aws

开发人员指南

AWS 数据库加密 SDK

AWS 数据库加密 SDK: 开发人员指南

Table of Contents

什么是 AWS 数据库加密 SDK？

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

AWS 数据库加密 SDK 是一组软件库，可让您在数据库设计中加入客户端加密。 AWS 数据库加密 SDK 提供记录级加密解决方案。您可以指定哪些字段经过加密，哪些字段包含在确保数据真实性的签名中。加密传输中敏感数据和静态敏感数据有助于确保您的明文数据不会提供给任何第三方，包括 AWS。 AWS 数据库加密 SDK 是根据 Apache 2.0 许可证免费提供的。

本开发人员指南提供了 AWS 数据库加密 SDK 的概念性概述，包括其架构简介、它如何保护您的数据、它与服务器端加密有何不同之处，以及为应用程序选择关键组件以帮助您入门的指南。

AWS 数据库加密软件开发工具包支持具有属性级加密功能的 Amazon DynamoDB。

AWS 数据库加密 SDK 具有以下优点：

专门针对数据库应用程序而设计

您无需成为密码专家即可使用 AWS 数据库加密 SDK。实施包括旨在处理您的现有应用程序的帮助程序方法。

在创建和配置所需组件后，加密客户端在您将记录添加到数据库时以透明方式加密并签署这些记录，并且在检索记录时验证和解密它们。

包括安全加密和签名

D AWS atabase Encryption SDK 包含安全的实现，即使用唯一的数据加密密钥对每条记录中的字段值进行加密，然后对记录进行签名以防止未经授权的更改，例如添加或删除字段或交换加密值。

使用来自任何源的加密材料

AWS 数据库加密 SDK 使用密钥环生成、加密和解密保护您的记录的唯一数据加密密钥。密钥环决定加密该数据密钥的包装密钥。

您可以使用来自任何来源的包装密钥，包括加密服务，例如 AWS Key Management Service（AWS KMS）或 AWS CloudHSM。 AWS 数据库加密 SDK 不需要 AWS 账户 任何 AWS 服务。

支持加密材料缓存

AWS KMS 分层密钥环是一种加密材料缓存解决方案，它使用 AWS KMS 保存在Amazon DynamoDB表中的受保护分支密钥，然后在本地缓存用于加密和解密操作的分支密钥材料，从而减少 AWS KMS 调用次数。它允许您在对称加密 KMS 密钥下保护您的加密材料，而无需在 AWS KMS 每次加密或解密记录时都调用。对于需要最大限度地减少调用的应用程序来说， AWS KMS 分层密钥环是一个不错的 AWS KMS选择。

可搜索的加密

您可以设计无需解密整个数据库即可搜索已加密记录的数据库。根据您的威胁模型和查询要求，您可以使用可搜索的加密对已加密数据库执行精确匹配搜索或更自定义的复杂查询。

支持多租户数据库架构

Dat AWS abase Encryption SDK 使您能够使用不同的加密材料隔离每个租户，从而保护存储在共享架构的数据库中的数据。如果您有多个用户在数据库中执行加密操作，请使用其中一个 AWS KMS 密钥环为每个用户提供一个用于其加密操作的不同密钥。有关更多信息，请参阅 使用多租户数据库。

支持无缝架构更新

在配置 AWS 数据库加密 SDK 时，您需要提供加密操作，告诉客户端要加密和签名哪些字段，哪些字段需要签名（但不加密），以及要忽略哪些字段。使用 AWS 数据库加密 SDK 保护记录后，您仍然可以对数据模型进行更改。您可以在单个部署中更新您的加密操作，例如添加或移除已加密的字段。

在开源存储库中开发

AWS 数据库加密 SDK 是在上的开源存储库中开发的 GitHub。您可以使用这些存储库查看代码、阅读和提交问题，并且查找特定于您的实施的信息。

适用于 DynamoDB 的 AWS 数据库加密 SDK

- 上的 aws-database-encryption-sdk-dynamodb 存储库 GitHub 支持 Java、.NET 和 Rust 中最新版本的 DynamoDB AWS 数据库加密 SDK。

  适用于 DynamoDB 的 AWS 数据库加密 SDK 是 Dafny 的产品，Dafny 是一种验证感知语言，你可以用它来编写规范、实现规范、代码和测试规范。结果为在确保功能正确性的框架中实施适用于 DynamoDB 的 AWS 数据库加密 SDK 功能的库。

支持和维护

AWS 数据库加密 SDK 使用与 AWS SDK 和工具相同的维护策略，包括版本控制和生命周期阶段。作为最佳实践，建议您使用适用于您的数据库实现的 AWS 数据库加密 SDK 的最新可用版本，并在新版本发布时进行升级。

有关更多信息，请参阅《工具参考指南》AWS SDKs和《工具参考指南》中的 AWS SDKs 和工具维护政策。

发送反馈

我们欢迎您提供反馈！如果您有任何疑问或意见或者要报告问题，请使用以下资源。

如果您在 AWS 数据库加密 SDK 中发现潜在的安全漏洞，请通知 AWS 安全部门。不要创建公开 GitHub 问题。

要提供对本文档的反馈，请使用任何页面上的反馈链接。

AWS 数据库加密 SDK 概念

> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

本主题介绍 AWS 数据库加密 SDK 中使用的概念和术语。

要了解 AWS 数据库加密 SDK 的组件是如何交互的，请参阅AWS 数据库加密 SDK 的工作原理。

要了解有关 AWS 数据库加密 SDK 的更多信息，请参阅以下主题。

- 了解 AWS 数据库加密 SDK 如何使用信封加密来保护您的数据。
- 了解信封加密的要素：保护记录的数据密钥以及保护数据密钥的包装密钥。
- 了解决定您使用哪种包装密钥的密钥环。
- 了解可增强加密过程完整性的加密上下文。
- 了解加密方法在您的记录中添加的材料描述。
- 了解告诉 AWS 数据库加密 SDK 要加密和签名的字段的加密操作。

主题

- 信封加密
- 数据密钥
- 包装密钥
- 密钥环
- 加密操作
- 材料描述
- 加密上下文
- 加密材料管理器
- 对称和非对称加密
- 密钥承诺
- 数字签名

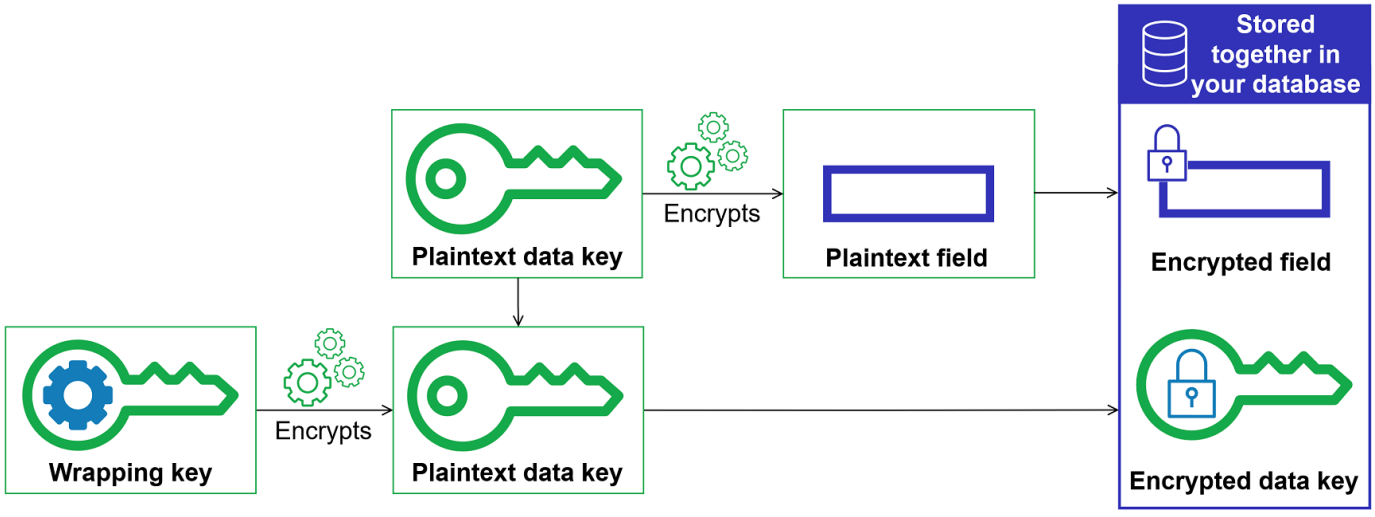
信封加密

加密的数据的安全性部分取决于如何保护可解密该数据的数据密钥。保护数据密钥的一种公认的最佳实践是对其进行加密。为此，您需要另一个加密密钥，称为密钥加密密钥或包装密钥。使用包装密钥加密数据密钥的做法称为信封加密。

保护数据密钥

AWS 数据库加密 SDK 使用唯一的数据密钥对每个字段进行加密。然后，对您指定的包装密钥下的每个数据密钥进行加密。它将已加密的数据密钥存储在材料描述中。

要指定包装密钥，您可以使用密钥环。



在多个包装密钥下加密相同的数据

您可以使用多个包装密钥加密数据密钥。您可能希望为不同的用户提供不同的包装密钥，或者提供不同类型的包装密钥，或者位于不同的位置。每个包装密钥都加密相同的数据密钥。 AWS 数据库加密 SDK 将所有加密的数据密钥与材料描述中的加密字段一起存储。

要解密数据，您需要至少提供一个可以解密已加密数据密钥的包装密钥。

结合多种算法的优势

为了加密您的数据，默认情况下， AWS 数据库加密 SDK 使用带有 AES-GCM 对称加密、基于 HMAC 的密钥派生函数 (HKDF) 和 ECDSA 签名的算法套件。要加密数据密钥，您可以指定适合您的包装密钥的对称或非对称加密算法。

通常，与非对称或公有密钥加密 相比，对称密钥加密算法速度更快，生成的密文更小。但公有密钥算法可提供固有的角色分离。为了结合每种算法的优势，您可以使用公有密钥加密功能加密数据密钥。

我们建议尽可能使用其中一个 AWS KMS 钥匙圈。使用AWS KMS 密钥环时，您可以选择通过将非对称 RSA AWS KMS key 指定为包装密钥来组合多种算法的优势。您也可以使用对称加密 KMS 密钥。

数据密钥

数据密钥是一种加密密钥， AWS 数据库加密 SDK 使用它来加密记录中在加密操作ENCRYPT_AND_SIGN中标记的字段。每个数据密钥都是一个符合加密密钥要求的字节数组。 AWS 数据库加密 SDK 使用唯一的数据密钥来加密每个属性。

您无需指定、生成、实施、扩展、保护或使用数据密钥。当您调用加密和解密操作时， AWS 数据库加密 SDK 会替您完成这些工作。

为了保护您的数据密钥， AWS 数据库加密 SDK 使用一个或多个密钥加密密钥（称为包装密钥）对其进行加密。在 AWS 数据库加密 SDK 使用您的纯文本数据密钥加密您的数据后，它会尽快将其从内存中删除。然后再将加密的数据密钥存储在材料描述中。有关更多信息，请参阅 AWS 数据库加密 SDK 的工作原理。

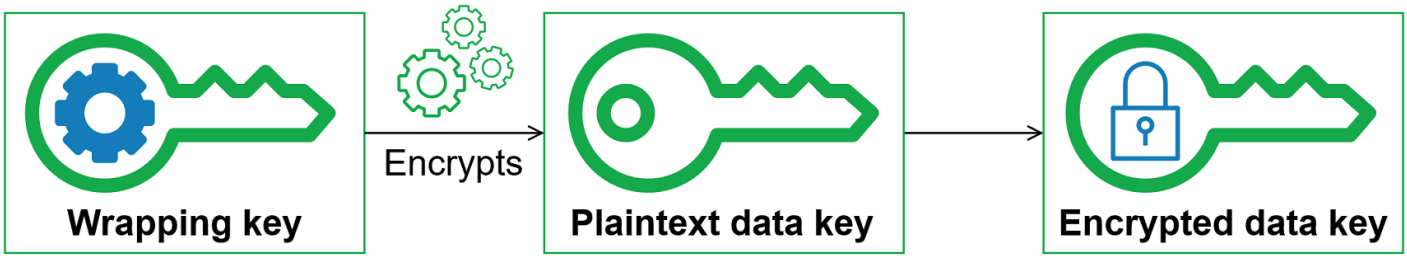
> **Tip**
>
> 在 AWS 数据库加密 SDK 中，我们将数据密钥与数据加密密钥区分开来。作为最佳实践，支持的所有算法套件都使用密钥派生函数。密钥派生函数将数据密钥作为输入，并返回实际用于加

密记录的数据加密密钥。因此，我们通常说数据是“根据”数据密钥加密的，而不是“由”数据密钥加密的。

每个加密的数据密钥都包含元数据，包括对其进行加密的包装密钥的标识符。此元数据使 AWS 数据库加密 SDK 能够在解密时识别有效的包装密钥。

包装密钥

包装密钥是一种密钥加密密钥， AWS 数据库加密 SDK 使用它来加密用于加密记录的数据密钥。可以使用一个或多个包装密钥加密每个数据密钥。在配置密钥环时，您可以决定使用哪些包装密钥来保护您的数据。



AWS 数据库加密 SDK 支持多种常用的封装密钥，例如 AWS Key Management Service(AWS KMS) 对称加密 KMS 密钥（包括多区域密钥）和非对称 RSA KMS AWS KMS 密钥、原始 AES-GCM（高级加密 Standard/Galois 计数器模式）密钥和原始 RSA 密钥。建议尽量使用 KMS 密钥。要决定应当使用哪个包装密钥，请参阅选择包装密钥。

在使用信封加密时，您需要保护包装密钥以防止未经授权的访问。您可以通过以下任何方式来执行此操作：

- 使用专用于该用途的服务，如 AWS Key Management Service （AWS KMS）。
- 使用硬件安全模块 (HSM)，例如，AWS CloudHSM 提供的模块。
- 使用其他密钥管理工具和服务。

如果您没有密钥管理系统，我们建议您使用 AWS KMS。 AWS 数据库加密 SDK AWS KMS 与集成，可帮助您保护和使用包装密钥。

密钥环

要指定用于加密和解密的包装密钥，您可以使用密钥环。您可以使用 AWS 数据库加密 SDK 提供的密钥环，也可以设计自己的实现。

密钥环 生成、加密和解密数据密钥。它还会生成用于计算签名中基于哈希的消息身份验证码 (HMACs) 的 MAC 密钥。定义密钥环时，您可以指定用于加密数据密钥的包装密钥。大多数密钥环至少指定一个包装密钥或一项提供和保护包装密钥的服务。加密时， AWS 数据库加密 SDK 使用密钥环中指定的所有包装密钥来加密数据密钥。有关选择和使用 AWS 数据库加密 SDK 定义的密钥环的帮助，请参阅使用密钥环。

加密操作

加密操作告诉加密程序要对记录中的每个字段执行哪些操作。

加密操作值可以是以下任何值：

- 加密并签名 – 加密字段。在签名中包含加密的字段。
- 仅签名 – 在签名中包含该字段。
- 在加密上下文中签名并包含-将该字段包含在签名和加密上下文中。

  默认情况下，分区和排序密钥是加密上下文中唯一包含的属性。您可以考虑定义其他字段，`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`以便分AWS KMS 层密钥环的分支密钥 ID 提供者可以识别从加密上下文中解密需要哪个分支密钥。有关更多信息，请参阅分支密钥 ID 供应商。

  > **Note**
  >
  > 要使用`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`加密操作，必须使用 AWS 数据库加密 SDK 的 3.3 或更高版本。在更新要包含的数据模型之前，先将新版本部署给所有读者`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。

- 不执行任何操作 – 不加密或在签名中包含该字段。

对于可能存储敏感数据的任何字段，请使用加密和签名。对于主键值（例如，DynamoDB 表中的分区键和排序键），请使用仅签名或签名并包含在加密上下文中。如果您指定了任何 “签名” 并包含在加密上下文中，则分区和排序属性也必须为 “签名” 并包含在加密上下文中。您不需要为材料描述指定加密操作。 AWS 数据库加密 SDK 会自动对存储材料描述的字段进行签名。

请谨慎选择您的加密操作。如有怀疑，请使用 Encrypt and sign (加密和签名)。使用 AWS 数据库加密 SDK 保护记录后，不能将现有`ENCRYPT_AND_SIGN`、`SIGN_ONLY`、或`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`字段更改为现有字段`DO_NOTHING`，也不能更改

分配给现有DO_NOTHING字段的加密操作。但是，您仍可以对数据模型进行其他更改。例如，您可以在单个部署中添加或移除加密字段。

材料描述

材料描述将用作加密记录的标题。当您使用 AWS 数据库加密 SDK 对字段进行加密和签名时，加密器会在组装加密材料时记录材料描述，并将材料描述存储在加密器添加到记录中的新字段 (aws_dbe_head) 中。

材料描述是一种可移植的格式化数据结构，其中包含数据密钥和其他信息的加密副本，例如加密算法、加密上下文以及加密和签名指令。加密程序将在汇编用于加密和签名的加密材料时记录材料描述。之后，当它需要汇编加密材料以验证和解密字段时，它将使用材料描述作为指南。

将加密的数据密钥以及加密的字段存储在一起可以简化解密操作，您不必将加密的数据密钥独立于它们加密的数据进行存储和管理。

有关材料描述的技术信息，请参阅 材料描述的格式。

加密上下文

为了提高加密操作的安全性， AWS 数据库加密 SDK 在所有加密和签署记录的请求中都包含加密上下文。

加密上下文 是一组名称值对，其中包含任意非机密经过身份验证的附加数据。 AWS 数据库加密 SDK 在加密环境中包含数据库的逻辑名称和主键值（例如，DynamoDB 表中的分区键和排序键）。当您加密和签名字段时，加密上下文以加密方式绑定到加密的记录，以便需要使用相同的加密上下文解密字段。

如果您使用 AWS KMS 密钥环，则 AWS 数据库加密 SDK 还会使用加密上下文在密钥环的调用中提供其他经过身份验证的数据 (AAD)。 AWS KMS

每当您使用默认算法套件时，加密材料管理程序（CMM）都会向加密上下文（由保留名称、aws-crypto-public-key 和表示公有验证密钥的值组成）添加一个名称/值对。公有验证密钥存储在材料描述中。

加密材料管理器

加密材料管理器（CMM）汇编用于加密、解密和对数据进行签名的加密材料。每当您使用默认算法套件时，加密材料都会包括明文和加密的数据密钥、对称签名密钥以及非对称签名密钥。您永远不会直接与 CMM 交互。加密和解密方法替您进行处理。

由于 CMM 充当 AWS 数据库加密 SDK 和密钥环之间的联络人，因此它是自定义和扩展（例如支持策略实施）的理想场所。您可以明确指定 CMM，但这不是必需的。当您指定密钥环时， AWS 数据库加密 SDK 会为您创建一个默认 CMM。默认 CMM 从您指定的密钥环获取加密或解密材料。这可能涉及调用一个加密服务，如 AWS Key Management Service (AWS KMS)。

对称和非对称加密

对称加密使用相同的密钥来加密和解密数据。

非对称加密使用数学相关的数据密钥对。密钥对中的一个密钥对数据进行加密；只有密钥对中的另一个密钥可以解密数据。

AWS 数据库加密 SDK 使用信封加密。使用对称数据密钥加密您的数据。使用一个或多个对称或非对称包装密钥加密对称数据密钥。它在记录中添加了材料描述，其中至少包括一个数据密钥的加密副本。

加密您的数据（对称加密）

为了加密您的数据， AWS 数据库加密 SDK 使用对称数据密钥和包含对称加密算法的算法套件。要解密数据， AWS 数据库加密 SDK 使用相同的数据密钥和相同的算法套件。

加密您的数据密钥（对称或非对称加密）

您为加密和解密操作提供的密钥环决定了对称数据密钥的加密和解密方式。您可以选择使用对称加密的密钥环，例如带有对称加密 KMS AWS KMS 密钥的密钥环，也可以选择使用非对称加密的密钥环，例如带有非对称 RSA KMS 密钥的 AWS KMS 密钥环。

密钥承诺

AWS 数据库加密 SDK 支持密钥承诺（有时称为稳健性），这是一种安全属性，可确保每个密文只能解密为单个纯文本。为此，密钥承诺确保仅使用加密记录的数据密钥来解密消息。 AWS 数据库加密 SDK 包括所有加密和解密操作的密钥承诺。

大多数现代对称密码（包括 AES）使用单个密钥对纯文本进行加密，例如 AWS 数据库加密 SDK 用来加密记录中标记的每个纯文本字段的唯一数据密钥。ENCRYPT_AND_SIGN使用相同的数据密钥解密这些记录会返回与原始数据相同的明文。使用不同的密钥解密通常会失败。虽然困难，但在技术上有可能使用两个不同的密钥解密加密文字。在极少数情况下，找到一个可以将加密文字不分解密成不同但仍然可以理解的明文的密钥是可行的。

AWS 数据库加密 SDK 始终使用一个唯一的数据密钥对每个属性进行加密。可能会使用多个包装密钥加密该数据密钥，但包装密钥始终加密相同的数据密钥。尽管如此，手动制作的复杂加密的记录实际上

可能包含不同的数据密钥，每个数据密钥都由不同的包装密钥加密。例如，如果一个用户对加密的记录进行解密，将返回 0x0（false），而另一个用户解密相同的加密记录则得到 0x1（true）。

为防止出现这种情况， AWS 数据库加密 SDK 在加密和解密时包含密钥承诺。加密方法以加密方式将生成加密文字的唯一数据密钥与密钥承诺绑定，密钥承诺是一种 HMAC 散列消息认证码（HMAC），使用数据密钥的推导根据材料描述计算得出。然后它将密钥承诺存储在材料描述中。当使用密钥承诺解密记录时， AWS 数据库加密 SDK 会验证数据密钥是否是该加密记录的唯一密钥。如果数据密钥验证失败，则解密操作将失败。

数字签名

AWS 数据库加密 SDK 使用经过身份验证的加密算法 AES-GCM 对您的数据进行加密，解密过程无需使用数字签名即可验证加密消息的完整性和真实性。但是，由于 AES-GCM 使用对称密钥，所以能够解密用于解密加密文字的数据密钥的任何人员都可以手动创建新的加密的加密文字，从而造成潜在的安全问题。例如，如果您使用 AWS KMS key 作为包装密钥，则具有`kms:Decrypt`权限的用户无需调用即可创建加密的密文。`kms:Encrypt`

为避免此问题，默认算法套件将椭圆曲线数字签名算法（ECDSA）签名添加到加密记录中。默认算法套件使用经过身份验证的加密算法 AES-GCM 对记录中标记为 `ENCRYPT_AND_SIGN` 的字段进行加密。然后，它会计算记录中标`ENCRYPT_AND_SIGN`有、和的字段上的基于哈希的消息身份验证码 (HMACs) 和非对称 ECDSA 签名。`SIGN_ONLY` `SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`解密过程使用签名来验证授权用户是否对记录进行了加密。

使用默认算法套件时， AWS 数据库加密 SDK 会为每条加密记录生成临时私钥和公钥对。 AWS 数据库加密 SDK 将公钥存储在材料描述中，并丢弃私钥。这样可以确保任何人都无法创建另一个使用公钥进行验证的签名。该算法将公钥与加密的数据密钥绑定为材料描述中的其他经过身份验证的数据，从而防止只能解密字段的用户更改公钥或影响签名验证。

AWS 数据库加密 SDK 始终包含 HMAC 验证。默认情况下，ECDSA 数字签名启用，但不是必需的。如果加密数据的用户和解密数据的用户同样受到信任，您可能会考虑使用不包括数字签名的算法条件以改进性能。有关选择替代算法套件的更多信息，参阅选择算法套件。

> **Note**
>
> 如果密钥环未在加密器和解密器之间划清界限，则数字签名不提供任何加密价值。

AWS KMS 密钥环（包括非对称 RSA AWS KMS 密钥环）可以根据密钥策略和 IAM 策略在加密器和解密器之间进行划分。 AWS KMS

由于其加密性质，以下密钥环无法在加密器和解密器之间进行划分：

- AWS KMS 分层钥匙圈
- AWS KMS ECDH 钥匙圈
- 原始 AES 密钥环
- 原始 RSA 密钥环
- 未加工的 ECDH 钥匙圈

AWS 数据库加密 SDK 的工作原理

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

AWS 数据库加密 SDK 提供专为保护存储在数据库中的数据而设计的客户端加密库。库包含可以直接扩展或使用的安全实施。有关定义和使用自定义组件的更多信息，请参阅数据库实现的 GitHub 存储库。

本节中的工作流程说明了 AWS 数据库加密 SDK 如何对数据库中的数据进行加密、签名、解密和验证。这些工作流使用抽象元素和默认特征描述基本流程。有关 AWS 数据库加密 SDK 如何与您的数据库实现配合使用的详细信息，请参阅数据库的加密内容主题。

AWS 数据库加密 SDK 使用信封加密来保护您的数据。每条记录都使用唯一的数据密钥进行加密。数据密钥用于为加密操作中标记为 `ENCRYPT_AND_SIGN` 的每个字段派生唯一的数据加密密钥。然后，使用您指定的包装密钥对数据密钥副本进行加密。要解密加密记录，AWS 数据库加密 SDK 使用您指定的包装密钥来解密至少一个加密的数据密钥。然后其可解密加密文字并返回一条明文条目。

有关 AWS 数据库加密 SDK 中使用的术语的更多信息，请参阅AWS 数据库加密 SDK 概念。

加密并签名

AWS 数据库加密 SDK 的核心是一个记录加密器，用于对数据库中的记录进行加密、签名、验证和解密。它取得您的记录的信息，以及要加密和签名的字段说明。它将从通过您指定的包装密钥配置的加密材料提供程序获取加密材料和加密材料的使用说明。

以下演练描述了 AWS 数据库加密 SDK 如何对您的数据条目进行加密和签名。

1. 加密材料管理器为 AWS 数据库加密 SDK 提供了唯一的数据加密密钥：一个纯文本数据密钥、一份由指定包装密钥加密的数据密钥副本和一个 MAC 密钥。

   > **Note**
   >
   > 您可以使用多个包装密钥加密数据密钥。每个包装密钥加密数据密钥的单独副本。 AWS 数据库加密 SDK 将所有加密的数据密钥存储在材料描述中。 AWS 数据库加密 SDK 在记录中添加一个用于存储材料描述的新字段（aws_dbe_head）。
   > 为数据密钥的每个加密副本派生一个 MAC 密钥。MAC 密钥不存储在材料描述中。反之，解密方法使用包装密钥再次派生 MAC 密钥。

2. 加密方法对您指定的加密操作中标记为 ENCRYPT_AND_SIGN 的每个字段进行加密。
3. 加密方法从数据密钥中派生 commitKey，并使用它生成密钥承诺值，然后再丢弃该数据密钥。
4. 加密方法将材料描述添加到记录中。材料描述包含加密的数据密钥以及有关加密记录的其他信息。有关材料描述中包含的信息的完整列表，请参阅材料描述格式。
5. 加密方法使用步骤 1 中返回的 MAC 密钥来计算基于哈希的消息身份验证码 (HMAC) 值，而不是材料描述、加密上下文以及加密操作中标记ENCRYPT_AND_SIGN的每个字段的规范化。SIGN_ONLY SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXTHMAC 值存储在加密方法添加到记录的新字段（aws_dbe_foot）中。
6. 加密方法根据材料描述、加密上下文和每个标有 ENCRYPT_AND_SIGN “或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT” 的字段的规范化计算 ECDSA 签名，并将 ECDSA 签名存储在字段中。SIGN_ONLY aws_dbe_foot

   > **Note**
   >
   > 默认情况下，ECDSA 签名处于启用状态，但不是必需的。

7. 加密方法将已加密和签名的记录存储在您的数据库中

解密并验证

1. 加密材料管理器（CMM）提供解密方法，其中解密材料存储在材料描述中，包括明本数据密钥和关联的 MAC 密钥。
   - CMM 使用指定的密钥环中的包装密钥为加密的数据密钥解密，然后返回明文数据密钥。
2. 解密方法比较并验证材料描述中的密钥承诺值。

3. 解密方法可验证签名字段中的签名。

   它SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT从您定义的允许未经身份验证的字段列表中识别哪些字段已标记ENCRYPT_AND_SIGNSIGN_ONLY、或。解密方法使用步骤 1 中返回的 MAC 密钥重新计算和比较标记为、或的字段的 HMAC 值。ENCRYPT_AND_SIGN SIGN_ONLY SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT然后，它使用存储在加密上下文中的公有密钥来验证 ECDSA 签名。
4. 解密方法使用明文数据密钥解密标记为 ENCRYPT_AND_SIGN 的每个值。然后，AWS 数据库加密 SDK 会丢弃纯文本数据密钥。
5. 解密方法返回明文记录。

AWS 数据库加密 SDK 中支持的算法套件

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

算法套件 是一组加密算法和相关的值。密码系统使用算法实现来生成密文。

AWS 数据库加密 SDK 使用算法套件对数据库中的字段进行加密和签名。所有支持的算法套件都使用带 Galois/Counter 模式 (GCM) 的高级加密标准 (AES) 算法（称为 AES-GCM）来加密原始数据。AWS 数据库加密 SDK 支持 256 位加密密钥。身份验证标签长度始终为 16 字节。

AWS 数据库加密 SDK 算法套件

算法	加密算法	数据密钥长度（位）	密钥派生算法	对称签名算法	非对称签名算法	密钥承诺
默认	AES-GCM	256	HKDF 以及 SHA-512	HMAC-SHA-384	ECDSA 以及 P-384 和 SHA-384	HKDF 以及 SHA-512
没有 ECDSA 数字签名的 AES-GCM	AES-GCM	256	HKDF 以及 SHA-512	HMAC-SHA-384	无	HKDF 以及 SHA-512

加密算法

与加密算法一起使用的名称和模式。 AWS 数据库加密 SDK 中的算法套件使用带 Galois/Counter 模式 (GCM) 的高级加密标准 (AES) 算法。

数据密钥长度

数据密钥的长度（以位为单位）。 AWS 数据库加密 SDK 支持 256 位数据密钥。数据密钥用作基于 HMAC 的密 extract-and-expand钥派生函数 (HKDF) 的输入。HKDF 的输出用作加密算法中的数据加密密钥。

密钥派生算法

基于 HMAC 的 extract-and-expand密钥派生函数 (HKDF)，用于派生数据加密密钥。 AWS 数据库加密 SDK 使用 RFC 5869 中定义的 HKDF。

- 使用的哈希函数是 SHA-512
- 对于提取步骤：
  - 不使用加密盐。根据 RFC，加密盐设置为包含零的字符串。
  - 输入密钥材料是密钥环中的数据密钥。
- 对于扩展步骤：
  - 输入伪随机密钥是提取步骤的输出。
  - 密钥标签是按大端字节顺序排列的 DERIVEKEY 字符串的 UTF-8 编码字节。
  - 输入信息是将算法 ID 和密钥标签（按此顺序）串联在一起的结果。
  - 输出加密材料的长度是数据密钥长度。该输出用作加密算法中的数据加密密钥。

对称签名算法

用于生成对称签名的基于哈希的消息身份验证码 (HMAC) 算法。所有支持的算法套件都包含 HMAC 验证。

AWS 数据库加密 SDK 对材料描述和所有标有ENCRYPT_AND_SIGNSIGN_ONLY、或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT的字段进行序列化。然后，它使用带有加密哈希函数算法 (SHA-384) 的 HMAC 对规范化进行签名。

对称 HMAC 签名存储在 AWS 数据库加密 SDK 添加到记录中的新字段 (aws_dbe_foot) 中。

非对称签名算法

用于生成非对称数字签名的签名算法。

AWS 数据库加密 SDK 对材料描述和所有标有ENCRYPT_AND_SIGNSIGN_ONLY、或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT的字段进行序列化。然后，它使用具有以下细节的椭圆曲线数字签名算法 (ECDSA) 对规范化进行签名：

- 使用的椭圆曲线是 P-384，定义见数字签名标准 (DSS) (FIPS PUB 186-4)。
- 使用的哈希函数是 SHA-384。

非对称 ECDSA 签名与现场对称 HMAC 签名一起存储。aws_dbe_foot

默认情况下包括 ECDSA 数字签名，但不是必需的。

密钥承诺

基于 HMAC 的 extract-and-expand密钥派生函数 (HKDF) 用于派生提交密钥。

- 使用的哈希函数是 SHA-512
- 对于提取步骤：
  - 不使用加密盐。根据 RFC，加密盐设置为包含零的字符串。
  - 输入密钥材料是密钥环中的数据密钥。
- 对于扩展步骤：
  - 输入伪随机密钥是提取步骤的输出。
  - 输入信息是按大字节顺序排列的COMMITKEY字符串的 UTF-8 编码字节。
  - 输出键控材料的长度为 256 位。此输出用作提交密钥。

提交密钥计算记录承诺，即不同的 256 位基于哈希的消息身份验证码 (HMAC) 哈希，而不是材料描述。有关向算法套件添加密钥承诺的技术说明，请参阅 Cryptology ePrint Archive AEADs 中的密钥提交。

默认的算法套件

默认情况下，AWS 数据库加密 SDK 使用带有 AES-GCM、基于 HMAC 的 extract-and-expand密钥派生函数 (HKDF)、HMAC 验证、ECDSA 数字签名、密钥承诺和 256 位加密密钥的算法套件。

默认算法套件包括 HMAC 验证（对称签名）和 ECDSA 数字签名（非对称签名）。这些签名存储在 AWS 数据库加密 SDK 添加到记录中的新字段 (aws_dbe_foot) 中。当授权策略允许一组用户加密数据，允许另一组用户解密数据时，ECDSA 数字签名特别有用。

默认算法套件还会派生一个密钥承诺 ——一个将数据密钥与记录关联的 HMAC 哈希。密钥承诺值是根据材料描述和提交密钥计算得出的 HMAC。然后，密钥承诺值将存储在材料描述中。密钥承诺确保每

个加密文字仅解密为一个明文。这些算法套件通过验证用作加密算法输入的数据密钥达到上述目的。加密时，算法套件会派生密钥承诺 HMAC。在解密之前，这些算法套件会验证数据密钥是否生成相同的密钥承诺 HMAC。如果没有，Decrypt 调用会失败。

没有 ECDSA 数字签名的 AES-GCM

尽管默认算法套件可能适用于大多数应用程序，但您可以选择其他算法套件。例如，没有ECDSA数字签名的算法套件可以满足某些信任模型。仅当加密数据的用户和解密数据的用户同样受到信任时，才使用此套件。

所有 AWS 数据库加密 SDK 算法套件都包含 HMAC 验证（对称签名）。唯一的区别是，没有ECDSA数字签名的AES-GCM算法套件缺少提供额外真实性和不可否认性的非对称签名。

例如，如果您的密钥环、、和中有多个包装密钥 `wrappingKeyAwrappingKeyB`，并且您使用`wrappingKeyA`解密记录`wrappingKeyC`，则 HMAC 对称签名会验证该记录是否由有权访问的用户加密。`wrappingKeyA`如果您使用默认算法套件，则会 HMACs提供相同的验证`wrappingKeyA`，并使用 ECDSA 数字签名来确保记录由具有加密权限的用户加密。`wrappingKeyA`

要选择不带数字签名的 AES-GCM 算法套件，请在加密配置中加入以下片段。

Java

以下代码段指定了没有 ECDSA 数字签名的 AES-GCM 算法套件。有关更多信息，请参阅 the section called “加密配置”。

```
.algorithmSuiteId(
    DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384)
```

C# / .NET

以下代码段指定了没有 ECDSA 数字签名的 AES-GCM 算法套件。有关更多信息，请参阅 the section called “加密配置”。

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

Rust

以下代码段指定了没有 ECDSA 数字签名的 AES-GCM 算法套件。有关更多信息，请参阅 the section called “加密配置”。

```
.algorithm_suite_id(
    DbeAlgorithmSuiteId::AlgAes256GcmHkdfSha512CommitKeyEcdsaP384SymsigHmacSha384,
)
```

将 AWS 数据库加密 SDK 与 AWS KMS

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

要使用 AWS 数据库加密 SDK，您需要配置密钥环并指定一个或多个包装密钥。如果您没有密钥基础设施，我们建议使用 AWS Key Management Service (AWS KMS)。

AWS 数据库加密 SDK 支持两种类型的 AWS KMS 密钥环。传统的 AWS KMS 密钥环使用 AWS KMS keys 生成、加密和解密数据密钥。您可以使用对称加密（SYMMETRIC_DEFAULT）或非对称 RSA KMS 密钥。由于 AWS 数据库加密 SDK 使用唯一的数据密钥对每条记录进行加密和签名，因此每次加密和解密操作都必须调 AWS KMS 用密 AWS KMS 钥环。对于需要最大限度地减少调用次数的应用程序 AWS KMS， AWS 数据库加密 SDK 还支持AWS KMS 分层密钥环。分层密钥环是一种加密材料缓存解决方案，它使用 AWS KMS 保存在Amazon DynamoDB表中的受保护分支密钥，然后在本地缓存用于加密和解密操作的分支密钥材料，从而减少 AWS KMS 调用次数。我们建议尽可能使用 AWS KMS 钥匙圈。

要与之交互 AWS KMS， AWS 数据库加密 SDK 需要使用以下 AWS KMS 模块 适用于 Java 的 AWS SDK。

准备将 AWS 数据库加密 SDK 与 AWS KMS

1. 创建一个 AWS 账户. 要了解如何操作，请参阅如何创建和激活新的亚马逊 Web Services 账户？在 AWS 知识中心中。
2. 创建对称加密 AWS KMS key。有关帮助信息，请参阅《AWS Key Management Service 开发人员指南》中的创建密钥。

   > **Tip**
   >
   > 要 AWS KMS key 以编程方式使用，您需要的 Amazon 资源名称 (ARN)。 AWS KMS key 要获得有关查找 AWS KMS key的 ARN 的帮助，请参阅《AWS Key Management Service 开发人员指南》中的查找密钥 ID 和 ARN。

3. 生成访问密钥 ID 和安全访问密钥。您可以使用 IAM 用户的访问密钥 ID 和私有访问密钥，也可以使用使用临时安全证书（包括访问密钥 ID、私有访问密钥和会话令牌）创建新会话。 AWS

Security Token Service 作为安全最佳实践，我们建议您使用临时证书，而不是与您的 IAM 用户或 AWS （根）用户账户关联的长期证书。

要创建具有访问密钥的 IAM 用户，请参阅《IAM 用户指南》中的创建 IAM 用户。

要生成临时安全凭证，请参阅《IAM 用户指南》中的请求临时安全凭证。

4. 使用中的说明适用于 Java 的 AWS SDK以及您在步骤 3 中生成的访问密钥 ID 和私有访问密钥来设置您的 AWS 证书。如果您生成了临时凭证，还需要指定会话令牌。

   此过程 AWS SDKs 允许您签署对 AWS 的请求。与之交互的 AWS 数据库加密 SDK 中的代码示例 AWS KMS 假设您已完成此步骤。

配置 AWS 数据库加密 SDK

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

AWS 数据库加密 SDK 的设计非常易于使用。尽管 AWS 数据库加密 SDK 有多个配置选项，但默认值是经过精心选择的，以便对大多数应用程序既实用又安全。但是，您可能需要调整配置以提高性能或在设计中加入自定义功能。

主题

选择编程语言

适用于 DynamoDB 的 AWS 数据库加密 SDK 有多种编程语言版本。语言实现旨在实现完全互操作并提供相同的功能，尽管这些功能可能以不同的方式实现。通常，您使用与您的应用程序兼容的库。

选择包装密钥

AWS 数据库加密 SDK 生成一个唯一的对称数据密钥来加密每个字段。您不需要配置、管理或使用数据密钥。 AWS 数据库加密 SDK 可以为你做这件事。

但是，必须选择一个或多个包装密钥来加密每个数据密钥。 AWS 数据库加密 SDK 支持 AWS Key Management Service（AWS KMS）对称加密 KMS 密钥和非对称 RSA KMS 密钥。它还支持您提供的不同大小的 AES 对称密钥和 RSA 非对称密钥。您应对包装密钥的安全性和耐用性负责，因此我们建议您在硬件安全模块或密钥基础设施服务（例如）中使用加密密钥 AWS KMS。

要指定用于加密和解密的包装密钥，您可以使用密钥环。根据您使用的密钥环类型，您可以指定一个包装密钥或多个相同或不同类型的包装密钥。如果您使用多个包装密钥来包装一个数据密钥，则每个包装密钥将加密同一数据密钥的副本。加密的数据密钥（每个包装密钥一个）与加密的字段一起存储在材

料描述中。要解密数据， AWS 数据库加密 SDK 必须首先使用您的一个包装密钥来解密加密的数据密钥。

我们建议尽可能使用其中一个 AWS KMS 钥匙圈。 AWS 数据库加密 SDK 提供了AWS KMS 密钥环和AWS KMS 分层密钥环，这减少了对的调用次数。 AWS KMS要在密钥环 AWS KMS key 中指定，请使用支持的 AWS KMS 密钥标识符。如果使用 AWS KMS 分层密钥环，则必须指定密钥 ARN。有关密钥的密钥标识符的 AWS KMS 详细信息，请参阅《AWS Key Management Service 开发人员指南》中的密钥标识符。

- 使用密 AWS KMS 钥环加密时，您可以为对称加密 KMS 密钥指定任何有效的密钥标识符（密钥 ARN、别名、别名 ARN 或密钥 ID）。如果使用非对称 RSA KMS 密钥，则必须指定密钥 ARN。

  如果您在加密时为 KMS 密钥指定别名名称或别名 ARN，则 AWS 数据库加密 SDK 会保存当前与该别名关联的密钥 ARN；但不会保存别名。对别名的更改不会影响用于解密数据密钥的 KMS 密钥。
- 默认情况下，密 AWS KMS 钥环在严格模式（您指定特定的 KMS 密钥）下解密记录。您必须使用密钥 ARN 标识 AWS KMS keys 以进行解密。

  使用密 AWS KMS 钥环加密时， AWS 数据库加密 SDK 会将的密钥 ARN 与加密的数据密钥一起存储在材料描述中。 AWS KMS key 在严格模式下解密时， AWS 数据库加密 SDK 在尝试使用包装密钥解密加密的数据密钥之前，会验证密钥环中是否出现相同的密钥 ARN。如果您使用不同的密钥标识符，即使标识符引用相同的密钥 AWS KMS key， AWS 数据库加密 SDK 也无法识别或使用。
- 在发现模式下解密时，不需指定任何包装密钥。首先， AWS 数据库加密 SDK 尝试使用存储在材料描述中的密钥 ARN 来解密记录。如果这不起作用，则无论谁拥有或有权访问该 KMS 密钥， AWS 数据库加密 SDK 都会要求 AWS KMS 使用加密记录的 KMS 密钥对记录进行解密。

要将原始 AES 密钥或原始 RSA 密钥对指定为密钥环中的包装密钥，必须指定命名空间和名称。解密时，必须为每个原始包装密钥使用与加密时完全相同的命名空间和名称。如果您使用不同的命名空间或名称，即使密钥材料相同， AWS 数据库加密 SDK 也无法识别或使用包装密钥。

创建发现筛选条件

解密使用 KMS 密钥加密的数据时，最佳实践是在严格模式下解密，也就是说，将包装密钥仅限于您指定的密钥。但是，如有必要，您也可以在发现模式下解密，在这种模式下，您无需指定任何包装密钥。在此模式下， AWS KMS 无论谁拥有或有权访问该 KMS 密钥，都可以使用加密数据密钥的 KMS 密钥对其进行解密。

如果您必须在发现模式下解密，我们建议您始终使用发现过滤器，该过滤器将可用于指定和分区中的 KMS 密钥限制在指定 AWS 账户 和分区中的密钥。发现筛选条件是可选的，但这是最佳实践。

使用下表确定发现筛选条件的分区值。

Region	分区
AWS 区域	`aws`
中国区域	`aws-cn`
AWS GovCloud (US) Regions	`aws-us-gov`

以下示例说明如何创建发现过滤器。在使用代码之前，请将示例值替换为 AWS 账户 和分区的有效值。

Java

```
// Create the discovery filter
DiscoveryFilter discoveryFilter = DiscoveryFilter.builder()
        .partition("aws")
        .accountIds(111122223333)
        .build();
```

C# / .NET

```
var discoveryFilter = new DiscoveryFilter
{
    Partition = "aws",
    AccountIds = 111122223333
};
```

Rust

```
// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .partition("aws")
    .account_ids(111122223333)
    .build()?;
```

使用多租户数据库

借助 AWS 数据库加密 SDK，您可以使用不同的加密材料隔离每个租户，从而为具有共享架构的数据库配置客户端加密。在考虑多租户数据库时，请花点时间查看您的安全要求以及多租户可能会如何影响这些要求。例如，使用多租户数据库可能会影响您将 AWS 数据库加密 SDK 与其他服务器端加密解决方案结合使用的能力。

如果您有多个用户在数据库中执行加密操作，则可以使用其中一个 AWS KMS 密钥环为每个用户提供一个用于其加密操作的不同密钥。管理多租户客户端加密解决方案的数据密钥可能会很复杂。建议尽可能按租户来组织数据。如果租户由主键值（例如，Amazon DynamoDB 表中的分区键）标识，则可以更加轻松地管理密钥。

您可以使用AWS KMS 密钥环使用不同的密钥 AWS KMS 环隔离每个租户，然后. AWS KMS keys根据每个租户的 AWS KMS 呼叫量，您可能需要使用 AWS KMS 分层密钥环来最大限度地减少对的呼叫。AWS KMSAWS KMS 分层密钥环是一种加密材料缓存解决方案，它使用 AWS KMS 保存在Amazon DynamoDB表中的受保护分支密钥，然后在本地缓存用于加密和解密操作的分支密钥材料，从而减少 AWS KMS 调用次数。必须使用 AWS KMS 分层密钥环在数据库中实现可搜索的加密。

创建签名的信标

D AWS atabase Encryption SDK 使用标准信标和复合信标来提供可搜索的加密解决方案，使您无需解密所查询的整个数据库即可搜索加密记录。但是， AWS 数据库加密 SDK 还支持签名信标，这些信标可以完全通过纯文本签名字段进行配置。签名信标是一种复合信标，用于对和SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT字段进行索引SIGN_ONLY和执行复杂查询。

举例来说，如果您有多租户数据库，则可能需要创建一个签名信标，使您能够在数据库中查询由特定租户密钥加密的记录。有关更多信息，请参阅 查询多租户数据库中的信标。

必须使用 AWS KMS 分层密钥环来创建签名的信标。

要配置签名的信标，请提供以下值。

Java

复合信标配置

以下示例在已签名的信标配置中本地定义已签名部件列表。

```
List<CompoundBeacon> compoundBeaconList = new ArrayList<>();
CompoundBeacon exampleCompoundBeacon = CompoundBeacon.builder()
```

```
    .name("compoundBeaconName")
    .split(".")
    .signed(signedPartList)
    .constructors(constructorList)
    .build();
compoundBeaconList.add(exampleCompoundBeacon);
```

信标版本定义

以下示例在信标版本中全局定义了已签名的部件列表。有关定义信标版本的更多信息，请参阅使用信标。

```
 List<BeaconVersion> beaconVersions = new ArrayList<>();
beaconVersions.add(
    BeaconVersion.builder()
        .standardBeacons(standardBeaconList)
        .compoundBeacons(compoundBeaconList)
        .signedParts(signedPartList)
        .version(1) // MUST be 1
        .keyStore(keyStore)
        .keySource(BeaconKeySource.builder()
            .single(SingleKeyStore.builder()
                .keyId(branchKeyId)
                .cacheTTL(6000)
                .build())
            .build())
        .build()
);
```

C# / .NET

查看完整的代码示例：BeaconConfig.cs

签名信标配置

以下示例在已签名的信标配置中本地定义已签名部件列表。

```
var compoundBeaconList = new List<CompoundBeacon>();
var exampleCompoundBeacon = new CompoundBeacon
 {
    Name = "compoundBeaconName",
    Split = ".",
    Signed = signedPartList,
```

```
    Constructors = constructorList
 };
compoundBeaconList.Add(exampleCompoundBeacon);
```

信标版本定义

以下示例在信标版本中全局定义了已签名的部件列表。有关定义信标版本的更多信息，请参阅使用信标。

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = keyStore,
        KeySource = new BeaconKeySource
        {
            Single = new SingleKeyStore
            {
                KeyId = branchKeyId,
                CacheTTL = 6000
            }
        }
    }
};
```

您可以在本地或全局定义的列表中定义已签名的部件。我们建议尽可能在信标版本的全局列表中定义已签名的部件。通过全局定义带符号的部件，您可以定义每个零件一次，然后在多个复合信标配置中重复使用这些部件。如果您只打算使用一次签名部件，则可以在已签名信标配置的本地列表中对其进行定义。你可以在构造函数列表中同时引用局部和全局部分。

如果您在全局范围内定义已签名部件列表，则必须提供构造器部件列表，这些构造器部分标识已签名信标可以在信标配置中组合字段的所有可能方式。

> **Note**
>
> 要全局定义已签名部件列表，必须使用 3.2 版或更高版本的 AWS 数据库加密 SDK。在全局定义任何新部分之前，先将新版本部署给所有读者。

您无法更新现有信标配置以全局定义已签名部件列表。

信标名称

查询信标时使用的名称。

已签名的信标名称不能与未加密的字段相同。任何两个信标的名称都不能相同。

分割字符

用于分隔构成签名信标的各个部分的字符。

分割字符不能出现在构成签名信标的任何字段的明文值中。

签名部分列表

标识已签名信标中包含的签名字段。

每个部分都必须包含名称、来源和前缀。来源是部件标识的SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT字段。来源必须是字段名称或引用嵌套字段值的索引。如果您的部件名称标识了来源，则可以省略来源， AWS 数据库加密 SDK 将自动使用该名称作为其来源。建议尽可能将来源指定为部分名称。前缀可以是任何字符串，但它必须是唯一的。签名信标中任何两个已签名的部分都不能具有相同的前缀。建议使用简短的值，以将该部分与复合信标提供的其他部分区分开来。

我们建议尽可能在全局范围内定义您的签名部件。如果您只打算在一个复合信标中使用签名部件，则可以考虑在本地定义签名部件。本地定义的部件不能与全局定义的部件具有相同的前缀或名称。

Java

```
List<SignedPart> signedPartList = new ArrayList<>);
    SignedPart signedPartExample = SignedPart.builder()
        .name("signedFieldName")
        .prefix("S-")
        .build();
    signedPartList.add(signedPartExample);
```

C# / .NET

```
var signedPartsList = new List<SignedPart>
{
    new SignedPart { Name = "signedFieldName1", Prefix = "S-" },
```

```
    new SignedPart { Name = "signedFieldName2", Prefix = "SF-" }
};
```

构造器列表（可选）

标识定义签名信标汇编签名部分的不同方式的构造函数。

如果您未指定构造函数列表，则 AWS 数据库加密 SDK 将使用以下默认构造函数组装签名的信标。

- 所有已签名的部分均按添加到已签名部分列表的顺序排列
- 所有部分均为必填项

构造函数

每个构造函数都是构造函数部分的有序列表，该列表定义了签名信标的一种汇编方式。构造函数部分按照添加到列表中的顺序连接在一起，其每个部分由指定的分割字符分隔。

每个构造函数部分都会命名签名部分，并定义该部分在构造函数中是必填项还是可选项。例如，如果要在 `Field1`、`Field1.Field2` 和 `Field1.Field2.Field3` 上查询签名信标，请将 `Field2` 和 `Field3` 标记为可选并创建一个构造函数。

每个构造函数必须具有至少一个必需部分。建议将每个构造函数的第一部分设为必填项，这样您就可以在查询中使用 `BEGINS_WITH` 运算符。

如果一个构造函数的所有必需部分都存在于记录中，则该构造函数成功。当您编写一条新记录时，签名信标使用构造函数列表来确定信标是否可以根据提供的值进行汇编。它尝试按照构造函数添加到构造函数列表的顺序汇编信标，并使用第一个成功的构造函数。如果任何构造函数都没有成功，则信标不会写入记录。

所有的读取者和写入者都应指定相同的构造函数顺序，以确保其查询结果正确无误。

使用以下过程指定您自己的构造函数列表。

1. 为每个签名部分创建一个构造函数部分，以定义该部分是否为比填项。

   构造函数部分名称必须是签名字段的名称。

   以下示例演示如何为一个签名字段创建构造函数部分。

   Java

   ```
   ConstructorPart field1ConstructorPart = ConstructorPart.builder()
           .name("Field1")
   ```

```
        .required(true)
        .build();
```

C# / .NET

```
var field1ConstructorPart = new ConstructorPart { Name = "Field1", Required
 = true };
```

2. 使用您在步骤 1 中创建的构造函数部分为汇编签名信标的每种可能方式创建构造函数。

   例如，如果您要查询 `Field1.Field2.Field3` 和 `Field4.Field2.Field3`，则必须创建两个构造函数。`Field1` 和 `Field4` 都可以是必填项，因为它们是在两个单独的构造函数中定义的。

   Java

```
// Create a list for Field1.Field2.Field3 queries
List<ConstructorPart> field123ConstructorPartList = new ArrayList<>();
field123ConstructorPartList.add(field1ConstructorPart);
field123ConstructorPartList.add(field2ConstructorPart);
field123ConstructorPartList.add(field3ConstructorPart);
Constructor field123Constructor = Constructor.builder()
        .parts(field123ConstructorPartList)
        .build();
// Create a list for Field4.Field2.Field1 queries
List<ConstructorPart> field421ConstructorPartList = new ArrayList<>();
field421ConstructorPartList.add(field4ConstructorPart);
field421ConstructorPartList.add(field2ConstructorPart);
field421ConstructorPartList.add(field1ConstructorPart);
Constructor field421Constructor = Constructor.builder()
        .parts(field421ConstructorPartList)
        .build();
```

   C# / .NET

```
// Create a list for Field1.Field2.Field3 queries
 var field123ConstructorPartList = new Constructor
{
    Parts = new List<ConstructorPart> { field1ConstructorPart,
 field2ConstructorPart, field3ConstructorPart }
};
// Create a list for Field4.Field2.Field1 queries
var field421ConstructorPartList = new Constructor
```

```
{
    Parts = new List<ConstructorPart> { field4ConstructorPart,
 field2ConstructorPart, field1ConstructorPart }
};
```

3. 创建一个构造函数列表，其中包含您在步骤 2 中创建的所有构造函数。

   Java

   ```
   List<Constructor> constructorList = new ArrayList<>();
   constructorList.add(field123Constructor)
   constructorList.add(field421Constructor)
   ```

   C# / .NET

   ```
   var constructorList = new List<Constructor>
   {
       field123Constructor,
       field421Constructor
   };
   ```

4. 在创建签名信标时指定 `constructorList`。

AWS 数据库加密 SDK 中的密钥存储

在 AWS 数据库加密软件开发工具包中，密钥存储是一个 Amazon DynamoDB 表，用于保存分层密钥环使用的AWS KMS 分层数据。密钥存储有助于减少使用分层密钥环执行加密操作所需的调用次数。AWS KMS

密钥库保留并管理分支密钥，分层密钥环使用这些密钥来执行信封加密和保护数据加密密钥。密钥库存储活动分支密钥和分支密钥的所有先前版本。活动分支密钥为最新分支密钥版本。分层密钥环对每个加密请求使用唯一的数据加密密钥，并使用从活动分支密钥派生的唯一包装密钥对每个数据加密密钥进行加密。分层密钥环依赖在活动分支密钥及其派生包装密钥之间建立的层次结构。

关键商店术语和概念

Key store (密钥存储)

用于保存分层数据（例如分支密钥和信标密钥）的 DynamoDB 表。

根密钥

对称加密 KMS 密钥，用于生成和保护密钥库中的分支密钥和信标密钥。

分支密钥

一种数据密钥，可重复用于派生用于信封加密的唯一包装密钥。您可以在一个密钥库中创建多个分支密钥，但是每个分支密钥一次只能有一个有效的分支密钥版本。活动分支密钥为最新分支密钥版本。

分支密钥是 AWS KMS keys 通过使用 k ms: GenerateDataKeyWithoutPlaintext 操作派生的。

包装密钥

一种唯一的数据密钥，用于加密操作中使用的数据加密密钥。

包装密钥源自分支密钥。有关密钥派生过程的更多信息，请参阅AWS KMS 分层密钥环技术细节。

数据加密密钥

用于加密操作的数据密钥。分层密钥环对每个加密请求使用唯一的数据加密密钥。

信标钥匙

一种数据密钥，用于生成用于可搜索加密的信标。有关更多信息，请参阅可搜索的加密。

实施最低权限

使用密钥库和 AWS KMS 分层密钥环时，我们建议您通过定义以下角色来遵循最低权限原则：

密钥库管理员

密钥库管理员负责创建和管理密钥库及其持久保存和保护的分支密钥。密钥存储管理员应该是唯一对用作您的密钥存储的 Amazon DynamoDB 表具有写入权限的用户。他们应该是唯一有权访问特权管理员操作（例如`CreateKey`和）的用户`VersionKey`。只有在静态配置密钥库操作时，才能执行这些操作。

`CreateKey`是一项特权操作，可以将新的 KMS 密钥 ARN 添加到您的密钥库许可名单。此 KMS 密钥可以创建新的活动分支密钥。我们建议限制对此操作的访问权限，因为一旦将 KMS 密钥添加到分支密钥存储中，便无法将其删除。

密钥库用户

在大多数用例中，密钥库用户在加密、解密、签名和验证数据时仅通过分层密钥环与密钥库进行交互。因此，他们只需要对用作您的密钥存储库的 Amazon DynamoDB 表具有读取权限。密钥库用户只需要访问使加密操作成为可能的使用操作，例如`GetActiveBranchKey`、`GetBranchKeyVersion`和`GetBeaconKey`。他们不需要权限即可创建或管理他们使用的分支密钥。

当密钥存储操作处于静态配置状态时，或者将密钥存储操作配置为用于发现时，您可以执行使用操作。将密钥存储操作配置为用于发现时，您无法执行管理员操作（`CreateKey`和`VersionKey`）。

如果您的分支密钥存储管理员在您的分支密钥存储中列入了多个 KMS 密钥，我们建议您的密钥存储用户配置其密钥存储操作以进行发现，以便他们的分层密钥环可以使用多个 KMS 密钥。

创建密钥库

在创建分支密钥或使用分AWS KMS 层密钥环之前，必须先创建密钥存储，即管理和保护分支密钥的 Amazon DynamoDB 表。

> **Important**
>
> 请勿删除保留分支密钥的 DynamoDB 表。如果删除此表，则将无法解密使用分层密钥环加密的任何数据。

按照 Amazon DynamoDB 开发者指南中的创建表过程进行操作，使用以下必需的字符串值作为分区键和排序键。

	分区键	排序键
基表	`branch-key-id`	`type`

逻辑密钥库名称

在命名用作密钥存储的 DynamoDB 表时，请务必仔细考虑在配置密钥存储操作时要指定的逻辑密钥存储名称。逻辑密钥库名称充当密钥库的标识符，在第一个用户最初定义后无法更改。在密钥存储操作中，必须始终指定相同的逻辑密钥存储名称。

DynamoDB 表名称和逻辑密钥存储名称之间必须存在 one-to-one映射。为简化 DynamoDB 还原操作，逻辑密钥存储名称以加密方式绑定到表中存储的所有数据。虽然逻辑密钥存储名称可能与您的 DynamoDB 表名称不同，但我们强烈建议将您的 DynamoDB 表名称指定为逻辑密钥存储名称。如果从备份中恢复 DynamoDB 表后您的表名称发生变化，则可以将逻辑密钥存储名称映射到新的 DynamoDB 表名称，以确保分层密钥环仍然可以访问您的密钥存储。

请勿在逻辑密钥存储库名称中包含机密或敏感信息。在 AWS KMS CloudTrail 事件中，逻辑密钥存储库名称以纯文本形式显示为。`tablename`

后续步骤

1. the section called “配置密钥存储操作”
2. the section called “创建分支密钥”
3. 创建 AWS KMS 分层密钥环

配置密钥存储操作

密钥存储操作决定了您的用户可以执行哪些操作，以及他们的 AWS KMS 分层密钥环如何使用您的密钥存储中允许列出的 KMS 密钥。 AWS 数据库加密 SDK 支持以下密钥存储操作配置。

静态

当您静态配置密钥存储时，密钥存储只能使用与您在配置密钥存储操作时提供的 KMS 密钥 ARN 关联的 KMS 密钥。`kmsConfiguration`如果在创建、版本控制或获取分支密钥时遇到不同的 KMS 密钥 ARN，则会引发异常。

您可以在中指定多区域 KMS 密钥`kmsConfiguration`，但该密钥的整个 ARN（包括区域）都保留在从 KMS 密钥派生的分支密钥中。您不能在其他区域指定密钥，必须提供完全相同的多区域密钥才能使值匹配。

静态配置密钥存储操作时，可以执行使用操作 (`GetActiveBranchKey`、`GetBranchKeyVersion`、`GetBeaconKey`) 和管理操作 (`CreateKey`和`VersionKey`)。 `CreateKey`是一项特权操作，可以将新的 KMS 密钥 ARN 添加到您的密钥库许可名单。此 KMS 密钥可以创建新的活动分支密钥。我们建议限制对此操作的访问权限，因为将 KMS 密钥添加到密钥存储库后，便无法将其删除。

Discovery

当您配置密钥库操作以进行发现时，密钥库可以使用密钥库中列入许可名单的任何 AWS KMS key ARN。但是，如果遇到多区域 KMS 密钥，并且该密钥的 ARN 中的区域与正在使用的客户端的区域不匹配，则会引发异常。 AWS KMS

配置密钥库以供发现时，您无法执行管理操作，例如`CreateKey`和`VersionKey`。您只能执行启用加密、解密、签名和验证操作的使用操作。有关更多信息，请参阅 the section called “实施最低权限”。

配置您的关键商店操作

在配置密钥存储操作之前，请确保满足以下先决条件。

- 确定您需要执行哪些操作。有关更多信息，请参阅 the section called “实施最低权限”。
- 选择逻辑密钥存储名称

  DynamoDB 表名称和逻辑密钥存储名称之间必须存在 one-to-one映射。逻辑密钥存储名称以加密方式绑定到表中存储的所有数据，以简化 DynamoDB 还原操作，在第一个用户最初定义后无法对其进行更改。在密钥存储操作中，必须始终指定相同的逻辑密钥存储名称。有关更多信息，请参阅 logical key store name。

静态配置

以下示例静态配置密钥存储操作。您必须指定用作密钥存储的 DynamoDB 表的名称、密钥存储的逻辑名称以及标识对称加密 KMS 密钥的 KMS 密钥 ARN。

> **Note**
>
> 请仔细考虑您在静态配置密钥存储服务时指定的 KMS 密钥 ARN。该`CreateKey`操作将 KMS 密钥 ARN 添加到您的分支密钥存储许可名单中。将 KMS 密钥添加到分支密钥存储库后，便无法将其删除。

Java

