aws

开发人员指南

Amazon Braket

Amazon Braket: 开发人员指南

Table of Contents

什么是 Amazon Braket？

> **Tip**
>
> 通过以下方式学习量子计算的基础 AWS！注册 Amazon Braket 数字学习计划，完成一系列学习课程和数字评估后，即可获得自己的数字徽章。

Amazon Braket 是一款完全托管的软件 AWS 服务 ，可帮助研究人员、科学家和开发人员开始使用量子计算。量子计算可解决经典计算机无法解决的计算问题，因为它利用了量子力学定律以新方式处理信息。

获取量子计算硬件可能既昂贵又不方便。由于访问权限有限，很难运行算法、优化设计、评估技术的当前状态以及规划何时投入资源以获得最大收益。Braket 可帮助您克服这些挑战。

Braket 提供对各种量子计算技术的单一接入点。使用 Braket，您可以：

- 探索和设计量子算法及混合算法。
- 在不同的量子电路模拟器上测试算法。
- 在不同类型的量子计算机上运行算法。
- 创建概念验证应用程序。

要定义量子问题并对量子计算机进行编程以解决这些问题，需要一套新技能。为帮助您获得这些技能，Braket 提供了不同的环境来模拟和运行您的量子算法。您可以找到最适合您要求的方法，并通过一组名为 Notebook 的示例环境快速入门。

Braket 开发分为三个阶段：

- 构建：Braket 提供完全托管的 Jupyter Notebook 环境，可以让您轻松上手。Braket Notebook 预装了示例算法、资源和开发人员工具，包括 Amazon Braket SDK。借助 Amazon Braket SDK，您可以构建量子算法，然后通过更改一行代码在不同的量子计算机和模拟器上对其进行测试和运行。
- 测试：Braket 提供对完全托管的高性能量子电路模拟器的访问。您可以测试和验证您的电路。Braket 处理所有底层软件组件和 Amazon Elastic Compute Cloud（Amazon EC2）集群，从而减轻了在传统高性能计算（HPC）基础设施上模拟量子电路的负担。
- 运行：Braket 可实现对不同类型的量子计算机的安全按需访问。您可以从、、和Rigetti访问基于门的量子计算机 AQT IonQIQM，也可以访问来自的模拟哈密顿仿真器。 QuEra同时，您无需预先承诺，也不需要通过个别提供商购买访问权限。

关于量子计算和 Braket

量子计算正处于早期发展阶段。重要的一点，我们要明白，目前不存在通用、容错的量子计算机。因此，特定类型的量子硬件更适合每个使用案例，访问各种计算硬件至关重要。Braket 通过第三方提供商提供各种硬件。

现有的量子硬件由于噪声而受到限制，这会带来错误。该行业正处于含噪中型量子（NISQ）时代。在 NISQ 时代，量子计算设备噪声太大，无法维持纯量子算法，如肖尔算法或格罗弗算法。在获得更好的量子误差校正之前，最实用的量子计算需要将经典（传统）计算资源与量子计算机相结合来创建混合算法。Braket 可帮助您使用混合量子算法。

在混合量子算法中，量子处理单元 (QPUs) 被用作协处理器 CPUs，从而加快经典算法中的特定计算。这些算法利用迭代处理，在这种处理中，计算在经典计算机和量子计算机之间移动。例如，量子计算在化学、优化和机器学习中的当前应用基于变分量子算法，变分量子算法是一种混合量子算法。在变分量子算法中，经典优化例程迭代调整参数化量子电路的参数，与根据机器学习训练集中的误差迭代调整神经网络权重的方式大致相同。Braket 提供对 PennyLane 开源软件库的访问权限，该库可帮助您使用变分量子算法。

量子计算在四个主要领域的计算中越来越受欢迎：

- 数论：包括因式分解和密码学（例如，肖尔算法是数论计算的主要量子方法）
- 优化：包括约束满足度、求解线性系统和机器学习
- Oracular 计算：包括搜索、隐藏子组和排序查找（例如，Grover 算法是预言机计算的主要量子方法）
- 模拟：包括直接模拟、节点不变量和量子近似优化算法（QAOA）应用

这些计算类别的应用可在金融服务、生物技术、制造和制药等行业中找到。Braket 提供了功能和示例 Notebook，除某些实际问题外，这些功能和示例 Notebook 已经可以应用于许多概念验证问题。

本节内容：

Amazon Braket 的工作方式

> **Tip**
>
> 通过以下方式学习量子计算的基础 AWS！注册 Amazon Braket 数字学习计划，完成一系列学习课程和数字评估后，即可获得自己的数字徽章。

Amazon Braket 提供对量子计算设备的按需访问，包括按需电路模拟器和不同类型的量子处理单元 ()。QPUs在 Amazon Braket 中，对设备的原子请求是一项量子任务。对于基于门的设备，此请求包括量子电路（包括测量指令和拍摄次数）和其他请求元数据。对于模拟哈密顿模拟器来说，量子任务包含量子寄存器的物理布局以及操纵场的时间和空间依赖性。

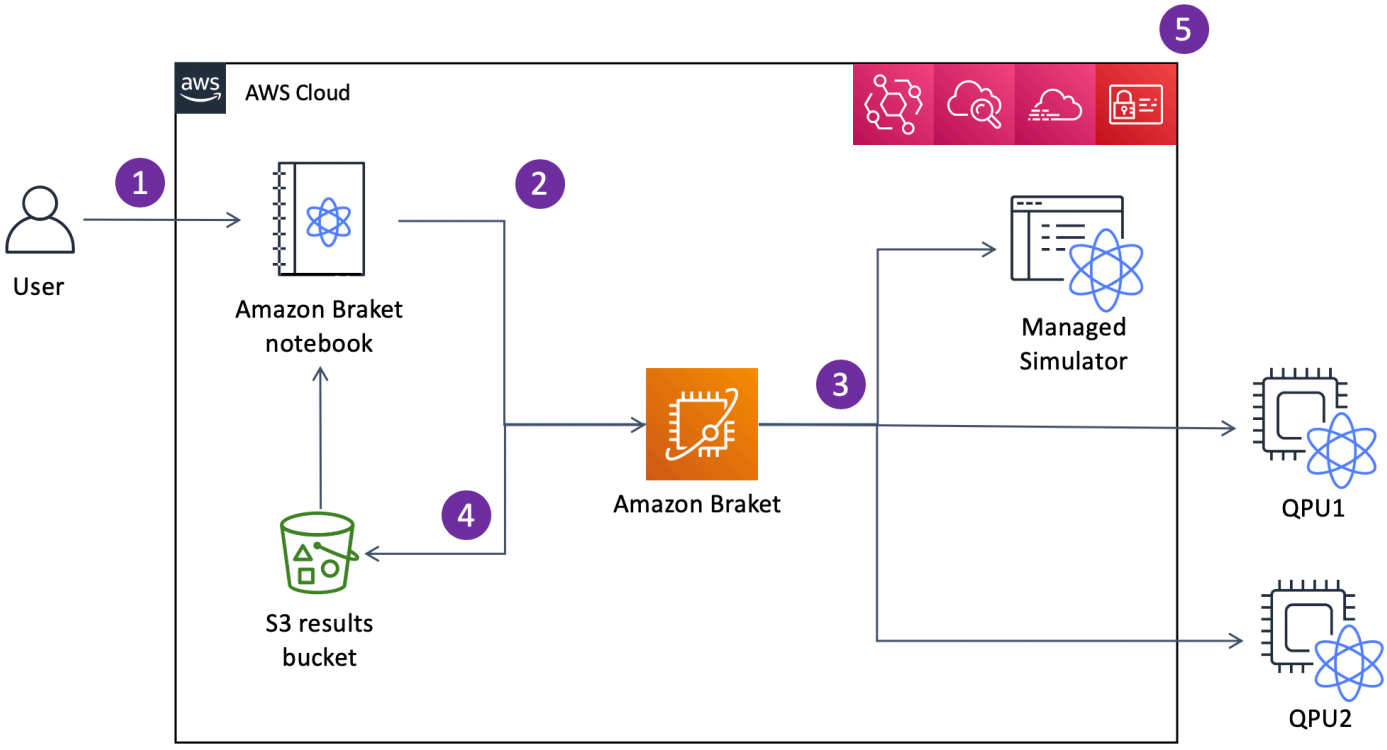
Braket Direct是一项计划，它扩展了你探索量子计算的方式 AWS，加速了研究和创新。您可以在各种量子设备上预留专用容量，直接与量子计算专家接触，并抢先访问下一代功能，包括来自 IonQ 的最新陷阱离子设备 Forte。

在本节中，我们将学习在 Amazon Braket 上运行量子任务的高级流程。

本节内容：

- Amazon Braket 量子任务流
- 第三方数据处理

Amazon Braket 量子任务流



使用Jupyter笔记本电脑，您可以从 Amazon Brake t 控制台或使用 Amazon Braket SDK 定义、提交和监控您的量子任务。您可以直接在 SDK 中构建量子电路。但是，对于模拟哈密顿仿真器，您可以定义寄存器布局和控制字段 (1)。定义量子任务后，您可以选择一台设备来运行该任务，然后将其提交给 Amazon Braket API (2)。根据您选择的设备，对量子任务排队直到设备可用为止，同时将任务发送到 QPU 或模拟器实施 (3)。Amazon Braket 允许您访问各种支持的量子设备 QPUs，包括按需模拟器、本地模拟器和嵌入式仿真器。

处理完您的量子任务后，Amazon Braket 会将结果返回到 Amazon S3 存储桶，数据存储在您的 AWS 账户 (4) 中。同时，SDK 会在后台轮询结果，并在量子任务完成时将其加载到 Jupyter Notebook 中。您还可以在 Amazon Braket 控制台的 Quantum Taskes 页面上或使用 Amazon Braket 的`GetQuantumTask`操作来查看和管理您的量子任务。API

Amazon Braket 与 AWS Identity and Access Management (IAM) CloudWatch、 AWS CloudTrail 亚马逊和亚马逊集成， EventBridge 用于用户访问管理、监控和记录以及基于事件的处理 (5)。

第三方数据处理

提交给 QPU 设备的量子任务在位于第三方提供商运营的设施中的量子计算机上处理。要了解有关 Amazon Braket 安全性和第三方处理的更多信息，请参阅 Amazon Braket 硬件提供商的安全性。

Amazon Braket 术语和概念

> **Tip**
>
> 通过以下方式学习量子计算的基础 AWS！注册 Amazon Braket 数字学习计划，完成一系列学习课程和数字评估后，即可获得自己的数字徽章。

Braket 中使用了以下概念和术语：

模拟哈密顿模拟

模拟哈密顿模拟（AHS）是一种独特的量子计算范式，用于实现多体系统瞬态量子动力学的直接模拟。在 AHS 中，用户直接指定一个时变哈密顿量，通过调整量子计算机，可以直接模拟该哈密顿量下的连续时间演变。AHS 设备通常是专用设备，而不是像基于门的设备这样的通用量子计算机。它们仅限于其可以模拟的一类哈密顿量。然而，由于这些哈密顿量是在设备上自然实现的，因此 AHS 不承担制定像电路那样的算法和执行门操作所需的开销。

Braket

我们以 Braket 表示法（量子力学中的标准表示法）命名了 Braket 服务。该表示法由保罗·狄拉克于 1939 年推出，用于描述量子系统的状态，也被称为狄拉克表示法。

Braket Direct

借助 Braket Direct，您可以保留对自己选择的不同量子设备的专门访问权限，与量子计算专家联系以获取工作负载指导，并尽早使用下一代功能，如可用性有限的新量子设备。

Braket 混合作业

Amazon Braket 有一项名为 Amazon Braket Hybrid Jobs 的功能，该功能可实现混合算法的完全托管执行。Braket 混合作业由三部分组成：

1. 算法的定义，可作为脚本、Python 模块或 Docker 容器提供。
2. 基于 Amazon EC2 的混合作业实例，用于运行您的算法。默认为 ml.m5.xlarge 实例。
3. 用于运行作为算法一部分的量子任务的量子设备。单个混合作业通常包含许多量子任务的集合。

设备

在 Amazon Braket 中，设备是可以运行量子任务的后端。设备可以是 QPU 或量子电路模拟器。要了解更多信息，请参阅 Amazon Braket 支持的设备。

错误缓解

错误缓解包括运行多个物理电路并将它们的测量结果组合在一起以获得更好的结果。有关更多信息，请参阅错误缓解技术。

基于门的量子计算

在基于门的量子计算（QC）（也称为基于电路的 QC）中，计算被分解为基本运算（门）。某些门集是通用的，这意味着每次计算都可以表示为这些门的有限序列。门是量子电路的基块，类似于经典数字电路的逻辑门。

门拍摄限制

门拍摄限制是指每次拍摄的总门数（所有门类型的总和）和每项任务的门拍摄数量。从数学上讲，门拍摄限制可以表示为：

```
Gateshot limit = (Gate count per shot) * (Shot count per task)
```

哈密顿量

物理系统的量子动力学由其哈密顿量决定，哈密顿量对有关系统各组成部分之间的相互作用和外生驱动力影响的所有信息进行编码。在经典机器上，N 量子比特系统的哈密顿量通常表示为复数的 2^N x 2^N 矩阵。通过在量子设备上运行模拟哈密顿模拟，可以避免这些指数级的资源需求。

脉冲

脉冲是传输到量子比特的瞬态物理信号。它由在帧中播放的波形来描述，该波形充当载波信号的支撑，并绑定到硬件通道或端口。客户可以通过提供调制高频正弦载波信号的模拟包络来设计自己的脉冲。该帧的独特特征是频率和相位，这些频率和相位通常被选为与量子比特的 |0−# 和 |1−# 能级之间的能量分离产生共振。因此，门被设置为具有预定形状和校准参数（如振幅、频率和持续时间）的脉冲。模板波形未涵盖的使用案例将通过自定义波形启用，自定义波形将通过提供由固定的物理周期时间分隔的值列表以单样本分辨率进行指定。

量子电路

量子电路是在基于门的量子计算机上定义计算的指令集。量子电路是一系列量子门，它们是 qubit 寄存器上的可逆变换，还有测量指令。

量子电路模拟器

量子电路模拟器是一种在经典计算机上运行并计算量子电路测量结果的计算机程序。对于一般电路，量子模拟的资源需求会随着要模拟的 qubits 的数量而呈指数级增长。Braket 提供对托管（通过 Braket 访问 API）和本地（Amazon Braket SDK 的一部分）量子电路模拟器的访问权限。

量子计算

量子计算机是一种使用量子力学现象（如叠加和纠缠）进行计算的物理设备。量子计算（QC）有不同的范式，如基于门的 QC。

量子处理单元（QPU）

QPU 是一种可以在量子任务上运行的物理量子计算设备。 QPUs 可以基于不同的 QC 范式，例如基于门的 QC。要了解更多信息，请参阅 Amazon Braket 支持的设备。

QPU 原生门

QPU 原生门可以直接映射到 QPU 控制系统的控制脉冲。无需进一步编译即可在 QPU 设备上运行原生门。QPU 支持的门的子集。您可以在 Amazon Braket 控制台的“设备”页面和 Braket SDK 中找到设备的原生门。

QPU 支持的门

QPU 支持的门是 QPU 设备接受的门。这些门可能不会直接在 QPU 上运行，这意味着它们可能需要分解成原生门。您可以在 Amazon Braket 控制台的“设备”页面和 Amazon Braket SDK 上找到支持的设备门。

量子任务

在 Braket 中，量子任务是对设备的原子请求。对于基于门的质量控制设备，这包括量子电路（包括测量指令和 shots 的数量）和其他请求元数据。您可以通过 Amazon Braket SDK 或直接使用该 CreateQuantumTask API 操作来创建量子任务。创建量子任务后，它将排队直到请求的设备变为可用为止。您可以在 Amazon Braket 控制台的“量子任务”页面上或使用 GetQuantumTask 或 SearchQuantumTasks API 操作来查看您的量子任务。

Qubit

量子计算机中的基本信息单位被称为 qubit（量子比特），就像经典计算中的位一样。qubit 是一个双能量子系统，可以通过不同的物理实现来实现，如超导电路或单个离子和原子。其他 qubit 类型基于光子、电子、核自旋或更奇特的量子系统。

Queue depth

Queue depth 指排队等候特定设备的量子任务和混合作业的数量。可通过 Braket Software Development Kit (SDK) 或 Amazon Braket Management Console 访问设备的量子任务和混合作业队列数。

1. 任务队列深度指等待以正常优先级运行的量子任务总数。
2. 优先任务队列深度是指等待通过 Amazon Braket Hybrid Jobs 运行的已提交量子任务的总数。混合作业启动后，这些任务优先于独立任务。

3. 混合作业队列深度是指当前在设备上排队的混合作业总数。Quantum tasks 作为混合作业的一部分提交，具有优先级，汇总在 Priority Task Queue 中。

Queue position

Queue position 指您的量子任务或混合作业在相应设备队列中的当前位置。它可以通过 Braket Software Development Kit (SDK) 或 Amazon Braket Management Console 获得，用于量子任务或混合作业。

Shots

由于量子计算本质上是有一定的概率性的，因此任何电路都需要多次评估才能得到准确的结果。单个电路的执行和测量被称为镜头。电路的拍摄次数（重复执行）是根据所需的结果精度来选择的。

AWS Amazon Braket 的术语和小贴士

IAM 策略

IAM 策略是允许或拒绝对 AWS 服务 和资源的权限的文档。IAM 策略允许您自定义用户对资源的访问级别。例如，您可以允许用户访问您中的所有 Amazon S3 存储桶 AWS 账户，或者仅允许用户访问特定存储桶。

- 最佳实践：授予权限时遵循最低权限的安全原则。通过遵循这一原则，您可以帮助防止用户或角色拥有的权限超过执行其量子任务所需的权限。例如，如果员工只需要访问特定存储桶，请在 IAM 策略中指定该存储桶，而不是向员工授予访问您 AWS 账户中所有存储桶的权限。

IAM 角色

IAM 角色是一种可以代入的身份，可以代入该身份来临时访问权限。您必须先对用户授予切换到您创建的 IAM 角色的权限，然后用户、应用程序或服务才能使用该角色。当有人担任 IAM 角色时，他们会放弃以前在先前角色下拥有的所有权限，并使用新角色的权限。

- 最佳实践：IAM 角色非常适合需要临时而不是长期授予服务或资源访问权限的情况。

Amazon S3 存储桶

亚马逊简单存储服务 (Amazon S3) Simp AWS 服务 le Service 允许您将数据作为对象存储在存储桶中。Amazon S3 存储桶提供无限的存储空间。Amazon S3 存储桶中的最大对象大小为 5 TB。您可以将任何类型的文件数据上传到 Amazon S3 存储桶，如图像、视频、文本文件、备份文件、网站媒体文件、存档文档以及您的 Braket 量子任务结果。

- 最佳实践：您可以设置权限以控制对 S3 存储桶的访问权限。有关更多信息，请参阅 Amazon S3 文档中的存储桶策略。

成本跟踪和节约

> **Tip**
>
> 通过以下方式学习量子计算的基础 AWS！注册 Amazon Braket 数字学习计划，完成一系列学习课程和数字评估后，即可获得自己的数字徽章。

借助 Amazon Braket，您可以按需访问量子计算资源，无需预先承诺。您仅需按实际用量付费。要了解有关定价的更多信息，请访问定价页面。

本节内容：

为 Amazon Braket 设置支出限额 QPUs

Amazon Braket 支出限额为量子处理单元提供了可选的每台设备的成本控制 ()。QPUs

支出限制的运作方式：Amazon Braket 会跟踪您的累积支出，并根据您配置的限额验证每个任务创建请求。如果某项任务的估计费用超过您的剩余支出上限，Amazon Braket 会立即拒绝该任务，并显示验证错误。您可以选择为支出限额配置时间段。通过配置时间段，您可以确保只能在该指定时间段内提交任务。在时间段之外提交的任务将被拒绝。

可选设计：除非您明确启用控件，否则现有工作流程将不受影响。您可以通过删除支出限额来取消所有限制。

> **Note**
>
> 支出限制仅适用于按需和混合作业 QPU 任务。它们不包括模拟器、托管笔记本电脑、Hybrid Job EC2 实例成本和 Braket Direct 预留。要全面管理所有 AWS 服务的成本，请继续使用AWS Budgets。

支出限制措施清单

搜寻

使用以下 AWS CLI 命令，您可以搜索和列出特定 AWS 区域和特定 Braket 设备的支出限制。

```
aws --region {device_region} braket search-spending-limits --filters
 name=deviceArn,operator=EQUAL,values={device_arn}
```

创建

使用以下 AWS CLI 命令，您可以为特定区域的指定量子设备创建新的支出限额。如果设备已存在支出限额，则该请求将被拒绝。

```
aws --region {device_region} braket create-spending-limit --device-arn {device_arn}
 --spending-limit {max_spend}
```

更新

使用以下 AWS CLI 命令，您可以将现有支出限额更新为新的最高支出价值。如果当前支出和排队支出的总和已经高于请求的新最高支出，则该请求将被拒绝。

```
aws --region {device_region} braket update-spending-limit --spending-limit-arn
 {spending_limit_arn} --spending-limit {new_max_spend}
```

如上例所示，您可以提供一个时间段来代替新的最高支出，或者在新的最高支出之外再提供一个时间段。

删除

使用以下 AWS CLI 命令，您可以删除现有支出限额。

```
aws --region {device_region} braket delete-spending-limit --spending-limit-arn
 {spending_limit_arn}
```

如上例所示，您可以提供一个时间段来代替新的最高支出，或者在新的最高支出之外再提供一个时间段。

尽管是可选的，但请务必将区域参数指定为最佳实践。在与设备不同的区域执行的命令将失败，或者如果是`SearchSpendingLimits`，则返回错误的结果。

有关如何使用支出限额的更多示例，请参阅示例笔记本。

任务验证的工作原理

当 AWS 账户发送原本有效的CreateQuantumTask请求时，它会受到以下门控行为的约束。注意：剩余预算是支出限额与排队和当前支出之和之间的差额。（参见下一节）

- 案例 1：任务设备没有支出限制：任务已创建。
- 案例 2：目标设备有支出限制，并且当前时间在支出限制的时间段内：
  - 如果任务的估计成本低于或等于剩余预算： CreateQuantumTask 成功，则任务即被创建。
  - 如果预估成本大于剩余预算：CreateQuantumTask失败，并且不创建任何任务。
- 案例 3：目标设备有支出限制，且当前时间超出支出限制的时间段：CreateQuantumTask失败，且未创建任何任务。

剩余预算是如何计算的

剩余预算是支出限额与当前支出和排队支出之和之间的差额。

当为具有支出限制的设备创建任务时，排队的支出将按任务的估计成本增加。下表的第一行列出了此事件。下表显示了排队的支出和当前支出的变化，具体取决于任务的进度。

旧的量子任务状态	新的量子任务状态	更改为排队支出	更改为当前支出
-	CREATED	按估计成本增加	无更改
CREATED	QUEUED	无更改	无更改
任何	正在运行	无更改	无更改
任何	正在取消	无更改	无更改
正在取消	CANCELLED	按估计成本降低	没有变化
任何	FAILED	按估计成本降低	无更改
正在运行	COMPLETED	按估计成本降低	按估计费用增加（根据部分完成的任务作相应调整）

边缘保护壳

问：在创建支出限额时，队列中已有的任务是否计入排队的支出？

答：不是。 已创建、排队或其他正在进行的任务不计入新创建的支出限额的排队支出。

问：通过更新支出限额来降低支出限额是否会导致已创建、已排队或其他正在进行的量子任务提前终止？

答：不是。

问：达到支出限制的结束时间是否会导致已创建、已排队或其他正在进行的量子任务提前终止？

答：不是。 无论支出限制状态如何，都允许完成已创建、已排队和其他正在进行的任务。

问：缺乏支出限额与零美元支出限额有何不同？

答：没有支出限制，可以不受限制地创建量子任务。零美元的支出限制会阻止所有量子任务。

问：过去或未来（future）的支出上限为零会阻碍所有量子任务的创建？

答：能。

问：在创建支出限额时，已在队列中的任务完成后，估计的费用是否会计入当前支出？

答：不是。 只有在支出限额处于活动状态时提交的任务才会计入累计支出。

近乎实时的成本跟踪

Braket SDK 为您提供了向量子工作负载中添加近乎实时的成本跟踪的选项。我们的每个示例笔记本都包含成本跟踪代码，可为您提供有关 Braket 量子处理单元 (QPUs) 和按需模拟器的最大成本估算。最高成本估算值将以美元显示，不包括任何积分或折扣。

> **Note**
>
> 显示的费用是根据您的 Amazon Braket 模拟器和量子处理单元（QPU）任务使用情况估算的费用。显示的预计费用可能与您的实际费用有所不同。预计费用不考虑任何折扣或积分，您可能会因使用其他服务 [Amazon Elastic Compute Cloud（Amazon EC2）] 等其他服务而收取的额外费用。

成本跟踪 SV1

为了演示如何使用成本跟踪功能，我们将构建一个 Bell State 电路并在我们的 SV1 模拟器上运行它。首先导入 Braket SDK 模块，定义贝尔状态并将 `Tracker()` 函数添加到我们的电路中：

```
#import any required modules
from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.tracking import Tracker

#create our bell circuit
circ = Circuit().h(0).cnot(0,1)
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
with Tracker() as tracker:
    task = device.run(circ, shots=1000).result()

#Your results
print(task.measurement_counts)
```

```
Counter({'00': 500, '11': 500})
```

当您运行 Notebook 时，您可以期待 Bell State 模拟的以下输出。跟踪器功能将显示发送的镜头数量、已完成的量子任务、执行时长、计费的执行持续时间以及以美元为单位的最大成本。每次模拟的执行时间可能会有所不同。

```
import datetime

tracker.quantum_tasks_statistics()
{'arn:aws:braket:::device/quantum-simulator/amazon/sv1':
 {'shots': 1000,
  'tasks': {'COMPLETED': 1},
  'execution_duration': datetime.timedelta(microseconds=4000),
  'billed_execution_duration': datetime.timedelta(seconds=3)}}

tracker.simulator_tasks_cost()
```

```
Decimal('0.0037500000')
```

使用成本跟踪器设置最高成本

您可以使用成本跟踪器来设置计划的最高成本。您可对想要为某一项目花费的金额设定上限。通过这种方式，您可以使用成本跟踪器在执行代码中构建成本控制逻辑。以下示例在 Rigetti QPU 上采用相同的电路，并将成本限制为 1 美元。在我们的代码中运行一次电路迭代的成本为 0.30 美元。我们已将逻辑

设置为重复迭代，直到总成本超过 1 美元；因此，代码片段将运行三次，直到下一次迭代超过 1 美元为止。通常，程序会继续迭代，直到达到所需的最大成本，在本例中为三次迭代。

```
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
with Tracker() as tracker:
    while tracker.qpu_tasks_cost() < 1:
        result = device.run(circ, shots=200).result()
print(tracker.quantum_tasks_statistics())
print(tracker.qpu_tasks_cost(), "USD")
```

```
{'arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3': {'shots': 600, 'tasks':
 {'COMPLETED': 3}}}
1.4400000000 USD
```

> **Note**
>
> 成本跟踪器不会跟踪失败的 TN1 量子任务的持续时间。在 TN1 模拟过程中，如果您的排练已完成，但收缩步骤失败，则您的排练费用将不会显示在成本跟踪器中。

节省成本的最佳实践

请考虑以下使用 Amazon Braket 的最佳实践。节省时间，最大限度地降低成本，并避免常见错误。

使用模拟器进行验证

- 在 QPU 上运行模拟器之前，请使用模拟器验证电路，这样您就可以微调电路，而不会因使用 QPU 而产生费用。
- 尽管在模拟器上运行电路的结果可能与在 QPU 上运行电路的结果不同，但您可以使用模拟器识别编码错误或配置问题。

限制用户访问某些设备

- 您可以设置限制，防止未经授权的用户在某些设备上提交量子任务。限制访问的推荐方法是使用 AWS IAM。有关如何操作的更多信息，请参阅限制访问。
- 我们建议您不要使用管理员账户来授予或限制用户访问 Amazon Braket 设备。

设置账单警报

- 您可以设置账单警报，以便在账单达到预设限额时发出通知。设置闹钟的推荐方法是通过 AWS Budgets。您可以设置自定义预算，并在费用或使用量可能超过预算金额时收到提醒。有关信息，请访问 AWS Budgets。

使用低拍摄计数测试 TN1 量子任务

- 模拟器的成本低于 QPUs，但是如果量子任务以高射击次数运行，则某些模拟器可能会很昂贵。我们建议您使用低 shot 计数来测试您的 TN1 任务。Shot 计数不影响 SV1 和本地模拟器任务的成本。

检查所有区域的量子任务

- 控制台仅显示您当前的量子任务 AWS 区域。在查找已提交的计费量子任务时，请务必查看所有区域。
- 您可以在“受支持的设备”文档页面上查看设备及其相关区域的列表。

Amazon Braket 的 API 参考和存储库

> **Tip**
>
> 通过以下方式学习量子计算的基础 AWS！注册 Amazon Braket 数字学习计划，完成一系列学习课程和数字评估后，即可获得自己的数字徽章。

Amazon Braket 提供了 APIs SDKs、和一个命令行界面，您可以使用它来创建和管理笔记本实例以及训练和部署模型。

- Amazon Braket Python SDK（推荐）
- Amazon Braket API 参考
- AWS Command Line Interface
- 适用于 .NET 的 AWS SDK
- 适用于 C++ 的 AWS SDK
- 适用于 Go 的 AWS SDKAPI Reference
- 适用于 Java 的 AWS SDK
- 适用于 JavaScript 的 AWS SDK

- 适用于 PHP 的 AWS SDK
- AWS SDK for Python (Boto)
- 适用于 Ruby 的 AWS SDK

您还可以从 Amazon Braket 教程 GitHub 存储库中获取代码示例。

- 支架教程 GitHub

核心存储库

下面显示了包含用于 Braket 的密钥包的核心存储库列表：

- Braket Python SDK：使用 Braket Python SDK 在 Python 编程语言的 Jupyter Notebook 上设置您的代码。设置好 Jupyter Notebook 后，您可以在 Braket 设备和模拟器上运行代码
- Braket 架构：Braket SDK 和 Braket 服务之间的合约。
- Braket 默认模拟器：我们所有用于 Braket 的本地量子模拟器（状态向量和密度矩阵）。

插件

然后是各种插件以及各种设备和编程工具。其中包括 Braket 支持的插件以及第三方支持的插件，如下所示。

Amazon Braket 支持的：

- Amazon Braket 算法库：用 Python 编写的预建量子算法目录。按原样运行它们，或者使用它们作为起点来构建更复杂的算法。
- Braket-PennyLane 插件-用PennyLane作 Braket 上的 QML 框架。

第三方（Braket 团队监控并做出贡献）：

- Qiskit-Braket 提供商：使用 Qiskit SDK 访问 Braket 资源。
- Braket-Julia SDK：（实验性）Braket SDK 的 Julia 原生版本

Amazon Braket 支持的区域和设备

> **Tip**
>
> 通过以下方式学习量子计算的基础 AWS！注册 Amazon Braket 数字学习计划，完成一系列学习课程和数字评估后，即可获得自己的数字徽章。

在 Amazon Braket 中，设备代表量子处理单元 (QPU) 或模拟器，你可以调用它来运行量子任务。Amazon Braket 允许从AQT、、IonQIQM、QuEra和访问 QPU 设备。Rigetti此外，还 AWS 提供对按需、本地和嵌入式模拟器的访问。有关嵌入式模拟器的更多信息，请参阅关于嵌入式模拟器。

