDK 用戶端，請在初始化程式碼中建立用戶端後要求 `aws-xray-sdk/lambda`模組。

Example blank-ruby/function/lambda_function.rb – 追蹤 AWS SDK 用戶端

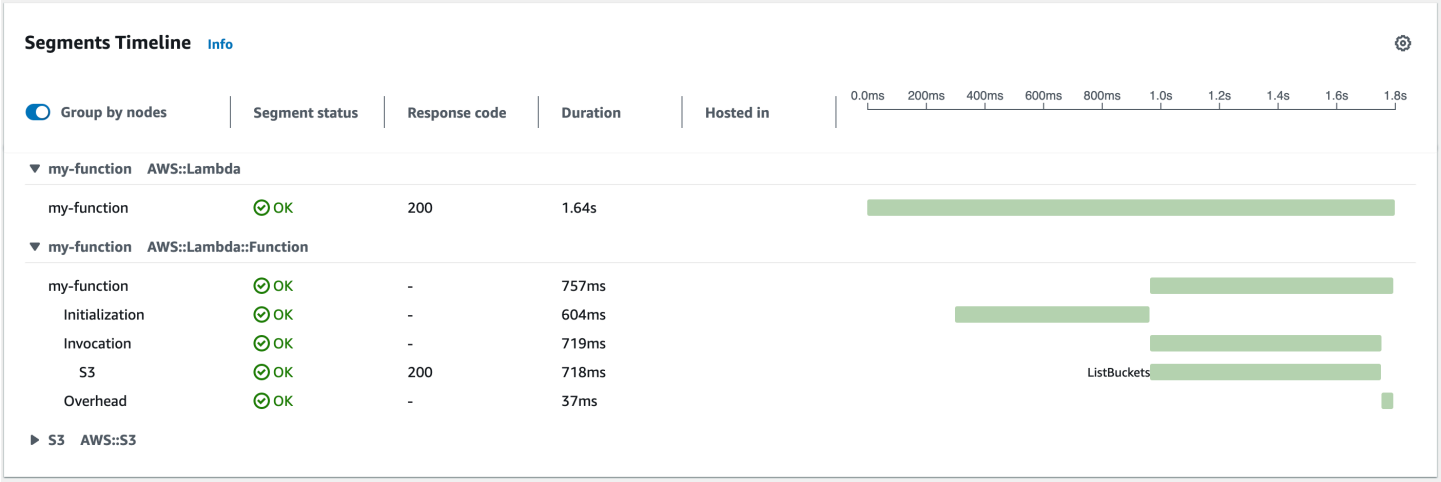
```
# lambda_function.rb
require 'logger'
require 'json'
require 'aws-sdk-lambda'
$client = Aws::Lambda::Client.new()
$client.get_account_settings()

require 'aws-xray-sdk/lambda'

def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  ...
```

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會在每個追蹤上記錄 2 個區段，這會在服務圖表上建立兩個節點。下圖反白顯示了這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為 my-function，但其中之一的來源為

AWS::Lambda，而另一個的來源為 AWS::Lambda::Function。如果 AWS::Lambda 區段顯示錯誤，Lambda 服務就會出現問題。如果 AWS::Lambda::Function 區段顯示錯誤，表示您的函數出現了問題。



此範例會展開 AWS::Lambda::Function 區段以顯示其三個子區段：

> **Note**
>
> AWS 目前正在對 Lambda 服務實作變更。由於這些變更，您可能會看到系統日誌訊息的結構和內容，與 AWS 帳戶中不同 Lambda 函數發出的追蹤區段之間存在細微差異。
> 此處顯示的追蹤範例說明了舊式函數區段。下列段落說明了舊式和新式區段之間的差異。
> 這些變化將在未來幾週內實作，除中國和 GovCloud 區域以外，所有 AWS 區域 中的所有函數都會轉換至使用新格式的日誌訊息和追蹤區段。

舊式函數區段包含下列子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行期為做好準備以處理下一個事件所花費的時間。

新式函數區段不包含 Invocation 子區段。相反地，客戶子區段會直接連接至函數區段。如需舊式和新式函數區段結構的詳細資訊，請參閱the section called “了解 X-Ray 追蹤”。

您也可以檢測 HTTP 用戶端、記錄 SQL 查詢，以及建立具有註釋和中繼資料的自訂子區段。如需詳細資訊，請參閱《 開發人員指南》中的適用於 Ruby 的 X-Ray 開發套件。 AWS X-Ray

章節

- 透過 Lambda API 啟用主動追蹤
- 使用 啟用主動追蹤 CloudFormation
- 將執行時間相依性儲存在圖層中

透過 Lambda API 啟用主動追蹤

若要使用 AWS CLI 或 AWS SDK 管理追蹤組態，請使用下列 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

下列範例 AWS CLI 命令會在名為 my-function 的函數上啟用主動追蹤。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

使用 啟用主動追蹤 CloudFormation

若要在 CloudFormation 範本中的 `AWS::Lambda::Function` 資源上啟用追蹤，請使用 `TracingConfig` 屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用 `Tracing` 屬性。

Example template.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

將執行時間相依性儲存在圖層中

如果您使用 X-Ray 開發套件來檢測函數程式碼的 AWS SDK 用戶端，您的部署套件可能會變得相當大。為了避免每次更新函數程式碼時上傳執行時間相依性，請將 X-Ray SDK 封裝在一個 Lambda 層中。

下面的例子顯示了儲存適用於 Ruby 的 X-Ray 開發套件的 `AWS::Serverless::LayerVersion` 資源。

Example template.yml - 相依性層

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: function/.
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-ruby-lib
      Description: Dependencies for the blank-ruby sample app.
      ContentUri: lib/.
      CompatibleRuntimes:
        - ruby2.5
```

透過此組態，您只有在變更執行時間相依性時才會更新程式庫層。由於函數部署套件僅含有您的程式碼，因此有助於減少上傳時間。

為相依性建立圖層需要建置變更，才能在部署之前產生圖層封存。如需工作範例，請參閱 blank-ruby 範例應用程式。

使用 Java 建置 Lambda 函數

您可以在 AWS Lambda中執行 Java 程式碼。Lambda 提供用於執行程式碼來處理事件的 Java 執行期。您的程式碼會在 Amazon Linux 環境中執行，其中包含您管理之 AWS Identity and Access Management (IAM) 角色的 AWS 登入資料。

Lambda 支援以下 Java 執行期。

Name	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
Java 25	`java25`	Amazon Linux 2023	2029 年 6 月 30 日	2029 年 7 月 31 日	2029 年 8 月 31 日
Java 21	`java21`	Amazon Linux 2023	2029 年 6 月 30 日	2029 年 7 月 31 日	2029 年 8 月 31 日
Java 17	`java17`	Amazon Linux 2	2027 年 6 月 30 日	2027 年 7 月 31 日	2027 年 8 月 31 日
Java 11	`java11`	Amazon Linux 2	2027 年 6 月 30 日	2027 年 7 月 31 日	2027 年 8 月 31 日
Java 8	`java8.al2`	Amazon Linux 2	2027 年 6 月 30 日	2027 年 7 月 31 日	2027 年 8 月 31 日

AWS 為 Java 函數提供下列程式庫。這些程式庫可透過 Maven Central Repository 取得。

- com.amazonaws:aws-lambda-java-core (必要) - 定義處理常式方法介面，以及執行期傳遞給處理常式的內容物件。如果您定義自己的輸入類型，這是您需要的唯一程式庫。
- com.amazonaws:aws-lambda-java-events - 來自調用 Lambda 函數的服務的輸入事件類型。
- com.amazonaws:aws-lambda-java-log4j2 - 一個 Apache Log4j 2 的附加程式庫，您可以使用它將當前調用的請求 ID 新增至函數日誌。
- AWS 適用於 Java 的 SDK 2.0 – 適用於 Java 程式設計語言的官方 AWS SDK。

請將這些程式庫新增至建置定義中，如下所示：

Gradle

```
dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
}
```

Maven

```
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.2.2</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-events</artifactId>
      <version>3.11.1</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-log4j2</artifactId>
      <version>1.5.1</version>
    </dependency>
  </dependencies>
```

Important

請勿使用 JDK API 的私有元件，如私有欄位、方法或類別。非公有 API 元件可能在任何更新中發生變更或被移除，導致您的應用程式中斷。

若要建立 Lambda 函數

1. 開啟 Lambda 主控台。
2. 選擇建立函數。
3. 進行下列設定：

- 函數名稱：輸入函數名稱。
- 執行時間：選擇 Java 25。

4. 選擇建立函數。

主控台將建立一個 Lambda 函數，它具有名為 Hello 的處理常式類別。由於 Java 是一種編譯語言，因此您無法在 Lambda 主控台中檢視或編輯原始碼，但可以修改其組態，加以調用，並設定觸發條件。

> **Note**
>
> 若要開始在您的本機環境中開發應用程式，請部署本指南 GitHub 儲存庫中可用的其中一個範例應用程式。

Hello 類別有一個名為 handleRequest 的函式，其接受事件物件與內容物件。這就是在調用函數時，Lambda 呼叫的處理常式函數。Java 函數執行期會從 Lambda 取得調用事件並其傳遞至處理常式。在函式組態中，處理常式值為 example.Hello::handleRequest。

若要更新函數的程式碼，您可建立部署套件，這是包含函數程式碼的 ZIP 封存檔。隨著函式開發的進展，您需要將函式程式碼存放於原始碼控制系統、加入程式庫並進行自動化部署。首先，透過命令列建立部署套件並更新您的程式碼。

除了傳遞調用事件外，函式執行期還會傳遞內容物件至處理常式。內容物件包含了有關調用、函式以及執行環境的額外資訊。更多詳細資訊將另由環境變數提供。

Lambda 函數隨附有 CloudWatch Logs 記錄群組。函數執行期會將每次調用的詳細資訊傳送至 CloudWatch Logs。它在調用期間會轉送您的函數輸出的任何記錄。如果您的函數傳回錯誤，Lambda 會對該錯誤進行格式化之後傳回給調用端。

主題

- 在 Java 中定義 Lambda 函數處理常式
- 使用 .zip 或 JAR 封存檔部署 Java Lambda 函數
- 使用容器映像部署 Java Lambda 函數
- 使用 Java Lambda 函數的層
- 自訂 Lambda Java 函數的序列化
- 自訂 Lambda 函數的 Java 執行時期啟動行為

- 使用 Lambda 內容物件擷取 Java 函數資訊
- 記錄和監控 Java Lambda 函數
- 在 中檢測 Java 程式碼 AWS Lambda
- 的 Java 範例應用程式 AWS Lambda

在 Java 中定義 Lambda 函數處理常式

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

本頁介紹了如何在 Java 中使用 Lambda 函式處理常式，包括專案設定選項、命名慣例及最佳實務。本頁還提供了一個 Java Lambda 函式範例，該函式會取得訂單的相關資訊、產生文字檔案收據，並將該檔案放入 Amazon Simple Storage Service (Amazon S3) 儲存貯體。如需編寫函數後如何部署函數的詳細資訊，請參閱the section called “部署 .zip 封存檔”或the section called “部署容器映像”。

章節

- 設定 Java 處理常式專案
- Java Lambda 函式程式碼範例
- Java 處理常式的有效類別定義
- 處理常式命名慣例
- 定義和存取輸入事件物件
- 存取和使用 Lambda 內容物件
- 在處理常式中使用適用於 Java v2 的 AWS SDK
- 存取環境變數
- 使用全域狀態
- Java Lambda 函數的程式碼最佳實務

設定 Java 處理常式專案

使用 Java 開發 Lambda 函式時，流程涉及編寫程式碼、編譯程式碼，以及將編譯後的成品部署至 Lambda。可以透過各種方式初始化 Java Lambda 專案。例如，您可以使用下列工具：適用於 Lambda 函式的 Maven Archetype、AWS SAM CLI 的 sam init 命令，甚或在偏好的 IDE (例如 IntelliJ IDEA 或 Visual Studio Code) 中建立標準 Java 專案設定。或者，您可以手動建立所需的檔案結構。

一個典型的 Java Lambda 函式專案遵循以下一般結構：

```
/project-root
    # src
        # main
            # java
                # example
                    # OrderHandler.java (contains main handler)
```

```
                    # <other_supporting_classes>
    # build.gradle OR pom.xml
```

可以使用 Maven 或 Gradle 建置專案並管理相依項。

函式的主要處理常式邏輯位於 `src/main/java/example` 目錄下的 Java 檔案中。在此頁面上的範例中，我們將該檔案命名為 `OrderHandler.java`。除了該檔案外，您亦可視需要納入其他 Java 類別。將函式部署至 Lambda 時，請務必指定包含 Lambda 在調用期間應調用的主要處理常式方法的 Java 類別。

Java Lambda 函式程式碼範例

以下 Java 21 Lambda 函式程式碼範例會取得訂單的相關資訊、產生文字檔案收據，並將該檔案放入 Amazon S3 儲存貯體。

Example **`OrderHandler.java`** Lambda 函數

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import software.amazon.awssdk.core.sync.RequestBody;
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;
import software.amazon.awssdk.services.s3.model.S3Exception;

import java.nio.charset.StandardCharsets;

/**
 * Lambda handler for processing orders and storing receipts in S3.
 */
public class OrderHandler implements RequestHandler<OrderHandler.Order, String> {

    private static final S3Client S3_CLIENT = S3Client.builder().build();

    /**
     * Record to model the input event.
     */
    public record Order(String orderId, double amount, String item) {}

    @Override
    public String handleRequest(Order event, Context context) {
        try {
```

```
            // Access environment variables
            String bucketName = System.getenv("RECEIPT_BUCKET");
            if (bucketName == null || bucketName.isEmpty()) {
                throw new IllegalArgumentException("RECEIPT_BUCKET environment variable
 is not set");
            }

            // Create the receipt content and key destination
            String receiptContent = String.format("OrderID: %s\nAmount: $%.2f\nItem:
 %s",
                    event.orderId(), event.amount(), event.item());
            String key = "receipts/" + event.orderId() + ".txt";

            // Upload the receipt to S3
            uploadReceiptToS3(bucketName, key, receiptContent);

            context.getLogger().log("Successfully processed order " + event.orderId() +
                    " and stored receipt in S3 bucket " + bucketName);
            return "Success";

        } catch (Exception e) {
            context.getLogger().log("Failed to process order: " + e.getMessage());
            throw new RuntimeException(e);
        }
    }

    private void uploadReceiptToS3(String bucketName, String key, String
 receiptContent) {
        try {
            PutObjectRequest putObjectRequest = PutObjectRequest.builder()
                    .bucket(bucketName)
                    .key(key)
                    .build();

            // Convert the receipt content to bytes and upload to S3
            S3_CLIENT.putObject(putObjectRequest,
 RequestBody.fromBytes(receiptContent.getBytes(StandardCharsets.UTF_8)));
        } catch (S3Exception e) {
            throw new RuntimeException("Failed to upload receipt to S3: " +
 e.awsErrorDetails().errorMessage(), e);
        }
    }
}
```

此 OrderHandler.java 檔案包含以下程式碼區段：

- package example：在 Java 中，此套件名稱可自訂，但必須與專案的目錄結構相符。本處採用 package example，原因在於目錄結構為 src/main/java/example。
- import 陳述式：用於匯入 Lambda 函式所需的 Java 類別。
- public class OrderHandler ...：此程式碼會定義您的 Java 類別，且必須為有效的類別定義。
- private static final S3Client S3_CLIENT ...：此程式碼在類別的所有方法外部初始化 S3 用戶端。這會使 Lambda 在初始化階段執行此程式碼。
- public record Order ...：在此自訂 Java 記錄中定義預期輸入事件的結構。
- public String handleRequest(Order event, Context context)：這是主要處理常式方法，其中包含應用程式的主要邏輯。
- private void uploadReceiptToS3(...) {}：這是主要 handleRequest 處理常式方法所參考的協助程式方法。

build.gradle 與 pom.xml 檔案範例

此函式附帶下列 build.gradle 或 pom.xml 檔案。

build.gradle

```
plugins {
    id 'java'
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.3'
    implementation 'software.amazon.awssdk:s3:2.28.29'
    implementation 'org.slf4j:slf4j-nop:2.0.16'
}

task buildZip(type: Zip) {
    from compileJava
    from processResources
    into('lib') {
```

```
        from configurations.runtimeClasspath
    }
}

java {
    sourceCompatibility = JavaVersion.VERSION_21
    targetCompatibility = JavaVersion.VERSION_21
}

build.dependsOn buildZip
```

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.example</groupId>
    <artifactId>example-java</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>example-java-function</name>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <maven.compiler.source>21</maven.compiler.source>
        <maven.compiler.target>21</maven.compiler.target>
    </properties>
    <dependencies>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.3</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.awssdk</groupId>
            <artifactId>s3</artifactId>
            <version>2.28.29</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-nop</artifactId>
            <version>2.0.16</version>
```

```
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>3.5.2</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.4.1</version>
                <configuration>
                    <createDependencyReducedPom>false</createDependencyReducedPom>
                    <filters>
                        <filter>
                            <artifact>*:*</artifact>
                            <excludes>
                                <exclude>META-INF/*</exclude>
                                <exclude>META-INF/versions/**</exclude>
                            </excludes>
                        </filter>
                    </filters>
                </configuration>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.13.0</version>
                <configuration>
                    <release>21</release>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

```
</project>
```

若要讓此函式正常運作，其執行角色必須允許 `s3:PutObject` 動作。此外，請確保定義 `RECEIPT_BUCKET` 環境變數。成功調用後，Amazon S3 儲存貯體應包含收據檔案。

> **Note**
>
> 此函式可能需要額外的組態設定，才能順利執行而不發生逾時。建議設定 256 MB 記憶體以及 10 秒逾時。首次調用可能因冷啟動而耗費更長時間。後續調用因執行環境可重複使用，執行速度會大幅提升。

Java 處理常式的有效類別定義

為了定義類別，aws-lambda-java-core 程式庫定義了處理常式方法的兩個介面。透過使用這些預設介面，可簡化處理常式的設定流程，並在編譯階段驗證方法簽章。

- com.amazonaws.services.lambda.runtime.RequestHandler
- com.amazonaws.services.lambda.runtime.RequestStreamHandler

`RequestHandler` 介面是一個一般類型，它有兩個參數：輸入類型和輸出類型。兩種類型都必須是物件。在此範例中，`OrderHandler` 類別會實作 `RequestHandler<OrderHandler.Order, String>`。輸入類型是我們在類別中定義的 `Order` 記錄，輸出類型是 `String`。

```
public class OrderHandler implements RequestHandler<OrderHandler.Order, String> {
    ...
}
```

當您使用此介面時，Java 執行時期會將事件反序列化為具有輸入類型的物件，並將輸出序列化為文字。當內建序列化與您的輸入和輸出類型一同作業時，請使用此介面。

若要使用自訂序列化，您可以實作 `RequestStreamHandler` 介面。透過此介面，Lambda 會將處理常式傳遞給一個輸入串流和輸出串流。處理常式會從輸入串流讀取位元組，寫入到輸出串流，並傳回 void 值。如需使用 Java 21 執行時期的實作範例，請參閱 HandlerStream.java。

若您的 Java 函式僅使用基本類型與泛型類型 (即 `String`、`Integer`、`List` 或 `Map`)，則無需實作介面。例如，若函式接收一個 `Map<String, String>` 輸入並傳回 `String`，則類別定義和處理常式簽章可能如下所示：

```
public class ExampleHandler {
    public String handleRequest(Map<String, String> input, Context context) {
        ...
    }
}
```

此外，如果未實作介面，則內容物件是選用項目。例如，類別定義和處理常式簽章可能如下所示：

```
public class NoContextHandler {
   public String handleRequest(Map<String, String> input) {
        ...
   }
}
```

處理常式命名慣例

對於 Java 中的 Lambda 函式，如果您實作 RequestHandler 或 RequestStreamHandler 介面，則主要處理常式方法必須命名為 handleRequest。此外，請在 handleRequest 方法上方加上 @Override 標籤。將函式部署到 Lambda 時，請以下列格式在函式的組態中指定主要處理常式：

- *<package>*.*<Class>* – 例如 example.OrderHandler。

對於 Java 中未實作 RequestHandler 或 RequestStreamHandler 介面的 Lambda 函式，您可以將處理常式命名為任何名稱。將函式部署到 Lambda 時，請以下列格式在函式的組態中指定主要處理常式：

- *<package>*.*<Class>*::*<handler_method_name>* – 例如 example.Handler::mainHandler。

定義和存取輸入事件物件

JSON 是 Lambda 函數最常見的標準輸入格式。在此範例中，函數預期輸入類似以下內容：

```
{
    "orderId": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

使用 Java 17 或更新版本開發 Lambda 函式時，可將預期輸入事件的結構定義為 Java 記錄。在此範例中，我們定義了 `OrderHandler` 類別中的一條記錄來代表 `Order` 物件：

```
public record Order(String orderId, double amount, String item) {}
```

該記錄符合預期的輸入結構。定義記錄之後，您即可編寫處理常式簽章，其會接收符合該記錄定義的 JSON 輸入。Java 執行時期會自動將此 JSON 反序列化為 Java 物件。然後，您可以存取該物件的欄位。例如，使用 `orderId` 可以從原始輸入擷取 `event.orderId` 的值。

> **Note**
>
> 僅 Java 17 及更新版本的執行時期支援 Java 記錄功能。在所有 Java 執行期中，可以使用類別來表示事件資料。在此類案例中，您可以使用 jackson 之類的程式庫來反序列化 JSON 輸入。

其他輸入事件類型

Java 中的 Lambda 函式支援多種可能的輸入事件：

- `Integer` `Long`、`Double`、等等 - 事件是沒有其他格式的數字，例如 `3.5`。Java 執行時期會將值轉換為指定類型的物件。
- `String` - 事件是 JSON 字串，包括引號，例如 。`“My string”`執行時期會將值轉換為不含引號的 `String` 物件。
- `List<Integer>` `List<String>`、`List<Object>`、等等 - 該事件是一個 JSON 陣列。執行階段會將它還原序列化為指定類型或介面的物件。
- `InputStream` - 該事件是任何 JSON 類型。執行階段會將文件的位元組串流傳遞給處理常式而不進行修改。您還原序列化輸入和並將輸出寫入輸出串流。
- 程式庫類型 – 對於由其他 AWS 服務傳送的事件，請使用 aws-lambda-java-events 程式庫中的類型。例如，若 Amazon Simple Queue Service (SQS) 調用您的 Lambda 函式，則以 `SQSEvent` 物件作為輸入。

存取和使用 Lambda 內容物件

Lambda 內容物件 包含有關調用、函數以及執行環境的資訊。在此範例中，內容物件的類型為 `com.amazonaws.services.lambda.runtime.Context`，並且內容物件也是主要處理常式函式的第二個引數。

```
public String handleRequest(Order event, Context context) {
    ...
}
```

如果您的類別實作了 RequestHandler 或 RequestStreamHandler 介面，則內容物件為必要引數。否則，內容物件為選用引數。如需有關可接受的有效處理常式簽章的詳細資訊，請參閱 the section called "Java 處理常式的有效類別定義"。

如果您使用 AWS SDK 呼叫其他服務，則有幾個關鍵區域需要內容物件。例如，若要為 Amazon CloudWatch 生成函式日誌，您可以使用 `context.getLogger()` 方法取得用於記錄的 `LambdaLogger` 物件。在此範例中，如果處理因任何原因失敗，我們可以使用記錄器來記錄錯誤訊息：

```
context.getLogger().log("Failed to process order: " + e.getMessage());
```

除了記錄日誌之外，您還可以將內容物件用於函式監控。如需內容物件的詳細資訊，請參閱the section called "Context"。

在處理常式中使用適用於 Java v2 的 AWS SDK

通常，您會使用 Lambda 函式與其他 AWS 資源互動或對其進行更新。與這些資源互動的最簡單方法便是使用適用於 Java v2 的 AWS SDK。

> **Note**
>
> 適用於 Java (v1) 的 AWS SDK 目前處於維護模式，將於 2025 年 12 月 31 日停止支援。建議今後僅使用適用於 Java v2 的 AWS SDK。

若要將 SDK 相依項新增至函式，請在適用於 Gradle 的 `build.gradle` 檔案或適用於 Maven 的 `pom.xml` 檔案中納入這些相依項。建議僅新增函式所需的程式庫。在先前的範例程式碼中，我們使用了 `software.amazon.awssdk.services.s3` 程式庫。在 Gradle 中，您可以透過在 `build.gradle` 的相依項區段中新增以下一行程式碼來新增此相依項：

```
implementation 'software.amazon.awssdk:s3:2.28.29'
```

在 Maven 中，在 `pom.xml` 的 `<dependencies>` 區段中新增以下程式碼行：

```
    <dependency>
```

```
        <groupId>software.amazon.awssdk</groupId>
        <artifactId>s3</artifactId>
        <version>2.28.29</version>
    </dependency>
```

> **Note**
>
> SDK 可能不是最新版本。請為應用程式選擇適當的 SDK 版本。

然後，直接在 Java 類別中匯入相依項：

```
import software.amazon.awssdk.services.s3.S3Client;
import software.amazon.awssdk.services.s3.model.PutObjectRequest;
import software.amazon.awssdk.services.s3.model.S3Exception;
```

隨後，範例程式碼會初始化 Amazon S3 用戶端，如下所示：

```
private static final S3Client S3_CLIENT = S3Client.builder().build();
```

在此範例中，我們在主要處理常式函式外部初始化 Amazon S3 用戶端，以免每次調用函式時都必須初始化該用戶端。初始化 SDK 用戶端之後，您即可用其與其他 AWS 服務互動。此範例程式碼會呼叫 Amazon S3 `PutObject` API，如下所示：

```
PutObjectRequest putObjectRequest = PutObjectRequest.builder()
    .bucket(bucketName)
    .key(key)
    .build();

// Convert the receipt content to bytes and upload to S3
S3_CLIENT.putObject(putObjectRequest,
 RequestBody.fromBytes(receiptContent.getBytes(StandardCharsets.UTF_8)));
```

存取環境變數

在處理常式程式碼中，您可以使用 `System.getenv()` 方法來引用任何環境變數。在此範例中，我們使用以下程式碼來參考定義的 `RECEIPT_BUCKET` 環境變數：

```
String bucketName = System.getenv("RECEIPT_BUCKET");
if (bucketName == null || bucketName.isEmpty()) {
```

```
    throw new IllegalArgumentException("RECEIPT_BUCKET environment variable is not
 set");
}
```

使用全域狀態

在首次調用函數之前，Lambda 會在初始化階段執行靜態程式碼和類別建構函數。在初始化期間建立的資源會在調用間隔期間保留在記憶體中，無需您在每次調用函式時都建立這些資源。

在範例程式碼中，S3 用戶端初始化程式碼位於主要處理常式方法外部。執行時期會在函式處理第一個事件之前初始化用戶端，且用戶端在所有調用中均可供重複使用。

Java Lambda 函數的程式碼最佳實務

請遵循下列清單中的準則，在建置 Lambda 函數時使用最佳編碼實務：

- 區隔 Lambda 處理常式與您的核心邏輯。能允許您製作更多可測單位的函式。
- 控制函數部署套件內的相依性。AWS Lambda 執行環境包含多個程式庫。若要啟用最新的一組功能與安全更新，Lambda 會定期更新這些程式庫。這些更新可能會為您的 Lambda 函數行為帶來細微的變更。若要完全掌控您函式所使用的相依性，請利用部署套件封裝您的所有相依性。
- 最小化依存項目的複雜性。偏好更簡易的框架，其可快速在執行環境啟動時載入。例如，偏好更簡易的 Java 相依性置入 (IoC) 架構如 Dagger 或 Guice，勝於複雜的架構如 Spring Framework。
- 將部署套件最小化至執行時期所必要的套件大小。這能減少您的部署套件被下載與呼叫前解壓縮的時間。對於以 Java 撰寫的函數，請避免上傳整個 AWS SDK 程式庫做為部署套件的一部分。或者，選擇性倚賴取得您需要的軟體開發套件元件的模組 (例如 DynamoDB、Amazon S3 開發套件模組，以及 Lambda 核心程式庫)。

請利用執行環境重新使用來改看函式的效能。在函式處理常式之外初始化 SDK 用戶端和資料庫連線，並在本機快取 `/tmp` 目錄中的靜態資產。由您函式的相同執行個體處理的後續叫用可以重複使用這些資源。這可藉由減少函數執行時間來節省成本。

若要避免叫用間洩漏潛在資料，請不要使用執行環境來儲存使用者資料、事件，或其他牽涉安全性的資訊。如果您的函式依賴無法存放在處理常式內記憶體中的可變狀態，請考慮為每個使用者建立個別函式或個別函式版本。

使用 Keep-Alive 指令維持持續連線的狀態。Lambda 會隨著時間的推移清除閒置連線。叫用函數時嘗試重複使用閒置連線將導致連線錯誤。若要維護持續連線，請使用與執行階段相關聯的 keep-alive (保持啟用) 指令。如需範例，請參閱在 Node.js 中重複使用 Keep-Alive 的連線。

使用環境變數將操作參數傳遞給您的函數。例如，如果您正在寫入到 Amazon S3 儲存貯體，而非對您正在寫入的儲存貯體名稱進行硬式編碼，請將儲存貯體名稱設定為環境變數。

避免在 Lambda 函數中使用遞迴調用，其中函數會調用自己或啟動可能再次調用函數的程序。這會導致意外的函式呼叫量與升高的成本。若您看到意外的調用數量，當更新程式碼時，請立刻將函數的預留並行設為 0，以調節對函數的所有調用。

請勿在您的 Lambda 函數程式碼中使用未記錄的非公有 API。對於 AWS Lambda 受管執行時間，Lambda 會定期將安全性和函數更新套用至 Lambda 的內部 API。這些內部 API 更新可能是向後不相容的，這會導致意外結果，例如若您的函數依賴於這些非公有 API，則叫用失敗。請參閱 API 參考查看公開可用 API 的清單。

撰寫等冪程式碼。為函數撰寫等冪程式碼可確保採用相同方式來處理重複事件。程式碼應正確驗證事件並正常處理重複的事件。如需詳細資訊，請參閱 How do I make my Lambda function idempotent? (如何讓 Lambda 函數等冪？)。

- 避免使用 Java DNS 快取。Lambda 函數已快取 DNS 回應。如果您使用其他 DNS 快取，則可能會發生連線逾時。

  `java.util.logging.Logger` 類別可以間接啟用 JVM DNS 快取。若要覆寫預設設定，請先將 networkaddress.cache.ttl 設定為 0，再初始化 `logger`。範例：

  ```
  public class MyHandler {
    // first set TTL property
    static{
     java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
    }
   // then instantiate logger
    var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
  }
  ```

- 將相依性 `.jar` 檔案置於不同的 /lib 目錄，縮短 Lambda 解壓縮以 Java 撰寫之部署套件的時間。這比將您所有函式程式碼全部放入具大量 `.class` 檔案的單一 jar 更快速。如需說明，請參閱 使用 .zip 或 JAR 封存檔部署 Java Lambda 函數。

使用 .zip 或 JAR 封存檔部署 Java Lambda 函數

AWS Lambda 函數的程式碼包含指令碼或編譯的程式及其相依性。使用部署套件將函數程式碼部署到 Lambda。Lambda 支援兩種類型的部署套件：容器映像和 .zip 封存檔。

此頁面說明如何將部署套件建立為 .zip 檔案或 Jar 檔案，然後使用部署套件， AWS Lambda 使用 AWS Command Line Interface (AWS CLI) 將函數程式碼部署至 。

> **Important**
>
> Java 25 引進Ahead-of-Time(AOT) 快取的支援。強烈建議在將函數部署為 .zip 或 JAR 檔案封存時不要使用 AOT 快取，因為當 Lambda 更新受管執行時間時，快取可能會導致非預期的行為。如需詳細資訊，請參閱 Ahead-of-Time(AOT) 和 CDS 快取。

章節

- 先決條件
- 工具與程式庫
- 使用 Gradle 建立部署套件
- 使用層來管理相依項
- 使用 Maven 建立部署套件
- 使用 Lambda 主控台上傳部署套件
- 使用 上傳部署套件 AWS CLI
- 使用 上傳部署套件 AWS SAM

先決條件

AWS CLI 是一種開放原始碼工具，可讓您使用命令列 Shell 中的 命令與 AWS 服務互動。若要完成本節中的步驟，您必須擁有 AWS CLI 版本 2。

工具與程式庫

AWS 為 Java 函數提供下列程式庫。這些程式庫可透過 Maven Central Repository 取得。

- com.amazonaws:aws-lambda-java-core (必要) - 定義處理常式方法介面，以及執行期傳遞給處理常式的內容物件。如果您定義自己的輸入類型，這是您需要的唯一程式庫。

- com.amazonaws:aws-lambda-java-events - 來自調用 Lambda 函數的服務的輸入事件類型。
- com.amazonaws:aws-lambda-java-log4j2 - 一個 Apache Log4j 2 的附加程式庫，您可以使用它將當前調用的請求 ID 新增至函數日誌。
- AWS 適用於 Java 的 SDK 2.0 – 適用於 Java 程式設計語言的官方 AWS SDK。

請將這些程式庫新增至建置定義中，如下所示：

Gradle

```
dependencies {
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
}
```

Maven

```
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-core</artifactId>
      <version>1.2.2</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-events</artifactId>
      <version>3.11.1</version>
    </dependency>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-log4j2</artifactId>
      <version>1.5.1</version>
    </dependency>
  </dependencies>
```

若要建立部署套件，請將函數程式碼和相依性編譯成單一 .zip 檔案或 Java 封存 (JAR) 檔案。針對 Gradle，請使用Zip建置類型。針對 Apache Maven，請使用 Maven Shade 外掛程式。若要上傳部署套件，請使用 Lambda 主控台、Lambda API 或 AWS Serverless Application Model (AWS SAM)。

> **Note**
>
> 若要將您部署套件大小保持較小，請以階層方式將您的函式相依項目進行封裝。各層可讓您獨立管理相依項目，並可供多個函數使用，也可和其他帳戶共用。如需詳細資訊，請參閱Lambda 層。

使用 Gradle 建立部署套件

若要在 Gradle 中建立具有函數程式碼和相依項的部署套件，請使用 `Zip` 建置類型。以下是完整的範本 build.gradle 檔案的範例：

Example build.gradle - 建置任務

```
task buildZip(type: Zip) {
    into('lib') {
        from(jar)
        from(configurations.runtimeClasspath)
    }
}
```

此建置組態會在 `build/distributions` 目錄中產生部署套件。在 `into('lib')` 陳述式中，`jar` 任務會將包含主要類別的 JAR 封存檔組合至名稱為 `lib` 的資料夾中。此外，`configurations.runtimeClassPath` 任務會將相依項程式庫從建置的類別路徑複製到名稱為 `lib` 的資料夾中。

Example build.gradle - 相依性

```
dependencies {
    ...
    implementation 'com.amazonaws:aws-lambda-java-core:1.2.2'
    implementation 'com.amazonaws:aws-lambda-java-events:3.11.1'
    implementation 'org.apache.logging.log4j:log4j-api:2.17.1'
    implementation 'org.apache.logging.log4j:log4j-core:2.17.1'
    runtimeOnly 'org.apache.logging.log4j:log4j-slf4j18-impl:2.17.1'
    runtimeOnly 'com.amazonaws:aws-lambda-java-log4j2:1.5.1'
    ...
}
```

Lambda 會以 Unicode 字母順序載入 JAR 檔案。如果 `lib` 目錄的多個 JAR 檔案包含相同類別，則會使用第一個 JAR。您可以使用以下 shell 指令碼來識別重複的類別：

Example test-zip.sh

```
mkdir -p expanded
unzip path/to/my/function.zip -d expanded
find ./expanded/lib -name '*.jar' | xargs -n1 zipinfo -1 | grep '.*.class' | sort |
 uniq -c | sort
```

使用層來管理相依項

您可以將函式的相依項封裝為層，使部署套件保持較小體積，獨立管理相依項。如需詳細資訊，請參閱the section called “層”。

使用 Maven 建立部署套件

要使用 Maven 建置部署套件，請使用 Maven Shade 外掛程式。該外掛程式會建立一個 JAR 檔案，其中包含編譯的函數代碼及其所有相依性。

Example pom.xml - 外掛程式組態

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
```

若要建置部署套件，請使用 `mvn package` 命令。

```
[INFO] Scanning for projects...
[INFO] ----------------------< com.example:java-maven >----------------------
[INFO] Building java-maven-function 1.0-SNAPSHOT
[INFO] --------------------------------[ jar ]---------------------------------
...
[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ java-maven ---
[INFO] Building jar: target/java-maven-1.0-SNAPSHOT.jar
[INFO]
[INFO] --- maven-shade-plugin:3.2.2:shade (default) @ java-maven ---
[INFO] Including com.amazonaws:aws-lambda-java-core:jar:1.2.2 in the shaded jar.
[INFO] Including com.amazonaws:aws-lambda-java-events:jar:3.11.1 in the shaded jar.
[INFO] Including joda-time:joda-time:jar:2.6 in the shaded jar.
[INFO] Including com.google.code.gson:gson:jar:2.8.6 in the shaded jar.
[INFO] Replacing original artifact with shaded artifact.
[INFO] Replacing target/java-maven-1.0-SNAPSHOT.jar with target/java-maven-1.0-
SNAPSHOT-shaded.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  8.321 s
[INFO] Finished at: 2020-03-03T09:07:19Z
[INFO] ------------------------------------------------------------------------
```

此命令會在 `target` 目錄中產生一個 JAR 檔案。

> **Note**
>
> 如果您正在使用多版本 JAR (MRJAR)，則必須在 `lib` 目錄中包含 MRJAR (即由 Maven Shade 外掛程式產生的陰影 JAR)，並在將部署套件上傳到 Lambda 之前對其進行壓縮。否則，Lambda 可能無法正確解壓縮 JAR 檔案，導致您的 `MANIFEST.MF` 檔案被忽略。

如果您使用附加器程式庫 (`aws-lambda-java-log4j2`)，則還必須配置 Maven Shade 外掛程式的轉換器。轉換器程式庫會合併出現在附加器程式庫和 Log4j 中快取檔案的版本。

Example pom.xml - 具備 Log4j 2 附加器的外掛程式組態

```
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.2</version>
```

```
        <configuration>
          <createDependencyReducedPom>false</createDependencyReducedPom>
        </configuration>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <transformers>
                <transformer
implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCacheFile
                </transformer>
              </transformers>
            </configuration>
          </execution>
        </executions>
        <dependencies>
          <dependency>
            <groupId>com.github.edwgiz</groupId>
            <artifactId>maven-shade-plugin.log4j2-cachefile-transformer</artifactId>
            <version>2.13.0</version>
          </dependency>
        </dependencies>
      </plugin>
```

使用 Lambda 主控台上傳部署套件

若要建立新函數，您必須先在主控台中建立函數，然後上傳您的 .zip 或 JAR 檔案。若要更新現有函數，請開啟函數的頁面，然後按照同樣的程序新增更新後的 .zip 或 JAR 檔案。

如果您的部署套件檔案小於 50 MB，您可以透過直接從本機電腦上傳檔案來建立或更新函數。若 .zip 或 JAR 檔案大於 50 MB，您必須先將套件上傳至 Amazon S3 儲存貯體。如需如何使用 將檔案上傳至 Amazon S3 儲存貯體的說明 AWS 管理主控台，請參閱 Amazon S3 入門。若要使用 上傳檔案 AWS CLI，請參閱AWS CLI 《 使用者指南》中的移動物件。

> **Note**
>
> 不能變更現有函數的部署套件類型 (.zip 或容器映像)。例如，您不能轉換容器映像函數以使用 .zip 封存檔。您必須建立新的函數。

若要建立新的函數 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇建立函數。
2. 選擇 Author from scratch (從頭開始撰寫)。
3. 在基本資訊下，請執行下列動作：
   a. 在函數名稱中輸入函數名稱。
   b. 在執行期中選取要使用的執行期。
   c. (選用) 在架構中選擇要用於函數的指令集架構。預設架構值為 x86_64。請確定函數的 .zip 部署套件與您選取的指令集架構相容。
4. (選用) 在許可下，展開變更預設執行角色。您可建立新的執行角色，或使用現有的角色。
5. 選擇 Create function (建立函數)。Lambda 會使用您選擇的執行期建立一個基本的「Hello world」函數。

若要從本機電腦上傳 .zip 或 JAR 封存檔 (主控台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳 .zip 或 JAR 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 .zip 或 .jar 檔案。
5. 若要上傳 .zip 或 JAR 檔案，請執行下列操作：
   a. 選取上傳，然後在檔案選擇器中選取您的 .zip 或 JAR 檔案。
   b. 選擇 Open (開啟)。
   c. 選擇儲存。

若要從 Amazon S3 儲存貯體上傳 .zip 或 JAR 封存檔 (控制台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳新 .zip 或 JAR 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 Amazon S3 位置。
5. 貼上 .zip 檔案的 Amazon S3 連結 URL，然後選擇儲存。

使用 上傳部署套件 AWS CLI

您可以使用 AWS CLI 建立新函數，或使用 .zip 或 JAR 檔案更新現有函數。使用 create-function 和 update-function-code 命令來部署您的 .zip 或 JAR 套件。如果您的檔案小於 50 MB，則可以從本機建置電腦的檔案位置上傳套件。若檔案較大，則必須先從 Amazon S3 儲存貯體上傳 .zip 或 JAR 套件。如需如何使用 將檔案上傳至 Amazon S3 儲存貯體的指示 AWS CLI，請參閱AWS CLI《 使用者指南》中的移動物件。

> **Note**
>
> 如果您使用 從 Amazon S3 儲存貯體上傳 .zip 或 JAR 檔案 AWS CLI，則儲存貯體必須與 AWS 區域 函數位於相同的 中。

若要使用 .zip 或 JAR 檔案搭配 建立新的函數 AWS CLI，您必須指定下列項目：

- 函數名稱 (`--function-name`)
- 函數的執行期 (`--runtime`)
- 函數執行角色的 Amazon Resource Name (ARN) (`--role`)
- 函數程式碼中處理常式方法的名稱 (`--handler`)

您也必須指定 .zip 或 JAR 檔案的位置。如果您的 .zip 或 JAR 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime java25 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--code` 選項。您只需針對版本控制的物件使用 `S3ObjectVersion` 參數。

```
aws lambda create-function --function-name myFunction \
--runtime java25 --handler example.handler \
--role arn:aws:iam::123456789012:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

若要使用 CLI 更新現有函數，您可以使用 `--function-name` 參數指定函數的名稱。您也必須指定要用來更新函數程式碼的 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 `--zip-file` 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 `--s3-bucket` 和 `--s3-key` 選項。您只需針對版本控制的物件使用 `--s3-object-version` 參數。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

使用 上傳部署套件 AWS SAM

您可以使用 AWS SAM 自動部署函數程式碼、組態和相依性。 AWS SAM 是 的延伸 CloudFormation ，提供定義無伺服器應用程式的簡化語法。下列範例範本會透過在 Gradle 使用的 `build/distributions` 目錄中的部署套件定義函數：

Example template.yml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: An AWS Lambda application that calls the Lambda API.
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/java-basic.zip
      Handler: example.Handler
      Runtime: java25
      Description: Java function
      MemorySize: 512
      Timeout: 10
      # Function's execution role
      Policies:
        - AWSLambdaBasicExecutionRole
        - AWSLambda_ReadOnlyAccess
        - AWSXrayWriteOnlyAccess
        - AWSLambdaVPCAccessExecutionRole
```

```
      Tracing: Active
```

若要建立函數，請使用 `package` 和 `deploy` 指令。這些命令是對 AWS CLI的自訂命令。它們會包裝其他命令，將部署套件上傳至 Amazon S3，使用物件 URI 重寫範本，並更新函數的程式碼。

下面的範例指令碼會執行 Gradle 建置並上傳到它建立的部署套件。它會在您第一次執行 CloudFormation 堆疊時建立堆疊。如果堆疊已存在，則指令碼會加以更新。

Example deploy.sh

```
#!/bin/bash
set -eo pipefail
aws cloudformation package --template-file template.yml --s3-bucket MY_BUCKET --output-
template-file out.yml
aws cloudformation deploy --template-file out.yml --stack-name java-basic --
capabilities CAPABILITY_NAMED_IAM
```

如需完整的工作範例，請參閱下列範例應用程式：

以 Java 編寫的範例 Lambda 應用程式

- example-java – Java 函式，示範如何使用 Lambda 處理訂單。此函數說明如何定義和還原序列化自訂輸入事件物件、使用 AWS SDK 和輸出記錄。
- java-basic - 具有單元測試和變數日誌組態的最小 Java 函數集合。
- java-events - Java 函數集合，其中包含如何處理來自各種服務 (例如 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis) 事件的骨架程式碼。這些函數使用最新版 aws-lambda-java-events 程式庫 (3.0.0 及更新版)。這些範例不需要 AWS SDK 做為相依性。
- s3-java - 一種 Java 函數，它處理來自 Amazon S3 的通知事件，並使用 Java Class Library (JCL) 以從上傳的映像檔案建立縮圖。
- layer-java – 此 Java 函式示範如何使用 Lambda 層，將相依項與核心函式程式碼分離封裝。

使用容器映像部署 Java Lambda 函數

您可以透過三種方式為 Java Lambda 函數建置容器映像：

- 使用 Java AWS 的基礎映像

  AWS 基礎映像會預先載入語言執行期、用來管理 Lambda 與函數程式碼之間互動的執行期界面用戶端，以及用於本機測試的執行期界面模擬器。

- 使用僅限 AWS 作業系統的基礎映像

  AWS 僅限作業系統的基礎映像包含 Amazon Linux 發行版本和執行時間界面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Java 的執行期介面用戶端。

- 使用非AWS 基礎映像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像與 Lambda 相容，您必須在映像中加入適用於 Java 的執行期介面用戶端。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

本頁面會說明如何為 Lambda 建置、測試和部署容器映像。

主題

- AWS Java 的基礎映像
- 使用 Java AWS 的基礎映像
- 透過執行期介面用戶端使用替代基礎映像

AWS Java 的基礎映像

AWS 為 Java 提供下列基礎映像：

Tags (標籤)	執行期	作業系統	Dockerfile	棄用
25	Java 25	Amazon Linux 2023	GitHub 上適用於 Java 25 的 Dockerfile	2029 年 6 月 30 日
21	Java 21	Amazon Linux 2023	Dockerfile for Java 21 on GitHub	2029 年 6 月 30 日
17	Java 17	Amazon Linux 2	Dockerfile for Java 17 on GitHub	2027 年 6 月 30 日
11	Java 11	Amazon Linux 2	Dockerfile for Java 11 on GitHub	2027 年 6 月 30 日
8.al2	Java 8	Amazon Linux 2	Dockerfile for Java 8 on GitHub	2027 年 6 月 30 日

Amazon ECR 儲存庫：gallery.ecr.aws/lambda/java

Java 21 和更新版本的基礎映像以 Amazon Linux 2023 最小容器映像為基礎。舊版基礎映像使用 Amazon Linux 2。與 Amazon Linux 2 相比，AL2023 具有多項優點，包括更小的部署足跡和更新版本的程式庫，如 `glibc`。

以 AL2023 為基礎的映像使用 `microdnf` (符號連結為 `dnf`) 而不是 `yum` 作為套件管理工具，後者是 Amazon Linux 2 中的預設套件管理工具。`microdnf` 是 `dnf` 的獨立實作。對於以 AL2023 為基礎的映像中包含的套件清單，請參閱 Comparing packages installed on Amazon Linux 2023 Container Images 中的 Minimal Container 欄。如需 AL2023 和 Amazon Linux 2 之間差異的詳細資訊，請參閱 AWS 運算部落格上的 Introducing the Amazon Linux 2023 runtime for AWS Lambda。

> **Note**
>
> 若要在本機執行AL2023-based映像，包括搭配 AWS Serverless Application Model (AWS SAM)，您必須使用 Docker 20.10.10 版或更新版本。

使用 Java AWS 的基礎映像

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- Java (例如，Amazon Corretto)
- Apache Maven 或 Gradle
- AWS CLI 第 2 版
- Docker (最低版本 25.0.0)
- Docker buildx 外掛程式。

從基礎映像建立映像

Maven

1. 執行下列命令，使用 Lambda 的原型建立 Maven 專案。下列是必要參數：

   - 服務 – 要在 Lambda 函數中使用的 AWS 服務 用戶端。如需可用來源的清單，請參閱 GitHub 上的 aws-sdk-java-v2/services。
   - region – 您要建立 Lambda 函數的 AWS 區域 。
   - groupId – 應用程式的完整套件命名空間。
   - artifactId – 您的專案名稱。這會成為您專案的目錄名稱。

   在 Linux 和 macOS 中，執行此命令：

   ```
   mvn -B archetype:generate \
      -DarchetypeGroupId=software.amazon.awssdk \
      -DarchetypeArtifactId=archetype-lambda -Dservice=s3 -Dregion=US_WEST_2 \
      -DgroupId=com.example.myapp \
      -DartifactId=myapp
   ```

   在 PowerShell 中，執行此命令：

   ```
   mvn -B archetype:generate `
      "-DarchetypeGroupId=software.amazon.awssdk" `
      "-DarchetypeArtifactId=archetype-lambda" "-Dservice=s3" "-Dregion=US_WEST_2"
    `
   ```

```
    "-DgroupId=com.example.myapp" `
    "-DartifactId=myapp"
```

Lambda 的 Maven 原型已預先設定為使用 Java SE 8 編譯，並包含對 適用於 Java 的 AWS SDK的相依性。如果您使用不同的原型或使用其他方法來建立專案，則必須為 Maven 設定 Java 編譯器並將 SDK 宣告為相依項。

2. 開啟 *myapp*/src/main/java/com/example/*myapp* 目錄並找到 App.java 檔案。這是 Lambda 函數的程式碼。可以使用提供的範本程式碼進行測試，也可以將其替換為您自己的程式碼。
3. 返回專案的根目錄，然後使用以下組態建立新的 Dockerfile：

   - 將 FROM 屬性設定為基礎映像的 URI。
   - 將 CMD 引數設定為 Lambda 函數處理常式。

   請注意，範例 Dockerfile 不包含 USER 指令。當您將容器映像部署到 Lambda 時，Lambda 會自動定義一個具有最低權限許可的預設 Linux 使用者。這與標準 Docker 行為不同，後者會在未提供 USER 指令時預設為 root 使用者。

   Example Dockerfile

   ```
   FROM public.ecr.aws/lambda/java:21

   # Copy function code and runtime dependencies from Maven layout
   COPY target/classes ${LAMBDA_TASK_ROOT}
   COPY target/dependency/* ${LAMBDA_TASK_ROOT}/lib/

   # Set the CMD to your handler (could also be done as a parameter override
    outside of the Dockerfile)
   CMD [ "com.example.myapp.App::handleRequest" ]
   ```

4. 編譯專案並收集執行期相依性。

   ```
   mvn compile dependency:copy-dependencies -DincludeScope=runtime
   ```

5. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。若要使映像與 Lambda 相容，必須使用 --provenance=false 選項。

   ```
   docker buildx build --platform linux/amd64 --provenance=false -t docker-
   image:test .
   ```

> **Note**
>
> 此命令會指定 `--platform linux/amd64` 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 `--platform linux/arm64` 選項。

Gradle

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir example
cd example
```

2. 執行以下命令，讓 Gradle 在環境的 `example` 目錄中產生新的 Java 應用程式專案。對於選取建置指令碼 DSL，選擇 2：Groovy。

```
gradle init --type java-application
```

3. 開啟 */example*/app/src/main/java/*example* 目錄並找到 `App.java` 檔案。這是 Lambda 函數的程式碼。可以使用以下範本程式碼進行測試，也可以將其替換為您自己的程式碼。

   Example App.java

```
package com.example;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
public class App implements RequestHandler<Object, String> {
    public String handleRequest(Object input, Context context) {
        return "Hello world!";
    }
}
```

4. 開啟 `build.gradle` 檔案。如果使用上一個步驟的範本函數程式碼，請將 `build.gradle` 的內容替換為下列值。如果使用自己的函數程式碼，則請根據需要修改 `build.gradle` 檔案。

Example build.gradle (Groovy DSL)

```
plugins {
  id 'java'
}
group 'com.example'
version '1.0-SNAPSHOT'
sourceCompatibility = 1.8
repositories {
  mavenCentral()
}
dependencies {
  implementation 'com.amazonaws:aws-lambda-java-core:1.2.1'
}
jar {
  manifest {
      attributes 'Main-Class': 'com.example.App'
  }
}
```

5. 步驟 2 中的 `gradle init` 命令也在 `app/test` 目錄中產生了一個虛擬測試案例。為實現本教學課程的目的，請刪除 `/test` 目錄，以略過正在執行的測試。
6. 建置專案。

```
gradle build
```

7. 在專案的根目錄 (`/example`) 中，使用以下組態建立一個 Dockerfile：

   - 將 FROM 屬性設定為基礎映像的 URI。
   - 使用 COPY 命令將函數程式碼和執行時期相依項複製到 {LAMBDA_TASK_ROOT}，一個 Lambda 定義的環境變數。
   - 將 CMD 引數設定為 Lambda 函數處理常式。

   請注意，範例 Dockerfile 不包含 USER 指令。當您將容器映像部署到 Lambda 時，Lambda 會自動定義一個具有最低權限許可的預設 Linux 使用者。這與標準 Docker 行為不同，後者會在未提供 USER 指令時預設為 `root` 使用者。

Example Dockerfile

```
FROM public.ecr.aws/lambda/java:21

# Copy function code and runtime dependencies from Gradle layout
COPY app/build/classes/java/main ${LAMBDA_TASK_ROOT}

# Set the CMD to your handler (could also be done as a parameter override
 outside of the Dockerfile)
CMD [ "com.example.App::handleRequest" ]
```

8. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。若要使映像與 Lambda 相容，必須使用 --provenance=false 選項。

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-
image:test .
```

> **Note**
>
> 此命令會指定 --platform linux/amd64 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 --platform linux/arm64 選項。

(選用) 在本機測試映像

1. 使用 docker run 命令啟動 Docker 影像。在此範例中，docker-image 為映像名稱，test 為標籤。

```
docker run --platform linux/amd64 -p 9000:8080 docker-image:test
```

此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將--platform linux/*arm64*選項改用選項。--platform linux/*amd64*

2. 從新的終端機視窗，將事件張貼至本機端點。

   Linux/macOS

   在 Linux 或 macOS 中，執行下列 `curl` 命令：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   在 PowerShell 中，執行下列 `Invoke-WebRequest` 命令：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

3. 取得容器 ID。

   ```
   docker ps
   ```

4. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

   ```
   docker kill 3766c4ab331c
   ```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將 `--region`值設定為您要 AWS 區域 在其中建立 Amazon ECR 儲存庫的 。
   - `111122223333` 以您的 AWS 帳戶 ID 取代 。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於相同的 AWS 區域 中。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
```

```
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - `docker-image:test` 為 Docker 映像檔的名稱和標籤。這是您在 `docker build` 命令中指定的映像名稱和標籤。
   - 將 `<ECRrepositoryUri>` 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像來建立函數。如需詳細資訊，請參閱Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 會將映像標籤解析為特定映像摘要。這表示如果您將用來部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。

若要將新映像部署至相同的 Lambda 函數，必須使用 update-function-code 命令，即使 Amazon ECR 中的映像標籤保持不變亦如此。在以下範例中，`--publish` 選項會使用更新的容器映像來建立新的函數版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

透過執行期介面用戶端使用替代基礎映像

如果您使用僅限作業系統的基礎映像或替代的基礎映像，則必須在映像中加入執行期介面用戶端。執行期介面用戶端會讓您擴充 Runtime API，管理 Lambda 與函數程式碼之間的互動。

在 Dockerfile 中安裝適用於 Java 的執行期介面用戶端，或作為專案中的相依項。例如，若要使用 Maven 套件管理員安裝執行期界面用戶端，請將以下內容新增至您的 pom.xml 檔案中：

```
<dependency>
    <groupId>com.amazonaws</groupId>
    <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
    <version>2.3.2</version>
</dependency>
```

如需套件詳細資訊，請參閱 Maven Central Repository 中的 AWS Lambda Java 執行期介面用戶端。您還可以在 AWS Lambda Java 支援程式庫 GitHub 儲存庫中檢閱執行期介面用戶端原始程式碼。

以下範例示範如何使用 Amazon Corretto 映像為 Java 建置容器映像。Amazon Corretto 是 Open Java Development Kit (OpenJDK) 的免費、多平台的生產就緒分佈。Maven 專案包括執行期介面用戶端作為相依項。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- Java (例如，Amazon Corretto)
- Apache Maven
- AWS CLI 第 2 版
- Docker (最低版本 25.0.0)
- Docker buildx 外掛程式。

使用替代基礎映像建立映像

1. 建立 Maven 專案。下列是必要參數：

   - groupId – 應用程式的完整套件命名空間。
   - artifactId – 您的專案名稱。這會成為您專案的目錄名稱。

   Linux/macOS

   ```
   mvn -B archetype:generate \
      -DarchetypeArtifactId=maven-archetype-quickstart \
      -DgroupId=example \
   ```

```
    -DartifactId=myapp \
    -DinteractiveMode=false
```

PowerShell

```
mvn -B archetype:generate `
    -DarchetypeArtifactId=maven-archetype-quickstart `
    -DgroupId=example `
    -DartifactId=myapp `
    -DinteractiveMode=false
```

2. 開啟專案目錄。

```
cd myapp
```

3. 開啟 pom.xml 檔案並將內容替換如下：此檔案包括 aws-lambda-java-runtime-interface-client 作為相依項。或者，您可以在 Dockerfile 中安裝執行期界面用戶端。不過，最簡單的方法是將程式庫包含為相依項。

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>example</groupId>
  <artifactId>hello-lambda</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>hello-lambda</name>
  <url>http://maven.apache.org</url>
  <properties>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>com.amazonaws</groupId>
      <artifactId>aws-lambda-java-runtime-interface-client</artifactId>
      <version>2.3.2</version>
    </dependency>
  </dependencies>
  <build>
```

```
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-dependency-plugin</artifactId>
        <version>3.1.2</version>
        <executions>
          <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
              <goal>copy-dependencies</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

4. 開啟 *myapp*/src/main/java/com/example/*myapp* 目錄並找到 App.java 檔案。這是 Lambda 函數的程式碼。將程式碼取代為以下內容。

   Example函數處理常式

   ```
   package example;

   public class App {
       public static String sayHello() {
           return "Hello world!";
       }
   }
   ```

5. 步驟 1 中的 mvn -B archetype:generate 命令也在 src/test 目錄中產生了一個虛擬測試案例。為實現本教學課程的目的，請將整個產生的 /test 目錄刪除，以略過執行測試程序。

6. 返回專案的根目錄，然後建立新的 Dockerfile。下列 Dockerfile 範例使用 Amazon Corretto 映像。Amazon Corretto 是 OpenJDK 的免費、多平台的生產就緒分佈。

   - 將 FROM 屬性設定為基礎映像的 URI。
   - 將 ENTRYPOINT 設為您希望 Docker 容器在啟動時執行的模組。在此案例中，模組是執行期界面用戶端。
   - 將 CMD 引數設定為 Lambda 函數處理常式。

請注意，範例 Dockerfile 不包含 USER 指令。當您將容器映像部署到 Lambda 時，Lambda 會自動定義一個具有最低權限許可的預設 Linux 使用者。這與標準 Docker 行為不同，後者會在未提供 USER 指令時預設為 root 使用者。

Example Dockerfile

```
FROM public.ecr.aws/amazoncorretto/amazoncorretto:21 as base

# Configure the build environment
FROM base as build
RUN yum install -y maven
WORKDIR /src

# Cache and copy dependencies
ADD pom.xml .
RUN mvn dependency:go-offline dependency:copy-dependencies

# Compile the function
ADD . .
RUN mvn package

# Copy the function artifact and dependencies onto a clean base
FROM base
WORKDIR /function

COPY --from=build /src/target/dependency/*.jar ./
COPY --from=build /src/target/*.jar ./

# Set runtime interface client as default command for the container runtime
ENTRYPOINT [ "/usr/bin/java", "-cp", "./*",
 "com.amazonaws.services.lambda.runtime.api.client.AWSLambda" ]
# Pass the name of the function handler as an argument to the runtime
CMD [ "example.App::sayHello" ]
```

7. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 docker-image 並為其提供 test 標籤。若要使映像與 Lambda 相容，必須使用 --provenance=false 選項。

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> 此命令會指定 `--platform linux/amd64` 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 `--platform linux/arm64` 選項。

(選用) 在本機測試映像

使用 執行期界面模擬器 以在本機測試映像。您可以將模擬器建置到映像中，也可以使用以下步驟，將其安裝在本機電腦。

若要在本機電腦上安裝並執行執行期介面模擬器

1. 在您的專案目錄中執行以下命令，從 GitHub 下載執行期介面模擬器 (x86-64 架構)，並安裝在本機電腦上。

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   若要安裝 arm64 模擬器，請將上一個命令中的 GitHub 儲存庫 URL 替換為以下內容：

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   ```

```
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

若要安裝 arm64 模擬器，請將 $downloadLink 更換為下列項目：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. 使用 docker run 命令啟動 Docker 影像。注意下列事項：

   - docker-image 是映像名稱，而 test 是標籤。
   - /usr/bin/java -cp './*' com.amazonaws.services.lambda.runtime.api.client.AWSLambda example.App::sayHello 是 Dockerfile 中的 ENTRYPOINT，後面接著 CMD。

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           /usr/bin/java -cp './*'
    com.amazonaws.services.lambda.runtime.api.client.AWSLambda
    example.App::sayHello
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
       /usr/bin/java -cp './*'
    com.amazonaws.services.lambda.runtime.api.client.AWSLambda
    example.App::sayHello
   ```

   此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

> **Note**
>
> 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將`--platform linux/arm64`選項改用選項。`--platform linux/amd64`

3. 將事件張貼至本機端點。

   Linux/macOS

   在 Linux 或 macOS 中，執行下列 `curl` 命令：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

   PowerShell

   在 PowerShell 中，執行下列 `Invoke-WebRequest` 命令：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{}' -ContentType "application/json"
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

   ```
   Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
   invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
    "application/json"
   ```

4. 取得容器 ID。

   ```
   docker ps
   ```

5. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將 `--region`值設定為您要 AWS 區域 在其中建立 Amazon ECR 儲存庫的 。
   - `111122223333` 以您的 AWS 帳戶 ID 取代 。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數位於相同的 AWS 區域 中。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
```

```
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - `docker-image:test` 為 Docker 映像檔的名稱和標籤。這是您在 `docker build` 命令中指定的映像名稱和標籤。
   - 將 `<ECRrepositoryUri>` 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 只要映像與 Lambda 函數位於相同的區域，您就可以使用不同 AWS 帳戶中的映像來建立函數。如需詳細資訊，請參閱Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 會將映像標籤解析為特定映像摘要。這表示如果您將用來部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。

若要將新映像部署至相同的 Lambda 函數，必須使用 update-function-code 命令，即使 Amazon ECR 中的映像標籤保持不變亦如此。在以下範例中，`--publish` 選項會使用更新的容器映像來建立新的函數版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

使用 Java Lambda 函數的層

使用 Lambda 層封裝要在多個函式之間重複使用的程式碼與相依項。層通常具備程式庫相依性、自訂執行期或組態檔案。建立層包含三個一般步驟：

1. 封裝層內容。這表示建立 .zip 封存檔，其中包含您要在函數中使用的相依項。
2. 在 Lambda 中建立層。
3. 將層新增至函數中。

主題

- 封裝層內容
- 在 Lambda 中建立層
- 將層新增至函式

封裝層內容

若要建立層，請將套件封裝成符合以下要求的 .zip 壓縮檔：

- 確保 Maven 或 Gradle 引用的 Java 版本，與您打算部署的函式所使用的 Java 版本一致。例如，對於 Java 25 函數，`mvn -v`命令應在輸出中列出 Java 25。
- 相依項必須儲存在 .zip 檔案根目錄下的 `java/lib` 目錄中。如需詳細資訊，請參閱每個 Lambda 執行時間的層路徑。
- 層中的套件必須與 Linux 相容。Lambda 函式會在 Amazon Linux 上執行。

您可以建立包含第三方 Java 程式庫的層，或您自訂的 Java 模組與套件的層。以下流程使用 Maven 實作。您也可以選用 Gradle 來封裝層內容。

使用 Maven 相依項建立層

1. 建立一個包含 `pom.xml` 檔案的 Apache Maven 專案，並在該檔案中定義相依項目。

   下列範例包含用於 JSON 處理的 Jackson Databind。<build> 區段透過 maven-dependency-plugin 為每個相依項建立獨立的 JAR 檔案，而非將其封裝成單一 uber-jar。若需建立 uber-jar，請改用 maven-shade-plugin。

Example pom.xml

```
<dependencies>
    <dependency>
        <groupId>com.fasterxml.jackson.core</groupId>
        <artifactId>jackson-databind</artifactId>
        <version>2.17.0</version>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.13.0</version>
            <configuration>
                <source>21</source>
                <target>21</target>
                <release>21</release>
            </configuration>
        </plugin>

        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-dependency-plugin</artifactId>
            <version>3.6.1</version>
            <executions>
                <execution>
                    <id>copy-dependencies</id>
                    <phase>package</phase>
                    <goals>
                        <goal>copy-dependencies</goal>
                    </goals>
                    <configuration>
                        <outputDirectory>${project.build.directory}/lib</
outputDirectory>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
```

```
</build>
```

2. 建置專案。此命令會在 `target/lib/` 目錄中建立所有相依項 JAR 檔案。

```
mvn clean package
```

3. 建立層所需的目錄結構：

```
mkdir -p java/lib
```

4. 將相依項 JAR 檔案複製至 `java/lib` 目錄：

```
cp target/lib/*.jar java/lib/
```

5. 壓縮層內容：

   Linux/macOS

   ```
   zip -r layer.zip java/
   ```

   PowerShell

   ```
   Compress-Archive -Path .\java -DestinationPath .\layer.zip
   ```

   .zip 檔案的目錄結構應如下所示：

   ```
   java/
   ### lib/
       ### jackson-databind-2.17.0.jar
       ### jackson-core-2.17.0.jar
       ### jackson-annotations-2.17.0.jar
   ```

   > **Note**
   >
   > 請確保 .zip 檔案在根層級包含 `java` 目錄，且該目錄內含 `lib` 子目錄。此結構可確保 Lambda 能順利找到並匯入程式庫。每個相依項都會作為獨立的 JAR 檔案保留，而非封裝成單一 uber-jar。

在 Lambda 中建立層

您可以使用 AWS CLI 或 Lambda 主控台發佈 layer。

AWS CLI

執行 publish-layer-version AWS CLI 命令來建立 Lambda layer：

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip
 --compatible-runtimes java25
```

相容的執行時期參數為選用參數。指定此參數後，Lambda 會據此在 Lambda 主控台中篩選層。

Console

建立圖層 (主控台)

1. 開啟 Lambda 主控台中的 層頁面。
2. 選擇 建立圖層。
3. 選擇上傳 .zip 檔案，然後上傳先前建立的 .zip 壓縮檔。
4. (選用) 在相容的執行時期欄位中，選擇與用於建置層的 Java 版本相對應的 Java 執行時期。
5. 選擇建立。

將層新增至函式

AWS CLI

若要將 layer 連接至函數，請執行 update-function-configuration AWS CLI 命令。對於 `--layers` 參數，請使用層 ARN。ARN 必須指定版本 (例如 `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`)。如需詳細資訊，請參閱層和層的版本。

```
aws lambda update-function-configuration --function-name my-function --cli-binary-
format raw-in-base64-out --layers "arn:aws:lambda:us-east-1:123456789012:layer:my-
layer:1"
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

Console

將層新增至函式

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函式。
3. 向下捲動至層區段，然後選擇新增層。
4. 在選擇層欄位中，選取自訂層，然後選擇要使用的層。

> **Note**
>
> 如果未在建立層時新增相容的執行時期，此處不會列出層。您可以改為指定層 ARN。

5. 選擇新增。

自訂 Lambda Java 函數的序列化

Lambda Java 受管執行時期支援 JSON 事件的自訂序列化。自訂序列化可以簡化程式碼，並可能提高效能。

主題

- 何時使用自訂序列化
- 實作自訂序列化
- 測試自訂序列化

何時使用自訂序列化

調用 Lambda 函數時，輸入事件資料需要還原序列化為 Java 物件，函數的輸出需要序列化回可作為函數回應傳回的格式。Lambda Java 受管執行時期提供預設的序列化和還原序列化功能，非常適合處理來自 Amazon API Gateway、Amazon Simple Queue Service (Amazon SQS) 等各種 AWS 服務的事件承載。若要在函數中使用這些服務整合事件，請將 aws-java-lambda-events 相依項新增至您的專案。此 AWS 程式庫包含代表這些服務整合事件的 Java 物件。

您也可以使用自己的物件來代表傳遞至 Lambda 函數的事件 JSON。受管執行時期會嘗試以預設行為將 JSON 序列化至物件的新執行個體。如果預設序列化程式沒有您的使用案例所需的行為，則請使用自訂序列化。

例如，假設您的函數處理程式需要 `Vehicle` 類別作為輸入，且其結構如下：

```
public class Vehicle {
    private String vehicleType;
    private long vehicleId;
}
```

不過，JSON 事件承載如下所示：

```
{
    "vehicle-type": "car",
    "vehicleID": 123
}
```

在這種情況下，受管執行時期的預設序列化期望 JSON 屬性名稱與駝峰式 Java 類別屬性名稱 (vehicleType、vehicleId) 相符。由於 JSON 事件中的屬性名稱不是駝峰式 (vehicle-type、vehicleID)，您必須使用自訂序列化。

實作自訂序列化

使用服務提供者介面載入您選擇的序列化程式，而不是受管執行時期的預設序列化邏輯。您可以使用標準 RequestHandler 介面，將 JSON 事件承載直接序列化至 Java 物件。

若要在 Lambda Java 函數中使用自訂序列化

1. 新增 aws-lambda-java-core 程式庫做為相依項。此程式庫包含 CustomPojoSerializer 介面，以及在 Lambda 中使用 Java 的其他介面定義。
2. 在專案的 src/main/resources/META-INF/services/ 目錄中建立名為 com.amazonaws.services.lambda.runtime.CustomPojoSerializer 的檔案。
3. 在此檔案中，指定自訂序列化工具實作的完全合格名稱，其必須實作 CustomPojoSerializer 介面。範例：

```
com.mycompany.vehicles.CustomLambdaSerialzer
```

4. 實作 CustomPojoSerializer 介面以提供您的自訂序列化邏輯。
5. 在 Lambda 函數中使用標準 RequestHandler 介面。受管執行時期將使用自訂序列化程式。

如需如何使用常用程式庫 (例如 fastJson、Gson、Moshi 和 jackson-jr) 實作自訂序列化的更多範例，請參閱 AWS GitHub 儲存庫中的 custom-serialization 範例。

測試自訂序列化

測試您的函數以確定序列化和還原序列化邏輯如預期般運作。您可以使用 AWS Serverless Application Model 命令列介面 (AWS SAM CLI) 模擬 Lambda 承載的調用。這可協助您在引進自訂序列化程式時，快速測試並重複執行函數。

1. 使用您想要用來調用函數的 JSON 事件承載建立一個檔案，然後呼叫 AWS SAM CLI。
2. 執行 sam local invoke 命令以在本機調用您的函數。範例：

```
sam local invoke -e src/test/resources/event.json
```

如需詳細資訊，請參閱 Locally invoke Lambda functions with AWS SAM。

自訂 Lambda 函數的 Java 執行時期啟動行為

此頁面說明 中 Java 函數的特定設定 AWS Lambda。您可以使用這些設定來自訂 Java 執行期啟動行為。這可以減少整體函數延遲並提高整體函數效能，而無需修改任何程式碼。

章節

- 了解 JAVA_TOOL_OPTIONS 環境變數
- Log4Shell 的 Log4j 修補程式 Log4Shell
- Ahead-of-Time(AOT) 和 CDS 快取

了解 JAVA_TOOL_OPTIONS 環境變數

在 Java 中，Lambda 支援 `JAVA_TOOL_OPTIONS` 環境變數，以在 Lambda 中設定其他命令列變數。您可以透過各種方式使用此環境變數，例如自訂分層編譯設定。下一個範例將示範如何針對此使用案例使用 `JAVA_TOOL_OPTIONS` 環境變數。

範例：自訂分層編譯設定

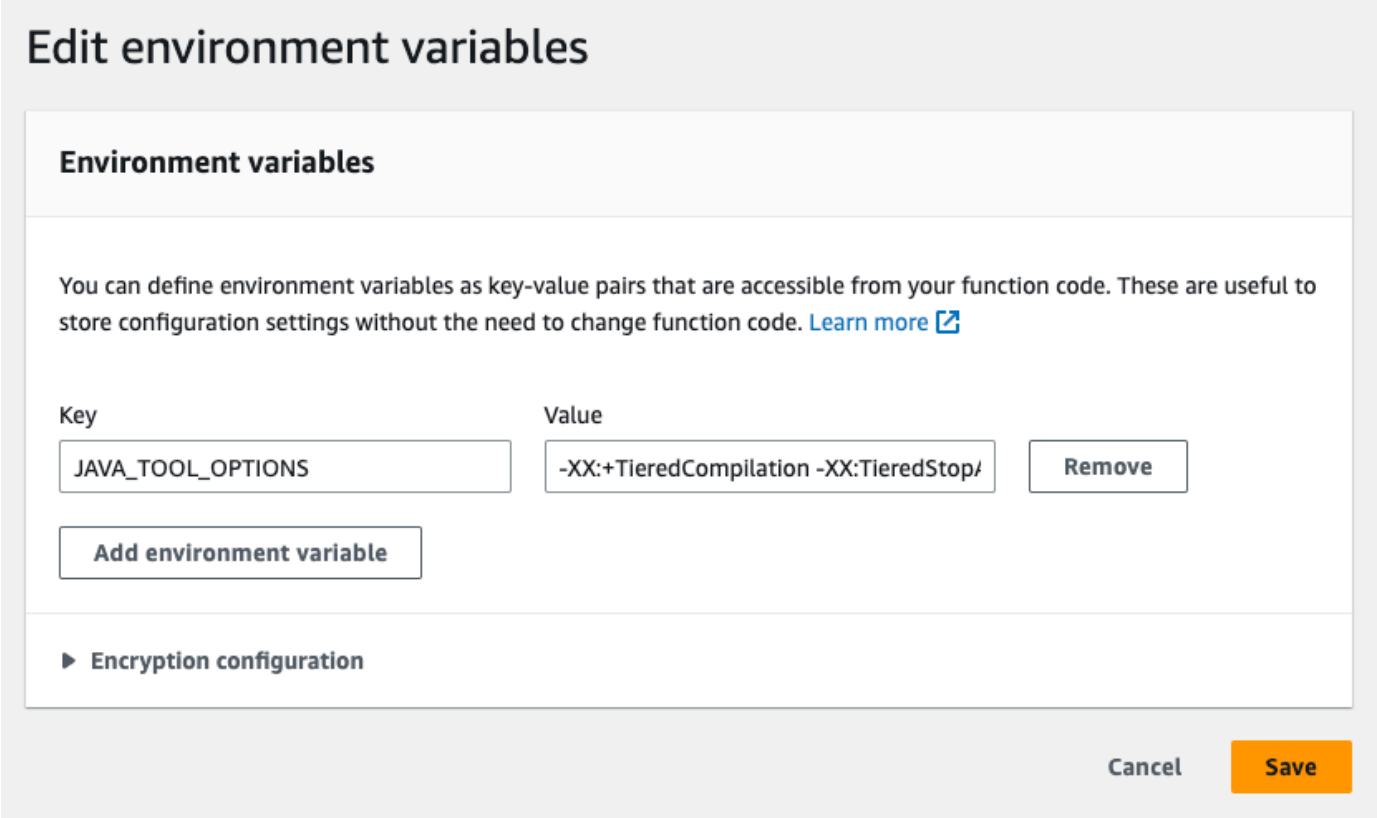
分層編譯是 Java 虛擬機器 (JVM) 的功能。您可以使用特定的分層編譯設定來充分利用 JVM 的即時 (JIT) 編譯器。通常，C1 編譯器針對快速啟動時間進行了最佳化。C2 編譯器針對最佳整體效能進行了最佳化，但也使用更多記憶體，並且需要更長的時間來達成目標。有 5 個不同等級的分層編譯。在層級 0，JVM 會解譯 Java 位元組程式碼。在層級 4，JVM 會使用 C2 編譯器來分析應用程式啟動期間收集的分析資料。隨著時間的推移，它會監控程式碼使用情況以識別最佳化。

自訂分層編譯層級可協助您調整 Java 函數效能。對於快速執行的小型函數，將分層編譯設定為層級 1 有助於透過讓 JVM 使用 C1 編譯器來改善冷啟動效能。此設定可快速產生最佳化的原生程式碼，但不會產生任何分析資料，也不會使用 C2 編譯器。對於運算密集的較大函數，將分層編譯設定為第 4 級，可最大限度地提高整體效能，而無需支付佈建每個 Lambda 執行環境後第一次調用期間的額外記憶體消耗和額外最佳化工作。

對於 Java 11 執行期及以下版本，Lambda 會使用預設 JVM 分層編譯設定。對於 Java 17 和 Java 21，Lambda 預設會將 JVM 設定為停止層級 1 的分層編譯。從 Java 25 開始，Lambda 預設仍會停止層級 1 的分層編譯，除非在使用 SnapStart 或佈建並行時，在這種情況下會使用預設 JVM 設定。這可改善 SnapStart 和佈建並行的效能，而不會產生冷啟動懲罰，因為分層編譯在這些情況下是在調用路徑之外執行。若要最大化此優點，您可以在 SnapStart 快照之前或預先佈建佈建並行執行環境時，使用 priming - 在函數初始化期間執行程式碼路徑來觸發 JIT。如需詳細資訊，請參閱部落格文章，使用進階入門策略搭配 SnapStart 最佳化 AWS Lambda 的冷啟動效能。

自訂分層編譯設定 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您要自訂分層編譯的 Java 函數。
3. 選擇組態索引標籤，然後選擇左側選單中的環境變數。
4. 選擇編輯。
5. 選擇 Add environment variable (新增環境變數)。
6. 針對索引鍵，輸入 `JAVA_TOOL_OPTIONS`。針對值，輸入 `-XX:+TieredCompilation -XX:TieredStopAtLevel=1`。



7. 選擇儲存。

> **Note**
>
> 您也可以使用 Lambda SnapStart 來解決冷啟動問題。SnapStart 會使用執行環境的快取快照來顯著改善啟動效能。如需有關 SnapStart 功能、限制和支援區域的詳細資訊，請參閱使用 Lambda SnapStart 改善啟動效能。

範例：使用 JAVA_TOOL_OPTIONS 自訂 GC 行為

Java 11 執行期使用串行垃圾回收器 (GC) 進行垃圾回收。依預設，Java 17 執行期也會使用串行垃圾回收器。但是，對於 Java 17，您也可以使用 JAVA_TOOL_OPTIONS 環境變數來變更預設垃圾回收器。可以在並行垃圾回收器和 Shenandoah 垃圾回收器之間進行選擇。

例如，如果工作負載使用更多記憶體和多個 CPU，則請考慮使用並行垃圾回收器以獲得更好的效能。可以透過將下列內容附加到 JAVA_TOOL_OPTIONS 環境變數的值來執行此操作：

```
-XX:+UseParallelGC
```

如果您的工作負載有許多短期物件，您可以透過啟用 Java 25 中引入的 Shenandoah 垃圾收集器的世代模式，從較低的記憶體耗用中獲益。若要執行此作業，請將下列項目附加至JAVA_TOOL_OPTIONS環境變數的值：

```
-XX:+UseShenandoahGC -XX:ShenandoahGCMode=generational
```

Log4Shell 的 Log4j 修補程式 Log4Shell

Java 8、11、17 和 21 的 Lambda 執行時間包含修補程式，可減輕 Log4Shell 漏洞 (CVE-2021-44228)，Log4j熱門的 Java 記錄架構。此修補程式會產生冷啟動效能額外負荷。如果您使用的是修補程式版本的 Log4j (2.17.0 版或更新版本），您可以停用此修補程式來改善冷啟動效能。若要停用修補程式，請將AWS_LAMBDA_DISABLE_CVE_2021_44228_PROTECTION環境變數設定為 true。

從 Java 25 開始，Lambda 執行時間不再包含 Log4Shell 修補程式。您必須驗證您使用的是 Log4j 2.17.0 版或更新版本。

Ahead-of-Time(AOT) 和 CDS 快取

從 Java 25 開始，Lambda 執行時間包含 Java 執行時間界面用戶端 (RIC) Ahead-of-Time (AOT) 快取，這是一種執行時間元件，可主動輪詢來自 Lambda 執行時間 API 的事件。這可改善冷啟動效能。

AOT 快取專屬於 JVM 組建。當 Lambda 更新受管執行期時，也會更新 RIC 的 AOT 快取。不過，如果您部署自己的 AOT 快取，這些快取可能會失效，或在執行時間更新後導致意外行為。因此，強烈建議在使用受管執行期時不要使用 AOT 快取。若要使用 AOT 快取，您應該使用容器映像部署函數。

AOT 快取無法與類別資料共用 (CDS) 快取搭配使用。如果您在 Lambda 函數中部署 CDS 快取，則 Lambda 會停用 AOT 快取。

使用 Lambda 內容物件擷取 Java 函數資訊

當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。此物件提供的方法和各項屬性提供了有關調用、函式以及執行環境的資訊。

內容方法

- `getRemainingTimeInMillis()` - 傳回執行逾時前剩餘的毫秒數。
- `getFunctionName()` – 傳回 Lambda 函數的名稱。
- `getFunctionVersion()` - 傳回函數的版本。
- `getInvokedFunctionArn()` - 傳回用於叫用此函數的 Amazon Resource Name (ARN)。指出叫用者是否指定版本號或別名。
- `getMemoryLimitInMB()` - 傳回分配給函數的記憶體數量。
- `getAwsRequestId()` - 傳回叫用請求的識別符。
- `getLogGroupName()` - 傳回函數的日誌群組。
- `getLogStreamName()` - 傳回函數執行個體的記錄串流。
- `getIdentity()` - (行動應用程式) 傳回已授權請求的 Amazon Cognito 身分的相關資訊。
- `getClientContext()` - (行動應用程式) 傳回用戶端應用程式提供給 Lambda 的用戶端內容。
- `getLogger()` - 傳回函數的 Logger 物件。

下面的例子顯示使用內容物件存取 Lambda 記錄器的函數。

Example Handler.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import java.util.Map;

// Handler value: example.Handler
public class Handler implements RequestHandler<Map<String,String>, Void>{

  @Override
  public Void handleRequest(Map<String,String> event, Context context)
```

```
  {
    LambdaLogger logger = context.getLogger();
    logger.log("EVENT TYPE: " + event.getClass());
    return null;
  }
}
```

函數會先記錄傳入事件的類別類型，再傳回 null。

Example記錄輸出

```
EVENT TYPE: class java.util.LinkedHashMap
```

內容物件的介面可在 aws-lambda-java-core 程式庫中使用。您可以實作此介面來建立測試的內容類別。下面的例子顯示針對大多數屬性和進行中測試記錄器傳回虛擬值的內容類別。

Example src/test/java/example/TestContext.java

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.CognitoIdentity;
import com.amazonaws.services.lambda.runtime.ClientContext;
import com.amazonaws.services.lambda.runtime.LambdaLogger;

public class TestContext implements Context{

  public TestContext() {}
  public String getAwsRequestId(){
    return new String("495b12a8-xmpl-4eca-8168-160484189f99");
  }
  public String getLogGroupName(){
    return new String("/aws/lambda/my-function");
  }
  public String getLogStreamName(){
    return new String("2020/02/26/[$LATEST]704f8dxmpla04097b9134246b8438f1a");
  }
  public String getFunctionName(){
    return new String("my-function");
  }
  public String getFunctionVersion(){
    return new String("$LATEST");
  }
```

```
  public String getInvokedFunctionArn(){
    return new String("arn:aws:lambda:us-east-2:123456789012:function:my-function");
  }
  public CognitoIdentity getIdentity(){
    return null;
  }
  public ClientContext getClientContext(){
    return null;
  }
  public int getRemainingTimeInMillis(){
    return 300000;
  }
  public int getMemoryLimitInMB(){
    return 512;
  }
  public LambdaLogger getLogger(){
    return new TestLogger();
  }

}
```

如需記錄日誌的詳細資訊，請參閱 記錄和監控 Java Lambda 函數。

範例應用程式中的內容

本指南的 GitHub 儲存庫包含示範內容物件使用方式的範例應用程式。每個範例應用程式都包含可輕鬆部署和清理的指令碼、AWS Serverless Application Model (AWS SAM) 範本和支援資源。

以 Java 編寫的範例 Lambda 應用程式

- example-java – Java 函式，示範如何使用 Lambda 處理訂單。此函式示範如何定義並反序列化自訂輸入事件物件、使用 AWS SDK 以及輸出日誌。
- java-basic - 具有單元測試和變數日誌組態的最小 Java 函數集合。
- java-events - Java 函數集合，其中包含如何處理來自各種服務 (例如 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis) 事件的骨架程式碼。這些函數使用最新版 aws-lambda-java-events 程式庫 (3.0.0 及更新版)。這些範例不需要 AWS 開發套件做為相依項目。
- s3-java - 一種 Java 函數，它處理來自 Amazon S3 的通知事件，並使用 Java Class Library (JCL) 以從上傳的映像檔案建立縮圖。
- layer-java – 此 Java 函式示範如何使用 Lambda 層，將相依項與核心函式程式碼分離封裝。

記錄和監控 Java Lambda 函數

AWS Lambda 會自動監控 Lambda 函數，並將日誌項目傳送至 Amazon CloudWatch。您的 Lambda 函數隨附有 CloudWatch Logs 日誌群組，且函數的每一執行個體各有一個日誌串流。Lambda 執行期環境會將每次調用的詳細資訊和函數程式碼的其他輸出，傳送至日誌串流。如需 CloudWatch Logs 的詳細資訊，請參閱將 Lambda 函式日誌傳送至 CloudWatch Logs。

若要由您的函數程式碼輸出日誌，可以使用 java.lang.System 的方法，或任何能寫入 stdout 或 stderr 的記錄模組。

章節

- 建立傳回日誌的函數
- 搭配 Java 使用 Lambda 進階日誌控制項
- 使用 Log4j2 和 SLF4J 實作進階日誌記錄
- 使用其他記錄工具和程式庫
- 使用 Powertools for AWS Lambda (Java) 和 AWS SAM 進行結構化記錄
- 在 Lambda 主控台檢視日誌
- 在 CloudWatch 主控台中檢視 記錄
- 使用 AWS Command Line Interface (AWS CLI) 檢視日誌
- 刪除日誌
- 日誌記錄程式碼範例

建立傳回日誌的函數

若要由您的函數程式碼輸出日誌，您可以使用 java.lang.System 的方法，或任何能寫入 `stdout` 或 `stderr` 的記錄模組。在 aws-lambda-java-core 程式庫會提供了一個名為 `LambdaLogger` 的記錄器類別，您可以從內容物件加以存取。記錄器類別支援多行日誌。

下面範例使用由內容物件提供的 `LambdaLogger` 記錄器。

Example Handler.java

```
// Handler value: example.Handler
public class Handler implements RequestHandler<Object, String>{
  Gson gson = new GsonBuilder().setPrettyPrinting().create();
  @Override
  public String handleRequest(Object event, Context context)
```

```
  {
    LambdaLogger logger = context.getLogger();
    String response = new String("SUCCESS");
    // log execution details
    logger.log("ENVIRONMENT VARIABLES: " + gson.toJson(System.getenv()));
    logger.log("CONTEXT: " + gson.toJson(context));
    // process event
    logger.log("EVENT: " + gson.toJson(event));
    return response;
  }
}
```

Example記錄格式

```
START RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Version: $LATEST
ENVIRONMENT VARIABLES:
{
    "_HANDLER": "example.Handler",
    "AWS_EXECUTION_ENV": "AWS_Lambda_java8",
    "AWS_LAMBDA_FUNCTION_MEMORY_SIZE": "512",
    ...
}
CONTEXT:
{
    "memoryLimit": 512,
    "awsRequestId": "6bc28136-xmpl-4365-b021-0ce6b2e64ab0",
    "functionName": "java-console",
    ...
}
EVENT:
{
  "records": [
    {
      "messageId": "19dd0b57-xmpl-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
       ...
    }
  ]
}
END RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0
REPORT RequestId: 6bc28136-xmpl-4365-b021-0ce6b2e64ab0 Duration: 198.50 ms Billed
 Duration: 724 ms Memory Size: 512 MB Max Memory Used: 90 MB Init Duration: 524.75 ms
```

Java 執行時間會記錄每次調用的 START、END 和 REPORT 行。報告明細行提供下列詳細資訊：

REPORT 行資料欄位

- RequestId - 進行調用的唯一請求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。當調用共用執行環境時，Lambda 會報告所有調用使用的記憶體上限。此行為可能會導致報告值高於預期值。
- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId - 對於追蹤的請求，這是 AWS X-Ray 追蹤 ID。
- SegmentId - 對於追蹤的請求，這是 X-Ray 區段 ID。
- 已取樣 - 對於追蹤的請求，這是取樣結果。

搭配 Java 使用 Lambda 進階日誌控制項

為了讓您更妥善地控制擷取、處理和使用函數日誌的方式，您可以針對支援的 Java 執行期設定下列記錄選項：

- 日誌格式 - 在純文字和結構化 JSON 格式之間為您的日誌進行選擇
- 日誌層級 - 對於 JSON 格式的日誌，請選擇 Lambda 傳送到 CloudWatch 的日誌之詳細等級，例如 ERROR、DEBUG 或 INFO
- 日誌群組 - 選擇您的函數將日誌傳送到的 CloudWatch 日誌群組

如需這些日誌選項的詳細資訊，以及如何設定函數以使用這些選項的說明，請參閱 the section called “設定 Lambda 函數的進階日誌記錄控制項”。

若要使用日誌格式和日誌層級選項與 Java Lambda 函數搭配使用，請參閱以下各章節中的指引。

搭配 Java 使用結構化 JSON 日誌格式

如果您為函數的日誌格式選取 JSON，Lambda 會以結構化 JSON 形式使用 `LambdaLogger` 類別傳送日誌輸出。每個 JSON 日誌物件都包含至少四個鍵值對，其中包含下列索引鍵：

- `"timestamp"` - 產生日誌訊息的時間
- `"level"` - 指派給訊息的日誌層級
- `"message"` - 日誌訊息的內容
- `"AWSrequestId"` - 進行調用的唯一請求 ID。

視您使用的記錄方法而定，以 JSON 格式擷取之函數的日誌輸出也可以包含其他鍵值對。

若要將您使用 `LambdaLogger` 記錄器建立的日誌指派一個層級，你需要在你的日誌命令提供一個 `LogLevel` 引數，如以下的例子。

Example Java 日誌程式碼

```
LambdaLogger logger = context.getLogger();
logger.log("This is a debug log", LogLevel.DEBUG);
```

此範例程式碼所輸出的日誌檔會在 CloudWatch Logs 中擷取，如下所示：

Example JSON 日誌記錄

```
{
    "timestamp":"2023-11-01T00:21:51.358Z",
    "level":"DEBUG",
    "message":"This is a debug log",
    "AWSrequestId":"93f25699-2cbf-4976-8f94-336a0aa98c6f"
}
```

如果您沒有為日誌輸出指派層級，Lambda 會自動為其指派層級 INFO。

如果您的程式碼已經使用另一個記錄程式庫來生成 JSON 結構化日誌，則不需要進行任何更改。Lambda 不會對任何已經進行 JSON 編碼的記錄進行雙重編碼。即使您將函數設定為使用 JSON 日誌格式，您的記錄輸出也會以您定義的 JSON 結構顯示於 CloudWatch 中。

搭配 Java 使用日誌層級篩選

若要 AWS Lambda 讓 根據應用程式日誌層級篩選您的應用程式日誌，您的函數必須使用 JSON 格式的日誌。您可以透過兩種方式達成此操作：

- 使用標準 `LambdaLogger` 建立日誌輸出，並將函數設定為使用 JSON 日誌格式。然後，Lambda 會使用 the section called “搭配 Java 使用結構化 JSON 日誌格式” 中所述 JSON 物件中的「層級」索

引鍵值組篩選您的日誌輸出。若要瞭解如何設定函數的日誌格式，請參閱 the section called “設定 Lambda 函數的進階日誌記錄控制項”。

- 使用其他日誌程式庫或方法，在您的程式碼中建立 JSON 結構化日誌，其中包含定義日誌輸出層級的「層級」索引鍵值組。您可以使用可將 JSON 日誌寫入 `stdout` 或 `stderr` 的任何記錄程式庫。例如，您可以使用 Powertools for AWS Lambda 或 Log4j2 套件，從您的程式碼產生 JSON 結構化日誌輸出。如需進一步了解，請參閱 the section called “使用 Powertools for AWS Lambda (Java) 和 AWS SAM 進行結構化記錄” 和 the section called “使用 Log4j2 和 SLF4J 實作進階日誌記錄”。

將函數設定為使用日誌層級篩選時，您必須從下列選項中選取要 Lambda 傳送至 CloudWatch Logs 的日誌層級：

日誌層級	標準用量
TRACE (大多數詳細資訊)	用於追蹤程式碼執行路徑的最精細資訊
DEBUG	系統偵錯的詳細資訊
INFO	記錄函數正常操作的訊息
WARN	有關可能導致未解決意外行為的潛在錯誤的消息
ERROR	有關阻止程式碼按預期執行的問題的訊息
FATAL (最少詳細資訊)	有關導致應用程式停止運作的嚴重錯誤訊息

若要讓 Lambda 篩選函數的日誌，您還必須在 JSON 日誌輸出中包含 `"timestamp"` 索引鍵值組。必須以有效的 RFC 3339 時間戳記格式指定時間。如果您沒有提供有效的時間戳記，Lambda 會為日誌指派層級 INFO，並為您新增時間戳記。

Lambda 會在選取的層級 (含) 和更低層級傳送日誌給 CloudWatch。例如，如果您設定 WARN 的日誌層級，Lambda 會傳送相對應於 WARN、ERROR 和 FATAL 層級的日誌檔。

使用 Log4j2 和 SLF4J 實作進階日誌記錄

> **Note**
>
> AWS Lambda 在其受管執行時間或基礎容器映像中不包含 Log4j2。因此，這些問題不受 CVE-2021-44228、CVE-2021-45046 以及 CVE-2021-45105 中描述的問題的影響。

對於客戶函數包含受影響的 Log4j2 版本的情況，我們已將變更套用至 Lambda Java 受管執行時間和基礎容器映像，這有助於緩解 CVE-2021-44228、CVE-2021-45046 和 CVE-2021-45105 中的問題。由於此變更，使用 Log4J2 的客戶可能會看到一個額外的日誌條目，類似於「`Transforming org/apache/logging/log4j/core/lookup/JndiLookup (java.net.URLClassLoader@...)`」。在 Log4J2 輸出中參考 jndi 映射器的任何日誌字串都將替換為「`Patched JndiLookup::lookup()`」。
除此變更之外，我們強烈建議其函數包含 Log4j2 的所有客戶將 Log4j2 更新至最新版本。具體而言，在函數中使用 aws-lambda-java-log4j2 程式庫的客戶應更新至 1.5.0 版 (或更高版本)，並重新部署其函數。此版本將基礎 Log4j2 公用程式相依性更新為 2.17.0 版 (或更高版本)。更新後的 aws-lambda-java-log4j2 二進位文件可在 Maven 儲存庫中找到，而其原始碼可在 GitHub 中找到。
最後，請注意，在任何情況下都不應使用與 aws-lambda-java-log4j (v1.0.0 or 1.0.1) 相關的任何程式庫。這些程式庫與 log4j 的 第 1.x 版本相關，該版本已於 2015 年停止使用。這些程式庫不受支援、未受維護、未經修補，且具有已知的安全性漏洞。

若要自訂日誌輸出、在單元測試期間支援日誌記錄，以及日誌 AWS SDK 呼叫，請使用 Apache Log4j2 搭配 SLF4J。Log4j 是適用於 Java 程式的日誌程式庫，讓您能夠配置日誌級別和使用附加器程式庫。SLF4J 是一個外觀程式庫，可讓您改變使用的程式庫，而無須您的函數程式碼。

若要將請求 ID 新增至函數的日誌中，請使用 aws-lambda-java-log4j2 程式庫中的附加器。

Example src/main/resources/log4j2.xml - 附加器組態

```
<Configuration>
  <Appenders>
    <Lambda name="Lambda" format="${env:AWS_LAMBDA_LOG_FORMAT:-TEXT}">
       <LambdaTextFormat>
         <PatternLayout>
             <pattern>%d{yyyy-MM-dd HH:mm:ss} %X{AWSRequestId} %-5p %c{1} - %m%n </
pattern>
         </PatternLayout>
       </LambdaTextFormat>
       <LambdaJSONFormat>
         <JsonTemplateLayout eventTemplateUri="classpath:LambdaLayout.json" />
       </LambdaJSONFormat>
     </Lambda>
   </Appenders>
   <Loggers>
     <Root level="${env:AWS_LAMBDA_LOG_LEVEL:-INFO}">
```

```
      <AppenderRef ref="Lambda"/>
    </Root>
    <Logger name="software.amazon.awssdk" level="WARN" />
    <Logger name="software.amazon.awssdk.request" level="DEBUG" />
  </Loggers>
</Configuration>
```

您可以透過在 <LambdaTextFormat> 和 <LambdaJSONFormat> 標籤下指定佈局來決定如何將 Log4j2 日誌配置為純文本或 JSON 輸出。

在此範例中，每一行前面都會以文字模式加上日期、時間、請求 ID、日誌層級和類別名稱。在 JSON 模式下<JsonTemplateLayout>，會與 aws-lambda-java-log4j2 程式庫一起隨附的組態搭配使用。

SLF4J 是使用 Java 程式碼進行日誌記錄的外觀程式庫。在您的函數程式碼中，您可以使用 SLF4J 記錄器工廠來擷取帶有日誌層級方法的記錄器，如 info() 和 warn()。在您的建置組態中，您可將日誌記錄程式庫和 SLF4J 轉接器包含在 classpath 中。透過在建置配置中更改程式庫，您可以在不更改函數代碼的情況下更改記錄器類型。需要 SLF4J 才能從 SDK for Java 中擷取日誌。

在下面範例程式碼中，處理常式類別會使用 SLF4J 來擷取記錄器。

Example src/main/java/example/HandlerS3.java – 透過 SLF4J 進行日誌記錄

```
package example;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;

import static org.apache.logging.log4j.CloseableThreadContext.put;


public class HandlerS3 implements RequestHandler<S3Event, String>{
    private static final Logger logger = LoggerFactory.getLogger(HandlerS3.class);

    @Override
    public String handleRequest(S3Event event, Context context) {
        for(var record : event.getRecords()) {
            try (var loggingCtx = put("awsRegion", record.getAwsRegion())) {
                loggingCtx.put("eventName", record.getEventName());
```

```
                loggingCtx.put("bucket", record.getS3().getBucket().getName());
                loggingCtx.put("key", record.getS3().getObject().getKey());

                logger.info("Handling s3 event");
            }
        }

        return "Ok";
    }
}
```

此程式碼產生的日誌輸出如以下所示。

Example記錄格式

```
{
    "timestamp": "2023-11-15T16:56:00.815Z",
    "level": "INFO",
    "message": "Handling s3 event",
    "logger": "example.HandlerS3",
    "AWSRequestId": "0bced576-3936-4e5a-9dcd-db9477b77f97",
    "awsRegion": "eu-south-1",
    "bucket": "java-logging-test-input-bucket",
    "eventName": "ObjectCreated:Put",
    "key": "test-folder/"
}
```

建置組態會使用 Lambda 附加器和 SLF4J 轉接器的執行期相依性，以及 Log4j2 的實作相依性。

Example build.gradle - 日誌記錄相依性

```
dependencies {
    ...
    'com.amazonaws:aws-lambda-java-log4j2:[1.6.0,)',
    'com.amazonaws:aws-lambda-java-events:[3.11.3,)',
    'org.apache.logging.log4j:log4j-layout-template-json:[2.17.1,)',
    'org.apache.logging.log4j:log4j-slf4j2-impl:[2.19.0,)',
    ...
}
```

當您在本機執行程式碼進行測試時，帶有 Lambda 記錄器的內容物件將無法使用，並且 Lambda 附加器沒有可使用的請求 ID。對於範例測試組態，請參閱下節中的範例應用程式。

使用其他記錄工具和程式庫

Powertools for AWS Lambda (Java) 是一種開發人員工具組，可實作無伺服器最佳實務並提高開發人員速度。記錄公用程式提供 Lambda 優化記錄器，其中包含有關所有函數之函數內容的其他資訊，輸出結構為 JSON。使用此公用程式執行下列操作：

- 從 Lambda 內容、冷啟動和 JSON 形式的結構記錄輸出中擷取關鍵欄位
- 在收到指示時記錄 Lambda 調用事件 (預設為停用)
- 透過日誌採樣僅列印調用百分比的所有日誌 (預設為停用)
- 在任何時間點將其他金鑰附加至結構化日誌
- 使用自訂日誌格式化程式 (自帶格式化程式)，以與組織的日誌記錄 RFC 相容的結構輸出日誌。

使用 Powertools for AWS Lambda (Java) 和 AWS SAM 進行結構化記錄

請依照下列步驟，使用 下載、建置和部署範例 Hello World Java 應用程式，並整合 Powertools for AWS Lambda (Java) 模組 AWS SAM。此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會調用、使用內嵌指標格式將日誌和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- Java 11
- AWS CLI 第 2 版
- AWS SAM CLI 1.75 版或更新版本。如果您有較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World Java 範本來初始化應用程式。

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 Enter。

> **Note**
>
> 對於 HelloWorldFunction may not have authorization defined, Is this okay?，確保輸入 y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要獲取該函數的日誌，請執行 sam 日誌。如需詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》 中的 使用日誌。

```
sam logs --stack-name sam-app
```

日誌輸出如下：

```
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.095000
 INIT_START Runtime Version: java:11.v15    Runtime Version ARN: arn:aws:lambda:eu-
central-1::runtime:0a25e3e7a1cc9ce404bc435eeb2ad358d8fa64338e618d0c224fe509403583ca
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.114000
 Picked up JAVA_TOOL_OPTIONS: -XX:+TieredCompilation -XX:TieredStopAtLevel=1
```

```
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:34.793000
 Transforming org/apache/logging/log4j/core/lookup/JndiLookup
 (lambdainternal.CustomerClassLoader@1a6c5a9e)
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:35.252000
 START RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765 Version: $LATEST
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.531000 {
  "_aws": {
    "Timestamp": 1675416276051,
    "CloudWatchMetrics": [
      {
        "Namespace": "sam-app-powerools-java",
        "Metrics": [
          {
            "Name": "ColdStart",
            "Unit": "Count"
          }
        ],
        "Dimensions": [
          [
            "Service",
            "FunctionName"
          ]
        ]
      }
    ]
  },
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "FunctionName": "sam-app-HelloWorldFunction-y9Iu1FLJJBGD",
  "functionVersion": "$LATEST",
  "ColdStart": 1.0,
  "Service": "service_undefined",
  "logStreamId": "2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
  "executionEnvironment": "AWS_Lambda_java11"
}
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.974000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.AWSXRayRecorder <init>
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000 Feb
 03, 2023 9:24:36 AM com.amazonaws.xray.config.DaemonConfiguration <init>
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:36.993000
 INFO: Environment variable AWS_XRAY_DAEMON_ADDRESS is set. Emitting to daemon on
 address XXXX.XXXX.XXXX.XXXX:2000.
```

```
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.331000
 09:24:37.294 [main] INFO  helloworld.App - {"version":null,"resource":"/
hello","path":"/hello/","httpMethod":"GET","headers":{"Accept":"*/
*","CloudFront-Forwarded-Proto":"https","CloudFront-Is-Desktop-
Viewer":"true","CloudFront-Is-Mobile-Viewer":"false","CloudFront-Is-
SmartTV-Viewer":"false","CloudFront-Is-Tablet-Viewer":"false","CloudFront-
Viewer-ASN":"16509","CloudFront-Viewer-Country":"IE","Host":"XXXX.execute-
api.eu-central-1.amazonaws.com","User-Agent":"curl/7.86.0","Via":"2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)","X-Amz-
Cf-Id":"t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q==","X-
Amzn-Trace-Id":"Root=1-63dcd2d1-25f90b9d1c753a783547f4dd","X-Forwarded-
For":"XX.XXX.XXX.XX, XX.XXX.XXX.XX","X-Forwarded-Port":"443","X-
Forwarded-Proto":"https"},"multiValueHeaders":{"Accept":["*/
*"],"CloudFront-Forwarded-Proto":["https"],"CloudFront-Is-Desktop-Viewer":
["true"],"CloudFront-Is-Mobile-Viewer":["false"],"CloudFront-Is-SmartTV-
Viewer":["false"],"CloudFront-Is-Tablet-Viewer":["false"],"CloudFront-Viewer-
ASN":["16509"],"CloudFront-Viewer-Country":["IE"],"Host":["XXXX.execute-
api.eu-central-1.amazonaws.com"],"User-Agent":["curl/7.86.0"],"Via":["2.0
 f0300a9921a99446a44423d996042050.cloudfront.net (CloudFront)"],"X-Amz-
Cf-Id":["t9W5ByT11HaY33NM8YioKECn_4eMpNsOMPfEVRczD7T1RdhbtiwV1Q=="],"X-
Amzn-Trace-Id":["Root=1-63dcd2d1-25f90b9d1c753a783547f4dd"],"X-Forwarded-
For":["XXX, XXX"],"X-Forwarded-Port":["443"],"X-Forwarded-Proto":
["https"]},"queryStringParameters":null,"multiValueQueryStringParameters":null,"pathParamet
{"accountId":"XXX","stage":"Prod","resourceId":"at73a1","requestId":"ba09ecd2-
acf3-40f6-89af-fad32df67597","operationName":null,"identity":
{"cognitoIdentityPoolId":null,"accountId":null,"cognitoIdentityId":null,"caller":null,"apiK
hello","httpMethod":"GET","apiId":"XXX","path":"/Prod/
hello/","authorizer":null},"body":null,"isBase64Encoded":false}
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:37.351000
 09:24:37.351 [main] INFO  helloworld.App - Retrieving https://
checkip.amazonaws.com
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.313000 {
  "function_request_id": "7fcf1548-d2d4-41cd-a9a8-6ae47c51f765",
  "traceId":
 "Root=1-63dcd2d1-25f90b9d1c753a783547f4dd;Parent=e29684c1be352ce4;Sampled=1",
  "xray_trace_id": "1-63dcd2d1-25f90b9d1c753a783547f4dd",
  "functionVersion": "$LATEST",
  "Service": "service_undefined",
  "logStreamId": "2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81",
  "executionEnvironment": "AWS_Lambda_java11"
}
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000 END
 RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765
```

```
2025/09/03/[$LATEST]851411a899b545eea2cffeba4cfbec81 2023-02-03T09:24:39.371000
 REPORT RequestId: 7fcf1548-d2d4-41cd-a9a8-6ae47c51f765    Duration: 4118.98 ms
 Billed Duration: 5275 ms    Memory Size: 512 MB    Max Memory Used: 152 MB    Init
 Duration: 1155.47 ms
XRAY TraceId: 1-63dcd2d1-25f90b9d1c753a783547f4dd    SegmentId: 3a028fee19b895cb
 Sampled: true
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

管理日誌保留

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，CloudWatch 會在該時間之後自動刪除日誌。若要設定日誌保留，請將下列項目新增至您的 AWS SAM 範本：

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

在 Lambda 主控台檢視日誌

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

在 CloudWatch 主控台中檢視 記錄

您可以使用 Amazon CloudWatch 主控台來檢視所有 Lambda 函數調用的日誌。

若要在 CloudWatch 主控台上檢視日誌

1. 在 CloudWatch 主控台上開啟日誌群組頁面。
2. 選擇您的函數的日誌群組 (/aws/lambda/*your-function-name*)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立其他執行個體以處理並行調用時，就會顯示日誌串流。若要尋找特定調用的日誌，建議您使用 檢測函數 AWS X-Ray。X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

使用 AWS Command Line Interface (AWS CLI) 檢視日誌

是一種 AWS CLI 開放原始碼工具，可讓您使用命令列 Shell 中的 命令與 AWS 服務互動。若要完成本節中的步驟，您必須擁有 AWS CLI 版本 2。

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example解碼日誌

在相同的命令提示中，使用 `base64` 公用程式來解碼日誌。下列範例顯示如何擷取 `my-function` 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
```

```
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 `base64` 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 `base64 -D`。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 `sed` 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 `get-log-events` 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 `get-logs.sh`。

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
```

```
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

日誌記錄程式碼範例

本指南的 GitHub 儲存庫包含示範各種日誌記錄組態使用方式的範例應用程式。每個範例應用程式都包含易於部署和清除的指令碼、 AWS SAM 範本和支援資源。

以 Java 編寫的範例 Lambda 應用程式

- example-java – Java 函式，示範如何使用 Lambda 處理訂單。此函數說明如何定義和還原序列化自訂輸入事件物件、使用 AWS SDK 和輸出記錄。
- java-basic - 具有單元測試和變數日誌組態的最小 Java 函數集合。
- java-events - Java 函數集合，其中包含如何處理來自各種服務 (例如 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis) 事件的骨架程式碼。這些函數使用最新版 aws-lambda-java-events 程式庫 (3.0.0 及更新版)。這些範例不需要 AWS SDK 做為相依性。
- s3-java - 一種 Java 函數，它處理來自 Amazon S3 的通知事件，並使用 Java Class Library (JCL) 以從上傳的映像檔案建立縮圖。
- layer-java – 此 Java 函式示範如何使用 Lambda 層，將相依項與核心函式程式碼分離封裝。

`java-basic` 範例應用程式會顯示支援日誌記錄測試的最小日誌記錄組態。處理常式程式碼會使用內容物件提供的 `LambdaLogger` 記錄器。對於測試，應用程式會使用實作具有 Log4j2 記錄器的 `LambdaLogger` 介面的自訂 `TestLogger` 類別。它使用 SLF4J 做為與 AWS SDK 相容的外觀。建置輸出中會排除記錄程式庫，使部署套件不會變太大。

在 中檢測 Java 程式碼 AWS Lambda

Lambda 與 整合 AWS X-Ray ，以協助您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追蹤請求，因為它會周遊您應用程式中的資源，其中可能包含 Lambda 函數和其他 AWS 服務。

若要將追蹤資料傳送至 X-Ray，您可以使用以下兩個 SDK 庫之一：

- AWS Distro for OpenTelemetry (ADOT) – OpenTelemetry (OTel) SDK 的安全、生產就緒、 AWS支援的分發。
- 適用於 JAVA 的 AWS X-Ray SDK – 用於生成追蹤資料並將其傳送至 X-Ray 的 SDK。
- Powertools for AWS Lambda (Java) – 開發人員工具組，用於實作無伺服器最佳實務並提高開發人員速度。

每個 SDK 均提供將遙測資料傳送至 X-Ray 服務的方法。然後，您可以使用 X-Ray 來檢視、篩選應用程式的效能指標並獲得洞察，從而識別問題和進行最佳化的機會。

> **Important**
>
> X-Ray 和 Powertools AWS Lambda SDKs是 提供的緊密整合檢測解決方案的一部分 AWS。ADOT Lambda Layers 是用於追蹤檢測之業界通用標準的一部分，這類檢測一般會收集更多資料，但可能不適用於所有使用案例。您可以使用任一解決方案在 X-Ray 中實作端對端追蹤。若要深入了解如何在兩者之間做選擇，請參閱在 AWS Distro for OpenTelemetry 和 X-Ray SDK 之間進行選擇。

章節

- 使用 Powertools for AWS Lambda (Java) 和 AWS SAM 進行追蹤
- 使用 Powertools for AWS Lambda (Java) 和 AWS CDK 進行追蹤
- 使用 ADOT 來檢測您的 Java 函數
- 使用 X-Ray SDK 來檢測 Java 功能
- 透過 Lambda 主控台來啟用追蹤
- 透過 Lambda API 啟用追蹤
- 使用 啟用追蹤 CloudFormation
- 解讀 X-Ray 追蹤
- 將執行時間相依項存放存在層中 (X-Ray SDK)

- 樣本應用程式中的 X-Ray 追蹤 (X-Ray SDK)

使用 Powertools for AWS Lambda (Java) 和 AWS SAM 進行追蹤

請依照下列步驟，使用 下載、建置和部署範例 Hello World Java 應用程式，並整合 Powertools for AWS Lambda (Java) 模組 AWS SAM。此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會調用、使用內嵌指標格式將日誌和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- Java 11 或更新版本
- AWS CLI 第 2 版
- AWS SAM CLI 1.75 版或更新版本。如果您有較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World Java 範本來初始化應用程式。

```
sam init --app-template hello-world-powertools-java --name sam-app --package-type
 Zip --runtime java11 --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 `Enter`。

> **Note**
>
> 對於 HelloWorldFunction may not have authorization defined, Is this okay?，確保輸入 y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

```
New XRay Service Graph
  Start time: 2025-02-03 14:31:48+01:00
  End time: 2025-02-03 14:31:48+01:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-y9Iu1FLJJBGD -
 Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 5.587
  Reference Id: 1 - client - sam-app-HelloWorldFunction-y9Iu1FLJJBGD - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
```

```
    - fault count(5XX): 0
    - total response time: 0

XRay Event [revision 3] at (2025-02-03T14:31:48.500000) with id
 (1-63dd0cc4-3c869dec72a586875da39777) and duration (5.603s)
 - 5.587s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD [HTTP: 200]
 - 4.053s - sam-app-HelloWorldFunction-y9Iu1FLJJBGD
   - 1.181s - Initialization
   - 4.037s - Invocation
     - 1.981s - ## handleRequest
       - 1.840s - ## getPageContents
   - 0.000s - Overhead
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

使用 Powertools for AWS Lambda (Java) 和 AWS CDK 進行追蹤

請依照下列步驟，使用 下載、建置和部署範例 Hello World Java 應用程式，並整合 Powertools for AWS Lambda (Java) 模組 AWS CDK。此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會調用、使用內嵌指標格式將日誌和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。函數會傳回 hello world 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- Java 11 或更新版本
- AWS CLI 第 2 版
- AWS CDK 第 2 版
- AWS SAM CLI 1.75 版或更新版本。如果您有較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS CDK 應用程式

1. 為您的新應用程式建立專案目錄。

```
mkdir hello-world
cd hello-world
```

2. 初始化應用程式。

```
cdk init app --language java
```

3. 使用以下命令來建立 Maven 專案：

```
mkdir app
cd app
mvn archetype:generate -DgroupId=helloworld -DartifactId=Function -
DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

4. 在 hello-world\app\Function 目錄中開啟 pom.xml，並將現有程式碼替換為下面的程式碼，其中包括 Powertools 的相依性和 Maven 外掛程式。

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>helloworld</groupId>
  <artifactId>Function</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Function</name>
  <url>http://maven.apache.org</url>
<properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <log4j.version>2.17.2</log4j.version>
</properties>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
```

```
            <artifactId>powertools-tracing</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-metrics</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>software.amazon.lambda</groupId>
            <artifactId>powertools-logging</artifactId>
            <version>1.3.0</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.2</version>
        </dependency>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-events</artifactId>
            <version>3.11.1</version>
        </dependency>
  </dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>aspectj-maven-plugin</artifactId>
            <version>1.14.0</version>
            <configuration>
                <source>${maven.compiler.source}</source>
                <target>${maven.compiler.target}</target>
                <complianceLevel>${maven.compiler.target}</complianceLevel>
                <aspectLibraries>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-tracing</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-metrics</artifactId>
                    </aspectLibrary>
                    <aspectLibrary>
```

```
                        <groupId>software.amazon.lambda</groupId>
                        <artifactId>powertools-logging</artifactId>
                    </aspectLibrary>
                </aspectLibraries>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>compile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.4.1</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <transformers>
                                <transformer

 implementation="com.github.edwgiz.maven_shade_plugin.log4j2_cache_transformer.PluginsCache
                                </transformer>
                            </transformers>
                            <createDependencyReducedPom>false</
createDependencyReducedPom>
                            <finalName>function</finalName>

                        </configuration>
                    </execution>
                </executions>
                <dependencies>
                    <dependency>
                        <groupId>com.github.edwgiz</groupId>
                        <artifactId>maven-shade-plugin.log4j2-cachefile-
transformer</artifactId>
                        <version>2.15</version>
                    </dependency>
```

```
                </dependencies>
        </plugin>
    </plugins>
</build>
</project>
```

5. 建立 hello-world\app\src\main\resource 目錄並為日誌組態建立 log4j.xml。

```
mkdir -p src/main/resource
cd src/main/resource
touch log4j.xml
```

6. 開啟 log4j.xml 並新增以下程式碼。

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
    <Appenders>
        <Console name="JsonAppender" target="SYSTEM_OUT">
            <JsonTemplateLayout
 eventTemplateUri="classpath:LambdaJsonLayout.json" />
        </Console>
    </Appenders>
    <Loggers>
        <Logger name="JsonLogger" level="INFO" additivity="false">
            <AppenderRef ref="JsonAppender"/>
        </Logger>
        <Root level="info">
            <AppenderRef ref="JsonAppender"/>
        </Root>
    </Loggers>
</Configuration>
```

7. 從 hello-world\app\Function\src\main\java\helloworld 目錄中開啟 App.java，並將現有程式碼替換為下面的程式碼。這是 Lambda 函數的程式碼。

```
package helloworld;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.HashMap;
import java.util.Map;
```

```
import java.util.stream.Collectors;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;
import software.amazon.lambda.powertools.logging.Logging;
import software.amazon.lambda.powertools.metrics.Metrics;
import software.amazon.lambda.powertools.tracing.CaptureMode;
import software.amazon.lambda.powertools.tracing.Tracing;

import static software.amazon.lambda.powertools.tracing.CaptureMode.*;

/**
 * Handler for requests to Lambda function.
 */
public class App implements RequestHandler<APIGatewayProxyRequestEvent,
 APIGatewayProxyResponseEvent> {
    Logger log = LogManager.getLogger(App.class);


    @Logging(logEvent = true)
    @Tracing(captureMode = DISABLED)
    @Metrics(captureColdStart = true)
    public APIGatewayProxyResponseEvent handleRequest(final
 APIGatewayProxyRequestEvent input, final Context context) {
        Map<String, String> headers = new HashMap<>();
        headers.put("Content-Type", "application/json");
        headers.put("X-Custom-Header", "application/json");

        APIGatewayProxyResponseEvent response = new APIGatewayProxyResponseEvent()
                .withHeaders(headers);
        try {
            final String pageContents = this.getPageContents("https://
checkip.amazonaws.com");
            String output = String.format("{ \"message\": \"hello world\",
 \"location\": \"%s\" }", pageContents);

            return response
                    .withStatusCode(200)
                    .withBody(output);
        } catch (IOException e) {
```

```
            return response
                    .withBody("{}")
                    .withStatusCode(500);
        }
    }
    @Tracing(namespace = "getPageContents")
    private String getPageContents(String address) throws IOException {
        log.info("Retrieving {}", address);
        URL url = new URL(address);
        try (BufferedReader br = new BufferedReader(new
 InputStreamReader(url.openStream()))) {
            return br.lines().collect(Collectors.joining(System.lineSeparator()));
        }
    }
}
```

8. 從 hello-world\src\main\java\com\myorg 目錄中開啟 HelloWorldStack.java，並將現有程式碼替換為下面的程式碼。此程式碼會使用 Lambda 建構函數和 ApiGatewayv2 建構函數來建立 REST API 和 Lambda 函數。

```
package com.myorg;

import software.amazon.awscdk.*;
import software.amazon.awscdk.services.apigatewayv2.alpha.*;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegration;
import
 software.amazon.awscdk.services.apigatewayv2.integrations.alpha.HttpLambdaIntegrationProps
import software.amazon.awscdk.services.lambda.Code;
import software.amazon.awscdk.services.lambda.Function;
import software.amazon.awscdk.services.lambda.FunctionProps;
import software.amazon.awscdk.services.lambda.Runtime;
import software.amazon.awscdk.services.lambda.Tracing;
import software.amazon.awscdk.services.logs.RetentionDays;
import software.amazon.awscdk.services.s3.assets.AssetOptions;
import software.constructs.Construct;

import java.util.Arrays;
import java.util.List;

import static java.util.Collections.singletonList;
import static software.amazon.awscdk.BundlingOutput.ARCHIVED;
```

```
public class HelloWorldStack extends Stack {
    public HelloWorldStack(final Construct scope, final String id) {
        this(scope, id, null);
    }

    public HelloWorldStack(final Construct scope, final String id, final StackProps
 props) {
        super(scope, id, props);

        List<String> functionPackagingInstructions = Arrays.asList(
                "/bin/sh",
                "-c",
                "cd Function " +
                        "&& mvn clean install " +
                        "&& cp /asset-input/Function/target/function.jar /asset-
output/"
        );
        BundlingOptions.Builder builderOptions = BundlingOptions.builder()
                .command(functionPackagingInstructions)
                .image(Runtime.JAVA_11.getBundlingImage())
                .volumes(singletonList(
                        // Mount local .m2 repo to avoid download all the
 dependencies again inside the container
                        DockerVolume.builder()
                                .hostPath(System.getProperty("user.home") +
 "/.m2/")
                                .containerPath("/root/.m2/")
                                .build()
                ))
                .user("root")
                .outputType(ARCHIVED);

        Function function = new Function(this, "Function", FunctionProps.builder()
                .runtime(Runtime.JAVA_11)
                .code(Code.fromAsset("app", AssetOptions.builder()
                        .bundling(builderOptions
                                .command(functionPackagingInstructions)
                                .build())
                        .build()))
                .handler("helloworld.App::handleRequest")
                .memorySize(1024)
                .tracing(Tracing.ACTIVE)
                .timeout(Duration.seconds(10))
                .logRetention(RetentionDays.ONE_WEEK)
```

```
                .build());

        HttpApi httpApi = new HttpApi(this, "sample-api", HttpApiProps.builder()
                .apiName("sample-api")
                .build());

        httpApi.addRoutes(AddRoutesOptions.builder()
                .path("/")
                .methods(singletonList(HttpMethod.GET))
                .integration(new HttpLambdaIntegration("function", function,
 HttpLambdaIntegrationProps.builder()
                        .payloadFormatVersion(PayloadFormatVersion.VERSION_2_0)
                        .build()))
                .build());

        new CfnOutput(this, "HttpApi", CfnOutputProps.builder()
                .description("Url for Http Api")
                .value(httpApi.getApiEndpoint())
                .build());
    }
}
```

9. 從 hello-world 目錄中開啟 pom.xml，並將現有程式碼替換為下面的程式碼。

```
<?xml version="1.0" encoding="UTF-8"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd"
         xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.myorg</groupId>
    <artifactId>hello-world</artifactId>
    <version>0.1</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <cdk.version>2.70.0</cdk.version>
        <constructs.version>[10.0.0,11.0.0)</constructs.version>
        <junit.version>5.7.1</junit.version>
    </properties>

    <build>
        <plugins>
```

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>exec-maven-plugin</artifactId>
                <version>3.0.0</version>
                <configuration>
                    <mainClass>com.myorg.HelloWorldApp</mainClass>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <!-- AWS Cloud Development Kit -->
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>aws-cdk-lib</artifactId>
            <version>${cdk.version}</version>
        </dependency>
        <dependency>
            <groupId>software.constructs</groupId>
            <artifactId>constructs</artifactId>
            <version>${constructs.version}</version>
        </dependency>
        <dependency>
          <groupId>org.junit.jupiter</groupId>
          <artifactId>junit-jupiter</artifactId>
          <version>${junit.version}</version>
          <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>apigatewayv2-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
```

```
        <dependency>
            <groupId>software.amazon.awscdk</groupId>
            <artifactId>apigatewayv2-integrations-alpha</artifactId>
            <version>${cdk.version}-alpha.0</version>
        </dependency>
    </dependencies>
</project>
```

10. 確保您位於 hello-world 目錄中並部署您的應用程式。

```
cdk deploy
```

11. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name HelloWorldStack --query
 'Stacks[0].Outputs[?OutputKey==`HttpApi`].OutputValue' --output text
```

12. 調用 API 端點：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

13. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

```
New XRay Service Graph
  Start time: 2025-02-03 14:59:50+00:00
  End time: 2025-02-03 14:59:50+00:00
  Reference Id: 0 - (Root) AWS::Lambda - sam-app-HelloWorldFunction-YBg8yfYtOc9j -
 Edges: [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
```

```
      - total response time: 0.924
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-YBg8yfYtOc9j
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0.016
  Reference Id: 2 - client - sam-app-HelloWorldFunction-YBg8yfYtOc9j - Edges: [0]
   Summary_statistics:
     - total requests: 0
     - ok count(2XX): 0
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 0

XRay Event [revision 1] at (2025-02-03T14:59:50.204000) with id
 (1-63dd2166-434a12c22e1307ff2114f299) and duration (0.924s)
 - 0.924s - sam-app-HelloWorldFunction-YBg8yfYtOc9j [HTTP: 200]
 - 0.016s - sam-app-HelloWorldFunction-YBg8yfYtOc9j
   - 0.739s - Initialization
   - 0.016s - Invocation
     - 0.013s - ## lambda_handler
       - 0.000s - ## app.hello
   - 0.000s - Overhead
```

14. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
cdk destroy
```

使用 ADOT 來檢測您的 Java 函數

ADOT 提供全受管 Lambda 層，包含使用 OTel SDK 收集遙測資料所需的一切內容。透過取用此層，您可以檢測 Lambda 函數，而無需修改任何函數程式碼。您還可以將層設定為對 OTel 進行自訂初始化。如需詳細資訊，請參閱 ADOT 文件中的針對 Lambda 上的 ADOT 收集器進行自訂組態設定。

對於 Java 執行時間，您可以選擇要取用的兩層：

- AWS ADOT Java （自動檢測代理程式） 的 受管 Lambda 層 – 此層會在啟動時自動轉換函數程式碼，以收集追蹤資料。如需有關如何搭配 ADOT Java 代理程式取用此層的詳細指示，請參閱 ADOT 文件中的適用於 Java 的AWS Distro for OpenTelemetry Lambda 支援 (自動檢測代理程式)。
- AWS 適用於 ADOT Java 的 受管 Lambda 層 – 此層還提供 Lambda 函數的內建檢測，但需要一些手動程式碼變更才能初始化 OTel SDK。如需有關如何取用此層的詳細指示，請參閱 ADOT 文件中的適用於 Java 的AWS Distro for OpenTelemetry Lambda 支援。

使用 X-Ray SDK 來檢測 Java 功能

若要記錄函數對應用程式中其他資源和服務呼叫的資料，請將適用於 Java 的 X-Ray 開發套件新增至您的建置組態。下列範例顯示 Gradle 建置組態，其中包含啟用 AWS SDK for Java 2.x 用戶端自動檢測的程式庫。

Example build.gradle - 追蹤相依性

```
dependencies {
    implementation platform('software.amazon.awssdk:bom:2.16.1')
    implementation platform('com.amazonaws:aws-xray-recorder-sdk-bom:2.11.0')
    ...
    implementation 'com.amazonaws:aws-xray-recorder-sdk-core'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk'
    implementation 'com.amazonaws:aws-xray-recorder-sdk-aws-sdk-v2-instrumentor'
    ...
}
```

新增正確的相依項並完成必要的程式碼變更後，請透過 Lambda 主控台或 API 在函數組態中啟用追蹤功能。

透過 Lambda 主控台來啟用追蹤

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態，然後選擇監控和操作工具。
4. 在其他監控工具欄位中，選擇編輯。

5. 在 CloudWatch Application Signals 和 AWS X-Ray 下，選擇啟用 Lambda 服務追蹤。
6. 選擇儲存。

透過 Lambda API 啟用追蹤

使用 AWS CLI 或 AWS SDK 在 Lambda 函數上設定追蹤，請使用下列 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

下列範例 AWS CLI 命令會在名為 my-function 的函數上啟用主動追蹤。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

使用 啟用追蹤 CloudFormation

若要在 CloudFormation 範本中的 `AWS::Lambda::Function` 資源上啟用追蹤，請使用 `TracingConfig` 屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用 `Tracing` 屬性。

Example template.yml - 追蹤組態

```
Resources:
```

```
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

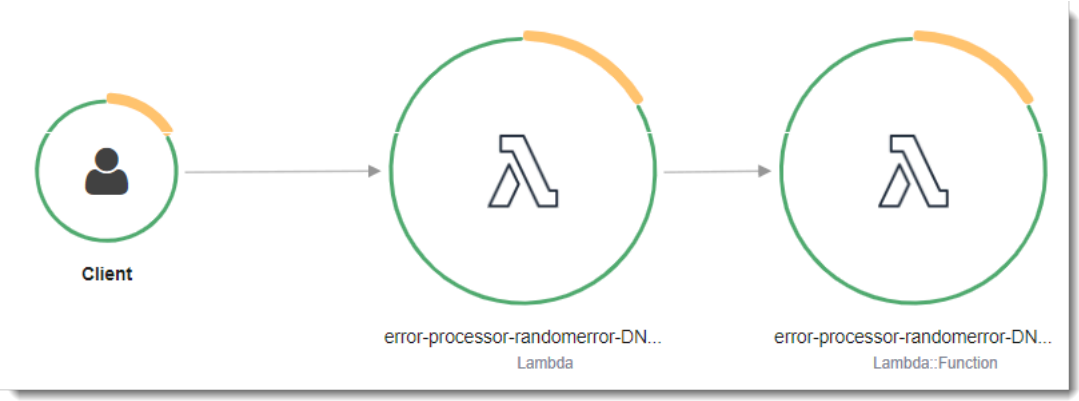
解讀 X-Ray 追蹤

您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將 AWSXRayDaemonWriteAccess 政策新增至執行角色。

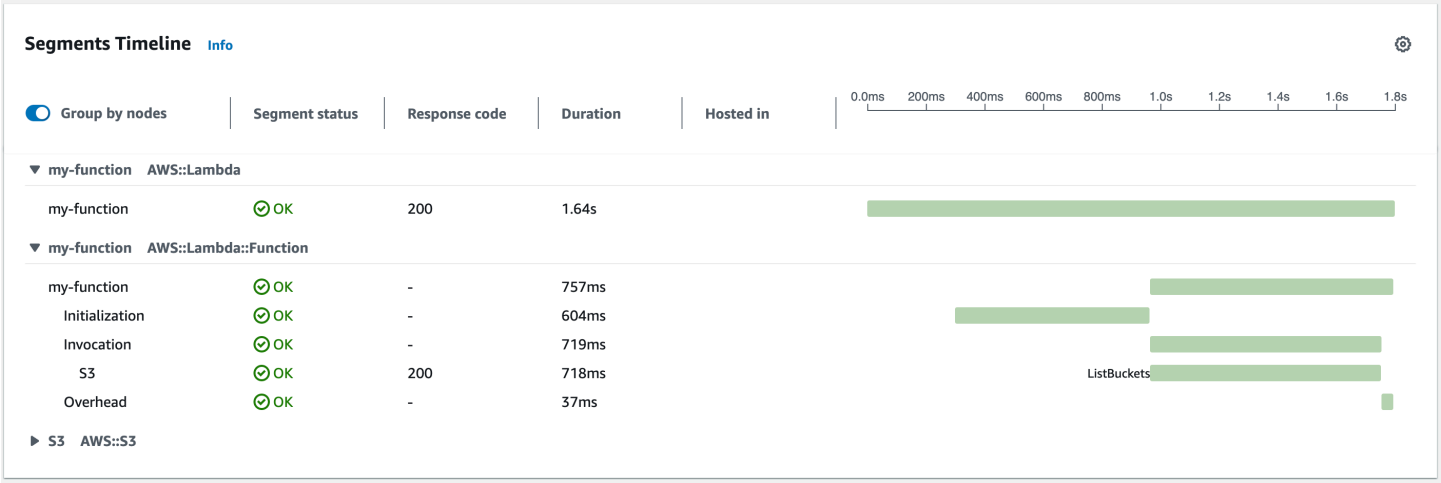
設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 服務圖顯示了有關應用程式及其所有元件的資訊。下列範例顯示了一個具有兩個函數的應用程式。主要函式會處理事件，有時會傳回錯誤。頂端的第二個函數會處理出現在第一個日誌群組中的錯誤，並使用 AWS SDK 呼叫 X-Ray、Amazon Simple Storage Service (Amazon S3) 和 Amazon CloudWatch Logs。

X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。不能針對函數設定 X-Ray 取樣率。

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會在每個追蹤上記錄 2 個區段，這會在服務圖表上建立兩個節點。下圖反白顯示了這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為 my-function，但其中之一的來源為 AWS::Lambda，而另一個的來源為 AWS::Lambda::Function。如果 AWS::Lambda 區段顯示錯誤，Lambda 服務就會出現問題。如果 AWS::Lambda::Function 區段顯示錯誤，表示您的函數出現了問題。



此範例會展開 AWS::Lambda::Function 區段以顯示其三個子區段：

> **Note**
>
> AWS 目前正在對 Lambda 服務實作變更。由於這些變更，您可能會看到系統日誌訊息的結構和內容，與 AWS 帳戶中不同 Lambda 函數發出的追蹤區段之間存在細微差異。
> 此處顯示的追蹤範例說明了舊式函數區段。下列段落說明了舊式和新式區段之間的差異。
> 這些變化將在未來幾週內實作，除中國和 GovCloud 區域以外，所有 AWS 區域 中的所有函數都會轉換至使用新格式的日誌訊息和追蹤區段。

舊式函數區段包含下列子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行期為做好準備以處理下一個事件所花費的時間。

新式函數區段不包含 `Invocation` 子區段。相反地，客戶子區段會直接連接至函數區段。如需舊式和新式函數區段結構的詳細資訊，請參閱the section called “了解 X-Ray 追蹤”。

> **Note**
>
> Lambda SnapStart 函數還包括一個 `Restore` 子區段。`Restore` 子區段會顯示 Lambda 還原快照、載入執行時期和執行任何還原後執行時期勾點所需的時間。還原快照的程序可能包括在 MicroVM 以外的活動上花費的時間。此時間在 `Restore` 子區段中報告。您不需要為在 MicroVM 外還原快照所花費的時間付費。

您也可以檢測 HTTP 用戶端、記錄 SQL 查詢，以及建立具有註釋和中繼資料的自訂子區段。如需詳細資訊，請參閱《AWS X-Ray 開發人員指南》中的 適用於 JAVA 的 AWS X-Ray SDK。

> **定價**
>
> 您可以在每月免費使用 X-Ray 追蹤，最高可達特定限制，作為 AWS 免費方案的一部分。達到閾值後，X-Ray 會收取追蹤儲存及擷取的費用。如需詳細資訊，請參閱 AWS X-Ray 定價。

將執行時間相依項存放存在層中 (X-Ray SDK)

如果您使用 X-Ray 開發套件來檢測函數程式碼的 AWS SDK 用戶端，您的部署套件可能會變得相當大。為了避免每次更新函數程式碼時上傳執行時間相依性，請將 X-Ray SDK 封裝在一個 Lambda 層中。

下面的範例顯示了可存放適用於 Java 的 適用於 Java 的 AWS SDK 和 X-Ray SDK 的 `AWS::Serverless::LayerVersion` 資源。

Example template.yml - 相依性層

```
Resources:
```

```
  function:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: build/distributions/blank-java.zip
      Tracing: Active
      Layers:
        - !Ref libs
      ...
  libs:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: blank-java-lib
      Description: Dependencies for the blank-java sample app.
      ContentUri: build/blank-java-lib.zip
      CompatibleRuntimes:
        - java25
```

透過此組態，您只有在變更執行時間相依性時才會更新程式庫層。由於函數部署套件僅含有您的程式碼，因此有助於減少上傳時間。

為相依性建立層需要建置配置變更，才能在部署之前產生層存檔。如需使用範例，請參閱 GitHub 上的 java-basic 範例應用程式。

樣本應用程式中的 X-Ray 追蹤 (X-Ray SDK)

本指南的 GitHub 儲存庫包含示範 X-Ray 追蹤使用方式的範例應用程式。每個範例應用程式都包含易於部署和清除的指令碼、 AWS SAM 範本和支援資源。

以 Java 編寫的範例 Lambda 應用程式

- example-java – Java 函式，示範如何使用 Lambda 處理訂單。此函數說明如何定義和還原序列化自訂輸入事件物件、使用 AWS SDK 和輸出記錄。
- java-basic - 具有單元測試和變數日誌組態的最小 Java 函數集合。
- java-events - Java 函數集合，其中包含如何處理來自各種服務 (例如 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis) 事件的骨架程式碼。這些函數使用最新版 aws-lambda-java-events 程式庫 (3.0.0 及更新版)。這些範例不需要 AWS SDK 做為相依性。
- s3-java - 一種 Java 函數，它處理來自 Amazon S3 的通知事件，並使用 Java Class Library (JCL) 以從上傳的映像檔案建立縮圖。
- layer-java – 此 Java 函式示範如何使用 Lambda 層，將相依項與核心函式程式碼分離封裝。

所有範例應用程式都已啟用 Lambda 函數的主動追蹤功能。例如，`s3-java`應用程式會顯示 AWS SDK for Java 2.x 用戶端的自動檢測、測試的區段管理、自訂子區段，以及使用 Lambda 層來存放執行時間相依性。

的 Java 範例應用程式 AWS Lambda

本指南的 GitHub 儲存庫提供示範在 中使用 Java 的範例應用程式 AWS Lambda。每個範例應用程式都包含易於部署和清除的指令碼、 CloudFormation 範本和支援資源。

以 Java 編寫的範例 Lambda 應用程式

- example-java – Java 函式，示範如何使用 Lambda 處理訂單。此函數說明如何定義和還原序列化自訂輸入事件物件、使用 AWS SDK 和輸出記錄。
- java-basic - 具有單元測試和變數日誌組態的最小 Java 函數集合。
- java-events - Java 函數集合，其中包含如何處理來自各種服務 (例如 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis) 事件的骨架程式碼。這些函數使用最新版 aws-lambda-java-events 程式庫 (3.0.0 及更新版)。這些範例不需要 AWS SDK 做為相依性。
- s3-java - 一種 Java 函數，它處理來自 Amazon S3 的通知事件，並使用 Java Class Library (JCL) 以從上傳的映像檔案建立縮圖。
- layer-java – 此 Java 函式示範如何使用 Lambda 層，將相依項與核心函式程式碼分離封裝。

在 Lambda 上執行熱門 Java 框架

- spring-cloud-function-samples – 來自 Spring 的範例，說明如何使用 Spring Cloud Function 架構來建立 AWS Lambda 函數。
- Serverless Spring Boot Application 示範 – 此範例展示如何在使用 (或不使用) SnapStart 的受管 Java 執行期中設定 Spring Boot 應用程式，或使用自訂執行期做為 GraalVM 原生映像檔。
- Serverless Micronaut Application 示範 – 此範例展示如何在使用 (或不使用) SnapStart 的受管 Java 執行期中使用 Micronaut，或使用自訂執行期做為 GraalVM 原生映像檔。請參閱《Micronaut/Lambda 指南》以進一步瞭解。
- Serverless Quarkus Application 示範 – 此範例展示如何在使用 (或不使用) SnapStart 的受管 Java 執行期中使用 Quarkus，或使用自訂執行期做為 GraalVM 原生映像檔。請參閱《Quarkus/Lambda 指南》和《Quarkus/SnapStart 指南》以進一步瞭解。

若您不熟悉 Java 中的 Lambda 函數，請從 `java-basic` 範例開始。若要開始使用 Lambda 事件來源，請參閱 `java-events` 範例。這兩個範例集都顯示使用 Lambda 的 Java 程式庫、環境變數、AWS SDK 和 AWS X-Ray SDK。這些範例需要最少的設定，不到一分鐘的時間就可以從命令列完成部署。

使用 Go 建置 Lambda 函數

Go 的實作方式與其他受管執行期不同。由於 Go 程式碼原生編譯至可執行的二進位檔，因此不需要專用語言執行時期。應使用僅限作業系統的執行時期 (provided 執行時期系列) 將 Go 函數部署至 Lambda。

主題

- Go 執行期支援
- 工具與程式庫
- 定義 Go 格式的 Lambda 函數處理常式
- 使用 Lambda 內容物件擷取 Go 函數資訊
- 使用 .zip 封存檔部署 Go Lambda 函數
- 使用容器映像來部署 Go Lambda 函數
- 使用 Go Lambda 函數的層
- 記錄和監控 Go Lambda 函數
- 在 中檢測 Go 程式碼 AWS Lambda

Go 執行期支援

Lambda 的 Go 1.x 受管執行時期已棄用。如果擁有使用 Go 1.x 執行時期的函數，則必須將函數遷移至 provided.al2023 或 provided.al2。provided.al2023 與 相比， 和 provided.al2執行時間提供多種優勢go1.x，包括支援 arm64 架構 (AWS Graviton2 處理器）、較小的二進位檔，以及稍快的調用時間。

本次遷移不需要變更任何程式碼。唯一必須做出的變更與建置部署套件的方式以及用來建立函數的執行期有關。如需詳細資訊，請參閱AWS 運算部落格上的將 AWS Lambda 函數從 Go1.x 執行時間遷移至 Amazon Linux 2 上的自訂執行時間。

Name	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
僅限作業系統的執行期	provided.al2023	Amazon Linux 2023	2029 年 6 月 30 日	2029 年 7 月 31 日	2029 年 8 月 31 日
僅限作業系統的執行期	provided.al2	Amazon Linux 2	2026 年 7 月 31 日	2026 年 8 月 31 日	2026 年 9 月 30 日

工具與程式庫

Lambda 為 Go 執行時間提供以下工具和程式庫：

- 適用於 Go 的 AWS SDK v2：適用於 Go 程式設計語言的官方 AWS SDK。
- github.com/aws/aws-lambda-go/lambda：針對 Go 實作 Lambda 程式設計模型。此套件由 AWS Lambda 用來叫用您的處理常式。
- github.com/aws/aws-lambda-go/lambdacontext：從內容物件取得內容資訊的輔助工具。
- github.com/aws/aws-lambda-go/events：此程式庫提供常用事件來源整合的類型定義。
- github.com/aws/aws-lambda-go/cmd/build-lambda-zip：這個工具可以用來在 Windows 上建立 .zip 檔案封存。

如需詳細資訊，請參閱 GitHub 上的 aws-lambda-go

Lambda 為 Go 執行時間提供下列範例應用程式：

以 Go 編寫的範例 Lambda 應用程式

- go-al2：傳回公有 IP 地址的「hello world」函數。此應用程式使用 `provided.al2` 自訂執行期。
- blank-go – Go 函數，顯示 Lambda Go 程式庫、記錄、環境變數和 AWS SDK 的使用方式。此應用程式使用 `go1.x` 執行期。

定義 Go 格式的 Lambda 函數處理常式

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

此頁面說明如何使用 Go 處理 Lambda 函數處理常式，包括專案設定、命名慣例和最佳實務。本頁還提供了一個 Go Lambda 函式範例，該函式會取得訂單的相關資訊、產生文字檔案收據，並將該檔案放入 Amazon Simple Storage Service (Amazon S3) 儲存貯體。如需編寫函數後如何部署函數的詳細資訊，請參閱the section called “部署 .zip 封存檔”或the section called “部署容器映像”。

主題

- 設定 Go 處理常式專案
- 範例 Go Lambda 函數程式碼
- 處理常式命名慣例
- 定義和存取輸入事件物件
- 存取和使用 Lambda 內容物件
- Go 處理常式的有效處理常式簽章
- 在處理常式中使用 適用於 Go 的 AWS SDK v2
- 存取環境變數
- 使用全域狀態
- Go Lambda 函數的程式碼最佳實務

設定 Go 處理常式專案

以 Go 撰寫的 Lambda 函數被製作成 Go 可執行檔。與初始化任何其他 Go 專案相同，您可以使用以下 `go mod init` 命令初始化一個 Go Lambda 函數專案：

```
go mod init example-go
```

此處 `example-go` 是模組名稱。您可以將其取代為任何值。此命令會初始化專案，並產生列有專案相依項的 `go.mod` 檔案。

使用 `go get` 命令將任何外部相依項新增至專案。例如，對於以 Go 編寫的所有 Lambda 函數程式碼，您需要納入 github.com/aws/aws-lambda-go/lambda 套件，它可針對 Go 實作 Lambda 程式設計模型。使用以下 `go get` 命令納入此套件：

```
go get github.com/aws/aws-lambda-go
```

函數程式碼應該存在於 Go 檔案中。在以下範例中，我們將此檔案命名為 main.go。在此檔案中，您會在處理常式方法中實作核心函數邏輯，以及呼叫此處理常式的 main() 函數。

範例 Go Lambda 函數程式碼

以下範例 Go Lambda 函數程式碼會取得訂單的相關資訊、產生文字檔案收據，並將此檔案放入 Amazon S3 儲存貯體。

Example **main.go** Lambda 函數

```
package main

import (
 "context"
 "encoding/json"
 "fmt"
 "log"
 "os"
 "strings"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

type Order struct {
 OrderID string  `json:"order_id"`
 Amount  float64 `json:"amount"`
 Item    string  `json:"item"`
}

var (
 s3Client *s3.Client
)

func init() {
 // Initialize the S3 client outside of the handler, during the init phase
 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  log.Fatalf("unable to load SDK config, %v", err)
```

```
 }

 s3Client = s3.NewFromConfig(cfg)
}

func uploadReceiptToS3(ctx context.Context, bucketName, key, receiptContent string)
 error {
 _, err := s3Client.PutObject(ctx, &s3.PutObjectInput{
  Bucket: &bucketName,
  Key:    &key,
  Body:   strings.NewReader(receiptContent),
 })
 if err != nil {
  log.Printf("Failed to upload receipt to S3: %v", err)
  return err
 }
 return nil
}

func handleRequest(ctx context.Context, event json.RawMessage) error {
 // Parse the input event
 var order Order
 if err := json.Unmarshal(event, &order); err != nil {
  log.Printf("Failed to unmarshal event: %v", err)
  return err
 }

 // Access environment variables
 bucketName := os.Getenv("RECEIPT_BUCKET")
 if bucketName == "" {
  log.Printf("RECEIPT_BUCKET environment variable is not set")
  return fmt.Errorf("missing required environment variable RECEIPT_BUCKET")
 }

 // Create the receipt content and key destination
 receiptContent := fmt.Sprintf("OrderID: %s\nAmount: $%.2f\nItem: %s",
  order.OrderID, order.Amount, order.Item)
 key := "receipts/" + order.OrderID + ".txt"

 // Upload the receipt to S3 using the helper method
 if err := uploadReceiptToS3(ctx, bucketName, key, receiptContent); err != nil {
  return err
 }
```

```
 log.Printf("Successfully processed order %s and stored receipt in S3 bucket %s",
 order.OrderID, bucketName)
 return nil
}

func main() {
 lambda.Start(handleRequest)
}
```

此 main.go 檔案包含以下程式碼區段：

- package main：在 Go 中，包含 func main() 函數的套件一律要命名為 main。
- import 區塊：用於納入 Lambda 函數所需的程式碼。
- type Order struct {} 區塊：定義此 Go struct 中預期輸入事件的形狀。
- var () 區塊：用於定義將在 Lambda 函數中使用的任何全域變數。
- func init() {}：在此 init() 方法的中包含您希望 Lambda 在初始化階段執行的任何程式碼。
- func uploadReceiptToS3(...) {}：這是主要 handleRequest 處理常式方法所參考的協助程式方法。
- func handleRequest(ctx context.Context, event json.RawMessage) error {}：這是主要處理常式方法，其中包含應用程式的主要邏輯。
- func main() {}：這是 Lambda 處理常式的必要進入點。lambda.Start() 方法的引數是主要處理常式方法。

若要讓此函數正常運作，其執行角色必須允許 s3:PutObject 動作。此外，請確保定義 RECEIPT_BUCKET 環境變數。成功調用後，Amazon S3 儲存貯體應包含收據檔案。

處理常式命名慣例

對於以 Go 編寫的 Lambda 函數，處理常式的命名不受限制。在此範例中，處理常式方法的名稱為 handleRequest。若要參考程式碼中的處理常式值，您可以使用 _HANDLER 環境變數。

對於使用 .zip 部署套件部署的 Go 函數，含有函數程式碼的可執行檔必須命名為 bootstrap。此外，bootstrap 檔案必須位於 .zip 檔案的根層級。對於使用容器映像部署的 Go 函數，可執行檔的名稱不受限制。

定義和存取輸入事件物件

JSON 是 Lambda 函數最常見的標準輸入格式。在此範例中，函數預期輸入類似以下內容：

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

使用以 Go 編寫的 Lambda 函數時，您可以將預期輸入事件的形狀定義為 Go struct。在此範例中，我們透過定義 struct 來代表 `Order`：

```
type Order struct {
    OrderID string  `json:"order_id"`
    Amount  float64 `json:"amount"`
    Item    string  `json:"item"`
}
```

此 struct 符合預期的輸入形狀。定義 struct 之後，可以撰寫處理常式簽章，該簽章採用與 encoding/json standard library 相容的一般 JSON 類型。然後，可以使用 func Unmarshal 函數將其還原序列化至該 struct。這在處理常式的前幾行中得以曾現：

```
func handleRequest(ctx context.Context, event json.RawMessage) error {
    // Parse the input event
    var order Order
    if err := json.Unmarshal(event, &order); err != nil {
        log.Printf("Failed to unmarshal event: %v", err)
        return err
    ...
}
```

在此還原序列化操作之後，您可以存取 `order` 變數的欄位。例如，使用 `"order_id"` 可以從原始輸入擷取 `order.OrderID` 的值。

> **Note**
>
> `encoding/json` 套件只能存取匯出的欄位。若要匯出，事件結構中的欄位名稱必須大寫。

存取和使用 Lambda 內容物件

Lambda 內容物件 包含有關調用、函數以及執行環境的資訊。在此範例中，我們在處理常式簽章中將此變數宣告為 `ctx`：

```
func handleRequest(ctx context.Context, event json.RawMessage) error {
    ...
}
```

ctx context.Context 輸入是函數處理常式的一個選用引數。如需可接受處理常式簽章的詳細資訊，請參閱the section called “Go 處理常式的有效處理常式簽章”。

如果您使用 AWS SDK 呼叫其他服務，則有幾個關鍵區域需要內容物件。例如，您可以使用內容物件載入正確的 AWS SDK 組態，以正確初始化 SDK 用戶端。如下所示：

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
```

SDK 呼叫本身可能需要內容物件做為輸入。例如，s3Client.PutObject 呼叫接受內容物件做為其第一個引數：

```
// Upload the receipt to S3
    _, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
        ...
    })
```

除了 AWS SDK 請求之外，您也可以將內容物件用於函數監控。如需內容物件的詳細資訊，請參閱the section called “Context”。

Go 處理常式的有效處理常式簽章

當您以 Go 建置 Lambda 函數處理常式時，會有數個選項，但務必堅守以下規則：

- 處理常式必須是一個函式。
- 處理常式採用 0 到 2 之間的引數。如果有兩個引數，第一個引數必須實作 context.Context。
- 處理常式傳回 0 到 2 之間的引數。若有單一傳回值，它必須實作 error。如果有兩個傳回值，第二個值必須實作 error。

下方將列出有效的處理常式簽章。TIn 和 TOut 皆代表與 encoding/json 標準程式庫相容的類型。如需詳細資訊，請參閱 func Unmarshal 以了解這些類型如何還原序列化。

-
```
func ()
```

- `func () error`
- `func () (TOut, error)`
- `func (TIn) error`
- `func (TIn) (TOut, error)`
- `func (context.Context) error`
- `func (context.Context) (TOut, error)`
- `func (context.Context, TIn) error`
- `func (context.Context, TIn) (TOut, error)`

在處理常式中使用 適用於 Go 的 AWS SDK v2

通常，您會使用 Lambda 函數與其他 AWS 資源互動或進行更新。與這些資源互動的最簡單方法便是使用 適用於 Go 的 AWS SDK v2。

> **Note**
>
> 適用於 Go 的 AWS SDK (v1) 目前處於維護模式，並且將於 2025 年 7 月 31 日終止支援。我們建議您在之後僅使用 適用於 Go 的 AWS SDK v2。

若要將 SDK 相依項新增至函數，請針對您需要的特定 SDK 用戶端使用 `go get` 命令。在前面的範例程式碼中，我們使用了 `config` 程式庫和 `s3` 程式庫。請在包含 `go.mod` 和 `main.go` 檔案的目錄中執行下列命令，來新增這些相依項：

```
go get github.com/aws/aws-sdk-go-v2/config
go get github.com/aws/aws-sdk-go-v2/service/s3
```

然後，在函數的 import 區塊中相應地匯入相依項：

```
import (
```

```
    ...
    "github.com/aws/aws-sdk-go-v2/config"
    "github.com/aws/aws-sdk-go-v2/service/s3"
)
```

在處理常式中使用 SDK 時，須對用戶端進行正確設定。最簡單的方法是使用預設憑證提供者鏈。此範例說明了載入此組態的其中一種方式：

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
    if err != nil {
        log.Printf("Failed to load AWS SDK config: %v", err)
        return err
    }
```

將此組態載入 `cfg` 變數後，可以將此變數傳遞至用戶端執行個體。範例程式碼會執行個體化一個 Amazon S3 用戶端，如下所示：

```
// Create an S3 client
    s3Client := s3.NewFromConfig(cfg)
```

在此範例中，我們在 `init()` 函數中初始化 Amazon S3 用戶端，以避免每次調用函數時都必須初始化它。問題是，在 `init()` 函數中，Lambda 無法存取內容物件。為解決此問題，您可以在初始化階段傳遞類似 `context.TODO()` 這樣的預留位置。稍後，當您使用用戶端進行呼叫時，再傳遞完整的內容物件。the section called “在 AWS SDK 用戶端初始化和呼叫中使用內容”對此解決方法也有所說明。

設定並初始化 SDK 用戶端之後，您就可以使用它與其他 AWS 服務互動。此範例程式碼會呼叫 Amazon S3 `PutObject` API，如下所示：

```
_, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
    Bucket: &bucketName,
    Key:    &key,
    Body:   strings.NewReader(receiptContent),
})
```

存取環境變數

在處理常式程式碼中，您可以使用 `os.Getenv()` 方法參考任何環境變數。在此範例中，我們使用以下程式碼來參考定義的 `RECEIPT_BUCKET` 環境變數：

```
// Access environment variables
    bucketName := os.Getenv("RECEIPT_BUCKET")
    if bucketName == "" {
        log.Printf("RECEIPT_BUCKET environment variable is not set")
        return fmt.Errorf("missing required environment variable RECEIPT_BUCKET")
    }
```

使用全域狀態

為了避免每次調用函數時建立新的資源，您可以在處理常式程式碼之外宣告和修改 Lambda 函數的全域變數。您可以在 `var` 區塊或陳述式中定義這些全域變數。此外，處理常式可能會宣告在初始化階段執行的 `init()` 函數。在 AWS Lambda 中，`init` 方法的行為與標準的 Go 程式相同。

Go Lambda 函數的程式碼最佳實務

請遵循下列清單中的準則，在建置 Lambda 函數時使用最佳編碼實務：

- 區隔 Lambda 處理常式與您的核心邏輯。能允許您製作更多可測單位的函式。
- 最小化依存項目的複雜性。偏好更簡易的框架，其可快速在執行環境啟動時載入。
- 將部署套件最小化至執行時期所必要的套件大小。這能減少您的部署套件被下載與呼叫前解壓縮的時間。

請利用執行環境重新使用來改看函式的效能。在函式處理常式之外初始化 SDK 用戶端和資料庫連線，並在本機快取 `/tmp` 目錄中的靜態資產。由您函式的相同執行個體處理的後續叫用可以重複使用這些資源。這可藉由減少函數執行時間來節省成本。

若要避免叫用間洩漏潛在資料，請不要使用執行環境來儲存使用者資料、事件，或其他牽涉安全性的資訊。如果您的函式依賴無法存放在處理常式內記憶體中的可變狀態，請考慮為每個使用者建立個別函式或個別函式版本。

使用 Keep-Alive 指令維持持續連線的狀態。Lambda 會隨著時間的推移清除閒置連線。叫用函數時嘗試重複使用閒置連線將導致連線錯誤。若要維護持續連線，請使用與執行階段相關聯的 keep-alive (保持啟用) 指令。如需範例，請參閱在 Node.js 中重複使用 Keep-Alive 的連線。

使用環境變數將操作參數傳遞給您的函數。例如，如果您正在寫入到 Amazon S3 儲存貯體，而非對您正在寫入的儲存貯體名稱進行硬式編碼，請將儲存貯體名稱設定為環境變數。

避免在 Lambda 函數中使用遞迴調用，其中函數會調用自己或啟動可能再次調用函數的程序。這會導致意外的函式呼叫量與升高的成本。若您看到意外的調用數量，當更新程式碼時，請立刻將函數的預留並行設為 0，以調節對函數的所有調用。

請勿在您的 Lambda 函數程式碼中使用未記錄的非公有 API。對於 AWS Lambda 受管執行時間，Lambda 會定期將安全性和函數更新套用至 Lambda 的內部 API。這些內部 API 更新可能是向後不相容的，這會導致意外結果，例如若您的函數依賴於這些非公有 API，則叫用失敗。請參閱 API 參考查看公開可用 API 的清單。

撰寫等冪程式碼。為函數撰寫等冪程式碼可確保採用相同方式來處理重複事件。程式碼應正確驗證事件並正常處理重複的事件。如需詳細資訊，請參閱 How do I make my Lambda function idempotent? (如何讓 Lambda 函數等冪？)。

使用 Lambda 內容物件擷取 Go 函數資訊

在 Lambda 中，內容物件提供的方法和各項屬性包含了有關調用、函數以及執行環境的資訊。當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。若要在處理常式中使用內容物件，您可以選擇將其宣告為處理常式的輸入參數。如果您想要在處理常式中執行以下動作，則需要使用內容物件：

- 您需要存取內容物件提供的任何全域變數、方法或屬性。這些方法和屬性對於諸如判斷調用函數的實體或測量函數的調用時間等任務很有用，如the section called “存取叫用內容資訊”所述。
- 您需要使用 適用於 Go 的 AWS SDK 來呼叫其他服務。對於大多數這樣的呼叫，內容物件是重要的輸入參數。如需更多詳細資訊，請參閱 the section called “在 AWS SDK 用戶端初始化和呼叫中使用內容”。

主題

- 內容物件中支援的變數、方法和屬性
- 存取叫用內容資訊
- 在 AWS SDK 用戶端初始化和呼叫中使用內容

內容物件中支援的變數、方法和屬性

Lambda 內容程式庫提供下列全域變數、方法和屬性。

全域變數

- `FunctionName` – Lambda 函數的名稱。
- `FunctionVersion` – 函數的版本。
- `MemoryLimitInMB` - 分配給函數的記憶體數量。
- `LogGroupName` - 函數的日誌群組。
- `LogStreamName` - 函數執行個體的記錄串流。

內容方法

- `Deadline` - 傳回執行逾時的日期，以 Unix 時間毫秒為單位。

內容屬性

- `InvokedFunctionArn` - 用於叫用此函數的 Amazon Resource Name (ARN)。指出叫用者是否指定版本號或別名。
- `AwsRequestID` - 叫用請求的識別符。
- `Identity` - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
- `ClientContext` - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。

存取叫用內容資訊

Lambda 函數有權存取有關其環境和叫用請求的中繼資料。這可以在套件內容進行存取。如果處理常式將 `context.Context` 納為參數，Lambda 即會將函數的相關資訊插入至內容的 `Value` 屬性。請注意，您需要匯入 `lambdacontext` 程式庫以存取 `context.Context` 物件的內容。

```
package main

import (
        "context"
        "log"
        "github.com/aws/aws-lambda-go/lambda"
        "github.com/aws/aws-lambda-go/lambdacontext"
)

func CognitoHandler(ctx context.Context) {
        lc, _ := lambdacontext.FromContext(ctx)
        log.Print(lc.Identity.CognitoIdentityPoolID)
}

func main() {
        lambda.Start(CognitoHandler)
}
```

在上面的範例中，`lc` 是用於取用內容物件擷取之資訊的變數，而 `log.Print(lc.Identity.CognitoIdentityPoolID)` 則會列印該資訊，在此例為 CognitoIdentityPoolID。

以下範例說明如何使用內容物件來監控執行 Lambda 函數的時間長度。這可讓您分析效能期望值，並在必要時據以調整您的函式程式碼。

```
package main
```

```
import (
        "context"
        "log"
        "time"
        "github.com/aws/aws-lambda-go/lambda"
)

func LongRunningHandler(ctx context.Context) (string, error) {

        deadline, _ := ctx.Deadline()
        deadline = deadline.Add(-100 * time.Millisecond)
        timeoutChannel := time.After(time.Until(deadline))

        for {

                select {

                case <- timeoutChannel:
                        return "Finished before timing out.", nil

                default:
                        log.Print("hello!")
                        time.Sleep(50 * time.Millisecond)
                }
        }
}

func main() {
        lambda.Start(LongRunningHandler)
}
```

在 AWS SDK 用戶端初始化和呼叫中使用內容

如果處理常式需要使用 適用於 Go 的 AWS SDK 來呼叫其他服務，請將內容物件做為處理常式的輸入。在 AWS 中，最佳實務是在大多數 AWS SDK 呼叫中傳遞內容物件。例如，Amazon S3 `PutObject` 呼叫接受內容物件 (`ctx`) 做為其第一個引數：

```
// Upload an object to S3
    _, err = s3Client.PutObject(ctx, &s3.PutObjectInput{
        ...
    })
```

若要正確初始化 SDK 用戶端，您也可以使用內容物件載入正確的組態，然後再將該組態物件傳遞給用戶端：

```
// Load AWS SDK configuration using the default credential provider chain
    cfg, err := config.LoadDefaultConfig(ctx)
    ...
    s3Client = s3.NewFromConfig(cfg)
```

如果您想要在主要處理常式之外初始化 SDK 用戶端 (即在初始化階段)，您可以傳遞預留位置內容物件：

```
func init() {
 // Initialize the S3 client outside of the handler, during the init phase
 cfg, err := config.LoadDefaultConfig(context.TODO())
 ...
 s3Client = s3.NewFromConfig(cfg)
}
```

如果您以這種方式初始化用戶端，請確保您從主要處理常式在 SDK 呼叫中傳遞正確的內容物件。

使用 .zip 封存檔部署 Go Lambda 函數

AWS Lambda 函數的程式碼包含指令碼或編譯的程式及其相依性。使用部署套件將函數程式碼部署到 Lambda。Lambda 支援兩種類型的部署套件：容器映像和 .zip 封存檔。

此頁面說明如何建立 .zip 檔案做為 Go 執行時間的部署套件，然後使用 .zip 檔案， AWS Lambda 使用 AWS 管理主控台、 AWS Command Line Interface (AWS CLI) 和 AWS Serverless Application Model () 將函數程式碼部署到AWS SAM。

請注意，Lambda 使用 POSIX 檔案許可，因此在建立 .zip 封存檔之前，您可能需要設定部署套件資料夾的許可。

章節

- 在 macOS 和 Linux 上建立 .zip 檔案
- 在 Windows 上建立 .zip 檔案
- 使用 .zip 檔案建立及更新 Go Lambda 函數

在 macOS 和 Linux 上建立 .zip 檔案

下列步驟會說明如何使用 `go build` 命令編譯可執行檔，以及如何為 Lambda 建立 .zip 檔案部署套件。在編譯程式碼之前，請確認已從 GitHub 安裝 lambda 套件。此模組會讓您實作執行期界面，管理 Lambda 與函數程式碼之間的互動。若要下載此程式庫，請執行下列命令。

```
go get github.com/aws/aws-lambda-go/lambda
```

如果您的函數使用 適用於 Go 的 AWS SDK，請下載標準 SDK 模組集，以及應用程式所需的任何 AWS 服務 API 用戶端。若要了解如何安裝適用於 Go 的 SDK，請參閱 適用於 Go 的 AWS SDK V2 入門。

使用提供的執行時期系列

Go 的實作方式與其他受管執行期不同。由於 Go 程式碼原生編譯至可執行的二進位檔，因此不需要專用語言執行時期。應使用僅限作業系統的執行時期 (`provided` 執行時期系列) 將 Go 函數部署至 Lambda。

建立 .zip 部署套件 (macOS/Linux) 的方式

1. 在含有應用程式 `main.go` 檔案的專案目錄中編譯可執行檔。注意下列事項：

- 可執行檔必須命名為 `bootstrap`。如需詳細資訊，請參閱處理常式命名慣例。
- 設定您的目標 指令集架構。僅限作業系統的執行時期同時支援 arm64 和 x86_64。
- 您可以使用選用 `lambda.norpc` 標籤，來排除 lambda 程式庫的遠端程序呼叫 (RPC) 元件。使用已棄用的 Go 1.x 執行時期時，才需要 RPC 元件。排除 RPC 會縮減部署套件的大小。

若是 arm64 架構：

```
GOOS=linux GOARCH=arm64 go build -tags lambda.norpc -o bootstrap main.go
```

若是 x86_64 架構：

```
GOOS=linux GOARCH=amd64 go build -tags lambda.norpc -o bootstrap main.go
```

2. (可選) 您可能需要用 Linux 上的 `CGO_ENABLED=0` 設定編譯套件：

```
GOOS=linux GOARCH=arm64 CGO_ENABLED=0 go build -o bootstrap -tags lambda.norpc
 main.go
```

此命令為標準 C 程式庫 (`libc`) 版本建立了一個穩定的二進位套件，這在 Lambda 和其他設備上可能不同。

3. 透過將可執行檔封裝在 .zip 檔案中建立部署套件。

```
zip myFunction.zip bootstrap
```

> **Note**
>
> `bootstrap` 檔案必須位於 .zip 檔案的根層級。

4. 建立函數。注意下列事項：

- 二進位檔必須命名為 `bootstrap`，但處理常式名稱可以是任何名稱。如需詳細資訊，請參閱處理常式命名慣例。
- 如果您使用的是 arm64，才需要選擇 `--architectures` 選項。預設值為 x86_64。
- 針對 `--role`，請指定 執行角色 的 Amazon Resource Name (ARN)。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

在 Windows 上建立 .zip 檔案

下列步驟會說明從 GitHub 下載適用於 Windows 的 build-lambda-zip 工具、編譯可執行檔，以及建立 .zip 部署套件的方式。

> **Note**
>
> 如果尚未這樣做，您必須安裝 git，然後將 `git` 可執行檔新增到 Windows %PATH% 環境變數。

在編譯程式碼之前，請確認已從 GitHub 安裝 lambda 程式庫。若要下載此程式庫，請執行下列命令。

```
go get github.com/aws/aws-lambda-go/lambda
```

如果您的函數使用 適用於 Go 的 AWS SDK，請下載標準 SDK 模組集，以及應用程式所需的任何 AWS 服務 API 用戶端。若要了解如何安裝適用於 Go 的 SDK，請參閱 適用於 Go 的 AWS SDK V2 入門。

使用提供的執行時期系列

Go 的實作方式與其他受管執行期不同。由於 Go 程式碼原生編譯至可執行的二進位檔，因此不需要專用語言執行時期。應使用僅限作業系統的執行時期 (`provided` 執行時期系列) 將 Go 函數部署至 Lambda。

建立 .zip 部署套件的方式 (Windows)

1. 從 GitHub 下載 build-lambda-zip 工具。

```
go install github.com/aws/aws-lambda-go/cmd/build-lambda-zip@latest
```

2. 使用您的 GOPATH 工具來建立 .zip 檔案。如果您已預設安裝 Go，則該工具通常位於 %USERPROFILE%\Go\bin 中。否則，導覽至您安裝 Go 執行期之處，然後執行下列其中一個動作：

   cmd.exe

   在 cmd.exe 中執行下列其中一項 (視您的目標 指令集架構 而定)。僅限作業系統的執行時期同時支援 arm64 和 x86_64。

   您可以使用選用 `lambda.norpc` 標籤，來排除 lambda 程式庫的遠端程序呼叫 (RPC) 元件。使用已棄用的 Go 1.x 執行時期時，才需要 RPC 元件。排除 RPC 會縮減部署套件的大小。

   Example- 適用 x86_64 架構

   ```
   set GOOS=linux
   set GOARCH=amd64
   set CGO_ENABLED=0
   go build -tags lambda.norpc -o bootstrap main.go
   %USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
   ```

   Example- 適用 arm64 架構

   ```
   set GOOS=linux
   set GOARCH=arm64
   set CGO_ENABLED=0
   go build -tags lambda.norpc -o bootstrap main.go
   %USERPROFILE%\Go\bin\build-lambda-zip.exe -o myFunction.zip bootstrap
   ```

   PowerShell

   在 PowerShell 中執行下列其中一項 (視您的目標 指令集架構 而定)。僅限作業系統的執行時期同時支援 arm64 和 x86_64。

   您可以使用選用 `lambda.norpc` 標籤，來排除 lambda 程式庫的遠端程序呼叫 (RPC) 元件。使用已棄用的 Go 1.x 執行時期時，才需要 RPC 元件。排除 RPC 會縮減部署套件的大小。

   若是 x86_64 架構：

   ```
   $env:GOOS = "linux"
   $env:GOARCH = "amd64"
   $env:CGO_ENABLED = "0"
   ```

```
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

若是 arm64 架構：

```
$env:GOOS = "linux"
$env:GOARCH = "arm64"
$env:CGO_ENABLED = "0"
go build -tags lambda.norpc -o bootstrap main.go
~\Go\Bin\build-lambda-zip.exe -o myFunction.zip bootstrap
```

3. 建立函數。注意下列事項：

   - 二進位檔必須命名為 `bootstrap`，但處理常式名稱可以是任何名稱。如需詳細資訊，請參閱處理常式命名慣例。
   - 如果您使用的是 arm64，才需要選擇 `--architectures` 選項。預設值為 x86_64。
   - 針對 `--role`，請指定 執行角色 的 Amazon Resource Name (ARN)。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--architectures arm64 \
--role arn:aws:iam::111122223333:role/lambda-ex \
--zip-file fileb://myFunction.zip
```

使用 .zip 檔案建立及更新 Go Lambda 函數

建立 .zip 部署套件後，您可以使用該套件建立新的 Lambda 函數或更新現有函數。您可以使用 Lambda 主控台、 AWS Command Line Interface和 Lambda API 部署 .zip 套件。您也可以使用 AWS Serverless Application Model (AWS SAM) 和 CloudFormation 建立並更新 Lambda 函數。

Lambda 的 .zip 部署套件大小上限為 250 MB (解壓縮)。請注意，此限制適用於您上傳的所有檔案 (包括任何 Lambda 層) 的大小總和。

Lambda 執行時間需有許可才能讀取部署套裝服務中的檔案。在 Linux 許可八進位標記法中，Lambda 需要 644 個許可 (rw-r--r--) 用於非可執行檔，以及 755 個許可 (rwxr-x) 用於目錄和可執行檔。

在 Linux 和 MacOS 中，使用 chmod 命令變更部署套件中檔案和目錄的檔案許可。例如，若要為非可執行檔提供正確的許可，請執行下列命令。

```
chmod 644 <filepath>
```

若要在 Windows 中變更檔案許可，請參閱 Microsoft Windows 文件的 Set, View, Change, or Remove Permissions on an Object。

> **Note**
>
> 如果您未授予 Lambda 存取部署套件中的目錄所需的許可，Lambda 會將這些目錄的許可設定為 755 (rwxr-xr-x)。

透過主控台使用 .zip 檔案建立及更新函數

若要建立新函數，您必須先在主控台中建立函數，然後上傳您的 .zip 封存檔。若要更新現有函數，請開啟函數的頁面，然後按照同樣的程序新增更新後的 .zip 檔案。

如果您的 .zip 檔案小於 50 MB，您可以透過直接從本機電腦上傳檔案來建立或更新函數。若 .zip 檔案大於 50 MB，您必須先將套件上傳至 Amazon S3 儲存貯體。如需如何使用 將檔案上傳至 Amazon S3 儲存貯體的指示 AWS 管理主控台，請參閱 Amazon S3 入門。若要使用 上傳檔案 AWS CLI，請參閱AWS CLI 《 使用者指南》中的移動物件。

> **Note**
>
> 您無法轉換現有的容器映像函數以使用 .zip 封存檔。您必須建立新的函數。

若要建立新的函數 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇建立函數。
2. 選擇 Author from scratch (從頭開始撰寫)。
3. 在基本資訊下，請執行下列動作：
   a. 在函數名稱中輸入函數名稱。
   b. 對於 Runtime (執行時間)，選擇 `provided.al2023`。
4. (選用) 在許可下，展開變更預設執行角色。您可建立新的執行角色，或使用現有的角色。
5. 選擇建立函數。Lambda 會使用您選擇的執行期建立一個基本的「Hello world」函數。

若要從本機電腦上傳 .zip 封存檔 (主控台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 .zip 檔案。
5. 若要上傳 .zip 檔案，請執行下列操作：
   a. 選擇上傳，然後在檔案選擇器中選取您的 .zip 檔案。
   b. 選擇 Open (開啟)。
   c. 選擇儲存。

若要從 Amazon S3 儲存貯體上傳 .zip 封存檔 (控制台)

1. 在 Lambda 主控台的函數頁面中選擇要上傳新 .zip 檔案的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中選擇上傳來源。
4. 選擇 Amazon S3 位置。
5. 貼上 .zip 檔案的 Amazon S3 連結 URL，然後選擇儲存。

使用 建立和更新具有 .zip 檔案的函數 AWS CLI

您可以使用 AWS CLI 建立新函數，或使用 .zip 檔案更新現有函數。使用 create-function 和 update-function-code 命令來部署您的 .zip 套件。如果您的 .zip 檔案小於 50 MB，則可以從本機建置電腦的檔案位置上傳 .zip 套件。若檔案較大，則必須先從 Amazon S3 儲存貯體上傳 .zip 套件。如需如何使用將檔案上傳至 Amazon S3 儲存貯體的指示 AWS CLI，請參閱AWS CLI 《 使用者指南》中的移動物件。

> **Note**
>
> 如果您使用 從 Amazon S3 儲存貯體上傳 .zip 檔案 AWS CLI，儲存貯體必須與 AWS 區域 函數位於相同的 中。

若要搭配 使用 .zip 檔案建立新的函數 AWS CLI，您必須指定下列項目：

- 函數名稱 (--function-name)
- 函數的執行期 (--runtime)
- 函數執行角色的 Amazon Resource Name (ARN) (--role)
- 函數程式碼中處理常式方法的名稱 (--handler)

您也必須指定 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 --zip-file 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 --code 選項。您只需針對版本控制的物件使用 S3ObjectVersion 參數。

```
aws lambda create-function --function-name myFunction \
--runtime provided.al2023 --handler bootstrap \
--role arn:aws:iam::111122223333:role/service-role/my-lambda-role \
--code S3Bucket=amzn-s3-demo-
bucket,S3Key=myFileName.zip,S3ObjectVersion=myObjectVersion
```

若要使用 CLI 更新現有函數，您可以使用 --function-name 參數指定函數的名稱。您也必須指定要用來更新函數程式碼的 .zip 檔案的位置。如果您的 .zip 檔案位於本機建置電腦上的資料夾中，請使用 --zip-file 選項來指定檔案路徑，如下列範例命令所示。

```
aws lambda update-function-code --function-name myFunction \
--zip-file fileb://myFunction.zip
```

若要在 Amazon S3 儲存貯體中指定 .zip 檔案的位置，請使用如下列範例命令所示的 --s3-bucket 和 --s3-key 選項。您只需針對版本控制的物件使用 --s3-object-version 參數。

```
aws lambda update-function-code --function-name myFunction \
--s3-bucket amzn-s3-demo-bucket --s3-key myFileName.zip --s3-object-version myObject
 Version
```

透過 Lambda API 使用 .zip 檔案建立及更新函數

若要使用 .zip 封存檔建立及更新函數，請使用下列 API 操作：

- CreateFunction
- UpdateFunctionCode

使用 建立和更新具有 .zip 檔案的函數 AWS SAM

AWS Serverless Application Model (AWS SAM) 是一種工具組，可協助簡化建置和執行無伺服器應用程式的程序 AWS。您可以在 YAML 或 JSON 範本中定義應用程式的資源，並使用 AWS SAM 命令列界面 (AWS SAM CLI) 來建置、封裝和部署應用程式。當您從 AWS SAM 範本建置 Lambda 函數時，AWS SAM 會自動使用函數程式碼和您指定的任何相依性建立 .zip 部署套件或容器映像。若要進一步了解如何使用 AWS SAM 來建置和部署 Lambda 函數，請參閱《 AWS Serverless Application Model 開發人員指南》中的 入門 AWS SAM。

您也可以使用 AWS SAM 來建立使用現有 .zip 檔案封存的 Lambda 函數。若要使用 建立 Lambda 函數 AWS SAM，您可以將 .zip 檔案儲存在 Amazon S3 儲存貯體或建置機器的本機資料夾中。如需如何使用 將檔案上傳至 Amazon S3 儲存貯體的指示 AWS CLI，請參閱AWS CLI 《 使用者指南》中的移動物件。

在您的 AWS SAM 範本中，`AWS::Serverless::Function`資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- `PackageType`：設定為 `Zip`
- `CodeUri`：設定為函數程式碼的 Amazon S3 URI、本機資料夾的路徑或 FunctionCode 物件
- `Runtime`：設定為所選執行期

如果您的 AWS SAM.zip 檔案大於 50MB，則不需要先將其上傳至 Amazon S3 儲存貯體。 AWS SAM 可以從本機建置機器的位置上傳最大允許大小為 250MB （解壓縮） 的 .zip 套件。

若要進一步了解如何在 中使用 .zip 檔案部署函數 AWS SAM，請參閱《 AWS SAM 開發人員指南》中的 AWS::Serverless::Function。

範例：使用 搭配 provided.al2023 AWS SAM 建置 Go 函數

1. 建立具有下列屬性的 AWS SAM 範本：
    - BuildMethod：指定應用程式的編譯器。請使用 `go1.x`。
    - Runtime：使用 `provided.al2023`。
    - CodeUri：輸入程式碼的路徑。

- Architectures：為 arm64 架構使用 [arm64]。若是 x86_64 指令集架構，請使用 [amd64] 或移除 Architectures 屬性。

Example template.yaml

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Metadata:
      BuildMethod: go1.x
    Properties:
      CodeUri: hello-world/ # folder where your main program resides
      Handler: bootstrap
      Runtime: provided.al2023
      Architectures: [arm64]
```

2. 使用 sam build 命令來編譯可執行檔。

```
sam build
```

3. 使用 sam deploy 命令將函數部署到 Lambda。

```
sam deploy --guided
```

使用 建立和更新具有 .zip 檔案的函數 CloudFormation

您可以使用 CloudFormation 建立使用 .zip 檔案封存的 Lambda 函數。若要使用 .zip 檔案建立 Lambda 函數，您必須先將檔案上傳至 Amazon S3 儲存貯體。如需如何使用 將檔案上傳至 Amazon S3 儲存貯體的指示 AWS CLI，請參閱《 使用者指南》中的移動物件。 AWS CLI

在您的 CloudFormation 範本中，AWS::Lambda::Function資源會指定您的 Lambda 函數。在本資源中設定下列屬性，以使用 .zip 封存檔建立函數：

- PackageType：設定為 Zip
- Code：在 S3Bucket 和 S3Key 欄位中輸入 Amazon S3 儲存貯體名稱和 .zip 檔案名稱。
- Runtime：設定為所選執行期

CloudFormation 產生的 .zip 檔案不能超過 4MB。若要進一步了解如何在 中使用 .zip 檔案部署函數 CloudFormation，請參閱CloudFormation 《 使用者指南》中的 AWS::Lambda::Function。

使用容器映像來部署 Go Lambda 函數

您可以透過兩種方式為 Go Lambda 函數建置容器映像：

- 使用僅限 AWS 作業系統的基礎映像

  Go 的實作方式與其他受管執行期不同。由於 Go 程式碼原生編譯至可執行的二進位檔，因此不需要專用語言執行時期。使用僅限作業系統的基礎映像來建置 Lambda 的 Go 映像。若要使映像檔與 Lambda 相容，您必須在映像中加入 `aws-lambda-go/lambda` 套件。
- 使用非AWS 基礎映像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像檔與 Lambda 相容，您必須在映像中加入 `aws-lambda-go/lambda` 套件。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

本頁面會說明如何為 Lambda 建置、測試和部署容器映像。

AWS 部署 Go 函數的基礎映像

Go 的實作方式與其他受管執行期不同。由於 Go 程式碼原生編譯至可執行的二進位檔，因此不需要專用語言執行時期。使用僅限作業系統的基礎映像，將 Go 函數部署至 Lambda。

Name	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
僅限作業系統的執行期	`provided.al2023`	Amazon Linux 2023	2029 年 6 月 30 日	2029 年 7 月 31 日	2029 年 8 月 31 日
僅限作業系統的執行期	`provided.al2`	Amazon Linux 2	2026 年 7 月 31 日	2026 年 8 月 31 日	2026 年 9 月 30 日

Amazon Elastic Container Registry 公有資源庫：gallery.ecr.aws/lambda/provided

Go 執行期介面用戶端

此 aws-lambda-go/lambda 套件包含執行期介面實作。如需在映像中使用 aws-lambda-go/lambda 的範例，請參閱 使用僅限 AWS 作業系統的基礎映像 或 使用非AWS 基礎映像。

使用僅限 AWS 作業系統的基礎映像

Go 的實作方式與其他受管執行期不同。由於 Go 程式碼原生編譯至可執行的二進位檔，因此不需要專用語言執行時期。使用僅限作業系統的基礎映像來建置 Go 函數的容器映像。

Tags (標籤)	執行期	作業系統	Dockerfile	棄用
al2023	僅限作業系統的執行期	Amazon Linux 2023	GitHub 上僅限作業系統之執行期的 Dockerfile	2029 年 6 月 30 日
al2	僅限作業系統的執行期	Amazon Linux 2	GitHub 上僅限作業系統之執行期的 Dockerfile	2026 年 7 月 31 日

如需有關這類基礎映像的詳細資訊，請參閱 Amazon ECR 公有庫中 提供的內容。

您必須將 aws-lambda-go/lambda 套件加入 Go 處理常式。此套件會實作 Go 的程式設計模型 (包括執行期介面)。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- Docker (最低版本 25.0.0)
- Docker buildx 外掛程式。
- Go

從 provided.al2023 基礎映像建立映像

使用 **provided.al2023** 基礎映像建置和部署 Go 函數

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir hello
cd hello
```

2. 初始化新的 Go 模組。

```
go mod init example.com/hello-world
```

3. 將 lambda 程式庫新增為新模組的相依項。

```
go get github.com/aws/aws-lambda-go/lambda
```

4. 建立名為 `main.go` 的檔案，然後在文字編輯器中開啟。這是 Lambda 函數的程式碼。可以使用以下範本程式碼進行測試，也可以將其替換為您自己的程式碼。

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
}

func main() {
 lambda.Start(handler)
}
```

5. 使用文字編輯器在專案目錄中建立 Dockerfile。

   - 下列範例 Dockerfile 使用多階段建置。這可讓您在每個步驟中使用不同的基礎映像。您可以使用一個映像 (例如 Go 基礎映像檔) 來編譯程式碼並建置可執行二進位檔案。然後，可以在最終的 FROM 陳述式中使用不同的映像 (例如 `provided.al2023`) 來定義您部署到 Lambda 的映像。建置程序與最終部署映像是分開的，因此最終映像僅包含執行應用程式所需的檔案。

- 您可以使用選用 `lambda.norpc` 標籤，來排除 lambda 程式庫的遠端程序呼叫 (RPC) 元件。使用已棄用的 Go 1.x 執行時期時，才需要 RPC 元件。排除 RPC 會縮減部署套件的大小。
- 請注意，範例 Dockerfile 不包含 USER 指令。當您將容器映像部署到 Lambda 時，Lambda 會自動定義一個具有最低權限許可的預設 Linux 使用者。這與標準 Docker 行為不同，後者會在未提供 USER 指令時預設為 `root` 使用者。

Example– 多階段建置 Dockerfile

> **Note**
>
> 確保您在 Dockerfile 中指定的 Go 版本 (例如，`golang:1.20`) 與用於建立應用程式的 Go 版本相同。

```
FROM golang:1.20 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# Build with optional lambda.norpc tag
COPY main.go .
RUN go build -tags lambda.norpc -o main main.go
# Copy artifacts to a clean image
FROM public.ecr.aws/lambda/provided:al2023
COPY --from=build /helloworld/main ./main
ENTRYPOINT [ "./main" ]
```

6. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 `docker-image` 並為其提供 `test` 標籤。若要使映像與 Lambda 相容，必須使用 `--provenance=false` 選項。

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> 此命令會指定 `--platform linux/amd64` 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 `--platform linux/arm64` 選項。

(選用) 在本機測試映像

使用 執行期界面模擬器 以在本機測試映像。執行期界面模擬器包含在 provided.al2023 基礎映像中。

若要在本機電腦上執行執行期界面模擬器

1. 使用 docker run 命令啟動 Docker 影像。注意下列事項：

   - docker-image 是映像名稱，而 test 是標籤。
   - ./main 是來自 Dockerfile 的 ENTRYPOINT。

   ```
   docker run -d -p 9000:8080 \
   --entrypoint /usr/local/bin/aws-lambda-rie \
   docker-image:test ./main
   ```

   此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

2. 從新的終端機視窗，使用 curl 命令將事件張貼至下列端點：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
   ```

   此命令會透過空白事件調用函數，並傳回一個回應。部分函數可能需要使用 JSON 承載。範例：

   ```
   curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
    '{"payload":"hello world!"}'
   ```

3. 取得容器 ID。

   ```
   docker ps
   ```

4. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

   ```
   docker kill 3766c4ab331c
   ```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將 `--region`值設定為您要在 AWS 區域 其中建立 Amazon ECR 儲存庫的 。
   - `111122223333` 以您的 AWS 帳戶 ID 取代 。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數 AWS 區域 位於相同的 中。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
```

```
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - `docker-image:test` 為 Docker 映像檔的名稱和標籤。這是您在 `docker build` 命令中指定的映像名稱和標籤。
   - 將 `<ECRrepositoryUri>` 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 您可以使用不同帳戶中的映像來建立函數 AWS ，只要映像與 Lambda 函數位於相同的區域即可。如需詳細資訊，請參閱Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 會將映像標籤解析為特定映像摘要。這表示如果您將用來部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。

若要將新映像部署至相同的 Lambda 函數，必須使用 update-function-code 命令，即使 Amazon ECR 中的映像標籤保持不變亦如此。在以下範例中，`--publish` 選項會使用更新的容器映像來建立新的函數版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

使用非AWS 基礎映像

您可以從非AWS 基礎映像建置 Go 的容器映像。下列步驟中的 Dockerfile 範例使用 Alpine 基礎映像。

您必須將 aws-lambda-go/lambda 套件加入 Go 處理常式。此套件會實作 Go 的程式設計模型 (包括執行期介面)。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- AWS CLI 第 2 版
- Docker (最低版本 25.0.0)
- Docker buildx 外掛程式。
- Go

使用替代基礎映像建立映像

使用 Alpine 基礎映像建置和部署 Go 函數

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir hello
cd hello
```

2. 初始化新的 Go 模組。

```
go mod init example.com/hello-world
```

3. 將 lambda 程式庫新增為新模組的相依項。

```
go get github.com/aws/aws-lambda-go/lambda
```

4. 建立名為 main.go 的檔案，然後在文字編輯器中開啟。這是 Lambda 函數的程式碼。可以使用以下範本程式碼進行測試，也可以將其替換為您自己的程式碼。

```
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, event events.APIGatewayProxyRequest)
 (events.APIGatewayProxyResponse, error) {
 response := events.APIGatewayProxyResponse{
```

```
  StatusCode: 200,
  Body:       "\"Hello from Lambda!\"",
 }
 return response, nil
}

func main() {
 lambda.Start(handler)
}
```

5. 使用文字編輯器在專案目錄中建立 Dockerfile。下列 Dockerfile 範例使用 Alpine 基礎映像。請注意，範例 Dockerfile 不包含 USER 指令。當您將容器映像部署到 Lambda 時，Lambda 會自動定義一個具有最低權限許可的預設 Linux 使用者。這與標準 Docker 行為不同，後者會在未提供 USER 指令時預設為 `root` 使用者。

   Example Dockerfile

   > **Note**
   >
   > 確保您在 Dockerfile 中指定的 Go 版本 (例如，`golang:1.20`) 與用於建立應用程式的 Go 版本相同。

```
FROM golang:1.20.2-alpine3.16 as build
WORKDIR /helloworld
# Copy dependencies list
COPY go.mod go.sum ./
# Build
COPY main.go .
RUN go build -o main main.go
# Copy artifacts to a clean image
FROM alpine:3.16
COPY --from=build /helloworld/main /main
ENTRYPOINT [ "/main" ]
```

6. 使用 docker build 命令建立 Docker 映像檔。以下範例將映像命名為 `docker-image` 並為其提供 `test` 標籤。若要使映像與 Lambda 相容，必須使用 `--provenance=false` 選項。

```
docker buildx build --platform linux/amd64 --provenance=false -t docker-image:test
 .
```

> **Note**
>
> 此命令會指定 `--platform linux/amd64` 選項，確保無論建置機器的架構為何，您的容器都與 Lambda 執行環境相容。如果您打算使用 ARM64 指令集架構建立 Lambda 函數，務必將命令變更為改用 `--platform linux/arm64` 選項。

(選用) 在本機測試映像

使用 執行期界面模擬器 以在本機測試映像。您可以將模擬器建置到映像中，也可以使用以下步驟，將其安裝在本機電腦。

若要在本機電腦上安裝並執行執行期介面模擬器

1. 在您的專案目錄中執行以下命令，從 GitHub 下載執行期介面模擬器 (x86-64 架構)，並安裝在本機電腦上。

   Linux/macOS

   ```
   mkdir -p ~/.aws-lambda-rie && \
       curl -Lo ~/.aws-lambda-rie/aws-lambda-rie https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie && \
       chmod +x ~/.aws-lambda-rie/aws-lambda-rie
   ```

   若要安裝 arm64 模擬器，請將上一個命令中的 GitHub 儲存庫 URL 替換為以下內容：

   ```
   https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie-arm64
   ```

   PowerShell

   ```
   $dirPath = "$HOME\.aws-lambda-rie"
   if (-not (Test-Path $dirPath)) {
       New-Item -Path $dirPath -ItemType Directory
   }

   $downloadLink = "https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/download/aws-lambda-rie"
   $destinationPath = "$HOME\.aws-lambda-rie\aws-lambda-rie"
   ```

```
Invoke-WebRequest -Uri $downloadLink -OutFile $destinationPath
```

若要安裝 arm64 模擬器，請將 $downloadLink 更換為下列項目：

```
https://github.com/aws/aws-lambda-runtime-interface-emulator/releases/latest/
download/aws-lambda-rie-arm64
```

2. 使用 docker run 命令啟動 Docker 影像。注意下列事項：

   - docker-image 是映像名稱，而 test 是標籤。
   - /main 是來自 Dockerfile 的 ENTRYPOINT。

   Linux/macOS

   ```
   docker run --platform linux/amd64 -d -v ~/.aws-lambda-rie:/aws-lambda -p
    9000:8080 \
       --entrypoint /aws-lambda/aws-lambda-rie \
       docker-image:test \
           /main
   ```

   PowerShell

   ```
   docker run --platform linux/amd64 -d -v "$HOME\.aws-lambda-rie:/aws-lambda" -p
    9000:8080 `
   --entrypoint /aws-lambda/aws-lambda-rie `
   docker-image:test `
       /main
   ```

   此命令將映像作為容器執行，並在 localhost:9000/2015-03-31/functions/function/invocations 建立本機端點。

   > **Note**
   >
   > 如果您為 ARM64 指令集架構建立 Docker 映像檔，請務必將--platform linux/*arm64*選項改用選項。--platform linux/*amd64*

3. 將事件張貼至本機端點。

Linux/macOS

在 Linux 或 macOS 中，執行下列 `curl` 命令：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d '{}'
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
curl "http://localhost:9000/2015-03-31/functions/function/invocations" -d
 '{"payload":"hello world!"}'
```

PowerShell

在 PowerShell 中，執行下列 `Invoke-WebRequest` 命令：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{}' -ContentType "application/json"
```

此命令會透過空白事件調用函數，並傳回一個回應。如果您使用自己的函數程式碼而不是範例函數程式碼，則可能需要使用 JSON 承載調用該函數。範例：

```
Invoke-WebRequest -Uri "http://localhost:9000/2015-03-31/functions/function/
invocations" -Method Post -Body '{"payload":"hello world!"}' -ContentType
 "application/json"
```

4. 取得容器 ID。

```
docker ps
```

5. 使用 docker kill 命令停止容器。在此命令中，將 3766c4ab331c 替換為上一步驟中的容器 ID。

```
docker kill 3766c4ab331c
```

部署映像

若要將映像上傳至 Amazon ECR 並建立 Lambda 函數

1. 使用 get-login-password 命令，向 Amazon ECR 登錄檔驗證 Docker CLI。
   - 將 `--region`值設定為您要在 AWS 區域 其中建立 Amazon ECR 儲存庫的 。
   - `111122223333` 以您的 AWS 帳戶 ID 取代 。

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS --
password-stdin 111122223333.dkr.ecr.us-east-1.amazonaws.com
```

2. 使用 create-repository 命令在 Amazon ECR 中建立儲存庫。

```
aws ecr create-repository --repository-name hello-world --region us-east-1 --image-
scanning-configuration scanOnPush=true --image-tag-mutability MUTABLE
```

> **Note**
>
> Amazon ECR 儲存庫必須與 Lambda 函數 AWS 區域 位於相同的 中。

如果成功，您將會看到以下回應：

```
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:111122223333:repository/hello-
world",
        "registryId": "111122223333",
        "repositoryName": "hello-world",
        "repositoryUri": "111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-
world",
        "createdAt": "2023-03-09T10:39:01+00:00",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": true
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
```

```
    }
}
```

3. 從上一步驟的輸出中複製 `repositoryUri`。
4. 執行 docker tag 命令，將 Amazon ECR 儲存庫中的本機映像標記為最新版本。在此命令中：
   - `docker-image:test` 為 Docker 映像檔的名稱和標籤。這是您在 `docker build` 命令中指定的映像名稱和標籤。
   - 將 `<ECRrepositoryUri>` 替換為複製的 `repositoryUri`。確保在 URI 的末尾包含 `:latest`。

```
docker tag docker-image:test <ECRrepositoryUri>:latest
```

範例：

```
docker tag docker-image:test 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

5. 執行 docker push 命令，將本機映像部署至 Amazon ECR 儲存庫。確保在儲存庫 URI 的末尾包含 `:latest`。

```
docker push 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest
```

6. 建立函數的執行角色 (若您還沒有的話)。在下一個步驟中您需要角色的 Amazon Resource Name (ARN)。
7. 建立 Lambda 函數。對於 `ImageUri`，從之前的設定中指定儲存庫 URI。確保在 URI 的末尾包含 `:latest`。

```
aws lambda create-function \
  --function-name hello-world \
  --package-type Image \
  --code ImageUri=111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --role arn:aws:iam::111122223333:role/lambda-ex
```

> **Note**
>
> 您可以使用不同帳戶中的映像來建立函數 AWS ，只要映像與 Lambda 函數位於相同的區域即可。如需詳細資訊，請參閱Amazon ECR 跨帳戶許可。

8. 調用函數。

```
aws lambda invoke --function-name hello-world response.json
```

您應該看到如下回應：

```
{
  "ExecutedVersion": "$LATEST",
  "StatusCode": 200
}
```

9. 若要查看函數的輸出，請檢查 `response.json` 檔案。

若要更新函數程式碼，您必須再次建置映像、將新映像上傳到 Amazon ECR 存放庫，然後使用 update-function-code 命令將映像部署到 Lambda 函數。

Lambda 會將映像標籤解析為特定映像摘要。這表示如果您將用來部署函數的映像標籤指向 Amazon ECR 中的新映像，Lambda 不會自動更新函數以使用新映像。

若要將新映像部署至相同的 Lambda 函數，必須使用 update-function-code 命令，即使 Amazon ECR 中的映像標籤保持不變亦如此。在以下範例中，`--publish` 選項會使用更新的容器映像來建立新的函數版本。

```
aws lambda update-function-code \
  --function-name hello-world \
  --image-uri 111122223333.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest \
  --publish
```

使用 Go Lambda 函數的層

不建議使用層來管理以 Go 編寫的 Lambda 函式相依項。這是因為以 Go 編寫的 Lambda 函數會編譯為單一可執行檔，您在部署函數時該檔會提供給 Lambda。此可執行檔包含經過編譯的函數程式碼及其所有相依項。使用層不僅會使程序複雜化，還會導致冷啟動時間增加，因為函數需要在初始化階段期間將額外的組件載入記憶體。

若要在 Go 處理常式中使用外部相依項，請直接將其包含在部署套件中。如此一來，您既能簡化部署程序，又能利用內建的 Go 編譯器最佳化。如需如何在函數中匯入和使用適用於 Go 的 AWS SDK 等相依項的範例，請參閱the section called “處理常式”。

記錄和監控 Go Lambda 函數

AWS Lambda 會自動代表您監控 Lambda 函數，並將日誌傳送至 Amazon CloudWatch。您的 Lambda 函數隨附有 CloudWatch Logs 日誌群組，且函數的每一執行個體各有一個日誌串流。Lambda 執行期環境會將每次調用的詳細資訊傳送至日誌串流，並且轉傳來自函數程式碼的日誌及其他輸出。如需詳細資訊，請參閱將 Lambda 函式日誌傳送至 CloudWatch Logs。

此頁面說明如何從 Lambda 函數的程式碼產生日誌輸出，並使用 AWS Command Line Interface、Lambda 主控台或 CloudWatch 主控台存取日誌。

章節

- 建立傳回日誌的函數
- 在 Lambda 主控台檢視日誌
- 在 CloudWatch 主控台中檢視 記錄
- 使用 AWS Command Line Interface (AWS CLI) 檢視日誌
- 刪除日誌

建立傳回日誌的函數

若要從您的函數程式碼輸出日誌，您可以使用 fmt 套件上的方法，或任何能寫入 `stdout` 或 `stderr` 的記錄程式庫。下列範例使用日誌套件。

Example main.go - 記錄

```
func handleRequest(ctx context.Context, event events.SQSEvent) (string, error) {
  // event
  eventJson, _ := json.MarshalIndent(event, "", "  ")
  log.Printf("EVENT: %s", eventJson)
  // environment variables
  log.Printf("REGION: %s", os.Getenv("AWS_REGION"))
  log.Println("ALL ENV VARS:")
  for _, element := range os.Environ() {
    log.Println(element)
  }
```

Example記錄格式

```
START RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Version: $LATEST
```

```
2020/03/27 03:40:05 EVENT: {
  "Records": [
    {
      "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
      "receiptHandle": "MessageReceiptHandle",
      "body": "Hello from SQS!",
      "md5OfBody": "7b27xmplb47ff90a553787216d55d91d",
      "md5OfMessageAttributes": "",
      "attributes": {
        "ApproximateFirstReceiveTimestamp": "1523232000001",
        "ApproximateReceiveCount": "1",
        "SenderId": "123456789012",
        "SentTimestamp": "1523232000000"
      },
      ...
2020/03/27 03:40:05 AWS_LAMBDA_LOG_STREAM_NAME=2020/03/27/
[$LATEST]569cxmplc3c34c7489e6a97ad08b4419
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_NAME=blank-go-function-9DV3XMPL6XBC
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_MEMORY_SIZE=128
2020/03/27 03:40:05 AWS_LAMBDA_FUNCTION_VERSION=$LATEST
2020/03/27 03:40:05 AWS_EXECUTION_ENV=AWS_Lambda_go1.x
END RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71
REPORT RequestId: dbda340c-xmpl-4031-8810-11bb609b4c71 Duration: 38.66 ms Billed
 Duration: 243 ms Memory Size: 128 MB Max Memory Used: 54 MB Init Duration: 203.69 ms
XRAY TraceId: 1-5e7d7595-212fxmpl9ee07c4884191322 SegmentId: 42ffxmpl0645f474 Sampled:
 true
```

Go 執行時間會記錄每次調用的 START、END 和 REPORT 行。報告明細行提供下列詳細資訊。

REPORT 行資料欄位

- RequestId - 進行調用的唯一請求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。當調用共用執行環境時，Lambda 會報告所有調用使用的記憶體上限。此行為可能會導致報告值高於預期值。
- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId - 對於追蹤的請求，這是 AWS X-Ray 追蹤 ID。

- SegmentId - 對於追蹤的請求，這是 X-Ray 區段 ID。
- 已取樣 - 對於追蹤的請求，這是取樣結果。

在 Lambda 主控台檢視日誌

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

在 CloudWatch 主控台中檢視 記錄

您可以使用 Amazon CloudWatch 主控台來檢視所有 Lambda 函數調用的日誌。

若要在 CloudWatch 主控台上檢視日誌

1. 在 CloudWatch 主控台上開啟日誌群組頁面。
2. 選擇您的函數的日誌群組 (/aws/lambda/***your-function-name***)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立其他執行個體以處理並行調用時，就會顯示日誌串流。若要尋找特定調用的日誌，建議您使用 檢測函數 AWS X-Ray。X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

使用 AWS Command Line Interface (AWS CLI) 檢視日誌

是一種 AWS CLI 開放原始碼工具，可讓您使用命令列 Shell 中的 命令與 AWS 服務互動。若要完成本節中的步驟，您必須擁有 AWS CLI 版本 2。

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 get-logs.sh。

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
```

```
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

在 中檢測 Go 程式碼 AWS Lambda

Lambda 與 整合 AWS X-Ray ，以協助您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追蹤請求，因為它會周遊您應用程式中的資源，其中可能包含 Lambda 函數和其他 AWS 服務。

若要將追蹤資料傳送至 X-Ray，您可以使用以下兩個 SDK 庫之一：

- AWS Distro for OpenTelemetry (ADOT) – OpenTelemetry (OTel) SDK 的安全、生產就緒、 AWS支援的分發。
- AWS 適用於 Go 的 X-Ray 開發套件 – 用於產生追蹤資料並將其傳送至 X-Ray 的開發套件。

每個 SDK 均提供將遙測資料傳送至 X-Ray 服務的方法。然後，您可以使用 X-Ray 來檢視、篩選應用程式的效能指標並獲得洞察，從而識別問題和進行最佳化的機會。

> **Important**
>
> X-Ray 和 Powertools AWS Lambda SDKs是 提供的緊密整合檢測解決方案的一部分 AWS。ADOT Lambda Layers 是用於追蹤檢測之業界通用標準的一部分，這類檢測一般會收集更多資料，但可能不適用於所有使用案例。您可以使用任一解決方案在 X-Ray 中實作端對端追蹤。若要深入了解如何在兩者之間做選擇，請參閱在 AWS Distro for OpenTelemetry 和 X-Ray SDK 之間進行選擇。

章節

- 使用 ADOT 來檢測您的 Go 函數
- 使用 X-Ray SDK 來檢測 Go 函數
- 透過 Lambda 主控台來啟用追蹤
- 透過 Lambda API 啟用追蹤
- 使用 啟用追蹤 CloudFormation
- 解讀 X-Ray 追蹤

使用 ADOT 來檢測您的 Go 函數

ADOT 提供全受管 Lambda 層，包含使用 OTel SDK 收集遙測資料所需的一切內容。透過取用此層，您可以檢測 Lambda 函數，而無需修改任何函數程式碼。您還可以將層設定為對 OTel 進行自訂初始化。如需詳細資訊，請參閱 ADOT 文件中的針對 Lambda 上的 ADOT 收集器進行自訂組態設定。

針對 Go 執行階段，您可以新增適用於 ADOT Go 的AWS 受管 Lambda 層來自動檢測您的函數。如需有關如何新增此層的詳細資訊，請參閱 ADOT 文件中的適用於 Go 的AWS Distro for OpenTelemetry Lambda 支援。

使用 X-Ray SDK 來檢測 Go 函數

若要記錄 Lambda 函數對應用程式中其他資源進行的呼叫詳細資訊，您也可以使用 AWS X-Ray SDK for Go。如要取得 SDK，使用 `go get` 從 SDK 的 GitHub 儲存庫下載即可：

```
go get github.com/aws/aws-xray-sdk-go
```

若要檢測 AWS SDK 用戶端，請將用戶端傳遞至 `xray.AWS()`方法。然後，便可以使用該方法的 `WithContext` 版本來追蹤呼叫。

```
svc := s3.New(session.New())
xray.AWS(svc.Client)
...
svc.ListBucketsWithContext(ctx aws.Context, input *ListBucketsInput)
```

新增正確的相依項並完成必要的程式碼變更後，請透過 Lambda 主控台或 API 在函數組態中啟用追蹤功能。

透過 Lambda 主控台來啟用追蹤

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態，然後選擇監控和操作工具。
4. 在其他監控工具欄位中，選擇編輯。
5. 在 CloudWatch Application Signals 和 AWS X-Ray 下，選擇啟用 Lambda 服務追蹤。
6. 選擇儲存。

透過 Lambda API 啟用追蹤

使用 AWS CLI 或 AWS SDK 在 Lambda 函數上設定追蹤，請使用下列 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

下列範例 AWS CLI 命令會在名為 my-function 的函數上啟用主動追蹤。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

使用 啟用追蹤 CloudFormation

若要在 CloudFormation 範本中的 `AWS::Lambda::Function` 資源上啟用追蹤，請使用 `TracingConfig` 屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用 `Tracing` 屬性。

Example template.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Serverless::Function
```

```
    Properties:
      Tracing: Active
      ...
```

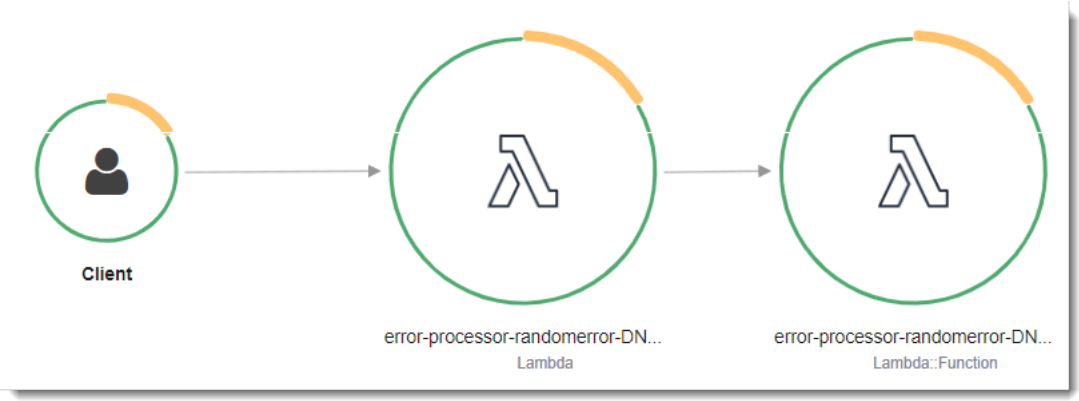
解讀 X-Ray 追蹤

您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將 AWSXRayDaemonWriteAccess 政策新增至執行角色。

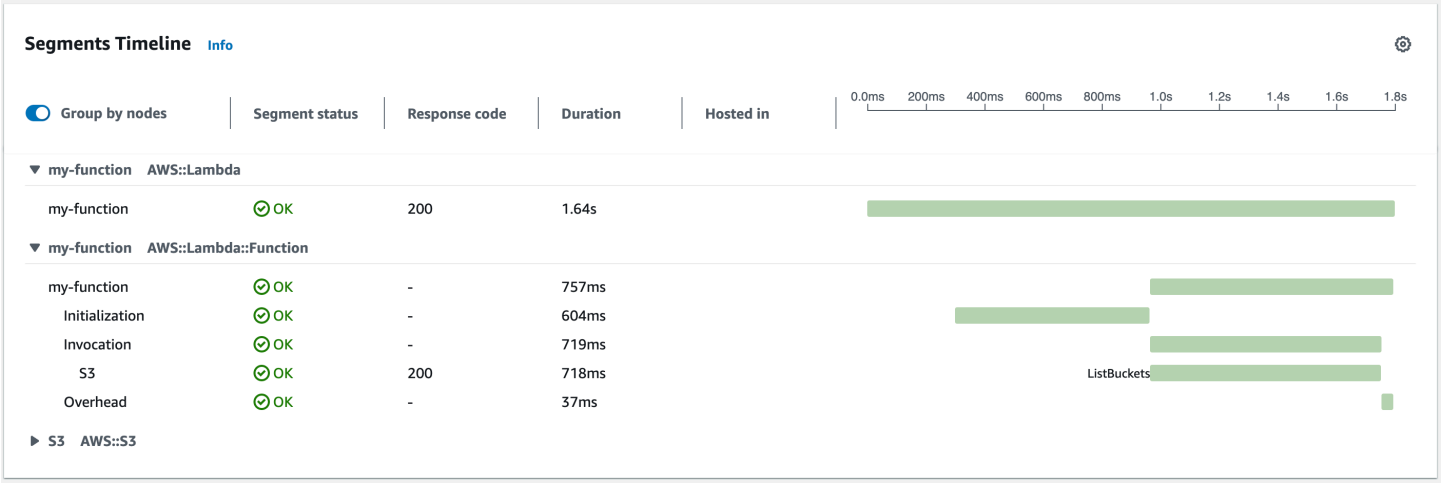
設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 服務圖顯示了有關應用程式及其所有元件的資訊。下列範例顯示了一個具有兩個函數的應用程式。主要函式會處理事件，有時會傳回錯誤。頂端的第二個函數會處理出現在第一個日誌群組中的錯誤，並使用 AWS SDK 呼叫 X-Ray、Amazon Simple Storage Service (Amazon S3) 和 Amazon CloudWatch Logs。

X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。不能針對函數設定 X-Ray 取樣率。

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會在每個追蹤上記錄 2 個區段，這會在服務圖表上建立兩個節點。下圖反白顯示了這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為 my-function，但其中之一的來源為 AWS::Lambda，而另一個的來源為 AWS::Lambda::Function。如果 AWS::Lambda 區段顯示錯誤，Lambda 服務就會出現問題。如果 AWS::Lambda::Function 區段顯示錯誤，表示您的函數出現了問題。



此範例會展開 AWS::Lambda::Function 區段以顯示其三個子區段：

> **Note**
>
> AWS 目前正在對 Lambda 服務實作變更。由於這些變更，您可能會看到系統日誌訊息的結構和內容，與 AWS 帳戶中不同 Lambda 函數發出的追蹤區段之間存在細微差異。此處顯示的追蹤範例說明了舊式函數區段。下列段落說明了舊式和新式區段之間的差異。這些變化將在未來幾週內實作，除中國和 GovCloud 區域以外，所有 AWS 區域 中的所有函數都會轉換至使用新格式的日誌訊息和追蹤區段。

舊式函數區段包含下列子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行期為做好準備以處理下一個事件所花費的時間。

新式函數區段不包含 `Invocation` 子區段。相反地，客戶子區段會直接連接至函數區段。如需舊式和新式函數區段結構的詳細資訊，請參閱the section called “了解 X-Ray 追蹤”。

您也可以檢測 HTTP 用戶端、記錄 SQL 查詢，以及建立具有註釋和中繼資料的自訂子區段。如需詳細資訊，請參閱《AWS X-Ray 開發人員指南》中的適用於 Go 的AWS X-Ray 開發套件。

> **定價**
>
> 您可以在每月免費使用 X-Ray 追蹤，最高可達特定限制，作為 AWS 免費方案的一部分。達到閾值後，X-Ray 會收取追蹤儲存及擷取的費用。如需詳細資訊，請參閱 AWS X-Ray 定價。

使用 C# 建置 Lambda 函數

您可以使用受管 .NET 8 執行時間、自訂執行時間或容器映像，在 Lambda 中執行 .NET 應用程式。編譯應用程式程式碼之後，您能以 .zip 檔或容器映像檔形式部署至 Lambda。Lambda 提供適用於 .NET 語言的以下執行期：

Name	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
.NET 10	`dotnet10`	Amazon Linux 2023	2028 年 11 月 14 日	2028 年 12 月 14 日	2029 年 1 月 15 日
.NET 9（僅限容器）	`dotnet9`	Amazon Linux 2023	2026 年 11 月 10 日	未排程	未排程
.NET 8	`dotnet8`	Amazon Linux 2023	2026 年 11 月 10 日	2026 年 12 月 10 日	2027 年 1 月 11 日

設定您的 .NET 開發環境

若要開發和建置 Lambda 函數，您可以使用任何常用的 .NET 整合式開發環境 (IDE)，包括 Microsoft Visual Studio、Visual Studio Code 和 JetBrains Rider。為了簡化您的開發體驗， AWS 提供一組 .NET 專案範本以及`Amazon.Lambda.Tools`命令列界面 (CLI)。

執行下列 .NET CLI 命令，即可安裝這些專案範本和命令列工具。

安裝 .NET 專案範本

若要安裝專案範本，請執行下列命令：

```
dotnet new install Amazon.Lambda.Templates
```

安裝和更新 CLI 工具

執行下列命令來安裝、更新和解除安裝 `Amazon.Lambda.Tools` CLI。

若要安裝命令列工具，請執行以下操作：

```
dotnet tool install -g Amazon.Lambda.Tools
```

若要更新命令列工具，請執行以下操作：

```
dotnet tool update -g Amazon.Lambda.Tools
```

若要解除安裝命令列工具，請執行以下操作：

```
dotnet tool uninstall -g Amazon.Lambda.Tools
```

定義 C# 格式的 Lambda 函數處理常式

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

本頁介紹了如何在 C# 中將 Lambda 函式處理常式與 .NET 受管執行時期搭配使用，包括專案設定選項、命名慣例及最佳實務。本頁還提供了一個 C# Lambda 函式範例，該函式會取得訂單的相關資訊、產生文字檔案收據，並將該檔案放入 Amazon Simple Storage Service (S3) 儲存貯體。如需編寫函數後如何部署函數的詳細資訊，請參閱the section called “部署套件”或the section called “部署容器映像”。

主題

- 設定 C# 處理常式專案
- C# Lambda 函式程式碼範例
- 類別庫處理常式
- 可執行組件處理常式
- C# 函式的有效處理常式簽章
- 處理常式命名慣例
- C# Lambda 函式中的序列化
- 以檔案為基礎的函數
- 存取和使用 Lambda 內容物件
- 在處理常式中使用 適用於 .NET 的 SDK v3
- 存取環境變數
- 使用全域狀態
- 使用 Lambda Annotations 架構簡化函數程式碼
- C# Lambda 函數的程式碼最佳實務

設定 C# 處理常式專案

使用 C# 開發 Lambda 函式時，流程涉及編寫程式碼以及將程式碼部署至 Lambda。有兩種不同的執行模型可用於在 .NET 中部署 Lambda 函式：類別庫方法和可執行組件方法。

在類別庫方法中，您需將函式程式碼封裝為 .NET 組件 (`.dll`)，並透過 .NET 受管執行時期 (`dotnet8`) 將其部署至 Lambda。對於處理常式名稱，Lambda 期望字串的格式為

AssemblyName::Namespace.Classname::Methodname。在函數的初始化階段，系統會初始化函數的類別，並執行建構函數中的任何程式碼。

在可執行組件方法中，您可以使用 C# 9 中首次推出的頂層陳述式功能。此方法會產生一個可執行組件，每當 Lambda 收到函數的調用命令時，就會執行該組件。此方法也會使用 .NET 受管執行時期 (dotnet8)。對於處理常式名稱，您需要將要執行的可執行組件名稱提供給 Lambda。

本頁上的主要範例示範了類別庫方法。您可以透過各種方式初始化 C# Lambda 專案，但最簡單的方法是搭配 Amazon.Lambda.Tools CLI 使用 .NET CLI。依照中the section called “開發環境”的步驟設定 Amazon.Lambda.Tools CLI。接著，使用下列命令初始化專案：

```
dotnet new lambda.EmptyFunction --name ExampleCS
```

此命令會產生下列檔案結構：

```
/project-root
    # src
        # ExampleCS
            # Function.cs (contains main handler)
            # Readme.md
            # aws-lambda-tools-defaults.json
            # ExampleCS.csproj
    # test
         # ExampleCS.Tests
            # FunctionTest.cs (contains main handler)
            # ExampleCS.Tests.csproj
```

在此檔案結構中，函式的主要處理常式邏輯位於 Function.cs 檔案中。

C# Lambda 函式程式碼範例

以下 C# Lambda 函式程式碼範例會取得訂單的相關資訊、產生文字檔案收據，並將該檔案放入 Amazon S3 儲存貯體。

Example **Function.cs** Lambda 函數

```
using System;
using System.Text;
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using Amazon.S3.Model;
```

```
// Assembly attribute to enable Lambda function logging
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace ExampleLambda;

public class Order
{
    public string OrderId { get; set; } = string.Empty;
    public double Amount { get; set; }
    public string Item { get; set; } = string.Empty;
}

public class OrderHandler
{
    private static readonly AmazonS3Client s3Client = new();

    public async Task<string> HandleRequest(Order order, ILambdaContext context)
    {
        try
        {
            string? bucketName = Environment.GetEnvironmentVariable("RECEIPT_BUCKET");
            if (string.IsNullOrWhiteSpace(bucketName))
            {
                throw new ArgumentException("RECEIPT_BUCKET environment variable is not
 set");
            }

            string receiptContent = $"OrderID: {order.OrderId}\nAmount:
 ${order.Amount:F2}\nItem: {order.Item}";
            string key = $"receipts/{order.OrderId}.txt";

            await UploadReceiptToS3(bucketName, key, receiptContent);

            context.Logger.LogInformation($"Successfully processed order
 {order.OrderId} and stored receipt in S3 bucket {bucketName}");
            return "Success";
        }
        catch (Exception ex)
        {
            context.Logger.LogError($"Failed to process order: {ex.Message}");
            throw;
        }
```

```
    }

    private async Task UploadReceiptToS3(string bucketName, string key, string
receiptContent)
    {
        try
        {
            var putRequest = new PutObjectRequest
            {
                BucketName = bucketName,
                Key = key,
                ContentBody = receiptContent,
                ContentType = "text/plain"
            };

            await s3Client.PutObjectAsync(putRequest);
        }
        catch (AmazonS3Exception ex)
        {
            throw new Exception($"Failed to upload receipt to S3: {ex.Message}", ex);
        }
    }
}
```

此 Function.cs 檔案包含以下程式碼區段：

- using 陳述式：使用這些陳述式來匯入 Lambda 函式所需的 C# 類別。
- [assembly: LambdaSerializer(...)]：LambdaSerializer 是組件屬性，指示 Lambda 先自動將 JSON 事件承載轉換為 C# 物件之後再傳遞至函式。
- namespace ExampleLambda：此程式碼會定義命名空間。在 C# 中，命名空間名稱不必與檔案名稱相符。
- public class Order {...}：此程式碼會定義預期輸入事件的結構。
- public class OrderHandler {...}：此程式碼會定義 C# 類別。您可以在其中定義主要處理常式方法及其他輔助方法。
- private static readonly AmazonS3Client s3Client = new();：此程式碼會在主要處理常式方法外部，透過預設憑證提供者鏈初始化 Amazon S3 用戶端。這會使 Lambda 在初始化階段執行此程式碼。
- public async ... HandleRequest (Order order, ILambdaContext context)：這是主要處理常式方法，其中包含應用程式的主要邏輯。

- `private async Task UploadReceiptToS3(...) {}`：這是主要 handleRequest 處理常式方法所參照的輔助方法。

由於此函式需要 Amazon S3 SDK 用戶端，因此您必須將其新增至專案的相依項。您可以導覽至 `src/ExampleCS` 並執行下列命令來完成此操作：

```
dotnet add package AWSSDK.S3
```

將中繼資料資訊新增至 aws-lambda-tools-defaults.json

依預設，產生的 `aws-lambda-tools-defaults.json` 檔案不包含函式的 `profile` 或 `region` 資訊。此外，請將 `function-handler` 字串更新為正確的值 (`ExampleCS::ExampleLambda.OrderHandler::HandleRequest`)。您可以手動自行此更新作業，並新增必要的中繼資料，以便函式使用特定的憑證設定檔與區域。例如，`aws-lambda-tools-defaults.json` 檔案應該與下列範例類似：

```
{
  "Information": [
    "This file provides default values for the deployment wizard inside Visual Studio
 and the AWS Lambda commands added to the .NET Core CLI.",
    "To learn more about the Lambda commands with the .NET Core CLI execute the
 following command at the command line in the project root directory.",
    "dotnet lambda help",
    "All the command line options for the Lambda command can be specified in this
 file."
  ],
  "profile": "default",
  "region": "us-east-1",
  "configuration": "Release",
  "function-architecture": "x86_64",
  "function-runtime": "dotnet8",
  "function-memory-size": 512,
  "function-timeout": 30,
  "function-handler": "ExampleCS::ExampleLambda.OrderHandler::HandleRequest"
}
```

若要讓此函式正常運作，其執行角色必須允許 `s3:PutObject` 動作。此外，請確保定義 `RECEIPT_BUCKET` 環境變數。成功調用後，Amazon S3 儲存貯體應包含收據檔案。

類別庫處理常式

本頁上的主要範例程式碼示範了類別庫處理常式。類別庫處理常式具有下列結構：

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace NAMESPACE;

...

public class CLASSNAME {
    public async Task<string> METHODNAME (...) {
    ...
    }
}
```

建立 Lambda 函式時，需要以字串的形式在處理常式欄位中向 Lambda 提供函式處理常式的相關資訊。目的是指示 Lambda 在調用函數時要在程式碼中執行哪種方法。在 C# 中，對於類別庫處理常式，處理常式字串的格式為 ASSEMBLY::TYPE::METHOD，其中：

- ASSEMBLY 是應用程式的 .NET 組件名稱。如果您使用 Amazon.Lambda.Tools CLI 建置應用程式，而沒有使用 .csproj 檔案中的 AssemblyName 屬性來設定組件名稱，則 ASSEMBLY 即為 .csproj 檔案的名稱。
- TYPE 是處理常式類型的完整名稱，即 NAMESPACE.CLASSNAME。
- METHOD 是程式碼中主要處理常式方法的名稱，即 METHODNAME。

在本頁上的主要範例程式碼中，如果組件名為 ExampleCS，則完整處理常式字串為 ExampleCS::ExampleLambda.OrderHandler::HandleRequest。

可執行組件處理常式

您也可以在 C# 中將 Lambda 函式定義為可執行組件。可執行組件處理常式利用 C# 的頂層陳述式功能。在該功能中，編譯器會產生 Main() 方法，並將函式程式碼放入其中。使用可執行組件時，必須引導 Lambda 執行期。若要執行此操作，請在程式碼中使用 LambdaBootstrapBuilder.Create 方法。此方法的輸入參數包含主要處理常式函式以及要使用的 Lambda 序列化程式。以下所示為 C# 中可執行組件處理常式的範例：

```
namespace GetProductHandler;
```

```
IDatabaseRepository repo = new DatabaseRepository();

await LambdaBootstrapBuilder.Create<APIGatewayProxyRequest>(Handler, new
 DefaultLambdaJsonSerializer())
    .Build()
    .RunAsync();

async Task<APIGatewayProxyResponse> Handler(APIGatewayProxyRequest apigProxyEvent,
 ILambdaContext context)
{
    var id = apigProxyEvent.PathParameters["id"];
    var databaseRecord = await this.repo.GetById(id);

    return new APIGatewayProxyResponse
    {
        StatusCode = (int)HttpStatusCode.OK,
        Body = JsonSerializer.Serialize(databaseRecord)
    };
};
```

在可執行組件處理常式的處理常式欄位中，指示 Lambda 如何執行程式碼的處理常式字串就是組件的名稱。在此範例中，該名稱為 `GetProductHandler`。

C# 函式的有效處理常式簽章

在 C# 中，有效的 Lambda 處理常式簽章包含 0 到 2 個引數。一般而言，處理常式簽章有兩個引數，如主要範例所示：

```
public async Task<string> HandleRequest(Order order, ILambdaContext context)
```

提供兩個引數時，第一個引數必須是事件輸入，第二個引數必須是 Lambda 內容物件。這兩個引數皆為選用。例如，以下 C# 中的 Lambda 處理常式簽章同樣有效：

- `public async Task<string> HandleRequest()`
- `public async Task<string> HandleRequest(Order order)`
- `public async Task<string> HandleRequest(ILambdaContext context)`

除處理常式簽章的基礎語法外，另有一些額外限制：

- 處理常式簽章中不得使用 unsafe 關鍵字。但處理常式方法及其相依項中可使用 unsafe 內容。如需詳細資訊，請參閱 Microsoft 文件網站上的 unsafe (C# reference)。
- 處理常式不得使用 params 關鍵字，也不得將 ArgIterator 作為輸入或傳回參數。這些關鍵字支援可變數量的參數。處理常式最多能接受兩個引數。
- 處理常式不可為泛型方法。換言之，其無法使用泛型類型參數，例如 <T>。
- Lambda 不支援簽章中包含 async void 的非同步處理常式。

處理常式命名慣例

C# 中的 Lambda 處理常式沒有嚴格的命名限制。不過，在部署函式時，必須確保向 Lambda 提供正確的處理常式字串。正確的處理常式字串視您要部署的是類別庫處理常式還是可執行組件處理常式而定。

雖然您可以為處理常式使用任何名稱，但 C# 中的函式名稱通常採用 PascalCase 命名方式。此外，雖然檔案名稱無需與類別名稱或處理常式名稱相符，但最佳實務通常是讓二者保持一致：例如類別名稱為 OrderHandler，檔案名稱則為 OrderHandler.cs。舉例而言，您可以將此範例中的檔案名稱從 Function.cs 修改為 OrderHandler.cs。

C# Lambda 函式中的序列化

JSON 是 Lambda 函數最常見的標準輸入格式。在此範例中，函數預期輸入類似以下內容：

```
{
    "orderId": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

在 C# 中，您可以定義類別中預期輸入事件的結構。在此範例中，我們定義 Order 類別來模擬這種輸入：

```
public class Order
{
    public string OrderId { get; set; } = string.Empty;
    public double Amount { get; set; }
    public string Item { get; set; } = string.Empty;
}
```

若您的 Lambda 函數使用輸入或輸出類型而非 Stream 物件，您必須將序列化程式庫新增至您的應用程式。如此，您可以將 JSON 輸入轉換為您所定義類別的執行個體。Lambda 中 C# 函式的序列化方法分為兩種：反射式序列化與原始碼產生式序列化。

反射式序列化

AWS 提供預先建置的程式庫，您可以快速新增至您的應用程式。這些程式庫使用反射實作序列化。使用下列任一套件實作反射式序列化：

- Amazon.Lambda.Serialization.SystemTextJson – 此套件在後端透過 System.Text.Json 執行序列化任務。
- Amazon.Lambda.Serialization.Json – 此套件在後端透過 Newtonsoft.Json 執行序列化任務。

您也可以實作 ILambdaSerializer 界面 (隨 Amazon.Lambda.Core 程式庫提供) 來建立自己的序列化程式庫。此界面定義了兩種方法：

- `T Deserialize<T>(Stream requestStream);`

  若實作此方法，會將請求承載從 Invoke API 還原序列化至傳遞到 Lambda 函數處理常式的物件。
- `T Serialize<T>(T response, Stream responseStream);`

  若實作此方法，會將從 Lambda 函數處理常式傳回的結果，序列化至 Invoke API 作業傳回的回應承載中。

本頁上的主要範例使用反射式序列化。以反射為基礎的序列化可立即搭配 運作 AWS Lambda ，無需額外設定，因此是簡化的理想選擇。不過，此方式確需佔用更多函式記憶體。由於執行時期反射，函式延遲也可能會更高。

原始碼產生式序列化

透過原始碼產生式序列化，序列化程式碼會在編譯時產生。此方式不需要使用反射，並且可提高函式的效能。若要在函式中使用原始碼產生式序列化，請執行以下操作：

- 建立一個繼承自 JsonSerializerContext 的新部分分類，為需要序列化或還原序列化的所有類型新增 JsonSerializable 屬性。
- 設定 LambdaSerializer 以使用 SourceGeneratorLambdaJsonSerializer<T>。
- 更新應用程式程式碼中的任何手動序列化與反序列化，以使用新建立的類別。

下列範例示範了如何修改本頁上採用反射式序列化的主要範例，使其改為使用原始碼產生式序列化。

```
using System.Text.Json;
using System.Text.Json.Serialization;

...

public class Order
{
    public string OrderId { get; set; } = string.Empty;
    public double Amount { get; set; }
    public string Item { get; set; } = string.Empty;
}

[JsonSerializable(typeof(Order))]
public partial class OrderJsonContext : JsonSerializerContext {}

public class OrderHandler
{

    ...

    public async Task<string> HandleRequest(string input, ILambdaContext context)
    {

    var order = JsonSerializer.Deserialize(input, OrderJsonContext.Default.Order);

    ...

    }

}
```

與反射式序列化相比，原始碼產生式序列化需要更多的設定。不過，由於程式碼在編譯時產生，因此使用原始碼產生式序列化的函式通常佔用較少的記憶體，並具有更好的效能。為協助消除函式冷啟動，建議改為使用原始碼產生式序列化。

> **Note**
>
> 如果您想透過 Lambda 使用原生預先編譯 (AOT)，必須使用原始碼產生式序列化。

以檔案為基礎的函數

在 .NET 10 中推出以檔案為基礎的應用程式，可讓您從單一.cs檔案建置 .NET 應用程式，而不需要.csproj檔案或目錄結構。Lambda 支援以檔案為基礎的函數，從 .NET 10 開始。它們提供簡化、輕量的方式來在 C# 中建置 Lambda 函數。

開始建立 C# 檔案型 Lambda 函數的最快方法是使用 Amazon.Lambda.Templates套件。若要安裝此套件，請執行下列命令：

```
dotnet new install Amazon.Lambda.Templates
```

接著，建立 C# 檔案型 Lambda 範例函數：

```
dotnet new lambda.FileBased -n MyLambdaFunction
```

以檔案為基礎的函數使用可執行的組件處理常式。因此，您必須包含 Amazon.Lambda.RuntimeSupport NuGet 套件，並使用 LambdaBootstrapBuilder.Create方法來註冊事件類型的 .NET 處理常式函數，並啟動 .NET Lambda 執行期用戶端。

根據預設，以檔案為基礎的函數會使用 .NET Native AOT，這需要來源產生的序列化。您可以在#:property PublishAot=false來源檔案中指定 來停用原生 AOT。如需在 Lambda 中使用原生 AOT 的詳細資訊，請參閱 the section called “原生 AOT 編譯”。

存取和使用 Lambda 內容物件

Lambda 內容物件 包含有關調用、函數以及執行環境的資訊。在此範例中，內容物件的類型為 Amazon.Lambda.Core.ILambdaContext，並且內容物件也是主要處理常式函式的第二個引數。

```
public async Task<string> HandleRequest(Order order, ILambdaContext context) {
    ...
}
```

內容物件為選用輸入。如需有關可接受的有效處理常式簽章的詳細資訊，請參閱 the section called “C# 函式的有效處理常式簽章”。

內容物件有助於將函式日誌輸出至 Amazon CloudWatch。您可以使用 context.getLogger() 方法取得用於記錄日誌的 LambdaLogger 物件。在此範例中，如果處理因任何原因失敗，我們可以使用記錄器來記錄錯誤訊息：

```
context.Logger.LogError($"Failed to process order: {ex.Message}");
```

除了記錄日誌之外，您還可以將內容物件用於函式監控。如需內容物件的詳細資訊，請參閱the section called “Context”。

在處理常式中使用 適用於 .NET 的 SDK v3

通常，您將使用 Lambda 函數與其他 AWS 資源互動或進行更新。與這些資源互動的最簡單方法是使用適用於 .NET 的 SDK v3。

> **Note**
>
> 適用於 .NET 的 SDK (v2) 已棄用。我們建議您僅使用 適用於 .NET 的 SDK v3。

您可以透過下列 `Amazon.Lambda.Tools` 命令將 SDK 相依項新增至專案：

```
dotnet add package <package_name>
```

例如，在本頁上的主要範例中，我們需透過 Amazon S3 API 將收據上傳至 S3。我們可以使用下列命令匯入 Amazon S3 SDK 用戶端：

```
dotnet add package AWSSDK.S3
```

此命令會將相依項新增至專案。您應該也會在專案的 `.csproj` 檔案中看到類似下列的程式碼行：

```
<PackageReference Include="AWSSDK.S3" Version="3.7.2.18" />
```

然後，直接在 C# 程式碼中匯入相依項：

```
using Amazon.S3;
using Amazon.S3.Model;
```

範例程式碼接著會初始化 Amazon S3 用戶端 (使用預設憑證提供者鏈)，如下所示：

```
private static readonly AmazonS3Client s3Client = new();
```

在此範例中，我們在主要處理常式函式外部初始化 Amazon S3 用戶端，以免每次調用函式時都必須初始化該用戶端。初始化 SDK 用戶端之後，您可以使用它來與其他 AWS 服務互動。此範例程式碼會呼叫 Amazon S3 `PutObject` API，如下所示：

```
var putRequest = new PutObjectRequest
{
    BucketName = bucketName,
    Key = key,
    ContentBody = receiptContent,
    ContentType = "text/plain"
};

await s3Client.PutObjectAsync(putRequest);
```

存取環境變數

在處理常式程式碼中，您可以使用 `System.Environment.GetEnvironmentVariable` 方法來引用任何環境變數。在此範例中，我們使用下列程式碼行來引用定義的 `RECEIPT_BUCKET` 環境變數：

```
string? bucketName = Environment.GetEnvironmentVariable("RECEIPT_BUCKET");
if (string.IsNullOrWhiteSpace(bucketName))
{
    throw new ArgumentException("RECEIPT_BUCKET environment variable is not set");
}
```

使用全域狀態

在首次調用函數之前，Lambda 會在初始化階段執行靜態程式碼和類別建構函數。在初始化期間建立的資源會在調用間隔期間保留在記憶體中，無需您在每次調用函式時都建立這些資源。

在範例程式碼中，S3 用戶端初始化程式碼位於主要處理常式方法外部。執行時期會在函式處理其第一個事件之前初始化用戶端，這可能導致處理時間延長。但後續事件的處理速度會大幅提升，因為 Lambda 無需再次初始化用戶端。

使用 Lambda Annotations 架構簡化函數程式碼

Lambda Annotations 是一套適用於 .NET 8 的架構，簡化了使用 C# 編寫 Lambda 函式的程序。Annotations 架構使用原始碼產生器來產生程式碼，將 Lambda 程式設計模型轉換為簡化的程式碼。透過 Annotations 架構，您可以取代使用一般程式設計模型編寫的大部分 Lambda 函數程式碼。

使用此架構編寫的程式碼使用更簡單的表達式，讓您可專注於商業邏輯。如需相關範例，請參閱 nuget 文件中的 Amazon.Lambda.Annotations。

如需使用 Lambda Annotations 的完整應用程式範例，請參閱 awsdocs/aws-doc-sdk-examples GitHub 儲存庫中的 PhotoAssetManager 範例。PamApiAnnotations 目錄中的主要 Function.cs 檔案使用 Lambda Annotations。為了進行比較，PamApi 目錄包含使用常規 Lambda 程式設計模型編寫的對等檔案。

使用 Lambda Annotations 架構進行相依性插入

您也可以透過 Lambda Annotations 架構，使用熟悉的語法將相依性插入新增至 Lambda 函數。將 [LambdaStartup] 屬性新增至 Startup.cs 檔案時，Lambda Annotations 架構會在編譯時產生所需的程式碼。

```
[LambdaStartup]
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IDatabaseRepository, DatabaseRepository>();
    }
}
```

Lambda 函數可以使用建構函數插入來插入服務，或使用 [FromServices] 屬性插入個別方法中。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function(IDatabaseRepository repo)
    {
        this._repo = repo;
    }

    [LambdaFunction]
    [HttpApi(LambdaHttpMethod.Get, "/product/{id}")]
```

```
    public async Task<Product> FunctionHandler([FromServices] IDatabaseRepository
 repository, string id)
    {
        return await this._repo.GetById(id);
    }
}
```

C# Lambda 函數的程式碼最佳實務

請遵循下列清單中的準則，在建置 Lambda 函數時使用最佳編碼實務：

- 區隔 Lambda 處理常式與您的核心邏輯。能允許您製作更多可測單位的函式。
- 控制函數部署套件內的相依性。 AWS Lambda 執行環境包含多個程式庫。若要啟用最新的一組功能與安全更新，Lambda 會定期更新這些程式庫。這些更新可能會為您的 Lambda 函數行為帶來細微的變更。若要完全掌控您函式所使用的相依性，請利用部署套件封裝您的所有相依性。
- 最小化依存項目的複雜性。偏好更簡易的框架，其可快速在執行環境啟動時載入。
- 將部署套件最小化至執行時間所必要的套件大小。這能減少您的部署套件被下載與呼叫前解壓縮的時間。對於以 .NET 撰寫的函數，請避免上傳整個 AWS SDK 程式庫做為部署套件的一部分。或者，選擇性倚賴取得您需要的軟體開發套件元件的模組 (例如 DynamoDB、Amazon S3 開發套件模組，以及 Lambda 核心程式庫)。

請利用執行環境重新使用來改看函式的效能。在函式處理常式之外初始化 SDK 用戶端和資料庫連線，並在本機快取 `/tmp` 目錄中的靜態資產。由您函式的相同執行個體處理的後續叫用可以重複使用這些資源。這可藉由減少函數執行時間來節省成本。

若要避免叫用間洩漏潛在資料，請不要使用執行環境來儲存使用者資料、事件，或其他牽涉安全性的資訊。如果您的函式依賴無法存放在處理常式內記憶體中的可變狀態，請考慮為每個使用者建立個別函式或個別函式版本。

使用 Keep-Alive 指令維持持續連線的狀態。Lambda 會隨著時間的推移清除閒置連線。叫用函數時嘗試重複使用閒置連線將導致連線錯誤。若要維護持續連線，請使用與執行階段相關聯的 keep-alive (保持啟用) 指令。如需範例，請參閱在 Node.js 中重複使用 Keep-Alive 的連線。

使用環境變數將操作參數傳遞給您的函數。例如，如果您正在寫入到 Amazon S3 儲存貯體，而非對您正在寫入的儲存貯體名稱進行硬式編碼，請將儲存貯體名稱設定為環境變數。

避免在 Lambda 函數中使用遞迴調用，其中函數會調用自己或啟動可能再次調用函數的程序。這會導致意外的函式呼叫量與升高的成本。若您看到意外的調用數量，當更新程式碼時，請立刻將函數的預留並行設為 `0`，以調節對函數的所有調用。

請勿在您的 Lambda 函數程式碼中使用未記錄的非公有 API。對於 AWS Lambda 受管執行期，Lambda 會定期將安全性和功能更新套用至 Lambda 的內部 APIs。這些內部 API 更新可能是向後不相容的，這會導致意外結果，例如若您的函數依賴於這些非公有 API，則叫用失敗。請參閱 API 參考查看公開可用 API 的清單。

撰寫等冪程式碼。為函數撰寫等冪程式碼可確保採用相同方式來處理重複事件。程式碼應正確驗證事件並正常處理重複的事件。如需詳細資訊，請參閱 How do I make my Lambda function idempotent? (如何讓 Lambda 函數等冪？)。

使用 .zip 封存檔建置和部署 C# Lambda 函數

.NET 部署套件 (.zip 封存檔) 包含函數的編譯組件，及其所有的組件相依項。該套件也包含 *proj*`.deps.json` 檔案。這會對 .NET 執行期發出訊號，告知函數的所有相依項和用於設定執行期的 *proj*`.runtimeconfig.json` 檔案。

若要部署個別 Lambda 函數，您可以使用 `Amazon.Lambda.Tools` .NET Lambda Global CLI。若使用 `dotnet lambda deploy-function` 命令，系統會自動建立 .zip 部署套件並部署至 Lambda。不過，我們建議您使用 AWS Serverless Application Model (AWS SAM) 或 AWS Cloud Development Kit (AWS CDK)，將 .NET 應用程式部署到 AWS。

無伺服器應用程式通常包含 Lambda 函數和其他受管 AWS 服務的組合，搭配使用以執行特定業務工作。AWS SAM 和 AWS CDK 簡化了透過其他 AWS 服務大規模建置和部署 Lambda 函數的作業。AWS SAM 範本規範提供了簡單而清晰的語法，用於描述構成無伺服器應用程式的 Lambda 函數、API、許可、組態和其他 AWS 資源。您可以使用 AWS CDK 將雲端基礎架構定義為程式碼，協助您運用 .NET 等現代程式設計語言，在雲端建置可靠、可擴展且經濟實惠的應用程式。AWS CDK 和 AWS SAM 都是使用 .NET Lambda Global CLI 來封裝函數。

雖然可以使用 .NET Core CLI，以 C# 函數使用 Lambda 層，但建議不要這麼做。使用層的 C# 函數會在 初始化階段 中手動載入共用組件，這可能會增加冷啟動時間。相反，在編譯時納入所有共用程式碼，避免在執行時期載入組件對效能產生影響。

如需使用 AWS SAM、AWS CDK 和 .NET Lambda Global CLI 建置和部署 .NET Lambda 函數的相關說明，請參閱以下各節。

主題

- 使用 .NET Lambda Global CLI
- 使用 AWS SAM 部署 C# Lambda 函數
- 使用 AWS CDK 部署 C# Lambda 函數
- 部署 ASP.NET 應用程式

使用 .NET Lambda Global CLI

透過 .NET CLI 和 .NET Lambda Global Tools 延伸模組 (`Amazon.Lambda.Tools`)，可以跨平台建立 .NET 式 Lambda 應用程式、封裝這些應用程式，然後部署到 Lambda。在本節中，您將學習如何使用 .NET CLI 和 Amazon Lambda 範本建立新的 Lambda .NET 專案，以及如何使用 `Amazon.Lambda.Tools` 封裝和部署這些專案。

主題

- 必要條件
- 使用 .NET CLI 建立 .NET 專案
- 使用 .NET CLI 部署 .NET 專案
- 透過 .NET CLI 使用 Lambda 層

必要條件

.NET 8 SDK

如果尚未安裝，請安裝 .NET 8 SDK 和執行時期。

AWS Amazon.Lambda.Templates .NET 專案範本

若要產生 Lambda 函數程式碼，請使用 Amazon.Lambda.Templates NuGet 套件。若要安裝此範本套件，請執行下列命令：

```
dotnet new install Amazon.Lambda.Templates
```

AWS Amazon.Lambda.Tools .NET Global CLI 工具

若要建立 Lambda 函數，請使用 Amazon.Lambda.Tools .NET Core Global Tools 延伸模組。若要安裝 Amazon.Lambda.Tools，請執行下列命令：

```
dotnet tool install -g Amazon.Lambda.Tools
```

如需有關 Amazon.Lambda.Tools .NET CLI 延伸模組的詳細資訊，請參閱 GitHub 上的適用於 .NET CLI 的 AWS 延伸模組儲存庫。

使用 .NET CLI 建立 .NET 專案

在 .NET CLI 中，在命令列中使用 `dotnet new` 命令建立 .NET 專案。Lambda 提供使用 `Amazon.Lambda.Templates` NuGet 套件的其他範本。

安裝此套件後，執行以下命令即可查看可用範本的清單。

```
dotnet new list
```

若要檢查範本的詳細資訊，請使用 help 選項。例如，若要查看有關 lambda.EmptyFunction 範本的詳細資訊，請執行以下命令。

```
dotnet new lambda.EmptyFunction --help
```

若要建立 .NET Lambda 函數的基本範本，請使用 lambda.EmptyFunction 範本。這會建立一個簡單的函數，此函數會接受字串作為輸入，並使用 ToUpper 方法轉換為大寫。此範本支援以下選項：

- --name - 函數的名稱。
- --region：要建立函數的 AWS 區域。
- --profile：您的 適用於 .NET 的 AWS SDK 憑證檔中的設定檔名稱。若要深入了解 .NET 中的憑證設定檔，請參閱《適用於 .NET 的 AWS SDK 開發人員指南》中的設定 AWS 憑證。

在此範例中，我們使用預設設定檔和 AWS 區域 設定建立了新的空函數 myDotnetFunction：

```
dotnet new lambda.EmptyFunction --name myDotnetFunction
```

此命令會在您的專案目錄中建立以下檔案和目錄。

```
### myDotnetFunction
    ### src
    #   ### myDotnetFunction
    #       ### Function.cs
    #       ### Readme.md
    #       ### aws-lambda-tools-defaults.json
    #       ### myDotnetFunction.csproj
    ### test
        ### myDotnetFunction.Tests
            ### FunctionTest.cs
            ### myDotnetFunction.Tests.csproj
```

在 src/myDotnetFunction 目錄下，檢查下列檔案：

- aws-lambda-tools-defaults.json：您在部署 Lambda 函數時，在此檔案中指定命令列選項。例如：

```
  "profile" : "default",
  "region" : "us-east-2",
  "configuration" : "Release",
  "function-architecture": "x86_64",
```

```
  "function-runtime":"dotnet8",
  "function-memory-size" : 256,
  "function-timeout" : 30,
  "function-handler" : "myDotnetFunction::myDotnetFunction.Function::FunctionHandler"
```

- Function.cs：您的 Lambda 處理常式函數程式碼。它是 C# 範本，包含了預設的 `Amazon.Lambda.Core` 程式庫和預設的 `LambdaSerializer` 屬性。如需關於序列化需求及選項的詳細資訊，請參閱C# Lambda 函式中的序列化。它也包含可編輯以套用 Lambda 函數程式碼的範例函數。

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted into
 a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializ

namespace myDotnetFunction;

public class Function
{

    /// <summary>
    /// A simple function that takes a string and does a ToUpper
    /// </summary#
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
    public string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}
```

- myDotnetFunction.csproj：MSBuild 檔案，列出構成應用程式的檔案和組件。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
```

```
    <!-- This property makes the build directory similar to a publish directory and
 helps the AWS .NET Lambda Mock Test Tool find project dependencies. -->
    <CopyLocalLockFileAssemblies>true</CopyLocalLockFileAssemblies>
    <!-- Generate ready to run images during publishing to improve cold start time.
 -->
    <PublishReadyToRun>true</PublishReadyToRun>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.SystemTextJson"
 Version="2.4.0" />
  </ItemGroup>
</Project>
```

- Readme：使用此檔案來記錄您的 Lambda 函數。

在 `myfunction/test` 目錄下，檢查下列檔案：

- myDotnetFunction.Tests.csproj：如上所述，這是 MSBuild 檔案，其列出了構成您的測試專案的檔案和組件。也請注意，它包含 `Amazon.Lambda.Core` 程式庫，以便您順利整合測試函數時所需要的任何 Lambda 範本。

```
<Project Sdk="Microsoft.NET.Sdk">
   ...

    <PackageReference Include="Amazon.Lambda.Core" Version="2.2.0 " />
   ...
```

- FunctionTest.cs：包含在 `src` 目錄中的相同的 C# 程式碼範本檔案。編輯此檔案以鏡像您的函數的生產程式碼，並在上傳 Lambda 函數至生產環境之前進行測試。

```
using Xunit;
using Amazon.Lambda.Core;
using Amazon.Lambda.TestUtilities;

using MyFunction;

namespace MyFunction.Tests
{
    public class FunctionTest
    {
        [Fact]
```

```
        public void TestToUpperFunction()
        {

            // Invoke the lambda function and confirm the string was upper cased.
            var function = new Function();
            var context = new TestLambdaContext();
            var upperCase = function.FunctionHandler("hello world", context);

            Assert.Equal("HELLO WORLD", upperCase);
        }
    }
}
```

使用 .NET CLI 部署 .NET 專案

若要建置部署套件並部署到 Lambda，請使用 `Amazon.Lambda.Tools` CLI 工具。若要從先前步驟中建立的檔案部署函數，請先瀏覽至包含函數 `.csproj` 檔案的資料夾。

```
cd myDotnetFunction/src/myDotnetFunction
```

若要將程式碼以 .zip 部署套件形式部署至 Lambda，請執行下列命令。選擇您的函數名稱。

```
dotnet lambda deploy-function myDotnetFunction
```

在部署期間，精靈會要求您選取 the section called “執行角色 (函數存取其他資源的許可)”。在此範例中，請選取 `lambda_basic_role`。

部署函數後，您可以使用 `dotnet lambda invoke-function` 命令在雲端進行測試。在 `lambda.EmptyFunction` 範本的程式碼範例中，您可以使用 `--payload` 選項傳入字串來測試函數。

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
```

如果函數部署成功，您應該會看到類似以下內容的輸出。

```
dotnet lambda invoke-function myDotnetFunction --payload "Just checking if everything
 is OK"
Amazon Lambda Tools for .NET Core applications (5.8.0)
```

```
Project Home: https://github.com/aws/aws-extensions-for-dotnet-cli, https://github.com/
aws/aws-lambda-dotnet

Payload:
"JUST CHECKING IF EVERYTHING IS OK"

Log Tail:
START RequestId: id Version: $LATEST
END RequestId: id
REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
 Size: 256 MB     Max Memory Used: 12 MB
```

透過 .NET CLI 使用 Lambda 層

> **Note**
>
> 雖然在 .NET 中可以將層與函式搭配使用，但建議不要這麼做。.NET 中使用層的函式會在 `Init` 階段手動將共用組件載入到記憶體中，這可能會增加冷啟動時間。請改在編譯時加入所有共用程式碼，利用 .NET 編譯器的內建最佳化功能。

.NET CLI 支援命令，協助您發布層及部署使用層的 C# 函數。若要將層發布至指定的 Amazon S3 儲存貯體，請在與 `.csproj` 檔案相同的目錄中執行以下命令：

```
dotnet lambda publish-layer <layer_name> --layer-type runtime-package-store --s3-
bucket <s3_bucket_name>
```

接著使用 .NET CLI 部署函數時，請在以下命令中指定要使用的層 ARN：

```
dotnet lambda deploy-function <function_name> --function-layers arn:aws:lambda:us-
east-1:123456789012:layer:layer-name:1
```

如需 Hello World 函數的完整範例，請參閱 blank-csharp-with-layer 範例。

使用 AWS SAM 部署 C# Lambda 函數

AWS Serverless Application Model(AWS SAM) 是一項工具組，可協助簡化在 AWS 上建置和執行無伺服器應用程式的程序。您可以在 YAML 或 JSON 範本中定義應用程式的資源，並使用 AWS SAM 命令列界面 (AWS SAM CLI) 來建置、封裝及部署應用程式。使用 AWS SAM 範本建置 Lambda 函數時，AWS SAM 會使用函數程式碼和您指定的任何相依項，自動建立 .zip 部署套件或容器映像檔。接

著 AWS SAM 會使用 CloudFormation 堆疊部署函數。若要進一步了解如何使用 AWS SAM 建置和部署 Lambda 函數，請參閱《AWS Serverless Application Model 開發人員指南》中的 AWS SAM入門。

下列步驟說明如何使用 AWS SAM 來下載、建置和部署範例 .NET Hello World 應用程式。此範例應用程式使用 Lambda 函數和 Amazon API Gateway 端點來實作基本 API 後端。當您傳送 HTTP GET 請求至 API Gateway 端點時，API Gateway 端點會調用您的 Lambda 函數。此函數會傳回「hello world」訊息，以及處理您請求之 Lambda 函數執行個體的 IP 地址。

當您使用 AWS SAM 建置和部署應用程式時，AWS SAM CLI 會在背景使用 `dotnet lambda package` 命令封裝個別 Lambda 函數程式碼套件。

必要條件

.NET 8 SDK

安裝 .NET 8 SDK 和執行時期。

AWS SAM CLI 1.39 版或更新版本

若要了解如何安裝最新版 AWS SAM CLI，請參閱安裝 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用以下命令，初始化使用 Hello world .NET 範本的應用程式。

```
sam init --app-template hello-world --name sam-app \
--package-type Zip --runtime dotnet8
```

此命令會在您的專案目錄中建立以下檔案和目錄。

```
### sam-app
    ### README.md
    ### events
    #   ### event.json
    ### omnisharp.json
    ### samconfig.toml
    ### src
    #   ### HelloWorld
    #       ### Function.cs
    #       ### HelloWorld.csproj
    #       ### aws-lambda-tools-defaults.json
    ### template.yaml
```

```
    ### test
        ### HelloWorld.Test
            ### FunctionTest.cs
            ### HelloWorld.Tests.csproj
```

2. 瀏覽至包含 `template.yaml file` 的目錄。此檔案是為應用程式定義 AWS 資源的範本，包括 Lambda 函數和 API Gateway API。

```
cd sam-app
```

3. 若要建置應用程式的原始碼，請執行下列命令。

```
sam build
```

4. 若要將應用程式部署至 AWS，請執行下列命令。

```
sam deploy --guided
```

此命令會透過下列一系列提示封裝並部署應用程式。若要接受預設選項，請按 Enter。

> **Note**
>
> 若顯示 HelloWorldFunction 可能未定義授權，仍要繼續嗎?，請務必輸入 y。

- 堆疊名稱：要部署至 CloudFormation 的堆疊名稱。此名稱對您的 AWS 帳戶和 AWS 區域而言都必須是唯一的。
- AWS 區域：您想要部署應用程式的目標 AWS 區域。
- 部署前確認變更：選取是，即可在 AWS SAM 部署應用程式變更之前手動檢閱任何變更集。如果選取否，AWS SAM CLI 會自動部署應用程式變更。
- 允許建立 SAM CLI IAM 角色：許多 AWS SAM 範本 (包括本範例中的 Hello world 範本) 都會建立 AWS Identity and Access Management (IAM) 角色，以授予 Lambda 函數存取其他 AWS 服務 的許可。選取「是」以提供許可，允許部署建立或修改 IAM 角色的 CloudFormation 堆疊。
- 停用復原：依照預設，如果在建立或部署堆疊期間 AWS SAM 發生錯誤，便會將堆疊復原至先前的版本。選取「否」以接受此預設設定。
- HelloWorldFunction 可能未定義授權，仍要繼續嗎：請輸入 y。
- 將引數儲存至 samconfig.toml：選取是以儲存您選擇的組態。您之後可以在沒有參數的情況下重新執行 `sam deploy`，以將變更部署到應用程式。

5. 應用程式部署完成後，CLI 會傳回 Hello World Lambda 函數的 Amazon Resource Name (ARN)，以及為應用程式建立的 IAM 角色。此外也會顯示您的 API Gateway API 端點。若要測試應用程式，請在瀏覽器中開啟端點。您應該會看到類似以下內容的回應。

```
{"message":"hello world","location":"34.244.135.203"}
```

6. 若要刪除資源，請執行下列命令。請注意，您建立的 API 端點，是可以透過網際網路存取的公有端點。建議您在測試後刪除此端點。

```
sam delete
```

後續步驟

若要進一步了解使用 AWS SAM 來建置和部署使用 .NET 的 Lambda 函數，請參閱下列資源：

- AWS Serverless Application Model (AWS SAM) 開發人員指南
- 使用 AWS Lambda 和 SAM CLI 建置無伺服器 .NET 應用程式

使用 AWS CDK 部署 C# Lambda 函數

AWS Cloud Development Kit (AWS CDK) 是開放原始碼軟體開發架構，用於透過現代程式設計語言和 .NET 等架構，將雲端基礎架構定義為程式碼。系統會執行 AWS CDK 專案以產生 CloudFormation 範本，然後用來部署程式碼。

若要使用 AWS CDK 建置和部署範例 Hello world .NET 應用程式，請依照下列各節的指示操作。範例應用程式會實作基本 API 後端 (包含 API Gateway 端點和 Lambda 函數)。當您傳送 HTTP GET 請求至端點時，API Gateway 會調用 Lambda 函數 此函數會傳回 Hello world 訊息，以及處理您請求之 Lambda 執行個體的 IP 地址。

必要條件

.NET 8 SDK

安裝 .NET 8 SDK 和執行時期。

AWS CDK 第 2 版

若要了解如何安裝最新版本的 AWS CDK，請參閱《AWS Cloud Development Kit (AWS CDK) v2 開發人員指南》中的開始使用 AWS CDK。

部署範例 AWS CDK 應用程式

1. 為範例應用程式建立專案目錄，並瀏覽至該目錄。

```
mkdir hello-world
cd hello-world
```

2. 執行下列命令以初始化新的 AWS CDK 應用程式。

```
cdk init app --language csharp
```

此命令會在您的專案目錄中建立以下檔案和目錄

```
### README.md
### cdk.json
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
```

3. 開啟 `src` 目錄，並使用 .NET CLI 建立新的 Lambda 函數。這是您將使用 AWS CDK 部署的功能。在此範例中，您將使用 `lambda.EmptyFunction` 範本建立名為 `HelloWorldLambda` 的 Hello world 函數。

```
cd src
dotnet new lambda.EmptyFunction -n HelloWorldLambda
```

完成此步驟後，專案目錄中的目錄結構應如下所示。

```
### README.md
### cdk.json
### src
    ### HelloWorld
    #   ### GlobalSuppressions.cs
    #   ### HelloWorld.csproj
    #   ### HelloWorldStack.cs
    #   ### Program.cs
    ### HelloWorld.sln
```

```
    ### HelloWorldLambda
        ### src
        #   ### HelloWorldLambda
        #       ### Function.cs
        #       ### HelloWorldLambda.csproj
        #       ### Readme.md
        #       ### aws-lambda-tools-defaults.json
        ### test
            ### HelloWorldLambda.Tests
                ### FunctionTest.cs
                ### HelloWorldLambda.Tests.csproj
```

4. 在 src/HelloWorld 目錄中，開啟 HelloWorldStack.cs 檔案。將檔案的內容取代為下列程式碼。

```
using Amazon.CDK;
using Amazon.CDK.AWS.Lambda;
using Amazon.CDK.AWS.Logs;
using Constructs;

namespace CdkTest
{
    public class HelloWorldStack : Stack
    {
        internal HelloWorldStack(Construct scope, string id, IStackProps props =
 null) : base(scope, id, props)
        {
            var buildOption = new BundlingOptions()
            {
                Image = Runtime.DOTNET_8.BundlingImage,
                User = "root",
                OutputType = BundlingOutput.ARCHIVED,
                Command = new string[]{
            "/bin/sh",
                "-c",
                " dotnet tool install -g Amazon.Lambda.Tools"+
                " && dotnet build"+
                " && dotnet lambda package --output-package /asset-output/
function.zip"
                }
            };
```

```
             var helloWorldLambdaFunction = new Function(this,
 "HelloWorldFunction", new FunctionProps
            {
                Runtime = Runtime.DOTNET_8,
                MemorySize = 1024,
                LogRetention = RetentionDays.ONE_DAY,
                Handler =
 "HelloWorldLambda::HelloWorldLambda.Function::FunctionHandler",
                Code = Code.FromAsset("./src/HelloWorldLambda/src/
HelloWorldLambda", new Amazon.CDK.AWS.S3.Assets.AssetOptions
                {
                    Bundling = buildOption
                }),
            });
        }
    }
}
```

這個程式碼是用來編譯和綁定應用程式程式碼，以及 Lambda 函數本身的定義。BundlingOptions 物件可建立一個 zip 檔及一組用於產生 zip 檔內容的命令。在此情況中，`dotnet lambda package` 命令用於編譯和產生 zip 檔。

5. 若要部署應用程式，請執行下列命令。

```
cdk deploy
```

6. 使用 .NET Lambda CLI 調用已部署的 Lambda 函數。

```
dotnet lambda invoke-function HelloWorldFunction -p "hello world"
```

7. 完成測試後，除非您想要保留建立的資源，否則可直接刪除。若要刪除資源，請執行下列命令。

```
cdk destroy
```

後續步驟

若要進一步了解使用 AWS CDK 來建置和部署使用 .NET 的 Lambda 函數，請參閱下列資源：

- 以 C# 使用 AWS CDK
- 使用 AWS CDK 建置、封裝和發布 .NET C# Lambda 函數

部署 ASP.NET 應用程式

除了託管事件驅動型函數之外，您還可以搭配使用 .NET 與 Lambda 來託管輕量型 ASP.NET 應用程式。您可以使用 `Amazon.Lambda.AspNetCoreServer` NuGet 套件來建置和部署 ASP.NET 應用程式。在本節中，您將了解如何使用 .NET Lambda CLI 工具，將 ASP.NET Web API 部署至 Lambda。

主題

- 必要條件
- 將 ASP.NET Web API 部署到 Lambda
- 將 ASP.NET Minimal API 部署到 Lambda

必要條件

.NET 8 SDK

安裝 .NET 8 SDK 和 ASP.NET Core 執行時期。

Amazon.Lambda.Tools

若要建立 Lambda 函數，請使用 Amazon.Lambda.Tools .NET Core Global Tools 延伸模組。若要安裝 Amazon.Lambda.Tools，請執行下列命令：

```
dotnet tool install -g Amazon.Lambda.Tools
```

如需有關 Amazon.Lambda.Tools .NET CLI 延伸模組的詳細資訊，請參閱 GitHub 上的適用於 .NET CLI 的 AWS 延伸模組儲存庫。

Amazon.Lambda.Templates

若要產生 Lambda 函數程式碼，請使用 Amazon.Lambda.Templates NuGet 套件。若要安裝此範本套件，請執行下列命令：

```
dotnet new --install Amazon.Lambda.Templates
```

將 ASP.NET Web API 部署到 Lambda

若要使用 ASP.NET 部署 Web API，您可以使用 .NET Lambda 範本建立新的 Web API 專案。使用下列命令來初始化新的 ASP.NET Web API 專案。在範例命令中，我們將專案命名為 `AspNetOnLambda`。

```
dotnet new serverless.AspNetCoreWebAPI -n AspNetOnLambda
```

此命令會在您的專案目錄中建立以下檔案和目錄。

```
.
### AspNetOnLambda
    ### src
    #   ### AspNetOnLambda
    #       ### AspNetOnLambda.csproj
    #       ### Controllers
    #       #   ### ValuesController.cs
    #       ### LambdaEntryPoint.cs
    #       ### LocalEntryPoint.cs
    #       ### Readme.md
    #       ### Startup.cs
    #       ### appsettings.Development.json
    #       ### appsettings.json
    #       ### aws-lambda-tools-defaults.json
    #       ### serverless.template
    ### test
        ### AspNetOnLambda.Tests
            ### AspNetOnLambda.Tests.csproj
            ### SampleRequests
            #   ### ValuesController-Get.json
            ### ValuesControllerTests.cs
            ### appsettings.json
```

當 Lambda 調用函數時，使用的進入點就是 `LambdaEntryPoint.cs` 檔案。.NET Lambda 範本建立的檔案包含下列程式碼。

```
namespace AspNetOnLambda;

public class LambdaEntryPoint : Amazon.Lambda.AspNetCoreServer.APIGatewayProxyFunction
{
    protected override void Init(IWebHostBuilder builder)
    {
        builder
            .UseStartup#Startup#();
    }

    protected override void Init(IHostBuilder builder)
    {
```

```
    }
}
```

Lambda 使用的進入點，必須繼承自 Amazon.Lambda.AspNetCoreServer 套件中三個基本類別的其中一個。這三個基本類別為：

- APIGatewayProxyFunction
- APIGatewayHttpApiV2ProxyFunction
- ApplicationLoadBalancerFunction

當您使用提供的 .NET Lambda 範本建立 LambdaEntryPoint.cs 檔案時，所使用的預設類別為 APIGatewayProxyFunction。您在函數中使用的基本類別，取決於位於 Lambda 函數前的 API 層。

三個基本類別都包含名為 FunctionHandlerAsync 的公有方法。Lambda 用來調用函數的處理常式字串中，會包含此方法的名稱。FunctionHandlerAsync 方法會將輸入事件承載轉換為正確的 ASP.NET 格式，並將 ASP.NET 回應轉換回 Lambda 回應承載。在顯示的範例 AspNetOnLambda 專案中，處理常式字串如下所示。

```
AspNetOnLambda::AspNetOnLambda.LambdaEntryPoint::FunctionHandlerAsync
```

若要將 API 部署至 Lambda，請執行下列命令，瀏覽至包含原始程式碼檔案的目錄，並使用 CloudFormation 部署函數。

```
cd AspNetOnLambda/src/AspNetOnLambda
dotnet lambda deploy-serverless
```

> **Tip**
>
> 當您使用 **dotnet lambda deploy-serverless** 命令部署 API 時，CloudFormation 會根據您在部署期間指定的堆疊名稱，為 Lambda 函數命名。若要為 Lambda 函數提供自訂名稱，請編輯 serverless.template 檔案，將 FunctionName 屬性新增至 AWS::Serverless::Function 資源。如需進一步了解，請參閱《CloudFormation 使用者指南》中的名稱類型。

將 ASP.NET Minimal API 部署到 Lambda

若要將 ASP.NET Minimal API 部署到 Lambda，您可以使用 .NET Lambda 範本建立新的 Minimal API 專案。使用下列命令來初始化新的 Minimal API 專案。在此範例中，我們將專案命名為 `MinimalApiOnLambda`。

```
dotnet new serverless.AspNetCoreMinimalAPI -n MinimalApiOnLambda
```

此命令會在您的專案目錄中建立以下檔案和目錄。

```
### MinimalApiOnLambda
    ### src
        ### MinimalApiOnLambda
            ### Controllers
            #   ### CalculatorController.cs
            ### MinimalApiOnLambda.csproj
            ### Program.cs
            ### Readme.md
            ### appsettings.Development.json
            ### appsettings.json
            ### aws-lambda-tools-defaults.json
            ### serverless.template
```

`Program.cs` 檔案包含下列程式碼。

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllers();

// Add AWS Lambda support. When application is run in Lambda Kestrel is swapped out as
 the web server with Amazon.Lambda.AspNetCoreServer. This
// package will act as the webserver translating request and responses between the
 Lambda event source and ASP.NET Core.
builder.Services.AddAWSLambdaHosting(LambdaEventSource.RestApi);

var app = builder.Build();


app.UseHttpsRedirection();
app.UseAuthorization();
app.MapControllers();
```

```
app.MapGet("/", () => "Welcome to running ASP.NET Core Minimal API on AWS Lambda");

app.Run();
```

若要將 Minimal API 設為在 Lambda 上執行，您可能需要編輯此程式碼，讓 Lambda 和 ASP.NET Core 之間的請求和回應能正確轉譯。根據預設，函數會針對 REST API 事件來源加以設定。若使用 HTTP API 或 Application Load Balancer，請以下列其中一個選項取代 `(LambdaEventSource.RestApi)`：

- `(LambdaEventSource.HttpAPi)`
- `(LambdaEventSource.ApplicationLoadBalancer)`

若要將 API 部署至 Lambda，請執行下列命令，瀏覽至包含原始程式碼檔案的目錄，並使用 CloudFormation 部署函數。

```
cd MinimalApiOnLambda/src/MinimalApiOnLambda
dotnet lambda deploy-serverless
```

使用 .NET Lambda 函式層

我們不建議使用層來管理在 .NET 中編寫的 Lambda 函式的相依項。由於 .NET 是編譯型語言，因此函式仍須在 Init 階段將所有共用組件手動載入記憶體中，這可能會增加冷啟動時間。使用層不僅會使部署程序複雜化，還會阻礙您利用內建編譯器最佳化功能。

若要在 .NET 處理常式中使用外部相依項，請在編譯時直接將其納入部署套件。如此一來，您既能簡化部署程序，又能利用內建的 .NET 編譯器最佳化功能。如需如何在函式中匯入與使用 NuGet 等相依項的範例，請參閱the section called “處理常式”。

使用容器映像部署 .NET Lambda 函數

您可以透過三種方式為 .NET Lambda 函數建置容器映像：

- 使用 .NET AWS 的基礎映像

  AWS 基礎映像會預先載入語言執行期、用來管理 Lambda 與函數程式碼之間互動的執行期界面用戶端，以及用於本機測試的執行期界面模擬器。

- 使用僅限 AWS 作業系統的基礎映像

  AWS 僅限作業系統的基礎映像包含 Amazon Linux 發行版本和執行時間界面模擬器。這些映像常用於為編譯語言 (如 Go 和 Rust) 和 Lambda 不提供基礎映像的語言或語言版本 (如 Node.js 19) 建置容器映像。您還可以使用僅限作業系統的基礎映像來實作自訂執行期。若要使映像檔與 Lambda 相容，您必須在映像中加入 適用於 .NET 的執行期介面用戶端。

- 使用非AWS 基礎映像

  您可以使用其他容器登錄檔中的替代基礎映像 (例如 Alpine Linux 或 Debian)。您也可以使用組織建立的自訂映像。若要使映像檔與 Lambda 相容，您必須在映像中加入 適用於 .NET 的執行期介面用戶端。

> **Tip**
>
> 若要縮短 Lambda 容器函數變成作用中狀態所需的時間，請參閱 Docker 文件中的使用多階段建置。若要建置有效率的容器映像，請遵循撰寫 Dockerfiles 的最佳實務。

本頁面會說明如何為 Lambda 建置、測試和部署容器映像。

主題

- AWS .NET 的基礎映像
- 使用 .NET AWS 的基礎映像
- 透過執行期介面用戶端使用替代基礎映像

AWS .NET 的基礎映像

AWS 為 .NET 提供下列基礎映像：

Tags (標籤)	執行期	作業系統	Dockerfile	棄用
10	.NET 10	Amazon Linux 2023	GitHub 上適用於 .NET 10 的 Dockerfile	2028 年 11 月 14 日
9	.NET 9	Amazon Linux 2023	GitHub 上適用於 .NET 9 的 Dockerfile	2026 年 11 月 10 日
8	.NET 8	Amazon Linux 2023	Dockerfile for .NET 8 on GitHub	2026 年 11 月 10 日

Amazon ECR 儲存庫：gallery.ecr.aws/lambda/dotnet

使用 .NET AWS 的基礎映像

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- .NET SDK – 下列步驟使用 .NET 8 基礎映像。確保您的 .NET 版本與您在 Dockerfile 中指定的基礎映像的版本相符。
- Docker (最低版本 25.0.0)
- Docker buildx 外掛程式。

使用基礎映像建立和部署映像

在下面的步驟中，您可以使用 Amazon.Lambda.Templates 和 Amazon.Lambda.Tools 來建立 .NET 專案。然後，您可以建置 Docker 映像檔，將該映像檔上傳到 Amazon ECR，然後將其部署到 Lambda 函數。

1. 安裝 Amazon.Lambda.Templates NuGet 套件。

```
dotnet new install Amazon.Lambda.Templates
```

2. 使用 `lambda.image.EmptyFunction` 範本建立 .NET 專案。

```
dotnet new lambda.image.EmptyFunction --name MyFunction --region us-east-1
```

專案檔案儲存在 *MyFunction*/src/*MyFunction* 目錄中：

- aws-lambda-tools-defaults.json：指定用於部署 Lambda 函式的命令列選項。
- Function.cs：您的 Lambda 處理常式函數程式碼。這是 C# 範本，包含了預設的 Amazon.Lambda.Core 程式庫和預設的 LambdaSerializer 屬性。如需有關序列化需求及選項的詳細資訊，請參閱 C# Lambda 函式中的序列化。可以使用提供的程式碼進行測試，也可以將其替換為您自己的程式碼。
- MyFunction.csproj – .NET 專案檔案，列出了構成應用程式的檔案與組件。
- Dockerfile：您可以使用提供的 Dockerfile 進行測試，也可以將其取代為自訂的 Dockerfile。如果使用自己的，請確保：
  - 將 FROM 屬性設定為基礎映像的 URI。基礎映像和 MyFunction.csproj 檔案中的 TargetFramework 都必須使用相同的 .NET 版本。例如，若要使用 .NET 9：
    - Dockerfile：FROM *public.ecr.aws/lambda/dotnet:9*
    - MyFunction.csproj：<TargetFramework>*net9.0*</TargetFramework>
  - 將 CMD 引數設定為 Lambda 函數處理常式。這應符合 aws-lambda-tools-defaults.json 中的 image-command。

3. 安裝 Amazon.Lambda.Tools .NET Global Tool。

```
dotnet tool install -g Amazon.Lambda.Tools
```

如果已安裝 Amazon.Lambda.Tools，則請確保您有最新版本。

```
dotnet tool update -g Amazon.Lambda.Tools
```

4. 如果您尚未這麼做，請將目錄變更為 *MyFunction*/src/*MyFunction*。

```
cd src/MyFunction
```

5. 使用 Amazon.Lambda.Tools 建置 Docker 映像檔，將其推送至新的 Amazon ECR 儲存庫，然後部署 Lambda 函數。

   對於 --function-role，指定函數執行角色的角色名稱 (而非 Amazon Resource Name (ARN))。例如 lambda-role。

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

如需有關 Amazon.Lambda.Tools .NET Global Tool 的詳細資訊，請參閱 GitHub 上的 .NET CLI 的AWS 延伸模組儲存庫。

6. 調用函數。

```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
```

如果一切順利，您會看到類似如下內容的回應：

```
Payload:
{"Lower":"testing the function","Upper":"TESTING THE FUNCTION"}

Log Tail:
INIT_REPORT Init Duration: 9999.81 ms   Phase: init     Status: timeout
START RequestId: 12378346-f302-419b-b1f2-deaa1e8423ed Version: $LATEST
END RequestId: 12378346-f302-419b-b1f2-deaa1e8423ed
REPORT RequestId: 12378346-f302-419b-b1f2-deaa1e8423ed  Duration: 3173.06 ms
 Billed Duration: 3174 ms        Memory Size: 512 MB     Max Memory Used: 24 MB
```

7. 刪除 Lambda 函數。

```
dotnet lambda delete-function MyFunction
```

透過執行期介面用戶端使用替代基礎映像

如果您使用僅限作業系統的基礎映像或替代的基礎映像，則必須在映像中加入執行期介面用戶端。執行期介面用戶端會讓您擴充 Runtime API，管理 Lambda 與函數程式碼之間的互動。

下列範例示範如何使用非AWS 基礎映像為 .NET 建置容器映像，以及如何新增 Amazon.Lambda.RuntimeSupport 套件，這是 .NET 的 Lambda 執行期介面用戶端。範例 Dockerfile 使用了 Microsoft .NET 8 基礎映像。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- .NET SDK – 下列步驟使用 .NET 9 基礎映像。確保您的 .NET 版本與您在 Dockerfile 中指定的基礎映像的版本相符。
- Docker (最低版本 25.0.0)
- Docker buildx 外掛程式。

使用替代基礎映像建立和部署映像

1. 安裝 Amazon.Lambda.Templates NuGet 套件。

```
dotnet new install Amazon.Lambda.Templates
```

2. 使用 `lambda.CustomRuntimeFunction` 範本建立 .NET 專案。此範本包含 Amazon.Lambda.RuntimeSupport 套件。

```
dotnet new lambda.CustomRuntimeFunction --name MyFunction --region us-east-1
```

3. 導覽至 *MyFunction*/src/*MyFunction* 目錄。這是儲存專案檔案的位置。檢查下列檔案：
   - aws-lambda-tools-defaults.json – 您在部署 Lambda 函數時，在此檔案中指定命令列選項。
   - Function.cs：此程式碼含有一個類別，其中包含可將 `Amazon.Lambda.RuntimeSupport` 程式庫初始化為自舉的 `Main` 方法。`Main` 方法是函數處理過程的進入點。`Main` 方法會將函數處理常式包裝在自舉可以使用的包裝函式之中。如需詳細資訊，請參閱 GitHub 儲存庫中的 Using Amazon.Lambda.RuntimeSupport as a class library。
   - MyFunction.csproj – 一個 .NET 專案檔案，列出了構成您的應用程式的檔案和組件。
   - Readme.md：此檔案包含有關範例 Lambda 函數的詳細資訊。
4. 開啓 `aws-lambda-tools-defaults.json` 檔案並「新增」下列幾行程式碼：

```
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
```

   - package-type：將部署套件定義為容器映像。
   - docker-host-build-output-dir：為建置過程設定輸出目錄。

   Example aws-lambda-tools-defaults.json

```
{
```

```
  "Information": [
    "This file provides default values for the deployment wizard inside Visual
 Studio and the AWS Lambda commands added to the .NET Core CLI.",
    "To learn more about the Lambda commands with the .NET Core CLI execute the
 following command at the command line in the project root directory.",
    "dotnet lambda help",
    "All the command line options for the Lambda command can be specified in this
 file."
  ],
  "profile": "",
  "region": "us-east-1",
  "configuration": "Release",
  "function-runtime": "provided.al2023",
  "function-memory-size": 256,
  "function-timeout": 30,
  "function-handler": "bootstrap",
  "msbuild-parameters": "--self-contained true",
  "package-type": "image",
  "docker-host-build-output-dir": "./bin/Release/lambda-publish"
}
```

5. 在 *MyFunction*/src/*MyFunction* 目錄建立 Dockerfile。下列範例 Dockerfile 使用 Microsoft .NET 基礎映像，而非 AWS 基礎映像。

   - 將 FROM 屬性設為基礎映像識別符。基礎映像和 MyFunction.csproj 檔案中的 TargetFramework 都必須使用相同的 .NET 版本。
   - 使用 COPY 命令將函數複製到 /var/task 目錄。
   - 將 ENTRYPOINT 設為您希望 Docker 容器在啟動時執行的模組。在這種情況下，該模組是會初始化 Amazon.Lambda.RuntimeSupport 程式庫的自舉。

   請注意，範例 Dockerfile 不包含 USER 指令。當您將容器映像部署到 Lambda 時，Lambda 會自動定義一個具有最低權限許可的預設 Linux 使用者。這與標準 Docker 行為不同，後者會在未提供 USER 指令時預設為 root 使用者。

   Example Dockerfile

```
# You can also pull these images from DockerHub amazon/aws-lambda-dotnet:8
FROM mcr.microsoft.com/dotnet/runtime:9.0

# Set the image's internal work directory
WORKDIR /var/task
```

```
# Copy function code to Lambda-defined environment variable
COPY "bin/Release/net9.0/linux-x64"  .