```
final KeyStore keystore = KeyStore.builder().KeyStoreConfig(
                  KeyStoreConfig.builder()
                          .ddbClient(DynamoDbClient.create())
                          .ddbTableName(keyStoreName)
                          .logicalKeyStoreName(logicalKeyStoreName)
                          .kmsClient(KmsClient.create())
                          .kmsConfiguration(KMSConfiguration.builder()
                                  .kmsKeyArn(kmsKeyArn)
                                  .build())
                          .build()).build();
```

C# / .NET

```
var kmsConfig = new KMSConfiguration { KmsKeyArn = kmsKeyArn };
  var keystoreConfig = new KeyStoreConfig
  {
      KmsClient = new AmazonKeyManagementServiceClient(),
      KmsConfiguration = kmsConfig,
      DdbTableName = keyStoreName,
      DdbClient = new AmazonDynamoDBClient(),
      LogicalKeyStoreName = logicalKeyStoreName
  };
  var keystore = new KeyStore(keystoreConfig);
```

Rust

```
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
```

```
let key_store_config = KeyStoreConfig::builder()
    .kms_client(aws_sdk_kms::Client::new(&sdk_config))
    .ddb_client(aws_sdk_dynamodb::Client::new(&sdk_config))
    .ddb_table_name(key_store_name)
    .logical_key_store_name(logical_key_store_name)
    .kms_configuration(KmsConfiguration::KmsKeyArn(kms_key_arn.to_string()))
    .build()?;

let keystore = keystore_client::Client::from_conf(key_store_config)?;
```

发现配置

以下示例配置了用于发现的密钥存储操作。您必须指定用作密钥存储的 DynamoDB 表的名称和逻辑密钥存储名称。

Java

```
final KeyStore keystore = KeyStore.builder().KeyStoreConfig(
                KeyStoreConfig.builder()
                        .ddbClient(DynamoDbClient.create())
                        .ddbTableName(keyStoreName)
                        .logicalKeyStoreName(logicalKeyStoreName)
                        .kmsClient(KmsClient.create())
                        .kmsConfiguration(KMSConfiguration.builder()
                                .discovery(Discovery.builder().build())
                                .build())
                        .build()).build();
```

C# / .NET

```
var keystoreConfig = new KeyStoreConfig
 {
     KmsClient = new AmazonKeyManagementServiceClient(),
     KmsConfiguration = new KMSConfiguration {Discovery = new Discovery()},
     DdbTableName = keyStoreName,
     DdbClient = new AmazonDynamoDBClient(),
     LogicalKeyStoreName = logicalKeyStoreName
 };
 var keystore = new KeyStore(keystoreConfig);
```

Rust

```
let key_store_config = KeyStoreConfig::builder()
        .kms_client(kms_client)
        .ddb_client(ddb_client)
        .ddb_table_name(key_store_name)
        .logical_key_store_name(logical_key_store_name)

 .kms_configuration(KmsConfiguration::Discovery(Discovery::builder().build()?))
        .build()?;
```

创建有效的分支密钥

分支密钥是派生自分支密钥的数据密钥 AWS KMS key ， AWS KMS 分层密钥环使用该密钥来减少调用的次数。 AWS KMS活动分支密钥为最新分支密钥版本。分层密钥环为每个加密请求生成唯一的数据密钥，并使用从活动分支密钥派生的唯一包装密钥对每个数据密钥进行加密。

要创建新的活动分支密钥，必须静态配置密钥存储操作。 CreateKey是一项特权操作，用于将密钥库操作配置中指定的 KMS 密钥 ARN 添加到密钥库许可名单中。然后，使用 KMS 密钥生成新的活动分支密钥。我们建议限制对此操作的访问权限，因为将 KMS 密钥添加到密钥存储库后，便无法将其删除。

我们建议通过应用程序控制平面中的 KeyStore 管理界面使用该CreateKey操作。这种方法符合密钥管理的最佳实践。

不要在数据平面中创建分支密钥。这种做法可能导致：

- 拨打不必要的电话 AWS KMS
- AWS KMS 在高并发环境中多次并发调用
- 多次 TransactWriteItems 调用后备的 DynamoDB 表。

该CreateKey操作在TransactWriteItems调用中包括条件检查，以防止覆盖现有的分支密钥。但是，在数据平面中创建密钥仍可能导致资源使用效率低下和潜在的性能问题。

您可以将密钥存储库中的一个 KMS 密钥列入许可名单，也可以通过更新您在密钥存储操作配置中指定的 KMS 密钥 ARN 并再次调用来允许列入多个 KMS 密钥。CreateKey如果您将多个 KMS 密钥列入许可名单，则您的密钥存储用户应配置其密钥存储操作以供发现，以便他们可以使用他们有权访问的密钥存储库中的任何允许列表的密钥。有关更多信息，请参阅 the section called “配置密钥存储操作”。

所需的权限

要创建分支密钥，您需要拥有密钥存储操作中指定的 KMS 密钥的 kms: GenerateDataKeyWithoutPlaintext 和 kms: ReEncrypt 权限。

创建分支密钥

以下操作使用您在密钥存储操作配置中指定的 KMS 密钥创建新的活动分支密钥，并将活动分支密钥添加到用作密钥存储的 DynamoDB 表中。

调用 `CreateKey` 时，您可以选择指定以下可选值。

- `branchKeyIdentifier`：定义自定义 `branch-key-id`。

  要创建自定义 `branch-key-id`，还必须加入包含 `encryptionContext` 参数的其他加密上下文。
- `encryptionContext`: 定义一组可选的非秘密密钥值对，用于在 kms: 调用中包含的加密上下文中提供额外的经过身份验证的数据 (AAD)。GenerateDataKeyWithoutPlaintext

  此额外加密上下文带有 `aws-crypto-ec:` 前缀。

Java

```
final Map<String, String> additionalEncryptionContext =
 Collections.singletonMap("Additional Encryption Context for",
         "custom branch key id");

  final String BranchKey = keystore.CreateKey(
         CreateKeyInput.builder()
                 .branchKeyIdentifier(custom-branch-key-id) //OPTIONAL
                 .encryptionContext(additionalEncryptionContext) //OPTIONAL

                 .build()).branchKeyIdentifier();
```

C# / .NET

```
var additionalEncryptionContext = new Dictionary<string, string>();
  additionalEncryptionContext.Add("Additional Encryption Context for", "custom
 branch key id");

  var branchKeyId = keystore.CreateKey(new CreateKeyInput
  {
      BranchKeyIdentifier = "custom-branch-key-id", // OPTIONAL
```

```
        EncryptionContext = additionalEncryptionContext // OPTIONAL
    });
```

Rust

```
let additional_encryption_context = HashMap::from([
    ("Additional Encryption Context for".to_string(), "custom branch key
 id".to_string())
]);

let branch_key_id = keystore.create_key()
    .branch_key_identifier("custom-branch-key-id") // OPTIONAL
    .encryption_context(additional_encryption_context) // OPTIONAL
    .send()
    .await?
    .branch_key_identifier
    .unwrap();
```

首先，CreateKey 操作生成以下值。

- 适用于 branch-key-id 的版本 4 通用唯一标识（UUID）（除非您指定了自定义 branch-key-id）。
- 适用于分支密钥版本的版本 4 UUID
- timestamp 必须采用协调世界时（UTC）ISO 8601 日期和时间格式。

然后，该CreateKey操作GenerateDataKeyWithoutPlaintext使用以下请求调用 kms:。

```
{
     "EncryptionContext": {
        "branch-key-id" : "branch-key-id",
        "type" : "type",
        "create-time" : "timestamp",
        "logical-key-store-name" : "the logical table name for your key store",
        "kms-arn" : the KMS key ARN,
        "hierarchy-version" : "1",
        "aws-crypto-ec:contextKey": "contextValue"
     },
     "KeyId": "the KMS key ARN you specified in your key store actions",
     "NumberOfBytes": "32"
  }
```

> **Note**
>
> 即使您尚未配置数据库的可搜索加密，CreateKey 操作也会创建活动分支密钥和信标密钥。两个密钥都存储在您的密钥库中。有关更多信息，请参阅使用分层密钥环进行可搜索加密。

接下来，该CreateKey操作调用 km ReEncrypt s:，通过更新加密上下文为分支密钥创建活动记录。

最后，该CreateKey操作调用 ddb: TransactWriteItems 来编写一个新项目，该项目将保留您在步骤 2 中创建的表中的分支密钥。项目具有以下属性。

```
{
      "branch-key-id" : branch-key-id,
      "type" : "branch:ACTIVE",
      "enc" : the branch key returned by the GenerateDataKeyWithoutPlaintext call,
      "version": "branch:version:the branch key version UUID",
      "create-time" : "timestamp",
      "kms-arn" : "the KMS key ARN you specified in Step 1",
      "hierarchy-version" : "1",
      "aws-crypto-ec:contextKey": "contextValue"
 }
```

轮换您的活动分支密钥

每个分支密钥一次仅能有一个活动版本。通常，每个有效的分支密钥版本都用于满足多个请求。但是您可以控制活动分支密钥的重复使用程度，并确定活动分支密钥的轮换频率。

分支密钥不用于加密明文数据密钥。它们用于派生对明文数据密钥进行加密的唯一包装密钥。包装密钥派生过程生成唯一的 32 字节包装密钥，其随机掩码为 28 字节。这意味着，在发生加密损耗之前，分支密钥可以派生出超过 79 万亿或 2^{96} 个唯一的包装密钥。尽管耗尽风险非常低，但由于业务或合同规则或政府法规，您可能需要轮换活动分支密钥。

在您轮换之前，分支密钥的活动版本会一直处于活动状态。以前版本的活动分支密钥不会用于执行加密操作，也不能用于派生新的包装密钥，但仍然可以查询这些密钥并提供包装密钥来解密它们在活动状态下加密的数据密钥。

> **Warning**
>
> 在测试环境中删除分支密钥是不可逆的。您无法恢复已删除的分支密钥。在测试环境中删除并重新创建具有相同 ID 的分支密钥时，可能会出现以下问题：

- 先前测试运行的材料可能仍保留在缓存中
- 某些测试主机或线程可能会使用已删除的分支密钥来加密数据
- 使用已删除分支加密的数据无法解密

要防止集成测试中出现加密失败，请执行以下操作：

- 在创建新的分支密钥之前重置分层密钥环引用或
- IDs 为每个测试使用唯一的分支密钥

所需的权限

要轮换分支密钥，您需要密钥存储操作中指定的 KMS 密钥的 kms: GenerateDataKeyWithoutPlaintext 和 kms: ReEncrypt 权限。

轮换有效的分支密钥

使用该VersionKey操作来轮换您的活动分支密钥。轮换活动分支密钥时，系统会创建新的分支密钥代替先前版本。当您轮换活动分支密钥时，branch-key-id 不会改变。在调用 VersionKey 时，必须指定用于标识当前活动分支密钥的 branch-key-id。

Java

```
keystore.VersionKey(
    VersionKeyInput.builder()
        .branchKeyIdentifier("branch-key-id")
        .build()
);
```

C# / .NET

```
keystore.VersionKey(new VersionKeyInput{BranchKeyIdentifier = branchKeyId});
```

Rust

```
keystore.version_key()
        .branch_key_identifier(branch_key_id)
        .send()
```

```
        .await?;
```

密钥环

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

AWS 数据库加密 SDK 使用密钥环来执行信封加密。密钥环生成、加密和解密数据密钥。密钥环确定保护每条加密记录的唯一数据密钥的来源，以及加密该数据密钥的包装密钥。您在加密时指定一个密钥环，并在解密时指定相同或不同的密钥环。

您可以单独使用每个密钥环，也可以将多个密钥环合并为一个多重密钥环。虽然大多数密钥环可以生成、加密和解密数据密钥，但您也可以创建只执行一项特定操作的密钥环，例如只生成数据密钥的密钥环，并将此密钥环与其他密钥环结合使用。

我们建议您使用可保护包装密钥并在安全边界内执行加密操作的密钥环，例如密 AWS KMS 钥环，它使用永不保密 AWS Key Management Service() AWS KMS keys AWS KMS的密钥环。您还可以编写一个使用封装密钥的密钥环，这些密钥存储在硬件安全模块 (HSMs) 中或受其他主密钥服务保护。

您的密钥环决定了哪些包装密钥保护您的数据密钥并最终保护您的数据。使用最安全且对您的任务实用的包装密钥。尽可能使用由硬件安全模块（HSM）或密钥管理基础设施保护的包装密钥，例如 AWS Key Management Service（AWS KMS）中的 KMS 密钥或 AWS CloudHSM 中的加密密钥。

AWS 数据库加密 SDK 提供了多种密钥环和密钥环配置，您可以创建自己的自定义密钥环。您也可以创建包含一个或多个相同或不同类型的密钥环的多重密钥环。

主题

- 密钥环的工作方式
- AWS KMS 钥匙圈
- AWS KMS 分层钥匙圈
- AWS KMS ECDH 钥匙圈
- 原始 AES 密钥环
- 原始 RSA 密钥环
- 未加工的 ECDH 钥匙圈
- 多重密钥环

密钥环的工作方式

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

在对数据库中的字段进行加密和签名时， AWS 数据库加密 SDK 会要求密钥环提供加密材料。密钥环返回一个明文数据密钥、由密钥环中的每个包装密钥加密的数据密钥副本，以及与数据密钥关联的 MAC 密钥。 AWS 数据库加密 SDK 使用明文密钥对数据进行加密，然后尽快从内存中删除明文数据密钥。然后， AWS 数据库加密 SDK 会添加材料描述，其中包括加密的数据密钥和其他信息，例如加密和签名指令。 AWS 数据库加密 SDK 使用 MAC 密钥计算基于哈希的消息身份验证码 (HMACs)，而不是材料描述和所有标记为或的字段的规范化。`ENCRYPT_AND_SIGN` `SIGN_ONLY`

解密数据时，您可以使用加密数据所用的密钥环，也可以使用其他密钥环。要解密数据，解密密钥环必须有权访问加密密钥环中的至少一个包装密钥。

AWS 数据库加密 SDK 将材料描述中的加密数据密钥传递到密钥环，并要求密钥环解密其中任何一个。密钥环使用其包装密钥以解密一个加密的数据密钥，并返回明文数据密钥。 AWS 数据库加密 SDK 使用明文数据密钥将对数据进行解密。如果密钥环中的所有包装密钥都无法解密任何加密的数据密钥，解密操作将失败。

您可以使用一个密钥环，也可以将相同类型或不同类型的密钥环组合到一个多重密钥环中。当您加密数据时，多重密钥环返回数据密钥的副本，该数据密钥使用构成该多重密钥环的所有密钥环中的所有包装密钥和与数据密钥关联的 MAC 密钥加密。您可以使用包含多重密钥环中任一包装密钥的密钥环解密数据。

AWS KMS 钥匙圈

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

AWS KMS 密钥环使用对称加密或非对称 RSA AWS KMS keys来生成、加密和解密数据密钥。 AWS Key Management Service (AWS KMS) 保护您的 KMS 密钥并在 FIPS 边界内执行加密操作。我们建议您尽可能使用 AWS KMS 密钥环或具有类似安全属性的密钥环。

您还可以在密钥环中使用对称多区域 KMS 密钥。 AWS KMS 有关使用多区域的更多详细信息和示例 AWS KMS keys，请参阅使用多区域 AWS KMS keys。有关多区域密钥的信息，请参阅《AWS Key Management Service 开发人员指南》中的使用多区域密钥。

AWS KMS 钥匙圈可以包括两种类型的包装钥匙：

- 生成器密钥：生成并加密明文数据密钥。加密数据的密钥环必须有一个生成器密钥。
- 其他密钥：加密生成器密钥生成的纯文本数据密钥。 AWS KMS 钥匙圈可以有零个或多个额外的钥匙。

您必须拥有生成器密钥才能加密记录。当 AWS KMS 密钥环只有一个 AWS KMS 密钥时，该密钥用于生成和加密数据密钥。

像所有钥匙圈一样， AWS KMS 钥匙圈可以单独使用，也可以与其他相同或不同类型的钥匙圈一起在多钥匙圈中使用。

主题

AWS KMS 密钥环所需的权限

AWS 数据库加密 SDK 不需要 AWS 账户 ，也不依赖于任何一个 AWS 服务。但是，要使用 AWS KMS 密钥环，您需要对 AWS 账户 密钥环 AWS KMS keys 中的具有以下最低权限。

- 要使用密 AWS KMS 钥环进行加密，您需要生成器密钥的 kms: GenerateDataKey 权限。您需要对密钥环中的所有其他密钥拥有 kms: encrypt 权限。 AWS KMS
- 要使用密钥环进行解密，您需要对密 AWS KMS 钥环中的至少一个密钥具有 kms: Decrypt 权限。 AWS KMS
- 要使用由密钥环组成的多密钥环进行加密，你需要获得生成器密 AWS KMS 钥环中生成器密钥的 kms: GenerateDataKey 权限。你需要对所有其他密钥环中的所有其他密钥具有 kms: encrypt 权限。 AWS KMS

- 要使用非对称 RSA AWS KMS 密钥环进行加密，您不需要 kms: GenerateDataKey 或 kms: Encrypt，因为在创建密钥环时必须指定要用于加密的公钥材料。使用此密钥环加密时不会 AWS KMS 发出任何呼叫。要使用非对称 RSA 密 AWS KMS 钥环进行解密，你需要 kms: Decrypt 权限。

有关权限的详细信息 AWS KMS keys，请参阅《AWS Key Management Service 开发人员指南》中的身份验证和访问控制。

在 AWS KMS 钥匙圈 AWS KMS keys 中识别

一个 AWS KMS 钥匙圈可以包括一个或多 AWS KMS keys个。要在 AWS KMS 密钥环 AWS KMS key 中指定，请使用支持的 AWS KMS 密钥标识符。可用于在密钥环 AWS KMS key 中识别的密钥标识符因操作和语言实现而异。有关 AWS KMS key密钥标识符的详细信息，请参阅《AWS Key Management Service 开发人员指南》中的密钥标识符。

作为最佳实践，请使用最适合您任务的密钥标识符。

- 要使用密 AWS KMS 钥环进行加密，您可以使用密钥 ID、密钥 ARN 、别名或别名 ARN 来加密数据。

> **Note**
>
> 如果您在加密密钥环中为 KMS 密钥指定别名名称或别名 ARN，则加密操作会将当前与该别名关联的密钥 ARN 保存在加密数据密钥的元数据中。它不会保存别名。更改别名不会影响用于解密加密数据密钥的 KMS 密钥。

- 要使用密 AWS KMS 钥环解密，必须使用密钥 ARN 进行识别。 AWS KMS keys有关更多信息，请参阅 选择包装密钥。
- 在用于加密和解密的密钥环中，您必须使用密钥 ARN 以标识 AWS KMS keys。

解密时， AWS 数据库加密 SDK 会在密 AWS KMS 钥环中搜索 AWS KMS key 可以解密其中一个加密数据密钥的。具体而言， AWS 数据库加密 SDK 对材料描述中的每个加密数据密钥使用以下模式。

- AWS 数据库加密 SDK 从材料描述的元数据 AWS KMS key 中获取加密数据密钥的密钥 ARN。
- AWS 数据库加密 SDK 在解密密钥环中搜索密钥匹配的 AWS KMS key ARN。
- 如果在密钥环中找到密钥 ARN 匹配的，则 AWS 数据库加密 SDK 会要求 AWS KMS 使用 KMS 密钥解密加密的数据密钥。 AWS KMS key
- 否则，它跳到下一个加密的数据密钥（如果有）。

创建密 AWS KMS 钥环

您可以为每个 AWS KMS 密钥环配置一个 AWS KMS key 或多个 AWS KMS keys 相同或不同的密钥环 AWS 账户 。 AWS 区域 AWS KMS key 必须是对称加密密钥（SYMMETRIC_DEFAULT）或非对称 RSA KMS 密钥。您也可以使用对称加密多区域 KMS 密钥。您可以在多 AWS KMS密钥环中使用一个或多个钥匙圈。

您可以创建用于加密和解密数据的密 AWS KMS 钥环，也可以创建专门用于加密或解密的 AWS KMS 密钥环。创建用于加密数据的 AWS KMS 密钥环时，必须指定生成器密钥，该密钥用于生成纯文本数据密钥并对其进行加密。 AWS KMS key 数据密钥在数学上与 KMS 密钥无关。然后，如果您愿意，则可以指定用于加密相同纯文本数据密钥的其他 AWS KMS keys 内容。要解密受此密钥环保护的加密字段，您使用的解密密钥环必须至少包含密钥环中 AWS KMS keys 定义的密钥环中的一个，或者不是。AWS KMS keys（没有的 AWS KMS 密钥环 AWS KMS keys 称为AWS KMS 发现密钥环。 ）

加密密钥环或多重密钥环中的所有包装密钥均必须能够加密数据密钥。如有任何包装密钥无法加密，此加密方法将失败。因此，调用方必须拥有密钥环中所有密钥的所需权限。如果您单独或在多重密钥环中使用 Discovery 密钥环加密数据，加密操作将失败。

以下示例使用该`CreateAwsKmsMrkMultiKeyring`方法创建具有对称加密 KMS AWS KMS 密钥的密钥环。该`CreateAwsKmsMrkMultiKeyring`方法会自动创建 AWS KMS 客户端，并确保密钥环能够正确处理单区域和多区域密钥。这些示例使用密钥 ARNs来识别 KMS 密钥。有关详细信息，请参阅 在 AWS KMS 钥匙圈 AWS KMS keys 中识别

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyArn)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createAwsKmsMrkMultiKeyringInput = new CreateAwsKmsMrkMultiKeyringInput
{
    Generator = kmsKeyArn
```

```
};
var awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

Rust

```
let provider_config = MaterialProvidersConfig::builder().build()?;
let mat_prov = client::Client::from_conf(provider_config)?;
let kms_keyring = mat_prov
    .create_aws_kms_mrk_multi_keyring()
    .generator(kms_key_id)
    .send()
    .await?;
```

以下示例使用该CreateAwsKmsRsaKeyring方法创建带有非对称 RSA KMS AWS KMS 密钥的密钥环。要创建非对称 RSA AWS KMS 密钥环，请提供以下值。

- kmsClient: 创建新 AWS KMS 客户端
- kmsKeyID: 用于识别您的非对称 RSA KMS 密钥的密钥 ARN
- publicKey: a 来 ByteBuffer 自 UTF-8 编码的 PEM 文件，该文件代表你传递给的密钥的公钥 kmsKeyID
- encryptionAlgorithm: 加密算法必须是RSAES_OAEP_SHA_256或 RSAES_OAEP_SHA_1

Java

```
 final MaterialProviders matProv = MaterialProviders.builder()
    .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
    .build();
final CreateAwsKmsRsaKeyringInput createAwsKmsRsaKeyringInput =
    CreateAwsKmsRsaKeyringInput.builder()
        .kmsClient(KmsClient.create())
        .kmsKeyId(rsaKMSKeyArn)
        .publicKey(publicKey)
        .encryptionAlgorithm(EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256)
        .build();
IKeyring awsKmsRsaKeyring =
 matProv.CreateAwsKmsRsaKeyring(createAwsKmsRsaKeyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createAwsKmsRsaKeyringInput = new CreateAwsKmsRsaKeyringInput
{
    KmsClient = new AmazonKeyManagementServiceClient(),
    KmsKeyId = rsaKMSKeyArn,
    PublicKey = publicKey,
    EncryptionAlgorithm = EncryptionAlgorithmSpec.RSAES_OAEP_SHA_256
};
IKeyring awsKmsRsaKeyring =
 matProv.CreateAwsKmsRsaKeyring(createAwsKmsRsaKeyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
let sdk_config =
 aws_config::load_defaults(aws_config::BehaviorVersion::latest()).await;
let kms_rsa_keyring = mpl
    .create_aws_kms_rsa_keyring()
    .kms_key_id(rsa_kms_key_arn)
    .public_key(public_key)

 .encryption_algorithm(aws_sdk_kms::types::EncryptionAlgorithmSpec::RsaesOaepSha256)
    .kms_client(aws_sdk_kms::Client::new(&sdk_config))
    .send()
    .await?;
```

使用多区域 AWS KMS keys

您可以在 AWS 数据库加密 SDK 中使用多区域 AWS KMS keys 作为包装密钥。如果您使用多区域密钥合而为一进行加密 AWS 区域，则可以使用其他密钥中的相关多区域密钥进行解密。 AWS 区域

多区域 KMS 密钥是一组 AWS KMS keys AWS 区域 具有相同密钥材料和密钥 ID 的不同密钥。您可以像在不同区域使用相同的密钥一样使用这些相关密钥。多区域密钥支持常见的灾难恢复和备份场景，这些场景要求在一个区域进行加密，并在另一个区域进行解密，而无需进行跨区域调用。 AWS KMS有关多区域密钥的信息，请参阅《AWS Key Management Service 开发人员指南》中的使用多区域密钥。

为了支持多区域密钥， AWS 数据库加密 SDK 包括密 AWS KMS multi-Region-aware 钥环。`CreateAwsKmsMrkMultiKeyring` 方法同时支持单区域密钥和多区域密钥。

- 对于单区域密钥，该 multi-Region-aware符号的行为就像单 AWS KMS 区域密钥环一样。该密钥尝试仅使用加密数据的单区域密钥来解密加密文字。为了简化您的 AWS KMS 密钥环体验，我们建议您在使用对称加密 KMS 密钥时使用该`CreateAwsKmsMrkMultiKeyring`方法。
- 对于多区域密钥，该 multi-Region-aware符号尝试使用加密数据的相同多区域密钥或您指定的区域中的相关多区域密钥来解密密文。

在使用多个 KMS 密 multi-Region-aware钥的密钥环中，您可以指定多个单区域和多区域密钥。但是，您只能在每组相关的多区域密钥中指定一个密钥。如果您使用相同的密钥 ID 指定多个密钥标识符，则构造函数调用将失败。

以下示例使用多区域 KMS AWS KMS 密钥创建密钥环。这些示例将多区域密钥指定为生成器密钥，将单区域密钥指定为子密钥。

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput createAwsKmsMrkMultiKeyringInput =
    CreateAwsKmsMrkMultiKeyringInput.builder()
            .generator(multiRegionKeyArn)
            .kmsKeyIds(Collections.singletonList(kmsKeyArn))
            .build();
IKeyring awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createAwsKmsMrkMultiKeyringInput = new CreateAwsKmsMrkMultiKeyringInput
{
    Generator = multiRegionKeyArn,
    KmsKeyIds = new List<String> { kmsKeyArn }
};
var awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
```

```
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let aws_kms_mrk_multi_keyring = mpl
    .create_aws_kms_mrk_multi_keyring()
    .generator(multiRegion_key_arn)
    .kms_key_ids(vec![key_arn.to_string()])
    .send()
    .await?;
```

使用多区域 AWS KMS 密钥环时，可以在严格模式或发现模式下解密密文。要在严格模式下解密密文，请在解密密文的区域中 multi-Region-aware使用相关多区域密钥的密钥 ARN 来实例化符号。如果您在其他区域（例如，加密记录的区域）中指定相关多区域密钥的密钥 ARN，则该 multi-Region-aware符号将为此进行跨区域调用。 AWS KMS key

在严格模式下解密时， multi-Region-aware符号需要密钥 ARN。仅接受每组相关的多区域密钥中的一个密钥 ARN。

您还可以在发现模式下使用 AWS KMS 多区域密钥进行解密。在发现模式下解密时，不需指定任何 AWS KMS keys。（有关单区域 AWS KMS 发现密钥环的信息，请参阅使用 AWS KMS 发现密钥环。）

如果您使用多区域密钥加密，则发现模式下的 multi-Region-aware符号将尝试使用本地区域中的相关多区域密钥进行解密。如果不存在，则调用失败。在发现模式下， AWS 数据库加密 SDK 不会尝试跨区域调用用于加密的多区域密钥。

使用 AWS KMS 发现密钥环

解密时，最佳做法是指定 AWS 数据库加密 SDK 可以使用的包装密钥。要遵循此最佳实践，请使用 AWS KMS 解密密钥环，将 AWS KMS 封装密钥限制在您指定的密钥范围内。但是，您也可以创建AWS KMS 发现密钥环，即不指定任何包装 AWS KMS 密钥的密钥环。

AWS 数据库加密 SDK 为 AWS KMS 多区域密钥提供了标准 AWS KMS 发现密钥环和发现密钥环。有关多区域密钥与 AWS 数据库加密 SDK 结合使用的信息，请参阅 使用多区域 AWS KMS keys。

由于 Discovery 密钥环未指定任何包装密钥，因此 Discovery 密钥无法加密数据。如果您单独或在多重密钥环中使用 Discovery 密钥环加密数据，加密操作将失败。

解密时，发现密钥环允许 AWS 数据库加密 SDK 要求使用加密数据密钥 AWS KMS 来解密任何加密的数据密钥，无论谁拥有或有权访问该 AWS KMS key 密钥。 AWS KMS key只有在调用方拥有 AWS KMS key的 `kms:Decrypt` 权限时，调用才会成功。

> **Important**
>
> 如果您在解密多密钥环中包含 AWS KMS 发现密钥环，则发现密钥环将覆盖多密钥环中其他密钥环中指定的所有 KMS 密钥限制。多重密钥环的行为类似于限制最少的密钥环。如果您单独或在多重密钥环中使用 Discovery 密钥环加密数据，加密操作将失败

为方便起见， AWS 数据库加密 SDK 提供了 AWS KMS 发现密钥环。不过，出于以下原因，建议尽可能使用更受限制的密钥环。

- 真实性 — AWS KMS 发现密钥环可以使用任何 AWS KMS key 用于加密材料描述中的数据密钥的密钥，前提是调用者有权使用该密钥 AWS KMS key 进行解密。这可能不是呼叫 AWS KMS key 者打算使用的。例如，其中一个加密的数据密钥可能是在任何人都可以 AWS KMS key 使用的安全性较低的情况下加密的。
- 延迟和性能 — AWS KMS 发现密钥环可能比其他密钥环慢得多，因为 AWS 数据库加密 SDK 会尝试解密所有加密的数据密钥，包括由 AWS KMS keys 其他 AWS 账户 和区域加密的数据密钥，而调用者无权使用这些密钥进行解密。 AWS KMS keys

如果您使用发现密钥环，我们建议您使用发现过滤器将可用的 KMS 密钥限制为指定 AWS 账户和分区中的密钥。如需帮助查找您的账户 ID 和分区，请参阅中的您的 AWS 账户 标识符和 ARN 格式。AWS 一般参考

以下代码示例使用发现过滤器实例化 AWS KMS 发现密钥环，该过滤器将 AWS 数据库加密 SDK 可以使用的 KMS 密钥限制为aws分区和111122223333示例账户中的密钥。

在使用此代码之前，请将示例 AWS 账户 和分区值替换为 AWS 账户 和分区的有效值。如果您的 KMS 密钥位于中国区域，请使用 aws-cn 分区值。如果您的 KMS 密钥位于 AWS GovCloud (US) Regions，请使用 aws-us-gov 分区值。对于所有其他 AWS 区域，请使用 aws 分区值。

Java

```
// Create discovery filter
DiscoveryFilter discoveryFilter = DiscoveryFilter.builder()
        .partition("aws")
        .accountIds(111122223333)
        .build();
// Create the discovery keyring
CreateAwsKmsMrkDiscoveryMultiKeyringInput createAwsKmsMrkDiscoveryMultiKeyringInput
 = CreateAwsKmsMrkDiscoveryMultiKeyringInput.builder()
```

```
        .discoveryFilter(discoveryFilter)
        .build();
IKeyring decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

C# / .NET

```
// Create discovery filter
var discoveryFilter = new DiscoveryFilter
{
    Partition = "aws",
    AccountIds = 111122223333
};
// Create the discovery keyring
var createAwsKmsMrkDiscoveryMultiKeyringInput = new
 CreateAwsKmsMrkDiscoveryMultiKeyringInput
{
    DiscoveryFilter = discoveryFilter
};
var decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

Rust

```
// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .partition("aws")
    .account_ids(111122223333)
    .build()?;

// Create the discovery keyring
let decrypt_keyring = mpl
    .create_aws_kms_mrk_discovery_multi_keyring()
    .discovery_filter(discovery_filter)
    .send()
    .await?;
```

使用 AWS KMS 区域发现密钥环

AWS KMS 区域发现密钥环是一种不指定 KMS 密钥 ARNs 的密钥环。相反，它允许 AWS 数据库加密 SDK 仅使用 KMS 密钥进行解密。 AWS 区域

使用 AWS KMS 区域发现密钥环解密时， AWS 数据库加密 SDK 会解密在指定项下加密的所有加密数据密钥。 AWS KMS key AWS 区域要成功，调用者必须拥有对数据密钥 AWS KMS keys 中至少一个加密数据密钥 AWS 区域 的`kms:Decrypt`权限。

与其他 Discovery 密钥环相同，Regional Discovery 密钥环对加密无效。该密钥环仅在解密加密字段时适用。如果您在用于加密和解密的多重密钥环中使用 Regional Discovery 密钥环，则该密钥环仅在解密时有效。如果您单独或在多重密钥环中使用多区域 Discovery 密钥环加密数据，加密操作将失败。

> **Important**
>
> 如果您在解密多密钥环中包含 AWS KMS 区域发现密钥环，则区域发现密钥环将覆盖多密钥环中其他密钥环中指定的所有 KMS 密钥限制。多重密钥环的行为类似于限制最少的密钥环。单独使用或在多重密钥环中使用时， AWS KMS Discovery 密钥环对加密无效。

AWS 数据库加密 SDK 中的区域发现密钥环仅尝试使用指定区域中的 KMS 密钥进行解密。使用发现密钥环时，需要在 AWS KMS 客户端上配置区域。这些 AWS 数据库加密 SDK 实现不会按区域筛选 KMS 密钥，但对指定区域之外的 KMS 密钥的解密请求 AWS KMS 将失败。

如果您使用发现密钥环，我们建议您使用发现过滤器将解密中使用的 KMS 密钥限制为指定 AWS 账户和分区中的密钥。

例如，以下代码使用发现过滤器创建 AWS KMS 区域发现密钥环。此密钥环将 AWS 数据库加密 SDK 限制为美国西部（俄勒冈）区域 (us-west-2) 账户 111122223333 中的 KMS 密钥。

Java

```
// Create the discovery filter
DiscoveryFilter discoveryFilter = DiscoveryFilter.builder()
        .partition("aws")
        .accountIds(111122223333)
        .build();
// Create the discovery keyring
CreateAwsKmsMrkDiscoveryMultiKeyringInput createAwsKmsMrkDiscoveryMultiKeyringInput
 = CreateAwsKmsMrkDiscoveryMultiKeyringInput.builder()
        .discoveryFilter(discoveryFilter)
        .regions("us-west-2")
        .build();
IKeyring decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

C# / .NET

```
// Create discovery filter
var discoveryFilter = new DiscoveryFilter
{
    Partition = "aws",
    AccountIds = 111122223333
};
// Create the discovery keyring
var createAwsKmsMrkDiscoveryMultiKeyringInput = new
 CreateAwsKmsMrkDiscoveryMultiKeyringInput
{
    DiscoveryFilter = discoveryFilter,
    Regions = us-west-2
};
var decryptKeyring =
 matProv.CreateAwsKmsMrkDiscoveryMultiKeyring(createAwsKmsMrkDiscoveryMultiKeyringInput);
```

Rust

```
// Create discovery filter
let discovery_filter = DiscoveryFilter::builder()
    .partition("aws")
    .account_ids(111122223333)
    .build()?;

// Create the discovery keyring
let decrypt_keyring = mpl
    .create_aws_kms_mrk_discovery_multi_keyring()
    .discovery_filter(discovery_filter)
    .regions(us-west-2)
    .send()
    .await?;
```

AWS KMS 分层钥匙圈

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

> **Note**
>
> 自 2023 年 7 月 24 日起，不支持在开发者预览期间创建的分支密钥。创建新的分支密钥以继续使用您在开发者预览版期间创建的密钥库。

使用 AWS KMS 分层密钥环，您可以在对称加密 KMS 密钥下保护您的加密材料，而无需在 AWS KMS 每次加密或解密记录时都调用。对于需要最大限度地减少调用的应用程序以及可以在不违反其安全要求的情况下重复使用某些加密材料的应用程序来说，这是一个不错的选择。 AWS KMS

分层密钥环是一种加密材料缓存解决方案，它使用 AWS KMS 保存在Amazon DynamoDB表中的受保护分支密钥，然后在本地缓存用于加密和解密操作的分支密钥材料，从而减少 AWS KMS 调用次数。DynamoDB 表用作管理和保护分支密钥的密钥存储。其存储活动分支密钥和分支密钥的所有先前版本。活动分支密钥为最新分支密钥版本。分层密钥环对每个加密请求使用唯一的数据加密密钥，并使用从活动分支密钥派生的唯一包装密钥对每个数据加密密钥进行加密。分层密钥环依赖在活动分支密钥及其派生包装密钥之间建立的层次结构。

分层密钥环通常使用各分支密钥版本满足多个请求。但是您可以控制活动分支密钥的重复使用程度，并确定活动分支密钥的轮换频率。在您轮换之前，分支密钥的活动版本会一直处于活动状态。活动分支密钥的先前版本不会用于执行加密操作，但仍可查询并用于解密操作。

当您实例化分层密钥环时，分层密钥环会创建本地缓存。您可以指定缓存限制，该限制定义了分支密钥材料在过期并从缓存中移出之前存储在本地缓存中的最长时间。首次在操作中指定 a `branch-key-id` 时，分层密钥环会 AWS KMS 调用解密分支密钥并组装分支密钥材料。然后，分支密钥材料存储在本地缓存中，并重复用于所有指定 `branch-key-id` 的加密和解密操作，直至缓存限制到期。将分支密钥材料存储在本地缓存中可以减少 AWS KMS 调用。例如，假设缓存限制为 15 分钟。如果您在该缓存限制内执行 10,000 次加密操作，则传统 AWS KMS 密钥环需要进行 10,000 次 AWS KMS 调用才能满足 10,000 次加密操作。如果您有一个处于活动状态`branch-key-id`，则分层密钥环只需要进行一次 AWS KMS 调用即可满足 10,000 个加密操作。

本地缓存将加密材料与解密材料分开。加密材料由活动分支密钥组合而成，并在缓存限制到期之前重复用于所有加密操作。解密材料是根据在加密字段的元数据中标识的分支密钥 ID 和版本汇编而成的，在缓存限制到期之前，它们可以重复用于与分支密钥 ID 和版本相关的所有解密操作。本地缓存可以一次存储同一个分支密钥的多个版本。将本地缓存配置为使用时branch key ID supplier，它还可以同时存储来自多个活动分支密钥的分支密钥材料。

> **Note**
>
> AWS 数据库加密 SDK 中所有提及分层密钥环的内容均指 AWS KMS 分层密钥环。

主题

工作原理

以下演练描述了分层密钥环如何汇编加密和解密材料，以及密钥环对加密和解密操作的不同调用。有关包装密钥派生和明文数据密钥加密过程的技术详细信息，请参阅 AWS KMS 分层密钥环技术详细信息。

加密并签名

以下演练描述了分层密钥环如何汇编加密材料并派生出唯一的包装密钥。

1. 加密方法要求分层密钥环提供加密材料。密钥环生成纯文本数据密钥，然后检查本地缓存中是否有有效的分支密钥材料来生成包装密钥。如果存在有效的分支密钥材料，则密钥环将进入步骤 4。
2. 如果没有有效的分支密钥材料，则分层密钥环会在密钥库中查询活动分支密钥。
    a. 密钥库调用 AWS KMS 解密活动分支密钥并返回纯文本活动分支密钥。标识活动分支密钥的数据会进行序列化，以便在解密调用 AWS KMS时提供额外验证数据。
    b. 密钥库返回纯文本分支密钥和标识该密钥的数据，例如分支密钥版本。
3. 分层密钥环汇编分支密钥材料（明文分支密钥和分支密钥版本），并将其副本存储在本地缓存中。
4. 分层密钥环从明文分支密钥和一个 16 字节的随机加密盐中派生出唯一的包装密钥。其使用派生包装密钥加密明文数据密钥的副本。

此加密方法使用加密材料对记录进行加密和签名。有关如何在 AWS 数据库加密 SDK 中对记录进行加密和签名的更多信息，请参阅加密和签名。

解密并验证

以下演练描述了分层密钥环如何汇编解密材料并解密加密数据密钥。

1. 该解密方法标识来自加密记录的材料描述字段中的加密数据密钥，并将其传递给分层密钥环。
2. 分层密钥环反序列化标识加密数据密钥的数据，包括分支密钥版本、16 字节的加密盐以及其他描述数据密钥加密方式的信息。

   有关更多信息，请参阅 AWS KMS 分层钥匙圈技术细节。
3. 分层密钥环会检查本地缓存中是否存在与步骤 2 标识的分支密钥版本相匹配的有效分支密钥材料。如果存在有效分支密钥材料，则密钥环将进入步骤 6。
4. 如果没有有效的分支密钥材料，则分层密钥环会在密钥库中查询与步骤 2 中确定的分支密钥版本相匹配的分支密钥。

   a. 密钥库调用 AWS KMS 解密分支密钥并返回纯文本活动分支密钥。标识活动分支密钥的数据会进行序列化，以便在解密调用 AWS KMS时提供额外验证数据。

   b. 密钥库返回纯文本分支密钥和标识该密钥的数据，例如分支密钥版本。
5. 分层密钥环汇编分支密钥材料（明文分支密钥和分支密钥版本），并将其副本存储在本地缓存中。
6. 分层密钥环使用汇编的分支密钥材料和步骤 2 标识的 16 字节加密盐重现加密数据密钥的唯一包装密钥。
7. 分层密钥环使用重现的包装密钥解密数据密钥并返回明文数据密钥。

该解密方法使用解密材料和明文数据密钥解密和验证记录。有关如何在 AWS 数据库加密 SDK 中解密和验证记录的更多信息，请参阅解密和验证。

先决条件

在创建和使用分层密钥环之前，请确保满足以下先决条件。

- 您或您的密钥库管理员已创建密钥库并创建了至少一个有效的分支密钥。
- 您已经配置了密钥存储操作。

> **Note**
>
> 如何配置密钥存储操作决定了您可以执行的操作以及分层密钥环可以使用哪些 KMS 密钥。有关更多信息，请参阅密钥存储操作。

- 您拥有访问和使用密钥库和分支密钥所需的 AWS KMS 权限。有关更多信息，请参阅 the section called “所需的权限”。
- 您已经查看了支持的缓存类型并配置了最适合您需求的缓存类型。有关更多信息，请参阅 the section called “选择缓存”。

所需的权限

AWS 数据库加密 SDK 不需要 AWS 账户 ，也不依赖于任何一个 AWS 服务。但是，要使用分层密钥环，您需要对 AWS 账户 密钥库中的对称加密 AWS KMS key具有以下最低权限。

- 要使用分层密钥环加密和解密数据，你需要 kms: Decrypt。
- 要创建和轮换分支密钥，你需要 k ms: GenerateDataKeyWithoutPlaintext 和 kms: ReEncrypt。

有关控制对分支密钥和密钥库的访问权限的更多信息，请参阅the section called “实施最低权限”。

选择缓存

分层密钥环 AWS KMS 通过在本地缓存加密和解密操作中使用的分支密钥材料来减少调用的次数。在创建分层密钥环之前，您需要决定要使用的缓存类型。您可以使用默认缓存或自定义缓存以最适合您的需求。

分层密钥环支持以下缓存类型：

- the section called “默认缓存”
- the section called “MultiThreaded 缓存”
- the section called “StormTracking 缓存”
- the section called “共享缓存”

默认缓存

对于大多数用户而言，默认缓存可满足其线程要求。默认缓存用于支持超多线程环境。当分支密钥材料条目过期时，默认缓存会 AWS KMS 提前 10 秒通知一个线程分支密钥材料条目将过期，从而防止多个线程调用。这样可以确保只有一个线程向发送刷新缓存的请求。 AWS KMS

Default 和 StormTracking 缓存支持相同的线程模型，但您只需要指定入口容量即可使用 Default 缓存。要进行更精细的缓存自定义，请使用。the section called “StormTracking 缓存”

除非要自定义可以存储在本地缓存中的分支密钥材料条目的数量，否则在创建分层密钥环时无需指定缓存类型。如果未指定缓存类型，则分层密钥环使用默认缓存类型并将条目容量设置为 1000。

要自定义默认缓存，请指定以下值：

- 条目容量：限制可以存储在本地缓存中的分支密钥材料条目的数量。

Java

```
.cache(CacheType.builder()
        .Default(DefaultCache.builder()
        .entryCapacity(100)
        .build())
```

C# / .NET

```
CacheType defaultCache = new CacheType
{
    Default = new DefaultCache{EntryCapacity = 100}
};
```

Rust

```
let cache: CacheType = CacheType::Default(
    DefaultCache::builder()
        .entry_capacity(100)
        .build()?,
);
```

MultiThreaded 缓存

MultiThreaded 缓存可在多线程环境中安全使用，但它不提供任何可最大限度减少 AWS KMS 或 Amazon DynamoDB 调用的功能。因此，当分支密钥材料条目到期时，所有线程均将同时收到通知。这可能会导致多次 AWS KMS 调用刷新缓存。

要使用 MultiThreaded 缓存，请指定以下值：

- 条目容量：限制可以存储在本地缓存中的分支密钥材料条目的数量。
- 条目修剪尾部大小：定义在达到条目容量时要修剪的条目数量。

Java

```
.cache(CacheType.builder()
        .MultiThreaded(MultiThreadedCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .build())
```

C# / .NET

```
CacheType multithreadedCache = new CacheType
{
    MultiThreaded = new MultiThreadedCache
    {
        EntryCapacity = 100,
        EntryPruningTailSize = 1
    }
};
```

Rust

```
CacheType::MultiThreaded(
            MultiThreadedCache::builder()
                    .entry_capacity(100)
                    .entry_pruning_tail_size(1)
                    .build()?)
```

StormTracking 缓存

StormTracking 缓存旨在支持大量多线程环境。当分支密钥材料条目过期时， StormTracking 缓存会提前通知一个线程分支密钥材料条目即将过期，从而防止多个线程调用 AWS KMS 。这样可以确保只有一个线程向发送刷新缓存的请求。 AWS KMS

要使用 StormTracking 缓存，请指定以下值：

- 条目容量：限制可以存储在本地缓存中的分支密钥材料条目的数量。

  默认值：1000 个条目
- 条目修剪尾部大小：定义一次要修剪的分支密钥材料条目的数量。

默认值：1 个条目

- 宽限期：定义在到期前尝试刷新分支密钥材料的秒数。

  默认值：10 秒

- 宽限间隔：定义两次尝试刷新分支密钥材料间隔的秒数。

  默认值：1 秒

- 扇出：定义可以同时尝试刷新分支密钥材料的次数。

  默认值：20 次尝试

- 传输中生存时间（TTL）：定义在分支密钥材料刷新尝试超时之前的秒数。每当缓存为响应 GetCacheEntry 而返回 NoSuchEntry 时，分支密钥均视为传输中，直至相同密钥与 PutCache 条目一起写入。

  默认值：10 秒

- 睡眠：定义超过 fanOut 时线程应睡眠的秒数。

  默认值：20 毫秒

Java

```
.cache(CacheType.builder()
        .StormTracking(StormTrackingCache.builder()
        .entryCapacity(100)
        .entryPruningTailSize(1)
        .gracePeriod(10)
        .graceInterval(1)
        .fanOut(20)
        .inFlightTTL(10)
        .sleepMilli(20)
        .build())
```

C# / .NET

```
CacheType stormTrackingCache = new CacheType
{
    StormTracking = new StormTrackingCache
    {
        EntryCapacity = 100,
```

```
        EntryPruningTailSize = 1,
        FanOut = 20,
        GraceInterval = 1,
        GracePeriod = 10,
        InFlightTTL = 10,
        SleepMilli = 20
    }
};
```

Rust

```
CacheType::StormTracking(
                StormTrackingCache::builder()
                    .entry_capacity(100)
                    .entry_pruning_tail_size(1)
                    .grace_period(10)
                    .grace_interval(1)
                    .fan_out(20)
                    .in_flight_ttl(10)
                    .sleep_milli(20)
                    .build()?)
```

共享缓存

默认情况下，每次实例化密钥环时，分层密钥环都会创建一个新的本地缓存。但是，共享缓存允许您在多个分层密钥环之间共享缓存，从而有助于节省内存。共享缓存不是为您实例化的每个分层密钥环创建新的加密材料缓存，而是在内存中只存储一个缓存，供所有引用它的分层密钥环使用。共享缓存可避免在密钥环之间重复加密材料，从而帮助优化内存使用率。相反，分层密钥环可以访问相同的底层缓存，从而减少总体内存占用。

创建共享缓存时，仍需要定义缓存类型。您可以指定the section called “默认缓存”the section called “MultiThreaded 缓存”、或the section called “StormTracking 缓存”作为缓存类型，也可以替换任何兼容的自定义缓存。

分区

多个分层密钥环可以使用单个共享缓存。使用共享缓存创建分层密钥环时，可以定义可选的分区 ID。分区 ID 可区分哪个分层密钥环正在写入缓存。如果两个分层密钥环引用相同的分区 ID 和分支密钥

IDlogical key store name，则两个密钥环将在缓存中共享相同的缓存条目。如果您创建了两个具有相同共享缓存但分区不同的分层密钥环 IDs，则每个密钥环只能访问共享缓存中自己指定的分区中的缓存条目。分区充当共享缓存中的逻辑分区，允许每个分层密钥环在自己的指定分区上独立运行，而不会干扰存储在另一个分区中的数据。

如果您打算重复使用或共享分区中的缓存条目，则必须定义自己的分区 ID。当您将分区 ID 传递给分层密钥环时，密钥环可以重复使用共享缓存中已存在的缓存条目，而不必再次检索和重新授权分支密钥材料。如果您未指定分区 ID，则每次实例化分层密钥环时，都会自动为密钥环分配一个唯一的分区 ID。

以下过程演示如何创建默认缓存类型的共享缓存并将其传递给分层密钥环。

1. 使用材料提供者库 CryptographicMaterialsCache (MPL) 创建 (CMC)。

   Java

   ```
   // Instantiate the MPL
   final MaterialProviders matProv =
       MaterialProviders.builder()
           .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
           .build();

   // Create a CacheType object for the Default cache
   final CacheType cache =
       CacheType.builder()
           .Default(DefaultCache.builder().entryCapacity(100).build())
           .build();

   // Create a CMC using the default cache
   final CreateCryptographicMaterialsCacheInput cryptographicMaterialsCacheInput =
       CreateCryptographicMaterialsCacheInput.builder()
           .cache(cache)
           .build();

   final ICryptographicMaterialsCache sharedCryptographicMaterialsCache =
       matProv.CreateCryptographicMaterialsCache(cryptographicMaterialsCacheInput);
   ```

   C# / .NET

   ```
   // Instantiate the MPL
   var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

   // Create a CacheType object for the Default cache
   ```

```
var cache = new CacheType { Default = new DefaultCache{EntryCapacity = 100} };

// Create a CMC using the default cache
var cryptographicMaterialsCacheInput = new
 CreateCryptographicMaterialsCacheInput {Cache = cache};

var sharedCryptographicMaterialsCache =
 materialProviders.CreateCryptographicMaterialsCache(cryptographicMaterialsCacheInput);
```

Rust

```
// Instantiate the MPL
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create a CacheType object for the default cache
let cache: CacheType = CacheType::Default(
    DefaultCache::builder()
        .entry_capacity(100)
        .build()?,
);

// Create a CMC using the default cache
let shared_cryptographic_materials_cache: CryptographicMaterialsCacheRef = mpl.
    create_cryptographic_materials_cache()
    .cache(cache)
    .send()
    .await?;
```

2. 为共享缓存创建CacheType对象。

   将sharedCryptographicMaterialsCache您在步骤 1 中创建的传递给新CacheType对象。

   Java

```
// Create a CacheType object for the sharedCryptographicMaterialsCache
final CacheType sharedCache =
    CacheType.builder()
        .Shared(sharedCryptographicMaterialsCache)
        .build();
```

C# / .NET

```
// Create a CacheType object for the sharedCryptographicMaterialsCache
var sharedCache = new CacheType { Shared = sharedCryptographicMaterialsCache };
```

Rust

```
// Create a CacheType object for the shared_cryptographic_materials_cache
let shared_cache: CacheType =
 CacheType::Shared(shared_cryptographic_materials_cache);
```

3. 将步骤 2 中的sharedCache对象传递到分层密钥环。

使用共享缓存创建分层密钥环时，可以选择定义一个partitionID以在多个分层密钥环之间共享缓存条目。如果您未指定分区 ID，则分层密钥环会自动为密钥环分配一个唯一的分区 ID。

> **Note**
>
> 如果您创建两个或更多引用相同分区 ID 和分支密钥 ID 的密钥环，则您的分层密钥环将在共享缓存中共享相同的缓存条目。logical key store name如果您不希望多个密钥环共享相同的缓存条目，则必须为每个分层密钥环使用唯一的分区 ID。

以下示例创建了一个分层密钥环branch key ID supplier，其缓存限制为 600 秒。有关以下分层密钥环配置中定义的值的更多信息，请参阅the section called “创建分层密钥环”。

Java

```
// Create the Hierarchical keyring
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
    CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(keystore)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(sharedCache)
        .partitionID(partitionID)
        .build();
final IKeyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
// Create the Hierarchical keyring
var createKeyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   Cache = sharedCache,
   TtlSeconds = 600,
   PartitionId = partitionID
};
var keyring =
 materialProviders.CreateAwsKmsHierarchicalKeyring(createKeyringInput);
```

Rust

```
// Create the Hierarchical keyring
let keyring1 = mpl
    .create_aws_kms_hierarchical_keyring()
    .key_store(key_store1)
    .branch_key_id(branch_key_id.clone())
    // CryptographicMaterialsCacheRef is an Rc (Reference Counted), so if you
 clone it to
    // pass it to different Hierarchical Keyrings, it will still point to the
 same
    // underlying cache, and increment the reference count accordingly.
    .cache(shared_cache.clone())
    .ttl_seconds(600)
    .partition_id(partition_id.clone())
    .send()
    .await?;
```

创建分层密钥环

要创建分层密钥环，必须提供以下值：

- 密钥库名称

  您或您的密钥库管理员创建的用作密钥存储的 DynamoDB 表的名称。

-

缓存限制生存时间（TTL）

本地缓存中的分支密钥材料条目在过期之前可使用的时长（以秒为单位）。缓存限制 TTL 决定了客户端调 AWS KMS 用授权使用分支密钥的频率。该值必须大于零。缓存限制 TTL 到期后，该条目将永远不会被提供，并将从本地缓存中逐出。

- 分支密钥标识符

  您可以静态配置用于标识密钥库中单个活动分支密钥的，也可以提供分支密钥 ID 供应商。`branch-key-id`

  分支密钥 ID 提供者使用存储在加密上下文中的字段来确定解密记录需要哪个分支密钥。默认情况下，加密上下文中仅包含分区和排序密钥。但是，您可以使用加密操作在`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`加密上下文中包含其他字段。

  对于每个租户都有自己的分支密钥的多租户数据库，我们强烈建议使用分支密钥 ID 供应商。您可以使用分支密钥 ID 供应商为分支密钥创建友好名称， IDs 以便轻松识别特定租户的正确分支密钥 ID。例如，易记名称使您可以将分支密钥引用为 `tenant1` 而非 `b3f61619-4d35-48ad-a275-050f87e15122`。

  对于解密操作，您可以静态配置单个分层密钥环以限制对单个租户进行解密，也可以使用分支密钥 ID 提供程序确定哪个租户负责解密记录。

- （可选）缓存

  如果要自定义缓存类型或可存储在本地缓存中分支密钥材料条目的数量，请在初始化密钥环时指定缓存类型和条目容量。

  分层密钥环支持以下缓存类型：默认、 MultiThreaded StormTracking、和共享。有关演示如何定义每种缓存类型的更多信息和示例，请参阅the section called “选择缓存”。

  如果未指定缓存，则分层密钥环会自动使用默认缓存类型并将条目容量设置为 1000。

- （可选）分区 ID

  如果指定the section called “共享缓存”，则可以选择定义分区 ID。分区 ID 可区分哪个分层密钥环正在写入缓存。如果您打算重复使用或共享分区中的缓存条目，则必须定义自己的分区 ID。您可以为分区 ID 指定任何字符串。如果您未指定分区 ID，则会在创建密钥环时自动为密钥环分配一个唯一的分区 ID。

有关更多信息，请参阅 Partitions。

> **Note**
>
> 如果您创建两个或更多引用相同分区 ID 和分支密钥 ID 的密钥环，则您的分层密钥环将在共享缓存中共享相同的缓存条目。logical key store name如果您不希望多个密钥环共享相同的缓存条目，则必须为每个分层密钥环使用唯一的分区 ID。

- （可选）授权令牌列表

  如果您通过授权控制对分层密钥环中 KMS 密钥的访问权限，则必须在初始化密钥环时提供所有必要的授权令牌。

使用静态分支密钥 ID 创建分层密钥环

以下示例演示如何创建具有静态分支密钥 ID、缓存限制 TTL 为 600 秒的分层密钥环。the section called “默认缓存”

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(branchKeyStoreName)
        .branchKeyId(branch-key-id)
        .ttlSeconds(600)
        .build();
final Keyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600
```

```
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let hierarchical_keyring = mpl
    .create_aws_kms_hierarchical_keyring()
    .branch_key_id(branch_key_id)
    .key_store(branch_key_store_name)
    .ttl_seconds(600)
    .send()
    .await?;
```

使用分支密钥 ID 供应商创建分层密钥环

以下过程演示如何使用分支密钥 ID 提供者创建分层密钥环。

1. 创建分支密钥 ID 供应商

   以下示例为步骤 1 中创建的两个分支密钥创建友好名称，并使用适用于 DynamoDB 客户端的 AWS 数据库加密 SDK 调CreateDynamoDbEncryptionBranchKeyIdSupplier用创建分支密钥 ID 提供者。