有关支持的量子硬件提供商的信息，请参阅向提交量子任务 QPUs。有关可用模拟器的信息，请参阅向模拟器提交量子任务。下表列出了可用设备和模拟器。

Provider	设备名称	范式	Type	设备 ARN	Region
AQT	IBEX-Q1	基于门	QPU	arn: aws: braket: eu-north-1::-Q1 device/qpu/aqt/Ibex	eu-north-1
IonQ	Forte-1	基于门	QPU	arn: aws: braket: us-east-1:: -1 device/qpu/ionq/Forte	us-east-1
IonQ	Forte-Enterprise-1	基于门	QPU	arn: aws: braket: us-east-1::-Enterprise-1 device/qpu/ionq/Forte	us-east-1
IQM	Garnet	基于门	QPU	arn: aws: braket: eu-north-1:: device/qpu/iqm/Garnet	eu-north-1
IQM	Emerald	基于门	QPU	arn: aws: braket: eu-north-1:: device/qpu/iqm/Emerald	eu-north-1

Provider	设备名称	范式	Type	设备 ARN	Region
QuEra	Aquila	模拟哈密顿模拟	QPU	arn: aws: braket: us-east-1:: device/qpu/quera/Aquila	us-east-1
Rigetti	Ankaa-3	基于门	QPU	arn: aws: braket: us-west-1:: -3 device/qpu/rigetti/Ankaa	us-west-1
AWS	braket_sv	基于门	本地模拟器	不适用（Braket SDK 中的本地模拟器）	不适用
AWS	braket_dm	基于门	本地模拟器	不适用（Braket SDK 中的本地模拟器）	不适用
AWS	braket_ahs	模拟哈密顿模拟	本地模拟器	不适用（Braket SDK 中的本地模拟器）	不适用
AWS	SV1	基于门	按需模拟器	arn: aws: braket:: 1 device/quantum-simulator/amazon/sv	us-east-1、us-west-1、us-west -1、us-west-2
AWS	DM1	基于门	按需模拟器	arn: aws: braket:: 1 device/quantum-simulator/amazon/dm	us-east-1、us-west-1、us-west -1、us-west-2

Provider	设备名称	范式	Type	设备 ARN	Region
AWS	TN1	基于门	按需模拟器	arn: aws: braket:: 1 device/quantum-simulator/amazon/tn	us-east-1、us-west-2 和 eu-west-2

> **Note**
>
> 设备区 ARNs 分大小写。例如，在使用AQT IBEX-Q1设备时，请验证设备 ARN 是否包含。'Ibex-Q1'

要查看有关可用于 Amazon Brake QPUs t 的更多详细信息，请参阅 Amazon Braket Quant um 计算机。

设备属性

对于所有设备，您可以在 Amazon Braket 控制台的“设备”选项卡上或通过 `GetDevice` API 找到更多设备属性，如设备拓扑、校准数据和原生门设置。使用模拟器构造电路时，Amazon Braket 要求您使用连续的量子比特或索引。使用 SDK 时，以下代码示例显示了如何访问每台可用设备和模拟器的设备属性。

```
from braket.aws import AwsDevice
from braket.devices import LocalSimulator

device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/sv1')
 # SV1
# device = LocalSimulator()
 # Local State Vector Simulator
# device = LocalSimulator("default")
 # Local State Vector Simulator
# device = LocalSimulator(backend="default")
 # Local State Vector Simulator
# device = LocalSimulator(backend="braket_sv")
 # Local State Vector Simulator
# device = LocalSimulator(backend="braket_dm")
 # Local Density Matrix Simulator
```

```
# device = LocalSimulator(backend="braket_ahs")
 # Local Analog Hamiltonian Simulation
# device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/tn1')
 # TN1
# device = AwsDevice('arn:aws:braket:::device/quantum-simulator/amazon/dm1')
 # DM1
# device = AwsDevice('arn:aws:braket:eu-north-1::device/qpu/aqt/Ibex-Q1')
 # AQT IBEX-Q1
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Forte-1')
 # IonQ Forte-1
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1')
 # IonQ Forte-Enterprise-1
# device = AwsDevice('arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet')
 # IQM Garnet
# device = AwsDevice('arn:aws:braket:eu-north-1::device/qpu/iqm/Emerald')
 # IQM Emerald
# device = AwsDevice('arn:aws:braket:us-east-1::device/qpu/quera/Aquila')
 # QuEra Aquila
# device = AwsDevice('arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3')
 # Rigetti Ankaa-3

# Get device properties
device.properties
```

Amazon Braket 的区域和端点数

有关区域和端点的完整列表，请参阅AWS 一般参考。

在 QPU 设备上运行的量子任务可以在该设备所在区域的 Amazon Braket 控制台中查看。使用 Amazon Braket SDK 时，无论您在哪个区域工作，都可以向任何 QPU 设备提交量子任务。SDK 会自动为指定的 QPU 创建与该区域的会话。

Amazon Braket 有以下几种可供选择： AWS 区域

区域名称	Region	Braket 端点
美国东部（弗吉尼亚州北部）	us-east-1	braket.us-east-1.amazonaws.com（IPv4 仅限） braket.us-east-1.api.aws（双堆栈）

区域名称	Region	Braket 端点
美国西部（北加利福尼亚）	us-west-1	braket.us-west-1.amazonaws.com（IPv4 仅限） braket.us-west-1.api.aws（双堆栈）
us-west-2（俄勒冈）	us-west-2	braket.us-west-2.amazonaws.com（IPv4 仅限） braket.us-west-2.api.aws（双堆栈）
eu-north-1（斯德哥尔摩）	eu-north-1	braket.eu-north-1.amazonaws.com（IPv4 仅限） braket.eu-north-1.api.aws（双堆栈）
eu-west-2（伦敦）	eu-west-2	braket.eu-west-2.amazonaws.com（IPv4 仅限） braket.eu-west-2.api.aws（双堆栈）

> **Note**
>
> Amazon Braket 软件开发工具包不支持仅限 IPv6网络的网络。

Amazon Braket 入门

> **Tip**
>
> 通过以下方式学习量子计算的基础 AWS！注册 Amazon Braket 数字学习计划，完成一系列学习课程和数字评估后，即可获得自己的数字徽章。

按照启用 Amazon Braket 中的说明操作后，就可以开始使用 Amazon Braket 了。

入门步骤包括：

- 启用 Amazon Braket
- 创建 Amazon Braket Notebook 实例
- 使用创建 Braket 笔记本实例 CloudFormation

启用 Amazon Braket

> **Tip**
>
> 通过以下方式学习量子计算的基础 AWS！注册 Amazon Braket 数字学习计划，完成一系列学习课程和数字评估后，即可获得自己的数字徽章。

您可以通过 AWS 控制台在您的账户中启用 Amazon Braket。

本节内容：

- 先决条件
- 启用 Amazon Braket 的步骤

先决条件

要启用和运行 Amazon Braket，您必须拥有启动 Amazon Braket 操作的用户或角色的权限。这些权限包含在 AmazonBraketFullAccess IAM 策略中（arn: aws: iam:: aws: policy/）。AmazonBraketFullAccess

> **Note**
>
> 如果您是管理员：
> 要向其他用户授予访问 Amazon Braket 的权限，请通过附加AmazonBraketFullAccess策略或附加您创建的自定义策略向用户授予权限。要详细了解使用 Amazon Braket 所需的权限，请参阅管理对 Amazon Braket 的访问权限。

启用 Amazon Braket 的步骤

1. 使用您的账户登录 Amazon Braket 控制台。 AWS 账户
2. 打开 Amazon Braket 控制台。
3. 在 Braket 登录页面上，单击“入门”，进入服务控制面板页面。服务控制面板顶部的警报将引导您完成以下三个步骤：
   a. 创建服务相关角色（SLR）
   b. 允许访问第三方量子计算机
   c. 创建新的 Jupyter Notebook 实例

要使用第三方量子设备，您需要同意有关您自己与这些设备之间数据传输的某些条件。 AWS本协议的条款和条件在 Amazon Braket 控制台的“权限和设置”页面的“常规”选项卡上提供。

> **Note**
>
> 无需同意“启用第三方设备”协议，即可使用不涉及任何第三方的量子设备，如 Braket 本地模拟器或按需模拟器。
> 如果您要访问第三方硬件，则每个账户只需接受这些条款即可使用第三方设备一次。

创建 Amazon Braket Notebook 实例

> **Tip**
>
> 通过以下方式学习量子计算的基础 AWS！注册 Amazon Braket 数字学习计划，完成一系列学习课程和数字评估后，即可获得自己的数字徽章。

Amazon Braket 提供完全托管的 Jupyter Notebook 供您入门。Amazon Braket 笔记本实例基于亚马逊 A SageMaker I 笔记本实例。以下步骤概括了如何为新客户和现有客户创建新的 Notebook 实例。

Amazon Braket 的新客户：

1. 打开 Amazon Braket 控制台，在左侧窗格中导航至控制面板页面。
2. 在控制面板页面中央的“欢迎使用 Amazon Braket”模式中单击“开始”。提供 Notebook 名称，创建默认的 Jupyter Notebook。
   a. 创建 Notebook 可能需要几分钟时间。
   b. 您的 Notebook 将列在“Notebook”页面上，状态为“待定”。
   c. 当您的笔记本实例准备就绪可供使用时，状态将更改为InService。
   d. 刷新页面，显示 Notebook 的更新状态。

Amazon Braket 的现有客户：

1. 打开 Amazon Braket 控制台，然后在左侧窗格中选择 Notebook。
2. 选择创建 Notebook 实例。
   a. 如果您的 Notebook 为零，请选择标准设置来创建默认的 Jupyter Notebook。
3. 仅使用字母数字和连字符输入 Notebook 实例名称，然后选择首选的可视模式。
4. 为 Notebook 启用或禁用 Notebook 非活动状态管理器。
   a. 如果启用，请在重置 Notebook 之前选择所需的空闲持续时间。重置 Notebook 后，计算费用将不再产生，但存储费用将继续。
   b. 要查看您的 Notebook 实例还剩多少空闲时间，请导航到命令栏，选择 Braket 选项卡，然后选择“不活动管理器”选项卡。

> **Note**
>
> 要保存您的工作，请将您的 SageMaker AI 笔记本实例与 Git 存储库集成。或者，将您的工作移到 `/Braket Algorithms` 和 `/Braket Examples` 文件夹之外，这样它们就不会被重启的 Notebook 实例所覆盖。

5. （可选）使用高级设置，您可以创建具有访问权限、其他配置和网络访问设置的 Notebook：
   a. 在 Notebook 配置中，选择实例类型。
      i. 默认情况下，选择经济实惠的标准实例类型 ml.t3.medium。要了解有关实例定价的更多信息，请参阅 Amazon SageMaker AI 定价。

b. 要将公共 Github 存储库与您的 Notebook 实例相关联，请单击 Git 存储库下拉列表，然后从“存储库”下拉菜单中选择 “从 URL 中克隆公共 Git 存储库”。在 Git 存储库 URL 文本栏中输入存储库的 URL。

c. 在访问权限中，配置任何可选的 IAM 角色、根访问权限和加密密钥。

d. 在网络访问中，为您的 Jupyter Notebook 实例配置自定义网络和访问设置。

6. 查看您的设置，并设置任何用于识别您的 Notebook 实例的标签。单击启动。

> **Note**
>
> 在 Amazon Braket 和亚马逊 AI 控制台中查看和管理你的 Amazon Braket 笔记本实例。SageMaker 其他 Amazon Braket 笔记本设置可通过控制台获得。SageMaker

如果您在 Amazon Braket 控制台中 AWS 使用 Amazon Braket SDK，并且插件已预先加载到您创建的笔记本中。要在自己的计算机上运行，请在运行命令`pip install amazon-braket-sdk`或运行插件命令时安装 SDK 和`pip install amazon-braket-pennylane-plugin` PennyLane 插件。

使用创建 Braket 笔记本实例 CloudFormation

> **Tip**
>
> 通过以下方式学习量子计算的基础 AWS！注册 Amazon Braket 数字学习计划，完成一系列学习课程和数字评估后，即可获得自己的数字徽章。

您可以使用 CloudFormation 来管理您的 Amazon Braket 笔记本实例。Braket 笔记本实例建立在 Amazon A SageMaker I 之上。使用 CloudFormation，您可以使用描述预期配置的模板文件来配置笔记本实例。模板文件以 JSON 或 YAML 格式写入。您可以采用有序且可重复的方式创建、更新和删除实例。当您管理 AWS 账户中的多个 Braket Notebook 实例时，您会发现这很有用。

为 Braket 笔记本创建 CloudFormation 模板后，您可以使用 CloudFormation 来部署该资源。有关更多信息，请参阅CloudFormation 用户指南中的在 CloudFormation 控制台上创建堆栈。

要使用创建 Braket 笔记本实例 CloudFormation，请执行以下三个步骤：

1. 创建 A SageMaker I 生命周期配置脚本。

2. 创建由 A SageMaker I 担任的 AWS Identity and Access Management (IAM) 角色。
3. 创建带有前缀的 SageMaker AI 笔记本实例 **amazon-braket-**

您可以对自己创建的所有 Braket Notebook 重复使用生命周期配置。您还可以为分配相同执行权限的 Braket Notebook 重复使用 IAM 角色。

本节内容：

步骤 1：创建 A SageMaker I 生命周期配置脚本

使用以下模板创建 A SageMaker I 生命周期配置脚本。该脚本为 Braket 自定义 SageMaker AI 笔记本实例。有关生命周期 CloudFormation 资源的配置选项，请参阅CloudFormation 用户指南AWS::SageMaker::NotebookInstanceLifecycleConfig中的。

```
BraketNotebookInstanceLifecycleConfig:
    Type: "AWS::SageMaker::NotebookInstanceLifecycleConfig"
    Properties:
      NotebookInstanceLifecycleConfigName: BraketLifecycleConfig-${AWS::StackName}
      OnStart:
      - Content:
          Fn::Base64: |
            #!/usr/bin/env bash
            sudo -u ec2-user -i #EOS
            curl -o braket-notebook-lcc.zip https://d3ded4lzb1lnme.cloudfront.net/
notebook/braket-notebook-lcc.zip
            unzip braket-notebook-lcc.zip
            ./install.sh
            EOS

            exit 0
```

第 2 步：创建由 Amazon A SageMaker I 担任的 IAM 角色

当您使用 Braket 笔记本实例时， SageMaker AI 会代表您执行操作。例如，假设您在受支持的设备上使用电路运行 Braket Notebook。在笔记本实例中， SageMaker AI 会为您在 Braket 上运行操作。笔

记本执行角色定义了允许 SageMaker AI 代表您执行的确切操作。有关更多信息，请参阅 SageMaker A mazon AI 开发者指南中的 SageMaker AI 角色。

使用以下示例创建拥有所需权限的 Braket Notebook 执行角色。您可以根据需要修改策略。

> **Note**
>
> 确保该角色有权对前缀为 `braketnotebookcdk-"` 的 Amazon S3 存储桶执行 `s3:ListBucket` 和 `s3:GetObject` 操作。生命周期配置脚本需要这些权限才能复制 Braket Notebook 安装脚本。

```
ExecutionRole:
    Type: "AWS::IAM::Role"
    Properties:
      RoleName: !Sub AmazonBraketNotebookRole-${AWS::StackName}
      AssumeRolePolicyDocument:
        Version: "2012-10-17"
        Statement:
        -
          Effect: "Allow"
          Principal:
            Service:
              - "sagemaker.amazonaws.com"
          Action:
          - "sts:AssumeRole"
      Path: "/service-role/"
      ManagedPolicyArns:
        - arn:aws:iam::aws:policy/AmazonBraketFullAccess
      Policies:
        -
          PolicyName: "AmazonBraketNotebookPolicy"
          PolicyDocument:
            Version: "2012-10-17"
            Statement:
              - Effect: Allow
                Action:
                  - s3:GetObject
                  - s3:PutObject
                  - s3:ListBucket
                Resource:
                  - arn:aws:s3:::amazon-braket-*
```

```
                - arn:aws:s3:::braketnotebookcdk-*
            - Effect: "Allow"
              Action:
                - "logs:CreateLogStream"
                - "logs:PutLogEvents"
                - "logs:CreateLogGroup"
                - "logs:DescribeLogStreams"
              Resource:
                - !Sub "arn:aws:logs:*:${AWS::AccountId}:log-group:/aws/sagemaker/*"
            - Effect: "Allow"
              Action:
                - braket:*
              Resource: "*"
```

步骤 3：创建带有前缀的 SageMaker AI 笔记本实例 **amazon-braket-**

使用 A SageMaker I 生命周期脚本以及在步骤 1 和步骤 2 中创建的 IAM 角色创建 A SageMaker I 笔记本实例。Notebook 实例是为 Braket 定制的，可以通过 Amazon Braket 控制台进行访问。有关此 CloudFormation 资源配置选项的更多信息，请参阅CloudFormation 用户指南AWS::SageMaker::NotebookInstance中的。

```
BraketNotebook:
    Type: AWS::SageMaker::NotebookInstance
    Properties:
      InstanceType: ml.t3.medium
      NotebookInstanceName: !Sub amazon-braket-notebook-${AWS::StackName}
      RoleArn: !GetAtt ExecutionRole.Arn
      VolumeSizeInGB: 30
      LifecycleConfigName: !GetAtt
 BraketNotebookInstanceLifecycleConfig.NotebookInstanceLifecycleConfigName
```

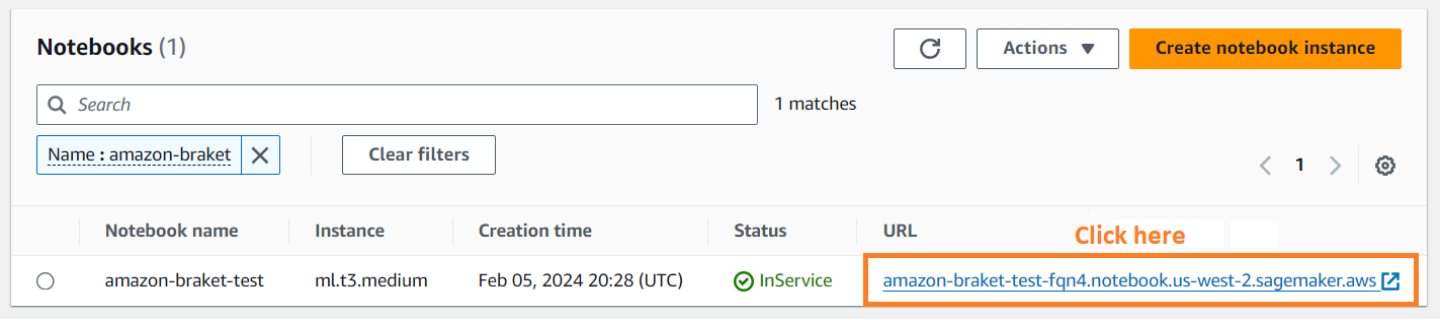
使用 Amazon Braket 构建您的量子任务

Braket 提供完全托管的 Jupyter Notebook 环境，可以让您轻松入门。Braket Notebook 预装了示例算法、资源和开发人员工具，包括 Amazon Braket SDK。借助 Amazon Braket SDK，您可以构建量子算法，然后通过更改一行代码在不同的量子计算机和模拟器上对其进行测试和运行。

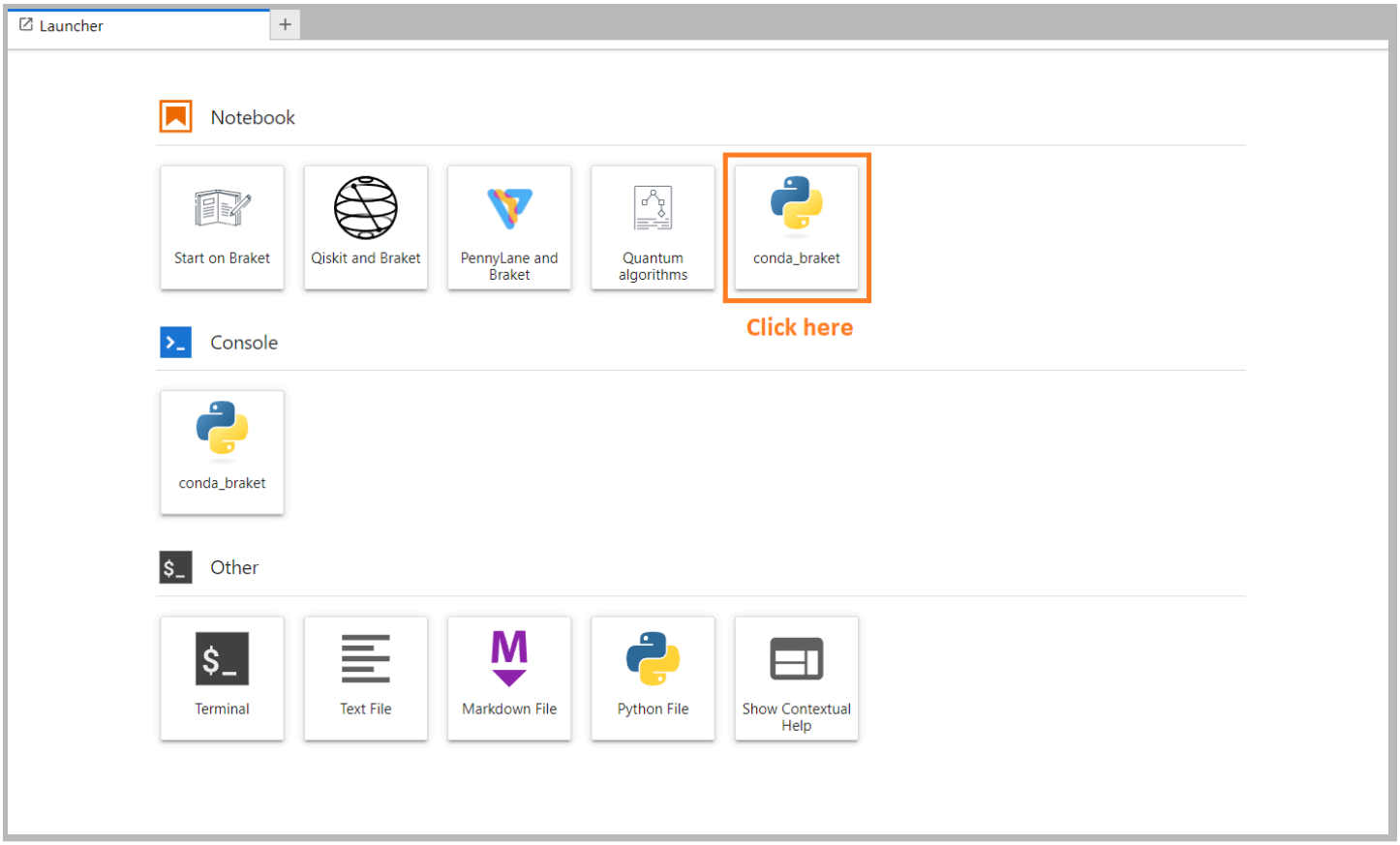
本节内容：

构建您的第一个电路

启动 Notebook 实例后，选择您刚刚创建的 Notebook，使用标准 Jupyter 界面打开该实例。



Amazon Braket Notebook 实例预装了 Amazon Braket SDK 及其所有依赖项。首先创建一个带有 `conda_braket` 内核的新 Notebook。



您可以从一个简单的“您好，世界！”示例开始。首先，构造一个准备贝尔态的电路，然后在不同设备上运行该电路以获得结果。

首先导入 Begin，导入 Amazon Braket SDK 模块并定义一个简单BRAKETlong的 SDK 模块并定义基本的 Bell State 电路。

```
import boto3
from braket.aws import AwsDevice
from braket.devices import LocalSimulator
from braket.circuits import Circuit

# Create the circuit
bell = Circuit().h(0).cnot(0, 1)
```

您可以使用以下命令对电路进行可视化：

```
print(bell)
```

```
T  : #  0  #  1  #
```

```
      #####
q0 : ## H ########
      #####   #
            #####
q1 : ######## X ##
            #####
T  : #  0  #  1  #
```

在本地模拟器上运行您的电路

接下来，选择运行电路的量子设备。Amazon Braket SDK 附带本地模拟器，用于快速制作原型和测试。对于较小的电路，我们建议使用本地模拟器，最大可达 25 个 qubits（取决于您的本地硬件）。

要实例化本地模拟器，请执行以下操作：

```
# Instantiate the local simulator
local_sim = LocalSimulator()
```

然后运行电路：

```
# Run the circuit
result = local_sim.run(bell, shots=1000).result()
counts = result.measurement_counts
print(counts)
```

您应该会看到这样的结果：

```
Counter({'11': 503, '00': 497})
```

正如预期的那样，您准备的具体贝尔态是 |00# 和 |11# 的相等叠加，以及与测量结果大致相等（最多为 shot 噪声）的 00 和 11 分布。

在按需模拟器上运行您的电路

Amazon Braket 还允许访问高性能按需模拟器 SV1，用于运行更大的电路。SV1 是一款按需状态向量模拟器，允许模拟多达 34 个 qubits 的量子电路。您可以SV1在 “支持的设备” 部分和 AWS 控制台中找到更多信息。在 SV1（和 TN1 及任何 QPU 上）上运行量子任务时，量子任务的结果将存储在您账户的 S3 存储桶中。如果您未指定存储桶，Braket SDK 会为您创建一个默认存储桶 `amazon-braket-{region}-{accountID}`。要了解更多信息，请参阅管理 Amazon Braket 的访问权限。

> **Note**
>
> 填写您的实际现有存储桶名称，以下示例显示 `amazon-braket-s3-demo-bucket` 为您的存储桶名称。Amazon Braket 的存储桶名称始终以 `amazon-braket-` 开头，其后跟的是您添加的其他识别字符。如果您需要有关如何设置 S3 存储桶的信息，请参阅 Amazon S3 入门。

```
# Get the account ID
aws_account_id = boto3.client("sts").get_caller_identity()["Account"]

# The name of the bucket
my_bucket = "amazon-braket-s3-demo-bucket"

# The name of the folder in the bucket
my_prefix = "simulation-output"
s3_folder = (my_bucket, my_prefix)
```

要运行电路 SV1，您必须提供先前在 `.run()` 调用中作为位置参数选择的 S3 存储桶的位置。

```
# Choose the cloud-based on-demand simulator to run your circuit
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")

# Run the circuit
task = device.run(bell, s3_folder, shots=100)

# Display the results
print(task.result().measurement_counts)
```

Amazon Braket 控制台提供了有关您的量子任务的更多信息。导航到控制台中的“量子任务”选项卡，您的量子任务应该位于列表顶部。或者，您可以使用唯一的量子任务 ID 或其他条件搜索您的量子任务。

> **Note**
>
> 90 天后，Amazon Braket 会自动删除与您的量子任务 IDs 相关的所有量子任务和其他元数据。有关更多信息，请参阅数据留存。

在 QPU 上运行

使用 Amazon Braket，您只需更改一行代码即可在物理量子计算机上运行前面的量子电路示例。Amazon Braket 允许访问各种量子处理单元 (QPU) 设备。您可以在 “支持的设备” 部分和 AWS 控制台的 “设备” 选项卡下找到有关不同设备和可用性窗口的信息。以下示例说明了如何实例化 IQM 设备。

```
# Choose the IQM hardware to run your circuit
device = AwsDevice("arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet")
```

或者选择带有以下代码的 IonQ 设备：

```
# Choose the Ionq device to run your circuit
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1")
```

选择设备后，在运行工作负载之前，您可以使用以下代码查询设备队列深度，以确定量子任务或混合作业的数量。此外，客户可在 Amazon Braket Management Console 的“设备”页面上查看设备特定的队列深度。

```
# Print your queue depth
print(device.queue_depth().quantum_tasks)
# Returns the number of quantum tasks queued on the device
# {<QueueType.NORMAL: 'Normal'>: '0', <QueueType.PRIORITY: 'Priority'>: '0'}

print(device.queue_depth().jobs)
# Returns the number of hybrid jobs queued on the device
# '2'
```

当您运行任务时，Amazon Braket SDK 会轮询结果（默认超时时间为 5 天）。您可以通过修改 `.run()` 命令中的 `poll_timeout_seconds` 参数来更改此默认值，如以下示例所示。请记住，如果您的轮询超时时间太短，则可能无法在轮询时间内返回结果，例如当 QPU 不可用且返回本地超时错误时。您可以通过调用 `task.result()` 函数来重新开始轮询。

```
# Define quantum task with 1 day polling timeout
task = device.run(bell, s3_folder, poll_timeout_seconds=24*60*60)
print(task.result().measurement_counts)
```

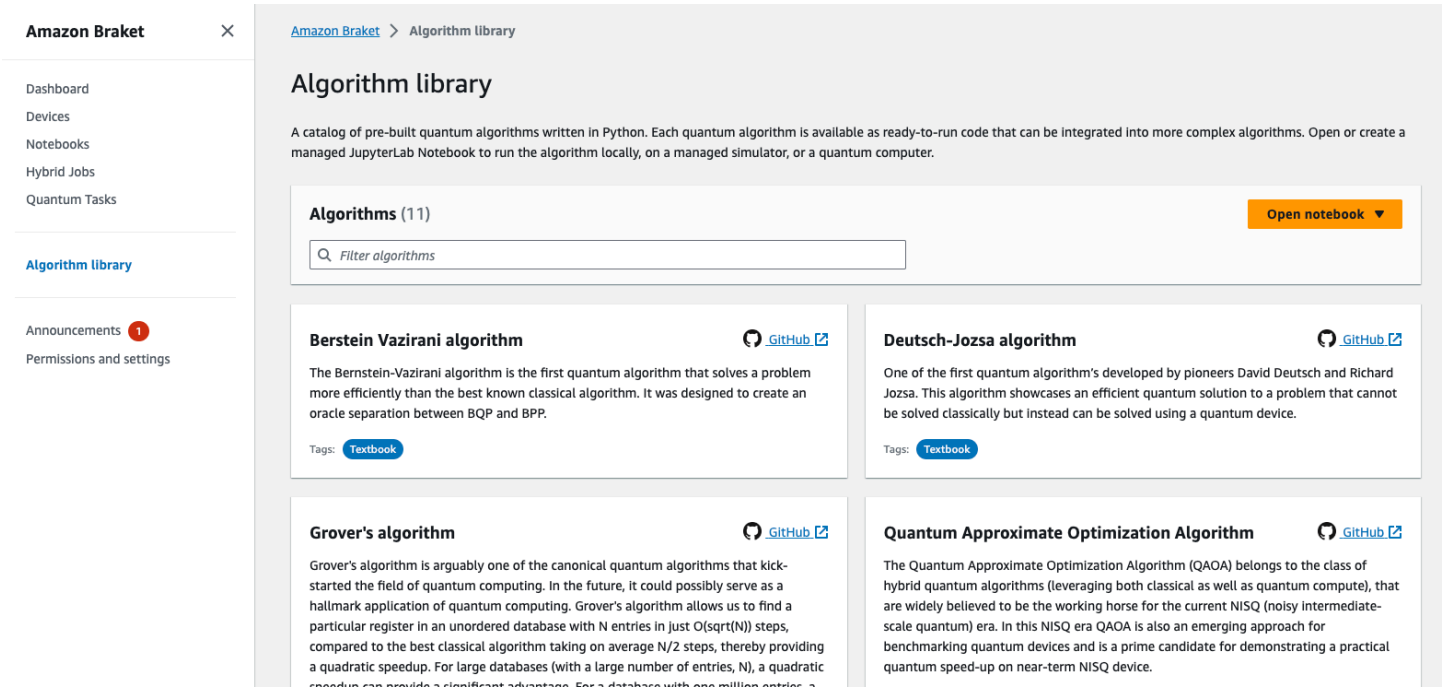
此外，提交量子任务或混合作业后，您可以调用 `queue_position()` 函数来检查队列位置。

```
print(task.queue_position().queue_position)
# Return the number of quantum tasks queued ahead of you
```

```
# '2'
```