# Set the entrypoint to the bootstrap
ENTRYPOINT ["/usr/bin/dotnet", "exec", "/var/task/bootstrap.dll"]
```

6. 安裝 Amazon.Lambda.Tools .NET Global Tools 延伸模組。

```
dotnet tool install -g Amazon.Lambda.Tools
```

如果已安裝 Amazon.Lambda.Tools，則請確保您有最新版本。

```
dotnet tool update -g Amazon.Lambda.Tools
```

7. 使用 Amazon.Lambda.Tools 建置 Docker 映像檔，將其推送至新的 Amazon ECR 儲存庫，然後部署 Lambda 函數。

對於 `--function-role`，指定函數執行角色的角色名稱 (而非 Amazon Resource Name (ARN))。例如 `lambda-role`。

```
dotnet lambda deploy-function MyFunction --function-role lambda-role
```

如需有關 Amazon.Lambda.Tools .NET CLI 延伸模組的詳細資訊，請參閱 GitHub 上的 .NET CLI 的AWS 延伸模組儲存庫。

8. 調用函數。

```
dotnet lambda invoke-function MyFunction --payload "Testing the function"
```

如果一切順利，您會看到下列項目：

```
Payload:
"TESTING THE FUNCTION"

Log Tail:
START RequestId: id Version: $LATEST
END RequestId: id
REPORT RequestId: id  Duration: 0.99 ms       Billed Duration: 1 ms         Memory
 Size: 256 MB     Max Memory Used: 12 MB
```

9. 刪除 Lambda 函數。

```
dotnet lambda delete-function MyFunction
```

將 .NET Lambda 函數程式碼編譯為原生執行時期格式

.NET 8 支援原生預先 (AOT) 編譯。使用原生 AOT，您可以將 Lambda 函數程式碼編譯為原生執行階段格式，這樣便不需要在執行階段編譯 .NET 程式碼。原生 AOT 編譯可縮短以 .NET 撰寫之 Lambda 函數的冷啟動時間。如需詳細資訊，請參閱 AWS 運算部落格上的適用於 AWS Lambda 的 .NET 8 執行時期簡介。

章節

- Lambda 執行時間
- 必要條件
- 開始使用
- 序列化
- 裁剪
- 疑難排解

Lambda 執行時間

若要使用原生 AOT 編譯部署 Lambda 函數建置，請使用受管 .NET 8 Lambda 執行時期。此執行時期支援同時使用 x86_64 和 arm64 架構。

不使用 AOT 來部署 .NET Lambda 函數時，您的應用程式會首先編譯成中繼語言 (IL) 程式碼。在執行時期，Lambda 執行時期中的即時 (JIT) 編譯器需使用 IL 程式碼，並根據需要編譯為機器碼。透過使用原生 AOT 預先編譯的 Lambda 函數，可以在部署函數時將程式碼編譯為機器程式碼，因此在執行之前，您不需要依賴 Lambda 執行時期中的 .NET 執行時期或 SDK 來編譯程式碼。

AOT 的一個限制是，應用程式碼必須在與 .NET 8 執行時期使用的 Amazon Linux 2023 (AL2023) 作業系統相同的環境中進行編譯。.NET Lambda CLI 提供使用 AL2023 映像檔在 Docker 容器中編譯應用程式的功能。

為了避免跨架構相容性的潛在問題，強烈建議您在具有您為函數設定之相同處理器架構的環境中編譯程式碼。若要進一步了解跨架構編譯的限制，請參閱 Microsoft .NET 文件中的跨編譯。

必要條件

Docker

> 若要使用原生 AOT，編譯函數程式碼時所用環境的 AL2023 作業系統必須與 .NET 8 執行時期相同。以下各節中的 .NET CLI 命令，使用 Docker 在 AL2023 環境中開發和建置 Lambda 函數。

.NET 8 SDK

原生 AOT 編譯是 .NET 8 的一項功能。除了執行時期之外，您也必須在建置機器上安裝 .NET 8 SDK。

Amazon.Lambda.Tools

若要建立 Lambda 函數，請使用 Amazon.Lambda.Tools .NET Core Global Tools 延伸模組。若要安裝 Amazon.Lambda.Tools，請執行下列命令：

```
dotnet tool install -g Amazon.Lambda.Tools
```

如需有關 Amazon.Lambda.Tools .NET CLI 延伸模組的詳細資訊，請參閱 GitHub 上的適用於 .NET CLI 的 AWS 延伸模組儲存庫。

Amazon.Lambda.Templates

若要產生 Lambda 函數程式碼，請使用 Amazon.Lambda.Templates NuGet 套件。若要安裝此範本套件，請執行下列命令：

```
dotnet new install Amazon.Lambda.Templates
```

開始使用

.NET Global CLI 和 AWS Serverless Application Model (AWS SAM) 都提供使用原生 AOT 建置應用程式的入門範本。若要建置您的第一個原生 AOT Lambda 函數，請按照下列指示中的步驟操作。

初始化和部署原生 AOT 編譯的 Lambda 函數

1. 使用原生 AOT 範本初始化新專案，然後瀏覽至包含所建立 `.cs` 和 `.csproj` 檔案的目錄。在此範例中，我們將函數命名為 `NativeAotSample`。

   ```
   dotnet new lambda.NativeAOT -n NativeAotSample
   cd ./NativeAotSample/src/NativeAotSample
   ```

   原生 AOT 範本建立的 `Function.cs` 檔案包含以下函數程式碼。

   ```
   using Amazon.Lambda.Core;
   using Amazon.Lambda.RuntimeSupport;
   using Amazon.Lambda.Serialization.SystemTextJson;
   using System.Text.Json.Serialization;
   ```

```
namespace NativeAotSample;

public class Function
{
    /// <summary>
    /// The main entry point for the Lambda function. The main function is called
 once during the Lambda init phase. It
    /// initializes the .NET Lambda runtime client passing in the function handler
 to invoke for each Lambda event and
    /// the JSON serializer to use for converting Lambda JSON format to the .NET
 types.
    /// </summary>
    private static async Task Main()
    {
        Func<string, ILambdaContext, string> handler = FunctionHandler;
        await LambdaBootstrapBuilder.Create(handler, new
 SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>())
            .Build()
            .RunAsync();
    }

    /// <summary>
    /// A simple function that takes a string and does a ToUpper.
    ///
    /// To use this handler to respond to an AWS event, reference the appropriate
 package from
    /// https://github.com/aws/aws-lambda-dotnet#events
    /// and change the string input parameter to the desired event type. When the
 event type
    /// is changed, the handler type registered in the main method needs to be
 updated and the LambdaFunctionJsonSerializerContext
    /// defined below will need the JsonSerializable updated. If the return type
 and event type are different then the
    /// LambdaFunctionJsonSerializerContext must have two JsonSerializable
 attributes, one for each type.
    ///
    // When using Native AOT extra testing with the deployed Lambda functions is
 required to ensure
    // the libraries used in the Lambda function work correctly with Native AOT. If
 a runtime
    // error occurs about missing types or methods the most likely solution will be
 to remove references to trim-unsafe
```

```
    // code or configure trimming options. This sample defaults to partial TrimMode
 because currently the AWS
    // SDK for .NET does not support trimming. This will result in a larger
 executable size, and still does not
    // guarantee runtime trimming errors won't be hit.
    /// </summary>
    /// <param name="input"></param>
    /// <param name="context"></param>
    /// <returns></returns>
    public static string FunctionHandler(string input, ILambdaContext context)
    {
        return input.ToUpper();
    }
}

/// <summary>
/// This class is used to register the input event and return type for the
 FunctionHandler method with the System.Text.Json source generator.
/// There must be a JsonSerializable attribute for each type used as the input and
 return type or a runtime error will occur
/// from the JSON serializer unable to find the serialization information for
 unknown types.
/// </summary>
[JsonSerializable(typeof(string))]
public partial class LambdaFunctionJsonSerializerContext : JsonSerializerContext
{
    // By using this partial class derived from JsonSerializerContext, we can
 generate reflection free JSON Serializer code at compile time
    // which can deserialize our class and properties. However, we must attribute
 this class to tell it what types to generate serialization code for.
    // See https://docs.microsoft.com/en-us/dotnet/standard/serialization/system-
text-json-source-generation
```

原生 AOT 會將應用程式編譯成單一原生二進位檔。該二進位檔的進入點是 `static Main` 方法。在 `static Main` 中，系統會引導 Lambda 執行期並設定 `FunctionHandler` 方法。在執行期引導程序中，原始碼產生的序列化程式是使用 `new SourceGeneratorLambdaJsonSerializer<LambdaFunctionJsonSerializerContext>()` 加以設定

2. 若要將應用程式部署到 Lambda，請確定本機環境中的 Docker 正在執行，並執行下列命令。

```
dotnet lambda deploy-function
```

在背景中，.NET Global CLI 會下載 AL2023 Docker 映像檔，並在執行中的容器內編譯應用程式程式碼。編譯後的二進位檔會輸出回本機檔案系統，然後再部署至 Lambda。

3. 執行以下命令來測試函數。以您在部署精靈中為函數選擇的名稱取代 <FUNCTION_NAME>。

```
dotnet lambda invoke-function <FUNCTION_NAME> --payload "hello world"
```

CLI 的回應包括冷啟動 (初始化持續時間) 的效能詳細資訊，以及函數調用的總執行時間。

4. 若要依照上述步驟刪除您建立的 AWS 資源，請執行下列命令。以您在部署精靈中為函數選擇的名稱取代 <FUNCTION_NAME>。刪除您不再使用的 AWS 資源，可為 AWS 帳戶避免不必要的費用。

```
dotnet lambda delete-function <FUNCTION_NAME>
```

序列化

若要使用原生 AOT 將函數部署至 Lambda，您的函數程式碼必須使用原始碼產生的序列化。原始碼產生器不會使用執行期反射，來收集序列化存取物件屬性所需的中繼資料，而是會產生建置應用程式時編譯的 C# 原始碼檔案。若要正確設定原始碼產生的序列化程式，請確認函數使用的所有輸入和輸出物件及自訂類型都包含在內。例如，接收來自 API Gateway REST API 的事件並傳回自訂 `Product` 類型的 Lambda 函數，會包含定義如下的序列化程式。

```
[JsonSerializable(typeof(APIGatewayProxyRequest))]
[JsonSerializable(typeof(APIGatewayProxyResponse))]
[JsonSerializable(typeof(Product))]
public partial class CustomSerializer : JsonSerializerContext
{
}
```

裁剪

本機 AOT 會在編譯過程中裁剪應用程式程式碼，以盡可能縮小二進位檔。與舊版 .NET 相比，適用於 Lambda 的 .NET 8 可提供更好的修剪支援。支援已新增至 Lambda 執行時期程式庫、AWS .NET SDK、.NET Lambda Annotations和 .NET 8 本身。

這些改進可能消除建置時間修剪警告，但 .NET 永遠不會完全安全修剪。這表示函數相依的程式庫，可能會在編譯過程中遭到部分裁剪。可以透過將 `TrimmerRootAssemblies` 定義為 `.csproj` 檔案的一部分來管理此問題，如下列範例所示。

```
<ItemGroup>
    <TrimmerRootAssembly Include="AWSSDK.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Core" />
    <TrimmerRootAssembly Include="AWSXRayRecorder.Handlers.AwsSdk" />
    <TrimmerRootAssembly Include="Amazon.Lambda.APIGatewayEvents" />
    <TrimmerRootAssembly Include="bootstrap" />
    <TrimmerRootAssembly Include="Shared" />
</ItemGroup>
```

請注意，當您收到修剪警告時，將產生警告的類別新增至 `TrimmerRootAssembly` 可能無法解決問題。修剪警告表示類別正在嘗試存取一些其他類別，它們在執行時期之前無法確定。若要避免執行時期錯誤，請將此第二個類別新增至 `TrimmerRootAssembly`。

若要進一步了解如何管理修剪警告，請參閱 Microsoft .NET 文件中的修剪警告簡介。

疑難排解

Error: Cross-OS native compilation is not supported (錯誤：不支援跨作業系統原生編譯)。

    您的 Amazon.Lambda.Tools .NET Core 全域工具版本太舊。請更新至最新版本再重試。

Docker: image operating system "linux" cannot be used on this platform (Docker：映像檔作業系統 "linux" 不能在此平台上使用)。

    系統上的 Docker 設定為使用 Windows 容器。請更換至 Linux 容器以執行原生 AOT 建置環境。

如需有關常見錯誤的詳細資訊，請參閱 GitHub 上的適用於 .NET 的 AWS NativeAOT 儲存庫。

使用 Lambda 內容物件擷取 C# 函數資訊

當 Lambda 執行您的函數時，它會將內容物件傳遞至處理常式。此物件提供的各項屬性包含了有關叫用、函式以及執行環境的資訊。

內容屬性

- `FunctionName` – Lambda 函數的名稱。
- `FunctionVersion` – 函數的版本。
- `InvokedFunctionArn` - 用於調用此函數的 Amazon Resource Name (ARN)。指出調用者是否指定版本號或別名。
- `MemoryLimitInMB` - 分配給函數的記憶體數量。
- `AwsRequestId` - 調用請求的識別符。
- `LogGroupName` - 函數的日誌群組。
- `LogStreamName` - 函數執行個體的記錄串流。
- `RemainingTime(TimeSpan)` - 執行逾時前剩餘的毫秒數。
- `Identity` - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
- `ClientContext` - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。
- `Logger` 函式的 Logger 物件。

基於監控目的，您可以使用 `ILambdaContext` 物件中的資訊來輸出函數調用的相關資訊。下列程式碼範例說明如何將內容資訊新增至結構化日誌記錄架構。在此範例中，函數會將 `AwsRequestId` 新增至日誌輸出。如果 Lambda 函數即將逾時，函數也會使用 `RemainingTime` 屬性取消傳輸中的任務。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        this._repo = new DatabaseRepository();
    }
```

```
    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request, ILambdaContext context)
    {
        Logger.AppendKey("AwsRequestId", context.AwsRequestId);

        var id = request.PathParameters["id"];

        using var cts = new CancellationTokenSource();

        try
        {
            cts.CancelAfter(context.RemainingTime.Add(TimeSpan.FromSeconds(-1)));

            var databaseRecord = await this._repo.GetById(id, cts.Token);

            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.OK,
                Body = JsonSerializer.Serialize(databaseRecord)
            };
        }
        catch (Exception ex)
        {
            return new APIGatewayProxyResponse
            {
                StatusCode = (int)HttpStatusCode.InternalServerError,
                Body = JsonSerializer.Serialize(new { error = ex.Message })
            };
        }
        finally
        {
            cts.Cancel();
        }
    }
}
```

記錄和監控 C# Lambda 函數

AWS Lambda 會自動監控 Lambda 函數，並將日誌項目傳送至 Amazon CloudWatch。您的 Lambda 函數隨附有 CloudWatch Logs 日誌群組，且函數的每一執行個體各有一個日誌串流。Lambda 執行期環境會將每次調用的詳細資訊和函數程式碼的其他輸出，傳送至日誌串流。如需 CloudWatch Logs 的詳細資訊，請參閱將 Lambda 函式日誌傳送至 CloudWatch Logs。

章節

- 建立傳回日誌的函數
- 搭配 .NET 使用 Lambda 進階日誌控制項
- 其他日誌工具和程式庫
- 使用 Powertools for AWS Lambda (.NET) 和 AWS SAM 用於結構化記錄
- 在 Lambda 主控台檢視日誌
- 在 CloudWatch 主控台中檢視 記錄
- 使用 AWS Command Line Interface (AWS CLI) 檢視日誌
- 刪除日誌

建立傳回日誌的函數

若要由您的函數程式碼輸出日誌，您可以使用主控台物件中的 ILambdaLogger、主控台類別的方法，或任何能寫入 `stdout` 或 `stderr` 的記錄程式庫。

.NET 執行期會記錄每次調用的 START、END 和 REPORT 行。報告明細行提供下列詳細資訊。

REPORT 行資料欄位

- RequestId - 進行調用的唯一請求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。當調用共用執行環境時，Lambda 會報告所有調用使用的記憶體上限。此行為可能會導致報告值高於預期值。
- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId - 對於追蹤的請求，這是 AWS X-Ray 追蹤 ID。

- SegmentId - 對於追蹤的請求，這是 X-Ray 區段 ID。
- 已取樣 - 對於追蹤的請求，這是取樣結果。

搭配 .NET 使用 Lambda 進階日誌控制項

為了讓您更妥善地控制擷取、處理和使用函數日誌的方式，您可以針對支援的 .NET 執行時期設定下列記錄選項：

- 日誌格式 - 在純文字和結構化 JSON 格式之間為您的日誌進行選擇
- 日誌層級 - 對於 JSON 格式的日誌，請選擇 Lambda 傳送到 CloudWatch 的日誌之詳細等級，例如 ERROR、DEBUG 或 INFO
- 日誌群組 - 選擇您的函數將日誌傳送到的 CloudWatch 日誌群組

如需這些日誌選項的詳細資訊，以及如何設定函數以使用這些選項的說明，請參閱 the section called “設定 Lambda 函數的進階日誌記錄控制項”。

若要使用日誌格式和日誌層級選項與 .NET Lambda 函數搭配使用，請參閱以下各章節中的指引。

搭配 .NET 使用結構化 JSON 日誌格式

如果您為函數的日誌格式選取 JSON，Lambda 會以結構化 JSON 形式使用 ILambdaLogger 來傳送日誌輸出。每個 JSON 日誌物件都包含至少五個鍵值對，其中包含下列索引鍵：

- `"timestamp"` - 產生日誌訊息的時間
- `"level"` - 指派給訊息的日誌層級
- `"requestId"` - 進行調用的唯一請求 ID。
- `"traceId"` - `_X_AMZN_TRACE_ID` 環境變數
- `"message"` - 日誌訊息的內容

`ILambdaLogger` 執行個體可以新增其他鍵值對，例如記錄例外狀況時。您也可以提供自己的其他參數，如 the section called “客戶提供的日誌參數” 章節中所述。

> **Note**
>
> 如果程式碼已使用另一個日誌程式庫來產生 JSON 格式的日誌，請確定函數的日誌格式設定為純文字。將日誌格式設定為 JSON 將導致日誌輸出進行雙重編碼。

下列記錄命令範例示範如何使用 INFO 層級來撰寫日誌訊息。

Example.NET 日誌程式碼

```
context.Logger.LogInformation("Fetching cart from database");
```

也可以使用一般日誌方法，將日誌層級作為引數，如下列範例所示。

```
context.Logger.Log(LogLevel.Information, "Fetching cart from database");
```

會在 CloudWatch Logs 中擷取這些範例程式碼片段所輸出的日誌，如下所示：

Example JSON 日誌記錄

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Fetching cart from database"
}
```

> **Note**
>
> 如果將函數的日誌格式設定為使用純文字而非 JSON，則訊息中擷取的日誌層級會遵循使用四字元標籤的 Microsoft 慣例。例如，Debug 的日誌層級在訊息中表示為 dbug。
> 將函數設定為使用 JSON 格式的日誌時，日誌中擷取的日誌層級會使用完整標籤，如範例 JSON 日誌記錄所示。

如果您沒有為日誌輸出指派層級，Lambda 會自動為其指派層級 INFO。

在 JSON 中記錄例外狀況

搭配使用結構化 JSON 日誌與 `ILambdaLogger` 時，您可以在程式碼中記錄例外狀況，如下列範例所示。

Example例外狀況日誌的使用情況

```
try
```

```
{
    connection.ExecuteQuery(query);
}
catch(Exception e)
{
    context.Logger.LogWarning(e, "Error executing query");
}
```

此程式碼輸出的日誌格式會顯示在下列範例 JSON 中。請注意，使用 LogWarning 呼叫中提供的訊息引數來填充 JSON 中的 message 屬性，而 errorMessage 屬性來自例外狀況本身的 Message 屬性。

Example JSON 日誌記錄

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Warning",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Error executing query",
    "errorType": "System.Data.SqlClient.SqlException",
    "errorMessage": "Connection closed",
    "stackTrace": ["<call exception.StackTrace>"]
}
```

如果函數的日誌格式設定為 JSON，則當您的程式碼擲出未攔截的例外狀況時，Lambda 也會輸出 JSON 格式的日誌訊息。下列程式碼片段和日誌訊息範例顯示如何記錄未攔截的例外狀況。

Example例外狀況代碼

```
throw new ApplicationException("Invalid data");
```

Example JSON 日誌記錄

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Error",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Invalid data",
    "errorType": "System.ApplicationException",
    "errorMessage": "Invalid data",
```

```
    "stackTrace": ["<call exception.StackTrace>"]
}
```

客戶提供的日誌參數

使用 JSON 格式的日誌訊息，可以提供額外的日誌參數，並將這些參數包含在日誌 `message` 中。下列程式碼片段範例顯示一個命令，它可新增兩個使用者提供的參數，分別標記為 `retryAttempt` 和 `uri`。在此範例中，這些參數的值來自傳遞至日誌命令的 `retryAttempt` 和 `uriDestination` 引數。

Example具有其他參數的 JSON 日誌命令

```
context.Logger.LogInformation("Starting retry {retryAttempt} to make GET request to
 {uri}", retryAttempt, uriDestination);
```

此命令輸出的日誌訊息會顯示在下列範例 JSON 中。

Example JSON 日誌記錄

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "Starting retry 1 to make GET request to http://example.com/",
    "retryAttempt": 1,
    "uri": "http://example.com/"
}
```

> **Tip**
>
> 在指定其他參數時，也可以使用位置屬性而非名稱。例如，上一個範例中的日誌命令也可以編寫如下：
>
> ```
> context.Logger.LogInformation("Starting retry {0} to make GET request to {1}",
>  retryAttempt, uriDestination);
> ```

請注意，當您提供額外的日誌參數時，Lambda 會將它們擷取為 JSON 日誌記錄中的頂層屬性。此方法與一些熱門的 .NET 日誌程式庫 (例如 `Serilog`) 不同，它會擷取個別子物件中的其他參數。

如果您為其他參數提供的引數是複雜物件，根據預設，Lambda 會使用 ToString() 方法提供值。若要指出引數應該是 JSON 序列化，請使用 @ 字首，如下列程式碼片段所示。在此範例中，User 是具有 FirstName 和 LastName 屬性的物件。

Example具有 JSON 序列化物件的 JSON 日誌命令

```
context.Logger.LogInformation("User {@user} logged in", User);
```

此命令輸出的日誌訊息會顯示在下列範例 JSON 中。

Example JSON 日誌記錄

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "User {@user} logged in",
    "user":
    {
        "FirstName": "John",
        "LastName": "Doe"
    }
}
```

如果其他參數的引數是陣列或實作 IList 或 IDictionary，則 Lambda 會將引數新增至 JSON 日誌訊息作為陣列，如下列範例 JSON 日誌記錄所示。在此範例中，{users} 會採用 IList 引數，其中包含與先前範例格式相同的 User 屬性的執行個體。Lambda 會將此 IList 轉換為陣列，使用 ToString 方法建立每個值。

Example具有 IList 引數的 JSON 日誌記錄

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "{users} have joined the group",
    "users":
    [
        "Rosalez, Alejandro",
```

```
        "Stiles, John"
    ]
}
```

也可以在日誌命令中使用 @ 字首，對清單進行 JSON 序列化。在下列範例 JSON 日誌記錄中，`users` 屬性為 JSON 序列化。

Example具有 JSON 序列化 IList 引數的 JSON 日誌記錄

```
{
    "timestamp": "2025-09-07T01:30:06.977Z",
    "level": "Information",
    "requestId": "8f711428-7e55-46f9-ae88-2a65d4f85fc5",
    "traceId": "1-6408af34-50f56f5b5677a7d763973804",
    "message": "{@users} have joined the group",
    "users":
    [
        {
            "FirstName": "Alejandro",
            "LastName": "Rosalez"
        },
        {
            "FirstName": "John",
            "LastName": "Stiles"
        }
    ]
}
```

搭配 .NET 使用日誌層級篩選

透過設定日誌層級篩選，您可以選擇只傳送特定詳細資訊層級或更低層級的日誌至 CloudWatch Logs。若要瞭解如何設定函數的日誌層級篩選，請參閱 the section called “日誌層級篩選”。

若要 AWS Lambda 讓 依日誌層級篩選日誌訊息，您可以使用 JSON 格式的日誌，或使用 .NET `Console`方法來輸出日誌訊息。若要建立 JSON 格式日誌，請將函數的日誌類型設定為 JSON，並使用 `ILambdaLogger` 執行個體。

使用 JSON 格式日誌，Lambda 會使用 the section called “搭配 .NET 使用結構化 JSON 日誌格式” 中所述 JSON 物件中的「層級」索引鍵值組篩選您的日誌輸出。

如果使用 .NET `Console` 方法，將訊息寫入 CloudWatch Logs，Lambda 會將日誌層級套用至您的訊息，如下所示：

- Console.WriteLine 方法 - Lambda 會套用 `INFO` 的日誌層級
- Console.Error 方法 - Lambda 會套用 `ERROR` 的日誌層級

將函數設定為使用日誌層級篩選時，您必須從下列選項中選取要 Lambda 傳送至 CloudWatch Logs 的日誌層級。請注意 Lambda 使用的日誌層級與 .NET `ILambdaLogger` 所使用的標準 Microsoft 層級的映射。

Lambda 日誌層級	同等 Microsoft 層級	標準用量
TRACE (大多數詳細資訊)	追蹤	用於追蹤程式碼執行路徑的最精細資訊
DEBUG	偵錯	系統偵錯的詳細資訊
INFO	資訊	記錄函數正常操作的訊息
WARN	警告	有關可能導致未解決意外行為的潛在錯誤的消息
ERROR	錯誤	有關阻止程式碼按預期執行的問題的訊息
FATAL (最少詳細資訊)	嚴重	有關導致應用程式停止運作的嚴重錯誤訊息

Lambda 會將選取的詳細資訊層級和更低層級的日誌傳送至 CloudWatch。例如，如果您設定 WARN 的日誌層級，Lambda 會傳送相對應於 WARN、ERROR 和 FATAL 層級的日誌檔。

其他日誌工具和程式庫

Powertools for AWS Lambda (.NET) 是一種開發人員工具組，可實作無伺服器最佳實務並提高開發人員速度。記錄公用程式提供 Lambda 優化記錄器，其中包含有關所有函數之函數內容的其他資訊，輸出結構為 JSON。使用此公用程式執行下列操作：

- 從 Lambda 內容、冷啟動和 JSON 形式的結構記錄輸出中擷取關鍵欄位
- 在收到指示時記錄 Lambda 調用事件 (預設為停用)
- 透過日誌採樣僅列印調用百分比的所有日誌 (預設為停用)

- 在任何時間點將其他金鑰附加至結構化日誌
- 使用自訂日誌格式化程式 (自帶格式化程式)，以與組織的日誌記錄 RFC 相容的結構輸出日誌。

使用 Powertools for AWS Lambda (.NET) 和 AWS SAM 用於結構化記錄

請依照下列步驟，使用整合式 Powertools for AWS Lambda () 下載、建置和部署範例 Hello World C# 應用程式。NET) 模組使用 AWS SAM。此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會調用、使用內嵌指標格式將日誌和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。該函數會傳回 `hello world` 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- .NET 8
- AWS CLI 第 2 版
- AWS SAM CLI 1.75 版或更新版本。如果您有較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World TypeScript 範本來初始化應用程式。

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 `Enter`。

> **Note**
>
> 對於 HelloWorldFunction may not have authorization defined, Is this okay?，確保輸入 y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl -X GET <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要獲取該函數的日誌，請執行 sam 日誌。如需詳細資訊，請參閱《AWS Serverless Application Model 開發人員指南》 中的 使用日誌。

```
sam logs --stack-name sam-app
```

日誌輸出如下：

```
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8
 2025-09-20T14:15:27.988000 INIT_START Runtime Version:
 dotnet:6.v13        Runtime Version ARN: arn:aws:lambda:ap-
southeast-2::runtime:699f346a05dae24c58c45790bc4089f252bf17dae3997e79b17d939a288aa1ec
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:28.229000
 START RequestId: bed25b38-d012-42e7-ba28-f272535fb80e Version: $LATEST
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:29.259000
 2025-09-20T14:15:29.201Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528962,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ColdStart","Unit":"Count"}],"Dimensions":
[["FunctionName"],["Service"]]}]},"FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","Service":"PowertoolsHelloWorld","ColdStart":1}
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.479000
 2025-09-20T14:15:30.479Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
 {"ColdStart":true,"XrayTraceId":"1-63f3807f-5dbcb9910c96f50742707542","CorrelationId":"d3d
```

```
a549-4d67b2fdc015","FunctionName":"sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionVersion":"$LATEST","FunctionMemorySize":256,"FunctionArn":"arn:aws:l
southeast-2:123456789012:function:sam-app-HelloWorldFunction-
haKIoVeose2p","FunctionRequestId":"bed25b38-d012-42e7-ba28-
f272535fb80e","Timestamp":"2025-09-20T14:15:30.4602970Z","Level":"Information","Service":"P
 world API - HTTP 200"}
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.599000
 2025-09-20T14:15:30.599Z        bed25b38-d012-42e7-ba28-f272535fb80e    info
   {"_aws":{"Timestamp":1676902528922,"CloudWatchMetrics":[{"Namespace":"sam-
app-logging","Metrics":[{"Name":"ApiRequestCount","Unit":"Count"}],"Dimensions":
[["Service"]]}]},"Service":"PowertoolsHelloWorld","ApiRequestCount":1}
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.680000 END
 RequestId: bed25b38-d012-42e7-ba28-f272535fb80e
2025/02/20/[$LATEST]4eaf8445ba7a4a93b999cb17fbfbecd8 2025-09-20T14:15:30.680000
 REPORT RequestId: bed25b38-d012-42e7-ba28-f272535fb80e  Duration: 2450.99 ms
  Billed Duration: 2692 ms Memory Size: 256 MB     Max Memory Used: 74 MB  Init
 Duration: 240.05 ms
XRAY TraceId: 1-63f3807f-5dbcb9910c96f50742707542       SegmentId: 16b362cd5f52cba0
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

管理日誌保留

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，CloudWatch 會在該時間之後自動刪除日誌。若要設定日誌保留，請將下列項目新增至您的 AWS SAM 範本：

```
Resources:
  HelloWorldFunction:
    Type: AWS::Serverless::Function
    Properties:
      # Omitting other properties

  LogGroup:
    Type: AWS::Logs::LogGroup
    Properties:
      LogGroupName: !Sub "/aws/lambda/${HelloWorldFunction}"
      RetentionInDays: 7
```

在 Lambda 主控台檢視日誌

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

在 CloudWatch 主控台中檢視 記錄

您可以使用 Amazon CloudWatch 主控台來檢視所有 Lambda 函數調用的日誌。

若要在 CloudWatch 主控台上檢視日誌

1. 在 CloudWatch 主控台上開啟日誌群組頁面。
2. 選擇您的函數的日誌群組 (/aws/lambda/***your-function-name***)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立其他執行個體以處理並行調用時，就會顯示日誌串流。若要尋找特定調用的日誌，建議您使用 檢測函數 AWS X-Ray。X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

使用 AWS Command Line Interface (AWS CLI) 檢視日誌

是一種 AWS CLI 開放原始碼工具，可讓您使用命令列 Shell 中的 命令與 AWS 服務互動。若要完成本節中的步驟，您必須擁有 AWS CLI 版本 2。

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
```

```
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 get-logs.sh。

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
```

```
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

在 中檢測 C# 程式碼 AWS Lambda

Lambda 與 整合 AWS X-Ray ，以協助您追蹤、偵錯和最佳化 Lambda 應用程式。您可以使用 X-Ray 來追蹤請求，因為它會周遊您應用程式中的資源，其中可能包含 Lambda 函數和其他 AWS 服務。

若要將追蹤資料傳送至 X-Ray，您可以使用以下三個 SDK 庫之一：

- AWS Distro for OpenTelemetry (ADOT) – OpenTelemetry (OTel) SDK 的安全、生產就緒、 AWS支援的分發。
- 適用於 .NET 的 AWS X-Ray SDK – 用於生成追蹤資料並將其傳送至 X-Ray 的 SDK。
- Powertools for AWS Lambda (.NET) – 開發人員工具組，用於實作無伺服器最佳實務並提高開發人員速度。

每個 SDK 均提供將遙測資料傳送至 X-Ray 服務的方法。然後，您可以使用 X-Ray 來檢視、篩選應用程式的效能指標並獲得洞察，從而識別問題和進行最佳化的機會。

> **Important**
>
> X-Ray 和 Powertools AWS Lambda SDKs是 提供的緊密整合檢測解決方案的一部分 AWS。ADOT Lambda Layers 是用於追蹤檢測之業界通用標準的一部分，這類檢測一般會收集更多資料，但可能不適用於所有使用案例。您可以使用任一解決方案在 X-Ray 中實作端對端追蹤。若要深入了解如何在兩者之間做選擇，請參閱在 AWS Distro for OpenTelemetry 和 X-Ray SDK 之間進行選擇。

章節

- 使用 Powertools for AWS Lambda (.NET) 和 AWS SAM 用於追蹤
- 使用 X-Ray SDK 來檢測 .NET 函數
- 透過 Lambda 主控台來啟用追蹤
- 透過 Lambda API 啟用追蹤
- 使用 啟用追蹤 CloudFormation
- 解讀 X-Ray 追蹤

使用 Powertools for AWS Lambda (.NET) 和 AWS SAM 用於追蹤

請依照下列步驟，使用整合式 Powertools for AWS Lambda () 下載、建置和部署範例 Hello World C# 應用程式。NET) 模組使用 AWS SAM。此應用程式實作了基本 API 後端，並使用 Powertools 發送日誌、指標和追蹤。其包含 Amazon API Gateway 端點和 Lambda 函數。當您將 GET 請求傳送至 API Gateway 端點時，Lambda 函數會調用、使用內嵌指標格式將日誌和指標傳送至 CloudWatch，並將追蹤傳送至 AWS X-Ray。函數會傳回 hello world 訊息。

先決條件

若要完成本節中的步驟，您必須執行下列各項：

- .NET 8
- AWS CLI 第 2 版
- AWS SAM CLI 1.75 版或更新版本。如果您有較舊版本的 AWS SAM CLI，請參閱升級 AWS SAM CLI。

部署範例 AWS SAM 應用程式

1. 使用 Hello World TypeScript 範本來初始化應用程式。

```
sam init --app-template hello-world-powertools-dotnet --name sam-app --package-type
 Zip --runtime dotnet6 --no-tracing
```

2. 建置應用程式。

```
cd sam-app && sam build
```

3. 部署應用程式。

```
sam deploy --guided
```

4. 依照螢幕上的提示操作。若要接受互動體驗中提供的預設選項，請按下 Enter。

> **Note**
>
> 對於 HelloWorldFunction may not have authorization defined, Is this okay?，確保輸入 y。

5. 取得已部署應用程式的 URL：

```
aws cloudformation describe-stacks --stack-name sam-app --query
 'Stacks[0].Outputs[?OutputKey==`HelloWorldApi`].OutputValue' --output text
```

6. 調用 API 端點：

```
curl <URL_FROM_PREVIOUS_STEP>
```

成功的話，您將會看到以下回應：

```
{"message":"hello world"}
```

7. 若要取得函數的追蹤，請執行 sam 追蹤。

```
sam traces
```

追蹤輸出如下：

```
New XRay Service Graph
  Start time: 2023-02-20 23:05:16+08:00
  End time: 2023-02-20 23:05:16+08:00
  Reference Id: 0 - AWS::Lambda - sam-app-HelloWorldFunction-pNjujb7mEoew - Edges:
 [1]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 2.814
  Reference Id: 1 - AWS::Lambda::Function - sam-app-HelloWorldFunction-pNjujb7mEoew
 - Edges: []
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
     - fault count(5XX): 0
     - total response time: 2.429
  Reference Id: 2 - (Root) AWS::ApiGateway::Stage - sam-app/Prod - Edges: [0]
   Summary_statistics:
     - total requests: 1
     - ok count(2XX): 1
     - error count(4XX): 0
```

```
      - fault count(5XX): 0
      - total response time: 2.839
  Reference Id: 3 - client - sam-app/Prod - Edges: [2]
   Summary_statistics:
      - total requests: 0
      - ok count(2XX): 0
      - error count(4XX): 0
      - fault count(5XX): 0
      - total response time: 0

XRay Event [revision 3] at (2023-02-20T23:05:16.521000) with id
 (1-63f38c2c-270200bf1d292a442c8e8a00) and duration (2.877s)
 - 2.839s - sam-app/Prod [HTTP: 200]
   - 2.836s - Lambda [HTTP: 200]
 - 2.814s - sam-app-HelloWorldFunction-pNjujb7mEoew [HTTP: 200]
 - 2.429s - sam-app-HelloWorldFunction-pNjujb7mEoew
   - 0.230s - Initialization
   - 2.389s - Invocation
     - 0.600s - ## FunctionHandler
       - 0.517s - Get Calling IP
   - 0.039s - Overhead
```

8. 這是可透過網際網路存取的公有 API 端點。建議您在測試後刪除端點。

```
sam delete
```

X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。不能針對函數設定 X-Ray 取樣率。

使用 X-Ray SDK 來檢測 .NET 函數

您可以測試函數代碼以記錄中繼資料並追蹤下游呼叫。若要記錄函數對其他資源和服務所發出呼叫的詳細資訊，請使用 適用於 .NET 的 AWS X-Ray SDK。要取得開發套件，請將 `AWSXRayRecorder` 套件新增到您的專案檔案中。

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <GenerateRuntimeConfigurationFiles>true</GenerateRuntimeConfigurationFiles>
    <AWSProjectType>Lambda</AWSProjectType>
  </PropertyGroup>
  <ItemGroup>
```

```
    <PackageReference Include="Amazon.Lambda.Core" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.SQSEvents" Version="2.1.0" />
    <PackageReference Include="Amazon.Lambda.Serialization.Json" Version="2.1.0" />
    <PackageReference Include="AWSSDK.Core" Version="3.7.103.24" />
    <PackageReference Include="AWSSDK.Lambda" Version="3.7.104.3" />
    <PackageReference Include="AWSXRayRecorder.Core" Version="2.13.0" />
    <PackageReference Include="AWSXRayRecorder.Handlers.AwsSdk" Version="2.11.0" />
  </ItemGroup>
</Project>
```

有一系列的 Nuget 套件提供自動檢測 AWS SDKs、實體架構和 HTTP 請求。若要查看完整的組態選項集，請參閱《AWS X-Ray 開發人員指南》中的適用於 .NET 的AWS X-Ray SDK。

新增所需的 Nuget 套件後，請設定自動檢測。最佳實務是在函數的處理常式函數之外執行此設定。這麼做可讓您利用執行環境重新使用來改善函數的效能。在下列程式碼範例中，在函數建構函數中呼叫 `RegisterXRayForAllServices`方法，以新增所有 AWS SDK 呼叫的檢測。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;

    public Function()
    {
        // Add auto instrumentation for all AWS SDK calls
        // It is important to call this method before initializing any SDK clients
        AWSSDKHandler.RegisterXRayForAllServices();
        this._repo = new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
```

```
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

透過 Lambda 主控台來啟用追蹤

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態，然後選擇監控和操作工具。
4. 在其他監控工具欄位中，選擇編輯。
5. 在 CloudWatch Application Signals 和 AWS X-Ray 下，選擇啟用 Lambda 服務追蹤。
6. 選擇儲存。

透過 Lambda API 啟用追蹤

使用 AWS CLI 或 AWS SDK 在 Lambda 函數上設定追蹤，請使用下列 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

下列範例 AWS CLI 命令會在名為 my-function 的函數上啟用主動追蹤。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

使用 啟用追蹤 CloudFormation

若要在 CloudFormation 範本中的 `AWS::Lambda::Function` 資源上啟用追蹤，請使用 `TracingConfig` 屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用 `Tracing` 屬性。

Example template.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

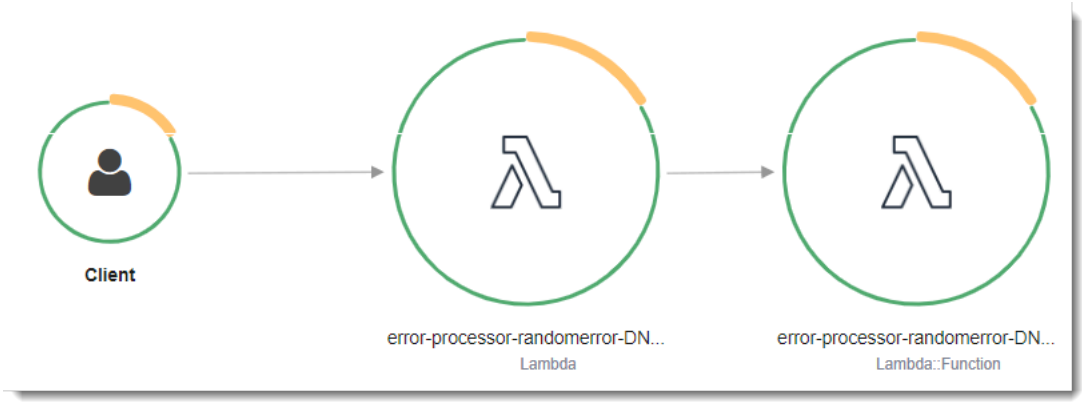
解讀 X-Ray 追蹤

您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將 AWSXRayDaemonWriteAccess 政策新增至執行角色。

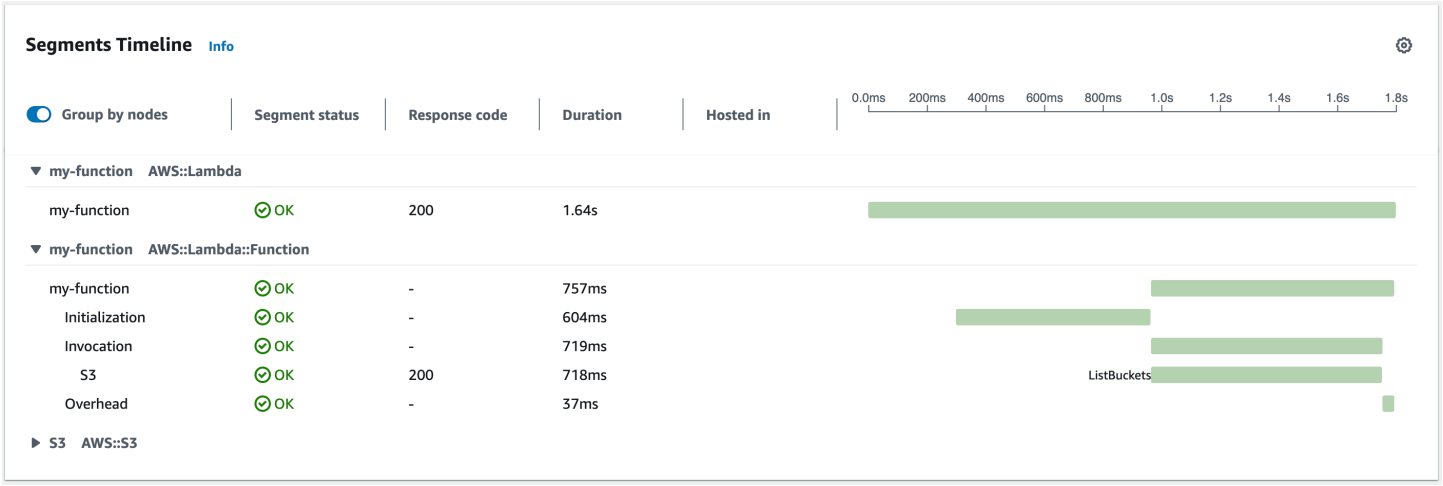
設定主動追蹤之後，您可以透過應用程式來觀察特定請求。X-Ray 服務圖顯示了有關應用程式及其所有元件的資訊。下列範例顯示了一個具有兩個函數的應用程式。主要函式會處理事件，有時會傳回錯誤。頂端的第二個函數會處理出現在第一個日誌群組中的錯誤，並使用 AWS SDK 呼叫 X-Ray、Amazon Simple Storage Service (Amazon S3) 和 Amazon CloudWatch Logs。

X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。不能針對函數設定 X-Ray 取樣率。

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會在每個追蹤上記錄 2 個區段，這會在服務圖表上建立兩個節點。下圖反白顯示了這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。下列範例顯示了具有這 2 個區段的追蹤。兩者都被命名為 my-function，但其中之一的來源為 `AWS::Lambda`，而另一個的來源為 `AWS::Lambda::Function`。如果 `AWS::Lambda` 區段顯示錯誤，Lambda 服務就會出現問題。如果 `AWS::Lambda::Function` 區段顯示錯誤，表示您的函數出現了問題。



此範例會展開 `AWS::Lambda::Function` 區段以顯示其三個子區段：

> **Note**
>
> AWS 目前正在對 Lambda 服務實作變更。由於這些變更，您可能會看到系統日誌訊息的結構和內容，與 AWS 帳戶中不同 Lambda 函數發出的追蹤區段之間存在細微差異。
> 此處顯示的追蹤範例說明了舊式函數區段。下列段落說明了舊式和新式區段之間的差異。
> 這些變化將在未來幾週內實作，除中國和 GovCloud 區域以外，所有 AWS 區域 中的所有函數都會轉換至使用新格式的日誌訊息和追蹤區段。

舊式函數區段包含下列子區段：

- 初始化 - 表示載入函數和執行初始化程式碼所花費的時間。只有函數的每個執行個體所處理的第一個事件會顯示此子區段。
- 調用 – 表示執行處理常式程式碼所花費的時間。
- 額外負荷 - 表示 Lambda 執行期為做好準備以處理下一個事件所花費的時間。

新式函數區段不包含 `Invocation` 子區段。相反地，客戶子區段會直接連接至函數區段。如需舊式和新式函數區段結構的詳細資訊，請參閱the section called “了解 X-Ray 追蹤”。

您也可以檢測 HTTP 用戶端、記錄 SQL 查詢，以及建立具有註釋和中繼資料的自訂子區段。如需詳細資訊，請參閱《AWS X-Ray 開發人員指南》中的 適用於 .NET 的 AWS X-Ray SDK 許可。

定價

您可以在每月免費使用 X-Ray 追蹤，最高可達特定限制，作為 AWS 免費方案的一部分。達到閾值後，X-Ray 會收取追蹤儲存及擷取的費用。如需詳細資訊，請參閱 AWS X-Ray 定價。

以 C# 測試 AWS Lambda 函數

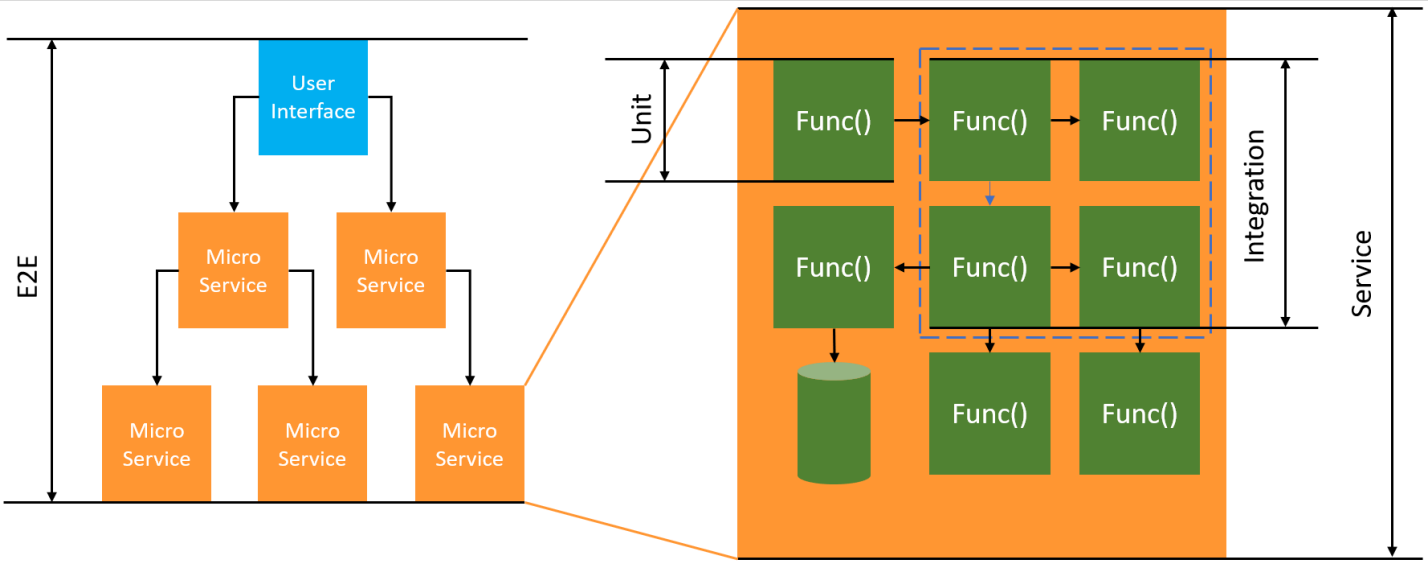
> **Note**
>
> 如需測試無伺服器解決方案之技術和最佳實務的完整介紹，請參閱測試函數章節。

測試無伺服器函數會使用傳統的測試類型和技術，但您也必須考慮測試整個無伺服器應用程式。以雲端為基礎的測試會為您的函數和無伺服器應用程式提供最準確的品質測量標準。

無伺服器應用程式架構包括透過 API 呼叫提供關鍵應用程式功能的受管服務。因此，您的開發週期應包括自動化測試，以便在函數和服務互動時驗證功能。

如果您未建立以雲端為基礎的測試，則可能會因本機環境與部署環境之間的差異而遇到問題。您的持續整合程序應先針對雲端佈建的一組資源進行測試，然後再將程式碼升級至下一個部署環境 (例如 QA、暫存或生產環境)。

繼續閱讀這份簡短指南，了解無伺服器應用程式的測試策略，或造訪無伺服器測試範例儲存庫，深入了解所選語言和執行期的特定實際範例。



若為無伺服器測試，您仍需要寫入單元、整合及端對端測試。

- 單元測試：針對一組隔離的程式碼區塊進行的測試。例如，驗證商業邏輯以計算指定的特定項目與目的地的運費。
- 整合測試：涉及到兩個以上元件或服務進行互動的測試 (通常在雲端環境)。例如，驗證函數是否有處理佇列中的事件。

- 端對端測試：驗證整個應用程式行為的測試。例如，確保基礎設施的設定正確無誤，以及事件如預期在服務之間流動，以記錄客戶的訂單。

測試無伺服器應用程式

通常會混合使用多種方法來測試無伺服器應用程式程式碼，包括在雲端進行測試、透過模擬物件進行測試，以及偶爾使用模擬器進行測試。

在雲端進行測試

在雲端進行測試對所有測試階段 (包括單元測試、整合測試和端對端測試) 來說都很有價值。您可以針對部署在雲端中的程式碼執行測試，並與雲端服務互動。這是最準確的程式碼品質測量方法。

您可以透過主控台使用測試事件，輕鬆在雲端對 Lambda 函數進行偵錯。一個測試事件是函數的 JSON 輸入。如果您的函數不需要輸入，該事件可以是空白的 JSON 文件 (`{}`)。主控台提供各種服務整合的範例事件。在主控台中建立事件後，您可以與團隊分享事件，讓測試變得更容易，結果更一致。

> **Note**
>
> 在控制台中測試函數是簡便快速的入門方式，而將測試週期自動化可確保應用程式的品質和開發速度。

測試工具

為了加速開發週期，您可以在測試函數時使用多種工具和技巧。例如，AWS SAM Accelerate 和 AWS CDK 監看模式都可以縮短更新雲端環境所需的時間。

您定義 Lambda 函數程式碼的方式，讓您可輕鬆加入單元測試。Lambda 需要使用公有無參數建構函數來初始化類別。引入第二個內部建構函數，可讓您控制應用程式使用的相依項。

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSerializer

namespace GetProductHandler;

public class Function
{
    private readonly IDatabaseRepository _repo;
```

```
    public Function(): this(null)
    {
    }

    internal Function(IDatabaseRepository repo)
    {
        this._repo = repo ?? new DatabaseRepository();
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request)
    {
        var id = request.PathParameters["id"];

        var databaseRecord = await this._repo.GetById(id);

        return new APIGatewayProxyResponse
        {
            StatusCode = (int)HttpStatusCode.OK,
            Body = JsonSerializer.Serialize(databaseRecord)
        };
    }
}
```

若要為此函數編寫測試，您可以初始化 Function 類別的新執行個體，並傳入 IDatabaseRepository 的模擬實作。以下範例使用 XUnit、Moq 和 FluentAssertions 來編寫簡單的測試，確保 FunctionHandler 會傳回 200 狀態碼。

```
using Xunit;
using Moq;
using FluentAssertions;

public class FunctionTests
{
    [Fact]
    public async Task TestLambdaHandler_WhenInputIsValid_ShouldReturn200StatusCode()
    {
        // Arrange
        var mockDatabaseRepository = new Mock<IDatabaseRepository>();

        var functionUnderTest = new Function(mockDatabaseRepository.Object);

        // Act
```

```
        var response = await functionUnderTest.FunctionHandler(new
 APIGatewayProxyRequest());

        // Assert
        response.StatusCode.Should().Be(200);
    }
}
```

如需更詳細的範例，包括非同步測試範例在內，請參閱 GitHub 上的 .NET 測試範例儲存庫。

使用 PowerShell 建置 Lambda 函數

以下各節說明以 PowerShell 撰寫 Lambda 函數程式碼時，如何套用常見的程式設計模式與核心概念。

Lambda 提供下列適用於 PowerShell 的範例應用程式：

- blank-powershell – PowerShell 函數，顯示記錄、環境變數和 AWS SDK 的使用。

開始之前，您必須先設定 PowerShell 開發環境。如需如何執行此動作的詳細資訊，請參閱設定 PowerShell 開發環境。

若要了解如何使用 AWSLambdaPSCore 模組從 範本下載範例 PowerShell 專案、建立 PowerShell 部署套件，以及將 PowerShell 函數部署至 AWS 雲端，請參閱 使用 .zip 封存檔部署 PowerShell Lambda 函數。

Lambda 提供適用於 .NET 語言的以下執行期：

Name	識別符	作業系統	取代日期	封鎖函數建立	封鎖函數更新
.NET 10	`dotnet10`	Amazon Linux 2023	2028 年 11 月 14 日	2028 年 12 月 14 日	2029 年 1 月 15 日
.NET 9（僅限容器）	`dotnet9`	Amazon Linux 2023	2026 年 11 月 10 日	未排程	未排程
.NET 8	`dotnet8`	Amazon Linux 2023	2026 年 11 月 10 日	2026 年 12 月 10 日	2027 年 1 月 11 日

主題

- 設定 PowerShell 開發環境
- 使用 .zip 封存檔部署 PowerShell Lambda 函數
- 定義以 PowerShell 編寫的 Lambda 函數處理常式
- 使用 Lambda 內容物件擷取 PowerShell 函數資訊
- 記錄和監控 Powershell Lambda 函數

設定 PowerShell 開發環境

Lambda 為 PowerShell 執行時間提供一組工具和程式庫。如需安裝指示，請參閱 Lambda Tools for PowerShell (在 GitHub 上)。

AWSLambdaPSCore 模組包括以下 cmdlet，以協助編寫和發佈 PowerShell Lambda 函數。

- Get-AWSPowerShellLambdaTemplate - 傳回入門範本的清單。
- New-AWSPowerShellLambda - 根據範本建立初步的 PowerShell 指令碼。
- Publish-AWSPowerShellLambda - 向 Lambda 發佈特定的 PowerShell 指令碼。
- New-AWSPowerShellLambdaPackage - 建立 Lambda 部署套件，您可以將其用於 CI/CD 系統進行部署。

使用 .zip 封存檔部署 PowerShell Lambda 函數

PowerShell 執行時間的部署套件包含 PowerShell 指令碼、PowerShell 指令碼所需要的 PowerShell 模組，以及代管 PowerShell Core 所需的組件。

建立 Lambda 函數

若要開始使用 Lambda 來編寫和叫用 PowerShell 指令碼，您可以使用 `New-AWSPowerShellLambda` cmdlet 根據範本建立入門指令碼。您可以使用 `Publish-AWSPowerShellLambda` cmdlet 將指令碼部署至 Lambda。然後，您可以透過命令列或 Lambda 主控台測試指令碼。

若要建立新的 PowerShell 指令碼、將它上傳並對其進行測試，請執行以下操作：

1. 若要檢視可用範本的清單，請執行以下命令：

```
PS C:\> Get-AWSPowerShellLambdaTemplate

Template               Description
--------               -----------
Basic                  Bare bones script
CodeCommitTrigger      Script to process AWS CodeCommit Triggers
...
```

2. 若要根據 `Basic` 範本建立範例指令碼，請執行下列命令：

```
New-AWSPowerShellLambda -ScriptName MyFirstPSScript -Template Basic
```

一個名為 `MyFirstPSScript.ps1` 的新檔案已建立在目前目錄下的新子目錄中。該目錄的名稱依據 `-ScriptName` 參數而定。您可以使用 `-Directory` 參數來選擇另一個目錄。

您可以看到新的檔案包含下列內容：

```
# PowerShell script file to run as a Lambda function
#
# When executing in Lambda the following variables are predefined.
#   $LambdaInput - A PSObject that contains the Lambda function input data.
#   $LambdaContext - An Amazon.Lambda.Core.ILambdaContext object that contains
 information about the currently running Lambda environment.
#
# The last item in the PowerShell pipeline is returned as the result of the Lambda
 function.
```

```
#
# To include PowerShell modules with your Lambda function, like the
 AWSPowerShell.NetCore module, add a "#Requires" statement
# indicating the module and version.

#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}

# Uncomment to send the input to CloudWatch Logs
# Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
```

3. 若要了解日誌訊息如何從 PowerShell 指令碼傳送至 Amazon CloudWatch Logs，請取消範例指令碼中 `Write-Host` 一行的註解。

   若要示範如何從您的 Lambda 函數傳回資料，請在指令碼的結尾處以 `$PSVersionTable` 新增一行。如此會將 `$PSVersionTable` 新增至 PowerShell 管道。在 PowerShell 指令碼完成之後，PowerShell 管道中的最後一個物件即為 Lambda 函數的傳回資料。`$PSVersionTable` 是 PowerShell 全域變數，它也提供有關執行環境的資訊。

   完成這些變更之後，最後兩行的範例指令碼應類似：

   ```
   Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 5)
   $PSVersionTable
   ```

4. 在您編輯 `MyFirstPSScript.ps1` 檔案之後，請將目錄變更為指令碼的位置。然後執行下列命令，將指令碼發佈至 Lambda：

   ```
   Publish-AWSPowerShellLambda -ScriptPath .\MyFirstPSScript.ps1 -Name
    MyFirstPSScript -Region us-east-2
   ```

   請注意，`-Name` 參數指定 Lambda 函數名稱，此名稱會出現在 Lambda 主控台。您可以使用此函式來手動叫用您的指令碼。

5. 使用 AWS Command Line Interface (AWS CLI) `invoke` 命令來叫用函數。

   ```
   > aws lambda invoke --function-name MyFirstPSScript out
   ```

定義以 PowerShell 編寫的 Lambda 函數處理常式

當 Lambda 函數被叫用時，Lambda 處理常式會叫用 PowerShell 指令碼。

當 PowerShell 指令碼被叫用時，會預先定義以下變數：

- *$LambdaInput* - 包含處理常式之輸入的 PSObject。此輸入可以是事件資料 (由事件來源發佈) 或您提供的自訂輸入，例如字串或任何自訂資料物件。
- *$LambdaContext* - Amazon.Lambda.Core.ILambdaContext 物件，您可以用它來存取有關目前叫用的資訊，例如，目前函數的名稱、記憶體限制，剩餘的執行時間和記錄。

例如，考量以下 PowerShell 範例程式碼。

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function Name:' $LambdaContext.FunctionName
```

此指令碼會傳回從 $LambdaContext 變數取得的 FunctionName 屬性。

> **Note**
>
> 您必須在 PowerShell 指令碼中使用 `#Requires` 陳述式，指示指令碼所要依據的模組。此陳述式執行兩個重要任務。1) 它會與其他開發人員進行通訊以決定指令碼要使用哪個模組。2) 它定義 AWS PowerShell 工具在部署過程中，封裝指令碼時所需要的相依模組。如需有關 PowerShell 中的 `#Requires` 陳述式的詳細資訊，請參閱關於需求。如需 PowerShell 部署套件的詳細資訊，請參閱 使用 .zip 封存檔部署 PowerShell Lambda 函數。
> 但您的 PowerShell Lambda 函數使用 AWS PowerShell cmdlet 時，請務必設定 `#Requires` 陳述式，它參考支援 PowerShell Core 的 `AWSPowerShell.NetCore` 模組，而非參考僅支援 Windows PowerShell 的 `AWSPowerShell` 模組。此外，請務必使用版本 3.3.270.0 或更新版本的 `AWSPowerShell.NetCore`，它可最佳化 cmdlet 匯入程序。如果您使用舊版本，將面臨較長的冷啟動時間。如需詳細資訊，請參閱 AWS Tools for PowerShell。

傳回資料

有些 Lambda 叫用旨在將資料傳回至呼叫者。例如，如果叫用是為了回應來自 API Gateway 的 Web 請求，則我們的 Lambda 函數必須傳回回應。對於 PowerShell Lambda 而言，新增至 PowerShell 管道的最後一個物件是來自 Lambda 叫用的傳回資料。如果該物件為字串，將以其原樣傳回。否則，會使用 `ConvertTo-Json` cmdlet 將該物件轉換為 JSON。

例如，請考量以下 PowerShell 陳述式，它將 $PSVersionTable 新增至 PowerShell 管道：

```
$PSVersionTable
```

在 PowerShell 指令碼完成之後，PowerShell 管道中的最後一個物件即為 Lambda 函數的傳回資料。$PSVersionTable 是 PowerShell 全域變數，它也提供有關執行環境的資訊。

使用 Lambda 內容物件擷取 PowerShell 函數資訊

當 Lambda 執行您的函數時，它會傳遞內容資訊，方法是讓 $LambdaContext 變數可用於處理常式。此變數提供的方法和各項屬性包含了有關叫用、函式以及執行環境的資訊。

內容屬性

- FunctionName – Lambda 函數的名稱。
- FunctionVersion – 函數的版本。
- InvokedFunctionArn - 用於調用此函數的 Amazon Resource Name (ARN)。指出調用者是否指定版本號或別名。
- MemoryLimitInMB - 分配給函數的記憶體數量。
- AwsRequestId - 調用請求的識別符。
- LogGroupName - 函數的日誌群組。
- LogStreamName - 函數執行個體的記錄串流。
- RemainingTime - 執行逾時前剩餘的毫秒數。
- Identity - (行動應用程式) 已授權請求的 Amazon Cognito 身分的相關資訊。
- ClientContext - (行動應用程式) 用戶端應用程式提供給 Lambda 的用戶端內容。
- Logger - 函數的 Logger 物件。

下列 PowerShell 程式碼片段顯示可列印一些內容資訊的簡易處理函式。

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host 'Function name:' $LambdaContext.FunctionName
Write-Host 'Remaining milliseconds:' $LambdaContext.RemainingTime.TotalMilliseconds
Write-Host 'Log group name:' $LambdaContext.LogGroupName
Write-Host 'Log stream name:' $LambdaContext.LogStreamName
```

記錄和監控 Powershell Lambda 函數

AWS Lambda 會自動代表您監控 Lambda 函數，並將日誌傳送至 Amazon CloudWatch。您的 Lambda 函數隨附有 CloudWatch Logs 日誌群組，且函數的每一執行個體各有一個日誌串流。Lambda 執行期環境會將每次調用的詳細資訊傳送至日誌串流，並且轉傳來自函數程式碼的日誌及其他輸出。如需詳細資訊，請參閱將 Lambda 函式日誌傳送至 CloudWatch Logs。

此頁面說明如何從 Lambda 函數的程式碼產生日誌輸出，並使用 AWS Command Line Interface、Lambda 主控台或 CloudWatch 主控台存取日誌。

章節

- 建立傳回日誌的函數
- 在 Lambda 主控台檢視日誌
- 在 CloudWatch 主控台中檢視 記錄
- 使用 AWS Command Line Interface (AWS CLI) 檢視日誌
- 刪除日誌

建立傳回日誌的函數

若要從函式程式碼輸出記錄，您可以使用 Microsoft.PowerShell.Utility 的 cmdlet 或任何寫入 `stdout` 或 `stderr` 的記錄模組。以下範例使用 `Write-Host`。

Example function/Handler.ps1 - 記錄

```
#Requires -Modules @{ModuleName='AWSPowerShell.NetCore';ModuleVersion='3.3.618.0'}
Write-Host `## Environment variables
Write-Host AWS_LAMBDA_FUNCTION_VERSION=$Env:AWS_LAMBDA_FUNCTION_VERSION
Write-Host AWS_LAMBDA_LOG_GROUP_NAME=$Env:AWS_LAMBDA_LOG_GROUP_NAME
Write-Host AWS_LAMBDA_LOG_STREAM_NAME=$Env:AWS_LAMBDA_LOG_STREAM_NAME
Write-Host AWS_EXECUTION_ENV=$Env:AWS_EXECUTION_ENV
Write-Host AWS_LAMBDA_FUNCTION_NAME=$Env:AWS_LAMBDA_FUNCTION_NAME
Write-Host PATH=$Env:PATH
Write-Host `## Event
Write-Host (ConvertTo-Json -InputObject $LambdaInput -Compress -Depth 3)
```

Example記錄格式

```
START RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Version: $LATEST
```

```
Importing module ./Modules/AWSPowerShell.NetCore/3.3.618.0/AWSPowerShell.NetCore.psd1
[Information] - ## Environment variables
[Information] - AWS_LAMBDA_FUNCTION_VERSION=$LATEST
[Information] - AWS_LAMBDA_LOG_GROUP_NAME=/aws/lambda/blank-powershell-
function-18CIXMPLHFAJJ
[Information] - AWS_LAMBDA_LOG_STREAM_NAME=2020/04/01/
[$LATEST]53c5xmpl52d64ed3a744724d9c201089
[Information] - AWS_EXECUTION_ENV=AWS_Lambda_dotnet6_powershell_1.0.0
[Information] - AWS_LAMBDA_FUNCTION_NAME=blank-powershell-function-18CIXMPLHFAJJ
[Information] - PATH=/var/lang/bin:/usr/local/bin:/usr/bin/:/bin:/opt/bin
[Information] - ## Event
[Information] -
{
    "Records": [
        {
            "messageId": "19dd0b57-b21e-4ac1-bd88-01bbb068cb78",
            "receiptHandle": "MessageReceiptHandle",
            "body": "Hello from SQS!",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1523232000000",
                "SenderId": "123456789012",
                "ApproximateFirstReceiveTimestamp": "1523232000001"
            },
            ...
END RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed
REPORT RequestId: 56639408-xmpl-435f-9041-ac47ae25ceed Duration: 3906.38 ms Billed
 Duration: 9867 ms Memory Size: 512 MB Max Memory Used: 367 MB Init Duration: 5960.19
 ms
XRAY TraceId: 1-5e843da6-733cxmple7d0c3c020510040 SegmentId: 3913xmpl20999446 Sampled:
 true
```

.NET 執行期會記錄每次調用的 START、END 和 REPORT 行。報告明細行提供下列詳細資訊。

REPORT 行資料欄位

- RequestId - 進行調用的唯一請求 ID。
- 持續時間 - 函數的處理常式方法處理事件所花費的時間量。
- 計費持續時間 - 調用的計費時間量。
- 記憶體大小 - 分配給函數的記憶體數量。
- 使用的記憶體上限 - 函數所使用的記憶體數量。當調用共用執行環境時，Lambda 會報告所有調用使用的記憶體上限。此行為可能會導致報告值高於預期值。

- 初始化持續時間 - 對於第一個提供的請求，這是執行期載入函數並在處理常式方法之外執行程式碼所花費的時間量。
- XRAY TraceId - 對於追蹤的請求，這是 AWS X-Ray 追蹤 ID。
- SegmentId - 對於追蹤的請求，這是 X-Ray 區段 ID。
- 已取樣 - 對於追蹤的請求，這是取樣結果。

在 Lambda 主控台檢視日誌

您可以在調用 Lambda 函數之後，使用 Lambda 主控台來檢視日誌輸出。

如果可以從內嵌程式碼編輯器測試您的程式碼，您會在執行結果中找到日誌。使用主控台測試功能以調用函數時，您會在詳細資訊區段找到日誌輸出。

在 CloudWatch 主控台中檢視 記錄

您可以使用 Amazon CloudWatch 主控台來檢視所有 Lambda 函數調用的日誌。

若要在 CloudWatch 主控台上檢視日誌

1. 在 CloudWatch 主控台上開啟日誌群組頁面。
2. 選擇您的函數的日誌群組 (/aws/lambda/***your-function-name***)。
3. 選擇日誌串流

每個日誌串流都會對應至函式的執行個體。當您更新 Lambda 函數，以及建立其他執行個體以處理並行調用時，就會顯示日誌串流。若要尋找特定調用的日誌，建議您使用 檢測函數 AWS X-Ray。X-Ray 會在追蹤內記錄有關請求和日誌串流的詳細資訊。

使用 AWS Command Line Interface (AWS CLI) 檢視日誌

是一種 AWS CLI 開放原始碼工具，可讓您使用命令列 Shell 中的 命令與 AWS 服務互動。若要完成本節中的步驟，您必須擁有 AWS CLI 版本 2。

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
    "ExecutedVersion": "$LATEST"
}
```

Example解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 `get-logs.sh`。

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
```

```
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```


刪除日誌

當您刪除函數時，不會自動刪除日誌群組。若要避免無限期地儲存日誌，請刪除日誌群組，或設定保留期間，系統會在該時間之後自動刪除日誌。

使用 Rust 建置 Lambda 函數

由於 Rust 會編譯成原生程式碼，因此您不需要專用的執行期即可在 Lambda 上執行 Rust 程式碼。反之，請使用 Rust 執行期用戶端在本機建置您的專案，然後使用僅限作業系統的執行期將其部署至 Lambda。當您使用僅限作業系統的執行時間時，Lambda 會自動將作業系統保持在最新修補程式的狀態。

適用於 Rust 的工具和程式庫

- 適用於 Rust 的 AWS SDK：適用於 Rust 的 AWS SDK 提供 Rust APIs來與 Amazon Web Services 基礎設施服務互動。
- Lambda 的 Rust 執行期用戶端：Rust 執行期用戶端可讓您輕鬆地執行以 Rust 撰寫的 Lambda 函數。
- Cargo Lambda：這是 Cargo 命令列工具的第三方開放原始碼延伸，可簡化 Rust Lambda 函數的建置和部署。
- Lambda HTTP：此程式庫提供一個包裝程式來處理 HTTP 事件。
- Lambda 延伸：此程式庫可支援使用 Rust 撰寫的 Lambda 延伸。
- AWS Lambda 事件：此程式庫提供常見事件來源整合的類型定義。

Rust 的範本 Lambda 應用程式

- 基本 Lambda 函數：顯示如何處理基本事件的 Rust 函數。
- 具有錯誤處理功能的 Lambda 函數：示範如何在 Lambda 中處理自訂 Rust 錯誤的 Rust 函數。
- 含有共用資源的 Lambda 函數：可在建立 Lambda 函數之前初始化共用資源的 Rust 專案。
- Lambda HTTP 事件：處理 HTTP 事件的 Rust 函數。
- 帶有 CORS 標頭的 Lambda HTTP 事件：使用 Tower 插入 CORS 標頭的 Rust 函數。
- Lambda REST API：使用 Axum 和 Diesel 連線至 PostgreSQL 資料庫的 REST API。
- Serverless Rust 示範：Rust 專案，顯示 Lambda 的 Rust 程式庫、記錄、環境變數和 AWS SDK 的使用方式。
- 基本 Lambda 延伸：顯示如何處理基本延伸事件的 Rust 延伸。
- Lambda 日誌 Amazon Data Firehose 延伸：顯示如何將 Lambda 日誌傳送至 Firehose 的 Rust 延伸。

主題

- 在 Rust 中定義 Lambda 函式處理常式
- 使用 Lambda 內容物件擷取 Rust 函數資訊
- 使用 Rust 處理 HTTP 事件
- 使用 .zip 封存檔部署 Rust Lambda 函數
- 使用 Rust Lambda 函式的層
- 記錄和監控 Rust Lambda 函數

在 Rust 中定義 Lambda 函式處理常式

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

本頁介紹了如何在 Rust 中使用 Lambda 函式處理常式，包括專案初始化、命名慣例及最佳實務。本頁還提供了一個 Rust Lambda 函式範例，該函式會取得訂單的相關資訊、產生文字檔案收據，並將此檔案放入 Amazon Simple Storage Service (S3) 儲存貯體。如需有關編寫函式後如何部署的詳細資訊，請參閱the section called “部署 .zip 封存檔”。

主題

- 設定 Rust 處理常式專案
- Lambda 函式程式碼範例
- Rust 處理常式的有效類別定義
- 處理常式命名慣例
- 定義和存取輸入事件物件
- 存取和使用 Lambda 內容物件
- 在處理常式 適用於 Rust 的 AWS SDK 中使用
- 存取環境變數
- 使用共用狀態
- Rust Lambda 函數的程式碼最佳實務

設定 Rust 處理常式專案

使用 Rust 開發 Lambda 函式時，流程涉及編寫程式碼、編譯程式碼，以及將編譯的成品部署至 Lambda。在 Rust 中設定 Lambda 處理常式專案，最簡單的方法便是使用 AWS Lambda Runtime for Rust。雖然名稱相同，但 Rust 的 AWS Lambda 執行期與適用於 Python、Java 或 Node.js 的 Lambda 中的執行期並不相同。反之， AWS Lambda Runtime for Rust 是一種木箱 (`lambda_runtime`)，支援在 Rust 中寫入 Lambda 函數並連接 AWS Lambda的執行環境。

使用下列命令來安裝 Cargo Lambda，這是 Cargo 命令列工具的第三方開放原始碼延伸，可簡化 Rust Lambda 函數的建置和部署：

```
cargo install cargo-lambda
```

成功安裝 `cargo-lambda` 之後，使用下列命令來初始化新的 Rust Lambda 函式處理常式專案：

```
cargo lambda new example-rust
```

當您執行此命令時，命令列介面 (CLI) 會詢問幾個有關 Lambda 函式的問題：

- HTTP 函式：如果您想要透過 API Gateway 或函式 URL 調用函式，請回答是。否則，請回答否。在本頁的範例程式碼中，由於我們使用自訂 JSON 事件來調用函式，因此回答否。
- 事件類型：如果您想要使用預先定義的事件結構來調用函式，請選取正確的預期事件類型。否則，請將此選項保留空白。在本頁的範例程式碼中，由於我們使用自訂 JSON 事件來調用函式，因此將此選項保留空白。

命令成功執行後，輸入專案的主目錄：

```
cd example-rust
```

此命令會在 `src` 目錄中產生 `generic_handler.rs` 檔案和 `main.rs` 檔案。`generic_handler.rs` 可用於自訂泛型事件處理常式。`main.rs` 檔案包含應用程式的主要邏輯。`Cargo.toml` 檔案包含有關套件的中繼資料，並列出其外部相依項。

Lambda 函式程式碼範例

以下 Rust Lambda 函式程式碼範例會取得訂單的相關資訊、產生文字檔案收據，並將此檔案放入 Amazon S3 儲存貯體。

Example **`main.rs`** Lambda 函數

```
use aws_sdk_s3::{Client, primitives::ByteStream};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use serde::{Deserialize, Serialize};
use serde_json::Value;
use std::env;

#[derive(Deserialize, Serialize)]
struct Order {
    order_id: String,
    amount: f64,
    item: String,
}

async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error> {
    let payload = event.payload;
```

```
    // Deserialize the incoming event into Order struct
    let order: Order = serde_json::from_value(payload)?;

    let bucket_name = env::var("RECEIPT_BUCKET")
        .map_err(|_| "RECEIPT_BUCKET environment variable is not set")?;

    let receipt_content = format!(
        "OrderID: {}\nAmount: ${:.2}\nItem: {}",
        order.order_id, order.amount, order.item
    );
    let key = format!("receipts/{}.txt", order.order_id);

    let config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
    let s3_client = Client::new(&config);

    upload_receipt_to_s3(&s3_client, &bucket_name, &key, &receipt_content).await?;

    Ok("Success".to_string())
}

async fn upload_receipt_to_s3(
    client: &Client,
    bucket_name: &str,
    key: &str,
    content: &str,
) -> Result<(), Error> {
    client
        .put_object()
        .bucket(bucket_name)
        .key(key)
        .body(ByteStream::from(content.as_bytes().to_vec()))  // Fixed conversion
        .content_type("text/plain")
        .send()
        .await?;

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
```

```
}
```

此 main.rs 檔案包含以下程式碼區段：

- use 陳述式：用於匯入 Lambda 函式所需的 Rust Crate 與方法。
- #[derive(Deserialize, Serialize)]：定義此 Rust 結構體中預期輸入事件的結構。
- async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error>：這是主要處理常式方法，其中包含應用程式的主要邏輯。
- async fn upload_receipt_to_s3 (...)：此為主要 function_handler 方法所參照的協助程式方法。
- #[tokio::main]：此為標記 Rust 程式進入點的巨集。其還會設定 Tokio 執行時期，允許 main() 方法使用 async/await 並以非同步方式執行。
- async fn main() -> Result<(), Error>：main() 函式是程式碼的進入點。我們會在其中將 function_handler 指定為主要處理常式方法。

Cargo.toml 檔案範例

下列 Cargo.toml 為此函式的配套檔案。

```
[package]
name = "example-rust"
version = "0.1.0"
edition = "2024"

[dependencies]
aws-config = "1.5.18"
aws-sdk-s3 = "1.78.0"
lambda_runtime = "0.13.0"
serde = { version = "1", features = ["derive"] }
serde_json = "1"
tokio = { version = "1", features = ["full"] }
```

若要讓此函式正常運作，其執行角色必須允許 s3:PutObject 動作。此外，請確保定義 RECEIPT_BUCKET 環境變數。成功調用後，Amazon S3 儲存貯體應包含收據檔案。

Rust 處理常式的有效類別定義

大多數情況下，在 Rust 中定義的 Lambda 處理常式簽章將具有下列格式：

```
async fn function_handler(event: LambdaEvent<T>) -> Result<U, Error>
```

對於此處理常式：

- 此處理常式名稱為 function_handler。
- 處理常式的單一輸入參數為 event，類型為 LambdaEvent<T>。
  - LambdaEvent 是來自 lambda_runtime 套件的包裝函式。借助此包裝函式，您可以存取內容物件，其中包含 Lambda 特定的中繼資料，例如調用的請求 ID。
  - T 是反序列化事件類型。例如，這可以是 serde_json::Value，允許處理常式接收所有泛型 JSON 輸入。或者，如果函式需要預先定義的特定輸入類型，這可以是 ApiGatewayProxyRequest 等類型。
- 處理常式的傳回類型為 Result<U, Error>。
  - U 是反序列化輸出類型。U 必須實作 serde::Serialize 特徵，Lambda 才能將傳回值轉換為 JSON 格式。舉例而言，U 可以是簡單類型，例如 String、serde_json::Value 或自訂結構體 (只要實作了 Serialize)。當程式碼到達 Ok(U) 陳述式時，表示執行成功，且函式會傳回類型為 U 的值。
  - 當程式碼遇到錯誤 (即 Err(Error)) 時，函式會在 Amazon CloudWatch 中記錄錯誤，並傳回類型為 Error 的錯誤回應。

在我們的範例中，處理常式簽章如下所示：

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error>
```

其他有效的處理常式簽章將會具有下列功能：

- 省略 LambdaEvent 包裝函式：如果省略 LambdaEvent，則會失去函式中 Lambda 內容物件的存取權。以下為此類簽章的範例：

  ```
  async fn handler(event: serde_json::Value) -> Result<String, Error>
  ```

- 使用單位類型作為輸入：對於 Rust，您可以使用單位類型來表示空白輸入。這通常用於具有定期排程調用的函式。以下為此類簽章的範例：

  ```
  async fn handler(_: ()) -> Result<Value, Error>
  ```

處理常式命名慣例

Rust 中的 Lambda 處理常式沒有嚴格的命名限制。雖然您可以為處理常式使用任何名稱，但 Rust 中的函式名稱通常採用 snake_case。

對於較小的應用程式，例如在此範例中，您可以使用單一的 main.rs 檔案來包含所有程式碼。對於較大的專案，main.rs 應該包含函式的進入點，但您可透過額外的檔案將程式碼拆分為邏輯模組。例如，您可能擁有下列檔案結構：

```
/example-rust
### src/
#   ### main.rs        # Entry point
#   ### handler.rs     # Contains main handler
#   ### services.rs    # [Optional] Back-end service calls
#   ### models.rs      # [Optional] Data models
### Cargo.toml
```

定義和存取輸入事件物件

JSON 是 Lambda 函數最常見的標準輸入格式。在此範例中，函數預期輸入類似以下內容：

```
{
    "order_id": "12345",
    "amount": 199.99,
    "item": "Wireless Headphones"
}
```

在 Rust 中，您可透過結構體定義預期輸入事件的結構。在此範例中，我們透過定義以下結構體來表示 Order：

```
#[derive(Deserialize, Serialize)]
struct Order {
    order_id: String,
    amount: f64,
    item: String,
}
```

此 struct 符合預期的輸入形狀。在此範例中，#[derive(Deserialize, Serialize)] 巨集會自動產生用於序列化和反序列化的程式碼。這意味著我們可透過 serde_json::from_value() 方法，將泛型輸入 JSON 類型反序列化至結構體中。這在處理常式的前幾行中得以曾現：

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error> {
    let payload = event.payload;

    // Deserialize the incoming event into Order struct
    let order: Order = serde_json::from_value(payload)?;
    ...
}
```

然後，您可以存取該物件的欄位。例如，使用 `order_id` 可以從原始輸入擷取 `order.order_id` 的值。

預先定義的輸入事件類型

`aws_lambda_events` 套件中有許多預先定義的可用輸入事件類型。例如，若要使用 API Gateway 調用函式，需包含下列匯入：

```
use aws_lambda_events::event::apigw::ApiGatewayProxyRequest;
```

然後，確保主要處理常式使用下列簽章：

```
async fn handler(event: LambdaEvent<ApiGatewayProxyRequest>) -> Result<String, Error> {
    let body = event.payload.body.unwrap_or_default();
    ...
}
```

如需有關其他預先定義輸入事件類型的詳細資訊，請參閱 aws_lambda_events crate。

存取和使用 Lambda 內容物件

Lambda 內容物件 包含有關調用、函數以及執行環境的資訊。在 Rust 中，`LambdaEvent` 包裝函式包含內容物件。例如，您可以使用內容物件，透過下列程式碼擷取目前調用的請求 ID：

```
async fn function_handler(event: LambdaEvent<Value>) -> Result<String, Error> {
    let request_id = event.context.request_id;
    ...
}
```

如需內容物件的詳細資訊，請參閱the section called “Context”。

在處理常式 適用於 Rust 的 AWS SDK 中使用

通常，您將使用 Lambda 函數與其他 AWS 資源互動或進行更新。與這些資源互動的最簡單方法便是使用 適用於 Rust 的 AWS SDK。

若要將 SDK 相依項新增至函式，請在 `Cargo.toml` 檔案中納入這些相依相。建議僅新增函式所需的程式庫。在先前的範例程式碼中，我們使用了 `aws_sdk_s3::Client`。在 `Cargo.toml` 檔案中，您可以透過在 `[dependencies]` 區段下新增以下行來納入此相依項：

```
aws-sdk-s3 = "1.78.0"
```

> **Note**
>
> 這可能並非最新版本。請為應用程式選擇適當的版本。

然後，直接在程式碼中匯入相依項：

```
use aws_sdk_s3::{Client, primitives::ByteStream};
```

隨後，範例程式碼會初始化 Amazon S3 用戶端，如下所示：

```
let config = aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let s3_client = Client::new(&config);
```

初始化 SDK 用戶端之後，您可以使用它來與其他 AWS 服務互動。範例程式碼會在 `upload_receipt_to_s3` 協助程式函式中呼叫 Amazon S3 `PutObject` API。

存取環境變數

在處理常式程式碼中，您可以使用 `env::var` 方法來引用任何環境變數。在此範例中，我們使用以下程式碼來參考定義的 `RECEIPT_BUCKET` 環境變數：

```
let bucket_name = env::var("RECEIPT_BUCKET")
    .map_err(|_| "RECEIPT_BUCKET environment variable is not set")?;
```

使用共用狀態

您可以宣告獨立於 Lambda 函數處理常式程式碼的共用變數。這些變數可協助您在函數接收任何事件之前在 初始化階段 過程中載入狀態資訊。例如，您可以透過更新 `main` 函式和處理常式簽章，將本頁的程式碼修改為在初始化 Amazon S3 用戶端時使用共用狀態：

```
async fn function_handler(client: &Client, event: LambdaEvent<Value>) -> Result<String,
 Error> {
    ...
    upload_receipt_to_s3(client, &bucket_name, &key, &receipt_content).await?;
    ...
}

...

#[tokio::main]
async fn main() -> Result<(), Error> {
    let shared_config = aws_config::from_env().load().await;
    let client = Client::new(&shared_config);
    let shared_client = &client;
    lambda_runtime::run(service_fn(move |event: LambdaEvent<Request>| async move {
        handler(&shared_client, event).await
    }))
    .await
```

Rust Lambda 函數的程式碼最佳實務

請遵循下列清單中的準則，在建置 Lambda 函數時使用最佳編碼實務：

- 區隔 Lambda 處理常式與您的核心邏輯。能允許您製作更多可測單位的函式。
- 最小化依存項目的複雜性。偏好更簡易的框架，其可快速在執行環境啟動時載入。
- 將部署套件最小化至執行時間所必要的套件大小。這能減少您的部署套件被下載與呼叫前解壓縮的時間。

請利用執行環境重新使用來改看函式的效能。在函式處理常式之外初始化 SDK 用戶端和資料庫連線，並在本機快取 `/tmp` 目錄中的靜態資產。由您函式的相同執行個體處理的後續叫用可以重複使用這些資源。這可藉由減少函數執行時間來節省成本。

若要避免叫用間洩漏潛在資料，請不要使用執行環境來儲存使用者資料、事件，或其他牽涉安全性的資訊。如果您的函式依賴無法存放在處理常式內記憶體中的可變狀態，請考慮為每個使用者建立個別函式或個別函式版本。

使用 Keep-Alive 指令維持持續連線的狀態。Lambda 會隨著時間的推移清除閒置連線。叫用函數時嘗試重複使用閒置連線將導致連線錯誤。若要維護持續連線，請使用與執行階段相關聯的 keep-alive (保持啟用) 指令。如需範例，請參閱在 Node.js 中重複使用 Keep-Alive 的連線。

使用環境變數將操作參數傳遞給您的函數。例如，如果您正在寫入到 Amazon S3 儲存貯體，而非對您正在寫入的儲存貯體名稱進行硬式編碼，請將儲存貯體名稱設定為環境變數。

避免在 Lambda 函數中使用遞迴調用，其中函數會調用自己或啟動可能再次調用函數的程序。這會導致意外的函式呼叫量與升高的成本。若您看到意外的調用數量，當更新程式碼時，請立刻將函數的預留並行設為 0，以調節對函數的所有調用。

請勿在您的 Lambda 函數程式碼中使用未記錄的非公有 API。對於 AWS Lambda 受管執行期，Lambda 會定期將安全性和功能更新套用至 Lambda 的內部 APIs。這些內部 API 更新可能是向後不相容的，這會導致意外結果，例如若您的函數依賴於這些非公有 API，則叫用失敗。請參閱 API 參考查看公開可用 API 的清單。

撰寫等冪程式碼。為函數撰寫等冪程式碼可確保採用相同方式來處理重複事件。程式碼應正確驗證事件並正常處理重複的事件。如需詳細資訊，請參閱 How do I make my Lambda function idempotent? (如何讓 Lambda 函數等冪？)。

使用 Lambda 內容物件擷取 Rust 函數資訊

當 Lambda 執行您的函數時，其會將內容物件新增至處理常式接收的 LambdaEvent。此物件提供的各項屬性包含了有關叫用、函式以及執行環境的資訊。

內容屬性

- `request_id`：Lambda 服務產生的 AWS 請求 ID。
- `deadline`：目前叫用的執行期限，以毫秒為單位。
- `invoked_function_arn`：被叫用之 Lambda 函數的 Amazon Resource Name (ARN)。
- `xray_trace_id`：目前調用的 AWS X-Ray 追蹤 ID。
- `client_content`：由 AWS 行動 SDK 傳送的用戶端內容物件。除非使用 AWS 行動 SDK 叫用 函數，否則此欄位為空白。
- `identity`：叫用該函數的 Amazon Cognito 身分。除非使用 Amazon Cognito 身分集區發出的 AWS 憑證對 Lambda API 進行叫用請求，否則此欄位為空。
- `env_config`：來自本機環境變數的 Lambda 函數組態。此屬性包括諸如函數名稱、記憶體分配、版本和日誌串流等資訊。

存取叫用內容資訊

Lambda 函數有權存取有關其環境和叫用請求的中繼資料。函數處理常式接收的 `LambaEvent` 物件包含 `context` 中繼資料：

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};

async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    let invoked_function_arn = event.context.invoked_function_arn;
    Ok(json!({ "message": format!("Hello, this is function
 {invoked_function_arn}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

使用 Rust 處理 HTTP 事件

Amazon API Gateway API、Application Load Balancer 以及 Lambda 函數 URL 可將 HTTP 事件傳送至 Lambda。您可以使用來自 crates.io 的 aws_lambda_events 套件來處理來自這些來源的事件。

Example– 處理 API Gateway 代理請求

注意下列事項：

- `use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse}`：aws_lambda_events 套件包含許多 Lambda 事件。為了減少編譯時間，請使用功能標誌來激活所需的事件。範例：`aws_lambda_events = { version = "0.8.3", default-features = false, features = ["apigw"] }`。
- `use http::HeaderMap`：此匯入要求您將 http 套件新增到相依項。

```
use aws_lambda_events::apigw::{ApiGatewayProxyRequest, ApiGatewayProxyResponse};
use http::HeaderMap;
use lambda_runtime::{service_fn, Error, LambdaEvent};

async fn handler(
    _event: LambdaEvent<ApiGatewayProxyRequest>,
) -> Result<ApiGatewayProxyResponse, Error> {
    let mut headers = HeaderMap::new();
    headers.insert("content-type", "text/html".parse().unwrap());
    let resp = ApiGatewayProxyResponse {
        status_code: 200,
        multi_value_headers: headers.clone(),
        is_base64_encoded: false,
        body: Some("Hello AWS Lambda HTTP request".into()),
        headers,
    };
    Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

Lambda 的 Rust 執行期用戶端也提供這些事件類型的抽象表示，讓您可以使用原生 HTTP 類型，而不論是哪個服務傳送事件。下列程式碼等同於前一個範例，而且可直接使用 Lambda 函數 URL、Application Load Balancer 和 API Gateway。

> **Note**
>
> 該 lambda_http 套件使用下面的 lambda_runtime 套件。不必單獨匯入 `lambda_runtime`。

Example– 處理 HTTP 請求

```
use lambda_http::{service_fn, Error, IntoResponse, Request, RequestExt, Response};

async fn handler(event: Request) -> Result<impl IntoResponse, Error> {
    let resp = Response::builder()
        .status(200)
        .header("content-type", "text/html")
        .body("Hello AWS Lambda HTTP request")
        .map_err(Box::new)?;
    Ok(resp)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_http::run(service_fn(handler)).await
}
```

如需如何使用 的另一個範例`lambda_http`，請參閱 AWS Labs GitHub 儲存庫上的 http-axum 程式碼範例。

Rust 的 HTTP Lambda 事件範例

- Lambda HTTP 事件：處理 HTTP 事件的 Rust 函數。
- 帶有 CORS 標頭的 Lambda HTTP 事件：使用 Tower 插入 CORS 標頭的 Rust 函數。
- 含有共用資源的 Lambda HTTP 事件：一個 Rust 函數，它使用在建立函數處理常式之前初始化的共用資源。

使用 .zip 封存檔部署 Rust Lambda 函數

此頁面說明如何編譯 Rust 函數，然後使用 AWS Lambda Cargo Lambda 將編譯的二進位檔部署至 。它還顯示如何使用 AWS Command Line Interface 和 AWS Serverless Application Model CLI 部署編譯的二進位檔。

章節

- 先決條件
- 在 macOS、Windows 或 Linux 上建置 Rust 函數
- 使用 Cargo Lambda 部署 Rust 函數二進位檔
- 使用 Cargo Lambda 叫用您的 Rust 函數

先決條件

- Rust
- AWS CLI 第 2 版

在 macOS、Windows 或 Linux 上建置 Rust 函數

下列步驟示範如何使用 Rust 為第一個 Lambda 函數建立專案，並使用 Cargo Lambda 編譯專案，這是 Cargo 命令列工具的第三方開放原始碼延伸模組，可簡化 Rust Lambda 函數的建置和部署。

1. 安裝 Cargo Lambda，這是 Cargo 命令列工具的第三方開放原始碼延伸，可簡化 Rust Lambda 函數的建置和部署：

   ```
   cargo install cargo-lambda
   ```

   如需其他安裝選項，請參閱 Cargo Lambda 文件中的安裝。
2. 建立套件結構。此命令會在 `src/main.rs` 中建立一些基礎函數程式碼。可以使用此程式碼進行測試，也可以將其替換為您自己的程式碼。

   ```
   cargo lambda new my-function
   ```
3. 在套件的根目錄中，執行 build 子命令來編譯函數中的程式碼。

   ```
   cargo lambda build --release
   ```

（選用）如果您想要在 Lambda 上使用 AWS Graviton2，請新增 `--arm64`旗標來編譯 ARM CPUs 的程式碼。

```
cargo lambda build --release --arm64
```

4. 部署 Rust 函數之前，請在機器上設定 AWS 登入資料。

```
aws configure
```

使用 Cargo Lambda 部署 Rust 函數二進位檔

使用 deploy 子命令將編譯後的二進位檔部署至 Lambda。此命令會建立執行角色，然後建立 Lambda 函數。若要指定現有的執行角色，請使用 --iam-role 標記。

```
cargo lambda deploy my-function
```

使用 部署 Rust 函數二進位檔 AWS CLI

您也可以使用 部署二進位檔 AWS CLI。

1. 使用 build 子命令，建置 .zip 部署套件。

```
cargo lambda build --release --output-format zip
```

2. 若要部署 .zip 套件至 Lambda，請執行 create-function 命令。

   - 對於 `--runtime`，請指定 `provided.al2023`。這是僅限作業系統的執行時期。僅限作業系統的執行時期用於將編譯的二進位檔和自訂執行時期部署至 Lambda。
   - 針對 `--role`，指定執行角色的 ARN。

```
aws lambda create-function \
     --function-name my-function \
     --runtime provided.al2023 \
     --role arn:aws:iam::111122223333:role/lambda-role \
     --handler rust.handler \
     --zip-file fileb://target/lambda/my-function/bootstrap.zip
```

使用 CLI 部署 Rust AWS SAM 函數二進位檔

您也可以使用 AWS SAM CLI 部署二進位檔。

1. 使用資源和屬性定義建立 AWS SAM 範本。對於 `Runtime`，請指定 `provided.al2023`。這是僅限作業系統的執行時期。僅限作業系統的執行時期用於將編譯的二進位檔和自訂執行時期部署至 Lambda。

   如需使用 部署 Lambda 函數的詳細資訊 AWS SAM，請參閱《 AWS Serverless Application Model 開發人員指南》中的 AWS::Serverless::Function。

   Example Rust 二進位檔的 SAM 資源和屬性定義

   ```
   AWSTemplateFormatVersion: '2010-09-09'
   Transform: AWS::Serverless-2016-10-31
   Description: SAM template for Rust binaries
   Resources:
     RustFunction:
       Type: AWS::Serverless::Function
       Properties:
         CodeUri: target/lambda/my-function/
         Handler: rust.handler
         Runtime: provided.al2023
   Outputs:
     RustFunction:
       Description: "Lambda Function ARN"
       Value: !GetAtt RustFunction.Arn
   ```

2. 使用 build 子命令來編譯函數。

   ```
   cargo lambda build --release
   ```

3. 使用 sam deploy 命令將函數部署到 Lambda。

   ```
   sam deploy --guided
   ```

如需使用 CLI 建置 Rust AWS SAM 函數的詳細資訊，請參閱《 AWS Serverless Application Model 開發人員指南》中的使用 Cargo Lambda 建置 Rust Lambda 函數。

使用 Cargo Lambda 叫用您的 Rust 函數

使用 invoke 子命令，透過承載來測試您的函數。

```
cargo lambda invoke --remote --data-ascii '{"command": "Hello world"}' my-function
```

使用 叫用 Rust 函數 AWS CLI

您也可以使用 AWS CLI 來叫用 函數。

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"command": "Hello world"}' /tmp/out.txt
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

使用 Rust Lambda 函式的層

不建議使用層來管理以 Rust 編寫的 Lambda 函式相依項。這是因為以 Rust 編寫的 Lambda 函式會編譯為單一可執行檔，並在部署函式時提供給 Lambda。此可執行檔包含經過編譯的函數程式碼及其所有相依項。使用層不僅會使程序複雜化，還會導致冷啟動時間增加，因為函數需要在初始化階段期間將額外的組件載入記憶體。

若要在 Rust 處理常式中使用外部相依項，請直接將其納入部署套件。如此一來，您既能簡化部署程序，又能利用內建的 Rust 編譯器最佳化。如需如何在函式中匯入與使用適用於 Rust 的 AWS SDK 等相依項的範例，請參閱the section called “處理常式”。

記錄和監控 Rust Lambda 函數

AWS Lambda 會自動代表您監控 Lambda 函數，並將日誌傳送至 Amazon CloudWatch。您的 Lambda 函數隨附有 CloudWatch Logs 日誌群組，且函數的每一執行個體各有一個日誌串流。Lambda 執行期環境會將每次調用的詳細資訊傳送至日誌串流，並且轉傳來自函數程式碼的日誌及其他輸出。如需詳細資訊，請參閱將 Lambda 函式日誌傳送至 CloudWatch Logs。如需設定日誌格式的資訊，請參閱 設定 JSON 和純文字日誌格式。此頁面說明如何從 Lambda 函數的程式碼中產生日誌輸出。

建立編寫日誌的函數

若要從您的函式程式碼輸出日誌，可使用寫入到 `stdout` 或 `stderr` 的任何日誌記錄函數，例如 `println!` 巨集。下列範例使用 `println!` 在函數處理常式啟動時和完成之前列印訊息。

```
use lambda_runtime::{service_fn, LambdaEvent, Error};
use serde_json::{json, Value};
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    println!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    println!("Rust function responds to {}", &first_name);
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    lambda_runtime::run(service_fn(handler)).await
}
```

實作帶有追蹤套件的進階日誌記錄

追蹤是檢測 Rust 程式以收集結構化、以事件為基礎的診斷資訊的架構。此架構提供公用程式來自訂日誌記錄輸出層級和格式，例如建立結構化的 JSON 日誌訊息。若要使用此架構，必須在實作函數處理常式之前初始化 `subscriber`。然後，您可以使用追蹤巨集 (例如 `debug`、`info` 和 `error`) 來指定每個案例所需的日誌記錄層級。

Example– 使用追蹤套件

注意下列事項：

- `tracing_subscriber::fmt().json()`：包含此選項時，日誌會格式化為 JSON。若要使用此選項，必須將 json 功能包含在 `tracing-subscriber` 相依項中 (例如，`tracing-subscriber = { version = "0.3.11", features = ["json"] }`)。
- `#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]`：每次叫用處理常式時，此註釋都會產生一個範圍。此範圍會將請求 ID 新增至每個日誌行。
- `{ %first_name }`：此建構模組將 `first_name` 欄位新增到使用該欄位的日誌行。此欄位的值對應具有相同名稱的變數。

```
use lambda_runtime::{service_fn, Error, LambdaEvent};
use serde_json::{json, Value};
#[tracing::instrument(skip(event), fields(req_id = %event.context.request_id))]
async fn handler(event: LambdaEvent<Value>) -> Result<Value, Error> {
    tracing::info!("Rust function invoked");
    let payload = event.payload;
    let first_name = payload["firstName"].as_str().unwrap_or("world");
    tracing::info!({ %first_name }, "Rust function responds to event");
    Ok(json!({ "message": format!("Hello, {first_name}!") }))
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt().json()
        .with_max_level(tracing::Level::INFO)
        // this needs to be set to remove duplicated information in the log.
        .with_current_span(false)
        // this needs to be set to false, otherwise ANSI color codes will
        // show up in a confusing manner in CloudWatch logs.
        .with_ansi(false)
        // disabling time is handy because CloudWatch will add the ingestion time.
        .without_time()
        // remove the name of the function from every log entry
        .with_target(false)
        .init();
    lambda_runtime::run(service_fn(handler)).await
}
```

叫用此 Rust 函數時，其會列印類似於以下內容的兩條日誌行：

```
{"level":"INFO","fields":{"message":"Rust function invoked"},"spans":
[{"req_id":"45daaaa7-1a72-470c-9a62-e79860044bb5","name":"handler"}]}
{"level":"INFO","fields":{"message":"Rust function responds to
 event","first_name":"David"},"spans":[{"req_id":"45daaaa7-1a72-470c-9a62-
e79860044bb5","name":"handler"}]}
```

使用 AWS Lambda 函數的最佳實務

以下是使用 AWS Lambda的推薦最佳實務：

主題

- 函數程式碼
- 函數組態
- 函數可擴展性
- 指標與警示
- 使用串流
- 安全最佳實務

函數程式碼

請利用執行環境重新使用來改看函式的效能。在函式處理常式之外初始化 SDK 用戶端和資料庫連線，並在本機快取 `/tmp` 目錄中的靜態資產。由您函式的相同執行個體處理的後續叫用可以重複使用這些資源。這可藉由減少函數執行時間來節省成本。

若要避免叫用間洩漏潛在資料，請不要使用執行環境來儲存使用者資料、事件，或其他牽涉安全性的資訊。如果您的函式依賴無法存放在處理常式內記憶體中的可變狀態，請考慮為每個使用者建立個別函式或個別函式版本。

使用 Keep-Alive 指令維持持續連線的狀態。Lambda 會隨著時間的推移清除閒置連線。叫用函數時嘗試重複使用閒置連線將導致連線錯誤。若要維護持續連線，請使用與執行階段相關聯的 keep-alive (保持啟用) 指令。如需範例，請參閱在 Node.js 中重複使用 Keep-Alive 的連線。

使用環境變數將操作參數傳遞給您的函數。例如，如果您正在寫入到 Amazon S3 儲存貯體，而非對您正在寫入的儲存貯體名稱進行硬式編碼，請將儲存貯體名稱設定為環境變數。

避免在 Lambda 函數中使用遞迴調用，其中函數會調用自己或啟動可能再次調用函數的程序。這會導致意外的函式呼叫量與升高的成本。若您看到意外的調用數量，當更新程式碼時，請立刻將函數的預留並行設為 `0`，以調節對函數的所有調用。

請勿在您的 Lambda 函數程式碼中使用未記錄的非公有 API。對於 AWS Lambda 受管執行期，Lambda 會定期將安全性和功能更新套用至 Lambda 的內部 APIs。這些內部 API 更新可能是向後不相容的，這會導致意外結果，例如若您的函數依賴於這些非公有 API，則叫用失敗。請參閱 API 參考查看公開可用 API 的清單。

撰寫等冪程式碼。為函數撰寫等冪程式碼可確保採用相同方式來處理重複事件。程式碼應正確驗證事件並正常處理重複的事件。如需詳細資訊，請參閱 How do I make my Lambda function idempotent? (如何讓 Lambda 函數等冪？)。

> **Note**
>
> 您可以使用 Powertools for AWS Lambda 讓函數成為等冪。如需詳細資訊，請參閱：
>
> - Python – 冪等性公用程式
> - TypeScript – 冪等性公用程式
> - Java – 冪等性公用程式
> - .NET – 冪等性公用程式

如需語言特定的程式碼最佳實務，請參閱下列章節：

- Node.js Lambda 函數的程式碼最佳實務
- TypeScript Lambda 函式的程式碼最佳實務
- Python Lambda 函數的程式碼最佳實務
- Ruby Lambda 函數的程式碼最佳實務
- Java Lambda 函數的程式碼最佳實務
- Go Lambda 函數的程式碼最佳實務
- C# Lambda 函數的程式碼最佳實務
- Rust Lambda 函數的程式碼最佳實務

函數組態

啟動您的 Lambda 函式的效能測試是確保您挑選最佳記憶體大小組態非常重要的一環。任何記憶體大小的增加能觸發相同的增加可用於函數的 CPU。函數的記憶體用量取決於每個呼叫，且可以在 Amazon CloudWatch 中檢視。每一次呼叫會產生 `REPORT:` 項目，如下所示：

```
REPORT RequestId: 3604209a-e9a3-11e6-939a-754dd98c7be3 Duration: 12.34 ms Billed
 Duration: 100 ms Memory Size: 128 MB Max Memory Used: 18 MB
```

藉由分析 `Max Memory Used:` 欄位，您可以判斷您的函式是否需要更多記憶體，或是否過度佈建函式記憶體大小。

若要為您的函數尋找正確的記憶體組態，我們建議您使用開放原始碼 AWS Lambda Power Tuning 專案。如需詳細資訊，請參閱 GitHub 上的 AWS Lambda 電源調校。

若要最佳化函數效能，我們也建議部署可充分利用進階向量延伸 2 (AVX2) 的程式庫。這可讓您處理繁重的工作負載，包括機器學習推論、媒體處理、高效能運算 (HPC)、科學模擬和財務建模。如需詳細資訊，請參閱使用 AVX2 建立更快的 AWS Lambda 函數。

對您的 Lambda 函數進行負載測試以判斷最佳的逾時值。分析您的函式執行多久時間是很重要的，如此您更能判斷任何相依服務的問題，這可能會增加超出預期的函式並行。當您的 Lambda 函數對可能無法處理 Lambda 擴展的資源發出網路呼叫時，這尤其重要。如需有關應用程式負載測試的詳細資訊，請參閱 AWS上的分散式負載測試。

設定 IAM 政策時，使用最嚴苛的許可。了解您的 Lambda 函數需要的資源與操作，並限制這些許可的執行角色。如需詳細資訊，請參閱在 中管理許可 AWS Lambda。

熟悉 Lambda 配額。承載大小、檔案描述項與 /tmp 空間，是在判斷執行時間資源限制時經常忽略的。

刪除您不再使用的 Lambda 函數。透過上述方法，未使用的函式不會對您的部署套件大小限制做不必要的計算。

如果您使用 Amazon Simple Queue Service 作為事件來源，請確保函數的預期叫用時間值不會超過佇列上的可見性逾時值。這同時適用於 CreateFunction 和 UpdateFunctionConfiguration。

- 在 CreateFunction 的情況下， AWS Lambda 會失敗函數建立程序。
- 如果是 UpdateFunctionConfiguration，則可能會導致重複呼叫函數。

函數可擴展性

熟悉您的上游和下游輸送量限制。雖然 Lambda 函數可隨負載進行無縫擴展，但上游和下游相依性可能不具有相同的輸送量能力。如果需要限制函數可擴展的高度，則可以在函數中設定預留並行。

在節流容限中建置。如果同步函數因為流量超過 Lambda 的擴展速率而遇到限流，則可以使用下列策略來改善限流容錯：

- 使用逾時、重試和退避 (具有抖動)。實作這些策略可順利完成重試的調用，並有助於確保 Lambda 可以在幾秒鐘內向上擴展，以最大程度減少最終使用者限流。
- 使用佈建並行。佈建並行是 Lambda 分配給函數的預先初始化執行環境的數目。Lambda 會在可用時使用佈建並行來處理傳入請求。如有需要，Lambda 也可以擴展函數，使其超過佈建並行設定。設定佈建並行會對 AWS 您的帳戶產生額外費用。

指標與警示

使用 將 CloudWatch 指標與 Lambda 搭配使用 與 CloudWatch 警示，而非建立或更新 Lambda 函數程式碼內的指標。這是追蹤 Lambda 函數運作狀態的更有效率方式，可讓您盡快在開發階段找出問題。例如，您可以根據預期的 Lambda 函數叫用持續時間來設定警示，以解決任何由函數程式碼導致的瓶頸或延遲。

使用內嵌指標格式 (EMF) 以非同步方式發出自訂指標。使用 EMF 透過函式的日誌發出指標，而不是向 CloudWatch 發出同步 API 呼叫。此方法可以減少延遲並改善效能。Powertools for AWS Lambda 中的 Metrics 公用程式會自動處理 EMF 格式。如需詳細資訊，請參閱 Powertools for AWS Lambda 文件中的 Python、TypeScript、Java 或 .NET Metrics 公用程式。如需使用 EMF 產生指標格式日誌的資訊，請參閱 Amazon CloudWatch User Guide 中的 Publishing logs with the embedded metric format。

使用結構化 JSON 記錄以提高可觀測性。結構化記錄可讓您更輕鬆地搜尋、篩選和分析函式的日誌。請考慮使用 Powertools for 的 Logger 公用程式 AWS Lambda ，在 JSON 中自動格式化日誌。如需詳細資訊，請參閱 Powertools for AWS Lambda 文件中的 Python、TypeScript、Java 或 .NET Logger 公用程式。

利用您的記錄程式庫與 AWS Lambda 指標與維度，找出應用程式錯誤 (例如，ERR、ERROR、WARNING 等等)。

使用 AWS 成本異常偵測來偵測帳戶中的異常活動。成本異常偵測會使用機器學習來持續監控您的成本和使用量，同時盡可能減少誤報警示。成本異常偵測使用來自 的資料 AWS Cost Explorer，最多可延遲 24 小時。因此，使用後最多可能需要 24 小時才能偵測到異常。若要開始使用成本異常偵測，您必須先使用註冊 Cost Explorer。然後，存取成本異常偵測。

使用串流

使用不同的批次與記錄大小測試讓每個事件來源的輪詢頻率調整為函式可以多快完成作業。CreateEventSourceMapping 批次大小參數控制每次呼叫時能傳送至您函數的記錄上限。更大的批次大小通常會更有效吸收更大集合的記錄的呼叫成本，增加您的傳輸量。

Lambda 預設會在記錄可用時立即叫用函數。如果 Lambda 從事件來源中讀取的批次只有一筆記錄，Lambda 只會傳送一筆記錄至函數。為避免調用具有少量記錄的函數，您可設定批次間隔，請求事件來源緩衝記錄最長達五分鐘。調用函數之前，Lambda 會繼續從事件來源中讀取記錄，直到收集到完整批次、批次間隔到期或者批次達到 6 MB 的承載限制。如需詳細資訊，請參閱批次處理行為。

> **Warning**
>
> Lambda 事件來源映射至少會處理每個事件一次，而且可能會重複處理記錄。為避免與重複事件相關的潛在問題，強烈建議您讓函數程式碼具有等冪性。若要進一步了解，請參閱 AWS 知識中心中的如何使 Lambda 函數具有等冪性。

啟用部分批次回應以進行串流處理。處理來自 Kinesis 或 DynamoDB Streams 等串流的記錄批次時，請啟用部分批次回應，以允許 Lambda 僅重試失敗的記錄，而不是重試整個批次。這樣可以提高處理效率並減少不必要的重新處理。您可以選擇使用 Powertools for 的 Batch 公用程式 AWS Lambda 來簡化批次處理模式。

> **Note**
>
> 您可以使用適用於 的 Powertools AWS Lambda 進行批次處理。如需詳細資訊，請參閱：
>
> - Python – 批次處理
> - TypeScript – 批次處理
> - Java – 批次處理
> - .NET – 批次處理

新增碎片以增加 Kinesis 串流處理輸送量。Kinesis 串流由一個或多個碎片組成。Lambda 可從 Kinesis 讀取資料的速率會隨著碎片數量而線性擴展。增加碎片的數量會直接增加最大並行 Lambda 函數叫用的數量，還會增加 Kinesis 串流處理輸送量。如需碎片與函數調用之間的關係詳細資訊，請參閱 the section called “輪詢和批次處理串流”。如果您正在增加 Kinesis 串流中碎片的數量，請確保為您的資料挑選良好的分割區索引鍵 (請參閱分割區索引鍵)，如此一來，相關的記錄會位於相同的碎片上，您的資料也會獲得妥善的分配。

在 IteratorAge 上使用 Amazon CloudWatch，判斷是否正在處理您的 Kinesis 串流。例如，將 CloudWatch 警示的最大設定設為 30000 (30 秒)。

安全最佳實務

使用 監控 的使用情況， AWS Lambda 因為它與安全最佳實務相關 AWS Security Hub CSPM。Security Hub CSPM 使用安全控制來評估資源組態和安全標準，以協助您符合各種合規架構。如需使用 Security Hub CSPM 評估 Lambda 資源的詳細資訊，請參閱 AWS Security Hub CSPM 《使用者指南》中的AWS Lambda 控制項。

使用 Amazon GuardDuty Lambda Protection 監控 Lambda 網路活動日誌。GuardDuty Lambda 保護可協助您在 中叫用 Lambda 函數時識別潛在的安全威脅 AWS 帳戶。例如，如果其中一個函數查詢與加密貨幣相關活動相關聯的 IP 位址。GuardDuty 會監控調用 Lambda 函數時產生的網路活動日誌。若要進一步了解，請參閱《Amazon GuardDuty 使用者指南》中的 Lambda Protection。

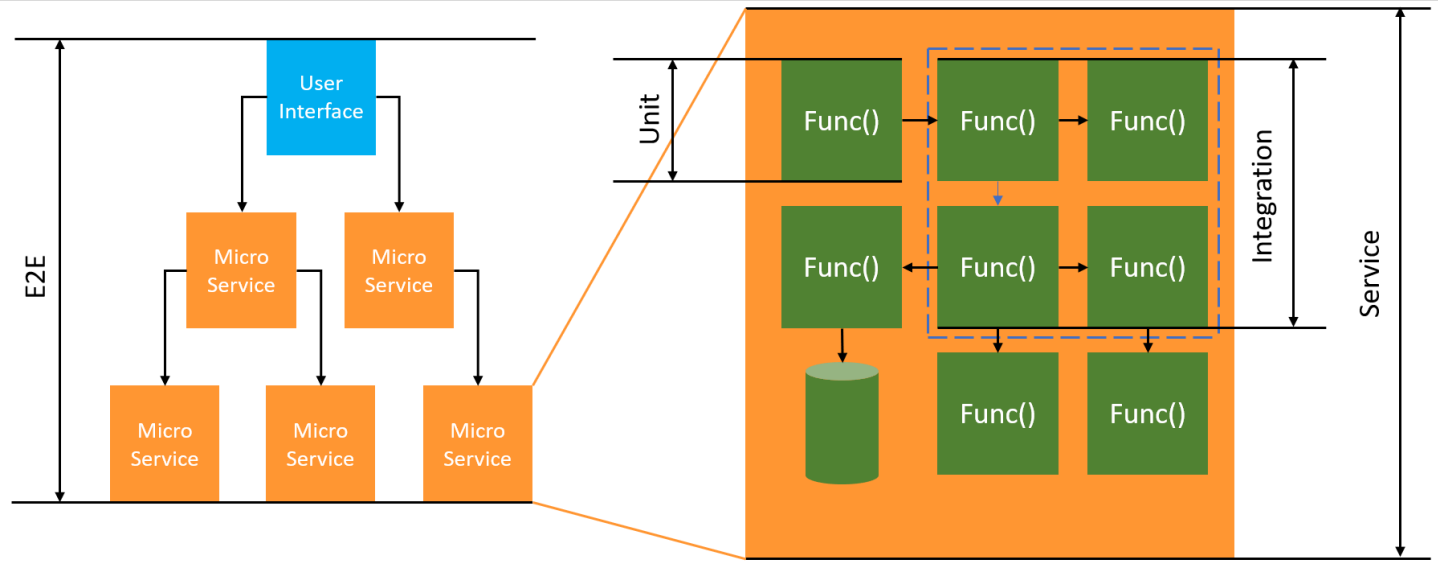
如何測試無伺服器函數和應用程式

測試無伺服器函數會使用傳統的測試類型和技術，但您也必須考慮測試整個無伺服器應用程式。以雲端為基礎的測試會為您的函數和無伺服器應用程式提供最準確的品質測量標準。

無伺服器應用程式架構包括透過 API 呼叫提供關鍵應用程式功能的受管服務。因此，您的開發週期應包括自動化測試，以便在函數和服務互動時驗證功能。

如果您未建立以雲端為基礎的測試，則可能會因本機環境與部署環境之間的差異而遇到問題。您的持續整合程序應先針對雲端佈建的一組資源進行測試，然後再將程式碼升級至下一個部署環境 (例如 QA、暫存或生產環境)。

繼續閱讀這份簡短指南，了解無伺服器應用程式的測試策略，或造訪無伺服器測試範例儲存庫，深入了解所選語言和執行期的特定實際範例。



若為無伺服器測試，您仍需要寫入單元、整合及端對端測試。

- 單元測試：針對一組隔離的程式碼區塊進行的測試。例如，驗證商業邏輯以計算指定的特定項目與目的地的運費。
- 整合測試：涉及到兩個以上元件或服務進行互動的測試 (通常在雲端環境)。例如，驗證函數是否有處理佇列中的事件。
- 端對端測試：驗證整個應用程式行為的測試。例如，確保基礎設施的設定正確無誤，以及事件如預期在服務之間流動，以記錄客戶的訂單。

目標業務成果

測試無伺服器解決方案時可能需要更多時間來設定測試，以驗證服務之間的事件驅動互動。閱讀本指南時，請謹記以下實際的商業原因：

- 提高應用程式的品質
- 降低建構功能和修復錯誤的時間

應用程式的品質取決於測試各種情境來驗證功能。仔細思考業務情境並自動化這些針對雲端服務進行的測試，將有助於提高應用程式的品質。

在反覆進行的開發週期中發現軟體錯誤和組態問題對成本和排程的影響最小。如果在開發過程中仍未發現問題，則在生產過程中尋找和修復問題時，將需要耗費更多人力。

經過妥善規劃的無伺服器測試策略可驗證 Lambda 函數和應用程式是否在雲端環境中如預期般運行，藉此提高軟體品質並縮短反覆運算的時間。

測試項目

建議您採用能測試受管服務 行為、雲端組態、安全政策以及程式碼整合的測試策略，以提升軟體品質。行為測試 (又稱為黑箱測試) 會在不了解所有內部情況的狀況下驗證系統是否如預期般正常運作。

- 執行單元測試以檢查 Lambda 函數內的商業邏輯。
- 確認整合服務實際上是否有調用，且輸入參數是否正確無誤。
- 檢查事件是否在工作流程中端對端通過所有預期的服務。

在傳統的伺服器型架構中，團隊通常會定義測試的範圍，且只納入在應用程式伺服器上執行的程式碼。其他元件、服務或相依項目通常會被認定屬於外部且超出測試範圍。

無伺服器應用程式通常由小型工作單位組成，例如從資料庫擷取產品、處理佇列項目或是調整儲存空間映像大小的 Lambda 函數。每個元件都會在各自的環境中執行。團隊可能會在單一應用程式中負起這類眾多小型單位的責任。

部分應用程式功能可完全委派給受管服務 (例如 Amazon S3)，也可以在不使用任何內部開發程式碼的情況下建立。您不需要測試這類受管服務，不過您確實需要測試與這些服務的整合。

如何測試無伺服器

您可能對測試在本機部署的應用程式的方式不陌生：您撰寫的測試會針對完全在桌面作業系統或容器內執行的程式碼進行測試。例如，您可以透過請求調用本機 Web 服務元件，然後對回應做出聲明。

無伺服器解決方案是根據您的函數程式碼和雲端受管服務 (例如佇列、資料庫、事件匯流排和簡訊傳送系統) 建置而成。這些元件皆透過 事件驅動的架構 連接，其中訊息 (稱為 事件) 會從一項資源流向另一項資源。這些互動可以是同步動作 (例如 Web 服務立即傳回結果時)，或是稍後完成的非同步動作 (例如將項目放置在佇列中，或展開工作流程步驟)。您的測試策略必須納入這兩種情境，並測試服務之間的互動。若為非同步互動，則可能需要偵測下游元件中可能無法立即觀測到的副作用。

複寫整個雲端環境 (包括佇列、資料庫資料表、事件匯流排、安全性政策等) 的作法並不實際。由於本機環境與雲端中部署的環境之間存在差異，因此您必定會遇到問題。環境之間的變化將增加重現和修復錯誤所需的時間。

在無伺服器應用程式中，架構元件通常完全位於雲端之中，因此您必須對雲端中的程式碼和服務進行測試，才能夠開發功能和修正錯誤。

測試技術

在實際情況下，您的測試策略可能會包括各種技術，以提升解決方案的品質。您將會採用快速互動測試來偵錯主控台中的函數、使用自動化單元測試來檢查隔離的商業邏輯、透過模擬來驗證對外部服務的呼叫，以及偶爾對模擬服務的模擬器進行測試。

- the section called “在雲端進行測試”：您可以部署基礎結構和程式碼，透過實際的服務、安全政策、組態和基礎結構的具體參數進行測試。以雲端為基礎的測試可為您的程式碼提供最精準的品質測量方法。

  您可以透過在主控台中偵錯函數，進而在雲端中迅速進行測試。您可以從範例測試事件資源庫中進行選擇，也可以建立自訂事件來單獨測試函數。您也可以透過主控台與團隊分享測試事件。

  若要在開發和建置生命週期中 自動化 測試，則必須在主控台之外進行測試。如需自動化策略和資源，請參閱本指南中特定語言的測試章節。
- the section called “透過模擬物件進行測試”：模擬物件是程式碼內用於模擬並取代外部服務的測試物件。模擬物件提供預先定義的行為來驗證服務呼叫和參數。假物件 是一種模擬實作，它會透過捷徑來簡化或提升效能。例如，假物件的資料存取物件可能會從記憶體內的資料儲存傳回資料。模擬物件可以模仿和簡化複雜的相依項目，但也可能產生更多的模擬來代替巢狀的相依項目。

- the section called “使用 AWS SAM CLI 進行本機測試”：使用 AWS SAM CLI 在與 Lambda 使用相同執行時間環境的 Docker 容器中，於本機叫用 AWS Lambda 函數。您可以測試函式邏輯與事件處理，無需部署至雲端。
- the section called “透過模擬進行測試”：使用 VS Code 中的 LocalStack 整合，在本機模擬多個 AWS 服務以測試服務整合。

在雲端進行測試

在雲端進行測試對所有測試階段 (包括單元測試、整合測試和端對端測試) 來說都很有價值。當您針對同時與雲端服務互動的雲端程式碼進行測試時，便可以獲得最精準的程式碼品質測量方法。

您可以透過在 AWS 管理主控台 加入測試事件輕鬆在雲端中執行 Lambda 函數。一個測試事件是函數的 JSON 輸入。如果您的函數不需要輸入，該事件可以是空白的 JSON 文件 (`{}`)。主控台提供各種服務整合的範例事件。在主控台中建立事件後，您也可以與團隊分享事件，讓測試變得更容易，結果更一致。

了解如何 在主控台中偵錯範例函數。

> **Note**
>
> 雖然在主控台中執行函數是一種快速偵錯的作法，但 自動化 測試週期是提高應用程式品質和開發速度的關鍵。

您可以在 無伺服器測試範例儲存庫 中取得測試自動化範例。下列命令列會執行自動化的 Python 整合測試範例：

```
python -m pytest -s tests/integration -v
```

雖然測試在本機執行，但它會與雲端資源互動。這些資源已使用 AWS Serverless Application Model 和 AWS SAM 命令列工具部署。測試程式碼會先擷取已部署的堆疊輸出，其中包括 API 端點、函數 ARN 和安全角色。接下來，測試會將請求傳送到 API 端點，該端點會以 Amazon S3 儲存貯體清單進行回應。此測試的執行對象完全是雲端資源，以驗證這些資源是否已完成部署、獲得保護並如預期般正常運作。

```
========================== test session starts ==========================
      platform darwin -- Python 3.10.10, pytest-7.3.1, pluggy-1.0.0
      -- /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-lambda/
venv/bin/python
      cachedir: .pytest_cache
```

```
      rootdir: /Users/t/code/aws/serverless-test-samples/python-test-samples/apigw-
lambda
      plugins: mock-3.10.0
      collected 1 item


      tests/integration/test_api_gateway.py::TestApiGateway::test_api_gateway

      --> Stack outputs:

        HelloWorldApi
        = https://p7teqs3162.execute-api.us-east-2.amazonaws.com/Prod/hello/
        > API Gateway endpoint URL for Prod stage for Hello World function

        PythonTestDemo
        = arn:aws:lambda:us-east-2:123456789012:function:testing-apigw-lambda-
PythonTestDemo-iSij8evaTdxl
        > Hello World Lambda Function ARN

        PythonTestDemoIamRole
        = arn:aws:iam::123456789012:role/testing-apigw-lambda-PythonTestDemoRole-
IZELQQ9MG4HQ
        > Implicit IAM Role created for Hello World function

      --> Found API endpoint for "testing-apigw-lambda" stack...
      --> https://p7teqs3162.execute-api.us-east-2.amazonaws.com/Prod/hello/
      API Gateway response:
      amplify-dev-123456789-deployment|myapp-prod-p-loggingbucket-123456|s3-java-
bucket-123456789
      PASSED

      ========================= 1 passed in 1.53s =========================
```

若是開發雲端原生應用程式，則在雲端進行測試可獲得以下優勢：

- 您可以測試 每項 可用的服務。
- 您一律會使用最新的服務 API 和傳回值。
- 雲端測試環境與您的生產環境非常類似。
- 測試可以涵蓋安全策略、服務配額、組態和基礎設施的具體參數。
- 每位開發人員都可以在雲端中迅速建立一或多個測試環境。
- 雲端測試可提高程式碼在生產環境中順利執行的把握度。

在雲中進行測試確實有一些缺點。部署到雲端環境的時間通常比部署到本機桌面環境的時間更長，是在雲端進行測試時最明顯的缺點。

幸運的是，AWS 無伺服器應用程式模型 (AWS SAM) Accelerate、AWS 雲端開發套件 (AWS CDK) 監看模式和 SST （第三方） 等工具可減少雲端部署反覆運算所涉及的延遲。這些工具可以監控您的基礎設施和程式碼，並自動將增量更新部署到您的雲端環境。

> **Note**
>
> 請參閱《無伺服器開發人員指南》中的如何將基礎設施建立為程式碼，以進一步了解 AWS Serverless Application Model CloudFormation和 AWS Cloud Development Kit (AWS CDK)。

不同於本機測試，在雲端進行測試需使用額外的資源，而這可能會產生服務費用。建立隔離的測試環境可能會增加 DevOps 團隊的負擔，尤其是嚴格控管帳戶和基礎設施的組織。即便如此，在面對複雜的基礎設施情境時，開發人員設定和維護複雜本機環境的時間成本，可能會與使用基礎設施即程式碼自動化工具建立的一次性測試環境相似 (或更高)。

即使有這些考量，在雲端進行測試仍然是保證無伺服器解決方案品質的 最佳作法。

透過模擬物件進行測試

使用模擬物件進行測試是一種技術，您可以在程式碼中建立替代物件，來模擬雲端服務的行為。

例如，您可以撰寫一個使用 Amazon S3 服務模擬的測試，該測試會在呼叫 CreateObject 方法時傳回特定回應。執行測試時，模擬物件會傳回該項已設計程式的回應，而不呼叫 Amazon S3 或任何其他服務端點。

模擬物件通常由模擬架構產生，以減少開發工作量。有些模擬架構是通用的，有些則是專門為 AWS SDKs 設計，例如 Moto，一種用於模擬 AWS 服務和資源的 Python 程式庫。

請注意，模擬物件與模擬器不同，模擬通常由開發人員建立或設定為測試程式碼的一部分，而模擬器是獨立的應用程式，它會以與模擬的系統相同的方式公開功能。

以下是使用模擬物件的優勢：

- 模擬物件可以模擬超出應用程式控制範圍的第三方服務，例如 API 和軟體即服務 (SaaS)供應商，無需直接存取這類服務。
- 模擬物件在測試失敗條件非常實用 (尤其當這種情況很難模擬時，例如服務中斷)。

- 模擬物件可以在設定後讓您迅速進行本機測試。
- 模擬物件可以為幾乎任何類型的物件提供替代行為，因此模擬策略可以為各種比模擬器更廣泛的服務建立涵蓋範圍。
- 當新功能或行為開放使用時，模擬物件測試便可以更迅速地做出回應。透過使用通用模擬架構，您可以在更新的 AWS SDK 可用時立即模擬新功能。

以下是模擬物件測試的缺點：

- 模擬物件通常需要進行相當複雜的設定和組態工作，尤其是在嘗試確定不同服務的傳回值以正確模擬回應的情況。
- 模擬物件是由開發人員撰寫、設定以及進行必要維護，而這會加重他們的責任。
- 您可能需要存取雲端，才能了解服務的 API 和傳回值。
- 模擬物件維護起來可能並不容易。當模擬的雲端 API 簽章發生變更，或傳回值結構描述發生變化時，便必須更新模擬。若要為了呼叫新的 API 而擴展應用程式邏輯，則也必須更新模擬物件。
- 使用模擬物件的測試可能會在桌面環境中順利通過，但在雲端中卡住。結果可能會與目前的 API 不相符。您無法對服務組態和配額進行測試。
- 模擬架構在測試或偵測 AWS Identity and Access Management (IAM) 政策或配額限制時受到限制。雖然模擬物件在授權失敗或超過配額時的模擬效果更佳，但您無法透過測試確定生產環境中實際將會發生的結果。

使用 AWS SAM CLI 進行本機測試

使用 AWS SAM CLI 在 Docker 容器中測試函式，這些容器使用與 AWS Lambda相同的執行時期環境。您可以在本機測試函式邏輯與事件處理，無需部署至雲端。如果您的函數對其他 發出 API 呼叫 AWS 服務，這些呼叫將到達真正的 AWS 資源。

使用本機容器進行測試的優勢包括：

- 使用 AWS Lambda 執行期環境進行準確的測試。
- 無需雲端部署即可快速啟用本機開發迭代。
- 支援使用熟悉的本機開發工具進行偵錯。

使用本機容器進行測試存在以下限制：

- AWS 服務 來自函數的 呼叫會與實際 AWS 資源互動，這可能會產生成本並影響生產資料。
- 需要在本機安裝並執行 Docker。

透過模擬進行測試

模擬器是在本機執行的應用程式，透過提供類似的 APIs 和傳回值來模擬 AWS 服務。LocalStack 是一種熱門的模擬工具，可提供完整的本機開發環境來測試服務間的整合運作。

LocalStack 是一種 AWS 雲端模擬器，可用於在本機測試無伺服器應用程式。您可以測試與 DynamoDB、Amazon S3 和 Amazon SQS 等服務整合的 Lambda 函數，而無需連線到實際 AWS 服務。您可以在 AWS Toolkit for VS Code 中使用 LocalStack。

以下是使用模擬器進行測試的優點：

- 模擬器可讓您快速進行本機開發迭代和測試。
- 模擬器為習慣在本機環境中開發程式碼的開發人員提供了一個熟悉的環境。例如，如果您熟悉 N 層應用程式的開發，則可能會具有類似於在生產環境、本機電腦中執行的資料庫引擎和 Web 伺服器，以提供快速、本機且隔離的測試功能。
- 模擬器不需要對雲端基礎設施 (例如開發人員的雲端帳戶) 進行任何變更，因此可以輕鬆透過現有的測試模式進行實作。
- 由於模擬器不會使用實際 AWS 資源，因此您在啟動多個 服務或讓某些資源長時間執行時，不會收到意外費用。

以下是使用模擬器進行測試的缺點：

- 模擬器可能不易進行設定和複製，尤其是用於 CI/CD 管道時。這可能會導致管理個人軟體的 IT 人員或開發人員的工作量有所提升。
- 模擬功能和 API 通常會跟不上服務更新。這可能會導致出現錯誤，因為測試的程式碼與實際的 API 不相符，且阻礙了新功能的採用。
- 模擬器需要在支援、更新、錯誤修復和功能平等方面有所提升。這些工作是模擬器開發者 (可能為第三方公司) 的責任。
- 使用模擬器的測試可能會在本機產生成功的結果，但可能會因為生產安全政策、服務間組態或超過 Lambda 配額而在雲端中產生失敗。

最佳實務

下列各節提供成功測試無伺服器應用程式的建議。

您可以在 無伺服器測試範例儲存庫 中找到測試和測試自動化的實際範例。

排定雲端測試的優先順序

在雲端進行測試可提供最可靠、精準且完整的測試涵蓋範圍。在雲端環境中執行測試不僅會全面測試商業邏輯，還會測試安全政策、服務組態、配額，以及最新的 API 簽章和傳回值。

構建程式碼以實現可測試性

將 Lambda 專屬程式碼與核心商業邏輯分隔開來，以簡化您的測試和 Lambda 函數。

您的 Lambda 函數 處理常式 應為一個小型的轉接器，它會接收事件資料，且只將重要的詳細資料傳遞給您的商業邏輯方法。您可以透過這項策略以商業邏輯為核心進行全面測試，無需擔心特定 Lambda 詳細資訊。您的 AWS Lambda 函數不需要設定複雜的環境或大量相依性，即可建立和初始化待測元件。

一般而言，您應撰寫一個處理常式，從傳入的 事件 和 內容 物件擷取和驗證資料，然後將該輸入傳送至執行商業邏輯的方法。

加快回饋迴圈的開發速度

您可以透過一些工具和技術加快回饋迴圈的開發速度。例如，AWS SAM Accelerate 和 AWS CDK 監看模式都可以縮短更新雲端環境所需的時間。

GitHub 無伺服器測試範例儲存庫 中的範例會探究其中一些技術。

我們也建議您在開發期間儘早建立和測試雲端資源，而不只是在登記至原始碼控制後才進行。這種作法可在制定解決方案時加快探索和實驗的速度。此外，從開發電腦自動化部署作業可協助您更快發現雲端組態問題，並減少更新和程式碼審查程序所浪費的人力。

全力進行整合測試

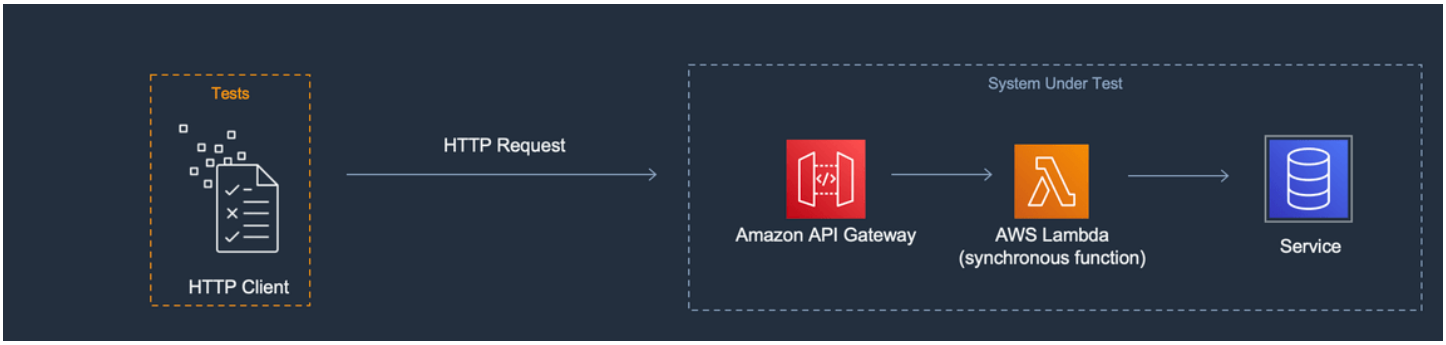
使用 Lambda 建置應用程式時，最佳作法是一起測試元件。

針對兩個 (或以上) 架構元件進行的測試稱為 整合測試。整合測試的目標不僅是了解程式碼將如何在各個元件中執行，還要了解託管程式碼的環境將會如何運作。端對端測試 屬於特殊類型的整合測試，它會驗證整個應用程式中的行為。

若要建置整合測試，請將應用程式部署至雲端環境。您可以透過本機環境或 CI/CD 管道完成這項動作。接著，請撰寫測試以訓練待測試的系統 (SUT)，並驗證預期的行為。

例如，待測試的系統可能是使用 API Gateway、Lambda 和 DynamoDB 的應用程式。測試可以對 API Gateway 端點進行合成 HTTP 呼叫，並驗證回應是否包含預期的有效負載。此測試會驗證 AWS

Lambda 程式碼是否正確，而且每個服務都已正確設定為處理請求，包括它們之間的 IAM 許可。此外，您可以將測試設計為寫入各種大小的紀錄，以驗證服務配額 (例如 DynamoDB 中的最大紀錄大小) 是否設定正確。



建立獨立的測試環境

在雲端中進行測試通常需要獨立的開發人員環境，好讓測試、資料和事件不會出現重疊的狀況。

其中一種方法是為每個開發人員提供專用 AWS 帳戶。這種作法可避免在共享程式碼基底中工作的多位開發人員在嘗試部署資源或調用 API 時可能出現的資源命名衝突。

自動化測試程序應為每個堆疊建立名稱獨一無二的資源。例如，您可以設定指令碼或 TOML 組態檔案，讓 AWS SAM CLI sam 部署或 sam 同步命令自動指定具有唯一字首的堆疊。

在某些情況下，開發人員會共用 AWS 帳戶。這可能是因為堆疊中有運作、佈建及設定時成本非常高的資源。例如，您可以共用資料庫，以便更容易正確設定和植入資料

如果開發人員共用帳戶，則應設定界限，以識別擁有權並排除重疊的狀況。其中一種作法是在堆疊名稱前面加上開發人員的使用者 ID。另一種流行的作法是根據 程式碼分支 設定堆疊。使用分支界限時，環境會獨立出來，但開發人員仍可以共用資源 (例如關聯式資料庫)。當開發人員一次處理多個分支時，這種作法便是最佳實務。

在雲端進行測試對所有測試階段 (包括單元測試、整合測試和端對端測試) 來說都很有價值。保持適當的隔離是必要的作法，但您仍會希望 QA 環境能盡可能與您的生產環境相似。因此，團隊會為 QA 環境加入變更控制程序。

生產前環境和生產環境方面，您通常會在帳戶層級設定界限，以便將工作負載隔離出來，使其不受擾鄰問題的影響，並實作最低權限安全控制以保護敏感資料的安全。工作負載具有配額。您不會希望測試消耗分配給生產環境 (擾鄰) 的配額，或可以存取客戶的資料。負載測試是應從生產堆疊中隔離出來的另一項活動。

在所有情況下，環境都應設定警示和控制項，以避免出現不必要的支出。例如，您可以限制可建立的資源類型、層級或大小，並在預估成本超過指定閾值時設定電子郵件提醒。

將模擬物件用於隔離的商業邏輯

模擬架構是撰寫快速單元測試的實用工具。當測試涵蓋複雜的內部商業邏輯 (例如數學或財務計算或模擬) 時，它們的優勢就會特別明顯。尋找具有大量測試案例或輸入變化的單元測試，其中這些輸入不會改變對其他雲端服務的模式或呼叫內容。

模擬物件單元測試所涵蓋的程式碼也應涵蓋在雲端的測試之中。此為建議作法，因為開發人員的筆記型電腦或建置機器環境的設定方式可能會不同於雲端中的生產環境。例如，使用特定輸入參數執行時，Lambda 函數使用的記憶體或時間可能比分配到的還多。或者，您的程式碼可能會含有未以相同方式 (或完全不同的方式) 設定的環境變數，且這些差異可能會導致程式碼出現不同的行為，或執行失敗。

模擬物件在整合測試中較不具優勢，因為實作必要模擬物件的人力需求會隨著連接點的數量而上升。端對端測試不應使用模擬物件，因為這些測試通常會面對無法使用模擬架構輕鬆模擬的狀態和複雜邏輯。

最後，請避免使用模擬雲端服務來驗證服務呼叫是否有正確實作。請改為在雲端中進行雲端服務呼叫，以驗證行為、組態和功能實作。

請謹慎使用模擬器

模擬器對於部分使用案例而言可能會很方便 (例如網際網路存取受限、不可靠或緩慢的開發團隊)。不過，在大多數情況下，請謹慎使用模擬器。

在不使用模擬器的情況下，您將能夠使用最新的服務功能和最新的 API 進行建置和創新。您無需等待供應商發布新版本，即可實現功能平等。您將可以減少購買和設定多個開發系統和建置機器的前期和持續支出。此外，您也可以避開許多雲端服務根本就不提供模擬器的問題。仰賴模擬的測試策略將使您無法使用這些服務 (進而衍生出成本可能更高的解決方法)，或產生未經過良好測試的程式碼和組態。

當您使用模擬進行測試時，您仍然必須在雲端中進行測試，以驗證組態，並測試與雲端服務的互動 (只能在模擬環境中進行模擬)。

在本機進行測試的難題

當您使用模擬器和模擬呼叫在本機電腦上進行測試時，當程式碼在 CI/CD 管道中從一個環境進入到另一個環境時，您可能會遇到測試不一致的情況。在電腦上驗證應用程式商業邏輯的單元測試可能無法準確測試雲端服務的關鍵面向。

以下範例提供了使用模擬物件和模擬器進行本機測試時應留意的情況：

範例：Lambda 函數建立 S3 儲存貯體

如果 Lambda 函數的邏輯仰賴於建立 S3 儲存貯體，則完整測試應確認已呼叫 Amazon S3，且儲存貯體已成功建立。

- 在模擬物件測試設定中，您可能會模擬成功回應，並可能加入測試案例來應對回應失敗的狀況。
- 在模擬測試案例中，系統可能會呼叫 CreateBucket API，但需要注意的是，進行本機呼叫的身分 不會 來自於 Lambda 服務。呼叫身分不會像在雲端中一樣擔任安全角色，因此會改用預留位置驗證 (可能會有更寬鬆的角色或使用者身分，在雲端中執行時會有所不同)。

模擬物件和模擬設定將測試 Lambda 函數在呼叫 Amazon S3 時會執行的動作；不過，這些測試將不會驗證設定的 Lambda 函數是否能成功建立 Amazon S3 儲存貯體。您必須確定指派給函數的角色具有允許函數執行 `s3:CreateBucket` 動作的附加安全政策。若沒有的話，該函數在部署到雲端環境時可能會出現失敗的狀況。

範例：Lambda 函數處理來自 Amazon SQS 佇列的訊息

如果 Amazon SQS 佇列是 Lambda 函數的來源，則完整測試應驗證訊息放入佇列時是否已成功調用 Lambda 函數。

模擬測試和模擬物件測試通常設定為直接執行 Lambda 函數程式碼，並透過傳遞 JSON 事件承載 (或還原序列化物件) 作為函數處理常式的輸入來模擬 Amazon SQS 整合。

模擬 Amazon SQS 整合的本機測試將測試 Amazon SQS 使用指定承載呼叫 Lambda 函數時，Lambda 函數會執行的動作，但測試不會驗證 Amazon SQS 在部署到雲端環境時是否會成功調用 Lambda 函數。

以下是您在使用 Amazon SQS 和 Lambda 時可能會遇到的一些組態問題範例：

- Amazon SQS 可見性逾時過低，導致原本只要調用一次，變成調用多次。
- Lambda 函數的執行角色不允許從佇列 (透過 `sqs:ReceiveMessage`、`sqs:DeleteMessage` 或 `sqs:GetQueueAttributes`) 讀取訊息。
- 傳遞至 Lambda 函數的範例事件超出 Amazon SQS 訊息大小配額。因此測試無效，因為 Amazon SQS 永遠無法傳送那麼大的訊息。

如上述範例所示，涵蓋商業邏輯但不涵蓋雲端服務之間組態的測試可能會產生不可靠的結果。

常見問答集

我有一個 Lambda 函數，它可以執行計算並傳回結果，無需呼叫任何其他服務。我真的需要在雲端進行測試嗎？

是。Lambda 函數具有可能改變測試結果的組態參數。所有的 Lambda 函數程式碼都有逾時和記憶體設定的相依項，如果未正確進行設定，可能會導致函數失敗。Lambda 政策也會向 Amazon CloudWatch 啟用標準輸出記錄。即使您的程式碼未直接呼叫 CloudWatch，也需要許可才能啟用記錄功能。此必要許可無法精確模擬。

雲端中的測試如何協助進行單元測試？ 如果測試在雲中進行且連接到其他資源，那這樣不就是整合測試嗎？

我們將 單元測試 定義為獨立在架構組件上運行的測試，但這並不會阻止測試加入可能會呼叫其他服務或使用某些網路通訊的元件。

許多無伺服器應用程式都具有可隔離進行測試的架構元件 (即使在雲端也是如此)。其中一個例子是接受輸入、處理資料並將訊息傳送至 Amazon SQS 佇列的 Lambda 函數。此函數的單元測試可能會測試輸入值是否會導致某些值出現在佇列訊息之中。

考慮使用「準備、執行、驗證」模式撰寫的測試：

- 準備：分配資源 (用於接收訊息的佇列和待測函數)。
- 執行：呼叫待測函數。
- 驗證：擷取函數發送的訊息，並驗證輸出。

模擬物件測試方法包括使用正在進行的模擬物件來模擬佇列，以及建立含有 Lambda 函數程式碼的類別或模組的進行中執行個體。驗證階段期間，佇列的訊息會從模擬物件進行擷取。

在雲端方法中，測試會針對測試目的建立 Amazon SQS 佇列，並透過設定為將隔離的 Amazon SQS 佇列作為輸出目的地使用的環境變數來部署 Lambda 函數。執行 Lambda 函數後，測試會從 Amazon SQS 佇列擷取訊息。

雲端測試會執行相同的程式碼、驗證相同的行為，並驗證應用程式功能是否正確。不過，它具有能夠驗證 Lambda 函數設定的額外優勢：IAM 角色、IAM 政策以及函數的逾時和記憶體設定。

後續步驟和資源

使用以下資源以進一步了解並探索測試的實際範例。

實作範例

GitHub 上的 無伺服器測試範例儲存庫 包含遵循本指南所述模式和最佳實務的具體測試範例。儲存庫包含前幾節所述之模擬物件、模擬和雲端測試程序的範例程式碼和引導式逐步解說。使用此儲存庫來取得來自 的最新無伺服器測試指引 AWS。

深入閱讀

請造訪 Serverless Land 以存取最新的部落格、影片和無 AWS 伺服器技術訓練。

也建議閱讀下列 AWS 部落格文章：

- 使用 Accelerate 加速無伺服器開發 AWS SAM(AWS 部落格文章）
- 使用 CDK Watch 提高開發速度 (AWS 部落格文章）
- 模擬服務與 AWS Step Functions Local 的整合 (AWS 部落格文章）
- 測試無伺服器應用程式入門 (AWS 部落格文章）

工具

- AWS SAM – 測試和偵錯無伺服器應用程式
- AWS SAM – 與自動化測試整合
- Lambda：在 Lambda 主控台中測試 Lambda 函數

使用 Lambda SnapStart 改善啟動效能

Lambda SnapStart 可提供低至一秒的啟動效能，通常不會變更函數程式碼。SnapStart 可讓您更輕鬆地建置高回應和可擴展應用程式，而無需佈建資源或實作複雜的效能最佳化。

啟動延遲的最大歸因 (通常稱為冷啟動時間) 是 Lambda 花費在初始化函數的時間，其中包括載入函數的程式碼、啟動執行階段以及初始化函數程式碼。使用 SnapStart，Lambda 會在您發佈函數版本時初始化您的函數。Lambda 會擷取已初始化執行環境的記憶體和磁碟狀態的 Firecracker microVM 快照、將快照加密，並智慧地進行快取以最佳化擷取延遲。

為了確保彈性，Lambda 會維護每個快照的多個副本。Lambda 會使用最新的執行時期和安全更新來自動修補快照及其副本。第一次調用函數版本時且呼叫縱向擴展時，Lambda 會從快取的快照恢復新的執行環境，而不是從頭開始執行，進而改善啟動延遲。

> **Important**
>
> 如果您的應用程式依賴狀態的唯一性，則必須評估函數程式碼，並確認其對快照作業具有復原能力。如需詳細資訊，請參閱使用 Lambda SnapStart 處理唯一性。

主題

- 何時使用 SnapStart
- 支援的功能和限制
- 支援的區域
- 相容性考量
- SnapStart 定價
- 啟動及管理 Lambda SnapStart
- 使用 Lambda SnapStart 處理唯一性
- 在 Lambda 函數快照之前或之後實作程式碼
- 監控 Lambda SnapStart
- Lambda SnapStart 的安全模型
- 最大化 Lambda SnapStart 效能
- 對 Lambda 函數的 SnapStart 錯誤進行故障診斷

何時使用 SnapStart

Lambda SnapStart 旨在解決一次性初始化程式碼帶來的延遲差異，例如載入模組相依項或架構。這些操作有時可能需要幾秒鐘才能在初始調用期間完成。在最佳情況下，使用 SnapStart 將此延遲從數秒減少到低至次秒。SnapStart 搭配大規模函數調用使用時效果最佳。不常調用的函數效能改進效果可能不會相同。

SnapStart 特別適用於兩種主要應用程式類型：

- 延遲敏感 API 和使用者流程：屬於關鍵 API 端點或面向使用者的流程的函數可受益於 SnapStart 降低的延遲和改善的回應時間。
- 延遲敏感資料處理工作流程：使用 Lambda 函數的限時資料處理工作流程可以透過減少異常函數初始化延遲來實現更佳的輸送量。

佈建並行可讓函數保持初始化，並準備好在兩位數的毫秒內回應。如果應用程式有嚴格的冷啟動延遲要求，且 SnapStart 無法充分解決，請使用佈建並行。

支援的功能和限制

SnapStart 適用於下列 Lambda 受管執行時期：

- Java 11 及更高版本
- Python 3.12 及更高版本
- .NET 8 及更高版本。如果使用的是適用於 .NET 的 Lambda Annotations 架構，請升級至 Amazon.Lambda.Annotations 1.6.0 版或更新版本，以確保與 SnapStart 的相容性。

不支援其他受管執行期 (例如 `nodejs24.x` 和 `ruby3.4`)、僅限作業系統的執行期 和容器映像。

SnapStart 不支援佈建並行、Amazon Elastic File System (Amazon EFS) 或大於 512 MB 的暫時性儲存。

> **Note**
>
> 您只能對指向版本的已發佈函數版本和別名使用 SnapStart。不能對函數的未發佈版本 ($LATEST) 使用 SnapStart。

支援的區域

Lambda SnapStart 已在所有商業區域中推出，但亞太區域 (紐西蘭) 與亞太區域 (台北) 除外。

相容性考量

透過 SnapStart，Lambda 會使用單一快照做為多個執行環境的初始狀態。如果您的函數在初始化階段使用以下任何一項，那麼您可能需要在使用 SnapStart 之前進行一些變更：

唯一性

如果您的初始化程式碼會產生包含在快照中的唯一內容，則在跨執行環境中重複使用該內容時，該內容可能不是唯一的。若要在使用 SnapStart 時保持唯一性，您必須在初始化後產生唯一的內容。這包括唯一 ID、唯一密碼，以及用來產生偽隨機性的熵。若要了解如何還原唯一性，請參閱：使用 Lambda SnapStart 處理唯一性。

網路連線

Lambda 從快照恢復函數時，無法保證函數在初始化階段建立的連線狀態。驗證網路連線的狀態，並視需要重新建立。在大多數情況下， AWS 軟體開發套件建立的網路連線會自動恢復。針對其他連線，請檢閱最佳實務。

暫存資料

某些函數會在初始化階段下載或初始化暫存資料，例如臨時憑證或快取的時間戳記。請先重新整理函數處理常式中的暫時性資料再使用處理常式，即使沒有使用 SnapStart 也是如此。

SnapStart 定價

> **Note**
>
> 對於 Java 受管執行時期，SnapStart 無需額外費用。我們會根據函數的請求數量、程式碼執行的時間，以及為函數配置的記憶體向您收費。

使用 SnapStart 的成本包括下列項目：

- 快取：對於在啟用 SnapStart 的情況下發布的每個函數版本，需要支付快取和維護快照的成本。價格取決於您配置給函數的記憶體容量。需要支付最少 3 小時的費用。只要您的函式處於作用

中狀態，就會持續產生費用。使用 ListVersionsByFunction API 動作來識別函式版本，然後使用 DeleteFunction 刪除未使用的版本。若要自動刪除未使用的函數版本，請參閱 Serverless Land 上的 Lambda 版本清除模式。

- 還原：每次從快照中還原函數執行個體時，都會支付還原費用。價格取決於您配置給函數的記憶體容量。

如同所有 Lambda 函數一樣，持續時間費用適用於函數處理常式中執行的程式碼。對於 SnapStart 函數，持續時間費用也適用於在處理常式之外宣告的初始化程式碼、執行時期進行載入所花的時間以及在執行時期勾點中執行的任何程式碼。持續時間是從程式碼開始執行的時間開始計算，直到傳回資料或結束為止，四捨五入至最接近的 1 ms。Lambda 會維護快照的快取副本以實現彈性，並自動套用軟體更新，例如執行時期升級和安全修補程式。每次 Lambda 重新執行初始化程式碼以套用軟體更新時，都會收取費用。

如需有關使用 SnapStart 的成本詳細資訊，請參閱 AWS Lambda 定價。

啟動及管理 Lambda SnapStart

若要使用 SnapStart，請對新的或現有的 Lambda 函數啟動 SnapStart。然後發布並調用函數版本。

主題

- 啟動 SnapStart (主控台)
- 啟動 SnapStart (AWS CLI)
- 啟動 SnapStart (API)
- Lambda SnapStart 和函數狀態
- 更新快照
- 使用 SnapStart AWS SDKs
- 搭配 CloudFormation AWS SAM、 和 使用 SnapStart AWS CDK
- 刪除快照

啟動 SnapStart (主控台)

為函數啟動 SnapStart

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 選擇組態，然後選擇一般組態。
4. 在一般組態窗格中，選擇編輯。
5. 在編輯基本設定頁面中，為 SnapStart 選擇已發布的版本。
6. 選擇 儲存 。
7. 發佈函數版本。Lambda 會初始化程式碼、建立初始化執行環境的快照，然後快取快照以實現低延遲存取。
8. 調用函數版本。

啟動 SnapStart (AWS CLI)

為現有函數啟動 SnapStart

1. 執行 update-function-configuration 命令搭配 --snap-start 選項來更新函數組態。

```
aws lambda update-function-configuration \
  --function-name my-function \
  --snap-start ApplyOn=PublishedVersions
```

2. 使用 publish-version 命令發佈函數版本。

```
aws lambda publish-version \
  --function-name my-function
```

3. 執行 get-function-configuration 命令並指定版本編號，確認已為函數版本啟動 SnapStart。以下範例命令指定第 1 版。

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

如果回應顯示 OptimizationStatus 為 On 且 State 為 Active，便會啟動 SnapStart，且快照可用於指定的函數版本。

```
"SnapStart": {
    "ApplyOn": "PublishedVersions",
    "OptimizationStatus": "On"
 },
 "State": "Active",
```

4. 執行 invoke 命令並指定版本來調用函數版本。以下範例調用第 1 版。

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

建立新函數時啟動 SnapStart

1. 執行 create-function 命令搭配 --snap-start 選項來建立函數。針對 --role，請指定執行角色的 Amazon Resource Name (ARN)。

```
aws lambda create-function \
  --function-name my-function \
  --runtime "java25" \
  --zip-file fileb://my-function.zip \
  --handler my-function.handler \
  --role arn:aws:iam::111122223333:role/lambda-ex \
  --snap-start ApplyOn=PublishedVersions
```

2. 使用 publish-version 命令來建立版本。

```
aws lambda publish-version \
  --function-name my-function
```

3. 執行 get-function-configuration 命令並指定版本編號，確認已為函數版本啟動 SnapStart。以下範例命令指定第 1 版。

```
aws lambda get-function-configuration \
  --function-name my-function:1
```

如果回應顯示 OptimizationStatus 為 On 且 State 為 Active，便會啟動 SnapStart，且快照可用於指定的函數版本。

```
"SnapStart": {
     "ApplyOn": "PublishedVersions",
     "OptimizationStatus": "On"
  },
  "State": "Active",
```

4. 執行 invoke 命令並指定版本來調用函數版本。以下範例調用第 1 版。

```
aws lambda invoke \
  --cli-binary-format raw-in-base64-out \
  --function-name my-function:1 \
  --payload '{ "name": "Bob" }' \
  response.json
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

啟動 SnapStart (API)

啟動 SnapStart

1. 執行以下任意一項：
   - 使用 CreateFunction API 動作搭配 SnapStart 參數，建立啟動 SnapStart 的新函數。
   - 使用 UpdateFunctionConfiguration 動作搭配 SnapStart 參數，為現有函數啟動 SnapStart。
2. 使用 PublishVersion 動作發佈函數版本。Lambda 會初始化程式碼、建立初始化執行環境的快照，然後快取快照以實現低延遲存取。
3. 使用 GetFunctionConfiguration 動作，確認已為函數版本啟動 SnapStart。指定版本編號以確認已為該版本啟動 SnapStart。如果回應顯示 OptimizationStatus 為 `On` 且 State 為 `Active`，便會啟動 SnapStart，且快照可用於指定的函數版本。

```
"SnapStart": {
        "ApplyOn": "PublishedVersions",
        "OptimizationStatus": "On"
    },
    "State": "Active",
```

4. 使用 Invoke 動作調用函數版本。

Lambda SnapStart 和函數狀態

使用 SnapStart時，可能會出現下列函數狀態。

待定

Lambda 正在初始化您的程式碼，並擷取初始化執行環境的快照。在函數上操作的任何調用或其他 API 動作都會失敗。

作用中

快照建立完成，您可以調用函數。若要使用 SnapStart，您必須調用已發佈的函數版本，而不是未發佈版本 ($LATEST)。

非作用中

當 Lambda 定期重新產生函式快照以套用軟體更新時，可能會出現 `Inactive` 狀態。在此執行個體中，如果函數無法初始化，則函數會進入`Inactive`狀態。

對於使用 Java 執行時期的函式，若已有 14 天未調用快照，Lambda 會將其刪除。如果您在 14 天後調用函數版本，Lambda 會傳回 `SnapStartNotReadyException` 回應並開始初始化新快照。等待函數版本進入 `Active` 狀態，然後再次調用它。

失敗

Lambda 在執行初始化程式碼或建立快照時發生錯誤。

更新快照

Lambda 會為每個已發佈的函數版本建立快照。若要更新快照，請發佈新函數版本。

使用 SnapStart AWS SDKs

若要從函數進行 AWS SDK 呼叫，Lambda 會透過擔任函數的執行角色來產生一組暫時性的登入資料。這些憑證在函數調用期間可當成環境變數使用。您不需要直接在程式碼中提供 SDK 的憑證。根據預設，憑證提供者鏈會依序檢查您可以設定憑證的每個位置，並選擇第一個可用的位置，通常是環境變數 (`AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY` 和 `AWS_SESSION_TOKEN`)。

> **Note**
>
> SnapStart 啟動時，Lambda 執行時期會自動使用容器憑證 (`AWS_CONTAINER_CREDENTIALS_FULL_URI` 和 `AWS_CONTAINER_AUTHORIZATION_TOKEN`)，而不是存取金鑰環境變數。這樣可防止憑證在函數還原之前過期。

搭配 CloudFormation AWS SAM、 和 使用 SnapStart AWS CDK

- AWS CloudFormation：在範本中宣告 SnapStart 實體。

- AWS Serverless Application Model (AWS SAM)：宣告範本中的 SnapStart 屬性。
- AWS Cloud Development Kit (AWS CDK)：使用 SnapStartProperty 類型。

刪除快照

Lambda 會在發生以下情況時刪除快照：

- 您刪除了函數或函數版本。
- (僅限 Java 執行時期) 您 14 天內都沒有調用函數版本。連續 14 天都沒有呼叫之後，函數版本會轉換為 Inactive (非作用中) 狀態。如果您在 14 天後調用函數版本，Lambda 會傳回 `SnapStartNotReadyException` 回應並開始初始化新快照。等待函數版本進入 Active (作用中) 狀態，然後再次調用它。

根據一般資料保護規範 (GDPR)，Lambda 會移除與已刪除快照相關聯的所有資源。

使用 Lambda SnapStart 處理唯一性

SnapStart 函數上的呼叫擴展時，Lambda 會使用單一初始化快照來恢復多個執行環境。如果您的初始化程式碼會產生包含在快照中的唯一內容，則在跨執行環境中重複使用該內容時，該內容可能不是唯一的。若要在使用 SnapStart 時保持唯一性，您必須在初始化後產生唯一的內容。這包括唯一 ID、唯一密碼，以及用來產生偽隨機性的熵。

以下是維持程式碼唯一性的建議最佳實務。對於 Java 函數，Lambda 還提供開放原始碼 SnapStart 掃描工具，協助檢查確保唯一性的程式碼。如果您在初始化階段產生唯一資料，便可使用執行階段掛鉤來還原唯一性。使用執行階段掛鉤，您可以在 Lambda 建立快照前執行特定程式碼，或在 Lambda 從快照恢復函數後立即執行特定程式碼。

避免儲存初始化期間依賴唯一性的狀態

在函數的初始化階段，請避免快取預定為唯一的資料，例如產生用於日誌記錄的唯一 ID 或設定隨機函數的種子。相反地，建議您在函數處理常式中產生唯一的資料或設定隨機函數的種子，或使用執行時期勾點進行這些操作。

以下程式碼範例示範如何在函數處理常式中產生 UUID。

Java

Example- 在函數處理常式中產生唯一 ID

```
import java.util.UUID;
  public class Handler implements RequestHandler<String, String> {
    private static UUID uniqueSandboxId = null;
    @Override
    public String handleRequest(String event, Context context) {
      if (uniqueSandboxId == null)
        uniqueSandboxId = UUID.randomUUID();
      System.out.println("Unique Sandbox Id: " + uniqueSandboxId);
      return "Hello, World!";
    }
  }
```

Python

Example- 在函數處理常式中產生唯一 ID

```
import json
```

```
import random
import time

unique_number = None

def lambda_handler(event, context):
    seed = int(time.time() * 1000)
    random.seed(seed)
    global unique_number
    if not unique_number:
        unique_number = random.randint(1, 10000)

    print("Unique number: ", unique_number)

    return "Hello, World!"
```

.NET

Example- 在函數處理常式中產生唯一 ID

```
namespace Example;
public class SnapstartExample
{
    private Guid _myExecutionEnvironmentGuid;
    public SnapstartExample()
    {
        // This GUID is set for non-restore use cases, such as testing or if
 SnapStart is turned off
        _myExecutionEnvironmentGuid = new Guid();
        // Register the method which will run after each restore. You may need to
 update Amazon.Lambda.Core to see this
        Amazon.Lambda.Core.SnapshotRestore.RegisterAfterRestore(MyAfterRestore);
    }

    private ValueTask MyAfterRestore()
    {
        // After restoring this snapshot to a new execution environment, update the
 GUID
        _myExecutionEnvironmentGuid = new Guid();
        return ValueTask.CompletedTask;
    }

    public string Handler()
```

```
    {
        return $"Hello World! My Execution Environment GUID is
 {_myExecutionEnvironmentGuid}";
    }
}
```

使用密碼編譯安全的虛擬隨機數產生器 (CSPRNGs)

如果您的應用程式依賴隨機性，建議您使用密碼編譯安全隨機數產生器 (CSPRNGs)。除了 OpenSSL 1.0.2 之外，Lambda 受管執行時期還包含以下內建 CSPRNG：

- Java：`java.security.SecureRandom`
- Python：`random.SystemRandom`
- .NET：`System.Security.Cryptography.RandomNumberGenerator`

此軟體總是會從 `/dev/random` 或 `/dev/urandom` 取得隨機數，也會透過 SnapStart 保持隨機性。

AWS 從下表指定的最低版本開始，密碼編譯程式庫會自動維持 SnapStart 的隨機性。如果您在 Lambda 函數中使用這些程式庫，請確定您使用以下最低版本或更新版本：

程式庫	支援的最低版本 (x86)	支援的最低版本 (ARM)
AWS libcrypto (AWS-LC)	1.16.0	1.30.0
AWS libcrypto FIPS	2.0.13	2.0.13

如果您透過以下程式庫將上述密碼編譯程式庫封裝為 Lambda 函數的遞移相依項，請務必使用以下最低版本或更新版本：

程式庫	支援的最低版本 (x86)	支援的最低版本 (ARM)
AWS SDK for Java 2.x	2.23.20	2.26.12
AWS Java 的常見執行期	0.29.8	0.29.25
Amazon Corretto Crypto Provider	2.4.1	2.4.1

程式庫	支援的最低版本 (x86)	支援的最低版本 (ARM)
Amazon Corretto Crypto Provider FIPS	2.4.1	2.4.1

以下範例示範如何使用 CSPRNG 來保證唯一數字序列，即使從快照還原函數也是如此。

Java

Example– java.security.SecureRandom

```
import java.security.SecureRandom;
  public class Handler implements RequestHandler<String, String> {
    private static SecureRandom rng = new SecureRandom();
    @Override
    public String handleRequest(String event, Context context) {
      for (int i = 0; i < 10; i++) {
        System.out.println(rng.next());
      }
      return "Hello, World!";
    }
  }
```

Python

Example：random.SystemRandom

```
import json
import random

secure_rng = random.SystemRandom()

def lambda_handler(event, context):
    random_numbers = [secure_rng.random() for _ in range(10)]

    for number in random_numbers:
        print(number)

    return "Hello, World!"
```

.NET

Example：RandomNumberGenerator

```
using Amazon.Lambda.Core;
using System.Security.Cryptography;
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeriali

namespace DotnetSecureRandom;

public class Function
{
    public string FunctionHandler()
    {
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            byte[] randomUnsignedInteger32Bytes = new byte[4];
            for (int i = 0; i < 10; i++)
            {
                rng.GetBytes(randomUnsignedInteger32Bytes);
                int randomInt32 = BitConverter.ToInt32(randomUnsignedInteger32Bytes,
 0);
                Console.WriteLine("{0:G}", randomInt32);
            }
        }
        return "Hello World!";
    }
}
```

SnapStart 掃描工具 (僅限 Java)

Lambda 提供 Java 掃描工具，可協助檢查確保唯一性的程式碼。SnapStart 掃描工具是一種開放原始碼 SpotBugs 外掛程式，可針對一組規則執行靜態分析。掃描工具有助於識別可能會破壞有關唯一性假設的潛在程式碼實作。如需安裝指示和掃描工具執行的檢查清單，請參閱 GitHub 上的 aws-lambda-snapstart-java-rules 儲存庫。

若要進一步了解如何使用 SnapStart 處理唯一性，請參閱 AWS 運算部落格上的使用 AWS Lambda SnapStart 更快速啟動。

在 Lambda 函數快照之前或之後實作程式碼

您可以在 Lambda 建立快照之前或 Lambda 從快照恢復函數之後，使用執行階段掛鉤來實作程式碼。執行時期勾點有許多用途，包括：

- 清除和初始化：建立快照之前，您可以使用執行時期勾點來執行清除或資源釋放操作。還原快照後，您可以使用執行時期勾點重新初始化快照中未擷取的任何資源或狀態。
- 動態組態：您可以在建立快照之前或還原之後，使用執行時期勾點動態更新組態或其他中繼資料。如果函數需要適應執行時期環境的變更，這很有用。
- 外部整合：您可以使用執行時期勾點與外部服務或系統整合，作為檢查點和還原程序的一部分，藉此完成傳送通知或更新外部狀態等操作。
- 效能調校：您可以使用執行時期勾點來微調函數的啟動順序，例如預先載入相依項。如需更多詳細資訊，請參閱 效能調校。

以下幾頁說明如何針對您偏好的執行時期實作執行時期勾點。

主題

- Lambda SnapStart 執行時期勾點 (Java)
- Lambda SnapStart 執行時期勾點 (Python)
- Lambda SnapStart 執行時期勾點 (.NET)

Lambda SnapStart 執行時期勾點 (Java)

您可以在 Lambda 建立快照之前或 Lambda 從快照恢復函數之後，使用執行階段掛鉤來實作程式碼。執行階段掛鉤可做為開放原始碼 Coordinated Restore at Checkpoint (CRaC) 專案的一部分使用。目前正在針對 Java 開發套件 (OpenJDK) 開發 CRaC。如需如何搭配參考應用程式使用 CRaC 的範例，請參閱 GitHub 上的 CRaC 儲存庫。CRaC 使用三個主要元素：

- `Resource` - 具有兩種方法 (`beforeCheckpoint()` 和 `afterRestore()`) 的介面。使用這些方法來實作您想在快照建立之前和還原之後執行的程式碼。
- `Context <R extends Resource>` - 若要接收檢查點和還原的通知，必須透過 `Context` 註冊 `Resource`。
- `Core` - 協調服務，透過靜態方法 `Core.getGlobalContext()` 提供預設的全域 `Context`。

如需 `Context` 和 `Resource` 的詳細資訊，請參閱 CRaC 文件中的套件 org.crac。

使用下列步驟，透過 org.crac 套件實作執行階段掛鉤。Lambda 執行階段包含自訂的 CRaC 內容實作，可在檢查點之前和還原之後呼叫執行階段掛鉤。

執行時期勾點註冊和執行

Lambda 執行執行時期勾點的順序取決於勾點的註冊順序。註冊順序遵循程式碼中的匯入、定義或執行順序。

- `beforeCheckpoint()`：依與註冊順序相反的順序執行
- `afterRestore()`：依註冊順序執行

確定已正確匯入所有已註冊的勾點，並包含在函數的程式碼中。如果您在單獨的檔案或模組中註冊執行時期勾點，則必須確保直接匯入相關模組，或將其作為較大套件的一部分匯入函數的處理常式檔案中。如果未在函數處理常式中匯入相關檔案或模組，則 Lambda 會忽略相關執行時期勾點。

> **Note**
>
> Lambda 建立快照時，初始化程式碼最多可能會執行 15 分鐘。時間上限為 130 秒或設定的函數逾時 (最長 900 秒)，以較高者為準。您的 `beforeCheckpoint()` 執行時間掛鉤會計入初始化程式碼時間限制。Lambda 還原快照時，執行時期必須載入，且 `afterRestore()` 執行時期勾點必須在逾時限制 (10 秒) 內完成。否則，您將收到 SnapStartTimeoutException 訊息。

步驟 1：更新建置組態

將 `org.crac` 相依性新增到建置組態。以下範例使用 Gradle。如需其他建置系統的範例，請參閱 Apache Maven 文件。

```
dependencies {
    compile group: 'com.amazonaws', name: 'aws-lambda-java-core', version: '1.2.1'
    # All other project dependecies go here:
    # ...
    # Then, add the org.crac dependency:
 implementation group: 'org.crac', name: 'crac', version: '1.4.0'
}
```

步驟 2：更新 Lambda 處理常式

Lambda 函數處理常式是您的函數程式碼中處理事件的方法。當有人呼叫您的函數時，Lambda 會執行處理常式方法。函數會執行，直到處理常式傳回回應、結束或逾時為止。

如需詳細資訊，請參閱在 Java 中定義 Lambda 函數處理常式。

以下處理常式範例示範如何在檢查點之前 (`beforeCheckpoint()`) 和還原之後 (`afterRestore()`) 執行程式碼。此處理常式也會向執行階段管理的全域 `Context` 註冊 `Resource`。

> **Note**
>
> Lambda 建立快照時，初始化程式碼最多可能會執行 15 分鐘。時間上限為 130 秒或設定的函數逾時 (最長 900 秒)，以較高者為準。您的 `beforeCheckpoint()` 執行時間掛鉤會計入初始化程式碼時間限制。Lambda 還原快照時，執行階段 (JVM) 必須載入，且 `afterRestore()` 執行階段掛鉤必須在逾時限制 (10 秒) 內完成。否則，您將收到 SnapStartTimeoutException 訊息。

```
...
  import org.crac.Resource;
  import org.crac.Core;
  ...
public class CRaCDemo implements RequestStreamHandler, Resource {
    public CRaCDemo() {
      Core.getGlobalContext().register(this);
    }
    public String handleRequest(String name, Context context) throws IOException {
      System.out.println("Handler execution");
      return "Hello " + name;
    }
    @Override
    public void beforeCheckpoint(org.crac.Context<? extends Resource> context)
        throws Exception {
      System.out.println("Before checkpoint");
    }
    @Override
    public void afterRestore(org.crac.Context<? extends Resource> context)
        throws Exception {
      System.out.println("After restore");
```

Context 僅會對已註冊物件維持 WeakReference。如果 Resource 是回收的垃圾，則執行階段掛鉤程式不會執行。您的程式碼必須維持對 Resource 的強式參考，才能保證執行階段掛鉤會執行。

以下是須避免的兩個模式範例：

Example- 沒有強式參考的物件

```
Core.getGlobalContext().register( new MyResource() );
```

Example- 匿名類別的物件

```
Core.getGlobalContext().register( new Resource() {

   @Override
   public void afterRestore(Context<? extends Resource> context) throws Exception {
    // ...
   }

   @Override
   public void beforeCheckpoint(Context<? extends Resource> context) throws Exception {
    // ...
   }

} );
```

而是維持強式參考。在下列範例中，已註冊的資源不是回收的垃圾，且執行階段掛鉤會一致地執行。

Example- 有強式參考的物件

```
Resource myResource = new MyResource(); // This reference must be maintained to prevent
 the registered resource from being garbage collected
Core.getGlobalContext().register( myResource );
```

Lambda SnapStart 執行時期勾點 (Python)

您可以在 Lambda 建立快照之前或 Lambda 從快照恢復函數之後，使用執行階段掛鉤來實作程式碼。Python 執行時期勾點作為 Snapshot Restore for Python library 的一部分提供，該程式庫包含在 Python 受管執行時期中。此程式庫提供兩個裝飾項目，可用於定義執行時期勾點：

- `@register_before_snapshot`：用於您要在 Lambda 建立快照之前執行的函數。
- `@register_after_restore`：用於您要在 Lambda 從快照恢復函數時執行的函數。

您也可以使用以下方法來註冊執行時期勾點的可呼叫項：

- `register_before_snapshot(func, *args, **kwargs)`
- `register_after_restore(func, *args, **kwargs)`

執行時期勾點註冊和執行

Lambda 執行執行時期勾點的順序取決於勾點的註冊順序：

- 快照之前：依與註冊順序相反的順序執行
- 快照之後：依註冊順序執行

執行時期勾點註冊的順序取決於您如何定義勾點。使用裝飾項目 (`@register_before_snapshot` 和 `@register_after_restore`) 時，註冊順序會遵循程式碼中的匯入、定義或執行順序。如果您需要對註冊順序進行更多控制，請使用 `register_before_snapshot()` 和 `register_after_restore()` 方法，而不要使用裝飾項目。

確定已正確匯入所有已註冊的勾點，並包含在函數的程式碼中。如果您在單獨的檔案或模組中註冊執行時期勾點，則必須確保直接匯入相關模組，或將其作為較大套件的一部分匯入函數的處理常式檔案中。如果未在函數處理常式中匯入相關檔案或模組，則 Lambda 會忽略相關執行時期勾點。

> **Note**
>
> Lambda 建立快照時，初始化程式碼最多可能會執行 15 分鐘。時間上限為 130 秒或設定的函數逾時 (最長 900 秒)，以較高者為準。您的 `@register_before_snapshot` 執行時間掛鉤會計入初始化程式碼時間限制。Lambda 還原快照時，執行時期必須載入，且 `@register_after_restore` 執行時期勾點必須在逾時限制 (10 秒) 內完成。否則，您將收到 SnapStartTimeoutException 訊息。

範例

以下處理常式範例示範如何在檢查點之前 (`@register_before_snapshot`) 和還原之後 (`@register_after_restore`) 執行程式碼。

```
from snapshot_restore_py import register_before_snapshot, register_after_restore

def lambda_handler(event, context):
```

```
    # Handler code

@register_before_snapshot
def before_checkpoint():
    # Logic to be executed before taking snapshots

@register_after_restore
def after_restore():
    # Logic to be executed after restore
```

如需更多範例，請參閱 AWS GitHub 儲存庫中的 Snapshot Restore for Python。

Lambda SnapStart 執行時期勾點 (.NET)

您可以在 Lambda 建立快照之前或 Lambda 從快照恢復函數之後，使用執行階段掛鉤來實作程式碼。Amazon.Lambda.Core 套件 (2.5.0 版或更新版本) 提供 .NET 執行時期勾點。此程式庫提供兩種定義執行時期勾點的方法：

- `RegisterBeforeSnapshot()`：建立快照前執行的程式碼
- `RegisterAfterSnapshot()`：從快照恢復函數後執行的程式碼

> **Note**
>
> 如果使用的是適用於 .NET 的 Lambda Annotations 架構，請升級至 Amazon.Lambda.Annotations 1.6.0 版或更新版本，以確保與 SnapStart 的相容性。

執行時期勾點註冊和執行

在初始化程式碼中註冊勾點。根據 Lambda 函數的執行模型考慮下列準則：

- 對於可執行的組裝方法，請在使用 `RunAsync` 啟動 Lambda 引導程序之前註冊勾點。
- 對於類別程式庫方法，請在處理常式類別建構函數中註冊勾點。
- 對於 ASP.NET Core 應用程式，請先註冊勾點，再呼叫 `WebApplications.Run` 方法。

若要在 .NET 中註冊 SnapStart 的執行時期勾點，請使用以下方法：

```
Amazon.Lambda.Core.SnapshotRestore.RegisterBeforeSnapshot(BeforeCheckpoint);
```

```
Amazon.Lambda.Core.SnapshotRestore.RegisterAfterRestore(AfterCheckpoint);
```

註冊多個勾點類型時，Lambda 執行執行執行時期勾點的順序取決於勾點的註冊順序：

- `RegisterBeforeSnapshot()`：依與註冊順序相反的順序執行
- `RegisterAfterSnapshot()`：依註冊順序執行

> **Note**
>
> Lambda 建立快照時，初始化程式碼最多可能會執行 15 分鐘。時間上限為 130 秒或設定的函數逾時 (最長 900 秒)，以較高者為準。您的 `RegisterBeforeSnapshot()` 執行時間掛鉤會計入初始化程式碼時間限制。Lambda 還原快照時，執行時期必須載入，且 `RegisterAfterSnapshot()` 執行時期勾點必須在逾時限制 (10 秒) 內完成。否則，您將收到 SnapStartTimeoutException 訊息。

範例

以下函數範例示範如何在檢查點之前 (`RegisterBeforeSnapshot`) 和還原之後 (`RegisterAfterRestore`) 執行程式碼。

```
public class SampleClass
{
    public SampleClass()
    {
        Amazon.Lambda.Core.SnapshotRestore.RegisterBeforeSnapshot(BeforeCheckpoint);
        Amazon.Lambda.Core.SnapshotRestore.RegisterAfterRestore(AfterCheckpoint);
    }

    private ValueTask BeforeCheckpoint()
    {
        // Add logic to be executed before taking the snapshot
        return ValueTask.CompletedTask;
    }

    private ValueTask AfterCheckpoint()
    {
        // Add logic to be executed after restoring the snapshot
        return ValueTask.CompletedTask;
    }
```

```
    public APIGatewayProxyResponse FunctionHandler(APIGatewayProxyRequest request,
 ILambdaContext context)
    {
        // Add business logic

        return new APIGatewayProxyResponse
        {
            StatusCode = 200
        };
    }
}
```

監控 Lambda SnapStart

您可 Lambda 使用 Amazon CloudWatch、AWS X-Ray 和 使用遙測 API 即時存取延伸功能的遙測資料 來監控 Lambda SnapStart 函數。

> **Note**
>
> `AWS_LAMBDA_LOG_GROUP_NAME` 和 `AWS_LAMBDA_LOG_STREAM_NAME` 環境變數無法在 Lambda SnapStart 函數中使用。

了解 SnapStart 的記錄和計費行為

SnapStart 函數的 CloudWatch 日誌串流格式有一些差異：

- 初始化日誌：建立新的執行環境時，`REPORT` 不會含有 `Init Duration` 欄位。這是因為 Lambda 會在您建立版本時初始化 SnapStart 函數，而非在函數調用期間。若為 SnapStart 函數，`Init Duration` 欄位位於 `INIT_REPORT` 記錄中。此記錄會顯示 初始化階段 的持續時間詳細資訊，包括任何 `beforeCheckpoint` 執行階段掛鉤的持續時間。
- 調用日誌：建立新的執行環境時，`REPORT` 會含有 `Restore Duration` 和 `Billed Restore Duration` 欄位：
  - `Restore Duration`：Lambda 還原快照、載入執行時期和執行任何還原後執行時期勾點所需的時間。還原快照的程序可能包括在 MicroVM 以外的活動上花費的時間。此時間在 `Restore Duration` 中報告。
  - `Billed Restore Duration`：Lambda 載入執行時期和執行任何還原後執行時期勾點所需的時間。

> **Note**
>
> 如同所有 Lambda 函數一樣，持續時間費用適用於函數處理常式中執行的程式碼。對於 SnapStart 函數，持續時間費用也適用於在處理常式之外宣告的初始化程式碼、執行時期進行載入所花的時間以及在執行時期勾點中執行的任何程式碼。

冷啟動持續時間是 `Restore Duration` + `Duration` 的總和。

下列 Lambda Insights 查詢範例傳回了 SnapStart 函數的延遲百分位數。如需 Lambda Insights 查詢的詳細資訊，請參閱：使用查詢故障排除函式的範例工作流程。

```
filter @type = "REPORT"
  | parse @log /\d+:\/aws\/lambda\/(?<function>.*)/
  | parse @message /Restore Duration: (?<restoreDuration>.*?) ms/
  | stats
count(*) as invocations,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 50) as p50,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 90) as p90,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99) as p99,
pct(@duration+coalesce(@initDuration,0)+coalesce(restoreDuration,0), 99.9) as p99.9
group by function, (ispresent(@initDuration) or ispresent(restoreDuration)) as
 coldstart
  | sort by coldstart desc
```

適用於 SnapStart 的 X-Ray 作用中追蹤

您可以使用 X-Ray 來追蹤 Lambda SnapStart 函數的請求。SnapStart 函數的 X-Ray 子區段有一些差異：

- SnapStart 函數沒有 `Initialization` 子區段。
- `Restore` 子區段會顯示 Lambda 還原快照、載入執行時期和執行任何還原後執行時期勾點所需的時間。還原快照的程序可能包括在 MicroVM 以外的活動上花費的時間。此時間在 `Restore` 子區段中報告。您不需要為在 MicroVM 外還原快照所花費的時間付費。

SnapStart 的遙測 API 事件

Lambda 會將下列 SnapStart 事件傳送至 遙測 API：

- `platform.restoreStart` - 顯示 `Restore` 階段開始的時間。
- `platform.restoreRuntimeDone` - 顯示 `Restore` 階段是否成功。執行階段傳送 `restore/next` 執行階段 API 請求時，Lambda 會產生此訊息。可能的狀態有三種：成功、失敗和逾時。
- `platform.restoreReport` - 顯示 `Restore` 階段持續的時間長度，以及您須為此階段支付的費用。

Amazon API Gateway 和函數 URL 指標

如果您是使用 API Gateway 建立 Web API，便可以使用 IntegrationLatency 指標來測量端對端延遲 (API Gateway 將請求轉送至後端與收到後端回應之間的時間)。

如果您使用的是 Lambda 函數 URL，則可以使用 UrlRequestLatency 指標來測量端對端延遲 (函數 URL 收到請求與函數 URL 傳回回應之間的時間)。

Lambda SnapStart 的安全模型

Lambda SnapStart 支援靜態加密。Lambda 使用 AWS KMS key 來加密快照。預設情況下，Lambda 使用 AWS 受管金鑰。如果此預設行為符合您的工作流程，您便不需要設定其他項目。若不符合，您可以在 create-function 或 update-function-configuration 命令中使用 `--kms-key-arn` 選項來提供 AWS KMS 客戶管理的金鑰。這麼做可控制 KMS 金鑰的輪換或滿足貴組織對管理 KMS 金鑰的要求。客戶受管的金鑰會產生標準的 AWS KMS 費用。如需詳細資訊，請參閱 AWS Key Management Service 定價。

當您刪除 SnapStart 函數或函數版本時，對該函數或函數版本發出的所有 `Invoke` 要求都會失敗。根據一般資料保護規範 (GDPR)，Lambda 會移除與已刪除快照相關聯的所有資源。

最大化 Lambda SnapStart 效能

主題

- 效能調校
- 網路最佳實務

效能調校

若要充分利用 SnapStart 的優勢，請針對您的執行時期考慮下列程式碼最佳化建議。

> **Note**
>
> SnapStart 搭配大規模函數調用使用時效果最佳。不常調用的函數效能改進效果可能不會相同。

Java

為了充分利用 SnapStart 的優勢，建議您在初始化程式碼 (而不是函數處理常式) 中預先載入會導致啟動延遲的相依項和初始化資源。這會將與大量類別載入相關聯的延遲移出調用路徑，進而透過 SnapStart 最佳化啟動效能。

如果您無法在初始化期間預先載入相依項和初始化資源，建議您使用虛擬調用預先載入它們。若要執行此作業，請從 AWS Labs GitHub 儲存庫上的寵物存放區函數更新函數處理常式程式碼，如下列範例所示。

```
private static SpringLambdaContainerHandler<AwsProxyRequest, AwsProxyResponse> handler;
  static {
      try {
          handler =
 SpringLambdaContainerHandler.getAwsProxyHandler(PetStoreSpringAppConfig.class);

          // Use the onStartup method of the handler to register the custom filter
          handler.onStartup(servletContext -> {
              FilterRegistration.Dynamic registration =
 servletContext.addFilter("CognitoIdentityFilter", CognitoIdentityFilter.class);
              registration.addMappingForUrlPatterns(EnumSet.of(DispatcherType.REQUEST),
 false, "/*");
          });
```

```
            // Send a fake Amazon API Gateway request to the handler to load classes
    ahead of time
            ApiGatewayRequestIdentity identity = new ApiGatewayRequestIdentity();
            identity.setApiKey("foo");
            identity.setAccountId("foo");
            identity.setAccessKey("foo");

            AwsProxyRequestContext reqCtx = new AwsProxyRequestContext();
            reqCtx.setPath("/pets");
            reqCtx.setStage("default");
            reqCtx.setAuthorizer(null);
            reqCtx.setIdentity(identity);

            AwsProxyRequest req = new AwsProxyRequest();
            req.setHttpMethod("GET");
            req.setPath("/pets");
            req.setBody("");
            req.setRequestContext(reqCtx);

            Context ctx = new TestContext();
            handler.proxy(req, ctx);


        } catch (ContainerInitializationException e) {
            // if we fail here. We re-throw the exception to force another cold start
            e.printStackTrace();
            throw new RuntimeException("Could not initialize Spring framework", e);
        }
    }
```

Python

若要充分利用 SnapStart 的優勢，在編寫 Python 函數時應重視高效率程式碼組織和資源管理。一般而言，應在初始化階段執行繁重的運算任務。這種方法會將耗時的作業移出調用路徑，進而改善函數的整體效能。為協助您有效實作此策略，我們提供以下最佳實務建議：

- 在函數處理常式外部匯入相依項。
- 在處理常式外部建立 boto3 執行個體。
- 在調用處理常式之前初始化靜態資源或組態。
- 請考慮使用快照前執行時期勾點處理資源密集型任務，例如下載外部檔案、預先載入 Django 等架構，或載入機器學習模型。

Example：針對 SnapStart 最佳化 Python 函數

```
# Import all dependencies outside of Lambda handler
from snapshot_restore_py import register_before_snapshot
import boto3
import pandas
import pydantic

# Create S3 and SSM clients outside of Lambda handler
s3_client = boto3.client("s3")

# Register the function to be called before snapshot
@register_before_snapshot
def download_llm_models():
    # Download an object from S3 and save to tmp
    # This files will persist in this snapshot
    with open('/tmp/FILE_NAME', 'wb') as f:
        s3_client.download_fileobj('amzn-s3-demo-bucket', 'OBJECT_NAME', f)
    ...

def lambda_handler(event, context):
    ...
```

.NET

若要減少即時 (JIT) 編譯和組裝載入時間，請考慮從 `RegisterBeforeCheckpoint` 執行時期勾點調用函數處理常式。由於 .NET 分層編譯的運作方式，多次調用處理常式可獲得最佳結果，如以下範例所示。

> **Important**
>
> 請確定虛擬函數調用不會產生非預期的副作用，例如啟動商業交易。

Example

```
public class Function
{
    public Function()
    {
        Amazon.Lambda.Core.SnapshotRestore.RegisterBeforeSnapshot(FunctionWarmup);
```

```
    }

    // Warmup method that calls the function handler before snapshot to warm up
 the .NET code and runtime.
    // This speeds up future cold starts after restoring from a snapshot.

    private async ValueTask FunctionWarmup()
    {
        var request = new APIGatewayProxyRequest
        {
            Path = "/heathcheck",
            HttpMethod = "GET"
        };

        for (var i = 0; i < 10; i++)
        {
            await FunctionHandler(request, null);
        }
    }

    public async Task<APIGatewayProxyResponse> FunctionHandler(APIGatewayProxyRequest
 request, ILambdaContext context)
    {
        //
        // Process HTTP request
        //

        var response = new APIGatewayProxyResponse
        {
            StatusCode = 200
        };

        return await Task.FromResult(response);
    }
}
```

網路最佳實務

Lambda 從快照恢復函數時，無法保證函數在初始化階段建立的連線狀態。在大多數情況下， AWS 軟體開發套件建立的網路連線會自動恢復。針對其他連線，建議遵循最佳實務操作。

重新建立網路連線

函數從快照恢復時，請務必重新建立網路連線。建議您在函數處理常式中重新建立網路連線。您也可以使用還原後執行時期勾點。

請勿使用主機名做為唯一的執行環境識別符

建議您不要使用 hostname 將執行環境識別為應用程式中唯一的節點或容器。對於 SnapStart，會使用單一快照做為多個執行環境的初始狀態。所有執行環境對 InetAddress.getLocalHost() (Java)、socket.gethostname() (Python) 和 Dns.GetHostName() (.NET) 都會傳回相同的 hostname 值。如果應用程式需要唯一的執行環境識別或 hostname 值，建議您在函數處理常式中產生唯一的 ID。或者，使用還原後執行時期勾點來產生唯一的 ID，然後使用這個唯一 ID 做為執行環境的識別符。

避免將連線繫結至固定來源連接埠

建議您避免將網路連線繫結至固定來源連接埠。函數從快照恢復時連線會重新建立，而繫結至固定來源連接埠的網路連線可能會失敗。

避免使用 Java DNS 快取

Lambda 函數已快取 DNS 回應。如果您透過 SnapStart 使用其他 DNS 快取，則當函數從快照恢復，可能會發生連線逾時的情況。

java.util.logging.Logger 類別可以間接啟用 JVM DNS 快取。若要覆寫預設設定，請先將 networkaddress.cache.ttl 設定為 0，再初始化 logger。範例：

```
public class MyHandler {
  // first set TTL property
  static{
   java.security.Security.setProperty("networkaddress.cache.ttl" , "0");
  }
 // then instantiate logger
  var logger = org.apache.logging.log4j.LogManager.getLogger(MyHandler.class);
}
```

為防止 Java 11 執行時期出現 UnknownHostException 失敗，建議將 networkaddress.cache.negative.ttl 設定為 0。在 Java 17 和更新版本的執行時期中，則不需要此步驟。您可以使用 AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0 環境變數為 Lambda 函數設定此屬性。

停用 JVM DNS 快取並不會停用 Lambda 的受管 DNS 快取。

對 Lambda 函數的 SnapStart 錯誤進行故障診斷

此頁面說明如何解決使用 Lambda SnapStart 時發生的常見問題，包括快照建立錯誤、逾時錯誤和內部服務錯誤。

SnapStartNotReadyException

錯誤：呼叫 Invoke20150331 操作時發生錯誤 (SnapStartNotReadyException)：Lambda 正在初始化函數。一旦函數狀態變為「作用中」，即可進行調用。

常見原因

當嘗試調用處於`Inactive`狀態的函數版本時，便會發生此錯誤。函數版本若超過 14 天未被調用，或在 Lambda 定期回收執行環境時，該函數版本的狀態會變為 `Inactive`

Resolution

等待函數版本進入 `Active` 狀態，然後再次調用它。

SnapStartTimeoutException

問題：當嘗試調用 SnapStart 函數版本時遇到 `SnapStartTimeoutException` 錯誤。

常見原因

在還原階段，Lambda 會還原 Java 執行時期，並執行任何還原後執行時期勾點。如果還原後執行時期勾點執行超過 10 秒，且`Restore`階段逾時，則當您嘗試調用函數時便會出現錯誤。網路連線和憑證問題也可能導致 `Restore` 階段逾時。

Resolution

檢查函數的 CloudWatch 日誌，看看還原階段是否發生了逾時錯誤。確定所有還原後勾點在 10 秒內完成。

Example CloudWatch 日誌

```
{ "cause": "Lambda couldn't restore the snapshot within the timeout limit. (Service:
 Lambda, Status Code: 408, Request ID: 11a222c3-410f-427c-ab22-931d6bcbf4f2)", "error":
 "Lambda.SnapStartTimeoutException"}
```

500 內部服務錯誤

錯誤：由於已達到並行快照建立限制，Lambda 無法建立新的快照。

常見原因

500 錯誤是 Lambda 服務本身的內部錯誤，而不是函數或程式碼的問題。這些錯誤通常是間歇性的。

Resolution

嘗試再次發布函數版本。

401 (未經授權)

錯誤：錯誤的工作階段字符或標頭金鑰

常見原因

搭配 Lambda SnapStart 使用AWS Systems Manager 參數存放區和 AWS Secrets Manager 延伸時，會發生此錯誤。

Resolution

AWS Systems Manager 參數存放區和 AWS Secrets Manager 擴充功能與 SnapStart 不相容。延伸項目會產生登入資料，以便在函數初始化 AWS Secrets Manager 期間與 通訊，這會導致登入資料錯誤在與 SnapStart 搭配使用時過期。

UnknownHostException (Java)

錯誤：無法執行 HTTP 請求：`abc.us-east-1.amazonaws.com` 的憑證不符合任何主體別名。

常見原因

Lambda 函數已快取 DNS 回應。如果您透過 SnapStart 使用其他 DNS 快取，則當函數從快照恢復，可能會發生連線逾時的情況。

Resolution

為防止 Java 11 執行時期出現 `UnknownHostException` 失敗，建議將 `networkaddress.cache.negative.ttl` 設定為 0。在 Java 17 和更新版本的執行時期中，則不需

要此步驟。您可以使用 AWS_LAMBDA_JAVA_NETWORKADDRESS_CACHE_NEGATIVE_TTL=0 環境變數為 Lambda 函數設定此屬性。

快照建立失敗

Error： AWS Lambda 無法叫用您的 SnapStart 函數。如果此錯誤持續存在，請檢查函數的 CloudWatch 日誌，看看是否存在初始化錯誤。

Resolution

檢閱函數的 Amazon CloudWatch 日誌，了解檢查點前的執行時期勾點逾時。您也可以嘗試發布新的函數版本，這麼做有時可以解決問題。

快照建立延遲

問題：發布新的函數版本時，函數會長時間處於Pending狀態。

常見原因

Lambda 建立快照時，初始化程式碼最多可能會執行 15 分鐘。時間限制為 130 秒或設定的函數逾時 (最長 900 秒)，以較長者為準。

如果函數已連接至 VPC，Lambda 可能還需要在函數的狀態轉變成Active之前建立網路介面。如果您嘗試在函數處於Pending狀態時調用函數版本，則可能會遇到 409 ResourceConflictException 錯誤。如果使用 Amazon API Gateway 端點調用函數，則可能會在 API Gateway 中遇到 500 錯誤。

Resolution

請至少等待 15 分鐘，讓函數版本完成初始化，然後再調用它。

租戶隔離

當您需要個別最終使用者或呼叫 Lambda 函數的租用戶進行隔離請求處理時，請使用租用戶隔離模式。此功能可簡化建置多租用戶應用程式，以處理租用戶特定的程式碼或資料，例如工作流程自動化的 SaaS 平台或程式碼執行，無需管理租用戶特定的函數資源和請求路由邏輯。

執行程式碼或處理個別租用戶或最終使用者的資料時，多租用戶應用程式有嚴格的隔離要求。使用租用戶隔離模式時，Lambda 會使用客戶指定的租用戶識別符，將請求路由到基礎函數執行環境，確保函數的執行環境僅用於提供來自指定最終使用者或租用戶的調用。Lambda 的函數執行環境利用 Firecracker 虛擬化來提供工作負載隔離。

當使用租用戶隔離模式的函數收到具有租用戶識別符的調用時，Lambda 會先嘗試尋找與該租用戶識別符相關聯的可用執行環境。如果不存在執行環境，Lambda 會建立新的執行環境並指派給該租用戶。隨著具有指定租用戶識別符的函數調用擴展，Lambda 會視需要尋找或建立新的執行環境。

主題

- 何時使用租戶隔離模式
- 支援的功能和限制
- 支援的 AWS 區域
- 考量事項
- 定價
- 隔離模式
- 啟用 Lambda 函數的租戶隔離
- 使用租戶隔離調用 Lambda 函數
- 在 Lambda 函數程式碼中存取租戶識別符
- 使用租戶隔離監控 Lambda 函數
- 故障診斷 Lambda 函數的租戶隔離

何時使用租戶隔離模式

當您需要使用單一 Lambda 函數為多個最終使用者或租用戶提供服務時，請使用租用戶隔離模式，同時確保用於處理個別租用戶調用的執行環境彼此保持隔離。下列多租戶應用程式需要嚴格隔離執行環境：

- 執行最終使用者提供的程式碼：維護個別租用戶的個別執行環境，可能會限制執行使用者提供的錯誤或惡意程式碼的影響。
- 處理租用戶特定資料：維護個別租用戶的個別執行環境，可防止敏感租用戶特定資料暴露給其他租用戶。

來自相同租用戶的多個調用請求可以重複使用相同的函數執行環境，從而減少冷啟動並改善延遲敏感應用程式的回應時間。

支援的功能和限制

使用函數 URLs、佈建並行或 SnapStart 的函數不支援租用戶隔離模式。 ??? SnapStart 您可以使用同步調用、非同步調用，或使用 Amazon API Gateway 做為事件觸發器，將請求傳送至租戶隔離的函數。

支援的 AWS 區域

除了亞太區域（紐西蘭）之外，所有商業區域都支援租戶隔離模式。

考量事項

搭配 Lambda 函數使用租戶隔離時，請記住下列事項：

- 不可變組態：租戶隔離是不可變的函數屬性。它只能在建立函數時啟用。
- 必要的 tenant-id 參數：必須使用 `tenant-id` 參數調用使用租用戶隔離模式的函數。省略此參數會導致函數叫用失敗。
- 執行角色適用於所有租用戶：來自所有租用戶的調用會使用 Lambda 函數執行角色中定義的許可。
- 並行：使用租用戶隔離時，函數的並行或擴展行為沒有任何變更。Lambda 會針對為 Lambda 函數設定的每 1，000 個並行執行，限制 2，500 個租用戶隔離的執行環境（作用中或閒置）。

定價

當 Lambda 建立新的租戶隔離執行環境時，會向您收取費用。價格取決於您配置給函數的記憶體量，以及您使用的 CPU 架構。如需詳細資訊，請參閱 AWS Lambda 定價。

隔離模式

下表概述具有和不具有租用戶隔離的 Lambda 函數之間的差異。

功能	使用租戶隔離	沒有租用戶隔離
隔離類型	租戶層級隔離	函數層級隔離
環境重複使用	執行環境絕不會在不同的租用戶間重複使用	相同函數的叫用之間可能會重複使用執行環境
資料隔離	無法存取來自其他租戶的資料	可能可以存取先前調用相同函數版本的資料
冷啟動	由於租戶特定環境而有更多冷啟動	由於環境重複使用，冷啟動次數較少
定價	標準 Lambda 定價以外的額外費用	標準 Lambda 定價

啟用 Lambda 函數的租戶隔離

若要啟用租戶隔離模式，請建立新的 Lambda 函數。您無法在現有函數上啟用租用戶隔離。

主題

- 啟用租戶隔離 （ 主控台 ）
- 啟用租戶隔離 (AWS CLI)
- 啟用租戶隔離 (API)
- 啟用租戶隔離 (CloudFormation)

啟用租戶隔離 （ 主控台 ）

使用主控台建立 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇建立函數。
3. 選取從頭開始撰寫。
4. 在基本資訊窗格中，為函數名稱輸入 **image-analysis**。
5. 針對執行期，選擇任何支援的 Lambda 執行期。
6. 在其他組態下，租用戶隔離模式下，選取啟用。
7. 檢閱您的設定，然後選擇建立 函數。

啟用租戶隔離 (AWS CLI)

建立具有租用戶隔離的函數

使用 CLI 建立新函數時，請將 `--tenancy-config '{"TenantIsolationMode": "PER_TENANT"}'` 選項新增至您的 create-function 請求。範例：

```
aws lambda create-function \
    --function-name image-analysis \
    --runtime nodejs24.x \
    --zip-file fileb://image-analysis-function.zip \
    --handler image-analysis-function.handler \
    --role arn:aws:iam:123456789012:role/execution-role \
    --tenancy-config '{"TenantIsolationMode": "PER_TENANT"}'
```

啟用租戶隔離 (API)

使用 Lambda API 啟用租戶隔離

1. 使用 CreateFunction API 動作搭配 `TenancyConfig` 參數，在啟用租用戶隔離的情況下建立新的函數。
2. 使用 GetFunctionConfiguration 動作，確認已啟用函數的租用戶隔離。如果回應顯示 `TenantIsolationMode`為 `PER_TENANT`，則會為函數啟用租用戶隔離：

```
"TenancyConfig": {
        "TenantIsolationMode": "PER_TENANT"
    }
```

使用 Invoke 動作調用函數版本。如需詳細資訊，請參閱使用租戶隔離調用 Lambda 函數。

啟用租戶隔離 (CloudFormation)

下列 CloudFormation 範本會建立新的 Lambda 函數，並啟用租用戶隔離：

```
MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-sample-python-lambda
      Runtime: python3.14
      Role: !GetAtt LambdaExecutionRole.Arn
      Handler: index.lambda_handler
      TenancyConfig:
        TenantIsolationMode: PER_TENANT
      Code:
        ZipFile: |
          import json

          def lambda_handler(event, context):
              return {
                  'statusCode': 200,
                  'body': json.dumps(f'Hello from Lambda! Tenant-ID:
 {context.tenant_id}')
              }
      Timeout: 10
      MemorySize: 128
```

使用租戶隔離調用 Lambda 函數

叫用已啟用租用戶隔離的函數時，您必須提供 tenant-id 參數。此參數可確保您的函數叫用是在專用於該特定租用戶的執行環境中處理。

使用租用戶隔離調用函數 (AWS CLI)

同步調用

對於同步調用，請將 --tenant-id 參數新增至您的調用命令：

```
aws lambda invoke \
    --function-name image-analysis \
    --tenant-id blue \
    response.json
```

非同步叫用

對於非同步調用，請同時包含 --tenant-id和 --invocation-type Event 參數：

```
aws lambda invoke \
    --function-name image-analysis \
    --tenant-id blue \
    --invocation-type Event \
    response.json
```

使用租用戶隔離 (API) 叫用函數

直接使用叫用 API 動作時，請在請求中使用 X-Amzn-Tenant-Id 參數包含租戶識別符。

範例 API 請求

```
POST /2015-03-31/functions/image-analysis/invocations HTTP/1.1
Host: lambda.us-east-1.amazonaws.com
Content-Type: application/json
Authorization: AWS4-HMAC-SHA256 Credential=...
X-Amz-Tenant-Id: blue

{
    "key1": "value1",
    "key2": "value2"
```

```
}
```

使用租用戶隔離調用函數 (API Gateway)

使用 API Gateway REST APIs 觸發租戶隔離的 Lambda 函數時，您必須設定 API Gateway 將用戶端請求屬性映射至 Lambda 預期的`X-Amz-Tenant-Id`標頭。API Gateway 使用 Lambda 的調用 API 動作，這需要使用 `X-Amz-Tenant-Id` HTTP 標頭傳遞租用戶 ID。您可以設定 API Gateway，使用從 HTTP 標頭、查詢參數或路徑參數等用戶端請求屬性取得的值，將此 HTTP 標頭注入 Lambda 調用請求。您必須先映射用戶端請求屬性，才能覆寫 `X-Amz-Tenant-Id`標頭。

> **Note**
>
> 您無法使用 HTTP APIs 叫用租戶隔離的 Lambda 函數，因為無法覆寫 `X-Amz-Tenant-Id`標頭。

使用請求標頭

設定您的 API Gateway 整合，將自訂標頭從用戶端請求映射至 `X-Amz-Tenant-Id` 標頭。下列範例顯示具有 `x-tenant-id`標頭的用戶端請求：

```
POST /api/process HTTP/1.1
Host: your-api-id.execute-api.us-east-1.amazonaws.com
Content-Type: application/json
x-tenant-id: blue

{
    "data": "sample payload"
}
```

在您的 API Gateway 方法組態中，您必須：

1. 啟用用戶端請求標頭參數（例如 `method.request.header.x-tenant-id`)
2. 使用 將用戶端標頭映射至 Lambda 整合標頭 `integration.request.header.X-Amz-Tenant-Id`

使用查詢參數

同樣地，您可以將查詢參數映射至 `X-Amz-Tenant-Id`標頭：

```
GET /api/process?tenant-id=blue&data=sample HTTP/1.1
Host: your-api-id.execute-api.us-east-1.amazonaws.com
```

設定 方法以啟用查詢參數，並將其映射至整合標頭。

使用租用戶隔離調用函數 (SDK)

使用 AWS SDKs 調用租戶隔離的函數時，請在調用請求中包含租戶識別符。

Python

```
import boto3
import json

lambda_client = boto3.client('lambda')

response = lambda_client.invoke(
    FunctionName='image-analysis',
    TenantId='blue',
    Payload=json.dumps({
        'key1': 'value1',
        'key2': 'value2'
    })
)

result = json.loads(response['Payload'].read())
```

Node.js

```
const AWS = require('aws-sdk');
const lambda = new AWS.Lambda();

const params = {
    FunctionName: 'image-analysis',
    TenantId: 'blue',
    Payload: JSON.stringify({
        key1: 'value1',
        key2: 'value2'
    })
};

lambda.invoke(params, (err, data) => {
```

```
    if (err) {
        console.error(err);
    } else {
        const result = JSON.parse(data.Payload);
        console.log(result);
    }
});
```

Java

```
import software.amazon.awssdk.services.lambda.LambdaClient;
import software.amazon.awssdk.services.lambda.model.InvokeRequest;
import software.amazon.awssdk.services.lambda.model.InvokeResponse;
import software.amazon.awssdk.core.SdkBytes;

public class TenantIsolationExample {

    public static void main(String[] args) {
        LambdaClient lambdaClient = LambdaClient.create();

        String payload = "{\"key1\": \"value1\", \"key2\": \"value2\"}";

        InvokeRequest request = InvokeRequest.builder()
                .functionName("image-analysis")
                .tenantId("blue")
                .payload(SdkBytes.fromUtf8String(payload))
                .build();

        InvokeResponse response = lambdaClient.invoke(request);
    }
}
```

在 Lambda 函數程式碼中存取租戶識別符

當您的 Lambda 函數已啟用租用戶隔離時，用於調用函數的租用戶識別符會在傳遞給函數處理常式的內容物件中可用。您可以使用此識別符來實作租戶特定的邏輯、監控和偵錯功能。

主題

- 存取租戶識別符
- 常用模式
- 監控與除錯

存取租戶識別符

租用戶識別符可透過內容物件的 `tenantId` 屬性使用。請注意，此屬性在調用階段期間可用，而不是在初始化階段。

Python

```
def lambda_handler(event, context):
    tenant_id = context.tenant_id
    print(f"Processing request for tenant: {tenant_id}")

    # Implement tenant-specific logic
    if tenant_id == "blue":
        return process_blue_tenant(event)
    elif tenant_id == "green":
        return process_green_tenant(event)
    else:
        return process_default_tenant(event)
```

Node.js

```
exports.handler = async (event, context) => {
    const tenantId = context.tenantId;
    console.log(`Processing request for tenant: ${tenantId}`);

    // Implement tenant-specific logic
    switch (tenantId) {
        case 'blue':
            return processBlueTenant(event);
        case 'green':
```

```
            return processGreenTenant(event);
        default:
            return processDefaultTenant(event);
    }
};
```

Java

```
public class TenantHandler implements RequestHandler<Map<String, Object>, String> {

    @Override
    public String handleRequest(Map<String, Object> event, Context context) {
        String tenantId = context.getTenantId();
        System.out.println("Processing request for tenant: " + tenantId);

        // Implement tenant-specific logic
        switch (tenantId) {
            case "blue":
                return processBlueTenant(event);
            case "green":
                return processGreenTenant(event);
            default:
                return processDefaultTenant(event);
        }
    }
}
```

常用模式

以下是在函數程式碼中使用租戶識別符的常見方法：

租用戶特定組態

使用租戶 ID 載入租戶特定的組態或設定：

```
def lambda_handler(event, context):
    tenant_id = context.tenant_id

    # Load tenant-specific configuration
    config = load_tenant_config(tenant_id)
    database_url = config['database_url']
    api_key = config['api_key']
```

```
    # Process with tenant-specific settings
    return process_request(event, database_url, api_key)
```

租戶特定資料存取

使用租戶 ID 來確保資料隔離和存取控制：

```
import boto3

def lambda_handler(event, context):
    tenant_id = context.tenant_id

    # Ensure data access is scoped to the tenant
    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table('user_data')

    user_id = event.get('userId')

    response = table.get_item(
        Key={
            'tenant_id': tenant_id,
            'user_id': user_id
        }
    )

    return process_results(response.get('Item'), tenant_id)
```

監控與除錯

當您啟用 JSON 記錄時，租戶識別符會自動包含在 Lambda 日誌中，讓您更輕鬆地監控和偵錯租戶特定問題。您也可以使用租戶 ID 進行自訂指標和追蹤。

Example具有租戶 ID 的自訂指標

下列範例示範如何使用租用戶 ID 建立租用戶特定的 CloudWatch 指標，以監控租用戶的使用模式和效能：

```
import boto3

def lambda_handler(event, context):
    tenant_id = context.tenant_id
```

```
    cloudwatch = boto3.client('cloudwatch')

    # Record tenant-specific metrics
    cloudwatch.put_metric_data(
        Namespace='MyApp/TenantMetrics',
        MetricData=[
            {
                'MetricName': 'RequestCount',
                'Dimensions': [
                    {
                        'Name': 'TenantId',
                        'Value': tenant_id
                    }
                ],
                'Value': 1,
                'Unit': 'Count'
            }
        ]
    )

    return process_request(event, tenant_id)
```

使用租戶隔離監控 Lambda 函數

您可以使用 Amazon CloudWatch 監控租戶隔離的 Lambda 函數 AWS X-Ray，並使用遙測 API 存取延伸項目的即時遙測資料。

了解租戶隔離模式的記錄

對於使用租用戶隔離的函數，當您啟用 JSON 記錄時，Lambda 會自動在函數日誌中包含租用戶識別符，讓您更輕鬆地監控和偵錯租用戶特定的問題。Lambda 會為每個執行環境建立個別的 CloudWatch 日誌串流。您可以使用 CloudWatch Logs Insights 透過依租用戶識別符篩選來尋找屬於特定租用戶的日誌串流：

```
fields @logStream, @message
| filter tenantId=='BlueTenant' or record.tenantId=='BlueTenant'
| stats count() as logCount by @logStream
| sort @timestamp desc
```

您也可以使用此參數來擷取所有日誌串流中的租戶特定日誌：

```
fields @message
| filter tenantId=='BlueTenant' or record.tenantId=='BlueTenant'
| limit 1000
```

tenantId 屬性包含於平台事件（例如 platform.start和 platform.report），以及您在函數程式碼中列印的自訂日誌，如下所示：

```
{
    "time": "2025-10-13T19:48:06.990Z",
    "type": "platform.start",
    "record": {
        "requestId": "a0f40320-b43c-44b3-91bf-d5b5240a1bed",
        "functionArn": "arn:aws:lambda:us-east-1:xxxxxx:function:multitenant-
function-1",
        "version": "$LATEST",
        "tenantId": "BlueTenant""
    }
}
{
    "timestamp": "2025-10-13T19:48:06.992Z",
    "level": "INFO",
```

```
    "requestId": "a0f40320-b43c-44b3-91bf-d5b5240a1bed",
    "tenantId": "BlueTenant"",
    "message": "custom log line1"
}
{
    "timestamp": "2025-10-13T19:48:07.022Z",
    "level": "WARN",
    "requestId": "a0f40320-b43c-44b3-91bf-d5b5240a1bed",
    "tenantId": "BlueTenant"",
    "message": "custom log line2"
}
```

故障診斷 Lambda 函數的租戶隔離

此頁面說明針對 使用租用戶隔離時發生的常見問題 AWS Lambda。

InvalidParameterValueException

錯誤 ：未指定租用戶 ID 組態，或未啟用租用戶隔離時傳遞至函數。

常見原因

在沒有租用戶 ID 的情況下叫用租用戶隔離的函數，或使用租用戶 ID 叫用non-tenant-isolated函數時，會發生此錯誤。

Resolution

如果函數已啟用租用戶隔離，請新增租用戶 ID，如果函數未啟用租用戶隔離，請移除租用戶 ID。

TooManyRequestsException

錯誤：超出速率

常見原因

除了根據最大並行執行和函數擴展速率的速率限制之外，Lambda 還會針對函數的每 1，000 個並行執行，將一次可存在的租用戶感知執行環境 （作用中或閒置） 數量上限限制為 2，500。

Resolution

若要修正此問題，您可以降低使用唯一租用戶識別符發出調用請求的速率、使用退避和抖動實作重試，或請求增加函數並行限制。

使用來自其他服務的事件叫用 Lambda AWS

有些 AWS 服務 可以使用觸發直接叫用 Lambda 函數。這些服務會將事件推送至 Lambda，並在發生指定事件時立即調用函數。觸發條件適用於離散事件和即時處理。當您使用 Lambda 主控台建立觸發時，主控台會與對應的 AWS 服務互動，以在該服務上設定事件通知。觸發條件實際上由產生事件的服務 (而非 Lambda) 儲存和管理。

事件是以 JSON 格式構建的資料。JSON 結構因產生它的服務和事件類型而異，但它們都包含函數處理事件所需的資料。

函數可以有多個觸發條件。每個觸發條件皆做為獨立調用函數的用戶端，而 Lambda 傳遞給函數的每個事件只會包含來自一個觸發條件的資料。Lambda 將事件文件轉換為物件並將其傳遞給您的函數處理常式。

視服務而定，事件驅動型調用可以是同步或非同步。

- 對於同步調用，產生事件的服務會等待來自您的函數回應。該服務定義函數需要在回應中傳回的資料。服務控制項錯誤策略，例如發生錯誤時是否重試。
- 對於非同步調用，Lambda 會先將事件排入佇列，再將事件傳送至您的函數。當 Lambda 將事件排入佇列時，它會立即向產生事件的服務傳送成功回應。在函數處理事件後，Lambda 不會向事件產生服務傳回回應。

建立觸發條件

建立觸發條件的最簡單方式是使用 Lambda 主控台。當您使用主控台建立觸發條件時，Lambda 會自動將必要的許可新增至函數的資源型政策。

若要使用 Lambda 主控台建立觸發條件

1. 開啟 Lambda 主控台中的函數頁面。
2. 選取您要為其建立觸發條件的函數。
3. 在函數概觀窗格中，選擇新增觸發條件。
4. 選取您要叫用函數 AWS 的服務。
5. 填寫觸發條件組態窗格中的選項，然後選擇新增。根據您 AWS 服務 選擇叫用函數的 ，觸發組態選項會有所不同。

可調用 Lambda 函數的服務

下表列出可調用 Lambda 函數的服務。

服務	調用的方法
Amazon Managed Streaming for Apache Kafka	事件來源映射
自我管理的 Apache Kafka	事件來源映射
Amazon API Gateway	事件驅動；同步調用
AWS CloudFormation	事件驅動；非同步叫用
Amazon CloudWatch Logs	事件驅動；非同步調用
AWS CodeCommit	事件驅動；非同步調用
AWS CodePipeline	事件驅動；非同步調用
Amazon Cognito	事件驅動；同步調用
AWS Config	事件驅動；非同步調用
Amazon Connect	事件驅動；同步調用
Amazon DocumentDB	事件來源映射
Amazon DynamoDB	事件來源映射
Elastic Load Balancing (Application Load Balancer)	事件驅動；同步調用
Amazon EventBridge (CloudWatch Events)	事件驅動型；非同步調用 (事件匯流排)、同步或非同步調用 (管道和排程)
AWS IoT	事件驅動；非同步調用
Amazon Kinesis	事件來源映射

服務	調用的方法
Amazon Data Firehose	事件驅動；同步調用
Amazon Lex	事件驅動；同步調用
Amazon MQ	事件來源映射
Amazon Simple Email Service	事件驅動；非同步調用
Amazon Simple Notification Service	事件驅動；非同步調用
Amazon Simple Queue Service	事件來源映射
Amazon Simple Storage Service (Amazon S3)	事件驅動；非同步調用
Amazon Simple Storage Service 批次	事件驅動；同步調用
Secrets Manager	秘密輪換
AWS Step Functions	事件驅動型；同步或非同步調用
Amazon VPC Lattice	事件驅動；同步調用

將 Lambda 與 Apache Kafka 搭配使用

Lambda 支援 Apache Kafka 作為事件來源。Apache Kafka 是一種開放原始碼事件串流平台，旨在處理高輸送量、即時資料管道和串流應用程式。可透過以下兩種主要方式將 Lambda 與 Apache Kafka 搭配使用：

- the section called “MSK” – Amazon Managed Streaming for Apache Kafka (Amazon MSK) 是一種完全受管的服務 AWS。Amazon MSK 可協助自動管理 Kafka 基礎結構，包括佈建、修補和擴展。
- the section called “自我管理的 Apache Kafka” – 在 AWS 術語中，自我管理叢集包含非AWS 託管的 Kafka 叢集。例如，您仍然可以將 Lambda 與託管於非AWS 雲端提供者的 Kafka 叢集搭配使用，例如 Confluent Cloud 或 Redpanda。

在決定使用 Amazon MSK 還是自我管理的 Apache Kafka 時，建議考量操作需求和控制要求。如果您想要 AWS 快速協助您以最少的營運開銷管理可擴展、生產就緒的 Kafka 設定，Amazon MSK 是更好的選擇。該服務能簡化安全性設定、監控機制與高可用性部署，讓您專注於應用程式開發，無需費心底層基礎結構管理。另一方面，自我管理的 Apache Kafka 更適合在非AWS 託管環境中執行的使用案例，包括內部部署叢集。

主題

- 搭配使用 Lambda 與 Amazon MSK
- 搭配使用 Lambda 與自我管理 Apache Kafka
- Lambda 中的 Apache Kafka 事件輪詢器擴展模式
- Lambda 中的 Apache Kafka 輪詢與串流開始位置
- Lambda 中可自訂的取用者群組 ID
- 從 Amazon MSK 和自我管理的 Apache Kafka 事件來源篩選事件
- 在 Lambda 中將結構描述登錄檔與 Kafka 事件來源搭配使用
- Kafka 事件來源的低延遲處理
- 設定 Kafka 事件來源的錯誤處理控制項
- 擷取 Amazon MSK 和自我管理的 Apache Kafka 事件來源的捨棄批次
- 使用 Kafka 主題做為失敗時的目的地
- Kafka 事件來源映射記錄
- 對 Kafka 事件來源映射錯誤進行疑難排解

搭配使用 Lambda 與 Amazon MSK

Amazon Managed Streaming for Apache Kafka (Amazon MSK) 是一項全受管服務，可讓您建立並執行使用 Apache Kafka 處理串流資料的應用程式。Amazon MSK 簡化了 Kafka 叢集的設定、擴展與管理流程。Amazon MSK 也可讓您更輕鬆地為多個可用區域設定應用程式，並使用 AWS Identity and Access Management (IAM) 確保安全。

本章說明了如何將 Amazon MSK 叢集用作 Lambda 函式的事件來源。將 Amazon MSK 與 Lambda 整合的一般流程包含以下步驟：

1. 叢集與網路設定：首先，設定 Amazon MSK 叢集。這包括設定允許 Lambda 存取叢集所需的正確聯網組態。
2. 事件來源映射設定：接著，建立 Lambda 所需的事件來源映射資源，將 Amazon MSK 叢集安全地連線至函式。
3. 函式與許可設定：最後，確保函式已正確設定，並且其執行角色具有必要許可。

> **Note**
>
> 您現在可以直接從 Lambda 或 Amazon MSK 主控台建立和管理 Amazon MSK 事件來源映射。兩個主控台都提供選項，可自動處理設定必要的 Lambda 執行角色許可，以便更簡化的組態程序。

如需如何設定 Lambda 與 Amazon MSK 叢集整合的範例，請參閱 AWS 《運算部落格》中的the section called “教學課程”使用 Amazon MSK 做為 的事件來源 AWS Lambda，以及《Amazon MSK 實驗室》中的 Amazon MSK Lambda 整合。

主題

- 範例事件
- 為 Lambda 設定 Amazon MSK 叢集與 Amazon VPC 網路
- 設定 Amazon MSK 事件來源映射的 Lambda 許可
- 設定 Lambda 的 Amazon MSK 事件來源
- 教學課程：使用 Amazon MSK 事件來源映射來調用 Lambda 函數

範例事件

Lambda 會在調用函數時，在事件參數中傳送訊息批次。事件酬載包含訊息陣列。陣列中的每個項目包含 Amazon MSK 主題和分割區識別符的詳細資訊，以及時間戳記和 base64 編碼的訊息。


```
{
   "eventSource":"aws:kafka",
   "eventSourceArn":"arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
                  ]
               }
            ]
         }
      ]
   }
}
```

為 Lambda 設定 Amazon MSK 叢集與 Amazon VPC 網路

若要將 AWS Lambda 函數連線至 Amazon MSK 叢集，您需要正確設定叢集及其所在的 Amazon Virtual Private Cloud (VPC)。本頁介紹了如何設定叢集與 VPC。如果叢集與 VPC 已正確設定，請參閱the section called “設定事件來源”，設定事件來源映射。

主題

- Lambda 與 MSK 整合的網路組態需求概觀
- 為 MSK 事件來源設定 NAT 閘道
- 設定 MSK 事件來源的 AWS PrivateLink 端點

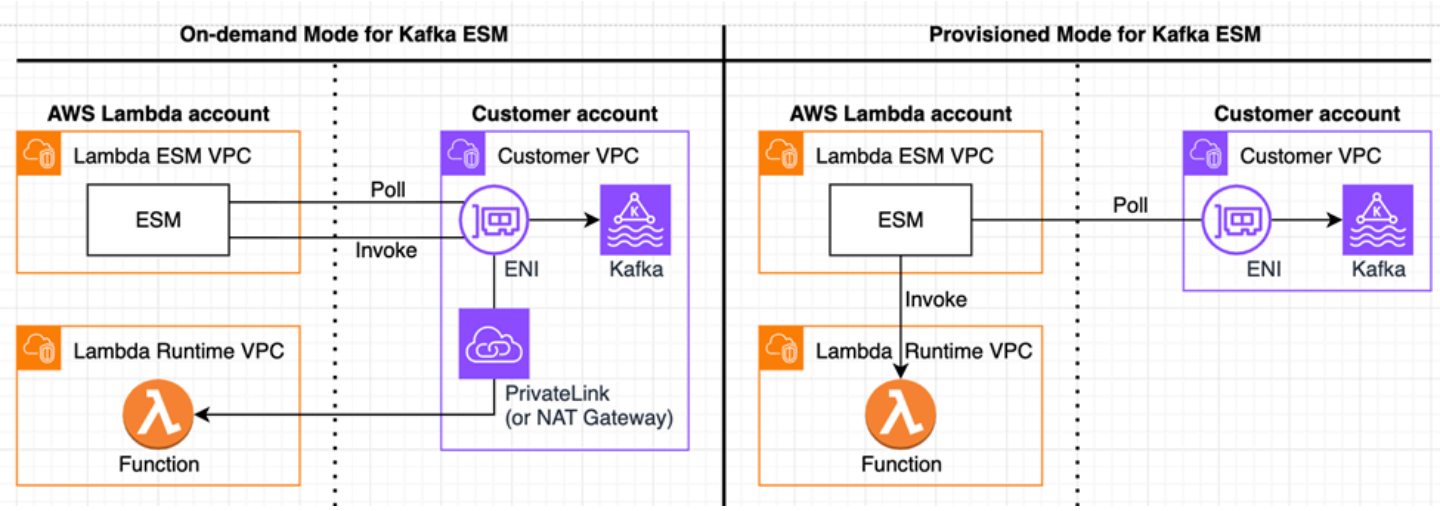
Lambda 與 MSK 整合的網路組態需求概觀

Lambda 與 MSK 整合所需的聯網組態取決於應用程式的網路架構。此整合涉及三種主要資源：Amazon MSK 叢集、Lambda 函式、Lambda 事件來源映射。這些資源各自位於不同的 VPC 內：

- Amazon MSK 叢集通常位於您管理的 VPC 的私有子網路內。
- 您的 Lambda 函數位於 Lambda AWS擁有的受管 VPC 中。
- 您的 Lambda 事件來源映射位於 Lambda AWS擁有的另一個受管 VPC 中，與包含函數的 VPC 分開。

事件來源映射是 MSK 叢集與 Lambda 函式之間的媒介資源。事件來源映射有兩個主要任務。首先，其會輪詢 MSK 叢集來取得新訊息。然後，其會使用這些訊息來調用 Lambda 函式。由於這三種資源位於不同的 VPC 內，輪詢與調用操作皆須跨 VPC 網路進行呼叫。

事件來源映射所需的網路組態取決於其是使用佈建模式還是隨需模式，如下圖所示：



在這兩種模式下，Lambda 事件來源映射輪詢 MSK 叢集取得新訊息的方式一致。為了在事件來源映射與 MSK 叢集之間建立連線，Lambda 會在私有子網路內建立 Hyperplane ENI (或在可用時重複使用現有介面)，以此建立安全連線。如圖所示，此 Hyperplane ENI 會使用 MSK 叢集的子網路和安全群組組態，而不是 Lambda 函式。

從叢集輪詢訊息之後，Lambda 調用函式的方式在每種模式下有所不同：

- 在佈建模式下，Lambda 會自動處理事件來源映射 VPC 與函式 VPC 之間的連線。因此，無需任何其他聯網元件，即能成功調用函式。
- 在隨需模式下，Lambda 事件來源映射會透過客戶自管 VPC 的路徑來調用函式。因此，您需要在 VPC 的公有子網路內設定 NAT 閘道，或者在 VPC 的私有子網路內設定 AWS PrivateLink 端點，這些端點需提供對 Lambda、AWS Security Token Service (STS) 的存取權，並可選擇性提供對 AWS Secrets Manager 的存取權。正確設定其中一個選項，即可在 VPC 與 Lambda 受管執行時期 VPC 之間建立連線，這是調用函式的必要條件。

NAT 閘道允許私有子網路內的資源存取公有網際網路。使用此組態表示您的流量在叫用 Lambda 函數之前周遊網際網路。 AWS PrivateLink 端點可讓私有子網路安全地連線至 AWS 服務或其他私有 VPC 資源，而不會周遊公有網際網路。有關如何設定這些資源的詳細資訊，請參閱the section called “為 MSK 事件來源設定 NAT 閘道”或the section called “設定 MSK 事件來源的 AWS PrivateLink 端點”。

截至目前，我們皆假設 MSK 叢集位於 VPC 的私有子網路內，這也是常見情況。不過，即使 MSK 叢集位於 VPC 的公有子網路內，您仍需設定 AWS PrivateLink 端點，才能啟用安全連線。下表根據設定 MSK 叢集和 Lambda 事件來源映射的方式，摘要說明了聯網組態要求：

MSK 叢集位置 (位於客戶自管 VPC 內)	Lambda 事件來源映射擴展模式	必要的聯網組態
私有子網路	隨需模式	NAT 閘道（位於 VPC 的公有子網路中）或 AWS PrivateLink 端點（位於 VPC 的私有子網路中），以啟用對 Lambda 的存取 AWS STS，以及選擇性的 Secrets Manager。
公有子網路	隨需模式	AWS PrivateLink 端點（在 VPC 的公有子網路中），以啟用對 Lambda 的存取 AWS

MSK 叢集位置 (位於客戶自管 VPC 內)	Lambda 事件來源映射擴展模式	必要的聯網組態
		STS，以及選用的 Secrets Manager。
私有子網路	佈建模式	無
公有子網路	佈建模式	無

此外，與 MSK 叢集相關聯的安全群組必須允許正確連接埠的流量。確保您已設定下列安全群組規則：

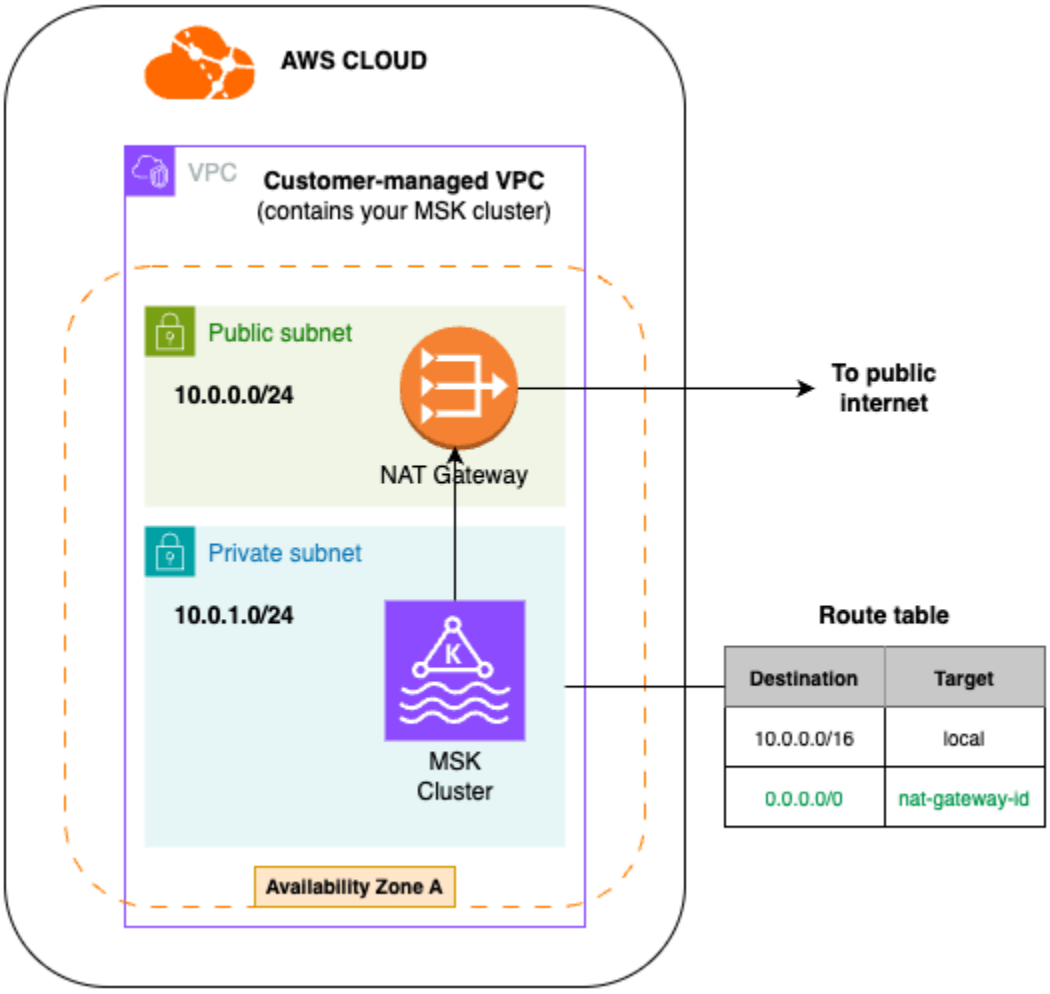
- 傳入規則：允許預設代理程式連接埠上的所有流量。MSK 使用的連接埠取決於叢集上的身分驗證類型：9098 用於 IAM 身分驗證、9096 用於 SASL/SCRAM，9094 用於 TLS。或者，您可使用自我參照安全群組規則，允許來自同一安全群組內其他執行個體的存取。
- 傳出規則 – 如果您的函數需要與其他 443 AWS 服務通訊，允許外部目的地連接埠上的所有流量。或者，如果您不需要與其他 AWS 服務通訊，您可以使用自我參考安全群組規則來限制對代理程式的存取。
- Amazon VPC 端點傳入規則：若您使用 Amazon VPC 端點，與該端點相關聯的安全群組必須允許來自叢集安全群組 443 連接埠的傳入流量。

為 MSK 事件來源設定 NAT 閘道

您可以設定 NAT 閘道，允許事件來源映射從叢集輪詢訊息，並透過 VPC 的路徑調用函式。僅在事件來源映射使用隨需模式，且叢集位於 VPC 的私有子網路內時才需要此操作。如果叢集位於 VPC 的公有子網路內，或者事件來源映射使用佈建模式，則無需設定 NAT 閘道。

NAT 閘道允許私有子網路內的資源存取公有網際網路。如需與 Lambda 建立私有連線，請改為參閱the section called “設定 MSK 事件來源的 AWS PrivateLink 端點”。

設定 NAT 閘道之後，必須設定正確的路由表。這將允許來自私有子網路的流量透過 NAT 閘道路由至公有網際網路。



下列步驟將引導您使用主控台設定 NAT 閘道。視需要針對每個可用區域 (AZ) 重複這些步驟。

設定 NAT 閘道和正確路由 (主控台)

1. 請遵循建立 NAT 閘道中的步驟，同時注意下列事項：
   - NAT 閘道應始終位於公有子網路內。建立具有公有連線的 NAT 閘道。
   - 如果 MSK 叢集跨多個可用區域複寫，請為每個可用區域建立一個 NAT 閘道。例如，在每個可用區域中，VPC 應該有一個包含叢集的私有子網路，以及一個包含 NAT 閘道的公有子網路。對於具有三個可用區域的設定，您將擁有三個私有子網路、三個公有子網路及三個 NAT 閘道。
2. 建立 NAT 閘道之後，開啟 Amazon VPC 主控台，然後在左側功能表中選擇路由表。
3. 選擇 Create route table (建立路由表)。
4. 將此路由表與包含 MSK 叢集的 VPC 建立關聯。或者，輸入路由表的名稱。
5. 選擇 Create route table (建立路由表)。

6. 選擇剛剛建立的路由表。
7. 在子網路關聯索引標籤中，選擇編輯子網路關聯。
   - 將此路由表與包含 MSK 叢集的私有子網路建立關聯。
8. 選擇 Edit routes (編輯路由)。
9. 選擇新增路由：
   1. 對於 Destination (目的地)，請選擇 `0.0.0.0/0`。
   2. 在目標欄位中，選擇 NAT 閘道。
   3. 在搜尋方塊中，選擇在步驟 1 中建立的 NAT 閘道。這應該是與包含 MSK 叢集的私有子網路 (您在步驟 6 中與此路由表相關聯的私有子網路) 位於同一可用區域中的 NAT 閘道。
10. 選擇儲存變更。

設定 MSK 事件來源的 AWS PrivateLink 端點

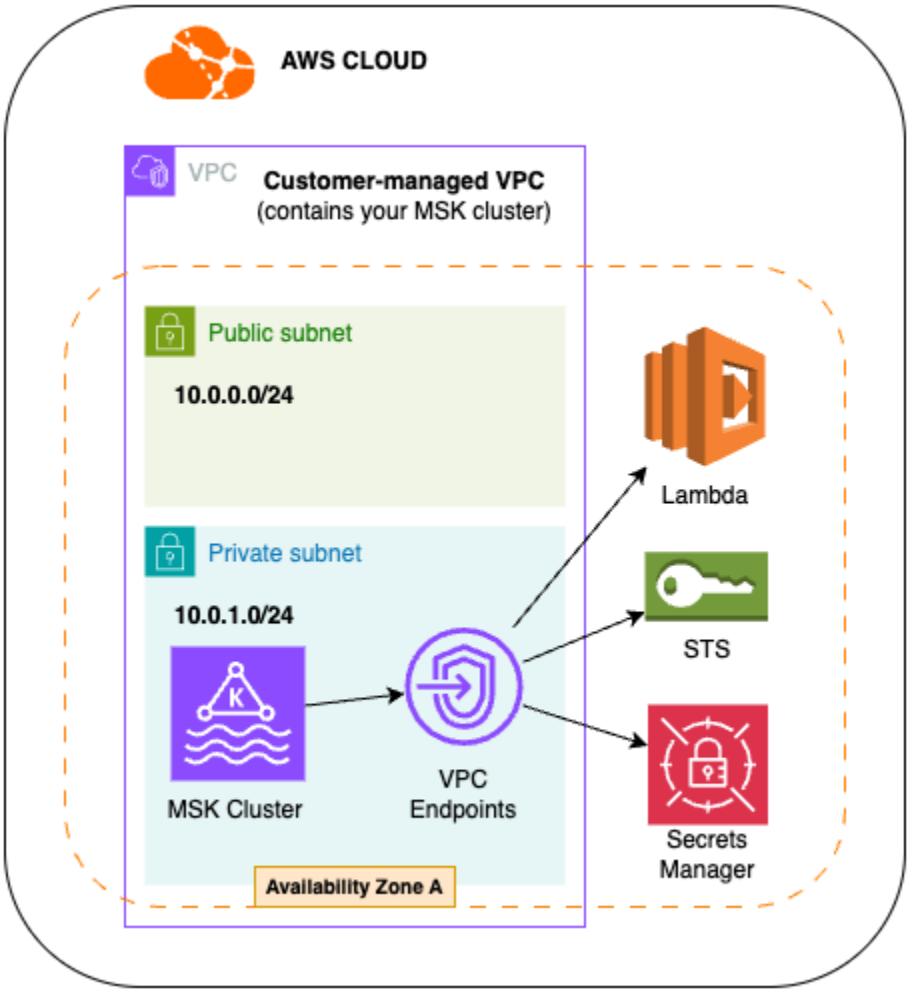
您可以將 AWS PrivateLink 端點設定為從叢集輪詢訊息，並透過 VPC 透過路徑叫用函數。這些端點應允許 MSK 叢集存取下列服務：

- Lambda 服務
- AWS Security Token Service (STS)
- (選用) AWS Secrets Manager 服務。若叢集身分驗證所需的秘密儲存在 Secrets Manager 中，則此為必要項目。

僅在事件來源映射使用隨需模式時，才需要設定 PrivateLink 端點。如果事件來源映射使用佈建模式，Lambda 會代為建立必要連線。

PrivateLink 端點允許安全的私有存取 AWS 服務 AWS PrivateLink。或者，若要設定 NAT 閘道以讓 MSK 叢集存取公有網際網路，請參閱the section called “為 MSK 事件來源設定 NAT 閘道”。

設定 VPC 端點之後，MSK 叢集應可直接且透過私有方式存取 Lambda、STS，以及選擇性地存取 Secrets Manager。



下列步驟將引導您使用主控台設定 PrivateLink 端點。視需要針對每個端點 (Lambda、STS、Secrets Manager) 重複這些步驟。

設定 VPC PrivateLink 端點 (主控台)

1. 開啟 Amazon VPC 主控台，然後在左側功能表中選擇端點。
2. 選擇建立端點。
3. 或者，輸入端點的名稱。
4. 在類型欄位中，選擇 AWS 服務。
5. 在服務欄位中，開始輸入服務的名稱。例如，若要建立連線至 Lambda 的端點，請在搜尋方塊中輸入 `lambda`。
6. 在結果中，您應能看到目前區域的服務端點。例如，在美國東部 (維吉尼亞北部) 區域，您應該會看到 `com.amazonaws.us-east-2.lambda`。選取該服務。
7. 在網路設定欄位中，選取包含 MSK 叢集的 VPC。

8. 在子網路欄位中，選取 MSK 叢集所在的可用區域。
   - 對於每個可用區域，在子網路 ID 欄位中，選擇包含 MSK 叢集的私有子網路。
9. 在安全群組欄位中，選取與 MSK 叢集相關聯的安全群組。
10. 選擇建立端點。

根據預設，Amazon VPC 端點具有開放的 IAM 政策，允許廣泛存取資源。最佳實務是限制這些政策，以使用該端點執行所需的動作。例如，對於 Secrets Manager 端點，您可以修改其政策，使其僅允許函式的執行角色存取秘密。

Example VPC 端點政策：Secrets Manager 端點

```
{
    "Statement": [
        {
            "Action": "secretsmanager:GetSecretValue",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws::iam::123456789012:role/my-role"
                ]
            },
            "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
        }
    ]
}
```

對於 AWS STS 和 Lambda 端點，您可以將呼叫委託人限制為 Lambda 服務委託人。但是，請確定在這些政策使用 `"Resource": "*"`。

Example VPC 端點政策 – AWS STS 端點

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
```

```
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example VPC 端點政策：Lambda 端點

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

設定 Amazon MSK 事件來源映射的 Lambda 許可

若要存取 Amazon MSK 叢集，函式和事件來源映射需要執行各種 Amazon MSK API 動作的許可。將這些許可新增至函式的執行角色。如果使用者需要存取權，請將必要許可新增至使用者或角色的身分政策。

AWSLambdaMSKExecutionRole 受管政策包含 Amazon MSK Lambda 事件來源映射所需的最低許可。若要簡化許可程序，您可以：

- 將 AWSLambdaMSKExecutionRole 受管政策連接至您的執行角色。
- 讓 Lambda 主控台為您產生許可。當您在主控台中建立 Amazon MSK 事件來源映射時，Lambda 會評估您的執行角色，並在缺少任何許可時提醒您。選擇產生許可以自動更新您的執行角色。如果您手動建立或修改執行角色政策，或者政策連接到多個角色，則此功能不適用。請注意，使用失敗時目的地或AWS Glue 結構描述登錄檔等進階功能時，執行角色可能仍需要額外的許可。

主題

- 所需的許可
- 可選的許可。

所需的許可

您的 Lambda 函數執行角色必須具有下列 Amazon MSK 事件來源映射的必要許可。這些許可包含在 AWSLambdaMSKExecutionRole 受管政策中。

CloudWatch Logs 許可

下列許可允許 Lambda 在 Amazon CloudWatch Logs 中建立和存放日誌。

- logs:CreateLogGroup
- logs:CreateLogStream
- 日誌：PutLogEvents

MSK 叢集許可

下列許可允許 Lambda 代表您存取您的 Amazon MSK 叢集：

- kafka:DescribeCluster
- kafka:DescribeClusterV2
- kafka:GetBootstrapBrokers

我們建議您使用 kafka：DescribeClusterV2，而非 kafka：DescribeCluster。v2 許可適用於佈建和無伺服器 Amazon MSK 叢集。您只需要政策中的其中一個許可。

VPC 許可

下列許可允許 Lambda 在連線至 Amazon MSK 叢集時建立和管理網路介面：

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets

- ec2:DescribeSecurityGroups

可選的許可。

您的 Lambda 函數可能需要許可，才能：

- 存取跨帳戶 Amazon MSK 叢集。對於跨帳戶事件來源映射，您需要執行角色中的 kafka：DescribeVpcConnection。建立跨帳戶事件來源映射的 IAM 主體需要 kafka：ListVpcConnections。
- 存取 SCRAM 秘密 (若使用 SASL/SCRAM 身分驗證)。此舉可讓函式透過使用者名稱與密碼連線至 Kafka。
- 描述 Secrets Manager 秘密 (若使用 SASL/SCRAM 或 mTLS 身分驗證)。此舉可讓函式擷取安全連線所需的憑證或認證。
- 如果您的 AWS Secrets Manager 秘密是使用 AWS KMS 客戶受管金鑰加密，請存取您的 AWS KMS 客戶受管金鑰。
- 存取結構描述登錄檔秘密 (若使用的結構描述登錄檔需進行身分驗證)：
  - 對於 AWS Glue 結構描述登錄檔：您的函數需求`glue:GetRegistry`和`glue:GetSchemaVersion`許可。這些許可能夠讓函式查詢並使用儲存在 AWS Glue中的訊息格式規則。
  - 對於使用 `BASIC_AUTH` 或 `CLIENT_CERTIFICATE_TLS_AUTH` 的 Confluent 結構描述登錄檔：函式需要對包含身分驗證憑證的秘密具有 `secretsmanager:GetSecretValue` 許可。這可讓函式擷取存取 Confluent 結構描述登錄檔所需的使用者名稱/密碼或憑證。
  - 對於私有 CA 憑證：函式需要對包含憑證的秘密具有 secretsmanager:GetSecretValue 許可。此舉可讓函式驗證使用自訂憑證的結構描述登錄檔身分。
- 如果您使用 IAM 身分驗證進行事件來源映射，請從主題存取 Kafka 叢集取用者群組和輪詢訊息。

這些對應至下列必要許可：

- kafka:ListScramSecrets – 允許列出用於 Kafka 身分驗證的 SCRAM 秘密
- secretsmanager:GetSecretValue – 允許從 Secrets Manager 擷取秘密
- kms:Decrypt - 允許使用 解密加密的資料 AWS KMS
- glue：GetRegistry - 允許存取 AWS Glue 結構描述登錄檔
- glue：GetSchemaVersion - 啟用從結構描述登錄檔擷取特定 AWS Glue 結構描述版本
- kafka-cluster：Connect - 准許連線至叢集並進行驗證
- kafka-cluster：AlterGroup - 准許加入叢集上的群組，相當於 Apache Kafka 的 READ GROUP ACL

- kafka-cluster：DescribeGroup - 准許描述叢集上的群組，相當於 Apache Kafka 的 DESCRIBE GROUP ACL
- kafka-cluster：DescribeTopic - 准許描述叢集上的主題，相當於 Apache Kafka 的 DESCRIBE TOPIC ACL
- kafka-cluster：ReadData - 准許從叢集上的主題讀取資料，相當於 Apache Kafka 的 READ TOPIC ACL

此外，若想將失敗調用的記錄傳送至失敗時目的地，根據目的地類型，您將需要下列許可：

- 對於 Amazon SQS 目的地：sqs:SendMessage – 允許將訊息傳送至 Amazon SQS 佇列
- 對於 Amazon SNS 目的地：sns:Publish – 允許將訊息發布至 Amazon SNS 主題
- 對於 Amazon S3 儲存貯體目的地：s3:PutObject 與 s3：ListBucket – 允許在 Amazon S3 儲存貯體中寫入及列出物件

如需疑難排解身分驗證與授權錯誤的相關資訊，請參閱the section called “疑難排解”。

設定 Lambda 的 Amazon MSK 事件來源

若要將 Amazon MSK 叢集用作 Lambda 函式的事件來源，需要建立將這兩項資源連線的事件來源映射。本頁介紹了如何為 Amazon MSK 建立事件來源映射。

本頁假設您已正確設定 MSK 叢集及其所在的 Amazon Virtual Private Cloud (VPC)。如需設定叢集或 VPC，請參閱the section called “叢集與網路設定”。若要設定錯誤處理的重試行為，請參閱 the section called “重試組態”。

主題

- 使用 Amazon MSK 叢集作為事件來源
- 在 Lambda 中設定 Amazon MSK 叢集身分驗證方法
- 為 Amazon MSK 事件來源建立 Lambda 事件來源映射
- 在 Lambda 中建立跨帳戶事件來源映射
- Lambda 中的所有 Amazon MSK 事件來源組態參數

使用 Amazon MSK 叢集作為事件來源

當您將 Apache Kafka 或 Amazon MSK 叢集新增為 Lambda 函數的觸發條件時，該叢集會用作事件來源。

Lambda 會根據您指定的起始位置，從 Kafka 主題 (您在 CreateEventSourceMapping 請求中指定為 `Topics`) 讀取事件資料。處理成功後，您的 Kafka 主題將遞交給 Kafka 叢集。

Lambda 會依序讀取每個 Kafka 主題分割區的訊息。單一 Lambda 承載可以包含來自多個分割區的訊息。有更多記錄可用時，Lambda 會根據您在 CreateEventSourceMapping 請求中指定的 BatchSize 值，繼續以批次方式來處理記錄，直到函式追上主題的進度為止。

Lambda 處理每個批次後，會遞交該批次中訊息的偏移量。如果函數針對批次中的任何訊息傳回錯誤，Lambda 會重試整個批次的訊息，直至處理成功或訊息過期。您可以將所有重試嘗試失敗時的記錄傳送至失敗時的目的地，以便稍後處理。

> **Note**
>
> 雖然 Lambda 函數的逾時上限通常為 15 分鐘，但 Amazon MSK、自我管理的 Apache Kafka、Amazon DocumentDB 以及 Amazon MQ for ActiveMQ 和 Amazon MQ for RabbitMQ 的事件來源映射只支援 14 分鐘逾時限制上限的函數。

在 Lambda 中設定 Amazon MSK 叢集身分驗證方法

Lambda 需要許可才能存取 Amazon MSK 叢集、擷取記錄和執行其他任務。Amazon MSK 支援多種方法向 MSK 叢集進行身分驗證。

叢集身分驗證方法

- 未驗證的存取
- SASL/SCRAM 身分驗證
- 交互 TLS 驗證
- IAM 身分驗證
- Lambda 選擇引導代理程式的方法

未驗證的存取

如果沒有用戶端透過網際網路存取叢集，您可以使用未驗證存取。

SASL/SCRAM 身分驗證

Lambda 支援 Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) 身分驗證，並使用 SHA-512 雜湊函式和 Transport Layer Security (TLS)

加密。若要讓 Lambda 連線至叢集，請將身分驗證憑證 (使用者名稱與密碼) 儲存在 Secrets Manager 秘密中，並在設定事件來源映射時參考此秘密。

如需有關使用 Secrets Manager 的詳細資訊，請參閱 Amazon Managed Streaming for Apache Kafka Developer Guide 中的 Sign-in credentials authentication with Secrets Manager。

> **Note**
>
> Amazon MSK 不支援 SASL/PLAIN 身分驗證。

交互 TLS 驗證

交互 TLS (mTLS) 可提供用戶端與伺服器之間的雙向身分驗證。用戶端會將憑證傳送至伺服器，供伺服器驗證用戶端。伺服器也會將憑證傳送至用戶端，供用戶端驗證伺服器。

針對 Amazon MSK 與 Lambda 的整合，MSK 叢集充當伺服器，而 Lambda 充當用戶端。

- 若要讓 Lambda 驗證 MSK 叢集，可以將用戶端憑證設定為 Secrets Manager 中的秘密，並在事件來源映射組態中參考此憑證。客戶憑證必須由伺服器信任存放區中的憑證授權機構 (CA) 簽署。
- MSK 叢集也會將伺服器憑證傳送至 Lambda。伺服器憑證必須由 AWS 信任存放區中的憑證授權機構 (CA) 簽署。

Amazon MSK 不支援自行簽署的伺服器憑證，因為 Amazon MSK 中的所有代理程式都使用由 Amazon Trust Services CA (Lambda 預設信任此機構) 簽署的公有憑證。

設定 mTLS 機密

CLIENT_CERTIFICATE_TLS_AUTH 機密必須有憑證欄位和私有金鑰欄位。若為加密的私有金鑰，機密需要私有金鑰密碼。憑證與私有金鑰均必須為 PEM 格式。

> **Note**
>
> Lambda 支援 PBES1 (但不支援 PBES2) 私有金鑰加密演算法。

憑證欄位必須包含憑證清單，以用戶端憑證開頭，隨後則是任何中繼憑證，並以根憑證結尾。每個憑證均必須以新的一行開始，結構如下：

```
-----BEGIN CERTIFICATE-----
        <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager 支援高達 65,536 個位元組的機密，此空間足以容納長憑證鏈。

私有金鑰必須為 PKCS #8 格式，結構如下：

```
-----BEGIN PRIVATE KEY-----
         <private key contents>
-----END PRIVATE KEY-----
```

對於已加密的私有金鑰，請使用下列結構：

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
          <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

下列範例顯示的是使用了已加密私有金鑰之 mTLS 身分驗證的機密內容。若為加密的私有金鑰，您可以在機密中包含私有金鑰密碼。

```
{
 "privateKeyPassword": "testpassword",
 "certificate": "-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
 "privateKey": "-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7KjmgP
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
```

```
}
```

如需有關適用於 Amazon MSK 的 mTLS 詳細資訊，以及如何產生用戶端憑證的說明，請參閱 Amazon Managed Streaming for Apache Kafka Developer Guide 中的 Mutual TLS client authentication for Amazon MSK。

IAM 身分驗證

您可以使用 AWS Identity and Access Management (IAM) 驗證連線至 MSK 叢集之用戶端的身分。透過 IAM 身分驗證，Lambda 憑藉函式執行角色中的許可連線至叢集、擷取記錄，以及執行其他必要動作。如需包含必要許可的政策範例，請參閱 Amazon Managed Streaming for Apache Kafka Developer Guide 中的 Create authorization policies for the IAM role。

若 MSK 叢集上的 IAM 身分驗證為作用中，且您未提供秘密，則 Lambda 會自動預設使用 IAM 身分驗證。

如需有關 Amazon MSK 中 IAM 身分驗證的詳細資訊，請參閱 IAM access control。

Lambda 選擇引導代理程式的方法

Lambda 會依據叢集上可用的身分驗證方法，以及您是否提供了身分驗證密碼，以此選擇引導代理程式。若您提供了 MTL 或 SASL/SCRAM 的密碼，則 Lambda 會自動選擇該身分驗證方法。若您未提供密碼，Lambda 會選取叢集上作用中的安全強度最高的身分驗證方法。以下是 Lambda 選擇代理程式的優先順序，身分驗證安全強度依次遞減：

- mTLS (已提供 mTLS 密碼)
- SASL/SCRAM (已提供 SASL /SCROM 密碼)
- SASL IAM (未提供任何密碼，且 IAM 身分驗證在作用中)
- 未驗證的 TLS (未提供任何密碼，且 IAM 身分驗證未在作用中)
- 純文字 (未提供任何密碼，且 IAM 身分驗證和未經身分驗證的 TLS 皆未在作用中)

> **Note**
>
> 若 Lambda 無法連線至最安全的代理程式類型，Lambda 便不會嘗試連線至其他 (安全強度較弱) 的代理程式類型。若您要讓 Lambda 選擇安全強度較弱較弱的代理程式類型，請停用叢集上所有安全強度較弱更高的身分驗證方法。

為 Amazon MSK 事件來源建立 Lambda 事件來源映射

若要建立事件來源映射，可以使用 Lambda 主控台、AWS Command Line Interface (CLI) 或 AWS SDK。

> **Note**
>
> 當您建立事件來源映射時，Lambda 會在包含 MSK 叢集的私有子網路內建立 Hyperplane ENI，使 Lambda 能建立安全連線。此 Hyperplane ENI 會使用 MSK 叢集的子網路和安全群組組態，而不是 Lambda 函式。

下列主控台步驟會將 Amazon MSK 叢集新增為 Lambda 函式的觸發程序。在此之下，此操作會建立事件來源映射資源。

將 Amazon MSK 觸發條件新增至 Lambda 函數（主控台）

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要新增 Amazon MSK 觸發程序的 Lambda 函式名稱。
3. 在函數概觀下，選擇新增觸發條件。
4. 在觸發程序組態欄位中，選擇 MSK。
5. 若要指定 Kafka 叢集詳細資訊，請執行下列動作：
    1. 對於 MSK cluster (MSK 叢集)，請選取您的叢集。
    2. 在主題名稱欄位中，輸入要從中取用訊息的 Kafka 主題名稱。
    3. (選用) 在取用者群組 ID 欄位中，輸入要加入的 Kafka 取用者群組 ID (如適用)。如需詳細資訊，請參閱the section called “取用者群組 ID”。
6. 在叢集身分驗證欄位中，進行必要的組態設定。如需有關叢集身分驗證的詳細資訊，請參閱the section called “叢集身分驗證”。
    - 如果希望 Lambda 在建立連線時對 MSK 叢集執行身分驗證，請開啟使用身分驗證。建議進行身分驗證。
    - 如果使用身分驗證，在身分驗證方法欄位中，選擇要使用的身分驗證方法。
    - 如果使用身分驗證，在 Secrets Manager 金鑰欄位中，選擇包含存取叢集所需身分驗證憑證的 Secrets Manager 金鑰。
7. 在事件輪詢器組態欄位中，進行必要的組態設定。

- 選擇啟用觸發程序，在建立完成後立即啟用觸發程序。
- 選擇是否要為事件來源映射設定佈建模式。如需詳細資訊，請參閱the section called “事件輪詢器擴展”。
  - 如果您設定佈建模式，請輸入最小值事件輪詢器的值、最大值事件輪詢器的值，以及 PollerGroupName 的選用值，以指定相同事件來源 VPC 內多個 ESMs 的分組。
- 在開始位置欄位中，選擇希望 Lambda 開始從串流讀取的方式。如需詳細資訊，請參閱the section called “輪詢與串流位置”。

8. 在批次處理欄位中，進行必要的組態設定。如需有關批次處理的詳細資訊，請參閱the section called “批次處理行為”。

   1. 對於 批次大小，輸入單一批次中要擷取的訊息數量上限。
   2. 在批次時段欄位中，輸入 Lambda 調用函式之前收集記錄所耗費的秒數上限。

9. 在篩選欄位中，進行必要的組態設定。如需有關篩選的詳細資訊，請參閱 the section called “事件篩選”。

   - 在篩選條件欄位中，新增篩選條件定義，決定是否處理事件。

10. 在失敗處理欄位中，進行必要的組態設定。如需有關失敗處理的詳細資訊，請參閱the section called “保留失敗的調用”。

    - 在失敗時目的地欄位中，指定失敗時目的地的 ARN。

11. 在標籤欄位中，輸入要與此事件來源映射建立關聯的標籤。
12. 若要建立觸發條件，請選擇 新增 。

您也可以使用 CLI AWS 搭配 create-event-source-mapping 命令來建立事件來源映射。下列範例建立了事件來源映射，可將 Lambda 函式 `my-msk-function` 映射至 `AWSKafkaTopic` 主題 (從 `LATEST` 訊息開始)。此命令還使用了 SourceAccessConfiguration 物件，可指示 Lambda 在連線至叢集時使用 SASL/SCRAM 身分驗證。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-
fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

如果叢集使用 mTLS 身分驗證，則需包含指定 CLIENT_CERTIFICATE_TLS_AUTH 和 Secrets Manager 金鑰 ARN 的 SourceAccessConfiguration 物件。如下列命令所示：

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI": "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]'
```

如果叢集使用 IAM 身分驗證，則不需要 SourceAccessConfiguration 物件。如下列命令所示：

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

在 Lambda 中建立跨帳戶事件來源映射

您可以使用多 VPC 私有連線，將 Lambda 函數連接到不同 AWS 帳戶中的佈建 MSK 叢集。多 VPC 連線使用 AWS PrivateLink，這會保留 AWS 網路中的所有流量。

> **Note**
>
> 您無法為無伺服器 MSK 叢集建立跨帳戶事件來源映射。

若要建立跨帳戶事件來源映射，您必須先為 MSK 叢集設定多 VPC 連線。建立事件來源映射時，請使用受管理的 VPC 連線 ARN 而非叢集 ARN，如下列範例所示。根據 MSK 叢集使用的驗證類型，CreateEventSourceMapping 操作也會有所不同。

Example— 為使用 IAM 身分驗證的叢集建立跨帳戶的事件來源映射

當叢集使用 IAM 角色型身分驗證時，您不需要 SourceAccessConfiguration 物件。範例：

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
```

```
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function
```

Example— 為使用 SASL/SCRAM 身分驗證的叢集建立跨帳戶的事件來源映射

如果叢集使用 SASL/SCRAM 身分驗證，則您必須包含指定 SASL_SCRAM_512_AUTH 和 Secrets Manager 機密 ARN 的 SourceAccessConfiguration。

透過 SASL/SCRAM 身分驗證，有兩種方式可將機密用於跨帳戶 Amazon MSK 事件來源映射：

- 在 Lambda 函數帳戶中建立機密，並將其與叢集機密同步。建立一個輪換以使兩個機密保持同步。此選項允許您從函數帳戶控制機密。
- 使用機密必須與 Amazon MSK 叢集相關聯。此機密必須允許 Lambda 函數帳戶的跨帳戶存取權。如需詳細資訊，請參閱不同帳戶中使用者 AWS Secrets Manager 機密的許可。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

Example— 為使用 mTLS 身分驗證的叢集建立跨帳戶的事件來源映射

如果叢集使用 mTLS 身分驗證，則您必須包含指定 CLIENT_CERTIFICATE_TLS_AUTH 和 Secrets Manager 機密 ARN 的 SourceAccessConfiguration。機密可以儲存在叢集帳戶或 Lambda 函數帳戶中。

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:vpc-connection/444455556666/
my-cluster-name/51jn98b4-0a61-46cc-b0a6-61g9a3d797d5-7 \
  --topics AWSKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "CLIENT_CERTIFICATE_TLS_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:444455556666:secret:my-secret"}]'
```

Lambda 中的所有 Amazon MSK 事件來源組態參數

所有 Lambda 事件來源類型都會共用相同的 CreateEventSourceMapping 和 UpdateEventSourceMapping API 操作。但是，僅有部分參數適用於 Amazon MSK，如下表所示。

參數	必要	預設	備註
AmazonManagedKafkaEventSourceConfig	N	包含預設為一個唯一值的 ConsumerGroupId 欄位。	只能在建立時進行設定
BatchSize	否	100	上限：10,000
DestinationConfig	N	N/A	the section called “保留失敗的調用”
已啟用	N	True	
BisectBatchOnFunctionError	N	False	the section called “重試組態”
FunctionResponseTypes	N	N/A	the section called “重試組態”
MaximumRecordAgeInSeconds	N	-1 （無限）	the section called “重試組態”
MaximumRetryAttempts	N	-1 （無限）	the section called “重試組態”
EventSourceArn	Y	N/A	只能在建立時進行設定
FilterCriteria	N	N/A	控制 Lambda 將哪些事件傳送至您的函數
FunctionName	是	N/A	
KMSKeyArn	N	N/A	the section called “篩選條件加密”

參數	必要	預設	備註
MaximumBatchingWindowInSeconds	N	500 毫秒	批次處理行為
ProvisionedPollersConfig	N	`MinimumPollers` ：若未指定，預設值為 1。 `MaximumPollers` ：若未指定，預設值為 200。 `PollerGroupName` ：不適用	the section called “佈建模式”
SourceAccessConfigurations	N	沒有憑證	您事件來源的 SASL/SCRAM 或 CLIENT_CERTIFICATE_TLS_AUTH (MutualTLS) 身分驗證憑證。
StartingPosition	Y	N/A	AT_TIMESTAMP、TRIM_HORIZON 或 LATEST 只能在建立時進行設定
StartingPositionTimestamp	N	N/A	StartingPosition 設定為 AT_TIMESTAMP 時需要
Tags (標籤)	N	N/A	the section called “事件來源映射標籤”

參數	必要	預設	備註
主題	Y	N/A	Kafka 主題名稱 只能在建立時進行設定

> **Note**
>
> 當您指定 時`PollerGroupName`，相同 Amazon VPC 中的多個 ESMs 可以共用事件輪詢單元 (EPU) 容量。您可以使用此選項來最佳化 ESMs 的佈建模式成本。ESM 分組的需求：
>
> - ESMs 必須位於相同的 Amazon VPC 內
> - 每個輪詢器群組最多 100 ESMs
> - 群組中所有 ESMs 的彙總輪詢器上限不能超過 2000
>
> 您可以更新 `PollerGroupName`，將 ESM 移至不同的群組，或將 `PollerGroupName`設定為空字串 ("")，從群組中移除 ESM。

教學課程：使用 Amazon MSK 事件來源映射來調用 Lambda 函數

在本教學課程中，您將執行下列操作：

- 在與現有 Amazon MSK 叢集相同的 AWS 帳戶中建立 Lambda 函數。
- 為 Lambda 設定聯網和身分驗證，以與 Amazon MSK 通訊。
- 設定 Lambda Amazon MSK 事件來源映射，當主題中出現事件時，該映射會執行您的 Lambda 函數。

完成這些步驟後，當事件傳送至 Amazon MSK 時，您將能夠設定 Lambda 函數，以使用您自己的自訂 Lambda 程式碼自動處理這些事件。

您可以使用此功能做些什麼？

範例解決方案：使用 MSK 事件來源映射為您的客戶提供即時分數。

請考慮以下案例：您的公司託管了一個 Web 應用程式，您的客戶可以在其中檢視即時事件的相關資訊，例如運動遊戲。來自遊戲的資訊更新會透過 Amazon MSK 上的一個 Kafka 主題提供給您的團隊。您想要設計一個使用 MSK 主題更新的解決方案，以便在您所開發的應用程式內為客戶提供即時事件的更新檢視。您已決定採用下列設計方法：您的用戶端應用程式將與 AWS中託管的無伺服器後端通訊。用戶端將使用 Amazon API Gateway WebSocket API，透過 WebSocket 工作階段連線。

在此解決方案中，您需要一個元件來讀取 MSK 事件、執行一些自訂邏輯來準備應用程式層的事件，然後將該資訊轉送至 API Gateway API。您可以使用 實作此元件 AWS Lambda，方法是在 Lambda 函數中提供自訂邏輯，然後使用 AWS Lambda Amazon MSK 事件來源映射呼叫它。

如需使用 Amazon API Gateway WebSocket API 實作解決方案的詳細資訊，請參閱 API Gateway 文件中的 WebSocket API tutorials。

先決條件

具有下列預先設定資源 AWS 的帳戶：

為了滿足這些先決條件，我們建議您遵循 Amazon MSK 文件中的 Getting started using Amazon MSK。

- Amazon MSK 叢集。請參閱開始使用 Amazon MSK中的建立 Amazon MSK 叢集。
- 下列組態：
  - 確定叢集安全設定中已啟用 IAM 角色型身分驗證。這會透過限制您的 Lambda 函數僅存取所需的 Amazon MSK 資源來提高您的安全性。在新的 Amazon MSK 叢集上預設會啟用此功能。
  - 確保您的叢集聯網設定中的公有存取已關閉。限制 Amazon MSK 叢集對網際網路的存取，可限制有多少中介機構處理您的資料，藉此提高您的安全性。在新的 Amazon MSK 叢集上預設會啟用此功能。
- Amazon MSK 叢集中用於此解決方案的 Kafka 主題。請參閱開始使用 Amazon MSK 中的建立主題。
- Kafka 管理員主機，設定為從 Kafka 叢集擷取資訊，並將 Kafka 事件傳送至您的主題進行測試，例如已安裝 Kafka 管理員 CLI 和 Amazon MSK IAM 程式庫的 Amazon EC2 執行個體。請參閱開始使用 Amazon MSK 中的建立用戶端機器。

設定這些資源後，請從 AWS 您的帳戶收集下列資訊，以確認您已準備好繼續。

- Amazon MSK 叢集的名稱。您可以在 Amazon MSK 主控台中找到此資訊。
- 叢集 UUID 是 Amazon MSK 叢集 ARN 的一部分，可以在 Amazon MSK 主控台中找到。請執行 Amazon MSK 文件中 Listing clusters 中的程序來尋找此資訊。

- 與您的 Amazon MSK 叢集相關聯的安全群組。您可以在 Amazon MSK 主控台中找到此資訊。在下列步驟中，將這些安全群組稱為 *clusterSecurityGroups*。
- 包含 Amazon MSK 叢集的 Amazon VPC 的 ID。您可以在 Amazon MSK 主控台中識別與 Amazon MSK 叢集相關聯的子網路，然後在 Amazon VPC 主控台中識別與子網路相關聯的 Amazon VPC，藉此找到此資訊。
- 解決方案中使用的 Kafka 主題的名稱。您可以從 Kafka 管理員主機使用 Kafka `topics` CLI 呼叫 Amazon MSK 叢集，藉此找到此資訊。如需關於主題 CLI 的詳細資訊，請參閱 Kafka 文件中的 Adding and removing topics。
- Kafka 主題的取用者群組名稱，適合供 Lambda 函數使用。此群組可由 Lambda 自動建立，因此不需要使用 Kafka CLI 建立。如果您需要管理取用者群組，以進一步了解取用者群組 CLI 的詳細資訊，則請參閱 Kafka 文件中的 Managing Consumer Groups。

您 AWS 帳戶中的下列許可：

- 建立和管理 Lambda 函數的許可。
- 建立 IAM 政策並與您的 Lambda 函數建立關聯的許可。
- 在託管 Amazon MSK 叢集的 Amazon VPC 中建立 Amazon VPC 端點和變更聯網組態的許可。

安裝 AWS Command Line Interface

如果您尚未安裝 AWS Command Line Interface，請依照安裝或更新最新版本的 AWS CLI中的步驟進行安裝。

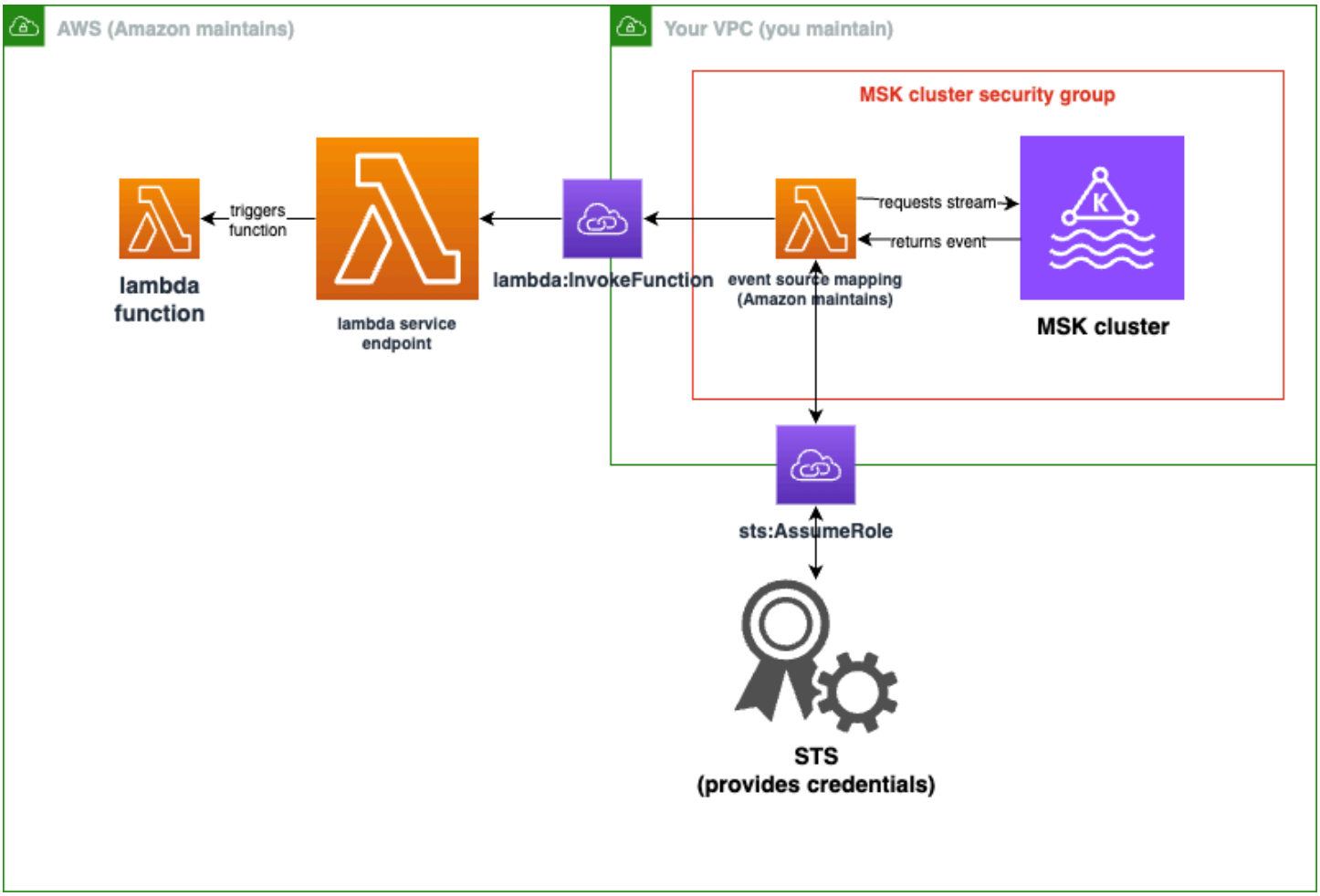
本教學課程需使用命令列終端機或 Shell 來執行命令。在 Linux 和 macOS 中，使用您偏好的 Shell 和套件管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。

設定供 Lambda 與 Amazon MSK 通訊的網路連線

使用 AWS PrivateLink 連接 Lambda 和 Amazon MSK。您可以在 Amazon VPC 主控台中建立介面 Amazon VPC 端點來執行此操作。如需關於聯網組態的詳細資訊，請參閱the section called “叢集與網路設定”。

當 Amazon MSK 事件來源映射代表 Lambda 函數執行時，它會擔任 Lambda 函數的執行角色。此 IAM 角色會授權映射存取 IAM 保護的資源，例如您的 Amazon MSK 叢集。雖然元件共用執行角色，但 Amazon MSK 映射和您的 Lambda 函數對各自的任務有不同的連線需求，如下圖所示。



事件來源映射屬於 Amazon MSK 叢集安全群組。在此聯網步驟中，從您的 Amazon MSK 叢集 VPC 建立 Amazon VPC 端點，將事件來源映射連線到 Lambda 和 STS 服務。保護這些端點，接受來自 Amazon MSK 叢集安全群組的流量。然後，調整 Amazon MSK 叢集安全群組，允許事件來源映射與 Amazon MSK 叢集通訊。

您可以使用 AWS 管理主控台來設定下列步驟。

若要設定介面 Amazon VPC 端點連線 Lambda 和 Amazon MSK

1. 為您的介面 Amazon VPC 端點 *endpointSecurityGroup* 建立安全群組，允許 443 上來自 *clusterSecurityGroups* 的傳入 TCP 流量。請遵循 Amazon EC2 文件中的 Create a security group 中的程序來建立安全群組。然後，遵循 Amazon EC2 文件中的 Add rules to a security group 中的程序，以新增適當的規則。

   使用下列資訊建立一個安全群組：

   在新增傳入規則時，為 *clusterSecurityGroups* 中的每個安全群組建立規則。對於每條規則：

   - 針對類型，選取 HTTPS。
   - 針對來源，選取其中一個 *clusterSecurityGroups*。
2. 建立一個端點，將 Lambda 服務連線至包含 Amazon MSK 叢集的 Amazon VPC。遵循 Create an interface endpoint 中的程序。

   使用下列資訊建立一個介面端點：

   - 針對服務名稱，選取 `com.amazonaws.`*regionName*`.lambda`，其中 *regionName* 會託管您的 Lambda 函數。
   - 針對 VPC，選取包含 Amazon MSK 叢集的 Amazon VPC。
   - 針對安全群組，選取先前建立的 *endpointSecurityGroup*。
   - 針對子網路，選取託管您的 Amazon MSK 叢集的子網路。
   - 針對政策，提供下列政策文件，這會保護端點以供 Lambda 服務主體用於 `lambda:InvokeFunction` 動作。

```
{
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
```

```
    ]
}
```

- 確保啟用 DNS 名稱保持設定狀態。

3. 建立端點，將 AWS STS 服務連線至包含 Amazon MSK 叢集的 Amazon VPC。遵循 Create an interface endpoint 中的程序。

   使用下列資訊建立一個介面端點：

   - 針對服務名稱，選取 AWS STS。
   - 針對 VPC，選取包含 Amazon MSK 叢集的 Amazon VPC。
   - 針對安全群組，選取 *endpointSecurityGroup*。
   - 針對子網路，選取託管您的 Amazon MSK 叢集的子網路。
   - 針對政策，提供下列政策文件，這會保護端點以供 Lambda 服務主體用於 `sts:AssumeRole` 動作。

   ```
   {
       "Statement": [
           {
               "Action": "sts:AssumeRole",
               "Effect": "Allow",
               "Principal": {
                   "Service": [
                       "lambda.amazonaws.com"
                   ]
               },
               "Resource": "*"
           }
       ]
   }
   ```

   - 確保啟用 DNS 名稱保持設定狀態。

4. 對於與您的 Amazon MSK 叢集相關聯的每個安全群組，亦即，在 *clusterSecurityGroups* 中，允許以下項目：

   - 允許 9098 上所有 *clusterSecurityGroups* 的傳入和傳出 TCP 流量，包括其本身以內的 TCP 流量。
   - 允許 443 上的所有傳出 TCP 流量。

根據預設，安全群組規則允許其中一些流量，因此如果您的叢集連接到單一安全群組，且該群組具有預設規則，則不需要額外的規則。若要調整安全群組規則，請遵循 Amazon EC2 文件中的 Add rules to a security group 中的程序。

使用下列資訊將規則新增至您的安全群組：

- 針對連接埠 9098 的每個傳入或傳出規則，提供
  - 針對類型，選取自訂 TCP。
  - 對於連接埠範圍，提供 9098。
  - 針對來源，提供其中一個 *clusterSecurityGroups*。
- 對於連接埠 443 的每條傳入規則，針對類型，選取 HTTPS。

為 Lambda 建立 IAM 角色，以從 Amazon MSK 主題讀取

識別 Lambda 從 Amazon MSK 主題讀取的身分驗證要求，然後在政策中定義這些要求。建立角色 *lambdaAuthRole*，授權 Lambda 使用這些許可。使用 `kafka-cluster` IAM 動作授權 Amazon MSK 叢集上的動作。然後，授權 Lambda 執行探索和連線至 Amazon MSK 叢集所需的 Amazon MSK `kafka` 和 Amazon EC2 動作，以及 CloudWatch 動作 (以便 Lambda 可以記錄其已完成的動作)。

若要描述 Lambda 從 Amazon MSK 讀取的身分驗證要求

1. 撰寫 IAM 政策文件 (JSON 文件) *clusterAuthPolicy*，允許 Lambda 使用您的 Kafka 取用者群組，從 Amazon MSK 叢集中的 Kafka 主題讀取。Lambda 要求在讀取時設定 Kafka 取用者群組。

   修改以下範本以符合您的先決條件：

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeGroup",
                "kafka-cluster:AlterGroup",
```

```
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:ReadData",
                "kafka-cluster:DescribeClusterDynamicConfiguration"
            ],
            "Resource": [
                "arn:aws:kafka:us-
east-1:111122223333:cluster/mskClusterName/cluster-uuid",
                "arn:aws:kafka:us-
east-1:111122223333:topic/mskClusterName/cluster-uuid/mskTopicName",
                "arn:aws:kafka:us-
east-1:111122223333:group/mskClusterName/cluster-uuid/mskGroupName"
            ]
        }
    ]
}
```

如需詳細資訊，請參閱 the section called “設定 許可”。在撰寫政策時：

- 使用 Amazon MSK 叢集 AWS 帳戶 的 和 取代 *us-east-1* AWS 區域 和 *111122223333*。
- 對於 *mskClusterName*，請提供 Amazon MSK 叢集的名稱。
- 對於 *cluster-uuid*，請提供 Amazon MSK 叢集 ARN 中的 UUID。
- 對於 *mskTopicName*，請提供 Kafka 主題的名稱。
- 針對 *mskGroupName*，請提供 Kafka 取用者群組的名稱。

2. 識別 Lambda 探索和連線 Amazon MSK 叢集以及記錄這些事件所需的 Amazon MSK、Amazon EC2 和 CloudWatch 許可。

   AWSLambdaMSKExecutionRole 受管政策允許定義必要的許可。在下列步驟中使用該政策。

   在生產環境中，評估 AWSLambdaMSKExecutionRole 以根據最低權限原則限制執行角色政策，然後為您的角色撰寫取代此受管政策的政策。

如需有關 IAM 政策語言的詳細資訊，請參閱 IAM 文件。

現在您已撰寫政策文件，請建立 IAM 政策，以便將其連接至您的角色。您可以使用主控台執行下列程序，以完成此操作。

若要從政策文件建立 IAM 政策

1. 登入 AWS 管理主控台 ，並在 https：//https://console.aws.amazon.com/iam/ 開啟 IAM 主控台。

2. 在左側的導覽窗格中，選擇 Policies (政策)。
3. 選擇建立政策。
4. 在政策編輯器中，選擇 JSON 選項。
5. 貼上 *clusterAuthPolicy*。
6. 將許可新增至政策後，請選擇下一步。
7. 在檢視與建立頁面上，為您正在建立的政策輸入政策名稱與描述 (選用)。檢視此政策中定義的許可，來查看您的政策所授予的許可。
8. 選擇 Create policy (建立政策) 儲存您的新政策。

如需詳細資訊，請參閱 IAM 文件中的 Creating IAM policies。

現在您擁有適當的 IAM 政策，請建立一個角色並將這些政策連接至該角色。您可以使用主控台執行下列程序，以完成此操作。

若要在 IAM 主控台中建立執行角色

1. 在 IAM 主控台中開啟 角色頁面。
2. 選擇建立角色。
3. 在受信任的實體類型下，選擇 AWS 服務。
4. 在 使用案例 下，選擇 Lambda。
5. 選擇下一步。
6. 選取以下政策：
   - *clusterAuthPolicy*
   - AWSLambdaMSKExecutionRole
7. 選擇下一步。
8. 針對角色名稱，輸入 *lambdaAuthRole*，然後選擇建立角色。

如需詳細資訊，請參閱the section called “執行角色 (函數存取其他資源的許可)”。

建立 Lambda 函數以從您的 Amazon MSK 主題中讀取

建立設定為使用您的 IAM 角色的 Lambda 函數。您可以使用主控台建立 Lambda 函數。

若要使用驗證組態建立 Lambda 函數

1. 開啟 Lambda 主控台，然後從標頭選取建立函數。
2. 選取從頭開始撰寫。
3. 針對函數名稱，提供您選擇的適當名稱。
4. 針對執行時期，選擇最新支援的 Node.js 版本，以使用本教學課程中提供的程式碼。
5. 選擇變更預設執行角色。
6. 選取使用現有角色。
7. 針對現有角色，選取 *lambdaAuthRole*。

在生產環境中，您通常需要將更多政策新增至 Lambda 函數的執行角色，以有意義的方式處理您的 Amazon MSK 事件。如需將政策新增至角色的詳細資訊，請參閱 IAM 文件中的 Add or remove identity permissions。

建立 Lambda 函數的事件來源映射

您的 Amazon MSK 事件來源映射為 Lambda 服務提供了在發生適當的 Amazon MSK 事件時調用 Lambda 所需的資訊。您可以使用主控台建立 Amazon MSK 映射。建立 Lambda 觸發條件，然後會自動設定事件來源映射。

若要建立 Lambda 觸發條件 (和事件來源映射)

1. 導覽至 Lambda 函數的概觀頁面。
2. 在函數概觀區段中，選擇左下角的新增觸發條件。
3. 在選取來源下拉式清單中，選取 Amazon MSK。
4. 請勿設定身分驗證。
5. 針對 MSK 叢集，請選取您的叢集名稱。
6. 針對批次大小，請輸入 1。此步驟可讓此功能更易於測試，而且不是生產中的理想值。
7. 針對主題名稱，請提供 Kafka 主題名稱。
8. 針對取用者群組 ID，請提供 Kafka 取用者群組的 ID。

更新您的 Lambda 函數以讀取串流資料

Lambda 透過事件方法參數提供關於 Kafka 事件的資訊。如需 Amazon MSK 事件的結構範例，請參閱the section called “範例事件”。了解如何解譯 Lambda 轉送的 Amazon MSK 事件之後，您可以變更 Lambda 函數程式碼，以使用它們提供的資訊。

將下列程式碼提供給 Lambda 函數，以記錄 Lambda Amazon MSK 事件的內容供測試之用：

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來取用 Amazon MSK 事件。

```
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KafkaEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace MSKLambda;

public class Function
{


    /// <param name="input">The event for the Lambda function handler to
 process.</param>
    /// <param name="context">The ILambdaContext that provides methods for
 logging and describing the Lambda environment.</param>
    /// <returns></returns>
    public void FunctionHandler(KafkaEvent evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Records)
        {
            Console.WriteLine("Key:" + record.Key);
            foreach (var eventRecord in record.Value)
            {
```

```
                var valueBytes = eventRecord.Value.ToArray();
                var valueText = Encoding.UTF8.GetString(valueBytes);

                Console.WriteLine("Message:" + valueText);
            }
        }
    }


}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來取用 Amazon MSK 事件。

```
package main

import (
 "encoding/base64"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.KafkaEvent) {
 for key, records := range event.Records {
  fmt.Println("Key:", key)

  for _, record := range records {
   fmt.Println("Record:", record)
```

```
   decodedValue, _ := base64.StdEncoding.DecodeString(record.Value)
   message := string(decodedValue)
   fmt.Println("Message:", message)
  }
 }
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來取用 Amazon MSK 事件。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent.KafkaEventRecord;

import java.util.Base64;
import java.util.Map;

public class Example implements RequestHandler<KafkaEvent, Void> {

    @Override
    public Void handleRequest(KafkaEvent event, Context context) {
        for (Map.Entry<String, java.util.List<KafkaEventRecord>> entry :
 event.getRecords().entrySet()) {
            String key = entry.getKey();
            System.out.println("Key: " + key);

            for (KafkaEventRecord record : entry.getValue()) {
```

```
                System.out.println("Record: " + record);

                byte[] value = Base64.getDecoder().decode(record.getValue());
                String message = new String(value);
                System.out.println("Message: " + message);
            }
        }

        return null;
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 來取用 Amazon MSK 事件。

```
exports.handler = async (event) => {
    // Iterate through keys
    for (let key in event.records) {
      console.log('Key: ', key)
      // Iterate through records
      event.records[key].map((record) => {
        console.log('Record: ', record)
        // Decode base64
        const msg = Buffer.from(record.value, 'base64').toString()
        console.log('Message:', msg)
      })
    }
}
```

搭配 Lambda 使用 TypeScript 來取用 Amazon MSK 事件。

```
import { MSKEvent, Context } from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "msk-handler-sample",
});

export const handler = async (
  event: MSKEvent,
  context: Context
): Promise<void> => {
  for (const [topic, topicRecords] of Object.entries(event.records)) {
    logger.info(`Processing key: ${topic}`);

    // Process each record in the partition
    for (const record of topicRecords) {
      try {
        // Decode the message value from base64
        const decodedMessage = Buffer.from(record.value, 'base64').toString();

        logger.info({
          message: decodedMessage
        });
      }
      catch (error) {
        logger.error('Error processing event', { error });
        throw error;
      }
    };
  }
}
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來取用 Amazon MSK 事件。

```
<?php
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

// using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kafka\KafkaEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): void
    {
        $kafkaEvent = new KafkaEvent($event);
        $this->logger->info("Processing records");
        $records = $kafkaEvent->getRecords();
```

```
        foreach ($records as $record) {
            try {
                $key = $record->getKey();
                $this->logger->info("Key: $key");

                $values = $record->getValue();
                $this->logger->info(json_encode($values));

                foreach ($values as $value) {
                    $this->logger->info("Value: $value");
                }

            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來取用 Amazon MSK 事件。

```
import base64

def lambda_handler(event, context):
    # Iterate through keys
```

```
    for key in event['records']:
        print('Key:', key)
        # Iterate through records
        for record in event['records'][key]:
            print('Record:', record)
            # Decode base64
            msg = base64.b64decode(record['value']).decode('utf-8')
            print('Message:', msg)
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來取用 Amazon MSK 事件。

```
require 'base64'

def lambda_handler(event:, context:)
  # Iterate through keys
  event['records'].each do |key, records|
    puts "Key: #{key}"

    # Iterate through records
    records.each do |record|
      puts "Record: #{record}"

      # Decode base64
      msg = Base64.decode64(record['value'])
      puts "Message: #{msg}"
    end
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 取用 Amazon MSK 事件。

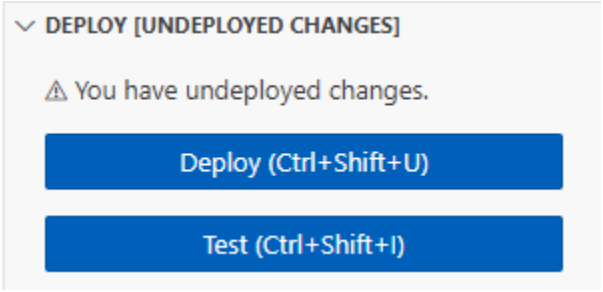
```
use aws_lambda_events::event::kafka::KafkaEvent;
use lambda_runtime::{run, service_fn, tracing, Error, LambdaEvent};
use base64::prelude::*;
use serde_json::{Value};
use tracing::{info};

/// Pre-Requisites:
/// 1. Install Cargo Lambda - see https://www.cargo-lambda.info/guide/getting-
started.html
/// 2. Add packages tracing, tracing-subscriber, serde_json, base64
///
/// This is the main body for the function.
/// Write your code inside it.
/// There are some code example in the following URLs:
/// - https://github.com/awslabs/aws-lambda-rust-runtime/tree/main/examples
/// - https://github.com/aws-samples/serverless-rust-demo/

async fn function_handler(event: LambdaEvent<KafkaEvent>) -> Result<Value, Error>
 {

    let payload = event.payload.records;

    for (_name, records) in payload.iter() {

        for record in records {

         let record_text = record.value.as_ref().ok_or("Value is None")?;
         info!("Record: {}", &record_text);

         // perform Base64 decoding
         let record_bytes = BASE64_STANDARD.decode(record_text)?;
```

```
            let message = std::str::from_utf8(&record_bytes)?;

            info!("Message: {}", message);
           }

    }

    Ok(().into())
}

#[tokio::main]
async fn main() -> Result<(), Error> {

    // required to enable CloudWatch error logging by the runtime
    tracing::init_default_subscriber();
    info!("Setup CW subscriber!");

    run(service_fn(function_handler)).await
}
```

您可以使用主控台將函數程式碼提供給 Lambda。

若要使用主控台程式碼編輯器更新函數程式碼

1. 開啟 Lambda 主控台的函數頁面，然後選取您的函數。
2. 選取程式碼索引標籤。
3. 在程式碼來源窗格中，選取您的原始程式碼檔案，然後在整合式程式碼編輯器中加以編輯。
4. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



測試您的 Lambda 函數，確認其已連線至您的 Amazon MSK 主題

您現在可以透過檢查 CloudWatch 事件日誌，驗證您的 Lambda 是否正在被事件來源調用。

若要驗證您的 Lambda 函數是否正在被調用

1. 使用您的 Kafka 管理員主機，透過 `kafka-console-producer` CLI 產生 Kafka 事件。如需詳細資訊，請參閱 Kafka 文件中的 Write some events into the topic。傳送足夠的事件，以填滿由批次大小定義的批次，用於上一個步驟中定義的事件來源映射，否則 Lambda 會等待調用更多資訊。
2. 如果您的函數執行，Lambda 會將發生的情況寫入 CloudWatch。在主控台中，導覽至您的 Lambda 函數詳情頁面。
3. 選取 Configuration (組態) 索引標籤。
4. 從側邊列，選取監控和操作工具。
5. 在記錄組態下，識別 CloudWatch 日誌群組。日誌群組應以 `/aws/lambda` 開頭。選擇日誌群組連結。
6. 在 CloudWatch 主控台中，檢查日誌事件，了解 Lambda 已傳送至日誌串流的日誌事件。識別是否有日誌事件包含來自 Kafka 事件的訊息，如下圖所示。如果有，您已成功使用 Lambda 事件來源映射將 Lambda 函數連線至 Amazon MSK。

```
2020-08-06T15:06:18.861-04:00	START RequestId: 88ebae59-be0c-4e22-9db7-4154b437e43a Version: $LATEST
2020-08-06T15:06:18.866-04:00	2020-08-06T19:06:18.866Z 88ebae59-be0c-4e22-9db7-4154b437e43a INFO Key: mytopic-0
2020-08-06T15:06:18.866-04:00	2020-08-06T19:06:18.866Z	88ebae59-be0c-4e22-9db7-4154b437e43a	INFO	Record: {
  topic: 'mytopic',
  partition: 0,
  offset: 38,
  timestamp: 1596740777633,
  timestampType: 'CREATE_TIME',
  value: 'TWVzc2FnZSAjMQ=='
}
2020-08-06T15:06:18.866-04:00	2020-08-06T19:06:18.866Z 88ebae59-be0c-4e22-9db7-4154b437e43a INFO Message: Message #1
2020-08-06T15:06:18.890-04:00	END RequestId: 88ebae59-be0c-4e22-9db7-4154b437e43a
```

搭配使用 Lambda 與自我管理 Apache Kafka

本主題說明如何搭配使用 Lambda 與自我管理 Kafka 叢集。在 AWS 術語中，自我管理叢集包含非 AWS 託管的 Kafka 叢集。例如，您可以透過雲端提供者 (例如 Confluent Cloud 或 Redpanda) 託管 Kafka 叢集。

本章說明了如何將自我管理的 Apache Kafka 叢集用作 Lambda 函式的事件來源。將自我管理的 Apache Kafka 與 Lambda 整合的一般流程包含以下步驟：

1. 叢集與網路設定：首先，使用正確的聯網組態設定自我管理的 Apache Kafka 叢集，允許 Lambda 存取叢集。
2. 事件來源映射設定：接著，建立 Lambda 所需的事件來源映射資源，將 Apache Kafka 叢集安全地連線至函式。
3. 函式與許可設定：最後，確保函式已正確設定，並且其執行角色具有必要許可。

Apache Kafka 作為事件來源時，其運作方式類似於使用 Amazon Simple Queue Service (Amazon SQS) 或 Amazon Kinesis。Lambda 會在內部輪詢事件來源中的新訊息，然後同步調用目標 Lambda 函數。Lambda 會批次讀取訊息，並將這些訊息作為事件酬載提供給函數。最大批次大小可進行設定 (預設為 100 則訊息)。如需詳細資訊，請參閱the section called “批次處理行為”。

若要最佳化自我管理 Apache Kafka 事件來源映射的輸送量，請設定佈建模式。在佈建模式中，可以定義配置給事件來源映射的事件輪詢器數量下限和上限。這可以提高事件來源映射處理意外訊息尖峰的能力。如需詳細資訊，請參閱佈建模式。

> **Warning**
>
> Lambda 事件來源映射至少會處理每個事件一次，而且可能會重複處理記錄。為避免與重複事件相關的潛在問題，強烈建議您讓函數程式碼具有等冪性。若要進一步了解，請參閱 AWS 知識中心中的如何使 Lambda 函數具有等冪性。

對於基於 Kafka 的事件來源，Lambda 支援處理控制參數，例如批次間隔和批次大小。如需詳細資訊，請參閱批次處理行為。

如需如何使用自我管理的 Kafka 作為事件來源的範例，請參閱 AWS 運算部落格上的使用自我託管的 Apache Kafka 作為 的事件來源 AWS Lambda。

主題

- 範例事件
- 為 Lambda 設定自我管理的 Apache Kafka 叢集與網路
- 設定 Lambda 執行角色許可
- 為 Lambda 設定自我管理的 Apache Kafka 事件來源

範例事件

Lambda 會在調用 Lambda 函數時，在事件參數中傳送訊息批次。事件酬載包含訊息陣列。陣列中的每個項目包含 Kafka 主題和 Kafka 分割區識別符的詳細資訊，以及時間戳記和 base64 編碼的訊息。

```
{
   "eventSource": "SelfManagedKafka",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
                  ]
               }
```

```
            ]
         }
      ]
   }
}
```

為 Lambda 設定自我管理的 Apache Kafka 叢集與網路

若要將 Lambda 函式連線至自我管理的 Apache Kafka 叢集，需要正確設定叢集及其所在的網路。本頁介紹了如何設定叢集與網路。如果叢集與網路已正確設定，請參閱the section called “設定事件來源”，設定事件來源映射。

主題

- 自我管理的 Apache Kafka 叢集設定
- 設定網路安全

自我管理的 Apache Kafka 叢集設定

您可以透過雲端提供者 (例如 Confluent Cloud 或 Redpanda) 託管自我管理的 Apache Kafka 叢集，也可以在自訂的基礎結構上執行該叢集。確保叢集已正確設定，並且可從 Lambda 事件來源映射所要連線的網路進行存取。

設定網路安全

若要透過事件來源映射授予 Lambda 對自我管理的 Apache Kafka 的完整存取權，您的叢集必須使用公有端點 (公有 IP 位址)，或者您必須提供建立叢集之 Amazon VPC 的存取權。

當您將自我管理的 Apache Kafka 與 Lambda 搭配使用時，請建立 AWS PrivateLink VPC 端點，為您的函數提供 Amazon VPC 中資源的存取權。

> **Note**
>
> 具有事件來源映射的函數需要使用 AWS PrivateLink VPC 端點，該映射使用事件輪詢器的預設 (隨需) 模式。如果您的事件來源映射使用佈建模式，則不需要設定 AWS PrivateLink VPC 端點。

建立端點以提供對下列資源的存取權：

- Lambda：為 Lambda 服務主體建立端點。
- AWS STS：為 AWS STS 建立端點，以便服務主體代表您擔任角色。
- Secrets Manager：如果您的叢集使用 Secrets Manager 來儲存憑證，則請為 Secrets Manager 建立端點。

或者，在 Amazon VPC 中的每個公有子網路上設定一個 NAT 閘道。如需更多詳細資訊，請參閱 the section called “Lambda 函數的網際網路存取”。

當您為自我管理的 Apache Kafka 建立事件來源映射時，Lambda 會檢查是否已存在彈性網絡介面 (ENI)，適用於為您的 Amazon VPC 設定的子網路和安全群組。如果 Lambda 找到現有的 ENI，它會嘗試重複使用它們。否則，Lambda 會建立新的 ENI 以連線至事件來源並調用您的函數。

> **Note**
>
> Lambda 函數一律會在 Lambda 服務所擁有的 VPC 內執行。函數的 VPC 組態不會影響事件來源映射。只有事件來源的聯網組態會決定 Lambda 如何連線至您的事件來源。

為包含叢集的 Amazon VPC 設定安全群組。根據預設，自我管理的 Apache Kafka 會使用下列連接埠：9092。

- 傳入規則：允許與事件來源相關聯之安全群組的預設代理程式連接埠上的所有流量。或者，您可使用自我參照安全群組規則，允許來自同一安全群組內其他執行個體的存取。
- 傳出規則：如果函式需要與 AWS 服務通訊，則允許外部目的地連接埠 443 上的所有流量。或者，如果不需要與其他 AWS 服務通訊，您也可以使用自我參照安全群組規則來限制代理程式的存取權。
- Amazon VPC 端點傳入規則：如果您使用的是 Amazon VPC 端點，與您的 Amazon VPC 端點相關聯的安全群組必須允許來自叢集安全群組的連接埠 443 上的傳入流量。

如果您的叢集使用身分驗證，則您也可以限制 Secrets Manager 端點的端點政策。若要呼叫 Secrets Manager API，Lambda 會使用您的函數角色，而不是 Lambda 服務主體。

Example VPC 端點政策 — Secrets Manager 端點

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
```

```
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws::iam::123456789012:role/my-role"
                ]
            },
            "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
        }
    ]
}
```

當您使用 Amazon VPC 端點時，AWS 會使用端點的彈性網絡介面 (ENI) 來路由您的 API 呼叫，以調用您的函數。Lambda 服務主體需要在使用這些 ENI 的任何角色和函數上呼叫 `lambda:InvokeFunction`。