   Java

   ```
   // Create friendly names for each branch-key-id
   class ExampleBranchKeyIdSupplier implements IDynamoDbKeyBranchKeyIdSupplier {
       private static String branchKeyIdForTenant1;
       private static String branchKeyIdForTenant2;

       public ExampleBranchKeyIdSupplier(String tenant1Id, String tenant2Id) {
           this.branchKeyIdForTenant1 = tenant1Id;
           this.branchKeyIdForTenant2 = tenant2Id;
       }
   // Create the branch key ID supplier
   final DynamoDbEncryption ddbEnc = DynamoDbEncryption.builder()
           .DynamoDbEncryptionConfig(DynamoDbEncryptionConfig.builder().build())
           .build();
   ```

```
final BranchKeyIdSupplier branchKeyIdSupplier =
 ddbEnc.CreateDynamoDbEncryptionBranchKeyIdSupplier(
    CreateDynamoDbEncryptionBranchKeyIdSupplierInput.builder()
            .ddbKeyBranchKeyIdSupplier(new ExampleBranchKeyIdSupplier(branch-
key-ID-tenant1, branch-key-ID-tenant2))
            .build()).branchKeyIdSupplier();
```

C# / .NET

```
// Create friendly names for each branch-key-id
 class ExampleBranchKeyIdSupplier : DynamoDbKeyBranchKeyIdSupplierBase {
    private String _branchKeyIdForTenant1;
    private String _branchKeyIdForTenant2;

    public ExampleBranchKeyIdSupplier(String tenant1Id, String tenant2Id) {
        this._branchKeyIdForTenant1 = tenant1Id;
        this._branchKeyIdForTenant2 = tenant2Id;
    }
// Create the branch key ID supplier
var ddbEnc = new DynamoDbEncryption(new DynamoDbEncryptionConfig());
var branchKeyIdSupplier = ddbEnc.CreateDynamoDbEncryptionBranchKeyIdSupplier(
    new CreateDynamoDbEncryptionBranchKeyIdSupplierInput
    {
        DdbKeyBranchKeyIdSupplier = new ExampleBranchKeyIdSupplier(branch-key-
ID-tenant1, branch-key-ID-tenant2)
    }).BranchKeyIdSupplier;
```

Rust

```
// Create friendly names for each branch_key_id
pub struct ExampleBranchKeyIdSupplier {
    branch_key_id_for_tenant1: String,
    branch_key_id_for_tenant2: String,
}

impl ExampleBranchKeyIdSupplier {
    pub fn new(tenant1_id: &str, tenant2_id: &str) -> Self {
        Self {
            branch_key_id_for_tenant1: tenant1_id.to_string(),
            branch_key_id_for_tenant2: tenant2_id.to_string(),
        }
    }
}
```

```
// Create the branch key ID supplier
let dbesdk_config = DynamoDbEncryptionConfig::builder().build()?;
let dbesdk = dbesdk_client::Client::from_conf(dbesdk_config)?;
let supplier = ExampleBranchKeyIdSupplier::new(tenant1_branch_key_id,
 tenant2_branch_key_id);

let branch_key_id_supplier = dbesdk
    .create_dynamo_db_encryption_branch_key_id_supplier()
    .ddb_key_branch_key_id_supplier(supplier)
    .send()
    .await?
    .branch_key_id_supplier
    .unwrap();
```

2. 创建分层密钥环

以下示例使用步骤 1 中创建的分支密钥 ID 供应商初始化分层密钥环，缓存限制 TLL 为 600 秒，最大缓存大小为 1000。

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(keystore)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(CacheType.builder() //OPTIONAL
                .Default(DefaultCache.builder()
                .entryCapacity(100)
                .build())
        .build();
final Keyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
```

```
    KeyStore = keystore,
    BranchKeyIdSupplier = branchKeyIdSupplier,
    TtlSeconds = 600,
    Cache = new CacheType
    {
         Default = new DefaultCache { EntryCapacity = 100 }
    }
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let hierarchical_keyring = mpl
    .create_aws_kms_hierarchical_keyring()
    .branch_key_id_supplier(branch_key_id_supplier)
    .key_store(key_store)
    .ttl_seconds(600)
    .send()
    .await?;
```

使用分层密钥环进行可搜索加密

可搜索加密使您无需解密整个数据库即可搜索加密的记录。该操作通过使用信标对加密字段的明文值进行索引来实现。要实现可搜索加密，必须使用分层密钥环。

密钥存储 CreateKey 操作将产生分支密钥和信标密钥。分支密钥用于记录加密和解密操作。信标密钥用于生成信标。

分支密钥和信标密钥受您在创建密钥存储服务时指定的相同 AWS KMS key 密钥和信标密钥的保护。在CreateKey操作调 AWS KMS 用生成分支密钥后，它使用以下请求第二次调用 k ms: GenerateDataKeyWithoutPlaintext 以生成信标密钥。

```
{
   "EncryptionContext": {
      "branch-key-id" : "branch-key-id",
      "type" : type,
      "create-time" : "timestamp",
      "logical-key-store-name" : "the logical table name for your key store",
```

```
      "kms-arn" : the KMS key ARN,
      "hierarchy-version" : 1
   },
   "KeyId": "the KMS key ARN",
   "NumberOfBytes": "32"
}
```

生成两个密钥后，该CreateKey操作会调用 ddb: TransactWriteItems 来编写两个新项目，它们将在分支密钥存储中保留分支密钥和信标密钥。

配置标准信标时， AWS 数据库加密 SDK 会在密钥库中查询信标密钥。然后，它使用基于 HMAC 的 extract-and-expand密钥派生函数 (H KDF) 将信标密钥与标准信标的名称相结合，为给定信标创建 HMAC 密钥。

与分支密钥不同，每个密钥库branch-key-id中只有一个信标密钥版本。信标密钥永远不会轮换。

定义您的信标密钥源

为标准信标和复合信标定义信标版本时，必须识别信标密钥并为信标密钥材料定义缓存限制生存时间（TTL）。信标密钥材料与分支密钥分开存储在单独的本地缓存中。以下片段演示了如何为单租户数据库定义 keySource。通过与之关联的 branch-key-id 来识别您的信标密钥。

Java

```
keySource(BeaconKeySource.builder()
        .single(SingleKeyStore.builder()
                .keyId(branch-key-id)
                .cacheTTL(6000)
                .build())
        .build())
```

C# / .NET

```
KeySource = new BeaconKeySource
{
    Single = new SingleKeyStore
    {
       KeyId = branch-key-id,
       CacheTTL = 6000
    }
}
```

Rust

```
.key_source(BeaconKeySource::Single(
   SingleKeyStore::builder()
       // `keyId` references a beacon key.
       // For every branch key we create in the keystore,
       // we also create a beacon key.
       // This beacon key is not the same as the branch key,
       // but is created with the same ID as the branch key.
       .key_id(branch_key_id)
       .cache_ttl(6000)
       .build()?,
))
```

在多租户数据库中定义信标源

如果您有多租户数据库，则在配置 keySource 时必须指定以下值。

- keyFieldName

  定义存储与信标密钥关联的 branch-key-id 的字段名称，该信标密钥用于为给定租户生成信标。keyFieldName 可以是任何字符串，但它对于数据库中所有其他字段必须是唯一的。当您将新记录写入数据库时，标识用于为该记录生成任何信标的信标密钥的 branch-key-id 将存储在此字段中。您必须在信标查询中包含此字段，并确定重新计算信标所需的相应信标密钥材料。有关更多信息，请参阅 查询多租户数据库中的信标。

- cacheTTL

  本地信标缓存中的信标密钥材料条目在过期之前可使用的时长（以秒为单位）。该值必须大于零。当缓存限制 TTL 到期时，该条目将从本地缓存中移出。

- （可选）缓存

  如果要自定义缓存类型或可存储在本地缓存中分支密钥材料条目的数量，请在初始化密钥环时指定缓存类型和条目容量。

  分层密钥环支持以下缓存类型：默认、 MultiThreaded StormTracking、和共享。有关演示如何定义每种缓存类型的更多信息和示例，请参阅the section called “选择缓存”。

  如果未指定缓存，则分层密钥环会自动使用默认缓存类型并将条目容量设置为 1000。

以下示例创建了一个分层密钥环，其分支密钥 ID 提供者缓存限制 TLL 为 600 秒，条目容量为 1000。

Java

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsHierarchicalKeyringInput keyringInput =
 CreateAwsKmsHierarchicalKeyringInput.builder()
        .keyStore(branchKeyStoreName)
        .branchKeyIdSupplier(branchKeyIdSupplier)
        .ttlSeconds(600)
        .cache(CacheType.builder() //OPTIONAL
                .Default(DefaultCache.builder()
                        .entryCapacity(1000)
                        .build())
                .build();
final IKeyring hierarchicalKeyring =
 matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

C# / .NET

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsHierarchicalKeyringInput
{
   KeyStore = keystore,
   BranchKeyIdSupplier = branchKeyIdSupplier,
   TtlSeconds = 600,
   Cache = new CacheType
   {
        Default = new DefaultCache { EntryCapacity = 1000 }
   }
};
var hierarchicalKeyring = matProv.CreateAwsKmsHierarchicalKeyring(keyringInput);
```

Rust

```
let provider_config = MaterialProvidersConfig::builder().build()?;
    let mat_prov = client::Client::from_conf(provider_config)?;
    let kms_keyring = mat_prov
        .create_aws_kms_hierarchical_keyring()
        .branch_key_id(branch_key_id)
```

```
        .key_store(key_store)
        .ttl_seconds(600)
        .send()
        .await?;
```

AWS KMS ECDH 钥匙圈

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

> **Important**
>
> AWS KMS ECDH 密钥环仅在 1.5.0 或更高版本的材料提供者库中可用。

AWS KMS ECDH 密钥环使用非对称密钥协议AWS KMS keys来派生双方共享的对称包装密钥。首先，密钥环使用 Elliptic Curve Diffie-Hellman (ECDH) 密钥协议算法，从发送者的 KMS 密钥对中的私钥和接收者的公钥中派生出共享密钥。然后，密钥环使用共享密钥来派生用于保护您的数据加密密钥的共享包装密钥。 AWS 数据库加密 SDK 使用 (KDF_CTR_HMAC_SHA384) 派生共享包装密钥的密钥派生函数符合 NIST 关于密钥派生的建议。

密钥派生函数返回 64 字节的密钥材料。为确保双方使用正确的密钥材料， AWS 数据库加密 SDK 使用前 32 字节作为承诺密钥，使用最后 32 字节作为共享封装密钥。解密时，如果密钥环无法复制存储在加密记录材料描述字段中的相同承诺密钥和共享包装密钥，则操作将失败。例如，如果您使用配置有 Alice 私钥和 Bob 公钥的密钥环对记录进行加密，则使用 B ob 的私钥和 Alice 的公钥配置的密钥环将复制相同的承诺密钥和共享包装密钥，并能够解密该记录。如果 Bob 的公钥不是来自 KMS 密钥对，那么 Bob 可以创建一个 Ra w ECDH 密钥环来解密记录。

AWS KMS ECDH 密钥环使用 AES-GCM 使用对称密钥对记录进行加密。然后使用 AES-GCM 使用派生的共享包装密钥对数据密钥进行信封加密。每个 AWS KMS ECDH 密钥环只能有一个共享的包装密钥，但您可以在多密钥环中单独或与其他密钥环一起包含多个 AWS KMS ECDH 密钥环。

主题

- AWS KMS ECDH 密钥环所需的权限
- 创建 AWS KMS ECDH 密钥环
- 创建 AWS KMS ECDH 发现密钥环

AWS KMS ECDH 密钥环所需的权限

AWS 数据库加密 SDK 不需要 AWS 帐户，也不依赖任何 AWS 服务。但是，要使用 AWS KMS ECDH 密钥环，您需要一个 AWS 帐户以及对密钥环 AWS KMS keys 中的以下最低权限。权限因您使用的密钥协议架构而异。

- 要使用密`KmsPrivateKeyToStaticPublicKey`钥协议架构加密和解密记录，您需要在发送者的非对称 KMS 密钥对DeriveSharedSecret上使用 kms: GetPublicKey 和 km s:。如果您在实例化密钥环时直接提供发送者的 DER 编码公钥，则只需要对发送者的非对称 KMS 密钥对DeriveSharedSecret具有 kms: 权限。
- 要使用密`KmsPublicKeyDiscovery`钥协议架构解密记录，您需要对指定的非对称 KMS 密钥对具有 kms: DeriveSharedSecret 和 kms: GetPublicKey 权限。

创建 AWS KMS ECDH 密钥环

要创建用于加密和解密数据的 AWS KMS ECDH 密钥环，必须使用密钥协议架构。`KmsPrivateKeyToStaticPublicKey`要使用密钥协议架构初始化 AWS KMS ECDH `KmsPrivateKeyToStaticPublicKey` 密钥环，请提供以下值：

- 发件人 AWS KMS key 身份证

  必须标识值为的非对称 NIST 推荐的椭圆曲线 (ECC) KMS 密钥对。`KeyUsage` `KEY_AGREEMENT`发送者的私钥用于派生共享密钥。

- （可选）发件人的公钥

  必须是 DER 编码的 X.509 公钥，也称为 `SubjectPublicKeyInfo` (SPKI)，如 RFC 5280 中所定义。

  该 AWS KMS GetPublicKey操作以所需的 DER 编码格式返回非对称 KMS 密钥对的公钥。

  要减少密钥环 AWS KMS 拨打的次数，您可以直接提供发件人的公钥。如果没有为发件人的公钥提供任何值，则密钥环会调用 AWS KMS 以检索发送者的公钥。

- 收件人的公钥

  您必须提供收件人的 DER 编码的 X.509 公钥，也称为 `SubjectPublicKeyInfo` (SPKI)，如 RFC 5280 中所定义。

  该 AWS KMS GetPublicKey操作以所需的 DER 编码格式返回非对称 KMS 密钥对的公钥。

- 曲线规格

  标识指定密钥对中的椭圆曲线规范。发件人和收件人的密钥对必须具有相同的曲线规格。

  有效值：ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512

- （可选）授权令牌列表

  如果您通过授权控制对 AWS KMS ECDH 密钥环中 KMS 密钥的访问权限，则在初始化密钥环时必须提供所有必要的授权令牌。

C# / .NET

以下示例使用发件人的 KMS 密钥、发件人的公钥和收件人的公钥创建一个 AWS KMS ECDH 密钥环。此示例使用可选senderPublicKey参数来提供发送者的公钥。如果您不提供发件人的公钥，则密钥环会调用 AWS KMS 以检索发件人的公钥。发件人和收件人的密钥对都在ECC_NIST_P256弯曲中。

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Must be DER-encoded X.509 public keys
var BobPublicKey = new MemoryStream(new byte[] { });
var AlicePublicKey = new MemoryStream(new byte[] { });

// Create the AWS KMS ECDH static keyring
var staticConfiguration = new KmsEcdhStaticConfigurations
{
    KmsPrivateKeyToStaticPublicKey = new KmsPrivateKeyToStaticPublicKeyInput
    {
        SenderKmsIdentifier = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab",
        SenderPublicKey = BobPublicKey,
        RecipientPublicKey = AlicePublicKey
    }
};

var createKeyringInput = new CreateAwsKmsEcdhKeyringInput
{
    CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
    KmsClient = new AmazonKeyManagementServiceClient(),
    KeyAgreementScheme = staticConfiguration
};
```

```
var keyring = materialProviders.CreateAwsKmsEcdhKeyring(createKeyringInput);
```

Java

以下示例使用发件人的 KMS 密钥、发件人的公钥和收件人的公钥创建一个 AWS KMS ECDH 密钥环。此示例使用可选senderPublicKey参数来提供发送者的公钥。如果您不提供发件人的公钥，则密钥环会调用 AWS KMS 以检索发件人的公钥。发件人和收件人的密钥对都在ECC_NIST_P256弯曲中。

```
// Retrieve public keys
// Must be DER-encoded X.509 public keys
ByteBuffer BobPublicKey = getPublicKeyBytes("arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab");
        ByteBuffer AlicePublicKey = getPublicKeyBytes("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321");

// Create the AWS KMS ECDH static keyring
        final CreateAwsKmsEcdhKeyringInput senderKeyringInput =
          CreateAwsKmsEcdhKeyringInput.builder()
            .kmsClient(KmsClient.create())
            .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
            .KeyAgreementScheme(
              KmsEcdhStaticConfigurations.builder()
                .KmsPrivateKeyToStaticPublicKey(
                  KmsPrivateKeyToStaticPublicKeyInput.builder()
                    .senderKmsIdentifier("arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab")
                    .senderPublicKey(BobPublicKey)
                    .recipientPublicKey(AlicePublicKey)
                    .build()).build()).build();
```

Rust

以下示例使用发件人的 KMS 密钥、发件人的公钥和收件人的公钥创建一个 AWS KMS ECDH 密钥环。此示例使用可选sender_public_key参数来提供发送者的公钥。如果您不提供发件人的公钥，则密钥环会调用 AWS KMS 以检索发件人的公钥。

```
// Retrieve public keys
// Must be DER-encoded X.509 keys
let public_key_file_content_sender =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_SENDER))?;
```

```
let parsed_public_key_file_content_sender = parse(public_key_file_content_sender)?;
let public_key_sender_utf8_bytes = parsed_public_key_file_content_sender.contents();

let public_key_file_content_recipient =
 std::fs::read_to_string(Path::new(EXAMPLE_KMS_ECC_PUBLIC_KEY_FILENAME_RECIPIENT))?;
let parsed_public_key_file_content_recipient =
 parse(public_key_file_content_recipient)?;
let public_key_recipient_utf8_bytes =
 parsed_public_key_file_content_recipient.contents();

// Create KmsPrivateKeyToStaticPublicKeyInput
let kms_ecdh_static_configuration_input =
    KmsPrivateKeyToStaticPublicKeyInput::builder()
        .sender_kms_identifier(arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab)
        // Must be a UTF8 DER-encoded X.509 public key
        .sender_public_key(public_key_sender_utf8_bytes)
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let kms_ecdh_static_configuration =
 KmsEcdhStaticConfigurations::KmsPrivateKeyToStaticPublicKey(kms_ecdh_static_configuration_i

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create AWS KMS ECDH keyring
let kms_ecdh_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client)
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_static_configuration)
    .send()
    .await?;
```

创建 AWS KMS ECDH 发现密钥环

解密时，最佳做法是指定 AWS 数据库加密 SDK 可以使用的密钥。要遵循此最佳实践，请使用带有密钥协议架构的 AWS KMS ECDH KmsPrivateKeyToStaticPublicKey 密钥环。但是，您也可以创

建 AWS KMS ECDH 发现密钥环，即 AWS KMS ECDH 密钥环，该密钥环可以解密任何记录，其中指定 KMS 密钥对的公钥与存储在加密记录的材料描述字段中的接收者的公钥相匹配。

> **Important**
>
> 使用密KmsPublicKeyDiscovery钥协议架构解密记录时，无论谁拥有所有公钥，都将接受所有公钥。

要使用密钥协议架构初始化 AWS KMS ECDH KmsPublicKeyDiscovery 密钥环，请提供以下值：

- 收件人的 AWS KMS key 身份证

  必须标识值为的非对称 NIST 推荐的椭圆曲线 (ECC) KMS 密钥对。KeyUsage KEY_AGREEMENT

- 曲线规格

  标识收件人的 KMS 密钥对中的椭圆曲线规范。

  有效值：ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512

- （可选）授权令牌列表

  如果您通过授权控制对 AWS KMS ECDH 密钥环中 KMS 密钥的访问权限，则在初始化密钥环时必须提供所有必要的授权令牌。

C# / .NET

以下示例创建了一个 AWS KMS ECDH 发现密钥环，曲线上有 KMS 密钥对。ECC_NIST_P256您必须对指定的 KMS 密钥对拥有 kms: GetPublicKey 和 kms: DeriveSharedSecret 权限。此密钥环可以解密任何记录，其中指定 KMS 密钥对的公钥与存储在加密记录的材料描述字段中的接收者的公钥相匹配。

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());

// Create the AWS KMS ECDH discovery keyring
var discoveryConfiguration = new KmsEcdhStaticConfigurations
{
    KmsPublicKeyDiscovery = new KmsPublicKeyDiscoveryInput
    {
```

```
        RecipientKmsIdentifier = "arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321"
    }

};
var createKeyringInput = new CreateAwsKmsEcdhKeyringInput
{
    CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
    KmsClient = new AmazonKeyManagementServiceClient(),
    KeyAgreementScheme = discoveryConfiguration
};
var keyring = materialProviders.CreateAwsKmsEcdhKeyring(createKeyringInput);
```

Java

以下示例创建了一个 AWS KMS ECDH 发现密钥环，曲线上有 KMS 密钥对。ECC_NIST_P256您必须对指定的 KMS 密钥对拥有 kms: GetPublicKey 和 kms: DeriveSharedSecret 权限。此密钥环可以解密任何记录，其中指定 KMS 密钥对的公钥与存储在加密记录的材料描述字段中的接收者的公钥相匹配。

```
// Create the AWS KMS ECDH discovery keyring
final CreateAwsKmsEcdhKeyringInput recipientKeyringInput =
  CreateAwsKmsEcdhKeyringInput.builder()
    .kmsClient(KmsClient.create())
    .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
    .KeyAgreementScheme(
      KmsEcdhStaticConfigurations.builder()
        .KmsPublicKeyDiscovery(
          KmsPublicKeyDiscoveryInput.builder()
            .recipientKmsIdentifier("arn:aws:kms:us-
west-2:111122223333:key/0987dcba-09fe-87dc-65ba-ab0987654321").build()
        ).build())
    .build();
```

Rust

```
// Create KmsPublicKeyDiscoveryInput
let kms_ecdh_discovery_static_configuration_input =
    KmsPublicKeyDiscoveryInput::builder()
        .recipient_kms_identifier(ecc_recipient_key_arn)
        .build()?;
```

```
let kms_ecdh_discovery_static_configuration =
 KmsEcdhStaticConfigurations::KmsPublicKeyDiscovery(kms_ecdh_discovery_static_configuration_

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create AWS KMS ECDH discovery keyring
let kms_ecdh_discovery_keyring = mpl
    .create_aws_kms_ecdh_keyring()
    .kms_client(kms_client.clone())
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(kms_ecdh_discovery_static_configuration)
    .send()
    .await?;
```

原始 AES 密钥环

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

AWS 数据库加密 SDK 允许您使用您提供的 AES 对称密钥作为包装密钥来保护您的数据密钥。您需要生成、存储和保护密钥材料，最好是在硬件安全模块（HSM）或密钥管理系统中操作。如果您需要提供包装密钥并在本地或离线加密数据密钥，则请使用原始 AES 密钥环。

原始 AES 密钥环使用 AES-GCM 算法以及您指定为字节数组的包装密钥对数据进行加密。每个原始 AES 密钥环中只能指定一个包装密钥，但每个多重密钥环中可以包含多个原始 AES 密钥环，该等密钥环可单独纳入或与其他密钥环一同纳入。

密钥命名空间和名称

为标识密钥环中的 AES 密钥，原始 AES 密钥环使用您提供的密钥命名空间和密钥名称。这些值不是机密的。它们以纯文本形式出现在 AWS 数据库加密 SDK 添加到记录中的材料描述中。建议在 HSM 或密钥管理系统中使用密钥命名空间与用于标识该系统中 AES 密钥的密钥名称。

> **Note**
>
> 密钥命名空间和密钥名称等同于 JceMasterKey 中的提供程序 ID（或提供程序）和密钥 ID 字段。

如果您通过构造不同的密钥环加密和解密给定字段，命名空间和名称值则至关重要。如果解密密钥环中的密钥命名空间和密钥名称与加密密钥环中的密钥命名空间和密钥名称不完全匹配、大小写不一致，即使密钥材料字节数相同，也不会使用解密密钥环。

例如，您可以使用密钥命名空间 HSM_01 和密钥名称 AES_256_012 定义原始 AES 密钥环。然后使用该密钥环加密部分数据。要解密这些数据，请使用相同的密钥命名空间、密钥名称和密钥材料构造原始 AES 密钥环。

以下示例说明了如何创建原始 AES 密钥。AESWrappingKey 变量代表您提供的密钥材料。

Java

```
final CreateRawAesKeyringInput keyringInput = CreateRawAesKeyringInput.builder()
        .keyName("AES_256_012")
        .keyNamespace("HSM_01")
        .wrappingKey(AESWrappingKey)
        .wrappingAlg(AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16)
        .build();
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(keyringInput);
```

C# / .NET

```
var keyNamespace = "HSM_01";
var keyName = "AES_256_012";

// This example uses the key generator in Bouncy Castle to generate the key
 material.
// In production, use key material from a secure source.
var aesWrappingKey = new
 MemoryStream(GeneratorUtilities.GetKeyGenerator("AES256").GenerateKey());

// Create the keyring
```

```
var keyringInput = new CreateRawAesKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    WrappingKey = AESWrappingKey,
    WrappingAlg = AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
};

var matProv = new MaterialProviders(new MaterialProvidersConfig());
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(keyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name("AES_256_012")
    .key_namespace("HSM_01")
    .wrapping_key(aes_key_bytes)
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;
```

原始 RSA 密钥环

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

原始 RSA 密钥环使用您提供的 RSA 公有密钥和私有密钥为本地内存中的数据密钥执行非对称加密和解密。您需要生成、存储和保护私有密钥，最好是在硬件安全模块（HSM）或密钥管理系统中操作。加密功能对 RSA 公有密钥下的数据密钥进行加密。解密功能使用私有密钥对数据密钥进行解密。您可以从几种 RSA 填充模式中进行选择。

加密和解密的原始 RSA 密钥环必须包含一个非对称公有密钥和私有密钥对。但是，您可以使用仅具有公有密钥的原始 RSA 密钥环加密数据，并使用仅具有私有密钥的原始 RSA 密钥环解密数据。您可以在多重密钥环中包含任何原始 RSA 密钥环。如果您为原始 RSA 密钥环配置公有密钥和私有密钥，请确保其属于同一个密钥对。

Raw RSA 密钥环与 RSA 非对称加密密钥一起使用 AWS Encryption SDK for Java 时，等同于并与之互操作。JceMasterKey

> **Note**
>
> 原始 RSA 密钥环不支持非对称 KMS 密钥。要使用非对称 RSA KMS 密钥，请构造 AWS KMS 密钥环。

命名空间和名称

为标识密钥环中的 RSA 密钥材料，原始 AES 密钥环使用您提供的命名空间和密钥名称。这些值不是机密的。它们以纯文本形式出现在 AWS 数据库加密 SDK 添加到记录中的材料描述中。建议在 HSM 或密钥管理系统中使用密钥命名空间与用于标识 RSA 密钥对（或其私有密钥）的密钥名称。

> **Note**
>
> 密钥命名空间和密钥名称等同于 `JceMasterKey` 中的提供程序 ID（或提供程序）和密钥 ID 字段。

如果您通过构造不同的密钥环加密和解密给定记录，命名空间和名称值则至关重要。如果解密密钥环中的密钥命名空间和密钥名称与加密密钥环中的密钥命名空间和密钥名称不完全匹配、大小写不一致，即使密钥来自相同的密钥对，也不会使用解密密钥环。

无论密钥环中包含 RSA 公有密钥、RSA 私有密钥还是密钥对中的两个密钥，加密和解密密钥环中密钥材料的密钥命名空间和密钥名称必须相同。例如，假设您使用包含密钥命名空间 `HSM_01` 和密钥名称 `RSA_2048_06` 的 RSA 公有密钥的原始 RSA 密钥环加密数据。要解密数据，请使用私有密钥（或密钥对）、相同的密钥命名空间和名称构造原始 RSA 密钥环。

填充模式

您必须为用于加密和解密的原始 RSA 密钥环指定填充模式，或者使用为您指定填充模式的语言实施功能。

AWS Encryption SDK 支持以下填充模式，受每种语言的限制。我们建议使用 OAEP 填充模式，尤其是带有 SHA-256 和 MGF1 SHA-256 填充的 OAEP。仅支持PKCS1填充模式是为了向后兼容。

- 带有 SHA-1 和 MGF1 SHA-1 填充的 OAEP
- 带有 SHA-256 和 MGF1 SHA-256 填充的 OAEP

- 带有 SHA-384 和 MGF1 SHA-384 填充的 OAEP
- 带有 SHA-512 和 MGF1 SHA-512 填充的 OAEP
- PKCS1 v1.5 填充

以下 Java 示例展示了如何使用 RSA 密钥对的公钥和私钥以及使用 SHA-256 和 MGF1 SHA-256 填充模式的 OAEP 创建原始 RSA 密钥环。RSAPublicKey 和 RSAPrivateKey 变量代表您提供的密钥材料。

Java

```
final CreateRawRsaKeyringInput keyringInput = CreateRawRsaKeyringInput.builder()
        .keyName("RSA_2048_06")
        .keyNamespace("HSM_01")
        .paddingScheme(PaddingScheme.OAEP_SHA256_MGF1)
        .publicKey(RSAPublicKey)
        .privateKey(RSAPrivateKey)
        .build();
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
IKeyring rawRsaKeyring = matProv.CreateRawRsaKeyring(keyringInput);
```

C# / .NET

```
var keyNamespace = "HSM_01";
var keyName = "RSA_2048_06";

// Get public and private keys from PEM files
var publicKey = new
 MemoryStream(System.IO.File.ReadAllBytes("RSAKeyringExamplePublicKey.pem"));
var privateKey = new
 MemoryStream(System.IO.File.ReadAllBytes("RSAKeyringExamplePrivateKey.pem"));

// Create the keyring input
var keyringInput = new CreateRawRsaKeyringInput
{
    KeyNamespace = keyNamespace,
    KeyName = keyName,
    PaddingScheme = PaddingScheme.OAEP_SHA512_MGF1,
    PublicKey = publicKey,
    PrivateKey = privateKey
```

```
};

// Create the keyring
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var rawRsaKeyring = matProv.CreateRawRsaKeyring(keyringInput);
```

Rust

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;
let raw_rsa_keyring = mpl
    .create_raw_rsa_keyring()
    .key_name("RSA_2048_06")
    .key_namespace("HSM_01")
    .padding_scheme(PaddingScheme::OaepSha256Mgf1)
    .public_key(RSA_public_key)
    .private_key(RSA_private_key)
    .send()
    .await?;
```

未加工的 ECDH 钥匙圈

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

> **Important**
>
> Raw ECDH 密钥环仅在材质提供者库的 1.5.0 版本中可用。

Raw ECDH 密钥环使用您提供的椭圆曲线公私钥对来派生出双方之间的共享包装密钥。首先，密钥环使用发送者的私钥、收件人的公钥和 Elliptic Curve Diffie-Hellman (ECDH) 密钥协议算法派生出共享密钥。然后，密钥环使用共享密钥来派生用于保护您的数据加密密钥的共享包装密钥。 AWS 数据库加密 SDK 使用 (KDF_CTR_HMAC_SHA384) 派生共享包装密钥的密钥派生函数符合 NIST 关于密钥派生的建议。

密钥派生函数返回 64 字节的密钥材料。为确保双方使用正确的密钥材料， AWS 数据库加密 SDK 使用前 32 字节作为承诺密钥，使用最后 32 字节作为共享封装密钥。解密时，如果密钥环无法复制存储

在加密记录材料描述字段中的相同承诺密钥和共享包装密钥，则操作将失败。例如，如果您使用配置有 Alice 私钥和 Bob 公钥的密钥环对记录进行加密，则使用 B ob 的私钥和 Alice 的公钥配置的密钥环将复制相同的承诺密钥和共享包装密钥，并能够解密该记录。如果 Bob 的公钥来自一 AWS KMS key 对，那么 Bob 可以创建 AWS KMS ECDH 密钥环来解密记录。

Raw ECDH 密钥环使用 AES-GCM 使用对称密钥对记录进行加密。然后使用 AES-GCM 使用派生的共享包装密钥对数据密钥进行信封加密。每个 Raw ECDH 密钥环只能有一个共享包装密钥，但您可以在多密钥环中单独或与其他密钥环一起包含多个 Raw ECDH 密钥环。

您负责生成、存储和保护您的私钥，最好是在硬件安全模块 (HSM) 或密钥管理系统中。发件人和收件人的密钥对在相同的椭圆曲线上。 AWS 数据库加密 SDK 支持以下椭圆曲线规格：

- `ECC_NIST_P256`
- `ECC_NIST_P384`
- `ECC_NIST_P512`

创建原始的 ECDH 密钥环

Raw ECDH 密钥环支持三种密钥协议架构：`RawPrivateKeyToStaticPublicKey`、`EphemeralPrivateKeyToStaticPublicKey`和。`PublicKeyDiscovery`选择的密钥协议架构决定了您可以执行哪些加密操作以及密钥材料的组装方式。

主题

- RawPrivateKeyToStaticPublicKey
- EphemeralPrivateKeyToStaticPublicKey
- PublicKeyDiscovery

RawPrivateKeyToStaticPublicKey

使用`RawPrivateKeyToStaticPublicKey`密钥协议架构在密钥环中静态配置发送者的私钥和收件人的公钥。此密钥协议架构可以加密和解密记录。

要使用密钥协议架构初始化 Raw ECDH `RawPrivateKeyToStaticPublicKey` 密钥环，请提供以下值：

- 发件人的私钥

  您必须提供发件人的 PEM 编码私钥（PKCS #8 PrivateKeyInfo 结构），如 RFC 5958 中所定义。

- 收件人的公钥

  您必须提供收件人的 DER 编码的 X.509 公钥，也称为 SubjectPublicKeyInfo (SPKI)，如 RFC 5280 中所定义。

  您可以指定非对称密钥协议 KMS 密钥对的公钥，也可以指定在外部生成的密钥对中的 AWS公钥。

- 曲线规格

  标识指定密钥对中的椭圆曲线规范。发件人和收件人的密钥对必须具有相同的曲线规格。

  有效值：ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512

C# / .NET

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var BobPrivateKey = new MemoryStream(new byte[] { });
     var AlicePublicKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH static keyring
     var staticConfiguration = new RawEcdhStaticConfigurations()
     {
      RawPrivateKeyToStaticPublicKey = new RawPrivateKeyToStaticPublicKeyInput
      {
       SenderStaticPrivateKey = BobPrivateKey,
       RecipientPublicKey = AlicePublicKey
      }
     };

     var createKeyringInput = new CreateRawEcdhKeyringInput()
     {
      CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
      KeyAgreementScheme = staticConfiguration
     };

     var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

以下 Java 示例使用RawPrivateKeyToStaticPublicKey密钥协议架构静态配置发送者的私钥和收件人的公钥。两个密钥对都在ECC_NIST_P256曲线上。

```
private static void StaticRawKeyring() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    KeyPair senderKeys = GetRawEccKey();
    KeyPair recipient = GetRawEccKey();

    // Create the Raw ECDH static keyring
    final CreateRawEcdhKeyringInput rawKeyringInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .RawPrivateKeyToStaticPublicKey(
                RawPrivateKeyToStaticPublicKeyInput.builder()
                  // Must be a PEM-encoded private key

 .senderStaticPrivateKey(ByteBuffer.wrap(senderKeys.getPrivate().getEncoded()))
                  // Must be a DER-encoded X.509 public key

 .recipientPublicKey(ByteBuffer.wrap(recipient.getPublic().getEncoded()))
                  .build()
            )
            .build()
        ).build();

    final IKeyring staticKeyring =
 materialProviders.CreateRawEcdhKeyring(rawKeyringInput);
}
```

Rust

以下 Python 示例使用raw_ecdh_static_configuration密钥协议架构静态配置发送者的私钥和接收者的公钥。两个密钥对必须位于同一条曲线上。

```
// Create keyring input
let raw_ecdh_static_configuration_input =
    RawPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .sender_static_private_key(private_key_sender_utf8_bytes)
```

```
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let raw_ecdh_static_configuration =
 RawEcdhStaticConfigurations::RawPrivateKeyToStaticPublicKey(raw_ecdh_static_configuration_i

// Instantiate the material providers library
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create raw ECDH static keyring
let raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(raw_ecdh_static_configuration)
    .send()
    .await?;
```

EphemeralPrivateKeyToStaticPublicKey

使用密钥协议架构配置的EphemeralPrivateKeyToStaticPublicKey密钥环在本地创建新的密钥对，并为每个加密调用派生一个唯一的共享包装密钥。

此密钥协议架构只能加密记录。要解密使用密EphemeralPrivateKeyToStaticPublicKey钥协议架构加密的记录，必须使用配置有相同收件人公钥的发现密钥协议架构。要解密，您可以使用带有密钥协议算法的原始 ECDH 密钥环，或者，如果接收者的公PublicKeyDiscovery钥来自非对称密钥协议 KMS 密钥对，则可以将 AWS KMS ECDH 密钥环与密钥协议架构一起使用。KmsPublicKeyDiscovery

要使用密钥协议架构初始化 Raw ECDH EphemeralPrivateKeyToStaticPublicKey 密钥环，请提供以下值：

- 收件人的公钥

  您必须提供收件人的 DER 编码的 X.509 公钥，也称为 SubjectPublicKeyInfo (SPKI)，如 RFC 5280 中所定义。

  您可以指定非对称密钥协议 KMS 密钥对的公钥，也可以指定在外部生成的密钥对中的 AWS公钥。

- 曲线规格

标识指定公钥中的椭圆曲线规范。

加密时，密钥环会在指定曲线上创建新的密钥对，并使用新的私钥和指定的公钥来派生共享的包装密钥。

有效值：ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512

C# / .NET

以下示例使用密钥协议架构创建一个 Raw ECDH EphemeralPrivateKeyToStaticPublicKey 密钥环。加密后，密钥环将在指定ECC_NIST_P256曲线上本地创建一个新的密钥对。

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var AlicePublicKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH ephemeral keyring
     var ephemeralConfiguration = new RawEcdhStaticConfigurations()
     {
      EphemeralPrivateKeyToStaticPublicKey = new
 EphemeralPrivateKeyToStaticPublicKeyInput
      {
       RecipientPublicKey = AlicePublicKey
      }
     };

     var createKeyringInput = new CreateRawEcdhKeyringInput()
     {
      CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
      KeyAgreementScheme = ephemeralConfiguration
     };

     var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

以下示例使用密钥协议架构创建一个 Raw ECDH EphemeralPrivateKeyToStaticPublicKey 密钥环。加密后，密钥环将在指定ECC_NIST_P256曲线上本地创建一个新的密钥对。

```
private static void EphemeralRawEcdhKeyring() {
    // Instantiate material providers
```

```
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    ByteBuffer recipientPublicKey = getPublicKeyBytes();

    // Create the Raw ECDH ephemeral keyring
    final CreateRawEcdhKeyringInput ephemeralInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .EphemeralPrivateKeyToStaticPublicKey(
              EphemeralPrivateKeyToStaticPublicKeyInput.builder()
                .recipientPublicKey(recipientPublicKey)
                .build()
            )
            .build()
        ).build();

    final IKeyring ephemeralKeyring =
 materialProviders.CreateRawEcdhKeyring(ephemeralInput);
}
```

Rust

以下示例使用密钥协议架构创建一个 Raw ECDH ephemeral_raw_ecdh_static_configuration 密钥环。加密后，密钥环将在指定曲线上本地创建一个新的密钥对。

```
// Create EphemeralPrivateKeyToStaticPublicKeyInput
let ephemeral_raw_ecdh_static_configuration_input =
    EphemeralPrivateKeyToStaticPublicKeyInput::builder()
        // Must be a UTF8 DER-encoded X.509 public key
        .recipient_public_key(public_key_recipient_utf8_bytes)
        .build()?;

let ephemeral_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::EphemeralPrivateKeyToStaticPublicKey(ephemeral_raw_ecdh_static

// Instantiate the material providers library
```

```
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

// Create raw ECDH ephemeral private key keyring
let ephemeral_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(ephemeral_raw_ecdh_static_configuration)
    .send()
    .await?;
```

PublicKeyDiscovery

解密时，最佳做法是指定 AWS 数据库加密 SDK 可以使用的包装密钥。要遵循此最佳实践，请使用同时指定发件人私钥和收件人公钥的 ECDH 密钥环。但是，您也可以创建原始ECDH发现密钥环，即原始ECDH密钥环，该密钥环可以解密任何记录，其中指定密钥的公钥与存储在加密记录的材料描述字段中的接收者的公钥相匹配。此密钥协议架构只能解密记录。

> **Important**
>
> 使用密PublicKeyDiscovery钥协议架构解密记录时，无论谁拥有所有公钥，都将接受所有公钥。

要使用密钥协议架构初始化 Raw ECDH PublicKeyDiscovery 密钥环，请提供以下值：

- 收件人的静态私钥

  您必须提供收件人的 PEM 编码私钥（PKCS #8 PrivateKeyInfo 结构），如 RFC 5958 中所定义。

- 曲线规格

  标识指定私钥中的椭圆曲线规范。发件人和收件人的密钥对必须具有相同的曲线规格。

  有效值：ECC_NIST_P256、ECC_NIS_P384、ECC_NIST_P512

C# / .NET

以下示例使用密钥协议架构创建一个 Raw ECDH PublicKeyDiscovery 密钥环。该密钥环可以解密任何记录，其中指定私钥的公钥与存储在加密记录的材料描述字段中的接收者的公钥相匹配。

```
// Instantiate material providers
var materialProviders = new MaterialProviders(new MaterialProvidersConfig());
     var AlicePrivateKey = new MemoryStream(new byte[] { });

     // Create the Raw ECDH discovery keyring
     var discoveryConfiguration = new RawEcdhStaticConfigurations()
     {
      PublicKeyDiscovery = new PublicKeyDiscoveryInput
      {
       RecipientStaticPrivateKey = AlicePrivateKey
      }
     };

     var createKeyringInput = new CreateRawEcdhKeyringInput()
     {
      CurveSpec = ECDHCurveSpec.ECC_NIST_P256,
      KeyAgreementScheme = discoveryConfiguration
     };

     var keyring = materialProviders.CreateRawEcdhKeyring(createKeyringInput);
```

Java

以下示例使用密钥协议架构创建一个 Raw ECDH PublicKeyDiscovery 密钥环。该密钥环可以解密任何记录，其中指定私钥的公钥与存储在加密记录的材料描述字段中的接收者的公钥相匹配。

```
private static void RawEcdhDiscovery() {
    // Instantiate material providers
    final MaterialProviders materialProviders =
      MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();

    KeyPair recipient = GetRawEccKey();

    // Create the Raw ECDH discovery keyring
    final CreateRawEcdhKeyringInput rawKeyringInput =
      CreateRawEcdhKeyringInput.builder()
        .curveSpec(ECDHCurveSpec.ECC_NIST_P256)
        .KeyAgreementScheme(
          RawEcdhStaticConfigurations.builder()
            .PublicKeyDiscovery(
              PublicKeyDiscoveryInput.builder()
```

```
                // Must be a PEM-encoded private key

 .recipientStaticPrivateKey(ByteBuffer.wrap(sender.getPrivate().getEncoded()))
                .build()
            )
            .build()
        ).build();

    final IKeyring publicKeyDiscovery  =
 materialProviders.CreateRawEcdhKeyring(rawKeyringInput);
}
```

Rust

以下示例使用密钥协议架构创建一个 Raw ECDH `discovery_raw_ecdh_static_configuration` 密钥环。此密钥环可以解密任何消息，其中指定私钥的公钥与存储在消息密文中的收件人的公钥相匹配。

```
// Create PublicKeyDiscoveryInput
let discovery_raw_ecdh_static_configuration_input =
    PublicKeyDiscoveryInput::builder()
        // Must be a UTF8 PEM-encoded private key
        .recipient_static_private_key(private_key_recipient_utf8_bytes)
        .build()?;

let discovery_raw_ecdh_static_configuration =

 RawEcdhStaticConfigurations::PublicKeyDiscovery(discovery_raw_ecdh_static_configuration_inp

// Create raw ECDH discovery private key keyring
let discovery_raw_ecdh_keyring = mpl
    .create_raw_ecdh_keyring()
    .curve_spec(ecdh_curve_spec)
    .key_agreement_scheme(discovery_raw_ecdh_static_configuration)
    .send()
    .await?;
```

多重密钥环

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

您可以将多个密钥环合并为一个多重密钥环。多重密钥环 是由一个或多个相同或不同类型的密钥环组成的密钥环。效果与连续使用多个密钥环类似。在使用多重密钥环加密数据时，其任意密钥环中的任意包装密钥都可以对数据进行解密。

在创建多重密钥环以加密数据时，您可以将一个密钥环指定为生成器密钥环。所有其他密钥环称为子密钥环。生成器密钥环生成并加密明文数据密钥。然后，所有子密钥环中的所有包装密钥都加密相同的明文数据密钥。对于多重密钥环中的每个包装密钥，多重密钥环都返回明文密钥和一个加密的数据密钥。如果生成器密钥环是 KMS 密钥环，则密钥 AWS KMS 环中的生成器密钥会生成并加密纯文本密钥。然后，密钥环 AWS KMS keys 中的所有其他密钥，以及多密 AWS KMS 钥环中所有子密钥环中的所有封装密钥，都将加密相同的纯文本密钥。

解密时， AWS 数据库加密 SDK 使用密钥环尝试解密其中一个加密的数据密钥。密钥环是按照在多重密钥环中指定的顺序调用的。只要任何密钥环中的任何密钥可以解密加密的数据密钥，处理就会立即停止。

要创建多重密钥环，首先要实例化子密钥环。在此示例中，我们使用 AWS KMS 密钥环和 Raw AES 密钥环，但您可以将任何支持的密钥环组合到一个多密钥环中。

Java

```
// 1. Create the raw AES keyring.
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateRawAesKeyringInput createRawAesKeyringInput =
 CreateRawAesKeyringInput.builder()
        .keyName("AES_256_012")
        .keyNamespace("HSM_01")
        .wrappingKey(AESWrappingKey)
        .wrappingAlg(AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16)
        .build();
IKeyring rawAesKeyring = matProv.CreateRawAesKeyring(createRawAesKeyringInput);

// 2. Create the AWS KMS keyring.
```

```
final CreateAwsKmsMrkMultiKeyringInput createAwsKmsMrkMultiKeyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyArn)
        .build();
IKeyring awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

C# / .NET

```
// 1. Create the raw AES keyring.
var keyNamespace = "HSM_01";
var keyName = "AES_256_012";

var matProv = new MaterialProviders(new MaterialProvidersConfig());
var createRawAesKeyringInput = new CreateRawAesKeyringInput
{
    KeyName = "keyName",
    KeyNamespace = "myNamespaces",
    WrappingKey = AESWrappingKey,
    WrappingAlg = AesWrappingAlg.ALG_AES256_GCM_IV12_TAG16
};
var rawAesKeyring = matProv.CreateRawAesKeyring(createRawAesKeyringInput);

// 2. Create the AWS KMS keyring.
//    We create a MRK multi keyring, as this interface also supports
//    single-region KMS keys,
//    and creates the KMS client for us automatically.
var createAwsKmsMrkMultiKeyringInput = new CreateAwsKmsMrkMultiKeyringInput
{
    Generator = keyArn
};
var awsKmsMrkMultiKeyring =
 matProv.CreateAwsKmsMrkMultiKeyring(createAwsKmsMrkMultiKeyringInput);
```

Rust

```
// 1. Create the raw AES keyring
let mpl_config = MaterialProvidersConfig::builder().build()?;
let mpl = mpl_client::Client::from_conf(mpl_config)?;

let raw_aes_keyring = mpl
    .create_raw_aes_keyring()
    .key_name("AES_256_012")
```

```
    .key_namespace("HSM_01")
    .wrapping_key(aes_key_bytes)
    .wrapping_alg(AesWrappingAlg::AlgAes256GcmIv12Tag16)
    .send()
    .await?;

// 2. Create the AWS KMS keyring
let aws_kms_mrk_multi_keyring = mpl
    .create_aws_kms_mrk_multi_keyring()
    .generator(key_arn)
    .send()
    .await?;
```

接下来创建多重密钥环并指定其生成器密钥环（如果有）。在此示例中，我们创建了一个多密钥环，其中密钥环是生成器 AWS KMS 密钥环，AES 密钥环是子密钥环。

Java

Java `CreateMultiKeyringInput` 构造函数允许您定义生成器密钥环和子密钥环。生成的 `createMultiKeyringInput` 对象不可变。

```
final CreateMultiKeyringInput createMultiKeyringInput =
 CreateMultiKeyringInput.builder()
        .generator(awsKmsMrkMultiKeyring)
        .childKeyrings(Collections.singletonList(rawAesKeyring))
        .build();
IKeyring multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);
```

C# / .NET

.NET `CreateMultiKeyringInput` 构造函数允许您定义生成器密钥环和子密钥环。生成的 `CreateMultiKeyringInput` 对象不可变。

```
var createMultiKeyringInput = new CreateMultiKeyringInput
{
    Generator = awsKmsMrkMultiKeyring,
    ChildKeyrings = new List<IKeyring> { rawAesKeyring }
};
var multiKeyring = matProv.CreateMultiKeyring(createMultiKeyringInput);
```

Rust

```
let multi_keyring = mpl
    .create_multi_keyring()
    .generator(aws_kms_mrk_multi_keyring)
    .child_keyrings(vec![raw_aes_keyring.clone()])
    .send()
    .await?;
```

现在，您就可以使用此多重密钥环加密和解密数据了。

可搜索的加密

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

可搜索加密使您无需解密整个数据库即可搜索加密的记录。该操作使用信标完成，信标在写入字段的明文值和实际存储在数据库中的加密值之间创建映射。 AWS 数据库加密 SDK 将信标存储在添加到记录中的新字段中。根据您使用的信标类型，您可以对加密的数据执行完全匹配搜索或更为自定义的复杂查询。

> **Note**
>
> AWS 数据库加密 SDK 中的可搜索加密不同于学术研究中定义的可搜索对称加密，例如可搜索的对称加密。

信标是一种截断的 HMAC 散列消息认证码（HMAC）标签，用于在字段的明文值和加密值之间创建映射。当您向配置为可搜索加密的加密字段写入新值时， AWS 数据库加密 SDK 会根据纯文本值计算 HMAC。此 HMAC 输出与该字段的明文值进行一对一（1:1）匹配。HMAC 输出会被截断，以便多个不同的明文值映射到同一个被截断的 HMAC 标签中。这些误报限制了未经授权的用户识别有关明文值的区别信息的能力。当您查询信标时， AWS 数据库加密 SDK 会自动筛选掉这些误报，并返回查询的明文结果。

为每个信标生成的平均误报数通过截断后剩余的信标长度决定。如需帮助确定适合您的实现的信标长度，请参阅确定信标长度。

> **Note**
>
> 可搜索加密旨在在未填充的新数据库中实现。在现有数据库中配置的任何信标都只能映射上传到数据库的新记录，信标无法映射现有的数据。

主题

- 可搜索的加密场景

信标是否适合我的数据集？

使用信标对加密的数据执行查询会降低与客户端加密数据库相关的性能成本。当您使用信标时，查询的效率和泄露的有关数据分布的信息量之间存在内在权衡。信标不会更改字段的加密状态。使用 AWS 数据库加密 SDK 对字段进行加密和签名时，该字段的纯文本值永远不会暴露给数据库。数据库将存储字段的随机加密值。

信标与计算信标所依据的加密字段一同存储。这意味着，即使未经授权的用户无法查看加密字段的明文值，他们也可以对信标进行统计分析，以了解有关数据集分布的更多信息，并在极端情况下识别信标映射到的明文值。配置信标的方式可以缓解这些风险。特别是，选择适当的信标长度可以帮助您保护数据集的机密性。

安全与性能

- 信标长度越短，越能保证安全性。
- 信标长度越长，越能保证性能。

可搜索的加密可能无法为所有数据集同时提供所需的性能和安全级别。配置任何信标之前，请查看您的威胁模型、安全要求和性能需求。

确定可搜索加密是否适合您的数据集时，请考虑以下数据集唯一性要求。

分布

信标所保证的安全程度取决于数据集的分布。将加密字段配置为可搜索加密时， AWS 数据库加密 SDK 会根据写入该字段的纯文本值计算 HMAC。为给定字段计算的所有信标均使用相同的密钥计算，但多租户数据库除外，它们对每个租户使用不同的密钥。这意味着，如果多次向该字段写入相同的明文值，则会为该明文值的每个实例创建相同的 HMAC 标签。

您应避免使用包含非常常见值的字段构造信标。例如，考虑一个存储伊利诺伊州每位居民地址的数据库。如果您从加密的 `City` 字段构建信标，则由于居住在芝加哥的伊利诺伊州人口比例很大，因此按“芝加哥”计算的信标将被过度代表。即使未经授权的用户只能读取加密的值和信标值，如果信标保留了此分布，它们也可能能够识别哪些记录包含芝加哥居民的数据。为了最大限度地减少所泄露的有关您的分布的区分信息量，您必须充分截断信标。隐藏这种不均匀分布所需的信标长度会带来巨大的性能成本，可能无法满足您的应用程序需求。

您必须仔细分析数据集的分布，以确定需要将信标截断多少。截断后剩余的信标长度与可以识别的关于分布的统计信息量直接相关。您可能需要选择较短的信标长度，以充分地最大限度减少泄露的有关数据集的区分信息量。

在极端情况下，您无法为分布不均匀的数据集计算信标长度，以有效地平衡性能和安全性。例如，您不应使用存储罕见疾病医学检查结果的字段构造信标。由于 NEGATIVE 结果预计在数据集中会更加普遍，因此可以很容易地通过 POSITIVE 结果的罕见程度来识别结果。当字段只包含两个可能的值时，隐藏分布非常困难。如果您使用的信标长度足够短，可以隐藏分布，则所有明文值都会映射到相同的 HMAC 标签。如果您使用更长的信标长度，则可以很明显看到哪些信标会映射到明文 POSITIVE 值。

相关性

强烈建议您避免使用具有相关值的字段构造不同的信标。使用相关字段构造的信标需要更短的信标长度，以充分地最大限度减少向未经授权的用户泄露的有关每个数据集分布的信息量。您必须仔细分析数据集，包括它的熵和相关值的联合分布，以确定需要将信标截断多少。如果产生的信标长度不能满足您的性能需求，则信标可能不适合您的数据集。

例如，您不应使用 City 和 ZIPCode 字段构造两个单独的信标，因为邮政编码可能只与一个城市相关联。通常，信标生产生误报会限制未经授权的用户识别有关您的数据集的区分信息的能力。但是 City 和 ZIPCode 字段之间的关联意味着未经授权的用户可以轻松识别哪些结果是误报，并区分不同的邮政编码。

您还应避免使用包含相同明文值的字段构造信标。例如，您不应使用 mobilePhone 和 preferredPhone 字段构造信标，因为它们可能具有相同的值。如果您从两个字段构造不同的信标，则 AWS 数据库加密 SDK 会使用不同的密钥为每个字段创建信标。这会为相同的明文值生成两个不同的 HMAC 标签。这两个不同的信标不太可能产生相同的误报，且未经授权的用户可能能够区分不同的电话号码。

即使您的数据集包含相关字段或具有不均匀的分布，您也可以使用较短的信标长度来构造能够保护数据集机密性的信标。但是，信标长度并不能保证数据集中的每个唯一值都会产生大量误报，从而有效地最大限度减少泄露的有关数据集的区分信息量。信标长度仅能估计产生的误报平均数。数据集分布越不均匀，信标长度在确定产生的误报平均数量方面的有效性就越低。

请仔细考虑您构造信标所依据的字段的分布，并考虑需要将信标长度截断多少才能满足您的安全要求。本章节中的以下主题假设您的信标分布均匀，并且不包含相关数据。

可搜索的加密场景

下面的示例演示了一种简单的可搜索加密解决方案。在应用中，本示例中使用的示例字段可能不符合信标的分布和关联唯一性建议。在阅读本章节中的可搜索加密概念时，您可以将此示例作为参考。

以一个名为 Employees 的跟踪公司员工数据的数据库为例。数据库中的每条记录都包含名为 employeeID LastName、FirstName、和 “地址” 的字段。Employees 数据库中的每个字段都由主键 EmployeeID 标识。

以下是数据库中的明文记录示例。

```
{
    "EmployeeID": 101,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}
```

如果您在加密操作中将 LastName 和 FirstName 字段标记为 ENCRYPT_AND_SIGN，则这些字段中的值在上传到数据库之前会在本地进行加密。上传的已加密数据是完全随机的，数据库无法将这些数据识别为受保护。它只检测典型的数据条目。这意味着，实际存储在数据库中的记录可能如下所示。

```
{
    "PersonID": 101,
    "LastName": "1d76e94a2063578637d51371b363c9682bad926cbd",
    "FirstName": "21d6d54b0aaabc411e9f9b34b6d53aa4ef3b0a35",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}
```

如果您需要在数据库中查询LastName字段中的精确匹配项，请配置一个名为的标准信标，LastName以将写入该LastName字段的纯文本值映射到存储在数据库中的加密值。

该信标根据字段 HMACs 中的纯文本值进行LastName计算。每个 HMAC 输出都被截断，因此不再与明文值完全匹配。例如，Jones 的完整哈希值和截断后的哈希值可能如下所示。

完整哈希值

2aa4e9b404c68182562b6ec761fcca5306de527826a69468885e59dc36d0c3f824bdd44cab45526f

截断后的哈希值

b35099d408c833

配置标准信标后，您可以在 LastName 字段上执行相等搜索。例如，如果要搜索Jones，请使用LastName信标执行以下查询。

```
LastName = Jones
```

AWS 数据库加密 SDK 会自动过滤掉误报并返回查询的纯文本结果。

信标

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

信标是一种截断的 HMAC 散列消息认证码（HMAC）标签，用于在写入字段的明文值和实际存储在数据库中的加密值之间创建映射。信标不会更改字段的加密状态。信标根据字段的明文值计算 HMAC，并将其与加密值一起存储。此 HMAC 输出与该字段的明文值进行一对一（1:1）匹配。HMAC 输出会被截断，以便多个不同的明文值映射到同一个被截断的 HMAC 标签中。这些误报限制了未经授权的用户识别有关明文值的区别信息的能力。

信标只能由您的加密操作中标有ENCRYPT_AND_SIGNSIGN_ONLY、或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT的字段构建。信标本身未经过签名或加密。您无法使用标记为 DO_NOTHING 的字段构造信标。

您配置的信标类型决定了您能够执行的查询类型。支持可搜索加密的信标类型有两种。标准信标执行相等搜索。复合信标则组合文字明文字符串和标准信标来执行复杂的数据库操作。配置信标后，必须先为每个信标配置二级索引，然后才能搜索加密的字段。有关更多信息，请参阅 通过使用信标配置二级索引。

主题

标准信标

标准信标是在数据库中实现可搜索加密的最简单方法。他们只能对单个加密字段或虚拟字段执行相等搜索。要了解如何配置标准信标，请参阅配置标准信标。

构造标准信标所依据的字段称为信标源。它用于标识信标需要映射的数据的位置。信标源可以是加密的字段，也可以是虚拟字段。每个标准信标中的信标源都必须是唯一的。您不能使用相同的信标源来配置两个信标。

标准信标可用于对加密或虚拟字段执行相等搜索。或者，它们可以用来构造复合信标以执行更复杂的数据库操作。为了帮助您组织和管理标准信标， AWS 数据库加密 SDK 提供了以下可选信标样式，用于定义标准信标的预期用途。有关更多信息，请参阅定义信标样式。

您可以创建对单个加密字段执行相等搜索的标准信标，也可以通过创建虚拟字段来创建对多个ENCRYPT_AND_SIGNSIGN_ONLY、和SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT字段的串联执行相等搜索的标准信标。

虚拟字段

虚拟字段是由一个或多个源字段构成的概念字段。创建虚拟字段不会向记录中写入新字段。虚拟字段未明确存储在数据库中。它用于标准信标配置中，从而向信标提供有关如何识别字段的特定片段或连接记录中的多个字段以执行特定查询的指令。一个虚拟字段需要至少一个加密字段。

> **Note**
>
> 以下示例演示了您可以使用虚拟字段执行的转换和查询的类型。在应用中，本示例中使用的示例字段可能不符合信标的分布和关联唯一性建议。

例如，如果要对 `FirstName` 和 `LastName` 字段的连接执行相等搜索，您可以创建以下虚拟字段之一。

- 一个虚拟 NameTag 字段，由 FirstName 字段的第一个字母后跟 LastName 字段构造而成，全部使用小写字母。此虚拟字段使您能够查询 NameTag=mjones。
- 一个虚拟 LastFirst 字段，由 LastName 字段后跟 FirstName 字段构造而成。此虚拟字段使您能够查询 LastFirst=JonesMary。