构建您的第一个量子算法

Amazon Braket 算法库是用 Python 编写的预先构建的量子算法的目录。按原样运行这些算法，或者将它们作为构建更复杂算法的起点使用。您可以从 Braket 控制台访问算法库。有关更多信息，请参阅 Braket Github 算法库。



Braket 控制台提供了算法库中每种可用算法的描述。选择一个 GitHub 链接以查看每种算法的详细信息，或者选择 “打开笔记本” 以打开或创建包含所有可用算法的笔记本。如果您选择 Notebook 选项，则可以在 Notebook 的根文件夹中找到 Braket 算法库。

在 SDK 中构造电路

本节提供了定义电路、查看可用门、扩展电路以及查看每个设备支持的门的示例。它还包含有关如何进行手动 qubits 分配、指示编译器完全按照定义运行电路以及如何使用噪声模拟器构造噪声电路的说明。

当然，你也可以在 Braket 中为各种门设置脉冲电平。 QPUs有关更多信息，请参阅 Amazon Braket 上的脉冲控制。

本节内容：

- 门和电路

- 程序集
- 部分测量
- 手动 qubit 分配
- 逐字记录编译
- 噪声模拟

门和电路

量子门和电路是在 Amazon Braket Python SDK 的 `braket.circuits` 类别中定义的。在 SDK 中，您可以通过调用 `Circuit()` 来实例化一个新的电路对象。

示例：定义电路

该示例首先定义了一个由四个 qubits（标记为 `q0`、`q1`、`q2` 和 `q3`）组成的样本电路，包括标准的单量子比特的 Hadamard 门和两个量子比特的 CNOT 门。您可以通过调用 `print` 函数来实现此电路的可视化，如下例所示。

```
# Import the circuit module
from braket.circuits import Circuit

# Define circuit with 4 qubits
my_circuit = Circuit().h(range(4)).cnot(control=0, target=2).cnot(control=1, target=3)
print(my_circuit)
```

```
T  : #  0  #     1     #
      #####
q0 : ## H ###############
      #####   #
      #####   #
q1 : ## H ###############
      #####   #     #
      ##### #####   #
q2 : ## H ### X ########
      ##### #####   #
      #####       #####
q3 : ## H ######### X ##
      #####       #####
T  : #  0  #     1     #
```

示例：定义参数化电路

在此示例中，我们定义了一个电路，该电路的门依赖于自由参数。我们可以通过指定这些参数的值来构建新电路，或者在提交电路时，在某些设备上将其作为量子任务运行。

```
from braket.circuits import Circuit, FreeParameter

# Define a FreeParameter to represent the angle of a gate
alpha = FreeParameter("alpha")

# Define a circuit with three qubits
my_circuit = Circuit().h(range(3)).cnot(control=0, target=2).rx(0, alpha).rx(1, alpha)
print(my_circuit)
```

您可以通过向电路提供单个参数 `float`（这是所有自由参数将采用的值）或关键字参数来指定每个参数的值，从而从参数化电路中构造一个新的非参数化电路，如下所示。

```
my_fixed_circuit = my_circuit(1.2)
my_fixed_circuit = my_circuit(alpha=1.2)
print(my_fixed_circuit)
```

请注意，`my_circuit` 是未修改的，因此您可以使用它来实例化许多具有固定参数值的新电路。

示例：修改电路中的门

以下示例定义了一个电路，该电路的门使用了控制和功率修改器。您可以使用这些修改来创建新的门，如受控的 Ry 门。

```
from braket.circuits import Circuit

# Create a bell circuit with a controlled x gate
my_circuit = Circuit().h(0).x(control=0, target=1)

# Add a multi-controlled Ry gate of angle .13
my_circuit.ry(angle=.13, target=2, control=(0, 1))

# Add a 1/5 root of X gate
my_circuit.x(0, power=1/5)

print(my_circuit)
```

只有本地模拟器支持门修改器。

示例：查看所有可用的门

以下示例说明了如何查看 Amazon Braket 中的所有可用门。

```
from braket.circuits import Gate
# Print all available gates in Amazon Braket
gate_set = [attr for attr in dir(Gate) if attr[0].isupper()]
print(gate_set)
```

这段代码的输出列出了所有的门。

```
['CCNot', 'CNot', 'CPhaseShift', 'CPhaseShift00', 'CPhaseShift01', 'CPhaseShift10',
 'CSwap', 'CV', 'CY', 'CZ', 'ECR', 'GPhase', 'GPi', 'GPi2', 'H', 'I', 'ISwap', 'MS',
 'PRx', 'PSwap', 'PhaseShift', 'PulseGate', 'Rx', 'Ry', 'Rz', 'S', 'Si', 'Swap', 'T',
 'Ti', 'U', 'Unitary', 'V', 'Vi', 'X', 'XX', 'XY', 'Y', 'YY', 'Z', 'ZZ']
```

通过调用该类电路的方法，可以将这些门中的任何一个附加到电路中。例如，调用 `circ.h(0)`，在第一个 qubit 门上添加 Hadamard 门。

> **Note**
>
> 门已追加到位，以下示例将上一个示例中列出的所有门添加到同一个电路中。

```
circ = Circuit()
# toffoli gate with q0, q1 the control qubits and q2 the target.
circ.ccnot(0, 1, 2)
# cnot gate
circ.cnot(0, 1)
# controlled-phase gate that phases the |11> state, cphaseshift(phi) =
 diag((1,1,1,exp(1j*phi))), where phi=0.15 in the examples below
circ.cphaseshift(0, 1, 0.15)
# controlled-phase gate that phases the |00> state, cphaseshift00(phi) =
 diag([exp(1j*phi),1,1,1])
circ.cphaseshift00(0, 1, 0.15)
# controlled-phase gate that phases the |01> state, cphaseshift01(phi) =
 diag([1,exp(1j*phi),1,1])
circ.cphaseshift01(0, 1, 0.15)
# controlled-phase gate that phases the |10> state, cphaseshift10(phi) =
 diag([1,1,exp(1j*phi),1])
circ.cphaseshift10(0, 1, 0.15)
# controlled swap gate
circ.cswap(0, 1, 2)
```

```
# swap gate
circ.swap(0,1)
# phaseshift(phi)= diag([1,exp(1j*phi)])
circ.phaseshift(0,0.15)
# controlled Y gate
circ.cy(0, 1)
# controlled phase gate
circ.cz(0, 1)
# Echoed cross-resonance gate applied to q0, q1
circ = Circuit().ecr(0,1)
# X rotation with angle 0.15
circ.rx(0, 0.15)
# Y rotation with angle 0.15
circ.ry(0, 0.15)
# Z rotation with angle 0.15
circ.rz(0, 0.15)
# Hadamard gates applied to q0, q1, q2
circ.h(range(3))
# identity gates applied to q0, q1, q2
circ.i([0, 1, 2])
# iswap gate, iswap = [[1,0,0,0],[0,0,1j,0],[0,1j,0,0],[0,0,0,1]]
circ.iswap(0, 1)
# pswap gate, PSWAP(phi) = [[1,0,0,0],[0,0,exp(1j*phi),0],[0,exp(1j*phi),0,0],
[0,0,0,1]]
circ.pswap(0, 1, 0.15)
# X gate applied to q1, q2
circ.x([1, 2])
# Y gate applied to q1, q2
circ.y([1, 2])
# Z gate applied to q1, q2
circ.z([1, 2])
# S gate applied to q0, q1, q2
circ.s([0, 1, 2])
# conjugate transpose of S gate applied to q0, q1
circ.si([0, 1])
# T gate applied to q0, q1
circ.t([0, 1])
# conjugate transpose of T gate applied to q0, q1
circ.ti([0, 1])
# square root of not gate applied to q0, q1, q2
circ.v([0, 1, 2])
# conjugate transpose of square root of not gate applied to q0, q1, q2
circ.vi([0, 1, 2])
# exp(-iXX theta/2)
```

```
circ.xx(0, 1, 0.15)
# exp(i(XX+YY) theta/4), where theta=0.15 in the examples below
circ.xy(0, 1, 0.15)
# exp(-iYY theta/2)
circ.yy(0, 1, 0.15)
# exp(-iZZ theta/2)
circ.zz(0, 1, 0.15)
# IonQ native gate GPi with angle 0.15 applied to q0
circ.gpi(0, 0.15)
# IonQ native gate GPi2 with angle 0.15 applied to q0
circ.gpi2(0, 0.15)
# IonQ native gate MS with angles 0.15, 0.15, 0.15 applied to q0, q1
circ.ms(0, 1, 0.15, 0.15, 0.15)
```

除了预定义的门设置之外，您还可以将自定义的单一门应用于电路。它们可以是单量子比特门（如以下源代码所示），也可以是应用于参数 `targets` 定义的 qubits 的多量子比特门。

```
import numpy as np

# Apply a general unitary
my_unitary = np.array([[0, 1],[1, 0]])
circ.unitary(matrix=my_unitary, targets=[0])
```

示例：扩展现有电路

您可以通过添加指令来扩展现有电路。`Instruction` 是一种量子指令，该指令描述了要在量子设备上执行的量子任务。`Instruction` 运算符仅包括 `Gate` 类型的对象。

```
# Import the Gate and Instruction modules
from braket.circuits import Gate, Instruction

# Add instructions directly.
circ = Circuit([Instruction(Gate.H(), 4), Instruction(Gate.CNot(), [4, 5])])

# Or with add_instruction/add functions
instr = Instruction(Gate.CNot(), [0, 1])
circ.add_instruction(instr)
circ.add(instr)

# Specify where the circuit is appended
circ.add_instruction(instr, target=[3, 4])
circ.add_instruction(instr, target_mapping={0: 3, 1: 4})
```

```
# Print the instructions
print(circ.instructions)
# If there are multiple instructions, you can print them in a for loop
for instr in circ.instructions:
     print(instr)

# Instructions can be copied
new_instr = instr.copy()
# Appoint the instruction to target
new_instr = instr.copy(target=[5, 6])
new_instr = instr.copy(target_mapping={0: 5, 1: 6})
```

示例：查看每台设备支持的门

模拟器支持 Braket SDK 中的所有门，但是 QPU 设备支持的子集较小。您可以在设备属性中找到设备支持的门。下面给出了 IonQ 设备的一个示例：

```
# Import the device module
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Aria-1")

# Get device name
device_name = device.name
# Show supportedQuantumOperations (supported gates for a device)
device_operations = device.properties.dict()['action']['braket.ir.openqasm.program']
['supportedOperations']
print('Quantum Gates supported by {}:\n {}'.format(device_name, device_operations))
```

```
Quantum Gates supported by Aria 1:
 ['x', 'y', 'z', 'h', 's', 'si', 't', 'ti', 'v', 'vi', 'rx', 'ry', 'rz', 'cnot',
 'swap', 'xx', 'yy', 'zz']
```

受支持的门可能需要编译成原生门，然后才能在量子硬件上运行。当您提交电路时，Amazon Braket 会自动执行此编译。

示例：以编程方式检索设备支持的原生门的保真度

您可以在 Braket 控制台的“设备”页面上查看保真度信息。有时，以编程方式访问相同的信息会很有帮助。以下代码显示如何提取 QPU 的两门之间的两个 qubit 门保真度。

```
# Import the device module
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")

# Specify the qubits
a=10
b=11
edge_properties_entry =
 device.properties.standardized.twoQubitProperties['10-11'].twoQubitGateFidelity
gate_name = edge_properties_entry[0].gateName
fidelity = edge_properties_entry[0].fidelity
print(f"Fidelity of the {gate_name} gate between qubits {a} and {b}: {fidelity}")
```

程序集

程序集在单个量子任务中高效运行多个量子电路。在这一项任务中，您可以提交多达 100 个量子电路或一个包含多达 100 个不同参数集的单个参数电路。此操作可最大限度地缩短后续电路执行的时间间隔，并降低量子任务处理开销。目前，Amazon Braket Local Simulator 等设备和设备都支持程序集 IQM。AQT Rigetti

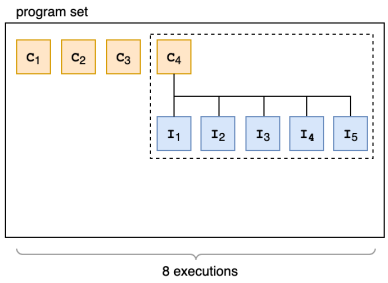
定义一个 ProgramSet

以下第一个代码示例演示了如何同时使用参数化电路和不带参数的电路来构建 `ProgramSet`。

```
from braket.aws import AwsDevice
from braket.circuits import Circuit, FreeParameter
from braket.program_sets.circuit_binding import CircuitBinding
from braket.program_sets import ProgramSet

# Initialize the quantum device
device = AwsDevice("arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet")

# Define circuits
circ1 = Circuit().h(0).cnot(0, 1)
circ2 = Circuit().rx(0, 0.785).ry(1, 0.393).cnot(1, 0)
circ3 = Circuit().t(0).t(1).cz(0, 1).s(0).cz(1, 2).s(1).s(2)
parameterize_circuit = Circuit().rx(0, FreeParameter("alpha")).cnot(0, 1).ry(1,
 FreeParameter("beta"))

# Create circuit bindings with different parameters
circuit_binding = CircuitBinding(
```

```
    circuit=parameterize_circuit,
    input_sets={
            'alpha': (0.10, 0.11, 0.22, 0.34, 0.45),
            'beta': (1.01, 1.01, 1.03, 1.04, 1.04),
    })

# Creating the program set
program_set_1 = ProgramSet([
    circ1,
    circ2,
    circ3,
    circuit_binding,
])
```

该程序集包含四个独特的程序：circ1、circ2、circ3 和 circuit_binding。circuit_binding 程序使用五个不同的参数绑定运行，创建了五个可执行文件。其他三个无参数程序分别创建一个可执行文件。这会生成八个可执行文件，如下图所示。



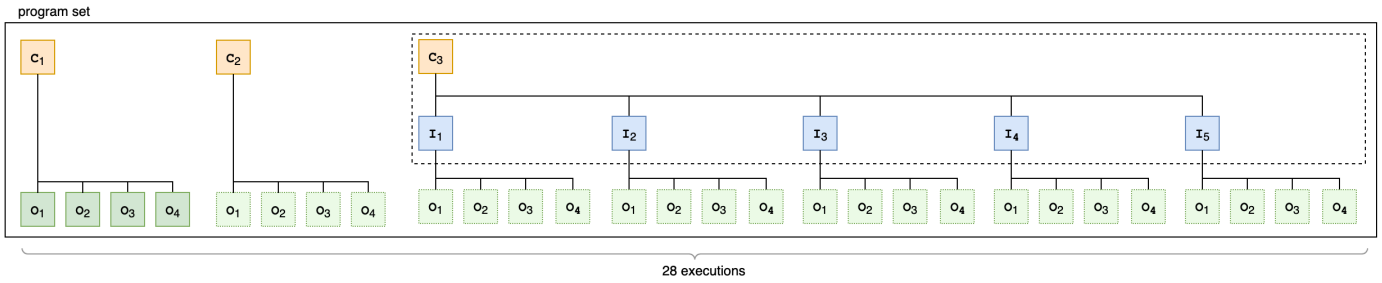

```
ProgramSet(
    [c1,
     c2,
     c3,
     CircuitBinding(c4, [i1, i2, i3, i4, i5])]
)
```

以下第二个代码示例演示了如何使用 product() 方法将同一组可观测值附加到程序集的每个可执行文件中。

```
from braket.circuits.observables import I, X, Y, Z

observables = [Z(0) @ Z(1), X(0) @ X(1), Z(0) @ X(1), X(0) @ Z(1)]

program_set_2 = ProgramSet.product(
    circuits=[circ1, circ2, circuit_binding],
    observables=observables
)
```

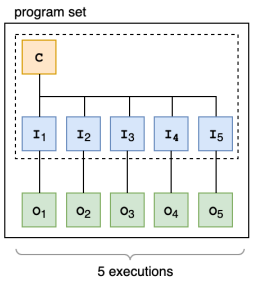
对于无参数程序，针对每个电路测量各个可观测值。对于参数化程序，针对每个输入集测量每个可观测值，如下图所示。

```
ProgramSet.product(
    circuits=[c1, c2, CircuitBinding(c3, [i1, i2, i3, i4, i5])],
    observables=[o1, o2, o3, o4]
)
```



下列第三个代码示例演示了如何使用 zip() 方法将单个可观测值与 ProgramSet 中的特定参数集配对。

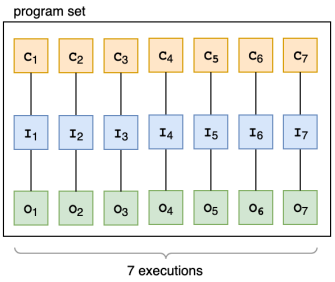
```
program_set_3 = ProgramSet.zip(
    circuits=circuit_binding,
    observables=observables + [Y(0) @ Y(1)]
)
```




```
ProgramSet.zip(
    circuits=CicruitBinding(
        circuit=c,
        input_sets=[i1, i2, i3, i4, i5]
    ),
    observables=[o1, o2, o3, o4, o5]
)
```

不选择 CircuitBinding()，您可以直接压缩带有电路和输入集列表的可观察值列表。

```
program_set_4 = ProgramSet.zip(
    circuits=[circ1, circ2, circ3],
    input_sets=[{}, {}, {}],
    observables=observables[:3]
)
```




```
ProgramSet.zip(
    circuits=[c1, c2, c3, c4, c5, c6, c7],
    input_sets=[i1, i2, i3, i4, i5, i6, i7],
    observables=[o1, o2, o3, o4, o5, o6, o7]
)
```

有关程序集的更多信息和示例，请参阅 amazon-braket-examples Github 中的程序集文件夹。

检查并运行设备上设置的程序

程序集的可执行文件数量等于其唯一参数绑定的电路的数量。使用以下代码示例计算电路可执行文件总数和总拍摄次数。

```
# Number of shots per executable
shots = 10
num_executables = program_set_1.total_executables

# Calculate total number of shots across all executables
total_num_shots = shots*num_executables
```

> **Note**
>
> 使用程序集，您需要根据程序集中所有电路的总拍摄次数来支付每项任务的单次费用和每次拍摄的费用。

要运行程序集，请使用以下代码示例。

```
# Run the program set
task = device.run(
   program_set_1, shots=total_num_shots,
)
```

使用 Rigetti 设备时，当任务部分完成及部分排队时，您的程序集可能会保持 RUNNING 状态。为了更快地获得结果，请考虑将您的程序集作为混合作业提交。

分析结果

运行以下代码，分析、测量 ProgramSet 中可执行文件的结果。

```
# Get the results from a program set
result = task.result()

# Get the first executbable
first_program = result[0]
first_executable = first_program[0]

# Inspect the results of the first executable
measurements_from_first_executable = first_executable.measurements
print(measurements_from_first_executable)
```

部分测量

不测量量子电路中的所有量子比特，而是使用部分测量来测量单个量子比特或量子比特子集。

> **Note**
>
> 其他功能，如中间电路测量和前馈操作，可作为实验功能提供，请参阅访问 IQM 设备上的动态电路。

示例：测量量子比特的子集

以下代码示例演示了通过在贝尔态电路中仅测量量子比特 0 来进行部分测量。

```
from braket.devices import LocalSimulator
from braket.circuits import Circuit

# Use the local state vector simulator
device = LocalSimulator()

# Define an example bell circuit and measure qubit 0
circuit = Circuit().h(0).cnot(0, 1).measure(0)

# Run the circuit
task = device.run(circuit, shots=10)

# Get the results
result = task.result()
```

```
# Print the circuit and measured qubits
print(circuit)
print()
print("Measured qubits: ", result.measured_qubits)
```

手动 qubit 分配

当您在量子计算机上从 Rigetti 运行量子电路时，您可以选择使用手动 qubit 分配来控制将哪些 qubits 用于您的算法。Amazon Braket 控制台和 Amazon Braket SDK 可帮助您检查所选量子处理单元（QPU）设备的最新校准数据，以便您可以为实验选择最佳 qubits。

通过手动 qubit 分配，能够更准确地运行电路及调查各个 qubit 特性。研究人员和高级用户可以根据最新的设备校准数据优化其电路设计，从而获得更准确的结果。

以下示例演示了如何进行 qubits 显式分配。

```
# Import the device module
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
circ = Circuit().h(0).cnot(0, 7)  # Indices of actual qubits in the QPU

# Set up S3 bucket (where results are stored)
my_bucket = "amazon-braket-s3-demo-bucket" # The name of the bucket
my_prefix = "your-folder-name" # The name of the folder in the bucket
s3_location = (my_bucket, my_prefix)

my_task = device.run(circ, s3_location, shots=100, disable_qubit_rewiring=True)
```

有关更多信息，请参阅本笔记本上 GitHub的 Amazon Braket 示例：在 QPU 设备上分配量子比特。

逐字记录编译

当您在基于门的量子计算机上运行量子电路时，您可以指示编译器完全按照定义运行您的电路，而无需做出任何修改。使用逐字记录编译，您可以指定要么完全按照指定方式保留整个电路，要么仅保留其中的特定部分（仅受 Rigetti 支持）。在为硬件基准测试或错误缓解协议开发算法时，您需要能够选择精确指定硬件上运行的门和电路布局。通过逐字记录编译，您可以关闭某些优化步骤来直接控制编译过程，从而确保您的电路完全按照设计运行。

、、和Rigetti设备支持逐字编译 AQT IonQIQM，并且需要使用原生门。使用逐字记录编译时，建议检查设备的拓扑结构，以确保在连接的 qubits 上调用门且电路使用硬件支持的原生门。以下示例说明如何以编程方式访问设备支持的原生门列表。

```
device.properties.paradigm.nativeGateSet
```

对于 Rigetti，必须通过设置 `disableQubitRewiring=True` 与逐字记录编译一起使用，进而关闭 qubit 重新布线。如果在编译中使用逐字记录框时设置 `disableQubitRewiring=False`，则量子电路验证失败且电路无法运行。

如果为电路启用了逐字记录编译且在不支持逐字记录编译的 QPU 上运行，则会生成一个错误，指明不支持的操作导致任务失败。随着越来越多的量子硬件原生支持编译器功能，该功能将扩展到包括这些设备。使用以下代码查询时，支持逐字记录编译的设备将其列为支持的操作。

```
from braket.aws import AwsDevice
from braket.device_schema.device_action_properties import DeviceActionType
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
device.properties.action[DeviceActionType.OPENQASM].supportedPragmas
```

使用逐字记录编译不会产生额外成本。对于在 Braket QPU 设备、Notebook 实例和按需模拟器上执行的量子任务，您需要继续按照“Amazon Braket 定价”页面上指定的当前费率付费。有关更多信息，请参阅逐字记录编译示例 Notebook。

> **Note**
>
> 如果您使用 OpenQasm 为AQT和IonQ设备编写电路，并且希望将电路直接映射到物理量子比特，则需要使用，因`#pragma braket verbatim`为 OpenQasm 会忽略`disableQubitRewiring`标志。

噪声模拟

要实例化本地噪声模拟器，您可以按如下方式更改后端。

```
# Import the device module
from braket.aws import AwsDevice

device = LocalSimulator(backend="braket_dm")
```

您可以通过两种方式构造噪声电路：

1. 自下而上地构造嘈杂电路。

2. 采用现有的无噪声电路，并在整个过程中注入噪声。

以下示例显示了使用具有去极化噪声和自定义 Kraus 通道的基本电路的方法。

```
import scipy.stats
import numpy as np

# Bottom up approach
# Apply depolarizing noise to qubit 0 with probability of 0.1
circ = Circuit().x(0).x(1).depolarizing(0, probability=0.1)

# Create an arbitrary 2-qubit Kraus channel
E0 = scipy.stats.unitary_group.rvs(4) * np.sqrt(0.8)
E1 = scipy.stats.unitary_group.rvs(4) * np.sqrt(0.2)
K = [E0, E1]

# Apply a two-qubit Kraus channel to qubits 0 and 2
circ = circ.kraus([0, 2], K)
```

```
from braket.circuits import Noise

# Inject noise approach
# Define phase damping noise
noise = Noise.PhaseDamping(gamma=0.1)
# The noise channel is applied to all the X gates in the circuit
circ = Circuit().x(0).y(1).cnot(0, 2).x(1).z(2)
circ_noise = circ.copy()
circ_noise.apply_gate_noise(noise, target_gates=Gate.X)
```

运行电路的用户体验与之前相同，如以下两个示例所示。

示例 1

```
task = device.run(circ, shots=100)
```

Or

示例 2

```
task = device.run(circ_noise, shots=100)
```

有关更多示例，请参阅 Braket 入门噪声模拟器示例

检查电路

Amazon Braket 中的量子电路有一个名为 Moments 的“伪时间”概念。每个 qubit 每 Moment 只能体验一个门。Moments 的目的是使电路及其门更易于寻址，并提供时间结构。

> **Note**
>
> 时刻通常不对应于 QPU 上执行门的实时。

电路深度由该电路中的总时刻数给出。您可以查看调用 circuit.depth 方法的电路深度，如以下示例所示。

```
from braket.circuits import Circuit

# Define a circuit with parametrized gates
circ = Circuit().rx(0, 0.15).ry(1, 0.2).cnot(0, 2).zz(1, 3, 0.15).x(0)
print(circ)
print('Total circuit depth:', circ.depth)
```

```
T  : #     0      #        1         #  2  #
      ############                    #####
q0 : ## Rx(0.15) ###################### X ##
      ############   #                #####
      ############   #   ############
q1 : ## Ry(0.20) ######### ZZ(0.15) ########
      ############   #   ############
                  #####      #
q2 : ############### X ######################
                  #####      #
                        ############
q3 : ###################### ZZ(0.15) ########
                        ############
T  : #     0      #        1         #  2  #
Total circuit depth: 3
```

以上电路的总电路深度为 3（显示为时刻 0、1 和 2）。您可以查看每个时刻的门操作情况。

Moments 函数作为键值对的字典。

- 键是 MomentsKey()，它包含伪时间和 qubit 信息。
- 该值的分配类型为 Instructions()。

```
moments = circ.moments
for key, value in moments.items():
    print(key)
    print(value, "\n")
```

```
MomentsKey(time=0, qubits=QubitSet([Qubit(0)]), moment_type=<MomentType.GATE: 'gate'>,
 noise_index=0, subindex=0)
Instruction('operator': Rx('angle': 0.15, 'qubit_count': 1), 'target':
 QubitSet([Qubit(0)]), 'control': QubitSet([]), 'control_state': (), 'power': 1)

MomentsKey(time=0, qubits=QubitSet([Qubit(1)]), moment_type=<MomentType.GATE: 'gate'>,
 noise_index=0, subindex=0)
Instruction('operator': Ry('angle': 0.2, 'qubit_count': 1), 'target':
 QubitSet([Qubit(1)]), 'control': QubitSet([]), 'control_state': (), 'power': 1)

MomentsKey(time=1, qubits=QubitSet([Qubit(0), Qubit(2)]), moment_type=<MomentType.GATE:
 'gate'>, noise_index=0, subindex=0)
Instruction('operator': CNot('qubit_count': 2), 'target': QubitSet([Qubit(0),
 Qubit(2)]), 'control': QubitSet([]), 'control_state': (), 'power': 1)

MomentsKey(time=1, qubits=QubitSet([Qubit(1), Qubit(3)]), moment_type=<MomentType.GATE:
 'gate'>, noise_index=0, subindex=0)
Instruction('operator': ZZ('angle': 0.15, 'qubit_count': 2), 'target':
 QubitSet([Qubit(1), Qubit(3)]), 'control': QubitSet([]), 'control_state': (), 'power':
 1)

MomentsKey(time=2, qubits=QubitSet([Qubit(0)]), moment_type=<MomentType.GATE: 'gate'>,
 noise_index=0, subindex=0)
Instruction('operator': X('qubit_count': 1), 'target': QubitSet([Qubit(0)]), 'control':
 QubitSet([]), 'control_state': (), 'power': 1)
```

您也可以通过 Moments 为电路添加门。

```
from braket.circuits import Instruction, Gate

new_circ = Circuit()
instructions = [Instruction(Gate.S(), 0),
                Instruction(Gate.CZ(), [1, 0]),
```

```
                Instruction(Gate.H(), 1)
                ]

new_circ.moments.add(instructions)
print(new_circ)
```

```
T  : #  0  #  1  #  2  #
      ##### #####
q0 : ## S ### Z ########
      ##### #####
               #   #####
q1 : ############## H ##
                    #####
T  : #  0  #  1  #  2  #
```

结果类型列表

使用测量电路时，Amazon Braket 可以使用 `ResultType` 返回不同类型的结果。电路可以返回以下类型的结果。

- `AdjointGradient` 返回所提供的可观测值的期望值的梯度（向量导数）。该可观测值使用伴随微分法根据指定参数作用于所提供的目标。只有当 shots=0 时才能使用此方法。
- `Amplitude` 返回输出波函数中指定量子态的振幅。它仅在 SV1 和本地模拟器上可用。
- `Expectation` 返回给定可观测值的期望值，该值可以通过本章后面介绍的 `Observable` 类来指定。必须指定 qubits 用于测量可观测值的目标，并且指定目标的数量必须等于可观测值所针对的 qubits 数量。如果未指定目标，则可观测值必须仅在 1 个 qubit 上运行，且并行应用于所有 qubits。
- `Probability` 返回测量计算基态的概率。如果未指定目标，则 `Probability` 返回测量所有基态的概率。如果指定了目标，则仅返回指定 qubits 上的基向量的边际概率。托管模拟器， QPUs 最大量子比特限制为 15 个，本地模拟器仅限于系统的内存大小。
- `Reduced density matrix` 返回 qubits 系统中指定目标 qubits 子系统的密度矩阵。为了限制此结果类型的大小，Braket 将 qubits 目标的最大数量限制为 8。
- `StateVector` 返回完整的状态向量。它可在本地模拟器上使用。
- `Sample` 返回指定目标 qubit 集和可观测值的测量计数。如果未指定目标，则可观测值必须仅在 1 个 qubit 上运行，且并行应用于所有 qubits。如果指定了目标，则仅返回指定 qubits 上的基向量的边际概率。

- `Variance` 返回指定目标 qubit 集的方差 (`mean([x-mean(x)]^2)`)，并作为请求的结果类型进行观察。如果未指定目标，则可观测值必须仅在 1 个 qubit 上运行，且并行应用于所有 qubits。否则，指定的目标数量必须等于可以应用可观测值的 qubits 的数量。

不同提供程序支持的结果类型：

	本地 SIM	SV1	DM1	TN1	AQT	IonQ	IQM	Rigetti
伴随梯度	N	Y	N	N	N	N	N	N
振幅	Y	Y	N	N	N	N	N	N
期望	Y	Y	Y	Y	Y	Y	Y	Y
概率	Y	Y	Y	N	Y	Y	Y	Y
低密度矩阵	Y	N	Y	N	N	N	N	N
状态向量	Y	N	N	N	N	N	N	N
样本	Y	Y	Y	Y	Y	Y	Y	Y
方差	Y	Y	Y	Y	Y	Y	Y	Y

您可以通过检查设备属性来查看受支持的结果类型，如以下示例所示。

```
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")

# Print the result types supported by this device
for iter in
 device.properties.action['braket.ir.openqasm.program'].supportedResultTypes:
    print(iter)
```

```
name='Sample' observables=['x', 'y', 'z', 'h', 'i'] minShots=10 maxShots=50000
name='Expectation' observables=['x', 'y', 'z', 'h', 'i'] minShots=10 maxShots=50000
name='Variance' observables=['x', 'y', 'z', 'h', 'i'] minShots=10 maxShots=50000
name='Probability' observables=None minShots=10 maxShots=50000
```