根據預設，Amazon VPC 端點具有開放的 IAM 政策，允許廣泛存取資源。最佳實務是限制這些政策，以使用該端點執行所需的動作。為了確保事件來源映射能夠調用 Lambda 函數，VPC 端點政策必須允許 Lambda 服務主體呼叫 `sts:AssumeRole` 和 `lambda:InvokeFunction`。限制您的 VPC 端點政策以僅允許源自您組織內部的 API 呼叫，可阻止事件來源映射正常運作，因此在這些政策中需要 `"Resource": "*"`。

下列範例 VPC 端點政策展示了如何授予 Lambda 服務主體對 AWS STS 和 Lambda 端點的必要存取權。

Example VPC 端點政策 – AWS STS 端點

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example VPC 端點政策 – Lambda 端點

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
  }
```

設定 Lambda 執行角色許可

除了存取自我管理的 Kafka 叢集之外，Lambda 函式還需要執行各種 API 動作的許可。您可以將這些許可新增到函數的執行角色。如果您的使用者需要存取任何 API 動作，請將必要的許可新增至 AWS Identity and Access Management (IAM) 使用者或角色的身分政策。

主題

- 必要的 Lambda 函數許可
- 選用 Lambda 函數許可
- 將許可新增至您的執行角色
- 使用 IAM 政策授予使用者存取權

必要的 Lambda 函數許可

若要在 Amazon CloudWatch Logs 中建立日誌並存放在日誌群組中，您的 Lambda 函數在其執行角色中必須具有下列許可：

- logs:CreateLogGroup
- logs:CreateLogStream
- 日誌：PutLogEvents

選用 Lambda 函數許可

您的 Lambda 函數可能需要許可，才能：

- 描述您 Secrets Manager 機密。
- 存取您的 AWS Key Management Service (AWS KMS) 客戶受管金鑰。
- 存取 Amazon VPC。
- 將失敗調用的記錄傳送到目的地。

Secrets Manager 和 AWS KMS 許可

根據您為 Kafka 代理程式設定的存取控制類型，您的 Lambda 函數可能需要存取 Secrets Manager 秘密或解密 AWS KMS 客戶受管金鑰的許可。若要連線至這些資源，函數的執行角色必須具有下列許可：

- secretsmanager:GetSecretValue
- kms:Decrypt

VPC 許可

如果只有某個 VPC 內的使用者可以存取自我管理 Apache Kafka 叢集，則您的 Lambda 函數必須具有存取 Amazon VPC 資源的許可。這些資源包括您的 VPC、子網路、安全群組和網路界面。若要連線至這些資源，函數的執行角色必須具有下列許可：

- ec2:CreateNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets
- ec2:DescribeSecurityGroups

將許可新增至您的執行角色

若要存取自我管理 Apache Kafka 叢集使用的其他 AWS 服務，Lambda 會使用您在 Lambda 函數執行角色中定義的許可政策。

根據預設，Lambda 不允許針對自我管理 Apache Kafka 叢集執行必要或選用的動作。您必須在 IAM 信任政策中為您的執行角色建立並定義這些動作。此範例會示範如何建立允許 Lambda 存取 Amazon VPC 資源的政策。

JSON

```
{
        "Version":"2012-10-17",
        "Statement":[
           {
              "Effect":"Allow",
              "Action":[
                 "ec2:CreateNetworkInterface",
                 "ec2:DescribeNetworkInterfaces",
                 "ec2:DescribeVpcs",
                 "ec2:DeleteNetworkInterface",
                 "ec2:DescribeSubnets",
                 "ec2:DescribeSecurityGroups"
              ],
              "Resource":"*"
           }
        ]
     }
```

使用 IAM 政策授予使用者存取權

根據預設，使用者和角色沒有執行事件來源 API 操作的許可。若要將存取權授予組織或帳戶中的使用者，您可能需要建立或更新身分型政策。如需詳細資訊，請參閱《IAM 使用者指南》中的使用政策控制對 AWS 資源的存取。

如需疑難排解身分驗證與授權錯誤的相關資訊，請參閱the section called “疑難排解”。

為 Lambda 設定自我管理的 Apache Kafka 事件來源

若要將自我管理的 Apache Kafka 叢集用作 Lambda 函式的事件來源，需要建立將這兩項資源連線的事件來源映射。本頁介紹了如何為自我管理的 Apache Kafka 建立事件來源映射。

本頁假設您已正確設定 Kafka 叢集及其所在的網路。如需設定叢集或網路，請參閱the section called “叢集與網路設定”。

主題

- 將自我管理的 Apache Kafka 叢集用作事件來源
- 在 Lambda 中設定叢集身分驗證方法
- 為自我管理的 Apache Kafka 事件來源建立 Lambda 事件來源映射
- Lambda 中所有自我管理的 Apache Kafka 事件來源組態參數

將自我管理的 Apache Kafka 叢集用作事件來源

當您將 Apache Kafka 或 Amazon MSK 叢集新增為 Lambda 函數的觸發條件時，該叢集會用作事件來源。

Lambda 會根據您指定的起始位置，從 Kafka 主題 (您在 CreateEventSourceMapping 請求中指定為 `Topics`) 讀取事件資料。處理成功後，您的 Kafka 主題將遞交給 Kafka 叢集。

Lambda 會依序讀取每個 Kafka 主題分割區的訊息。單一 Lambda 承載可以包含來自多個分割區的訊息。有更多記錄可用時，Lambda 會根據您在 CreateEventSourceMapping 請求中指定的 BatchSize 值，繼續以批次方式來處理記錄，直到函式追上主題的進度為止。

Lambda 處理每個批次後，會遞交該批次中訊息的偏移量。如果函數針對批次中的任何訊息傳回錯誤，Lambda 會重試整個批次的訊息，直至處理成功或訊息過期。您可以將所有重試嘗試失敗時的記錄傳送至失敗時的目的地，以便稍後處理。

> **Note**
>
> 雖然 Lambda 函數的逾時上限通常為 15 分鐘，但 Amazon MSK、自我管理的 Apache Kafka、Amazon DocumentDB 以及 Amazon MQ for ActiveMQ 和 Amazon MQ for RabbitMQ 的事件來源映射只支援 14 分鐘逾時限制上限的函數。

在 Lambda 中設定叢集身分驗證方法

Lambda 支持多種方法來進行您自我管理 Apache Kafka 叢集的身分驗證。請確保您已設定 Kafka 叢集使用其中一種支援的身分驗證方法。如需有關 Kafka 安全性的詳細資訊，請參閱 Kafka 文件的「安全性」一節。

SASL/SCRAM 身分驗證

Lambda 支援 Simple Authentication and Security Layer/Salted Challenge Response Authentication Mechanism (SASL/SCRAM) 身分驗證與 Transport Layer Security (TLS) 加密 (`SASL_SSL`)。Lambda

會傳送加密的憑證，以便向叢集進行身分驗證。Lambda 不支援含純文字的 SASL/SCRAM (SASL_PLAINTEXT)。如需 SASL/SCRAM 身分驗證的詳細資訊，請參閱 RFC 5802。

Lambda 也支援 SASL/PLAIN 身分驗證。由於這套機制使用純文字憑證，與伺服器的連線必須使用 TLS 加密，以確保憑證受到保護。

若使用 SASL 身分驗證，您需將登入憑證儲存為 AWS Secrets Manager中的秘密。如需使用 Secrets Manager 的詳細資訊，請參閱AWS Secrets Manager 《 使用者指南》中的建立 AWS Secrets Manager 秘密。

> **Important**
>
> 若要使用 Secrets Manager 進行身分驗證，秘密必須與 Lambda 函數存放在相同的 AWS 區域中。

交互 TLS 驗證

相互 TLS (mTLS) 可提供用戶端與伺服器之間的雙向身分驗證。用戶端會將憑證傳送至伺服器以供伺服器驗證用戶端，而伺服器會將憑證傳送至用戶端以供用戶端驗證伺服器。

在自我管理 Apache Kafka 中，Lambda 會以用戶端的身分運作。您可以設定用戶端憑證 (做為 Secrets Manager 中的機密) 來驗證 Lambda 與 Kafka 代理程式。用戶端憑證必須由伺服器信任存放區中的憑證授權機構簽署。

Kafka 叢集會傳送伺服器憑證到 Lambda 來驗證 Kafka 代理程式與 Lambda。伺服器憑證可以是公有憑證授權機構憑證或私有憑證授權機構/自行簽署的憑證。公有憑證授權機構憑證必須由 Lambda 信任存放區中的憑證授權機構 (CA) 簽署。若為私有憑證授權機構/自行簽署的憑證，您可以設定伺服器根憑證授權機構憑證 (做為 Secrets Manager 中的機密)。Lambda 使用根憑證來驗證 Kafka 代理程式。

如需有關 mTLS 的詳細資訊，請參閱將 Amazon MSK 的相互 TLS 身分驗證作為事件來源。

設定用戶端憑證機密

CLIENT_CERTIFICATE_TLS_AUTH 機密必須有憑證欄位和私有金鑰欄位。若為加密的私有金鑰，機密需要私有金鑰密碼。憑證與私有金鑰均必須為 PEM 格式。

> **Note**
>
> Lambda 支援 PBES1 (但不支援 PBES2) 私有金鑰加密演算法。

憑證欄位必須包含憑證清單，以用戶端憑證開頭，隨後則是任何中繼憑證，並以根憑證結尾。每個憑證均必須以新的一行開始，結構如下：

```
-----BEGIN CERTIFICATE-----
            <certificate contents>
-----END CERTIFICATE-----
```

Secrets Manager 支援高達 65,536 個位元組的機密，此空間足以容納長憑證鏈。

私有金鑰必須為 PKCS #8 格式，結構如下：

```
-----BEGIN PRIVATE KEY-----
             <private key contents>
-----END PRIVATE KEY-----
```

對於已加密的私有金鑰，請使用下列結構：

```
-----BEGIN ENCRYPTED PRIVATE KEY-----
              <private key contents>
-----END ENCRYPTED PRIVATE KEY-----
```

下列範例顯示的是使用了已加密私有金鑰之 mTLS 身分驗證的機密內容。若為加密的私有金鑰，請在機密中包含私有金鑰密碼。

```
{"privateKeyPassword":"testpassword",
"certificate":"-----BEGIN CERTIFICATE-----
MIIE5DCCAsygAwIBAgIRAPJdwaFaNRrytHBto0j5BA0wDQYJKoZIhvcNAQELBQAw
...
j0Lh4/+1HfgyE2KlmII36dg4IMzNjAFEBZiCRoPimO40s1cRqtFHXoal0QQbIlxk
cmUuiAii9R0=
-----END CERTIFICATE-----
-----BEGIN CERTIFICATE-----
MIIFgjCCA2qgAwIBAgIQdjNZd6uFf9hbNC5RdfmHrzANBgkqhkiG9w0BAQsFADBb
...
rQoiowbbk5wXCheYSANQIfTZ6weQTgiCHCCbuuMKNVS95FkXm0vqVD/YpXKwA/no
c8PH3PSoAaRwMMgOSA2ALJvbRz8mpg==
-----END CERTIFICATE-----",
"privateKey":"-----BEGIN ENCRYPTED PRIVATE KEY-----
MIIFKzBVBgkqhkiG9w0BBQ0wSDAnBgkqhkiG9w0BBQwwGgQUiAFcK5hT/X7Kjmgp
...
QrSekqF+kWzmB6nAfSzgO9IaoAaytLvNgGTckWeUkWn/V0Ck+LdGUXzAC4RxZnoQ
```

```
zp2mwJn2NYB7AZ7+imp0azDZb+8YG2aUCiyqb6PnnA==
-----END ENCRYPTED PRIVATE KEY-----"
}
```

設定伺服器根憑證授權機構憑證機密

如果您的 Kafka 代理程式使用 TLS 加密與私有憑證授權機構簽署的憑證，則建立此機密。您可以使用 TLS 加密以供 VPC、SASL/SCRAM、SASL/PLAIN 或 mTLS 身分驗證之用。

伺服器根憑證授權機構憑證機密必須有包含 Kafka 代理程式根憑證授權機構憑證 (格式為 PEM) 的欄位。下列範例說明機密的結構。

```
{"certificate":"-----BEGIN CERTIFICATE-----
MIID7zCCAtegAwIBAgIBADANBgkqhkiG9w0BAQsFADCBmDELMAkGA1UEBhMCVVMx
EDAOBgNVBAgTB0FyaXpvbmExEzARBgNVBAcTClNjb3R0c2RhbGUxJTAjBgNVBAoT
HFN0YXJmaWVsZCBUZWNobm9sb2dpZXMsIEluYy4xOzA5BgNVBAMTMlN0YXJmaWVs
ZCBTZXJ2aWNlcyBSb290IENlcnRpZmljYXRlIEF1dG...
-----END CERTIFICATE-----"
}
```

為自我管理的 Apache Kafka 事件來源建立 Lambda 事件來源映射

若要建立事件來源映射，可以使用 Lambda 主控台、AWS Command Line Interface (CLI) 或 AWS SDK。

下列主控台步驟會將自我管理的 Apache Kafka 叢集新增為 Lambda 函式的觸發程序。在此之下，此操作會建立事件來源映射資源。

先決條件

- 自我管理 Apache Kafka 叢集。Lambda 支援 Apache Kafka 版本 0.10.1.0 及更高版本。
- 具有許可的執行角色，可存取自我管理的 Kafka 叢集使用 AWS 的資源。

新增自我管理 Kafka 叢集 (主控台)

按照下列步驟，將您的 Apache Kafka 叢集和 Kafka 主題新增為 Lambda 函數的觸發條件。

將 Apache Kafka 觸發條件新增至您的 Lambda 函數 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇 Lambda 函數的名稱。

3. 在函數概觀下，選擇新增觸發條件。
4. 在 Trigger configuration (觸發條件) 下，執行下列動作：
   a. 選擇 Apache Kafka 觸發條件類型。
   b. 對於 Bootstrap 伺服器，輸入叢集中 Kafka 代理程式的主機和連接埠對地址，然後選擇 新增。針對叢集中的每個 Kafka 代理程式重複此操作。
   c. 對於 Topic name (主題名稱)，輸入用於在叢集中存儲記錄之 Kafka 主題的名稱。
   d. 如果您設定佈建模式，請輸入最小值事件輪詢器的值、最大值事件輪詢器的值，以及 PollerGroupName 的選用值，以指定相同事件來源 VPC 內多個 ESMs 的分組。
   e. (選用) 對於 批次大小，輸入單一批次中接收的最大記錄數。
   f. 對於 Batch window (批次時段)，輸入 Lambda 調用函數之前收集記錄所花費的最長秒數。
   g. (選用) 對於取用者群組 ID ，輸入要加入的 Kafka 取用者群組 ID。
   h. (選用) 在開始位置欄位中，選擇最新，可從最新記錄開始讀取串流；選擇水平修剪，可從最早的可用記錄開始；選擇在時間戳記，可指定開始讀取的時間戳記。
   i. (選用) 若為 VPC，請為您的 Kafka 叢集選擇 Amazon VPC。然後，選擇 VPC 子網路 和 VPC 安全群組 。

      如果只有您的 VPC 內的使用者會存取代理程式，則必須要有此設定。

   j. (選用) 對於 身分驗證 ，選擇 新增 ，然後執行下列動作：
      i. 選擇您叢集中 Kafka 代理程式的存取或身分驗證協定。
         - 如果您的 Kafka 代理程式使用 SASL/PLAIN 身分驗證，請選擇 BASIC_AUTH。
         - 如果您的代理程式使用 SASL/SCRAM 身分驗證，請選擇其中一種 SASL_SCRAM 通訊協定。
         - 如果您要設定 mTLS 身分驗證，請選擇 CLIENT_CERTIFICATE_TLS_AUTH 通訊協定。
      ii. 若為 SASL/SCRAM 或 mTLS 身分驗證，請選擇包含 Kafka 叢集憑證的 Secrets Manager 機密金鑰。
   k. (選用) 若為 加密 ，如果您的 Kafka 代理程式使用私有憑證授權機構簽署的憑證，請選擇包含 Kafka 代理程式用於 TLS 加密的根憑證授權機構憑證的 Secrets Manager 機密。

      此設定適用於 SASL/SCRAM 或 SASL/PLAIN 的 TLS 加密，也適用於 mTLS 身分驗證。

   l. 若要建立處於停用狀態的觸發條件以進行測試 (建議做法)，請取消勾選 啟用觸發條件 。或者，若要立即啟用觸發條件，請選取 啟用觸發條件。

5. 若要建立觸發條件，請選擇 新增 。

新增自我管理 Kafka 叢集 (AWS CLI)

使用下列範例 AWS CLI 命令來建立和檢視 Lambda 函數的自我管理 Apache Kafka 觸發條件。

使用 SASL/SCRAM

如果 Kafka 使用者透過網際網路存取您的 Kafka 代理程式，請指定針對 SASL/SCRAM 身分驗證建立的 Secrets Manager 機密。下列範例使用 create-event-source-mapping AWS CLI 命令，將名為 的 Lambda 函數映射`my-kafka-function`至名為 的 Kafka 主題`AWSKafkaTopic`。

```
aws lambda create-event-source-mapping \
  --topics AWSKafkaTopic \
  --source-access-configuration Type=SASL_SCRAM_512_AUTH,URI=arn:aws:secretsmanager:us-
east-1:111122223333:secret:MyBrokerSecretName \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

使用 VPC

如果只有您 VPC 內的 Kafka 使用者可存取您的 Kafka 代理程式，則必須指定您的 VPC、子網路和 VPC 安全群組。下列範例使用 create-event-source-mapping AWS CLI 命令，將名為 的 Lambda 函數映射`my-kafka-function`至名為 的 Kafka 主題`AWSKafkaTopic`。

```
aws lambda create-event-source-mapping \
  --topics AWSKafkaTopic \
  --source-access-configuration '[{"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0011001100"}, {"Type": "VPC_SUBNET", "URI":
 "subnet:subnet-0022002200"}, {"Type": "VPC_SECURITY_GROUP", "URI":
 "security_group:sg-0123456789"}]' \
  --function-name arn:aws:lambda:us-east-1:111122223333:function:my-kafka-function \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc3.xyz.com:9092", "abc2.xyz.com:9092"]}}'
```

使用 檢視狀態 AWS CLI

下列範例使用 get-event-source-mapping AWS CLI 命令來描述您建立的事件來源映射狀態。

```
aws lambda get-event-source-mapping
```

```
    --uuid dh38738e-992b-343a-1077-3478934hjkfd7
```

Lambda 中所有自我管理的 Apache Kafka 事件來源組態參數

所有 Lambda 事件來源類型都會共用相同的 CreateEventSourceMapping 和 UpdateEventSourceMapping API 操作。不過，僅有部分參數適用於自我管理的 Apache Kafka，如下表所示。

參數	必要	預設	備註
BatchSize	否	100	上限：10,000
DestinationConfig	N	N/A	the section called “保留失敗的調用”
已啟用	N	True	
FilterCriteria	N	N/A	控制 Lambda 將哪些事件傳送至您的函數
FunctionName	是	N/A	
KMSKeyArn	N	N/A	the section called “篩選條件加密”
MaximumBatchingWindowInSeconds	N	500 毫秒	批次處理行為
ProvisionedPollersConfig	N	`MinimumPollers` ：若未指定，預設值為 1。 `MaximumPollers` ：若未指定，預設值為 200。 `PollerGroupName` ：不適用	the section called “佈建模式”

參數	必要	預設	備註
SelfManagedEventSource	Y	N/A	Kafka 代理程式清單。只能在建立時進行設定
SelfManagedKafkaEventSourceConfig	N	包含預設為一個唯一值的 ConsumerGroupId 欄位。	只能在建立時進行設定
SourceAccessConfigurations	N	沒有憑證	叢集的 VPC 資訊或身分驗證憑證 對於 SASL_PLAIN，設定為 BASIC_AUTH
StartingPosition	Y	N/A	AT_TIMESTAMP、TRIM_HORIZON 或 LATEST 只能在建立時進行設定
StartingPositionTimestamp	N	N/A	StartingPosition 設定為 AT_TIMESTAMP 時需要
Tags (標籤)	N	N/A	the section called “事件來源映射標籤”
主題	Y	N/A	主題名稱 只能在建立時進行設定

> **Note**
>
> 當您指定 時`PollerGroupName`，相同 Amazon VPC 中的多個 ESMs 可以共用事件輪詢單元 (EPU) 容量。您可以使用此選項來最佳化 ESM 的佈建模式成本。ESM 分組的需求：

- ESMs 必須位於相同的 Amazon VPC 內
- 每個輪詢器群組最多 100 ESMs
- 群組中所有 ESMs 的彙總輪詢器上限不能超過 2000

您可以更新 `PollerGroupName`，將 ESM 移至不同的群組，或將 `PollerGroupName`設定為空字串 (`""`)，從群組中移除 ESM。

Lambda 中的 Apache Kafka 事件輪詢器擴展模式

對於 Amazon MSK 和自我管理的 Apache Kafka 事件來源映射，可從以下兩種事件輪詢器擴展模式中任擇其一：

- 隨需模式 (預設)
- 佈建模式

隨需模式 (預設)

當您最初建立 Kafka 事件來源時，Lambda 會配置預設數量的事件輪詢器來處理 Kafka 主題中的所有分割區。Lambda 會根據訊息負載自動增加或減少事件輪詢器數量。

每 1 分鐘，Lambda 會評估主題中所有分割區的偏移延遲。如果偏移延遲太高，則表示分割區接收訊息的速度比 Lambda 處理訊息的速度更快。如有必要，Lambda 會新增或移除主題的事件輪詢器。此新增或移除事件輪詢器的自動擴展程序會在評估後三分鐘內發生。

如果您的目標 Lambda 函數遭限流，則 Lambda 會減少事件輪詢器的數量。此動作可透過減少事件輪詢器可擷取和傳送至函數的訊息數量，減少函數的工作負載。

佈建模式

對於您需要微調事件來源映射輸送量的工作負載，可以使用佈建模式。在佈建模式中，可以定義佈建的事件輪詢器數量的下限和上限。這些佈建的事件輪詢器專用於您的事件來源映射，並且可以透過回應性自動擴展處理意外的訊息尖峰。我們建議您針對效能需求嚴格的 Kafka 工作負載使用佈建模式。

在 Lambda 中，事件輪詢器是具有不同事件來源類型之輸送量功能的運算單位。對於 Amazon MSK 和自我管理的 Apache Kafka，每個事件輪詢器最多可以處理 5 MB/秒的輸送量或最多 5 個並行調用。例如，如果您的事件來源產生平均承載 1 MB，且函數的平均持續時間為 1 秒，則單一 Kafka 事件輪詢器可以支援 5 MB/秒輸送量和 5 個並行 Lambda 調用（假設沒有承載轉換）。對於 Amazon SQS，每個事件輪詢器最多可以處理每秒 1 MB 的輸送量或最多 10 個並行調用。使用佈建模式會根據您的事件輪詢器用量產生額外費用。如需定價詳細資訊，請參閱 AWS Lambda 定價。

> **Note**
>
> 使用佈建模式時，您不需要建立 AWS PrivateLink VPC 端點，也不需要在網路組態中授予相關聯的許可。

在佈建模式中，事件輪詢器數目下限 (MinimumPollers) 的接受值範圍介於 1 到 200 之間，包括 1 和 200 在內。事件輪詢器數目上限 (MaximumPollers) 的接受值範圍介於 1 到 2,000 之間，包括 1 和 2,000 在內。MaximumPollers 必須大於或等於 MinimumPollers。此外，為了在分割區內維持有序處理，Lambda 將 MaximumPollers 限制為不超過主題中的分割區數量。

如需選擇適當的事件輪詢器數量下限值和上限值的詳細資訊，請參閱最佳實務。

您可以使用主控台或 Lambda API，為 Kafka 事件來源映射設定佈建模式。

為現有事件來源映射設定佈建模式 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇具有要設定佈建模式之事件來源映射的函式。
3. 選擇組態，然後選擇觸發條件。
4. 選擇要設定佈建模式的事件來源映射，然後選擇編輯。
5. 在佈建模式欄位中，選取設定。
   - 針對事件輪詢器下限，輸入介於 1 到 200 之間的值。如果您沒有指定值，則 Lambda 會選擇預設值 1。
   - 針對事件輪詢器上限，輸入介於 1 到 2,000 之間的值。此值必須大於或等於事件輪詢器下限值。如果您沒有指定值，則 Lambda 會選擇預設值 200。
6. 選擇儲存。

您可以使用 EventSourceMappingConfiguration 中的 ProvisionedPollerConfig 物件，以程式設計方式設定佈建模式。例如，下列 UpdateEventSourceMapping CLI 命令會將 MinimumPollers 值設定為 5，將 MaximumPollers 值設定為 100。

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{"MinimumPollers": 5, "MaximumPollers": 100}'
```

設定佈建模式後，可以透過監控 ProvisionedPollers 指標來觀察工作負載的事件輪詢器使用情況。如需詳細資訊，請參閱the section called “事件來源映射指標”。

若要停用佈建模式並返回預設 (隨需) 模式，您可以使用下列 UpdateEventSourceMapping CLI 命令：

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
```

```
    --provisioned-poller-config '{}'
```

進階錯誤處理和效能功能

對於已啟用佈建模式的 Kafka 事件來源映射，您可以設定其他功能來改善錯誤處理和效能：

- 重試組態 – 控制 Lambda 如何處理失敗的記錄，以及重試次數上限、記錄存留期限制、批次分割和部分批次回應。
- Kafka 失敗的目的地 – 將失敗的記錄傳送至 Kafka 主題，以供日後處理或分析。

使用佈建模式時的最佳實務和考量

事件來源映射的事件輪詢器上限和下限的最佳組態取決於應用程式的效能需求。我們建議您從預設的事件輪詢器下限開始，以基準化效能設定檔。根據觀察到的訊息處理模式和所需的效能設定檔來調整您的組態。

對於具有尖峰流量和嚴格效能需求的工作負載，請增加事件輪詢器下限，以處理訊息的突然激增。若要判斷所需的事件輪詢器下限，請考慮工作負載的每秒訊息數和平均承載大小，並使用單一事件輪詢器的輸送容量 (至多 5 MBps) 做為參考。

為了在分割區內維持有序處理，Lambda 將事件輪詢器的上限限制為主題中的分割區數量。此外，事件來源映射可以擴展到的事件輪詢器上限取決於函數的並行設定。

啟用佈建模式時，請更新您的網路設定以移除 AWS PrivateLink VPC 端點和相關聯的許可。

佈建模式的成本最佳化

佈建模式定價

佈建模式會根據佈建的最低事件輪詢器，以及在自動擴展期間使用的事件輪詢器來收費。使用稱為事件輪詢單位 (EPU) 的計費單位來計算費用。您支付使用的 EPUs數量和持續時間，以 Event-Poller-Unit-hours 為單位。您可以將佈建模式與效能敏感應用程式的單一 ESM 搭配使用，也可以將相同 VPC 中的多個 ESMs 分組，以共用 EPU 容量和成本。我們將深入了解可協助您最佳化佈建模式成本的兩項功能。如需定價詳細資訊，請參閱 AWS Lambda 定價。

增強的 EPU 使用率

每個 EPU 最多支援 20 MB/s 的事件輪詢輸送量容量，並支援預設 10 個事件輪詢器。當您透過設定最低和最高輪詢器為 Kafka ESM 建立佈建模式時，它會根據每個 EPUs 的預設 10 個事件輪詢器，使用最低輪詢器編號來佈建 EPU。不過，每個事件輪詢器可以獨立擴展，以支援高達 5 MB/s 的輸送量容

量，這可能需要特定 EPU 上較低的事件輪詢器密度，並且可以觸發 EPUs 的擴展。EPU 上配置的事件輪詢器數量取決於每個事件輪詢器耗用的運算容量。這種增強型 EPU 使用率的方法可讓具有不同輸送量需求的事件輪詢器有效地利用 EPU 容量，從而降低所有 ESMs 的成本。

ESM 分組

若要進一步最佳化佈建模式成本，您可以將多個 Kafka ESMs 分組以共用 EPU 容量。透過 ESM 分組和增強的 EPU 使用率，相較於在單一 ESM 模式下執行，您可以降低低輸送量工作負載的佈建模式成本高達 90%。所有需要少於 1 個 EPU 容量ESMs ESM 都將受益於 ESM 分組。這類 ESMs通常只需要幾個最低事件輪詢器，即可支援其輸送量需求。此功能可讓您針對所有 Kafka 工作負載採用佈建模式，並受益於結構描述驗證、Avro/Protobuf 事件篩選、低延遲調用和僅在佈建模式下可用的增強型錯誤處理等功能。

當您為相同 Amazon VPC 中的多個 ESMs 設定具有相同值的`PollerGroupName`參數時，這些 ESMs 會共用 EPU 資源，而不是每個需要專用 EPU 容量。每個輪詢器群組最多可以分組 100 ESMs，且群組中所有 ESMs 的彙總輪詢器上限不能超過 2000 個。

設定 ESM 分組 （主控台）

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函數。
3. 選擇組態，然後觸發條件。
4. 建立新的 Kafka 事件來源映射或編輯現有的事件來源映射時，請選取佈建模式下的設定。
5. 針對事件輪詢器下限，輸入介於 1 到 200 之間的值。
6. 針對事件輪詢器上限，輸入介於 1 到 2,000 之間的值。
7. 針對輪詢器群組名稱，輸入群組的識別符。針對您要分組的其他 ESMs 使用相同的名稱。
8. 選擇儲存。

設定 ESM 分組 (AWS CLI)

下列範例會使用名為 的輪詢器群組建立 ESM`production-app-group`：

```
aws lambda create-event-source-mapping \
  --function-name myFunction1 \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/MyCluster/abcd1234 \
  --topics topic1 \
  --starting-position LATEST \
  --provisioned-poller-config '{
```

```
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": "production-app-group"
  }'
```

若要將另一個 ESM 新增至相同的群組（共用 EPU 容量），請使用相同的 PollerGroupName：

```
aws lambda create-event-source-mapping \
  --function-name myFunction2 \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/MyCluster/abcd1234 \
  --topics topic2 \
  --starting-position LATEST \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": "production-app-group"
  }'
```

> **Note**
>
> 您可以更新 PollerGroupName，將 ESM 移至不同的群組，或傳遞 的空字串 ("")，從群組中移除 ESMPollerGroupName：

```
# Move ESM to a different group
aws lambda update-event-source-mapping \
  --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": "new-group-name"
  }'

# Remove ESM from group (use dedicated resources)
aws lambda update-event-source-mapping \
  --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
  --provisioned-poller-config '{
    "MinimumPollers": 1,
    "MaximumPollers": 10,
    "PollerGroupName": ""
  }'
```

分組策略考量事項

- 應用程式界限 – 屬於相同應用程式或服務的群組 ESMs，以實現更好的成本分配和管理。考慮使用命名慣例，例如 `app-name-environment`（例如 `order-processor-prod`）。
- 流量模式 – 避免將具有高輸送量和尖峰流量模式ESMs 分組，因為這可能會導致資源爭用。
- 爆量半徑 – 如果共用基礎設施遇到問題，請考慮影響。相同群組中的所有 ESMs都會受到共用資源限制的影響。對於關鍵任務工作負載，您可能想要使用不同的群組或專用 ESMs。

成本最佳化範例

假設您擁有 10 ESMs，每個 ESM 設定了 1 個事件輪詢器，輸送量低於 2 MB/s：

沒有分組：

- 每個 ESM 都需要自己的 EPU
- 所需的 EPUs 總數：10
- 每個 EPU 的成本：美國東部（維吉尼亞北部）每小時 0.185 美元
- 每月 EPU 成本 (720 小時）：10 × 720 × $0.185 = $1，332

使用分組：

- 所有 10 ESMs 共用 EPU 容量
- 10 個事件輪詢器適合 1 個 EPU（每個 EPU 支援有新的 10 個輪詢器）
- 所需的 EPUs 總數：1
- 每月 EPU 成本 (720 小時）：1 × 720 × $0.185 = $133.20
- 節省成本：90%（每月節省 1，198.80 美元）

Lambda 中的 Apache Kafka 輪詢與串流開始位置

StartingPosition 參數會告知 Lambda 何時開始從 Amazon MSK 或自我管理的 Apache Kafka 串流中讀取訊息。有三種選項可供選擇：

- 最新：Lambda 將從 Kafka 主題中最新一筆記錄之後開始讀取。
- 水平修剪：Lambda 將從 Kafka 主題中最後一筆未修剪的記錄開始讀取。這也是主題中最舊的記錄。
- 在時間戳記：Lambda 將從由時間戳記 (以 Unix 時間秒為單位) 定義的位置開始讀取。使用 StartingPositionTimestamp 參數來指定時間戳記。

在建立或更新事件來源映射期間，串流輪詢最終會一致：

- 在建立事件來源映射期間，從串流開始輪詢事件可能需要幾分鐘時間。
- 在更新事件來源映射期間，從串流停止並重新開始輪詢事件最多可能需要 90 秒。

這種行為表示，如果您將 LATEST 指定為串流的開始位置，事件來源映射可能會在建立或更新期間遺漏事件。為確保不會遺漏任何事件，需指定 TRIM_HORIZON 或 AT_TIMESTAMP。

Lambda 中可自訂的取用者群組 ID

將 Amazon MSK 或自我管理的 Apache Kafka 設定為事件來源時，可指定取用者群組 ID。此取用者群組 ID 是您希望 Lambda 函數加入之 Kafka 取用者群組的現有識別符。您可以使用此功能將任何進行中的 Kafka 記錄處理設定從其他取用者無縫遷移至 Lambda。

Kafka 會將訊息分發給取用者群組中的所有取用者。若您指定的取用者群組 ID 已有其他作用中取用者，Lambda 將僅能從 Kafka 主題接收部分訊息。如果您希望 Lambda 處理主題中的所有訊息，請關閉該取用者群組中的任何其他取用者。

此外，如果您指定取用者群組 ID，且 Kafka 找到具有相同 ID 的有效現有取用者群組，則 Lambda 會忽略用於事件來源映射的 StartingPosition。相反的，Lambda 會根據取用者群組的承諾偏移量開始處理記錄。如果您指定取用者群組 ID，但 Kafka 找不到現有的取用者群組，則 Lambda 會使用指定的 `StartingPosition` 來設定事件來源。

您指定的取用者群組 ID 在所有 Kafka 事件來源中必須是唯一的。使用指定的取用者群組 ID 建立 Kafka 事件來源映射之後，您就無法更新此值。

從 Amazon MSK 和自我管理的 Apache Kafka 事件來源篩選事件

您可以使用事件篩選來控制 Lambda 將哪些記錄從串流或佇列中傳送至函數。如需事件篩選運作方式的一般資訊，請參閱the section called “事件篩選”。

> **Note**
>
> Amazon MSK 和自我管理的 Apache Kafka 事件來源映射，僅支援依據 `value` 鍵進行篩選。

主題

- Kafka 事件篩選基礎知識

Kafka 事件篩選基礎知識

假設生產者正在將訊息寫入 Kafka 叢集中的某個主題，其格式可能為有效的 JSON 或純文字字串。範例記錄如下所示，訊息在 `value` 欄位中會轉換為 Base64 編碼字串。

```
{
    "mytopic-0":[
        {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "value":"SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
            "headers":[]
        }
    ]
}
```

假設 Apache Kafka 生產者正在以下列 JSON 格式將訊息寫入到您的主題。

```
{
    "device_ID": "AB1234",
    "session":{
        "start_time": "yyyy-mm-ddThh:mm:ss",
        "duration": 162
```

```
    }
}
```

您可以使用 value 索引鍵來篩選記錄。假設您只想篩選 device_ID 以字母 AB 開頭的記錄。FilterCriteria 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : { \"device_ID\" : [ { \"prefix\": \"AB\" } ] } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 Pattern 在純文字 JSON 中擴展的值。

```
{
    "value": {
        "device_ID": [ { "prefix": "AB" } ]
      }
}
```

您可以使用主控台、AWS CLI 或 AWS SAM 範本新增篩選條件。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 來建立具有這些篩選條件標準的新事件來源映射，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
```

```
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : { \"device_ID\" :
 [ { \"prefix\":  \"AB\" } ] } }"}]}'
```

AWS SAM

若要使用 AWS SAM 新增此篩選條件，請將下列程式碼片段新增到事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : { "device_ID" : [ { "prefix":  "AB" } ] } }'
```

使用 Kafka 時，您也可以篩選純文字字串格式的訊息記錄。假設您想忽略字串為「錯誤」的訊息。`FilterCriteria` 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"value\" : [ { \"anything-but\": [ \"error\" ] } ] }"
        }
    ]
}
```

補充說明，此處是篩選條件的 `Pattern` 在純文字 JSON 中擴展的值。

```
{
    "value": [
        {
        "anything-but": [ "error" ]
        }
    ]
}
```

您可以使用主控台、AWS CLI 或 AWS SAM 範本新增篩選條件。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "value" : [ { "anything-but": [ "error" ] } ] }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 來建立具有這些篩選條件標準的新事件來源映射，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:kafka:us-east-2:123456789012:cluster/my-cluster/
b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"value\" : [ { \"anything-but\":
 [ \"error\" ] } ] }"}]}'
```

AWS SAM

若要使用 AWS SAM 新增此篩選條件，請將下列程式碼片段新增到事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "value" : [ { "anything-but": [ "error" ] } ] }'
```

Kafka 訊息必須是 UTF-8 編碼的字串，其格式可為純文字字串或 JSON。這是因為 Lambda 會在套用篩選條件之前，將 Kafka 位元組陣列解碼成 UTF-8。如果您的訊息使用其他編碼方式 (例如 UTF-16 或 ASCII)，或者訊息格式與 `FilterCriteria` 格式不相符，則 Lambda 只會處理中繼資料篩選條件。下表摘要說明特定行為：

傳入訊息 格式	訊息屬性的篩選條件模式格式	產生的動作
純文字的字串	純文字的字串	根據您的篩選條件標準之 Lambda 篩選條件。
純文字的字串	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
純文字的字串	有效的 JSON	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	純文字的字串	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	有效的 JSON	根據您的篩選條件標準之 Lambda 篩選條件。
非 UTF-8 編碼字串	JSON、純字串或沒有模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。

在 Lambda 中將結構描述登錄檔與 Kafka 事件來源搭配使用

結構描述登錄檔可協助您定義並管理資料串流結構描述。結構描述定義資料記錄的結構和格式。在 Kafka 事件來源映射的情境中，您可以設定結構描述登錄檔，在 Kafka 訊息送達 Lambda 函式之前，依據預先定義的結構描述驗證其結構與格式。這會將資料治理層新增至應用程式，讓您能夠有效管理資料格式、確保結構描述合規，並透過事件篩選來最佳化成本。

此功能適用於所有程式設計語言，但請考慮以下要點：

- Powertools for Lambda 提供 Java、Python 及 TypeScript 的專屬支援，不僅能維護與現有 Kafka 開發模式的一致性，更可讓您直接存取業務物件，無需自訂反序列化程式碼
- 此功能僅適用於採用佈建模式的事件來源映射。結構描述登錄檔不支援隨需模式下的事件來源映射。如果您使用佈建模式且已設定結構描述登錄檔，將無法變更為隨需模式，除非您先移除結構描述登錄檔組態。如需詳細資訊，請參閱the section called “佈建模式”
- 每個事件來源映射 (ESM) 僅能設定一個結構描述登錄檔。將結構描述登錄檔與 Kafka 事件來源搭配使用，可能會增加 Lambda 事件輪詢單元 (EPU) 用量，此單元為佈建模式的一個定價維度。

主題

- 結構描述登錄檔選項
- Lambda 如何對 Kafka 訊息執行結構描述驗證
- 設定 Kafka 結構描述登錄檔
- Avro 與 Protobuf 的篩選
- 承載格式與反序列化行為
- 在 Lambda 函式中處理反序列化資料
- 結構描述登錄檔的身分驗證方法
- 結構描述登錄檔問題的錯誤處理與疑難排解

結構描述登錄檔選項

Lambda 支援下列結構描述登錄檔選項：

- AWS Glue 結構描述登錄檔
- Confluent Cloud 結構描述登錄檔
- 自我管理的 Confluent 結構描述登錄檔

結構描述登錄檔支援驗證採用下列資料格式的訊息：

- Apache Avro
- 通訊協定緩衝區 (Protobuf)
- JSON 結構描述 (JSON-SE)

若要使用結構描述登錄檔，請先確保事件來源映射處於佈建模式。當您使用結構描述登錄檔時，Lambda 會將結構描述的相關中繼資料新增至承載。如需詳細資訊，請參閱承載格式與反序列化行為。

Lambda 如何對 Kafka 訊息執行結構描述驗證

當您設定結構描述登錄檔時，Lambda 會為每個 Kafka 訊息執行下列步驟：

1. Lambda 會從叢集輪詢 Kafka 記錄。
2. Lambda 會根據結構描述登錄檔中的特定結構描述，驗證記錄中選定的訊息屬性。
   - 如果在登錄檔中找不到與訊息相關聯的結構描述，Lambda 會將訊息傳送至 DLQ，並標示原因代碼 `SCHEMA_NOT_FOUND`。
3. 為了驗證訊息，Lambda 會依據結構描述登錄檔組態對訊息進行反序列化。如果設定了事件篩選，Lambda 會接著根據設定的篩選條件執行篩選。
   - 如果反序列化失敗，Lambda 會將訊息傳送至 DLQ，並標示原因代碼 `DESERIALIZATION_ERROR`。如果未設定 DLQ，Lambda 會捨棄訊息。
4. 如果訊息透過結構描述登錄檔進行驗證，且未依篩選條件篩選出，Lambda 將會使用該訊息調用函式。

此功能旨在驗證已使用與結構描述登錄檔整合的 Kafka 用戶端所產生的訊息。建議將 Kafka 生產者設定為與結構描述登錄檔搭配使用，以便建立格式正確的訊息。

設定 Kafka 結構描述登錄檔

下列主控台步驟會將 Kafka 結構描述登錄檔組態新增至事件來源映射。

將 Kafka 結構描述登錄檔組態新增至事件來源映射 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇 Configuration (組態)。

3. 選擇觸發程序。
4. 選取要為其設定結構描述登錄檔的 Kafka 事件來源映射，然後選擇編輯。
5. 在事件輪詢器組態欄位中，選擇設定結構描述登錄檔。事件來源映射必須處於佈建模式，方可查看此選項。
6. 對於結構描述登錄 URI，輸入 AWS Glue 結構描述登錄檔的 ARN，或 Confluent Cloud 結構描述登錄檔的 HTTPS URL 或自我管理的 Confluent 結構描述登錄檔。
7. 下列組態步驟會告知 Lambda 如何存取結構描述登錄檔。如需詳細資訊，請參閱???。
   - 在存取組態類型欄位中，選擇 Lambda 用於存取結構描述登錄檔的身分驗證類型。
   - 在存取組態 URI 欄位中，輸入 Secrets Manager 秘密的 ARN，向結構描述登錄檔進行身分驗證 (如適用)。確保函式的執行角色包含正確的許可。
8. 僅當結構描述登錄檔由私有憑證認證機構 (CA) 或未納入 Lambda 信任存放區的憑證認證機構 (CA) 簽署時，加密欄位才適用。如適用，請提供私密金鑰，其中應包含結構描述登錄檔用於 TLS 加密的私有 CA 憑證。
9. 在事件記錄格式欄位中，選擇希望 Lambda 在完成結構描述驗證後將記錄交付給函式的方式。如需詳細資訊，請參閱承載格式範例。
   - 如果選擇 JSON，Lambda 會以標準 JSON 格式傳送您在下方「結構描述驗證屬性」中選取的屬性。對於您未選取的屬性，Lambda 會依原狀傳送這些屬性。
   - 如果選擇 SOURCE，Lambda 會以原始來源格式傳送您在下方「結構描述驗證屬性」中選取的屬性。
10. 在結構描述驗證屬性欄位中，選取您希望 Lambda 使用結構描述登錄檔進行驗證與反序列化的訊息屬性。必須至少選取 KEY 或 VALUE 其中一項。若您選擇 JSON 作為事件記錄格式，Lambda 還會在將選定的訊息屬性傳送至函式之前，對其進行反序列化。如需詳細資訊，請參閱承載格式與反序列化行為。
11. 選擇儲存。

您也可以使用 Lambda API 來建立或更新帶有結構描述登錄檔組態的事件來源映射。下列範例示範如何使用 設定 AWS Glue 或 Confluent 結構描述登錄檔 AWS CLI，其對應至 API 參考中的 UpdateEventSourceMapping 和 CreateEventSourceMapping API 操作： AWS Lambda

> **Important**
>
> 如果您使用 AWS CLI 或 `update-event-source-mapping` API 更新任何結構描述登錄檔組態欄位，則必須更新結構描述登錄檔組態的所有欄位。

Create Event Source Mapping

```
aws lambda create-event-source-mapping \
  --function-name my-schema-validator-function \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/
a1b2c3d4-5678-90ab-cdef-11111EXAMPLE \
  --topics my-kafka-topic \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=1 \
  --amazon-managed-kafka-event-source-mapping '{
      "SchemaRegistryConfig" : {
          "SchemaRegistryURI": "https://abcd-ef123.us-west-2.aws.confluent.cloud",
          "AccessConfigs": [{
              "Type": "BASIC_AUTH",
              "URI": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:secretName"
          }],
          "EventRecordFormat": "JSON",
          "SchemaValidationConfigs": [
          {
              "Attribute": "KEY"
          },
          {
              "Attribute": "VALUE"
          }]
      }
  }'
```

Update AWS Glue Schema Registry

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {
            "SchemaRegistryURI": "arn:aws:glue:us-east-1:123456789012:registry/
registryName",
            "EventRecordFormat": "JSON",
            "SchemaValidationConfigs": [
            {
                "Attribute": "KEY"
            },
            {
                "Attribute": "VALUE"
            }]
```

```
        }
    }'
```

Update Confluent Schema Registry with Authentication

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {
            "SchemaRegistryURI": "https://abcd-ef123.us-west-2.aws.confluent.cloud",
            "AccessConfigs": [{
                "Type": "BASIC_AUTH",
                "URI": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:secretName"
            }],
            "EventRecordFormat": "JSON",
            "SchemaValidationConfigs": [
            {
                "Attribute": "KEY"
            },
            {
                "Attribute": "VALUE"
            }]
        }
    }'
```

Update Confluent Schema Registry without Authentication

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {
            "SchemaRegistryURI": "https://abcd-ef123.us-west-2.aws.confluent.cloud",
            "EventRecordFormat": "JSON",
            "SchemaValidationConfigs": [
            {
                "Attribute": "KEY"
            },
            {
                "Attribute": "VALUE"
            }]
        }
    }'
```

Remove Schema Registry Configuration

若要從事件來源映射中移除結構描述登錄檔組態，可以使用 AWS Lambda API Reference 中的 CLI 命令 UpdateEventSourceMapping。

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --amazon-managed-kafka-event-source-mapping '{
        "SchemaRegistryConfig" : {}
    }'
```

Avro 與 Protobuf 的篩選

將 Avro 或 Protobuf 格式與結構描述登錄檔搭配使用時，即可將事件篩選功能套用至 Lambda 函式。篩選模式會在結構描述驗證後，套用至資料反序列化後的傳統 JSON 表示法。例如，使用定義產品詳細資訊 (包含價格) 的 Avro 結構描述，您可以根據價格值篩選訊息：

> **Note**
>
> 當 Avro 進行反序列化時，會轉換為標準 JSON 格式，也意味著無法直接轉換回 Avro 物件。如需轉換為 Avro 物件，請改用 SOURCE 格式。
> 在 Protobuf 反序列化過程中，產生 JSON 的欄位名稱與結構描述中定義的欄位名稱相符，而非如 Protobuf 常規作法般為駝峰式大小寫。在建立篩選模式時，請記住這一點。

```
aws lambda create-event-source-mapping \
    --function-name myAvroFunction \
    --topics myAvroTopic \
    --starting-position TRIM_HORIZON \
    --kafka-bootstrap-servers '["broker1:9092", "broker2:9092"]' \
    --schema-registry-config '{
        "SchemaRegistryURI": "arn:aws:glue:us-east-1:123456789012:registry/
myAvroRegistry",
        "EventRecordFormat": "JSON",
        "SchemaValidationConfigs": [
            {
                "Attribute": "VALUE"
            }
        ]
    }' \
```

```
    --filter-criteria '{
        "Filters": [
            {
                "Pattern": "{ \"value\" : { \"field_1\" : [\"value1\"], \"field_2\" :
 [\"value2\"] } }"
            }
        ]
    }'
```

在此範例中，篩選模式會分析 value 物件，將 field_1 中符合 "value1"、field_2 中符合 "value2" 的訊息進行比對。在 Lambda 將訊息從 Avro 格式轉換為 JSON 格式之後，篩選條件會針對反序列化後的資料進行評估。

如需有關事件篩選的詳細資訊，請參閱 Lambda 事件篩選。

承載格式與反序列化行為

使用結構描述登錄檔時，Lambda 會以類似於常規事件承載的格式將最終承載傳送至函式，但會包含部分額外欄位。這些額外欄位取決於 SchemaValidationConfigs 參數。對於您選取進行驗證的每個屬性 (鍵或值)，Lambda 會將相應的結構描述中繼資料新增至承載。

> **Note**
>
> 必須將 aws-lambda-java-events 更新至 3.16.0 版或更新版本，才能使用結構描述中繼資料欄位。

例如，若驗證 value 欄位，Lambda 會將名為 valueSchemaMetadata 的欄位新增至承載。同理，如果驗證 key 欄位，Lambda 會新增名為 keySchemaMetadata 的欄位。此中繼資料包含資料格式與驗證所用結構描述 ID 的相關資訊：

```
"valueSchemaMetadata": {
    "dataFormat": "AVRO",
    "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
}
```

EventRecordFormat 參數可以設定為 JSON 或 SOURCE，這決定了 Lambda 在將經過結構描述驗證的資料傳送至函式之前的處理方式。每個選項都提供不同的處理功能：

- JSON：Lambda 將經過驗證的屬性反序列化為標準 JSON 格式，便於資料在原生 JSON 支援的語言中直接使用。若無需保留原始二進位格式或無需使用產生的類別，此格式尤其適用。

- SOURCE：Lambda 會將資料的原始二進位格式保留為 Base64 編碼字串，允許直接轉換為 Avro 或 Protobuf 物件。當使用強型別語言或需要保持 Avro 或 Protobuf 結構描述的完整功能時，此格式至關重要。

基於上述格式特性與語言專屬考量，建議採用下列格式：

依據程式設計語言建議的格式

Language	Avro	Protobuf	JSON
Java	SOURCE	SOURCE	SOURCE
Python	JSON	JSON	JSON
NodeJS	JSON	JSON	JSON
.NET	SOURCE	SOURCE	SOURCE
其他	JSON	JSON	JSON

下列各節詳細介紹了這些格式，也為每種格式提供了承載範例。

JSON format (JSON 格式)

若選擇 `JSON` 作為 `EventRecordFormat`，Lambda 會驗證並反序列化您在 `SchemaValidationConfigs` 欄位中選取的訊息屬性 (`key` 及/或 `value` 屬性)。Lambda 會將這些選定的屬性，以其標準 JSON 表述的 Base64 編碼字串形式，傳遞至您的函式中。

> **Note**
>
> 當 Avro 進行反序列化時，會轉換為標準 JSON 格式，也意味著無法直接轉換回 Avro 物件。如需轉換為 Avro 物件，請改用 SOURCE 格式。
> 在 Protobuf 反序列化過程中，產生 JSON 的欄位名稱與結構描述中定義的欄位名稱相符，而非如 Protobuf 常規作法般為駝峰式大小寫。在建立篩選模式時，請記住這一點。

下列承載範例假設您選擇 `JSON` 作為 `EventRecordFormat`，同時選擇 `key` 與 `value` 屬性作為 `SchemaValidationConfigs`：

```
{
```

```
   "eventSource":"aws:kafka",
   "eventSourceArn":"arn:aws:kafka:us-east-1:123456789012:cluster/vpc-2priv-2pub/
a1b2c3d4-5678-90ab-cdef-EXAMPLE11111-1",
   "bootstrapServers":"b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
   "records":{
      "mytopic-0":[
         {
            "topic":"mytopic",
            "partition":0,
            "offset":15,
            "timestamp":1545084650987,
            "timestampType":"CREATE_TIME",
            "key":"abcDEFghiJKLmnoPQRstuVWXyz1234==", //Base64 encoded string of JSON
            "keySchemaMetadata": {
                "dataFormat": "AVRO",
                "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
            },
            "value":"abcDEFghiJKLmnoPQRstuVWXyz1234", //Base64 encoded string of JSON
            "valueSchemaMetadata": {
                "dataFormat": "AVRO",
                "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
            },
            "headers":[
               {
                  "headerKey":[
                     104,
                     101,
                     97,
                     100,
                     101,
                     114,
                     86,
                     97,
                     108,
                     117,
                     101
                  ]
               }
            ]
         }
      ]
   }
```

```
}
```

在此範例中：

- key 與 value 經反序列化後，皆以其 JSON 表述的 Base64 編碼字串形式呈現。
- Lambda 會包含 keySchemaMetadata 與 valueSchemaMetadata 中兩種屬性的結構描述中繼資料。
- 函式可以解碼 key 和 value 字串，以存取反序列化後的 JSON 資料。

對於非強型別語言，例如 Python 或 Node.js，建議採用 JSON 格式。這些語言原生支援將 JSON 轉換為物件。

來源格式

如果您選擇 SOURCE 作為 EventRecordFormat，Lambda 仍然會根據結構描述登錄檔驗證記錄，但會將原始二進位資料傳送至函式，而不進行反序列化。此二進位資料會以原始位元組資料的 Base64 編碼字串形式傳送，並移除生產者附加的中繼資料。因此，您可在函式程式碼中，將原始二進位資料直接轉換為 Avro 與 Protobuf 物件。我們建議將 Powertools 用於 AWS Lambda，這會還原序列化原始二進位資料，並直接為您提供 Avro 和 Protobuf 物件。

例如，若設定 Lambda 來驗證 key 與 value 屬性，但採用 SOURCE 格式，函式會收到如下承載：

```
{
    "eventSource": "aws:kafka",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/vpc-2priv-2pub/
a1b2c3d4-5678-90ab-cdef-EXAMPLE11111-1",
    "bootstrapServers": "b-2.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092,b-1.demo-cluster-1.a1bcde.c1.kafka.us-
east-1.amazonaws.com:9092",
    "records": {
        "mytopic-0": [
            {
                "topic": "mytopic",
                "partition": 0,
                "offset": 15,
                "timestamp": 1545084650987,
                "timestampType": "CREATE_TIME",
                "key": "abcDEFghiJKLmnoPQRstuVWXyz1234==", // Base64 encoded string of
 Original byte data, producer-appended metadata removed
                "keySchemaMetadata": {
```

```
                    "dataFormat": "AVRO",
                    "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
                },
                "value": "abcDEFghiJKLmnoPQRstuVWXyz1234==", // Base64 encoded string
 of Original byte data, producer-appended metadata removed
                "valueSchemaMetadata": {
                    "dataFormat": "AVRO",
                    "schemaId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
                },
                "headers": [
                    {
                        "headerKey": [
                            104,
                            101,
                            97,
                            100,
                            101,
                            114,
                            86,
                            97,
                            108,
                            117,
                            101
                        ]
                    }
                ]
            }
        ]
    }
}
```

在此範例中：

- key 和 value 都包含原始二進位資料作為 Base64 編碼字串。
- 函式需要使用適當的程式庫來處理反序列化。

若您使用 Avro 產生的物件或 Protobuf 產生的物件 (特別是在 Java 函式中)，則建議將 EventRecordFormat 設定為 SOURCE。這是因為 Java 屬於強型別語言，需針對 Avro 與 Protobuf 格式使用特定的反序列化。在函式程式碼中，您可以使用偏好的 Avro 或 Protobuf 程式庫將資料反序列化。

在 Lambda 函式中處理反序列化資料

Powertools for AWS Lambda 可協助您根據您使用的格式，將函數程式碼中的 Kafka 記錄還原序列化。此公用程式透過處理資料轉換並提供即用型物件，簡化了 Kafka 記錄的處理流程。

若要在函數 AWS Lambda 中使用 Powertools for ，您需要在建置 Lambda 函數時新增 Powertools AWS Lambda 做為 layer 或包含它做為相依性。如需設定說明和詳細資訊，請參閱 Powertools for AWS Lambda documentation for your preferred language：

- Powertools for AWS Lambda (Java)
- Powertools for AWS Lambda (Python)
- Powertools for AWS Lambda (TypeScript)
- Powertools for AWS Lambda (.NET)

> **Note**
>
> 使用結構描述登錄檔整合時，可選擇 SOURCE 或 JSON 格式。每個選項都支援不同的序列化格式，如下所示：
>
> | 格式 | 支援 |
> | --- | --- |
> | SOURCE | Avro 與 Protobuf (透過 Lambda 結構描述登錄檔整合) |
> | JSON | JSON 資料 |

使用 SOURCE或 JSON 格式時，您可以使用 Powertools for AWS 協助還原序列化函數程式碼中的資料。以下是如何處理不同資料格式的範例：

AVRO

Java 範例：

```
package org.demo.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
```

```
import org.demo.kafka.avro.AvroProduct;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import software.amazon.lambda.powertools.kafka.Deserialization;
import software.amazon.lambda.powertools.kafka.DeserializationType;
import software.amazon.lambda.powertools.logging.Logging;

public class AvroDeserializationFunction implements
 RequestHandler<ConsumerRecords<String, AvroProduct>, String> {

    private static final Logger LOGGER =
 LoggerFactory.getLogger(AvroDeserializationFunction.class);

    @Override
    @Logging
    @Deserialization(type = DeserializationType.KAFKA_AVRO)
    public String handleRequest(ConsumerRecords<String, AvroProduct> records,
 Context context) {
        for (ConsumerRecord<String, AvroProduct> consumerRecord : records) {
            LOGGER.info("ConsumerRecord: {}", consumerRecord);

            AvroProduct product = consumerRecord.value();
            LOGGER.info("AvroProduct: {}", product);

            String key = consumerRecord.key();
            LOGGER.info("Key: {}", key);
        }

        return "OK";
    }

}
```

Python 範例：

```
from aws_lambda_powertools.utilities.kafka_consumer.kafka_consumer import
 kafka_consumer
from aws_lambda_powertools.utilities.kafka_consumer.schema_config import
 SchemaConfig
```

```
from aws_lambda_powertools.utilities.kafka_consumer.consumer_records import
 ConsumerRecords

from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger

logger = Logger(service="kafkaConsumerPowertools")

value_schema_str = open("customer_profile.avsc", "r").read()

schema_config = SchemaConfig(
value_schema_type="AVRO",
value_schema=value_schema_str)

@kafka_consumer(schema_config=schema_config)
def lambda_handler(event: ConsumerRecords, context:LambdaContext):

  for record in event.records:
      value = record.value
      logger.info(f"Received value: {value}")
```

TypeScript 範例：

```
import { kafkaConsumer } from '@aws-lambda-powertools/kafka';

import type { ConsumerRecords } from '@aws-lambda-powertools/kafka/types';
import { Logger } from '@aws-lambda-powertools/logger';
import type { Context } from 'aws-lambda';

const logger = new Logger();

type Value = {
    id: number;
    name: string;
    price: number;
};

const schema = '{
    "type": "record",
    "name": "Product",
    "fields": [
        { "name": "id", "type": "int" },
        { "name": "name", "type": "string" },
```

```
        { "name": "price", "type": "double" }
    ]
}';

export const handler = kafkaConsumer<string, Value>(
    (event: ConsumerRecords<string, Value>, _context: Context) => {
        for (const record of event.records) {
            logger.info(Processing record with key: ${record.key});
            logger.info(Record value: ${JSON.stringify(record.value)});
            // You can add more processing logic here
        }
    },
    {
        value: {
            type: 'avro',
            schema: schema,
        },
    }
);
```

.NET 範例：

```
using Amazon.Lambda.Core;
using AWS.Lambda.Powertools.Kafka;
using AWS.Lambda.Powertools.Kafka.Avro;
using AWS.Lambda.Powertools.Logging;
using Com.Example;

// Assembly attribute to enable the Lambda function's Kafka event to be converted
 into a .NET class.
[assembly: LambdaSerializer(typeof(PowertoolsKafkaAvroSerializer))]

namespace ProtoBufClassLibrary;

public class Function
{
    public string FunctionHandler(ConsumerRecords<string, CustomerProfile> records,
 ILambdaContext context)
    {
        foreach (var record in records)
        {
            Logger.LogInformation("Processing messagem from topic: {topic}",
 record.Topic);
```

```
            Logger.LogInformation("Partition: {partition}, Offset: {offset}",
 record.Partition, record.Offset);
            Logger.LogInformation("Produced at: {timestamp}", record.Timestamp);

            foreach (var header in record.Headers.DecodedValues())
            {
                Logger.LogInformation($"{header.Key}: {header.Value}");
            }

            Logger.LogInformation("Processing order for: {fullName}",
 record.Value.FullName);
        }

        return "Processed " + records.Count() + " records";
    }
}
```

PROTOBUF

Java 範例：

```
package org.demo.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.demo.kafka.protobuf.ProtobufProduct;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

import software.amazon.lambda.powertools.kafka.Deserialization;
import software.amazon.lambda.powertools.kafka.DeserializationType;
import software.amazon.lambda.powertools.logging.Logging;

public class ProtobufDeserializationFunction
        implements RequestHandler<ConsumerRecords<String, ProtobufProduct>, String>
 {

    private static final Logger LOGGER =
 LoggerFactory.getLogger(ProtobufDeserializationFunction.class);

    @Override
```

```
    @Logging
    @Deserialization(type = DeserializationType.KAFKA_PROTOBUF)
    public String handleRequest(ConsumerRecords<String, ProtobufProduct> records,
 Context context) {
        for (ConsumerRecord<String, ProtobufProduct> consumerRecord : records) {
            LOGGER.info("ConsumerRecord: {}", consumerRecord);

            ProtobufProduct product = consumerRecord.value();
            LOGGER.info("ProtobufProduct: {}", product);

            String key = consumerRecord.key();
            LOGGER.info("Key: {}", key);
        }

        return "OK";
    }

}
```

Python 範例：

```
from aws_lambda_powertools.utilities.kafka_consumer.kafka_consumer import
 kafka_consumer

from aws_lambda_powertools.utilities.kafka_consumer.schema_config import
 SchemaConfig
from aws_lambda_powertools.utilities.kafka_consumer.consumer_records import
 ConsumerRecords

from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger

from user_pb2 import User # protobuf generated class

logger = Logger(service="kafkaConsumerPowertools")

schema_config = SchemaConfig(
value_schema_type="PROTOBUF",
value_schema=User)

@kafka_consumer(schema_config=schema_config)
def lambda_handler(event: ConsumerRecords, context:LambdaContext):
```

```
  for record in event.records:
      value = record.value
      logger.info(f"Received value: {value}")
```

TypeScript 範例：

```
import { kafkaConsumer } from '@aws-lambda-powertools/kafka';
import type { ConsumerRecords } from '@aws-lambda-powertools/kafka/types';
import { Logger } from '@aws-lambda-powertools/logger';
import type { Context } from 'aws-lambda';
import { Product } from './product.generated.js';

const logger = new Logger();

type Value = {
    id: number;
    name: string;
    price: number;
};

export const handler = kafkaConsumer<string, Value>(
    (event: ConsumerRecords<string, Value>, _context: Context) => {
        for (const record of event.records) {
            logger.info(Processing record with key: ${record.key});
            logger.info(Record value: ${JSON.stringify(record.value)});
        }
    },
    {
        value: {
            type: 'protobuf',
            schema: Product,
        },
    }
);
```

.NET 範例：

```
using Amazon.Lambda.Core;
using AWS.Lambda.Powertools.Kafka;
using AWS.Lambda.Powertools.Kafka.Protobuf;
using AWS.Lambda.Powertools.Logging;
using Com.Example;
```

```
// Assembly attribute to enable the Lambda function's Kafka event to be converted
 into a .NET class.
[assembly: LambdaSerializer(typeof(PowertoolsKafkaProtobufSerializer))]

namespace ProtoBufClassLibrary;

public class Function
{
    public string FunctionHandler(ConsumerRecords<string, CustomerProfile> records,
 ILambdaContext context)
    {
        foreach (var record in records)
        {
            Logger.LogInformation("Processing messagem from topic: {topic}",
 record.Topic);
            Logger.LogInformation("Partition: {partition}, Offset: {offset}",
 record.Partition, record.Offset);
            Logger.LogInformation("Produced at: {timestamp}", record.Timestamp);

            foreach (var header in record.Headers.DecodedValues())
            {
                Logger.LogInformation($"{header.Key}: {header.Value}");
            }

            Logger.LogInformation("Processing order for: {fullName}",
 record.Value.FullName);
        }

        return "Processed " + records.Count() + " records";
    }
}
```

JSON

Java 範例：

```
package org.demo.kafka;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.amazonaws.services.lambda.runtime.Context;
```

```
import com.amazonaws.services.lambda.runtime.RequestHandler;

import software.amazon.lambda.powertools.kafka.Deserialization;
import software.amazon.lambda.powertools.kafka.DeserializationType;
import software.amazon.lambda.powertools.logging.Logging;

public class JsonDeserializationFunction implements
 RequestHandler<ConsumerRecords<String, Product>, String> {

    private static final Logger LOGGER =
 LoggerFactory.getLogger(JsonDeserializationFunction.class);

    @Override
    @Logging
    @Deserialization(type = DeserializationType.KAFKA_JSON)
    public String handleRequest(ConsumerRecords<String, Product> consumerRecords,
 Context context) {
        for (ConsumerRecord<String, Product> consumerRecord : consumerRecords) {
            LOGGER.info("ConsumerRecord: {}", consumerRecord);

            Product product = consumerRecord.value();
            LOGGER.info("Product: {}", product);

            String key = consumerRecord.key();
            LOGGER.info("Key: {}", key);
        }

        return "OK";
    }
}
```

Python 範例：

```
from aws_lambda_powertools.utilities.kafka_consumer.kafka_consumer import
 kafka_consumer

from aws_lambda_powertools.utilities.kafka_consumer.schema_config import
 SchemaConfig
from aws_lambda_powertools.utilities.kafka_consumer.consumer_records import
 ConsumerRecords

from aws_lambda_powertools.utilities.typing import LambdaContext
from aws_lambda_powertools import Logger
```

```
logger = Logger(service="kafkaConsumerPowertools")

schema_config = SchemaConfig(value_schema_type="JSON")

@kafka_consumer(schema_config=schema_config)
def lambda_handler(event: ConsumerRecords, context:LambdaContext):

  for record in event.records:
      value = record.value
      logger.info(f"Received value: {value}")
```

TypeScript 範例：

```
import { kafkaConsumer } from '@aws-lambda-powertools/kafka';
import type { ConsumerRecords } from '@aws-lambda-powertools/kafka/types';
import { Logger } from '@aws-lambda-powertools/logger';
import type { Context } from 'aws-lambda';

const logger = new Logger();

type Value = {
    id: number;
    name: string;
    price: number;
};

export const handler = kafkaConsumer<string, Value>(
    (event: ConsumerRecords<string, Value>, _context: Context) => {
        for (const record of event.records) {
            logger.info(Processing record with key: ${record.key});
            logger.info(Record value: ${JSON.stringify(record.value)});
            // You can add more processing logic here
        }
    },
    {
        value: {
            type: 'json',
        },
    }
);
```

.NET 範例：

```
using Amazon.Lambda.Core;
using AWS.Lambda.Powertools.Kafka;
using AWS.Lambda.Powertools.Kafka.Json;
using AWS.Lambda.Powertools.Logging;
using Com.Example;

// Assembly attribute to enable the Lambda function's Kafka event to be converted
 into a .NET class.
[assembly: LambdaSerializer(typeof(PowertoolsKafkaJsonSerializer))]

namespace JsonClassLibrary;

public class Function
{
    public string FunctionHandler(ConsumerRecords<string, CustomerProfile> records,
 ILambdaContext context)
    {
        foreach (var record in records)
        {
            Logger.LogInformation("Processing messagem from topic: {topic}",
 record.Topic);
            Logger.LogInformation("Partition: {partition}, Offset: {offset}",
 record.Partition, record.Offset);
            Logger.LogInformation("Produced at: {timestamp}", record.Timestamp);

            foreach (var header in record.Headers.DecodedValues())
            {
                Logger.LogInformation($"{header.Key}: {header.Value}");
            }

            Logger.LogInformation("Processing order for: {fullName}",
 record.Value.FullName);
        }

        return "Processed " + records.Count() + " records";
    }
}
```

結構描述登錄檔的身分驗證方法

若要使用結構描述登錄檔，Lambda 需要能夠對其進行安全存取。如果您使用 AWS Glue 結構描述登錄檔，Lambda 依賴 IAM 身分驗證。這表示函數的執行角色必須具有下列許可，才能存取 AWS Glue 登錄檔：

- AWS Glue Web API Reference 中的 GetRegistry
- AWS Glue Web API Reference 中的 GetSchemaVersion

必要 IAM 政策範例：

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "glue:GetRegistry",
                "glue:GetSchemaVersion"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

> **Note**
>
> 對於 AWS Glue 結構描述登錄檔，如果您`AccessConfigs`提供 AWS Glue 登錄檔，Lambda 將傳回驗證例外狀況。

若您使用 Confluent 結構描述登錄檔，可為 KafkaSchemaRegistryAccessConfig 物件的 `Type` 參數選擇以下三種身分驗證方法中一種：

- BASIC_AUTH：Lambda 會透過使用者名稱與密碼，或 API 金鑰與 API 秘密身分驗證方式存取登錄庫。如果選擇此選項，請在 URI 欄位中提供包含憑證的 Secrets Manager ARN。
- CLIENT_CERTIFICATE_TLS_AUTH：Lambda 透過用戶端憑證進行雙向 TLS 身分驗證。若要使用此選項，Lambda 需要同時存取憑證和私有金鑰。在 URI 欄位中提供包含這些憑證的 Secrets Manager ARN。
- NO_AUTH：公有 CA 憑證必須由位於 Lambda 信任存放區中的憑證認證機構 (CA) 簽署。對於私有 CA/自我簽署憑證，您需要設定伺服器根 CA 憑證。若要使用此選項，請省略 `AccessConfigs` 參數。

此外，如果 Lambda 需要存取私有 CA 憑證來驗證結構描述登錄檔的 TLS 憑證，請選擇 `SERVER_ROOT_CA_CERT` 作為 `Type`，並在 URI 欄位中提供憑證對應的 Secrets Manager ARN。

> **Note**
>
> 若要在主控台中設定 `SERVER_ROOT_CA_CERT` 選項，需要在加密欄位中提供包含該憑證的秘密 ARN。

結構描述登錄檔的驗證組態，需獨立於 Kafka 叢集的任何驗證設定。即使兩者採用相似的驗證方法，仍須分別進行設定。

結構描述登錄檔問題的錯誤處理與疑難排解

在將結構描述登錄檔與 Amazon MSK 事件來源搭配使用時，可能會遭遇各種錯誤。本節提供了常見問題及其解決方法的指導。

組態錯誤

這些錯誤通常發生在設定結構描述登錄檔組態時。

需要佈建模式

錯誤訊息：`SchemaRegistryConfig is only available for Provisioned Mode. To configure Schema Registry, please enable Provisioned Mode by specifying MinimumPollers in ProvisionedPollerConfig.`

解決方案：透過在 `ProvisionedPollerConfig` 中設定 `MinimumPollers` 參數，為事件來源映射啟用佈建模式。

無效的結構描述登錄檔 URL

錯誤訊息：`Malformed SchemaRegistryURI provided. Please provide a valid URI or ARN. For example, https://schema-registry.example.com:8081 or arn:aws:glue:us-east-1:123456789012:registry/ExampleRegistry.`

解決方案：為 Confluent 結構描述登錄檔提供有效的 HTTPS URL，或為 AWS Glue 結構描述登錄檔提供有效的 ARN。

無效或缺少事件記錄格式

錯誤訊息：`EventRecordFormat is a required field for SchemaRegistryConfig. Please provide one of supported format types: SOURCE, JSON.`

解決方案：在結構描述登錄檔組態中，將 SOURCE 或 JSON 指定為 EventRecordFormat。

重複驗證屬性

錯誤訊息：`Duplicate KEY/VALUE Attribute in SchemaValidationConfigs. SchemaValidationConfigs must contain at most one KEY/VALUE Attribute.`

解決方案：從 SchemaValidationConfigs 中移除重複的 KEY 或 VALUE 屬性。每種屬性類型只能出現一次。

缺少驗證組態

錯誤訊息：`SchemaValidationConfigs is a required field for SchemaRegistryConfig.`

解決方案：將 SchemaValidationConfigs 新增至組態，指定至少一個驗證屬性 (KEY 或 VALUE)。

存取與許可錯誤

當 Lambda 陰許可或身分驗證問題而無法存取結構描述登錄檔時，就會發生這些錯誤。

AWS Glue 結構描述登錄檔存取遭拒

錯誤訊息：`Cannot access Glue Schema with provided role. Please ensure the provided role can perform the GetRegistry and GetSchemaVersion Actions on your schema.`

解決方案：將必要許可 (`glue:GetRegistry` 與 `glue:GetSchemaVersion`) 新增至函式的執行角色。

Confluent 結構描述登錄檔存取遭拒

錯誤訊息：`Cannot access Confluent Schema with the provided access configuration.`

解決方案：確認身分驗證憑證 (儲存於 Secrets Manager 中) 是否正確，以及是否具有存取結構描述登錄檔的必要許可。

跨帳戶 AWS Glue 結構描述登錄檔

錯誤訊息：`Cross-account Glue Schema Registry ARN not supported.`

解決方案：使用與 Lambda 函數位於相同 AWS 帳戶的 AWS Glue 結構描述登錄檔。

跨區域 AWS Glue 結構描述登錄檔

錯誤訊息：`Cross-region Glue Schema Registry ARN not supported.`

解決方案：使用與 Lambda 函數位於相同區域的 AWS Glue 結構描述登錄檔。

秘密存取問題

錯誤訊息：`Lambda received InvalidRequestException from Secrets Manager.`

解決方法：確認函數的執行角色具有存取秘密的許可，而且如果從其他帳戶存取 ，則不會使用預設 AWS KMS 金鑰加密秘密。

連線錯誤

當 Lambda 無法與結構描述登錄檔建立連線時，就會發生這些錯誤。

VPC 連線問題

錯誤訊息：`Cannot connect to your Schema Registry. Your Kafka cluster's VPC must be able to connect to the schema registry. You can provide access by configuring AWS PrivateLink or a NAT Gateway or VPC Peering between Kafka Cluster VPC and the schema registry VPC.`

解決方案：設定您的 VPC 聯網，以允許使用 AWS PrivateLink NAT Gateway 或 VPC 對等互連連線至結構描述登錄檔。

TLS 交握失敗

錯誤訊息：`Unable to establish TLS handshake with the schema registry. Please provide correct CA-certificate or client certificate using Secrets Manager to access your schema registry.`

解決方案：確認 CA 憑證與用戶端憑證 (適用於 mTLS) 是否正確，是否在 Secrets Manager 中正確設定。

限流

錯誤訊息：`Receiving throttling errors when accessing the schema registry. Please increase API TPS limits for your schema registry.`

解決方案：提高結構描述登錄檔的 API 速率限制，或降低應用程式的請求速率。

自我管理結構描述登錄檔錯誤

錯誤訊息：`Lambda received an internal server an unexpected error from the provided self-managed schema registry.`

解決方案：檢查自我管理結構描述登錄檔伺服器的運作狀態與組態。

Kafka 事件來源的低延遲處理

AWS Lambda 對於需要低於 100 毫秒一致end-to-end延遲的應用程式，原生支援低延遲事件處理。本頁提供了啟用低延遲工作流程的組態詳細資訊和建議。

啟用低延遲處理

為實現 Kafka 事件來源映射的低延遲處理，需進行以下基礎組態設定：

- 啟用佈建模式。如需詳細資訊，請參閱佈建模式。
- 將事件來源映射的 `MaximumBatchingWindowInSeconds` 參數設定為 0。如需詳細資訊，請參閱批次處理行為。

微調低延遲 Kafka ESM

參考下列建議，最佳化 Kafka 事件來源映射來達成低延遲目標：

佈建模式組態

在 Kafka 事件來源映射的佈建模式下，Lambda 允許透過設定稱為事件輪詢器的資源數量下限與上限，實現對事件來源映射輸送量的微調。事件輪詢器 (或輪詢器) 代表在佈建模式下支援事件來源映射的運算資源，並且會配置高達 5 MB/s 的輸送量。每個事件輪詢器最多支援 5 個並行 Lambda 調用。

若要判斷應用程式的最佳輪詢器組態，建議考量尖峰擷取速率與處理需求。我們來看一個簡化範例：

在批次大小為 20 筆記錄且目標函式平均持續時間為 50 毫秒的情況下，每個輪詢器在 5 MB/s 的限制下每秒可處理 2,000 筆記錄。其計算方式為：(20 筆記錄 × 1000 毫秒/50 毫秒) × 5 個並行 Lambda 調用。因此，如果所需的尖峰擷取速率為每秒 20,000 筆記錄，將至少需要 10 個事件輪詢器。

> **Note**
>
> 建議將額外的事件輪詢器佈建為緩衝區，避免持續以容量上限運作。

佈建模式會在設定的事件輪詢器下限與上限範圍內，根據流量模式自動擴展事件輪詢器。這可能會觸發重新平衡，進而引入額外的延遲。將事件輪詢器下限與上限設定為相同的值，即可停用自動擴展。

其他考量

其他考量事項包括：

- 調用 Lambda 目標函式時產生的冷啟動可能會增加端對端延遲。若要降低此風險，建議在事件來源映射的目標函式上啟用佈建並行或 SnapStart。此外，最佳化函式的記憶體配置，確保一致且最佳的執行效能。
- 當 `MaximumBatchingWindowInSeconds` 設定為 0 時，Lambda 會立即處理任何可用的記錄，而不會等待累積到完整的批次大小。例如，若批次大小設定為 1,000 筆記錄，但只有 100 筆記錄可用，Lambda 會立即處理這 100 筆記錄，而不是等待累積到完整的 1,000 筆記錄。

> **Important**
>
> 低延遲處理的最佳組態會因特定工作負載而存在顯著差異。強烈建議使用實際工作負載測試不同的組態，判斷適合使用案例的最佳設定。

設定 Kafka 事件來源的錯誤處理控制項

您可以設定 Lambda 如何處理 Kafka 事件來源映射的錯誤和重試。這些組態可協助您控制 Lambda 如何處理失敗的記錄和管理重試行為。

可用的重試組態

下列重試組態適用於 Amazon MSK 和自我管理 Kafka 事件來源：

- 重試次數上限 – 當函數傳回錯誤時，Lambda 重試次數上限。這不會計入初始調用嘗試。預設值為 -1 （無限）。當您同時設定無限次重試和失敗時的目的地時，Lambda 會自動套用最多 10 次重試嘗試。
- 記錄存留期上限 – Lambda 傳送給函數的記錄存留期上限。預設值為 -1 （無限）。
- 錯誤時分割批次 – 當函數傳回錯誤時，請將批次分割為兩個較小的批次，並個別重試。這有助於隔離有問題的記錄。
- 部分批次回應 – 允許函數傳回批次中哪些記錄處理失敗的資訊，因此 Lambda 只能重試失敗的記錄。

設定錯誤處理控制 （主控台）

您可以在 Lambda 主控台中建立或更新 Kafka 事件來源映射時設定重試行為。

設定 Kafka 事件來源的重試行為 （主控台）

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您的函數名稱。
3. 執行以下任意一項：
   - 若要新增新的 Kafka 觸發條件，請在函數概觀下，選擇新增觸發條件。
   - 若要修改現有的 Kafka 觸發條件，請選擇觸發條件，然後選擇編輯。
4. 在事件輪詢器組態下，選取佈建模式以設定錯誤處理控制項：
   a. 對於重試嘗試，輸入重試嘗試次數上限 (0-10000，或無限輸入 -1)。
   b. 針對記錄存留期上限，以秒為單位輸入存留期上限 (60-604800，無限輸入 -1)。
   c. 若要在發生錯誤時啟用批次分割，請選取錯誤時分割批次。
   d. 若要啟用部分批次回應，請選取 ReportBatchItemFailures。
5. 選擇新增或儲存。

設定重試行為 (AWS CLI)

使用下列 AWS CLI 命令來設定 Kafka 事件來源映射的重試行為。

使用重試組態建立事件來源映射

下列範例會使用錯誤處理控制項建立自我管理的 Kafka 事件來源映射：

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics my-kafka-topic \
  --source-access-configuration Type=SASL_SCRAM_512_AUTH,URI=arn:aws:secretsmanager:us-
east-1:111122223333:secret:MyBrokerSecretName \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc.xyz.com:9092"]}}' \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=1 \
  --maximum-retry-attempts 3 \
  --maximum-record-age-in-seconds 3600 \
  --bisect-batch-on-function-error \
  --function-response-types "ReportBatchItemFailures"
```

對於 Amazon MSK 事件來源：

```
aws lambda create-event-source-mapping \
  --event-source-arn arn:aws:kafka:us-east-1:111122223333:cluster/my-cluster/fc2f5bdf-
fd1b-45ad-85dd-15b4a5a6247e-2 \
  --topics AWSMSKKafkaTopic \
  --starting-position LATEST \
  --function-name my-kafka-function \
  --source-access-configurations '[{"Type": "SASL_SCRAM_512_AUTH","URI":
 "arn:aws:secretsmanager:us-east-1:111122223333:secret:my-secret"}]' \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=1 \
  --maximum-retry-attempts 3 \
  --maximum-record-age-in-seconds 3600 \
  --bisect-batch-on-function-error \
  --function-response-types "ReportBatchItemFailures"
```

更新重試組態

使用 `update-event-source-mapping`命令來修改現有事件來源映射的重試組態：

```
aws lambda update-event-source-mapping \
```

```
  --uuid 12345678-1234-1234-1234-123456789012 \
  --maximum-retry-attempts 5 \
  --maximum-record-age-in-seconds 7200 \
  --bisect-batch-on-function-error \
  --function-response-types "ReportBatchItemFailures"
```

PartialBatchResponse

部分批次回應也稱為 ReportBatchItemFailures，是 Lambda 與 Kafka 來源整合時處理錯誤的重要功能。如果沒有此功能，當批次中其中一個項目發生錯誤時，會導致重新處理該批次中的所有訊息。啟用並實作部分批次回應後，處理常式只會傳回失敗訊息的識別符，讓 Lambda 只重試這些特定項目。這可讓您更好地控制包含失敗訊息的批次處理方式。

若要報告批次錯誤，您將使用此 JSON 結構描述：

```
{
  "batchItemFailures": [
    {
      "itemIdentifier": {
        "partition": "topic-partition_number",
        "offset": 100
      }
    },
    ...
  ]
}
```

> **Important**
>
> 如果您傳回空的有效 JSON 或 null，事件來源映射會將批次視為已成功處理。傳回的任何無效 topic-partition_number 或位移若不存在於調用事件中，將視為失敗，並重試整個批次。

下列程式碼範例示範如何為從 Kafka 來源接收事件的 Lambda 函數實作部分批次回應。此函數會在回應中報告批次項目失敗，指示 Lambda 稍後重試這些訊息。

以下是顯示此方法的 Python Lambda 處理常式實作：

```
import base64
from typing import Any, Dict, List
```

```
def lambda_handler(event: Dict[str, Any], context: Any) -> Dict[str, List[Dict[str,
 Dict[str, Any]]]]:
    failures: List[Dict[str, Dict[str, Any]]] = []
    records_dict = event.get("records", {})

    for topic_partition, records_list in records_dict.items():
        for record in records_list:
            topic = record.get("topic")
            partition = record.get("partition")
            offset = record.get("offset")
            value_b64 = record.get("value")

            try:
                data = base64.b64decode(value_b64).decode("utf-8")
                process_message(data)
            except Exception as exc:
                print(f"Failed to process record topic={topic} partition={partition}
 offset={offset}: {exc}")
                item_identifier: Dict[str, Any] = {
                    "partition": f"{topic}-{partition}",
                    "offset": int(offset) if offset is not None else None,
                }
                failures.append({"itemIdentifier": item_identifier})

    return {"batchItemFailures": failures}

def process_message(data: str) -> None:
    # Your business logic for a single message
    pass
```

以下是 Node.js 版本：

```
const { Buffer } = require("buffer");

const handler = async (event) => {
  const failures = [];

  for (let topicPartition in event.records) {
    const records = event.records[topicPartition];

    for (const record of records) {
      const topic = record.topic;
      const partition = record.partition;
```

```
      const offset = record.offset;
      const valueBase64 = record.value;
      const data = Buffer.from(valueBase64, "base64").toString("utf8");

      try {
        await processMessage(data);
      } catch (error) {
        console.error("Failed to process record", { topic, partition, offset, error });
        const itemIdentifier = {
          "partition": `${topic}-${partition}`,
          "offset": Number(offset),
        };
        failures.push({ itemIdentifier });
      }
    }
  }

  return { batchItemFailures: failures };
};

async function processMessage(payload) {
  // Your business logic for a single message
}

module.exports = { handler };
```

擷取 Amazon MSK 和自我管理的 Apache Kafka 事件來源的捨棄批次

若要保留失敗的事件來源映射調用記錄，請將目標地新增到函數的事件來源映射中。傳送至該目的地的每筆記錄都是包含失敗調用之中繼資料的 JSON 文件。對於 Amazon S3 目的地，Lambda 還會將整個調用記錄與中繼資料一起傳送。您可以設定任何 Amazon SNS 主題、Amazon SQS 佇列、Amazon S3 儲存貯體或 Kafka 做為目的地。

透過 Amazon S3 目的地，您可以使用 Amazon S3 事件通知功能，在物件上傳至您的目的地 S3 儲存貯體時接收通知。您也可以設定 S3 事件通知來調用另一個 Lambda 函數，以對失敗的批次執行自動處理。

執行角色必須具有目的地的許可：

- 對於 SQS 目的地：sqs：SendMessage
- 對於 SNS 目的地：sns：Publish
- 對於 S3 目的地：s3：PutObject 和 s3：ListBucket
- 對於 Kafka 目的地：kafka-cluster：WriteData

您可以將 Kafka 主題設定為 Kafka 事件來源映射的失敗時目的地。當 Lambda 在耗盡重試嘗試後或記錄超過最長存留期後無法處理記錄時，Lambda 會將失敗的記錄傳送至指定的 Kafka 主題，以供稍後處理。請參閱the section called “Kafka 失敗時目的地”。

必須在 Kafka 叢集 VPC 內為失敗時目的地服務部署 VPC 端點。

此外，如果您在目的地上設定 KMS 金鑰，則視目的地類型而定，Lambda 需要下列許可：

- 如果您已針對 S3 目的地使用自己的 KMS 金鑰啟用加密，則需要 kms:GenerateDataKey。如果 KMS 金鑰和 S3 儲存貯體目的地與 Lambda 函數和執行角色位於不同的帳戶中，請將 KMS 金鑰設定為信任執行角色以允許 kms:GenerateDataKey。
- 如果您已針對 SQS 目的地使用自己的 KMS 金鑰啟用加密，則需要 kms:Decrypt 和 kms:GenerateDataKey。如果 KMS 金鑰和 SQS 佇列目的地與 Lambda 函數和執行角色位於不同的帳戶中，請將 KMS 金鑰設定為信任執行角色，以允許 kms:Decrypt、kms:GenerateDataKey、kms:DescribeKey 以及 kms:ReEncrypt。
- 如果您已針對 SNS 目的地使用自己的 KMS 金鑰啟用加密，則需要 kms:Decrypt 和 kms:GenerateDataKey。如果 KMS 金鑰和 SNS 主題目的地與 Lambda 函數和執行角色位於不同的帳戶中，請將 KMS 金鑰設定為信任執行角色，以允許 kms:Decrypt、kms:GenerateDataKey、kms:DescribeKey 以及 kms:ReEncrypt。

設定 Kafka 事件來源映射的失敗時目的地

若要使用主控台設定失敗時的目的地，請依照下列步驟執行：

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在函數概觀下，選擇新增目的地。
4. 針對來源，請選擇事件來源映射調用。
5. 對於事件來源映射，請選擇針對此函數設定的事件來源。
6. 對於條件，選取失敗時。對於事件來源映射調用，這是唯一可接受的條件。
7. 對於目標類型，請選擇 Lambda 將調用記錄傳送至的目標類型。
8. 對於目的地，請選擇一個資源。
9. 選擇儲存。

您也可以使用 AWS CLI設定失敗時的目的地。例如，下列 create-event-source-mapping 命令會將具有 SQS 失敗時的目的地的事件來源映射新增至 `MyFunction`：

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

下列 update-event-source-mapping 命令會將 S3 失敗時的目的地新增至與輸入 `uuid` 相關聯的事件來源：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": "arn:aws:s3:::dest-bucket"}}'
```

若要移除目的地，請提供空白字串作為 `destination-config` 參數的引數：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Amazon S3 目的地的安全最佳實務

刪除已設定為目的地的 S3 儲存貯體而不從函數的組態中移除目的地，可能會產生安全風險。如果其他使用者知道目的地儲存貯體的名稱，他們可以在其 AWS 帳戶中重新建立儲存貯體。失敗調用的記錄會被傳送到其儲存貯體，可能公開來自您函數的資料。

> **Warning**
>
> 為了確保無法將來自函數的調用記錄傳送到另一個 中的 S3 儲存貯體 AWS 帳戶，請將條件新增至函數的執行角色，以限制您帳戶中儲存貯體的`s3:PutObject`許可。

下列範例顯示的 IAM 政策，將函數的 `s3:PutObject` 許可限制為帳戶中的儲存貯體。此政策也為 Lambda 提供了使用 S3 儲存貯體做為目的地所需的 `s3:ListBucket` 許可。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
}
```

若要使用 AWS 管理主控台 或 將許可政策新增至函數的執行角色 AWS CLI，請參閱下列程序中的指示：

Console

若要將許可政策新增至函數的執行角色 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選取您要修改其執行角色的 Lambda 函數。
3. 在組態索引標籤中，選擇許可。
4. 在執行角色索引標籤中，選取函數的角色名稱，以開啟角色的 IAM 主控台頁面。
5. 透過下列步驟將許可政策新增至角色：

   a. 在許可政策窗格中，選擇新增許可 ，然後選取建立內嵌政策。
   b. 在政策編輯器中，選取 JSON。
   c. 將您要新增的政策貼入編輯器 (取代現有的 JSON)，然後選擇下一步。
   d. 在政策詳細資訊下，輸入政策名稱。
   e. 選擇建立政策。

AWS CLI

若要將許可政策新增至函數的執行角色 (CLI)

1. 建立具有所需許可的 JSON 政策文件，並將其儲存在本機目錄中。
2. 使用 IAM `put-role-policy` CLI 命令，將許可新增至函數的執行角色。從您儲存 JSON 政策文件的目錄執行下列命令，並將角色名稱、政策名稱和政策文件取代為您自己的值。

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

SNS 和 SQS 調用記錄範例

下列範例顯示 Lambda 針對失敗的 Kafka 事件來源調用而傳送至 SNS 主題或 SQS 佇列目的地。`recordsInfo` 之下的每個索引鍵都包含 Kafka 主題和分割區，以連字號分隔。例如，對於金鑰 `"Topic-0"`，`Topic` 是 Kafka 主題，並且 `0` 是分區。對於每個主題和分區，您可以使用偏移和時間戳記資料來查找原始調用記錄。

```
{
```

```
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
        }
    }
}
```

S3 目的地調用記錄範例

對於 S3 的目的地，Lambda 會將整個調用記錄與中繼資料一起傳送至目的地。下列範例顯示 Lambda 針對失敗的 Kafka 事件來源調用而傳送至 S3 儲存貯體目的地。除了上一個 SQS 和 SNS 目的地範例中的所有欄位之外，此 payload 欄位還包含原始調用記錄做為逸出 JSON 字串。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted" | "MaximumPayloadSizeExceeded",
        "approximateInvokeCount": 1
    },
    "responseContext": { // null if record is MaximumPayloadSizeExceeded
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KafkaBatchInfo": {
        "batchSize": 500,
        "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/
vpc-2priv-2pub/751d2973-a626-431c-9d4e-d7975eb44dd7-2",
        "bootstrapServers": "...",
        "payloadSize": 2039086, // In bytes
        "recordsInfo": {
            "Topic-0": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            },
            "Topic-1": {
                "firstRecordOffset":
 "49601189658422359378836298521827638475320189012309704722",
                "lastRecordOffset":
 "49601189658422359378836298522902373528957594348623495186",
                "firstRecordTimestamp": "2019-11-14T00:38:04.835Z",
                "lastRecordTimestamp": "2019-11-14T00:38:05.580Z",
            }
```

```
        }
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

> **Tip**
>
> 建議您在目的地儲存貯體上啟用 S3 版本控制。

使用 Kafka 主題做為失敗時的目的地

您可以將 Kafka 主題設定為 Kafka 事件來源映射的失敗時目的地。當 Lambda 在耗盡重試嘗試後或記錄超過最長存留期後無法處理記錄時，Lambda 會將失敗的記錄傳送至指定的 Kafka 主題，以供稍後處理。當您同時設定無限次重試和失敗時的目的地時，Lambda 會自動套用最多 10 次重試嘗試。

Kafka 失敗時目的地的運作方式

當您將 Kafka 主題設定為失敗時的目的地時，Lambda 會充當 Kafka 生產者，並將失敗的記錄寫入目的地主題。這會在 Kafka 基礎設施內建立無效字母主題 (DLT) 模式。

- 相同的叢集需求 – 目的地主題必須與來源主題存在於相同的 Kafka 叢集中。
- 實際記錄內容 – Kafka 目的地會收到實際失敗的記錄以及失敗中繼資料。
- 防止遞迴 – Lambda 透過封鎖來源和目的地主題相同的組態來防止無限迴圈。

設定 Kafka 失敗時的目的地

您可以在建立或更新 Kafka 事件來源映射時，將 Kafka 主題設定為失敗時的目的地。

設定 Kafka 目的地 （主控台）

將 Kafka 主題設定為失敗時的目的地 （主控台）

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您的函數名稱。
3. 執行以下任意一項：
    - 若要新增新的 Kafka 觸發條件，請在函數概觀下，選擇新增觸發條件。
    - 若要修改現有的 Kafka 觸發條件，請選擇觸發條件，然後選擇編輯。
4. 在其他設定下，針對失敗目的地，選擇 Kafka 主題。
5. 針對主題名稱，輸入您要傳送失敗記錄的 Kafka 主題名稱。
6. 選擇新增或儲存。

設定 Kafka 目的地 (AWS CLI)

使用 `kafka://`字首將 Kafka 主題指定為失敗時的目的地。

使用 Kafka 目的地建立事件來源映射

下列範例會建立 Amazon MSK 事件來源映射，並以 Kafka 主題做為失敗時的目的地：

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics AWSKafkaTopic \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123 \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --destination-config '{"OnFailure":{"Destination":"kafka://failed-records-topic"}}'
```

對於自我管理的 Kafka，請使用相同的語法：

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics AWSKafkaTopic \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc.xyz.com:9092"]}}' \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --destination-config '{"OnFailure":{"Destination":"kafka://failed-records-topic"}}'
```

更新 Kafka 目的地

使用 `update-event-source-mapping`命令來新增或修改 Kafka 目的地：

```
aws lambda update-event-source-mapping \
  --uuid 12345678-1234-1234-1234-123456789012 \
  --destination-config '{"OnFailure":{"Destination":"kafka://failed-records-topic"}}'
```

Kafka 目的地的記錄格式

當 Lambda 將失敗的記錄傳送到 Kafka 主題時，每個訊息都會包含有關失敗和實際記錄內容的中繼資料。

失敗中繼資料

中繼資料包含記錄失敗原因的相關資訊，以及原始批次的詳細資訊：

```
{
  "requestContext": {
    "requestId": "e4b46cbf-b738-xmpl-8880-a18cdf61200e",
```

```
    "functionArn": "arn:aws:lambda:us-east-1:123456789012:function:my-function:
$LATEST",
    "condition": "RetriesExhausted",
    "approximateInvokeCount": 3
  },
  "responseContext": {
    "statusCode": 200,
    "executedVersion": "$LATEST",
    "functionError": "Unhandled"
  },
  "version": "1.0",
  "timestamp": "2019-11-14T18:16:05.568Z",
  "KafkaBatchInfo": {
    "batchSize": 1,
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123",
    "bootstrapServers": "b-1.mycluster.abc123.kafka.us-east-1.amazonaws.com:9098",
    "payloadSize": 1162,
    "recordInfo": {
      "offset": "4960118965842235937883629852182763847532018901230970472 2",
      "timestamp": "2019-11-14T18:16:04.835Z"
    }
  },
  "payload": {
    "bootstrapServers": "b-1.mycluster.abc123.kafka.us-east-1.amazonaws.com:9098",
    "eventSource": "aws:kafka",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123",
    "records": {
      "my-topic-0": [
        {
          "headers": [],
          "key": "dGVzdC1rZXk=",
          "offset": 100,
          "partition": 0,
          "timestamp": 1749116692330,
          "timestampType": "CREATE_TIME",
          "topic": "my-topic",
          "value": "dGVzdC12YWx1ZQ=="
        }
      ]
    }
  }
}
```

分割區索引鍵行為

在產生目的地主題時，Lambda 會使用原始記錄中的相同分割區索引鍵。如果原始記錄沒有金鑰，Lambda 會在目的地主題中的所有可用分割區中使用 Kafka 的預設循環配置分割。

要求與限制

- 需要佈建模式 – Kafka 故障時目的地僅適用於啟用佈建模式的事件來源映射。
- 僅限相同叢集 – 目的地主題必須與來源主題存在於相同的 Kafka 叢集中。
- 主題許可 – 您的事件來源映射必須具有目的地主題的寫入許可。範例：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ClusterPermissions",
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:Connect",
                "kafka-cluster:DescribeCluster",
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:WriteData",
                "kafka-cluster:ReadData"
            ],
            "Resource": [
                "arn:aws:kafka:*:*:cluster/*"
            ]
        },
        {
            "Sid": "TopicPermissions",
            "Effect": "Allow",
            "Action": [
                "kafka-cluster:DescribeTopic",
                "kafka-cluster:WriteData",
                "kafka-cluster:ReadData"
            ],
            "Resource": [
                "arn:aws:kafka:*:*:topic/*/*"
            ]
        },
        {
            "Effect": "Allow",
```

```
            "Action": [
                "kafka:DescribeCluster",
                "kafka:GetBootstrapBrokers",
                "kafka:Produce"
            ],
            "Resource": "arn:aws:kafka:*:*:cluster/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeSubnets",
                "ec2:DescribeSecurityGroups"
            ],
            "Resource": "*"
        }
    ]
}
```

- **無遞迴 – 目的地主題名稱不能與來源主題名稱相同。**

Kafka 事件來源映射記錄

您可以設定 Kafka 事件來源映射的系統層級記錄，以啟用和篩選 Lambda 事件輪詢器傳送至 CloudWatch 的系統日誌。

此功能僅適用於 Kafka 事件來源映射，以及佈建模式。

對於使用記錄組態的事件來源映射，您也可以立即從主控台 Lambda > 其他資源 > 事件來源映射頁面的監控索引標籤中，從預先建置的日誌查詢檢查系統日誌。

記錄的運作方式

當您在事件來源映射中設定日誌層級的記錄組態時，Lambda 事件輪詢器會傳送對應的日誌 （事件來源映射系統日誌）。

事件來源映射會與您的 Lambda 函數重複使用相同的日誌目的地。請確定 Lambda 函數的執行角色具有必要的記錄許可。

事件來源映射將擁有自己的日誌串流，其中日期和事件來源映射 UUID 作為日誌串流名稱，例如 `2020/01/01/12345678-1234-1234-1234-12345678901`。

對於事件來源映射系統日誌，您可以選擇下列日誌層級。

日誌層級	Usage
DEBUG (大多數詳細資訊)	事件來源處理進度的詳細資訊
INFO	有關事件來源映射正常操作的訊息
WARN (最少詳細資訊)	有關可能導致意外行為之潛在警告和錯誤的訊息

當您選取日誌層級時，Lambda 事件輪詢器會在該層級和更低層級傳送日誌。例如，如果您將事件來源映射系統日誌層級設定為 INFO，事件輪詢器不會在 DEBUG 層級傳送日誌輸出。

設定 記錄

您可以在建立或更新 Kafka 事件來源映射時設定記錄設定。

設定記錄 （主控台）

設定記錄 （主控台）

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您的函數名稱。
3. 執行以下任意一項：
   - 若要新增新的 Kafka 觸發條件，請在函數概觀下，選擇新增觸發條件。
   - 若要修改現有的 Kafka 觸發條件，請選擇觸發條件，然後選擇編輯。
4. 在事件輪詢器組態下，針對佈建模式，啟用設定核取方塊。此時會顯示日誌層級設定。
5. 按一下日誌層級下拉式清單，並選取事件來源映射的層級。
6. 選擇底部的新增或儲存，以建立或更新事件來源映射。

設定記錄 (AWS CLI)

使用 記錄建立事件來源映射

下列範例會使用記錄組態建立 Amazon MSK 事件來源映射：

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics AWSKafkaTopic \
  --event-source-arn arn:aws:kafka:us-east-1:123456789012:cluster/my-cluster/abc123 \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --logging-config '{"SystemLogLevel":"DEBUG"}'
```

對於自我管理的 Kafka，請使用相同的語法：

```
aws lambda create-event-source-mapping \
  --function-name my-kafka-function \
  --topics AWSKafkaTopic \
  --self-managed-event-source '{"Endpoints":{"KAFKA_BOOTSTRAP_SERVERS":
["abc.xyz.com:9092"]}}' \
  --starting-position LATEST \
  --provisioned-poller-config MinimumPollers=1,MaximumPollers=3 \
  --logging-config '{"SystemLogLevel":"DEBUG"}'
```

更新記錄組態

使用 update-event-source-mapping命令來新增或修改記錄組態：

```
aws lambda update-event-source-mapping \
  --uuid 12345678-1234-1234-1234-123456789012 \
  --logging-config '{"SystemLogLevel":"WARN"}'
```

Kafka 事件來源映射系統日誌的記錄格式

當 Lambda 事件輪詢器傳送日誌時，每個日誌項目都包含一般事件來源映射中繼資料，以及事件特定內容。

WARN 日誌記錄

WARN 記錄包含來自事件輪詢器的錯誤或警告，並在事件發生時發出。例如：

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1546347650000,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:12345678-1234-1234-1234-123456789012",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/tests-
cluster/87654321-4321-4321-4321-876543221-s1",
    "eventProcessorId": "12345678-1234-1234-1234-123456789012/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "Timeout expired while fetching topic metadata",
        "errorCode": "org.apache.kafka.common.errors.TimeoutException"
    }
}
```

INFO 日誌記錄

INFO 記錄包含每個事件輪詢器中的 Kafka 取用者用戶端組態，並在建立或變更取用者時發出。例如：

```
{
    "eventType": "POLLER_STATUS_EVENT",
    "timestamp": 1546347660000,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:12345678-1234-1234-1234-123456789012",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/tests-
cluster/87654321-4321-4321-4321-876543221-s1",
```

```
    "eventProcessorId": "12345678-1234-1234-1234-123456789012/0",
    "logLevel": "INFO",
    "kafkaEventSourceConnection": {
        "brokerEndpoints": "boot-abcd1234.c2.kafka-serverless.us-
east-1.amazonaws.com:9098",
        "consumerId": "12345678-1234-1234-1234-123456789012-0",
        "topics": [
            "test"
        ],
        "consumerGroupId": "12345678-1234-1234-1234-123456789012",
        "securityProtocol": "SASL_SSL",
        "saslMechanism": "AWS_MSK_IAM",
        "totalPartitionCount": 2,
        "assignedPartitionCount": 2,
        "partitionsAssignmentGeneration": 5,
        "assignedPartitions": [
            "test-0",
            "test-1"
        ],
        "networkConfig": {
            "ipAddresses": [
                "10.100.141.1"
            ],
            "subnetCidrBlock": "10.100.128.0/20",
            "securityGroups": [
                "sg-abcdefabcdefabcdef"
            ]
        }
    }
}
```

DEBUG 日誌記錄

DEBUG 日誌包含事件來源映射處理中的 Kafka 偏移相關資訊，並且每分鐘發出偏移資訊。例如：

```
{
    "eventType": "KAFKA_STATUS_EVENT",
    "timestamp": 1546347670000,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:12345678-1234-1234-1234-123456789012",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/tests-
cluster/87654321-4321-4321-4321-876543221-s1",
    "eventProcessorId": "12345678-1234-1234-1234-123456789012/0",
    "logLevel": "DEBUG",
```

```
    "kafkaPartitionOffsets": {
        "partition": "test-1",
        "endOffset": 5004,
        "consumedOffset": 5003,
        "processedOffset": 5003,
        "committedOffset": 5004
    }
}
```

對 Kafka 事件來源映射錯誤進行疑難排解

下列主題針對您在將 Amazon MSK 或自我管理的 Apache Kafka 與 Lambda 搭配使用時可能遇到的錯誤和問題，提供疑難排解建議。

如需更多疑難排解說明，請造訪 AWS 知識中心。

身分驗證和授權錯誤

如果遺失了從 Kafka 叢集取用資料的任何必要許可，Lambda 會在 LastProcessingResult 下的事件來源映射中顯示下列其中一種錯誤訊息。

錯誤訊息

- 叢集無法授權 Lambda
- SASL 身分驗證失敗
- 伺服器無法驗證 Lambda
- Lambda 無法驗證伺服器
- 提供的憑證或私有金鑰無效

叢集無法授權 Lambda

對於 SASL/SCRAM 或 mTLS，此錯誤表示提供的使用者不具備下列 Kafka 存取控制清單 (ACL) 的所有許可：

- DescribeConfigs 叢集
- 描述群組
- 讀取群組
- 描述主題
- 讀取主題

當您建立具有必要的 `kafka-cluster` 許可之 Kafka ACL 時，請將主題和群組指定為資源。主題名稱必須與事件來源映射中的主題相符。群組名稱必須與事件來源映射的 UUID 相符。

將必要的許可新增至執行角色後，可能需要數分鐘變更才會生效。

以下是針對此問題啟用記錄組態後的範例 ESM 系統層級日誌：

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567890123,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE11111",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE11111-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "Not authorized to access topics: [my-topic]",
        "errorCode": "org.apache.kafka.common.errors.TopicAuthorizationException"
    }
}
```

SASL 身分驗證失敗

若使用 SASL/SCRAM 或 SASL/PLAIN，此錯誤表示所提供的登入憑證無效。

對於 IAM 存取控制，執行角色缺少叢集的 `kafka-cluster:Connect` 許可。將此許可新增至角色，並將叢集的 Amazon Resource Name (ARN) 指定為資源。

您可能會間歇性地看到此錯誤發生。TCP 連線數量超過服務配額後，此叢集就會拒絕連線。Lambda 會關閉並重試，直到連線成功為止。Lambda 連接到叢集並輪詢記錄之後，上次處理結果會變更為 OK。

以下是在使用 IAM 身分驗證時，針對此問題啟用記錄組態後的範例 ESM 系統層級日誌：

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567890456,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE22222-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "[a1b2c3d4-5678-90ab-cdef-EXAMPLE22222]: Access denied",
        "errorCode": "org.apache.kafka.common.errors.SaslAuthenticationException"
    }
}
```

伺服器無法驗證 Lambda

此錯誤表示 Kafka 代理程式無法驗證 Lambda。此狀況的發生原因如下：

- 您並未提供 mTLS 身分驗證的用戶端憑證。
- 您已提供用戶端憑證，但並未將 Kafka 代理程式設定為使用 mTLS 身分驗證。
- 用戶端憑證不受 Kafka 代理程式信任。

Lambda 無法驗證伺服器

此錯誤表示 Lambda 無法驗證 Kafka 代理程式。此狀況的發生原因如下：

- 對於自我管理的 Apache Kafka：Kafka 代理程式會使用自我簽署憑證或私有 CA，但不會提供伺服器根 CA 憑證。
- 對於自我管理的 Apache Kafka：伺服器根 CA 憑證與簽署代理程式憑證的根 CA 不相符。
- 主機名稱驗證失敗，因為代理程式的憑證不包含代理程式做為主體替代名稱的 DNS 名稱或 IP 地址。

提供的憑證或私有金鑰無效

此錯誤表示 Kafka 取用者無法使用提供的憑證或私有金鑰。請確定憑證和金鑰使用 PEM 格式，且私有金鑰加密使用 PBES1 演算法。

以下是針對此問題啟用記錄組態後的範例 ESM 系統層級日誌：

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567891234,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE44444",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE44444-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE44444/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "Invalid PEM keystore configs",
        "errorCode": "org.apache.kafka.common.errors.InvalidConfigurationException"
    }
}
```

網路和連線錯誤

網路組態問題可防止 Lambda 連線至 Kafka 叢集。下列主題說明常見的網路相關錯誤。

錯誤訊息

- 安全群組組態導致連線逾時
- 無法解析 Kafka 代理程式端點

安全群組組態導致連線逾時

如果與 Kafka 叢集相關聯的安全群組不允許來自本身的傳入流量，則 Lambda 無法連線至叢集。確定安全群組的傳入規則允許來自 Kafka 代理程式連接埠上安全群組本身的流量。

以下是針對此問題啟用記錄組態後的範例 ESM 系統層級日誌：

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567892345,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE55555",
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE55555-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE55555/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "Timeout expired while fetching topic metadata",
        "errorCode": "org.apache.kafka.common.errors.TimeoutException"
    }
}
```

您也可以檢查 Kafka 消費者 INFO 日誌，以驗證連線和網路組態。`brokerEndpoints` 欄位顯示 Kafka 代理程式地址，`securityProtocol`而 `saslMechanism`（如果適用）顯示身分驗證方法，而 `networkConfig` 欄位顯示事件來源映射所使用的 IP 地址、子網路 CIDR 區塊和安全群組。確認列出的安全群組允許必要的傳入流量：

```
{
    "eventType": "POLLER_STATUS_EVENT",
    "timestamp": 1734567892456,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-11111EXAMPLE",
```

```
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-cluster/
a1b2c3d4-5678-90ab-cdef-11111EXAMPLE-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE/0",
    "logLevel": "INFO",
    "kafkaEventSourceConnection": {
        "brokerEndpoints": "boot-abcd1234.c2.kafka-serverless.us-
east-1.amazonaws.com:9098",
        "consumerId": "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE-0",
        "topics": [
            "my-topic"
        ],
        "consumerGroupId": "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE",
        "securityProtocol": "SASL_SSL",
        "saslMechanism": "AWS_MSK_IAM",
        "totalPartitionCount": 2,
        "assignedPartitionCount": 2,
        "partitionsAssignmentGeneration": 1,
        "assignedPartitions": [
            "my-topic-0",
            "my-topic-1"
        ],
        "networkConfig": {
            "ipAddresses": [
                "10.0.0.37"
            ],
            "subnetCidrBlock": "10.0.0.32/28",
            "securityGroups": [
                "sg-0123456789abcdef0"
            ]
        }
    }
}
```

無法解析 Kafka 代理程式端點

此錯誤表示 Kafka 叢集不存在或已刪除。確認事件來源映射中指定的叢集存在且處於作用中狀態。

以下是針對此問題啟用記錄組態後的範例 ESM 系統層級日誌：

```
{
    "eventType": "ESM_PROCESSING_EVENT",
    "timestamp": 1734567893456,
    "resourceArn": "arn:aws:lambda:us-east-1:123456789012:event-source-
mapping:a1b2c3d4-5678-90ab-cdef-EXAMPLE66666",
```

```
    "eventSourceArn": "arn:aws:kafka:us-east-1:123456789012:cluster/my-kafka-
cluster/12345678-abcd-1234-efgh-EXAMPLE66666-1",
    "eventProcessorId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE66666/0",
    "logLevel": "WARN",
    "error": {
        "errorMessage": "No resolvable bootstrap urls given in bootstrap.servers",
        "errorCode": "org.apache.kafka.common.config.ConfigException"
    }
}
```

事件來源映射錯誤

當您將 Apache Kafka 叢集新增為 Lambda 函數的事件來源時，如果您的函數遇到錯誤，則您的 Kafka 取用者會停止處理記錄。主題分割區的取用者是訂閱、讀取和處理記錄者。您的其他 Kafka 取用者可以繼續處理記錄，只要他們沒有遇到相同的錯誤。

若要確定停止取用者的原因，請檢查 EventSourceMapping 的回應中的 StateTransitionReason 欄位。下列清單說明了您可能收到的事件來源錯誤：

ESM_CONFIG_NOT_VALID

事件來源映射組態無效。

EVENT_SOURCE_AUTHN_ERROR

Lambda 無法驗證事件來源。

EVENT_SOURCE_AUTHZ_ERROR

Lambda 沒有存取事件來源所需的許可。

FUNCTION_CONFIG_NOT_VALID

函數組態無效。

> **Note**
>
> 如果您的 Lambda 事件記錄超過允許的 6 MB 大小限制，則可能不會進行處理。

使用 Amazon API Gateway 端點調用 Lambda 函數

您可以使用 Amazon API Gateway 為您的 Lambda 函數建立具有 HTTP 端點的 Web API。API Gateway 提供了用於建立和記錄 Web API 的工具，可將 HTTP 請求路由至 Lambda 函數。您可以使用身分驗證和授權控制來保護對 API 的存取。您的 API 可以透過網際網路提供流量，也可以只在 VPC 內存取。

> **Tip**
>
> Lambda 提供兩種透過 HTTP 端點調用函數的方式：API Gateway 與 Lambda 函數 URL。如果不確定哪種方法最適合自己的使用案例，請參閱the section called “API Gateway 與函數 URL 比較”。

API 中的資源定義一個或多個方法，例如 GET 或 POST。方法具有將請求路由到 Lambda 函數或其他整合類型的整合。您可以個別定義每個資源和方法，或使用特殊資源和方法類型來比對所有符合某模式的請求。代理資源會擷取資源下的所有路徑。ANY 方法會擷取所有 HTTP 方法。

章節

- 選擇 API 類型
- 將端點新增至您的 Lambda 函數
- Proxy 整合
- 事件格式
- 回應格式
- 許可
- 範例應用程式
- Powertools for AWS Lambda 的事件處理常式
- 教學課程：搭配使用 Lambda 與 API Gateway
- 使用 API Gateway API 處理 Lambda 錯誤
- 選取一種使用 HTTP 請求調用 Lambda 函數的方法

選擇 API 類型

API Gateway 支援三種調用 Lambda 函數的 API 類型：

- HTTP API：輕量型的低延遲 RESTful API。
- REST API：功能豐富的可自訂 RESTful API。
- WebSocket API：與用戶端維持持續連線以進行全雙工通訊的 Web API。

HTTP API 和 REST API 都是處理 HTTP 請求並傳回回應的 RESTful API。HTTP API 較新，並且是使用 API Gateway 版本 2 API 建置而成。下列功能是 HTTP API 的新功能：

HTTP API 功能

- 自動部署 - 當您修改路由或整合時，變更會自動部署至已啟用自動部署的階段。
- 預設階段 - 您可以建立預設階段 (`$default`)，在 API URL 的根路徑提供請求。對於具名階段，您必須在路徑的開頭加入階段名稱。
- CORS 組態 - 您可以設定 API 將 CORS 標頭新增到傳出回應中，而不是在函數程式碼中手動新增。

REST API 是 API Gateway 自推出以來支援的典型 RESTful API。REST API 目前具有更多的自訂、整合和管理功能。

REST API 功能

- 整合類型 - REST API 支援自訂 Lambda 整合。您可以使用自訂整合，只將請求的本文傳送到函數，或者在將請求本文傳送到函數之前套用轉換範本。
- 存取控制 - REST API 支援其他身分驗證和授權選項。
- 監控和追蹤 – REST APIs支援 AWS X-Ray 追蹤和其他記錄選項。

如需詳細比較，請參閱《API Gateway 開發人員指南》中的 Choose between HTTP APIs and REST APIs。

WebSocket API 也使用 API Gateway 版本 2 API，並支援類似的功能集。對於可從用戶端與 API 之間持續連線獲益的應用程式，請使用 WebSocket API。WebSocket API 提供全雙工通訊，這意味著用戶端和 API 都可以連續傳送訊息，而無需等待回應。

HTTP API 支援簡化的事件格式 (2.0 版)。如需 HTTP API 事件的範例，請參閱在 API Gateway 中建立 HTTP APIs AWS Lambda 代理整合。

如需詳細資訊，請參閱在 API Gateway 中建立 HTTP APIs AWS Lambda 代理整合。

將端點新增至您的 Lambda 函數

若要將公有端點新增至您的 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在函數概觀下，選擇新增觸發條件。
4. 選取 API Gateway (API Gateway)。
5. 選擇 Create an API (建立 API) 或 Use an existing API (使用現有 API)。
   a. 全新 API：對於 API type (API 類型)，請選擇 HTTP API。如需詳細資訊，請參閱選擇 API 類型。
   b. 現有 API：從下拉式清單中選取 API 或輸入 API ID (例如，r3pmxmplak)。
6. 在 Security (安全性) 中，選擇 Open (開啟)。
7. 選擇 Add (新增)。

Proxy 整合

API Gateway API 由階段、資源、方法和整合所組成。階段和資源決定端點的路徑：

API 路徑格式

- `/prod/` - `prod` 階段和根資源。
- `/prod/user` - `prod` 階段和 `user` 資源。
- `/dev/{proxy+}` - `dev` 階段中的任何路由。
- `/` - (HTTP API) 預設階段和根資源。

Lambda 整合將路徑和 HTTP 方法組合映射到一個 Lambda 函數。您可以設定 API Gateway 依現狀傳遞 HTTP 請求的主體 (自訂整合)，或將請求主體封裝在一個包含所有請求資訊 (包括標頭、資源、路徑和方法) 的文件中。

如需設定詳細資訊，請參閱 Lambda proxy integrations in API Gateway。

事件格式

Amazon API Gateway 使用包含 HTTP 請求之 JSON 表示的事件來同步調用您的函數。對於自訂整合，事件是請求的本文。對於代理整合，事件具有已定義的結構。如需 API Gateway REST API 的代理事件範例，請參閱《API Gateway 開發人員指南》中的代理整合之 Lambda 函數的輸入格式。

回應格式

API Gateway 會等待函數的回應並將結果轉達給呼叫者。對於自訂整合，您可以定義整合回應和方法回應，將函數的輸出轉換為 HTTP 回應。對於代理整合，函數必須以特定格式的回應表示作出回應。

下列範例顯示來自 Node.js 函數的回應物件。回應物件表示包含 JSON 文件的成功 HTTP 回應。

Example index.mjs - 代理整合回應物件 (Node.js)

```
var response = {
      "statusCode": 200,
      "headers": {
        "Content-Type": "application/json"
      },
      "isBase64Encoded": false,
      "multiValueHeaders": {
        "X-Custom-Header": ["My value", "My other value"],
      },
      "body": "{\n  \"TotalCodeSize\": 104330022,\n  \"FunctionCount\": 26\n}"
    }
```

Lambda 執行時間會將回應物件序列化為 JSON，並將其傳送至 API。API 會剖析該回應並用它來建立 HTTP 回應，然後將其傳送到發出原始請求的用戶端。

Example HTTP 回應

```
< HTTP/1.1 200 OK
  < Content-Type: application/json
  < Content-Length: 55
  < Connection: keep-alive
  < x-amzn-RequestId: 32998fea-xmpl-4268-8c72-16138d629356
  < X-Custom-Header: My value
  < X-Custom-Header: My other value
  < X-Amzn-Trace-Id: Root=1-5e6aa925-ccecxmplbae116148e52f036
  <
  {
```

```
    "TotalCodeSize": 104330022,
    "FunctionCount": 26
  }
```

許可

Amazon API Gateway 會取得許可，從函數的以資源為基礎的政策中調用您的函數。您可以將呼叫許可授與整個 API，或將有限存取權授與階段、資源或方法。

當您透過使用 Lambda 主控台、使用 API Gateway 主控台或在 AWS SAM 範本中，將 API 新增至函數時，會自動更新函數的以資源為基礎的政策。範例函數政策範例如下。

Example函數政策

JSON

```
{
  "Version":"2012-10-17",
  "Id": "default",
  "Statement": [
    {
      "Sid": "nodejs-apig-functiongetEndpointPermissionProd-BWDBXMPLXE2F",
      "Effect": "Allow",
      "Principal": {
        "Service": "apigateway.amazonaws.com"
      },
      "Action": "lambda:InvokeFunction",
      "Resource": "arn:aws:lambda:us-east-2:111122223333:function:nodejs-apig-
function-1G3MXMPLXVXYI",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "111122223333"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:execute-api:us-
east-2:111122223333:ktyvxmpls1/*/GET/"
        }
      }
    }
  ]
}
```

您可以使用下列 API 操作來手動管理函數政策許可：

- AddPermission
- RemovePermission
- GetPolicy

若要將調用許可授與現有 API，請使用 `add-permission` 命令。範例：

```
aws lambda add-permission \
  --function-name my-function \
  --statement-id apigateway-get --action lambda:InvokeFunction \
  --principal apigateway.amazonaws.com \
  --source-arn "arn:aws:execute-api:us-east-2:123456789012:mnh1xmpli7/default/GET/"
```

您應該會看到下列輸出：

```
{
    "Statement": "{\"Sid\":\"apigateway-test-2\",\"Effect\":\"Allow\",\"Principal
\":{\"Service\":\"apigateway.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-east-2:123456789012:function:my-function
\",\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":\"arn:aws:execute-api:us-
east-2:123456789012:mnh1xmpli7/default/GET\"}}}"
}
```

> **Note**
>
> 如果您的函數和 API 位於不同的 中 AWS 區域，來源 ARN 中的區域識別符必須符合函數的區域，而不是 API 的區域。當 API Gateway 調用函數時，它會使用以 API 的 ARN 為基礎的資源 ARN，但會加以修改以符合函數的區域。

此範例中的來源 ARN 會將許可授與 API 的預設階段中根資源 GET 方法的整合 (ID 為 `mnh1xmpli7`)。您可以在來源 ARN 中使用星號，將許可授與多個階段、方法或資源。

資源模式

- `mnh1xmpli7/*/GET/*` - 所有階段中所有資源上的 GET 方法。
- `mnh1xmpli7/prod/ANY/user` - `prod` 階段中 `user` 資源上的 ANY 方法。

- `mnh1xmpli7/*/*/*` - 所有階段中所有資源上的任何方法。

如需檢視政策和移除陳述式的詳細資訊，請參閱 在 Lambda 中檢視資源型 IAM 政策。

範例應用程式

具有 Node.js 的 API Gateway 範例應用程式包含具有 範本的 AWS SAM 函數，該範本會建立已啟用 AWS X-Ray 追蹤的 REST API。它也包括用於部署、調用函數、測試 API 和清理的指令碼。

Powertools for AWS Lambda 的事件處理常式

撰寫 API Gateway 端點 (HTTP 或 REST) 調用的 Lambda 函數時，來自 Powertools for AWS Lambda 工具組的事件處理常式提供路由、中介軟體、CORS 組態、OpenAPI 規格產生、請求驗證、錯誤處理和其他實用功能。該事件處理常式公用程式適用於 Python 與 TypeScript/JavaScript。如需詳細資訊，請參閱 Powertools for AWS Lambda (Python) 文件中的事件處理常式 REST API，以及 Powertools for AWS Lambda (TypeScript) 文件中的事件處理常式 HTTP API。

Python

```
from aws_lambda_powertools import Logger
from aws_lambda_powertools.event_handler import APIGatewayRestResolver
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools.utilities.typing.lambda_context import LambdaContext

app = APIGatewayRestResolver()
logger = Logger()

@app.get("/healthz")
def ping():
    return {"message": "health status ok"}

@logger.inject_lambda_context(correlation_id_path=correlation_paths.API_GATEWAY_REST)
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

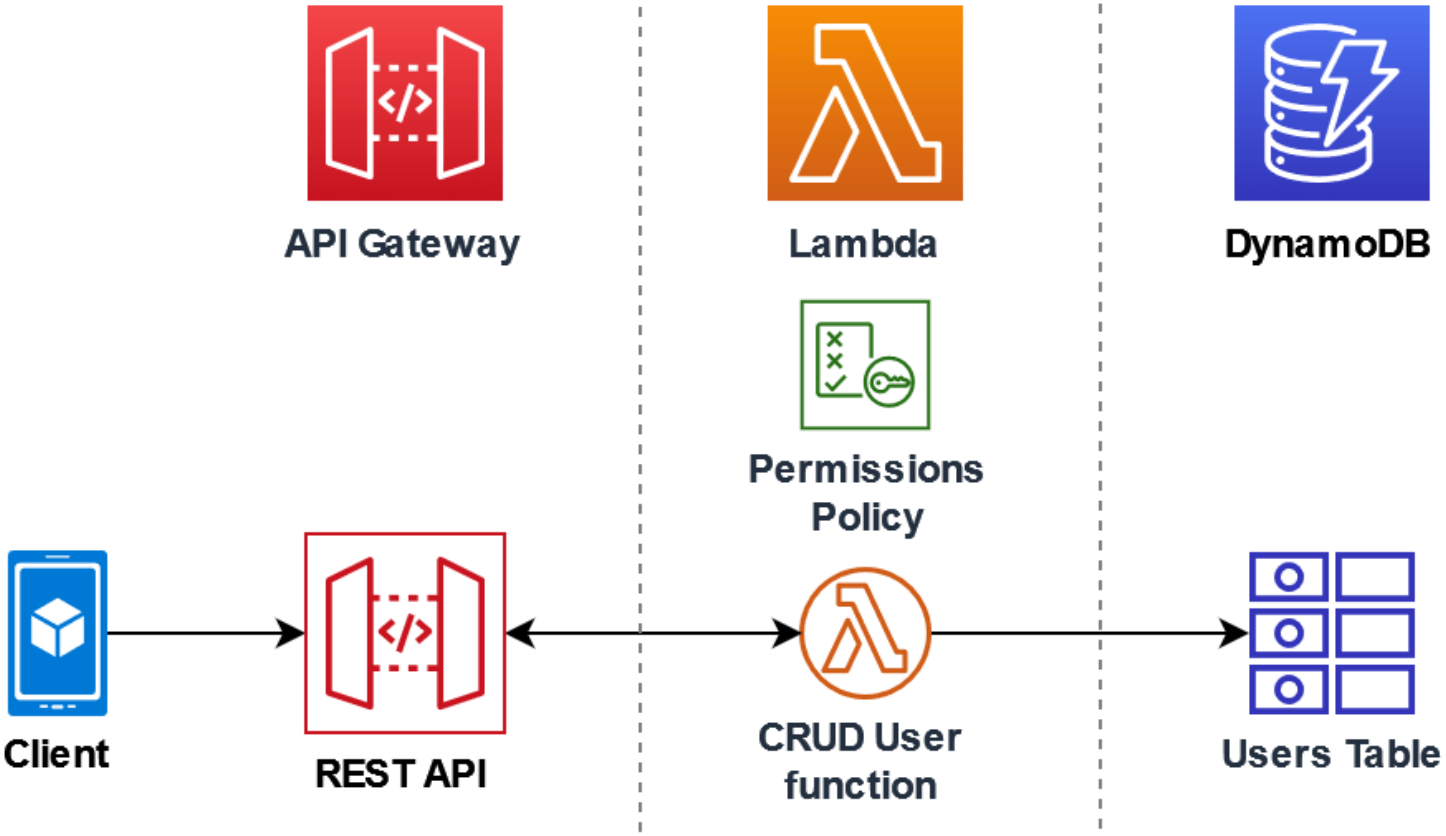
Typescript

```
import { Router } from '@aws-lambda-powertools/event-handler/experimental-rest';
import { Logger } from '@aws-lambda-powertools/logger';
```

```
import {
  correlationPaths,
  search,
} from '@aws-lambda-powertools/logger/correlationId';
import type { Context } from 'aws-lambda/handler';

const logger = new Logger({
  correlationIdSearchFn: search,
});

const app = new Router({ logger });

app.get("/healthz", async () => {
  return { message: "health status ok" };
});

export const handler = async (event: unknown, context: Context) => {
  // You can continue using other utilities just as before
  logger.addContext(context);
  logger.setCorrelationId(event, correlationPaths.API_GATEWAY_REST);
  return app.resolve(event, context);
};
```

教學課程：搭配使用 Lambda 與 API Gateway

在此教學課程中，您將建立 REST API，並透過此 API 調用 Lambda 函數。Lambda 函數會對 DynamoDB 資料表執行建立、讀取、更新及刪除 (CRUD) 操作。這裡提供的函數僅供示範，您將學習如何設定可調用任何 Lambda 函數的 API Gateway REST API。



使用 API Gateway 為使用者提供安全的 HTTP 端點以調用 Lambda 函數，並透過流量限流以及自動驗證和授權 API 呼叫，協助管理函數的大量呼叫。API Gateway 也提供使用 AWS Identity and Access Management (IAM) 和 Amazon Cognito 的彈性安全控制。對於需要預先授權才能呼叫應用程式的使用案例，這非常有用。

> **Tip**
>
> Lambda 提供兩種透過 HTTP 端點調用函數的方式：API Gateway 與 Lambda 函數 URL。如果不確定哪種方法最適合自己的使用案例，請參閱the section called “API Gateway 與函數 URL 比較”。

完成本教學課程需逐一進行以下階段：

1. 以 Python 或 Node.js 建立並設定 Lambda 函數，用於對 DynamoDB 資料表執行操作。
2. 在 API Gateway 中建立 REST API 以連接 Lambda 函數。
3. 建立 DynamoDB 資料表，然後在主控台中使用您的 Lambda 函數進行測試。
4. 在終端內使用 curl 部署 API 並測試完整設定。

完成這些階段後，您將了解如何使用 API Gateway 建立 HTTP 端點，以安全地調用任何規模的 Lambda 函數。您也會學習如何部署 API，以及如何在控制台中以及使用終端傳送 HTTP 請求來進行測試。

建立許可政策

在建立 Lambda 函數的執行角色之前，您必須先建立許可政策，以授予函數存取所需 AWS 資源的許可。在本教學課程中，政策允許 Lambda 對 DynamoDB 資料表執行 CRUD 操作，以及寫入 Amazon CloudWatch Logs。

建立政策

1. 開啟 IAM 主控台中的政策頁面。
2. 選擇建立政策。
3. 選擇 JSON 索引標籤，然後將下列政策貼到 JSON 編輯器。

   JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "Stmt1428341300017",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem",
        "dynamodb:Query",
        "dynamodb:Scan",
        "dynamodb:UpdateItem"
      ],
      "Effect": "Allow",
      "Resource": "*"
    },
    {
      "Sid": "",
      "Resource": "*",
      "Action": [
        "logs:CreateLogGroup",
        "logs:CreateLogStream",
        "logs:PutLogEvents"
```

```
      ],
      "Effect": "Allow"
    }
  ]
}
```

4. 選擇下一步：標籤。
5. 選擇下一步：檢閱。
6. 在檢閱政策下，針對政策名稱，輸入 **lambda-apigateway-policy**。
7. 選擇建立政策。

建立執行角色

執行角色是授予 Lambda 函數存取 AWS 服務 和資源許可的 AWS Identity and Access Management (IAM) 角色。若要讓函數對 DynamoDB 資料表執行操作，您需附加在上個步驟中建立的許可政策。

建立執行角色並附加自訂許可政策

1. 開啟 IAM 主控台中的角色頁面。
2. 選擇 建立角色。
3. 信任的實體類型請選擇 AWS 服務，使用案例則選擇 Lambda。
4. 選擇下一步。
5. 在政策搜尋方塊中，輸入 **lambda-apigateway-policy**。
6. 在搜尋結果中，選取您建立的政策 (`lambda-apigateway-policy`)，然後選擇下一步。
7. 在角色詳細資料底下，角色名稱請輸入 **lambda-apigateway-role**，然後選擇建立角色。

建立 Lambda 函式

1. 開啟 Lambda 主控台的函數頁面，然後選擇建立函數。
2. 選擇從頭開始撰寫。
3. 針對函數名稱，請輸入 `LambdaFunctionOverHttps`。
4. 在執行時期欄位中，選擇最新版本的 Node.js 或 Python 執行時期。
5. 在許可下，展開變更預設執行角色。
6. 選擇使用現有角色，然後選取先前建立的 **lambda-apigateway-role** 角色。
7. 選擇建立函數。

8. 在程式碼來源窗格中，以下列 Node.js 或 Python 程式碼取代預設程式碼。

Node.js

`region` 設定必須符合您部署函數和建立 DynamoDB 資料表 AWS 區域 的 。

Example index.mjs

```
import { DynamoDBDocumentClient, PutCommand, GetCommand,
         UpdateCommand, DeleteCommand} from "@aws-sdk/lib-dynamodb";
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";

const ddbClient = new DynamoDBClient({ region: "us-east-2" });
const ddbDocClient = DynamoDBDocumentClient.from(ddbClient);

// Define the name of the DDB table to perform the CRUD operations on
const tablename = "lambda-apigateway";

/**
 * Provide an event that contains the following keys:
 *
 *   - operation: one of 'create,' 'read,' 'update,' 'delete,' or 'echo'
 *   - payload: a JSON object containing the parameters for the table item
 *     to perform the operation on
 */
export const handler = async (event, context) => {

     const operation = event.operation;

     if (operation == 'echo'){
          return(event.payload);
     }

    else {
        event.payload.TableName = tablename;
        let response;

        switch (operation) {
          case 'create':
               response = await ddbDocClient.send(new
 PutCommand(event.payload));
               break;
          case 'read':
```

```
                response = await ddbDocClient.send(new
 GetCommand(event.payload));
                break;
            case 'update':
                response = ddbDocClient.send(new UpdateCommand(event.payload));
                break;
            case 'delete':
                response = ddbDocClient.send(new DeleteCommand(event.payload));
                break;
            default:
              response = 'Unknown operation: ${operation}';
            }
        console.log(response);
        return response;
    }
};
```

Python

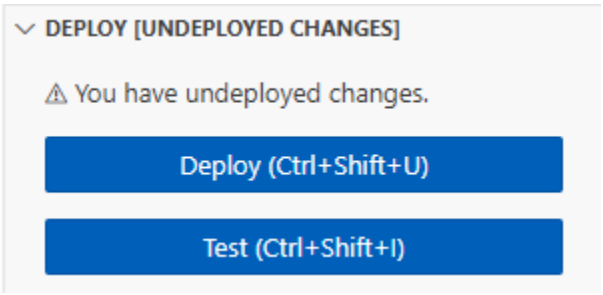
Example lambda_function.py

```
import boto3

# Define the DynamoDB table that Lambda will connect to
table_name = "lambda-apigateway"

# Create the DynamoDB resource
dynamo = boto3.resource('dynamodb').Table(table_name)

# Define some functions to perform the CRUD operations
def create(payload):
    return dynamo.put_item(Item=payload['Item'])

def read(payload):
    return dynamo.get_item(Key=payload['Key'])

def update(payload):
    return dynamo.update_item(**{k: payload[k] for k in ['Key',
 'UpdateExpression',
    'ExpressionAttributeNames', 'ExpressionAttributeValues'] if k in payload})

def delete(payload):
    return dynamo.delete_item(Key=payload['Key'])
```

```
def echo(payload):
    return payload

operations = {
    'create': create,
    'read': read,
    'update': update,
    'delete': delete,
    'echo': echo,
}

def lambda_handler(event, context):
    '''Provide an event that contains the following keys:
      - operation: one of the operations in the operations dict below
      - payload: a JSON object containing parameters to pass to the
        operation being performed
    '''

    operation = event['operation']
    payload = event['payload']

    if operation in operations:
        return operations[operation](payload)

    else:
        raise ValueError(f'Unrecognized operation "{operation}"')
```

> **Note**
>
> 在此範例中，DynamoDB 資料表的名稱定義為函數程式碼中的變數。在實際的應用程式中，最佳實務是將此參數做為環境變數來傳遞，並避免對資料表名稱進行硬式編碼。如需詳細資訊，請參閱使用 AWS Lambda 環境變數。

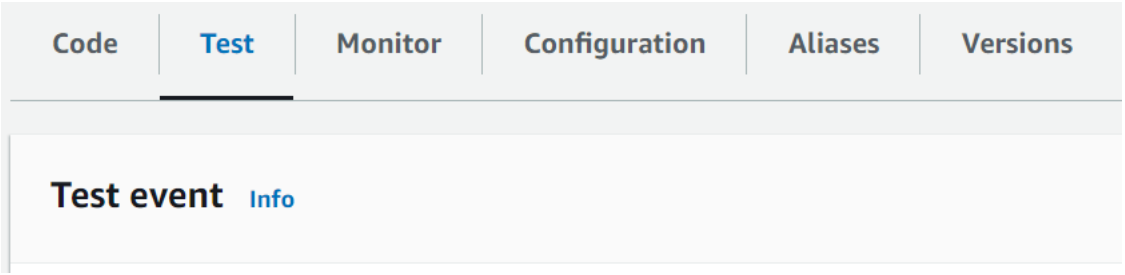
9. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



測試函數

在將函數與 API Gateway 整合之前，請確認已成功部署該函數。使用 Lambda 主控台將測試事件傳送至函式。

1. 在函式的 Lambda 主控台頁面中，選擇測試索引標籤。



2. 向下捲動至事件 JSON 區段，以下列內容取代預設事件。此事件符合 Lambda 函式預期的結構。

```
{
    "operation": "echo",
    "payload": {
        "somekey1": "somevalue1",
        "somekey2": "somevalue2"
    }
}
```

3. 選擇測試。
4. 在執行函式：成功欄位中，展開詳細資訊。您應該會看到下列回應：

```
{
  "somekey1": "somevalue1",
  "somekey2": "somevalue2"
}
```

使用 API Gateway 建立 REST API

在此步驟中，您將建立用來調用 Lambda 函數的 API Gateway REST API。

若要建立 API

1. 開啟 API Gateway 主控台。
2. 選擇 建立 API 。
3. 在 REST API 方塊中，選擇 建置 。
4. 在 API 詳細資訊下，讓新增 API維持在已選取的狀態，然後對於 API 名稱，輸入 **DynamoDBOperations**。
5. 選擇 建立 API 。

在 REST API 上建立資源

若要將 HTTP 方法新增到 API 中，首先需為該方法建立用來操作的資源。您可以在此建立資源來管理 DynamoDB 資料表。

若要建立資源

1. 在 API Gateway 主控台的 API 資源頁面上，選擇建立資源。
2. 在資源詳細資訊中，針對資源名稱輸入 **DynamoDBManager**。
3. 選擇 建立資源 。

建立 HTTP POST 方法

在此步驟中，您將為 DynamoDBManager 資源建立方法 (POST)。您需將此 POST 方法連結到 Lambda 函數，如此一來當方法收到 HTTP 請求，API Gateway 就會調用 Lambda 函數。

> **Note**
>
> 基於本教學課程的目的，會使用一個 HTTP 方法 (POST) 來調用單一 Lambda 函數，該函數會對 DynamoDB 資料表執行所有操作。在實際的應用程式中，最佳實務是針對每項操作使用不同的 Lambda 函數和 HTTP 方法。如需詳細資訊，請參閱無伺服器園地中的 The Lambda Monolith。

建立 POST 方法

1. 在 API 的資源頁面上，確定已反白選取 /DynamoDBManager 資源。然後，在方法窗格中，選擇建立方法。
2. 針對方法類型，選擇 POST。
3. 對於整合類型，讓 Lambda 函數維持在已選取的狀態。
4. 對於 Lambda 函數，請為函數 (LambdaFunctionOverHttps) 選擇 Amazon Resource Name (ARN)。
5. 選擇建立方法。

建立 DynamoDB 資料表

建立空白的 DynamoDB 資料表，Lambda 函數會對該資料表執行 CRUD 操作。

若要建立 DynamoDB 資料表

1. 開啟 DynamoDB 主控台的資料表頁面。
2. 選擇 建立資料表 。
3. 在 Table details (資料表詳細資訊) 下，執行下列動作：
   1. 對於 Table name (資料表名稱)，請輸入 **lambda-apigateway**。
   2. 對於 Partition key (分割區索引鍵)，輸入**id**，並保持資料類型設定為 String (字串)。
4. 在 Table settings (資料表設定) 下，保留 Default settings (預設設定)。
5. 選擇 建立資料表 。

測試 API Gateway、Lambda 和 DynamoDB 的整合

您現在已準備好測試 API Gateway API 方法與 Lambda 函數和 DynamoDB 資料表的整合。使用 API Gateway 主控台，您可以利用主控台的測試功能，將請求直接傳送至您的 POST 方法。在此步驟中，首先需使用 create 操作將新項目新增至 DynamoDB 資料表，然後使用 update 操作來修改項目。

測試 1：在 DynamoDB 資料表中建立新項目

1. 在 API Gateway 主控台中，選擇您的 API (DynamoDBOperations)。
2. 在 DynamoDBManager 資源下方，選擇 POST 方法。
3. 選擇測試標籤。您可能需要選擇向右箭頭按鈕才能顯示此索引標籤。

4. 在測試方法下，讓查詢字串和標頭留空。對於請求主體，貼上下列 JSON：

```
{
  "operation": "create",
  "payload": {
    "Item": {
      "id": "1234ABCD",
      "number": 5
    }
  }
}
```

5. 選擇 測試 。

   測試完成時顯示的結果應該會顯示 200 狀態。此狀態碼表示 create 操作成功。

   若要確認，您可以檢查 DynamoDB 資料表現在是否包含新項目。

6. 開啟 DynamoDB 主控台的 資料表頁面 ，然後選擇 lambda-apigateway 資料表。

7. 選擇 探索資料表項目 。在 Items returned (傳回的項目) 窗格中，應該會看到一個包含 id 1234ABCD 和 number 5 的項目。範例：

測試 2 ：更新 DynamoDB 資料表中的項目

1. 在 API Gateway 主控台中，返回到 POST 方法的測試分頁。

2. 在測試方法下，讓查詢字串和標頭留空。對於請求主體，貼上下列 JSON：

```
{
    "operation": "update",
    "payload": {
        "Key": {
            "id": "1234ABCD"
        },
```

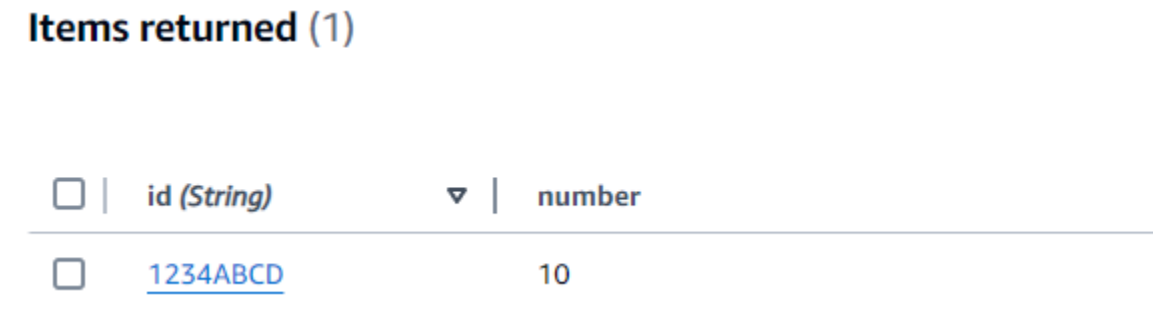
```
        "UpdateExpression": "SET #num = :newNum",
        "ExpressionAttributeNames": {
            "#num": "number"
        },
        "ExpressionAttributeValues": {
            ":newNum": 10
        }
    }
}
```

3. 選擇 測試 。

   測試完成時顯示的結果應該會顯示 200 狀態。此狀態碼表示 update 操作成功。

   若要確認，請檢查 DynamoDB 資料表中的項目是否已修改。

4. 開啟 DynamoDB 主控台的 資料表頁面 ，然後選擇 lambda-apigateway 資料表。
5. 選擇 探索資料表項目 。在 Items returned (傳回的項目) 窗格中，應該會看到一個包含 id 1234ABCD 和 number 10 的項目。



部署 API

為了讓用戶端能呼叫您的 API，您必須建立部署並建立相關聯的階段。階段代表 API 的快照，包括其方法和整合項目。

部署 API

1. 開啟 API Gateway 主控台中的 API 頁面，然後選擇 DynamoDBOperations API。
2. 在 API 的資源頁面上，選擇部署 API。
3. 對於階段，請選擇*新增階段*，然後在階段名稱輸入 **test**。
4. 選擇部署。

5. 在階段詳細資訊窗格中，複製調用 URL。您將在下一個步驟中使用此資料來透過 HTTP 請求調用函數。

使用 curl 來透過 HTTP 請求調用函數

您現在可以透過向 API 發出 HTTP 請求來調用 Lambda 函數。在此步驟中，您將在 DynamoDB 資料表中建立新項目，然後對該項目執行讀取、更新和刪除操作。

若要使用 curl 在 DynamoDB 資料表中建立項目

1. 在本機電腦上開啟終端或命令提示，並使用在上一個步驟中複製的調用 URL 執行下列 `curl` 命令。此命令使用下列選項：

   - `-H`：將自訂標頭新增至請求。此處用於指定內容類型為 JSON。
   - `-d`：在請求內文中傳送資料。此選項預設使用 HTTP POST 方法。

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   -d '{"operation": "create", "payload": {"Item": {"id": "5678EFGH", "number":
    15}}}'
   ```

   PowerShell

   ```
   curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
   DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\":
    \"create\", \"payload\": {\"Item\": {\"id\": \"5678EFGH\", \"number\": 15}}}'
   ```

   如果操作成功，您應該會看到傳回的回應及 HTTP 狀態碼 200。

2. 您也可以使用 DynamoDB 主控台執行下列步驟，驗證新項目是否在您的資料表中：

   1. 開啟 DynamoDB 主控台的 資料表頁面 ，然後選擇 `lambda-apigateway` 資料表。
   2. 選擇 探索資料表項目 。在 Items returned (傳回的項目) 窗格中，應該會看到一個包含 id `5678EFGH` 和 number `15` 的項目。

若要使用 curl 讀取 DynamoDB 資料表中的項目

- 在終端或命令提示中執行以下 `curl` 命令，讀取剛建立項目的值。使用您自己的調用 URL。

  Linux/macOS

  ```
  curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
   \
  -H "Content-Type: application/json" \
  -d '{"operation": "read", "payload": {"Key": {"id": "5678EFGH"}}}'
  ```

  PowerShell

  ```
  curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
  DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\": \"read
  \", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}}}'
  ```

  視您選擇 Node.js 或 Python 函數程式碼而定，您應該會看到類似下列其中一項的輸出：

  Node.js

  ```
  {"$metadata":
  {"httpStatusCode":200,"requestId":"7BP3G5Q0C0O1E50FBQI9NS099JVV4KQNSO5AEMVJF66Q9ASUAAJG"
  "attempts":1,"totalRetryDelay":0},"Item":{"id":"5678EFGH","number":15}}
  ```

  Python

  ```
  {"Item":{"id":"5678EFGH","number":15},"ResponseMetadata":
  {"RequestId":"QNDJICE52E86B82VETR6RKBE5BVV4KQNSO5AEMVJF66Q9ASUAAJG",
  "HTTPStatusCode":200,"HTTPHeaders":{"server":"Server","date":"Wed, 31 Jul 2024
   00:37:01 GMT","content-type":"application/x-amz-json-1.0",
  "content-length":"52","connection":"keep-alive","x-amzn-
  requestid":"QNDJICE52E86B82VETR6RKBE5BVV4KQNSO5AEMVJF66Q9ASUAAJG","x-amz-
  crc32":"2589610852"},
  "RetryAttempts":0}}
  ```

若要使用 curl 更新 DynamoDB 資料表中的項目

1. 在終端或命令提示中執行下列 `curl` 命令，透過變更 `number` 值來更新剛建立的項目。使用您自己的調用 URL。

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   -d '{"operation": "update", "payload": {"Key": {"id": "5678EFGH"},
    "UpdateExpression": "SET #num = :new_value", "ExpressionAttributeNames":
    {"#num": "number"}, "ExpressionAttributeValues": {":new_value": 42}}}'
   ```

   PowerShell

   ```
   curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
   DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\":
    \"update\", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}, \"UpdateExpression
   \": \"SET #num = :new_value\", \"ExpressionAttributeNames\": {\"#num\": \"number
   \"}, \"ExpressionAttributeValues\": {\":new_value\": 42}}}'
   ```

2. 若要確認項目的 `number` 值已更新，請執行另一個讀取命令：

   Linux/macOS

   ```
   curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
    \
   -H "Content-Type: application/json" \
   -d '{"operation": "read", "payload": {"Key": {"id": "5678EFGH"}}}'
   ```

   PowerShell

   ```
   curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
   DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\": \"read
   \", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}}}'
   ```

若要使用 curl 刪除 DynamoDB 資料表中的項目

1. 在終端或命令提示中，執行以下 `curl` 命令，刪除剛建立的項目。使用您自己的調用 URL。

Linux/macOS

```
curl https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/DynamoDBManager
 \
-H "Content-Type: application/json" \
-d '{"operation": "delete", "payload": {"Key": {"id": "5678EFGH"}}}'
```

PowerShell

```
curl.exe 'https://l8togsqxd8.execute-api.us-east-2.amazonaws.com/test/
DynamoDBManager' -H 'Content-Type: application/json' -d '{\"operation\":
 \"delete\", \"payload\": {\"Key\": {\"id\": \"5678EFGH\"}}}'
```

2. 確認刪除操作成功。在 DynamoDB 主控台探索項目頁面的已傳回項目窗格中，確認具有 id 5678EFGH 的項目已不存在於資料表中。

清除資源 (選用)

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。透過刪除您不再使用 AWS 的資源，您可以避免不必要的 費用 AWS 帳戶。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **confirm**，然後選擇刪除。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇刪除。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 API

1. 開啟 API Gateway 主控台中的 API 頁面 。
2. 選取您建立的 API。
3. 選擇 動作 、刪除 。
4. 選擇 刪除 。

若要刪除 DynamoDB 資料表

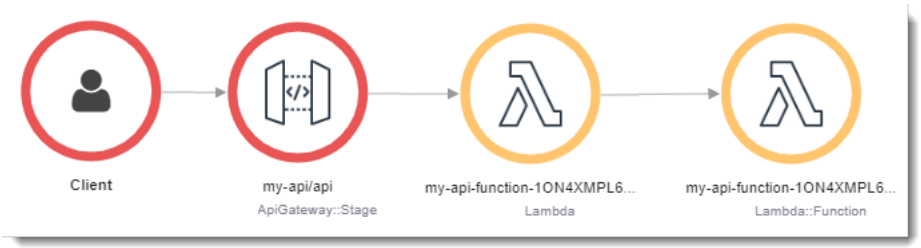
1. 開啟 DynamoDB 主控台的 資料表頁面 。
2. 選取您建立的資料表。
3. 選擇 刪除 。
4. 在文字方塊中輸入 **`delete`**。
5. 選擇 刪除資料表 。

使用 API Gateway API 處理 Lambda 錯誤

API Gateway 會將所有調用和函數錯誤視為內部錯誤。如果 Lambda API 拒絕調用請求，則 API Gateway 會傳回 500 錯誤代碼。如果函數執行但傳回錯誤，或傳回格式錯誤的回應，API Gateway 會傳回 502。在這兩種情況下，API Gateway 的回應主體為 `{"message": "Internal server error"}`。

> **Note**
>
> API Gateway 不會重試任何 Lambda 調用。如果 Lambda 傳回錯誤，API Gateway 會將錯誤回應傳回至用戶端。

下列範例顯示導致函數錯誤和 API Gateway 傳回 502 的請求的 X-Ray 流程圖。用戶端會收到一般錯誤訊息。

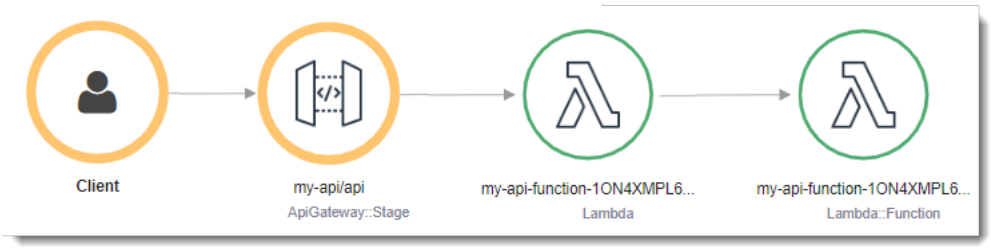


若要自訂錯誤回應，您必須在程式碼中發現錯誤，並以必要的格式格式化回應。

Example index.mjs - 格式錯誤

```
var formatError = function(error){
  var response = {
    "statusCode": error.statusCode,
    "headers": {
      "Content-Type": "text/plain",
      "x-amzn-ErrorType": error.code
    },
    "isBase64Encoded": false,
    "body": error.code + ": " + error.message
  }
  return response
}
```

API Gateway 將此回應轉換為具有自定義狀態碼和主體的 HTTP 錯誤。在流程圖中，函數節點是綠色的，因為它會處理錯誤。



選取一種使用 HTTP 請求調用 Lambda 函數的方法

Lambda 的許多常見使用案例涉及使用 HTTP 請求調用您的函數。例如，您可能希望 Web 應用程式透過瀏覽器請求調用您的函式。Lambda 函式也可用於建立完整的 REST API、處理來自行動應用程式的使用者互動、透過 HTTP 呼叫處理來自外部服務的資料，或建立自訂 Webhook。

以下各節說明透過 HTTP 調用 Lambda 的選擇，並提供資訊以協助您針對特定使用案例做出正確決策。

選取 HTTP 調用方法時，您有哪些選擇？

Lambda 提供兩種主要方法來使用 HTTP 請求調用函數 - 函數 URL 和 API Gateway。這兩種選項的主要差異如下所示：

- Lambda 函數 URL 可為 Lambda 函數提供簡單、直接的 HTTP 端點。已針對簡單性和成本效益對其進行最佳化，並提供透過 HTTP 公開 Lambda 函數的最快路徑。
- API Gateway 是一種更進階的服務，用於建置功能完整的 API。API Gateway 已針對大規模建置和管理生產 API 進行最佳化，並提供完整的安全、監控和流量管理工具。

您已知道自己需求時的建議

如果您已經清楚自己的需求，以下是基本建議：

建議函數 URL 用於簡單的應用程式或原型設計，其中您只需要基本的身分驗證方法和請求/回應處理，並想要將成本和複雜性降至最低。

API Gateway 是大規模生產應用程式或需要更進階功能的情況的更佳選擇，例如 OpenAPI Description 支援、身分驗證選項、自訂網域名稱或豐富的請求/回應處理，包括限流、快取和請求/回應轉換。

選擇調用 Lambda 函數的方法時應考慮的事項

在函數 URL 和 API Gateway 之間選取時，需要考慮下列因素：

- 您的身分驗證需要，例如是否需要 OAuth 或 Amazon Cognito 來驗證使用者
- 您的擴展需求以及您要實作之 API 的複雜性
- 您是否需要進階功能，例如請求驗證和請求/回應格式化
- 您的監控需求
- 您的成本目標

透過了解這些因素，可以選擇最能平衡您的安全性、複雜性和成本需求的選項。

下列資訊摘要說明兩個選項之間的主要差異。

身分驗證

- 函數 URLs提供基本身分驗證選項。 AWS Identity and Access Management 可以將端點設定為公有 (無身分驗證) 或需要 IAM 身分驗證。透過 IAM 身分驗證，您可以使用標準 AWS 登入資料或 IAM 角色來控制存取。雖然設定簡單，但相較於其他驗證方法，此方法提供的選項有限。
- API Gateway 可存取更全面的身分驗證選項。除了 IAM 身分驗證之外，也可以使用 Lambda 授權工具 (自訂身分驗證邏輯)、Amazon Cognito 使用者集區和 OAuth2.0 流程。此彈性可讓您實作複雜的身分驗證機制，包括第三方身分驗證提供者、權杖型身分驗證和多重要素身分驗證。

請求/回應處理

- 函數 URL 提供基本的 HTTP 請求和回應處理。它們支援標準 HTTP 方法，並包含內建的跨來源資源共用 (CORS) 支援。雖然他們可以自然處理 JSON 承載和查詢參數，但它們不提供請求轉換或驗證功能。回應處理同樣簡單 – 用戶端接收來自 Lambda 函數的回應，就像 Lambda 傳回它一樣。
- API Gateway 可提供複雜的請求和回應處理功能。您可以定義請求驗證器，使用映射範本轉換請求和回應，設定請求/回應標頭，以及實作回應快取。API Gateway 也支援二進位承載和自訂網域名稱，並且可以在回應到達用戶端之前對其進行修改。可以使用 JSON 結構描述來設定請求/回應驗證和轉換的模型。

擴展

- 函數 URL 會根據 Lambda 函數的並行限制直接擴展，並透過將函數擴展到其設定的並行限制上限來處理流量尖峰。達到該限制後，Lambda 會使用 HTTP 429 回應來回應其他請求。沒有內建佇列機制，因此處理擴展完全取決於 Lambda 函數的組態。根據預設，Lambda 函數每個 有 1，000 個並行執行的限制 AWS 區域。
- 除了 Lambda 自己的擴展之外，API Gateway 還提供其他擴展功能。它包含內建的請求佇列和限流控制，可讓您更輕鬆地管理流量尖峰。根據預設，API Gateway 每秒最多可以處理 10,000 個請求，高載容量為每秒 5,000 個請求。它也提供在不同層級調節請求的工具 (API、階段或方法)，以保護後端。

監控

- 函數 URL 透過 Amazon CloudWatch 指標提供基本監控，包括請求計數、延遲和錯誤率。可以存取標準 Lambda 指標和日誌，它們會顯示傳入函數的原始請求。雖然這可提供基本的操作可見性，但指標主要著重於函數執行。
- API Gateway 提供全面的監控功能，包括詳細的指標、記錄和追蹤選項。可以透過 CloudWatch 來監控 API 呼叫、延遲、錯誤率和快取命中率/遺失率。API Gateway 也會整合 與 AWS X-Ray 以進行分散式追蹤，並提供可自訂的記錄格式。

Cost

- 函數 URL 遵循標準 Lambda 定價模型 – 您只需支付函數調用和運算時間的費用。URL 端點本身不收取額外費用。如果您不需要 API Gateway 的其他功能，這對於簡單的 API 或低流量應用程式而言是一個具成本效益的選擇。

- API Gateway 提供免費方案，其中包含針對 REST API 收到的一百萬個 API 呼叫，以及針對 HTTP API 收到的一百萬個 API 呼叫。之後，API Gateway 會針對 API 呼叫、資料傳輸和快取 (如果啟用) 收取費用。請參閱 API Gateway 定價頁面，了解您自己的使用案例費用。

其他功能

- 函數 URL 旨在實現簡便性和直接 Lambda 整合。它們支援 HTTP 和 HTTPS 端點，提供內建 CORS 支援，並提供雙堆疊 (IPv4 和 IPv6) 端點。雖然它們缺乏進階功能，但它們在您需要快速、直接地透過 HTTP 公開 Lambda 函數時表現卓越。
- API Gateway 包含許多其他功能，例如 API 版本控制、階段管理、用於用量計劃的 API 金鑰、透過 Swagger/OpenAPI 的 API 文件、WebSocket API、VPC 中的私有 API 以及 WAF 整合，以提供額外的安全性。它還支援 Canary 部署、用於測試的模擬整合，以及與 Lambda AWS 服務 以外的其他整合。

選取調用 Lambda 函數的方法

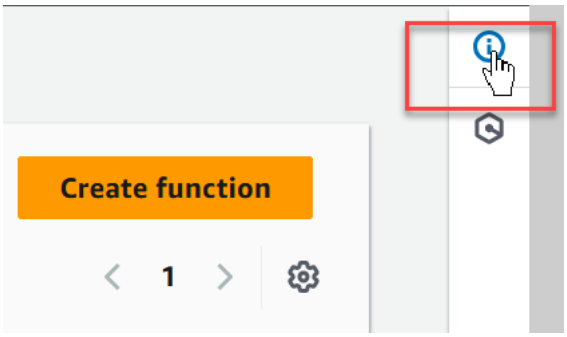
現在您已了解在 Lambda 函數 URL 和 API Gateway 之間進行選擇的條件，以及它們之間的主要差異，您可以選擇最符合您需求的選項，並使用下列資源來協助您開始使用。

Function URLs

使用下列資源開始使用函數 URL

- 遵循教學課程建立具有函數 URL 的 Lambda 函數
- 在本指南的 the section called “函數 URL” 章節中進一步了解 函數 URL
- 透過執行以下操作，嘗試主控台內的引導式教學課程建立簡單的 Web 應用程式：

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇畫面右上角的圖示以開啟說明面板。



3. 選取教學課程。
4. 在建立簡單的 Web 應用程式中，選擇開始教學課程。

API Gateway

使用下列資源開始使用 Lambda 和 API Gateway

- 依照教學課程搭配使用 Lambda 與 API Gateway 來建立與後端 Lambda 函數整合的 REST API。
- 在 Amazon API Gateway 開發人員指南的下列章節中，進一步了解 API Gateway 提供的不同類型 API：
  - API Gateway REST API
  - API Gateway HTTP API
  - API Gateway WebSocket API
- 請嘗試 Amazon API Gateway 開發人員指南的教學課程和研討會一節中的一個或多個範例。

AWS Lambda 搭配 使用 AWS Infrastructure Composer

AWS Infrastructure Composer 是一種視覺化建置器，用於卸載現代應用程式 AWS。您可以透過在視覺化畫布 AWS 服務 中拖曳、分組和連線來設計應用程式架構。Infrastructure Composer 會從您的設計建立基礎設施即程式碼 (IaC) 範本，您可以使用 AWS SAM 或 CloudFormation 部署。

將 Lambda 函數匯出至 Infrastructure Composer

您可以使用 Lambda 主控台根據現有 Lambda 函數的組態建立新專案，開始使用 Infrastructure Composer。若要將函數的組態和程式碼匯出至 Infrastructure Composer 以建立新專案，請執行下列動作：

1. 開啟 Lambda 主控台中的函數頁面。
2. 選取您希望使用做為 Infrastructure Composer 專案基礎的函數。
3. 在函數概觀窗格中，選擇匯出至 Infrastructure Composer。

   若要將函數的組態和程式碼匯出至 Infrastructure Composer，Lambda 會在您的帳戶中建立 Amazon S3 儲存貯體來暫時存放此資料。
4. 在對話方塊中，選擇確認並建立專案以接受此儲存貯體的預設名稱，並將函數的設定和程式碼匯出至 Infrastructure Composer。

5. (選擇性) 若要為 Lambda 建立的 Amazon S3 儲存貯體選擇其他名稱，請輸入新名稱，然後選擇確認並建立專案。Amazon S3 儲存貯體的名稱必須是全域唯一的，並遵循儲存貯體命名規則。
6. 若要在 Infrastructure Composer 中儲存專案和函數檔案，請啟用本機同步模式。

> **Note**
>
> 如果您之前已使用匯出至應用程式編寫器功能，並使用預設名稱建立 Amazon S3 儲存貯體，則 Lambda 可以重複使用此儲存貯體 (如果儲存貯體仍然存在)。接受對話方塊中的預設儲存貯體名稱，以重新使用現有儲存貯體。

Amazon S3 傳輸儲存貯體組態

Lambda 建立用來傳輸函數組態的 Amazon S3 儲存貯體，會使用 AES 256 加密標準的自動加密物件。Lambda 也會將儲存貯體設定為使用儲存貯體擁有者條件，以確保只有您的 AWS 帳戶 能夠將物件新增至儲存貯體。

Lambda 會將儲存貯體設定為在上傳物件 10 天後自動刪除物件。但是，Lambda 不會自動刪除儲存貯體本身。若要從 中刪除儲存貯體 AWS 帳戶，請遵循刪除儲存貯體中的指示。預設儲存貯體名稱使用字首 `lambdasam`、10 位數英數字串，以及 AWS 區域 您在其中建立函數的 ：

```
lambdasam-06f22da95b-us-east-1
```

為避免將額外費用新增至您的 AWS 帳戶，我們建議您在完成將函數匯出至 Infrastructure Composer 後，立即刪除 Amazon S3 儲存貯體。

適用標準 Amazon S3 定價。

所需的許可

若要使用 Lambda 與 Infrastructure Composer 功能的整合，您需要特定許可才能下載 AWS SAM 範本，並將函數的組態寫入 Amazon S3。

若要下載 AWS SAM 範本，您必須具有使用下列 API 動作的許可：

- GetPolicy
- iam:GetPolicyVersion
- iam:GetRole

- iam:GetRolePolicy
- iam:ListAttachedRolePolicies
- iam:ListRolePolicies
- iam:ListRoles

您可以將 `AWSLambda_ReadOnlyAccess` AWS 受管政策新增至 IAM 使用者角色，以授予使用所有這些動作的許可。

若要讓 Lambda 將函數的組態寫入 Amazon S3，您必須擁有使用下列 API 動作的許可：

- S3:PutObject
- S3:CreateBucket
- S3:PutBucketEncryption
- S3:PutBucketLifecycleConfiguration

如果您無法將函數組態匯出至 Infrastructure Composer，請檢查您的帳戶是否具有執行這些操作所需要的許可。如果您擁有所需要的許可，但仍然無法匯出函數組態，請檢查任何可能會限制 Amazon S3 存取的資源型政策。

其他資源

如需取得如何根據現有 Lambda 函數在 Infrastructure Composer 中設計無伺服器應用程式的詳細教學課程，請參閱the section called “基礎設施即程式碼 (IaC)”。

若要使用 Infrastructure Composer 和使用 Lambda AWS SAM 設計和部署完整的無伺服器應用程式，您也可以遵循AWS 無伺服器模式研討會中的AWS Infrastructure Composer 教學課程。

AWS Lambda 搭配 使用 CloudFormation

在 AWS CloudFormation 範本中，您可以將 Lambda 函數指定為自訂資源的目標。使用自訂資源來處理參數、擷取組態值，或在堆疊生命週期事件 AWS 服務 期間呼叫其他 。

下列範例叫用在範本中其他地方定義的函數。

Example- 自訂資源定義

```
Resources:
  primerinvoke:
    Type: AWS::CloudFormation::CustomResource
    Version: "1.0"
    Properties:
      ServiceToken: !GetAtt primer.Arn
      FunctionName: !Ref randomerror
```

服務權杖是函數的 Amazon Resource Name (ARN)，在您建立、更新或刪除堆疊時 CloudFormation 叫用該函數。您也可以包含其他屬性，例如 FunctionName，這些屬性會照原樣 CloudFormation 傳遞給您的函數。

CloudFormation 使用包含回呼 URL 的事件非同步叫用 Lambda 函數。

Example– CloudFormation 訊息事件

```
{
    "RequestType": "Create",
    "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
    "ResponseURL": "https://cloudformation-custom-resource-response-useast1.s3-us-
east-1.amazonaws.com/arn%3Aaws%3Acloudformation%3Aus-east-1%3A123456789012%3Astack/
lambda-error-processor/1134083a-2608-1e91-9897-022501a2c456%7Cprimerinvoke
%7C5d478078-13e9-baf0-464a-7ef285ecc786?
AWSAccessKeyId=AKIAIOSFODNN7EXAMPLE&Expires=1555451971&Signature=28UijZePE5I4dvukKQqM
%2F9Rf1o4%3D",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
    "LogicalResourceId": "primerinvoke",
    "ResourceType": "AWS::CloudFormation::CustomResource",
    "ResourceProperties": {
```

```
        "ServiceToken": "arn:aws:lambda:us-east-1:123456789012:function:lambda-error-
processor-primer-14ROR2T3JKU66",
        "FunctionName": "lambda-error-processor-randomerror-ZWUC391MQAJK"
    }
}
```

函式負責回傳回應到回呼 URL，說明呼叫成功或失敗。如需完整的回應語法，請參閱自訂資源回應物件。

Example– CloudFormation 自訂資源回應

```
{
    "Status": "SUCCESS",
    "PhysicalResourceId": "2019/04/18/[$LATEST]b3d1bfc65f19ec610654e4d9b9de47a0",
    "StackId": "arn:aws:cloudformation:us-east-1:123456789012:stack/lambda-error-
processor/1134083a-2608-1e91-9897-022501a2c456",
    "RequestId": "5d478078-13e9-baf0-464a-7ef285ecc786",
    "LogicalResourceId": "primerinvoke"
}
```

CloudFormation 提供名為 的程式庫`cfn-response`，可處理傳送回應。如果您在範本中定義函數，您可以要求程式庫名稱。 CloudFormation 然後， 會將程式庫新增至其為函數建立的部署套件。

如果自訂資源使用的函數已連接彈性網路介面，則請將下列資源新增至 VPC 政策，其中 **region** 是函數所在的區域 (不含破折號)。例如，`us-east-1` 為 `useast1`。這將允許自訂資源回應將訊號傳回 CloudFormation 堆疊的回呼 URL。

```
arn:aws:s3:::cloudformation-custom-resource-response-region",
"arn:aws:s3:::cloudformation-custom-resource-response-region/*",
```

下列的範例函式會叫用第二個函式。如果呼叫成功，函數會傳送成功回應給 CloudFormation，且堆疊更新會繼續。範本使用 AWS Serverless Application Model提供的 AWS::Serverless::Function 資源類型。

Example– 自訂資源函數

```
Transform: 'AWS::Serverless-2016-10-31'
Resources:
  primer:
    Type: AWS::Serverless::Function
    Properties:
```

```
      Handler: index.handler
      Runtime: nodejs16.x
      InlineCode: |
        var aws = require('aws-sdk');
        var response = require('cfn-response');
        exports.handler = function(event, context) {
            // For Delete requests, immediately send a SUCCESS response.
            if (event.RequestType == "Delete") {
                response.send(event, context, "SUCCESS");
                return;
            }
            var responseStatus = "FAILED";
            var responseData = {};
            var functionName = event.ResourceProperties.FunctionName
            var lambda = new aws.Lambda();
            lambda.invoke({ FunctionName: functionName }, function(err, invokeResult) {
                if (err) {
                    responseData = {Error: "Invoke call failed"};
                    console.log(responseData.Error + ":\n", err);
                }
                else responseStatus = "SUCCESS";
                response.send(event, context, responseStatus, responseData);
            });
        };
      Description: Invoke a function to create a log stream.
      MemorySize: 128
      Timeout: 8
      Role: !GetAtt role.Arn
      Tracing: Active
```

如果未在範本中定義自訂資源叫用的函數，您可以從 AWS CloudFormation 《 使用者指南》中的 `cfn-response` cfn-response 模組取得 的原始程式碼。

如需自訂資源的詳細資訊，請參閱 AWS CloudFormation 使用者指南中的自訂資源。

使用 Lambda 處理 Amazon DocumentDB 事件

您可以透過將 Amazon DocumentDB 叢集設定為事件來源，使用 Lambda 函數處理 Amazon DocumentDB (with MongoDB compatibility) 變更串流中的事件。接著，您可以在每次使用 Amazon DocumentDB 叢集變更資料時，調用 Lambda 函數來自動執行事件驅動的工作負載。

> **Note**
>
> Lambda 僅支援 Amazon DocumentDB 4.0 和 5.0 版。Lambda 不支援 3.6 版。
> 此外，針對事件來源映射，Lambda 僅支援執行個體型叢集和區域叢集。Lambda 不支援 彈性叢集 或 全域叢集。使用 Lambda 做為用戶端連線至 Amazon DocumentDB 時不適用此限制。Lambda 可以連線至所有叢集類型來執行 CRUD 操作。

Lambda 會依照抵達的順序處理來自 Amazon DocumentDB 變更串流的事件。因此，函數一次只能處理來自 Amazon DocumentDB 的一個並行調用。若要監控函數，您可以追蹤其並行指標。

> **Warning**
>
> Lambda 事件來源映射至少會處理每個事件一次，而且可能會重複處理記錄。為避免與重複事件相關的潛在問題，強烈建議您讓函數程式碼具有等冪性。若要進一步了解，請參閱 AWS 知識中心中的如何使 Lambda 函數具有等冪性。

主題

- Amazon DocumentDB 事件範例
- 先決條件和許可
- 設定網路安全
- 建立 Amazon DocumentDB 事件來源映射 (主控台)
- 建立 Amazon DocumentDB 事件來源映射 (SDK 或 CLI)
- 輪詢和串流開始位置
- 監控 Amazon DocumentDB 事件來源
- 教學課程： AWS Lambda 搭配 Amazon DocumentDB Streams 使用

Amazon DocumentDB 事件範例


```
{
    "eventSourceArn": "arn:aws:rds:us-
east-1:123456789012:cluster:canaryclusterb2a659a2-qo5tcmqkcl03",
    "events": [
        {
            "event": {
                "_id": {
                    "_data": "0163eeb6e7000000090100000009000041e1"
                },
                "clusterTime": {
                    "$timestamp": {
                        "t": 1676588775,
                        "i": 9
                    }
                },
                "documentKey": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    }
                },
                "fullDocument": {
                    "_id": {
                        "$oid": "63eeb6e7d418cd98afb1c1d7"
                    },
                    "anyField": "sampleValue"
                },
                "ns": {
                    "db": "test_database",
                    "coll": "test_collection"
                },
                "operationType": "insert"
            }
        }
    ],
    "eventSource": "aws:docdb"
}
```


如需有關此範例中事件及其形狀的詳細資訊，請參閱 MongoDB 文件網站上的變更事件。

先決條件和許可

將 Amazon DocumentDB 作為 Lambda 函數的事件來源使用前，請留意以下先決條件。您必須：

- 在與函數相同的 AWS 帳戶 和 中擁有現有的 Amazon DocumentDB AWS 區域 叢集。如果您沒有現有叢集，則可以按照《Amazon DocumentDB 開發人員指南》中開始使用 Amazon DocumentDB所述步驟建立叢集。或者，教學課程： AWS Lambda 搭配 Amazon DocumentDB Streams 使用 中的第一組步驟會引導您建立包含所有必要先決條件的 Amazon DocumentDB 叢集。
- 允許 Lambda 存取與您 Amazon DocumentDB 叢集相關聯的 Amazon Virtual Private Cloud (Amazon VPC) 資源。如需詳細資訊，請參閱設定網路安全。
- 在您的 Amazon DocumentDB 叢集上啟用 TLS。這是預設設定。若停用 TLS，Lambda 將無法與您的叢集進行通訊。
- 啟用 Amazon DocumentDB 叢集上的變更串流。如需詳細資訊，請參閱《Amazon DocumentDB 開發人員指南》中的透過 Amazon DocumentDB 使用變更串流 。
- 為 Lambda 提供憑證以存取您的 Amazon DocumentDB 叢集。設定事件來源時，請提供包含存取叢集所需驗證詳細資料 (使用者名稱和密碼) 的 AWS Secrets Manager 金鑰。若要在設定期間提供此金鑰，請執行下列其中一項操作：
  - 如果您要使用 Lambda 主控台進行設定，請在 Secrets Manager 金鑰欄位中提供此金鑰。
  - 如果您使用 AWS Command Line Interface (AWS CLI) 進行設定，請在 `source-access-configurations`選項中提供此金鑰。您可以連同 `create-event-source-mapping` 或 `update-event-source-mapping` 命令包括此選項。例如：

    ```
    aws lambda create-event-source-mapping \
        ...
        --source-access-configurations
     '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
    west-2:123456789012:secret:DocDBSecret-AbC4E6"}]' \
        ...
    ```
- 您必須向 Lambda 授與許可，才能管理與 Amazon DocumentDB 串流相關的資源。將下列許可手動新增到函數的 執行角色：
  - rds:DescribeDBClusters
  - rds:DescribeDBClusterParameters
  - rds:DescribeDBSubnetGroups
  - ec2:CreateNetworkInterface
  - ec2:DescribeNetworkInterfaces

- ec2:DescribeVpcs
- ec2:DeleteNetworkInterface
- ec2:DescribeSubnets
- ec2:DescribeSecurityGroups
- kms:Decrypt
- secretsmanager:GetSecretValue

- 傳送至 Lambda 的 Amazon DocumentDB 變更串流事件大小請勿超過 6 MB。Lambda 支援的承載大小上限為 6MB。如果變更串流嘗試向 Lambda 傳送大於 6MB 的事件，Lambda 會捨棄訊息並發出 `OversizedRecordCount` 指標。Lambda 會盡力發送所有指標。

> **Note**
>
> 雖然 Lambda 函數的逾時上限通常為 15 分鐘，但 Amazon MSK、自我管理的 Apache Kafka、Amazon DocumentDB 以及 Amazon MQ for ActiveMQ 和 Amazon MQ for RabbitMQ 的事件來源映射只支援 14 分鐘逾時限制上限的函數。此限制條件可確保事件來源映射能夠正確處理函數錯誤和重試。

設定網路安全

若要透過事件來源映射授予 Lambda 對 Amazon DocumentDB 的完整存取權，您的叢集必須使用公有端點 (公有 IP 位址)，或者您必須提供建立叢集之 Amazon VPC 的存取權。

當您將 Amazon DocumentDB 與 Lambda 搭配使用時，請建立 AWS PrivateLink VPC 端點，為您的函數提供 Amazon VPC 中資源的存取權。

> **Note**
>
> AWS PrivateLink 具有使用事件輪詢器預設（隨需）模式之事件來源映射的函數需要 VPC 端點。如果您的事件來源映射使用佈建模式，則不需要設定 AWS PrivateLink VPC 端點。

建立端點以提供對下列資源的存取權：

- Lambda：為 Lambda 服務主體建立端點。
- AWS STS — 為 建立端點， AWS STS 以便服務主體代表您擔任角色。

- Secrets Manager：如果您的叢集使用 Secrets Manager 來儲存憑證，則請為 Secrets Manager 建立端點。

或者，在 Amazon VPC 中的每個公有子網路上設定一個 NAT 閘道。如需詳細資訊，請參閱the section called “Lambda 函數的網際網路存取”。

當您為 Amazon DocumentDB 建立事件來源映射時，Lambda 會檢查是否已存在彈性網絡介面 (ENI)，適用於為您的 Amazon VPC 設定的子網路和安全群組。如果 Lambda 找到現有的 ENI，它會嘗試重複使用它們。否則，Lambda 會建立新的 ENI 以連線至事件來源並調用您的函數。

> **Note**
>
> Lambda 函數一律會在 Lambda 服務所擁有的 VPC 內執行。函數的 VPC 組態不會影響事件來源映射。只有事件來源的聯網組態會決定 Lambda 如何連線至您的事件來源。

為包含叢集的 Amazon VPC 設定安全群組。依預設，Amazon DocumentDB 會使用下列連接埠：27017。

- 傳入規則：允許與事件來源相關聯之安全群組的預設代理程式連接埠上的所有流量。或者，您可使用自我參照安全群組規則，允許來自同一安全群組內其他執行個體的存取。
- 傳出規則 – 如果您的函數需要與服務通訊443，允許外部目的地連接埠上的所有流量 AWS 。或者，如果您不需要與其他 AWS 服務通訊，您也可以使用自我參考安全群組規則來限制對代理程式的存取。
- Amazon VPC 端點傳入規則：如果您使用的是 Amazon VPC 端點，與您的 Amazon VPC 端點相關聯的安全群組必須允許來自叢集安全群組的連接埠 443 上的傳入流量。

如果您的叢集使用身分驗證，則您也可以限制 Secrets Manager 端點的端點政策。若要呼叫 Secrets Manager API，Lambda 會使用您的函數角色，而不是 Lambda 服務主體。

Example VPC 端點政策 — Secrets Manager 端點

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
              "Principal": {
```

```
                "AWS": [
                    "arn:aws::iam::123456789012:role/my-role"
                ]
            },
            "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
        }
    ]
}
```

當您使用 Amazon VPC 端點時， 會使用端點的彈性網路界面 (ENI) AWS 路由您的 API 呼叫來叫用函數。Lambda 服務主體需要在使用這些 ENI 的任何角色和函數上呼叫 `lambda:InvokeFunction`。

根據預設，Amazon VPC 端點具有開放的 IAM 政策，允許廣泛存取資源。最佳實務是限制這些政策，以使用該端點執行所需的動作。為了確保事件來源映射能夠調用 Lambda 函數，VPC 端點政策必須允許 Lambda 服務主體呼叫 `sts:AssumeRole` 和 `lambda:InvokeFunction`。限制您的 VPC 端點政策以僅允許源自您組織內部的 API 呼叫，可阻止事件來源映射正常運作，因此在這些政策中需要 `"Resource": "*"`。

下列範例 VPC 端點政策展示了如何授予 Lambda 服務主體對 AWS STS 和 Lambda 端點的必要存取權。

Example VPC 端點政策 — AWS STS 端點

```
{
    "Statement": [
        {
            "Action": "sts:AssumeRole",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example VPC 端點政策 – Lambda 端點

```
{
```

```
    "Statement": [
        {
            "Action": "lambda:InvokeFunction",
            "Effect": "Allow",
            "Principal": {
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

建立 Amazon DocumentDB 事件來源映射 (主控台)

若要讓 Lambda 函數從 Amazon DocumentDB 叢集的變更串流中讀取，請建立 DocumentDB 事件來源映射。本節會說明如何透過 Lambda 主控台執行這項操作。如需 AWS SDK 和 AWS CLI 說明，請參閱 the section called “建立 Amazon DocumentDB 事件來源映射 (SDK 或 CLI)”。

建立 Amazon DocumentDB 事件來源映射 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 在 函式概觀 下，選擇 新增觸發條件 。
4. 在觸發條件組態底下的下拉式清單中，選擇 DocumentDB。
5. 設定需要的選項，然後選擇新增。

Lambda 支援以下 Amazon DocumentDB 事件來源的選項：

- DocumentDB 叢集：選取 Amazon DocumentDB 叢集。
- 啟用觸發條件：選擇您是否要立即啟用觸發條件。若勾選此核取方塊，您的函數會在建立事件來源映射時立即開始接收來自指定 Amazon DocumentDB 變更串流的流量。建議您取消勾選此核取方塊，以便在停用狀態下建立事件來源映射來進行測試。完成建立後，您隨時可以啟動事件來源映射。
- 資料庫名稱： 輸入叢集內要使用的資料庫名稱。
- (選用) 集合名稱：輸入資料庫內要使用的集合名稱。如果您未指定集合，Lambda 會偵聽資料庫中每個集合中的所有事件。
- 批次大小： 設定單一批次中要擷取的訊息數目上限 (最高 10,000)。預設批次大小為 100。

- 開始位置： 選擇串流中要從中開始讀取記錄的位置。
  - 最新： 僅處理已新增到串流的新記錄。Lambda 建立完事件來源後，您的函數才會開始處理記錄。這表示系統可能會捨棄部分記錄，直到您的事件來源成功建立為止。
  - 水平修剪 - 處理所有在串流中的記錄。Lambda 會透過叢集的日誌保留持續時間來決定開始讀取事件的時間點。具體來說，Lambda 會從 `current_time - log_retention_duration` 開始進行讀取。變更串流必須在此時間戳記前處於作用中狀態，Lambda 才能正確讀取所有事件。
  - At timestamp (在時間戳記為) - 從特定時間開始處理記錄。變更串流必須在指定時間戳記前處於作用中狀態，Lambda 才能正確讀取所有事件。
- 身分驗證： 選擇在叢集中存取中介裝置的身分驗證方式。
  - BASIC_AUTH： 使用基本身分驗證下，您必須提供含有憑證的 Secrets Manager 金鑰，才能存取叢集。
- Secrets Manager 金鑰：選擇含有存取 Amazon DocumentDB 叢集所需身分驗證詳細資料 (使用者名稱和密碼) 的 Secrets Manager 金鑰。
- (選用) 批次間隔：在調用函數前收集記錄的最長時間 (秒)，上限為 300 秒。
- (選用) 完整文件組態：文件更新作業方面，請選擇要傳送至串流的內容。預設值為 `Default`，這表示 Amazon DocumentDB 針對每個變更串流事件僅會傳送說明所做變更的差異。如需有關此欄位的詳細資訊，請參閱 MongoDB Javadoc API 文件中的 FullDocument。
  - 預設： Lambda 僅會傳送說明所做變更的部分文件。
  - UpdateLookup： Lambda 會傳送說明變更的差異，以及整個文件的副本。

建立 Amazon DocumentDB 事件來源映射 (SDK 或 CLI)

若要使用 AWS SDK 來建立或管理 Amazon DocumentDB 事件來源映射，您可以使用下列 API 操作：

- CreateEventSourceMapping
- ListEventSourceMappings
- GetEventSourceMapping
- UpdateEventSourceMapping
- DeleteEventSourceMapping

若要使用 建立事件來源映射 AWS CLI，請使用 `create-event-source-mapping`命令。以下範例使用此命令，將名為 `my-function` 的函數映射至 Amazon DocumentDB 變更串流。事件來源是由 Amazon Resource Name (ARN) 指定，批次大小為 500，且開始時間為 Unix 時間的時間戳記。

此命令也會指定 Lambda 用來連線至 Amazon DocumentDB 的 Secrets Manager 金鑰。此外也包括 document-db-event-source-config 參數，這個參數指定了要從哪個資料庫和集合讀取。

```
aws lambda create-event-source-mapping --function-name my-function \
    --event-source-arn arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy
    --batch-size 500 \
    --starting-position AT_TIMESTAMP \
    --starting-position-timestamp 1541139109 \
    --source-access-configurations
 '[{"Type":"BASIC_AUTH","URI":"arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocDBSecret-BAtjxi"}]' \
    --document-db-event-source-config '{"DatabaseName":"test_database",
 "CollectionName": "test_collection"}' \
```

您應該會看到類似下面的輸出：

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541348195.412,
    "LastProcessingResult": "No records processed",
    "State": "Creating",
    "StateTransitionReason": "User action"
}
```

完成建立後，您可以使用 update-event-source-mapping 命令來更新 Amazon DocumentDB 事件來源的設定。以下命令將批次大小更新為 1,000，並將批次間格更新為 10 秒。對於此命令，您必須擁有事件來源映射的 UUID (可使用 list-event-source-mapping 命令或 Lambda 主控台擷取)。

```
aws lambda update-event-source-mapping --function-name my-function \
    --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
    --batch-size 1000 \
```

```
    --batch-window 10
```

您應該會看到類似以下內容的輸出：

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "BatchSize": 500,
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "MaximumBatchingWindowInSeconds": 0,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
    "LastProcessingResult": "OK",
    "State": "Updating",
    "StateTransitionReason": "User action"
}
```

Lambda 會以非同步方式更新設定，因此處理完成之前您可能無法在輸出中看到這些變更。若要檢視事件來源映射的目前設定，請使用 get-event-source-mapping 命令。

```
aws lambda get-event-source-mapping --uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b
```

您應該會看到類似以下內容的輸出：

```
{
    "UUID": "2b733gdc-8ac3-cdf5-af3a-1827b3b11284",
    "DocumentDBEventSourceConfig": {
        "CollectionName": "test_collection",
        "DatabaseName": "test_database",
        "FullDocument": "Default"
    },
    "BatchSize": 1000,
    "MaximumBatchingWindowInSeconds": 10,
    "EventSourceArn": "arn:aws:rds:us-west-2:123456789012:cluster:privatecluster7de2-
epzcyvu4pjoy",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "LastModified": 1541359182.919,
```

```
    "LastProcessingResult": "OK",
    "State": "Enabled",
    "StateTransitionReason": "User action"
}
```

若要刪除 Amazon DocumentDB 事件來源映射，請使用 delete-event-source-mapping 命令。

```
aws lambda delete-event-source-mapping \
    --uuid 2b733gdc-8ac3-cdf5-af3a-1827b3b11284
```

輪詢和串流開始位置

請注意，建立和更新事件來源映射期間的串流輪詢最終會一致。

- 在建立事件來源映射期間，從串流開始輪詢事件可能需要幾分鐘時間。
- 在更新事件來源映射期間，從串流停止並重新開始輪詢事件可能需要幾分鐘時間。

這種行為表示如果您指定 LATEST 當作串流的開始位置，事件來源映射可能會在建立或更新期間遺漏事件。若要確保沒有遺漏任何事件，請將串流開始位置指定為 TRIM_HORIZON 或 AT_TIMESTAMP。

監控 Amazon DocumentDB 事件來源

為協助您監控 Amazon DocumentDB 事件來源，Lambda 會在您的函數處理完一批記錄後發出 IteratorAge 指標。迭代器存留期是最近事件和目前時間戳記之間的差距。基本上，IteratorAge 指標會指出批次中最後處理的記錄經過了多久的時間。如果函數目前正在處理新的事件，可使用迭代器存留期來預估記錄新增與函數實際處理之間的延遲。從 IteratorAge 的增加趨勢可看出您函數的問題。如需詳細資訊，請參閱將 CloudWatch 指標與 Lambda 搭配使用。

Amazon DocumentDB 變更串流未最佳化，無法處理事件之間的大量時間間隔。如果 Amazon DocumentDB 事件來源長時間未收到任何事件，Lambda 可能會停用事件來源映射。此期間長度可能從幾週到幾個月不等，取決於叢集大小和其他工作負載。

Lambda 支援的承載上限為 6MB。不過，Amazon DocumentDB 變更串流事件的大小可達 16MB。如果變更串流嘗試向 Lambda 傳送大於 6MB 的變更串流事件，Lambda 會捨棄訊息並發出 OversizedRecordCount 指標。Lambda 會盡力發送所有指標。

教學課程： AWS Lambda 搭配 Amazon DocumentDB Streams 使用

在本教學課程中，您將建立一個基礎 Lambda 函數，它會從 Amazon DocumentDB (with MongoDB compatibility) 變更串流中取用事件。完成本教學課程需逐一進行以下階段：

- 設定您的 Amazon DocumentDB 叢集、連線到叢集，然後在叢集上啟用變更串流。
- 建立 Lambda 函數，並將 Amazon DocumentDB 叢集設定為函數的事件來源。
- 將項目插入 Amazon DocumentDB 資料庫中，即可測試設定。

建立 Amazon DocumentDB 叢集

1. 開啟 Amazon DocumentDB 主控台。在叢集下，選擇建立。
2. 使用下列組態建立叢集：
   - 在叢集類型欄位中，選擇執行個體型叢集。此為預設選項。
   - 在叢集組態欄位中，確保已選取引擎版本 5.0.0。此為預設選項。
   - 在執行個體組態欄位中：
     - 在資料庫執行個體類別欄位中，選取記憶體最佳化類別。此為預設選項。
     - 在常規複本執行個體數量欄位中，選擇 1。
     - 在執行個體類別欄位中，使用預設選項。
   - 在身分驗證欄位中，輸入主要使用者的使用者名稱，然後選擇自我管理。輸入密碼，然後進行確認。
   - 請保留所有其他預設設定。
3. 選擇 Create Cluster (建立叢集)。

在 Secrets Manager 中建立密碼

當 Amazon DocumentDB 正在建立叢集時，請建立 AWS Secrets Manager 秘密來存放資料庫登入資料。在後續步驟中建立 Lambda 事件來源映射時，您需要提供此秘密。

在 Secrets Manager 中建立密碼

1. 開啟 Secrets Manager 主控台，並選擇儲存新密碼。
2. 針對選擇密碼類型，選擇以下選項：
   - 在基本詳細資訊下：
     - 密碼類型：Amazon DocumentDB 資料庫的憑證
     - 在憑證欄位中，輸入用於建立 Amazon DocumentDB 叢集的相同使用者名稱與密碼。
     - 資料庫：選擇您的 Amazon DocumentDB 叢集。
     - 選擇下一步。

3. 針對設定密碼，選擇下列選項：
   - 秘密名稱：DocumentDBSecret
   - 選擇下一步。
4. 選擇下一步。
5. 選擇儲存。
6. 重新整理主控台以確認您已成功儲存 DocumentDBSecret 密碼。

請記錄秘密 ARN。在後續步驟中需要它。

連線至叢集

使用 連線至您的 Amazon DocumentDB 叢集 AWS CloudShell

1. 在 Amazon DocumentDB 管理主控台的叢集下，找到已建立的叢集。按一下叢集旁的核取方塊，選擇對應叢集。
2. 選擇連線至叢集。CloudShell 執行命令畫面隨即顯示。
3. 在新增環境名稱欄位中，輸入唯一名稱 (例如 "test")，然後選擇建立並執行。
4. 出現提示時，輸入您的密碼。當提示變為 `rs0 [direct: primary] <env-name>>`，表示已成功連線至 Amazon DocumentDB 叢集。

啟用變更串流

在本教學課程中，您將追蹤 Amazon DocumentDB 叢集中 docdbdemo 資料庫 products 集合的變更。可以透過啟用變更串流來完成此操作。

在叢集內建立新資料庫

1. 執行下列命令，建立名為 docdbdemo 的新資料庫：

```
use docdbdemo
```

2. 在終端機視窗中，使用下列命令將記錄插入 docdbdemo：

```
db.products.insertOne({"hello":"world"})
```

   您應該會看到如下所示的輸出：

```
{
  acknowledged: true,
  insertedId: ObjectId('67f85066ca526410fd531d59')
}
```

3. 接下來，使用以下命令在 docdbdemo 資料庫的 products 集合上啟用變更串流：

```
db.adminCommand({modifyChangeStreams: 1,
    database: "docdbdemo",
    collection: "products",
    enable: true});
```

您應該會看到類似下面的輸出：

```
{ "ok" : 1, "operationTime" : Timestamp(1680126165, 1) }
```

建立介面 VPC 端點

接下來，建立介面 VPC 端點，以確保 Lambda 和 Secrets Manager (稍後用來儲存我們的叢集存取憑證) 可以連線到您的預設 VPC。

建立介面 VPC 端點

1. 開啟 VPC 主控台。在左側選單的虛擬私有雲端下，選擇端點。
2. 選擇建立端點。使用下列組態建立端點：
   - 針對名稱標籤，輸入 lambda-default-vpc。
   - 針對服務類別，選擇 AWS 服務。
   - 針對服務，在搜尋方塊中輸入 lambda。選擇格式為 com.amazonaws.<region>.lambda 的服務。
   - 在 VPC 欄位中，選擇 Amazon DocumentDB 叢集所在的 VPC。這通常是預設 VPC。
   - 針對子網路，請核取每個可用區域旁邊的方塊。請選擇每個可用區域的正確子網路。
   - 針對 IP 地址類型，請選擇 IPv4。
   - 在安全群組欄位中，選擇 Amazon DocumentDB 叢集所使用的安全群組。這通常是 default 安全群組。
   - 請保留所有其他預設設定。

- 選擇建立端點。

3. 再次選擇建立端點。使用下列組態建立端點：

   - 針對名稱標籤，輸入 `secretsmanager-default-vpc`。
   - 針對服務類別，選擇 AWS 服務。
   - 針對服務，在搜尋方塊中輸入 `secretsmanager`。選擇格式為 `com.amazonaws.<region>.secretsmanager` 的服務。
   - 在 VPC 欄位中，選擇 Amazon DocumentDB 叢集所在的 VPC。這通常是預設 VPC。
   - 針對子網路，請核取每個可用區域旁邊的方塊。請選擇每個可用區域的正確子網路。
   - 針對 IP 地址類型，請選擇 IPv4。
   - 在安全群組欄位中，選擇 Amazon DocumentDB 叢集所使用的安全群組。這通常是 `default` 安全群組。
   - 請保留所有其他預設設定。
   - 選擇建立端點。

這就完成了本教學課程的叢集設定部分。

建立執行角色

在接下來的一組步驟中，您將建立 Lambda 函數。首先，您需要建立授予函數存取叢集許可的執行角色。您可先建立 IAM 政策，然後將此政策連接到 IAM 角色。

建立 IAM 政策

1. 開啟 IAM 主控台的政策頁面，並選擇建立政策。
2. 選擇 JSON 標籤。在下列政策中，將陳述式最後一行中的 Secrets Manager 資源 ARN 取代為先前的密碼 ARN，並將政策複製到編輯器中。

   JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "LambdaESMNetworkingAccess",
            "Effect": "Allow",
```

```
            "Action": [
                "ec2:CreateNetworkInterface",
                "ec2:DescribeNetworkInterfaces",
                "ec2:DescribeVpcs",
                "ec2:DeleteNetworkInterface",
                "ec2:DescribeSubnets",
                "ec2:DescribeSecurityGroups",
                "kms:Decrypt"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMAccess",
            "Effect": "Allow",
            "Action": [
                "rds:DescribeDBClusters",
                "rds:DescribeDBClusterParameters",
                "rds:DescribeDBSubnetGroups"
            ],
            "Resource": "*"
        },
        {
            "Sid": "LambdaDocDBESMGetSecretValueAccess",
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": "arn:aws:secretsmanager:us-
east-1:123456789012:secret:DocumentDBSecret"
        }
    ]
}
```

3. 選擇下一步：標籤，然後選擇下一步：檢閱。
4. 對於 Name (名稱)，輸入 AWSDocumentDBLambdaPolicy。
5. 選擇建立政策。

建立 IAM 角色

1. 開啟 IAM 主控台的角色頁面，然後選擇建立角色。
2. 針對選取信任的實體，請選擇以下選項：

- 信任的實體類型： AWS 服務
- 服務或使用案例：Lambda
- 選擇下一步。

3. 針對新增許可，請選擇您剛建立的 AWSDocumentDBLambdaPolicy 政策以及 AWSLambdaBasicExecutionRole，授予函數寫入 Amazon CloudWatch Logs 的許可。
4. 選擇下一步。
5. 在角色名稱中，輸入 AWSDocumentDBLambdaExecutionRole。
6. 選擇建立角色。

建立 Lambda 函式

本教學課程使用 Python 3.14 執行時間，但我們也提供其他執行時間的範例程式碼檔案。您可以在下列方塊中選取索引標籤，查看您感興趣的執行期程式碼。

程式碼會接收 Amazon DocumentDB 事件輸入，並處理其包含的訊息。

建立 Lambda 函數

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇建立函數。
3. 選擇從頭開始撰寫
4. 在基本資訊下，請執行下列動作：

   a. 在函數名稱輸入 ProcessDocumentDBRecords
   b. 針對執行期，選擇 Python 3.14。
   c. 對於 Architecture (架構)，選擇 x86_64。
5. 在變更預設執行角色索引標籤中，執行下列操作：

   a. 展開索引標籤，然後選擇使用現有角色。
   b. 選擇您之前建立的 AWSDocumentDBLambdaExecutionRole。
6. 選擇建立函數。

部署函數程式碼

1. 在以下方塊中選擇 Python 索引標籤，然後複製程式碼。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
using Amazon.Lambda.Core;
using System.Text.Json;
using System;
using System.Collections.Generic;
using System.Text.Json.Serialization;
//Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson

namespace LambdaDocDb;

public class Function
{

     /// <summary>
    /// Lambda function entry point to process Amazon DocumentDB events.
    /// </summary>
    /// <param name="event">The Amazon DocumentDB event.</param>
    /// <param name="context">The Lambda context object.</param>
    /// <returns>A string to indicate successful processing.</returns>
    public string FunctionHandler(Event evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Events)
        {
            ProcessDocumentDBEvent(record, context);
        }
```

```
        return "OK";
    }

     private void ProcessDocumentDBEvent(DocumentDBEventRecord record,
ILambdaContext context)
    {

        var eventData = record.Event;
        var operationType = eventData.OperationType;
        var databaseName = eventData.Ns.Db;
        var collectionName = eventData.Ns.Coll;
        var fullDocument = JsonSerializer.Serialize(eventData.FullDocument,
new JsonSerializerOptions { WriteIndented = true });

        context.Logger.LogLine($"Operation type: {operationType}");
        context.Logger.LogLine($"Database: {databaseName}");
        context.Logger.LogLine($"Collection: {collectionName}");
        context.Logger.LogLine($"Full document:\n{fullDocument}");
    }



    public class Event
    {
        [JsonPropertyName("eventSourceArn")]
        public string EventSourceArn { get; set; }

        [JsonPropertyName("events")]
        public List<DocumentDBEventRecord> Events { get; set; }

        [JsonPropertyName("eventSource")]
        public string EventSource { get; set; }
    }

    public class DocumentDBEventRecord
    {
        [JsonPropertyName("event")]
        public EventData Event { get; set; }
    }

    public class EventData
    {
        [JsonPropertyName("_id")]
        public IdData Id { get; set; }
```

```
        [JsonPropertyName("clusterTime")]
        public ClusterTime ClusterTime { get; set; }

        [JsonPropertyName("documentKey")]
        public DocumentKey DocumentKey { get; set; }

        [JsonPropertyName("fullDocument")]
        public Dictionary<string, object> FullDocument { get; set; }

        [JsonPropertyName("ns")]
        public Namespace Ns { get; set; }

        [JsonPropertyName("operationType")]
        public string OperationType { get; set; }
    }

    public class IdData
    {
        [JsonPropertyName("_data")]
        public string Data { get; set; }
    }

    public class ClusterTime
    {
        [JsonPropertyName("$timestamp")]
        public Timestamp Timestamp { get; set; }
    }

    public class Timestamp
    {
        [JsonPropertyName("t")]
        public long T { get; set; }

        [JsonPropertyName("i")]
        public int I { get; set; }
    }

    public class DocumentKey
    {
        [JsonPropertyName("_id")]
        public Id Id { get; set; }
    }
```

```
    public class Id
    {
        [JsonPropertyName("$oid")]
        public string Oid { get; set; }
    }

    public class Namespace
    {
        [JsonPropertyName("db")]
        public string Db { get; set; }

        [JsonPropertyName("coll")]
        public string Coll { get; set; }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
package main

import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
)

type Event struct {
```

```
 Events []Record `json:"events"`
}

type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
  FullDocument interface{} `json:"fullDocument"`
 } `json:"event"`
}

func main() {
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
import java.util.List;
import java.util.Map;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class Example implements RequestHandler<Map<String, Object>, String> {

    @SuppressWarnings("unchecked")
    @Override
    public String handleRequest(Map<String, Object> event, Context context) {
        List<Map<String, Object>> events = (List<Map<String, Object>>)
 event.get("events");
        for (Map<String, Object> record : events) {
            Map<String, Object> eventData = (Map<String, Object>)
 record.get("event");
            processEventData(eventData);
        }

        return "OK";
    }

    @SuppressWarnings("unchecked")
    private void processEventData(Map<String, Object> eventData) {
        String operationType = (String) eventData.get("operationType");
        System.out.println("operationType: %s".formatted(operationType));

        Map<String, Object> ns = (Map<String, Object>) eventData.get("ns");
```

```
        String db = (String) ns.get("db");
        System.out.println("db: %s".formatted(db));
        String coll = (String) ns.get("coll");
        System.out.println("coll: %s".formatted(coll));

        Map<String, Object> fullDocument = (Map<String, Object>)
 eventData.get("fullDocument");
        System.out.println("fullDocument: %s".formatted(fullDocument));
    }

}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
    });
    return 'OK';
};

const logDocumentDBEvent = (record) => {
    console.log('Operation type: ' + record.event.operationType);
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument,
 null, 2));
};
```

使用 TypeScript 搭配 Lambda 使用 Amazon DocumentDB 事件

```
import { DocumentDBEventRecord, DocumentDBEventSubscriptionContext } from
 'aws-lambda';

console.log('Loading function');

export const handler = async (
  event: DocumentDBEventSubscriptionContext,
  context: any
): Promise<string> => {
  event.events.forEach((record: DocumentDBEventRecord) => {
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record: DocumentDBEventRecord): void => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
  console.log('collection: ' + record.event.ns.coll);
  console.log('Full document:', JSON.stringify(record.event.fullDocument,
 null, 2));
};
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
<?php

require __DIR__.'/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Handler;

class DocumentDBEventHandler implements Handler
{
    public function handle($event, Context $context): string
    {

        $events = $event['events'] ?? [];
        foreach ($events as $record) {
            $this->logDocumentDBEvent($record['event']);
        }
        return 'OK';
    }

    private function logDocumentDBEvent($event): void
    {
        // Extract information from the event record

        $operationType = $event['operationType'] ?? 'Unknown';
        $db = $event['ns']['db'] ?? 'Unknown';
        $collection = $event['ns']['coll'] ?? 'Unknown';
        $fullDocument = $event['fullDocument'] ?? [];

        // Log the event details

        echo "Operation type: $operationType\n";
        echo "Database: $db\n";
        echo "Collection: $collection\n";
        echo "Full document: " . json_encode($fullDocument,
 JSON_PRETTY_PRINT) . "\n";
    }
}
return new DocumentDBEventHandler();
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
        log_document_db_event(record)
    return 'OK'

def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
    print(f"collection: {collection}")
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}

  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

Rust

適用於 Rust 的 SDK

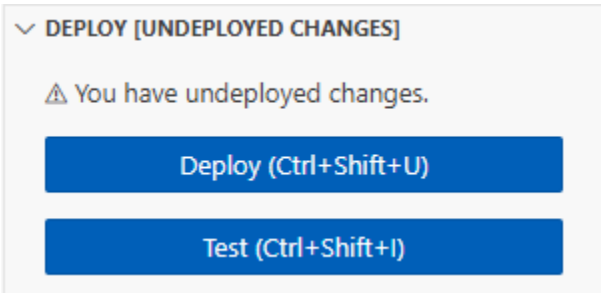
> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::documentdb::{DocumentDbEvent, DocumentDbInnerEvent},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features
 = ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<DocumentDbEvent>) ->Result<(),
 Error> {

    tracing::info!("Event Source ARN: {:?}", event.payload.event_source_arn);
    tracing::info!("Event Source: {:?}", event.payload.event_source);

    let records = &event.payload.events;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }
```

```
    for record in records{
        log_document_db_event(record);
    }

    tracing::info!("Document db records processed");

    // Prepare the response
    Ok(())

}

fn log_document_db_event(record: &DocumentDbInnerEvent)-> Result<(), Error>{
    tracing::info!("Change Event: {:?}", record.event);

    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

2. 在 Lambda 主控台的程式碼來源窗格中，將程式碼貼到程式碼編輯器中，取代 Lambda 建立的程式碼。
3. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：



建立 Lambda 事件來源映射

建立可將 Amazon DocumentDB 變更串流與 Lambda 函數建立關聯的事件來源映射。建立此事件來源映射之後， 會 AWS Lambda 立即開始輪詢串流。

建立事件來源映射

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您之前建立的 `ProcessDocumentDBRecords` 函數。
3. 選擇組態索引標籤，然後選擇左側功能表中的觸發程序。
4. 選擇 Add trigger (新增觸發條件)。
5. 在觸發條件組態下，針對來源選取 Amazon DocumentDB。
6. 使用下列組態建立事件來源映射：
   - Amazon DocumentDB 叢集：選擇您先前建立的叢集。
   - 資料庫名稱：docdbdemo
   - 集合名稱：產品
   - 批次大小：1
   - 起始位置：最新
   - 身分驗證：BASIC_AUTH
   - Secrets Manager 金鑰：選擇 Amazon DocumentDB 叢集對應的秘密。其名稱應該類似 `rds!cluster-12345678-a6f0-52c0-b290-db4aga89274f`。
   - 批次時段：1
   - 完整文件組態：UpdateLookup
7. 選擇新增。建立事件來源映射可能需要幾分鐘的時間。

測試 函數

等待事件來源映射進入已啟用狀態。這可能需要幾分鐘的時間。接著，透過插入、更新和刪除資料庫記錄來測試端對端設定。開始之前：

1. 在 CloudShell 環境中重新連線至 Amazon DocumentDB 叢集。
2. 執行下列命令，確保使用的是 docdbdemo 資料庫：

```
use docdbdemo
```

插入記錄

將記錄插入到 docdbdemo 資料庫的 products 集合中：

```
db.products.insertOne({"name":"Pencil", "price": 1.00})
```

透過檢查 CloudWatch Logs，確認函式已成功處理此事件。您應該會看到如下所示的日誌項目：

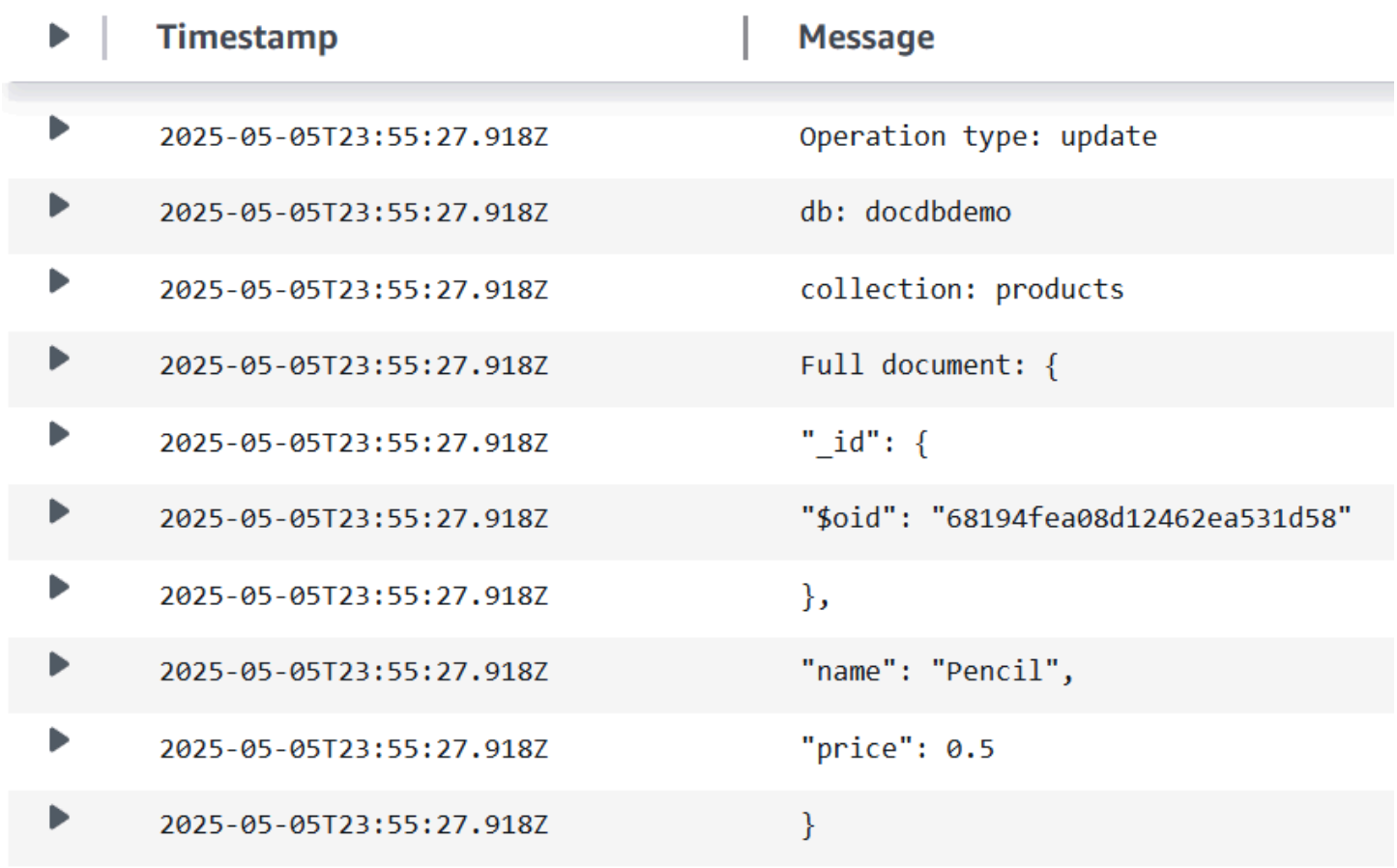
Timestamp	Message
2025-05-05T23:55:23.474Z	Operation type: insert
2025-05-05T23:55:23.474Z	db: docdbdemo
2025-05-05T23:55:23.474Z	collection: products
2025-05-05T23:55:23.474Z	Full document: {
2025-05-05T23:55:23.474Z	"_id": {
2025-05-05T23:55:23.474Z	"$oid": "68194fea08d12462ea531d58"
2025-05-05T23:55:23.474Z	},
2025-05-05T23:55:23.474Z	"name": "Pencil",
2025-05-05T23:55:23.474Z	"price": 1
2025-05-05T23:55:23.474Z	}

更新記錄

使用下列命令更新剛剛插入的記錄：

```
db.products.updateOne(
    { "name": "Pencil" },
    { $set: { "price": 0.50 }}
)
```

透過檢查 CloudWatch Logs，確認函式已成功處理此事件。您應該會看到如下所示的日誌項目：



Timestamp	Message
2025-05-05T23:55:27.918Z	Operation type: update
2025-05-05T23:55:27.918Z	db: docdbdemo
2025-05-05T23:55:27.918Z	collection: products
2025-05-05T23:55:27.918Z	Full document: {
2025-05-05T23:55:27.918Z	"_id": {
2025-05-05T23:55:27.918Z	"$oid": "68194fea08d12462ea531d58"
2025-05-05T23:55:27.918Z	},
2025-05-05T23:55:27.918Z	"name": "Pencil",
2025-05-05T23:55:27.918Z	"price": 0.5
2025-05-05T23:55:27.918Z	}

刪除記錄

使用下列命令刪除剛剛更新的記錄：

```
db.products.deleteOne( { "name": "Pencil" } )
```

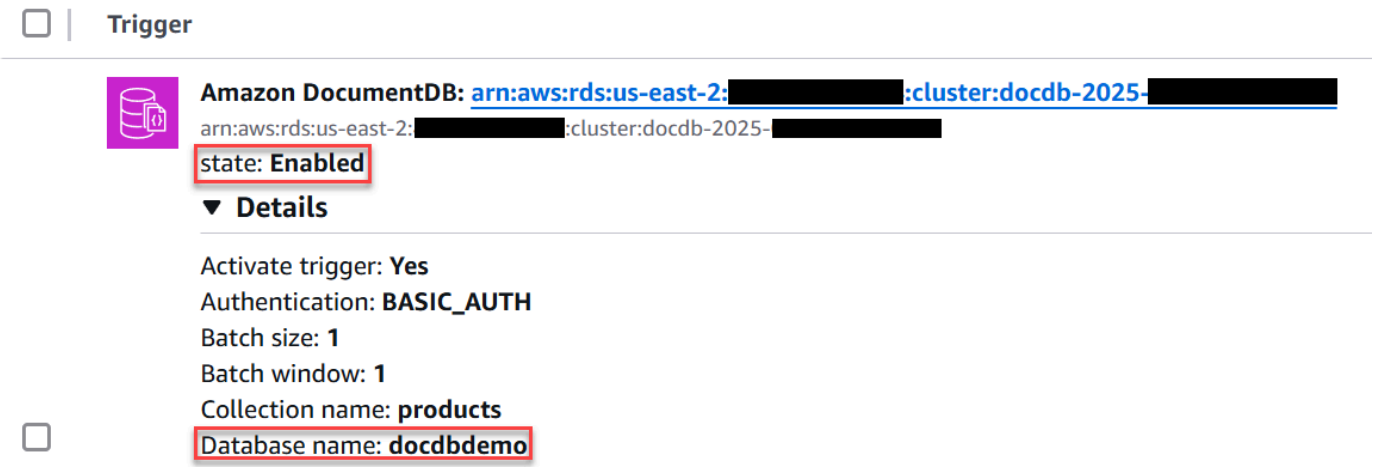
透過檢查 CloudWatch Logs，確認函式已成功處理此事件。您應該會看到如下所示的日誌項目：

Timestamp	Message
2025-05-05T23:55:32.362Z	Operation type: delete
2025-05-05T23:55:32.362Z	db: docdbdemo
2025-05-05T23:55:32.362Z	collection: products
2025-05-05T23:55:32.362Z	Full document: {}

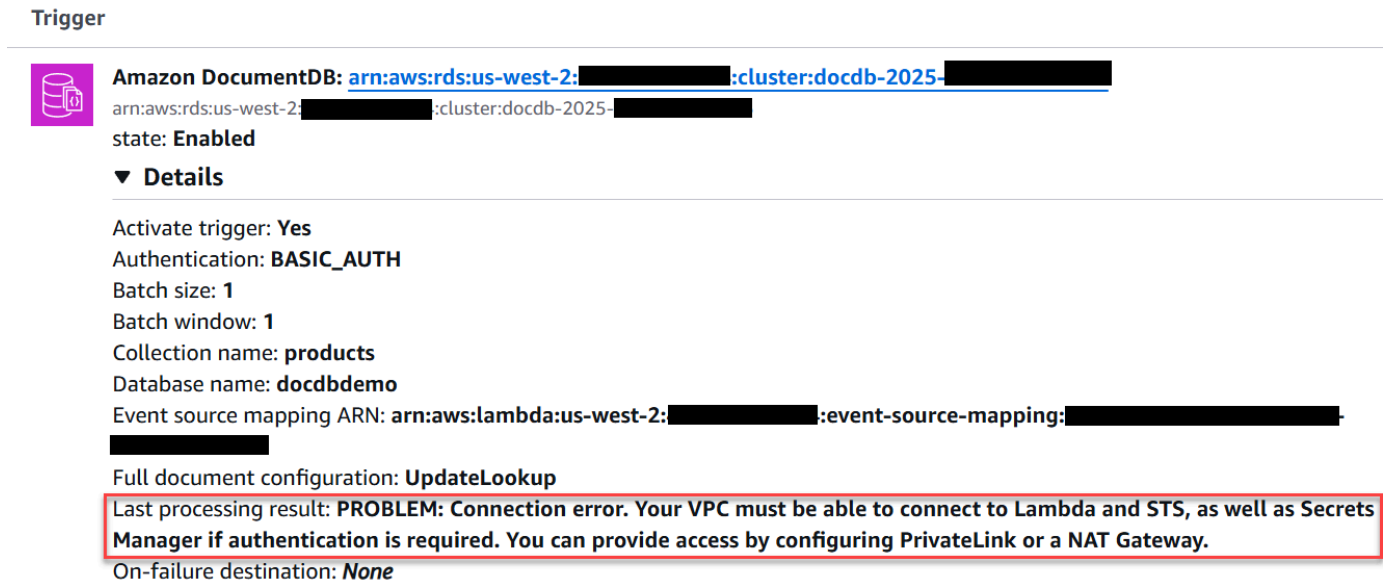
疑難排解

若未在函式的 CloudWatch 日誌中看到任何資料庫事件，請檢查下列項目：

- 確認 Lambda 事件來源映射 (也稱為觸發程序) 處於已啟用狀態。事件來源映射的建立可能需要數分鐘時間。
- 若事件來源映射已啟用，但您仍然無法在 CloudWatch 中看到資料庫事件：
  - 確認事件來源映射中的資料庫名稱設定為 docdbdemo。



  - 檢查事件來源映射的上次處理結果欄位，看是否出現 "PROBLEM: Connection error. Your VPC must be able to connect to Lambda and STS, as well as Secrets Manager if authentication is required." 訊息。若看到此錯誤，確認已建立 Lambda 與 Secrets Manager VPC 介面端點，且這些端點使用的 VPC 及子網路與 Amazon DocumentDB 叢集所使用的相同。



清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。透過刪除您不再使用的 AWS 資源，可為 AWS 帳戶避免不必要的費用。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **`confirm`**，然後選擇刪除。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇刪除。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇刪除。

刪除 VPC 端點。

1. 開啟 VPC 主控台。在左側選單的虛擬私有雲端下，選擇端點。

2. 選擇您建立的端點。
3. 選擇 Actions (動作)、Delete VPC endpoints (刪除 VPC 端點)。
4. 在文字輸入欄位中輸入 **delete**。
5. 選擇 刪除。

刪除 Amazon DocumentDB 叢集

1. 開啟 Amazon DocumentDB 主控台。
2. 選擇您為本教學課程建立的 Amazon DocumentDB 叢集，並停用刪除保護。
3. 在主叢集頁面中，再次選擇您的 Amazon DocumentDB 叢集。
4. 選擇 動作、刪除。
5. 針對建立最終叢集快照，請選取否。
6. 在文字輸入欄位中輸入 **delete**。
7. 選擇 刪除。

在 Secrets Manager 中刪除密碼

1. 開啟 Secrets Manager 主控台。
2. 選擇您為此教學課程建立的密碼。
3. 選擇動作、刪除機密。
4. 選擇 Schedule deletion (排定刪除)。

AWS Lambda 搭配 Amazon DynamoDB 使用

> **Note**
>
> 如要將資料傳送到 Lambda 函數以外的目標，或在傳送資料之前讓資料更豐富，請參閱 Amazon EventBridge Pipes。

您可以使用 AWS Lambda 函數來處理 Amazon DynamoDB 串流中的記錄。您可以透過 DynamoDB Streams，以在每次更新 DynamoDB 資料表時，觸發 Lambda 函數來執行額外的工作。

處理 DynamoDB 串流時，您需要實作部分批次回應邏輯，避免批次中部分記錄處理失敗時，已成功處理的記錄被重新執行。適用於 的 Powertools 的 Batch Processor 公用程式 AWS Lambda 可在 Python、TypeScript、.NET 和 Java 中使用，並透過自動處理部分批次回應邏輯、縮短開發時間並改善可靠性來簡化此實作。

主題

- 輪詢和批次處理串流
- 輪詢和串流開始位置
- DynamoDB Streams 中的碎片同時讀取
- 範例事件
- 使用 Lambda 處理 DynamoDB 記錄
- 使用 DynamoDB 和 Lambda 設定部分批次回應
- 在 Lambda 中保留 DynamoDB 事件來源的捨棄記錄
- 在 Lambda 中實作有狀態的 DynamoDB 串流處理
- Amazon DynamoDB 事件來源映射的 Lambda 參數
- 搭配 DynamoDB 事件來源使用事件篩選
- 教學課程： AWS Lambda 搭配 Amazon DynamoDB 串流使用

輪詢和批次處理串流

Lambda 會輪詢您 DynamoDB 串流中的碎片，其記錄的基本速率為每秒 4 次。當記錄可用時，Lambda 會調用您的函數，並等待結果。如果處理成功，Lambda 會恢復輪詢，直到收到多筆記錄。

Lambda 預設會在記錄可用時立即調用函數。如果 Lambda 從事件來源中讀取的批次只有一筆記錄，Lambda 只會傳送一筆記錄至函數。為避免調用具有少量記錄的函數，您可設定批次間隔，請求事件來源緩衝記錄最長達五分鐘。調用函數之前，Lambda 會繼續從事件來源中讀取記錄，直到收集到完整批次、批次間隔到期或者批次達到 6 MB 的承載限制。如需詳細資訊，請參閱批次處理行為。

> **Warning**
>
> Lambda 事件來源映射至少會處理每個事件一次，而且可能會重複處理記錄。為避免與重複事件相關的潛在問題，強烈建議您讓函數程式碼具有等冪性。若要進一步了解，請參閱 AWS 知識中心中的如何使 Lambda 函數具有等冪性。

Lambda 在傳送下一批進行處理前不會等待任何已設定的擴充完成。換句話說，當 Lambda 處理下一批記錄時，您的擴充功能可能會繼續執行。如果您違反任何帳戶的 並行 設定或限制，便可能會產生限流的問題。若要偵測此是否為潛在問題，請監控您的函數，並確認您看到的 並行指標 是否高於事件來源映射的預期值。由於兩次調用之間的時間很短，Lambda 可能會短暫報告比碎片數目更高的並行用量。即使對於沒有延伸項目的 Lambda 函數也可能如此。

設定 ParallelizationFactor 設定來同時處理 DynamoDB 資料串流的一個碎片與多個 Lambda 調用。您可以透過從 1 (預設) 到 10 的並行化因子指定 Lambda 從碎片輪詢的並行批次數。例如，當 `ParallelizationFactor` 設定為 2 時，您最多可以有 200 個並行 Lambda 調用，來處理 100 個 DynamoDB 串流碎片 (不過在實務中，`ConcurrentExecutions` 指標可能有不同值)。當資料量急劇波動並且 IteratorAge 較高時，這有助於向上擴展處理輸送量。如果增加每個碎片的並行批次數量，Lambda 仍會確保在項目 (分割區和排序索引鍵) 層級進行依序處理。

輪詢和串流開始位置

請注意，建立和更新事件來源映射期間的串流輪詢最終會一致。

- 在建立事件來源映射期間，從串流開始輪詢事件可能需要幾分鐘時間。
- 在更新事件來源映射期間，從串流停止並重新開始輪詢事件可能需要幾分鐘時間。

這種行為表示如果您指定 `LATEST` 當作串流的開始位置，事件來源映射可能會在建立或更新期間遺漏事件。若要確保沒有遺漏任何事件，請將串流開始位置指定為 `TRIM_HORIZON`。

DynamoDB Streams 中的碎片同時讀取

對於不是全域資料表的單一區域資料表，您最多可以設計兩個 Lambda 函數，以便同時讀取同一個 DynamoDB Streams 碎片。超過此限制會導致請求限流。對於全域資料表，建議您將同時函數的數量限制為一個，以避免請求限流。

範例事件

Example

```
{
  "Records": [
    {
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "NewImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES",
        "SequenceNumber": "111",
        "SizeBytes": 26
      },
      "awsRegion": "us-west-2",
      "eventName": "INSERT",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
    },
    {
      "eventID": "2",
      "eventVersion": "1.0",
      "dynamodb": {
```

```
        "OldImage": {
          "Message": {
            "S": "New item!"
          },
          "Id": {
            "N": "101"
          }
        },
        "SequenceNumber": "222",
        "Keys": {
          "Id": {
            "N": "101"
          }
        },
        "SizeBytes": 59,
        "NewImage": {
          "Message": {
            "S": "This item has changed"
          },
          "Id": {
            "N": "101"
          }
        },
        "StreamViewType": "NEW_AND_OLD_IMAGES"
      },
      "awsRegion": "us-west-2",
      "eventName": "MODIFY",
      "eventSourceARN": "arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525",
      "eventSource": "aws:dynamodb"
    }
  ]}
```

使用 Lambda 處理 DynamoDB 記錄

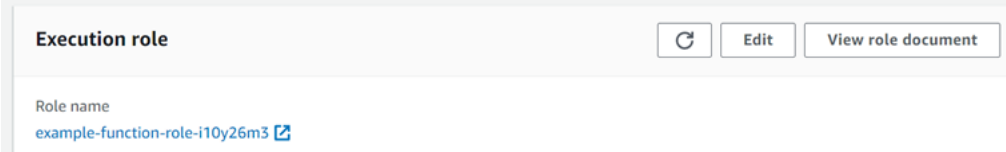
建立事件來源映射，指示 Lambda 從您的串流傳送記錄至 Lambda 函數。您可以建立多個事件來源映射，來使用多個 Lambda 函數處理相同資料，或使用單一函數處理來自多個串流的項目。

您可以設定事件來源映射，以處理來自不同 AWS 帳戶中的串流的記錄。如需詳細資訊，請參閱 the section called “跨帳戶映射”。

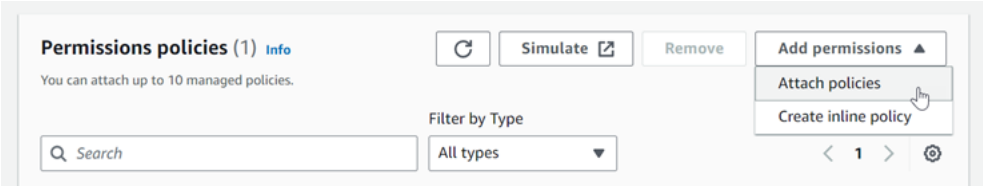
若要將函數設定為從 DynamoDB Streams 讀取，請將 AWSLambdaDynamoDBExecutionRole AWS 受管政策連接至您的執行角色，然後建立 DynamoDB 觸發。

若要新增許可並建立觸發條件

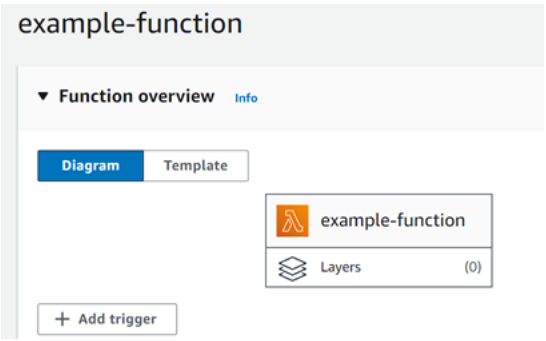
1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 依序選擇 Configuration (組態) 索引標籤和 Permissions (許可)。
4. 在角色名稱下面，選擇執行角色連結。此連結會在 IAM 主控台中開啟該角色。



5. 選擇新增許可，然後選擇連接政策。



6. 在搜尋欄位中輸入 AWSLambdaDynamoDBExecutionRole。將此政策新增至您的執行角色。這是 AWS 受管政策，其中包含函數從 DynamoDB 串流讀取所需的許可。如需此政策的詳細資訊，請參閱《AWS 受管政策參考》中的 AWSLambdaDynamoDBExecutionRole。
7. 在 Lambda 主控台中返回您的 Lambda 函數。在函數概觀下，選擇新增觸發條件。



8. 選擇觸發條件類型。
9. 設定需要的選項，然後選擇新增。

Lambda 支援 DynamoDB 事件來源的下列選項：

事件來源選項

- DynamoDB 資料表 - 從中讀取記錄的 DynamoDB 資料表。

- 批次大小 – 每個批次中要傳送至函數的記錄數量，最高為 10,000。Lambda 會將批次中所有記錄以單一呼叫傳送至函數，前提是事件的總大小不超過同步調用的酬載限制 (6 MB)。
- 批次間隔 - 指定調用函數前收集記錄的最長時間 (秒)。
- 開始位置 - 只處理新記錄，或所有現有的記錄。
  - 最新 - 處理已新增到串流的記錄。
  - 水平修剪 - 處理所有在串流中的記錄。

  處理任何現有的記錄後，該函式已跟上進度並持續處理新的記錄。
- 故障目的地 - 用於無法處理之記錄的標準 SQS 佇列或標準 SNS 主題。當 Lambda 捨棄太舊或已耗盡所有重試的一批記錄時，Lambda 會將該批次的詳細資料傳送至此佇列或主題。
- 重試嘗試 - 當函數傳回錯誤時，Lambda 重試的次數上限。這不適用於服務錯誤或調節，其中批次並沒有到達函數。
- 記錄最大存留期 - Lambda 傳送至函數之記錄的最大存留期。
- 在錯誤時分割批次 - 當函數傳回錯誤時，先將批次分割為兩個，再進行重試。您原始的批次大小設定仍會維持不變。
- 每個碎片的並行批次 - 同時處理來自同一個碎片的多個批次。
- 已啟用 - 設定為 true 可啟用事件來源映射。設定為 false 以停止處理記錄。Lambda 會追蹤上次處理的進度，並在重新啟用映射時從該時間點恢復處理。

> **Note**
>
> 對於由 DynamoDB 觸發條件中的 Lambda 調用的 GetRecords API 呼叫，您不需要付費。

若要稍後管理事件來源的組態，請選擇設計工具中的觸發。

建立跨帳戶事件來源映射

Amazon DynamoDB 現在支援以資源為基礎的政策。使用此功能，您可以在另一個帳戶中 AWS 帳戶使用 Lambda 函數處理來自 DynamoDB 串流的資料。

若要使用不同 DynamoDB 串流為您的 Lambda 函數建立事件來源映射 AWS 帳戶，您必須使用資源型政策來設定串流，以授予 Lambda 函數讀取記錄的許可。若要了解如何設定串流以進行跨帳戶存取，請參閱《Amazon DynamoDB 開發人員指南》中的使用跨帳戶 Lambda 函數共用存取權。

使用為 Lambda 函數提供必要許可的資源型政策設定串流後，請使用跨帳戶串流 ARN 建立事件來源映射。您可以在跨帳戶 DynamoDB 主控台的資料表匯出和串流索引標籤下找到串流 ARN。

使用 Lambda 主控台時，請將串流 ARN 直接貼到事件來源映射建立頁面的 DynamoDB 資料表輸入欄位。

注意：不支援跨區域觸發。

使用 DynamoDB 和 Lambda 設定部分批次回應

取用和處理事件來源的串流資料時，依預設，只有在批次成功完成時，Lambda 檢查點才會到批次的最高序號。Lambda 會將所有其他結果視為完全失敗，並重試處理批次，直至達到重試限制。若要在處理串流的批次時允許部分成功，請開啟 `ReportBatchItemFailures`。允許部分成功有助於減少記錄的重試次數，但其不會完全消除在成功記錄中重試的可能性。

若要開啟 `ReportBatchItemFailures`，請在 FunctionResponseTypes 清單中包含枚舉值 **`ReportBatchItemFailures`**。此清單指示已為您的函數啟用哪些回應類型。您可以在建立或更新事件來源映射時設定此清單。

> **Note**
>
> 即使函式程式碼傳回了部分批次失敗回應，除非針對事件來源映射明確啟用 `ReportBatchItemFailures` 功能，否則 Lambda 將不會處理這些回應。

報告語法

設定批次項目失敗的報告時，會傳回 `StreamsEventResponse` 類別，其中包含批次項目失敗的清單。您可以使用 `StreamsEventResponse` 物件，來傳回批次中第一個失敗記錄的序號。您還可以使用正確的回應語法，建立自己的自訂類別。下列 JSON 結構顯示所需的回應語法：

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> 如果 batchItemFailures 陣列包含多個項目，則 Lambda 會使用具有最低序列號的記錄作為檢查點。然後，Lambda 會重試從該檢查點開始的所有記錄。

成功與失敗條件

如果您傳回下列任一項目，Lambda 會將批次視為完全成功：

- 空白 batchItemFailure 清單
- Null batchItemFailure 清單
- 空白 EventResponse
- Null EventResponse

如果您傳回下列任一項目，Lambda 會將批次視為完全失敗：

- 空白字串 itemIdentifier
- Null itemIdentifier
- 具有錯誤金鑰名稱的 itemIdentifier

Lambda 會根據您的重試政策來重試失敗。

將批次平分

如果您的調用失敗且 BisectBatchOnFunctionError 已開啟，則無論您的 ReportBatchItemFailures 設定如何，批次都會被平分。

收到部分批次成功回應且 BisectBatchOnFunctionError 和 ReportBatchItemFailures 均開啟時，批次會依傳回的序號進行平分，並且 Lambda 僅會重試剩餘的記錄。

為了簡化部分批次回應邏輯的實作，請考慮使用 Powertools 中的 Batch Processor 公用程式 AWS Lambda，其會自動為您處理這些複雜性。

以下範例函數程式碼會傳回批次中失敗訊息 ID 的清單：

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
```

```
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)
```

```
type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""

 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }

 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;

import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }
```

```
  return { batchItemFailures: [] };
};
```

使用 TypeScript 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
import {
  DynamoDBBatchResponse,
  DynamoDBBatchItemFailure,
  DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
  event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
  const batchItemFailures: DynamoDBBatchItemFailure[] = [];
  let curRecordSequenceNumber;

  for (const record of event.Records) {
    curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

    if (curRecordSequenceNumber) {
      batchItemFailures.push({
        itemIdentifier: curRecordSequenceNumber,
      });
    }
  }

  return { batchItemFailures: batchItemFailures };
};
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
```

```
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
```

```
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;

    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
```

```
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

使用 Powertools 進行 AWS Lambda 批次處理器

Powertools for 的批次處理器公用程式 AWS Lambda 會自動處理部分批次回應邏輯，降低實作批次失敗報告的複雜性。以下是使用批次處理器的範例：

Python

> **Note**
>
> 如需完整的範例和設定說明，請參閱 batch processor documentation。

使用 AWS Lambda 批次處理器處理 DynamoDB 串流記錄。

```
import json
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities.batch import BatchProcessor, EventType,
 process_partial_response
from aws_lambda_powertools.utilities.data_classes import DynamoDBStreamEvent
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.DynamoDBStreams)
logger = Logger()

def record_handler(record):
    logger.info(record)
    # Your business logic here
    # Raise an exception to mark this record as failed

def lambda_handler(event, context: LambdaContext):
    return process_partial_response(
        event=event,
        record_handler=record_handler,
        processor=processor,
        context=context
    )
```

TypeScript

> **Note**
>
> 如需完整的範例和設定說明，請參閱 batch processor documentation。

使用 AWS Lambda 批次處理器處理 DynamoDB 串流記錄。

```
import { BatchProcessor, EventType, processPartialResponse } from '@aws-lambda-
powertools/batch';
import { Logger } from '@aws-lambda-powertools/logger';
import type { DynamoDBStreamEvent, Context } from 'aws-lambda';

const processor = new BatchProcessor(EventType.DynamoDBStreams);
const logger = new Logger();

const recordHandler = async (record: any): Promise<void> => {
    logger.info('Processing record', { record });
    // Your business logic here
    // Throw an error to mark this record as failed
};

export const handler = async (event: DynamoDBStreamEvent, context: Context) => {
    return processPartialResponse(event, recordHandler, processor, {
        context,
    });
};
```

Java

> **Note**
>
> 如需完整的範例和設定說明，請參閱 batch processor documentation。

使用 AWS Lambda 批次處理器處理 DynamoDB 串流記錄。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
```

```
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import software.amazon.lambda.powertools.batch.BatchMessageHandlerBuilder;
import software.amazon.lambda.powertools.batch.handler.BatchMessageHandler;

public class DynamoDBStreamBatchHandler implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {

    private final BatchMessageHandler<DynamodbEvent, StreamsEventResponse> handler;

    public DynamoDBStreamBatchHandler() {
        handler = new BatchMessageHandlerBuilder()
                .withDynamoDbBatchHandler()
                .buildWithRawMessageHandler(this::processMessage);
    }

    @Override
    public StreamsEventResponse handleRequest(DynamodbEvent ddbEvent, Context
 context) {
        return handler.processBatch(ddbEvent, context);
    }

    private void processMessage(DynamodbEvent.DynamodbStreamRecord
 dynamodbStreamRecord, Context context) {
        // Process the change record
    }
}
```

.NET

> **Note**
>
> 如需完整的範例和設定說明，請參閱 batch processor documentation。

使用 AWS Lambda 批次處理器處理 DynamoDB 串流記錄。

```
using System;
using System.Threading;
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;
using Amazon.Lambda.Serialization.SystemTextJson;
using AWS.Lambda.Powertools.BatchProcessing;
```

```
[assembly: LambdaSerializer(typeof(DefaultLambdaJsonSerializer))]

namespace HelloWorld;

public class Customer
{
    public string? CustomerId { get; set; }
    public string? Name { get; set; }
    public string? Email { get; set; }
    public DateTime CreatedAt { get; set; }
}

internal class TypedDynamoDbRecordHandler : ITypedRecordHandler<Customer>
{
    public async Task<RecordHandlerResult> HandleAsync(Customer customer,
 CancellationToken cancellationToken)
    {
        if (string.IsNullOrEmpty(customer.Email))
        {
            throw new ArgumentException("Customer email is required");
        }

        return await Task.FromResult(RecordHandlerResult.None);
    }
}

public class Function
{
    [BatchProcessor(TypedRecordHandler = typeof(TypedDynamoDbRecordHandler))]
    public BatchItemFailuresResponse HandlerUsingTypedAttribute(DynamoDBEvent _)
    {
        return TypedDynamoDbStreamBatchProcessor.Result.BatchItemFailuresResponse;
    }
}
```

在 Lambda 中保留 DynamoDB 事件來源的捨棄記錄

DynamoDB 事件來源映射的錯誤處理取決於錯誤是在調用函數之前還是在調用函數期間發生：

- 調用前：如果 Lambda 事件來源映射因為限流或其他問題而無法調用函數，它會重試，直到記錄過期或超過事件來源映射上設定的最長存留期 (MaximumRecordAgeInSeconds)。

- 在調用期間：如果調用函數但傳回錯誤，Lambda 會重試，直到記錄過期、超過最長存留期 (MaximumRecordAgeInSeconds)，或達到設定的重試配額 (MaximumRetryAttempts)。對於函數錯誤，您也可以設定 BisectBatchOnFunctionError，將失敗的批次分割為兩個較小的批次，隔離錯誤的記錄並避免逾時。分割批次不會消耗重試配額。

如果錯誤處理措施失敗，Lambda 會捨棄相應記錄，並繼續處理串流中的批次。使用預設設定時，這表示不良的記錄可能會封鎖受影響碎片上的處理長達一天。若要避免此情況，在設定函數的事件來源映射時，請使用合理的重試次數和符合您使用案例的記錄最大保留期。

設定失敗調用的目的地

若要保留失敗的事件來源映射調用記錄，請將目標地新增到函數的事件來源映射中。傳送至該目的地的每筆記錄都是包含失敗調用之中繼資料的 JSON 文件。對於 Amazon S3 目的地，Lambda 還會將整個調用記錄與中繼資料一起傳送。您可以設定任何 Amazon SNS 主題、Amazon SQS 佇列、Amazon S3 儲存貯體或 Kafka 做為目的地。

透過 Amazon S3 目的地，您可以使用 Amazon S3 事件通知功能，在物件上傳至您的目的地 S3 儲存貯體時接收通知。您也可以設定 S3 事件通知來調用另一個 Lambda 函數，以對失敗的批次執行自動處理。

執行角色必須具有目的地的許可：

- 對於 SQS 目的地：sqs：SendMessage
- 對於 SNS 目的地：sns：Publish
- 對於 S3 目的地：s3：PutObject 和 s3：ListBucket
- 對於 Kafka 目的地：kafka-cluster：WriteData

您可以將 Kafka 主題設定為 Kafka 事件來源映射的失敗時目的地。當 Lambda 在耗盡重試嘗試後或記錄超過最長存留期後無法處理記錄時，Lambda 會將失敗的記錄傳送至指定的 Kafka 主題，以供稍後處理。請參閱the section called “Kafka 失敗時目的地”。

如果您已使用自己的 KMS 金鑰為 S3 目的地啟用加密，則函數的執行角色也必須具有呼叫 kms:GenerateDataKey 的許可。如果 KMS 金鑰和 S3 儲存貯體目的地與 Lambda 函數和執行角色位於不同的帳戶中，請將 KMS 金鑰設定為信任執行角色以允許 kms:GenerateDataKey。

若要使用主控台設定失敗時的目的地，請依照下列步驟執行：

1. 開啟 Lambda 主控台中的 函數頁面。

2. 選擇一個函數。
3. 在函數概觀下，選擇新增目的地。
4. 針對來源，請選擇事件來源映射調用。
5. 對於事件來源映射，請選擇針對此函數設定的事件來源。
6. 對於條件，選取失敗時。對於事件來源映射調用，這是唯一可接受的條件。
7. 對於目標類型，請選擇 Lambda 將調用記錄傳送至的目標類型。
8. 對於目的地，請選擇一個資源。
9. 選擇儲存。

您也可以使用 AWS Command Line Interface () 設定失敗時的目的地AWS CLI。例如，下列 create-event-source-mapping 命令會將具有 SQS 失敗時的目的地的事件來源映射新增至 `MyFunction`：

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

下列 update-event-source-mapping 命令會更新事件來源映射，以在嘗試兩次重試後，或在記錄超過一小時時，將失敗的調用記錄傳送至 SNS 目的地。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

系統會以非同步的方式套用更新的設定，在處理完成之前不會反映在輸出中。使用 get-event-source-mapping 命令檢視目前的狀態。

若要移除目的地，請提供空白字串作為 `destination-config` 參數的引數：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Amazon S3 目的地的安全最佳實務

刪除已設定為目的地的 S3 儲存貯體而不從函數的組態中移除目的地，可能會產生安全風險。如果其他使用者知道目的地儲存貯體的名稱，他們可以在其 AWS 帳戶中重新建立儲存貯體。失敗調用的記錄會被傳送到其儲存貯體，可能公開來自您函數的資料。

> **Warning**
>
> 為了確保無法將來自函數的調用記錄傳送到另一個 中的 S3 儲存貯體 AWS 帳戶，請將條件新增至函數的執行角色，以限制您帳戶中儲存貯體的`s3:PutObject`許可。

下列範例顯示的 IAM 政策，將函數的 `s3:PutObject` 許可限制為帳戶中的儲存貯體。此政策也為 Lambda 提供了使用 S3 儲存貯體做為目的地所需的 `s3:ListBucket` 許可。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
    ]
}
```

若要使用 AWS 管理主控台 或 將許可政策新增至函數的執行角色 AWS CLI，請參閱下列程序中的指示：

Console

若要將許可政策新增至函數的執行角色 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選取您要修改其執行角色的 Lambda 函數。
3. 在組態索引標籤中，選擇許可。
4. 在執行角色索引標籤中，選取函數的角色名稱，以開啟角色的 IAM 主控台頁面。
5. 透過下列步驟將許可政策新增至角色：
   a. 在許可政策窗格中，選擇新增許可 ，然後選取建立內嵌政策。
   b. 在政策編輯器中，選取 JSON。
   c. 將您要新增的政策貼入編輯器 (取代現有的 JSON)，然後選擇下一步。
   d. 在政策詳細資訊下，輸入政策名稱。
   e. 選擇建立政策。

AWS CLI

若要將許可政策新增至函數的執行角色 (CLI)

1. 建立具有所需許可的 JSON 政策文件，並將其儲存在本機目錄中。
2. 使用 IAM `put-role-policy` CLI 命令，將許可新增至函數的執行角色。從您儲存 JSON 政策文件的目錄執行下列命令，並將角色名稱、政策名稱和政策文件取代為您自己的值。

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Amazon SNS 和 Amazon SQS 調用記錄範例

下列範例顯示了 Lambda 傳送至 DynamoDB 串流的 SQS 或 SNS 目的地的一條調用記錄。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
```

```
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:13:49.717Z",
    "DDBStreamBatchInfo": {
        "shardId": "shardId-00000001573689847184-864758bb",
        "startSequenceNumber": "800000000003126276362",
        "endSequenceNumber": "800000000003126276362",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:13:19Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:13:19Z",
        "batchSize": 1,
        "streamArn": "arn:aws:dynamodb:us-east-2:123456789012:table/mytable/
stream/2019-11-14T00:04:06.388"
    }
}
```

您可以使用此資訊來從串流擷取受影響的記錄，以進行疑難排解。實際的記錄不包含在內，因此您必須處理此記錄，並在因過期而遺失之前從資料串流中擷取它們。

Amazon S3 調用記錄範例

下列範例顯示了 Lambda 傳送至 DynamoDB 串流的 S3 儲存貯體目的地的一條調用記錄。除了上一個 SQS 和 SNS 目的地範例中的所有欄位之外，此 `payload` 欄位還包含原始調用記錄做為逸出 JSON 字串。

```
{
    "requestContext": {
        "requestId": "316aa6d0-8154-xmpl-9af7-85d5f4a6bc81",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
```

```
    "version": "1.0",
    "timestamp": "2019-11-14T00:13:49.717Z",
    "DDBStreamBatchInfo": {
        "shardId": "shardId-000000015736898471 84-864758bb",
        "startSequenceNumber": "80000000000 3126276362",
        "endSequenceNumber": "80000000000 3126276362",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:13:19Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:13:19Z",
        "batchSize": 1,
        "streamArn": "arn:aws:dynamodb:us-east-2:123456789012:table/mytable/
stream/2019-11-14T00:04:06.388"
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

包含調用記錄的 S3 物件使用以下命名慣例：

```
aws/lambda/<ESM-UUID>/<shardID>/YYYY/MM/DD/YYYY-MM-DDTHH.MM.SS-<Random UUID>
```

在 Lambda 中實作有狀態的 DynamoDB 串流處理

Lambda 函數可執行持續串流處理應用程式。串流表示持續在應用程式中流動的無限制資料。若要分析此持續更新輸入中的資訊，您可以使用定義的時段來限制包含的記錄。

輪轉時段是定期開啟和關閉的不同時段。依預設，Lambda 調用是無狀態的，您無法在沒有外部資料庫的情況下，將其用於處理多個持續調用的資料。然而，使用輪轉時段，您可以在不同的調用間維護狀態。此狀態包含之前為目前時段處理之訊息的彙總結果。狀態可以是每個分區最多 1 MB。如果超過該大小，則 Lambda 會提前終止時段。

串流中的每個記錄都屬於一個特定時段。Lambda 至少會處理一次每筆記錄，但不保證每筆記錄只會處理一次。在極少數情況下，例如錯誤處理，某些記錄可能會處理多次。第一次時一律會依序處理記錄。如果多次處理記錄，則可能不會按順序處理。

彙總與處理

調用您的使用者管理函數進行彙總，以及處理該彙總的最終結果。Lambda 會彙總時段中接收的所有記錄。您可以在多個批次中接收這些記錄，各自作為單獨的調用。每次調用會收到一個狀態。因此，當使用輪轉時段時，您的 Lambda 函數回應必須包含 `state` 屬性。如果回應不包含 `state` 屬性，Lambda 會將此視為失敗的調用。為了滿足此條件，您的函數可以返回一個 `TimeWindowEventResponse` 物件，它具有下列 JSON 形狀：

Example **`TimeWindowEventResponse`** 值

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> 對於 Java 函數，我們建議使用 `Map<String, String>` 來表示狀態。

在時段結束時，標記 `isFinalInvokeForWindow` 會設定為 `true` 以指示這是最終狀態，並且可隨時進行處理。處理完成後，時段結束並完成最終調用，然後丟棄該狀態。

在時段結束時，Lambda 會針對彙總結果上的動作使用最終處理。您的最終處理將同步調用。成功調用後，您的函數檢查點序號和串流處理將會繼續。如果調用失敗，則您的 Lambda 函數會暫停進一步處理，直至成功調用。

Example DynamodbTimeWindowEvent

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"New item!"
```

```
            },
            "Id":{
               "N":"101"
            }
         },
         "SequenceNumber":"111",
         "SizeBytes":26,
         "StreamViewType":"NEW_AND_OLD_IMAGES"
      },
      "eventSourceARN":"stream-ARN"
   },
   {
      "eventID":"2",
      "eventName":"MODIFY",
      "eventVersion":"1.0",
      "eventSource":"aws:dynamodb",
      "awsRegion":"us-east-1",
      "dynamodb":{
         "Keys":{
            "Id":{
               "N":"101"
            }
         },
         "NewImage":{
            "Message":{
               "S":"This item has changed"
            },
            "Id":{
               "N":"101"
            }
         },
         "OldImage":{
            "Message":{
               "S":"New item!"
            },
            "Id":{
               "N":"101"
            }
         },
         "SequenceNumber":"222",
         "SizeBytes":59,
         "StreamViewType":"NEW_AND_OLD_IMAGES"
      },
      "eventSourceARN":"stream-ARN"
```

```
      },
      {
         "eventID":"3",
         "eventName":"REMOVE",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "OldImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"333",
            "SizeBytes":38,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      }
   ],
    "window": {
        "start": "2020-07-30T17:00:00Z",
        "end": "2020-07-30T17:05:00Z"
    },
    "state": {
        "1": "state1"
    },
    "shardId": "shard123456789",
    "eventSourceARN": "stream-ARN",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

Configuration

您可以在建立或更新事件來源對映時設定輪轉時段。若要設定輪轉時段，請以秒為單位指定時段 (TumblingWindowInSeconds)。下列 example AWS Command Line Interface (AWS CLI) 命令會建立輪轉時段為 120 秒的串流事件來源映射。針對彙總與處理定義的 Lambda 函數命名為 `tumbling-window-example-function`。

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table/
stream/2024-06-10T19:26:16.525 \
--function-name tumbling-window-example-function \
--starting-position TRIM_HORIZON \
--tumbling-window-in-seconds 120
```

Lambda 根據記錄插入串流的時間，確定輪轉時段邊界。所有記錄都有 Lambda 在邊界確定中使用的近似時間戳記。

輪轉時段彙總不支援重新分區。分區結束後，Lambda 會考慮關閉時段，並且子分區會以新的狀態開始自己的時段。

輪轉時段完全支援現有的重試政策 `maxRetryAttempts` 和 `maxRecordAge`。

Example Handler.py - 彙總與處理

下列 Python 函數示範了如何彙總，然後處理您的最終狀態：

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
```

```
    for record in event['Records']:
        state[record['dynamodb']['NewImage']['Id']] = state.get(record['dynamodb']
['NewImage']['Id'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

Amazon DynamoDB 事件來源映射的 Lambda 參數

所有 Lambda 事件來源類型都會共用相同的 CreateEventSourceMapping 和 UpdateEventSourceMapping API 操作。但是，只有一些參數適用於 DynamoDB Streams。

參數	必要	預設	備註
BatchSize	否	100	上限：10,000
BisectBatchOnFunctionError	N	false	無
DestinationConfig	N	N/A	捨棄記錄的標準 Amazon SQS 佇列或標準 Amazon SNS 主題目的地
已啟用	N	true	無
EventSourceArn	Y	N/A	資料串流或串流消費者的 ARN
FilterCriteria	N	N/A	控制 Lambda 將哪些事件傳送至您的函數
FunctionName	是	N/A	無
FunctionResponseTypes	N	N/A	若要讓函數報告批次中的特定失敗，請將值 `ReportBatchItemFailures` 包含在 `FunctionResponseTypes`

參數	必要	預設	備註
			中。如需詳細資訊，請參閱使用 DynamoDB 和 Lambda 設定部分批次回應。
MaximumBatchingWindowInSeconds	N	0	無
MaximumRecordAgeInSeconds	N	-1	-1 代表無限：系統會重試失敗的記錄，直到記錄過期為止。DynamoDB Streams 的資料保留限制為 24 小時。 下限：-1 上限：604,800
MaximumRetryAttempts	N	-1	-1 代表無限：系統會重試失敗的記錄，直到記錄過期為止 下限：0 上限：10,000
ParallelizationFactor	N	1	上限：10
StartingPosition	Y	N/A	TRIM_HORIZON 或 LATEST
TumblingWindowInSeconds	N	N/A	下限：0 上限：900

搭配 DynamoDB 事件來源使用事件篩選

您可以使用事件篩選來控制 Lambda 將哪些記錄從串流或佇列中傳送至函數。如需事件篩選運作方式的一般資訊，請參閱the section called “事件篩選”。

本節重點介紹 DynamoDB 事件來源的事件篩選。

> **Note**
>
> DynamoDB 事件來源映射僅支援依據 dynamodb 鍵進行篩選。

主題

- DynamoDB 事件
- 使用資料表屬性進行篩選
- 使用布林表達式進行篩選
- 使用 Exists 運算子
- DynamoDB 篩選的 JSON 格式

DynamoDB 事件

假設您有一個包含主索引鍵 `CustomerName` 和屬性 `AccountManager` 與 `PaymentTerms` 的 DynamoDB 表格。以下顯示 DynamoDB 表格串流中的範例記錄。

```
{
      "eventID": "1",
      "eventVersion": "1.0",
      "dynamodb": {
          "ApproximateCreationDateTime": "1678831218.0",
          "Keys": {
              "CustomerName": {
                  "S": "AnyCompany Industries"
              }
          },
          "NewImage": {
              "AccountManager": {
                  "S": "Pat Candella"
              },
              "PaymentTerms": {
```

```
                "S": "60 days"
            },
            "CustomerName": {
                "S": "AnyCompany Industries"
            }
        },
        "SequenceNumber": "111",
        "SizeBytes": 26,
        "StreamViewType": "NEW_IMAGE"
    }
}
```

若要根據 DynamoDB 表格中的索引鍵值和屬性值進行篩選，請使用記錄中的 dynamodb 索引鍵。下列各節提供不同的篩選條件類型範例。

使用資料表索引進行篩選

假設您希望函數僅處理主索引鍵 `CustomerName` 為 "AnyCompany Industries" 的記錄。`FilterCriteria` 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"Keys\" : { \"CustomerName\" : { \"S\" :
 [ \"AnyCompany Industries\" ] } } } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 `Pattern` 在純文字 JSON 中擴展的值。

```
{
    "dynamodb": {
        "Keys": {
            "CustomerName": {
                "S": [ "AnyCompany Industries" ]
                }
            }
        }
}
```

您可以使用主控台、AWS CLI 或 AWS SAM 範本新增篩選條件。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 來建立具有這些篩選條件標準的新事件來源映射，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"Keys\" :
 { \"CustomerName\" : { \"S\" : [ \"AnyCompany Industries\" ] } } } }"}]}'
```

AWS SAM

若要使用 AWS SAM 新增此篩選條件，請將下列程式碼片段新增到事件來源的 YAML 範本。

```
FilterCriteria:
   Filters:
     - Pattern: '{ "dynamodb" : { "Keys" : { "CustomerName" : { "S" : [ "AnyCompany
 Industries" ] } } } }'
```

使用資料表屬性進行篩選

使用 DynamoDB 時，您也可以使用 `NewImage` 和 `OldImage` 索引鍵來篩選屬性值。假設您想篩選最新表格映像中的 `AccountManager` 屬性為 "Pat Candella" 或 "Shirley Rodriguez" 的記錄。`FilterCriteria` 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\", \"Shirley Rodriguez\" ] } } } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 `Pattern` 在純文字 JSON 中擴展的值。

```
{
    "dynamodb": {
        "NewImage": {
            "AccountManager": {
                "S": [ "Pat Candella", "Shirley Rodriguez" ]
            }
        }
    }
}
```

您可以使用主控台、AWS CLI 或 AWS SAM 範本新增篩選條件。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat Candella",
 "Shirley Rodriguez" ] } } } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 來建立具有這些篩選條件標準的新事件來源映射，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
```

```
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\", \"Shirley Rodriguez
\" ] } } } }"}]}'
```

AWS SAM

若要使用 AWS SAM 新增此篩選條件，請將下列程式碼片段新增到事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella", "Shirley Rodriguez" ] } } } }'
```

使用布林表達式進行篩選

您也可以使用布林值 AND 運算式建立篩選條件。這些運算式可以同時包含資料表的索引鍵和屬性參數。假設您想篩選記錄，其中 `AccountManager` 的 `NewImage` 值是「Pat Candella」，`OldImage` 值是「Terry Whitlock」。`FilterCriteria` 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"dynamodb\" : { \"NewImage\" : { \"AccountManager\" : { \"S
\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" : { \"OldImage\" : { \"AccountManager
\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 `Pattern` 在純文字 JSON 中擴展的值。

```
{
    "dynamodb" : {
```

```
        "NewImage" : {
            "AccountManager" : {
                "S" : [
                    "Pat Candella"
                ]
            }
        }
    },
    "dynamodb": {
        "OldImage": {
            "AccountManager": {
                "S": [
                    "Terry Whitlock"
                ]
            }
        }
    }
}
```

您可以使用主控台、AWS CLI 或 AWS SAM 範本新增篩選條件。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 來建立具有這些篩選條件標準的新事件來源映射，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:dynamodb:us-east-2:123456789012:table/my-table \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"dynamodb\" : { \"NewImage
\" : { \"AccountManager\" : { \"S\" : [ \"Pat Candella\" ] } } } , \"dynamodb\" :
 { \"OldImage\" : { \"AccountManager\" : { \"S\" : [ \"Terry Whitlock\" ] } } } }
 "}]}'
```

AWS SAM

若要使用 AWS SAM 新增此篩選條件，請將下列程式碼片段新增到事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "dynamodb" : { "NewImage" : { "AccountManager" : { "S" : [ "Pat
 Candella" ] } } } , "dynamodb" : { "OldImage" : { "AccountManager" : { "S" :
 [ "Terry Whitlock" ] } } } }'
```

> **Note**
>
> DynamoDB 事件篩選不支援使用數值運算子 (數值等於和數值範圍)。即使資料表中的項目儲存為數字，這些參數也會轉換為 JSON 記錄物件中的字串。

使用 Exists 運算子

由於來自 DynamoDB 的 JSON 事件物件的構造方式，使用 Exists 運算子需要特別注意。Exists 運算子只能在事件 JSON 的分葉節點上運作，因此如果您的篩選模式使用 Exists 來測試中繼節點，則無法運作。請考慮下列 DynamoDB 資料表項目。

```
{
  "UserID": {"S": "12345"},
  "Name": {"S": "John Doe"},
  "Organizations": {"L": [
      {"S":"Sales"},
      {"S":"Marketing"},
      {"S":"Support"}
    ]
```

```
  }
}
```

您可能想要建立如下所示的篩選條件模式，以測試包含 "Organizations" 的事件：

```
{ "dynamodb" : { "NewImage" : { "Organizations" : [ { "exists": true } ] } } }
```

不過，此篩選條件模式永遠不會傳回相符項目，因為 "Organizations" 不是分葉節點。下列範例顯示了如何正確使用 Exists 運算子來建構所需的篩選條件模式：

```
{ "dynamodb" : { "NewImage" : {"Organizations": {"L": {"S": [ {"exists":
 true } ] } } } } }
```

DynamoDB 篩選的 JSON 格式

若要正確篩選來自 DynamoDB 來源的事件，資料欄位和資料欄位 (dynamodb) 的篩選條件標準都必須是有效的 JSON 格式。如果其中一個欄位不是有效的 JSON 格式，則 Lambda 會捨棄訊息或擲回例外狀況。下表摘要說明特定行為：

傳入資料格式	資料屬性的篩選條件模式格式	產生的動作
有效的 JSON	有效的 JSON	根據您的篩選條件標準之 Lambda 篩選條件。
有效的 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	非 JSON	Lambda 會在事件來源映射建立或更新時擲回例外狀況。資料屬性的篩選條件模式必須是有效的 JSON 格式。
非 JSON	有效的 JSON	Lambda 捨棄記錄。
非 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。

傳入資料格式	資料屬性的篩選條件模式格式	產生的動作
非 JSON	非 JSON	Lambda 會在事件來源映射建立或更新時擲回例外狀況。資料屬性的篩選條件模式必須是有效的 JSON 格式。

教學課程： AWS Lambda 搭配 Amazon DynamoDB 串流使用

在此教學課程中，您建立 Lambda 函數以從 Amazon DynamoDB Streams 中取用事件。

先決條件

安裝 AWS Command Line Interface

如果您尚未安裝 AWS Command Line Interface，請依照安裝或更新最新版本的 AWS CLI中的步驟進行安裝。

本教學課程需使用命令列終端機或 Shell 來執行命令。在 Linux 和 macOS 中，使用您偏好的 Shell 和套件管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。

建立執行角色

建立 執行角色，授予函數存取 AWS 資源的許可。

若要建立執行角色

1. 在 IAM 主控台中開啟角色頁面。
2. 選擇建立角色。
3. 建立具備下列屬性的角色。
   - 信任實體 - Lambda。

- 許可 - AWSLambdaDynamoDBExecutionRole。
- 角色名稱 - **lambda-dynamodb-role**。

AWSLambdaDynamoDBExecutionRole 具備函數自 DynamoDB 讀取項目以及寫入日誌到 CloudWatch Logs 時所需的許可。

建立函數

建立 Lambda 函數來處理 DynamoDB 事件。函數程式碼將一些傳入的事件資料寫入 CloudWatch Logs。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
```

```
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
```

```
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }

 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 DynamoDB 事件。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
```

```
public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 來使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
```

```
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

使用 TypeScript 搭配 Lambda 來使用 DynamoDB 事件。

```
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來使用 DynamoDB 事件。

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;
```

```
require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;

    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
```

```
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 DynamoDB 事件。

```
import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來使用 DynamoDB 事件。

```
def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來使用 DynamoDB 事件。

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
```

```
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
```

```
    lambda_runtime::run(func).await?;
    Ok(())

}
```

建立函數

1. 將範本程式碼複製到名為 `example.js` 的檔案。
2. 建立部署套件。

```
zip function.zip example.js
```

3. 使用 `create-function` 命令建立一個 Lambda 函數。

```
aws lambda create-function --function-name ProcessDynamoDBRecords \
    --zip-file fileb://function.zip --handler example.handler --runtime nodejs24.x
 \
    --role arn:aws:iam::111122223333:role/lambda-dynamodb-role
```

測試 Lambda 函數

在此步驟中，您會使用 `invoke` AWS Lambda CLI 命令和下列範例 DynamoDB 事件手動叫用 Lambda 函數。將下列內容複製到名為 `input.txt` 的檔案。

Example input.txt

```
{
   "Records":[
      {
         "eventID":"1",
         "eventName":"INSERT",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
```

```
            },
            "NewImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"111",
            "SizeBytes":26,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"2",
         "eventName":"MODIFY",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "NewImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "OldImage":{
               "Message":{
                  "S":"New item!"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"222",
```

```
            "SizeBytes":59,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      },
      {
         "eventID":"3",
         "eventName":"REMOVE",
         "eventVersion":"1.0",
         "eventSource":"aws:dynamodb",
         "awsRegion":"us-east-1",
         "dynamodb":{
            "Keys":{
               "Id":{
                  "N":"101"
               }
            },
            "OldImage":{
               "Message":{
                  "S":"This item has changed"
               },
               "Id":{
                  "N":"101"
               }
            },
            "SequenceNumber":"333",
            "SizeBytes":38,
            "StreamViewType":"NEW_AND_OLD_IMAGES"
         },
         "eventSourceARN":"stream-ARN"
      }
   ]
}
```

執行以下 invoke 命令。

```
aws lambda invoke --function-name ProcessDynamoDBRecords \
    --cli-binary-format raw-in-base64-out \
    --payload file://input.txt outputfile.txt
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

該函數會在回應本文中傳回字串 `message` 訊息。

在 `outputfile.txt` 檔案中確認輸出。

建立啟用串流的 DynamoDB 資料表

建立啟用串流的 Amazon DynamoDB 資料表。

建立 DynamoDB 資料表

1. 開啟 DynamoDB 主控台。
2. 選擇 Create Table (建立資料表)。
3. 根據下列設定建立資料表。
   - Table name (資料表名稱) – **`lambda-dynamodb-stream`**
   - Primary key (主要金鑰) - **`id`** (字串)
4. 選擇建立。

啟用串流

1. 開啟 DynamoDB 主控台。
2. 選擇 資料表。
3. 選擇 lambda-dynamodb-stream 資料表。
4. 在 匯出與串流 下，選擇 DynamoDB 串流詳細資訊 。
5. 選擇 Turn on (開啟)。
6. 對於檢視類型，選擇僅索引鍵屬性。
7. 選擇開啟串流。

寫下串流 ARN。在下個步驟將串流與您的 Lambda 函數相關聯時會需要用到它。如需啟用串流的詳細資訊，請參閱使用 DynamoDB Streams 擷取資料表活動。

在 中新增事件來源 AWS Lambda

在 AWS Lambda中建立事件來源映射。事件來源映射可將 DynamoDB 串流與您的 Lambda 函數相關聯。建立此事件來源映射後， 會 AWS Lambda 開始輪詢串流。

執行下列 AWS CLI `create-event-source-mapping` 命令。命令執行後，請記下 UUID。使用任何命令時，您會需要此 UUID 以參考到事件來源映射，例如當刪除事件來源映射的時候。

```
aws lambda create-event-source-mapping --function-name ProcessDynamoDBRecords \
    --batch-size 100 --starting-position LATEST --event-source DynamoDB-stream-arn
```

這會在指定的 DynamoDB 串流和 Lambda 函數之間建立映射。您可以將 DynamoDB 串流與多個 Lambda 函數相關聯，也可以將相同的 Lambda 函數與多個串流相關聯。但是 Lambda 函數會共用其所共有之串流的讀取傳送量。

您可以執行下列命令來取得事件來源映射的清單。

```
aws lambda list-event-source-mappings
```

該清單傳回所有您建立的事件來源映射，並針對每個映射顯示 `LastProcessingResult` 和其他內容。此欄位可用在發生問題時，提供資訊豐富的訊息。例如 `No records processed`（表示 AWS Lambda 尚未開始輪詢或串流中沒有記錄）和 `OK`（表示 AWS Lambda 成功從串流讀取記錄並叫用您的 Lambda 函數）的值表示沒有問題。如果發生問題，您會收到錯誤訊息。

如果您有許多事件來源映射，請使用函數式稱參數來縮小結果。

```
aws lambda list-event-source-mappings --function-name ProcessDynamoDBRecords
```

測試設定

測試端對端的體驗。當您更新資料表時，DynamoDB 會將事件記錄寫入串流。當 AWS Lambda 輪詢串流時，若偵測到串流上的新記錄，便會透過傳送事件到函數來代表您調用 Lambda 函數。

1. 在 DynamoDB 主控台上針對資料表新增、更新、刪除項目。DynamoDB 會將這些動作的記錄寫入串流。
2. AWS Lambda 會輪詢串流，並在偵測到串流的更新時，透過傳入在串流中找到的事件資料來叫用 Lambda 函數。
3. 您的函數會執行並在 Amazon CloudWatch 中建立日誌。您可以驗證在 Amazon CloudWatch 主控台中報告的日誌。

後續步驟

此教學課程介紹了使用 Lambda 處理 DynamoDB 串流事件的基礎知識。對於生產工作負載，建議實作部分批次回應邏輯，更有效率地處理個別記錄失敗。適用於 的 Powertools 批次處理器公用程式 AWS Lambda 可在 Python、TypeScript、.NET 和 Java 中使用，並為此提供強大的解決方案，可自動處理部分批次回應的複雜性，並減少成功處理記錄的重試次數。

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。透過刪除您不再使用 AWS 的資源，您可以避免不必要的 費用 AWS 帳戶。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **confirm**，然後選擇刪除。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇刪除。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 DynamoDB 資料表

1. 開啟 DynamoDB 主控台的 資料表頁面 。
2. 選取您建立的資料表。
3. 選擇 刪除 。
4. 在文字方塊中輸入 **delete**。
5. 選擇 刪除資料表 。

使用 Lambda 函數處理 Amazon EC2 生命週期事件

您可以使用 AWS Lambda 處理來自 Amazon Elastic Compute Cloud 的生命週期事件，並管理 Amazon EC2 資源。Amazon EC2 就生命週期事件向 Amazon EventBridge (CloudWatch Events) 傳送事件，例如執行個體狀態變更、Amazon Elastic Block Store 磁碟區快照完成，或排定 Spot 執行個體將終止。您可以設定 EventBridge (CloudWatch Events)，將這些事件轉遞至 Lambda 函數以進行處理。

EventBridge (CloudWatch Events) 會非同步地叫用您的 Lambda 函數與來自 Amazon EC2 的事件文件。

Example執行個體生命週期事件

```
{
    "version": "0",
    "id": "b6ba298a-7732-2226-xmpl-976312c1a050",
    "detail-type": "EC2 Instance State-change Notification",
    "source": "aws.ec2",
    "account": "111122223333",
    "time": "2019-10-02T17:59:30Z",
    "region": "us-east-1",
    "resources": [
        "arn:aws:ec2:us-east-1:111122223333:instance/i-0c314xmplcd5b8173"
    ],
    "detail": {
        "instance-id": "i-0c314xmplcd5b8173",
        "state": "running"
    }
}
```

如需設定事件的詳細資訊，請參閱依排程調用 Lambda 函數。如需處理 Amazon EBS 快照通知的範例函數，請參閱適用於 Amazon EBS 的 EventBridge 排程器。

您也可以使用 AWS 開發套件，搭配 Amazon EC2 API 來管理執行個體和其他資源。

將許可授予 EventBridge (CloudWatch Events)

若要處理來自 Amazon EC2 的生命週期事件，EventBridge (CloudWatch Events) 需要叫用您函數的許可。此許可來自函數的資源型政策。如果您使用 EventBridge (CloudWatch Events) 主控台設定事件觸發條件，則主控台會代您更新資源型政策。否則，新增如下的聲明：

Example Amazon EC2 生命週期通知的資源型政策聲明

```
{
  "Sid": "ec2-events",
  "Effect": "Allow",
  "Principal": {
    "Service": "events.amazonaws.com"
  },
  "Action": "lambda:InvokeFunction",
  "Resource": "arn:aws:lambda:us-east-1:12456789012:function:my-function",
  "Condition": {
    "ArnLike": {
      "AWS:SourceArn": "arn:aws:events:us-east-1:12456789012:rule/*"
    }
  }
}
```

若要新增聲明，請使用 `add-permission` AWS CLI 命令。

```
aws lambda add-permission --action lambda:InvokeFunction --statement-id ec2-events \
--principal events.amazonaws.com --function-name my-function --source-arn
 'arn:aws:events:us-east-1:12456789012:rule/*'
```

如果您的函數使用 AWS 開發套件來管理 Amazon EC2 資源，請將 Amazon EC2 許可新增到函數的執行角色。

使用 Lambda 處理 Application Load Balancer 請求

您可以使用 Lambda 函數處理來自 Application Load Balancer 的請求。Elastic Load Balancing 支援 Lambda 函數作為 Application Load Balancer 的目標。使用負載平衡器規則，根據路徑或標頭值將 HTTP 請求路由至特定函式。由您的 Lambda 函數處理請求並傳回 HTTP 回應。

Elastic Load Balancer 將使用含有請求內文和中繼資料的事件，同步叫用您的 Lambda 函數。

Example Application Load Balancer 請求事件

```
{
    "requestContext": {
        "elb": {
            "targetGroupArn": "arn:aws:elasticloadbalancing:us-
east-1:123456789012:targetgroup/lambda-279XGJDqGZ5rsrHC2Fjr/49e9d65c45c6791a"
        }
    },
    "httpMethod": "GET",
    "path": "/lambda",
    "queryStringParameters": {
        "query": "1234ABCD"
    },
    "headers": {
        "accept": "text/html,application/xhtml+xml,application/xml;q=0.9,image/
webp,image/apng,*/*;q=0.8",
        "accept-encoding": "gzip",
        "accept-language": "en-US,en;q=0.9",
        "connection": "keep-alive",
        "host": "lambda-alb-123578498.us-east-1.elb.amazonaws.com",
        "upgrade-insecure-requests": "1",
        "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/71.0.3578.98 Safari/537.36",
        "x-amzn-trace-id": "Root=1-5c536348-3d683b8b04734faae651f476",
        "x-forwarded-for": "72.12.164.125",
        "x-forwarded-port": "80",
        "x-forwarded-proto": "http",
        "x-imforwards": "20"
    },
    "body": "",
    "isBase64Encoded": False
}
```

您的函數會處理事件，並將 JSON 格式的回應文件傳回給負載平衡器。Elastic Load Balancing 會將文件轉換成 HTTP 成功或錯誤回應，並傳回給使用者。

Example回應文件格式

```
{
    "statusCode": 200,
    "statusDescription": "200 OK",
    "isBase64Encoded": False,
    "headers": {
        "Content-Type": "text/html"
    },
    "body": "<h1>Hello from Lambda!</h1>"
}
```

若要設定 Application Load Balancer 作為函數觸發條件，請授予 Elastic Load Balancing 執行函數的許可、建立目標群組將請求路由至函數，並且為負載平衡器加入規則以傳送請求至目標群組。

使用 `add-permission` 命令，將許可陳述式加入至函式以資源為基礎的政策。

```
aws lambda add-permission --function-name alb-function \
--statement-id load-balancer --action "lambda:InvokeFunction" \
--principal elasticloadbalancing.amazonaws.com
```

您應該會看到下列輸出：

```
{
    "Statement": "{\"Sid\":\"load-balancer\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"elasticloadbalancing.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction
\",\"Resource\":\"arn:aws:lambda:us-west-2:123456789012:function:alb-function\"}"
}
```

如需有關設定 Application Load Balancer 接聽程式和目標群組的說明，請參閱 Application Load Balancer 使用者指南中的 Lambda 函數作為目標。

Powertools for AWS Lambda 的事件處理常式

來自 Powertools for AWS Lambda 工具組的事件處理常式，可在撰寫 Application Load Balancer 調用的 Lambda 函數時，提供路由、中介軟體、CORS 組態、OpenAPI 規格產生、請求驗證、錯誤處理和其他實用功能。該事件處理常式公用程式適用於 Python。如需詳細資訊，請參閱 Powertools for AWS Lambda (Python) 文件中的事件處理常式 REST API。

Python

```
import requests
from requests import Response

from aws_lambda_powertools import Logger, Tracer
from aws_lambda_powertools.event_handler import ALBResolver
from aws_lambda_powertools.logging import correlation_paths
from aws_lambda_powertools.utilities.typing import LambdaContext

tracer = Tracer()
logger = Logger()
app = ALBResolver()


@app.get("/todos")
@tracer.capture_method
def get_todos():
    todos: Response = requests.get("https://jsonplaceholder.typicode.com/todos")
    todos.raise_for_status()

    # for brevity, we'll limit to the first 10 only
    return {"todos": todos.json()[:10]}


# You can continue to use other utilities just as before
@logger.inject_lambda_context(correlation_id_path=correlation_paths.APPLICATION_LOAD_BALANCER)
@tracer.capture_lambda_handler
def lambda_handler(event: dict, context: LambdaContext) -> dict:
    return app.resolve(event, context)
```

依排程調用 Lambda 函數

Amazon EventBridge 排程器是無伺服器排程器，可讓您從單一受管的中央服務建立、執行及管理任務。使用 EventBridge 排程器，您可以使用週期性模式的 Cron 和 Rate 運算式來建立排程，或設定一次性呼叫。您可以設定彈性的交付時段、定義重試次數上限，以及設定未處理事件的最長保留時間。

當您使用 Lambda 設定 EventBridge 排程器時，EventBridge 排程器會以非同步方式調用 Lambda 函數。本頁面說明如何使用 EventBridge 排程器，依照排程調用 Lambda 函數。

設定執行角色

當您建立新排程時，EventBridge 排程器必須具有代表您調用其目標 API 操作的權限。您可以使用執行角色，授與 EventBridge 排程器這些許可。排程執行角色所連接的許可政策會定義哪些是必要許可。許可是否為必要權限，取決於您希望 EventBridge 排程器調用的目標 API。

您在 EventBridge 排程器主控台建立排程時 (如以下程序所述)，EventBridge 排程器會根據您選取的目標自動設定執行角色。如果您要使用 EventBridge 排程器 SDK、AWS CLI 或 CloudFormation 建立排程，您必須具備現有的執行角色，授與 EventBridge 排程器調用目標所需的許可。如需手動設定排程執行角色的詳細資訊，請參閱《EventBridge 排程器使用者指南》中的設定執行角色。

建立排程

使用主控台建立排程

1. 前往 https://console.aws.amazon.com/scheduler/home 開啟 Amazon EventBridge 排程器。
2. 在排程頁面上，選擇建立排程。
3. 在指定排程詳細資訊頁面的排程名稱和描述區段中，執行以下動作：

   a. 在排程名稱中，輸入排程的名稱，例如 **`MyTestSchedule`**。

   b. (選用) 在描述中，輸入對排程的描述，例如 **`My first schedule`**。

   c. 針對排程群組，從下拉式清單中選擇排程群組。如果您沒有群組，請選擇預設值。若要建立排程群組，請選擇建立自己的排程。

      您可以使用排程群組，為不同群組的排程加上標籤。

4. • 選擇排程選項。

頻率	執行此作業...	
一次性排程 一次性排程只會在您指定的日期與時間調用目標一次。	針對日期和時間執行以下動作： • 依 YYYY/MM/DD 格式輸入有效日期。 • 依 hh:mm 格式輸入時間戳記 (24 小時)。 • 針對時區選擇時區。	

頻率	執行此作業...	
週期性排程 週期性排程會依您指定的頻率，使用 cron 或 Rate 運算式調用目標。	a. 在排程模式中，執行下列其中一項動作： • 若要使用 Cron 運算式定義排程，請選擇 Cron 排程，然後輸入 Cron 運算式。 • 若要使用 Rate 表達式定義排程，請選擇 Rate 排程，然後輸入 Rate 表達式。 如需 Cron 和 Rate 運算式的詳細資訊，請參閱《Amazon EventBridge 排程器使用者指南》中的 EventBridge 排程器上的排程類型。 b. 對於彈性時段，選擇關閉可關閉此選項，或者也能選擇其中一個預先定義的時間範圍。例如，如果您選擇 15 分鐘並設定週期性排程，每小時調用目標一次，則排程會在每小時一開始的 15 分鐘內執行。	

5. (選用) 如果您在上一個步驟中選擇週期性排程，請在時間範圍區段執行以下動作：

   a. 針對時區選擇時區。

   b. 對於開始日期和時間，依 YYYY/MM/DD 格式輸入有效日期，接著依 24 小時的 hh:mm 格式指定時間戳記。

   c. 對於結束日期和時間，依 YYYY/MM/DD 格式輸入有效日期，接著依 24 小時的 hh:mm 格式指定時間戳記。

6. 選擇下一步。
7. 在選取目標頁面上，選擇 EventBridge 排程器調用的 AWS API 操作：
   a. 選擇AWS Lambda調用。
   b. 在調用區段中，選取函數或選擇建立新的 Lambda 函數。
   c. (選用) 輸入 JSON 承載。如果您未輸入承載，EventBridge 排程器會使用空白事件來調用函數。
8. 選擇下一步。
9. 在設定頁面執行以下動作：
   a. 若要開啟排程，請在排程狀態底下切換到啟用排程。
   b. 若要設定排程的重試政策，請在重試政策和無效字母佇列 (DLQ) 底下執行以下動作：
      - 切換到重試。
      - 針對事件的最長存留期，輸入 EventBridge 排程器保留未處理事件的最大時數和分鐘數。
      - 時間最長可設為 24 小時。
      - 針對重試次數上限，輸入目標傳回錯誤時，EventBridge 排程器重新嘗試執行排程的次數上限。

        最大值為重試 185 次。

      設定好重試政策後，如果排程無法調用其目標，EventBridge 排程器會重新執行排程。一旦設定此功能，您就必須設定排程的最長保留時間和重試次數。
   c. 選擇 EventBridge 排程器儲存未交付事件的位置。

無效字母佇列 (DLQ) 選項	執行此作業...
不儲存	選擇無。
將事件儲存在您建立排程所使用的同一個 AWS 帳戶	a. 選擇在目前的 AWS 帳戶選取 Amazon SQS 佇列作為 DLQ。 b. 選擇 Amazon SQS 佇列的 Amazon Resource Name (ARN)。

無效字母佇列 (DLQ) 選項	執行此作業...
將事件儲存在建立排程所用帳戶以外的 AWS 帳戶	a. 選擇在其他 AWS 帳戶指定 Amazon SQS 佇列作為 DLQ。 b. 輸入 Amazon SQS 佇列的 Amazon Resource Name (ARN)。

d. 若要使用由客戶管理的金鑰加密您的目標輸入，請在加密底下選擇自訂加密設定 (進階)。

   如果選擇此選項，請輸入現有的 KMS 金鑰 ARN，或選擇建立 AWS KMS key，以導覽至 AWS KMS 控制台。如需 EventBridge 排程器如何加密靜態資料的詳細資訊，請參閱《Amazon EventBridge 排程器使用者指南》中的靜態加密。

e. 若要讓 EventBridge 排程器為您建立新的執行角色，請選擇為此排程建立新角色。接著輸入角色名稱。如果您選擇此選項，EventBridge 排程器會將範本化目標所需的必要許可與角色連接。

10. 選擇下一步。
11. 在檢閱和建立排程頁面上，檢閱排程的詳細資訊。在每個區段中選擇編輯，即可返回該步驟並編輯其詳細資訊。
12. 選擇建立排程。

    您可以在排程頁面檢視新建立和現有的排程。在狀態欄底下，確認您的新排程狀態為已啟用。

若要確認 EventBridge 排程器是否順利調用函數，請檢查 Amazon CloudWatch 日誌。

相關資源

如需 EventBridge 排程器的詳細資訊，請參閱下列內容：

- EventBridge 排程器使用者指南
- EventBridge 排程器 API 參考
- EventBridge 排程器定價

搭配使用 AWS Lambda 與 AWS IoT

AWS IoT 提供網際網路連線裝置 (例如感應器) 和 AWS 雲端間的安全通訊。這可以讓您收集、存放和分析來自多個裝置的遙測資料。

您可以為您的裝置建立 AWS IoT 規則，使其和 AWS 服務互動。AWS IoT 規則引擎提供 SQL 類型的語言，可讓您從訊息酬載中選取資料，以及傳送資料至其他服務，例如 Amazon S3、Amazon DynamoDB 和 AWS Lambda。當您希望叫用其他 AWS 服務或第三方服務時，您可以定義規則來叫用 Lambda 函數。

當傳入的 IoT 訊息觸發規則時，AWS IoT 會以非同步方式叫用您的 Lambda 函數，並將資料從 IoT 訊息傳遞至函數。

以下範例示範從溫室感應器讀取濕度。資料列和資料行的值會識別感應器的位置。此範例事件是以 AWS IoT 規則教學中的溫室類型為基礎。

Example AWS IoT 訊息事件

```
{
    "row" : "10",
    "pos" : "23",
    "moisture" : "75"
}
```

針對非同步叫用，Lambda 會將訊息排入佇列，並且在您的函式傳回錯誤時重試。為您的函數設定目的地來保留函數無法處理的事件。

您需要授予許可，AWS IoT 服務才能叫用您的 Lambda 函數。使用 `add-permission` 命令，將許可陳述式加入至函式以資源為基礎的政策。

```
aws lambda add-permission --function-name my-function \
--statement-id iot-events --action "lambda:InvokeFunction" --principal
 iot.amazonaws.com
```

您應該會看到下列輸出：

```
{
    "Statement": "{\"Sid\":\"iot-events\",\"Effect\":\"Allow\",\"Principal\":
{\"Service\":\"iot.amazonaws.com\"},\"Action\":\"lambda:InvokeFunction\",\"Resource\":
\"arn:aws:lambda:us-east-1:123456789012:function:my-function\"}"
```

```
}
```

如需如何搭配使用 Lambda 與 AWS IoT 的詳細資訊，請參閱建立 AWS Lambda 規則。

使用 Lambda 處理來自 Amazon Kinesis Data Streams 的記錄

您可以使用 Lambda 函數來處理 Amazon Kinesis 資料串流中的記錄。您可以將 Lambda 函數對應至 Kinesis Data Streams 共用輸送量取用程式 (標準迭代程式)，或對應至具有增強散發功能的專用輸送量取用程式。對於標準迭代程式，Lambda 會使用 HTTP 協定輪詢 Kinesis 串流中的每個碎片以尋找記錄。事件來源映射會與碎片的其他取用者共用讀取傳輸量。

如需 Kinesis 資料串流的詳細資訊，請參閱從 Amazon Kinesis Data Streams 讀取資料。

> **Note**
>
> Kinesis 會收取每個碎片的費用，以及從串流讀取資料的增強型散發。如需定價的詳細資訊，請參閱 Amazon Kinesis 定價。

輪詢和批次處理串流

Lambda 會從資料串流讀取記錄並透過包含串流記錄的事件同步調用函數。Lambda 會讀取批次中的記錄並調用函數，以處理來自該批次的記錄。每個批次包含來自單一碎片/資料串流的記錄。

您的 Lambda 函數是適用於資料串流的取用者應用程式。其會從每個碎片中一次處理一個批次的記錄。您可以將 Lambda 函數對應至共用輸送量取用程式 (標準迭代程式)，或對應至具有增強散發功能的專用輸送量取用程式。

- 標準迭代器：Lambda 會輪詢 Kinesis 串流中的每個碎片，其記錄的基本速率為每秒一次。有更多記錄可用時，Lambda 會持續處理批次，直到函數掌握串流資訊。事件來源映射會與碎片的其他取用者共用讀取傳輸量。
- 增強散發功能：若要將延遲降至最低，並將讀取傳輸量增至最大，請建立具有增強散發功能的資料串流取用者。增強散發取用者會取得每個碎片的專用連線，其不會影響其他從串流讀取的應用程式。串流取用者會使用 HTTP/2 來減少延遲，方法為透過長期連線將記錄推送至 Lambda，以及透過壓縮請求標頭。您可以使用 Kinesis RegisterStreamConsumer API，建立串流取用者。

```
aws kinesis register-stream-consumer \
--consumer-name con1 \
--stream-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream
```

您應該會看到下列輸出：

```
{
    "Consumer": {
        "ConsumerName": "con1",
        "ConsumerARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream/
consumer/con1:1540591608",
        "ConsumerStatus": "CREATING",
        "ConsumerCreationTimestamp": 1540591608.0
    }
}
```

若要增加函數處理記錄的速度，請將碎片新增至您的資料串流。Lambda 會依序在每個碎片中處理記錄。如果您的函數傳回錯誤，則會停止處理碎片中的其他記錄。碎片越多，要一次處理的批次越多，這會降低錯誤對並行的影響。

如果您的函數無法擴展至可處理並行批次的總數，您可以請求增加配額，或為您的函數保留並行。

Lambda 預設會在記錄可用時立即調用函數。如果 Lambda 從事件來源中讀取的批次只有一筆記錄，Lambda 只會傳送一筆記錄至函數。為避免調用具有少量記錄的函數，您可設定批次間隔，請求事件來源緩衝記錄最長達五分鐘。調用函數之前，Lambda 會繼續從事件來源中讀取記錄，直到收集到完整批次、批次間隔到期或者批次達到 6 MB 的承載限制。如需更多詳細資訊，請參閱 批次處理行為。

> **Warning**
>
> Lambda 事件來源映射至少會處理每個事件一次，而且可能會重複處理記錄。為避免與重複事件相關的潛在問題，強烈建議您讓函數程式碼具有等冪性。如需詳細資訊，請參閱 AWS 知識中心中的如何讓 Lambda 函數具有冪等性。

Lambda 在傳送下一批進行處理前不會等待任何已設定的擴充完成。換句話說，當 Lambda 處理下一批記錄時，您的擴充功能可能會繼續執行。如果您違反任何帳戶的 並行 設定或限制，便可能會產生限流的問題。若要偵測此是否為潛在問題，請監控您的函數，並確認您看到的 並行指標 是否高於事件來源映射的預期值。由於兩次調用之間的時間很短，Lambda 可能會短暫報告比碎片數目更高的並行用量。即使對於沒有延伸項目的 Lambda 函數也可能如此。

設定 ParallelizationFactor 設定，同時使用多個 Lambda 調用來處理 Kinesis 資料串流的一個碎片。您可以透過從 1 (預設) 到 10 的並行化因子指定 Lambda 從碎片輪詢的並行批次數。例如，當 `ParallelizationFactor` 設定為 2 時，您最多可以有 200 個並行 Lambda 調用，來處理 100 個 Kinesis 資料碎片 (不過在實務中，`ConcurrentExecutions` 指標可能有不同值)。當資料量急劇波

動並且 IteratorAge 高時，這有助於縱向擴展處理輸送量。如果增加每個碎片的並行批次數量，則 Lambda 仍會確保在分割區索引鍵層級進行順序處理。

您也可以搭配 Kinesis 彙總使用 ParallelizationFactor。事件來源映射的行為取決於您是否使用增強型散發：

- 沒有使用增強型散發：彙總事件內的所有事件都必須具有相同的分割區索引鍵。分割區索引鍵也必須與彙總事件的分割區索引鍵相符。如果彙總事件內的事件具有不同的分割區索引鍵，則 Lambda 無法保證依分割區索引鍵依序處理事件。
- 使用增強型散發：首先，Lambda 會將彙總的事件解碼為其個別事件。彙總的事件可以具有與其包含的事件不同的分割區索引鍵。不過，未對應至分割區索引鍵的事件會遭到捨棄和遺失。Lambda 不會處理這些事件，也不會將其傳送至設定的失敗目的地。

範例事件

Example

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
        },
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
```

```
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692540925702759324208523137515618",
                "data": "VGhpcyBpcyBvbmx5IGEgdGVzdC4=",
                "approximateArrivalTimestamp": 1545084711.166
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692540925702759324208523137515618",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-
stream"
        }
    ]
}
```