或者，如果您要对加密字段的特定片段执行相等搜索，请创建一个用于标识您要查询的片段的虚拟字段。

例如，如果您要使用 IP 地址的前三个片段查询加密的 IPAddress 字段，请创建以下虚拟字段。

- 一个虚拟 IPSegment 字段，由 Segments('.', 0, 3) 构造而成。此虚拟字段使您能够查询 IPSegment=192.0.2。该查询返回 IPAddress 值以"192.0.2"开头的所有记录。

虚拟字段必须是唯一的。不能用完全相同的源字段构造两个虚拟字段。

如需帮助配置虚拟字段和使用虚拟字段的信标，请参阅创建虚拟字段。

复合信标

复合信标创建的索引可以提高查询性能，并且使您能够执行更复杂的数据库操作。您可以使用复合信标组合文字明文字符串和标准信标来对加密记录执行复杂的查询，例如从单个索引中查询两种不同的记录类型或使用排序键查询字段组合。有关更多复合信标解决方案示例，参阅选择信标类型。

复合信标可以由标准信标或标准信标和带符号字段的组合构建。它们由各部分的列表构造。所有复合信标都应包括用于识别信标中包含的 ENCRYPT_AND_SIGN 字段的加密部分列表。每个 ENCRYPT_AND_SIGN 字段都必须用标准信标识别。更复杂的复合信标还可能包括标识信标中包含的纯文本SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT字段的签名部分列表，以及标识复合信标组装字段的所有可能方式的构造器部分列表。

> **Note**
>
> AWS 数据库加密 SDK 还支持签名信标，这些信标可以完全通过纯文本SIGN_ONLY和字段进行配置。SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT签名信标是一种复合信标，用于对已签名但未加密的字段进行索引和执行复杂查询。有关更多信息，请参阅 创建签名的信标。

如需帮助配置复合信标，请参阅配置复合信标。

您配置复合信标的方式决定了您可以执行的查询类型。举例来说，您可以将一些已加密和签名的部分设为可选，以提高查询的灵活性。有关复合信标可以执行的查询类型的更多信息，请参阅 查询信标。

计划信标

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

信标旨在在未填充的新数据库中实现。在现有数据库中配置的任何信标将只会映射写入数据库的新记录。信标是根据字段的明文值计算出来的，一旦字段被加密，信标就无法映射现有数据。使用信标写入新记录后，您将无法更新信标的配置。但是，您可以为添加到记录中的新字段添加新信标。

要实现可搜索的加密，必须使用 AWS KMS 分层密钥环来生成、加密和解密用于保护记录的数据密钥。有关更多信息，请参阅 使用分层密钥环进行可搜索加密。

在为可搜索加密配置信标之前，您需要查看加密要求、数据库访问模式和威胁模型，以确定适合您的数据库的最佳解决方案。

您配置的信标类型决定了您可以执行的查询类型。您在标准信标配置中指定的信标长度决定了给定信标产生的预期误报数量。强烈建议您在配置信标之前确定并计划需要执行的查询类型。一旦您使用了信标，就无法更新配置。

强烈建议您在配置任何信标之前查看并完成以下任务。

- 确定信标是否适合您的数据集
- 选择信标类型
- 选择信标长度
- 选择信标名称

在为数据库计划可搜索的加密解决方案时，请记住以下信标唯一性要求。

- 每个标准信标都必须具有唯一的信标源

  不能通过同一个加密字段或虚拟字段构造多个标准信标。

  但是，单个标准信标却可用于构造多个复合信标。
- 避免利用与现有标准信标重叠的源字段创建虚拟字段

  通过包含用于创建另一个标准信标的源字段的虚拟字段构造标准信标会同时降低两个信标的安全性。

  有关更多信息，请参阅 虚拟字段的安全考虑因素。

多租户数据库的考虑因素

要查询在多租户数据库中配置的信标，必须包含用于存储与在查询中加密记录的租户关联的 branch-key-id 的字段。在定义信标密钥源时定义此字段。要使查询成功，此字段中的值必须确定重新计算信标所需的相应信标密钥材料。

在配置信标之前，必须决定如何计划在查询中包含 branch-key-id。有关在查询中包含 branch-key-id 的不同方式的更多信息，请参阅 查询多租户数据库中的信标。

选择信标类型

> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

使用可搜索的加密，您可以通过将加密字段中的明文值映射到信标来搜索加密记录。您配置的信标类型决定您可以执行的查询类型。

强烈建议您在配置信标之前确定并计划需要执行的查询类型。配置信标后，必须先为每个信标配置二级索引，然后才能搜索加密的字段。有关更多信息，请参阅 通过使用信标配置二级索引。

信标在写入字段的明文值和实际存储在数据库中的加密值之间创建映射。您无法比较两个标准信标的值，即使其中包含相同的底层明文。两个标准信标将为相同的明文值生成两个不同的 HMAC 标签。因此，标准信标无法执行以下查询。

- *beacon1* = *beacon2*
- *beacon1* IN (*beacon2*)
- *value* IN (*beacon1*, *beacon2*, ...)
- CONTAINS(*beacon1*, *beacon2*)

只有比较复合信标的签名部分，您才能执行上述查询，但 CONTAINS 运算符除外，您可以将其与复合信标一起使用，以识别组合信标所包含的加密或签名字段的完整值。比较签名的部分时，您可以选择包含加密部分的前缀，但不能包括字段的加密值。有关标准信标和复合信标可以执行的查询类型的更多信息，参阅查询信标。

查看数据库访问模式时，请考虑以下可搜索的加密解决方案。以下示例定义了满足不同的加密和查询要求需要配置哪个信标。

标准信标

标准信标只能执行相等搜索。您可以使用标准信标执行以下查询。

查询单个已加密字段

如果您要识别包含加密字段特定值的记录，请创建标准信标。

示例

在以下示例中，考虑一个名为 UnitInspection 的数据库，该数据库用于跟踪生产设施的检查数据。数据库中的每条记录都包含名为 work_id、inspection_date、inspector_id_last4 和 unit 的字段。完整的检查人员 ID 是一个介于 0 到 99999999 之间的数字。但是，为了确保数据集的均匀分布，inspector_id_last4 只存储检查人员 ID 的最后四位数字。数据库中的每个字段都由主键 work_id 标识。inspector_id_last4 和 unit 字段在加密操作中被标记为 ENCRYPT_AND_SIGN。

以下是 UnitInspection 数据库中明文条目的示例。

```
{
    "work_id": "1c7fcff3-6e74-41a8-b7f7-925dc039830b",
    "inspection_date": 2023-06-07,
    "inspector_id_last4": 8744,
    "unit": 229304973450
}
```

查询记录中的单个加密字段

> 如果需要对 inspector_id_last4 字段进行加密，但您仍需要查询它以获得完全匹配的字段，请通过 inspector_id_last4 字段构造一个标准信标。然后，请使用标准信标创建二级索引。您可以使用此二级索引来查询加密的 inspector_id_last4 字段。

要获得配置标准信标的帮助，请参阅配置标准信标。

查询虚拟字段

虚拟字段是由一个或多个源字段构成的概念字段。如果要对加密字段的特定分段执行相等搜索，或者对多个字段的连接执行相等搜索，请使用虚拟字段构造标准信标。所有虚拟字段都必须至少包括一个加密的源字段。

示例

以下示例为 Employees 数据库创建虚拟字段。以下是 Employees 数据库中的明文记录示例。

```
{
    "EmployeeID": 101,
    "SSN": 000-00-0000,
    "LastName": "Jones",
    "FirstName": "Mary",
    "Address": {
                "Street": "123 Main",
                "City": "Anytown",
                "State": "OH",
                "ZIPCode": 12345
    }
}
```

查询加密字段的某个分段

在本示例中，SSN 字段已加密。

如果要使用社会保障号码的最后四位数字查询 SSN 字段，则请创建一个虚拟字段来标识您计划查询的分段。

使用 Suffix(4) 构造的虚拟 Last4SSN 字段构造便于您查询 Last4SSN=0000。使用此虚拟字段来构造标准信标。然后，请使用标准信标创建二级索引。您可以使用此二级索引在虚拟字段上进行查询。此查询返回 SSN 值以您指定的最后四位数字结尾的所有记录。

查询多个字段的连接

> **Note**
>
> 以下示例演示了您可以使用虚拟字段执行的转换和查询的类型。在应用中，本示例中使用的示例字段可能不符合信标的分布和关联唯一性建议。

如果要对 FirstName 和 LastName 字段的连接执行相等搜索，则可以创建一个虚拟 NameTag 字段，该字段由 FirstName 字段的第一个字母后跟 LastName 字段组成，全部用小写字母。使用此虚拟字段来构造标准信标。然后，请使用标准信标创建二级索引。您可以使用此二级索引在虚拟字段上查询 NameTag=mjones。

必须至少对其中一个源字段进行加密。可以加密 FirstName 或 LastName 中的一个，或者同时加密此两者。任何纯文本源字段都必须标记为SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT在您的加密操作中。

如需帮助配置虚拟字段和使用虚拟字段的信标，请参阅创建虚拟字段。

复合信标

复合信标根据文字明文字符串和标准信标创建索引，以执行复杂的数据库操作。您可以使用复合信标执行以下查询。

在单个索引上查询加密字段组合

如果您需要在单个索引上查询加密字段的组合，请创建一个复合信标，该信标将为每个加密字段构造的各个标准信标组合在一起，形成一个索引。

配置复合信标后，您可以创建一个二级索引，以将复合信标指定为分区键来执行完全匹配查询，或者使用排序键来执行更复杂的查询。将复合信标指定为排序键的二级索引可以执行完全匹配查询和更为自定义的复杂查询。

示例

在以下示例中，考虑一个名为 UnitInspection 的数据库，该数据库用于跟踪生产设施的检查数据。数据库中的每条记录都包含名为 work_id、inspection_date、inspector_id_last4 和 unit 的字段。完整的检查人员 ID 是一个介于 0 到 99999999 之间的数字。但是，为了确保数据集的均匀分布，inspector_id_last4 只存储检查人员 ID 的最后四位数字。数据库中的每个字段都由主键 work_id 标识。inspector_id_last4 和 unit 字段在加密操作中被标记为 ENCRYPT_AND_SIGN。

以下是 UnitInspection 数据库中明文条目的示例。

```
{
    "work_id": "1c7fcff3-6e74-41a8-b7f7-925dc039830b",
    "inspection_date": 2023-06-07,
    "inspector_id_last4": 8744,
    "unit": 229304973450
}
```

对已加密字段的组合执行相等搜索

如果要查询 UnitInspection 数据库以获得 inspector_id_last4.unit 上的完全匹配项，请先为 inspector_id_last4 和 unit 字段创建不同的标准信标。然后，通过两个标准信标创建一个复合信标。

配置复合信标后，创建一个二级索引，以将复合信标指定为分区键。使用此二级索引查询 inspector_id_last4.unit 上的完全匹配项。例如，您可以查询此信标以查找检查人员对给定单位执行的检查列表。

对已加密字段的组合执行复杂查询

如果要在 inspector_id_last4 和 inspector_id_last4.unit 上查询 UnitInspection 数据库，请先为 inspector_id_last4 和 unit 字段创建不同的标准信标。然后，通过两个标准信标创建一个复合信标。

配置复合信标后，创建二级索引，以将复合信标指定为排序键。使用此二级索引查询 UnitInspection 数据库以获得以特定检查人员开头的条目，或者查询数据库以获得特定单位 ID 范围内由特定检查人员检查过的所有单元的列表。您还可以在 inspector_id_last4.unit 上执行完全匹配搜索。

如需帮助配置复合信标，请参阅配置复合信标。

在单个索引上查询加密字段和明文字段的组合

如果您需要在单个索引上查询加密字段和明文字段的组合，请创建一个复合信标，该信标将各个标准信标和明文字段组合在一起，形成一个索引。用于构造复合信标的纯文本字段必须标记，SIGN_ONLY或者SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT在您的加密操作中标记。

配置复合信标后，您可以创建一个二级索引，以将复合信标指定为分区键来执行完全匹配查询，或者使用排序键来执行更复杂的查询。将复合信标指定为排序键的二级索引可以执行完全匹配查询和更为自定义的复杂查询。

示例

在以下示例中，考虑一个名为 UnitInspection 的数据库，该数据库用于跟踪生产设施的检查数据。数据库中的每条记录都包含名为 work_id、inspection_date、inspector_id_last4 和 unit 的字段。完整的检查人员 ID 是一个介于 0 到 99999999 之间的数字。但是，为了确保数据集的均匀分布，inspector_id_last4 只存储检查人员 ID 的最后四位数字。数据库中的每个字段都由主键 work_id 标识。inspector_id_last4 和 unit 字段在加密操作中被标记为 ENCRYPT_AND_SIGN。

以下是 UnitInspection 数据库中明文条目的示例。

```
{
    "work_id": "1c7fcff3-6e74-41a8-b7f7-925dc039830b",
    "inspection_date": 2023-06-07,
    "inspector_id_last4": 8744,
    "unit": 229304973450
}
```

对字段的组合执行相等搜索

如果要查询 UnitInspection 数据库以获得特定检查人员在特定日期进行的检查，请先为 inspector_id_last4 字段创建标准信标。inspector_id_last4 字段在加密操作中被标记为 ENCRYPT_AND_SIGN。所有已加密部分都需要自己的标准信标。inspection_date 字段被标记为 SIGN_ONLY，且不需要标准信标。接下来，从 inspection_date 字段和 inspector_id_last4 标准信标创建复合信标。

配置复合信标后，创建一个二级索引，以将复合信标指定为分区键。使用此二级索引查询数据库以获得与特定检查人员和检查日期完全匹配的记录。例如，您可以查询数据库以获取 ID 结尾为 8744 的检查人员在特定日期进行的所有检查的列表。

对字段的组合执行复杂查询

如果要查询数据库以获得在 inspection_date 范围内进行的检查，或者查询数据库以获得受 inspector_id_last4 或 inspection_date 约束的在特定 inspector_id_last4.unit 进行的检查，请先为 inspector_id_last4 和 unit 字段创建不同的标准信标。然后，从明文 inspection_date 字段和两个标准信标创建复合信标。

配置复合信标后，创建二级索引，以将复合信标指定为排序键。使用此二级索引对特定检查人员在特定日期进行的检查执行查询。例如，您可以查询数据库中以获得同一日期检查的所有单位的列表。或者，您可以查询数据库以获取在给定的检查日期范围间对特定单位进行的所有检查的列表。

如需帮助配置复合信标，请参阅配置复合信标。

选择信标长度

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

当您向配置为可搜索加密的加密字段写入新值时，AWS 数据库加密 SDK 会根据纯文本值计算 HMAC。此 HMAC 输出与该字段的明文值进行一对一（1:1）匹配。HMAC 输出会被截断，以便多个不同的明文值映射到同一个被截断的 HMAC 标签中。这些碰撞或误报限制了未经授权的用户识别有关明文值的区别信息的能力。

为每个信标生成的平均误报数通过截断后剩余的信标长度决定。配置标准信标时，只需要定义信标长度。复合信标使用构造它们的标准信标的信标长度。

信标不会更改字段的加密状态。但是，当您使用信标时，查询的效率和泄露的有关数据分布的信息量之间存在内在权衡。

可搜索加密的目标在于，使用信标对加密数据执行查询，从而降低与客户端加密数据库相关的性能成本。信标与计算信标所依据的加密字段一同存储。这意味着，它们可以揭示有关您的数据集分布的区别信息。在极端情况下，未经授权的用户可能能够分析披露的有关您的分布的信息，并用它来识别字段的明文值。选择适当的信标长度可以帮助降低这些风险并保护分发的机密性。

查看您的威胁模型，以确定所需的安全级别。例如，有权访问您的数据库但不应访问明文数据的人越多，您就越想保护数据集分布的机密性。为了提高机密性，信标需要产生更多的误报。提高机密性会导致查询性能降低。

安全与性能

- 信标长度过长会导致产生的误报太少，并且可能会泄露有关数据集分布的区别信息。
- 信标长度过短会导出产生的误报太多，并且会增加查询的性能成本，因为它需要对数据库进行更广泛的扫描。

在确定适合解决方案的信标长度时，必须找到一个能够充分保护数据安全性且不会对查询性能产生超出绝对必要影响的长度。信标保留的安全程度取决于数据集的分布以及构造信标所依据的字段的相关性。以下主题假设您的信标分布均匀，并且其中不包含相关数据。

主题

计算信标长度

信标长度以比特为单位进行定义，是指截断后保留的 HMAC 标签的位数。建议的信标长度因数据集的分布、相关值的存在以及您的特定安全和性能要求而异。如果您的数据集分布均匀，您可以使用以下方

程和过程来帮助确定最适合您的实现的信标长度。这些方程仅会估计信标将产生的平均误报数，并不能保证数据集中的每个唯一值都会产生特定数量的误报。

> **Note**
>
> 这些方程的有效性则取决于数据集的分布。如果您的数据集分布不均匀，请参阅 信标是否适合我的数据集？。
> 通常情况下，您的数据集离均匀分布越远，您就越需要缩短信标长度。

1.

估算总量

总量是指构造标准信标所依据的字段中的唯一值的预期数量，而不是字段中存储的值的预期总数。例如，考虑一个用于标识员工会议地点的加密 Room 字段。Room 字段预计将存储 100000 个总值，但员工只能预留 50 个不同的会议室用于会议。这意味着总量为 50，因为 Room 字段中只能存储 50 个可能的唯一值。

> **Note**
>
> 如果您的标准信标由虚拟字段构造，则用于计算信标长度的总量是虚拟字段创建的唯一组合数。

在估算总量时，请务必考虑数据集的预计增长。使用信标写入新记录后，您将无法更新信标长度。查看您的威胁模型和任何现有的数据库解决方案，以估算您预计该字段在未来五年内将存储的唯一值的数量。

您的总量不需要很精确。首先，确定当前数据库中唯一值的数量，或者估算第一年预计存储的唯一值的数量。接下来，通过以下问题来帮助您确定未来五年内唯一值的预计增长情况。

- 您是否期望唯一值乘以 10？
- 您是否期望唯一值乘以 100？
- 您是否期望唯一值乘以 1000？

50000 和 60000 个唯一值之间的差异并不显著，两者都将产生相同的建议信标长度。但是，50000 和 500000 个唯一值之间的差异将显著影响建议的信标长度。

考虑查看常见数据类型（例如邮政编码或姓氏）出现频率的相关公共数据。举例来说，美国有 41707 个邮政编码。您使用的总量应与您自己的数据库成正比。如果数据库中的 ZIPCode 字段包含来自整个美国的数据，则即使 ZIPCode 字段当前没有 41707 个唯一值，也可以将总量定义为 41707。如果数据库中的 ZIPCode 字段仅包含来自单个州的数据，并且到目前为止仅包含来自单一州的数据，则可以将总量定义为该州的邮政编码总数，而不是 41704 个。

2. 计算建议的预期碰撞次数范围

   要确定给定字段的适当信标长度，您必须首先确定预期碰撞次数的适当范围。预期的碰撞次数表示映射到特定 HMAC 标签的唯一明文值的平均预期数量。一个唯一的明文值的预期误报数比预期的碰撞次数少一。

   建议预期的碰撞次数大于或等于二，且小于总量的平方根。只有当您的总量具有 16 个或更多唯一值时，以下方程才有效。

   ```
   2 ≤ number of collisions < √(Population)
   ```

   如果碰撞次数少于两次，则信标产生的误报数将会太少。建议将两个作为预期碰撞的最小数量，因为这意味着，平均来说，字段中的每个唯一值都会通过映射到另一个唯一值来产生至少一个误报。

3. 计算信标长度的建议范围

   确定预期碰撞的最小和最大次数后，使用以下公式来确定适当信标长度的范围。

   $$\text{number of collisions} = \text{Population} * 2^{-(\text{beacon length})}$$

   首先，求解信标长度，其中预期碰撞次数等于二（建议的最小预期碰撞数）。

   $$2 = \text{Population} * 2^{-(\text{beacon length})}$$

   然后，求解信标长度，其中预期碰撞次数等于总量的平方根（建议的最小预期碰撞数）。

   $$\sqrt{(\text{Population})} = \text{Population} * 2^{-(\text{beacon length})}$$

   建议将此方程产生的输出向下舍入到较短的信标长度。举例来说，如果方程产生的信标长度为 15.6，则建议将该值向下舍入到 15 位，而不是四舍五入到 16 位。

4. 选择信标长度

这些方程仅确定您的字段的建议信标长度范围。建议尽量使用较短的信标长度，以保护数据集的安全性。但是，您实际使用的信标长度却由您的威胁模型决定。在审查威胁模型以确定字段的最佳信标长度时，请考虑您的性能要求。

使用较短的信标长度会降低查询性能，而使用较长的信标长度则会降低安全性。通常，如果您的数据集分布不均匀，或者您根据相关字段构造不同的信标，则需要使用较短的信标长度来最大限度地减少泄露的有关数据集分布的信息量。

如果您查看威胁模型，并确定泄露的有关字段分布的任何区别信息不会对您的整体安全构成威胁，则可以选择使用比您计算的建议范围更长的信标长度。例如，如果将某个字段的建议信标长度范围计算为 9—16 位，则可以选择使用 24 位的信标长度来避免任何性能损失。

请谨慎选择信标长度。使用信标写入新记录后，您将无法更新信标长度。

示例

假设一个在加密操作中将 `unit` 字段标记为 `ENCRYPT_AND_SIGN` 的数据库。要为 `unit` 字段配置标准信标，我们需要确定 `unit` 字段的预期误报数量和信标长度。

1. 估算总量

   在审查了威胁模型和当前的数据库解决方案之后，预计 `unit` 字段最终将有 100000 个唯一值。

   这意味着总量 = 100000。

2. 计算预期碰撞次数的建议范围。

   在此示例中，预期的碰撞次数应当介于 2—316 之间。

   ```
   2 ≤ number of collisions < √(Population)
   ```

   a.
   ```
   2 ≤ number of collisions < √(100,000)
   ```

   b.
   ```
   2 ≤ number of collisions < 316
   ```

3. 计算信标长度的建议范围。

   在本示例中，信标长度应当介于 9–16 位之间。

```
number of collisions = Population * 2^-(beacon length)
```

a. 计算信标长度，其中预期的碰撞次数等于步骤 2 中确定的最小值。

```
2 = 100,000 * 2^-(beacon length)
```

信标长度 = 15.6 或 15 位

b. 计算信标长度，其中预期的碰撞次数等于步骤 2 中确定的最大值。

```
316 = 100,000 * 2^-(beacon length)
```

信标长度 = 8.3 或 8 位

4. 确定适合您的安全和性能要求的信标长度。

对于低于 15 的每一个位，性能成本和安全性会翻一番。

- 16 位
  - 平均而言，每个唯一值将映射到 1.5 个其他单位。
  - 安全性：具有相同的截断 HMAC 标签的两条记录具有相同明文值的可能性为 66%。
  - 性能：查询将针对您实际请求的每 10 条记录检索 15 条记录。
- 14 位
  - 平均而言，每个唯一值将映射到 6.1 个其他单位。
  - 安全性：具有相同的截断 HMAC 标签的两条记录具有相同明文值的可能性为 33%。
  - 性能：查询将针对您实际请求的每 10 条记录检索 30 条记录。

选择信标名称

> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

每个信标都由唯一的信标名称标识。配置信标后，信标名称就是您在查询加密的字段时使用的名称。信标名称可以与加密的字段或虚拟字段相同，但不能与未加密的字段相同。两个不同的信标不能具有相同的信标名称。

有关演示如何命名和配置信标的示例，请参阅配置信标。

命名标准信标

命名标准信标时，强烈建议尽可能将您的信标名称解析为信标源。这意味着信标名称和构造标准信标所依据的加密字段或虚拟字段的名称是相同的。例如，如果您要为名为 `LastName` 的加密字段创建标准信标，则您的信标名称也应当为 `LastName`。

当您的信标名称与信标源相同时，您可以从配置中省略信标源， AWS 数据库加密 SDK 将自动使用信标名称作为信标源。

配置信标

> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

支持可搜索加密的信标类型有两种。标准信标执行相等搜索。它们是在数据库中实现可搜索加密的最简单方法。复合信标则组合文字明文字符串和标准信标来执行更复杂的查询。

信标旨在在未填充的新数据库中实现。在现有数据库中配置的任何信标将只会映射写入数据库的新记录。信标是根据字段的明文值计算出来的，一旦字段被加密，信标就无法映射现有数据。使用信标写入新记录后，您将无法更新信标的配置。但是，您可以为添加到记录中的新字段添加新信标。

确定访问模式后，配置信标应该是数据库实现的第二步。然后，在配置所有信标之后，您需要创建AWS KMS 分层密钥环、定义信标版本、为每个信标配置二级索引、定义加密操作以及配置数据库和 AWS 数据库加密 SDK 客户端。有关更多信息，请参阅使用信标。

为了更便于定义信标版本，建议为标准信标和复合信标创建列表。在配置信标时，将您创建的每个信标添加到相应的标准或复合信标列表中。

主题

配置标准信标

标准信标是在数据库中实现可搜索加密的最简单方法。他们只能对单个加密字段或虚拟字段执行相等搜索。

示例配置语法

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beaconName")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# / .NET

```
var standardBeaconList = new List<StandardBeacon>();
StandardBeacon exampleStandardBeacon = new StandardBeacon
  {
    Name = "beaconName",
    Length = 10
  };
standardBeaconList.Add(exampleStandardBeacon);
```

Rust

```
let standard_beacon_list = vec![

 StandardBeacon::builder().name("beacon_name").length(beacon_length_in_bits).build()?,
```

要配置标准信标，请提供以下值。

信标名称

您在查询已加密字段时使用的名称。

信标名称可以与加密的字段或虚拟字段相同，但不能与未加密的字段相同。强烈建议尽可能使用构造标准信标所依据的加密字段或虚拟字段的名称。两个不同的信标不能具有相同的信标名称。如需帮助确定最适合您的实现的信标名称，请参阅选择信标名称。

信标长度

截断后保留的信标哈希值的位数。

信标长度决定了给定信标所产生的误报平均数。要获得确定适合您的实现的信标长度的更多信息和帮助，请参阅确定信标长度。

信标源（可选）

构造标准信标所依据的字段。

信标源必须是字段名称或引用嵌套字段值的索引。当您的信标名称与信标源相同时，您可以从配置中省略信标源， AWS 数据库加密 SDK 将自动使用该信标名称作为信标源。

创建虚拟字段

要创建虚拟字段，您必须提供虚拟字段的名称和源字段的列表。将源字段添加到虚拟部分列表的顺序决定了连接这些字段以构建虚拟字段的顺序。以下示例将两个源字段完全连接在一起以创建一个虚拟字段。

> **Note**
>
> 我们建议您在填充数据库之前验证您的虚拟字段是否产生了预期的结果。有关更多信息，请参阅测试信标输出。

Java

参见完整的代码示例：VirtualBeaconSearchableEncryptionExample.java

```
List<VirtualPart> virtualPartList = new ArrayList<>();
    virtualPartList.add(sourceField1);
    virtualPartList.add(sourceField2);

VirtualField virtualFieldName = VirtualField.builder()
    .name("virtualFieldName")
    .parts(virtualPartList)
    .build();
```

```
List<VirtualField> virtualFieldList = new ArrayList<>();
    virtualFieldList.add(virtualFieldName);
```

C# / .NET

参见完整的代码示例：VirtualBeaconSearchableEncryptionExample.cs

```
var virtualPartList = new List<VirtualPart> { sourceField1, sourceField2 };

var virtualFieldName = new VirtualField
{
    Name = "virtualFieldName",
    Parts = virtualPartList
};

var virtualFieldList = new List<VirtualField> { virtualFieldName };
```

Rust

参见完整的代码示例：virtual _beacon_searchable_encryption.rs

```
let virtual_part_list = vec![source_field_one, source_field_two];

let state_and_has_test_result_field = VirtualField::builder()
    .name("virtual_field_name")
    .parts(virtual_part_list)
    .build()?;

let virtual_field_list = vec![virtual_field_name];
```

要使用源字段的特定片段创建虚拟字段，必须先定义该转换，然后再将源字段添加到虚拟部分列表中。

虚拟字段的安全考虑因素

信标不会改变字段的加密状态。但是，当您使用信标时，查询的效率和泄露的有关数据分布的信息量之间存在内在权衡。您配置信标的方式决定了该信标所确保的安全级别。

避免利用与现有标准信标重叠的源字段创建虚拟字段。创建包含源字段（已用于创建标准信标）的虚拟字段可能会降低两个信标的安全级别。安全性的降低程度取决于其他源字段所添加的熵水平。熵水平由附加源字段中唯一值的分布以及附加源字段占虚拟字段整体大小的位数决定。

您可以使用总量和信标长度来确定虚拟字段的源字段是否确保了数据集的安全性。总量是字段中唯一值的预期数量。您的总量不需要很精确。如需帮助估算某个字段的总量，请参阅估算总量。

查看虚拟字段的安全性时，请考虑以下示例。

- Beacon1 由 `FieldA` 构造而成。`FieldA` 具有大于 $2^{(\text{Beacon1 长度})}$ 的总量。
- Beacon2 由 `VirtualField` 构造而成，而后者由 `FieldA`、`FieldB`、`FieldC` 和 `FieldD` 构造而成。`FieldB`、`FieldC` 和 `FieldD` 加在一起的总量大于 2^N

如果以下语句为真，则 Beacon2 将保留 Beacon1 和 Beacon2 的安全性：

```
N ≥ (Beacon1 length)/2
```

and

```
N ≥ (Beacon2 length)/2
```

定义信标样式

标准信标可用于对加密或虚拟字段执行相等搜索。或者，它们可以用来构造复合信标以执行更复杂的数据库操作。为了帮助您组织和管理标准信标， AWS 数据库加密 SDK 提供了以下可选信标样式，用于定义标准信标的预期用途。

> **Note**
>
> 要定义信标样式，必须使用 3.2 版或更高版本的 AWS 数据库加密 SDK。在向信标配置中添加信标样式之前，请将新版本部署给所有读者。

PartOnly

定义为的标准信标`PartOnly`只能用于定义复合信标的加密部分。您不能直接查询`PartOnly`标准信标。

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
```

```
    .name("beaconName")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .partOnly(PartOnly.builder().build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# /.NET

```
new StandardBeacon
{
    Name = "beaconName",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        PartOnly = new PartOnly()
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon_name")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::PartOnly(PartOnly::builder().build()?))
    .build()?
```

Shared

默认情况下，每个标准信标都会生成一个唯一的 HMAC 密钥用于信标计算。因此，您无法对来自两个独立标准信标的加密字段执行相等搜索。定义为的标准信标Shared使用来自另一个标准信标的 HMAC 密钥进行计算。

例如，如果您需要将beacon1字段与字段进行比较，请定义beacon2为使用来beacon2自 HMAC 密钥进行计算beacon1的Shared信标。

> **Note**
>
> 在配置任何Shared信标之前，请考虑您的安全和性能需求。 Shared信标可能会增加可以识别的有关数据集分布的统计信息量。例如，它们可能会揭示哪些共享字段包含相同的纯文本值。

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beacon2")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .shared(Shared.builder().other("beacon1").build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# /.NET

```
new StandardBeacon
{
    Name = "beacon2",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        Shared = new Shared { Other = "beacon1" }
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon2")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::Shared(
       Shared::builder().other("beacon1").build()?,
    ))
```

```
    .build()?
```

AsSet

默认情况下，如果字段值是一个集合，则 AWS 数据库加密 SDK 会计算该集合的单个标准信标。因此，您无法执行查询加密字段CONTAINS(*a*, :*value*)在*a*哪里。定义为的标准信标AsSet计算集合中每个单独元素的单个标准信标值，并将信标值作为集合存储在项目中。这样， AWS 数据库加密 SDK 就可以执行查询CONTAINS(*a*, :*value*)。

要定义AsSet标准信标，集合中的元素必须来自相同的总体，这样它们才能使用相同的信标长度。如果在计算信标值时发生冲突，则信标集的元素可能少于纯文本集。

> **Note**
>
> 在配置任何AsSet信标之前，请考虑您的安全和性能需求。 AsSet信标可能会增加可以识别的有关数据集分布的统计信息量。例如，它们可能会显示纯文本集的大小。

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beaconName")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .asSet(AsSet.builder().build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# /.NET

```
new StandardBeacon
{
    Name = "beaconName",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
```

```
        AsSet = new AsSet()
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon_name")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::AsSet(AsSet::builder().build()?))
    .build()?
```

SharedSet

定义为的标准信标SharedSet结合了Shared和AsSet函数，因此您可以对集合和字段的加密值执行相等搜索。这样， AWS 数据库加密 SDK 就可以执行查询，CONTAINS(*a*, *b*)其中*a*是加密集和*b*加密字段。

> **Note**
>
> 在配置任何Shared信标之前，请考虑您的安全和性能需求。 SharedSet信标可能会增加可以识别的有关数据集分布的统计信息量。例如，它们可能会显示纯文本集的大小或哪些共享字段包含相同的纯文本值。

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("beacon2")
    .length(beaconLengthInBits)
    .style(
        BeaconStyle.builder()
           .sharedSet(SharedSet.builder().other("beacon1").build())
        .build()
    )
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# /.NET

```
new StandardBeacon
{
    Name = "beacon2",
    Length = beaconLengthInBits,
    Style = new BeaconStyle
    {
        SharedSet = new SharedSet { Other = "beacon1" }
    }
}
```

Rust

```
StandardBeacon::builder()
    .name("beacon2")
    .length(beacon_length_in_bits)
    .style(BeaconStyle::SharedSet(
        SharedSet::builder().other("beacon1").build()?,
    ))
    .build()?
```

配置复合信标

复合信标组合文字明文字符串和标准信标来执行复杂的数据库操作，例如从单个索引中查询两种不同的记录类型或使用排序键查询字段组合。复合信标可以通过ENCRYPT_AND_SIGNSIGN_ONLY、和SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT字段构建。您必须为复合信标中包含的每个加密字段创建标准信标。

> **Note**
>
> 我们建议您在填充数据库之前验证您的复合信标是否产生了预期的结果。有关更多信息，请参阅测试信标输出。

示例配置语法

Java

复合信标配置

以下示例在复合信标配置中本地定义加密和签名的部件列表。

```
List<CompoundBeacon> compoundBeaconList = new ArrayList<>();
CompoundBeacon exampleCompoundBeacon = CompoundBeacon.builder()
    .name("compoundBeaconName")
    .split(".")
    .encrypted(encryptedPartList)
    .signed(signedPartList)
    .constructors(constructorList)
    .build();
compoundBeaconList.add(exampleCompoundBeacon);
```

信标版本定义

以下示例在信标版本中全局定义了加密和已签名的部件列表。有关定义信标版本的更多信息，请参阅使用信标。

```
 List<BeaconVersion> beaconVersions = new ArrayList<>();
beaconVersions.add(
    BeaconVersion.builder()
        .standardBeacons(standardBeaconList)
        .compoundBeacons(compoundBeaconList)
        .encryptedParts(encryptedPartList)
        .signedParts(signedPartList)
        .version(1) // MUST be 1
        .keyStore(keyStore)
        .keySource(BeaconKeySource.builder()
            .single(SingleKeyStore.builder()
                .keyId(branchKeyId)
                .cacheTTL(6000)
                .build())
            .build())
        .build()
);
```

C# / .NET

查看完整的代码示例：BeaconConfig.cs

复合信标配置

以下示例在复合信标配置中本地定义加密和签名的部件列表。

```
var compoundBeaconList = new List<CompoundBeacon>();
var exampleCompoundBeacon = new CompoundBeacon
 {
    Name = "compoundBeaconName",
    Split = ".",
    Encrypted = encryptedPartList,
    Signed = signedPartList,
    Constructors = constructorList
 };
compoundBeaconList.Add(exampleCompoundBeacon);
```

信标版本定义

以下示例在信标版本中全局定义了加密和已签名的部件列表。有关定义信标版本的更多信息，请参阅使用信标。

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        EncryptedParts = encryptedPartsList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = keyStore,
        KeySource = new BeaconKeySource
        {
            Single = new SingleKeyStore
            {
                KeyId = branchKeyId,
                CacheTTL = 6000
            }
        }
    }
};
```

Rust

查看完整的代码示例：beacon _config.rs

复合信标配置

以下示例在复合信标配置中本地定义加密和签名的部件列表。

```
let compound_beacon_list = vec![
    CompoundBeacon::builder()
        .name("compound_beacon_name")
        .split(".")
        .encrypted(encrypted_parts_list)
        .signed(signed_parts_list)
        .constructors(constructor_list)
        .build()?
```

信标版本定义

以下示例在信标版本中全局定义了加密和已签名的部件列表。有关定义信标版本的更多信息，请参阅使用信标。

```
let beacon_versions = BeaconVersion::builder()
    .standard_beacons(standard_beacon_list)
    .compound_beacons(compound_beacon_list)
    .encrypted_parts(encrypted_parts_list)
    .signed_parts(signed_parts_list)
    .version(1) // MUST be 1
    .key_store(key_store.clone())
    .key_source(BeaconKeySource::Single(
        SingleKeyStore::builder()
            .key_id(branch_key_id)
            .cache_ttl(6000)
            .build()?,
    ))
    .build()?;
let beacon_versions = vec![beacon_versions];
```

您可以在本地或全局定义的列表中定义加密部分和签名部分。我们建议尽可能在信标版本的全局列表中定义加密和签名的部分。通过全局定义加密和签名的部件，您可以定义每个部分一次，然后在多个复合信标配置中重复使用这些部件。如果您只打算使用一次加密或已签名的部件，则可以在复合信标配置的本地列表中对其进行定义。你可以在构造函数列表中同时引用局部和全局部分。

如果您在全局范围内定义加密和签名的部件列表，则必须提供构造器部件列表，这些构造器部分标识复合信标可以在复合信标配置中组合字段的所有可能方式。

> **Note**
>
> 要全局定义加密和已签名的部件列表，必须使用 3.2 版或更高版本的 AWS 数据库加密 SDK。
> 在全局定义任何新部分之前，先将新版本部署给所有读者。
> 您无法更新现有信标配置以全局定义加密和已签名的部件列表。

要配置复合信标，请提供以下值。

信标名称

您在查询已加密字段时使用的名称。

信标名称可以与加密的字段或虚拟字段相同，但不能与未加密的字段相同。任何两个信标的名称都不能相同。如需帮助确定最适合您的实现的信标名称，请参阅选择信标名称。

分割字符

用于分隔构成复合信标的各个部分的字符。

分割字符不能出现在构成复合信标的任何字段的明文值中。

加密部件清单

标识复合信标中包含的 ENCRYPT_AND_SIGN 字段。

每个部分都必须包含名称和前缀。部分名称必须是根据加密字段构造的标准信标的名称。前缀可以是任何字符串，但它必须是唯一的。加密部分不能与已签名部分的前缀相同。建议使用简短的值，以将该部分与复合信标提供的其他部分区分开来。

我们建议尽可能在全局范围内定义您的加密部分。如果您只打算在一个复合信标中使用加密部件，则可以考虑在本地定义加密部件。本地定义的加密部分不能与全局定义的加密部分具有相同的前缀或名称。

Java

```
List<EncryptedPart> encryptedPartList = new ArrayList<>);
EncryptedPart encryptedPartExample = EncryptedPart.builder()
    .name("standardBeaconName")
    .prefix("E-")
    .build();
encryptedPartList.add(encryptedPartExample);
```

C# / .NET

```
var encryptedPartList = new List<EncryptedPart>();
var encryptedPartExample = new EncryptedPart
 {
    Name = "compoundBeaconName",
    Prefix = "E-"
 };
encryptedPartList.Add(encryptedPartExample);
```

Rust

```
let encrypted_parts_list = vec![
    EncryptedPart::builder()
        .name("standard_beacon_name")
        .prefix("E-")
        .build()?
];
```

签名零件清单

标识复合信标中包含的签名字段。

> **Note**
>
> 签名部分是可选的。您可以配置不引用任何签名部件的复合信标。

每个部分都必须包含名称、来源和前缀。来源是部件标识的SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT字段。来源必须是字段名称或引用嵌套字段值的索引。如果您的部件名称标识了来源，则可以省略来源， AWS 数据库加密 SDK 将自动使用该名称作为其来源。建议尽可能将来源指定为部分名称。前缀可以是任何字符串，但它必须是唯一的。已签名的部分不能与加密的部分具有相同的前缀。建议使用简短的值，以将该部分与复合信标提供的其他部分区分开来。

我们建议尽可能在全局范围内定义您的签名部件。如果您只打算在一个复合信标中使用签名部件，则可以考虑在本地定义签名部件。本地定义的签名部分不能与全局定义的签名部分具有相同的前缀或名称。

Java

```
List<SignedPart> signedPartList = new ArrayList<>);
SignedPart signedPartExample = SignedPart.builder()
    .name("signedFieldName")
    .prefix("S-")
    .build();
signedPartList.add(signedPartExample);
```

C# / .NET

```
var signedPartsList = new List<SignedPart>
{
    new SignedPart { Name = "signedFieldName1", Prefix = "S-" },
    new SignedPart { Name = "signedFieldName2", Prefix = "SF-" }
};
```

Rust

```
let signed_parts_list = vec![
    SignedPart::builder()
        .name("signed_field_name_1")
        .prefix("S-")
        .build()?,
   SignedPart::builder()
        .name("signed_field_name_2")
        .prefix("SF-")
        .build()?,
];
```

构造器列表

标识定义复合信标汇编加密和签名部分的不同方式的构造函数。你可以在构造函数列表中同时引用局部和全局部分。

如果使用全局定义的加密和签名部分构造复合信标，则必须提供构造函数列表。

如果您不使用任何全局定义的加密或签名部分来构造复合信标，则构造函数列表是可选的。如果您未指定构造函数列表，则 AWS 数据库加密 SDK 将使用以下默认构造函数组装复合信标。

- 所有已签名的部分均按添加到已签名部分列表的顺序排列
- 所有已加密的部分均按添加到已加密部分列表的顺序排列

- 所有部分均为必填项

构造函数

每个构造函数都是构造函数部分的有序列表，该列表定义了组合信标的一种汇编方式。构造函数部分按照添加到列表中的顺序连接在一起，其每个部分由指定的分割字符分隔。

每个构造函数部分都命名加密部分或签名部分，并定义该部分在构造函数中是必填项还是可选项。例如，如果要在 `Field1`、`Field1.Field2` 和 `Field1.Field2.Field3` 上查询复合信标，请将 `Field2` 和 `Field3` 标记为可选并创建一个构造函数。

每个构造函数必须具有至少一个必需部分。建议将每个构造函数的第一部分设为必填项，这样您就可以在查询中使用 `BEGINS_WITH` 运算符。

如果一个构造函数的所有必需部分都存在于记录中，则该构造函数成功。当您编写一条新记录时，复合信标使用构造函数列表来确定信标是否可以根据提供的值进行汇编。它尝试按照构造函数添加到构造函数列表的顺序汇编信标，并使用第一个成功的构造函数。如果任何构造函数都没有成功，则信标不会写入记录。

所有的读取者和写入者都应指定相同的构造函数顺序，以确保其查询结果正确无误。

使用以下过程指定您自己的构造函数列表。

1. 为每个加密部分和签名部分创建一个构造函数部分，以定义该部分是否为比填项。

   构造函数部分名称必须是它所代表的标准信标或前面字段的名称。

   Java

   ```
   ConstructorPart field1ConstructorPart = ConstructorPart.builder()
           .name("Field1")
           .required(true)
           .build();
   ```

   C# / .NET

   ```
   var field1ConstructorPart = new ConstructorPart { Name = "Field1", Required
    = true };
   ```

   Rust

   ```
   let field_1_constructor_part = ConstructorPart::builder()
   ```

```
    .name("field_1")
    .required(true)
    .build()?;
```

2. 使用您在步骤 1 中创建的构造函数部分为汇编复合信标的每种可能方式创建构造函数。

   例如，如果您要查询 Field1.Field2.Field3 和 Field4.Field2.Field3，则必须创建两个构造函数。Field1 和 Field4 都可以是必填项，因为它们是在两个单独的构造函数中定义的。

   Java

```
// Create a list for Field1.Field2.Field3 queries
List<ConstructorPart> field123ConstructorPartList = new ArrayList<>();
field123ConstructorPartList.add(field1ConstructorPart);
field123ConstructorPartList.add(field2ConstructorPart);
field123ConstructorPartList.add(field3ConstructorPart);
Constructor field123Constructor = Constructor.builder()
        .parts(field123ConstructorPartList)
        .build();
// Create a list for Field4.Field2.Field1 queries
List<ConstructorPart> field421ConstructorPartList = new ArrayList<>();
field421ConstructorPartList.add(field4ConstructorPart);
field421ConstructorPartList.add(field2ConstructorPart);
field421ConstructorPartList.add(field1ConstructorPart);
Constructor field421Constructor = Constructor.builder()
        .parts(field421ConstructorPartList)
        .build();
```

   C# / .NET

```
// Create a list for Field1.Field2.Field3 queries
 var field123ConstructorPartList = new Constructor
{
    Parts = new List<ConstructorPart> { field1ConstructorPart,
 field2ConstructorPart, field3ConstructorPart }
};
// Create a list for Field4.Field2.Field1 queries
var field421ConstructorPartList = new Constructor
{
    Parts = new List<ConstructorPart> { field4ConstructorPart,
 field2ConstructorPart, field1ConstructorPart }
};
```

Rust

```
// Create a list for field1.field2.field3 queries
let field1_field2_field3_constructor = Constructor::builder()
    .parts(vec![
        field1_constructor_part,
        field2_constroctor_part.clone(),
        field3_constructor_part,
    ])
    .build()?;

// Create a list for field4.field2.field1 queries
let field4_field2_field1_constructor = Constructor::builder()
    .parts(vec![
        field4_constructor_part,
        field2_constroctor_part.clone(),
        field1_constructor_part,
    ])
    .build()?;
```

3. 创建一个构造函数列表，其中包含您在步骤 2 中创建的所有构造函数。

Java

```
List<Constructor> constructorList = new ArrayList<>();
constructorList.add(field123Constructor)
constructorList.add(field421Constructor)
```

C# / .NET

```
var constructorList = new List<Constructor>
{
    field123Constructor,
    field421Constructor
};
```

Rust

```
let constructor_list = vec![
    field1_field2_field3_constructor,
    field4_field2_field1_constructor,
```

```
];
```

4. 指定constructorList何时创建复合信标。

示例配置

> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

以下示例演示如何配置标准信标和复合信标。以下配置不提供信标长度。如需帮助确定适合您的配置的信标长度，请参阅选择信标长度。

要查看演示如何配置和使用信标的完整代码示例，请参阅上的 aws-database-encryption-sdk-dynamodb 存储库中的 Java、.NET 和 Rust 可搜索加密示例。 GitHub

主题

- 标准信标
- 复合信标

标准信标

如果要在 `inspector_id_last4` 字段中查询精确匹配项，请使用以下配置创建标准信标。

Java

```
List<StandardBeacon> standardBeaconList = new ArrayList<>();
StandardBeacon exampleStandardBeacon = StandardBeacon.builder()
    .name("inspector_id_last4")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(exampleStandardBeacon);
```

C# / .NET

```
var standardBeaconList = new List<StandardBeacon>>);
StandardBeacon exampleStandardBeacon = new StandardBeacon
  {
    Name = "inspector_id_last4",
```

```
    Length = 10
  };
standardBeaconList.Add(exampleStandardBeacon);
```

Rust

```
let last4_beacon = StandardBeacon::builder()
    .name("inspector_id_last4")
    .length(10)
    .build()?;

let unit_beacon = StandardBeacon::builder().name("unit").length(30).build()?;

let standard_beacon_list = vec![last4_beacon, unit_beacon];
```

复合信标

如果要在 inspector_id_last4 和 inspector_id_last4.unit 上查询 UnitInspection 数据库，请使用以下配置创建一个复合信标。此复合信标只需要加密部分。

Java

```
// 1. Create standard beacons for the inspector_id_last4 and unit fields.
List<StandardBeacon> standardBeaconList = new ArrayList<>);
StandardBeacon inspectorBeacon = StandardBeacon.builder()
    .name("inspector_id_last4")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(inspectorBeacon);

StandardBeacon unitBeacon = StandardBeacon.builder()
    .name("unit")
    .length(beaconLengthInBits)
    .build();
standardBeaconList.add(unitBeacon);

// 2. Define the encrypted parts.
List<EncryptedPart> encryptedPartList = new ArrayList<>);

// Each encrypted part needs a name and prefix
// The name must be the name of the standard beacon
```

```
// The prefix must be unique
// For this example we use the prefix "I-" for "inspector_id_last4"
// and "U-" for "unit"
EncryptedPart encryptedPartInspector = EncryptedPart.builder()
    .name("inspector_id_last4")
    .prefix("I-")
    .build();
encryptedPartList.add(encryptedPartInspector);

EncryptedPart encryptedPartUnit = EncryptedPart.builder()
    .name("unit")
    .prefix("U-")
    .build();
encryptedPartList.add(encryptedPartUnit);

// 3. Create the compound beacon.
// This compound beacon only requires a name, split character,
// and list of encrypted parts
CompoundBeacon inspectorUnitBeacon = CompoundBeacon.builder()
    .name("inspectorUnitBeacon")
    .split(".")
    .sensitive(encryptedPartList)
    .build();
```

C# / .NET

```
// 1. Create standard beacons for the inspector_id_last4 and unit fields.
StandardBeacon inspectorBeacon = new StandardBeacon
 {
   Name = "inspector_id_last4",
   Length = 10
 };
standardBeaconList.Add(inspectorBeacon);
StandardBeacon unitBeacon = new StandardBeacon
 {
    Name = "unit",
    Length = 30
 };
standardBeaconList.Add(unitBeacon);

// 2. Define the encrypted parts.
var last4EncryptedPart = new EncryptedPart
```

```
// Each encrypted part needs a name and prefix
// The name must be the name of the standard beacon
// The prefix must be unique
// For this example we use the prefix "I-" for "inspector_id_last4"
// and "U-" for "unit"
var last4EncryptedPart = new EncryptedPart
 {
   Name = "inspector_id_last4",
   Prefix = "I-"
 };
encryptedPartList.Add(last4EncryptedPart);

var unitEncryptedPart = new EncryptedPart
 {
   Name = "unit",
   Prefix = "U-"
 };
encryptedPartList.Add(unitEncryptedPart);

// 3. Create the compound beacon.
// This compound beacon only requires a name, split character,
// and list of encrypted parts
var compoundBeaconList = new List<CompoundBeacon>>);
var inspectorCompoundBeacon = new CompoundBeacon
  {
      Name = "inspector_id_last4",
      Split = ".",
      Encrypted = encryptedPartList
  };
compoundBeaconList.Add(inspectorCompoundBeacon);
```

Rust

```
// 1. Create standard beacons for the inspector_id_last4 and unit fields.
let last4_beacon = StandardBeacon::builder()
    .name("inspector_id_last4")
    .length(10)
    .build()?;

let unit_beacon = StandardBeacon::builder().name("unit").length(30).build()?;

let standard_beacon_list = vec![last4_beacon, unit_beacon];
```

```
// 2. Define the encrypted parts.
// The name must be the name of the standard beacon
// The prefix must be unique
// For this example we use the prefix "I-" for "inspector_id_last4"
// and "U-" for "unit"
let encrypted_parts_list = vec![
    EncryptedPart::builder()
        .name("inspector_id_last4")
        .prefix("I-")
        .build()?,
    EncryptedPart::builder().name("unit").prefix("U-").build()?,
];

// 3. Create the compound beacon
// This compound beacon only requires a name, split character,
// and list of encrypted parts
let compound_beacon_list = vec![CompoundBeacon::builder()
    .name("last4UnitCompound")
    .split(".")
    .encrypted(encrypted_parts_list)
    .build()?];
```

使用信标

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

信标使您无需解密查询中的整个数据库即可搜索加密的记录。信标旨在在未填充的新数据库中实现。在现有数据库中配置的任何信标将只会映射写入数据库的新记录。信标是根据字段的明文值计算出来的，一旦字段被加密，信标就无法映射现有数据。使用信标写入新记录后，您将无法更新信标的配置。但是，您可以为添加到记录中的新字段添加新信标。

配置信标后，您必须先完成以下步骤，然后才能开始填充数据库并对信标执行查询。

1. 创建 AWS KMS 分层密钥环

   要使用可搜索的加密，必须使用 AWS KMS 分层密钥环来生成、加密和解密用于保护记录的数据密钥。

配置信标后，汇编分层密钥环先决条件并创建您的分层密钥环。

有关为什么需要分层密钥环的更多详细信息，请参阅使用分层密钥环进行可搜索的加密。

2.

定义信标版本

指定您的keyStorekeySource、、您配置的所有标准信标的列表、您配置的所有复合信标的列表、加密部分的列表、签名部分的列表和信标版本。必须为信标版本指定 1。有关定义您的 keySource 的指导，请参阅 定义您的信标密钥源。

以下 Java 示例定义单租户数据库的信标版本。如需帮助定义多租户数据库的信标版本，请参阅多租户数据库的可搜索加密。

Java

```
 List<BeaconVersion> beaconVersions = new ArrayList<>();
beaconVersions.add(
    BeaconVersion.builder()
        .standardBeacons(standardBeaconList)
        .compoundBeacons(compoundBeaconList)
        .encryptedParts(encryptedPartsList)
        .signedParts(signedPartsList)
        .version(1) // MUST be 1
        .keyStore(keyStore)
        .keySource(BeaconKeySource.builder()
            .single(SingleKeyStore.builder()
                .keyId(branchKeyId)
                .cacheTTL(6000)
                .build())
            .build())
        .build()
);
```

C# / .NET

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
```

```
        CompoundBeacons = compoundBeaconList,
        EncryptedParts = encryptedPartsList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = branchKeyStoreName,
        KeySource = new BeaconKeySource
        {
            Single = new SingleKeyStore
            {
                KeyId = branch-key-id,
                CacheTTL = 6000
            }
        }
    }
};
```

Rust

```
let beacon_version = BeaconVersion::builder()
    .standard_beacons(standard_beacon_list)
    .compound_beacons(compound_beacon_list)
    .version(1) // MUST be 1
    .key_store(key_store.clone())
    .key_source(BeaconKeySource::Single(
        SingleKeyStore::builder()
            // `keyId` references a beacon key.
            // For every branch key we create in the keystore,
            // we also create a beacon key.
            // This beacon key is not the same as the branch key,
            // but is created with the same ID as the branch key.
            .key_id(branch_key_id)
            .cache_ttl(6000)
            .build()?,
    ))
    .build()?;
let beacon_versions = vec![beacon_version];
```

3. 配置二级索引

   配置信标后，必须先配置反映每个信标的二级索引，然后才能搜索加密的字段。有关更多信息，请参阅 通过使用信标配置二级索引。

4. 定义您的加密操作

必须将用于构造标准信标的所有字段标记为 ENCRYPT_AND_SIGN。用于构造信标的所有其他字段都必须标记SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

5. 配置 AWS 数据库加密 SDK 客户端

   要配置保护您的 DynamoDB 表中的表项目的 AWS 数据库加密 SDK 客户端，请参阅 DynamoDB 的 Java 客户端加密库。

查询信标

您配置的信标类型决定了您能够执行的查询类型。标准信标使用筛选条件表达式来执行相等搜索。复合信标则组合文字明文字符串和标准信标来执行复杂的查询。查询加密的数据时，您可以搜索信标名称。

您无法比较两个标准信标的值，即使其中包含相同的底层明文。两个标准信标将为相同的明文值生成两个不同的 HMAC 标签。因此，标准信标无法执行以下查询。

- *beacon1* = *beacon2*
- *beacon1* IN (*beacon2*)
- *value* IN (*beacon1*, *beacon2*, ...)
- CONTAINS(*beacon1*, *beacon2*)

复合信标可以执行以下查询。

- BEGINS_WITH(*a*)，其中 *a* 反映了汇编的复合信标开头的字段的完整值。您不能使用 BEGINS_WITH 运算符标识以特定子字符串开头的值。但是，您可以使用 BEGINS_WITH(*S_*)，其中 *S_* 反映了汇编的复合信标开头的部分的前缀。
- CONTAINS(*a*)，其中 *a* 反映了汇编的复合信标包含的字段的完整值。您不能使用 CONTAINS 运算符标识包含特定子字符串或某个集中的值记录。

  例如，您不能执行查询 CONTAINS(*path*, "*a*"，其中 *a* 反映了某个集中的值。
- 您可以比较复合信标的签名部分。比较签名的部分时，您可以选择将加密部分的前缀附加到一个或多个签名部分中，但不能在任何查询中包括加密字段的值。

  例如，您可以比较已签名的部分并对 *signedField1* = *signedField2* 或 *value* IN (*signedField1*, *signedField2*, ...) 进行查询。

您还可以通过查询 *signedField1.A_* = *signedField2.B_* 来比较已签名部分和加密部分的前缀。

- *field* BETWEEN *a* AND *b*，其中 *a* 和 *b* 是签名部分。您可以选择将加密部分的前缀附加到一个或多个签名部分中，但不能在任何查询中包括加密字段的值。

您必须为对复合信标的查询中包含的每个部分添加前缀。例如，如果您从两个字段 encryptedField 和 signedField 构造了一个复合信标 compoundBeacon，则在查询信标时必须包含为这两个部分配置的前缀。

```
compoundBeacon = E_encryptedFieldValue.S_signedFieldValue
```

多租户数据库的可搜索加密

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

要在数据库中实现可搜索的加密，必须使用 AWS KMS 分层密钥环。分 AWS KMS 层密钥环生成、加密和解密用于保护记录的数据密钥。它还创建用于生成信标的信标密钥。在多租户数据库中使用 AWS KMS 分层密钥环时，每个租户都有不同的分支密钥和信标密钥。要查询多租户数据库中的加密数据，必须确定用于生成所查询信标的信标密钥材料。有关更多信息，请参阅 the section called “使用分层密钥环进行可搜索加密”。

在为多租户数据库定义信标版本时，请指定您配置的所有标准信标的列表、您配置的所有复合信标的列表、信标版本和 keySource。您必须将信标密钥源定义为 MultiKeyStore，并包括 keyFieldName、一个本地信标密钥缓存的缓存生存时间和本地信标密钥缓存的最大缓存大小。

如果您已配置任何已签名的信标，则必须将其包含在您的 compoundBeaconList 中。签名信标是一种复合信标，用于对和SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT字段进行索引SIGN_ONLY和执行复杂查询。

Java

```
List<BeaconVersion> beaconVersions = new ArrayList<>();
    beaconVersions.add(
        BeaconVersion.builder()
                .standardBeacons(standardBeaconList)
```

```
                .compoundBeacons(compoundBeaconList)
                .version(1) // MUST be 1
                .keyStore(branchKeyStoreName)
                .keySource(BeaconKeySource.builder()
                        .multi(MultiKeyStore.builder()
                                .keyFieldName(keyField)
                                .cacheTTL(6000)
                                .maxCacheSize(10)
                        .build())
                .build())
        .build()
    );
```

C# / .NET

```
var beaconVersions = new List<BeaconVersion>
{
    new BeaconVersion
    {
        StandardBeacons = standardBeaconList,
        CompoundBeacons = compoundBeaconList,
        EncryptedParts = encryptedPartsList,
        SignedParts = signedPartsList,
        Version = 1, // MUST be 1
        KeyStore = branchKeyStoreName,
        KeySource = new BeaconKeySource
        {
            Multi = new MultiKeyStore
            {
                KeyId = branch-key-id,
                CacheTTL = 6000,
                MaxCacheSize = 10
            }
        }
    }
};
```

Rust

```
let beacon_version = BeaconVersion::builder()
    .standard_beacons(standard_beacon_list)
    .compound_beacons(compound_beacon_list)
    .version(1) // MUST be 1
```

```
    .key_store(key_store.clone())
    .key_source(BeaconKeySource::Multi(
        MultiKeyStore::builder()
            // `keyId` references a beacon key.
            // For every branch key we create in the keystore,
            // we also create a beacon key.
            // This beacon key is not the same as the branch key,
            // but is created with the same ID as the branch key.
            .key_id(branch_key_id)
            .cache_ttl(6000)
            .max_cache_size(10)
            .build()?,
    ))
    .build()?;
let beacon_versions = vec![beacon_version];
```

keyFieldName

keyFieldName 定义存储与信标密钥关联的 branch-key-id 的字段名称，该信标密钥用于为给定租户生成信标。

当您将新记录写入数据库时，标识用于为该记录生成任何信标的信标密钥的 branch-key-id 将存储在此字段中。

默认情况下，keyField 是不显式存储在数据库中的概念字段。Dat AWS abase Encryption SDK branch-key-id 从材料描述中的加密数据密钥中识别出来，并将该值存储在概念中，keyField供您在复合信标和签名信标中引用。由于材料描述已签名，因此概念 keyField 被视为已签名的部分。