要调用 `ResultType`，请将其追加到电路中，如以下示例所示。

```
from braket.circuits import Circuit, Observable

circ = Circuit().h(0).cnot(0, 1).amplitude(state=["01", "10"])
circ.probability(target=[0, 1])
circ.probability(target=0)
circ.expectation(observable=Observable.Z(), target=0)
circ.sample(observable=Observable.X(), target=0)
circ.state_vector()
circ.variance(observable=Observable.Z(), target=0)

# Print one of the result types assigned to the circuit
print(circ.result_types[0])
```

> **Note**
>
> 不同的量子设备提供不同格式的结果。例如，Rigetti 设备返回测量值，而 IonQ 设备则提供概率。Amazon Braket SDK 为所有结果提供了一个测量属性。但是，对于返回概率的设备，这些测量值是事后计算的，且基于概率，无法进行逐次测量。要确定结果是否要进行后期计算，请检查结果对象上的 `measurements_copied_from_device`。此操作在 Amazon Braket 软件开发工具包 GitHub 存储库中的 gate_model_quantum_task_result.py 文件中有详细介绍。

可观测值

Amazon Braket 的 `Observable` 类可测量特定的可观测值。

您只能将一个唯一非身份可观测值应用于每个 qubit。如果您将两个或多个不同的非身份可观测值指定为相同的 qubit，则会发生错误。为此，张量乘积的每个因子都算作一个单独的可观测值。这意味着您可以将多个张量乘积放在同一个 qubit 上，前提是作用于 qubit 的因子保持不变。

可以扩展可观测值并添加其他可观测值（无论是否扩展）。这将创建可在 `AdjointGradient` 结果类型中使用的 Sum。

Observable 类包括以下可观测值。

```
import numpy as np

Observable.I()
Observable.H()
Observable.X()
Observable.Y()
Observable.Z()

# Get the eigenvalues of the observable
print("Eigenvalue:", Observable.H().eigenvalues)
# Or rotate the basis to be computational basis
print("Basis rotation gates:", Observable.H().basis_rotation_gates)

# Get the tensor product of the observable for the multi-qubit case
tensor_product = Observable.Y() @ Observable.Z()
# View the matrix form of an observable by using
print("The matrix form of the observable:\n", Observable.Z().to_matrix())
print("The matrix form of the tensor product:\n", tensor_product.to_matrix())

# Factorize an observable in the tensor form
print("Factorize an observable:", tensor_product.factors)

# Self-define observables, given it is a Hermitian
print("Self-defined Hermitian:", Observable.Hermitian(matrix=np.array([[0, 1], [1,
 0]])))

print("Sum of other (scaled) observables:", 2.0 * Observable.X() @ Observable.X() + 4.0
 * Observable.Z() @ Observable.Z())
```

```
Eigenvalue: [ 1. -1.]
Basis rotation gates: (Ry('angle': -0.7853981633974483, 'qubit_count': 1),)
The matrix form of the observable:
 [[ 1.+0.j  0.+0.j]
 [ 0.+0.j -1.+0.j]]
The matrix form of the tensor product:
 [[ 0.+0.j  0.+0.j  0.-1.j  0.+0.j]
 [ 0.+0.j -0.+0.j  0.+0.j  0.+1.j]
 [ 0.+1.j  0.+0.j  0.+0.j  0.+0.j]
 [ 0.+0.j  0.-1.j  0.+0.j -0.+0.j]]
Factorize an observable: (Y('qubit_count': 1), Z('qubit_count': 1))
```

```
Self-defined Hermitian: Hermitian('qubit_count': 1, 'matrix': [[0.+0.j 1.+0.j], [1.+0.j
 0.+0.j]])
Sum of other (scaled) observables: Sum(TensorProduct(X('qubit_count': 1),
 X('qubit_count': 1)), TensorProduct(Z('qubit_count': 1), Z('qubit_count': 1)))
```

参数

电路可以包含自由参数。这些自由参数只需要构造一次即可运行多次，并且可用于计算梯度。

每个自由参数都使用字符串编码的名称，该名称用于：

- 设置参数值
- 确定要使用的参数

```
from braket.circuits import Circuit, FreeParameter, observables
from braket.parametric import FreeParameter

theta = FreeParameter("theta")
phi = FreeParameter("phi")
circ = Circuit().h(0).rx(0, phi).ry(0, phi).cnot(0, 1).xx(0, 1, theta)
```

伴随梯度

SV1 设备计算可观测期望值（包括多项哈密顿量）的伴随梯度。要区分参数，请指定其名称（字符串格式）或通过直接引用来指定。

```
from braket.aws import AwsDevice
from braket.devices import Devices

device = AwsDevice(Devices.Amazon.SV1)

circ.adjoint_gradient(observable=3 * Observable.Z(0) @ Observable.Z(1) - 0.5 *
 observables.X(0), parameters = ["phi", theta])
```

如果将固定参数值作为参数传递给参数化电路，自由参数将会被移除。由于自由参数已不存在，用 `AdjointGradient` 运行此电路会产生错误。以下代码示例演示了正确及错误的用法：

```
# Will error, as no free parameters will be present
#device.run(circ(0.2), shots=0)
```

```
# Will succeed
device.run(circ, shots=0, inputs={'phi': 0.2, 'theta': 0.2})
```

获取专家建议

直接在 Braket 管理控制台中与量子计算专家联系，获取有关工作负载的更多指导。

要通过 Braket Direct 浏览专家建议选项，请打开 Braket 控制台，在左侧窗格中选择 Braket Direct，然后导航到“专家建议”部分。以下专家建议选项也可用：

- Braket 办公时间：Braket 办公时间为 1:1 会议，先到先得，每月举行一次。每个可用的办公时间段均为 30 分钟，且免费。与Braket专家交谈可以探索 use-case-to-device拟合度，确定最适合使用Braket进行算法的选项，并获得有关如何使用某些Braket功能的建议，例如Amazon Braket混合任务、Braket Pulse或模拟哈密尔顿模拟，从而帮助您更快地从构思到执行。
  - 要注册 Braket 办公时间，请选择“注册”并填写联系信息、工作负载详情和所需讨论的主题。
  - 您将通过电子邮件收到下一个可用时段的日历邀请。

  > **Note**
  >
  > 对于紧急问题或快速故障排除问题，我们建议您联系 AWS 支持。对于非紧急问题，您也可以使用 AWS re:Post 论坛或 Quantum Computing Stack Exchange，在那里您可以浏览之前回答过的问题并提出新问题。

- 量子硬件提供商提供的产品：IonQ、QuEra 和 Rigetti 分别通过 AWS Marketplace提供专业服务产品。
  - 要浏览他们的产品，请选择 Connect 并浏览他们的列表。
  - 要详细了解上提供的专业服务 AWS Marketplace，请参阅专业服务产品。
- Amazon Quantum Solutions Lab（QSL）：QSL 是一个合作研究和专业服务团队，由量子计算专家组成，可以帮助您有效地探索量子计算并评测该技术的当前性能。
  - 要联系 QSL，请选择 Connect，然后填写联系信息和使用案例详细信息。
  - QSL 团队将通过电子邮件与您联系，告知后续步骤。

使用 OpenQASM 3.0 运行您的电路

Amazon Braket 现在支持适用于基于门的量子设备和模拟器的 OpenQASM 3.0。本用户指南提供了有关 Braket 支持的 OpenQASM 3.0 子集的信息。Braket 客户现在可以选择使用 SDK 提交 Braket 电

路，也可以使用 Amazon Braket API 和 Amazon Braket Python SDK 直接向所有基于门的设备提供 OpenQASM 3.0 字符型。

本指南中的主题将引导您了解如何完成以下量子任务的各种示例。

- 在不同的 Braket 设备上创建和提交 OpenQASM 量子任务
- 访问受支持的操作和结果类型
- 使用 OpenQASM 模拟噪声
- 在 OpenQASM 中使用逐字记录编译
- 排查 OpenQASM 问题

本指南还介绍了某些硬件特有的功能，这些功能可以通过 Braket 上的 OpenQASM 3.0 实现，并提供了更多资源的链接。

本节内容：

- 什么是 OpenQASM 3.0？
- 何时使用 OpenQASM 3.0
- OpenQASM 3.0 的工作方式
- 先决条件
- Braket 支持哪些 OpenQASM 功能？
- 创建并提交示例 OpenQASM 3.0 量子任务
- 在不同的 Braket 设备上支持 OpenQASM
- 使用 OpenQASM 3.0 模拟噪声
- Qubit 使用 OpenQASM 3.0 重新布线
- 使用 OpenQASM 3.0 进行逐字记录编译
- Braket 控制台
- 其他资源
- 使用 OpenQASM 3.0 计算梯度
- 使用 OpenQASM 3.0 测量特定的量子比特

什么是 OpenQASM 3.0？

开放量子汇编语言（OpenQASM）是量子指令的中间表示形式。OpenQASM 是一个开源框架，广泛用于规范基于门的设备的量子程序。使用 OpenQASM，用户可以对构成量子计算基块的量子门和测量操作进行编程。许多量子编程库都使用先前版本的 OpenQASM (2.0) 来描述基本程序。

新版本的 OpenQASM (3.0) 扩展了之前的版本，增加了更多功能，如脉冲电平控制、门定时和经典控制流，以弥合最终用户界面和硬件描述语言之间的差距。当前版本 3.0 的详细信息和规格可在 GitHub OpenQasm 3.x 实时规格中找到。OpenQasm 的未来发展由 OpenQasm 3.0 技术指导委员会管理，该委员会与 IBM、微软和因斯布鲁克大学一起 AWS 是该委员会的成员。

何时使用 OpenQASM 3.0

OpenQASM 提供了一个富有表现力的框架，可通过非特定架构的低级控件来指定量子程序，因而非常适合作为多个基于门的设备的表示形式。Braket 对 OpenQASM 的支持进一步推动了其作为开发基于门的量子算法的一致方法的采用，从而减少了用户在多个框架中学习和维护库的需求。

如果您在 OpenQASM 3.0 中已有程序库，则可以对其进行调整，使其与 Braket 配合使用，而不必完全重写这些电路。研究人员和开发人员还应受益于越来越多的支持 OpenQASM 算法开发的可用第三方库。

OpenQASM 3.0 的工作方式

Braket 对 OpenQASM 3.0 的支持提供了与当前中间表示法相同的功能。这意味着，您今天在硬件设备和使用 Braket 的按需模拟器上能做的任何事情，都可以使用 Braket API 在 OpenQASM 上处理。您可以通过直接向所有基于门的设备提供 OpenQASM 字符串来运行 OpenQASM 3.0 程序，其方式类似于当前向 Braket 上的设备提供电路。Braket 用户还可以集成支持 OpenQASM 3.0 的第三方库。本指南的其他部分详细介绍了如何开发用于 Braket 的 OpenQASM 表示形式。

先决条件

要在 Amazon Braket 上使用 OpenQASM 3.0，您必须拥有 Amazon Braket Python 架构的 1.8.0 版本和 Amazon Braket Python SDK 的 1.17.0 版本或更高版本。

如果您是首次接触 Amazon Braket 的用户，则需要启用 Amazon Braket。有关说明，请参阅启用 Amazon Braket。

Braket 支持哪些 OpenQASM 功能？

下节列出了 Braket 支持的 OpenQASM 3.0 数据类型、语句和编译指令。

本节内容：

受支持的 OpenQASM 数据类型

Amazon Braket 支持以下 OpenQASM 数据类型：

- 非负整数用于（虚拟和物理）量子比特索引：
  - `cnot q[0], q[1];`
  - `h $0;`
- 浮点数或常数可用于门旋转角度：
  - `rx(-0.314) $0;`
  - `rx(pi/4) $0;`

> **Note**
>
> pi 是 OpenQASM 中的内置常量，不能用作参数名称。

- 在用于定义一般哈密特量可观测值的结果类型编译指示和单一编译指示中，允许使用复数数组（虚数部分使用 OpenQASM `im` 表示法）：
  - `#pragma braket unitary [[0, -1im], [1im, 0]] q[0]`
  - `#pragma braket result expectation hermitian([[0, -1im], [1im, 0]]) q[0]`

受支持的 OpenQASM 语句

Amazon Braket 支持以下 OpenQASM 语句。

- `Header: OPENQASM 3;`
- 经典位声明：

  - `bit b1;` (equivalently, `creg b1;`)
  - `bit[10] b2;` (equivalently, `creg b2[10];`)
- 量子比特声明：
  - `qubit b1;` (equivalently, `qreg b1;`)
  - `qubit[10] b2;` (equivalently, `qreg b2[10];`)
- 在数组内建立索引：`q[0]`
- 输入：`input float alpha;`
- 物理 qubits 的规格：`$0`
- 设备上受支持的门和操作：
  - `h $0;`
  - `iswap q[0], q[1];`

> **Note**
>
> 设备支持的门可以在 OpenQASM 操作的设备属性中找到；使用这些门不需要任何门定义。

- 逐字记录表声明。目前，我们不支持方框持续时间表示法。原生门和物理 qubits 必须放在逐字记录框中。

```
#pragma braket verbatim
box{
    rx(0.314) $0;
}
```

- 在 qubits 或整个 qubit 寄存器上进行测量和测量分配。
  - `measure $0;`
  - `measure q;`
  - `measure q[0];`
  - `b = measure q;`
  - `measure q # b;`
- Barrier 语句通过防止跨屏障边界的门重新排序和优化，提供对电路编译和执行的明确控制。它们还在执行期间强制执行严格的时间顺序，确保屏障之前的所有操作在后续操作开始之前完成。

- `barrier;`
- `barrier q[0], q[1];`
- `barrier $3, $6;`

Braket OpenQASM 编译指示

Amazon Braket 支持下列 OpenQASM 编译指示说明。

- 噪声编译指示
  - `#pragma braket noise bit_flip(0.2) q[0]`
  - `#pragma braket noise phase_flip(0.1) q[0]`
  - `#pragma braket noise pauli_channel`
- 逐字编译指示
  - `#pragma braket verbatim`
- 结果类型编译指示
  - 基数不变结果类型：
    - 状态向量：`#pragma braket result state_vector`
    - 密度矩阵：`#pragma braket result density_matrix`
  - 梯度计算编译指示：
    - 伴随渐变：`#pragma braket result adjoint_gradient expectation(2.2 * x[0] @ x[1]) all`
  - Z 基准结果类型：
    - 振幅：`#pragma braket result amplitude "01"`
    - 概率：`#pragma braket result probability q[0], q[1]`
  - 基础轮换结果类型
    - 期望：`#pragma braket result expectation x(q[0]) @ y([q1])`
    - 方差：`#pragma braket result variance hermitian([[0, -1im], [1im, 0]]) $0`
    - 示例：`#pragma braket result sample h($1)`

> **Note**
>
> 由于 OpenQASM 3.0 向后兼容 OpenQASM 2.0，因而使用 2.0 编写的程序可以在 Braket 上运行。但是，Braket 支持的 OpenQASM 3.0 功能确实存在一些细微的语法差异，如 `qreg` vs `creg` 和 `qubit` vs `bit`。测量句法也有差异，需要用正确的语法来支持这些语法。

本地模拟器上对 OpenQASM 的高级功能支持

`LocalSimulator` 支持高级 OpenQASM 功能，这些功能不是作为 Braket 的 QPU 模拟器或按需模拟器的一部分提供的。在 `LocalSimulator` 中，仅支持下列功能：

- 门修改器
- OpenQASM 内置门
- 经典变量
- 经典运算
- 定制门
- 经典控制
- QASM 文件
- 子程序

有关每项高级功能的示例，请参阅此样本 Notebook。有关完整的 OpenQASM 规范，请访问 OpenQASM 网站。

支持的操作和语法 OpenPulse

支持 OpenPulse 的数据类型

Cal 块：

```
cal {
    ...
}
```

Defcal 块：

```
// 1 qubit
```

```
defcal x $0 {
...
}

// 1 qubit w. input parameters as constants
defcal my_rx(pi) $0 {
...
}

// 1 qubit w. input parameters as free parameters
defcal my_rz(angle theta) $0 {
...
}

// 2 qubit (above gate args are also valid)
defcal cz $1, $0 {
...
}
```

帧:

```
frame my_frame = newframe(port_0, 4.5e9, 0.0);
```

波形：

```
// prebuilt
waveform my_waveform_1 = constant(1e-6, 1.0);

//arbitrary
waveform my_waveform_2 = {0.1 + 0.1im, 0.1 + 0.1im, 0.1, 0.1};
```

自定义门校准示例：

```
cal {
    waveform wf1 = constant(1e-6, 0.25);
}

defcal my_x $0 {
   play(wf1, q0_rf_frame);
}

defcal my_cz $1, $0 {
```

```
    barrier q0_q1_cz_frame, q0_rf_frame;
    play(q0_q1_cz_frame, wf1);
    delay[300ns] q0_rf_frame
    shift_phase(q0_rf_frame, 4.366186381749424);
    delay[300ns] q0_rf_frame;
    shift_phase(q0_rf_frame.phase, 5.916747563126659);
    barrier q0_q1_cz_frame, q0_rf_frame;
    shift_phase(q0_q1_cz_frame, 2.183093190874712);
}

bit[2] ro;
my_x $0;
my_cz $1,$0;
c[0] = measure $0;
```

任意脉冲示例：

```
bit[2] ro;
cal {
    waveform wf1 = {0.1 + 0.1im, 0.1 + 0.1im, 0.1, 0.1};
    barrier q0_drive, q0_q1_cross_resonance;
    play(q0_q1_cross_resonance, wf1);
    delay[300ns] q0_drive;
    shift_phase(q0_drive, 4.366186381749424);
    delay[300dt] q0_drive;
   barrier q0_drive, q0_q1_cross_resonance;
   play(q0_q1_cross_resonance, wf1);
    ro[0] = capture_v0(r0_measure);
    ro[1] = capture_v0(r1_measure);
}
```

创建并提交示例 OpenQASM 3.0 量子任务

你可以使用 Amazon Braket Python SDK、Boto3 或，向亚马逊 Braket AWS CLI 设备提交 OpenQasm 3.0 量子任务。

本节内容：

- 一个 OpenQASM 3.0 程序示例
- 使用 Python SDK 创建 OpenQASM 3.0 量子任务
- 使用 Boto3 创建 OpenQASM 3.0 量子任务
- 使用创建 OpenQasm 3.0 任务 AWS CLI

一个 OpenQASM 3.0 程序示例

要创建 OpenQASM 3.0 任务，您可以从一个基本的 OpenQASM 3.0 程序 (ghz.qasm) 开始，该程序准备 GHZ 状态，如以下示例所示。

```
// ghz.qasm
// Prepare a GHZ state
OPENQASM 3;

qubit[3] q;
bit[3] c;

h q[0];
cnot q[0], q[1];
cnot q[1], q[2];

c = measure q;
```

使用 Python SDK 创建 OpenQASM 3.0 量子任务

您可以使用以下代码及 Amazon Braket Python SDK 将此程序提交到 Amazon Braket 设备。请务必将 Amazon S3 存储桶位置“amzn-s3-demo-bucket”替换为您自己的 Amazon S3 存储桶名称。

```
with open("ghz.qasm", "r") as ghz:
    ghz_qasm_string = ghz.read()

# Import the device module
from braket.aws import AwsDevice
# Choose the Rigetti device
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
from braket.ir.openqasm import Program

program = Program(source=ghz_qasm_string)
my_task = device.run(program)

# Specify an optional s3 bucket location and number of shots
s3_location = ("amzn-s3-demo-bucket", "openqasm-tasks")
my_task = device.run(
    program,
    s3_location,
    shots=100,
)
```

使用 Boto3 创建 OpenQASM 3.0 量子任务

您还可以使用 AWS 适用于 Braket 的 Python SDK（Boto3），利用 OpenQASM 3.0 字符串创建量子任务，如以下示例所示。以下代码片段引用了 ghz.qasm，它准备了 GHZ 状态，如上所示。

```
import boto3
import json

my_bucket = "amzn-s3-demo-bucket"
s3_prefix = "openqasm-tasks"

with open("ghz.qasm") as f:
    source = f.read()

action = {
    "braketSchemaHeader": {
        "name": "braket.ir.openqasm.program",
        "version": "1"
    },
    "source": source
}
device_parameters = {}
device_arn = "arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3"
shots = 100

braket_client = boto3.client('braket', region_name='us-west-1')
rsp = braket_client.create_quantum_task(
    action=json.dumps(
        action
    ),
    deviceParameters=json.dumps(
        device_parameters
    ),
    deviceArn=device_arn,
    shots=shots,
    outputS3Bucket=my_bucket,
    outputS3KeyPrefix=s3_prefix,
)
```

使用创建 OpenQasm 3.0 任务 AWS CLI

AWS Command Line Interface（CLI）也可用于提交 OpenQASM 3.0 程序，如以下示例所示。

```
aws braket create-quantum-task \
    --region "us-west-1" \
    --device-arn "arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3" \
    --shots 100 \
    --output-s3-bucket "amzn-s3-demo-bucket" \
    --output-s3-key-prefix "openqasm-tasks" \
    --action '{
        "braketSchemaHeader": {
            "name": "braket.ir.openqasm.program",
            "version": "1"
        },
        "source": $(cat ghz.qasm)
    }'
```

在不同的 Braket 设备上支持 OpenQASM

对于支持 OpenQASM 3.0 的设备，`action` 字段支持通过 `GetDevice` 响应执行新操作，如下列 Rigetti 和 IonQ 设备示例所示。

```
//OpenQASM as available with the Rigetti device capabilities
{
    "braketSchemaHeader": {
        "name": "braket.device_schema.rigetti.rigetti_device_capabilities",
        "version": "1"
    },
    "service": {...},
    "action": {
        "braket.ir.jaqcd.program": {...},
        "braket.ir.openqasm.program": {
            "actionType": "braket.ir.openqasm.program",
            "version": [
                "1"
            ],
            ….
        }
    }
}

//OpenQASM as available with the IonQ device capabilities
{
    "braketSchemaHeader": {
        "name": "braket.device_schema.ionq.ionq_device_capabilities",
```

```
        "version": "1"
    },
    "service": {...},
    "action": {
        "braket.ir.jaqcd.program": {...},
        "braket.ir.openqasm.program": {
            "actionType": "braket.ir.openqasm.program",
            "version": [
                "1"
            ],
            ….
        }
    }
}
```

对于支持脉冲控制的设备，pulse 字段显示在 GetDevice 响应中。以下示例显示了 Rigetti 设备的 pulse 字段。

```
// Rigetti
{
  "pulse": {
    "braketSchemaHeader": {
      "name": "braket.device_schema.pulse.pulse_device_action_properties",
      "version": "1"
    },
    "supportedQhpTemplateWaveforms": {
      "constant": {
        "functionName": "constant",
        "arguments": [
          {
            "name": "length",
            "type": "float",
            "optional": false
          },
          {
            "name": "iq",
            "type": "complex",
            "optional": false
          }
        ]
      },
      ...
    },
```

```
    "ports": {
      "q0_ff": {
        "portId": "q0_ff",
        "direction": "tx",
        "portType": "ff",
        "dt": 1e-9,
        "centerFrequencies": [
          375000000
        ]
      },
      ...
    },
    "supportedFunctions": {
      "shift_phase": {
        "functionName": "shift_phase",
        "arguments": [
          {
            "name": "frame",
            "type": "frame",
            "optional": false
          },
          {
            "name": "phase",
            "type": "float",
            "optional": false
          }
        ]
      },
     ...
    },
    "frames": {
      "q0_q1_cphase_frame": {
        "frameId": "q0_q1_cphase_frame",
        "portId": "q0_ff",
        "frequency": 462475694.24460185,
        "centerFrequency": 375000000,
        "phase": 0,
        "associatedGate": "cphase",
        "qubitMappings": [
          0,
          1
        ]
      },
      ...
```

```
    },
    "supportsLocalPulseElements": false,
    "supportsDynamicFrames": false,
    "supportsNonNativeGatesWithPulses": false,
    "validationParameters": {
      "MAX_SCALE": 4,
      "MAX_AMPLITUDE": 1,
      "PERMITTED_FREQUENCY_DIFFERENCE": 400000000
    }
  }
}
```

前面的字段详细说明了以下内容：

端口：

描述了在 QPU 上声明的预制外部 (extern) 设备端口以及给定端口的相关属性。此结构中列出的所有端口都预先声明为用户提交的 OpenQASM 3.0 程序中的有效标识符。端口的其他属性包括：

- 端口 ID (portId)
  - 在 OpenQASM 3.0 中声明为标识符的端口名称。
- 方向 (direction)
  - 端口的方向。驱动端口传输脉冲（“tx”方向），而测量端口接收脉冲（“rx”方向）。
- 端口类型 (portType)
  - 此端口负责的操作类型（例如，驱动、捕获或 ff - fast-flux）。
- Dt (dt)
  - 表示给定端口上的单个采样时间步长，以秒为单位。
- 量子比特映射 (qubitMappings)
  - 与给定端口关联的量子比特。
- 中心频率 (centerFrequencies)
  - 端口上所有预先声明或用户定义的帧的相关中心频率列表。有关更多信息，请参阅“帧”。
- QHP 特定属性 () qhpSpecificProperties
  - 一张可选地图，详细介绍有关 QHP 特定端口的现有属性。

帧：

描述了在 QPU 上声明的预制外部帧以及与这些帧相关的属性。此结构中列出的所有帧都预先声明为用户提交的 `OpenQASM 3.0` 程序中的有效标识符。帧的其他属性包括：

- 帧编号 (frameId)
  - 在 OpenQASM 3.0 中声明为标识符的帧名称。
- 端口 ID (portId)
  - 帧的关联硬件端口。
- 频率 (frequency)
  - 帧的默认初始频率。
- 中心频率 (centerFrequency)
  - 帧频率带宽的中心。通常，只能将帧调整到中心频率周围的特定带宽。因此，频率调整应保持在中心频率的给定增量之内。您可以在验证参数中找到带宽值。
- 阶段 (phase)
  - 帧的默认初始阶段。
- 关联门 (associatedGate)
  - 与给定帧关联的门。
- 量子比特映射 (qubitMappings)
  - 与给定帧关联的量子比特。
- QHP 特定属性 () qhpSpecificProperties
  - 一张可选地图，详细说明有关 QHP 特定帧的现有属性。

SupportsDynamicFrames:

描述了帧是否可以通过 OpenPulse `newframe` 函数在 `cal` 或 `defcal` 块中声明。如果该值为 false，则只能在程序中使用帧结构中列出的帧。

SupportedFunctions:

除了给定函数的关联参数、参数类型和返回类型之外，还描述了设备支持的 OpenPulse 函数。要查看使用这些OpenPulse函数的示例，请参阅OpenPulse规范。目前，Braket 支持：

- shift_phase
  - 按指定值移动帧的相位
- set_phase

  - 将帧的相位设置为指定值
- swap_phases
  - 在两帧之间交换相位。
- shift_frequency
  - 按指定值移动帧的频率
- set_frequency
  - 将帧频设置为指定值
- play
  - 安排波形
- capture_v0
  - 将捕获帧上的值返回到位寄存器

SupportedQhpTemplateWaveforms:

描述了设备上可用的预先构造的波形函数以及相关的参数和类型。默认情况下，Braket Pulse 在所有设备上提供预先构造的波形例程，它们是：

Constant

$$Constant(t, \tau, iq) = iq$$

τ 是波形长度，iq 是一个复数。

```
def constant(length, iq)
```

Gaussian

$$Gaussian(t, \tau, \sigma, A = 1, ZaE = 0) =$$
$$\frac{A}{1 - ZaE * \exp\left(-\frac{1}{2}\left(\frac{\tau}{2\sigma}\right)^2\right)}\left[\exp\left(-\frac{1}{2}\left(\frac{t-\frac{\tau}{2}}{\sigma}\right)^2\right) - ZaE * \exp\left(-\frac{1}{2}\left(\frac{\tau}{2\sigma}\right)^2\right)\right]$$

τ 是波形长度，σ 是高斯宽度，A 是振幅。如果将 ZaE 设置为 True，则对 Gaussian 进行偏移和重新缩放，使其在波形的开头和结尾处都等于零，且最大达到 A。

```
def gaussian(length, sigma, amplitude=1, zero_at_edges=False)
```

DRAG Gaussian

$$DRAG\_Gaussian(t, \tau, \sigma, \beta, A = 1, ZaE = 0) =$$

$$\frac{A}{1 - ZaE * \exp\left(-\frac{1}{2}\left(\frac{\tau}{2\sigma}\right)^2\right)}\left(1 - i\beta\frac{t - \frac{\tau}{2}}{\sigma^2}\right)\left[\exp\left(-\frac{1}{2}\left(\frac{t - \frac{\tau}{2}}{\sigma}\right)^2\right) - ZaE * \exp\left(-\frac{1}{2}\left(\frac{\tau}{2\sigma}\right)^2\right)\right]$$

τ 是波形长度，σ 是高斯宽度，β 是自由参数，A 是振幅。如果将 ZaE 设置为 True，则对绝热门导数去除（DRAG）Gaussian 进行偏移和重新缩放，使其在波形的开头和结尾处都等于零，实数部分最大达到 A。有关阻力波形的更多信息，请参阅论文 Simple Pulses for Elimination of Leakage in Weakly Nonlinear Qubits。

```
def drag_gaussian(length, sigma, beta, amplitude=1, zero_at_edges=False)
```

Erf Square

$$\text{Erf\_Square}(t, L, W, \sigma, A = 1, ZaE = 0) =$$

$$A \times \frac{\text{erf}((t - t_1)/\sigma) + \text{erf}(-(t - t_2)/\sigma)}{2 \times \text{erf}(W/2\sigma))}$$

其中：L 是长度，W 是波形宽度，σ 定义了边缘上升和下降的速度，t_1=(L-W)/2 和 $t_2$2=(L+W)/2，A 是振幅。如果将 ZaE 设置为 True，则对 Gaussian 进行偏移和重新缩放，使其在波形的开头和结尾处都等于零，且最大达到 A。以下方程是波形的重新缩放版本。

$$\text{Erf\_Square}(..., ZaE = 1) = (a \times \text{Erf\_Square}(..., ZaE = 0) - bA)/(a - b)$$

其中：a=erf(W/2σ) 且 b=erf(-t_1/σ)/2+erf(t $_2$/σ)/2 。

```
def erf_square(length, width, sigma, amplitude=1, zero_at_edges=False)
```

SupportsLocalPulseElements:

描述了脉冲元素（如端口、帧和波形）是否可以在 defcal 块中进行本地定义。如果值为 false，则必须以 cal 块形式定义元素。

SupportsNonNativeGatesWithPulses:

描述了我们是否可以将非原生门与脉冲程序结合使用。例如，如果不先通过 defcal 为所使用的量子比特定义门，就不能像程序中的 H 门一样使用非原生门。您可以在“设备功能”下方找到原生门 nativeGateSet 键列表。

ValidationParameters:

描述了脉冲元件验证边界，包括：

- （任意及预先构造的）波形的最大扩展/最大振幅值
- 所提供中心频率的最大频率带宽（单位为赫兹）
- length/duration 以秒为单位的最小脉冲
- 以秒为单位的最 length/duration 大脉冲