使用 Lambda 處理 Amazon Kinesis Data Streams 記錄

若要使用 Lambda 處理 Amazon Kinesis Data Streams 記錄，請建立 Lambda 事件來源映射。您可以將 Lambda 函式映射至標準迭代器或增強型散發取用者。如需詳細資訊，請參閱輪詢和批次處理串流。

建立 Kinesis 事件來源映射

若要使用資料串流中的記錄調用您的 Lambda 函數，請建立一個事件來源映射。您可以建立多個事件來源映射，來使用多個 Lambda 函數處理相同資料，或使用單一函數處理來自多個資料串流的項目。處理來自多個串流的項目時，每個批次將僅包含來自單一碎片或串流的記錄。

您可以設定事件來源映射，以處理來自不同 AWS 帳戶中的串流的記錄。如需詳細資訊，請參閱 the section called “跨帳戶映射”。

建立事件來源映射之前，您需要授予 Lambda 函數從 Kinesis 資料串流讀取的許可。Lambda 需要以下許可，才能管理與 Kinesis 資料串流相關的資源：

- kinesis:DescribeStream
- kinesis:DescribeStreamSummary
- kinesis:GetRecords
- kinesis:GetShardIterator

- kinesis:ListShards
- kinesis:SubscribeToShard

AWS 受管政策 AWSLambdaKinesisExecutionRole 包含這些許可。如下列程序所述，將此受管政策新增至您的函數。

> **Note**
>
> - 無需 `kinesis:ListStreams` 許可，即可為 Kinesis 建立並管理事件來源映射。但是，如果您在主控台中建立事件來源映射卻並不具備此許可，將無法從下拉式清單中選取 Kinesis 串流，且主控台會顯示錯誤。若要建立事件來源映射，需要手動輸入串流的 Amazon Resource Name (ARN)。
> - Lambda 會在重試失敗的調用時發起 `kinesis:GetRecords` 與 `kinesis:GetShardIterator` API 呼叫。

AWS 管理主控台

若要將 Kinesis 許可新增至函數

1. 開啟 Lambda 主控台的函數頁面，然後選取您的函數。
2. 在組態索引標籤中，選擇許可。
3. 在執行角色窗格的角色名稱下，選擇函數執行角色的連結。此連結會在 IAM 主控台中開啟該角色的頁面。
4. 在許可政策窗格中，選擇新增許可，然後選取附接政策。
5. 在搜尋欄位中輸入 **AWSLambdaKinesisExecutionRole**。
6. 選取政策旁邊的核取方塊，然後選擇新增許可。

AWS CLI

若要將 Kinesis 許可新增至函數

- 執行下列 CLI 命令，將 `AWSLambdaKinesisExecutionRole` 政策新增至函數的執行角色：

```
aws iam attach-role-policy \
--role-name MyFunctionRole \
```

```
--policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaKinesisExecutionRole
```

AWS SAM

若要將 Kinesis 許可新增至函數

- 在函數定義中，新增 `Policies` 屬性，如下列範例所示：

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs24.x
      Policies:
        - AWSLambdaKinesisExecutionRole
```

設定需要的許可後，建立事件來源映射。

AWS 管理主控台

若要建立 Kinesis 事件來源映射

1. 開啟 Lambda 主控台的函數頁面，然後選取您的函數。
2. 在函數概觀窗格中，選擇新增觸發條件。
3. 在觸發條件組態下，針對來源選取 Kinesis。
4. 選取您要為之建立事件來源映射的 Kinesis 串流，以及 (選用) 串流的取用者。
5. (選用) 編輯事件來源映射的批次大小、開始位置和批次時段。
6. 選擇新增。

當從主控台建立事件來源映射時，您的 IAM 角色必須有 kinesis:ListStreams 和 kinesis:ListStreamConsumers 許可。

AWS CLI

若要建立 Kinesis 事件來源映射

- 執行下列 CLI 命令以建立 Kinesis 事件來源映射。根據您的使用案例，選擇您自己的批次大小和開始位置。

```
aws lambda create-event-source-mapping \
--function-name MyFunction \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--starting-position LATEST \
--batch-size 100
```

若要指定批次處理時段，請新增 `--maximum-batching-window-in-seconds` 選項。如需使用此參數和其他參數的詳細資訊，請參閱《AWS CLI 命令參考》中的 create-event-source-mapping。

AWS SAM

若要建立 Kinesis 事件來源映射

- 在函數定義中，新增 `KinesisEvent` 屬性，如下列範例所示：

```
Resources:
  MyFunction:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: ./my-function/
      Handler: index.handler
      Runtime: nodejs24.x
      Policies:
        - AWSLambdaKinesisExecutionRole
      Events:
        KinesisEvent:
          Type: Kinesis
          Properties:
            Stream: !GetAtt MyKinesisStream.Arn
            StartingPosition: LATEST
            BatchSize: 100

  MyKinesisStream:
    Type: AWS::Kinesis::Stream
```

```
    Properties:
      ShardCount: 1
```

若要進一步了解如何在 中建立 Kinesis Data Streams 的事件來源映射 AWS SAM，請參閱《 AWS Serverless Application Model 開發人員指南》中的 Kinesis。

輪詢和串流開始位置

請注意，建立和更新事件來源映射期間的串流輪詢最終會一致。

- 在建立事件來源映射期間，從串流開始輪詢事件可能需要幾分鐘時間。
- 在更新事件來源映射期間，從串流停止並重新開始輪詢事件可能需要幾分鐘時間。

這種行為表示如果您指定 LATEST 當作串流的開始位置，事件來源映射可能會在建立或更新期間遺漏事件。若要確保沒有遺漏任何事件，請將串流開始位置指定為 TRIM_HORIZON 或 AT_TIMESTAMP。

建立跨帳戶事件來源映射

Amazon Kinesis Data Streams 支援資源型政策。因此，您可以在另一個帳戶中 AWS 帳戶 使用 Lambda 函數處理擷取至串流的資料。

若要使用不同 Kinesis 串流為您的 Lambda 函數建立事件來源映射 AWS 帳戶，您必須使用資源型政策來設定串流，以授予 Lambda 函數讀取項目的許可。若要了解如何設定串流以允許跨帳戶存取，請參閱《Amazon Kinesis Streams Developer 指南》中的使用跨帳戶 AWS Lambda 函數共用存取權。

使用為您的 Lambda 函數提供所需許可的資源型政策設定串流後，使用上一節所述的任何方法建立事件來源映射。

如果您選擇使用 Lambda 主控台來建立事件來源映射，則請將串流的 ARN 直接貼入輸入欄位。如果您想要為串流指定取用者，則貼上取用者的 ARN 即會自動填入串流欄位。

使用 Kinesis Data Streams 和 Lambda 設定部分批次回應

取用和處理事件來源的串流資料時，依預設，只有在批次成功完成時，Lambda 檢查點才會到批次的最高序號。Lambda 會將所有其他結果視為完全失敗，並重試處理批次，直至達到重試限制。若要在處理串流的批次時允許部分成功，請開啟 ReportBatchItemFailures。允許部分成功有助於減少記錄的重試次數，但其不會完全消除在成功記錄中重試的可能性。

若要開啟 ReportBatchItemFailures，請在 FunctionResponseTypes 清單中包含枚舉值 **ReportBatchItemFailures**。此清單指示已為您的函數啟用哪些回應類型。您可以在建立或更新事件來源映射時設定此清單。

> **Note**
>
> 即使函式程式碼傳回了部分批次失敗回應，除非針對事件來源映射明確啟用 ReportBatchItemFailures 功能，否則 Lambda 將不會處理這些回應。

報告語法

設定批次項目失敗的報告時，會傳回 StreamsEventResponse 類別，其中包含批次項目失敗的清單。您可以使用 StreamsEventResponse 物件，來傳回批次中第一個失敗記錄的序號。您還可以使用正確的回應語法，建立自己的自訂類別。下列 JSON 結構顯示所需的回應語法：

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "<SequenceNumber>"
        }
    ]
}
```

> **Note**
>
> 如果 batchItemFailures 陣列包含多個項目，則 Lambda 會使用具有最低序列號的記錄作為檢查點。然後，Lambda 會重試從該檢查點開始的所有記錄。

成功與失敗條件

如果您傳回下列任一項目，Lambda 會將批次視為完全成功：

- 空白 batchItemFailure 清單
- Null batchItemFailure 清單
- 空白 EventResponse
- Null EventResponse

如果您傳回下列任一項目，Lambda 會將批次視為完全失敗：

- 空白字串 `itemIdentifier`
- Null `itemIdentifier`
- 具有錯誤金鑰名稱的 `itemIdentifier`

Lambda 會根據您的重試政策來重試失敗。

將批次平分

如果您的調用失敗且 `BisectBatchOnFunctionError` 已開啟，則無論您的 `ReportBatchItemFailures` 設定如何，批次都會被平分。

收到部分批次成功回應且 `BisectBatchOnFunctionError` 和 `ReportBatchItemFailures` 均開啟時，批次會依傳回的序號進行平分，並且 Lambda 僅會重試剩餘的記錄。

為了簡化部分批次回應邏輯的實作，請考慮使用 Powertools 中的 Batch Processor 公用程式 AWS Lambda，其會自動為您處理這些複雜性。

以下範例函數程式碼會傳回批次中失敗訊息 ID 的清單：

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;
```

```
// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
```

```
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

透過使用 Java 的 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Javascript 搭配 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
```

```
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

使用 TypeScript 搭配 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
```

```
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php
```

```
# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();

        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
```

```
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

透過使用 Python 的 Lambda 報告 Kinesis 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來報告 Kinesis 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end
```

```
def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
```

```
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());

    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
```

```
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

使用 Powertools 進行 AWS Lambda 批次處理器

Powertools for 的批次處理器公用程式 AWS Lambda 會自動處理部分批次回應邏輯，降低實作批次失敗報告的複雜性。以下是使用批次處理器的範例：

Python

> **Note**
>
> 如需完整的範例和設定說明，請參閱 batch processor documentation。

使用 AWS Lambda 批次處理器處理 Kinesis Data Streams 串流記錄。

```
import json
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities.batch import BatchProcessor, EventType,
 process_partial_response
from aws_lambda_powertools.utilities.data_classes import KinesisEvent
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.KinesisDataStreams)
logger = Logger()

def record_handler(record):
    logger.info(record)
    # Your business logic here
    # Raise an exception to mark this record as failed

def lambda_handler(event, context: LambdaContext):
```

```
    return process_partial_response(
        event=event,
        record_handler=record_handler,
        processor=processor,
        context=context
    )
```

TypeScript

> **Note**
>
> 如需完整的範例和設定說明，請參閱 batch processor documentation。

使用 AWS Lambda 批次處理器處理 Kinesis Data Streams 串流記錄。

```
import { BatchProcessor, EventType, processPartialResponse } from '@aws-lambda-
powertools/batch';
import { Logger } from '@aws-lambda-powertools/logger';
import type { KinesisEvent, Context } from 'aws-lambda';

const processor = new BatchProcessor(EventType.KinesisDataStreams);
const logger = new Logger();

const recordHandler = async (record: any): Promise<void> => {
    logger.info('Processing record', { record });
    // Your business logic here
    // Throw an error to mark this record as failed
};

export const handler = async (event: KinesisEvent, context: Context) => {
    return processPartialResponse(event, recordHandler, processor, {
        context,
    });
};
```

Java

> **Note**
>
> 如需完整的範例和設定說明，請參閱 batch processor documentation。

使用 AWS Lambda 批次處理器處理 Kinesis Data Streams 串流記錄。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import software.amazon.lambda.powertools.batch.BatchMessageHandlerBuilder;
import software.amazon.lambda.powertools.batch.handler.BatchMessageHandler;

public class KinesisStreamBatchHandler implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    private final BatchMessageHandler<KinesisEvent, StreamsEventResponse> handler;

    public KinesisStreamBatchHandler() {
        handler = new BatchMessageHandlerBuilder()
                .withKinesisBatchHandler()
                .buildWithRawMessageHandler(this::processMessage);
    }

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent kinesisEvent, Context
 context) {
        return handler.processBatch(kinesisEvent, context);
    }

    private void processMessage(KinesisEvent.KinesisEventRecord kinesisEventRecord,
 Context context) {
        // Process the stream record
    }
}
```

.NET

> **Note**
>
> 如需完整的範例和設定說明，請參閱 batch processor documentation。

使用 AWS Lambda 批次處理器處理 Kinesis Data Streams 串流記錄。

```
using System;
using System.Threading;
```

```
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using Amazon.Lambda.Serialization.SystemTextJson;
using AWS.Lambda.Powertools.BatchProcessing;

[assembly: LambdaSerializer(typeof(DefaultLambdaJsonSerializer))]

namespace HelloWorld;

public class OrderEvent
{
    public string? OrderId { get; set; }
    public string? CustomerId { get; set; }
    public decimal Amount { get; set; }
    public DateTime OrderDate { get; set; }
}

internal class TypedKinesisRecordHandler : ITypedRecordHandler<OrderEvent>
{
    public async Task<RecordHandlerResult> HandleAsync(OrderEvent orderEvent,
 CancellationToken cancellationToken)
    {
        if (string.IsNullOrEmpty(orderEvent.OrderId))
        {
            throw new ArgumentException("Order ID is required");
        }

        return await Task.FromResult(RecordHandlerResult.None);
    }
}

public class Function
{
    [BatchProcessor(TypedRecordHandler = typeof(TypedKinesisRecordHandler))]
    public BatchItemFailuresResponse HandlerUsingTypedAttribute(KinesisEvent _)
    {
        return TypedKinesisStreamBatchProcessor.Result.BatchItemFailuresResponse;
    }
}
```

在 Lambda 中保留 Kinesis Data Streams 事件來源的捨棄批次記錄

Kinesis 事件來源映射的錯誤處理取決於錯誤是在調用函數之前還是在調用函數期間發生：

- 調用前：如果 Lambda 事件來源映射因為限流或其他問題而無法調用函數，它會重試，直到記錄過期或超過事件來源映射上設定的最長存留期 (MaximumRecordAgeInSeconds)。
- 在調用期間：如果調用函數但傳回錯誤，Lambda 會重試，直到記錄過期、超過最長存留期 (MaximumRecordAgeInSeconds)，或達到設定的重試配額 (MaximumRetryAttempts)。對於函數錯誤，您也可以設定 BisectBatchOnFunctionError，將失敗的批次分割為兩個較小的批次，隔離錯誤的記錄並避免逾時。分割批次不會消耗重試配額。

如果錯誤處理措施失敗，Lambda 會捨棄相應記錄，並繼續處理串流中的批次。使用預設設定時，這表示不良的記錄可能會封鎖受影響碎片上的處理長達一週。若要避免此情況，在設定函數的事件來源映射時，請使用合理的重試次數和符合您使用案例的記錄最大保留期。

設定失敗調用的目的地

若要保留失敗的事件來源映射調用記錄，請將目標地新增到函數的事件來源映射中。傳送至該目的地的每筆記錄都是包含失敗調用之中繼資料的 JSON 文件。對於 Amazon S3 目的地，Lambda 還會將整個調用記錄與中繼資料一起傳送。您可以設定任何 Amazon SNS 主題、Amazon SQS 佇列、Amazon S3 儲存貯體或 Kafka 做為目的地。

透過 Amazon S3 目的地，您可以使用 Amazon S3 事件通知功能，在物件上傳至您的目的地 S3 儲存貯體時接收通知。您也可以設定 S3 事件通知來調用另一個 Lambda 函數，以對失敗的批次執行自動處理。

執行角色必須具有目的地的許可：

- 對於 SQS 目的地：sqs：SendMessage
- 對於 SNS 目的地：sns：Publish
- 對於 S3 目的地：s3：PutObject 和 s3：ListBucket
- 對於 Kafka 目的地：kafka-cluster：WriteData

您可以將 Kafka 主題設定為 Kafka 事件來源映射的失敗時目的地。當 Lambda 在耗盡重試嘗試後或記錄超過最長存留期後無法處理記錄時，Lambda 會將失敗的記錄傳送至指定的 Kafka 主題，以供稍後處理。請參閱the section called “Kafka 失敗時目的地”。

如果您已使用自己的 KMS 金鑰為 S3 目的地啟用加密，則函數的執行角色也必須具有呼叫 kms:GenerateDataKey 的許可。如果 KMS 金鑰和 S3 儲存貯體目的地與 Lambda 函數和執行角色位於不同的帳戶中，請將 KMS 金鑰設定為信任執行角色以允許 kms:GenerateDataKey。

若要使用主控台設定失敗時的目的地，請依照下列步驟執行：

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在函數概觀下，選擇新增目的地。
4. 針對來源，請選擇事件來源映射調用。
5. 對於事件來源映射，請選擇針對此函數設定的事件來源。
6. 對於條件，選取失敗時。對於事件來源映射調用，這是唯一可接受的條件。
7. 對於目標類型，請選擇 Lambda 將調用記錄傳送至的目標類型。
8. 對於目的地，請選擇一個資源。
9. 選擇儲存。

您也可以使用 AWS Command Line Interface () 設定失敗時的目的地AWS CLI。例如，下列 create-event-source-mapping 命令會將具有 SQS 失敗時的目的地的事件來源映射新增至 `MyFunction`：

```
aws lambda create-event-source-mapping \
--function-name "MyFunction" \
--event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sqs:us-
east-1:123456789012:dest-queue"}}'
```

下列 update-event-source-mapping 命令會更新事件來源映射，以在嘗試兩次重試後，或在記錄超過一小時時，將失敗的調用記錄傳送至 SNS 目的地。

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--maximum-retry-attempts 2 \
--maximum-record-age-in-seconds 3600 \
--destination-config '{"OnFailure": {"Destination": "arn:aws:sns:us-
east-1:123456789012:dest-topic"}}'
```

系統會以非同步的方式套用更新的設定，在處理完成之前不會反映在輸出中。使用 get-event-source-mapping 命令檢視目前的狀態。

若要移除目的地，請提供空白字串作為 destination-config 參數的引數：

```
aws lambda update-event-source-mapping \
--uuid f89f8514-cdd9-4602-9e1f-01a5b77d449b \
--destination-config '{"OnFailure": {"Destination": ""}}'
```

Amazon S3 目的地的安全最佳實務

刪除已設定為目的地的 S3 儲存貯體而不從函數的組態中移除目的地，可能會產生安全風險。如果其他使用者知道目的地儲存貯體的名稱，他們可以在其 AWS 帳戶中重新建立儲存貯體。失敗調用的記錄會被傳送到其儲存貯體，可能公開來自您函數的資料。

> **Warning**
>
> 為了確保無法將來自函數的調用記錄傳送到另一個 中的 S3 儲存貯體 AWS 帳戶，請將條件新增至函數的執行角色，以限制您帳戶中儲存貯體的s3:PutObject許可。

下列範例顯示的 IAM 政策，將函數的 s3:PutObject 許可限制為帳戶中的儲存貯體。此政策也為 Lambda 提供了使用 S3 儲存貯體做為目的地所需的 s3:ListBucket 許可。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "S3BucketResourceAccountWrite",
            "Effect": "Allow",
            "Action": [
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::*/*",
                "arn:aws:s3:::*"
            ],
            "Condition": {
                "StringEquals": {
                    "s3:ResourceAccount": "111122223333"
                }
            }
        }
```

```
    ]
}
```

若要使用 AWS 管理主控台 或 將許可政策新增至函數的執行角色 AWS CLI，請參閱下列程序中的指示：

Console

若要將許可政策新增至函數的執行角色 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選取您要修改其執行角色的 Lambda 函數。
3. 在組態索引標籤中，選擇許可。
4. 在執行角色索引標籤中，選取函數的角色名稱，以開啟角色的 IAM 主控台頁面。
5. 透過下列步驟將許可政策新增至角色：
   a. 在許可政策窗格中，選擇新增許可 ，然後選取建立內嵌政策。
   b. 在政策編輯器中，選取 JSON。
   c. 將您要新增的政策貼入編輯器 (取代現有的 JSON)，然後選擇下一步。
   d. 在政策詳細資訊下，輸入政策名稱。
   e. 選擇建立政策。

AWS CLI

若要將許可政策新增至函數的執行角色 (CLI)

1. 建立具有所需許可的 JSON 政策文件，並將其儲存在本機目錄中。
2. 使用 IAM `put-role-policy` CLI 命令，將許可新增至函數的執行角色。從您儲存 JSON 政策文件的目錄執行下列命令，並將角色名稱、政策名稱和政策文件取代為您自己的值。

```
aws iam put-role-policy \
--role-name my_lambda_role \
--policy-name LambdaS3DestinationPolicy \
--policy-document file://my_policy.json
```

Amazon SNS 和 Amazon SQS 調用記錄範例

下列範例顯示 Lambda 針對失敗的 Kinesis 事件來源調用而傳送至 SQS 佇列或 SNS 主題。由於 Lambda 只會傳送這些目標類型的中繼資料，因此請使用 `streamArn`、`shardId`、`startSequenceNumber` 和 `endSequenceNumber` 欄位來取得完整的原始記錄。`KinesisBatchInfo` 屬性中顯示的所有欄位一律都會存在。

```
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KinesisBatchInfo": {
        "shardId": "shardId-000000000001",
        "startSequenceNumber":
 "49601189658422359378836298521827638475320189012309704722",
        "endSequenceNumber":
 "49601189658422359378836298522902373528957594348623495186",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:38:04.835Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:38:05.580Z",
        "batchSize": 500,
        "streamArn": "arn:aws:kinesis:us-east-2:123456789012:stream/mystream"
    }
}
```

您可以使用此資訊來從串流擷取受影響的記錄，以進行疑難排解。實際的記錄不包含在內，因此您必須處理此記錄，並在因過期而遺失之前從資料串流中擷取它們。

Amazon S3 調用記錄範例

下列範例顯示 Lambda 針對失敗的 Kinesis 事件來源調用而傳送至 Amazon S3 儲存貯體的內容。除了上一個 SQS 和 SNS 目的地範例中的所有欄位之外，此 `payload` 欄位還包含原始調用記錄做為逸出 JSON 字串。

```
{
    "requestContext": {
        "requestId": "c9b8fa9f-5a7f-xmpl-af9c-0c604cde93a5",
        "functionArn": "arn:aws:lambda:us-east-2:123456789012:function:myfunction",
        "condition": "RetryAttemptsExhausted",
        "approximateInvokeCount": 1
    },
    "responseContext": {
        "statusCode": 200,
        "executedVersion": "$LATEST",
        "functionError": "Unhandled"
    },
    "version": "1.0",
    "timestamp": "2019-11-14T00:38:06.021Z",
    "KinesisBatchInfo": {
        "shardId": "shardId-000000000001",
        "startSequenceNumber":
 "49601189658422359378836298521827638475320189012309704722",
        "endSequenceNumber":
 "49601189658422359378836298522902373528957594348623495186",
        "approximateArrivalOfFirstRecord": "2019-11-14T00:38:04.835Z",
        "approximateArrivalOfLastRecord": "2019-11-14T00:38:05.580Z",
        "batchSize": 500,
        "streamArn": "arn:aws:kinesis:us-east-2:123456789012:stream/mystream"
    },
    "payload": "<Whole Event>" // Only available in S3
}
```

包含調用記錄的 S3 物件使用以下命名慣例：

```
aws/lambda/<ESM-UUID>/<shardID>/YYYY/MM/DD/YYYY-MM-DDTHH.MM.SS-<Random UUID>
```

在 Lambda 中實作有狀態的 Kinesis 資料串流處理

Lambda 函數可執行持續串流處理應用程式。串流表示持續在應用程式中流動的無限制資料。若要分析此持續更新輸入中的資訊，您可以使用定義的時段來限制包含的記錄。

輪轉時段是定期開啟和關閉的不同時段。依預設，Lambda 調用是無狀態的，您無法在沒有外部資料庫的情況下，將其用於處理多個持續調用的資料。然而，使用輪轉時段，您可以在不同的調用間維護狀態。此狀態包含之前為目前時段處理之訊息的彙總結果。狀態可以是每個分區最多 1 MB。如果超過該大小，則 Lambda 會提前終止時段。

串流中的每個記錄都屬於一個特定時段。Lambda 至少會處理一次每筆記錄，但不保證每筆記錄只會處理一次。在極少數情況下，例如錯誤處理，某些記錄可能會處理多次。第一次時一律會依序處理記錄。如果多次處理記錄，則可能不會按順序處理。

彙總與處理

調用您的使用者管理函數進行彙總，以及處理該彙總的最終結果。Lambda 會彙總時段中接收的所有記錄。您可以在多個批次中接收這些記錄，各自作為單獨的調用。每次調用會收到一個狀態。因此，當使用輪轉時段時，您的 Lambda 函數回應必須包含 `state` 屬性。如果回應不包含 `state` 屬性，Lambda 會將此視為失敗的調用。為了滿足此條件，您的函數可以返回一個 `TimeWindowEventResponse` 物件，它具有下列 JSON 形狀：

Example **`TimeWindowEventResponse`** 值

```
{
    "state": {
        "1": 282,
        "2": 715
    },
    "batchItemFailures": []
}
```

> **Note**
>
> 對於 Java 函數，我們建議使用 `Map<String, String>` 來表示狀態。

在時段結束時，標記 `isFinalInvokeForWindow` 會設定為 `true` 以指示這是最終狀態，並且可隨時進行處理。處理完成後，時段結束並完成最終調用，然後丟棄該狀態。

在時段結束時，Lambda 會針對彙總結果上的動作使用最終處理。您的最終處理將同步調用。成功調用後，您的函數檢查點序號和串流處理將會繼續。如果調用失敗，則您的 Lambda 函數會暫停進一步處理，直至成功調用。

Example KinesisTimeWindowEvent

```
{
    "Records": [
        {
            "kinesis": {
```

```
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1607497475.000
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-kinesis-role",
            "awsRegion": "us-east-1",
            "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-
stream"
        }
    ],
    "window": {
        "start": "2020-12-09T07:04:00Z",
        "end": "2020-12-09T07:06:00Z"
    },
    "state": {
        "1": 282,
        "2": 715
    },
    "shardId": "shardId-000000000006",
    "eventSourceARN": "arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream",
    "isFinalInvokeForWindow": false,
    "isWindowTerminatedEarly": false
}
```

Configuration

您可以在建立或更新事件來源對映時設定輪轉時段。若要設定輪轉時段，請以秒為單位指定時段 (TumblingWindowInSeconds)。下列 example AWS Command Line Interface (AWS CLI) 命令會建立輪轉時段為 120 秒的串流事件來源映射。針對彙總與處理定義的 Lambda 函數命名為 `tumbling-window-example-function`。

```
aws lambda create-event-source-mapping \
--event-source-arn arn:aws:kinesis:us-east-1:123456789012:stream/lambda-stream \
--function-name tumbling-window-example-function \
--starting-position TRIM_HORIZON \
```

```
--tumbling-window-in-seconds 120
```

Lambda 根據記錄插入串流的時間，確定輪轉時段邊界。所有記錄都有 Lambda 在邊界確定中使用的近似時間戳記。

輪轉時段彙總不支援重新分區。分區結束後，Lambda 會考慮關閉目前時段，並且任何子分區會以全新的狀態開始自己的時段。當沒有新記錄新增至目前的時段時，Lambda 會等待最多 2 分鐘，然後假設該時段已結束。這有助於確保函數讀取目前時段中的所有記錄，即使記錄是間歇性新增的。

輪轉時段完全支援現有的重試政策 `maxRetryAttempts` 和 `maxRecordAge`。

Example Handler.py - 彙總與處理

下列 Python 函數示範了如何彙總，然後處理您的最終狀態：

```
def lambda_handler(event, context):
    print('Incoming event: ', event)
    print('Incoming state: ', event['state'])

#Check if this is the end of the window to either aggregate or process.
    if event['isFinalInvokeForWindow']:
        # logic to handle final state of the window
        print('Destination invoke')
    else:
        print('Aggregate invoke')

#Check for early terminations
    if event['isWindowTerminatedEarly']:
        print('Window terminated early')

    #Aggregation logic
    state = event['state']
    for record in event['Records']:
        state[record['kinesis']['partitionKey']] = state.get(record['kinesis']
['partitionKey'], 0) + 1

    print('Returning state: ', state)
    return {'state': state}
```

適用於 Amazon Kinesis Data Streams 事件來源映射的 Lambda 參數

所有 Lambda 事件來源映射都會共用相同的 CreateEventSourceMapping 和 UpdateEventSourceMapping API 操作。但是，只有一些參數適用於 Kinesis。

參數	必要	預設	備註
BatchSize	否	100	上限：10,000
BisectBatchOnFunctionError	N	false	無
DestinationConfig	N	N/A	捨棄記錄的 Amazon SQS 佇列或 Amazon SNS 主題目標。如需更多詳細資訊，請參閱 設定失敗調用的目的地。
已啟用	N	true	無
EventSourceArn	是	N/A	資料串流或串流消費者的 ARN
FunctionName	是	N/A	無
FunctionResponseTypes	N	N/A	若要讓函數報告批次中的特定失敗，請將值 `ReportBatchItemFailures` 包含在 `FunctionResponseTypes` 中。如需更多詳細資訊，請參閱 使用 Kinesis Data Streams 和 Lambda 設定部分批次回應。
MaximumBatchingWindowInSeconds	N	0	無
MaximumRecordAgeInSeconds	N	-1	-1 表示無限：Lambda 不會捨棄記錄 (Kinesis

參數	必要	預設	備註
			Data Streams 資料保留設定仍然適用) 下限：-1 上限：604,800
MaximumRetryAttempts	N	-1	-1 代表無限：系統會重試失敗的記錄，直到記錄過期為止 下限：-1 上限：10,000
ParallelizationFactor	N	1	上限：10
StartingPosition	是	N/A	AT_TIMESTAMP、TRIM_HORIZON 或 LATEST
StartingPositionTimestamp	N	N/A	僅當 StartingPosition 設定為 AT_TIMESTAMP 時才有效。這是開始讀取的時間 (以 Unix 時間秒為單位)
TumblingWindowInSeconds	N	N/A	下限：0 上限：900

搭配 Kinesis 事件來源使用事件篩選

您可以使用事件篩選來控制 Lambda 將哪些記錄從串流或佇列中傳送至函數。如需事件篩選運作方式的一般資訊，請參閱the section called “事件篩選”。

本節重點介紹 Kinesis 事件來源的事件篩選。

> **Note**
>
> Kinesis 事件來源映射僅支援依據 `data` 鍵進行篩選。

主題

- Kinesis 事件篩選基本概念
- 篩選 Kinesis 彙總記錄

Kinesis 事件篩選基本概念

假設生產者將 JSON 格式的資料放入您的 Kinesis 資料串流中。範例記錄如下所示，JSON 資料在 `data` 欄位中會轉換為 Base64 編碼字串。

```
{
    "kinesis": {
        "kinesisSchemaVersion": "1.0",
        "partitionKey": "1",
        "sequenceNumber": "49590338271490256608559692538361571095921575989136588898",
        "data":
 "eyJSZWNvcmROdW1iZXIiOiAiMDAwMSIsICJUaW1lU3RhbXAiOiAieXl5eS1tbS1kZFRoaDptbTpzcyIsICJSZXF1ZXN0Q
        "approximateArrivalTimestamp": 1545084650.987
        },
    "eventSource": "aws:kinesis",
    "eventVersion": "1.0",
    "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
    "eventName": "aws:kinesis:record",
    "invokeIdentityArn": "arn:aws:iam::123456789012:role/lambda-role",
    "awsRegion": "us-east-2",
    "eventSourceARN": "arn:aws:kinesis:us-east-2:123456789012:stream/lambda-stream"
}
```

只要生產者放入串流中的資料是有效的 JSON，您就可以使用事件篩選透過 `data` 索引鍵來篩選記錄。假設生產者以下列 JSON 格式將記錄放入 Kinesis 串流中。

```
{
    "record": 12345,
    "order": {
        "type": "buy",
```

```
        "stock": "ANYCO",
        "quantity": 1000
        }
}
```

若僅篩選訂單類型為「購買」的記錄，FilterCriteria 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"order\" : { \"type\" : [ \"buy\" ] } } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 Pattern 在純文字 JSON 中擴展的值。

```
{
    "data": {
        "order": {
            "type": [ "buy" ]
            }
      }
}
```

您可以使用主控台、AWS CLI 或 AWS SAM 範本新增篩選條件。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "data" : { "order" : { "type" : [ "buy" ] } } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 來建立具有這些篩選條件標準的新事件來源映射，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
```

```
    --event-source-arn arn:aws:kinesis:us-east-2:123456789012:stream/my-stream \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"order\" : { \"type
\" : [ \"buy\" ] } } }"}]}'
```

AWS SAM

若要使用 AWS SAM 新增此篩選條件，請將下列程式碼片段新增到事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "order" : { "type" : [ "buy" ] } } }'
```

若要正確篩選來自 Kinesis 來源的事件，資料欄位和資料欄位的篩選條件標準都必須是有效的 JSON 格式。如果其中一個欄位不是有效的 JSON 格式，則 Lambda 會捨棄訊息或擲回例外狀況。下表摘要說明特定行為：

傳入資料格式	資料屬性的篩選條件模式格式	產生的動作
有效的 JSON	有效的 JSON	根據您的篩選條件標準之 Lambda 篩選條件。
有效的 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	非 JSON	Lambda 會在事件來源映射建立或更新時擲回例外狀況。資料屬性的篩選條件模式必須是有效的 JSON 格式。
非 JSON	有效的 JSON	Lambda 捨棄記錄。

傳入資料格式	資料屬性的篩選條件模式格式	產生的動作
非 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
非 JSON	非 JSON	Lambda 會在事件來源映射建立或更新時擲回例外狀況。資料屬性的篩選條件模式必須是有效的 JSON 格式。

篩選 Kinesis 彙總記錄

使用 Kinesis，可以將多個記錄彙總到單一 Kinesis 資料串流記錄中，以提高資料輸送量。Lambda 只會在您使用 Kinesis 增強型展開傳送時，將篩選條件標準套用至彙總記錄。不支援使用標準 Kinesis 篩選彙總記錄。使用增強型展開傳送時，您可以設定 Kinesis 專用輸送量取用程式，以充當 Lambda 函數的觸發條件。Lambda 接著會篩選彙總記錄，並僅傳遞符合篩選條件標準的記錄。

若要進一步了解 Kinesis 記錄彙總，請參閱「Kinesis Producer Library (KPL) 主要概念」頁面上的彙總部分。若要進一步了解如何搭配使用 Lambda 與 Kinesis 增強型展開傳送，請參閱 AWS 運算部落格上的使用 Amazon Kinesis Data Streams 增強型展開傳送和 AWS Lambda 來提高即時串流處理效能。

教學課程：搭配 Kinesis Data Streams 使用 Lambda

在本教學課程中，您會建立 Lambda 函數，以取用 Amazon Kinesis 資料串流中的事件。

1. 自訂應用程式將記錄寫入串流。
2. AWS Lambda 會輪詢串流，並在偵測到串流中的新記錄時，叫用您的 Lambda 函數。
3. AWS Lambda 會透過擔任您在建立 Lambda 函數時指定的執行角色來執行 Lambda 函數。

先決條件

安裝 AWS Command Line Interface

如果您尚未安裝 AWS Command Line Interface，請依照安裝或更新最新版本的 AWS CLI中的步驟進行安裝。

本教學課程需使用命令列終端機或 Shell 來執行命令。在 Linux 和 macOS 中，使用您偏好的 Shell 和套件管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。

建立執行角色

建立 執行角色，授予函數存取 AWS 資源的許可。

若要建立執行角色

1. 在 IAM 主控台中開啟角色頁面。
2. 選擇建立角色。
3. 建立具備下列屬性的角色。
   - 信任實體 – AWS Lambda。
   - Permissions (許可) - AWSLambdaKinesisExecutionRole。
   - 角色名稱 – **`lambda-kinesis-role`**。

AWSLambdaKinesisExecutionRole 政策具備函數自 Kinesis 讀取項目以及寫入日誌到 CloudWatch Logs 時所需的許可。

建立函數

建立可處理 Kinesis 訊息的 Lambda 函數。函數程式碼會將 Kinesis 記錄的事件 ID 和事件資料記錄到 CloudWatch Logs 中。

本教學課程使用 Node.js 24 執行時間，但我們也提供其他執行時間語言的範例程式碼。您可以在下列方塊中選取索引標籤，查看您感興趣的執行期程式碼。此步驟要使用的 JavaScript 程式碼，位於 JavaScript 索引標籤中顯示的第一個範例。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
```

```
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main
```

```
import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

使用 TypeScript 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
```

```
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
```

```
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 Kinesis 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
```

```
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來使用 Kinesis 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
```

```
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

建立函數

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir kinesis-tutorial
cd kinesis-tutorial
```

2. 將範例 JavaScript 程式碼複製到名為 `index.js` 的新檔案。
3. 建立部署套件。

```
zip function.zip index.js
```

4. 使用 `create-function` 命令建立一個 Lambda 函數。

```
aws lambda create-function --function-name ProcessKinesisRecords \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
--role arn:aws:iam::111122223333:role/lambda-kinesis-role
```

測試 Lambda 函數

使用 invoke AWS Lambda CLI 命令和範例 Kinesis 事件手動叫用 Lambda 函數。

測試 Lambda 函數

1. 將以下 JSON 複製到一個檔案中並儲存為 input.txt。

```
{
    "Records": [
        {
            "kinesis": {
                "kinesisSchemaVersion": "1.0",
                "partitionKey": "1",
                "sequenceNumber":
 "49590338271490256608559692538361571095921575989136588898",
                "data": "SGVsbG8sIHRoaXMgaXMgYSB0ZXN0Lg==",
                "approximateArrivalTimestamp": 1545084650.987
            },
            "eventSource": "aws:kinesis",
            "eventVersion": "1.0",
            "eventID":
 "shardId-000000000006:49590338271490256608559692538361571095921575989136588898",
            "eventName": "aws:kinesis:record",
            "invokeIdentityArn": "arn:aws:iam::111122223333:role/lambda-kinesis-
role",
            "awsRegion": "us-east-2",
            "eventSourceARN": "arn:aws:kinesis:us-east-2:111122223333:stream/
lambda-stream"
        }
    ]
}
```

2. 使用 invoke 命令來傳送事件到函數。

```
aws lambda invoke --function-name ProcessKinesisRecords \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資

訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

回應已儲存至 out.txt。

建立 Kinesis 串流

使用 create-stream 命令來建立串流。

```
aws kinesis create-stream --stream-name lambda-stream --shard-count 1
```

執行下列 describe-stream 命令以取得串流 ARN。

```
aws kinesis describe-stream --stream-name lambda-stream
```

您應該會看到下列輸出：

```
{
    "StreamDescription": {
        "Shards": [
            {
                "ShardId": "shardId-000000000000",
                "HashKeyRange": {
                    "StartingHashKey": "0",
                    "EndingHashKey": "340282366920746074317682119384634633455"
                },
                "SequenceNumberRange": {
                    "StartingSequenceNumber":
 "49591073947768692513481539594623130411957558361251844610"
                }
            }
        ],
        "StreamARN": "arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream",
        "StreamName": "lambda-stream",
        "StreamStatus": "ACTIVE",
        "RetentionPeriodHours": 24,
        "EnhancedMonitoring": [
            {
                "ShardLevelMetrics": []
            }
        ],
        "EncryptionType": "NONE",
```

```
        "KeyId": null,
        "StreamCreationTimestamp": 1544828156.0
    }
}
```

在下一個步驟中，您會使用串流 ARN 與您的 Lambda 函數的串流建立關聯。

在 中新增事件來源 AWS Lambda

執行下列 AWS CLI `add-event-source` 命令。

```
aws lambda create-event-source-mapping --function-name ProcessKinesisRecords \
--event-source  arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream \
--batch-size 100 --starting-position LATEST
```

記下映射 ID 以供後續使用。您可以執行 `list-event-source-mappings` 命令來取得事件來源映射的清單。

```
aws lambda list-event-source-mappings --function-name ProcessKinesisRecords \
--event-source arn:aws:kinesis:us-east-1:111122223333:stream/lambda-stream
```

在回應中，您可以確認狀態值為 `enabled`。您可以停用事件來源映射來暫時暫停輪詢，而不會遺失任何記錄。

測試設定

若要測試事件來源映射，請新增事件記錄到您的 Kinesis 串流。`--data` 值是一個字串，CLI 會先將該字串編碼為 base64，再將它傳送到 Kinesis。您可以執行相同命令一次以上，以新增多筆記錄到串流中。

```
aws kinesis put-record --stream-name lambda-stream --partition-key 1 \
--data "Hello, this is a test."
```

Lambda 使用執行角色自串流讀取記錄。接著，它調用您的 Lambda 函數，以記錄批次來傳遞。函數會解碼來自每個記錄的資料並加以記錄，將輸出傳送至 CloudWatch Logs。在 CloudWatch 主控台中檢視日誌。

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。透過刪除您不再使用 AWS 的資源，您可以避免不必要的 費用 AWS 帳戶。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇刪除。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **`confirm`**，然後選擇 刪除 。

刪除 Kinesis 串流

1. 登入 AWS 管理主控台 並開啟位於 https：//https://console.aws.amazon.com/kinesis 的 Kinesis 主控台。
2. 選取您建立的串流。
3. 選擇 動作、刪除。
4. 在文字輸入欄位中輸入 **`delete`**。
5. 選擇 刪除 。

搭配 Kubernetes 使用 Lambda

您可以使用 Kubernetes 專用 AWS 控制器 (ACK) 或 Crossplane，透過 Kubernetes API 來部署和管理 Lambda 函數。

Kubernetes 專用 AWS 控制器 (ACK)

您可以使用 ACK，從 Kubernetes API 部署和管理 AWS 資源。透過 ACK，AWS 提供諸如以下 AWS 服務專用的開放原始碼自訂控制器：Lambda、Amazon Elastic Container Registry (Amazon ECR)、Amazon Simple Storage Service (Amazon S3) 和 Amazon SageMaker AI。每個支援的 AWS 服務都有專屬自訂控制器。在您的 Kubernetes 叢集中，為每個要使用的 AWS 服務安裝控制器。接著建立自訂資源定義 (CRD)以定義 AWS 資源。

建議您使用 Helm 3.8 或更新版本，安裝 ACK 控制器。每個 ACK 控制器都有專屬 Helm Chart，這會安裝控制器、CRD 和 Kubernetes RBAC 規則。如需詳細資訊，請參閱 ACK 文件中的 Install an ACK Controller。

建立 ACK 自訂資源之後，您可以像使用任何其他內建 Kubernetes 物件一樣加以使用。例如，您可以使用偏好的 Kubernetes 工具鏈 (包括 kubectl)，部署和管理 Lambda 函數。

以下是一些透過 ACK 佈建 Lambda 函數的範例使用案例：

- 您的組織使用角色型存取控制 (RBAC) 和服務帳戶的 IAM 角色，建立許可界限。您可以使用 ACK，為 Lambda 重複使用此安全模型，不須建立新的使用者和政策。
- 您的組織有 DevOps 程序，可使用 Kubernetes 清單檔案，將資源部署到 Amazon Elastic Kubernetes Service (Amazon EKS) 叢集中。您可以透過 ACK，使用清單檔案佈建 Lambda 函數，不須建立另外的基礎設施即程式碼範本。

如需使用 ACK 的詳細資訊，請參閱 ACK 文件中的 Lambda 教學課程。

Crossplane

Crossplane 是開放原始碼雲端原生運算基金會 (CNCF) 專案，使用 Kubernetes 來管理雲端基礎設施資源。開發人員可以透過 Crossplane 請求基礎設施，不須了解其複雜性。平台團隊保留基礎設施佈建和管理方式的控制權。

您可以使用 Crossplane，透過偏好的 Kubernetes 工具鏈 (例如 kubectl) 以及任何可將清單檔案部署到 Kubernetes 的 CI/CD 管道，部署和管理 Lambda 函數。以下是一些透過 Crossplane 佈建 Lambda 函數的範例使用案例：

- 您的組織希望確保 Lambda 函數具有正確標籤來強制遵循法規。平台團隊可以使用 Crossplane 組合，透過 API 抽象來定義此政策。開發人員隨後可以使用這些抽象來部署具標籤的 Lambda 函數。
- 您的專案搭配 Kubernetes 使用 GITOps。在此模型中，Kubernetes 會持續協調 git 儲存庫 (所需狀態) 與叢集內執行的資源 (目前狀態)。如果有差異，GITOps 程序會自動變更叢集。您可以搭配 Kubernetes 使用 GITOps，利用諸如 CRD 和控制器的熟悉 Kubernetes 工具和概念，透過 Crossplane 來部署和管理 Lambda 函數。

若要進一步了解如何搭配 Lambda 使用 Crossplane，請參閱下列內容：

- AWS Blueprints for Crossplane：此儲存庫包括如何使用 Crossplane 部署 AWS 資源 (包括 Lambda 函數) 的範例。

> **Note**
>
> 目前正在積極開發 AWS Blueprints for Crossplane，因此不得用於生產環境。

- 使用 Amazon EKS 和 Crossplane 部署 Lambda：此影片展示透過 Crossplane 部署 AWS 無伺服器架構的進階範例，並從開發人員和平台的角度探索設計。

搭配使用 Lambda 與 Amazon MQ

> **Note**
>
> 如要將資料傳送到 Lambda 函數以外的目標，或在傳送資料之前讓資料更豐富，請參閱 Amazon EventBridge Pipes。

Amazon MQ 是一項受管訊息代理程式服務，適用於 Apache ActiveMQ 和 RabbitMQ。訊息代理程式透過主題或佇列事件目的地，允許軟體應用程式和元件使用各種程式設計語言、作業系統和正式簡訊協定來進行通訊。

Amazon MQ 還可以透過安裝 ActiveMQ 或 RabbitMQ 代理程式，並提供不同的網路拓撲和其他基礎架構需求，來代表您管理 Amazon Elastic Compute Cloud (Amazon EC2) 執行個體。

您可以使用 Lambda 函數來處理您的 Amazon MQ 訊息代理程式中的記錄。Lambda 透過事件來源映射叫用函數，這是從您的代理程式讀取訊息並同步叫用函數的 Lambda 資源。

> **Warning**
>
> Lambda 事件來源映射至少會處理每個事件一次，而且可能會重複處理記錄。為避免與重複事件相關的潛在問題，強烈建議您讓函數程式碼具有等冪性。若要進一步了解，請參閱 AWS 知識中心中的如何使 Lambda 函數具有等冪性。

Amazon MQ 事件來源映射具有下列組態限制：

- 並行：使用 Amazon MQ 事件來源映射的 Lambda 函數具有預設並行上限設定。若使用 ActiveMQ，Lambda 服務會將並行執行環境的數量上限設為每個 Amazon MQ 事件來源映射 5 個。若使用 RabbitMQ，並行執行環境的數量上限為每個 Amazon MQ 事件來源映射 1 個。即使您變更函數的保留或佈建並行設定，Lambda 服務還是無法提供更多可用的執行環境。若要請求增加單一 Amazon MQ 事件來源映射的預設並行上限， 支援 請聯絡事件來源映射 UUID 以及 區域。由於增加會套用在特定事件來源映射層級，而不是帳戶或區域層級，因此您需要為每個事件來源映射手動請求擴展增加。
- 跨帳戶 - Lambda 不支援跨帳戶處理。您無法用 Lambda 處理來自不同 AWS 帳戶中 Amazon MQ 訊息代理程式的記錄。
- 身分驗證支援 - 對於 ActiveMQ，僅支援 ActiveMQ SimpleAuthenticationPlugin。對於 RabbitMQ，僅支援 PLAIN 身分驗證機制。使用者必須使用 AWS Secrets Manager 來管理其登入資料。如需

ActiveMQ 身分驗證的詳細資訊，請參閱 Amazon MQ 開發人員指南中的將 ActiveMQ 代理程式與 LDAP 整合。

- 連線配額 - 代理程式每個線路層級協定允許的連線數量上限。此配額以代理程式執行個體類型為基礎。如需詳細資訊，請參閱 Amazon MQ 開發人員指南中的 Amazon MQ 中的配額的代理程式。
- 連線 - 您可以在公有或私有 Virtual Private Cloud (VPC) 中建立代理程式。若是私有 VPC，您的 Lambda 函數需要存取 VPC 才能接收訊息。如需詳細資訊，請參閱本節稍後的the section called “設定網路安全”。
- 事件目的地 - 僅支援佇列目的地。然而，您可以使用虛擬主題，當作為佇列與 Lambda 互動時，其行為在內部可作為主題。如需詳細資訊，請參閱 Apache ActiveMQ 網站上的虛擬目的地，以及 RabbitMQ 網站上的虛擬主機。
- 網路拓撲 - 對於 ActiveMQ，每個事件來源映射僅支援單一執行個體或待命代理程式。對於 RabbitMQ，每個事件來源映射僅支援單一執行個體代理程式或叢集部署。單一執行個體代理程式需要容錯移轉端點。如需這些代理程式部署模式的詳細資訊，請參閱 Amazon MQ 開發人員指南中的 ActiveMQ 代理程式架構和 Rabbit MQ 代理程式架構。
- 協定 - 支援的協定取決於 Amazon MQ 整合的類型。
  - 對於 ActiveMQ 整合，Lambda 會使用 OpenWire/Java Message Service (JMS) 協定來取用訊息。不支援透過其他協定來取用訊息。在 JMS 協定中，僅支援 `TextMessage` 和 `BytesMessage`。Lambda 也支援 JMS 自訂屬性。如需有關 OpenWire 協定的詳細資訊，請參閱 Apache ActiveMQ 網站上的 OpenWire。
  - 對於 RabbitMQ 整合，Lambda 透過 AMQP 0-9-1 協定來取用訊息。不支援透過其他協定來取用訊息。如需 RabbitMQ 實作 AMQP 0-9-1 協定的詳細資訊，請參閱 RabbitMQ 網站上的 AMQP 0-9-1 完整參考指南。

Lambda 會自動支援 Amazon MQ 支援的最新版 ActiveMQ 和 RabbitMQ。如需支援的最新版，請參閱 Amazon MQ 開發人員指南中的 Amazon MQ 版本備註。

> **Note**
>
> 根據預設，Amazon MQ 具有每週代理程式維護時段。代理程式在該時段不可用。若是無待命狀態的代理程式，則 Lambda 無法在該時段處理任何訊息。

主題

- 了解 Amazon MQ 的 Lambda 取用者群組
- 設定 Lambda 的 Amazon MQ 事件來源

- 事件來源映射參數
- 從 Amazon MQ 事件來源篩選事件
- 對 Amazon MQ 事件來源映射錯誤進行疑難排解

了解 Amazon MQ 的 Lambda 取用者群組

若要與 Amazon MQ 互動，Lambda 會建立可以從您的 Amazon MQ 代理程式讀取的取用者群體。建立取用者群組時，會使用與事件來源映射 UUID 相同的 ID。

對於 Amazon MQ 事件來源，Lambda 會批次處理記錄，並在單個承載中將它們傳送到您的函數。要控制行為，您可以設定批次間隔和批次大小。Lambda 會提取訊息，直到它處理最大為 6 MB 的承載大小、批次間隔過期或記錄數達到完整批次大小。如需詳細資訊，請參閱批次處理行為。

取用者群組會將訊息擷取為位元組 BLOB，並透過 base64 將其編碼成單一 JSON 承載，然後調用您的函數。如果函數針對批次中的任何訊息傳回錯誤，Lambda 會重試整個批次的訊息，直至處理成功或訊息過期。

> **Note**
>
> 雖然 Lambda 函數的逾時上限通常為 15 分鐘，但 Amazon MSK、自我管理的 Apache Kafka、Amazon DocumentDB 以及 Amazon MQ for ActiveMQ 和 Amazon MQ for RabbitMQ 的事件來源映射只支援 14 分鐘逾時限制上限的函數。此限制條件可確保事件來源映射能夠正確處理函數錯誤和重試。

您可以使用 Amazon CloudWatch 中的 `ConcurrentExecutions` 指標，監控指定函數的並行用量。如需並行的詳細資訊，請參閱the section called “設定預留並行”。

Example Amazon MQ 記錄事件

ActiveMQ

```
{
   "eventSource": "aws:mq",
   "eventSourceArn": "arn:aws:mq:us-
east-2:111122223333:broker:test:b-9bcfa592-423a-4942-879d-eb284b418fc8",
   "messages": [
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
```

```
        "messageType": "jms/text-message",
        "deliveryMode": 1,
        "replyTo": null,
        "type": null,
        "expiration": "60000",
        "priority": 1,
        "correlationId": "myJMSCoID",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"QUJDOkFBQUE=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
        "properties": {
          "index": "1",
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
      },
      {
        "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
        "messageType": "jms/bytes-message",
        "deliveryMode": 1,
        "replyTo": null,
        "type": null,
        "expiration": "60000",
        "priority": 2,
        "correlationId": "myJMSCoID1",
        "redelivered": false,
        "destination": {
          "physicalName": "testQueue"
        },
        "data":"LQaGQ82S48k=",
        "timestamp": 1598827811958,
        "brokerInTime": 1598827811958,
        "brokerOutTime": 1598827811959,
        "properties": {
          "index": "1",
          "doAlarm": "false",
          "myCustomProperty": "value"
        }
```

```
      }
   ]
}
```

RabbitMQ

```
{
  "eventSource": "aws:rmq",
  "eventSourceArn": "arn:aws:mq:us-
east-2:111122223333:broker:pizzaBroker:b-9bcfa592-423a-4942-879d-eb284b418fc8",
  "rmqMessagesByQueue": {
    "pizzaQueue::/": [
      {
        "basicProperties": {
          "contentType": "text/plain",
          "contentEncoding": null,
          "headers": {
            "header1": {
              "bytes": [
                118,
                97,
                108,
                117,
                101,
                49
              ]
            },
            "header2": {
              "bytes": [
                118,
                97,
                108,
                117,
                101,
                50
              ]
            },
            "numberInHeader": 10
          },
          "deliveryMode": 1,
          "priority": 34,
          "correlationId": null,
```

```
          "replyTo": null,
          "expiration": "60000",
          "messageId": null,
          "timestamp": "Jan 1, 1970, 12:33:41 AM",
          "type": null,
          "userId": "AIDACKCEVSQ6C2EXAMPLE",
          "appId": null,
          "clusterId": null,
          "bodySize": 80
        },
        "redelivered": false,
        "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
      }
    ]
  }
}
```

> **Note**
>
> 在 RabbitMQ 範例中，`pizzaQueue` 是 RabbitMQ 佇列的名稱，`/` 是虛擬主機的名稱。接收訊息時，事件來源會在 `pizzaQueue::/` 列出訊息。

設定 Lambda 的 Amazon MQ 事件來源

主題

- 設定網路安全
- 建立事件來源映射

設定網路安全

若要透過事件來源映射授予 Lambda 對 Amazon MQ 的完整存取權，您的代理程式必須使用公有端點 (公有 IP 位址)，或者您必須提供建立代理程式之 Amazon VPC 的存取權。

當您將 Amazon MQ 與 Lambda 搭配使用時，請建立 AWS PrivateLink VPC 端點，為您的函數提供 Amazon VPC 中資源的存取權。

> **Note**
>
> AWS PrivateLink 具有使用事件輪詢器預設（隨需）模式之事件來源映射的函數需要 VPC 端點。如果您的事件來源映射使用佈建模式，則不需要設定 AWS PrivateLink VPC 端點。

建立端點以提供對下列資源的存取權：

- Lambda：為 Lambda 服務主體建立端點。
- AWS STS — 為 建立端點， AWS STS 以便服務主體代表您擔任角色。
- Secrets Manager：如果您的代理程式使用 Secrets Manager 來儲存憑證，則請為 Secrets Manager 建立端點。

或者，在 Amazon VPC 中的每個公有子網路上設定一個 NAT 閘道。如需詳細資訊，請參閱the section called “Lambda 函數的網際網路存取”。

當您為 Amazon MQ 建立事件來源映射時，Lambda 會檢查是否已存在彈性網絡介面 (ENI)，適用於為您的 Amazon VPC 設定的子網路和安全群組。如果 Lambda 找到現有的 ENI，則它會嘗試重複使用它們。否則，Lambda 會建立新的 ENI 以連線至事件來源並調用您的函數。

> **Note**
>
> Lambda 函數一律會在 Lambda 服務所擁有的 VPC 內執行。函數的 VPC 組態不會影響事件來源映射。只有事件來源的聯網組態會決定 Lambda 如何連線至您的事件來源。

為包含代理程式的 Amazon VPC 設定安全群組。根據預設，Amazon MQ 會使用下列連接埠：61617 (Amazon MQ for ActiveMQ) 和 5671 (Amazon MQ for RabbitMQ)。

- 傳入規則：允許與事件來源相關聯之安全群組的預設代理程式連接埠上的所有流量。或者，您可使用自我參照安全群組規則，允許來自同一安全群組內其他執行個體的存取。
- 傳出規則 – 如果您的函數需要與服務通訊443，允許外部目的地連接埠上的所有流量 AWS 。或者，如果您不需要與其他 AWS 服務通訊，您也可以使用自我參考安全群組規則來限制對代理程式的存取。
- Amazon VPC 端點傳入規則：如果您使用的是 Amazon VPC 端點，與您的 Amazon VPC 端點相關聯的安全群組必須允許來自代理程式安全群組的連接埠 443 上的傳入流量。

如果代理程式使用身分驗證，也可以限制 Secrets Manager 端點的端點政策。若要呼叫 Secrets Manager API，Lambda 會使用您的函數角色，而不是 Lambda 服務主體。

Example VPC 端點政策 — Secrets Manager 端點

```
{
      "Statement": [
          {
              "Action": "secretsmanager:GetSecretValue",
              "Effect": "Allow",
              "Principal": {
                  "AWS": [
                      "arn:aws::iam::123456789012:role/my-role"
                  ]
              },
              "Resource": "arn:aws::secretsmanager:us-west-2:123456789012:secret:my-
secret"
          }
      ]
  }
```

當您使用 Amazon VPC 端點時， 會使用端點的彈性網路界面 (ENI) AWS 路由您的 API 呼叫來叫用函數。Lambda 服務主體需要在使用這些 ENI 的任何角色和函數上呼叫 `lambda:InvokeFunction`。

根據預設，Amazon VPC 端點具有開放的 IAM 政策，允許廣泛存取資源。最佳實務是限制這些政策，以使用該端點執行所需的動作。為了確保事件來源映射能夠調用 Lambda 函數，VPC 端點政策必須允許 Lambda 服務主體呼叫 `sts:AssumeRole` 和 `lambda:InvokeFunction`。限制您的 VPC 端點政策以僅允許源自您組織內部的 API 呼叫，可阻止事件來源映射正常運作，因此在這些政策中需要 `"Resource": "*"`。

下列範例 VPC 端點政策展示了如何授予 Lambda 服務主體對 AWS STS 和 Lambda 端點的必要存取權。

Example VPC 端點政策 — AWS STS 端點

```
{
      "Statement": [
          {
              "Action": "sts:AssumeRole",
              "Effect": "Allow",
              "Principal": {
```

```
                "Service": [
                    "lambda.amazonaws.com"
                ]
            },
            "Resource": "*"
        }
    ]
}
```

Example VPC 端點政策 – Lambda 端點

```
{
      "Statement": [
          {
              "Action": "lambda:InvokeFunction",
              "Effect": "Allow",
              "Principal": {
                  "Service": [
                      "lambda.amazonaws.com"
                  ]
              },
              "Resource": "*"
          }
      ]
  }
```

建立事件來源映射

建立事件來源映射，指示 Lambda 將記錄從 Amazon MQ 代理程式傳送至 Lambda 函數。您可以建立多個事件來源映射，來使用多個函數處理相同資料，或使用單一函數處理來自多個來源的項目。

若要設定您的函數以從 Amazon MQ 讀取，請新增必要的許可，並在 Lambda 主控台中建立 MQ 觸發條件。

若要從 Amazon MQ 代理程式讀取記錄，您的 Lambda 函數需要將下列許可。您可以透過將許可陳述式新增至您的函數執行角色，授予 Lambda 許可，以與您的 Amazon MQ 代理程式及其基礎資源互動：

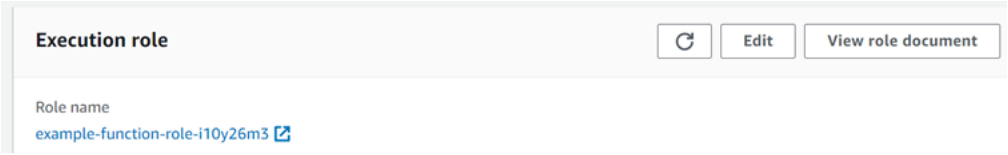
- mq:DescribeBroker
- secretsmanager:GetSecretValue
- ec2:CreateNetworkInterface

- ec2:DeleteNetworkInterface
- ec2:DescribeNetworkInterfaces
- ec2:DescribeSecurityGroups
- ec2:DescribeSubnets
- ec2:DescribeVpcs
- logs:CreateLogGroup
- logs:CreateLogStream
- 日誌：PutLogEvents

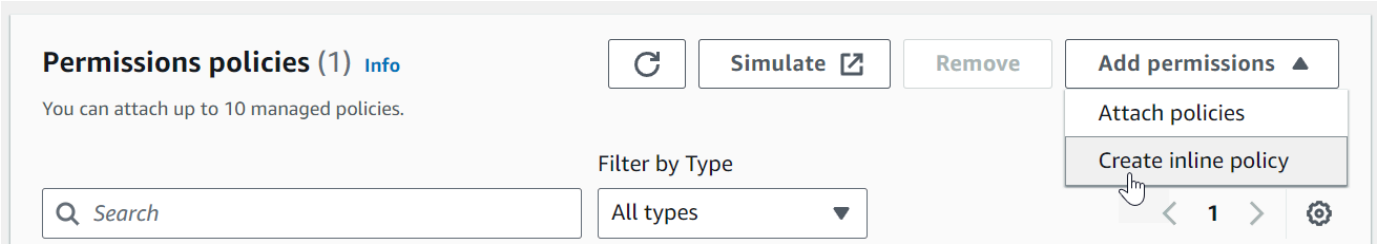
> **Note**
>
> 使用加密的客戶管理金鑰時，也請新增 `kms:Decrypt` 許可。

若要新增許可並建立觸發條件

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 依序選擇 Configuration (組態) 索引標籤和 Permissions (許可)。
4. 在角色名稱下面，選擇執行角色連結。此連結會在 IAM 主控台中開啟該角色。



5. 選擇新增許可，然後選擇建立內嵌政策。



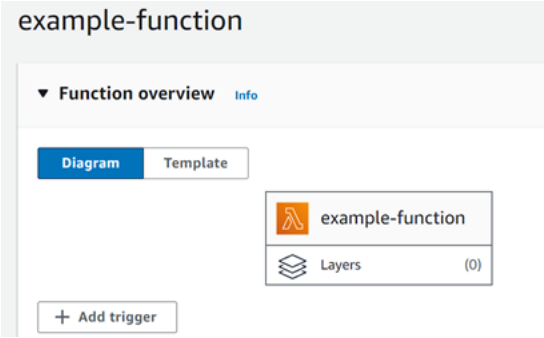
6. 在政策編輯器中，選擇 JSON。輸入下列政策。您的函數需要這些許可才能從 Amazon MQ 代理程式讀取。

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "mq:DescribeBroker",
          "secretsmanager:GetSecretValue",
          "ec2:CreateNetworkInterface",
          "ec2:DeleteNetworkInterface",
          "ec2:DescribeNetworkInterfaces",
          "ec2:DescribeSecurityGroups",
          "ec2:DescribeSubnets",
          "ec2:DescribeVpcs",
          "logs:CreateLogGroup",
          "logs:CreateLogStream",
          "logs:PutLogEvents"
        ],
        "Resource": "*"
      }
    ]
  }
```

> **Note**
>
> 使用加密的客戶管理金鑰時，還必須新增 `kms:Decrypt` 許可。

7. 選擇下一步。輸入政策名稱，然後選擇建立政策。
8. 在 Lambda 主控台中返回您的 Lambda 函數。在函數概觀下，選擇新增觸發條件。



9. 選擇 MQ 觸發條件類型。
10. 設定需要的選項，然後選擇新增。

Lambda 支援 Amazon MQ 事件來源的下列選項：

- MQ broker (MQ 代理程式) – 選取 Amazon MQ 代理程式。
- Batch size (批次大小) - 設定單一批次中要擷取的訊息數目上限。
- Queue name (佇列名稱) - 輸入要取用的 Amazon MQ 佇列。
- Source access configuration (來源存取組態) - 輸入儲存您代理程式憑證的虛擬主機資訊和 Secrets Manager 機密。
- Enable trigger (啟用觸發條件) - 停用觸發條件以停止處理記錄。

若要啟用或停用觸發條件 (或將其刪除)，請在設計工具中選擇 MQ 觸發條件。若要重新設定觸發條件，請使用事件來源映射 API 操作。

事件來源映射參數

所有 Lambda 事件來源類型都會共用相同的 CreateEventSourceMapping 和 UpdateEventSourceMapping API 操作。但是，只有一些參數適用於 Amazon MQ 和 RabbitMQ。

參數	必要	預設	備註
BatchSize	否	100	上限：10,000
已啟用	N	true	無
FunctionName	是	N/A	無

參數	必要	預設	備註
FilterCriteria	N	N/A	控制 Lambda 將哪些事件傳送至您的函數
MaximumBatchingWindowInSeconds	N	500 毫秒	批次處理行為
佇列	N	N/A	要使用的 Amazon MQ 代理程式目的地佇列的名稱。
SourceAccessConfigurations	N	N/A	若為 ActiveMQ，可使用 BASIC_AUTH 憑證。若為 RabbitMQ，可同時包含 BASIC_AUTH 憑證和 VIRTUAL_HOST 資訊。

從 Amazon MQ 事件來源篩選事件

您可以使用事件篩選來控制 Lambda 將哪些記錄從串流或佇列中傳送至函數。如需事件篩選運作方式的一般資訊，請參閱the section called “事件篩選”。

本節重點介紹 Amazon MQ 事件來源的事件篩選。

> **Note**
>
> Amazon MQ 事件來源映射僅支援依據 `data` 鍵進行篩選。

主題

- Amazon MQ 事件篩選基本概念

Amazon MQ 事件篩選基本概念

假設您的 Amazon MQ 訊息佇列包含有效 JSON 格式或純字串的訊息。範例記錄如下所示，資料在 `data` 欄位中會轉換為 Base64 編碼字串。

ActiveMQ

```
{
    "messageID": "ID:b-9bcfa592-423a-4942-879d-eb284b418fc8-1.mq.us-
east-2.amazonaws.com-37557-1234520418293-4:1:1:1:1",
    "messageType": "jms/text-message",
    "deliveryMode": 1,
    "replyTo": null,
    "type": null,
    "expiration": "60000",
    "priority": 1,
    "correlationId": "myJMSCoID",
    "redelivered": false,
    "destination": {
      "physicalName": "testQueue"
    },
    "data":"QUJDOkFBQUE=",
    "timestamp": 1598827811958,
    "brokerInTime": 1598827811958,
    "brokerOutTime": 1598827811959,
    "properties": {
      "index": "1",
      "doAlarm": "false",
      "myCustomProperty": "value"
    }
}
```

RabbitMQ

```
{
    "basicProperties": {
        "contentType": "text/plain",
        "contentEncoding": null,
        "headers": {
            "header1": {
                "bytes": [
                  118,
                  97,
```

```
                    108,
                    117,
                    101,
                    49
                ]
            },
            "header2": {
                "bytes": [
                    118,
                    97,
                    108,
                    117,
                    101,
                    50
                ]
            },
            "numberInHeader": 10
        },
        "deliveryMode": 1,
        "priority": 34,
        "correlationId": null,
        "replyTo": null,
        "expiration": "60000",
        "messageId": null,
        "timestamp": "Jan 1, 1970, 12:33:41 AM",
        "type": null,
        "userId": "AIDACKCEVSQ6C2EXAMPLE",
        "appId": null,
        "clusterId": null,
        "bodySize": 80
        },
    "redelivered": false,
    "data": "eyJ0aW1lb3V0IjowLCJkYXRhIjoiQ1pybWYwR3c4T3Y0YnFMUXhENEUifQ=="
}
```

對於 Active MQ 和 Rabbit MQ 代理程式，您可以使用事件篩選透過 `data` 索引鍵來篩選記錄。假設您的 Amazon MQ 佇列包含以下 JSON 格式的訊息。

```
{
    "timeout": 0,
    "IPAddress": "203.0.113.254"
}
```

若要僅篩選 timeout 欄位大於 0 的記錄，FilterCriteria 物件將如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : { \"timeout\" : [ { \"numeric\": [ \">\",
 0] } } ] } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 Pattern 在純文字 JSON 中擴展的值。

```
{
    "data": {
        "timeout": [ { "numeric": [ ">", 0 ] } ]
        }
}
```

您可以使用主控台、AWS CLI 或 AWS SAM 範本新增篩選條件。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 來建立具有這些篩選條件標準的新事件來源映射，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : { \"timeout\" :
 [ { \"numeric\": [ \">\", 0 ] } ] } }"}]}'
```

AWS SAM

若要使用 AWS SAM 新增此篩選條件，請將下列程式碼片段新增到事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : { "timeout" : [ { "numeric": [ ">", 0 ] } ] } }'
```

使用 Amazon MQ，您也可以篩選訊息為純字串的記錄。假設您只想處理訊息以「結果：」開頭的記錄。`FilterCriteria` 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"data\" : [ { \"prefix\": \"Result: \" } ] }"
        }
    ]
}
```

補充說明，此處是篩選條件的 `Pattern` 在純文字 JSON 中擴展的值。

```
{
    "data": [
        {
        "prefix": "Result: "
        }
    ]
}
```

您可以使用主控台、AWS CLI 或 AWS SAM 範本新增篩選條件。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "data" : [ { "prefix": "Result: " } ] }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 來建立具有這些篩選條件標準的新事件來源映射，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:mq:us-east-2:123456789012:broker:my-
broker:b-8ac7cc01-5898-482d-be2f-a6b596050ea8 \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"data\" : [ { \"prefix\":
 \"Result: \" } ] }"}]}'
```

AWS SAM

若要使用 AWS SAM 新增此篩選條件，請將下列程式碼片段新增到事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "data" : [ { "prefix": "Result " } ] }'
```

Amazon MQ 訊息必須是 UTF-8 編碼的字串，可以是純字串或 JSON 格式。這是因為 Lambda 會在套用篩選條件之前，將 Amazon MQ 位元組陣列解碼成 UTF-8。如果您的訊息使用其他編碼方式 (例如 UTF-16 或 ASCII)，或者訊息格式與 `FilterCriteria` 格式不相符，則 Lambda 只會處理中繼資料篩選條件。下表摘要說明特定行為：

傳入訊息 格式	訊息屬性的篩選條件模式格式	產生的動作
純文字的字串	純文字的字串	根據您的篩選條件標準之 Lambda 篩選條件。
純文字的字串	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
純文字的字串	有效的 JSON	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	純文字的字串	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	有效的 JSON	根據您的篩選條件標準之 Lambda 篩選條件。
非 UTF-8 編碼字串	JSON、純字串或沒有模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。

對 Amazon MQ 事件來源映射錯誤進行疑難排解

當 Lambda 函數遇到無法復原的錯誤時，您的 Amazon MQ 取用者將停止處理記錄。任何其他取用者可能會繼續處理，只要他們沒有遇到相同的錯誤。若要判斷停止 `StateTransitionReason` 取用者的潛在原因，請檢查傳回詳細資料中的 `EventSourceMapping` 欄位，以取得下列其中一個程式碼：

`ESM_CONFIG_NOT_VALID`

事件來源對應組態無效。

EVENT_SOURCE_AUTHN_ERROR

Lambda 驗證事件來源失敗。

EVENT_SOURCE_AUTHZ_ERROR

Lambda 沒有存取事件來源所需的許可。

FUNCTION_CONFIG_NOT_VALID

該函數的配置無效。

如果記錄因大小而遭 Lambda 棄置，則也將不會得到處理。Lambda 記錄的大小限制為 6MB。若要在函數錯誤時重新傳遞訊息，您可以使用無效字母佇列 (DLQ)。如需詳細資訊，請參閱 Apache ActiveMQ 網站上的訊息重新傳遞和 DLQ 處理，以及 RabbitMQ 網站上的可靠性指南。

> **Note**
>
> Lambda 不支援自訂重新傳遞政策，相反，Lambda 會使用具有來自 Apache ActiveMQ 網站重新傳遞政策頁面中之預設值的政策，並將 `maximumRedeliveries` 設定為 6。

AWS Lambda 搭配 Amazon RDS 使用

您可以直接或透過 Amazon RDS Proxy 將 Lambda 函數連線到 Amazon Relational Database Service (Amazon RDS)。直接連線適用於簡單的案例，生產環境則建議使用代理。資料庫代理管理許多共用資料庫連線，讓函數在不耗盡資料庫連線的情況下達到高並行層級。

我們建議將 Amazon RDS Proxy 用於 Lambda 函數，這些函數會頻繁進行短資料庫連線，或是開啟和關閉大量資料庫連線。如需詳細資訊，請參閱《Amazon Relational Database Service 開發人員指南》中的 Automatically connecting a Lambda function and a DB instance 一節。

> **Tip**
>
> 若要將 Lambda 函式快速連線至 Amazon RDS 資料庫，您可以使用主控台內的引導精靈。若要開啟精靈，請執行下列操作：
>
> 1. 開啟 Lambda 主控台中的函數頁面。
> 2. 選取您要連接資料庫的函數。
> 3. 在組態索引標籤，選取 RDS 資料庫。
> 4. 選擇連接至 RDS 資料庫。
>
> 將函數連接至資料庫之後，您可以選擇新增代理來建立代理。

設定函數以使用 RDS 資源

在 Lambda 主控台中，您可以佈建和設定 Amazon RDS 資料庫執行個體和代理資源。您可以在組態索引標籤下導覽至 RDS 資料庫來執行此操作。或者，您也可以在 Amazon RDS 主控台中建立與設定 Lambda 函數的連線。設定 RDS 資料庫執行個體以與 Lambda 搭配使用時，請注意以下條件：

- 若要連線到資料庫，您的函數必須位於資料庫執行所在的相同 Amazon VPC 內。
- 您可以搭配 MySQL、MariaDB、PostgreSQL 或 Microsoft SQL Server 引擎，使用 Amazon RDS 資料庫。
- 您也可以搭配 MySQL 或 PostgreSQL 引擎，使用 Aurora DB 叢集。
- 您需要提供 Secrets Manager 秘密以用於資料庫身分驗證。
- IAM 角色必須提供使用秘密的許可，而受信任的政策必須允許 Amazon RDS 擔任該角色。
- 使用主控台設定 Amazon RDS 資源，並將其連線至函數的 IAM 主體必須具有下列許可：

許可政策範例

> **Note**
>
> 只有在您設定 Amazon RDS Proxy 來管理資料庫連線集區時，才需要這些 Amazon RDS Proxy 許可。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:CreateSecurityGroup",
        "ec2:DescribeSecurityGroups",
        "ec2:DescribeSubnets",
        "ec2:DescribeVpcs",
        "ec2:AuthorizeSecurityGroupIngress",
        "ec2:AuthorizeSecurityGroupEgress",
        "ec2:RevokeSecurityGroupEgress",
        "ec2:CreateNetworkInterface",
        "ec2:DeleteNetworkInterface",
        "ec2:DescribeNetworkInterfaces"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "rds-db:connect",
        "rds:CreateDBProxy",
        "rds:CreateDBInstance",
        "rds:CreateDBSubnetGroup",
        "rds:DescribeDBClusters",
        "rds:DescribeDBInstances",
        "rds:DescribeDBSubnetGroups",
        "rds:DescribeDBProxies",
        "rds:DescribeDBProxyTargets",
        "rds:DescribeDBProxyTargetGroups",
```

```
        "rds:RegisterDBProxyTargets",
        "rds:ModifyDBInstance",
        "rds:ModifyDBProxy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:ListFunctions",
        "lambda:UpdateFunctionConfiguration"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "iam:AttachRolePolicy",
        "iam:CreateRole",
        "iam:CreatePolicy"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetResourcePolicy",
        "secretsmanager:GetSecretValue",
        "secretsmanager:DescribeSecret",
        "secretsmanager:ListSecretVersionIds",
        "secretsmanager:CreateSecret"
      ],
      "Resource": "*"
    }
  ]
}
```

Amazon RDS 會按資料庫執行個體大小收取代理程式的小時費率，請參閱 RDS 代理定價以了解詳細資訊。如需代理連線的一般詳細資訊，請參閱《Amazon RDS 使用者指南》中的使用 Amazon RDS Proxy。

Amazon RDS 連線的 SSL/TLS 要求

若要對 Amazon RDS 資料庫執行個體進行安全的 SSL/TLS 連線，您的 Lambda 函式必須使用信任的憑證來驗證資料庫伺服器的身分。Lambda 會根據您的部署套件類型，採用不同方式處理這些憑證：

- .zip 壓縮檔：憑證處理方式會因執行時期而異：
  - Node.js 18 及更早版本：Lambda 會自動包含 CA 憑證與 RDS 憑證。
  - Node.js 20 及更新版本：Lambda 預設不再載入額外的 CA 憑證。將 `NODE_EXTRA_CA_CERTS` 環境變數設為 `/var/runtime/ca-cert.pem`。

  AWS 區域 將新的 Amazon RDS 憑證新增至 Lambda 受管執行期最多可能需要 4 週的時間。
- 容器映像： AWS 基礎映像僅包含 CA 憑證。如果您的函式需連線至 Amazon RDS 資料庫執行個體，必須在容器映像中包含對應的憑證。在您的 Dockerfile 中，下載與您託管資料庫 AWS 區域 的對應的憑證套件。範例：

```
RUN curl https://truststore.pki.rds.amazonaws.com/us-east-1/us-east-1-bundle.pem -o /
us-east-1-bundle.pem
```

此命令會下載 Amazon RDS 憑證套件，並將其儲存在容器根目錄中的絕對路徑 `/us-east-1-bundle.pem`。在函式程式碼中設定資料庫連線時，必須參考此確切路徑。範例：

Node.js

由於 Node.js 資料庫用戶端需要記憶體中的實際憑證內容，而不只是憑證檔案的路徑，因此 `readFileSync` 函式為必要項目。如果沒有 `readFileSync` 函式，用戶端會將路徑字串解譯為憑證內容，導致「憑證鏈中存在自我簽署憑證」錯誤。

Example OCI 函式的 Node.js 連線組態

```
import { readFileSync } from 'fs';

// ...

let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: {
```

```
        ca: readFileSync('/us-east-1-bundle.pem') // Load RDS certificate content
 from file into memory
    }
};
```

Python

Example OCI 函式的 Python 連線組態

```
connection = pymysql.connect(
    host=proxy_host_name,
    user=db_username,
    password=token,
    db=db_name,
    port=port,
    ssl={'ca': '/us-east-1-bundle.pem'}  #Path to the certificate in container
)
```

Java

對於使用 JDBC 連線的 Java 函式，連線字串必須包含：

- `useSSL=true`
- `requireSSL=true`
- 指向容器映像中 Amazon RDS 憑證位置的 `sslCA` 參數

Example OCI 函式的 Java 連線字串

```
// Define connection string
String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true&sslCA=/us-east-1-bundle.pem", // Path to the certificate
 in container
        System.getenv("ProxyHostName"),
        System.getenv("Port"),
        System.getenv("DBName"));
```

.NET

Example OCI 函式中 MySQL 連線的 .NET 連線字串

```
/// Build the Connection String with the Token
```

```
string connectionString =
 $"Server={Environment.GetEnvironmentVariable("RDS_ENDPOINT")};" +
                          $"Port={Environment.GetEnvironmentVariable("RDS_PORT")};" +

 $"Uid={Environment.GetEnvironmentVariable("RDS_USERNAME")};" +
                          $"Pwd={authToken};" +
                          "SslMode=Required;" +
                          "SslCa=/us-east-1-bundle.pem";  // Path to the certificate
 in container
```

Go

對於使用 MySQL 連線的 Go 函式，請將 Amazon RDS 憑證載入憑證集區，並向 MySQL 驅動程式註冊該憑證。連線字串隨後必須透過 `tls` 參數引用此組態。

Example OCI 函式中 MySQL 連線的 Go 程式碼

```
import (
    "crypto/tls"
    "crypto/x509"
    "os"
    "github.com/go-sql-driver/mysql"
)

...

// Create certificate pool and register TLS config
rootCertPool := x509.NewCertPool()
pem, err := os.ReadFile("/us-east-1-bundle.pem")  // Path to the certificate in
 container
if err != nil {
    panic("failed to read certificate file: " + err.Error())
}
if ok := rootCertPool.AppendCertsFromPEM(pem); !ok {
    panic("failed to append PEM")
}

mysql.RegisterTLSConfig("custom", &tls.Config{
    RootCAs: rootCertPool,
})

dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?allowCleartextPasswords=true&tls=custom",
    dbUser, authenticationToken, dbEndpoint, dbName,
```

```
)
```

Ruby

Example OCI 函式的 Ruby 連線組態

```
conn = Mysql2::Client.new(
    host: endpoint,
    username: user,
    password: token,
    port: port,
    database: db_name,
    sslca: '/us-east-1-bundle.pem',  # Path to the certificate in container
    sslverify: true
)
```

連線至 Lambda 函數中的 Amazon RDS 資料庫

以下程式碼範例示範如何實作連線至 Amazon RDS 資料庫的 Lambda 函式。該函數會提出簡單的資料庫請求並傳回結果。

> **Note**
>
> 這些程式碼範例僅適用於 .zip 部署套件。如果使用容器映像部署函式，您必須在函式程式碼中指定 Amazon RDS 憑證檔案，如上一節所述。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 連線至 Lambda 函式中的 Amazon RDS 資料庫。

```
using System.Data;
```

```
using System.Text.Json;
using Amazon.Lambda.APIGatewayEvents;
using Amazon.Lambda.Core;
using MySql.Data.MySqlClient;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace aws_rds;

public class InputModel
{
    public string key1 { get; set; }
    public string key2 { get; set; }
}

public class Function
{
    /// <summary>
    // Handles the Lambda function execution for connecting to RDS using IAM
 authentication.
    /// </summary>
    /// <param name="input">The input event data passed to the Lambda function</
param>
    /// <param name="context">The Lambda execution context that provides runtime
 information</param>
    /// <returns>A response object containing the execution result</returns>

    public async Task<APIGatewayProxyResponse>
 FunctionHandler(APIGatewayProxyRequest request, ILambdaContext context)
    {
        // Sample Input: {"body": "{\"key1\":\"20\", \"key2\":\"25\"}"}
        var input = JsonSerializer.Deserialize<InputModel>(request.Body);

        /// Obtain authentication token
        var authToken = RDSAuthTokenGenerator.GenerateAuthToken(
            Environment.GetEnvironmentVariable("RDS_ENDPOINT"),
            Convert.ToInt32(Environment.GetEnvironmentVariable("RDS_PORT")),
            Environment.GetEnvironmentVariable("RDS_USERNAME")
        );

        /// Build the Connection String with the Token
```

```
        string connectionString =
$"Server={Environment.GetEnvironmentVariable("RDS_ENDPOINT")};" +

$"Port={Environment.GetEnvironmentVariable("RDS_PORT")};" +

$"Uid={Environment.GetEnvironmentVariable("RDS_USERNAME")};" +
                                  $"Pwd={authToken};";


        try
        {
            await using var connection = new MySqlConnection(connectionString);
            await connection.OpenAsync();

            const string sql = "SELECT @param1 + @param2 AS Sum";

            await using var command = new MySqlCommand(sql, connection);
            command.Parameters.AddWithValue("@param1", int.Parse(input.key1 ??
"0"));
            command.Parameters.AddWithValue("@param2", int.Parse(input.key2 ??
"0"));

            await using var reader = await command.ExecuteReaderAsync();
            if (await reader.ReadAsync())
            {
                int result = reader.GetInt32("Sum");

                //Sample Response: {"statusCode":200,"body":"{\"message\":\"The
sum is: 45\"}","isBase64Encoded":false}
                return new APIGatewayProxyResponse
                {
                    StatusCode = 200,
                    Body = JsonSerializer.Serialize(new { message = $"The sum is:
{result}" })
                };
            }

        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }

        return new APIGatewayProxyResponse
```

```
        {
            StatusCode = 500,
            Body = JsonSerializer.Serialize(new { error = "Internal server
 error" })
        };
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
/*
Golang v2 code here.
*/

package main

import (
 "context"
 "database/sql"
 "encoding/json"
 "fmt"
 "os"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
)

type MyEvent struct {
 Name string `json:"name"`
```

```
}

func HandleRequest(event *MyEvent) (map[string]interface{}, error) {

 var dbName string = os.Getenv("DatabaseName")
 var dbUser string = os.Getenv("DatabaseUser")
 var dbHost string = os.Getenv("DBHost") // Add hostname without https
 var dbPort int = os.Getenv("Port")      // Add port number
 var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
 var region string = os.Getenv("AWS_REGION")

 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  panic("configuration error: " + err.Error())
 }

 authenticationToken, err := auth.BuildAuthToken(
  context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
 if err != nil {
  panic("failed to create authentication token: " + err.Error())
 }

 dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
  dbUser, authenticationToken, dbEndpoint, dbName,
 )

 db, err := sql.Open("mysql", dsn)
 if err != nil {
  panic(err)
 }

 defer db.Close()

 var sum int
 err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
 if err != nil {
  panic(err)
 }
 s := fmt.Sprint(sum)
 message := fmt.Sprintf("The selected sum is: %s", s)

 messageBytes, err := json.Marshal(message)
 if err != nil {
  return nil, err
```

```
 }

 messageString := string(messageBytes)
 return map[string]interface{}{
  "statusCode": 200,
  "headers":    map[string]string{"Content-Type": "application/json"},
  "body":       messageString,
 }, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.rdsdata.RdsDataClient;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementRequest;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementResponse;
import software.amazon.awssdk.services.rdsdata.model.Field;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
```

```
public class RdsLambdaHandler implements
 RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    @Override
    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent
 event, Context context) {
        APIGatewayProxyResponseEvent response = new
 APIGatewayProxyResponseEvent();

        try {
            // Obtain auth token
            String token = createAuthToken();

            // Define connection configuration
            String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true",
                    System.getenv("ProxyHostName"),
                    System.getenv("Port"),
                    System.getenv("DBName"));

            // Establish a connection to the database
            try (Connection connection =
 DriverManager.getConnection(connectionString, System.getenv("DBUserName"),
 token);
                 PreparedStatement statement =
 connection.prepareStatement("SELECT ? + ? AS sum")) {

                statement.setInt(1, 3);
                statement.setInt(2, 2);

                try (ResultSet resultSet = statement.executeQuery()) {
                    if (resultSet.next()) {
                        int sum = resultSet.getInt("sum");
                        response.setStatusCode(200);
                        response.setBody("The selected sum is: " + sum);
                    }
                }
            }

        } catch (Exception e) {
            response.setStatusCode(500);
            response.setBody("Error: " + e.getMessage());
        }
```

```
        return response;
    }

    private String createAuthToken() {
        // Create RDS Data Service client
        RdsDataClient rdsDataClient = RdsDataClient.builder()
                .region(Region.of(System.getenv("AWS_REGION")))
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        // Define authentication request
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
                .resourceArn(System.getenv("ProxyHostName"))
                .secretArn(System.getenv("DBUserName"))
                .database(System.getenv("DBName"))
                .sql("SELECT 'RDS IAM Authentication'")
                .build();

        // Execute request and obtain authentication token
        ExecuteStatementResponse response =
 rdsDataClient.executeStatement(request);
        Field tokenField = response.records().get(0).get(0);

        return tokenField.stringValue();
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;
```

```
}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

使用 TypeScript 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

// RDS settings
// Using '!' (non-null assertion operator) to tell the TypeScript compiler that
 the DB settings are not null or undefined,
const proxy_host_name = process.env.PROXY_HOST_NAME!
const port = parseInt(process.env.PORT!)
const db_name = process.env.DB_NAME!
const db_user_name = process.env.DB_USER_NAME!
const aws_region = process.env.AWS_REGION!


async function createAuthToken(): Promise<string> {

    // Create RDS Signer object
    const signer = new Signer({
        hostname: proxy_host_name,
        port: port,
        region: aws_region,
        username: db_user_name
    });

    // Request authorization token from RDS, specifying the username
    const token = await signer.getAuthToken();
    return token;
```

```
}

async function dbOps(): Promise<mysql.QueryResult | undefined> {
    try {
        // Obtain auth token
        const token = await createAuthToken();
        const conn = await mysql.createConnection({
            host: proxy_host_name,
            user: db_user_name,
            password: token,
            database: db_name,
            ssl: 'Amazon RDS' // Ensure you have the CA bundle for SSL connection
        });
        const [rows, fields] = await conn.execute('SELECT ? + ? AS sum', [3, 2]);
        console.log('result:', rows);
        return rows;
    }
    catch (err) {
        console.log(err);
    }
}

export const lambdaHandler = async (event: any): Promise<{ statusCode: number;
 body: string }> => {
    // Execute database flow
    const result = await dbOps();

    // Return error is result is undefined
    if (result == undefined)
        return {
            statusCode: 500,
            body: JSON.stringify(`Error with connection to DB host`)
        }

    // Return result
    return {
        statusCode: 200,
        body: JSON.stringify(`The selected sum is: ${result[0].sum}`)
    };
};
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
<?php
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;
use Aws\Rds\AuthTokenGenerator;
use Aws\Credentials\CredentialProvider;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }


    private function getAuthToken(): string {
        // Define connection authentication parameters
        $dbConnection = [
            'hostname' => getenv('DB_HOSTNAME'),
            'port' => getenv('DB_PORT'),
            'username' => getenv('DB_USERNAME'),
            'region' => getenv('AWS_REGION'),
        ];
```

```
        // Create RDS AuthTokenGenerator object
        $generator = new
AuthTokenGenerator(CredentialProvider::defaultProvider());

        // Request authorization token from RDS, specifying the username
        return $generator->createToken(
            $dbConnection['hostname'] . ':' . $dbConnection['port'],
            $dbConnection['region'],
            $dbConnection['username']
        );
    }

    private function getQueryResults() {
        // Obtain auth token
        $token = $this->getAuthToken();

        // Define connection configuration
        $connectionConfig = [
            'host' => getenv('DB_HOSTNAME'),
            'user' => getenv('DB_USERNAME'),
            'password' => $token,
            'database' => getenv('DB_NAME'),
        ];

        // Create the connection to the DB
        $conn = new PDO(

"mysql:host={$connectionConfig['host']};dbname={$connectionConfig['database']}",
            $connectionConfig['user'],
            $connectionConfig['password'],
            [
                PDO::MYSQL_ATTR_SSL_CA => '/path/to/rds-ca-2019-root.pem',
                PDO::MYSQL_ATTR_SSL_VERIFY_SERVER_CERT => true,
            ]
        );

        // Obtain the result of the query
        $stmt = $conn->prepare('SELECT ?+? AS sum');
        $stmt->execute([3, 2]);

        return $stmt->fetch(PDO::FETCH_ASSOC);
    }
```

```
    /**
     * @param mixed $event
     * @param Context $context
     * @return array
     */
    public function handle(mixed $event, Context $context): array
    {
        $this->logger->info("Processing query");

        // Execute database flow
        $result = $this->getQueryResults();

        return [
            'sum' => $result['sum']
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
import json
import os
import boto3
import pymysql

# RDS settings
proxy_host_name = os.environ['PROXY_HOST_NAME']
port = int(os.environ['PORT'])
db_name = os.environ['DB_NAME']
```

```
db_user_name = os.environ['DB_USER_NAME']
aws_region = os.environ['AWS_REGION']


# Fetch RDS Auth Token
def get_auth_token():
    client = boto3.client('rds')
    token = client.generate_db_auth_token(
        DBHostname=proxy_host_name,
        Port=port
        DBUsername=db_user_name
        Region=aws_region
    )
    return token

def lambda_handler(event, context):
    token = get_auth_token()
    try:
        connection = pymysql.connect(
            host=proxy_host_name,
            user=db_user_name,
            password=token,
            db=db_name,
            port=port,
            ssl={'ca': 'Amazon RDS'}  # Ensure you have the CA bundle for SSL
 connection
        )

        with connection.cursor() as cursor:
            cursor.execute('SELECT %s + %s AS sum', (3, 2))
            result = cursor.fetchone()

        return result

    except Exception as e:
        return (f"Error: {str(e)}")  # Return an error message if an exception
 occurs
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
# Ruby code here.

require 'aws-sdk-rds'
require 'json'
require 'mysql2'

def lambda_handler(event:, context:)
  endpoint = ENV['DBEndpoint'] # Add the endpoint without https"
  port = ENV['Port']           # 3306
  user = ENV['DBUser']
  region = ENV['DBRegion']     # 'us-east-1'
  db_name = ENV['DBName']

  credentials = Aws::Credentials.new(
    ENV['AWS_ACCESS_KEY_ID'],
    ENV['AWS_SECRET_ACCESS_KEY'],
    ENV['AWS_SESSION_TOKEN']
  )
  rds_client = Aws::RDS::AuthTokenGenerator.new(
    region: region,
    credentials: credentials
  )

  token = rds_client.auth_token(
    endpoint: endpoint+ ':' + port,
    user_name: user,
    region: region
  )

  begin
    conn = Mysql2::Client.new(
```

```
      host: endpoint,
      username: user,
      password: token,
      port: port,
      database: db_name,
      sslca: '/var/task/global-bundle.pem',
      sslverify: true,
      enable_cleartext_plugin: true
    )
    a = 3
    b = 2
    result = conn.query("SELECT #{a} + #{b} AS sum").first['sum']
    puts result
    conn.close
    {
      statusCode: 200,
      body: result.to_json
    }
  rescue => e
    puts "Database connection failed due to #{e}"
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
use aws_config::BehaviorVersion;
use aws_credential_types::provider::ProvideCredentials;
use aws_sigv4::{
    http_request::{sign, SignableBody, SignableRequest, SigningSettings},
    sign::v4,
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
```

```
use serde_json::{json, Value};
use sqlx::postgres::PgConnectOptions;
use std::env;
use std::time::{Duration, SystemTime};

const RDS_CERTS: &[u8] = include_bytes!("global-bundle.pem");

async fn generate_rds_iam_token(
    db_hostname: &str,
    port: u16,
    db_username: &str,
) -> Result<String, Error> {
    let config = aws_config::load_defaults(BehaviorVersion::v2024_03_28()).await;

    let credentials = config
        .credentials_provider()
        .expect("no credentials provider found")
        .provide_credentials()
        .await
        .expect("unable to load credentials");
    let identity = credentials.into();
    let region = config.region().unwrap().to_string();

    let mut signing_settings = SigningSettings::default();
    signing_settings.expires_in = Some(Duration::from_secs(900));
    signing_settings.signature_location =
 aws_sigv4::http_request::SignatureLocation::QueryParams;

    let signing_params = v4::SigningParams::builder()
        .identity(&identity)
        .region(&region)
        .name("rds-db")
        .time(SystemTime::now())
        .settings(signing_settings)
        .build()?;

    let url = format!(
        "https://{db_hostname}:{port}/?Action=connect&DBUser={db_user}",
        db_hostname = db_hostname,
        port = port,
        db_user = db_username
    );

    let signable_request =
```

```
        SignableRequest::new("GET", &url, std::iter::empty(),
 SignableBody::Bytes(&[]))
            .expect("signable request");

    let (signing_instructions, _signature) =
        sign(signable_request, &signing_params.into())?.into_parts();

    let mut url = url::Url::parse(&url).unwrap();
    for (name, value) in signing_instructions.params() {
        url.query_pairs_mut().append_pair(name, &value);
    }

    let response = url.to_string().split_off("https://".len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(handler)).await
}

async fn handler(_event: LambdaEvent<Value>) -> Result<Value, Error> {
    let db_host = env::var("DB_HOSTNAME").expect("DB_HOSTNAME must be set");
    let db_port = env::var("DB_PORT")
        .expect("DB_PORT must be set")
        .parse::<u16>()
        .expect("PORT must be a valid number");
    let db_name = env::var("DB_NAME").expect("DB_NAME must be set");
    let db_user_name = env::var("DB_USERNAME").expect("DB_USERNAME must be set");

    let token = generate_rds_iam_token(&db_host, db_port, &db_user_name).await?;

    let opts = PgConnectOptions::new()
        .host(&db_host)
        .port(db_port)
        .username(&db_user_name)
        .password(&token)
        .database(&db_name)
        .ssl_root_cert_from_pem(RDS_CERTS.to_vec())
        .ssl_mode(sqlx::postgres::PgSslMode::Require);

    let pool = sqlx::postgres::PgPoolOptions::new()
        .connect_with(opts)
```

```
        .await?;

    let result: i32 = sqlx::query_scalar("SELECT $1 + $2")
        .bind(3)
        .bind(2)
        .fetch_one(&pool)
        .await?;

    println!("Result: {:?}", result);

    Ok(json!({
        "statusCode": 200,
        "content-type": "text/plain",
        "body": format!("The selected sum is: {result}")
    }))
}
```

處理來自 Amazon RDS 的事件通知

您可以使用 Lambda 來處理 Amazon RDS 資料庫的事件通知。Amazon RDS 會將通知傳送到 Amazon Simple Notification Service (Amazon SNS) 主題，您可以進行設定，透過該主題叫用 Lambda 函數。Amazon SNS 會將來自 Amazon RDS 的訊息包裝在自己的事件文件中，並將其傳送到函數。

如需設定 Amazon RDS 資料庫以傳送通知的詳細資訊，請參閱使用 Amazon RDS 事件通知。

Example Amazon SNS 事件中的 Amazon RDS 訊息

```
{
        "Records": [
          {
            "EventVersion": "1.0",
            "EventSubscriptionArn": "arn:aws:sns:us-east-2:123456789012:rds-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
            "EventSource": "aws:sns",
            "Sns": {
              "SignatureVersion": "1",
              "Timestamp": "2023-01-02T12:45:07.000Z",
              "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi
+tE/1+82j...65r==",
              "SigningCertUrl": "https://sns.us-east-2.amazonaws.com/
SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
```

```
            "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
            "Message": "{\"Event Source\":\"db-instance\",\"Event Time\":\"2023-01-02
 12:45:06.000\",\"Identifier Link\":\"https://console.aws.amazon.com/rds/home?
region=eu-west-1#dbinstance:id=dbinstanceid\",\"Source ID\":\"dbinstanceid\",\"Event ID
\":\"http://docs.amazonwebservices.com/AmazonRDS/latest/UserGuide/USER_Events.html#RDS-
EVENT-0002\",\"Event Message\":\"Finished DB Instance backup\"}",
            "MessageAttributes": {},
            "Type": "Notification",
            "UnsubscribeUrl": "https://sns.us-east-2.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-2:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
            "TopicArn":"arn:aws:sns:us-east-2:123456789012:sns-lambda",
            "Subject": "RDS Notification Message"
          }
        }
      ]
    }
```

完成 Lambda 和 Amazon RDS 教學課程

- 使用 Lambda 函數來存取 Amazon RDS：在《Amazon RDS 誰用著指南》中，學習如何使用 Lambda 函數並透過 Amazon RDS Proxy 將資料寫入 Amazon RDS 資料庫。每當新增訊息，您的 Lambda 函數將從 Amazon SQS 佇列中讀取記錄，然後將新項目寫入資料庫中的資料表。

為基於 Lambda 的應用程式選擇資料庫服務

許多無伺服器應用程式需要存放和擷取資料。 AWS 提供多種資料庫選項，可與 Lambda 函數搭配使用。最熱門的兩個選擇是 NoSQL 資料庫服務 Amazon DynamoDB，以及傳統的關聯式資料庫解決方案 Amazon RDS。下列各節介紹了這兩種服務與 Lambda 搭配使用時的主要差異，並協助您為無伺服器應用程式選擇合適的資料庫服務。

若要進一步了解 提供的其他資料庫服務 AWS，以及更普遍地了解其使用案例和權衡，請參閱選擇 AWS 資料庫服務。所有 AWS 資料庫服務都與 Lambda 相容，但並非所有資料庫服務都適合自身特定使用案例。

選取與 Lambda 搭配使用的資料庫服務時，您有哪些選擇？

AWS 提供多個資料庫服務。對於無伺服器應用程式，最熱門的兩個選擇是 DynamoDB 和 Amazon RDS。

- DynamoDB 是一種全受管 NoSQL 資料庫服務，針對無伺服器應用程式進行了最佳化。該服務可在任何規模下提供無縫擴展且穩定的個位數毫秒級效能。
- Amazon RDS 是一種受管關聯式資料庫服務，支援包括 MySQL 與 PostgreSQL 在內的多種資料庫引擎。該服務透過受管基礎結構提供熟悉的 SQL 功能。

您已知道自己需求時的建議

如果您已經清楚自己的需求，以下是基本建議：

對於需要穩定低延遲效能、自動擴展且不涉及複雜聯結或交易的無伺服器應用程式，建議選擇 DynamoDB。由於其無伺服器特性，該服務特別適合基於 Lambda 的應用程式。

若需複雜的 SQL 查詢、聯結操作，或是現有應用程式已使用關聯式資料庫時，Amazon RDS 是更佳選擇。但請注意，將 Lambda 函式連線至 Amazon RDS 需要額外的組態，並且可能會影響冷啟動時間。

選取資料庫服務時應考量的事項

為 Lambda 應用程式選取 DynamoDB 或是 Amazon RDS 時，建議考量下列因素：

- 連線管理與冷啟動
- 資料存取模式
- 查詢複雜性
- 資料一致性需求
- 擴展特性
- 成本模型

透過了解這些因素，您可以選取最符合自身特定使用案例需求的選項。

連線管理與冷啟動

- DynamoDB 對所有操作均使用 HTTP API。Lambda 函式無需維護連線即可發出即時請求，具備更佳的冷啟動效能。每個請求都會使用 AWS 登入資料進行身分驗證，而不會產生連線額外負荷。
- 由於 Amazon RDS 使用的是傳統資料庫連線，因此需要管理連線集區。這可能會影響冷啟動，因為新的 Lambda 執行個體需要建立連線。您需實作連線集區策略，並且可能要使用 Amazon RDS Proxy 來有效管理連線。請注意，使用 Amazon RDS Proxy 會額外產生費用。

資料存取模式

- DynamoDB 最適用於已知的存取模式和單一資料表設計。其非常適合需要根據主索引鍵或次要索引，以穩定低延遲方式存取資料的 Lambda 應用程式。
- Amazon RDS 具備彈性，能因應複雜查詢與多變的存取模式。當您的 Lambda 函式需要跨多個資料表執行獨特的客製化查詢或複雜的跨表聯結時，其適用性更佳。

查詢複雜性

- DynamoDB 在基於鍵值的簡易操作及預先定義的存取模式方面表現出色。複雜的查詢必須圍繞索引結構進行設計，並且必須在應用程式程式碼中處理聯結。
- Amazon RDS 支援包含聯結、子查詢與彙總運算的複雜 SQL 查詢。當需要複雜的資料操作時，此優勢可以簡化您的 Lambda 函式程式碼。

資料一致性需求

- DynamoDB 提供最終一致性與強一致性選項，其中強一致性適用於單一項目讀取。雖然支援交易功能，但存在部分限制。
- Amazon RDS 提供完整的原子性、一致性、隔離性與持久性 (ACID) 合規保障，支援複雜交易。如果您的 Lambda 函式需要複雜的交易或跨多筆記錄的強一致性，Amazon RDS 可能更加合適。

擴展特性

- DynamoDB 會根據您的工作負載自動擴展。其無需預先佈建，即可處理 Lambda 函式中的流量突增情況。您可以使用隨需容量模式，僅需為所使用的容量付費，完全符合 Lambda 的擴展模型。
- Amazon RDS 會根據您選擇的執行個體規格，提供固定的容量。如果多個 Lambda 函式同時嘗試連線，則可能會超過連線配額。您需謹慎管理連線集區，並可能要實作重試邏輯。

成本模型

- DynamoDB 的定價模式與無伺服器應用程式高度契合。若使用隨需容量，您只需為 Lambda 函式執行的實際讀取與寫入作業付費。閒置時間不收取任何費用。
- Amazon RDS 會針對執行中的執行個體收費，無論用量如何。對於無伺服器應用程式中典型的零星工作負載，這可能較不具成本效益。不過，對於輸送量高且用量穩定的工作負載，其可能更經濟實惠。

開始使用所選資料庫服務

您已了解在 DynamoDB 和 Amazon RDS 之間進行選擇的標準，以及二者之間的主要差異。接下來，您可以選取最符合自身需求的選項，並使用下列資源來協助自己開始使用所選的資料庫服務。

DynamoDB

透過下列資源開始使用 DynamoDB

- 如需 DynamoDB 服務的簡介，請參閱 Amazon DynamoDB Developer Guide 中的 What is DynamoDB? 部分。
- 請依照搭配 API Gateway 使用 Lambda 教學課程，了解如何透過 Lambda 函式在回應 API 請求時，對 DynamoDB 資料表執行 CRUD 操作的範例。
- 請參閱《Amazon DynamoDB 開發人員指南》中的使用 DynamoDB 和 AWS SDKs 進行程式設計，以進一步了解如何使用其中一個 SDK 從 Lambda 函數內存取 AWS SDKs DynamoDB。DynamoDB

Amazon RDS

透過下列資源開始使用 Amazon RDS

- 如需 Amazon RDS 服務簡介，請參閱 Amazon Relational Database Service User Guide 中的 What is Amazon Relational Database Service (Amazon RDS)? 部分。
- 請依照 Amazon Relational Database Service User Guide 中的教學課程 Using a Lambda function to access an Amazon RDS database 執行操作。
- 請參閱 the section called “RDS”，了解有關搭配 Amazon RDS 使用 Lambda 的詳細資訊。

使用 Lambda 處理 Amazon S3 事件通知

您可以使用 Lambda 來處理來自 Amazon Simple Storage Service 的事件通知。建立或刪除物件時，Amazon S3 可以將事件傳送至 Lambda 函數。您在儲存貯體上設定通知設定，並授予 Amazon S3 許可以在函數以資源為基礎的許可政策上叫用函數。

> **Warning**
>
> 如果 Lambda 函數使用的是觸發的相同儲存貯體，這可能會造成函數在迴圈中執行。例如，如果儲存貯體在物件每次上傳時都觸發函數，且該函數會將物件上傳至儲存貯體，則函數會間接地觸發本身。若要避免此狀況，請使用兩個儲存貯體，或將觸發設定為僅套用至傳入物件所用的字首。

Amazon S3 使用包含物件詳細資訊的事件以非同步方式叫用您的函數。以下範例顯示當部署套裝服務上傳至 Amazon S3 時，Amazon S3 傳送的事件。

Example Amazon S3 通知事件

```
{
  "Records": [
    {
      "eventVersion": "2.1",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-2",
      "eventTime": "2019-09-03T19:37:27.192Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "AWS:AIDAINPONIXQXHT3IKHL2"
      },
      "requestParameters": {
        "sourceIPAddress": "205.255.255.255"
      },
      "responseElements": {
        "x-amz-request-id": "D82B88E5F771F645",
        "x-amz-id-2":
 "vlR7PnpV2Ce81l0PRw6jlUpck7Jo5ZsQjryTjKlc5aLWGVHPZLj5NeC6qMa0emYBDXOo6QBU0Wo="
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "828aa6fc-f7b5-4305-8584-487c791949c1",
```

```
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "A3I5XTEXAMAI3E"
          },
          "arn": "arn:aws:s3:::lambda-artifacts-deafc19498e3f2df"
        },
        "object": {
          "key": "b21b84d653bb07b05b1e6b33684dc11b",
          "size": 1305107,
          "eTag": "b21b84d653bb07b05b1e6b33684dc11b",
          "sequencer": "0C0F6F405D6ED209E1"
        }
      }
    }
  ]
}
```

若要叫用您的函數，Amazon S3 需要該函數以資源為基礎政策的許可。當您在 Lambda 主控台中設定 Amazon S3 觸發條件時，主控台會修改以資源為基礎的政策，讓 Amazon S3 在儲存貯體名稱與帳戶 ID 相符時叫用該函數。如果您在 Amazon S3 中設定通知，您要使用 Lambda API 更新政策。您也可以使用 Lambda API 將許可授予另一個帳戶，或將許可限制為指定的別名。

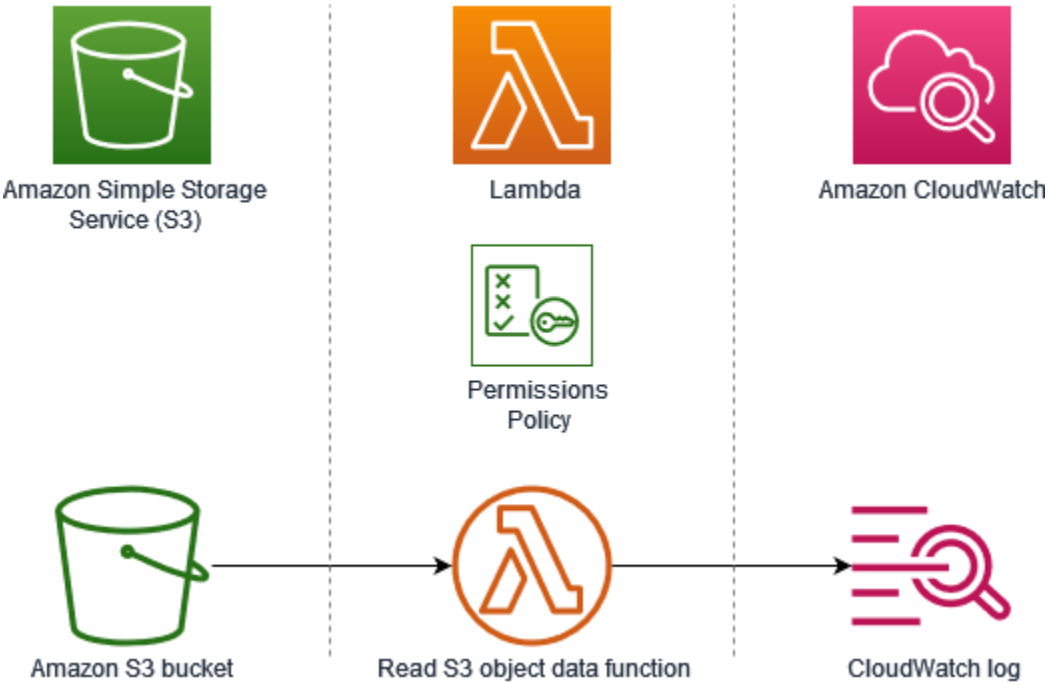
如果您的函數使用 AWS SDK 來管理 Amazon S3 資源，則其執行角色也需要 Amazon S3 許可。

主題

- 教學課程：使用 Amazon S3 觸發條件調用 Lambda 函數
- 教學課程：使用 Amazon S3 觸發條件建立縮圖影像

教學課程：使用 Amazon S3 觸發條件調用 Lambda 函數

在本教學課程中，您將使用主控台建立 Lambda 函數，並設定 Amazon Simple Storage Service (Amazon S3) 儲存貯體的觸發條件。每當有物件新增至 Amazon S3 儲存貯體，該函數都會執行，然後將物件類型輸出至 Amazon CloudWatch Logs。

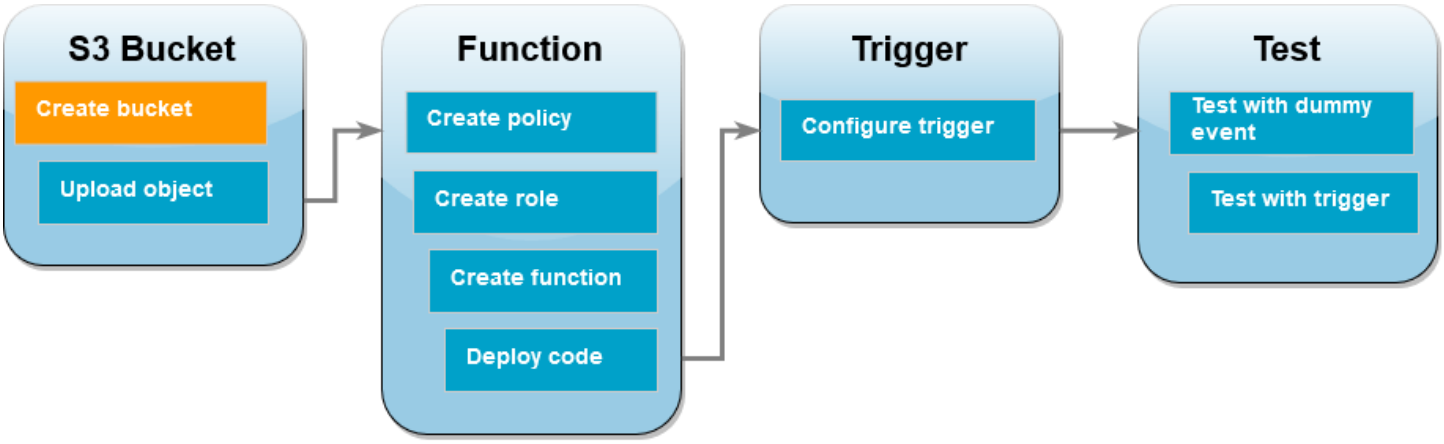


本教學課程示範如何。

1. 建立 Amazon S3 儲存貯體。
2. 建立一個 Lambda 函數，用於傳回 Amazon S3 儲存貯體物件的物件類型。
3. 設定一個 Lambda 觸發條件，用於在有物件上傳至儲存貯體時調用函數。
4. 先使用虛擬事件測試函數，再使用觸發條件進行測試。

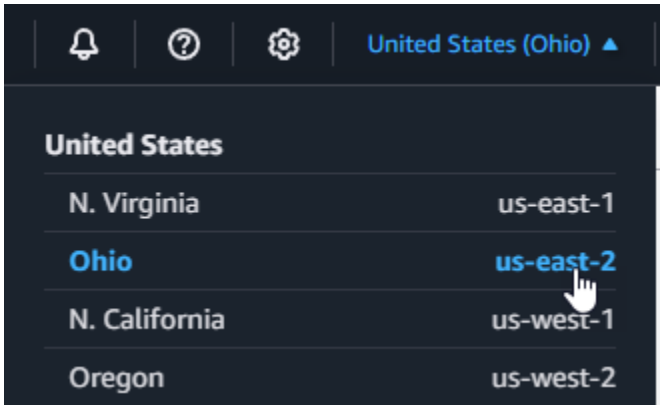
完成這些步驟後，您將了解如何設定 Lambda 函數，讓函數在 Amazon S3 儲存貯體中有物件新增或刪除時執行。您只能使用 AWS 管理主控台 來完成本教學課程。

建立 Amazon S3 儲存貯體



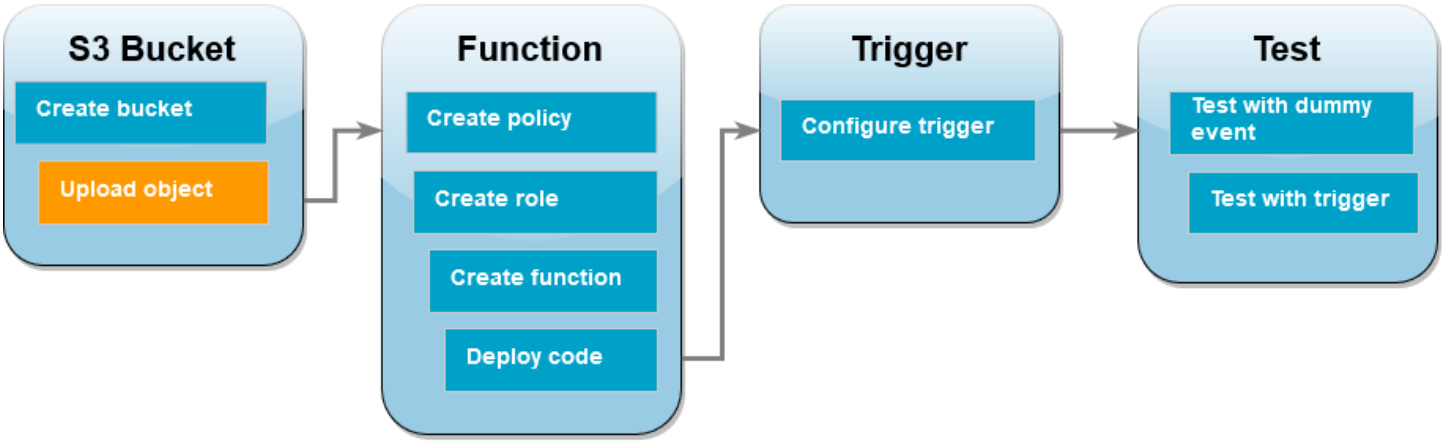
建立 Amazon S3 儲存貯體

1. 開啟 Amazon S3 主控台，然後選取一般用途儲存貯體頁面。
2. 選取最 AWS 區域 接近您地理位置的 。可使用螢幕頂端的下拉式清單來變更區域。在本教學課程稍後的階段，您必須在同一區域中建立 Lambda 函數。



3. 選擇建立儲存貯體。
4. 在 General configuration (一般組態) 下，執行下列動作：
   a. 在儲存貯體類型欄位中，選取一般用途。
   b. 在儲存貯體名稱欄位中，輸入符合 Amazon S3 儲存貯體命名規則的全域唯一名稱。儲存貯體名稱只能包含小寫字母、數字、句點 (.) 和連字號 (-)。
5. 其他所有選項維持設為預設值，然後選擇建立儲存貯體。

將測試物件上傳至儲存貯體



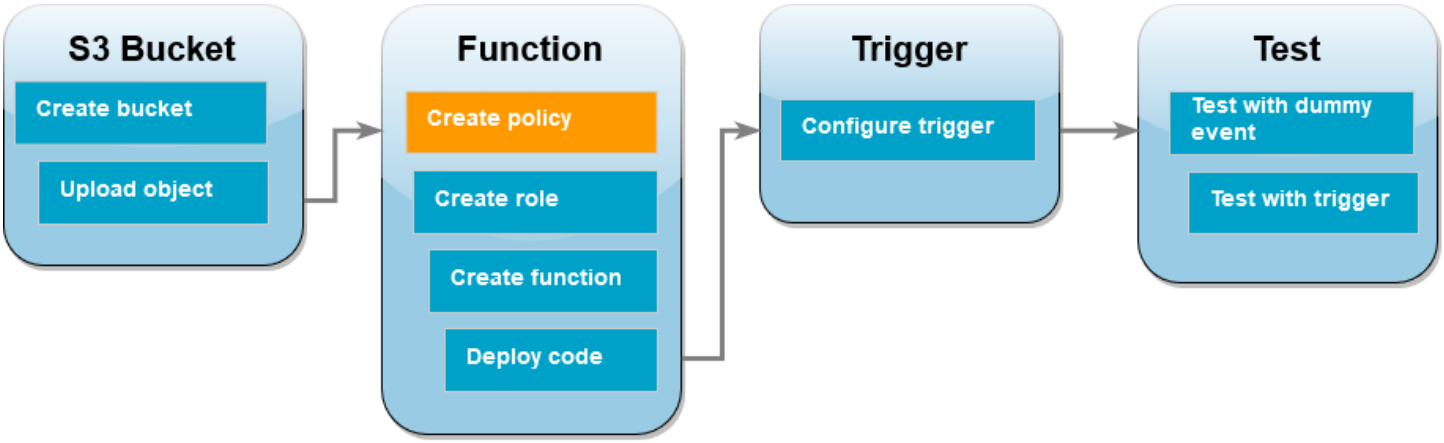
上傳測試物件

1. 開啟 Amazon S3 主控台的儲存貯體頁面，然後選擇您在上一個步驟中建立的儲存貯體。

2. 選擇上傳。
3. 選擇新增檔案，然後選取要上傳的物件。您可以選取任何檔案 (例如 HappyFace.jpg)。
4. 選擇開啟，然後選擇上傳。

在本教學課程的稍後部分，您將使用此物件測試 Lambda 函數。

建立許可政策



建立許可政策，以允許 Lambda 從 Amazon S3 儲存貯體取得物件，以及寫入 Amazon CloudWatch Logs。

建立政策

1. 開啟 IAM 主控台中的政策頁面。
2. 選擇建立政策。
3. 選擇 JSON 索引標籤，然後將下列政策貼到 JSON 編輯器。

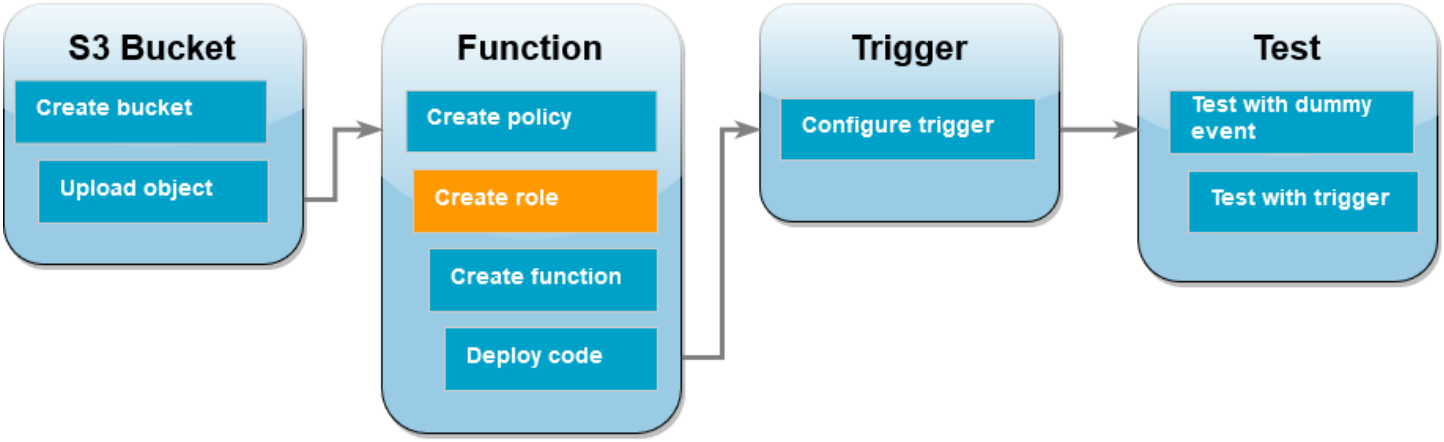
JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
```

```
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. 選擇下一步：標籤。
5. 選擇下一步：檢閱。
6. 在檢閱政策下，針對政策名稱，輸入 **s3-trigger-tutorial**。
7. 選擇建立政策。

建立執行角色



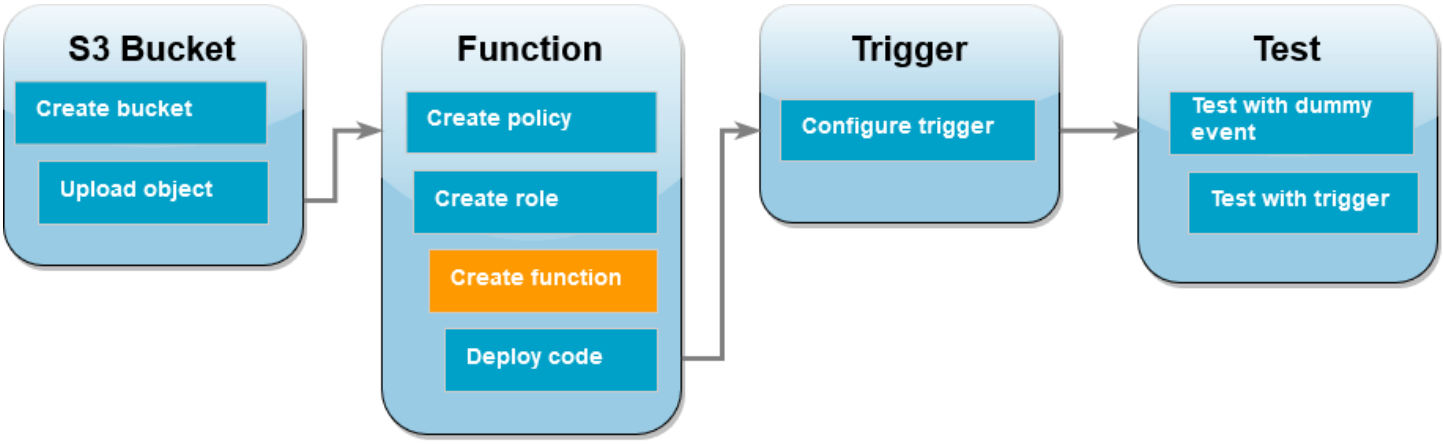
執行角色是授予 Lambda 函數存取 AWS 服務 和資源許可的 AWS Identity and Access Management (IAM) 角色。在此步驟中，使用您在上一個步驟中建立的許可政策來建立執行角色。

建立執行角色並附加自訂許可政策

1. 開啟 IAM 主控台中的角色頁面。
2. 選擇 建立角色。
3. 信任的實體類型請選擇 AWS 服務，使用案例則選擇 Lambda。
4. 選擇下一步。

5. 在政策搜尋方塊中，輸入 **s3-trigger-tutorial**。
6. 在搜尋結果中，選取您建立的政策 (s3-trigger-tutorial)，然後選擇下一步。
7. 在角色詳細資料底下，角色名稱請輸入 **lambda-s3-trigger-role**，然後選擇建立角色。

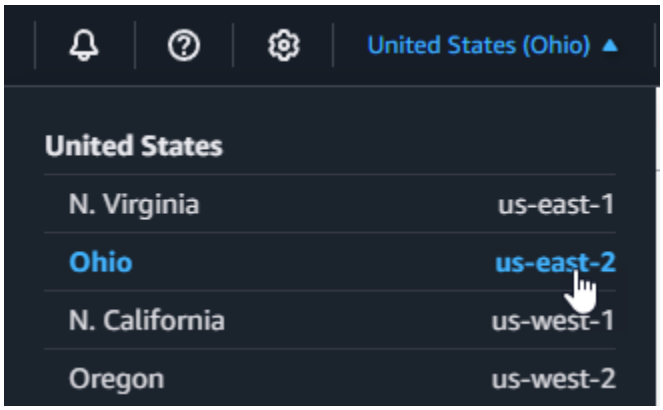
建立 Lambda 函式



使用 Python 3.14 執行時間在主控台中建立 Lambda 函數。

建立 Lambda 函數

1. 開啟 Lambda 主控台中的函數頁面。
2. 請確定您使用 AWS 區域 與建立 Amazon S3 儲存貯體相同的 。您可使用螢幕頂端的下拉式清單來變更區域。



3. 選擇建立函數。
4. 選擇從頭開始撰寫
5. 在基本資訊下，請執行下列動作：

   a. 在函數名稱輸入 s3-trigger-tutorial

b. 針對執行期，選擇 Python 3.14。

c. 對於 Architecture (架構)，選擇 x86_64。

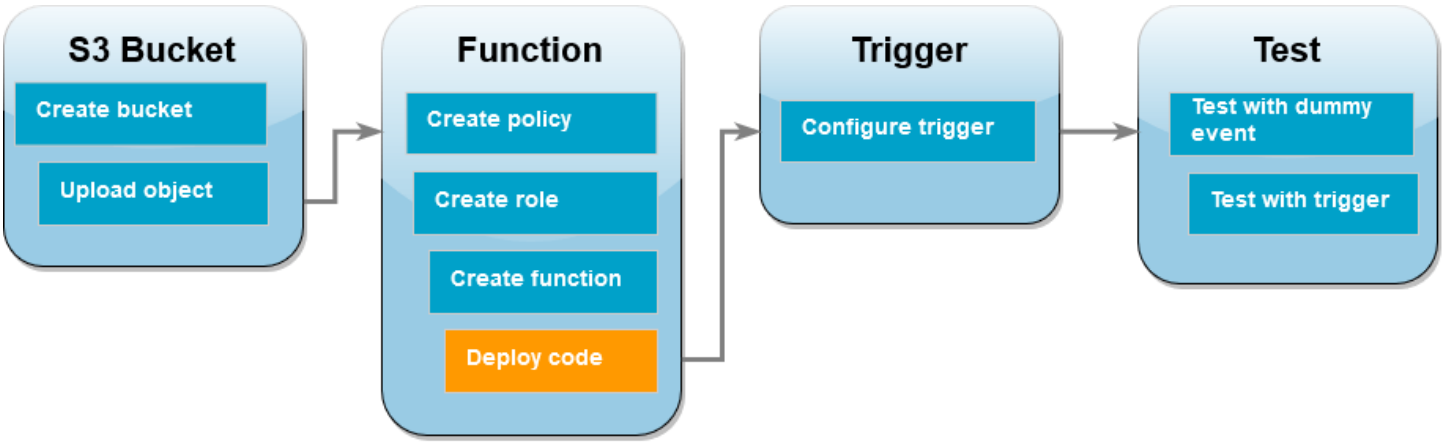
6. 在變更預設執行角色索引標籤中，執行下列操作：

a. 展開索引標籤，然後選擇使用現有角色。

b. 選擇您之前建立的 `lambda-s3-trigger-role`。

7. 選擇建立函數。

部署函數程式碼



本教學課程使用 Python 3.14 執行時間，但我們也提供其他執行時間的範例程式碼檔案。您可以在下列方塊中選取索引標籤，查看您感興趣的執行期程式碼。

該 Lambda 函數收到 Amazon S3 傳來的 `event` 參數後，會從該參數擷取上傳物件的金鑰名稱以及儲存貯體的名稱。然後，函數會使用 中的 get_object 方法 適用於 Python (Boto3) 的 AWS SDK 擷取物件的中繼資料，包括上傳物件的內容類型 (MIME 類型）。

部署函數程式碼

1. 在以下方塊中選擇 Python 索引標籤，然後複製程式碼。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson

namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }
```

```
        public async Task<string> Handler(S3Event evt, ILambdaContext
 context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key =
 HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");

                var objectResult = await _s3Client.GetObjectAsync(bucket,
 key);

                context.Logger.LogLine($"Returning {objectResult.Key}");

                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
 {e.Message}");

                return string.Empty;
            }
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

func handler(ctx context.Context, s3Event events.S3Event) error {
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
   Bucket: &bucket,
   Key:    &key,
  })
```

```
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNo
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger =
 LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
          HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

          logger.info("Successfully retrieved " + srcBucket + "/" + srcKey +
 " of type " + headObject.contentType());

          return "Ok";
        } catch (Exception e) {
          throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String
 bucket, String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 來使用 S3 事件。

```
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

export const handler = async (event, context) => {

    // Get the object from the event and show its content type
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\
+/g, ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;

    } catch (err) {
        console.log(err);
        const message = `Error getting object ${key} from bucket ${bucket}.
 Make sure they exist and your bucket is in the same region as this
 function.`;
        console.log(message);
        throw new Error(message);
    }
};
```

使用 TypeScript 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> =>
 {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, ' '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來使用 S3 事件。

```
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';


class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());
```

```
            try {
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as
 this function.' . "\n";
                throw $e;
            }
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 S3 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')
```

```
def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']
['key'], encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
        print(e)
        print('Error getting object {} from bucket {}. Make sure they
 exist and your bucket is in the same region as this function.'.format(key,
 bucket))
        raise e
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來使用 S3 事件。

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'

def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
```

```
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']
['key'], Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they
 exist and your bucket is in the same region as this function."
    raise e
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

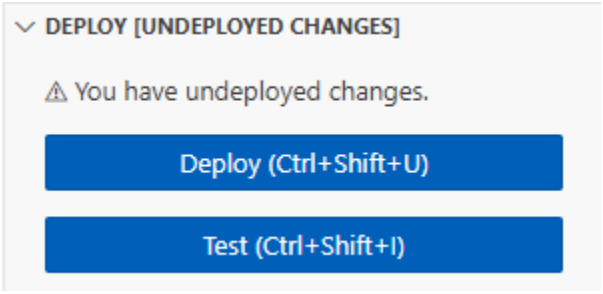
使用 Rust 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
#[tokio::main]
async fn main() -> Result<(), Error> {
```

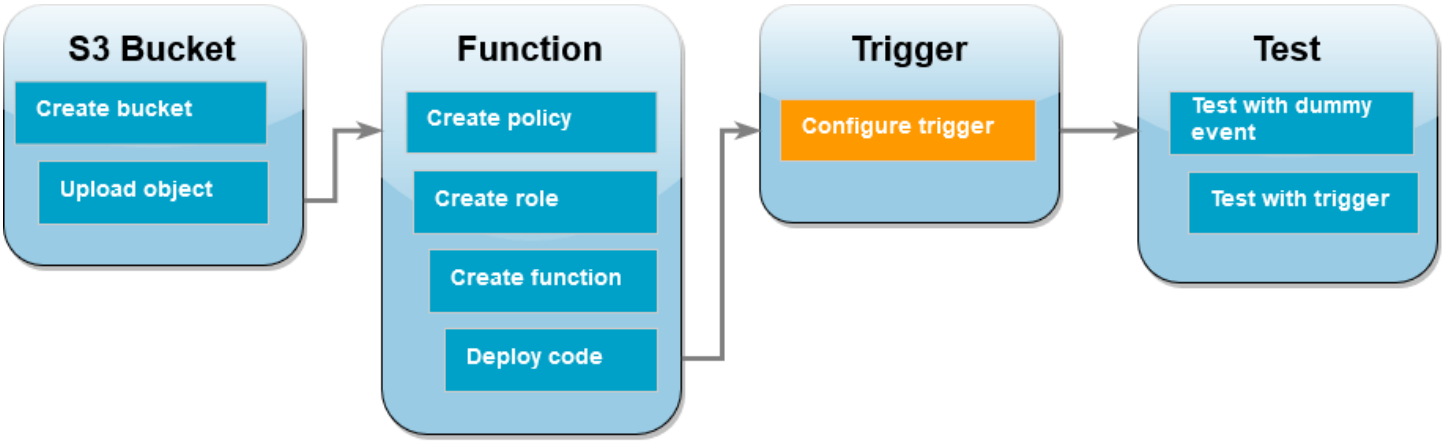
```
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}

async fn function_handler(
    s3_client: &Client,
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request
 from SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket =
 evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket name to
 exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object
 key to exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;

    match s3_get_object_result {
```

```
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
 contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
}
```

2. 在 Lambda 主控台的程式碼來源窗格中，將程式碼貼到程式碼編輯器中，取代 Lambda 建立的程式碼。
3. 在 DEPLOY 區段中，選擇部署以更新函數的程式碼：

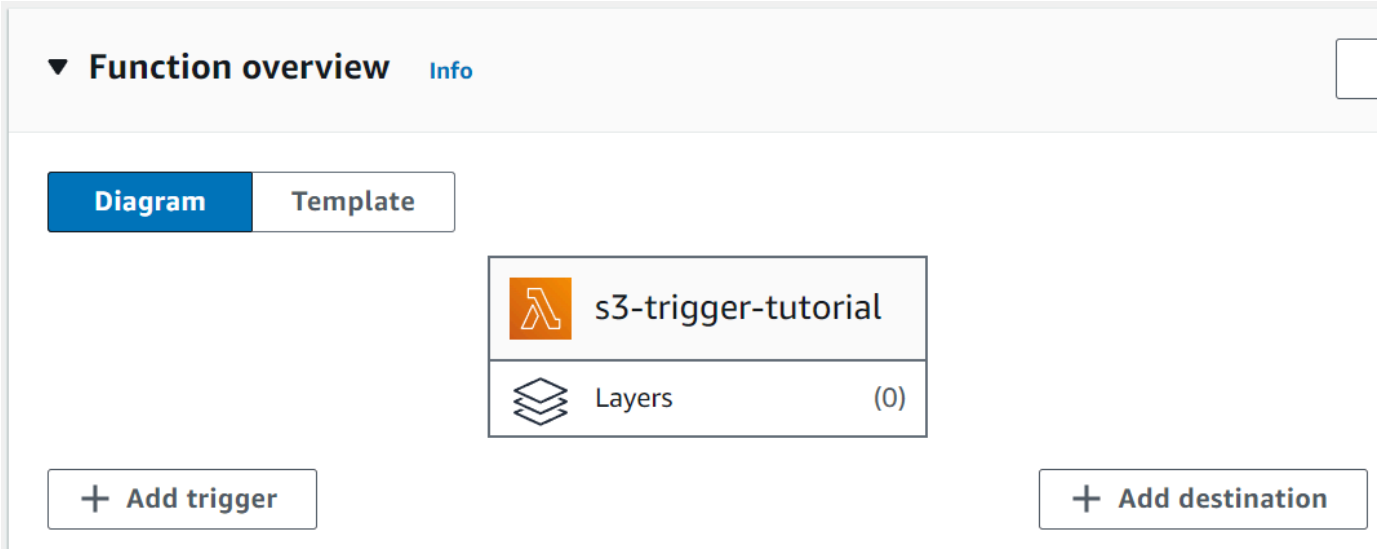


建立 Amazon S3 觸發條件



若要建立 Amazon S3 觸發條件

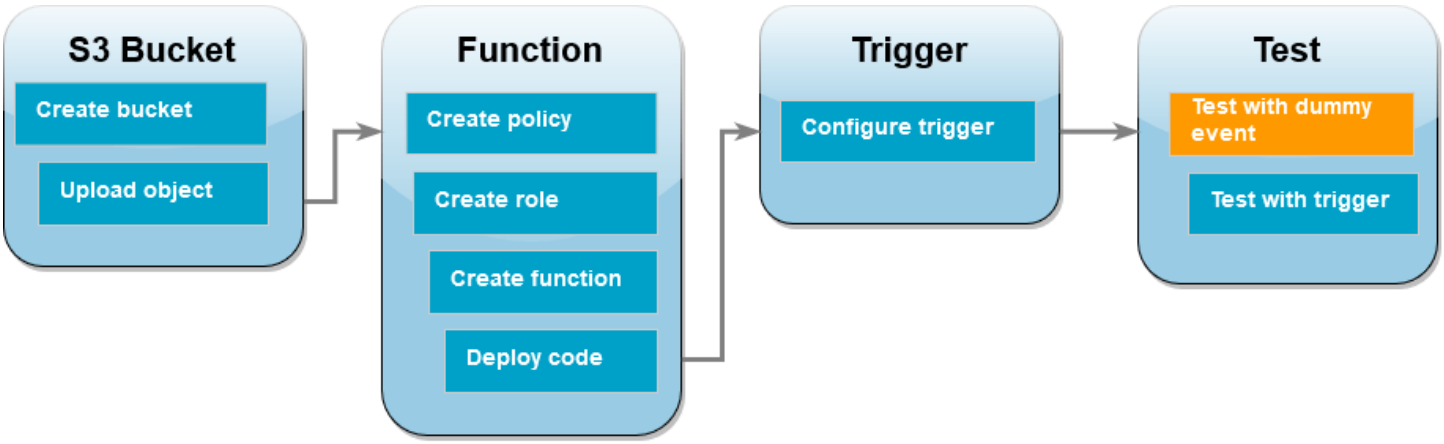
1. 在函數概觀窗格中，選擇新增觸發條件。



2. 選取 S3。
3. 在儲存貯體下，選取您在本教學課程中稍早建立的儲存貯體。
4. 在事件類型下，選取所有物件建立事件。
5. 在遞迴調用下，選取核取方塊，確認您了解不建議使用相同的 Amazon S3 儲存貯體進行輸入和輸出作業。
6. 選擇新增。

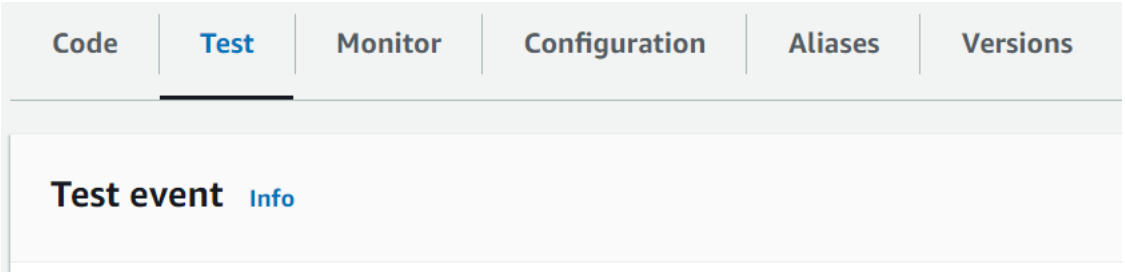
> **Note**
>
> 當您使用 Lambda 主控台為 Lambda 函數建立 Amazon S3 觸發條件時，Amazon S3 會在您指定的儲存貯體上設定事件通知。在設定此事件通知之前，Amazon S3 會執行一系列檢查，以確認該事件目的地存在，且具有所需的 IAM 政策。Amazon S3 也會對針對為該儲存貯體設定的任何其他事件通知執行這些測試。
>
> 由於此檢查的存在，如果儲存貯體先前已為不再存在的資源設定事件目的地，或為沒有必要許可政策的資源設定事件目的地，則 Amazon S3 將無法建立新的事件通知。您將看到以下錯誤訊息，指出無法建立觸發條件：
>
> ```
> An error occurred when creating the trigger: Unable to validate the following
>  destination configurations.
> ```
>
> 如果您先前已使用相同的儲存貯體為另一個 Lambda 函數設定觸發條件，且在之後刪除了相關函數或修改了其許可政策，則會顯示此錯誤。

使用虛擬事件來測試 Lambda 函數



使用虛擬事件來測試 Lambda 函數

1. 在函數的 Lambda 主控台頁面中，選擇測試索引標籤。



2. 事件名稱輸入 `MyTestEvent`。
3. 在事件 JSON 中，貼上以下測試事件。務必取代這些值：

   - 將 `us-east-1` 取代為您用來建立 Amazon S3 儲存貯體的區域。
   - 將 `amzn-s3-demo-bucket` 的兩個執行個體取代為您的 Amazon S3 儲存貯體名稱。
   - 將 `test%2FKey` 取代為您之前上傳到儲存貯體的測試物件名稱 (例如 `HappyFace.jpg`)。

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
```

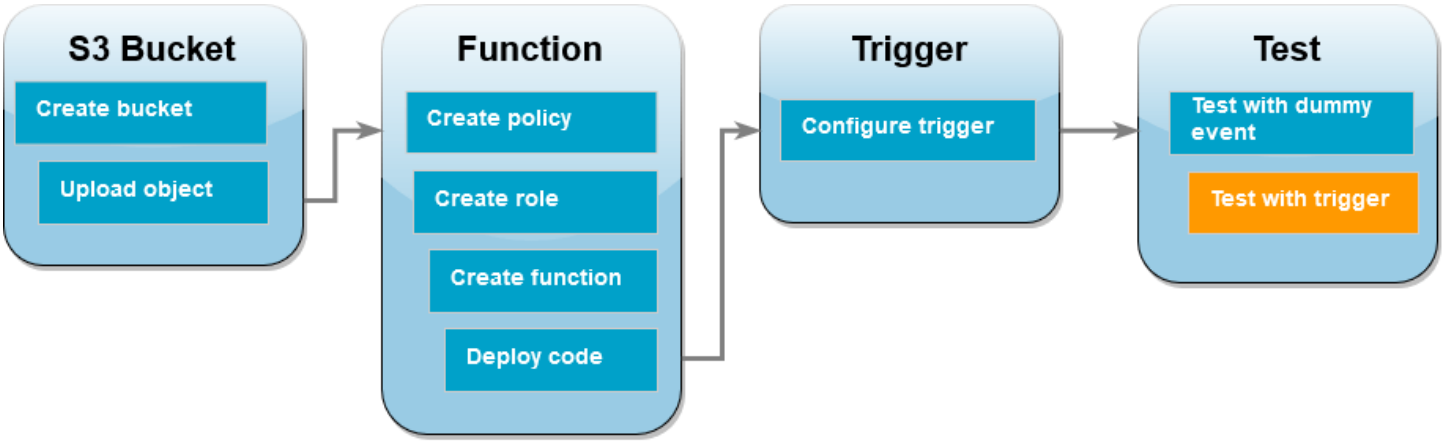
```
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

4. 選擇儲存。
5. 選擇測試。
6. 如果函數成功運作，您會在執行結果索引標籤中看到類似以下內容的輸出。

```
Response
"image/jpeg"

Function Logs
START RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6 Version: $LATEST
2021-02-18T21:40:59.280Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    INPUT
 BUCKET AND KEY:  { Bucket: 'amzn-s3-demo-bucket', Key: 'HappyFace.jpg' }
2021-02-18T21:41:00.215Z    12b3cae7-5f4e-415e-93e6-416b8f8b66e6    INFO    CONTENT
 TYPE: image/jpeg
```

```
END RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6
REPORT RequestId: 12b3cae7-5f4e-415e-93e6-416b8f8b66e6    Duration: 976.25 ms
 Billed Duration: 977 ms    Memory Size: 128 MB    Max Memory Used: 90 MB    Init
 Duration: 430.47 ms

Request ID
12b3cae7-5f4e-415e-93e6-416b8f8b66e6
```

使用 Amazon S3 觸發條件測試 Lambda 函數



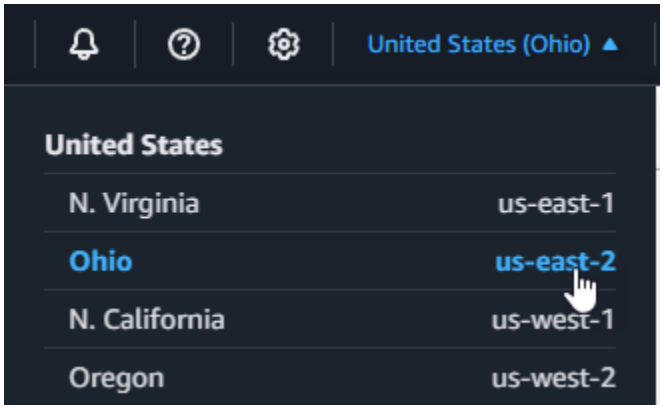
若要使用已設定的觸發條件來測試函數，需使用主控台將物件上傳至 Amazon S3 儲存貯體。若要確認 Lambda 函數如預期執行，請使用 CloudWatch Logs 來檢視函數的輸出。

將物件上傳至 Amazon S3 儲存貯體

1. 開啟 Amazon S3 主控台的儲存貯體頁面，然後選擇您稍早建立的儲存貯體。
2. 選擇上傳。
3. 選擇新增檔案，然後使用檔案選擇器選擇您要上傳的物件。此物件可以是您自選的任何檔案。
4. 選擇開啟，然後選擇上傳。

若要使用 CloudWatch Logs 驗證函數調用

1. 開啟 CloudWatch 主控台。
2. 請確定您在 AWS 區域 建立 Lambda 函數的相同 中工作。您可使用螢幕頂端的下拉式清單來變更區域。



3. 選擇日誌，然後選擇日誌群組。
4. 為函數 (`/aws/lambda/s3-trigger-tutorial`) 選擇日誌群組名稱。
5. 在日誌串流下，選擇最新的日誌串流。
6. 如果系統已正確調用函數以回應 Amazon S3 觸發條件，您會看到類似以下內容的輸出。您看到的 CONTENT TYPE 取決於您上傳到儲存貯體的檔案類型。

```
2022-05-09T23:17:28.702Z 0cae7f5a-b0af-4c73-8563-a3430333cc10 INFO CONTENT
 TYPE: image/jpeg
```

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。透過刪除您不再使用 AWS 的資源，您可以避免不必要的 費用 AWS 帳戶。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **`confirm`**，然後選擇刪除。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇刪除。

4. 在文字輸入欄位中輸入角色的名稱，然後選擇刪除。

刪除 S3 儲存貯體

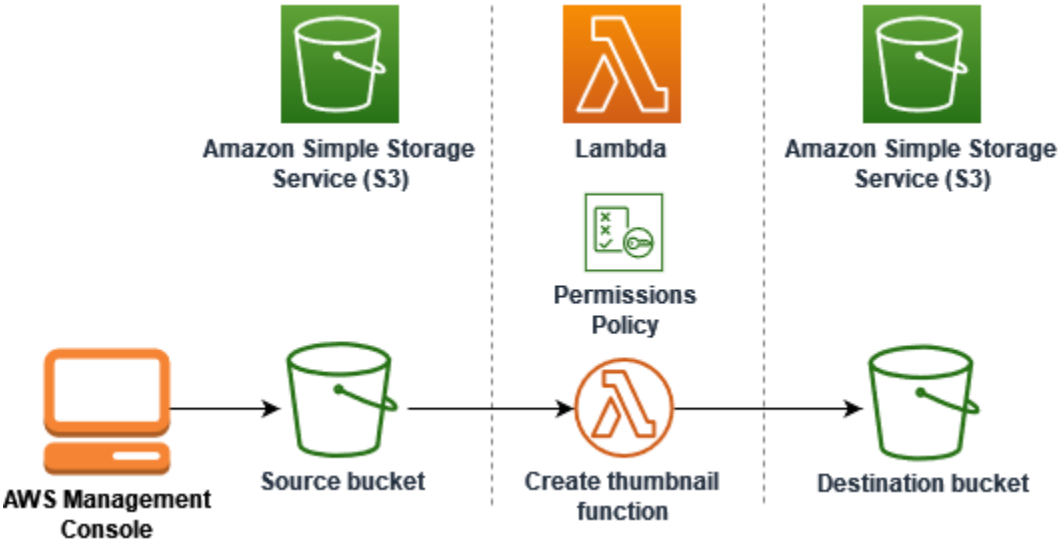
1. 開啟 Amazon S3 主控台。
2. 選擇您建立的儲存貯體。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入儲存貯體的名稱。
5. 選擇刪除儲存貯體。

後續步驟

在 教學課程：使用 Amazon S3 觸發條件建立縮圖影像 中，Amazon S3 觸發條件會調用函數，為上傳至 S3 儲存貯體的每個影像檔建立縮圖影像。本教學課程需要中等程度的 AWS 和 Lambda 網域知識。它示範如何使用 AWS Command Line Interface (AWS CLI) 建立資源，以及如何為函數及其相依性建立 .zip 檔案封存部署套件。

教學課程：使用 Amazon S3 觸發條件建立縮圖影像

在本教學課程中，您將建立和設定 Lambda 函數，它會調整新增至 Amazon Simple Storage Service (Amazon S3) 儲存貯體的映像。當您將映像檔案新增至儲存貯體時，Amazon S3 會調用 Lambda 函數。然後，函數會建立映像的縮圖版本，並將其輸出到不同 Amazon S3 儲存貯體。



請執行下列步驟以完成本教學課程：

1. 建立來源和目的地 Amazon S3 儲存貯體，並上傳範例映像。
2. 建立可調整映像大小並將縮圖輸出到 Amazon S3 儲存貯體的 Lambda 函數。
3. 設定一個 Lambda 觸發條件，在物件上傳至來源儲存貯體時調用函數。
4. 首先使用虛擬事件測試函數，然後透過將映像上傳到來源儲存貯體來測試函數。

完成這些步驟後，將了解如何使用 Lambda 在新增至 Amazon S3 儲存貯體的物件上執行檔案處理任務。您可以使用 AWS Command Line Interface (AWS CLI) 或 來完成本教學課程 AWS 管理主控台。

如果您正在尋找更簡單的範例來學習如何為 Lambda 設定 Amazon S3 觸發條件，則可以嘗試教學課程：使用 Amazon S3 觸發條件調用 Lambda 函數。

主題

- 先決條件
- 建立兩個 Amazon S3 儲存貯體
- 將測試映像上傳到來源儲存貯體
- 建立許可政策
- 建立執行角色
- 建立函數部署套件
- 建立 Lambda 函式
- 設定 Amazon S3 以調用函數
- 使用虛擬事件來測試 Lambda 函數
- 使用 Amazon S3 觸發條件測試函數
- 清除您的資源

先決條件

如果您想要使用 AWS CLI 來完成教學課程，請安裝最新版本的 AWS Command Line Interface。

對於 Lambda 函數程式碼，您可以使用 Python 或 Node.js。為您想要使用的語言安裝語言支援工具和套件管理工具。

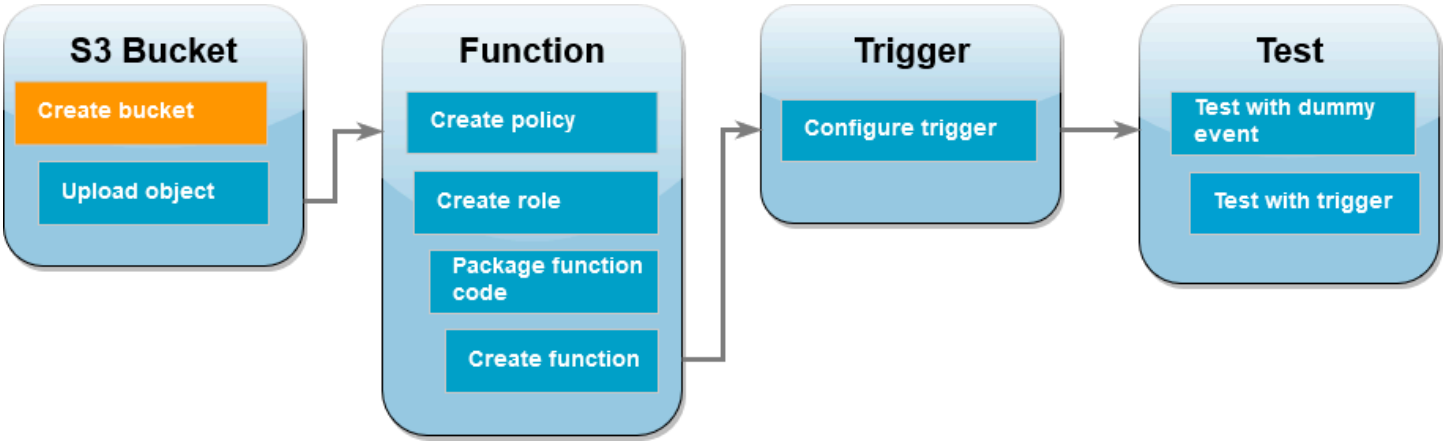
安裝 AWS Command Line Interface

如果您尚未安裝 AWS Command Line Interface，請依照安裝或更新最新版本的 AWS CLI中的步驟進行安裝。

本教學課程需使用命令列終端機或 Shell 來執行命令。在 Linux 和 macOS 中，使用您偏好的 Shell 和套件管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。

建立兩個 Amazon S3 儲存貯體

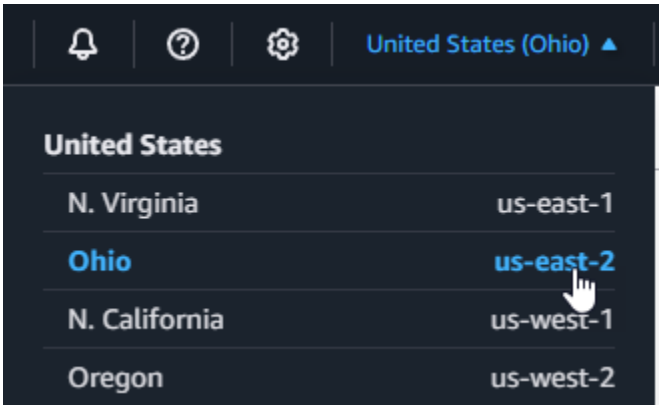


首先建立兩個 Amazon S3 儲存貯體。第一個儲存貯體是將接收映像上傳的來源儲存貯體。當您調用函數時，Lambda 會使用第二個儲存貯體來儲存已調整大小的縮圖。

AWS 管理主控台

建立 Amazon S3 儲存貯體 (主控台)

1. 開啟 Amazon S3 主控台，然後選取一般用途儲存貯體頁面。
2. 選取最 AWS 區域 接近您地理位置的 。可使用螢幕頂端的下拉式清單來變更區域。在本教學課程稍後的階段，您必須在同一區域中建立 Lambda 函數。



3. 選擇建立儲存貯體。
4. 在 General configuration (一般組態) 下，執行下列動作：
   a. 在儲存貯體類型欄位中，選取一般用途。
   b. 在儲存貯體名稱欄位中，輸入符合 Amazon S3 儲存貯體命名規則的全域唯一名稱。儲存貯體名稱只能包含小寫字母、數字、句點 (.) 和連字號 (-)。
5. 其他所有選項維持設為預設值，然後選擇建立儲存貯體。
6. 重複步驟 1 到 5 來建立目的地儲存貯體。對於儲存貯體名稱，輸入 **amzn-s3-demo-source-bucket-resized**，其中 **amzn-s3-demo-source-bucket** 是您剛才建立的來源儲存貯體名稱。

AWS CLI

建立 Amazon S3 儲存貯體 (AWS CLI)

1. 執行下列 CLI 命令以建立來源儲存貯體。您為儲存貯體選擇的名稱必須是全域唯一的，並遵循 Amazon S3 儲存貯體命名規則。名稱只能包含小寫字母、數字、句點 (.) 和連字號 (-)。對於 `region` 和 `LocationConstraint`，請選擇最接近您地理位置的 AWS 區域。

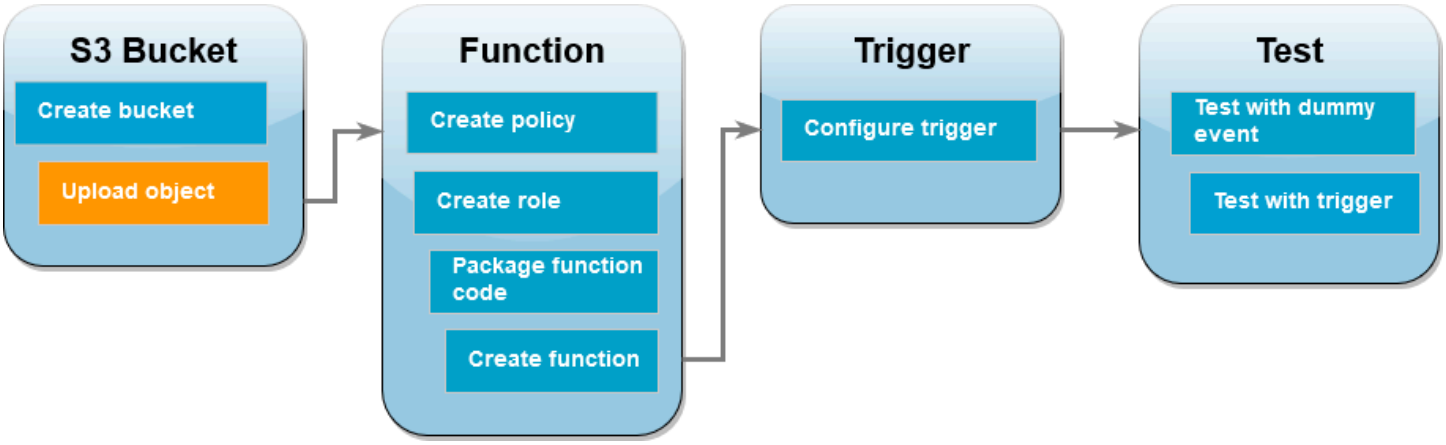
```
aws s3api create-bucket --bucket amzn-s3-demo-source-bucket --region us-east-1 \
--create-bucket-configuration LocationConstraint=us-east-1
```

   稍後在教學課程中，您必須在與來源儲存貯體 AWS 區域 相同的 中建立 Lambda 函數，因此請記下您選擇的區域。
2. 執行下列命令以建立目的地儲存貯體。對於儲存貯體名稱，必須使用 **amzn-s3-demo-source-bucket-resized**，其中 **amzn-s3-demo-source-bucket** 是您在步驟 1 中建立

的來源儲存貯體名稱。針對 region和 LocationConstraint，選擇 AWS 區域 您用來建立來源儲存貯體的相同 。

```
aws s3api create-bucket --bucket amzn-s3-demo-source-bucket-resized --region us-east-1 \
--create-bucket-configuration LocationConstraint=us-east-1
```

將測試映像上傳到來源儲存貯體



稍後在教學課程中，您將使用 AWS CLI 或 Lambda 主控台叫用 Lambda 函數來測試 Lambda 函數。若要確認函数運作正常，來源儲存貯體需要包含測試映像。此映像可以是您選擇的任何 JPG 或 PNG 檔案。

AWS 管理主控台

將測試映像上傳到來源儲存貯體 (主控台)

1. 開啟 Amazon S3 主控台的儲存貯體頁面。
2. 選取您在上一個步驟所建立的來源儲存貯體。
3. 選擇上傳。
4. 選擇新增檔案，然後使用檔案選擇器選擇您要上傳的物件。
5. 選擇開啟，然後選擇上傳。

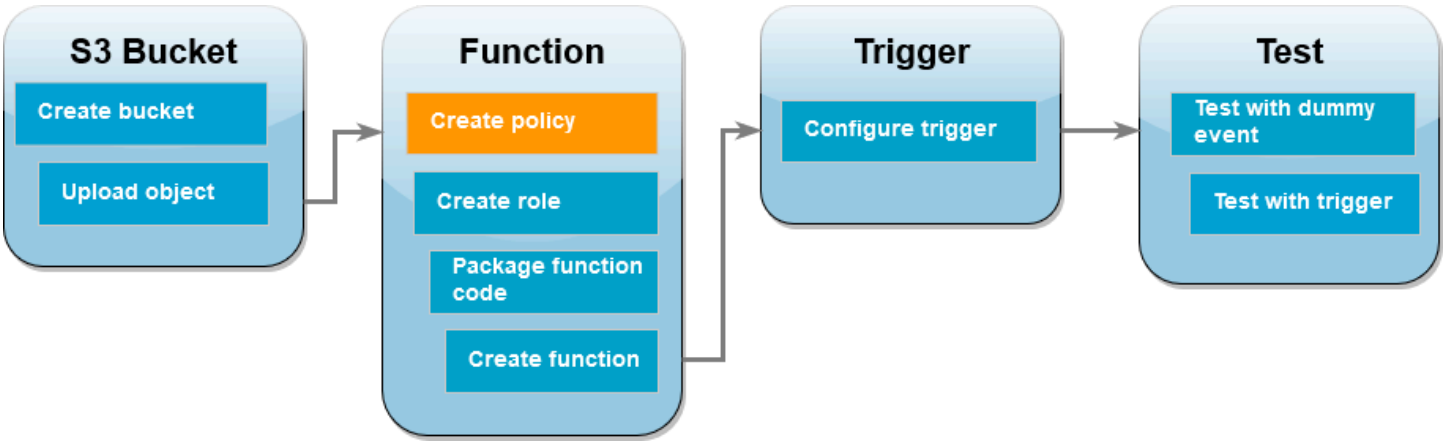
AWS CLI

將測試映像上傳到來源儲存貯體 (AWS CLI)

- 從包含要上傳之映像的目錄中，執行下列 CLI 命令。將 `--bucket` 參數取代為來源儲存貯體名稱。對於 `--key` 和 `--body` 參數，請使用測試映像的檔案名稱。

```
aws s3api put-object --bucket amzn-s3-demo-source-bucket --key HappyFace.jpg --
body ./HappyFace.jpg
```

建立許可政策



建立 Lambda 函數的第一個步驟是建立許可政策。此政策為您的函數提供存取其他 AWS 資源所需的許可。在本教學課程中，該政策為 Lambda 提供了 Amazon S3 儲存貯體的讀取和寫入許可，並允許其寫入到 Amazon CloudWatch Logs 日誌。

AWS 管理主控台

建立政策 (主控台)

1. 開啟 AWS Identity and Access Management (IAM) 主控台的政策頁面。
2. 選擇建立政策。
3. 選擇 JSON 索引標籤，然後將下列政策貼到 JSON 編輯器。

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

4. 選擇下一步。
5. 在政策詳細資訊下，針對政策政策名稱 ，輸入 **LambdaS3Policy**。
6. 選擇建立政策。

AWS CLI

建立政策 (AWS CLI)

1. 將下面的 JSON 儲存在名為 `policy.json` 的檔案中。

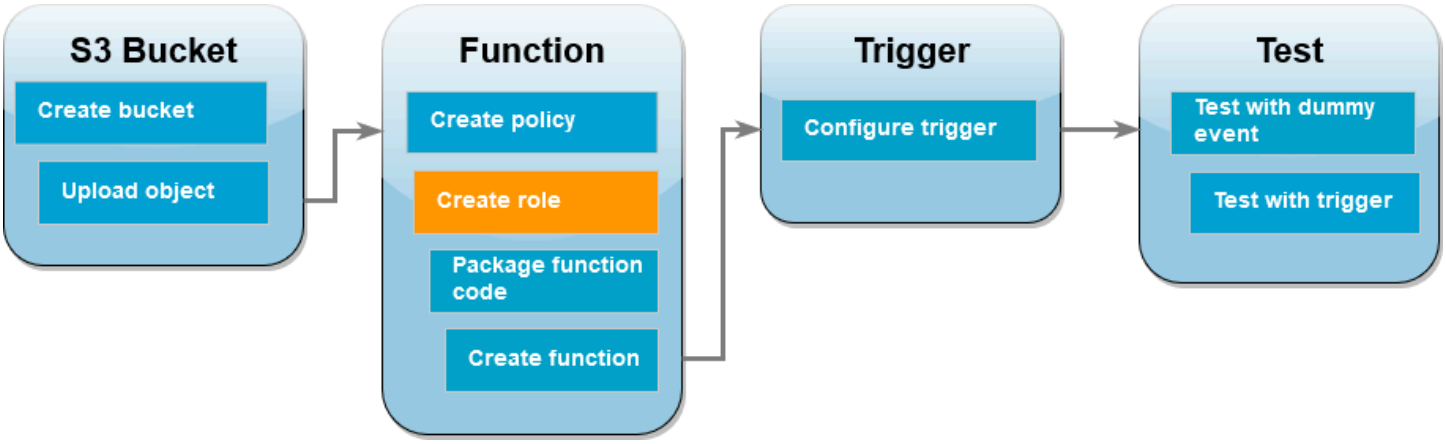
```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
```

```
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

2. 在儲存 JSON 政策文件的目錄中執行下列 CLI 命令。

```
aws iam create-policy --policy-name LambdaS3Policy --policy-document file://
policy.json
```

建立執行角色



執行角色是授予 Lambda 函數存取 AWS 服務 和資源許可的 IAM 角色。若要為函數提供 Amazon S3 儲存貯體的讀取和寫入存取權，請連接您在上個步驟建立的許可政策。

AWS 管理主控台

建立執行角色並連接許可政策 (主控台)

1. 開啟 (IAM) 主控台的角色頁面。
2. 選擇建立角色。
3. 對於可信實體類型，選取 AWS 服務，然後針對使用案例選取 Lambda。
4. 選擇下一步。
5. 執行下列動作，新增您在上個步驟中建立的許可政策：
   a. 在政策搜尋方塊中，輸入 **LambdaS3Policy**。
   b. 在搜尋結果中，選取 LambdaS3Policy 的核取方塊。
   c. 選擇下一步。
6. 在角色詳細資訊下，針對角色名稱，輸入 **LambdaS3Role**。
7. 選擇建立角色。

AWS CLI

建立執行角色並連接許可政策 (AWS CLI)

1. 將下面的 JSON 儲存在名為 `trust-policy.json` 的檔案中。此信任政策允許 Lambda 透過授予服務主體呼叫 AWS Security Token Service (AWS STS) `AssumeRole`動作的許可來使用角色的`lambda.amazonaws.com`許可。

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
```

```
}
```

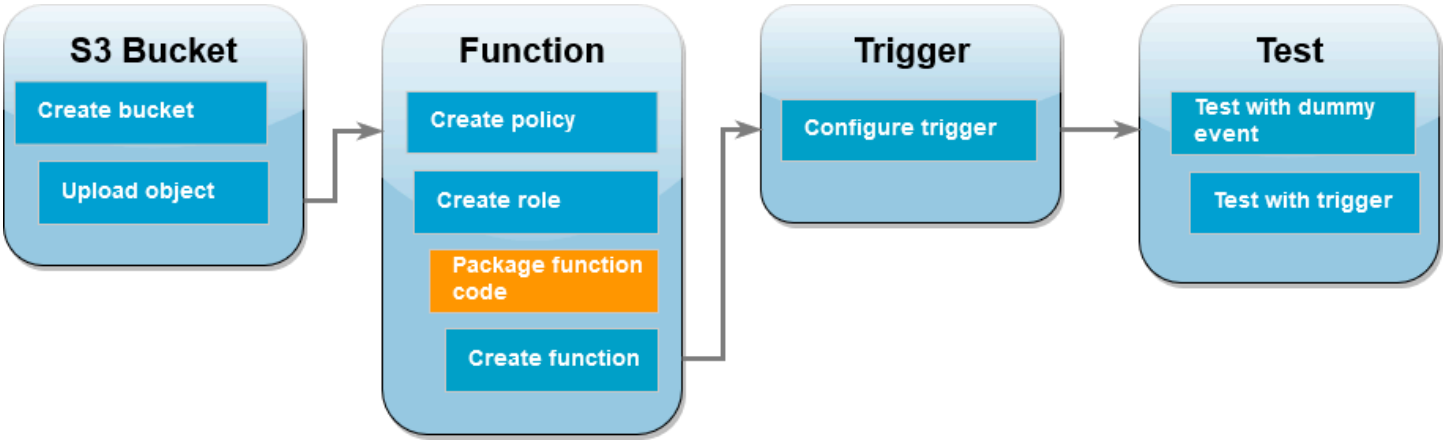
2. 在儲存 JSON 信任政策文件的目錄中，執行下列 CLI 命令，建立執行角色。

```
aws iam create-role --role-name LambdaS3Role --assume-role-policy-document
 file://trust-policy.json
```

3. 執行下列 CLI 命令，連接您在上個步驟中建立的許可政策。將政策 ARN 中的 AWS 帳戶 號碼取代為您自己的帳戶號碼。

```
aws iam attach-role-policy --role-name LambdaS3Role --policy-arn
 arn:aws:iam::123456789012:policy/LambdaS3Policy
```

建立函數部署套件



要建立函數，需建立包含函數程式碼和其相依項的部署套件。對於此 `CreateThumbnail` 函數，函數程式碼使用單獨的程式庫來調整映像大小。依照所選語言的指示，建立包含所需程式庫的部署套件。

Node.js

建立部署套件 (Node.js)

1. 為函數程式碼和相依項建立名為 `lambda-s3` 的目錄並導覽到該目錄。

```
mkdir lambda-s3
cd lambda-s3
```

2. 使用 npm 建立新 Node.js 專案。若要接受互動體驗中提供的預設選項，請按下 `Enter`。

```
npm init
```

3. 將以下函數程式碼儲存在名為 index.mjs 檔案中。請務必以您建立來源和目的地儲存貯體的 AWS 區域 取代 us-east-1。

```
// dependencies
import { S3Client, GetObjectCommand, PutObjectCommand } from '@aws-sdk/client-
s3';

import { Readable } from 'stream';

import sharp from 'sharp';
import util from 'util';


// create S3 client
const s3 = new S3Client({region: 'us-east-1'});

// define the handler function
export const handler = async (event, context) => {

// Read options from the event parameter and get the source bucket
console.log("Reading options from event:\n", util.inspect(event, {depth: 5}));
  const srcBucket = event.Records[0].s3.bucket.name;

// Object key may have spaces or unicode non-ASCII characters
const srcKey    = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/
g, " "));
const dstBucket = srcBucket + "-resized";
const dstKey    = "resized-" + srcKey;

// Infer the image type from the file suffix
const typeMatch = srcKey.match(/\.([^.]*)$/);
if (!typeMatch) {
  console.log("Could not determine the image type.");
  return;
}

// Check that the image type is supported
const imageType = typeMatch[1].toLowerCase();
if (imageType != "jpg" && imageType != "png") {
  console.log(`Unsupported image type: ${imageType}`);
```

```
  return;
}

// Get the image from the source bucket. GetObjectCommand returns a stream.
try {
  const params = {
    Bucket: srcBucket,
    Key: srcKey
  };
  var response = await s3.send(new GetObjectCommand(params));
  var stream = response.Body;

// Convert stream to buffer to pass to sharp resize function.
  if (stream instanceof Readable) {
    var content_buffer = Buffer.concat(await stream.toArray());

  } else {
    throw new Error('Unknown object stream type');
  }


} catch (error) {
  console.log(error);
  return;
}


// set thumbnail width. Resize will set the height automatically to maintain
 aspect ratio.
const width  = 200;

// Use the sharp module to resize the image and save in a buffer.
try {
  var output_buffer = await sharp(content_buffer).resize(width).toBuffer();

} catch (error) {
  console.log(error);
  return;
}

// Upload the thumbnail image to the destination bucket
try {
  const destparams = {
    Bucket: dstBucket,
```

```
    Key: dstKey,
    Body: output_buffer,
    ContentType: "image"
  };

  const putResult = await s3.send(new PutObjectCommand(destparams));

  } catch (error) {
    console.log(error);
    return;
  }

  console.log('Successfully resized ' + srcBucket + '/' + srcKey +
    ' and uploaded to ' + dstBucket + '/' + dstKey);
  };
```

4. 在 lambda-s3 目錄中，使用 npm 安裝 Sharp 程式庫。請注意，最新的 Sharp 版本 (0.33) 與 Lambda 不相容。安裝版本 0.32.6 以完成本教學課程。

```
npm install sharp@0.32.6
```

npm install 命令為您的模組建立一個 node_modules 目錄。在此步驟之後，目錄結構應如下所示。

```
lambda-s3
|- index.mjs
|- node_modules
|  |- base64js
|  |- bl
|  |- buffer
...
|- package-lock.json
|- package.json
```

5. 建立包含函數程式碼及其相依項 .zip 部署套件。在 MacOS 或 Linux 中，執行下列命令。

```
zip -r function.zip .
```

在 Windows 中，使用您偏好的 zip 公用程式建立 .zip 檔案。請確保 index.mjs、package.json 和 package-lock.json 檔案以及 node_modules 目錄全部都位於 .zip 檔案的根目錄。

Python

建立部署套件 (Python)

1. 將程式碼範例儲存為名為 lambda_function.py 的檔案。

```
import boto3
import os
import sys
import uuid
from urllib.parse import unquote_plus
from PIL import Image
import PIL.Image

s3_client = boto3.client('s3')

def resize_image(image_path, resized_path):
  with Image.open(image_path) as image:
    image.thumbnail(tuple(x / 2 for x in image.size))
    image.save(resized_path)

def lambda_handler(event, context):
  for record in event['Records']:
    bucket = record['s3']['bucket']['name']
    key = unquote_plus(record['s3']['object']['key'])
    tmpkey = key.replace('/', '')
    download_path = '/tmp/{}{}'.format(uuid.uuid4(), tmpkey)
    upload_path = '/tmp/resized-{}'.format(tmpkey)
    s3_client.download_file(bucket, key, download_path)
    resize_image(download_path, upload_path)
    s3_client.upload_file(upload_path, '{}-resized'.format(bucket), 'resized-
{}'.format(key))
```

2. 在建立 lambda_function.py 檔案的同一個目錄中，建立名為 package 的新目錄並安裝 Pillow (PIL) 程式庫和 適用於 Python (Boto3) 的 AWS SDK。雖然 Lambda Python 執行期包含 Boto3 SDK 的版本，但建議您將函數的所有相依項新增至部署套件，即使其包含在執行期中。如需詳細資訊，請參閱 Python 中的執行期相依項。

```
mkdir package
pip install \
--platform manylinux2014_x86_64 \
--target=package \
--implementation cp \
```

```
--python-version 3.12 \
--only-binary=:all: --upgrade \
pillow boto3
```

Pillow 程式庫中包含 C/C ++ 程式碼。使用 `--platform manylinux_2014_x86_64` 和 `--only-binary=:all:` 選項，pip 便會下載並安裝適用的 Pillow 版本，其中包含與 Amazon Linux 2 作業系統相容的預先編譯二進位檔。這可確保無論本機建置機器的作業系統和架構為何，您的部署套件都能在 Lambda 執行環境中運作。

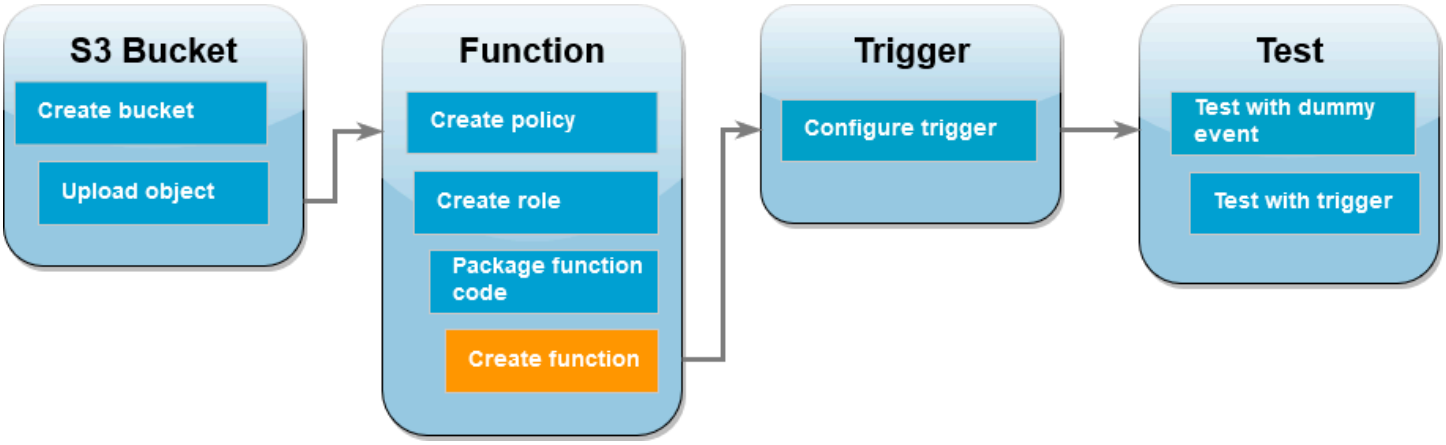
3. 建立一個包含應用程式碼以及 Pillow 和 Boto3 程式庫的 .zip 檔案。在 Linux 或 MacOS 中，使用命令列界面執行下列命令。

```
cd package
zip -r ../lambda_function.zip .
cd ..
zip lambda_function.zip lambda_function.py
```

在 Windows 中，使用您偏好的 zip 工具建立 `lambda_function.zip` 檔案。確保包含相依項的 `lambda_function.py` 檔案和資料夾全部都在 .zip 檔案的根目錄中。

您也可以使用 Python 虛擬環境建立部署套件。請參閱 使用 .zip 封存檔部署 Python Lambda 函數

建立 Lambda 函式



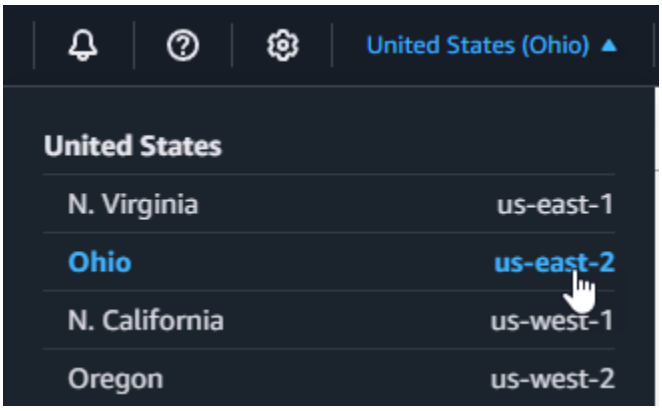
您可以使用 AWS CLI 或 Lambda 主控台建立 Lambda 函數。依照所選語言的指示建立函數。

AWS 管理主控台

建立函數的方式 (主控台)

要使用主控台建立 Lambda 函數，首先建立包含一些 'Hello world' 程式碼的基本函數。然後，透過上傳您在上一個步驟中建立的 .zip 或 JAR 檔案，將此程式碼取代為您自己的函數程式碼。

1. 開啟 Lambda 主控台中的函數頁面。
2. 請確定您使用 AWS 區域 與建立 Amazon S3 儲存貯體相同的 。可使用螢幕頂端的下拉式清單來變更區域。



3. 選擇 Create function (建立函數)。
4. 選擇 Author from scratch (從頭開始撰寫)。
5. 在基本資訊下，請執行下列動作：
   a. 針對函數名稱，請輸入 **`CreateThumbnail`**。
   b. 對於執行時期，請根據您為函數選擇的語言，選擇 Node.js 22.x 或 Python 3.12。
   c. 對於 Architecture (架構)，選擇 x86_64。
6. 在變更預設執行角色索引標籤中，執行下列操作：
   a. 展開索引標籤，然後選擇使用現有角色。
   b. 選擇您之前建立的 `LambdaS3Role`。
7. 選擇建立函數。

上傳函數程式碼 (主控台)

1. 在程式碼來源窗格中選擇上傳來源。
2. 選擇 .zip 檔案。

3. 選擇上傳。
4. 在檔案選擇器中，選取 .zip 檔案，並選擇開啟。
5. 選擇儲存。

AWS CLI

建立函數 (AWS CLI)

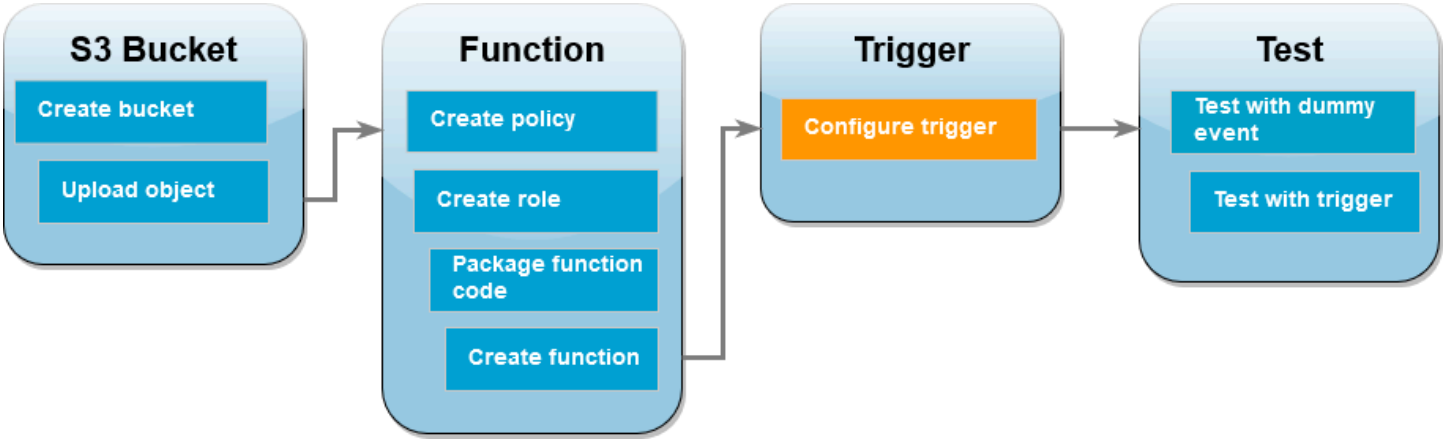
- 執行所選語言的 CLI 命令。針對 `role` 參數，請務必將 取代123456789012為您自己的 AWS 帳戶 ID。對於 `region` 參數，將 `us-east-1` 取代為您建立 Amazon S3 儲存貯體所在的區域。
  - 對於 Node.js，請從包含 `function.zip` 檔案的目錄中執行下列命令。

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
--timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-1
```

  - 對於 Python，請從包含 `lambda_function.zip` 檔案的目錄中執行下列命令。

```
aws lambda create-function --function-name CreateThumbnail \
--zip-file fileb://lambda_function.zip --handler
 lambda_function.lambda_handler \
--runtime python3.14 --timeout 10 --memory-size 1024 \
--role arn:aws:iam::123456789012:role/LambdaS3Role --region us-east-1
```

設定 Amazon S3 以調用函數



若要在將映像上傳至來源儲存貯體時執行 Lambda 函數，您需要設定函數的觸發條件。可以使用主控台或 AWS CLI來設定 Amazon S3 觸發條件。

> **Important**
>
> 此程序會將 Amazon S3 儲存貯體設定為每次在儲存貯體中建立物件時即會調用您的函數。請務必僅在來源儲存貯體上進行設定。如果 Lambda 函數在進行調用的同一個儲存貯體中建立物件，則可以在一個迴圈中連續調用函數。這可能會導致未預期的費用向您的 收費 AWS 帳戶。

AWS 管理主控台

設定 Amazon S3 觸發條件 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇您的函數 (`CreateThumbnail`)。
2. 選擇 Add trigger (新增觸發條件)。
3. 選取 S3。
4. 在儲存貯體下，選取您的來源儲存貯體。
5. 在事件類型下，選取所有物件建立事件。
6. 在遞迴調用下，選取核取方塊，確認您了解不建議使用相同的 Amazon S3 儲存貯體進行輸入和輸出作業。您可以閱讀無伺服器園地中 導致 Lambda 函數失控的遞迴模式，進一步了解 Lambda 中的遞迴調用模式。
7. 選擇新增。

   當您使用 Lambda 主控台建立觸發條件時，Lambda 會自動建立資源型政策，為您選取的服務授予調用函數的許可。

AWS CLI

設定 Amazon S3 觸發條件 (AWS CLI)

1. 若要讓 Amazon S3 來源儲存貯體在新增映像檔案時調用函數，您首先需要使用資源型政策為函數設定許可。資源型政策陳述式提供叫用 函數的其他 AWS 服務 許可。若要授予 Amazon S3 調用函數的許可，請執行下列 CLI 命令。請務必以您自己的 AWS 帳戶 ID 取代 `source-account` 參數，並使用您自己的來源儲存貯體名稱。

   ```
   aws lambda add-permission --function-name CreateThumbnail \
   ```

```
--principal s3.amazonaws.com --statement-id s3invoke --action
 "lambda:InvokeFunction" \
--source-arn arn:aws:s3:::amzn-s3-demo-source-bucket \
--source-account 123456789012
```

使用此命令定義的政策允許 Amazon S3 僅在來源儲存貯體上發生動作時調用函數。

> **Note**
>
> 雖然 Amazon S3 儲存貯體名稱全域唯一，但是在使用資源型政策時，最佳實務是指定儲存貯體必須屬於您的帳戶。這是因為如果您刪除儲存貯體，另一個 可以使用相同的 Amazon Resource Name (ARN) AWS 帳戶 建立儲存貯體。

2. 將下面的 JSON 儲存在名為 `notification.json` 的檔案中。套用至來源儲存貯體時，此 JSON 會設定儲存貯體，以便在每次新增新物件時傳送通知至 Lambda 函數。將 AWS 帳戶 Lambda 函數 ARN AWS 區域 中的數字 和 取代為您自己的帳號和區域。

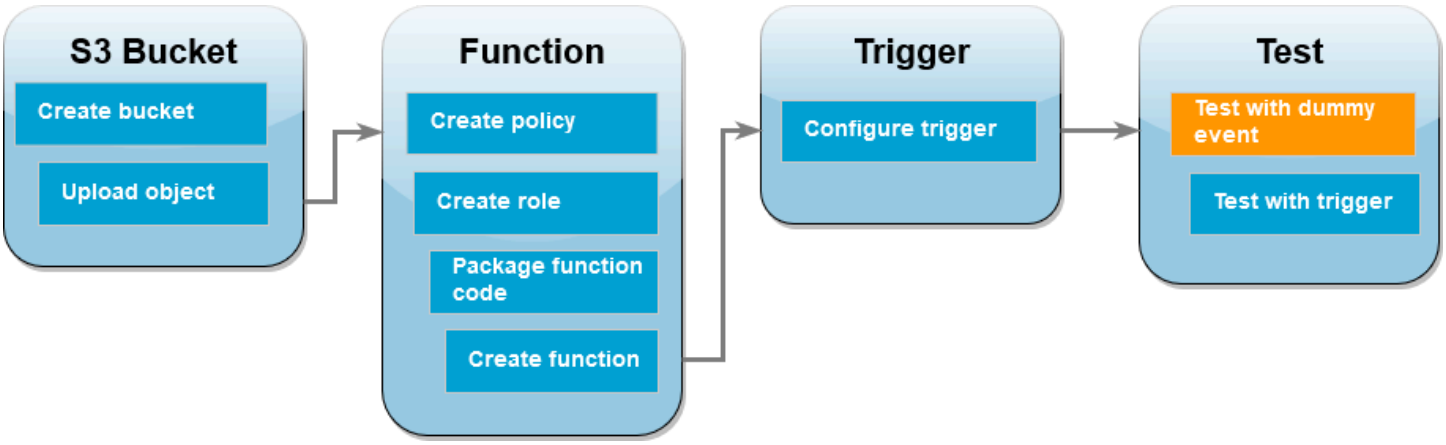
```
{
"LambdaFunctionConfigurations": [
    {
      "Id": "CreateThumbnailEventConfiguration",
      "LambdaFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:CreateThumbnail",
      "Events": [ "s3:ObjectCreated:Put" ]
    }
  ]
}
```

3. 執行下列 CLI 命令，將您建立的 JSON 檔案中的通知設定套用至來源儲存貯體。用您自己的來源儲存貯體名稱取代 `amzn-s3-demo-source-bucket`。

```
aws s3api put-bucket-notification-configuration --bucket amzn-s3-demo-source-
bucket \
--notification-configuration file://notification.json
```

若要深入了解 `put-bucket-notification-configuration` 命令和 `notification-configuration` 選項，請參閱 AWS CLI 命令參考中的 put-bucket-notification-configuration。

使用虛擬事件來測試 Lambda 函數



在將映像檔案新增至 Amazon S3 來源儲存貯體以測試整個設定之前，可以透過使用虛擬事件調用 Lambda 函數來測試其是否正常運作。Lambda 中的事件是一種 JSON 格式的文件，它包含供函數處理的資料。Amazon S3 調用函數時，傳送至函數的事件會包含諸如儲存貯體名稱、儲存貯體 ARN 和物件金鑰等資訊。

AWS 管理主控台

使用虛擬事件來測試 Lambda 函數 (主控台)

1. 開啟 Lambda 主控台的函數頁面，然後選擇您的函數 (`CreateThumbnail`)。
2. 選擇測試標籤。
3. 若要建立測試事件，請在測試事件窗格中執行下列動作：
   a. 在測試事件動作下方，選取建立新事件。
   b. 事件名稱輸入 **`myTestEvent`**。
   c. 對於範本，選取 S3 Put。
   d. 將下列參數的值取代為您自己的值。
      - 對於 `awsRegion`，`us-east-1`將 取代 AWS 區域 為您在其中建立 Amazon S3 儲存貯體的 。
      - 對於 `name`，將 `amzn-s3-demo-bucket` 取代為 Amazon S3 來源儲存貯體的名稱。
      - 對於 `key`，將 `test%2Fkey` 取代為您在步驟 將測試映像上傳到來源儲存貯體 中上傳至來源儲存貯體之測試物件的檔案名稱。

```
{
```

```
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

e. 選擇儲存。

4. 在測試事件窗格中，選擇測試。

5. 若要檢查您的函數已建立大小經過調整的映像版本並將其存放在目標 Amazon S3 儲存貯體中，請執行以下操作：

a. 開啟 Amazon S3 主控台的儲存貯體頁面。

b. 選擇目標儲存貯體，並確認在物件窗格中列出已調整大小的檔案。

AWS CLI

使用虛擬事件來測試 Lambda 函數 (AWS CLI)

1. 將下面的 JSON 儲存在名為 dummyS3Event.json 的檔案中。將下列參數的值取代為您自己的值：

   - 對於 awsRegion，us-east-1將 取代 AWS 區域 為您在其中建立 Amazon S3 儲存貯體的 。
   - 對於 name，將 amzn-s3-demo-bucket 取代為 Amazon S3 來源儲存貯體的名稱。
   - 對於 key，將 test%2Fkey 取代為您在步驟 將測試映像上傳到來源儲存貯體 中上傳至來源儲存貯體之測試物件的檔案名稱。

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2": "EXAMPLE123/5678abcdefghijklambdaisawesome/
mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
```

```
          "name": "amzn-s3-demo-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::amzn-s3-demo-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```

2. 在您儲存 dummyS3Event.json 檔案的目錄中，透過執行下列 CLI 命令來調用函數。此命令透過將 RequestResponse 指定為調用類型參數的值來同步調用 Lambda 函數。若要進一步了解同步和非同步調用，請參閱調用 Lambda 函數。

```
aws lambda invoke --function-name CreateThumbnail \
--invocation-type RequestResponse --cli-binary-format raw-in-base64-out \
--payload file://dummyS3Event.json outputfile.txt
```

如果使用的是 AWS CLI 版本 2，則需要 cli-binary-format 選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。如需詳細資訊，請參閱《支援AWS CLI 的全域命令列選項》。

3. 確認函數已建立映像的縮圖版本，並將其儲存到目標 Amazon S3 儲存貯體。執行以下 CLI 命令，將 amzn-s3-demo-source-bucket-resized 取代為您自己的目的地儲存貯體的名稱。

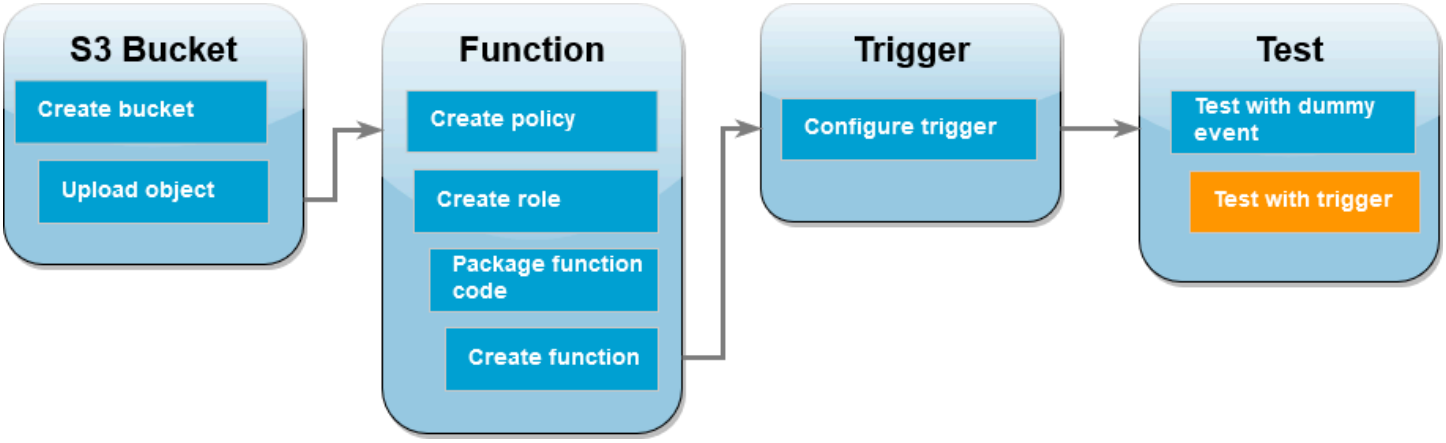
```
aws s3api list-objects-v2 --bucket amzn-s3-demo-source-bucket-resized
```

您應該會看到類似下列的輸出。Key 參數會顯示調整大小後的映像檔案名稱。

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
```

```
            "LastModified": "2023-06-06T21:40:07+00:00",
            "ETag": "\"d8ca652ffe83ba6b721ffc20d9d7174a\"",
            "Size": 2633,
            "StorageClass": "STANDARD"
        }
    ]
}
```

使用 Amazon S3 觸發條件測試函數



既然您已確認 Lambda 函數運作正常，便可將映像檔案新增至 Amazon S3 來源儲存貯體來測試完整的設定。將映像新增至來源儲存貯體時，應該會自動調用 Lambda 函數。函數會建立已調整大小的檔案版本，並將其存放在目標儲存貯體中。

AWS 管理主控台

使用 Amazon S3 觸發條件測試 Lambda 函數 (主控台)

1. 若要將映像上傳到 Amazon S3 儲存貯體，請執行以下操作：
   a. 開啟 Amazon S3 主控台的儲存貯體頁面，並選擇來源儲存貯體。
   b. 選擇上傳。
   c. 選擇新增檔案，然後使用檔案選擇器選擇您要上傳的映像檔案。映像物件可以是任何 .jpg 或 .png 檔案。
   d. 選擇開啟，然後選擇上傳。
2. 透過執行下列操作，確認 Lambda 已在目標儲存貯體中儲存了調整大小後的映像檔案版本：
   a. 導覽回 Amazon S3 主控台的儲存貯體頁面，然後選擇目的地儲存貯體。

b. 在物件窗格中，現在應該會看到兩個已調整大小的映像檔案，分別來自 Lambda 函數的每個測試。若要下載已調整大小的映像，請選取檔案，然後選擇下載。

AWS CLI

使用 Amazon S3 觸發條件測試 Lambda 函數 (AWS CLI)

1. 從包含要上傳之映像的目錄中，執行下列 CLI 命令。將 `--bucket` 參數取代為來源儲存貯體名稱。對於 `--key` 和 `--body` 參數，請使用測試映像的檔案名稱。測試映像可以是任何 .jpg 或 .png 檔案。

```
aws s3api put-object --bucket amzn-s3-demo-source-bucket --key SmileyFace.jpg --body ./SmileyFace.jpg
```

2. 確認函數已建立映像的縮圖版本，並將其儲存到目標 Amazon S3 儲存貯體。執行以下 CLI 命令，將 `amzn-s3-demo-source-bucket-resized` 取代為您自己的目的地儲存貯體的名稱。

```
aws s3api list-objects-v2 --bucket amzn-s3-demo-source-bucket-resized
```

如果函數成功運作，則您會看到類似以下內容的輸出。您的目標儲存貯體現在應包含兩個已調整大小的檔案。

```
{
    "Contents": [
        {
            "Key": "resized-HappyFace.jpg",
            "LastModified": "2023-06-07T00:15:50+00:00",
            "ETag": "\"7781a43e765a8301713f533d70968a1e\"",
            "Size": 2763,
            "StorageClass": "STANDARD"
        },
        {
            "Key": "resized-SmileyFace.jpg",
            "LastModified": "2023-06-07T00:13:18+00:00",
            "ETag": "\"ca536e5a1b9e32b22cd549e18792cdbc\"",
            "Size": 1245,
            "StorageClass": "STANDARD"
        }
    ]
```

```
}
```

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。透過刪除您不再使用 AWS 的資源，您可以避免不必要的 費用 AWS 帳戶。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **confirm**，然後選擇 Delete (刪除)。

刪除您建立的政策

1. 開啟 IAM 主控台中的 Policies (政策) 頁面。
2. 選取您建立的政策 (AWSLambdaS3Policy)。
3. 選擇政策動作，然後刪除。
4. 選擇 刪除 。

若要刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇刪除。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇刪除。

刪除 S3 儲存貯體

1. 開啟 Amazon S3 主控台。
2. 選擇您建立的儲存貯體。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入儲存貯體的名稱。

5. 選擇刪除儲存貯體。

在 Lambda 函式中使用 Secrets Manager 秘密

AWS Secrets Manager 可協助您管理 Lambda 函數所需的登入資料、API 金鑰和其他秘密。與直接使用 AWS SDK 擷取秘密相比，您有兩種主要方法來擷取 Lambda 函數中的秘密，同時提供更好的效能和更低的成本：

- AWS 參數和秘密 Lambda 延伸 - 與執行時間無關的解決方案，提供簡單的 HTTP 界面以擷取秘密
- 適用於 AWS Lambda 參數公用程式的 Powertools - 一種程式碼整合解決方案，支援具有內建轉換的多個提供者 (Secrets Manager、參數存放區、AppConfig)

這兩種方案都會維護秘密的本機快取，無需函式在每次調用時都呼叫 Secrets Manager。當您的函式請求秘密時，會先檢查快取。如果秘密可用且未過期，則會立即傳回。否則，會從 Secrets Manager 中擷取秘密，快取後再傳回。此快取機制透過最大限度地減少 API 呼叫次數來縮短回應時間並降低成本。

選擇合適方案

在延伸模組與 PowerTools 之間進行選擇時，建議考量以下因素：

在下列情況下使用 AWS 參數和秘密 Lambda 延伸模組：

- 您想要一種與執行時期無關的解決方案，並且其可與任何 Lambda 執行時期搭配使用
- 您不想將程式碼相依項新增至函式
- 您僅需從 Secrets Manager 或 Parameter Store 中擷取秘密

在下列情況下，使用 Powertools 做為 AWS Lambda 參數公用程式：

- 您希望獲得與應用程式程式碼整合的開發體驗
- 您需要支援多個提供者 (Secrets Manager、Parameter Store、AppConfig)
- 您需要內建的資料轉換功能 (JSON 剖析、base64 解碼)
- 您目前使用的是 Python、TypeScript、Java 或 .NET 執行時期

何時將 Secrets Manager 與 Lambda 搭配使用

將 Secrets Manager 與 Lambda 搭配使用的常見案例包括：

- 儲存函式用來連線至 Amazon RDS 或其他資料庫的資料庫憑證

- 管理函式呼叫之外部服務的 API 金鑰
- 儲存加密金鑰或其他敏感組態資料
- 自動輪換憑證，而無需更新函式程式碼

使用 AWS 參數和秘密 Lambda 延伸模組

Lambda 延伸模組的 AWS 參數和秘密使用與任何 Lambda 執行時間相容的簡單 HTTP 介面。依預設，該功能會快取秘密 300 秒 (5 分鐘)，並且最多可保留 1,000 個秘密。您可以使用環境變數自訂這些設定。

在 Lambda 函式中使用 Secrets Manager

本節假設您已擁有一個 Secrets Manager 秘密。若要建立秘密，請參閱建立 AWS Secrets Manager 秘密。

建立部署套件

選擇您偏好的執行時期，並依照步驟建立從 Secrets Manager 中擷取秘密的函式。範例函式會從 Secrets Manager 中擷取秘密，並可用來存取應用程式中的資料庫憑證、API 金鑰或其他敏感組態資料。

Python

若要建立 Python 函數

1. 建立一個新的專案目錄並導覽至該目錄。範例：

```
mkdir my_function
cd my_function
```

2. 建立名為 `lambda_function.py` 的檔案，並貼上以下程式碼。對於 `secret_name`，請使用秘密的名稱或 Amazon Resource Name (ARN)。

```
import json
import os
import requests

def lambda_handler(event, context):
    try:
        # Replace with the name or ARN of your secret
```

```
        secret_name = "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME"

        secrets_extension_endpoint = f"http://localhost:2773/secretsmanager/get?
secretId={secret_name}"
        headers = {"X-Aws-Parameters-Secrets-Token":
 os.environ.get('AWS_SESSION_TOKEN')}

        response = requests.get(secrets_extension_endpoint, headers=headers)
        print(f"Response status code: {response.status_code}")

        secret = json.loads(response.text)["SecretString"]
        print(f"Retrieved secret: {secret}")

        return {
            'statusCode': response.status_code,
            'body': json.dumps({
                'message': 'Successfully retrieved secret',
                'secretRetrieved': True
            })
        }

    except Exception as e:
        print(f"Error: {str(e)}")
        return {
            'statusCode': 500,
            'body': json.dumps({
                'message': 'Error retrieving secret',
                'error': str(e)
            })
        }
```

3. 建立名為 `requirements.txt` 的檔案，並貼上此內容：

```
requests
```

4. 安裝相依項：

```
pip install -r requirements.txt -t .
```

5. 建立一個包含所有檔案的 .zip 檔案：

```
zip -r function.zip .
```

Node.js

若要建立 Node.js 函數

1. 建立一個新的專案目錄並導覽至該目錄。範例：

```
mkdir my_function
cd my_function
```

2. 建立名為 index.mjs 的檔案，並貼上以下程式碼。對於 secret_name，請使用秘密的名稱或 Amazon Resource Name (ARN)。

```
import http from 'http';

export const handler = async (event) => {
    try {
        // Replace with the name or ARN of your secret
        const secretName = "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME";
        const options = {
            hostname: 'localhost',
            port: 2773,
            path: `/secretsmanager/get?secretId=${secretName}`,
            headers: {
                'X-Aws-Parameters-Secrets-Token': process.env.AWS_SESSION_TOKEN
            }
        };

        const response = await new Promise((resolve, reject) => {
            http.get(options, (res) => {
                let data = '';
                res.on('data', (chunk) => { data += chunk; });
                res.on('end', () => {
                    resolve({
                        statusCode: res.statusCode,
                        body: data
                    });
                });
            }).on('error', reject);
        });

        const secret = JSON.parse(response.body).SecretString;
        console.log('Retrieved secret:', secret);
```

```
        return {
            statusCode: response.statusCode,
            body: JSON.stringify({
                message: 'Successfully retrieved secret',
                secretRetrieved: true
            })
        };
    } catch (error) {
        console.error('Error:', error);
        return {
            statusCode: 500,
            body: JSON.stringify({
                message: 'Error retrieving secret',
                error: error.message
            })
        };
    }
};
```

3. 建立一個包含 index.mjs 檔案的 .zip 檔案：

```
zip -r function.zip index.mjs
```

Java

若要建立 Lambda 函數

1. 建立一個 Maven 專案：

```
mvn archetype:generate \
    -DgroupId=example \
    -DartifactId=lambda-secrets-demo \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DarchetypeVersion=1.4 \
    -DinteractiveMode=false
```

2. 導覽至專案目錄：

```
cd lambda-secrets-demo
```

3. 開啟 pom.xml 並將內容替換如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://
maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>example</groupId>
    <artifactId>lambda-secrets-demo</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>com.amazonaws</groupId>
            <artifactId>aws-lambda-java-core</artifactId>
            <version>1.2.1</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.4</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            <createDependencyReducedPom>false</
createDependencyReducedPom>
                            <finalName>function</finalName>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
```

```
        </plugins>
    </build>
</project>
```

4. 將 /lambda-secrets-demo/src/main/java/example/App.java 重新命名為 Hello.java，以符合 Lambda 的預設 Java 處理常式名稱 (example.Hello::handleRequest)：

```
mv src/main/java/example/App.java src/main/java/example/Hello.java
```

5. 開啟 Hello.java 檔案並將內容替換如下。對於 secretName，請使用秘密的名稱或 Amazon Resource Name (ARN)。

```
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import java.net.URI;
import java.net.http.HttpClient;
import java.net.http.HttpRequest;
import java.net.http.HttpResponse;

public class Hello implements RequestHandler<Object, String> {
    private final HttpClient client = HttpClient.newHttpClient();

    @Override
    public String handleRequest(Object input, Context context) {
        try {
            // Replace with the name or ARN of your secret
            String secretName = "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME";
            String endpoint = "http://localhost:2773/secretsmanager/get?
secretId=" + secretName;

            HttpRequest request = HttpRequest.newBuilder()
                .uri(URI.create(endpoint))
                .header("X-Aws-Parameters-Secrets-Token",
 System.getenv("AWS_SESSION_TOKEN"))
                .GET()
                .build();

            HttpResponse<String> response = client.send(request,
                HttpResponse.BodyHandlers.ofString());
```

```
            String secret = response.body();
            secret = secret.substring(secret.indexOf("SecretString") + 15);
            secret = secret.substring(0, secret.indexOf("\""));

            System.out.println("Retrieved secret: " + secret);
            return String.format(
                "{\"statusCode\": %d, \"body\": \"%s\"}",
                response.statusCode(), "Successfully retrieved secret"
            );

        } catch (Exception e) {
            e.printStackTrace();
            return String.format(
                "{\"body\": \"Error retrieving secret: %s\"}",
                e.getMessage()
            );
        }
    }
}
```

6. 移除測試目錄。Maven 預設會建立此目錄，但此範例中不需要此目錄。

```
rm -rf src/test
```

7. 建置專案：

```
mvn package
```

8. 下載 JAR 檔案 (`target/function.jar`) 以供日後使用。

建立函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇建立函數。
3. 選取從頭開始撰寫。
4. 針對函數名稱，請輸入 **`secret-retrieval-demo`**。
5. 選擇偏好的執行時期。
6. 選擇建立函數。

若要上傳部署套件

1. 在函式的程式碼索引標籤中，選擇上傳來源，然後選擇 .zip 檔案 (適用於 Python 與 Node.js) 或 .jar 檔案 (適用於 Java)。
2. 上傳先前建立的部署套件。
3. 選擇儲存。

新增延伸

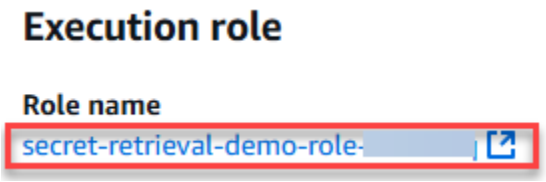
將 AWS Parameters and Secrets Lambda 延伸模組新增為 layer

1. 在函式的程式碼索引標籤中，向下捲動至層。
2. 選擇 Add a layer (新增 layer)。
3. 選擇 AWS 層。
4. 選擇 AWS-Parameters-and-Secrets-Lambda-Extension。
5. 選擇最新版本。
6. 選擇新增。

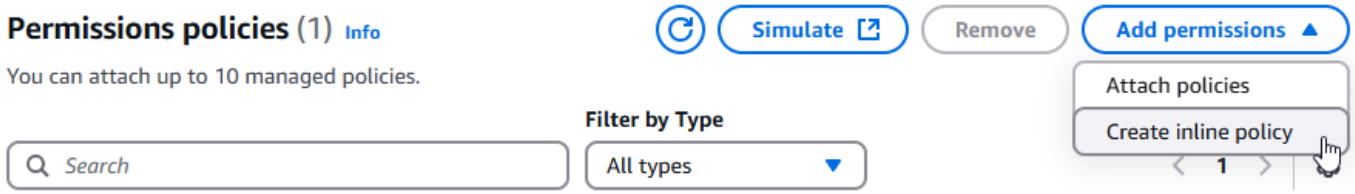
新增許可

若要將 Secrets Manager 許可新增至執行角色

1. 依序選擇 Configuration (組態) 索引標籤和 Permissions (許可)。
2. 在角色名稱下面，選擇執行角色連結。此連結會在 IAM 主控台中開啟該角色。



3. 選擇新增許可，然後選擇建立內嵌政策。



4. 選擇 JSON 索引標籤，並新增下列政策。在 `Resource` 欄位中輸入秘密的 ARN。

JSON

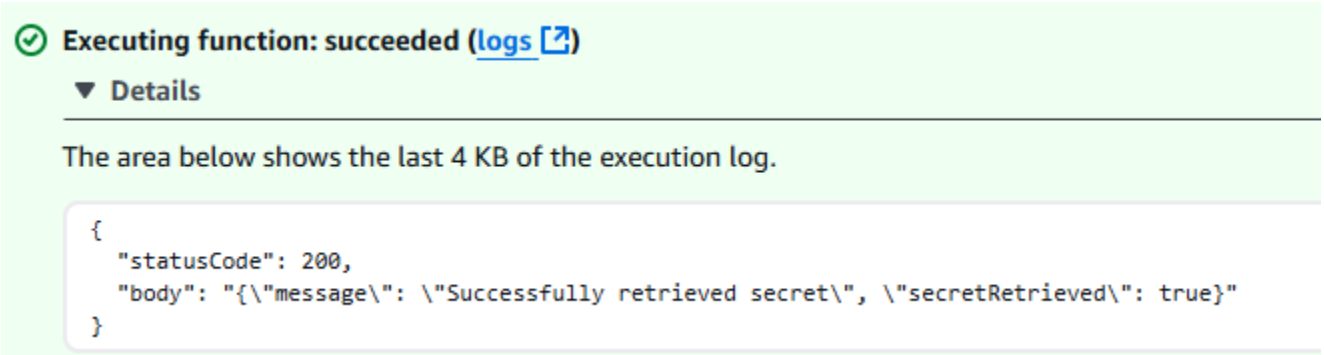
```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "secretsmanager:GetSecretValue",
            "Resource": "arn:aws:secretsmanager:us-
east-1:111122223333:secret:SECRET_NAME"
        }
    ]
}
```

5. 選擇下一步。
6. 輸入政策的名稱。
7. 選擇建立政策。

測試函數

若要測試函數

1. 返回 Lambda 主控台。
2. 選取測試索引標籤。
3. 選擇測試。您應該會看到下列回應：



環境變數

AWS Parameters and Secrets Lambda 延伸模組使用以下預設設定。您可以建立對應的環境變數來覆寫這些設定。若要檢視函式的目前設定，請將 PARAMETERS_SECRETS_EXTENSION_LOG_LEVEL 設定為 DEBUG。該延伸模組會在每個函式調用開始時，將其組態資訊記錄至 CloudWatch Logs 中。

設定	預設值	有效值	環境變數	詳細資訊
HTTP 連接埠	2773	1 - 65535	PARAMETERS_SECRETS_EXTENSION_HTTP_PORT	本機 HTTP 伺服器的連接埠
快取已啟用	TRUE	TRUE | FALSE	PARAMETERS_SECRETS_EXTENSION_CACHE_ENABLED	啟用或停用快取
快取大小	1000	0 - 1000	PARAMETERS_SECRETS_EXTENSION_CACHE_SIZE	設定為 0 以停用快取
Secrets Manager TTL	300 秒	0 - 300 秒	SECRETS_MANAGER_TTL	快取秘密的存留時間。設定為 0 以停用快取。如果 PARAMETERS_SECRETS_EXTENSION_CACHE_SIZE 的值為 0，則會忽略此變數。
Parameter Store TTL	300 秒	0 - 300 秒	SSM_PARAMETER_STORE_TTL	快取參數的存留時間。設定為 0 以停用快取。如果 PARAMETERS_SECRETS_EXTENSION_CACHE_SIZE 的值為 0，則會忽略此變數。
日誌層級	INFO	DEBUG | INFO | WARN |	PARAMETERS_SECRETS_EXTENSION_LOG_LEVEL	延伸日誌中報告的詳細資訊層級

設定	預設值	有效值	環境變數	詳細資訊
		ERROR | NONE		
連線數量上限	3	1 或以上	PARAMETERS_SECRETS_EXTENSION_MAX_CONNECTIONS	向 Parameter Store 或 Secrets Manager 發起請求的 HTTP 連線數量上限
Secrets Manager 逾時	0 (無逾時)	所有整數	SECRETS_MANAGER_TIMEOUT_MILLIS	向 Secrets Manager 發出之請求的逾時 (以毫秒為單位)
Parameter Store 逾時	0 (無逾時)	所有整數	SSM_PARAMETER_STORE_TIMEOUT_MILLIS	向 Parameter Store 發出之請求的逾時 (以毫秒為單位)

使用秘密輪換

如果經常輪換秘密，預設的 300 秒快取持續時間可能會導致函式使用過時的秘密。有兩種方式可確保函式使用最新的秘密值：

- 將 SECRETS_MANAGER_TTL 環境變數設定為較低值 (以秒為單位)，即可降低快取 TTL。例如，將其設定為 60，可確保函式絕不會使用超過一分鐘的舊秘密。
- 在秘密請求中使用 AWSCURRENT 或 AWSPREVIOUS 預備標籤，可確保取得所需的特定版本：

```
secretsmanager/get?secretId=YOUR_SECRET_NAME&versionStage=AWSCURRENT
```

請選擇最能平衡效能與新鮮度需求的方案。TTL 較低意味著會更頻繁地呼叫 Secrets Manager，但能確保您使用的是最新的秘密值。

從 Powertools for 使用參數公用程式 AWS Lambda

Powertools for 的參數公用程式 AWS Lambda 提供統一界面，用於從多個提供者擷取秘密，包括 Secrets Manager、參數存放區和 AppConfig。該方案會處理快取、轉換作業，且相較於延伸模組方案，能提供整合更緊密的開發體驗。

參數公用程式的優點

- 多個提供者 – 使用同一個介面從 Secrets Manager、Parameter Store 和 AppConfig 中擷取參數
- 內建轉換功能 – 自動 JSON 剖析、base64 解碼和其他資料轉換
- 整合式快取 – 具有 TTL 支援的可設定快取，可減少 API 呼叫次數
- 型別安全 – 在 TypeScript 與其他支援的執行時期中提供強型別支援
- 錯誤處理 – 內建重試邏輯和錯誤處理

程式碼範例

下列範例示範如何在不同的執行時期中使用參數公用程式擷取秘密：

Python

> **Note**
>
> 如需完整的範例和設定說明，請參閱 Parameters utility documentation。

使用 Powertools for AWS Lambda Parameters 公用程式從 Secrets Manager 擷取秘密。

```
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities import parameters

logger = Logger()

def lambda_handler(event, context):
    try:
        # Get secret with caching (default TTL: 5 seconds)
        secret_value = parameters.get_secret("my-secret-name")

        # Get secret with custom TTL
        secret_with_ttl = parameters.get_secret("my-secret-name", max_age=300)

        # Get secret and transform JSON
        secret_json = parameters.get_secret("my-json-secret", transform="json")

        logger.info("Successfully retrieved secrets")
```

```
        return {
            'statusCode': 200,
            'body': 'Successfully retrieved secrets'
        }

    except Exception as e:
        logger.error(f"Error retrieving secret: {str(e)}")
        return {
            'statusCode': 500,
            'body': f'Error: {str(e)}'
        }
```

TypeScript

> **Note**
>
> 如需完整的範例和設定說明，請參閱 Parameters utility documentation。

使用 Powertools for AWS Lambda Parameters 公用程式從 Secrets Manager 擷取秘密。

```
import { Logger } from '@aws-lambda-powertools/logger';
import { getSecret } from '@aws-lambda-powertools/parameters/secrets';
import type { Context } from 'aws-lambda';

const logger = new Logger();

export const handler = async (event: any, context: Context) => {
    try {
        // Get secret with caching (default TTL: 5 seconds)
        const secretValue = await getSecret('my-secret-name');

        // Get secret with custom TTL
        const secretWithTtl = await getSecret('my-secret-name', { maxAge: 300 });

        // Get secret and transform JSON
        const secretJson = await getSecret('my-json-secret', { transform: 'json' });

        logger.info('Successfully retrieved secrets');

        return {
            statusCode: 200,
            body: 'Successfully retrieved secrets'
```

```
        };

    } catch (error) {
        logger.error('Error retrieving secret', { error });
        return {
            statusCode: 500,
            body: `Error: ${error}`
        };
    }
};
```

Java

> **Note**
>
> 如需完整的範例和設定說明，請參閱 Parameters utility documentation。

使用 Powertools for AWS Lambda Parameters 公用程式從 Secrets Manager 擷取秘密。

```
import software.amazon.lambda.powertools.logging.Logging;
import software.amazon.lambda.powertools.parameters.SecretsProvider;
import software.amazon.lambda.powertools.parameters.ParamManager;
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class SecretHandler implements RequestHandler<Object, String> {

    private final SecretsProvider secretsProvider =
 ParamManager.getSecretsProvider();

    @Logging
    @Override
    public String handleRequest(Object input, Context context) {
        try {
            // Get secret with caching (default TTL: 5 seconds)
            String secretValue = secretsProvider.get("my-secret-name");

            // Get secret with custom TTL (300 seconds)
            String secretWithTtl = secretsProvider.withMaxAge(300).get("my-secret-
name");

            // Get secret and transform JSON
```

```
            MySecret secretJson = secretsProvider.get("my-json-secret",
 MySecret.class);

            return "Successfully retrieved secrets";

        } catch (Exception e) {
            return "Error retrieving secret: " + e.getMessage();
        }
    }

    public static class MySecret {
        // Define your secret structure here
    }
}
```

.NET

> **Note**
>
> 如需完整的範例和設定說明，請參閱 Parameters utility documentation。

使用 Powertools for AWS Lambda Parameters 公用程式從 Secrets Manager 擷取秘密。

```
using AWS.Lambda.Powertools.Logging;
using AWS.Lambda.Powertools.Parameters;
using Amazon.Lambda.Core;

[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeriali

public class Function
{
    private readonly ISecretsProvider _secretsProvider;

    public Function()
    {
        _secretsProvider = ParametersManager.SecretsProvider;
    }

    [Logging]
    public async Task<string> FunctionHandler(object input, ILambdaContext context)
    {
```

```
        try
        {
            // Get secret with caching (default TTL: 5 seconds)
            var secretValue = await _secretsProvider.GetAsync("my-secret-name");

            // Get secret with custom TTL
            var secretWithTtl = await
 _secretsProvider.WithMaxAge(TimeSpan.FromMinutes(5))
                .GetAsync("my-secret-name");

            // Get secret and transform JSON
            var secretJson = await _secretsProvider.GetAsync<MySecret>("my-json-
secret");

            return "Successfully retrieved secrets";
        }
        catch (Exception e)
        {
            return $"Error retrieving secret: {e.Message}";
        }
    }

    public class MySecret
    {
        // Define your secret structure here
    }
}
```

設定和許可

若要使用參數公用程式，您需要：

1. AWS Lambda 為您的執行時間安裝適用於 的 Powertools。如需詳細資訊，請參閱Powertools for AWS Lambda。
2. 將必要的 IAM 許可新增至函式的執行角色。請參閱 在 中管理許可 AWS Lambda 以取得詳細資訊。
3. 透過環境變數來進行選用設定。

所需的 IAM 許可與延伸模組方案相同。參數公用程式會根據您的組態自動處理對 Secrets Manager 的快取和 API 呼叫。

搭配 Amazon SQS 使用 Lambda

> **Note**
>
> 如要將資料傳送到 Lambda 函數以外的目標，或在傳送資料之前讓資料更豐富，請參閱 Amazon EventBridge Pipes。

您可以使用 Lambda 函數處理 Amazon Simple Queue Service (Amazon SQS) 佇列中的訊息。Lambda 事件來源映射既支援標準佇列，也支援先進先出 (FIFO) 佇列。您也可以使用佈建模式為 Amazon SQS 事件來源映射配置專用輪詢資源。Lambda 函數和 Amazon SQS 佇列必須位於相同的中 AWS 區域，雖然它們可以位於不同的 AWS 帳戶 中。

處理 Amazon SQS 訊息時，您需要實作部分批次回應邏輯，避免批次中部分訊息處理失敗時，已成功處理的訊息被重新執行。Powertools for 的 Batch Processor 公用程式可自動處理部分批次回應邏輯、縮短開發時間並改善可靠性，藉此 AWS Lambda 簡化此實作。

主題

- 了解 Amazon SQS 事件來源映射的輪詢和批次處理行為
- 搭配 Amazon SQS 事件來源映射使用佈建模式
- 設定 Amazon SQS 事件來源映射的佈建模式
- 標準佇列訊息事件範例
- FIFO 佇列訊息事件範例
- 建立和設定 Amazon SQS 事件來源映射
- 設定 SQS 事件來源映射的擴展行為
- 在 Lambda 中處理 SQS 事件來源的錯誤
- Amazon SQS 事件來源映射的 Lambda 參數
- 對 Amazon SQS 事件來源使用事件篩選
- 教學課程：搭配 Amazon SQS 使用 Lambda
- 教學課程：使用跨帳戶 Amazon SQS 佇列做為事件來源

了解 Amazon SQS 事件來源映射的輪詢和批次處理行為

使用 Amazon SQS 事件來源映射時，Lambda 會輪詢佇列，並在出現事件時同步調用函數。每個事件可以包含佇列中的一批訊息。Lambda 一次接收一個批次的這些事件，並為每個批次調用函數一次。當您的函數成功處理批次時，Lambda 會從佇列中刪除其訊息。

當 Lambda 接收到一個批次時，訊息會留在佇列中，但是在佇列的可見性逾時期間會變成隱藏。如果您的函數成功處理批次中的全部訊息，Lambda 會從佇列中刪除訊息。根據預設，如果函數在處理批次時遇到錯誤，則該批次中的所有訊息會在可見性逾時時間到了之後再次顯示在佇列中。因此，您的函數程式碼必須能夠多次處理相同的訊息，而不會產生副作用。

> **Warning**
>
> Lambda 事件來源映射至少會處理每個事件一次，而且可能會重複處理記錄。為避免與重複事件相關的潛在問題，強烈建議您讓函數程式碼具有等冪性。若要進一步了解，請參閱 AWS 知識中心中的如何使 Lambda 函數具有等冪性。

若要防止 Lambda 多次處理訊息，您可以設定事件來源映射，在函數回應中包含批次項目失敗，或者可以使用 DeleteMessage API 動作，在 Lambda 函數成功處理訊息時從佇列中移除訊息。

如需 Lambda 支援用於 SQS 事件來源映射的組態參數的詳細資訊，請參閱the section called “建立 SQS 事件來源映射”。

搭配 Amazon SQS 事件來源映射使用佈建模式

對於您需要微調事件來源映射輸送量的工作負載，可以使用佈建模式。在佈建模式中，可以定義佈建的事件輪詢器數量的下限和上限。這些佈建的事件輪詢器專用於您的事件來源映射，並且可以透過回應性自動擴展處理意外的訊息尖峰。使用佈建模式設定的 Amazon SQS 事件來源映射擴展速度快 3 倍（每分鐘最多 1，000 個並行調用），並支援比預設 Amazon SQS 事件來源映射功能高 16 倍的並行調用（最多 20，000 個並行調用）。我們建議您針對具有嚴格效能需求的 Amazon SQS 事件驅動型工作負載使用佈建模式，例如處理市場資料饋送的金融服務公司、提供即時個人化建議的電子商務平台，以及管理即時玩家互動的遊戲公司。使用佈建模式會產生額外費用。如需詳細的定價，請參閱 AWS Lambda 定價。

佈建模式中的每個事件輪詢器最多可處理每秒 1 MB 的輸送量、最多 10 個並行調用，或每秒最多 10 個 Amazon SQS 輪詢 API 呼叫。事件輪詢器 (MinimumPollers) 數目下限的接受值範圍介於 2 到 200 之間，預設值為 2。事件輪詢器 (MaximumPollers) 數目上限的接受值範圍介於 2 到 2，000 之間，預設值為 200。MaximumPollers 必須大於或等於 MinimumPollers。

判斷所需的事件輪詢器

若要預估在 SQS ESM 使用佈建模式時確保最佳訊息處理效能所需的事件輪詢器數量，請為您的應用程式收集下列指標：需要低延遲處理的每秒尖峰 SQS 事件數、平均 SQS 事件承載大小、平均 Lambda 函數持續時間，以及設定的批次大小。

首先，您可以使用下列公式，為您的工作負載預估事件輪詢器支援的每秒 SQS 事件數 (EPS)：

```
EPS per event poller =
        minimum(
            ceiling(1024 / average event size in KB),
            ceiling(10 / average function duration in seconds) * batch size,
            min(100, 10 * batch size)
                )
```

然後，您可以使用以下公式計算所需的最低輪詢器數量。此計算可確保您佈建足夠的容量來處理尖峰流量需求。

```
Required event pollers = (Peak number of events per second in Queue) / EPS per event
 poller
```

假設工作負載的預設批次大小為 10、平均事件大小為 3 KB、平均函數持續時間為 100 毫秒，且需要每秒處理 1，000 個事件。在此案例中，每個事件輪詢器每秒將支援約 100 個事件 (EPS)。因此，您應該將最低輪詢器設定為 10，以充分處理您的尖峰流量需求。如果您的工作負載具有相同的特性，但平均函數持續時間為 1 秒，則每個輪詢器將僅支援 10 個 EPS，要求您設定 100 個最低輪詢器，以低延遲支援每秒 1，000 個事件。

建議使用預設批次大小 10 或更高，以最大限度地提高佈建模式事件輪詢器的效率。較高的批次大小可讓每個輪詢器在每次調用時處理更多事件，以提高輸送量和成本效益。規劃事件輪詢器容量時，請考量潛在的流量峰值，並考慮將 minimumPollers 值稍高於計算的最小值，以提供緩衝區。此外，隨著時間的推移監控工作負載特性，因為訊息大小、函數持續時間或流量模式的變更可能需要調整事件輪詢器組態，以維持最佳效能和成本效益。為了精確規劃容量，我們建議您測試特定工作負載，以判斷每個事件輪詢器可以驅動的實際 EPS。

設定 Amazon SQS 事件來源映射的佈建模式

您可以使用主控台或 Lambda API，為 Amazon SQS 事件來源映射設定佈建模式。

設定現有 Amazon SQS 事件來源映射的佈建模式 （主控台）

1. 開啟 Lambda 主控台中的函數頁面。

2. 選擇具有您要為其設定佈建模式之 Amazon SQS 事件來源映射的函數。
3. 選擇組態，然後選擇觸發條件。
4. 選擇您要為其設定佈建模式的 Amazon SQS 事件來源映射，然後選擇編輯。
5. 在事件來源映射組態下，選擇設定佈建模式。
   - 針對最低事件輪詢器，輸入介於 2 到 200 之間的值。如果您未指定值，Lambda 會選擇預設值 2。
   - 針對事件輪詢器上限，輸入介於 2 到 2，000 之間的值。此值必須大於或等於事件輪詢器下限值。如果您沒有指定值，則 Lambda 會選擇預設值 200。
6. 選擇儲存。

您可以使用 中的 `ProvisionedPollerConfig` 物件，以程式設計方式設定佈建模式`EventSourceMappingConfiguration`。例如，下列 `UpdateEventSourceMapping` CLI 命令會設定 5 `MinimumPollers`的值，以及 100 `MaximumPollers`的值。

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{"MinimumPollers": 5, "MaximumPollers": 100}'
```

設定佈建模式後，可以透過監控 `ProvisionedPollers` 指標來觀察工作負載的事件輪詢器使用情況。如需詳細資訊，請參閱事件來源映射指標。

若要停用佈建模式並返回預設（隨需）模式，您可以使用下列 `UpdateEventSourceMapping` CLI 命令：

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --provisioned-poller-config '{}'
```

> **Note**
>
> 佈建模式無法與最大並行設定搭配使用。使用佈建模式時，您可以透過事件輪詢器的數量上限來控制並行上限。

如需設定佈建模式的詳細資訊，請參閱 the section called “建立映射”。

標準佇列訊息事件範例

Example Amazon SQS 訊息事件 (標準佇列)


```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {
                "myAttribute": {
                    "stringValue": "myValue",
                    "stringListValues": [],
                    "binaryListValues": [],
                    "dataType": "String"
                }
            },
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
            "awsRegion": "us-east-2"
        },
        {
            "messageId": "2e1424d4-f796-459a-8184-9c92662be6da",
            "receiptHandle": "AQEBzWwaftRI0KuVm4tP+/7q1rGgNqicHq...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082650636",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082650649"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:my-queue",
```

```
            "awsRegion": "us-east-2"
        }
    ]
}
```

依預設，Lambda 會一次輪詢佇列中最多 10 則訊息，並將該批次傳送給函數。為避免調用具有少量記錄的函數，您可設定批次間隔，讓事件來源緩衝記錄最長達五分鐘。調用函數之前，Lambda 會繼續從標準佇列輪詢訊息，直至批次間隔到期、達到調用承載大小配額，或達到設定的批次大小上限為止。

如果您使用的是批次時間範圍，並且您的 SQS 佇列包含非常低的流量，Lambda 可能會等到 20 秒，然後再調用您的函數。即使您將批次時間範圍設定為低於 20 秒也是如此。

> **Note**
>
> 在 Java 中，還原序列化 JSON 時可能會遇到 null 指標錯誤。這可能是由於 JSON 物件映射器對「Records」和「eventSourceARN」案例轉換的方式所致。

FIFO 佇列訊息事件範例

對於 FIFO 佇列，記錄包含與重複資料刪除和定序相關的其他屬性。

Example Amazon SQS 訊息事件 (FIFO 佇列)

```
{
    "Records": [
        {
            "messageId": "11d6ee51-4cc7-4302-9e22-7cd8afdaadf5",
            "receiptHandle": "AQEBBX8nesZEXmkhsmZeyIE8iQAMig7qw...",
            "body": "Test message.",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1573251510774",
                "SequenceNumber": "18849496460467696128",
                "MessageGroupId": "1",
                "SenderId": "AIDAIO23YVJENQZJOL4VO",
                "MessageDeduplicationId": "1",
                "ApproximateFirstReceiveTimestamp": "1573251510774"
            },
            "messageAttributes": {},
            "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
            "eventSource": "aws:sqs",
```

```
            "eventSourceARN": "arn:aws:sqs:us-east-2:123456789012:fifo.fifo",
            "awsRegion": "us-east-2"
        }
    ]
}
```

建立和設定 Amazon SQS 事件來源映射

若要使用 Lambda 處理 Amazon SQS 訊息，請正確設定佇列，然後建立 Lambda 事件來源映射。

主題

- 配置要搭配 Lambda 使用的佇列
- 設定 Lambda 執行角色許可
- 建立 SQS 事件來源映射

配置要搭配 Lambda 使用的佇列

如果您還沒有建立 Amazon SQS 佇列，請建立一個來做為 Lambda 函數的事件來源。Lambda 函數和 Amazon SQS 佇列必須位於相同的 中 AWS 區域，雖然它們可以位於不同的 AWS 帳戶 中。

為讓您的函數有時間處理每個記錄批次，請將來源佇列的可見性逾時設定為函數設定之逾時的至少六倍。萬一您的函數在處理前一個批次時遭到調節，額外的時間可允許 Lambda 重試。

> **Note**
>
> 函式的逾時必須小於或等於佇列的可見性逾時。當您建立或更新事件來源映射時，Lambda 會驗證此項要求，在函式逾時超過佇列的可見性逾時之時傳回錯誤。

根據預設，只要 Lambda 在處理批次的過程中遇到錯誤，則該批次中的所有訊息都會傳回佇列。在可見性逾時時間到了之後，這些訊息會再次對 Lambda 可見。您可以將事件來源映射設定為使用部分批次回應，從而僅將失敗的訊息傳回佇列。此外，如果函數多次處理訊息失敗，Amazon SQS 可將訊息傳送到無效字母佇列。我們建議將來源佇列的再驅動政策 的 `maxReceiveCount` 設定為 5 或更大。這讓 Lambda 在將失敗的訊息傳送到無效字母佇列之前，有機會重試幾次。

設定 Lambda 執行角色許可

AWSLambdaSQSQueueExecutionRole AWS 受管政策包含 Lambda 從 Amazon SQS 佇列讀取所需的許可。您可以將此受管政策新增至函數的執行角色。

或者，如果您使用的是加密佇列，也需要將下列許可新增至您的執行角色：

- kms:Decrypt

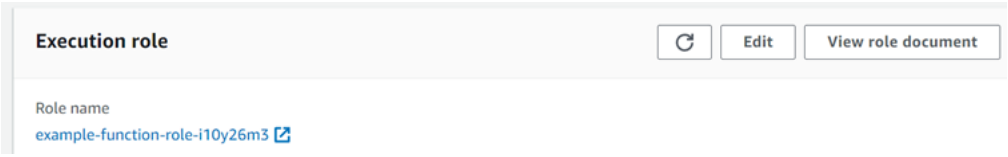
建立 SQS 事件來源映射

建立事件來源映射，指示 Lambda 從您的佇列傳送項目至 Lambda 函數。您可以建立多個事件來源映射，使用單一函數處理來自多個佇列的項目。當 Lambda 調用目標函數時，事件可能包含多個項目，最多為可設定的最大批次大小。

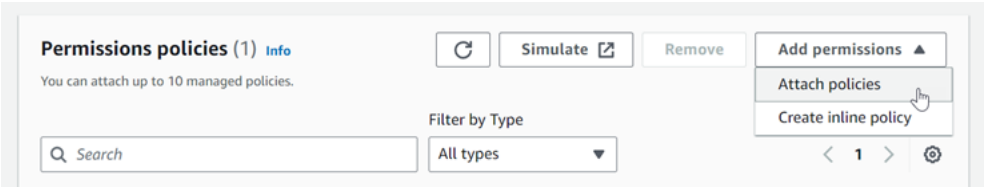
若要將函數設定為從 Amazon SQS 讀取，請將 AWSLambdaSQSQueueExecutionRole AWS 受管政策連接至您的執行角色。然後，透過以下步驟從主控台建立 SQS 事件來源映射。

若要新增許可並建立觸發條件

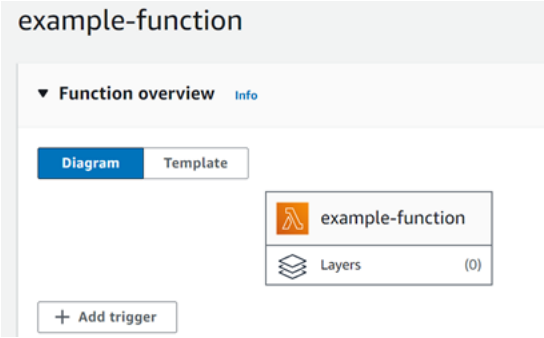
1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 依序選擇 Configuration (組態) 索引標籤和 Permissions (許可)。
4. 在角色名稱下面，選擇執行角色連結。此連結會在 IAM 主控台中開啟該角色。



5. 選擇新增許可，然後選擇連接政策。



6. 在搜尋欄位中輸入 AWSLambdaSQSQueueExecutionRole。將此政策新增至您的執行角色。這是 AWS 受管政策，其中包含函數從 Amazon SQS 佇列讀取所需的許可。如需此政策的詳細資訊，請參閱《AWS 受管政策參考》中的 AWSLambdaSQSQueueExecutionRole 一節。
7. 在 Lambda 主控台中返回您的 Lambda 函數。在函數概觀下，選擇新增觸發條件。



8. 選擇觸發條件類型。
9. 設定需要的選項，然後選擇新增。

Lambda 支援 Amazon SQS 事件來源的下列組態選項：

SQS 佇列

要從中讀取記錄的 Amazon SQS 佇列。Lambda 函數和 Amazon SQS 佇列必須位於相同的 中 AWS 區域，雖然它們可以位於不同的 AWS 帳戶 中。

啟用觸發條件

事件來源映射的狀態。系統會預設選取啟用觸發條件。

批次大小

每個批次中要傳送至函數的記錄數量上限。若是標準佇列，這最高可達 10,000 個記錄。若是 FIFO 佇列，最大值為 10。批次大小若超過 10，您還必須將批次間隔 (`MaximumBatchingWindowInSeconds`) 設定為至少 1 秒。

設定函數逾時以允許足夠時間來處理整個項目批次。如果項目需要長時間處理，請選擇較小的批次大小。大型批次大小可提升效率，適用非常快速或開銷龐大的工作負載。如果您在函數上設定保留並行，應最少設定五個並行執行，藉此降低 Lambda 調用函數時的調節錯誤機率。

Lambda 會將批次中所有記錄以單一呼叫傳送至函數，前提是事件的總大小不超過同步調用的調用承載大小配額 (6 MB)。Lambda 和 Amazon SQS 兩者均會產生每筆記錄的中繼資料。這個額外的中繼資料會計入總酬載大小，並且可能會導致批次中傳送的記錄總數低於您設定的批次大小。Amazon SQS 傳送的中繼資料欄位長度可能有所不同。如需 Amazon SQS 中繼資料欄位的詳細資訊，請參閱 Amazon Simple Queue Service API 參考中的 ReceiveMessage API 操作文件。

批次視窗

調用函式前收集記錄的最長時間 (單位為秒)。這僅適用於標準佇列。

如果您使用的批次間隔大於 0 秒，則必須考慮佇列可見性逾時中增加的處理時間。我們建議將您的佇列可見性逾時設定為函數逾時的六倍，再加上 `MaximumBatchingWindowInSeconds` 的值。這可讓您的 Lambda 函數有時間處理每個事件批次，並在發生節流錯誤時重試。

當訊息可供使用，Lambda 會開始批次處理訊息。Lambda 會開始一次處理五個批次，而函數有五個並行調用。如果訊息仍然可用，Lambda 每分鐘會為您的函數新增最多 300 個並行調用，最多 1，250 個並行調用。使用佈建模式時，每個事件輪詢器最多可處理每秒 1 MB 的輸送量、最多 10 個並行調用，或每秒最多 10 個 Amazon SQS 輪詢 API 呼叫。Lambda 會在設定的最小值和最大值之間擴展事件輪詢器的數量，快速新增每分鐘最多 1，000 個並行調用，以提供 Amazon SQS 事件的低延遲處理。您可以透過這些最小和最大事件輪詢器設定來控制擴展和並行。若要進一步了解函數擴展和並行，請參閱 了解 Lambda 函數擴展。

若要處理更多訊息，您可以最佳化 Lambda 函數以達到更高的輸送量。如需詳細資訊，請參閱了解如何使用 Amazon SQS 標準佇列 AWS Lambda 擴展。

篩選條件

新增篩選條件來控制 Lambda 要傳送哪些事件給函數進行處理。如需詳細資訊，請參閱控制 Lambda 將哪些事件傳送至您的函數。

並行上限

事件來源可以調用的並行函數上限。無法與已啟用佈建模式搭配使用。如需詳細資訊，請參閱設定 Amazon SQS 事件來源的並行上限。

佈建模式

啟用時， 會為您的事件來源映射配置專用輪詢資源。您可以設定事件輪詢器的最小 (2-200) 和最大 (2-2000) 數量。每個事件輪詢器最多可以處理每秒 1 MB 的輸送量、最多 10 個並行調用，或每秒最多 10 個 Amazon SQS 輪詢 API 呼叫。

> **Note**
>
> 注意：您無法同時使用佈建模式和最大並行。啟用佈建模式時，請使用最大輪詢器設定來控制並行。

設定 SQS 事件來源映射的擴展行為

您可以透過最大並行設定或啟用佈建模式來控制 Amazon SQS 事件來源映射的擴展行為。這些是互斥選項。

根據預設，Lambda 會根據訊息磁碟區自動擴展事件輪詢器。當您啟用佈建模式時，您可以配置最小和最大數量的專用輪詢資源，這些資源仍然準備好處理預期的流量模式。這可讓您以兩種方式最佳化事件來源映射的效能：

- 標準模式（預設）：Lambda 會自動管理擴展，從少量輪詢器開始，並根據工作負載向上或向下擴展。
- 佈建模式：您可以設定具有最小和最大限制的專用輪詢資源，使擴展速度加快 3 倍，處理容量提高 16 倍。

針對標準佇列，Lambda 會使用長輪詢來輪詢佇列，直至其處於作用中狀態。當訊息可用時，Lambda 會開始一次處理五個批次，而函數有五個並行調用。如果訊息仍然可用，Lambda 會增加讀取批次的程序數量，每分鐘最多增加 300 個並行調用。事件來源映射可同時處理的調用數目上限為 1，250。當流量較低時，Lambda 會將處理縮減為五個並行調用，並且可以最佳化為最少 2 個並行調用，以減少 Amazon SQS 呼叫和對應的成本。然而，如果啟用並行上限設定，將無法使用此最佳化。

針對 FIFO 佇列，Lambda 會按照其接收的順序傳送訊息到您的函數。當您傳送訊息到 FIFO 佇列時，您可以指定訊息群組 ID。Amazon SQS 可確保相同群組中的訊息依序傳遞至 Lambda。當 Lambda 將訊息讀取到批次時，每個批次可能包含來自多個訊息群組的訊息，但訊息的順序會保持不變。如果您的函數傳回錯誤，函數會先在受影響的訊息上嘗試所有重試，之後 Lambda 才會從相同群組接收到額外訊息。

使用佈建模式時，每個事件輪詢器最多可以處理每秒 1 MB 的輸送量、最多 10 個並行調用，或每秒最多 10 個 Amazon SQS 輪詢 API 呼叫。Lambda 會在您設定的最小值和最大值之間擴展事件輪詢器數量，快速新增高達每分鐘 1，000 個並行，以提供 Amazon SQS 事件的一致、低延遲處理。使用佈建模式會產生額外費用。如需詳細的定價，請參閱 AWS Lambda 定價。每個事件輪詢器對您的 SQS 佇列使用長輪詢，每秒最多 10 個輪詢，這會產生 SQS API 請求成本。如需詳細資訊，請參閱 Amazon SQS 定價。您可以透過這些最小和最大事件輪詢器設定來控制擴展和並行，而不是使用最大並行設定，因為這些選項無法一起使用。

> **Note**
>
> 您無法同時使用最大並行設定和佈建模式。啟用佈建模式時，您可以透過事件輪詢器的最小和最大數量來控制 Amazon SQS 事件來源映射的擴展和並行。

設定 Amazon SQS 事件來源的並行上限

您可以使用最大並行設定來控制 SQS 事件來源的擴展行為。請注意，最大並行不能在已啟用佈建模式的情況下使用。並行上限設定限制了 Amazon SQS 事件來源可以調用的函數並行執行個體數。並行上限是事件來源層級的設定。如果您將多個 Amazon SQS 事件來源映射到一個函數，那麼每個事件來源都可以有個別的並行上限設定。您可以使用並行上限來防止一個佇列用完函數的所有預留並行配額，或其餘的帳戶並行配額。對 Amazon SQS 事件來源設定並行無需付費。

重要的是，並行上限和預留並行是兩項獨立的設定。請勿將並行上限設為超過函數的預留並行。若您設定了並行上限，請確定函數的預留並行大於或等於函數上所有 Amazon SQS 事件來源的總並行上限。若小於此上限，Lambda 可能會限流您的訊息。

當您帳戶的並行配額設定為預設值 1,000 時，除非您指定最大並行，否則 Amazon SQS 事件來源映射最高可以擴展至調用數量為該值的函數執行個體。

即便帳戶的預設並行配額增加，Lambda 也可能無法調用數量為新配額的並行函數執行個體。根據預設，對於一個 Amazon SQS 事件來源映射，Lambda 可以擴展為調用最多 1,250 個並行函數執行個體。如果這不足以滿足您的使用案例，請聯絡 AWS 支援，以討論增加您帳戶的 Amazon SQS 事件來源映射並行。

> **Note**
>
> 對於 FIFO 佇列，並行調用的上限是訊息群組 ID (`messageGroupId`) 的數量或最大並行設定 (以較低者為準)。例如，如果您有六個訊息群組 ID，而最大並行設定為 10，則函數最多可以有六個並行調用。

您可以對新的和現有的 Amazon SQS 事件來源映射設定並行上限。

使用 Lambda 主控台設定並行上限

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 在函數概觀下，選擇 SQS。這會開啟 Configuration (組態) 索引標籤。
4. 選取 Amazon SQS 觸發條件，然後選擇編輯。
5. Maximum concurrency (並行上限) 請輸入介於 2 到 1,000 之間的數字。若要關閉並行上限，請將方塊保留空白。
6. 選擇儲存。

使用 AWS Command Line Interface (AWS CLI) 設定並行上限

使用 update-event-source-mapping 命令和 `--scaling-config` 選項。範例：

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config '{"MaximumConcurrency":5}'
```

若要關閉並行上限，請為 `--scaling-config` 輸入空值：

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --scaling-config "{}"
```

使用 Lambda API 設定並行上限

使用 CreateEventSourceMapping 或UpdateEventSourceMapping 動作搭配 ScalingConfig 物件。

在 Lambda 中處理 SQS 事件來源的錯誤

為了處理與 SQS 事件來源相關的錯誤，Lambda 會自動使用重試策略搭配退避策略。您也可以透過設定 SQS 事件來源映射，來自訂錯誤處理行為，以傳回部分批次回應。

失敗調用的輪詢策略

當調用失敗，Lambda 會在實作輪詢策略時嘗試重試調用。根據 Lambda 是否因函數程式碼錯誤或限流而發生失敗，輪詢策略會略有不同。

- 如果是函數程式碼導致錯誤，則 Lambda 會停止處理並重試調用。同時，Lambda 會逐漸退避，減少分配給 Amazon SQS 事件來源映射的並行數量。在佇列的可見性逾時時間到了之後，訊息會再次重新出現在佇列中。
- 如果是限流導致調用失敗，Lambda 會減少分配給 Amazon SQS 事件來源映射的並行數量，逐漸重試輪詢。Lambda 會持續重試訊息，直到訊息的時間戳記超過佇列的可見性逾時為止，此時 Lambda 會捨棄訊息。

實作部分批次回應

當 Lambda 函數在處理批次時遇到錯誤，根據預設，該批次中的所有訊息會再次顯示在佇列中，包含 Lambda 已順利處理的訊息。因此，您的函數可能最後會處理數次相同的訊息。

若要避免重新處理失敗批次中成功處理過的訊息，您可以設定事件來源映射，僅讓失敗的訊息再次可見。我們將其稱為部分批次回應。若要開啟部分批次回應，請在設定事件來源映射時為 FunctionResponseTypes 動作指定 `ReportBatchItemFailures`。這可以讓您的函數傳回部分成功，有助於減少記錄上不必要的重試次數。

> **Note**
>
> 適用於 的 Powertools 批次處理器公用程式會自動 AWS Lambda 處理所有部分批次回應邏輯。此公用程式簡化了批次處理模式的實作流程，也減少了正確處理批次項目失敗所需的自訂程式碼。該工具組支援 Python、Java、Typescript 與 .NET。

啟動 `ReportBatchItemFailures` 時，Lambda 不會在函數調用失敗時縮減訊息輪詢的規模。如果您預期部分訊息會失敗，且不希望這些失敗影響到訊息的處理速度，則請使用 `ReportBatchItemFailures`。

> **Note**
>
> 使用部分批次回應時，請注意下列事項：
>
> - 如果函數擲出例外情況，便會將整個批次視為完全失敗。
> - 如果您將此功能與 FIFO 佇列一起使用，您的函數應該在第一次失敗後停止處理訊息，並傳回 `batchItemFailures` 中所有失敗與尚未處理的訊息。這有助於保留佇列中訊息的順序。

若要啟動部分批次報告

1. 檢閱實作部分批次回應的最佳實務。
2. 執行以下命令以便為函數啟用 `ReportBatchItemFailures`。若要擷取事件來源映射的 UUID，請執行 list-event-source-mappings AWS CLI 命令。

```
aws lambda update-event-source-mapping \
--uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
--function-response-types "ReportBatchItemFailures"
```

3. 更新函數程式碼以擷取所有例外狀況，並傳回 `batchItemFailures` JSON 回應中的失敗訊息。`batchItemFailures` 回應必須含有以 `itemIdentifier` JSON 值表示的訊息 ID 清單。

例如，假設您有五則訊息的批次，其中，訊息 ID 分別為 id1、id2、id3、id4，以及 id5。您的函數已成功處理 id1、id3，以及 id5。若要讓訊息 id2 和 id4 再次於佇列中可見，您的函數應傳回以下回應：

```
{
  "batchItemFailures": [
        {
            "itemIdentifier": "id2"
        },
        {
            "itemIdentifier": "id4"
        }
    ]
}
```

以下範例函數程式碼會傳回批次中失敗訊息 ID 的清單：

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be
 converted into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJson
namespace sqsSample;
```

```
public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
            throw new Exception("No Body in SQS Message.");
        }
        context.Logger.LogInformation($"Processed message {message.Body}");
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {
  if len(message.Body) > 0 {
   // Your message processing condition here
   fmt.Printf("Successfully processed message: %s\n", message.Body)
  } else {
   // Message processing failed
   fmt.Printf("Failed to process message %s\n", message.MessageId)
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
  }
 }

 sqsBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
```

```
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context)
 {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<SQSBatchResponse.BatchItemFailure>();

        for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
            try {
```

```
                //process your message
            } catch (Exception e) {
                //Add failed message identifier to the batchItemFailures
 list
                batchItemFailures.add(new
 SQSBatchResponse.BatchItemFailure(message.getMessageId()));
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Node.js 20.x Lambda runtime, AWS SDK for Javascript V3
export const handler = async (event, context) => {
    const batchItemFailures = [];
    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }
    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
    }
```

```
    console.log(`Processed message: ${record.body}`);
}
```

使用 TypeScript 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure,
 SQSRecord } from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();

        foreach ($records as $record) {
```

```
            try {
                // Assuming the SQS message is in JSON format
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS
 records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 報告 SQS 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}
```

```
        for record in event["Records"]:
            try:
                print(f"Processed message: {record['body']}")
            except Exception as e:
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 報告 SQS 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
    return sqs_batch_response
```

```
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
```

```
        batch_item_failures,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(function_handler)).await
}
```

如果失敗的事件未傳回佇列，請參閱 AWS 知識中心中的如何疑難排解 Lambda 函數 SQS ReportBatchItemFailures？。

成功與失敗條件

如果您的函數傳回下列任一項目，Lambda 會將批次視為完全成功：

- 空白 `batchItemFailures` 清單
- Null `batchItemFailures` 清單
- 空白 `EventResponse`
- Null `EventResponse`

如果您的函數傳回下列任一項目，Lambda 會將批次視為完全失敗：

- 無效的 JSON 回應
- 空白字串 `itemIdentifier`
- Null `itemIdentifier`
- 具有錯誤金鑰名稱的 `itemIdentifier`
- 具有不存在之訊息 ID 的 `itemIdentifier` 值

CloudWatch 指標

若要判斷您的函數是否正確報告批次項目失敗，您可以在 Amazon CloudWatch 中監控 `NumberOfMessagesDeleted` 和 `ApproximateAgeOfOldestMessage` Amazon SQS 指標。

- `NumberOfMessagesDeleted` 會追蹤從佇列移除的訊息數目。如果下降到 0，表示您的函數回應並未正確傳回失敗訊息。

- `ApproximateAgeOfOldestMessage` 會追蹤最舊訊息停留在佇列中的時間長度。此指標的急劇增加可能表示您的函數並未正確傳回失敗訊息。

使用 Powertools 進行 AWS Lambda 批次處理器

Powertools for 的批次處理器公用程式 AWS Lambda 會自動處理部分批次回應邏輯，降低實作批次失敗報告的複雜性。以下是使用批次處理器的範例：

Python

> **Note**
>
> 如需完整的範例和設定說明，請參閱 batch processor documentation。

使用 AWS Lambda 批次處理器處理 Amazon SQS 訊息。

```
import json
from aws_lambda_powertools import Logger
from aws_lambda_powertools.utilities.batch import BatchProcessor, EventType,
 process_partial_response
from aws_lambda_powertools.utilities.data_classes import SQSEvent
from aws_lambda_powertools.utilities.typing import LambdaContext

processor = BatchProcessor(event_type=EventType.SQS)
logger = Logger()

def record_handler(record):
    logger.info(record)
    # Your business logic here
    # Raise an exception to mark this record as failed

def lambda_handler(event, context: LambdaContext):
    return process_partial_response(
        event=event,
        record_handler=record_handler,
        processor=processor,
        context=context
    )
```

TypeScript

> **Note**
>
> 如需完整的範例和設定說明，請參閱 batch processor documentation。

使用 AWS Lambda 批次處理器處理 Amazon SQS 訊息。

```
import { BatchProcessor, EventType, processPartialResponse } from '@aws-lambda-
powertools/batch';
import { Logger } from '@aws-lambda-powertools/logger';
import type { SQSEvent, Context } from 'aws-lambda';

const processor = new BatchProcessor(EventType.SQS);
const logger = new Logger();

const recordHandler = async (record: any): Promise<void> => {
    logger.info('Processing record', { record });
    // Your business logic here
    // Throw an error to mark this record as failed
};

export const handler = async (event: SQSEvent, context: Context) => {
    return processPartialResponse(event, recordHandler, processor, {
        context,
    });
};
```

Amazon SQS 事件來源映射的 Lambda 參數

所有 Lambda 事件來源類型都會共用相同的 CreateEventSourceMapping 和 UpdateEventSourceMapping API 操作。但是，只有一些參數適用於 Amazon SQS。

參數	必要	預設	備註
BatchSize	否	10	對於標準佇列，最大值為 10,000。對於

參數	必要	預設	備註
			FIFO 隊列，最大值為 10。
已啟用	N	true	無
EventSourceArn	Y	N/A	資料串流或串流消費者的 ARN
FunctionName	是	N/A	無
FilterCriteria	N	N/A	控制 Lambda 將哪些事件傳送至您的函數
FunctionResponseTypes	N	N/A	若要讓函數報告批次中的特定失敗，請將值 `ReportBatchItemFailures` 包含在 `FunctionResponseTypes` 中。如需詳細資訊，請參閱實作部分批次回應。
MaximumBatchingWindowInSeconds	N	0	FIFO 佇列不支援批次間隔
ProvisionedPollerConfig	N	N/A	設定 SQS 事件來源映射的專用事件輪詢器數量下限 (2-200) 和上限 (2-2000)。每個輪詢器最多可處理每秒 1 MB 的輸送量和 10 個並行調用。
ScalingConfig	N	N/A	設定 Amazon SQS 事件來源的並行上限

對 Amazon SQS 事件來源使用事件篩選

您可以使用事件篩選來控制 Lambda 將哪些記錄從串流或佇列中傳送至函數。如需事件篩選運作方式的一般資訊，請參閱the section called “事件篩選”。

本節重點介紹了 Amazon SQS 事件來源的事件篩選。

> **Note**
>
> Amazon SQS 事件來源映射僅支援依據 body 鍵進行篩選。

主題

- Amazon SQS 事件篩選基本概念

Amazon SQS 事件篩選基本概念

假設您的 Amazon SQS 佇列包含以下 JSON 格式的訊息。

```
{
    "RecordNumber": 1234,
    "TimeStamp": "yyyy-mm-ddThh:mm:ss",
    "RequestCode": "AAAA"
}
```

此佇列的範例記錄如下所示。

```
{
    "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
    "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
    "body": "{\n "RecordNumber": 1234,\n "TimeStamp": "yyyy-mm-ddThh:mm:ss",\n
 "RequestCode": "AAAA"\n}",
    "attributes": {
        "ApproximateReceiveCount": "1",
        "SentTimestamp": "1545082649183",
        "SenderId": "AIDAIENQZJOLO23YVJ4VO",
        "ApproximateFirstReceiveTimestamp": "1545082649185"
        },
    "messageAttributes": {},
    "md5OfBody": "e4e68fb7bd0e697a0ae8f1bb342846b3",
    "eventSource": "aws:sqs",
```

```
    "eventSourceARN": "arn:aws:sqs:us-west-2:123456789012:my-queue",
    "awsRegion": "us-west-2"
}
```

若要根據 Amazon SQS 訊息的內容進行篩選，請使用 Amazon SQS 訊息記錄中的 body 金鑰。假設您只想處理 Amazon SQS 訊息中的 `RequestCode` 為 "BBBB" 的記錄。`FilterCriteria` 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RequestCode\" : [ \"BBBB\" ] } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 `Pattern` 在純文字 JSON 中擴展的值。

```
{
    "body": {
        "RequestCode": [ "BBBB" ]
        }
}
```

您可以使用主控台、AWS CLI 或 AWS SAM 範本新增篩選條件。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "body" : { "RequestCode" : [ "BBBB" ] } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 來建立具有這些篩選條件標準的新事件來源映射，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
```

```
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RequestCode\" :
 [ \"BBBB\" ] } }"}]}'
```

AWS SAM

若要使用 AWS SAM 新增此篩選條件，請將下列程式碼片段新增到事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RequestCode" : [ "BBBB" ] } }'
```

假設您希望函數僅處理 RecordNumber 大於 9999 的記錄。FilterCriteria 物件如下所示。

```
{
    "Filters": [
        {
            "Pattern": "{ \"body\" : { \"RecordNumber\" : [ { \"numeric\": [ \">\",
 9999 ] } ] } }"
        }
    ]
}
```

補充說明，此處是篩選條件的 Pattern 在純文字 JSON 中擴展的值。

```
{
    "body": {
        "RecordNumber": [
            {
                "numeric": [ ">", 9999 ]
            }
        ]
    }
}
```

您可以使用主控台、AWS CLI 或 AWS SAM 範本新增篩選條件。

Console

若要使用主控台新增此篩選條件，請遵循 將篩選條件標準連接至事件來源映射 (主控台) 中的指示，並針對篩選條件標準輸入下列字串。

```
{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }
```

AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 來建立具有這些篩選條件標準的新事件來源映射，請執行下列命令。

```
aws lambda create-event-source-mapping \
    --function-name my-function \
    --event-source-arn arn:aws:sqs:us-east-2:123456789012:my-queue \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

若要將這些篩選條件標準新增到現有事件來源映射，請執行下列命令。

```
aws lambda update-event-source-mapping \
    --uuid "a1b2c3d4-5678-90ab-cdef-11111EXAMPLE" \
    --filter-criteria '{"Filters": [{"Pattern": "{ \"body\" : { \"RecordNumber\" :
 [ { \"numeric\": [ \">\", 9999 ] } ] } }"}]}'
```

AWS SAM

若要使用 AWS SAM 新增此篩選條件，請將下列程式碼片段新增到事件來源的 YAML 範本。

```
FilterCriteria:
  Filters:
    - Pattern: '{ "body" : { "RecordNumber" : [ { "numeric": [ ">", 9999 ] } ] } }'
```

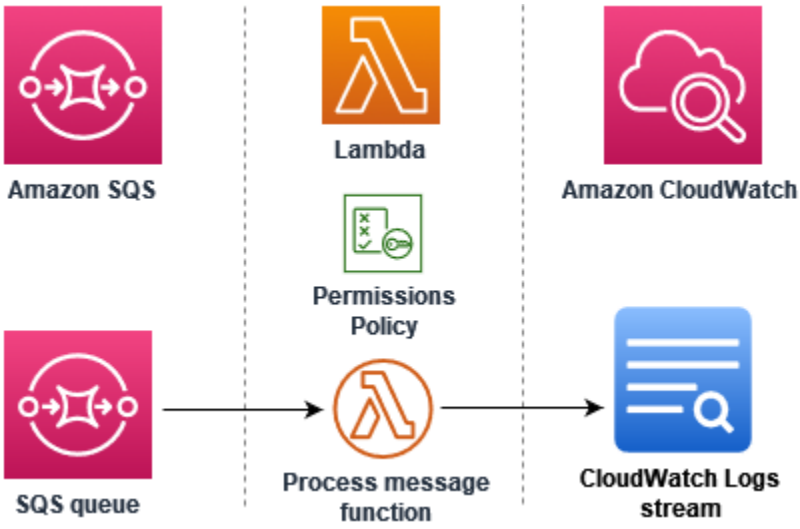
針對 Amazon SQS，訊息內文可以是任何字串。但是，如果您的 `FilterCriteria` 預期 body 為有效的 JSON 格式，這就會造成問題。反之亦然 — 如果傳入的訊息內文是有效的 JSON 格式，但您的選條件標準預期 body 應為純字串，則此可能會導致意外的行為。

為了避免此問題，請確定 `FilterCriteria` 中的內文格式與從佇列收到的訊息中的 `body` 之預期格式相符。篩選訊息之前，Lambda 會自動評估傳入訊息內文之格式和 `body` 的篩選條件模式之格式。如果有不相符的情形，Lambda 就會捨棄訊息。下表摘要說明此評估：

傳入訊息 **body** 格式	篩選條件模式 **body** 格式	產生的動作
純文字的字串	純文字的字串	根據您的篩選條件標準之 Lambda 篩選條件。
純文字的字串	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
純文字的字串	有效的 JSON	Lambda 捨棄訊息。
有效的 JSON	純文字的字串	Lambda 捨棄訊息。
有效的 JSON	資料屬性沒有篩選條件模式	Lambda 篩選條件 (僅限其他中繼資料屬性) 會根據您的篩選條件標準而定。
有效的 JSON	有效的 JSON	根據您的篩選條件標準之 Lambda 篩選條件。

教學課程：搭配 Amazon SQS 使用 Lambda

在本教學課程中，您將建立一個 Lambda 函數，該函數取用 Amazon Simple Queue Service (Amazon SQS) 佇列中的訊息。只要有新訊息加入佇列，Lambda 函數就會執行。函數會將訊息寫入 Amazon CloudWatch Logs 串流。下圖顯示可用來完成教學課程的 AWS 資源。



請執行下列步驟以完成本教學課程：

1. 建立將訊息寫入 CloudWatch Logs 的 Lambda 函數。
2. 建立 Amazon SQS 佇列。
3. 建立 Lambda 事件來源映射。事件來源映射會讀取 Amazon SQS 佇列，並在新增訊息時調用 Lambda 函數。
4. 透過將訊息新增至佇列並監控 CloudWatch Logs 中的結果來測試設定。

先決條件

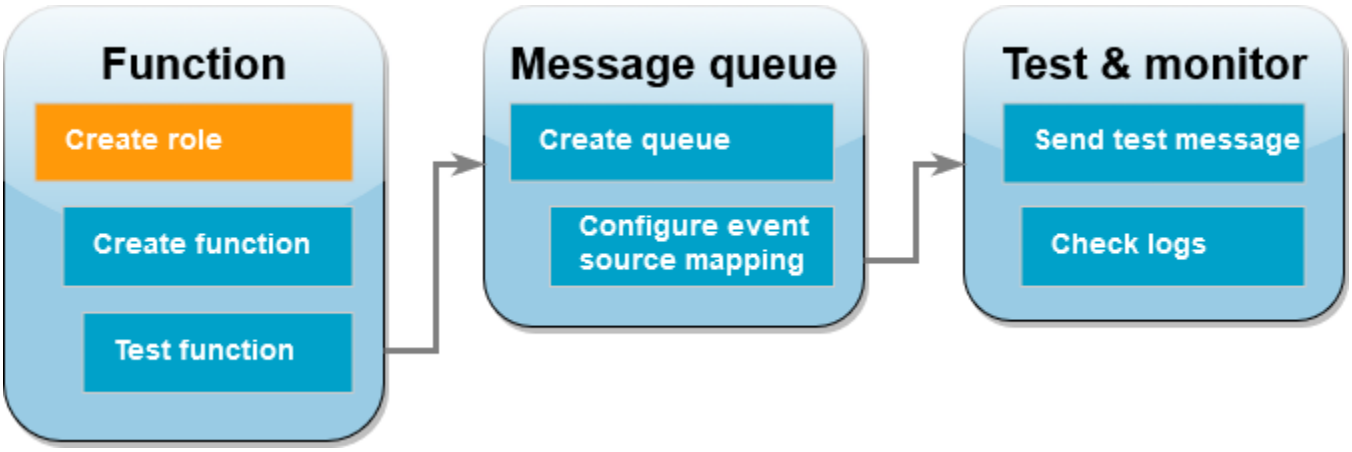
安裝 AWS Command Line Interface

如果您尚未安裝 AWS Command Line Interface，請依照安裝或更新最新版本的 AWS CLI中的步驟進行安裝。

本教學課程需使用命令列終端機或 Shell 來執行命令。在 Linux 和 macOS 中，使用您偏好的 Shell 和套件管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。

建立執行角色



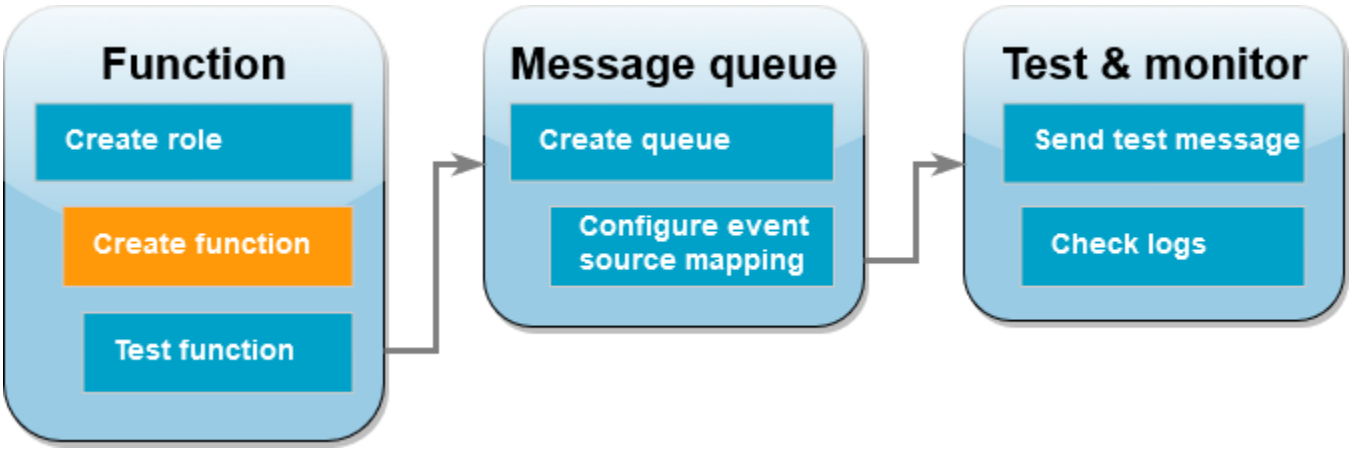
執行角色是授予 Lambda 函數存取 AWS 服務 和資源許可的 AWS Identity and Access Management (IAM) 角色。若要允許函數從 Amazon SQS 中讀取項目，請連接 AWSLambdaSQSQueueExecutionRole 許可政策。

建立執行角色並連接 Amazon SQS 許可政策

1. 開啟 IAM 主控台中的角色頁面。
2. 選擇 建立角色。
3. 針對信任的實體類型，請選擇 AWS 服務。
4. 針對使用案例，請選擇 Lambda。
5. 選擇下一步。
6. 在許可政策搜尋方塊中，輸入 **AWSLambdaSQSQueueExecutionRole**。
7. 選取 AWSLambdaSQSQueueExecutionRole 政策，然後選擇下一步。
8. 針對角色詳細資訊下的角色名稱，請輸入 **lambda-sqs-role**，然後選擇建立角色。

角色建立後，請記下執行角色的 Amazon Resource Name (ARN)。在後續步驟中會需要用到它。

建立函數



建立 Lambda 函數，它處理 Amazon SQS 訊息。函數程式碼會將 Amazon SQS 的訊息內文記錄到 Amazon CloudWatch Logs。

本教學課程使用 Node.js 24 執行時間，但我們也提供其他執行時間語言的範例程式碼。您可以在下列方塊中選取索引標籤，查看您感興趣的執行期程式碼。此步驟要使用的 JavaScript 程式碼，位於 JavaScript 索引標籤中顯示的第一個範例。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri
```

```
namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }
```

```
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

使用 TypeScript 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";
```

```
export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
```

```
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 SQS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來使用 SQS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
```

```
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
```

```
        .init();

    run(service_fn(function_handler)).await
}
```

若要建立 Node.js Lambda 函數

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir sqs-tutorial
cd sqs-tutorial
```

2. 將範例 JavaScript 程式碼複製到名為 `index.js` 的新檔案。
3. 使用以下 `zip` 命令建立部署套件。

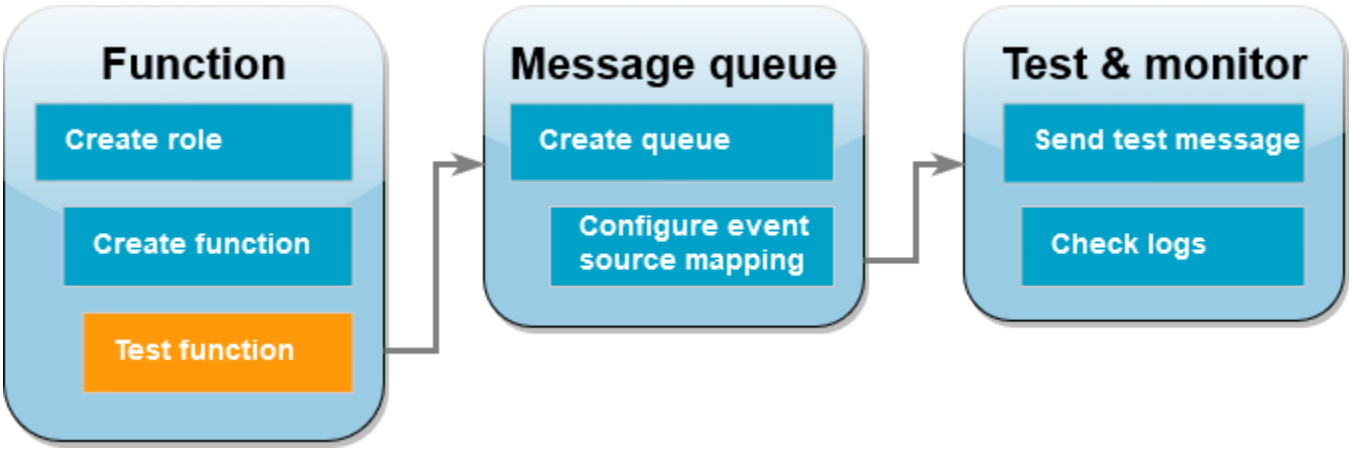
```
zip function.zip index.js
```

4. 使用 create-function AWS CLI 命令建立 Lambda 函數。針對 `role` 參數，輸入您之前建立的執行角色 ARN。

> **Note**
>
> 該 Lambda 函數和 Amazon SQS 佇列必須位於相同的 AWS 區域。

```
aws lambda create-function --function-name ProcessSQSRecord \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
--role arn:aws:iam::111122223333:role/lambda-sqs-role
```

測試函數



使用 invoke AWS CLI 命令和範例 Amazon SQS 事件手動叫用 Lambda 函數。

使用範例事件調用 Lambda 函數

1. 將下面的 JSON 儲存為名為 input.json 的檔案。此 JSON 會模擬 Amazon SQS 可能傳送至 Lambda 函數的事件，其中 "body" 包含佇列中的實際訊息。在此範例中，訊息為 "test"。

   Example Amazon SQS 事件

   這是測試事件，您不需要變更訊息或帳號。

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "test",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:my-queue",
            "awsRegion": "us-east-1"
        }
    ]
```

```
}
```

2. 執行下列叫用 AWS CLI 命令。此命令在回應中傳回 CloudWatch 日誌。如需擷取日誌的詳細資訊，請參閱使用 存取日誌 AWS CLI。

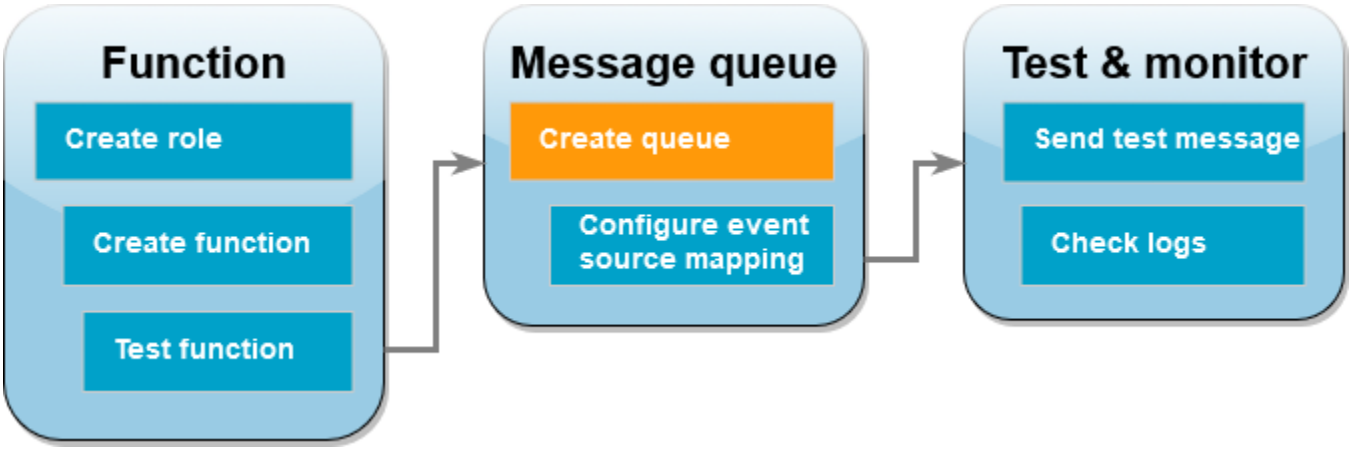
```
aws lambda invoke --function-name ProcessSQSRecord --payload file://input.json out
 --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 `aws configure set cli-binary-format raw-in-base64-out`。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

3. 在回應中查找 `INFO` 日誌。這是 Lambda 函數記錄訊息內文的位置。您應該會看到類似以下內容的輸出：

```
2023-09-11T22:45:04.271Z 348529ce-2211-4222-9099-59d07d837b60 INFO Processed
 message test
2023-09-11T22:45:04.288Z 348529ce-2211-4222-9099-59d07d837b60 INFO done
```

建立 Amazon SQS 佇列



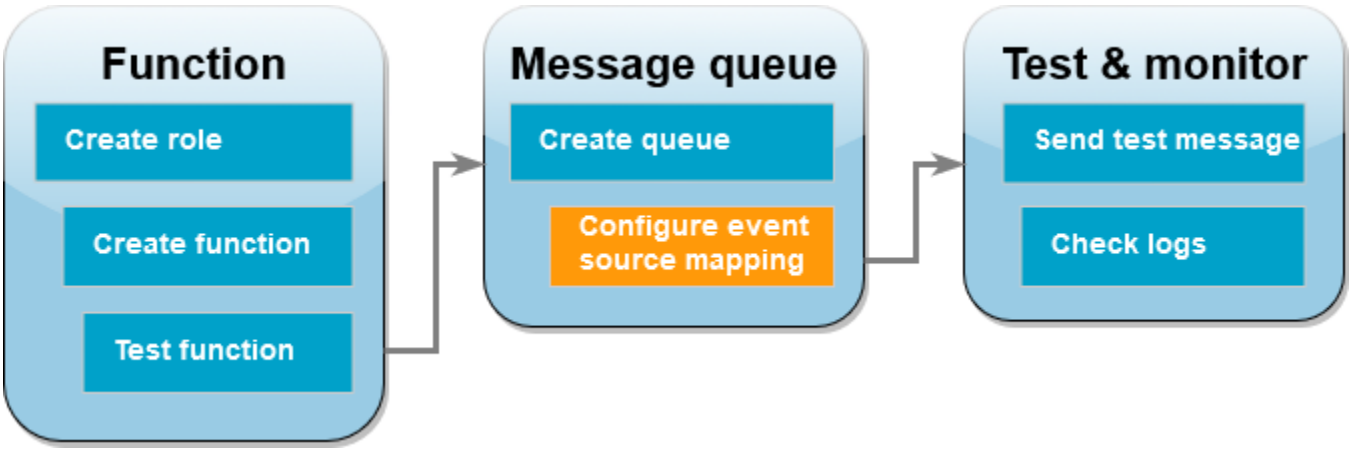
建立 Lambda 函數可用作事件來源的 Amazon SQS 佇列。該 Lambda 函數和 Amazon SQS 佇列必須位於相同的 AWS 區域。

建立佇列

1. 開啟 Amazon SQS 主控台。
2. 選擇建立佇列。
3. 輸入佇列的名稱。將所有其他選項保留為預設值。
4. 選擇建立佇列。

建立佇列後，請記下其 ARN。在下個步驟中，將佇列與您的 Lambda 函數建立關聯時會需要用到它。

設定事件來源



透過建立事件來源映射，將 Amazon SQS 佇列連線至 Lambda 函數。事件來源映射會讀取 Amazon SQS 佇列，並在新增訊息時調用 Lambda 函數。

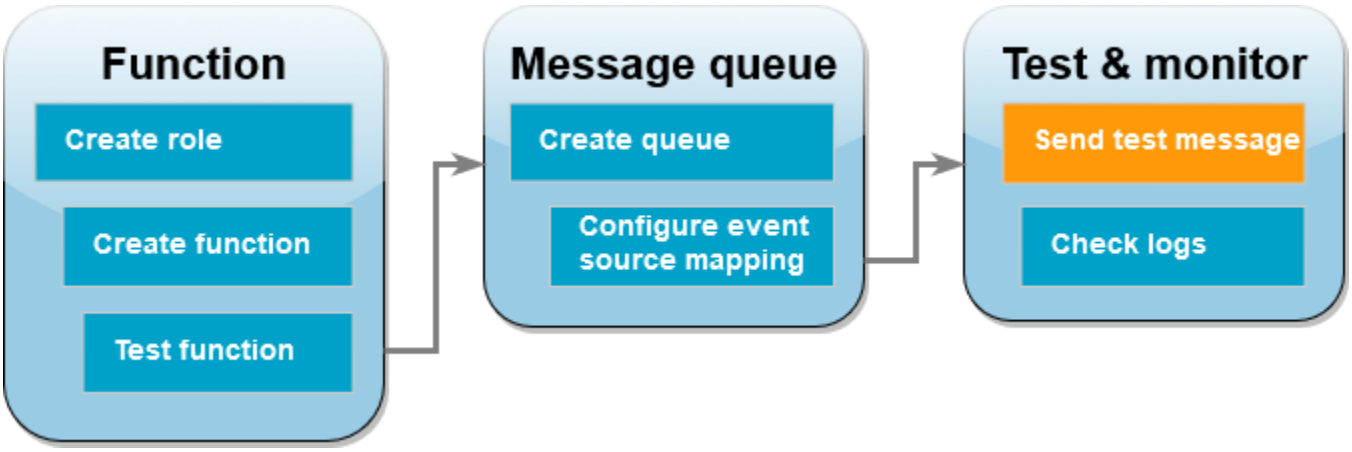
若要在 Amazon SQS 佇列和 Lambda 函數之間建立映射，請使用 create-event-source-mapping AWS CLI 命令。範例：

```
aws lambda create-event-source-mapping --function-name ProcessSQSRecord  --batch-size
 10 \
--event-source-arn arn:aws:sqs:us-east-1:111122223333:my-queue
```

若要獲取事件來源映射清單，請使用 list-event-source-mappings 命令。範例：

```
aws lambda list-event-source-mappings --function-name ProcessSQSRecord
```

傳送測試訊息

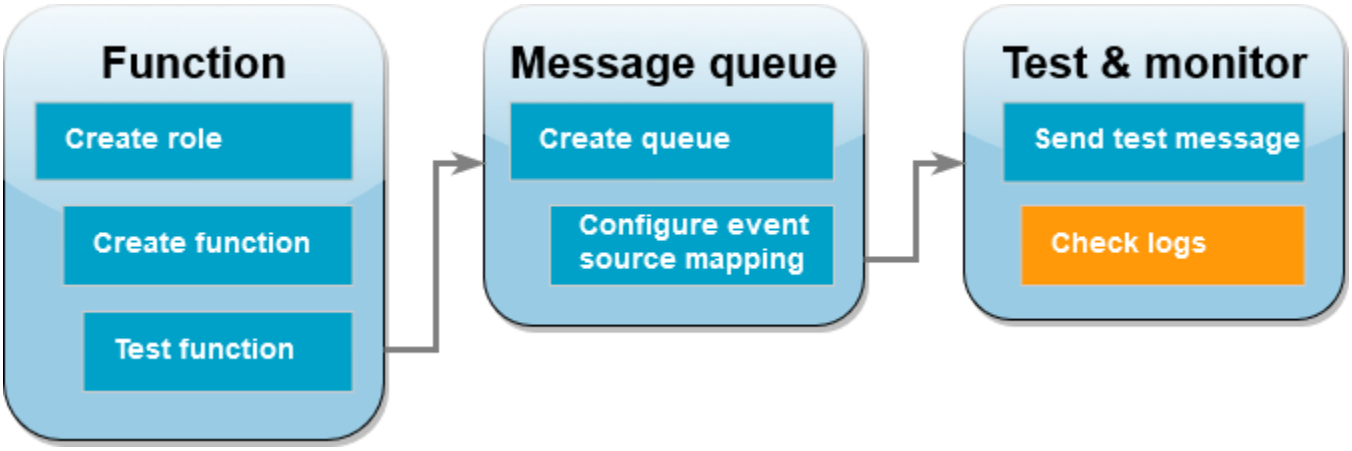


將 Amazon SQS 訊息傳送至 Lambda 函數

1. 開啟 Amazon SQS 主控台。
2. 選擇您稍早建立的佇列。
3. 選擇傳送及接收訊息。
4. 在訊息內文下，輸入測試訊息，例如「這是測試訊息」。
5. 選擇傳送訊息。

Lambda 輪詢佇列以查看是否有更新。當有新訊息時，Lambda 會使用佇列中的此新事件資料來叫用您的函數。如果函數處理常式傳回而無例外情況，則 Lambda 會認為訊息已成功處理，並開始讀取佇列中的新訊息。成功處理訊息之後，Lambda 從佇列中自動刪除它。如果處理常式擲出例外情況，Lambda 會認為訊息批次未成功處理，並且 Lambda 會調用具有相同訊息批次的函數。

檢查 CloudWatch 日誌



確認函數已處理訊息

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇 ProcessSQSRecord 函數。
3. 選擇監控。
4. 選擇檢視 CloudWatch 日誌。
5. 在 CloudWatch 主控台中，選擇函數的日誌串流。
6. 查找 INFO 日誌。這是 Lambda 函數記錄訊息內文的位置。應能看到您從 Amazon SQS 佇列傳送的訊息。範例：

```
2023-09-11T22:49:12.730Z b0c41e9c-0556-5a8b-af83-43e59efeec71 INFO Processed
 message this is a test message.
```

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。透過刪除您不再使用 AWS 的資源，您可以避免不必要的 費用 AWS 帳戶。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇刪除。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **confirm**，然後選擇 刪除 。

刪除 Amazon SQS 佇列

1. 登入 AWS 管理主控台 並開啟位於 https：//https://console.aws.amazon.com/sqs/ 的 Amazon SQS 主控台。
2. 選取您建立的佇列。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入 **confirm**。
5. 選擇 刪除 。

教學課程：使用跨帳戶 Amazon SQS 佇列做為事件來源

在本教學課程中，您會建立 Lambda 函數，使用來自不同 AWS 帳戶中 Amazon Simple Queue Service (Amazon SQS) 佇列的訊息。本教學課程涉及兩個 AWS 帳戶：帳戶 A 是指包含 Lambda 函數的帳戶，帳戶 B 是指包含 Amazon SQS 佇列的帳戶。

先決條件

安裝 AWS Command Line Interface

如果您尚未安裝 AWS Command Line Interface，請依照安裝或更新最新版本的 AWS CLI中的步驟進行安裝。

本教學課程需使用命令列終端機或 Shell 來執行命令。在 Linux 和 macOS 中，使用您偏好的 Shell 和套件管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。

建立執行角色 (帳戶 A)

在帳戶 A 中，建立 執行角色，授予您的函數存取所需 AWS 資源的許可。

若要建立執行角色

1. 在 AWS Identity and Access Management (IAM) 主控台中開啟角色頁面。

2. 選擇建立角色。
3. 建立具備下列屬性的角色。
   - 信任實體 – AWS Lambda。
   - 許可 - AWSLambdaSQSQueueExecutionRole
   - 角色名稱 - **cross-account-lambda-sqs-role**。

AWSLambdaSQSQueueExecutionRole 政策具備函數自 Amazon SQS 讀取項目以及寫入日誌到 Amazon CloudWatch Logs 時所需的許可。

建立函數 (帳戶 A)

在帳戶 A 中建立 Lambda 函數，它會處理 Amazon SQS 訊息。該 Lambda 函數和 Amazon SQS 佇列必須位於相同的 AWS 區域。

下列 Node.js 程式碼範例將每則訊息寫入 CloudWatch Logs 的日誌中。

Example index.mjs

```
export const handler = async function(event, context) {
  event.Records.forEach(record => {
    const { body } = record;
    console.log(body);
  });
  return {};
}
```

建立函數

> **Note**
>
> 遵循這些步驟，建立 Node.js 函式。對於其他語言，步驟類似，但有些細節不同。

1. 將程式碼範例儲存為名為 `index.mjs` 的檔案。
2. 建立部署套件。

```
zip function.zip index.mjs
```

3. 使用 `create-function` AWS Command Line Interface (AWS CLI) 命令建立 函數。將 `arn:aws:iam::111122223333:role/cross-account-lambda-sqs-role` 取代為先前建立之執行角色的 ARN。

```
aws lambda create-function --function-name CrossAccountSQSExample \
--zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
--role arn:aws:iam::111122223333:role/cross-account-lambda-sqs-role
```

測試函數 (帳戶 A)

在帳戶 A 中，使用 `invoke` AWS CLI 命令和範例 Amazon SQS 事件手動測試 Lambda 函數。

如果處理常式均正常傳回而無例外情況，Lambda 會認為訊息已成功處理，並開始讀取佇列中的新訊息。成功處理訊息之後，Lambda 從佇列中自動刪除它。如果處理常式擲出例外情況，Lambda 會認為訊息批次未成功處理，並且 Lambda 會調用具有相同訊息批次的函數。

1. 將下面的 JSON 儲存為名為 `input.txt` 的檔案。