您也可以将作为SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT字段包含keyField在加密操作中，以将该字段显式存储在数据库中。如果这样做，则当您每次向数据库写入记录时都必须将 branch-key-id 手动包含在 keyField 中。

查询多租户数据库中的信标

要查询信标，您必须在查询中包含 keyField，以确定重新计算信标所需的相应信标密钥材料。必须指定与用于生成记录信标的信标密钥关联的 branch-key-id。您不能在分支密钥 ID 供应程序中指定用于标识租户的 branch-key-id 的易记名称。您可以通过以下方式将 keyField 包含在查询中。

复合信标

无论您是否明确将 keyField 存储在记录中，您都可以将 keyField 作为签名部分直接包含在复合信标中。keyField 签名部分必须为必填项。

例如，如果要从两个字段 encryptedField 和 signedField 中构造复合信标 compoundBeacon，则还必须将 keyField 作为签名部分包含在内。这使您能够对 compoundBeacon 执行以下查询。

```
compoundBeacon = E_encryptedFieldValue.S_signedFieldValue.K_branch-key-id
```

签名信标

AWS 数据库加密 SDK 使用标准信标和复合信标来提供可搜索的加密解决方案。这些信标必须至少包括一个加密字段。但是， AWS 数据库加密 SDK 还支持签名信标，这些信标可以完全通过纯文本SIGN_ONLY和字段进行配置。SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT

签名信标可以由单个部分构造。无论您是否明确将 keyField 存储在记录中，您都可以从 keyField 中构造签名信标，并使用它来创建复合查询，该查询将对 keyField 签名信标的查询与对其他信标的查询组合在一起。例如，您可以执行以下查询。

```
keyField = K_branch-key-id AND compoundBeacon =
 E_encryptedFieldValue.S_signedFieldValue
```

如需帮助配置签名信标，请参阅 创建签名的信标

直接在 **keyField** 上查询

如果您在加密操作中指定了 keyField 并将该字段显式存储在记录中，则可以创建一个复合查询，该查询将对信标的查询与对 keyField 的查询组合在一起。如果要查询标准信标，可以选择直接在 keyField 上查询。例如，您可以执行以下查询。

```
keyField = branch-key-id AND standardBeacon = S_standardBeaconValue
```

AWS 适用于 DynamoDB 的数据库加密 SDK

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

适用于 DynamoDB 的 AWS 数据库加密软件开发工具包是一个软件库，可让您在 Amazon DynamoDB 设计中加入客户端加密。适用于 DynamoDB 的 AWS 数据库加密 SDK 提供属性级加密，使您能够指定要加密哪些项目以及要在签名中包含哪些项目，以确保数据的真实性。加密传输中敏感数据和静态敏感数据有助于确保您的明文数据不会提供给任何第三方，包括 AWS。

> **Note**
>
> AWS 数据库加密 SDK 不支持 PartiQL。

在 DynamoDB 中，表是项目的集合。每个项目 都是属性 的集合。每个属性都有各自的名称和值。适用于 DynamoDB 的 AWS 数据库加密 SDK 对属性的值进行加密。然后，它将通过属性计算签名。您可以指定哪些属性值要加密，哪些属性值要包含在加密操作的签名中。

本章中的主题概述了适用于 DynamoDB 的 AWS 数据库加密 SDK，包括哪些字段已加密、客户端安装和配置指南以及可帮助您入门的 Java 示例。

主题

客户端加密和服务器端加密

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

适用于 DynamoDB 的 AWS 数据库加密 SDK 支持客户端加密，即在将表数据发送到数据库之前对其进行加密。但是，DynamoDB 提供服务器端静态加密功能，该功能会在您将表保存到磁盘时以透明方式进行加密并在您访问表时进行解密。

您选择的工具取决于数据的敏感性和应用程序的安全性要求。您可以同时使用适用于 DynamoDB 的 AWS 数据库加密 SDK 和静态加密。当您将已加密且签名的项目发送到 DynamoDB 时，DynamoDB 不会识别受保护的项目。它仅检测带有二进制属性值的典型表项目。

服务器端静态加密

DynamoDB 支持静态加密，这是一项服务器端加密功能，利用此功能，DynamoDB 可以在将表保存到磁盘时以透明方式进行加密并且在您访问表时进行解密。

当您使用 AWS SDK 与 DynamoDB 交互时，默认情况下，您的数据在通过 HTTPS 连接传输时会进行加密，在 DynamoDB 终端节点进行解密，然后重新加密，然后再存储在 DynamoDB 中。

- 默认加密。DynamoDB 在写入所有表时，以透明方式对其进行加密和解密。没有启用或禁用静态加密的选项。
- DynamoDB 创建和管理加密密钥。每个表的唯一键受 AWS KMS key 保护，该密钥绝不会让 AWS Key Management Service（AWS KMS）处于不加密状态。默认情况下，DynamoDB 在 DynamoDB 服务账户中使用 AWS 拥有的密钥，但您可以在账户中选择一个 AWS 托管式密钥 或客户托管密钥来保护您的部分或全部表。
- 所有表数据均已在磁盘上加密。当加密表保存到磁盘时，DynamoDB 会加密所有表数据，包括主键以及本地和全局二级索引。如果表具有排序键，则标记范围边界的一些排序键将以明文形式存储在表元数据中。
- 与表相关的对象也被加密。只要将 DynamoDB 流、全局表和备份写入到持久性媒体，静态加密就会保护它们。
- 您的项目在您进行访问时解密。当您访问表时，DynamoDB 会解密包含目标项目的表的一部分并向您返回明文项目。

AWS 适用于 DynamoDB 的数据库加密 SDK

客户端加密为您的数据提供 end-to-end保护，无论是传输中的数据还是静态数据，从 DynamoDB 的源数据到 DynamoDB 中的存储。您的纯文本数据永远不会泄露给任何第三方，包括。 AWS您可以将适用于 DynamoDB 的 AWS 数据库加密软件开发工具包与新的 DynamoDB 表配合使用，也可以将现有的 Amazon DynamoDB 表迁移到最新版本的 DynamoDB 数据库加密软件开发工具包。 AWS

- 您的传输中数据和静态数据均受保护。它永远不会暴露给任何第三方，包括 AWS。
- 您可以为表项目签名。您可以定向适用于 DynamoDB 的 AWS 数据库加密 SDK 以计算表项目的全部或部分的签名，包括主键属性。此签名允许您整体检测项目的未经授权的更改，包括添加或删除属性，或者交换属性值。
- 您可以通过选择密钥环的方式确定如何保护您的数据。您的密钥环决定了哪些包装密钥保护您的数据密钥并最终保护您的数据。使用最安全且对您的任务实用的包装密钥。
- 适用于 DynamoDB 的 AWS 数据库加密 SDK 不会加密整个表。您可以选择在项目中加密哪些属性。适用于 DynamoDB 的 AWS 数据库加密 SDK 不会加密整个项目。它不会加密属性名称或主键（分区键和排序键）属性的名称或值。

AWS Encryption SDK

如果您要加密存储在 DynamoDB 中的数据，我们建议使用适用于 DynamoDB 的数据库加密 SD AWS K。

AWS Encryption SDK 是一个客户端加密库，可帮助您加密和解密通用数据。尽管它可以保护任何类型的数据，但它不适用于结构化数据，如数据库记录。与适用于 DynamoDB 的 AWS 数据库加密 SDK 不同，它无法提供项目级别的完整性检查，也没有识别属性或阻止加密 AWS Encryption SDK 主密钥的逻辑。

如果您使用 AWS Encryption SDK 来加密表中的任何元素，请记住它与适用于 DynamoDB 的 AWS 数据库加密 SDK 不兼容。您无法使用一个库进行加密而使用另一个库进行解密。

哪些域已被加密和签名？

> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

DynamoDB AWS 数据库加密软件开发工具包是一个专为亚马逊 DynamoDB 应用程序设计的客户端加密库。Amazon DynamoDB 将数据存储在表中，表是项目的集合。每个项目 都是属性 的集合。每个属

性都有各自的名称和值。适用于 DynamoDB 的 AWS 数据库加密 SDK 对属性的值进行加密。然后，它将通过属性计算签名。您可以指定哪些属性值要加密，哪些属性值要包含在签名中。

加密可保护属性值的机密性。签名提供了所有已签名属性及其相互关系的完整性，并提供了身份验证。它使您能够整体检测项目的未经授权的更改（包括添加或删除属性），或者用一个加密值替换另一个加密至。

在加密项目中，某些数据保持明文形式，包括表名称、所有属性名称、未加密的属性值、主键（分区键和排序键）属性的名称和值以及属性类型。请勿在这些域中存储敏感数据。

有关适用于 DynamoDB 的 AWS 数据库加密 SDK 的工作原理的更多信息，请参阅。AWS 数据库加密 SDK 的工作原理

> **Note**
>
> 适用于 DynamoDB 的 AWS 数据库加密 SDK 主题中所有提及属性操作的内容均指加密操作。

主题

加密属性值

适用于 DynamoDB 的 AWS 数据库加密 SDK 对您指定的属性的值（但不加密属性名称或类型）进行加密。要确定加密的属性值，请使用属性操作。

例如，此项目包含 `example` 和 `test` 属性。

```
'example': 'data',
'test': 'test-value',
...
```

如果您加密了 `example` 属性，但未加密 `test` 属性，结果将类似于以下内容。加密的 `example` 属性值是二进制数据，而不是字符串。

```
'example': Binary(b"'b\x933\x9a+s\xf1\xd6a\xc5\xd5\x1aZ\xed\xd6\xce\xe9X\xf0T\xcb\x9fY
\x9f\xf3\xc9C\x83\r\xbb\\"),
'test': 'test-value'
```

```
...
```

每个项目的主键属性（分区键和排序键）必须保持明文形式，因为 DynamoDB 使用它们在表中查找项目。应该对它们进行签名而不是加密。

适用于 DynamoDB 的 AWS 数据库加密 SDK 可为您识别主键属性，并确保其值已签名，但未加密。此外，如果您标识了主键，然后尝试对其进行加密，客户端将引发异常。

客户端将材料描述存储在添加到项目的新属性（aws_dbe_head）中。材料描述说明了项目是如何加密和签名的。客户端使用此信息来验证和解密项目。存储材料描述的字段没有加密。

签署项目

加密指定属性值后，适用于 DynamoDB 的 AWS 数据库加密 SDK 会计算基于哈希的消息身份验证码 HMACs () 和数字签名，而不是材料描述、加密上下文以及属性操作中标记、或的每个字段ENCRYPT_AND_SIGN的规范化。SIGN_ONLYSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT默认情况下，ECDSA 签名处于启用状态，但不是必需的。客户端将 HMACs 和签名存储在添加到项目的新属性 (aws_dbe_foot) 中。

DynamoDB 中的可搜索加密

要配置 Amazon DynamoDB 表以进行可搜索的加密，必须使用 AWS KMS 分层密钥环来生成、加密和解密用于保护项目的数据密钥。您还必须在表加密配置中包含 SearchConfig。

> **Note**
>
> 如果您使用适用于 DynamoDB 的 Java 客户端加密库，则必须使用适用于 DynamoDB 的 AWS 低级数据库加密 SDK API 来加密、签名、验证和解密您的表格项目。DynamoDB 增强版客户端和较低级别 DynamoDBItemEncryptor 不支持可搜索的加密。

主题

- 通过使用信标配置二级索引
- 测试信标输出

通过使用信标配置二级索引

配置信标后，您必须先配置反映每个信标的二级索引，然后才能搜索加密的属性。

配置标准信标或复合信标时，AWS 数据库加密 SDK 会在信标名称中添加aws_dbe_b_前缀，以便服务器可以轻松识别信标。例如，如果您将复合信标命名为 compoundBeacon，则信标的完整名称实际上为 aws_dbe_b_compoundBeacon。如果您要配置包含标准信标或复合信标的二级索引，则必须在标识信标名称时包含 aws_dbe_b_ 前缀。

分区键和排序键

您将无法加密主键值。您的分区和排序密钥必须经过签名。您的主键值不能是标准或复合信标。

除非您指定任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性SIGN_ONLY，否则您的主键值必须是，分区和排序属性也必须是SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

您的主键值可以是已签名的信标。如果您为每个主键值配置了不同的签名信标，则必须指定属性名称以将主键值标识为已签名信标名称。但是，AWS 数据库加密 SDK 不会为已签名的信标添加aws_dbe_b_前缀。即使您为主键值配置了不同的签名信标，您也只需要在配置二级索引时为主键值指定属性名称。

本地二级索引

本地二级索引的排序键可以是信标。

如果您为排序键指定信标，类型必须为 String。如果您为排序键指定标准信标或复合信标，则必须在指定信标名称时包含 aws_dbe_b_ 前缀。如果您指定签名信标，则请指定不包含任何前缀的信标名称。

全局二级索引

全局二级索引的分区键和排序键都可以是信标。

如果您为分区键或排序键指定信标，则类型必须为 String。如果您为排序键指定标准信标或复合信标，则必须在指定信标名称时包含 aws_dbe_b_ 前缀。如果您指定签名信标，则请指定不包含任何前缀的信标名称。

属性投影

投影是从表复制到二级索引的属性集。表的分区键和排序键始终投影到索引中；您可以投影其他属性以支持应用程序的查询要求。DynamoDB 为属性投影提供三种不同的选项：KEYS_ONLY、INCLUDE 和 ALL。

如果使用 INCLUDE 属性投影在信标上进行搜索，则您必须指定构造信标所用的所有属性的名称以及包含 aws_dbe_b_ 前缀的信标名称。例如，如果通过 field1、field2 和 field3 配置了复合信标 compoundBeacon，则必须在投影中指定 aws_dbe_b_compoundBeacon、field1、field2 和 field3。

全局二级索引只能使用投影中显式指定的属性，但本地二级索引可以使用任何属性。

测试信标输出

如果您配置了复合信标或使用虚拟字段构造了信标，我们建议您在填充 DynamoDB 表之前验证这些信标是否产生了预期的输出。

AWS 数据库加密 SDK 提供的`DynamoDbEncryptionTransforms`服务可帮助您对虚拟场和复合信标输出进行故障排除。

测试虚拟字段

以下代码段创建测试项目，使用 D ynamoDB 表加密配置定义`DynamoDbEncryptionTransforms`服务，并演示如何`ResolveAttributes`使用来验证虚拟字段是否产生预期的输出。

Java

查看完整的代码示例：VirtualBeaconSearchableEncryptionExample.java

```
// Create test items
final PutItemRequest itemWithHasTestResultPutRequest = PutItemRequest.builder()
    .tableName(ddbTableName)
    .item(itemWithHasTestResult)
    .build();

final PutItemResponse itemWithHasTestResultPutResponse =
 ddb.putItem(itemWithHasTestResultPutRequest);

final PutItemRequest itemWithNoHasTestResultPutRequest = PutItemRequest.builder()
    .tableName(ddbTableName)
    .item(itemWithNoHasTestResult)
    .build();

final PutItemResponse itemWithNoHasTestResultPutResponse =
 ddb.putItem(itemWithNoHasTestResultPutRequest);

// Define the DynamoDbEncryptionTransforms service
final DynamoDbEncryptionTransforms trans = DynamoDbEncryptionTransforms.builder()
    .DynamoDbTablesEncryptionConfig(encryptionConfig).build();

// Verify configuration
final ResolveAttributesInput resolveInput = ResolveAttributesInput.builder()
    .TableName(ddbTableName)
```

```
    .Item(itemWithHasTestResult)
    .Version(1)
    .build();
final ResolveAttributesOutput resolveOutput = trans.ResolveAttributes(resolveInput);

// Verify that VirtualFields has the expected value
Map<String, String> vf = new HashMap<>();
vf.put("stateAndHasTestResult", "CAt");
assert resolveOutput.VirtualFields().equals(vf);
```

C# / .NET

参见完整的代码示例：VirtualBeaconSearchableEncryptionExample.cs。

```
 // Create item with hasTestResult=true
var itemWithHasTestResult = new Dictionary<String, AttributeValue>
{
    ["customer_id"] = new AttributeValue("ABC-123"),
    ["create_time"] = new AttributeValue { N = "1681495205" },
    ["state"] = new AttributeValue("CA"),
    ["hasTestResult"] = new AttributeValue { BOOL = true }
};

// Create item with hasTestResult=false
var itemWithNoHasTestResult = new Dictionary<String, AttributeValue>
{
    ["customer_id"] = new AttributeValue("DEF-456"),
    ["create_time"] = new AttributeValue { N = "1681495205" },
    ["state"] = new AttributeValue("CA"),
    ["hasTestResult"] = new AttributeValue { BOOL = false }
};

// Define the DynamoDbEncryptionTransforms service
var trans = new DynamoDbEncryptionTransforms(encryptionConfig);

// Verify configuration
var resolveInput = new ResolveAttributesInput
{
    TableName = ddbTableName,
    Item = itemWithHasTestResult,
    Version = 1
};
var resolveOutput = trans.ResolveAttributes(resolveInput);
```

```
// Verify that VirtualFields has the expected value
Debug.Assert(resolveOutput.VirtualFields.Count == 1);
Debug.Assert(resolveOutput.VirtualFields["stateAndHasTestResult"] == "CAt");
```

Rust

参见完整的代码示例：virtual _beacon_searchable_encryption.rs。

```
// Create item with hasTestResult=true
let item_with_has_test_result = HashMap::from([
    (
        "customer_id".to_string(),
        AttributeValue::S("ABC-123".to_string()),
    ),
    (
        "create_time".to_string(),
        AttributeValue::N("1681495205".to_string()),
    ),
    ("state".to_string(), AttributeValue::S("CA".to_string())),
    ("hasTestResult".to_string(), AttributeValue::Bool(true)),
]);

// Create item with hasTestResult=false
let item_with_no_has_test_result = HashMap::from([
    (
        "customer_id".to_string(),
        AttributeValue::S("DEF-456".to_string()),
    ),
    (
        "create_time".to_string(),
        AttributeValue::N("1681495205".to_string()),
    ),
    ("state".to_string(), AttributeValue::S("CA".to_string())),
    ("hasTestResult".to_string(), AttributeValue::Bool(false)),
]);

// Define the transform service
let trans = transform_client::Client::from_conf(encryption_config.clone())?;

// Verify the configuration
let resolve_output = trans
    .resolve_attributes()
    .table_name(ddb_table_name)
    .item(item_with_has_test_result.clone())
```

```
    .version(1)
    .send()
    .await?;

// Verify that VirtualFields has the expected value
let virtual_fields = resolve_output.virtual_fields.unwrap();
assert_eq!(virtual_fields.len(), 1);
assert_eq!(virtual_fields["stateAndHasTestResult"], "CAt");
```

测试复合信标

以下代码段创建了一个测试项目，使用 D ynamoDB 表加密配置定义DynamoDbEncryptionTransforms服务，并演示了如何ResolveAttributes使用来验证复合信标是否产生了预期的输出。

Java

查看完整的代码示例：CompoundBeaconSearchableEncryptionExample.java

```
// Create an item with both attributes used in the compound beacon.
final HashMap<String, AttributeValue> item = new HashMap<>();
item.put("work_id", AttributeValue.builder().s("9ce39272-8068-4efd-a211-
cd162ad65d4c").build());
item.put("inspection_date", AttributeValue.builder().s("2023-06-13").build());
item.put("inspector_id_last4", AttributeValue.builder().s("5678").build());
item.put("unit", AttributeValue.builder().s("011899988199").build());

// Define the DynamoDbEncryptionTransforms service
final DynamoDbEncryptionTransforms trans = DynamoDbEncryptionTransforms.builder()
    .DynamoDbTablesEncryptionConfig(encryptionConfig).build();

// Verify configuration
final ResolveAttributesInput resolveInput = ResolveAttributesInput.builder()
    .TableName(ddbTableName)
    .Item(item)
    .Version(1)
    .build();

final ResolveAttributesOutput resolveOutput = trans.ResolveAttributes(resolveInput);

// Verify that CompoundBeacons has the expected value
Map<String, String> cbs = new HashMap<>();
```

```
cbs.put("last4UnitCompound", "L-5678.U-011899988199");
assert resolveOutput.CompoundBeacons().equals(cbs);
// Note : the compound beacon actually stored in the table is not
 "L-5678.U-011899988199"
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
// and therefore the text is replaced by the associated beacon
```

C# / .NET

查看完整的代码示例：CompoundBeaconSearchableEncryptionExample.cs

```
// Create an item with both attributes used in the compound beacon
var item = new Dictionary<String, AttributeValue>
{
    ["work_id"] = new AttributeValue("9ce39272-8068-4efd-a211-cd162ad65d4c"),
    ["inspection_date"] = new AttributeValue("2023-06-13"),
    ["inspector_id_last4"] = new AttributeValue("5678"),
    ["unit"] = new AttributeValue("011899988199")
};

// Define the DynamoDbEncryptionTransforms service
var trans = new DynamoDbEncryptionTransforms(encryptionConfig);

// Verify configuration
var resolveInput = new ResolveAttributesInput
{
    TableName = ddbTableName,
    Item = item,
    Version = 1
};
var resolveOutput = trans.ResolveAttributes(resolveInput);


// Verify that CompoundBeacons has the expected value
Debug.Assert(resolveOutput.CompoundBeacons.Count == 1);
Debug.Assert(resolveOutput.CompoundBeacons["last4UnitCompound"] ==
 "L-5678.U-011899988199");
// Note : the compound beacon actually stored in the table is not
 "L-5678.U-011899988199"
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
// and therefore the text is replaced by the associated beacon
```

Rust

查看完整的代码示例：compound_beacon_searchable_encr y pt ion.rs

```
// Create an item with both attributes used in the compound beacon
let item = HashMap::from([
    (
        "work_id".to_string(),
        AttributeValue::S("9ce39272-8068-4efd-a211-cd162ad65d4c".to_string()),
    ),
    (
        "inspection_date".to_string(),
        AttributeValue::S("2023-06-13".to_string()),
    ),
    (
        "inspector_id_last4".to_string(),
        AttributeValue::S("5678".to_string()),
    ),
    (
        "unit".to_string(),
        AttributeValue::S("011899988199".to_string()),
    ),
]);

// Define the transforms service
let trans = transform_client::Client::from_conf(encryption_config.clone())?;

// Verify configuration
let resolve_output = trans
    .resolve_attributes()
    .table_name(ddb_table_name)
    .item(item.clone())
    .version(1)
    .send()
    .await?;

// Verify that CompoundBeacons has the expected value
Dlet compound_beacons = resolve_output.compound_beacons.unwrap();
assert_eq!(compound_beacons.len(), 1);
assert_eq!(
    compound_beacons["last4UnitCompound"],
    "L-5678.U-011899988199"
);
// but rather something like "L-abc.U-123", as both parts are EncryptedParts
```

```
// and therefore the text is replaced by the associated beacon
```

更新您的数据模型

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

在为 DynamoDB 配置 AWS 数据库加密 SDK 时，您需要提供属性操作。在加密时，D AWS atabase Encryption SDK 使用属性操作来识别哪些属性需要加密和签名，哪些属性需要签名（但不加密），哪些要忽略。您还可以定义允许的未签名属性以明确告诉客户端哪些属性被排除在签名之外。解密时，AWS 数据库加密 SDK 使用您定义的允许的未签名属性来识别签名中未包含哪些属性。属性操作不会保存在加密项目中， AWS 数据库加密 SDK 也不会自动更新您的属性操作。

仔细选择属性操作。如有怀疑，请使用 Encrypt and sign (加密和签名)。使用 AWS 数据库加密 SDK 保护您的项目后，您无法将现有ENCRYPT_AND_SIGNSIGN_ONLY、或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性更改为DO_NOTHING。但是，您可以安全地进行以下更改。

- 添加新ENCRYPT_AND_SIGN的SIGN_ONLY、和SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性
- 移除现有属性
- 将现有ENCRYPT_AND_SIGN属性更改为SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT
- 将现有SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性更改为ENCRYPT_AND_SIGN
- 添加新的 DO_NOTHING 属性
- 将现有的 SIGN_ONLY 属性更改为 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT
- 将现有的 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性更改为 SIGN_ONLY

可搜索加密的注意事项

在您更新数据模型之前，请仔细考虑您的更新会如何影响您通过这些属性构造的任何信标。使用信标写入新记录后，您将无法更新信标的配置。您将无法更新与用于构造信标的属性相关联的属性操作。如果

您移除现有属性及其关联信标，则将无法使用该信标来查询现有记录。您可以为添加到记录中的新字段创建新信标，但不能通过更新现有信标来包含新字段。

SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性的注意事项

默认情况下，分区和排序密钥是加密上下文中唯一包含的属性。您可以考虑定义其他字段，SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT以便分AWS KMS 层密钥环的分支密钥 ID 提供者可以识别从加密上下文中解密需要哪个分支密钥。有关更多信息，请参阅分支密钥 ID 供应商。如果您指定了任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性，则分区和排序属性也必须是SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

> **Note**
>
> 要使用SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT加密操作，必须使用 AWS 数据库加密 SDK 的 3.3 或更高版本。在更新要包含的数据模型之前，先将新版本部署给所有读者SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

添加新ENCRYPT_AND_SIGN的SIGN_ONLY、和SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性

要添加新的ENCRYPT_AND_SIGNSIGN_ONLY、或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性，请在属性操作中定义新属性。

您不能移除现有DO_NOTHING属性并将其作为ENCRYPT_AND_SIGNSIGN_ONLY、或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性重新添加。

使用带注释的数据类

如果您使用 TableSchema 定义了属性操作，则请将新的属性添加到带注释的数据类中。如果您没有为新属性指定属性操作注释，则默认情况下，客户端将对新属性进行加密和签名（除非该属性是主键的一部分）。如果您只想对新属性进行签名，则必须使用@DynamoDBEncryptionSignOnly或@DynamoDBEncryptionSignAndIncludeInEncryptionConte释添加新属性。

使用对象模型

如果您手动定义了属性操作，请将新属性添加到对象模型中的属性操作中，并指定ENCRYPT_AND_SIGNSIGN_ONLY、或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT作为属性操作。

移除现有属性

如果您决定不再需要某个属性，则可以停止向该属性写入数据，或者将其正式从属性操作中移除。当您停止向某个属性写入新数据时，该属性仍会显示在您的属性操作中。如果您将来需要重新开始使用该属性，则此操作可能会帮到您。正式从属性操作中移除属性并不能将其从数据集中移除。您的数据集仍将包含具有该属性的项目。

要正式移除现有ENCRYPT_AND_SIGNSIGN_ONLYSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、或DO_NOTHING属性，请更新您的属性操作。

如果您移除某个 DO_NOTHING 属性，不得将该属性从允许的未签名属性中移除。即使您不再向该属性写入新值，客户端仍需要知道该属性未签名，以读取包含该属性的现有项目。

使用带注释的数据类

如果您使用 TableSchema 定义了属性操作，请将从带注释的数据类中移除属性。

使用对象模型

如果您手动定义了属性操作，请从对象模型中的属性操作中移除属性。

将现有ENCRYPT_AND_SIGN属性更改为SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT

要将现有ENCRYPT_AND_SIGN属性更改为SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，必须更新属性操作。部署更新后，客户端将能够验证和解密写入属性的现有值，但却只能对写入该属性的新值进行签名。

> **Note**
>
> 在将现有ENCRYPT_AND_SIGN属性更改为SIGN_ONLY或之前，请仔细考虑您的安全要求SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。任何可以存储敏感数据的属性都应加密。

使用带注释的数据类

如果您使用定义了属性操作TableSchema，请更新现有属性，以便在带@DynamoDBEncryptionSignAndIncludeInEncryptionContext注释的数据类中包含@DynamoDBEncryptionSignOnly或注释。

使用对象模型

如果您手动定义了属性操作，请将与现有属性关联的属性操作从对象模型更新ENCRYPT_AND_SIGN为SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT在对象模型中。

将现有SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性更改为 ENCRYPT_AND_SIGN

要将现有SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性更改为ENCRYPT_AND_SIGN，必须更新您的属性操作。部署更新后，客户端将能够验证写入属性的现有值，并且能够对写入该属性的新值进行加密和签名。

使用带注释的数据类

如果您使用定义了属性操作TableSchema，请从现有属性中移除@DynamoDBEncryptionSignOnly或@DynamoDBEncryptionSignAndIncludeInEncryptionConte释。

使用对象模型

如果您手动定义了属性操作，请在对象模型ENCRYPT_AND_SIGN中将与该属性关联的属性操作从SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT更新为。

添加新的 DO_NOTHING 属性

为了降低添加新 DO_NOTHING 属性时发生错误的风险，建议您在命名 DO_NOTHING 属性时指定一个不同的前缀，然后使用该前缀来定义允许的未签名属性。

您不能从带注释的数据类中移除现有ENCRYPT_AND_SIGNSIGN_ONLY、或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性，然后再将该属性作为属性重新添加。DO_NOTHING您只能添加全新的 DO_NOTHING 属性。

添加新的 DO_NOTHING 属性的步骤取决于您是在列表中明确定义允许的未签名属性还是使用前缀对其进行定义。

使用允许的未签名属性前缀

如果您使用 TableSchema 定义了属性操作，请使用 @DynamoDBEncryptionDoNothing 注释将新的 DO_NOTHING 属性添加到带注释的数据类中。如果您手动定义了属性操作，请更新您的属性操作，

以包含新属性。请务必使用 DO_NOTHING 属性操作显式配置新属性。在新属性的名称中必须包含相同的独特前缀。

使用允许的未签名属性列表

1. 将新的 DO_NOTHING 属性添加到允许的未签名属性列表中，并部署更新的列表。
2. 部署步骤 1 的更改。

   在更改传播有需要读取此数据的所有主机之前，您无法继续执行步骤 3。
3. 将新的 DO_NOTHING 属性添加到您的属性操作中。

   a. 如果您使用 TableSchema 定义了属性操作，请使用 @DynamoDBEncryptionDoNothing 注释将新的 DO_NOTHING 属性添加到带注释的数据类中。

   b. 如果您手动定义了属性操作，请更新您的属性操作，以包含新属性。请务必使用 DO_NOTHING 属性操作显式配置新属性。
4. 部署步骤 3 的更改。

将现有的 SIGN_ONLY 属性更改为 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT

要将现有的 SIGN_ONLY 属性更改为 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，您必须更新属性操作。部署更新后，客户端将能够验证写入属性的现有值，并将继续对写入该属性的新值进行签名。写入该属性的新值将包含在加密上下文中。

如果您指定了任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性，则分区和排序属性也必须是SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

使用带注释的数据类

如果您使用定义了属性操作TableSchema，请将与该属性关联的属性操作从更新@DynamoDBEncryptionSignOnly为@DynamoDBEncryptionSignAndIncludeInEncryptionConte

使用对象模型

如果您手动定义了属性操作，请在对象模型中将与属性关联的属性操作从 SIGN_ONLY 更新为 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

将现有的 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性更改为 SIGN_ONLY

要将现有的 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 属性更改为 SIGN_ONLY，您必须更新属性操作。部署更新后，客户端将能够验证写入属性的现有值，并将继续对写入该属性的新值进行签名。写入该属性的新值将不会包含在加密上下文中。

在将现有SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性更改为之前SIGN_ONLY，请仔细考虑您的更新会如何影响分支密钥 ID 供应商的功能。

使用带注释的数据类

如果您使用定义了属性操作TableSchema，请将与该属性关联的属性操作从更新@DynamoDBEncryptionSignAndIncludeInEncryptionContext为@DynamoDBEncryptionSignOr

使用对象模型

如果您手动定义了属性操作，请在对象模型中将与属性关联的属性操作从 SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT 更新为 SIGN_ONLY。

AWS 适用于 DynamoDB 的数据库加密 SDK 可用编程语言

适用于 DynamoDB 的 AWS 数据库加密 SDK 适用于以下编程语言。特定于语言的库各不相同，但生成的实现是可互操作的。您可以使用一种语言实施进行加密，并使用另一种语言实施进行解密。互操作性可能受到语言约束的限制。如果是这样，这些约束将在有关语言实施的主题中进行描述。

主题

- Java
- .NET
- Rust

Java

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

本主题说明如何安装并使用适用于 DynamoDB 的 Java 客户端加密库的版本 3.x。有关使用适用于 DynamoDB 的 AWS 数据库加密 SDK 进行编程的详细信息，请参阅上的-dynamodb 存储库中的 Java 示例。 aws-database-encryption-sdk GitHub

> **Note**
>
> 以下主题重点侧重于适用于 DynamoDB 的 Java 客户端加密库的版本 3.x。
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。 AWS 数据库加密 SDK 继续支持旧版 DynamoDB 加密客户端版本。

主题

先决条件

在安装适用于 DynamoDB 的 Java 客户端加密库的版本 3.x 之前，请确保满足以下先决条件。

Java 开发环境

您需要使用 Java 8 或更高版本。在 Oracle 网站上，转到 Java SE 下载，然后下载并安装 Java SE Development Kit (JDK)。

如果使用 Oracle JDK，您还必须下载并安装 Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files。

AWS SDK for Java 2.x

适用于 DynamoDB 的 AWS 数据库加密 SDK 需要的 Dy namoDB 增强型客户端模块。 AWS SDK for Java 2.x可以安装整个开发工具包或仅安装此模块。

有关更新版本的信息 适用于 Java 的 AWS SDK，请参阅从 1.x 版迁移到 2.x 版。 适用于 Java 的 AWS SDK

可通过 Apache Maven 获得。 适用于 Java 的 AWS SDK 你可以声明整个模块的依赖关系 适用于 Java 的 AWS SDK，也可以只声明dynamodb-enhanced模块的依赖关系。

适用于 Java 的 AWS SDK 使用 Apache Maven 安装

- 要导入整个 适用于 Java 的 AWS SDK以作为依赖项，请在 pom.xml 文件中对其进行声明。
- 要仅为 适用于 Java 的 AWS SDK中的 Amazon DynamoDB 模块创建依赖项，请按照指定特定模块的说明进行操作。将 groupId 设置为 software.amazon.awssdk，并将 artifactID 设置为 dynamodb-enhanced。

> **Note**
>
> 如果您使用 AWS KMS 密钥环或 AWS KMS 分层密钥环，则还需要为模块创建依赖关系。 AWS KMS 将 groupId 设置为 software.amazon.awssdk，并将 artifactID 设置为 kms。

安装

您可以按以下方式安装适用于 DynamoDB 的 Java 客户端加密库的版本 3.x。

使用 Apache Maven

适用于 Java 的 Amazon DynamoDB Encryption Client 通过 Apache Maven 提供，并具有以下依赖项定义。

```
<dependency>
  <groupId>software.amazon.cryptography</groupId>
  <artifactId>aws-database-encryption-sdk-dynamodb</artifactId>
  <version>version-number</version>
</dependency>
```

使用 Gradle Kotlin

通过将以下内容添加到 Gradle 项目的依赖项部分，您可以使用 Gradle 在适用于 Java 的 Amazon DynamoDB Encryption Client 上声明依赖项。

```
implementation("software.amazon.cryptography:aws-database-encryption-sdk-
dynamodb:version-number")
```

手动方式

要安装适用于 DynamoDB 的 Java 客户端加密库，请克隆或下载-dynamodb 存储库。aws-database-encryption-sdk GitHub

安装 SDK 后，请先查看本指南中的示例代码和上的 aws-database-encryption-sdk-dynamodb 存储库中的 Java 示例。 GitHub

使用适用于 DynamoDB 的 Java 客户端加密库

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

本主题介绍了适用于 DynamoDB 的 Java 客户端加密库的版本 3.x 中的的一些函数和帮助程序类。

有关使用适用于 DynamoDB 的 Java 客户端加密库进行编程的详细信息，请参阅 J ava 示例、-dynamodb 存储库中的 Java 示例。 aws-database-encryption-sdk GitHub

主题

- 项目加密程序
- 适用于 DynamoDB 的 AWS 数据库加密 SDK 中的属性操作
- 适用于 DynamoDB 的 AWS 数据库加密 SDK 中的加密配置
- 使用 AWS 数据库加密 SDK 更新项目
- 解密签名集

项目加密程序

DynamoDB AWS 数据库加密 SDK 的核心是一个项目加密器。您可以使用适用于 DynamoDB 的 Java 客户端加密库的版本 3.x，以通过以下方式对您的 DynamoDB 表项目进行加密、签名、验证和解密。

DynamoDB 增强版客户端

您可以使用 `DynamoDbEncryptionInterceptor` 配置 DynamoDB 增强版客户端，以通过 DynamoDB `PutItem` 请求在客户端自动对项目进行加密和签名。使用 DynamoDB 增强型客户端，您可以使用带注释的数据类来定义属性操作。建议尽量使用 DynamoDB 增强型客户端。

DynamoDB 增强版客户端不支持可搜索的加密。

> **Note**
>
> AWS 数据库加密 SDK 不支持对嵌套属性进行标注。

低级 DynamoDB API

您可以使用 `DynamoDbEncryptionInterceptor` 配置低级 DynamoDB API，以通过 DynamoDB `PutItem` 请求在客户端自动对项目进行加密和签名。

您必须通过使用低级 DynamoDB API 来使用可搜索加密。

较低级别的 **`DynamoDbItemEncryptor`**

较低级别的 `DynamoDbItemEncryptor` 无需调用 DynamoDB 即可直接对您的表项目进行加密、签名或解密和验证。它不会发出 DynamoDB `PutItem` 或 `GetItem` 请求。举例来说，您可以使用较低级别的 `DynamoDbItemEncryptor` 直接解密和验证已经检索到的 DynamoDB 项目。

较低级别的 `DynamoDbItemEncryptor` 不支持可搜索加密。

适用于 DynamoDB 的 AWS 数据库加密 SDK 中的属性操作

属性操作决定哪些属性值经过加密和签名，哪些仅经过签名，哪些经过签名并包含在加密上下文中，哪些会被忽略。

> **Note**
>
> 要使用`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`加密操作，必须使用 AWS 数据库加密 SDK 的 3.3 或更高版本。在更新要包含的数据模型之前，先将新版本部署给所有读者`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。

如果您使用低级 DynamoDB API 或较低级别的 `DynamoDbItemEncryptor`，则必须手动定义属性操作。如果您使用 DynamoDB 增强型客户端，则可以手动定义属性操作，也可以使用带注释的数据类来生成一个 `TableSchema`。为了简化配置过程，建议使用带注释的数据类。使用带注释的数据类时，您只需要对对象建模一次。

> **Note**
>
> 定义属性操作后，必须定义将哪些属性排除在签名之外。为了将来更方便添加新的未签名属性，建议您选择一个不同的前缀（例如“:”）来标识您的未签名属性。在定义 DynamoDB 架构和属性操作时，将此前缀包含在标记为 DO_NOTHING 的所有属性的属性名称中。

使用带注释的数据类

使用带注释的数据类通过 DynamoDB 增强版客户端和 DynamoDbEncryptionInterceptor 指定您的属性操作。适用于 DynamoDB 的 AWS 数据库加密 SDK 使用标准的 DynamoDB 属性注释，该注释可定义属性类型以确定如何保护属性。默认情况下，除主键以外的所有属性均加密并签名，主键已签名但未加密。

> **Note**
>
> 要使用SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT加密操作，必须使用 AWS 数据库加密 SDK 的 3.3 或更高版本。在更新要包含的数据模型之前，先将新版本部署给所有读者SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

有关 aws-database-encryption-sdk DynamoDB 增强型客户端注释的更多指导，请参阅上 GitHub 的-dynamodb 存储库中的 SimpleClass.java。

默认情况下，主键属性已签名但未加密（SIGN_ONLY），而所有其他属性均经过加密和签名（ENCRYPT_AND_SIGN）。如果将任何属性定义为SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，则分区和排序属性也必须是SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。要指定例外情况，请使用在适用于 DynamoDB 的 Java 客户端加密库中定义的加密注释。例如，如果您只想对某个特定属性进行签名，请使用 @DynamoDbEncryptionSignOnly 注释。如果要对特定属性进行签名并将其包含在加密上下文中，请使用@DynamoDbEncryptionSignAndIncludeInEncryptionContext。如果对特定属性既不要签名也不要加密（DO_NOTHING），请使用 @DynamoDbEncryptionDoNothing 注释。

> **Note**
>
> AWS 数据库加密 SDK 不支持对嵌套属性进行标注。

以下示例显示了用于定义ENCRYPT_AND_SIGNSIGN_ONLY、和DO_NOTHING属性操作的注释。有关显示用于定义的注释的示例SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，请参阅SimpleClass4.java。

```
@DynamoDbBean
public class SimpleClass {

    private String partitionKey;
    private int sortKey;
    private String attribute1;
    private String attribute2;
    private String attribute3;

    @DynamoDbPartitionKey
    @DynamoDbAttribute(value = "partition_key")
    public String getPartitionKey() {
        return this.partitionKey;
    }

    public void setPartitionKey(String partitionKey) {
        this.partitionKey = partitionKey;
    }

    @DynamoDbSortKey
    @DynamoDbAttribute(value = "sort_key")
    public int getSortKey() {
        return this.sortKey;
    }

    public void setSortKey(int sortKey) {
        this.sortKey = sortKey;
    }

    public String getAttribute1() {
        return this.attribute1;
    }

    public void setAttribute1(String attribute1) {
        this.attribute1 = attribute1;
    }

    @DynamoDbEncryptionSignOnly
    public String getAttribute2() {
```

```
        return this.attribute2;
    }

    public void setAttribute2(String attribute2) {
        this.attribute2 = attribute2;
    }

    @DynamoDbEncryptionDoNothing
    public String getAttribute3() {
        return this.attribute3;
    }

    @DynamoDbAttribute(value = ":attribute3")
    public void setAttribute3(String attribute3) {
        this.attribute3 = attribute3;
    }

}
```

使用带注释的数据类创建 TableSchema，如下面的代码段所示。

```
final TableSchema<SimpleClass> tableSchema = TableSchema.fromBean(SimpleClass.class);
```

手动定义您的属性操作

要手动指定属性操作，请创建一个 Map 对象，在该对象中，名称/值对表示属性名称和指定的操作。

指定 ENCRYPT_AND_SIGN 以对属性进行加密和签名。指定 SIGN_ONLY 以对属性进行签名，但不进行加密。指定SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT对属性进行签名并将其包含在加密上下文中。如果不对属性进行签名，也将无法对其进行加密。指定 DO_NOTHING 以忽略某个属性。

分区和排序属性必须为SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。如果将任何属性定义为SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，则分区和排序属性也必须是SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

> **Note**
>
> 要使用SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT加密操作，必须使用 AWS 数据库加密 SDK 的 3.3 或更高版本。在更新要包含的数据模型之前，先将新版本部署给所有读者SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be signed
attributeActionsOnEncrypt.put("partition_key",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
// The sort attribute must be signed
attributeActionsOnEncrypt.put("sort_key",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute3",
 CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT);
attributeActionsOnEncrypt.put(":attribute4", CryptoAction.DO_NOTHING);
```

适用于 DynamoDB 的 AWS 数据库加密 SDK 中的加密配置

使用 AWS 数据库加密 SDK 时，必须为 DynamoDB 表显式定义加密配置。加密配置中所需的值取决于您是手动定义属性操作还是使用带注释的数据类来进行定义。

以下代码段使用 DynamoDB 增强版客户端、`TableSchema` 以及不同前缀定义的允许的未签名属性来定义 DynamoDB 表加密配置。

```
final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
tableConfigs.put(ddbTableName,
        DynamoDbEnhancedTableEncryptionConfig.builder()
            .logicalTableName(ddbTableName)
            .keyring(kmsKeyring)
            .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
            .schemaOnEncrypt(tableSchema)
            // Optional: only required if you use beacons
            .search(SearchConfig.builder()
                    .writeVersion(1) // MUST be 1
                    .versions(beaconVersions)
                    .build())
            .build());
```

逻辑表名

适用于您的 DynamoDB 表的逻辑表名称。

为简化 DynamoDB 还原操作，逻辑表名称以加密方式绑定到表中存储的所有数据。强烈建议您在首次定义加密配置时将 DynamoDB 表名指定为逻辑表名。必须始终指定相同的逻辑表名。要成功

解密，逻辑表名称必须与加密时所指定的名称相匹配。如果您的 DynamoDB 表名称在从备份中恢复 DynamoDB 表后发生更改，则逻辑表名称可确保解密操作仍能识别该表。

允许的未签名属性

在您的属性操作中标记为 DO_NOTHING 的属性。

允许的未签名属性将告诉客户端哪些属性被排除在签名之外。客户端假设，所有的其他属性都包含在签名中。然后，在解密记录时，客户端会从您指定的允许的未签名属性中确定需要验证哪些属性以及需要忽略哪些属性。您将不能从允许的未签名属性中移除属性。

您可以通过创建一个列出所有 DO_NOTHING 属性的数组来显式定义允许的未签名属性。您还可以在命名 DO_NOTHING 属性时指定不同的前缀，并使用前缀告诉客户端哪些属性未签名。强烈建议指定一个不同的前缀，因为它可以简化未来添加新的 DO_NOTHING 属性的过程。有关更多信息，请参阅 更新您的数据模型。

如果您没有为所有 DO_NOTHING 属性指定前缀，可以配置一个 allowedUnsignedAttributes 数组，该数组将显式列出客户端在解密时遇到这些属性时应该取消签名的所有属性。您只有在绝对必要时，才应显式定义允许的未签名属性。

搜索配置（可选）

SearchConfig 将定义信标版本。

必须指定 SearchConfig 才能使用可搜索的加密或签名信标。

算法套件（可选）

algorithmSuiteId 定义 AWS 数据库加密 SDK 使用哪种算法套件。

除非您明确指定替代算法套件，否则 AWS 数据库加密 SDK 将使用默认算法套件。默认算法套件将 AES-GCM 算法与密钥派生、数字签名和密钥承诺结合使用。尽管默认算法套件可能适用于大多数应用程序，但您可以选择备用算法套件。例如，没有数字签名的算法套件可以满足某些信任模型的需求。有关 AWS 数据库加密 SDK 支持的算法套件的信息，请参阅AWS 数据库加密 SDK 中支持的算法套件。

要选择没有 ECDSA 数字签名的 AES-GCM 算法套件，请在表加密配置中加入以下片段。

```
.algorithmSuiteId(
    DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384)
```

使用 AWS 数据库加密 SDK 更新项目

UpdateItem对于已加密或签名的项目，数据库加密 SDK 不支持 ddb:。 AWS 要更新加密或签名的项目，必须使用 ddb: PutItem。如果将同一个主键指定为 PutItem 请求中现有的项目，则新项目将完全替代现有项目。更新项目后，您还可以使用 CLOBBER 在保存时清除和替换所有属性。

解密签名集

在 AWS 数据库加密 SDK 的 3.0.0 和 3.1.0 版本中，如果您将集合类型属性定义为SIGN_ONLY，则该集的值将按照提供的顺序进行规范化。DynamoDB 不保留集合的顺序。因此，包含该集合的项目的签名验证可能会失败。如果集合值的返回顺序与提供给 AWS 数据库加密 SDK 的顺序不同，即使集合的属性包含相同的值，签名验证也会失败。

> **Note**
>
> AWS 数据库加密 SDK 3.1.1 及更高版本对所有集合类型属性的值进行规范化，因此读取值的顺序与写入 DynamoDB 的顺序相同。

如果签名验证失败，则解密操作将失败并返回以下错误消息。

```
software.amazon.cryptography.dbencryptionsdk.struc StructuredEncryptionException: 没有匹配的收件人标签。
```

如果您收到上述错误消息，并且认为要解密的项目包含使用版本 3.0.0 或 3.1.0 签名的集合，请查看-dy aws-database-encryption-sdk namodb-java 存储库的DecryptWithPermute目录，了解如何成功验证该集合 GitHub 的详细信息。

Java 示例

> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

以下示例为您演示如何使用适用于 DynamoDB 的 Java 客户端加密库来保护应用程序中的表项目。你可以在上的 aws-database-encryption-sdk-dynamodb 存储库的 Java 示例中找到更多示例（并贡献自己的示例）。 GitHub

以下示例演示了如何在未填充的全新 Amazon DynamoDB 表中配置适用于 DynamoDB 的 Java 客户端加密库。如果您想配置现有的 Amazon DynamoDB 表以进行客户端加密，请参阅 将版本 3.x 添加到现有表。

主题

- 使用 DynamoDB 增强版客户端
- 使用低级 DynamoDB API
- 使用较低的级别 DynamoDbItemEncryptor

使用 DynamoDB 增强版客户端

以下示例展示了如何作为 DynamoDB API 调用的一部分，结合使用 DynamoDB 增强型客户端和 `DynamoDbEncryptionInterceptor` 与 AWS KMS 密钥环对 DynamoDB 表项目进行加密。

您可以在 DynamoDB 增强版客户端中使用任何支持的密钥环，但我们建议尽可能使用其中 AWS KMS 一个密钥环。

> **Note**
>
> DynamoDB 增强版客户端不支持可搜索的加密。将 `DynamoDbEncryptionInterceptor` 与低级 DynamoDB API 一起使用，以便使用可搜索加密。

查看完整的代码示例：EnhancedPutGetExample.java

步骤 1：创建 AWS KMS 密钥环

以下示例使用`CreateAwsKmsMrkMultiKeyring`对称加密 KMS AWS KMS 密钥创建密钥环。`CreateAwsKmsMrkMultiKeyring` 方法可确保密钥环能够正确处理单区域密钥和多区域密钥。

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

步骤 2：根据带注释的数据类创建表架构

以下示例使用带注释的数据类创建 TableSchema。

此示例假设带注释的数据类和属性操作是使用 SimpleClass.java 定义的。有关为属性操作添加注释的更多指导，请参阅 使用带注释的数据类。

> **Note**
>
> AWS 数据库加密 SDK 不支持对嵌套属性进行标注。

```
final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);
```

步骤 3：定义从签名中可以排除哪些属性

以下示例假设所有 DO_NOTHING 属性共享不同的前缀“:”，并使用该前缀定义允许的未签名属性。客户端假设任何带有“:”前缀的属性名称都被排除在签名之外。有关更多信息，请参阅 Allowed unsigned attributes。

```
final String unsignedAttrPrefix = ":";
```

步骤 4：创建加密配置

以下示例定义了一个 tableConfigs 映射，该映射表示 DynamoDB 表的加密配置。

此示例将 DynamoDB 表名称指定为逻辑表名称。强烈建议您在首次定义加密配置时将 DynamoDB 表名指定为逻辑表名。有关更多信息，请参阅 适用于 DynamoDB 的 AWS 数据库加密 SDK 中的加密配置。

> **Note**
>
> 要使用可搜索的加密或签名信标，您还必须在加密配置中包括 SearchConfig。

```
final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
```

```
tableConfigs.put(ddbTableName,
    DynamoDbEnhancedTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .schemaOnEncrypt(tableSchema)
        .build());
```

步骤 5：创建 **DynamoDbEncryptionInterceptor**

以下示例使用步骤 4 中的 tableConfigs 中创建一个新的 DynamoDbEncryptionInterceptor。

```
final DynamoDbEncryptionInterceptor interceptor =
    DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
        CreateDynamoDbEncryptionInterceptorInput.builder()
            .tableEncryptionConfigs(tableConfigs)
            .build()
    );
```

第 6 步：创建新的 AWS SDK DynamoDB 客户端

以下示例使用步骤 5 中的创建了一个新 AWS 的 SDK DynamoDB 客户端**interceptor**。

```
final DynamoDbClient ddb = DynamoDbClient.builder()
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
                        .addExecutionInterceptor(interceptor)
                        .build())
        .build();
```

步骤 7：创建 DynamoDB 增强版客户端并创建表

以下示例使用步骤 6 中创建的 AWS SDK DynamoDB 客户端创建 DynamoDB 增强型客户端，并使用带注释的数据类创建表。

```
final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
        .dynamoDbClient(ddb)
        .build();
final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
```

步骤 8：对表项目进行加密和签名

以下示例使用 DynamoDB 增强版客户端将项目放入 DynamoDB 表中。该项目在发送到 DynamoDB 之前将在客户端进行加密和签名。

```
final SimpleClass item = new SimpleClass();
item.setPartitionKey("EnhancedPutGetExample");
item.setSortKey(0);
item.setAttribute1("encrypt and sign me!");
item.setAttribute2("sign me!");
item.setAttribute3("ignore me!");

table.putItem(item);
```

使用低级 DynamoDB API

以下示例演示如何使用带有 AWS KMS 密钥环的低级 DynamoDB API，通过您的 DynamoDB `PutItem` 请求在客户端对项目自动进行加密和签名。

您可以使用任何支持的密钥环，但我们建议尽可能使用其中一个 AWS KMS 密钥环。

查看完整的代码示例：BasicPutGetExample.java

步骤 1：创建 AWS KMS 密钥环

以下示例使用`CreateAwsKmsMrkMultiKeyring`对称加密 KMS AWS KMS 密钥创建密钥环。`CreateAwsKmsMrkMultiKeyring` 方法可确保密钥环能够正确处理单区域密钥和多区域密钥。

```
final MaterialProviders matProv = MaterialProviders.builder()
         .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
         .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

步骤 2：配置属性操作

以下示例定义了一个 `attributeActionsOnEncrypt` 映射，该映射表示表项目的示例属性操作。

> **Note**
>
> 以下示例未将任何属性定义为SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。如果您指定了任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性，则分区和排序属性也必须是SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("partition_key", CryptoAction.SIGN_ONLY);
// The sort attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("sort_key", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put(":attribute3", CryptoAction.DO_NOTHING);
```

步骤 3：定义从签名中可以排除哪些属性

以下示例假设所有 DO_NOTHING 属性共享不同的前缀“:”，并使用该前缀定义允许的未签名属性。客户端假设任何带有“:”前缀的属性名称都被排除在签名之外。有关更多信息，请参阅 Allowed unsigned attributes。

```
final String unsignedAttrPrefix = ":";
```

步骤 4：定义 DynamoDB 表的加密配置

以下示例定义了一个 `tableConfigs` 映射，该映射表示此 DynamoDB 表的加密配置。

此示例将 DynamoDB 表名称指定为逻辑表名称。强烈建议您在首次定义加密配置时将 DynamoDB 表名指定为逻辑表名。有关更多信息，请参阅 适用于 DynamoDB 的 AWS 数据库加密 SDK 中的加密配置。

> **Note**
>
> 要使用可搜索的加密或签名信标，您还必须在加密配置中包括 `SearchConfig`。

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
```

```
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .attributeActionsOnEncrypt(attributeActionsOnEncrypt)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .build();
tableConfigs.put(ddbTableName, config);
```

步骤 5：创建 **DynamoDbEncryptionInterceptor**

以下示例使用步骤 4 中的 tableConfigs 创建 DynamoDbEncryptionInterceptor。

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

第 6 步：创建新的 AWS SDK DynamoDB 客户端

以下示例使用步骤 5 中的创建了一个新 AWS 的 SDK DynamoDB 客户端**interceptor**。

```
final DynamoDbClient ddb = DynamoDbClient.builder()
        .overrideConfiguration(
                ClientOverrideConfiguration.builder()
                        .addExecutionInterceptor(interceptor)
                        .build())
        .build();
```

步骤 7：对 DynamoDB 表项目进行加密和签名

以下示例定义了一个 item 映射，该映射表示示例表项目并将该项目放入 DynamoDB 表中。该项目在发送到 DynamoDB 之前将在客户端进行加密和签名。

```
final HashMap<String, AttributeValue> item = new HashMap<>();
item.put("partition_key", AttributeValue.builder().s("BasicPutGetExample").build());
item.put("sort_key", AttributeValue.builder().n("0").build());
item.put("attribute1", AttributeValue.builder().s("encrypt and sign me!").build());
item.put("attribute2", AttributeValue.builder().s("sign me!").build());
item.put(":attribute3", AttributeValue.builder().s("ignore me!").build());

final PutItemRequest putRequest = PutItemRequest.builder()
        .tableName(ddbTableName)
```

```
        .item(item)
        .build();

final PutItemResponse putResponse = ddb.putItem(putRequest);
```

使用较低的级别 DynamoDbItemEncryptor

以下示例说明如何使用带有 AWS KMS 密钥环的较低级别 DynamoDbItemEncryptor 来直接对表项目进行加密和签名。DynamoDbItemEncryptor 不会将项目放入 DynamoDB 表中。

您可以在 DynamoDB 增强版客户端中使用任何支持的密钥环，但我们建议尽可能使用其中 AWS KMS 一个密钥环。

> **Note**
>
> 较低级别的 DynamoDbItemEncryptor 不支持可搜索加密。将 DynamoDbEncryptionInterceptor 与低级 DynamoDB API 一起使用，以便使用可搜索加密。

查看完整的代码示例：ItemEncryptDecryptExample.java

步骤 1：创建 AWS KMS 密钥环

以下示例使用CreateAwsKmsMrkMultiKeyring对称加密 KMS AWS KMS 密钥创建密钥环。CreateAwsKmsMrkMultiKeyring 方法可确保密钥环能够正确处理单区域密钥和多区域密钥。

```
final MaterialProviders matProv = MaterialProviders.builder()
         .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
         .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

步骤 2：配置属性操作

以下示例定义了一个 attributeActionsOnEncrypt 映射，该映射表示表项目的示例属性操作。

> **Note**
>
> 以下示例未将任何属性定义为SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。如果您指定了任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性，则分区和排序属性也必须是SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

```
final Map<String, CryptoAction> attributeActionsOnEncrypt = new HashMap<>();
// The partition attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("partition_key", CryptoAction.SIGN_ONLY);
// The sort attribute must be SIGN_ONLY
attributeActionsOnEncrypt.put("sort_key", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
attributeActionsOnEncrypt.put("attribute2", CryptoAction.SIGN_ONLY);
attributeActionsOnEncrypt.put(":attribute3", CryptoAction.DO_NOTHING);
```

步骤 3：定义从签名中可以排除哪些属性

以下示例假设所有 DO_NOTHING 属性共享不同的前缀“:”，并使用该前缀定义允许的未签名属性。客户端假设任何带有“:”前缀的属性名称都被排除在签名之外。有关更多信息，请参阅 Allowed unsigned attributes。

```
final String unsignedAttrPrefix = ":";
```

步骤 4：定义 **DynamoDbItemEncryptor** 配置

以下示例定义 DynamoDbItemEncryptor 的配置。

此示例将 DynamoDB 表名称指定为逻辑表名称。强烈建议您在首次定义加密配置时将 DynamoDB 表名指定为逻辑表名。有关更多信息，请参阅 适用于 DynamoDB 的 AWS 数据库加密 SDK 中的加密配置。

```
final DynamoDbItemEncryptorConfig config = DynamoDbItemEncryptorConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .attributeActionsOnEncrypt(attributeActionsOnEncrypt)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        .build();
```

步骤 5：创建 **DynamoDbItemEncryptor**

以下示例使用步骤 4 中的 config 创建新的 DynamoDbItemEncryptor。

```
final DynamoDbItemEncryptor itemEncryptor = DynamoDbItemEncryptor.builder()
        .DynamoDbItemEncryptorConfig(config)
        .build();
```

步骤 6：直接对表项目进行加密和签名

以下示例使用 DynamoDbItemEncryptor 直接对项目进行加密和签名。DynamoDbItemEncryptor 不会将项目放入 DynamoDB 表中。