OpenQASM 支持的操作、结果和结果类型

要了解每台设备支持哪些 OpenQASM 3.0 功能，可以参考设备功能输出 action 字段中的 braket.ir.openqasm.program 键。例如，以下是 Braket 状态向量模拟器 SV1 支持的操作和结果类型。

```
...
  "action": {
    "braket.ir.jaqcd.program": {
      ...
    },
 "braket.ir.openqasm.program": {
      "version": [
        "1.0"
      ],
      "actionType": "braket.ir.openqasm.program",
      "supportedOperations": [
        "ccnot",
        "cnot",
        "cphaseshift",
        "cphaseshift00",
        "cphaseshift01",
        "cphaseshift10",
        "cswap",
        "cy",
        "cz",
        "h",
        "i",
```

```
        "iswap",
        "pswap",
        "phaseshift",
        "rx",
        "ry",
        "rz",
        "s",
        "si",
        "swap",
        "t",
        "ti",
        "v",
        "vi",
        "x",
        "xx",
        "xy",
        "y",
        "yy",
        "z",
        "zz"
      ],
      "supportedPragmas": [
        "braket_unitary_matrix"
      ],
      "forbiddenPragmas": [],
      "maximumQubitArrays": 1,
      "maximumClassicalArrays": 1,
      "forbiddenArrayOperations": [
        "concatenation",
        "negativeIndex",
        "range",
        "rangeWithStep",
        "slicing",
        "selection"
      ],
      "requiresAllQubitsMeasurement": true,
      "supportsPhysicalQubits": false,
      "requiresContiguousQubitIndices": true,
      "disabledQubitRewiringSupported": false,
      "supportedResultTypes": [
        {
          "name": "Sample",
          "observables": [
            "x",
```

```
            "y",
            "z",
            "h",
            "i",
            "hermitian"
          ],
          "minShots": 1,
          "maxShots": 100000
        },
        {
          "name": "Expectation",
          "observables": [
            "x",
            "y",
            "z",
            "h",
            "i",
            "hermitian"
          ],
          "minShots": 0,
          "maxShots": 100000
        },
        {
          "name": "Variance",
          "observables": [
            "x",
            "y",
            "z",
            "h",
            "i",
            "hermitian"
          ],
          "minShots": 0,
          "maxShots": 100000
        },
        {
          "name": "Probability",
          "minShots": 1,
          "maxShots": 100000
        },
        {
          "name": "Amplitude",
          "minShots": 0,
          "maxShots": 0
```

```
        }
        {
          "name": "AdjointGradient",
          "minShots": 0,
          "maxShots": 0
        }
      ]
    }
  },
...
```

使用 OpenQASM 3.0 模拟噪声

要使用 Open 模拟噪声QASM3，可以使用 pragma 指令添加噪音运算符。例如，要模拟之前提供的 GHZ 程序的噪声版本，您可以提交以下 OpenQASM 程序。

```
// ghz.qasm
// Prepare a GHZ state
OPENQASM 3;

qubit[3] q;
bit[3] c;

h q[0];
#pragma braket noise depolarizing(0.75) q[0] cnot q[0], q[1];
#pragma braket noise depolarizing(0.75) q[0]
#pragma braket noise depolarizing(0.75) q[1] cnot q[1], q[2];
#pragma braket noise depolarizing(0.75) q[0]
#pragma braket noise depolarizing(0.75) q[1]

c = measure q;
```

以下列表给出了所有受支持的编译指示噪声运算符的规格。

```
#pragma braket noise bit_flip(<float in [0,1/2]>) <qubit>
#pragma braket noise phase_flip(<float in [0,1/2]>) <qubit>
#pragma braket noise pauli_channel(<float>, <float>, <float>)  <qubit>
#pragma braket noise depolarizing(<float in [0,3/4]>) <qubit>
#pragma braket noise two_qubit_depolarizing(<float in [0,15/16]>) <qubit>, <qubit>
#pragma braket noise two_qubit_dephasing(<float in [0,3/4]>) <qubit>, <qubit>
#pragma braket noise amplitude_damping(<float in [0,1]>) <qubit>
```

```
#pragma braket noise generalized_amplitude_damping(<float in [0,1]> <float in [0,1]>)
 <qubit>
#pragma braket noise phase_damping(<float in [0,1]>) <qubit>
#pragma braket noise kraus([[<complex m0_00>, ], ...], [[<complex m1_00>, ], ...], ...)
 <qubit>[, <qubit>]     // maximum of 2 qubits and maximum of 4 matrices for 1 qubit,
 16 for 2
```

Kraus 运算符

要生成 Kraus 运算符，可以遍历矩阵列表，将矩阵的每个元素打印为复杂表达式。

使用克劳斯运算符时，请记住以下几点：

- qubits 的数量不得超过 2。架构中的当前定义设定了此限制。
- 参数列表的长度必须是 8 的倍数。这意味着它必须仅由 2x2 矩阵组成。
- 总长度不超过 2^{2*num_qubits} 矩阵。这意味着，1 个 qubit 有 4 个矩阵，2 个 qubits 有 16 个矩阵。
- 所有提供的矩阵均为完全正迹线保持(CPTP）。
- Kraus 运算符及其转置共轭的乘积需要相加得出一个单位矩阵。

Qubit 使用 OpenQASM 3.0 重新布线

Amazon Braket Rigetti 支持设备上的 OpenQASM 中的物理 qubit 符号（要了解更多信息，请参阅本页面）。在将物理 qubits 与原生重新布线策略配合使用时，请确保 qubits 已连接到所选设备上。或者，如果改用 qubit 寄存器，则默认情况下，Rigetti 设备上会启用部分重新布线策略。

```
// ghz.qasm
// Prepare a GHZ state
OPENQASM 3;

h $0;
cnot $0, $1;
cnot $1, $2;

measure $0;
measure $1;
measure $2;
```

使用 OpenQASM 3.0 进行逐字记录编译

当您在 Rigetti 和 IonQ 等供应商提供的量子计算机上运行量子电路时，您可以指示编译器完全按照定义运行您的电路，而无需做出任何修改。此功能称为逐字记录编译。使用Rigetti设备，您可以精确地指定要保留的内容——要么是整个电路，要么仅保留其中的特定部分。如果仅保留电路的特定部分，需要在保留区域内使用原生门。目前，IonQ 仅支持整个电路的逐字记录编译，因此电路中的每条指令都需要放在逐字记录框中。

使用 OpenQASM，您可以围绕代码框明确指定逐字记录编译指示，然后该代码保持不变，不会被硬件的低级编译例程优化。以下代码示例演示了如何使用 `#pragma braket verbatim` 指令实现这一点。

```
OPENQASM 3;

bit[2] c;

#pragma braket verbatim
box{
    rx(0.314159) $0;
    rz(0.628318) $0, $1;
    cz $0, $1;
}

c[0] = measure $0;
c[1] = measure $1;
```

有关逐字编译过程的更多详细信息，包括示例和最佳实践，请参阅 github 存储库中提供的 Verbatim 编译示例笔记本。 amazon-braket-examples

Braket 控制台

OpenQASM 3.0 任务可用，可在 Amazon Braket 控制台中进行管理。在控制台上，您在 OpenQASM 3.0 中提交量子任务的体验与提交现有量子任务的体验相同。

其他资源

OpenQASM 在所有 Amazon Braket 区域中都可用。

有关在 Amazon Braket 上开始使用 OpenQasm 的笔记本示例，请参阅 Braket 教程。 GitHub

使用 OpenQASM 3.0 计算梯度

在 shots=0（精确）模式下运行时，Amazon Braket 支持在按需模拟器和本地模拟器上计算梯度。通过使用伴随微分法，可以实现这一点。要指定所计算的梯度，可以提供相应的编译指示，如以下示例中的代码所示。

```
OPENQASM 3.0;
input float alpha;

bit[2] b;
qubit[2] q;

h q[0];
h q[1];
rx(alpha) q[0];
rx(alpha) q[1];
b[0] = measure q[0];
b[1] = measure q[1];

#pragma braket result adjoint_gradient h(q[0]) @ i(q[1]) alpha
```

您不必明确列出所有单独的参数，可以在编译指示中指定 all 关键字。这样可以计算出所列的所有参数的梯度，当 input 参数数量非常大时，这可能是一个方便的选择。在这种情况下，编译指示与以下示例中的代码类似。

```
#pragma braket result adjoint_gradient h(q[0]) @ i(q[1]) all
```

Amazon Braket 的 OpenQASM 3.0 实现支持所有可观察类型，包括单个运算符、张量乘积、哈密特量可观测值和 Sum 可观测值。计算梯度时要使用的特定运算符必须封装在 expectation() 函数中，而且必须明确指定可观测值的每一项所作用的量子比特。

使用 OpenQASM 3.0 测量特定的量子比特

Amazon Braket 提供的局部状态向量模拟器和局部密度矩阵模拟器支持提交可以选择性测量电路量子比特子集的 OpenQASM 程序。这种能力通常被称为部分测量，可以更有针对性、更高效地进行量子计算。例如，在以下代码片段中，您可以创建一个双量子比特电路，并选择仅测量第一个量子比特，而不测量第二个量子比特。

```
partial_measure_qasm = """
OPENQASM 3.0;
```

```
bit[1] b;
qubit[2] q;
h q[0];
cnot q[0], q[1];
b[0] = measure q[0];
"""
```

在该例中，我们有一个包含两个量子比特的量子电路，q[0] 和 q[1]，但是我们只对测量第一个量子比特的状态感兴趣。这是通过直线 b[0] = measure q[0] 来实现的，它测量的是 quit[0] 的状态并将结果存储在经典位 b[0] 中。要运行此部分测量场景，我们可以在 Amazon Braket 提供的本地状态向量模拟器上运行以下代码。

```
from braket.devices import LocalSimulator

local_sim = LocalSimulator()
partial_measure_local_sim_task =
 local_sim.run(OpenQASMProgram(source=partial_measure_qasm), shots = 10)
partial_measure_local_sim_result = partial_measure_local_sim_task.result()
print(partial_measure_local_sim_result.measurement_counts)
print("Measured qubits: ", partial_measure_local_sim_result.measured_qubits)
```

您可以通过检查设备动作属性中的 requiresAllQubitsMeasurement 字段来检查设备是否支持部分测量；如果是 False，则支持部分测量。

```
from braket.devices import Devices

AwsDevice(Devices.Rigetti.Ankaa3).properties.action['braket.ir.openqasm.program'].requiresAllQu
```

这里，requiresAllQubitsMeasurement 是 False，这表明并非所有量子比特都必须进行测量。

探索实验能力

实验功能允许访问可用性有限的硬件和新出现的新软件功能。这些功能可能会影响超出标准规格的设备性能。您可以通过 Amazon Braket SDK 按任务自动启用实验性软件功能。

要使用实验能力，请在创建量子任务时指定experimental_capabilities参数。将此参数设置为"ALL"，以启用该任务的所有可用实验功能。以下示例说明如何在设备上运行电路时启用实验功能：

```
from braket.aws import AwsDevice

device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/quera/Aquila")
```

```
task = device.run(
   circuit,
   shots=1000,
   experimental_capabilities="ALL"
)
```

> **Note**
>
> 这些功能是实验性的，可能会更改，恕不另行通知。设备性能可能与公布的规格不同，结果可能与标准操作有所不同。您必须为每项任务明确启用实验能力。没有此参数的任务将仅使用标准设备功能。

本节内容：

在 Aquila 上 QuEra 访问本地停机功能

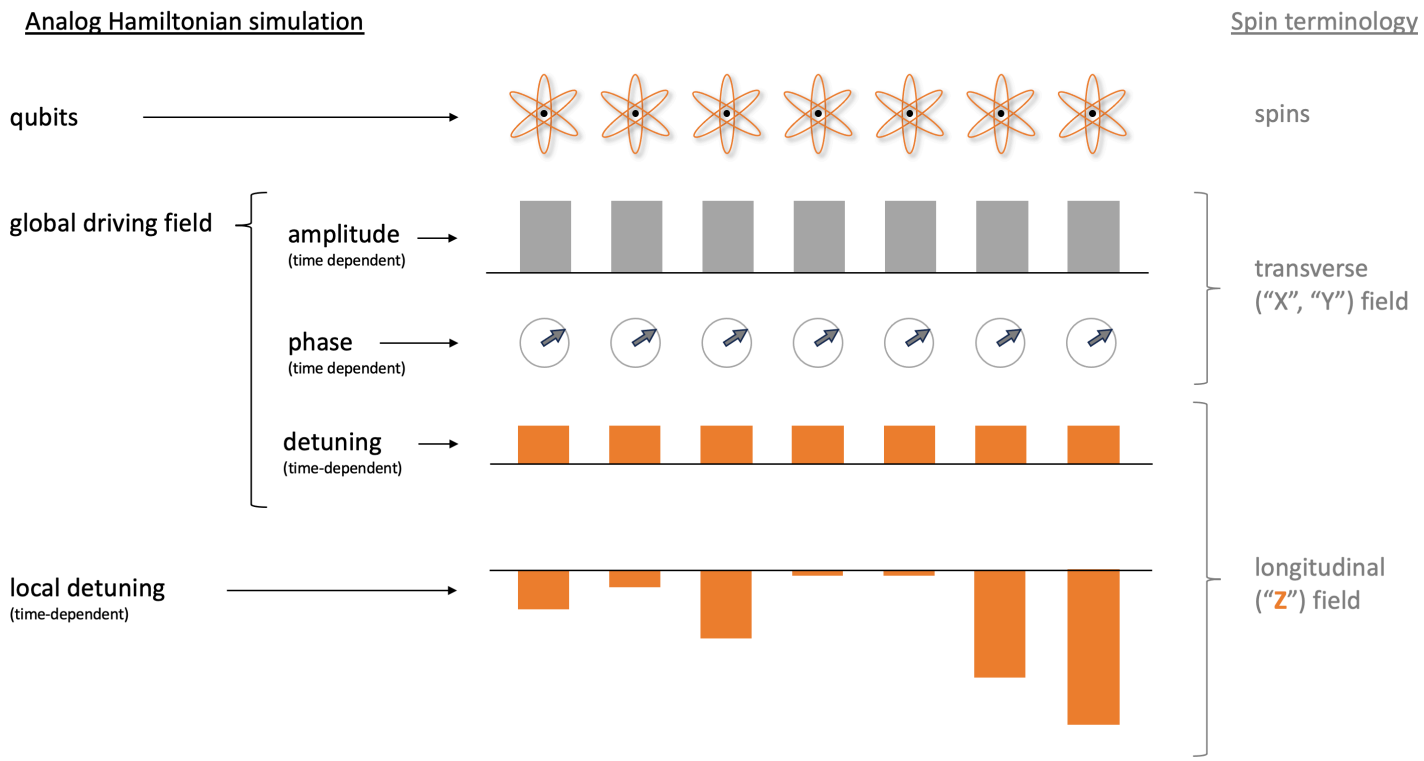
局部失谐是一个新的时变控制字段，具有可自定义的空间规律。LD 场根据可自定义的空间规律影响量子比特，从而针对不同的量子比特实现不同的哈密顿量子比特，而不仅仅是均匀驱动场和 Rydberg-Rydberg 相互作用所能创造的范围。

约束：

局部失谐字段的空间规律可以针对每个 AHS 程序进行自定义，但在整个程序过程中是恒定的。局部失谐字段的时间序列必须从零开始和结束，并且所有值都小于等于零。此外，局部失谐字段的参数受数值约束的限制，可通过 Braket SDK 的特定设备属性部分 `aquila_device.properties.paradigm.rydberg.rydbergLocal` 查看。

限制:

在运行使用局部失谐场（即使其振幅在哈密顿量中设置为恒定零）的量子程序时，设备的去相干速度比 Aquila 属性的性能部分中列出的 T2 时间更快。如不必要，最佳做法是省略 AHS 程序的哈密顿量中的局部失谐字段。



示例：

1. 模拟自旋系统中非均匀纵向磁场的影响

   虽然驱动场的振幅和相位对量子比特的影响与横向磁场对自旋的影响相同，但驱动场失谐和局部失谐之和对量子比特产生的影响与纵向磁场对自旋造成的影响相同。通过对局部失谐场的空间控制，可以模拟更复杂的自旋系统。

2. 准备非平衡初始状态

   示例 Notebook 用 Rydberg 原子模拟晶格计理论，展示了当系统向 Z2 有序相退火时，如何抑制 9 原子线性排列的中心原子受到激发。准备步骤完成后，局部失谐场会缩小，AHS 程序继续模拟系统从这种特定的非平衡状态开始的时间演变。

3. 求解加权优化问题

   示例 Notebook 最大重量独立套装（MWIS）展示了如何解决 Aquila 上的 MWIS 问题。局部失谐字段用于定义单位磁盘图节点上的权重，这些节点的边缘由 Rydberg 阻塞效应实现。从均匀基态开始，逐渐增大局部失谐场，使系统过渡到 MWIS 哈密顿量的基态，以找到问题的解决方案。

在 Aquila 上 QuEra 访问高大的几何形状

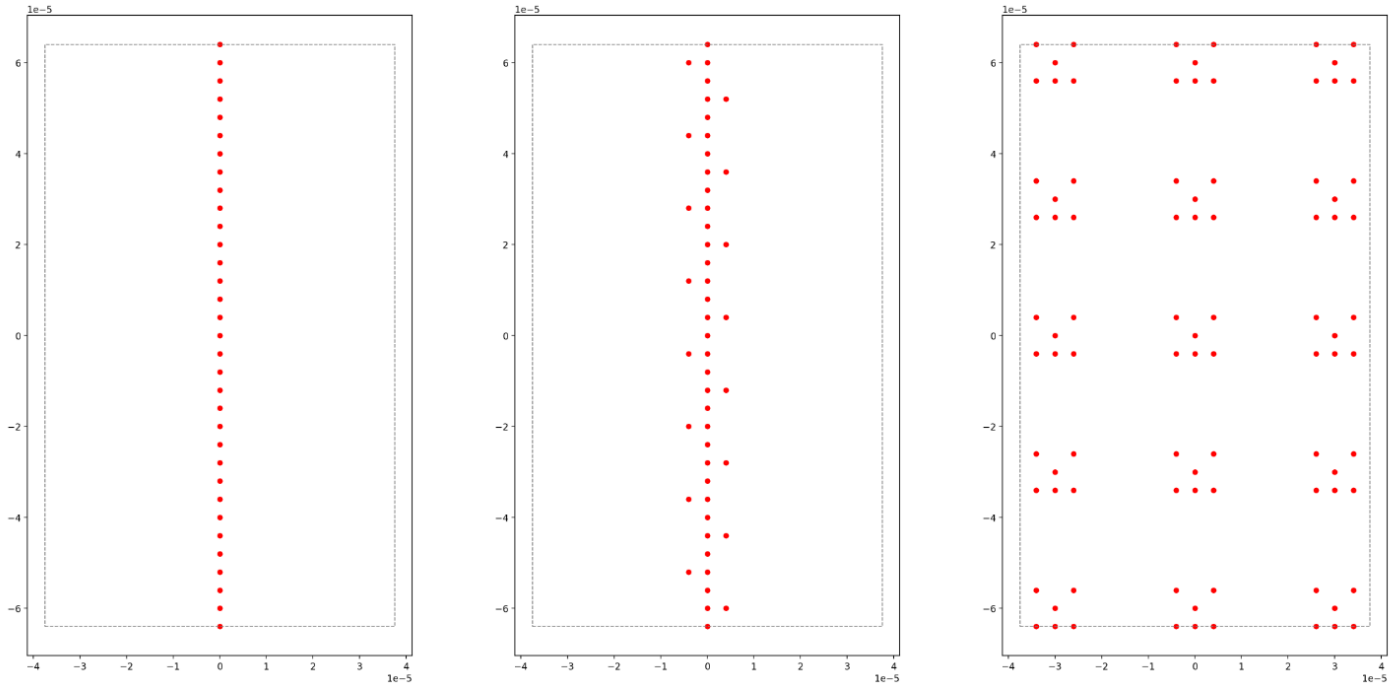
高几何图形功能可指定高度增加的几何图形。有了这种能力，您的 AHS 程序的原子排列可以在 y 方向上跨越额外的长度，超出 Aquila 的常规能力。

约束：

高几何形状的最大高度为 0.000128 米（128 微米）。

限制:

为您的账户启用此实验功能后，设备属性页面和 `GetDevice` 调用中显示的功能将继续反映常规的高度下限。当 AHS 程序使用超出常规能力的原子排列时，预计填充误差会增加。在任务结果的 `pre_sequence` 部分，您会发现 0 数量的意外增加现象，这反过来又减少了获得完美初始化排列的机会。在有许多原子的行中，这种效果最明显。



示例：

1. 更大的一维和准一维排列

   原子链和梯子状排列可以扩展到更大的原子数。通过将长方向定向平行于 y，可以对这些模型的更长实例进行编程。

2. 为多路复用小几何图形执行任务提供更大空间

示例 Notebook Aquila 上的并行量子任务展示了如何充分利用可用区域：将相关几何体的多路复用副本放在一个原子排列中。可用区域越多，可以放置的副本就越多。

在 Aquila 上 QuEra 可以看到紧凑的几何形状

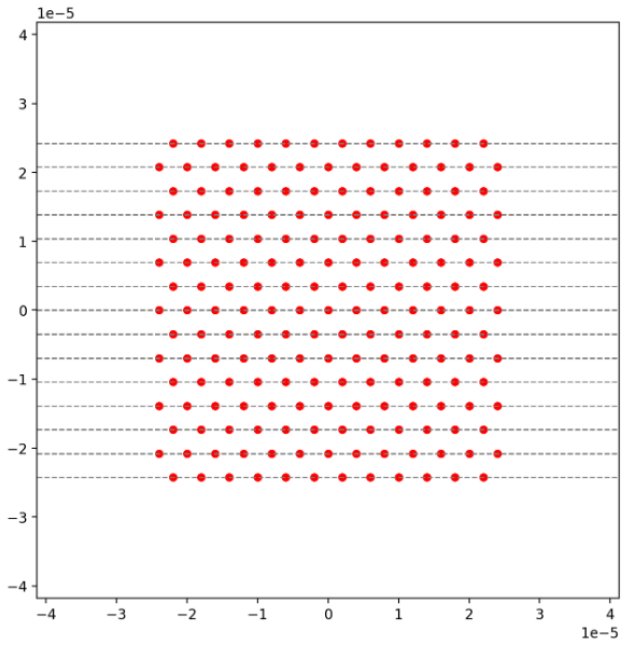
紧凑的几何结构特征可指定相邻行间距较短的几何图形。在 AHS 程序中，原子成行排列，由最小的垂直间距隔开。任意两个原子位点的 y 坐标必须为零（同一行），或者差值大于最小行间距（不同的行）。凭借紧凑的几何形状能力，可以缩小最小行距，从而创建更紧密的原子排列。虽然这种扩展不会改变原子之间的最小欧几里得距离要求，但它创建晶格，其中遥远的原子占据彼此更近的相邻行，一个值得注意的例子是三角形晶格。

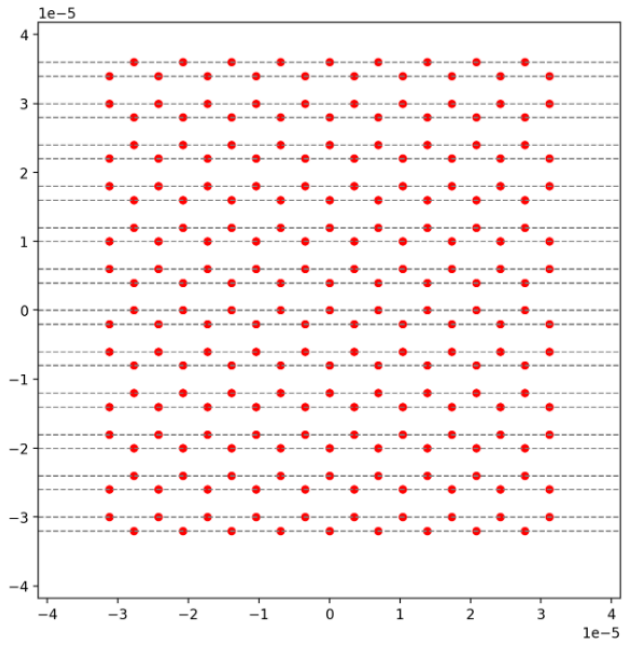
约束：

紧凑几何形状的最小行间距为 0.000002 米（2 微米）。

限制:

为您的账户启用此实验功能后，设备属性页面和 `GetDevice` 调用中显示的功能将继续反映常规的高度下限。当 AHS 程序使用超出常规能力的原子排列时，预计填充误差会增加。客户会在任务结果的 `pre_sequence` 部分发现更多意外的 0，这反过来又减少了获得完美初始化安排的机会。在有许多原子的行中，这种效果最明显。





示例：

1. 具有小晶格常数的非矩形晶格

   行间距更紧凑，可创建晶格，其中与某些原子最近的邻居在对角线方向上。值得注意的例子是三角形、六角形和 Kagome 晶格以及一些准晶体。

2. 可调晶格系列

   在 AHS 程序中，通过调整原子对之间的距离可以调整相互作用。更紧的行间距可保证以更大的自由度调整不同原子对彼此之间的相互作用，因为定义原子结构的角度和距离受最小行间距的约束较少。一个值得注意的例子是具有不同键长的 Shastry-Sutherland 晶格家族。

IQM 设备上的动态电路

IQM 设备上的动态电路支持中间电路测量（MCM）和前馈操作。有了这些功能，量子研究人员和开发人员能够实现具有条件逻辑和量子比特重用功能的高级量子算法。该实验功能有助于探索具有更高资源效率的量子算法，并研究量子误差缓解和纠错方案。

主要说明：

- `measure_ff`：实现前馈控制的测量，测量量子比特并使用反馈键存储结果。
- `cc_prx`: 实现经典控制的轮换，该轮换仅在与给定反馈键关联的结果测量 |1# 状态时适用。

Amazon Braket 通过 OpenQASM、Amazon Braket SDK 和 Amazon Braket Qiskit Provider 支持动态电路。

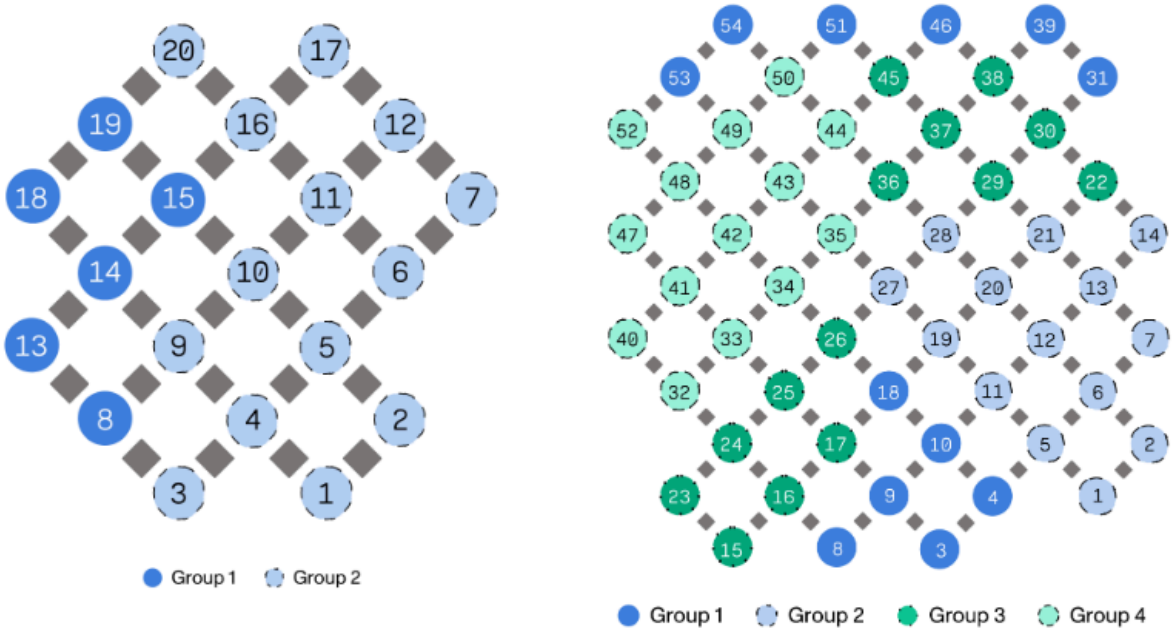
约束：

1. `measure_ff` 说明中的反馈键必须是唯一的。
2. `cc_prx` 必须在 `measure_ff` 之后，使用相同的反馈键。
3. 在单个电路中，量子比特的前馈只能由一个量子比特控制，可以由其自身控制，也可由另一个量子比特控制。在不同的电路中，您可以有不同的控制对。
   a. 例如，如果量子比特 1 由量子比特 2 控制，则无法在相同电路中由量子比特 3 控制。在量子比特 1 和量子比特 2 之间，应用控制的次数没有限制。量子比特 2 可由量子比特 3（或量子比特 1）控制，除非对量子比特 2 进行了主动重置。
4. 控制只能应用于同一组中的量子比特。IQM Garnet 和 Emerald 设备的量子比特组如下图所示。

5. 具有这些功能的程序必须作为逐字记录程序提交。要了解有关逐字记录程序的更多信息，请参阅使用 OpenQASM 3.0 进行逐字记录编译。

限制:

MCM 只能用于程序中的前馈控制。MCM 结果（0 或 1）不会作为任务结果的一部分返回。



这些图像显示了两个 IQM 设备的量子比特分组。Garnet 20 量子比特设备包含 2 组量子比特，而 Emerald 54 量子比特设备包含 4 组量子比特。

示例：

1. 通过主动重置来重复使用量子比特

   带有条件复位操作的 MCM 允许在单个电路执行中重复使用量子比特。这就降低了电路深度要求，提高了量子设备的资源利用率。

2. 主动位翻转保护

   动态电路可检测位翻转错误，并根据测量结果进行校正操作。该实现用作量子误差检测实验。

3. 传送实验

状态隐形传态使用局部量子运算和来自的经典信息传输量子比特态。 MCMs门传送无需直接进行量子运算即可实现量子比特之间的门。这些实验演示了三个关键领域的基础子程序：量子误差校正、基于测量的量子计算和量子通信。

4. 开放量子系统模拟

   动态电路通过数据量子比特和环境纠缠以及环境测量对量子系统中的噪声进行建模。这种方法使用特定的量子比特来表示数据和环境元素。噪声通道可通过对环境施加的门和测量值进行设计。

有关使用动态电路的更多信息，请参阅 Amazon Braket Notebook 存储库中的其他示例。

Amazon Braket 上的脉冲控制

脉冲是控制量子计算机中量子比特的模拟信号。使用 Amazon Braket 上的某些设备，您可以访问脉冲控制功能，使用脉冲提交电路。你可以通过 Braket SDK、OpenQasm 3.0 或直接通过 Braket 访问脉冲控制。 APIs首先介绍 Braket 中脉冲控制的一些关键概念。

本节内容：

- 帧
- 端口
- 波形
- 使用 Hello Pulse
- 使用脉冲访问原生门

帧

帧是一种软件抽象，既充当量子程序中的时钟，又充当相位。每次使用时，时钟时间都会递增，并且有状态的载波信号由频率定义。向量子比特传输信号时，帧决定量子比特的载波频率、相位偏移以及波形包络的发射时间。在 Braket Pulse 中，构造帧取决于设备、频率和相位。根据设备的不同，您可以选择预定义的帧，也可以通过提供端口来实例化新帧。

```
from braket.aws import AwsDevice
from braket.pulse import Frame, Port

# Predefined frame from a device
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
drive_frame = device.frames["Transmon_5_charge_tx"]
```

```
# Create a custom frame
readout_frame = Frame(frame_id="r0_measure", port=Port("channel_0", dt=1e-9),
 frequency=5e9, phase=0)
```