```
{
    "Records": [
        {
            "messageId": "059f36b4-87a3-44ab-83d2-661975830a7d",
            "receiptHandle": "AQEBwJnKyrHigUMZj6rYigCgxlaS3SLy0a...",
            "body": "test",
            "attributes": {
                "ApproximateReceiveCount": "1",
                "SentTimestamp": "1545082649183",
                "SenderId": "AIDAIENQZJOLO23YVJ4VO",
                "ApproximateFirstReceiveTimestamp": "1545082649185"
            },
            "messageAttributes": {},
            "md5OfBody": "098f6bcd4621d373cade4e832627b4f6",
            "eventSource": "aws:sqs",
            "eventSourceARN": "arn:aws:sqs:us-east-1:111122223333:example-queue",
            "awsRegion": "us-east-1"
        }
    ]
}
```

上述 JSON 會模擬 Amazon SQS 可能傳送至 Lambda 函數的事件，其中 "body" 包含佇列中的實際訊息。

2. 執行下列 invoke AWS CLI 命令。

```
aws lambda invoke --function-name CrossAccountSQSExample \
--cli-binary-format raw-in-base64-out \
--payload file://input.txt outputfile.txt
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

3. 在檔案 outputfile.txt 中確認輸出。

建立 Amazon SQS 佇列 (帳戶 B)

在帳戶 B 中，建立帳戶 A 中 Lambda 函數可用作事件來源的 Amazon SQS 佇列。該 Lambda 函數和 Amazon SQS 佇列必須位於相同的 AWS 區域。

建立佇列

1. 開啟 Amazon SQS 主控台。
2. 選擇 建立佇列 。
3. 建立具備下列屬性的佇列。
   - Type (類型) - Standard (標準)
   - Name (名稱) - LambdaCrossAccountQueue
   - Configuration (組態) - 保留預設設定。
   - 存取政策 - 選擇進階。貼入下列 JSON 政策中。取代以下的值：
     - 111122223333帳戶 A 的 ： AWS 帳戶 ID
     - 444455556666帳戶 B 的 ： AWS 帳戶 ID

     JSON

```
{
    "Version":"2012-10-17",
```

```
    "Id": "Queue1_Policy_UUID",
    "Statement": [
        {
            "Sid": "Queue1_AllActions",
            "Effect": "Allow",
            "Principal": {
                "AWS": [
                    "arn:aws:iam::111122223333:role/cross-account-lambda-
sqs-role"
                ]
            },
            "Action": "sqs:*",
            "Resource": "arn:aws:sqs:us-
east-1:444455556666:LambdaCrossAccountQueue"
        }
    ]
}
```

此政策許可帳戶 A 中的 Lambda 執行角色取用此 Amazon SQS 佇列中的訊息。

4. 建立佇列後，記錄其 Amazon Resource Name (ARN)。在下個步驟中，將佇列與您的 Lambda 函數建立關聯時會需要用到它。

設定事件來源 (帳戶 A)

在帳戶 A 中，執行下列`create-event-source-mapping` AWS CLI 命令，在帳戶 B 中的 Amazon SQS 佇列與 Lambda 函數之間建立事件來源映射。將 `arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue` 取代為您在上一個步驟中建立的 Amazon SQS 佇列 ARN。

```
aws lambda create-event-source-mapping --function-name CrossAccountSQSExample --batch-
size 10 \
--event-source-arn arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue
```

若要取得事件來源映射的清單，請執行下列命令。

```
aws lambda list-event-source-mappings --function-name CrossAccountSQSExample \
--event-source-arn arn:aws:sqs:us-east-1:444455556666:LambdaCrossAccountQueue
```

測試設定

現在您可以測試設定，如下所示：

1. 在帳戶 B 中，開啟 Amazon SQS 主控台。
2. 選擇您先前建立的 LambdaCrossAccountQueue。
3. 選擇傳送及接收訊息。
4. 在 Message body (訊息主體) 中，輸入測試訊息。
5. 選擇傳送訊息。

帳戶 A 中您的 Lambda 函數應該會收到訊息。Lambda 會繼續輪詢佇列是否有更新。當有新訊息時，Lambda 會使用佇列中的此新事件資料來調用您的函數。您的函數會執行並在 Amazon CloudWatch 中建立日誌。可在 CloudWatch 主控台中檢視日誌。

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。透過刪除您不再使用 AWS 的資源，您可以避免不必要的 費用 AWS 帳戶。

在帳戶 A 中，清除您的執行角色和 Lambda 函數。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。
2. 選取您建立的執行角色。
3. 選擇刪除。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **`confirm`**，然後選擇 刪除 。

在帳戶 B 中，清除 Amazon SQS 佇列。

刪除 Amazon SQS 佇列

1. 登入 AWS 管理主控台 並開啟位於 https：//https://console.aws.amazon.com/sqs/ 的 Amazon SQS 主控台。
2. 選取您建立的佇列。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入 **confirm**。
5. 選擇 刪除 。

使用 Step Functions 協同運作 Lambda 函式

AWS Step Functions 提供視覺化工作流程協同運作，以協調 Lambda 函數與其他 AWS 服務。透過原生整合 220 種以上的 AWS 服務，以及全受管、零維護的基礎設施， Step Functions 是您需要視覺化工作流程設計和全受管服務整合時的理想選擇。

對於使用 Lambda 內工作流程邏輯與商業邏輯一起運作的標準程式設計語言進行協調，請考慮 Lambda 耐用函數。如需在這些選項之間進行選擇的說明，請參閱耐用函數或 Step Functions。

例如，處理訂單可能需要驗證訂單詳細資訊、檢查庫存程度、處理付款和產生發票。為每個任務編寫單獨的 Lambda 函式，並使用 Step Functions 來管理工作流程。Step Functions 會協調函式之間的資料流程，並在每個步驟中處理錯誤。這種區隔可確保工作流程即使日益複雜，也將更容易進行視覺化、修改和維護。

何時將 Step Functions 與 Lambda 搭配使用

以下是 Step Functions 特別適合協調 Lambda 型應用程式的良好範例。

- 循序處理
- 複雜錯誤處理
- 條件式工作流程與人工核准
- 平行處理

循序處理

循序處理是指一個任務必須完成後，下一個任務才能開始。例如，在訂單處理系統中，訂單驗證完成後才能開始付款處理，而發票產生必須等待付款確認。需要為每個任務編寫單獨的 Lambda 函式，並使用 Step Functions 來管理順序和處理函式之間的資料流程。

反模式範例

單一 Lambda 函式透過下列方式來管理整個訂單處理工作流程：

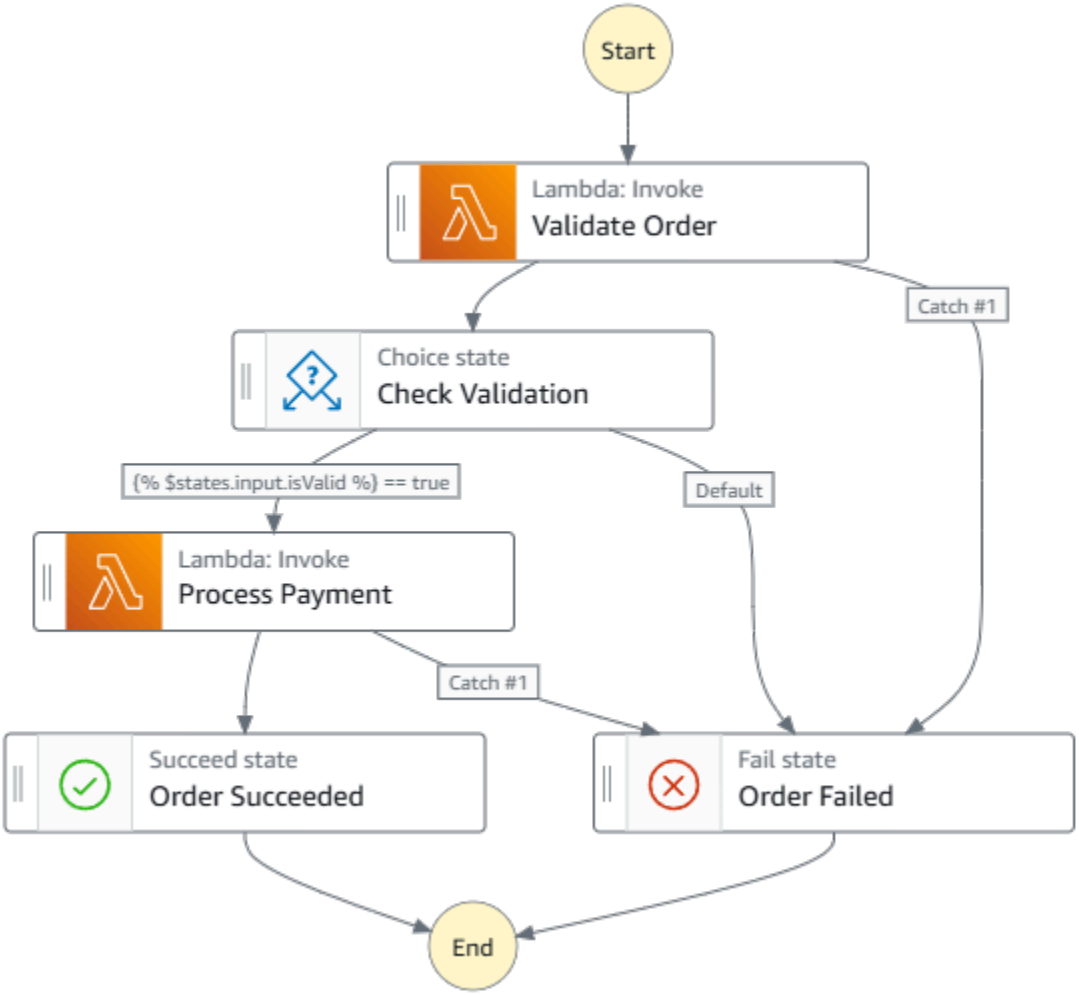
- 依序調用其他 Lambda 函式
- 剖析與驗證各函式的回應
- 實作錯誤處理與復原邏輯
- 管理函式之間的資料流程

建議方法

使用兩個 Lambda 函式：一個用於驗證訂單，另一個用於處理付款。Step Functions 透過下列方式協調這些函式：

- 以正確順序執行任務
- 在函式之間傳遞資料
- 在各步驟實作錯誤處理
- 使用 Choice 狀態確保僅有效訂單能進入付款流程

Example工作流程圖



> **Note**
>
> 程式碼優先替代方案：如需使用程式碼型檢查點和重試的循序處理，請參閱 Lambda 耐久函數步驟。

複雜錯誤處理

儘管 Lambda 為非同步調用和事件來源映射提供重試功能，但 Step Functions 為複雜的工作流程提供了更精密的錯誤處理。您可以使用指數退避設定自動重試，並為不同類型的錯誤設定不同的重試政策。如果重試用盡，請使用 `Catch` 將錯誤路由至後援狀態。當您需要協調多個函式和服務的工作流程層級錯誤處理時，這尤其有用。

若要進一步了解如何在狀態機器中處理 Lambda 函數錯誤，請參閱 AWS Step Functions 研討會中的處理錯誤。

反模式範例

單一 Lambda 函式會處理下列所有項目：

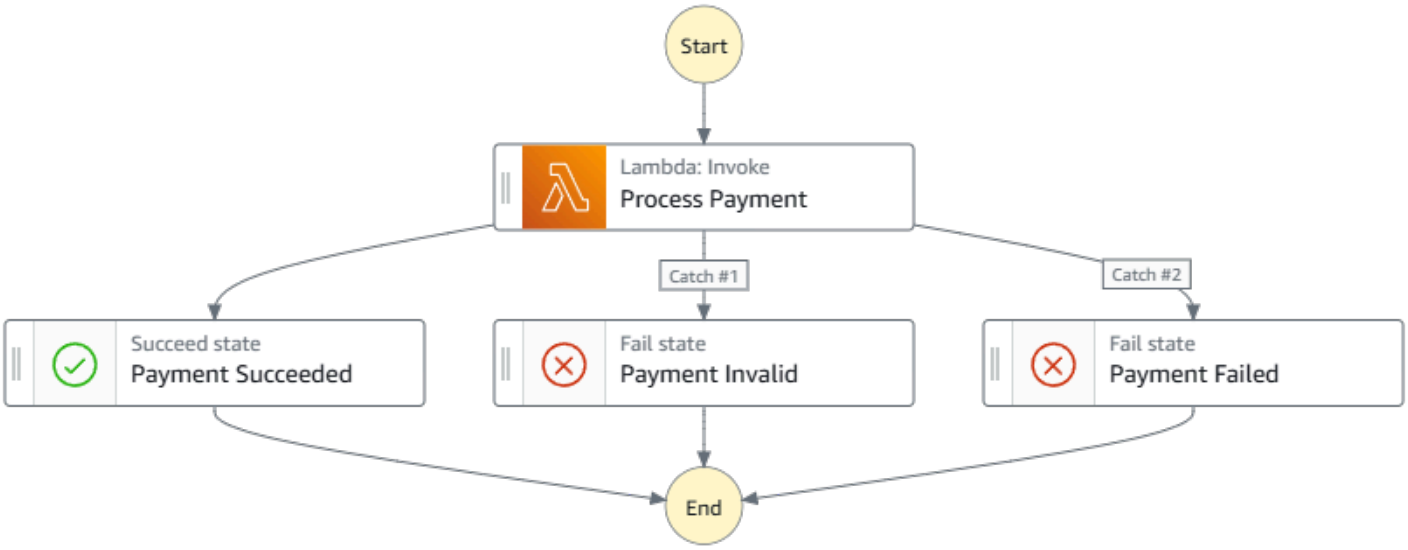
- 嘗試呼叫付款處理服務
- 如果付款服務無法使用，函式會等待並於稍後重試。
- 為等待時間實作自訂指數退避
- 所有嘗試都失敗後，擷取錯誤並選擇另一個流程

建議方法

使用僅專注於付款處理的單一 Lambda 函式。Step Functions 透過下列方式管理錯誤處理：

- 使用可設定的退避期間自動重試失敗的任務
- 根據錯誤類型套用不同的重試政策
- 將不同類型的錯誤路由至相應的後援狀態
- 維護錯誤處理狀態與歷史記錄

Example工作流程圖



> **Note**
>
> 程式碼優先替代方案：耐用的 函數透過可設定的重試策略提供 try-catch 錯誤處理。請參閱耐用函數中的錯誤處理。

條件式工作流程與人工核准

使用 Step Functions 的 Choice 狀態，根據函式輸出路由工作流程，並使用 waitForTaskToken 尾碼暫停工作流程以等待人工決策。例如，若要處理信用額度提高請求，可使用 Lambda 函式評估風險因素。然後，透過 Step Functions 將高風險請求路由至手動核准，將低風險請求路由至自動核准。

若要部署使用回呼任務字符整合模式的範例工作流程，請參閱 AWS Step Functions 研討會中的使用任務字符回呼。

反模式範例

單一 Lambda 函式透過下列方式管理複雜的核准工作流程：

- 實作巢狀條件邏輯來評估額度請求
- 根據請求金額調用不同的核准函式
- 管理多個核准路徑與決策點
- 追蹤待核准的狀態

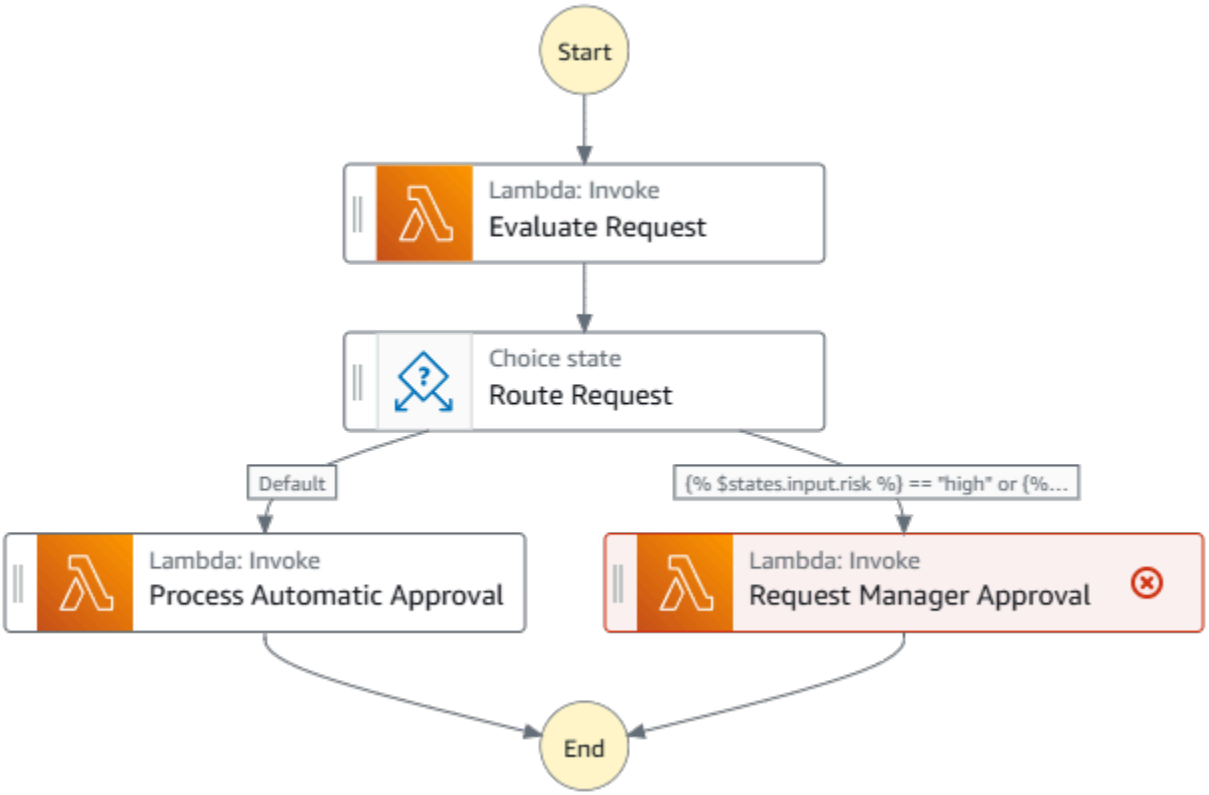
- 實作核准逾時與通知邏輯

建議方法

使用三個 Lambda 函式：一個用於評估每筆請求的風險，一個用於核准低風險請求，另一個用於將高風險請求路由至經理以供檢閱。Step Functions 透過下列方式管理工作流程：

- 使用 Choice 狀態根據金額與風險等級來路由請求
- 在等待人工核准時暫停執行
- 管理待核准逾時
- 提供每個請求目前狀態的可見性

Example工作流程圖



> **Note**
>
> 程式碼優先替代方案：耐用的 函數支援human-in-the-loop工作流程的回呼。請參閱耐用函數中的回呼。

平行處理

Step Functions 提供三種平行處理的實作方式：

- 平行狀態會同時執行工作流程的多個分支。需要平行執行不同的函式時 (例如在擷取映像中繼資料時產生縮圖)，可使用此狀態。
- 內嵌映射狀態會處理資料陣列，最多支援 40 個並行迭代。此狀態適用於需要對每個項目執行相同操作的中小型資料集。
- 分散式映射狀態會處理大規模平行處理，最多支援 10,000 個並行執行，同時支援 JSON 陣列和 Amazon Simple Storage Service (Amazon S3) 資料來源。處理大型資料集或需要更高並行性時，可使用此狀態。

反模式範例

單一 Lambda 函式嘗試透過下列方式管理平行處理：

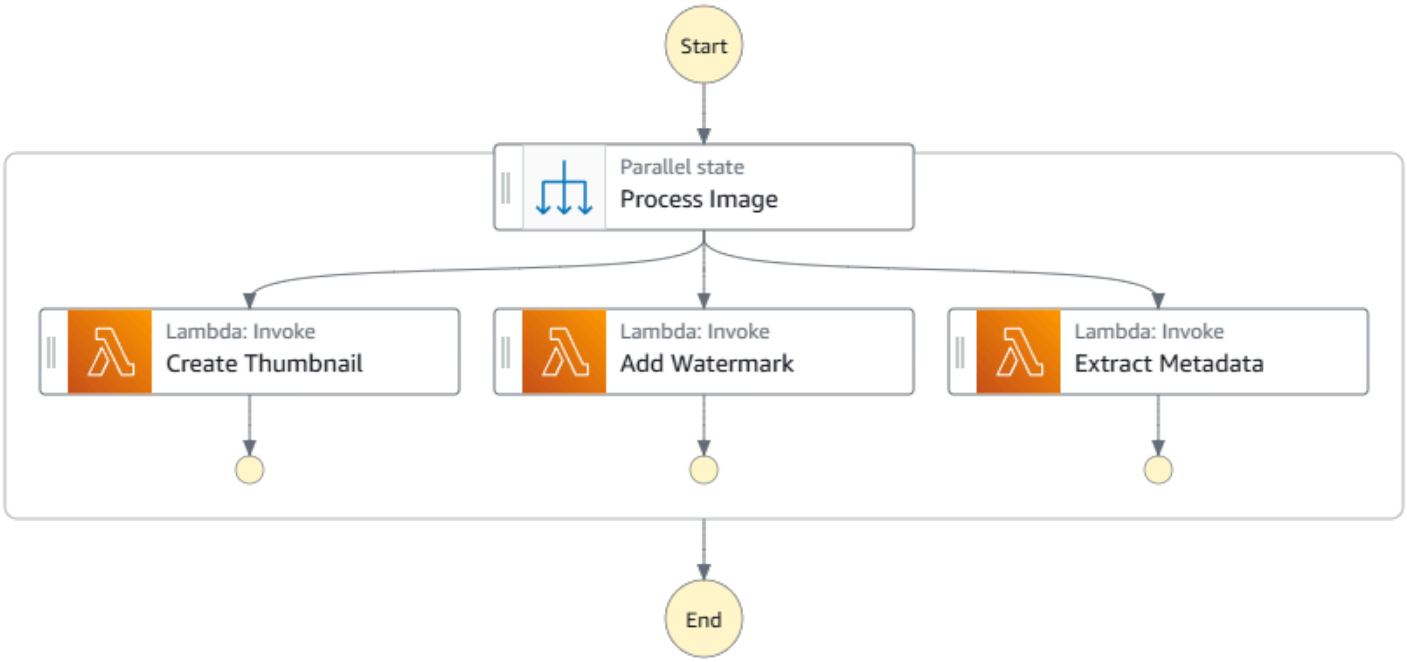
- 同時調用多個映像處理函式
- 實作自訂的平行執行邏輯
- 管理各平行任務的逾時與錯誤處理
- 收集並彙總所有函式的結果

建議方法

使用三個 Lambda 函式：一個用於建立縮圖映像、一個用於新增浮水印，另一個用於擷取中繼資料。Step Functions 透過下列方式管理這些函式：

- 使用平行狀態同時執行所有函式
- 將每個函式的結果收集到排序陣列
- 管理所有平行執行的逾時與錯誤處理
- 僅在所有平行分支完成後才繼續進行

Example工作流程圖



> **Note**
>
> 程式碼優先替代方案：耐用的 函數提供 `parallel()`和 `map()`操作。請參閱平行執行。

不適用 Step Functions 搭配 Lambda 的時機

並非所有以 Lambda 為基礎的應用程式都能從使用 Step Functions 中受益。選擇應用程式架構時，建議考量以下案例。

- 簡易的應用程式
- 複雜的資料處理
- CPU 密集型工作負載

簡易的應用程式

> **Note**
>
> 對於不需要視覺化設計或廣泛服務整合的工作流程，Lambda 耐用函數可能是將工作流程邏輯保留在 Lambda 內更簡單的替代方案。

對於不需要複雜協同運作的應用程式，使用 Step Functions 可能會增加不必要的複雜性。例如，若只是處理來自 Amazon SQS 佇列的訊息或回應 Amazon EventBridge 事件，您可以將這些服務設定為直接調用 Lambda 函式。同理，如果應用程式僅包含一或兩個具有直接錯誤處理的 Lambda 函式，則 Lambda 直接調用或事件驅動型架構可能更易於部署與維護。

複雜的資料處理

您可以使用 Step Functions 的分散式映射狀態，透過 Lambda 函式並行處理大型 Amazon S3 資料集。這對於許多大規模平行工作負載有效，包括處理 JSON 或 CSV 檔案等半結構化資料。但是，對於更複雜的資料轉換或進階分析，建議考量以下替代方案：

- 資料轉換管道： AWS Glue 用於處理來自多個來源的結構化或半結構化資料的 ETL 任務。當您需要內建資料目錄和結構描述管理功能時 AWS Glue ， 特別有用。
- 資料分析：針對 PB 級資料分析，可以使用 Amazon EMR，尤其是當您需要 Apache Hadoop 生態系統工具或用於超過 Lambda 記憶體限制的機器學習工作負載時。

CPU 密集型工作負載

儘管 Step Functions 可以協調 CPU 密集型任務，但由於 Lambda 函式的 CPU 資源有限，可能不適合處理此類工作負載。對於工作流程中需要大量運算資源的操作，建議考量以下替代方案：

- 容器協同運作：使用 Step Functions 來管理 Amazon Elastic Container Service (Amazon ECS) 任務，取得更加一致且可擴展的運算資源。
- 批次處理： AWS Batch 與 Step Functions 整合，以管理需要持續 CPU 用量的運算密集型批次任務。

使用 Amazon S3 批次事件調用 Lambda 函數

您可以使用 Amazon S3 批次操作，在大型 Amazon S3 物件組合叫用 Lambda 函數。Amazon S3 會追蹤批次操作的進度、傳送通知以及儲存顯示每個動作狀態的完成報告。

若要執行批次操作，您可以建立 Amazon S3 批次操作任務。當您建立任務時，請提供資訊清單 (物件清單)，並設定要在這些物件上執行的動作。

當批次任務開始時，Amazon S3 會針對資訊清單中的每個物件同步叫用 Lambda 函數。事件參數包括儲存貯體和物件的名稱。

下列範例顯示 Amazon S3 針對 amzn-s3-demo-bucket 儲存貯體中名稱為 customerImage1.jpg 的物件，傳送事件至 Lambda 函數。

Example Amazon S3 批次請求事件

```
{
"invocationSchemaVersion": "1.0",
    "invocationId": "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
    "job": {
        "id": "f3cc4f60-61f6-4a2b-8a21-d07600c373ce"
    },
    "tasks": [
        {
            "taskId": "dGFza2lkZ29lc2hlcmUK",
            "s3Key": "customerImage1.jpg",
            "s3VersionId": "1",
            "s3BucketArn": "arn:aws:s3:::amzn-s3-demo-bucket"
        }
    ]
}
```

您的 Lambda 函數必須傳回一個 JSON 物件，並有如下例所示的欄位。您可以從事件參數複製 `invocationId` 和 `taskId`。您可以在 `resultString` 返回子串。Amazon S3 會在完成報告中儲存 `resultString` 值。

Example Amazon S3 批次請求回應

```
{
```

```
  "invocationSchemaVersion": "1.0",
  "treatMissingKeysAs" : "PermanentFailure",
  "invocationId" : "YXNkbGZqYWRmaiBhc2RmdW9hZHNmZGpmaGFzbGtkaGZza2RmaAo",
  "results": [
    {
      "taskId": "dGFza2lkZ29lc2hlcmUK",
      "resultCode": "Succeeded",
      "resultString": "[\"Alice\", \"Bob\"]"
    }
  ]
}
```

從 Amazon S3 批次操作叫用 Lambda 函數

您可以使用不合格或合格的函數 ARN 叫用 Lambda 函數。如果您要在整個批次任務中使用相同的函數版本，請在建立任務時，在 `FunctionARN` 參數中設定特定的函數版本。如果您設定別名或 $LATEST 限定詞，如果在任務執行期間更新別名或 $LATEST，批次任務就會立即開始呼叫新版函數。

請注意，您無法重複使用現有的 Amazon S3 事件型函數進行批次操作。這是因為 Amazon S3 批次操作將不同的事件參數傳遞給 Lambda 函數，並預期傳回有特定 JSON 結構的訊息。

在您針對 Amazon S3 批次任務建立的資源型政策中，請確認您已針對任務設定權限以叫用您的 Lambda 函數。

在函數的執行角色中，針對 Amazon S3 設定信任政策以在角色執行您的函數時取得該角色。

如果您的函數使用 AWS SDK 來管理 Amazon S3 資源，您需要在執行角色中新增 Amazon S3 許可。

執行任務時，Amazon S3 會啟動多個函數執行個體以並行處理 Amazon S3 物件，直至達到函數的並行限制。Amazon S3 會限制執行個體的初始漸進測試，以避免較小任務造成過多成本。

如果 Lambda 函數傳回 `TemporaryFailure` 回應程式碼，Amazon S3 就會重試操作。

如需 Amazon S3 批次操作的詳細資訊，請參閱 Amazon S3 開發人員指南中的執行批次操作。

如需如何在 Amazon S3 批次操作中使用 Lambda 函數的範例，請參閱 Amazon S3 開發人員指南中的從 Amazon S3 批次操作中叫用 Lambda 函數。

使用 Amazon SNS 通知調用 Lambda 函數

您可以使用 Lambda 函數來處理 Amazon Simple Notification Service (Amazon SNS) 通知。Amazon SNS 支援 Lambda 函數作為傳送至主題之訊息的目標。您可以將自己的函數訂閱至相同帳戶或其他 AWS 帳戶中的主題。如需詳細演練，請參閱the section called “教學課程”。

Lambda 僅支援標準 SNS 主題的 SNS 觸發條件。不支援 FIFO 主題。

Lambda 透過將訊息排入佇列並處理重試，以非同步方式處理 SNS 訊息。如果 Amazon SNS 無法連接至 Lambda 或訊息遭到拒絕，Amazon SNS 會在數小時中以增加的間隔重試。如需詳細資訊，請參閱 Amazon SNS 常見問答集中的可靠性。

> **Warning**
>
> Lambda 非同步調用至少會處理每個事件一次，而且可能會重複處理記錄。為避免與重複事件相關的潛在問題，強烈建議您讓函數程式碼具有等冪性。如需詳細資訊，請參閱 AWS 知識中心中的如何使 Lambda 函式具有冪等性。

Powertools for AWS Lambda 的冪等性公用程式

Powertools for AWS Lambda 的冪等性公用程式可讓 Lambda 函式具有冪等性。該工具組支援 Python、TypeScript、Java 與 .NET。如需詳細資訊，請參閱 Powertools for AWS Lambda (Python) documentation 中的 Idempotency utility、Powertools for AWS Lambda (TypeScript) documentation 中的 Idempotency Utility、Powertools for AWS Lambda (Java) documentation 中的 Idempotency Utility，以及 Powertools for AWS Lambda (.NET) documentation 中的 Idempotency Utility。

主題

- 使用主控台為 Lambda 函數新增 Amazon SNS 主題觸發條件
- 手動新增 Lambda 函數的 Amazon SNS 主題觸發條件
- 範例 SNS 事件形狀
- 教學課程： AWS Lambda 搭配 Amazon Simple Notification Service 使用

使用主控台為 Lambda 函數新增 Amazon SNS 主題觸發條件

若要新增 SNS 主題做為 Lambda 函數的觸發條件，最簡單的方法便是使用 Lambda 主控台。當您透過主控台新增觸發條件時，Lambda 會自動設定必要的許可和訂閱，以開始從 SNS 主題接收事件。

若要新增 SNS 主題做為 Lambda 函數的觸發條件 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇要為其新增觸發條件的函數名稱。
3. 選擇組態，然後觸發條件。
4. 選擇 Add trigger (新增觸發條件)。
5. 在觸發條件組態底下的下拉式清單中，選擇 SNS。
6. 針對 SNS 主題，選擇要訂閱的 SNS 主題。

手動新增 Lambda 函數的 Amazon SNS 主題觸發條件

若要手動設定 Lambda 函數的 SNS 觸發條件，您需要完成以下步驟：

- 為函數定義資源型政策，以允許 SNS 調用它。
- 讓 Lambda 函數訂閱 Amazon SNS 主題。

> **Note**
>
> 如果 SNS 主題和 Lambda 函數位於不同的 AWS 帳戶中，您還需要授予函數額外許可，以允許跨帳戶訂閱 SNS 主題。如需詳細資訊，請參閱跨帳戶授予 Amazon SNS 訂閱的許可。

您可以使用 AWS Command Line Interface (AWS CLI) 來完成這兩個步驟。首先，使用以下 AWS CLI 命令，為 Lambda 函數定義資源型政策以允許 SNS 調用。請務必以您的 Lambda 函數名稱取代 `--function-name`，並以您的 SNS 主題 ARN 取代 `--source-arn` 的值。

```
aws lambda add-permission --function-name example-function \
    --source-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com
```

若要讓函數訂閱 SNS 主題，請使用以下 AWS CLI 命令。以您的 SNS 主題 ARN 取代 `--topic-arn` 的值，並以您的 Lambda 函數 ARN 取代 `--notification-endpoint` 的值。

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:123456789012:sns-topic-for-lambda \
```

```
    --notification-endpoint arn:aws:lambda:us-east-1:123456789012:function:example-
function
```

範例 SNS 事件形狀

Amazon SNS 使用包含訊息和中繼資料的事件，以非同步方式叫用您的函數。

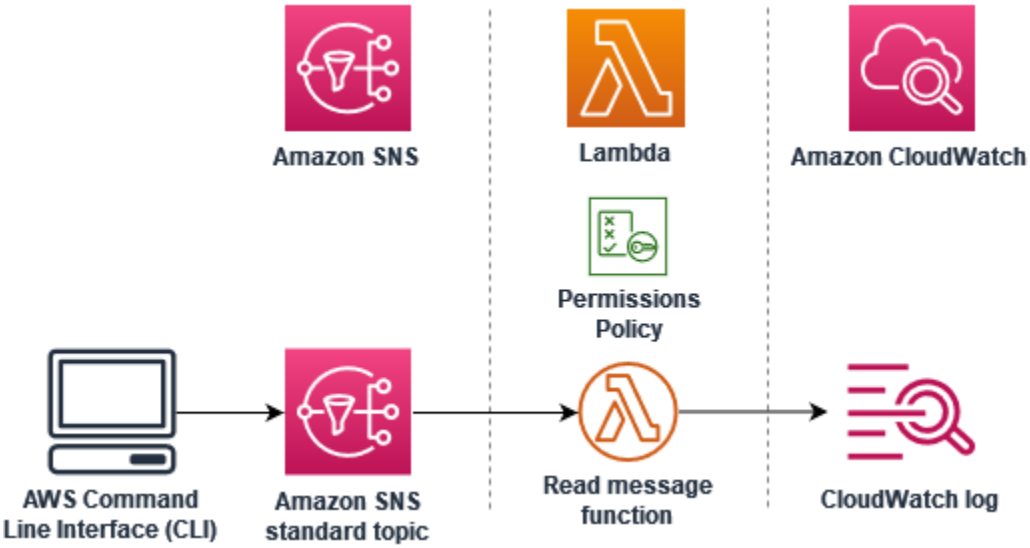
Example Amazon SNS 訊息事件

```
{
  "Records": [
    {
      "EventVersion": "1.0",
      "EventSubscriptionArn": "arn:aws:sns:us-east-1:123456789012:sns-lambda:21be56ed-
a058-49f5-8c98-aedd2564c486",
      "EventSource": "aws:sns",
      "Sns": {
        "SignatureVersion": "1",
        "Timestamp": "2019-01-02T12:45:07.000Z",
        "Signature": "tcc6faL2yUC6dgZdmrwh1Y4cGa/ebXEkAi6RibDsvpi+tE/1+82j...65r==",
        "SigningCertURL": "https://sns.us-east-1.amazonaws.com/
SimpleNotificationService-ac565b8b1a6c5d002d285f9598aa1d9b.pem",
        "MessageId": "95df01b4-ee98-5cb9-9903-4c221d41eb5e",
        "Message": "Hello from SNS!",
        "MessageAttributes": {
          "Test": {
            "Type": "String",
            "Value": "TestString"
          },
          "TestBinary": {
            "Type": "Binary",
            "Value": "TestBinary"
          }
        },
        "Type": "Notification",
        "UnsubscribeUrl": "https://sns.us-east-1.amazonaws.com/?
Action=Unsubscribe&amp;SubscriptionArn=arn:aws:sns:us-east-1:123456789012:test-
lambda:21be56ed-a058-49f5-8c98-aedd2564c486",
        "TopicArn":"arn:aws:sns:us-east-1:123456789012:sns-lambda",
        "Subject": "TestInvoke"
      }
    }
  ]
```

```
}
```

教學課程： AWS Lambda 搭配 Amazon Simple Notification Service 使用

在本教學課程中，您會在一個 中使用 Lambda 函數 AWS 帳戶 ，以個別方式訂閱 Amazon Simple Notification Service (Amazon SNS) 主題 AWS 帳戶。將訊息發佈至 Amazon SNS 主題時，Lambda 函數會讀取訊息的內容，並將其輸出到 Amazon CloudWatch Logs。若要完成本教學課程，請使用 AWS Command Line Interface (AWS CLI)。



請執行下列步驟以完成本教學課程：

- 在帳戶 A 中建立 Amazon SNS 主題。
- 在帳戶 B 中建立 Lambda 函數，以讀取主題的訊息。
- 在帳戶 B 中建立主題的訂閱。
- 將訊息發佈至帳戶 A 中的 Amazon SNS 主題，並確認帳戶 B 中的 Lambda 函數是否有將其輸出至 CloudWatch Logs。

完成這些步驟後，您將了解如何設定 Amazon SNS 主題來調用 Lambda 函數。您也將了解如何建立 AWS Identity and Access Management (IAM) 政策，為另一個 中的資源提供叫用 Lambda AWS 帳戶的許可。

在此教學課程中，您會使用兩種獨立的 AWS 帳戶。這些 AWS CLI 命令使用兩個名為 `accountA`和的命名設定檔來說明這一點`accountB`，每個設定檔都設定為與不同的設定檔搭配使用 AWS 帳戶。若要了解如何將 AWS CLI 設定為使用不同的設定檔，請參閱《 第 AWS Command Line Interface 2 版使用者指南》中的組態和登入資料檔案設定。請務必 AWS 區域 為兩個設定檔設定相同的預設值。

如果您為兩者建立的 AWS CLI 設定檔 AWS 帳戶 使用不同的名稱，或者您使用預設設定檔和一個具名設定檔，請視需要在下列步驟中修改 AWS CLI 命令。

先決條件

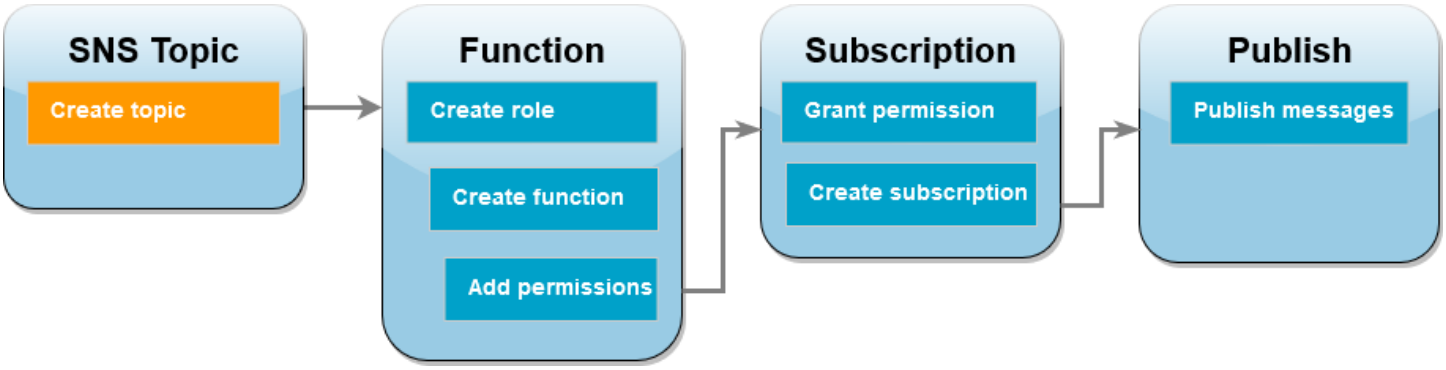
安裝 AWS Command Line Interface

如果您尚未安裝 AWS Command Line Interface，請依照安裝或更新最新版本的 AWS CLI中的步驟進行安裝。

本教學課程需使用命令列終端機或 Shell 來執行命令。在 Linux 和 macOS 中，使用您偏好的 Shell 和套件管理工具。

> **Note**
>
> 在 Windows 中，作業系統的內建終端不支援您常與 Lambda 搭配使用的某些 Bash CLI 命令 (例如 `zip`)。若要取得 Ubuntu 和 Bash 的 Windows 整合版本，請安裝適用於 Linux 的 Windows 子系統。

建立 Amazon SNS 主題 (帳戶 A)



若要建立 主題

- 在帳戶 A 中，使用下列 AWS CLI 命令建立 Amazon SNS 標準主題。

  ```
  aws sns create-topic --name sns-topic-for-lambda --profile accountA
  ```

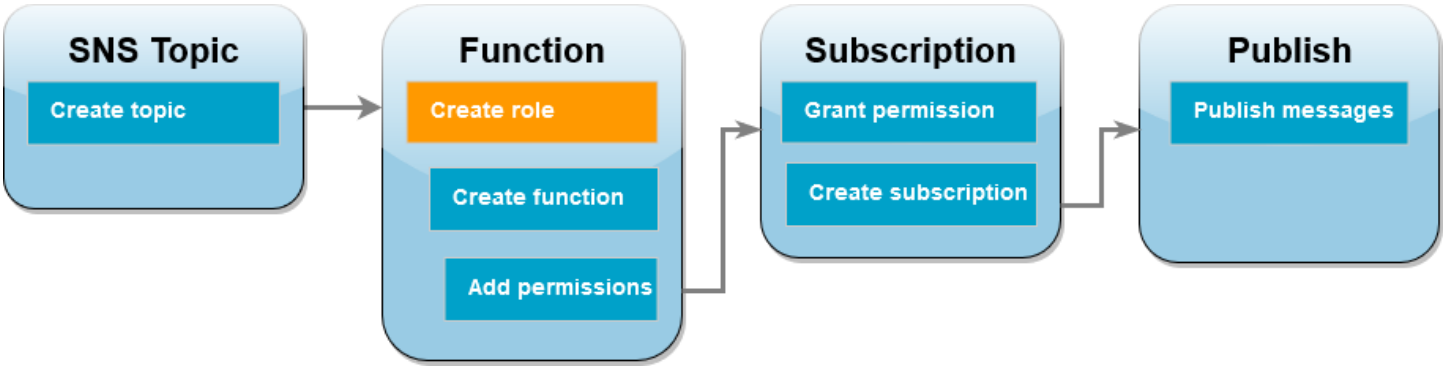
  您應該會看到類似下列的輸出。

  ```
  {
  ```

```
    "TopicArn": "arn:aws:sns:us-west-2:123456789012:sns-topic-for-lambda"
}
```

記下主題的 Amazon Resource Name (ARN)。當您新增許可到 Lambda 函數以訂閱主題時，稍後會在教學課程中用上它。

建立函數執行角色 (帳戶 B)

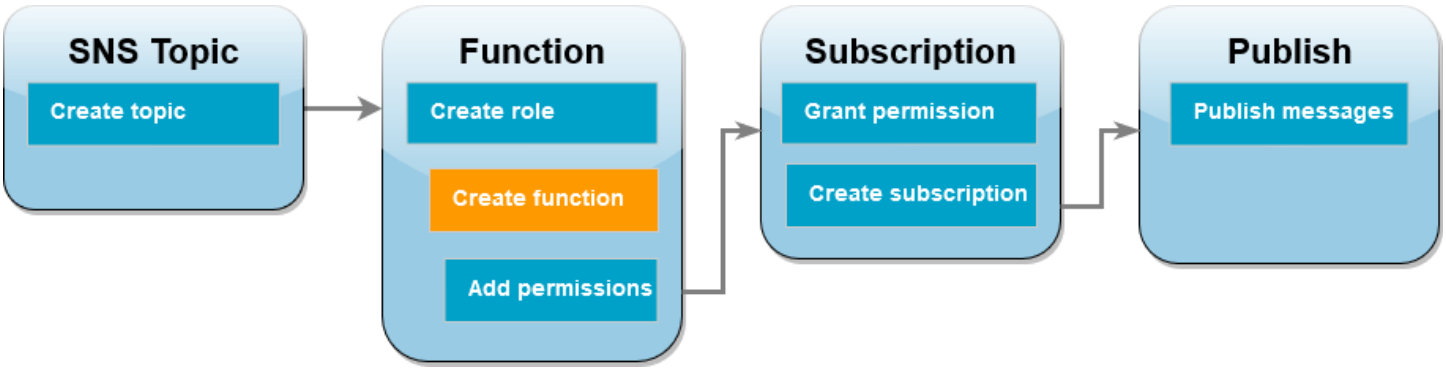


執行角色是授予 Lambda 函數存取 AWS 服務 和資源許可的 IAM 角色。在帳戶 B 建立函數前，您必須建立一個角色，該角色會授與將日誌寫入 CloudWatch Logs 的函數基本許可。我們將在稍後的步驟中新增要從 Amazon SNS 主題讀取的許可。

若要建立執行角色

1. 在帳戶 B 中開啟 IAM 主控台的角色頁面。
2. 選擇建立角色。
3. 針對信任的實體類型，請選擇 AWS 服務。
4. 針對使用案例，請選擇 Lambda。
5. 選擇下一步。
6. 透過下列步驟將基本許可政策新增至角色：
   a. 在許可政策搜尋方塊中，輸入 **AWSLambdaBasicExecutionRole**。
   b. 選擇下一步。
7. 執行下列動作來完成角色的建立：
   a. 在角色詳細資訊下方的角色名稱中輸入 **lambda-sns-role**。
   b. 選擇建立角色。

建立 Lambda 函數 (帳戶 B)



建立可處理 Amazon SQS 訊息的 Lambda 函數。函數程式碼會將每筆記錄的訊息內容記錄到 Amazon CloudWatch Logs。

本教學課程使用 Node.js 24 執行時間，但我們也提供其他執行時間語言的範例程式碼。您可以在下列方塊中選取索引標籤，查看您感興趣的執行期程式碼。此步驟要使用的 JavaScript 程式碼，位於 JavaScript 索引標籤中顯示的第一個範例。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;
```

```
public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed record
 {record.Sns.Message}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;

    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

使用 TypeScript 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";
```

```
export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
```

```
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;

class Handler extends SnsHandler
{
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 SNS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here

    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來使用 SNS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
  raise
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
```

```
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);

    // Implement your record handling code here.

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

建立函數

1. 建立專案的目錄，然後切換至該目錄。

```
mkdir sns-tutorial
cd sns-tutorial
```

2. 將範例 JavaScript 程式碼複製到名為 `index.js` 的新檔案。
3. 使用以下 `zip` 命令建立部署套件。

```
zip function.zip index.js
```

4. 執行下列 AWS CLI 命令，在帳戶 B 中建立 Lambda 函數。

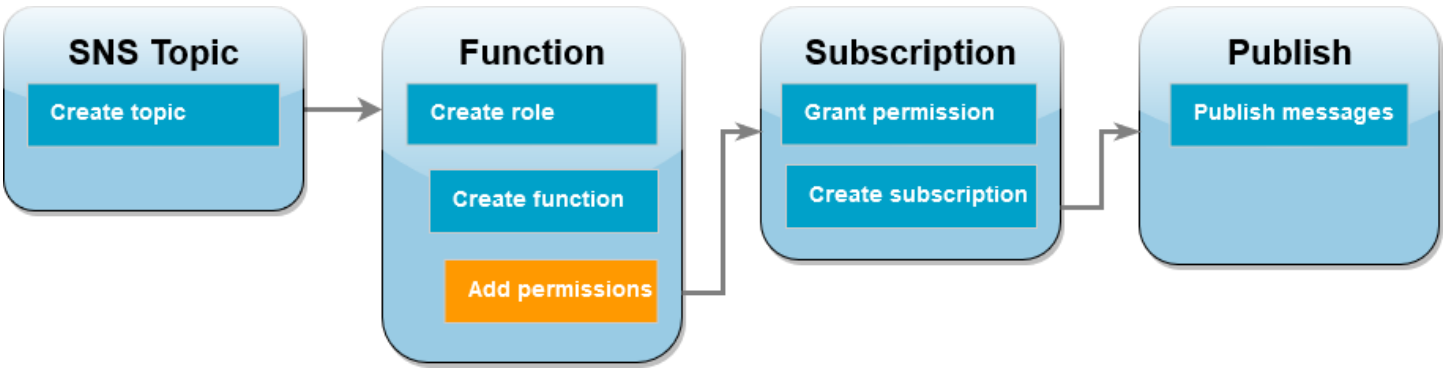
```
aws lambda create-function --function-name Function-With-SNS \
    --zip-file fileb://function.zip --handler index.handler --runtime nodejs24.x \
    --role arn:aws:iam::<AccountB_ID>:role/lambda-sns-role  \
    --timeout 60 --profile accountB
```

您應該會看到類似下列的輸出。

```
{
    "FunctionName": "Function-With-SNS",
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:Function-With-
SNS",
    "Runtime": "nodejs24.x",
    "Role": "arn:aws:iam::123456789012:role/lambda_basic_role",
    "Handler": "index.handler",
    ...
    "RuntimeVersionConfig": {
        "RuntimeVersionArn": "arn:aws:lambda:us-
west-2::runtime:7d5f06b69c951da8a48b926ce280a9daf2e8bb1a74fc4a2672580c787d608206"
    }
}
```

5. 記錄函數的 Amazon Resource Name (ARN)。當您新增許可以允許 Amazon SNS 調用您的函數時，稍後會在教學課程中用上它。

為函數新增許可 (帳戶 B)



若要使用 Amazon SNS 調用函數，您必須在以資源為基礎之政策的陳述式中授予許可。您可以使用 命令新增此陳述式 AWS CLI `add-permission`。

若要授予 Amazon SNS 調用您函數的許可

- 在帳戶 B 中，針對您先前記錄的 Amazon SNS 主題，使用 ARN 執行下列 AWS CLI 命令。

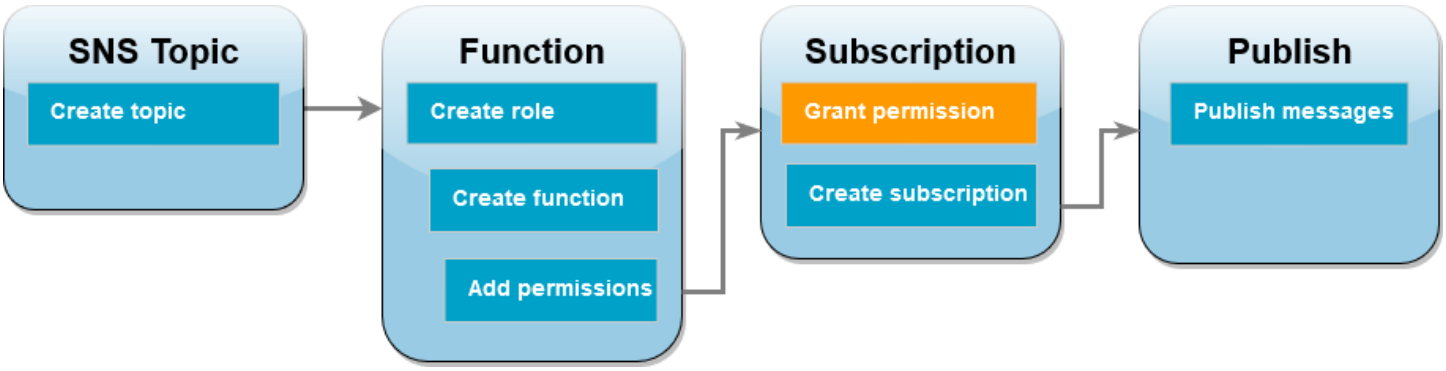
```
aws lambda add-permission --function-name Function-With-SNS \
    --source-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --statement-id function-with-sns --action "lambda:InvokeFunction" \
    --principal sns.amazonaws.com --profile accountB
```

您應該會看到類似下列的輸出。

```
{
    "Statement": "{\"Condition\":{\"ArnLike\":{\"AWS:SourceArn\":
      \"arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda\"}},
      \"Action\":[\"lambda:InvokeFunction\"],
      \"Resource\":\"arn:aws:lambda:us-east-1:<AccountB_ID>:function:Function-With-
SNS\",
      \"Effect\":\"Allow\",\"Principal\":{\"Service\":\"sns.amazonaws.com\"},
      \"Sid\":\"function-with-sns\"}"
}
```

> **Note**
>
> 如果具有 Amazon SNS 主題的帳戶託管在選擇加入 AWS 區域中，您需要在委託人中指定區域。例如，如果您使用亞太區域 (香港)的 Amazon SNS 主題，則需要為主體指定 `sns.ap-east-1.amazonaws.com`，而不是 `sns.amazonaws.com`。

授予 Amazon SNS 訂閱的跨帳戶許可 (帳戶 A)



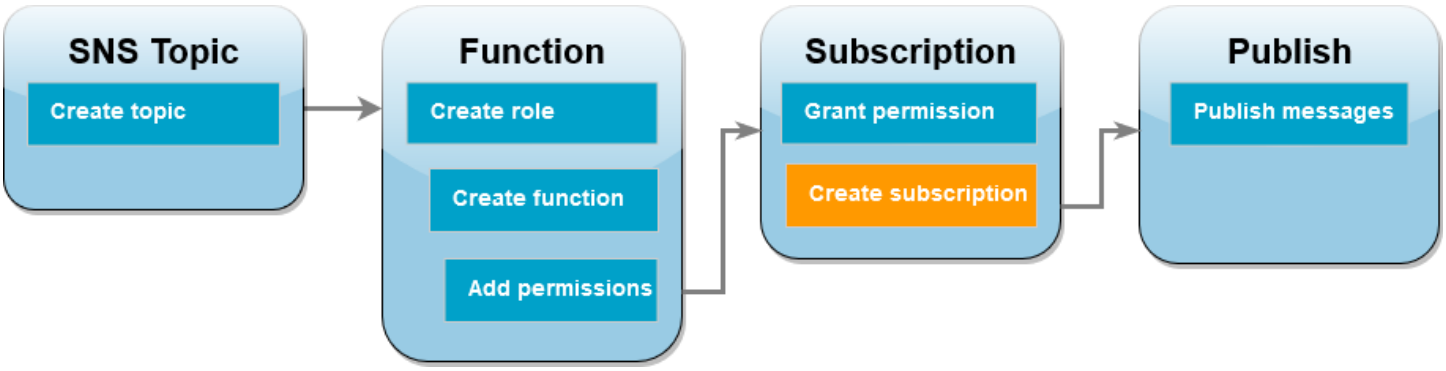
若要讓帳戶 B 中的 Lambda 函數訂閱您在帳戶 A 中建立的 Amazon SNS 主題，您必須向帳戶 B 授予訂閱您主題的許可。您可以使用 AWS CLI `add-permission`命令授予此許可。

若要向帳戶 B 授予訂閱主題的許可

- 在帳戶 A 中，執行下列 AWS CLI 命令。將 ARN 用於之前記錄的 Amazon SNS 主題。

```
aws sns add-permission --label lambda-access --aws-account-id <AccountB_ID> \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --action-name Subscribe ListSubscriptionsByTopic --profile accountA
```

建立訂閱 (帳戶 B)



在帳戶 B 中，您現在讓 Lambda 函數訂閱您在教學課程開始時透過帳戶 A 建立的 Amazon SNS 主題。當訊息傳送至此主題 (`sns-topic-for-lambda`) 時，Amazon SNS 會調用您帳戶 B 中的 Lambda 函數 `Function-With-SNS`。

若要建立訂閱

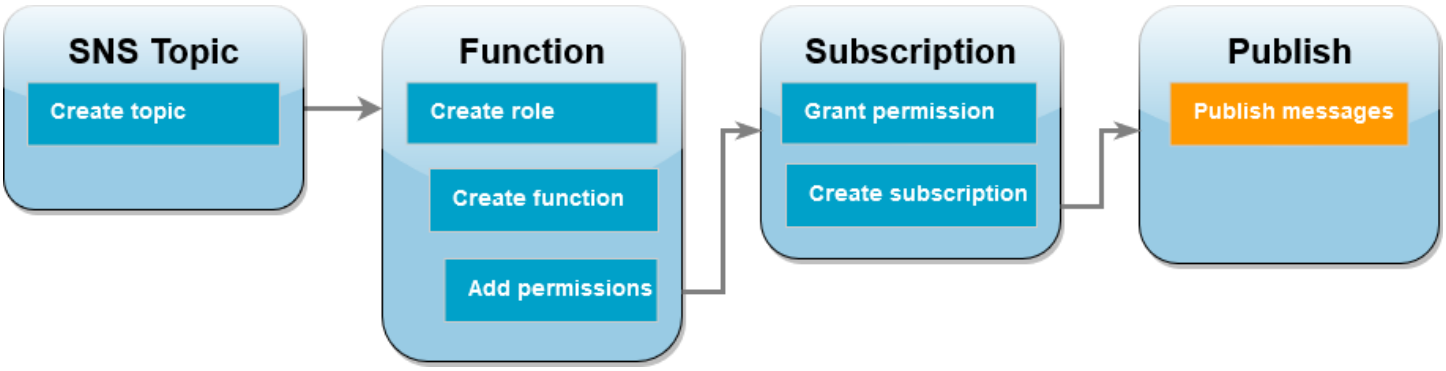
- 在帳戶 B 中，執行下列 AWS CLI 命令。使用您建立主題的預設區域，以及主題和 Lambda 函數的 ARN。

```
aws sns subscribe --protocol lambda \
    --region us-east-1 \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --notification-endpoint arn:aws:lambda:us-
east-1:<AccountB_ID>:function:Function-With-SNS \
    --profile accountB
```

您應該會看到類似下列的輸出。

```
{
    "SubscriptionArn": "arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-
lambda:5d906xxxx-7c8x-45dx-a9dx-0484e31c98xx"
}
```

將訊息發佈至主題 (帳戶 A 和帳戶 B)



帳戶 B 中的 Lambda 函數已訂閱您帳戶 A 中的 Amazon SNS 主題後，是時候將訊息發佈至您的主題來測試您的設定了。若要確認 Amazon SNS 是否已調用 Lambda 函數，請使用 CloudWatch Logs 來檢視函數的輸出。

若要將訊息發佈到您的主題並檢視函數的輸出

1. 輸入 `Hello World` 至文字檔，並儲存為 `message.txt`。
2. 從您儲存文字檔案的相同目錄中，在帳戶 A 中執行下列 AWS CLI 命令。 針對您自己的主題使用 ARN。

```
aws sns publish --message file://message.txt --subject Test \
    --topic-arn arn:aws:sns:us-east-1:<AccountA_ID>:sns-topic-for-lambda \
    --profile accountA
```

這將傳回具有唯一識別符的訊息 ID，表示 Amazon SNS 已接受訊息。接著，Amazon SNS 會嘗試將訊息傳遞給主題訂閱者。若要確認 Amazon SNS 是否已調用 Lambda 函數，請使用 CloudWatch Logs 來檢視函數的輸出：

3. 在帳戶 B中開啟 Amazon CloudWatch 主控台的日誌群組頁面。
4. 為函數 (`/aws/lambda/Function-With-SNS`) 選擇日誌群組名稱。
5. 選擇最新的日誌串流。
6. 如果您的函數有被正確調用，您將會看到類似下方的輸出，顯示您發佈到主題的訊息內容。

```
2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO Processed
 message Hello World
2023-07-31T21:42:51.250Z c1cba6b8-ade9-4380-aa32-d1a225da0e48 INFO done
```

清除您的資源

除非您想要保留為此教學課程建立的資源，否則您現在便可刪除。透過刪除您不再使用 AWS 的資源，您可以避免不必要的 費用 AWS 帳戶。

在帳戶 A 中，清除您的 Amazon SNS 主題。

刪除 Amazon SNS 主題

1. 在 Amazon SNS 主控台開啟 Topics (主題) 頁面。
2. 選擇您建立的主題。
3. 選擇 刪除 。
4. 在文字輸入欄位中輸入 **`delete me`**。
5. 選擇 刪除 。

在帳戶 B 中，清除您的執行角色、Lambda 函數以及 Amazon SNS 訂閱。

刪除執行角色

1. 開啟 IAM 主控台中的 角色頁面 。

2. 選取您建立的執行角色。
3. 選擇刪除。
4. 在文字輸入欄位中輸入角色的名稱，然後選擇 刪除 。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **`confirm`**，然後選擇 刪除 。

刪除 Amazon SNS 訂閱

1. 在 Amazon SNS 主控台開啟 Subscriptions (訂閱) 頁面。
2. 選取您建立的訂閱。
3. 選擇刪除，刪除。

在 中管理許可 AWS Lambda

您可以使用 AWS Identity and Access Management (IAM) 來管理 中的許可 AWS Lambda。使用 Lambda 函數時，需要考慮兩個主要的許可類別：

- Lambda 函數執行 API 動作和存取其他 AWS 資源所需的許可
- 其他 AWS 使用者和實體存取 Lambda 函數所需的許可

Lambda 函數通常需要存取其他 AWS 資源，並對這些資源執行各種 API 操作。例如，您可能有一個 Lambda 函數，透過更新 Amazon DynamoDB 資料庫中的項目來回應事件。在這種情況下，函數需要存取資料庫的許可，以及在該資料庫中放置或更新項目的許可。

可以在稱為執行角色的特殊 IAM 角色中定義 Lambda 函數所需的許可。在此角色中，您可以連接政策，定義函數存取其他 AWS 資源所需的每個許可，以及從事件來源讀取。每個 Lambda 函數都必須擁有執行角色。您的執行角色必須至少能夠存取 Amazon CloudWatch，因為 Lambda 函數預設會記錄到 CloudWatch Logs。可以將 `AWSLambdaBasicExecutionRole` 受管政策連接至執行角色，以滿足此需求。

若要授予其他 AWS 帳戶、組織和服務存取 Lambda 資源的許可，您有幾個選項：

- 您可以使用身分型政策授予其他使用者對 Lambda 資源的存取權。以身分為基礎的政策可直接套用到使用者或與使用者相關連的群組和角色。
- 您可以使用以資源為基礎的政策，為其他帳戶提供存取 Lambda 資源的 AWS 服務 許可。當使用者嘗試存取 Lambda 資源時，Lambda 會同時考慮身分型政策 (針對使用者)，以及以資源型政策 (針對資源)。當 Amazon Simple Storage Service (Amazon S3) 之類的 AWS 服務呼叫您的 Lambda 函數時，Lambda 只會考慮以資源為基礎的政策。
- 您可以使用屬性型存取控制 (ABAC) 模型來控制對 Lambda 函數的存取。透過 ABAC，您可以將標籤連接至 Lambda 函數、在特定 API 請求中傳遞標籤，或將其連接至提出請求的 IAM 主體。在 IAM 政策的條件元素中指定相同的標籤以控制函數存取。

在 中 AWS，最佳實務是僅授予執行任務所需的許可 (最低權限許可)。若要在 Lambda 中實作此功能，建議從 AWS 受管政策開始。您可以依原狀使用這些受管政策，或將其作為起點，撰寫限制程度更高的專屬政策。

為了協助您微調最低權限存取許可，Lambda 提供了一些其他條件，可供您包含在政策中。如需詳細資訊，請參閱the section called “資源與條件”。

如需 IAM 的詳細資訊，請參閱《IAM 使用者指南》。

使用執行角色定義 Lambda 函數許可

Lambda 函數的執行角色是 AWS Identity and Access Management (IAM) 角色，授予函數存取 AWS 服務 和資源的許可。例如，您可以建立一個執行角色，該角色有權向 Amazon CloudWatch 傳送日誌並向 AWS X-Ray上傳追蹤資料。本頁提供有關如何建立、檢視和管理 Lambda 函數執行角色的資訊。

當您調用函數時，Lambda 會自動擔任您的執行角色。您應該避免在函數程式碼中手動呼叫 `sts:AssumeRole` 以承擔執行角色。如果您的使用案例請求角色擔任自己，則您必須將角色本身作為受信任主體包含在角色的信任政策中。如需如何修改角色信任政策的詳細資訊，請參閱《IAM 使用者指南》中的修改角色信任政策 (主控台)。

為了讓 Lambda 能夠正確擔任執行角色，角色的信任政策必須將 Lambda 服務主體 (`lambda.amazonaws.com`) 指定為受信任的服務。

主題

- 在 IAM 主控台中建立執行角色
- 使用 建立和管理角色 AWS CLI
- 為 Lambda 執行角色授予最低權限存取權
- 在執行角色中檢視和更新許可
- 在執行角色中使用 AWS 受管政策
- 使用來源函數 ARN 控制函數存取行為

在 IAM 主控台中建立執行角色

根據預設，當您在 Lambda 主控台中建立函數時，Lambda 會建立具有最低許可的執行角色。具體而言，此執行角色包含 `AWSLambdaBasicExecutionRole` 受管政策，該政策會授予您的函數將事件記錄到 Amazon CloudWatch Logs 的基本許可。您可以在許可區段中選取建立預設角色。

您可以在許可區段中選取使用另一個角色，以選擇現有角色。如果您的 Lambda 函數需要額外許可來執行任務，例如更新 Amazon DynamoDB 資料庫中的項目以回應事件，您可以建立具有必要許可的自訂執行角色。若要這樣做，請在許可區段中選取使用另一個角色，這會開啟一個抽屜，您可以在其中自訂許可。

從主控台設定執行角色

1. 在角色詳細資訊區段中輸入角色名稱。
2. 在政策區段中，選取使用現有政策。

3. 選取您要連接到角色的 AWS 受管政策。例如，如果您的函數需要存取 DynamoDB，請選取 AWSLambdaDynamoDBExecutionRole 受管政策。
4. 選擇建立角色。

或者，當您在 Lambda 主控台中建立函數時，您可以將先前建立的任何執行角色連接到函數。如果您想要將新的執行角色連接到現有的函數，請遵循更新函數的執行角色中的步驟。

使用 建立和管理角色 AWS CLI

若要使用 AWS Command Line Interface (AWS CLI) 建立執行角色，請使用 create-role命令。使用此命令時，您可以指定內嵌信任政策。角色的信任政策授予指定主體擔任該角色的許可。在下列範例中，您授予 Lambda 服務主體擔任您的角色的許可。注意，JSON 字串中轉義引號的請求有所不同，這取決於您的 shell。

```
aws iam create-role \
  --role-name lambda-ex \
  --assume-role-policy-document '{"Version": "2012-10-17",       "Statement":
 [{ "Effect": "Allow", "Principal": {"Service": "lambda.amazonaws.com"}, "Action":
 "sts:AssumeRole"}]}'
```

您也可使用單獨的 JSON 檔案來定義角色的信任政策。在下列範例中，`trust-policy.json` 為當前目錄中的檔案。

Example trust-policy.json

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

```
aws iam create-role \
  --role-name lambda-ex \
  --assume-role-policy-document file://trust-policy.json
```

使用 attach-policy-to-role 命令將許可新增至角色。下列命令可將 AWSLambdaBasicExecutionRole 受管政策新增至 lambda-ex 執行角色。

```
aws iam attach-role-policy --role-name lambda-ex --policy-arn arn:aws:iam::aws:policy/
service-role/AWSLambdaBasicExecutionRole
```

建立執行角色之後，將其連接至您的函數。當您在 Lambda 主控台中建立函數時，可以將先前建立的任何執行角色連接至該函數。如果您想要將新的執行角色連接至現有函數，請遵循the section called "更新函數的執行角色"中的步驟。

為 Lambda 執行角色授予最低權限存取權

當您第一次為 Lambda 函數建立 IAM 角色時，有時可能會授予超出所需的許可。在生產環境中發佈您的函數之前，最佳實務是調整政策以僅包含必要的許可。如需詳細資訊，請參閱《IAM 使用者指南》中的套用最低權限許可。

使用 IAM Access Analyzer 來協助識別 IAM 執行角色政策的必要許可。IAM Access Analyzer 會在您指定的日期範圍內檢閱您的 AWS CloudTrail 日誌，並產生僅具有該函數在此期間使用之許可的政策範本。您可以使用範本建立具有精細許可的受管政策，然後將其連接至 IAM 角色。如此一來，您只會授予角色與特定使用案例 AWS 的資源互動所需的許可。

如需詳細資訊，請參閱《IAM 使用者指南》中的根據存取活動產生政策。

在執行角色中檢視和更新許可

本主題涵蓋如何檢視和更新函數的執行角色。

主題

- 檢視函數的執行角色
- 更新函數的執行角色

檢視函數的執行角色

若要檢視函數的執行角色，請使用 Lambda 主控台。

若要檢視函數的執行角色 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 選擇 組態 ，然後選擇 許可 。
4. 在執行角色下，您可以檢視目前用作函數執行角色的角色。為了方便起見，您可以在資源摘要區段下檢視函數可以存取的所有資源和動作。也可以從下拉式清單中選擇一種服務，以查看與該服務相關的許可。

更新函數的執行角色

您可以隨時從函式的執行角色新增或移除許可，或將函式設定為使用不同的角色。如果您的函數需要存取任何其他服務或資源，您必須將必要的許可新增至執行角色。

將許可新增至您的函數時，亦請對其程式碼或組態進行細微更新。若您函數的執行中執行個體具有已過期的憑證，上述動作會強制停止並取代這些執行個體。

若要更新函數的執行角色，可以使用 Lambda 主控台。

若要更新函數的執行角色 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數的名稱。
3. 選擇 組態 ，然後選擇 許可 。
4. 在執行角色下面，選擇編輯。
5. 如果您想要更新您的函數以使用其他角色做為執行角色，請在現有角色下的下拉式功能表中選擇新角色。

   > **Note**
   >
   > 如果您想要更新現有執行角色中的許可，您只能在 AWS Identity and Access Management (IAM) 主控台中執行。

   如果您想要建立新的角色以用作執行角色，請選擇從執行角色下的 AWS 政策範本建立新角色。然後，在角色名稱下輸入新角色的名稱，並在政策範本下指定要連接至新角色的任何政策。
6. 選擇 Save (儲存)。

在執行角色中使用 AWS 受管政策

下列 AWS 受管政策提供使用 Lambda 功能所需的許可。

變更	描述	日期
AWSLambdaMSKExecutionRole – Lambda 已將 kafka:DescribeClusterV2 許可新增至此政策。	AWSLambdaMSKExecutionRole 授予從 Amazon Managed Streaming for Apache Kafka (Amazon MSK) 叢集中讀取和存取記錄、管理彈性網絡介面 (ENI)，和寫入 CloudWatch Logs 的許可。	2022 年 6 月 17 日
AWSLambdaBasicExecutionRole – Lambda 開始追蹤對此政策的變更。	AWSLambdaBasicExecutionRole 授予將日誌上傳至 CloudWatch 的許可。	2022 年 2 月 14 日
AWSLambdaDynamoDBExecutionRole – Lambda 開始追蹤對此政策的變更。	AWSLambdaDynamoDBExecutionRole 授予從 Amazon DynamoDB 串流讀取記錄和寫入 CloudWatch Logs 的許可。	2022 年 2 月 14 日
AWSLambdaKinesisExecutionRole – Lambda 開始追蹤對此政策的變更。	AWSLambdaKinesisExecutionRole 授予從 Amazon Kinesis 資料串流讀取事件和寫入 CloudWatch Logs 的許可。	2022 年 2 月 14 日
AWSLambdaMSKExecutionRole – Lambda 開始追蹤對此政策的變更。	AWSLambdaMSKExecutionRole 授予從 Amazon Managed Streaming for Apache Kafka (Amazon MSK) 叢集中讀取和存取記錄、管理彈性網絡介面 (ENI)，和寫入 CloudWatch Logs 的許可。	2022 年 2 月 14 日

變更	描述	日期
AWSLambdaSQSQueueExecutionRole – Lambda 開始追蹤對此政策的變更。	AWSLambdaSQSQueueExecutionRole 授予從 Amazon Simple Queue Service (Amazon SQS) 佇列中讀取訊息並寫入 CloudWatch Logs 的許可。	2022 年 2 月 14 日
AWSLambdaVPCAccessExecutionRole – Lambda 開始追蹤對此政策的變更。	AWSLambdaVPCAccessExecutionRole 授予管理 Amazon VPC 內 ENI 和寫入 CloudWatch Logs 的許可。	2022 年 2 月 14 日
AWSXRayDaemonWriteAccess – Lambda 開始追蹤對此政策的變更。	AWSXRayDaemonWriteAccess 授予將追蹤資料上傳至 X-Ray 的許可。	2022 年 2 月 14 日
CloudWatchLambdaInsightsExecutionRolePolicy – Lambda 開始追蹤對此政策的變更。	CloudWatchLambdaInsightsExecutionRolePolicy 授予將執行時間指標寫入 CloudWatch Lambda Insights 的許可。	2022 年 2 月 14 日
AmazonS3ObjectLambdaExecutionRolePolicy – Lambda 開始追蹤對此政策的變更。	AmazonS3ObjectLambdaExecutionRolePolicy 會授予許可，以與 Amazon Simple Storage Service (Amazon S3) 物件 Lambda 互動，並寫入 CloudWatch Logs。	2022 年 2 月 14 日

針對某些功能，Lambda 主控台會嘗試將遺漏的許可新增到您客戶受管政策中的執行角色。這些政策的數量可能會相當多。請在啟用功能前將相關受 AWS 管理政策新增到您的執行角色，以避免建立額外政策。

當您使用事件來源映射來調用函數時，Lambda 會使用執行角色來讀取事件資料。例如，Kinesis 的事件來源映射會從資料串流讀取事件，並將這些事件批次傳送到函數。

當服務在您的帳戶中擔任角色時，您可以在角色信任政策中包含 `aws:SourceAccount` 和 `aws:SourceArn` 全域條件內容金鑰，以將角色的存取限制為僅由預期資源產生的請求。如需詳細資訊，請參閱 AWS Security Token Service的預防跨服務混淆代理人。

除了 AWS 受管政策之外，Lambda 主控台還提供範本，用於建立具有其他使用案例許可的自訂政策。當您在 Lambda 主控台中建立函數時，您可以選擇使用來自一個或多個範本的許可建立新的執行角色。當您從藍圖建立函式時，或是您設定需要存取其他服務的選項時，也會自動套用這些範本。範例範本可在此指南的 GitHub 儲存庫取得。

使用來源函數 ARN 控制函數存取行為

您的 Lambda 函數程式碼通常會向其他 AWS 服務發出 API 請求。若要提出這些請求，Lambda 會擔任函數的執行角色，以產生一組暫時性憑證。這些憑證在函數調用期間可當成環境變數使用。使用 AWS SDK 時，您不需要直接在程式碼中提供 SDK 的憑證。根據預設，憑證提供者鏈會依序檢查您可以設定憑證的每個位置，並選取第一個可用的位置，通常是環境變數 (`AWS_ACCESS_KEY_ID`、`AWS_SECRET_ACCESS_KEY` 和 `AWS_SESSION_TOKEN`)。

如果請求是來自您執行環境內的 AWS API 請求，Lambda 會將來源函數 ARN 注入登入資料內容。針對 Lambda 代表您在執行環境外發出的下列 AWS API 請求，Lambda 也會插入來源函數 ARN：

服務	Action	Reason
CloudWatch Logs	`CreateLogGroup`，`CreateLogStream`，`PutLogEvents`	將日誌儲存到 CloudWatch Logs 日誌群組中
X-Ray	`PutTraceSegments`	將追蹤資料傳送到 X-Ray
Amazon EFS	`ClientMount`	將函數連接到 Amazon Elastic File System (Amazon EFS) 檔案系統

Lambda 使用相同執行角色在執行環境之外代表您進行的其他 AWS API 呼叫不包含來源函數 ARN。這類執行環境外的 API 呼叫範例包括：

- 呼叫 AWS Key Management Service (AWS KMS) 以自動加密和解密您的環境變數。

- 對 Amazon Elastic Compute Cloud (Amazon EC2) 發出的呼叫，目的是為已啟用 VPC 的函數建立彈性網絡介面 (ENI)。
- 呼叫 AWS 服務，例如 Amazon Simple Queue Service (Amazon SQS)，從設定為事件來源映射的事件來源讀取。

您可以使用憑證內容中的來源函數 ARN，驗證對資源的呼叫是否來自特定 Lambda 函數的程式碼。若要驗證這一點，請在 IAM 身分型政策或服務控制政策 (SCP) 中使用 `lambda:SourceFunctionArn` 條件索引鍵。

> **Note**
>
> 您無法在資源型政策中使用 `lambda:SourceFunctionArn` 條件索引鍵。

在您的身分型政策或 SCP 中使用此條件索引鍵，您可以為函數程式碼對其他 AWS 服務進行的 API 操作實作安全控制。此操作具有一些關鍵的安全保護作用，例如，協助您識別憑證洩漏的來源。

> **Note**
>
> `lambda:SourceFunctionArn` 條件索引鍵與 `lambda:FunctionArn` 和 `aws:SourceArn` 條件索引鍵不同。`lambda:FunctionArn` 條件索引鍵僅適用於事件來源映射，會協助定義您的事件來源可以呼叫哪些函數。`aws:SourceArn` 條件索引鍵僅適用於 Lambda 函數是目標資源的政策，並有助於定義哪些其他 AWS 服務 和資源可以叫用該函數。`lambda:SourceFunctionArn` 條件索引鍵可套用至任何身分型政策或 SCP，以定義具有對其他資源進行特定 AWS API 呼叫許可的特定 Lambda 函數。

若要在政策中使用 `lambda:SourceFunctionArn`，請將其作為任何 ARN 條件運算子的條件，納入到政策中。索引鍵的值必須為有效的 ARN。

例如，假設您的 Lambda 函數程式碼會進行 `s3:PutObject` 呼叫，該呼叫以特定 Amazon S3 儲存貯體為目標。您可能希望僅允許一個特定 Lambda 函數擁有對該儲存貯體的 `s3:PutObject` 存取權。在此情況下，您函數的執行角色應連接有類似以下的政策：

Example授予特定 Lambda 函數存取權以存取 Amazon S3 資源的政策

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ExampleSourceFunctionArn",
            "Effect": "Allow",
            "Action": "s3:PutObject",
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Condition": {
                "ArnEquals": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
        }
    ]
}
```

此政策僅在來源為具有 ARN `arn:aws:lambda:us-east-1:123456789012:function:source_lambda` 的 Lambda 函數時允許 `s3:PutObject` 存取權。此政策不允許對任何其他呼叫身分的 `s3:PutObject` 存取權。即使不同函數或實體以相同執行角色進行 `s3:PutObject` 呼叫也是如此。

> **Note**
>
> `lambda:SourceFunctionARN` 條件索引鍵不支援 Lambda 函數版本或函數別名。如果您將 ARN 用於特定函數版本或別名，則您的函數將無權執行您指定的動作。務必為您的函數使用不符資格的 ARN，而不使用版本或別名後綴。

您也可以在 SCP 中使用 `lambda:SourceFunctionArn`。例如，假設您要僅讓單一 Lambda 函數的程式碼或來自 Amazon 虛擬私有雲端 (VPC) 的呼叫能存取儲存貯體。以下 SCP 對此進行說明。

Example在特定條件下拒絕對 Amazon S3 的存取的政策

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "StringNotEqualsIfExists": {
                    "aws:SourceVpc": [
                        "vpc-12345678"
                    ]
                }
            }
        },
        {
            "Action": [
                "s3:*"
            ],
            "Resource": "arn:aws:s3:::lambda_bucket/*",
            "Effect": "Deny",
            "Condition": {
                "ArnNotEqualsIfExists": {
                    "lambda:SourceFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:source_lambda"
                }
            }
        }
    ]
}
```

除非 S3 動作來自具有 ARN `arn:aws:lambda:*:123456789012:function:source_lambda` 的特定 Lambda 函數，或是其來自指定的 VPC，否則此政策會拒絕所有 S3 動作。僅在請求中包含 `aws:SourceVpc` 索引鍵時，`StringNotEqualsIfExists` 運算子才會告訴 IAM 處理此條件。同樣

地，僅在 lambda:SourceFunctionArn 存在時，IAM 才會考慮 ArnNotEqualsIfExists 運算子。

授予其他 AWS 實體存取 Lambda 函數的權限

若要授予其他 AWS 帳戶、組織和服務存取 Lambda 資源的許可，您有幾個選項：

- 您可以使用身分型政策授予其他使用者對 Lambda 資源的存取權。以身分為基礎的政策可直接套用到使用者或與使用者相關連的群組和角色。
- 您可以使用以資源為基礎的政策，為其他帳戶提供存取 Lambda 資源的 AWS 服務 許可。當使用者嘗試存取 Lambda 資源時，Lambda 會同時考慮身分型政策 (針對使用者)，以及以資源型政策 (針對資源)。當 Amazon Simple Storage Service (Amazon S3) 之類的 AWS 服務呼叫您的 Lambda 函數時，Lambda 只會考慮以資源為基礎的政策。
- 您可以使用屬性型存取控制 (ABAC) 模型來控制對 Lambda 函數的存取。透過 ABAC，您可以將標籤連接至 Lambda 函數、在特定 API 請求中傳遞標籤，或將其連接至提出請求的 IAM 主體。在 IAM 政策的條件元素中指定相同的標籤以控制函數存取。

為了協助您微調最低權限存取許可，Lambda 提供了一些其他條件，可供您包含在政策中。如需詳細資訊，請參閱the section called “資源與條件”。

Lambda 適用的身分型 IAM 政策

您可以在 AWS Identity and Access Management (IAM) 中使用身分型政策來授予帳戶中的使用者 Lambda 的存取權。身分型政策可套用至使用者、使用者群組或角色。您可以授予另一個帳戶的使用者許可，讓他們可以擔任您帳戶中的角色並存取 Lambda 資源。

Lambda 提供 AWS 受管政策，授予 Lambda API 動作的存取權，以及在某些情況下，其他用於開發和管理 Lambda 資源的 AWS 服務存取權。Lambda 會視需要更新這些受管政策，來確保您的使用者可在新功能發行時進行存取。

- AWSLambda_FullAccess - 對於用於開發和維護 Lambda 資源的 Lambda 動作和其他 AWS 服務，授予完整存取權限。
- AWSLambda_ReadOnlyAccess - 授予 Lambda 資源的唯讀存取權限。
- AWSLambdaRole - 授予調用 Lambda 函數的許可。

AWS 受管政策授予 API 動作的許可，而不限制使用者可以修改的 Lambda 函數或層。如需進行更精細的控制，您可以建立自己的政策，來限制使用者的許可範圍。

主題

- 授予使用者對 Lambda 函數的存取權

- 授予使用者對 Lambda 層的存取權

授予使用者對 Lambda 函數的存取權

使用身分型政策以允許使用者、使用者群組或角色對 Lambda 函數執行操作。

> **Note**
>
> 對於定義為容器映像的函數，存取映像的使用者許可必須在 Amazon Elastic Container Registry (Amazon ECR) 中設定。如需範例，請參閱 Amazon ECR 儲存庫政策。

以下顯示具受限範圍之許可政策的範例。這可讓使用者建立和管理以指定字首 (`intern-`) 命名並以指定執行角色設定的 Lambda 函數。

Example函數開發政策

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ReadOnlyPermissions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetAccountSettings",
                "lambda:GetEventSourceMapping",
                "lambda:GetFunction",
                "lambda:GetFunctionConfiguration",
                "lambda:GetFunctionCodeSigningConfig",
                "lambda:GetFunctionConcurrency",
                "lambda:ListEventSourceMappings",
                "lambda:ListFunctions",
                "lambda:ListTags",
                "iam:ListRoles"
            ],
            "Resource": "*"
        },
        {
            "Sid": "DevelopFunctions",
```

```
            "Effect": "Allow",
            "NotAction": [
                "lambda:AddPermission",
                "lambda:PutFunctionConcurrency"
            ],
            "Resource": "arn:aws:lambda:*:*:function:intern-*"
        },
        {
            "Sid": "DevelopEventSourceMappings",
            "Effect": "Allow",
            "Action": [
                "lambda:DeleteEventSourceMapping",
                "lambda:UpdateEventSourceMapping",
                "lambda:CreateEventSourceMapping"
            ],
            "Resource": "*",
            "Condition": {
                "ArnLike": {
                    "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
                }
            }
        },
        {
            "Sid": "PassExecutionRole",
            "Effect": "Allow",
            "Action": [
                "iam:ListRolePolicies",
                "iam:ListAttachedRolePolicies",
                "iam:GetRole",
                "iam:GetRolePolicy",
                "iam:PassRole",
                "iam:SimulatePrincipalPolicy"
            ],
            "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
        },
        {
            "Sid": "ViewLogs",
            "Effect": "Allow",
            "Action": [
                "logs:*"
            ],
            "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
        }
    ]
```

```
}
```

會根據許可支援的資源和條件來將此政策中的許可組織為陳述式。

- ReadOnlyPermissions - Lambda 主控台會在您瀏覽和檢視函數時使用這些許可。他們不支援資源模式或條件。

```
        "Action": [
            "lambda:GetAccountSettings",
            "lambda:GetEventSourceMapping",
            "lambda:GetFunction",
            "lambda:GetFunctionConfiguration",
            "lambda:GetFunctionCodeSigningConfig",
            "lambda:GetFunctionConcurrency",
            "lambda:ListEventSourceMappings",
            "lambda:ListFunctions",
            "lambda:ListTags",
            "iam:ListRoles"
        ],
        "Resource": "*"
```

- DevelopFunctions - 使用會對字首為 intern- 的函數進行操作的 Lambda 動作 (除了 AddPermission 和 PutFunctionConcurrency)。AddPermission 會在該函數中修改以資源為基礎的政策，並且可能會產生安全隱憂。PutFunctionConcurrency 會為某函數保留擴展容量並可從其他函數取走容量。

```
        "NotAction": [
            "lambda:AddPermission",
            "lambda:PutFunctionConcurrency"
        ],
        "Resource": "arn:aws:lambda:*:*:function:intern-*"
```

- DevelopEventSourceMappings – 在字首為 intern- 的函數上管理事件來源映射。這些動作會在事件來源映射上操作，但您可以使用條件來依函數進行限制。

```
        "Action": [
            "lambda:DeleteEventSourceMapping",
            "lambda:UpdateEventSourceMapping",
```

```
            "lambda:CreateEventSourceMapping"
        ],
        "Resource": "*",
        "Condition": {
            "StringLike": {
                "lambda:FunctionArn": "arn:aws:lambda:*:*:function:intern-*"
            }
        }
```

- `PassExecutionRole` - 檢視並僅傳遞名為 `intern-lambda-execution-role` 的角色，必須由具有 IAM 許可的使用者建立和管理該角色。當您將執行角色指派至函數時就會用到 `PassRole`。

```
        "Action": [
            "iam:ListRolePolicies",
            "iam:ListAttachedRolePolicies",
            "iam:GetRole",
            "iam:GetRolePolicy",
            "iam:PassRole",
            "iam:SimulatePrincipalPolicy"
        ],
        "Resource": "arn:aws:iam::*:role/intern-lambda-execution-role"
```

- `ViewLogs` - 使用 CloudWatch Logs 來檢視字首為 `intern-` 的函數的日誌。

```
        "Action": [
            "logs:*"
        ],
        "Resource": "arn:aws:logs:*:*:log-group:/aws/lambda/intern-*"
```

此政策可讓使用者開始使用 Lambda，而不需使其他使用者的資源承受風險。它不允許使用者設定由觸發的函數或呼叫其他 AWS 服務，這需要更廣泛的 IAM 許可。這也不包含對不支援 CloudWatch 和 X-Ray 這類範圍受限政策之服務的許可。對這些服務使用唯讀政策來為使用者提供指標和追蹤資料的存取。

當您為函數設定觸發時，您需要存取 ，才能使用叫用函數 AWS 的服務。例如，若要設定 Amazon S3 觸發，您需要使用 Amazon S3 動作的許可來管理儲存貯體通知。這些許可包含在 AWSLambda_FullAccess 受管政策中。

授予使用者對 Lambda 層的存取權

使用身分型政策以允許使用者、使用者群組或角色對 Lambda 層執行操作。下列政策授予使用者建立 layer 以及搭配函數使用的許可。資源模式允許使用者使用任何 AWS 區域 和任何 layer 版本，只要 layer 的名稱以 開頭`test-`即可。

Example圖層開發政策

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "PublishLayers",
            "Effect": "Allow",
            "Action": [
                "lambda:PublishLayerVersion"
            ],
            "Resource": "arn:aws:lambda:*:*:layer:test-*"
        },
        {
            "Sid": "ManageLayerVersions",
            "Effect": "Allow",
            "Action": [
                "lambda:GetLayerVersion",
                "lambda:DeleteLayerVersion"
            ],
            "Resource": "arn:aws:lambda:*:*:layer:test-*:*"
        }
    ]
}
```

您也可以使用 `lambda:Layer` 條件來在函數建立和設定期間強制執行 layer 的使用。例如，您可以防止使用者使用其他帳戶發佈的 layer。下列政策會將條件新增至 `CreateFunction`，且 `UpdateFunctionConfiguration` 動作需要來自帳戶 `123456789012` 指定的任何 layer。

JSON

```
{
```

```
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ConfigureFunctions",
            "Effect": "Allow",
            "Action": [
                "lambda:CreateFunction",
                "lambda:UpdateFunctionConfiguration"
            ],
            "Resource": "*",
            "Condition": {
                "ForAllValues:StringLike": {
                    "lambda:Layer": [
                        "arn:aws:lambda:*:123456789012:layer:*:*"
                    ]
                }
            }
        }
    ]
}
```

若要確保條件是否套用，請確認沒有其他陳述式授予使用者這些動作的許可。

在 Lambda 中檢視資源型 IAM 政策

Lambda 支援對 Lambda 函數和層使用資源型許可政策。您可以使用資源型政策，將存取權授予其他 AWS 帳戶、組織或服務。以資源為基礎的政策適用於單一函式、版本、別名或 layer 版本。

Console

檢視函式以資源為基礎的政策

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇 Configuration (組態)，然後選擇 Permissions (權限)。
4. 向下捲動至 Resource-based policy (資源型政策)，然後選擇 View policy document (檢視政策文件)。以資源為基礎的政策會顯示當另一個帳戶 AWS 或服務嘗試存取 函數時所套用的許可。下列範例顯示的陳述式允許 Amazon S3 針對帳戶 123456789012 中名稱為 `amzn-s3-demo-bucket` 的儲存貯體，叫用名為 `my-function` 的函數。

Example資源型政策

```
{
    "Version":"2012-10-17",
    "Id": "default",
    "Statement": [
        {
            "Sid": "lambda-allow-s3-my-function",
            "Effect": "Allow",
            "Principal": {
              "Service": "s3.amazonaws.com"
            },
            "Action": "lambda:InvokeFunction",
            "Resource":  "arn:aws:lambda:us-east-2:123456789012:function:my-
function",
            "Condition": {
              "StringEquals": {
                "AWS:SourceAccount": "123456789012"
              },
              "ArnLike": {
                "AWS:SourceArn": "arn:aws:s3:::amzn-s3-demo-bucket"
              }
            }
        }
     ]
}
```

AWS CLI

若要檢查函式的以資源為基礎的政策，請使用 `get-policy` 命令。

```
aws lambda get-policy \
  --function-name my-function \
  --output text
```

您應該會看到下列輸出：

```
{"Version":"2012-10-17",       "Id":"default","Statement":
[{"Sid":"sns","Effect":"Allow","Principal":
```

```
{"Service":"s3.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambd
east-2:123456789012:function:my-function","Condition":{"ArnLike":
{"AWS:SourceArn":"arn:aws:sns:us-east-2:123456789012:lambda*"}}}]}
```

針對版本和別名，在函式名稱的後方附加版本編號或別名。

```
aws lambda get-policy --function-name my-function:PROD
```

若要將許可從函式中移除，請使用 remove-permission。

```
aws lambda remove-permission \
  --function-name example \
  --statement-id sns
```

使用 get-layer-version-policy 指令檢視層級上的許可。

```
aws lambda get-layer-version-policy \
  --layer-name my-layer \
  --version-number 3 \
  --output text
```

您應該會看到下列輸出：

```
b0cd9796-d4eb-4564-939f-de7fe0b42236    {"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:
west-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"
```

使用 remove-layer-version-permission 以從策略中移除陳述式。

```
aws lambda remove-layer-version-permission --layer-name my-layer --version-number 3
 --statement-id engineering-org
```

支援的 API 動作

下列 Lambda API 動作支援資源型政策：

- CreateAlias

- DeleteAlias
- DeleteFunction
- DeleteFunctionConcurrency
- DeleteFunctionEventInvokeConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias
- GetFunction
- GetFunctionConcurrency
- GetFunctionConfiguration
- GetFunctionEventInvokeConfig
- GetPolicy
- GetProvisionedConcurrencyConfig
- Invoke
- InvokeFunctionUrl (僅限許可)
- ListAliases
- ListFunctionEventInvokeConfigs
- ListProvisionedConcurrencyConfigs
- ListTags
- ListVersionsByFunction
- PublishVersion
- PutFunctionConcurrency
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig
- TagResource
- UntagResource
- UpdateAlias
- UpdateFunctionCode
- UpdateFunctionEventInvokeConfig

授予 Lambda 函數存取權 AWS 服務

當您使用 AWS 服務來叫用函數時，您可以在以資源為基礎的政策的陳述式中授予許可。您可以將陳述式套用至整個函數，或將陳述式限制在單一版本或別名。

> **Note**
>
> 當您透過Lambda 主控台將觸發新增至函數時，主控台會更新函數的以資源為基礎的政策來允許服務進行叫用。若要將許可授予無法在 Lambda 主控台中使用的其他帳戶或服務，您可使用 AWS CLI。

使用 add-permission 命令新增陳述式。最簡單的以資源為基礎的政策陳述式可讓服務叫用函式。下列命令可授予 Amazon Simple Notification Service 調用名為 `my-function` 之函數的許可。

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id sns \
  --principal sns.amazonaws.com \
  --output text
```

您應該會看到下列輸出：

```
{"Sid":"sns","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lambda:us-
east-2:123456789012:function:my-function"}
```

這可讓 Amazon SNS 呼叫函數的 Invoke API 動作，但不會限制會觸發調用的 Amazon SNS 主題。若要確保您的函式只會被特定的資源叫用，請使用 `source-arn` 選項來指定資源的 Amazon Resource Name (ARN)。下列命令僅允許 Amazon SNS 為名為 `my-topic` 的主題訂閱叫用函數。

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id sns-my-topic \
  --principal sns.amazonaws.com \
  --source-arn arn:aws:sns:us-east-2:123456789012:my-topic
```

有些服務可在不同的帳戶中叫用函式。如果您指定的來源 ARN 中有帳戶 ID，您可以放心。然而，對於 Amazon S3，來源是一種儲存貯體，其 ARN 沒有帳戶 ID。您可以刪除該儲存貯體，且其他帳戶可使用相同的名稱建立儲存貯體。透過您的帳戶 ID 使用 `source-account` 選項來確保只有帳戶中的資源可叫用該函式。

```
aws lambda add-permission \
  --function-name my-function \
  --action lambda:InvokeFunction \
  --statement-id s3-account \
  --principal s3.amazonaws.com \
  --source-arn arn:aws:s3:::amzn-s3-demo-bucket \
  --source-account 123456789012
```

授予組織對函數的存取

若要向 AWS Organizations 中的組織授予許可，需將組織 ID 指定為 `principal-org-id`。下列 add-permission 命令將調用存取權授予組織 `o-a1b2c3d4e5f` 中的所有使用者。

```
aws lambda add-permission \
  --function-name example \
  --statement-id PrincipalOrgIDExample \
  --action lambda:InvokeFunction \
  --principal * \
  --principal-org-id o-a1b2c3d4e5f
```

> **Note**
>
> 在此命令中，`Principal` 是 `*`。這意味著組織 `o-a1b2c3d4e5f` 中的所有使用者都會取得函數叫用許可。如果您將 AWS 帳戶 或 角色指定為 `Principal`，則只有該委託人才能取得函數叫用許可，但前提是他們也是`o-a1b2c3d4e5f`組織的一部分。

此命令建立類似以下的資源型政策：

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
```

```
        {
            "Sid": "PrincipalOrgIDExample",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-east-2:123456789012:function:example",
            "Condition": {
                "StringEquals": {
                    "aws:PrincipalOrgID": "o-a1b2c3d4e5f"
                }
            }
        }
    ]
}
```

如需詳細資訊，請參閱《IAM 使用者指南》中的 aws:PrincipalOrgID。

授予 Lambda 函數對其他帳戶的存取權

若要與另一個 共用函數 AWS 帳戶，請將跨帳戶許可陳述式新增至函數的資源型政策。執行 add-permission 命令，並將帳戶 ID 指定為 `principal`。以下範例會授予帳戶 `111122223333` 以 `prod` 別名叫用 `my-function` 的許可。

```
aws lambda add-permission \
  --function-name my-function:prod \
  --statement-id xaccount \
  --action lambda:InvokeFunction \
  --principal 111122223333 \
  --output text
```

您應該會看到下列輸出：

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:l
east-1:123456789012:function:my-function"}
```

以資源為基礎的政策會授予其他帳戶存取函數的許可，但不允許該帳戶中的使用者超過其許可。另一個帳戶中的使用者必須具有相對應的使用者許可才能使用 Lambda API。

若要限制另一個帳戶中使用者或角色的存取，請指定身分的完整 ARN 作為主體。例如：`arn:aws:iam::123456789012:user/developer`。

別名會限制其他帳戶可叫用的版本。這需要其他帳戶在函式 ARN 中包含別名。

```
aws lambda invoke \
  --function-name arn:aws:lambda:us-east-2:123456789012:function:my-function:prod out
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "1"
}
```

然後，函數擁有者可以更新別名以指向新版本，而無需呼叫者變更他們叫用函數的方式。這可確保另一個帳戶不需要變更其程式碼來使用新版本，而且它只具有叫用與別名關聯之函數版本的許可。

您可以為在現有函式上操作之大部分 API 動作授予跨帳戶存取。例如，您可以授予對 `lambda:ListAliases` 的存取，來讓帳戶取得別名的清單，或 `lambda:GetFunction` 讓他們下載函式程式碼。個別新增每個許可，或使用 `lambda:*` 來授予對指定函式進行所有動作的存取。

若要授予其他帳戶多個動作或是不在函數上操作之動作的許可，我們建議您請使用 IAM 角色。

將 Lambda 層存取權授予其他帳戶

若要與另一個 共用 layer AWS 帳戶，請將跨帳戶許可陳述式新增至 layer 的資源型政策。執行 add-layer-version-permission 命令，並將帳戶 ID 指定為 `principal`。在各陳述式中，您可將許可授予單一帳戶、所有帳戶或 AWS Organizations 中的某個組織。

以下列範例會授予帳戶 111122223333 存取 `bash-runtime` layer 第 2 版的許可。

```
aws lambda add-layer-version-permission \
  --layer-name bash-runtime \
  --version-number 2 \
  --statement-id xaccount \
  --action lambda:GetLayerVersion \
  --principal 111122223333 \
  --output text
```

您應該會看到類似下列的輸出：

```
{"Sid":"xaccount","Effect":"Allow","Principal":
{"AWS":"arn:aws:iam::111122223333:root"},"Action":"lambda:GetLayerVersion","Resource":"arn:aws:
east-1:123456789012:layer:bash-runtime:2"}
```

許可僅適用於單一層版本。每次建立新的層版本時均需重複此程序。

若要將許可授予 AWS Organizations 組織中的所有帳戶，請使用 `organization-id` 選項。以下範例授予組織 `o-t194hfs8cz` 內的所有帳戶使用第 3 版 `my-layer` 的許可。

```
aws lambda add-layer-version-permission \
  --layer-name my-layer \
  --version-number 3 \
  --statement-id engineering-org \
  --principal '*' \
  --action lambda:GetLayerVersion \
  --organization-id o-t194hfs8cz \
  --output text
```

您應該會看到下列輸出：

```
{"Sid":"engineering-
org","Effect":"Allow","Principal":"*","Action":"lambda:GetLayerVersion","Resource":"arn:aws:lam
east-2:123456789012:layer:my-layer:3","Condition":{"StringEquals":
{"aws:PrincipalOrgID":"o-t194hfs8cz"}}}"
```

若要授予多個帳戶或組織許可，您必須新增多個陳述式。

使用 Lambda 中的屬性型存取控制

搭配屬性型存取控制 (ABAC)，您可以使用標籤來控制對 Lambda 資源的存取。您可以將標籤連接至特定 Lambda 資源、將其連接至特定 API 請求，或將其連接至提出請求的 AWS Identity and Access Management (IAM) 委託人。如需如何 AWS 授予屬性型存取的詳細資訊，請參閱《IAM 使用者指南》中的使用標籤控制對 AWS 資源的存取。

您可以使用 ABAC 來授予最低權限，無需在 IAM 政策中指定 Amazon Resource Name (ARN) 或 ARN 模式。可以在 IAM 政策 的條件元素中指定標籤，來控制存取。使用 ABAC 可以更輕鬆地擴展，因為在建立新資源時，您無需更新 IAM 政策，而是將標籤新增至新資源以控制存取。

在 Lambda 中，標籤適用於以下資源：

- 函數 – 如需標記函數的詳細資訊，請參閱the section called “Tags (標籤)”。
- 程式碼簽署組態 – 如需標記程式碼簽署組態的詳細資訊，請參閱the section called “程式碼簽署組態標籤”。
- 事件來源映射 – 如需標記事件來源映射的詳細資訊，請參閱the section called “事件來源映射標籤”。

層不支援標籤。

您可以使用以下條件索引鍵，根據標籤撰寫 IAM 政策規則：

- aws:ResourceTag/tag-key：依據連接至 Lambda 資源的標籤來控制存取。
- aws:RequestTag/tag-key：要求請求中有標籤，例如在建立新函數時。
- aws:PrincipalTag/tag-key：依據連接至其 IAM 使用者 或角色的標籤，控制 IAM 主體 (提出請求者) 允許執行的操作。
- aws:TagKeys：控制可否於請求中使用特定標籤索引鍵。

只能為支援它們的動作指定條件。如需每個 Lambda 動作所支援的條件清單，請參閱《服務授權參考》中的 Actions, resources, and condition keys for AWS Lambda。如需 aws：ResourceTag/tag-key 支援，請參閱「由 定義的資源類型」 AWS Lambda。如需 aws：RequestTag/tag-key 和 aws：TagKeys 支援，請參閱「由 定義的動作」 AWS Lambda。

主題

- 依標籤保護您的函數

依標籤保護您的函數

以下步驟會示範使用 ABAC 設定函數許可的一種方法。在此範例案例中，您將會建立四個 IAM 許可政策。然後，您會將這些政策連接至新的 IAM 角色。最後，您將建立 IAM 使用者，並授予該使用者擔任新角色的許可。

主題

- 先決條件
- 步驟 1：要求新函數具有標籤
- 步驟 2：依據連接至 Lambda 函數和 IAM 主體的標籤來允許動作
- 步驟 3：授予 List 許可

- 步驟 4：授予 IAM 許可
- 步驟 5：建立 IAM 角色
- 步驟 6：建立 IAM 使用者
- 步驟 7：測試許可
- 步驟 8：清理資源

先決條件

請確定您擁有 Lambda 執行角色。您將會在授予 IAM 許可以及建立 Lambda 函數時使用此角色。

步驟 1：要求新函數具有標籤

在搭配使用 ABAC 和 Lambda 時，最佳實務是要求所有函數都具有標籤。這有助於確保您的 ABAC 許可政策如預期般運作。

建立與以下範例類似的 IAM 政策。此政策會使用 aws:RequestTag/tag-key、aws:ResourceTag/tag-key 與 aws:TagKeys 條件金鑰，要求新建立的函式以及建立這些函式的 IAM 主體都必須具有 `project` 標籤。`ForAllValues` 修飾詞會確保 `project` 是唯一受允許的標籤。如果您不納入 `ForAllValues` 修飾詞，則只要使用者也傳遞 `project`，其便也可新增其他標籤至函數。

Example– 要求新函數具有標籤

JSON

```
{
    "Version":"2012-10-17",
    "Statement": {
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction",
        "lambda:TagResource"
      ],
      "Resource": "arn:aws:lambda:*:*:function:*",
      "Condition": {
        "StringEquals": {
          "aws:RequestTag/project": "${aws:PrincipalTag/project}",
          "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
        },
        "ForAllValues:StringEquals": {
```

```
          "aws:TagKeys": "project"
        }
      }
    }
  }
```

步驟 2：依據連接至 Lambda 函數和 IAM 主體的標籤來允許動作

使用 aws:ResourceTag/tag-key 條件索引鍵來建立第二個 IAM 政策，以要求主體的標籤與連接至函數的標籤相符。以下範例政策會允許具有 `project` 標籤的主體呼叫具有 `project` 標籤的函數。如果函數有任何其他標籤，則會該動作將遭拒。

Example– 要求函數和 IAM 主體的標籤相符

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Effect": "Allow",
        "Action": [
          "lambda:InvokeFunction",
          "lambda:GetFunction"
        ],
        "Resource": "arn:aws:lambda:*:*:function:*",
        "Condition": {
          "StringEquals": {
            "aws:ResourceTag/project": "${aws:PrincipalTag/project}"
          }
        }
      }
    ]
  }
```

步驟 3：授予 List 許可

建立允許主體列出 Lambda 函數和 IAM 角色的政策。此政策會允許主體在主控台以及呼叫 API 動作時查看所有 Lambda 函數和 IAM 角色。

Example– 授予 Lambda 和 IAM 的 List 許可

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
      {
        "Sid": "AllResourcesLambdaNoTags",
        "Effect": "Allow",
        "Action": [
          "lambda:GetAccountSettings",
          "lambda:ListFunctions",
          "iam:ListRoles"
        ],
        "Resource": "*"
      }
    ]
  }
```

步驟 4：授予 IAM 許可

建立允許 iam:PassRole 的政策。在您將執行角色指派給函數時，便會需要此許可。在以下範例政策中，用您 Lambda 執行角色的 ARN 來取代範例 ARN。

> **Note**
>
> 請勿將政策中的 `ResourceTag` 條件金鑰與 `iam:PassRole` 動作搭配使用。您不能使用該 IAM 角色的標籤來控制可傳遞該角色的人員。如需將角色傳遞至服務所需的許可的詳細資訊，請參閱授予使用者將角色傳遞至 AWS 服務的許可。

Example– 授予傳遞執行角色的許可

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
```

```
      {
        "Sid": "VisualEditor0",
        "Effect": "Allow",
        "Action": [
          "iam:PassRole"
        ],
        "Resource": "arn:aws:iam::111122223333:role/lambda-ex"
      }
    ]
  }
```

步驟 5：建立 IAM 角色

最佳實務為使用角色來委派許可。建立名為 abac-project-role 的 IAM 角色：

- 在步驟 1：選取可信任實體時：選擇 AWS account (帳戶)，然後選擇 This account (此帳戶)。
- 在步驟 2：新增許可時：連接您於前一步中建立的四個 IAM 政策。
- 在步驟 3：命名、檢閱和建立時：選擇 Add tag (新增標籤)。在 Key (索引鍵) 欄位，輸入 `project`。請勿輸入值。

步驟 6：建立 IAM 使用者

建立名為 abac-test-user 的 IAM 使用者。在 Set permissions (設定許可) 區段中，選擇 Attach existing policies directly (直接連接現有政策)，接著選擇 Create policy (建立政策)。輸入以下政策定義。以您的 AWS 帳戶 ID 來取代 *111122223333*。此政策允許 `abac-test-user` 擔任 `abac-project-role`。

Example– 允許 IAM 使用者擔任 ABAC 角色

JSON

```
{
    "Version":"2012-10-17",
    "Statement": {
      "Effect": "Allow",
      "Action": "sts:AssumeRole",
      "Resource": "arn:aws:iam::111122223333:role/abac-project-role"
    }
  }
```

步驟 7：測試許可

1. 以 身分登入 AWS 主控台`abac-test-user`。如需詳細資訊，請參閱以 IAM 使用者身分登入。
2. 切換到 `abac-project-role` 角色。如需詳細資訊，請參閱切換到角色 (主控台)。
3. 建立 Lambda 函數。
   - 在 Permissions (許可) 下，選擇 Change default execution role (變更預設執行角色)，然後在 Execution role (執行角色) 中選擇 Use an existing role (使用現有角色)。選擇與您在 步驟 4：授予 IAM 許可 中使用的相同執行角色。
   - 在 Advanced settings (進階設定)下，選擇 Enable tags (啟用標籤)，然後選擇 Add new tag (新增標籤)。在 Key (索引鍵) 欄位，輸入 `project`。請勿輸入值。
4. 測試函數。
5. 建立第二個 Lambda 函數，並新增一個不同的標籤，例如 `environment`。此操作應會失敗，因為您在 步驟 1：要求新函數具有標籤 中建立的 ABAC 政策僅允許主體建立具有 `project` 標籤的函數。
6. 建立無標籤的第三個函數。此操作應會失敗，因為您在 步驟 1：要求新函數具有標籤 中建立的 ABAC 政策不允許主體建立無標籤的函數。

此授權策略允許您控制存取，且無需為每個新使用者建立新政策。若要將授予新使用者存取權限，只需要向其授予擔任與其受指派專案相對應角色的許可。

步驟 8：清理資源

刪除 IAM 角色

1. 開啟 IAM 主控台中的角色頁面。
2. 選取您在步驟 5 中建立的角色。
3. 選擇 刪除。
4. 若要確認刪除，請在文字輸入欄位中輸入角色名稱。
5. 選擇 刪除。

若要刪除 IAM 使用者

1. 開啟 IAM 主控台的使用者頁面。
2. 選取您在步驟 6 中建立的 IAM 使用者。

3. 選擇 刪除。
4. 若要確認刪除，請在文字輸入欄位中輸入使用者名稱。
5. 選擇刪除使用者。

若要刪除 Lambda 函數

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇您建立的函數。
3. 選擇 Actions (動作)、Delete (刪除)。
4. 在文字輸入欄位中輸入 **`confirm`**，然後選擇 刪除 。

微調政策的資源和條件區段

您可以在 AWS Identity and Access Management (IAM) 政策中指定資源和條件，來限制使用者許可的範圍。政策中的每個動作都支援資源和條件類型組合，組合內容取決於動作的行為。

每個 IAM 政策陳述式授予在資源上執行動作的許可。當動作不在具名資源上執行動作，或是當您授予對所有資源執行動作的許可，政策中資源的值是萬用字元 (*)。對於許多動作，您可以透過指定資源的 Amazon Resource Name (ARN) 或符合多個資源的 ARN 模式，來限制使用者可以修改的資源。

根據資源類型，關於如何限制動作範圍的一般設計如下：

- 函數 - 在函數上執行的動作可依據函數、版本或別名 ARN 限定於特定函數。
- 事件來源映射 - 動作可依據 ARN 限定於特定事件來源映射資源。事件來源映射一律與函數相關聯。也可以使用 `lambda:FunctionArn` 條件，依據相關的函數限定動作。
- 層 - 與 Layer 使用和許可相關的動作作用於層的某個版本。
- 程式碼簽署組態 - 動作可依據 ARN 限定於特定程式碼簽署組態資源。
- 標籤 - 使用標準標籤條件。如需詳細資訊，請參閱the section called “屬性型存取控制”。

若要依照資源限制許可，請依照 ARN 指定資源。

Lambda 資源 ARN 格式

- 函數 - `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*
- 函數版本 - `arn:aws:lambda:`*`us-west-2`*`:`*`123456789012`*`:function:`*`my-function`*`:`*`1`*

- 函數別名 - arn:aws:lambda:*us-west-2*:*123456789012*:function:*my-function*:*TEST*
- 事件來源映射 - arn:aws:lambda:*us-west-2*:*123456789012*:event-source-mapping:*fa123456-14a1-4fd2-9fec-83de64ad683de6d47*
- 層 - arn:aws:lambda:*us-west-2*:*123456789012*:layer:*my-layer*
- 層版本 - arn:aws:lambda:*us-west-2*:*123456789012*:layer:*my-layer*:*1*
- 程式碼簽署組態 – arn:aws:lambda:*us-west-2*:*123456789012*:code-signing-config:*my-csc*

例如，以下政策允許使用者在 AWS 帳戶 123456789012my-function中呼叫美國西部 （奧勒岡） AWS 區域中名為 的函數。

Example呼叫函數政策

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "Invoke",
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:my-
function"
        }
    ]
}
```

此為特例，其中動作識別符 (lambda:InvokeFunction) 與 API 操作 (Invoke) 不同。對於其他動作，動作識別符就是開頭為 lambda: 的操作名稱。

章節

- 了解政策中的條件區段
- 參考政策資源區段中的函數
- 支援的 IAM 動作和函數行為

了解政策中的條件區段

條件是選用的政策元素，會套用額外的邏輯來判斷是否允許動作。除了所有動作皆支援的一般條件之外，Lambda 還會定義您可用來限制某些動作之其他參數值的條件類型。

例如，`lambda:Principal` 條件讓您可在函數的資源型政策中，限制使用者可授予叫用存取權限的服務或帳號。以下政策會讓使用者可將許可授予 Amazon Simple Notification Service (Amazon SNS) 主題以叫用名為 `test` 的函數。

Example管理函數政策許可

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Sid": "ManageFunctionPolicy",
            "Effect": "Allow",
            "Action": [
                "lambda:AddPermission",
                "lambda:RemovePermission"
            ],
            "Resource": "arn:aws:lambda:us-west-2:123456789012:function:test:*",
            "Condition": {
                "StringEquals": {
                    "lambda:Principal": "sns.amazonaws.com"
                }
            }
        }
    ]
}
```

條件要求委託人是 Amazon SNS 而不是其他服務或帳戶。資源模式要求函數名稱是 `test` 並包含版本編號或別名。例如 `test:v1`。

如需 Lambda 和其他 AWS 服務的資源和條件的詳細資訊，請參閱《服務授權參考》中的 AWS 服務的動作、資源和條件索引鍵。

參考政策資源區段中的函數

您可以使用 Amazon Resource Name (ARN) 來參考政策陳述式中的 Lambda 函數。函數 ARN 的格式取決於您是參考整個函數 (不合格)、函數版本或別名 (合格)。

進行 Lambda API 呼叫時，使用者可以指定版本或別名，方法是在 GetFunction `FunctionName` 參數中傳遞版本 ARN 或別名 ARN，或者在 GetFunction `Qualifier` 參數中設定值。Lambda 會比較 IAM 政策中的資源元素與 API 呼叫中傳遞的 `FunctionName` 和 `Qualifier`，從而作出授權決策。如果不符，Lambda 會拒絕該請求。

無論是允許還是拒絕對函數執行操作，都必須在政策陳述式中使用正確的函數 ARN 類型，才能達到預期的結果。例如，如果您的政策引用了不合格的 ARN，Lambda 會接受參考不合格 ARN 的請求，但拒絕參考合格 ARN 的請求。

> **Note**
>
> 不能使用萬用字元 (*) 以讓帳戶 ID 相符。如需有關已接受語法的詳細資訊，請參閱 IAM User Guide 中的 IAM JSON policy reference。

Example允許叫用不合格的 ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction"
        }
    ]
}
```

如果您的政策引用了特定合格的 ARN，Lambda 會接受參考該 ARN 的請求，但拒絕參考不合格 ARN 或不同的合格 ARN 的請求，例如 `myFunction:2`。

Example允許叫用特定的合格 ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:1"
        }
    ]
}
```

如果您的政策使用 `:*` 參考任何合格的 ARN，Lambda 會接受任何合格的 ARN，但拒絕參考不合格 ARN 的請求。

Example允許叫用任何合格的 ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:*"
        }
    ]
}
```

如果您的政策使用 `*` 參考任何 ARN，Lambda 會接受任何合格或不合格的 ARN。

Example允許叫用任何合格或不合格的 ARN

JSON

```
{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "lambda:InvokeFunction",
            "Resource": "arn:aws:lambda:us-
west-2:123456789012:function:myFunction*"
        }
    ]
}
```

支援的 IAM 動作和函數行為

動作定義了可透過 IAM 政策許可的項目。若要查看 Lambda 中支援的動作清單，請參閱《服務授權參考》中的 Actions, resources, and condition keys for AWS Lambda。在大多數情況下，當 IAM 動作許可某個 Lambda API 動作時，該 IAM 動作的名稱會與 Lambda API 動作的名稱相同，但以下情況除外：

API 動作	IAM 動作
Invoke	`lambda:InvokeFunction`
GetLayerVersion GetLayerVersionByArn	`lambda:GetLayerVersion`

除了 Service Authorization Reference 中定義的資源和條件之外，Lambda 還針對特定動作支援下列資源和條件。其中許多與政策資源區段中的參考函數有關。您可依據函數、版本或別名 ARN，將在函數上操作的動作限於特定函數，如下表所述。

Action	資源	條件
AddPermission	函式版本	N/A

Action	資源	條件
RemovePermission 調用：(許可：`lambda:InvokeFunction`)	函式別名	
UpdateFunctionConfiguration	N/A	`lambda:CodeSigningConfigArn`
CreateFunctionUrlConfig DeleteFunctionUrlConfig GetFunctionUrlConfig UpdateFunctionUrlConfig	函式別名	N/A

中的安全性 AWS Lambda

的雲端安全性 AWS 是最高優先順序。身為 AWS 客戶，您可以受益於資料中心和網路架構，這些架構專為滿足最安全敏感組織的需求而建置。

安全性是 AWS 與您之間的共同責任。共同責任模型 將此描述為雲端的安全和雲端內的安全：

- 雲端的安全性 – AWS 負責保護在 AWS Cloud AWS 服務 中執行的基礎設施。 AWS 也為您提供可安全使用的服務。在 AWS 合規計畫中，第三方稽核員會定期測試並驗證我們的安全功效。若要了解適用的合規計劃 AWS Lambda，請參閱AWS 服務 合規計劃範圍中的 。
- 雲端內部的安全 – 您的責任取決於所使用的 AWS 服務。您也必須對其他因素負責，包括資料的機密性、您公司的要求和適用法律和法規。

本文件有助於您了解如何在使用 Lambda 時套用共同責任模型。下列主題將示範如何設定 Lambda 以達到您的安全和合規目標。您也會了解如何使用其他 AWS 服務 來協助您監控和保護 Lambda 資源。

如需有關將安全性原則套用至 Lambda 應用程式的詳細資訊，請參閱無伺服器園地中的安全性。

主題

- AWS Lambda 中的資料保護
- 使用 Lambda 的服務連結角色
- 的 Identity and Access Management AWS Lambda
- 建立 Lambda 函數和層的控管策略
- 的合規驗證AWS Lambda
- AWS Lambda 中的恢復能力
- 中的基礎設施安全AWS Lambda
- 使用公有端點保護工作負載
- 使用程式碼簽署來驗證 Lambda 的程式碼完整性

AWS Lambda 中的資料保護

AWS 共同的責任模型適用於 AWS Lambda 中的資料保護。如此模型所述，AWS 負責保護執行所有 AWS 雲端 的全球基礎設施。您負責維護在此基礎設施上託管內容的控制權。您也同時負責所使用 AWS 服務 的安全組態和管理任務。如需資料隱私權的詳細資訊，請參閱資料隱私權常見問答集。如需

有關歐洲資料保護的相關資訊，請參閱 AWS 安全性部落格上的 AWS 共同的責任模型和 GDPR 部落格文章。

基於資料保護目的，建議您使用 AWS IAM Identity Center 或 AWS Identity and Access Management (IAM) 保護 AWS 帳戶 憑證，並設定個人使用者。如此一來，每個使用者都只會獲得授與完成其任務所必須的許可。我們也建議您採用下列方式保護資料：

- 每個帳戶均要使用多重要素驗證 (MFA)。
- 使用 SSL/TLS 與 AWS 資源通訊。我們需要 TLS 1.2 並建議使用 TLS 1.3。
- 使用 AWS CloudTrail 設定 API 和使用者活動日誌記錄。如需使用 CloudTrail 追蹤擷取 AWS 活動的相關資訊，請參閱「AWS CloudTrail 使用者指南」中的使用 CloudTrail 追蹤。
- 使用 AWS 加密解決方案，以及 AWS 服務 內的所有預設安全控制項。
- 使用進階的受管安全服務 (例如 Amazon Macie)，協助探索和保護儲存在 Amazon S3 的敏感資料。
- 如果您在透過命令列介面或 API 存取 AWS 時，需要 FIPS 140-3 驗證的加密模組，請使用 FIPS 端點。如需有關 FIPS 和 FIPS 端點的更多相關資訊，請參閱聯邦資訊處理標準 (FIPS) 140-3。

我們強烈建議您絕對不要將客戶的電子郵件地址等機密或敏感資訊，放在標籤或自由格式的文字欄位中，例如名稱欄位。這包括透過主控台、API、AWS CLI 或 AWS SDK，使用 Lambda 或其他 AWS 服務時。您在標籤或自由格式文字欄位中輸入的任何資料都可能用於計費或診斷日誌。如果您提供外部伺服器的 URL，我們強烈建議請勿在驗證您對該伺服器請求的 URL 中包含憑證資訊。

章節

- 傳輸中加密
- AWS Lambda 的靜態資料加密

傳輸中加密

Lambda API 端點僅支援透過 HTTPS 的安全連線。當您使用 AWS 管理主控台、AWS 開發套件或 Lambda API 來管理 資源時，所有通訊都會使用 Transport Layer Security (TLS) 加密。如需完整的 API 端點清單，請參閱《AWS 一般參考》中的AWS 區域和端點。

當您將您的函數連線至檔案系統時，Lambda 會針對所有連線使用傳輸中加密。如需詳細資訊，請參閱 Amazon Elastic File System 使用者指南中的 Amazon EFS 的資料加密。

使用環境變數時，您可以啟用主控台加密協助程式以使用用戶端加密來保護傳輸中的環境變數。如需更多詳細資訊，請參閱 保護 Lambda 環境變數。

AWS Lambda 的靜態資料加密

Lambda 一律使用 AWS 擁有的金鑰或 AWS 受管金鑰 為下列資源提供靜態加密：

- 環境變數
- 您上傳到 Lambda 的檔案，包括部署套件和層封存
- 事件來源映射篩選條件物件

您可以選擇將 Lambda 設定為使用客戶受管金鑰來加密環境變數、.zip 部署套件和篩選條件物件。

根據預設，Amazon CloudWatch Logs 和 AWS X-Ray 也可加密資料，並可設定為使用客戶受管金鑰。如需詳細資訊，請參閱在 CloudWatch Logs 中加密日誌資料和AWS X-Ray 中的資料保護。

監控 Lambda 的加密金鑰

當您搭配 Lambda 使用 AWS KMS 客戶受管金鑰時，您可以使用 AWS CloudTrail。下列範例是 Lambda 為存取客戶受管金鑰加密的資料而進行的 `Decrypt`、 `DescribeKey` 和 `GenerateDataKey` 呼叫的 CloudTrail 事件。

Decrypt

如果您使用 AWS KMS 客戶受管金鑰來加密篩選條件物件，Lambda 會在您嘗試以純文字存取該金鑰時 (例如，從 `ListEventSourceMappings` 呼叫)，代表您傳送 `Decrypt` 請求。下面的範例事件會記錄 `Decrypt` 操作：

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
```

```
            "attributes": {
                "creationDate": "2024-05-30T00:45:23Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "lambda.amazonaws.com"
    },
    "eventTime": "2024-05-30T01:05:46Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "Decrypt",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "lambda.amazonaws.com",
    "userAgent": "lambda.amazonaws.com",
    "requestParameters": {
        "keyId": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "encryptionContext": {
            "aws-crypto-public-key": "ABCD
+7876787678+CDEFGHIJKL/888666888999888555444111555222888333111==",
            "aws:lambda:EventSourceArn": "arn:aws:sqs:eu-west-1:123456789012:sample-
source",
            "aws:lambda:FunctionArn": "arn:aws:lambda:eu-
west-1:123456789012:function:sample-function"
        },
        "encryptionAlgorithm": "SYMMETRIC_DEFAULT"
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "sessionCredentialFromConsole": "true"
```

```
}
```

DescribeKey

如果您使用 AWS KMS 客戶受管金鑰來加密篩選條件物件，Lambda 會在您嘗試存取該金鑰時 (例如，從 GetEventSourceMapping 呼叫)，代表您傳送 DescribeKey 請求。下面的範例事件會記錄 DescribeKey 操作：

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
            },
            "attributes": {
                "creationDate": "2024-05-30T00:45:23Z",
                "mfaAuthenticated": "false"
            }
        }
    },
    "eventTime": "2024-05-30T01:09:40Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "DescribeKey",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "54.240.197.238",
    "userAgent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36
 (KHTML, like Gecko) Chrome/125.0.0.0 Safari/537.36",
    "requestParameters": {
        "keyId": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111"
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
```

```
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management",
    "tlsDetails": {
        "tlsVersion": "TLSv1.3",
        "cipherSuite": "TLS_AES_256_GCM_SHA384",
        "clientProvidedHostHeader": "kms.eu-west-1.amazonaws.com"
    },
    "sessionCredentialFromConsole": "true"
}
```

GenerateDataKey

當您使用 AWS KMS 客戶受管金鑰來加密 CreateEventSourceMapping 或 UpdateEventSourceMapping 呼叫中的篩選條件物件時，Lambda 會代表您傳送 GenerateDataKey 請求，以產生資料金鑰來加密篩選條件 (封套加密)。下面的範例事件會記錄 GenerateDataKey 操作：

```
{
    "eventVersion": "1.09",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "AROA123456789EXAMPLE:example",
        "arn": "arn:aws:sts::123456789012:assumed-role/role-name/example",
        "accountId": "123456789012",
        "accessKeyId": "ASIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AROA123456789EXAMPLE",
                "arn": "arn:aws:iam::123456789012:role/role-name",
                "accountId": "123456789012",
                "userName": "role-name"
```

```
            },
            "attributes": {
                "creationDate": "2024-05-30T00:06:07Z",
                "mfaAuthenticated": "false"
            }
        },
        "invokedBy": "lambda.amazonaws.com"
    },
    "eventTime": "2024-05-30T01:04:18Z",
    "eventSource": "kms.amazonaws.com",
    "eventName": "GenerateDataKey",
    "awsRegion": "eu-west-1",
    "sourceIPAddress": "lambda.amazonaws.com",
    "userAgent": "lambda.amazonaws.com",
    "requestParameters": {
        "numberOfBytes": 32,
        "keyId": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111",
        "encryptionContext": {
            "aws-crypto-public-key": "ABCD
+7876787678+CDEFGHIJKL/888666888999888555444111555222888333111==",
            "aws:lambda:EventSourceArn": "arn:aws:sqs:eu-west-1:123456789012:sample-
source",
            "aws:lambda:FunctionArn": "arn:aws:lambda:eu-
west-1:123456789012:function:sample-function"
        },
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEaaaaa",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLEbbbbb",
    "readOnly": true,
    "resources": [
        {
            "accountId": "AWS Internal",
            "type": "AWS::KMS::Key",
            "ARN": "arn:aws:kms:eu-west-1:123456789012:key/a1b2c3d4-5678-90ab-cdef-
EXAMPLE11111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "123456789012",
    "eventCategory": "Management"
```

```
}
```

使用 Lambda 的服務連結角色

Lambda 使用 AWS Identity and Access Management (IAM) 服務連結角色。服務連結角色是直接連結至 Lambda 的唯一 IAM 角色類型。服務連結角色由 Lambda 預先定義，並包含服務代表您呼叫其他 AWS 服務所需的許可。

Lambda 定義其服務連結角色的許可，只有 Lambda 可以擔任其角色。定義的許可包括信任政策和許可政策，且該許可政策無法附加至其他 IAM 實體。

您必須先刪除服務連結角色的相關資源，才能將其刪除。這可保護您的 Lambda 資源，因為您不會不小心移除存取資源的許可。

如需有關支援服務連結角色的其他 服務的資訊，請參閱AWS 使用 IAM 的服務，並在服務連結角色欄中尋找具有是的服務。選擇具有連結的是，以檢視該服務的服務連結角色文件。

Lambda 的服務連結角色許可

Lambda 使用名為 AWSServiceRoleForLambda 的服務連結角色。 服務連結角色信任下列服務以擔任角色：

- `lambda.amazonaws.com`

名為 AWSLambdaServiceRolePolicy 的角色許可政策可讓 Lambda 對指定的資源完成下列動作：

- 動作：`ec2:TerminateInstances`在 上`arn:aws:ec2:*:*:instance/*`，條件`ec2:ManagedResourceOperator`等於 `scaler.lambda.amazonaws.com`
- 動作：`ec2:DescribeInstanceStatus` 和 `*` 的 `ec2:DescribeInstances`

您必須設定許可，以允許您的使用者、群組或角色建立、編輯或刪除服務連結角色。如需詳細資訊，請參閱 IAM 使用者指南中的服務連結角色許可。

如需受管政策更新，請參閱 Lambda 受管政策。

為 Lambda 建立服務連結角色

您不需要手動建立服務連結角色，當您在 AWS 管理主控台、 AWS CLI或 AWS API 中建立 Lambda 容量提供者時，Lambda 會為您建立服務連結角色。

若您刪除此服務連結角色，之後需要再次建立，您可以在帳戶中使用相同程序重新建立角色。當您建立 Lambda 容量提供者時，Lambda 會再次為您建立服務連結角色。

您也可以使用 IAM 主控台建立具有 AWSServiceRoleForLambda 使用案例的服務連結角色。在 AWS CLI 或 AWS API 中，使用服務名稱建立`lambda.amazonaws.com`服務連結角色。如需詳細資訊，請參閱《IAM 使用者指南》中的「建立服務連結角色」。如果您刪除此服務連結角色，您可以使用此相同的程序以再次建立該角色。

編輯 Lambda 的服務連結角色

Lambda 不允許您編輯 AWSServiceRoleForLambda 服務連結角色。因為有各種實體可能會參考服務連結角色，所以您無法在建立角色之後變更角色名稱。然而，您可使用 IAM 來編輯角色描述。如需詳細資訊，請參閱《IAM 使用者指南》中的編輯服務連結角色。

刪除 Lambda 的服務連結角色

若您不再使用需要服務連結角色的功能或服務，我們建議您刪除該角色。如此一來，您就沒有未主動監控或維護的未使用實體。然而，在手動刪除服務連結角色之前，您必須先清除資源。

> **Note**
>
> 如果 Lambda 服務在您嘗試刪除資源時使用角色，則刪除可能會失敗。若此情況發生，請等待數分鐘後並再次嘗試操作。

刪除 AWSServiceRoleForLambda 使用的 Lambda 資源

1. 從您的帳戶中移除所有 Lambda 容量提供者。您可以使用 Lambda 主控台、CLI 或 API 來執行此操作。
2. 在嘗試刪除服務連結角色之前，請確認您的帳戶中沒有 Lambda 容量提供者。

使用 IAM 手動刪除服務連結角色

使用 IAM 主控台 AWS CLI、 或 AWS API 來刪除 AWSServiceRoleForLambda 服務連結角色。如需詳細資訊，請參閱《IAM 使用者指南》中的刪除服務連結角色。

Lambda 服務連結角色支援的區域

Lambda 不支援在提供服務的每個區域中使用服務連結角色。下列區域支援 AWSServiceRoleForLambda。

區域名稱	區域身分	Lambda 支援
美國東部 (維吉尼亞北部)	us-east-1	是
美國東部 (俄亥俄)	us-east-2	是
美國西部 (加利佛尼亞北部)	us-west-1	是
美國西部 (奧勒岡)	us-west-2	是
Africa (Cape Town)	af-south-1	否
亞太地區 (香港)	ap-east-1	是
亞太地區 (雅加達)	ap-southeast-3	是
亞太區域 （ 曼谷 ）	ap-southeast-7	是
亞太區域 (孟買)	ap-south-1	是
亞太區域 (大阪)	ap-northeast-3	否
亞太區域 (首爾)	ap-northeast-2	否
亞太區域 (新加坡)	ap-southeast-1	是
亞太區域 (雪梨)	ap-southeast-2	是
亞太區域 (東京)	ap-northeast-1	是
加拿大 (中部)	ca-central-1	否
歐洲 (法蘭克福)	eu-central-1	是
歐洲 (愛爾蘭)	eu-west-1	是
歐洲 (倫敦)	eu-west-2	是
歐洲 (米蘭)	eu-south-1	否
Europe (Paris)	eu-west-3	否

區域名稱	區域身分	Lambda 支援
Europe (Stockholm)	eu-north-1	否
Middle East (Bahrain)	me-south-1	否
中東 (阿拉伯聯合大公國)	me-central-1	否
南美洲 (聖保羅)	sa-east-1	否
AWS GovCloud （美國東部）	us-gov-east-1	否
AWS GovCloud （美國西部）	us-gov-west-1	否

的 Identity and Access Management AWS Lambda

AWS Identity and Access Management (IAM) 是一種 AWS 服務 ，可協助管理員安全地控制對 AWS 資源的存取。IAM 管理員會控制誰經過身分身分驗證 (已登入) 並得到授權 (具有許可) 來使用 Lambda 資源。IAM 是您可以免費使用 AWS 服務 的 。

主題

- 目標對象
- 使用身分驗證
- 使用政策管理存取權
- AWS Lambda 如何使用 IAM
- 的身分型政策範例 AWS Lambda
- AWS 的 受管政策 AWS Lambda
- 對 AWS Lambda 身分和存取進行故障診斷

目標對象

使用方式 AWS Identity and Access Management (IAM) 會根據您的角色而有所不同：

- 服務使用者 — 若無法存取某些功能，請向管理員申請所需許可 (請參閱 對 AWS Lambda 身分和存取進行故障診斷)

- 服務管理員 — 負責設定使用者存取權並提交相關許可請求 (請參閱 AWS Lambda 如何使用 IAM)
- IAM 管理員 — 撰寫政策以管理存取控制 (請參閱 的身分型政策範例 AWS Lambda)

使用身分驗證

身分驗證是您 AWS 使用身分憑證登入 的方式。您必須以 AWS 帳戶根使用者、IAM 使用者或擔任 IAM 角色身分進行身分驗證。

您可以使用身分來源的登入資料，例如 AWS IAM Identity Center (IAM Identity Center)、單一登入身分驗證或 Google/Facebook 登入資料，以聯合身分的形式登入。如需有關登入的詳細資訊，請參閱《AWS 登入 使用者指南》中的如何登入您的 AWS 帳戶。

對於程式設計存取， AWS 提供 SDK 和 CLI 以密碼編譯方式簽署請求。如需詳細資訊，請參閱《IAM 使用者指南》中的 API 請求的AWS 第 4 版簽署程序。

AWS 帳戶 根使用者

當您建立 時 AWS 帳戶，您會從一個名為 AWS 帳戶 theroot 使用者的登入身分開始，該身分具有對所有 AWS 服務 和 資源的完整存取權。強烈建議不要使用根使用者來執行日常任務。有關需要根使用者憑證的任務，請參閱《IAM 使用者指南》中的需要根使用者憑證的任務。

聯合身分

最佳實務是要求人類使用者使用聯合身分提供者，以 AWS 服務 使用臨時憑證存取 。

聯合身分是您企業目錄、Web 身分提供者的使用者，或使用來自身分來源的 AWS 服務 憑證存取 Directory Service 。聯合身分會擔任角色，而該角色會提供臨時憑證。

若需集中化管理存取權限，建議使用 AWS IAM Identity Center。如需詳細資訊，請參閱 AWS IAM Identity Center 使用者指南中的什麼是 IAM Identity Center？。

IAM 使用者和群組

IAM 使用者https://docs.aws.amazon.com/IAM/latest/UserGuide/id_users.html是一種身分具備單人或應用程式的特定許可權。建議以臨時憑證取代具備長期憑證的 IAM 使用者。如需詳細資訊，請參閱《IAM 使用者指南》中的要求人類使用者使用聯合身分提供者來 AWS 使用臨時憑證存取 。

IAM 群組會指定 IAM 使用者集合，使管理大量使用者的許可權更加輕鬆。如需詳細資訊，請參閱《IAM 使用者指南》中的 IAM 使用者的使用案例。

IAM 角色

IAM 角色https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html的身分具有特定許可權，其可以提供臨時憑證。您可以透過從使用者切換到 IAM 角色（主控台）或呼叫 AWS CLI 或 AWS API 操作來擔任角色。如需詳細資訊，請參閱《IAM 使用者指南》中的擔任角色的方法。

IAM 角色適用於聯合身分使用者存取、臨時 IAM 使用者許可、跨帳戶存取權與跨服務存取，以及在 Amazon EC2 執行的應用程式。如需詳細資訊，請參閱《IAM 使用者指南》中的 IAM 中的快帳戶資源存取。

使用政策管理存取權

您可以透過建立政策並將其連接到身分或資源 AWS 來控制 AWS 中的存取。政策定義與身分或資源相關聯的許可。當委託人提出請求時 AWS ， 會評估這些政策。大多數政策會以 JSON 文件 AWS 形式存放在 中。如需進一步了解 JSON 政策文件，請參閱《IAM 使用者指南》中的 JSON 政策概觀。

管理員會使用政策，透過定義哪些主體可在哪些條件下對哪些資源執行動作，以指定可存取的範圍。

預設情況下，使用者和角色沒有許可。IAM 管理員會建立 IAM 政策並將其新增至角色，供使用者後續擔任。IAM 政策定義動作的許可，無論採用何種方式執行。

身分型政策

身分型政策是附加至身分 (使用者、使用者群組或角色) 的 JSON 許可政策文件。這類政策控制身分可對哪些資源執行哪些動作，以及適用的條件。如需了解如何建立身分型政策，請參閱《IAM 使用者指南》中的透過客戶管理政策定義自訂 IAM 許可。

身分型政策可分為內嵌政策 (直接內嵌於單一身分) 與受管政策 (可附加至多個身分的獨立政策)。如需了解如何在受管政策及內嵌政策之間做選擇，請參閱《IAM 使用者指南》中的在受管政策與內嵌政策之間選擇。

資源型政策

資源型政策是附加到資源的 JSON 政策文件。範例包括 IAM 角色信任政策與 Amazon S3 儲存貯體政策。在支援資源型政策的服務中，服務管理員可以使用它們來控制對特定資源的存取權限。您必須在資源型政策中指定主體。

資源型政策是位於該服務中的內嵌政策。您無法在資源型政策中使用來自 IAM 的 AWS 受管政策。

其他政策類型

AWS 支援其他政策類型，可設定更多常見政策類型授予的最大許可：

- 許可界限 — 設定身分型政策可授與 IAM 實體的最大許可。如需詳細資訊，請參閱《 IAM 使用者指南》中的 IAM 實體許可界限。
- 服務控制政策 (SCP) — 為 AWS Organizations中的組織或組織單位指定最大許可。如需詳細資訊，請參閱《AWS Organizations 使用者指南》中的服務控制政策。
- 資源控制政策 (RCP) — 設定您帳戶中資源可用許可的上限。如需詳細資訊，請參閱《AWS Organizations 使用者指南》中的資源控制政策 (RCP)。
- 工作階段政策 — 在以程式設計方式為角色或聯合身分使用者建立臨時工作階段時，以參數形式傳遞的進階政策。如需詳細資訊，請參《IAM 使用者指南》中的工作階段政策。

多種政策類型

當多種類型的政策適用於請求時，產生的許可會更複雜而無法理解。若要了解如何 AWS 在涉及多種政策類型時決定是否允許請求，請參閱《IAM 使用者指南》中的政策評估邏輯。

AWS Lambda 如何使用 IAM

在您使用 IAM 管理 Lambda 的存取權之前，請了解搭配 Lambda 使用的 IAM 功能有哪些。

IAM 功能	Lambda 支援
身分型政策	是
資源型政策	是
政策動作	是
政策資源	是
政策條件索引鍵 (服務特定)	是
ACL	否
ABAC(政策中的標籤)	部分
臨時憑證	是
轉送存取工作階段 (FAS)	否

IAM 功能	Lambda 支援
服務角色	是
服務連結角色	部分

若要全面了解 Lambda 和其他 AWS 服務如何與大多數 IAM 功能搭配使用，請參閱《AWS IAM 使用者指南》中的與 IAM 搭配使用的 服務。

適用於 Lambda 的身分型政策

支援身分型政策：是

身分型政策是可以附加到身分 (例如 IAM 使用者、使用者群組或角色) 的 JSON 許可政策文件。這些政策可控制身分在何種條件下能對哪些資源執行哪些動作。如需了解如何建立身分型政策，請參閱《IAM 使用者指南》中的透過客戶管理政策定義自訂 IAM 許可。

使用 IAM 身分型政策，您可以指定允許或拒絕的動作和資源，以及在何種條件下允許或拒絕動作。如要了解您在 JSON 政策中使用的所有元素，請參閱《IAM 使用者指南》中的 IAM JSON 政策元素參考。

適用於 Lambda 的身分型政策範例

若要檢視 Lambda 身分型政策範例，請參閱的身分型政策範例 AWS Lambda。

Lambda 內的資源型政策

支援資源型政策：是

資源型政策是附加到資源的 JSON 政策文件。資源型政策的最常見範例是 IAM 角色信任政策和 Amazon S3 儲存貯體政策。在支援資源型政策的服務中，服務管理員可以使用它們來控制對特定資源的存取權限。對於附加政策的資源，政策會定義指定的主體可以對該資源執行的動作以及在何種條件下執行的動作。您必須在資源型政策中指定主體。委託人可以包含帳戶、使用者、角色、聯合身分使用者或 AWS 服務。

如需啟用跨帳戶存取權，您可以在其他帳戶內指定所有帳戶或 IAM 實體作為資源型政策的主體。如需詳細資訊，請參閱《IAM 使用者指南》中的 IAM 中的快帳戶資源存取。

您可以將資源型政策連接至 Lambda 函數或層。此政策定義哪些主體可以對函數或層執行動作。

若要了解如何將資源型政策連接到函數或層，請參閱在 Lambda 中檢視資源型 IAM 政策。

適用於 Lambda 的政策動作

支援政策動作：是

管理員可以使用 AWS JSON 政策來指定誰可以存取內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

JSON 政策的 `Action` 元素描述您可以用來允許或拒絕政策中存取的動作。政策會使用動作來授予執行相關聯動作的許可。

如要查看 Lambda 動作的清單，請參閱《服務授權參考》中的 Actions defined by AWS Lambda。

Lambda 中的政策動作會在動作之前使用以下字首：

```
lambda
```

若要在單一陳述式中指定多個動作，請用逗號分隔。

```
"Action": [
      "lambda:action1",
      "lambda:action2"
         ]
```

若要檢視 Lambda 身分型政策範例，請參閱的身分型政策範例 AWS Lambda。

適用於 Lambda 的政策資源

支援政策資源：是

管理員可以使用 AWS JSON 政策來指定誰可以存取內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

`Resource` JSON 政策元素可指定要套用動作的物件。最佳實務是使用其 Amazon Resource Name (ARN) 來指定資源。若動作不支援資源層級許可，使用萬用字元 (*) 表示該陳述式適用於所有資源。

```
"Resource": "*"
```

若要查看 Lambda 資源類型及其 ARN 的清單，請參閱《服務授權參考》中的 Resource types defined by AWS Lambda。若要了解您可以使用哪些動作指定每個資源的 ARN，請參閱 AWS Lambda定義的動作。

若要檢視 Lambda 身分型政策範例，請參閱的身分型政策範例 AWS Lambda。

適用於 Lambda 的政策條件索引鍵

支援服務特定政策條件金鑰：是

管理員可以使用 AWS JSON 政策來指定誰可以存取內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

`Condition` 元素會根據定義的條件，指定陳述式的執行時機。您可以建立使用條件運算子的條件運算式 (例如等於或小於)，來比對政策中的條件和請求中的值。若要查看所有 AWS 全域條件索引鍵，請參閱《IAM 使用者指南》中的AWS 全域條件內容索引鍵。

如要查看 Lambda 條件索引鍵的清單，請參閱《服務授權參考》中的 Condition keys for AWS Lambda。若要了解您可以使用條件索引鍵的動作和資源，請參閱 定義的動作 AWS Lambda。

若要檢視 Lambda 身分型政策範例，請參閱的身分型政策範例 AWS Lambda。

Lambda 中的 ACL

支援 ACL：否

存取控制清單 (ACL) 可控制哪些主體 (帳戶成員、使用者或角色) 擁有存取某資源的許可。ACL 類似於資源型政策，但它們不使用 JSON 政策文件格式。

使用 Lambda 的 ABAC

支援 ABAC (政策中的標籤)：部分

屬性型存取控制 (ABAC) 是一種授權策略，根據稱為標籤的屬性定義許可權。您可以將標籤連接至 IAM 實體 AWS 和資源，然後設計 ABAC 政策，以便在委託人的標籤符合資源上的標籤時允許操作。

如需根據標籤控制存取，請使用 `aws:ResourceTag/`*`key-name`*、`aws:RequestTag/`*`key-name`* 或 `aws:TagKeys` 條件索引鍵，在政策的條件元素中，提供標籤資訊。

如果服務支援每個資源類型的全部三個條件金鑰，則對該服務而言，值為 Yes。如果服務僅支援某些資源類型的全部三個條件金鑰，則值為 Partial。

如需 ABAC 的詳細資訊，請參閱《IAM 使用者指南》中的使用 ABAC 授權定義許可。如要查看含有設定 ABAC 步驟的教學課程，請參閱《IAM 使用者指南》中的使用屬性型存取控制 (ABAC)。

如需標記 Lambda 資源的詳細資訊，請參閱使用 Lambda 中的屬性型存取控制。

將臨時憑證與 Lambda 搭配使用

支援臨時憑證：是

臨時登入資料提供對 AWS 資源的短期存取，並在您使用聯合或切換角色時自動建立。 AWS 建議您動態產生臨時登入資料，而不是使用長期存取金鑰。如需詳細資訊，請參閱《IAM 使用者指南》中的 IAM 中的臨時安全憑證與可與 IAM 搭配運作的AWS 服務。

轉送 Lambda 的存取工作階段

支援轉寄存取工作階段 (FAS)：否

轉送存取工作階段 (FAS) 使用呼叫 的委託人許可 AWS 服務，並結合 AWS 服務 請求向下游服務提出請求。如需提出 FAS 請求時的政策詳細資訊，請參閱轉發存取工作階段。

Lambda 的服務角色

支援服務角色：是

服務角色是服務擔任的 IAM 角色，可代您執行動作。IAM 管理員可以從 IAM 內建立、修改和刪除服務角色。如需詳細資訊，請參閱《IAM 使用者指南》中的建立角色以委派許可給 AWS 服務。

在 Lambda 中，服務角色稱為執行角色。

> **Warning**
>
> 變更執行角色的許可有可能會讓 Lambda 功能出現故障。

Lambda 的服務連結角色

支援服務連結角色：部分

服務連結角色是連結至 的一種服務角色 AWS 服務。服務可以擔任代表您執行動作的角色。服務連結角色會出現在您的 中 AWS 帳戶 ，並由服務擁有。IAM 管理員可以檢視，但不能編輯服務連結角色的許可。

Lambda 沒有服務連結角色，但 Lambda@Edge 則有。如需詳細資訊，請參閱《Amazon CloudFront 開發人員指南》中的 Lambda@Edge 的服務連結角色。

如需建立或管理服務連結角色的詳細資訊，請參閱可搭配 IAM 運作的AWS 服務。在資料表中尋找服務，其中包含服務連結角色欄中的 Yes。選擇是連結，以檢視該服務的服務連結角色文件。

的身分型政策範例 AWS Lambda

根據預設，使用者和角色不具備建立或修改 Lambda 資源的許可。若要授予使用者對其所需資源執行動作的許可，IAM 管理員可以建立 IAM 政策。

如需了解如何使用這些範例 JSON 政策文件建立 IAM 身分型政策，請參閱《IAM 使用者指南》中的建立 IAM 政策 (主控台)。

如需 Lambda 所定義之動作和資源類型的詳細資訊，包括每種資源類型的 ARN 格式，請參閱《服務授權參考》中的適用於 AWS Lambda的動作、資源和條件索引鍵。

主題

- 政策最佳實務
- 使用 Lambda 主控台
- 允許使用者檢視他們自己的許可

政策最佳實務

身分型政策會判斷您帳戶中的某個人員是否可以建立、存取或刪除 Lambda 資源。這些動作可能會讓您的 AWS 帳戶產生費用。當您建立或編輯身分型政策時，請遵循下列準則及建議事項：

- 開始使用 AWS 受管政策並邁向最低權限許可 – 若要開始將許可授予您的使用者和工作負載，請使用將許可授予許多常見使用案例的 AWS 受管政策。它們可在您的 中使用 AWS 帳戶。我們建議您定義特定於使用案例 AWS 的客戶受管政策，進一步減少許可。如需更多資訊，請參閱《IAM 使用者指南》中的 AWS 受管政策或任務職能的AWS 受管政策。
- 套用最低權限許可 – 設定 IAM 政策的許可時，請僅授予執行任務所需的許可。為實現此目的，您可以定義在特定條件下可以對特定資源採取的動作，這也稱為最低權限許可。如需使用 IAM 套用許可的更多相關資訊，請參閱《IAM 使用者指南》中的 IAM 中的政策和許可。
- 使用 IAM 政策中的條件進一步限制存取權 – 您可以將條件新增至政策，以限制動作和資源的存取。例如，您可以撰寫政策條件，指定必須使用 SSL 傳送所有請求。如果透過特定 ，例如 使用服務動作 AWS 服務，您也可以使用條件來授予存取 CloudFormation。如需詳細資訊，請參閱《IAM 使用者指南》中的 IAM JSON 政策元素：條件。

- 使用 IAM Access Analyzer 驗證 IAM 政策，確保許可安全且可正常運作 – IAM Access Analyzer 驗證新政策和現有政策，確保這些政策遵從 IAM 政策語言 (JSON) 和 IAM 最佳實務。IAM Access Analyzer 提供 100 多項政策檢查及切實可行的建議，可協助您撰寫安全且實用的政策。如需詳細資訊，請參閱《IAM 使用者指南》中的使用 IAM Access Analyzer 驗證政策。
- 需要多重要素驗證 (MFA) – 如果您的案例需要 IAM 使用者或 中的根使用者 AWS 帳戶，請開啟 MFA 以提高安全性。如需在呼叫 API 操作時請求 MFA，請將 MFA 條件新增至您的政策。如需詳細資訊，請參閱《IAM 使用者指南》中的透過 MFA 的安全 API 存取。

如需 IAM 中最佳實務的相關資訊，請參閱《IAM 使用者指南》中的 IAM 安全最佳實務。

使用 Lambda 主控台

若要存取 AWS Lambda 主控台，您必須擁有一組最低許可。這些許可必須允許您列出和檢視 中 Lambda 資源的詳細資訊 AWS 帳戶。如果您建立比最基本必要許可更嚴格的身分型政策，則對於具有該政策的實體 (使用者或角色) 而言，主控台就無法如預期運作。

對於僅呼叫 AWS CLI 或 AWS API 的使用者，您不需要允許最低主控台許可。反之，只需允許存取符合他們嘗試執行之 API 操作的動作就可以了。

如需針對函式開發授予最低存取權的範例政策，請參閱 授予使用者對 Lambda 函數的存取權。除了 Lambda API，Lambda 主控台會使用其他服務來顯示觸發組態，並讓您新增新的觸發。如果您的使用者搭配其他服務使用 Lambda，他們就必須也要存取這些服務。如需透過 Lambda 設定其他服務的詳細資訊，請參閱 使用來自其他服務的事件叫用 Lambda AWS。

允許使用者檢視他們自己的許可

此範例會示範如何建立政策，允許 IAM 使用者檢視附加到他們使用者身分的內嵌及受管政策。此政策包含在主控台或使用 或 AWS CLI AWS API 以程式設計方式完成此動作的許可。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
```

```
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS 的 受管政策 AWS Lambda

AWS 受管政策是由 AWS AWS 受管政策建立和管理的獨立政策旨在為許多常用案例提供許可，以便您可以開始將許可指派給使用者、群組和角色。

請記住， AWS 受管政策可能不會授予特定使用案例的最低權限許可，因為這些許可可供所有 AWS 客戶使用。我們建議您定義特定於使用案例的客戶管理政策，以便進一步減少許可。

您無法變更 AWS 受管政策中定義的許可。如果 AWS 更新受 AWS 管政策中定義的許可，則更新會影響政策連接的所有委託人身分 （使用者、群組和角色）。 AWS 服務 當新的 啟動或新的 API 操作可用於現有服務時， AWS 最有可能更新 AWS 受管政策。

如需詳細資訊，請參閱 IAM 使用者指南中的 AWS 受管政策。

主題

- AWS 受管政策：AWSLambda_FullAccess
- AWS 受管政策：AWSLambda_ReadOnlyAccess
- AWS 受管政策：AWSLambdaBasicExecutionRole

- AWS 受管政策：AWSLambdaBasicDurableExecutionRolePolicy
- AWS 受管政策：AWSLambdaDynamoDBExecutionRole
- AWS 受管政策：AWSLambdaENIManagementAccess
- AWS 受管政策：AWSLambdaInvocation-DynamoDB
- AWS 受管政策：AWSLambdaKinesisExecutionRole
- AWS 受管政策：AWSLambdaMSKExecutionRole
- AWS 受管政策：AWSLambdaRole
- AWS 受管政策：AWSLambdaSQSQueueExecutionRole
- AWS 受管政策：AWSLambdaVPCAccessExecutionRole
- AWS 受管政策：AWSLambdaManagedEC2ResourceOperator
- AWS 受管政策：AWSLambdaServiceRolePolicy
- AWS 受管政策的 Lambda 更新

AWS 受管政策：AWSLambda_FullAccess

此政策也會授予完整存取權給 Lambda 動作。它還將許可授予用於開發和維護 Lambda 資源的其他 AWS 服務。

您可以將 `AWSLambda_FullAccess` 政策連接至使用者、群組與角色。

許可詳細資訊

此政策包含以下許可：

- `lambda`：允許委託人完整存取 Lambda。
- `cloudformation` – 允許主體描述 AWS CloudFormation 堆疊並列出這些堆疊中的資源。
- `cloudwatch`：允許委託人列出 Amazon CloudWatch 指標並取得指標資料。
- `ec2`：允許委託人描述安全群組、子網路和 VPC。
- `iam`：允許委託人取得政策、政策版本、角色、角色政策、連接角色政策以及角色清單。此政策也允許委託人將角色傳遞給 Lambda。將執行角色指派給函數時，便會使用 `PassRole` 許可。建立服務連結角色時，會使用 `CreateServiceLinkedRole`許可。
- `kms` – 允許主體列出別名並描述磁碟區加密的金鑰。
- `logs`：允許主體描述日誌串流、取得日誌事件、篩選日誌事件，以及開始和停止 Live Tail 工作階段。

- states – 允許主體描述和列出 AWS Step Functions 狀態機器。
- tag：允許委託人根據其標籤取得資源。
- xray – 允許主體取得 AWS X-Ray 追蹤摘要，並擷取 ID 指定的追蹤清單。

如需有關此政策的詳細資訊 (包括 JSON 政策文件和政策版本)，請參閱《AWS 受管政策參考指南》中的 AWSLambda_FullAccess。

AWS 受管政策：AWSLambda_ReadOnlyAccess

此政策授予對 Lambda 資源和用於開發和維護 Lambda 資源之其他服務 AWS 的唯讀存取權。

您可以將 AWSLambda_ReadOnlyAccess 政策連接至使用者、群組與角色。

許可詳細資訊

此政策包含以下許可：

- lambda：允許委託人取得和列出所有資源。
- cloudformation – 允許主體描述和列出 AWS CloudFormation 堆疊，並列出這些堆疊中的資源。
- cloudwatch：允許委託人列出 Amazon CloudWatch 指標並取得指標資料。
- ec2：允許委託人描述安全群組、子網路和 VPC。
- iam：允許委託人取得政策、政策版本、角色、角色政策、連接角色政策以及角色清單。
- kms：允許委託人列出別名。
- logs：允許主體描述日誌串流、取得日誌事件、篩選日誌事件，以及開始和停止 Live Tail 工作階段。
- states – 允許主體描述和列出 AWS Step Functions 狀態機器。
- tag：允許委託人根據其標籤取得資源。
- xray – 允許主體取得 AWS X-Ray 追蹤摘要，並擷取 ID 指定的追蹤清單。

如需有關此政策的詳細資訊 (包括 JSON 政策文件和政策版本)，請參閱《AWS 受管政策參考指南》中的 AWSLambda_ReadOnlyAccess。

AWS 受管政策：AWSLambdaBasicExecutionRole

此政策授予將日誌上傳至 CloudWatch Logs 的許可。

您可以將 AWSLambdaBasicExecutionRole 政策連接至使用者、群組與角色。

如需有關此政策的詳細資訊 (包括 JSON 政策文件和政策版本)，請參閱《AWS 受管政策參考指南》中的 AWSLambdaBasicExecutionRole。

AWS 受管政策：AWSLambdaBasicDurableExecutionRolePolicy

此政策提供 CloudWatch Logs 的寫入許可，以及 Lambda 耐用函數使用的耐久執行 APIs讀取/寫入許可。此政策提供 Lambda 耐用函數所需的基本許可，其使用耐用APIs 來保留進度，並維持函數叫用之間的狀態。

您可以將 AWSLambdaBasicDurableExecutionRolePolicy 政策連接至使用者、群組與角色。

許可詳細資訊

此政策包含以下許可：

- logs – 允許主體建立日誌群組和日誌串流，並將日誌事件寫入 CloudWatch Logs。
- lambda – 允許主體檢查持久性執行狀態，並擷取 Lambda 持久性函數的持久性執行狀態。

若要檢視政策的詳細資訊，包括最新版本的 JSON 政策文件，請參閱《 AWS 受管政策參考指南》中的 AWSLambdaBasicDurableExecutionRolePolicy。

AWS 受管政策：AWSLambdaDynamoDBExecutionRole

此政策授予從 Amazon DynamoDB 串流讀取記錄和寫入 CloudWatch Logs 的許可。

您可以將 AWSLambdaDynamoDBExecutionRole 政策連接至使用者、群組與角色。

如需有關此政策的詳細資訊 (包括 JSON 政策文件和政策版本)，請參閱《AWS 受管政策參考指南》中的 AWSLambdaDynamoDBExecutionRole。

AWS 受管政策：AWSLambdaENIManagementAccess

此政策授予建立、描述和刪除已啟用 VPC 之 Lambda 函數所使用之彈性網路界面的許可。

您可以將 AWSLambdaENIManagementAccess 政策連接至使用者、群組與角色。

如需有關此政策的詳細資訊 (包括 JSON 政策文件和政策版本)，請參閱《AWS 受管政策參考指南》中的 AWSLambdaENIManagementAccess。

AWS 受管政策：AWSLambdaInvocation-DynamoDB

此政策授予 Amazon DynamoDB Streams 的讀取存取權。

您可以將 AWSLambdaInvocation-DynamoDB 政策連接至使用者、群組與角色。

如需有關此政策的詳細資訊 (包括 JSON 政策文件和政策版本)，請參閱《AWS 受管政策參考指南》中的 AWSLambdaInvocation-DynamoDB。

AWS 受管政策：AWSLambdaKinesisExecutionRole

此政策授予從 Amazon Kinesis 資料串流讀取事件和寫入 CloudWatch Logs 的許可。

您可以將 AWSLambdaKinesisExecutionRole 政策連接至使用者、群組與角色。

如需有關此政策的詳細資訊 (包括 JSON 政策文件和政策版本)，請參閱《AWS 受管政策參考指南》中的 AWSLambdaKinesisExecutionRole。

AWS 受管政策：AWSLambdaMSKExecutionRole

此政策授予從 Amazon Managed Streaming for Apache Kafka 叢集中讀取和存取記錄、管理彈性網絡界面，以及寫入 CloudWatch Logs 的許可。

您可以將 AWSLambdaMSKExecutionRole 政策連接至使用者、群組與角色。

如需有關此政策的詳細資訊 (包括 JSON 政策文件和政策版本)，請參閱《AWS 受管政策參考指南》中的 AWSLambdaMSKExecutionRole。

AWS 受管政策：AWSLambdaRole

此政策授予調用 Lambda 函數的許可。

您可以將 AWSLambdaRole 政策連接至使用者、群組與角色。

如需有關此政策的詳細資訊 (包括 JSON 政策文件和政策版本)，請參閱《AWS 受管政策參考指南》中的 AWSLambdaRole。

AWS 受管政策：AWSLambdaSQSQueueExecutionRole

此政策授予從 Amazon Simple Queue Service 佇列中讀取和刪除訊息以及寫入 CloudWatch Logs 的許可。

您可以將 AWSLambdaSQSQueueExecutionRole 政策連接至使用者、群組與角色。

如需有關此政策的詳細資訊 (包括 JSON 政策文件和政策版本)，請參閱《AWS 受管政策參考指南》中的 AWSLambdaSQSQueueExecutionRole。

AWS 受管政策：AWSLambdaVPCAccessExecutionRole

此政策授予在 Amazon Virtual Private Cloud 中管理彈性網路界面和寫入 CloudWatch Logs 的許可。

您可以將 `AWSLambdaVPCAccessExecutionRole` 政策連接至使用者、群組與角色。

如需有關此政策的詳細資訊 (包括 JSON 政策文件和政策版本)，請參閱《AWS 受管政策參考指南》中的 AWSLambdaVPCAccessExecutionRole。

AWS 受管政策：AWSLambdaManagedEC2ResourceOperator

此政策可為 Lambda 容量提供者啟用自動化 Amazon Elastic Compute Cloud 執行個體管理。它將許可授予 Lambda 擴展器服務，以代表您執行執行個體生命週期操作。

您可以將 `AWSLambdaManagedEC2ResourceOperator` 政策連接至使用者、群組與角色。

許可詳細資訊

此政策包含以下許可：

- `ec2:RunInstances` – 允許 Lambda 以 ec2：ManagedResourceOperator 等於 scaler.lambda.amazonaws.com 的條件啟動新的 Amazon EC2 執行個體，並將 AMI 用量限制為僅限 Amazon 擁有的映像。
- `ec2:DescribeInstances` 和 `ec2:DescribeInstanceStatus` – 允許 Lambda 監控執行個體狀態並擷取執行個體資訊。
- `ec2:CreateTags` – 允許 Lambda 標記 Amazon EC2 資源以進行管理和識別。
- `ec2:DescribeAvailabilityZones` – 允許 Lambda 檢視執行個體置放決策的可用區域。
- `ec2:DescribeCapacityReservations` – 允許 Lambda 檢查容量保留，以獲得最佳執行個體放置。
- `ec2:DescribeInstanceTypes` 和 `ec2:DescribeInstanceTypeOfferings` – 允許 Lambda 檢閱可用的執行個體類型及其方案。
- `ec2:DescribeSubnets` – 允許 Lambda 檢查子網路組態以進行網路規劃。
- `ec2:DescribeSecurityGroups` – 允許 Lambda 擷取網路介面組態的安全群組資訊。
- `ec2:CreateNetworkInterface` – 允許 Lambda 建立網路介面和管理子網路和安全群組關聯。
- `ec2:AttachNetworkInterface` – 允許 Lambda 將網路介面連接到條件`ec2:ManagedResourceOperator`等於 scaler.lambda.amazonaws.com 的 Amazon EC2 執行個體。

如需此政策的詳細資訊，包括 JSON 政策文件和政策版本，請參閱《 AWS 受管政策參考指南》中的 AWSLambdaManagedEC2ResourceOperator。

AWS 受管政策：AWSLambdaServiceRolePolicy

此政策會連接到名為 AWSServiceRoleForLambda 的服務連結角色，以允許 Lambda 終止作為 Lambda 容量提供者一部分管理的執行個體。

許可詳細資訊

此政策包含以下許可：

- `ec2:TerminateInstances` – 允許 Lambda 終止條件為 ec2：ManagedResourceOperator 等於 scaler.lambda.amazonaws.com 的 EC2 執行個體。
- `ec2:DescribeInstanceStatus` 和 `ec2:DescribeInstances` – 允許 Lambda 描述 EC2 執行個體。

如需此政策的詳細資訊，請參閱使用 Lambda 的服務連結角色。

AWS 受管政策的 Lambda 更新

變更	描述	Date
AWSLambdaManagedEC2ResourceOperator – 新政策	Lambda 新增了新的受管政策，為 Lambda 容量提供者啟用自動化 Amazon EC2 執行個體管理，允許擴展器服務執行執行個體生命週期操作。	2025 年 11 月 30 日
AWSLambdaServiceRolePolicy – 新政策	Lambda 新增了服務連結角色的新受管政策，以允許 Lambda 終止做為 Lambda 容量提供者一部分管理的執行個體。	2025 年 11 月 30 日
AWSLambda_FullAccess – 變更	Lambda 已更新`AWSLambda_FullAccess` 政策，以允許 `kms:DescribeKey` 和	2025 年 11 月 30 日

變更	描述	Date
	`iam:CreateServiceLinkedRole` 動作。	
AWSLambdaBasicDurableExecutionRolePolicy – 新受管政策	Lambda 發佈了新的受管政策`AWSLambdaBasicDurableExecutionRolePolicy`，為 CloudWatch Logs 提供寫入許可，並為 Lambda 耐用函數使用的持久執行 APIs 提供讀取/寫入許可。	2025 年 12 月 1 日
AWSLambda_ReadOnlyAccess 和 AWSLambda_FullAccess：變更	Lambda 更新了 `AWSLambda_ReadOnlyAccess` 和 `AWSLambda_FullAccess` 政策，允許 `logs:StartLiveTail` 和 `logs:StopLiveTail` 動作。	2025 年 3 月 17 日
AWSLambdaVPCAccessExecutionRole：變更	Lambda 更新了 `AWSLambdaVPCAccessExecutionRole` 政策以允許動作 `ec2:DescribeSubnets` 。	2024 年 1 月 5 日
AWSLambda_ReadOnlyAccess：變更	Lambda 已更新`AWSLambda_ReadOnlyAccess` 政策，允許主體列出 CloudFormation 堆疊。	2023 年 7 月 27 日
AWS Lambda 開始追蹤變更	AWS Lambda 已開始追蹤其 AWS 受管政策的變更。	2023 年 7 月 27 日

對 AWS Lambda 身分和存取進行故障診斷

請使用以下資訊來協助您診斷和修正使用 Lambda 和 IAM 時可能遇到的常見問題。

主題

- 我未獲授權，不得在 Lambda 中執行動作
- 我未獲得執行 iam:PassRole 的授權
- 我想要允許 以外的人員 AWS 帳戶 存取我的 Lambda 資源

我未獲授權，不得在 Lambda 中執行動作

如果您收到錯誤，告知您未獲授權執行動作，您的政策必須更新，允許您執行動作。

下列範例錯誤會在mateojackson IAM 使用者嘗試使用主控台檢視一個虛構 *my-example-widget* 資源的詳細資訊，但卻無虛構 lambda:*GetWidget* 許可時發生。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 lambda:GetWidget on resource: my-example-widget
```

在此情況下，必須更新 mateojackson 使用者的政策，允許使用 lambda:*GetWidget* 動作存取 *my-example-widget* 資源。

如果您需要協助，請聯絡您的 AWS 管理員。您的管理員提供您的簽署憑證。

我未獲得執行 iam:PassRole 的授權

如果您收到錯誤，告知您無權執行 iam:PassRole 動作，則必須更新您的政策，以允許您將角色傳遞至 Lambda。

有些 AWS 服務 可讓您將現有角色傳遞給該服務，而不是建立新的服務角色或服務連結角色。如需執行此作業，您必須擁有將角色傳遞至該服務的許可。

當名為 marymajor 的 IAM 使用者嘗試使用主控台在 Lambda 中執行動作時，發生下列範例錯誤。但是，動作請求服務具備服務角色授予的許可。Mary 沒有將角色傳遞給服務的許可。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

在這種情況下，Mary 的政策必須更新，允許她執行 iam:PassRole 動作。

如果您需要協助，請聯絡您的 AWS 管理員。您的管理員提供您的簽署憑證。

我想要允許 以外的人員 AWS 帳戶 存取我的 Lambda 資源

您可以建立一個角色，讓其他帳戶中的使用者或您組織外部的人員存取您的資源。您可以指定要允許哪些信任物件取得該角色。針對支援基於資源的政策或存取控制清單 (ACL) 的服務，您可以使用那些政策來授予人員存取您的資源的許可。

如需進一步了解，請參閱以下內容：

- 若要了解 Lambda 是否支援這些功能，請參閱 AWS Lambda 如何使用 IAM。
- 若要了解如何 AWS 帳戶 在您擁有的 資源之間提供存取權，請參閱《IAM 使用者指南》中的在您擁有 AWS 帳戶 的另一個 IAM 使用者中提供存取權。
- 若要了解如何將資源的存取權提供給第三方 AWS 帳戶，請參閱《IAM 使用者指南》中的將存取權提供給第三方 AWS 帳戶 擁有。
- 如需了解如何透過聯合身分提供存取權，請參閱《IAM 使用者指南》中的將存取權提供給在外部進行身分驗證的使用者 (聯合身分)。
- 如需了解使用角色和資源型政策進行跨帳戶存取之間的差異，請參閱《IAM 使用者指南》中的 IAM 中的跨帳戶資源存取。

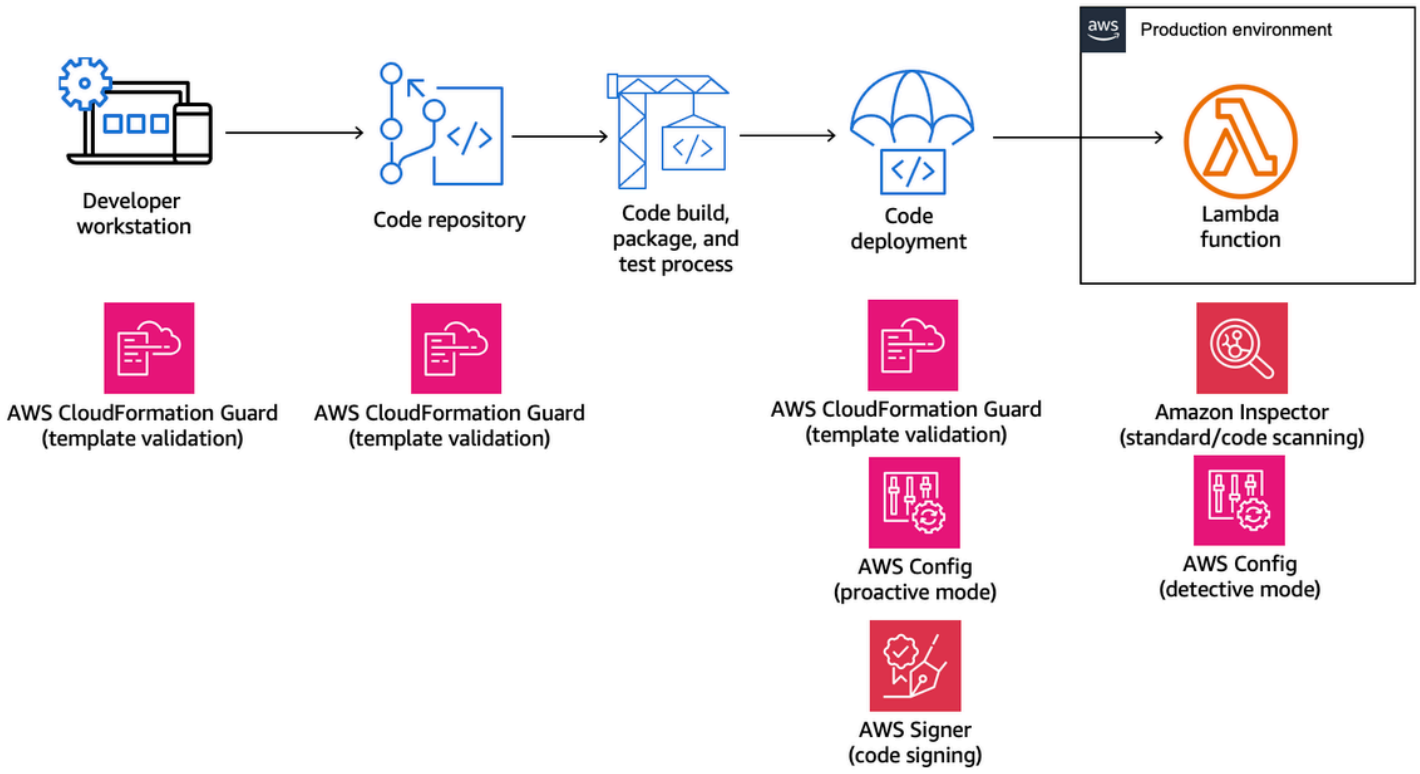
建立 Lambda 函數和層的控管策略

若要建置和部署無伺服器、雲端原生應用程式，您必須透過適當的控管和防護機制來確保敏捷性和加速上市。您要設定商業級別優先順序，可以強調敏捷性做為首要任務，或者透過控管、防護機制和控制項來強調風險規避。實際上，您不能採用「非此即彼」的策略，而要選擇能在軟體開發週期中平衡敏捷性和防護機制的「兼顧」策略。無論這些要求位於公司生命週期的哪個階段，控管功能都有可能成為您的流程和工具鏈當中的一項實作要求。

以下是一些組織可能為 Lambda 實作的控管控制項範例：

- Lambda 函數不可公開存取。
- Lambda 函數必須附加到 VPC。
- Lambda 函數不應使用已棄用的執行期。
- Lambda 函數必須有一組必要標籤進行標記。
- Lambda 層不可在組織外部存取。
- 附加安全群組的 Lambda 函數必須在函數和安全群組之間有配對的標籤。
- 具有附加層的 Lambda 函數必須使用核准的版本
- Lambda 環境變數必須使用客戶自管金鑰進行靜態加密。

下圖是深入控管策略的範例，此類策略會在軟體開發和部署過程當中實作控制項和政策：



下列主題說明如何針對新創公司和企業在組織中實作開發和部署 Lambda 函數的控制項。您的組織可能有現成的工具。下列主題採用模組化方法來實作這些控制項，可讓您挑選實際需要的元件。

主題

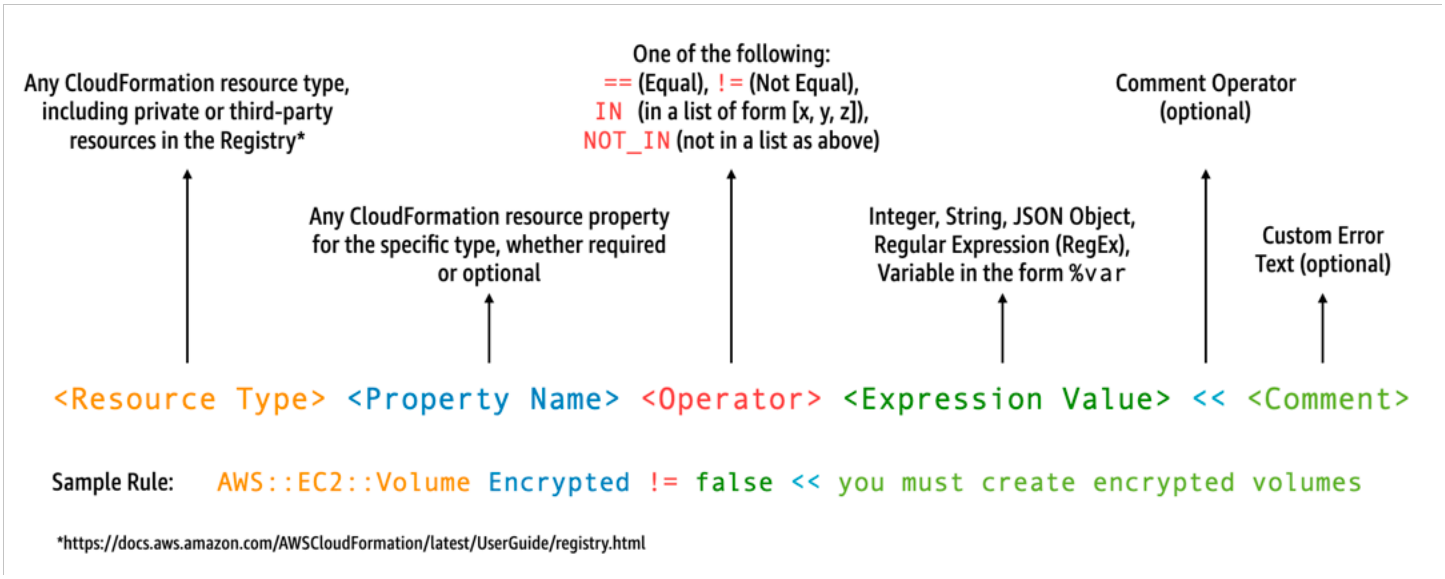
- 使用 AWS CloudFormation Guard 的 Lambda 主動式控制項
- 使用 實作 Lambda 的預防性控制 AWS Config
- 使用 偵測不合規的 Lambda 部署和組態 AWS Config
- 使用 AWS Signer 的 Lambda 程式碼簽署
- 使用 Amazon Inspector 自動化 Lambda 的安全評估
- 實作 Lambda 安全性與合規性的可觀測性

使用 AWS CloudFormation Guard 的 Lambda 主動式控制項

AWS CloudFormation Guard 是一種開放原始碼的通用型政策即程式碼評估工具。透過比照政策規則驗證基礎設施即程式碼 (IaC) 範本和服務組合，它可被用於實現預防性控管與合規。這些規則還可以依據您的團隊或組織要求進行自訂。對於 Lambda 函數，Guard 規則可透過在建立或更新 Lambda 函數時定義所需的必要屬性設定，控制資源建立和組態更新。

合規管理員會定義部署和更新 Lambda 函數所需的控制項和控管政策清單。平台管理員在 CI/CD 管道中實作控制項，做為程式碼儲存庫的預遞交驗證 Webhook，並為開發人員提供用於在本機工作站驗證範本和程式碼的命令列工具。開發人員使用命令列工具編寫程式碼並驗證範本，然後將程式碼遞交到儲存庫。儲存庫隨後會在將這些程式碼部署到 AWS 環境前透過 CI/CD 管道自動對其進行驗證。

Guard 讓您可以使用特定領域的語言，如下編寫您的規則並實作您的控制項。



例如，假設您希望確定開發人員僅選擇最新的執行期。您可以指定兩種不同的政策，一種用來識別已棄用的執行期，另一種則用來確定即將棄用的執行期。為此，您可能要編寫以下 `etc/rules.guard` 檔案：

```
let lambda_functions = Resources.*[
    Type == "AWS::Lambda::Function"
]

rule lambda_already_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
```

```
                Runtime !in ["dotnetcore3.1", "nodejs12.x", "python3.6", "python2.7",
 "dotnet5.0", "dotnetcore2.1", "ruby2.5", "nodejs10.x", "nodejs8.10", "nodejs4.3",
 "nodejs6.10", "dotnetcore1.0", "dotnetcore2.0", "nodejs4.3-edge", "nodejs"] <<Lambda
 function is using a deprecated runtime.>>
            }
        }
    }
}

rule lambda_soon_to_be_deprecated_runtime when %lambda_functions !empty {
    %lambda_functions {
        Properties {
            when Runtime exists {
                Runtime !in ["nodejs16.x", "nodejs14.x", "python3.7", "java8",
 "dotnet7", "go1.x", "ruby2.7", "provided"] <<Lambda function is using a runtime that
 is targeted for deprecation.>>
            }
        }
    }
}
```

現在，假設您要編寫定義 Lambda 函數的以下 `iac/lambda.yaml` CloudFormation 範本：

```
  Fn:
    Type: AWS::Lambda::Function
    Properties:
      Runtime: python3.7
      CodeUri: src
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      Layers:
        - arn:aws:lambda:us-east-1:111122223333:layer:LambdaInsightsExtension:35
```

在安裝 Guard 公用程式後，驗證您的範本：

```
cfn-guard validate --rules etc/rules.guard --data iac/lambda.yaml
```

輸出如下：

```
lambda.yaml Status = FAIL
FAILED rules
rules.guard/lambda_soon_to_be_deprecated_runtime
```

```
---
Evaluating data lambda.yaml against rules rules.guard
Number of non-compliant resources 1
Resource = Fn {
  Type      = AWS::Lambda::Function
  Rule = lambda_soon_to_be_deprecated_runtime {
    ALL {
      Check =  Runtime not IN
 ["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]
 {
        ComparisonError {
          Message          = Lambda function is using a runtime that is targeted for
 deprecation.
          Error            = Check was not compliant as property [/Resources/
Fn/Properties/Runtime[L:88,C:15]] was not present in [(resolved, Path=[L:0,C:0]
 Value=["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"])]
        }
          PropertyPath    = /Resources/Fn/Properties/Runtime[L:88,C:15]
          Operator        = NOT IN
          Value           = "python3.7"
          ComparedWith    =
 [["nodejs16.x","nodejs14.x","python3.7","java8","dotnet7","go1.x","ruby2.7","provided"]]
          Code:
               86.  Fn:
               87.    Type: AWS::Lambda::Function
               88.    Properties:
               89.      Runtime: python3.7
               90.      CodeUri: src
               91.      Handler: fn.handler

      }
    }
  }
}
```

Guard 可讓您的開發人員透過本機開發工作站即可了解，他們需要更新範本以便使用組織允許的執行期。這發生在遞交至程式碼儲存庫之前，隨後會使 CI/CD 管道中的檢查失敗。因此，您的開發人員將取得此回饋，以便了解如何開發合乎規範的範本，並且將他們的時間轉移到編寫可創造商業價值的程式碼。此控制可套用到本機的開發人員工作站，在預遞交驗證 Webhook 和/或在部署前的 CI/CD 管道中套用。

警告

如果您使用 AWS Serverless Application Model (AWS SAM) 範本定義 Lambda 函數，注意您需要如下更新 Guard 規則以便搜尋 `AWS::Serverless::Function` 資源類型。

```
let lambda_functions = Resources.*[
    Type == "AWS::Serverless::Function"
]
```

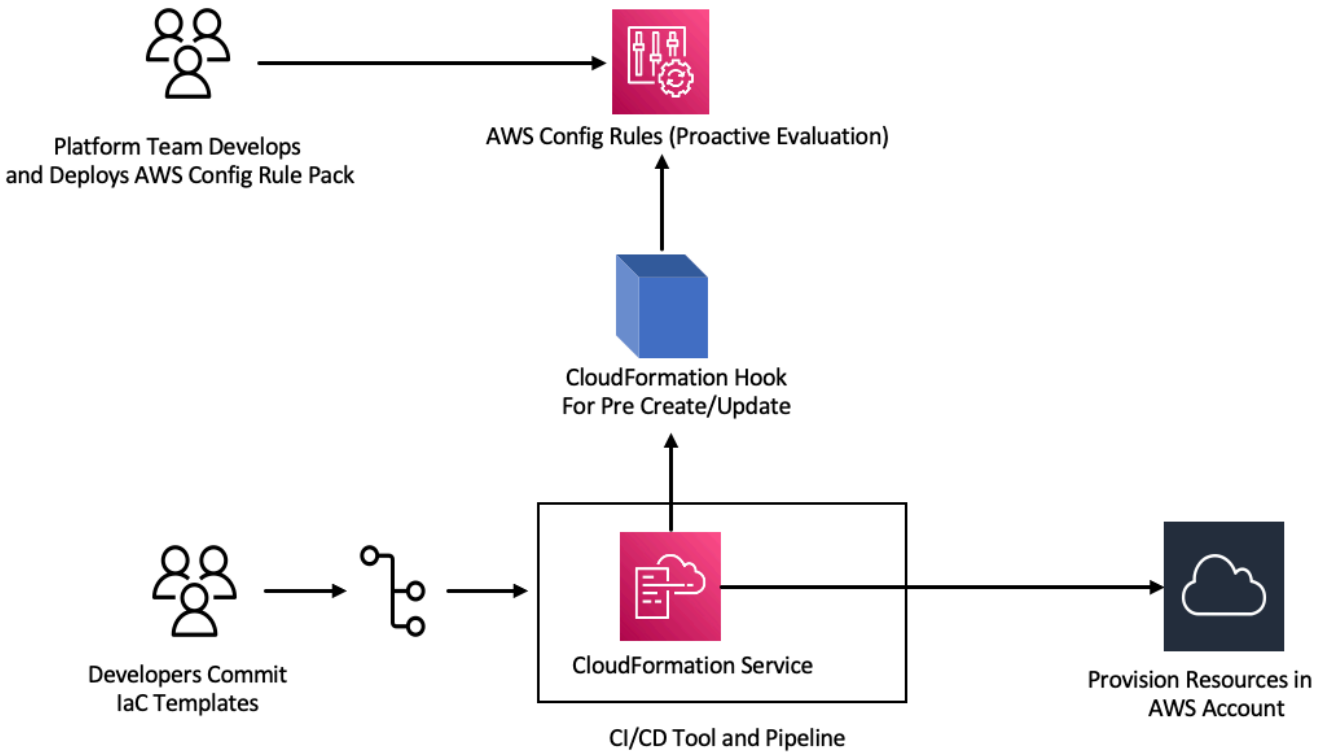
Guard 還預期屬性會包含在資源定義當中。同時，AWS SAM 範本允許在單獨的全域區段指定屬性。「全域」區段裡定義的屬性不使用您的 Guard 規則進行驗證。

如 Guard 故障診斷文件中所述，應注意 Guard 不支援簡寫形式的內部函數 (例如 `!GetAtt` 或 `!Sub`)，而要求使用展開形式：`Fn::GetAtt` 和 `Fn::Sub`。(先前的範例不會評估 Role 屬性，因此為了簡便起見而使用簡寫形式的內部函數。)

使用 實作 Lambda 的預防性控制 AWS Config

儘早在開發過程中確保無伺服器應用程式的合規性至關重要。在本主題中，我們將說明如何使用 AWS Config 實作預防性控制項。這可讓您在開發過程的早期實作合規檢查，並且使您可以在 CI/CD 管道中實作相同的控制項。這也會標準化 規則集中受管儲存庫中的控制項，讓您可以在 AWS 帳戶中一致地套用控制項。

例如，假設您的合規管理員定義了一項要求，以確保所有 Lambda 函數都包含 AWS X-Ray 追蹤。使用 AWS Config的主動模式，您可以在部署之前對 Lambda 函數資源執行合規檢查，藉由提供更快速的基礎設施意見回饋作為程式碼範本，降低部署設定不當的 Lambda 函數並節省開發人員時間的風險。以下是使用 進行預防性控制的流程視覺化 AWS Config：



考慮所有 Lambda 函數都必須啟用追蹤功能的要求。為了回應，平台團隊會識別在所有帳戶中主動執行特定 AWS Config 規則的需求。此規則會將任何未設定 X-Ray 追蹤組態的 Lambda 函數標記為不合規資源。團隊開發規則，將其封裝在一致性套件中，並將一致性套件部署到所有 AWS 帳戶，以確保組織中的所有帳戶統一套用這些控制項。您可以採用 AWS CloudFormation Guard 2.x.x 語法編寫該規則，它將採用以下形式：

```
rule name when condition { assertion }
```

以下是檢查以確保 Lambda 函數已啟用追蹤功能的範例 Guard 規則：

```
rule lambda_tracing_check {
  when configuration.tracingConfig exists {
      configuration.tracingConfig.mode == "Active"
  }
}
```

平台團隊透過強制每個 AWS CloudFormation 部署叫用預先建立/更新掛鉤，採取進一步的動作。他們會全權負責開發此勾點和設定管道、強化合規規則的集中控制，並確保在所有部署中一致地套用這些規則。若要開發、封裝和註冊掛鉤，請參閱 CloudFormation 命令列界面 (CFN-CLI) 文件中的開發 AWS CloudFormation 掛鉤。您可以使用 CloudFormation CLI 來建立勾點專案：

```
cfn init
```

此命令會要求您提供有關勾點專案的一些基本資訊，並在其中建立包含下列檔案的專案：

```
README.md
<hook-name>.json
rpdk.log
src/handler.py
template.yml
hook-role.yaml
```

做為勾點開發人員，您需要在 <hook-name>.json 組態檔案中新增所需的目標資源類型。在下列組態中，勾點被設為在使用 CloudFormation 建立任何 Lambda 函數之前執行。您也可以為 preUpdate 和 preDelete 動作新增類似的處理常式。

```
    "handlers": {
        "preCreate": {
            "targetNames": [
                "AWS::Lambda::Function"
            ],
            "permissions": []
        }
    }
```

您也需要確保 CloudFormation 勾點具有呼叫 AWS Config APIs的適當許可。您可以透過更新名為 hook-role.yaml 的角色定義檔案來執行該動作。角色定義檔案預設具有下列信任政策，以允許 CloudFormation 擔任相關角色。

```
      AssumeRolePolicyDocument:
        Version: '2012-10-17'
        Statement:
          - Effect: Allow
            Principal:
              Service:
                - hooks.cloudformation.amazonaws.com
                - resources.cloudformation.amazonaws.com
```

若要允許此勾點呼叫組態 API，您必須將下列許可新增至政策陳述式。然後，您可以使用 cfn submit 命令提交勾點專案，CloudFormation 會在其中為您建立具有所需許可的角色。

```
      Policies:
        - PolicyName: HookTypePolicy
          PolicyDocument:
            Version: '2012-10-17'
            Statement:
              - Effect: Allow
                Action:
                  - "config:Describe*"
                  - "config:Get*"
                  - "config:List*"
                  - "config:SelectResourceConfig"
                Resource: "*
```

接下來，您需要在 src/handler.py 檔案中編寫 Lambda 函數。在此檔案中，您將在啟動該專案時發現已建立的名為 preCreate、preUpdate 和 preDelete 的方法。您旨在撰寫一個常用、可重複使用的函數，使用 在 AWS Config 主動模式下呼叫 StartResourceEvaluation API 適用於 Python (Boto3) 的 AWS SDK。此 API 呼叫會將資源屬性做為輸入，並且比照規則定義評估資源。

```
def validate_lambda_tracing_config(resource_type, function_properties:
 MutableMapping[str, Any]) -> ProgressEvent:
  LOG.info("Fetching proactive data")
  config_client = boto3.client('config')
  resource_specs = {
      'ResourceId': 'MyFunction',
      'ResourceType': resource_type,
```

```
      'ResourceConfiguration': json.dumps(function_properties),
      'ResourceConfigurationSchemaType': 'CFN_RESOURCE_SCHEMA'
  }
  LOG.info("Resource Specifications:", resource_specs)
  eval_response = config_client.start_resource_evaluation(EvaluationMode='PROACTIVE',
 ResourceDetails=resource_specs, EvaluationTimeout=60)
  ResourceEvaluationId = eval_response.ResourceEvaluationId
  compliance_response =
 config_client.get_compliance_details_by_resource(ResourceEvaluationId=ResourceEvaluationId)
  LOG.info("Compliance Verification:",
 compliance_response.EvaluationResults[0].ComplianceType)
  if "NON_COMPLIANT" == compliance_response.EvaluationResults[0].ComplianceType:
      return ProgressEvent(status=OperationStatus.FAILED, message="Lambda function
 found with no tracing enabled : FAILED", errorCode=HandlerErrorCode.NonCompliant)
  else:
      return ProgressEvent(status=OperationStatus.SUCCESS, message="Lambda function
 found with tracing enabled : PASS.")
```

現在，您可以從處理常式為預先建立的勾點呼叫常用的函數。以下是處理常式的範例：

```
@hook.handler(HookInvocationPoint.CREATE_PRE_PROVISION)
def pre_create_handler(
        session: Optional[SessionProxy],
        request: HookHandlerRequest,
        callback_context: MutableMapping[str, Any],
        type_configuration: TypeConfigurationModel
) -> ProgressEvent:
    LOG.info("Starting execution of the hook")
    target_name = request.hookContext.targetName
    LOG.info("Target Name:", target_name)
    if "AWS::Lambda::Function" == target_name:
        return validate_lambda_tracing_config(target_name,
            request.hookContext.targetModel.get("resourceProperties")
        )
    else:
        raise exceptions.InvalidRequest(f"Unknown target type: {target_name}")
```

在此步驟之後，您可以註冊勾點並將其設定為接聽所有 AWS Lambda 函數建立事件。

開發人員會使用 Lambda 為無伺服器微服務準備基礎設施即程式碼 (IaC) 範本。此準備工作包括遵守內部標準，然後進行本機測試並將範本遞交至儲存庫。以下是範例 IaC 範本：

```
  MyLambdaFunction:
```

```
  Type: 'AWS::Lambda::Function'
  Properties:
    Handler: index.handler
    Role: !GetAtt LambdaExecutionRole.Arn
    FunctionName: MyLambdaFunction
    Code:
      ZipFile: |
        import json

        def handler(event, context):
            return {
                'statusCode': 200,
                'body': json.dumps('Hello World!')
            }
    Runtime: python3.14
    TracingConfig:
        Mode: PassThrough
    MemorySize: 256
    Timeout: 10
```

做為 CI/CD 程序的一部分，當部署 CloudFormation 範本時，CloudFormation 服務會在佈建 `AWS::Lambda::Function` 資源類型前直接調用預先建立/更新勾點。勾點利用以主動模式執行的 AWS Config 規則來驗證 Lambda 函數組態是否包含必要的追蹤組態。來自勾點的回應將決定下一個步驟。如果合規，勾點將發出成功訊號，CloudFormation 會繼續佈建資源。若不合規，CloudFormation 堆疊部署將會失敗，管道立即停止，而系統會記錄詳細資訊以供後續審核。合規通知將傳送至相關的利害關係人。

您可以在 CloudFormation 主控台中找到勾點成功/失敗的資訊：

Stack info | Events | Resources | Outputs | Parameters | Template | Change sets

Events (19)

Search events

Timestamp	Logical ID	Status	Status reason	Hook invocations
2023-08-29 23:50:23 UTC-0500	HookTestStack	ROLLBACK_COMPLETE	-	-
2023-08-29 23:50:22 UTC-0500	LambdaExecutionRole	DELETE_COMPLETE	-	-
2023-08-29 23:50:21 UTC-0500	MyApi	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	LambdaExecutionRole	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:20 UTC-0500	MyLambdaFunction	DELETE_COMPLETE	-	-
2023-08-29 23:50:20 UTC-0500	MyApi	DELETE_IN_PROGRESS	-	-
2023-08-29 23:50:18 UTC-0500	HookTestStack	ROLLBACK_IN_PROGRESS	The following resource(s) failed to create: [MyLambdaFunction]. Rollback requested by user.	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_FAILED	The following hook(s) failed: [AWSSamples::LambdaTracingCheck::Hook]	-
2023-08-29 23:50:17 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:16 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	AWSSamples::LambdaTracingCheck::Hook
2023-08-29 23:50:15 UTC-0500	MyLambdaFunction	CREATE_IN_PROGRESS	-	-
2023-08-29 23:50:14 UTC-0500	LambdaExecutionRole	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_COMPLETE	-	-
2023-08-29 23:49:59 UTC-0500	MyApi	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	Resource creation Initiated	-
2023-08-29 23:49:58 UTC-0500	LambdaExecutionRole	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:58 UTC-0500	MyApi	CREATE_IN_PROGRESS	-	-
2023-08-29 23:49:55 UTC-0500	HookTestStack	CREATE_IN_PROGRESS	User Initiated	-
2023-08-29 23:49:50 UTC-0500	HookTestStack	REVIEW_IN_PROGRESS	User Initiated	-

如果已經為 CloudFormation 勾點啟用日誌，您可以擷取勾點評估結果。以下是狀態為失敗的勾點範例日誌，表示 Lambda 函數未啟用 X-Ray：

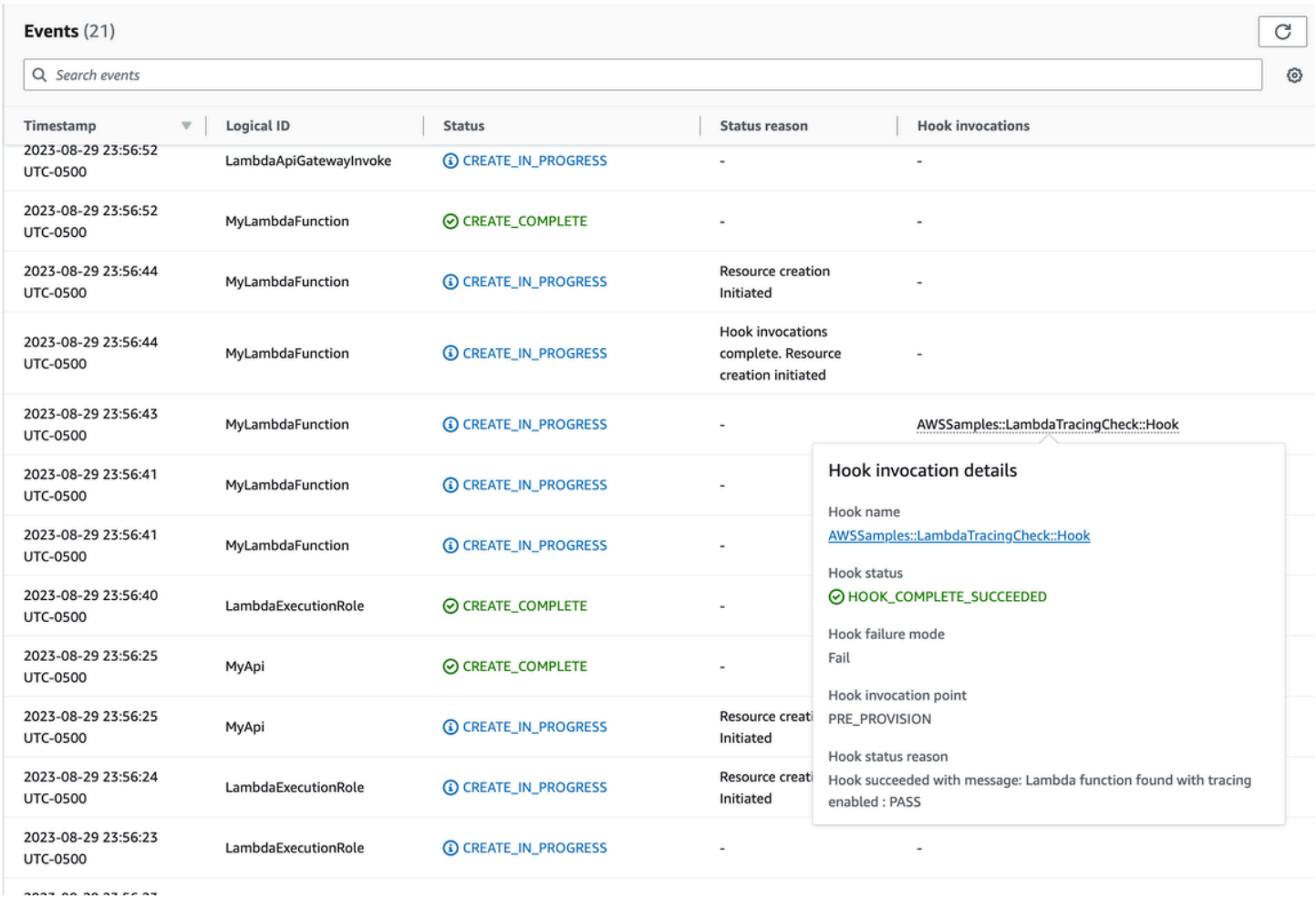
2023-08-29T23:50:17.574-05:00 ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'…

```
ProgressEvent(status=<OperationStatus.FAILED: 'FAILED'>, errorCode=<HandlerErrorCode.NonCompliant: 'NonCompliant'>, message='Lambda
function found with no tracing enabled : FAILED', result=None, callbackContext=None, callbackDelaySeconds=0, resourceModel=None,
resourceModels=None, nextToken=None)
```

Copy

No newer events at this moment. *Auto retry paused.* Resume

如果開發人員選擇將 IaC 變更為將 `TracingConfig Mode` 值更新為 `Active` 並重新部署，勾點將成功執行且堆疊將繼續建立 Lambda 資源。



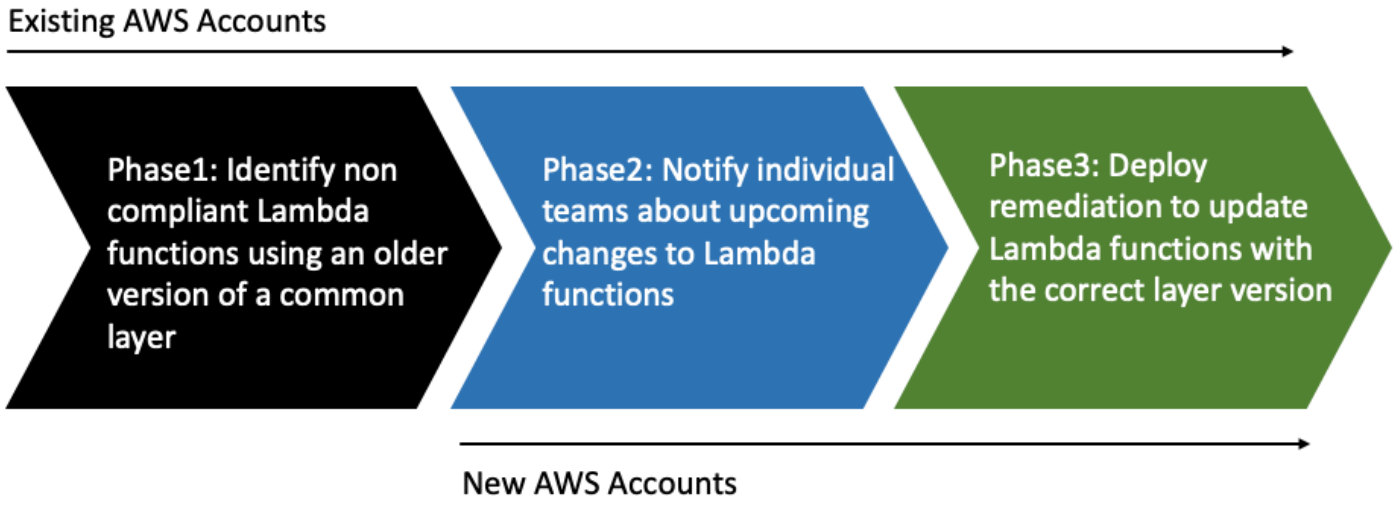
透過這種方式，您可以在 AWS 帳戶中開發和部署無伺服器資源時， AWS Config 以主動模式使用 實作預防性控制。透過將 AWS Config 規則整合到 CI/CD 管道，您可以識別並有針對性地封鎖不合規的資源部署，例如缺少主動追蹤組態的 Lambda 函數。這可確保只有符合最新控管政策的資源才會部署到您的 AWS 環境中。

使用 偵測不合規的 Lambda 部署和組態 AWS Config

除了主動評估之外， AWS Config 也可以被動偵測不符合控管政策的資源部署和組態。這一點很重要，因為控管政策會隨著您的組織學習和實作新的最佳實務而演進。

請考慮在部署或更新 Lambda 函數時設定全新政策的情形：所有 Lambda 函數都必須始終使用特定且經過核准的 Lambda 層版本。您可以將 AWS Config 設為監控新的或更新函數的層組態。如果 AWS Config 偵測到未使用已核准 layer 版本的函數，則會將該函數標記為不合規資源。您可以選擇使用 AWS Systems Manager 自動化文件指定修復動作， AWS Config 以設定 自動修復資源。例如，您可以使用 在 Python 中撰寫自動化文件 適用於 Python (Boto3) 的 AWS SDK，這會更新不合規函數以指向核准的 layer 版本。因此， 同時 AWS Config 做為偵測和修正控制，自動化合規管理。

我們將此過程分解為三個重要的實作階段：



階段 1：確定存取資源

首先在您的帳戶 AWS Config 中啟用 ，並將其設定為記錄 AWS Lambda 函數。這可讓 AWS Config 觀察何時建立或更新 Lambda 函數。然後，您可以設定自訂政策規則，以檢查使用 AWS CloudFormation Guard 語法的特定策略違規情況。Guard 規則採用下列一般形式：

```
rule name when condition { assertion }
```

以下是一條範例規則，用於檢查確保層未被設為舊的層版本：

```
rule desiredlayer when configuration.layers !empty {
```

```
    some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn
}
```

我們來了解一下規則語法和結構：

- 規則名稱：在提供的範例中，規則的名稱為 `desiredlayer`。
- 條件：此子句指定應該檢查規則的條件。在提供的範例中，條件為 `configuration.layers ! empty`。這意味著只有當組態中的 `layers` 屬性不為空時，才應評估資源。
- 聲明：在 `when` 子句後，聲明將決定檢查什麼規則。聲明 `some configuration.layers[*].arn != CONFIG_RULE_PARAMETERS.OldLayerArn` 會檢查任何 Lambda 層 ARN 是否與 `OldLayerArn` 值不相符。如果不相符，聲明為 true 且規則通過；否則，它將會失敗。

`CONFIG_RULE_PARAMETERS` 是一組使用 AWS Config 規則設定的特殊參數。在這種情況下，`OldLayerArn` 是 `CONFIG_RULE_PARAMETERS` 內的一項參數。它允許使用者提供其認為舊的或已棄用的特定 ARN 值，然後，規則會檢查任何 Lambda 函數是否使用這個舊的 ARN 值。

階段 2：視覺化與設計

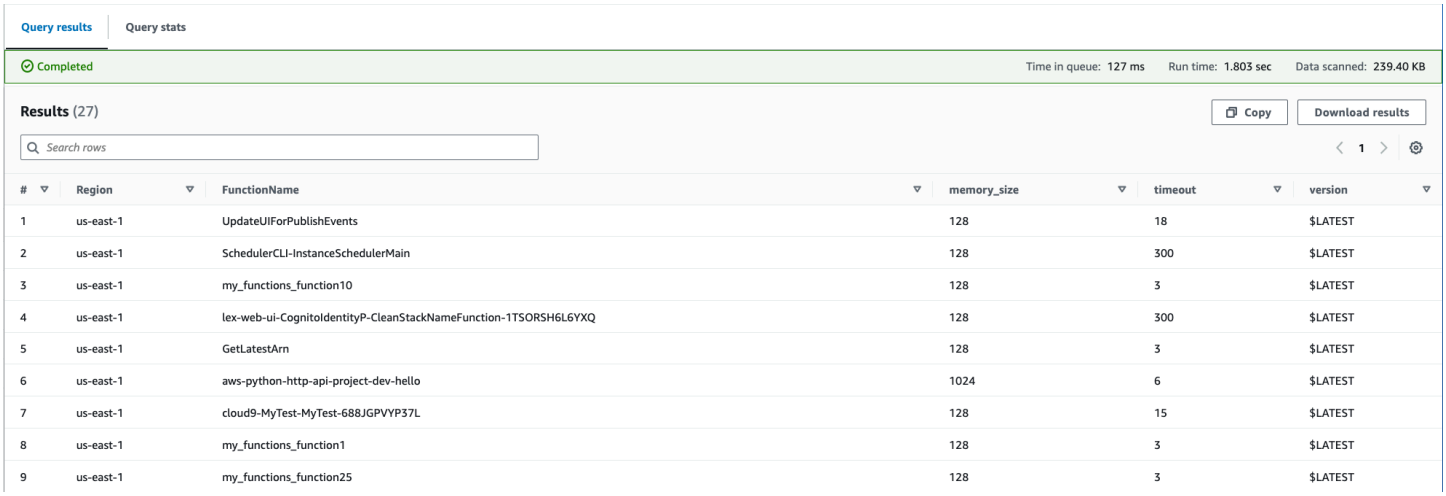
AWS Config 會收集組態資料，並將該資料存放在 Amazon Simple Storage Service (Amazon S3) 儲存貯體中。您可以使用 Amazon Athena 直接從 S3 儲存貯體查詢這些資料。藉助 Athena，您可以在組織層級彙整此類資料，為所有帳戶產生資源組態的整體全貌。若要設定資源組態資料的彙總，請參閱 AWS 雲端操作和管理部落格上的使用 Athena 和 Amazon Quick 視覺化 AWS Config 資料。

以下是使用特定層 ARN 識別所有 Lambda 函數的範例 Athena 查詢：

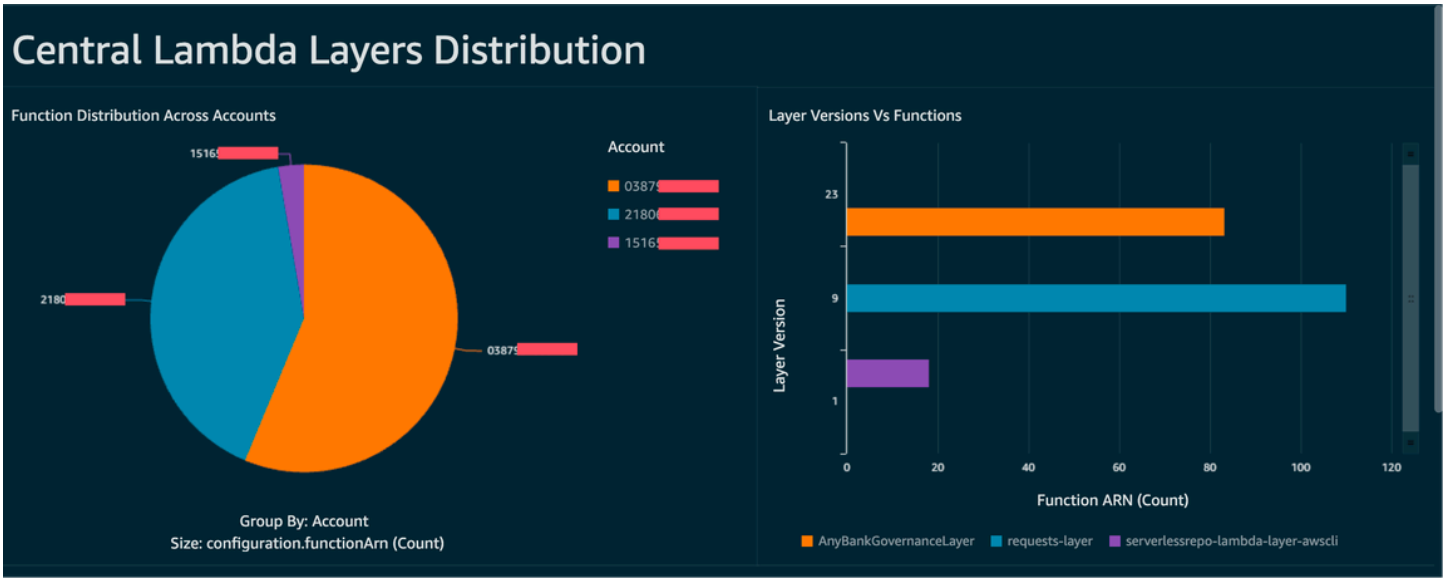
```
WITH unnested AS (
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
    WHERE
```

```
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.version') AS version
  FROM
    unnested
  WHERE
    lambda_configuration LIKE '%arn:aws:lambda:us-
east-1:111122223333:layer:AnyGovernanceLayer:24%'
```

以下是查詢的結果：



透過彙總整個組織 AWS Config 的資料，您可以使用 Amazon Quick 建立儀表板。透過將 Athena 結果匯入 Quick，您可以將 Lambda 函數遵循 layer 版本規則的程度視覺化。此儀表板可以突出顯示合規和不合規的資源，依下一個區段中所述協助您確定強制政策。下圖是一個範例儀表板，它會報告在組織內部套用至函數的層版本的分佈情況。



階段 3：實作與強制執行

您現在可以選擇性地將您在階段 1 中建立的層版本規則與透過 Systems Manager 自動化文件執行的修補動作配對，該自動化文件是您使用 適用於 Python (Boto3) 的 AWS SDK編寫的 Python 指令碼。此指令碼會為每個 Lambda 函數呼叫 UpdateFunctionConfiguration API 動作，使用新的層 ARN 更新函數組態。或者，您可以讓指令碼提交提取請求至程式碼儲存庫，以更新層 ARN。這樣，未來的程式碼部署也會使用正確的層 ARN 進行更新。

使用 AWS Signer 的 Lambda 程式碼簽署

AWS Signer 是一項全受管程式碼簽署服務，可讓您對照數位簽章驗證您的程式碼，以確認該程式碼未經變更且來自受信任的發布者。AWS Signer 可與 AWS Lambda 搭配使用，以便驗證函數和層在部署到您的 AWS 環境前未經變更。這可以保護您的組織免受惡意行為者侵害，這些行為者可能已取得憑證以建立新的或更新現有的函數。

若要為 Lambda 函數設定程式碼簽署，請先建立啟用版本控制的 S3 儲存貯體。之後，使用 AWS Signer 建立簽署設定檔，將 Lambda 指定為平台，然後指定簽署設定檔的有效天數。範例：

```
Signer:
  Type: AWS::Signer::SigningProfile
  Properties:
    PlatformId: AWSLambda-SHA384-ECDSA
    SignatureValidityPeriod:
      Type: DAYS
      Value: !Ref pValidDays
```

接下來使用簽署設定檔和 Lambda 來建立簽署組態。當簽署組態遇到與預期的數位簽章不相符的成品時，您必須指定要執行何種操作：警告 (但允許部署) 或強制執行 (並封鎖部署)。以下範例設為強制執行並封鎖部署。

```
SigningConfig:
  Type: AWS::Lambda::CodeSigningConfig
  Properties:
    AllowedPublishers:
      SigningProfileVersionArns:
        - !GetAtt Signer.ProfileVersionArn
    CodeSigningPolicies:
      UntrustedArtifactOnDeployment: Enforce
```

您現有的 AWS Signer 設定為由 Lambda 封鎖不受信任的部署。假設您已經完成功能請求的編碼，現在已準備好部署函數。第一個步驟是壓縮程式碼及適當相依性，然後使用您建立的簽署設定檔簽署成品。為此，您可以上傳壓縮成品到 S3 儲存貯體，然後開始簽署任務。

```
aws signer start-signing-job \
--source 's3={bucketName=your-versioned-bucket,key=your-prefix/your-zip-
artifact.zip,version=QyaJ3c4qa50LXV.9VaZgXHlsGbvCXxpT}' \
--destination 's3={bucketName=your-versioned-bucket,prefix=your-prefix/}' \
--profile-name your-signer-id
```

您將取得如下輸出，其中 jobId 是在目的地儲存貯體和字首中建立的物件，jobOwner 是執行該任務的 12 位數 AWS 帳戶 ID。

```
{
    "jobId": "87a3522b-5c0b-4d7d-b4e0-4255a8e05388",
    "jobOwner": "111122223333"
  }
```

現在，您可以使用已簽署的 S3 物件和您建立的程式碼簽署組態來部署您的函數。

```
  Fn:
    Type: AWS::Serverless::Function
    Properties:
      CodeUri: s3://your-versioned-bucket/your-prefix/87a3522b-5c0b-4d7d-
b4e0-4255a8e05388.zip
      Handler: fn.handler
      Role: !GetAtt FnRole.Arn
      CodeSigningConfigArn: !Ref pSigningConfigArn
```

或者，您還可以使用原始未簽署的來源壓縮成品測試函數部署。部署會失敗並顯示以下訊息：

```
Lambda cannot deploy the function. The function or layer might be signed using a
 signature that the client is not configured to accept. Check the provided signature
 for unsigned.
```

若您使用 AWS Serverless Application Model (AWS SAM) 建置與部署您的函數，套件命令將處理壓縮成品上傳到 S3，還會開始簽署任務並取得已簽署成品。您可以使用以下命令和參數來執行此動作：

```
sam package -t your-template.yaml \
--output-template-file your-output.yaml \
--s3-bucket your-versioned-bucket \
--s3-prefix your-prefix \
--signing-profiles your-signer-id
```

AWS Signer 會協助您確認部署到帳戶中的壓縮成品是否受信任用於部署。您可以在 CI/CD 管道中包含上述程序，並要求所有函數都使用先前主題中所述的技術附加程式碼簽署組態。透過搭配 Lambda 函數部署使用程式碼簽署，您可以防止可能已取得憑證的惡意行為者建立或更新函數，將惡意程式碼注入函數中。

使用 Amazon Inspector 自動化 Lambda 的安全評估

Amazon Inspector 是一項漏洞管理服務，可持續掃描工作負載，以尋找已知的軟體漏洞和意外的網路暴露。Amazon Inspector 會建立調查結果以描述漏洞、識別受影響的資源、評定漏洞嚴重性並提供修補指引。

Amazon Inspector 支援為 Lambda 函數和層提供持續的自動化安全性漏洞評估。Amazon Inspector 為 Lambda 提供兩種掃描類型：

- Lambda 標準掃描 (預設)：掃描 Lambda 函數及其層內的應用程式相依性，以尋找套件漏洞。
- Lambda 程式碼掃描：掃描函數和層內的自訂應用程式程式碼，以尋找程式碼漏洞。您可以啟動 Lambda 標準掃描，或同時啟動 Lambda 標準掃描和 Lambda 程式碼掃描。

若要啟用 Amazon Inspector，請導覽至 Amazon Inspector 主控台，展開設定區段，然後選擇帳戶管理。在帳戶索引標籤上，選擇啟動，然後選取其中一個掃描選項。

您可以為多個帳戶啟用 Amazon Inspector，並在設定 Amazon Inspector 時為特定帳戶的組織委派管理 Amazon Inspector 的許可。啟用時，您需要透過建立以下角色來授予 Amazon Inspector 許可：`AWSServiceRoleForAmazonInspector2`。Amazon Inspector 主控台允許您使用一鍵式選項建立此角色。

對於 Lambda 標準掃描，Amazon Inspector 會在下列情況下啟動 Lambda 函數的漏洞掃描。

- 一旦 Amazon Inspector 發現現有 Lambda 函數。
- 當您部署新的 Lambda 函數時。
- 當您對現有的 Lambda 函數或其層的應用程式程式碼或相依性部署更新時。
- 每當 Amazon Inspector 新增一個常見漏洞和暴露 (CVE) 項目到其資料庫，而該 CVE 與您的函數相關時。

對於 Lambda 程式碼掃描，Amazon Inspector 會使用自動推理和機器學習來評估您的 Lambda 函數應用程式程式碼，以分析您的應用程式程式碼是否符合整體安全規範。若 Amazon Inspector 在您的 Lambda 函數應用程式程式碼中偵測到漏洞，Amazon Inspector 會產生一份詳細的程式碼漏洞調查結果。如需檢視可能偵測項目的清單，請參閱 Amazon CodeGuru Detector 程式庫。

若要檢視調查結果，請前往 Amazon Inspector 主控台。在調查結果選單上，選擇依 Lambda 函數以顯示對 Lambda 函數執行的安全性掃描結果。

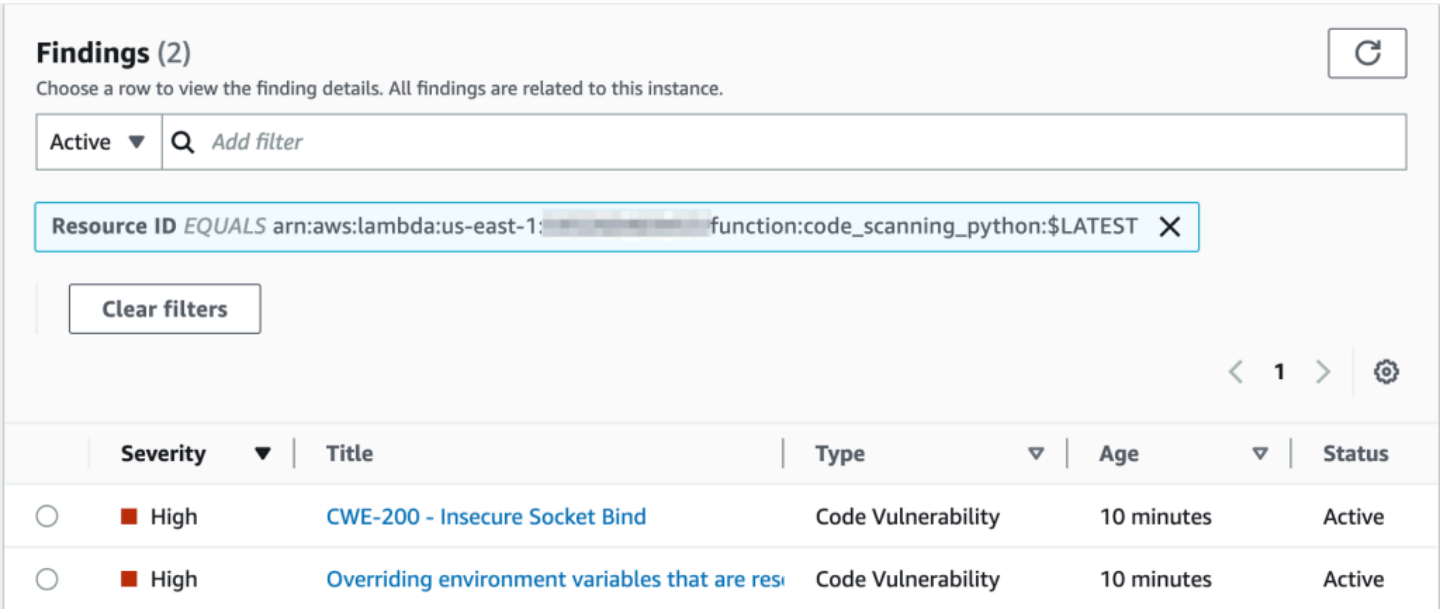
若要從標準掃描中排除 Lambda 函數，請使用以下鍵值對標記函數：

- `Key:InspectorExclusion`
- `Value:LambdaStandardScanning`

若要從程式碼掃描中排除 Lambda 函數，請使用以下鍵值對標記函數：

- `Key:InspectorCodeExclusion`
- `Value:LambdaCodeScanning`

例如，如下圖所示，Amazon Inspector 會自動偵測漏洞並將發現的漏洞分類為程式碼漏洞類型，這表示這些漏洞存在於函數的程式碼中，而不是在其中一個程式碼相依程式庫中。您可以一次檢查一個特定函數或多個函數的這些詳細資訊。



您可以深入研究每項結果，並了解如何修補問題。

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior.

Finding ID: arn:aws:inspector2:us-east-1:[redacted]:finding/[redacted]

Overriding environment variables that are reserved by AWS Lambda might lead to unexpected behavior or failure of the Lambda function.

Finding overview	
AWS account ID	[redacted]
Severity	High
Type	Code Vulnerability
Detector name	Override of reserved variable names in a Lambda function
Relevant CWE	--
Rule ID	Rule-434311
Detector tags	#availability, #aws-python-sdk, #aws-lambda, #data-integrity, #maintainability, #security, #security-context, #python
Fix available	Yes
Created at	March 29, 2023 10:08 AM (UTC-04:00)

Vulnerability details	
File path	lambda_function.py

Vulnerability location

```
3  import socket
4
5  def lambda_handler(event, context):
6
7      # print("Scenario 1");
8      os.environ['_HANDLER'] = 'hello'
9      # print("Scenario 1 ends")
10
11     # print("Scenario 2");
12     s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
13     s.bind(('',0))
```

Suggested remediation

Your code attempts to override an environment variable that is reserved by the Lambda runtime environment. This can lead to unexpected behavior and might break the execution of your Lambda function.

在使用您的 Lambda 函數時，確認您遵守 Lambda 函數的命名慣例。如需更多詳細資訊，請參閱 使用 Lambda 環境變數。

您要負責執行自己接受的修補建議。在接受前，請務必檢閱修補建議。您可能需要編輯修補建議，以確保您的程式碼符合您的預期。

實作 Lambda 安全性與合規性的可觀測性

AWS Config 是尋找和修正不合規 AWS Serverless 資源的實用工具。您對無伺服器資源所做的每個變更都會記錄在其中 AWS Config。此外， AWS Config 可讓您在 S3 上存放組態快照資料。您可以使用 Amazon Athena 和 Amazon Quick 建立儀表板並查看 AWS Config 資料。在 使用 偵測不合規的 Lambda 部署和組態 AWS Config 中，我們討論了如何視覺化 Lambda 層等特定組態。本主題將展開介紹這些概念。

Lambda 組態的可見性

您可以使用查詢來提取重要的組態，例如 AWS 帳戶 ID、區域、 AWS X-Ray 追蹤組態、VPC 組態、記憶體大小、執行時間和標籤。以下是您可以用來從 Athena 提取這些資訊的範例查詢：

```
WITH unnested AS (
    SELECT
      item.awsaccountid AS account_id,
      item.awsregion AS region,
      item.configuration AS lambda_configuration,
      item.resourceid AS resourceid,
      item.resourcename AS resourcename,
      item.configuration AS configuration,
      json_parse(item.configuration) AS lambda_json
    FROM
      default.aws_config_configuration_snapshot,
      UNNEST(configurationitems) as t(item)
    WHERE
      "dt" = 'latest'
      AND item.resourcetype = 'AWS::Lambda::Function'
  )

  SELECT DISTINCT
    account_id,
    tags,
    region as Region,
    resourcename as FunctionName,
    json_extract_scalar(lambda_json, '$.memorySize') AS memory_size,
    json_extract_scalar(lambda_json, '$.timeout') AS timeout,
    json_extract_scalar(lambda_json, '$.runtime') AS version
    json_extract_scalar(lambda_json, '$.vpcConfig.SubnetIds') AS vpcConfig
    json_extract_scalar(lambda_json, '$.tracingConfig.mode') AS tracingConfig
  FROM
    unnested
```

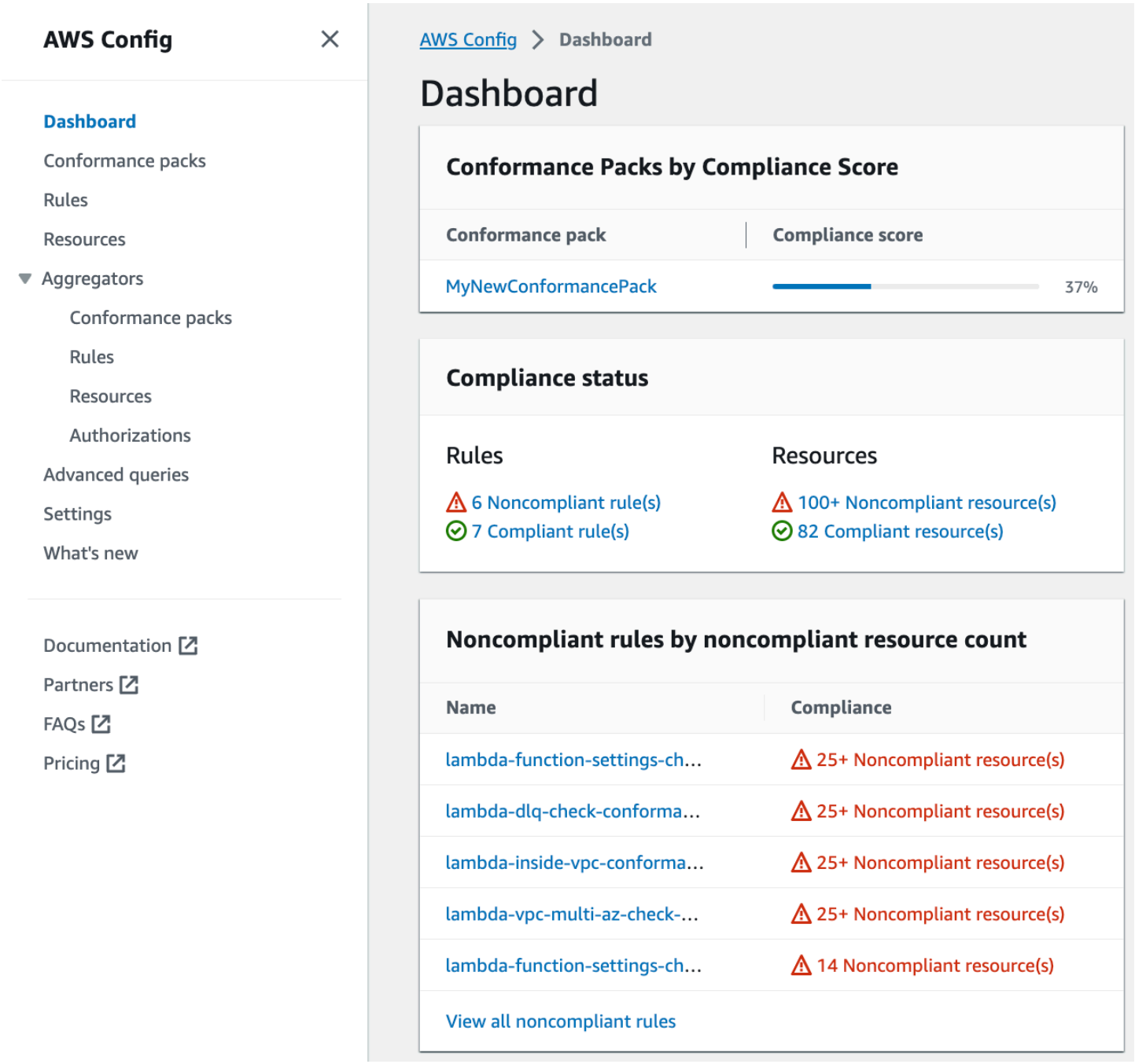
您可以使用查詢來建置快速儀表板並視覺化資料。若要彙總 AWS 資源組態資料、在 Athena 中建立資料表，並在 Athena 的資料上建置快速儀表板，請參閱 AWS 雲端操作和管理部落格中的使用 Athena 和 Amazon Quick 視覺化 AWS Config 資料。注意，此查詢還會擷取函數的標籤資訊。這可讓您更深入地了解工作負載和環境，尤其當您使用自訂標籤時。

如需有關可採取之動作的詳細資訊，請參閱本主題稍後的 處理可觀測性調查結果 區段。

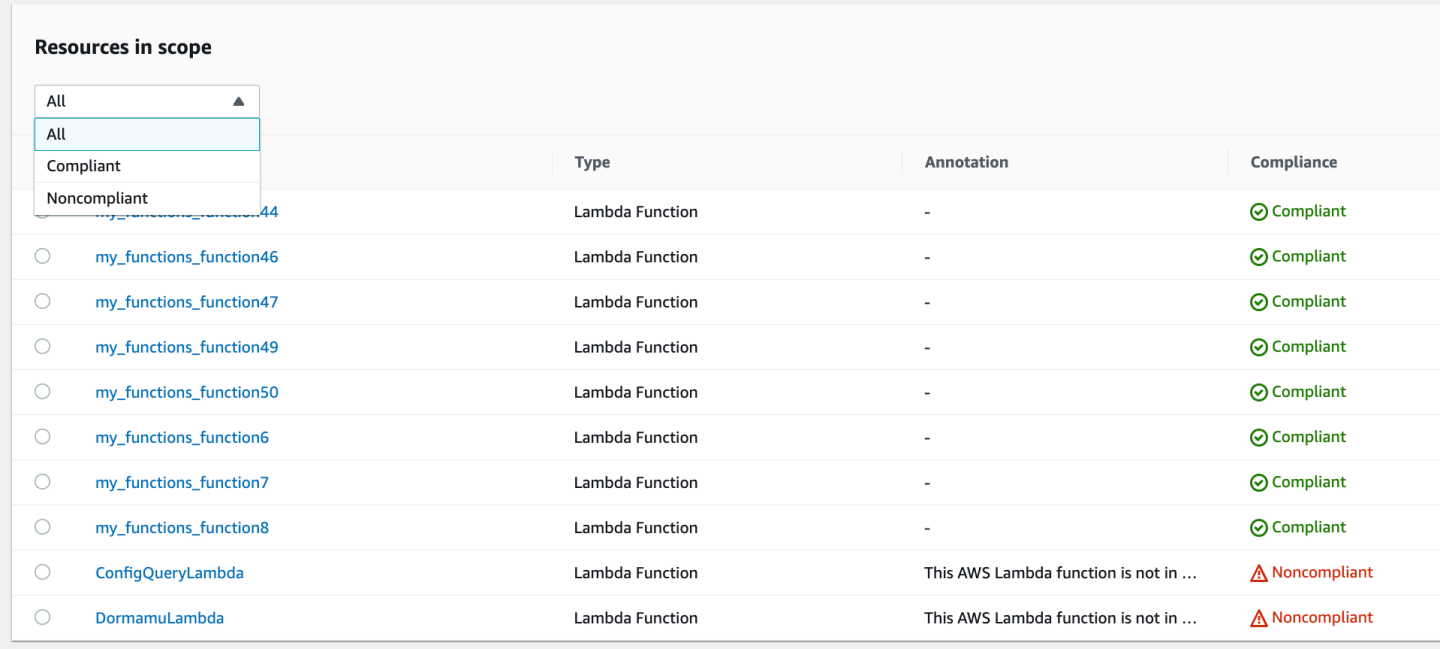
Lambda 合規可見性

使用 產生的資料 AWS Config，您可以建立組織層級儀表板來監控合規性。這允許您一致地追蹤與監控：

- 依合規分數劃分的合規性套件
- 依不合規資源劃分的規則
- 合規狀態

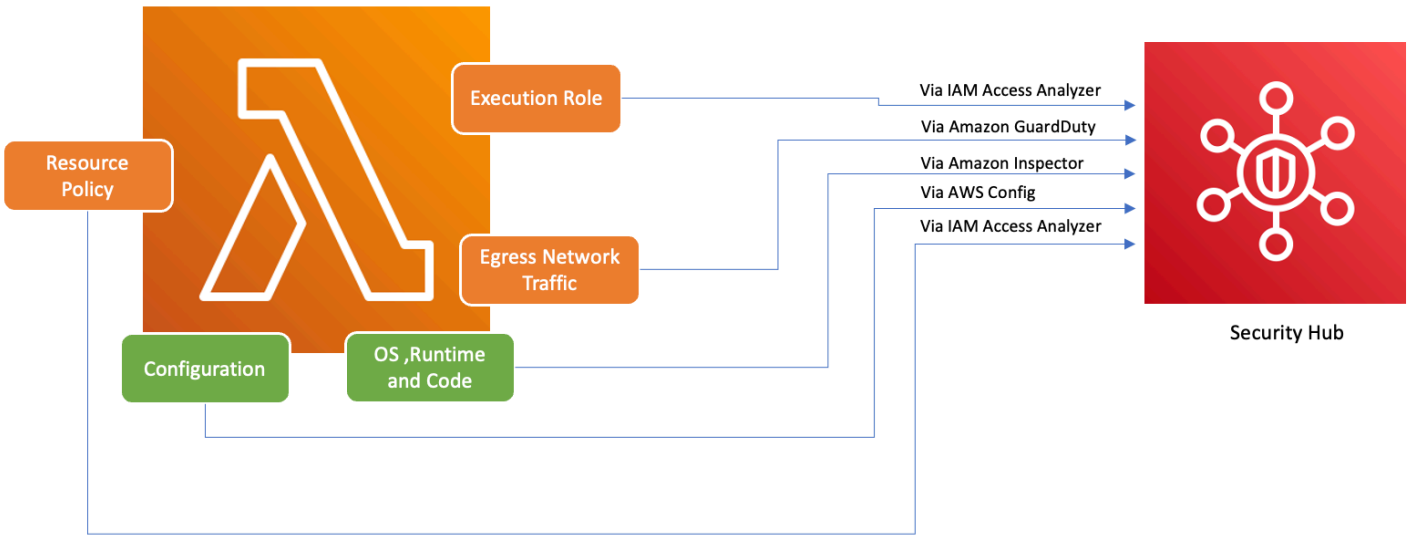


檢查每條規則，以便為該規則識別不合規的資源。例如，如果您的組織要求所有 Lambda 函數都必須與 VPC 建立關聯，而且您已部署 AWS Config 規則來識別合規，您可以在上面的清單中選取`lambda-inside-vpc`規則。



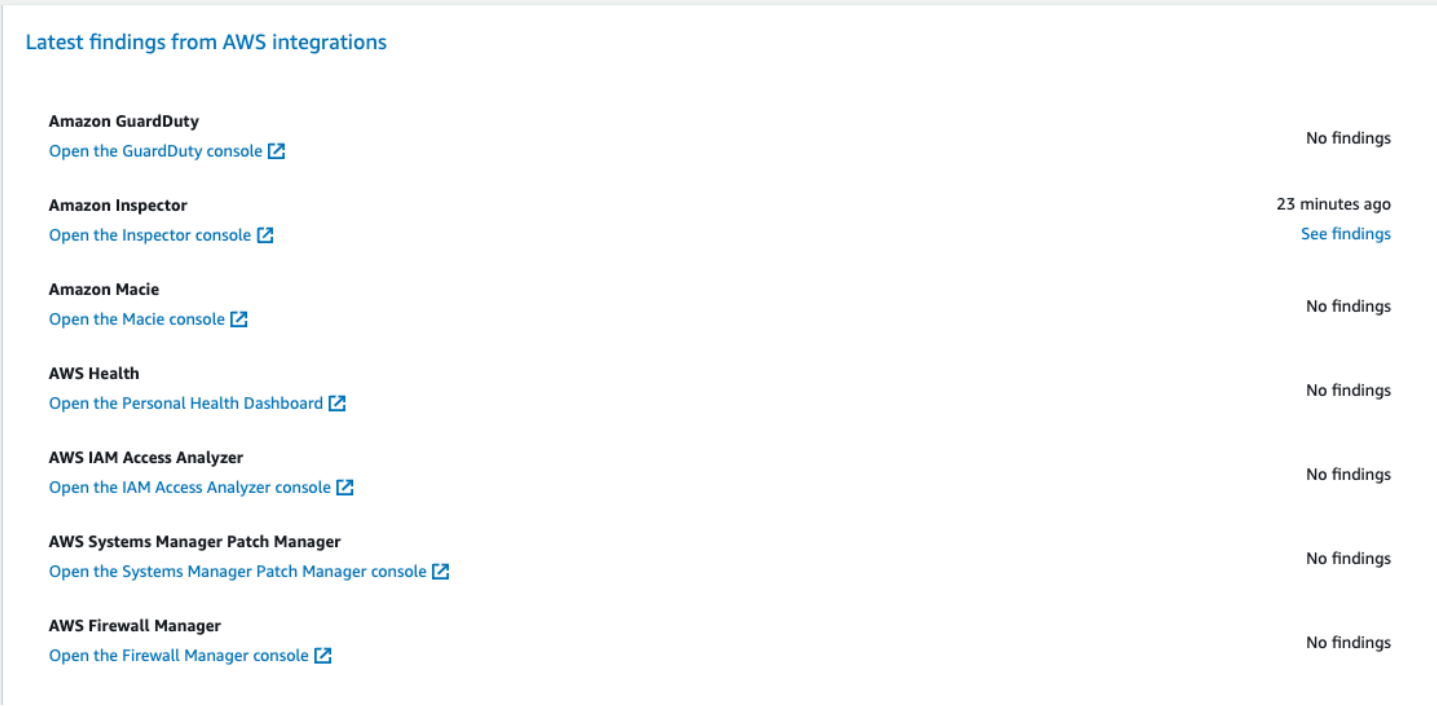
如需有關可採取之動作的詳細資訊，請參閱下方的 處理可觀測性調查結果 區段。

使用 Security Hub CSPM 對 Lambda 函數邊界的可見性

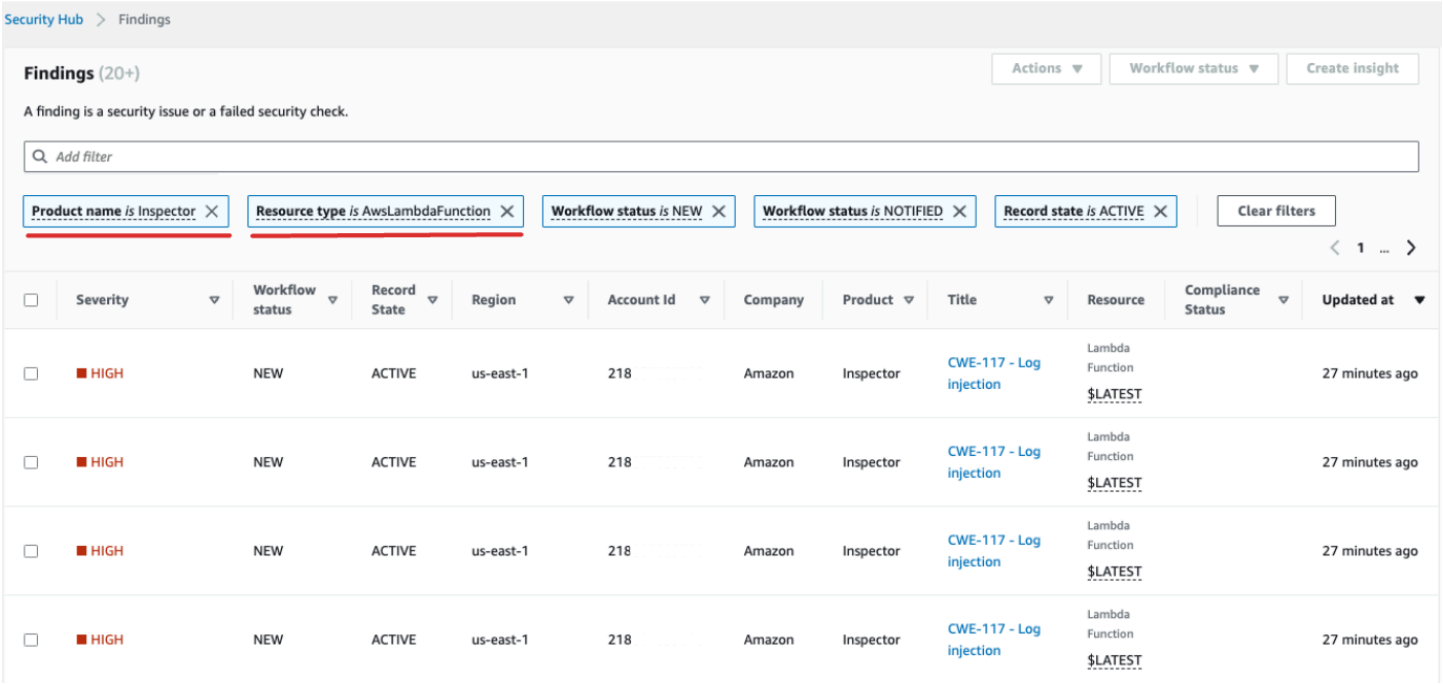


為了確保安全使用包括 Lambda 在內的 AWS 服務， AWS 推出了基礎安全最佳實務 1.0.0 版。這組最佳實務提供明確的準則，以保護 AWS 環境中的資源和資料，強調維護強式安全狀態的重要性。透過提供統一的安全和合規中心來 AWS Security Hub CSPM 補充這一點。它會彙總、組織 Amazon Inspector 和 Amazon GuardDuty 等多項 AWS 服務的安全性調查結果 AWS Identity and Access Management Access Analyzer，並排定其優先順序。

如果您在 AWS 組織中啟用 Security Hub CSPM、Amazon Inspector、IAM Access Analyzer 和 GuardDuty，Security Hub CSPM 會自動彙總這些服務的問題清單。例如，我們來考慮一下 Amazon Inspector。使用 Security Hub CSPM，您可以有效地識別 Lambda 函數中的程式碼和套件漏洞。在 Security Hub CSPM 主控台中，導覽至標記來自 AWS 整合的最新調查結果的底部區段。在這裡，您可以檢視和分析來自各種整合 AWS 服務的問題清單。



若要查看詳細資訊，請選擇第二欄中的查看調查結果連結。這會顯示依產品 (如 Amazon Inspector) 篩選的調查結果清單。若要將搜尋限定於 Lambda 函數，請將 `ResourceType` 設為 `AwsLambdaFunction`。這會顯示 Amazon Inspector 中與 Lambda 函數相關的調查結果。



對於 GuardDuty，您可以識別可疑的網路流量模式。此類異常可能表明 Lambda 函數中存在潛在的惡意程式碼。

藉助 IAM Access Analyzer，您可以檢查政策，特別是那些包含條件陳述式的政策，這些陳述式會授予函數存取外部實體的權限。此外，當 IAM Access Analyzer 在 Lambda API 中使用 AddPermission 操作並結合 `EventSourceToken` 時，還會對許可集合進行評估。

處理可觀測性調查結果

考慮到 Lambda 函數可能的廣泛組態及其獨特要求，一種標準化的自動修補解決方案可能不適用於所有情況。此外，在各種環境中，變更的實作方式也存在差異。如果您遇到任何看似不合規的組態，請考慮下列指導方針：

1. 標記策略

   我們建議實作全面的標記策略。每個 Lambda 函數都應使用關鍵資訊進行標記，例如：

   - 擁有者：對函數負責的人員或團隊。
   - 環境：生產、整備、開發或沙盒。
   - 應用程式：此函數所屬的更廣泛上下文 (如果適用)。

2. 擁有者外展

有別於自動執行重大變更 (如 VPC 組態調整)，主動聯絡不合規函數的擁有者 (透過擁有者標籤識別) 可為其提供充足的時間，以便其：

- 調整 Lambda 函數的不合規組態。
- 提供說明並請求例外狀況，或完善合規標準。

3. 維護組態管理資料庫 (CMDB)

   雖然標籤可以提供直接上下文，但維護集中式 CMDB 可以提供更深入的洞見。它可以保留有關每個 Lambda 函數的更精細資訊、它的相依性和其他關鍵中繼資料。CMDB 是稽核、合規性檢查和識別函數擁有者的重要資源。

隨著無伺服器基礎設施的不斷演進，採用主動的監控策略變得至關重要。透過 AWS Config Security Hub CSPM 和 Amazon Inspector 等工具，可以快速識別潛在的異常或不合規組態。然而，單純使用工具無法確保完全的合規性或最佳化組態。將這些工具與妥善記載的程序及最佳實務進行配對至關重要。

- 回饋迴圈：一旦執行修補步驟，務必確保有回饋迴圈。這意味著定期重新檢視不合規的資源，以確定它們是否已更新或仍存在相同的問題。
- 記錄：始終記錄可觀測性、所執行的動作以及任何授予的例外狀況。適當記錄不僅有助於稽核，還可協助強化該程序以便在未來提高合規性與安全性。
- 培訓和認知：確保所有利害關係人，尤其是 Lambda 函數擁有者，定期接受培訓並了解最佳實務、組織政策以及合規性要求。定期舉辦研討會、網路研討會或提供培訓課程，可進一步確保每個人在安全性和合規性方面取得共識。

總之，儘管工具和技術提供了強大的偵測與標記潛在問題的功能，但人的因素 (理解、溝通、培訓和記錄) 仍然是關鍵。它們共同構成一個強效的組合，可確保您的 Lambda 函數和更廣泛的基礎設施保持合規、安全並針對您的商業需求提供最佳化。

的合規驗證AWS Lambda

在多個 AWS Lambda 合規計劃中，第三方稽核人員會評估 AWS 的安全與合規。這些計劃包括 SOC、PCI、FedRAMP、HIPAA 等等。

如需特定合規計劃範圍內的 AWS 服務清單，請參閱合規計劃內的 AWS 服務。如需一般資訊，請參閱 AWS 合規計劃。

您可使用 AWS Artifact 下載第三方稽核報告。如需詳細資訊，請參閱在 AWS Artifact 中下載報告。

您使用 Lambda 的合規責任，取決於資料的機密性、您公司的合規目標及適用法律和法規。您可以實作控管控制項，以確保公司的 Lambda 函數符合您的合規需求。如需更多詳細資訊，請參閱 建立 Lambda 函數和層的控管策略。

AWS Lambda 中的恢復能力

AWS 全球基礎設施是以 AWS 區域與可用區域為中心建置的。AWS區域提供多個分開且隔離的實際可用區域，它們以低延遲、高輸送量和高度備援聯網功能相互連結。透過可用區域，您所設計與操作的應用程式和資料庫，就能夠在可用區域之間自動容錯移轉，而不會發生中斷。可用區域的可用性、容錯能力和擴充能力，均較單一或多個資料中心的傳統基礎設施還高。

如需 AWS 區域與可用區域的詳細資訊，請參閱 AWS 全球基礎設施。

除了 AWS 全球基礎設施，Lambda 還提供數種功能，可協助支援資料的彈性和備份需求。

- 版本控制 - 您可以在 Lambda 中使用版本控制，在開發時儲存您函數的程式碼和組態。搭配別名，您可以使用版本控制來執行藍綠和輪流部署。如需詳細資訊，請參閱 管理 Lambda 函數版本。
- 擴展 - 當您的函數在處理先前請求的同時收到請求，Lambda 會啟動函數的另一個執行個體來處理增加的負載。Lambda 會自動擴展以處理每個區域 1,000 個並行執行，如果需要，可以增加配額。如需詳細資訊，請參閱 了解 Lambda 函數擴展。
- 高可用性 - Lambda 會在多個可用區域中執行您的函數，確保單一區域的服務中斷時，其可用於處理事件。如果您將函式設定為連接到您帳戶中的虛擬私有雲端 (VPC)，請在多個可用區域中指定子網路，以確保高可用性。如需詳細資訊，請參閱 讓 Lambda 函數存取 Amazon VPC 中的資源。
- 預留並行 - 若要確保您的函數可隨時擴展以處理額外的請求，您可以為其預留並行。為函式設定預留並行，確保其可擴展為 (但不超過) 指定的並行叫用數目。這可確保您不會因為取用所有可用並行的其他函式而遺失請求。如需詳細資訊，請參閱 設定函數的預留並行。

- 重試 - 對於非同步叫用以及其他服務所觸發的叫用子集，Lambda 會在發生錯誤時重試，且重試之間有延遲。同步調用函數的其他用戶端和 AWS 服務負責執行重試。如需詳細資訊，請參閱 了解 Lambda 中的重試行為。
- 無效字母佇列 - 對於非同步叫用，您可以設定 Lambda，以便在所有重試都失敗時，將請求傳送到無效字母佇列。無效字母佇列是可接收事件以便排除故障或重新處理的 Amazon SNS 主題或 Amazon SQS 佇列。如需詳細資訊，請參閱 新增無效字母佇列。

中的基礎設施安全AWS Lambda

做為一種受管服務，AWS Lambda 受 AWS 全域網路安全的保護。如需 AWS 安全服務以及 AWS 如何保護基礎設施的相關資訊，請參閱 AWS 雲端安全。若要使用基礎設施安全性的最佳實務以設計您的 AWS 環境，請參閱安全性支柱 AWS 架構良好的框架中的基礎設施保護。

您可使用 AWS 發佈的 API 呼叫，透過網路來存取 Lambda。使用者端必須支援下列專案：

- Transport Layer Security (TLS)。我們需要 TLS 1.2 並建議使用 TLS 1.3。
- 具備完美轉送私密(PFS)的密碼套件，例如 DHE (Ephemeral Diffie-Hellman)或 ECDHE (Elliptic Curve Ephemeral Diffie-Hellman)。現代系統(如 Java 7 和更新版本)大多會支援這些模式。

使用公有端點保護工作負載

對於可公開存取的工作負載，AWS 提供了許多有助於降低特定風險的功能和服務。本節涵蓋應用程式使用者的身分驗證和授權，以及保護 API 端點。

身分驗證和授權

身分驗證與身分相關，授權涉及動作。使用身分驗證來控制誰可以調用 Lambda 函數，然後使用授權來控制調用者可以執行的動作。對於許多應用程式，IAM 足以管理這兩個控制機制。

對於具有外部使用者的應用程式 (例如 Web 或行動應用程式)，通常使用 JSON Web 權杖 (JWT) 來管理身分驗證和授權。與傳統的伺服器型密碼管理不同，JWT 在每次請求時都會從用戶端傳遞。它們是一種加密的安全方式，可使用從用戶端傳遞的資料來驗證身分和宣告。對於 Lambda 型應用程式，這可讓您保護對每個 API 端點的每次呼叫，而無需依賴中央伺服器進行身分驗證。

您可以使用 Amazon Cognito 實作 JWT。Amazon Cognito 是一種使用者目錄服務，可以處理註冊、身分驗證、帳戶復原及其他常見帳戶管理操作。Amplify Framework 提供了一些程式庫，可簡化將此服務整合到前端應用程式的程序。您也可以考慮使用第三方合作夥伴服務，例如 Auth0。

鑑於身分提供者服務的關鍵安全角色，請務必使用專業工具來保護您的應用程式。不建議您自己編寫服務，來處理身分驗證或授權。自訂程式庫中的任何漏洞都可能對您的工作負載及其資料的安全性產生重大影響。

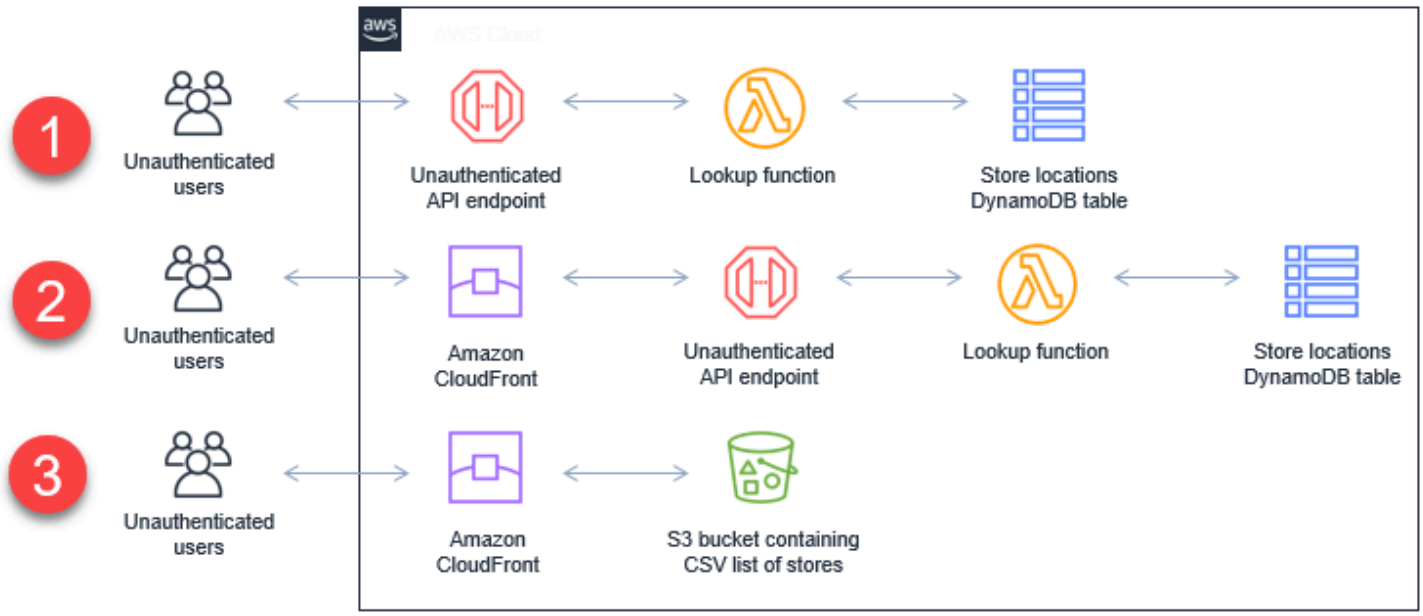
保護 API 端點

對於無伺服器應用程式，公開為後端應用程式提供服務的偏好方式是使用 Amazon API Gateway。這可協助您保護 API 免受惡意使用者或流量激增的侵害。

API Gateway 為無伺服器開發人員提供兩種端點類型：REST API 和 HTTP API。兩者都支援使用 AWS Lambda、IAM 或 Amazon Cognito 進行授權。使用 IAM 或 Amazon Cognito 時，會評估傳入請求，如果這些請求缺少所需的權杖或包含無效的身分驗證，則請求會遭到拒絕。您無需為這些請求付費，並且這些請求不計入任何限流配額。

公有網際網路上的任何人皆可存取未驗證的 API 路由，因此建議您限制使用未驗證的 API。如果必須使用未進行身分驗證的 API，請務必保護這些 API 免受常見風險，例如拒絕服務 (DoS) 攻擊。將 AWS WAF 套用至這些 API，有助於保護應用程式免受 SQL injection 隱碼攻擊和跨網站指令碼 (XSS) 攻擊。使用 API 金鑰時，API Gateway 還會在 AWS 帳戶層級和每個用戶端層級實作限流。

在許多情況下，未驗證的 API 提供的功能可以透過替代方式來實現。例如，Web 應用程式可以向未登入的使用者提供來自 DynamoDB 資料表的客戶零售店清單。此請求可能源自前端 Web 應用程式或任何其他呼叫 URL 端點的來源。此圖表比較了以下三種解決方案：



1. 網際網路上的任何人都可以呼叫此未驗證的 API。在拒絕服務攻擊中，可能會耗盡基礎資料表上的 API 限流限制、Lambda 並行或 DynamoDB 佈建讀取容量。

2. 在 API 端點前面使用適當的存留時間 (TTL) 組態的 CloudFront 分佈將承受 DoS 攻擊中的大部分流量，而無需變更用於擷取資料的底層解決方案。
3. 或者，對於很少變更的靜態資料，CloudFront 分佈可以提供來自 Amazon S3 儲存貯體的資料。

使用程式碼簽署來驗證 Lambda 的程式碼完整性

程式碼簽署可協助確保僅受信任的程式碼部署到您的 Lambda 函式中。您可以使用 AWS Signer為您的函數建立數位簽章程式碼套件。當您將程式碼簽署組態新增至函式時，Lambda 會驗證所有新的程式碼部署是否皆由信任的來源簽署。由於程式碼簽署驗證檢查會在部署時執行，因此不會影響函式執行。

> **Important**
>
> 程式碼簽署組態只能防止新部署未簽署的程式碼。如果將程式碼簽署組態新增至具有未簽署程式碼的現有函式，該程式碼會持續執行，直到您部署新的程式碼套件為止。

當您為函式啟用程式碼簽署時，新增至函式的所有層也必須由一個允許的簽署設定檔來簽署。

使用 AWS Signer 或 程式碼簽署無需額外費用 AWS Lambda。

簽署驗證

在您將已簽署的程式碼套件部署至函數時，Lambda 會執行下列驗證檢查：

1. 完整性：驗證程式碼套件自簽署後未被修改。Lambda 將套件的雜湊與簽章中的雜湊作比較。
2. 有效期：驗證程式碼套件的簽章尚未到期。
3. 一致性：驗證程式碼套件是透過允許的簽署設定檔進行簽署
4. 撤銷撤銷：驗證程式碼套件的簽章未被撤銷。

建立程式碼簽署組態時，您可以使用 UntrustedArtifactOnDeployment 參數，指定 Lambda 如何因應有效期、一致性或撤銷狀態檢查失敗。您可以選擇下列動作之一：

- `Warn`：此動作是預設設定。Lambda 允許部署程式碼套件，但會發出警告。Lambda 會發出新的 Amazon CloudWatch 指標 (`SignatureValidationErrors`)，還會將警告儲存在 CloudTrail 日誌中。
- `Enforce`：Lambda 會發出警告 (與 `Warn` 動作相同) 並阻止程式碼套件的部署。

主題

- 建立 Lambda 的程式碼簽署組態
- 針對 Lambda 程式碼簽署組態設定 IAM 政策
- 在程式碼簽署組態上使用標籤

建立 Lambda 的程式碼簽署組態

若要啟用函數的程式碼簽署，您可以建立程式碼簽署組態並將其附加至函數。程式碼簽署組態會定義允許簽署描述檔的清單，以及在任何驗證檢查失敗時要採取的政策動作。

> **Note**
>
> 定義為容器映像的函數不支援程式碼簽署。

章節

- 組態先決條件
- 建立程式碼簽署組態
- 啟用函數的程式碼簽署

組態先決條件

在您可以設定 Lambda 函數的程式碼簽署之前，請使用 AWS Signer 執行下列動作：

- 建立一或多個簽署設定檔。
- 使用簽署設定檔為函式建立簽署的程式碼套件。

建立程式碼簽署組態

程式碼簽署組態會定義允許的簽署描述檔和簽署驗證政策。

建立程式碼簽署組態 (主控台)

1. 開啟 Lambda 主控台中的 Code signing configurations (程式碼簽署組態) 頁面。
2. 選擇建立組態。

3. 針對 Description (描述)，輸入組態的描述性名稱。
4. 在 Signing profiles (簽署描述檔) 下，可將多達 20 個簽署描述檔新增至組態。
   a. 針對 Signing profile version ARN (簽署描述檔版本 ARN)，選擇描述檔版本的 Amazon Resource Name (ARN)，或輸入 ARN。
   b. 若要新增額外的簽署描述檔，選擇 Add signing profiles (新增簽署描述檔)。
5. 在 Signature validation policy (簽署驗證政策) 下，選擇 Warn (警告) 或 Enforce (強制)。
6. 選擇建立組態。

啟用函數的程式碼簽署

若要為函式啟用程式碼簽署，需將程式碼簽署組態與函式建立關聯。

> **Important**
>
> 程式碼簽署組態只能防止新部署未簽署的程式碼。如果將程式碼簽署組態新增至具有未簽署程式碼的現有函式，該程式碼會持續執行，直到您部署新的程式碼套件為止。

將程式碼簽署組態與函數 (主控台) 關聯

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您要啟用程式碼簽署的函數。
3. 開啟 Configuration (組態) 索引標籤。
4. 向下捲動並選擇程式碼簽署。
5. 選擇編輯。
6. 在 Edit code signing (編輯程式碼簽署) 中，選擇此函數的程式碼簽署組態。
7. 選擇儲存。

針對 Lambda 程式碼簽署組態設定 IAM 政策

若要授予使用者存取 Lambda 程式碼簽署 API 操作的許可，請將一或多個政策陳述式附加至使用者政策。如需使用者政策的詳細資訊，請參閱Lambda 適用的身分型 IAM 政策。

下列範例政策陳述式會授予建立、更新及擷取程式碼簽署組態的許可。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
          "lambda:CreateCodeSigningConfig",
          "lambda:UpdateCodeSigningConfig",
          "lambda:GetCodeSigningConfig"
        ],
      "Resource": "*"
    }
  ]
}
```

管理員可以使用 CodeSigningConfigArn 條件金鑰，來指定開發人員必須用於建立或更新函數的程式碼簽署組態。

下列範例政策陳述式會授予建立函數的許可。政策聲明包括 lambda:CodeSigningConfigArn 條件以指定允許的程式碼簽署組態。如果 CodeSigningConfigArn 參數遺失或不符合條件中的值，Lambda 會阻止 CreateFunction API 請求。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "AllowReferencingCodeSigningConfig",
      "Effect": "Allow",
      "Action": [
        "lambda:CreateFunction"
      ],
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "lambda:CodeSigningConfigArn": "arn:aws:lambda:us-
east-1:111122223333:code-signing-config:csc-0d4518bd353a0a7c6"
```

```
        }
      }
    }
  ]
}
```

在程式碼簽署組態上使用標籤

您可以標記程式碼簽署組態來整理和管理您的資源。標籤是與跨 AWS 服務支援的資源相關聯的自由格式索引鍵值對。如需標籤使用案例的詳細資訊，請參閱《標記 AWS 資源和標籤編輯器指南》中的常見標記策略。

您可以使用 AWS Lambda API 來檢視和更新標籤。也可以在 Lambda 主控台中管理特定程式碼簽署時檢視和更新標籤。

章節

- 使用標籤所需的許可
- 搭配使用標籤與 Lambda 主控台
- 搭配 使用標籤 AWS CLI

使用標籤所需的許可

若要允許 AWS Identity and Access Management (IAM) 身分 (使用者、群組或角色) 讀取或設定資源上的標籤，請為其授予相應許可：

- lambda:ListTags：當資源具有標籤時，將此許可授予給需要對其呼叫 `ListTags` 的任何人員。對於已標記函數，`GetFunction` 也需要此許可。
- lambda:TagResource：將此許可授予給需要呼叫 `TagResource` 或在建立時執行標記的任何人員。

或者，也可以考慮授予 lambda：UntagResource 許可，以允許對資源進行 `UntagResource` 呼叫。

如需詳細資訊，請參閱Lambda 適用的身分型 IAM 政策。

搭配使用標籤與 Lambda 主控台

可以使用 Lambda 主控台建立具有標籤的程式碼簽署組態、將標籤新增至現有程式碼簽署組態，以及依標籤篩選程式碼簽署組態。

若要在您建立程式碼簽署組態時新增標籤

1. 開啟 Lambda 主控台中的程式碼簽署組態。
2. 從內容窗格的標頭中，選擇建立組態。
3. 在內容窗格頂端的區段中，設定程式碼簽署組態。如需關於設定程式碼簽署組態的詳細資訊，請參閱the section called “程式碼簽署”。
4. 在 標籤 區域，選擇 新增。
5. 在索引鍵欄位中，輸入標籤索引鍵。如需有關標記限制的資訊，請參閱《 AWS 標記資源和標籤編輯器指南》中的標記命名限制和要求。
6. 選擇建立組態。

若要將標籤新增至現有程式碼簽署組態

1. 開啟 Lambda 主控台中的程式碼簽署組態。
2. 選擇程式碼簽署組態的名稱。
3. 在詳細資訊窗格下方的索引標籤中，選擇標籤。
4. 選擇管理標籤。
5. 選擇 Add new tag (新增標籤)。
6. 在索引鍵欄位中，輸入標籤索引鍵。如需有關標記限制的資訊，請參閱《 AWS 標記資源和標籤編輯器指南》中的標記命名限制和要求。
7. 選擇儲存。

若要依標籤篩選程式碼簽署組態

1. 開啟 Lambda 主控台中的程式碼簽署組態。
2. 選取搜尋方塊。
3. 從下拉式清單中的標籤子標題下方，選取您的標籤。
4. 選取使用：「tag-name」以查看所有標記有此索引鍵的程式碼簽署組態，或選擇運算子以進一步依值篩選。
5. 選取標籤值，依標籤索引鍵和值的組合進行篩選。

搜尋方塊也支援搜尋標籤索引鍵。輸入索引鍵的名稱，以便在清單中尋找該索引鍵。

搭配 使用標籤 AWS CLI

可以使用 Lambda API 在現有的 Lambda 資源 (包括程式碼簽署組態) 上新增和移除標籤。也可以在建立程式碼簽署組態時新增標籤，這可讓您在整個生命週期中標記資源。

使用 Lambda 標籤 API 來更新標籤

可以透過 TagResource 和 UntagResource API 操作來新增和移除受支援 Lambda 資源的標籤。

可使用 AWS CLI來呼叫這些操作。若要將標籤新增至現有資源，請使用 `tag-resource` 命令。此範例會新增兩個標籤，一個包含索引鍵 *Department*，另一個包含索引鍵 *CostCenter*。

```
aws lambda tag-resource \
--resource arn:aws:lambda:us-east-2:123456789012:resource-type:my-resource \
--tags Department=Marketing,CostCenter=1234ABCD
```

若要移除標籤，請使用 `untag-resource` 命令。此範例會移除具有索引鍵 *Department* 的標籤。

```
aws lambda untag-resource --resource arn:aws:lambda:us-east-1:123456789012:resource-
type:resource-identifier \
--tag-keys Department
```

在建立程式碼簽署組態時新增標籤

若要建立帶有標籤的新 Lambda 程式碼簽署組態，請使用 CreateCodeSigningConfig API 操作。指定 `Tags` 參數。您可以使用 `create-code-signing-config` AWS CLI 命令和 `--tags`選項來呼叫此操作。如需關於 CLI 命令的詳細資訊，請參閱《AWS CLI 命令參考》中的 create-code-signing-config。

將 `Tags` 參數與 `CreateCodeSigningConfig` 搭配使用之前，請確保您的角色除了此操作所需的一般許可外，還擁有標記資源的許可。如需有關標記許可的詳細資訊，請參閱the section called “使用標籤所需的許可”。

使用 Lambda 標籤 API 來檢視標籤

若要檢視套用至特定 Lambda 資源的標籤，請使用 `ListTags` API 操作。如需詳細資訊，請參閱 ListTags。

您可以使用 `list-tags` AWS CLI 命令呼叫此操作，方法是提供 ARN (Amazon Resource Name)。

```
aws lambda list-tags --resource arn:aws:lambda:us-east-1:123456789012:resource-type:resource-identifier
```

依標籤篩選資源

您可以使用 AWS Resource Groups Tagging API GetResources API 操作，依標籤篩選資源。GetResources 操作可接收最多 10 個篩選條件，每個篩選條件皆包含標籤索引鍵與最多 10 個標籤值。為 GetResources 提供一個 ResourceType，即可依特定資源類型進行篩選。

您可以使用 get-resources AWS CLI 命令呼叫此操作。如需使用 get-resources 的範例，請參閱《AWS CLI 命令參考》中的 get-resources。

監控、偵錯與疑難排解 Lambda 函式

AWS Lambda 與其他 AWS 服務整合，以協助您監控和疑難排解 Lambda 函數。Lambda 代表您自動監控 Lambda 函數，並透過 Amazon CloudWatch 報告指標。為了協助您在程式碼執行時對其進行監控，Lambda 會自動追蹤請求的次數、每個請求的調用持續時間、以及導致錯誤的請求次數。

您可以使用其他 AWS 服務來疑難排解 Lambda 函數。本節描述了如何使用這些 AWS 服務來監控、追蹤、偵錯及疑難排解您的 Lambda 函數和應用程式。如需有關每個執行階段中函數記錄和錯誤的詳細資訊，請參閱個別執行階段區段。

章節

- 定價
- 將 CloudWatch 指標與 Lambda 搭配使用
- 使用 Lambda 函式日誌
- 使用 記錄 AWS Lambda API 呼叫 AWS CloudTrail
- 使用 視覺化 Lambda 函數叫用 AWS X-Ray
- 使用 Amazon CloudWatch Lambda 洞察監控函數效能
- 監控 Lambda 應用程式
- 使用 Amazon CloudWatch Application Signals 監控應用程式效能
- 使用 Visual Studio Code 遠端偵錯 Lambda 函式

定價

CloudWatch 有永久免費方案。除了免費方案閾值之外，CloudWatch 還會收取指標、儀表板、警示、記錄和洞察的費用。如需詳細資訊，請參閱 Amazon CloudWatch 定價。

將 CloudWatch 指標與 Lambda 搭配使用

當您的 AWS Lambda 函數處理完事件時，Lambda 會將調用的相關指標傳送至 Amazon CloudWatch。您不需要將任何其他許可授予執行角色，即可接收函數指標，而且無需為這些指標額外付費。

與 Lambda 函數關聯的指標有許多種類型，包括調用指標、效能指標、並行指標、非同步調用指標和事件來源映射指標。如需更多詳細資訊，請參閱 the section called “指標類型”。

在 CloudWatch 主控台上，您可以檢視這些指標並使用它們建置圖表和儀表板。也可以設定警示來回應使用率、效能或錯誤率變更。Lambda 每隔 1 分鐘會將指標資料傳送至 CloudWatch。如果想要更快地了解 Lambda 函數，則可以建立高解析度自訂指標。啟用自訂指標和 CloudWatch 警示將會產生費用。如需詳細資訊，請參閱 Amazon CloudWatch 定價。

檢視 Lambda 函數的指標

可以透過 CloudWatch 主控台檢視 Lambda 函數的指標。在主控台中，您可以依函式名稱、別名、版本或事件來源映射 UUID 篩選函式指標，並對其排序。

若要在 CloudWatch 主控台上檢視指標

1. 開啟 CloudWatch 主控台的指標頁面 (AWS/Lambda 命名空間)。
2. 在瀏覽索引標籤的指標下，選擇下列任一維度：
   - 依函數名稱 (FunctionName) - 檢視函數所有版本和別名的彙總指標。
   - 依資源 (Resource) - 檢視函數版本或別名的指標。
   - 執行版本 (ExecutedVersion) - 檢視別名與版本組合的指標。使用 ExecutedVersion 維度來比較兩個函式版本的錯誤率，這兩個版本都是加權別名的目標。
   - 依事件來源映射 UUID (EventSourceMappingUUID) – 檢視事件來源映射的指標。
   - 跨所有函數 (無) - 檢視目前 AWS 區域中所有函數的彙總指標。
3. 選擇指標。指標應該會自動顯示在視覺化圖形中以及圖形化指標索引標籤下方。

根據預設，圖表會針對所有指標使用 Sum 統計資料。若要選擇不同的統計資料並自訂圖表，請使用 圖表化指標 標籤中的選項。

> **Note**
>
> 指標上的時間戳記會反映呼叫函數的時間。根據調用的持續時間，這可能是發出指標前的幾分鐘。例如，如果函數逾時 10 分鐘，請查看過去 10 分鐘之前以獲取精確的指標。

如需有關 CloudWatch 的詳細資訊，請參閱《Amazon CloudWatch 使用者指南》。

Lambda 函數的指標類型

下一節描述 CloudWatch 主控台中可用的 Lambda 指標的類型。

主題

- 呼叫指標
- 部署指標
- 效能指標
- 並行指標
- 非同步調用指標
- 事件來源映射指標

呼叫指標

調用指標是 Lambda 函數調用結果的二進制指標。檢視下列有 Sum 統計資料的指標。例如，若函數傳回錯誤，Lambda 會傳送具有值 1 的 `Errors` 指標。若要取得每分鐘發生的函數錯誤數目，請以一分鐘為其檢視 `Errors` 指標的 Sum。

- `Invocations` – 您的函數程式碼被調用的次數，包含成功調用和造成函數錯誤的調用。如果呼叫請求受到調節，或導致呼叫錯誤，系統就不會記錄呼叫。`Invocations` 的值等於計費的請求數目。
- `Errors` - 導致函數錯誤的呼叫數目。函數錯誤包含程式碼擲回的例外，以及 Lambda 執行時間擲回的例外。執行時間會針對如逾時和組態錯誤等問題傳回錯誤。若要計算錯誤率，將 `Errors` 的值除以 `Invocations` 的值。請注意，錯誤指標上的時間戳記會反映調用函數的時間，而不是錯誤發生的時間。
- `DeadLetterErrors` - 如為非同步調用，則為 Lambda 嘗試傳送事件至無效字母佇列 (DLQ) 但失敗的次數。由於資源或大小限制設定不正確，可能發生無效字母錯誤。
- `DestinationDeliveryFailures`：如為非同步調用和支援的 事件來源映射，則為 Lambda 嘗試傳送事件至 目的地 但失敗的次數。事件來源映射方面，Lambda 支援串流來源 (DynamoDB 和

Kinesis) 的目的地。傳遞錯誤可能因權限錯誤、資源設定不正確或大小限制而發生。如果您設定的目的地是不受支援的類型 (例如 Amazon SQS FIFO 佇列或 Amazon SNS FIFO 主題)，也可能發生此錯誤。

- `Throttles` - 調節的調用請求次數。當所有函數執行個體正在處理請求且沒有並行可供擴展規模時，Lambda 會使用 `TooManyRequestsException` 錯誤拒絕其他請求。限流的請求和其他調用錯誤不會計為 `Invocations` 或 `Errors`。

> **Note**
>
> 使用 Lambda 受管執行個體時，Lambda 會提供精細的限流指標，以識別造成限流的特定限制條件。當執行環境發生調節時，會以 1 的值發出以下其中一個子指標，其餘三個則以 0 的值發出。`Throttles` 指標一律會與這些子指標一起發出。
>
> - `CPUThrottles` – 由於執行環境上的 CPU 耗盡而調節的調用。
> - `MemoryThrottles` – 因執行環境記憶體耗盡而調節的叫用。
> - `DiskThrottles` – 因執行環境上的磁碟耗盡而調節的調用。
> - `ConcurrencyThrottles` – 達到執行環境並行限制時調節的呼叫。

- `OversizedRecordCount`：針對 Amazon DocumentDB 事件來源，您的函數從變更串流接收到大小超過 6 MB 的事件數量。Lambda 會捨棄訊息並發出此指標。
- `ProvisionedConcurrencyInvocations` - 使用佈建並行調用函數程式碼的次數。
- `ProvisionedConcurrencySpilloverInvocations` - 當所有佈建並行都處於使用中狀態時，使用標準並行調用函數程式碼的次數。
- `RecursiveInvocationsDropped` – Lambda 因為偵測到您的函數是無限遞迴迴圈的一部分而停止調用函數的次數。透過追蹤支援的 AWS SDKs 新增的中繼資料，遞迴迴圈偵測會監控在請求鏈中調用函數的次數。如果函數作為請求鏈的一部分被調用超過 16 次，則 Lambda 會放棄下一次調用。如果您停用遞迴迴圈偵測，則不會發出此指標。如需使用此功能的詳細資訊，請參閱「使用 Lambda 遞迴迴圈偵測功能防止無限迴圈」。

部署指標

部署指標提供有關 Lambda 函式部署事件及相關驗證流程的資訊。

- `SignatureValidationErrors` – 當程式碼簽署組態政策設定為 `Warn` 時，程式碼套件部署發生簽署驗證失敗的次數。當有效期、不一致或撤銷狀態檢查失敗，但因 `Warn` 政策設定仍允許部署時，將發出此指標。如需有關程式碼簽署的詳細資訊，請參閱使用程式碼簽署來驗證 Lambda 的程式碼完整性。

效能指標

效能指標提供單一函數調用的效能詳細資料。例如，Duration 指標表示您的函式處理事件時以豪秒計算的時間。若要了解您的函數如何處理事件，請檢視這些有 Average 或 Max 統計資料的指標。

- Duration - 您的函數程式碼處理一個事件時所花費的時間。調用的計費期間是 Duration 四捨五入到最接近毫秒的值。Duration 不包含冷啟動時間。
- PostRuntimeExtensionsDuration - 在函數程式碼完成後，執行時間在執行程式碼進行擴展時所用的累計時間。
- IteratorAge：針對 DynamoDB、Kinesis 和 Amazon DocumentDB 事件來源，則為事件中最後一筆記錄的留存期 (以毫秒為單位)。這個指標用來衡量串流接收記錄到事件來源映射將事件傳送至函數之間的時間。
- OffsetLag – 若為 Apache Kafka 和 Amazon Managed Streaming for Apache Kafka (Amazon MSK) 事件來源，則為寫入主題的最後一筆記錄與函數取用者群組處理的最後一筆記錄之間的偏移量差異。雖然 Kafka 主題會有多個分區，但這個指標可以在主題層級測量偏移延遲。

Duration 也支援百分位數 (p) 統計資料。使用百分位數排除偏差 Average 和 Maximum 統計資料的離群值。例如，p95 統計資料會顯示調用的最大持續時間為 95%，排除最慢的 5%。如需詳細資訊，請參閱和《Amazon CloudWatch 使用者指南》中的百分位數。

並行指標

Lambda 會將並行指標報告為跨函數、版本、別名或 AWS 區域處理事件的執行個體數彙總計數。若要查看您與達到並行限制的接近程度，請使用 Max 統計資料檢視這些指標。

- ConcurrentExecutions - 處理事件的函數執行個體數目。如果此數目達到區域的並行執行配額，或是函數的保留並行上限，Lambda 就會限流額外的調用請求。
- ProvisionedConcurrentExecutions - 使用佈建並行處理事件的函數執行個體數目。針對具有佈建並行之別名或版本的每次調用，Lambda 都會發出目前的計數。如果函式處於非作用中狀態或未收到請求，Lambda 不會發出此指標。
- ProvisionedConcurrencyUtilization - 對於版本或別名，則為 ProvisionedConcurrentExecutions 的值除以設定的佈建並行總量。例如，如果您為函數設定 10 的佈建並行數，且您的 ProvisionedConcurrentExecutions 為 7，則您的 ProvisionedConcurrencyUtilization 為 0.7。

由於此指標基於 ProvisionedConcurrentExecutions，如果函式處於非作用中狀態或未收到請求，Lambda 不會發出此指標。如果將 ProvisionedConcurrencyUtilization 用作 CloudWatch 警示的基礎，需注意此事項。

- UnreservedConcurrentExecutions - 若為區域，則是沒有保留並行的函數所處理的事件數目。
- ClaimedAccountConcurrency – 對於區域，無法用於隨需調用的並行數量。ClaimedAccountConcurrency 等於 UnreservedConcurrentExecutions 加上配置並行的數量 (亦即預留並行總數加上佈建並行總數)。如需詳細資訊，請參閱使用 ClaimedAccountConcurrency 指標。

非同步調用指標

非同步調用指標提供有關來自事件來源和直接調用的非同步調用詳細資料。您可以設定閾值和警示，以便在發生某些變更時通知您。例如排入處理佇列的事件數目意外增加時 (AsyncEventsReceived)。或是某事件已等待處理很長一段時間時 (AsyncEventAge)。

- AsyncEventsReceived – Lambda 成功排入處理佇列的事件數目。此指標可讓您深入了解 Lambda 函數接收的事件數量。監控此指標並設定閾值警示以檢查問題。例如，偵測傳送至 Lambda 的不必要事件數量，並快速診斷因不正確的觸發程序或函數組態所造成的問題。AsyncEventsReceived 和 Invocations 之間的不相符項目可能表示處理差異、捨棄的事件或潛在的待處理佇列。
- AsyncEventAge – Lambda 成功將事件排入佇列到調用函數之間的時間。因調用失敗或限流而重試事件時，此指標的值會增加。監控此指標，並針對發生佇列累積時的不同統計資料設定閾值警示。若要對此指標的增加情形進行疑難排解，請查看 Errors 指標以識別函數錯誤，並查看 Throttles 指標以找出並行問題。
- AsyncEventsDropped – 在未成功執行函數的情況下捨棄的事件數目。如果您有設定無效字母佇列 (DLQ) 或 OnFailure 目的地，則事件會在捨棄之前傳送至該處。有多種原因會導致事件遭捨棄。例如，事件可能超過事件存留期上限、用盡重試次數上限，或是預留並行可能設定為 0。若要對捨棄事件的原因進行疑難排解，請查看 Errors 指標以識別函數錯誤，並查看 Throttles 指標來找出並行問題。

事件來源映射指標

事件來源映射指標可讓您深入了解事件來源映射的處理行為。

目前，事件來源映射指標可用於 Amazon SQS、Kinesis、DynamoDB、Amazon MSK 和自我管理 Apache Kafka 事件來源。

對於具有指標組態的事件來源映射，您也可以立即從主控台 Lambda > 其他資源 > 事件來源映射頁面的監控索引標籤中檢查所有 ESM 相關指標。

若要啟用指標或事件來源映射 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇您要為其啟用指標的函數。
3. 選擇組態，然後選擇觸發條件。
4. 選擇您要為其啟用指標的事件來源映射，然後選擇編輯。
5. 在事件來源映射組態下，選擇啟用指標，或從指標下拉式清單中選取 。
6. 選擇儲存。

您也可以透過使用 EventSourceMappingConfiguration 中的 EventSourceMappingMetricsConfig 物件，以程式設計方式啟用事件來源映射的指標。例如，下列 UpdateEventSourceMapping CLI 命令會啟用事件來源映射的指標：

```
aws lambda update-event-source-mapping \
    --uuid a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 \
    --metrics-config Metrics=EventCount
```

有 3 個指標上線：EventCount、 ErrorCount和 KafkaMetrics，而且每個群組都有多個指標。並非每個指標都適用於每個事件來源。下表摘要說明每種事件來源類型的支援指標。

您必須選擇加入指標群組，才能接收指標相關指標。例如，在指標組態中將 EventCount 設定為具有：(PolledEventCount、FilteredOutEventCount、InvokedEventCount、FailedInvokeEventCour OnFailureDestinationDeliveredEventCount和 DeletedEventCount)。

事件來源映射指標	指標群組	Amazon SQS	Kinesis 和 DynamoDB 串流	Amazon MSK 和自我管理 Apache Kafka
PolledEventCount	EventCount	是	是	是
FilteredOutEventCount	EventCount	是	是	是

事件來源映射指標	指標群組	Amazon SQS	Kinesis 和 DynamoDB 串流	Amazon MSK 和自我管理 Apache Kafka
InvokedEventCount	EventCount	是	是	是
FailedInvokeEventCount	EventCount	是	是	是
DroppedEventCount	EventCount	否	是	是
OnFailureDestinationDeliveredEventCount	EventCount	否	是	是
DeletedEventCount	EventCount	是	否	否
CommittedEventCount	EventCount	否	否	是
PollingErrorCount	ErrorCount	否	否	是
InvokeErrorCount	ErrorCount	否	否	是
OnFailureDestinationDeliveryErrorCount	ErrorCount	否	否	是

事件來源映射指標	指標群組	Amazon SQS	Kinesis 和 DynamoDB 串流	Amazon MSK 和自我管理 Apache Kafka
SchemaRegistryErrorCount	ErrorCount	否	否	是
CommitErrorCount	ErrorCount	否	否	是
MaxOffsetLag	KafkaMetrics	否	否	是
SumOffsetLag	KafkaMetrics	否	否	是

此外，如果您的事件來源映射處於佈建模式，則 Lambda 會提供下列指標：

- ProvisionedPollers – 對於處於佈建模式的事件來源映射，主動執行的事件輪詢器數量。使用MAX數學檢視此指標。
- （僅限 Amazon MSK 和自我管理的 Apache Kafka 事件來源） EventPollerUnit – 對於佈建模式中的事件來源映射，則為主動執行的事件輪詢器單位數量。使用SUM數學檢視此指標。
- (Amazon MSK 和自我管理 Apache Kafka 事件來源） EventPollerThroughputInBytes – 對於佈建模式中的事件來源映射，從事件來源輪詢的事件輪詢器總記錄大小。它可以告訴您目前的輪詢輸送量。使用SUM數學檢視此指標。

以下是每個指標的詳細資訊：

- PolledEventCount – Lambda 從事件來源順利讀取的事件數量。如果 Lambda 輪詢事件，但收到空白輪詢 (沒有新記錄)，則 Lambda 會為此指標發出 0 值。使用此指標可偵測您的事件來源映射是否在正確輪詢新事件。
- FilteredOutEventCount – 對於篩選條件的事件來源映射，該篩選條件所篩選出來的事件數量。使用此指標可偵測您的事件來源映射是否正在適當地篩選出事件。對於符合篩選條件的事件，Lambda 會發出 0 指標。

- `InvokedEventCount` – 調用 Lambda 函數的事件數量。使用此指標可驗證事件是否在正在適當地調用您的函數。如果事件導致函數錯誤或限流，則 `InvokedEventCount` 可能會因為自動重試而多次計算同一輪詢事件。

  > **Warning**
  >
  > Lambda 事件來源映射至少會處理每個事件一次，而且可能會發生重複處理記錄的情況。因此，在涉及事件計數的指標中，可能會將事件多次納入計算。

- `FailedInvokeEventCount` – Lambda 嘗試用來調用您的函數但失敗的事件數量。調用可能失敗的原因包括網路組態問題、許可不正確，或是已刪除 Lambda 函數、版本或別名。如果您的事件來源映射已啟用部分批次回應，則 `FailedInvokeEventCount` 會在回應中包括含非空白 `BatchItemFailures` 的事件。

  > **Note**
  >
  > `FailedInvokeEventCount` 指標的時間戳記表示函數調用結束的時間。此行為與其他 Lambda 調用錯誤指標不同，其他 Lambda 調用錯誤指標是在函數調用開始時加上時間戳記。

- `DroppedEventCount` – Lambda 因過期或重試耗盡而捨棄的事件數量。具體而言，這是超過您為 `MaximumRecordAgeInSeconds` 或 `MaximumRetryAttempts` 所設定值的記錄數量。重要的是，這不包括由於超過事件來源的保留期限設定而過期的記錄數量。捨棄的事件也不包括您傳送至失敗時的目的地的事件。使用此指標可偵測不斷增多的積壓事件。
- `OnFailureDestinationDeliveredEventCount` – 對於已設定失敗時的目的地的事件來源映射，傳送至該目的地的事件數量。使用此指標可監控與此事件來源的調用相關的函數錯誤。如果傳送至目的地失敗，則 Lambda 會以如下方式處理指標：
  - Lambda 不會發出 `OnFailureDestinationDeliveredEventCount` 指標。
  - 對於 `DestinationDeliveryFailures` 指標，Lambda 會發出 1。
  - 對於 `DroppedEventCount` 指標，Lambda 發出的數字等於傳送失敗的事件數量。
- `DeletedEventCount` – Lambda 在處理後成功刪除的事件數。如果 Lambda 嘗試刪除事件但失敗，則 Lambda 會發出 0 指標。使用此指標可確定已從事件來源中刪除順利處理的事件。
- `CommittedEventCount` – Lambda 在處理後成功遞交的事件數量。這是 Kafka 事件來源映射中每個分割區的最後一個和目前遞交位移差異的總和。
- `PollingErrorCount` – Lambda 無法從事件來源輪詢請求的錯誤數目。Lambda 只會在發生錯誤時發出此指標資料。

- `InvokeErrorCount` – Lambda 無法叫用函數的錯誤數目。請注意，調用是批次記錄。數字位於批次層級，而非記錄計數層級。Lambda 只會在發生錯誤時發出此指標資料。
- `SchemaRegistryErrorCount` – Lambda 無法擷取結構描述或使用結構描述還原序列化的錯誤數目。Lambda 只會在發生錯誤時發出此指標資料。
- `CommitErrorCount` – Lambda 無法遞交至 Kafka 叢集的錯誤數目。Lambda 只會在發生錯誤時發出此指標資料。
- `MaxOffsetLag` – 事件來源映射中所有分割區的偏移延遲上限（最新和遞交偏移之間的差異）。
- `SumOffsetLag` – 事件來源映射中所有分割區的位移延遲總和。