```
final Map<String, AttributeValue> originalItem = new HashMap<>();
originalItem.put("partition_key",
 AttributeValue.builder().s("ItemEncryptDecryptExample").build());
originalItem.put("sort_key", AttributeValue.builder().n("0").build());
originalItem.put("attribute1", AttributeValue.builder().s("encrypt and sign
 me!").build());
originalItem.put("attribute2", AttributeValue.builder().s("sign me!").build());
originalItem.put(":attribute3", AttributeValue.builder().s("ignore me!").build());

final Map<String, AttributeValue> encryptedItem = itemEncryptor.EncryptItem(
        EncryptItemInput.builder()
                .plaintextItem(originalItem)
                .build()
).encryptedItem();
```

将现有 DynamoDB 表配置为使用适用于 DynamoDB AWS 的数据库加密 SDK

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

使用适用于 DynamoDB 的 Java 客户端加密库的 3.x 版本，您可以配置现有的 Amazon DynamoDB 表以进行客户端加密。本主题提供有关添加版本 3.x 到已填充的现有 DynamoDB 表要采取的三个步骤的指导。

先决条件

适用于 DynamoDB 的 Java 客户端加密库中的版本 3.x 需要 AWS SDK for Java 2.x 中提供的 DynamoDB 增强型客户端。如果您仍在使用 Dynamo DBMapper，则必须迁移 AWS SDK for Java 2.x 到才能使用 DynamoDB 增强型客户端。

按照从 适用于 Java 的 AWS SDK的版本 1.x 迁移到 2.x 的说明进行操作。

然后，按照说明开始使用 DynamoDB 增强型客户端 API。

在将表配置为使用适用于 DynamoDB 的 Java 客户端加密库之前，您需要使用带注释的数据类生成 `TableSchema`，并且需要创建增强型客户端。

步骤 1：准备读取和写入加密项目

完成以下步骤，为 AWS 数据库加密 SDK 客户端做好读取和写入加密项目的准备。部署以下更改后，您的客户端将继续读取和写入明文项目。它不会对写入表中的任何新项目进行加密或签名，但却能够在加密项目显示后立即对其进行解密。这些更改使得客户端为开始加密新项目做好准备。在继续执行下一步操作之前，必须将以下更改部署到每一个读取器。

1. 定义您的属性操作

   更新带注释的数据类以包含属性操作，这些操作定义哪些属性值将被加密和签名，哪些将仅被签名，哪些将被忽略。

   有关 DynamoDB 增强型客户端注释的更多指导，请参阅上 GitHub的 aws-database-encryption-sdk-dynamodb 存储库中的 SimpleClass.java。

   默认情况下，主键属性已签名但未加密（`SIGN_ONLY`），而所有其他属性均经过加密和签名（`ENCRYPT_AND_SIGN`）。要指定例外情况，请使用在适用于 DynamoDB 的 Java 客户端加密库中定义的加密注释。例如，如果您只想对某个特定属性进行签名，请使用 `@DynamoDbEncryptionSignOnly` 注释。如果要对特定属性进行签名并将其包含在加密上下文中，请使用`@DynamoDbEncryptionSignAndIncludeInEncryptionContext`注释。如果对特定属性既不要签名也不要加密（`DO_NOTHING`），请使用 `@DynamoDbEncryptionDoNothing` 注释。

   > **Note**
   >
   > 如果您指定了任何`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`属性，则分区和排序属性也必须是`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。有关显示用于定义的注释的示例`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`，请参阅 SimpleClass4.java。

有关注释的示例，请参阅 使用带注释的数据类。

2. 定义从签名中将可以排除哪些属性

以下示例假设所有 DO_NOTHING 属性共享不同的前缀“:”，并使用该前缀定义允许的未签名属性。客户端将假设任何带有“:”前缀的属性名称都被排除在签名之外。有关更多信息，请参阅 Allowed unsigned attributes。

```
final String unsignedAttrPrefix = ":";
```

3. 创建密钥环

以下示例创建一个 AWS KMS 密钥环。 AWS KMS 密钥环使用对称加密或非对称 RSA AWS KMS keys 来生成、加密和解密数据密钥。

该示例使用 CreateMrkMultiKeyring 创建带有对称加密 KMS 密钥的 AWS KMS 密钥环。CreateAwsKmsMrkMultiKeyring 方法可确保密钥环能够正确处理单区域密钥和多区域密钥。

```
final MaterialProviders matProv = MaterialProviders.builder()
        .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
        .build();
final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
        .generator(kmsKeyId)
        .build();
final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

4. 定义 DynamoDB 表的加密配置

以下示例定义了一个 tableConfigs 映射，该映射表示此 DynamoDB 表的加密配置。

此示例将 DynamoDB 表名称指定为逻辑表名称。强烈建议您在首次定义加密配置时将 DynamoDB 表名指定为逻辑表名。有关更多信息，请参阅 适用于 DynamoDB 的 AWS 数据库加密 SDK 中的加密配置。

必须指定 FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT 作为明文替代。此策略继续读取和写入明文项目，读取加密项目，并使客户端做好准备以写入加密项目。

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
```

```
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .schemaOnEncrypt(tableSchema)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)

 .plaintextOverride(PlaintextOverride.FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

5. 创建 **DynamoDbEncryptionInterceptor**

   以下示例使用步骤 3 中的 tableConfigs 创建 DynamoDbEncryptionInterceptor。

```
DynamoDbEncryptionInterceptor interceptor = DynamoDbEncryptionInterceptor.builder()
        .config(DynamoDbTablesEncryptionConfig.builder()
                .tableEncryptionConfigs(tableConfigs)
                .build())
        .build();
```

步骤 2：写入已加密和签名项目

更新 DynamoDbEncryptionInterceptor 配置中的明文策略，以允许客户端写入已加密和签名的项目。部署好以下更改后，客户端将根据您在步骤 1 中配置的属性操作对新项目进行加密和签名。客户将能够读取明文项目以及已加密和签名的项目。

在继续执行步骤 3 之前，您必须对表格中所有现有的明文项目进行加密和签名。您无法运行单一指标或查询来快速加密您的现有明文项目。使用对您的系统最有意义的过程。例如，您可以使用异步过程，以缓慢扫描表，然后使用您定义的属性操作和加密配置重写项目。要识别表中的纯文本项目，我们建议您扫描所有不包含 AWS 数据库加密 SDK 在项目加密aws_dbe_head和签名时添加的和aws_dbe_foot属性的项目。

以下示例更新了步骤 1 中的表加密配置。您必须使用 FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT 更新明文替代。该策略继续读取明文项目，但也会读取和写入加密项目。DynamoDbEncryptionInterceptor使用更新的内容创建一个新的tableConfigs。

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
```

```
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .schemaOnEncrypt(tableSchema)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)

 .plaintextOverride(PlaintextOverride.FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

步骤 3：仅读取已加密和签名项目

在对所有项目进行加密和签名后，请更新 DynamoDbEncryptionInterceptor 配置中的明文替代，以便仅允许客户端读取和写入已加密和签名的项目。部署好以下更改后，客户端将根据您在步骤 1 中配置的属性操作对新项目进行加密和签名。客户只能够读取已加密和签名的项目。

以下示例更新了步骤 2 中的表加密配置。您可以使用 FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT 更新明文替代，也可以从配置中移除明文策略。默认情况下，客户端仅读取和写入已加密和签名的项目。DynamoDbEncryptionInterceptor使用更新的内容创建一个新的tableConfigs。

```
final Map<String, DynamoDbTableEncryptionConfig> tableConfigs = new HashMap<>();
final DynamoDbTableEncryptionConfig config = DynamoDbTableEncryptionConfig.builder()
        .logicalTableName(ddbTableName)
        .partitionKeyName("partition_key")
        .sortKeyName("sort_key")
        .schemaOnEncrypt(tableSchema)
        .keyring(kmsKeyring)
        .allowedUnsignedAttributePrefix(unsignedAttrPrefix)
        // Optional: you can also remove the plaintext policy from your configuration

 .plaintextOverride(PlaintextOverride.FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT)
        .build();
tableConfigs.put(ddbTableName, config);
```

迁移到适用于 DynamoDB 的 Java 客户端加密库的版本 3.x

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

适用于 DynamoDB 的 Java 客户端加密库的版本 3.x 是对 2.x 代码库的重大改写。它包括许多更新，例如新的结构化数据格式、改进的多租户支持、无缝架构更改，以及可搜索的加密支持。本主题提供有关如何将代码迁移到版本 3.x 的指导。

从版本 1.x 迁移到 2.x

在迁移到版本 3.x 之前先迁移到版本 2.x。版本 2.x 已将最新提供程序的符号从 `MostRecentProvider` 更改为 `CachingMostRecentProvider`。如果您当前使用的是带有 `MostRecentProvider` 符号的适用于 DynamoDB 的 Java 客户端加密库的版本 1.x，必须将代码中的符号名称更新为 `CachingMostRecentProvider`。要了解更多信息，请参阅最新提供程序的更新。

从版本 2.x 迁移到 3.x

以下步骤说明如何将您的代码从适用于 DynamoDB 的 Java 客户端加密库的版本 2.x 迁移到版本 3.x。

步骤 1：准备读取新格式的项目

完成以下步骤，准备您的 AWS 数据库加密 SDK 客户端，以读取新格式的项目。部署以下更改后，您的客户端将继续以与版本 2.x 相同的行为方式运行。您的客户将继续读取和写入 2.x 格式中的项目，但是这些更改使客户端做好读取新格式项目的准备。

将你的版本更新 适用于 Java 的 AWS SDK 到 2.x 版

适用于 DynamoDB 的 Java 客户端加密库的版本 3.x 需要 DynamoDB 增强型客户端。DynamoDB 增强版客户端取代了之前版本中使用的 DBMapper Dynamo。要使用增强型客户端，您必须使用 AWS SDK for Java 2.x。

按照从 适用于 Java 的 AWS SDK的版本 1.x 迁移到 2.x 的说明进行操作。

有关需要哪些 AWS SDK for Java 2.x 模块的更多信息，请参阅先决条件。

配置您的客户端，以读取由旧版本加密的项目

以下过程概述了以下代码示例中所演示的步骤。

1. 创建一个密钥环。

   密钥环和加密材料管理程序将取代以前版本的适用于 DynamoDB 的 Java 客户端加密库中使用的加密材料提供程序。

> **Important**
>
> 您在创建密钥环时指定的包装密钥必须与版本 2.x 中用于加密材料提供程序的包装密钥相同。

2. 创建一个带注释的类的表架构。

   此步骤用于定义开始以新格式编写项目时将使用的属性操作。

   有关使用全新 DynamoDB 增强版客户端的指南，请参阅《适用于 Java 的 AWS SDK 开发人员指南》中的生成一个 TableSchema。

   以下示例假设您已使用新的属性操作注释从版本 2.x 中更新您的带注释类。有关为属性操作添加注释的更多指导，请参阅 使用带注释的数据类。

   > **Note**
   >
   > 如果您指定了任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性，则分区和排序属性也必须是SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。有关显示用于定义的注释的示例SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，请参阅 SimpleClass4.java。

3. 定义从签名中可以排除哪些属性。
4. 配置在 2.x 版本建模类中配置的属性操作的显式映射。

   此步骤定义用于以旧格式编写项目的属性操作。

5. 配置 DynamoDBEncryptor 您在适用于 DynamoDB 的 Java 客户端加密库的版本 2.x 中使用的 。
6. 配置遗留行为。
7. 创建 DynamoDbEncryptionInterceptor。
8. 创建一个新的 AWS SDK DynamoDB 客户端。
9. 创建 DynamoDBEnhancedClient 并使用您的建模类创建表。

   要了解 DynamoDB 增强版客户端的更多信息，请参阅创建增强型客户端。

```
public class MigrationExampleStep1 {
```

```
   public static void MigrationStep1(String kmsKeyId, String ddbTableName, int
sortReadValue) {
       // 1. Create a Keyring.
       //    This example creates an AWS KMS Keyring that specifies the
       //    same kmsKeyId previously used in the version 2.x configuration.
       //    It uses the 'CreateMrkMultiKeyring' method to create the
       //    keyring, so that the keyring can correctly handle both single
       //    region and Multi-Region KMS Keys.
       //    Note that this example uses the AWS SDK for Java v2 KMS client.
       final MaterialProviders matProv = MaterialProviders.builder()
               .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
               .build();
       final CreateAwsKmsMrkMultiKeyringInput keyringInput =
CreateAwsKmsMrkMultiKeyringInput.builder()
               .generator(kmsKeyId)
               .build();
       final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

       // 2. Create a Table Schema over your annotated class.
       //    For guidance on using the new attribute actions
       //    annotations, see SimpleClass.java in the
       //    aws-database-encryption-sdk-dynamodb GitHub repository.
       //    All primary key attributes must be signed but not encrypted
       //    and by default all non-primary key attributes
       //    are encrypted and signed (ENCRYPT_AND_SIGN).
       //    If you want a particular non-primary key attribute to be signed but
       //    not encrypted, use the 'DynamoDbEncryptionSignOnly' annotation.
       //    If you want a particular attribute to be neither signed nor encrypted
       //    (DO_NOTHING), use the 'DynamoDbEncryptionDoNothing' annotation.
       final TableSchema<SimpleClass> schemaOnEncrypt =
TableSchema.fromBean(SimpleClass.class);

       // 3. Define which attributes the client should expect to be excluded
       //    from the signature when reading items.
       //    This value represents all unsigned attributes across the entire
       //    dataset.
       final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");

       // 4. Configure an explicit map of the attribute actions configured
       //    in your version 2.x modeled class.
       final Map<String, CryptoAction> legacyActions = new HashMap<>();
       legacyActions.put("partition_key", CryptoAction.SIGN_ONLY);
       legacyActions.put("sort_key", CryptoAction.SIGN_ONLY);
```

```
        legacyActions.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
        legacyActions.put("attribute2", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute3", CryptoAction.DO_NOTHING);

        // 5. Configure the DynamoDBEncryptor that you used in version 2.x.
        final AWSKMS kmsClient = AWSKMSClientBuilder.defaultClient();
        final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kmsClient,
kmsKeyId);
        final DynamoDBEncryptor oldEncryptor = DynamoDBEncryptor.getInstance(cmp);

        // 6. Configure the legacy behavior.
        //    Input the DynamoDBEncryptor and attribute actions created in
        //    the previous steps. For Legacy Policy, use
        //    'FORCE_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT'. This policy continues to
read
        //    and write items using the old format, but will be able to read
        //    items written in the new format as soon as they appear.
        final LegacyOverride legacyOverride = LegacyOverride
                .builder()
                .encryptor(oldEncryptor)
                .policy(LegacyPolicy.FORCE_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT)
                .attributeActionsOnEncrypt(legacyActions)
                .build();

        // 7. Create a DynamoDbEncryptionInterceptor with the above configuration.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .legacyOverride(legacyOverride)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );

        // 8. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 7.
```

```
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 9. Create the DynamoDbEnhancedClient using the AWS SDK DynamoDb client
        //    created in Step 8, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

步骤 2：用新格式编写项目

将步骤 1 中的更改部署到所有读取器后，请完成以下步骤，将 AWS 数据库加密 SDK 客户端配置为以新格式写入项目。部署以下更改后，您的客户端将继续读取旧格式的项目，并且开始以新格式编写和读取项目。

以下过程概述了以下代码示例中所演示的步骤。

1. 继续配置密钥环、表架构、旧属性操作 allowedUnsignedAttributes 和 DynamoDBEncryptor，就像在步骤 1 中执行的操作一样。
2. 将遗留行为更新为仅使用新格式编写新项目。
3. 创建 DynamoDbEncryptionInterceptor
4. 创建一个新的 AWS SDK DynamoDB 客户端。
5. 创建 DynamoDBEnhancedClient 并使用您的建模类创建表。

   要了解 DynamoDB 增强版客户端的更多信息，请参阅创建增强型客户端。

```
public class MigrationExampleStep2 {

    public static void MigrationStep2(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Continue to configure your keyring, table schema, legacy
        //    attribute actions, allowedUnsignedAttributes, and
```

```
        //    DynamoDBEncryptor as you did in Step 1.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        final TableSchema<SimpleClass> schemaOnEncrypt =
TableSchema.fromBean(SimpleClass.class);

        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");

        final Map<String, CryptoAction> legacyActions = new HashMap<>();
        legacyActions.put("partition_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("sort_key", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute1", CryptoAction.ENCRYPT_AND_SIGN);
        legacyActions.put("attribute2", CryptoAction.SIGN_ONLY);
        legacyActions.put("attribute3", CryptoAction.DO_NOTHING);

        final AWSKMS kmsClient = AWSKMSClientBuilder.defaultClient();
        final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kmsClient,
kmsKeyId);
        final DynamoDBEncryptor oldEncryptor = DynamoDBEncryptor.getInstance(cmp);

        // 2. Update your legacy behavior to only write new items using the new
        //    format.
        //    For Legacy Policy, use 'FORBID_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT'. This
policy
        //    continues to read items in both formats, but will only write items
        //    using the new format.
        final LegacyOverride legacyOverride = LegacyOverride
                .builder()
                .encryptor(oldEncryptor)
                .policy(LegacyPolicy.FORBID_LEGACY_ENCRYPT_ALLOW_LEGACY_DECRYPT)
                .attributeActionsOnEncrypt(legacyActions)
                .build();

        // 3. Create a DynamoDbEncryptionInterceptor with the above configuration.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
HashMap<>();
        tableConfigs.put(ddbTableName,
```

```
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .legacyOverride(legacyOverride)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );

        // 4. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 3.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 5. Create the DynamoDbEnhancedClient using the AWS SDK DynamoDb Client
 created
        //    in Step 4, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

部署步骤 2 的更改后，必须使用新格式重新加密表中的所有旧项目，然后才能继续执行步骤 3。您无法运行单一指标或查询来快速加密您的现有项目。使用对您的系统最有意义的过程。例如，您可以使用异步过程，以缓慢扫描表，然后使用您定义的新属性操作和加密配置重写项目。

步骤 3：只能以新格式读取和编写项目

使用新格式重新加密表格中的所有项目后，您可以从配置中移除遗留行为。完成以下步骤以将客户端配置为仅以新格式读取和编写项目。

以下过程概述了以下代码示例中所演示的步骤。

1. 继续配置密钥环、表架构和 allowedUnsignedAttributes，就像在步骤 1 中执行的操作一样。从您的配置中移除旧属性操作和 DynamoDBEncryptor。
2. 创建 DynamoDbEncryptionInterceptor。
3. 创建一个新的 AWS SDK DynamoDB 客户端。
4. 创建 DynamoDBEnhancedClient 并使用您的建模类创建表。

   要了解 DynamoDB 增强版客户端的更多信息，请参阅创建增强型客户端。

```
public class MigrationExampleStep3 {

    public static void MigrationStep3(String kmsKeyId, String ddbTableName, int
 sortReadValue) {
        // 1. Continue to configure your keyring, table schema,
        //    and allowedUnsignedAttributes as you did in Step 1.
        //    Do not include the configurations for the DynamoDBEncryptor or
        //    the legacy attribute actions.
        final MaterialProviders matProv = MaterialProviders.builder()
                .MaterialProvidersConfig(MaterialProvidersConfig.builder().build())
                .build();
        final CreateAwsKmsMrkMultiKeyringInput keyringInput =
 CreateAwsKmsMrkMultiKeyringInput.builder()
                .generator(kmsKeyId)
                .build();
        final IKeyring kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);

        final TableSchema<SimpleClass> schemaOnEncrypt =
 TableSchema.fromBean(SimpleClass.class);

        final List<String> allowedUnsignedAttributes = Arrays.asList("attribute3");


        // 3. Create a DynamoDbEncryptionInterceptor with the above configuration.
        //    Do not configure any legacy behavior.
        final Map<String, DynamoDbEnhancedTableEncryptionConfig> tableConfigs = new
 HashMap<>();
        tableConfigs.put(ddbTableName,
                DynamoDbEnhancedTableEncryptionConfig.builder()
                        .logicalTableName(ddbTableName)
                        .keyring(kmsKeyring)
```

```
                        .allowedUnsignedAttributes(allowedUnsignedAttributes)
                        .schemaOnEncrypt(tableSchema)
                        .build());
        final DynamoDbEncryptionInterceptor interceptor =
                DynamoDbEnhancedClientEncryption.CreateDynamoDbEncryptionInterceptor(
                        CreateDynamoDbEncryptionInterceptorInput.builder()
                                .tableEncryptionConfigs(tableConfigs)
                                .build()
                );

        // 4. Create a new AWS SDK DynamoDb client using the
        //    interceptor from Step 3.
        final DynamoDbClient ddb = DynamoDbClient.builder()
                .overrideConfiguration(
                        ClientOverrideConfiguration.builder()
                                .addExecutionInterceptor(interceptor)
                                .build())
                .build();

        // 5. Create the DynamoDbEnhancedClient using the AWS SDK Client
        //    created in Step 4, and create a table with your modeled class.
        final DynamoDbEnhancedClient enhancedClient = DynamoDbEnhancedClient.builder()
                .dynamoDbClient(ddb)
                .build();
        final DynamoDbTable<SimpleClass> table = enhancedClient.table(ddbTableName,
 tableSchema);
    }
}
```

.NET

本主题介绍如何安装和使用版本 3。 DynamoDB 的.NET 客户端加密库中的 x。有关使用适用于 DynamoDB 的 AWS 数据库加密 SDK 进行编程的详细信息，请参阅上-dynamodb 存储库中的 aws-database-encryption-sdk .NET 示例。 GitHub

DynamoDB 的.NET 客户端加密库适用于使用 C# 和其他.NET 编程语言编写应用程序的开发人员。它在 Windows、macOS 和 Linux 上受支持。

适用于 DynamoDB 的 AWS 数据库加密 SDK 的所有编程语言实现均可互操作。但是，列表或地图数据类型 适用于 .NET 的 SDK 不支持空值。这意味着，如果您使用适用于 DynamoDB 的 Java 客户端加密库来编写包含列表或地图数据类型的空值的项目，则无法使用适用于 DynamoDB 的.NET 客户端加密库来解密和读取该项目。

主题

为 DynamoDB 安装.NET 客户端加密库

DynamoDB 的.NET 客户端加密库以 AWS.cryptography 的形式提供。 DbEncryptionSDK。 DynamoDb打包进去 NuGet。有关安装和构建库的详细信息，请参阅-dynamodb 存储库中的 .NET README.md 文件。 aws-database-encryption-sdk即使您没有 AWS Key Management Service 使用 () 密钥，DynamoDB 适用于 .NET 的 SDK 的.NET 客户端加密库也需要。AWS KMS随 适用于 .NET 的 SDK NuGet 软件包一起安装。

版本 3。 DynamoDB 的.NET 客户端加密库中的 x 支持.NET 6.0 和.NET Framework net48 及更高版本。

使用.NET 调试

DynamoDB 的.NET 客户端加密库不会生成任何日志。DynamoDB 的.NET 客户端加密库中的异常会生成异常消息，但不会生成堆栈跟踪。

为了帮助您进行调试，请务必在 适用于 .NET 的 SDK中启用日志记录功能。中的日志和错误消息 适用于 .NET 的 SDK 可以帮助您区分在 DynamoDB 的.NET 客户端加密库中出现的 适用于 .NET 的 SDK 错误和 DynamoDB 的.NET 客户端加密库中出现的错误。有关 适用于 .NET 的 SDK 日志记录的帮助，请参阅AWSLogging《适用于 .NET 的 AWS SDK 开发人员指南》。（要查看该主题，请展开 Open to view .NET Framework content 部分。)

使用适用于 DynamoDB 的.NET 客户端加密库

本主题解释了版本 3 中的一些函数和辅助类。 DynamoDB 的.NET 客户端加密库中的 x。

有关使用适用于 DynamoDB 的.NET 客户端加密库进行编程的详细信息，请参阅上的-dynamodb 存储库中的 aws-database-encryption-sdk.NET 示例。 GitHub

主题

- 适用于 DynamoDB 的 AWS 数据库加密 SDK 中的属性操作
- 适用于 DynamoDB 的 AWS 数据库加密 SDK 中的加密配置
- 使用 AWS 数据库加密 SDK 更新项目

项目加密程序

DynamoDB AWS 数据库加密 SDK 的核心是一个项目加密器。你可以使用版本 3。 DynamoDB 的.NET 客户端加密库中的 x，用于通过以下方式对您的 DynamoDB 表项目进行加密、签名、验证和解密。

适用于 DynamoDB 的低级 AWS 数据库加密 SDK API

> 您可以使用表加密配置来构建 DynamoDB 客户端，该客户端会自动使用您的 DynamoDB 请求在客户端对项目进行加密和签名。PutItem您可以直接使用此客户端，也可以构造文档模型或对象持久化模型。
>
> 您必须使用适用于 DynamoDB API 的低级 AWS 数据库加密 SDK 才能使用可搜索的加密。

较低级别的 **DynamoDbItemEncryptor**

> 较低级别的 DynamoDbItemEncryptor 无需调用 DynamoDB 即可直接对您的表项目进行加密、签名或解密和验证。它不会发出 DynamoDB PutItem 或 GetItem 请求。举例来说，您可以使用较低级别的 DynamoDbItemEncryptor 直接解密和验证已经检索到的 DynamoDB 项目。如果您使用较低级别DynamoDbItemEncryptor，我们建议您使用低级编程模型，该模型 适用于 .NET 的 SDK 用于与 DynamoDB 通信。
>
> 较低级别的 DynamoDbItemEncryptor 不支持可搜索加密。

适用于 DynamoDB 的 AWS 数据库加密 SDK 中的属性操作

属性操作决定哪些属性值经过加密和签名，哪些仅经过签名，哪些经过签名并包含在加密上下文中，哪些会被忽略。

要使用.NET 客户端指定属性操作，请使用对象模型手动定义属性操作。通过创建一个Dictionary对象来指定您的属性操作，其中名称-值对表示属性名称和指定操作。

指定 ENCRYPT_AND_SIGN 以对属性进行加密和签名。指定 SIGN_ONLY 以对属性进行签名，但不进行加密。指定SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT对属性进行签名并将其包含在加密上下文中。如果不对属性进行签名，也将无法对其进行加密。指定 DO_NOTHING 以忽略某个属性。

分区和排序属性必须为SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。如果将任何属性定义为SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，则分区和排序属性也必须是SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

> **Note**
>
> 定义属性操作后，必须定义将哪些属性排除在签名之外。为了将来更方便添加新的未签名属性，建议您选择一个不同的前缀（例如“:”）来标识您的未签名属性。在定义 DynamoDB 架构和属性操作时，将此前缀包含在标记为 DO_NOTHING 的所有属性的属性名称中。

以下对象模型演示了如何使用.NET 客户端指定ENCRYPT_AND_SIGNSIGN_ONLYSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、、和DO_NOTHING属性操作。此示例使用前缀 “:” 来标识DO_NOTHING属性。

> **Note**
>
> 要使用SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT加密操作，必须使用 AWS 数据库加密 SDK 的 3.3 或更高版本。在更新要包含的数据模型之前，先将新版本部署给所有读者SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, // The
 partition attribute must be signed
    ["sort_key"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT, // The sort
 attribute must be signed
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    ["attribute3"] = CryptoAction.SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT,
    [":attribute4"] = CryptoAction.DO_NOTHING
};
```

适用于 DynamoDB 的 AWS 数据库加密 SDK 中的加密配置

使用 AWS 数据库加密 SDK 时，必须为 DynamoDB 表显式定义加密配置。加密配置中所需的值取决于您是手动定义属性操作还是使用带注释的数据类来进行定义。

以下代码段使用适用于 DynamoDB API 的 AWS 低级数据库加密 SDK 定义了 DynamoDB 表加密配置，并允许使用不同前缀定义的未签名属性。

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    // Optional: SearchConfig only required if you use beacons
    Search = new SearchConfig
    {
        WriteVersion = 1, // MUST be 1
        Versions = beaconVersions
    }
};
tableConfigs.Add(ddbTableName, config);
```

逻辑表名

适用于您的 DynamoDB 表的逻辑表名称。

为简化 DynamoDB 还原操作，逻辑表名称以加密方式绑定到表中存储的所有数据。强烈建议您在首次定义加密配置时将 DynamoDB 表名指定为逻辑表名。必须始终指定相同的逻辑表名。要成功解密，逻辑表名称必须与加密时所指定的名称相匹配。如果您的 DynamoDB 表名称在从备份中恢复 DynamoDB 表后发生更改，则逻辑表名称可确保解密操作仍能识别该表。

允许的未签名属性

在您的属性操作中标记为 DO_NOTHING 的属性。

允许的未签名属性将告诉客户端哪些属性被排除在签名之外。客户端假设，所有的其他属性都包含在签名中。然后，在解密记录时，客户端会从您指定的允许的未签名属性中确定需要验证哪些属性以及需要忽略哪些属性。您将不能从允许的未签名属性中移除属性。

您可以通过创建一个列出所有 DO_NOTHING 属性的数组来显式定义允许的未签名属性。您还可以在命名 DO_NOTHING 属性时指定不同的前缀，并使用前缀告诉客户端哪些属性未签名。强烈建议指定一个不同的前缀，因为它可以简化未来添加新的 DO_NOTHING 属性的过程。有关更多信息，请参阅 更新您的数据模型。

如果您没有为所有 DO_NOTHING 属性指定前缀，可以配置一个 allowedUnsignedAttributes 数组，该数组将显式列出客户端在解密时遇到这些属性时应该取消签名的所有属性。您只有在绝对必要时，才应显式定义允许的未签名属性。

搜索配置（可选）

SearchConfig 将定义信标版本。

必须指定 SearchConfig 才能使用可搜索的加密或签名信标。

算法套件（可选）

algorithmSuiteId 定义 AWS 数据库加密 SDK 使用哪种算法套件。

除非您明确指定替代算法套件，否则 AWS 数据库加密 SDK 将使用默认算法套件。默认算法套件将 AES-GCM 算法与密钥派生、数字签名和密钥承诺结合使用。尽管默认算法套件可能适用于大多数应用程序，但您可以选择备用算法套件。例如，没有数字签名的算法套件可以满足某些信任模型的需求。有关 AWS 数据库加密 SDK 支持的算法套件的信息，请参阅AWS 数据库加密 SDK 中支持的算法套件。

要选择没有 ECDSA 数字签名的 AES-GCM 算法套件，请在表加密配置中加入以下片段。

```
AlgorithmSuiteId =
 DBEAlgorithmSuiteId.ALG_AES_256_GCM_HKDF_SHA512_COMMIT_KEY_SYMSIG_HMAC_SHA384
```

使用 AWS 数据库加密 SDK 更新项目

UpdateItem对于包含加密或签名属性的项目，数据库加密 SDK 不支持 ddb:。 AWS 要更新加密或已签名的属性，必须使用 ddb: PutItem。如果将同一个主键指定为 PutItem 请求中现有的项目，则新项目将完全替代现有项目。更新项目后，您还可以使用 CLOBBER 在保存时清除和替换所有属性。

.NET 示

以下示例向您展示如何使用适用于 DynamoDB 的.NET 客户端加密库来保护应用程序中的表项目。要查找更多示例（并贡献自己的示例），请参阅上的 aws-database-encryption-sdk-dy namodb 存储库中的.NET 示例。 GitHub

以下示例演示了如何在未填充的全新 Amazon DynamoDB 表中为 DynamoDB 配置.NET 客户端加密库。如果您想配置现有的 Amazon DynamoDB 表以进行客户端加密，请参阅 将版本 3.x 添加到现有表。

主题

- 使用适用于 DynamoDB 的低级 AWS 数据库加密 SDK API
- 使用较低的级别 DynamoDbItemEncryptor

使用适用于 DynamoDB 的低级 AWS 数据库加密 SDK API

以下示例说明如何使用适用于 DynamoDB 的低级 AWS 数据库加密 SDK API 和密钥环，通过AWS KMS 您的 DynamoDB 请求在客户端自动加密和签名项目。 `PutItem`

您可以使用任何支持的密钥环，但我们建议尽可能使用其中一个 AWS KMS 密钥环。

查看完整的代码示例：BasicPutGetExample.cs

步骤 1：创建 AWS KMS 密钥环

以下示例使用`CreateAwsKmsMrkMultiKeyring`对称加密 KMS AWS KMS 密钥创建密钥环。`CreateAwsKmsMrkMultiKeyring` 方法可确保密钥环能够正确处理单区域密钥和多区域密钥。

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

步骤 2：配置属性操作

以下示例定义了一个`attributeActionsOnEncrypt`字典，该字典表示表格项目的示例属性操作。

> **Note**
>
> 以下示例未将任何属性定义为`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。如果您指定了任何`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`属性，则分区和排序属性也必须是`SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT`。

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
```

```
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

步骤 3：定义从签名中可以排除哪些属性

以下示例假设所有 DO_NOTHING 属性共享不同的前缀“:”，并使用该前缀定义允许的未签名属性。客户端假设任何带有“:”前缀的属性名称都被排除在签名之外。有关更多信息，请参阅 Allowed unsigned attributes。

```
const String unsignAttrPrefix = ":";
```

步骤 4：定义 DynamoDB 表的加密配置

以下示例定义了一个 tableConfigs 映射，该映射表示此 DynamoDB 表的加密配置。

此示例将 DynamoDB 表名称指定为逻辑表名称。强烈建议您在首次定义加密配置时将 DynamoDB 表名指定为逻辑表名。有关更多信息，请参阅 适用于 DynamoDB 的 AWS 数据库加密 SDK 中的加密配置。

> **Note**
>
> 要使用可搜索的加密或签名信标，您还必须在加密配置中包括 SearchConfig。

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix
};
tableConfigs.Add(ddbTableName, config);
```

第 5 步：创建新的 AWS SDK DynamoDB 客户端

以下示例使用步骤 4 中的创建了一个新 AWS 的 SDK DynamoDB 客户端**TableEncryptionConfigs**。

```
var ddb = new Client.DynamoDbClient(
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

步骤 6：对 DynamoDB 表格项目进行加密和签名

以下示例定义了一个代表示例表项目的item字典，并将该项目放入 DynamoDB 表中。该项目在发送到 DynamoDB 之前将在客户端进行加密和签名。

```
var item = new Dictionary<String, AttributeValue>
{
    ["partition_key"] = new AttributeValue("BasicPutGetExample"),
    ["sort_key"] = new AttributeValue { N = "0" },
    ["attribute1"] = new AttributeValue("encrypt and sign me!"),
    ["attribute2"] = new AttributeValue("sign me!"),
    [":attribute3"] = new AttributeValue("ignore me!")
};

PutItemRequest putRequest = new PutItemRequest
{
    TableName = ddbTableName,
    Item = item
};

PutItemResponse putResponse = await ddb.PutItemAsync(putRequest);
```

使用较低的级别 **DynamoDbItemEncryptor**

以下示例说明如何使用带有 AWS KMS 密钥环的较低级别 DynamoDbItemEncryptor 来直接对表项目进行加密和签名。DynamoDbItemEncryptor 不会将项目放入 DynamoDB 表中。

您可以在 DynamoDB 增强版客户端中使用任何支持的密钥环，但我们建议尽可能使用其中 AWS KMS 一个密钥环。

> **Note**
>
> 较低级别的 DynamoDbItemEncryptor 不支持可搜索加密。使用适用于 DynamoDB 的低级 AWS 数据库加密 SDK API 来使用可搜索的加密。

查看完整的代码示例：ItemEncryptDecryptExample.cs

步骤 1：创建 AWS KMS 密钥环

以下示例使用CreateAwsKmsMrkMultiKeyring对称加密 KMS AWS KMS 密钥创建密钥环。CreateAwsKmsMrkMultiKeyring 方法可确保密钥环能够正确处理单区域密钥和多区域密钥。

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

步骤 2：配置属性操作

以下示例定义了一个attributeActionsOnEncrypt字典，该字典表示表格项目的示例属性操作。

> **Note**
>
> 以下示例未将任何属性定义为SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。如果您指定了任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性，则分区和排序属性也必须是SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

```
var attributeActionsOnEncrypt = new Dictionary<String, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

步骤 3：定义从签名中可以排除哪些属性

以下示例假设所有 DO_NOTHING 属性共享不同的前缀“:”，并使用该前缀定义允许的未签名属性。客户端假设任何带有“:”前缀的属性名称都被排除在签名之外。有关更多信息，请参阅 Allowed unsigned attributes。

```
String unsignAttrPrefix = ":";
```

步骤 4：定义 DynamoDbItemEncryptor 配置

以下示例定义 DynamoDbItemEncryptor 的配置。

此示例将 DynamoDB 表名称指定为逻辑表名称。强烈建议您在首次定义加密配置时将 DynamoDB 表名指定为逻辑表名。有关更多信息，请参阅 适用于 DynamoDB 的 AWS 数据库加密 SDK 中的加密配置。

```
var config = new DynamoDbItemEncryptorConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix
};
```

步骤 5：创建 DynamoDbItemEncryptor

以下示例使用步骤 4 中的 config 创建新的 DynamoDbItemEncryptor。

```
var itemEncryptor = new DynamoDbItemEncryptor(config);
```

步骤 6：直接对表项目进行加密和签名

以下示例使用 DynamoDbItemEncryptor 直接对项目进行加密和签名。DynamoDbItemEncryptor 不会将项目放入 DynamoDB 表中。

```
var originalItem = new Dictionary<String, AttributeValue>
{
    ["partition_key"] = new AttributeValue("ItemEncryptDecryptExample"),
    ["sort_key"] = new AttributeValue { N = "0" },
    ["attribute1"] = new AttributeValue("encrypt and sign me!"),
    ["attribute2"] = new AttributeValue("sign me!"),
    [":attribute3"] = new AttributeValue("ignore me!")
};

var encryptedItem = itemEncryptor.EncryptItem(
    new EncryptItemInput { PlaintextItem = originalItem }
).EncryptedItem;
```

将现有 DynamoDB 表配置为使用适用于 DynamoDB AWS 的数据库加密 SDK

使用版本 3。 x 在 DynamoDB 的.NET 客户端加密库中，您可以将现有的 Amazon DynamoDB 表配置为用于客户端加密。本主题提供有关添加版本 3.x 到已填充的现有 DynamoDB 表要采取的三个步骤的指导。

步骤 1：准备读取和写入加密项目

完成以下步骤，为 AWS 数据库加密 SDK 客户端做好读取和写入加密项目的准备。部署以下更改后，您的客户端将继续读取和写入明文项目。它不会对写入表中的任何新项目进行加密或签名，但却能够在加密项目显示后立即对其进行解密。这些更改使得客户端为开始加密新项目做好准备。在继续执行下一步操作之前，必须将以下更改部署到每一个读取器。

1. 定义您的属性操作

   创建对象模型以定义哪些属性值将进行加密和签名，哪些仅进行签名，哪些将被忽略。

   默认情况下，主键属性已签名但未加密（SIGN_ONLY），而所有其他属性均经过加密和签名（ENCRYPT_AND_SIGN）。

   指定 ENCRYPT_AND_SIGN 以对属性进行加密和签名。指定 SIGN_ONLY 以对属性进行签名，但不进行加密。指定SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT要签名并归因并将其包含在加密上下文中。如果不对属性进行签名，也将无法对其进行加密。指定 DO_NOTHING 以忽略某个属性。有关更多信息，请参阅 适用于 DynamoDB 的 AWS 数据库加密 SDK 中的属性操作。

   > **Note**
   >
   > 如果您指定了任何SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT属性，则分区和排序属性也必须是SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

```
var attributeActionsOnEncrypt = new Dictionary<string, CryptoAction>
{
    ["partition_key"] = CryptoAction.SIGN_ONLY, // The partition attribute must be
 SIGN_ONLY
    ["sort_key"] = CryptoAction.SIGN_ONLY, // The sort attribute must be SIGN_ONLY
    ["attribute1"] = CryptoAction.ENCRYPT_AND_SIGN,
    ["attribute2"] = CryptoAction.SIGN_ONLY,
    [":attribute3"] = CryptoAction.DO_NOTHING
};
```

2. 定义从签名中将可以排除哪些属性

以下示例假设所有 DO_NOTHING 属性共享不同的前缀“:”，并使用该前缀定义允许的未签名属性。客户端将假设任何带有“:”前缀的属性名称都被排除在签名之外。有关更多信息，请参阅 Allowed unsigned attributes。

```
const String unsignAttrPrefix = ":";
```

3. 创建密钥环

以下示例创建一个 AWS KMS 密钥环。 AWS KMS 密钥环使用对称加密或非对称 RSA AWS KMS keys 来生成、加密和解密数据密钥。

该示例使用 CreateMrkMultiKeyring 创建带有对称加密 KMS 密钥的 AWS KMS 密钥环。CreateAwsKmsMrkMultiKeyring 方法可确保密钥环能够正确处理单区域密钥和多区域密钥。

```
var matProv = new MaterialProviders(new MaterialProvidersConfig());
var keyringInput = new CreateAwsKmsMrkMultiKeyringInput { Generator = kmsKeyId };
var kmsKeyring = matProv.CreateAwsKmsMrkMultiKeyring(keyringInput);
```

4. 定义 DynamoDB 表的加密配置

以下示例定义了一个 tableConfigs 映射，该映射表示此 DynamoDB 表的加密配置。

此示例将 DynamoDB 表名称指定为逻辑表名称。强烈建议您在首次定义加密配置时将 DynamoDB 表名指定为逻辑表名。

必须指定 FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT 作为明文替代。此策略继续读取和写入明文项目，读取加密项目，并使客户端做好准备以写入加密项目。

有关表加密配置中包含的值的更多信息，请参阅适用于 DynamoDB 的 AWS 数据库加密 SDK 中的加密配置。

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
```

```
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    PlaintextOverride = FORCE_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

5. 创建新的 AWS SDK DynamoDB 客户端

   以下示例使用步骤 4 中的创建了一个新 AWS 的 SDK DynamoDB 客户端**TableEncryptionConfigs**。

```
var ddb = new Client.DynamoDbClient(
    new DynamoDbTablesEncryptionConfig { TableEncryptionConfigs = tableConfigs });
```

步骤 2：写入已加密和签名项目

更新表加密配置中的明文策略，以允许客户端写入加密和签名的项目。部署好以下更改后，客户端将根据您在步骤 1 中配置的属性操作对新项目进行加密和签名。客户将能够读取明文项目以及已加密和签名的项目。

在继续执行步骤 3 之前，您必须对表格中所有现有的明文项目进行加密和签名。您无法运行单一指标或查询来快速加密您的现有明文项目。使用对您的系统最有意义的过程。例如，您可以使用异步过程，以缓慢扫描表，然后使用您定义的属性操作和加密配置重写项目。要识别表中的纯文本项目，我们建议您扫描所有不包含 AWS 数据库加密 SDK 在项目加密`aws_dbe_head`和签名时添加的和`aws_dbe_foot`属性的项目。

以下示例更新了步骤 1 中的表加密配置。您必须使用 `FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT` 更新明文替代。该策略继续读取明文项目，但也会读取和写入加密项目。使用更新后的 AWS DynamoDB 客户端创建一个新的 SDK DynamoDB 客户端。`TableEncryptionConfigs`

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
```

```
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    PlaintextOverride = FORBID_WRITE_PLAINTEXT_ALLOW_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

步骤 3：仅读取已加密和签名项目

对所有项目进行加密和签名后，请更新表加密配置中的明文替代，使其仅允许客户端读取和写入加密和签名的项目。部署好以下更改后，客户端将根据您在步骤 1 中配置的属性操作对新项目进行加密和签名。客户只能够读取已加密和签名的项目。

以下示例更新了步骤 2 中的表加密配置。您可以使用 FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT 更新明文替代，也可以从配置中移除明文策略。默认情况下，客户端仅读取和写入已加密和签名的项目。使用更新后的 AWS DynamoDB 客户端创建一个新的 SDK DynamoDB 客户端。TableEncryptionConfigs

```
Dictionary<String, DynamoDbTableEncryptionConfig> tableConfigs =
    new Dictionary<String, DynamoDbTableEncryptionConfig>();
DynamoDbTableEncryptionConfig config = new DynamoDbTableEncryptionConfig
{
    LogicalTableName = ddbTableName,
    PartitionKeyName = "partition_key",
    SortKeyName = "sort_key",
    AttributeActionsOnEncrypt = attributeActionsOnEncrypt,
    Keyring = kmsKeyring,
    AllowedUnsignedAttributePrefix = unsignAttrPrefix,
    // Optional: you can also remove the plaintext policy from your configuration
    PlaintextOverride = FORBID_WRITE_PLAINTEXT_FORBID_READ_PLAINTEXT
};
tableConfigs.Add(ddbTableName, config);
```

Rust

本主题介绍如何安装和使用版本 1。 适用于 DynamoDB 的 Rust 客户端加密库中的 x。有关使用适用于 DynamoDB 的 AWS 数据库加密 SDK 进行编程的详细信息，请参阅上的-dynamodb 存储库中的 aws-database-encryption-sdk Rust 示例。 GitHub

适用于 DynamoDB 的 AWS 数据库加密 SDK 的所有编程语言实现均可互操作。

主题

- 先决条件

先决条件

在安装适用于 DynamoDB 的 Rust 客户端加密库之前，请确保满足以下先决条件。

安装 Rust 和 Cargo

使用 r ustup 安装当前稳定版本的 Rust。

有关下载和安装 rustup 的更多信息，请参阅《货运手册》中的安装程序。

安装

适用于 DynamoDB 的 Rust 客户端加密库在 Crates.io 上以箱子形式aws-db-esdk提供。有关安装和构建库的详细信息，请参阅-dynamodb 存储库中的 README.md 文件。 aws-database-encryption-sdk GitHub

手动方式

要安装适用于 DynamoDB 的 Rust 客户端加密库，请克隆或下载-dynamodb 存储库。aws-database-encryption-sdk GitHub

安装最新版本

在您的项目目录中运行以下 Cargo 命令：

```
cargo add aws-db-esdk
```

或者在你的 Cargo.toml 中添加以下一行：

```
aws-db-esdk = "<version>"
```

使用 DynamoDB 的 Rust 客户端加密库

本主题解释了版本 1 中的一些函数和辅助类。 适用于 DynamoDB 的 Rust 客户端加密库中的 x。

有关使用适用于 DynamoDB 的 Rust 客户端加密库进行编程的详细信息，请参阅上的-dynamodb 存储库中的 aws-database-encryption-sdk Rust 示例。 GitHub

主题

项目加密程序

DynamoDB AWS 数据库加密 SDK 的核心是一个项目加密器。你可以使用版本 1。 DynamoDB 的 Rust 客户端加密库中的 x，用于通过以下方式对您的 DynamoDB 表项目进行加密、签名、验证和解密。

适用于 DynamoDB 的低级 AWS 数据库加密 SDK API

> 您可以使用表加密配置来构建 DynamoDB 客户端，该客户端会自动使用您的 DynamoDB 请求在客户端对项目进行加密和签名。`PutItem`
>
> 您必须使用适用于 DynamoDB API 的低级 AWS 数据库加密 SDK 才能使用可搜索的加密。
>
> 有关演示如何使用适用于 DynamoDB API 的低级 AWS 数据库加密 SDK 的示例，请参阅上的-dynamodb 存储库中的 basic_get_put_example.rs。 aws-database-encryption-sdk GitHub

较低级别的 **`DynamoDbItemEncryptor`**

> 较低级别的 `DynamoDbItemEncryptor` 无需调用 DynamoDB 即可直接对您的表项目进行加密、签名或解密和验证。它不会发出 DynamoDB `PutItem` 或 `GetItem` 请求。举例来说，您可以使用较低级别的 `DynamoDbItemEncryptor` 直接解密和验证已经检索到的 DynamoDB 项目。
>
> 较低级别的 `DynamoDbItemEncryptor` 不支持可搜索加密。
>
> 有关演示如何使用较低级别`DynamoDbItemEncryptor`的示例，请参阅上的-dynamodb 存储库中的 item_encrypt_decryp t.rs。 aws-database-encryption-sdk GitHub

适用于 DynamoDB 的 AWS 数据库加密 SDK 中的属性操作

属性操作决定哪些属性值经过加密和签名，哪些仅经过签名，哪些经过签名并包含在加密上下文中，哪些会被忽略。

要使用 Rust 客户端指定属性操作，请使用对象模型手动定义属性操作。通过创建一个`HashMap`对象来指定您的属性操作，其中名称-值对表示属性名称和指定操作。

指定 ENCRYPT_AND_SIGN 以对属性进行加密和签名。指定 SIGN_ONLY 以对属性进行签名，但不进行加密。指定SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT对属性进行签名并将其包含在加密上下文中。如果不对属性进行签名，也将无法对其进行加密。指定 DO_NOTHING 以忽略某个属性。

分区和排序属性必须为SIGN_ONLY或SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。如果将任何属性定义为SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT，则分区和排序属性也必须是SIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT。

> **Note**
>
> 定义属性操作后，必须定义将哪些属性排除在签名之外。为了将来更方便添加新的未签名属性，建议您选择一个不同的前缀（例如“:”）来标识您的未签名属性。在定义 DynamoDB 架构和属性操作时，将此前缀包含在标记为 DO_NOTHING 的所有属性的属性名称中。

以下对象模型演示了如何使用 Rust 客户端指定ENCRYPT_AND_SIGNSIGN_ONLYSIGN_AND_INCLUDE_IN_ENCRYPTION_CONTEXT、、和DO_NOTHING属性操作。此示例使用前缀 “:” 来标识DO_NOTHING属性。

```
let attribute_actions_on_encrypt = HashMap::from([
    ("partition_key".to_string(), CryptoAction::SignOnly),
    ("sort_key".to_string(), CryptoAction::SignOnly),
    ("attribute1".to_string(), CryptoAction::EncryptAndSign),
    ("attribute2".to_string(), CryptoAction::SignOnly),
    (":attribute3".to_string(), CryptoAction::DoNothing),
]);
```

适用于 DynamoDB 的 AWS 数据库加密 SDK 中的加密配置

使用 AWS 数据库加密 SDK 时，必须为 DynamoDB 表显式定义加密配置。加密配置中所需的值取决于您是手动定义属性操作还是使用带注释的数据类来进行定义。

以下代码段使用适用于 DynamoDB API 的 AWS 低级数据库加密 SDK 定义了 DynamoDB 表加密配置，并允许使用由不同前缀定义的未签名属性。

```
let table_config = DynamoDbTableEncryptionConfig::builder()
    .logical_table_name(ddb_table_name)
    .partition_key_name("partition_key")
    .sort_key_name("sort_key")
    .attribute_actions_on_encrypt(attribute_actions_on_encrypt)
    .keyring(kms_keyring)
```

```
    .allowed_unsigned_attribute_prefix(UNSIGNED_ATTR_PREFIX)
    // Specifying an algorithm suite is optional
    .algorithm_suite_id(
        DbeAlgorithmSuiteId::AlgAes256GcmHkdfSha512CommitKeyEcdsaP384SymsigHmacSha384,
    )
    .build()?;

let table_configs = DynamoDbTablesEncryptionConfig::builder()
    .table_encryption_configs(HashMap::from([(ddb_table_name.to_string(),
 table_config)]))
    .build()?;
```

逻辑表名

适用于您的 DynamoDB 表的逻辑表名称。

为简化 DynamoDB 还原操作，逻辑表名称以加密方式绑定到表中存储的所有数据。强烈建议您在首次定义加密配置时将 DynamoDB 表名指定为逻辑表名。必须始终指定相同的逻辑表名。要成功解密，逻辑表名称必须与加密时所指定的名称相匹配。如果您的 DynamoDB 表名称在从备份中恢复 DynamoDB 表后发生更改，则逻辑表名称可确保解密操作仍能识别该表。

允许的未签名属性

在您的属性操作中标记为 DO_NOTHING 的属性。

允许的未签名属性将告诉客户端哪些属性被排除在签名之外。客户端假设，所有的其他属性都包含在签名中。然后，在解密记录时，客户端会从您指定的允许的未签名属性中确定需要验证哪些属性以及需要忽略哪些属性。您将不能从允许的未签名属性中移除属性。

您可以通过创建一个列出所有 DO_NOTHING 属性的数组来显式定义允许的未签名属性。您还可以在命名 DO_NOTHING 属性时指定不同的前缀，并使用前缀告诉客户端哪些属性未签名。强烈建议指定一个不同的前缀，因为它可以简化未来添加新的 DO_NOTHING 属性的过程。有关更多信息，请参阅 更新您的数据模型。

如果您没有为所有 DO_NOTHING 属性指定前缀，可以配置一个 allowedUnsignedAttributes 数组，该数组将显式列出客户端在解密时遇到这些属性时应该取消签名的所有属性。您只有在绝对必要时，才应显式定义允许的未签名属性。

搜索配置（可选）

SearchConfig 将定义信标版本。

必须指定 SearchConfig 才能使用可搜索的加密或签名信标。

算法套件（可选）

`algorithmSuiteId` 定义 AWS 数据库加密 SDK 使用哪种算法套件。

除非您明确指定替代算法套件，否则 AWS 数据库加密 SDK 将使用默认算法套件。默认算法套件将 AES-GCM 算法与密钥派生、数字签名和密钥承诺结合使用。尽管默认算法套件可能适用于大多数应用程序，但您可以选择备用算法套件。例如，没有数字签名的算法套件可以满足某些信任模型的需求。有关 AWS 数据库加密 SDK 支持的算法套件的信息，请参阅AWS 数据库加密 SDK 中支持的算法套件。

要选择没有 ECDSA 数字签名的 AES-GCM 算法套件，请在表加密配置中加入以下片段。