端口

端口是一种软件抽象，代表任何控制量子比特的 input/output 硬件组件。它可以帮助硬件供应商提供一个接口，用户可以通过该界面进行交互以操作和观测量子比特。端口有一个表示连接器名称的单个字符串。该字符串还显示了最小时间增量，该增量指定了我们可以定义波形的精细程度。

```
from braket.pulse import Port

Port0 = Port("channel_0", dt=1e-9)
```

波形

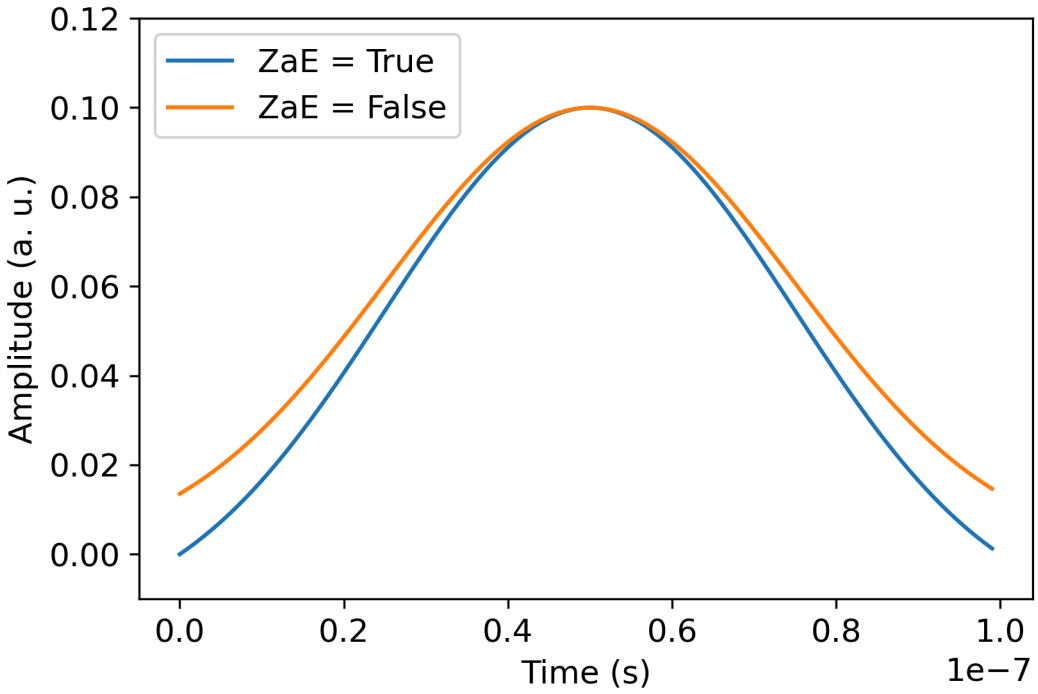
波形是一种时变包络，我们可以用它在输出端口上发射信号或通过输入端口捕获信号。您可以通过复数列表直接指定波形，也可以使用波形模板生成硬件提供商提供的列表。

```
from braket.pulse import ArbitraryWaveform, ConstantWaveform
import numpy as np

cst_wfm = ConstantWaveform(length=1e-7, iq=0.1)
arb_wf = ArbitraryWaveform(amplitudes=np.linspace(0, 100))
```

Braket Pulse 提供了一个标准波形库，包括恒定波形、高斯波形和绝热门导数去除（DRAG）波形。您可以通过 `sample` 函数检索波形数据，以绘制波形，如以下示例所示。

```
from braket.pulse import GaussianWaveform
import numpy as np
import matplotlib.pyplot as plt

zero_at_edge1 = GaussianWaveform(1e-7, 25e-9, 0.1, True)
# or zero_at_edge1 = GaussianWaveform(1e-7, 25e-9, 0.1)
zero_at_edge2 = GaussianWaveform(1e-7, 25e-9, 0.1, False)

times_1 = np.arange(0, zero_at_edge1.length, drive_frame.port.dt)
times_2 = np.arange(0, zero_at_edge2.length, drive_frame.port.dt)

plt.plot(times_1, zero_at_edge1.sample(drive_frame.port.dt))
```

```
plt.plot(times_2, zero_at_edge2.sample(drive_frame.port.dt))
```



上图描绘了从 GaussianWaveform 中创建的高斯波形。我们选择的脉冲长度为 100 纳米，宽度为 25 纳米，振幅为 0.1（任意单位）。波形在脉冲窗口中居中。GaussianWaveform 接受布尔参数 zero_at_edges（图例中的 ZaE）。设置为 True 时，此参数会偏移高斯波形，使 t=0 和 t= length 处的点为零，并重新缩放其振幅，使最大值与 amplitude 参数相对应。

使用 Hello Pulse

在本节中，您将了解如何使用 Rigetti 设备上的 Pulse 直接表征和构造单个量子比特门。对量子比特施加电磁场会导致 Rabi 振荡，在量子比特的 0 状态和 1 状态之间切换。通过校准脉冲的长度和相位，Rabi 振荡可以计算出单个量子比特门。在这里，我们将确定测量 pi/2 脉冲的最佳脉冲长度，该脉冲是用于构建更复杂的脉冲序列的基本模块。

首先，要构建脉冲序列，请导入 PulseSequence 类。

```
from braket.aws import AwsDevice
from braket.circuits import FreeParameter
from braket.devices import Devices
from braket.pulse import PulseSequence, GaussianWaveform
```

```
import numpy as np
```

接下来，使用 QPU 的 Amazon Resource Name（ARN）实例化一台新的 Braket 设备。以下命令块使用 Rigetti Ankaa-3：

```
device = AwsDevice(Devices.Rigetti.Ankaa3)
```

以下脉冲序列包括两个部分：播放波形和测量量子比特。脉冲序列通常可以应用于帧。有一些例外，如屏障和延迟，可以应用于量子比特。在构造脉冲序列之前，必须检索可用的帧。驱动框架用于施加脉冲以实现 Rabi 振荡，读出帧用于测量量子比特状态。此示例使用量子比特 25 的帧。

```
drive_frame = device.frames["Transmon_25_charge_tx"]
readout_frame = device.frames["Transmon_25_readout_rx"]
```

现在，创建要在驱动器帧中播放的波形。这样做的目的是描述量子比特在不同脉冲长度下的行为。每次您将播放一个长度不同的波形。不是每次都实例化一个新波形，而是在脉冲序列中使用 Braket 支持的 `FreeParameter`。您可以使用自由参数创建一次波形和脉冲序列，然后使用不同的输入值运行相同的脉冲序列。

```
waveform = GaussianWaveform(FreeParameter("length"), FreeParameter("length") * 0.25,
 0.2, False)
```

最后，将它们组合成脉冲序列。在脉冲序列中，`play` 播放驱动器帧上的指定波形，`capture_v0` 测量读出帧的状态。

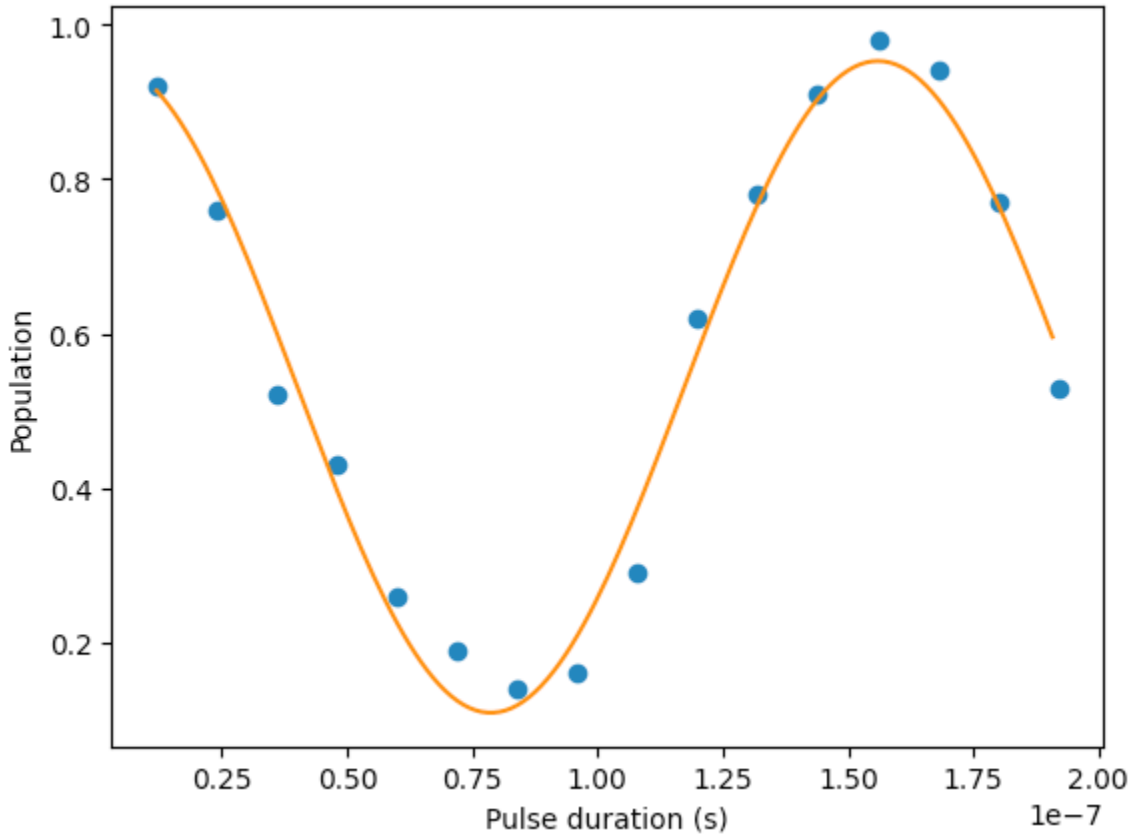
```
pulse_sequence = (
    PulseSequence()
    .play(drive_frame, waveform)
    .capture_v0(readout_frame)
)
```

扫描一定范围的脉冲长度，然后将其提交给 QPU。在 QPU 上执行脉冲序列之前，请绑定自由参数的值。

```
start_length = 12e-9
end_length = 2e-7
lengths = np.arange(start_length, end_length, 12e-9)
N_shots = 100

tasks = [
```

```
    device.run(pulse_sequence(length=length), shots=N_shots)
    for length in lengths
]

probability_of_zero = [
    task.result().measurement_counts['0']/N_shots
    for task in tasks
]
```

量子比特测量的统计数据显示了在 0 状态和 1 状态之间振荡的量子比特的振荡动力学。从测量数据中，您可以提取 Rabi 频率并微调脉冲长度以实现特定的 1 量子比特门。例如，根据下图中的数据，周期约为 154 纳米。因此，pi/2 旋转门对应于长度为 38.5 纳米的脉冲序列。



你好 Pulse 使用 OpenPulse

OpenPulse是一种用于指定通用量子器件脉冲电平控制的语言，是 OpenQasm 3.0 规范的一部分。Amazon Braket 支持 OpenPulse 使用 OpenQASM 3.0 表示形式直接对脉冲进行编程。

Braket 使用 OpenPulse 在原生指令中表达脉冲的底层中间表示形式。OpenPulse 支持以 `defcal`（“定义校准”的缩写）声明形式添加指令校准。通过这些声明，您可以在较低级别的控制语法中指定门指令的实现。

您可以使用以下命令查看 Brake PulseSequence t 的 OpenPulse 程序。

```
print(pulse_sequence.to_ir())
```

您也可以直接构造 OpenPulse 程序。

```
from braket.ir.openqasm import Program

openpulse_script = """
OPENQASM 3.0;
cal {
    bit[1] psb;
    waveform my_waveform = gaussian(12.0ns, 3.0ns, 0.2, false);
    play(Transmon_25_charge_tx, my_waveform);
    psb[0] = capture_v0(Transmon_25_readout_rx);
}
"""
```

使用脚本创建 Program 对象。然后，将该程序提交给 QPU。

```
from braket.aws import AwsDevice
from braket.devices import Devices
from braket.ir.openqasm import Program

program = Program(source=openpulse_script)

device = AwsDevice(Devices.Rigetti.Ankaa3)
task = device.run(program, shots=100)
```

使用脉冲访问原生门

研究人员通常需要确切了解特定 QPU 支持的原生门是如何实现为脉冲的。脉冲序列由硬件提供商仔细校准，但是访问脉冲序列为研究人员提供了设计更好的门或探索错误缓解协议的机会，如通过拉伸特定门的脉冲来进行零噪声外推。

Amazon Braket 支持以编程方式从 Rigetti 访问原生门。

```
import math
from braket.aws import AwsDevice
from braket.circuits import Circuit, GateCalibrations, QubitSet
from braket.circuits.gates import Rx
```

```
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")

calibrations = device.gate_calibrations
print(f"Downloaded {len(calibrations)} calibrations.")
```

> **Note**
>
> 硬件提供商定期校准 QPU，通常每天不止一次。通过 Braket SDK，您能够获得最新的门校准。

```
device.refresh_gate_calibrations()
```

要检索给定的原生门，如 RX 或 XY 门，您需要传递相关 `Gate` 对象和量子比特。例如，您可以检查应用于 qubit 0 的 RX(π/2) 的脉冲实现。

```
rx_pi_2_q0 = (Rx(math.pi/2), QubitSet(0))

pulse_sequence_rx_pi_2_q0 = calibrations.pulse_sequences[rx_pi_2_q0]
```

您可以使用 `filter` 函数创建一组经过筛选的校准。您传递了一个门列表或一个 `QubitSet` 列表。以下代码创建了两个集合，其中包含 RX(π/2) 和 qubit 0 的所有校准。

```
rx_calibrations = calibrations.filter(gates=[Rx(math.pi/2)])
q0_calibrations = calibrations.filter(qubits=QubitSet([0]))
```

现在，您可以通过附加自定义校准集来提供或修改原生门的操作。例如，考虑以下电路：

```
bell_circuit = (
    Circuit()
    .rx(0, math.pi/2)
    .rx(1, math.pi/2)
    .iswap(0, 1)
    .rx(1, -math.pi/2)
)
```

您可以通过将 `PulseSequence` 对象字典传递给 `gate_definitions` 关键字参数 `qubit 0` 来运行它，为 `rx` 门开启自定义门校准。您可以根据 `GateCalibrations` 对象的属性 `pulse_sequences` 构造字典。所有未指定的门都被量子硬件提供商的脉冲校准所取代。

```
nb_shots = 50
custom_calibration = GateCalibrations({rx_pi_2_q0: pulse_sequence_rx_pi_2_q0})
task = device.run(bell_circuit, gate_definitions=custom_calibration.pulse_sequences,
 shots=nb_shots)
```

模拟哈密顿模拟

模拟哈密顿模拟（AHS）是量子计算的新兴范式，与传统的量子电路模型有很大的不同。它不是一系列门，即每个电路一次只能作用于几个量子比特。AHS 程序由相关哈密顿量的时变和空间相关参数定义。系统的哈密顿量对其能级和外力的影响进行编码，它们共同控制着其状态的时间演变。对于 N 量子比特系统，哈密顿量可以用 $2^N X 2^N$ 个复数方阵表示。

能够执行 AHS 的量子设备旨在通过仔细调整其内部控制参数，近似定制哈密顿量下的量子系统的时间演变。例如，调整相干驱动场的振幅和失谐参数。AHS 范式非常适合模拟具有许多相互作用粒子的量子系统的静态和动态特性，比如在凝聚态物理学或量子化学中。专门设计的量子处理单元 (QPUs)，例如的 Aquila设备QuEra，是为了利用AHS的力量，以创新的方式解决传统数字量子计算方法无法解决的问题。

本节内容：

- Hello AHS：运行您的第一个模拟哈密顿模拟
- 使用 A QuEra quila 提交模拟节目

Hello AHS：运行您的第一个模拟哈密顿模拟

本节提供了有关运行第一个模拟哈密顿模拟的信息。

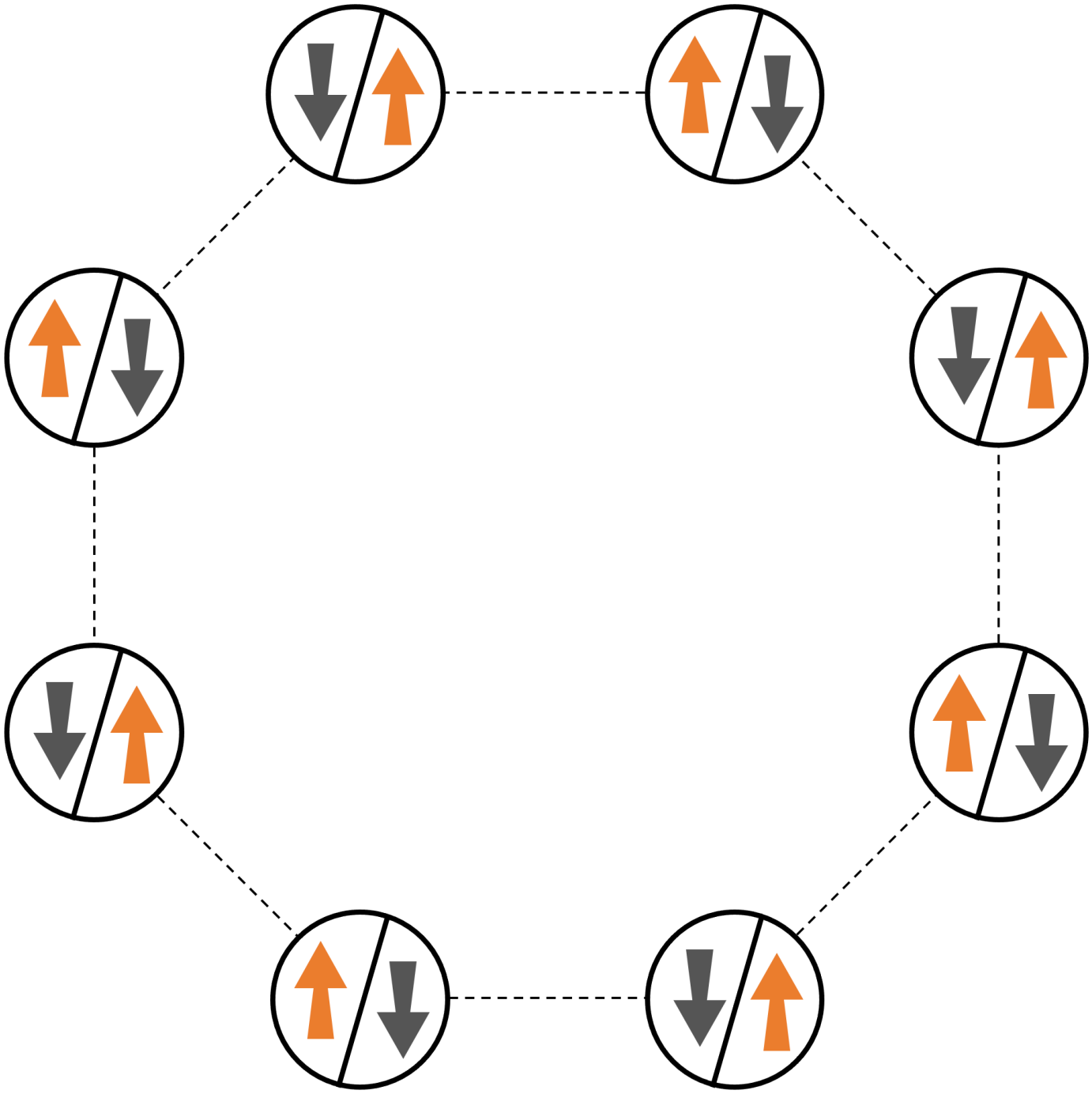
本节内容：

- 交互式旋转链
- 排列
- 相互作用
- 驱动场
- AHS 程序
- 在本地模拟器上运行
- 分析模拟器结果
- Run QuEra ning on 的 Aquila QPU

- 分析结果
- 后续步骤

交互式旋转链

举一个由许多相互作用粒子组成的系统的典型示例，让我们考虑一个由八个旋转组成的环（每个旋转都可以处于“向上”|↑# 和“向下”|↓# 的状态）。尽管规模很小，但该模型系统已经表现出自然存在的磁性材料的一些有趣现象。在此例中，我们将展示如何准备一个所谓的反铁磁阶数，即连续的自旋指向相反的方向。



排列

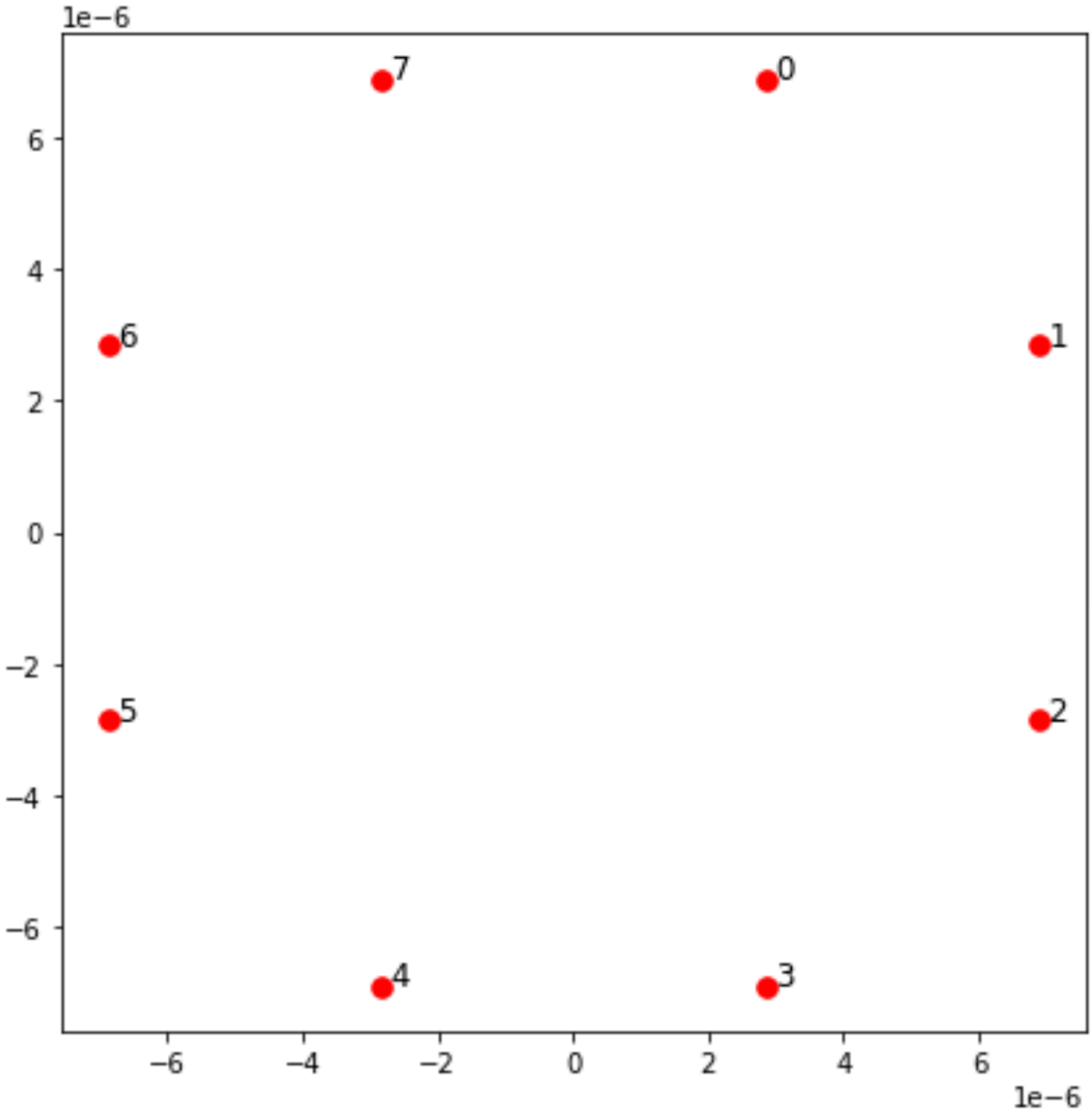
我们将使用一个中性原子来代表每次自旋，“向上”和“向下”自旋态将分别以激发的里德伯格态和原子的基态编码。首先，创建二维排列。我们可以用以下代码对上面的旋转圈进行编程。

先决条件：您需要 pip 安装 Braket SDK。（如果您使用的是 Braket 托管的 Notebook 实例，则此 SDK 已预先安装在 Notebook 中。）要重现绘图，您还需要使用 shell 命令 `pip install matplotlib` 单独安装 matplotlib。

```
from braket.ahs.atom_arrangement import AtomArrangement
import numpy as np
import matplotlib.pyplot as plt  # Required for plotting

a = 5.7e-6  # Nearest-neighbor separation (in meters)

register = AtomArrangement()
register.add(np.array([0.5, 0.5 + 1/np.sqrt(2)]) * a)
register.add(np.array([0.5 + 1/np.sqrt(2), 0.5]) * a)
register.add(np.array([0.5 + 1/np.sqrt(2), - 0.5]) * a)
register.add(np.array([0.5, - 0.5 - 1/np.sqrt(2)]) * a)
register.add(np.array([-0.5, - 0.5 - 1/np.sqrt(2)]) * a)
register.add(np.array([-0.5 - 1/np.sqrt(2), - 0.5]) * a)
register.add(np.array([-0.5 - 1/np.sqrt(2), 0.5]) * a)
register.add(np.array([-0.5, 0.5 + 1/np.sqrt(2)]) * a)
```

我们还可以绘制散点图，

```
fig, ax = plt.subplots(1, 1, figsize=(7, 7))
xs, ys = [register.coordinate_list(dim) for dim in (0, 1)]
ax.plot(xs, ys, 'r.', ms=15)

for idx, (x, y) in enumerate(zip(xs, ys)):
    ax.text(x, y, f" {idx}", fontsize=12)

plt.show()  # This will show the plot below in an ipython or jupyter session
```



相互作用

为准备反铁磁相，我们需要诱导相邻自旋之间的相互作用。为此，我们使用了 van der Waals 相互作用，该相互作用是由中性原子设备（如来自 QuEra 的 Aquila 设备）原生实现的。这种相互作用的哈密顿项使用自旋表示，可以表示为所有自旋对 (j,k) 的总和。

$$H_{\text{interaction}} = \sum_{j=1}^{N-1} \sum_{k=j+1}^{N} V_{j,k} n_j n_k$$

其中：nj=|↑$_j$##↑$_j$| 是一个运算符，仅当自旋 j 处于“向上”状态时才取值 1，否则取值 0。强度为 $V_{j,k}=C_6/(d_{j,k})^6$，其中：C_6 是固定系数，$d_{j,k}$ 是自旋 j 和 k 之间的欧几里得距离。这个相互作用项的直接影响是，任何自旋 j 和自旋 k 都是“向上”状态的能量都会升高（按量 $V_{j,k}$）。通过精心设计 AHS 程序的其余部分，这种相互作用可防止相邻的旋转都处于“向上”状态，这种效果通常被称为“Rydberg 阻塞”。

驱动场

AHS 程序开始时，所有旋转（默认情况下）都以“向下”状态开始，它们处于所谓的铁磁阶段。着眼于准备反铁磁相位的目标，我们指定了一个时变相干驱动场，该驱动场可以平稳地将自旋从这种状态过渡到多体状态，首选为“向上”状态。相应的哈密顿量可以写成

$$H_{\text{drive}}(t) = \sum_{k=1}^{N} \frac{1}{2}\Omega(t)\left[e^{i\phi(t)} S_{-,k} + e^{-i\phi(t)} S_{+,k}\right] - \sum_{k=1}^{N} \Delta(t) n_k$$

其中 Ω(t)、(t)、Δ(t) 是时变全局振幅（又名 Rabi 频率）、相位和失谐均匀地影响所有自旋的驱动场。其中：$S_{-,k}$ =|↓$_k$##↑$_k$|and $S_{+,k}=(S_{-,k})^{\dagger}$=|↑$_k$##↓$_k$|分别是 spin k 的降低和升高运算符，$n_k$=|↑$_k$##↑$_k$|和以前的运算符相同。驱动场的 Ω 部分同时连贯地耦合所有旋转的“向下”和“向上”状态，而 Δ 部分控制“向上”状态的能量奖励。

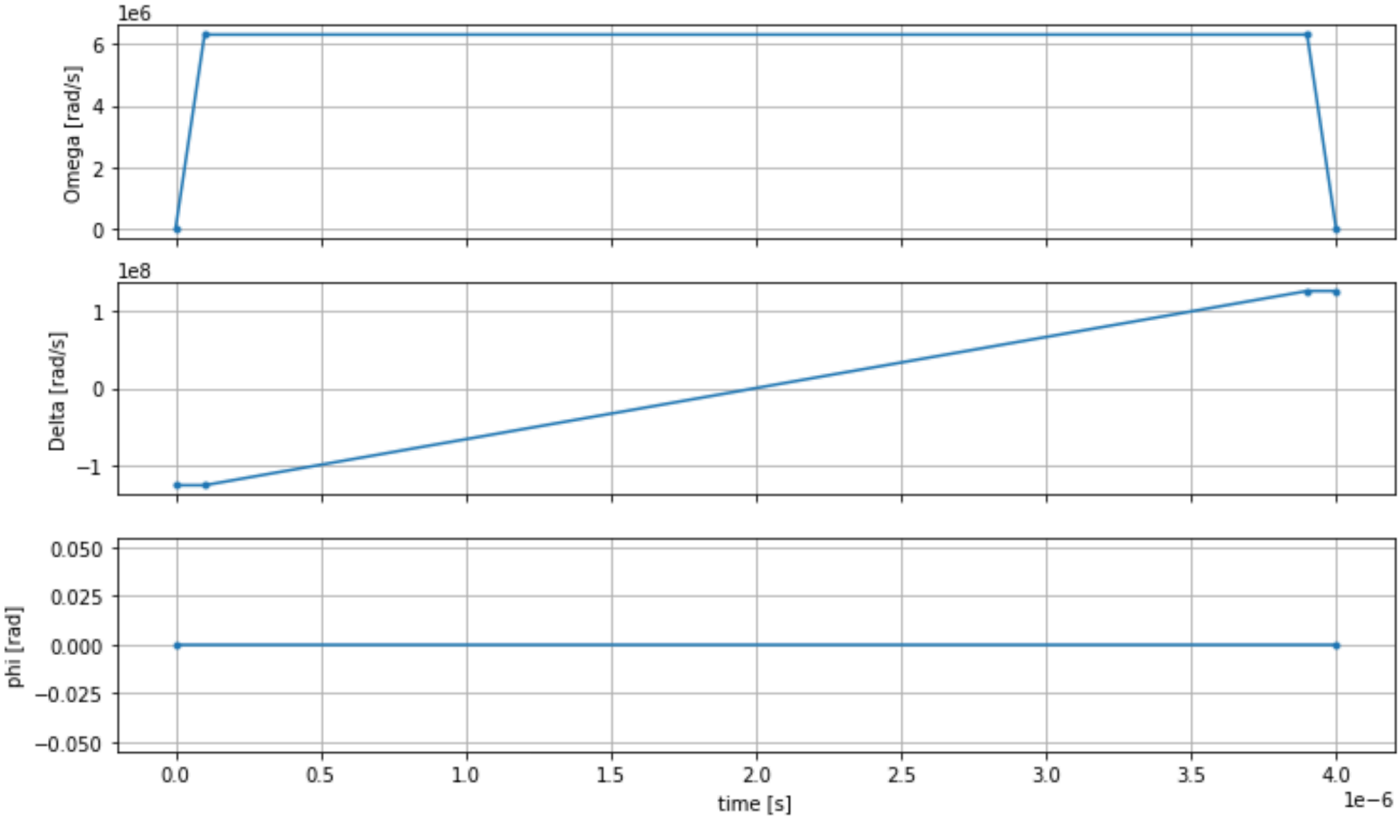
为了对从铁磁相向到反铁磁相的平滑过渡进行编程，我们使用以下代码指定驱动场。

```
from braket.timings.time_series import TimeSeries
from braket.ahs.driving_field import DrivingField

# Smooth transition from "down" to "up" state
time_max = 4e-6  # seconds
time_ramp = 1e-7  # seconds
omega_max = 6300000.0  # rad / sec
delta_start = -5 * omega_max
delta_end = 5 * omega_max

omega = TimeSeries()
omega.put(0.0, 0.0)
omega.put(time_ramp, omega_max)
omega.put(time_max - time_ramp, omega_max)
omega.put(time_max, 0.0)
```

```
delta = TimeSeries()
delta.put(0.0, delta_start)
delta.put(time_ramp, delta_start)
delta.put(time_max - time_ramp, delta_end)
delta.put(time_max, delta_end)

phi = TimeSeries().put(0.0, 0.0).put(time_max, 0.0)

drive = DrivingField(
   amplitude=omega,
   phase=phi,
   detuning=delta
)
```