如果您的事件來源映射已停用，則您不會收到事件來源映射指標。如果 CloudWatch 或 Lambda 的可用性下降，則您還可能會看到缺少指標。

使用 Lambda 函式日誌

為了協助您故障診斷故障， 會代表您 AWS Lambda 自動監控 Lambda 函數。您可以使用 Lambda 主控台、CloudWatch 主控台、 AWS Command Line Interface (AWS CLI)、CloudWatch API 來檢視 Lambda 函數的日誌。您也可以設定 Lambda 將日誌傳送至 Amazon S3 與 Firehose。

只要函式的執行角色具有必要許可，Lambda 就會擷取函式處理的所有請求的日誌，並將日誌傳送至預設目的地 Amazon CloudWatch Logs。您也可以透過 Lambda 主控台，將 Amazon S3 或 Firehose 設定為日誌記錄目的地。

- CloudWatch Logs 是 Lambda 函式預設的日誌記錄目的地。CloudWatch Logs 具有即時日誌檢視與分析功能，且支援根據日誌資料建立指標與警示。
- Amazon S3 在長期儲存方面具有經濟效益，且可透過 Athena 等服務對日誌進行分析。延遲通常較高。
- Firehose 提供受管日誌串流功能，可將日誌串流至多種目的地。如果您需要將日誌傳送至 AWS 其他服務 （例如 OpenSearch Service 或 Redshift Data API) 或第三方平台 （例如 Datadog、New Relic 或 Splunk)，Firehose 會透過提供預先建置的整合來簡化該程序。您也可以將日誌串流至自訂 HTTP 端點，無需額外設定基礎結構。

選擇用於傳送日誌的服務目的地

在選擇要用作函式日誌的服務目的地時，建立考量下列關鍵因素：

- 成本管理因服務而異。Amazon S3 通常是最經濟實惠的長期儲存選擇，而 CloudWatch Logs 支援即時檢視日誌、處理日誌和設定提醒。Firehose 成本包括串流服務費用以及與所設定串流目的地相關的費用。
- 分析功能因服務而異。CloudWatch Logs 在即時監控方面表現出色，能夠與其他 CloudWatch 功能 (例如 Logs Insights 和 Live Tail) 原生整合。Amazon S3 與 Athena 等分析工具搭配運作良好，並且可以與各種服務整合，但可能需要額外的設定。Firehose 透過提供預先建置的整合，簡化直接串流至特定 AWS 服務 （例如 OpenSearch Service 和 Redshift Data API) 和支援的第三方平台 （例如 Datadog 和 Splunk)，進而減少組態工作。
- 設定程序與易用性因服務而異。CloudWatch Logs 是預設的日誌目的地，無需額外設定即可立即運作，支援透過 CloudWatch 主控台直觀地檢視與分析日誌。若需將日誌傳送至 Amazon S3，應先在 Lambda 主控台中執行部分初始設定，並設定儲存貯體許可。若需將日誌直接傳送至 OpenSearch Service 等服務或第三方分析平台，Firehose 可以簡化該流程。

設定日誌目的地

AWS Lambda 支援函數日誌的多個目的地。本指南介紹了可用的日誌記錄目的地，能協助您視需求做出正確選擇。無論選擇哪種目的地，Lambda 皆能提供用於控制日誌格式、篩選條件與傳送方式的相關選項。

Lambda 支援 JSON 與純文字格式的函式日誌。JSON 結構化日誌能夠提升可搜尋性，支援自動化分析；而純文字日誌較為簡潔，可能降低儲存成本。您可以透過設定系統與應用程式日誌的日誌層級，控制 Lambda 傳送至所選目的地的日誌。篩選功能可協助您管理儲存成本，讓您在偵錯期間更輕鬆地找到相關的日誌項目。

如需每個目的地的詳細設定說明，請參閱下列各節：

- 將 Lambda 函式日誌傳送至 CloudWatch Logs
- 將 Lambda 函式日誌傳送至 Firehose
- 將 Lambda 函式日誌傳送至 Amazon S3

設定 Lambda 函數的進階日誌記錄控制項

為了讓您更全面地掌控函式日誌的擷取、處理與使用方式，Lambda 提供了下列日誌記錄組態選項：

- 日誌格式 – 在純文字和結構化 JSON 格式之間為函式日誌選擇一種格式。
- 日誌層級 – 對於 JSON 結構化日誌，請選擇 Lambda 要傳送至 CloudWatch 之日誌的詳細層級，例如 `FATAL`、`ERROR`、`WARN`、`INFO`、`DEBUG` 和 `TRACE`。
- 日誌群組 – 選擇函式要將日誌傳送到的 CloudWatch 日誌群組。

若要進一步了解如何設定進階日誌記錄控制項，請參閱下列章節：

- 設定 JSON 和純文字日誌格式
- 日誌層級篩選
- 設定 CloudWatch 日誌群組

設定 JSON 和純文字日誌格式

將日誌輸出擷取為 JSON 索引鍵值組可以在偵錯函數時更容易搜尋和篩選。使用 JSON 格式的日誌檔，您也可以在日誌檔中新增標籤和內容資訊。這可以幫助您對大量日誌資料執行自動分析。除非您的開發工作流程仰賴使用純文字 Lambda 日誌的現有工具，否則建議您為日誌格式選取 JSON。

> **Lambda 受管執行個體**
>
> Lambda 受管執行個體僅支援 JSON 日誌格式。當您建立受管執行個體函數時，Lambda 會自動將日誌格式設定為 JSON，而且您無法將其變更為純文字。如需受管執行個體的詳細資訊，請參閱 Lambda 受管執行個體。

對於所有 Lambda 受管執行期，您可以選擇是以非結構化純文字還是 JSON 格式將函數的系統日誌傳送至 CloudWatch Logs 日誌。系統日誌是 Lambda 產生的日誌，有時也稱為平台事件日誌。

對於支援的執行期，當您使用其中一種支援的內建記錄方法時，Lambda 也能以結構化 JSON 格式輸出函數的應用程式日誌 (函數程式碼產生的日誌)。當您針對這些執行期設定函數的日誌格式時，您選擇的組態會同時套用至系統和應用程式日誌。

對於支援的執行期，如果您的函數使用支援的記錄程式庫或方法，則不需要變更 Lambda 現有程式碼，即可擷取結構化 JSON 中的記錄。

> **Note**
>
> 使用 JSON 日誌格式新增其他中繼資料，並將日誌訊息編碼為包含一系列索引鍵值組的 JSON 物件。因此，函數日誌訊息的大小可能會增加。

支援的執行期和記錄方法

Lambda 目前支援針對下列執行期輸出 JSON 結構化應用程式日誌的選項。

Language	支援的版本
Java	除了 Amazon Linux 1 上的 Java 8 以外的所有 Java 執行期
.NET	.NET 8 及更新版本
Node.js	Node.js 16 及更高版本
Python	Python 3.8 及更高版本
Rust	N/A

若要讓 Lambda 以結構化 JSON 格式將函數的應用程式日誌傳送至 CloudWatch，您的函數必須使用下列內建記錄工具來輸出日誌檔：

- Java：`LambdaLogger`記錄器或 Log4j2。如需詳細資訊，請參閱the section called “日誌”。
- .NET：內容物件上的`ILambdaLogger`執行個體。如需詳細資訊，請參閱記錄和監控 C# Lambda 函數。
- Node.js - 主控台方法 `console.trace`、`console.debug`、`console.log`、`console.info`、`console.error`、 和 `console.warn`。如需詳細資訊，請參閱the section called “日誌”。
- Python：標準 Python `logging`程式庫。如需詳細資訊，請參閱the section called “日誌”。
- Rust：`tracing`木箱。如需詳細資訊，請參閱the section called “日誌”。

對於其他受管的 Lambda 執行期，Lambda 目前僅支援以結構化 JSON 格式擷取系統記錄。不過，您仍然可以使用 Powertools 等記錄工具來擷取任何執行時間中結構式 JSON 格式的應用程式日誌，用於 AWS Lambda 輸出 JSON 格式的日誌輸出。

預設日誌格式

目前，所有 Lambda 執行期的預設日誌格式都是純文字。對於 Lambda 受管執行個體，日誌格式一律為 JSON，且無法變更。

如果您已經使用 Powertools for 之類的日誌程式庫 AWS Lambda ，以 JSON 結構化格式產生函數日誌，則如果您選取 JSON 日誌格式，則不需要變更程式碼。Lambda 不會對任何已經進行 JSON 編碼的日誌進行雙重編碼，因此您函數的應用程式日誌將繼續像以前一樣被擷取。

系統日誌的 JSON 格式

當您將函數的日誌檔格式設定為 JSON 時，每個系統日誌項目 (平台事件) 都會擷取為 JSON 物件，其中包含與下列索引鍵配對的索引鍵值：

- `"time"` - 產生日誌訊息的時間
- `"type"` - 記錄的事件類型
- `"record"` - 日誌輸出的內容

`"record"` 值的格式會根據日誌的事件類型而有所不同。如需更多資訊，請參閱the section called “遙測 API `Event` 物件類型”。如需有關指派給系統日誌事件的日誌層級的詳細資訊，請參閱 the section called “系統日誌層級事件映射”。

為了進行比較，下列兩個範例會以純文字和結構化 JSON 格式顯示相同的記錄輸出。請注意，在大多數情況下，系統日誌事件在 JSON 格式輸出時所包含的資訊會比以純文字輸出時更多。

Example純文字：

```
2024-03-13 18:56:24.046000 fbe8c1   INIT_START  Runtime Version:
 python:3.12.v18  Runtime Version ARN: arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0
```

Example結構化的 JSON：

```
{
  "time": "2024-03-13T18:56:24.046Z",
  "type": "platform.initStart",
  "record": {
    "initializationType": "on-demand",
    "phase": "init",
    "runtimeVersion": "python:3.12.v18",
    "runtimeVersionArn": "arn:aws:lambda:eu-
west-1::runtime:edb5a058bfa782cb9cedc6d534ac8b8c193bc28e9a9879d9f5ebaaf619cd0fc0"
  }
}
```

> **Note**
>
> the section called “遙測 API” 始終以 JSON 格式發出平台事件，例如 START 和 REPORT。設定 Lambda 傳送至 CloudWatch 的系統日誌格式不會影響 Lambda 遙測 API 行為。

應用程式日誌的 JSON 格式

當您將函數的日誌格式設定為 JSON 時，使用支援的記錄程式庫和方法撰寫的應用程式日誌輸出會擷取為 JSON 物件，其中包含具有以下索引鍵的鍵值對。

- `"timestamp"` - 產生日誌訊息的時間
- `"level"` - 指派給訊息的日誌層級
- `"message"` - 日誌訊息的內容
- `"requestId"` (Python、.NET 和 Node.js) 或 `"AWSrequestId"` (Java) - 函數調用的唯一請求 ID

依據您的函數使用的執行期和記錄方法，此 JSON 物件還可能包含其他鍵值對。例如，在 Node.js 中，如果您的函數使用 `console` 方法來記錄使用多個引數的錯誤物件，JSON 物件將包含具有索引鍵 `errorMessage`、`errorType` 和 `stackTrace` 的額外鍵值對。若要進一步了解不同 Lambda 執行期中的 JSON 格式的日誌，請參閱 the section called “日誌”、the section called “日誌” 和 the section called “日誌”。

> **Note**
>
> Lambda 用於時間戳記值的關鍵對於系統日誌和應用程式日誌而言不同。對於系統記錄，Lambda 會使用索引鍵 `"time"` 來維持遙測 API 的一致性。對於應用程式日誌，Lambda 會遵循支援的執行期和使用 `"timestamp"` 的慣例。

為了進行比較，下列兩個範例會以純文字和結構化 JSON 格式顯示相同的記錄輸出。

Example純文字：

```
2024-10-27T19:17:45.586Z 79b4f56e-95b1-4643-9700-2807f4e68189 INFO some log message
```

Example結構化的 JSON：

```
{
    "timestamp":"2024-10-27T19:17:45.586Z",
    "level":"INFO",
    "message":"some log message",
    "requestId":"79b4f56e-95b1-4643-9700-2807f4e68189"
}
```

設定函數的日誌格式

若要設定函數的日誌格式，您可以使用 Lambda 主控台或 AWS Command Line Interface (AWS CLI)。您也可以使用 CreateFunction 和 UpdateFunctionConfiguration Lambda API 命令、 AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function 資源和 CloudFormation AWS::Lambda::Function 資源來設定函數的日誌格式。

變更函數的日誌格式不會影響儲存在 CloudWatch Logs 日誌中的現有記錄。只有新的日誌檔會使用更新的格式。

如果您將函數的日誌格式變更為 JSON 但未設定日誌層級，Lambda 會自動將函數的應用程序日誌層級和系統日誌層級設為 INFO。這意味著 Lambda 只會傳送 INFO 及更低層級的日誌輸出至

CloudWatch Logs。若要進一步了解應用程式和系統日誌層級篩選，請參閱 the section called “日誌層級篩選”

> **Note**
>
> 對於 Python 執行期，當函數的日誌格式設定為純文字時，預設的日誌層級設定為 WARN。這意味著 Lambda 只會傳送 WARN 及更低層級的日誌輸出至 CloudWatch Logs。將函數的日誌格式變更為 JSON 會改變此預設行為。若要進一步了解以 Python 記錄日誌，請參閱 the section called “日誌”。

對於發出內嵌指標格式 (EMF) 日誌的 Node.js 函數，將函數的日誌格式變更為 JSON 可能會導致 CloudWatch 無法識別您的指標。

> **Important**
>
> 如果您的函數使用 Powertools for AWS Lambda (TypeScript) 或開放原始碼 EMF 用戶端程式庫來發出 EMF 日誌，請將 Powertools 和 EMF 程式庫更新為最新版本，以確保 CloudWatch 可以繼續正確剖析您的日誌。如果您切換到 JSON 日誌格式，我們也建議您進行測試，以確保與函數的內嵌指標相容。如需有關發出 EMF 日誌檔之 node.js 函數的進一步建議，請參閱 the section called “搭配結構化 JSON 日誌使用內嵌指標格式 (EMF) 用戶端程式庫”。

若要設定函數的日誌格式 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇一個函數。
3. 在函數組態頁面上，選擇監視和操作工具。
4. 在日誌組態窗格中，選擇編輯。
5. 在日誌內容之下，針對日誌檔格式選取文字或 JSON。
6. 選擇儲存。

若要變更現有函數的日誌格式 (AWS CLI)

- 若要變更現有函數的日誌格式，請使用 update-function-configuration 命令。將 `LoggingConfig` 中 `LogFormat` 選項設定為 `JSON` 或 `Text`。

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogFormat=JSON
```

若要在建立函數 (AWS CLI) 時設定記錄格式

- 若要在建立新函數時設定日誌格式，請使用 create-function 命令中的 `--logging-config` 選項。將 `LogFormat` 設定為 `JSON` 或 `Text`。下列範例命令會建立一個 Node.js 函數，以結構化 JSON 輸出日誌。

  如果您在建立函數時未指定日誌格式，Lambda 會針對您選取的執行期版本使用預設日誌格式。如需有關預設記錄格式的資訊，請參閱 the section called “預設日誌格式”。

  ```
  aws lambda create-function \
    --function-name myFunction \
    --runtime nodejs24.x \
    --handler index.handler \
    --zip-file fileb://function.zip \
    --role arn:aws:iam::123456789012:role/LambdaRole \
    --logging-config LogFormat=JSON
  ```

日誌層級篩選

Lambda 可以篩選函數的日誌，以便僅將特定詳細資料層級或更低層級的日誌傳送到 CloudWatch 日誌。您可以針對函數的系統日誌 (Lambda 產生的日誌) 和應用程式日誌 (函數程式碼產生的日誌) 分別設定日誌層級篩選。

對於 the section called “支援的執行期和記錄方法”，您無需對函數程式碼進行任何變更，Lambda 即可篩選函數的應用程式日誌。

對於所有其他執行期和記錄方法，您的函數程式碼必須將日誌事件輸出至 `stdout` 或 `stderr` 作為 JSON 格式的物件，其中包含與索引鍵 `"level"` 配對的索引鍵值)。例如，Lambda 會將下列輸出解譯為 `stdout` 作為 DEBUG 層級日誌檔。

```
print('{"level": "debug", "msg": "my debug log", "timestamp":
 "2024-11-02T16:51:31.587199Z"}')
```

如果 "level" 值欄位無效或遺失，Lambda 會將日誌輸出指派層級 INFO。若要讓 Lambda 使用時間戳記欄位，您必須以有效的 RFC 3339 時間戳記格式指定時間。如果您沒有提供有效的時間戳記，Lambda 會為日誌指派層級 INFO，並為您新增時間戳記。

命名時間戳記索引鍵時，請遵循您使用的執行期慣例。Lambda 支援受管理執行期使用的大多數通用命名慣例。

> **Note**
>
> 若要使用日誌層級篩選，您的函數必須設定為使用 JSON 記錄格式。所有 Lambda 受管執行期的預設日誌格式目前都是純文字。若要瞭解如何將函數的日誌格式設定為 JSON，請參閱 the section called “設定函數的日誌格式”。

對於應用程式日誌 (由函數程式碼生成的日誌)，您可以在以下日誌層級之間進行選擇。

日誌層級	標準用量
TRACE (大多數詳細資訊)	用於追蹤程式碼執行路徑的最精細資訊
DEBUG	系統偵錯的詳細資訊
INFO	記錄函數正常操作的訊息
WARN	有關可能導致未解決意外行為的潛在錯誤的消息
ERROR	有關阻止程式碼按預期執行的問題的訊息
FATAL (最少詳細資訊)	有關導致應用程式停止運作的嚴重錯誤訊息

當您選取日誌層級時，Lambda 會將該層級或更低層級的日誌傳送至 CloudWatch Logs 日誌。例如，如果您將函數的應用程式日誌層級設定為 WARN，Lambda 就不會在 INFO 和 DEBUG 層級傳送日誌輸出。日誌篩選的預設應用程式日誌層級為 INFO。

當 Lambda 篩選函數的應用程式日誌時，沒有層級的日誌訊息將被指派日誌等級 INFO。

對於系統日誌檔 (Lambda 服務產生的日誌檔)，您可以在下列日誌層級進行選擇。

日誌層級	Usage
DEBUG (大多數詳細資訊)	系統偵錯的詳細資訊
INFO	記錄函數正常操作的訊息
WARN (最少詳細資訊)	有關可能導致未解決意外行為的潛在錯誤的消息

當您選取日誌層級時，Lambda 會在該層級 (含) 或更低層級傳送日誌。例如，如果您將函數的系統日誌層級設定為 INFO，Lambda 不會在 DEBUG 層級傳送日誌輸出。

根據預設，Lambda 會將系統日誌層級設定為 INFO。透過此設定，Lambda 會自動將 `"start"` 和 `"report"` 日誌訊息傳送到 CloudWatch。若要接收更多或更少詳細的系統日誌，請將日誌層級變更為 DEBUG 或 WARN。若要查看 Lambda 映射不同系統日誌事件的記錄層級清單，請參閱 the section called “系統日誌層級事件映射”。

設定日誌層級篩選

若要為您的函數設定應用程式和系統日誌層級篩選，您可以使用 Lambda 主控台或 AWS Command Line Interface (AWS CLI)。您也可以使用 CreateFunction 和 UpdateFunctionConfiguration Lambda API 命令、 AWS Serverless Application Model (AWS SAM) AWS::Serverless::Function 資源和 CloudFormation AWS::Lambda::Function 資源來設定函數的日誌層級。

請注意，如果您在程式碼中設定函數的日誌層級，此設定會優先於您的任何其他日誌層級設定。例如，如果您使用 Python `logging` `setLevel()` 方法將函數的記錄層級設定為 INFO，則此設定的優先級將高於您使用 Lambda 主控台設定的 WARN 層級。

若要設定現有函數的應用程式或系統日誌層級 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在函數組態頁面上，選擇監視和操作工具。
4. 在日誌組態窗格中，選擇編輯。
5. 在日誌內容之下，針對日誌檔格式，確保已選取 JSON
6. 使用選項按鈕，為您的函數選擇所需的應用程式日誌層級和系統日誌層級。
7. 選擇儲存。

若要設定現有函數的應用程式或系統日誌層級 (AWS CLI)

- 若要變更現有函數的應用程式或系統日誌層級，請使用 update-function-configuration 命令。使用 SystemLogLevel 將 --logging-config 設定為 DEBUG、INFO 或 WARN 中的其中一項。設定 ApplicationLogLevel 為 DEBUG、INFO、WARN、ERROR 或 FATAL 之一。

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogFormat=JSON,ApplicationLogLevel=ERROR,SystemLogLevel=WARN
```

若要在建立函數時設定日誌層級篩選

- 若要在建立新函數時設定日誌層級篩選，請使用 create-function 命令將 --logging-config 設定為 SystemLogLevel 和 ApplicationLogLevel 金鑰。設定 SystemLogLevel 為 DEBUG、INFO 或 WARN 之一。設定 ApplicationLogLevel 為 DEBUG、INFO、WARN、ERROR 或 FATAL 之一。

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole \
  --logging-config LogFormat=JSON,ApplicationLogLevel=ERROR,SystemLogLevel=WARN
```

系統日誌層級事件映射

對於 Lambda 產生的系統層級日誌事件，以下資料表定義指派給每個事件的日誌層級。若要進一步瞭解資料表中所列事件，請參閱 the section called “Event 結構描述參考”

事件名稱	條件	指派的日誌層級
initStart	runtimeVersion已設定	INFO
initStart	runtimeVersion未設定	DEBUG
initRuntimeDone	status=success	DEBUG

事件名稱	條件	指派的日誌層級
initRuntimeDone	status!=success	WARN
initReport	initializationType!=on-demand	INFO
initReport	initializationType=on-demand	DEBUG
initReport	status!=success	WARN
restoreStart	runtimeVersion已設定	INFO
restoreStart	runtimeVersion未設定	DEBUG
restoreRuntimeDone	status=success	DEBUG
restoreRuntimeDone	status!=success	WARN
restoreReport	status=success	INFO
restoreReport	status!=success	WARN
入門	-	INFO
runtimeDone	status=success	DEBUG
runtimeDone	status!=success	WARN
報告	status=success	INFO
報告	status!=success	WARN
副檔名	state=success	INFO
副檔名	state!=success	WARN
logSubscription	-	INFO
telemetrySubscription	-	INFO
logsDropped	-	WARN

> **Note**
>
> the section called “遙測 API” 始終發出一組完整的平台事件。設定 Lambda 傳送至 CloudWatch 的系統日誌層級不會影響 Lambda 遙測 API 行為。

使用自訂執行期的應用程式日誌層級篩選

當您為函數設定應用程式日誌層級篩選時，Lambda 會在幕後使用 `AWS_LAMBDA_LOG_LEVEL` 環境變數在執行期設定應用程式日誌層級。Lambda 也會使用 `AWS_LAMBDA_LOG_FORMAT` 環境變數來設定函數的日誌格式。您可以使用這些變數，將 Lambda 進階日誌控制項整合至自訂執行期。

若要透過 Lambda 主控台 AWS CLI和 Lambda APIs 使用自訂執行期來設定函數的記錄設定，請設定您的自訂執行期來檢查這些環境變數的值。然後，您可以根據您選取的日誌格式和日誌層級來設定執行期的日誌程式。

將 Lambda 函式日誌傳送至 CloudWatch Logs

依預設，如果函式的執行角色具有必要許可，Lambda 會自動擷取所有函式調用的日誌，並將日誌傳送至 CloudWatch Logs。這些日誌預設存儲在名為 /aws/lambda/*<function-name>* 的日誌群組中。若要增強偵錯功能，您可以將自訂日誌記錄陳述式插入程式碼。如此，Lambda 將與 CloudWatch Logs 無縫整合。您可以視需要使用 Lambda 主控台、AWS CLI 或 Lambda API 設定函式將日誌傳送給其他群組。如需進一步了解，請參閱the section called “設定 CloudWatch 函式日誌”。

您可以使用 Lambda 主控台、CloudWatch 主控台、AWS Command Line Interface (AWS CLI) 或 CloudWatch API 來檢視 Lambda 函數的日誌。如需詳細資訊，請參閱檢視 Lambda 函數的 CloudWatch 日誌。

> **Note**
>
> 在函數調用後，日誌可能需要 5 到 10 分鐘才會顯示。

所需的 IAM 許可

您的執行角色需要以下許可，才能將日誌上傳至 CloudWatch Logs。

- `logs:CreateLogGroup`
- `logs:CreateLogStream`

- `logs:PutLogEvents`

如需詳細資訊，請參閱《Amazon CloudWatch 使用者指南》中的 Using identity-based policies (IAM policies) for CloudWatch Logs。

可以使用 Lambda 提供的 `AWSLambdaBasicExecutionRole` AWS 受管政策來新增 CloudWatch Logs 許可。執行以下命令，將此政策新增至您的角色：

```
aws iam attach-role-policy --role-name your-role --policy-arn arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole
```

如需更多詳細資訊，請參閱 the section called “AWS 受管政策”。

定價

使用 Lambda 日誌無需支付額外費用；但仍需支付標準 CloudWatch Logs 費用。如需詳細資訊，請參閱 CloudWatch 定價。

設定 CloudWatch 日誌群組

根據預設，CloudWatch 會在第一次調用時自動為您的函數建立名為 `/aws/lambda/<function name>` 的日誌群組。若要將函數設定為將日誌傳送到現有的日誌群組，或為您的函數建立新的日誌群組，您可以使用 Lambda 主控台或 AWS CLI. 您也可以使用 CreateFunction 和 UpdateFunctionConfiguration Lambda API 命令和 AWS Serverless Application Model (AWS SAM) AWS：：Serverless：：Function 資源來設定自訂日誌群組。

您可以設定多個 Lambda 函數，將日誌傳送至同一個 CloudWatch 日誌群組。例如，您可以使用單一日誌群組來儲存組成特定應用程式之所有 Lambda 函數的記錄。當您針對 Lambda 函數使用自訂日誌群組時，Lambda 建立的日誌串流會包含函數名稱和函數版本。如此可確保日誌訊息和函數之間的映射會被保留，即使您對多個函數使用相同的日誌群組也是如此。

自訂日誌群組的日誌串流命名格式遵循下列慣例：

```
YYYY/MM/DD/<function_name>[<function_version>][<execution_environment_GUID>]
```

請注意，設定自訂日誌群組時，您為日誌群組選取的名稱必須遵循 CloudWatch Logs 命名規則。此外，自訂日誌群組名稱不得以字串 `aws/` 開頭。如果您以 `aws/` 開頭建立自訂日誌群組，Lambda 將無法建立日誌群組。因此，您的函數的日誌將不會傳送到 CloudWatch。

若要變更函數的日誌群組 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 在函數組態頁面上，選擇監視和操作工具。
4. 在日誌組態窗格中，選擇編輯。
5. 在日誌群組窗格中，對於 CloudWatch 日誌群組，選擇自訂。
6. 在自訂日誌群組之下，輸入要將函數傳送日誌前往的 CloudWatch 日誌群組的名稱。如果您輸入現有日誌群組的名稱，則您的函數將使用該群組。如果沒有具有您輸入名稱的日誌群組，則 Lambda 會以該名稱為您的函數建立新的日誌群組。

若要變更函數的日誌群組 (AWS CLI)

- 若要變更現有函數的日誌群組，請使用 update-function-configuration 命令。

```
aws lambda update-function-configuration \
  --function-name myFunction \
  --logging-config LogGroup=myLogGroup
```

若要在建立函數 (AWS CLI) 時指定自訂日誌群組

- 若要在使用 建立新的 Lambda 函數時指定自訂日誌群組 AWS CLI，請使用 `--logging-config`選項。下列範例命令會建立 Node.js Lambda 函數，該函數會將日誌檔傳送至名為 `myLogGroup` 的日誌群組。

```
aws lambda create-function \
  --function-name myFunction \
  --runtime nodejs24.x \
  --handler index.handler \
  --zip-file fileb://function.zip \
  --role arn:aws:iam::123456789012:role/LambdaRole \
  --logging-config LogGroup=myLogGroup
```

執行角色許可

為了讓您的函數將日誌傳送到 CloudWatch Logs，其必須具有 logs:PutLogEvents 許可。在您透過 Lambda 主控台設定函式的日誌群組時，Lambda 會在下列條件下將此許可新增至角色：

- 服務目的地設定為 CloudWatch Logs
- 函式的執行角色缺少將日誌上傳至 CloudWatch Logs (預設目的地) 的許可

> **Note**
>
> Lambda 不會針對 Amazon S3 或 Firehose 日誌目的地新增任何 Put 許可。

當 Lambda 新增此許可時，它會授予將日誌傳送至任何 CloudWatch Logs 日誌群組的函數許可。

若要防止 Lambda 自動更新函數的執行角色並改為手動編輯，請展開許可，然後取消勾選新增所需許可。

當您使用 設定函數的日誌群組時 AWS CLI，Lambda 不會自動新增`logs:PutLogEvents`許可。如果函數的執行角色尚不具備許可，請將其新增至函數的執行角色。這些許可包含在 AWSLambdaBasicExecutionRole 受管政策中。

Lambda 受管執行個體的 CloudWatch 記錄

使用 Lambda 受管執行個體時，將日誌傳送至 CloudWatch Logs 還有其他考量：

VPC 聯網需求

Lambda 受管執行個體會在 VPC 中客戶擁有的 EC2 執行個體上執行。若要將日誌傳送至 CloudWatch Logs，並將追蹤傳送至 X-Ray，您必須確保這些 AWS APIs 可從 VPC 路由。您有多種選擇：

- AWS PrivateLink （建議）：使用 AWS PrivateLink 為 CloudWatch Logs 和 X-Ray 服務建立 VPC 端點。這可讓您的執行個體私密存取這些服務，而不需要網際網路閘道或 NAT 閘道。如需詳細資訊，請參閱搭配介面 VPC 端點使用 CloudWatch Logs。
- NAT 閘道：設定 NAT 閘道以允許從私有子網路進行傳出網際網路存取。
- 網際網路閘道：對於公有子網路，請確定您的 VPC 已設定網際網路閘道。

如果 CloudWatch Logs 或 X-Ray APIs 無法從您的 VPC 路由，您的函數日誌和追蹤將不會交付。

並行調用和日誌歸因

Lambda 受管執行個體執行環境可以同時處理多個調用。當多個調用同時執行時，其日誌項目會交錯在相同的日誌串流中。若要有效篩選和分析來自並行調用的日誌，您應該確保每個日誌項目都包含 AWS 請求 ID。

我們建議採用下列其中一種方法：

- 使用預設 Lambda 執行期記錄器（建議）：Lambda 受管執行期提供的預設記錄程式庫會自動在每個日誌項目中包含請求 ID。
- 實作結構化 JSON 記錄：如果您要建置自訂執行時間或需要自訂記錄，請實作 JSON 格式的日誌，其中包含每個項目中的請求 ID。Lambda 受管執行個體僅支援 JSON 日誌格式。在 JSON 日誌中包含 `requestId` 欄位，以透過調用啟用篩選：

```
{
  "timestamp": "2025-01-15T10:30:00.000Z",
  "level": "INFO",
  "requestId": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "message": "Processing request"
}
```

透過請求 ID 屬性，您可以使用 CloudWatch Logs Insights 查詢來篩選特定調用的 CloudWatch Logs 日誌項目。例如：

```
fields @timestamp, @message
| filter requestId = "a1b2c3d4-e5f6-7890-abcd-ef1234567890"
| sort @timestamp asc
```

如需 Lambda 受管執行個體記錄需求的詳細資訊，請參閱 了解 Lambda 受管執行個體執行環境。

檢視 Lambda 函數的 CloudWatch 日誌

您可以使用 Lambda 主控台、Amazon CloudWatch logs AWS CLI。 CloudWatch AWS Command Line Interface 請遵循下列各章節中的說明以存取函數的日誌。

使用 CloudWatch Logs Live Tail 串流函數日誌

Amazon CloudWatch Logs Live Tail 可直接在 Lambda 主控台中顯示新日誌事件的串流清單，協助您快速對函數進行疑難排解。可以透過 Lambda 函數即時地檢視和篩選擷取的日誌，快速偵測並解決問題。

> **Note**
>
> Live Tail 工作階段會按工作階段使用時間 (每分鐘) 產生費用。如需定價的詳細資訊，請參閱 Amazon CloudWatch 定價。

比較 Live Tail 與 --log-type Tail

CloudWatch Logs Live Tail 與 Lambda API 中的 LogType： Tail 選項 (`--log-type Tail` 中的) 之間有幾個差異 AWS CLI：

- `--log-type Tail` 只會傳回調用日誌的前 4 KB。Live Tail 不會共用此限制，每秒最多可接收 500 個日誌事件。
- `--log-type Tail` 會擷取並傳送含回應的日誌，這可能會對函數的回應延遲造成影響。Live Tail 不會對函數回應延遲造成影響。
- `--log-type Tail` 僅支援同步調用。Live Tail 對同步和非同步調用皆適用。

> **Note**
>
> Lambda 受管執行個體不支援 `--log-type Tail`選項。使用 CloudWatch Logs Live Tail 或直接查詢 CloudWatch Logs 來檢視受管執行個體函數的日誌。

許可

啟動和停止 CloudWatch Logs Live Tail 工作階段需要以下許可：

- `logs:DescribeLogGroups`
- `logs:StartLiveTail`
- `logs:StopLiveTail`

在 Lambda 主控台中啟動 Live Tail 工作階段

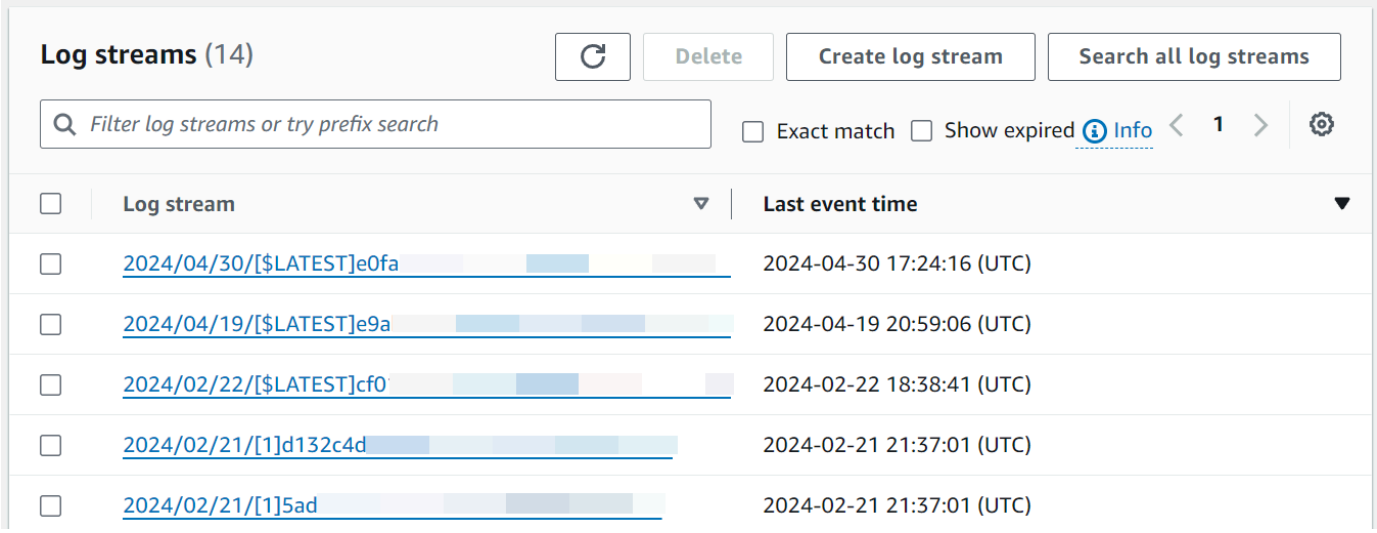
1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函數的名稱。
3. 選擇測試標籤。
4. 在測試事件窗格中，選擇 CloudWatch Logs Live Tail。
5. 對於選取日誌群組，預設選取函數的日誌群組。一次最多可以選取五個日誌群組。
6. (選用) 若要僅顯示包含特定字詞或其他字串的日誌事件，請在新增篩選條件模式方塊中輸入字詞或字串。篩選條件欄位區分大小寫。可以在此欄位中包含多個詞彙和模式運算子，包括常規表達式 (regex)。如需詳細資訊，請參閱《Amazon CloudWatch Logs 使用者指南》中的篩選條件模式語法。

7. 選擇 開始使用。相符的日誌事件開始出現在視窗中。
8. 若要停止 Live Tail 工作階段，請選擇停止。

> **Note**
>
> Live Tail 工作階段會在閒置 15 分鐘後或者 Lambda 主控台工作階段逾時時自動停止。

使用主控台存取函數日誌

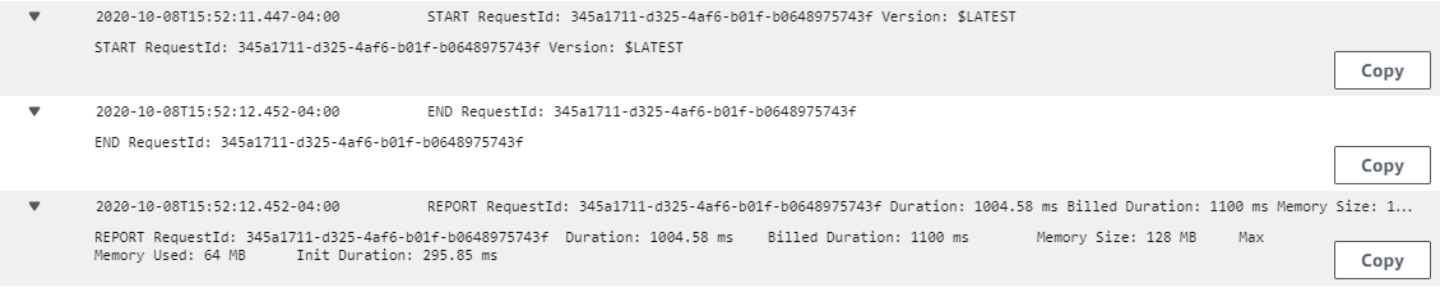
1. 開啟 Lambda 主控台中的函數頁面。
2. 選取函數。
3. 選擇 Monitor (監控) 索引標籤。
4. 選擇檢視 CloudWatch 日誌以打開 CloudWatch 主控台。
5. 向下捲動並選擇要查看的函數調用日誌串流。



Lambda 函數的每個執行個體都具有專用的日誌串流。如果函數向上擴展，則每個並行執行個體都具有自己的日誌串流。每次建立新的執行環境來回應調用，皆會產生新的日誌串流。日誌串流的命名慣例如下：

```
YYYY/MM/DD[Function version][Execution environment GUID]
```

單一執行環境會在生命週期內寫入相同的日誌串流。日誌串流包含來自該執行環境的訊息，以及來自 Lambda 函數程式碼的任何輸出。每則訊息都會加上時間戳記，包括自訂日誌。即使您的函數未記錄來自程式碼的任何輸出，每次調用也都會產生三個最少的日誌陳述式 (START、END 和 REPORT)：



這些日誌會顯示：

- RequestId – 這是每個請求產生的唯一 ID。如果 Lambda 函數重試請求，則此 ID 不會變更，且會在每次後續重試的日誌中顯示。
- 開始/結束 – 此組標記用於界定單次調用範圍，位於其間的所有日誌行皆屬同次調用。
- 持續時間 – 處理常式函式的總調用時間，不含 INIT 程式碼。
- 計費持續時間 – 為了方便計費，套用四捨五入邏輯。
- 記憶體大小 - 配置給函式的記憶體數量。
- 使用的記憶體上限 – 調用期間使用的記憶體上限。
- 初始化持續時間 – 在主要處理常式外部執行 INIT 程式碼區段所耗費的時間。

使用 存取日誌 AWS CLI

AWS CLI 是一種開放原始碼工具，可讓您使用命令列 Shell 中的 命令與 AWS 服務互動。若要完成本節中的步驟，您必須擁有 AWS CLI 版本 2。

您可以透過 AWS CLI，使用 `--log-type` 命令選項來擷取要調用的日誌。其回應將包含 `LogResult` 欄位，內含該次調用的 base64 編碼日誌 (最大達 4 KB)。

Example擷取日誌 ID

下列範例顯示如何從名稱為 `my-function` 的函數的 `LogResult` 欄位來擷取日誌 ID。

```
aws lambda invoke --function-name my-function out --log-type Tail
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "LogResult":
 "U1RBUlQgUmVxdWVzdElkOiA4N2QwNDRiOC1mMTU0LTExZTgtOGNkYS0yOTc0YzVlNGZiMjEgVmVyc2lvb...",
```

```
    "ExecutedVersion": "$LATEST"
}
```

Example解碼日誌

在相同的命令提示中，使用 base64 公用程式來解碼日誌。下列範例顯示如何擷取 my-function 的 base64 編碼日誌。

```
aws lambda invoke --function-name my-function out --log-type Tail \
--query 'LogResult' --output text --cli-binary-format raw-in-base64-out | base64 --
decode
```

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

您應該會看到下列輸出：

```
START RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8 Version: $LATEST
"AWS_SESSION_TOKEN": "AgoJb3JpZ2luX2VjELj...", "_X_AMZN_TRACE_ID": "Root=1-5d02e5ca-
f5792818b6fe8368e5b51d50;Parent=191db58857df8395;Sampled=0"",ask/lib:/opt/lib",
END RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8
REPORT RequestId: 57f231fb-1730-4395-85cb-4f71bd2b87b8  Duration: 79.67 ms      Billed
 Duration: 80 ms         Memory Size: 128 MB     Max Memory Used: 73 MB
```

該 base64 公用程式可在 Linux、macOS 和 Ubuntu on Windows 上使用。macOS 使用者可能需要使用 base64 -D。

Example get-logs.sh 指令碼

在相同的命令提示中，使用下列指令碼下載最後五個日誌事件。該指令碼使用 sed 以從輸出檔案移除引述，並休眠 15 秒以使日誌可供使用。輸出包括來自 Lambda 的回應以及來自 get-log-events 命令的輸出。

複製下列程式碼範例的內容，並將您的 Lambda 專案目錄儲存為 get-logs.sh。

如果您使用的是第 2 AWS CLI 版，則需要 cli-binary-format選項。若要讓此成為預設的設定，請執行 aws configure set cli-binary-format raw-in-base64-out。若要取得更多資訊，請參閱《AWS Command Line Interface 使用者指南第 2 版》中 AWS CLI 支援的全域命令列選項。

```
#!/bin/bash
```

```
aws lambda invoke --function-name my-function --cli-binary-format raw-in-base64-out --
payload '{"key": "value"}' out
sed -i'' -e 's/"//g' out
sleep 15
aws logs get-log-events --log-group-name /aws/lambda/my-function --log-stream-
name stream1 --limit 5
```

Example macOS 和 Linux (僅限)

在相同的命令提示中，macOS 和 Linux 使用者可能需要執行下列命令，以確保指令碼可執行。

```
chmod -R 755 get-logs.sh
```

Example擷取最後五個記錄事件

在相同的命令提示中，執行下列指令碼以取得最後五個日誌事件。

```
./get-logs.sh
```

您應該會看到下列輸出：

```
{
    "StatusCode": 200,
    "ExecutedVersion": "$LATEST"
}
{
    "events": [
        {
            "timestamp": 1559763003171,
            "message": "START RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf Version:
 $LATEST\n",
            "ingestionTime": 1559763003309
        },
        {
            "timestamp": 1559763003173,
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tENVIRONMENT VARIABLES\r{\r  \"AWS_LAMBDA_FUNCTION_VERSION\": \"$LATEST\",
\r ...",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003173,
```

```
            "message": "2019-06-05T19:30:03.173Z\t4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tINFO\tEVENT\r{\r  \"key\": \"value\"\r}\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "END RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf\n",
            "ingestionTime": 1559763018353
        },
        {
            "timestamp": 1559763003218,
            "message": "REPORT RequestId: 4ce9340a-b765-490f-ad8a-02ab3415e2bf
\tDuration: 26.73 ms\tBilled Duration: 27 ms \tMemory Size: 128 MB\tMax Memory Used: 75
 MB\t\n",
            "ingestionTime": 1559763018353
        }
    ],
    "nextForwardToken": "f/34783877304859518393868359594929986069206639495374241795",
    "nextBackwardToken": "b/34783877303811383369537420289090800615709599058929582080"
}
```

剖析日誌和結構化日誌記錄

透過 CloudWatch Logs Insights，您可以使用特殊查詢語法來搜尋和分析日誌資料。它透過多個日誌群組執行查詢，並使用 glob 和規則表達式模式比對提供功能強大的篩選。

您可以在 Lambda 函式中實作結構化日誌記錄，充分運用這些功能優勢。結構化日誌記錄會將日誌整理成預先定義的格式，以利查詢。使用日誌層級是產生易於篩選的日誌的重要第一步，能將資訊性訊息與警告或錯誤區分開。例如，考量以下 Node.js 程式碼：

```
exports.handler = async (event) => {
    console.log("console.log - Application is fine")
    console.info("console.info - This is the same as console.log")
    console.warn("console.warn - Application provides a warning")
    console.error("console.error - An error occurred")
}
```

產生的 CloudWatch 日誌檔案包含指定日誌層級的個別欄位：

```
START RequestId: 99d91f9b-2dff-40ad-b9c8-664020094109 Version: $LATEST
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    INFO    console.log - Application is fine
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    INFO    console.info - This is the same as console.log
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    WARN    console.warn - Application provides a warning
2020-10-22T12:21:28.268Z        99d91f9b-2dff-40ad-b9c8-664020094109    ERROR   console.error - An error occurred
END RequestId: 99d91f9b-2dff-40ad-b9c8-664020094109
REPORT RequestId: 99d91f9b-2dff-40ad-b9c8-664020094109  Duration: 3.13 ms       Billed Duration: 100 ms Memory Size: 128 MB
Max Memory Used: 64 MB  Init Duration: 142.18 ms
```

然後，CloudWatch Logs Insights 查詢可以依日誌層級進行篩選。例如，若要僅查詢錯誤，您可以使用下列查詢：

```
fields @timestamp, @message
| filter @message like /ERROR/
| sort @timestamp desc
```

JSON 結構化日誌記錄

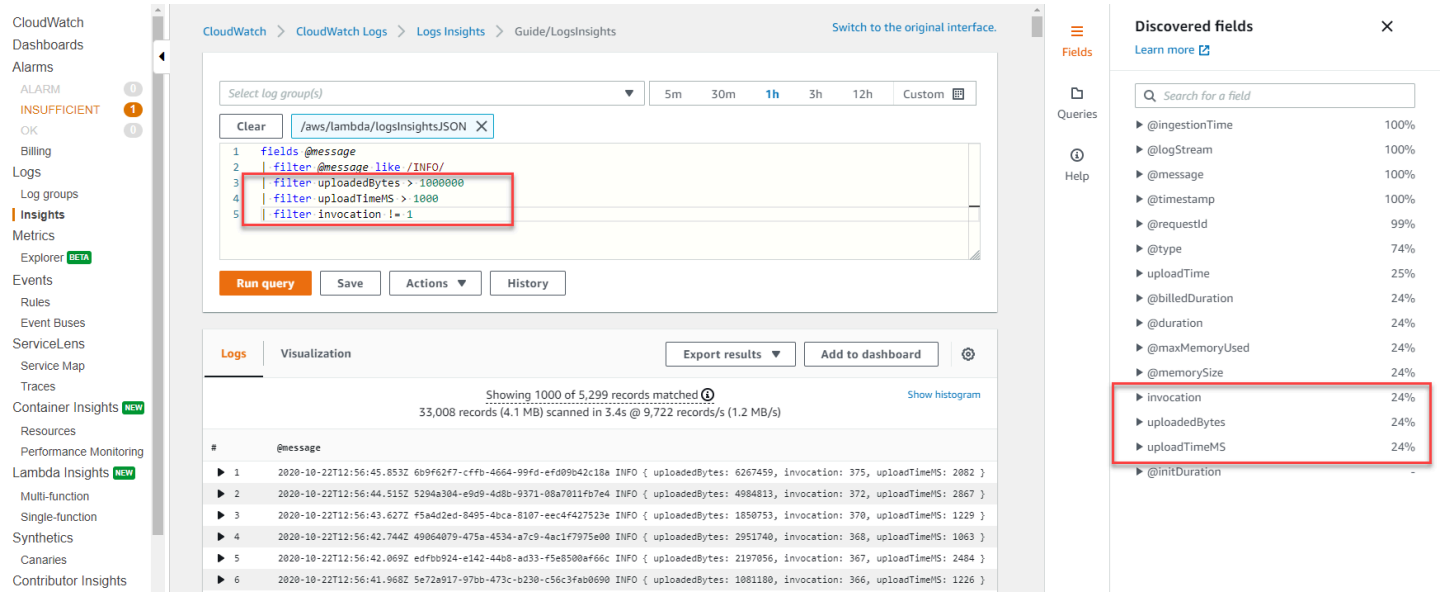
JSON 通常用於提供應用程式日誌的結構。在下列範例中，日誌已轉換為 JSON 以輸出三個相異值：

```
START RequestId: 22fda338-7b46-4c49-9531-efd6e8568480 Version: $LATEST
2020-10-22T11:35:59.216Z        22fda338-7b46-4c49-9531-efd6e8568480    INFO    { uploadedBytes: 5453396,
invocation: 5, uploadTimeMS: 4519 }
END RequestId: 22fda338-7b46-4c49-9531-efd6e8568480
REPORT RequestId: 22fda338-7b46-4c49-9531-efd6e8568480  Duration: 1.25 ms       Billed Duration: 100 ms Memory
Size: 128 MB    Max Memory Used: 65 MB
```

CloudWatch Logs Insights 功能會自動探索 JSON 輸出中的值，並將訊息剖析為欄位，而無需自訂 glob 或規則表達式。透過使用 JSON 結構的日誌，下列查詢會尋找上傳檔案大於 1 MB、上傳時間超過 1 秒且調用不是冷啟動的調用：

```
fields @message
| filter @message like /INFO/
| filter uploadedBytes > 1000000
| filter uploadTimeMS > 1000
| filter invocation != 1
```

此查詢可能產生以下結果：



JSON 中的已探索欄位會自動填入右側的已探索欄位功能表中。Lambda 服務發出的標準欄位字首為 '@'，您可以用相同的方式查詢這些欄位。Lambda 日誌一律包含 @timestamp、@logStream、@message、@requestId、@duration、@billedDuration、@type、@maxMem 欄位。如果為函數啟用 X-Ray，日誌也會包含 @xrayTraceId 和 @xraySegmentId。

當 Amazon S3、Amazon SQS 或 Amazon EventBridge 等 AWS 事件來源調用您的函數時，整個事件會以 JSON 物件輸入的形式提供給函數。透過在函式的第一行中記錄此事件，您可以使用 CloudWatch Logs Insights 查詢任何巢狀欄位。

實用的 Insights 查詢

下表顯示可用於監控 Lambda 函數的範例 Insights 查詢。

Description	範例查詢語法

最後 100 個錯誤	`fields Timestamp, LogLevel, Message`
	`| filter LogLevel == "ERR"`
	`| sort @timestamp desc`
	`| limit 100`

前 100 個最高計費調用	`filter @type = "REPORT"`
	`| fields @requestId, @billedDuration`
	`| sort by @billedDuration desc`
	`| limit 100`

Description	範例查詢語法

冷啟動佔總調用次數的百分比	`filter @type = "REPORT"`
	`| stats sum(strcontains(@message,`
	`"Init Duration"))/count(*) * 100 as`
	`coldStartPct, avg(@duration)`
	`by bin(5m)`

Lambda 持續時間的百分位數報告	`filter @type = "REPORT"`
	`| stats`
	`avg(@billedDuration) as Average,`
	`percentile(@billedDuration, 99)`
	`as NinetyNinth,`
	`percentile(@billedDuration, 95)`
	`as NinetyFifth,`
	`percentile(@billedDuration, 90)`
	`as Ninetieth`
	`by bin(30m)`

Lambda 記憶體用量的百分位數報告	`filter @type="REPORT"`
	`| stats avg(@maxMemoryUsed/1024/102`
	`4) as mean_MemoryUsed,`
	`min(@maxMemoryUsed/1024/1024) as`
	`min_MemoryUsed,`
	`max(@maxMemoryUsed/1024/1024) as`
	`max_MemoryUsed,`
	`percentile(@maxMemoryUsed/1`
	`024/1024, 95) as Percentile95`

使用 100% 指派記憶體的調用	`filter @type = "REPORT" and @maxMemor`
	`yUsed=@memorySize`
	`| stats`
	`count_distinct(@requestId)`
	`by bin(30m)`

Description	範例查詢語法

跨調用使用的平均記憶體	`avgMemoryUsedPERC,`
	`    avg(@billedDuration) as avgDurati`
	`onMS`
	`    by bin(5m)`

視覺化記憶體統計資料	`filter @type = "REPORT"`
	`| stats`
	`    max(@maxMemoryUsed / 1024 / 1024)`
	` as maxMemMB,`
	`    avg(@maxMemoryUsed / 1024 / 1024)`
	` as avgMemMB,`
	`    min(@maxMemoryUsed / 1024 / 1024)`
	` as minMemMB,`
	`    (avg(@maxMemoryUsed / 1024 /`
	` 1024) / max(@memorySize / 1024 /`
	` 1024)) * 100 as avgMemUsedPct,`
	`    avg(@billedDuration) as avgDurati`
	`onMS`
	`    by bin(30m)`

Lambda 結束的調用	`filter @message like /Process exited/` `| stats count() by bin(30m)`
逾時的調用	`filter @message like /Task timed out/` `| stats count() by bin(30m)`

延遲報告	`filter @type = "REPORT"`
	`| stats avg(@duration), max(@dura`
	`tion), min(@duration)`
	`  by bin(5m)`

Description	範例查詢語法
過度佈建的記憶體	`filter @type = "REPORT" | stats max(@memorySize / 1024 / 1024) as provisonedMemMB, min(@maxMemoryUsed / 1024 / 1024) as smallestMemReqMB, avg(@maxMemoryUsed / 1024 / 1024) as avgMemUsedMB, max(@maxMemoryUsed / 1024 / 1024) as maxMemUsedMB, provisonedMemMB - maxMemUsedMB as overProvisionedMB`

日誌視覺化和儀表板

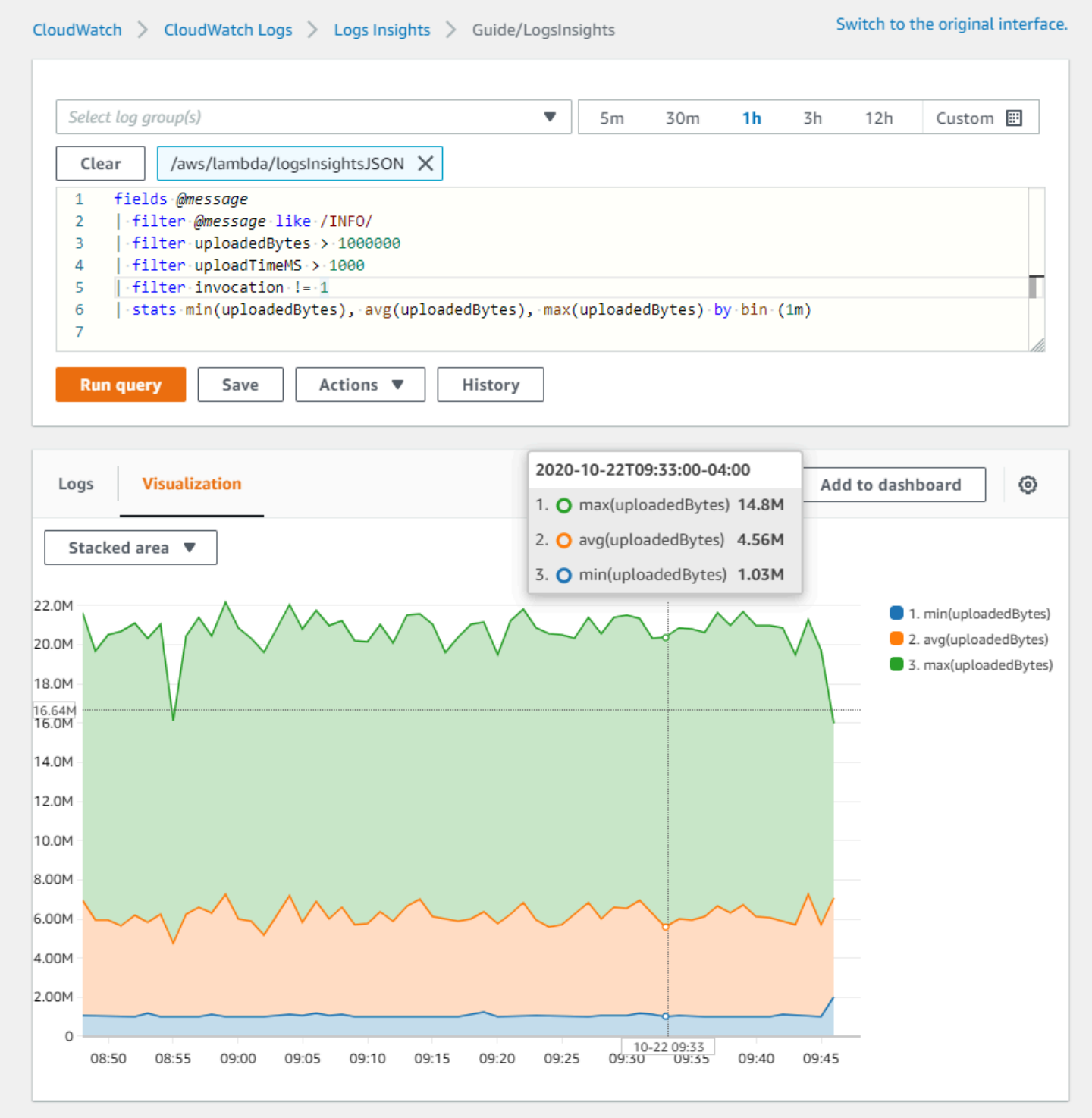
對於任何 CloudWatch Logs Insights 查詢，您都可以將結果匯出為 Markdown 或 CSV 格式。在某些情況下，如果至少有一個彙總函式，透過查詢建立視覺化呈現可能會更有用。stats 函式可讓您定義彙總與分組規則。

上一個 logInsightsJSON 範例根據上傳大小和上傳時間進行篩選，並排除第一次調用。這會產生資料表。為了監控生產系統，視覺化最小、最大和平均檔案大小來尋找極端值可能更有用。為此，請套用具有所需彙總的 stats 函數，並按時間值 (例如每分鐘) 進行分組：

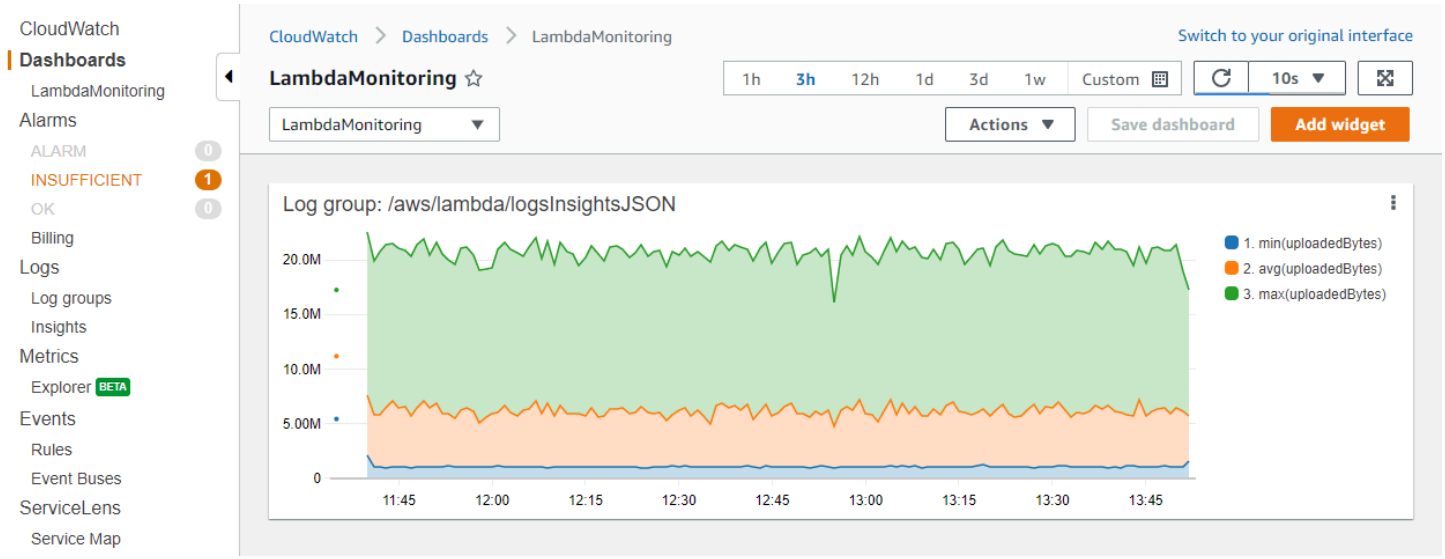
例如，考量以下查詢。此查詢與 the section called “JSON 結構化日誌記錄” 區段中的範例查詢相同，但新增了額外的彙總函式：

```
fields @message
| filter @message like /INFO/
| filter uploadedBytes > 1000000
| filter uploadTimeMS > 1000
| filter invocation != 1
| stats min(uploadedBytes), avg(uploadedBytes), max(uploadedBytes) by bin (1m)
```

之所以納入這些彙總計算，是因視覺化呈現檔案大小的最小值、最大值與平均值，將更有助於找出異常值。您可以在視覺化索引標籤中檢視結果：



完成建置視覺化後，您可以選擇將圖形新增至 CloudWatch 儀表板。為此，請選擇視覺化上方的新增至儀表板。這會將查詢新增為小工具，並可讓您選取自動重新整理間隔，從而讓您更輕鬆地持續監控結果：



將 Lambda 函式日誌傳送至 Firehose

Lambda 主控台現在提供將函式日誌傳送至 Firehose 的選項。此功能可將日誌即時串流至 Firehose 支援的多種目的地，包括第三方分析工具與自訂端點。

> **Note**
>
> 您可以透過 Lambda 主控台、AWS CLI、AWS CloudFormation 和所有 AWS SDK，設定將 Lambda 函式日誌傳送至 Firehose。

定價

如需定價詳細資訊，請參閱 Amazon CloudWatch 定價。

Firehose 日誌目的地的必要許可

在使用 Lambda 主控台將 Firehose 設定為函式的日誌目的地時，您需要滿足如下條件：

1. 將 CloudWatch Logs 與 Lambda 搭配使用所需的 IAM 許可。
2. 設定與 Firehose 搭配使用的訂閱篩選條件。此篩選條件會定義要傳送至 Firehose 串流的日誌事件。

將 Lambda 函式日誌傳送至 Firehose

在 Lambda 主控台中，您可以在建立新函式之後將函式日誌直接傳送至 Firehose。若要執行此動作，請執行下列步驟：

1. 登入 AWS 管理主控台，並開啟 Lambda 主控台。
2. 選擇函式的名稱。
3. 選擇 Configuration (組態) 索引標籤。
4. 選擇監控與操作工具索引標籤。
5. 在「日誌記錄組態」區段中，選擇編輯。
6. 在「日誌內容」區段中，選取一種日誌格式。
7. 在「日誌目的地」區段中，完成下列步驟：
   a. 選擇目的地服務。
   b. 選擇建立新日誌群組，或使用現有日誌群組。

   > **Note**
   >
   > 若為 Firehose 目的地選擇現有日誌群組，請確保所選日誌群組的類型為 `Delivery` 日誌群組。

   c. 選擇 Firehose 串流。
   d. CloudWatch `Delivery` 日誌群組隨即顯示。
8. 選擇儲存。

> **Note**
>
> 若主控台中提供的 IAM 角色沒有必要許可，則目的地設定會失敗。若要修正此問題，請參閱「Firehose 日誌目的地的必要許可」一節，提供必要許可。

跨帳戶日誌記錄

您可以設定 Lambda 將日誌傳送至不同 AWS 帳戶的 Firehose 傳輸串流。這需要設定目的地，並在兩個帳戶中設定適當的許可。

如需有關設定跨帳戶日誌記錄的詳細說明，包括必要的 IAM 角色與政策，請參閱 CloudWatch Logs 文件中的 Setting up a new cross-account subscription。

將 Lambda 函式日誌傳送至 Amazon S3

您可以透過 Lambda 主控台設定 Lambda 函式，將日誌直接傳送至 Amazon S3。此功能為長期日誌儲存提供了一種經濟實惠的解決方案，還可使用 Athena 等服務實現強大的分析功能。

> **Note**
>
> 您可以透過 Lambda 主控台、AWS CLI、AWS CloudFormation 和所有 AWS SDK，設定將 Lambda 函式日誌傳送至 Amazon S3。

定價

如需定價詳細資訊，請參閱 Amazon CloudWatch 定價。

Amazon S3 日誌目的地的必要許可

在使用 Lambda 主控台將 Amazon S3 設定為函式的日誌目的地時，您需要滿足如下條件：

1. 將 CloudWatch Logs 與 Lambda 搭配使用所需的 IAM 許可。
2. 設定 CloudWatch Logs 訂閱篩選條件，將 Lambda 函式日誌傳送至 Amazon S3。此篩選條件會定義要傳送至 Amazon S3 儲存貯體的日誌事件。

設定 CloudWatch Logs 訂閱篩選條件，將 Lambda 函式日誌傳送至 Amazon S3

要將日誌從 CloudWatch Logs 傳送至 Amazon S3，您需建立訂閱篩選條件。此篩選條件會定義要傳送至 Amazon S3 儲存貯體的日誌事件。您的 Amazon S3 儲存貯體必須與日誌群組位於同一個區域。

建立 Amazon S3 的訂閱篩選條件

1. 建立 Amazon Simple Storage Service (Amazon S3) 儲存貯體。我們建議您使用專為 CloudWatch Logs 建立的儲存貯體。不過，如果您想要使用現有的儲存貯體，請跳到步驟 2。

   執行以下命令，將預留位置 Region 換成您想要使用的區域：

   ```
   aws s3api create-bucket --bucket amzn-s3-demo-bucket2 --create-bucket-configuration
    LocationConstraint=region
   ```

> **Note**
>
> amzn-s3-demo-bucket2 為 Amazon S3 儲存貯體名稱的範例。該名稱已被保留。若要讓此程序正常運作，您必須將其取代為唯一的 Amazon S3 儲存貯體名稱。

下列為範例輸出：

```
{
    "Location": "/amzn-s3-demo-bucket2"
}
```

2. 建立 IAM 角色，授予 CloudWatch Logs 將資料放入 Amazon S3 儲存貯體的許可。此政策包含一個 aws:SourceArn 全域條件內容索引鍵，可協助防範混淆代理人安全問題。如需詳細資訊，請參閱混淆代理人防範。

   a. 使用文字編輯器在 ~/TrustPolicyForCWL.json 檔案中建立信任政策，如下所示：

   ```
   {
       "Statement": {
           "Effect": "Allow",
           "Principal": { "Service": "logs.amazonaws.com" },
           "Condition": {
               "StringLike": {
                   "aws:SourceArn": "arn:aws:logs:region:123456789012:*"
               }
            },
           "Action": "sts:AssumeRole"
       }
   }
   ```

   b. 使用 create-role 命令來建立 IAM 角色，並指定信任政策檔案。請注意傳回的 Role.Arn 值，因您將在後續步驟需要此值：

   ```
   aws iam create-role \
    --role-name CWLtoS3Role \
    --assume-role-policy-document file://~/TrustPolicyForCWL.json
   {
       "Role": {
           "AssumeRolePolicyDocument": {
               "Statement": {
   ```

```
                "Action": "sts:AssumeRole",
                "Effect": "Allow",
                "Principal": {
                    "Service": "logs.amazonaws.com"
                },
                "Condition": {
                    "StringLike": {
                        "aws:SourceArn": "arn:aws:logs:region:123456789012:*"
                    }
                }
            }
        },
        "RoleId": "AAOIIAH450GAB4HC5F431",
        "CreateDate": "2015-05-29T13:46:29.431Z",
        "RoleName": "CWLtoS3Role",
        "Path": "/",
        "Arn": "arn:aws:iam::123456789012:role/CWLtoS3Role"
    }
}
```

3. 建立許可政策以定義 CloudWatch Logs 可在您的帳戶上執行哪些動作。首先，使用文字編輯器來建立檔案 ~/PermissionsForCWL.json 中的許可政策：

```
{
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:PutObject"],
      "Resource": ["arn:aws:s3:::amzn-s3-demo-bucket2/*"]
    }
  ]
}
```

使用下列 put-role-policy 命令，將許可政策與角色建立關聯：

```
aws iam put-role-policy --role-name CWLtoS3Role --policy-name Permissions-Policy-For-S3 --policy-document file://~/PermissionsForCWL.json
```

4. 建立一個 Delivery 日誌群組，或使用現有的 Delivery 日誌群組。

```
aws logs create-log-group --log-group-name my-logs --log-group-class DELIVERY --region REGION_NAME
```

5. 使用 PutSubscriptionFilter 命令設定目的地

```
aws logs put-subscription-filter
--log-group-name my-logs
--filter-name my-lambda-delivery
--filter-pattern ""
--destination-arn arn:aws:s3:::amzn-s3-demo-bucket2
--role-arn arn:aws:iam::123456789012:role/CWLtoS3Role
--region REGION_NAME
```

將 Lambda 函式日誌傳送至 Amazon S3

在 Lambda 主控台中，您可以在建立新函式之後將函式日誌直接傳送至 Amazon S3。若要執行此動作，請執行下列步驟：

1. 登入 AWS 管理主控台，並開啟 Lambda 主控台。
2. 選擇函式的名稱。
3. 選擇 Configuration (組態) 索引標籤。
4. 選擇監控與操作工具索引標籤。
5. 在「日誌記錄組態」區段中，選擇編輯。
6. 在「日誌內容」區段中，選取一種日誌格式。
7. 在「日誌目的地」區段中，完成下列步驟：
   a. 選擇目的地服務。
   b. 選擇建立新日誌群組，或使用現有日誌群組。

   > **Note**
   >
   > 若為 Amazon S3 目的地選擇現有日誌群組，請確保所選日誌群組的類型為 Delivery 日誌群組。

   c. 選擇要用作函式日誌目的地的 Amazon S3 儲存貯體。
   d. CloudWatch Delivery 日誌群組隨即顯示。
8. 選擇儲存。

> **Note**
>
> 若主控台中提供的 IAM 角色沒有必要許可，則目的地設定會失敗。若要修正此問題，請參閱 Amazon S3 日誌目的地的必要許可。

跨帳戶日誌記錄

您可以設定 Lambda 將日誌傳送至不同 AWS 帳戶的 Amazon S3 儲存貯體。這需要設定目的地，並在兩個帳戶中設定適當的許可。

如需有關設定跨帳戶日誌記錄的詳細說明，包括必要的 IAM 角色與政策，請參閱 CloudWatch Logs 文件中的 Setting up a new cross-account subscription。

使用 記錄 AWS Lambda API 呼叫 AWS CloudTrail

AWS Lambda 已與 整合AWS CloudTrail，此服務提供使用者、角色或 所採取動作的記錄 AWS 服務。CloudTrail 會擷取 Lambda 的 API 呼叫當作事件。擷取的呼叫包括從 Lambda 主控台進行的呼叫，以及針對 Lambda API 操作的程式碼呼叫。您可以利用 CloudTrail 收集的資訊來判斷向 Lambda 發出的請求，以及發出請求的 IP 位址、時間和其他詳細資訊。

每一筆事件或日誌專案都會包含產生請求者的資訊。身分資訊可協助您判斷下列事項：

- 該請求是使用根使用者還是使用者憑證提出。
- 請求是否代表 IAM Identity Center 使用者提出。
- 提出該請求時，是否使用了特定角色或聯合身分使用者的暫時安全憑證。
- 該請求是否由另一項 AWS 服務服務提出。

當您建立帳戶 AWS 帳戶 時CloudTrail 會在 中處於作用中狀態，而且您會自動存取 CloudTrail 事件歷史記錄。CloudTrail 事件歷史記錄為 AWS 區域中過去 90 天記錄的管理事件，提供可檢視、可搜尋、可下載且不可變的記錄。如需詳細資訊，請參閱「AWS CloudTrail 使用者指南」中的使用 CloudTrail 事件歷史記錄。檢視事件歷史記錄不會產生 CloudTrail 費用。

如需 AWS 帳戶 過去 90 天內持續記錄的事件，請建立線索或 CloudTrail Lake 事件資料存放區。

CloudTrail 追蹤

線索能讓 CloudTrail 將日誌檔案交付至 Amazon S3 儲存貯體。使用 建立的所有線索 AWS 管理主控台 都是多區域。您可以使用 AWS CLI建立單一或多區域追蹤。建議您建立多區域追蹤，因為您擷取 AWS 區域 帳戶中所有 的活動。如果您建立單一區域追蹤，您只能檢視追蹤 AWS 區域中記錄的事件。如需追蹤的詳細資訊，請參閱《AWS CloudTrail 使用者指南》中的為您的 AWS 帳戶建立追蹤和為組織建立追蹤。

您可以透過建立追蹤，免費將持續管理事件的一個複本從 CloudTrail 傳遞至您的 Amazon S3 儲存貯體，但這樣做會產生 Amazon S3 儲存費用。如需 CloudTrail 定價的詳細資訊，請參閱 AWS CloudTrail 定價。如需 Amazon S3 定價的相關資訊，請參閱 Amazon S3 定價。

CloudTrail Lake 事件資料存放區

CloudTrail Lake 讓您能夠對事件執行 SQL 型查詢。CloudTrail Lake 會將分列式 JSON 格式的現有事件轉換為 Apache ORC 格式。ORC 是一種單欄式儲存格式，針對快速擷取資料進行了最佳化。系統會將事件彙總到事件資料存放區中，事件資料存放區是事件的不可變集合，其依據為您透過套用進階事件選取器選取的條件。套用於事件資料存放區的選取器控制哪些事件持續存在並可供您

查詢。如需 CloudTrail Lake 的詳細資訊，請參閱AWS CloudTrail 《 使用者指南》中的使用 AWS CloudTrail Lake。

CloudTrail Lake 事件資料存放區和查詢會產生費用。建立事件資料存放區時，您可以選擇要用於事件資料存放區的定價選項。此定價選項將決定擷取和儲存事件的成本，以及事件資料存放區的預設和最長保留期。如需 CloudTrail 定價的詳細資訊，請參閱 AWS CloudTrail 定價。

CloudTrail 中的 Lambda 資料事件

資料事件提供在資源上或在資源中執行的資源操作的相關資訊 (例如，讀取或寫入 Amazon S3 物件)。這些也稱為資料平面操作。資料事件通常是高數量級的活動。根據預設，CloudTrail 不會記錄大多數資料事件，而 CloudTrail 事件歷史記錄也不會記錄它們。

根據預設，為支援的服務記錄的其中一個 CloudTrail 資料事件為 `LambdaESMDisabled`。若要進一步了解使用此事件來協助對 Lambda 事件來源映射問題進行疑難排解，請參閱the section called “使用 CloudTrail 疑難排解已停用的 Lambda 事件來源”。

資料事件需支付額外的費用。如需 CloudTrail 定價的詳細資訊，請參閱 AWS CloudTrail 定價。

您可以使用 CloudTrail 主控台 AWS CLI或 CloudTrail API 操作來記錄`AWS::Lambda::Function`資源類型的資料事件。如需有關如何記錄資料事件的詳細資訊，請參閱《AWS CloudTrail 使用者指南》中的 Logging data events with the AWS 管理主控台 和 Logging data events with the AWS Command Line Interface。

下表列出您可以記錄其資料事件的 Lambda 資源類型。資料事件類型 (主控台) 資料行顯示從 CloudTrail 主控台上的資料事件類型清單中選擇的值。resources.type 值欄會顯示值，您會在使用 AWS CLI 或 CloudTrail APIs 設定進階事件選取器時指定此`resources.type`值。記錄到 CloudTrail 的資料 API 資料行會針對資源類型顯示記錄到 CloudTrail 的 API 呼叫。

資料事件類型 (主控台)	resources.type 值	記錄到 CloudTrail 的資料 API
Lambda	`AWS::Lambda::Function`	Invoke

您可以設定進階事件選取器來篩選 `eventName`、`readOnly` 和 `resources.ARN` 欄位，以僅記錄對您重要的事件。下列範例是資料事件組態的 JSON 檢視，該組態僅記錄特定函數的事件。如需有關這些欄位的詳細資訊，請參閱《AWS CloudTrail API 參考》中的 AdvancedFieldSelector。

```
[
  {
    "name": "function-invokes",
    "fieldSelectors": [
      {
        "field": "eventCategory",
        "equals": [
          "Data"
        ]
      },
      {
        "field": "resources.type",
        "equals": [
          "AWS::Lambda::Function"
        ]
      },
      {
        "field": "resources.ARN",
        "equals": [
          "arn:aws:lambda:us-east-1:111122223333:function:hello-world"
        ]
      }
    ]
  }
]
```

CloudTrail 中的 Lambda 管理事件

管理事件提供有關在 資源上執行的管理操作的資訊 AWS 帳戶。這些也稱為控制平面操作。依預設，CloudTrail 會記錄管理事件。

Lambda 支援將下列 API 動作記錄為 CloudTrail 日誌檔案中的管理事件。

> **Note**
>
> 在 CloudTrail 日誌檔案中，`eventName` 可能包含日期與版本資訊，但仍參照至相同的公有 API 動作。例如，`GetFunction` 動作會顯示為 `GetFunction20150331v2`。下列清單指定事件名稱何時與 API 動作名稱不同。

- AddLayerVersionPermission

- AddPermission (事件名稱：AddPermission20150331v2)
- CreateAlias (事件名稱：CreateAlias20150331)
- CreateEventSourceMapping (事件名稱：CreateEventSourceMapping20150331)
- CreateFunction (事件名稱：CreateFunction20150331)

  (忽略來自 CreateFunction 的 CloudTrail 日誌中的 Environment 和 ZipFile 參數)。
- CreateFunctionUrlConfig
- DeleteAlias (事件名稱：DeleteAlias20150331)
- DeleteCodeSigningConfig
- DeleteEventSourceMapping (事件名稱：DeleteEventSourceMapping20150331)
- DeleteFunction (事件名稱：DeleteFunction20150331)
- DeleteFunctionConcurrency (事件名稱：DeleteFunctionConcurrency20171031)
- DeleteFunctionUrlConfig
- DeleteProvisionedConcurrencyConfig
- GetAlias (事件名稱：GetAlias20150331)
- GetEventSourceMapping
- GetFunction
- GetFunctionUrlConfig
- GetFunctionConfiguration
- GetLayerVersionPolicy
- GetPolicy
- ListEventSourceMappings
- ListFunctions
- ListFunctionUrlConfigs
- PublishLayerVersion (事件名稱：PublishLayerVersion20181031)

  (從 CloudTrail 日誌針對 PublishLayerVersion 省略 ZipFile 參數)。
- PublishVersion (事件名稱：PublishVersion20150331)
- PutFunctionConcurrency (事件名稱：PutFunctionConcurrency20171031)
- PutFunctionCodeSigningConfig
- PutFunctionEventInvokeConfig
- PutProvisionedConcurrencyConfig

- PutRuntimeManagementConfig
- RemovePermission (事件名稱：RemovePermission20150331v2)
- TagResource (事件名稱：TagResource20170331v2)
- UntagResource (事件名稱：UntagResource20170331v2)
- UpdateAlias (事件名稱：UpdateAlias20150331)
- UpdateCodeSigningConfig
- UpdateEventSourceMapping (事件名稱：UpdateEventSourceMapping20150331)
- UpdateFunctionCode (事件名稱：UpdateFunctionCode20150331v2)

  (從 CloudTrail 日誌針對 UpdateFunctionCode 省略 ZipFile 參數)。
- UpdateFunctionConfiguration (事件名稱：UpdateFunctionConfiguration20150331v2)

  (從 CloudTrail 日誌針對 UpdateFunctionConfiguration 省略 Environment 參數)。
- UpdateFunctionEventInvokeConfig
- UpdateFunctionUrlConfig

使用 CloudTrail 疑難排解已停用的 Lambda 事件來源

當您使用 UpdateEventSourceMapping API 動作變更事件來源映射的狀態時，該 API 呼叫會在 CloudTrail 中記錄為管理事件。由於發生錯誤，事件來源映射也可直接轉換為 Disabled 狀態。

針對下列服務，當您的事件來源轉換為已停用狀態時，Lambda 會將 LambdaESMDisabled 資料事件發布至 CloudTrail：

- Amazon Simple Queue Service (Amazon SQS)
- Amazon DynamoDB
- Amazon Kinesis

Lambda 不支援任何其他事件來源映射類型的此事件。

若要在所支援服務的事件來源映射轉換為 Disabled 狀態時接收提醒，請使用 LambdaESMDisabled CloudTrail 事件在 Amazon CloudWatch 中設定警示。若要進一步了解如何設定 CloudWatch 警示，請參閱 Creating CloudWatch alarms for CloudTrail events: examples。

LambdaESMDisabled 事件訊息中的 serviceEventDetails 實體包含下列其中一個錯誤代碼。

RESOURCE_NOT_FOUND

請求中指定的資源不存在。

FUNCTION_NOT_FOUND

附加至事件來源的函數不存在。

REGION_NAME_NOT_VALID

提供給事件來源或函數的區域名稱無效。

AUTHORIZATION_ERROR

權限尚未設定，或設定錯誤。

FUNCTION_IN_FAILED_STATE

函數程式碼無法編譯、發生無法復原的例外狀況，或發生錯誤的部署。

Lambda 事件範例

一個事件代表任何來源提出的單一請求，並包含請求 API 操作的相關資訊、操作的日期和時間、請求參數等等。CloudTrail 日誌檔案並非依公有 API 呼叫的堆疊追蹤排序，因此事件不會以任何特定順序出現。

以下範例顯示的是 `GetFunction` 和 `DeleteFunction` 動作的 CloudTrail 日誌項目。

> **Note**
>
> `eventName` 可能包含日期與版本資訊，如 `"GetFunction20150331"`，但仍參照至相同的公有 API。

```
{
  "Records": [
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
```

```
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:03:36Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "GetFunction",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "errorCode": "AccessDenied",
      "errorMessage": "User: arn:aws:iam::111122223333:user/myUserName is not
 authorized to perform: lambda:GetFunction on resource: arn:aws:lambda:us-
west-2:111122223333:function:other-acct-function",
      "requestParameters": null,
      "responseElements": null,
      "requestID": "7aebcd0f-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "e92a3e85-8ecd-4d23-8074-843aabfe89bf",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    },
    {
      "eventVersion": "1.03",
      "userIdentity": {
        "type": "IAMUser",
        "principalId": "A1B2C3D4E5F6G7EXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/myUserName",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "myUserName"
      },
      "eventTime": "2015-03-18T19:04:42Z",
      "eventSource": "lambda.amazonaws.com",
      "eventName": "DeleteFunction20150331",
      "awsRegion": "us-east-1",
      "sourceIPAddress": "127.0.0.1",
      "userAgent": "Python-httplib2/0.8 (gzip)",
      "requestParameters": {
        "functionName": "basic-node-task"
      },
      "responseElements": null,
      "requestID": "a2198ecc-cda1-11e4-aaa2-e356da31e4ff",
      "eventID": "20b84ce5-730f-482e-b2b2-e8fcc87ceb22",
      "eventType": "AwsApiCall",
      "recipientAccountId": "111122223333"
    }
```

```
    ]
}
```

如需有關 CloudTrail 記錄內容的資訊，請參閱《AWS CloudTrail 使用者指南》中的 CloudTrail record contents。

使用 視覺化 Lambda 函數叫用 AWS X-Ray

您可以使用 AWS X-Ray 視覺化應用程式的元件、識別效能瓶頸，以及對導致錯誤的請求進行故障診斷。您的 Lambda 函數會將追蹤資料傳送至 X-Ray，而且 X-Ray 會處理資料，以產生服務映射和可搜尋的追蹤摘要。

Lambda 支援兩種 X-Ray 追蹤模式：`Active` 和 `PassThrough`。啟用 `Active` 追蹤時，Lambda 會自動為函式調用建立追蹤區段，並將其傳送至 X-Ray。另一方面，`PassThrough` 模式僅會將追蹤內容傳播至下游服務。若您已為函式啟用 `Active` 追蹤，Lambda 會自動將取樣請求的追蹤傳送至 X-Ray。通常，上游服務 (例如 Amazon API Gateway) 或託管於 Amazon EC2 且嵌入 X-Ray SDK 的應用程式，會決定是否應該追蹤傳入請求，而後將該取樣決策新增為追蹤標頭。Lambda 會使用該標頭來決定是否傳送追蹤。來自上游訊息生產者的追蹤 (例如 Amazon SQS) 會自動連結至來自下游 Lambda 函數的追蹤，進而建立整個應用程式的端對端檢視。如需詳細資訊，請參閱《AWS X-Ray 開發人員指南》中的追蹤事件導向應用程式。

> **Note**
>
> Lambda 函數若具有 Amazon Managed Streaming for Apache Kafka (Amazon MSK)、自我管理的 Apache Kafka、帶有 ActiveMQ 和 RabbitMQ 的 Amazon MQ，或是 Amazon DocumentDB 事件來源映射，目前不支援 X-Ray 追蹤。

若要使用控制台在 Lambda 函數上切換主動追蹤，請按照下列步驟操作：

開啟主動追蹤

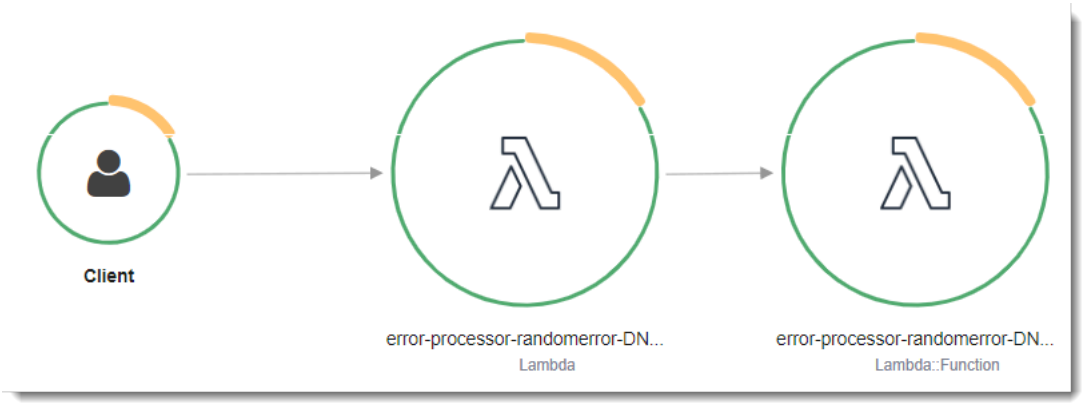
1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 選擇組態，然後選擇監控和操作工具。
4. 在其他監控工具欄位中，選擇編輯。
5. 在 CloudWatch Application Signals 和 AWS X-Ray 下，選擇啟用 Lambda 服務追蹤。
6. 選擇儲存。

您的函數需要將追蹤資料上傳至 X-Ray 的許可。當您在 Lambda 主控台中啟用追蹤時，Lambda 會將必要的許可新增至函數的執行角色。否則，請將 AWSXRayDaemonWriteAccess 政策新增至執行角色。

X-Ray 無法追蹤應用程式的所有請求。X-Ray 會套用取樣演算法以確保追蹤的效率，同時仍提供所有請求的代表範本。取樣率為每秒 1 次請求和 5% 的額外請求。不能針對函數設定 X-Ray 取樣率。

了解 X-Ray 追蹤

在 X-Ray 中，追蹤會記錄一或多個服務所處理之要求的相關資訊。Lambda 會在每個追蹤上記錄 2 個區段，這會在服務圖表上建立兩個節點。下圖反白顯示了這兩個節點：



左側第一個節點代表接收調用請求的 Lambda 服務。第二個節點代表您特定的 Lambda 函數。

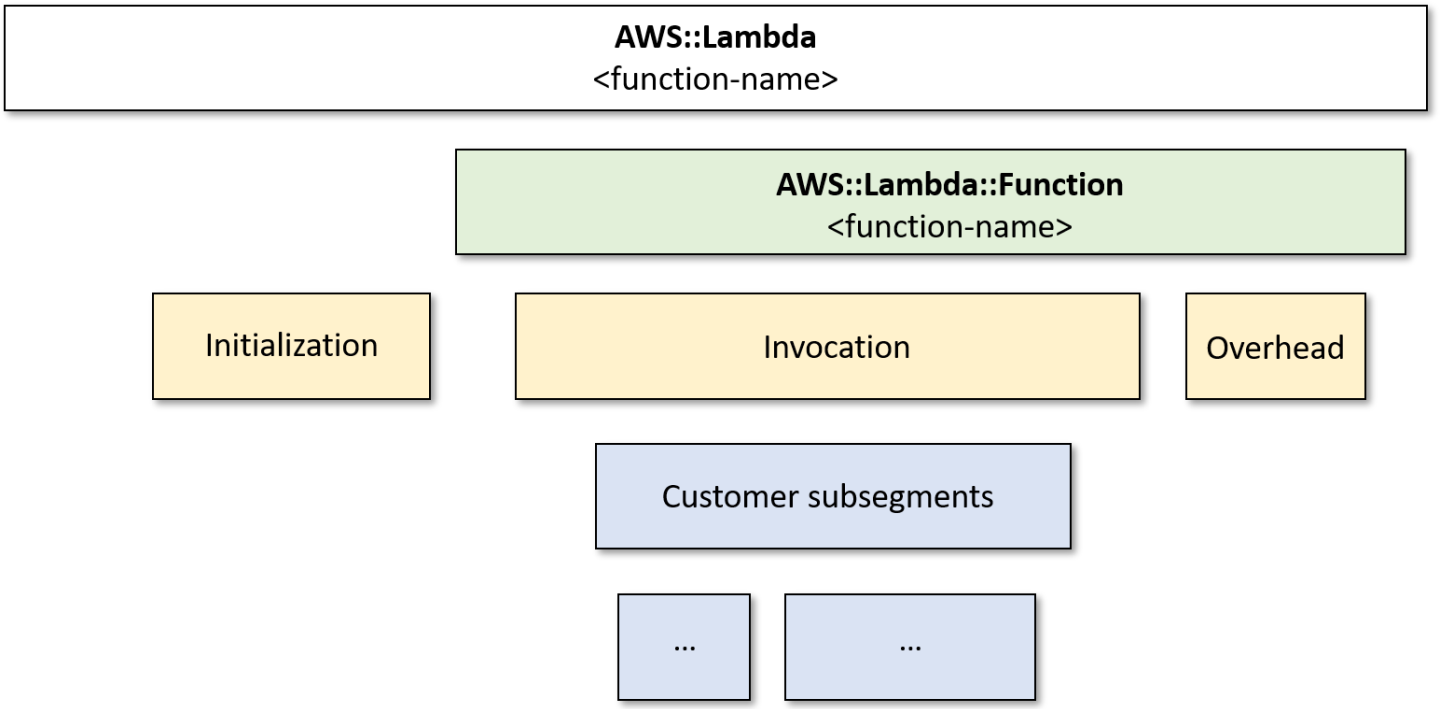
為 Lambda 服務記錄的區段 `AWS::Lambda` 涵蓋準備 Lambda 執行環境所需的所有步驟。這包括排程 MicroVM、使用您設定的資源建立或取消凍結執行環境，以及下載函數程式碼和所有層。

`AWS::Lambda::Function` 區段用於函數要完成的工作。

> **Note**
>
> AWS 目前正在對 Lambda 服務實作變更。由於這些變更，您可能會看到系統日誌訊息的結構和內容，與 AWS 帳戶中不同 Lambda 函數發出的追蹤區段之間存在細微差異。
> 此變更會影響該函數區段的子區段。以下段落說明這些子區段的舊格式和新格式。
> 這些變化將在未來幾週內實作，除中國和 GovCloud 區域以外，所有 AWS 區域 中的所有函數都會轉換至使用新格式的日誌訊息和追蹤區段。

舊式 AWS X-Ray Lambda 區段結構

`AWS::Lambda` 區段的舊式 X-Ray 結構如下所示：



在此格式中，函數區段具有 `Initialization`、`Invocation` 和 `Overhead` 的子區段。另外，對於 Lambda SnapStart 還有一個 `Restore` 子區段 (此圖表中未顯示)。

`Initialization` 子區段代表 Lambda 執行環境生命週期的初始化階段。在此階段，Lambda 會初始化延伸模組、初始化執行時期，並執行函數的初始化程式碼。

`Invocation` 子區段表示調用階段，其中 Lambda 調用函數處理常式。這從執行時間和延伸註冊開始，並在執行時間準備傳送響應時結束。

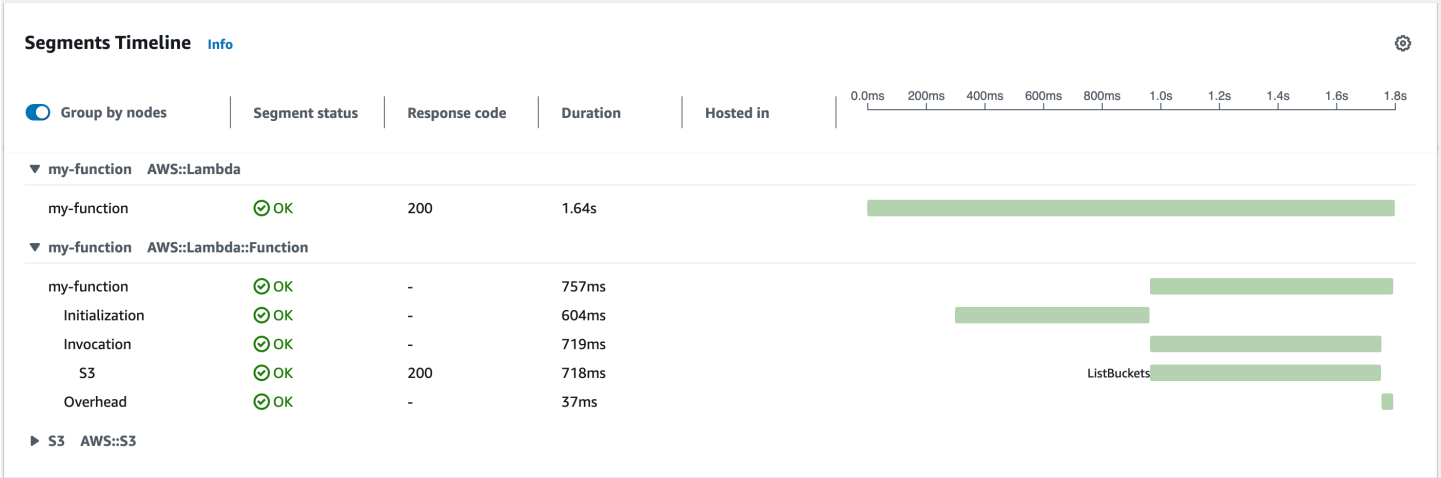
(僅限 Lambda SnapStart) `Restore` 子區段會顯示 Lambda 還原快照、載入執行時期和執行任何還原後執行時期勾點所需的時間。還原快照的程序可能包括在 MicroVM 以外的活動上花費的時間。此時間在 `Restore` 子區段中報告。您不需要為在 MicroVM 外還原快照所花費的時間付費。

`Overhead` 子區段表示當執行時間傳送響應和下一次調用信號之間的時間發生的階段。在此期間，執行時間會完成與調用相關的所有工作，並準備凍結沙盒。

> **Important**
>
> 您可以使用 X-Ray 開發套件來擴充 `Invocation` 子區段與下游呼叫、註釋和中繼資料的其他子區段。您無法直接存取函數區段，或在處理常式調用範圍之外記錄完成的工作。

如需 Lambda 執行環境階段的詳細資訊，請參閱the section called “執行環境”。

下圖顯示了使用舊式 X-Ray 結構的追蹤範例。

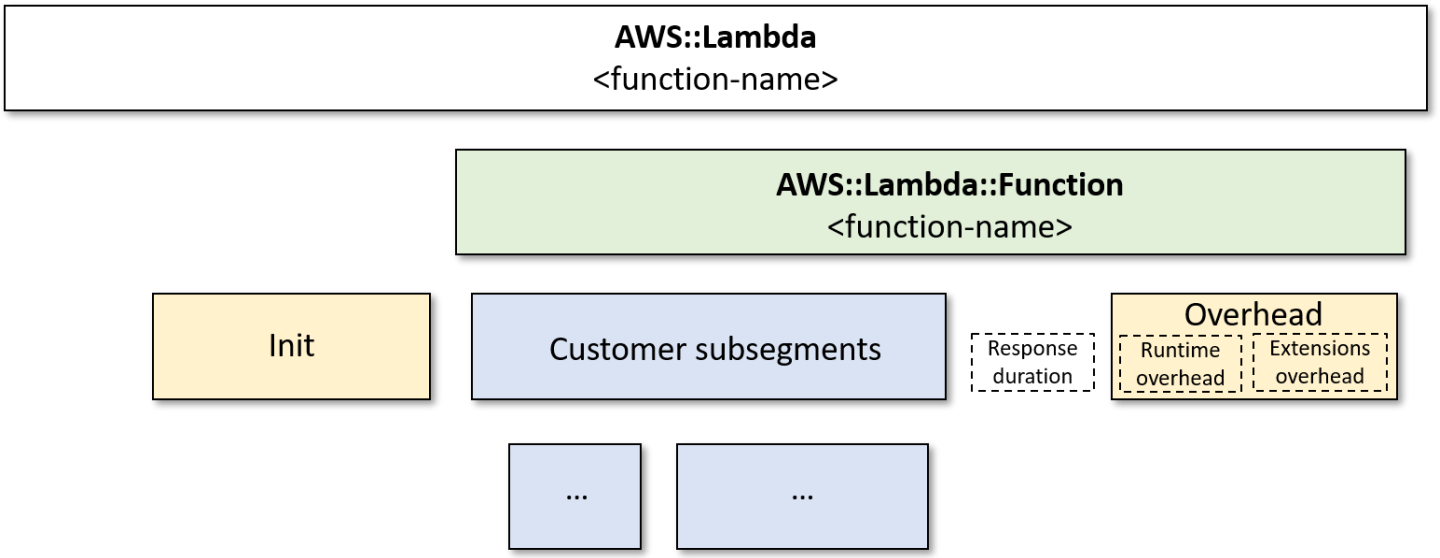


請注意此範例中的兩個區段。兩者都被命名為 my-function，但其中之一的來源為 `AWS::Lambda`，而另一個的來源為 `AWS::Lambda::Function`。如果 `AWS::Lambda` 區段顯示錯誤，Lambda 服務就會出現問題。如果 `AWS::Lambda::Function` 區段顯示錯誤，表示您的函數出現了問題。

> **Note**
>
> 偶爾您可能會注意到 X-Ray 追蹤中的函數初始化和調用階段之間存在很大的差距。對於使用佈建並行的函數，這是因為 Lambda 會在調用之前才初始化函數執行個體。對於使用未預留 (隨需) 並行的函數，即使沒有調用，Lambda 也可能會主動初始化函數執行個體。從視覺上看，這兩種情況都會顯示為初始化和調用階段之間的時間差距。

新樣式 AWS X-Ray 的 Lambda 區段結構

`AWS::Lambda` 區段的新式 X-Ray 結構如下所示：

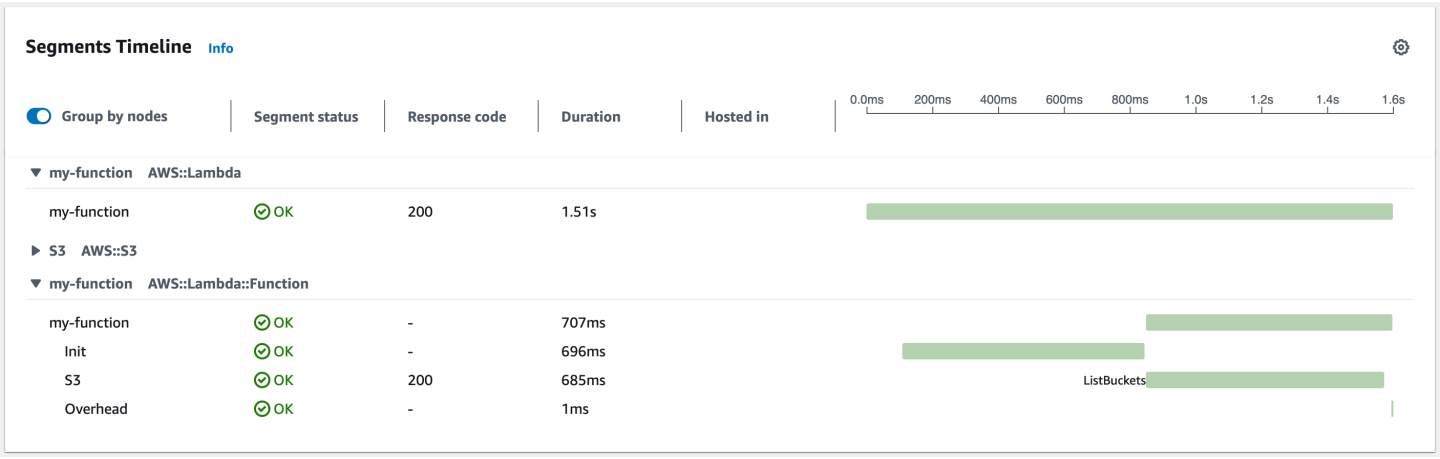


在此新格式中，`Init` 子區段和以往一樣代表 Lambda 執行環境生命週期的初始化階段。

新格式中沒有調用區段。而客戶子區段則直接連接到 `AWS::Lambda::Function` 區段。此區段包含以下指標做為註釋：

- `aws.responseLatency`：函數執行所需的時間
- `aws.responseDuration`：將回應傳輸給客戶所需的時間
- `aws.runtimeOverhead`：完成執行時期所需的額外時間
- `aws.extensionOverhead`：延伸功能完成所需的額外時間

下圖顯示了使用新式 X-Ray 結構的追蹤範例。



請注意此範例中的兩個區段。兩者都被命名為 my-function，但其中之一的來源為 `AWS::Lambda`，而另一個的來源為 `AWS::Lambda::Function`。如果 `AWS::Lambda` 區段顯示錯誤，Lambda 服務就會出現問題。如果 `AWS::Lambda::Function` 區段顯示錯誤，表示您的函數出現了問題。

如需在 Lambda 追蹤的語言特定簡介，請參閱下列主題：

- 在 中檢測 Node.js 程式碼 AWS Lambda
- 在 中檢測 Python 程式碼 AWS Lambda
- 在 中檢測 Ruby 程式碼 AWS Lambda
- 在 中檢測 Java 程式碼 AWS Lambda
- 在 中檢測 Go 程式碼 AWS Lambda
- 在 中檢測 C# 程式碼 AWS Lambda

如需支援主動設備測試之服務的完整清單，請參閱《 AWS X-Ray 開發人員指南》中的 Supported AWS 服務 一節。

Lambda 中的預設追蹤行為

若未啟用 `Active` 追蹤，Lambda 會預設為使用 `PassThrough` 追蹤模式。

在 `PassThrough` 模式下，Lambda 會將 X-Ray 追蹤標頭轉送至下游服務，但不會自動傳送追蹤。即使追蹤標頭包含對請求進行取樣的決策，情況也是如此。如果上游服務不提供 X-Ray 追蹤標頭，Lambda 會產生標頭，並決定不進行取樣。不過，您可以從函式程式碼呼叫追蹤程式庫來傳送自己的追蹤。

> **Note**
>
> 先前，當上游服務 (如 Amazon API Gateway) 新增追蹤標頭時，Lambda 會自動傳送追蹤。Lambda 不自動傳送追蹤資料，讓您能自主控制，只追蹤對自己而言重要的函式。若解決方案取決於此被動追蹤行為，請切換至 `Active` 追蹤。

執行角色許可

Lambda 需要下列許可才能傳送追蹤資料到 X-Ray。新增許可到您的函數的執行角色。

- xray:PutTraceSegments
- xray:PutTelemetryRecords

這些許可包含在 AWSXRayDaemonWriteAccess 受管政策中。

透過 Lambda API 啟用 **Active** 追蹤

若要使用 AWS CLI 或 AWS SDK 管理追蹤組態，請使用下列 API 操作：

- UpdateFunctionConfiguration
- GetFunctionConfiguration
- CreateFunction

下列範例 AWS CLI 命令會在名為 my-function 的函數上啟用主動追蹤。

```
aws lambda update-function-configuration --function-name my-function \
--tracing-config Mode=Active
```

追蹤模式是您發布函數版本時版本特定組態的一部分。您無法變更已發佈版本上的追蹤模式。

使用 啟用**Active**追蹤 CloudFormation

若要在 CloudFormation 範本中的 `AWS::Lambda::Function` 資源上啟用追蹤，請使用 `TracingConfig` 屬性。

Example function-inline.yml - 追蹤組態

```
Resources:
  function:
    Type: AWS::Lambda::Function
    Properties:
      TracingConfig:
        Mode: Active
      ...
```

對於 AWS Serverless Application Model (AWS SAM) `AWS::Serverless::Function` 資源，請使用 `Tracing` 屬性。

Example template.yml - 追蹤組態

```
Resources:
  function:
```

```
    Type: AWS::Serverless::Function
    Properties:
      Tracing: Active
      ...
```

使用 Amazon CloudWatch Lambda 洞察監控函數效能

Amazon CloudWatch Lambda Insights 會收集和彙總無伺服器應用程式的 Lambda 函數執行時間效能指標和日誌。本頁說明如何啟用和使用 Lambda Insights 來診斷 Lambda 函數的問題。

章節

- Lambda Insights 如何監控無伺服器應用程式
- 定價
- 支援的執行期
- 在 Lambda 主控台中啟用 Lambda Insights
- 以程式設計方式啟用 Lambda Insights
- 使用 Lambda Insights 儀表板
- 偵測函式異常的工作流程範例
- 使用查詢故障排除函式的範例工作流程
- 後續步驟？

Lambda Insights 如何監控無伺服器應用程式

CloudWatch Lambda Insights 是一個監控與故障排除解決方案，適用於在 AWS Lambda 上執行的無伺服器應用程式。此解決方案會收集、彙總和摘要系統層級的指標，包括 CPU 時間、記憶體、磁碟和網路使用量。它也會收集、彙總和摘要診斷資訊，例如冷啟動和 Lambda 工作人員關閉，協助您隔離 Lambda 函數問題並快速加以解決。

Lambda Insights 使用新的 CloudWatch Lambda Insights 擴展功能，提供它作為一個 Lambda 層。當您在 Lambda 函數上為支援的執行時間啟用此擴展功能時，它會收集系統層級指標，並對 Lambda 函數的每個叫用發出單一效能日誌事件。CloudWatch 使用內嵌的指標格式，從日誌事件中擷取指標。如需詳細資訊，請參閱使用AWS Lambda 延伸項目。

Lambda Insights 層會擴展 `/aws/lambda-insights/` 日誌群組的 `CreateLogStream` 和 `PutLogEvents`。

定價

當您為 Lambda 函數啟用 Lambda Insights 時，Lambda Insights 會針對每個函數報告 8 個指標，而每個函數呼叫都會傳送約 1KB 的記錄資料至 CloudWatch。您只需要為 Lambda Insights 報告的函數指

標和記錄付費。沒有最低費用或強制性的服務使用政策。如果未叫用函數，則不需為 Lambda Insights 支付費用。如需定價範例，請參閱 Amazon CloudWatch 定價。

支援的執行期

您可以使用 Lambda Insights 搭配任何支援 Lambda 擴展功能的執行時間。

在 Lambda 主控台中啟用 Lambda Insights

您可以對新函數和現有 Lambda 函數啟用 Lambda Insights 增強型監控功能。當您在 Lambda 主控台的函數中針對支援的執行時間啟用 Lambda Insights 時，Lambda 會將 Lambda Insights 擴展功能新增至您的函數，並驗證或嘗試將 `CloudWatchLambdaInsightsExecutionRolePolicy` 政策連接至函數的執行角色。

若要在 Lambda 主控台中啟用 Lambda Insights

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇函數。
3. 選擇 Configuration (組態) 索引標籤。
4. 在左側功能表中選擇監控和操作工具。
5. 在其他監控工具窗格上選擇編輯。
6. 在 CloudWatch Lambda Insights 中，開啟增強型監控。
7. 選擇儲存。

以程式設計方式啟用 Lambda Insights

您也可以使用 AWS Command Line Interface (AWS CLI)、AWS Serverless Application Model (SAM) CLI、CloudFormation 或 AWS Cloud Development Kit (AWS CDK) 啟用 Lambda Insights。當您以程式設計方式在函數中對受支援的執行時間啟用 Lambda Insights 時，CloudWatch 會將 `CloudWatchLambdaInsightsExecutionRolePolicy` 政策連接至函數的執行角色。

如需詳細資訊，請參閱 Amazon CloudWatch 使用者指南中的 Lambda Insights 入門。

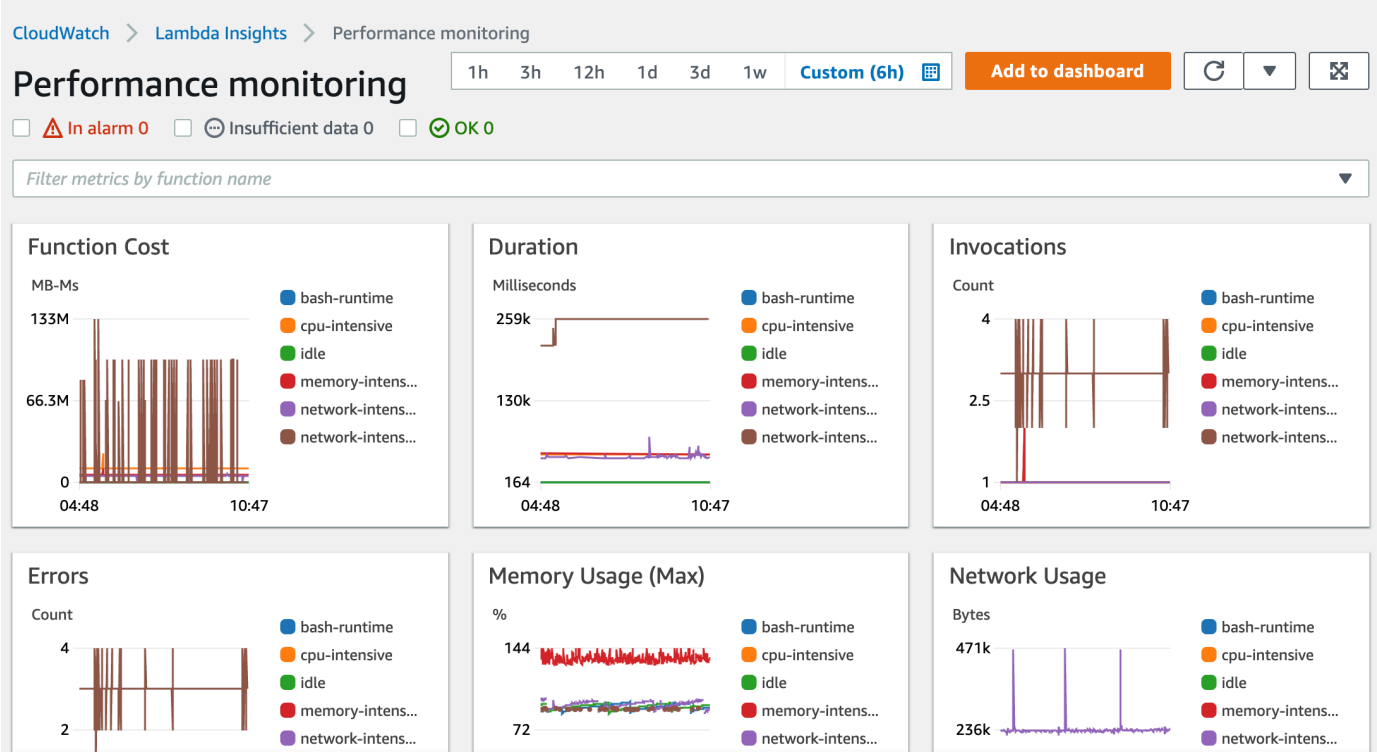
使用 Lambda Insights 儀表板

Lambda Insights 儀表板在 CloudWatch 主控台中有兩種檢視：多函數概觀和單一函數檢視。多函數概觀會彙總目前 AWS 帳戶和區域中 Lambda 函數的執行時間指標。單一函數檢視會顯示單一 Lambda 函數的可用執行時間指標。

您可以使用 CloudWatch 主控台中的 Lambda Insights 儀表板多函數概觀來識別過度使用和使用不足的 Lambda 函數。您可以使用 CloudWatch 主控台中的 Lambda Insights 儀表板單一函數檢視來對個別請求進行故障排除。

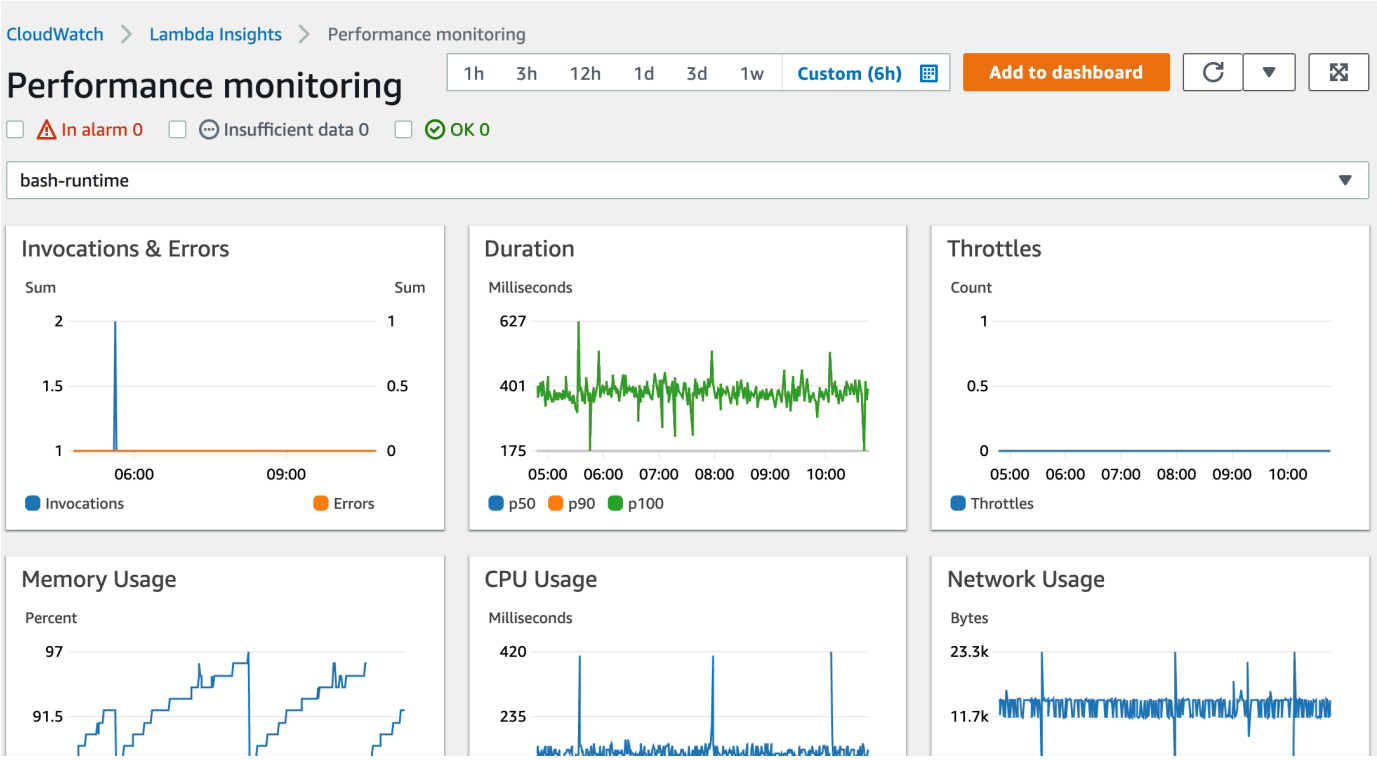
檢視所有函式的執行階段指標

1. 在 CloudWatch 主控台中開啟多函數頁面。
2. 從預先定義的時間範圍中選擇，或選擇自訂的時間範圍。
3. (選用) 選擇 Add to dashboard (新增至儀表板)，將小工具新增至 CloudWatch 儀表板。



檢視單一函式的執行階段指標

1. 在 CloudWatch 主控台中開啟單一函數頁面。
2. 從預先定義的時間範圍中選擇，或選擇自訂的時間範圍。
3. (選用) 選擇 Add to dashboard (新增至儀表板)，將小工具新增至 CloudWatch 儀表板。



如需詳細資訊，請參閱建立和使用 CloudWatch 儀表板上的小工具。

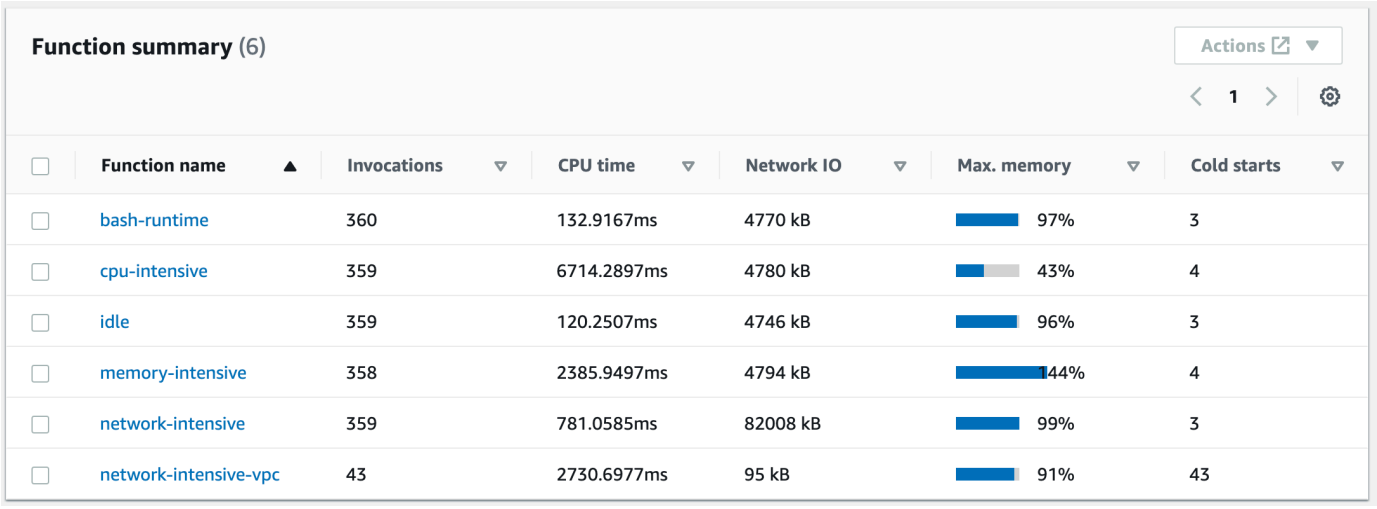
偵測函式異常的工作流程範例

您可以使用 Lambda Insights 儀表板上的多函數概觀來識別和偵測您的函數是否有運算記憶體異常。例如，如果多函式概觀指出某個函式正在使用大量記憶體，您可以在記憶體使用量窗格中檢視詳細的記憶體使用量。然後，您可以移至「指標」儀表板以啟用異常偵測或建立警示。

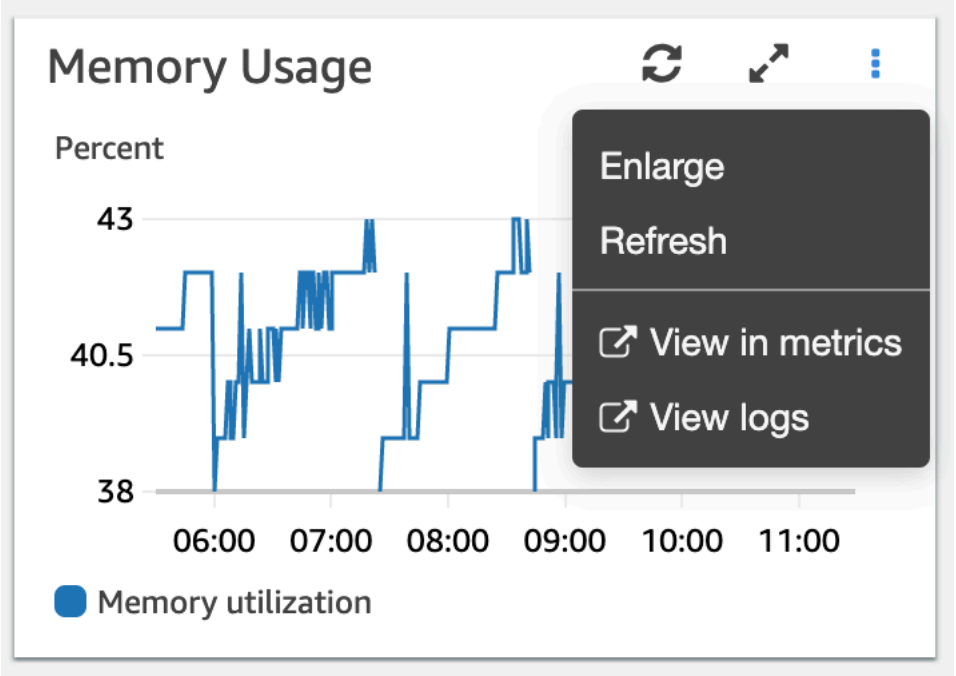
啟用對函式的異常偵測

1. 在 CloudWatch 主控台中開啟多函數頁面。
2. 在函式摘要下方，選擇您的函式名稱。

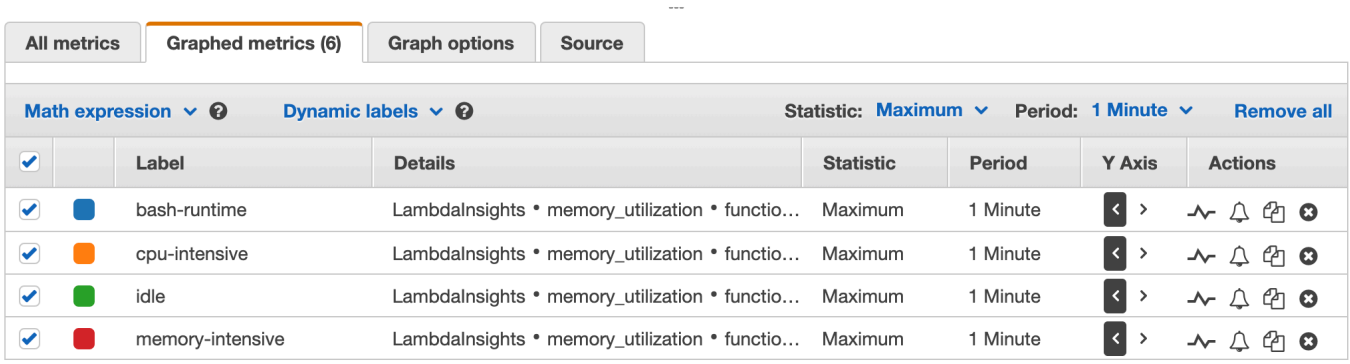
   單一函式檢視隨即開啟，其中包含函式執行階段指標。



3. 在記憶體使用量窗格中，選擇三個垂直點，然後選擇在指標中檢視以開啟指標儀表板。



4. 在圖形化指標索引標籤的動作欄中，選擇第一個圖示，以啟用對函式的異常偵測。



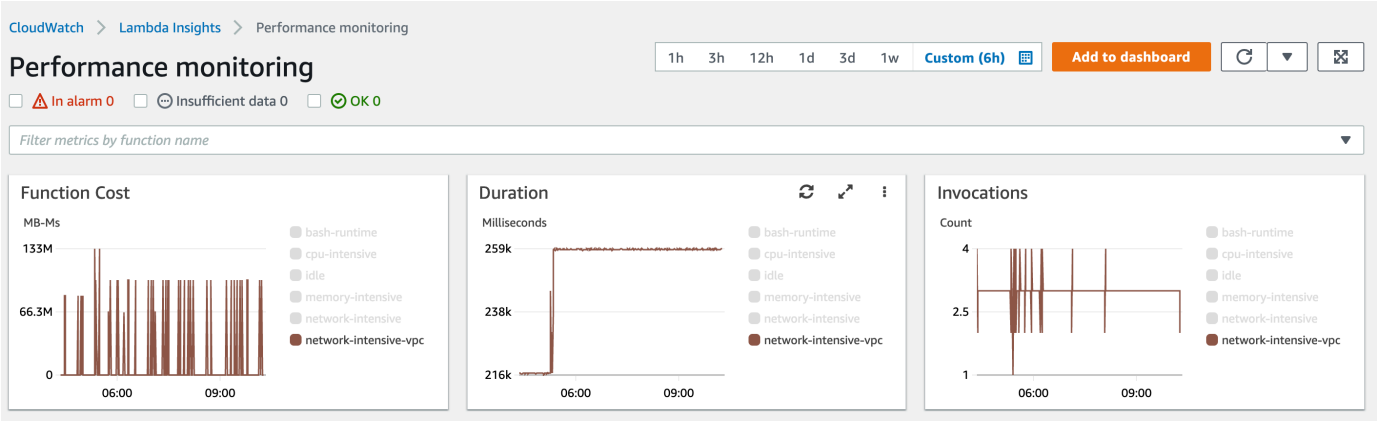
如需詳細資訊，請參閱使用 CloudWatch 異常偵測。

使用查詢故障排除函式的範例工作流程

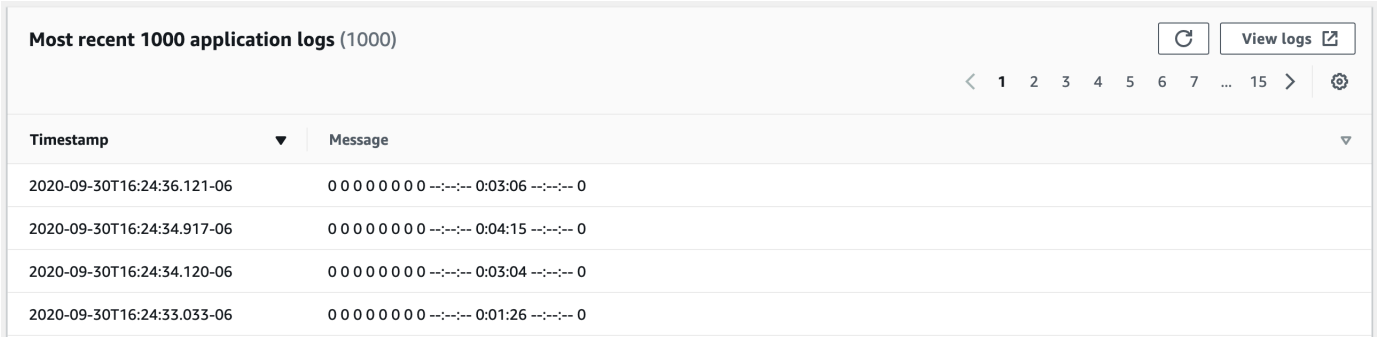
您可以使用 Lambda Insights 儀表板上的單一函數檢視來識別函數持續時間中峰值的根本原因。例如，如果多函式概觀指出函式持續時間大幅增加，您可以暫停或選擇持續時間窗格中的每個函式，以判斷哪個函式造成增加。然後，您可以移至單一函式檢視並檢閱應用程式日誌，以判斷根本原因。

在函式上執行查詢

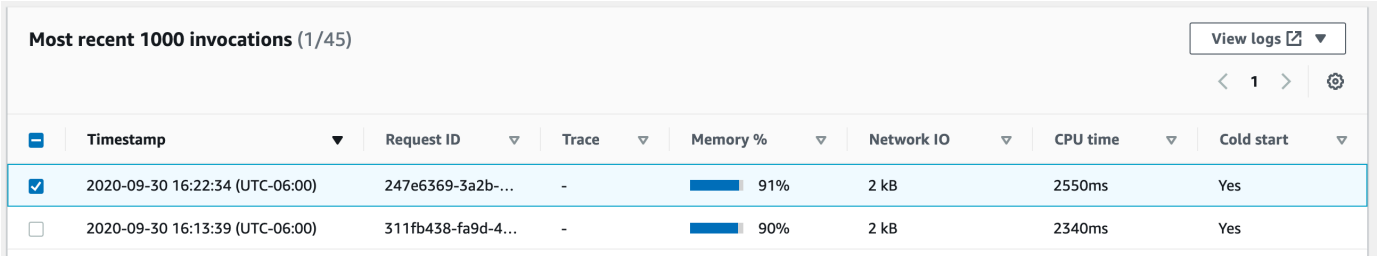
1. 在 CloudWatch 主控台中開啟多函數頁面。
2. 在持續時間窗格中，選擇您的函式以篩選持續時間量度。



3. 開啟單一函式頁面。
4. 選擇依函式名稱篩選指標下拉式清單，然後選擇您的函式。
5. 若要檢視 最近 1000 個應用程式日誌，請選擇應用程式日誌索引標籤。
6. 檢閱時間戳記和訊息，以識別您要故障排除的叫用請求。



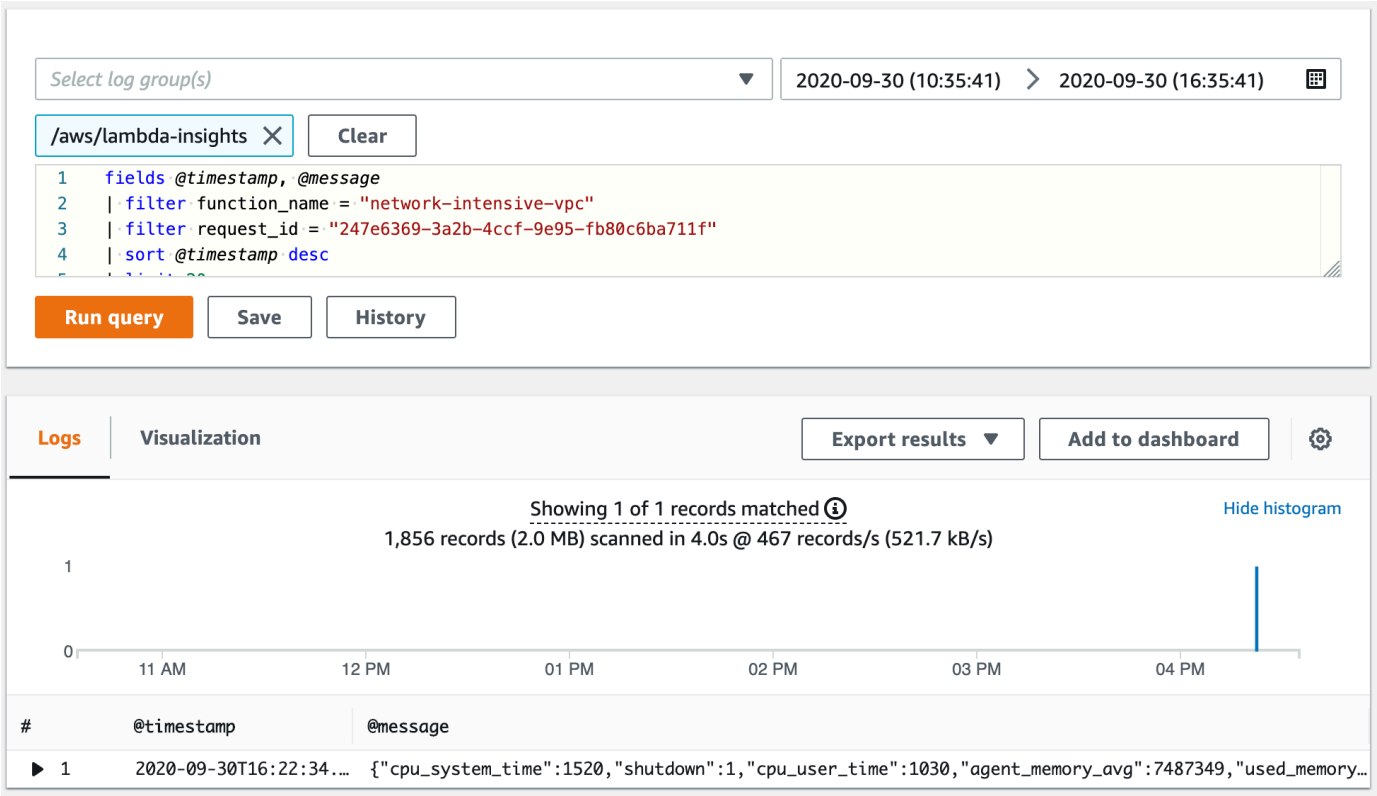
7. 若要顯示最近的 1000 次叫用，請選擇叫用索引標籤。
8. 針對您要故障排除的叫用請求，選取時間戳記或訊息。



9. 選擇檢視日誌的下拉式清單，然後選擇檢視效能日誌。

   您的函式自動產生的查詢會在日誌深入資訊儀表板中開啟。

10. 選擇執行查詢以產生叫用請求的日誌訊息。



後續步驟？

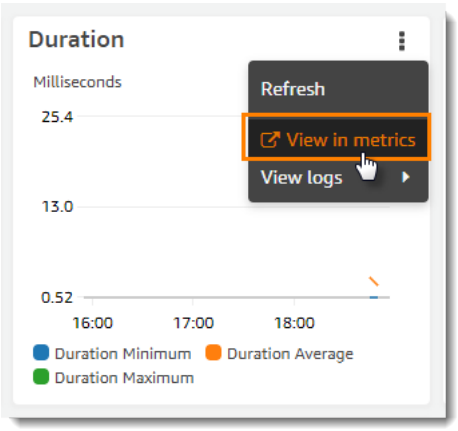
- 了解如何在 Amazon CloudWatch 使用者指南中的建立儀表板中建立 CloudWatch Logs 儀表板。
- 了解如何在 Amazon CloudWatch 使用者指南中的新增查詢至儀表板或匯出查詢結果中將查詢新增至 CloudWatch Logs 儀表板。

監控 Lambda 應用程式

Lambda 主控台的應用程式區段包含監控索引標籤，可以在其中檢閱 Amazon CloudWatch 儀表板，其中包含應用程式中資源的彙總指標。

若要監控 Lambda 應用程式

1. 開啟 Lambda 主控台中的 Applications (應用程式) 頁面。
2. 選擇 Monitoring (監控)。
3. 若要查看任何圖形中指標的詳細資訊，請從下拉式功能表中選擇在指標中檢視。



   圖形會出現在新的標籤中，相關的指標會列在圖形下方。您可以自訂此圖形的檢視，包括變更所顯示的指標和資源、統計、期間和其他因素，以更加了解目前的情況。

在預設情況下，Lambda 主控台會顯示基本儀表板。透過將一個或多個 Amazon CloudWatch 儀表板新增至您的應用程式範本，可使用 AWS::CloudWatch::Dashboard 資源類型自訂此頁面。當您的範本包含一或多個儀表板時，此頁面會顯示您的儀表板，而非預設的儀表板。您可以使用頁面右上角的下拉式功能表切換不同的儀表板。以下範例以單一小工具建立儀表板，此工具可繪製呈現 `my-function` 函數叫用次數的圖形。

Example函數儀表板範本

```
Resources:
  MyDashboard:
    Type: AWS::CloudWatch::Dashboard
    Properties:
      DashboardName: my-dashboard
      DashboardBody: |
        {
```

```
            "widgets": [
                {
                    "type": "metric",
                    "width": 12,
                    "height": 6,
                    "properties": {
                        "metrics": [
                            [
                                "AWS/Lambda",
                                "Invocations",
                                "FunctionName",
                                "my-function",
                                {
                                    "stat": "Sum",
                                    "label": "MyFunction"
                                }
                            ],
                            [
                                {
                                    "expression": "SUM(METRICS())",
                                    "label": "Total Invocations"
                                }
                            ]
                        ],
                        "region": "us-east-1",
                        "title": "Invocations",
                        "view": "timeSeries",
                        "stacked": false
                    }
                }
            ]
        }
```

如需編寫 CloudWatch 儀表板和小工具的詳細資訊，請參閱 Amazon CloudWatch API 參考中的儀表板主體結構和語法。

使用 Amazon CloudWatch Application Signals 監控應用程式效能

Amazon CloudWatch Application Signals 是應用程式效能監控 (APM) 解決方案，可讓開發人員和操作人員監控使用 Lambda 建置之無伺服器應用程式的運作狀態和效能。您可以從 Lambda 主控台一鍵啟用 Application Signals，且無需將任何設備測試代碼或外部相依關係新增至 Lambda 函數。啟用 Application Signals 後，您可以在 CloudWatch 主控台中檢視收集的所有指標和追蹤。此頁面說明如何為應用程式啟用和檢視 Application Signals 遙測資料。

主題

- Application Signals 與 Lambda 的整合方式
- 定價
- 支援的執行期
- 在 Lambda 主控台中啟用 Application Signals
- 使用 Application Signals 儀表板

Application Signals 與 Lambda 的整合方式

Application Signals 會使用透過 Lambda 層提供的增強型 AWS Distro for OpenTelemetry (ADOT) 程式庫，自動檢測您的 Lambda 函數。Application Signals 會讀取層收集的資料，並為您的應用程式產生關鍵效能指標的儀表板。

可以在 Lambda 主控台中啟用 Application Signals，一鍵連接此層。從主控台啟用 Application Signals 時，Lambda 會代表您執行下列動作：

- 更新函數的執行角色以包含 `CloudWatchLambdaApplicationSignalsExecutionRolePolicy`。此政策提供用於 Application Signals 的 AWS X-Ray 和 CloudWatch 日誌群組的寫入存取權。
- 將層新增至您的函數，以自動檢測函數並擷取遙測資料，例如請求、可用性、延遲、錯誤和故障。若要確定 Application Signals 正常運作，請從函數中移除現有的任何 X-Ray SDK 設備測試代碼。自訂 X-Ray SDK 設備測試代碼可能會干擾層提供的設備測試。
- 將 `AWS_LAMBDA_EXEC_WRAPPER` 環境變數新增至您的函數，並將其值設定為 `/opt/otel-instrument`。此環境變數會修改函數的啟動行為以利用 Application Signals 層，這是進行適當設備測試的必要動作。如果此環境變數已存在，則請確定它已設定為所需的值。

定價

對 Lambda 函數使用 Application Signals 會產生成本。如需定價資訊，請參閱 Amazon CloudWatch pricing。

支援的執行期

Application Signals 與 Lambda 的整合適用於下列執行時期：

- .NET 8
- Java 11
- Java 17
- Java 21
- Python 3.10
- Python 3.11
- Python 3.12
- Python 3.13
- Node.js 18.x
- Node.js 20.x
- Node.js 22.x

在 Lambda 主控台中啟用 Application Signals

您可以使用支援的執行時期，在任何現有的 Lambda 函數上啟用 Application Signals。下列步驟說明如何在 Lambda 主控台中一鍵啟用 Application Signals。

若要在 Lambda 主控台中啟用 Application Signals

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函數。
3. 選擇 Configuration (組態) 索引標籤。
4. 在左側功能表中選擇監控和操作工具。
5. 在其他監控工具窗格上選擇編輯。
6. 在 CloudWatch Application Signals 和 AWS X-Ray 以及 Application Signals 下，選擇啟用。
7. 選擇儲存。

如果這是第一次為函數啟用 Application Signals，則您還必須在 CloudWatch 主控台中為 Application Signals 執行一次性服務探索設定。完成此一次性服務探索設定後，Application Signals 會自動探索您在所有區域中啟用 Application Signals 的任何其他 Lambda 函數。

> **Note**
>
> 調用更新的函數之後，服務資料最多可能需要 10 分鐘才會開始顯示在 CloudWatch 主控台的 Application Signals 儀表板中。

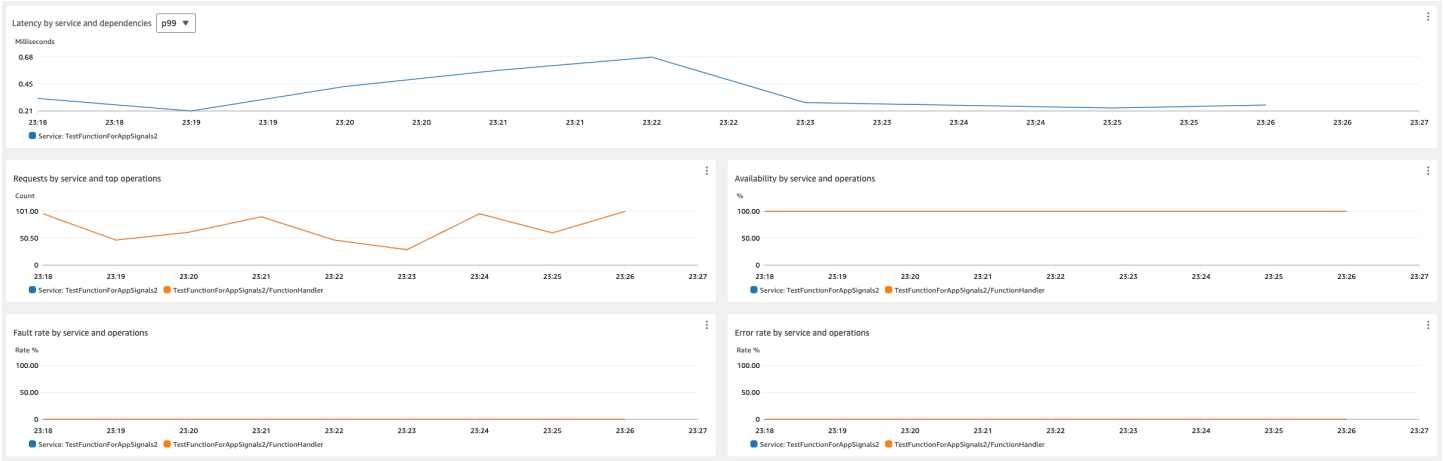
使用 Application Signals 儀表板

啟用函數的 Application Signals 之後，可以在 CloudWatch 主控台中視覺化應用程式指標。可以透過下列步驟，從 Lambda 主控台快速檢視關聯的 Application Signals 儀表板：

若要檢視函數的 Application Signals 儀表板

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函數。
3. 選擇 Monitor (監控) 索引標籤。
4. 選擇檢視 Application Signals 按鈕。系統會直接帶您檢視 CloudWatch 主控台中服務的 Application Signals 概觀。

例如，下列螢幕擷取畫面會顯示 10 分鐘時間範圍內函數的延遲、請求數、可用性、故障率和錯誤率指標。



若要充分利用與 Application Signals 的整合，可以為應用程式建立服務層級目標 (SLO)。例如，可以建立延遲 SLO 以確定應用程式快速回應使用者請求，以及追蹤運行時間的可用性 SLO。SLO 可協助您在效能下降或中斷影響使用者之前偵測到它們。如需詳細資訊，請參閱《Amazon CloudWatch 使用者指南》中的服務層級目標 (SLO)。

使用 Visual Studio Code 遠端偵錯 Lambda 函式

透過 AWS Toolkit for Visual Studio Code 中的遠端偵錯功能，您可以對直接在 AWS 雲端中執行的 Lambda 函式進行偵錯。此功能在調查難以在本機重現，或僅透過日誌難以診斷的問題時相當實用。

透過遠端偵錯功能，您可以：

- 在 Lambda 函式程式碼中設定中斷點。
- 即時逐步執行程式碼。
- 在執行時期內檢查變數與狀態。
- 偵錯部署至 AWS 的 Lambda 函式，包括 VPC 中或具有特定 IAM 許可的函式。

支援的執行期

下列執行時期支援遠端偵錯功能：

- Python (AL2023)
- Java
- JavaScript/Node.js (AL2023)

> **Note**
>
> x86_64 與 arm64 架構都支援遠端偵錯功能。

安全性與遠端偵錯

遠端偵錯功能會在現有的 Lambda 安全邊界內運作。使用者可以透過 `UpdateFunctionConfiguration` 許可將層連接至函式，該許可已經能夠存取函式環境變數與組態。遠端偵錯功能不會超出現有許可範圍，而是會透過安全通道與自動工作階段管理來新增額外的安全控制項。此外，遠端偵錯功能完全由客戶控制，需要明確的許可與動作：

- IoT 安全通道建立：AWS Toolkit 必須建立一個 IoT 安全通道，這只有在使用者明確許可時才能使用 `iot:OpenTunnel` 執行此動作。
- 偵錯層連接與權杖管理：偵錯程序會透過下列控制項維持安全性：

- 偵錯層必須連接至 Lambda 函式，此程序需要下列許可：`lambda:UpdateFunctionConfiguration` 與 `lambda:GetLayerVersion`。
- 在每個偵錯工作階段開始之前，必須更新函式環境變數中的安全權杖 (透過 `iot:OpenTunnel` 產生)，此程序也需要 `lambda:UpdateFunctionConfiguration`。
- 出於安全考量，此權杖會自動輪換，偵錯層也會在每個偵錯工作階段結束時自動移除，無法重複使用。

> **Note**
>
> x86_64 與 arm64 架構都支援遠端偵錯功能。

必要條件

開始遠端偵錯之前，確認已滿足以下條件：

1. Lambda 函式已部署至您的 AWS 帳戶。
2. AWS Toolkit for Visual Studio Code。如需安裝說明，請參閱 Setting up the AWS Toolkit for Visual Studio Code。
3. 安裝的 AWS Toolkit 為 3.69.0 或更新版本。
4. 已在 AWS Toolkit for Visual Studio Code 中設定 AWS 憑證。如需更多詳細資訊，請參閱 身分驗證與存取控制。

遠端偵錯 Lambda 函式

請依照下列步驟啟動遠端偵錯工作階段：

1. 選擇左側邊欄中的 AWS 圖示，在 VS Code 中開啟 AWS Explorer。
2. 展開 Lambda 區段檢視函式。
3. 在要偵錯的函式上按一下滑鼠右鍵。
4. 從內容功能表中，選擇遠端調用。
5. 在開啟的調用視窗中，勾選啟用偵錯方塊。
6. 按一下調用，啟動遠端偵錯工作階段。

> **Note**
>
> Lambda 函式的函式程式碼與所有連接層，合併大小上限為 250 MB。遠端偵錯層可使函式大小增加約 40 MB。

當您執行下列操作時，遠端偵錯工作階段將結束：

- 在遠端調用組態畫面中，選擇移除偵錯設定。
- 在 VS Code 偵錯控制項中，點選中斷連線圖示。
- 在 VS Code 編輯器中，選取處理常式檔案。

> **Note**
>
> 在上次調用結束後，若閒置超過 60 秒，系統會自動移除偵錯層。

停用遠端偵錯

有三種方式可以停用此功能：

- 拒絕函式更新：將 `lambda:UpdateFunctionConfiguration` 設定為 deny。
- 限制 IoT 許可：拒絕 IoT 相關許可
- 封鎖偵錯層：拒絕下列 ARN 的 `lambda:GetLayerVersion`：
  - `arn:aws:lambda:*:*:layer:LDKLayerX86:*`
  - `arn:aws:lambda:*:*:layer:LDKLayerArm64:*`

> **Note**
>
> 停用此功能可防止在函式組態更新期間新增偵錯層。

其他資訊

如需有關在 VS Code 中使用 Lambda 的詳細資訊，請參閱使用 VS Code 在本機開發 Lambda 函式。

如需有關疑難排解、進階使用案例與區域可用性的詳細說明，請參閱 AWS Toolkit for Visual Studio Code User Guide 中的 Remote debugging Lambda functions。

使用層管理 Lambda 相依項

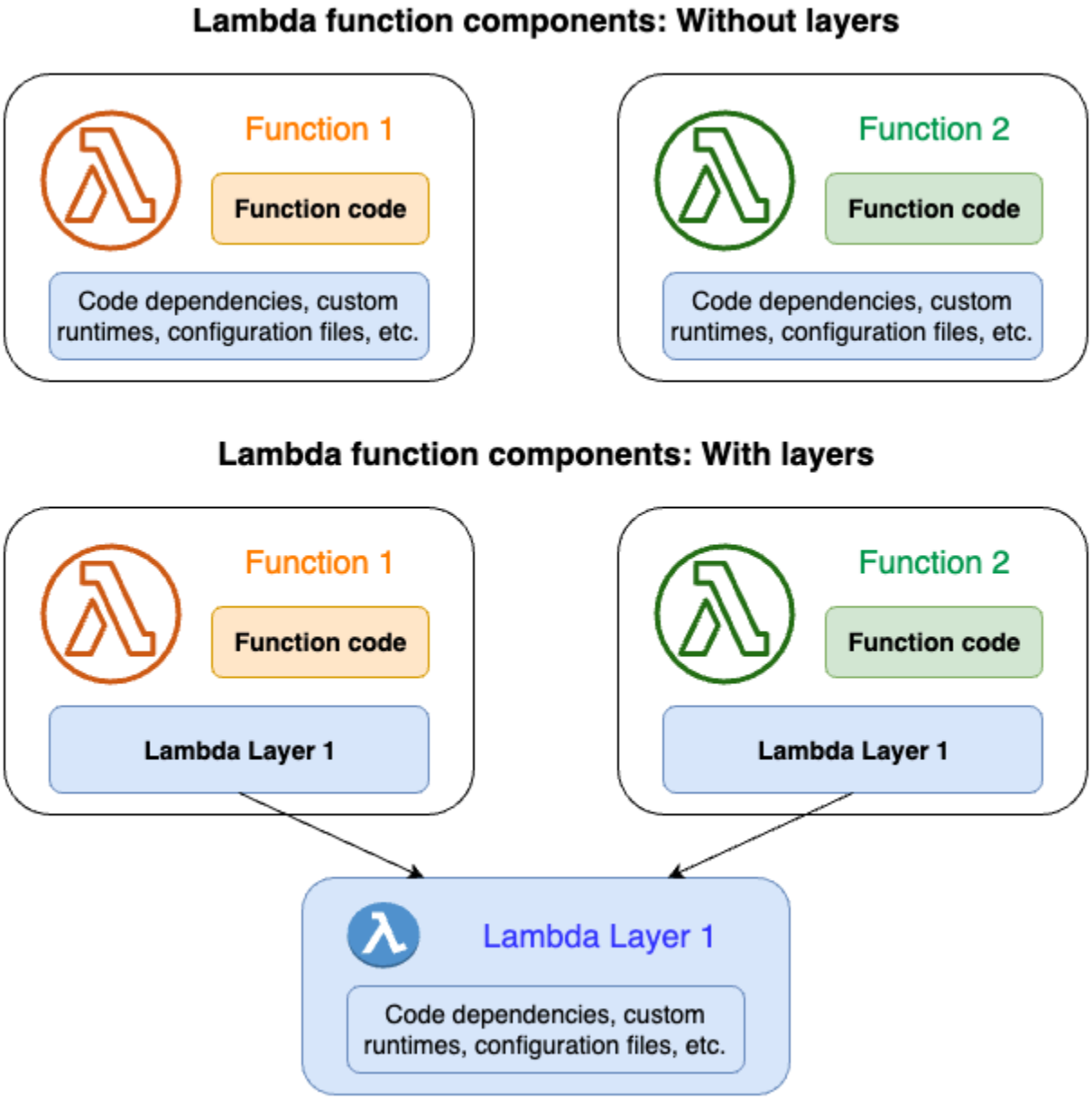
Lambda 層是含有補充程式碼或資料的 .zip 封存檔。層通常具備程式庫相依性、自訂執行期或組態檔案。

以下是您可能會考慮使用層的多種原因：

- 縮減部署套件的大小。切勿將所有函數的相依項以及函數程式碼加入部署套件，而是將它們放在層裡面。這可以使部署套件在容量小的情況下同時保持條理。
- 若要將核心函數邏輯與相依項分隔開來。您可以透過層獨立於函數程式碼更新函數相依項，反之亦然。這能夠促進關注點分離的原則，且有助於您將重心放在函數邏輯上。
- 若要跨多個函數共享相依項。建立層後，您可以將其套用於帳戶中的函數，數量無任何限制。如果沒有使用層，則必須在每個個別的部署套件中加入相同的相依項。
- 若要使用 Lambda 主控台程式碼編輯器。程式碼編輯器是快速測試次要函數程式碼更新的實用工具。不過，如果您的部署套件太大，便無法使用編輯器。使用層可以縮減套件的大小，並取得程式碼編輯器的使用權限。
- 鎖定嵌入式 SDK 版本。隨著 AWS 發布新服務與功能，嵌入式 SDK 可能會在未經通知的情況下變更。您可以使用所需的特定版本建立 Lambda 層，鎖定 SDK 的版本。如此一來，即使內嵌於服務中的版本變更，函數也會永遠使用層中的版本。

如果您在 Go 或 Rust 中使用 Lambda 函數，則建議不要使用層。對於 Go 和 Rust 函數，可以提供函數程式碼作為可執行檔，其中包含編譯的函數程式碼及其所有相依項。將相依項放在層中會強制函數在初始化階段期間手動載入其他組件，這可能會增加冷啟動時間。為了獲得最佳的 Go 和 Rust 函數效能，請包含您的相依項以及部署套件。

下圖會說明共用相依項的兩個函數之間的概略架構差異。一個函數使用 Lambda 層，而另一個函數則不使用。



將層新增至 Lambda 函數時，Lambda 會將層內容擷取至函數執行環境中的 `/opt` 目錄。所有原生支援的 Lambda 執行期皆包含 `/opt` 目錄中特定目錄的路徑。如此一來，您的函數便可以存取您的層內容。如需有關這類特定路徑以及如何正確封裝層的詳細資訊，請參閱 the section called “封裝層”。

每個函數最多可包含五個圖層。此外，您只能將層與 部署為 .zip 封存檔 的 Lambda 函數搭配使用。對於定義為容器映像的函數，您可以在建立容器映像時封裝偏好的執行期和所有程式碼相依項。如需詳細資訊，請參閱 AWS 運算部落格的在容器映像中使用 Lambda 圖層和擴展功能。

主題

- 如何使用層
- 層和層的版本
- 封裝層內容
- 在 Lambda 中建立和刪除層
- 為函數新增層
- AWS CloudFormation 與圖層搭配使用
- AWS SAM 與圖層搭配使用

如何使用層

若要建立層，請將相依項封裝到 .zip 檔案中，方法類似於您建立一般部署套件的方式。更具體來說，建立和使用層的一般程序包括以下三個步驟：

- 首先，封裝層內容。這表示您必須建立一個 .zip 封存檔。如需更多詳細資訊，請參閱 the section called “封裝層”。
- 接著，在 Lambda 中建立層。如需更多詳細資訊，請參閱 the section called “建立和刪除層”。
- 將層新增到您的函數中。如需更多詳細資訊，請參閱 the section called “新增層”。

層和層的版本

層版本是特定層版本不可變的快照。建立新層時，Lambda 會建立版本編號為 1 的新層版本。每次將更新發佈至層時，Lambda 都會遞增版本編號並建立新的層版本。

每個層版本皆由唯一的 Amazon Resource Name (ARN) 進行識別。向函式新增層時，必須指定要使用的層的確切版本 (例如 `arn:aws:lambda:us-east-1:123456789012:layer:my-layer:1`)。

封裝層內容

Lambda 層是含有補充程式碼或資料的 .zip 封存檔。層通常具備程式庫相依性、自訂執行期或組態檔案。

本節會說明如何正確封裝層內容。若要進一步了解有關層的概念性資訊以及您可能會考慮使用的原因，請參閱 Lambda 層。

建立層的第一步是將所有層內容綁定至 .zip 封存檔。由於 Lambda 函數是在 Amazon Linux 上執行，因此您的層內容必須能夠在 Linux 環境中編譯和建置。

為確保層內容在 Linux 環境中正常運作，建議使用 Docker 等工具來建立層內容。

主題

- 每個 Lambda 執行時間的層路徑

每個 Lambda 執行時間的層路徑

將層新增至函數時，Lambda 會將層內容載入該執行環境的 `/opt` 目錄。在每一次 Lambda 執行期中，`PATH` 變數已包含 `/opt` 目錄中的特定資料夾路徑。為確保 Lambda 能夠擷取層內容，層 .zip 檔案應該將相依項置於以下任一資料夾路徑中：

執行期	路徑
Node.js	`nodejs/node_modules`
	`nodejs/node18/node_modules` (`NODE_PATH`)
	`nodejs/node20/node_modules` (`NODE_PATH`)
	`nodejs/node22/node_modules` (`NODE_PATH`)
Python	`python`
	`python/lib/`*`python3.x`*`/site-packages` (網站目錄)
Java	`java/lib` (`CLASSPATH`)
Ruby	`ruby/gems/3.4.0` (`GEM_PATH`)

執行期	路徑
	ruby/lib (RUBYLIB)
所有執行時間	bin (PATH)
	lib (LD_LIBRARY_PATH)

下列範例展示如何在圖層 .zip 封存中建構資料夾。

Node.js

Example適用於 Node.js 的 AWS X-Ray 開發套件的檔案結構

```
xray-sdk.zip
# nodejs/node_modules/aws-xray-sdk
```

Python

Example

```
python/              # Required top-level directory
### requests/
### boto3/
### numpy/
### (dependencies of the other packages)
```

Ruby

Example JSON gem 的檔案結構

```
json.zip
# ruby/gems/3.4.0/
                | build_info
                | cache
                | doc
                | extensions
                | gems
                | # json-2.1.0
                # specifications
```

```
                # json-2.1.0.gemspec
```

Java

Example Jackson JAR 檔案的檔案結構

```
layer_content.zip
# java
    # lib
        # jackson-core-2.17.0.jar
        # <other potential dependencies>
        # ...
```

All

Example JQ 程式庫的檔案結構

```
jq.zip
# bin/jq
```

如需封裝、建立和新增層的語言特定說明，請參閱下列頁面：

- Node.js – the section called “層”
- Python – the section called “層”
- Ruby – the section called “層”
- Java – the section called “層”

對於以 Go 和 Rust 編寫的 Lambda 函式，不建議使用層來管理相依項。這是因為以此類語言編寫的 Lambda 函式會編譯為單一可執行檔，並在部署函式時提供給 Lambda。此可執行檔包含經過編譯的函數程式碼及其所有相依項。使用層不僅會使程序複雜化，還會導致冷啟動時間增加，因為函數需要在初始化階段期間將額外的組件載入記憶體。

若要在 Go 和 Rust Lambda 函式中使用外部相依項，請將其直接包含在部署套件中。

在 Lambda 中建立和刪除層

Lambda 層是含有補充程式碼或資料的 .zip 封存檔。層通常具備程式庫相依性、自訂執行期或組態檔案。

本節會說明如何在 Lambda 中建立和刪除層。若要進一步了解有關層的概念性資訊以及您可能會考慮使用的原因，請參閱 Lambda 層。

封裝層內容 後，下一步是在 Lambda 中建立層。本節會示範如何僅使用 Lambda 主控台或 Lambda API 建立和刪除層。若要使用 AWS CloudFormation 建立層，請參閱 the section called “具有 CloudFormation 的層”。若要使用 AWS Serverless Application Model (AWS SAM) 建立層，請參閱 the section called “具有 AWS SAM 的層”。

主題

- 建立圖層
- 刪除圖層版本

建立圖層

若要建立層，您可以從本機電腦或 Amazon Simple Storage Service (Amazon S3) 中上傳 .zip 封存檔。設定函數的執行環境時，Lambda 會將層內容擷取到 `/opt` 目錄中。

層可以有一個或多個 層版本。建立層時，Lambda 將層版本設定為版本 1。您可以隨時變更既有層版本的許可。不過，若要更新程式碼或進行其他組態變更，您必須建立新的層版本。

建立圖層 (主控台)

1. 開啟 Lambda 主控台中的 層頁面。
2. 選擇 建立圖層。
3. 在 Layer configuration (圖層組態) 下，為 Name (名稱) 輸入圖層的名稱。
4. (選用) 在 Description (說明) 中，輸入 Layer 的說明。
5. 若要上傳 Layer 程式碼，請執行下列其中一個動作：
   - 若要從電腦上傳 .zip 檔案，請選擇 Upload a .zip file (上傳 .zip 檔案)。然後，選擇 Upload (上傳) 以選取您的本機 .zip 檔案。
   - 若要從 Amazon S3 上傳檔案，請選擇 Upload a file from Amazon S3 (從 Amazon S3 上傳檔案)。然後，對於 Amazon S3 連結 URL，輸入檔案的連結。

6. (選用) 對於相容架構，選擇一個值或兩個值。如需詳細資訊，請參閱the section called “指令集 (ARM/x86)”。
7. (選擇性) 在 相容執行期 中選擇相容於您的層的執行期。
8. (選擇性) 在 License (授權) 中，輸入任何必要的授權資訊。
9. 選擇建立。

或者，您可以執行 publish-layer-version AWS Command Line Interface (CLI) 命令。範例：

```
aws lambda publish-layer-version --layer-name my-layer --zip-file fileb://layer.zip --
compatible-runtimes python3.14
```

每當您執行 `publish-layer-version` 時，Lambda 都會建立新的層版本。

刪除圖層版本

若要刪除層版本，請使用 DeleteLayerVersion API 操作。例如，以指定的 layer 名稱和 layer 版本執行 delete-layer-version AWS CLI 命令。

```
aws lambda delete-layer-version --layer-name my-layer --version-number 1
```

刪除層版本之後，您無法再設定 Lambda 函數以便使用它。不過，凡已使用該版本的任何函式均能繼續對其進行存取。此外，Lambda 永遠不會重複使用層名稱的版本編號。

計算配額時，刪除層版本意味著它不會再計入儲存函數和層的預設 75 GB 配額。不過，對於使用已刪除層版本的函數，層內容仍會計入函數的部署套件大小配額 (即 .zip 檔案封存為 250 MB)。

為函數新增層

Lambda 層是含有補充程式碼或資料的 .zip 封存檔。層通常具備程式庫相依性、自訂執行期或組態檔案。

本節會說明如何將層新增至 Lambda 函數。若要進一步了解有關層的概念性資訊以及您可能會考慮使用的原因，請參閱 Lambda 層。

您必須先執行下列動作，才能設定 Lambda 函數以使用層：

- 封裝層內容
- 在 Lambda 中建立層
- 確認您擁有對層版本呼叫 GetLayerVersion API 的許可。對於 中的函數 AWS 帳戶，您必須在使用者政策中擁有此許可。若要在其他帳號中使用圖層，其他帳戶的擁有者必須在資源型策略中授與您的帳戶許可。如需範例，請參閱 the section called “其他帳戶的層存取權”。

您最多可以將五個層新增至 Lambda 函數。函數和所有圖層的解壓縮大小總計不得超過解壓縮部署套件大小 250 MB 的配額。如需詳細資訊，請參閱Lambda 配額。

您的函數可以繼續使用您已新增的任何層版本，即使該層版本已被刪除，或您存取層的許可被撤銷後也是如此。但是，您不能建立使用已刪除圖層版本的新函數。

將層新增至函式

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇函式。
3. 向下捲動至層區段，然後選擇新增層。
4. 在選擇層下方選擇層來源：
   a. AWS layer：從 AWS受管擴充功能清單中選擇。
   b. 自訂層：選擇在您 AWS 帳戶中建立的層。
   c. 指定 ARN：若要使用來自不同 AWS 帳戶 的 layer，例如第三方延伸模組，請輸入 Amazon Resource Name (ARN)。
5. 選擇新增。

新增層的順序即 Lambda 將層內容合併至執行環境的順序。您可以使用主控台來變更層合併順序。

若要更新函數的層合併順序 (主控台)

1. 開啟 Lambda 主控台中的函數頁面。
2. 選擇要設定的函數。
3. 在 Layers (層) 下方，選擇 Edit (編輯)
4. 選擇其中一個層。
5. 選擇 Merge earlier (先前合併) 或 Merge later (稍後合併) 來調整層的順序。
6. 選擇儲存。

層已設定版本控制。每個層版本的內容都是不可變的。層擁有者可發行新的層版本，以提供更新內容。您可以使用主控台來更新函數附加的層版本。

若要更新函數的層版本 (主控台)

1. 開啟 Lambda 主控台中的層頁面。
2. 選擇您要更新版本的層。
3. 選擇使用此版本的函數標籤。
4. 選擇您要修改的函數，然後選擇編輯。
5. 在層版本中選擇要變更的層版本。
6. 選擇 Update functions (更新函數)。

您無法跨 AWS 帳戶更新函數層版本。

尋找圖層資訊

若要在您的帳戶中尋找與函數執行期相容的層，請使用 ListLayers API。例如，您可以使用下列 list-layers AWS Command Line Interface (CLI) 命令：

```
aws lambda list-layers --compatible-runtime python3.14
```

您應該會看到類似下列的輸出：

```
{
    "Layers": [
        {
            "LayerName": "my-layer",
```

```
            "LayerArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-layer",
            "LatestMatchingVersion": {
                "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
                "Version": 2,
                "Description": "My layer",
                "CreatedDate": "2025-04-15T00:37:46.592+0000",
                "CompatibleRuntimes": [
                    "python3.14"
                ]
            }
        }
    ]
}
```

若要在您的帳戶中列出所有層，請忽略 `--compatible-runtime` 選項。回應詳細資訊會顯示各個層的最新版本。

您還可使用 ListLayerVersions API 取得層的最新版本。舉例來說，您可以使用下列 `list-layer-versions` CLI 命令：

```
aws lambda list-layer-versions --layer-name my-layer
```

您應該會看到類似下列的輸出：

```
{
    "LayerVersions": [
        {
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:2",
            "Version": 2,
            "Description": "My layer",
            "CreatedDate": "2023-11-15T00:37:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        },
        {
            "LayerVersionArn": "arn:aws:lambda:us-east-2:123456789012:layer:my-
layer:1",
            "Version": 1,
            "Description": "My layer",
```

```
            "CreatedDate": "2023-11-15T00:27:46.592+0000",
            "CompatibleRuntimes": [
                "java11"
            ]
        }
    ]
}
```

AWS CloudFormation 與圖層搭配使用

您可以使用 CloudFormation 建立層，並將層與 Lambda 函數相關聯。下列範例範本會建立名為 `my-lambda-layer` 的層，並使用 Layers 屬性將該層連接至 Lambda 函數。

在此範例中，範本指定現有 IAM 執行角色的 Amazon Resource Name (ARN)。也可以使用 CloudFormation AWS::IAM::Role 資源在範本中建立新的執行角色。

您的函數不需要任何特殊許可即可使用層。

```
---
Description: CloudFormation Template for Lambda Function with Lambda Layer
Resources:
  MyLambdaLayer:
    Type: AWS::Lambda::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      Content:
        S3Bucket: amzn-s3-demo-bucket
        S3Key: my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Lambda::Function
    Properties:
      FunctionName: my-lambda-function
      Runtime: python3.9
      Handler: index.handler
      Timeout: 10
      Role: arn:aws:iam::111122223333:role/my_lambda_role
      Layers:
        - !Ref MyLambdaLayer
```

AWS SAM 與圖層搭配使用

您可以使用 AWS Serverless Application Model (AWS SAM) 在應用程式中自動建立層。`AWS::Serverless::LayerVersion` 資源類型會建立可從 Lambda 函數組態參考的圖層版本。

```
AWSTemplateFormatVersion: '2010-09-09'
Transform: 'AWS::Serverless-2016-10-31'
Description: AWS SAM Template for Lambda Function with Lambda Layer

Resources:
  MyLambdaLayer:
    Type: AWS::Serverless::LayerVersion
    Properties:
      LayerName: my-lambda-layer
      Description: My Lambda Layer
      ContentUri: s3://amzn-s3-demo-bucket/my-layer.zip
      CompatibleRuntimes:
        - python3.9
        - python3.10
        - python3.11

  MyLambdaFunction:
    Type: AWS::Serverless::Function
    Properties:
      FunctionName: MyLambdaFunction
      Runtime: python3.9
      Handler: app.handler
      CodeUri: s3://amzn-s3-demo-bucket/my-function
      Layers:
        - !Ref MyLambdaLayer
```

使用 Lambda 擴充功能來增強 Lambda 函數

您可以使用 Lambda 擴展功能來增強 Lambda 函數。例如，使用 Lambda 擴展將函數與您偏好的監控、可觀度、安全性和治理工具整合。您可以從 AWS Lambda 合作夥伴提供的廣泛工具集中選擇，也可以建立自己的 Lambda 擴展。

Lambda 支援外部和內部擴展。外部延伸項目會在執行環境中作為獨立的處理序執行，並在完全處理函式叫用之後繼續執行。由於延伸項目會以單獨的程序執行，因此您可以使用與函數不同的語言來撰寫。所有 Lambda 執行期 支援延伸。

內部延伸項目會執行作為執行階段程序的一部分。您的函式透過使用包裝函式指令碼或進行中的機制存取內部延伸項目，例如 `JAVA_TOOL_OPTIONS`。如需詳細資訊，請參閱修改執行階段環境。

您可以使用 Lambda 主控台、 AWS Command Line Interface (AWS CLI) 或基礎設施做為程式碼 (IaC) 服務和工具，例如 AWS Serverless Application Model (AWS SAM) 和 Terraform CloudFormation，將延伸項目新增至函數。

您需要依據延伸項目所耗用的執行時間付費 (以 1 毫秒為單位)。安裝自己的延伸項目不需花費任何費用。如需延伸項目的定價資訊，請參閱AWS Lambda 定價。如需合作夥伴延伸項目的定價資訊，請參閱合作夥伴的網站。如需官方合作夥伴擴充功能清單，請參閱 the section called “延伸合作夥伴”。

主題

- 執行環境
- 對效能和資源的影響
- 許可
- 設定 Lambda 延伸模組
- AWS Lambda 延伸合作夥伴
- 使用 Lambda 延伸 API 來建立延伸項目
- 使用遙測 API 即時存取延伸功能的遙測資料

執行環境

Lambda 會在執行環境中叫用您的函數，該環境可提供安全且隔離的執行時間環境。執行環境會管理執行函數所需的資源，並為函數的執行時間和延伸項目提供生命週期支援。

執行環境的生命週期包含下列階段：

- **Init**：在此階段中，Lambda 會使用設定的資源建立或解除凍結執行環境，下載函數程式碼和所有層，初始化所有擴展功能，初始化執行時間，然後執行函數的初始化程式碼 (主處理常式之外的程式碼)。此 Init 階段發生在第一次調用期間，或者在函數調用之前發生 (如果您已啟用佈建並行)。

  Init 階段分為三個子階段：Extension init、Runtime init 以及 Function init。這些子階段可確保所有擴展功能和執行時間在函數程式碼執行之前完成其設定任務。

  在 Lambda SnapStart 啟動的情況下，發佈函數版本時會發生 Init 階段。Lambda 會儲存初始化執行環境的記憶體和磁碟狀態快照、保留加密的快照，並快取以進行低延遲存取。如果您擁有檢查點前的執行時期勾點，則程式碼會在 Init 階段結束時執行。
- **Restore** (僅限 SnapStart)：第一次叫用 SnapStart 函數且函數擴展時，Lambda 會從保留的快照繼續新的執行環境，而不是從頭開始初始化函數。如果您設定了還原後執行時期勾點，其程式碼會在 Restore 階段結束時執行。您需支付還原後執行時期勾點期間的費用。執行時期必須完成載入，且還原後執行時期勾點必須在逾時限制 (10 秒) 內完成。否則，您將收到 SnapStartTimeoutException 訊息。Restore 階段完成時，Lambda 會調用函數處理常式 (調用階段)。
- **Invoke**：在此階段中，Lambda 會調用函數處理常式。函數執行完成之後，Lambda 會準備處理另一個函數調用。
- **Shutdown**：如果 Lambda 函數在一段時間內未收到任何調用，就會觸發此階段。在 Shutdown 階段中，Lambda 會關閉執行時間、警示擴展功能以便它們完全停止，然後移除環境。Lambda 會傳送 Shutdown 事件到每個擴展功能，這會告訴擴展功能環境即將關閉。

在 Init 階段期間，Lambda 會將包含擴展功能的圖層擷取到執行環境中的 /opt 目錄。Lambda 在 /opt/extensions/ 目錄中尋找擴展功能，將每個檔案解譯為啟動擴展的可執行引導程序，並平行啟動所有擴展。

對效能和資源的影響

函式延伸項目的大小會計入部署套件的大小限制。針對 .zip 封存檔，函數和所有延伸項目解壓縮後的總大小不能超過解壓縮後 250 MB 的部署套件大小限制。

延伸項目可能會影響您的函式效能，因為它們會共用 CPU、記憶體和儲存體等函式資源。例如，如果延伸項目執行運算密集的作業，您可能會看到函數的執行持續時間增加。

在 Lambda 叫用函數之前，每個擴展必須完成其初始化。因此，耗用大量初始化時間的延伸可能會增加函式叫用的延遲。

若要測量延伸項目在函式執行後所花費的額外時間，您可以使用 `PostRuntimeExtensionsDuration` 函式指標。若要測量使用的記憶體的增加情況，您可以使用 `MaxMemoryUsed` 指標。若要了解特定延伸項目的影響，您可以並排執行不同版本的函式。

> **Note**
>
> `MaxMemoryUsed` 指標是 Lambda Insights 收集的指標之一，並非 Lambda 原生指標。

許可

延伸項目可存取與函式相同的資源。因為延伸項目是在與函式相同的環境中執行的，所以許可會在函式和延伸項目之間共用。

對於 .zip 檔案封存，您可以建立 CloudFormation 範本，以簡化將相同延伸組態 - 包括 AWS Identity and Access Management (IAM) 許可 - 連接至多個函數的任務。

設定 Lambda 延伸模組

設定延伸項目 (.zip 封存檔)

您可以將擴展功能作為 Lambda 層新增至函數。使用圖層可讓您在整個組織或整個 Lambda 開發人員社群中共用擴展功能。您可以將一或多個延伸項目新增至圖層。您可以為函式最多註冊 10 個延伸項目。

您可以使用與任何圖層相同的方法將延伸項目新增到您的函式中。如需更多詳細資訊，請參閱 Lambda 層。

將延伸項目新增到您的函式 (主控台)

1. 開啟 Lambda 主控台中的 函數頁面。
2. 選擇一個函數。
3. 如果尚未選取，請選擇 Code (程式碼) 標籤。
4. 在 Layers 下方，選擇 Edit (編輯)。
5. 在選擇圖層中，選擇指定 ARN。
6. 在指定 ARN中，輸入延伸圖層的 Amazon Resource Name (ARN)。
7. 選擇新增。

在容器映像中使用延伸項目

您可以將延伸項目新增至容器映像中。ENTRYPOINT 容器映像設定指定函數的主要程序。在 Dockerfile 中進行 ENTRYPOINT 設定，或設定為函數組態覆寫。

您可以在容器中執行多個程序。Lambda 會管理主程序的生命週期和任何額外程序。Lambda 會使用 Extensions API 來管理擴展生命週期。

範例：新增外部延伸項目

外部擴展會在不同於 Lambda 函數的程序中執行。Lambda 會在 `/opt/extensions/` 目錄中開始每個擴展的程序。Lambda 使用 Extensions API 來管理擴展生命週期。函數執行完成後，Lambda 會將 `Shutdown` 事件傳送至每個外部擴展。

Example將外部延伸項目新增至 Python 基礎映像

```
FROM public.ecr.aws/lambda/python:3.11
```

```
# Copy and install the app
COPY /app /app
WORKDIR /app
RUN pip install -r requirements.txt

# Add an extension from the local directory into /opt/extensions
ADD my-extension.zip /opt/extensions
CMD python ./my-function.py
```

後續步驟

若要深入了解延伸項目，我們建議您使用下列資源：

- 如需基礎工作範例，請參閱 AWS 運算部落格上的建置 AWS Lambda 的延伸項目。
- 如需 AWS Lambda 合作夥伴提供的延伸項目相關資訊，請參閱 AWS 運算部落格上的 AWS Lambda 延伸項目簡介。
- 若要檢視可用的範例延伸項目和包裝函數指令碼，請參閱 AWS 範例 GitHub 儲存庫上的 AWS Lambda 延伸項目。

AWS Lambda 延伸合作夥伴

AWS Lambda 與多個第三方實體合作，提供能夠與您的 Lambda 函數整合的延伸。下列清單詳細介紹了可供您隨時使用的第三方延伸。

- AppDynamics – 提供 Node.js 或 Python Lambda 函數的自動檢測，並提供關於函數效能的可視性和提醒。
- Axiom – 提供監控 Lambda 函數效能和彙總系統層級指標的儀表板。
- Datadog – 透過使用指標、追蹤和日誌，提供對無伺服器應用程式的全面、即時可視性。
- Dynatrace – 提供追蹤和指標可視性，並充分利用 AI 在整個應用程式堆疊中進行自動錯誤偵測和根本原因分析。
- Elastic – 提供應用程式效能監控 (APM)，以使用相關聯的追蹤、指標和日誌識別並解決根本原因問題。
- Epsagon – 監聽調用事件，存放追蹤，並將其平行傳送至 Lambda 函數執行。
- Fastly –保護您的 Lambda 函數免受可疑活動的影響，例如注入式攻擊、透過憑證填充的帳戶接管、惡意機器人和 API 濫用。
- HashiCorp Vault – 管理機密，並使其可供開發人員在函數程式碼中使用，而不讓函數保存庫知道。
- Honeycomb – 用於對應用程式堆疊偵錯的可觀察性工具。
- Lumigo – Profile Lambda 函數會調用並收集用於排解無伺服器和微型服務環境問題的指標。
- New Relic – 與 Lambda 功能一起執行，自動收集、增強遙測，並將其傳輸至 New Relic 的統一可觀察性平台。
- Sedai – 一個由 AI/ML 提供支援的自主雲端管理平台，可為雲端操作團隊提供持續最佳化，以最大限度地大規模節省雲端成本、提高效能和可用性。
- Sentry – 診斷、修復和最佳化 Lambda 函數的效能。
- Site24x7 – 實現 Lambda 環境的即時可觀察性
- Splunk – 收集高解析度、低延遲指標，以高效且有效率地監控 Lambda 函數。
- Sumo Logic – 提供對無伺服器應用程式運作狀態和效能的可視性。
- Salt Security：透過自動設定和支援各種執行時期，簡化 Lambda 函數的 API 狀態管理和 API 安全。

AWS 受管延伸

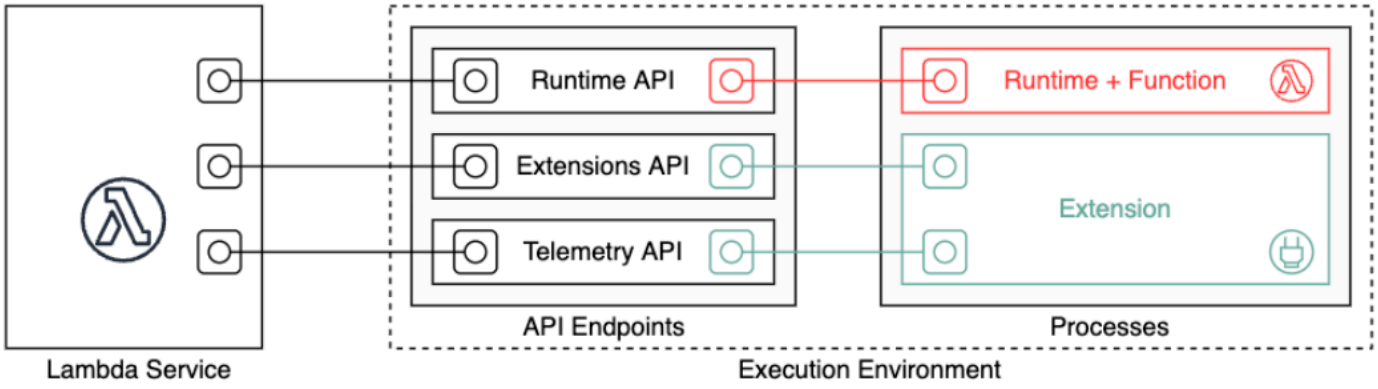
AWS 提供其自身的受管延伸，包括：

- AWS AppConfig – 使用功能標誌和動態資料來更新 Lambda 函數。您還可以使用此延伸來更新其他動態組態，如維運調節和調校。
- Amazon CodeGuru Profiler – 透過確定應用程式最昂貴的程式碼行，並提供改善程式碼的建議，來改善應用程式效能並降低成本。
- CloudWatch Lambda Insights – 透過自動儀表板來監控、疑難排解和最佳化 Lambda 函數的效能。
- AWS Distro for OpenTelemetry (ADOT) – 讓函數能將追蹤資料傳送至 AWS X-Ray 這類 AWS 監控服務，以及 Honeycomb 和 Lightstep 等支援 OpenTelemetry 的目的地。
- AWS Parameters and Secrets – 安全地擷取 AWS Systems Manager Parameter Store 中的參數和 Lambda 函式 AWS Secrets Manager 中的秘密。

如需其他延伸範例和示範專案，請參閱 AWS Lambda 延伸。

使用 Lambda 延伸 API 來建立延伸項目

Lambda 函數作者使用延伸項目將 Lambda 與其偏好的監控、可觀度、安全性和治理工具整合在一起。函數作者可以使用來自 AWS、 AWS 合作夥伴和開放原始碼專案的擴充功能。如需使用延伸模組的詳細資訊，請參閱運算部落格上的 AWS 介紹 AWS Lambda 延伸模組。本節說明如何使用 Lambda 延伸 API、Lambda 執行環境生命週期以及 Lambda 延伸 API 參考。



作為延伸項目的作者，您可以使用 Lambda Extensions API 以便深入整合到 Lambda 執行環境中。您的延伸項目可以註冊函數和執行環境生命週期事件。為了回應這些事件，您可以啟動新程序、執行邏輯，以及控制並參與 Lambda 生命週期的所有階段：初始化、調用和關閉。此外，您可以使用 Runtime Logs API 來接收日誌串流。

延伸項目會在執行環境中作為獨立的處理序執行，並在完全處理函數調用之後繼續執行。由於延伸項目會以程序執行，因此您可以使用與函數不同的語言來撰寫。我們建議您使用編譯語言實作延伸項目。在這種情況下，延伸項目是獨立的二進位檔案，會與支援的執行時間相容。所有 Lambda 執行期 支援延伸。如果您使用非編譯語言，請確定您在延伸項目中包含相容的執行階段。

Lambda 也支援內部延伸項目。內部延伸項目會作為獨立執行緒在執行時間程序中執行。執行時間會啟動並停止內部延伸項目。與 Lambda 環境整合的另一種方法是使用特定語言的環境變數和包裝函數指令碼。您可使用這些來設定執行時間環境，並修改執行時間程序的啟動行為。

您可以使用兩種方法將延伸項目新增至函數。對於部署為 .zip 封存檔案的函數，您可以將延伸項目部署為 layer。對於定義為容器映像的函數，可以將延伸項目新增至容器映像。

> **Note**
>
> 如需延伸項目和包裝函式指令碼的範例，請參閱 AWS 範例 GitHub 儲存庫上的AWS Lambda 延伸項目。

主題

- Lambda 執行環境生命週期
- Extensions API 參考

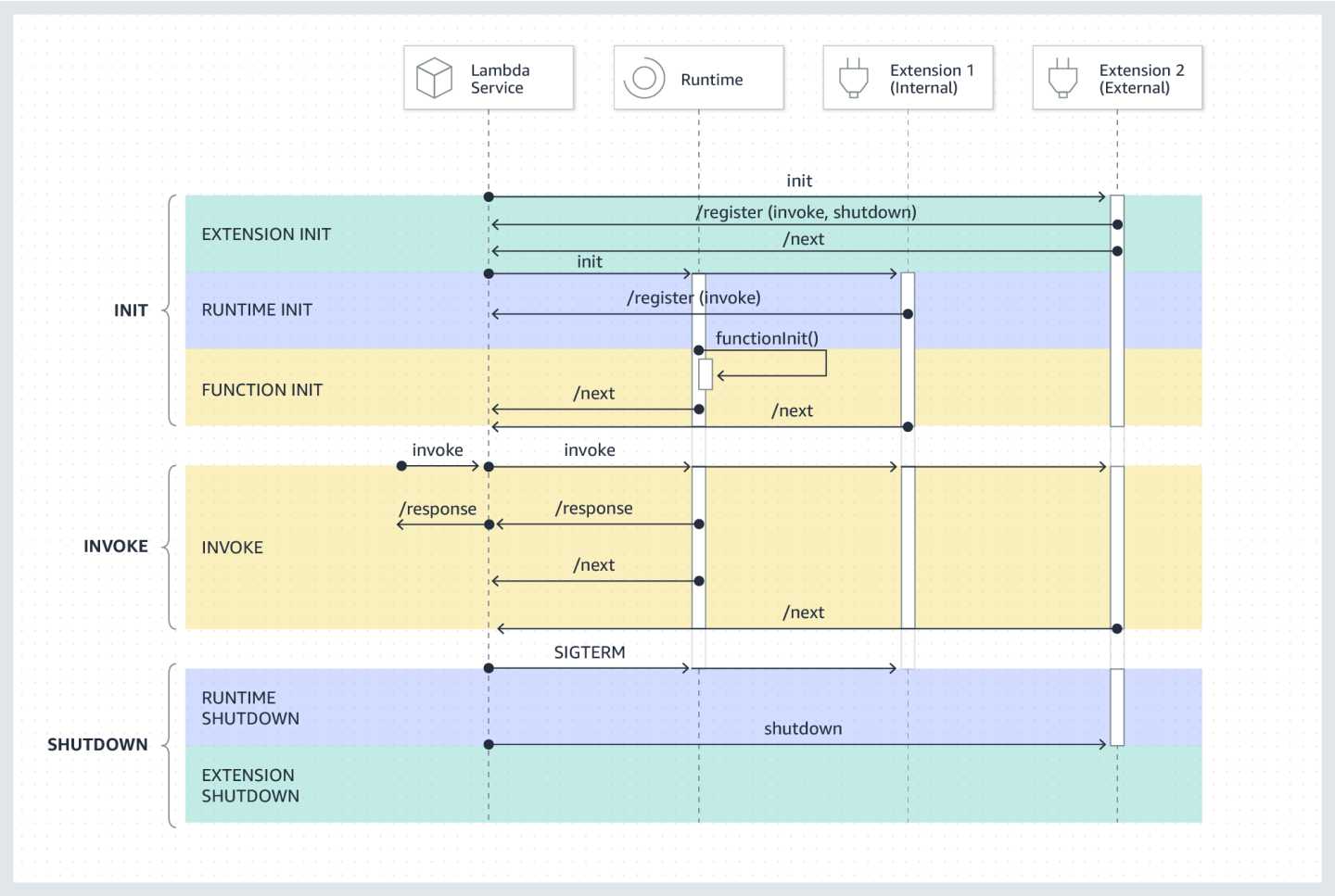
Lambda 執行環境生命週期

執行環境的生命週期包含下列階段：

- **Init**：在此階段中，Lambda 會使用設定的資源建立或解除凍結執行環境，下載函數程式碼和所有層，初始化所有擴展功能，初始化執行時間，然後執行函數的初始化程式碼 (主處理常式之外的程式碼)。此 `Init` 階段發生在第一次調用期間，或者在函數調用之前發生 (如果您已啟用佈建並行)。

  `Init` 階段分為三個子階段：`Extension init`、`Runtime init` 以及 `Function init`。這些子階段可確保所有擴展功能和執行時間在函數程式碼執行之前完成其設定任務。
- **Invoke**：在此階段中，Lambda 會調用函數處理常式。函數執行完成之後，Lambda 會準備處理另一個函數調用。
- **Shutdown**：如果 Lambda 函數在一段時間內未收到任何調用，就會觸發此階段。在 `Shutdown` 階段中，Lambda 會關閉執行時間、警示擴展功能以便它們完全停止，然後移除環境。Lambda 會傳送 `Shutdown` 事件到每個擴展功能，這會告訴擴展功能環境即將關閉。

每個階段都會從 Lambda 到執行時間和所有已註冊延伸項目的事件開始。執行時間和每個已註冊延伸項目都會透過傳送 `Next` API 請求來表示已完成。當每個處理已完成且沒有擱置的事件時，Lambda 凍結執行環境。



主題

- 初始化階段
- 調用階段
- 關閉階段
- 許可與組態
- 失敗處理
- 故障排除延伸項目

初始化階段

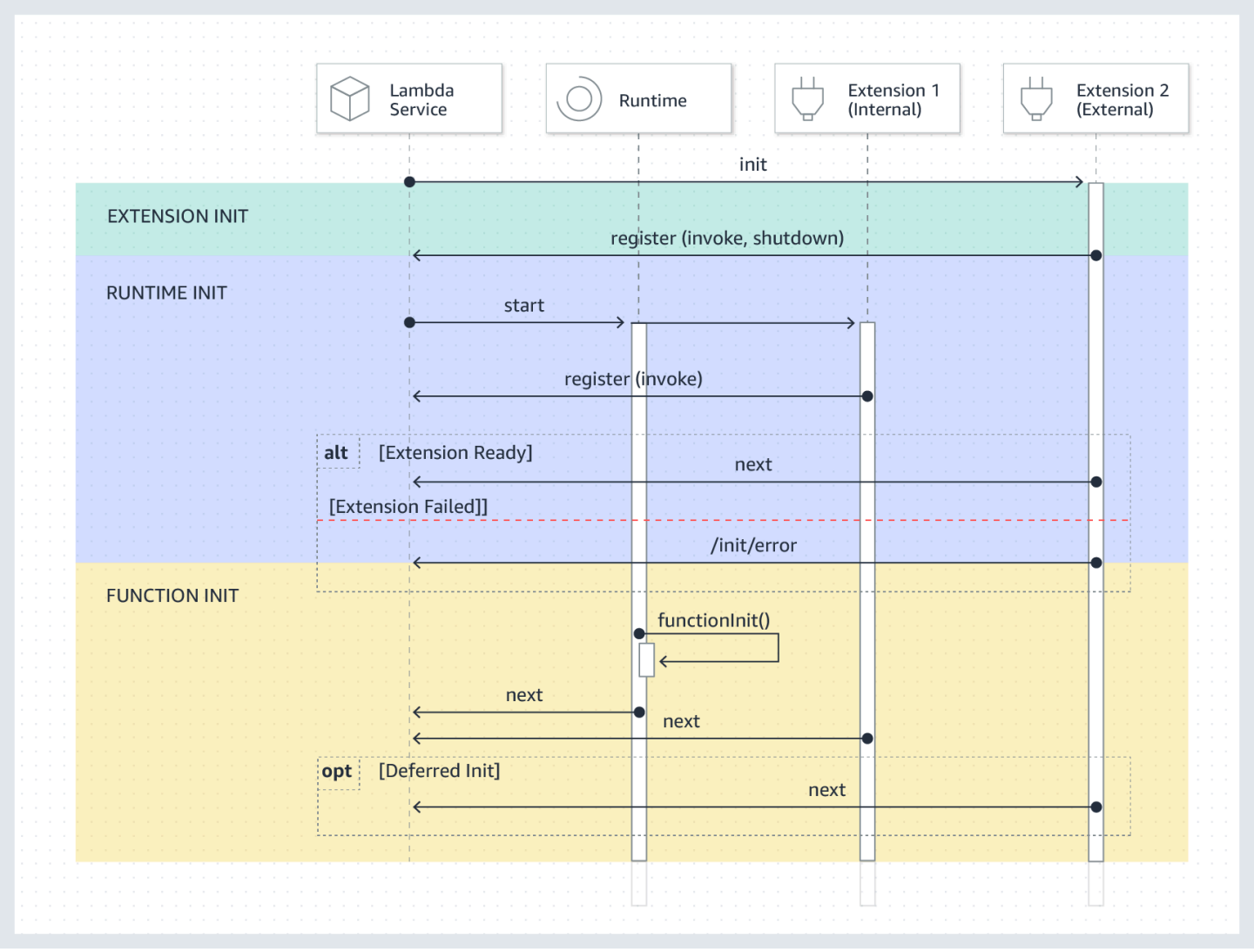
在 `Extension init` 階段中，每個延伸項目都需要向 Lambda 註冊才能接收事件。Lambda 會使用延伸項目的完整檔案名稱來驗證延伸項目是否已完成啟動程序。因此，每個 `Register` API 呼叫必須包含延伸項目完整檔案名稱的 `Lambda-Extension-Name` 標頭。

您可以為函式最多註冊 10 個延伸項目。此限制會透過 `Register` API 呼叫強制執行。

在每個延伸項目註冊後，Lambda 會啟動 `Runtime init` 階段。執行時間程序會呼叫 `functionInit` 以啟動 `Function init` 階段。

`Init` 階段會在執行階段之後完成，每個已註冊延伸項目都會透過傳送 `Next` API 請求來表示已完成。

> **Note**
>
> 延伸項目可以在 `Init` 階段的任何時候完成初始化。



調用階段

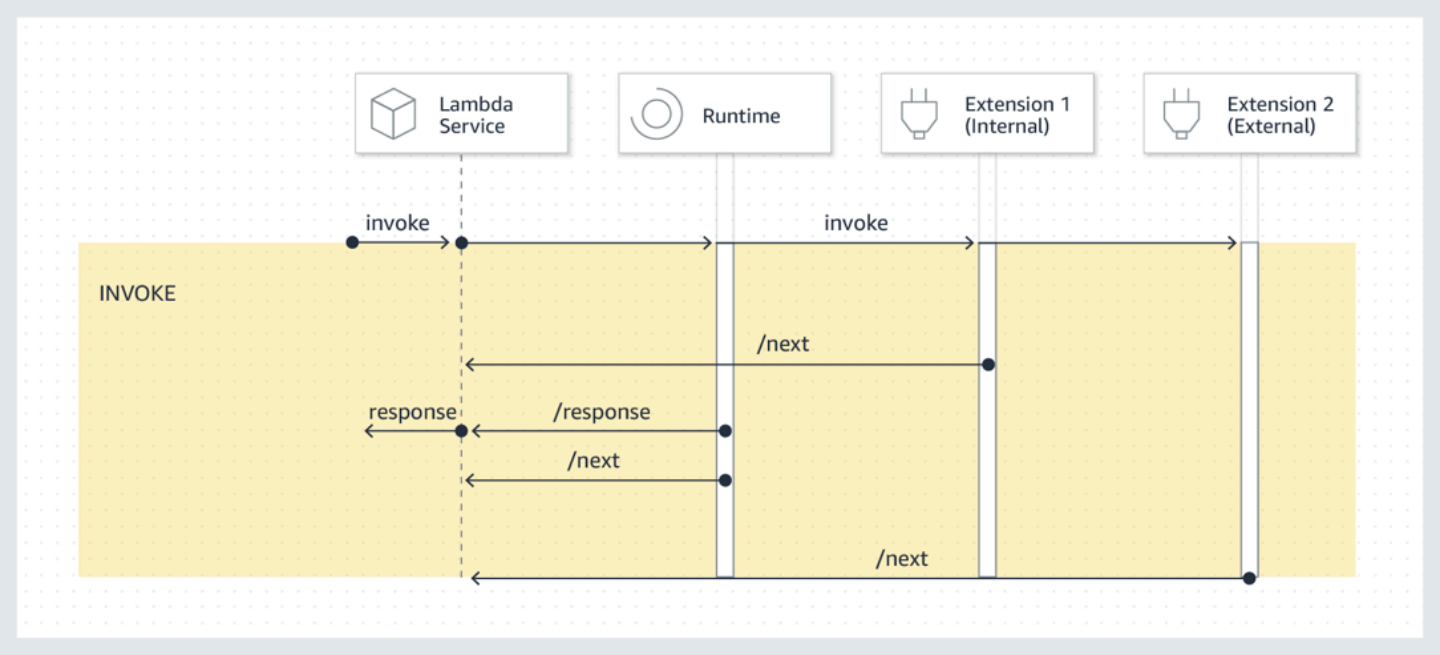
當調用 Lambda 函數來回應 Next API 請求時，Lambda 會將 Invoke 事件傳送至執行時間，並傳送至針對該 Invoke 事件已註冊的每個延伸項目。

> **Note**
>
> Lambda 受管執行個體：Lambda 受管執行個體函數的延伸模組無法註冊Invoke事件。由於 Lambda 受管執行個體支援單一執行環境中的並行調用，因此不支援該Invoke事件。延伸模組只能註冊Shutdown事件。如果您需要追蹤調用開始和完成的時間，請透過遙測 API 使用platform.report平台事件。

在調用期間，外部延伸項目會與函式平行執行。函式完成後，它們也會繼續執行。這可讓您擷取診斷資訊，或將日誌、指標和追蹤傳送至您選擇的位置。

從執行時間中收到函數回應之後，即使延伸項目仍在執行中，Lambda 也會將回應傳回給用戶端。

Invoke 階段會在執行階段後結束，所有延伸項目訊號都透過傳送 Next API 請求完成。



每個承載：傳送到執行時間 (和 Lambda 函數) 的事件載有整個請求、標頭 (例如 RequestId) 和承載。傳送至每個延伸項目的事件會包含描述事件內容的中繼資料。此生命週期事件包括事件的類型、函數逾時的時間 (deadlineMs) requestId、調用函數的 Amazon Resource Name (ARN) 以及追蹤標頭。

想要存取函式事件主體的延伸項目，您可以使用與延伸項目通訊的執行階段 SDK。函式開發人員會使用執行階段內 SDK，在調用函式時將承載傳送至延伸項目。

以下是承載範例：

```
{
    "eventType": "INVOKE",
    "deadlineMs": 676051,
    "requestId": "3da1f2dc-3222-475e-9205-e2e6c6318895",
    "invokedFunctionArn": "arn:aws:lambda:us-
east-1:123456789012:function:ExtensionTest",
    "tracing": {
        "type": "X-Amzn-Trace-Id",
        "value":
 "Root=1-5f35ae12-0c0fec141ab77a00bc047aa2;Parent=2be948a625588e32;Sampled=1"
    }
 }
```

持續時間限制：該函數的逾時設置限制了整個 Invoke 階段的持續時間。例如，如果您將函式逾時設定為 360 秒，則函式和所有延伸項目都需要在 360 秒內完成。請注意，沒有獨立的調用後階段。持續時間是執行時期和所有延伸項目的調用所花費的總時間，直到函數和所有延伸項目完成執行後才會計算。

效能影響和延伸項目負擔：延伸項目可能會影響函式效能。身為延伸項目的作者，您可以控制延伸項目的效能影響。例如，如果延伸項目執行運算密集型操作，函式的持續時間便會延長，因為延伸項目和函式程式碼會共用相同的 CPU 資源。此外，如果您的延伸項目在函式調用完成後執行大量作業，函式的持續時間就會增加，因為 Invoke 階段會持續進行，直到所有延伸項目表示它們已完成為止。

> **Note**
>
> Lambda 會根據函數的記憶體設定來分配 CPU 功率。您可能會在較低的記憶體設定下看到執行和初始化持續時間增加，因為函數和延伸項目程序正在競爭相同的 CPU 資源。若要減少執行和初始化持續時間，請嘗試增加記憶體設定。

為了協助識別延伸在 Invoke 階段上造成的效能衝擊，Lambda 會輸出 PostRuntimeExtensionsDuration 指標。此指標會測量執行階段 Next API 請求與上次延伸 Next API 請求之間所花費的累計時間。若要測量使用的記憶體增加的情況，請使用 MaxMemoryUsed 指標。如需函式指標的詳細資訊，請參閱將 CloudWatch 指標與 Lambda 搭配使用。

函式開發人員可以並排執行不同版本的函式，以了解特定延伸項目的影響。我們建議延伸項目作者發佈預期的資源消耗，以便函式開發人員更容易選擇合適的延伸項目。

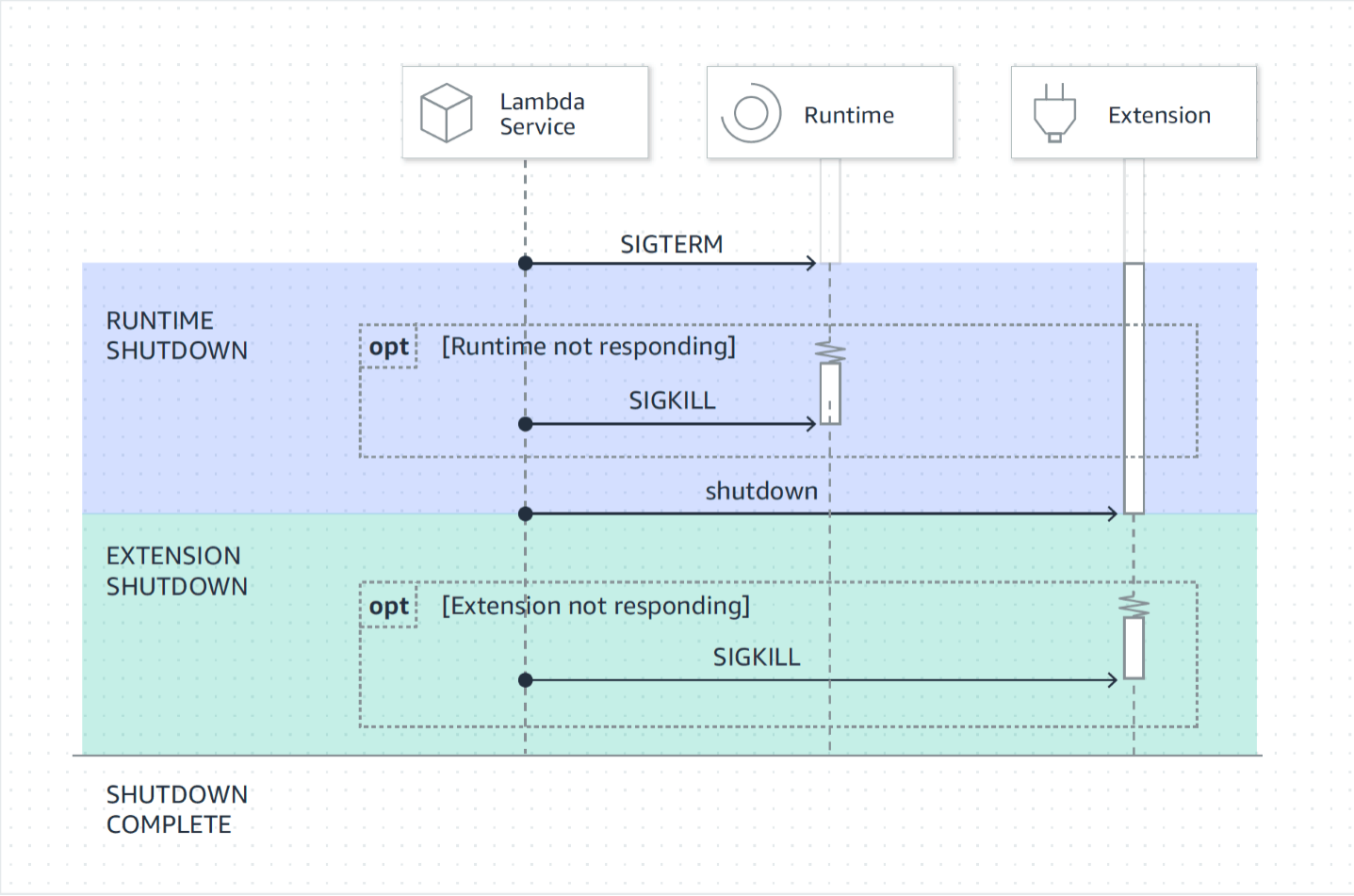
關閉階段

當 Lambda 即將關閉執行時間，它會將 Shutdown 事件傳送至每個已註冊外部延伸。延伸項目可以使用此時間進行最終清理工作。系統會傳送 Shutdown 事件以回應 Next API 請求。

持續時間限制：Shutdown 階段的最長持續時間視已註冊延伸項目的組態而定：

- 0 毫秒 - 沒有註冊延伸的函數
- 500 毫秒 - 具有已註冊內部延伸項目的函數
- 2,000 毫秒 - 具有一個或多個已註冊外部延伸項目的函數

如果執行時間或延伸項目未在限制內回應 Shutdown 事件，Lambda 會使用 SIGKILL 訊號結束程序。



事件承載：Shutdown 事件會包含關閉的原因和剩餘時間 (以毫秒為單位)。

shutdownReason 包含以下值：

- SPINDOWN - 正常關閉
- TIMEOUT - 持續時間限制逾時
- FAILURE - 錯誤條件，例如 out-of-memory 事件

```
{
  "eventType": "SHUTDOWN",
  "shutdownReason": "reason for shutdown",
  "deadlineMs": "the time and date that the function times out in Unix time
 milliseconds"
}
```

許可與組態

延伸項目會在與 Lambda 函數相同的執行環境中執行。延伸項目也會與函式共用資源，例如 CPU、記憶體和 /tmp 儲存磁碟。此外，延伸模組會使用與 函數相同的 AWS Identity and Access Management (IAM) 角色和安全性內容。

檔案系統和網路存取許可：延伸項目在與函式執行階段相同的檔案系統和網路名稱命名空間中執行。這表示延伸項目必須與相關的作業系統相容。如果延伸項目需要任何其他傳出網路流量規則，您必須將這些規則套用至函數組態。

> **Note**
>
> 由於函式程式碼目錄是唯讀的目錄，因此延伸項目無法修改函式程式碼。

環境變數：延伸項目可以存取函式的環境變數，但下列特定於執行階段程序的變數除外：

- AWS_EXECUTION_ENV
- AWS_LAMBDA_LOG_GROUP_NAME
- AWS_LAMBDA_LOG_STREAM_NAME
- AWS_XRAY_CONTEXT_MISSING
- AWS_XRAY_DAEMON_ADDRESS
- LAMBDA_RUNTIME_DIR

- LAMBDA_TASK_ROOT
- _AWS_XRAY_DAEMON_ADDRESS
- _AWS_XRAY_DAEMON_PORT
- _HANDLER

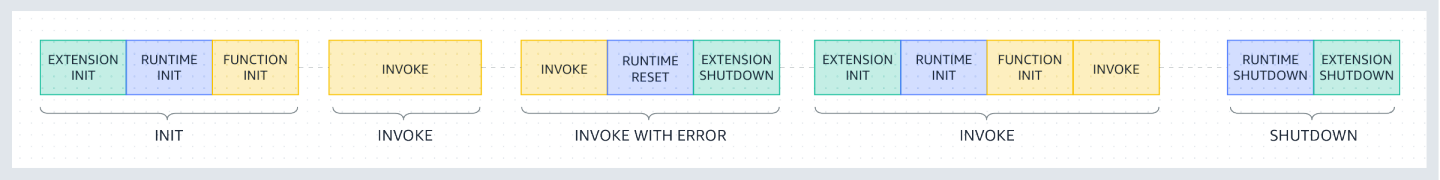
> **Note**
>
> 偵測 Lambda 受管執行個體：延伸項目可以檢查AWS_LAMBDA_INITIALIZATION_TYPE環境變數，以判斷它們是否在 Lambda 受管執行個體上執行，而不是在 Lambda （預設） 函數上執行。這是擴充功能根據執行環境類型調整其行為的建議方法。

失敗處理

初始化失敗：如果延伸失敗，Lambda 會重新啟動執行環境以強制執行一致的行為，並促進延伸項目快速失敗。此外，對於某些客戶，延伸項目必須符合任務關鍵需求，例如日誌、安全性、治理和遙測收集。

調用失敗 (例如記憶體不足、函式逾時)：因為延伸項目程序與執行時間共享資源，所以記憶體耗盡會對其產生影響。當執行階段失敗時，所有延伸項目和執行階段本身都會參與此 Shutdown 階段。此外，執行時間會自動重新啟動，這可能是為了要做為目前調用的一部分，或是透過延遲的重新初始化機制重新啟動。

如果在 Invoke 期間發生失敗 (例如函數逾時或執行時間錯誤)，則 Lambda 服務會執行重設。重設的行為會與 Shutdown 事件一樣。首先，Lambda 關閉執行時間，然後將 Shutdown 事件傳送給每個已註冊外部延伸項目。事件會包括關閉的原因。如果將此環境用於新的調用，延伸項目和執行階段會重新初始化為下次調用的一部分。



如需上圖更為詳細的說明，請參閱 調用階段期間出現的故障。

延伸項目日誌：Lambda 會將延伸項目的日誌輸出傳送至 CloudWatch Logs。Lambda 也會在 Init 期間為每個延伸項目產生額外的日誌事件。日誌事件會記錄成功時的名稱和註冊偏好設定 (事件、設定)，或失敗時的失敗原因。

故障排除延伸項目

- 如果 Register 請求失敗，請確定 Register API 呼叫中的 Lambda-Extension-Name 標頭包含延伸項目的完整檔案名稱。
- 如果內部延伸項目的 Register 請求失敗，請確定請求不會註冊 Shutdown 事件。

Extensions API 參考

對於 Extensions API 版本 2020-01-01 的 OpenAPI 規範可以在這裡找到：extensions-api.zip

您可以從 AWS_LAMBDA_RUNTIME_API 環境變數擷取 API 端點的值。若要傳送 Register 請求，請在每個 API 路徑之前使用前綴 2020-01-01/。例如：

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
```

API 方法

- 登錄
- 下一頁
- 初次化錯誤
- 結束錯誤

登錄

在 Extension init 階段中，所有延伸項目都需要向 Lambda 註冊才能接收事件。Lambda 會使用延伸項目的完整檔案名稱來驗證延伸項目是否已完成啟動程序。因此，每個 Register API 呼叫必須包含延伸項目完整檔案名稱的 Lambda-Extension-Name 標頭。

執行階段程序會啟動和停止內部延伸項目，因此不允許它們註冊 Shutdown 事件。

路徑 – /extension/register

方法 – POST

請求標頭

- Lambda-Extension-Name - 延伸項目的完整檔案名稱。必要：是。類型：字串

- `Lambda-Extension-Accept-Feature` - 用來在註冊期間指定選用的延伸項目特色。必要：否。類型：逗號分隔的字串。可使用此設定來指定的特色：
  - `accountId` - 如果指定此項，延伸項目註冊回應將包含與待註冊延伸項目之 Lambda 函數相關聯的帳戶 ID。

請求內文參數

- `events` - 要註冊的事件陣列。必要：否。類型：字串陣列。有效字串：`INVOKE`、`SHUTDOWN`。

> **Note**
>
> Lambda 受管執行個體：Lambda 受管執行個體函數的延伸模組只能註冊`SHUTDOWN`事件。嘗試註冊`INVOKE`事件將導致錯誤。這是因為 Lambda 受管執行個體支援單一執行環境中的並行調用。

回應標頭

- `Lambda-Extension-Identifier` - 產生所有後續請求所需的唯一代理程式識別符 (UUID 字串)。

回應代碼

- 200 - 回應主體包含的函數名稱、函數版本和處理常式名稱。
- 400 - 錯誤請求
- 403 - 禁止
- 500 - 容器錯誤。不可復原的狀態。延伸項目應會立即結束。

Example範例請求主體

```
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

Example範例回應主題

```
{
    "functionName": "helloWorld",
```

```
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler"
}
```

Example具有選用 accountId 特色的回應內文範例

```
{
    "functionName": "helloWorld",
    "functionVersion": "$LATEST",
    "handler": "lambda_function.lambda_handler",
    "accountId": "123456789012"
}
```

下一頁

延伸項目會傳送 Next API 請求以接收下一個事件，可以是 Invoke 事件或 Shutdown 事件。回應主體包含承載，這是包含事件資料的 JSON 文件。

延伸項目會傳送 Next API 請求，以表示它已準備好接收新事件。這是一個封鎖調用。

請勿在 GET 調用上設定逾時，因為延伸項目可以暫停一段時間，直到有事件傳回為止。

路徑 – /extension/event/next

方法 – GET

請求標頭

- Lambda-Extension-Identifier - 延伸項目的唯一識別符 (UUID 字串)。必要：是。類型：UUID 字串。

回應標頭

- Lambda-Extension-Event-Identifier - 延伸項目的唯一識別符 (UUID 字串)。

回應代碼

- 200 - 回應包含下個事件 (EventInvoke 或 EventShutdown) 的相關資訊。
- 403 - 禁止
- 500 - 容器錯誤。不可復原的狀態。延伸項目應會立即結束。

初次化錯誤

延伸項目會使用此方法向 Lambda 報告初始化錯誤。當延伸項目註冊後無法初始化時會調用它。Lambda 收到錯誤後，進一步的 API 呼叫沒有成功。延伸項目應該在收到 Lambda 的回應之後退出。

路徑 – `/extension/init/error`

方法 – POST

請求標頭

- `Lambda-Extension-Identifier` - 延伸項目的唯一識別符。必要：是。類型：UUID 字串。
- `Lambda-Extension-Function-Error-Type` - 擴展遇到的錯誤類型。必要：是。此標頭包含一個字串值。Lambda 可接受任何字串，但我們建議使用格式 <category.reason>。例如：
  - Extension.NoSuchHandler
  - Extension.APIKeyNotFound
  - Extension.ConfigInvalid
  - Extension.UnknownReason

請求內文參數

- `ErrorRequest` - 關於錯誤的資訊。必要：否。

此欄位是具有下列結構的 JSON 物件：

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

請注意，Lambda 接受 `errorType` 的任何值。

下列範例顯示 Lambda 函數錯誤訊息，其中函數無法剖析調用中提供的事件資料。

Example函數錯誤

```
{
```

```
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

回應代碼

- 202 - 已接受
- 400 - 錯誤請求
- 403 - 禁止
- 500 - 容器錯誤。不可復原的狀態。延伸項目應會立即結束。

結束錯誤

延伸項目會使用此方法，在結束前向 Lambda 報告錯誤。當您遇到意外失敗時呼叫它。Lambda 收到錯誤後，進一步的 API 呼叫沒有成功。延伸項目應該在收到 Lambda 的回應之後退出。

路徑 – `/extension/exit/error`

方法 – POST

請求標頭

- `Lambda-Extension-Identifier` - 延伸項目的唯一識別符。必要：是。類型：UUID 字串。
- `Lambda-Extension-Function-Error-Type` - 擴展遇到的錯誤類型。必要：是。此標頭包含一個字串值。Lambda 可接受任何字串，但我們建議使用格式 <category.reason>。例如：
  - Extension.NoSuchHandler
  - Extension.APIKeyNotFound
  - Extension.ConfigInvalid
  - Extension.UnknownReason

請求內文參數

- `ErrorRequest` - 關於錯誤的資訊。必要：否。

此欄位是具有下列結構的 JSON 物件：

```
{
      errorMessage: string (text description of the error),
      errorType: string,
      stackTrace: array of strings
}
```

請注意，Lambda 接受 errorType 的任何值。

下列範例顯示 Lambda 函數錯誤訊息，其中函數無法剖析調用中提供的事件資料。

Example函數錯誤

```
{
      "errorMessage" : "Error parsing event data.",
      "errorType" : "InvalidEventDataException",
      "stackTrace": [ ]
}
```

回應代碼

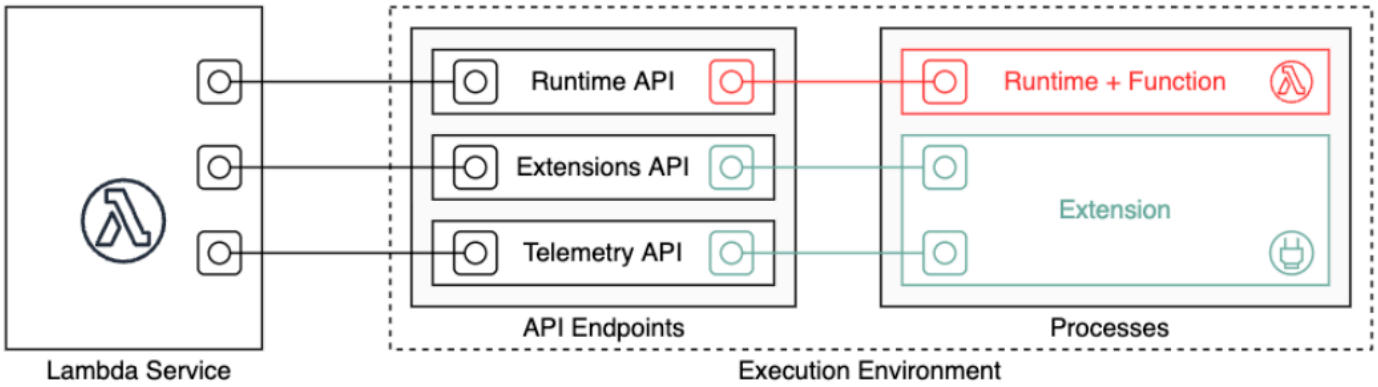
- 202 - 已接受
- 400 - 錯誤請求
- 403 - 禁止
- 500 - 容器錯誤。不可復原的狀態。延伸項目應會立即結束。

使用遙測 API 即時存取延伸功能的遙測資料

遙測 API 可讓您的延伸項目直接從 Lambda 接收遙測資料。在函數初始化和調用期間，Lambda 會自動擷取遙測資料，包括日誌、平台指標和平台追蹤。遙測 API 使延伸項目可以近乎即時地從 Lambda 直接存取這些遙測資料。

在 Lambda 執行環境中，您可以讓 Lambda 延伸項目訂閱遙測串流。訂閱後，Lambda 會自動將所有遙測資料傳送至您的延伸項目。隨後，您便能靈活地處理、篩選資料，並將資料傳送至偏好的目的地，例如 Amazon Simple Storage Service (Amazon S3) 儲存貯體或第三方可觀測性工具提供者。

下圖顯示延伸 API 和遙測 API 如何從執行環境中將延伸項目連結至 Lambda。另外，執行期 API 也會將執行期和函數連接至 Lambda。



Important

Lambda 遙測 API 優先於 Lambda 日誌 API。雖然日誌 API 仍然正常運作，但我們建議您未來只使用遙測 API。您可以使用遙測 API 或日誌 API 訂閱遙測串流的延伸項目。使用其中一個 API 進行訂閱後，若嘗試使用其他 API 進行任何訂閱，都會傳回錯誤。

Lambda 受管執行個體結構描述版本需求

Lambda 受管執行個體僅支援遙測 API `2025-01-29` 的結構描述版本。訂閱受管執行個體函數的遙測串流時，您必須在訂閱請求`"schemaVersion": "2025-01-29"`中使用 。使用先前的結構描述版本會導致 Lambda 拒絕事件。
`2025-01-29` 結構描述版本回溯相容，可與 Lambda 受管執行個體和 Lambda （預設） 函數搭配使用。我們建議將此版本用於所有新的延伸，以確保兩種部署模型的相容性。

延伸項目可使用遙測 API 來訂閱三個不同的遙測串流：

- 平台遙測 – 日誌、指標和追蹤，描述與執行環境執行階段生命週期、延伸項目生命週期和函數調用相關的事件和錯誤。
- 函數日誌 – Lambda 函數程式碼產生的自訂日誌。
- 延伸項目日誌 - Lambda 延伸項目程式碼產生的自訂日誌。

> **Note**
>
> Lambda 會將日誌和指標傳送到 CloudWatch，並將追蹤傳送至 X-Ray (如果您已啟動追蹤)，即使延伸項目有訂閱遙測串流也一樣。

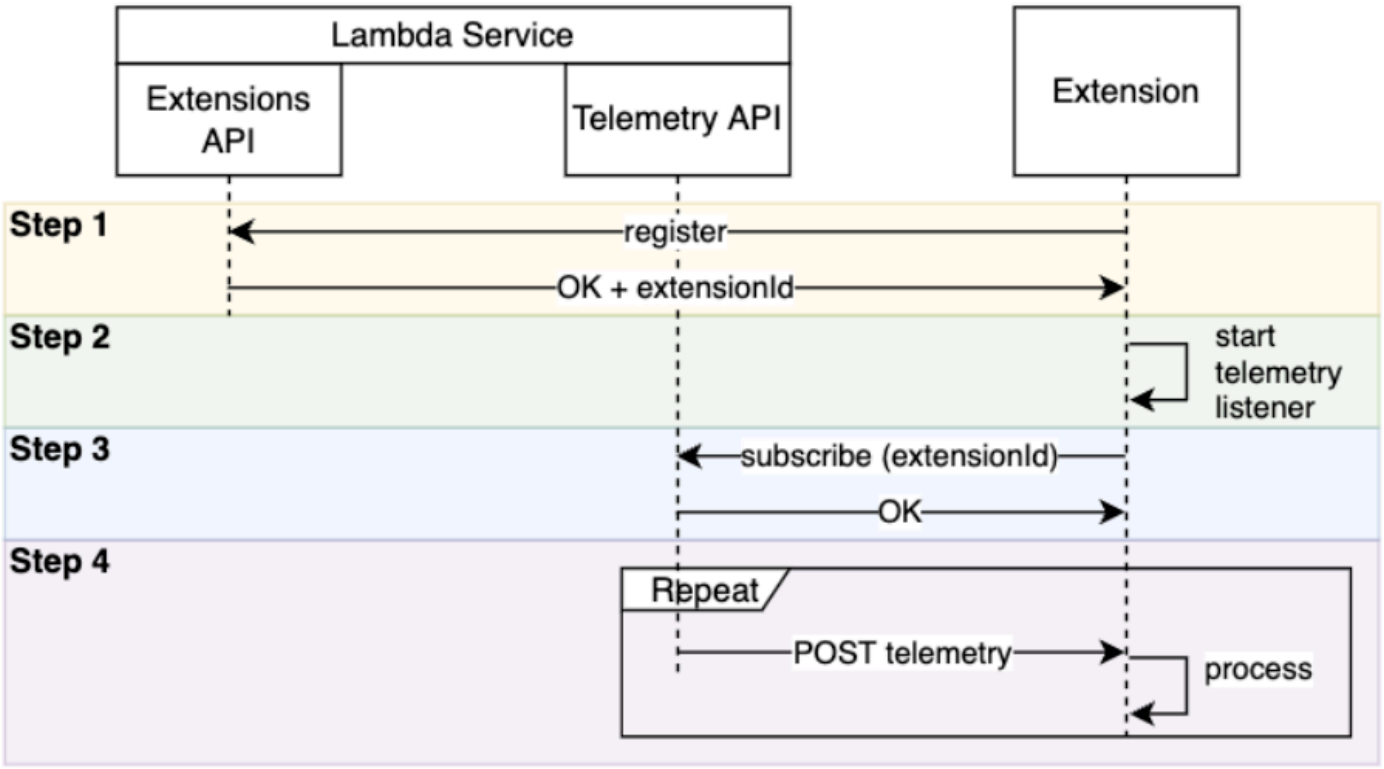
章節

- 使用遙測 API 建立延伸項目
- 註冊延伸項目
- 建立遙測接聽程式
- 指定目的地通訊協定
- 設定記憶體使用量和緩衝
- 將訂閱請求傳送至遙測 API
- 輸入遙測 API 訊息
- Lambda 遙測 API 參考
- Lambda 遙測 API Event 結構描述參考
- 將 Lambda 遙測 APIEvent 物件轉換為 OpenTelemetry 跨度
- 使用 Lambda Logs API

使用遙測 API 建立延伸項目

Lambda 延伸項目會在執行環境中做為獨立的程序執行。延伸項目可以在函數調用完成後繼續執行。由於延伸項目為獨立的處理序，因此您可以使用與函數程式碼不同的語言來撰寫。我們建議您使用 Golang 或 Rust 等編譯語言編寫延伸項目。在這種情況下，延伸項目是獨立的二進位檔案，且與任何支援的執行階段相容。

下圖說明建立延伸項目的四步驟程序，讓延伸項目使用遙測 API 接收和處理遙測資料。



以下是各步驟的詳細說明：

1. 使用 the section called “Extensions API” 註冊延伸項目。這會為您提供 `Lambda-Extension-Identifier`，接著需要在下列步驟中使用。如需有關如何註冊延伸項目的詳細資訊，請參閱：the section called “註冊延伸項目”。
2. 建立遙測接聽程式。這可以是基本的 HTTP 或 TCP 伺服器。Lambda 使用遙測接聽程式的 URI 來將遙測資料傳送至延伸項目。如需詳細資訊，請參閱the section called “建立遙測接聽程式”。
3. 使用遙測 API 中的訂閱 API，為您的延伸項目訂閱需要的遙測串流。在此步驟中，您需要遙測接聽程式的 URI。如需詳細資訊，請參閱the section called “將訂閱請求傳送至遙測 API”。
4. 透過遙測接聽程式從 Lambda 取得遙測資料。您可以對這些資料執行任何自訂處理，例如將資料分派到 Amazon S3 或外部可檢視性服務。

> **Note**
>
> Lambda 函數的執行環境可以在其生命週期中多次啟動和停止。一般來說，延伸項目程式碼會在函數調用期間執行，並且在關閉階段執行最多 2 秒。我們建議在遙測傳送到您的接聽程式時

進行批次處理。然後，使用 Invoke 和 Shutdown 生命週期事件將每個批次傳送到所需的目的地。

註冊延伸項目

您必須先註冊 Lambda 延伸項目，才能訂閱遙測資料。註冊會在延伸功能初始化階段進行。下列範例顯示註冊延伸項目的 HTTP 請求。

```
POST http://${AWS_LAMBDA_RUNTIME_API}/2020-01-01/extension/register
 Lambda-Extension-Name: lambda_extension_name
{
    'events': [ 'INVOKE', 'SHUTDOWN']
}
```

如果請求成功，訂閱者會收到 HTTP 200 成功回應。回應標頭包含 Lambda-Extension-Identifier。回應內文包含函數的其他屬性。

```
HTTP/1.1 200 OK
Lambda-Extension-Identifier: a1b2c3d4-5678-90ab-cdef-EXAMPLE11111
{
    "functionName": "lambda_function",
    "functionVersion": "$LATEST",
    "handler": "lambda_handler",
    "accountId": "123456789012"
}
```

如需更多資訊，請參閱the section called “Extensions API 參考”。

建立遙測接聽程式

Lambda 延伸項目必須具有可處理遙測 API 所傳入請求的接聽程式。下列程式碼顯示以 Golang 實作的遙測接聽程式實作範例：

```
// Starts the server in a goroutine where the log events will be sent
func (s *TelemetryApiListener) Start() (string, error) {
 address := listenOnAddress()
 l.Info("[listener:Start] Starting on address", address)
 s.httpServer = &http.Server{Addr: address}
 http.HandleFunc("/", s.http_handler)
 go func() {
```

```
  err := s.httpServer.ListenAndServe()
  if err != http.ErrServerClosed {
   l.Error("[listener:goroutine] Unexpected stop on Http Server:", err)
   s.Shutdown()
  } else {
   l.Info("[listener:goroutine] Http Server closed:", err)
  }
 }()
 return fmt.Sprintf("http://%s/", address), nil
}

// http_handler handles the requests coming from the Telemetry API.
// Everytime Telemetry API sends log events, this function will read them from the
 response body
// and put into a synchronous queue to be dispatched later.
// Logging or printing besides the error cases below is not recommended if you have
 subscribed to
// receive extension logs. Otherwise, logging here will cause Telemetry API to send new
 logs for
// the printed lines which may create an infinite loop.
func (s *TelemetryApiListener) http_handler(w http.ResponseWriter, r *http.Request) {
 body, err := ioutil.ReadAll(r.Body)
 if err != nil {
  l.Error("[listener:http_handler] Error reading body:", err)
  return
 }

 // Parse and put the log messages into the queue
 var slice []interface{}
 _ = json.Unmarshal(body, &slice)

 for _, el := range slice {
  s.LogEventsQueue.Put(el)
 }

 l.Info("[listener:http_handler] logEvents received:", len(slice), " LogEventsQueue
 length:", s.LogEventsQueue.Len())
 slice = nil
}
```

指定目的地通訊協定

使用遙測 API 訂閱以接收遙測資料時，除了目的地 URI 之外，您還可以指定目的地通訊協定：

```
{
    "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
    }
}
```

Lambda 接受兩種通訊協定用於接收遙測資料：

- HTTP (建議) - Lambda 會以 JSON 格式的記錄陣列將遙測資料傳遞至本機 HTTP 端點 (`http://sandbox.localdomain:${PORT}/${PATH}`)。$PATH 為選用參數。Lambda 僅支援 HTTP，不支援 HTTPS。Lambda 會透過 POST 請求傳遞遙測資料。
- TCP - Lambda 使用以換行分隔的 JSON (NDJSON) 格式將遙測資料傳遞至 TCP 連接埠。

> **Note**
>
> 強烈建議使用 HTTP，而不是使用 TCP。若使用 TCP，Lambda 平台無法確認何時將遙測資料傳遞至應用程式層。因此，如果您的延伸項目損毀，遙測資料可能會遺失。HTTP 沒有此限制。

訂閱以接收遙測資料之前，需先建立本機 HTTP 接聽程式或 TCP 連接埠。在設定期間，請注意下列事項：

- Lambda 只會將遙測資料傳送至執行環境內的目的地。
- 如果沒有接聽程式，或者如果 POST 請求遇到錯誤，則 Lambda 會重新嘗試傳送遙測資料 (使用回詢)。如果遙測接聽程式損毀，會在 Lambda 重新啟動執行環境之後繼續接收遙測資料。
- Lambda 保留連接埠 9001。沒有其他連接埠編號限制或建議。

設定記憶體使用量和緩衝

隨著訂閱用戶數量增加，執行環境的記憶體使用量會線性增加。訂閱會耗用記憶體資源，因為每個訂閱都會開啟新的記憶體緩衝區來存放遙測資料。緩衝區記憶體用量會計入執行環境中的整體記憶體耗用量。

透過遙測 API 來訂閱以接收遙測資料時，您可以選擇先緩衝遙測資料再批次傳遞給訂閱用戶。若要最佳化記憶體用量，您可以指定緩衝組態：

```
{
    "buffering": {
        "maxBytes": 256*1024,
        "maxItems": 1000,
        "timeoutMs": 100
    }
}
```

參數	Description	預設值和限制
maxBytes	記憶體中要緩衝的遙測資料量上限 (位元組)。	預設：262,144 下限：262,144 上限：1,048,576
maxItems	記憶體中要緩衝的事件數目上限。	預設：10,000 下限：1,000 上限：10,000
timeoutMs	緩衝一個批次的時間上限 (毫秒)。	預設：1,000 下限：25 上限：30,000

當設定緩衝時，請記住以下幾點：

- 如果有任何輸入串流關閉，則 Lambda 會排清日誌。例如，如果執行期損毀，就可能會發生這種情況。
- 每個訂閱用戶可以在訂閱請求中自訂其緩衝組態。
- 決定讀取資料的緩衝區大小時，請預期接收的有效負載大小為 2 * maxBytes + metadataBytes (其中 maxBytes 是緩衝設定的元件)。若要評估要考量的 metadataBytes 數量，請檢閱下列中繼資料。Lambda 會將類似的中繼資料新增至每筆記錄：

```
{
   "time": "2022-08-20T12:31:32.123Z",
```

```
    "type": "function",
    "record": "Hello World"
}
```

- 如果訂閱者處理傳入遙測資料的速度不夠快，或是函數程式碼產生了極大量的日誌，Lambda 可能會捨棄記錄，以確保記憶體使用率維持在限制範圍內。發生這種情況時，Lambda 會傳送 `platform.logsDropped` 事件。

將訂閱請求傳送至遙測 API

Lambda 延伸項目可透過將訂閱請求傳送至遙測 API 來訂閱以接收遙測資料。訂閱請求應包含您希望延伸項目訂閱的事件類型相關資訊。此外，請求可以包含傳遞目的地資訊和緩衝組態。

傳送訂閱請求之前，您必須有延伸功能 ID (`Lambda-Extension-Identifier`)。向延伸 API 註冊您的延伸項目時，您可從 API 回應中取得延伸項目 ID。

訂閱會在延伸項目初始化階段進行。下列範例顯示訂閱全部三個遙測串流的 HTTP 請求：平台遙測、函數日誌和延伸項目日誌。

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2025-01-29",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   },
   "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
   }
}
```

如果請求成功，訂閱者會收到 HTTP 200 成功回應。

```
HTTP/1.1 200 OK
```

```
"OK"
```

輸入遙測 API 訊息

使用遙測 API 訂閱之後，延伸項目會自動透過 POST 請求開始接收來自 Lambda 的遙測資料。每個 POST 請求主體包含 Event 對象的數組。每個 Event 項目都有以下結構描述：

```
{
   time: String,
   type: String,
   record: Object
}
```

- time 屬性定義了 Lambda 平台產生事件的時間。這與事件實際發生的時間不同。time 的字串值是 ISO 8601 格式的時間戳記。
- type 屬性定義了事件類型。下表說明所有可能的值。
- record 屬性定義了包含遙測資料的 JSON 物件。此 JSON 物件的結構描述取決於 type。

> **使用並行調用的事件排序**
>
> 對於 Lambda 受管執行個體，多個函數調用可以在相同的執行環境中同時執行。在此情況下，不同並行調用之間無法保證 platform.start和 platform.report事件的順序。延伸模組必須處理平行執行的多個調用的事件，不應採用循序排序。
> 為了將事件適當地歸因於特定調用，延伸模組應使用這些平台事件中存在requestId的欄位。每個調用都有唯一的請求 ID，該請求 ID 在該調用的所有事件中都保持一致，允許延伸項目即使在不按順序到達時也能正確地關聯事件。

下表摘要說明 Event 物件的所有類型，以及每個事件類型之遙測 API Event 結構描述參考的連結。

Category	事件類型	Description	事件記錄結構描述
平台事件	platform.initStart	函數初始化已開始。	the section called “platform.initStart” 結構描述

Category	事件類型	Description	事件記錄結構描述
平台事件	platform.initRuntimeDone	函數初始化已完成。	the section called "platform.initRuntimeDone" 結構描述
平台事件	platform.initReport	函數初始化報告。	the section called "platform.initReport" 結構描述
平台事件	platform.start	函數調用已開始。	the section called "platform.start" 結構描述
平台事件	platform.runtimeDone	執行階段已完成處理的事件，結果為成功或失敗。	the section called "platform.runtimeDone" 結構描述
平台事件	platform.report	函數調用報告。	the section called "platform.report" 結構描述
平台事件	platform.restoreStart	執行時間還原已開始。	the section called "platform.restoreStart" 結構描述
平台事件	platform.restoreRuntimeDone	執行時間還原已完成。	the section called "platform.restoreRuntimeDone" 結構描述

Category	事件類型	Description	事件記錄結構描述
平台事件	`platform.restoreReport`	執行時間還原報告。	the section called “`platform.restoreReport`” 結構描述
平台事件	`platform.telemetrySubscription`	延伸項目已訂閱遙測 API。	the section called “`platform.telemetrySubscription`” 結構描述
平台事件	`platform.logsDropped`	Lambda 已捨棄日誌項目。	the section called “`platform.logsDropped`” 結構描述
函數日誌	`function`	函數程式碼的日誌行。	the section called “`function`” 結構描述
延伸項目日誌	`extension`	延伸項目程式碼的日誌行。	the section called “`extension`” 結構描述

Lambda 遙測 API 參考

使用 Lambda 遙測 API 端點來讓延伸項目訂閱遙測串流。您可以從 AWS_LAMBDA_RUNTIME_API 環境變數中擷取遙測 API 端點。若要傳送 API 請求，請附加 API 版本 (2022-07-01/) 和telemetry/。例如：

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

如需訂閱回應版本 2025-01-29 的 OpenAPI 規格 (OAS) 定義，請參閱：

- HTTP – telemetry-api-http-schema.zip
- TCP – telemetry-api-tcp-schema.zip

API 操作

- 訂閱

訂閱

若要訂閱遙測串流，Lambda 延伸項目可以傳送訂閱 API 請求。

- 路徑 – /telemetry
- Method – PUT
- 標頭
  - Content-Type: application/json
- 請求內文參數
  - schemaVersion
    - 必要：是
    - 類型：字串
    - 有效值："2025-01-29"、"2022-12-13" 或 "2022-07-01"
    - 注意：Lambda 受管執行個體需要 "2025-01-29"。此版本與 Lambda （預設） 函數回溯相容。
  - destination – 定義遙測事件目的地和事件傳遞通訊協定的組態設定。
    - 必要：是
    - 類型：物件

```
{
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080"
}
```

- protocol – Lambda 用來傳送遙測資料的通訊協定。
  - 必要：是
  - 類型：字串
  - 有效值："HTTP"|"TCP"
- URI – 要傳送遙測資料的目的地 URI。
  - 必要：是
  - 類型：字串
- 如需詳細資訊，請參閱the section called “指定目的地通訊協定”。

- types – 您希望延伸項目訂閱的遙測類型。
  - 必要：是
  - 類型：字串陣列
  - 有效值："platform"|"function"|"extension"
- buffering – 事件緩衝的組態設定。
  - 必要：否
  - 類型：物件

```
{
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   }
}
```

  - maxItems – 記憶體中要緩衝的事件數目上限。
    - 必要：否
    - 類型：整數
    - 預設：1,000
    - 下限：1,000

    - 上限：10,000
- maxBytes – 記憶體中要緩衝的遙測資料量上限 (位元組)。
    - 必要：否
    - 類型：整數
    - 預設：262,144
    - 下限：262,144
    - 上限：1,048,576
- timeoutMs - 緩衝批次處理的時間上限 (毫秒)。
    - 必要：否
    - 類型：整數
    - 預設：1,000
    - 下限：25
    - 上限：30,000
- 如需詳細資訊，請參閱the section called “設定記憶體使用量和緩衝”。

訂閱 API 請求範例

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry HTTP/1.1
{
   "schemaVersion": "2025-01-29",
   "types": [
        "platform",
        "function",
        "extension"
   ],
   "buffering": {
        "maxItems": 1000,
        "maxBytes": 256*1024,
        "timeoutMs": 100
   },
   "destination": {
        "protocol": "HTTP",
        "URI": "http://sandbox.localdomain:8080"
   }
}
```

如果請求成功，延伸項目會收到 HTTP 200 成功回應：

```
HTTP/1.1 200 OK
"OK"
```

如果訂閱請求失敗，延伸項目會收到錯誤回應。例如：

```
HTTP/1.1 400 OK
{
    "errorType": "ValidationError",
    "errorMessage": "URI port is not provided; types should not be empty"
}
```

以下是延伸項目可能收到的其他幾個回應代碼：

- 200 - 請求已成功完成
- 202 - 已接受請求。本機測試環境中的訂閱請求回應
- 400 - 錯誤的請求
- 500 - 服務錯誤

Lambda 遙測 API **Event** 結構描述參考

使用 Lambda 遙測 API 端點來讓延伸項目訂閱遙測串流。您可以從 AWS_LAMBDA_RUNTIME_API 環境變數中擷取遙測 API 端點。若要傳送 API 請求，請附加 API 版本 (2022-07-01/) 和telemetry/。例如：

```
http://${AWS_LAMBDA_RUNTIME_API}/2022-07-01/telemetry/
```

如需訂閱回應版本 2025-01-29 的 OpenAPI 規格 (OAS) 定義，請參閱：

- HTTP – telemetry-api-http-schema.zip
- TCP – telemetry-api-tcp-schema.zip

下表摘要說明遙測 API 支援的所有 Event 物件類型。

Category	事件類型	Description	事件記錄結構描述
平台事件	platform.initStart	函數初始化已開始。	the section called “platform.initStart ” 結構描述
平台事件	platform.initRuntimeDone	函數初始化已完成。	the section called “platform.initRuntimeDone ” 結構描述
平台事件	platform.initReport	函數初始化報告。	the section called “platform.initReport ” 結構描述
平台事件	platform.start	函數調用已開始。	the section called “platform.start ” 結構描述

Category	事件類型	Description	事件記錄結構描述
平台事件	platform.runtimeDone	執行階段已完成處理的事件，結果為成功或失敗。	the section called “platform.runtimeDone ”結構描述
平台事件	platform.report	函數調用報告。	the section called “platform.report ”結構描述
平台事件	platform.restoreStart	執行時間還原已開始。	the section called “platform.restoreStart ”結構描述
平台事件	platform.restoreRuntimeDone	執行時間還原已完成。	the section called “platform.restoreRuntimeDone ”結構描述
平台事件	platform.restoreReport	執行時間還原報告。	the section called “platform.restoreReport ”結構描述
平台事件	platform.telemetrySubscription	延伸項目已訂閱遙測 API。	the section called “platform.telemetrySubscription ”結構描述
平台事件	platform.logsDropped	Lambda 已捨棄日誌項目。	the section called “platform.logsDropped ”結構描述

Category	事件類型	Description	事件記錄結構描述
函數日誌	`function`	函數程式碼的日誌行。	the section called “`function`” 結構描述
延伸項目日誌	`extension`	延伸項目程式碼的日誌行。	the section called “`extension `” 結構描述

內容

- 遙測 API Event 物件類型
  - platform.initStart
  - platform.initRuntimeDone
  - platform.initReport
  - platform.start
  - platform.runtimeDone
  - platform.report
  - platform.restoreStart
  - platform.restoreRuntimeDone
  - platform.restoreReport
  - platform.extension
  - platform.telemetrySubscription
  - platform.logsDropped
  - function
  - extension
- 共用物件類型
  - InitPhase
  - InitReportMetrics
  - InitType
  - ReportMetrics
  - RestoreReportMetrics
  - RuntimeDoneMetrics

- Span
- Status
- TraceContext
- TracingType

遙測 API **Event** 物件類型

本節詳細說明 Lambda 遙測 API 支援的 Event 物件類型。在事件描述中，問號 (?) 表示該屬性可能不存在於物件中。

platform.initStart

platform.initStart 事件表示函數初始化階段已開始。platform.initStart Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.initStart
- record: PlatformInitStart
```

PlatformInitStart 物件具有下列屬性：

- functionName – String
- functionVersion – String
- initializationType – the section called “InitType” 物件
- instanceId? – String
- instanceMaxMemory? – Integer
- phase – the section called “InitPhase” 物件
- runtimeVersion? – String
- runtimeVersionArn? – String

以下是 platform.initStart 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.initStart",
```

```
    "record": {
        "initializationType": "on-demand",
        "phase": "init",
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

platform.initRuntimeDone

platform.initRuntimeDone 事件表示函數初始化階段已完成。platform.initRuntimeDone Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.initRuntimeDone
- record: PlatformInitRuntimeDone
```

PlatformInitRuntimeDone 物件具有下列屬性：

- initializationType – the section called “InitType” 物件
- phase – the section called “InitPhase” 物件
- status – the section called “Status” 物件
- spans? – the section called “Span” 物件的清單

以下是 platform.initRuntimeDone 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initRuntimeDone",
    "record": {
        "initializationType": "on-demand"
        "status": "success",
        "spans": [
            {
                "name": "someTimeSpan",
```

```
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 70.5
            }
        ]
    }
}
```

platform.initReport

platform.initReport 事件包含函數初始化階段的整體報告。platform.initReport Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.initReport
- record: PlatformInitReport
```

PlatformInitReport 物件具有下列屬性：

- errorType? - 字串
- initializationType – the section called “InitType” 物件
- phase – the section called “InitPhase” 物件
- metrics – the section called “InitReportMetrics” 物件
- spans? – the section called “Span” 物件的清單
- status – the section called “Status” 物件

以下是 platform.initReport 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.initReport",
    "record": {
        "initializationType": "on-demand",
        "status": "success",
        "phase": "init",
        "metrics": {
            "durationMs": 125.33
        },
        "spans": [
```

```
            {
                "name": "someTimeSpan",
                "start": "2022-06-02T12:02:33.913Z",
                "durationMs": 90.1
            }
        ]
    }
}
```

platform.start

platform.start 事件表示函數調用階段已開始。platform.start Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.start
- record: PlatformStart
```

PlatformStart 物件具有下列屬性：

- requestId – String
- version? – String
- tracing? – the section called “TraceContext”

以下是 platform.start 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.start",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
        "version": "$LATEST",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        }
    }
}
```

platform.runtimeDone

platform.runtimeDone 事件表示函數調用階段已完成。platform.runtimeDone Event 物件的結構如下：

> **Lambda 受管執行個體**
>
> Lambda 受管執行個體不支援 platform.runtimeDone事件。在受管執行個體上執行的延伸模組不會收到此事件，因為延伸模組無法訂閱受管執行個體上的INVOKE事件。由於並行執行模型可同時處理多個調用，延伸模組無法對個別調用執行調用後處理，因為它們通常在 Lambda （預設） 函數上執行。
> 對於受管執行個體，通常包含在 中的 responseLatency和 responseDuration 範圍platform.runtimeDone可改為在 platform.report 事件中使用。如需詳細資訊，請參閱 the section called “platform.report”。

```
Event: Object
- time: String
- type: String = platform.runtimeDone
- record: PlatformRuntimeDone
```

PlatformRuntimeDone 物件具有下列屬性：

- errorType? – String
- metrics? – the section called “RuntimeDoneMetrics” 物件
- requestId – String
- status – the section called “Status” 物件
- spans? – the section called “Span” 物件的清單
- tracing? – the section called “TraceContext” 物件

以下是 platform.runtimeDone 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.runtimeDone",
    "record": {
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
```

```
        "status": "success",
        "tracing": {
            "spanId": "54565fb41ac79632",
            "type": "X-Amzn-Trace-Id",
            "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
        ],
        "metrics": {
            "durationMs": 140.0,
            "producedBytes": 16
        }
    }
}
```

platform.report

platform.report 事件包含函數調用階段的整體報告。platform.report Event 物件的結構如下：

> **Lambda 受管執行個體**
>
> 與 Lambda （預設）函數相比，Lambda 受管執行個體platform.report的事件具有不同的指標和範圍。對於受管執行個體：
>
> - 範圍：包含 responseLatency和 responseDuration而非 extensionOverhead。extensionOverhead 範圍不可用，因為由於並行執行模型，延伸模組無法訂閱受管執行個體上的INVOKE事件。
> - 指標：僅包含 durationMs。不包含下列指標：billedDurationMs、maxMemoryUsedMB、 initDurationMs和 memorySizeMB。這些每次調用指標不適用於並行執行環境。如需資源使用率指標，請使用監控 Lambda 受管執行個體或 Lambda Insights。

```
Event: Object
```

```
- time: String
- type: String = platform.report
- record: PlatformReport
```

PlatformReport 物件具有下列屬性：

- metrics – the section called "ReportMetrics" 物件
- requestId – String
- spans? – the section called "Span" 物件的清單
- status – the section called "Status" 物件
- tracing? – the section called "TraceContext" 物件

以下是 platform.report 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:01:15.000Z",
    "type": "platform.report",
    "record": {
        "metrics": {
            "billedDurationMs": 694,
            "durationMs": 693.92,
            "initDurationMs": 397.68,
            "maxMemoryUsedMB": 84,
            "memorySizeMB": 128
        },
        "requestId": "6d68ca91-49c9-448d-89b8-7ca3e6dc66aa",
    }
}
```

platform.restoreStart

platform.restoreStart 事件表示函數環境還原事件已開始。在環境還原事件中，Lambda 會從快取的快照建立環境，而不是從頭開始初始化。如需詳細資訊，請參閱Lambda SnapStart。platform.restoreStart Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.restoreStart
- record: PlatformRestoreStart
```

PlatformRestoreStart 物件具有下列屬性：

- functionName – String
- functionVersion – String
- instanceId? – String
- instanceMaxMemory? – String
- runtimeVersion? – String
- runtimeVersionArn? – String

以下是 platform.restoreStart 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreStart",
    "record": {
        "runtimeVersion": "nodejs-14.v3",
        "runtimeVersionArn": "arn",
        "functionName": "myFunction",
        "functionVersion": "$LATEST",
        "instanceId": "82561ce0-53dd-47d1-90e0-c8f5e063e62e",
        "instanceMaxMemory": 256
    }
}
```

platform.restoreRuntimeDone

platform.restoreRuntimeDone 事件表示函數環境還原事件已完成。在環境還原事件中，Lambda 會從快取的快照建立環境，而不是從頭開始初始化。如需詳細資訊，請參閱Lambda SnapStart。platform.restoreRuntimeDone Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.restoreRuntimeDone
- record: PlatformRestoreRuntimeDone
```

PlatformRestoreRuntimeDone 物件具有下列屬性：

- errorType? – String
- spans? – the section called “Span” 物件的清單

- status – the section called “Status” 物件

以下是 platform.restoreRuntimeDone 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreRuntimeDone",
    "record": {
        "status": "success",
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 80.0
            }
        ]
    }
}
```

platform.restoreReport

platform.restoreReport 事件包含函數還原事件的整體報告。platform.restoreReport Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.restoreReport
- record: PlatformRestoreReport
```

PlatformRestoreReport 物件具有下列屬性：

- errorType? - 字串
- metrics? – the section called “RestoreReportMetrics” 物件
- spans? – the section called “Span” 物件的清單
- status – the section called “Status” 物件

以下是 platform.restoreReport 類型 Event 的範例：

```
{
```

```
    "time": "2022-10-12T00:00:15.064Z",
    "type": "platform.restoreReport",
    "record": {
        "status": "success",
        "metrics": {
            "durationMs": 15.19
        },
        "spans": [
            {
                "name": "someTimeSpan",
                "start": "2022-08-02T12:01:23:521Z",
                "durationMs": 30.0
            }
        ]
    }
}
```

platform.extension

extension 事件包含延伸項目程式碼的日誌。extension Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

PlatformExtension 物件具有下列屬性：

- events – String 清單
- name – String
- state – String

以下是 platform.extension 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:02:15.000Z",
    "type": "platform.extension",
    "record": {
        "events": [ "INVOKE", "SHUTDOWN" ],
        "name": "my-telemetry-extension",
        "state": "Ready"
```

```
    }
}
```

platform.telemetrySubscription

platform.telemetrySubscription 事件包含延伸項目訂閱的相關資訊。platform.telemetrySubscription Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = platform.telemetrySubscription
- record: PlatformTelemetrySubscription
```

PlatformTelemetrySubscription 物件具有下列屬性：

- name – String
- state – String
- types – String 清單

以下是 platform.telemetrySubscription 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.telemetrySubscription",
    "record": {
        "name": "my-telemetry-extension",
        "state": "Subscribed",
        "types": [ "platform", "function" ]
    }
}
```

platform.logsDropped

platform.logsDropped 事件包含已捨棄事件的相關資訊。當函數以過高速率輸出日誌，使得 Lambda 無法及時處理時，Lambda 會發出 platform.logsDropped 事件。當 Lambda 無法以函數產生日誌的速度將它們傳送到 CloudWatch 或訂閱了遙測 API 的延伸時，它會捨棄日誌以防止函數的執行速度變慢。platform.logsDropped Event 物件的結構如下：

```
Event: Object
- time: String
```

```
- type: String = platform.logsDropped
- record: PlatformLogsDropped
```

PlatformLogsDropped 物件具有下列屬性：

- droppedBytes – Integer
- droppedRecords – Integer
- reason – String

以下是 platform.logsDropped 類型 Event 的範例：

```
{
    "time": "2022-10-12T00:02:35.000Z",
    "type": "platform.logsDropped",
    "record": {
        "droppedBytes": 12345,
        "droppedRecords": 123,
        "reason": "Some logs were dropped because the downstream consumer is slower
 than the logs production rate"
    }
}
```

function

function 事件包含函數程式碼的日誌。function Event 物件的結構如下：

```
Event: Object
- time: String
- type: String = function
- record: {}
```

record 欄位的格式取決於函數的日誌檔是以純文字格式還是 JSON 格式而定。若要進一步瞭解日誌格式設定選項，請參閱 the section called “日誌格式”

以下是日誌格式為純文字時類型 function 的範例 Event。

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": "[INFO] Hello world, I am a function!"
```

```
}
```

以下是日誌格式為 JSON 時的類型 `function` 的範例 `Event`。

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "function",
    "record": {
        "timestamp": "2022-10-12T00:03:50.000Z",
        "level": "INFO",
        "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
        "message": "Hello world, I am a function!"
    }
}
```

> **Note**
>
> 如果您使用的描述版本比 2022-12-13 版本舊，則即使函數的 `"record"` 日誌格式配置為 JSON，也始終將其呈現為字串。對於 Lambda 受管執行個體，您必須使用結構描述版本 2025-01-29。

`extension`

`extension` 事件包含延伸項目程式碼的日誌。`extension` `Event` 物件的結構如下：

```
Event: Object
- time: String
- type: String = extension
- record: {}
```

`record` 欄位的格式取決於函數的日誌檔是以純文字格式還是 JSON 格式而定。若要進一步瞭解日誌格式設定選項，請參閱 the section called “日誌格式”

以下是日誌格式為純文字時類型 `extension` 的範例 `Event`。

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": "[INFO] Hello world, I am an extension!"
```

```
}
```

以下是日誌格式為 JSON 時的類型 `extension` 的範例 Event。

```
{
    "time": "2022-10-12T00:03:50.000Z",
    "type": "extension",
    "record": {
       "timestamp": "2022-10-12T00:03:50.000Z",
       "level": "INFO",
       "requestId": "79b4f56e-95b1-4643-9700-2807f4e68189",
       "message": "Hello world, I am an extension!"
    }
}
```

> **Note**
>
> 如果您使用的描述版本比 2022-12-13 版本舊，則即使函數的 `"record"` 日誌格式配置為 JSON，也始終將其呈現為字串。對於 Lambda 受管執行個體，您必須使用結構描述版本 2025-01-29。

共用物件類型

本節詳細說明 Lambda 遙測 API 支援的共用物件類型。

`InitPhase`

字串列舉，描述初始化步驟發生時的階段。多數情況下，Lambda 會在 `init` 階段執行函數初始化程式碼。但是在某些錯誤情況下，Lambda 可能會在 `invoke` 階段重新執行函數初始化程式碼。(這稱為隱藏的初始化。)

- 類型 – `String`
- 有效值 – `init|invoke|snap-start`

`InitReportMetrics`

包含初始化階段相關指標的物件。

- 類型 – `Object`

InitReportMetrics 物件的結構如下：

```
InitReportMetrics: Object
- durationMs: Double
```

下列為 InitReportMetrics 物件的範例:

```
{
    "durationMs": 247.88
}
```

InitType

字串列舉，描述 Lambda 如何初始化環境。

- 類型 – String
- 有效值 – on-demand|provisioned-concurrency

ReportMetrics

包含已完成階段相關指標的物件。

- 類型 – Object

ReportMetrics 物件的結構如下：

```
ReportMetrics: Object
- billedDurationMs: Integer
- durationMs: Double
- initDurationMs?: Double
- maxMemoryUsedMB: Integer
- memorySizeMB: Integer
- restoreDurationMs?: Double
```

下列為 ReportMetrics 物件的範例:

```
{
    "billedDurationMs": 694,
    "durationMs": 693.92,
    "initDurationMs": 397.68,
```

```
    "maxMemoryUsedMB": 84,
    "memorySizeMB": 128
}
```

RestoreReportMetrics

包含已完成還原階段相關指標的物件。

- 類型 – Object

RestoreReportMetrics 物件的結構如下：

```
RestoreReportMetrics: Object
- durationMs: Double
```

下列為 RestoreReportMetrics 物件的範例:

```
{
    "durationMs": 15.19
}
```

RuntimeDoneMetrics

包含調用階段相關指標的物件。

- 類型 – Object

RuntimeDoneMetrics 物件的結構如下：

```
RuntimeDoneMetrics: Object
- durationMs: Double
- producedBytes?: Integer
```

下列為 RuntimeDoneMetrics 物件的範例:

```
{
    "durationMs": 200.0,
    "producedBytes": 15
}
```

Span

包含跨度詳細資訊的物件。跨度表示追蹤中的工作或作業單位。如需跨度的詳細資訊，請參閱 OpenTelemetry 文件網站的追蹤 API 頁面上的跨度。

Lambda 針對 `platform.RuntimeDone` 事件支援下列跨度：

- `responseLatency` 跨度描述 Lambda 函數開始傳送回應所花費的時間。
- `responseDuration` 跨度描述 Lambda 函數完成傳送整個回應所花費的時間。
- `runtimeOverhead` 跨度描述 Lambda 執行期表示已準備好處理下一個函數調用所花費的時間。這是執行期傳回函數回應後，呼叫下一個調用 API 所花費的時間。

下列為 `responseLatency` 跨度物件的範例:

```
{
        "name": "responseLatency",
        "start": "2022-08-02T12:01:23.521Z",
        "durationMs": 23.02
      }
```

Status

描述初始化或呼叫階段狀態的物件。如果狀態為 `failure` 或 `error`，則 `Status` 物件也會包含描述錯誤的 `errorType` 欄位。

- 類型 – `Object`
- 有效狀態值 – `success|failure|error|timeout`

TraceContext

描述追蹤屬性的物件。

- 類型 – `Object`

`TraceContext` 物件的結構如下：

```
TraceContext: Object
- spanId?: String
- type: TracingType enum
```

```
- value: String
```

下列為 TraceContext 物件的範例:

```
{
    "spanId": "073a49012f3c312e",
    "type": "X-Amzn-Trace-Id",
    "value":
 "Root=1-62e900b2-710d76f009d6e7785905449a;Parent=0efbd19962d95b05;Sampled=1"
}
```

TracingType

字串列舉，描述 the section called “TraceContext” 物件中追蹤的類型。

- 類型 – String
- 有效值 – X-Amzn-Trace-Id

將 Lambda 遙測 API`Event` 物件轉換為 OpenTelemetry 跨度

AWS Lambda 遙測 API 結構描述在語義上與 OpenTelemetry (OTel) 相容。這表示您可以將 AWS Lambda 遙測 API `Event` 物件轉換為 OpenTelemetry (OTel) 範圍。轉換時，不應將單一 `Event` 物件映射到單一 oTel 跨度。而是應該在單一 oTel 跨度中顯示與生命週期階段相關的全部三個事件。例如，`start`、`runtimeDone` 和 `runtimeReport` 事件代表單一函數叫用。將這三個事件做為一個單一的 oTel 跨度呈現。

您可以使用跨度事件或子 (巢狀) 跨度轉換事件。此頁面上的表格針對這兩種做法，說明遙測 API 結構描述屬性與 oTel 跨度屬性之間的映射。如需 OTel 跨度的詳細資訊，請參閱 OpenTelemetry 文件網站的追蹤 API 頁面上的跨度。

章節

- 透過跨度事件映射到 OTel 跨度
- 透過子跨度映射到 OTel 跨度

透過跨度事件映射到 OTel 跨度

在下表中，e 代表來自遙測來源的事件。

映射 *Start 事件

OpenTelemetry	Lambda 遙測 API 結構描述
`Span.Name`	延伸項目會根據 `type` 欄位產生此值。
`Span.StartTime`	請使用 `e.time`。
`Span.EndTime`	N/A，因為事件尚未完成。
`Span.Kind`	設定為 `Server`。
`Span.Status`	設定為 `Unset`。
`Span.TraceId`	剖析 中找到的 AWS X-Ray 標頭`e.tracing.value` ，然後使用 `TraceId`值。
`Span.ParentId`	剖析 `e.tracing.value` 中找到的 X-Ray 標頭，然後使用 `Parent` 值。

OpenTelemetry	Lambda 遙測 API 結構描述
Span.SpanId	使用 e.tracing.spanId （如果可用)。若無法使用，請產生一個新的 SpanId。
Span.SpanContext.TraceState	X-Ray 追蹤內容則為 N/A。
Span.SpanContext.TraceFlags	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Sampled 值。
Span.Attributes	延伸項目可以在此處新增任何自訂值。

映射 *RuntimeDone 事件

OpenTelemetry	Lambda 遙測 API 結構描述
Span.Name	延伸項目會根據 type 欄位產生值。
Span.StartTime	使用相符 *Start 事件中的 e.time。 或使用 e.time - e.metrics.durationMs 。
Span.EndTime	N/A，因為事件尚未完成。
Span.Kind	設定為 Server。
Span.Status	如果 e.status 不等於 success，則設定為 Error。 否則，請設定為 Ok。
Span.Events[]	請使用 e.spans[] 。
Span.Events[i].Name	請使用 e.spans[i].name 。
Span.Events[i].Time	請使用 e.spans[i].start 。

OpenTelemetry	Lambda 遙測 API 結構描述
Span.TraceId	剖析 中找到的 AWS X-Ray 標頭e.tracing .value ，然後使用 TraceId值。
Span.ParentId	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Parent 值。
Span.SpanId	使用 *Start 事件的相同 SpanId。如果無法使用，則使用 e.tracing.spanId 或產生新的 SpanId。
Span.SpanContext.TraceState	X-Ray 追蹤內容則為 N/A。
Span.SpanContext.TraceFlags	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Sampled 值。
Span.Attributes	延伸項目可以在此處新增任何自訂值。

映射 *Report 事件

OpenTelemetry	Lambda 遙測 API 結構描述
Span.Name	延伸項目會根據 type 欄位產生值。
Span.StartTime	使用相符 *Start 事件中的 e.time。 或使用 e.time - e.metrics.durationMs 。
Span.EndTime	請使用 e.time。
Span.Kind	設定為 Server。
Span.Status	使用與 *RuntimeDone 事件相同的值。
Span.TraceId	剖析 中找到的 AWS X-Ray 標頭e.tracing .value ，然後使用 TraceId值。

OpenTelemetry	Lambda 遙測 API 結構描述
Span.ParentId	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Parent 值。
Span.SpanId	使用 *Start 事件的相同 SpanId。如果無法使用，則使用 e.tracing.spanId 或產生新的 SpanId。
Span.SpanContext.TraceState	X-Ray 追蹤內容則為 N/A。
Span.SpanContext.TraceFlags	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Sampled 值。
Span.Attributes	延伸項目可以在此處新增任何自訂值。

透過子跨度映射到 OTel 跨度

下表說明如何針對 *RuntimeDone 跨度，透過子 (巢狀) 跨度將 Lambda 遙測 API 事件轉換為 OTel 跨度。如需 *Start 和 *Report 映射，請參閱「the section called “透過跨度事件映射到 OTel 跨度”」中的表格，因為對子跨度來說是相同的。在本表中，e 代表來自遙測來源的事件。

映射 *RuntimeDone 事件

OpenTelemetry	Lambda 遙測 API 結構描述
Span.Name	延伸項目會根據 type 欄位產生值。
Span.StartTime	使用相符 *Start 事件中的 e.time。 或使用 e.time - e.metrics.durationMs 。
Span.EndTime	N/A，因為事件尚未完成。
Span.Kind	設定為 Server。
Span.Status	如果 e.status 不等於 success，則設定為 Error。

OpenTelemetry	Lambda 遙測 API 結構描述
	否則，請設定為 Ok。
Span.TraceId	剖析 中找到的 AWS X-Ray 標頭e.tracing .value ，然後使用 TraceId值。
Span.ParentId	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Parent 值。
Span.SpanId	使用 *Start 事件的相同 SpanId。如果無法使用，則使用 e.tracing.spanId 或產生新的 SpanId。
Span.SpanContext.TraceState	X-Ray 追蹤內容則為 N/A。
Span.SpanContext.TraceFlags	剖析 e.tracing.value 中找到的 X-Ray 標頭，然後使用 Sampled 值。
Span.Attributes	延伸項目可以在此處新增任何自訂值。
ChildSpan[i].Name	請使用 e.spans[i].name 。
ChildSpan[i].StartTime	請使用 e.spans[i].start 。
ChildSpan[i].EndTime	請使用 e.spans[i].start + e.spans[i].durations 。
ChildSpan[i].Kind	與父項 Span.Kind 相同。
ChildSpan[i].Status	與父項 Span.Status 相同。
ChildSpan[i].TraceId	與父項 Span.TraceId 相同。
ChildSpan[i].ParentId	使用父項 Span.SpanId 。
ChildSpan[i].SpanId	產生新的 SpanId。
ChildSpan[i].SpanContext.Tr aceState	X-Ray 追蹤內容則為 N/A。

OpenTelemetry	Lambda 遙測 API 結構描述
`ChildSpan[i].SpanContext.TraceFlags`	與父項 `Span.SpanContext.TraceFlags` 相同。

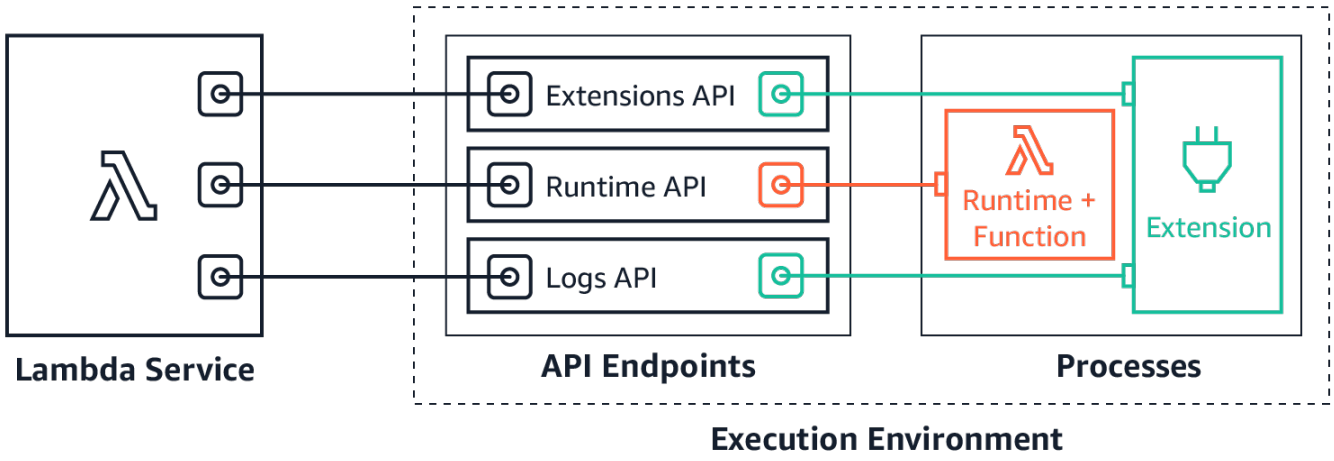
使用 Lambda Logs API

> **Important**
>
> Lambda 遙測 API 優先於 Lambda 日誌 API。雖然日誌 API 仍然正常運作，但我們建議您未來只使用遙測 API。您可以使用遙測 API 或日誌 API 訂閱遙測串流的延伸項目。使用其中一個 API 進行訂閱後，若嘗試使用其他 API 進行任何訂閱，都會傳回錯誤。

> **Lambda 受管執行個體不支援 Logs API**
>
> Lambda 受管執行個體不支援 Logs API。如果您使用的是受管執行個體函數，請改用遙測 API。遙測 API 提供從 Lambda 函數收集和處理遙測資料的增強功能。

Lambda 會自動擷取執行時間日誌並將其串流至 Amazon CloudWatch。此日誌串流包含函數程式碼和延伸項目產生的日誌，以及 Lambda 作為函數調用一部分產生的日誌。

Lambda 延伸項目可以使用 Lambda Runtime Logs API，直接從 Lambda 執行環境中訂閱記錄串流。Lambda 會將日誌串流至延伸項目，延伸項目隨後可處理、篩選日誌並將其傳送至任何偏好目的地。



Logs API 允許延伸項目訂閱三種不同的日誌串流：

- Lambda 函數產生並寫入到 `stdout` 或 `stderr` 的函數日誌。
- 延伸項目程式碼產生的延伸項目日誌。

- Lambda 平台日誌，它記錄與調用和延伸項目相關的事件和錯誤。

> **Note**
>
> 即便延伸項目訂閱一個或多個日誌串流，Lambda 也會將所有日誌傳送至 CloudWatch。

主題

- 訂閱接收日誌
- 記憶體用量
- 目的地協定
- 緩衝組態
- 訂閱範例
- Logs API 的範本程式碼
- Logs API 參考
- 日誌訊息

訂閱接收日誌

Lambda 延伸項目可透過傳送訂閱請求至 Logs API 來訂閱接收日誌。

若要訂閱接收日誌，您需要延伸項目識別符 (`Lambda-Extension-Identifier`)。首先註冊延伸項目以接收延伸項目識別符。然後在初始化期間訂閱 Logs API。初始化階段完成後，Lambda 不會處理訂閱請求。

> **Note**
>
> Logs API 訂閱為等冪操作。重複的訂閱請求不會導致重複訂閱。

記憶體用量

隨著訂閱者數量的增加，記憶體用量會線性增加。訂閱會耗用記憶體資源，因為每個訂閱都會開啟新的記憶體緩衝區來存放日誌。為了協助最佳化記憶體用量，您可以調整緩衝組態。緩衝區記憶體用量會計入執行環境中的整體記憶體耗用量。

目的地協定

您可以選擇下列其中一個協定來接收日誌：

1. HTTP (建議) - Lambda 會以 JSON 格式的記錄陣列將日誌傳遞至本機 HTTP 端點 (`http://sandbox.localdomain:${PORT}/${PATH}`)。$PATH 為選用參數。請注意，僅支援 HTTP，而不是 HTTPS。您可以選擇透過 PUT 或 POST 接收日誌。
2. TCP - Lambda 以換行分隔的 JSON (NDJSON) 格式將日誌傳遞至 TCP 連接埠。

建議使用 HTTP，而不是使用 TCP。使用 TCP，Lambda 平台無法確認何時將日誌傳遞至應用程式層。因此，如果您的延伸項目損毀，可能會遺失日誌。HTTP 不會共享此限制。

我們也建議您先設定本機 HTTP 接聽程式或 TCP 連接埠，然後再訂閱接收日誌。在設定期間，請注意下列事項：

- Lambda 只會將日誌傳送至執行環境內的目的地。
- 如果沒有接聽程式，或者如果 POST 或 PUT 請求導致錯誤，則 Lambda 會重新嘗試傳送日誌 (使用輪詢)。如果日誌訂閱者當機，則在 Lambda 重新啟動執行環境之後會繼續接收日誌。
- Lambda 保留連接埠 9001。沒有其他連接埠編號限制或建議。

緩衝組態

Lambda 可以緩衝日誌並將其提供給訂閱者。您可以透過指定下列選用欄位，在訂閱請求中設定此行為。請注意，Lambda 會對您未指定的任何欄位使用預設值。

- timeoutMs - 緩衝批次處理的時間上限 (毫秒)。預設：1,000。下限：25。上限：30,000。
- maxBytes - 記憶體中要緩衝的日誌大小上限 (位元組)。預設：262,144。下限：262,144。上限：1,048,576。
- maxItems – 記憶體中要緩衝的事件數目上限。預設：10,000。下限：1,000。上限：10,000。

在緩衝組態期間，請注意下列幾點：

- 如果任何輸入串流被關閉，例如，如果執行時間崩潰，Lambda 會清除日誌。
- 每個訂閱者可以在訂閱請求中指定不同的緩衝組態。
- 考慮讀取資料所需的緩衝區大小。預計接收最大為 `2*maxBytes+metadata` 的承載，其中 `maxBytes` 在訂閱請求中進行設定。例如，Lambda 將下列中繼資料位元組新增至每個記錄：

```
{
"time": "2020-08-20T12:31:32.123Z",
"type": "function",
"record": "Hello World"
}
```

- 如果訂閱者無法足夠快速地處理傳入日誌，Lambda 可能會丟棄日誌，以確保記憶體使用率受到限制。若要指示丟棄的記錄數量，Lambda 會傳送 `platform.logsDropped` 日誌。如需詳細資訊，請參閱the section called “Lambda：並非所有函數的日誌都會顯示”。

訂閱範例

訂閱平台和函數日誌的請求如下列範例所示。

```
PUT http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs HTTP/1.1
{ "schemaVersion": "2020-08-15",
  "types": [
      "platform",
      "function"
    ],
  "buffering": {
      "maxItems": 1000,
      "maxBytes": 262144,
      "timeoutMs": 100
    },
  "destination": {
    "protocol": "HTTP",
    "URI": "http://sandbox.localdomain:8080/lambda_logs"
  }
}
```

如果請求成功，訂閱者會收到 HTTP 200 成功回應。

```
HTTP/1.1 200 OK
"OK"
```

Logs API 的範本程式碼

如需示範如何將日誌傳送至自訂目的地的範例程式碼，請參閱 AWS 運算部落格上的使用 AWS Lambda 擴充功能將日誌傳送至自訂目的地。

對於顯示如何開發基本 Lambda 延伸及訂閱 Logs API 的 Python 和 Go 程式碼範例，請參閱 AWS Samples GitHub 儲存庫上的 AWS Lambda Extensions。如需建置 Lambda 延伸項目的詳細資訊，請參閱 the section called “Extensions API”。

Logs API 參考

您可以從 AWS_LAMBDA_RUNTIME_API 環境變數中擷取 Logs API 端點。若要傳送 API 請求，請在 API 路徑之前使用前綴 2020-08-15/。例如：

```
http://${AWS_LAMBDA_RUNTIME_API}/2020-08-15/logs
```

Logs API 的 OpenAPI 規範 (版本 2020-08-15) 可參閱此文件：logs-api-request.zip

訂閱

若要訂閱 Lambda 執行環境中可用的一個或多個日誌串流，延伸項目會傳送 Subscribe API 請求。

路徑 – /logs

方法 – PUT

主體參數

destination – 請參閱the section called “目的地協定”。必要：是。類型：字串。

buffering – 請參閱the section called “緩衝組態”。必要：否。類型：字串。

types - 要接收的日誌類型陣列。必要：是。類型：字串陣列。有效值：「平台」、「函數」、「延伸項目」。

schemaVersion - 必要：否。預設值："2020-08-15"。將延伸項目設定為 "2021-03-18"，可接收 platform.runtimeDone 訊息。

回應參數

訂閱回應的 OpenAPI 規範 (版本 2020-08-15)，適用於 HTTP 和 TCP 通訊協定：

- HTTP：logs-api-http-response.zip
- TCP：logs-api-tcp-response.zip

回應代碼

- 200 - 請求已成功完成

- 202 - 已接受請求。在本機測試期間回應訂閱請求。
- 4XX - 錯誤請求
- 500 - 服務錯誤

如果請求成功，訂閱者會收到 HTTP 200 成功回應。

```
HTTP/1.1 200 OK
"OK"
```

如果請求失敗，訂閱者會收到錯誤回應。例如：

```
HTTP/1.1 400 OK
{
    "errorType": "Logs.ValidationError",
    "errorMessage": URI port is not provided; types should not be empty"
}
```

日誌訊息

Logs API 允許延伸項目訂閱三種不同的日誌串流：

- 函數 - Lambda 函數產生並寫入到 `stdout` 或 `stderr` 的日誌。
- 延伸項目 - 延伸項目程式碼產生的日誌。
- 平台 - 執行時間平台產生的日誌，它們記錄與調用和延伸相關的事件和錯誤。

主題

- 函數日誌
- 延伸項目日誌
- 平台日誌

函數日誌

Lambda 函數和內部延伸項目會產生函數日誌並將它們寫入 `stdout` 或 `stderr`。

下列範例顯示函數日誌訊息的格式。{ "time": "2020-08-20T12:31:32.123Z", "type": "function", "record": "ERROR encountered. Stack trace:\n\my-function (line 10)\n" }

延伸項目日誌

延伸項目可產生延伸日誌。日誌格式與函數日誌的格式相同。

平台日誌

Lambda 會產生平台事件 (例如 platform.start、platform.end 以及 platform.fault) 的日誌訊息。

或者，您可以訂閱包含 platform.runtimeDone 日誌訊息的 2021-03-18 版本的 Logs API 結構描述。

平台日誌訊息範例

下面的範例顯示了平台開始和平台結束日誌。這些日誌指出 RequestID 所指定之調用的調用開始時間和調用結束時間。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.start",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.end",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56"}
}
```

platform.initRuntimeDone 日誌訊息顯示子階段 Runtime init 的狀態，該子階段是初始化生命週期階段的一部分。Runtime init 成功時，執行階段會傳送 /next 執行階段 API 請求 (針對 on-demand 和 provisioned-concurrency 初始化類型) 或 restore/next (針對 snap-start 初始化類型)。下列範例顯示 snap-start 初始化類型的成功 platform.initRuntimeDone 日誌訊息。

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initRuntimeDone",
  "record":{
      "initializationType":"snap-start",
      "status":"success"
  }
}
```

platform.initReport 日誌訊息會顯示 Init 階段持續的時間長度，以及您須為此階段支付的費用。當初始化類型為 provisioned-concurrency，Lambda 會在調用期間傳送此訊息。當初始化類型為 snap-start，Lambda 會在還原快照之後傳送此訊息。下列範例顯示 snap-start 初始化類型的 platform.initReport 日誌訊息。

```
{
  "time":"2022-07-17T18:41:57.083Z",
  "type":"platform.initReport",
  "record":{
      "initializationType":"snap-start",
      "metrics":{
          "durationMs":731.79,
          "billedDurationMs":732
          }
  }
}
```

平台報告日誌包含 RequestID 所指定之調用的相關指標。只有調用包含冷啟動時，initDurationMs 欄位才會包含在日誌中。如果 AWS X-Ray 追蹤處於作用中狀態，則日誌包含 X-Ray 中繼資料。下列範例顯示包含冷啟動之調用的平台報告日誌。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.report",
    "record": {"requestId": "6f7f0961f83442118a7af6fe80b88d56",
        "metrics": {"durationMs": 101.51,
            "billedDurationMs": 300,
            "memorySizeMB": 512,
            "maxMemoryUsedMB": 33,
            "initDurationMs": 116.67
        }
    }
}
```

平台故障日誌會擷取執行時間或執行環境錯誤。平台錯誤日誌訊息如下列範例所示。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.fault",
    "record": "RequestId: d783b35e-a91d-4251-af17-035953428a2c Process exited before
 completing request"
```

```
}
```

> **Note**
>
> AWS 目前正在對 Lambda 服務實作變更。由於這些變更，您可能會看到系統日誌訊息的結構和內容，與 AWS 帳戶中不同 Lambda 函數發出的追蹤區段之間存在細微差異。
> 受此變更影響的日誌輸出之一是平台故障日誌 "record" 欄位。下列範例顯示了舊格式和新格式的說明性 "record" 欄位。新式故障日誌包含更簡潔的訊息
> 這些變化將在未來幾週內實作，除中國和 GovCloud 區域以外，所有 AWS 區域 中的所有函數都會轉換至使用新格式的日誌訊息和追蹤區段。

Example平台故障日誌記錄 (舊式)

```
"record":"RequestId: ...\tError: Runtime exited with error: exit status
 255\nRuntime.ExitError"
```

Example平台故障日誌記錄 (新式)

```
"record":"RequestId: ... Status: error\tErrorType: Runtime.ExitError"
```

當延伸項目註冊延伸項目 API 時，Lambda 會產生平台延伸項目日誌。平台延伸項目訊息如下列範例所示。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.extension",
    "record": {"name": "Foo.bar",
        "state": "Ready",
        "events": ["INVOKE", "SHUTDOWN"]
     }
}
```

當延伸項目訂閱日誌 API 時，Lambda 會產生平台日誌訂閱日誌。日誌訂閱訊息如下列範例所示。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsSubscription",
```

```
    "record": {"name": "Foo.bar",
        "state": "Subscribed",
        "types": ["function", "platform"],
    }
}
```

當延伸項目無法處理正在接收的日誌數量時，Lambda 產生的日誌會捨棄平台日誌。platform.logsDropped 日誌訊息如下列範例所示。

```
{
    "time": "2020-08-20T12:31:32.123Z",
    "type": "platform.logsDropped",
    "record": {"reason": "Consumer seems to have fallen behind as it has not
 acknowledged receipt of logs.",
        "droppedRecords": 123,
        "droppedBytes" 12345
    }
}
```

platform.restoreStart 日誌訊息會顯示 Restore 階段開始的時間 (僅限 snap-start 初始化類型)。範例：

```
{
  "time":"2022-07-17T18:43:44.782Z",
  "type":"platform.restoreStart",
  "record":{}
}
```

platform.restoreReport 日誌訊息會顯示 Restore 階段持續的時間長度，以及您須為此階段付費的毫秒數 (僅限 snap-start 初始化類型)。範例：

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreReport",
  "record":{
      "metrics":{
          "durationMs":70.87,
          "billedDurationMs":13
      }
  }
}
```

平台 **runtimeDone** 訊息

如果您在訂閱請求中將結構描述版本設定為 "2021-03-18"，在函數調用成功完成或發生錯誤之後，Lambda 會傳送 `platform.runtimeDone` 訊息。延伸項目可以使用此訊息停止此函數調用的所有遙測集合。

結構描述版本 2021-03-18 中的日誌事件類型的 OpenAPI 規範可參閱此文件：schema-2021-03-18.zip

當執行時間傳送 `Next` 或 `Error` 執行時間 API 請求時，Lambda 會產生 `platform.runtimeDone` 日誌訊息。`platform.runtimeDone` 日誌會通知 Logs API 的使用者，告知他們函數調用完成。延伸項目可以使用此資訊來決定何時傳送該調用期間收集的所有遙測。

範例

當函數調用完成時，在執行時間傳送 NEXT 請求之後，Lambda 會傳送 `platform.runtimeDone` 訊息。下列範例顯示每個狀態值的訊息：成功、失敗和逾時。

Example成功訊息範例

```
{
    "time": "2021-02-04T20:00:05.123Z",
    "type": "platform.runtimeDone",
    "record": {
       "requestId":"6f7f0961f83442118a7af6fe80b88",
       "status": "success"
    }
}
```

Example失敗訊息範例

```
{
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "failure"
   }
}
```

Example逾時訊息範例

```
{
```

```
   "time": "2021-02-04T20:00:05.123Z",
   "type": "platform.runtimeDone",
   "record": {
      "requestId":"6f7f0961f83442118a7af6fe80b88",
      "status": "timeout"
  }
}
```

Example platform.restoreRuntimeDone 訊息範例 (僅限 **snap-start** 初始化類型)

platform.restoreRuntimeDone 日誌訊息會顯示 `Restore` 階段是否成功。執行階段傳送 `restore/next` 執行階段 API 請求時，Lambda 會產生此訊息。可能的狀態有三種：成功、失敗和逾時。成功的 platform.restoreRuntimeDone 日誌訊息如下列範例所示。