```
.algorithm_suite_id(
    DbeAlgorithmSuiteId::AlgAes256GcmHkdfSha512CommitKeyEcdsaP384SymsigHmacSha384,
)
```

使用 AWS 数据库加密 SDK 更新项目

UpdateItem对于包含加密或签名属性的项目，数据库加密 SDK 不支持 ddb:。 AWS 要更新加密或已签名的属性，必须使用 ddb: PutItem。如果将同一个主键指定为 `PutItem` 请求中现有的项目，则新项目将完全替代现有项目。

旧版 DynamoDB 加密客户端

2023 年 6 月 9 日，我们的客户端加密库更名为 AWS 数据库加密 SDK。 AWS 数据库加密 SDK 继续支持旧版 DynamoDB 加密客户端版本。有关客户端加密库中随重命名发生更改的不同部分的更多信息，请参阅 Amazon DynamoDB Encryption Client 重命名。

要迁移到最新版本的适用于 DynamoDB 的 Java 客户端加密库，请参阅 迁移到版本 3.x。

主题

- AWS 适用于 DynamoDB 的数据库加密 SDK 版本支持
- DynamoDB 加密客户端的工作原理
- Amazon DynamoDB Encryption Client 概念
- 加密材料提供程序
- Amazon DynamoDB Encryption Client 可用的编程语言
- 更改数据模型
- 排查 DynamoDB 加密客户端应用程序中的问题

AWS 适用于 DynamoDB 的数据库加密 SDK 版本支持

旧版章节中的主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。

下表所列为在 Amazon DynamoDB 中支持客户端加密的语言和版本。

编程语言	版本	SDK 主要版本的生命周期阶段
Java	版本 1.x	End-of-Support 阶段，2022 年 7 月生效
Java	版本 2.x	正式发布（GA）
Java	版本 3.x	正式发布（GA）
Python	版本 1.x	End-of-Support 阶段，2022 年 7 月生效
Python	版本 2.x	End-of-Support 阶段，2022 年 7 月生效
Python	版本 3.x	正式发布（GA）

DynamoDB 加密客户端的工作原理

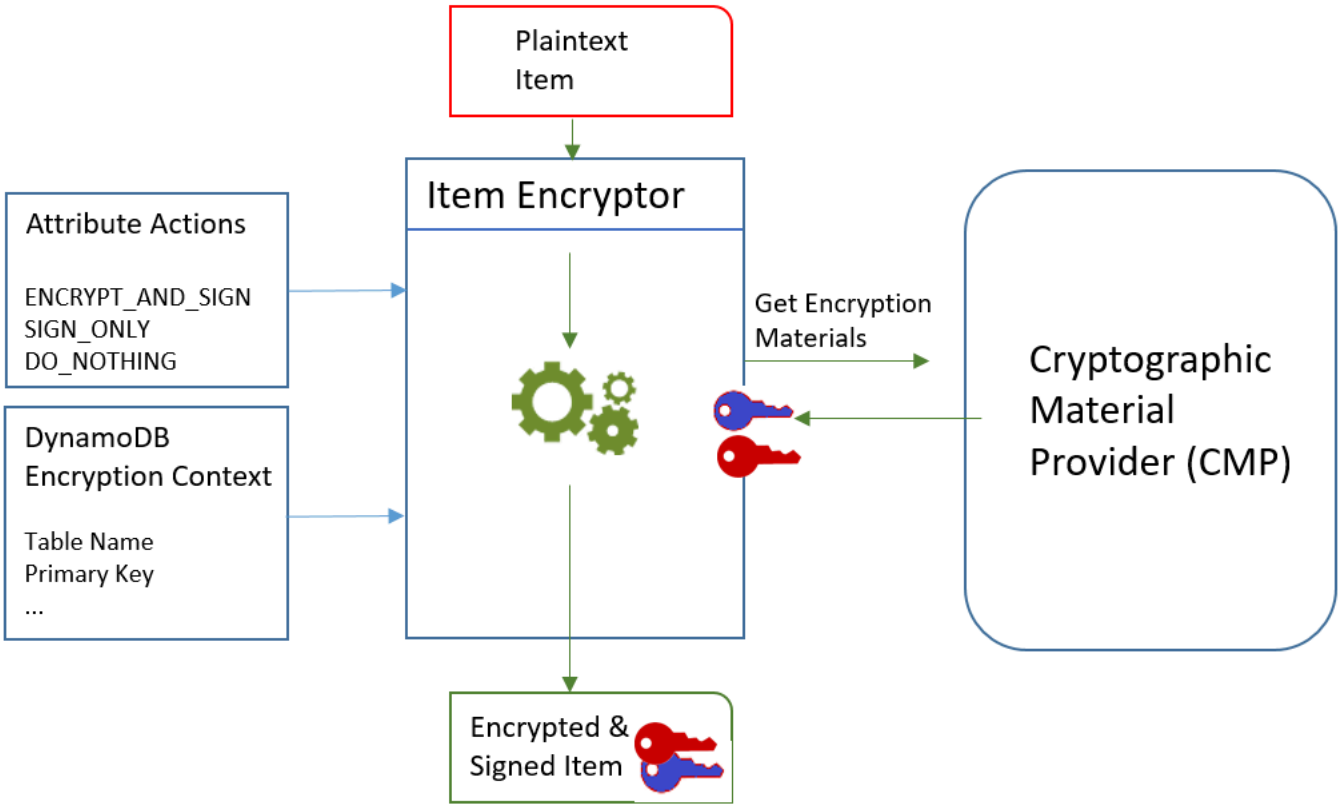
> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

DynamoDB 加密客户端专门设计为保护存储在 DynamoDB 中的数据。库包含可以直接扩展或使用的安全实施。大多数元素由抽象元素表示，因此可以创建和使用兼容的自定义组件。

为表项目加密和签名

负责对表项目进行加密、签名、验证和解密的项目加密程序是 DynamoDB 加密客户端的核心。它取得表项目信息，以及要加密和签名的项目说明，它将从您选择并配置的加密材料提供程序获取加密材料和加密材料的使用说明。

下图显示了此流程的高级视图。



要对表项目进行加密和签名，DynamoDB 加密客户端需要：

- 表的相关信息。它从提供的 DynamoDB 加密上下文获取有关表的信息。某些帮助程序从 DynamoDB 获取必需信息并创建 DynamoDB 加密上下文。

  > **Note**
  >
  > DynamoDB 加密客户端中的 DynamoDB 加密上下文与 () 和中的加密上下文无关。 AWS Key Management Service AWS KMS AWS Encryption SDK

- 要加密和签名的属性。它从提供的属性操作获取此信息。
- 加密材料，包括加密密钥和签名密钥。它从您选择并配置的加密材料提供程序 (CMP) 获取这些信息。

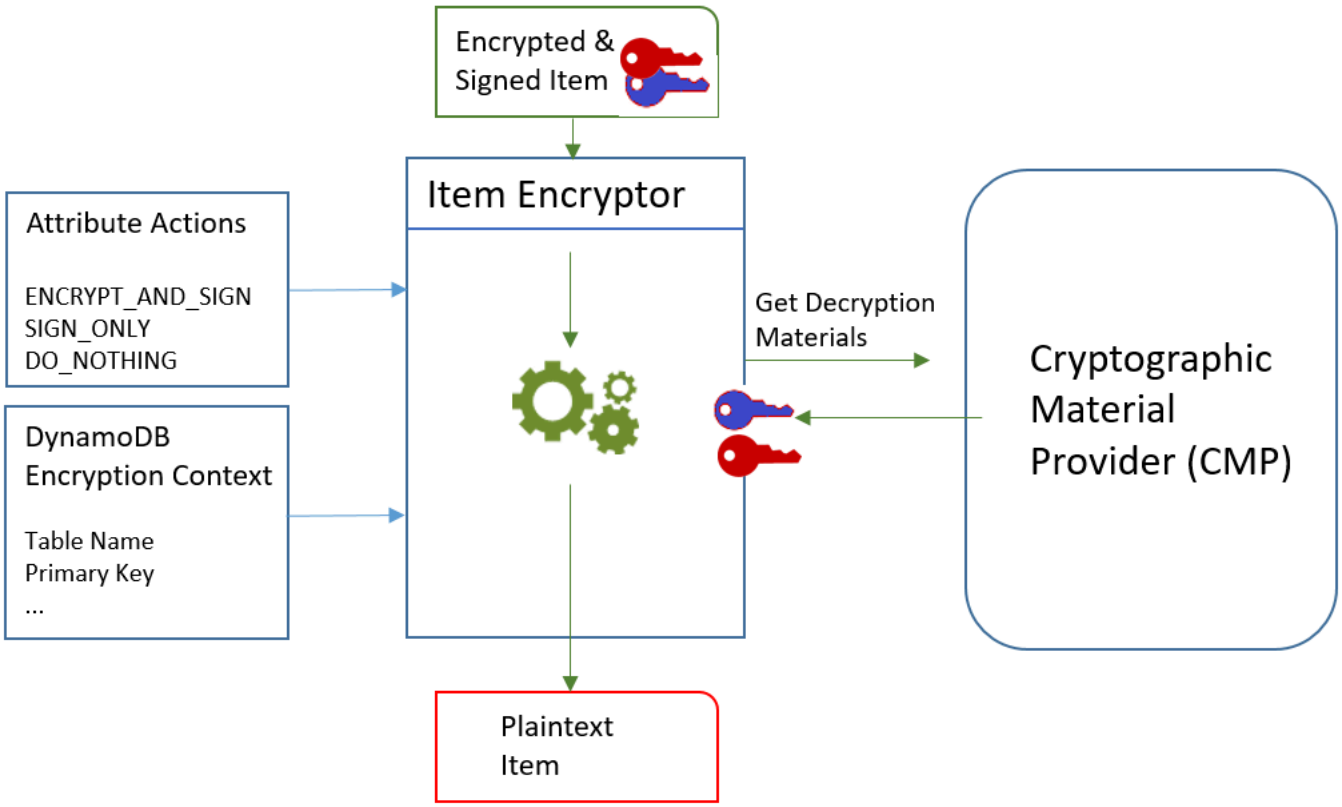
- 为项目加密和签名的说明。CMP 会将加密材料（包括加密和签名算法）使用说明添加到实际材料描述。

项目加密程序将使用所有这些元素为项目加密和签名。此外，项目加密程序将两个属性添加到项目：包含加密和签名说明（实际材料描述）的材料描述属性以及包含签名的属性。可以直接与项目加密程序交互，或使用与项目加密程序交互的帮助程序功能以实施安全默认行为。

结果是包含已加密和已签名数据的 DynamoDB 项目。

验证和解密表项目

这些组件还一起运行来验证和解密项目，如下图所示。



要验证和解密项目，DynamoDB 加密客户端要相同的组件、配置相同的组件或专门为解密项目设计的组件，如下所示：

- 来自 DynamoDB 加密上下文的有关表的信息。
- 要验证和解密的属性。它将从属性操作获取这些属性。
- 您选择和配置的加密材料提供程序（CMP）中的解密材料，包括验证和解密密钥。

已加密项目不包括为它加密所使用的 CMP 的任何记录。您必须提供相同的 CMP、配置相同的 CMP 或设计为解密项目的 CMP。

- 有关如何加密项目和为项目签名的信息，包括加密和签名算法。客户端将从项目的材料描述属性中获取这些信息。

项目加密程序将使用所有这些元素验证和解密项目。它还将删除材料描述和签名属性。结果是明文 DynamoDB 项目。

Amazon DynamoDB Encryption Client 概念

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

本主题介绍 Amazon DynamoDB Encryption Client 中使用的概念和术语。

要了解 DynamoDB 加密客户端的组件如何交互，请参阅 DynamoDB 加密客户端的工作原理。

主题

- 加密材料提供程序 (CMP)
- 项目加密程序
- 属性操作
- 材料描述
- DynamoDB 加密上下文
- 提供程序存储

加密材料提供程序 (CMP)

在实施 DynamoDB 加密客户端时，第一批任务中有一个任务是选择加密材料提供程序（CMP）（又称为加密材料提供程序）。您的选择决定了剩下的大部分实施操作。

加密材料提供程序 (CMP) 收集、汇编并返回项目加密程序用于为表项目加密和签名的加密材料。CMP 确定要使用的加密算法以及如何生成和保护加密和签名密钥。

CMP 与项目加密程序交互。项目加密程序从 CMP 请求加密或解密材料，而 CMP 将这些材料返回给项目加密程序。然后，项目加密程序使用加密材料为项目加密和签名，或验证和解密项目。

在配置客户端时指定 CMP。您可以创建兼容的自定义 CMP，也可以使用库 CMPs 中的众多自定义 CMP 之一。大多数 CMPs 都适用于多种编程语言。

项目加密程序

项目加密程序是为 DynamoDB 加密客户端执行加密操作的低级别组件。它从加密材料提供程序 (CMP) 请求加密材料，然后使用 CMP 返回的材料为表项目加密和签名，或验证和解密表项目。

可以直接与项目加密程序交互或使用库提供的帮助程序。例如，适用于 Java 的 DynamoDB 加密客户端包含可与 `DynamoDBMapper` 一起使用的 `AttributeEncryptor` 帮助程序类，而不是直接与 `DynamoDBEncryptor` 项目加密程序交互。Python 库包含与项目加密程序交互的 `EncryptedTable`、`EncryptedClient` 和 `EncryptedResource` 帮助程序类。

属性操作

属性操作告知项目加密程序将对项目的每个属性执行哪些操作。

属性操作值可以是以下任何值：

- 加密和签名 – 加密属性值。在项目签名中包含属性（名称和值）。
- 仅签名 – 在项目签名中包含属性。
- 不执行任何操作 – 不为属性加密或签名。

对于可能存储敏感数据的任何属性，请使用 Encrypt and sign (加密和签名)。对于主键属性（分区键和排序键），使用 Sign only。不会为材料描述属性和签名属性签名或加密。无需为这些属性指定属性操作。

仔细选择属性操作。如有怀疑，请使用 Encrypt and sign (加密和签名)。一旦使用 DynamoDB 加密客户端来保护表项，就无法在不冒签名验证错误风险的情况下更改属性的操作。有关更多信息，请参阅更改数据模型。

> **Warning**
>
> 请勿加密主键属性。它们必须保留为明文，以便 DynamoDB 查找项目而无需运行全表扫描。

如果 DynamoDB 加密上下文标识您的主键属性，则客户端将在您尝试加密这些属性时引发错误。

对于每种编程语言而言，用于指定属性操作的技术各不相同。此技术还可能特定于使用的帮助程序类。

有关详细信息，请参阅编程语言对应的文档。

- Python
- Java

材料描述

已加密表项目的材料描述 包含有关如何为表项目加密和签名的信息（如加密算法）。加密材料提供程序 (CMP) 将在汇编用于加密和签名的加密材料时记录材料描述。之后，当它需要汇编加密材料以验证和解密项目时，它将使用材料描述作为指南。

在 DynamoDB 加密客户端中，材料描述引用三个相关元素：

请求的材料描述

某些加密材料提供程序 (CMPs) 允许您指定高级选项，例如加密算法。要指明您的选择，请将名称-值对添加到表项目加密请求中 DynamoDB 加密上下文的材料描述属性中。此元素也称为请求的材料描述。请求的材料描述中的有效值由您选择的 CMP 定义。

> **Note**
>
> 由于材料描述会覆盖安全默认值，因此建议忽略请求的材料描述，除非有不得已的原因要使用它。

实际材料描述

加密材料提供者 (CMPs) 返回的材料描述称为实际材料描述。它描述 CMP 在汇编加密材料时使用的实际值。它一般包括请求的描述材料（如有），请求的描述材料有增加和更改。

材料描述属性

客户端将实际材料描述保存在已加密项目的材料描述属性 中。材料描述属性名称为 `amzn-ddb-map-desc` 并且其值为实际材料描述。客户端将使用材料描述属性中的值验证和解密项目。

DynamoDB 加密上下文

DynamoDB 加密上下文 为加密材料提供程序（CMP）提供有关表和项目的信息。在高级实施中，DynamoDB 加密上下文可以包括请求的材料描述。

在对表项目进行加密时，DynamoDB 加密上下文将以加密方式绑定到加密的属性值。当您解密时，如果 DynamoDB 加密上下文与用于加密的 DynamoDB 加密上下文不是区分大小写的完全匹配，则解密操作将失败。如果您与项目加密程序直接交互，则必须在调用加密或解密方法时提供 DynamoDB 加密上下文。大多数帮助程序将创建 DynamoDB 加密上下文。

> **Note**
>
> DynamoDB 加密客户端中的 DynamoDB 加密上下文与 () 和中的加密上下文无关。 AWS Key Management Service AWS KMS AWS Encryption SDK

DynamoDB 加密上下文可以包含以下字段。所有字段和值均为可选项。

- 表名
- 分区键名称
- 排序键名称
- 属性名称/值对
- 请求的材料描述

提供程序存储

提供商存储是返回加密材料提供者 (CMPs) 的组件。提供商商店可以创建 CMPs 或从其他来源（例如其他提供商商店）获取它们。提供商存储将其创建的 CMPs 版本保存在永久存储中，其中每个存储的 CMP 都由请求者的材料名称和版本号标识。

DynamoDB 加密客户端中的最新提供程序来自提供程序存储，但您可以使用提供程序存储为 CMPs 任何组件提供该提供程序。 CMPs 每个最新提供程序都与一个提供商存储相关联，但一个提供商存储可以 CMPs 向多个主机的多个请求者提供服务。

提供商商店按需创建新版本，并返回新版本和现有版本。 CMPs 它还将返回指定材料名称的最新版本号。这使请求者知道提供程序存储何时具有它可请求的 CMP 的新版本。

DynamoDB 加密客户端包括 MetaStore一个，它是一个提供商存储，它使用存储在 DynamoDB 中并使用内部 DynamoDB 加密客户端加密的密钥创建 Wr CMPs apped。

了解更多：

- 提供程序存储：Java、Python
- MetaStore: Java、Python

加密材料提供程序

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

您在使用 DynamoDB 加密客户端时所做的最重要的决策是选择加密材料提供程序（CMP）。CMP 组装加密材料并将其返回到项目加密程序。它还确定如何生成加密密钥和签名密钥，新密钥材料是否对于每个项目进行生成或重复使用以及所使用的加密和签名算法。

您可以选择 DynamoDB 加密客户端库中提供的实施中的 CMP 或构建兼容的自定义 CMP。您的 CMP 选择还可能取决于使用的编程语言。

本主题介绍了最常见的内容， CMPs 并提供了一些建议，以帮助您为应用程序选择最佳方案。

Direct KMS 材料提供程序

Direct KMS 材料提供程序借助 AWS KMS key 保护您的表项目，该主密钥绝不会让 AWS Key Management Service（AWS KMS）处于不加密状态。您的应用程序不必生成或管理任何加密材料。由于该 AWS KMS key 提供程序使用为每个项目生成唯一的加密和签名密钥，因此它 AWS KMS 每次加密或解密项目时都会调用。

如果您使用 AWS KMS 并且每笔交易一次 AWS KMS 调用适合您的应用程序，那么此提供商是一个不错的选择。

有关更多信息，请参阅 Direct KMS 材料提供程序。

已包装的材料提供程序（已包装的 CMP）

利用已包装的材料提供程序（已包装的 CMP），您可以在 DynamoDB 加密客户端外部生成和管理包装密钥和签名密钥。

已包装的 CMP 会为每个项目生成唯一加密密钥。然后，它会使用您提供的包装（或解开包装）密钥和签名密钥。因此，您可确定如何生成包装密钥和签名密钥以及这些密钥对于每个项目是唯一的还是可重复使用。Wrapped CMP 是 Di rect KMS 提供程序的安全替代方案，适用于不使用加密材料 AWS KMS 且可以安全管理加密材料的应用程序。

有关更多信息，请参阅 已包装的材料提供程序。

最新提供程序

最新提供程序是一个加密材料提供程序（CMP），旨在处理提供程序存储。它 CMPs从提供商商店获取，并从中获取返回的加密材料。 CMPs最新提供程序通常使用每个 CMP 来满足加密材料的多个请求，但您可以使用提供程序存储的功能来控制可重复使用材料的范围，决定轮换其 CMP 的频率以及甚至在不更改最新提供程序的情况下更改所使用的 CMP 的类型。

您可以结合使用最新提供程序与任何兼容的提供程序存储。DynamoDB 加密客户端包括 MetaStore 一个，这是一个返回 Wrapped 的提供商存储。 CMPs

对于需要最大程度地减少对加密源的调用的应用程序，以及能够在不违反应用程序的安全性要求的情况下重复使用某些加密材料的应用程序，最新提供程序是个很好的选择。例如，它允许您在 in AWS Key Management Service(AWS KMS) 下保护您的加密材料，而无需AWS KMS key在 AWS KMS 每次加密或解密项目时都调用。

有关更多信息，请参阅 最新提供程序。

静态材料提供程序

静态材料提供商专为测试、 proof-of-concept演示和传统兼容性而设计。它不会为每个项目生成任何唯一加密材料。它将返回您提供的相同加密密钥和签名密钥，而且这些密钥会直接用于加密、解密和签署您的表项目。

> **Note**
>
> Java 库中的非对称静态提供程序不是一种静态提供程序。它仅提供已包装的 CMP 的替代构造函数。它在生产使用时是安全的，但您应尽可能地直接使用已包装的 CMP。

主题

- Direct KMS 材料提供程序
- 已包装的材料提供程序
- 最新提供程序

- 静态材料提供程序

Direct KMS 材料提供程序

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

Direct KMS 材料提供程序（Direct KMS 提供程序）借助从不让 AWS Key Management Service（AWS KMS）处于不加密状态的 AWS KMS key 保护您的表项目。此加密材料提供程序为每个表项目返回唯一的加密密钥和签名密钥。为此，它会在您 AWS KMS 每次加密或解密项目时调用。

如果您以高频率和大规模处理 DynamoDB 项目，则可能会超出限制，从而导致处理 AWS KMS requests-per-second延迟。如果需要超出限制，请在 AWS 支持 中心并创建案例。您也可以考虑使用密钥重复次数使用有限的加密材料提供程序，例如最新提供程序。

要使用 Direcrypt KMS 提供程序 AWS 账户，调用者必须至少拥有一个权限 AWS KMS key，才能在上调用GenerateDataKey和解密操作。 AWS KMS key AWS KMS key 必须是对称加密密钥；DynamoDB 加密客户端不支持非对称加密。如果您使用的是 DynamoDB 全局表，则可能需要指定一个 AWS KMS 多区域密钥。有关更多信息，请参阅 使用方法。

> **Note**
>
> 当您使用 Direct KMS 提供程序时，您的主密钥属性的名称和值将以纯文本形式显示在AWS KMS 加密上下文和相关 AWS CloudTrail AWS KMS 操作日志中。但是，DynamoDB 加密客户端从不公开任意加密属性值的明文。

直接 KMS 提供程序是 DynamoDB 加密客户端支持的几个加密材料提供商 (CMPs) 之一。有关另一个的信息 CMPs，请参阅加密材料提供程序。

有关示例代码，请参阅：

- Java：AwsKmsEncryptedItem
- Python: aws-kms-encrypted-table，aws-kms-encrypted-item

主题

使用方法

要创建 Direct KMS 提供程序，请使用密钥 ID 参数在您的账户中指定对称加密 KMS 密钥。密钥 ID 参数的值可以是 AWS KMS key的密钥 ID、密钥 ARN、别名名称或别名 ARN。有关密钥标识符的详细信息，请参阅《AWS Key Management Service 开发人员指南》中的密钥标识符。

Direct KMS 提供程序需要对称的加密 KMS 密钥。不能使用非对称 KMS 密钥。但是，可以使用多区域 KMS 键、包含导入的密钥材料的 KMS 密钥，或自定义密钥存储中的 KMS 密钥。您必须拥有 KMS 密钥的 kms: GenerateDataKey 和 kms: decrypt 权限。因此，您必须使用客户托管的密钥，而不是托 AWS 管或 AWS 拥有的 KMS 密钥。

适用于 Python 的 DynamoDB 加密客户端在密钥 ID 参数值（如果包含密钥 ID 参数值）中确定从该区域 AWS KMS 调用的区域。否则，它将使用 AWS KMS 客户端中的区域（如果您指定）或您在中配置的区域 适用于 Python (Boto3) 的 AWS SDK。有关 Python 中区域选择的信息，请参阅 Python AWS 开发工具包 (Boto3) API 参考中的配置。

如果您指定的客户端包含区域，则适用于 Java 的 DynamoDB 加密客户端 AWS KMS 会确定从客户端中的区域进行 AWS KMS 调用的区域。否则，它将使用您在 适用于 Java 的 AWS SDK中配置的区域。有关中区域选择的信息 适用于 Java 的 AWS SDK，请参阅《 适用于 Java 的 AWS SDK 开发者指南》中的AWS 区域 选择。

Java

```
// Replace the example key ARN and Region with valid values for your application
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

Python

以下示例使用密钥 ARN 指定 AWS KMS key。如果您的密钥标识符不包括 AWS 区域，则 DynamoDB 加密客户端将从已配置的 Botocore 会话（如果有）或 Boto 默认会话中获取区域。

```
# Replace the example key ID with a valid value
kms_key = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key)
```

如果您使用的是 Amazon DynamoDB 全局表，我们建议您使用多区域密钥对数据进行 AWS KMS 加密。多区域密钥不同 AWS 区域 ，可以互换使用，因为它们具有相同的密钥 ID 和密钥材料。 AWS KMS keys 有关详细信息，请参阅《AWS Key Management Service 开发人员指南》中的使用多区域密钥。

> **Note**
>
> 如果您使用的是全局表版本 2017.11.29，则必须设置属性操作，这样，保留的复制字段就不会被加密或签名。有关更多信息，请参阅 旧版本全局表存在的问题。

要在 DynamoDB 加密客户端中使用多区域密钥，请创建多区域密钥并将其复制到应用程序运行所在的区域。然后将 Direct KMS 提供程序配置为使用 DynamoDB 加密客户端调用 AWS KMS所在区域中的多区域密钥。

以下示例将 DynamoDB 加密客户端配置为在美国东部（弗吉尼亚州北部）（us-east-1）区域中的加密数据，并使用多区域密钥，在美国西部（俄勒冈州）（us-west-2）区域中对其进行解密。

Java

在此示例中，DynamoDB 加密客户端在客户端中获取从该区域进行 AWS KMS 调用的区域。 AWS KMS `keyArn` 值标识同一区域中的多区域密钥。

```
// Encrypt in us-east-1

// Replace the example key ARN and Region with valid values for your application
final String usEastKey = 'arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
final String region = 'us-east-1'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, usEastKey);
```

```
// Decrypt in us-west-2
```

```
// Replace the example key ARN and Region with valid values for your application
final String usWestKey = 'arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
final String region = 'us-west-2'

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, usWestKey);
```

Python

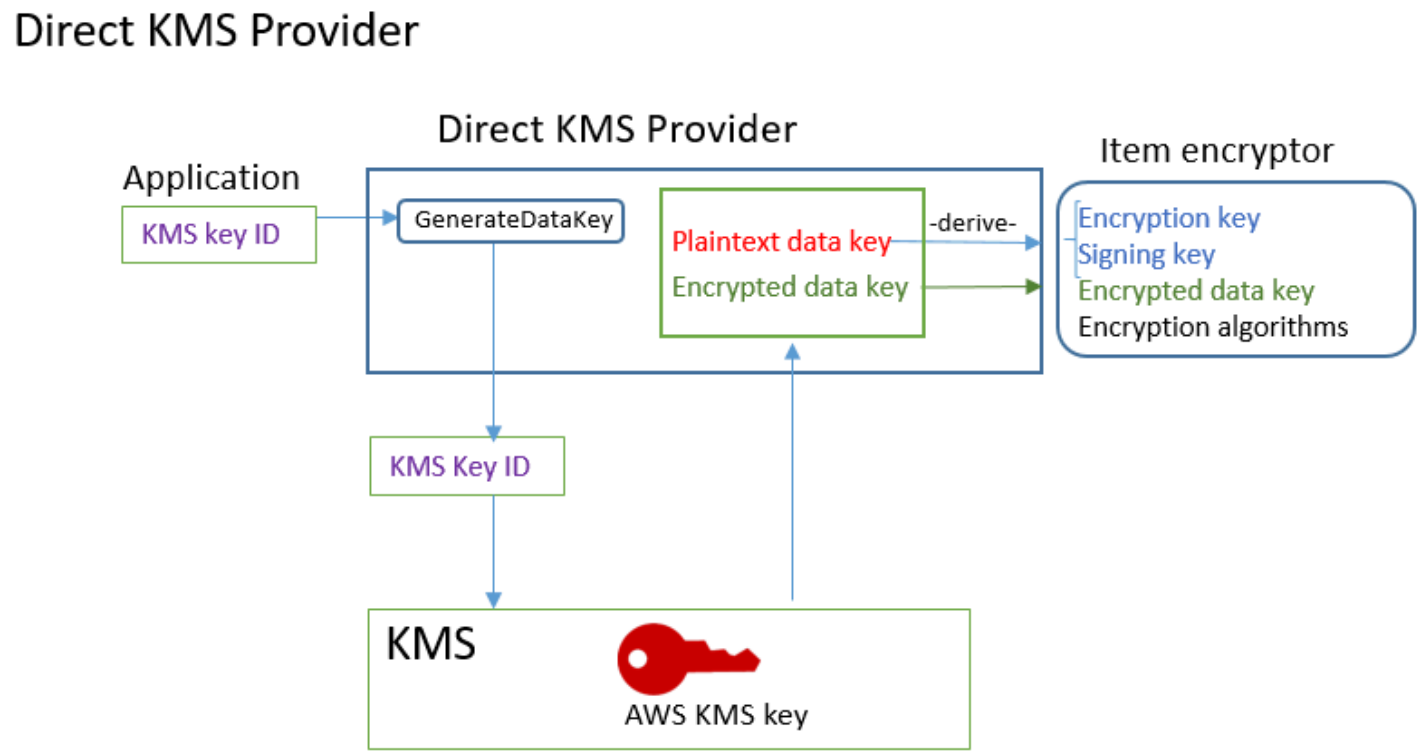
在此示例中，DynamoDB 加密客户端在密钥 ARN 中获取了从该区域进行 AWS KMS 呼叫的区域。

```
# Encrypt in us-east-1

# Replace the example key ID with a valid value
us_east_key = 'arn:aws:kms:us-east-1:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=us_east_key)
```

```
# Decrypt in us-west-2

# Replace the example key ID with a valid value
us_west_key = 'arn:aws:kms:us-west-2:111122223333:key/
mrk-1234abcd12ab34cd56ef1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=us_west_key)
```

工作原理

Direct KMS 提供程序返回受您指定的 AWS KMS key 保护的加密密钥和签名密钥，如下图所示。



- 要生成加密材料，Direct KMS 提供程序会要求 AWS KMS 使用您指定的为每个项目生成一个 AWS KMS key 唯一的数据密钥。它从数据密钥的明文副本中派生项目的加密密钥和签名密钥，然后返回加密密钥和签名密钥以及加密的数据密钥，后者存储在项目的材料说明属性中。

  项目加密程序使用加密密钥和签名密钥并尽快将它们从内存中删除。仅其派生自的数据密钥的加密副本保存在加密项目中。
- 要生成解密材料，Direct KMS 提供商会要求 AWS KMS 解密加密的数据密钥。然后，它从明文数据密钥派生验证密钥和签名密钥，然后将这些密钥返回至项目加密程序。

  项目加密程序会验证项目，而且如果验证成功，会解密加密的值。然后，它会尽快从内存中删除这些密钥。

获取加密材料

本部分详细介绍了 Direct KMS 提供程序收到来自项目加密程序的加密材料请求时的输入、输出和处理。

输入（来自应用程序）

- 的密钥 ID AWS KMS key。

输入（来自项目加密程序）

- DynamoDB 加密上下文

输出（至项目加密程序）

- 加密密钥（明文）
- 签名密钥
- 在实际材料描述中：这些值保存在客户端将添加到项目的材料描述属性中。
  - amzn-ddb-env-key: Base64 编码的数据密钥由加密 AWS KMS key
  - amzn-ddb-env-alg: 加密算法，默认为 AES/256
  - amzn-ddb-sig-alg: 签名算法，默认情况下，Hmac /256 SHA256
  - amzn-ddb-wrap-alg: kms

Processing

1. Direct KMS 提供者发送 AWS KMS 请求，要求使用指定的 AWS KMS key 为该项目生成唯一的数据密钥。该操作会返回明文密钥以及由 AWS KMS key加密的副本。这称为初始密钥材料。

   请求包括 AWS KMS 加密上下文中的以下明文值。这些非密钥值以加密方式绑定到加密对象，因此，解密时需要相同的加密上下文。您可以使用这些值 AWS KMS 在AWS CloudTrail 日志中标识对的调用。

   - amzn-ddb-env-alg — 加密算法，默认为 AES/256
   - amzn-ddb-sig-alg — 签名算法，默认为 Hmac /256 SHA256
   - （可选）aws-kms-table— *`table name`*
   - （可选）*`partition key name`*—*`partition key value`*（二进制值采用 Base64 编码）
   - （可选）*`sort key name`*—*`sort key value`*（二进制值采用 Base64 编码）

   直接 KMS 提供程序从该项目的 DynamoDB AWS KMS 加密上下文中获取加密上下文的值。如果 DynamoDB 加密上下文不包含值（例如表名），则加密上下文中将省略该名称/值对。 AWS KMS
2. Direct KMS 提供程序从数据密钥派生对称加密密钥和签名密钥。默认情况下，它使用安全哈希算法 (SHA) 256 和RFC5869 基于 HMAC 的密钥派生函数来派生 256 位 AES 对称加密密钥和 256 位 HMAC-SHA-256 签名密钥。
3. Direct KMS 提供程序将输出返回到项目加密程序。

4. 项目加密程序通过使用在实际材料说明中指定的算法，使用加密密钥加密指定的属性并使用签名密钥签署它们。它会尽快从内存中删除明文密钥。

获取解密材料

本部分详细介绍了 Direct KMS 提供程序在从项目加密程序收到解密材料的请求时的输入、输出和处理。

输入（来自应用程序）

- 的密钥 ID AWS KMS key。

  密钥 ID 的值可以是 AWS KMS key的密钥 ID、密钥 ARN、别名名称或别名 ARN。未包含在密钥 ID 中的任何值，如区域，必须在 AWS 命名配置文件中可用。密钥 ARN 提供了 AWS KMS 需要的所有值。

输入（来自项目加密程序）

- 包含材料描述属性内容的 DynamoDB 加密上下文的副本。

输出（至项目加密程序）

- 加密密钥（明文）
- 签名密钥

Processing

1. Direct KMS 提供程序从加密项目中的材料描述属性获取加密数据密钥。
2. 它要求 AWS KMS 使用指定的 AWS KMS key 来解密加密的数据密钥。此操作会返回明文密钥。

   此请求必须使用用于生成和加密数据密钥的相同的 AWS KMS 加密上下文。

   - aws-kms-table – *table name*
   - *partition key name*—*partition key value*（二进制值采用 Base64 编码）
   - （可选）*sort key name*—*sort key value*（二进制值采用 Base64 编码）
   - amzn-ddb-env-alg — 加密算法，默认为 AES/256
   - amzn-ddb-sig-alg — 签名算法，默认为 Hmac /256 SHA256

3. Direct KMS 提供商使用安全哈希算法 (SHA) 256 和RFC5869 基于 HMAC 的密钥派生函数从数据密钥中派生 256 位 AES 对称加密密钥和 256 位 HMAC-SHA-256 签名密钥。
4. Direct KMS 提供程序将输出返回到项目加密程序。
5. 项目加密程序将使用此签名密钥验证项目。如果它成功，则会使用对称加密密钥来解决加密的属性值。这些操作使用在实际材料说明中指定的加密和签名算法。项目加密程序它会尽快从内存中删除明文密钥。

已包装的材料提供程序

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

利用已包装的材料提供程序（已包装的 CMP），您可以通过 DynamoDB 加密客户端的任何源使用包装密钥和签名密钥。Wrapped CMP 不依赖于任何 AWS 服务。但是，必须在客户端之外生成和管理包装密钥与签名密钥，包括提供正确的密钥来验证和解密项目。

已包装的 CMP 将为每个项目生成一个唯一项目加密密钥。它使用提供的包装密钥包装项目加密密钥并将已包装的项目加密密钥保存在项目的材料描述属性中。由于包装密钥和签名密钥是您提供的，因此由您确定包装密钥和签名密钥的生成方式以及这些密钥对每个项目是否唯一或是否会重复使用。

已包装的 CMP 是安全实现，并且是可管理加密材料的应用程序的不二选择。

Wrapped CMP 是 DynamoDB 加密客户端支持的几个加密材料提供程序 (CMPs) 之一。有关另一个的信息 CMPs，请参阅加密材料提供程序。

有关示例代码，请参阅：

- Java：AsymmetricEncryptedItem
- Python: wrapped-rsa-encrypted-table，wrapped-symmetric-encrypted-table

主题

- 使用方法

- 工作原理

使用方法

要创建已包装的 CMP，请指定包装密钥（加密时需要）、解开包装密钥（解密时需要）以及签名密钥。加密和解密项目时，必须提供密钥。

包装密钥、解开包装密钥和签名密钥可以是对称密钥或非对称密钥对。

Java

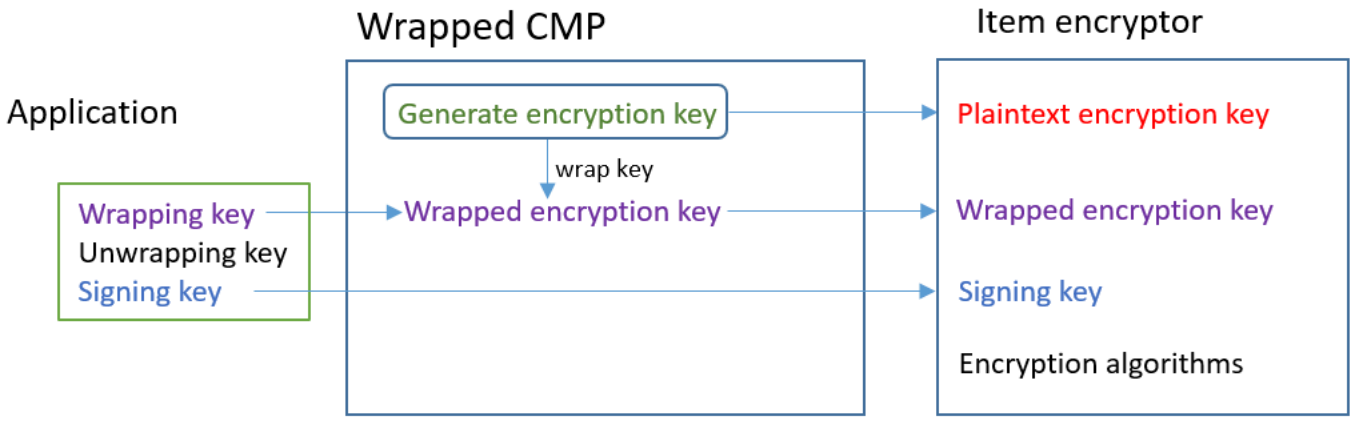
```
// This example uses asymmetric wrapping and signing key pairs
final KeyPair wrappingKeys = ...
final KeyPair signingKeys = ...

final WrappedMaterialsProvider cmp =
    new WrappedMaterialsProvider(wrappingKeys.getPublic(),
                                 wrappingKeys.getPrivate(),
                                 signingKeys);
```

Python

```
# This example uses symmetric wrapping and signing keys
wrapping_key = ...
signing_key  = ...

wrapped_cmp = WrappedCryptographicMaterialsProvider(
    wrapping_key=wrapping_key,
    unwrapping_key=wrapping_key,
    signing_key=signing_key
)
```

工作原理

已包装的 CMP 将为每个项目生成一个新的项目加密密钥。它将使用提供的包装密钥、解开包装密钥和签名密钥，如下图中所示。



获取加密材料

本部分详述了已包装的材料提供程序（已包装的 CMP）在收到加密材料请求时的输入、输出和处理。

输入（来自应用程序）

- 包装密钥：高级加密标准 (AES) 对称密钥或 RSA 公钥。在加密任何属性值时都需要。否则，它为可选项，将忽略。
- 解开包装密钥：可选，将忽略。
- 签名密钥

输入（来自项目加密程序）

- DynamoDB 加密上下文

输出（至项目加密程序）：

- 明文项目加密密钥
- 签名密钥（保持不变）
- 实际材料描述：这些值保存在客户端添加到项目的材料描述属性中。
  - `amzn-ddb-env-key`：Base64 编码的已包装项目加密密钥
  - `amzn-ddb-env-alg`：用于加密项目的加密算法。默认值为 AES-256-CBC。
  - `amzn-ddb-wrap-alg`：已包装的 CMP 用于包装项目加密密钥的包装算法。如果包装密钥为 AES 密钥，则将使用未填充的 `AES-Keywrap` 包装此密钥，如 RFC 3394 中所定义。如果包装密钥是 RSA 密钥，则使用带填充的 RSA OAEP 对密钥进行加密。 MGF1

Processing

加密项目时，将传入包装密钥和签名密钥。解开包装密钥为可选项，将忽略。

1. 已包装的 CMP 将为每个表项目生成一个唯一对称项目加密密钥。
2. 它使用指定的包装密钥包装项目加密密钥。之后，它将尽快从内存中删除此密钥。
3. 它将返回明文项目加密密钥、提供的签名密钥、包含已包装项目加密密钥的实际材料描述以及加密算法和包装算法。
4. 项目加密程序将使用此明文加密密钥加密项目。它使用提供的签名密钥为项目签名。之后，它将尽快从内存中删除明文密钥。它将实际材料描述中的字段（包括已包装的加密密钥 (amzn-ddb-env-key)）复制到项目的材料描述属性中。

获取解密材料

本部分详述了已包装的材料提供程序（已包装的 CMP）在收到解密材料请求时的输入、输出和处理。

输入（来自应用程序）

- 包装密钥：可选，将忽略。
- 解开包装密钥：同一高级加密标准 (AES) 对称密钥或对应加密所用 RSA 公有密钥的 RSA 私有密钥。在加密任何属性值时都需要。否则，它为可选项，将忽略。
- 签名密钥

输入（来自项目加密程序）

- 包含材料描述属性内容的 DynamoDB 加密上下文的副本。

输出（至项目加密程序）

- 明文项目加密密钥
- 签名密钥（保持不变）

Processing

解密项目时，将传入解开包装密钥和签名密钥。包装密钥为可选项，将忽略。

1. 已包装的 CMP 将从项目的材料描述属性中获取已包装的项目加密密钥。

2. 它使用解开包装密钥和算法解开包装项目加密密钥。
3. 它将明文项目加密密钥、签名密钥以及加密和签名算法返回项目加密程序。
4. 项目加密程序将使用此签名密钥验证项目。如果验证成功，则项目加密程序将使用项目加密密钥解密项目。之后，它将尽快从内存中删除明文密钥。

最新提供程序

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

最新提供程序是一个加密材料提供程序（CMP），旨在处理提供程序存储。它 CMPs 从提供商商店获取，并从中获取返回的加密材料。 CMPs它通常使用每个 CMP 来满足针对加密材料的多个请求。但您可以使用其提供程序存储的功能来控制材料被重复使用的程度，确定其 CMP 被轮换的频率甚至是更改它使用的 CMP 的类型而不更改最新提供程序。

> **Note**
>
> 与“最新提供程序”的 `MostRecentProvider` 符号关联的代码可能会在进程的生命周期内将加密材料存储在内存中。它可能会使调用方使用他们不再有权使用的密钥。`MostRecentProvider` 符号在受支持的较早版本的 DynamoDB 加密客户端中已被弃用，并已从 2.0.0 版本中移除。它被 `CachingMostRecentProvider` 符号所取代。有关更多信息，请参阅 最新提供程序的更新。

对于需要最大程度地减少对提供程序存储及其加密源的调用的应用程序，以及能够在不违反应用程序的安全性要求的情况下重复使用某些加密材料的应用程序，最新提供程序是一个很好的选择。例如，它允许您在 in AWS Key Management Service(AWS KMS) 下保护您的加密材料，而无需AWS KMS key在 AWS KMS 每次加密或解密项目时都调用。

您选择的提供商存储决定了最新提供程序使用的类型以及它获得新 CMP 的频率。 CMPs 您可以将任何兼容的提供程序存储与最新提供程序结合使用，包括您设计的自定义提供程序存储。

DynamoDB 加密客户端包括MetaStore一个用于创建和返回包装材料提供者（已包装）的。 CMPs将其生成的 Wrap CMPs ped 的多个版本 MetaStore 保存在内部 DynamoDB 表中，并通过 DynamoDB 加密客户端的内部实例使用客户端加密对其进行保护。

您可以将配置 MetaStore 为使用任何类型的内部 CMP 来保护表中的材料，包括生成受您保护的加密材料的 Di rect KMS 提供程序 AWS KMS key、使用您提供的封装和签名密钥的 Wrapped CMP，或者您设计的兼容自定义 CMP。

有关示例代码，请参阅：

- Java：MostRecentEncryptedItem
- Python：most_recent_provider_encrypted_table

主题

使用方法

要创建最新提供程序，您需要创建和配置一个提供程序存储，然后创建使用该提供程序存储的最新提供程序。

以下示例说明如何创建使用的最新提供程序， MetaStore 并使用直接 KMS 提供程序提供的加密材料保护其内部 DynamoDB 表中的版本。这些示例使用 `CachingMostRecentProvider` 符号。

每个最新提供程序都有一个用于在 MetaStore 表 CMPs 中标识其名称的名称、一个 time-to-live(TTL) 设置和一个决定缓存可以容纳多少条目的缓存大小设置。这些示例将缓存大小设置为 1000 个条目，并将 TTL 设置为 60 秒。

Java

```
// Set the name for MetaStore's internal table
final String keyTableName = 'metaStoreTable'

// Set the Region and AWS KMS key
final String region = 'us-west-2'
final String keyArn = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
```

```
// Set the TTL and cache size
final long ttlInMillis = 60000;
final long cacheSize = 1000;

// Name that identifies the MetaStore's CMPs in the provider store
final String materialName = 'testMRP'

// Create an internal DynamoDB client for the MetaStore
final AmazonDynamoDB ddb =
 AmazonDynamoDBClientBuilder.standard().withRegion(region).build();

// Create an internal Direct KMS Provider for the MetaStore
final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider kmsProv = new DirectKmsMaterialProvider(kms,
 keyArn);

// Create an item encryptor for the MetaStore,
// including the Direct KMS Provider
final DynamoDBEncryptor keyEncryptor = DynamoDBEncryptor.getInstance(kmsProv);

// Create the MetaStore
final MetaStore metaStore = new MetaStore(ddb, keyTableName, keyEncryptor);

//Create the Most Recent Provider
final CachingMostRecentProvider cmp = new CachingMostRecentProvider(metaStore,
 materialName, ttlInMillis, cacheSize);
```

Python

```
# Designate an AWS KMS key
kms_key_id = 'arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'

# Set the name for MetaStore's internal table
meta_table_name = 'metaStoreTable'

# Name that identifies the MetaStore's CMPs in the provider store
material_name = 'testMRP'

# Create an internal DynamoDB table resource for the MetaStore
meta_table = boto3.resource('dynamodb').Table(meta_table_name)
```

```
# Create an internal Direct KMS Provider for the MetaStore
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)

# Create the MetaStore with the Direct KMS Provider
meta_store = MetaStore(
    table=meta_table,
    materials_provider=kms_cmp
)

# Create a Most Recent Provider using the MetaStore
#    Sets the TTL (in seconds) and cache size (# entries)
most_recent_cmp = MostRecentProvider(
    provider_store=meta_store,
    material_name=material_name,
    version_ttl=60.0,
    cache_size=1000
)
```

工作原理

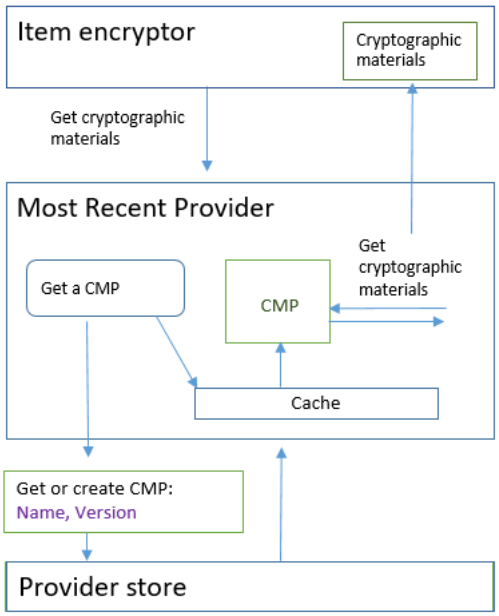
最新提供商 CMPs 来自提供商商店。然后，它使用 CMP 生成由它返回到项目加密程序的加密材料。

关于最新提供程序

最新提供程序从提供程序存储中获得加密材料提供程序（CMP）。然后，它使用 CMP 生成由它返回的加密材料。每个最新提供商都与一个提供商商店相关联，但一个提供商商店可以 CMPs 向多个主机上的多个提供商提供服务。

最新提供程序可与来自任何提供程序存储的任何兼容的 CMP 一起使用。它从 CMP 请求加密或解密材料，并将输出返回给项目加密程序。而不执行任何加密操作。

为了从其提供程序存储请求 CMP，最新提供程序将提供其材料名称以及要使用的现有 CMP 的版本。对于加密材料，最新提供程序始终请求最高（“最新”）版本。对于解密材料，它请求用于创建加密材料的 CMP 的版本，如下图所示。



最新提供程序将提供程序存储返回的 CMPs 版本保存在内存中的本地 “最近最少使用” (LRU) 缓存中。缓存使最新提供商能够获取所需的内容 CMPs ，而无需为每件商品调用提供商商店。您可以按需清除该缓存。

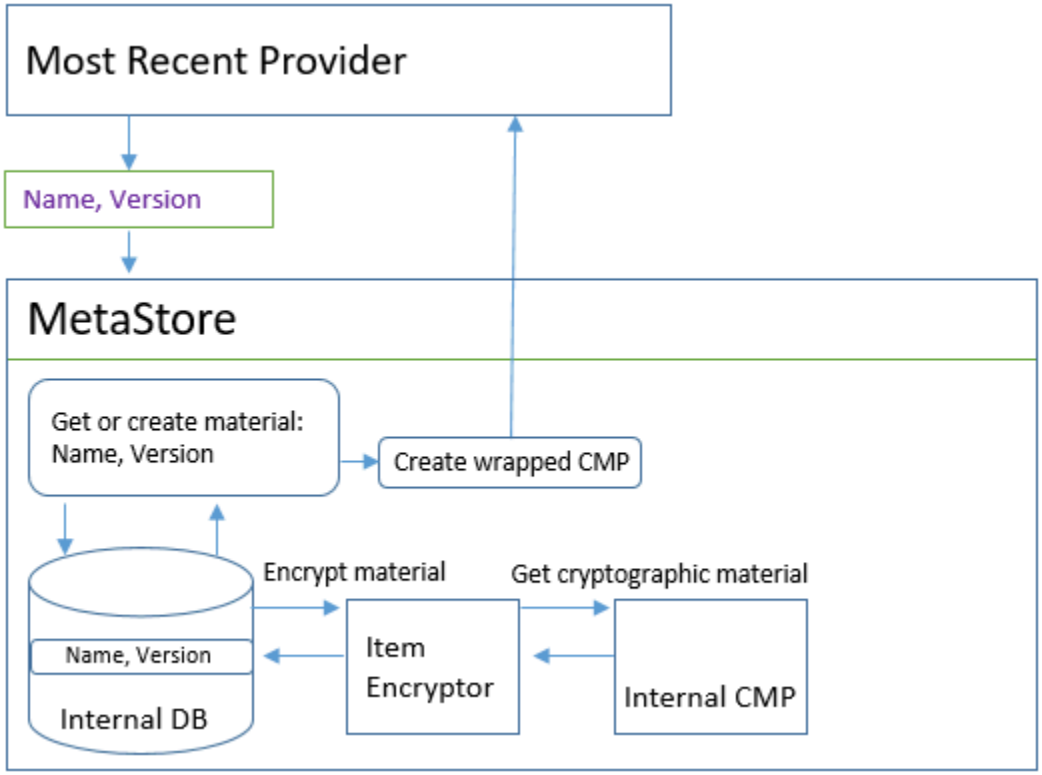
最新提供程序使用可配置的time-to-live值，您可以根据应用程序的特性进行调整。

关于 MetaStore

您可以将最新提供程序与任何提供程序存储结合使用，包括兼容的自定义提供程序存储。DynamoDB 加密客户端包括 MetaStore一个安全实现，您可以对其进行配置和自定义。

A MetaStore是一个提供商存储，用于创建并返回使用 Wr ap CMPs ped CMPs 所需的包装密钥、解包密钥和签名密钥配置的 Wrapped。对于最新提供商来说，A MetaStore 是一个安全的选项，因为 Wrapped CMPs 总是为每个项目生成唯一的项目加密密钥。只有保护项目加密密钥和签名密钥的包装密钥才会被重用。

下图显示了的组件 MetaStore 及其与最新提供程序的交互方式。



MetaStore 生成 Wrapped CMPs，然后将它们（以加密形式）存储在内部 DynamoDB 表中。分区键是最新提供程序材料的名称；排序键则是其版本号。该表中的材料由内部 DynamoDB 加密客户端保护，包括一个项目加密程序和内部加密材料提供程序（CMP）。

您可以在中使用任何类型的内部 CMP MetaStore，包括直接 KMS 提供程序、包含您提供的加密材料的 Wrapped CMP 或兼容的自定义 CMP。如果您的内部 CMP MetaStore 是直接 KMS 提供商，则您的可重复使用的封装和签名密钥将受到 AWS KMS keyin AWS Key Management Service(AWS KMS)的保护。 AWS KMS 每次向其内部表添加新的 CMP 版本或从其内部表中获取 CMP 版本时，都会 MetaStore 调用。

设置一个 time-to-live值

您可以为创建的每个最新提供程序设置一个 time-to-live (TTL) 值。通常情况下，请在您的应用程序中使用实用的最低 TTL 值。

最新提供程序的 `CachingMostRecentProvider` 符号中的 TTL 值的使用已更改。

> **Note**
>
> 最新提供程序的 `MostRecentProvider` 符号在受支持的较早版本的 DynamoDB 加密客户端中已被弃用，并已从 2.0.0 版本中移除。它被 `CachingMostRecentProvider` 符号所取代。建议您尽快更新代码。有关更多信息，请参阅 最新提供程序的更新。

`CachingMostRecentProvider`

`CachingMostRecentProvider` 以两种不同的方式使用 TTL 值。

- TTL 决定了最新提供程序在提供程序存储中检查新版本的 CMP 的频率。如果有新版本可用，最新提供程序将会替换其 CMP 并刷新其加密材料。否则，它将继续使用它的当前 CMP 和加密材料。
- TTL 决定了可以在缓存 CMPs 中使用多长时间。在使用缓存的 CMP 进行加密之前，最新提供程序会评估其在缓存中存在的时间。如果 CMP 缓存时间超过 TTL，则 CMP 将从缓存中被驱逐，最新提供程序将从其提供程序存储中获取最新版本的新 CMP。

`MostRecentProvider`

在 `MostRecentProvider` 中，TTL 决定了最新提供程序在提供程序存储中检查新版本的 CMP 的频率。如果有新版本可用，最新提供程序将会替换其 CMP 并刷新其加密材料。否则，它将继续使用它的当前 CMP 和加密材料。

TTL 并不能确定新的 CMP 版本的创建频率。您可以通过轮换加密材料来创建新的 CMP 版本。

理想的 TTL 值将因应用程序及其延迟和可用性目标而异。低 TTL 可缩短加密材料在内存中的存储时间，从而改善安全状况。而且，TTL 低时，会更频繁地刷新关键信息。例如，如果您的内部 CMP 是 Direct KMS 提供程序，它会更频繁地验证调用方是否仍有权使用 AWS KMS key。

但是，如果 TTL 过短，频繁调用提供程序存储可能会增加您的成本，并导致您的提供程序存储限制来自您的应用程序和共享您的服务账户的其他应用程序的请求。通过将 TTL 与轮换加密材料的速度进行协调，可能也会让您受益。

测试期间，在不同工作负载下更改 TTL 和缓存大小，直到找到适合您的应用程序以及您的安全和性能标准的配置。

轮换加密材料

当最新提供程序需要加密材料时，它始终使用其所知道的最新版本的 CMP。它检查新版本的频率由您在配置最新提供程序时设置的 time-to-live(TTL) 值决定。

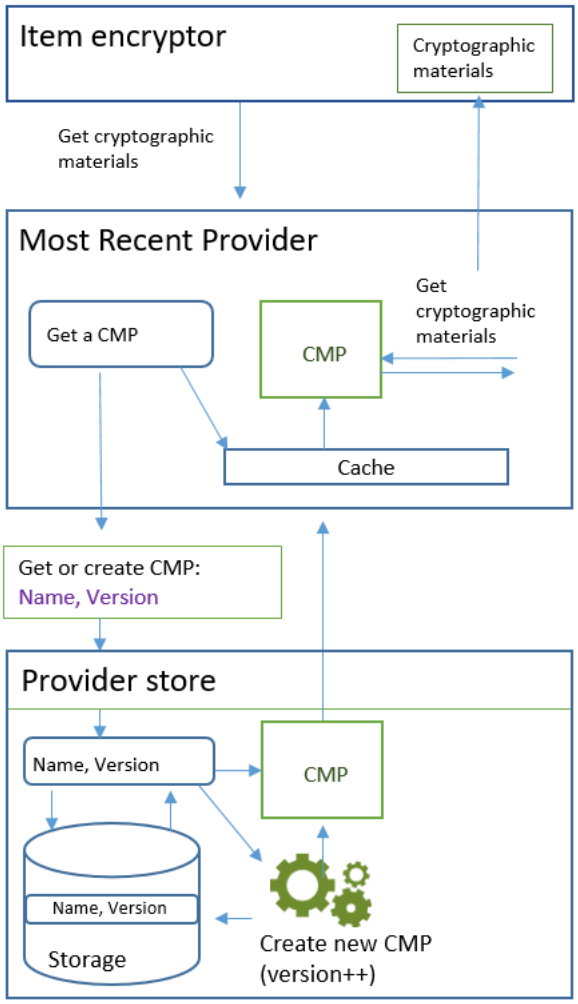
当 TTL 到期时，最新提供程序会在提供程序存储中检查新版本的 CMP。如果有可用版本，则最新提供程序会获取它并替换其缓存中的 CMP。它将使用此 CMP 及其加密材料，直到发现提供程序存储有更新的版本。

要让提供程序存储为最新提供程序创建新版本的 CMP，请使用最新提供程序的材料名称调用提供程序存储的“创建新提供程序”操作。提供程序存储将创建一个新 CMP 并在其内部存储中以更高的版本号保存加密复本。（它还将返回 CMP，但您可以丢弃它。）因此，下次最新提供程序向提供程序商店查询其最大版本号时 CMPs，它会获得新的更大版本号，并在随后向存储请求时使用该版本号来查看是否创建了新版本的 CMP。

您可以基于时间、已处理的项目或属性数或者对您的应用程序有意义的其他指标计划您的“创建新提供程序”调用。

获取加密材料

最新提供程序使用以下过程（如图所示）来获取它返回到项目加密程序的加密材料。输出取决于提供程序存储返回的 CMP 的类型。最新提供程序可以使用任何兼容的提供程序存储，包括 DynamoDB 加密客户端中包含的。 MetaStore



使用CachingMostRecentProvider符号创建最新提供程序时，需要指定提供程序存储区、最新提供程序的名称和 time-to-live(TTL) 值。您也可以选择指定缓存大小，该大小决定缓存中可以存在的最大加密材料数量。

当项目加密程序向最新提供程序请求加密材料时，最新提供程序首先会在其缓存中搜索其 CMP 的最新版本。

- 如果它在缓存中找到了最新版本的 CMP 且 CMP 没有超出 TTL 值，则最新提供程序将使用 CMP 来生成加密材料。然后，它将加密材料返回到项目加密程序。此操作不需要调用提供程序存储。
- 如果最新版本的 CMP 不在其缓存中，或者如果它在缓存中但已超出其 TTL 值，则最新提供程序将从其提供程序存储请求 CMP。该请求包含最新提供程序材料名称以及它知道的最高版本号。
  1. 提供程序存储从其持久性存储返回 CMP。如果提供程序存储是 MetaStore，则使用最新提供程序材料名称作为分区键，使用版本号作为排序键，从其内部 DynamoDB 表中获取加密的 Wrapped CMP。 MetaStore 使用其内部项目加密器和内部 CMP 来解密 Wrapped CMP。然后，它将明文

CMP 返回到最新提供程序。如果内部 CMP 是 Direct KMS 提供程序，此步骤将包含对 AWS Key Management Service (AWS KMS) 的调用。

2. CMP 将 `amzn-ddb-meta-id` 域添加到实际材料描述。该域的值是 CMP 在其内部表中的材料名称和版本。提供程序存储将 CMP 返回到最新提供程序。
3. 最新提供程序在内存中缓存 CMP。
4. 最新提供程序使用 CMP 生成加密材料。然后，它将加密材料返回到项目加密程序。

获取解密材料

当项目加密程序向最新提供程序请求解密材料时，最新提供程序将使用以下过程来获取并返回这些材料。

1. 最新提供程序向提供程序存储询问用于加密项目的加密材料的版本号。提供程序存储传入来自项目的材料描述属性的实际材料描述。
2. 提供程序存储从实际材料描述中的 `amzn-ddb-meta-id` 域获取加密 CMP 版本号并将其返回到最新提供程序。
3. 最新提供程序在缓存中搜索用于加密和签署项目的 CMP 版本。

- 如果发现缓存中存在匹配版本的 CMP，并且 CMP 未超过 time-to-live (TTL) 值，则最新提供程序会使用 CMP 生成解密材料。然后，它将解密材料返回到项目加密程序。此操作不需要调用提供程序存储或任何其他 CMP。
- 如果匹配版本的 CMP 不在其缓存中，或者如果缓存的 AWS KMS key 已超出其 TTL 值，则最新提供程序将从其提供程序存储请求 CMP。它将在请求中发送其材料名称和加密 CMP 版本号。
  1. 提供程序存储将最新提供程序名称用作分区键并将版本号用作排序键，以便在其持久性存储中搜索 CMP。
     - 如果名称和版本号不在其持久性存储中，提供程序存储将引发异常。如果提供程序存储用于生成 CMP，那么 CMP 应该存储在其持久性存储中，除非它被意外删除。
     - 如果具有匹配的名称和版本号的 CMP 位于提供程序存储的持久性存储中，提供程序存储会将指定 CMP 返回到最新提供程序。

       如果提供商存储是 MetaStore，则它会从其 DynamoDB 表中获取加密的 CMP。然后，它从其内部 CMP 获取加密材料以解密已加密的 CMP，再将 CMP 返回到最新提供程序。如果内部 CMP 是 Direct KMS 提供程序，此步骤将包含对 AWS Key Management Service (AWS KMS) 的调用。
  2. 最新提供程序在内存中缓存 CMP。

3. 最新提供程序使用 CMP 生成解密材料。然后，它将解密材料返回到项目加密程序。

最新提供程序的更新

最新提供程序的符号已从 `MostRecentProvider` 更改为 `CachingMostRecentProvider`。

> **Note**
>
> `MostRecentProvider` 符号代表最新提供程序，在适用于 Java 的 DynamoDB 加密客户端 1.15 版本和适用于 Python 的 DynamoDB 加密客户端 1.3 版本中已被弃用，并已从两种语言实现的 DynamoDB 加密客户端 2.0.0 版本中移除。可改用 `CachingMostRecentProvider`。

`CachingMostRecentProvider` 实现了以下更改：

- 当加密材料在内存中的时间超过配置的 time-to-live (TTL) 值时，会`CachingMostRecentProvider`定期将其从内存中删除。

  `MostRecentProvider` 可能会在进程的整个生命周期内将加密材料存储在内存中。因此，最新提供程序可能不知道授权更改。它可能会在调用方使用加密密钥的权限被撤消后使用它们。

  如果您无法更新到此新版本，则可以通过定期在缓存上调用 `clear()` 方法来获得类似的效果。此方法将手动刷新缓存内容，并要求最新提供程序请求新的 CMP 和新的加密材料。
- `CachingMostRecentProvider` 还包括缓存大小设置，该设置可让您更好地控制缓存。

要更新到 `CachingMostRecentProvider`，您必须更改代码中的符号名称。在所有其他方面，`CachingMostRecentProvider` 完全向后兼容 `MostRecentProvider`。您无需重新加密任何表项目。

但是，`CachingMostRecentProvider` 会生成更多对底层密钥基础设施的调用。它在每个 time-to-live (TTL) 间隔中至少调用一次提供商存储区。具有大量活动状态 CMPs（由于频繁轮换）的应用程序或具有大型队列的应用程序最有可能对这种变化很敏感。

在发布更新后的代码之前，请对其进行全面测试，确保更频繁的调用不会损害您的应用程序或导致提供商所依赖的服务（例如 AWS Key Management Service (AWS KMS) 或 Amazon DynamoDB）的限制。要缓解任何性能问题，请`CachingMostRecentProvider`根据您观察到 time-to-live的性能特征调整缓存大小和缓存的大小。有关指南，请参阅设置一个 time-to-live值。

静态材料提供程序

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

静态材料提供商（Static CMP）是一个非常简单的加密材料提供商（CMP），用于测试、 proof-of-concept演示和传统兼容性。

要使用静态 CMP 加密表项目，请提供高级加密标准 (AES) 对称加密和签名密钥或密钥对。必须提供相同的密钥才能解密加密的项目。静态 CMP 不会执行任何加密操作。相反，它会将提供的加密密钥原封不动地传递给项目加密程序。项目加密程序将直接使用此加密密钥加密项目。然后，它将直接使用签名密钥为项目签名。

由于静态 CMP 不会生成任何唯一加密材料，因此将使用同一加密密钥加密且通过同一签名密钥签名您处理的所有表项目。当使用同一密钥加密众多项目中的属性值或使用同一密钥或密钥对为所有项目签名时，将面临超出密钥加密限制的风险。

> **Note**
>
> Java 库中的非对称静态提供程序不是一种静态提供程序。它仅提供已包装的 CMP 的替代构造函数。它对生产使用是安全的，但应尽可能直接使用已包装的 CMP。

静态 CMP 是 DynamoDB 加密客户端支持的几个加密材料提供程序 (CMPs) 之一。有关另一个的信息 CMPs，请参阅加密材料提供程序。

有关示例代码，请参阅：

- Java：SymmetricEncryptedItem

主题

- 使用方法
- 工作原理

使用方法

要创建静态提供程序，请提供加密密钥或密钥对和签名密钥或密钥对。需要提供密钥材料才能加密和解密表项目。

Java

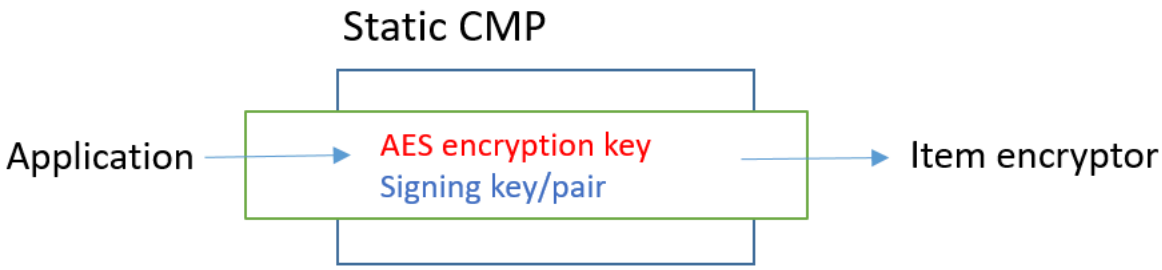
```
// To encrypt
SecretKey cek = ...;        // Encryption key
SecretKey macKey =  ...;    // Signing key
EncryptionMaterialsProvider provider = new SymmetricStaticProvider(cek, macKey);

// To decrypt
SecretKey cek = ...;        // Encryption key
SecretKey macKey =  ...;    // Verification key
EncryptionMaterialsProvider provider = new SymmetricStaticProvider(cek, macKey);
```

Python

```
# You can provide encryption materials, decryption materials, or both
encrypt_keys = EncryptionMaterials(
    encryption_key = ...,
    signing_key = ...
)

decrypt_keys = DecryptionMaterials(
    decryption_key = ...,
    verification_key = ...
)

static_cmp = StaticCryptographicMaterialsProvider(
    encryption_materials=encrypt_keys
    decryption_materials=decrypt_keys
)
```

工作原理

静态提供程序将提供的加密和签名密钥传递到项目加密程序，而后项目加密程序直接使用这些密钥为表项目加密和签名。除非为每个项目提供了不同的密钥，否则对所有项目使用相同的密钥。