我们可以使用以下脚本可视化驱动场的时间序列。

```
fig, axes = plt.subplots(3, 1, figsize=(12, 7), sharex=True)

ax = axes[0]
time_series = drive.amplitude.time_series
ax.plot(time_series.times(), time_series.values(), '.-')
ax.grid()
ax.set_ylabel('Omega [rad/s]')

ax = axes[1]
time_series = drive.detuning.time_series
ax.plot(time_series.times(), time_series.values(), '.-')
ax.grid()
ax.set_ylabel('Delta [rad/s]')

ax = axes[2]
time_series = drive.phase.time_series
# Note: time series of phase is understood as a piecewise constant function
ax.step(time_series.times(), time_series.values(), '.-', where='post')
ax.set_ylabel('phi [rad]')
ax.grid()
ax.set_xlabel('time [s]')

plt.show()  # This will show the plot below in an ipython or jupyter session
```



AHS 程序

寄存器、驱动场（以及隐式的范德华相互作用）构成了模拟哈密顿模拟程序 ahs_program。

```
from braket.ahs.analog_hamiltonian_simulation import AnalogHamiltonianSimulation

ahs_program = AnalogHamiltonianSimulation(
   register=register,
   hamiltonian=drive
)
```

在本地模拟器上运行

由于此示例很小（少于 15 次旋转），因此在兼容 AHS 的 QPU 上运行之前，我们可以在 Braket SDK 附带的本地 AHS 模拟器上运行它。由于本地模拟器可免费使用 Braket SDK，因而这是确保我们的代码能够正确执行的最佳实践。

在这里，我们可以将拍摄次数设置为较大的值（比如 100 万），因为本地模拟器会跟踪量子态的时间演变并从最终状态中抽取样本；因此，可以增加拍摄次数，而总运行时仅略有增加。

```
from braket.devices import LocalSimulator

device = LocalSimulator("braket_ahs")

result_simulator = device.run(
   ahs_program,
   shots=1_000_000
).result()  # Takes about 5 seconds
```

分析模拟器结果

我们可以使用以下函数汇总拍摄结果，该函数可以推断每次旋转的状态（可以是“d”代表“向下”，“u”表示“向上”，“e“代表空场地），并计算每种配置在拍摄中发生的次数。

```
from collections import Counter


def get_counts(result):
    """Aggregate state counts from AHS shot results

    A count of strings (of length = # of spins) are returned, where
    each character denotes the state of a spin (site):
      e: empty site
      u: up state spin
      d: down state spin

    Args:
      result
 (braket.tasks.analog_hamiltonian_simulation_quantum_task_result.AnalogHamiltonianSimulationQua

    Returns
       dict: number of times each state configuration is measured

    """
    state_counts = Counter()
    states = ['e', 'u', 'd']
    for shot in result.measurements:
        pre = shot.pre_sequence
        post = shot.post_sequence
        state_idx = np.array(pre) * (1 + np.array(post))
        state = "".join(map(lambda s_idx: states[s_idx], state_idx))
        state_counts.update((state,))
    return dict(state_counts)
```

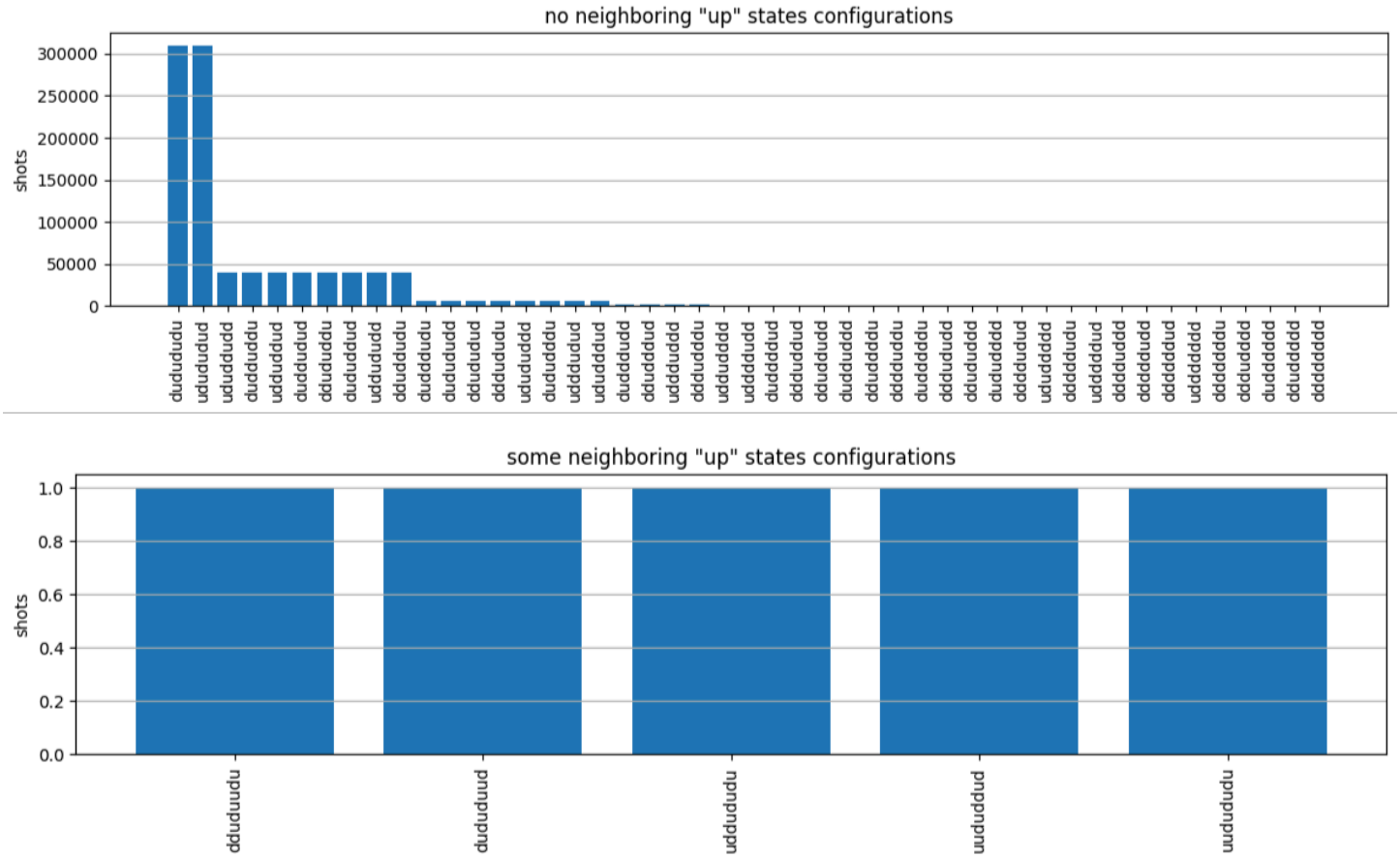
```
counts_simulator = get_counts(result_simulator)  # Takes about 5 seconds
print(counts_simulator)
```

```
*[Output]*
{'dddddddd': 5, 'ddddddddu': 12, 'ddddddud': 15, ...}
```

在这里，counts 是一本字典，它计算了拍摄中观察到的每种状态配置的次数。我们还可以使用以下代码实现它们的可视化。

```
from collections import Counter


def has_neighboring_up_states(state):
    if 'uu' in state:
        return True
    if state[0] == 'u' and state[-1] == 'u':
        return True
    return False


def number_of_up_states(state):
    return Counter(state)['u']


def plot_counts(counts):
    non_blockaded = []
    blockaded = []
    for state, count in counts.items():
        if not has_neighboring_up_states(state):
            collection = non_blockaded
        else:
            collection = blockaded
        collection.append((state, count, number_of_up_states(state)))

    blockaded.sort(key=lambda _: _[1], reverse=True)
    non_blockaded.sort(key=lambda _: _[1], reverse=True)

    for configurations, name in zip((non_blockaded,
                                     blockaded),
                                    ('no neighboring "up" states',
                                     'some neighboring "up" states')):
```

```
        plt.figure(figsize=(14, 3))
        plt.bar(range(len(configurations)), [item[1] for item in configurations])
        plt.xticks(range(len(configurations)))
        plt.gca().set_xticklabels([item[0] for item in configurations], rotation=90)
        plt.ylabel('shots')
        plt.grid(axis='y')
        plt.title(f'{name} configurations')
        plt.show()


plot_counts(counts_simulator)
```



从图中，我们可以读出以下观测结果，以验证我们成功制备了反铁磁相。

1. 通常，非阻塞状态（没有两个相邻的旋转处于“向上”状态）比至少有一对相邻旋转都处于“向上”状态的状态更为常见。
2. 通常，除非配置被阻止，否则会优先选择激励更多“向上”状态。
3. 最常见的状态确实是完美的反铁磁态 "dudududu" 和 "udududud"。

4. 第二种常见的状态是只有 3 个“向上”激励，连续间隔为 1、2、2 的状态。这表明范德华的相互作用也会对最近的邻居产生影响（尽管该影响要小得多）。

Run QuEra ning on 的 Aquila QPU

先决条件：除了 pip 安装 Braket SDK 之外，如果您不熟悉 Amazon Braket，请确保您已完成必要的入门步骤。

> **Note**
>
> 如果您使用的是 Braket 托管的 Notebook 实例，则该实例预装了 Braket SDK。

安装所有依赖项后，我们就可以连接到 Aquila QPU 了。

```
from braket.aws import AwsDevice

aquila_qpu = AwsDevice("arn:aws:braket:us-east-1::device/qpu/quera/Aquila")
```

为了使我们的 AHS 程序适合 QuEra 机器，我们需要对所有值进行四舍五入，以符合 Aquila QPU 规定的精度水平。（这些要求受名称中带有“分辨率”的设备参数的约束。我们可以通过在 Notebook 中执行 `aquila_qpu.properties.dict()` 来看到它们。有关 Aquila 功能和要求的更多详细信息，请参阅 Aquila Notebook 简介。）我们可以通过调用 `discretize` 方法来做到这一点。

```
discretized_ahs_program = ahs_program.discretize(aquila_qpu)
```

现在，我们可以在 Aquila QPU 上运行该程序（目前只运行 100 次拍摄）。

> **Note**
>
> 在 Aquila 处理器上运行此程序将产生一定的成本。Amazon Braket SDK 包含一个成本追踪器，这样，客户能够设置成本限额并近乎实时地跟踪成本。

```
task = aquila_qpu.run(discretized_ahs_program, shots=100)

metadata = task.metadata()
task_arn = metadata['quantumTaskArn']
```

```
task_status = metadata['status']

print(f"ARN: {task_arn}")
print(f"status: {task_status}")
```

```
*[Output]*
ARN: arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef
status: CREATED
```

由于量子任务的运行时差异很大（取决于可用窗口和 QPU 利用率），因此记下量子任务 ARN 是个好主意，这样我们就可以在后续时间使用以下代码片段检查其状态了。

```
# Optionally, in a new python session
from braket.aws import AwsQuantumTask

SAVED_TASK_ARN = "arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef"

task = AwsQuantumTask(arn=SAVED_TASK_ARN)
metadata = task.metadata()
task_arn = metadata['quantumTaskArn']
task_status = metadata['status']

print(f"ARN: {task_arn}")
print(f"status: {task_status}")
```

```
*[Output]*
ARN: arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef
status: COMPLETED
```

状态为“已完成”（也可以从 Amazon Braket 控制台的量子任务页面进行检查）后，我们可以通过以下方式查询结果：

```
result_aquila = task.result()
```

分析结果

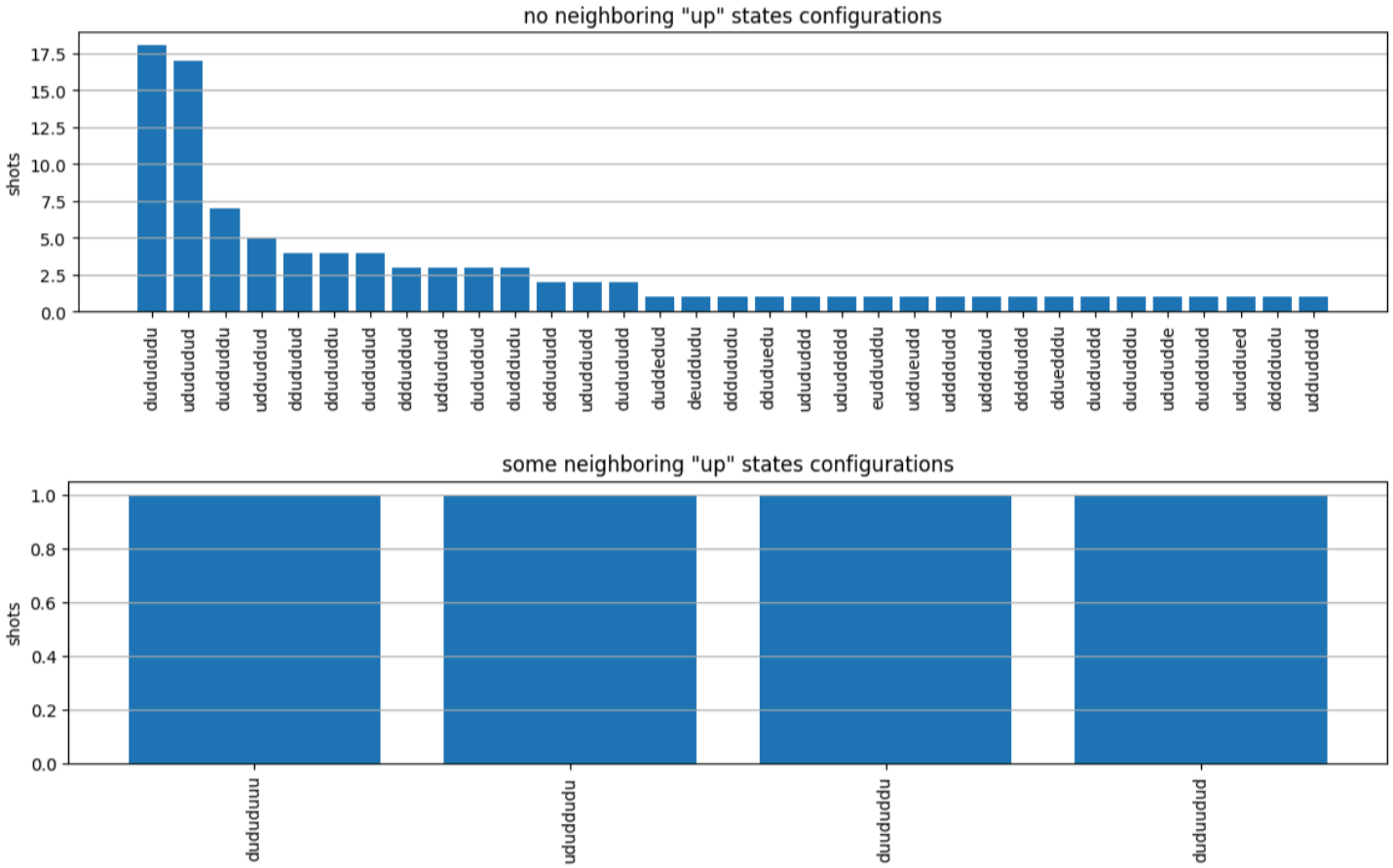
我们可以使用与以前相同的 `get_counts` 函数计算计数。

```
counts_aquila = get_counts(result_aquila)
   print(counts_aquila)
```

```
*[Output]*
{'ddddududd': 2, 'dududududu': 18, 'dddududud': 4, ...}
```

然后，用 plot_counts 绘制函数：

```
plot_counts(counts_aquila)
```



请注意，一小部分拍摄的场地为空（标为“e”）。这是由于 Aquila QPU 的每个原子制备存在 1%-2% 的缺陷。除此之外，由于拍摄次数少，结果在预期的统计波动范围内与模拟相符。

后续步骤

恭喜！您现在已经使用本地 AHS 模拟器和 Aquila QPU 在 Amazon Braket 上运行了第一个 AHS 工作负载。

要了解有关 Rydberg 物理学、模拟哈密顿模拟和 Aquila 设备的更多信息，请参阅我们的示例 Notebook。

使用 A QuEra quila 提交模拟节目

本页提供了有关来自 QuEra 的 Aquila 机器能力的整个文档。此处涵盖的详细信息如下：

1. 通过以下方法模拟的参数化哈密顿量模型 Aquila
2. AHS 程序参数
3. AHS 结果内容
4. Aquila 能力参数

本节内容：

- 哈密顿量
- Braket AHS 程序架构
- Braket AHS 任务结果架构
- QuEra 设备属性架构

哈密顿量

来自 QuEra 的 Aquila 机器以原生方式对以下（时变）哈密顿量程进行模拟。

$$H(t) = \sum_{k=1}^{N} H_{\text{drive},k}(t) + \sum_{k=1}^{N} H_{\text{local detuning},k}(t) + \sum_{k=1}^{N-1} \sum_{l=k+1}^{N} V_{\text{vdw},k,l}$$

> **Note**
>
> 访问局部失谐是一项实验能力，可通过 Braket Direct 申请获得。

where

- $H_{drive,k}(t) = (\ ^1/_2\ \Omega(t)e^{i\phi(t)}S_{-,k} + \ ^1/_2\ \Omega(t)e^{-i\phi(t)}\ S_{+,k}) + (-\Delta_{global}(t)n_k)$，
  - Ω(t) 是时变全局驱动振幅（又名 Rabi 频率），单位为（弧度/秒）
  - ϕ(t) 是时变全局相位，单位为弧度

  - $S_{-,k}$ 和 $S_{+,k}$ 是原子 k 的自旋降低和升高运算符（在基数中，|↓#=|g#,|↑#=|r），它们是 S_{-}=|g##r|,$S_{+}=(S_{-})^{\dagger}$=|r##g|）
  - $\Delta_{global}(t)$ 是时变全局失谐
  - n_k 是原子 k 的里德伯格状态的投影运算符（也就是说，n=|r##r|）
- $H_{local\ detuning,k}(t)=-\Delta_{local}(t)h_k n_k$
  - $\Delta_{local}(t)$ 是局部频移的时变因子，单位为（弧度/秒）
  - h_k 是与位置相关的因子，介于 0.0 到 1.0 之间的无量纲数字
- $V_{vdw,k,l}=C_6/(d_{k,l})^6 n_k n_l$，
  - C_6 是范德华系数，单位为 (rad / s) * (m)^6
  - $d_{k,l}$ 是原子 k 和 l 之间的欧几里得距离，单位为米。

用户可以通过 Braket AHS 程序架构控制以下参数。

- 二维原子排列（每个原子 k 的 x_k 和 y_k 坐标，单位为 um），它控制成对原子距离 $d_{k,l}$，其中：k,l=1,2,…N
- Ω(t)，时变全局 Rabi 频率，单位为（弧度/秒）
- ϕ(t)，时变全局相位，单位为（弧度）
- $\Delta_{global}(t)$，时变全局失谐，单位为（弧度/秒）
- $\Delta_{local}(t)$，局部失谐振幅的时变（全局）因子，单位为（弧度/秒）
- h_k，与地点相关的局部失谐振幅的（静态）因子，属于介于 0.0 和 1.0 之间的无量纲数字

> **Note**
>
> 用户无法控制涉及哪些级别（即 S_{-}、S_{+}、n 运算符是固定的），也无法控制里德伯-里德伯交互系数 (C_6) 的强度。

Braket AHS 程序架构

braket.ir.ahs.program_v1.Program 对象（示例）

> **Note**
>
> 如果您的账户未启用局部失谐功能，请在以下示例中使用 `localDetuning=[]`。

```
Program(
    braketSchemaHeader=BraketSchemaHeader(
        name='braket.ir.ahs.program',
        version='1'
    ),
    setup=Setup(
        ahs_register=AtomArrangement(
            sites=[
                [Decimal('0'), Decimal('0')],
                [Decimal('0'), Decimal('4e-6')],
                [Decimal('4e-6'), Decimal('0')]
            ],
            filling=[1, 1, 1]
        )
    ),
    hamiltonian=Hamiltonian(
        drivingFields=[
            DrivingField(
                amplitude=PhysicalField(
                    time_series=TimeSeries(
                        values=[Decimal('0'), Decimal('15700000.0'),
 Decimal('15700000.0'), Decimal('0')],
                        times=[Decimal('0'), Decimal('0.000001'), Decimal('0.000002'),
 Decimal('0.000003')]
                    ),
                    pattern='uniform'
                ),
                phase=PhysicalField(
                    time_series=TimeSeries(
                        values=[Decimal('0'), Decimal('0')],
                        times=[Decimal('0'), Decimal('0.000003')]
                    ),
                    pattern='uniform'
                ),
                detuning=PhysicalField(
                    time_series=TimeSeries(
                        values=[Decimal('-54000000.0'), Decimal('54000000.0')],
                        times=[Decimal('0'), Decimal('0.000003')]
                    ),
                    pattern='uniform'
                )
            )
        ],
```

```
        localDetuning=[
            LocalDetuning(
                magnitude=PhysicalField(
                    times_series=TimeSeries(
                        values=[Decimal('0'), Decimal('25000000.0'),
 Decimal('25000000.0'), Decimal('0')],
                        times=[Decimal('0'), Decimal('0.000001'), Decimal('0.000002'),
 Decimal('0.000003')]
                    ),
                    pattern=Pattern([Decimal('0.8'), Decimal('1.0'), Decimal('0.9')])
                )
            )
        ]
    )
)
```

JSON（示例）

> **Note**
>
> 如果您的账户未启用局部失谐功能，请在以下示例中使用 "localDetuning": []。

```
{
    "braketSchemaHeader": {
        "name": "braket.ir.ahs.program",
        "version": "1"
    },
    "setup": {
        "ahs_register": {
            "sites": [
                [0E-7, 0E-7],
                [0E-7, 4E-6],
                [4E-6, 0E-7]
            ],
            "filling": [1, 1, 1]
        }
    },
    "hamiltonian": {
        "drivingFields": [
            {
                "amplitude": {
```

```
                    "time_series": {
                        "values": [0.0, 15700000.0, 15700000.0, 0.0],
                        "times": [0E-9, 0.000001000, 0.000002000, 0.000003000]
                    },
                    "pattern": "uniform"
                },
                "phase": {
                    "time_series": {
                        "values": [0E-7, 0E-7],
                        "times": [0E-9, 0.000003000]
                    },
                    "pattern": "uniform"
                },
                "detuning": {
                    "time_series": {
                        "values": [-54000000.0, 54000000.0],
                        "times": [0E-9, 0.000003000]
                    },
                    "pattern": "uniform"
                }
            }
        ],
        "localDetuning": [
            {
                "magnitude": {
                    "time_series": {
                        "values": [0.0, 25000000.0, 25000000.0, 0.0],
                        "times": [0E-9, 0.000001000, 0.000002000, 0.000003000]
                    },
                    "pattern": [0.8, 1.0, 0.9]
                }
            }
        ]
    }
}
```

主要字段

程序字段	类型	描述
setup.ahs_register.sites	List[List[Decimal]]	镊子捕获原子的二维坐标清单

程序字段	类型	描述
setup.ahs_register.filling	List[int]	用 1 标记占据陷阱位点的原子，用 0 标记占据空位点的原子
hamiltonian.drivingFields[].amplitude.time_series.times	List[Decimal]	驱动振幅的时间点，Omega(t)
hamiltonian.drivingFields[].amplitude.time_series.values	List[Decimal]	驱动振幅值，Omega(t)
hamiltonian.drivingFields[].amplitude.pattern	str	驱动振幅的空间规律，Omega(t)；必须是“均匀的”
hamiltonian.drivingFields[].phase.time_series.times	List[Decimal]	驾驶相位的时间点，phi(t)
hamiltonian.drivingFields[].phase.time_series.values	List[Decimal]	驱动相位的值，phi(t)
hamiltonian.drivingFields[].phase.pattern	str	驱动相位的空间规律，phi(t)；必须是“均匀的”
hamiltonian.drivingFields[].detuning.time_series.times	List[Decimal]	驱动的时间点，delta_Global(t)
hamiltonian.drivingFields[].detuning.time_series.values	List[Decimal]	驱动失谐值，delta_Global(t)
hamiltonian.drivingFields[].detuning.pattern	str	驾驶失谐的空间规律，delta_Global(t)；必须是“均匀的”

程序字段	类型	描述
hamiltonian.localDetuning[].magnitude.time_series.times	List[Decimal]	局部失谐振幅的时变因子的时间点，delta_Local(t)
hamiltonian.localDetuning[].magnitude.time_series.values	List[Decimal]	局部失谐振幅的时变因子值，delta_Local(t)
hamiltonian.localDetuning[].magnitude.pattern	List[Decimal]	局部失谐振幅的站点相关因子 h_k（数值对应于 setup.ahs_register.sites 中的站点）

元数据字段

程序字段	类型	描述
braketSchemaHeader.name	str	架构的名称；必须是“braket.ir.ahs.program”
braketSchemaHeader. 版本	str	架构版本

Braket AHS 任务结果架构

braket.tasks.analog_hamiltonian_simulation_quantum_task_result AnalogHamiltonianSimulationQuantumTaskResult（示例）

```
AnalogHamiltonianSimulationQuantumTaskResult(
    task_metadata=TaskMetadata(
        braketSchemaHeader=BraketSchemaHeader(
            name='braket.task_result.task_metadata',
            version='1'
        ),
        id='arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef',
        shots=2,
```

```
        deviceId='arn:aws:braket:us-east-1::device/qpu/quera/Aquila',
        deviceParameters=None,
        createdAt='2022-10-25T20:59:10.788Z',
        endedAt='2022-10-25T21:00:58.218Z',
        status='COMPLETED',
        failureReason=None
    ),
    measurements=[
        ShotResult(
            status=<AnalogHamiltonianSimulationShotStatus.SUCCESS: 'Success'>,

            pre_sequence=array([1, 1, 1, 1]),
            post_sequence=array([0, 1, 1, 1])
        ),

        ShotResult(
            status=<AnalogHamiltonianSimulationShotStatus.SUCCESS: 'Success'>,

            pre_sequence=array([1, 1, 0, 1]),
            post_sequence=array([1, 0, 0, 0])
        )
    ]
)
```

JSON（示例）

```
{
    "braketSchemaHeader": {
        "name": "braket.task_result.analog_hamiltonian_simulation_task_result",
        "version": "1"
    },
    "taskMetadata": {
        "braketSchemaHeader": {
            "name": "braket.task_result.task_metadata",
            "version": "1"
        },
        "id": "arn:aws:braket:us-east-1:123456789012:quantum-task/12345678-90ab-
cdef-1234-567890abcdef",
        "shots": 2,
        "deviceId": "arn:aws:braket:us-east-1::device/qpu/quera/Aquila",

        "createdAt": "2022-10-25T20:59:10.788Z",
        "endedAt": "2022-10-25T21:00:58.218Z",
```

```
        "status": "COMPLETED"

    },
    "measurements": [
        {
            "shotMetadata": {"shotStatus": "Success"},
            "shotResult": {
                "preSequence": [1, 1, 1, 1],
                "postSequence": [0, 1, 1, 1]
            }
        },
        {
            "shotMetadata": {"shotStatus": "Success"},
            "shotResult": {
                "preSequence": [1, 1, 0, 1],
                "postSequence": [1, 0, 0, 0]
            }
        }
    ],
    "additionalMetadata": {
        "action": {...}
        "queraMetadata": {
            "braketSchemaHeader": {
                "name": "braket.task_result.quera_metadata",
                "version": "1"
            },
            "numSuccessfulShots": 100
        }
    }
}
```

主要字段

任务结果字段	类型	描述
measurements[].shotResult.preSequence	List[int]	每次拍摄的预序测量位（每个原子位点一个）：如果位点为空，则为 0，如果位点已填满，则为 1，在运行量子进化的脉冲序列之前测量
measurements[].shotResult.postSequence	List[int]	每次拍摄的序列后测量位：如果原子处于里德伯格状态或位点为空，

任务结果字段	类型	描述
		则为 0；如果原子处于基态，则为 1，在运行量子进化的脉冲序列末尾测量

元数据字段

任务结果字段	类型	描述
braketSchemaHeader.name	str	架构名称；必须是“braket.task_result.analog_hamiltonian_simulation_task_result.result”
braketSchemaHeader. 版本	str	架构版本
任务元数据。 braketSchemaHeader.name	str	架构名称；必须是“braket.task_result.task_metadata”
任务元数据。 braketSchemaHeader. 版本	str	架构版本
taskmetadata.id	str	量子任务 ID。对于 AWS 量子任务，这是量子任务 ARN。
taskMetadata.shots	int	量子任务的拍摄次数
taskmetadata.shots.deviceId	str	运行量子任务的设备的 ID。

任务结果字段	类型	描述
		对于 AWS 设备，这是设备 ARN。
taskMetadata.shots.createdAt	str	创建的时间戳；格式必须采用 ISO-8 RFC3339 601/ 字符串格式：mm: ss.sssz。YYYY-MM-DDTHH默认值：无。
taskMetadata.shots.endedAt	str	量子任务结束的时间戳；格式必须采用 ISO-8 RFC3339 601/ 字符串格式：mm: ss.sssz。YYYY-MM-DDTHH默认值：无。
taskMetadata.shots.status	str	量子任务的状态（已创建、已排队、正在运行、已完成、失败）。默认值：无。
taskMetadata.shots.failureReason	str	量子任务失败的原因。默认值：无。

任务结果字段	类型	描述
additionalMetadata.action	braket.ir.ahs.program_v1.Program	（请参阅 Braket AHS 程序架构部分）
其他元数据.action。 braketSchemaHeader.querametadata.name	str	架构名称；必须是“braket.task_result.quera_metadata”
其他元数据.action。 braketSchemaHeader.querametaData.version	str	架构版本
其他元数据.action。 numSuccessfulShots	int	完全成功拍摄的次数；必须等于请求的拍摄次数
measurements[].shotMetadata.shotStatus	int	拍摄状态（成功、部分成功、失败）；必须为“成功”