```
{
  "time":"2022-07-17T18:43:45.936Z",
  "type":"platform.restoreRuntimeDone",
  "record":{
      "status":"success"
  }
}
```

針對 Lambda 中的問題進行故障診斷

下列主題提供您在使用 Lambda API、主控台或工具時可能遭遇錯誤和問題的故障診斷建議。如果您發現未列在此處的問題，您可以使用此頁面上的 Feedback (意見回饋) 按鈕來報告。

如需更多故障診段建議和常見支援問題的解答，請瀏覽 AWS 知識中心。

如需有關偵錯和疑難排解 Lambda 應用程式的詳細資訊，請參閱無伺服器園地中的偵錯。

主題

- 對 Lambda 中的組態問題進行疑難排解
- 針對 Lambda 中的部署問題進行疑難排解
- 針對 Lambda 中的調用問題進行疑難排解
- 針對 Lambda 中的執行問題進行疑難排解
- 對 Lambda 中的事件來源映射問題進行疑難排解
- 針對 Lambda 中的聯網問題進行疑難排解

對 Lambda 中的組態問題進行疑難排解

函式組態設定可能會影響 Lambda 函式的整體效能和行為。這些設定可能不會造成實際的函式錯誤，但可能會導致意外的逾時和結果。

下列主題針對您可能遇到的與 Lambda 函式組態設定相關的常見問題，提供疑難排解建議。

主題

- 記憶體組態
- CPU 受限的組態
- 逾時
- 調用之間的記憶體外洩
- 非同步結果傳回至稍後的調用

記憶體組態

您可以將 Lambda 函式設定為使用 128 MB 至 10,240 MB 記憶體。依預設，在主控台中建立的任何函數都會被指派最小數量的記憶體。許多 Lambda 函式在此設定下限下仍具有效能。但是，若要匯入大型程式庫或完成記憶體密集型任務，128 MB 並不足夠。

如果函式的執行速度比預期慢得多，則首先應增大記憶體設定。對於記憶體受限的函數，這將解決瓶頸問題，並可能改善函數的效能。

CPU 受限的組態

對於運算密集型操作，如果函式效能慢於預期，這可能是由於函式屬於 CPU 密集型。在這種情況下，函數的運算容量跟不上工作的速度。

雖然 Lambda 不允許您直接修改 CPU 組態，但 CPU 可以透過記憶體設定間接控制。當您配置更多記憶體時，Lambda 服務會按比例配置更多虛擬 CPU。當記憶體設定為 1.8 GB 時，Lambda 函式會配置一個完整的 vCPU；超過此數值後，函式將可使用多個 vCPU 核心。檔記憶設定為 10,240 MB 時，將有 6 個 vCPU 可供使用。換言之，您可以透過增加記憶體配置來改善效能，即使函式並未用完所有記憶體。

逾時

Lambda 函式的逾時可設定為 1 至 900 秒 (15 分鐘)。依預設，Lambda 主控台會將此設定為 3 秒。逾時值是一種安全閥，可確保函式不會無限期執行。達到逾時值後，Lambda 會停止函式調用。

將逾時值設定為接近函數的平均持續時間會提高函數意外逾時的風險。函數的持續時間會有所不同，這取決於資料傳輸和處理量，以及函數與之互動的任何服務的延遲。逾時的常見原因包括：

- 從 S3 儲存貯體或其他資料存放區下載資料時，下載量較大或下載時間較平均時間長。
- 函數會向另一個服務提出請求，這需要更長的時間才能回應。
- 提供給函數的參數要求函數具有更高的運算複雜性，這會導致調用時間更長。

測試應用程式時，請確保測試可準確反映資料的大小和數量以及真實的參數值。重要的是，使用工作負載合理預期上限的資料集。

此外，只要可行，就在工作負載中實作上限限制。在本範例中，應用程式可以使用每種檔案類型的大小上限。然後，您可以測試應用程式在一系列預期檔案大小 (最高可達並包括上限) 下的效能。

調用之間的記憶體外洩

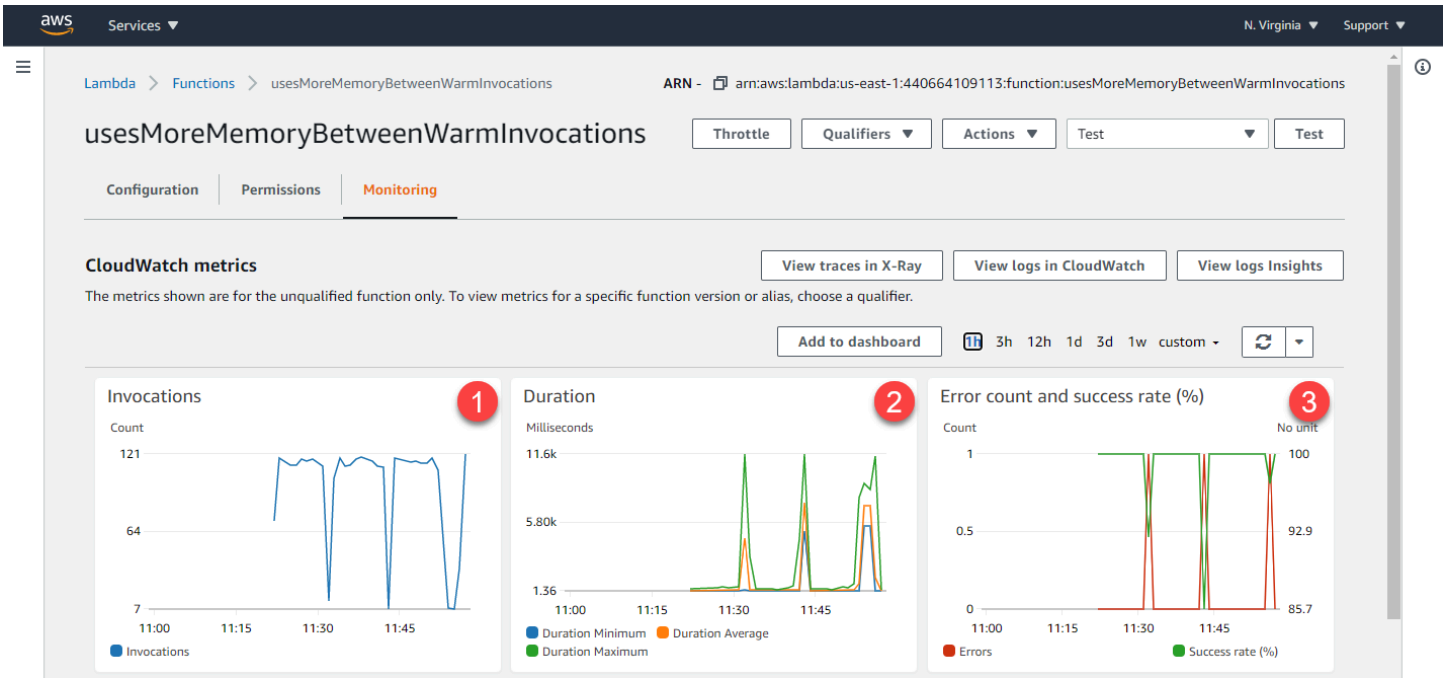
Lambda 調用 INIT 階段儲存的全域變數和物件會在暖調用之間保留其狀態。它們只會在執行環境第一次執行 (也稱為「冷啟動」) 時完全重設。處理常式結束時，儲存在處理常式中的變數都會銷毀。最佳實務是使用 INIT 階段設定資料庫連線、載入程式庫、建立快取，以及載入不可變的資產。

在同一執行環境中跨多個調用使用第三方程式庫時，需要查閱其在無伺服器運算環境中的使用說明文件。有些資料庫連線和日誌記錄程式庫可能會儲存中繼調用結果和其他資料。這會導致這些程式庫的記憶體使用量隨著後續暖調用而增加。在此情況下，您可能會發現 Lambda 函式耗盡記憶體，即使自訂程式碼正確處理變數也是如此。

此問題會影響暖執行環境中發生的調用。例如，下列程式碼會在調用之間造成記憶體外洩。Lambda 函數透過增加全域陣列的大小，在每次調用時使用額外的記憶體：

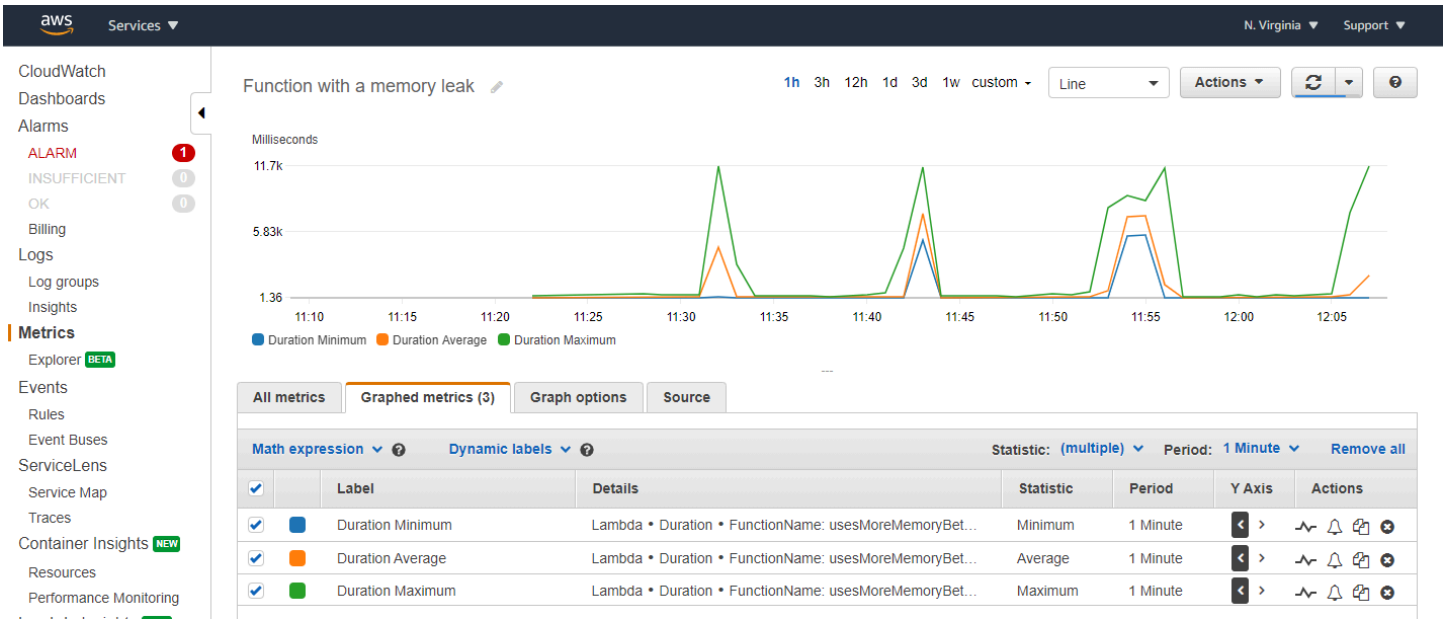
```
let a = []

exports.handler = async (event) => {
    a.push(Array(100000).fill(1))
}
```

使用 128 MB 的記憶體設定，在調用此函式 1000 次之後，Lambda 函式的監控索引標籤會顯示發生記憶體外洩時典型的調用、持續時間和錯誤計數變更：



1. 調用：由於調用需要更長的時間才能完成，因此穩定的交易速率會定期中斷。狀態穩定時，記憶體外洩不會耗用函數配置的所有記憶體。隨著效能降低，作業系統正在對本機儲存體進行分頁，以容納函數所需的不斷增長的記憶體，這會導致完成的交易數減少。
2. 持續時間：在函式耗盡記憶體之前，會穩定地以兩位數毫秒速率完成調用。分頁發生時，持續時間會增加一個數量級。
3. 錯誤計數：因為記憶體外洩超過配置的記憶體，最終函式因運算超過逾時而出錯，或是執行環境終止函式。

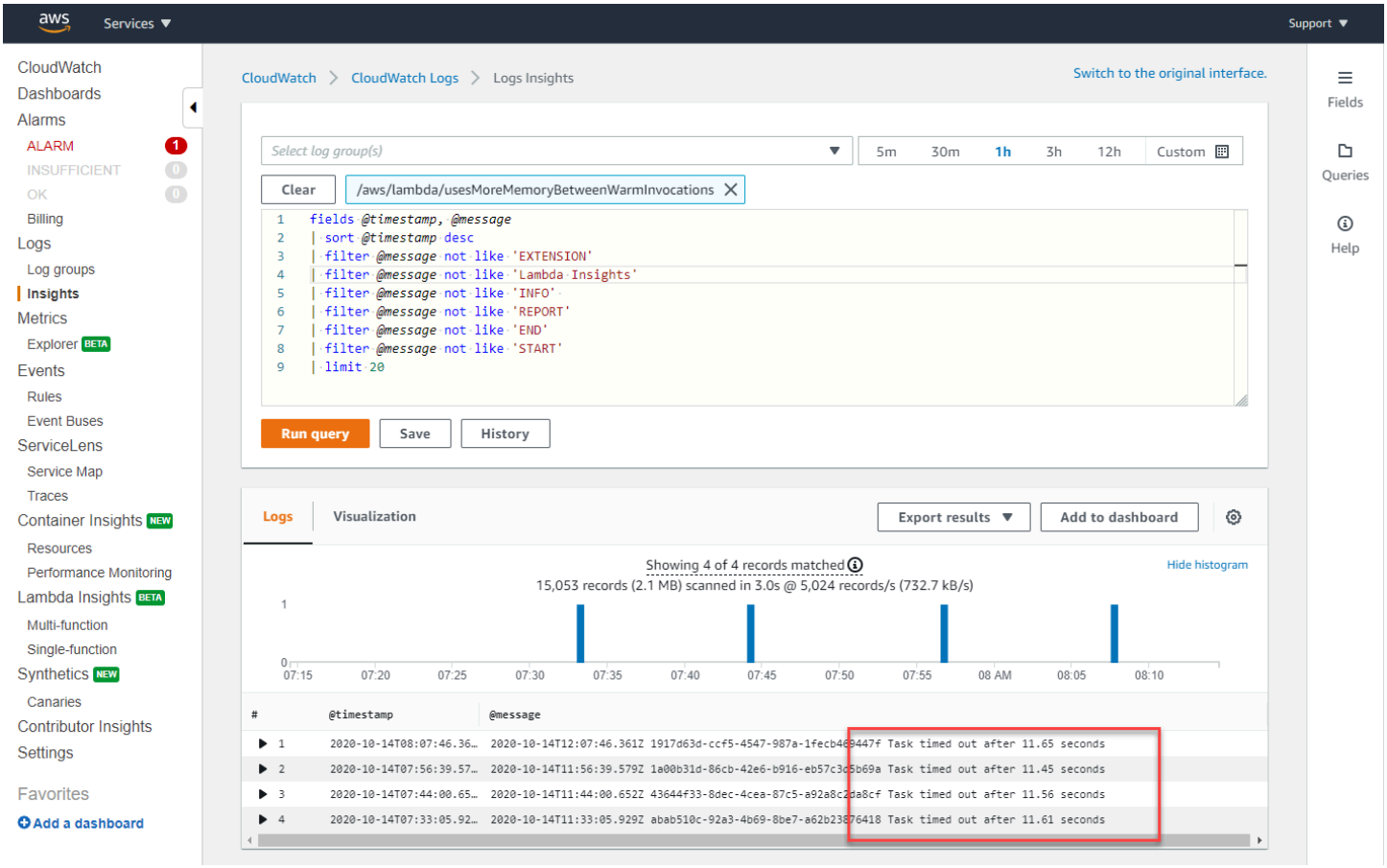
出錯後，Lambda 會重新啟動執行環境，這解釋了為什麼在全部三個圖形中會回到原始狀態。展開 CloudWatch 的持續時間指標可提供最短、最長和平均持續時間統計資料的更多詳細資訊：



若要尋找 1000 個調用中產生的錯誤，可以使用 CloudWatch Insights 查詢語言。下列查詢會排除資訊日誌，僅報告錯誤：

```
fields @timestamp, @message
| sort @timestamp desc
| filter @message not like 'EXTENSION'
| filter @message not like 'Lambda Insights'
| filter @message not like 'INFO'
| filter @message not like 'REPORT'
| filter @message not like 'END'
| filter @message not like 'START'
```

在針對此函數的日誌群組執行時，這表示週期性錯誤是逾時造成的：

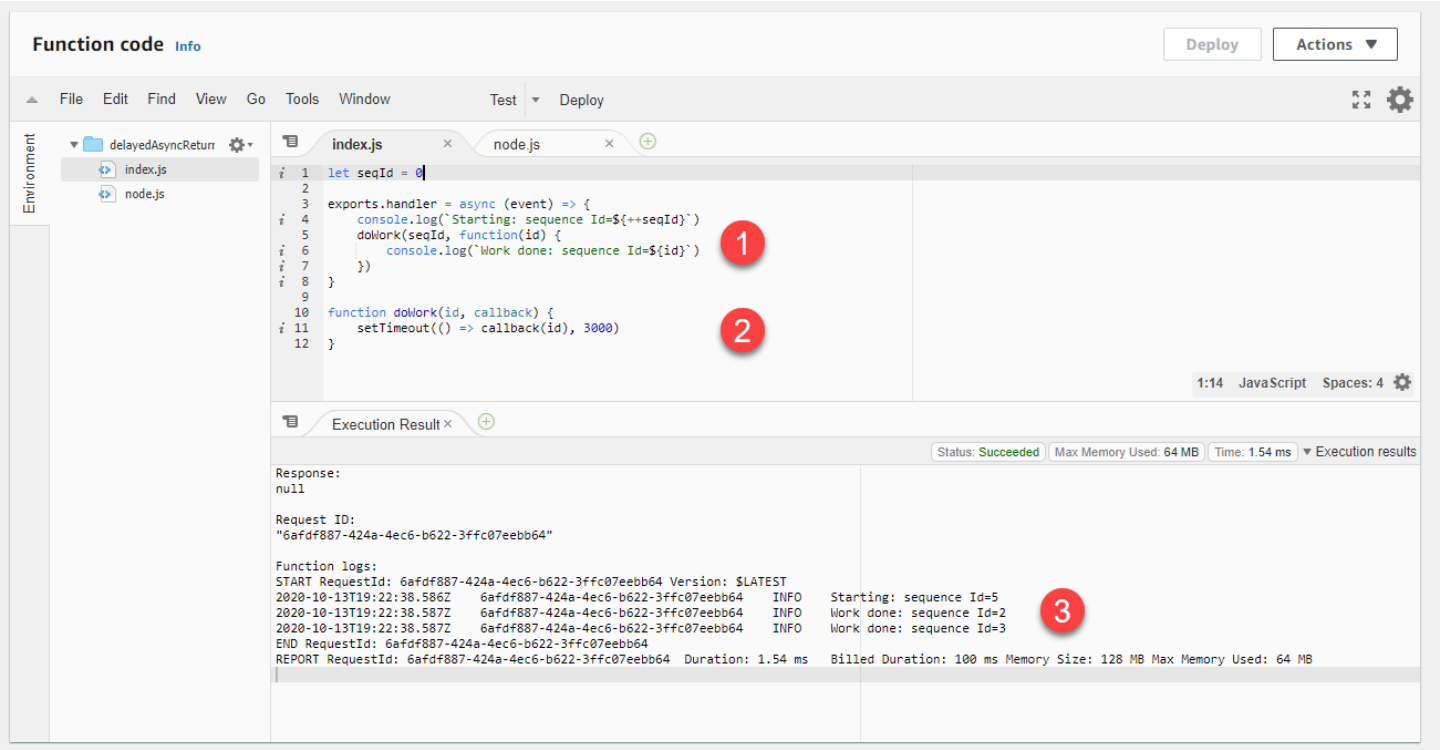


非同步結果傳回至稍後的調用

對於使用非同步模式的函數程式碼，一次調用的回呼結果可能在未來的調用中傳回。此範例使用了 Node.js，但相同的邏輯可套用至使用非同步模式的其他執行時期。函數使用 JavaScript 中的傳統回呼語法。它會呼叫一個非同步函數，該函數具有增量計數器，可追蹤調用次數：

```
let seqId = 0

exports.handler = async (event, context) => {
    console.log(`Starting: sequence Id=${++seqId}`)
    doWork(seqId, function(id) {
        console.log(`Work done: sequence Id=${id}`)
    })
}

function doWork(id, callback) {
    setTimeout(() => callback(id), 3000)
}
```

連續調用多次時，回呼的結果會出現在後續調用中：



1. 程式碼呼叫 doWork 函式，將回呼函式最為最後一個參數傳入。
2. doWork 函式需要一段時間才能完成，然後才會調用回呼。
3. 函式的日誌記錄指出調用在 doWork 函式完成執行之前結束。此外，在開始反覆運算後，正在處理先前反覆運算的回呼，如日誌中所示。

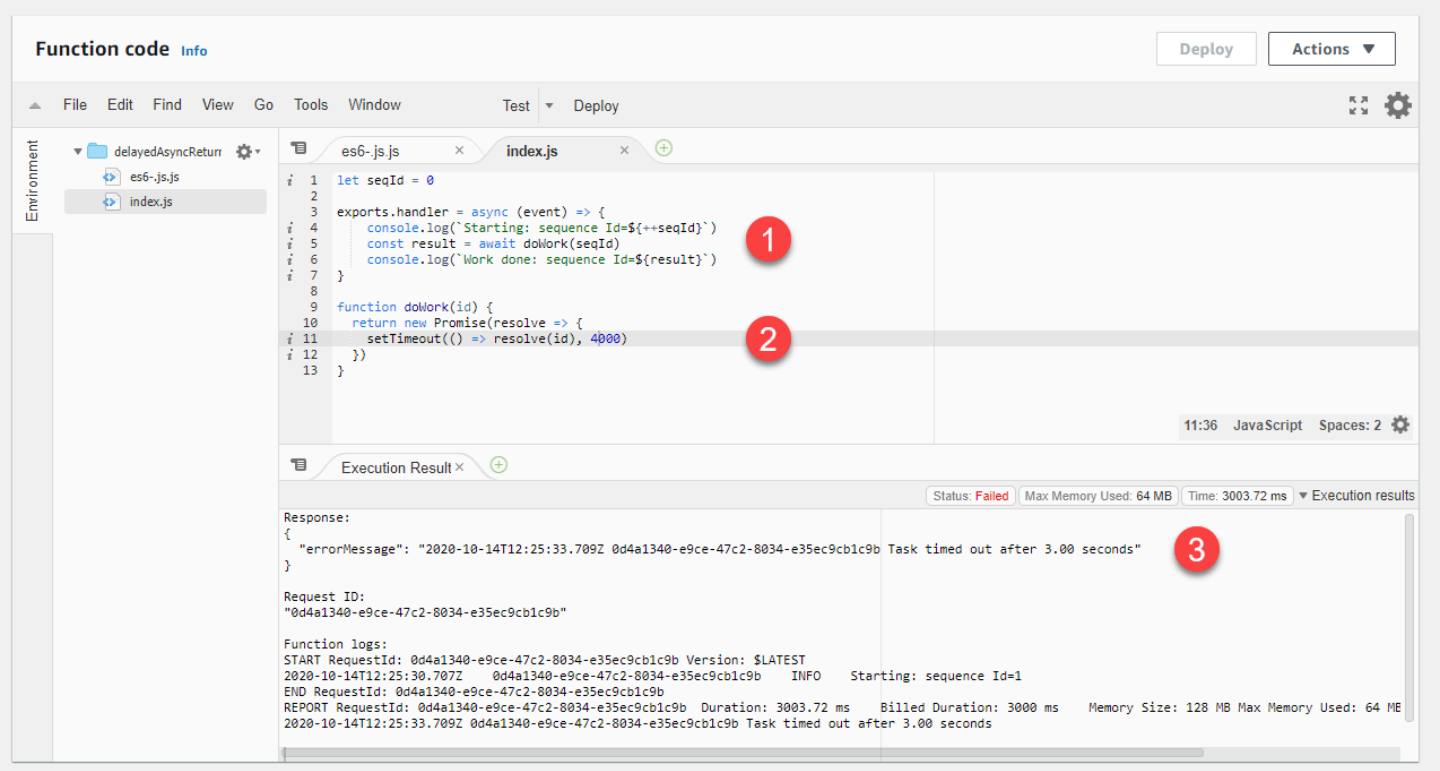
在 JavaScript 中，非同步回呼會使用事件迴圈處理。其他執行時期使用不同機制處理並行。當函式的執行環境結束時，Lambda 會凍結環境，直至下一次調用。恢復後，JavaScript 會繼續處理事件迴圈，此時迴圈包含先前調用的非同步回呼。如果沒有此內容，函數可能會無故執行程式碼並傳回任意資料。事實上，它確實是執行時期並行與執行環境交互的成品。

這會造成前一次調用的隱私資料可能出現在後續的調用中。有兩種方法可以防止或偵測到此行為。首先，JavaScript 提供非同步和等待關鍵字來簡化非同步開發，並強制程式碼執行等待非同步呼叫完成。可以使用此方法重寫上述函數，如下所示：

```
let seqId = 0
exports.handler = async (event) => {
    console.log(`Starting: sequence Id=${++seqId}`)
    const result = await doWork(seqId)
    console.log(`Work done: sequence Id=${result}`)
```

```
}

function doWork(id) {
  return new Promise(resolve => {
    setTimeout(() => resolve(id), 4000)
  })
}
```

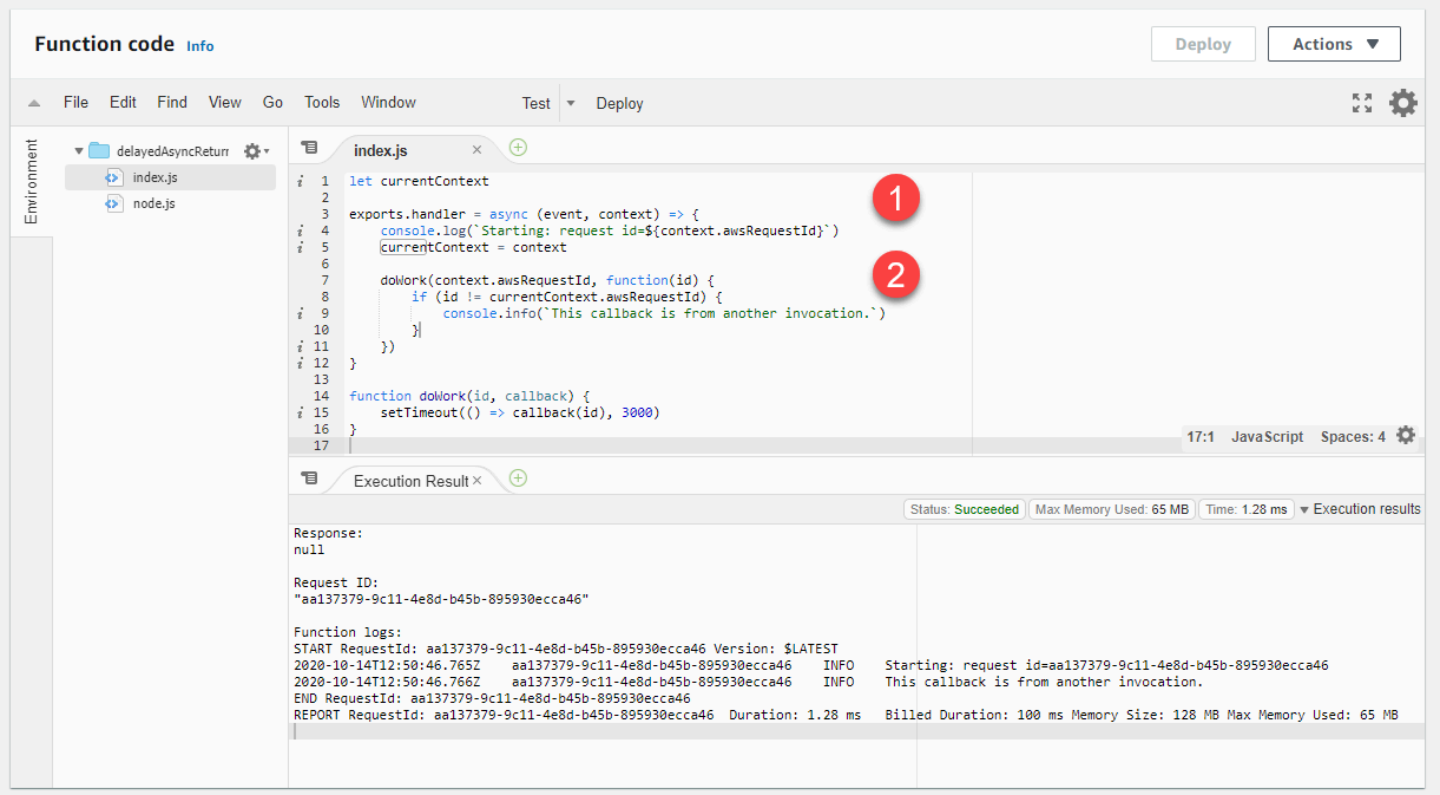
使用此語法可防止處理常式在非同步函數完成之前結束。在這種情況下，如果回呼花費的時間超過 Lambda 函數的逾時時間，函數會擲回錯誤，而不是在稍後的調用中傳回回呼結果：



1. 程式碼會使用處理常式中的等待關鍵字呼叫非同步 doWork 函式。
2. doWork 函式需要一段時間才能完成，然後才會解析承諾。
3. 函式會逾時，因為 doWork 耗費的時間超過逾時限制允許的時間，而且回呼結果不會在稍後的調用中傳回。

一般而言，您應該確定程式碼中的任何背景程序或回呼在程式碼結束前完成。如果這在您的使用案例中無法做到，可以使用識別符來確定回呼屬於目前的調用。為此，可以使用內容物件提供的 awsRequestId。透過將此值傳遞至非同步回呼，您可以將傳遞的值與目前的值進行比較，以偵測回呼是否來自另一個調用：

```
let currentContext

exports.handler = async (event, context) => {
    console.log(`Starting: request id=$\{context.awsRequestId}`)
    currentContext = context

    doWork(context.awsRequestId, function(id) {
        if (id != currentContext.awsRequestId) {
            console.info(`This callback is from another invocation.`)
        }
    })

}

function doWork(id, callback) {
    setTimeout(() => callback(id), 3000)

}
```



1. Lambda 函數處理常式採用內容參數，提供對唯一調用請求 ID 的存取權。

2. awsRequestId 會傳遞至 doWork 函式。在回呼中，ID 會與目前調用的 awsRequestId 進行比較。如果這些值不同，程式碼可以採取相應的動作。

針對 Lambda 中的部署問題進行疑難排解

當您更新函數時，Lambda 會透過啟動包含更新程式碼或設定的函數新執行個體，來部署變更。部署錯誤會導致您無法使用新版本，而造成這類錯誤的可能原因包含您部署套件、程式碼、許可或工具的問題。

當您使用 Lambda API 或 等用戶端直接部署更新至函數時 AWS CLI，您可以直接在輸出中看到來自 Lambda 的錯誤。如果您使用 AWS CloudFormation AWS CodeDeploy、 或 等服務 AWS CodePipeline，請在該服務的日誌或事件串流中尋找來自 Lambda 的回應。

下列主題提供您在使用 Lambda API、主控台或工具時可能遭遇錯誤和問題的故障診斷建議。如果您發現未列在此處的問題，您可以使用此頁面上的 Feedback (意見回饋) 按鈕來報告。

如需更多故障診段建議和常見支援問題的解答，請瀏覽 AWS 知識中心。

如需有關偵錯和疑難排解 Lambda 應用程式的詳細資訊，請參閱無伺服器園地中的偵錯。

主題

- 一般：許可遭拒/無法載入此類檔案
- 一般：呼叫 UpdateFunctionCode 時發生錯誤
- Amazon S3：錯誤代碼 PermanentRedirect。
- 一般：找不到、無法載入、無法匯入、找不到類別、沒有此類檔案或目錄
- 一般：未定義的方法處理常式
- 一般：超過 Lambda 程式碼儲存限制
- Lambda：分層轉換失敗
- Lambda：InvalidParameterValueException 或 RequestEntityTooLargeException
- Lambda：InvalidParameterValueException
- Lambda：並行和記憶體配額
- Lambda：佈建並行的別名組態無效

一般：許可遭拒/無法載入此類檔案

錯誤：EACCES：拒絕許可，開啟 '/var/task/index.js'

錯誤：無法載入這類檔案 – 函數

錯誤：[Errno 13] 拒絕許可：'/var/task/function.py'

Lambda 執行時間需有許可才能讀取部署套裝服務中的檔案。在 Linux 許可八進位標記法中，Lambda 需要 644 個許可 (rw-r--r--) 用於非可執行檔，以及 755 個許可 (rwxr-x) 用於目錄和可執行檔。

在 Linux 和 MacOS 中，使用 chmod 命令變更部署套件中檔案和目錄的檔案許可。例如，若要為非可執行檔提供正確的許可，請執行下列命令。

```
chmod 644 <filepath>
```

若要在 Windows 中變更檔案許可，請參閱 Microsoft Windows 文件的 Set, View, Change, or Remove Permissions on an Object。

> **Note**
>
> 如果您未授予 Lambda 存取部署套件中的目錄所需的許可，Lambda 會將這些目錄的許可設定為 755 (rwxr-xr-x)。

一般：呼叫 UpdateFunctionCode 時發生錯誤

錯誤：呼叫 UpdateFunctionCode 操作時發生錯誤 (RequestEntityTooLargeException)

當您將部署套裝服務或 Layer 存檔直接上傳至 Lambda 時，ZIP 檔案的大小限制為 50 MB。若要上傳更大的檔案，請將它存放在 Amazon S3 中並使用 S3Bucket 和 S3Key 參數。

> **Note**
>
> 當您直接使用 AWS CLI、 AWS SDK 或其他方式上傳檔案時，二進位 ZIP 檔案會轉換為 base64，這會將其大小增加約 30%。若要允許此操作，以及請求中其他參數的大小，Lambda 套用的實際請求大小限制會更大。因此，50 MB 的限制是概略值。

Amazon S3：錯誤代碼 PermanentRedirect。

錯誤：GetObject 時發生錯誤。S3 錯誤代碼：PermanentRedirect. S3 錯誤訊息：儲存貯體位於此區域：us-east-2。請使用此區域重試請求

當您從 Amazon S3 儲存貯體上傳函數的部署套件時，儲存貯體必須位於與函數相同的區域。當您在呼叫 UpdateFunctionCode 時指定 Amazon S3 物件，或在 AWS CLI 或 AWS SAM CLI 中使用 套件和部署命令時，可能會發生此問題。為開發應用程式的每個區域建立部署成品儲存貯體。

一般：找不到、無法載入、無法匯入、找不到類別、沒有此類檔案或目錄

錯誤： Cannot find module 'function' (找不到 'function' 模組)

錯誤：無法載入這類檔案 – 函數

錯誤： Unable to import module 'function' (無法匯入 'function' 模組)

錯誤： Class not found: function.Handler (找不到類別：function.Handler)

錯誤： fork/exec /var/task/function: no such file or directory (fork/exec /var/task/function：找不到檔案或目錄)

錯誤： Unable to load type 'Function.Handler' from assembly 'Function'. (無法從 'Function' 組件載入 'Function.Handler' 類型。)

您函數處理器組態中的檔案或類別名稱與您的程式碼不相符。如需詳細資訊，請參閱下一節。

一般：未定義的方法處理常式

錯誤： index.handler is undefined or not exported (index.handler 未定義或尚未匯出)

錯誤： Handler 'handler' missing on module 'function' ('function' 模組上找不到 'handler' 處理器)

錯誤： 未定義 #<LambdaHandler:0x000055b76ccebf98> 的 `handler' 方法

錯誤： No public method named handleRequest with appropriate method signature found on class function.Handler (在 function.Handler 類別上找不到具備適當方法簽章，名為 handleRequest 的公有方法)

錯誤： Unable to find method 'handleRequest' in type 'Function.Handler' from assembly 'Function' (在來自 'Function' 組件的 'Function.Handler' 類型中找不到 'handleRequest' 方法)

您函數處理器組態中的處理器方法名稱與您的程式碼不相符。每個執行時間都會定義處理器的命名慣例，例如 *filename.methodname*。處理器是您函數程式碼中的方法，執行時間會在調用您的函數時執行該方法。

針對某些語言，Lambda 提供了程式庫，其中包含預期處理器方法具備特定名稱的界面。如需每種語言的處理器命名詳細資訊，請參閱下列主題。

- 使用 Node.js 建置 Lambda 函數
- 使用 Python 建置 Lambda 函數
- 使用 Ruby 建置 Lambda 函數
- 使用 Java 建置 Lambda 函數
- 使用 Go 建置 Lambda 函數
- 使用 C# 建置 Lambda 函數
- 使用 PowerShell 建置 Lambda 函數

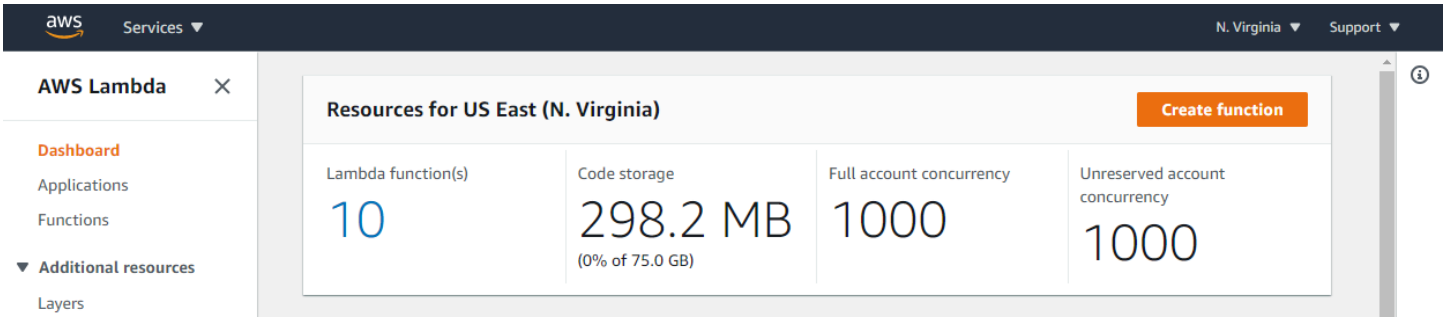
一般：超過 Lambda 程式碼儲存限制

錯誤：超過程式碼儲存限制。

Lambda 會將函式程式碼儲存在帳戶私有的內部 S3 儲存貯體中。每個 AWS 帳戶在每個區域中配置 75 GB 的儲存空間。程式碼儲存包含 Lambda 函數和層使用的總儲存空間。如果您達到配額，在嘗試部署新函數時，會收到 CodeStorageExceededException。

透過清理舊版本的函式、移除未使用的程式碼或使用 Lambda 層，即可管理可用的儲存空間。此外，最佳實務是針對個別工作負載使用個別 AWS 帳戶，以協助管理儲存配額。

您可以在 Lambda 主控台的儀表板子選單下檢視儲存體總用量：



Lambda：分層轉換失敗

錯誤：Lambda 分層轉換失敗。如需有關解決此問題的建議，請參閱《Lambda 使用者指南》中的「Lambda 部署問題疑難排解」頁面。

當您使用分層設定 Lambda 函數，Lambda 會將該分層與函數程式碼合併。如果此程序無法完成，Lambda 便會傳回此錯誤。如果出現此錯誤，請執行下列步驟：

- 從分層中刪除所有未使用的檔案

- 刪除分層中的所有符號連結
- 重新命名任何與函數分層中目錄名稱相同的所有檔案

Lambda：InvalidParameterValueException 或 RequestEntityTooLargeException

錯誤：InvalidParameterValueException：Lambda 因為您提供的環境變數超過 4KB 限制，Lambda 無法設定您的環境變數。測量的字串：{"A1":"uSFeY5cyPiPn7AtnX5BsM...

錯誤： RequestEntityTooLargeException：請求必須小於 5120 位元組的 UpdateFunctionConfiguration 操作

儲存在函數組態中的變數物件大小上限不得超過 4,096 個位元組。這包括金鑰名稱、值、引號、逗號和括號。HTTP 請求主體的大小總計也受到限制。

```
{
    "FunctionName": "my-function",
    "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-function",
    "Runtime": "nodejs24.x",
    "Role": "arn:aws:iam::123456789012:role/lambda-role",
    "Environment": {
        "Variables": {
            "BUCKET": "amzn-s3-demo-bucket",
            "KEY": "file.txt"
        }
    },
    ...
}
```

在此範例中，物件是 39 個字元，並在其存放為字串 `{"BUCKET":"amzn-s3-demo-bucket","KEY":"file.txt"}` (不含空格) 時，最多佔用 39 個位元組。環境變數值中每個標準 ASCII 字元使用一個位元組。每個延伸的 ASCII 字元和 Unicode 字元可以使用 2 個位元組到 4 個位元組。

Lambda：InvalidParameterValueException

錯誤： InvalidParameterValueException：因為您提供的環境變數已包含目前不支援修改的預留金鑰，Lambda 無法設定您的環境變數。

Lambda 保留一些環境變數金鑰以供內部使用。例如，執行時間使用的 AWS_REGION 可決定目前區域，而且不可置換。但是，執行時間使用的其他變數，例如 PATH，可在函數組態中擴充。如需完整清單，請參閱定義執行時間環境變數。

Lambda：並行和記憶體配額

錯誤：指定的函數 ConcurrentExecutions 使帳戶的 UnreservedConcurrentExecution 降低到其最小值以下

錯誤：「MemorySize」值無法滿足限制條件：成員的值必須小於或等於 3008

當您超過帳戶的並行或記憶體配額時，就會發生這些錯誤。新 AWS 帳戶已減少並行和記憶體配額。若要解決與並行相關的錯誤，您可以請求提高配額。您無法請求增加記憶體配額。

- 並行：如果您嘗試使用保留或佈建的並行建立函數，或者您的每個函數並行請求 (PutFunctionConcurrency) 超過帳戶的並行配額，便可能發生錯誤。
- 記憶體：如果分配給函數的記憶體數量超過帳戶的記憶體配額，就會發生錯誤。

Lambda：佈建並行的別名組態無效

錯誤：佈建並行的別名組態無效

當您嘗試更新 Lambda 函式的程式碼或組態，而某個具有佈建並行的別名指向有問題的版本時，就會發生此錯誤。Lambda 會為佈建並行預先初始化執行環境，若這些環境因程式碼錯誤、資源限制或受影響的堆疊和別名而無法正確初始化，部署就會失敗。如果遇到此問題，請執行下列步驟：

1. 復原別名：暫時更新別名，使其指向先前可正常運作的版本。

```
aws lambda update-alias \
 --function-name <function-name> \
 --name <alias-name> \
 --function-version <known-good-version>
```

2. 修正 Lambda 初始化程式碼：確保在處理常式外部執行的初始化程式碼不包含任何未捕捉的例外狀況，並完成用戶端與連線的初始化。
3. 安全地重新部署：部署已修正的程式碼並發布新版本。接著，更新別名，使其指向已修正的版本。如有必要，也可以重新啟用佈建並行。

如果使用 AWS CloudFormation，請更新堆疊定義，`FunctionVersion:!GetAtt version.Version`讓別名指向工作版本：

```
alias:
 Type: AWS::Lambda::Alias
 Properties:
 FunctionName: !Ref function
FunctionVersion: !GetAtt version.Version
 Name: BLUE
 ProvisionedConcurrencyConfig:
 ProvisionedConcurrentExecutions: 1
```

針對 Lambda 中的調用問題進行疑難排解

當您調用 Lambda 函數時，Lambda 會先驗證請求並檢查擴展容量，再將事件傳送到您的函數，或是事件佇列 (針對非同步調用)。導致調用錯誤的可能原因包含請求參數、事件結構、函數設定、使用者許可、資源許可或限制等問題。

如果您直接調用函數，您會在 Lambda 的回應中看到調用錯誤。如果您透過事件來源映射或是其他服務以非同步方式調用您的函數，您可能會在日誌、無效字母佇列或是失敗事件目的地上找到錯誤。錯誤處理選項和重試行為會因您調用函數的方式，以及錯誤的類型而有所不同。

如需 `Invoke` 操作可以傳回的錯誤類型清單，請參閱調用。

主題

- Lambda：函數在初始化階段逾時 (Sandbox.Timedout)
- IAM：lambda:InvokeFunction 未獲授權
- Lambda：找不到有效的引導程序 (Runtime.InvalidEntrypoint)
- Lambda：無法執行操作 ResourceConflictException
- Lambda：函數卡在待定狀態
- Lambda：一個函數正在使用所有並行
- 一般：無法使用其他帳戶或服務調用函數
- 一般：函數調用正在循環
- Lambda：具有佈建並行的別名路由
- Lambda：使用佈建並行的冷啟動

- Lambda：新版本的冷啟動
- Lambda：未預期的 Node.js 在執行時間結束 (Runtime.NodejsExit)
- EFS：函數無法掛載 EFS 檔案系統
- EFS：函數無法連線到 EFS 檔案系統
- EFS：因為逾時，函數無法掛載 EFS 檔案系統
- Lambda：Lambda 偵測到耗時太久的 IO 程序
- 容器：CodeArtifactUserException 錯誤
- 容器：InvalidEntrypoint 錯誤

Lambda：函數在初始化階段逾時 (Sandbox.Timedout)

錯誤：任務在 3.00 秒後逾時

當初始化階段出現逾時，Lambda 會在下一個調用請求到達時重新執行`Init`階段，以再次初始化執行環境。(我們將其稱為隱藏的初始化。) 不過，如果函數設定了較短的逾時持續時間 (通常大約 3 秒)，則隱藏的初始化可能無法在配置的逾時前完成，導致`Init`階段再次逾時。或者，隱藏的初始化雖然完成，但留給調用階段的時間不夠，導致後者無法完成，從而造成`Invoke`階段逾時。

若要減少逾時錯誤，請使用以下其中一種或幾種策略：

- 延長函數逾時期間：延長逾時，讓`Init`和`Invoke`階段有時間成功完成。
- 增加分配給函數的記憶體：更多的記憶體意味著更多的 CPU 運算能力，這可以加快`Init`和`Invoke`階段的速度。
- 最佳化函數初始化程式碼：縮短初始化所需的時間，以確保`Init`和`Invoke`階段可以在設定的逾時內完成。

IAM：lambda:InvokeFunction 未獲授權

錯誤：使用者 arn:aws:iam::123456789012:user/developer 無權對資源 my-function 執行 lambda:InvokeFunction

您的使用者或您擔任的角色必須有調用函數的許可。此要求也適用於 Lambda 函數及其他調用函數的運算資源。將 AWS 受管政策 AWSLambdaRole 新增至您的使用者，或新增允許對目標函數`lambda:InvokeFunction`執行動作的自訂政策。

> **Note**
>
> IAM 動作的名稱 (`lambda:InvokeFunction`) 指的是 `Invoke` Lambda API 操作。

如需詳細資訊，請參閱 在 中管理許可 AWS Lambda。

Lambda：找不到有效的引導程序 (Runtime.InvalidEntrypoint)

錯誤：找不到有效的引導程序：[/var/task/bootstrap /opt/bootstrap]

當部署套件的根層級不包含名為 `bootstrap` 的可執行檔時，通常會發生此錯誤。例如，如果您使用 .zip 檔案部署 `provided.al2023` 函數，則 `bootstrap` 檔案必須位於 .zip 檔案的根層級，而不在目錄中。

Lambda：無法執行操作 ResourceConflictException

錯誤：ResourceConflictException：此時無法執行操作。函數量前處於下列狀態：擱置中

當您在建立時將函數連接到 Virtual Private Cloud (VPC) 時，函數會在 Lambda 建立彈性網路界面的同時，進入 `Pending` 狀態。在此期間，您無法調用或修改函數。如果您在建立後將函數連接到 VPC，則可以在更新擱置時調用該函數，但無法修改其程式碼或組態。

如需詳細資訊，請參閱 Lambda 函數狀態。

Lambda：函數卡在待定狀態

錯誤： 函數停留在 *`Pending`* 狀態幾分鐘的時間。

如果函數卡在 `Pending` 狀態的時間超過六分鐘，請呼叫下列其中一個 API 操作來解除封鎖：

- UpdateFunctionCode
- UpdateFunctionConfiguration
- PublishVersion

Lambda 會取消待處理的操作並將該函數放入 `Failed` 狀態。您接著可以嘗試另一個更新。

Lambda：一個函數正在使用所有並行

問題：單一函數正在使用所有可用的並行，造成其他函數遭到調節。

若要將 區域中 AWS 您帳戶的可用並行分割為集 AWS 區，請使用預留並行。預留並行可確保函數一律會擴展至其受到指派的並行，且函數擴展的並行也不會超過其受到指派的並行。

一般：無法使用其他帳戶或服務調用函數

問題：您可以直接調用函數，但當其他服務或帳戶調用該函數時，它不會執行。

您在函數以資源為基礎的政策中，授予其他服務和帳戶調用函數的許可。如果調用者屬於另一個帳戶，則該使用者也必須具備函數的調用許可。

一般：函數調用正在循環

問題： 在迴圈中連續調用函數。

這通常發生在函數在觸發它的相同 AWS 服務中管理資源時。例如，可以建立函數，將物件存放在所設定的 Amazon Simple Storage Service (Amazon S3) 儲存貯體中，該儲存貯體具有再次調用函數的通知。若要阻止函數執行，可將函數的可用並行處理降為零，這會限制所有將來的調用。然後，識別造成遞迴調用的程式碼路徑或組態錯誤。Lambda 會自動偵測並停止某些 AWS 服務和 SDKs遞迴迴圈。如需詳細資訊，請參閱the section called “遞迴迴圈偵測”。

Lambda：具有佈建並行的別名路由

問題： 在別名路由期間佈建並行溢出調用。

Lambda 使用簡單的概率模型來在兩個函數版本之間分配流量。在流量較低時，您可能會看到每個版本已設定流量百分比與實際流量百分比之間，存在很大差異。如果您的函數使用佈建並行，透過在別名路由作用期間設定較高數目的已佈建並行執行個體，則可以避免溢出調用。

Lambda：使用佈建並行的冷啟動

問題：啟用佈建的並行後，您會看到冷啟動。

當函數上的並行執行次數少於或等於已設定的佈建並行層級，則不應該發生任何冷啟動。若要協助您確認佈建的並行是否能正常運作，請執行下列動作：

- 請在函數版本或別名上檢查佈建的並行是否啟用。

> **Note**
>
> 函數的未發佈版本 ($LATEST) 無法設定佈建並行。

- 確保您的觸發條件調用的是正確的函數版本或別名。例如，如果您使用的是 Amazon API Gateway，請檢查 API Gateway 調用的函數版本或別名具有佈建的並行，而不是 $LATEST。若要確認正在使用佈建的並行，您可以檢查 ProvisionedConcurrencyInvocations Amazon CloudWatch 指標。非零值表示函數正在初始化執行環境上處理調用。
- 判斷您的函數並行是否超過已佈建並行的設定層級，方法是檢查 ProvisionedConcurrencySpilloverInvocations CloudWatch 指標。非零值表示所有已佈建的並行處於使用中的狀態，而某些調用會在冷啟動時發生。
- 檢查調用頻率 (每秒請求數)。具有佈建並行的函數，每個佈建並行的請求速率上限為每秒 10 個。例如，設定為具備 100 個佈建並行的函數每秒可以處理 1,000 個請求。如果調用速率超過每秒 1,000 個請求，就可能會發生冷啟動。

Lambda：新版本的冷啟動

問題：在部署新版的函數時，您會看到冷啟動。

當您更新函數別名時，Lambda 會根據別名上設定的權重，自動將佈建的並行移至新版本。

錯誤：KMSDisabledException：因為使用的 KMS 金鑰已停用，Lambda 無法解密環境變數。請檢查函數的 KMS 金鑰設定。

如果您的 AWS Key Management Service (AWS KMS) 金鑰已停用，或如果允許 Lambda 使用該金鑰的授予遭到撤銷，則可能會發生此錯誤。如果授權遺失，請將函數設定為使用不同的金鑰。然後重新分配自訂金鑰以重新建立授權。

Lambda：未預期的 Node.js 在執行時間結束 (Runtime.NodejsExit)

問題：Lambda 執行期用戶端偵測到非預期的 Node.js 結束碼。

當您的函數在解決所有 Promises 之前結束時，就會發生此錯誤，例如由於程式碼錯誤。當 Node.js 偵測到防止 Promises 解決的死結時，也可能發生這種情況。此錯誤只會影響非同步樣式處理常式，不會影響回呼樣式處理常式。

受影響的執行時間：Node.js 18 及更新版本。

若要解決此問題：

1. 檢查您的函數程式碼是否有非同步處理常式中未解決的承諾。
2. 在函數完成之前，確保所有承諾都已正確解決（已解決或拒絕）。

3. 檢閱您的程式碼是否有非同步操作中的潛在競爭條件。

如需 Node.js 結束代碼和程序終止的詳細資訊，請參閱 Node.js 文件。

EFS：函數無法掛載 EFS 檔案系統

錯誤：EFSMountFailureException：函數無法使用存取點 arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd 掛載 EFS 檔案系統。

對檔案系統的掛載要求已遭拒。檢查函數的許可，並確認其檔案系統和存取點是否存在，且可供使用。

EFS：函數無法連線到 EFS 檔案系統

錯誤：EFSMountConnectivityException：函數無法使用存取點 arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd 連線至 Amazon EFS 檔案系統。請檢查您的網路組態，並再試一次。

函數無法使用 NFS 通訊協定 (TCP 連接埠 2049) 建立函數檔案系統的連線。為 VPC 的子網路檢查安全群組與路由組態。

如果您在更新函數的 VPC 組態設定後遇到這些錯誤，請嘗試卸載並重新掛載檔案系統。

EFS：因為逾時，函數無法掛載 EFS 檔案系統

錯誤：EFSMountTimeoutException：由於掛載逾時，函數無法使用存取點 {arn:aws:elasticfilesystem:us-east-2:123456789012:access-point/fsap-015cxmplb72b405fd} 掛載 EFS 檔案系統。

函數可以連線至函數的檔案系統，但掛載操作逾時。稍後再試一次，並考慮限制函數的並行數量，以減少檔案系統的負載。

Lambda：Lambda 偵測到耗時太久的 IO 程序

EFSIOException：函數執行個體已停止，因為 Lambda 偵測到耗時太久的 IO 處理序。

先前的調用逾時，而且 Lambda 無法終止函數處理常式。當附加的檔案系統用完高載額度，且基準輸送量不足時，可能會發生此問題。若要增加輸送量，您可以增加檔案系統的大小，或使用佈建的輸送量。

容器：CodeArtifactUserException 錯誤

錯誤：CodeArtifactUserPendingException 錯誤訊息

CodeArtifact 正在等待最佳化。當 Lambda 完成最佳化時，函式會轉換為作用中狀態。HTTP 回應代碼 409。

錯誤：CodeArtifactUserDeletedException 錯誤訊息

CodeArtifact 已排程待刪除。HTTP 回應代碼 409。

錯誤：CodeArtifactUserFailedException 錯誤訊息

Lambda 無法最佳化程式碼。您需要更正程式碼並重新上傳。HTTP 回應代碼 409。

容器：InvalidEntrypoint 錯誤

錯誤：Runtime.ExitError 或 "errorType": "Runtime.InvalidEntrypoint"

確認容器映像的 ENTRYPOINT 包含作為位置的絕對路徑。同時請確認映像不含作為 ENTRYPOINT 的符號連結。

錯誤：您正在使用 CloudFormation 範本，且您的容器 ENTRYPOINT 正在以 null 或空值覆寫。

檢閱 CloudFormation 範本中的 ImageConfig 資源。如果您在範本中宣告 `ImageConfig` 資源，則必須為所有三個屬性提供非空值。

針對 Lambda 中的執行問題進行疑難排解

當 Lambda 執行時間執行您的函數程式碼時，可能會在已經處理事件一段時間的函數執行個體上處理事件，或是需要初始化新的執行個體。函數初始化期間、您的處理器程式碼處理事件時，或是您的函數傳回回應時 (或是無法傳回時)，都可能會發生錯誤。

造成函數執行錯誤的可能原因包含了您程式碼、函數組態、下游資源或是許可的問題。如果您直接叫用您的函數，您會在 Lambda 的回應中看到函數錯誤。如果您透過事件來源映射或是其他服務以非同步方式叫用您的函數，您可能會在日誌、無效字母佇列或是失敗的目的地上找到錯誤。錯誤處理選項和重試行為會因您叫用函數的方式，以及錯誤的類型而有所不同。

當您的函數程式碼或 Lambda 執行時間傳回錯誤時，Lambda 回應中的狀態碼將會是 200 OK。名為 `X-Amz-Function-Error` 的標頭指示回應中存在錯誤。400 和 500 系列的狀態碼為叫用錯誤預留。

主題

- Lambda：使用 Visual Studio Code 進行遠端偵錯
- Lambda：執行時間太長
- Lambda：非預期的事件承載
- Lambda：出乎意料的大型承載大小
- Lambda：JSON 編碼與解碼錯誤
- Lambda：日誌或追蹤沒有出現
- Lambda：並非所有函數的日誌都會顯示
- Lambda：該函數在執行完成之前傳回
- Lambda：執行非預期的函式版本或別名
- Lambda：偵測無限迴圈
- 一般：下游服務無法使用
- AWS SDK：版本和更新
- Python：程式庫的載入不正確
- Java：從 Java 11 更新至 Java 17 後，函數需要更長的時間來處理事件
- Kafka：處理和重試組態問題時發生錯誤

Lambda：使用 Visual Studio Code 進行遠端偵錯

問題：在實際 AWS 環境中難以疑難排解複雜的 Lambda 函數行為

Lambda 透過 AWS Toolkit for Visual Studio Code提供遠端偵錯功能。如需設定與一般說明，請參閱使用 Visual Studio Code 遠端偵錯 Lambda 函式。

如需疑難排解、進階使用案例和區域可用性的詳細說明，請參閱 AWS Toolkit for Visual Studio Code《 使用者指南》中的遠端偵錯 Lambda 函數。

Lambda：執行時間太長

問題：函數執行時間過長。

如果您的程式碼在 Lambda 中執行的時間比在本機機器上長，可能是因為函數可用的記憶體或處理能力受到限制。設定函數使用額外記憶體以同時增加記憶體和 CPU。

Lambda：非預期的事件承載

問題：函式因格式錯誤的 JSON 或資料驗證不完整而出現錯誤。

所有的 Lambda 函數都會在處理常式的第一個參數中接收事件承載。事件承載是 JSON 結構，可能包含陣列和巢狀元素。

當上游服務未使用可靠的流程來檢查其 JSON 結構時，可能會出現 JSON 格式錯誤的問題。當服務串接文字字串或嵌入尚未經過消毒的使用者輸入時，就會發生這種情況。JSON 也經常被序列化，以便在服務之間傳遞。一律做為 JSON 的生產者和消費者剖析 JSON 結構，以確定結構有效。

同樣地，不檢查事件承載中的值範圍可能導致錯誤。此範例顯示計算預扣稅的函數：

```
exports.handler = async (event) => {
    let pct = event.taxPct
    let salary = event.salary

    // Calculate % of paycheck for taxes
    return (salary * pct)
}
```

此函數使用事件承載中的薪資和稅率執行計算。但是，程式碼無法檢查屬性是否存在。它也無法檢查資料類型，或確定界限，例如確定稅率在 0 和 1 之間。因此，這些邊界以外的值將產生無意義的結果。類型不正確或缺少屬性會導致執行時期錯誤。

建立測試以確定您的函數能夠處理較大的承載。Lambda 事件承載的大小上限為 1 MB。視內容而定，較大的承載可能表示傳遞到函數的項目較多，或內嵌在 JSON 屬性中的二進位資料較多。在這兩種情況下，都可能導致 Lambda 函數的處理量增加。

較大的承載也可能導致逾時。例如，Lambda 函數每 100 毫秒處理一筆記錄，逾時 3 秒。承載中 0-29 個項目的處理成功。但是，一旦承載中的項目數超過 30，函數就會逾時並擲回錯誤。為避免這種情況，請確定設定超時，以處理預期最大項目數的額外處理時間。

Lambda：出乎意料的大型承載大小

問題：函式因大型承載而逾時或導致錯誤。

較大的承載可能導致逾時和錯誤。建議建立測試，確保函式能夠處理預期的最大承載量，並確保函式逾時設定正確。

此外，某些事件承載可能包含指向其他資源的指標。例如，具有 128 MB 記憶體的 Lambda 函式，可以對以物件形式儲存於 S3 的 JPG 檔案進行映像處理。對於較小的映像檔案，函數可如預期執行。不

過，當提供較大的 JPG 檔案做為輸入時，Lambda 函數會因為記憶體耗盡而擲出錯誤。為避免這種情況，測試案例應包含預期資料大小上限的範例。程式碼也應驗證承載大小。

Lambda：JSON 編碼與解碼錯誤

問題：剖析 JSON 輸入時發生 NoSuchKey 例外狀況。

進行檢查，確保正確處理 JSON 屬性。例如，對於 S3 產生的事件，`s3.object.key` 屬性包含經過 URL 編碼的物件索引鍵名稱。許多函數會將此屬性當成文字來處理，以載入引用的 S3 物件：

Example

```
const originalText = await s3.getObject({
  Bucket: event.Records[0].s3.bucket.name,
  Key: event.Records[0].s3.object.key
}).promise()
```

此程式碼可與金鑰名稱 `james.jpg` 搭配使用，但在名稱為 `james beswick.jpg` 時擲出 NoSuchKey 錯誤。由於 URL 編碼會在金鑰名稱中轉換空格和其他字元，因此必須先確定函數解碼金鑰，再使用此資料：

Example

```
const originalText = await s3.getObject({
  Bucket: event.Records[0].s3.bucket.name,
  Key: decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, " "))
}).promise()
```

Lambda：日誌或追蹤沒有出現

問題： 日誌未出現在 CloudWatch Logs 中。

問題：追蹤不會出現在 中 AWS X-Ray。

您的函數需要許可才能呼叫 CloudWatch Logs 和 X-Ray。更新其執行角色以授予許可。新增下列受管政策來啟用日誌和追蹤。

- AWSLambdaBasicExecutionRole
- AWSXRayDaemonWriteAccess

將許可新增至您的函數時，亦請對其程式碼或組態進行細微更新。若您函數的執行中執行個體具有已過期的憑證，上述動作會強制停止並取代這些執行個體。

> **Note**
>
> 在函數調用後，日誌可能需要 5 到 10 分鐘才會顯示。

Lambda：並非所有函數的日誌都會顯示

問題：即使我有正確的許可，但 CloudWatch Logs 缺少函數日誌

如果您的 AWS 帳戶 達到其 CloudWatch Logs 配額限制，CloudWatch 會調節函數記錄。發生這種情況時，您的函數的一些日誌輸出可能不會在 CloudWatch Logs 中顯示。

如果您的函數輸出日誌的速度過快，Lambda 無法處理它們，這也有可能導致日誌輸出不在 CloudWatch Logs 中顯示。當 Lambda 無法以函數產生日誌的速度將它們傳送到 CloudWatch 時，它會捨棄日誌以防止函數的執行速度變慢。當日誌輸送量單一日誌串流超過 2 MB/s 時，預期會持續觀察到日誌遭捨棄。

如果函數設定為使用 JSON 格式日誌，Lambda 會嘗試在捨棄日誌時將`logsDropped` 事件傳送至 CloudWatch Logs。不過，當 CloudWatch 對函數的日誌記錄進行限流時，此事件可能無法到達 CloudWatch Logs，因此當 Lambda 捨棄記錄時，您不一定會看到記錄。

若要檢查您的 AWS 帳戶 是否已達到其 CloudWatch Logs 配額限制，請執行下列動作：

1. 開啟 Service Quotas 主控台。
2. 在導覽窗格中，選擇 AWS services (AWS 服務)。
3. 對於 AWS services 清單，搜尋並選取 Amazon CloudWatch Logs。
4. 在 Service Quotas 清單中，選擇 `CreateLogGroup throttle limit in transactions per second`、`CreateLogStream throttle limit in transactions per second` 和 `PutLogEvents throttle limit in transactions per second` 配額，以檢視您的使用率。

您也可以設定 CloudWatch 警示，以便在帳戶利用率超過您為這些配額指定的限制時提醒您。請參閱建立以靜態閾值為基礎的 CloudWatch 警示，以了解詳細資訊。

如果 CloudWatch Logs 的預設配額限制不足以滿足您的使用案例，您可以請求提高配額。

Lambda：該函數在執行完成之前傳回

問題：(Node.js) 函數在程式碼完成執行前傳回

包括 AWS SDK 在內的許多程式庫會以非同步方式運作。當您進行網路呼叫或是執行需要等待回應的其他操作時，程式庫會傳回稱為 promise 的物件，在背景追蹤操作的進度。

若要等待 promise 解析為回應，請使用 `await` 關鍵字。這會阻止您的處理器在包含回應的 promise 解析為物件前執行。如果您不需要在程式碼中使用回應中的資料，您可以直接將 promise 傳回執行時間。

有些程式庫不會傳回 promise，但是可以包裝在傳回 promise 的程式碼中。如需詳細資訊，請參閱在 Node.js 中定義 Lambda 函數處理常式。

Lambda：執行非預期的函式版本或別名

問題：未調用的函式版本或別名

當您在主控台或使用 發佈新的 Lambda 函數時 AWS SAM，最新的程式碼版本會以 表示$LATEST。依預設，未指定版本或別名的調用會自動指向函式程式碼的 $LATEST 版本。

如果使用特定的函式版本或別名，這些是除了 $LATEST 之外不可變的已發布函式版本。對這些函式進行疑難排解時，請先確定呼叫者已調用預期的版本或別名。您可以透過檢查函式日誌來完成此操作。所調用的函式版本會始終顯示在 START 日誌行中：

```
2020-11-13T09:53:21.179-05:00          START RequestId: a37400cb-f3bc-441b-8592-dcb9fa552995 Version: 1

2020-11-13T08:14:27.834-05:00          START RequestId: f4943cb9-58a1-472a-af81-5801b6f0eb8d Version: $LATEST
```

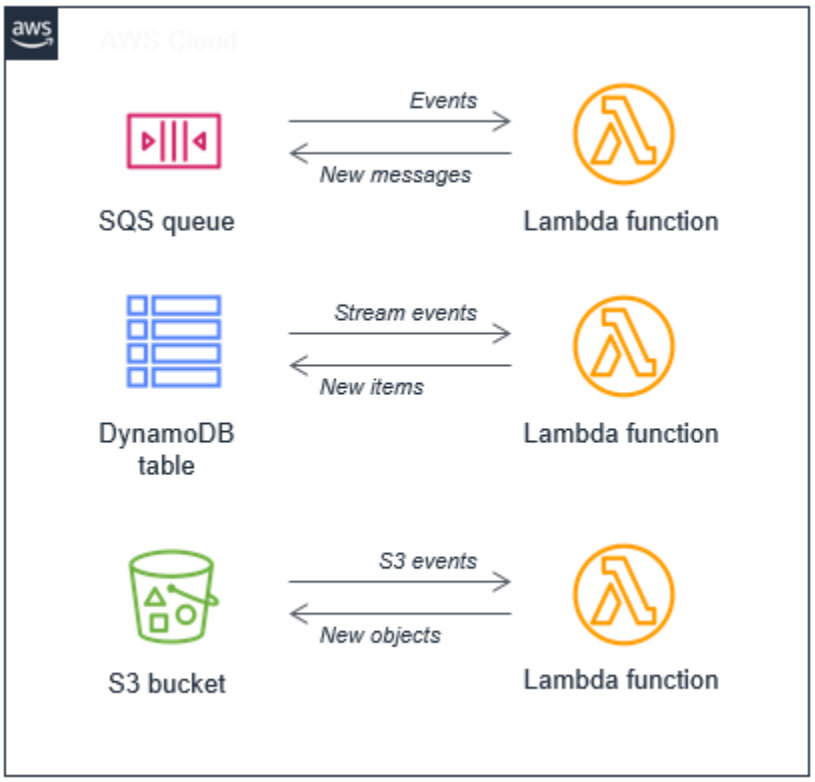
Lambda：偵測無限迴圈

問題：因 Lambda 函式而導致無限迴圈模式

Lambda 函數中有兩種無限迴圈類型。第一種在函式本身內部，由一個永不結束的迴圈造成。調用只會在函式逾時之時結束。您可以透過監控逾時來識別這些情況，然後修正迴圈行為。

第二種迴圈類型介於 Lambda 函數和其他 AWS 資源之間。當來自 S3 儲存貯體等資源的事件調用 Lambda 函數，然後該函數與相同來源的資源交互以觸發另一個事件時，就會發生這些情況。這會再次

調用函式，進而與同一個 S3 儲存貯體建立另一次互動，依此類推。這些類型的迴圈可能由許多不同的 AWS 事件來源造成，包括 Amazon SQS 佇列和 DynamoDB 資料表。您可以使用遞迴迴圈偵測功能來識別這些模式。



透過確保 Lambda 函式寫入與取用端資源不同的資源，即可避免這些迴圈。如果必須將資料發布回取用端資源，請確保新資料不會觸發相同的事件。或者，使用事件篩選功能。例如，以下是針對 S3 和 DynamoDB 資源無限迴圈的兩種建議解決方案：

- 如果寫回同一個 S3 儲存貯體，請使用與事件觸發程序不同的字首或尾碼。
- 如果將項目寫入同一個 DynamoDB 資料表，請包含取用端 Lambda 函式可以篩選的屬性。如果 Lambda 找到該屬性，則不會導致另一次調用。

一般：下游服務無法使用

問題：Lambda 函式所依賴的下游服務無法使用

對於呼叫第三方端點或其他下游資源的 Lambda 函式，確保其具備處理服務錯誤與逾時的能力。這些下游資源可能具有可變回應時間，或可能因服務中斷而無法使用。根據實作，如果未在函式程式碼內處理服務的錯誤回應，這些下游錯誤可能顯示為 Lambda 逾時或例外狀況。

每當函式依賴下游服務時 (例如 API 呼叫)，請實作適當的錯誤處理和重試邏輯。對於關鍵服務，Lambda 函式應將指標或日誌發布到 CloudWatch。例如，若第三方付款 API 無法使用，Lambda 函式可以記錄此資訊。然後，您可以設定 CloudWatch 警示來傳送與這些錯誤相關的通知。

由於 Lambda 可以快速擴展，非無伺服器下游服務可能難以應對流量激增的狀況。有三種常見的應對方法：

- 快取：如果第三方服務傳回的值不會經常變更，建議快取這些結果。您可以將這些值儲存於函式的全域變數中或其他服務中。例如，來自 Amazon RDS 執行個體的產品清單查詢結果可以在函數內儲存一段時間，以防備援查詢。
- 佇列：儲存或更新資料時，在 Lambda 函式與資源之間新增 Amazon SQS 佇列。在下游服務處理訊息時，佇列會持久保留資料。
- 代理：通常使用長效連線 (例如 Amazon RDS 執行個體)，使用代理層來匯集和重複使用這些連線。對於關聯式資料庫，Amazon RDS Proxy 是一種服務，旨在協助改善以 Lambda 為基礎的應用程式的可擴展性和彈性。

AWS SDK：版本和更新

問題：執行時間中包含的 AWS SDK 不是最新版本

問題：執行時間更新中包含的 AWS SDK 會自動更新

解譯語言的執行時間包括 AWS SDK 的版本。Lambda 會定期更新這些執行時期，使用最新的 SDK 版本。若要查詢執行時期內含的 SDK 版本，請參閱下列章節：

- 執行時期內含 SDK 版本 (Node.js)
- 執行時期內含 SDK 版本 (Python)
- 執行時期內含 SDK 版本 (Ruby)

若要使用較新版本的 AWS SDK，或將函數鎖定到特定版本，您可以將程式庫與函數程式碼綁定，或建立 Lambda 層。如需建立包含相依性部署套件的詳細資訊，請參閱下列主題：

Node.js

使用 .zip 封存檔部署 Node.js Lambda 函數

Python

使用 .zip 封存檔部署 Python Lambda 函數

Ruby

使用 .zip 封存檔部署 Ruby Lambda 函數

Java

使用 .zip 或 JAR 封存檔部署 Java Lambda 函數

Go

使用 .zip 封存檔部署 Go Lambda 函數

C#

使用 .zip 封存檔建置和部署 C# Lambda 函數

PowerShell

使用 .zip 封存檔部署 PowerShell Lambda 函數

Python：程式庫的載入不正確

問題：(Python) 有些程式庫無法從部署套件正確載入

使用以 C 或 C++ 撰寫延伸模組的程式庫必須在處理器架構與 Lambda 相同的環境中編譯 (Amazon Linux)。如需詳細資訊，請參閱使用 .zip 封存檔部署 Python Lambda 函數。

Java：從 Java 11 更新至 Java 17 後，函數需要更長的時間來處理事件

問題：(Java) 從 Java 11 更新至 Java 17 後，函數需要更長的時間來處理事件

使用 JAVA_TOOL_OPTIONS 參數調校編譯器。Java 17 和更新版本的 Lambda 執行時期變更了預設的編譯器選項。此變更可改善短期函數的冷啟動時間，但先前的行為更適合長時間執行的運算密集函數。將 JAVA_TOOL_OPTIONS 設定為 -XX:-TieredCompilation 以還原至 Java 11 的行為。如需 JAVA_TOOL_OPTIONS 參數的詳細資訊，請參閱 the section called “了解 JAVA_TOOL_OPTIONS 環境變數”。

Kafka：處理和重試組態問題時發生錯誤

問題：Kafka 事件來源映射無法設定重試設定或失敗時的目的地

Kafka 重試組態和故障時目的地僅適用於啟用佈建模式的事件來源映射。在嘗試設定重試組態ProvisionedPollerConfig之前，請確定您已在 MinimumPollers中設定 。

常見的組態錯誤：

- 使用 bisect 批次無限次重試 – 當 `MaximumRetryAttempts` 設定為 -1 （無限） `BisectBatchOnFunctionError`時，您無法啟用 。設定有限重試限制或停用平分批次。
- 相同主題遞迴 – Kafka 失敗時目的地主題不能與您的任何來源主題相同。為您的無效字母主題選擇不同的主題名稱。
- 無效的 Kafka 目的地格式 – 將 Kafka 主題指定為失敗時的目的地時，請使用 `kafka://<topic-name>` 格式。
- kafka：WriteData 許可問題 – 確保您的執行角色具有目的地主題的`kafka-cluster:WriteData`許可。主題不存在逾時例外狀況，或寫入 API 限流問題可能需要提高帳戶限制。

對 Lambda 中的事件來源映射問題進行疑難排解

Lambda 中與事件來源映射相關的問題可能較為複雜，因為涉及在多個服務中進行除錯。此外，事件來源行為可能會因所使用的具體事件來源而有所不同。本節列出了涉及事件來源映射的常見問題，也提供了如何識別並進行疑難排解的指導。

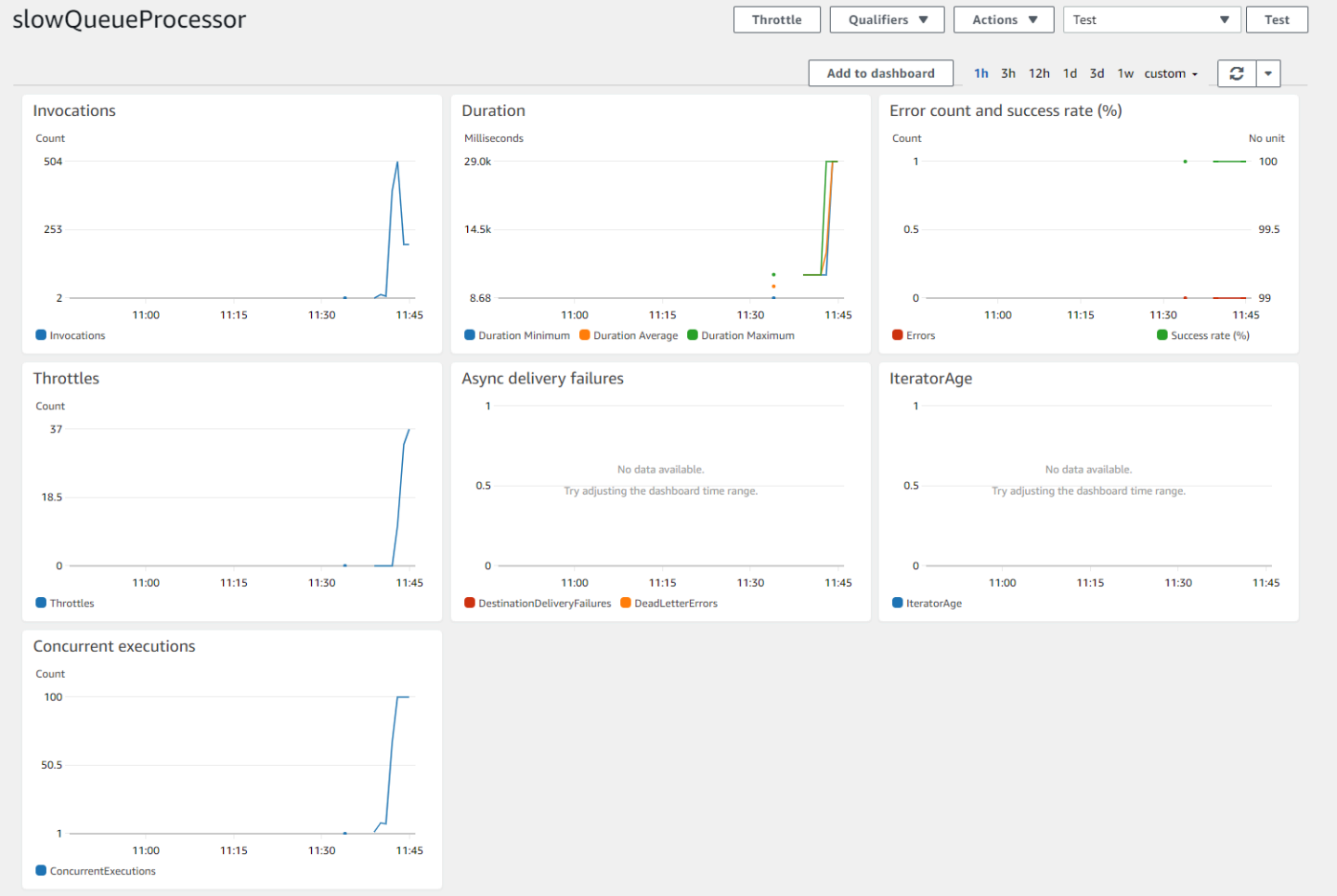
> **Note**
>
> 本節使用 Amazon SQS 事件來源作為說明，但其原則同樣適用於其他為 Lambda 函式佇列訊息的事件來源映射。

識別和管理限流

在 Lambda 中，當達到函式或帳戶的並行限制時，就會發生限流。建議參考下列範例，其中有從 Amazon SQS 佇列讀取訊息的 Lambda 函式。該 Lambda 函式模擬 30 秒的調用時間，且批次大小為 1。這表示函式每 30 秒僅能處理 1 則訊息：

```
const doWork = (ms) => new Promise(resolve => setTimeout(resolve, ms))

exports.handler = async (event) => {
    await doWork(30000)

}
```

由於調用時間過長，訊息進入佇列的速度會超過處理速度。如果帳戶的未預留並行為 100，Lambda 會向上擴展至 100 個並行執行，隨後便會發生限流。您可以在函數的 CloudWatch 指標中看到此模式：



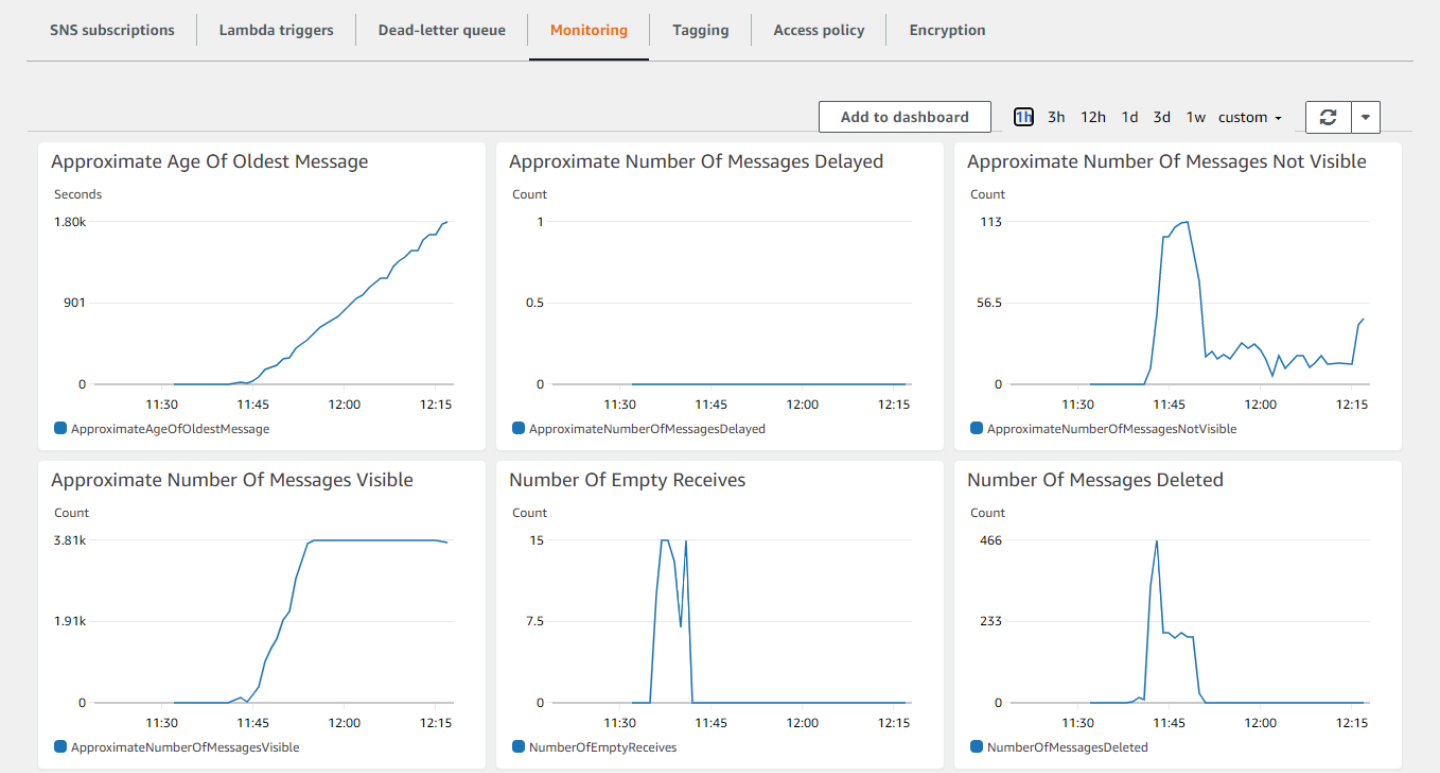
函式的 CloudWatch 指標顯示沒有錯誤，但並行執行圖表顯示已達到 100 的並行上限。因此，限流圖表顯示限流已生效。

您可以透過 CloudWatch 警示來偵測限流狀況，在函式的限流指標大於 0 時即觸發警示。在您確認限流問題後，有以下幾種解決方案可供選擇：

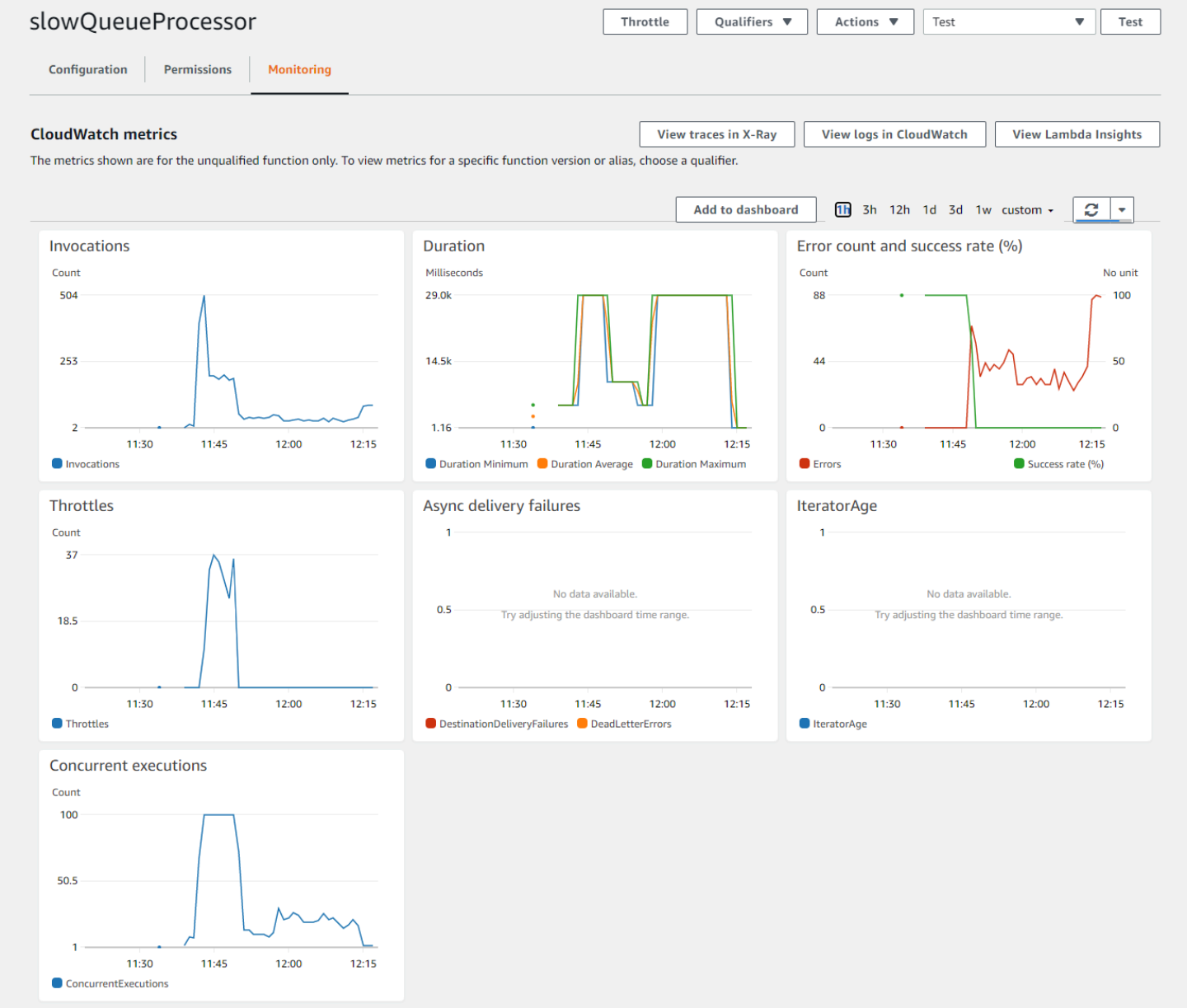
- 請求此區域的 AWS Support 團隊增加並行。
- 識別函數中的效能問題，以提高處理速度，從而改善輸送量。
- 增加函式的批次大小，以便每次調用能處理更多訊息。

處理函數中的錯誤

如果處理函式擲出錯誤，Lambda 會將訊息傳回 SQS 佇列。Lambda 可防止函式擴展，避免出現大規模錯誤。CloudWatch 中的下列 SQS 指標表示佇列處理出現問題：

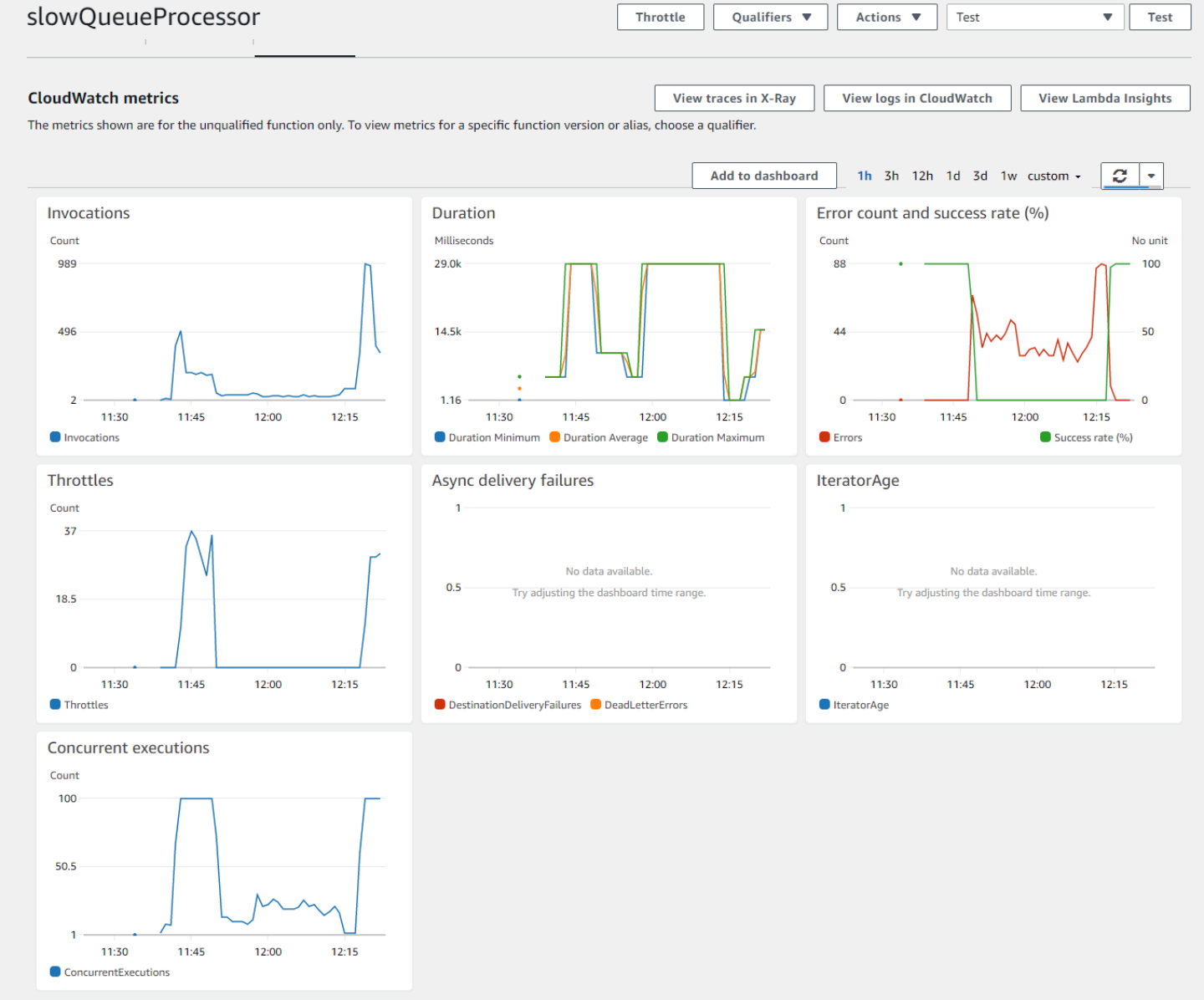


特別是，最舊訊息的存留期和可見訊息的數量都在增加，但沒有訊息被刪除。佇列會繼續成長，但訊息未處理。處理 Lambda 函數的 CloudWatch 指標也表示存在問題：



錯誤計數指標不為零且不斷增加，同時並行執行已減少且限流已停止。這表示 Lambda 已因錯誤而停止向上擴展函式。函式的 CloudWatch 日誌提供錯誤類型的詳細資訊。

您可以透過先識別導致錯誤的函數，然後尋找並消除錯誤來解決此問題。修正錯誤並部署新版函式程式碼後，CloudWatch 指標會顯示處理復原：



在這裡，錯誤計數指標會降至零，成功率指標會恢復至 100%。Lambda 會再次開始向上擴展函式，如並行執行圖表中所示。

識別和處理背壓

如果事件生產者為 SQS 佇列產生訊息的速度持續快於 Lambda 函式的處理速度，就會產生背壓。在這種情況下，SQS 監控會顯示最舊訊息的存留期以線性方式增長，同時顯示可見訊息的大致數量。可以使用 CloudWatch 警示來偵測佇列中的背壓。

消除背壓的步驟取決於您的工作負載。如果主要目標是提高 Lambda 函式的處理能力與輸送量，有以下選項可用選擇：

- 請求特定區域的 AWS Support 團隊增加並行。
- 增加函式的批次大小，以便每次調用能處理更多訊息。

針對 Lambda 中的聯網問題進行疑難排解

根據預設，Lambda 會在擁有 AWS 服務及網際網路連線能力的內部 virtual private cloud (VPC) 中執行您的函數。若要存取本機網路資源，您可以設定您的函數，使其連接到您帳戶中的 VPC。當您使用此功能時，您可以使用 Amazon Virtual Private Cloud (Amazon VPC) 資源管理函數的網際網路存取能力及網路連線能力。

網路連線錯誤可能是由於 VPC 路由組態、安全群組規則、 AWS Identity and Access Management (IAM) 角色許可或網路位址轉譯 (NAT) 的問題，或是 IP 地址或網路介面等資源的可用性所造成。視問題而定，如果請求無法到達其目標，您可能會看到特定錯誤或逾時。

主題

- VPC：函數會失去網際網路存取或逾時
- VPC：TCP 或 UDP 連線間歇性失敗
- VPC：函數需要存取 而不 AWS 服務 使用網際網路
- VPC：已達到彈性網絡介面限制
- EC2：具有「lambda」類型的彈性網路介面
- DNS：遇到 UNKNOWNHOSTEXCEPTION，無法連線至主機

VPC：函數會失去網際網路存取或逾時

問題：Lambda 函數在連線到 VPC 之後失去網際網路存取能力。

錯誤：Error: connect ETIMEDOUT 176.32.98.189:443 (錯誤：連線 ETIMEDOUT 176.32.98.189:443)

錯誤：Error: Task timed out after 10.00 seconds (錯誤：任務在 10.00 秒後逾時)

錯誤： ReadTimeoutError：讀取逾時。(讀取逾時 = 15)

當您將函數連線到 VPC 時，所有傳出請求都會通過 VPC。若要連線到網際網路，請設定您的 VPC 從函數的子網路將傳出流量傳送到公有子網路中的 NAT 閘道。如需詳細資訊和範例 VPC 組態，請參閱 the section called “Lambda 函數的網際網路存取”。

若部分 TCP 連線逾時，且子網路使用網路存取控制清單 (NACL)，請參閱 the section called “VPC：TCP 或 UDP 連線間歇性失敗”。否則，此問題很可能是封包分散所致。Lambda 函數無法處理傳入的分段 TCP 請求，因為 Lambda 不支援 TCP 或 ICMP 的 IP 分段。

VPC：TCP 或 UDP 連線間歇性失敗

> **Note**
>
> 此問題僅在子網路使用網路存取控制清單 (ACL) 時才會發生。Lambda 連線至子網路並不需要網路 ACL。

問題：Lambda 間歇性斷開與您已設定網路存取控制清單 (ACL) 的 VPC 子網路連線。

對於啟用 VPC 的 Lambda 函數， 會在客戶帳戶中 AWS 建立超平面 ENIs，並使用暫時性連接埠102465535將 Lambda 連線至客戶的 VPC。若在目標子網路中使用網路 ACL，則必須允許 TCP 與 UDP 使用 1024 至 65535 的連接埠範圍。若不允許此完整連接埠範圍，可能會導致連線間歇性失敗。

VPC：函數需要存取 而不 AWS 服務 使用網際網路

問題：您的 Lambda 函數需要在不使用網際網路 AWS 服務 的情況下存取 。

若要 AWS 服務 從沒有網際網路存取的私有子網路將函數連接至 ，請使用 VPC 端點。

VPC：已達到彈性網絡介面限制

錯誤：ENILimitReachedException: The elastic network interface limit was reached for the function's VPC. (ENILimitReachedException：已到達函數 VPC 彈性網路界面的限制。)

當您將 Lambda 函數連線到 VPC 時，Lambda 會為每個連接到函數的子網路和安全群組組合建立彈性網絡介面。預設服務配額為每個 VPC 250 個網路介面。若要請求提升配額，您可以使用 Service Quotas 主控台。

EC2：具有「lambda」類型的彈性網路介面

錯誤代碼：Client.OperationNotPermitted

錯誤訊息：無法針對此類型的介面修改安全群組

如果您嘗試修改由 Lambda 所管理的彈性網絡介面 (ENI)，將會收到此錯誤訊息。`ModifyNetworkInterfaceAttribute` 不包含在 Lambda 為更新彈性網路介面上的操作所建立的 Lambda API 中。

DNS：遇到 UNKNOWNHOSTEXCEPTION，無法連線至主機

錯誤訊息：UNKNOWNHOSTEXCEPTION

Lambda 函數最多支援 20 個並行 TCP 連線的 DNS 解析。您的函數可能是達到了該限制。最常見的 DNS 請求乃透過 UDP 完成。如果您的函數只進行 UDP DNS 連線，則問題不太可能是這個。此錯誤通常是由於配置錯誤或基礎設施降級而出現的，因此在深入檢查 DNS 流量之前，請確認 DNS 基礎設施已正確設定且運作狀態良好，而且 Lambda 函數參考的是 DNS 中指定的主機。

如果您判斷問題與 TCP 連線上限相關，請注意，您並無法請求提高此限制。如果 Lambda 函數因為 DNS 承載高而退回到 TCP DNS，請確認您的解決方案使用的程式庫支援 EDNS。如需有關 EDNS 的詳細資訊，請參閱 RFC 6891 標準。如果 DNS 承載持續超過 EDNS 大小上限，您的解決方案可能仍會達到 TCP DNS 限制。

Lambda 範例應用程式

本指南的 GitHub 儲存庫包含示範各種語言和 AWS 服務使用方式的範例應用程式。每個範例應用程式都包含可用於輕鬆部署和清理的指令碼和支援資源。

Node.js

以 Node.js 編寫的範例 Lambda 應用程式

- blank-nodejs - 一個 Node.js 函數，它示範如何使用記錄、環境變數、AWS X-Ray 追蹤、層、單元測試以及 AWS 開發套件。
- nodejs-apig - 具有公有 API 端點的函數，它會處理來自 API Gateway 的事件並傳回 HTTP 回應。

Python

以 Python 編寫的範例 Lambda 應用程式

- blank-python - 一種 Python 函數，它示範如何使用記錄、環境變數、AWS X-Ray 追蹤、層、單元測試和 AWS 開發套件。

Ruby

以 Ruby 編寫的範例 Lambda 應用程式

- blank-ruby - 一種 Ruby 函數，它示範如何使用記錄、環境變數、AWS X-Ray 追蹤、層、單元測試和 AWS 開發套件。
- AWS Lambda 的 Ruby 程式碼範例 - 以 Ruby 編寫的程式碼範例，示範如何與 AWS Lambda 進行互動。

Java

以 Java 編寫的範例 Lambda 應用程式

- example-java – Java 函式，示範如何使用 Lambda 處理訂單。此函式示範如何定義並反序列化自訂輸入事件物件、使用 AWS SDK 以及輸出日誌。
- java-basic - 具有單元測試和變數日誌組態的最小 Java 函數集合。

- java-events - Java 函數集合，其中包含如何處理來自各種服務 (例如 Amazon API Gateway、Amazon SQS 和 Amazon Kinesis) 事件的骨架程式碼。這些函數使用最新版 aws-lambda-java-events 程式庫 (3.0.0 及更新版)。這些範例不需要 AWS 開發套件做為相依項目。
- s3-java - 一種 Java 函數，它處理來自 Amazon S3 的通知事件，並使用 Java Class Library (JCL) 以從上傳的映像檔案建立縮圖。
- layer-java – 此 Java 函式示範如何使用 Lambda 層，將相依項與核心函式程式碼分離封裝。

在 Lambda 上執行熱門 Java 框架

- spring-cloud-function-samples – 來自 Spring 的範例會示範如何使用 Spring Cloud Function 框架來建立 AWS Lambda 函數。
- Serverless Spring Boot Application 示範 – 此範例展示如何在使用 (或不使用) SnapStart 的受管 Java 執行期中設定 Spring Boot 應用程式，或使用自訂執行期做為 GraalVM 原生映像檔。
- Serverless Micronaut Application 示範 – 此範例展示如何在使用 (或不使用) SnapStart 的受管 Java 執行期中使用 Micronaut，或使用自訂執行期做為 GraalVM 原生映像檔。請參閱《Micronaut/Lambda 指南》以進一步瞭解。
- Serverless Quarkus Application 示範 – 此範例展示如何在使用 (或不使用) SnapStart 的受管 Java 執行期中使用 Quarkus，或使用自訂執行期做為 GraalVM 原生映像檔。請參閱《Quarkus/Lambda 指南》和《Quarkus/SnapStart 指南》以進一步瞭解。

Go

Lambda 為 Go 執行時間提供下列範例應用程式：

以 Go 編寫的範例 Lambda 應用程式

- go-al2：傳回公有 IP 地址的「hello world」函數。此應用程式使用 `provided.al2` 自訂執行期。
- blank-go - 一種 Go 函數，它示範如何使用 Lambda 的 Go 程式庫、記錄、環境變數和 AWS 開發套件。此應用程式使用 `go1.x` 執行期。

C#

以 C# 編寫的範例 Lambda 應用程式

- blank-csharp - 一種 C# 函數，它示範如何使用 Lambda 的 .NET 程式庫、記錄、環境變數、AWS X-Ray 追蹤、單元測試和 AWS 開發套件。

- blank-csharp-with-layer – C# 函數，使用 .NET CLI 建立封裝函數相依項的層。
- ec2-spot – 在 Amazon EC2 中管理 Spot 執行個體請求的函數。

PowerShell

Lambda 提供下列適用於 PowerShell 的範例應用程式：

- blank-powershell - 一種 PowerShell 函數，它示範如何使用日誌記錄、環境變數和 AWS 開發套件。

若要部署範例應用程式，請依照其 README 檔案中的指示。

搭配 AWS SDK 使用 Lambda

AWS 軟體開發套件 (SDKs) 適用於許多熱門的程式設計語言。每個 SDK 都提供 API、程式碼範例和說明文件，讓開發人員能夠更輕鬆地以偏好的語言建置應用程式。

SDK 文件	代碼範例
適用於 C++ 的 AWS SDK	適用於 C++ 的 AWS SDK 程式碼範例
AWS CLI	AWS CLI 程式碼範例
適用於 Go 的 AWS SDK	適用於 Go 的 AWS SDK 程式碼範例
適用於 Java 的 AWS SDK	適用於 Java 的 AWS SDK 程式碼範例
適用於 JavaScript 的 AWS SDK	適用於 JavaScript 的 AWS SDK 程式碼範例
適用於 Kotlin 的 AWS SDK	適用於 Kotlin 的 AWS SDK 程式碼範例
適用於 .NET 的 AWS SDK	適用於 .NET 的 AWS SDK 程式碼範例
適用於 PHP 的 AWS SDK	適用於 PHP 的 AWS SDK 程式碼範例
AWS Tools for PowerShell	AWS Tools for PowerShell 程式碼範例
適用於 Python (Boto3) 的 AWS SDK	適用於 Python (Boto3) 的 AWS SDK 程式碼範例
適用於 Ruby 的 AWS SDK	適用於 Ruby 的 AWS SDK 程式碼範例
適用於 Rust 的 AWS SDK	適用於 Rust 的 AWS SDK 程式碼範例
適用於 SAP ABAP 的 AWS SDK	適用於 SAP ABAP 的 AWS SDK 程式碼範例
適用於 Swift 的 AWS SDK	適用於 Swift 的 AWS SDK 程式碼範例

如需 Lambda 專屬範例，請參閱 使用 AWS SDKs Lambda 程式碼範例。

可用性範例

找不到所需的內容嗎？ 請使用本頁面底部的提供意見回饋連結申請程式碼範例。

使用 AWS SDKs Lambda 程式碼範例

下列程式碼範例示範如何搭配 AWS 軟體開發套件 (SDK) 使用 Lambda。

基本概念是程式碼範例，這些範例說明如何在服務內執行基本操作。

Actions 是大型程式的程式碼摘錄，必須在內容中執行。雖然動作會告訴您如何呼叫個別服務函數，但您可以在其相關情境中查看內容中的動作。

案例是向您展示如何呼叫服務中的多個函數或與其他 AWS 服務組合來完成特定任務的程式碼範例。

AWS 社群貢獻是由多個團隊所建立和維護的範例 AWS。若要提供意見回饋，請使用連結儲存庫中提供的機制。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

程式碼範例

- 使用 AWS SDKs Lambda 基本範例
  - Hello Lambda
  - 使用 AWS SDK 了解 Lambda 的基本概念
  - 使用 AWS SDKs的 Lambda 動作
    - 搭配使用 CreateAlias 與 CLI
    - CreateFunction 搭配 AWS SDK 或 CLI 使用
    - 搭配使用 DeleteAlias 與 CLI
    - DeleteFunction 搭配 AWS SDK 或 CLI 使用
    - 搭配使用 DeleteFunctionConcurrency 與 CLI
    - 搭配使用 DeleteProvisionedConcurrencyConfig 與 CLI
    - 搭配使用 GetAccountSettings 與 CLI
    - 搭配使用 GetAlias 與 CLI
    - GetFunction 搭配 AWS SDK 或 CLI 使用
    - 搭配使用 GetFunctionConcurrency 與 CLI
    - 搭配使用 GetFunctionConfiguration 與 CLI
    - 搭配使用 GetPolicy 與 CLI
    - 搭配使用 GetProvisionedConcurrencyConfig 與 CLI

- Invoke 搭配 AWS SDK 或 CLI 使用
- ListFunctions 搭配 AWS SDK 或 CLI 使用
- 搭配使用 ListProvisionedConcurrencyConfigs 與 CLI
- 搭配使用 ListTags 與 CLI
- 搭配使用 ListVersionsByFunction 與 CLI
- 搭配使用 PublishVersion 與 CLI
- 搭配使用 PutFunctionConcurrency 與 CLI
- 搭配使用 PutProvisionedConcurrencyConfig 與 CLI
- 搭配使用 RemovePermission 與 CLI
- 搭配使用 TagResource 與 CLI
- 搭配使用 UntagResource 與 CLI
- 搭配使用 UpdateAlias 與 CLI
- UpdateFunctionCode 搭配 AWS SDK 或 CLI 使用
- UpdateFunctionConfiguration 搭配 AWS SDK 或 CLI 使用

- 使用 AWS SDKs Lambda 案例
  - 使用 SDK 透過 Lambda 函數自動確認已知的 Amazon Cognito 使用者 AWS
  - 使用 SDK 透過 Lambda 函數自動遷移已知的 Amazon Cognito 使用者 AWS
  - 建立 API Gateway REST API 以追蹤 COVID-19 資料
  - 建立出借圖書館 REST API
  - 使用 Step Functions 建立傳訊應用程式
  - 建立相片資產管理應用程式，讓使用者以標籤管理相片
  - 使用 API Gateway 建立 websocket 聊天應用程式
  - 建立可分析客戶意見回饋並合成音訊的應用程式
  - 從瀏覽器調用 Lambda 函式
  - 使用 S3 Object Lambda 轉換應用程式的資料
  - 使用 API Gateway 來調用 Lambda 函式
  - 使用 Step Functions 調用 Lambda 函式
  - 使用排程事件來調用 Lambda 函式
  - 使用 Amazon Neptune API 開發可查詢圖形資料的 Lambda 函式
  - 使用 AWS SDK 進行 Amazon Cognito 使用者身分驗證後，使用 Lambda 函數寫入自訂活動資料

- Lambda 的無伺服器範例
  - 連線至 Lambda 函數中的 Amazon RDS 資料庫
  - 使用 Kinesis 觸發條件調用 Lambda 函數
  - 使用 DynamoDB 觸發條件調用 Lambda 函數
  - 使用 Amazon DocumentDB 觸發條件調用 Lambda 函數
  - 使用 Amazon MSK 觸發條件調用 Lambda 函數
  - 使用 Amazon S3 觸發條件調用 Lambda 函數
  - 使用 Amazon SNS 觸發條件調用 Lambda 函數
  - 使用 Amazon SQS 觸發條件調用 Lambda 函數
  - 使用 Kinesis 觸發條件報告 Lambda 函數的批次項目失敗
  - 使用 DynamoDB 觸發條件報告 Lambda 函式的批次項目失敗
  - 使用 Amazon SQS 觸發條件報告 Lambda 函數的批次項目失敗
- AWS Lambda 的社群貢獻
  - 建置和測試無伺服器應用程式

使用 AWS SDKs Lambda 基本範例

下列程式碼範例示範如何 AWS Lambda 搭配 AWS SDKs 使用 的基本概念。

範例

- Hello Lambda
- 使用 AWS SDK 了解 Lambda 的基本概念
- 使用 AWS SDKs的 Lambda 動作
  - 搭配使用 CreateAlias 與 CLI
  - CreateFunction 搭配 AWS SDK 或 CLI 使用
  - 搭配使用 DeleteAlias 與 CLI
  - DeleteFunction 搭配 AWS SDK 或 CLI 使用
  - 搭配使用 DeleteFunctionConcurrency 與 CLI
  - 搭配使用 DeleteProvisionedConcurrencyConfig 與 CLI
  - 搭配使用 GetAccountSettings 與 CLI
  - 搭配使用 GetAlias 與 CLI

- GetFunction 搭配 AWS SDK 或 CLI 使用
- 搭配使用 GetFunctionConcurrency 與 CLI
- 搭配使用 GetFunctionConfiguration 與 CLI
- 搭配使用 GetPolicy 與 CLI
- 搭配使用 GetProvisionedConcurrencyConfig 與 CLI
- Invoke 搭配 AWS SDK 或 CLI 使用
- ListFunctions 搭配 AWS SDK 或 CLI 使用
- 搭配使用 ListProvisionedConcurrencyConfigs 與 CLI
- 搭配使用 ListTags 與 CLI
- 搭配使用 ListVersionsByFunction 與 CLI
- 搭配使用 PublishVersion 與 CLI
- 搭配使用 PutFunctionConcurrency 與 CLI
- 搭配使用 PutProvisionedConcurrencyConfig 與 CLI
- 搭配使用 RemovePermission 與 CLI
- 搭配使用 TagResource 與 CLI
- 搭配使用 UntagResource 與 CLI
- 搭配使用 UpdateAlias 與 CLI
- UpdateFunctionCode 搭配 AWS SDK 或 CLI 使用
- UpdateFunctionConfiguration 搭配 AWS SDK 或 CLI 使用

Hello Lambda

下列程式碼範例示範如何開始使用 Lambda。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
namespace LambdaActions;

using Amazon.Lambda;

public class HelloLambda
{
    static async Task Main(string[] args)
    {
        var lambdaClient = new AmazonLambdaClient();

        Console.WriteLine("Hello AWS Lambda");
        Console.WriteLine("Let's get started with AWS Lambda by listing your
 existing Lambda functions:");

        var response = await lambdaClient.ListFunctionsAsync();
        response.Functions.ForEach(function =>
        {

 Console.WriteLine($"{function.FunctionName}\t{function.Description}");
        });
    }
}
```

- 如需 API 詳細資訊，請參閱《適用於 .NET 的 AWS SDK API 參考》中的 ListFunctions。

C++

適用於 C++ 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

CMakeLists.txt CMake 檔案的程式碼。

```
# Set the minimum required version of CMake for this project.
cmake_minimum_required(VERSION 3.13)
```

```
# Set the AWS service components used by this project.
set(SERVICE_COMPONENTS lambda)

# Set this project's name.
project("hello_lambda")

# Set the C++ standard to use to build this target.
# At least C++ 11 is required for the AWS SDK for C++.
set(CMAKE_CXX_STANDARD 11)

# Use the MSVC variable to determine if this is a Windows build.
set(WINDOWS_BUILD ${MSVC})

if (WINDOWS_BUILD) # Set the location where CMake can find the installed
 libraries for the AWS SDK.
    string(REPLACE ";" "/aws-cpp-sdk-all;" SYSTEM_MODULE_PATH
 "${CMAKE_SYSTEM_PREFIX_PATH}/aws-cpp-sdk-all")
    list(APPEND CMAKE_PREFIX_PATH ${SYSTEM_MODULE_PATH})
endif ()

# Find the AWS SDK for C++ package.
find_package(AWSSDK REQUIRED COMPONENTS ${SERVICE_COMPONENTS})

if (WINDOWS_BUILD AND AWSSDK_INSTALL_AS_SHARED_LIBS)
     # Copy relevant AWS SDK for C++ libraries into the current binary directory
 for running and debugging.

     # set(BIN_SUB_DIR "/Debug") # if you are building from the command line you
 may need to uncomment this
                                     # and set the proper subdirectory to the
 executables' location.

     AWSSDK_CPY_DYN_LIBS(SERVICE_COMPONENTS ""
 ${CMAKE_CURRENT_BINARY_DIR}${BIN_SUB_DIR})
endif ()

add_executable(${PROJECT_NAME}
        hello_lambda.cpp)

target_link_libraries(${PROJECT_NAME}
        ${AWSSDK_LINK_LIBRARIES})
```

hello_lambda.cpp 來源檔案的程式碼。

```
#include <aws/core/Aws.h>
#include <aws/lambda/LambdaClient.h>
#include <aws/lambda/model/ListFunctionsRequest.h>
#include <iostream>

/*
 *  A "Hello Lambda" starter application which initializes an AWS Lambda (Lambda)
 client and lists the Lambda functions.
 *
 *  main function
 *
 *  Usage: 'hello_lambda'
 *
 */

int main(int argc, char **argv) {
    Aws::SDKOptions options;
    // Optionally change the log level for debugging.
//   options.loggingOptions.logLevel = Utils::Logging::LogLevel::Debug;
    Aws::InitAPI(options); // Should only be called once.
    int result = 0;
    {
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region (overrides config file).
        // clientConfig.region = "us-east-1";

        Aws::Lambda::LambdaClient lambdaClient(clientConfig);
        std::vector<Aws::String> functions;
        Aws::String marker; // Used for pagination.

        do {
            Aws::Lambda::Model::ListFunctionsRequest request;
            if (!marker.empty()) {
                request.SetMarker(marker);
            }

            Aws::Lambda::Model::ListFunctionsOutcome outcome =
 lambdaClient.ListFunctions(
                    request);

            if (outcome.IsSuccess()) {
```

```
                const Aws::Lambda::Model::ListFunctionsResult
 &listFunctionsResult = outcome.GetResult();
                std::cout << listFunctionsResult.GetFunctions().size()
                          << " lambda functions were retrieved." << std::endl;

                for (const Aws::Lambda::Model::FunctionConfiguration
 &functionConfiguration: listFunctionsResult.GetFunctions()) {
                    functions.push_back(functionConfiguration.GetFunctionName());
                    std::cout << functions.size() << "  "
                              << functionConfiguration.GetDescription() <<
 std::endl;
                    std::cout << "   "
                              <<
 Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                      functionConfiguration.GetRuntime()) << ": "
                              << functionConfiguration.GetHandler()
                              << std::endl;
                }
                marker = listFunctionsResult.GetNextMarker();
            } else {
                std::cerr << "Error with Lambda::ListFunctions. "
                          << outcome.GetError().GetMessage()
                          << std::endl;
                result = 1;
                break;
            }
        } while (!marker.empty());
    }


    Aws::ShutdownAPI(options); // Should only be called once.
    return result;
}
```

- 如需 API 詳細資訊，請參閱《適用於 C++ 的 AWS SDK API 參考》中的 ListFunctions。

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
)

// main uses the AWS SDK for Go (v2) to create an AWS Lambda client and list up
 to 10
// functions in your account.
// This example uses the default settings specified in your shared credentials
// and config files.
func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  fmt.Println("Couldn't load default configuration. Have you set up your AWS
 account?")
  fmt.Println(err)
  return
 }
 lambdaClient := lambda.NewFromConfig(sdkConfig)

 maxItems := 10
 fmt.Printf("Let's list up to %v functions for your account.\n", maxItems)
 result, err := lambdaClient.ListFunctions(ctx, &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
```

```
 })
 if err != nil {
  fmt.Printf("Couldn't list functions for your account. Here's why: %v\n", err)
  return
 }
 if len(result.Functions) == 0 {
  fmt.Println("You don't have any functions!")
 } else {
  for _, function := range result.Functions {
   fmt.Printf("\t%v\n", *function.FunctionName)
  }
 }
}
```

- 如需 API 詳細資訊，請參閱《適用於 Go 的 AWS SDK API 參考》中的 ListFunctions。

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /**
     * Lists the AWS Lambda functions associated with the current AWS account.
     *
     * @param awsLambda an instance of the {@link LambdaClient} class, which is
used to interact with the AWS Lambda service
     *
     * @throws LambdaException if an error occurs while interacting with the AWS
Lambda service
     */
    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
```

```
                System.out.println("The function name is " +
 config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Java 2.x API 參考》中的 ListFunctions。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import { LambdaClient, paginateListFunctions } from "@aws-sdk/client-lambda";

const client = new LambdaClient({});

export const helloLambda = async () => {
  const paginator = paginateListFunctions({ client }, {});
  const functions = [];

  for await (const page of paginator) {
    const funcNames = page.Functions.map((f) => f.FunctionName);
    functions.push(...funcNames);
  }

  console.log("Functions:");
  console.log(functions.join("\n"));
  return functions;
};
```

- 如需 API 詳細資訊，請參閱《適用於 JavaScript 的 AWS SDK API 參考》中的 ListFunctions。

Python

適用於 Python (Boto3) 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import boto3


def main():
    """
    List the Lambda functions in your AWS account.
    """
    # Create the Lambda client
    lambda_client = boto3.client("lambda")

    # Use the paginator to list the functions
    paginator = lambda_client.get_paginator("list_functions")
    response_iterator = paginator.paginate()

    print("Here are the Lambda functions in your account:")
    for page in response_iterator:
        for function in page["Functions"]:
            print(f"  {function['FunctionName']}")


if __name__ == "__main__":
    main()
```

- 如需 API 詳細資訊，請參閱《適用於 Python (Boto3) 的AWS 開發套件 API 參考》中的「ListFunctions」。

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
require 'aws-sdk-lambda'

# Creates an AWS Lambda client using the default credentials and configuration
def lambda_client
  Aws::Lambda::Client.new
end

# Lists the Lambda functions in your AWS account, paginating the results if
 necessary
def list_lambda_functions
  lambda = lambda_client

  # Use a pagination iterator to list all functions
  functions = []
  lambda.list_functions.each_page do |page|
    functions.concat(page.functions)
  end

  # Print the name and ARN of each function
  functions.each do |function|
    puts "Function name: #{function.function_name}"
    puts "Function ARN: #{function.function_arn}"
    puts
  end

  puts "Total functions: #{functions.count}"
end

list_lambda_functions if __FILE__ == $PROGRAM_NAME
```

- 如需 API 詳細資訊，請參閱《適用於 Ruby 的 AWS SDK API 參考》中的 ListFunctions。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 AWS SDK 了解 Lambda 的基本概念

下列程式碼範例示範如何：

- 建立 IAM 角色和 Lambda 函數，然後上傳處理常式程式碼。
- 調用具有單一參數的函數並取得結果。
- 更新函數程式碼並使用環境變數進行設定。
- 調用具有新參數的函數並取得結果。顯示傳回的執行日誌。
- 列出您帳戶的函數，然後清理相關資源。

如需詳細資訊，請參閱使用主控台建立 Lambda 函數。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

建立執行 Lambda 動作的方法。

```
namespace LambdaActions;

using Amazon.Lambda;
using Amazon.Lambda.Model;

/// <summary>
/// A class that implements AWS Lambda methods.
/// </summary>
public class LambdaWrapper
{
```

```
    private readonly IAmazonLambda _lambdaService;

    /// <summary>
    /// Constructor for the LambdaWrapper class.
    /// </summary>
    /// <param name="lambdaService">An initialized Lambda service client.</param>
    public LambdaWrapper(IAmazonLambda lambdaService)
    {
        _lambdaService = lambdaService;
    }

    /// <summary>
    /// Creates a new Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function.</param>
    /// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
    /// bucket where the zip file containing the code is located.</param>
    /// <param name="s3Key">The Amazon S3 key of the zip file.</param>
    /// <param name="role">The Amazon Resource Name (ARN) of a role with the
    /// appropriate Lambda permissions.</param>
    /// <param name="handler">The name of the handler function.</param>
    /// <returns>The Amazon Resource Name (ARN) of the newly created
    /// Lambda function.</returns>
    public async Task<string> CreateLambdaFunctionAsync(
        string functionName,
        string s3Bucket,
        string s3Key,
        string role,
        string handler)
    {
        // Defines the location for the function code.
        // S3Bucket - The S3 bucket where the file containing
        //            the source code is stored.
        // S3Key    - The name of the file containing the code.
        var functionCode = new FunctionCode
        {
            S3Bucket = s3Bucket,
            S3Key = s3Key,
        };

        var createFunctionRequest = new CreateFunctionRequest
        {
            FunctionName = functionName,
            Description = "Created by the Lambda .NET API",
```

```
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
            Role = role,
        };

        var reponse = await
 _lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
    }


    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
    public async Task<bool> DeleteFunctionAsync(string functionName)
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }


    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for
    /// which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string
 functionName)
    {
```

```
        var functionRequest = new GetFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
        return response.Configuration;
    }


    /// <summary>
    /// Invoke a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// invoke.</param
    /// <param name="parameters">The parameter values that will be passed to the
function.</param>
    /// <returns>A System Threading Task.</returns>
    public async Task<string> InvokeFunctionAsync(
        string functionName,
        string parameters)
    {
        var payload = parameters;
        var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };

        var response = await _lambdaService.InvokeAsync(request);
        MemoryStream stream = response.Payload;
        string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
        return returnValue;
    }


    /// <summary>
    /// Get a list of Lambda functions.
    /// </summary>
    /// <returns>A list of FunctionConfiguration objects.</returns>
    public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
    {
        var functionList = new List<FunctionConfiguration>();
```

```
        var functionPaginator =
            _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
        await foreach (var function in functionPaginator.Functions)
        {
            functionList.Add(function);
        }

        return functionList;
    }


    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };

        var response = await
 _lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
       Console.WriteLine($"The Function was last modified at
 {response.LastModified}.");
    }


    /// <summary>
    /// Update the code of a Lambda function.
    /// </summary>
```

```
    /// <param name="functionName">The name of the function to update.</param>
    /// <param name="functionHandler">The code that performs the function's
 actions.</param>
    /// <param name="environmentVariables">A dictionary of environment
 variables.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> UpdateFunctionConfigurationAsync(
        string functionName,
        string functionHandler,
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
 environmentVariables },
        };

        var response = await
 _lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }


}
```

建立可執行該案例的函數。

```
global using System.Threading.Tasks;
global using Amazon.IdentityManagement;
global using Amazon.Lambda;
global using LambdaActions;
global using LambdaScenarioCommon;
global using Microsoft.Extensions.DependencyInjection;
global using Microsoft.Extensions.Hosting;
global using Microsoft.Extensions.Logging;
global using Microsoft.Extensions.Logging.Console;
```

```
global using Microsoft.Extensions.Logging.Debug;


using Amazon.Lambda.Model;
using Microsoft.Extensions.Configuration;

namespace LambdaBasics;

public class LambdaBasics
{
    private static ILogger logger = null!;

    static async Task Main(string[] args)
    {
        // Set up dependency injection for the Amazon service.
        using var host = Host.CreateDefaultBuilder(args)
            .ConfigureLogging(logging =>
                logging.AddFilter("System", LogLevel.Debug)
                    .AddFilter<DebugLoggerProvider>("Microsoft",
 LogLevel.Information)
                    .AddFilter<ConsoleLoggerProvider>("Microsoft",
 LogLevel.Trace))
            .ConfigureServices((_, services) =>
            services.AddAWSService<IAmazonLambda>()
            .AddAWSService<IAmazonIdentityManagementService>()
            .AddTransient<LambdaWrapper>()
            .AddTransient<LambdaRoleWrapper>()
            .AddTransient<UIWrapper>()
        )
        .Build();

        var configuration = new ConfigurationBuilder()
            .SetBasePath(Directory.GetCurrentDirectory())
            .AddJsonFile("settings.json") // Load test settings from .json file.
            .AddJsonFile("settings.local.json",
            true) // Optionally load local settings.
        .Build();


        logger = LoggerFactory.Create(builder => { builder.AddConsole(); })
            .CreateLogger<LambdaBasics>();

        var lambdaWrapper = host.Services.GetRequiredService<LambdaWrapper>();
```

```
        var lambdaRoleWrapper =
host.Services.GetRequiredService<LambdaRoleWrapper>();
        var uiWrapper = host.Services.GetRequiredService<UIWrapper>();

        string functionName = configuration["FunctionName"]!;
        string roleName = configuration["RoleName"]!;
        string policyDocument = "{" +
            " \"Version\": \"2012-10-17\"," +
            " \"Statement\": [ " +
            "    {" +
            "        \"Effect\": \"Allow\"," +
            "        \"Principal\": {" +
            "            \"Service\": \"lambda.amazonaws.com\" " +
            "    }," +
            "        \"Action\": \"sts:AssumeRole\" " +
            "    }" +
            "]" +
        "}";

        var incrementHandler = configuration["IncrementHandler"];
        var calculatorHandler = configuration["CalculatorHandler"];
        var bucketName = configuration["BucketName"];
        var incrementKey = configuration["IncrementKey"];
        var calculatorKey = configuration["CalculatorKey"];
        var policyArn = configuration["PolicyArn"];

        uiWrapper.DisplayLambdaBasicsOverview();

        // Create the policy to use with the AWS Lambda functions and then attach
the
        // policy to a new role.
        var roleArn = await lambdaRoleWrapper.CreateLambdaRoleAsync(roleName,
policyDocument);

        Console.WriteLine("Waiting for role to become active.");
        uiWrapper.WaitABit(15, "Wait until the role is active before trying to
use it.");

        // Attach the appropriate AWS Identity and Access Management (IAM) role
policy to the new role.
        var success = await
lambdaRoleWrapper.AttachLambdaRolePolicyAsync(policyArn, roleName);
        uiWrapper.WaitABit(10, "Allow time for the IAM policy to be attached to
the role.");
```

```
        // Create the Lambda function using a zip file stored in an Amazon Simple
Storage Service
        // (Amazon S3) bucket.
        uiWrapper.DisplayTitle("Create Lambda Function");
        Console.WriteLine($"Creating the AWS Lambda function: {functionName}.");
        var lambdaArn = await lambdaWrapper.CreateLambdaFunctionAsync(
            functionName,
            bucketName,
            incrementKey,
            roleArn,
            incrementHandler);

        Console.WriteLine("Waiting for the new function to be available.");
        Console.WriteLine($"The AWS Lambda ARN is {lambdaArn}");

        // Get the Lambda function.
        Console.WriteLine($"Getting the {functionName} AWS Lambda function.");
        FunctionConfiguration config;
        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.State != State.Active);

        Console.WriteLine($"\nThe function, {functionName} has been created.");
        Console.WriteLine($"The runtime of this Lambda function is
{config.Runtime}.");

        uiWrapper.PressEnter();

        // List the Lambda functions.
        uiWrapper.DisplayTitle("Listing all Lambda functions.");
        var functions = await lambdaWrapper.ListFunctionsAsync();
        DisplayFunctionList(functions);

        uiWrapper.DisplayTitle("Invoke increment function");
        Console.WriteLine("Now that it has been created, invoke the Lambda
increment function.");
        string? value;
        do
        {
            Console.Write("Enter a value to increment: ");
```

```
            value = Console.ReadLine();
        }
        while (string.IsNullOrEmpty(value));

        string functionParameters = "{" +
            "\"action\": \"increment\", " +
            "\"x\": \"" + value + "\"" +
        "}";
        var answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
        Console.WriteLine($"{value} + 1 = {answer}.");

        uiWrapper.DisplayTitle("Update function");
        Console.WriteLine("Now update the Lambda function code.");
        await lambdaWrapper.UpdateFunctionCodeAsync(functionName, bucketName,
calculatorKey);

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

        await lambdaWrapper.UpdateFunctionConfigurationAsync(
            functionName,
            calculatorHandler,
            new Dictionary<string, string> { { "LOG_LEVEL", "DEBUG" } });

        do
        {
            config = await lambdaWrapper.GetFunctionAsync(functionName);
            Console.Write(".");
        }
        while (config.LastUpdateStatus == LastUpdateStatus.InProgress);

        uiWrapper.DisplayTitle("Call updated function");
        Console.WriteLine("Now call the updated function...");

        bool done = false;

        do
        {
            string? opSelected;
```

```
            Console.WriteLine("Select the operation to perform:");
            Console.WriteLine("\t1. add");
            Console.WriteLine("\t2. subtract");
            Console.WriteLine("\t3. multiply");
            Console.WriteLine("\t4. divide");
            Console.WriteLine("\tOr enter \"q\" to quit.");
            Console.WriteLine("Enter the number (1, 2, 3, 4, or q) of the
operation you want to perform: ");
            do
            {
                Console.Write("Your choice? ");
                opSelected = Console.ReadLine();
            }
            while (opSelected == string.Empty);

            var operation = (opSelected) switch
            {
                "1" => "add",
                "2" => "subtract",
                "3" => "multiply",
                "4" => "divide",
                "q" => "quit",
                _ => "add",
            };

            if (operation == "quit")
            {
                done = true;
            }
            else
            {
                // Get two numbers and an action from the user.
                value = string.Empty;
                do
                {
                    Console.Write("Enter the first value: ");
                    value = Console.ReadLine();
                }
                while (value == string.Empty);

                string? value2;
                do
                {
```

```
                    Console.Write("Enter a second value: ");
                    value2 = Console.ReadLine();
                }
                while (value2 == string.Empty);

                functionParameters = "{" +
                    "\"action\": \"" + operation + "\", " +
                    "\"x\": \"" + value + "\"," +
                    "\"y\": \"" + value2 + "\"" +
                "}";

                answer = await lambdaWrapper.InvokeFunctionAsync(functionName,
functionParameters);
                Console.WriteLine($"The answer when we {operation} the two
numbers is: {answer}.");
            }

            uiWrapper.PressEnter();
        } while (!done);

        // Delete the function created earlier.

        uiWrapper.DisplayTitle("Clean up resources");
        // Detach the IAM policy from the IAM role.
        Console.WriteLine("First detach the IAM policy from the role.");
        success = await lambdaRoleWrapper.DetachLambdaRolePolicyAsync(policyArn,
roleName);
        uiWrapper.WaitABit(15, "Let's wait for the policy to be fully detached
from the role.");

        Console.WriteLine("Delete the AWS Lambda function.");
        success = await lambdaWrapper.DeleteFunctionAsync(functionName);
        if (success)
        {
            Console.WriteLine($"The {functionName} function was deleted.");
        }
        else
        {
            Console.WriteLine($"Could not remove the function {functionName}");
        }

        // Now delete the IAM role created for use with the functions
        // created by the application.
        Console.WriteLine("Now we can delete the role that we created.");
```

```
        success = await lambdaRoleWrapper.DeleteLambdaRoleAsync(roleName);
        if (success)
        {
            Console.WriteLine("The role has been successfully removed.");
        }
        else
        {
            Console.WriteLine("Couldn't delete the role.");
        }

        Console.WriteLine("The Lambda Scenario is now complete.");
        uiWrapper.PressEnter();

        // Displays a formatted list of existing functions returned by the
        // LambdaMethods.ListFunctions.
        void DisplayFunctionList(List<FunctionConfiguration> functions)
        {
            functions.ForEach(functionConfig =>
            {

 Console.WriteLine($"{functionConfig.FunctionName}\t{functionConfig.Description}");
            });
        }
    }
}


namespace LambdaActions;

using Amazon.IdentityManagement;
using Amazon.IdentityManagement.Model;

public class LambdaRoleWrapper
{
    private readonly IAmazonIdentityManagementService _lambdaRoleService;

    public LambdaRoleWrapper(IAmazonIdentityManagementService lambdaRoleService)
    {
        _lambdaRoleService = lambdaRoleService;
    }

    /// <summary>
    /// Attach an AWS Identity and Access Management (IAM) role policy to the
```

```
    /// IAM role to be assumed by the AWS Lambda functions created for the
 scenario.
    /// </summary>
    /// <param name="policyArn">The Amazon Resource Name (ARN) of the IAM
 policy.</param>
    /// <param name="roleName">The name of the IAM role to attach the IAM policy
 to.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> AttachLambdaRolePolicyAsync(string policyArn, string
 roleName)
    {
        var response = await _lambdaRoleService.AttachRolePolicyAsync(new
 AttachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    /// <summary>
    /// Create a new IAM role.
    /// </summary>
    /// <param name="roleName">The name of the IAM role to create.</param>
    /// <param name="policyDocument">The policy document for the new IAM role.</
param>
    /// <returns>A string representing the ARN for newly created role.</returns>
    public async Task<string> CreateLambdaRoleAsync(string roleName, string
 policyDocument)
    {
        var request = new CreateRoleRequest
        {
            AssumeRolePolicyDocument = policyDocument,
            RoleName = roleName,
        };

        var response = await _lambdaRoleService.CreateRoleAsync(request);
        return response.Role.Arn;
    }

    /// <summary>
    /// Deletes an IAM role.
    /// </summary>
    /// <param name="roleName">The name of the role to delete.</param>
    /// <returns>A Boolean value indicating the success of the operation.</
returns>
    public async Task<bool> DeleteLambdaRoleAsync(string roleName)
    {
```

```
        var request = new DeleteRoleRequest
        {
            RoleName = roleName,
        };

        var response = await _lambdaRoleService.DeleteRoleAsync(request);
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }

    public async Task<bool> DetachLambdaRolePolicyAsync(string policyArn, string
 roleName)
    {
        var response = await _lambdaRoleService.DetachRolePolicyAsync(new
 DetachRolePolicyRequest { PolicyArn = policyArn, RoleName = roleName });
        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
}


namespace LambdaScenarioCommon;

public class UIWrapper
{
    public readonly string SepBar = new('-', Console.WindowWidth);

    /// <summary>
    /// Show information about the AWS Lambda Basics scenario.
    /// </summary>
    public void DisplayLambdaBasicsOverview()
    {
        Console.Clear();

        DisplayTitle("Welcome to AWS Lambda Basics");
        Console.WriteLine("This example application does the following:");
        Console.WriteLine("\t1. Creates an AWS Identity and Access Management
 (IAM) role that will be assumed by the functions we create.");
        Console.WriteLine("\t2. Attaches an IAM role policy that has Lambda
 permissions.");
        Console.WriteLine("\t3. Creates a Lambda function that increments the
 value passed to it.");
        Console.WriteLine("\t4. Calls the increment function and passes a
 value.");
        Console.WriteLine("\t5. Updates the code so that the function is a simple
 calculator.");
```

```
        Console.WriteLine("\t6. Calls the calculator function with the values
entered.");
        Console.WriteLine("\t7. Deletes the Lambda function.");
        Console.WriteLine("\t7. Detaches the IAM role policy.");
        Console.WriteLine("\t8. Deletes the IAM role.");
        PressEnter();
    }

    /// <summary>
    /// Display a message and wait until the user presses enter.
    /// </summary>
    public void PressEnter()
    {
        Console.Write("\nPress <Enter> to continue. ");
        _ = Console.ReadLine();
        Console.WriteLine();
    }

    /// <summary>
    /// Pad a string with spaces to center it on the console display.
    /// </summary>
    /// <param name="strToCenter">The string to be centered.</param>
    /// <returns>The padded string.</returns>
    public string CenterString(string strToCenter)
    {
        var padAmount = (Console.WindowWidth - strToCenter.Length) / 2;
        var leftPad = new string(' ', padAmount);
        return $"{leftPad}{strToCenter}";
    }

    /// <summary>
    /// Display a line of hyphens, the centered text of the title and another
    /// line of hyphens.
    /// </summary>
    /// <param name="strTitle">The string to be displayed.</param>
    public void DisplayTitle(string strTitle)
    {
        Console.WriteLine(SepBar);
        Console.WriteLine(CenterString(strTitle));
        Console.WriteLine(SepBar);
    }

    /// <summary>
    /// Display a countdown and wait for a number of seconds.
```

```
    /// </summary>
    /// <param name="numSeconds">The number of seconds to wait.</param>
    public void WaitABit(int numSeconds, string msg)
    {
        Console.WriteLine(msg);

        // Wait for the requested number of seconds.
        for (int i = numSeconds; i > 0; i--)
        {
            System.Threading.Thread.Sleep(1000);
            Console.Write($"{i}...");
        }

        PressEnter();
    }
}
```

定義增量一個數字的 Lambda 處理常式。

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaIncrement;

public class Function
{

    /// <summary>
    /// A simple function increments the integer parameter.
    /// </summary>
    /// <param name="input">A JSON string containing an action, which must be
    /// "increment" and a string representing the value to increment.</param>
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the incremented value of the parameter.</
returns>
```

```
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
    {
        if (input["action"] == "increment")
        {
            int inputValue = Convert.ToInt32(input["x"]);
            return inputValue + 1;
        }
        else
        {
            return 0;
        }
    }
}
```

定義可執行算術運算的第二個 Lambda 處理常式。

```
using Amazon.Lambda.Core;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaCalculator;

public class Function
{

    /// <summary>
    /// A simple function that takes two number in string format and performs
    /// the requested arithmetic function.
    /// </summary>
    /// <param name="input">JSON data containing an action, and x and y values.
    /// Valid actions include: add, subtract, multiply, and divide.</param>
    /// <param name="context">The context object passed by Lambda containing
    /// information about invocation, function, and execution environment.</
param>
    /// <returns>A string representing the results of the calculation.</returns>
    public int FunctionHandler(Dictionary<string, string> input, ILambdaContext
 context)
```

```
    {
        var action = input["action"];
        int x = Convert.ToInt32(input["x"]);
        int y = Convert.ToInt32(input["y"]);
        int result;
        switch (action)
        {
            case "add":
                result = x + y;
                break;
            case "subtract":
                result = x - y;
                break;
            case "multiply":
                result = x * y;
                break;
            case "divide":
                if (y == 0)
                {
                    Console.Error.WriteLine("Divide by zero error.");
                    result = 0;
                }
                else
                    result = x / y;
                break;
            default:
                Console.Error.WriteLine($"{action} is not a valid operation.");
                result = 0;
                break;
        }
        return result;
    }
}
```

- 如需 API 詳細資訊，請參閱《適用於 .NET 的 AWS SDK API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke

- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

C++

適用於 C++ 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
//! Get started with functions scenario.
/*!
 \param clientConfig: AWS client configuration.
 \return bool: Successful completion.
 */
bool AwsDoc::Lambda::getStartedWithFunctionsScenario(
        const Aws::Client::ClientConfiguration &clientConfig) {

    Aws::Lambda::LambdaClient client(clientConfig);

    // 1. Create an AWS Identity and Access Management (IAM) role for Lambda
 function.
    Aws::String roleArn;
    if (!getIamRoleArn(roleArn, clientConfig)) {
        return false;
    }

    // 2. Create a Lambda function.
    int seconds = 0;
    do {
        Aws::Lambda::Model::CreateFunctionRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
        request.SetTimeout(15);
        request.SetMemorySize(128);
```

```
        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_9);
#endif
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();

        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
 buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);

        Aws::Lambda::Model::CreateFunctionOutcome outcome =
 client.CreateFunction(
```

```
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
seconds
                      << " seconds elapsed." << std::endl;
            break;
        }
        else if (outcome.GetError().GetErrorType() ==
                 Aws::Lambda::LambdaErrors::INVALID_PARAMETER_VALUE &&
                 outcome.GetError().GetMessage().find("role") >= 0) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout
                        << "Waiting for the IAM role to become available as a
CreateFunction parameter. "
                        << seconds
                        << " seconds elapsed." << std::endl;

                std::cout << outcome.GetError().GetMessage() << std::endl;
            }
        }
        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            deleteIamRole(clientConfig);
            return false;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (60 > seconds);

    std::cout << "The current Lambda function increments 1 by an input." <<
std::endl;

    // 3.  Invoke the Lambda function.
    {
        int increment = askQuestionForInt("Enter an increment integer: ");

        Aws::Lambda::Model::InvokeResult invokeResult;
        Aws::Utils::Json::JsonValue jsonPayload;
        jsonPayload.WithString("action", "increment");
        jsonPayload.WithInteger("number", increment);
        if (invokeLambdaFunction(jsonPayload, Aws::Lambda::Model::LogType::Tail,
```

```
                                invokeResult, client)) {
            Aws::Utils::Json::JsonValue jsonValue(invokeResult.GetPayload());
            Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                    jsonValue.View().GetAllObjects();
            auto iter = values.find("result");
            if (iter != values.end() && iter->second.IsIntegerType()) {
                {
                    std::cout << INCREMENT_RESUlT_PREFIX
                              << iter->second.AsInteger() << std::endl;
                }
            }
            else {
                std::cout << "There was an error in execution. Here is the log."
                          << std::endl;
                Aws::Utils::ByteBuffer buffer =
 Aws::Utils::HashingUtils::Base64Decode(
                        invokeResult.GetLogResult());
                std::cout << "With log " << buffer.GetUnderlyingData() <<
 std::endl;
            }
        }
    }

    std::cout
            << "The Lambda function will now be updated with new code. Press
 return to continue, ";
    Aws::String answer;
    std::getline(std::cin, answer);

    // 4.  Update the Lambda function code.
    {
        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
```

```
#endif
            deleteLambdaFunction(client);
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);

        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
 client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
 std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    }

    std::cout
            << "This function uses an environment variable to control the logging
 level."
            << std::endl;
    std::cout
            << "UpdateFunctionConfiguration will be used to set the LOG_LEVEL to
 DEBUG."
            << std::endl;
    seconds = 0;

    // 5.  Update the Lambda function configuration.
    do {
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
        Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
        request.SetFunctionName(LAMBDA_NAME);
```

```
        Aws::Lambda::Model::Environment environment;
        environment.AddVariables("LOG_LEVEL", "DEBUG");
        request.SetEnvironment(environment);

        Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
client.UpdateFunctionConfiguration(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda configuration was successfully updated."
                      << std::endl;
            break;
        }

            // RESOURCE_IN_USE: function code update not completed.
        else if (outcome.GetError().GetErrorType() !=
                 Aws::Lambda::LambdaErrors::RESOURCE_IN_USE) {
            if ((seconds % 10) == 0) { // Log status every 10 seconds.
                std::cout << "Lambda function update in progress . After " <<
seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }

    } while (0 < seconds);

    if (0 > seconds) {
        std::cerr << "Function failed to become active." << std::endl;
    }
    else {
        std::cout << "Updated function active after " << seconds << " seconds."
                  << std::endl;
    }

    std::cout
            << "\nThe new code applies an arithmetic operator to two variables, x
an y."
            << std::endl;
```

```
    std::vector<Aws::String> operators = {"plus", "minus", "times", "divided-
by"};
    for (size_t i = 0; i < operators.size(); ++i) {
        std::cout << "   " << i + 1 << " " << operators[i] << std::endl;
    }

    // 6.  Invoke the updated Lambda function.
    do {
        int operatorIndex = askQuestionForIntRange("Select an operator index 1 -
 4 ", 1,
                                                   4);
        int x = askQuestionForInt("Enter an integer for the x value ");
        int y = askQuestionForInt("Enter an integer for the y value ");

        Aws::Utils::Json::JsonValue calculateJsonPayload;
        calculateJsonPayload.WithString("action", operators[operatorIndex - 1]);
        calculateJsonPayload.WithInteger("x", x);
        calculateJsonPayload.WithInteger("y", y);
        Aws::Lambda::Model::InvokeResult calculatedResult;
        if (invokeLambdaFunction(calculateJsonPayload,
                                 Aws::Lambda::Model::LogType::Tail,
                                 calculatedResult, client)) {
            Aws::Utils::Json::JsonValue jsonValue(calculatedResult.GetPayload());
            Aws::Map<Aws::String, Aws::Utils::Json::JsonView> values =
                    jsonValue.View().GetAllObjects();
            auto iter = values.find("result");
            if (iter != values.end() && iter->second.IsIntegerType()) {
                std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                          << operators[operatorIndex - 1] << " "
                          << y << " is " << iter->second.AsInteger() <<
 std::endl;
            }
            else if (iter != values.end() && iter->second.IsFloatingPointType())
 {
                std::cout << ARITHMETIC_RESULT_PREFIX << x << " "
                          << operators[operatorIndex - 1] << " "
                          << y << " is " << iter->second.AsDouble() << std::endl;
            }
            else {
                std::cout << "There was an error in execution. Here is the log."
                          << std::endl;
                Aws::Utils::ByteBuffer buffer =
 Aws::Utils::HashingUtils::Base64Decode(
                        calculatedResult.GetLogResult());
```

```
                std::cout << "With log " << buffer.GetUnderlyingData() <<
std::endl;
            }
        }

        answer = askQuestion("Would you like to try another operation? (y/n) ");
    } while (answer == "y");

    std::cout
            << "A list of the lambda functions will be retrieved. Press return to
continue, ";
    std::getline(std::cin, answer);

    // 7.  List the Lambda functions.

    std::vector<Aws::String> functions;
    Aws::String marker;

    do {
        Aws::Lambda::Model::ListFunctionsRequest request;
        if (!marker.empty()) {
            request.SetMarker(marker);
        }

        Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
                request);

        if (outcome.IsSuccess()) {
            const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
            std::cout << result.GetFunctions().size()
                      << " lambda functions were retrieved." << std::endl;

            for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
                functions.push_back(functionConfiguration.GetFunctionName());
                std::cout << functions.size() << "  "
                          << functionConfiguration.GetDescription() << std::endl;
                std::cout << "   "
                          <<
Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                  functionConfiguration.GetRuntime()) << ": "
                          << functionConfiguration.GetHandler()
                          << std::endl;
```

```
            }
            marker = result.GetNextMarker();
        }
        else {
            std::cerr << "Error with Lambda::ListFunctions. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    } while (!marker.empty());

    // 8.  Get a Lambda function.
    if (!functions.empty()) {
        std::stringstream question;
        question << "Choose a function to retrieve between 1 and " <<
functions.size()
                 << " ";
        int functionIndex = askQuestionForIntRange(question.str(), 1,

static_cast<int>(functions.size()));

        Aws::String functionName = functions[functionIndex - 1];

        Aws::Lambda::Model::GetFunctionRequest request;
        request.SetFunctionName(functionName);

        Aws::Lambda::Model::GetFunctionOutcome outcome =
client.GetFunction(request);

        if (outcome.IsSuccess()) {
            std::cout << "Function retrieve.\n" <<

outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
                      << std::endl;
        }
        else {
            std::cerr << "Error with Lambda::GetFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
    }

    std::cout << "The resources will be deleted. Press return to continue, ";
    std::getline(std::cin, answer);
```

```
    // 9.  Delete the Lambda function.
    bool result = deleteLambdaFunction(client);

    // 10. Delete the IAM role.
    return result && deleteIamRole(clientConfig);
}

//! Routine which invokes a Lambda function and returns the result.
/*!
 \param jsonPayload: Payload for invoke function.
 \param logType: Log type setting for invoke function.
 \param invokeResult: InvokeResult object to receive the result.
 \param client: Lambda client.
 \return bool: Successful completion.
 */
bool
AwsDoc::Lambda::invokeLambdaFunction(const Aws::Utils::Json::JsonValue
 &jsonPayload,
                                     Aws::Lambda::Model::LogType logType,
                                     Aws::Lambda::Model::InvokeResult
 &invokeResult,
                                     const Aws::Lambda::LambdaClient &client) {
    int seconds = 0;
    bool result = false;
    /*
     * In this example, the Invoke function can be called before recently created
 resources are
     * available.  The Invoke function is called repeatedly until the resources
 are
     * available.
     */
    do {
        Aws::Lambda::Model::InvokeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetLogType(logType);
        std::shared_ptr<Aws::IOStream> payload =
 Aws::MakeShared<Aws::StringStream>(
                "FunctionTest");
        *payload << jsonPayload.View().WriteReadable();
        request.SetBody(payload);
        request.SetContentType("application/json");
        Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

        if (outcome.IsSuccess()) {
```

```
            invokeResult = std::move(outcome.GetResult());
            result = true;
            break;
        }

            // ACCESS_DENIED: because the role is not available yet.
            // RESOURCE_CONFLICT: because the Lambda function is being created or
 updated.
        else if ((outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::ACCESS_DENIED) ||
                 (outcome.GetError().GetErrorType() ==
                  Aws::Lambda::LambdaErrors::RESOURCE_CONFLICT)) {
            if ((seconds % 5) == 0) { // Log status every 10 seconds.
                std::cout << "Waiting for the invoke api to be available, status
 " <<
                          ((outcome.GetError().GetErrorType() ==
                            Aws::Lambda::LambdaErrors::ACCESS_DENIED ?
                            "ACCESS_DENIED" : "RESOURCE_CONFLICT")) << ". " <<
 seconds
                          << " seconds elapsed." << std::endl;
            }
        }
        else {
            std::cerr << "Error with Lambda::InvokeRequest. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            break;
        }
        ++seconds;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    } while (seconds < 60);

    return result;
}
```

- 如需 API 詳細資訊，請參閱《適用於 C++ 的 AWS SDK API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke

- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

建立互動式案例，示範如何開始使用 Lambda 函數。

```
import (
 "archive/zip"
 "bytes"
 "context"
 "encoding/base64"
 "encoding/json"
 "errors"
 "fmt"
 "log"
 "os"
 "strings"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/iam"
 iamtypes "github.com/aws/aws-sdk-go-v2/service/iam/types"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/lambda/actions"
)

// GetStartedFunctionsScenario shows you how to use AWS Lambda to perform the
 following
// actions:
```

```
//
//  1. Create an AWS Identity and Access Management (IAM) role and Lambda
 function, then upload handler code.
//  2. Invoke the function with a single parameter and get results.
//  3. Update the function code and configure with an environment variable.
//  4. Invoke the function with new parameters and get results. Display the
 returned execution log.
//  5. List the functions for your account, then clean up resources.
type GetStartedFunctionsScenario struct {
 sdkConfig       aws.Config
 functionWrapper actions.FunctionWrapper
 questioner      demotools.IQuestioner
 helper          IScenarioHelper
 isTestRun       bool
}

// NewGetStartedFunctionsScenario constructs a GetStartedFunctionsScenario
 instance from a configuration.
// It uses the specified config to get a Lambda client and create wrappers for
 the actions
// used in the scenario.
func NewGetStartedFunctionsScenario(sdkConfig aws.Config, questioner
 demotools.IQuestioner,
 helper IScenarioHelper) GetStartedFunctionsScenario {
 lambdaClient := lambda.NewFromConfig(sdkConfig)
 return GetStartedFunctionsScenario{
  sdkConfig:       sdkConfig,
  functionWrapper: actions.FunctionWrapper{LambdaClient: lambdaClient},
  questioner:      questioner,
  helper:          helper,
 }
}

// Run runs the interactive scenario.
func (scenario GetStartedFunctionsScenario) Run(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong with the demo.\n")
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Println("Welcome to the AWS Lambda get started with functions demo.")
 log.Println(strings.Repeat("-", 88))
```

```
 role := scenario.GetOrCreateRole(ctx)
 funcName := scenario.CreateFunction(ctx, role)
 scenario.InvokeIncrement(ctx, funcName)
 scenario.UpdateFunction(ctx, funcName)
 scenario.InvokeCalculator(ctx, funcName)
 scenario.ListFunctions(ctx)
 scenario.Cleanup(ctx, role, funcName)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}

// GetOrCreateRole checks whether the specified role exists and returns it if it
 does.
// Otherwise, a role is created that specifies Lambda as a trusted principal.
// The AWSLambdaBasicExecutionRole managed policy is attached to the role and the
 role
// is returned.
func (scenario GetStartedFunctionsScenario) GetOrCreateRole(ctx context.Context)
 *iamtypes.Role {
 var role *iamtypes.Role
 iamClient := iam.NewFromConfig(scenario.sdkConfig)
 log.Println("First, we need an IAM role that Lambda can assume.")
 roleName := scenario.questioner.Ask("Enter a name for the role:",
 demotools.NotEmpty{})
 getOutput, err := iamClient.GetRole(ctx, &iam.GetRoleInput{
  RoleName: aws.String(roleName)})
 if err != nil {
  var noSuch *iamtypes.NoSuchEntityException
  if errors.As(err, &noSuch) {
   log.Printf("Role %v doesn't exist. Creating it....\n", roleName)
  } else {
   log.Panicf("Couldn't check whether role %v exists. Here's why: %v\n",
    roleName, err)
  }
 } else {
  role = getOutput.Role
  log.Printf("Found role %v.\n", *role.RoleName)
 }
 if role == nil {
  trustPolicy := PolicyDocument{
   Version: "2012-10-17",
```

```
   Statement: []PolicyStatement{{
    Effect:    "Allow",
    Principal: map[string]string{"Service": "lambda.amazonaws.com"},
    Action:    []string{"sts:AssumeRole"},
   }},
  }
  policyArn := "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
  createOutput, err := iamClient.CreateRole(ctx, &iam.CreateRoleInput{
   AssumeRolePolicyDocument: aws.String(trustPolicy.String()),
   RoleName:                 aws.String(roleName),
  })
  if err != nil {
   log.Panicf("Couldn't create role %v. Here's why: %v\n", roleName, err)
  }
  role = createOutput.Role
  _, err = iamClient.AttachRolePolicy(ctx, &iam.AttachRolePolicyInput{
   PolicyArn: aws.String(policyArn),
   RoleName:  aws.String(roleName),
  })
  if err != nil {
   log.Panicf("Couldn't attach a policy to role %v. Here's why: %v\n", roleName,
 err)
  }
  log.Printf("Created role %v.\n", *role.RoleName)
  log.Println("Let's give AWS a few seconds to propagate resources...")
  scenario.helper.Pause(10)
 }
 log.Println(strings.Repeat("-", 88))
 return role
}

// CreateFunction creates a Lambda function and uploads a handler written in
 Python.
// The code for the Python handler is packaged as a []byte in .zip format.
func (scenario GetStartedFunctionsScenario) CreateFunction(ctx context.Context,
 role *iamtypes.Role) string {
 log.Println("Let's create a function that increments a number.\n" +
  "The function uses the 'lambda_handler_basic.py' script found in the \n" +
  "'handlers' directory of this project.")
 funcName := scenario.questioner.Ask("Enter a name for the Lambda function:",
 demotools.NotEmpty{})
 zipPackage := scenario.helper.CreateDeploymentPackage("lambda_handler_basic.py",
 fmt.Sprintf("%v.py", funcName))
 log.Printf("Creating function %v and waiting for it to be ready.", funcName)
```

```
 funcState := scenario.functionWrapper.CreateFunction(ctx, funcName,
 fmt.Sprintf("%v.lambda_handler", funcName),
  role.Arn, zipPackage)
 log.Printf("Your function is %v.", funcState)
 log.Println(strings.Repeat("-", 88))
 return funcName
}

// InvokeIncrement invokes a Lambda function that increments a number. The
 function
// parameters are contained in a Go struct that is used to serialize the
 parameters to
// a JSON payload that is passed to the function.
// The result payload is deserialized into a Go struct that contains an int
 value.
func (scenario GetStartedFunctionsScenario) InvokeIncrement(ctx context.Context,
 funcName string) {
 parameters := actions.IncrementParameters{Action: "increment"}
 log.Println("Let's invoke our function. This function increments a number.")
 parameters.Number = scenario.questioner.AskInt("Enter a number to increment:",
 demotools.NotEmpty{})
 log.Printf("Invoking %v with %v...\n", funcName, parameters.Number)
 invokeOutput := scenario.functionWrapper.Invoke(ctx, funcName, parameters,
 false)
 var payload actions.LambdaResultInt
 err := json.Unmarshal(invokeOutput.Payload, &payload)
 if err != nil {
  log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
   funcName, err)
 }
 log.Printf("Invoking %v with %v returned %v.\n", funcName, parameters.Number,
 payload)
 log.Println(strings.Repeat("-", 88))
}

// UpdateFunction updates the code for a Lambda function by uploading a simple
 arithmetic
// calculator written in Python. The code for the Python handler is packaged as a
// []byte in .zip format.
// After the code is updated, the configuration is also updated with a new log
// level that instructs the handler to log additional information.
func (scenario GetStartedFunctionsScenario) UpdateFunction(ctx context.Context,
 funcName string) {
 log.Println("Let's update the function to an arithmetic calculator.\n" +
```

```
  "The function uses the 'lambda_handler_calculator.py' script found in the \n" +
  "'handlers' directory of this project.")
 scenario.questioner.Ask("Press Enter when you're ready.")
 log.Println("Creating deployment package...")
 zipPackage :=
 scenario.helper.CreateDeploymentPackage("lambda_handler_calculator.py",
  fmt.Sprintf("%v.py", funcName))
 log.Println("...and updating the Lambda function and waiting for it to be
 ready.")
 funcState := scenario.functionWrapper.UpdateFunctionCode(ctx, funcName,
 zipPackage)
 log.Printf("Updated function %v. Its current state is %v.", funcName, funcState)
 log.Println("This function uses an environment variable to control logging
 level.")
 log.Println("Let's set it to DEBUG to get the most logging.")
 scenario.functionWrapper.UpdateFunctionConfiguration(ctx, funcName,
  map[string]string{"LOG_LEVEL": "DEBUG"})
 log.Println(strings.Repeat("-", 88))
}

// InvokeCalculator invokes the Lambda calculator function. The parameters are
 stored in a
// Go struct that is used to serialize the parameters to a JSON payload. That
 payload is then passed
// to the function.
// The result payload is deserialized to a Go struct that stores the result as
 either an
// int or float32, depending on the kind of operation that was specified.
func (scenario GetStartedFunctionsScenario) InvokeCalculator(ctx context.Context,
 funcName string) {
 wantInvoke := true
 choices := []string{"plus", "minus", "times", "divided-by"}
 for wantInvoke {
  choice := scenario.questioner.AskChoice("Select an arithmetic operation:\n",
 choices)
  x := scenario.questioner.AskInt("Enter a value for x:", demotools.NotEmpty{})
  y := scenario.questioner.AskInt("Enter a value for y:", demotools.NotEmpty{})
  log.Printf("Invoking %v %v %v...", x, choices[choice], y)
  calcParameters := actions.CalculatorParameters{
   Action: choices[choice],
   X:      x,
   Y:      y,
  }
```

```
  invokeOutput := scenario.functionWrapper.Invoke(ctx, funcName, calcParameters,
 true)
  var payload any
  if choice == 3 { // divide-by results in a float.
   payload = actions.LambdaResultFloat{}
  } else {
   payload = actions.LambdaResultInt{}
  }
  err := json.Unmarshal(invokeOutput.Payload, &payload)
  if err != nil {
   log.Panicf("Couldn't unmarshal payload from invoking %v. Here's why: %v\n",
    funcName, err)
  }
  log.Printf("Invoking %v with %v %v %v returned %v.\n", funcName,
   calcParameters.X, calcParameters.Action, calcParameters.Y, payload)
  scenario.questioner.Ask("Press Enter to see the logs from the call.")
  logRes, err := base64.StdEncoding.DecodeString(*invokeOutput.LogResult)
  if err != nil {
   log.Panicf("Couldn't decode log result. Here's why: %v\n", err)
  }
  log.Println(string(logRes))
  wantInvoke = scenario.questioner.AskBool("Do you want to calculate again? (y/
n)", "y")
 }
 log.Println(strings.Repeat("-", 88))
}

// ListFunctions lists up to the specified number of functions for your account.
func (scenario GetStartedFunctionsScenario) ListFunctions(ctx context.Context) {
 count := scenario.questioner.AskInt(
  "Let's list functions for your account. How many do you want to see?",
 demotools.NotEmpty{})
 functions := scenario.functionWrapper.ListFunctions(ctx, count)
 log.Printf("Found %v functions:", len(functions))
 for _, function := range functions {
  log.Printf("\t%v", *function.FunctionName)
 }
 log.Println(strings.Repeat("-", 88))
}

// Cleanup removes the IAM and Lambda resources created by the example.
func (scenario GetStartedFunctionsScenario) Cleanup(ctx context.Context, role
 *iamtypes.Role, funcName string) {
```

```
 if scenario.questioner.AskBool("Do you want to clean up resources created for
 this example? (y/n)",
  "y") {
  iamClient := iam.NewFromConfig(scenario.sdkConfig)
  policiesOutput, err := iamClient.ListAttachedRolePolicies(ctx,
   &iam.ListAttachedRolePoliciesInput{RoleName: role.RoleName})
  if err != nil {
   log.Panicf("Couldn't get policies attached to role %v. Here's why: %v\n",
    *role.RoleName, err)
  }
  for _, policy := range policiesOutput.AttachedPolicies {
   _, err = iamClient.DetachRolePolicy(ctx, &iam.DetachRolePolicyInput{
    PolicyArn: policy.PolicyArn, RoleName: role.RoleName,
   })
   if err != nil {
    log.Panicf("Couldn't detach policy %v from role %v. Here's why: %v\n",
     *policy.PolicyArn, *role.RoleName, err)
   }
  }
  _, err = iamClient.DeleteRole(ctx, &iam.DeleteRoleInput{RoleName:
 role.RoleName})
  if err != nil {
   log.Panicf("Couldn't delete role %v. Here's why: %v\n", *role.RoleName, err)
  }
  log.Printf("Deleted role %v.\n", *role.RoleName)

  scenario.functionWrapper.DeleteFunction(ctx, funcName)
  log.Printf("Deleted function %v.\n", funcName)
 } else {
  log.Println("Okay. Don't forget to delete the resources when you're done with
 them.")
 }
}

// IScenarioHelper abstracts I/O and wait functions from a scenario so that they
// can be mocked for unit testing.
type IScenarioHelper interface {
 Pause(secs int)
 CreateDeploymentPackage(sourceFile string, destinationFile string) *bytes.Buffer
}

// ScenarioHelper lets the caller specify the path to Lambda handler functions.
type ScenarioHelper struct {
 HandlerPath string
```

```
}

// Pause waits for the specified number of seconds.
func (helper *ScenarioHelper) Pause(secs int) {
 time.Sleep(time.Duration(secs) * time.Second)
}

// CreateDeploymentPackage creates an AWS Lambda deployment package from a source
 file. The
// deployment package is stored in .zip format in a bytes.Buffer. The buffer can
 be
// used to pass a []byte to Lambda when creating the function.
// The specified destinationFile is the name to give the file when it's deployed
 to Lambda.
func (helper *ScenarioHelper) CreateDeploymentPackage(sourceFile string,
 destinationFile string) *bytes.Buffer {
 var err error
 buffer := &bytes.Buffer{}
 writer := zip.NewWriter(buffer)
 zFile, err := writer.Create(destinationFile)
 if err != nil {
  log.Panicf("Couldn't create destination archive %v. Here's why: %v\n",
 destinationFile, err)
 }
 sourceBody, err := os.ReadFile(fmt.Sprintf("%v/%v", helper.HandlerPath,
 sourceFile))
 if err != nil {
  log.Panicf("Couldn't read handler source file %v. Here's why: %v\n",
   sourceFile, err)
 } else {
  _, err = zFile.Write(sourceBody)
  if err != nil {
   log.Panicf("Couldn't write handler %v to zip archive. Here's why: %v\n",
    sourceFile, err)
  }
 }
 err = writer.Close()
 if err != nil {
  log.Panicf("Couldn't close zip writer. Here's why: %v\n", err)
 }
 return buffer
}
```

建立包裝個別 Lambda 動作的結構。

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}


// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(ctx context.Context, functionName
 string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(ctx,
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
  state = funcOutput.Configuration.State
 }
 return state
}
```

```
// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(ctx context.Context, functionName
 string, handlerName string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.CreateFunction(ctx, &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython39,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}
```

```
// UpdateFunctionCode updates the code for the Lambda function specified by
 functionName.
// The existing code for the Lambda function is entirely replaced by the code in
 the
// zipPackage buffer. After the update action is called, a
 lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(ctx context.Context,
 functionName string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.UpdateFunctionCode(ctx,
 &lambda.UpdateFunctionCodeInput{
  FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
 })
 if err != nil {
  log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
 functionName, err)
 } else {
  waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}



// UpdateFunctionConfiguration updates a map of environment variables configured
 for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(ctx context.Context,
 functionName string, envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(ctx,
 &lambda.UpdateFunctionConfigurationInput{
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
```

```
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}



// ListFunctions lists up to maxItems functions for the account. This function
 uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(ctx context.Context, maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(ctx)
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}



// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(ctx context.Context, functionName
 string) {
 _, err := wrapper.LambdaClient.DeleteFunction(ctx, &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}
```

```
// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(ctx context.Context, functionName string,
 parameters any, getLog bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(ctx, &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
 }
 return invokeOutput
}



// IncrementParameters is used to serialize parameters to the increment Lambda
 handler.
type IncrementParameters struct {
 Action string `json:"action"`
 Number int    `json:"number"`
}

// CalculatorParameters is used to serialize parameters to the calculator Lambda
 handler.
type CalculatorParameters struct {
 Action string `json:"action"`
 X      int    `json:"x"`
 Y      int    `json:"y"`
}

// LambdaResultInt is used to deserialize an int result from a Lambda handler.
```

```
type LambdaResultInt struct {
 Result int `json:"result"`
}

// LambdaResultFloat is used to deserialize a float32 result from a Lambda
 handler.
type LambdaResultFloat struct {
 Result float32 `json:"result"`
}
```

定義增量一個數字的 Lambda 處理常式。

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)


def lambda_handler(event, context):
    """
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
    return response
```

定義可執行算術運算的第二個 Lambda 處理常式。

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}


def lambda_handler(event, context):
    """
    Accepts an action and two numbers, performs the specified action on the
 numbers,
    and returns the result.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
    result = None
    try:
```

```
        if func is not None and x is not None and y is not None:
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

- 如需 API 詳細資訊，請參閱《適用於 Go 的 AWS SDK API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
/*
 *  Lambda function names appear as:
 *
 *  arn:aws:lambda:us-west-2:335556666777:function:HelloFunction
 *
```

```
 *  To find this value, look at the function in the AWS Management Console.
 *
 *  Before running this Java code example, set up your development environment,
 including your credentials.
 *
 *  For more information, see this documentation topic:
 *
 *  https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-
started.html
 *
 *  This example performs the following tasks:
 *
 * 1. Creates an AWS Lambda function.
 * 2. Gets a specific AWS Lambda function.
 * 3. Lists all Lambda functions.
 * 4. Invokes a Lambda function.
 * 5. Updates the Lambda function code and invokes it again.
 * 6. Updates a Lambda function's configuration value.
 * 7. Deletes a Lambda function.
 */

public class LambdaScenario {
    public static final String DASHES = new String(new char[80]).replace("\0",
 "-");

    public static void main(String[] args) throws InterruptedException {
        final String usage = """

            Usage:
                <functionName> <role> <handler> <bucketName> <key>\s

            Where:
                functionName - The name of the Lambda function.\s
                role - The AWS Identity and Access Management (IAM) service role
 that has Lambda permissions.\s
                handler - The fully qualified method name (for example,
 example.Handler::handleRequest).\s
                bucketName - The Amazon Simple Storage Service (Amazon S3) bucket
 name that contains the .zip or .jar used to update the Lambda function's code.\s
                key - The Amazon S3 key name that represents the .zip or .jar
 (for example, LambdaHello-1.0-SNAPSHOT.jar).
                """;

        if (args.length != 5) {
```

```
            System.out.println(usage);
            return;
        }

        String functionName = args[0];
        String role = args[1];
        String handler = args[2];
        String bucketName = args[3];
        String key = args[4];
        LambdaClient awsLambda = LambdaClient.builder()
            .build();

        System.out.println(DASHES);
        System.out.println("Welcome to the AWS Lambda Basics scenario.");
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("1. Create an AWS Lambda function.");
        String funArn = createLambdaFunction(awsLambda, functionName, key,
bucketName, role, handler);
        System.out.println("The AWS Lambda ARN is " + funArn);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("2. Get the " + functionName + " AWS Lambda
function.");
        getFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("3. List all AWS Lambda functions.");
        listFunctions(awsLambda);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("4. Invoke the Lambda function.");
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("5. Update the Lambda function code and invoke it
again.");
```

```
        updateFunctionCode(awsLambda, functionName, bucketName, key);
        System.out.println("*** Sleep for 1 min to get Lambda function ready.");
        Thread.sleep(60000);
        invokeFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("6. Update a Lambda function's configuration value.");
        updateFunctionConfiguration(awsLambda, functionName, handler);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("7. Delete the AWS Lambda function.");
        LambdaScenario.deleteLambdaFunction(awsLambda, functionName);
        System.out.println(DASHES);

        System.out.println(DASHES);
        System.out.println("The AWS Lambda scenario completed successfully");
        System.out.println(DASHES);
        awsLambda.close();
    }

    /**
     * Creates a new Lambda function in AWS using the AWS Lambda Java API.
     *
     * @param awsLambda    the AWS Lambda client used to interact with the AWS
Lambda service
     * @param functionName the name of the Lambda function to create
     * @param key          the S3 key of the function code
     * @param bucketName   the name of the S3 bucket containing the function code
     * @param role         the IAM role to assign to the Lambda function
     * @param handler      the fully qualified class name of the function handler
     * @return the Amazon Resource Name (ARN) of the created Lambda function
     */
    public static String createLambdaFunction(LambdaClient awsLambda,
                                               String functionName,
                                               String key,
                                               String bucketName,
                                               String role,
                                               String handler) {

        try {
            LambdaWaiter waiter = awsLambda.waiter();
            FunctionCode code = FunctionCode.builder()
```

```
                .s3Key(key)
                .s3Bucket(bucketName)
                .build();

            CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
                .functionName(functionName)
                .description("Created by the Lambda Java API")
                .code(code)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .role(role)
                .build();

            // Create a Lambda function using a waiter
            CreateFunctionResponse functionResponse =
awsLambda.createFunction(functionRequest);
            GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();
            WaiterResponse<GetFunctionResponse> waiterResponse =
waiter.waitUntilFunctionExists(getFunctionRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            return functionResponse.functionArn();

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }

    /**
     * Retrieves information about an AWS Lambda function.
     *
     * @param awsLambda    an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to retrieve
information about
     */
    public static void getFunction(LambdaClient awsLambda, String functionName) {
        try {
            GetFunctionRequest functionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
```

```
                .build();

            GetFunctionResponse response =
awsLambda.getFunction(functionRequest);
            System.out.println("The runtime of this Lambda function is " +
response.configuration().runtime());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Lists the AWS Lambda functions associated with the current AWS account.
     *
     * @param awsLambda an instance of the {@link LambdaClient} class, which is
used to interact with the AWS Lambda service
     *
     * @throws LambdaException if an error occurs while interacting with the AWS
Lambda service
     */
    public static void listFunctions(LambdaClient awsLambda) {
        try {
            ListFunctionsResponse functionResult = awsLambda.listFunctions();
            List<FunctionConfiguration> list = functionResult.functions();
            for (FunctionConfiguration config : list) {
                System.out.println("The function name is " +
config.functionName());
            }

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Invokes a specific AWS Lambda function.
     *
     * @param awsLambda    an instance of {@link LambdaClient} to interact with
the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to be invoked
     */
```

```
   public static void invokeFunction(LambdaClient awsLambda, String
functionName) {
       InvokeResponse res;
       try {
           // Need a SdkBytes instance for the payload.
           JSONObject jsonObj = new JSONObject();
           jsonObj.put("inputValue", "2000");
           String json = jsonObj.toString();
           SdkBytes payload = SdkBytes.fromUtf8String(json);

           InvokeRequest request = InvokeRequest.builder()
               .functionName(functionName)
               .payload(payload)
               .build();

           res = awsLambda.invoke(request);
           String value = res.payload().asUtf8String();
           System.out.println(value);

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }

   /**
    * Updates the code for an AWS Lambda function.
    *
    * @param awsLambda  the AWS Lambda client
    * @param functionName the name of the Lambda function to update
    * @param bucketName the name of the S3 bucket where the function code is
located
    * @param key the key (file name) of the function code in the S3 bucket
    * @throws LambdaException if there is an error updating the function code
    */
   public static void updateFunctionCode(LambdaClient awsLambda, String
functionName, String bucketName, String key) {
       try {
           LambdaWaiter waiter = awsLambda.waiter();
           UpdateFunctionCodeRequest functionCodeRequest =
UpdateFunctionCodeRequest.builder()
               .functionName(functionName)
               .publish(true)
               .s3Bucket(bucketName)
```

```
                .s3Key(key)
                .build();

            UpdateFunctionCodeResponse response =
awsLambda.updateFunctionCode(functionCodeRequest);
            GetFunctionConfigurationRequest getFunctionConfigRequest =
GetFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .build();

            WaiterResponse<GetFunctionConfigurationResponse> waiterResponse =
waiter
                .waitUntilFunctionUpdated(getFunctionConfigRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            System.out.println("The last modified value is " +
response.lastModified());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Updates the configuration of an AWS Lambda function.
     *
     * @param awsLambda     the {@link LambdaClient} instance to use for the AWS
Lambda operation
     * @param functionName  the name of the AWS Lambda function to update
     * @param handler       the new handler for the AWS Lambda function
     *
     * @throws LambdaException if there is an error while updating the function
configuration
     */
    public static void updateFunctionConfiguration(LambdaClient awsLambda, String
functionName, String handler) {
        try {
            UpdateFunctionConfigurationRequest configurationRequest =
UpdateFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .build();
```

```
            awsLambda.updateFunctionConfiguration(configurationRequest);

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }

    /**
     * Deletes an AWS Lambda function.
     *
     * @param awsLambda     an instance of the {@link LambdaClient} class, which
 is used to interact with the AWS Lambda service
     * @param functionName  the name of the Lambda function to be deleted
     *
     * @throws LambdaException if an error occurs while deleting the Lambda
 function
     */
    public static void deleteLambdaFunction(LambdaClient awsLambda, String
 functionName) {
        try {
            DeleteFunctionRequest request = DeleteFunctionRequest.builder()
                .functionName(functionName)
                .build();

            awsLambda.deleteFunction(request);
            System.out.println("The " + functionName + " function was deleted");

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
}
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Java 2.x API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke

- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

建立 AWS Identity and Access Management (IAM) 角色，授予 Lambda 寫入日誌的許可。

```
    logger.log(`Creating role (${NAME_ROLE_LAMBDA})...`);
    const response = await createRole(NAME_ROLE_LAMBDA);

import { AttachRolePolicyCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} policyArn
 * @param {string} roleName
 */
export const attachRolePolicy = (policyArn, roleName) => {
  const command = new AttachRolePolicyCommand({
    PolicyArn: policyArn,
    RoleName: roleName,
  });

  return client.send(command);
};
```

建立 Lambda 函數並上傳處理常式程式碼。

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

調用具有單一參數的函數並取得結果。

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

更新函數程式碼並設定具有環境變數的 Lambda 環境。

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
```

```
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};

const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  const result = client.send(command);
  waitForFunctionUpdated({ FunctionName: funcName });
  return result;
};
```

列出您帳戶的函數。

```
const listFunctions = () => {
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

刪除 IAM 角色與 Lambda 函數。

```
import { DeleteRoleCommand, IAMClient } from "@aws-sdk/client-iam";

const client = new IAMClient({});

/**
 *
 * @param {string} roleName
 */
```

```
export const deleteRole = (roleName) => {
  const command = new DeleteRoleCommand({ RoleName: roleName });
  return client.send(command);
};

/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- 如需 API 詳細資訊，請參閱《適用於 JavaScript 的 AWS SDK API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Kotlin

適用於 Kotlin 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
suspend fun main(args: Array<String>) {
    val usage = """
        Usage:
```

```
            <functionName> <role> <handler> <bucketName> <updatedBucketName>
<key>

       Where:
           functionName - The name of the AWS Lambda function.
           role - The AWS Identity and Access Management (IAM) service role that
has AWS Lambda permissions.
           handler - The fully qualified method name (for example,
example.Handler::handleRequest).
           bucketName - The Amazon Simple Storage Service (Amazon S3) bucket
name that contains the ZIP or JAR used for the Lambda function's code.
           updatedBucketName - The Amazon S3 bucket name that contains the .zip
or .jar used to update the Lambda function's code.
           key - The Amazon S3 key name that represents the .zip or .jar file
(for example, LambdaHello-1.0-SNAPSHOT.jar).
           """

   if (args.size != 6) {
       println(usage)
       exitProcess(1)
   }

   val functionName = args[0]
   val role = args[1]
   val handler = args[2]
   val bucketName = args[3]
   val updatedBucketName = args[4]
   val key = args[5]

   println("Creating a Lambda function named $functionName.")
   val funArn = createScFunction(functionName, bucketName, key, handler, role)
   println("The AWS Lambda ARN is $funArn")

   // Get a specific Lambda function.
   println("Getting the $functionName AWS Lambda function.")
   getFunction(functionName)

   // List the Lambda functions.
   println("Listing all AWS Lambda functions.")
   listFunctionsSc()

   // Invoke the Lambda function.
   println("*** Invoke the Lambda function.")
   invokeFunctionSc(functionName)
```

```
    // Update the AWS Lambda function code.
    println("*** Update the Lambda function code.")
    updateFunctionCode(functionName, updatedBucketName, key)

    // println("*** Invoke the function again after updating the code.")
    invokeFunctionSc(functionName)

    // Update the AWS Lambda function configuration.
    println("Update the run time of the function.")
    updateFunctionConfiguration(functionName, handler)

    // Delete the AWS Lambda function.
    println("Delete the AWS Lambda function.")
    delFunction(functionName)
}

suspend fun createScFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String,
): String {
    val functionCode =
        FunctionCode {
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
            role = myRole
            runtime = Runtime.Java17
        }

    // Create a Lambda function using a waiter
    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
```

```
            functionName = myFunctionName
        }
        return functionResponse.functionArn.toString()
    }
}

suspend fun getFunction(functionNameVal: String) {
    val functionRequest =
        GetFunctionRequest {
            functionName = functionNameVal
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val response = awsLambda.getFunction(functionRequest)
        println("The runtime of this Lambda function is
 ${response.configuration?.runtime}")
    }
}

suspend fun listFunctionsSc() {
    val request =
        ListFunctionsRequest {
            maxItems = 10
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val response = awsLambda.listFunctions(request)
        response.functions?.forEach { function ->
            println("The function name is ${function.functionName}")
        }
    }
}

suspend fun invokeFunctionSc(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            payload = byteArray
            logType = LogType.Tail
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
```

```
        val res = awsLambda.invoke(request)
        println("The function payload is
 ${res.payload?.toString(Charsets.UTF_8)}")
    }
}

suspend fun updateFunctionCode(
    functionNameVal: String?,
    bucketName: String?,
    key: String?,
) {
    val functionCodeRequest =
        UpdateFunctionCodeRequest {
            functionName = functionNameVal
            publish = true
            s3Bucket = bucketName
            s3Key = key
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val response = awsLambda.updateFunctionCode(functionCodeRequest)
        awsLambda.waitUntilFunctionUpdated {
            functionName = functionNameVal
        }
        println("The last modified value is " + response.lastModified)
    }
}

suspend fun updateFunctionConfiguration(
    functionNameVal: String?,
    handlerVal: String?,
) {
    val configurationRequest =
        UpdateFunctionConfigurationRequest {
            functionName = functionNameVal
            handler = handlerVal
            runtime = Runtime.Java17
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        awsLambda.updateFunctionConfiguration(configurationRequest)
    }
}
```

```
suspend fun delFunction(myFunctionName: String) {
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Kotlin API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
namespace Lambda;

use Aws\S3\S3Client;
use GuzzleHttp\Psr7\Stream;
use Iam\IAMService;
```

```
class GettingStartedWithLambda
{
    public function run()
    {
        echo("\n");
        echo("---------------------------------------\n");
        print("Welcome to the AWS Lambda getting started demo using PHP!\n");
        echo("---------------------------------------\n");

        $clientArgs = [
            'region' => 'us-west-2',
            'version' => 'latest',
            'profile' => 'default',
        ];
        $uniqid = uniqid();

        $iamService = new IAMService();
        $s3client = new S3Client($clientArgs);
        $lambdaService = new LambdaService();

        echo "First, let's create a role to run our Lambda code.\n";
        $roleName = "test-lambda-role-$uniqid";
        $rolePolicyDocument = "{
            \"Version\": \"2012-10-17\",
            \"Statement\": [
                {
                    \"Effect\": \"Allow\",
                    \"Principal\": {
                        \"Service\": \"lambda.amazonaws.com\"
                    },
                    \"Action\": \"sts:AssumeRole\"
                }
            ]
        }";
        $role = $iamService->createRole($roleName, $rolePolicyDocument);
        echo "Created role {$role['RoleName']}.\n";

        $iamService->attachRolePolicy(
            $role['RoleName'],
            "arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole"
        );
        echo "Attached the AWSLambdaBasicExecutionRole to {$role['RoleName']}.
\n";
```

```
        echo "\nNow let's create an S3 bucket and upload our Lambda code there.
\n";
        $bucketName = "amzn-s3-demo-bucket-$uniqid";
        $s3client->createBucket([
            'Bucket' => $bucketName,
        ]);
        echo "Created bucket $bucketName.\n";

        $functionName = "doc_example_lambda_$uniqid";
        $codeBasic = __DIR__ . "/lambda_handler_basic.zip";
        $handler = "lambda_handler_basic";
        $file = file_get_contents($codeBasic);
        $s3client->putObject([
            'Bucket' => $bucketName,
            'Key' => $functionName,
            'Body' => $file,
        ]);
        echo "Uploaded the Lambda code.\n";

        $createLambdaFunction = $lambdaService->createFunction($functionName,
 $role, $bucketName, $handler);
        // Wait until the function has finished being created.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['State'] == "Pending");
        echo "Created Lambda function {$getLambdaFunction['Configuration']
['FunctionName']}.\n";

        sleep(1);

        echo "\nOk, let's invoke that Lambda code.\n";
        $basicParams = [
            'action' => 'increment',
            'number' => 3,
        ];
        /** @var Stream $invokeFunction */
        $invokeFunction = $lambdaService->invoke($functionName, $basicParams)
['Payload'];
        $result = json_decode($invokeFunction->getContents())->result;
        echo "After invoking the Lambda code with the input of
 {$basicParams['number']} we received $result.\n";

        echo "\nSince that's working, let's update the Lambda code.\n";
```

```
        $codeCalculator = "lambda_handler_calculator.zip";
        $handlerCalculator = "lambda_handler_calculator";
        echo "First, put the new code into the S3 bucket.\n";
        $file = file_get_contents($codeCalculator);
        $s3client->putObject([
            'Bucket' => $bucketName,
            'Key' => $functionName,
            'Body' => $file,
        ]);
        echo "New code uploaded.\n";

        $lambdaService->updateFunctionCode($functionName, $bucketName,
 $functionName);
        // Wait for the Lambda code to finish updating.
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "New Lambda code uploaded.\n";

        $environment = [
            'Variable' => ['Variables' => ['LOG_LEVEL' => 'DEBUG']],
        ];
        $lambdaService->updateFunctionConfiguration($functionName,
 $handlerCalculator, $environment);
        do {
            $getLambdaFunction = $lambdaService-
>getFunction($createLambdaFunction['FunctionName']);
        } while ($getLambdaFunction['Configuration']['LastUpdateStatus'] !==
 "Successful");
        echo "Lambda code updated with new handler and a LOG_LEVEL of DEBUG for
 more information.\n";

        echo "Invoke the new code with some new data.\n";
        $calculatorParams = [
            'action' => 'plus',
            'x' => 5,
            'y' => 4,
        ];
        $invokeFunction = $lambdaService->invoke($functionName,
 $calculatorParams, "Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
```

```
        echo "Indeed, {$calculatorParams['x']} + {$calculatorParams['y']} does
equal $result.\n";
        echo "Here's the extra debug info: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nBut what happens if you try to divide by zero?\n";
        $divZeroParams = [
            'action' => 'divide',
            'x' => 5,
            'y' => 0,
        ];
        $invokeFunction = $lambdaService->invoke($functionName, $divZeroParams,
"Tail");
        $result = json_decode($invokeFunction['Payload']->getContents())->result;
        echo "You get a |$result| result.\n";
        echo "And an error message: ";
        echo base64_decode($invokeFunction['LogResult']) . "\n";

        echo "\nHere's all the Lambda functions you have in this Region:\n";
        $listLambdaFunctions = $lambdaService->listFunctions(5);
        $allLambdaFunctions = $listLambdaFunctions['Functions'];
        $next = $listLambdaFunctions->get('NextMarker');
        while ($next != false) {
            $listLambdaFunctions = $lambdaService->listFunctions(5, $next);
            $next = $listLambdaFunctions->get('NextMarker');
            $allLambdaFunctions = array_merge($allLambdaFunctions,
$listLambdaFunctions['Functions']);
        }
        foreach ($allLambdaFunctions as $function) {
            echo "{$function['FunctionName']}\n";
        }

        echo "\n\nAnd don't forget to clean up your data!\n";

        $lambdaService->deleteFunction($functionName);
        echo "Deleted Lambda function.\n";
        $iamService->deleteRole($role['RoleName']);
        echo "Deleted Role.\n";
        $deleteObjects = $s3client->listObjectsV2([
            'Bucket' => $bucketName,
        ]);
        $deleteObjects = $s3client->deleteObjects([
            'Bucket' => $bucketName,
            'Delete' => [
```

```
                'Objects' => $deleteObjects['Contents'],
            ]
        ]);
        echo "Deleted all objects from the S3 bucket.\n";
        $s3client->deleteBucket(['Bucket' => $bucketName]);
        echo "Deleted the bucket.\n";
    }
}
```

- 如需 API 詳細資訊，請參閱《適用於 PHP 的 AWS SDK API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Python

適用於 Python (Boto3) 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

定義增量一個數字的 Lambda 處理常式。

```
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)


def lambda_handler(event, context):
    """
```

```
    Accepts an action and a single number, performs the specified action on the
 number,
    and returns the result. The only allowable action is 'increment'.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the action.
    """
    result = None
    action = event.get("action")
    if action == "increment":
        result = event.get("number", 0) + 1
        logger.info("Calculated result of %s", result)
    else:
        logger.error("%s is not a valid action.", action)

    response = {"result": result}
    return response
```

定義可執行算術運算的第二個 Lambda 處理常式。

```
import logging
import os


logger = logging.getLogger()

# Define a list of Python lambda functions that are called by this AWS Lambda
 function.
ACTIONS = {
    "plus": lambda x, y: x + y,
    "minus": lambda x, y: x - y,
    "times": lambda x, y: x * y,
    "divided-by": lambda x, y: x / y,
}


def lambda_handler(event, context):
```

```
    """
    Accepts an action and two numbers, performs the specified action on the
 numbers,
    and returns the result.

    :param event: The event dict that contains the parameters sent when the
 function
                  is invoked.
    :param context: The context in which the function is called.
    :return: The result of the specified action.
    """
    # Set the log level based on a variable configured in the Lambda environment.
    logger.setLevel(os.environ.get("LOG_LEVEL", logging.INFO))
    logger.debug("Event: %s", event)

    action = event.get("action")
    func = ACTIONS.get(action)
    x = event.get("x")
    y = event.get("y")
    result = None
    try:
        if func is not None and x is not None and y is not None:
            result = func(x, y)
            logger.info("%s %s %s is %s", x, action, y, result)
        else:
            logger.error("I can't calculate %s %s %s.", x, action, y)
    except ZeroDivisionError:
        logger.warning("I can't divide %s by 0!", x)

    response = {"result": result}
    return response
```

建立可包裝 Lambda 動作的函數。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource
```

```
    @staticmethod
    def create_deployment_package(source_file, destination_file):
        """
        Creates a Lambda deployment package in .zip format in an in-memory
buffer. This
        buffer can be passed directly to Lambda when creating the function.

        :param source_file: The name of the file that contains the Lambda handler
                            function.
        :param destination_file: The name to give the file when it's deployed to
Lambda.
        :return: The deployment package.
        """
        buffer = io.BytesIO()
        with zipfile.ZipFile(buffer, "w") as zipped:
            zipped.write(source_file, destination_file)
        buffer.seek(0)
        return buffer.read()

    def get_iam_role(self, iam_role_name):
        """
        Get an AWS Identity and Access Management (IAM) role.

        :param iam_role_name: The name of the role to retrieve.
        :return: The IAM role.
        """
        role = None
        try:
            temp_role = self.iam_resource.Role(iam_role_name)
            temp_role.load()
            role = temp_role
            logger.info("Got IAM role %s", role.name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "NoSuchEntity":
                logger.info("IAM role %s does not exist.", iam_role_name)
            else:
                logger.error(
                    "Couldn't get IAM role %s. Here's why: %s: %s",
                    iam_role_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return role
```

```
    def create_iam_role_for_lambda(self, iam_role_name):
        """
        Creates an IAM role that grants the Lambda function basic permissions. If
a
        role with the specified name already exists, it is used for the demo.

        :param iam_role_name: The name of the role to create.
        :return: The role and a value that indicates whether the role is newly
created.
        """
        role = self.get_iam_role(iam_role_name)
        if role is not None:
            return role, False

        lambda_assume_role_policy = {
            "Version":"2012-10-17",
            "Statement": [
                {
                    "Effect": "Allow",
                    "Principal": {"Service": "lambda.amazonaws.com"},
                    "Action": "sts:AssumeRole",
                }
            ],
        }
        policy_arn = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"

        try:
            role = self.iam_resource.create_role(
                RoleName=iam_role_name,
                AssumeRolePolicyDocument=json.dumps(lambda_assume_role_policy),
            )
            logger.info("Created role %s.", role.name)
            role.attach_policy(PolicyArn=policy_arn)
            logger.info("Attached basic execution policy to role %s.", role.name)
        except ClientError as error:
            if error.response["Error"]["Code"] == "EntityAlreadyExists":
                role = self.iam_resource.Role(iam_role_name)
                logger.warning("The role %s already exists. Using it.",
iam_role_name)
            else:
                logger.exception(
                    "Couldn't create role %s or attach policy %s.",
```

```
                    iam_role_name,
                    policy_arn,
                )
                raise

        return role, True

    def get_function(self, function_name):
        """
        Gets data about a Lambda function.

        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return response


    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
    ):
        """
        Deploys a Lambda function.

        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
This
                             must include the file name and the function name.
        :param iam_role: The IAM role to use for the function.
```

```
        :param deployment_package: The deployment package that contains the
function
                                   code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.9",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn


    def delete_function(self, function_name):
        """
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise


    def invoke_function(self, function_name, function_params, get_log=False):
```

```
        """
        Invokes a Lambda function.

        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This
dict
                                is serialized to JSON before it is sent to
Lambda.
        :param get_log: When true, the last 4 KB of the execution log are
included in
                        the response.
        :return: The response from the function invocation.
        """
        try:
            response = self.lambda_client.invoke(
                FunctionName=function_name,
                Payload=json.dumps(function_params),
                LogType="Tail" if get_log else "None",
            )
            logger.info("Invoked function %s.", function_name)
        except ClientError:
            logger.exception("Couldn't invoke function %s.", function_name)
            raise
        return response


    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that
contains
        the code for the function.

        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes
in
                                   .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
        except ClientError as err:
            logger.error(
```

```
                "Couldn't update function %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def update_function_configuration(self, function_name, env_vars):
        """
        Updates the environment variables for a Lambda function.

        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
                    print(func["FunctionName"])
                    desc = func.get("Description")
```

```
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
        except ClientError as err:
            logger.error(
                "Couldn't list functions. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

建立可執行該案例的函數。

```
class UpdateFunctionWaiter(CustomWaiter):
    """A custom waiter that waits until a function is successfully updated."""

    def __init__(self, client):
        super().__init__(
            "UpdateSuccess",
            "GetFunction",
            "Configuration.LastUpdateStatus",
            {"Successful": WaitState.SUCCESS, "Failed": WaitState.FAILURE},
            client,
        )

    def wait(self, function_name):
        self._wait(FunctionName=function_name)


def run_scenario(lambda_client, iam_resource, basic_file, calculator_file,
 lambda_name):
    """
    Runs the scenario.

    :param lambda_client: A Boto3 Lambda client.
    :param iam_resource: A Boto3 IAM resource.
    :param basic_file: The name of the file that contains the basic Lambda
 handler.
```

```
    :param calculator_file: The name of the file that contains the calculator
Lambda handler.
    :param lambda_name: The name to give resources created for the scenario, such
as the
                        IAM role and the Lambda function.
    """
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

    print("-" * 88)
    print("Welcome to the AWS Lambda getting started with functions demo.")
    print("-" * 88)

    wrapper = LambdaWrapper(lambda_client, iam_resource)

    print("Checking for IAM role for Lambda...")
    iam_role, should_wait = wrapper.create_iam_role_for_lambda(lambda_name)
    if should_wait:
        logger.info("Giving AWS time to create resources...")
        wait(10)

    print(f"Looking for function {lambda_name}...")
    function = wrapper.get_function(lambda_name)
    if function is None:
        print("Zipping the Python script into a deployment package...")
        deployment_package = wrapper.create_deployment_package(
            basic_file, f"{lambda_name}.py"
        )
        print(f"...and creating the {lambda_name} Lambda function.")
        wrapper.create_function(
            lambda_name, f"{lambda_name}.lambda_handler", iam_role,
deployment_package
        )
    else:
        print(f"Function {lambda_name} already exists.")
    print("-" * 88)

    print(f"Let's invoke {lambda_name}. This function increments a number.")
    action_params = {
        "action": "increment",
        "number": q.ask("Give me a number to increment: ", q.is_int),
    }
    print(f"Invoking {lambda_name}...")
    response = wrapper.invoke_function(lambda_name, action_params)
    print(
```

```
        f"Incrementing {action_params['number']} resulted in "
        f"{json.load(response['Payload'])}"
    )
    print("-" * 88)

    print(f"Let's update the function to an arithmetic calculator.")
    q.ask("Press Enter when you're ready.")
    print("Creating a new deployment package...")
    deployment_package = wrapper.create_deployment_package(
        calculator_file, f"{lambda_name}.py"
    )
    print(f"...and updating the {lambda_name} Lambda function.")
    update_waiter = UpdateFunctionWaiter(lambda_client)
    wrapper.update_function_code(lambda_name, deployment_package)
    update_waiter.wait(lambda_name)
    print(f"This function uses an environment variable to control logging
level.")
    print(f"Let's set it to DEBUG to get the most logging.")
    wrapper.update_function_configuration(
        lambda_name, {"LOG_LEVEL": logging.getLevelName(logging.DEBUG)}
    )

    actions = ["plus", "minus", "times", "divided-by"]
    want_invoke = True
    while want_invoke:
        print(f"Let's invoke {lambda_name}. You can invoke these actions:")
        for index, action in enumerate(actions):
            print(f"{index + 1}: {action}")
        action_params = {}
        action_index = q.ask(
            "Enter the number of the action you want to take: ",
            q.is_int,
            q.in_range(1, len(actions)),
        )
        action_params["action"] = actions[action_index - 1]
        print(f"You've chosen to invoke 'x {action_params['action']} y'.")
        action_params["x"] = q.ask("Enter a value for x: ", q.is_int)
        action_params["y"] = q.ask("Enter a value for y: ", q.is_int)
        print(f"Invoking {lambda_name}...")
        response = wrapper.invoke_function(lambda_name, action_params, True)
        print(
            f"Calculating {action_params['x']} {action_params['action']}
{action_params['y']} "
            f"resulted in {json.load(response['Payload'])}"
```

```
        )
        q.ask("Press Enter to see the logs from the call.")
        print(base64.b64decode(response["LogResult"]).decode())
        want_invoke = q.ask("That was fun. Shall we do it again? (y/n) ",
 q.is_yesno)
    print("-" * 88)

    if q.ask(
        "Do you want to list all of the functions in your account? (y/n) ",
 q.is_yesno
    ):
        wrapper.list_functions()
    print("-" * 88)

    if q.ask("Ready to delete the function and role? (y/n) ", q.is_yesno):
        for policy in iam_role.attached_policies.all():
            policy.detach_role(RoleName=iam_role.name)
        iam_role.delete()
        print(f"Deleted role {lambda_name}.")
        wrapper.delete_function(lambda_name)
        print(f"Deleted function {lambda_name}.")

    print("\nThanks for watching!")
    print("-" * 88)


if __name__ == "__main__":
    try:
        run_scenario(
            boto3.client("lambda"),
            boto3.resource("iam"),
            "lambda_handler_basic.py",
            "lambda_handler_calculator.py",
            "doc_example_lambda_calculator",
        )
    except Exception:
        logging.exception("Something went wrong with the demo!")
```

- 如需 API 詳細資訊，請參閱下列《適用於 Python (Boto3) 的AWS 開發套件 API 參考》中的下列主題。
  - CreateFunction

- DeleteFunction
- GetFunction
- Invoke
- ListFunctions
- UpdateFunctionCode
- UpdateFunctionConfiguration

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

為能夠寫入日誌的 Lambda 函數設定先決條件 IAM 許可。

```
# Get an AWS Identity and Access Management (IAM) role.
#
# @param iam_role_name: The name of the role to retrieve.
# @param action: Whether to create or destroy the IAM apparatus.
# @return: The IAM role.
def manage_iam(iam_role_name, action)
  case action
  when 'create'
    create_iam_role(iam_role_name)
  when 'destroy'
    destroy_iam_role(iam_role_name)
  else
    raise "Incorrect action provided. Must provide 'create' or 'destroy'"
  end
end

private

def create_iam_role(iam_role_name)
  role_policy = {
```

```
      'Version': '2012-10-17',
      'Statement': [
        {
          'Effect': 'Allow',
          'Principal': { 'Service': 'lambda.amazonaws.com' },
          'Action': 'sts:AssumeRole'
        }
      ]
    }
    role = @iam_client.create_role(
      role_name: iam_role_name,
      assume_role_policy_document: role_policy.to_json
    )
    @iam_client.attach_role_policy(
      {
        policy_arn: 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole',
        role_name: iam_role_name
      }
    )
    wait_for_role_to_exist(iam_role_name)
    @logger.debug("Successfully created IAM role: #{role['role']['arn']}")
    sleep(10)
    [role, role_policy.to_json]
  end

  def destroy_iam_role(iam_role_name)
    @iam_client.detach_role_policy(
      {
        policy_arn: 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole',
        role_name: iam_role_name
      }
    )
    @iam_client.delete_role(role_name: iam_role_name)
    @logger.debug("Detached policy & deleted IAM role: #{iam_role_name}")
  end

  def wait_for_role_to_exist(iam_role_name)
    @iam_client.wait_until(:role_exists, { role_name: iam_role_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  end
```

定義 Lambda 處理常式，該處理常式以作為調用參數提供的數字遞增。

```
require 'logger'

# A function that increments a whole number by one (1) and logs the result.
# Requires a manually-provided runtime parameter, 'number', which must be Int
#
# @param event [Hash] Parameters sent when the function is invoked
# @param context [Hash] Methods and properties that provide information
# about the invocation, function, and execution environment.
# @return incremented_number [String] The incremented number.
def lambda_handler(event:, context:)
  logger = Logger.new($stdout)
  log_level = ENV['LOG_LEVEL']
  logger.level = case log_level
                 when 'debug'
                   Logger::DEBUG
                 when 'info'
                   Logger::INFO
                 else
                   Logger::ERROR
                 end
  logger.debug('This is a debug log message.')
  logger.info('This is an info log message. Code executed successfully!')
  number = event['number'].to_i
  incremented_number = number + 1
  logger.info("You provided #{number.round} and it was incremented to
 #{incremented_number.round}")
  incremented_number.round.to_s
end
```

將您的 Lambda 函數壓縮到部署套件中。

```
  # Creates a Lambda deployment package in .zip format.
  #
  # @param source_file: The name of the object, without suffix, for the Lambda
 file and zip.
  # @return: The deployment package.
  def create_deployment_package(source_file)
    Dir.chdir(File.dirname(__FILE__))
```

```
    if File.exist?('lambda_function.zip')
      File.delete('lambda_function.zip')
      @logger.debug('Deleting old zip: lambda_function.zip')
    end
    Zip::File.open('lambda_function.zip', create: true) do |zipfile|
      zipfile.add('lambda_function.rb', "#{source_file}.rb")
    end
    @logger.debug("Zipping #{source_file}.rb into: lambda_function.zip.")
    File.read('lambda_function.zip').to_s
  rescue StandardError => e
    @logger.error("There was an error creating deployment package:\n
#{e.message}")
  end
```

建立新 Lambda 函數。

```
  # Deploys a Lambda function.
  #
  # @param function_name: The name of the Lambda function.
  # @param handler_name: The fully qualified name of the handler function.
  # @param role_arn: The IAM role to use for the function.
  # @param deployment_package: The deployment package that contains the function
code in .zip format.
  # @return: The Amazon Resource Name (ARN) of the newly created function.
  def create_function(function_name, handler_name, role_arn, deployment_package)
    response = @lambda_client.create_function({
                                                role: role_arn.to_s,
                                                function_name: function_name,
                                                handler: handler_name,
                                                runtime: 'ruby2.7',
                                                code: {
                                                  zip_file: deployment_package
                                                },
                                                environment: {
                                                  variables: {
                                                    'LOG_LEVEL' => 'info'
                                                  }
                                                }
                                              })
    @lambda_client.wait_until(:function_active_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
```

```
      w.delay = 5
    end
    response
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error creating #{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

使用選用的執行階段參數調用 Lambda 函數。

```
  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name }
    params[:payload] = payload unless payload.nil?
    @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
#{e.message}")
  end
```

更新 Lambda 函數的組態以注入新的環境變數。

```
  # Updates the environment variables for a Lambda function.
  # @param function_name: The name of the function to update.
  # @param log_level: The log level of the function.
  # @return: Data about the update, including the status.
  def update_function_configuration(function_name, log_level)
    @lambda_client.update_function_configuration({
                                                   function_name: function_name,
                                                   environment: {
                                                     variables: {
                                                       'LOG_LEVEL' => log_level
                                                     }
                                                   }
                                                 })
```

```
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

使用包含不同程式碼的不同部署套件來更新 Lambda 函數的程式碼。

```
  # Updates the code for a Lambda function by submitting a .zip archive that
contains
  # the code for the function.
  #
  # @param function_name: The name of the function to update.
  # @param deployment_package: The function code to update, packaged as bytes in
  #                            .zip format.
  # @return: Data about the update, including the status.
  def update_function_code(function_name, deployment_package)
    @lambda_client.update_function_code(
      function_name: function_name,
      zip_file: deployment_package
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating function code for:
#{function_name}:\n #{e.message}")
    nil
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
  end
```

使用內建分頁程式列出所有現有的 Lambda 函數。

```
# Lists the Lambda functions for the current account.
def list_functions
  functions = []
  @lambda_client.list_functions.each do |response|
    response['functions'].each do |function|
      functions.append(function['function_name'])
    end
  end
  functions
rescue Aws::Lambda::Errors::ServiceException => e
  @logger.error("There was an error listing functions:\n #{e.message}")
end
```

刪除特定 Lambda 函數。

```
# Deletes a Lambda function.
# @param function_name: The name of the function to delete.
def delete_function(function_name)
  print "Deleting function: #{function_name}..."
  @lambda_client.delete_function(
    function_name: function_name
  )
  print 'Done!'.green
rescue Aws::Lambda::Errors::ServiceException => e
  @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
end
```

- 如需 API 詳細資訊，請參閱《適用於 Ruby 的 AWS SDK API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

在此案例中使用具有相依項的 Cargo.toml。

```
[package]
name = "lambda-code-examples"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/
reference/manifest.html

[dependencies]
aws-config = { version = "1.0.1", features = ["behavior-version-latest"] }
aws-sdk-ec2 = { version = "1.3.0" }
aws-sdk-iam = { version = "1.3.0" }
aws-sdk-lambda = { version = "1.3.0" }
aws-sdk-s3 = { version = "1.4.0" }
aws-smithy-types = { version = "1.0.1" }
aws-types = { version = "1.0.1" }
clap = { version = "4.4", features = ["derive"] }
tokio = { version = "1.20.1", features = ["full"] }
tracing-subscriber = { version = "0.3.15", features = ["env-filter"] }
tracing = "0.1.37"
serde_json = "1.0.94"
anyhow = "1.0.71"
uuid = { version = "1.3.3", features = ["v4"] }
lambda_runtime = "0.8.0"
serde = "1.0.164"
```

一個公用程式集合，可簡化此案例的 Lambda 呼叫。此檔案是套件中的 src/ations.rs。

```
use anyhow::anyhow;
use aws_sdk_iam::operation::{create_role::CreateRoleError,
 delete_role::DeleteRoleOutput};
use aws_sdk_lambda::{
    operation::{
        delete_function::DeleteFunctionOutput, get_function::GetFunctionOutput,
        invoke::InvokeOutput, list_functions::ListFunctionsOutput,
        update_function_code::UpdateFunctionCodeOutput,
        update_function_configuration::UpdateFunctionConfigurationOutput,
    },
    primitives::ByteStream,
    types::{Environment, FunctionCode, LastUpdateStatus, State},
};
use aws_sdk_s3::{
    error::ErrorMetadata,
    operation::{delete_bucket::DeleteBucketOutput,
 delete_object::DeleteObjectOutput},
    types::CreateBucketConfiguration,
};
use aws_smithy_types::Blob;
use serde::{ser::SerializeMap, Serialize};
use std::{fmt::Display, path::PathBuf, str::FromStr, time::Duration};
use tracing::{debug, info, warn};

/* Operation describes  */
#[derive(Clone, Copy, Debug, Serialize)]
pub enum Operation {
    #[serde(rename = "plus")]
    Plus,
    #[serde(rename = "minus")]
    Minus,
    #[serde(rename = "times")]
    Times,
    #[serde(rename = "divided-by")]
    DividedBy,
}

impl FromStr for Operation {
    type Err = anyhow::Error;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        match s {
            "plus" => Ok(Operation::Plus),
            "minus" => Ok(Operation::Minus),
```

```
            "times" => Ok(Operation::Times),
            "divided-by" => Ok(Operation::DividedBy),
            _ => Err(anyhow!("Unknown operation {s}")),
        }
    }
}

impl Display for Operation {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Operation::Plus => write!(f, "plus"),
            Operation::Minus => write!(f, "minus"),
            Operation::Times => write!(f, "times"),
            Operation::DividedBy => write!(f, "divided-by"),
        }
    }
}

/**
 * InvokeArgs will be serialized as JSON and sent to the AWS Lambda handler.
 */
#[derive(Debug)]
pub enum InvokeArgs {
    Increment(i32),
    Arithmetic(Operation, i32, i32),
}

impl Serialize for InvokeArgs {
    fn serialize<S>(&self, serializer: S) -> Result<S::Ok, S::Error>
    where
        S: serde::Serializer,
    {
        match self {
            InvokeArgs::Increment(i) => serializer.serialize_i32(*i),
            InvokeArgs::Arithmetic(o, i, j) => {
                let mut map: S::SerializeMap =
 serializer.serialize_map(Some(3))?;
                map.serialize_key(&"op".to_string())?;
                map.serialize_value(&o.to_string())?;
                map.serialize_key(&"i".to_string())?;
                map.serialize_value(&i)?;
                map.serialize_key(&"j".to_string())?;
                map.serialize_value(&j)?;
                map.end()
```

```
            }
        }
    }
}

/** A policy document allowing Lambda to execute this function on the account's
 behalf. */
const ROLE_POLICY_DOCUMENT: &str = r#"{
    "Version":"2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": { "Service": "lambda.amazonaws.com" },
            "Action": "sts:AssumeRole"
        }
    ]
}"#;

/**
 * A LambdaManager gathers all the resources necessary to run the Lambda example
 scenario.
 * This includes instantiated aws_sdk clients and details of resource names.
 */
pub struct LambdaManager {
    iam_client: aws_sdk_iam::Client,
    lambda_client: aws_sdk_lambda::Client,
    s3_client: aws_sdk_s3::Client,
    lambda_name: String,
    role_name: String,
    bucket: String,
    own_bucket: bool,
}

// These unit type structs provide nominal typing on top of String parameters for
 LambdaManager::new
pub struct LambdaName(pub String);
pub struct RoleName(pub String);
pub struct Bucket(pub String);
pub struct OwnBucket(pub bool);

impl LambdaManager {
    pub fn new(
        iam_client: aws_sdk_iam::Client,
        lambda_client: aws_sdk_lambda::Client,
```

```
        s3_client: aws_sdk_s3::Client,
        lambda_name: LambdaName,
        role_name: RoleName,
        bucket: Bucket,
        own_bucket: OwnBucket,
    ) -> Self {
        Self {
            iam_client,
            lambda_client,
            s3_client,
            lambda_name: lambda_name.0,
            role_name: role_name.0,
            bucket: bucket.0,
            own_bucket: own_bucket.0,
        }
    }

    /**
     * Load the AWS configuration from the environment.
     * Look up lambda_name and bucket if none are given, or generate a random
name if not present in the environment.
     * If the bucket name is provided, the caller needs to have created the
bucket.
     * If the bucket name is generated, it will be created.
     */
    pub async fn load_from_env(lambda_name: Option<String>, bucket:
Option<String>) -> Self {
        let sdk_config = aws_config::load_from_env().await;
        let lambda_name = LambdaName(lambda_name.unwrap_or_else(|| {
            std::env::var("LAMBDA_NAME").unwrap_or_else(|_|
"rust_lambda_example".to_string())
        }));
        let role_name = RoleName(format!("{}_role", lambda_name.0));
        let (bucket, own_bucket) =
            match bucket {
                Some(bucket) => (Bucket(bucket), false),
                None => (
                    Bucket(std::env::var("LAMBDA_BUCKET").unwrap_or_else(|_| {
                        format!("rust-lambda-example-{}", uuid::Uuid::new_v4())
                    })),
                    true,
                ),
            };
```

```
        let s3_client = aws_sdk_s3::Client::new(&sdk_config);

        if own_bucket {
            info!("Creating bucket for demo: {}", bucket.0);
            s3_client
                .create_bucket()
                .bucket(bucket.0.clone())
                .create_bucket_configuration(
                    CreateBucketConfiguration::builder()

  .location_constraint(aws_sdk_s3::types::BucketLocationConstraint::from(
                            sdk_config.region().unwrap().as_ref(),
                        ))
                        .build(),
                )
                .send()
                .await
                .unwrap();
        }

        Self::new(
            aws_sdk_iam::Client::new(&sdk_config),
            aws_sdk_lambda::Client::new(&sdk_config),
            s3_client,
            lambda_name,
            role_name,
            bucket,
            OwnBucket(own_bucket),
        )
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;
```

```
        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }

    /**
     * Create a function, uploading from a zip file.
     */
    pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
anyhow::Error> {
        let code = self.prepare_function(zip_file, None).await?;

        let key = code.s3_key().unwrap().to_string();

        let role = self.create_role().await.map_err(|e| anyhow!(e))?;

        info!("Created iam role, waiting 15s for it to become active");
        tokio::time::sleep(Duration::from_secs(15)).await;

        info!("Creating lambda function {}", self.lambda_name);
        let _ = self
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
            .code(code)
            .role(role.arn())
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;
```

```
        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
            .send()
            .await?;

        Ok(key)
    }

    /**
     * Create an IAM execution role for the managed Lambda function.
     * If the role already exists, use that instead.
     */
    async fn create_role(&self) -> Result<aws_sdk_iam::types::Role,
CreateRoleError> {
        info!("Creating execution role for function");
        let get_role = self
            .iam_client
            .get_role()
            .role_name(self.role_name.clone())
            .send()
            .await;
        if let Ok(get_role) = get_role {
            if let Some(role) = get_role.role {
                return Ok(role);
            }
        }

        let create_role = self
            .iam_client
            .create_role()
            .role_name(self.role_name.clone())
            .assume_role_policy_document(ROLE_POLICY_DOCUMENT)
            .send()
            .await;

        match create_role {
            Ok(create_role) => match create_role.role {
                Some(role) => Ok(role),
                None => Err(CreateRoleError::generic(
                    ErrorMetadata::builder()
```

```
                        .message("CreateRole returned empty success")
                        .build(),
                )),
            },
            Err(err) => Err(err.into_service_error()),
        }
    }

    /**
     * Poll `is_function_ready` with a 1-second delay. It returns when the
function is ready or when there's an error checking the function's state.
     */
    pub async fn wait_for_function_ready(&self) -> Result<(), anyhow::Error> {
        info!("Waiting for function");
        while !self.is_function_ready(None).await? {
            info!("Function is not ready, sleeping 1s");
            tokio::time::sleep(Duration::from_secs(1)).await;
        }
        Ok(())
    }

    /**
     * Check if a Lambda function is ready to be invoked.
     * A Lambda function is ready for this scenario when its state is active and
its LastUpdateStatus is Successful.
     * Additionally, if a sha256 is provided, the function must have that as its
current code hash.
     * Any missing properties or failed requests will be reported as an Err.
     */
    async fn is_function_ready(
        &self,
        expected_code_sha256: Option<&str>,
    ) -> Result<bool, anyhow::Error> {
        match self.get_function().await {
            Ok(func) => {
                if let Some(config) = func.configuration() {
                    if let Some(state) = config.state() {
                        info!(?state, "Checking if function is active");
                        if !matches!(state, State::Active) {
                            return Ok(false);
                        }
                    }
                    match config.last_update_status() {
                        Some(last_update_status) => {
```

```
                            info!(?last_update_status, "Checking if function is
ready");
                            match last_update_status {
                                LastUpdateStatus::Successful => {
                                    // continue
                                }
                                LastUpdateStatus::Failed |
LastUpdateStatus::InProgress => {
                                    return Ok(false);
                                }
                                unknown => {
                                    warn!(
                                        status_variant = unknown.as_str(),
                                        "LastUpdateStatus unknown"
                                    );
                                    return Err(anyhow!(
                                        "Unknown LastUpdateStatus, fn config is
{config:?}"
                                    ));
                                }
                            }
                        }
                        None => {
                            warn!("Missing last update status");
                            return Ok(false);
                        }
                    };
                    if expected_code_sha256.is_none() {
                        return Ok(true);
                    }
                    if let Some(code_sha256) = config.code_sha256() {
                        return Ok(code_sha256 ==
expected_code_sha256.unwrap_or_default());
                    }
                }
            }
            Err(e) => {
                warn!(?e, "Could not get function while waiting");
            }
        }
        Ok(false)
    }

    /** Get the Lambda function with this Manager's name. */
```

```
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
            .get_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

    /** List all Lambda functions in the current Region. */
    pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
        info!("Listing lambda functions");
        self.lambda_client
            .list_functions()
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

    /** Invoke the lambda function using calculator InvokeArgs. */
    pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
        debug!(?payload, "Sending payload");
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
            .payload(Blob::new(payload))
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

    /** Given a Path to a zip file, update the function's code and wait for the
update to finish. */
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
```

```
        let function_code = self.prepare_function(zip_file, Some(key)).await?;

        info!("Updating code for {}", self.lambda_name);
        let update = self
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(update)
    }

    /** Update the environment for a function. */
    pub async fn update_function_configuration(
        &self,
        environment: Environment,
    ) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
        info!(
            ?environment,
            "Updating environment for {}", self.lambda_name
        );
        let updated = self
            .lambda_client
            .update_function_configuration()
            .function_name(self.lambda_name.clone())
            .environment(environment)
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(updated)
    }

    /** Delete a function and its role, and if possible or necessary, its
 associated code object and bucket. */
    pub async fn delete_function(
```

```
        &self,
        location: Option<String>,
    ) -> (
        Result<DeleteFunctionOutput, anyhow::Error>,
        Result<DeleteRoleOutput, anyhow::Error>,
        Option<Result<DeleteObjectOutput, anyhow::Error>>,
    ) {
        info!("Deleting lambda function {}", self.lambda_name);
        let delete_function = self
            .lambda_client
            .delete_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        info!("Deleting iam role {}", self.role_name);
        let delete_role = self
            .iam_client
            .delete_role()
            .role_name(self.role_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from);

        let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
            if let Some(location) = location {
                info!("Deleting object {location}");
                Some(
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                info!(?location, "Skipping delete object");
                None
            };

        (delete_function, delete_role, delete_object)
    }
```

```
    pub async fn cleanup(
        &self,
        location: Option<String>,
    ) -> (
        (
            Result<DeleteFunctionOutput, anyhow::Error>,
            Result<DeleteRoleOutput, anyhow::Error>,
            Option<Result<DeleteObjectOutput, anyhow::Error>>,
        ),
        Option<Result<DeleteBucketOutput, anyhow::Error>>,
    ) {
        let delete_function = self.delete_function(location).await;

        let delete_bucket = if self.own_bucket {
            info!("Deleting bucket {}", self.bucket);
            if delete_function.2.is_none() ||
 delete_function.2.as_ref().unwrap().is_ok() {
                Some(
                    self.s3_client
                        .delete_bucket()
                        .bucket(self.bucket.clone())
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                None
            }
        } else {
            info!("No bucket to clean up");
            None
        };

        (delete_function, delete_bucket)
    }
}

/**
 * Testing occurs primarily as an integration test running the `scenario` bin
 successfully.
 * Each action relies deeply on the internal workings and state of Amazon Simple
 Storage Service (Amazon S3), Lambda, and IAM working together.
 * It is therefore infeasible to mock the clients to test the individual actions.
```

```
 */
#[cfg(test)]
mod test {
    use super::{InvokeArgs, Operation};
    use serde_json::json;

    /** Make sure that the JSON output of serializing InvokeArgs is what's
 expected by the calculator. */
    #[test]
    fn test_serialize() {
        assert_eq!(json!(InvokeArgs::Increment(5)), 5);
        assert_eq!(
            json!(InvokeArgs::Arithmetic(Operation::Plus, 5, 7)).to_string(),
            r#"{"op":"plus","i":5,"j":7}"#.to_string(),
        );
    }
}
```

從前端到後端執行該案例的二進位檔案，使用命令列旗標來控制某些行為。此檔案是套件中的 src/bin/scenario.rs。

```

/*
## Service actions

Service actions wrap the SDK call, taking a client and any specific parameters
 necessary for the call.

* CreateFunction
* GetFunction
* ListFunctions
* Invoke
* UpdateFunctionCode
* UpdateFunctionConfiguration
* DeleteFunction

## Scenario
A scenario runs at a command prompt and prints output to the user on the result
 of each service action. A scenario can run in one of two ways: straight through,
 printing out progress as it goes, or as an interactive question/answer script.

## Getting started with functions
```

```
Use an SDK to manage AWS Lambda functions: create a function, invoke it, update
 its code, invoke it again, view its output and logs, and delete it.

This scenario uses two Lambda handlers:
_Note: Handlers don't use AWS SDK API calls._

The increment handler is straightforward:

1. It accepts a number, increments it, and returns the new value.
2. It performs simple logging of the result.

The arithmetic handler is more complex:
1. It accepts a set of actions ['plus', 'minus', 'times', 'divided-by'] and two
 numbers, and returns the result of the calculation.
2. It uses an environment variable to control log level (such as DEBUG, INFO,
 WARNING, ERROR).
It logs a few things at different levels, such as:
    * DEBUG: Full event data.
    * INFO: The calculation result.
    * WARN~ING~: When a divide by zero error occurs.
    * This will be the typical `RUST_LOG` variable.


The steps of the scenario are:

1. Create an AWS Identity and Access Management (IAM) role that meets the
 following requirements:
    * Has an assume_role policy that grants 'lambda.amazonaws.com' the
 'sts:AssumeRole' action.
    * Attaches the 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' managed role.
    * _You must wait for ~10 seconds after the role is created before you can use
 it!_
2. Create a function (CreateFunction) for the increment handler by packaging it
 as a zip and doing one of the following:
    * Adding it with CreateFunction Code.ZipFile.
    * --or--
    * Uploading it to Amazon Simple Storage Service (Amazon S3) and adding it
 with CreateFunction Code.S3Bucket/S3Key.
    * _Note: Zipping the file does not have to be done in code._
    * If you have a waiter, use it to wait until the function is active.
 Otherwise, call GetFunction until State is Active.
3. Invoke the function with a number and print the result.
```

```
4. Update the function (UpdateFunctionCode) to the arithmetic handler by
 packaging it as a zip and doing one of the following:
    * Adding it with UpdateFunctionCode ZipFile.
    * --or--
    * Uploading it to Amazon S3 and adding it with UpdateFunctionCode S3Bucket/
S3Key.
5. Call GetFunction until Configuration.LastUpdateStatus is 'Successful' (or
 'Failed').
6. Update the environment variable by calling UpdateFunctionConfiguration and
 pass it a log level, such as:
    * Environment={'Variables': {'RUST_LOG': 'TRACE'}}
7. Invoke the function with an action from the list and a couple of values.
 Include LogType='Tail' to get logs in the result. Print the result of the
 calculation and the log.
8. [Optional] Invoke the function to provoke a divide-by-zero error and show the
 log result.
9. List all functions for the account, using pagination (ListFunctions).
10. Delete the function (DeleteFunction).
11. Delete the role.

Each step should use the function created in Service Actions to abstract calling
 the SDK.
 */

use aws_sdk_lambda::{operation::invoke::InvokeOutput, types::Environment};
use clap::Parser;
use std::{collections::HashMap, path::PathBuf};
use tracing::{debug, info, warn};
use tracing_subscriber::EnvFilter;

use lambda_code_examples::actions::{
    InvokeArgs::{Arithmetic, Increment},
    LambdaManager, Operation,
};

#[derive(Debug, Parser)]
pub struct Opt {
    /// The AWS Region.
    #[structopt(short, long)]
    pub region: Option<String>,

    // The bucket to use for the FunctionCode.
    #[structopt(short, long)]
    pub bucket: Option<String>,
```

```
    // The name of the Lambda function.
    #[structopt(short, long)]
    pub lambda_name: Option<String>,

    // The number to increment.
    #[structopt(short, long, default_value = "12")]
    pub inc: i32,

    // The left operand.
    #[structopt(long, default_value = "19")]
    pub num_a: i32,

    // The right operand.
    #[structopt(long, default_value = "23")]
    pub num_b: i32,

    // The arithmetic operation.
    #[structopt(short, long, default_value = "plus")]
    pub operation: Operation,

    #[structopt(long)]
    pub cleanup: Option<bool>,

    #[structopt(long)]
    pub no_cleanup: Option<bool>,
}

fn code_path(lambda: &str) -> PathBuf {
    PathBuf::from(format!("../target/lambda/{lambda}/bootstrap.zip"))
}

fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
```

```
}

async fn main_block(
    opt: &Opt,
    manager: &LambdaManager,
    code_location: String,
) -> Result<(), anyhow::Error> {
    let invoke = manager.invoke(Increment(opt.inc)).await?;
    log_invoke_output(&invoke, "Invoked function configured as increment");

    let update_code = manager
        .update_function_code(code_path("arithmetic"), code_location.clone())
        .await?;

    let code_sha256 = update_code.code_sha256().unwrap_or("Unknown SHA");
    info!(?code_sha256, "Updated function code with arithmetic.zip");

    let arithmetic_args = Arithmetic(opt.operation, opt.num_a, opt.num_b);
    let invoke = manager.invoke(arithmetic_args).await?;
    log_invoke_output(&invoke, "Invoked function configured as arithmetic");

    let update = manager
        .update_function_configuration(
            Environment::builder()
                .set_variables(Some(HashMap::from([(
                    "RUST_LOG".to_string(),
                    "trace".to_string(),
                )])))
                .build(),
        )
        .await?;
    let updated_environment = update.environment();
    info!(?updated_environment, "Updated function configuration");

    let invoke = manager
        .invoke(Arithmetic(opt.operation, opt.num_a, opt.num_b))
        .await?;
    log_invoke_output(
        &invoke,
        "Invoked function configured as arithmetic with increased logging",
    );

    let invoke = manager
        .invoke(Arithmetic(Operation::DividedBy, opt.num_a, 0))
```

```
        .await?;
    log_invoke_output(
        &invoke,
        "Invoked function configured as arithmetic with divide by zero",
    );

    Ok::<(), anyhow::Error>(())
}

#[tokio::main]
async fn main() {
    tracing_subscriber::fmt()
        .without_time()
        .with_file(true)
        .with_line_number(true)
        .with_env_filter(EnvFilter::from_default_env())
        .init();

    let opt = Opt::parse();
    let manager = LambdaManager::load_from_env(opt.lambda_name.clone(),
 opt.bucket.clone()).await;

    let key = match manager.create_function(code_path("increment")).await {
        Ok(init) => {
            info!(?init, "Created function, initially with increment.zip");
            let run_block = main_block(&opt, &manager, init.clone()).await;
            info!(?run_block, "Finished running example, cleaning up");
            Some(init)
        }
        Err(err) => {
            warn!(?err, "Error happened when initializing function");
            None
        }
    };

    if Some(false) == opt.cleanup || Some(true) == opt.no_cleanup {
        info!("Skipping cleanup")
    } else {
        let delete = manager.cleanup(key).await;
        info!(?delete, "Deleted function & cleaned up resources");
    }
}
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Rust API reference 中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
   TRY.
       "Create an AWS Identity and Access Management (IAM) role that grants AWS
Lambda permission to write to logs."
       DATA(lv_policy_document) = `{` &&
           `"Version":"2012-10-17",       ` &&
                 `"Statement": [` &&
                   `{` &&
                     `"Effect": "Allow",` &&
                     `"Action": [` &&
                       `"sts:AssumeRole"` &&
                     `],` &&
                     `"Principal": {` &&
                       `"Service": [` &&
                         `"lambda.amazonaws.com"` &&
                       `]` &&
                     `}` &&
                   `}` &&
                 `]` &&
               `}`.
```

```
        TRY.
            DATA(lo_create_role_output) = lo_iam->createrole(
                    iv_rolename = iv_role_name
                    iv_assumerolepolicydocument = lv_policy_document
                    iv_description = 'Grant lambda permission to write to
 logs' ).
            DATA(lv_role_arn) = lo_create_role_output->get_role( )->get_arn( ).
            MESSAGE 'IAM role created.' TYPE 'I'.
            WAIT UP TO 10 SECONDS.            " Make sure that the IAM role is
 ready for use. "
          CATCH /aws1/cx_iamentityalrdyexex.
            DATA(lo_role) = lo_iam->getrole( iv_rolename = iv_role_name ).
            lv_role_arn = lo_role->get_role( )->get_arn( ).
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iammalformedplydocex.
            MESSAGE 'Policy document in the request is malformed.' TYPE 'E'.
        ENDTRY.

        TRY.
            lo_iam->attachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' ).
            MESSAGE 'Attached policy to the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'E'.
          CATCH /aws1/cx_iamplynotattachableex.
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

        " Create a Lambda function and upload handler code. "
        " Lambda function performs 'increment' action on a number. "
        TRY.
            lo_lmd->createfunction(
                 iv_functionname = iv_function_name
                 iv_runtime = `python3.9`
                 iv_role = lv_role_arn
```

```
                iv_handler = iv_handler
                io_code = io_initial_zip_file
                iv_description = 'AWS Lambda code example' ).
            MESSAGE 'Lambda function created.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        " Verify the function is in Active state "
        WHILE lo_lmd->getfunction( iv_functionname = iv_function_name )-
>get_configuration( )->ask_state( ) <> 'Active'.
          IF sy-index = 10.
            EXIT.               " Maximum 10 seconds. "
          ENDIF.
          WAIT UP TO 1 SECONDS.
        ENDWHILE.

        "Invoke the function with a single parameter and get results."
        TRY.
            DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
                `"action": "increment",`  &&
                `"number": 10` &&
              `}` ).
            DATA(lo_initial_invoke_output) = lo_lmd->invoke(
                       iv_functionname = iv_function_name
                       iv_payload = lv_json ).
            ov_initial_invoke_payload = lo_initial_invoke_output->get_payload( ).
           " ov_initial_invoke_payload is returned for testing purposes. "
            DATA(lo_writer_json) = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_initial_invoke_payload RESULT
 XML lo_writer_json.
            DATA(lv_result) = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
```

```
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.

        " Update the function code and configure its Lambda environment with an
 environment variable. "
        " Lambda function is updated to perform 'decrement' action also. "
        TRY.
            lo_lmd->updatefunctioncode(
                  iv_functionname = iv_function_name
                  iv_zipfile = io_updated_zip_file ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function code updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdcodestorageexcdex.
            MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        TRY.
            DATA lt_variables TYPE /aws1/
cl_lmdenvironmentvaria00=>tt_environmentvariables.
            DATA ls_variable LIKE LINE OF lt_variables.
            ls_variable-key = 'LOG_LEVEL'.
            ls_variable-value = NEW /aws1/cl_lmdenvironmentvaria00( iv_value =
 'info' ).
            INSERT ls_variable INTO TABLE lt_variables.

            lo_lmd->updatefunctionconfiguration(
                  iv_functionname = iv_function_name
                  io_environment = NEW /aws1/cl_lmdenvironment( it_variables =
 lt_variables ) ).
            WAIT UP TO 10 SECONDS.            " Make sure that the update is
 completed. "
            MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourceconflictex.
```

```
            MESSAGE 'Resource already exists or another operation is in
 progress.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
        ENDTRY.

        "Invoke the function with new parameters and get results. Display the
 execution log that's returned from the invocation."
        TRY.
            lv_json = /aws1/cl_rt_util=>string_to_xstring(
              `{`  &&
                `"action": "decrement",`  &&
                `"number": 10` &&
              `}` ).
            DATA(lo_updated_invoke_output) = lo_lmd->invoke(
                       iv_functionname = iv_function_name
                       iv_payload = lv_json ).
            ov_updated_invoke_payload = lo_updated_invoke_output->get_payload( ).
           " ov_updated_invoke_payload is returned for testing purposes. "
            lo_writer_json = cl_sxml_string_writer=>create( type =
 if_sxml=>co_xt_json ).
            CALL TRANSFORMATION id SOURCE XML ov_updated_invoke_payload RESULT
 XML lo_writer_json.
            lv_result = cl_abap_codepage=>convert_from( lo_writer_json-
>get_output( ) ).
            MESSAGE 'Lambda function invoked.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdinvrequestcontex.
            MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'E'.
          CATCH /aws1/cx_lmdunsuppedmediatyp00.
            MESSAGE 'Invoke request body does not have JSON as its content type.'
 TYPE 'E'.
        ENDTRY.

        " List the functions for your account. "
        TRY.
            DATA(lo_list_output) = lo_lmd->listfunctions( ).
            DATA(lt_functions) = lo_list_output->get_functions( ).
            MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
```

```
        ENDTRY.

        " Delete the Lambda function. "
        TRY.
            lo_lmd->deletefunction( iv_functionname = iv_function_name ).
            MESSAGE 'Lambda function deleted.' TYPE 'I'.
          CATCH /aws1/cx_lmdinvparamvalueex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_lmdresourcenotfoundex.
            MESSAGE 'The requested resource does not exist.' TYPE 'W'.
        ENDTRY.

        " Detach role policy. "
        TRY.
            lo_iam->detachrolepolicy(
                iv_rolename  = iv_role_name
                iv_policyarn = 'arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole' ).
            MESSAGE 'Detached policy from the IAM role.' TYPE 'I'.
          CATCH /aws1/cx_iaminvalidinputex.
            MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'W'.
          CATCH /aws1/cx_iamplynotattachableex.
            MESSAGE 'Service role policies can only be attached to the service-
linked role for their service.' TYPE 'E'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

        " Delete the IAM role. "
        TRY.
            lo_iam->deleterole( iv_rolename = iv_role_name ).
            MESSAGE 'IAM role deleted.' TYPE 'I'.
          CATCH /aws1/cx_iamnosuchentityex.
            MESSAGE 'The requested resource entity does not exist.' TYPE 'W'.
          CATCH /aws1/cx_iamunmodableentityex.
            MESSAGE 'Service that depends on the service-linked role is not
 modifiable.' TYPE 'E'.
        ENDTRY.

      CATCH /aws1/cx_rt_service_generic INTO lo_exception.
        DATA(lv_error) = lo_exception->get_longtext( ).
```

```
        MESSAGE lv_error TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請參閱《適用於 SAP ABAP 的AWS SDK API 參考》中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions
  - UpdateFunctionCode
  - UpdateFunctionConfiguration

Swift

適用於 Swift 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

定義第一個 Lambda 函式，此函式只會遞增指定的值。

```
// swift-tools-version: 5.9
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
//
// The swift-tools-version declares the minimum version of Swift required to
// build this package.

import PackageDescription

let package = Package(
    name: "increment",
    // Let Xcode know the minimum Apple platforms supported.
    platforms: [
        .macOS(.v13)
```

```
    ],
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        .package(
            url: "https://github.com/swift-server/swift-aws-lambda-runtime.git",
            branch: "main"),
    ],
    targets: [
        // Targets are the basic building blocks of a package, defining a module
 or a test suite.
        // Targets can depend on other targets in this package and products
        // from dependencies.
        .executableTarget(
            name: "increment",
            dependencies: [
                .product(name: "AWSLambdaRuntime", package: "swift-aws-lambda-
runtime"),
            ],
            path: "Sources"
        )
    ]
)

import Foundation
import AWSLambdaRuntime

/// Represents the contents of the requests being received from the client.
/// This structure must be `Decodable` to indicate that its initializer
/// converts an external representation into this type.
struct Request: Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The number to act upon.
    let number: Int
}

/// The contents of the response sent back to the client. This must be
/// `Encodable`.
struct Response: Encodable, Sendable {
    /// The resulting value after performing the action.
    let answer: Int?
}
```

```
/// The Lambda function body.
///
/// - Parameters:
///   - event: The `Request` describing the request made by the
///     client.
///   - context: A `LambdaContext` describing the context in
///     which the lambda function is running.
///
/// - Returns: A `Response` object that will be encoded to JSON and sent
///   to the client by the Lambda runtime.
let incrementLambdaRuntime = LambdaRuntime {
        (event: Request, context: LambdaContext) -> Response in
    let action = event.action
    var answer: Int?

    if action != "increment" {
        context.logger.error("Unrecognized operation: \"\(action)\". The only
 supported action is \"increment\".")
    } else {
        answer = event.number + 1
        context.logger.info("The calculated answer is \(answer!).")
    }

    let response = Response(answer: answer)
    return response
}

// Run the Lambda runtime code.

try await incrementLambdaRuntime.run()
```

定義第二個 Lambda 函式，此函式會算術運算兩個數字。

```
// swift-tools-version: 5.9
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
//
// The swift-tools-version declares the minimum version of Swift required to
// build this package.

import PackageDescription
```

```
let package = Package(
    name: "calculator",
    // Let Xcode know the minimum Apple platforms supported.
    platforms: [
        .macOS(.v13)
    ],
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        .package(
            url: "https://github.com/swift-server/swift-aws-lambda-runtime.git",
            branch: "main"),
    ],
    targets: [
        // Targets are the basic building blocks of a package, defining a module
 or a test suite.
        // Targets can depend on other targets in this package and products
        // from dependencies.
        .executableTarget(
            name: "calculator",
            dependencies: [
                .product(name: "AWSLambdaRuntime", package: "swift-aws-lambda-
runtime"),
            ],
            path: "Sources"
        )
    ]
)

import Foundation
import AWSLambdaRuntime

/// Represents the contents of the requests being received from the client.
/// This structure must be `Decodable` to indicate that its initializer
/// converts an external representation into this type.
struct Request: Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The first number to act upon.
    let x: Int
    /// The second number to act upon.
    let y: Int
}

/// A dictionary mapping operation names to closures that perform that
```

```
/// operation and return the result.
let actions = [
    "plus": { (x: Int, y: Int) -> Int in
        return x + y
    },
    "minus": { (x: Int, y: Int) -> Int in
        return x - y
    },
    "times": { (x: Int, y: Int) -> Int in
        return x * y
    },
    "divided-by": { (x: Int, y: Int) -> Int in
        return x / y
    }
]

/// The contents of the response sent back to the client. This must be
/// `Encodable`.
struct Response: Encodable, Sendable {
    /// The resulting value after performing the action.
    let answer: Int?
}


/// The Lambda function's entry point. Called by the Lambda runtime.
///
/// - Parameters:
///   - event: The `Request` describing the request made by the
///     client.
///   - context: A `LambdaContext` describing the context in
///     which the lambda function is running.
///
/// - Returns: A `Response` object that will be encoded to JSON and sent
///   to the client by the Lambda runtime.
let calculatorLambdaRuntime = LambdaRuntime {
        (_ event: Request, context: LambdaContext) -> Response in
    let action = event.action
    var answer: Int?
    var actionFunc: ((Int, Int) -> Int)?

    // Get the closure to run to perform the calculation.

    actionFunc = await actions[action]
```

```
    guard let actionFunc else {
        context.logger.error("Unrecognized operation '\(action)\'")
        return Response(answer: nil)
    }

    // Perform the calculation and return the answer.

    answer = actionFunc(event.x, event.y)

    guard let answer else {
        context.logger.error("Error computing \(event.x) \(action) \(event.y)")
    }
    context.logger.info("\(event.x) \(action) \(event.y) = \(answer)")

    return Response(answer: answer)
}

try await calculatorLambdaRuntime.run()
```

定義將會調用兩個 Lambda 函式的主要程式。

```
// swift-tools-version: 5.9
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
//
// The swift-tools-version declares the minimum version of Swift required to
// build this package.

import PackageDescription

let package = Package(
    name: "lambda-basics",
    // Let Xcode know the minimum Apple platforms supported.
    platforms: [
        .macOS(.v13)
    ],
    dependencies: [
        // Dependencies declare other packages that this package depends on.
        .package(
            url: "https://github.com/awslabs/aws-sdk-swift",
            from: "1.0.0"),
        .package(
```

```
            url: "https://github.com/apple/swift-argument-parser.git",
            branch: "main"
        )
    ],
    targets: [
        // Targets are the basic building blocks of a package, defining a module
 or a test suite.
        // Targets can depend on other targets in this package and products
        // from dependencies.
        .executableTarget(
            name: "lambda-basics",
            dependencies: [
                .product(name: "AWSLambda", package: "aws-sdk-swift"),
                .product(name: "AWSIAM", package: "aws-sdk-swift"),
                .product(name: "ArgumentParser", package: "swift-argument-
parser")
            ],
            path: "Sources"
        )
    ]
)

//
/// An example demonstrating a variety of important AWS Lambda functions.

import ArgumentParser
import AWSIAM
import SmithyWaitersAPI
import AWSClientRuntime
import AWSLambda
import Foundation

/// Represents the contents of the requests being received from the client.
/// This structure must be `Decodable` to indicate that its initializer
/// converts an external representation into this type.
struct IncrementRequest: Encodable, Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The number to act upon.
    let number: Int
}

struct Response: Encodable, Decodable, Sendable {
    /// The resulting value after performing the action.
```

```
    let answer: Int?
}

struct CalculatorRequest: Encodable, Decodable, Sendable {
    /// The action to perform.
    let action: String
    /// The first number to act upon.
    let x: Int
    /// The second number to act upon.
    let y: Int
}

let exampleName = "SwiftLambdaRoleExample"
let basicsFunctionName = "lambda-basics-function"

/// The ARN of the standard IAM policy for execution of Lambda functions.
let policyARN = "arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"

struct ExampleCommand: ParsableCommand {
    // -MARK: Command arguments
    @Option(help: "Name of the IAM Role to use for the Lambda functions")
    var role = exampleName
    @Option(help: "Zip archive containing the 'increment' lambda function")
    var incpath: String
    @Option(help: "Zip archive containing the 'calculator' lambda function")
    var calcpath: String
    @Option(help: "Name of the Amazon S3 Region to use (default: us-east-1)")
    var region = "us-east-1"

    static var configuration = CommandConfiguration(
        commandName: "lambda-basics",
        abstract: """
        This example demonstrates several common operations using AWS Lambda.
        """,
        discussion: """
        """
    )

    /// Returns the specified IAM role object.
    ///
    /// - Parameters:
    ///   - iamClient: `IAMClient` to use when looking for the role.
    ///   - roleName: The name of the role to check.
```

```
    ///
    /// - Returns: The `IAMClientTypes.Role` representing the specified role.
    func getRole(iamClient: IAMClient, roleName: String) async throws
                 -> IAMClientTypes.Role {
        do {
            let roleOutput = try await iamClient.getRole(
                input: GetRoleInput(
                    roleName: roleName
                )
            )

            guard let role = roleOutput.role else {
                throw ExampleError.roleNotFound
            }
            return role
        } catch {
            throw ExampleError.roleNotFound
        }
    }

    /// Create the AWS IAM role that will be used to access AWS Lambda.
    ///
    /// - Parameters:
    ///   - iamClient: The AWS `IAMClient` to use.
    ///   - roleName: The name of the AWS IAM role to use for Lambda.
    ///
    /// - Throws: `ExampleError.roleCreateError`
    ///
    /// - Returns: The `IAMClientTypes.Role` struct that describes the new role.
    func createRoleForLambda(iamClient: IAMClient, roleName: String) async throws
-> IAMClientTypes.Role {
        let output = try await iamClient.createRole(
            input: CreateRoleInput(
                assumeRolePolicyDocument:
                """
                {
                    "Version":"2012-10-17",
                    "Statement": [
                        {
                            "Effect": "Allow",
                            "Principal": {"Service": "lambda.amazonaws.com"},
                            "Action": "sts:AssumeRole"
                        }
                    ]
```

```
                }
                """,
                roleName: roleName
            )
        )

        guard let role = output.role else {
            throw ExampleError.roleCreateError
        }

        // Wait for the role to be ready for use.

        _ = try await iamClient.waitUntilRoleExists(
            options: WaiterOptions(
                maxWaitTime: 20,
                minDelay: 0.5,
                maxDelay: 2
            ),
            input: GetRoleInput(roleName: roleName)
        )

        return role
    }

    /// Detect whether or not the AWS Lambda function with the specified name
    /// exists, by requesting its function information.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - name: The name of the AWS Lambda function to find.
    ///
    /// - Returns: `true` if the Lambda function exists. Otherwise `false`.
    func doesLambdaFunctionExist(lambdaClient: LambdaClient, name: String) async
-> Bool {
        do {
            _ = try await lambdaClient.getFunction(
                input: GetFunctionInput(functionName: name)
            )
        } catch {
            return false
        }

        return true
    }
```

```
    /// Create the specified AWS Lambda function.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to create.
    ///   - roleArn: The ARN of the role to apply to the function.
    ///   - path: The path of the Zip archive containing the function.
    ///
    /// - Returns: `true` if the AWS Lambda was successfully created; `false`
    ///   if it wasn't.
    func createFunction(lambdaClient: LambdaClient, functionName: String,
                                roleArn: String?, path: String) async throws ->
Bool {
        do {
            // Read the Zip archive containing the AWS Lambda function.

            let zipUrl = URL(fileURLWithPath: path)
            let zipData = try Data(contentsOf: zipUrl)

            // Create the AWS Lambda function that runs the specified code,
            // using the name given on the command line. The Lambda function
            // will run using the Amazon Linux 2 runtime.

            _ = try await lambdaClient.createFunction(
                input: CreateFunctionInput(
                    code: LambdaClientTypes.FunctionCode(zipFile: zipData),
                    functionName: functionName,
                    handler: "handle",
                    role: roleArn,
                    runtime: .providedal2
                )
            )
        } catch {
            print("*** Error creating Lambda function:")
            dump(error)
            return false
        }

        // Wait for a while to be sure the function is done being created.

        let output = try await lambdaClient.waitUntilFunctionActiveV2(
            options: WaiterOptions(
                maxWaitTime: 20,
```

```
                minDelay: 0.5,
                maxDelay: 2
            ),
            input: GetFunctionInput(functionName: functionName)
        )

        switch output.result {
            case .success:
                return true
            case .failure:
                return false
        }
    }

    /// Update the AWS Lambda function with new code to run when the function
    /// is invoked.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to update.
    ///   - path: The pathname of the Zip file containing the packaged Lambda
    ///     function.
    /// - Throws: `ExampleError.zipFileReadError`
    /// - Returns: `true` if the function's code is updated successfully.
    ///   Otherwise, returns `false`.
    func updateFunctionCode(lambdaClient: LambdaClient, functionName: String,
                            path: String) async throws -> Bool {
        let zipUrl = URL(fileURLWithPath: path)
        let zipData: Data

        // Read the function's Zip file.

        do {
            zipData = try Data(contentsOf: zipUrl)
        } catch {
            throw ExampleError.zipFileReadError
        }

        // Update the function's code and wait for the updated version to be
        // ready for use.

        do {
            _ = try await lambdaClient.updateFunctionCode(
                input: UpdateFunctionCodeInput(
```

```
                    functionName: functionName,
                    zipFile: zipData
                )
            )
        } catch {
            return false
        }

        let output = try await lambdaClient.waitUntilFunctionUpdatedV2(
            options: WaiterOptions(
                maxWaitTime: 20,
                minDelay: 0.5,
                maxDelay: 2
            ),
            input: GetFunctionInput(
                functionName: functionName
            )
        )

        switch output.result {
            case .success:
                return true
            case .failure:
                return false
        }
    }

    /// Tell the server-side component to log debug output by setting its
    /// environment's `LOG_LEVEL` to `DEBUG`.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to enable debug
    ///     logging for.
    ///
    /// - Throws: `ExampleError.environmentResponseMissingError`,
    ///   `ExampleError.updateFunctionConfigurationError`,
    ///   `ExampleError.environmentVariablesMissingError`,
    ///   `ExampleError.logLevelIncorrectError`,
    ///   `ExampleError.updateFunctionConfigurationError`
    func enableDebugLogging(lambdaClient: LambdaClient, functionName: String)
async throws {
        let envVariables = [
            "LOG_LEVEL": "DEBUG"
```

```
        ]
        let environment = LambdaClientTypes.Environment(variables: envVariables)

        do {
            let output = try await lambdaClient.updateFunctionConfiguration(
                input: UpdateFunctionConfigurationInput(
                    environment: environment,
                    functionName: functionName
                )
            )

            guard let response = output.environment else {
                throw ExampleError.environmentResponseMissingError
            }

            if response.error != nil {
                throw ExampleError.updateFunctionConfigurationError
            }

            guard let retVariables = response.variables else {
                throw ExampleError.environmentVariablesMissingError
            }

            for envVar in retVariables {
                if envVar.key == "LOG_LEVEL" && envVar.value != "DEBUG" {
                    print("*** Log level is not set to DEBUG!")
                    throw ExampleError.logLevelIncorrectError
                }
            }
        } catch {
            throw ExampleError.updateFunctionConfigurationError
        }
    }

    /// Returns an array containing the names of all AWS Lambda functions
    /// available to the user.
    ///
    /// - Parameter lambdaClient: The `IAMClient` to use.
    ///
    /// - Throws: `ExampleError.listFunctionsError`
    ///
    /// - Returns: An array of lambda function name strings.
    func getFunctionNames(lambdaClient: LambdaClient) async throws -> [String] {
        let pages = lambdaClient.listFunctionsPaginated(
```

```
            input: ListFunctionsInput()
        )

        var functionNames: [String] = []

        for try await page in pages {
            guard let functions = page.functions else {
                throw ExampleError.listFunctionsError
            }

            for function in functions {
                functionNames.append(function.functionName ?? "<unknown>")
            }
        }

        return functionNames
    }

    /// Invoke the Lambda function to increment a value.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `IAMClient` to use.
    ///   - number: The number to increment.
    ///
    /// - Throws: `ExampleError.noAnswerReceived`, `ExampleError.invokeError`
    ///
    /// - Returns: An integer number containing the incremented value.
    func invokeIncrement(lambdaClient: LambdaClient, number: Int) async throws ->
Int {
        do {
            let incRequest = IncrementRequest(action: "increment", number:
number)
            let incData = try! JSONEncoder().encode(incRequest)

            // Invoke the lambda function.

            let invokeOutput = try await lambdaClient.invoke(
                input: InvokeInput(
                    functionName: "lambda-basics-function",
                    payload: incData
                )
            )
```

```
            let response = try! JSONDecoder().decode(Response.self,
from:invokeOutput.payload!)

            guard let answer = response.answer else {
                throw ExampleError.noAnswerReceived
            }
            return answer

        } catch {
            throw ExampleError.invokeError
        }
    }

    /// Invoke the calculator Lambda function.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `IAMClient` to use.
    ///   - action: Which arithmetic operation to perform: "plus", "minus",
    ///     "times", or "divided-by".
    ///   - x: The first number to use in the computation.
    ///   - y: The second number to use in the computation.
    ///
    /// - Throws: `ExampleError.noAnswerReceived`, `ExampleError.invokeError`
    ///
    /// - Returns: The computed answer as an `Int`.
    func invokeCalculator(lambdaClient: LambdaClient, action: String, x: Int, y:
Int) async throws -> Int {
        do {
            let calcRequest = CalculatorRequest(action: action, x: x, y: y)
            let calcData = try! JSONEncoder().encode(calcRequest)

            // Invoke the lambda function.

            let invokeOutput = try await lambdaClient.invoke(
                input: InvokeInput(
                    functionName: "lambda-basics-function",
                    payload: calcData
                )
            )

            let response = try! JSONDecoder().decode(Response.self,
from:invokeOutput.payload!)

            guard let answer = response.answer else {
```

```
                throw ExampleError.noAnswerReceived
            }
            return answer

        } catch {
            throw ExampleError.invokeError
        }

    }

    /// Perform the example's tasks.
    func basics() async throws {
        let iamClient = try await IAMClient(
            config: IAMClient.IAMClientConfiguration(region: region)
        )

        let lambdaClient = try await LambdaClient(
            config: LambdaClient.LambdaClientConfiguration(region: region)
        )

        /// The IAM role to use for the example.
        var iamRole: IAMClientTypes.Role

        // Look for the specified role. If it already exists, use it. If not,
        // create it and attach the desired policy to it.

        do {
            iamRole = try await getRole(iamClient: iamClient, roleName: role)
        } catch ExampleError.roleNotFound {
            // The role wasn't found, so create it and attach the needed
            // policy.

            iamRole = try await createRoleForLambda(iamClient: iamClient,
roleName: role)

            do {
                _ = try await iamClient.attachRolePolicy(
                    input: AttachRolePolicyInput(policyArn: policyARN, roleName:
role)
                )
            } catch {
                throw ExampleError.policyError
            }
        }
```

```
        // Give the policy time to attach to the role.

        sleep(5)

        // Look to see if the function already exists. If it does, throw an
        // error.

        if await doesLambdaFunctionExist(lambdaClient: lambdaClient, name:
basicsFunctionName) {
            throw ExampleError.functionAlreadyExists
        }

        // Create, then invoke, the "increment" version of the calculator
        // function.

        print("Creating the increment Lambda function...")
        if try await createFunction(lambdaClient: lambdaClient, functionName:
basicsFunctionName,
                                    roleArn: iamRole.arn, path: incpath) {
            print("Running increment function calls...")
            for number in 0...4 {
                do {
                    let answer = try await invokeIncrement(lambdaClient:
lambdaClient, number: number)
                    print("Increment \(number) = \(answer)")
                } catch {
                    print("Error incrementing \(number): ",
error.localizedDescription)
                }
            }
        } else {
            print("*** Failed to create the increment function.")
        }

        // Enable debug logging.

        print("\nEnabling debug logging...")
        try await enableDebugLogging(lambdaClient: lambdaClient, functionName:
basicsFunctionName)

        // Change it to a basic arithmetic calculator. Then invoke it a few
        // times.
```

```
        print("\nReplacing the Lambda function with a calculator...")

        if try await updateFunctionCode(lambdaClient: lambdaClient, functionName:
basicsFunctionName,
                                        path: calcpath) {
            print("Running calculator function calls...")
            for x in [6, 10] {
                for y in [2, 4] {
                    for action in ["plus", "minus", "times", "divided-by"] {
                        do {
                            let answer = try await invokeCalculator(lambdaClient:
lambdaClient, action: action, x: x, y: y)
                            print("\(x) \(action) \(y) = \(answer)")
                        } catch {
                            print("Error calculating \(x) \(action) \(y): ",
error.localizedDescription)
                        }
                    }
                }
            }
        }

        // List all lambda functions.

        let functionNames = try await getFunctionNames(lambdaClient:
lambdaClient)

        if functionNames.count > 0 {
            print("\nAWS Lambda functions available on your account:")
            for name in functionNames {
                print("  \(name)")
            }
        }

        // Delete the lambda function.

        print("Deleting lambda function...")

        do {
            _ = try await lambdaClient.deleteFunction(
                input: DeleteFunctionInput(
                    functionName: "lambda-basics-function"
                )
            )
```

```
        } catch {
            print("Error: Unable to delete the function.")
        }

        // Detach the role from the policy, then delete the role.

        print("Deleting the AWS IAM role...")

        do {
            _ = try await iamClient.detachRolePolicy(
                input: DetachRolePolicyInput(
                    policyArn: policyARN,
                    roleName: role
                )
            )
            _ = try await iamClient.deleteRole(
                input: DeleteRoleInput(
                    roleName: role
                )
            )
        } catch {
            throw ExampleError.deleteRoleError
        }
    }
}

// -MARK: - Entry point

/// The program's asynchronous entry point.
@main
struct Main {
    static func main() async {
        let args = Array(CommandLine.arguments.dropFirst())

        do {
            let command = try ExampleCommand.parse(args)
            try await command.basics()
        } catch {
            ExampleCommand.exit(withError: error)
        }
    }
}
```

```
/// Errors thrown by the example's functions.
enum ExampleError: Error {
    /// An AWS Lambda function with the specified name already exists.
    case functionAlreadyExists
    /// The specified role doesn't exist.
    case roleNotFound
    /// Unable to create the role.
    case roleCreateError
    /// Unable to delete the role.
    case deleteRoleError
    /// Unable to attach a policy to the role.
    case policyError
    /// Unable to get the executable directory.
    case executableNotFound
    /// An error occurred creating a lambda function.
    case createLambdaError
    /// An error occurred invoking the lambda function.
    case invokeError
    /// No answer received from the invocation.
    case noAnswerReceived
    /// Unable to list the AWS Lambda functions.
    case listFunctionsError
    /// Unable to update the AWS Lambda function.
    case updateFunctionError
    /// Unable to update the function configuration.
    case updateFunctionConfigurationError
    /// The environment response is missing after an
    /// UpdateEnvironmentConfiguration attempt.
    case environmentResponseMissingError
    /// The environment variables are missing from the EnvironmentResponse and
    /// no errors occurred.
    case environmentVariablesMissingError
    /// The log level is incorrect after attempting to set it.
    case logLevelIncorrectError
    /// Unable to load the AWS Lambda function's Zip file.
    case zipFileReadError

    var errorDescription: String? {
        switch self {
        case .functionAlreadyExists:
            return "An AWS Lambda function with that name already exists."
        case .roleNotFound:
            return "The specified role doesn't exist."
        case .deleteRoleError:
```

```
            return "Unable to delete the AWS IAM role."
        case .roleCreateError:
            return "Unable to create the specified role."
        case .policyError:
            return "An error occurred attaching the policy to the role."
        case .executableNotFound:
            return "Unable to find the executable program directory."
        case .createLambdaError:
            return "An error occurred creating a lambda function."
        case .invokeError:
            return "An error occurred invoking a lambda function."
        case .noAnswerReceived:
            return "No answer received from the lambda function."
        case .listFunctionsError:
            return "Unable to list the AWS Lambda functions."
        case .updateFunctionError:
            return "Unable to update the AWS lambda function."
        case .updateFunctionConfigurationError:
            return "Unable to update the AWS lambda function configuration."
        case .environmentResponseMissingError:
            return "The environment is missing from the response after updating
 the function configuration."
        case .environmentVariablesMissingError:
            return "While no error occurred, no environment variables were
 returned following function configuration."
        case .logLevelIncorrectError:
            return "The log level is incorrect after attempting to set it to
 DEBUG."
        case .zipFileReadError:
            return "Unable to read the AWS Lambda function."
        }
    }
}
```

- 如需 API 詳細資訊，請參閱適用於 Swift 的AWS SDK API 參考中的下列主題。
  - CreateFunction
  - DeleteFunction
  - GetFunction
  - Invoke
  - ListFunctions

- UpdateFunctionCode
- UpdateFunctionConfiguration

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 AWS SDKs的 Lambda 動作

下列程式碼範例示範如何使用 AWS SDKs 執行個別 Lambda 動作。每個範例均包含 GitHub 的連結，您可以在連結中找到設定和執行程式碼的相關說明。

這些摘錄會呼叫 Lambda API，是必須在內容中執行之大型程式的程式碼摘錄。您可以在 使用 AWS SDKs Lambda 案例 中查看內容中的動作。

下列範例僅包含最常使用的動作。如需完整清單，請參閱《AWS Lambda API 參考》。

範例

- 搭配使用 CreateAlias 與 CLI
- CreateFunction 搭配 AWS SDK 或 CLI 使用
- 搭配使用 DeleteAlias 與 CLI
- DeleteFunction 搭配 AWS SDK 或 CLI 使用
- 搭配使用 DeleteFunctionConcurrency 與 CLI
- 搭配使用 DeleteProvisionedConcurrencyConfig 與 CLI
- 搭配使用 GetAccountSettings 與 CLI
- 搭配使用 GetAlias 與 CLI
- GetFunction 搭配 AWS SDK 或 CLI 使用
- 搭配使用 GetFunctionConcurrency 與 CLI
- 搭配使用 GetFunctionConfiguration 與 CLI
- 搭配使用 GetPolicy 與 CLI
- 搭配使用 GetProvisionedConcurrencyConfig 與 CLI
- Invoke 搭配 AWS SDK 或 CLI 使用
- ListFunctions 搭配 AWS SDK 或 CLI 使用
- 搭配使用 ListProvisionedConcurrencyConfigs 與 CLI
- 搭配使用 ListTags 與 CLI

- 搭配使用 ListVersionsByFunction 與 CLI
- 搭配使用 PublishVersion 與 CLI
- 搭配使用 PutFunctionConcurrency 與 CLI
- 搭配使用 PutProvisionedConcurrencyConfig 與 CLI
- 搭配使用 RemovePermission 與 CLI
- 搭配使用 TagResource 與 CLI
- 搭配使用 UntagResource 與 CLI
- 搭配使用 UpdateAlias 與 CLI
- UpdateFunctionCode 搭配 AWS SDK 或 CLI 使用
- UpdateFunctionConfiguration 搭配 AWS SDK 或 CLI 使用

搭配使用 **CreateAlias** 與 CLI

下列程式碼範例示範如何使用 `CreateAlias`。

CLI

AWS CLI

建立 Lambda 函數的別名

以下 `create-alias` 範例會建立一個名為 `LIVE` 的別名，此別名指向 `my-function` Lambda 函數的版本 1。

```
aws lambda create-alias \
    --function-name my-function \
    --description "alias for live version of function" \
    --function-version 1 \
    --name LIVE
```

輸出：

```
{
    "FunctionVersion": "1",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "873282ed-4cd3-4dc8-a069-d0c647e470c6",
```

```
    "Description": "alias for live version of function"
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的設定 Lambda 函數別名。 AWS

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 CreateAlias。

PowerShell

Tools for PowerShell V4

範例 1：此範例會為指定的版本和路由組態建立新的 Lambda 別名，以指定其收到的調用請求百分比。

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 CreateAlias。

Tools for PowerShell V5

範例 1：此範例會為指定的版本和路由組態建立新的 Lambda 別名，以指定其收到的調用請求百分比。

```
New-LMAlias -FunctionName "MylambdaFunction123" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6} -Description "Alias
 for version 4" -FunctionVersion 4 -Name "PowershellAlias"
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 CreateAlias。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

`CreateFunction` 搭配 AWS SDK 或 CLI 使用

下列程式碼範例示範如何使用 `CreateFunction`。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 了解基本概念

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
/// <summary>
/// Creates a new Lambda function.
/// </summary>
/// <param name="functionName">The name of the function.</param>
/// <param name="s3Bucket">The Amazon Simple Storage Service (Amazon S3)
/// bucket where the zip file containing the code is located.</param>
/// <param name="s3Key">The Amazon S3 key of the zip file.</param>
/// <param name="role">The Amazon Resource Name (ARN) of a role with the
/// appropriate Lambda permissions.</param>
/// <param name="handler">The name of the handler function.</param>
/// <returns>The Amazon Resource Name (ARN) of the newly created
/// Lambda function.</returns>
public async Task<string> CreateLambdaFunctionAsync(
    string functionName,
    string s3Bucket,
    string s3Key,
    string role,
    string handler)
{
    // Defines the location for the function code.
    // S3Bucket - The S3 bucket where the file containing
    //            the source code is stored.
    // S3Key    - The name of the file containing the code.
    var functionCode = new FunctionCode
    {
        S3Bucket = s3Bucket,
        S3Key = s3Key,
    };

    var createFunctionRequest = new CreateFunctionRequest
```

```
        {
            FunctionName = functionName,
            Description = "Created by the Lambda .NET API",
            Code = functionCode,
            Handler = handler,
            Runtime = Runtime.Dotnet6,
            Role = role,
        };

        var reponse = await
 _lambdaService.CreateFunctionAsync(createFunctionRequest);
        return reponse.FunctionArn;
    }
```

- 如需 API 詳細資訊，請參閱《適用於 .NET 的 AWS SDK API 參考》中的「CreateFunction」。

C++

適用於 C++ 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
 (overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::CreateFunctionRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        request.SetDescription(LAMBDA_DESCRIPTION); // Optional.
#if USE_CPP_LAMBDA_FUNCTION
        request.SetRuntime(Aws::Lambda::Model::Runtime::provided_al2);
```

```
        request.SetTimeout(15);
        request.SetMemorySize(128);

        // Assume the AWS Lambda function was built in Docker with same
 architecture
        // as this code.
#if  defined(__x86_64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::x86_64});
#elif defined(__aarch64__)
        request.SetArchitectures({Aws::Lambda::Model::Architecture::arm64});
#else
#error "Unimplemented architecture"
#endif // defined(architecture)
#else
        request.SetRuntime(Aws::Lambda::Model::Runtime::python3_9);
#endif
        request.SetRole(roleArn);
        request.SetHandler(LAMBDA_HANDLER_NAME);
        request.SetPublish(true);
        Aws::Lambda::Model::FunctionCode code;
        std::ifstream ifstream(INCREMENT_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();

        code.SetZipFile(Aws::Utils::ByteBuffer((unsigned char *)
 buffer.str().c_str(),
                                               buffer.str().length()));
        request.SetCode(code);
```

```
        Aws::Lambda::Model::CreateFunctionOutcome outcome =
 client.CreateFunction(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda function was successfully created. " <<
 seconds
                      << " seconds elapsed." << std::endl;
            break;
        }

        else {
            std::cerr << "Error with CreateFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
            deleteIamRole(clientConfig);
            return false;
        }
```

- 如需 API 詳細資訊，請參閱《適用於 C++ 的 AWS SDK API 參考》中的「CreateFunction」。

CLI

AWS CLI

若要建立 Lambda 函數

下列 `create-function` 範例會建立名為 `my-function` 的 Lambda 函數。

```
aws lambda create-function \
    --function-name my-function \
    --runtime nodejs22.x \
    --zip-file fileb://my-function.zip \
    --handler my-function.handler \
    --role arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
tges6bf4
```

`my-function.zip` 的內容：

```
This file is a deployment package that contains your function code and any
 dependencies.
```

輸出：

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "PFn4S+er27qk+UuZSTKEQfNKG/XNn7QJs90mJgq6oH8=",
    "FunctionName": "my-function",
    "CodeSize": 308,
    "RevisionId": "873282ed-4cd3-4dc8-a069-d0c647e470c6",
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
    "Timeout": 3,
    "LastModified": "2025-10-14T22:26:11.234+0000",
    "Handler": "my-function.handler",
    "Runtime": "nodejs22.x",
    "Description": ""
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的設定 Lambda 函式記憶體。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 CreateFunction。

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import (
```

```
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// CreateFunction creates a new Lambda function from code contained in the
 zipPackage
// buffer. The specified handlerName must match the name of the file and function
// contained in the uploaded code. The role specified by iamRoleArn is assumed by
// Lambda and grants specific permissions.
// When the function already exists, types.StateActive is returned.
// When the function is created, a lambda.FunctionActiveV2Waiter is used to wait
 until the
// function is active.
func (wrapper FunctionWrapper) CreateFunction(ctx context.Context, functionName
 string, handlerName string,
 iamRoleArn *string, zipPackage *bytes.Buffer) types.State {
 var state types.State
 _, err := wrapper.LambdaClient.CreateFunction(ctx, &lambda.CreateFunctionInput{
  Code:         &types.FunctionCode{ZipFile: zipPackage.Bytes()},
  FunctionName: aws.String(functionName),
  Role:         iamRoleArn,
  Handler:      aws.String(handlerName),
  Publish:      true,
  Runtime:      types.RuntimePython39,
 })
 if err != nil {
  var resConflict *types.ResourceConflictException
  if errors.As(err, &resConflict) {
```

```
   log.Printf("Function %v already exists.\n", functionName)
   state = types.StateActive
  } else {
   log.Panicf("Couldn't create function %v. Here's why: %v\n", functionName, err)
  }
 } else {
  waiter := lambda.NewFunctionActiveV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}
```

- 如需 API 詳細資訊，請參閱《適用於 Go 的 AWS SDK API 參考》中的「CreateFunction」。

Java

適用於 Java 2.x 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /**
     * Creates a new Lambda function in AWS using the AWS Lambda Java API.
     *
     * @param awsLambda    the AWS Lambda client used to interact with the AWS
 Lambda service
     * @param functionName the name of the Lambda function to create
     * @param key          the S3 key of the function code
     * @param bucketName   the name of the S3 bucket containing the function code
     * @param role         the IAM role to assign to the Lambda function
```

```
     * @param handler      the fully qualified class name of the function handler
     * @return the Amazon Resource Name (ARN) of the created Lambda function
     */
    public static String createLambdaFunction(LambdaClient awsLambda,
                                               String functionName,
                                               String key,
                                               String bucketName,
                                               String role,
                                               String handler) {

        try {
            LambdaWaiter waiter = awsLambda.waiter();
            FunctionCode code = FunctionCode.builder()
                .s3Key(key)
                .s3Bucket(bucketName)
                .build();

            CreateFunctionRequest functionRequest =
CreateFunctionRequest.builder()
                .functionName(functionName)
                .description("Created by the Lambda Java API")
                .code(code)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .role(role)
                .build();

            // Create a Lambda function using a waiter
            CreateFunctionResponse functionResponse =
awsLambda.createFunction(functionRequest);
            GetFunctionRequest getFunctionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();
            WaiterResponse<GetFunctionResponse> waiterResponse =
waiter.waitUntilFunctionExists(getFunctionRequest);
            waiterResponse.matched().response().ifPresent(System.out::println);
            return functionResponse.functionArn();

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
        return "";
    }
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Java 2.x API 參考》中的「CreateFunction」。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
const createFunction = async (funcName, roleArn) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${funcName}.zip`);

  const command = new CreateFunctionCommand({
    Code: { ZipFile: code },
    FunctionName: funcName,
    Role: roleArn,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- 如需 API 詳細資訊，請參閱《適用於 JavaScript 的 AWS SDK API 參考》中的「CreateFunction」。

Kotlin

適用於 Kotlin 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
suspend fun createNewFunction(
    myFunctionName: String,
    s3BucketName: String,
    myS3Key: String,
    myHandler: String,
    myRole: String,
): String? {
    val functionCode =
        FunctionCode {
            s3Bucket = s3BucketName
            s3Key = myS3Key
        }

    val request =
        CreateFunctionRequest {
            functionName = myFunctionName
            code = functionCode
            description = "Created by the Lambda Kotlin API"
            handler = myHandler
            role = myRole
            runtime = Runtime.Java17
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
        val functionResponse = awsLambda.createFunction(request)
        awsLambda.waitUntilFunctionActive {
            functionName = myFunctionName
        }
        return functionResponse.functionArn
    }
}
```

- 如需 API 詳細資訊，請參閱《適用於 Kotlin 的AWS 開發套件 API 參考》中的「CreateFunction」。

PHP

適用於 PHP 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    public function createFunction($functionName, $role, $bucketName, $handler)
    {
        //This assumes the Lambda function is in an S3 bucket.
        return $this->customWaiter(function () use ($functionName, $role,
$bucketName, $handler) {
            return $this->lambdaClient->createFunction([
                'Code' => [
                    'S3Bucket' => $bucketName,
                    'S3Key' => $functionName,
                ],
                'FunctionName' => $functionName,
                'Role' => $role['Arn'],
                'Runtime' => 'python3.9',
                'Handler' => "$handler.lambda_handler",
            ]);
        });
    }
```

- 如需 API 詳細資訊，請參閱《適用於 PHP 的 AWS SDK API 參考》中的「CreateFunction」。

PowerShell

Tools for PowerShell V4

範例 1：此範例會在 AWS Lambda 中建立新的 C# (dotnetcore1.0 執行時間） 函數名為 MyFunction，提供從本機檔案系統上 zip 檔案的函數編譯二進位檔 （ 可使用相對或絕對路徑）。C# Lambda 函數會使用 AssemblyName::Namespace.ClassName::MethodName 指派方法來指定函數的處理常式。您應該適當地取代處理常式規格的組件名稱 (不含 .dll 尾碼)、命名空間、類別名稱和方法名稱部分。新函數將透過提供的值來設定環境變數 'envvar1' 和 'envvar2'。

```
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -ZipFilename .\MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

輸出：

```
CodeSha256       : /NgBMd...gq71I=
CodeSize         : 214784
DeadLetterConfig :
Description      : My C# Lambda Function
Environment      : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn      : arn:aws:lambda:us-west-2:123456789012:function:ToUpper
FunctionName     : MyFunction
Handler          : AssemblyName::Namespace.ClassName::MethodName
KMSKeyArn        :
LastModified     : 2016-12-29T23:50:14.207+0000
MemorySize       : 128
Role             : arn:aws:iam::123456789012:role/LambdaFullExecRole
Runtime          : dotnetcore1.0
Timeout          : 3
Version          : $LATEST
VpcConfig        :
```

範例 2：此範例類似於上一個範例，但函數二進位檔會先上傳到 Amazon S3 儲存貯體 (必須與預期的 Lambda 函數位於相同區域)，然後在建立函數時參考產生的 S3 物件。

```
Write-S3Object -BucketName amzn-s3-demo-bucket -Key MyFunctionBinaries.zip -
File .\MyFunctionBinaries.zip
```

```
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -BucketName amzn-s3-demo-bucket `
        -Key MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 CreateFunction。

Tools for PowerShell V5

範例 1：此範例會在 AWS Lambda 中建立新的 C# (dotnetcore1.0 執行時間） 函數名為 MyFunction，提供從本機檔案系統上 zip 檔案的函數編譯二進位檔 （ 可使用相對或絕對路徑）。C# Lambda 函數會使用 AssemblyName::Namespace.ClassName::MethodName 指派方法來指定函數的處理常式。您應該適當地取代處理常式規格的組件名稱 (不含 .dll 尾碼)、命名空間、類別名稱和方法名稱部分。新函數將透過提供的值來設定環境變數 'envvar1' 和 'envvar2'。

```
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -ZipFilename .\MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

輸出：

```
CodeSha256       : /NgBMd...gq71I=
CodeSize         : 214784
DeadLetterConfig :
Description      : My C# Lambda Function
Environment      : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn      : arn:aws:lambda:us-west-2:123456789012:function:ToUpper
FunctionName     : MyFunction
Handler          : AssemblyName::Namespace.ClassName::MethodName
KMSKeyArn        :
LastModified     : 2016-12-29T23:50:14.207+0000
MemorySize       : 128
Role             : arn:aws:iam::123456789012:role/LambdaFullExecRole
```

```
Runtime          : dotnetcore1.0
Timeout          : 3
Version          : $LATEST
VpcConfig        :
```

範例 2：此範例類似於上一個範例，但函數二進位檔會先上傳到 Amazon S3 儲存貯體 (必須與預期的 Lambda 函數位於相同區域)，然後在建立函數時參考產生的 S3 物件。

```
Write-S3Object -BucketName amzn-s3-demo-bucket -Key MyFunctionBinaries.zip -
File .\MyFunctionBinaries.zip
Publish-LMFunction -Description "My C# Lambda Function" `
        -FunctionName MyFunction `
        -BucketName amzn-s3-demo-bucket `
        -Key MyFunctionBinaries.zip `
        -Handler "AssemblyName::Namespace.ClassName::MethodName" `
        -Role "arn:aws:iam::123456789012:role/LambdaFullExecRole" `
        -Runtime dotnetcore1.0 `
        -Environment_Variable @{ "envvar1"="value";"envvar2"="value" }
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V5)》中的 CreateFunction。

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def create_function(
        self, function_name, handler_name, iam_role, deployment_package
```

```
    ):
        """
        Deploys a Lambda function.

        :param function_name: The name of the Lambda function.
        :param handler_name: The fully qualified name of the handler function.
This
                             must include the file name and the function name.
        :param iam_role: The IAM role to use for the function.
        :param deployment_package: The deployment package that contains the
function
                                   code in .zip format.
        :return: The Amazon Resource Name (ARN) of the newly created function.
        """
        try:
            response = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="AWS Lambda doc example",
                Runtime="python3.9",
                Role=iam_role.arn,
                Handler=handler_name,
                Code={"ZipFile": deployment_package},
                Publish=True,
            )
            function_arn = response["FunctionArn"]
            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)
            logger.info(
                "Created function '%s' with ARN: '%s'.",
                function_name,
                response["FunctionArn"],
            )
        except ClientError:
            logger.error("Couldn't create function %s.", function_name)
            raise
        else:
            return function_arn
```

- 如需 API 詳細資訊，請參閱《適用於 Python (Boto3) 的AWS 開發套件 API 參考》中的「CreateFunction」。

Ruby

適用於 Ruby 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Deploys a Lambda function.
  #
  # @param function_name: The name of the Lambda function.
  # @param handler_name: The fully qualified name of the handler function.
  # @param role_arn: The IAM role to use for the function.
  # @param deployment_package: The deployment package that contains the function
 code in .zip format.
  # @return: The Amazon Resource Name (ARN) of the newly created function.
  def create_function(function_name, handler_name, role_arn, deployment_package)
    response = @lambda_client.create_function({
                                                role: role_arn.to_s,
                                                function_name: function_name,
                                                handler: handler_name,
                                                runtime: 'ruby2.7',
                                                code: {
                                                  zip_file: deployment_package
                                                },
                                                environment: {
                                                  variables: {
                                                    'LOG_LEVEL' => 'info'
                                                  }
```

```
                                                  }
                                                })
   @lambda_client.wait_until(:function_active_v2, { function_name:
function_name }) do |w|
     w.max_attempts = 5
     w.delay = 5
   end
   response
 rescue Aws::Lambda::Errors::ServiceException => e
   @logger.error("There was an error creating #{function_name}:\n #{e.message}")
 rescue Aws::Waiters::Errors::WaiterFailed => e
   @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
 end
```

- 如需 API 詳細資訊，請參閱《適用於 Ruby 的 AWS SDK API 參考》中的「CreateFunction」。

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
   /**
    * Create a function, uploading from a zip file.
    */
   pub async fn create_function(&self, zip_file: PathBuf) -> Result<String,
anyhow::Error> {
       let code = self.prepare_function(zip_file, None).await?;

       let key = code.s3_key().unwrap().to_string();

       let role = self.create_role().await.map_err(|e| anyhow!(e))?;

       info!("Created iam role, waiting 15s for it to become active");
```

```
        tokio::time::sleep(Duration::from_secs(15)).await;

        info!("Creating lambda function {}", self.lambda_name);
        let _ = self
            .lambda_client
            .create_function()
            .function_name(self.lambda_name.clone())
            .code(code)
            .role(role.arn())
            .runtime(aws_sdk_lambda::types::Runtime::Providedal2)
            .handler("_unused")
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        self.lambda_client
            .publish_version()
            .function_name(self.lambda_name.clone())
            .send()
            .await?;

        Ok(key)
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
```

```
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
    }
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Rust API 參考》中的 CreateFunction。

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    TRY.
        lo_lmd->createfunction(
            iv_functionname = iv_function_name
            iv_runtime = `python3.9`
            iv_role = iv_role_arn
            iv_handler = iv_handler
            io_code = io_zip_file
            iv_description = 'AWS Lambda code example' ).
        MESSAGE 'Lambda function created.' TYPE 'I'.
      CATCH /aws1/cx_lmdcodesigningcfgno00.
        MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodestorageexcdex.
        MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodeverification00.
        MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
```

```
      CATCH /aws1/cx_lmdinvalidcodesigex.
        MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請參閱《適用於 SAP ABAP 的AWS 開發套件 API 參考》中的 CreateFunction。

Swift

適用於 Swift 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import AWSClientRuntime
import AWSLambda
import Foundation

        do {
            // Read the Zip archive containing the AWS Lambda function.

            let zipUrl = URL(fileURLWithPath: path)
            let zipData = try Data(contentsOf: zipUrl)

            // Create the AWS Lambda function that runs the specified code,
```

```
            // using the name given on the command line. The Lambda function
            // will run using the Amazon Linux 2 runtime.

            _ = try await lambdaClient.createFunction(
                input: CreateFunctionInput(
                    code: LambdaClientTypes.FunctionCode(zipFile: zipData),
                    functionName: functionName,
                    handler: "handle",
                    role: roleArn,
                    runtime: .providedal2
                )
            )
        } catch {
            print("*** Error creating Lambda function:")
            dump(error)
            return false
        }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Swift API reference 中的 CreateFunction。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **DeleteAlias** 與 CLI

下列程式碼範例示範如何使用 DeleteAlias。

CLI

AWS CLI

若要刪除 Lambda 函數的別名

下列 delete-alias 範例會從 my-function Lambda 函數中刪除名為 LIVE 的別名。

```
aws lambda delete-alias \
    --function-name my-function \
    --name LIVE
```

此命令不會產生輸出。

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的設定 Lambda 函數別名。 AWS

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 DeleteAlias。

PowerShell

Tools for PowerShell V4

範例 1：此範例會刪除命令中提到的 Lambda 函數別名。

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 DeleteAlias。

Tools for PowerShell V5

範例 1：此範例會刪除命令中提到的 Lambda 函數別名。

```
Remove-LMAlias -FunctionName "MylambdaFunction123" -Name "NewAlias"
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 DeleteAlias。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

DeleteFunction 搭配 AWS SDK 或 CLI 使用

下列程式碼範例示範如何使用 `DeleteFunction`。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 了解基本概念

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /// <summary>
    /// Delete an AWS Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// delete.</param>
    /// <returns>A Boolean value that indicates the success of the action.</
returns>
    public async Task<bool> DeleteFunctionAsync(string functionName)
    {
        var request = new DeleteFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.DeleteFunctionAsync(request);

        // A return value of NoContent means that the request was processed.
        // In this case, the function was deleted, and the return value
        // is intentionally blank.
        return response.HttpStatusCode == System.Net.HttpStatusCode.NoContent;
    }
```

- 如需 API 詳細資訊，請參閱《適用於 .NET 的 AWS SDK API 參考》中的 DeleteFunction。

C++

適用於 C++ 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

    Aws::Lambda::Model::DeleteFunctionRequest request;
    request.SetFunctionName(LAMBDA_NAME);

    Aws::Lambda::Model::DeleteFunctionOutcome outcome = client.DeleteFunction(
            request);

    if (outcome.IsSuccess()) {
        std::cout << "The lambda function was successfully deleted." <<
std::endl;
    }
    else {
        std::cerr << "Error with Lambda::DeleteFunction. "
                  << outcome.GetError().GetMessage()
                  << std::endl;
    }
```

- 如需 API 詳細資訊，請參閱《適用於 C++ 的 AWS SDK API 參考》中的 DeleteFunction。

CLI

AWS CLI

範例 1：若要依函數名稱刪除 Lambda 函數

下列 delete-function 範例會透過指定函數的名稱來刪除名為 my-function 的 Lambda 函數。

```
aws lambda delete-function \
    --function-name my-function
```

此命令不會產生輸出。

範例 2：若要依函數 ARN 刪除 Lambda 函數

下列 delete-function 範例會透過指定函數的 ARN 來刪除名為 my-function 的 Lambda 函數。

```
aws lambda delete-function \
    --function-name arn:aws:lambda:us-west-2:123456789012:function:my-function
```

此命令不會產生輸出。

範例 3：若要依部分函數 ARN 刪除 Lambda 函數

下列 delete-function 範例會透過指定函數的部分 ARN 來刪除名為 my-function 的 Lambda 函數。

```
aws lambda delete-function \
    --function-name 123456789012:function:my-function
```

此命令不會產生輸出。

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的 AWS Lambda 函數組態。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 DeleteFunction。

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// DeleteFunction deletes the Lambda function specified by functionName.
func (wrapper FunctionWrapper) DeleteFunction(ctx context.Context, functionName
 string) {
 _, err := wrapper.LambdaClient.DeleteFunction(ctx, &lambda.DeleteFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't delete function %v. Here's why: %v\n", functionName, err)
 }
}
```

- 如需 API 詳細資訊，請參閱《適用於 Go 的 AWS SDK API 參考》中的 DeleteFunction。

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
   /**
    * Deletes an AWS Lambda function.
    *
    * @param awsLambda     an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
    * @param functionName  the name of the Lambda function to be deleted
    *
    * @throws LambdaException if an error occurs while deleting the Lambda
function
    */
   public static void deleteLambdaFunction(LambdaClient awsLambda, String
functionName) {
       try {
           DeleteFunctionRequest request = DeleteFunctionRequest.builder()
               .functionName(functionName)
               .build();

           awsLambda.deleteFunction(request);
           System.out.println("The " + functionName + " function was deleted");

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Java 2.x API 參考》中的 DeleteFunction。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
/**
 * @param {string} funcName
 */
const deleteFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new DeleteFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- 如需 API 詳細資訊，請參閱《適用於 JavaScript 的 AWS SDK API 參考》中的 DeleteFunction。

Kotlin

適用於 Kotlin 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
suspend fun delLambdaFunction(myFunctionName: String) {
    val request =
        DeleteFunctionRequest {
            functionName = myFunctionName
        }

    LambdaClient { region = "us-east-1" }.use { awsLambda ->
```

```
        awsLambda.deleteFunction(request)
        println("$myFunctionName was deleted")
    }
}
```

- 如需 API 詳細資訊，請參閱《適用於 Kotlin 的AWS 開發套件 API 參考》中的「DeleteFunction」。

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    public function deleteFunction($functionName)
    {
        return $this->lambdaClient->deleteFunction([
            'FunctionName' => $functionName,
        ]);
    }
```

- 如需 API 詳細資訊，請參閱《適用於 PHP 的 AWS SDK API 參考》中的 DeleteFunction。

PowerShell

Tools for PowerShell V4

範例 1：此範例會刪除 Lambda 函數的特定版本

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 DeleteFunction。

Tools for PowerShell V5

範例 1：此範例會刪除 Lambda 函數的特定版本

```
Remove-LMFunction -FunctionName "MylambdaFunction123" -Qualifier '3'
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V5)》中的 DeleteFunction。

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def delete_function(self, function_name):
        """
        Deletes a Lambda function.

        :param function_name: The name of the function to delete.
        """
        try:
            self.lambda_client.delete_function(FunctionName=function_name)
        except ClientError:
            logger.exception("Couldn't delete function %s.", function_name)
            raise
```

- 如需 API 詳細資訊，請參閱《適用於 Python (Boto3) 的AWS 開發套件 API 參考》中的「DeleteFunction」。

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Deletes a Lambda function.
  # @param function_name: The name of the function to delete.
  def delete_function(function_name)
    print "Deleting function: #{function_name}..."
    @lambda_client.delete_function(
      function_name: function_name
    )
    print 'Done!'.green
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error deleting #{function_name}:\n #{e.message}")
  end
```

- 如需 API 詳細資訊，請參閱《適用於 Ruby 的 AWS SDK API 參考》中的 DeleteFunction。

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
   /** Delete a function and its role, and if possible or necessary, its
associated code object and bucket. */
   pub async fn delete_function(
       &self,
       location: Option<String>,
   ) -> (
       Result<DeleteFunctionOutput, anyhow::Error>,
       Result<DeleteRoleOutput, anyhow::Error>,
       Option<Result<DeleteObjectOutput, anyhow::Error>>,
   ) {
       info!("Deleting lambda function {}", self.lambda_name);
       let delete_function = self
           .lambda_client
           .delete_function()
           .function_name(self.lambda_name.clone())
           .send()
           .await
           .map_err(anyhow::Error::from);

       info!("Deleting iam role {}", self.role_name);
       let delete_role = self
           .iam_client
           .delete_role()
           .role_name(self.role_name.clone())
           .send()
           .await
           .map_err(anyhow::Error::from);

       let delete_object: Option<Result<DeleteObjectOutput, anyhow::Error>> =
           if let Some(location) = location {
               info!("Deleting object {location}");
               Some(
```

```
                    self.s3_client
                        .delete_object()
                        .bucket(self.bucket.clone())
                        .key(location)
                        .send()
                        .await
                        .map_err(anyhow::Error::from),
                )
            } else {
                info!(?location, "Skipping delete object");
                None
            };

        (delete_function, delete_role, delete_object)
    }
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Rust API 參考》中的 DeleteFunction。

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    TRY.
        lo_lmd->deletefunction( iv_functionname = iv_function_name ).
        MESSAGE 'Lambda function deleted.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
```

```
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請參閱《適用於 SAP ABAP 的AWS 開發套件API 參考》中的 DeleteFunction。

Swift

適用於 Swift 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import AWSClientRuntime
import AWSLambda
import Foundation

        do {
            _ = try await lambdaClient.deleteFunction(
                input: DeleteFunctionInput(
                    functionName: "lambda-basics-function"
                )
            )
        } catch {
            print("Error: Unable to delete the function.")
        }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Swift API reference 中的 DeleteFunction。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **DeleteFunctionConcurrency** 與 CLI

下列程式碼範例示範如何使用 DeleteFunctionConcurrency。

CLI

AWS CLI

若要從函數中移除預留並行執行限制

下列 delete-function-concurrency 範例會從 my-function 函數中刪除預留並行執行限制。

```
aws lambda delete-function-concurrency \
    --function-name  my-function
```

此命令不會產生輸出。

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中 Lambda 函數的預留並行。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 DeleteFunctionConcurrency。

PowerShell

Tools for PowerShell V4

範例 1：此範例會移除 Lambda 函數的函數並行。

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 DeleteFunctionConcurrency。

Tools for PowerShell V5

範例 1：此範例會移除 Lambda 函數的函數並行。

```
Remove-LMFunctionConcurrency -FunctionName "MylambdaFunction123"
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 DeleteFunctionConcurrency。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **DeleteProvisionedConcurrencyConfig** 與 CLI

下列程式碼範例示範如何使用 DeleteProvisionedConcurrencyConfig。

CLI

AWS CLI

若要刪除已佈建的並行組態

下列 delete-provisioned-concurrency-config 範例會刪除指定函數之 GREEN 別名的已佈建並行組態。

```
aws lambda delete-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier GREEN
```

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 DeleteProvisionedConcurrencyConfig。

PowerShell

Tools for PowerShell V4

範例 1：此範例會移除特定別名的已佈建並行組態。

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 DeleteProvisionedConcurrencyConfig。

Tools for PowerShell V5

範例 1：此範例會移除特定別名的已佈建並行組態。

```
Remove-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 DeleteProvisionedConcurrencyConfig。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **GetAccountSettings** 與 CLI

下列程式碼範例示範如何使用 GetAccountSettings。

CLI

AWS CLI

擷取 區域中您帳戶的詳細資訊 AWS

下列 get-account-settings 範例顯示帳戶的 Lambda 限制和用量資訊。

```
aws lambda get-account-settings
```

輸出：

```
{
    "AccountLimit": {
       "CodeSizeUnzipped": 262144000,
       "UnreservedConcurrentExecutions": 1000,
       "ConcurrentExecutions": 1000,
       "CodeSizeZipped": 52428800,
       "TotalCodeSize": 80530636800
    },
    "AccountUsage": {
       "FunctionCount": 4,
       "TotalCodeSize": 9426
    }
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的 AWS Lambda 限制。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 GetAccountSettings。

PowerShell

Tools for PowerShell V4

範例 1：此範例會顯示以比較帳戶限制和帳戶用量

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

輸出：

```
TotalCodeSizeLimit TotalCodeSizeUsed
------------------ -----------------
       80530636800          15078795
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 GetAccountSettings。

Tools for PowerShell V5

範例 1：此範例會顯示以比較帳戶限制和帳戶用量

```
Get-LMAccountSetting | Select-Object
 @{Name="TotalCodeSizeLimit";Expression={$_.AccountLimit.TotalCodeSize}},
 @{Name="TotalCodeSizeUsed";Expression={$_.AccountUsage.TotalCodeSize}}
```

輸出：

```
TotalCodeSizeLimit TotalCodeSizeUsed
------------------ -----------------
       80530636800          15078795
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 GetAccountSettings。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **GetAlias** 與 CLI

下列程式碼範例示範如何使用 `GetAlias`。

CLI

AWS CLI

若要擷取函數別名的詳細資訊

下列 get-alias 範例會顯示 my-function Lambda 函數中名為 LIVE 的別名詳細資訊。

```
aws lambda get-alias \
    --function-name my-function \
    --name LIVE
```

輸出：

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "594f41fb-b85f-4c20-95c7-6ca5f2a92c93",
    "Description": "alias for live version of function"
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的設定 Lambda 函數別名。 AWS

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 GetAlias。

PowerShell

Tools for PowerShell V4

範例 1：此範例會擷取特定 Lambda 函數別名的路由組態權重。

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

輸出：

```
AdditionalVersionWeights
------------------------
```

```
{[1, 0.6]}
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 GetAlias。

Tools for PowerShell V5

範例 1：此範例會擷取特定 Lambda 函數別名的路由組態權重。

```
Get-LMAlias -FunctionName "MylambdaFunction123" -Name "newlabel1" -Select
 RoutingConfig
```

輸出：

```
AdditionalVersionWeights
------------------------
{[1, 0.6]}
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 GetAlias。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

`GetFunction` 搭配 AWS SDK 或 CLI 使用

下列程式碼範例示範如何使用 `GetFunction`。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 了解基本概念

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /// <summary>
    /// Gets information about a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function for
    /// which to retrieve information.</param>
    /// <returns>Async Task.</returns>
    public async Task<FunctionConfiguration> GetFunctionAsync(string
functionName)
    {
        var functionRequest = new GetFunctionRequest
        {
            FunctionName = functionName,
        };

        var response = await _lambdaService.GetFunctionAsync(functionRequest);
        return response.Configuration;
    }
```

- 如需 API 詳細資訊，請參閱《適用於 .NET 的 AWS SDK API 參考》中的 GetFunction。

C++

適用於 C++ 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::GetFunctionRequest request;
        request.SetFunctionName(functionName);
```

```
        Aws::Lambda::Model::GetFunctionOutcome outcome =
 client.GetFunction(request);

        if (outcome.IsSuccess()) {
            std::cout << "Function retrieve.\n" <<

 outcome.GetResult().GetConfiguration().Jsonize().View().WriteReadable()
                      << std::endl;
        }
        else {
            std::cerr << "Error with Lambda::GetFunction. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- 如需 API 詳細資訊，請參閱《適用於 C++ 的 AWS SDK API 參考》中的 GetFunction。

CLI

AWS CLI

若要擷取函數相關資訊

下列 `get-function` 範例顯示 `my-function` 函數的相關資訊。

```
aws lambda get-function \
    --function-name  my-function
```

輸出：

```
{
    "Concurrency": {
        "ReservedConcurrentExecutions": 100
    },
    "Code": {
        "RepositoryType": "S3",
        "Location": "https://awslambda-us-west-2-tasks.s3.us-
west-2.amazonaws.com/snapshots/123456789012/my-function..."
    },
    "Configuration": {
        "TracingConfig": {
```

```
            "Mode": "PassThrough"
        },
        "Version": "$LATEST",
        "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
        "FunctionName": "my-function",
        "VpcConfig": {
            "SubnetIds": [],
            "VpcId": "",
            "SecurityGroupIds": []
        },
        "MemorySize": 128,
        "RevisionId": "28f0fb31-5c5c-43d3-8955-03e76c5c1075",
        "CodeSize": 304,
        "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
        "Handler": "index.handler",
        "Role": "arn:aws:iam::123456789012:role/service-role/helloWorldPython-
role-uy3l9qyq",
        "Timeout": 3,
        "LastModified": "2025-09-24T18:20:35.054+0000",
        "Runtime": "nodejs22.x",
        "Description": ""
    }
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的設定 Lambda 函式記憶體。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 GetFunction。

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import (
 "bytes"
```

```
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// GetFunction gets data about the Lambda function specified by functionName.
func (wrapper FunctionWrapper) GetFunction(ctx context.Context, functionName
 string) types.State {
 var state types.State
 funcOutput, err := wrapper.LambdaClient.GetFunction(ctx,
 &lambda.GetFunctionInput{
  FunctionName: aws.String(functionName),
 })
 if err != nil {
  log.Panicf("Couldn't get function %v. Here's why: %v\n", functionName, err)
 } else {
  state = funcOutput.Configuration.State
 }
 return state
}
```

- 如需 API 詳細資訊，請參閱《適用於 Go 的 AWS SDK API 參考》中的 GetFunction。

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /**
     * Retrieves information about an AWS Lambda function.
     *
     * @param awsLambda    an instance of the {@link LambdaClient} class, which
is used to interact with the AWS Lambda service
     * @param functionName the name of the AWS Lambda function to retrieve
information about
     */
    public static void getFunction(LambdaClient awsLambda, String functionName) {
        try {
            GetFunctionRequest functionRequest = GetFunctionRequest.builder()
                .functionName(functionName)
                .build();

            GetFunctionResponse response =
awsLambda.getFunction(functionRequest);
            System.out.println("The runtime of this Lambda function is " +
response.configuration().runtime());

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Java 2.x API 參考》中的 GetFunction。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
const getFunction = (funcName) => {
  const client = new LambdaClient({});
  const command = new GetFunctionCommand({ FunctionName: funcName });
  return client.send(command);
};
```

- 如需 API 詳細資訊，請參閱《適用於 JavaScript 的 AWS SDK API 參考》中的 GetFunction。

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    public function getFunction($functionName)
    {
        return $this->lambdaClient->getFunction([
            'FunctionName' => $functionName,
        ]);
    }
```

- 如需 API 詳細資訊，請參閱《適用於 PHP 的 AWS SDK API 參考》中的 GetFunction。

Python

適用於 Python (Boto3) 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def get_function(self, function_name):
        """
        Gets data about a Lambda function.

        :param function_name: The name of the function.
        :return: The function data.
        """
        response = None
        try:
            response =
 self.lambda_client.get_function(FunctionName=function_name)
        except ClientError as err:
            if err.response["Error"]["Code"] == "ResourceNotFoundException":
                logger.info("Function %s does not exist.", function_name)
            else:
                logger.error(
                    "Couldn't get function %s. Here's why: %s: %s",
                    function_name,
                    err.response["Error"]["Code"],
                    err.response["Error"]["Message"],
                )
                raise
        return response
```

- 如需 API 詳細資訊，請參閱《適用於 Python (Boto3) 的AWS 開發套件 API 參考》中的「GetFunction」。

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Gets data about a Lambda function.
  #
  # @param function_name: The name of the function.
  # @return response: The function data, or nil if no such function exists.
  def get_function(function_name)
    @lambda_client.get_function(
      {
        function_name: function_name
      }
    )
  rescue Aws::Lambda::Errors::ResourceNotFoundException => e
    @logger.debug("Could not find function: #{function_name}:\n #{e.message}")
    nil
  end
```

- 如需 API 詳細資訊，請參閱《適用於 Ruby 的 AWS SDK API 參考》中的 GetFunction。

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /** Get the Lambda function with this Manager's name. */
    pub async fn get_function(&self) -> Result<GetFunctionOutput, anyhow::Error>
{
        info!("Getting lambda function");
        self.lambda_client
            .get_function()
            .function_name(self.lambda_name.clone())
            .send()
            .await
            .map_err(anyhow::Error::from)
    }
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Rust API 參考》中的 GetFunction。

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    TRY.
        oo_result = lo_lmd->getfunction( iv_functionname = iv_function_name ).
" oo_result is returned for testing purposes. "
        MESSAGE 'Lambda function information retrieved.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
```

```
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
 TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請參閱《適用於 SAP ABAP 的AWS 開發套件 API 參考》中的 GetFunction。

Swift

適用於 Swift 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Detect whether or not the AWS Lambda function with the specified name
    /// exists, by requesting its function information.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - name: The name of the AWS Lambda function to find.
    ///
    /// - Returns: `true` if the Lambda function exists. Otherwise `false`.
    func doesLambdaFunctionExist(lambdaClient: LambdaClient, name: String) async
 -> Bool {
        do {
            _ = try await lambdaClient.getFunction(
                input: GetFunctionInput(functionName: name)
            )
        } catch {
```

```
            return false
        }

        return true
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Swift API reference 中的 GetFunction。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **GetFunctionConcurrency** 與 CLI

下列程式碼範例示範如何使用 GetFunctionConcurrency。

CLI

AWS CLI

若要檢視函數的預留並行設定

下列 get-function-concurrency 範例會擷取指定函數的預留並行設定。

```
aws lambda get-function-concurrency \
    --function-name my-function
```

輸出：

```
{
    "ReservedConcurrentExecutions": 250
}
```

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 GetFunctionConcurrency。

PowerShell

Tools for PowerShell V4

範例 1：此範例會取得 Lambda 函數的預留並行

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

輸出：

```
ReservedConcurrentExecutions
----------------------------
100
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 GetFunctionConcurrency。

Tools for PowerShell V5

範例 1：此範例會取得 Lambda 函數的預留並行

```
Get-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -Select *
```

輸出：

```
ReservedConcurrentExecutions
----------------------------
100
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 GetFunctionConcurrency。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **GetFunctionConfiguration** 與 CLI

下列程式碼範例示範如何使用 `GetFunctionConfiguration`。

CLI

AWS CLI

若要擷取 Lambda 函數的版本特定設定

下列 `get-function-configuration` 範例顯示 `my-function` 函數第 2 版的設定。

```
aws lambda get-function-configuration \
    --function-name  my-function:2
```

輸出：

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
    "Handler": "index.handler"
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的 AWS Lambda 函數組態。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 GetFunctionConfiguration。

PowerShell

Tools for PowerShell V4

範例 1：此範例會傳回 Lambda 函數的版本特定組態。

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

輸出：

```
CodeSha256                  : uWOW0R7z+f0VyLuUg7+/D08hkMFsq0SF4seuyUZJ/R8=
CodeSize                    : 1426
DeadLetterConfig            : Amazon.Lambda.Model.DeadLetterConfig
Description                 : Verson 3 to test Aliases
Environment                 : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn                 : arn:aws:lambda:us-
east-1:123456789012:function:MylambdaFunction123
                              :PowershellAlias
FunctionName                : MylambdaFunction123
Handler                     : lambda_function.launch_instance
KMSKeyArn                   :
LastModified                : 2019-12-25T09:52:59.872+0000
LastUpdateStatus            : Successful
LastUpdateStatusReason      :
LastUpdateStatusReasonCode  :
Layers                      : {}
MasterArn                   :
MemorySize                  : 128
RevisionId                  : 5d7de38b-87f2-4260-8f8a-e87280e10c33
Role                        : arn:aws:iam::123456789012:role/service-role/lambda
Runtime                     : python3.8
State                       : Active
StateReason                 :
StateReasonCode             :
Timeout                     : 600
TracingConfig               : Amazon.Lambda.Model.TracingConfigResponse
Version                     : 4
VpcConfig                   : Amazon.Lambda.Model.VpcConfigDetail
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 GetFunctionConfiguration。

Tools for PowerShell V5

範例 1：此範例會傳回 Lambda 函數的版本特定組態。

```
Get-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Qualifier
 "PowershellAlias"
```

輸出：

```
CodeSha256                  : uWOW0R7z+f0VyLuUg7+/D08hkMFsq0SF4seuyUZJ/R8=
CodeSize                    : 1426
DeadLetterConfig            : Amazon.Lambda.Model.DeadLetterConfig
Description                 : Verson 3 to test Aliases
Environment                 : Amazon.Lambda.Model.EnvironmentResponse
FunctionArn                 : arn:aws:lambda:us-
east-1:123456789012:function:MylambdaFunction123
                              :PowershellAlias
FunctionName                : MylambdaFunction123
Handler                     : lambda_function.launch_instance
KMSKeyArn                   :
LastModified                : 2019-12-25T09:52:59.872+0000
LastUpdateStatus            : Successful
LastUpdateStatusReason      :
LastUpdateStatusReasonCode  :
Layers                      : {}
MasterArn                   :
MemorySize                  : 128
RevisionId                  : 5d7de38b-87f2-4260-8f8a-e87280e10c33
Role                        : arn:aws:iam::123456789012:role/service-role/lambda
Runtime                     : python3.8
State                       : Active
StateReason                 :
StateReasonCode             :
Timeout                     : 600
TracingConfig               : Amazon.Lambda.Model.TracingConfigResponse
Version                     : 4
VpcConfig                   : Amazon.Lambda.Model.VpcConfigDetail
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 GetFunctionConfiguration。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **GetPolicy** 與 CLI

下列程式碼範例示範如何使用 GetPolicy。

CLI

AWS CLI

若要擷取函數、版本或別名的資源型 IAM 政策

下列 get-policy 範例顯示 my-function Lambda 函數的政策資訊。

```
aws lambda get-policy \
    --function-name my-function
```

輸出：

```
{
    "Policy": {
        "Version":"2012-10-17",
        "Id":"default",
        "Statement":
        [
            {
                "Sid":"iot-events",
                "Effect":"Allow",
                "Principal": {"Service":"iotevents.amazonaws.com"},
                "Action":"lambda:InvokeFunction",
                "Resource":"arn:aws:lambda:us-west-2:123456789012:function:my-
function"
            }
        ]
    },
    "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668"
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的使用資源型 Lambda 政策。 AWS

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 GetPolicy。

PowerShell

Tools for PowerShell V4

範例 1：此範例顯示 Lambda 函數的函數政策

```
Get-LMPolicy -FunctionName test -Select Policy
```

輸出：

```
{"Version":"2012-10-17",       "Id":"default","Statement":
[{"Sid":"xxxx","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lamb
east-1:123456789102:function:test"}]}
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 GetPolicy。

Tools for PowerShell V5

範例 1：此範例顯示 Lambda 函數的函數政策

```
Get-LMPolicy -FunctionName test -Select Policy
```

輸出：

```
{"Version":"2012-10-17",       "Id":"default","Statement":
[{"Sid":"xxxx","Effect":"Allow","Principal":
{"Service":"sns.amazonaws.com"},"Action":"lambda:InvokeFunction","Resource":"arn:aws:lamb
east-1:123456789102:function:test"}]}
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V5)》中的 GetPolicy。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **GetProvisionedConcurrencyConfig** 與 CLI

下列程式碼範例示範如何使用 `GetProvisionedConcurrencyConfig`。

CLI

AWS CLI

若要檢視已佈建的並行組態

下列 get-provisioned-concurrency-config 範例會顯示指定函數之 BLUE 別名的已佈建並行組態詳情。

```
aws lambda get-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE
```

輸出：

```
{
    "RequestedProvisionedConcurrentExecutions": 100,
    "AvailableProvisionedConcurrentExecutions": 100,
    "AllocatedProvisionedConcurrentExecutions": 100,
    "Status": "READY",
    "LastModified": "2019-12-31T20:28:49+0000"
}
```

- 如需 API 詳細資訊，請參閱《AWS CLI API 參考》中的 GetProvisionedConcurrencyConfig。

PowerShell

Tools for PowerShell V4

範例 1：此範例會取得 Lambda 函數指定別名的已佈建並行組態。

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

輸出：

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
LastModified                             : 2020-01-15T03:21:26+0000
```

```
RequestedProvisionedConcurrentExecutions : 70
Status                                   : IN_PROGRESS
StatusReason                             :
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 GetProvisionedConcurrencyConfig。

Tools for PowerShell V5

範例 1：此範例會取得 Lambda 函數指定別名的已佈建並行組態。

```
C:\>Get-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
Qualifier "NewAlias1"
```

輸出：

```
AllocatedProvisionedConcurrentExecutions : 0
AvailableProvisionedConcurrentExecutions : 0
LastModified                             : 2020-01-15T03:21:26+0000
RequestedProvisionedConcurrentExecutions : 70
Status                                   : IN_PROGRESS
StatusReason                             :
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 GetProvisionedConcurrencyConfig。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

`Invoke` 搭配 AWS SDK 或 CLI 使用

下列程式碼範例示範如何使用 `Invoke`。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 了解基本概念

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /// <summary>
    /// Invoke a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to
    /// invoke.</param
    /// <param name="parameters">The parameter values that will be passed to the
function.</param>
    /// <returns>A System Threading Task.</returns>
    public async Task<string> InvokeFunctionAsync(
        string functionName,
        string parameters)
    {
        var payload = parameters;
        var request = new InvokeRequest
        {
            FunctionName = functionName,
            Payload = payload,
        };

        var response = await _lambdaService.InvokeAsync(request);
        MemoryStream stream = response.Payload;
        string returnValue =
System.Text.Encoding.UTF8.GetString(stream.ToArray());
        return returnValue;
    }
```

- 如需 API 詳細資訊，請參閱《適用於 .NET 的 AWS SDK API 參考》中的 Invoke。

C++

適用於 C++ 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
       Aws::Client::ClientConfiguration clientConfig;
       // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
       // clientConfig.region = "us-east-1";

   Aws::Lambda::LambdaClient client(clientConfig);

       Aws::Lambda::Model::InvokeRequest request;
       request.SetFunctionName(LAMBDA_NAME);
       request.SetLogType(logType);
       std::shared_ptr<Aws::IOStream> payload =
Aws::MakeShared<Aws::StringStream>(
               "FunctionTest");
       *payload << jsonPayload.View().WriteReadable();
       request.SetBody(payload);
       request.SetContentType("application/json");
       Aws::Lambda::Model::InvokeOutcome outcome = client.Invoke(request);

       if (outcome.IsSuccess()) {
           invokeResult = std::move(outcome.GetResult());
           result = true;
           break;
       }

       else {
           std::cerr << "Error with Lambda::InvokeRequest. "
                     << outcome.GetError().GetMessage()
                     << std::endl;
           break;
       }
```

- 如需 API 詳細資訊，請參閱《適用於 C++ 的 AWS SDK API 參考》中的 Invoke。

CLI

AWS CLI

範例 1：若要同步調用 Lambda 函數

下列 invoke 範例會同步調用 my-function 函數。如果您使用 CLI AWS 第 2 版，則需要 cli-binary-format選項。如需詳細資訊，請參閱《AWS 命令列介面使用者指南》中的 AWS CLI 支援的全域命令列選項。

```
aws lambda invoke \
    --function-name my-function \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

輸出：

```
{
    "ExecutedVersion": "$LATEST",
    "StatusCode": 200
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的同步調用 Lambda 函式。

範例 2：若要非同步調用 Lambda 函數

下列 invoke 範例會非同步調用 my-function 函數。如果您使用 CLI AWS 第 2 版，則需要 cli-binary-format選項。如需詳細資訊，請參閱《AWS 命令列介面使用者指南》中的 AWS CLI 支援的全域命令列選項。

```
aws lambda invoke \
    --function-name my-function \
    --invocation-type Event \
    --cli-binary-format raw-in-base64-out \
    --payload '{ "name": "Bob" }' \
    response.json
```

輸出：

```
{
    "StatusCode": 202
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的非同步調用 Lambda 函式。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 Invoke。

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}
```

```
// Invoke invokes the Lambda function specified by functionName, passing the
 parameters
// as a JSON payload. When getLog is true, types.LogTypeTail is specified, which
 tells
// Lambda to include the last few log lines in the returned result.
func (wrapper FunctionWrapper) Invoke(ctx context.Context, functionName string,
 parameters any, getLog bool) *lambda.InvokeOutput {
 logType := types.LogTypeNone
 if getLog {
  logType = types.LogTypeTail
 }
 payload, err := json.Marshal(parameters)
 if err != nil {
  log.Panicf("Couldn't marshal parameters to JSON. Here's why %v\n", err)
 }
 invokeOutput, err := wrapper.LambdaClient.Invoke(ctx, &lambda.InvokeInput{
  FunctionName: aws.String(functionName),
  LogType:      logType,
  Payload:      payload,
 })
 if err != nil {
  log.Panicf("Couldn't invoke function %v. Here's why: %v\n", functionName, err)
 }
 return invokeOutput
}
```

- 如需 API 詳細資訊，請參閱《適用於 Go 的 AWS SDK API 參考》中的 Invoke。

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /**
```

```
    * Invokes a specific AWS Lambda function.
    *
    * @param awsLambda    an instance of {@link LambdaClient} to interact with
the AWS Lambda service
    * @param functionName the name of the AWS Lambda function to be invoked
    */
   public static void invokeFunction(LambdaClient awsLambda, String
functionName) {
       InvokeResponse res;
       try {
           // Need a SdkBytes instance for the payload.
           JSONObject jsonObj = new JSONObject();
           jsonObj.put("inputValue", "2000");
           String json = jsonObj.toString();
           SdkBytes payload = SdkBytes.fromUtf8String(json);

           InvokeRequest request = InvokeRequest.builder()
               .functionName(functionName)
               .payload(payload)
               .build();

           res = awsLambda.invoke(request);
           String value = res.payload().asUtf8String();
           System.out.println(value);

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Java 2.x API 參考》中的 Invoke。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
const invoke = async (funcName, payload) => {
  const client = new LambdaClient({});
  const command = new InvokeCommand({
    FunctionName: funcName,
    Payload: JSON.stringify(payload),
    LogType: LogType.Tail,
  });

  const { Payload, LogResult } = await client.send(command);
  const result = Buffer.from(Payload).toString();
  const logs = Buffer.from(LogResult, "base64").toString();
  return { logs, result };
};
```

- 如需 API 詳細資訊，請參閱《適用於 JavaScript 的 AWS SDK API 參考》中的 Invoke。

Kotlin

適用於 Kotlin 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
suspend fun invokeFunction(functionNameVal: String) {
    val json = """{"inputValue":"1000"}"""
    val byteArray = json.trimIndent().encodeToByteArray()
    val request =
        InvokeRequest {
            functionName = functionNameVal
            logType = LogType.Tail
            payload = byteArray
        }

    LambdaClient { region = "us-west-2" }.use { awsLambda ->
        val res = awsLambda.invoke(request)
        println("${res.payload?.toString(Charsets.UTF_8)}")
        println("The log result is ${res.logResult}")
```

```
    }
}
```

- 如需 API 的詳細資訊，請參閱《適用於 Kotlin 的AWS 開發套件 API 參考》中的「Invoke」。

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    public function invoke($functionName, $params, $logType = 'None')
    {
        return $this->lambdaClient->invoke([
            'FunctionName' => $functionName,
            'Payload' => json_encode($params),
            'LogType' => $logType,
        ]);
    }
```

- 如需 API 詳細資訊，請參閱《適用於 PHP 的 AWS SDK API 參考》中的 Invoke。

Python

適用於 Python (Boto3) 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper:
```

```
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def invoke_function(self, function_name, function_params, get_log=False):
        """
        Invokes a Lambda function.