获取加密材料

本部分详述了静态材料提供程序（静态 CMP）在收到加密材料请求时的输入、输出和处理。

输入（来自应用程序）

- 加密密钥 – 这必须是对称密钥（如高级加密标准（AES）密钥）。
- 签名密钥 – 这可以是对称密钥或非对称密钥对。

输入（来自项目加密程序）

- DynamoDB 加密上下文

输出（至项目加密程序）

- 作为输入传递的加密密钥。
- 作为输入传递的签名密钥。
- 实际材料描述：请求的材料描述（如有），不做更改。

获取解密材料

本部分详述了静态材料提供程序（静态 CMP）在收到解密材料请求时的输入、输出和处理。

尽管它获取加密材料和获取解密材料的方法不同，但此行为是相同的。

输入（来自应用程序）

- 加密密钥 – 这必须是对称密钥（如高级加密标准（AES）密钥）。
- 签名密钥 – 这可以是对称密钥或非对称密钥对。

输入（来自项目加密程序）

- DynamoDB 加密上下文（未使用）

输出（至项目加密程序）

- 作为输入传递的加密密钥。
- 作为输入传递的签名密钥。

Amazon DynamoDB Encryption Client 可用的编程语言

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

Amazon DynamoDB Encryption Client 适用于以下编程语言。特定于语言的库各不相同，但生成的实现是可互操作的。例如，您可以使用 Java 客户端加密（和签署）项目，并使用 Python 客户端解密项目。

有关更多信息，请参阅相应主题。

主题

- 适用于 Java 的 Amazon DynamoDB Encryption Client
- 适用于 Python 的 DynamoDB 加密客户端

适用于 Java 的 Amazon DynamoDB Encryption Client

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本

1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

本主题介绍了如何安装和使用适用于 Java 的 Amazon DynamoDB Encryption Client。有关使用 DynamoDB 加密客户端进行编程的详细信息，请参阅 Java 示例、存储库 GitHub中的 aws-dynamodb-encryption-java示例以及 DynamoDB 加密客户端的 Javadoc。

> **Note**
>
> 版本 1. x。 适用于 Java 的 DynamoDB 加密客户端中的 x 已于 2022 end-of-support 年 7 月开始分阶段生效。请尽快升级到更新的版本。

主题

先决条件

在安装适用于 Java 的 Amazon DynamoDB Encryption Client 之前，请确保满足以下先决条件。

Java 开发环境

您需要使用 Java 8 或更高版本。在 Oracle 网站上，转到 Java SE 下载，然后下载并安装 Java SE Development Kit (JDK)。

如果使用 Oracle JDK，您还必须下载并安装 Java Cryptography Extension (JCE) Unlimited Strength Jurisdiction Policy Files。

适用于 Java 的 AWS SDK

即使您的应用程序未与 DynamoDB 交互，DynamoDB 加密客户端也需要的 DynamoDB 模块。 适用于 Java 的 AWS SDK 可以安装整个开发工具包或仅安装此模块。如果使用的是 Maven，则将 `aws-java-sdk-dynamodb` 添加到 `pom.xml` 文件。

有关安装和配置的更多信息 适用于 Java 的 AWS SDK，请参阅适用于 Java 的 AWS SDK。

安装

您可以通过下列方式安装适用于 Java 的 Amazon DynamoDB Encryption Client。

手动方式

要安装适用于 Java 的 Amazon DynamoDB 加密客户端，请克隆或下载存储库。aws-dynamodb-encryption-java GitHub

使用 Apache Maven

适用于 Java 的 Amazon DynamoDB Encryption Client 通过 Apache Maven 提供，并具有以下依赖项定义。

```
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-dynamodb-encryption-java</artifactId>
  <version>version-number</version>
</dependency>
```

安装完软件开发工具包后，请先查看本指南中的示例代码并打开 D ynamoDB 加密客户端 Jav adoc。GitHub

使用适用于 Java 的 DynamoDB 加密客户端

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

本主题介绍了 Java 中的 DynamoDB 加密客户端的可能在其他编程语言实施中找不到的一些功能。

有关使用 DynamoDB 加密客户端进行编程的详细信息，请参阅 Java 示例、 GitHub上面的示例以及 DynamoDB 加密客户端的 Javadoc。aws-dynamodb-encryption-java repository

主题

物品加密器： AttributeEncryptor 和 Dynamo DBEncryptor

Java 中的 DynamoDB 加密客户端有两个项目加密器：较低级别的 Dynamo 和。DBEncryptor AttributeEncryptor

AttributeEncryptor是一个帮助程序类，可帮助您在 DynamoD B DBMapper 加密客户端DynamoDB Encryptor中 适用于 Java 的 AWS SDK 使用 Dynamo。如果您结合使用 AttributeEncryptor 和 DynamoDBMapper，则当您保存项目时，它会透明对项目进行加密并签名。当您加载项目时，它还会透明地验证和解密您的项目。

配置保存行为

您可以使用 AttributeEncryptor 和 DynamoDBMapper 来添加或替换具有仅已签名（或已加密和签名）的属性的表项目。对于这些任务，我们建议您将其配置为使用 PUT 保存行为，如以下示例所示。否则，您可能无法解密您的数据。

```
DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

如果您使用默认保存行为（仅更新在表项目中建模的属性），则未建模的属性将不会包含在签名中，且不会由表写入更改。因此，在以后读取所有的属性时，签名将无法验证，因为它不包括未建模的属性。

您也可以使用 CLOBBER 保存行为。该行为与 PUT 保存行为相同，只不过它将禁用乐观锁并覆盖表中的项目。

为防止签名错误，如果 AttributeEncryptor 与未配置 CLOBBER 或 PUT 保存行为的 DynamoDBMapper 一起使用，则 DynamoDB 加密客户端会抛出运行时系统异常。

要查看示例中使用的此代码，请参阅使用发电机 DBMapper和中aws-dynamodb-encryption-java存储库中的 AwsKmsEncryptedObject GitHub.java 示例。

Java 中的属性操作

属性操作确定加密并签名的属性值、仅签名的属性值以及忽略的属性值。用于指定属性操作的方法取决于您使用的是DynamoDBMapper和AttributeEncryptor还是较低级别的 Dynam DBEncryptor o。

> **Important**
>
> 使用属性操作对表项进行加密后，在数据模型中添加或删除属性可能会导致签名验证错误，从而使您无法解密数据。有关详细说明，请参阅更改数据模型。

Dynamo 的属性动作 DBMapper

当您使用 DynamoDBMapper 和 AttributeEncryptor 时，使用注释指定属性操作。DynamoDB 加密客户端使用标准 DynamoDB 属性注释，该注释可定义属性类型以确定如何保护属性。默认情况下，除主键以外的所有属性均加密并签名，主键已签名但未加密。

> **Note**
>
> 不要使用 @Dynamo Attribut DBVersion e 注解对属性的值进行加密，尽管你可以（也应该）对它们进行签名。否则，使用其值的情况将产生意外后果。

```
// Attributes are encrypted and signed
@DynamoDBAttribute(attributeName="Description")

// Partition keys are signed but not encrypted
@DynamoDBHashKey(attributeName="Title")

// Sort keys are signed but not encrypted
@DynamoDBRangeKey(attributeName="Author")
```

要指定例外情况，请使用在适用于 Java 的 DynamoDB 加密客户端中定义的加密注释。如果您在类级别指定这些注释，它们将成为该类的默认值。

```
// Sign only
@DoNotEncrypt

// Do nothing; not encrypted or signed
@DoNotTouch
```

例如，这些注释签署但未加密 PublicationYear 属性，而且未加密或签署 ISBN 属性值。

```
// Sign only (override the default)
@DoNotEncrypt
@DynamoDBAttribute(attributeName="PublicationYear")

// Do nothing (override the default)
@DoNotTouch
@DynamoDBAttribute(attributeName="ISBN")
```

Dynamo 的属性动作 DBEncryptor

要在DBEncryptor直接使用 Dynamo 时指定属性操作，请创建一个HashMap对象，其中名称/值对表示属性名称和指定操作。

有效值适用于在 EncryptionFlags 枚举类型中定义的属性操作。您可以结合使用 ENCRYPT 和 SIGN，单独使用 SIGN，或同时忽略。但是，如果您单独使用 ENCRYPT，则 DynamoDB 加密客户端会抛出错误。您无法加密未签名的属性。

```
ENCRYPT
SIGN
```

> **Warning**
>
> 请勿加密主键属性。它们必须保留为明文，以便 DynamoDB 查找项目而无需运行全表扫描。

如果您在加密上下文中指定一个主键，然后为主键属性的属性操作指定 ENCRYPT，则 DynamoDB 加密客户端会抛出异常。

例如，以下 Java 代码创建了一个actions HashMap 对record项目中的所有属性进行加密和签名。例外是分区键和排序键属性（这些属性已签名但未加密）以及 test 属性（该属性未签名或未加密）。

```
final EnumSet<EncryptionFlags> signOnly = EnumSet.of(EncryptionFlags.SIGN);
final EnumSet<EncryptionFlags> encryptAndSign = EnumSet.of(EncryptionFlags.ENCRYPT,
 EncryptionFlags.SIGN);
final Map<String, Set<EncryptionFlags>> actions = new HashMap<>();

for (final String attributeName : record.keySet()) {
  switch (attributeName) {
    case partitionKeyName: // no break; falls through to next case
```

```
    case sortKeyName:
      // Partition and sort keys must not be encrypted, but should be signed
      actions.put(attributeName, signOnly);
      break;
    case "test":
      // Don't encrypt or sign
      break;
    default:
      // Encrypt and sign everything else
      actions.put(attributeName, encryptAndSign);
      break;
  }
}
```

然后，当您调用 DynamoDBEncryptor 的 encryptRecord 方法时，将映射指定为 attributeFlags 参数的值。例如，这个对 encryptRecord 的调用使用 actions 映射。

```
// Encrypt the plaintext record
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

覆盖表名称

在 DynamoDB 加密客户端中，DynamoDB 表的名称是传递到加密和解密方法的 DynamoDB 加密上下文的元素。对表项目进行加密或签名时，DynamoDB 加密上下文（包括表名称）以加密方式绑定到加密文字。如果传递给解密方法的 DynamoDB 加密上下文与传递给加密方法的 DynamoDB 加密上下文不匹配，则解密操作将失败。

有时，表的名称会发生变化，例如备份表或执行point-in-time 恢复时。解密或验证这些项目的签名时，必须传递用于对项目进行加密和签名的相同 DynamoDB 加密上下文，包括原始表名称。不需要当前表名称。

使用 DynamoDBEncryptor 时，您将手动汇编 DynamoDB 加密上下文。但是，如果使用 DynamoDBMapper，AttributeEncryptor 会为您创建 DynamoDB 加密上下文，包括当前表名称。要告知 AttributeEncryptor 使用其他表名称创建加密上下文，请使用 EncryptionContextOverrideOperator。

例如，以下代码创建加密材料提供程序 (CMP) 和 DynamoDBEncryptor 的实例。然后，它调用 DynamoDBEncryptor 的 setEncryptionContextOverrideOperator 方法。它使用 overrideEncryptionContextTableName 运算符，该运算符将覆盖一个表名称。通过这种方式配置它后，AttributeEncryptor 会创建一个 DynamoDB 加密

上下文，其中包含 newTableName 以代替 oldTableName。有关完整的示例，请参阅 EncryptionContextOverridesWithDynamoDBMapper.java。

```
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);

encryptor.setEncryptionContextOverrideOperator(EncryptionContextOperators.overrideEncryptionCon
                oldTableName, newTableName));
```

当您调用 DynamoDBMapper 的加载方法（该方法解密并验证项目）时，您指定原始表名称。

```
mapper.load(itemClass, DynamoDBMapperConfig.builder()

 .withTableNameOverride(DynamoDBMapperConfig.TableNameOverride.withTableNameReplacement(oldTabl
                .build());
```

您还可以使用 overrideEncryptionContextTableNameUsingMap 运算符，该运算符将覆盖多个表名称。

表名称覆盖运算符通常在解密数据和验证签名时使用。但是，您可以使用它们在加密和签名时将 DynamoDB 加密上下文中的表名称设置为其他值。

如果使用 DynamoDBEncryptor，请不要使用表名称覆盖运算符。而是使用原始表名称创建一个加密上下文，并将其提交给解密方法。

适用于 Java 的 DynamoDB 加密客户端的示例代码

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

以下示例为您演示如何使用适用于 Java 的 DynamoDB 加密客户端来保护应用程序中的 DynamoDB 表项目。你可以在上aws-dynamodb-encryption-java存储库的示例目录中找到更多示例（并贡献自己的示例）GitHub。

主题

- 使用发电机 DBEncryptor
- 使用发电机 DBMapper

使用发电机 DBEncryptor

此示例说明如何将较低级别的 Dynamo DBEncryptor 与 Di rect KMS 提供程序配合使用。直接 KMS 提供商在您指定的 AWS KMS keyin AWS Key Management Service (AWS KMS) 下生成并保护其加密材料。

您可以将任何兼容的加密材料提供程序 (CMP) 与一起使用`DynamoDBEncryptor`，也可以将直接 KMS 提供程序与`DynamoDBMapper`和AttributeEncryptor一起使用。

查看完整的代码示例：AwsKmsEncryptedItem.java

步骤 1：创建 Direct KMS 提供程序

创建指定区域的 AWS KMS 客户端实例。然后，使用客户端实例借助您的首选 AWS KMS key创建 Direct KMS 提供程序的实例。

此示例使用 Amazon 资源名称 (ARN) 来标识 AWS KMS key，但您可以使用任何有效的密钥标识符。

```
final String keyArn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
final String region = "us-west-2";

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

步骤 2：创建项目

此示例定义了`record` HashMap 表示示例表项的。

```
final String partitionKeyName = "partition_attribute";
final String sortKeyName = "sort_attribute";

final Map<String, AttributeValue> record = new HashMap<>();
record.put(partitionKeyName, new AttributeValue().withS("value1"));
record.put(sortKeyName, new AttributeValue().withN("55"));
record.put("example", new AttributeValue().withS("data"));
record.put("numbers", new AttributeValue().withN("99"));
```

```
record.put("binary", new AttributeValue().withB(ByteBuffer.wrap(new byte[]{0x00,
 0x01, 0x02})));
record.put("test", new AttributeValue().withS("test-value"));
```

步骤 3：创建发电机 DBEncryptor

使用 Direct KMS 提供程序创建 DynamoDBEncryptor 的实例。

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp);
```

步骤 4：创建 DynamoDB 加密上下文

DynamoDB 加密上下文包含有关表结构以及其如何加密和签名的信息。如果使用的是 DynamoDBMapper，则 AttributeEncryptor 会为您创建加密上下文。

```
final String tableName = "testTable";

final EncryptionContext encryptionContext = new EncryptionContext.Builder()
    .withTableName(tableName)
    .withHashKeyName(partitionKeyName)
    .withRangeKeyName(sortKeyName)
    .build();
```

步骤 5：创建属性操作对象

属性操作确定已加密并签名的项目属性、仅签名的属性以及未加密或签名的属性。

在 Java 中，要指定属性操作，需要创建属性名称和EncryptionFlags值对。 HashMap

例如，以下 Java 代码创建了一个actions HashMap 对record项目中的所有属性进行加密和签名，但分区键和排序密钥属性（已签名但未加密）以及未签名或加密的test属性除外。

```
final EnumSet<EncryptionFlags> signOnly = EnumSet.of(EncryptionFlags.SIGN);
final EnumSet<EncryptionFlags> encryptAndSign = EnumSet.of(EncryptionFlags.ENCRYPT,
 EncryptionFlags.SIGN);
final Map<String, Set<EncryptionFlags>> actions = new HashMap<>();

for (final String attributeName : record.keySet()) {
  switch (attributeName) {
    case partitionKeyName: // fall through to the next case
    case sortKeyName:
      // Partition and sort keys must not be encrypted, but should be signed
      actions.put(attributeName, signOnly);
```

```
      break;
    case "test":
      // Neither encrypted nor signed
      break;
    default:
      // Encrypt and sign all other attributes
      actions.put(attributeName, encryptAndSign);
      break;
  }
}
```

步骤 6：加密并签名项目

要加密并签名项目，请对 `encryptRecord` 的实例调用 `DynamoDBEncryptor` 方法。指定表项目 (`record`)、属性操作 (`actions`) 和加密上下文 (`encryptionContext`)。

```
final Map<String, AttributeValue> encrypted_record = encryptor.encryptRecord(record,
 actions, encryptionContext);
```

步骤 7：将项目放入 DynamoDB 表中

最后，将已加密且签名的项目放入 DynamoDB 表中。

```
final AmazonDynamoDB ddb = AmazonDynamoDBClientBuilder.defaultClient();
ddb.putItem(tableName, encrypted_record);
```

使用发电机 DBMapper

以下示例为您演示如何结合使用 DynamoDB Mapper 帮助程序类与 Direct KMS 提供程序。Direct KMS 提供程序借助您指定的 AWS Key Management Service（AWS KMS）中的 AWS KMS key 生成并保护其加密材料。

您可以结合使用任何兼容的加密材料提供程序 (CMP) 与 `DynamoDBMapper`，也可以结合使用 Direct KMS 提供程序与低级别 `DynamoDBEncryptor`。

查看完整的代码示例：AwsKmsEncryptedObject.java

步骤 1：创建 Direct KMS 提供程序

创建指定区域的 AWS KMS 客户端实例。然后，使用客户端实例借助您的首选 AWS KMS key创建 Direct KMS 提供程序的实例。

此示例使用 Amazon 资源名称 (ARN) 来标识 AWS KMS key，但您可以使用任何有效的密钥标识符。

```
final String keyArn = "arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab";
final String region = "us-west-2";

final AWSKMS kms = AWSKMSClientBuilder.standard().withRegion(region).build();
final DirectKmsMaterialProvider cmp = new DirectKmsMaterialProvider(kms, keyArn);
```

第 2 步：创建 DynamoDB 加密器和 Dynamo DBMapper

使用您在上一步中创建的 Direct KMS 提供程序创建 DynamoDB Encryptor 的实例。您需要实例化低级别 DynamoDB Encryptor 才能使用 DynamoDB Mapper。

接着，创建 DynamoDB 数据库的实例和映射器配置，然后使用它们创建 DynamoDB Mapper 的实例。

> **Important**
>
> 当使用 DynamoDBMapper 添加或编辑已签名（或已加密并签名）项目时，将其配置为使用保存行为（如包含所有属性的 PUT），如以下示例所示。否则，您可能无法解密您的数据。

```
final DynamoDBEncryptor encryptor = DynamoDBEncryptor.getInstance(cmp)
final AmazonDynamoDB ddb =
 AmazonDynamoDBClientBuilder.standard().withRegion(region).build();

DynamoDBMapperConfig mapperConfig =
 DynamoDBMapperConfig.builder().withSaveBehavior(SaveBehavior.PUT).build();
DynamoDBMapper mapper = new DynamoDBMapper(ddb, mapperConfig, new
 AttributeEncryptor(encryptor));
```

步骤 3：定义您的 DynamoDB 表

接下来，定义您的 DynamoDB 表。使用注释指定属性操作。此示例将创建一个 DynamoDB 表、ExampleTable，以及一个表示表项目的 DataPoJo 类。

在此示例表中，将为主键属性签名，但不进行加密。这适用于使用 @DynamoDBHashKey 进行注释的 partition_attribute 以及使用 @DynamoDBRangeKey 进行注释的 sort_attribute。

将为使用 @DynamoDBAttribute 进行注释的属性（如 some numbers）加密并签名。使用 @DoNotEncrypt（仅签名）或 DynamoDB 加密客户端定义的 @DoNotTouch（不进行加密或签名）加密注释的属性则例外。例如，由于 leave me 属性具有 @DoNotTouch 注释，因此不会为其加密或签名。

```
@DynamoDBTable(tableName = "ExampleTable")
public static final class DataPoJo {
  private String partitionAttribute;
  private int sortAttribute;
  private String example;
  private long someNumbers;
  private byte[] someBinary;
  private String leaveMe;

  @DynamoDBHashKey(attributeName = "partition_attribute")
  public String getPartitionAttribute() {
    return partitionAttribute;
  }

  public void setPartitionAttribute(String partitionAttribute) {
    this.partitionAttribute = partitionAttribute;
  }

  @DynamoDBRangeKey(attributeName = "sort_attribute")
  public int getSortAttribute() {
    return sortAttribute;
  }

  public void setSortAttribute(int sortAttribute) {
    this.sortAttribute = sortAttribute;
  }

  @DynamoDBAttribute(attributeName = "example")
  public String getExample() {
    return example;
  }

  public void setExample(String example) {
    this.example = example;
  }

  @DynamoDBAttribute(attributeName = "some numbers")
  public long getSomeNumbers() {
```

```
    return someNumbers;
  }

  public void setSomeNumbers(long someNumbers) {
    this.someNumbers = someNumbers;
  }

  @DynamoDBAttribute(attributeName = "and some binary")
  public byte[] getSomeBinary() {
    return someBinary;
  }

  public void setSomeBinary(byte[] someBinary) {
    this.someBinary = someBinary;
  }

  @DynamoDBAttribute(attributeName = "leave me")
  @DoNotTouch
  public String getLeaveMe() {
    return leaveMe;
  }

  public void setLeaveMe(String leaveMe) {
    this.leaveMe = leaveMe;
  }

  @Override
  public String toString() {
    return "DataPoJo [partitionAttribute=" + partitionAttribute + ", sortAttribute="
        + sortAttribute + ", example=" + example + ", someNumbers=" + someNumbers
        + ", someBinary=" + Arrays.toString(someBinary) + ", leaveMe=" + leaveMe +
 "]";
  }
}
```

步骤 4：加密并保存表项目

现在，当您创建一个表项目并使用 DynamoDB Mapper 保存它时，会在将此项目添加到表之前自动对其进行加密和签名。

此示例定义一个名为 `record` 的表项目。在将此表项目保存到表中之前，将基于 `DataPoJo` 类中的注释为其属性加密和签名。在此示例中，将为 `PartitionAttribute`、`SortAttribute` 和

LeaveMe 之外的所有属性加密和签名。仅为 PartitionAttribute 和 SortAttributes 进行签名。不会为 LeaveMe 属性加密或签名。

要为 record 项目加密并签名，然后将其添加到 ExampleTable，请调用 DynamoDBMapper 类的 save 方法。由于您的 DynamoDB Mapper 配置为使用 PUT 保存行为，因此项目将替换具有相同主键的任何项目，而不是更新这些项目。这将确保签名匹配，并且您可以在从表中获取项目时为其解密。

```
DataPoJo record = new DataPoJo();
record.setPartitionAttribute("is this");
record.setSortAttribute(55);
record.setExample("data");
record.setSomeNumbers(99);
record.setSomeBinary(new byte[]{0x00, 0x01, 0x02});
record.setLeaveMe("alone");

mapper.save(record);
```

适用于 Python 的 DynamoDB 加密客户端

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

本主题介绍了如何安装和使用适用于 Python 的 DynamoDB 加密客户端。您可以在上的aws-dynamodb-encryption-python存储库中找到代码 GitHub，包括完整且经过测试的示例代码，以帮助您入门。

> **Note**
>
> 版本 1. x。 x 和 2。 x。 适用于 Python 的 DynamoDB 加密客户端的 x 已于 2022 end-of-support 年 7 月开始分阶段生效。请尽快升级到更新的版本。

主题

- 先决条件
- 安装
- 使用适用于 Python 的 DynamoDB 加密客户端
- 适用于 Python 的 DynamoDB 加密客户端的示例代码

先决条件

在安装适用于 Python 的 Amazon DynamoDB Encryption Client 之前，请确保满足以下先决条件。

支持的 Python 版本

对于 Python 版本 3.3.0 及更高版本，亚马逊 DynamoDB 加密客户端需要 Python 3.8 或更高版本。要下载 Python，请参阅 Python 下载。

适用于 Python 的 Amazon DynamoDB Encryption Client 的早期版本支持 Python 2.7 和 Python 3.4 及更高版本，但我们建议您使用 DynamoDB 加密客户端的最新版本。

适用于 Python 的 pip 安装工具

Python 3.6 及更高版本包括 pip，但您可能需要对其进行升级。有关升级或安装 pip 的更多信息，请参阅 pip 文档中的安装。

安装

可以使用 pip 安装适用于 Python 的 Amazon DynamoDB Encryption Client，如以下示例中所示。

安装最新版本

```
pip install dynamodb-encryption-sdk
```

有关使用 pip 安装和升级程序包的详细信息，请参阅安装程序包。

DynamoDB 加密客户端要求在所有平台上使用加密库。所有 pip 版本在 Windows 上安装和构建加密库。pip 8.1 和更高版本在 Linux 上安装和构建加密库。如果使用早期版本的 pip 并且 Linux 环境没有构建加密库所需的工具，则需要安装这些工具。有关更多信息，请参阅在 Linux 上构建加密。

您可以从存储库中获取 DynamoDB 加密客户端的最新aws-dynamodb-encryption-python开发版本。GitHub

安装 DynamoDB 加密客户端后，先从在本指南中查找示例 Python 代码。

使用适用于 Python 的 DynamoDB 加密客户端

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

本主题介绍了适用于 Python 的 DynamoDB 加密客户端的可能在其他编程语言实施中找不到的一些功能。这些功能旨在更轻松地以最安全的方式使用 DynamoDB 加密客户端。除非您有不寻常的使用案例，否则我们建议您使用这些功能。

有关使用 DynamoDB 加密客户端进行编程的详细信息，请参阅本指南中的 Python 示例、存储库 GitHub中的 aws-dynamodb-encryption-python示例以及 DynamoDB 加密客户端的 Python 文档。

主题

- 客户端帮助程序类
- TableInfo 班级
- Python 中的属性操作

客户端帮助程序类

适用于 Python 的 DynamoDB 加密客户端包括多个对 DynamoDB 的 Boto 3 类进行镜像的客户端帮助程序类。这些帮助程序类旨在更轻松地向您的现有 DynamoDB 应用程序添加加密和签名并且避免最常见问题，如下所示：

- 通过向对象添加主密钥的覆盖操作，或者在您的AttributeActions`AttributeActions`对象明确要求客户端加密主密钥时抛出异常，防止您对项目中的主密钥进行加密。如果您的 `AttributeActions` 对象中的默认操作为 `DO_NOTHING`，则客户端帮助程序类会对主键使用该操作。否则，它们使用 `SIGN_ONLY`。
- 创建TableInfo 对象并根据对 Dynamo DB 的调用填充 DynamoDB 加密上下文。这有助于确保您的 DynamoDB 加密上下文准确且客户端可以标识主键。
- 支持方法（如 `put_item` 和 `get_item`），这些方法在您在 DynamoDB 表中写入或读取时会以透明方式加密和解密表项目。仅不支持 `update_item` 方法。

您可以使用客户端帮助程序类而不是直接与较低级别的项目加密程序交互。除非您需要在项目加密程序中设置高级选项，否则使用这些类。

客户端帮助程序类包括：

- EncryptedTable适用于使用 DynamoDB 中的表资源一次处理一张表的应用程序。
- EncryptedResource适用于使用 DynamoDB 中的服务资源类进行批处理的应用程序。
- EncryptedClient适用于在 DynamoDB 中使用较低级别客户端的应用程序。

要使用客户端帮助程序类，调用者必须具有在目标表上调用 Dynam DescribeTableoDB 操作的权限。

TableInfo 班级

该TableInfo类是一个代表一个 DynamoDB 表的辅助类，其中包含用于其主键和二级索引的字段。它有助于您获取有关表的准确的实时信息。

如果您使用的是客户端帮助程序类，它会为您创建并使用 `TableInfo` 对象。否则，您可以明确创建一个。有关示例，请参阅使用项目加密程序。

当您在`TableInfo`对象上调用该`refresh_indexed_attributes`方法时，它会通过调用 DynamoDB DescribeTable操作来填充该对象的属性值。查询表要比硬编码索引名称更加可靠。`TableInfo` 类还包括 `encryption_context_values` 属性，该属性提供了 DynamoDB 加密上下文所需的值。

要使用该`refresh_indexed_attributes`方法，调用者必须具有在目标表上调用 Dynam DescribeTableoDB 操作的权限。

Python 中的属性操作

属性操作告知项目加密程序将对项目的每个属性执行哪些操作。要在 Python 中指定属性操作，请创建具有默认操作和针对特定属性的任何例外的 `AttributeActions` 对象。有效值将在 `CryptoAction` 枚举类型中定义。

> **Important**
>
> 使用属性操作对表项进行加密后，在数据模型中添加或删除属性可能会导致签名验证错误，从而使您无法解密数据。有关详细说明，请参阅更改数据模型。

```
DO_NOTHING = 0
```

```
SIGN_ONLY = 1
ENCRYPT_AND_SIGN = 2
```

例如，此 AttributeActions 对象建立 ENCRYPT_AND_SIGN 作为所有属性的默认值，并且指定 ISBN 和 PublicationYear 属性的例外。

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
        'ISBN': CryptoAction.DO_NOTHING,
        'PublicationYear': CryptoAction.SIGN_ONLY
    }
)
```

如果您使用的是客户端帮助程序类，则无需指定主键属性的属性操作。该客户端帮助程序类阻止您加密主键。

如果您未使用客户端帮助程序类且默认操作为 ENCRYPT_AND_SIGN，则必须为主键指定操作。对主键建议的操作为 SIGN_ONLY。要轻松实现此操作，请使用 set_index_keys 方法，该方法对主键使用 SIGN_ONLY，或者使用 DO_NOTHING，这是默认操作。

> **Warning**
>
> 请勿加密主键属性。它们必须保留为明文，以便 DynamoDB 查找项目而无需运行全表扫描。

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
)
actions.set_index_keys(*table_info.protected_index_keys())
```

适用于 Python 的 DynamoDB 加密客户端的示例代码

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

以下示例为您演示如何使用适用于 Python 的 DynamoDB 加密客户端来保护应用程序中的 DynamoDB 数据。你可以在上aws-dynamodb-encryption-python存储库的示例目录中找到更多示例（并贡献自己的示例）GitHub。

主题

- 使用 EncryptedTable 客户端帮助器类
- 使用项目加密程序

使用 EncryptedTable 客户端帮助器类

以下示例为您演示如何结合使用 Direct KMS 提供程序和此EncryptedTable客户端帮助程序类。此示例使用相同的加密材料提供程序，如使用项目加密程序示例所示。但是，它使用的是 EncryptedTable 类，而不是与低级别项目加密程序直接交互。

通过比较这些示例，您可以看到客户端帮助程序类为您执行的工作。这包括创建 DynamoDB 加密上下文和确保始终为主键属性签名，但绝不加密。要创建加密上下文并发现主密钥，客户端帮助程序类会调用 DynamoDB 操作DescribeTable。要运行此代码，您必须具有调用此操作的权限。

请参阅完整的代码示例：aws_kms_encrypted_table.py

步骤 1：创建表

首先使用表名称创建标准 DynamoDB 表的实例。

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

步骤 2：创建加密材料提供程序

创建您选择的加密材料提供程序 (CMP) 的实例。

此示例使用 Direct KMS 提供程序，但您可以使用任何兼容的 CMP。要创建 Direct KMS 提供程序，请指定 AWS KMS key。此示例使用的 Amazon 资源名称 (ARN) AWS KMS key，但您可以使用任何有效的密钥标识符。

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

步骤 3：创建属性操作对象

属性操作告知项目加密程序将对项目的每个属性执行哪些操作。此示例中的 AttributeActions 对象加密并签名所有项目，除 test 属性以外，后者已被忽略。

请勿在使用客户端帮助程序类时为主键属性指定属性操作。EncryptedTable 类会签名但绝不加密主键属性。

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
)
```

步骤 4：创建加密表

使用标准表、Direct KMS 提供程序和属性操作创建加密表。此步骤将完成配置。

```
encrypted_table = EncryptedTable(
    table=table,
    materials_provider=kms_cmp,
    attribute_actions=actions
)
```

步骤 5：将明文项目放入表中

在对 encrypted_table 调用 put_item 方法时，您的表项目会以透明方式加密、签名并添加到您的 DynamoDB 表。

首先，定义表项目。

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_attribute': 55
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

然后，请项目放入表中。

```
encrypted_table.put_item(Item=plaintext_item)
```

要从采用加密形式的 DynamoDB 表中获取该项目，请对 `table` 对象调用 `get_item` 方法。要获取已解密的项目，请对 `get_item` 对象调用 `encrypted_table` 方法。

使用项目加密程序

此示例向您展示在加密表项目时如何直接与 DynamoDB 加密客户端中的项目加密程序交互，而不是使用与项目加密程序交互的客户端帮助程序类。

使用此方法时，将手动创建 DynamoDB 加密上下文和配置对象（`CryptoConfig`）。此外，您还会在加密一个调用中的项目并将它们放置在您在单独调用中的 DynamoDB 表中。这允许您自定义 `put_item` 调用并使用 DynamoDB 加密客户端来加密并签名绝不发送 DynamoDB 的结构化数据。

此示例使用 Direct KMS 提供程序，但您可以使用任何兼容的 CMP。

请参阅完整的代码示例：aws_kms_encrypted_item.py

步骤 1：创建表

首先使用表名称创建标准 DynamoDB 表资源的实例。

```
table_name='test-table'
table = boto3.resource('dynamodb').Table(table_name)
```

步骤 2：创建加密材料提供程序

创建您选择的加密材料提供程序 (CMP) 的实例。

此示例使用 Direct KMS 提供程序，但您可以使用任何兼容的 CMP。要创建 Direct KMS 提供程序，请指定 AWS KMS key。此示例使用的 Amazon 资源名称 (ARN) AWS KMS key，但您可以使用任何有效的密钥标识符。

```
kms_key_id='arn:aws:kms:us-
west-2:111122223333:key/1234abcd-12ab-34cd-56ef-1234567890ab'
kms_cmp = AwsKmsCryptographicMaterialsProvider(key_id=kms_key_id)
```

第 3 步：使用 TableInfo 辅助类

要从 DynamoDB 获取有关表的信息，请创建帮助类的实例。TableInfo当您直接使用项目加密程序时，需要创建一个 `TableInfo` 实例并调用其方法：客户端帮助程序类可为您执行此操作。

的refresh_indexed_attributes方法TableInfo使用 DescribeTableDynamoDB 操作来获取有关表的实时、准确的信息。这包括其主键及其本地和全局二级索引。调用方需要具有调用 DescribeTable 的权限。

```
table_info = TableInfo(name=table_name)
table_info.refresh_indexed_attributes(table.meta.client)
```

步骤 4：创建 DynamoDB 加密上下文

DynamoDB 加密上下文包含有关表结构以及其如何加密和签名的信息。此示例明确创建 DynamoDB 密上下文，因为它与项目加密程序交互。客户端帮助程序类为您创建 DynamoDB 加密上下文。

要获取分区键和排序键，可以使用TableInfo辅助类的属性。

```
index_key = {
    'partition_attribute': 'value1',
    'sort_attribute': 55
}

encryption_context = EncryptionContext(
    table_name=table_name,
    partition_key_name=table_info.primary_index.partition,
    sort_key_name=table_info.primary_index.sort,
    attributes=dict_to_ddb(index_key)
)
```

步骤 5：创建属性操作对象

属性操作告知项目加密程序将对项目的每个属性执行哪些操作。此示例中的 AttributeActions 对象加密并签名所有项目，除已签名但未加密的主键属性和已被忽略的 test 属性以外。

当您与项目加密程序直接交互且您的默认操作为 ENCRYPT_AND_SIGN 时，您必须为主键指定一个替代操作。您可以使用 set_index_keys 方法，该方法对主键使用 SIGN_ONLY；如果是默认操作，也可以使用 DO_NOTHING。

为了指定主键，此示例使用TableInfo对象中的索引键，该索引键通过调用 DynamoDB 来填充。此方法要比硬编码主键名称更加安全。

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={'test': CryptoAction.DO_NOTHING}
```

```
)
actions.set_index_keys(*table_info.protected_index_keys())
```

步骤 6：创建项目的配置

要配置 DynamoDB 加密客户端，请使用您刚才在表项目的配置中CryptoConfig创建的对象。客户端帮助程序类 CryptoConfig 为您创建。

```
crypto_config = CryptoConfig(
    materials_provider=kms_cmp,
    encryption_context=encryption_context,
    attribute_actions=actions
)
```

步骤 7：加密项目

此步将加密并签名项目，但不会将项目放入 DynamoDB 表中。

当您使用客户端帮助程序类时，您的项目会以透明方式加密并签名，然后在您调用帮助程序类的 put_item 方法时添加到您的 DynamoDB 表。当您直接使用项目加密程序时，encrypt 和 put 操作是相互独立的。

首先，创建一个明文项目。

```
plaintext_item = {
    'partition_attribute': 'value1',
    'sort_key': 55,
    'example': 'data',
    'numbers': 99,
    'binary': Binary(b'\x00\x01\x02'),
    'test': 'test-value'
}
```

然后，对该项目进行加密和签名。encrypt_python_item 方法需要 CryptoConfig 配置对象。

```
encrypted_item = encrypt_python_item(plaintext_item, crypto_config)
```

步骤 8：将项目放入表中

此步会将已加密且签名的项目放入 DynamoDB 表中。

```
table.put_item(Item=encrypted_item)
```

要查看加密项目，请对原始 get_item 对象调用 table 方法，而不是对 encrypted_table 对象。它会从 DynamoDB 表获取该项目，无需验证和解密它。

```
encrypted_item = table.get_item(Key=partition_key)['Item']
```

下图展示了一个已加密且签名的示例表项目的一部分。

加密属性值为二进制数据。主键属性 (partition_attribute 和 sort_attribute) 和 test 属性的名称和值保持明文形式。输出还显示包含签名 (*amzn-ddb-map-sig*) 的属性和材料描述属性 (*amzn-ddb-map-desc*)。

```
{
    '*amzn-ddb-map-desc*': Binary(b'\x00\x00\x00\x00\x00\x00\x10amzn-ddb-env-alg\
\x00\x00\x00\xe0AQEBAHhA84wnXjEJdBbBBylRUFcZZK2j7xwh6UyLoL28nQ
+0FAAAAH4wfAYJKoZIhvcNAQcGoG8wbQIBADBoBgkqhkiG9w0BBwEwHgYJYIZIAWUDBAEuMBEEDPeFBydmoJD
izYl0R0C4M7wAK6E1/N/bgTmHI=\x00\x00\x00\x17amzn-ddb-map-signingAlg\x00\x00\x00\nHmacS
\x00\x00\x00\x11/CBC/PKCS5Padding\x00\x00\x00\x10amzn-ddb-sig-alg\x00\x00\x00\x0eHmac
\x00\x00\x00\x0faws-kms-ec-attr\x00\x00\x00\x06*keys*'),
    '*amzn-ddb-map-sig*': Binary(b"\xd3\xc6\xc7\n\xb7#\x13\xd1Y\xea\xe4.|^\xbd\xdf\xe
    'binary': Binary(b'!"\xc5\x92\xd7\x13\x1d\xe8Bs\x9b\x7f\xa8\x8e\x9c\xcf\x10\x1e\x
    'example': Binary(b"'b\x933\x9a+s\xf1\xd6a\xc5\xd5\x1aZ\xed\xd6\xce\xe9X\xf0T\xcb
    'numbers': Binary(b'\xd5\xa0\\d\xcc\x85\xf5\x1e\xb9-f!\xb9\xb8\x8a\x1aT\xbaq\xf7\
    'partition_attribute': 'value1',
    'sort_attribute': 55,
    'test': 'test-value'
}
```

更改数据模型

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

每次加密或解密项目时，您需要提供属性操作，这些操作告诉 DynamoDB 加密客户端要加密并签名的属性、要签名（但不加密）的属性以及要忽略的属性。属性操作不会保存在加密的项目中，并且 DynamoDB 加密客户端不会自动更新您的属性操作。

> **Important**
>
> DynamoDB 加密客户端不支持对现有的未加密 DynamoDB 表数据进行加密。

每当您更改数据模型时（也即，在表项目中添加或删除属性时），都有可能出错。如果您指定的属性操作未考虑到该项目中的所有属性，则该项目可能不会按您希望的方式进行加密和签名。更重要的是，如果您在解密项目时提供的属性操作与在加密项目时提供的属性操作不同，则签名验证可能会失败。

例如，如果用于加密项目的属性操作告知其签署 `test` 属性，则项目中的签名将包含 `test` 属性。但是，如果用于解密项目的属性操作未考虑到 `test` 属性，则验证会失败，因为客户端将尝试验证与包含 `test` 属性的签名。

当多个应用程序读取和写入相同的 DynamoDB 项目时，这是一个特别的问题，因为 DynamoDB 加密客户端必须为所有应用程序中的项目计算相同的签名。对于任何分布式应用程序来说，这也是一个问题，因为属性操作的更改必须传播到所有主机。即使您的 DynamoDB 表是由一个主机在一个过程中访问的，但如果项目变得更加复杂，则建立最佳实践过程也将有助于防止错误。

为避免签名验证错误阻止您读取表项目，请使用以下指南。

- 添加属性 — 如果新的属性更改了属性操作，请在将新属性包括在项目中之前完全部署属性操作更改。
- 移除属性 — 如果您停止在项目中使用属性，请不要更改属性操作。
- 更改操作 — 使用属性操作配置对表项目进行加密后，如果不重新加密表中的每个项目，就无法安全地更改默认操作或现有属性的操作。

签名验证错误可能很难解决，因此最好的方法是防止它们发生。

主题

添加属性

在向表项目添加新属性时，可能需要更改属性操作。为了防止签名验证错误，我们建议您分两步实施此更改。在开始第二阶段之前，请验证第一阶段是否已完成。

1. 在读取或写入表的所有应用程序中更改属性操作。部署这些更改并确认更新已传播到所有目标主机。
2. 将值写入表项目中的新属性。

这种两阶段方法可确保所有应用程序和主机具有相同的属性操作，并且在遇到新属性之前将计算相同的签名。即使该属性的操作为不执行任何操作（不加密或签名），这也很重要，因为某些加密程序的默认设置是加密和签名。

以下示例显示此过程中第一阶段的代码。它们添加了一个新的项目属性 `link`，该属性存储指向另一个表项目的链接。由于此链接必须保持为纯文本格式，因此该示例向其分配仅签名操作。完全部署此更改，然后验证所有应用程序和主机都具有新的属性操作之后，可以开始在表项目中使用 `link` 属性。

Java DynamoDB Mapper

当使用 DynamoDB Mapper 和 AttributeEncryptor 时，默认情况下，除主键以外的所有属性均加密并签名，而主键已签名但未加密。要指定仅签名操作，请使用 @DoNotEncrypt 注释。

本示例将 @DoNotEncrypt 注释用于新的 `link` 属性。

```
@DynamoDBTable(tableName = "ExampleTable")
public static final class DataPoJo {
  private String partitionAttribute;
  private int sortAttribute;
  private String link;

  @DynamoDBHashKey(attributeName = "partition_attribute")
  public String getPartitionAttribute() {
    return partitionAttribute;
  }

  public void setPartitionAttribute(String partitionAttribute) {
    this.partitionAttribute = partitionAttribute;
  }

  @DynamoDBRangeKey(attributeName = "sort_attribute")
  public int getSortAttribute() {
    return sortAttribute;
  }

  public void setSortAttribute(int sortAttribute) {
    this.sortAttribute = sortAttribute;
  }
```

```
  @DynamoDBAttribute(attributeName = "link")
  @DoNotEncrypt
  public String getLink() {
    return link;
  }

  public void setLink(String link) {
    this.link = link;
  }

  @Override
  public String toString() {
    return "DataPoJo [partitionAttribute=" + partitionAttribute + ",
        sortAttribute=" + sortAttribute + ",
        link=" + link + "]";
  }
}
```

Java DynamoDB encryptor

在较低级别的 DynamoDB 加密程序中，必须为每个属性设置操作。本示例使用一个开关语句，其中默认值为 `encryptAndSign`，并为分区键、排序键和新的 `link` 属性指定了例外。在此示例中，如果在使用链接属性代码之前未完全部署它，则链接属性将由某些应用程序加密和签名，而仅由其他应用程序签名。

```
for (final String attributeName : record.keySet()) {
    switch (attributeName) {
        case partitionKeyName:
            // fall through to the next case
        case sortKeyName:
            // partition and sort keys must be signed, but not encrypted
            actions.put(attributeName, signOnly);
            break;
        case "link":
            // only signed
            actions.put(attributeName, signOnly);
            break;
        default:
            // Encrypt and sign all other attributes
            actions.put(attributeName, encryptAndSign);
            break;
    }
```

```
}
```

Python

在适用于 Python 的 DynamoDB 加密客户端中，您可以为所有属性指定默认操作，然后指定例外。

如果您使用的是 Python 客户端帮助程序类，则无需指定主键属性的属性操作。该客户端帮助程序类阻止您加密主键。但是，如果不使用客户端帮助程序类，则必须在分区键和排序键上设置 SIGN_ONLY 操作。如果您不小心加密了分区键或排序键，那么在没有全表扫描的情况下将无法恢复数据。

本示例为新的 `link` 属性指定一个例外，该例外将获取 `SIGN_ONLY` 操作。

```
actions = AttributeActions(
    default_action=CryptoAction.ENCRYPT_AND_SIGN,
    attribute_actions={
      'example': CryptoAction.DO_NOTHING,
      'link': CryptoAction.SIGN_ONLY
    }
)
```

删除属性

如果您在使用 DynamoDB 加密客户端加密的项目中不再需要某个属性，则可以停止使用该属性。但是，请勿删除或更改该属性的操作。如果这样做，然后遇到具有该属性的项目，则为该项目计算的签名将与原始签名不匹配，并且签名验证将失败。

尽管您可能很想从代码中删除该属性的所有痕迹，但请添加一条注释，指出不再使用该项目，而不是删除它。即使您进行全表扫描以删除该属性的所有实例，具有该属性的加密项目也可能会缓存或在配置中的某个地方处于正在进行状态。

排查 DynamoDB 加密客户端应用程序中的问题

> **Note**
>
> 我们的客户端加密库已重命名为 AWS 数据库加密 SDK。以下主题提供有关适用于 Java 的 DynamoDB 加密客户端版本 1.x—2.x 以及适用于 Python 的 DynamoDB 加密客户端版本 1.x—3.x 的信息。有关更多信息，请参阅适用于 DynamoDB 的AWS 数据库加密 SDK 版本支持。

本部分介绍了您在使用 DynamoDB 加密客户端时可能遇到的问题并提供了解决这些问题的建议。

要提供有关 DynamoDB 加密客户端的反馈，请在或存储库中提交问题。aws-dynamodb-encryption-javaaws-dynamodb-encryption-python GitHub

要提供对本文档的反馈，请使用任何页面上的反馈链接。

主题

拒绝访问

问题：拒绝您的应用程序访问其所需的资源。

建议：了解所需权限并将权限添加到您的应用程序所运行的安全环境。

详细信息

要运行使用 DynamoDB 加密客户端库的应用程序，调用方必须具有使用其组件的权限。否则，将会拒绝他们访问必要元素。

- DynamoDB 加密客户端不需要 Amazon Web Services（AWS）账户，也不依赖任何 AWS 服务。但是，如果您的应用程序使用 AWS，则需要有权使用该账户的 AWS 账户和用户。
- DynamoDB 加密客户端不需要 Amazon DynamoDB。但是，如果使用客户端的应用程序创建 DynamoDB 表、将项目放入表中或从表中获取项目，则调用方必须具有在您的 AWS 账户账户中使用所需 DynamoDB 操作的权限。有关详细信息，请参阅《Amazon DynamoDB 开发人员指南》中的访问控制主题。
- 如果您的应用程序使用适用于 Python 的 DynamoDB 加密客户端中的客户端帮助程序类，则调用者必须具有调用 DynamoDB 操作的权限。DescribeTable
- DynamoDB 加密客户端不 AWS Key Management Service 需要 ()。AWS KMS但是，如果您的应用程序使用 Di rect KMS 材料提供程序，或者它使用的是最新提供程序和正在使用的提供程序存储区 AWS KMS，则调用方必须拥有使用 AWS KMS GenerateDataKey和解密操作的权限。

签名验证失败

问题：由于签名验证失败，无法解密某个项目。该项目也可能未按您希望的进行加密和签名。

建议：确保您提供的属性操作考虑到该项目中的所有属性。当解密某个项目时，请确保提供与用于加密该项目的操作匹配的属性操作。

详细信息

您提供的属性操作告诉 DynamoDB 加密客户端要加密并签名的属性、要签名（但不加密）的属性以及要忽略的属性。

如果您指定的属性操作未考虑到该项目中的所有属性，则该项目可能不会按您希望的方式进行加密和签名。如果您在解密项目时提供的属性操作与在加密项目时提供的属性操作不同，则签名验证可能会失败。这是分布式应用程序中的特定问题，在分布式应用程序中，新属性操作可能不会传播到所有主机。

签名验证错误很难解决。为了帮助防止此类错误，请在更改数据模型时采取额外的预防措施。有关更多信息，请参阅 更改数据模型。

旧版本全局表存在的问题

问题：由于签名验证失败，旧版本的 Amazon DynamoDB 全局表中的项目无法解密。

建议：设置属性操作，使得保留的复制字段不会被加密或签名。

详细信息

您可以将 DynamoDB 加密客户端与 DynamoDB 全局表结合使用。我们建议您使用带有多区域 KMS 密钥的全局表，并将 KMS 密钥复制到复制全局表的所有 AWS 区域 位置。

从全局表版本 2019.11.21 开始，您无需任何特殊配置即可将全局表与 DynamoDB 加密客户端结合使用。但是，如果您使用全局表版本 2017.11.29，则必须确保所保留的复制字段不会被加密或签名。

如果您使用全局表版本 2017.11.29，则必须将以下属性的属性操作设置为 Java 中的 `DO_NOTHING` 或 Python 中的 `@DoNotTouch`。

- `aws:rep:deleting`
- `aws:rep:updatetime`
- `aws:rep:updateregion`

如果您使用任何其他版本的全局表，则无需执行任何操作。

最新提供程序表现不佳

问题：您的应用程序响应速度较差，尤其是在更新到较新版本的 DynamoDB 加密客户端之后。

建议：调整 time-to-live值和缓存大小。

详细信息

最新提供程序旨在通过允许有限地重用加密材料来提高使用 DynamoDB 加密客户端的应用程序的性能。为应用程序配置最新提供程序时，您必须在提高性能与缓存和重用所产生的安全问题之间取得平衡。

在较新版本的 DynamoDB 加密客户端中， time-to-live(TTL) 值决定了缓存的加密材料提供程序 CMPs () 的使用时长。TTL 还决定最新提供程序检查新版本的 CMP 的频率。

如果您的 TTL 过长，则应用程序可能会违反您的业务规则或安全标准。如果 TTL 过短，则频繁调用提供程序存储可能会导致您的提供程序存储限制来自您的应用程序和共享您的服务账户的其他应用程序的请求。要解决此问题，请将 TTL 和缓存大小调整为符合延迟和可用性目标并且符合安全标准的值。有关详细信息，请参阅设置一个 time-to-live值。

Amazon DynamoDB Encryption Client 重命名

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

2023 年 6 月 9 日，我们的客户端加密库更名为 AWS 数据库加密 SDK。 AWS 数据库加密软件开发工具包与亚马逊 DynamoDB 兼容。它可以解密和读取旧版 DynamoDB 加密客户端加密的项目。有关旧版 DynamoDB 加密客户端版本的更多信息，请参阅 AWS 适用于 DynamoDB 的数据库加密 SDK 版本支持。

AWS 数据库加密 SDK 提供版本 3。 适用于 DynamoDB 的 Java 客户端加密库的 x，这是对适用于 Java 的 DynamoDB 加密客户端的重大改写。它包括许多更新，例如新的结构化数据格式、改进的多租户支持、无缝架构更改，以及可搜索的加密支持。

要详细了解 AWS 数据库加密 SDK 引入的新功能，请参阅以下主题。

可搜索的加密

您可以设计无需解密整个数据库即可搜索已加密记录的数据库。根据您的威胁模型和查询要求，您可以使用可搜索的加密对已加密记录执行精确匹配搜索或更自定义的复杂查询。

密钥环

AWS 数据库加密 SDK 使用密钥环来执行信封加密。密钥环生成、加密和解密用于保护您的记录的数据密钥。 AWS 数据库加密 SDK 支持使用对称加密或非对称 RSA AWS KMS keys来保护您的数据密钥的密钥 AWS KMS 环，以及 AWS KMS 分层密钥环，使您能够使用对称加密 KMS 密钥保护您的加密材料，而无需在 AWS KMS 每次加密或解密记录时都调用。您还可以使用原始 AES 密钥环和原始 RSA 密钥环指定自己的密钥材料。

无缝架构更改

在配置 AWS 数据库加密 SDK 时，您需要提供加密操作，告诉客户端要加密和签名哪些字段，哪些字段需要签名（但不加密），以及要忽略哪些字段。使用 AWS 数据库加密 SDK 保护记录后，您仍然可以更改数据模型。您可以在单个部署中更新您的加密操作，例如添加或移除已加密的字段。

配置现有 DynamoDB 表以进行客户端加密

旧版本的 DynamoDB 加密客户端旨在在未填充的新表中实现。使用适用于 DynamoDB 的 AWS 数据库加密软件开发工具包，您可以将现有的亚马逊 DynamoDB 表迁移到版本 3。 适用于 DynamoDB 的 Java 客户端加密库中的 x。

参考

我们的客户端加密库已重命名为 AWS 数据库加密 SDK。本开发人员指南仍提供有关 DynamoDB 加密客户端的信息。

以下主题提供了 AWS 数据库加密 SDK 的技术细节。

材料描述的格式

材料描述将用作加密记录的标题。当您使用 AWS 数据库加密 SDK 对字段进行加密和签名时，加密器会在组装加密材料时记录材料描述，并将材料描述存储在加密器添加到记录中的新字段 (aws_dbe_head) 中。材料描述是一种可移植的格式化数据结构，其中包含加密的数据密钥，以及有关如何对记录进行加密和签名的信息。下表描述了构成材料描述的值。字节是按显示的顺序附加的。

值	长度（字节）
Version	1
Signatures Enabled	1
Record ID	32
Encrypt Legend	变量
Encryption Context Length	2
???	变量
Encrypted Data Key Count	1
Encrypted Data Keys	变量
Record Commitment	1

版本

该 aws_dbe_head 字段格式的版本。

已启用签名

对是否为此记录启用 ECDSA 数字签名进行编码。

字节值	含义
0x01	已启用 ECDSA 数字签名（默认）
0x00	已禁用 ECDSA 数字签名

记录编号

随机生成的 256 位值，用于标识记录。记录 ID：

- 唯一地标识加密记录。
- 将材料描述与加密记录绑定。

加密传奇

对已加密的身份验证字段进行的序列化描述。加密图例用于确定解密方法应尝试解密哪些字段。

字节值	含义
0x65	ENCRYPT_AND_SIGN
0x73	SIGN_ONLY

按如下方法序列化加密图例：

1. 按字典顺序按代表其规范路径的字节序列排列。
2. 对于每个字段，按顺序附加上面指定的字节值之一，从而指示是否应加密该字段。

加密上下文长度

加密上下文的长度。这是一个解释为 16 位无符号整数的 2 字节值。最大长度为 65535 个字节。

加密上下文

一组名称值对，其中包含任意非机密经过身份验证的附加数据。

启用 ECDSA 数字签名后，加密上下文将包含密钥值对。{"aws-crypto-footer-ecdsa-key": Qtxt} Qtxt表示按照 SEC 1 版本 2.0 Q 压缩然后经过 base64 编码的椭圆曲线点。

加密数据密钥计数

加密的数据密钥数。这是一个解释为 8 位无符号整数的 1 字节值，它指定加密的数据密钥数。每条记录中加密数据密钥的最大数量为 255。

加密的数据密钥

加密的数据密钥序列。序列长度由加密的数据密钥数和每个密钥的长度决定。该序列包含至少一个加密的数据密钥。

下表描述了组成每个加密的数据密钥的字段。字节是按显示的顺序附加的。

加密的数据密钥结构

字段	长度（字节）
Key Provider ID Length	2
Key Provider ID	变量。等于在前 2 个字节（密钥提供程序 ID 长度）中指定的值。
Key Provider Information Length	2
Key Provider Information	变量。等于在前 2 个字节（密钥提供程序信息长度）中指定的值。
Encrypted Data Key Length	2
Encrypted Data Key	变量。等于在前 2 个字节（加密的数据密钥长度）中指定的值。

密钥提供商 ID 长度

密钥提供程序标识符的长度。这是一个解释为 16 位无符号整数的 2 字节值，它指定包含密钥提供程序 ID 的字节数。

密钥提供商 ID

密钥提供程序标识符。它用于指示加密的数据密钥的提供程序，并且可以进行扩展。

密钥提供者信息长度

密钥提供程序信息的长度。这是一个解释为 16 位无符号整数的 2 字节值，它指定包含密钥提供程序信息的字节数。

密钥提供商信息

密钥提供程序信息。这是由密钥提供程序决定的。

当您使用 AWS KMS 密钥环时，此值包含的亚马逊资源名称 (ARN)。 AWS KMS key

加密数据密钥长度

加密的数据密钥的长度。这是一个解释为 16 位无符号整数的 2 字节值，它指定包含加密的数据密钥的字节数。

加密的数据密钥

加密的数据密钥。这是密钥提供程序加密的数据密钥。

记录承诺

使用提交密钥计算出的 256 位基于哈希的消息身份验证码 (HMAC) 哈希值，计算出前面的所有材料描述字节。

AWS KMS 分层钥匙圈技术细节

AWS KMS 分层密钥环使用唯一的数据密钥来加密每个字段，并使用派生自活动分支密钥的唯一包装密钥对每个数据密钥进行加密。该技术使用计数器模式的密钥派生和带有 HMAC SHA-256 的伪随机函数，通过以下输入派生出 32 字节的包装密钥。

- 一个 16 字节的随机加密盐
- 活动分支密钥
- 密钥提供程序标识符 “” aws-kms-hierarchy 的 UTF-8 编码值

分层密钥环使用派生的包装密钥，使用带有 16 字节身份验证标签和以下输入的 AES-GCM-256 对明文数据密钥的副本进行加密。

- 派生的包装密钥用作 AES-GCM 密码密钥
- 数据密钥用作 AES-GCM 消息
- 使用 12 字节的随机初始化向量（IV）作为 AES-GCM IV
- 包含以下序列化值的其他额外验证数据（AAD）。

值	长度（字节）	解释为
"aws-kms-hierarchy"	17	UTF-8 编码
分支密钥标识符	变量	UTF-8 编码
分支密钥版本	16	UTF-8 编码
加密上下文	变量	UTF-8 编码密钥值对

《AWS 数据库加密 SDK 开发人员指南》的文档历史记录

下表介绍了本文档的一些重要更改。除了这些主要更改以外，我们还会经常更新文档，以改进说明和示例以及处理您发送给我们的反馈意见。要获得有关重要更改的通知，请订阅 RSS 源。

变更	说明	日期
新特征	添加了 AWS KMS ECDH 密钥环和 Ra w EC DH 密钥环的文档。	2024 年 6 月 17 日
公开发行（GA）版本	引入对 DynamoDB 的.NET 客户端加密库的支持。	2024 年 1 月 17 日
公开发行（GA）版本	更新了 3.x 版适用于 DynamoDB 的 Java 客户端加密库的 GA 版本的文档。 **Warning** 不再支持开发者预览版期间所创建的分支密钥。	2023 年 7 月 24 日
DynamoDB 加密客户端的品牌名称重塑	客户端加密库已重命名为 AWS 数据库加密 SDK。	2023 年 6 月 9 日
预览版	添加并更新了 3.x 版适用于 DynamoDB 的 Java 客户端加密库的文档，其中包括新的结构化数据格式、改进的多租户支持、无缝架构更改和可搜索的加密支持。	2023 年 6 月 9 日
文档更改	将“客户主密钥 (CMK)” AWS Key Management Service 一词	2021 年 8 月 30 日

	替换为 “AWS KMS key和 KMS 密钥”。	
新功能	添加了对 AWS Key Management Service (AWS KMS) 多区域密钥的支持。多区域密钥是不同的 AWS KMS 密钥 AWS 区域 ，可以互换使用，因为它们具有相同的密钥 ID 和密钥材料。	2021 年 6 月 8 日
新示例	添加了在 Java DBMapper 中使用 Dynamo 的示例。	2018 年 9 月 6 日
Python 支持	除了 Java 之外，增加了对 Python 的支持。	2018 年 5 月 2 日
初始版本	本文档的初始版本。	2018 年 5 月 2 日

本文属于机器翻译版本。若本译文内容与英语原文存在差异，则一律以英文原文为准。