        :param function_name: The name of the function to invoke.
        :param function_params: The parameters of the function as a dict. This
dict
                                is serialized to JSON before it is sent to
Lambda.
        :param get_log: When true, the last 4 KB of the execution log are
included in
                        the response.
        :return: The response from the function invocation.
        """
        try:
            response = self.lambda_client.invoke(
                FunctionName=function_name,
                Payload=json.dumps(function_params),
                LogType="Tail" if get_log else "None",
            )
            logger.info("Invoked function %s.", function_name)
        except ClientError:
            logger.exception("Couldn't invoke function %s.", function_name)
            raise
        return response
```

- 如需 API 的詳細資訊，請參閱《適用於 Python (Boto3) 的AWS 開發套件 API 參考》中的 Invoke。

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Invokes a Lambda function.
  # @param function_name [String] The name of the function to invoke.
  # @param payload [nil] Payload containing runtime parameters.
  # @return [Object] The response from the function invocation.
  def invoke_function(function_name, payload = nil)
    params = { function_name: function_name }
    params[:payload] = payload unless payload.nil?
    @lambda_client.invoke(params)
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error executing #{function_name}:\n
 #{e.message}")
  end
```

- 如需 API 詳細資訊，請參閱《適用於 Ruby 的 AWS SDK API 參考》中的 Invoke。

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /** Invoke the lambda function using calculator InvokeArgs. */
    pub async fn invoke(&self, args: InvokeArgs) -> Result<InvokeOutput,
 anyhow::Error> {
        info!(?args, "Invoking {}", self.lambda_name);
        let payload = serde_json::to_string(&args)?;
        debug!(?payload, "Sending payload");
        self.lambda_client
            .invoke()
            .function_name(self.lambda_name.clone())
            .payload(Blob::new(payload))
            .send()
            .await
            .map_err(anyhow::Error::from)
    }

fn log_invoke_output(invoke: &InvokeOutput, message: &str) {
    if let Some(payload) = invoke.payload().cloned() {
        let payload = String::from_utf8(payload.into_inner());
        info!(?payload, message);
    } else {
        info!("Could not extract payload")
    }
    if let Some(logs) = invoke.log_result() {
        debug!(?logs, "Invoked function logs")
    } else {
        debug!("Invoked function had no logs")
    }
}
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Rust API 參考》中的 Invoke。

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
   TRY.
       DATA(lv_json) = /aws1/cl_rt_util=>string_to_xstring(
         `{`  &&
           `"action": "increment",`  &&
           `"number": 10` &&
         `}` ).
       oo_result = lo_lmd->invoke(                 " oo_result is returned for
testing purposes. "
                 iv_functionname = iv_function_name
                 iv_payload = lv_json ).
       MESSAGE 'Lambda function invoked.' TYPE 'I'.
     CATCH /aws1/cx_lmdinvparamvalueex.
       MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvrequestcontex.
       MESSAGE 'Unable to parse request body as JSON.' TYPE 'E'.
     CATCH /aws1/cx_lmdinvalidzipfileex.
       MESSAGE 'The deployment package could not be unzipped.' TYPE 'E'.
     CATCH /aws1/cx_lmdrequesttoolargeex.
       MESSAGE 'Invoke request body JSON input limit was exceeded by the request
payload.' TYPE 'E'.
     CATCH /aws1/cx_lmdresourceconflictex.
       MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
     CATCH /aws1/cx_lmdresourcenotfoundex.
       MESSAGE 'The requested resource does not exist.' TYPE 'E'.
     CATCH /aws1/cx_lmdserviceexception.
       MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
     CATCH /aws1/cx_lmdtoomanyrequestsex.
       MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
     CATCH /aws1/cx_lmdunsuppedmediatyp00.
```

```
        MESSAGE 'Invoke request body does not have JSON as its content type.'
TYPE 'E'.
   ENDTRY.
```

- 如需 API 詳細資訊，請參閱《適用於 SAP ABAP 的AWS SDK API 參考》中的 Invoke。

Swift

適用於 Swift 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Invoke the Lambda function to increment a value.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `IAMClient` to use.
    ///   - number: The number to increment.
    ///
    /// - Throws: `ExampleError.noAnswerReceived`, `ExampleError.invokeError`
    ///
    /// - Returns: An integer number containing the incremented value.
    func invokeIncrement(lambdaClient: LambdaClient, number: Int) async throws ->
 Int {
        do {
            let incRequest = IncrementRequest(action: "increment", number:
 number)
            let incData = try! JSONEncoder().encode(incRequest)

            // Invoke the lambda function.

            let invokeOutput = try await lambdaClient.invoke(
                input: InvokeInput(
                    functionName: "lambda-basics-function",
```

```
                    payload: incData
                )
            )

            let response = try! JSONDecoder().decode(Response.self,
from:invokeOutput.payload!)

            guard let answer = response.answer else {
                throw ExampleError.noAnswerReceived
            }
            return answer

        } catch {
            throw ExampleError.invokeError
        }
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Swift API reference 中的 Invoke。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

`ListFunctions` 搭配 AWS SDK 或 CLI 使用

下列程式碼範例示範如何使用 `ListFunctions`。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 了解基本概念

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /// <summary>
    /// Get a list of Lambda functions.
    /// </summary>
    /// <returns>A list of FunctionConfiguration objects.</returns>
    public async Task<List<FunctionConfiguration>> ListFunctionsAsync()
    {
        var functionList = new List<FunctionConfiguration>();

        var functionPaginator =
            _lambdaService.Paginators.ListFunctions(new ListFunctionsRequest());
        await foreach (var function in functionPaginator.Functions)
        {
            functionList.Add(function);
        }

        return functionList;
    }
```

- 如需 API 詳細資訊，請參閱《適用於 .NET 的 AWS SDK API 參考》中的 ListFunctions。

C++

適用於 C++ 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

    std::vector<Aws::String> functions;
    Aws::String marker;
```

```
   do {
       Aws::Lambda::Model::ListFunctionsRequest request;
       if (!marker.empty()) {
           request.SetMarker(marker);
       }

       Aws::Lambda::Model::ListFunctionsOutcome outcome = client.ListFunctions(
               request);

       if (outcome.IsSuccess()) {
           const Aws::Lambda::Model::ListFunctionsResult &result =
outcome.GetResult();
           std::cout << result.GetFunctions().size()
                     << " lambda functions were retrieved." << std::endl;

           for (const Aws::Lambda::Model::FunctionConfiguration
&functionConfiguration: result.GetFunctions()) {
               functions.push_back(functionConfiguration.GetFunctionName());
               std::cout << functions.size() << "  "
                         << functionConfiguration.GetDescription() << std::endl;
               std::cout << "   "
                         <<
Aws::Lambda::Model::RuntimeMapper::GetNameForRuntime(
                                 functionConfiguration.GetRuntime()) << ": "
                         << functionConfiguration.GetHandler()
                         << std::endl;
           }
           marker = result.GetNextMarker();
       }
       else {
           std::cerr << "Error with Lambda::ListFunctions. "
                     << outcome.GetError().GetMessage()
                     << std::endl;
       }
   } while (!marker.empty());
```

- 如需 API 詳細資訊，請參閱《適用於 C++ 的 AWS SDK API 參考》中的 ListFunctions。

CLI

AWS CLI

若要擷取 Lambda 函數的清單

下列 `list-functions` 範例會顯示目前使用者的所有函數清單。

```
aws lambda list-functions
```

輸出：

```
{
    "Functions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
            "FunctionName": "helloworld",
            "MemorySize": 128,
            "RevisionId": "1718e831-badf-4253-9518-d0644210af7b",
            "CodeSize": 294,
            "FunctionArn": "arn:aws:lambda:us-
west-2:123456789012:function:helloworld",
            "Handler": "helloworld.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-
role-zgur6bf4",
            "Timeout": 3,
            "LastModified": "2025-09-23T18:32:33.857+0000",
            "Runtime": "nodejs22.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
```

```
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2025-10-01T16:47:28.490+0000",
            "Runtime": "nodejs22.x",
            "Description": ""
        },
        {
            "Layers": [
                {
                    "CodeSize": 41784542,
                    "Arn": "arn:aws:lambda:us-
west-2:420165488524:layer:AWSLambda-Python37-SciPy1x:2"
                },
                {
                    "CodeSize": 4121,
                    "Arn": "arn:aws:lambda:us-
west-2:123456789012:layer:pythonLayer:1"
                }
            ],
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "ZQukCqxtkqFgyF2cU41Avj99TKQ/hNihPtDtRcc08mI=",
            "FunctionName": "my-python-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 128,
            "RevisionId": "80b4eabc-acf7-4ea8-919a-e874c213707d",
            "CodeSize": 299,
```

```
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
python-function",
            "Handler": "lambda_function.lambda_handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/my-python-
function-role-z5g7dr6n",
            "Timeout": 3,
            "LastModified": "2025-10-01T19:40:41.643+0000",
            "Runtime": "python3.11",
            "Description": ""
        }
    ]
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的設定 Lambda 函式記憶體。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 ListFunctions。

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
```

```
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// ListFunctions lists up to maxItems functions for the account. This function
 uses a
// lambda.ListFunctionsPaginator to paginate the results.
func (wrapper FunctionWrapper) ListFunctions(ctx context.Context, maxItems int)
 []types.FunctionConfiguration {
 var functions []types.FunctionConfiguration
 paginator := lambda.NewListFunctionsPaginator(wrapper.LambdaClient,
 &lambda.ListFunctionsInput{
  MaxItems: aws.Int32(int32(maxItems)),
 })
 for paginator.HasMorePages() && len(functions) < maxItems {
  pageOutput, err := paginator.NextPage(ctx)
  if err != nil {
   log.Panicf("Couldn't list functions for your account. Here's why: %v\n", err)
  }
  functions = append(functions, pageOutput.Functions...)
 }
 return functions
}
```

- 如需 API 詳細資訊，請參閱《適用於 Go 的 AWS SDK API 參考》中的 ListFunctions。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
const listFunctions = () => {
```

```
  const client = new LambdaClient({});
  const command = new ListFunctionsCommand({});

  return client.send(command);
};
```

- 如需 API 詳細資訊，請參閱《適用於 JavaScript 的 AWS SDK API 參考》中的 ListFunctions。

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    public function listFunctions($maxItems = 50, $marker = null)
    {
        if (is_null($marker)) {
            return $this->lambdaClient->listFunctions([
                'MaxItems' => $maxItems,
            ]);
        }

        return $this->lambdaClient->listFunctions([
            'Marker' => $marker,
            'MaxItems' => $maxItems,
        ]);
    }
```

- 如需 API 詳細資訊，請參閱《適用於 PHP 的 AWS SDK API 參考》中的 ListFunctions。

PowerShell

Tools for PowerShell V4

範例 1：此範例會顯示具有已排序程式碼大小的所有 Lambda 函數

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

輸出：

```
FunctionName                                                    Runtime   Timeout
 CodeSize
------------                                                    -------   -------
 --------
test                                                            python2.7       3
   243
MylambdaFunction123                                             python3.8     600
   659
myfuncpython1                                                   python3.8     303
   675
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 ListFunctions。

Tools for PowerShell V5

範例 1：此範例會顯示具有已排序程式碼大小的所有 Lambda 函數

```
Get-LMFunctionList | Sort-Object -Property CodeSize | Select-Object FunctionName,
 RunTime, Timeout, CodeSize
```

輸出：

```
FunctionName                                                    Runtime   Timeout
 CodeSize
------------                                                    -------   -------
 --------
test                                                            python2.7       3
   243
MylambdaFunction123                                             python3.8     600
   659
```

```
myfuncpython1                                                    python3.8      303
   675
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V5)》中的 ListFunctions。

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def list_functions(self):
        """
        Lists the Lambda functions for the current account.
        """
        try:
            func_paginator = self.lambda_client.get_paginator("list_functions")
            for func_page in func_paginator.paginate():
                for func in func_page["Functions"]:
                    print(func["FunctionName"])
                    desc = func.get("Description")
                    if desc:
                        print(f"\t{desc}")
                    print(f"\t{func['Runtime']}: {func['Handler']}")
        except ClientError as err:
            logger.error(
                "Couldn't list functions. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
```

```
            raise
```

- 如需 API 詳細資訊，請參閱《適用於 Python (Boto3) 的AWS 開發套件 API 參考》中的「ListFunctions」。

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Lists the Lambda functions for the current account.
  def list_functions
    functions = []
    @lambda_client.list_functions.each do |response|
      response['functions'].each do |function|
        functions.append(function['function_name'])
      end
    end
    functions
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error listing functions:\n #{e.message}")
  end
```

- 如需 API 詳細資訊，請參閱《適用於 Ruby 的 AWS SDK API 參考》中的 ListFunctions。

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
   /** List all Lambda functions in the current Region. */
   pub async fn list_functions(&self) -> Result<ListFunctionsOutput,
anyhow::Error> {
       info!("Listing lambda functions");
       self.lambda_client
           .list_functions()
           .send()
           .await
           .map_err(anyhow::Error::from)
   }
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Rust API 參考》中的 ListFunctions。

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
   TRY.
       oo_result = lo_lmd->listfunctions( ).       " oo_result is returned for
testing purposes. "
```

```
        DATA(lt_functions) = oo_result->get_functions( ).
        MESSAGE 'Retrieved list of Lambda functions.' TYPE 'I'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
 TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請參閱《適用於 SAP ABAP 的AWS SDK API 參考》中的 ListFunctions。

Swift

適用於 Swift 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Returns an array containing the names of all AWS Lambda functions
    /// available to the user.
    ///
    /// - Parameter lambdaClient: The `IAMClient` to use.
    ///
    /// - Throws: `ExampleError.listFunctionsError`
    ///
    /// - Returns: An array of lambda function name strings.
    func getFunctionNames(lambdaClient: LambdaClient) async throws -> [String] {
        let pages = lambdaClient.listFunctionsPaginated(
            input: ListFunctionsInput()
        )
```

```
        var functionNames: [String] = []

        for try await page in pages {
            guard let functions = page.functions else {
                throw ExampleError.listFunctionsError
            }

            for function in functions {
                functionNames.append(function.functionName ?? "<unknown>")
            }
        }

        return functionNames
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Swift API reference 中的 ListFunctions。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **ListProvisionedConcurrencyConfigs** 與 CLI

下列程式碼範例示範如何使用 `ListProvisionedConcurrencyConfigs`。

CLI

AWS CLI

若要取得已佈建並行組態的清單。

下列 `list-provisioned-concurrency-configs` 範例會列出指定函數的已佈建並行組態。

```
aws lambda list-provisioned-concurrency-configs \
    --function-name my-function
```

輸出：

```
{
```

```
    "ProvisionedConcurrencyConfigs": [
        {
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function:GREEN",
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
            "Status": "READY",
            "LastModified": "2019-12-31T20:29:00+0000"
        },
        {
            "FunctionArn": "arn:aws:lambda:us-east-2:123456789012:function:my-
function:BLUE",
            "RequestedProvisionedConcurrentExecutions": 100,
            "AvailableProvisionedConcurrentExecutions": 100,
            "AllocatedProvisionedConcurrentExecutions": 100,
            "Status": "READY",
            "LastModified": "2019-12-31T20:28:49+0000"
        }
    ]
}
```

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 ListProvisionedConcurrencyConfigs。

PowerShell

Tools for PowerShell V4

範例 1：此範例會擷取 Lambda 函數的已佈建並行組態清單。

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 ListProvisionedConcurrencyConfigs。

Tools for PowerShell V5

範例 1：此範例會擷取 Lambda 函數的已佈建並行組態清單。

```
Get-LMProvisionedConcurrencyConfigList -FunctionName "MylambdaFunction123"
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 ListProvisionedConcurrencyConfigs。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **ListTags** 與 CLI

下列程式碼範例示範如何使用 `ListTags`。

CLI

AWS CLI

若要擷取 Lambda 函數的標籤清單

下列 `list-tags` 範例顯示連接至 `my-function` Lambda 函數的標籤。

```
aws lambda list-tags \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function
```

輸出：

```
{
    "Tags": {
        "Category": "Web Tools",
        "Department": "Sales"
    }
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的標記 Lambda 函數。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 ListTags。

PowerShell

Tools for PowerShell V4

範例 1：擷取目前在指定函數中設定的標籤及其值。

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

輸出：

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
Washington Olympia
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 ListTags。

Tools for PowerShell V5

範例 1：擷取目前在指定函數中設定的標籤及其值。

```
Get-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction"
```

輸出：

```
Key        Value
---        -----
California Sacramento
Oregon     Salem
Washington Olympia
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 ListTags。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **ListVersionsByFunction** 與 CLI

下列程式碼範例示範如何使用 `ListVersionsByFunction`。

CLI

AWS CLI

若要擷取函數的版本清單

下列 `list-versions-by-function` 範例會顯示 `my-function` Lambda 函數的版本清單。

```
aws lambda list-versions-by-function \
    --function-name my-function
```

輸出：

```
{
    "Versions": [
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "$LATEST",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "93017fc9-59cb-41dc-901b-4845ce4bf668",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:$LATEST",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-10-01T16:47:28.490+0000",
            "Runtime": "nodejs10.x",
            "Description": ""
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
```

```
            },
            "Version": "1",
            "CodeSha256": "5tT2qgzYUHoqwR616pZ2dpkn/0J1FrzJmlKidWaaCgk=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "949c8914-012e-4795-998c-e467121951b1",
            "CodeSize": 304,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:1",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-09-26T20:28:40.438+0000",
            "Runtime": "nodejs10.x",
            "Description": "new version"
        },
        {
            "TracingConfig": {
                "Mode": "PassThrough"
            },
            "Version": "2",
            "CodeSha256": "sU0cJ2/hOZevwV/lTxCuQqK3gDZP3i8gUoqUUVRmY6E=",
            "FunctionName": "my-function",
            "VpcConfig": {
                "SubnetIds": [],
                "VpcId": "",
                "SecurityGroupIds": []
            },
            "MemorySize": 256,
            "RevisionId": "cd669f21-0f3d-4e1c-9566-948837f2e2ea",
            "CodeSize": 266,
            "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:2",
            "Handler": "index.handler",
            "Role": "arn:aws:iam::123456789012:role/service-role/
helloWorldPython-role-uy3l9qyq",
            "Timeout": 3,
            "LastModified": "2019-10-01T16:47:28.490+0000",
```

```
            "Runtime": "nodejs10.x",
            "Description": "newer version"
        }
    ]
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的設定 Lambda 函數別名。 AWS

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 ListVersionsByFunction。

PowerShell

Tools for PowerShell V4

範例 1：此範例會傳回每個 Lambda 函數版本的版本特定組態清單。

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

輸出：

```
FunctionName        Runtime   MemorySize Timeout CodeSize LastModified
      RoleName
------------        -------   ---------- ------- -------- ------------
      --------
MylambdaFunction123 python3.8        128     600      659
 2020-01-10T03:20:56.390+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:19:02.238+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:39:36.779+0000 lambda
MylambdaFunction123 python3.8        128     600     1426
 2019-12-25T09:52:59.872+0000 lambda
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 ListVersionsByFunction。

Tools for PowerShell V5

範例 1：此範例會傳回每個 Lambda 函數版本的版本特定組態清單。

```
Get-LMVersionsByFunction -FunctionName "MylambdaFunction123"
```

輸出：

```
FunctionName        Runtime   MemorySize Timeout CodeSize LastModified
      RoleName
------------        -------   ---------- ------- -------- ------------
      --------
MylambdaFunction123 python3.8        128     600      659
 2020-01-10T03:20:56.390+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:19:02.238+0000 lambda
MylambdaFunction123 python3.8        128       5     1426
 2019-12-25T09:39:36.779+0000 lambda
MylambdaFunction123 python3.8        128     600     1426
 2019-12-25T09:52:59.872+0000 lambda
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 ListVersionsByFunction。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **PublishVersion** 與 CLI

下列程式碼範例示範如何使用 PublishVersion。

CLI

AWS CLI

若要建立新版本的函數

以下 publish-version 範例會發布新版本的 my-function Lambda 函數。

```
aws lambda publish-version \
    --function-name my-function
```

輸出：

```
{
    "TracingConfig": {
        "Mode": "PassThrough"
```

```
    },
    "CodeSha256": "dBG9m8SGdmlEjw/JYXlhhvCrAv5TxvXsbL/RMr0fT/I=",
    "FunctionName": "my-function",
    "CodeSize": 294,
    "RevisionId": "f31d3d39-cc63-4520-97d4-43cd44c94c20",
    "MemorySize": 128,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:3",
    "Version": "2",
    "Role": "arn:aws:iam::123456789012:role/service-role/MyTestFunction-role-
zgur6bf4",
    "Timeout": 3,
    "LastModified": "2019-09-23T18:32:33.857+0000",
    "Handler": "my-function.handler",
    "Runtime": "nodejs10.x",
    "Description": ""
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的設定 Lambda 函數別名。 AWS

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 PublishVersion。

PowerShell

Tools for PowerShell V4

範例 1：此範例會為 Lambda 函數程式碼的現有快照建立版本

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 PublishVersion。

Tools for PowerShell V5

範例 1：此範例會為 Lambda 函數程式碼的現有快照建立版本

```
Publish-LMVersion -FunctionName "MylambdaFunction123" -Description "Publishing
 Existing Snapshot of function code as a  new version through Powershell"
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 PublishVersion。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **PutFunctionConcurrency** 與 CLI

下列程式碼範例示範如何使用 PutFunctionConcurrency。

CLI

AWS CLI

若要設定函數的預留並行限制

下列 put-function-concurrency 範例會設定 my-function 函數的 100 個預留並行執行。

```
aws lambda put-function-concurrency \
    --function-name  my-function  \
    --reserved-concurrent-executions 100
```

輸出：

```
{
    "ReservedConcurrentExecutions": 100
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中 Lambda 函數的預留並行。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 PutFunctionConcurrency。

PowerShell

Tools for PowerShell V4

範例 1：此範例會整體套用函數的並行設定。

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 PutFunctionConcurrency。

Tools for PowerShell V5

範例 1：此範例會整體套用函數的並行設定。

```
Write-LMFunctionConcurrency -FunctionName "MylambdaFunction123" -
ReservedConcurrentExecution 100
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 PutFunctionConcurrency。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **PutProvisionedConcurrencyConfig** 與 CLI

下列程式碼範例示範如何使用 PutProvisionedConcurrencyConfig。

CLI

AWS CLI

若要配置佈建並行

下列 put-provisioned-concurrency-config 範例會為指定函數的 BLUE 別名配置 100 個佈建並行。

```
aws lambda put-provisioned-concurrency-config \
    --function-name my-function \
    --qualifier BLUE \
    --provisioned-concurrent-executions 100
```

輸出：

```
{
    "Requested ProvisionedConcurrentExecutions": 100,
    "Allocated ProvisionedConcurrentExecutions": 0,
    "Status": "IN_PROGRESS",
    "LastModified": "2019-11-21T19:32:12+0000"
}
```

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 PutProvisionedConcurrencyConfig。

PowerShell

Tools for PowerShell V4

範例 1：此範例會將佈建並行組態新增至函數別名

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 PutProvisionedConcurrencyConfig。

Tools for PowerShell V5

範例 1：此範例會將佈建並行組態新增至函數別名

```
Write-LMProvisionedConcurrencyConfig -FunctionName "MylambdaFunction123" -
ProvisionedConcurrentExecution 20 -Qualifier "NewAlias1"
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 PutProvisionedConcurrencyConfig。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **RemovePermission** 與 CLI

下列程式碼範例示範如何使用 `RemovePermission`。

CLI

AWS CLI

若要從現有的 Lambda 函數中移除許可

下列 `remove-permission` 範例會移除許可以調用名為 `my-function` 的函數。

```
aws lambda remove-permission \
```

```
    --function-name my-function \
    --statement-id sns
```

此命令不會產生輸出。

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的使用資源型 Lambda 政策。 AWS

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 RemovePermission。

PowerShell

Tools for PowerShell V4

範例 1：此範例會移除 Lambda 函數之指定 StatementId 的函數政策。

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 RemovePermission。

Tools for PowerShell V5

範例 1：此範例會移除 Lambda 函數之指定 StatementId 的函數政策。

```
$policy =  Get-LMPolicy -FunctionName "MylambdaFunction123" -Select Policy |
 ConvertFrom-Json| Select-Object -ExpandProperty Statement
Remove-LMPermission -FunctionName "MylambdaFunction123" -StatementId
 $policy[0].Sid
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 RemovePermission。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 `TagResource` 與 CLI

下列程式碼範例示範如何使用 `TagResource`。

CLI

AWS CLI

若要將標籤新增至現有 Lambda 函數

下列 `tag-resource` 範例會將索引鍵名稱為 `DEPARTMENT` 且值為 `Department A` 的標籤新增至指定的 Lambda 函數。

```
aws lambda tag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tags "DEPARTMENT=Department A"
```

此命令不會產生輸出。

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的標記 Lambda 函數。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 TagResource。

PowerShell

Tools for PowerShell V4

範例 1：將三個標籤 (華盛頓、奧勒岡和加利佛尼亞) 及其相關聯的值新增至其 ARN 識別的指定函數。

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 TagResource。

Tools for PowerShell V5

範例 1：將三個標籤 (華盛頓、奧勒岡和加利佛尼亞) 及其相關聯的值新增至其 ARN 識別的指定函數。

```
Add-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -Tag @{ "Washington" = "Olympia";
 "Oregon" = "Salem"; "California" = "Sacramento" }
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 TagResource。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **UntagResource** 與 CLI

下列程式碼範例示範如何使用 UntagResource。

CLI

AWS CLI

若要從現有的 Lambda 函數中移除標籤

下列 untag-resource 範例會從 my-function Lambda 函數中移除索引鍵名稱為 DEPARTMENT 的標籤。

```
aws lambda untag-resource \
    --resource arn:aws:lambda:us-west-2:123456789012:function:my-function \
    --tag-keys DEPARTMENT
```

此命令不會產生輸出。

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的標記 Lambda 函數。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 UntagResource。

PowerShell

Tools for PowerShell V4

範例 1：從函數中移除提供的標籤。除非已指定 -Force 切換，否則 cmdlet 會提示進行確認。對服務進行單一呼叫以移除標籤。

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

範例 2：從函數中移除提供的標籤。除非已指定 -Force 切換，否則 cmdlet 會提示進行確認。按照提供的標籤對服務進行呼叫後。

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 UntagResource。

Tools for PowerShell V5

範例 1：從函數中移除提供的標籤。除非已指定 -Force 切換，否則 cmdlet 會提示進行確認。對服務進行單一呼叫以移除標籤。

```
Remove-LMResourceTag -Resource "arn:aws:lambda:us-
west-2:123456789012:function:MyFunction" -TagKey
 "Washington","Oregon","California"
```

範例 2：從函數中移除提供的標籤。除非已指定 -Force 切換，否則 cmdlet 會提示進行確認。按照提供的標籤對服務進行呼叫後。

```
"Washington","Oregon","California" | Remove-LMResourceTag -Resource
 "arn:aws:lambda:us-west-2:123456789012:function:MyFunction"
```

- 如需 API 詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 UntagResource。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

搭配使用 **UpdateAlias** 與 CLI

下列程式碼範例示範如何使用 UpdateAlias。

CLI

AWS CLI

若要更新函數別名

下列 update-alias 範例會更新名為 LIVE 的別名，以指向 my-function Lambda 函數的第 3 版。

```
aws lambda update-alias \
    --function-name my-function \
    --function-version 3 \
    --name LIVE
```

輸出：

```
{
    "FunctionVersion": "3",
    "Name": "LIVE",
    "AliasArn": "arn:aws:lambda:us-west-2:123456789012:function:my-
function:LIVE",
    "RevisionId": "594f41fb-b85f-4c20-95c7-6ca5f2a92c93",
    "Description": "alias for live version of function"
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的設定 Lambda 函數別名。 AWS

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 UpdateAlias。

PowerShell

Tools for PowerShell V4

範例 1：此範例會更新現有 Lambda 函數別名的組態。它會更新 RoutingConfiguration 值，將 60% (0.6) 的流量轉移到第 1 版

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 UpdateAlias。

Tools for PowerShell V5

範例 1：此範例會更新現有 Lambda 函數別名的組態。它會更新 RoutingConfiguration 值，將 60% (0.6) 的流量轉移到第 1 版

```
Update-LMAlias -FunctionName "MylambdaFunction123" -Description
 " Alias for version 2" -FunctionVersion 2 -Name "newlabel1" -
RoutingConfig_AdditionalVersionWeight @{Name="1";Value="0.6}
```

- 如需 API 的詳細資訊，請參閱 AWS Tools for PowerShell Cmdlet Reference (V5) 中的 UpdateAlias。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

`UpdateFunctionCode` 搭配 AWS SDK 或 CLI 使用

下列程式碼範例示範如何使用 `UpdateFunctionCode`。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 了解基本概念

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /// <summary>
    /// Update an existing Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the Lambda function to update.</
param>
    /// <param name="bucketName">The bucket where the zip file containing
    /// the Lambda function code is stored.</param>
    /// <param name="key">The key name of the source code file.</param>
    /// <returns>Async Task.</returns>
    public async Task UpdateFunctionCodeAsync(
        string functionName,
```

```
        string bucketName,
        string key)
    {
        var functionCodeRequest = new UpdateFunctionCodeRequest
        {
            FunctionName = functionName,
            Publish = true,
            S3Bucket = bucketName,
            S3Key = key,
        };

        var response = await
_lambdaService.UpdateFunctionCodeAsync(functionCodeRequest);
        Console.WriteLine($"The Function was last modified at
{response.LastModified}.");
    }
```

- 如需 API 詳細資訊，請參閱《適用於 .NET 的 AWS SDK API 參考》中的 UpdateFunctionCode。

C++

適用於 C++ 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);

        Aws::Lambda::Model::UpdateFunctionCodeRequest request;
        request.SetFunctionName(LAMBDA_NAME);
```

```
        std::ifstream ifstream(CALCULATOR_LAMBDA_CODE.c_str(),
                               std::ios_base::in | std::ios_base::binary);
        if (!ifstream.is_open()) {
            std::cerr << "Error opening file " << INCREMENT_LAMBDA_CODE << "." <<
 std::endl;

#if USE_CPP_LAMBDA_FUNCTION
            std::cerr
                    << "The cpp Lambda function must be built following the
 instructions in the cpp_lambda/README.md file. "
                    << std::endl;
#endif
            deleteLambdaFunction(client);
            deleteIamRole(clientConfig);
            return false;
        }

        Aws::StringStream buffer;
        buffer << ifstream.rdbuf();
        request.SetZipFile(
                Aws::Utils::ByteBuffer((unsigned char *) buffer.str().c_str(),
                                       buffer.str().length()));
        request.SetPublish(true);

        Aws::Lambda::Model::UpdateFunctionCodeOutcome outcome =
 client.UpdateFunctionCode(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda code was successfully updated." <<
 std::endl;
        }
        else {
            std::cerr << "Error with Lambda::UpdateFunctionCode. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- 如需 API 詳細資訊，請參閱《適用於 C++ 的 AWS SDK API 參考》中的 UpdateFunctionCode。

CLI

AWS CLI

若要更新 Lambda 函數的程式碼

下列 update-function-code 範例會使用指定 zip 檔案的內容替換 my-function 函數未發布 ($LATEST) 版本的程式碼。

```
aws lambda update-function-code \
    --function-name  my-function \
    --zip-file fileb://my-function.zip
```

輸出：

```
{
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的 AWS Lambda 函數組態。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 UpdateFunctionCode。

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import (
 "bytes"
 "context"
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// UpdateFunctionCode updates the code for the Lambda function specified by
 functionName.
// The existing code for the Lambda function is entirely replaced by the code in
 the
// zipPackage buffer. After the update action is called, a
 lambda.FunctionUpdatedV2Waiter
// is used to wait until the update is successful.
func (wrapper FunctionWrapper) UpdateFunctionCode(ctx context.Context,
 functionName string, zipPackage *bytes.Buffer) types.State {
 var state types.State
```

```
 _, err := wrapper.LambdaClient.UpdateFunctionCode(ctx,
 &lambda.UpdateFunctionCodeInput{
  FunctionName: aws.String(functionName), ZipFile: zipPackage.Bytes(),
 })
 if err != nil {
  log.Panicf("Couldn't update code for function %v. Here's why: %v\n",
 functionName, err)
 } else {
  waiter := lambda.NewFunctionUpdatedV2Waiter(wrapper.LambdaClient)
  funcOutput, err := waiter.WaitForOutput(ctx, &lambda.GetFunctionInput{
   FunctionName: aws.String(functionName)}, 1*time.Minute)
  if err != nil {
   log.Panicf("Couldn't wait for function %v to be active. Here's why: %v\n",
 functionName, err)
  } else {
   state = funcOutput.Configuration.State
  }
 }
 return state
}
```

- 如需 API 詳細資訊，請參閱《適用於 Go 的 AWS SDK API 參考》中的 UpdateFunctionCode。

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /**
     * Updates the code for an AWS Lambda function.
     *
     * @param awsLambda  the AWS Lambda client
     * @param functionName the name of the Lambda function to update
```

```
    * @param bucketName the name of the S3 bucket where the function code is
located
    * @param key the key (file name) of the function code in the S3 bucket
    * @throws LambdaException if there is an error updating the function code
    */
   public static void updateFunctionCode(LambdaClient awsLambda, String
functionName, String bucketName, String key) {
       try {
           LambdaWaiter waiter = awsLambda.waiter();
           UpdateFunctionCodeRequest functionCodeRequest =
UpdateFunctionCodeRequest.builder()
                .functionName(functionName)
                .publish(true)
                .s3Bucket(bucketName)
                .s3Key(key)
                .build();

           UpdateFunctionCodeResponse response =
awsLambda.updateFunctionCode(functionCodeRequest);
           GetFunctionConfigurationRequest getFunctionConfigRequest =
GetFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .build();

           WaiterResponse<GetFunctionConfigurationResponse> waiterResponse =
waiter
                .waitUntilFunctionUpdated(getFunctionConfigRequest);
           waiterResponse.matched().response().ifPresent(System.out::println);
           System.out.println("The last modified value is " +
response.lastModified());

       } catch (LambdaException e) {
           System.err.println(e.getMessage());
           System.exit(1);
       }
   }
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Java 2.x API 參考》中的 UpdateFunctionCode。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
const updateFunctionCode = async (funcName, newFunc) => {
  const client = new LambdaClient({});
  const code = await readFile(`${dirname}../functions/${newFunc}.zip`);
  const command = new UpdateFunctionCodeCommand({
    ZipFile: code,
    FunctionName: funcName,
    Architectures: [Architecture.arm64],
    Handler: "index.handler", // Required when sending a .zip file
    PackageType: PackageType.Zip, // Required when sending a .zip file
    Runtime: Runtime.nodejs16x, // Required when sending a .zip file
  });

  return client.send(command);
};
```

- 如需 API 詳細資訊，請參閱《適用於 JavaScript 的 AWS SDK API 參考》中的 UpdateFunctionCode。

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    public function updateFunctionCode($functionName, $s3Bucket, $s3Key)
```

```
    {
        return $this->lambdaClient->updateFunctionCode([
            'FunctionName' => $functionName,
            'S3Bucket' => $s3Bucket,
            'S3Key' => $s3Key,
        ]);
    }
```

- 如需 API 詳細資訊，請參閱《適用於 PHP 的 AWS SDK API 參考》中的 UpdateFunctionCode。

PowerShell

Tools for PowerShell V4

範例 1：使用指定 zip 檔案中包含的新內容，更新名為 'MyFunction' 的函數。對於 C# .NET Core Lambda 函數，zip 檔案應包含已編譯的組件。

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

範例 2：此範例類似於上一個範例，但使用包含更新程式碼的 Amazon S3 物件來更新函數。

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName amzn-s3-demo-bucket -
Key UpdatedCode.zip
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 UpdateFunctionCode。

Tools for PowerShell V5

範例 1：使用指定 zip 檔案中包含的新內容，更新名為 'MyFunction' 的函數。對於 C# .NET Core Lambda 函數，zip 檔案應包含已編譯的組件。

```
Update-LMFunctionCode -FunctionName MyFunction -ZipFilename .\UpdatedCode.zip
```

範例 2：此範例類似於上一個範例，但使用包含更新程式碼的 Amazon S3 物件來更新函數。

```
Update-LMFunctionCode -FunctionName MyFunction -BucketName amzn-s3-demo-bucket -
Key UpdatedCode.zip
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V5)》中的 UpdateFunctionCode。

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_code(self, function_name, deployment_package):
        """
        Updates the code for a Lambda function by submitting a .zip archive that
 contains
        the code for the function.

        :param function_name: The name of the function to update.
        :param deployment_package: The function code to update, packaged as bytes
 in
                                   .zip format.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_code(
                FunctionName=function_name, ZipFile=deployment_package
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
```

```
            )
            raise
        else:
            return response
```

- 如需 API 詳細資訊，請參閱《適用於 Python (Boto3) 的AWS 開發套件 API 參考》中的「UpdateFunctionCode」。

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Updates the code for a Lambda function by submitting a .zip archive that
 contains
  # the code for the function.
  #
  # @param function_name: The name of the function to update.
  # @param deployment_package: The function code to update, packaged as bytes in
  #                            .zip format.
  # @return: Data about the update, including the status.
  def update_function_code(function_name, deployment_package)
    @lambda_client.update_function_code(
```

```
      function_name: function_name,
      zip_file: deployment_package
    )
    @lambda_client.wait_until(:function_updated_v2, { function_name:
function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating function code for:
#{function_name}:\n #{e.message}")
    nil
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to update:\n
#{e.message}")
  end
```

- 如需 API 詳細資訊，請參閱《適用於 Ruby 的 AWS SDK API 參考》中的 UpdateFunctionCode。

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /** Given a Path to a zip file, update the function's code and wait for the
update to finish. */
    pub async fn update_function_code(
        &self,
        zip_file: PathBuf,
        key: String,
    ) -> Result<UpdateFunctionCodeOutput, anyhow::Error> {
        let function_code = self.prepare_function(zip_file, Some(key)).await?;

        info!("Updating code for {}", self.lambda_name);
```

```
        let update = self
            .lambda_client
            .update_function_code()
            .function_name(self.lambda_name.clone())
            .s3_bucket(self.bucket.clone())
            .s3_key(function_code.s3_key().unwrap().to_string())
            .send()
            .await
            .map_err(anyhow::Error::from)?;

        self.wait_for_function_ready().await?;

        Ok(update)
    }

    /**
     * Upload function code from a path to a zip file.
     * The zip file must have an AL2 Linux-compatible binary called `bootstrap`.
     * The easiest way to create such a zip is to use `cargo lambda build --
output-format Zip`.
     */
    async fn prepare_function(
        &self,
        zip_file: PathBuf,
        key: Option<String>,
    ) -> Result<FunctionCode, anyhow::Error> {
        let body = ByteStream::from_path(zip_file).await?;

        let key = key.unwrap_or_else(|| format!("{}_code", self.lambda_name));

        info!("Uploading function code to s3://{}/{}", self.bucket, key);
        let _ = self
            .s3_client
            .put_object()
            .bucket(self.bucket.clone())
            .key(key.clone())
            .body(body)
            .send()
            .await?;

        Ok(FunctionCode::builder()
            .s3_bucket(self.bucket.clone())
            .s3_key(key)
            .build())
```

```
    }
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Rust API 參考》中的 UpdateFunctionCode。

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    TRY.
        oo_result = lo_lmd->updatefunctioncode(     " oo_result is returned for
testing purposes. "
              iv_functionname = iv_function_name
              iv_zipfile = io_zip_file ).

        MESSAGE 'Lambda function code updated.' TYPE 'I'.
      CATCH /aws1/cx_lmdcodesigningcfgno00.
        MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodestorageexcdex.
        MESSAGE 'Maximum total code size per account exceeded.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodeverification00.
        MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidcodesigex.
        MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
```

```
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請參閱《適用於 SAP ABAP 的AWS SDK API 參考》中的 UpdateFunctionCode。

Swift

適用於 Swift 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import AWSClientRuntime
import AWSLambda
import Foundation

        let zipUrl = URL(fileURLWithPath: path)
        let zipData: Data

        // Read the function's Zip file.

        do {
            zipData = try Data(contentsOf: zipUrl)
        } catch {
            throw ExampleError.zipFileReadError
        }

        // Update the function's code and wait for the updated version to be
        // ready for use.

        do {
            _ = try await lambdaClient.updateFunctionCode(
                input: UpdateFunctionCodeInput(
                    functionName: functionName,
                    zipFile: zipData
                )
```

```
            )
        } catch {
            return false
        }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Swift API reference 中的 UpdateFunctionCode。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

`UpdateFunctionConfiguration` 搭配 AWS SDK 或 CLI 使用

下列程式碼範例示範如何使用 `UpdateFunctionConfiguration`。

動作範例是大型程式的程式碼摘錄，必須在內容中執行。您可以在下列程式碼範例的內容中看到此動作：

- 了解基本概念

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /// <summary>
    /// Update the code of a Lambda function.
    /// </summary>
    /// <param name="functionName">The name of the function to update.</param>
    /// <param name="functionHandler">The code that performs the function's
actions.</param>
    /// <param name="environmentVariables">A dictionary of environment
variables.</param>
    /// <returns>A Boolean value indicating the success of the action.</returns>
    public async Task<bool> UpdateFunctionConfigurationAsync(
        string functionName,
```

```
        string functionHandler,
        Dictionary<string, string> environmentVariables)
    {
        var request = new UpdateFunctionConfigurationRequest
        {
            Handler = functionHandler,
            FunctionName = functionName,
            Environment = new Amazon.Lambda.Model.Environment { Variables =
environmentVariables },
        };

        var response = await
_lambdaService.UpdateFunctionConfigurationAsync(request);

        Console.WriteLine(response.LastModified);

        return response.HttpStatusCode == System.Net.HttpStatusCode.OK;
    }
```

- 如需 API 詳細資訊，請參閱《適用於 .NET 的 AWS SDK API 參考》中的 UpdateFunctionConfiguration。

C++

適用於 C++ 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
        Aws::Client::ClientConfiguration clientConfig;
        // Optional: Set to the AWS Region in which the bucket was created
(overrides config file).
        // clientConfig.region = "us-east-1";

    Aws::Lambda::LambdaClient client(clientConfig);
```

```
        Aws::Lambda::Model::UpdateFunctionConfigurationRequest request;
        request.SetFunctionName(LAMBDA_NAME);
        Aws::Lambda::Model::Environment environment;
        environment.AddVariables("LOG_LEVEL", "DEBUG");
        request.SetEnvironment(environment);

        Aws::Lambda::Model::UpdateFunctionConfigurationOutcome outcome =
 client.UpdateFunctionConfiguration(
                request);

        if (outcome.IsSuccess()) {
            std::cout << "The lambda configuration was successfully updated."
                      << std::endl;
            break;
        }

        else {
            std::cerr << "Error with Lambda::UpdateFunctionConfiguration. "
                      << outcome.GetError().GetMessage()
                      << std::endl;
        }
```

- 如需 API 詳細資訊，請參閱《適用於 C++ 的 AWS SDK API 參考》中的 UpdateFunctionConfiguration。

CLI

AWS CLI

若要修改函數的組態

下列 `update-function-configuration` 範例會將 `my-function` 函數未發布 ($LATEST) 版本的記憶體大小修改為 256 MB。

```
aws lambda update-function-configuration \
    --function-name  my-function \
    --memory-size 256
```

輸出：

```
{
```

```
    "FunctionName": "my-function",
    "LastModified": "2019-09-26T20:28:40.438+0000",
    "RevisionId": "e52502d4-9320-4688-9cd6-152a6ab7490d",
    "MemorySize": 256,
    "Version": "$LATEST",
    "Role": "arn:aws:iam::123456789012:role/service-role/my-function-role-
uy3l9qyq",
    "Timeout": 3,
    "Runtime": "nodejs10.x",
    "TracingConfig": {
        "Mode": "PassThrough"
    },
    "CodeSha256": "5tT2qgzYUHaqwR716pZ2dpkn/0J1FrzJmlKidWoaCgk=",
    "Description": "",
    "VpcConfig": {
        "SubnetIds": [],
        "VpcId": "",
        "SecurityGroupIds": []
    },
    "CodeSize": 304,
    "FunctionArn": "arn:aws:lambda:us-west-2:123456789012:function:my-function",
    "Handler": "index.handler"
}
```

如需詳細資訊，請參閱《AWS Lambda 開發人員指南》中的 AWS Lambda 函數組態。

- 如需 API 詳細資訊，請參閱《AWS CLI 命令參考》中的 UpdateFunctionConfiguration。

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import (
 "bytes"
 "context"
```

```
 "encoding/json"
 "errors"
 "log"
 "time"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/lambda"
 "github.com/aws/aws-sdk-go-v2/service/lambda/types"
)

// FunctionWrapper encapsulates function actions used in the examples.
// It contains an AWS Lambda service client that is used to perform user actions.
type FunctionWrapper struct {
 LambdaClient *lambda.Client
}



// UpdateFunctionConfiguration updates a map of environment variables configured
 for
// the Lambda function specified by functionName.
func (wrapper FunctionWrapper) UpdateFunctionConfiguration(ctx context.Context,
 functionName string, envVars map[string]string) {
 _, err := wrapper.LambdaClient.UpdateFunctionConfiguration(ctx,
 &lambda.UpdateFunctionConfigurationInput{
  FunctionName: aws.String(functionName),
  Environment:  &types.Environment{Variables: envVars},
 })
 if err != nil {
  log.Panicf("Couldn't update configuration for %v. Here's why: %v",
 functionName, err)
 }
}
```

- 如需 API 詳細資訊，請參閱《適用於 Go 的 AWS SDK API 參考》中的 UpdateFunctionConfiguration。

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /**
     * Updates the configuration of an AWS Lambda function.
     *
     * @param awsLambda     the {@link LambdaClient} instance to use for the AWS
Lambda operation
     * @param functionName  the name of the AWS Lambda function to update
     * @param handler       the new handler for the AWS Lambda function
     *
     * @throws LambdaException if there is an error while updating the function
configuration
     */
    public static void updateFunctionConfiguration(LambdaClient awsLambda, String
functionName, String handler) {
        try {
            UpdateFunctionConfigurationRequest configurationRequest =
UpdateFunctionConfigurationRequest.builder()
                .functionName(functionName)
                .handler(handler)
                .runtime(Runtime.JAVA17)
                .build();

            awsLambda.updateFunctionConfiguration(configurationRequest);

        } catch (LambdaException e) {
            System.err.println(e.getMessage());
            System.exit(1);
        }
    }
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Java 2.x API 參考》中的 UpdateFunctionConfiguration。

JavaScript

適用於 JavaScript (v3) 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
const updateFunctionConfiguration = (funcName) => {
  const client = new LambdaClient({});
  const config = readFileSync(`${dirname}../functions/config.json`).toString();
  const command = new UpdateFunctionConfigurationCommand({
    ...JSON.parse(config),
    FunctionName: funcName,
  });
  const result = client.send(command);
  waitForFunctionUpdated({ FunctionName: funcName });
  return result;
};
```

- 如需 API 詳細資訊，請參閱《適用於 JavaScript 的 AWS SDK API 參考》中的 UpdateFunctionConfiguration。

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    public function updateFunctionConfiguration($functionName, $handler,
 $environment = '')
    {
```

```
        return $this->lambdaClient->updateFunctionConfiguration([
            'FunctionName' => $functionName,
            'Handler' => "$handler.lambda_handler",
            'Environment' => $environment,
        ]);
    }
```

- 如需 API 詳細資訊，請參閱《適用於 PHP 的 AWS SDK API 參考》中的 UpdateFunctionConfiguration。

PowerShell

Tools for PowerShell V4

範例 1：此範例會更新現有的 Lambda 函數組態

```
Update-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Handler
 "lambda_function.launch_instance" -Timeout 600 -Environment_Variable
 @{ "envvar1"="value";"envvar2"="value" } -Role arn:aws:iam::123456789101:role/
service-role/lambda -DeadLetterConfig_TargetArn arn:aws:sns:us-east-1:
 123456789101:MyfirstTopic
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V4)》中的 UpdateFunctionConfiguration。

Tools for PowerShell V5

範例 1：此範例會更新現有的 Lambda 函數組態

```
Update-LMFunctionConfiguration -FunctionName "MylambdaFunction123" -Handler
 "lambda_function.launch_instance" -Timeout 600 -Environment_Variable
 @{ "envvar1"="value";"envvar2"="value" } -Role arn:aws:iam::123456789101:role/
service-role/lambda -DeadLetterConfig_TargetArn arn:aws:sns:us-east-1:
 123456789101:MyfirstTopic
```

- 如需 API 詳細資訊，請參閱《AWS Tools for PowerShell Cmdlet 參考 (V5)》中的 UpdateFunctionConfiguration。

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper:
    def __init__(self, lambda_client, iam_resource):
        self.lambda_client = lambda_client
        self.iam_resource = iam_resource


    def update_function_configuration(self, function_name, env_vars):
        """
        Updates the environment variables for a Lambda function.

        :param function_name: The name of the function to update.
        :param env_vars: A dict of environment variables to update.
        :return: Data about the update, including the status.
        """
        try:
            response = self.lambda_client.update_function_configuration(
                FunctionName=function_name, Environment={"Variables": env_vars}
            )
        except ClientError as err:
            logger.error(
                "Couldn't update function configuration %s. Here's why: %s: %s",
                function_name,
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
        else:
            return response
```

- 如需 API 詳細資訊，請參閱《適用於 Python (Boto3) 的AWS 開發套件 API 參考》中的「UpdateFunctionConfiguration」。

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
class LambdaWrapper
  attr_accessor :lambda_client, :cloudwatch_client, :iam_client

  def initialize
    @lambda_client = Aws::Lambda::Client.new
    @cloudwatch_client = Aws::CloudWatchLogs::Client.new(region: 'us-east-1')
    @iam_client = Aws::IAM::Client.new(region: 'us-east-1')
    @logger = Logger.new($stdout)
    @logger.level = Logger::WARN
  end

  # Updates the environment variables for a Lambda function.
  # @param function_name: The name of the function to update.
  # @param log_level: The log level of the function.
  # @return: Data about the update, including the status.
  def update_function_configuration(function_name, log_level)
    @lambda_client.update_function_configuration({
                                                   function_name: function_name,
                                                   environment: {
                                                     variables: {
                                                       'LOG_LEVEL' => log_level
                                                     }
                                                   }
                                                 })
    @lambda_client.wait_until(:function_updated_v2, { function_name:
 function_name }) do |w|
      w.max_attempts = 5
      w.delay = 5
```

```
    end
  rescue Aws::Lambda::Errors::ServiceException => e
    @logger.error("There was an error updating configurations for
#{function_name}:\n #{e.message}")
  rescue Aws::Waiters::Errors::WaiterFailed => e
    @logger.error("Failed waiting for #{function_name} to activate:\n
#{e.message}")
  end
```

- 如需 API 詳細資訊，請參閱《適用於 Ruby 的 AWS SDK API 參考》中的 UpdateFunctionConfiguration。

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    /** Update the environment for a function. */
    pub async fn update_function_configuration(
        &self,
        environment: Environment,
    ) -> Result<UpdateFunctionConfigurationOutput, anyhow::Error> {
        info!(
            ?environment,
            "Updating environment for {}", self.lambda_name
        );
        let updated = self
            .lambda_client
            .update_function_configuration()
            .function_name(self.lambda_name.clone())
            .environment(environment)
            .send()
            .await
            .map_err(anyhow::Error::from)?;
```

```
        self.wait_for_function_ready().await?;

        Ok(updated)
    }
```

- 如需 API 詳細資訊，請參閱《AWS SDK for Rust API 參考》中的 UpdateFunctionConfiguration。

SAP ABAP

適用於 SAP ABAP 的開發套件

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
    TRY.
        oo_result = lo_lmd->updatefunctionconfiguration(     " oo_result is
returned for testing purposes. "
              iv_functionname = iv_function_name
              iv_runtime = iv_runtime
              iv_description  = 'Updated Lambda function'
              iv_memorysize  = iv_memory_size ).

        MESSAGE 'Lambda function configuration/settings updated.' TYPE 'I'.
      CATCH /aws1/cx_lmdcodesigningcfgno00.
        MESSAGE 'Code signing configuration does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdcodeverification00.
        MESSAGE 'Code signature failed one or more validation checks for
signature mismatch or expiration.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvalidcodesigex.
        MESSAGE 'Code signature failed the integrity check.' TYPE 'E'.
      CATCH /aws1/cx_lmdinvparamvalueex.
        MESSAGE 'The request contains a non-valid parameter.' TYPE 'E'.
      CATCH /aws1/cx_lmdresourceconflictex.
        MESSAGE 'Resource already exists or another operation is in progress.'
TYPE 'E'.
      CATCH /aws1/cx_lmdresourcenotfoundex.
```

```
        MESSAGE 'The requested resource does not exist.' TYPE 'E'.
      CATCH /aws1/cx_lmdserviceexception.
        MESSAGE 'An internal problem was encountered by the AWS Lambda service.'
TYPE 'E'.
      CATCH /aws1/cx_lmdtoomanyrequestsex.
        MESSAGE 'The maximum request throughput was reached.' TYPE 'E'.
    ENDTRY.
```

- 如需 API 詳細資訊，請參閱《適用於 SAP ABAP 的AWS SDK API 參考》中的 UpdateFunctionConfiguration。

Swift

適用於 Swift 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

```
import AWSClientRuntime
import AWSLambda
import Foundation

    /// Tell the server-side component to log debug output by setting its
    /// environment's `LOG_LEVEL` to `DEBUG`.
    ///
    /// - Parameters:
    ///   - lambdaClient: The `LambdaClient` to use.
    ///   - functionName: The name of the AWS Lambda function to enable debug
    ///     logging for.
    ///
    /// - Throws: `ExampleError.environmentResponseMissingError`,
    ///   `ExampleError.updateFunctionConfigurationError`,
    ///   `ExampleError.environmentVariablesMissingError`,
    ///   `ExampleError.logLevelIncorrectError`,
    ///   `ExampleError.updateFunctionConfigurationError`
    func enableDebugLogging(lambdaClient: LambdaClient, functionName: String)
async throws {
```

```
        let envVariables = [
            "LOG_LEVEL": "DEBUG"
        ]
        let environment = LambdaClientTypes.Environment(variables: envVariables)

        do {
            let output = try await lambdaClient.updateFunctionConfiguration(
                input: UpdateFunctionConfigurationInput(
                    environment: environment,
                    functionName: functionName
                )
            )

            guard let response = output.environment else {
                throw ExampleError.environmentResponseMissingError
            }

            if response.error != nil {
                throw ExampleError.updateFunctionConfigurationError
            }

            guard let retVariables = response.variables else {
                throw ExampleError.environmentVariablesMissingError
            }

            for envVar in retVariables {
                if envVar.key == "LOG_LEVEL" && envVar.value != "DEBUG" {
                    print("*** Log level is not set to DEBUG!")
                    throw ExampleError.logLevelIncorrectError
                }
            }
        } catch {
            throw ExampleError.updateFunctionConfigurationError
        }
    }
```

- 如需 API 詳細資訊，請參閱 AWS SDK for Swift API reference 中的 UpdateFunctionConfiguration。

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 AWS SDKs Lambda 案例

下列程式碼範例示範如何在 Lambda AWS SDKs中實作常見案例。這些案例會向您展示如何呼叫 Lambda 中的多個函數或與其他 AWS 服務組合來完成特定任務。每個案例均包含完整原始碼的連結，您可在連結中找到如何設定和執行程式碼的相關指示。

案例的目標是獲得中等水平的經驗，協助您了解內容中的服務動作。

範例

- 使用 SDK 透過 Lambda 函數自動確認已知的 Amazon Cognito 使用者 AWS
- 使用 SDK 透過 Lambda 函數自動遷移已知的 Amazon Cognito 使用者 AWS
- 建立 API Gateway REST API 以追蹤 COVID-19 資料
- 建立出借圖書館 REST API
- 使用 Step Functions 建立傳訊應用程式
- 建立相片資產管理應用程式，讓使用者以標籤管理相片
- 使用 API Gateway 建立 websocket 聊天應用程式
- 建立可分析客戶意見回饋並合成音訊的應用程式
- 從瀏覽器調用 Lambda 函式
- 使用 S3 Object Lambda 轉換應用程式的資料
- 使用 API Gateway 來調用 Lambda 函式
- 使用 Step Functions 調用 Lambda 函式
- 使用排程事件來調用 Lambda 函式
- 使用 Amazon Neptune API 開發可查詢圖形資料的 Lambda 函式
- 使用 AWS SDK 進行 Amazon Cognito 使用者身分驗證後，使用 Lambda 函數寫入自訂活動資料

使用 SDK 透過 Lambda 函數自動確認已知的 Amazon Cognito 使用者 AWS

下列程式碼範例示範如何使用 Lambda 函數自動確認已知的 Amazon Cognito 使用者。

- 設定使用者集區以呼叫 `PreSignUp` 觸發條件的 Lambda 函數。
- 使用 Amazon Cognito 註冊使用者。
- Lambda 函數會掃描 DynamoDB 資料表，並自動確認已知使用者。
- 以新使用者身分登入，然後清除資源。

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

在命令提示中執行互動式案例。

```
import (
 "context"
 "errors"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// AutoConfirm separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type AutoConfirm struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewAutoConfirm constructs a new auto confirm runner.
func NewAutoConfirm(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) AutoConfirm {
 scenario := AutoConfirm{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
```

```
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddPreSignUpTrigger adds a Lambda handler as an invocation target for the
 PreSignUp trigger.
func (runner *AutoConfirm) AddPreSignUpTrigger(ctx context.Context, userPoolId
 string, functionArn string) {
 log.Printf("Let's add a Lambda function to handle the PreSignUp trigger from
 Cognito.\n" +
  "This trigger happens when a user signs up, and lets your function take action
 before the main Cognito\n" +
  "sign up processing occurs.\n")
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.PreSignUp, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
 log.Printf("Lambda function %v added to user pool %v to handle the PreSignUp
 trigger.\n",
  functionArn, userPoolId)
}

// SignUpUser signs up a user from the known user table with a password you
 specify.
func (runner *AutoConfirm) SignUpUser(ctx context.Context, clientId string,
 usersTable string) (string, string) {
 log.Println("Let's sign up a user to your Cognito user pool. When the user's
 email matches an email in the\n" +
  "DynamoDB known users table, it is automatically verified and the user is
 confirmed.")

 knownUsers, err := runner.helper.GetKnownUsers(ctx, usersTable)
 if err != nil {
  panic(err)
 }
 userChoice := runner.questioner.AskChoice("Which user do you want to use?\n",
 knownUsers.UserNameList())
 user := knownUsers.Users[userChoice]
```

```
 var signedUp bool
 var userConfirmed bool
 password := runner.questioner.AskPassword("Enter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !signedUp {
  log.Printf("Signing up user '%v' with email '%v' to Cognito.\n", user.UserName,
 user.UserEmail)
  userConfirmed, err = runner.cognitoActor.SignUp(ctx, clientId, user.UserName,
 password, user.UserEmail)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("Enter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   signedUp = true
  }
 }
 log.Printf("User %v signed up, confirmed = %v.\n", user.UserName, userConfirmed)

 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// SignInUser signs in a user.
func (runner *AutoConfirm) SignInUser(ctx context.Context, clientId string,
 userName string, password string) string {
 runner.questioner.Ask("Press Enter when you're ready to continue.")
 log.Printf("Let's sign in as %v...\n", userName)
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Printf("Successfully signed in. Your access token starts with: %v...\n",
 (*authResult.AccessToken)[:10])
 log.Println(strings.Repeat("-", 88))
 return *authResult.AccessToken
}
```

```
// Run runs the scenario.
func (runner *AutoConfirm) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(ctx, stackOutputs["TableName"])

 runner.AddPreSignUpTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["AutoConfirmFunctionArn"])
 runner.resources.triggers = append(runner.resources.triggers, actions.PreSignUp)
 userName, password := runner.SignUpUser(ctx, stackOutputs["UserPoolClientId"],
 stackOutputs["TableName"])
 runner.helper.ListRecentLogEvents(ctx, stackOutputs["AutoConfirmFunction"])
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
  runner.SignInUser(ctx, stackOutputs["UserPoolClientId"], userName, password))

 runner.resources.Cleanup(ctx)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

使用 Lambda 函數來處理 PreSignUp 觸發條件。

```
import (
```

```
 "context"
 "log"
 "os"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 dynamodbtypes "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

const TABLE_NAME = "TABLE_NAME"

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}

// GetKey marshals the user email value to a DynamoDB key format.
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PreSignUp event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be confirmed and verified.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsPreSignup) (events.CognitoEventUserPoolsPreSignup,
 error) {
 log.Printf("Received presignup from %v for user '%v'", event.TriggerSource,
 event.UserName)
 if event.TriggerSource != "PreSignUp_SignUp" {
```

```
  // Other trigger sources, such as PreSignUp_AdminInitiateAuth, ignore the
 response from this handler.
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserEmail: event.Request.UserAttributes["email"],
 }
 log.Printf("Looking up email %v in table %v.\n", user.UserEmail, tableName)
 output, err := h.dynamoClient.GetItem(ctx, &dynamodb.GetItemInput{
  Key:       user.GetKey(),
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Error looking up email %v.\n", user.UserEmail)
  return event, err
 }
 if output.Item == nil {
  log.Printf("Email %v not found. Email verification is required.\n",
 user.UserEmail)
  return event, err
 }

 err = attributevalue.UnmarshalMap(output.Item, &user)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB item. Here's why: %v\n", err)
  return event, err
 }

 if user.UserName != event.UserName {
  log.Printf("UserEmail %v found, but stored UserName '%v' does not match
 supplied UserName '%v'. Verification is required.\n",
   user.UserEmail, user.UserName, event.UserName)
 } else {
  log.Printf("UserEmail %v found with matching UserName %v. User is confirmed.
\n", user.UserEmail, user.UserName)
  event.Response.AutoConfirmUser = true
  event.Response.AutoVerifyEmail = true
 }

 return event, err
}

func main() {
```

```
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
}
```

建立執行一般任務的 struct。

```
import (
 "context"
 "log"
 "strings"
 "time"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
}
```

```
// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
```

```
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
```

```
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

建立包裝 Amazon Cognito 動作的 struct。

```
import (
 "context"
 "errors"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
)

type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
```

```
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
```

```
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
```

```
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
```

```
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
```

```
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

建立包裝 DynamoDB 動作的 struct。

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
```

```
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
```

```
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
```

```
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

建立包裝 CloudWatch Logs 動作的 struct。

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs/types"
)

type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
```

```
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

建立包裝 CloudFormation 動作的結構。

```
import (
 "context"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
```

```
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
)

// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

清除資源。

```
import (
 "context"
 "log"
 "user_pools_and_lambda_triggers/actions"

 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// Resources keeps track of AWS resources created during an example and handles
```

```
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(ctx, accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
```

```
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- 如需 API 詳細資訊，請參閱《適用於 Go 的 AWS SDK API 參考》中的下列主題。
  - DeleteUser
  - InitiateAuth
  - SignUp
  - UpdateUserPool

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

設定互動式 "Scenario" 執行。JavaScript (v3) 範例會共用案例執行器，以簡化複雜的範例。完整的原始程式碼位於 GitHub。

```
import { AutoConfirm } from "./scenario-auto-confirm.js";

/**
```

```
 * The context is passed to every scenario. Scenario steps
 * will modify the context.
 */
const context = {
  errors: [],
  users: [
    {
      UserName: "test_user_1",
      UserEmail: "test_email_1@example.com",
    },
    {
      UserName: "test_user_2",
      UserEmail: "test_email_2@example.com",
    },
    {
      UserName: "test_user_3",
      UserEmail: "test_email_3@example.com",
    },
  ],
};

/**
 * Three Scenarios are created for the workflow. A Scenario is an orchestration
 class
 * that simplifies running a series of steps.
 */
export const scenarios = {
  // Demonstrate automatically confirming known users in a database.
  "auto-confirm": AutoConfirm(context),
};

// Call function if run directly
import { fileURLToPath } from "node:url";
import { parseScenarioArgs } from "@aws-doc-sdk-examples/lib/scenario/index.js";

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  parseScenarioArgs(scenarios, {
    name: "Cognito user pools and triggers",
    description:
      "Demonstrate how to use the AWS SDKs to customize Amazon Cognito
 authentication behavior.",
  });
}
```

此案例展示如何自動確認已知使用者。它會協調範例步驟。

```
import { wait } from "@aws-doc-sdk-examples/lib/utils/util-timers.js";
import {
  Scenario,
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/scenario.js";

import {
  getStackOutputs,
  logCleanUpReminder,
  promptForStackName,
  promptForStackRegion,
  skipWhenErrors,
} from "./steps-common.js";
import { populateTable } from "./actions/dynamodb-actions.js";
import {
  addPreSignUpHandler,
  deleteUser,
  getUser,
  signIn,
  signUpUser,
} from "./actions/cognito-actions.js";
import {
  getLatestLogStreamForLambda,
  getLogEvents,
} from "./actions/cloudwatch-logs-actions.js";

/**
 * @typedef {{
 *   errors: Error[],
 *   password: string,
 *   users: { UserName: string, UserEmail: string }[],
 *   selectedUser?: string,
 *   stackName?: string,
 *   stackRegion?: string,
 *   token?: string,
 *   confirmDeleteSignedInUser?: boolean,
 *   TableName?: string,
```

```
 *   UserPoolClientId?: string,
 *   UserPoolId?: string,
 *   UserPoolArn?: string,
 *   AutoConfirmHandlerArn?: string,
 *   AutoConfirmHandlerName?: string
 * }} State
 */

const greeting = new ScenarioOutput(
  "greeting",
  (/** @type {State} */ state) => `This demo will populate some users into the \
database created as part of the "${state.stackName}" stack. \
Then the AutoConfirmHandler will be linked to the PreSignUp \
trigger from Cognito. Finally, you will choose a user to sign up.`,
  { skipWhen: skipWhenErrors },
);

const logPopulatingUsers = new ScenarioOutput(
  "logPopulatingUsers",
  "Populating the DynamoDB table with some users.",
  { skipWhenErrors: skipWhenErrors },
);

const logPopulatingUsersComplete = new ScenarioOutput(
  "logPopulatingUsersComplete",
  "Done populating users.",
  { skipWhen: skipWhenErrors },
);

const populateUsers = new ScenarioAction(
  "populateUsers",
  async (/** @type {State} */ state) => {
    const [_, err] = await populateTable({
      region: state.stackRegion,
      tableName: state.TableName,
      items: state.users,
    });
    if (err) {
      state.errors.push(err);
    }
  },
  {
    skipWhen: skipWhenErrors,
  },
```

```
);

const logSetupSignUpTrigger = new ScenarioOutput(
  "logSetupSignUpTrigger",
  "Setting up the PreSignUp trigger for the Cognito User Pool.",
  { skipWhen: skipWhenErrors },
);

const setupSignUpTrigger = new ScenarioAction(
  "setupSignUpTrigger",
  async (/** @type {State} */ state) => {
    const [_, err] = await addPreSignUpHandler({
      region: state.stackRegion,
      userPoolId: state.UserPoolId,
      handlerArn: state.AutoConfirmHandlerArn,
    });
    if (err) {
      state.errors.push(err);
    }
  },
  {
    skipWhen: skipWhenErrors,
  },
);

const logSetupSignUpTriggerComplete = new ScenarioOutput(
  "logSetupSignUpTriggerComplete",
  (
    /** @type {State} */ state,
  ) => `The lambda function "${state.AutoConfirmHandlerName}" \
has been configured as the PreSignUp trigger handler for the user pool
 "${state.UserPoolId}".`,
  { skipWhen: skipWhenErrors },
);

const selectUser = new ScenarioInput(
  "selectedUser",
  "Select a user to sign up.",
  {
    type: "select",
    choices: (/** @type {State} */ state) => state.users.map((u) => u.UserName),
    skipWhen: skipWhenErrors,
    default: (/** @type {State} */ state) => state.users[0].UserName,
  },
```

```
);

const checkIfUserAlreadyExists = new ScenarioAction(
  "checkIfUserAlreadyExists",
  async (/** @type {State} */ state) => {
    const [user, err] = await getUser({
      region: state.stackRegion,
      userPoolId: state.UserPoolId,
      username: state.selectedUser,
    });

    if (err?.name === "UserNotFoundException") {
      // Do nothing. We're not expecting the user to exist before
      // sign up is complete.
      return;
    }

    if (err) {
      state.errors.push(err);
      return;
    }

    if (user) {
      state.errors.push(
        new Error(
          `The user "${state.selectedUser}" already exists in the user pool
 "${state.UserPoolId}".`,
        ),
      );
    }
  },
  {
    skipWhen: skipWhenErrors,
  },
);

const createPassword = new ScenarioInput(
  "password",
  "Enter a password that has at least eight characters, uppercase, lowercase,
 numbers and symbols.",
  { type: "password", skipWhen: skipWhenErrors, default: "Abcd1234!" },
);

const logSignUpExistingUser = new ScenarioOutput(
```

```
  "logSignUpExistingUser",
  (/** @type {State} */ state) => `Signing up user "${state.selectedUser}".`,
  { skipWhen: skipWhenErrors },
);

const signUpExistingUser = new ScenarioAction(
  "signUpExistingUser",
  async (/** @type {State} */ state) => {
    const signUp = (password) =>
      signUpUser({
        region: state.stackRegion,
        userPoolClientId: state.UserPoolClientId,
        username: state.selectedUser,
        email: state.users.find((u) => u.UserName === state.selectedUser)
          .UserEmail,
        password,
      });

    let [_, err] = await signUp(state.password);

    while (err?.name === "InvalidPasswordException") {
      console.warn("The password you entered was invalid.");
      await createPassword.handle(state);
      [_, err] = await signUp(state.password);
    }

    if (err) {
      state.errors.push(err);
    }
  },
  { skipWhen: skipWhenErrors },
);

const logSignUpExistingUserComplete = new ScenarioOutput(
  "logSignUpExistingUserComplete",
  (/** @type {State} */ state) =>
    `"${state.selectedUser} was signed up successfully.`,
  { skipWhen: skipWhenErrors },
);

const logLambdaLogs = new ScenarioAction(
  "logLambdaLogs",
  async (/** @type {State} */ state) => {
    console.log(
```

```
      "Waiting a few seconds to let Lambda write to CloudWatch Logs...\n",
    );
    await wait(10);

    const [logStream, logStreamErr] = await getLatestLogStreamForLambda({
      functionName: state.AutoConfirmHandlerName,
      region: state.stackRegion,
    });
    if (logStreamErr) {
      state.errors.push(logStreamErr);
      return;
    }

    console.log(
      `Getting some recent events from log stream "${logStream.logStreamName}"`,
    );
    const [logEvents, logEventsErr] = await getLogEvents({
      functionName: state.AutoConfirmHandlerName,
      region: state.stackRegion,
      eventCount: 10,
      logStreamName: logStream.logStreamName,
    });
    if (logEventsErr) {
      state.errors.push(logEventsErr);
      return;
    }

    console.log(logEvents.map((ev) => `\t${ev.message}`).join(""));
  },
  { skipWhen: skipWhenErrors },
);

const logSignInUser = new ScenarioOutput(
  "logSignInUser",
  (/** @type {State} */ state) => `Let's sign in as ${state.selectedUser}`,
  { skipWhen: skipWhenErrors },
);

const signInUser = new ScenarioAction(
  "signInUser",
  async (/** @type {State} */ state) => {
    const [response, err] = await signIn({
      region: state.stackRegion,
      clientId: state.UserPoolClientId,
```

```
      username: state.selectedUser,
      password: state.password,
    });

    if (err?.name === "PasswordResetRequiredException") {
      state.errors.push(new Error("Please reset your password."));
      return;
    }

    if (err) {
      state.errors.push(err);
      return;
    }

    state.token = response?.AuthenticationResult?.AccessToken;
  },
  { skipWhen: skipWhenErrors },
);

const logSignInUserComplete = new ScenarioOutput(
  "logSignInUserComplete",
  (/** @type {State} */ state) =>
    `Successfully signed in. Your access token starts with:
 ${state.token.slice(0, 11)}`,
  { skipWhen: skipWhenErrors },
);

const confirmDeleteSignedInUser = new ScenarioInput(
  "confirmDeleteSignedInUser",
  "Do you want to delete the currently signed in user?",
  { type: "confirm", skipWhen: skipWhenErrors },
);

const deleteSignedInUser = new ScenarioAction(
  "deleteSignedInUser",
  async (/** @type {State} */ state) => {
    const [_, err] = await deleteUser({
      region: state.stackRegion,
      accessToken: state.token,
    });

    if (err) {
      state.errors.push(err);
    }
```

```
  },
  {
    skipWhen: (/** @type {State} */ state) =>
      skipWhenErrors(state) || !state.confirmDeleteSignedInUser,
  },
);

const logErrors = new ScenarioOutput(
  "logErrors",
  (/** @type {State}*/ state) => {
    const errorList = state.errors
      .map((err) => ` - ${err.name}: ${err.message}`)
      .join("\n");
    return `Scenario errors found:\n${errorList}`;
  },
  {
    // Don't log errors when there aren't any!
    skipWhen: (/** @type {State} */ state) => state.errors.length === 0,
  },
);

export const AutoConfirm = (context) =>
  new Scenario(
    "AutoConfirm",
    [
      promptForStackName,
      promptForStackRegion,
      getStackOutputs,
      greeting,
      logPopulatingUsers,
      populateUsers,
      logPopulatingUsersComplete,
      logSetupSignUpTrigger,
      setupSignUpTrigger,
      logSetupSignUpTriggerComplete,
      selectUser,
      checkIfUserAlreadyExists,
      createPassword,
      logSignUpExistingUser,
      signUpExistingUser,
      logSignUpExistingUserComplete,
      logLambdaLogs,
      logSignInUser,
      signInUser,
```

```
      logSignInUserComplete,
      confirmDeleteSignedInUser,
      deleteSignedInUser,
      logCleanUpReminder,
      logErrors,
    ],
    context,
  );
```

這些是與其他案例共用的步驟。

```
import {
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/scenario.js";
import { getCfnOutputs } from "@aws-doc-sdk-examples/lib/sdk/cfn-outputs.js";

export const skipWhenErrors = (state) => state.errors.length > 0;

export const getStackOutputs = new ScenarioAction(
  "getStackOutputs",
  async (state) => {
    if (!state.stackName || !state.stackRegion) {
      state.errors.push(
        new Error(
          "No stack name or region provided. The stack name and \
region are required to fetch CFN outputs relevant to this example.",
        ),
      );
      return;
    }

    const outputs = await getCfnOutputs(state.stackName, state.stackRegion);
    Object.assign(state, outputs);
  },
);

export const promptForStackName = new ScenarioInput(
  "stackName",
  "Enter the name of the stack you deployed earlier.",
  { type: "input", default: "PoolsAndTriggersStack" },
```

```
);

export const promptForStackRegion = new ScenarioInput(
  "stackRegion",
  "Enter the region of the stack you deployed earlier.",
  { type: "input", default: "us-east-1" },
);

export const logCleanUpReminder = new ScenarioOutput(
  "logCleanUpReminder",
  "All done. Remember to run 'cdk destroy' to teardown the stack.",
  { skipWhen: skipWhenErrors },
);
```

具有 Lambda 函數之 PreSignUp 觸發條件的處理常式。

```
import type { PreSignUpTriggerEvent, Handler } from "aws-lambda";
import type { UserRepository } from "./user-repository";
import { DynamoDBUserRepository } from "./user-repository";

export class PreSignUpHandler {
  private userRepository: UserRepository;

  constructor(userRepository: UserRepository) {
    this.userRepository = userRepository;
  }

  private isPreSignUpTriggerSource(event: PreSignUpTriggerEvent): boolean {
    return event.triggerSource === "PreSignUp_SignUp";
  }

  private getEventUserEmail(event: PreSignUpTriggerEvent): string {
    return event.request.userAttributes.email;
  }

  async handlePreSignUpTriggerEvent(
    event: PreSignUpTriggerEvent,
  ): Promise<PreSignUpTriggerEvent> {
    console.log(
      `Received presignup from ${event.triggerSource} for user
 '${event.userName}'`,
    );
```

```
    if (!this.isPreSignUpTriggerSource(event)) {
      return event;
    }

    const eventEmail = this.getEventUserEmail(event);
    console.log(`Looking up email ${eventEmail}.`);
    const storedUserInfo =
      await this.userRepository.getUserInfoByEmail(eventEmail);

    if (!storedUserInfo) {
      console.log(
        `Email ${eventEmail} not found. Email verification is required.`,
      );
      return event;
    }

    if (storedUserInfo.UserName !== event.userName) {
      console.log(
        `UserEmail ${eventEmail} found, but stored UserName
 '${storedUserInfo.UserName}' does not match supplied UserName
 '${event.userName}'. Verification is required.`,
      );
    } else {
      console.log(
        `UserEmail ${eventEmail} found with matching UserName
 ${storedUserInfo.UserName}. User is confirmed.`,
      );
      event.response.autoConfirmUser = true;
      event.response.autoVerifyEmail = true;
    }
    return event;
  }
}

const createPreSignUpHandler = (): PreSignUpHandler => {
  const tableName = process.env.TABLE_NAME;
  if (!tableName) {
    throw new Error("TABLE_NAME environment variable is not set");
  }

  const userRepository = new DynamoDBUserRepository(tableName);
  return new PreSignUpHandler(userRepository);
};
```

```
export const handler: Handler = async (event: PreSignUpTriggerEvent) => {
  const preSignUpHandler = createPreSignUpHandler();
  return preSignUpHandler.handlePreSignUpTriggerEvent(event);
};
```

CloudWatch Logs 動作的模組。

```
import {
  CloudWatchLogsClient,
  GetLogEventsCommand,
  OrderBy,
  paginateDescribeLogStreams,
} from "@aws-sdk/client-cloudwatch-logs";

/**
 * Get the latest log stream for a Lambda function.
 * @param {{ functionName: string, region: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cloudwatch-logs").LogStream | null,
 unknown]>}
 */
export const getLatestLogStreamForLambda = async ({ functionName, region }) => {
  try {
    const logGroupName = `/aws/lambda/${functionName}`;
    const cwlClient = new CloudWatchLogsClient({ region });
    const paginator = paginateDescribeLogStreams(
      { client: cwlClient },
      {
        descending: true,
        limit: 1,
        orderBy: OrderBy.LastEventTime,
        logGroupName,
      },
    );

    for await (const page of paginator) {
      return [page.logStreams[0], null];
    }
  } catch (err) {
    return [null, err];
  }
```

```
};

/**
 * Get the log events for a Lambda function's log stream.
 * @param {{
 *   functionName: string,
 *   logStreamName: string,
 *   eventCount: number,
 *   region: string
 * }} config
 * @returns {Promise<[import("@aws-sdk/client-cloudwatch-logs").OutputLogEvent[]
 | null, unknown]>}
 */
export const getLogEvents = async ({
  functionName,
  logStreamName,
  eventCount,
  region,
}) => {
  try {
    const cwlClient = new CloudWatchLogsClient({ region });
    const logGroupName = `/aws/lambda/${functionName}`;
    const response = await cwlClient.send(
      new GetLogEventsCommand({
        logStreamName: logStreamName,
        limit: eventCount,
        logGroupName: logGroupName,
      }),
    );

    return [response.events, null];
  } catch (err) {
    return [null, err];
  }
};
```

Amazon Cognito 動作的模組。

```
import {
  AdminGetUserCommand,
  CognitoIdentityProviderClient,
```

```
  DeleteUserCommand,
  InitiateAuthCommand,
  SignUpCommand,
  UpdateUserPoolCommand,
} from "@aws-sdk/client-cognito-identity-provider";

/**
 * Connect a Lambda function to the PreSignUp trigger for a Cognito user pool
 * @param {{ region: string, userPoolId: string, handlerArn: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").UpdateUserPoolCommandOutput | null, unknown]>}
 */
export const addPreSignUpHandler = async ({
  region,
  userPoolId,
  handlerArn,
}) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({
      region,
    });

    const command = new UpdateUserPoolCommand({
      UserPoolId: userPoolId,
      LambdaConfig: {
        PreSignUp: handlerArn,
      },
    });

    const response = await cognitoClient.send(command);
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Attempt to register a user to a user pool with a given username and password.
 * @param {{
 *   region: string,
 *   userPoolClientId: string,
 *   username: string,
 *   email: string,
 *   password: string
```

```
 * }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").SignUpCommandOutput | null, unknown]>}
 */
export const signUpUser = async ({
  region,
  userPoolClientId,
  username,
  email,
  password,
}) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({
      region,
    });

    const response = await cognitoClient.send(
      new SignUpCommand({
        ClientId: userPoolClientId,
        Username: username,
        Password: password,
        UserAttributes: [{ Name: "email", Value: email }],
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Sign in a user to Amazon Cognito using a username and password authentication
 flow.
 * @param {{ region: string, clientId: string, username: string, password:
 string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").InitiateAuthCommandOutput | null, unknown]>}
 */
export const signIn = async ({ region, clientId, username, password }) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({ region });
    const response = await cognitoClient.send(
      new InitiateAuthCommand({
        AuthFlow: "USER_PASSWORD_AUTH",
```

```
        ClientId: clientId,
        AuthParameters: { USERNAME: username, PASSWORD: password },
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Retrieve an existing user from a user pool.
 * @param {{ region: string, userPoolId: string, username: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").AdminGetUserCommandOutput | null, unknown]>}
 */
export const getUser = async ({ region, userPoolId, username }) => {
  try {
    const cognitoClient = new CognitoIdentityProviderClient({ region });
    const response = await cognitoClient.send(
      new AdminGetUserCommand({
        UserPoolId: userPoolId,
        Username: username,
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};

/**
 * Delete the signed-in user. Useful for allowing a user to delete their
 * own profile.
 * @param {{ region: string, accessToken: string }} config
 * @returns {Promise<[import("@aws-sdk/client-cognito-identity-
provider").DeleteUserCommandOutput | null, unknown]>}
 */
export const deleteUser = async ({ region, accessToken }) => {
  try {
    const client = new CognitoIdentityProviderClient({ region });
    const response = await client.send(
      new DeleteUserCommand({ AccessToken: accessToken }),
    );
```

```
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};
```

DynamoDB 動作的模組。

```
import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
import {
  BatchWriteCommand,
  DynamoDBDocumentClient,
} from "@aws-sdk/lib-dynamodb";

/**
 * Populate a DynamoDB table with provide items.
 * @param {{ region: string, tableName: string, items: Record<string,
 unknown>[] }} config
 * @returns {Promise<[import("@aws-sdk/lib-dynamodb").BatchWriteCommandOutput |
 null, unknown]>}
 */
export const populateTable = async ({ region, tableName, items }) => {
  try {
    const ddbClient = new DynamoDBClient({ region });
    const docClient = DynamoDBDocumentClient.from(ddbClient);
    const response = await docClient.send(
      new BatchWriteCommand({
        RequestItems: {
          [tableName]: items.map((item) => ({
            PutRequest: {
              Item: item,
            },
          })),
        },
      }),
    );
    return [response, null];
  } catch (err) {
    return [null, err];
  }
};
```

- 如需 API 詳細資訊，請參閱《適用於 JavaScript 的 AWS SDK API 參考》中的下列主題。
  - DeleteUser
  - InitiateAuth
  - SignUp
  - UpdateUserPool

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 SDK 透過 Lambda 函數自動遷移已知的 Amazon Cognito 使用者 AWS

下列程式碼範例示範如何使用 Lambda 函數自動遷移已知的 Amazon Cognito 使用者。

- 設定使用者集區以呼叫 `MigrateUser` 觸發條件的 Lambda 函數。
- 使用不在使用者集區中的使用者名稱和電子郵件登入 Amazon Cognito。
- Lambda 函數會掃描 DynamoDB 資料表，並自動將已知使用者遷移至使用者集區。
- 執行忘記密碼流程，重設已遷移使用者的密碼。
- 以新使用者身分登入，然後清除資源。

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

在命令提示中執行互動式案例。

```
import (
 "context"
 "errors"
```

```
 "fmt"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// MigrateUser separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type MigrateUser struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewMigrateUser constructs a new migrate user runner.
func NewMigrateUser(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) MigrateUser {
 scenario := MigrateUser{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddMigrateUserTrigger adds a Lambda handler as an invocation target for the
 MigrateUser trigger.
func (runner *MigrateUser) AddMigrateUserTrigger(ctx context.Context, userPoolId
 string, functionArn string) {
 log.Printf("Let's add a Lambda function to handle the MigrateUser trigger from
 Cognito.\n" +
  "This trigger happens when an unknown user signs in, and lets your function
 take action before Cognito\n" +
  "rejects the user.\n\n")
```

```
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.UserMigration, HandlerArn:
 aws.String(functionArn)})
 if err != nil {
  panic(err)
 }
 log.Printf("Lambda function %v added to user pool %v to handle the MigrateUser
 trigger.\n",
  functionArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}

// SignInUser adds a new user to the known users table and signs that user in to
 Amazon Cognito.
func (runner *MigrateUser) SignInUser(ctx context.Context, usersTable string,
 clientId string) (bool, actions.User) {
 log.Println("Let's sign in a user to your Cognito user pool. When the username
 and email matches an entry in the\n" +
  "DynamoDB known users table, the email is automatically verified and the user
 is migrated to the Cognito user pool.")

 user := actions.User{}
 user.UserName = runner.questioner.Ask("\nEnter a username:")
 user.UserEmail = runner.questioner.Ask("\nEnter an email that you own. This
 email will be used to confirm user migration\n" +
  "during this example:")

 runner.helper.AddKnownUser(ctx, usersTable, user)

 var err error
 var resetRequired *types.PasswordResetRequiredException
 var authResult *types.AuthenticationResultType
 signedIn := false
 for !signedIn && resetRequired == nil {
  log.Printf("Signing in to Cognito as user '%v'. The expected result is a
 PasswordResetRequiredException.\n\n", user.UserName)
  authResult, err = runner.cognitoActor.SignIn(ctx, clientId, user.UserName, "_")
  if err != nil {
   if errors.As(err, &resetRequired) {
    log.Printf("\nUser '%v' is not in the Cognito user pool but was found in the
 DynamoDB known users table.\n"+
```

```
     "User migration is started and a password reset is required.",
 user.UserName)
   } else {
    panic(err)
   }
  } else {
   log.Printf("User '%v' successfully signed in. This is unexpected and probably
 means you have not\n"+
    "cleaned up a previous run of this scenario, so the user exist in the Cognito
 user pool.\n"+
    "You can continue this example and select to clean up resources, or manually
 remove\n"+
    "the user from your user pool and try again.", user.UserName)
   runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)
   signedIn = true
  }
 }

 log.Println(strings.Repeat("-", 88))
 return resetRequired != nil, user
}

// ResetPassword starts a password recovery flow.
func (runner *MigrateUser) ResetPassword(ctx context.Context, clientId string,
 user actions.User) {
 wantCode := runner.questioner.AskBool(fmt.Sprintf("In order to migrate the user
 to Cognito, you must be able to receive a confirmation\n"+
  "code by email at %v. Do you want to send a code (y/n)?", user.UserEmail), "y")
 if !wantCode {
  log.Println("To complete this example and successfully migrate a user to
 Cognito, you must enter an email\n" +
   "you own that can receive a confirmation code.")
  return
 }
 codeDelivery, err := runner.cognitoActor.ForgotPassword(ctx, clientId,
 user.UserName)
 if err != nil {
  panic(err)
 }
 log.Printf("\nA confirmation code has been sent to %v.",
 *codeDelivery.Destination)
 code := runner.questioner.Ask("Check your email and enter it here:")
```

```
 confirmed := false
 password := runner.questioner.AskPassword("\nEnter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !confirmed {
  log.Printf("\nConfirming password reset for user '%v'.\n", user.UserName)
  err = runner.cognitoActor.ConfirmForgotPassword(ctx, clientId, code,
 user.UserName, password)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("\nEnter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   confirmed = true
  }
 }
 log.Printf("User '%v' successfully confirmed and migrated.\n", user.UserName)
 log.Println("Signing in with your username and password...")
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, user.UserName,
 password)
 if err != nil {
  panic(err)
 }
 log.Printf("Successfully signed in. Your access token starts with: %v...\n",
 (*authResult.AccessToken)[:10])
 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)

 log.Println(strings.Repeat("-", 88))
}

// Run runs the scenario.
func (runner *MigrateUser) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
```

```
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]

 runner.AddMigrateUserTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["MigrateUserFunctionArn"])
 runner.resources.triggers = append(runner.resources.triggers,
 actions.UserMigration)
 resetNeeded, user := runner.SignInUser(ctx, stackOutputs["TableName"],
 stackOutputs["UserPoolClientId"])
 if resetNeeded {
  runner.helper.ListRecentLogEvents(ctx, stackOutputs["MigrateUserFunction"])
  runner.ResetPassword(ctx, stackOutputs["UserPoolClientId"], user)
 }

 runner.resources.Cleanup(ctx)

 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

使用 Lambda 函數來處理 `MigrateUser` 觸發條件。

```
import (
 "context"
 "log"
 "os"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
```

```
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/expression"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
)

const TABLE_NAME = "TABLE_NAME"

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string `dynamodbav:"UserName"`
 UserEmail string `dynamodbav:"UserEmail"`
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the MigrateUser event by looking up a user in an Amazon
 DynamoDB table and
// specifying whether they should be migrated to the user pool.
func (h *handler) HandleRequest(ctx context.Context, event
 events.CognitoEventUserPoolsMigrateUser)
 (events.CognitoEventUserPoolsMigrateUser, error) {
 log.Printf("Received migrate trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 if event.TriggerSource != "UserMigration_Authentication" {
  return event, nil
 }
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName: event.UserName,
 }
 log.Printf("Looking up user '%v' in table %v.\n", user.UserName, tableName)
 filterEx := expression.Name("UserName").Equal(expression.Value(user.UserName))
 expr, err := expression.NewBuilder().WithFilter(filterEx).Build()
 if err != nil {
  log.Printf("Error building expression to query for user '%v'.\n",
 user.UserName)
  return event, err
 }
 output, err := h.dynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName:                 aws.String(tableName),
  FilterExpression:          expr.Filter(),
  ExpressionAttributeNames:  expr.Names(),
```

```
  ExpressionAttributeValues: expr.Values(),
 })
 if err != nil {
  log.Printf("Error looking up user '%v'.\n", user.UserName)
  return event, err
 }
 if len(output.Items) == 0 {
  log.Printf("User '%v' not found, not migrating user.\n", user.UserName)
  return event, err
 }

 var users []UserInfo
 err = attributevalue.UnmarshalListOfMaps(output.Items, &users)
 if err != nil {
  log.Printf("Couldn't unmarshal DynamoDB items. Here's why: %v\n", err)
  return event, err
 }

 user = users[0]
 log.Printf("UserName '%v' found with email %v. User is migrated and must reset
 password.\n", user.UserName, user.UserEmail)
 event.CognitoEventUserPoolsMigrateUserResponse.UserAttributes =
 map[string]string{
  "email":          user.UserEmail,
  "email_verified": "true", // email_verified is required for the forgot password
 flow.
 }
 event.CognitoEventUserPoolsMigrateUserResponse.FinalUserStatus =
 "RESET_REQUIRED"
 event.CognitoEventUserPoolsMigrateUserResponse.MessageAction = "SUPPRESS"

 return event, err
}

func main() {
 ctx := context.Background()
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Panicln(err)
 }
 h := handler{
  dynamoClient: dynamodb.NewFromConfig(sdkConfig),
 }
 lambda.Start(h.HandleRequest)
```

```
}
```

建立執行一般任務的 struct。

```
import (
 "context"
 "log"
 "strings"
 "time"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}
```

```
// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}

// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
```

```
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

建立包裝 Amazon Cognito 動作的 struct。

```
import (
 "context"
 "errors"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
)

type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}



// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
```

```
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
```

```
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}
```

```
// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
```

```
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
```

```
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

建立包裝 DynamoDB 動作的 struct。

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}
```

```
// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
```

```
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
```

```
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

建立包裝 CloudWatch Logs 動作的 struct。

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs/types"
)

type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}

// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
```

```
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

建立包裝 CloudFormation 動作的結構。

```
import (
 "context"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
)

// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}
```

```
// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

清除資源。

```
import (
 "context"
 "log"
 "user_pools_and_lambda_triggers/actions"

 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}
```

```
func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(ctx, accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
```

```
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- 如需 API 詳細資訊，請參閱《適用於 Go 的 AWS SDK API 參考》中的下列主題。
  - ConfirmForgotPassword
  - DeleteUser
  - ForgotPassword
  - InitiateAuth
  - SignUp
  - UpdateUserPool

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

建立 API Gateway REST API 以追蹤 COVID-19 資料

以下程式碼範例示範如何建立 REST API，此 API 使用虛構資料模擬追蹤美國 COVID-19 每日病例的系統。

Python

適用於 Python 的 SDK (Boto3)

示範如何使用 AWS Chalice 搭配 適用於 Python (Boto3) 的 AWS SDK 來建立使用 Amazon API Gateway AWS Lambda和 Amazon DynamoDB 的無伺服器 REST API。REST API 使用虛構資料模擬追蹤美國 COVID-19 每日病例的系統。了解如何：

- 使用 AWS Chalice 在 Lambda 函數中定義路由，該函數稱為 來處理透過 API Gateway 發出的 REST 請求。
- 使用 Lambda 函式在 DynamoDB 資料表中擷取和存放資料，以便為 REST 請求提供服務。
- 在 AWS CloudFormation 範本中定義資料表結構和安全角色資源。
- 使用 AWS Chalice 和 CloudFormation 封裝和部署所有必要的資源。
- 使用 CloudFormation 清理所有已建立的資源。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- API Gateway
- CloudFormation
- DynamoDB
- Lambda

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

建立出借圖書館 REST API

下列程式碼範例顯示如何使用 Amazon Aurora 資料庫支援的 REST API 來建立出借圖書館，讓贊助人可以借書與還書。

Python

適用於 Python 的 SDK (Boto3)

示範如何使用 適用於 Python (Boto3) 的 AWS SDK 搭配 Amazon Relational Database Service (Amazon RDS) API 和 AWS Chalice 來建立由 Amazon Aurora 資料庫支援的 REST API。Web 服務是完全無伺服器的，表示這是一種贊助人可以借書與還書的簡單出借圖書館。了解如何：

- 建立與管理無伺服器的 Aurora 資料庫叢集。
- 使用 AWS Secrets Manager 管理資料庫登入資料。
- 實作資料儲存層，該層使用 Amazon RDS 將資料移入和移出資料庫。
- 使用 AWS Chalice 將無伺服器 REST API 部署至 Amazon API Gateway 和 AWS Lambda。
- 使用 Request 套件來將請求傳送到 Web 服務。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- API Gateway
- Aurora
- Lambda

- Secrets Manager

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Step Functions 建立傳訊應用程式

下列程式碼範例示範如何建立 AWS Step Functions 訊息應用程式，從資料庫資料表擷取訊息記錄。

Python

適用於 Python 的 SDK (Boto3)

示範如何使用 適用於 Python (Boto3) 的 AWS SDK 搭配 AWS Step Functions 來建立訊息應用程式，從 Amazon DynamoDB 資料表擷取訊息記錄，並使用 Amazon Simple Queue Service (Amazon SQS) 傳送它們。狀態機器會與 AWS Lambda 函數整合，以掃描資料庫是否有未傳送的訊息。

- 建立從 Amazon DynamoDB 資料表擷取和更新訊息記錄的狀態機器。
- 更新狀態機器定義，以便也向 Amazon Simple Queue Service (Amazon SQS) 傳送訊息。
- 開始和停用狀態機器執行。
- 使用服務整合從狀態機器連接至 Lambda、DynamoDB 和 Amazon SQS。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- DynamoDB
- Lambda
- Amazon SQS
- 步驟函數

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

建立相片資產管理應用程式，讓使用者以標籤管理相片

下列程式碼範例示範如何建立無伺服器應用程式，讓使用者以標籤管理相片。

.NET

適用於 .NET 的 SDK

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

C++

適用於 C++ 的 SDK

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Java

適用於 Java 2.x 的 SDK

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

JavaScript

適用於 JavaScript (v3) 的 SDK

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

Kotlin

SDK for Kotlin

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

PHP

適用於 PHP 的 SDK

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3

- Amazon SNS

Rust

適用於 Rust 的 SDK

顯示如何開發照片資產管理應用程式，以便使用 Amazon Rekognition 偵測圖片中的標籤，並將其儲存以供日後擷取。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

如要深入探索此範例的來源，請參閱 AWS 社群上的文章。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon Rekognition
- Amazon S3
- Amazon SNS

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 API Gateway 建立 websocket 聊天應用程式

下列程式碼範例示範如何建立由建置於 Amazon API Gateway 上的 websocket API 提供服務的聊天應用程式。

Python

適用於 Python 的 SDK (Boto3)

示範如何使用 適用於 Python (Boto3) 的 AWS SDK 搭配 Amazon API Gateway V2 來建立與 AWS Lambda 和 Amazon DynamoDB 整合的 Websocket API。

- 建立由 API Gateway 提供服務的 websocket API。
- 定義 Lambda 處理常式，該常式將連接存放在 DynamoDB 中，並將訊息傳送給其他聊天參與者。

- 連接至 websocket 聊天應用程式，並使用 Websockets 套件傳送訊息。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

建立可分析客戶意見回饋並合成音訊的應用程式

下列程式碼範例示範如何建立應用程式，以分析客戶評論卡、從其原始語言進行翻譯、判斷對方情緒，以及透過翻譯後的文字產生音訊檔案。

.NET

適用於 .NET 的 SDK

此範例應用程式會分析和存儲客戶的意見回饋卡。具體來說，它滿足了紐約市一家虛構飯店的需求。飯店以實體評論卡的形式收到賓客以各種語言撰寫的意見回饋。這些意見回饋透過 Web 用戶端上傳至應用程式。評論卡的影像上傳後，系統會執行下列步驟：

- 文字內容是使用 Amazon Textract 從影像中擷取。
- Amazon Comprehend 會決定擷取文字及其用語的情感。
- 擷取的文字內容會使用 Amazon Translate 翻譯成英文。
- Amazon Polly 會使用擷取的文字內容合成音訊檔案。

完整的應用程式可透過 AWS CDK 部署。如需原始程式碼和部署的說明，請參閱 GitHub 中的專案。

此範例中使用的服務

- Amazon Comprehend
- Lambda
- Amazon Polly

- Amazon Textract
- Amazon Translate

Java

適用於 Java 2.x 的 SDK

此範例應用程式會分析和存儲客戶的意見回饋卡。具體來說，它滿足了紐約市一家虛構飯店的需求。飯店以實體評論卡的形式收到賓客以各種語言撰寫的意見回饋。這些意見回饋透過 Web 用戶端上傳至應用程式。評論卡的影像上傳後，系統會執行下列步驟：

- 文字內容是使用 Amazon Textract 從影像中擷取。
- Amazon Comprehend 會決定擷取文字及其用語的情感。
- 擷取的文字內容會使用 Amazon Translate 翻譯成英文。
- Amazon Polly 會使用擷取的文字內容合成音訊檔案。

完整的應用程式可透過 AWS CDK 部署。如需原始程式碼和部署的說明，請參閱 GitHub 中的專案。

此範例中使用的服務

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

JavaScript

適用於 JavaScript (v3) 的 SDK

此範例應用程式會分析和存儲客戶的意見回饋卡。具體來說，它滿足了紐約市一家虛構飯店的需求。飯店以實體評論卡的形式收到賓客以各種語言撰寫的意見回饋。這些意見回饋透過 Web 用戶端上傳至應用程式。評論卡的影像上傳後，系統會執行下列步驟：

- 文字內容是使用 Amazon Textract 從影像中擷取。
- Amazon Comprehend 會決定擷取文字及其用語的情感。
- 擷取的文字內容會使用 Amazon Translate 翻譯成英文。

- Amazon Polly 會使用擷取的文字內容合成音訊檔案。

完整的應用程式可透過 AWS CDK 部署。如需原始程式碼和部署的說明，請參閱 GitHub 中的專案。以下摘錄顯示如何在 Lambda 函數內 適用於 JavaScript 的 AWS SDK 使用 。

```
import {
  ComprehendClient,
  DetectDominantLanguageCommand,
  DetectSentimentCommand,
} from "@aws-sdk/client-comprehend";

/**
 * Determine the language and sentiment of the extracted text.
 *
 * @param {{ source_text: string}} extractTextOutput
 */
export const handler = async (extractTextOutput) => {
  const comprehendClient = new ComprehendClient({});

  const detectDominantLanguageCommand = new DetectDominantLanguageCommand({
    Text: extractTextOutput.source_text,
  });

  // The source language is required for sentiment analysis and
  // translation in the next step.
  const { Languages } = await comprehendClient.send(
    detectDominantLanguageCommand,
  );

  const languageCode = Languages[0].LanguageCode;

  const detectSentimentCommand = new DetectSentimentCommand({
    Text: extractTextOutput.source_text,
    LanguageCode: languageCode,
  });

  const { Sentiment } = await comprehendClient.send(detectSentimentCommand);

  return {
    sentiment: Sentiment,
    language_code: languageCode,
  };
};
```

```
import {
  DetectDocumentTextCommand,
  TextractClient,
} from "@aws-sdk/client-textract";

/**
 * Fetch the S3 object from the event and analyze it using Amazon Textract.
 *
 * @param {import("@types/aws-lambda").EventBridgeEvent<"Object Created">}
 eventBridgeS3Event
 */
export const handler = async (eventBridgeS3Event) => {
  const textractClient = new TextractClient();

  const detectDocumentTextCommand = new DetectDocumentTextCommand({
    Document: {
      S3Object: {
        Bucket: eventBridgeS3Event.bucket,
        Name: eventBridgeS3Event.object,
      },
    },
  });

  // Textract returns a list of blocks. A block can be a line, a page, word, etc.
  // Each block also contains geometry of the detected text.
  // For more information on the Block type, see https://docs.aws.amazon.com/
textract/latest/dg/API_Block.html.
  const { Blocks } = await textractClient.send(detectDocumentTextCommand);

  // For the purpose of this example, we are only interested in words.
  const extractedWords = Blocks.filter((b) => b.BlockType === "WORD").map(
    (b) => b.Text,
  );

  return extractedWords.join(" ");
};
```

```
import { PollyClient, SynthesizeSpeechCommand } from "@aws-sdk/client-polly";
import { S3Client } from "@aws-sdk/client-s3";
import { Upload } from "@aws-sdk/lib-storage";

/**
 * Synthesize an audio file from text.
```

```
 *
 * @param {{ bucket: string, translated_text: string, object: string}}
 sourceDestinationConfig
 */
export const handler = async (sourceDestinationConfig) => {
  const pollyClient = new PollyClient({});

  const synthesizeSpeechCommand = new SynthesizeSpeechCommand({
    Engine: "neural",
    Text: sourceDestinationConfig.translated_text,
    VoiceId: "Ruth",
    OutputFormat: "mp3",
  });

  const { AudioStream } = await pollyClient.send(synthesizeSpeechCommand);

  const audioKey = `${sourceDestinationConfig.object}.mp3`;

  // Store the audio file in S3.
  const s3Client = new S3Client();
  const upload = new Upload({
    client: s3Client,
    params: {
      Bucket: sourceDestinationConfig.bucket,
      Key: audioKey,
      Body: AudioStream,
      ContentType: "audio/mp3",
    },
  });

  await upload.done();
  return audioKey;
};
```

```
import {
  TranslateClient,
  TranslateTextCommand,
} from "@aws-sdk/client-translate";

/**
 * Translate the extracted text to English.
 *
```

```
 * @param {{ extracted_text: string, source_language_code: string}}
 textAndSourceLanguage
 */
export const handler = async (textAndSourceLanguage) => {
  const translateClient = new TranslateClient({});

  const translateCommand = new TranslateTextCommand({
    SourceLanguageCode: textAndSourceLanguage.source_language_code,
    TargetLanguageCode: "en",
    Text: textAndSourceLanguage.extracted_text,
  });

  const { TranslatedText } = await translateClient.send(translateCommand);

  return { translated_text: TranslatedText };
};
```

此範例中使用的服務

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

Ruby

SDK for Ruby

此範例應用程式會分析和存儲客戶的意見回饋卡。具體來說，它滿足了紐約市一家虛構飯店的需求。飯店以實體評論卡的形式收到賓客以各種語言撰寫的意見回饋。這些意見回饋透過 Web 用戶端上傳至應用程式。評論卡的影像上傳後，系統會執行下列步驟：

- 文字內容是使用 Amazon Textract 從影像中擷取。
- Amazon Comprehend 會決定擷取文字及其用語的情感。
- 擷取的文字內容會使用 Amazon Translate 翻譯成英文。
- Amazon Polly 會使用擷取的文字內容合成音訊檔案。

完整的應用程式可透過 AWS CDK 部署。如需原始程式碼和部署的說明，請參閱 GitHub 中的專案。

此範例中使用的服務

- Amazon Comprehend
- Lambda
- Amazon Polly
- Amazon Textract
- Amazon Translate

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

從瀏覽器調用 Lambda 函式

下列程式碼範例示範如何從瀏覽器叫用 AWS Lambda 函數。

JavaScript

適用於 JavaScript (v2) 的 SDK

您可以建立瀏覽器型應用程式，該應用程式使用 AWS Lambda 函數更新具有使用者選擇的 Amazon DynamoDB 資料表。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- DynamoDB
- Lambda

適用於 JavaScript (v3) 的開發套件

您可以建立瀏覽器型應用程式，該應用程式使用 AWS Lambda 函數更新具有使用者選擇的 Amazon DynamoDB 資料表。此應用程式使用 適用於 JavaScript 的 AWS SDK v3。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- DynamoDB
- Lambda

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 S3 Object Lambda 轉換應用程式的資料

下列程式碼範例示範如何使用 S3 Object Lambda 轉換應用程式的資料。

.NET

適用於 .NET 的 SDK

展示如何將自訂程式碼新增至標準 S3 GET 請求，以修改從 S3 擷取的請求物件，如此一來，物件方能符合請求用戶端或應用程式的需求。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- Lambda
- Amazon S3

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 API Gateway 來調用 Lambda 函式

下列程式碼範例示範如何建立 Amazon API Gateway 調用的 AWS Lambda 函數。

Java

適用於 Java 2.x 的 SDK

示範如何使用 Lambda Java 執行時間 API 建立 AWS Lambda 函數。此範例會叫用不同的 AWS 服務來執行特定的使用案例。此範例示範如何建立 Amazon API Gateway 調用的 Lambda 函式，該函數會掃描 Amazon DynamoDB 資料表中的工作週年紀念日，並使用 Amazon Simple Notification Service (Amazon SNS) 傳送文字訊息給您的員工，在他們的週年紀念日向他們道賀。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- API Gateway

- DynamoDB
- Lambda
- Amazon SNS

JavaScript

適用於 JavaScript (v3) 的 SDK

示範如何使用 Lambda JavaScript 執行時間 API 建立 AWS Lambda 函數。此範例會叫用不同的 AWS 服務來執行特定的使用案例。此範例示範如何建立 Amazon API Gateway 調用的 Lambda 函式，該函數會掃描 Amazon DynamoDB 資料表中的工作週年紀念日，並使用 Amazon Simple Notification Service (Amazon SNS) 傳送文字訊息給您的員工，在他們的週年紀念日向他們道賀。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例也可在 適用於 JavaScript 的 AWS SDK v3 開發人員指南中取得。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

Python

適用於 Python (Boto3) 的開發套件

此範例顯示如何建立和使用目標為 AWS Lambda 函數的 Amazon API Gateway REST API。Lambda 處理常式會展示如何根據 HTTP 方法來路由；如何從查詢字串、標頭和本文中取得資料；以及如何傳回 JSON 回應。

- 部署 Lambda 函式。
- 建立 API Gateway REST API。
- 建立目標為 Lambda 函式的 REST 資源。
- 授與許可讓 API Gateway 調用 Lambda 函式。

- 使用 Request 套件來將請求傳送到 REST API。
- 清理示範期間建立的所有資源。

這個範例在 GitHub 上的檢視效果最佳。如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda
- Amazon SNS

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Step Functions 調用 Lambda 函式

下列程式碼範例示範如何建立依序叫用 AWS Lambda 函數 AWS Step Functions 的狀態機器。

Java

適用於 Java 2.x 的 SDK

顯示如何使用 AWS Step Functions 和 建立無 AWS 伺服器工作流程 AWS SDK for Java 2.x。每個工作流程步驟都是使用 AWS Lambda 函數實作。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- DynamoDB
- Lambda
- Amazon SES
- 步驟函數

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用排程事件來調用 Lambda 函式

下列程式碼範例示範如何建立由 Amazon EventBridge 排程事件呼叫的 AWS Lambda 函數。

Java

適用於 Java 2.x 的 SDK

顯示如何建立叫用 AWS Lambda 函數的 Amazon EventBridge 排程事件。將 EventBridge 設定為在調用 Lambda 函式時使用 Cron 表達式來進行排程。在此範例中，您會使用 Lambda Java 執行時期 API 建立 Lambda 函式。此範例會叫用不同的 AWS 服務來執行特定的使用案例。此範例示範如何建立應用程式，將行動裝置文字訊息傳送給員工，在他們的週年紀念日向他們道賀。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

JavaScript

適用於 JavaScript (v3) 的 SDK

顯示如何建立叫用 AWS Lambda 函數的 Amazon EventBridge 排程事件。將 EventBridge 設定為在調用 Lambda 函式時使用 Cron 表達式來進行排程。在此範例中，您會使用 Lambda JavaScript 執行時期 API 建立 Lambda 函式。此範例會叫用不同的 AWS 服務來執行特定的使用案例。此範例示範如何建立應用程式，將行動裝置文字訊息傳送給員工，在他們的週年紀念日向他們道賀。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例也可在 適用於 JavaScript 的 AWS SDK v3 開發人員指南中取得。

此範例中使用的服務

- CloudWatch Logs

- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

Python

適用於 Python 的 SDK (Boto3)

此範例說明如何將 AWS Lambda 函數註冊為排程 Amazon EventBridge 事件的目標。Lambda 處理常式會將合適的訊息和完整的事件資料寫入 Amazon CloudWatch Logs 中以供日後擷取。

- 部署 Lambda 函式。
- 建立一個 EventBridge 排程事件，並將 Lambda 函式做為目標。
- 授予許可讓 EventBridge 調用 Lambda 函式。
- 列印 CloudWatch Logs 中的最新資料，以顯示排程調用的結果。
- 清理示範期間建立的所有資源。

這個範例在 GitHub 上的檢視效果最佳。如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- CloudWatch Logs
- DynamoDB
- EventBridge
- Lambda
- Amazon SNS

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Amazon Neptune API 開發可查詢圖形資料的 Lambda 函式

下列程式碼範例示範如何使用 Neptune API 查詢圖形資料。

Java

適用於 Java 2.x 的 SDK

示範如何使用 Amazon Neptune Java API 建立 Lambda 函式，查詢 VPC 中的圖形資料。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- Lambda
- Neptune

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 AWS SDK 進行 Amazon Cognito 使用者身分驗證後，使用 Lambda 函數寫入自訂活動資料

下列程式碼範例示範如何在 Amazon Cognito 使用者身分驗證之後，使用 Lambda 函數撰寫自訂活動資料。

- 使用管理員函數將使用者新增至使用者集區。
- 設定使用者集區以呼叫 `PostAuthentication` 觸發條件的 Lambda 函數。
- 將新使用者登入 Amazon Cognito。
- Lambda 函數會將自訂資訊寫入 CloudWatch Logs 和 DynamoDB 資料表。
- 從 DynamoDB 資料表取得並顯示自訂資料，然後清除資源。

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在 AWS 程式碼範例儲存庫中設定和執行。

在命令提示中執行互動式案例。

```
import (
 "context"
 "errors"
 "log"
 "strings"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// ActivityLog separates the steps of this scenario into individual functions so
 that
// they are simpler to read and understand.
type ActivityLog struct {
 helper       IScenarioHelper
 questioner   demotools.IQuestioner
 resources    Resources
 cognitoActor *actions.CognitoActions
}

// NewActivityLog constructs a new activity log runner.
func NewActivityLog(sdkConfig aws.Config, questioner demotools.IQuestioner,
 helper IScenarioHelper) ActivityLog {
 scenario := ActivityLog{
  helper:       helper,
  questioner:   questioner,
  resources:    Resources{},
  cognitoActor: &actions.CognitoActions{CognitoClient:
 cognitoidentityprovider.NewFromConfig(sdkConfig)},
 }
 scenario.resources.init(scenario.cognitoActor, questioner)
 return scenario
}

// AddUserToPool selects a user from the known users table and uses administrator
 credentials to add the user to the user pool.
func (runner *ActivityLog) AddUserToPool(ctx context.Context, userPoolId string,
 tableName string) (string, string) {
```

```
 log.Println("To facilitate this example, let's add a user to the user pool using
 administrator privileges.")
 users, err := runner.helper.GetKnownUsers(ctx, tableName)
 if err != nil {
  panic(err)
 }
 user := users.Users[0]
 log.Printf("Adding known user %v to the user pool.\n", user.UserName)
 err = runner.cognitoActor.AdminCreateUser(ctx, userPoolId, user.UserName,
 user.UserEmail)
 if err != nil {
  panic(err)
 }
 pwSet := false
 password := runner.questioner.AskPassword("\nEnter a password that has at least
 eight characters, uppercase, lowercase, numbers and symbols.\n"+
  "(the password will not display as you type):", 8)
 for !pwSet {
  log.Printf("\nSetting password for user '%v'.\n", user.UserName)
  err = runner.cognitoActor.AdminSetUserPassword(ctx, userPoolId, user.UserName,
 password)
  if err != nil {
   var invalidPassword *types.InvalidPasswordException
   if errors.As(err, &invalidPassword) {
    password = runner.questioner.AskPassword("\nEnter another password:", 8)
   } else {
    panic(err)
   }
  } else {
   pwSet = true
  }
 }

 log.Println(strings.Repeat("-", 88))

 return user.UserName, password
}

// AddActivityLogTrigger adds a Lambda handler as an invocation target for the
 PostAuthentication trigger.
func (runner *ActivityLog) AddActivityLogTrigger(ctx context.Context, userPoolId
 string, activityLogArn string) {
 log.Println("Let's add a Lambda function to handle the PostAuthentication
 trigger from Cognito.\n" +
```

```
  "This trigger happens after a user is authenticated, and lets your function
 take action, such as logging\n" +
  "the outcome.")
 err := runner.cognitoActor.UpdateTriggers(
  ctx, userPoolId,
  actions.TriggerInfo{Trigger: actions.PostAuthentication, HandlerArn:
 aws.String(activityLogArn)})
 if err != nil {
  panic(err)
 }
 runner.resources.triggers = append(runner.resources.triggers,
 actions.PostAuthentication)
 log.Printf("Lambda function %v added to user pool %v to handle
 PostAuthentication Cognito trigger.\n",
  activityLogArn, userPoolId)

 log.Println(strings.Repeat("-", 88))
}

// SignInUser signs in as the specified user.
func (runner *ActivityLog) SignInUser(ctx context.Context, clientId string,
 userName string, password string) {
 log.Printf("Now we'll sign in user %v and check the results in the logs and the
 DynamoDB table.", userName)
 runner.questioner.Ask("Press Enter when you're ready.")
 authResult, err := runner.cognitoActor.SignIn(ctx, clientId, userName, password)
 if err != nil {
  panic(err)
 }
 log.Println("Sign in successful.",
  "The PostAuthentication Lambda handler writes custom information to CloudWatch
 Logs.")

 runner.resources.userAccessTokens = append(runner.resources.userAccessTokens,
 *authResult.AccessToken)
}

// GetKnownUserLastLogin gets the login info for a user from the Amazon DynamoDB
 table and displays it.
func (runner *ActivityLog) GetKnownUserLastLogin(ctx context.Context, tableName
 string, userName string) {
 log.Println("The PostAuthentication handler also writes login data to the
 DynamoDB table.")
 runner.questioner.Ask("Press Enter when you're ready to continue.")
```

```
 users, err := runner.helper.GetKnownUsers(ctx, tableName)
 if err != nil {
  panic(err)
 }
 for _, user := range users.Users {
  if user.UserName == userName {
   log.Println("The last login info for the user in the known users table is:")
   log.Printf("\t%+v", *user.LastLogin)
  }
 }
 log.Println(strings.Repeat("-", 88))
}

// Run runs the scenario.
func (runner *ActivityLog) Run(ctx context.Context, stackName string) {
 defer func() {
  if r := recover(); r != nil {
   log.Println("Something went wrong with the demo.")
   runner.resources.Cleanup(ctx)
  }
 }()

 log.Println(strings.Repeat("-", 88))
 log.Printf("Welcome\n")

 log.Println(strings.Repeat("-", 88))

 stackOutputs, err := runner.helper.GetStackOutputs(ctx, stackName)
 if err != nil {
  panic(err)
 }
 runner.resources.userPoolId = stackOutputs["UserPoolId"]
 runner.helper.PopulateUserTable(ctx, stackOutputs["TableName"])
 userName, password := runner.AddUserToPool(ctx, stackOutputs["UserPoolId"],
 stackOutputs["TableName"])

 runner.AddActivityLogTrigger(ctx, stackOutputs["UserPoolId"],
 stackOutputs["ActivityLogFunctionArn"])
 runner.SignInUser(ctx, stackOutputs["UserPoolClientId"], userName, password)
 runner.helper.ListRecentLogEvents(ctx, stackOutputs["ActivityLogFunction"])
 runner.GetKnownUserLastLogin(ctx, stackOutputs["TableName"], userName)

 runner.resources.Cleanup(ctx)
```

```
 log.Println(strings.Repeat("-", 88))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("-", 88))
}
```

使用 Lambda 函數來處理 PostAuthentication 觸發條件。

```
import (
 "context"
 "fmt"
 "log"
 "os"
 "time"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 dynamodbtypes "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

const TABLE_NAME = "TABLE_NAME"

// LoginInfo defines structured login data that can be marshalled to a DynamoDB
 format.
type LoginInfo struct {
 UserPoolId string `dynamodbav:"UserPoolId"`
 ClientId   string `dynamodbav:"ClientId"`
 Time       string `dynamodbav:"Time"`
}

// UserInfo defines structured user data that can be marshalled to a DynamoDB
 format.
type UserInfo struct {
 UserName  string    `dynamodbav:"UserName"`
 UserEmail string    `dynamodbav:"UserEmail"`
 LastLogin LoginInfo `dynamodbav:"LastLogin"`
}
```

```
// GetKey marshals the user email value to a DynamoDB key format.
func (user UserInfo) GetKey() map[string]dynamodbtypes.AttributeValue {
 userEmail, err := attributevalue.Marshal(user.UserEmail)
 if err != nil {
  panic(err)
 }
 return map[string]dynamodbtypes.AttributeValue{"UserEmail": userEmail}
}

type handler struct {
 dynamoClient *dynamodb.Client
}

// HandleRequest handles the PostAuthentication event by writing custom data to
 the logs and
// to an Amazon DynamoDB table.
func (h *handler) HandleRequest(ctx context.Context,
 event events.CognitoEventUserPoolsPostAuthentication)
 (events.CognitoEventUserPoolsPostAuthentication, error) {
 log.Printf("Received post authentication trigger from %v for user '%v'",
 event.TriggerSource, event.UserName)
 tableName := os.Getenv(TABLE_NAME)
 user := UserInfo{
  UserName:  event.UserName,
  UserEmail: event.Request.UserAttributes["email"],
  LastLogin: LoginInfo{
   UserPoolId: event.UserPoolID,
   ClientId:   event.CallerContext.ClientID,
   Time:       time.Now().Format(time.UnixDate),
  },
 }
 // Write to CloudWatch Logs.
 fmt.Printf("%#v", user)

 // Also write to an external system. This examples uses DynamoDB to demonstrate.
 userMap, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshal to DynamoDB map. Here's why: %v\n", err)
 } else if len(userMap) == 0 {
  log.Printf("User info marshaled to an empty map.")
 } else {
  _, err := h.dynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
   Item:      userMap,
```

```
    TableName: aws.String(tableName),
   })
   if err != nil {
    log.Printf("Couldn't write to DynamoDB. Here's why: %v\n", err)
   } else {
    log.Printf("Wrote user info to DynamoDB table %v.\n", tableName)
   }
  }

  return event, nil
}

func main() {
  ctx := context.Background()
  sdkConfig, err := config.LoadDefaultConfig(ctx)
  if err != nil {
   log.Panicln(err)
  }
  h := handler{
   dynamoClient: dynamodb.NewFromConfig(sdkConfig),
  }
  lambda.Start(h.HandleRequest)
}
```

建立執行一般任務的 struct。

```
import (
 "context"
 "log"
 "strings"
 "time"
 "user_pools_and_lambda_triggers/actions"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)
```

```
// IScenarioHelper defines common functions used by the workflows in this
 example.
type IScenarioHelper interface {
 Pause(secs int)
 GetStackOutputs(ctx context.Context, stackName string) (actions.StackOutputs,
 error)
 PopulateUserTable(ctx context.Context, tableName string)
 GetKnownUsers(ctx context.Context, tableName string) (actions.UserList, error)
 AddKnownUser(ctx context.Context, tableName string, user actions.User)
 ListRecentLogEvents(ctx context.Context, functionName string)
}

// ScenarioHelper contains AWS wrapper structs used by the workflows in this
 example.
type ScenarioHelper struct {
 questioner  demotools.IQuestioner
 dynamoActor *actions.DynamoActions
 cfnActor    *actions.CloudFormationActions
 cwlActor    *actions.CloudWatchLogsActions
 isTestRun   bool
}

// NewScenarioHelper constructs a new scenario helper.
func NewScenarioHelper(sdkConfig aws.Config, questioner demotools.IQuestioner)
 ScenarioHelper {
 scenario := ScenarioHelper{
  questioner:  questioner,
  dynamoActor: &actions.DynamoActions{DynamoClient:
 dynamodb.NewFromConfig(sdkConfig)},
  cfnActor:    &actions.CloudFormationActions{CfnClient:
 cloudformation.NewFromConfig(sdkConfig)},
  cwlActor:    &actions.CloudWatchLogsActions{CwlClient:
 cloudwatchlogs.NewFromConfig(sdkConfig)},
 }
 return scenario
}

// Pause waits for the specified number of seconds.
func (helper ScenarioHelper) Pause(secs int) {
 if !helper.isTestRun {
  time.Sleep(time.Duration(secs) * time.Second)
 }
}
```

```
// GetStackOutputs gets the outputs from the specified CloudFormation stack in a
 structured format.
func (helper ScenarioHelper) GetStackOutputs(ctx context.Context, stackName
 string) (actions.StackOutputs, error) {
 return helper.cfnActor.GetOutputs(ctx, stackName), nil
}

// PopulateUserTable fills the known user table with example data.
func (helper ScenarioHelper) PopulateUserTable(ctx context.Context, tableName
 string) {
 log.Printf("First, let's add some users to the DynamoDB %v table we'll use for
 this example.\n", tableName)
 err := helper.dynamoActor.PopulateTable(ctx, tableName)
 if err != nil {
  panic(err)
 }
}

// GetKnownUsers gets the users from the known users table in a structured
 format.
func (helper ScenarioHelper) GetKnownUsers(ctx context.Context, tableName string)
 (actions.UserList, error) {
 knownUsers, err := helper.dynamoActor.Scan(ctx, tableName)
 if err != nil {
  log.Printf("Couldn't get known users from table %v. Here's why: %v\n",
 tableName, err)
 }
 return knownUsers, err
}

// AddKnownUser adds a user to the known users table.
func (helper ScenarioHelper) AddKnownUser(ctx context.Context, tableName string,
 user actions.User) {
 log.Printf("Adding user '%v' with email '%v' to the DynamoDB known users
 table...\n",
  user.UserName, user.UserEmail)
 err := helper.dynamoActor.AddUser(ctx, tableName, user)
 if err != nil {
  panic(err)
 }
}

// ListRecentLogEvents gets the most recent log stream and events for the
 specified Lambda function and displays them.
```

```
func (helper ScenarioHelper) ListRecentLogEvents(ctx context.Context,
 functionName string) {
 log.Println("Waiting a few seconds to let Lambda write to CloudWatch Logs...")
 helper.Pause(10)
 log.Println("Okay, let's check the logs to find what's happened recently with
 your Lambda function.")
 logStream, err := helper.cwlActor.GetLatestLogStream(ctx, functionName)
 if err != nil {
  panic(err)
 }
 log.Printf("Getting some recent events from log stream %v\n",
 *logStream.LogStreamName)
 events, err := helper.cwlActor.GetLogEvents(ctx, functionName,
 *logStream.LogStreamName, 10)
 if err != nil {
  panic(err)
 }
 for _, event := range events {
  log.Printf("\t%v", *event.Message)
 }
 log.Println(strings.Repeat("-", 88))
}
```

建立包裝 Amazon Cognito 動作的 struct。

```
import (
 "context"
 "errors"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider"
 "github.com/aws/aws-sdk-go-v2/service/cognitoidentityprovider/types"
)

type CognitoActions struct {
 CognitoClient *cognitoidentityprovider.Client
}
```

```
// Trigger and TriggerInfo define typed data for updating an Amazon Cognito
 trigger.
type Trigger int

const (
 PreSignUp Trigger = iota
 UserMigration
 PostAuthentication
)

type TriggerInfo struct {
 Trigger    Trigger
 HandlerArn *string
}

// UpdateTriggers adds or removes Lambda triggers for a user pool. When a trigger
 is specified with a `nil` value,
// it is removed from the user pool.
func (actor CognitoActions) UpdateTriggers(ctx context.Context, userPoolId
 string, triggers ...TriggerInfo) error {
 output, err := actor.CognitoClient.DescribeUserPool(ctx,
 &cognitoidentityprovider.DescribeUserPoolInput{
  UserPoolId: aws.String(userPoolId),
 })
 if err != nil {
  log.Printf("Couldn't get info about user pool %v. Here's why: %v\n",
 userPoolId, err)
  return err
 }
 lambdaConfig := output.UserPool.LambdaConfig
 for _, trigger := range triggers {
  switch trigger.Trigger {
  case PreSignUp:
   lambdaConfig.PreSignUp = trigger.HandlerArn
  case UserMigration:
   lambdaConfig.UserMigration = trigger.HandlerArn
  case PostAuthentication:
   lambdaConfig.PostAuthentication = trigger.HandlerArn
  }
 }
 _, err = actor.CognitoClient.UpdateUserPool(ctx,
 &cognitoidentityprovider.UpdateUserPoolInput{
  UserPoolId:   aws.String(userPoolId),
```

```
  LambdaConfig: lambdaConfig,
 })
 if err != nil {
  log.Printf("Couldn't update user pool %v. Here's why: %v\n", userPoolId, err)
 }
 return err
}



// SignUp signs up a user with Amazon Cognito.
func (actor CognitoActions) SignUp(ctx context.Context, clientId string, userName
 string, password string, userEmail string) (bool, error) {
 confirmed := false
 output, err := actor.CognitoClient.SignUp(ctx,
 &cognitoidentityprovider.SignUpInput{
  ClientId: aws.String(clientId),
  Password: aws.String(password),
  Username: aws.String(userName),
  UserAttributes: []types.AttributeType{
   {Name: aws.String("email"), Value: aws.String(userEmail)},
  },
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't sign up user %v. Here's why: %v\n", userName, err)
  }
 } else {
  confirmed = output.UserConfirmed
 }
 return confirmed, err
}



// SignIn signs in a user to Amazon Cognito using a username and password
 authentication flow.
func (actor CognitoActions) SignIn(ctx context.Context, clientId string, userName
 string, password string) (*types.AuthenticationResultType, error) {
 var authResult *types.AuthenticationResultType
```

```
 output, err := actor.CognitoClient.InitiateAuth(ctx,
 &cognitoidentityprovider.InitiateAuthInput{
  AuthFlow:       "USER_PASSWORD_AUTH",
  ClientId:       aws.String(clientId),
  AuthParameters: map[string]string{"USERNAME": userName, "PASSWORD": password},
 })
 if err != nil {
  var resetRequired *types.PasswordResetRequiredException
  if errors.As(err, &resetRequired) {
   log.Println(*resetRequired.Message)
  } else {
   log.Printf("Couldn't sign in user %v. Here's why: %v\n", userName, err)
  }
 } else {
  authResult = output.AuthenticationResult
 }
 return authResult, err
}



// ForgotPassword starts a password recovery flow for a user. This flow typically
 sends a confirmation code
// to the user's configured notification destination, such as email.
func (actor CognitoActions) ForgotPassword(ctx context.Context, clientId string,
 userName string) (*types.CodeDeliveryDetailsType, error) {
 output, err := actor.CognitoClient.ForgotPassword(ctx,
 &cognitoidentityprovider.ForgotPasswordInput{
  ClientId: aws.String(clientId),
  Username: aws.String(userName),
 })
 if err != nil {
  log.Printf("Couldn't start password reset for user '%v'. Here;s why: %v\n",
 userName, err)
 }
 return output.CodeDeliveryDetails, err
}



// ConfirmForgotPassword confirms a user with a confirmation code and a new
 password.
func (actor CognitoActions) ConfirmForgotPassword(ctx context.Context, clientId
 string, code string, userName string, password string) error {
```

```
 _, err := actor.CognitoClient.ConfirmForgotPassword(ctx,
 &cognitoidentityprovider.ConfirmForgotPasswordInput{
  ClientId:         aws.String(clientId),
  ConfirmationCode: aws.String(code),
  Password:         aws.String(password),
  Username:         aws.String(userName),
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't confirm user %v. Here's why: %v", userName, err)
  }
 }
 return err
}



// DeleteUser removes a user from the user pool.
func (actor CognitoActions) DeleteUser(ctx context.Context, userAccessToken
 string) error {
 _, err := actor.CognitoClient.DeleteUser(ctx,
 &cognitoidentityprovider.DeleteUserInput{
  AccessToken: aws.String(userAccessToken),
 })
 if err != nil {
  log.Printf("Couldn't delete user. Here's why: %v\n", err)
 }
 return err
}



// AdminCreateUser uses administrator credentials to add a user to a user pool.
 This method leaves the user
// in a state that requires they enter a new password next time they sign in.
func (actor CognitoActions) AdminCreateUser(ctx context.Context, userPoolId
 string, userName string, userEmail string) error {
 _, err := actor.CognitoClient.AdminCreateUser(ctx,
 &cognitoidentityprovider.AdminCreateUserInput{
  UserPoolId:     aws.String(userPoolId),
  Username:       aws.String(userName),
```

```
  MessageAction:  types.MessageActionTypeSuppress,
  UserAttributes: []types.AttributeType{{Name: aws.String("email"), Value:
 aws.String(userEmail)}},
 })
 if err != nil {
  var userExists *types.UsernameExistsException
  if errors.As(err, &userExists) {
   log.Printf("User %v already exists in the user pool.", userName)
   err = nil
  } else {
   log.Printf("Couldn't create user %v. Here's why: %v\n", userName, err)
  }
 }
 return err
}



// AdminSetUserPassword uses administrator credentials to set a password for a
 user without requiring a
// temporary password.
func (actor CognitoActions) AdminSetUserPassword(ctx context.Context, userPoolId
 string, userName string, password string) error {
 _, err := actor.CognitoClient.AdminSetUserPassword(ctx,
 &cognitoidentityprovider.AdminSetUserPasswordInput{
  Password:   aws.String(password),
  UserPoolId: aws.String(userPoolId),
  Username:   aws.String(userName),
  Permanent:  true,
 })
 if err != nil {
  var invalidPassword *types.InvalidPasswordException
  if errors.As(err, &invalidPassword) {
   log.Println(*invalidPassword.Message)
  } else {
   log.Printf("Couldn't set password for user %v. Here's why: %v\n", userName,
 err)
  }
 }
 return err
}
```

建立包裝 DynamoDB 動作的 struct。

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/feature/dynamodb/attributevalue"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb"
 "github.com/aws/aws-sdk-go-v2/service/dynamodb/types"
)

// DynamoActions encapsulates the Amazon Simple Notification Service (Amazon SNS)
 actions
// used in the examples.
type DynamoActions struct {
 DynamoClient *dynamodb.Client
}

// User defines structured user data.
type User struct {
 UserName  string
 UserEmail string
 LastLogin *LoginInfo `dynamodbav:",omitempty"`
}

// LoginInfo defines structured custom login data.
type LoginInfo struct {
 UserPoolId string
 ClientId   string
 Time       string
}

// UserList defines a list of users.
type UserList struct {
 Users []User
}

// UserNameList returns the usernames contained in a UserList as a list of
 strings.
func (users *UserList) UserNameList() []string {
 names := make([]string, len(users.Users))
```

```
 for i := 0; i < len(users.Users); i++ {
  names[i] = users.Users[i].UserName
 }
 return names
}

// PopulateTable adds a set of test users to the table.
func (actor DynamoActions) PopulateTable(ctx context.Context, tableName string)
 error {
 var err error
 var item map[string]types.AttributeValue
 var writeReqs []types.WriteRequest
 for i := 1; i < 4; i++ {
  item, err = attributevalue.MarshalMap(User{UserName: fmt.Sprintf("test_user_
%v", i), UserEmail: fmt.Sprintf("test_email_%v@example.com", i)})
  if err != nil {
   log.Printf("Couldn't marshall user into DynamoDB format. Here's why: %v\n",
 err)
   return err
  }
  writeReqs = append(writeReqs, types.WriteRequest{PutRequest:
 &types.PutRequest{Item: item}})
 }
 _, err = actor.DynamoClient.BatchWriteItem(ctx, &dynamodb.BatchWriteItemInput{
  RequestItems: map[string][]types.WriteRequest{tableName: writeReqs},
 })
 if err != nil {
  log.Printf("Couldn't populate table %v with users. Here's why: %v\n",
 tableName, err)
 }
 return err
}

// Scan scans the table for all items.
func (actor DynamoActions) Scan(ctx context.Context, tableName string) (UserList,
 error) {
 var userList UserList
 output, err := actor.DynamoClient.Scan(ctx, &dynamodb.ScanInput{
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't scan table %v for items. Here's why: %v\n", tableName,
 err)
 } else {
```

```
  err = attributevalue.UnmarshalListOfMaps(output.Items, &userList.Users)
  if err != nil {
   log.Printf("Couldn't unmarshal items into users. Here's why: %v\n", err)
  }
 }
 return userList, err
}

// AddUser adds a user item to a table.
func (actor DynamoActions) AddUser(ctx context.Context, tableName string, user
 User) error {
 userItem, err := attributevalue.MarshalMap(user)
 if err != nil {
  log.Printf("Couldn't marshall user to item. Here's why: %v\n", err)
 }
 _, err = actor.DynamoClient.PutItem(ctx, &dynamodb.PutItemInput{
  Item:      userItem,
  TableName: aws.String(tableName),
 })
 if err != nil {
  log.Printf("Couldn't put item in table %v. Here's why: %v", tableName, err)
 }
 return err
}
```

建立包裝 CloudWatch Logs 動作的 struct。

```
import (
 "context"
 "fmt"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs"
 "github.com/aws/aws-sdk-go-v2/service/cloudwatchlogs/types"
)

type CloudWatchLogsActions struct {
 CwlClient *cloudwatchlogs.Client
}
```

```
// GetLatestLogStream gets the most recent log stream for a Lambda function.
func (actor CloudWatchLogsActions) GetLatestLogStream(ctx context.Context,
 functionName string) (types.LogStream, error) {
 var logStream types.LogStream
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.DescribeLogStreams(ctx,
 &cloudwatchlogs.DescribeLogStreamsInput{
  Descending:   aws.Bool(true),
  Limit:        aws.Int32(1),
  LogGroupName: aws.String(logGroupName),
  OrderBy:      types.OrderByLastEventTime,
 })
 if err != nil {
  log.Printf("Couldn't get log streams for log group %v. Here's why: %v\n",
 logGroupName, err)
 } else {
  logStream = output.LogStreams[0]
 }
 return logStream, err
}

// GetLogEvents gets the most recent eventCount events from the specified log
 stream.
func (actor CloudWatchLogsActions) GetLogEvents(ctx context.Context, functionName
 string, logStreamName string, eventCount int32) (
 []types.OutputLogEvent, error) {
 var events []types.OutputLogEvent
 logGroupName := fmt.Sprintf("/aws/lambda/%s", functionName)
 output, err := actor.CwlClient.GetLogEvents(ctx,
 &cloudwatchlogs.GetLogEventsInput{
  LogStreamName: aws.String(logStreamName),
  Limit:         aws.Int32(eventCount),
  LogGroupName:  aws.String(logGroupName),
 })
 if err != nil {
  log.Printf("Couldn't get log event for log stream %v. Here's why: %v\n",
 logStreamName, err)
 } else {
  events = output.Events
 }
 return events, err
}
```

建立包裝 CloudFormation 動作的結構。

```
import (
 "context"
 "log"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/service/cloudformation"
)

// StackOutputs defines a map of outputs from a specific stack.
type StackOutputs map[string]string

type CloudFormationActions struct {
 CfnClient *cloudformation.Client
}

// GetOutputs gets the outputs from a CloudFormation stack and puts them into a
 structured format.
func (actor CloudFormationActions) GetOutputs(ctx context.Context, stackName
 string) StackOutputs {
 output, err := actor.CfnClient.DescribeStacks(ctx,
 &cloudformation.DescribeStacksInput{
  StackName: aws.String(stackName),
 })
 if err != nil || len(output.Stacks) == 0 {
  log.Panicf("Couldn't find a CloudFormation stack named %v. Here's why: %v\n",
 stackName, err)
 }
 stackOutputs := StackOutputs{}
 for _, out := range output.Stacks[0].Outputs {
  stackOutputs[*out.OutputKey] = *out.OutputValue
 }
 return stackOutputs
}
```

清除資源。

```
import (
 "context"
 "log"
 "user_pools_and_lambda_triggers/actions"

 "github.com/awsdocs/aws-doc-sdk-examples/gov2/demotools"
)

// Resources keeps track of AWS resources created during an example and handles
// cleanup when the example finishes.
type Resources struct {
 userPoolId       string
 userAccessTokens []string
 triggers         []actions.Trigger

 cognitoActor *actions.CognitoActions
 questioner   demotools.IQuestioner
}

func (resources *Resources) init(cognitoActor *actions.CognitoActions, questioner
 demotools.IQuestioner) {
 resources.userAccessTokens = []string{}
 resources.triggers = []actions.Trigger{}
 resources.cognitoActor = cognitoActor
 resources.questioner = questioner
}

// Cleanup deletes all AWS resources created during an example.
func (resources *Resources) Cleanup(ctx context.Context) {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong during cleanup.\n%v\n", r)
   log.Println("Use the AWS Management Console to remove any remaining resources
 \n" +
    "that were created for this scenario.")
  }
 }()

 wantDelete := resources.questioner.AskBool("Do you want to remove all of the AWS
 resources that were created "+
  "during this demo (y/n)?", "y")
 if wantDelete {
```

```
  for _, accessToken := range resources.userAccessTokens {
   err := resources.cognitoActor.DeleteUser(ctx, accessToken)
   if err != nil {
    log.Println("Couldn't delete user during cleanup.")
    panic(err)
   }
   log.Println("Deleted user.")
  }
  triggerList := make([]actions.TriggerInfo, len(resources.triggers))
  for i := 0; i < len(resources.triggers); i++ {
   triggerList[i] = actions.TriggerInfo{Trigger: resources.triggers[i],
 HandlerArn: nil}
  }
  err := resources.cognitoActor.UpdateTriggers(ctx, resources.userPoolId,
 triggerList...)
  if err != nil {
   log.Println("Couldn't update Cognito triggers during cleanup.")
   panic(err)
  }
  log.Println("Removed Cognito triggers from user pool.")
 } else {
  log.Println("Be sure to remove resources when you're done with them to avoid
 unexpected charges!")
 }
}
```

- 如需 API 詳細資訊，請參閱《適用於 Go 的 AWS SDK API 參考》中的下列主題。
  - AdminCreateUser
  - AdminSetUserPassword
  - DeleteUser
  - InitiateAuth
  - UpdateUserPool

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

Lambda 的無伺服器範例

下列程式碼範例示範如何使用 Lambda AWS SDKs。

範例

- 連線至 Lambda 函數中的 Amazon RDS 資料庫
- 使用 Kinesis 觸發條件調用 Lambda 函數
- 使用 DynamoDB 觸發條件調用 Lambda 函數
- 使用 Amazon DocumentDB 觸發條件調用 Lambda 函數
- 使用 Amazon MSK 觸發條件調用 Lambda 函數
- 使用 Amazon S3 觸發條件調用 Lambda 函數
- 使用 Amazon SNS 觸發條件調用 Lambda 函數
- 使用 Amazon SQS 觸發條件調用 Lambda 函數
- 使用 Kinesis 觸發條件報告 Lambda 函數的批次項目失敗
- 使用 DynamoDB 觸發條件報告 Lambda 函式的批次項目失敗
- 使用 Amazon SQS 觸發條件報告 Lambda 函數的批次項目失敗

連線至 Lambda 函數中的 Amazon RDS 資料庫

下列程式碼範例示範如何實作連線至 RDS 資料庫的 Lambda 函數。該函數會提出簡單的資料庫請求並傳回結果。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 連線至 Lambda 函式中的 Amazon RDS 資料庫。

```
using System.Data;
using System.Text.Json;
```

```
using Amazon.Lambda.APIGatewayEvents;
using Amazon.Lambda.Core;
using MySql.Data.MySqlClient;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace aws_rds;

public class InputModel
{
    public string key1 { get; set; }
    public string key2 { get; set; }
}

public class Function
{
    /// <summary>
    // Handles the Lambda function execution for connecting to RDS using IAM
 authentication.
    /// </summary>
    /// <param name="input">The input event data passed to the Lambda function</
param>
    /// <param name="context">The Lambda execution context that provides runtime
 information</param>
    /// <returns>A response object containing the execution result</returns>

    public async Task<APIGatewayProxyResponse>
 FunctionHandler(APIGatewayProxyRequest request, ILambdaContext context)
    {
        // Sample Input: {"body": "{\"key1\":\"20\", \"key2\":\"25\"}"}
        var input = JsonSerializer.Deserialize<InputModel>(request.Body);

        /// Obtain authentication token
        var authToken = RDSAuthTokenGenerator.GenerateAuthToken(
            Environment.GetEnvironmentVariable("RDS_ENDPOINT"),
            Convert.ToInt32(Environment.GetEnvironmentVariable("RDS_PORT")),
            Environment.GetEnvironmentVariable("RDS_USERNAME")
        );

        /// Build the Connection String with the Token
```

```
        string connectionString =
$"Server={Environment.GetEnvironmentVariable("RDS_ENDPOINT")};" +

$"Port={Environment.GetEnvironmentVariable("RDS_PORT")};" +

$"Uid={Environment.GetEnvironmentVariable("RDS_USERNAME")};" +
                                  $"Pwd={authToken};";


        try
        {
            await using var connection = new MySqlConnection(connectionString);
            await connection.OpenAsync();

            const string sql = "SELECT @param1 + @param2 AS Sum";

            await using var command = new MySqlCommand(sql, connection);
            command.Parameters.AddWithValue("@param1", int.Parse(input.key1 ??
"0"));
            command.Parameters.AddWithValue("@param2", int.Parse(input.key2 ??
"0"));

            await using var reader = await command.ExecuteReaderAsync();
            if (await reader.ReadAsync())
            {
                int result = reader.GetInt32("Sum");

                //Sample Response: {"statusCode":200,"body":"{\"message\":\"The
sum is: 45\"}","isBase64Encoded":false}
                return new APIGatewayProxyResponse
                {
                    StatusCode = 200,
                    Body = JsonSerializer.Serialize(new { message = $"The sum is:
{result}" })
                };
            }

        }
        catch (Exception ex)
        {
            Console.WriteLine($"Error: {ex.Message}");
        }

        return new APIGatewayProxyResponse
```

```
        {
            StatusCode = 500,
            Body = JsonSerializer.Serialize(new { error = "Internal server
 error" })
        };
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
/*
Golang v2 code here.
*/

package main

import (
 "context"
 "database/sql"
 "encoding/json"
 "fmt"
 "os"

 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/feature/rds/auth"
 _ "github.com/go-sql-driver/mysql"
)

type MyEvent struct {
 Name string `json:"name"`
```

```
}

func HandleRequest(event *MyEvent) (map[string]interface{}, error) {

 var dbName string = os.Getenv("DatabaseName")
 var dbUser string = os.Getenv("DatabaseUser")
 var dbHost string = os.Getenv("DBHost") // Add hostname without https
 var dbPort int = os.Getenv("Port")      // Add port number
 var dbEndpoint string = fmt.Sprintf("%s:%d", dbHost, dbPort)
 var region string = os.Getenv("AWS_REGION")

 cfg, err := config.LoadDefaultConfig(context.TODO())
 if err != nil {
  panic("configuration error: " + err.Error())
 }

 authenticationToken, err := auth.BuildAuthToken(
  context.TODO(), dbEndpoint, region, dbUser, cfg.Credentials)
 if err != nil {
  panic("failed to create authentication token: " + err.Error())
 }

 dsn := fmt.Sprintf("%s:%s@tcp(%s)/%s?tls=true&allowCleartextPasswords=true",
  dbUser, authenticationToken, dbEndpoint, dbName,
 )

 db, err := sql.Open("mysql", dsn)
 if err != nil {
  panic(err)
 }

 defer db.Close()

 var sum int
 err = db.QueryRow("SELECT ?+? AS sum", 3, 2).Scan(&sum)
 if err != nil {
  panic(err)
 }
 s := fmt.Sprint(sum)
 message := fmt.Sprintf("The selected sum is: %s", s)

 messageBytes, err := json.Marshal(message)
 if err != nil {
  return nil, err
```

```
 }

 messageString := string(messageBytes)
 return map[string]interface{}{
  "statusCode": 200,
  "headers":    map[string]string{"Content-Type": "application/json"},
  "body":       messageString,
 }, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyRequestEvent;
import com.amazonaws.services.lambda.runtime.events.APIGatewayProxyResponseEvent;
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;
import software.amazon.awssdk.regions.Region;
import software.amazon.awssdk.services.rdsdata.RdsDataClient;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementRequest;
import software.amazon.awssdk.services.rdsdata.model.ExecuteStatementResponse;
import software.amazon.awssdk.services.rdsdata.model.Field;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
```

```
public class RdsLambdaHandler implements
 RequestHandler<APIGatewayProxyRequestEvent, APIGatewayProxyResponseEvent> {

    @Override
    public APIGatewayProxyResponseEvent handleRequest(APIGatewayProxyRequestEvent
 event, Context context) {
        APIGatewayProxyResponseEvent response = new
 APIGatewayProxyResponseEvent();

        try {
            // Obtain auth token
            String token = createAuthToken();

            // Define connection configuration
            String connectionString = String.format("jdbc:mysql://%s:%s/%s?
useSSL=true&requireSSL=true",
                    System.getenv("ProxyHostName"),
                    System.getenv("Port"),
                    System.getenv("DBName"));

            // Establish a connection to the database
            try (Connection connection =
 DriverManager.getConnection(connectionString, System.getenv("DBUserName"),
 token);
                 PreparedStatement statement =
 connection.prepareStatement("SELECT ? + ? AS sum")) {

                statement.setInt(1, 3);
                statement.setInt(2, 2);

                try (ResultSet resultSet = statement.executeQuery()) {
                    if (resultSet.next()) {
                        int sum = resultSet.getInt("sum");
                        response.setStatusCode(200);
                        response.setBody("The selected sum is: " + sum);
                    }
                }
            }

        } catch (Exception e) {
            response.setStatusCode(500);
            response.setBody("Error: " + e.getMessage());
        }
```

```
        return response;
    }

    private String createAuthToken() {
        // Create RDS Data Service client
        RdsDataClient rdsDataClient = RdsDataClient.builder()
                .region(Region.of(System.getenv("AWS_REGION")))
                .credentialsProvider(DefaultCredentialsProvider.create())
                .build();

        // Define authentication request
        ExecuteStatementRequest request = ExecuteStatementRequest.builder()
                .resourceArn(System.getenv("ProxyHostName"))
                .secretArn(System.getenv("DBUserName"))
                .database(System.getenv("DBName"))
                .sql("SELECT 'RDS IAM Authentication'")
                .build();

        // Execute request and obtain authentication token
        ExecuteStatementResponse response =
 rdsDataClient.executeStatement(request);
        Field tokenField = response.records().get(0).get(0);

        return tokenField.stringValue();
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
/*
Node.js code here.
*/
// ES6+ example
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

async function createAuthToken() {
  // Define connection authentication parameters
  const dbinfo = {

    hostname: process.env.ProxyHostName,
    port: process.env.Port,
    username: process.env.DBUserName,
    region: process.env.AWS_REGION,

  }

  // Create RDS Signer object
  const signer = new Signer(dbinfo);

  // Request authorization token from RDS, specifying the username
  const token = await signer.getAuthToken();
  return token;
}

async function dbOps() {

  // Obtain auth token
  const token = await createAuthToken();
  // Define connection configuration
  let connectionConfig = {
    host: process.env.ProxyHostName,
    user: process.env.DBUserName,
    password: token,
    database: process.env.DBName,
    ssl: 'Amazon RDS'
  }
  // Create the connection to the DB
  const conn = await mysql.createConnection(connectionConfig);
  // Obtain the result of the query
  const [res,] = await conn.execute('select ?+? as sum', [3, 2]);
  return res;
```

```
}

export const handler = async (event) => {
  // Execute database flow
  const result = await dbOps();
  // Return result
  return {
    statusCode: 200,
    body: JSON.stringify("The selected sum is: " + result[0].sum)
  }
};
```

使用 TypeScript 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
import { Signer } from "@aws-sdk/rds-signer";
import mysql from 'mysql2/promise';

// RDS settings
// Using '!' (non-null assertion operator) to tell the TypeScript compiler that
 the DB settings are not null or undefined,
const proxy_host_name = process.env.PROXY_HOST_NAME!
const port = parseInt(process.env.PORT!)
const db_name = process.env.DB_NAME!
const db_user_name = process.env.DB_USER_NAME!
const aws_region = process.env.AWS_REGION!


async function createAuthToken(): Promise<string> {

    // Create RDS Signer object
    const signer = new Signer({
        hostname: proxy_host_name,
        port: port,
        region: aws_region,
        username: db_user_name
    });

    // Request authorization token from RDS, specifying the username
    const token = await signer.getAuthToken();
    return token;
```

```
}

async function dbOps(): Promise<mysql.QueryResult | undefined> {
    try {
        // Obtain auth token
        const token = await createAuthToken();
        const conn = await mysql.createConnection({
            host: proxy_host_name,
            user: db_user_name,
            password: token,
            database: db_name,
            ssl: 'Amazon RDS' // Ensure you have the CA bundle for SSL connection
        });
        const [rows, fields] = await conn.execute('SELECT ? + ? AS sum', [3, 2]);
        console.log('result:', rows);
        return rows;
    }
    catch (err) {
        console.log(err);
    }
}

export const lambdaHandler = async (event: any): Promise<{ statusCode: number;
 body: string }> => {
    // Execute database flow
    const result = await dbOps();

    // Return error is result is undefined
    if (result == undefined)
        return {
            statusCode: 500,
            body: JSON.stringify(`Error with connection to DB host`)
        }

    // Return result
    return {
        statusCode: 200,
        body: JSON.stringify(`The selected sum is: ${result[0].sum}`)
    };
};
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
<?php
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;
use Aws\Rds\AuthTokenGenerator;
use Aws\Credentials\CredentialProvider;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }


    private function getAuthToken(): string {
        // Define connection authentication parameters
        $dbConnection = [
            'hostname' => getenv('DB_HOSTNAME'),
            'port' => getenv('DB_PORT'),
            'username' => getenv('DB_USERNAME'),
            'region' => getenv('AWS_REGION'),
        ];
```

```
        // Create RDS AuthTokenGenerator object
        $generator = new
AuthTokenGenerator(CredentialProvider::defaultProvider());

        // Request authorization token from RDS, specifying the username
        return $generator->createToken(
            $dbConnection['hostname'] . ':' . $dbConnection['port'],
            $dbConnection['region'],
            $dbConnection['username']
        );
    }

    private function getQueryResults() {
        // Obtain auth token
        $token = $this->getAuthToken();

        // Define connection configuration
        $connectionConfig = [
            'host' => getenv('DB_HOSTNAME'),
            'user' => getenv('DB_USERNAME'),
            'password' => $token,
            'database' => getenv('DB_NAME'),
        ];

        // Create the connection to the DB
        $conn = new PDO(

"mysql:host={$connectionConfig['host']};dbname={$connectionConfig['database']}",
            $connectionConfig['user'],
            $connectionConfig['password'],
            [
                PDO::MYSQL_ATTR_SSL_CA => '/path/to/rds-ca-2019-root.pem',
                PDO::MYSQL_ATTR_SSL_VERIFY_SERVER_CERT => true,
            ]
        );

        // Obtain the result of the query
        $stmt = $conn->prepare('SELECT ?+? AS sum');
        $stmt->execute([3, 2]);

        return $stmt->fetch(PDO::FETCH_ASSOC);
    }
```

```
    /**
     * @param mixed $event
     * @param Context $context
     * @return array
     */
    public function handle(mixed $event, Context $context): array
    {
        $this->logger->info("Processing query");

        // Execute database flow
        $result = $this->getQueryResults();

        return [
            'sum' => $result['sum']
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
import json
import os
import boto3
import pymysql

# RDS settings
proxy_host_name = os.environ['PROXY_HOST_NAME']
port = int(os.environ['PORT'])
db_name = os.environ['DB_NAME']
```

```
db_user_name = os.environ['DB_USER_NAME']
aws_region = os.environ['AWS_REGION']


# Fetch RDS Auth Token
def get_auth_token():
    client = boto3.client('rds')
    token = client.generate_db_auth_token(
        DBHostname=proxy_host_name,
        Port=port
        DBUsername=db_user_name
        Region=aws_region
    )
    return token

def lambda_handler(event, context):
    token = get_auth_token()
    try:
        connection = pymysql.connect(
            host=proxy_host_name,
            user=db_user_name,
            password=token,
            db=db_name,
            port=port,
            ssl={'ca': 'Amazon RDS'}  # Ensure you have the CA bundle for SSL
 connection
        )

        with connection.cursor() as cursor:
            cursor.execute('SELECT %s + %s AS sum', (3, 2))
            result = cursor.fetchone()

        return result

    except Exception as e:
        return (f"Error: {str(e)}")  # Return an error message if an exception
 occurs
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
# Ruby code here.

require 'aws-sdk-rds'
require 'json'
require 'mysql2'

def lambda_handler(event:, context:)
  endpoint = ENV['DBEndpoint'] # Add the endpoint without https"
  port = ENV['Port']           # 3306
  user = ENV['DBUser']
  region = ENV['DBRegion']     # 'us-east-1'
  db_name = ENV['DBName']

  credentials = Aws::Credentials.new(
    ENV['AWS_ACCESS_KEY_ID'],
    ENV['AWS_SECRET_ACCESS_KEY'],
    ENV['AWS_SESSION_TOKEN']
  )
  rds_client = Aws::RDS::AuthTokenGenerator.new(
    region: region,
    credentials: credentials
  )

  token = rds_client.auth_token(
    endpoint: endpoint+ ':' + port,
    user_name: user,
    region: region
  )

  begin
    conn = Mysql2::Client.new(
```

```
      host: endpoint,
      username: user,
      password: token,
      port: port,
      database: db_name,
      sslca: '/var/task/global-bundle.pem',
      sslverify: true,
      enable_cleartext_plugin: true
    )
    a = 3
    b = 2
    result = conn.query("SELECT #{a} + #{b} AS sum").first['sum']
    puts result
    conn.close
    {
      statusCode: 200,
      body: result.to_json
    }
  rescue => e
    puts "Database connection failed due to #{e}"
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 連線至 Lambda 函數中的 Amazon RDS 資料庫。

```
use aws_config::BehaviorVersion;
use aws_credential_types::provider::ProvideCredentials;
use aws_sigv4::{
    http_request::{sign, SignableBody, SignableRequest, SigningSettings},
    sign::v4,
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
```

```
use serde_json::{json, Value};
use sqlx::postgres::PgConnectOptions;
use std::env;
use std::time::{Duration, SystemTime};

const RDS_CERTS: &[u8] = include_bytes!("global-bundle.pem");

async fn generate_rds_iam_token(
    db_hostname: &str,
    port: u16,
    db_username: &str,
) -> Result<String, Error> {
    let config = aws_config::load_defaults(BehaviorVersion::v2024_03_28()).await;

    let credentials = config
        .credentials_provider()
        .expect("no credentials provider found")
        .provide_credentials()
        .await
        .expect("unable to load credentials");
    let identity = credentials.into();
    let region = config.region().unwrap().to_string();

    let mut signing_settings = SigningSettings::default();
    signing_settings.expires_in = Some(Duration::from_secs(900));
    signing_settings.signature_location =
 aws_sigv4::http_request::SignatureLocation::QueryParams;

    let signing_params = v4::SigningParams::builder()
        .identity(&identity)
        .region(&region)
        .name("rds-db")
        .time(SystemTime::now())
        .settings(signing_settings)
        .build()?;

    let url = format!(
        "https://{db_hostname}:{port}/?Action=connect&DBUser={db_user}",
        db_hostname = db_hostname,
        port = port,
        db_user = db_username
    );

    let signable_request =
```

```
        SignableRequest::new("GET", &url, std::iter::empty(),
 SignableBody::Bytes(&[]))
            .expect("signable request");

    let (signing_instructions, _signature) =
        sign(signable_request, &signing_params.into())?.into_parts();

    let mut url = url::Url::parse(&url).unwrap();
    for (name, value) in signing_instructions.params() {
        url.query_pairs_mut().append_pair(name, &value);
    }

    let response = url.to_string().split_off("https://".len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    run(service_fn(handler)).await
}

async fn handler(_event: LambdaEvent<Value>) -> Result<Value, Error> {
    let db_host = env::var("DB_HOSTNAME").expect("DB_HOSTNAME must be set");
    let db_port = env::var("DB_PORT")
        .expect("DB_PORT must be set")
        .parse::<u16>()
        .expect("PORT must be a valid number");
    let db_name = env::var("DB_NAME").expect("DB_NAME must be set");
    let db_user_name = env::var("DB_USERNAME").expect("DB_USERNAME must be set");

    let token = generate_rds_iam_token(&db_host, db_port, &db_user_name).await?;

    let opts = PgConnectOptions::new()
        .host(&db_host)
        .port(db_port)
        .username(&db_user_name)
        .password(&token)
        .database(&db_name)
        .ssl_root_cert_from_pem(RDS_CERTS.to_vec())
        .ssl_mode(sqlx::postgres::PgSslMode::Require);

    let pool = sqlx::postgres::PgPoolOptions::new()
        .connect_with(opts)
```

```
        .await?;

    let result: i32 = sqlx::query_scalar("SELECT $1 + $2")
        .bind(3)
        .bind(2)
        .fetch_one(&pool)
        .await?;

    println!("Result: {:?}", result);

    Ok(json!({
        "statusCode": 200,
        "content-type": "text/plain",
        "body": format!("The selected sum is: {result}")
    }))
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Kinesis 觸發條件調用 Lambda 函數

下列程式碼範例示範如何實作 Lambda 函數，以便接收在收到來自 Kinesis 串流的訊息時觸發的事件。此函數會擷取 Kinesis 承載、從 Base64 解碼，並記錄記錄內容。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
```

```
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegrationSampleCode;

public class Function
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task FunctionHandler(KinesisEvent evnt, ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return;
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                throw;
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
    }
```

```
    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent) error {
 if len(kinesisEvent.Records) == 0 {
  log.Printf("empty Kinesis event received")
  return nil
 }

 for _, record := range kinesisEvent.Records {
  log.Printf("processed Kinesis event with EventId: %v", record.EventID)
```

```
  recordDataBytes := record.Kinesis.Data
  recordDataText := string(recordDataBytes)
  log.Printf("record data: %v", recordDataText)
  // TODO: Do interesting work based on the new data
 }
 log.Printf("successfully processed %v records", len(kinesisEvent.Records))
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;

public class Handler implements RequestHandler<KinesisEvent, Void> {
    @Override
    public Void handleRequest(final KinesisEvent event, final Context context) {
        LambdaLogger logger = context.getLogger();
        if (event.getRecords().isEmpty()) {
            logger.log("Empty Kinesis Event received");
            return null;
```

```
        }
        for (KinesisEvent.KinesisEventRecord record : event.getRecords()) {
            try {
                logger.log("Processed Event with EventId: "+record.getEventID());
                String data = new String(record.getKinesis().getData().array());
                logger.log("Data:"+ data);
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex) {
                logger.log("An error occurred:"+ex.getMessage());
                throw ex;
            }
        }
        logger.log("Successfully processed:"+event.getRecords().size()+"
 records");
        return null;
    }

}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
```

```
    } catch (err) {
      console.error(`An error occurred ${err}`);
      throw err;
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

使用 TypeScript 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
```

```
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      throw err;
    }
    logger.info(`Successfully processed ${event.Records.length} records.`);
  }
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Kinesis\KinesisHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';
```

```
class Handler extends KinesisHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleKinesis(KinesisEvent $event, Context $context): void
    {
        $this->logger->info("Processing records");
        $records = $event->getRecords();
        foreach ($records as $record) {
            $data = $record->getData();
            $this->logger->info(json_encode($data));
            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 Kinesis 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import base64
def lambda_handler(event, context):

    for record in event['Records']:
        try:
            print(f"Processed Kinesis Event - EventID: {record['eventID']}")
            record_data = base64.b64decode(record['kinesis']
['data']).decode('utf-8')
            print(f"Record Data: {record_data}")
            # TODO: Do interesting work based on the new data
        except Exception as e:
            print(f"An error occurred {e}")
            raise e
    print(f"Successfully processed {len(event['Records'])} records.")
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來使用 Kinesis 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
```

```
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue => err
      $stderr.puts "An error occurred #{err}"
      raise err
    end
  end
  puts "Successfully processed #{event['Records'].length} records."
end

def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('UTF-8')
  # Placeholder for actual async work
  # You can use Ruby's asynchronous programming tools like async/await or fibers
 here.
  return data
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來使用 Kinesis 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::kinesis::KinesisEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) -> Result<(), Error>
 {
    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }
```

```
    event.payload.records.iter().for_each(|record| {
        tracing::info!("EventId:
 {}",record.event_id.as_deref().unwrap_or_default());

        let record_data = std::str::from_utf8(&record.kinesis.data);

        match record_data {
            Ok(data) => {
                // log the record data
                tracing::info!("Data: {}", data);
            }
            Err(e) => {
                tracing::error!("Error: {}", e);
            }
        }
    });

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 DynamoDB 觸發條件調用 Lambda 函數

下列程式碼範例示範如何實作 Lambda 函式，以便接收在收到來自 DynamoDB 串流的訊息時觸發的事件。函數會擷取 DynamoDB 承載並記下記錄內容。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public void FunctionHandler(DynamoDBEvent dynamoEvent, ILambdaContext
 context)
    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");

        foreach (var record in dynamoEvent.Records)
        {
            context.Logger.LogInformation($"Event ID: {record.EventID}");
```

```
            context.Logger.LogInformation($"Event Name: {record.EventName}");

            context.Logger.LogInformation(JsonSerializer.Serialize(record));
        }

        context.Logger.LogInformation("Stream processing complete.");
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-lambda-go/events"
 "fmt"
)

func HandleRequest(ctx context.Context, event events.DynamoDBEvent) (*string,
 error) {
 if len(event.Records) == 0 {
  return nil, fmt.Errorf("received empty event")
 }

 for _, record := range event.Records {
   LogDynamoDBRecord(record)
 }
```

```
 message := fmt.Sprintf("Records processed: %d", len(event.Records))
 return &message, nil
}

func main() {
 lambda.Start(HandleRequest)
}

func LogDynamoDBRecord(record events.DynamoDBEventRecord){
 fmt.Println(record.EventID)
 fmt.Println(record.EventName)
 fmt.Printf("%+v\n", record.Change)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 DynamoDB 事件。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import
 com.amazonaws.services.lambda.runtime.events.DynamodbEvent.DynamodbStreamRecord;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public class example implements RequestHandler<DynamodbEvent, Void> {

    private static final Gson GSON = new
 GsonBuilder().setPrettyPrinting().create();

    @Override
    public Void handleRequest(DynamodbEvent event, Context context) {
```

```
        System.out.println(GSON.toJson(event));
        event.getRecords().forEach(this::logDynamoDBRecord);
        return null;
    }

    private void logDynamoDBRecord(DynamodbStreamRecord record) {
        System.out.println(record.getEventID());
        System.out.println(record.getEventName());
        System.out.println("DynamoDB Record: " +
 GSON.toJson(record.getDynamodb()));
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 來使用 DynamoDB 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
};

const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

使用 TypeScript 搭配 Lambda 來使用 DynamoDB 事件。

```
export const handler = async (event, context) => {
    console.log(JSON.stringify(event, null, 2));
    event.Records.forEach(record => {
        logDynamoDBRecord(record);
    });
}
const logDynamoDBRecord = (record) => {
    console.log(record.eventID);
    console.log(record.eventName);
    console.log(`DynamoDB Record: ${JSON.stringify(record.dynamodb)}`);
};
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來使用 DynamoDB 事件。

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\DynamoDb\DynamoDbHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends DynamoDbHandler
{
    private StderrLogger $logger;
```

```
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleDynamoDb(DynamoDbEvent $event, Context $context): void
    {
        $this->logger->info("Processing DynamoDb table items");
        $records = $event->getRecords();

        foreach ($records as $record) {
            $eventName = $record->getEventName();
            $keys = $record->getKeys();
            $old = $record->getOldImage();
            $new = $record->getNewImage();

            $this->logger->info("Event Name:".$eventName."\n");
            $this->logger->info("Keys:". json_encode($keys)."\n");
            $this->logger->info("Old Image:". json_encode($old)."\n");
            $this->logger->info("New Image:". json_encode($new));

            // TODO: Do interesting work based on the new data

            // Any exception thrown will be logged and the invocation will be
 marked as failed
        }

        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords items");
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 DynamoDB 事件。

```
import json

def lambda_handler(event, context):
    print(json.dumps(event, indent=2))

    for record in event['Records']:
        log_dynamodb_record(record)

def log_dynamodb_record(record):
    print(record['eventID'])
    print(record['eventName'])
    print(f"DynamoDB Record: {json.dumps(record['dynamodb'])}")
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來使用 DynamoDB 事件。

```
def lambda_handler(event:, context:)
    return 'received empty event' if event['Records'].empty?

    event['Records'].each do |record|
      log_dynamodb_record(record)
    end

    "Records processed: #{event['Records'].length}"
  end

  def log_dynamodb_record(record)
    puts record['eventID']
    puts record['eventName']
    puts "DynamoDB Record: #{JSON.generate(record['dynamodb'])}"
  end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來使用 DynamoDB 事件。

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord},
   };


// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
```

```
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<Event>) ->Result<(), Error> {

    let records = &event.payload.records;
    tracing::info!("event payload: {:?}",records);
    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_dynamo_dbrecord(record);
    }

    tracing::info!("Dynamo db records processed");

    // Prepare the response
    Ok(())

}

fn log_dynamo_dbrecord(record: &EventRecord)-> Result<(), Error>{
    tracing::info!("EventId: {}", record.event_id);
    tracing::info!("EventName: {}", record.event_name);
    tracing::info!("DynamoDB Record: {:?}", record.change );
    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())
```

```
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Amazon DocumentDB 觸發條件調用 Lambda 函數

下列程式碼範例示範如何實作 Lambda 函數，以便接收在收到來自 DocumentDB 變更串流的訊息時觸發的事件。函數會擷取 DocumentDB 承載並記下記錄內容。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
using Amazon.Lambda.Core;
using System.Text.Json;
using System;
using System.Collections.Generic;
using System.Text.Json.Serialization;
//Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace LambdaDocDb;

public class Function
{

     /// <summary>
    /// Lambda function entry point to process Amazon DocumentDB events.
    /// </summary>
```

```
    /// <param name="event">The Amazon DocumentDB event.</param>
    /// <param name="context">The Lambda context object.</param>
    /// <returns>A string to indicate successful processing.</returns>
    public string FunctionHandler(Event evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Events)
        {
            ProcessDocumentDBEvent(record, context);
        }

        return "OK";
    }

     private void ProcessDocumentDBEvent(DocumentDBEventRecord record,
ILambdaContext context)
    {

        var eventData = record.Event;
        var operationType = eventData.OperationType;
        var databaseName = eventData.Ns.Db;
        var collectionName = eventData.Ns.Coll;
        var fullDocument = JsonSerializer.Serialize(eventData.FullDocument, new
JsonSerializerOptions { WriteIndented = true });

        context.Logger.LogLine($"Operation type: {operationType}");
        context.Logger.LogLine($"Database: {databaseName}");
        context.Logger.LogLine($"Collection: {collectionName}");
        context.Logger.LogLine($"Full document:\n{fullDocument}");
    }



    public class Event
    {
        [JsonPropertyName("eventSourceArn")]
        public string EventSourceArn { get; set; }

        [JsonPropertyName("events")]
        public List<DocumentDBEventRecord> Events { get; set; }

        [JsonPropertyName("eventSource")]
        public string EventSource { get; set; }
    }
```

```
public class DocumentDBEventRecord
{
    [JsonPropertyName("event")]
    public EventData Event { get; set; }
}

public class EventData
{
    [JsonPropertyName("_id")]
    public IdData Id { get; set; }

    [JsonPropertyName("clusterTime")]
    public ClusterTime ClusterTime { get; set; }

    [JsonPropertyName("documentKey")]
    public DocumentKey DocumentKey { get; set; }

    [JsonPropertyName("fullDocument")]
    public Dictionary<string, object> FullDocument { get; set; }

    [JsonPropertyName("ns")]
    public Namespace Ns { get; set; }

    [JsonPropertyName("operationType")]
    public string OperationType { get; set; }
}

public class IdData
{
    [JsonPropertyName("_data")]
    public string Data { get; set; }
}

public class ClusterTime
{
    [JsonPropertyName("$timestamp")]
    public Timestamp Timestamp { get; set; }
}

public class Timestamp
{
    [JsonPropertyName("t")]
    public long T { get; set; }
```

```
        [JsonPropertyName("i")]
        public int I { get; set; }
    }

    public class DocumentKey
    {
        [JsonPropertyName("_id")]
        public Id Id { get; set; }
    }

    public class Id
    {
        [JsonPropertyName("$oid")]
        public string Oid { get; set; }
    }

    public class Namespace
    {
        [JsonPropertyName("db")]
        public string Db { get; set; }

        [JsonPropertyName("coll")]
        public string Coll { get; set; }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
package main
```

```
import (
 "context"
 "encoding/json"
 "fmt"

 "github.com/aws/aws-lambda-go/lambda"
)

type Event struct {
 Events []Record `json:"events"`
}

type Record struct {
 Event struct {
  OperationType string `json:"operationType"`
  NS            struct {
   DB   string `json:"db"`
   Coll string `json:"coll"`
  } `json:"ns"`
  FullDocument interface{} `json:"fullDocument"`
 } `json:"event"`
}

func main() {
 lambda.Start(handler)
}

func handler(ctx context.Context, event Event) (string, error) {
 fmt.Println("Loading function")
 for _, record := range event.Events {
  logDocumentDBEvent(record)
 }

 return "OK", nil
}

func logDocumentDBEvent(record Record) {
 fmt.Printf("Operation type: %s\n", record.Event.OperationType)
 fmt.Printf("db: %s\n", record.Event.NS.DB)
 fmt.Printf("collection: %s\n", record.Event.NS.Coll)
 docBytes, _ := json.MarshalIndent(record.Event.FullDocument, "", "  ")
 fmt.Printf("Full document: %s\n", string(docBytes))
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
import java.util.List;
import java.util.Map;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;

public class Example implements RequestHandler<Map<String, Object>, String> {

    @SuppressWarnings("unchecked")
    @Override
    public String handleRequest(Map<String, Object> event, Context context) {
        List<Map<String, Object>> events = (List<Map<String, Object>>)
 event.get("events");
        for (Map<String, Object> record : events) {
            Map<String, Object> eventData = (Map<String, Object>)
 record.get("event");
            processEventData(eventData);
        }

        return "OK";
    }

    @SuppressWarnings("unchecked")
    private void processEventData(Map<String, Object> eventData) {
        String operationType = (String) eventData.get("operationType");
        System.out.println("operationType: %s".formatted(operationType));

        Map<String, Object> ns = (Map<String, Object>) eventData.get("ns");
```

```
        String db = (String) ns.get("db");
        System.out.println("db: %s".formatted(db));
        String coll = (String) ns.get("coll");
        System.out.println("coll: %s".formatted(coll));

        Map<String, Object> fullDocument = (Map<String, Object>)
 eventData.get("fullDocument");
        System.out.println("fullDocument: %s".formatted(fullDocument));
    }

}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
console.log('Loading function');
exports.handler = async (event, context) => {
    event.events.forEach(record => {
        logDocumentDBEvent(record);
    });
    return 'OK';
};

const logDocumentDBEvent = (record) => {
    console.log('Operation type: ' + record.event.operationType);
    console.log('db: ' + record.event.ns.db);
    console.log('collection: ' + record.event.ns.coll);
    console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

使用 TypeScript 搭配 Lambda 使用 Amazon DocumentDB 事件

```
import { DocumentDBEventRecord, DocumentDBEventSubscriptionContext } from 'aws-
lambda';

console.log('Loading function');

export const handler = async (
  event: DocumentDBEventSubscriptionContext,
  context: any
): Promise<string> => {
  event.events.forEach((record: DocumentDBEventRecord) => {
    logDocumentDBEvent(record);
  });
  return 'OK';
};

const logDocumentDBEvent = (record: DocumentDBEventRecord): void => {
  console.log('Operation type: ' + record.event.operationType);
  console.log('db: ' + record.event.ns.db);
  console.log('collection: ' + record.event.ns.coll);
  console.log('Full document:', JSON.stringify(record.event.fullDocument, null,
 2));
};
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
<?php
```

```
require __DIR__.'/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Handler;

class DocumentDBEventHandler implements Handler
{
    public function handle($event, Context $context): string
    {

        $events = $event['events'] ?? [];
        foreach ($events as $record) {
            $this->logDocumentDBEvent($record['event']);
        }
        return 'OK';
    }

    private function logDocumentDBEvent($event): void
    {
        // Extract information from the event record

        $operationType = $event['operationType'] ?? 'Unknown';
        $db = $event['ns']['db'] ?? 'Unknown';
        $collection = $event['ns']['coll'] ?? 'Unknown';
        $fullDocument = $event['fullDocument'] ?? [];

        // Log the event details

        echo "Operation type: $operationType\n";
        echo "Database: $db\n";
        echo "Collection: $collection\n";
        echo "Full document: " . json_encode($fullDocument, JSON_PRETTY_PRINT) .
 "\n";
    }
}
return new DocumentDBEventHandler();
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
import json

def lambda_handler(event, context):
    for record in event.get('events', []):
        log_document_db_event(record)
    return 'OK'

def log_document_db_event(record):
    event_data = record.get('event', {})
    operation_type = event_data.get('operationType', 'Unknown')
    db = event_data.get('ns', {}).get('db', 'Unknown')
    collection = event_data.get('ns', {}).get('coll', 'Unknown')
    full_document = event_data.get('fullDocument', {})

    print(f"Operation type: {operation_type}")
    print(f"db: {db}")
    print(f"collection: {collection}")
    print("Full document:", json.dumps(full_document, indent=2))
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
require 'json'

def lambda_handler(event:, context:)
  event['events'].each do |record|
    log_document_db_event(record)
  end
  'OK'
end

def log_document_db_event(record)
  event_data = record['event'] || {}
  operation_type = event_data['operationType'] || 'Unknown'
  db = event_data.dig('ns', 'db') || 'Unknown'
  collection = event_data.dig('ns', 'coll') || 'Unknown'
  full_document = event_data['fullDocument'] || {}

  puts "Operation type: #{operation_type}"
  puts "db: #{db}"
  puts "collection: #{collection}"
  puts "Full document: #{JSON.pretty_generate(full_document)}"
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 使用 Amazon DocumentDB 事件。

```
use lambda_runtime::{service_fn, tracing, Error, LambdaEvent};
use aws_lambda_events::{
    event::documentdb::{DocumentDbEvent, DocumentDbInnerEvent},
   };
```

```
// Built with the following dependencies:
//lambda_runtime = "0.11.1"
//serde_json = "1.0"
//tokio = { version = "1", features = ["macros"] }
//tracing = { version = "0.1", features = ["log"] }
//tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }
//aws_lambda_events = "0.15.0"

async fn function_handler(event: LambdaEvent<DocumentDbEvent>) ->Result<(),
 Error> {

    tracing::info!("Event Source ARN: {:?}", event.payload.event_source_arn);
    tracing::info!("Event Source: {:?}", event.payload.event_source);

    let records = &event.payload.events;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(());
    }

    for record in records{
        log_document_db_event(record);
    }

    tracing::info!("Document db records processed");

    // Prepare the response
    Ok(())

}

fn log_document_db_event(record: &DocumentDbInnerEvent)-> Result<(), Error>{
    tracing::info!("Change Event: {:?}", record.event);

    Ok(())

}

#[tokio::main]
async fn main() -> Result<(), Error> {
```

```
    tracing_subscriber::fmt()
    .with_max_level(tracing::Level::INFO)
    .with_target(false)
    .without_time()
    .init();

    let func = service_fn(function_handler);
    lambda_runtime::run(func).await?;
    Ok(())

}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Amazon MSK 觸發條件調用 Lambda 函數

下列程式碼範例示範如何實作 Lambda 函數，以便接收在收到來自 Amazon MSK 叢集的訊息時觸發的事件。函數會擷取 MSK 承載並記下記錄內容。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來取用 Amazon MSK 事件。

```
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.KafkaEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace MSKLambda;

public class Function
{


    /// <param name="input">The event for the Lambda function handler to
 process.</param>
    /// <param name="context">The ILambdaContext that provides methods for
 logging and describing the Lambda environment.</param>
    /// <returns></returns>
    public void FunctionHandler(KafkaEvent evnt, ILambdaContext context)
    {

        foreach (var record in evnt.Records)
        {
            Console.WriteLine("Key:" + record.Key);
            foreach (var eventRecord in record.Value)
            {
                var valueBytes = eventRecord.Value.ToArray();
                var valueText = Encoding.UTF8.GetString(valueBytes);

                Console.WriteLine("Message:" + valueText);
            }
        }
    }


}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來取用 Amazon MSK 事件。

```
package main

import (
 "encoding/base64"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.KafkaEvent) {
 for key, records := range event.Records {
  fmt.Println("Key:", key)

  for _, record := range records {
   fmt.Println("Record:", record)

   decodedValue, _ := base64.StdEncoding.DecodeString(record.Value)
   message := string(decodedValue)
   fmt.Println("Message:", message)
  }
 }
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來取用 Amazon MSK 事件。

```
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent;
import com.amazonaws.services.lambda.runtime.events.KafkaEvent.KafkaEventRecord;

import java.util.Base64;
import java.util.Map;

public class Example implements RequestHandler<KafkaEvent, Void> {

    @Override
    public Void handleRequest(KafkaEvent event, Context context) {
        for (Map.Entry<String, java.util.List<KafkaEventRecord>> entry :
 event.getRecords().entrySet()) {
            String key = entry.getKey();
            System.out.println("Key: " + key);

            for (KafkaEventRecord record : entry.getValue()) {
                System.out.println("Record: " + record);

                byte[] value = Base64.getDecoder().decode(record.getValue());
                String message = new String(value);
                System.out.println("Message: " + message);
            }
        }

        return null;
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 來取用 Amazon MSK 事件。

```
exports.handler = async (event) => {
    // Iterate through keys
    for (let key in event.records) {
      console.log('Key: ', key)
      // Iterate through records
      event.records[key].map((record) => {
        console.log('Record: ', record)
        // Decode base64
        const msg = Buffer.from(record.value, 'base64').toString()
        console.log('Message:', msg)
      })
    }
}
```

搭配 Lambda 使用 TypeScript 來取用 Amazon MSK 事件。

```
import { MSKEvent, Context } from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "msk-handler-sample",
});

export const handler = async (
```

```
  event: MSKEvent,
  context: Context
): Promise<void> => {
  for (const [topic, topicRecords] of Object.entries(event.records)) {
    logger.info(`Processing key: ${topic}`);

    // Process each record in the partition
    for (const record of topicRecords) {
      try {
        // Decode the message value from base64
        const decodedMessage = Buffer.from(record.value, 'base64').toString();

        logger.info({
          message: decodedMessage
        });
      }
      catch (error) {
        logger.error('Error processing event', { error });
        throw error;
      }
    };
  }
}
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來取用 Amazon MSK 事件。

```
<?php
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

// using bref/bref and bref/logger for simplicity
```

```
use Bref\Context\Context;
use Bref\Event\Kafka\KafkaEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): void
    {
        $kafkaEvent = new KafkaEvent($event);
        $this->logger->info("Processing records");
        $records = $kafkaEvent->getRecords();

        foreach ($records as $record) {
            try {
                $key = $record->getKey();
                $this->logger->info("Key: $key");

                $values = $record->getValue();
                $this->logger->info(json_encode($values));

                foreach ($values as $value) {
                    $this->logger->info("Value: $value");
                }

            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");
```

```
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來取用 Amazon MSK 事件。

```
import base64

def lambda_handler(event, context):
    # Iterate through keys
    for key in event['records']:
        print('Key:', key)
        # Iterate through records
        for record in event['records'][key]:
            print('Record:', record)
            # Decode base64
            msg = base64.b64decode(record['value']).decode('utf-8')
            print('Message:', msg)
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來取用 Amazon MSK 事件。

```
require 'base64'

def lambda_handler(event:, context:)
  # Iterate through keys
  event['records'].each do |key, records|
    puts "Key: #{key}"

    # Iterate through records
    records.each do |record|
      puts "Record: #{record}"

      # Decode base64
      msg = Base64.decode64(record['value'])
      puts "Message: #{msg}"
    end
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 取用 Amazon MSK 事件。

```
use aws_lambda_events::event::kafka::KafkaEvent;
use lambda_runtime::{run, service_fn, tracing, Error, LambdaEvent};
use base64::prelude::*;
use serde_json::{Value};
use tracing::{info};

/// Pre-Requisites:
/// 1. Install Cargo Lambda - see https://www.cargo-lambda.info/guide/getting-
started.html
/// 2. Add packages tracing, tracing-subscriber, serde_json, base64
///
/// This is the main body for the function.
/// Write your code inside it.
/// There are some code example in the following URLs:
/// - https://github.com/awslabs/aws-lambda-rust-runtime/tree/main/examples
/// - https://github.com/aws-samples/serverless-rust-demo/

async fn function_handler(event: LambdaEvent<KafkaEvent>) -> Result<Value, Error>
 {

    let payload = event.payload.records;

    for (_name, records) in payload.iter() {

        for record in records {

         let record_text = record.value.as_ref().ok_or("Value is None")?;
         info!("Record: {}", &record_text);

         // perform Base64 decoding
         let record_bytes = BASE64_STANDARD.decode(record_text)?;
         let message = std::str::from_utf8(&record_bytes)?;

         info!("Message: {}", message);
        }

    }

    Ok(().into())
}

#[tokio::main]
```

```
async fn main() -> Result<(), Error> {

    // required to enable CloudWatch error logging by the runtime
    tracing::init_default_subscriber();
    info!("Setup CW subscriber!");

    run(service_fn(function_handler)).await
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Amazon S3 觸發條件調用 Lambda 函數

下列程式碼範例顯示如何實作 Lambda 函數來接收上傳物件至 S3 儲存貯體時觸發的事件。此函數會從事件參數擷取 S3 儲存貯體名稱和物件金鑰，並呼叫 Amazon S3 API 以擷取和記錄物件的內容類型。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Threading.Tasks;
using Amazon.Lambda.Core;
using Amazon.S3;
using System;
using Amazon.Lambda.S3Events;
using System.Web;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace S3Integration
{
    public class Function
    {
        private static AmazonS3Client _s3Client;
        public Function() : this(null)
        {
        }

        internal Function(AmazonS3Client s3Client)
        {
            _s3Client = s3Client ?? new AmazonS3Client();
        }

        public async Task<string> Handler(S3Event evt, ILambdaContext context)
        {
            try
            {
                if (evt.Records.Count <= 0)
                {
                    context.Logger.LogLine("Empty S3 Event received");
                    return string.Empty;
                }

                var bucket = evt.Records[0].S3.Bucket.Name;
                var key = HttpUtility.UrlDecode(evt.Records[0].S3.Object.Key);

                context.Logger.LogLine($"Request is for {bucket} and {key}");

                var objectResult = await _s3Client.GetObjectAsync(bucket, key);

                context.Logger.LogLine($"Returning {objectResult.Key}");

                return objectResult.Key;
            }
            catch (Exception e)
            {
                context.Logger.LogLine($"Error processing request -
 {e.Message}");

                return string.Empty;
```

```
            }
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "log"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/s3"
)

func handler(ctx context.Context, s3Event events.S3Event) error {
 sdkConfig, err := config.LoadDefaultConfig(ctx)
 if err != nil {
  log.Printf("failed to load default config: %s", err)
  return err
 }
 s3Client := s3.NewFromConfig(sdkConfig)

 for _, record := range s3Event.Records {
  bucket := record.S3.Bucket.Name
  key := record.S3.Object.URLDecodedKey
```

```
  headOutput, err := s3Client.HeadObject(ctx, &s3.HeadObjectInput{
   Bucket: &bucket,
   Key:    &key,
  })
  if err != nil {
   log.Printf("error getting head of object %s/%s: %s", bucket, key, err)
   return err
  }
  log.Printf("successfully retrieved %s/%s of type %s", bucket, key,
 *headOutput.ContentType)
 }

 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import software.amazon.awssdk.services.s3.model.HeadObjectRequest;
import software.amazon.awssdk.services.s3.model.HeadObjectResponse;
import software.amazon.awssdk.services.s3.S3Client;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
```

```
import com.amazonaws.services.lambda.runtime.events.S3Event;
import
 com.amazonaws.services.lambda.runtime.events.models.s3.S3EventNotification.S3EventNotifi

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Handler implements RequestHandler<S3Event, String> {
    private static final Logger logger = LoggerFactory.getLogger(Handler.class);
    @Override
    public String handleRequest(S3Event s3event, Context context) {
        try {
          S3EventNotificationRecord record = s3event.getRecords().get(0);
          String srcBucket = record.getS3().getBucket().getName();
          String srcKey = record.getS3().getObject().getUrlDecodedKey();

          S3Client s3Client = S3Client.builder().build();
          HeadObjectResponse headObject = getHeadObject(s3Client, srcBucket,
 srcKey);

          logger.info("Successfully retrieved " + srcBucket + "/" + srcKey + " of
 type " + headObject.contentType());

          return "Ok";
        } catch (Exception e) {
          throw new RuntimeException(e);
        }
    }

    private HeadObjectResponse getHeadObject(S3Client s3Client, String bucket,
 String key) {
        HeadObjectRequest headObjectRequest = HeadObjectRequest.builder()
                .bucket(bucket)
                .key(key)
                .build();
        return s3Client.headObject(headObjectRequest);
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 來使用 S3 事件。

```
import { S3Client, HeadObjectCommand } from "@aws-sdk/client-s3";

const client = new S3Client();

export const handler = async (event, context) => {

    // Get the object from the event and show its content type
    const bucket = event.Records[0].s3.bucket.name;
    const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g,
 ' '));

    try {
        const { ContentType } = await client.send(new HeadObjectCommand({
            Bucket: bucket,
            Key: key,
        }));

        console.log('CONTENT TYPE:', ContentType);
        return ContentType;

    } catch (err) {
        console.log(err);
        const message = `Error getting object ${key} from bucket ${bucket}. Make
 sure they exist and your bucket is in the same region as this function.`;
        console.log(message);
        throw new Error(message);
    }
};
```

使用 TypeScript 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { S3Event } from 'aws-lambda';
import { S3Client, HeadObjectCommand } from '@aws-sdk/client-s3';

const s3 = new S3Client({ region: process.env.AWS_REGION });

export const handler = async (event: S3Event): Promise<string | undefined> => {
  // Get the object from the event and show its content type
  const bucket = event.Records[0].s3.bucket.name;
  const key = decodeURIComponent(event.Records[0].s3.object.key.replace(/\+/g, '
 '));
  const params = {
    Bucket: bucket,
    Key: key,
  };
  try {
    const { ContentType } = await s3.send(new HeadObjectCommand(params));
    console.log('CONTENT TYPE:', ContentType);
    return ContentType;
  } catch (err) {
    console.log(err);
    const message = `Error getting object ${key} from bucket ${bucket}. Make sure
 they exist and your bucket is in the same region as this function.`;
    console.log(message);
    throw new Error(message);
  }
};
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來使用 S3 事件。

```
<?php

use Bref\Context\Context;
use Bref\Event\S3\S3Event;
use Bref\Event\S3\S3Handler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';


class Handler extends S3Handler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    public function handleS3(S3Event $event, Context $context) : void
    {
        $this->logger->info("Processing S3 records");

        // Get the object from the event and show its content type
        $records = $event->getRecords();

        foreach ($records as $record)
        {
            $bucket = $record->getBucket()->getName();
            $key = urldecode($record->getObject()->getKey());

            try {
                $fileSize = urldecode($record->getObject()->getSize());
                echo "File Size: " . $fileSize . "\n";
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                echo $e->getMessage() . "\n";
                echo 'Error getting object ' . $key . ' from bucket ' .
 $bucket . '. Make sure they exist and your bucket is in the same region as this
 function.' . "\n";
                throw $e;
            }
        }
```

```
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 S3 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
import json
import urllib.parse
import boto3

print('Loading function')

s3 = boto3.client('s3')


def lambda_handler(event, context):
    #print("Received event: " + json.dumps(event, indent=2))

    # Get the object from the event and show its content type
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'],
 encoding='utf-8')
    try:
        response = s3.get_object(Bucket=bucket, Key=key)
        print("CONTENT TYPE: " + response['ContentType'])
        return response['ContentType']
    except Exception as e:
```

```
        print(e)
        print('Error getting object {} from bucket {}. Make sure they exist and
 your bucket is in the same region as this function.'.format(key, bucket))
        raise e
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來使用 S3 事件。

```
require 'json'
require 'uri'
require 'aws-sdk'

puts 'Loading function'

def lambda_handler(event:, context:)
  s3 = Aws::S3::Client.new(region: 'region') # Your AWS region
  # puts "Received event: #{JSON.dump(event)}"

  # Get the object from the event and show its content type
  bucket = event['Records'][0]['s3']['bucket']['name']
  key = URI.decode_www_form_component(event['Records'][0]['s3']['object']['key'],
 Encoding::UTF_8)
  begin
    response = s3.get_object(bucket: bucket, key: key)
    puts "CONTENT TYPE: #{response.content_type}"
    return response.content_type
  rescue StandardError => e
    puts e.message
    puts "Error getting object #{key} from bucket #{bucket}. Make sure they exist
 and your bucket is in the same region as this function."
    raise e
  end
```

```
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來使用 S3 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::s3::S3Event;
use aws_sdk_s3::{Client};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};


/// Main function
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    // Initialize the AWS SDK for Rust
    let config = aws_config::load_from_env().await;
    let s3_client = Client::new(&config);

    let res = run(service_fn(|request: LambdaEvent<S3Event>| {
        function_handler(&s3_client, request)
    })).await;

    res
}
```

```
async fn function_handler(
    s3_client: &Client,
    evt: LambdaEvent<S3Event>
) -> Result<(), Error> {
    tracing::info!(records = ?evt.payload.records.len(), "Received request from
 SQS");

    if evt.payload.records.len() == 0 {
        tracing::info!("Empty S3 event received");
    }

    let bucket = evt.payload.records[0].s3.bucket.name.as_ref().expect("Bucket
 name to exist");
    let key = evt.payload.records[0].s3.object.key.as_ref().expect("Object key to
 exist");

    tracing::info!("Request is for {} and object {}", bucket, key);

    let s3_get_object_result = s3_client
        .get_object()
        .bucket(bucket)
        .key(key)
        .send()
        .await;

    match s3_get_object_result {
        Ok(_) => tracing::info!("S3 Get Object success, the s3GetObjectResult
 contains a 'body' property of type ByteStream"),
        Err(_) => tracing::info!("Failure with S3 Get Object request")
    }

    Ok(())
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Amazon SNS 觸發條件調用 Lambda 函數

下列程式碼範例顯示如何實作 Lambda 函數，以便接收在收到來自 SNS 主題的訊息時觸發的事件。函數會從事件參數擷取訊息，並記錄每一則訊息的內容。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SNSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SnsIntegration;

public class Function
{
    public async Task FunctionHandler(SNSEvent evnt, ILambdaContext context)
    {
        foreach (var record in evnt.Records)
        {
            await ProcessRecordAsync(record, context);
        }
        context.Logger.LogInformation("done");
    }

    private async Task ProcessRecordAsync(SNSEvent.SNSRecord record,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed record
 {record.Sns.Message}");
```

```
            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, snsEvent events.SNSEvent) {
 for _, record := range snsEvent.Records {
  processMessage(record)
 }
```

```
 fmt.Println("done")
}

func processMessage(record events.SNSEventRecord) {
 message := record.SNS.Message
 fmt.Printf("Processed message: %s\n", message)
 // TODO: Process your record here
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package example;

import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.LambdaLogger;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SNSEvent;
import com.amazonaws.services.lambda.runtime.events.SNSEvent.SNSRecord;


import java.util.Iterator;
import java.util.List;

public class SNSEventHandler implements RequestHandler<SNSEvent, Boolean> {
    LambdaLogger logger;
```

```
    @Override
    public Boolean handleRequest(SNSEvent event, Context context) {
        logger = context.getLogger();
        List<SNSRecord> records = event.getRecords();
        if (!records.isEmpty()) {
            Iterator<SNSRecord> recordsIter = records.iterator();
            while (recordsIter.hasNext()) {
                processRecord(recordsIter.next());
            }
        }
        return Boolean.TRUE;
    }

    public void processRecord(SNSRecord record) {
        try {
            String message = record.getSNS().getMessage();
            logger.log("message: " + message);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
```

```
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record) {
  try {
    const message = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

使用 TypeScript 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SNSEvent, Context, SNSHandler, SNSEventRecord } from "aws-lambda";

export const functionHandler: SNSHandler = async (
  event: SNSEvent,
  context: Context
): Promise<void> => {
  for (const record of event.Records) {
    await processMessageAsync(record);
  }
  console.info("done");
};

async function processMessageAsync(record: SNSEventRecord): Promise<any> {
  try {
    const message: string = JSON.stringify(record.Sns.Message);
    console.log(`Processed message ${message}`);
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
```

```
    throw err;
  }
}
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

/*
Since native PHP support for AWS Lambda is not available, we are utilizing Bref's
 PHP functions runtime for AWS Lambda.
For more information on Bref's PHP runtime for Lambda, refer to: https://bref.sh/
docs/runtimes/function

Another approach would be to create a custom runtime.
A practical example can be found here: https://aws.amazon.com/blogs/apn/aws-
lambda-custom-runtime-for-php-a-practical-example/
*/

// Additional composer packages may be required when using Bref or any other PHP
 functions runtime.
// require __DIR__ . '/vendor/autoload.php';

use Bref\Context\Context;
use Bref\Event\Sns\SnsEvent;
use Bref\Event\Sns\SnsHandler;

class Handler extends SnsHandler
{
```

```
    public function handleSns(SnsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $message = $record->getMessage();

            // TODO: Implement your custom processing logic here
            // Any exception thrown will be logged and the invocation will be
 marked as failed

            echo "Processed Message: $message" . PHP_EOL;
        }
    }
}

return new Handler();
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 SNS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for record in event['Records']:
        process_message(record)
    print("done")

def process_message(record):
    try:
        message = record['Sns']['Message']
        print(f"Processed message {message}")
        # TODO; Process your record here
```

```
    except Exception as e:
        print("An error occurred")
        raise e
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來使用 SNS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].map { |record| process_message(record) }
end

def process_message(record)
  message = record['Sns']['Message']
  puts("Processing message: #{message}")
rescue StandardError => e
  puts("Error processing message: #{e}")
  raise
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來使用 SNS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sns::SnsEvent;
use aws_lambda_events::sns::SnsRecord;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};
use tracing::info;

// Built with the following dependencies:
//  aws_lambda_events = { version = "0.10.0", default-features = false, features
 = ["sns"] }
//  lambda_runtime = "0.8.1"
//  tokio = { version = "1", features = ["macros"] }
//  tracing = { version = "0.1", features = ["log"] }
//  tracing-subscriber = { version = "0.3", default-features = false, features =
 ["fmt"] }

async fn function_handler(event: LambdaEvent<SnsEvent>) -> Result<(), Error> {
    for event in event.payload.records {
        process_record(&event)?;
    }

    Ok(())
}

fn process_record(record: &SnsRecord) -> Result<(), Error> {
    info!("Processing SNS Message: {}", record.sns.message);

    // Implement your record handling code here.

    Ok(())
}
```

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        .with_target(false)
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Amazon SQS 觸發條件調用 Lambda 函數

下列程式碼範例示範如何實作 Lambda 函數，以便接收在收到來自 SQS 佇列的訊息時觸發的事件。函數會從事件參數擷取訊息，並記錄每一則訊息的內容。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;


// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace SqsIntegrationSampleCode
{
    public async Task FunctionHandler(SQSEvent evnt, ILambdaContext context)
    {
        foreach (var message in evnt.Records)
        {
            await ProcessMessageAsync(message, context);
        }

        context.Logger.LogInformation("done");
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        try
        {
            context.Logger.LogInformation($"Processed message {message.Body}");

            // TODO: Do interesting work based on the new message
            await Task.CompletedTask;
        }
        catch (Exception e)
        {
            //You can use Dead Letter Queue to handle failures. By configuring a
 Lambda DLQ.
            context.Logger.LogError($"An error occurred");
            throw;
        }

    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package integration_sqs_to_lambda

import (
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(event events.SQSEvent) error {
 for _, record := range event.Records {
  err := processMessage(record)
  if err != nil {
   return err
  }
 }
 fmt.Println("done")
 return nil
}

func processMessage(record events.SQSMessage) error {
 fmt.Printf("Processed message %s\n", record.Body)
 // TODO: Do interesting work based on the new message
 return nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSEvent.SQSMessage;

public class Function implements RequestHandler<SQSEvent, Void> {
    @Override
    public Void handleRequest(SQSEvent sqsEvent, Context context) {
        for (SQSMessage msg : sqsEvent.getRecords()) {
            processMessage(msg, context);
        }
        context.getLogger().log("done");
        return null;
    }

    private void processMessage(SQSMessage msg, Context context) {
        try {
            context.getLogger().log("Processed message " + msg.getBody());

            // TODO: Do interesting work based on the new message

        } catch (Exception e) {
            context.getLogger().log("An error occurred");
            throw e;
        }
```

```
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message) {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

使用 TypeScript 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, Context, SQSHandler, SQSRecord } from "aws-lambda";
```

```
export const functionHandler: SQSHandler = async (
  event: SQSEvent,
  context: Context
): Promise<void> => {
  for (const message of event.Records) {
    await processMessageAsync(message);
  }
  console.info("done");
};

async function processMessageAsync(message: SQSRecord): Promise<any> {
  try {
    console.log(`Processed message ${message.body}`);
    // TODO: Do interesting work based on the new message
    await Promise.resolve(1); //Placeholder for actual async work
  } catch (err) {
    console.error("An error occurred");
    throw err;
  }
}
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
```

```
use Bref\Event\InvalidLambdaEvent;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        foreach ($event->getRecords() as $record) {
            $body = $record->getBody();
            // TODO: Do interesting work based on the new message
        }
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 來使用 SQS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event, context):
    for message in event['Records']:
        process_message(message)
    print("done")

def process_message(message):
    try:
        print(f"Processed message {message['body']}")
        # TODO: Do interesting work based on the new message
    except Exception as err:
        print("An error occurred")
        raise err
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來使用 SQS 事件。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def lambda_handler(event:, context:)
  event['Records'].each do |message|
    process_message(message)
  end
  puts "done"
end

def process_message(message)
  begin
    puts "Processed message #{message['body']}"
    # TODO: Do interesting work based on the new message
```

```
  rescue StandardError => err
    puts "An error occurred"
    raise err
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來使用 SQS 事件。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::event::sqs::SqsEvent;
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<SqsEvent>) -> Result<(), Error> {
    event.payload.records.iter().for_each(|record| {
        // process the record
        tracing::info!("Message body: {}",
 record.body.as_deref().unwrap_or_default())
    });

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
```

```
        .init();

    run(service_fn(function_handler)).await
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Kinesis 觸發條件報告 Lambda 函數的批次項目失敗

下列程式碼範例示範如何針對接收來自 Kinesis 串流之事件的 Lambda 函數，實作部分批次回應。此函數會在回應中報告批次項目失敗，指示 Lambda 稍後重試這些訊息。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text;
using System.Text.Json.Serialization;
using Amazon.Lambda.Core;
using Amazon.Lambda.KinesisEvents;
using AWS.Lambda.Powertools.Logging;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace KinesisIntegration;

public class Function
```

```
{
    // Powertools Logger requires an environment variables against your function
    // POWERTOOLS_SERVICE_NAME
    [Logging(LogEvent = true)]
    public async Task<StreamsEventResponse> FunctionHandler(KinesisEvent evnt,
 ILambdaContext context)
    {
        if (evnt.Records.Count == 0)
        {
            Logger.LogInformation("Empty Kinesis Event received");
            return new StreamsEventResponse();
        }

        foreach (var record in evnt.Records)
        {
            try
            {
                Logger.LogInformation($"Processed Event with EventId:
 {record.EventId}");
                string data = await GetRecordDataAsync(record.Kinesis, context);
                Logger.LogInformation($"Data: {data}");
                // TODO: Do interesting work based on the new data
            }
            catch (Exception ex)
            {
                Logger.LogError($"An error occurred {ex.Message}");
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                return new StreamsEventResponse
                {
                    BatchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>
                    {
                        new StreamsEventResponse.BatchItemFailure
 { ItemIdentifier = record.Kinesis.SequenceNumber }
                    }
                };
            }
        }
        Logger.LogInformation($"Successfully processed {evnt.Records.Count}
 records.");
        return new StreamsEventResponse();
```

```
    }

    private async Task<string> GetRecordDataAsync(KinesisEvent.Record record,
 ILambdaContext context)
    {
        byte[] bytes = record.Data.ToArray();
        string data = Encoding.UTF8.GetString(bytes);
        await Task.CompletedTask; //Placeholder for actual async work
        return data;
    }
}

public class StreamsEventResponse
{
    [JsonPropertyName("batchItemFailures")]
    public IList<BatchItemFailure> BatchItemFailures { get; set; }
    public class BatchItemFailure
    {
        [JsonPropertyName("itemIdentifier")]
        public string ItemIdentifier { get; set; }
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 來報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
```

```
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, kinesisEvent events.KinesisEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, record := range kinesisEvent.Records {
  curRecordSequenceNumber := ""

  // Process your record
  if /* Your record processing condition here */ {
   curRecordSequenceNumber = record.Kinesis.SequenceNumber
  }

  // Add a condition to check if the record processing failed
  if curRecordSequenceNumber != "" {
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": curRecordSequenceNumber})
  }
 }

 kinesisBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return kinesisBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

透過使用 Java 的 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.KinesisEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

public class ProcessKinesisRecords implements RequestHandler<KinesisEvent,
 StreamsEventResponse> {

    @Override
    public StreamsEventResponse handleRequest(KinesisEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (KinesisEvent.KinesisEventRecord kinesisEventRecord :
 input.getRecords()) {
            try {
                //Process your record
                KinesisEvent.Record kinesisRecord =
 kinesisEventRecord.getKinesis();
                curRecordSequenceNumber = kinesisRecord.getSequenceNumber();

            } catch (Exception e) {
```

```
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse(batchItemFailures);
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Javascript 搭配 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
exports.handler = async (event, context) => {
  for (const record of event.Records) {
    try {
      console.log(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
      console.log(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      console.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
```

```
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  console.log(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(payload) {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

使用 TypeScript 搭配 Lambda 報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import {
  KinesisStreamEvent,
  Context,
  KinesisStreamHandler,
  KinesisStreamRecordPayload,
  KinesisStreamBatchResponse,
} from "aws-lambda";
import { Buffer } from "buffer";
import { Logger } from "@aws-lambda-powertools/logger";

const logger = new Logger({
  logLevel: "INFO",
  serviceName: "kinesis-stream-handler-sample",
});

export const functionHandler: KinesisStreamHandler = async (
  event: KinesisStreamEvent,
  context: Context
): Promise<KinesisStreamBatchResponse> => {
  for (const record of event.Records) {
    try {
      logger.info(`Processed Kinesis Event - EventID: ${record.eventID}`);
      const recordData = await getRecordDataAsync(record.kinesis);
```

```
      logger.info(`Record Data: ${recordData}`);
      // TODO: Do interesting work based on the new data
    } catch (err) {
      logger.error(`An error occurred ${err}`);
      /* Since we are working with streams, we can return the failed item
 immediately.
           Lambda will immediately begin to retry processing from this failed
 item onwards. */
      return {
        batchItemFailures: [{ itemIdentifier: record.kinesis.sequenceNumber }],
      };
    }
  }
  logger.info(`Successfully processed ${event.Records.length} records.`);
  return { batchItemFailures: [] };
};

async function getRecordDataAsync(
  payload: KinesisStreamRecordPayload
): Promise<string> {
  var data = Buffer.from(payload.data, "base64").toString("utf-8");
  await Promise.resolve(1); //Placeholder for actual async work
  return data;
}
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 來報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php
```

```
# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\Kinesis\KinesisEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $kinesisEvent = new KinesisEvent($event);
        $this->logger->info("Processing records");
        $records = $kinesisEvent->getRecords();

        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
```

```
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

透過使用 Python 的 Lambda 報告 Kinesis 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["kinesis"]["sequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 來報告 Kinesis 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'aws-sdk'

def lambda_handler(event:, context:)
  batch_item_failures = []

  event['Records'].each do |record|
    begin
      puts "Processed Kinesis Event - EventID: #{record['eventID']}"
      record_data = get_record_data_async(record['kinesis'])
      puts "Record Data: #{record_data}"
      # TODO: Do interesting work based on the new data
    rescue StandardError => err
      puts "An error occurred #{err}"
      # Since we are working with streams, we can return the failed item
 immediately.
      # Lambda will immediately begin to retry processing from this failed item
 onwards.
      return { batchItemFailures: [{ itemIdentifier: record['kinesis']
['sequenceNumber'] }] }
    end
  end

  puts "Successfully processed #{event['Records'].length} records."
  { batchItemFailures: batch_item_failures }
end
```

```
def get_record_data_async(payload)
  data = Base64.decode64(payload['data']).force_encoding('utf-8')
  # Placeholder for actual async work
  sleep(1)
  data
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 來報告 Kinesis 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::kinesis::KinesisEvent,
    kinesis::KinesisEventRecord,
    streams::{KinesisBatchItemFailure, KinesisEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn function_handler(event: LambdaEvent<KinesisEvent>) ->
 Result<KinesisEventResponse, Error> {
    let mut response = KinesisEventResponse {
        batch_item_failures: vec![],
    };

    if event.payload.records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in &event.payload.records {
        tracing::info!(
            "EventId: {}",
```

```
            record.event_id.as_deref().unwrap_or_default()
        );

        let record_processing_result = process_record(record);

        if record_processing_result.is_err() {
            response.batch_item_failures.push(KinesisBatchItemFailure {
                item_identifier: record.kinesis.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }

    tracing::info!(
        "Successfully processed {} records",
        event.payload.records.len()
    );

    Ok(response)
}

fn process_record(record: &KinesisEventRecord) -> Result<(), Error> {
    let record_data = std::str::from_utf8(record.kinesis.data.as_slice());

    if let Some(err) = record_data.err() {
        tracing::error!("Error: {}", err);
        return Err(Error::from(err));
    }

    let record_data = record_data.unwrap_or_default();

    // do something interesting with the data
    tracing::info!("Data: {}", record_data);

    Ok(())
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
```

```
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 DynamoDB 觸發條件報告 Lambda 函式的批次項目失敗

下列程式碼範例示範如何針對接收來自 DynamoDB 串流之事件的 Lambda 函式，實作部分批次回應。此函數會在回應中報告批次項目失敗，指示 Lambda 稍後重試這些訊息。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using System.Text.Json;
using System.Text;
using Amazon.Lambda.Core;
using Amazon.Lambda.DynamoDBEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
```

```
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri

namespace AWSLambda_DDB;

public class Function
{
    public StreamsEventResponse FunctionHandler(DynamoDBEvent dynamoEvent,
 ILambdaContext context)

    {
        context.Logger.LogInformation($"Beginning to process
 {dynamoEvent.Records.Count} records...");
        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 List<StreamsEventResponse.BatchItemFailure>();
        StreamsEventResponse streamsEventResponse = new StreamsEventResponse();

        foreach (var record in dynamoEvent.Records)
        {
            try
            {
                var sequenceNumber = record.Dynamodb.SequenceNumber;
                context.Logger.LogInformation(sequenceNumber);
            }
            catch (Exception ex)
            {
                context.Logger.LogError(ex.Message);
                batchItemFailures.Add(new StreamsEventResponse.BatchItemFailure()
 { ItemIdentifier = record.Dynamodb.SequenceNumber });
            }
        }

        if (batchItemFailures.Count > 0)
        {
            streamsEventResponse.BatchItemFailures = batchItemFailures;
        }

        context.Logger.LogInformation("Stream processing complete.");
        return streamsEventResponse;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

type BatchItemFailure struct {
 ItemIdentifier string `json:"ItemIdentifier"`
}

type BatchResult struct {
 BatchItemFailures []BatchItemFailure `json:"BatchItemFailures"`
}

func HandleRequest(ctx context.Context, event events.DynamoDBEvent)
 (*BatchResult, error) {
 var batchItemFailures []BatchItemFailure
 curRecordSequenceNumber := ""

 for _, record := range event.Records {
  // Process your record
  curRecordSequenceNumber = record.Change.SequenceNumber
 }

 if curRecordSequenceNumber != "" {
  batchItemFailures = append(batchItemFailures, BatchItemFailure{ItemIdentifier:
 curRecordSequenceNumber})
```

```
 }

 batchResult := BatchResult{
  BatchItemFailures: batchItemFailures,
 }

 return &batchResult, nil
}

func main() {
 lambda.Start(HandleRequest)
}
```

Java

適用於 Java 2.x 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.DynamodbEvent;
import com.amazonaws.services.lambda.runtime.events.StreamsEventResponse;
import com.amazonaws.services.lambda.runtime.events.models.dynamodb.StreamRecord;

import java.util.ArrayList;
import java.util.List;

public class ProcessDynamodbRecords implements RequestHandler<DynamodbEvent,
 StreamsEventResponse> {

    @Override
```

```
    public StreamsEventResponse handleRequest(DynamodbEvent input, Context
 context) {

        List<StreamsEventResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<>();
        String curRecordSequenceNumber = "";

        for (DynamodbEvent.DynamodbStreamRecord dynamodbStreamRecord :
 input.getRecords()) {
          try {
                //Process your record
                StreamRecord dynamodbRecord = dynamodbStreamRecord.getDynamodb();
                curRecordSequenceNumber = dynamodbRecord.getSequenceNumber();

            } catch (Exception e) {
                /* Since we are working with streams, we can return the failed
 item immediately.
                   Lambda will immediately begin to retry processing from this
 failed item onwards. */
                batchItemFailures.add(new
 StreamsEventResponse.BatchItemFailure(curRecordSequenceNumber));
                return new StreamsEventResponse(batchItemFailures);
            }
        }

       return new StreamsEventResponse();
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
export const handler = async (event) => {
  const records = event.Records;
  let curRecordSequenceNumber = "";

  for (const record of records) {
    try {
      // Process your record
      curRecordSequenceNumber = record.dynamodb.SequenceNumber;
    } catch (e) {
      // Return failed record's sequence number
      return { batchItemFailures: [{ itemIdentifier:
 curRecordSequenceNumber }] };
    }
  }

  return { batchItemFailures: [] };
};
```

使用 TypeScript 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
import {
  DynamoDBBatchResponse,
  DynamoDBBatchItemFailure,
  DynamoDBStreamEvent,
} from "aws-lambda";

export const handler = async (
  event: DynamoDBStreamEvent
): Promise<DynamoDBBatchResponse> => {
  const batchItemFailures: DynamoDBBatchItemFailure[] = [];
  let curRecordSequenceNumber;

  for (const record of event.Records) {
    curRecordSequenceNumber = record.dynamodb?.SequenceNumber;

    if (curRecordSequenceNumber) {
      batchItemFailures.push({
        itemIdentifier: curRecordSequenceNumber,
      });
    }
  }
```

```
  return { batchItemFailures: batchItemFailures };
};
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
<?php

# using bref/bref and bref/logger for simplicity

use Bref\Context\Context;
use Bref\Event\DynamoDb\DynamoDbEvent;
use Bref\Event\Handler as StdHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler implements StdHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handle(mixed $event, Context $context): array
    {
        $dynamoDbEvent = new DynamoDbEvent($event);
```

```
        $this->logger->info("Processing records");

        $records = $dynamoDbEvent->getRecords();
        $failedRecords = [];
        foreach ($records as $record) {
            try {
                $data = $record->getData();
                $this->logger->info(json_encode($data));
                // TODO: Do interesting work based on the new data
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $failedRecords[] = $record->getSequenceNumber();
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords records");

        // change format for the response
        $failures = array_map(
            fn(string $sequenceNumber) => ['itemIdentifier' => $sequenceNumber],
            $failedRecords
        );

        return [
            'batchItemFailures' => $failures
        ];
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python (Boto3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
def handler(event, context):
    records = event.get("Records")
    curRecordSequenceNumber = ""

    for record in records:
        try:
            # Process your record
            curRecordSequenceNumber = record["dynamodb"]["SequenceNumber"]
        except Exception as e:
            # Return failed record's sequence number
            return {"batchItemFailures":[{"itemIdentifier":
 curRecordSequenceNumber}]}

    return {"batchItemFailures":[]}
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
def lambda_handler(event:, context:)
    records = event["Records"]
    cur_record_sequence_number = ""

    records.each do |record|
      begin
        # Process your record
        cur_record_sequence_number = record["dynamodb"]["SequenceNumber"]
      rescue StandardError => e
        # Return failed record's sequence number
        return {"batchItemFailures" => [{"itemIdentifier" =>
 cur_record_sequence_number}]}
      end
    end

    {"batchItemFailures" => []}
  end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 報告 DynamoDB 批次項目失敗。

```
use aws_lambda_events::{
    event::dynamodb::{Event, EventRecord, StreamRecord},
    streams::{DynamoDbBatchItemFailure, DynamoDbEventResponse},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

/// Process the stream record
fn process_record(record: &EventRecord) -> Result<(), Error> {
    let stream_record: &StreamRecord = &record.change;
```

```
    // process your stream record here...
    tracing::info!("Data: {:?}", stream_record);

    Ok(())
}

/// Main Lambda handler here...
async fn function_handler(event: LambdaEvent<Event>) ->
 Result<DynamoDbEventResponse, Error> {
    let mut response = DynamoDbEventResponse {
        batch_item_failures: vec![],
    };

    let records = &event.payload.records;

    if records.is_empty() {
        tracing::info!("No records found. Exiting.");
        return Ok(response);
    }

    for record in records {
        tracing::info!("EventId: {}", record.event_id);

        // Couldn't find a sequence number
        if record.change.sequence_number.is_none() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: Some("".to_string()),
            });
            return Ok(response);
        }

        // Process your record here...
        if process_record(record).is_err() {
            response.batch_item_failures.push(DynamoDbBatchItemFailure {
                item_identifier: record.change.sequence_number.clone(),
            });
            /* Since we are working with streams, we can return the failed item
 immediately.
            Lambda will immediately begin to retry processing from this failed
 item onwards. */
            return Ok(response);
        }
    }
```

```
    tracing::info!("Successfully processed {} record(s)", records.len());

    Ok(response)
}

#[tokio::main]
async fn main() -> Result<(), Error> {
    tracing_subscriber::fmt()
        .with_max_level(tracing::Level::INFO)
        // disable printing the name of the module in every log line.
        .with_target(false)
        // disabling time is handy because CloudWatch will add the ingestion
 time.
        .without_time()
        .init();

    run(service_fn(function_handler)).await
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

使用 Amazon SQS 觸發條件報告 Lambda 函數的批次項目失敗

下列程式碼範例示範如何針對接收來自 SQS 佇列之事件的 Lambda 函數，實作部分批次回應。此函數會在回應中報告批次項目失敗，指示 Lambda 稍後重試這些訊息。

.NET

適用於 .NET 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 .NET 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
using Amazon.Lambda.Core;
using Amazon.Lambda.SQSEvents;

// Assembly attribute to enable the Lambda function's JSON input to be converted
 into a .NET class.
[assembly:
 LambdaSerializer(typeof(Amazon.Lambda.Serialization.SystemTextJson.DefaultLambdaJsonSeri
namespace sqsSample;

public class Function
{
    public async Task<SQSBatchResponse> FunctionHandler(SQSEvent evnt,
 ILambdaContext context)
    {
        List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 List<SQSBatchResponse.BatchItemFailure>();
        foreach(var message in evnt.Records)
        {
            try
            {
                //process your message
                await ProcessMessageAsync(message, context);
            }
            catch (System.Exception)
            {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.Add(new
 SQSBatchResponse.BatchItemFailure{ItemIdentifier=message.MessageId});
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }

    private async Task ProcessMessageAsync(SQSEvent.SQSMessage message,
 ILambdaContext context)
    {
        if (String.IsNullOrEmpty(message.Body))
        {
            throw new Exception("No Body in SQS Message.");
        }
        context.Logger.LogInformation($"Processed message {message.Body}");
```

```
        // TODO: Do interesting work based on the new message
        await Task.CompletedTask;
    }
}
```

Go

SDK for Go V2

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Go 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
package main

import (
 "context"
 "fmt"
 "github.com/aws/aws-lambda-go/events"
 "github.com/aws/aws-lambda-go/lambda"
)

func handler(ctx context.Context, sqsEvent events.SQSEvent)
 (map[string]interface{}, error) {
 batchItemFailures := []map[string]interface{}{}

 for _, message := range sqsEvent.Records {
  if len(message.Body) > 0 {
   // Your message processing condition here
   fmt.Printf("Successfully processed message: %s\n", message.Body)
  } else {
   // Message processing failed
   fmt.Printf("Failed to process message %s\n", message.MessageId)
   batchItemFailures = append(batchItemFailures, map[string]interface{}
{"itemIdentifier": message.MessageId})
  }
```

```
 }

 sqsBatchResponse := map[string]interface{}{
  "batchItemFailures": batchItemFailures,
 }
 return sqsBatchResponse, nil
}

func main() {
 lambda.Start(handler)
}
```

Java

SDK for Java 2.x

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Java 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import com.amazonaws.services.lambda.runtime.Context;
import com.amazonaws.services.lambda.runtime.RequestHandler;
import com.amazonaws.services.lambda.runtime.events.SQSEvent;
import com.amazonaws.services.lambda.runtime.events.SQSBatchResponse;

import java.util.ArrayList;
import java.util.List;

public class ProcessSQSMessageBatch implements RequestHandler<SQSEvent,
 SQSBatchResponse> {
    @Override
    public SQSBatchResponse handleRequest(SQSEvent sqsEvent, Context context) {
         List<SQSBatchResponse.BatchItemFailure> batchItemFailures = new
 ArrayList<SQSBatchResponse.BatchItemFailure>();
```

```
        for (SQSEvent.SQSMessage message : sqsEvent.getRecords()) {
            try {
                //process your message
            } catch (Exception e) {
                //Add failed message identifier to the batchItemFailures list
                batchItemFailures.add(new
 SQSBatchResponse.BatchItemFailure(message.getMessageId()));
            }
        }
        return new SQSBatchResponse(batchItemFailures);
    }
}
```

JavaScript

適用於 JavaScript (v3) 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 JavaScript 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Node.js 20.x Lambda runtime, AWS SDK for Javascript V3
export const handler = async (event, context) => {
    const batchItemFailures = [];
    for (const record of event.Records) {
        try {
            await processMessageAsync(record, context);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }
    return { batchItemFailures };
};

async function processMessageAsync(record, context) {
    if (record.body && record.body.includes("error")) {
        throw new Error("There is an error in the SQS Message.");
    }
```

```
    console.log(`Processed message: ${record.body}`);
}
```

使用 TypeScript 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
import { SQSEvent, SQSBatchResponse, Context, SQSBatchItemFailure, SQSRecord }
 from 'aws-lambda';

export const handler = async (event: SQSEvent, context: Context):
 Promise<SQSBatchResponse> => {
    const batchItemFailures: SQSBatchItemFailure[] = [];

    for (const record of event.Records) {
        try {
            await processMessageAsync(record);
        } catch (error) {
            batchItemFailures.push({ itemIdentifier: record.messageId });
        }
    }

    return {batchItemFailures: batchItemFailures};
};

async function processMessageAsync(record: SQSRecord): Promise<void> {
    if (record.body && record.body.includes("error")) {
        throw new Error('There is an error in the SQS Message.');
    }
    console.log(`Processed message ${record.body}`);
}
```

PHP

適用於 PHP 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 PHP 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
<?php

use Bref\Context\Context;
use Bref\Event\Sqs\SqsEvent;
use Bref\Event\Sqs\SqsHandler;
use Bref\Logger\StderrLogger;

require __DIR__ . '/vendor/autoload.php';

class Handler extends SqsHandler
{
    private StderrLogger $logger;
    public function __construct(StderrLogger $logger)
    {
        $this->logger = $logger;
    }

    /**
     * @throws JsonException
     * @throws \Bref\Event\InvalidLambdaEvent
     */
    public function handleSqs(SqsEvent $event, Context $context): void
    {
        $this->logger->info("Processing SQS records");
        $records = $event->getRecords();

        foreach ($records as $record) {
            try {
                // Assuming the SQS message is in JSON format
                $message = json_decode($record->getBody(), true);
                $this->logger->info(json_encode($message));
                // TODO: Implement your custom processing logic here
            } catch (Exception $e) {
                $this->logger->error($e->getMessage());
                // failed processing the record
                $this->markAsFailed($record);
            }
        }
        $totalRecords = count($records);
        $this->logger->info("Successfully processed $totalRecords SQS records");
```

```
    }
}

$logger = new StderrLogger();
return new Handler($logger);
```

Python

適用於 Python 的 SDK (Boto3)

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Python 搭配 Lambda 報告 SQS 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0

def lambda_handler(event, context):
    if event:
        batch_item_failures = []
        sqs_batch_response = {}

        for record in event["Records"]:
            try:
                print(f"Processed message: {record['body']}")
            except Exception as e:
                batch_item_failures.append({"itemIdentifier":
 record['messageId']})

        sqs_batch_response["batchItemFailures"] = batch_item_failures
        return sqs_batch_response
```

Ruby

SDK for Ruby

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Ruby 搭配 Lambda 報告 SQS 批次項目失敗。

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
require 'json'

def lambda_handler(event:, context:)
  if event
    batch_item_failures = []
    sqs_batch_response = {}

    event["Records"].each do |record|
      begin
        # process message
      rescue StandardError => e
        batch_item_failures << {"itemIdentifier" => record['messageId']}
      end
    end

    sqs_batch_response["batchItemFailures"] = batch_item_failures
    return sqs_batch_response
  end
end
```

Rust

適用於 Rust 的 SDK

> **Note**
>
> GitHub 上提供更多範例。尋找完整範例，並了解如何在無伺服器範例儲存庫中設定和執行。

使用 Rust 搭配 Lambda 報告 SQS 批次項目失敗。

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0
use aws_lambda_events::{
    event::sqs::{SqsBatchResponse, SqsEvent},
    sqs::{BatchItemFailure, SqsMessage},
};
use lambda_runtime::{run, service_fn, Error, LambdaEvent};

async fn process_record(_: &SqsMessage) -> Result<(), Error> {
    Err(Error::from("Error processing message"))
}

async fn function_handler(event: LambdaEvent<SqsEvent>) ->
 Result<SqsBatchResponse, Error> {
    let mut batch_item_failures = Vec::new();
    for record in event.payload.records {
        match process_record(&record).await {
            Ok(_) => (),
            Err(_) => batch_item_failures.push(BatchItemFailure {
                item_identifier: record.message_id.unwrap(),
            }),
        }
    }

    Ok(SqsBatchResponse {
        batch_item_failures,
    })
}

#[tokio::main]
async fn main() -> Result<(), Error> {
```

```
    run(service_fn(function_handler)).await
}
```

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

AWS Lambda 的社群貢獻

AWS 社群貢獻是由多個團隊所建立和維護的範例 AWS。若要提供意見回饋，請使用連結儲存庫中提供的機制。

範例

- 建置和測試無伺服器應用程式

建置和測試無伺服器應用程式

下列程式碼範例示範如何搭配 Lambda 和 DynamoDB 使用 API Gateway 來建置和測試無伺服器應用程式

.NET

適用於 .NET 的 SDK

示範如何使用 .NET SDK 建置和測試無伺服器應用程式，而該應用程式是由具有 Lambda 和 DynamoDB 的 API Gateway 組成。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda

Go

SDK for Go V2

示範如何使用 Go SDK 建置和測試無伺服器應用程式，而該應用程式是由具有 Lambda 和 DynamoDB 的 API Gateway 組成。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda

Java

適用於 Java 2.x 的 SDK

示範如何使用 Java SDK 建置和測試無伺服器應用程式，而該應用程式是由具有 Lambda 和 DynamoDB 的 API Gateway 組成。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- API Gateway
- DynamoDB
- Lambda

Rust

適用於 Rust 的 SDK

示範如何使用 Rust SDK 建置和測試無伺服器應用程式，而該應用程式是由具有 Lambda 和 DynamoDB 的 API Gateway 組成。

如需完整的原始碼和如何設定及執行的指示，請參閱 GitHub 上的完整範例。

此範例中使用的服務

- API Gateway

- DynamoDB
- Lambda

如需 AWS SDK 開發人員指南和程式碼範例的完整清單，請參閱 搭配 AWS SDK 使用 Lambda。此主題也包含有關入門的資訊和舊版 SDK 的詳細資訊。

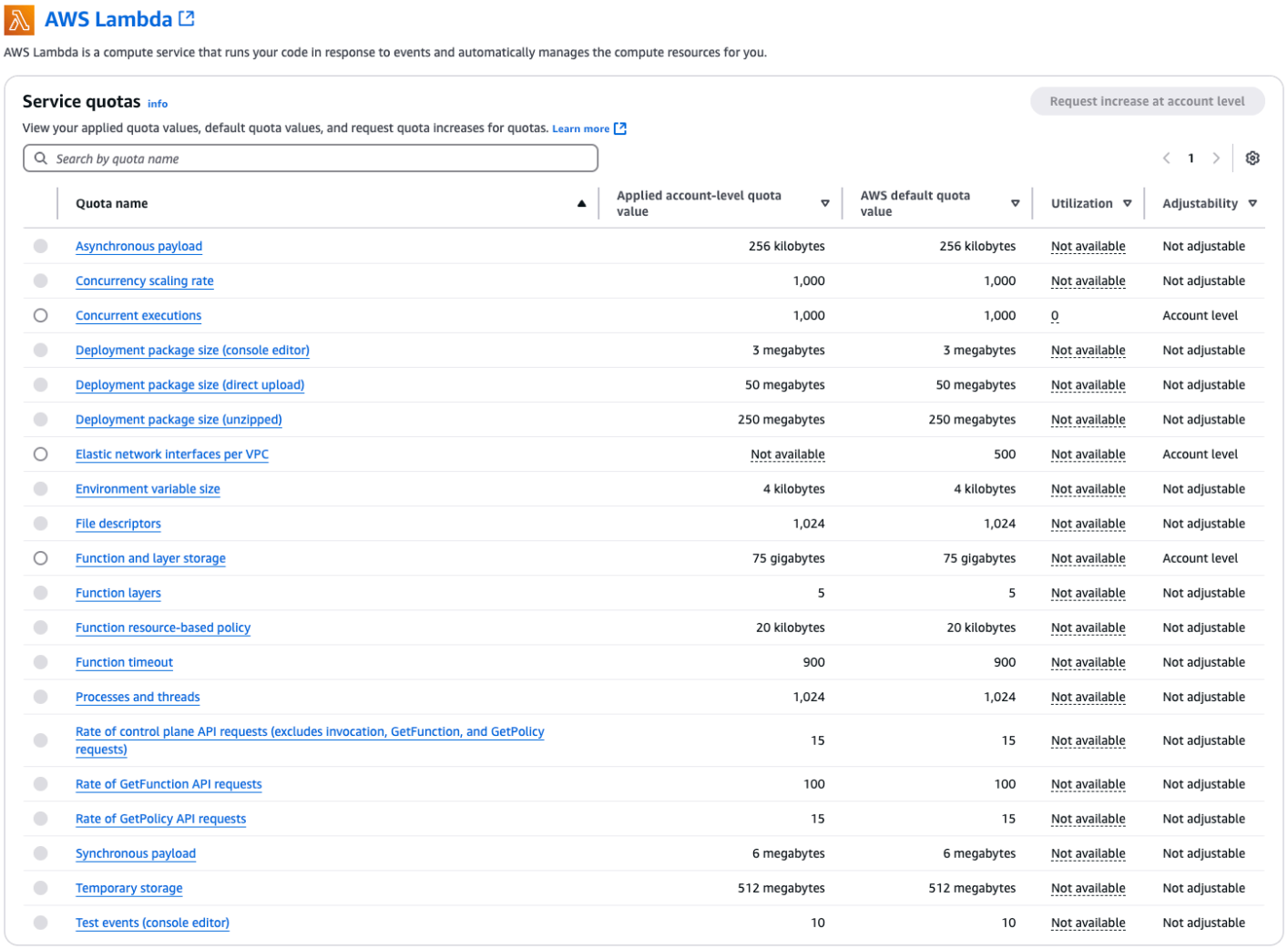
Lambda 配額

> **Important**
>
> 新的 AWS 帳戶 已減少並行和記憶體配額。 會根據您的用量自動 AWS 提高這些配額。

AWS Lambda 旨在快速擴展以滿足需求，讓您的 函數可擴展以為應用程式中的流量提供服務。Lambda 專為在調用之間不保留或不依賴狀態的短效運算任務設計。在單一調用中，程式碼最多可執行 15 分鐘，單一函數最多可使用 10,240 MB 的記憶體。

請務必了解為保護您的帳戶和其他客戶的工作負載而部署的防護機制。服務配額存在於所有 AWS 服務中，包含您無法變更的硬性限制，以及您可以請求增加的軟性限制。根據預設，所有新帳戶都會獲指派允許探索 AWS 服務的配額設定檔。

若要查看適用於您帳戶的配額，請導覽至 Service Quotas 儀表板。您可以在此處檢視服務配、請求提高服務配額，以及檢視目前的使用率。從這裡，您可以向下切入至特定 AWS 服務，例如 Lambda：



AWS Lambda

AWS Lambda is a compute service that runs your code in response to events and automatically manages the compute resources for you.

Service quotas info

Request increase at account level

View your applied quota values, default quota values, and request quota increases for quotas. Learn more

Search by quota name

1

Quota name	Applied account-level quota value	AWS default quota value	Utilization	Adjustability
Asynchronous payload	256 kilobytes	256 kilobytes	Not available	Not adjustable
Concurrency scaling rate	1,000	1,000	Not available	Not adjustable
Concurrent executions	1,000	1,000	0	Account level
Deployment package size (console editor)	3 megabytes	3 megabytes	Not available	Not adjustable
Deployment package size (direct upload)	50 megabytes	50 megabytes	Not available	Not adjustable
Deployment package size (unzipped)	250 megabytes	250 megabytes	Not available	Not adjustable
Elastic network interfaces per VPC	Not available	500	Not available	Account level
Environment variable size	4 kilobytes	4 kilobytes	Not available	Not adjustable
File descriptors	1,024	1,024	Not available	Not adjustable
Function and layer storage	75 gigabytes	75 gigabytes	Not available	Account level
Function layers	5	5	Not available	Not adjustable
Function resource-based policy	20 kilobytes	20 kilobytes	Not available	Not adjustable
Function timeout	900	900	Not available	Not adjustable
Processes and threads	1,024	1,024	Not available	Not adjustable
Rate of control plane API requests (excludes invocation, GetFunction, and GetPolicy requests)	15	15	Not available	Not adjustable
Rate of GetFunction API requests	100	100	Not available	Not adjustable
Rate of GetPolicy API requests	15	15	Not available	Not adjustable
Synchronous payload	6 megabytes	6 megabytes	Not available	Not adjustable
Temporary storage	512 megabytes	512 megabytes	Not available	Not adjustable
Test events (console editor)	10	10	Not available	Not adjustable

下列各節依類別列出了 Lambda 中的預設配額和限制。

主題

- 運算和儲存
- 函數組態、部署和執行
- Lambda API 請求
- 其他服務

運算和儲存

Lambda 會為運算和儲存資源的數量設定配額，您可以使用它們來執行並存放函數。並行執行和儲存的配額適用於每個 AWS 區域. 彈性網絡介面 (ENI) 配額適用於每個虛擬私有雲端 (VPC) (不受區域影

響)。下列配額可以從其預設值增加。如需詳細資訊，請參閱《Service Quotas 使用者指南》中的請求提高配額。

資源	預設配額	最多可提高至
並行執行數	1,000	數萬
儲存已上傳的函數 (.zip 封存檔) 和層。每個函數版本和 layer 版本都會消耗儲存空間。 如需管理程式碼儲存的最佳實務，請參閱無伺服器園地中的 監控 Lambda 程式碼儲存。	75 GB	TB
儲存定義為容器映像的函數。這些映像會存放在 Amazon ECR 中。	請參閱 Amazon ECR 服務配額。	
每個 Virtual Private Cloud (VPC) 的彈性網路介面 Note 與其他服務 (例如 Amazon Elastic File System (Amazon EFS)) 共用此配額。請參閱 Amazon VPC 配額。	500	數千

如需有關並行及 Lambda 如何擴展函數並行以回應流量的詳細資訊，請參閱 了解 Lambda 函數擴展。

函數組態、部署和執行

以下配額可套用到函數組態、部署和執行。除非另有說明，否則無法變更。

Note

Lambda 文件、日誌訊息和主控台會使用縮寫 MB (而非 MiB) 來參考 1,024 KB。

資源	配額
函式記憶體分配	128 MB 至 10,240 MB，以 1 MB 遞增。 注意：Lambda 會按設定的記憶體數量比例來配置 CPU 功率。可使用記憶體 (MB) 設定來增加或減少配置給函數的記憶體和 CPU 功率。在 1,769 MB 時，一個函數相當於一個 vCPU。
函數逾時	900 秒 (15 分鐘)
函數環境變數	對於函數相關聯的所有環境變量而言總計為 4 KB
函式以資源為基礎的政策	20 KB
函式 Layer	5 層
函數並行擴展限制	對於每個函數，每 10 秒 1,000 個執行環境
調用承載 (請求和回應)	請求和回應各 6 MB (同步) 每個串流回應 200 MB （同步） 1 MB (非同步) 1 MB (請求行與標頭值的總大小)
串流回應 的頻寬	函數回應的前 6 MB 不受限制 若回應大於 6 MB，則該回應的剩餘部分為 2 MBps
部署套件 (.zip 封存檔) 大小	50 MB (經壓縮；透過 Lambda API 或 SDK上傳時)。使用 Amazon S3 上傳較大的檔案。 50 MB (透過 Lambda 主控台上傳時)

資源	配額
	250 MB (未經壓縮；包含層和自訂執行時期之部署套件內容的大小上限。)
容器映像設定大小	16 KB
容器映像程式碼套件大小	10 GB (未壓縮影像大小上限，包括所有圖層)
測試事件 (主控台編輯器)	10
/tmp 目錄儲存	選擇介於 512 MB 與 10,240 MB 的數量，增量為 1 MB。
檔案描述項	1,024 **Note** Lambda 受管執行個體使用較高的檔案描述項限制 4，096。如需詳細資訊，請參閱了解 Lambda 受管執行個體執行環境。
執行程序/執行緒	1,024 **Note** Lambda 受管執行個體使用 Bottlerocket 的預設程序和執行緒限制。如需詳細資訊，請參閱了解 Lambda 受管執行個體執行環境。

Lambda API 請求

下列配額與 Lambda API 請求關聯。

資源	配額
每個區域的每函數調用請求數 (同步)	執行環境的每個執行個體每秒最多可處理 10 個請求。換句話說，總調用上限是並行上限的 10 倍。請參閱 了解 Lambda 函數擴展。
每個區域的每函數調用請求數 (非同步)	執行環境的每個執行個體都可以處理無限數量的請求。換句話說，總調用上限僅取決於函數可用的並行。請參閱 了解 Lambda 函數擴展。
每個函數版本或別名的調用頻率 (每秒請求數)	10 x 配置的佈建並行 **Note** 此配額僅適用於使用佈建並行的函數。
GetFunction API 請求	每秒 100 個請求。無法提高。
GetPolicy API 請求	每秒 15 個請求。無法提高。
控制平面 API 請求的其餘部分 (不包括調用、GetFunction 和 GetPolicy 請求)	所有 API 每秒 15 個請求 (不是每個 API 每秒 15 個請求)。無法提高。

其他服務

其他服務的配額，例如 AWS Identity and Access Management (IAM)、Amazon CloudFront (Lambda@Edge) 和 Amazon Virtual Private Cloud (Amazon VPC)，可能會影響您的 Lambda 函數。如需詳細資訊，請參閱 Amazon Web Services 一般參考 中的 AWS 服務 quotas 和 使用來自其他服務的事件叫用 Lambda AWS。

許多涉及 Lambda 的應用程式都使用多個 AWS 服務。由於不同的服務在各種功能上具有不同的配額，因此管理整個應用程式中的這些配額可能頗具挑戰性。例如，API Gateway 的預設限流限制為每秒 10,000 個請求，而 Lambda 的預設並行限制為 1,000 個。基於此不一致特性，API Gateway 傳入的請求量可能會超出 Lambda 的處理能力。透過請求提高 Lambda 並行限制來符合預期的流量水準，即可解決此問題。

對應用程式進行負載測試，能讓您在部署至生產環境前，全面監控應用程式的效能表現。在負載測試期間，您可以找出任何可能成為您期望之流量等級的限制因素的配額，並採取相應措施。

文件歷史記錄

下表說明 2018 年 5 月後 AWS Lambda 開發人員指南的重要變更。如需有關此文件更新的通知，您可以訂閱 RSS 摘要。

變更	描述	日期
AWS 受管政策更新	Lambda 已發佈新的受管政策`AWSLambdaBasicDurableExecutionRolePolicy` ，為 CloudWatch Logs 提供寫入許可，並為 Lambda 耐用函數使用的持久執行 APIs 提供讀取/寫入許可。	2025 年 12 月 1 日
Lambda 的服務連結角色	Lambda 現在支援服務連結角色 (`AWSServiceRoleForLambda`)，允許 Lambda 終止作為 Lambda 容量提供者一部分管理的執行個體。	2025 年 11 月 30 日
Lambda 受管執行個體	Lambda 現在支援受管執行個體，為您的函數提供專用運算容量，具有增強的效能、安全性和控制能力。如需詳細資訊，請參閱 Lambda 受管執行個體。	2025 年 11 月 30 日
AWS 受管政策更新	Lambda 新增了新的 AWS 受管政策 (`AWSLambdaBasicDurableExecutionRolePolicy`)。	2025 年 11 月 30 日
AWS 受管政策更新	Lambda 已更新現有的 AWS 受管政策 `AWSLambda_FullAccess`)，允許 `kms:DescribeKey` 和	2025 年 11 月 30 日

	iam:CreateServiceLinkedRole 動作。	
AWS 受管政策更新	Lambda 新增了新的 AWS 受管政策 (AWSLambdaManagedEC2ResourceOperator)，以啟用 Lambda 容量提供者的自動化 Amazon EC2 執行個體管理。如需詳細資訊，請參閱 Lambda 對 AWS 受管政策的更新。	2025 年 11 月 30 日
AWS 受管政策更新	Lambda 新增了新的 AWS 受管政策 (AWSLambdaServiceRolePolicy)。如需詳細資訊，請參閱 Lambda 對 AWS 受管政策的更新。	2025 年 11 月 30 日
AWS 受管政策更新	Lambda 已更新兩個現有的 AWS 受管政策 (AWSLambda_ReadOnlyAccess 和 AWSLambda_FullAccess)。	2025 年 2 月 20 日
Node.js 22.x 執行時期	Lambda 現在支援 Node.js 22 做為受管理的執行時期和容器基礎映像檔 (nodejs22.x)。	2024 年 11 月 22 日
Python 和 .NET 的 SnapStart 支援	Lambda SnapStart 現已可用於 Python 和 .NET 受管執行時期，從 python3.12 和 dotnet8 開始。	2024 年 11 月 18 日
Python 3.13 執行時期	Lambda 現在支援 Python 3.13 做為受管理的執行時期和容器基礎映像。	2024 年 11 月 13 日

.zip 部署套件的客戶受管加密	Lambda 現在支援 .zip 部署套件 AWS KMS 的客戶受管金鑰加密。	2024 年 11 月 8 日
支援新區域中的 SnapStart	下列區域現在可提供 Lambda SnapStart：歐洲 (西班牙)、歐洲 (蘇黎世)、亞太區域 (墨爾本)、亞太區域 (海德拉巴) 以及中東 (UAE)。	2024 年 1 月 12 日
AWS 受管政策更新	Lambda 已更新現有的 AWS 受管政策 (`AWSLambdaVPCAccessExecutionRole`)。	2024 年 1 月 5 日
Python 3.12 執行期	Lambda 現在支援 Python 3.12 做為受管理的執行期和容器基礎映像。如需詳細資訊，請參閱 AWS 運算部落格上的 Python 3.12 執行時間現可在 中使用 AWS Lambda。	2023 年 12 月 14 日
Java 21 執行期	Lambda 現在支援 Java 21 做為受管理的執行期和容器基礎映像檔 (`java21`)。	2023 年 11 月 16 日
Node.js 20.x 執行期	Lambda 現在支援 Node.js 20 做為受管理的執行期和容器基礎映像檔 (`nodejs20.x`)。如需詳細資訊，請參閱 AWS 運算部落格上的 Node.js 20.x 執行時間現可在 中使用 AWS Lambda。	2023 年 11 月 14 日

provided.al2023 執行期	Lambda 現在支援 Amazon Linux 2023 做為受管執行期和容器基礎映像。如需詳細資訊，請參閱 AWS 運算部落格上的 的 Amazon Linux 2023 執行期簡介 AWS Lambda。	2023 年 11 月 9 日
IPv6 支援雙堆疊子網路	Lambda 現在也支援雙堆疊子網路的傳出 IPv6 流量。如需詳細資訊，請參閱 IPv6 支援。	2023 年 10 月 12 日
測試無伺服器函數和應用程式	瞭解在雲端中偵錯和自動測試無伺服器函數的技術。我們目前已在 Python 和 Typescript 程式語言區段加入測試章節和相關資源。如需詳細資訊，請參閱 測試無伺服器函數和應用程式。	2023 年 6 月 16 日
Ruby 3.2 執行期	Lambda 現在支援全新的 Ruby 3.2 執行期。如需詳細資訊，請參閱 使用 Ruby 建置 Lambda 函數。	2023 年 6 月 7 日
回應串流	Lambda 現在支援來自函數的串流回應。如需詳細資訊，請參閱設定 Lambda 函數以串流回應。	2023 年 4 月 6 日
非同步調用指標	Lambda 已發佈非同步調用指標。如需詳細資訊，請參閱非同步調用指標。	2023 年 2 月 9 日

執行階段版本控制項	Lambda 發佈了新的執行階段版本，包含安全性更新、錯誤修正和新功能。您現在可以控制何時將函數更新為新的執行階段版本。如需詳細資訊，請參閱 Lambda 執行階段更新。	2023 年 1 月 23 日
Lambda SnapStart	使用 Lambda SnapStart 縮短 Java 函數的啟動時間，而無需佈建額外資源或實作複雜的效能最佳化。如需詳細資訊，請參閱使用 Lambda SnapStart 提升啟動效能。	2022 年 11 月 28 日
Node.js 18 執行階段	Lambda 現在支援 Node.js 18 的新執行階段。Node.js 18 使用 Amazon Linux 2。如需詳細資訊，請參閱使用 Node.js 建立 Lambda 函數。	2022 年 11 月 18 日
lambda:SourceFunctionArn 條件索引鍵	對於 AWS 資源，`lambda:SourceFunctionArn` 條件索引鍵會依 Lambda 函數的 ARN 篩選對資源的存取。如需詳細資訊，請參閱使用 Lambda 執行環境憑證。	2022 年 7 月 1 日
Node.js 16 執行期	Lambda 現在支援 Node.js 16 的新執行期。Node.js 16 使用 Amazon Linux 2。如需詳細資訊，請參閱使用 Node.js 建立 Lambda 函數。	2022 年 5 月 11 日
Lambda 函數 URL	Lambda 現可支援函數 URL，這是 Lambda 函數專用的 HTTP(S) 端點。如需詳細資訊，請參閱 Lambda 函數 URL。	2022 年 4 月 6 日

AWS Lambda 主控台中的共用測試事件	Lambda 現可支援與相同 AWS 帳戶中的其他使用者共享測試事件。如需詳細資訊，請參閱在主控台中測試 Lambda 函數。	2022 年 3 月 16 日
資源型政策中的 PrincipalOrgId	Lambda 現可支援將許可授予 AWS Organizations中的組織。如需詳細資訊，請參閱對 AWS Lambda使用以資源為基礎的政策。	2022 年 3 月 11 日
.NET 6 執行期	Lambda 現在支援 .NET 6 的新執行期。如需詳細資訊，請參閱 Lambda 執行期。	2022 年 2 月 23 日
篩選 Kinesis、DynamoDB 和 Amazon SQS 等事件來源的事件	Lambda 現在支援篩選 Kinesis、DynamoDB 和 Amazon SQS 等事件來源的事件 如需詳細資訊，請參閱 Lambda 事件篩選。	2021 年 11 月 24 日
Amazon MSK 的 mTLS 身分驗證和自我管理的 Apache Kafka 事件來源	Lambda 現在支援 Amazon MSK 的 mTLS 身分驗證和自我管理的 Apache Kafka 事件來源。如需詳細資訊，請參閱搭配使用 Lambda 與 Amazon MSK。	2021 年 11 月 19 日
Lambda on Graviton2	Lambda 現在支援 Graviton2 使用 arm64 架構運作。如需詳細資訊，請參閱 Lambda 指令集架構。	2021 年 9 月 29 日
Python 3.9 執行期	Lambda 現在支援 Python 3.9 的新執行期。如需詳細資訊，請參閱 Lambda 執行期。	2021 年 8 月 16 日

適用於 Node.js、Python 和 Java 的新執行期版本	新執行期版本適用於 Node.js、Python 和 Java。如需詳細資訊，請參閱 Lambda 執行期。	2021 年 7 月 21 日
支援 RabbitMQ 作為 Lambda 的事件來源	Lambda 現在支援 Amazon MQ for RabbitMQ 作為事件來源。Amazon MQ 是一種適用於 Apache ActiveMQ 和 RabbitMQ 的受管型訊息代理程式服務，可讓您輕鬆地在雲端設定及操作訊息代理程式。如需詳細資訊，請參閱搭配使用 Lambda 與 Amazon MQ。	2021 年 7 月 7 日
Lambda 上適用於自我管理型 Kafka 的 SASL/PLAIN 身分驗證	SASL/PLAIN 現在是 Lambda 上適用於自我管理型 Kafka 事件來源的受支援身分驗證機制。已在其自我管理型 Kafka 叢集上使用 SASL/PLAIN 的客戶現在可以輕鬆地使用 Lambda 來建置消費者應用程式，而不必修改身分驗證方式。如需詳細資訊，請參閱搭配使用 Lambda 與自我管理 Apache Kafka。	2021 年 6 月 29 日
Lambda Extensions API	Lambda 擴展的一般可用性。使用擴展來增強 Lambda 函數。您可以使用 Lambda 合作夥伴提供的擴展，也可以建立自己的 Lambda 擴展。如需詳細資訊，請參閱 Lambda Extensions API。	2021 年 5 月 24 日
全新 Lambda 主控台體驗	Lambda 主控台已重新設計，以改善效能和一致性。	2021 年 3 月 2 日

Node.js 14 執行期	Lambda 現在支援 Node.js 14 的新執行期。Node.js 14 使用 Amazon Linux 2。如需詳細資訊，請參閱使用 Node.js 建立 Lambda 函數。	2021 年 1 月 27 日
Lambda 容器映像	Lambda 現在支援定義為容器映像的函數。您可以結合容器工具的靈活性和 Lambda 的敏捷性和操作簡便性來建置應用程式。如需詳細資訊，請參閱將容器映像與 Lambda 搭配使用。	2020 年 12 月 1 日
Lambda 函數的程式碼簽署	Lambda 現在支援程式碼簽署。系統管理員可以將 Lambda 函數設定為只接受在部署時簽署的程式碼。Lambda 會檢查簽名，以確保程式碼不會遭到更改或竄改。此外，Lambda 可確保程式碼在接受部署之前，由受信任的開發人員簽署。如需詳細資訊，請參閱設定 Lambda 的程式碼簽署。	2020 年 11 月 23 日
預覽：Lambda Runtime Logs API	Lambda 現在支援 Runtime Logs API。Lambda 擴展可以使用 Logs API 來訂閱執行環境中的記錄串流。如需詳細資訊，請參閱 Lambda Runtime Logs API。	2020 年 11 月 12 日

Amazon MQ 的新事件來源	Lambda 現在支援 Amazon MQ 作為事件來源。使用 Lambda 函數來處理您的 Amazon MQ 訊息代理程式中的記錄。如需詳細資訊，請參閱搭配使用 Lambda 與 Amazon MQ。	2020 年 11 月 5 日
預覽：Lambda Extensions API	使用 Lambda 擴展來增強 Lambda 函數。您可以使用 Lambda 合作夥伴提供的擴展，也可以建立自己的 Lambda 擴展。如需詳細資訊，請參閱 Lambda Extensions API。	2020 年 10 月 8 日
AL2 支援 Java 8 和自訂執行期	Lambda 目前在 Amazon Linux 2 中支援 Java 8 和自訂執行期。如需詳細資訊，請參閱 Lambda 執行期。	2020 年 8 月 12 日
Amazon Managed Streaming for Apache Kafka 的新事件來源	Lambda 現在支援 Amazon MSK 作為事件來源。搭配使用 Lambda 函數與 Amazon MSK 來處理 Kafka 主題中的記錄。如需詳細資訊，請參閱搭配使用 Lambda 與 Amazon MSK。	2020 年 8 月 11 日
Amazon VPC 設定的 IAM 條件金鑰	您現在可以針對 VPC 設定使用 Lambda 特定條件金鑰。例如，您可以請求組織中的所有功能都連線到 VPC。您也可以指定函數的使用者可以和不可以使用的子網路和安全性群組。如需詳細資訊，請參閱針對 IAM 函數設定 VPC。	2020 年 8 月 10 日

Kinesis HTTP/2 串流取用程式的並行設定	您現在可以針對具有增強散發 (HTTP/2 資料串流) 的 Kinesis 取用程式使用下列並行設定：ParallelizationFactor、MaximumRetryAttempts、MaximumRecordAgeInSeconds、DestinationConfig 和 BisectBatchOnFunctionError。如需詳細資訊，請參閱搭配使用 AWS Lambda 與 Amazon Kinesis。	2020 年 7 月 7 日
Kinesis HTTP/2 串流取用程式的批次間隔	您現在可以設定 HTTP/2 串流的批次間隔 (MaximumBatchingWindowInSeconds)。Lambda 會從串流中讀取記錄，直到收集到完整批次，或直到批次間隔到期。如需詳細資訊，請參閱搭配使用 AWS Lambda 與 Amazon Kinesis。	2020 年 6 月 18 日
支援 Amazon EFS 檔案系統	您現在可以將 Amazon EFS 檔案系統連線到您的 Lambda 函數，以獲得共用網路檔案存取權。如需詳細資訊，請參閱設定 Lambda 函數的檔案系統存取權。	2020 年 6 月 16 日

AWS CDK Lambda 主控台中的範例應用程式	Lambda 主控台現在包含使用 AWS Cloud Development Kit (AWS CDK) for TypeScript 的範例應用程式。 AWS CDK 是一個架構，可讓您在 TypeScript、Python、Java 或 .NET 中定義應用程式資源。	2020 年 6 月 1 日
支援 中的 .NET Core 3.1.0 執行時間 AWS Lambda	AWS Lambda 現在支援 .NET Core 3.1.0 執行時間。如需詳細資訊，請參閱 .NET Core CLI。	2020 年 3 月 31 日
支援 API Gateway HTTP API	更新和擴增文件以便搭配使用 Lambda 與 API Gateway，包括對 HTTP API 的支援。新增使用 建立 API 和 函數的範例應用程式 CloudFormation。如需詳細資訊，請參閱搭配使用 Lambda 與 Amazon API Gateway。	2020 年 3 月 23 日
Ruby 2.7	Ruby 2.7 有新的執行期可用，ruby2.7 是第一個可使用 Amazon Linux 2 的 Ruby 執行期。如需詳細資訊，請參閱使用 Ruby 建立 Lambda 函數。	2020 年 2 月 19 日

並行指標	Lambda 現在會針對所有函數、別名和版本來報告 ConcurrentExecutions 指標。您可以在函式的監督頁面檢視此指標的圖表。先前，ConcurrentExecutions 只會報告帳戶層級和使用保留並行的函式。如需詳細資訊，請參閱 AWS Lambda 函數指標。	2020 年 2 月 18 日
函數狀態更新	預設情況下，所有函數皆會強制執行函數狀態。當您將函數連線到 VPC 時，Lambda 會建立共用的彈性網路介面。如此一來，您的函數即可在不建立其他網路介面的情況下輕鬆擴展規模。在此期間，您無法對函數執行其他操作，包括更新其組態和發佈版本。在某些情況下，調用也會受到影響。關於函數目前狀態的詳細資訊可從 Lambda API 中獲得。 此更新將分階段發佈。如需詳細資訊，請參閱 AWS 運算部落格上的已更新 VPC 聯網的 Lambda 狀態生命週期。如需狀態的詳細資訊，請參閱 AWS Lambda 函數狀態。	2020 年 1 月 24 日

函數組態 API 輸出更新	將原因代碼新增至連線到 VPC 函數的 StateReasonCode (InvalidSubnet, InvalidSecurityGroup) and LastUpdateStatusReasonCode (SubnetOutOfIPAddresses, InvalidSubnet, InvalidSecurityGroup)。如需狀態的詳細資訊，請參閱 AWS Lambda 函數狀態。	2020 年 1 月 20 日
佈建並行	您現在可以為函數版本或別名來配置佈建並行。佈建並行可讓函數在不造成延遲波動的情況下進行擴展。如需詳細資訊，請參閱管理 Lambda 函數的並行。	2019 年 12 月 3 日
建立資料庫代理	您現在可以使用 Lambda 主控台為 Lambda 函數建立資料庫代理。資料庫代理可讓函數在不耗盡資料庫連線的情況下達到高並行層級。如需詳細資訊，請參閱設定 Lambda 函數的資料庫存取權。	2019 年 12 月 3 日
持續時間指標支援百分位數	您現在可以根據百分位數來篩選持續時間指標。如需詳細資訊，請參閱 AWS Lambda 指標。	2019 年 11 月 26 日

增加串流事件來源的並行	DynamoDB 串流和 Kinesis 串流事件來源映射的新選項，可讓您從每個碎片一次處理多個批次。當您增加每個分片的並行批次數量時，函數的並行最高可以是串流中碎片數量的 10 倍。如需詳細資訊，請參閱 Lambda 事件來源映射。	2019 年 11 月 25 日
函數狀態	當您建立或更新函數時，函數會進入待定狀態，Lambda 會同時佈建資源以提供支援。如果您將函數連線至 VPC，Lambda 可立即建立共用的彈性網路介面，而不是在調用函數時建立網路介面。這可為連線至 VPC 的函數帶來較佳效能，但可能需要更新您的自動化。如需詳細資訊，請參閱 AWS Lambda 函數狀態。	2019 年 11 月 25 日
處理非同步調用選項時的錯誤	新的組態選項可用於非同步調用。您可以設定 Lambda，使其限制重試次數，並設定事件存留期的上限。如需詳細資訊，請參閱設定處理非同步調用時的錯誤。	2019 年 11 月 25 日

處理串流事件來源時的錯誤	有新的組態選項，可用於讀取自串流的事件來源映射。您可以設定 DynamoDB 串流和 Kinesis 串流事件來源映射以限制重試，並設定記錄保留期限的上限。發生錯誤時，您可以先將事件來源映射設定為分割批次，再進行重試，並將失敗批次的呼叫記錄傳送到佇列或主題。如需詳細資訊，請參閱 Lambda 事件來源映射。	2019 年 11 月 25 日
非同步調用的目的地	您現在可以設定 Lambda，使其將非同步調用的記錄傳送到另一個服務。呼叫記錄包含事件、內容和函數回應的詳細資訊。您可以將呼叫記錄傳送到 SQS 佇列、SNS 主題、Lambda 函數或 EventBridge 事件匯流排。如需詳細資訊，請參閱設定非同步調用的目的地。	2019 年 11 月 25 日
適用於 Node.js、Python 和 Java 的新執行期	新的執行期可用於 Node.js 12、Python 3.8 和 Java 11。如需詳細資訊，請參閱 Lambda 執行期。	2019 年 11 月 18 日
FIFO 佇列支援 Amazon SQS 事件來源	您現在可以建立從先進先出 (FIFO) 佇列讀取內容的事件來源映射。先前，只支援標準佇列。如需詳細資訊，請參閱搭配使用 Lambda 與 Amazon SQS。	2019 年 11 月 18 日

在 Lambda 主控台中建立應用程式	現在通常可以在 Lambda 主控台中建立應用程式。如需說明，請參閱在 Lambda 主控台中管理應用程式。	2019 年 10 月 31 日
在 Lambda 主控台中建立應用程式 (Beta 版)	您現在可以在 Lambda 主控台中建立具有整合式持續交付管道的 Lambda 應用程式。主控台會提供範例應用程式，您可以使用這些應用程式做為自己專案的起點。在 AWS CodeCommit 和 GitHub 之間進行來源控制。每次您將變更推送到儲存庫時，包含的管道便會自動建置和部署它們。如需說明，請參閱在 Lambda 主控台中管理應用程式。	2019 年 10 月 3 日
改善 VPC 連線函數的效能	Lambda 現在使用 Virtual Private Cloud (VPC)) 子網路中所有函數共享的新彈性網路介面類型。當您將函數連接到 VPC 時，Lambda 會為您選擇的每個安全群組和子網路組合建立網路介面。當有共享網路介面可供使用時，函數就不再需要在擴展時建立額外的網路介面。這大幅改善了啟動時間。如需詳細資訊，請參閱設定 Lambda 函數存取 VPC 中的資源。	2019 年 9 月 3 日

串流批次設定	您現在可以為 Amazon DynamoDB 和 Amazon Kinesis 事件來源映射設定批次時段。設定最多五分鐘的批次時段來緩衝傳入的記錄，直到完整批次可用為止。這可減少當串流較不活躍時調用函式的次數。	2019 年 8 月 29 日
整合 CloudWatch Logs Insights	Lambda 主控台中的監控頁面現在包含來自 Amazon CloudWatch Logs Insights 的報告。	2019 年 6 月 18 日
Amazon Linux 2018.03	正在更新 Lambda 執行環境以使用 Amazon Linux 2018.03。如需詳細資訊，請參閱執行環境。	2019 年 5 月 21 日
Node.js 10	適用於 Node.js 10、nodejs 10.x. 的新執行期 這個執行期使用 Node.js 10.15，且系統會定期使用最新版的 Node.js 10 來更新此執行期。Node.js 10 也是第一個使用 Amazon Linux 2 的執行期。如需詳細資訊，請參閱使用 Node.js 建立 Lambda 函數。	2019 年 5 月 13 日
GetLayerVersionByArn API	使用 GetLayerVersionByArn API 輸入版本 ARN，下載 Layer 版本資訊。相較於 GetLayerVersion，GetLayerVersionByArn 可讓您直接使用 ARN，而不用再經過剖析取得 Layer 名稱和版本編號。	2019 年 4 月 25 日

Ruby	AWS Lambda 現在支援具有新執行時間的 Ruby 2.5。如需詳細資訊，請參閱使用 Ruby 建立 Lambda 函數。	2018 年 11 月 29 日
圖層	使用 Lambda 層，您可以單獨封裝和部署程式庫、自訂執行期及其他依存項目，與您的函數程式碼分開。與您的其他帳戶甚至世界各地的人共享您的 Layer。如需詳細資訊，請參閱 Lambda 層。	2018 年 11 月 29 日
自訂執行期	使用您慣用的程式設計語言建置自訂執行期以執行 Lambda 函數。如需詳細資訊，請參閱自訂 Lambda 執行期。	2018 年 11 月 29 日
Application Load Balancer 觸發	Elastic Load Balancing 現在支援 Lambda 函數作為 Application Load Balancer 的目標。如需詳細資訊，請參閱搭配使用 Lambda 與 Application Load Balancer。	2018 年 11 月 29 日
使用 Kinesis HTTP/2 串流消費者作為觸發條件	您可以使用 Kinesis HTTP/2 資料串流消費者，將事件傳送至 AWS Lambda。串流消費者有來自資料串流中每個碎片的專屬讀取傳輸量，並使用 HTTP/2 將延遲降至最低。如需詳細資訊，請參閱搭配使用 Lambda 與 Kinesis。	2018 年 11 月 19 日

Python 3.7	AWS Lambda 現在支援具有新執行時間的 Python 3.7。如需詳細資訊，請參閱使用 Python 建置 Lambda 函數。	2018 年 11 月 19 日
提高非同步函數呼叫的承載限制	非同步調用的承載大小上限已從 128 KB 提高至 256 KB，以符合來自 Amazon SNS 觸發的訊息大小上限。如需詳細資訊，請參閱 Lambda 配額。	2018 年 11 月 16 日
AWS GovCloud （美國東部）區域	AWS Lambda 現在可在 AWS GovCloud （美國東部） 區域使用。	2018 年 11 月 12 日
將 AWS SAM 主題移至單獨的開發人員指南	許多主題著重於使用 AWS Serverless Application Model () 建置無伺服器應用程式AWS SAM。這些主題已移至 AWS Serverless Application Model 開發人員指南。	2018 年 10 月 25 日
在主控台中檢視 Lambda 應用程式	您可以在 Lambda 主控台的應用程式頁面中檢視 Lambda 應用程式的狀態。此頁面顯示 CloudFormation 堆疊的狀態。它包含多個頁面的連結，可讓您檢視有關堆疊中的資源的詳細資訊。您也可以檢視應用程式的彙總指標並建立自訂的監控儀表板。	2018 年 10 月 11 日
函數執行逾時限制	若要允許長時間執行的函數，可設定的最長執行逾時可從 5 分鐘增加為 15 分鐘。如需詳細資訊，請參閱 Lambda 限制。	2018 年 10 月 10 日

支援 中的 PowerShell Core 語言 AWS Lambda	AWS Lambda 現在支援 PowerShell Core 語言。如需詳細資訊，請參閱以 PowerShell 撰寫 Lambda 函數的程式設計模型。	2018 年 9 月 11 日
支援 中的 .NET Core 2.1.0 執行時間 AWS Lambda	AWS Lambda 現在支援 .NET Core 2.1.0 執行時間。如需詳細資訊，請參閱 .NET Core CLI。	2018 年 7 月 9 日
現在可以透過 RSS 獲得更新	您現在可以訂閱 RSS 摘要，以追蹤本指南的發行版本。	2018 年 7 月 5 日
支援 Amazon SQS 作為事件來源	AWS Lambda 現在支援 Amazon Simple Queue Service (Amazon SQS) 做為事件來源。如需詳細資訊，請參閱調用 Lambda 函數。	2018 年 6 月 28 日
中國 (寧夏) 區域	AWS Lambda 現已在中國（寧夏）區域提供。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2018 年 6 月 28 日

舊版更新

下表說明 2018 年六月前每個 AWS Lambda 開發人員指南版本的重要變更。

變更	描述	Date
執行期支援 Node.js 執行期 8.10	AWS Lambda 現在支援 Node.js 執行時間 8.10 版。如需詳細資訊，請參閱使用 Node.js 建置 Lambda 函數。	2018 年 4 月 2 日

變更	描述	Date
函式和別名修訂 ID	AWS Lambda 現在支援函數版本和別名上的修訂 IDs。在更新函式版本或別名資源時，您可以使用這些 ID 來追蹤和套用條件更新。	2018 年 1 月 25 日
Go 和 .NET 2.0 的執行期支援	AWS Lambda 已新增 Go 和 .NET 2.0 的執行時間支援。如需詳細資訊，請參閱使用 Go 建置 Lambda 函數及使用 C# 建置 Lambda 函數。	2018 年 1 月 15 日
主控台重新設計	AWS Lambda 已推出新的 Lambda 主控台來簡化您的體驗，並新增 Cloud9 程式碼編輯器來增強您的功能偵錯和修訂函數程式碼。	2017 年 11 月 30 日
為個別函式設定並行限制	AWS Lambda 現在支援設定個別函數的並行限制。如需詳細資訊，請參閱設定函數的預留並行。	2017 年 11 月 30 日
使用別名轉移流量	AWS Lambda 現在支援使用別名轉移流量。如需詳細資訊，請參閱使用加權別名建立滾動部署。	2017 年 11 月 28 日
逐步程式碼部署	AWS Lambda 現在支援利用 Code Deploy 安全地部署 Lambda 函數的新版本。如需詳細資訊，請參閱逐步程式碼部署。	2017 年 11 月 28 日
中國 (北京) 區域	AWS Lambda 現已在中國（北京）區域提供。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2017 年 11 月 9 日
介紹 SAM Local	AWS Lambda 推出 SAM Local（現稱為 SAM CLI)，此 AWS CLI 工具提供環境供您在本機開發、測試和分析無伺服器應用程式，再上傳至 Lambda 執行期。如需詳細資訊，請參閱測試與偵錯無伺服器應用程式。	2017 年 8 月 11 日
加拿大 (中部) 區域	AWS Lambda 現在可在加拿大（中部）區域使用。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2017 年 6 月 22 日

變更	描述	Date
南美洲 (聖保羅) 區域	AWS Lambda 現已在南美洲（聖保羅）區域提供。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2017 年 6 月 6 日
AWS Lambda 的 支援 AWS X-Ray。	Lambda 推出對 X-Ray 的支援，它許您針對 Lambda 應用程式偵測、分析和最佳化效能問題。如需詳細資訊，請參閱使用 視覺化 Lambda 函數叫用 AWS X-Ray。	2017 年 4 月 19 日
亞太地區 (孟買) 區域	AWS Lambda 現可於亞太區域（孟買）區域使用。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2017 年 3 月 28 日
AWS Lambda 現在支援 Node.js 執行時間 v6.10	AWS Lambda 新增對 Node.js 執行期 v6.10 的支援。如需詳細資訊，請參閱使用 Node.js 建置 Lambda 函數。	2017 年 3 月 22 日
歐洲 (倫敦) 區域	AWS Lambda 現已在歐洲（倫敦）區域提供。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2017 年 2 月 1 日
AWS Lambda 支援 .NET 執行時間、Lambda@Edge（預覽）、無效字母佇列和無伺服器應用程式的自動化部署。	AWS Lambda 新增對 C# 的支援。如需詳細資訊，請參閱使用 C# 建置 Lambda 函數。 Lambda@Edge 可讓您在 AWS Edge 位置執行 Lambda 函數，以回應 CloudFront 事件。如需詳細資訊，請參閱使用 Lambda@Edge 在邊緣進行自訂。	2016 年 12 月 3 日
AWS Lambda 新增 Amazon Lex 做為支援的事件來源。	使用 Lambda 和 Amazon Lex，您便能針對 Slack 及 Facebook 之類的各種服務，快速建置聊天機器人。	2016 年 11 月 30 日
美國西部 (加利佛尼亞北部) 區域	AWS Lambda 現已在美國西部（加利佛尼亞北部）區域提供。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2016 年 11 月 21 日

變更	描述	Date
引進 AWS SAM ，用於建立和部署 Lambda 型應用程式，以及使用 Lambda 函數組態設定的環境變數。	AWS SAM：您現在可以使用 AWS SAM 來定義在無伺服器應用程式中表達資源的語法。為了部署應用程式，只需指定需要的資源當做應用程式的一部分，以及於 AWS CloudFormation 範本檔案中與其相關聯的許可政策 (使用 JSON 或 YAML 撰寫)，封裝部署成品並部署範本。 環境變數：您可以使用環境變數來指定您的函數程式碼外的 Lambda 函數的組態設定。如需詳細資訊，請參閱使用 Lambda 環境變數。	2016 年 11 月 18 日
亞太 (首爾) 區域	AWS Lambda 現可於亞太區域 （ 首爾 ） 區域使用。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2016 年 8 月 29 日
亞太地區 (雪梨) 區域	亞太區域 (雪梨) 現在可以使用 Lambda。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2016 年 6 月 23 日
更新至 Lambda 主控台	已更新 Lambda 主控台來簡化角色建立流程。	2016 年 6 月 23 日
AWS Lambda 現在支援 Node.js 執行時間 v4.3	AWS Lambda 新增對 Node.js 執行期 v4.3 的支援。如需詳細資訊，請參閱使用 Node.js 建置 Lambda 函數。	2016 年 4 月 7 日
歐洲 (法蘭克福) 區域	歐洲 (法蘭克福) 區域現在可以使用 Lambda。如需有關 Lambda 區域和端點的詳細資訊，請參閱《AWS 一般參考》中的區域與端點。	2016 年 3 月 14 日
VPC 支援	您現在可以設定 Lambda 函數來存取 VPC 中的資源。如需詳細資訊，請參閱讓 Lambda 函數存取 Amazon VPC 中的資源。	2016 年 2 月 11 日
Lambda 執行期已更新。	已更新執行環境。	2015 年 11 月 4 日

變更	描述	Date
版本控制支援、用於開發 Lambda 函數程式碼的 Python、已排程事件，以及增加執行期	您現在可以使用 Python 開發您的 Lambda 函式程式碼。如需詳細資訊，請參閱使用 Python 建置 Lambda 函數。 版本控制：您可以維護一個或多個 Lambda 函式的版本。版本控制功能可以控制讓哪個 Lambda 函式版本在不同的環境中執行 (例如，開發、測試或生產)。如需詳細資訊，請參閱管理 Lambda 函數版本。 已排程事件：您也可使用 Lambda 主控台來設定 Lambda，以便在排程時間定期調用您的程式碼。您可以指定固定頻率 (時數、日數或週數)，或是指定一個 Cron 表達式。如需詳細資訊，請參閱依排程調用 Lambda 函數。 增加執行期：您現在可以設定 Lambda 函式執行五分鐘，允許更長的函式執行期，例如大量資料擷取和處理工作。	2015 年 10 月 8 日
支援 DynamoDB Streams	現在普遍都可使用 DynamoDB Streams，而且您可以在所有提供 DynamoDB 的地區使用它。您可以為您的資料表啟用 DynamoDB Streams，並使用 Lambda 函數當做資料表的觸發。觸發是您回應 DynamoDB 資料表更新所採取的自訂動作。如需範例演練，請參閱 教學課程： AWS Lambda 搭配 Amazon DynamoDB 串流使用。	2015 年 7 月 14 日
Lambda 現在支援使用與 REST 相容的用戶端調用 Lambda 函數。	到目前為止，若要從 Web、行動裝置或 IoT 應用程式叫用 Lambda 函數，您需要開發套件 AWS SDKs AWS SDK、適用於 Android 的 AWS SDK 或適用於 iOS 的 AWS SDK)。現在，Lambda 支援透過自訂的 API (您可使用 Amazon API Gateway 建立)，使用與 REST 相容的用戶端調用 Lambda 函數。您可以將請求傳送到 Lambda 函式端點 URL。您可以在端點上設定安全性，以允許開放存取，利用 AWS Identity and Access Management (IAM) 授權存取，或使用 API 金鑰來測量其他人對您的 Lambda 函數的存取。 如需入門練習範例，請參閱 使用 Amazon API Gateway 端點調用 Lambda 函數。	2015 年 7 月 9 日

變更	描述	Date
Lambda 主控台現在提供藍圖，可輕鬆建立 Lambda 函數並進行測試。	Lambda 主控台提供一組藍圖。每個藍圖為您的 Lambda 函式提供範例事件來源組態和範本程式碼，而該函式則是用於輕鬆建立以 Lambda 為基礎的應用程式。所有的 Lambda 入門練習現在皆使用藍圖。	2015 年 7 月 9 日
Lambda 現在支援用 Java 撰寫 Lambda 函數。	您現在可以使用 Java 撰寫 Lambda 程式碼。如需詳細資訊，請參閱使用 Java 建置 Lambda 函數。	2015 年 6 月 15 日
Lambda 現在支援在建立或更新 Lambda 函數時，指定 Amazon S3 物件作為函數 .zip 檔。	您可以將 Lambda 函式部署套件 (.zip 檔案) 上傳至相同區域內的 Amazon S3 儲存貯體，該處是您想要建立 Lambda 函式所在。然後，在建立或更新 Lambda 函式時，指定儲存貯體名稱和物件金鑰名稱。	2015 年 5 月 28 日
Lambda 現在已普遍可用，並已增加行動後端的支援	Lambda 現在已普遍適用於生產用途。此版本也推出了新功能，可使用 Lambda 更輕鬆建置行動裝置、平板電腦及物聯網 (IoT) 後端，無需佈建或管理基礎設施便可自動擴展。Lambda 現在支援即時 (同步) 與非同步事件。其他功能包括更簡單的事件來源組態和管理。許可模型和程式設計模型已因推出 Lambda 函式適用的資源政策而更加簡化。	2015 年 4 月 9 日
預覽版	AWS Lambda 開發人員指南的預覽版。	2014 年 11 月 13 日

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。