QuEra 设备属性架构

braket.device_schema.quera.quera_device_capabilities_v1。 QueraDeviceCapabilities（示例）

```
QueraDeviceCapabilities(
    service=DeviceServiceProperties(
        braketSchemaHeader=BraketSchemaHeader(
            name='braket.device_schema.device_service_properties',
            version='1'
            ),
            executionWindows=[
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.MONDAY: 'Monday'>,
                    windowStartHour=datetime.time(1, 0),
```

```
                    windowEndHour=datetime.time(23, 59, 59)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.TUESDAY: 'Tuesday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(12, 0)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.WEDNESDAY: 'Wednesday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(12, 0)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.FRIDAY: 'Friday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(23, 59, 59)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.SATURDAY: 'Saturday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(23, 59, 59)
                ),
                DeviceExecutionWindow(
                    executionDay=<ExecutionDay.SUNDAY: 'Sunday'>,
                    windowStartHour=datetime.time(0, 0),
                    windowEndHour=datetime.time(12, 0)
                )
            ],
            shotsRange=(1, 1000),
            deviceCost=DeviceCost(
                price=0.01,
                unit='shot'
            ),
            deviceDocumentation=
                DeviceDocumentation(
                    imageUrl='https://
a.b.cdn.console.awsstatic.com/59534b58c709fc239521ef866db9ea3f1aba73ad3ebcf60c23914ad8c5c5c878/
a6cfc6fca26cf1c2e1c6.png',
                    summary='Analog quantum processor based on neutral atom arrays',
                    externalDocumentationUrl='https://www.quera.com/aquila'
                ),
                deviceLocation='Boston, USA',
                updatedAt=datetime.datetime(2024, 1, 22, 12, 0,
 tzinfo=datetime.timezone.utc),
```

```
                getTaskPollIntervalMillis=None
    ),
    action={
        <DeviceActionType.AHS: 'braket.ir.ahs.program'>: DeviceActionProperties(
                version=['1'],
                actionType=<DeviceActionType.AHS: 'braket.ir.ahs.program'>
            )
    },
    deviceParameters={},
    braketSchemaHeader=BraketSchemaHeader(
        name='braket.device_schema.quera.quera_device_capabilities',
        version='1'
    ),
    paradigm=QueraAhsParadigmProperties(
        ...
        # See https://github.com/amazon-braket/amazon-braket-schemas-python/blob/main/
src/braket/device_schema/quera/quera_ahs_paradigm_properties_v1.py
        ...
    )
)
```

JSON（示例）

```
{
    "service": {
        "braketSchemaHeader": {
            "name": "braket.device_schema.device_service_properties",
            "version": "1"
        },
        "executionWindows": [
            {
                "executionDay": "Monday",
                "windowStartHour": "01:00:00",
                "windowEndHour": "23:59:59"
            },
            {
                "executionDay": "Tuesday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "12:00:00"
            },
            {
                "executionDay": "Wednesday",
                "windowStartHour": "00:00:00",
```

```
                "windowEndHour": "12:00:00"
            },
            {
                "executionDay": "Friday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "23:59:59"
            },
            {
                "executionDay": "Saturday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "23:59:59"
            },
            {
                "executionDay": "Sunday",
                "windowStartHour": "00:00:00",
                "windowEndHour": "12:00:00"
            }
        ],
        "shotsRange": [
            1,
            1000
        ],
        "deviceCost": {
            "price": 0.01,
            "unit": "shot"
        },
        "deviceDocumentation": {
            "imageUrl": "https://
a.b.cdn.console.awsstatic.com/59534b58c709fc239521ef866db9ea3f1aba73ad3ebcf60c23914ad8c5c5c878/
a6cfc6fca26cf1c2e1c6.png",
            "summary": "Analog quantum processor based on neutral atom arrays",
            "externalDocumentationUrl": "https://www.quera.com/aquila"
        },
        "deviceLocation": "Boston, USA",
        "updatedAt": "2024-01-22T12:00:00+00:00"
    },
    "action": {
        "braket.ir.ahs.program": {
            "version": [
                "1"
            ],
            "actionType": "braket.ir.ahs.program"
        }
    },
```

```
    "deviceParameters": {},
    "braketSchemaHeader": {
        "name": "braket.device_schema.quera.quera_device_capabilities",
        "version": "1"
    },
    "paradigm": {
        ...
        # See Aquila device page > "Calibration" tab > "JSON" page
        ...
    }
}
```

服务属性字段

服务属性字段	类型	描述
service.executionWindows[].executionDay	ExecutionDay	执行窗口的天数；必须是“每天”、“工作日”、“周末”、“星期一”、“星期二”、“星期三”、“星期四”、“星期五”、“星期六”或“星期日”
Service.executionWindows []。 windowStartHour	datetime.time	执行窗口开始时间的 UTC 24 小时格式
Service.executionWindows []。 windowEndHour	datetime.time	执行窗口结束时间的 UTC 24 小时格式
service.qpu_Capabilities.service.shotsRange	Tuple[int, int]	设备的最小和最大拍摄次数
service.qpu_capabilities.service.deviceCost.price	浮点数	设备价格（以美元计）
service.qpu_capabilities.service.deviceCost.unit	str	计费单位，例如：“分钟”、“小时”、“拍摄”、“任务”

元数据字段

元数据字段	类型	描述
action[].version	str	AHS 程序架构的版本
action[].actionType	ActionType	AHS 程序架构名称；必须是“braket.ir.ahs.program”
服务。 braketSchemaHeader.name	str	架构的名称；必须是“braket.device_schema.device_service_properties”
服务。 braketSchemaHeader. 版本	str	架构版本
service.deviceDocumentation.imageUrl	str	设备图片的 URL
service.deviceDocumentation.summary	str	设备的简要描述
服务设备文档。 externalDocumentationUrl	str	外部文档 URL
service.deviceLocation	str	设备的地理位置
service.updatedAt	datetime	上次更新设备属性的时间

使用 AWS Boto3

Boto3 是 Python 的 AWS 软件开发工具包。使用 Boto3，Python 开发者可以创建、配置和管理 AWS 服务，比如 Amazon Braket。Boto3 提供了面向对象的 API，以及对 Amazon Braket 的低级别访问。

按照 Boto3 快速入门指南中的说明进行操作，了解如何安装和配置 Boto3。

Boto3 提供了与 Amazon Braket Python SDK 配合使用的核心功能，可帮助您配置和运行量子任务。Python 客户总是需要安装 Boto3，因为这是核心实现。如果您想使用其他辅助方法，则还需要安装 Amazon Braket 软件开发工具包。

例如，当您调用 `CreateQuantumTask` 时，Amazon Braket SDK 会将请求提交给 Boto3，然后由 Boto3 调用 AWS API。

本节内容：

打开 Amazon Braket Boto3 客户端

要实现 Boto3 与 Amazon Braket 的配合使用，您必须导入 Boto3，然后定义用于连接 Amazon Braket API 的客户端。在以下示例中，Boto3客户端名为 `braket`。

```
import boto3
import botocore

braket = boto3.client("braket")
```

> **Note**
>
> 支@@ 架支撑 IPv6。如果您使用的是 IPv6仅限网络或希望确保您的工作负载使用 IPv6 流量，请使用双栈和 FIPS 端点指南中概述的双栈端点。

既然您已经建立了 `braket` 客户端，就可以通过 Amazon Braket 服务提出请求和处理回复了。您可以在 API 参考中获取有关请求和响应数据的更多详细信息。

以下示例演示了如何处理设备和量子任务。

- 搜索设备
- 检索设备
- 创建量子任务
- 检索量子任务
- 搜索量子任务
- 取消量子任务

搜索设备

- search_devices(**kwargs)

使用指定的过滤器搜索设备。

```
# Pass search filters and optional parameters when sending the
# request and capture the response
response = braket.search_devices(filters=[{
    'name': 'deviceArn',
    'values': ['arn:aws:braket:::device/quantum-simulator/amazon/sv1']
}], maxResults=10)

print(f"Found {len(response['devices'])} devices")

for i in range(len(response['devices'])):
    device = response['devices'][i]
    print(device['deviceArn'])
```

检索设备

- get_device(deviceArn)

检索在 Amazon Braket 中可用的设备。

```
# Pass the device ARN when sending the request and capture the repsonse
response = braket.get_device(deviceArn='arn:aws:braket:::device/quantum-simulator/
amazon/sv1')

print(f"Device {response['deviceName']} is {response['deviceStatus']}")
```

创建量子任务

- create_quantum_task(**kwargs)

创建量子任务。

```
# Create parameters to pass into create_quantum_task()
kwargs = {
```

```
    # Create a Bell pair
    'action': '{"braketSchemaHeader": {"name": "braket.ir.jaqcd.program", "version":
 "1"}, "results": [], "basis_rotation_instructions": [], "instructions": [{"type": "h",
 "target": 0}, {"type": "cnot", "control": 0, "target": 1}]}',
    # Specify the SV1 Device ARN
    'deviceArn': 'arn:aws:braket:::device/quantum-simulator/amazon/sv1',
    # Specify 2 qubits for the Bell pair
    'deviceParameters': '{"braketSchemaHeader": {"name":
 "braket.device_schema.simulators.gate_model_simulator_device_parameters",
 "version": "1"}, "paradigmParameters": {"braketSchemaHeader": {"name":
 "braket.device_schema.gate_model_parameters", "version": "1"}, "qubitCount": 2}}',
    # Specify where results should be placed when the quantum task completes.
    # You must ensure the S3 Bucket exists before calling create_quantum_task()
    'outputS3Bucket': 'amazon-braket-examples',
    'outputS3KeyPrefix': 'boto-examples',
    # Specify number of shots for the quantum task
    'shots': 100
}

# Send the request and capture the response
response = braket.create_quantum_task(**kwargs)

print(f"Quantum task {response['quantumTaskArn']} created")
```

检索量子任务

- `get_quantum_task(quantumTaskArn)`

检索指定的量子任务。

```
# Pass the quantum task ARN when sending the request and capture the response
response = braket.get_quantum_task(quantumTaskArn='arn:aws:braket:us-
west-1:123456789012:quantum-task/ce78c429-cef5-45f2-88da-123456789012')

print(response['status'])
```

搜索量子任务

- `search_quantum_tasks(**kwargs)`

搜索与指定过滤值匹配的量子任务。

```
# Pass search filters and optional parameters when sending the
# request and capture the response
response = braket.search_quantum_tasks(filters=[{
    'name': 'deviceArn',
    'operator': 'EQUAL',
    'values': ['arn:aws:braket:::device/quantum-simulator/amazon/sv1']
}], maxResults=25)

print(f"Found {len(response['quantumTasks'])} quantum tasks")

for n in range(len(response['quantumTasks'])):
    task = response['quantumTasks'][n]
    print(f"Quantum task {task['quantumTaskArn']} for {task['deviceArn']} is
 {task['status']}")
```

取消量子任务

- `cancel_quantum_task(quantumTaskArn)`

取消指定的量子任务。

```
# Pass the quantum task ARN when sending the request and capture the response
response = braket.cancel_quantum_task(quantumTaskArn='arn:aws:braket:us-
west-1:123456789012:quantum-task/ce78c429-cef5-45f2-88da-123456789012')

print(f"Quantum task {response['quantumTaskArn']} is {response['cancellationStatus']}")
```

为 Boto3 和 Braket SDK AWS CLI 配置配置文件

除非您另有明确说明，否则 Amazon Braket SDK 依赖于默认 AWS CLI 证书。我们建议您在托管 Amazon Braket Notebook 上运行时保留默认设置，因为您必须提供有权启动 Notebook 实例的 IAM 角色。

或者，如果您在本地运行代码（例如在 Amazon EC2 实例上），则可以建立命名 AWS CLI 配置文件。您可以为每个配置文件指定不同的权限集，而不必定期覆盖默认配置文件。

本节简要说明了如何配置这样的 CLI `profile` 以及如何将该配置文件合并到 Amazon Braket 中，以便使用该配置文件的权限进行 API 调用。

本节内容：

- 步骤 1：配置本地 C AWS LI profile
- 步骤 2：建立 Boto3 会话对象
- 第 3 步：将 Boto3 会话合并到 Braket 中 AwsSession

步骤 1：配置本地 C AWS LI **profile**

解释如何创建用户以及如何配置非默认配置文件已超出本文档的讨论范围。有关这些主题的更多信息，请参阅：

- 入门
- 配置 AWS CLI 要使用 AWS IAM Identity Center

要使用 Amazon Braket，您必须向该用户以及相关的 CLI `profile` 提供必要的 Braket 权限。例如，您可以附加AmazonBraketFullAccess策略。

步骤 2：建立 Boto3 会话对象

要建立 Boto3 会话对象，请使用以下代码示例。

```
from boto3 import Session

# Insert CLI profile name here
boto_sess = Session(profile_name=`profile`)
```

> **Note**
>
> 如果预期的 API 调用具有基于区域的限制，而这些限制与您的 `profile` 默认区域不一致，则可以为 Boto3 会话指定一个区域，如以下示例所示。

```
# Insert CLI profile name _and_ region
boto_sess = Session(profile_name=`profile`, region_name=`region`)
```

对于指定为的参数`region`，请替换一个与 Amazon Braket 可用的值相对应的值`us-east-1`，例如`us-west-1`、等。 AWS 区域

第 3 步：将 Boto3 会话合并到 Braket 中 AwsSession

以下示例说明如何初始化 Boto3 Braket 会话并在该会话中实例化设备。

```
from braket.aws import AwsSession, AwsDevice

# Initialize Braket session with Boto3 Session credentials
aws_session = AwsSession(boto_session=boto_sess)

# Instantiate any Braket QPU device with the previously initiated AwsSession
sim_arn = 'arn:aws:braket:::device/quantum-simulator/amazon/sv1'
device = AwsDevice(sim_arn, aws_session=aws_session)
```

设置完成后，您可以向该实例化的 AwsDevice 对象提交量子任务（例如，通过调用 device.run(…) 命令）。该设备进行的所有 API 调用都可以使用与您之前指定为 profile 的 CLI 配置文件关联的 IAM 凭证。

使用 Amazon Braket 测试您的量子任务

Amazon Braket 提供各种高性能量子电路模拟器，可帮助您在实际的量子硬件上运行量子算法之前对其进行测试和验证。这些模拟器处理复杂的底层软件和基础设施以及 Amazon Elastic Compute Cloud (Amazon EC2) 集群，从而减轻了在传统高性能计算（HPC）基础设施上模拟量子电路的负担。这些资源可专注于量子应用的开发和优化工作。

借助 Braket 的模拟器，您可以彻底测试您的量子电路和算法，不受物理量子设备的约束和限制。这样，您就能够探索各种量子计算概念，从基本的量子门和电路到更先进的量子算法和错误缓解技术。

Braket SDK 简化了向模拟器提交量子任务的过程，可控制模拟参数，如镜头数量和噪声模型，从而更好地了解量子算法的行为。您还可以使用 Amazon Braket Hybrid Jobs 功能将经典计算元素和量子计算元素结合起来，从而进一步扩大测试和验证的范围。

通过在 Braket 的模拟器上彻底测试您的量子任务，您可以获得宝贵的见解，完善算法，并确保其正确性，然后再将其部署到真正的量子硬件上。这有助于缩短开发时间，最大限度地减少错误，并最终加快您在量子计算领域的进展。

本节内容：

向模拟器提交量子任务

Amazon Braket 允许访问多个模拟器，这些模拟器可以测试您的量子任务。您可以单独提交量子任务，也可以运行多个程序。

模拟器

- 密度矩阵模拟器，DM1:`arn:aws:braket:::device/quantum-simulator/amazon/dm1`
- 状态向量模拟器，SV1:`arn:aws:braket:::device/quantum-simulator/amazon/sv1`
- 张量网络模拟器，TN1:`arn:aws:braket:::device/quantum-simulator/amazon/tn1`
- 本地模拟器：`LocalSimulator()`

> **Note**
>
> 您可以取消按需模拟器CREATED状态下的 QPUs 量子任务。您可以尽最大努力取消该QUEUED州的量子任务，用于按需模拟器和. QPUs 请注意，在 QPU 可用性窗口期间，QPU QUEUED 量子任务不太可能成功取消。

本节内容：

局部状态向量模拟器 (`braket_sv`)

本地状态向量模拟器 (`braket_sv`) 是在您的环境中本地运行的 Amazon Braket SDK 的一部分。它非常适合在小型电路（最多 25 个 qubits）上进行快速原型设计，具体取决于您的 Braket Notebook 实例或本地环境的硬件规格。

本地模拟器支持 Amazon Braket SDK 中的所有门，但是 QPU 设备支持的子集较小。您可以在设备属性中找到设备支持的门。

> **Note**
>
> 本地模拟器支持高级 OpenQASM 功能，QPU 设备或其他模拟器可能不支持这些功能。有关支持的功能的更多信息，请参阅 OpenQASM 本地模拟器 Notebook 中提供的示例。

有关如何使用模拟器的更多信息，请参阅 Amazon Braket 示例。

局部密度矩阵模拟器 (braket_dm)

本地密度矩阵模拟器 (braket_dm) 是在您的环境中本地运行的 Amazon Braket SDK 的一部分。它非常适合在有噪声（最多 12 个 qubits）的小型电路上进行快速原型设计，具体取决于您的 Braket Notebook 实例或本地环境的硬件规格。

您可以使用门噪声操作（如位翻转和去极化误差）从头开始构建常见的噪声电路。您还可以将噪声运算应用于现有电路的特定 qubits 和门，这些电路将在有噪声和无噪声的情况下运行。

给定指定数量的 shots，braket_dm 本地模拟器可以提供以下结果：

- 约化密度矩阵：Shots = 0

> **Note**
>
> 本地模拟器支持高级 OpenQASM 功能，QPU 设备或其他模拟器可能不支持这些功能。有关支持的功能的更多信息，请参阅 OpenQASM 本地模拟器 Notebook 中提供的示例。

要了解有关局部密度矩阵模拟器的更多信息，请参阅 Braket 噪声模拟器入门示例。

本地 AHS 模拟器 (braket_ahs)

本地 AHS（模拟哈密顿模拟）模拟器 (braket_ahs) 是在您的环境中本地运行的 Amazon Braket SDK 的一部分。它可以用来模拟 AHS 程序的结果。它非常适合在小型寄存器（最多 10-12 个原子）上进行原型设计，具体取决于您的 Braket Notebook 实例或本地环境的硬件规格。

本地模拟器支持具有一个均匀驱动场、一个（非均匀）移位场和任意原子排列的 AHS 程序。有关详细信息，请参阅 Braket AHS 类和 Braket AHS 程序架构。

要了解有关本地 AHS 模拟器的更多信息，请参阅 Hello AHS：运行您的第一个模拟哈密顿模拟页面和模拟哈密顿模拟示例 Notebook。

状态向量模拟器 (SV1)

SV1 是一款按需、高性能、通用状态向量模拟器。它可以模拟多达 34 个 qubits 的电路。根据所用门的类型和其他因素，您可以预计 34-qubit、密集的方形电路（电路深度 = 34）大约需要 1-2 小时才能完成。带 all-to-all栅极的电路非常适合SV1。它以诸如完整状态向量或振幅数组之类的形式返回结果。

SV1 的最大运行时为 6 小时。它的默认并发量子任务数量为 35 个，并发量子任务数量最多为 100 个（us-west-1 和 eu-west-2 中为 50 个）。

SV1 结果

SV1 在给定指定数量的情况下，可以提供以下结果 shots：

- 示例：Shots > 0
- 期望：Shots >= 0
- 方差：Shots >= 0
- 概率: Shots > 0
- 振幅：Shots = 0
- 伴随梯度：Shots = 0

有关结果的更多信息，请参阅结果类型。

SV1 始终可用，它可以按需运行您的电路，并且可以并行运行多个电路。运行时随操作数线性缩放，随 qubits 数指数级缩放。shots 的数量对运行时的影响很小。要了解更多信息，请访问比较模拟器。

模拟器支持 Braket SDK 中的所有门，但是 QPU 设备支持的子集较小。您可以在设备属性中找到设备支持的门。

密度矩阵模拟器 (DM1)

DM1 是一款按需提供的高性能密度矩阵模拟器。它可以模拟多达 17 个 qubits 的电路。

DM1 最大运行时为 6 小时，默认为 35 个并发量子任务，最多 50 个并发量子任务。

DM1 结果

DM1 在给定指定数量的情况下，可以提供以下结果 shots：

- 示例：Shots > 0
- 期望：Shots >= 0
- 方差：Shots >= 0
- 概率: Shots > 0
- 降低密度矩阵：Shots = 0，最大值为 8 个 qubits

有关这些类型的更多信息，请参阅结果类型。

DM1 始终可用，它可以按需运行您的电路，并且可以并行运行多个电路。运行时随操作数线性缩放，随 qubits 数指数级缩放。shots 的数量对运行时的影响很小。要了解更多信息，请参阅比较模拟器。

噪声门和局限性

```
AmplitudeDamping
    Probability has to be within [0,1]
BitFlip
    Probability has to be within [0,0.5]
Depolarizing
    Probability has to be within [0,0.75]
GeneralizedAmplitudeDamping
    Probability has to be within [0,1]
PauliChannel
    The sum of the probabilities has to be within [0,1]
Kraus
    At most 2 qubits
    At most 4 (16) Kraus matrices for 1 (2) qubit
PhaseDamping
    Probability has to be within [0,1]
PhaseFlip
    Probability has to be within [0,0.5]
TwoQubitDephasing
    Probability has to be within [0,0.75]
TwoQubitDepolarizing
    Probability has to be within [0,0.9375]
```

张量网络模拟器 (TN1)

TN1是一款按需、高性能、张量网络模拟器。 TN1可以模拟某些电路类型，最大为 50qubits，电路深度等于 100 或更小。 TN1对于稀疏电路、带有局部门的电路以及其他具有特殊结构的电路（例如量子傅里叶变换 (QFT) 电路）特别强大。 TN1分两个阶段运作。首先，排练阶段会尝试为您的电路确定有效的计算路径，因而 TN1 可以估计下一阶段（称为收缩阶段）的运行时。如果估计的收缩时间超过 TN1 模拟运行时限制，TN1 不会尝试收缩。

TN1 的运行时限制为 6 小时。它限制为最多 10 个（在 eu-west-2 中为 5 个）并发量子任务。

TN1 结果

收缩阶段包括一系列矩阵乘法。一系列乘法一直持续到达到结果或确定无法得出结果为止。

注意：Shots 必须大于 0。

结果类型包括：

- 样本
- 期望
- 方差

有关结果的更多信息，请参阅结果类型。

TN1 始终可用，它可以按需运行您的电路，并且可以并行运行多个电路。要了解更多信息，请参阅比较模拟器。

模拟器支持 Braket SDK 中的所有门，但是 QPU 设备支持的子集较小。您可以在设备属性中找到设备支持的门。

访问 Amazon Braket GitHub 存储库获取TN1 示例笔记本，以帮助您入门。TN1

使用 TN1 的最佳实践

- 避免 all-to-all回路。
- 用少量 shots 测试一个新的电路或一类电路，以了解电路的 TN1“硬度”。
- 将大型 shot 模拟拆分为多个量子任务。

嵌入式模拟器简介

嵌入式模拟器通过将模拟直接嵌入到算法代码中来运行。此外，它包含在同一个容器中，并直接在混合作业实例上运行模拟。这种方法对于消除通常与模拟和远程设备之间的通信相关的瓶颈非常有用。嵌入式模拟器通过将所有计算保持在单一、有凝聚力的环境中，可以大大降低内存需求并减少实现目标结果所需的电路执行次数。与依赖远程模拟的传统设置相比，这可以显著提高性能，通常提高九倍或更多。有关嵌入式模拟器如何提高性能和简化混合作业的更多信息，请参阅使用 Amazon Braket Hybrid Jobs 文档页面。

PennyLane的闪电模拟器

你可以在 Brak PennyLane et 上使用闪电模拟器作为 Braket 上的嵌入式模拟器。借助 PennyLane闪电模拟器，您可以使用高级梯度计算方法（例如伴随微分）来更快地评估梯度。lightning.qubit 模拟器可通过 Brak NBIs et 作为设备使用，也可以作为嵌入式模拟器使用，而 lightning.gpu 模拟器需要作为带有 GPU 实例的嵌入式模拟器运行。有关使用 lightning.gpu 的示例，请参阅 Braket Hybrid Jobs Notebook 中的嵌入式模拟器。

比较 Amazon Braket 模拟器

本节通过描述一些概念、限制和使用案例，帮助您选择最适合您的量子任务的 Amazon Braket 模拟器。

在本地模拟器和按需模拟器 (SV1, TN1, DM1) 之间进行选择

本地模拟器的性能取决于托管本地环境的硬件，如用于运行模拟器的 Braket Notebook 实例。按需模拟器在 AWS 云端运行，旨在超越典型的本地环境。按需模拟器针对较大的电路进行了优化，但是每个量子任务或批量子任务会增加一些延迟开销。如果涉及许多量子任务，这可能意味着需要权衡取舍。鉴于这些一般性能特征，以下指导可以帮助您选择如何运行模拟，包括带有噪声的模拟。

对于模拟：

- 当使用量少于 18 个 qubits 时，请使用本地模拟器。
- 当使用量为 18-24 个 qubits 时，请根据工作负载选择模拟器。
- 当使用量超过 24 个 qubits 时，请使用按需模拟器。

对于噪声模拟：

- 当使用量少于 9 个 qubits 时，请使用本地模拟器。
- 当使用量为 9-12 个 qubits 时，请根据工作负载选择模拟器。
- 当使用数超过 12 个 qubits 时，请使用 DM1。

什么是状态向量模拟器？

SV1 是一个通用状态向量模拟器。它存储着量子态的全波函数，并按顺序将门运算应用于该状态。它存储了所有可能性，即使是极不可能的也是如此。SV1 模拟器执行量子任务的运行时随着电路中门的数量而线性增加。

什么是密度矩阵模拟器？

DM1 用噪声模拟量子电路。它存储着系统的全密度矩阵，并按顺序应用电路的门和噪声运算。最终的密度矩阵包含有关电路运行后量子态的完整信息。运行时通常随操作数线性扩展，随 qubits 数呈指数级扩展。

什么是张量网络模拟器？

TN1 将量子电路编码成结构化图。

- 图的节点由量子门或 qubits 组成。
- 图形的边缘表示门之间的连接。

由于这种结构，TN1 可以为相对较大且复杂的量子电路找到模拟解。

TN1 需要两个阶段

通常，TN1 采用两阶段方法来模拟量子计算。

- 排练阶段：在这个阶段，TN1 想出一种高效遍历图表的方法，包括访问每个节点，这样您就可以获得您想要的测量结果。作为客户，您看不到此阶段，因为 TN1 同时为您执行两个阶段。它完成了第一阶段，并根据实际限制决定是否单独执行第二阶段。模拟开始后，您对该决定没有任何意见。
- 收缩阶段：此阶段类似于经典计算机中计算的执行阶段。该阶段包括一系列矩阵乘法。这些乘法的顺序对计算难度有很大的影响。因此，首先要完成排练阶段，以便在图中找到最有效的计算路径。在排练阶段找到收缩路径后，TN1 将电路的门收缩在一起以生成模拟结果。

TN1 图表类似于地图

打个比方，您可以将底层 TN1 图表与城市街道进行比较。在采用规划网格的城市中，使用地图可以很容易找到通往目的地的路线。在一个无规划街道、街道名称重复等的城市中，通过查看地图可能很难找到通往目的地的路线。

如果 TN1 没有进行排练阶段，那就像在城市的街道上漫步寻找目的地，而不是先看地图。就步行时间而言，花更多的时间看地图确实可以获得回报。同样，排练阶段也提供了有价值的信息。

您可能会说，TN1 对它所穿越的底层电路的结构有一定的“意识”。它在排练阶段获得了这种意识。

问题类型最适合每种类型的模拟器

SV1 非常适合于主要依赖于具有一定数量的 qubits 和门的任何类别的问题。通常，所需时间会随着门数的增加而呈线性增长，而不取决于 shots 的数量。SV1 通常比 28 个 qubits 以下的电路 TN1 快。

SV1 对于较大的 qubit 数字，可能会变慢，因为它实际上模拟了所有可能性，即使是极不可能的情况下也是如此。它无法确定可能出现哪些结果。因此，要进行 30-qubit 评估，SV1 必须计算 2^30 个配置。受内存和存储限制，Amazon Braket SV1 模拟器的限值为 34 个 qubits，是一个实际限制。您可以这样想：每当您向 qubit 中添加一个 SV1 时，问题就会变得困难一倍。

对于许多类问题，由于 TN1 利用图结构，与 SV1 相比，TN1 可以在现实时间内评估更大的电路。它本质上是从一开始就跟踪解决方案的演变，并且仅保留有助于高效遍历的配置。换句话说，它保存配置以创建矩阵乘法的顺序，从而简化计算过程。

qubits 和门的数量对 TN1 来说很重要，但图表结构更重要。例如，TN1 非常擅长评估门为短距离的电路（图表）（也就是说，每个 qubit 门仅通过门连接到其最近的邻居 qubits），以及连接（或门）具有相似范围的电路（图表）。TN1 的典型范围是让每个 qubit 仅与 5 个 qubits 之外的另外 qubits 交谈。如果结构的大部分可以分解为更简单的关系，如这些关系，这些关系可以用更多、更小或更统一的矩阵表示，则 TN1 可以有效地执行评估。

TN1 的限制

TN1 可能比 SV1 慢一些，取决于图表的结构复杂度。对于某些图表，由于以下两个原因之一，TN1 会在排练阶段结束后终止模拟并显示 FAILED 的状态：

- 无法找到路径：如果图形太复杂，则很难找到一条好的遍历路径，模拟器就会放弃计算。TN1 无法进行收缩。您可能会看到类似于以下内容的错误消息：`No viable contraction path found.`
- 收缩阶段太困难：在某些图表中，TN1 可以找到遍历路径，但是评估起来非常漫长且非常耗时。在这种情况下，收缩非常昂贵，以至于成本高到令人望而却步，而是 TN1 在排练阶段之后退出。您可能会看到类似于以下内容的错误消息：`Predicted runtime based on best contraction path found exceeds TN1 limit.`

> **Note**
>
> 即使没有进行收缩且您会看到 FAILED 状态，您也需要支付排练阶段的 TN1 费用。

预测的运行时也取决于 shot 计数。在最坏的情况下，TN1 收缩时间线性地取决于 shot 的计数。该电路可以用更少的 shots 进行收缩。例如，您可以提交一个包含 100 个 shots 的量子任务，该任务 TN1 决定不可收缩，但是如果您只有 10 的量子任务重新提交，则会继续收缩。在这种情况下，要获得 100 个样本，您可以为同一电路提交 10 个 shots 的 10 个量子任务，最后合并结果。

作为最佳实践，我们建议您始终使用几个 shots（如 10）来测试您的电路或电路等级，以了解您的电路对 TN1 而言有多难，然后再继续使用更大的 shots 数量。

> **Note**
>
> 形成收缩阶段的一系列乘法从小的 NxN 矩阵开始。例如，2-qubit 门需要一个 4x4 矩阵。在被判定为太困难的收缩期间所需的中间矩阵是巨大的。这样的计算需要几天才能完成。这就是 Amazon Braket 不尝试极其复杂的收缩的原因。

并发

所有 Braket 模拟器都能使您同时运行多个回路。并发限制因模拟器和区域而异。有关并发限制的更多信息，请参阅配额页面。

Amazon Braket 上的量子任务示例

本节介绍运行示例量子任务的各个阶段，从选择设备到查看结果。作为 Amazon Braket 的最佳实践，我们建议您首先在模拟器上运行电路，如 SV1。

本节内容：

- 指定设备
- 提交量子任务示例
- 提交参数化任务
- 指定 shots
- 轮询结果
- 示例：查看结果

指定设备

首先，为您的量子任务选择并指定设备。此示例显示了如何选择模拟器 SV1。

```
from braket.aws import AwsDevice

# Choose the on-demand simulator to run the circuit
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
```

您可以按如下方式查看此设备的某些属性：

```
print(device.name)
for iter in device.properties.action['braket.ir.jaqcd.program']:
    print(iter)
```

```
SV1
('version', ['1.0', '1.1'])
('actionType', 'braket.ir.jaqcd.program')
('supportedOperations', ['ccnot', 'cnot', 'cphaseshift', 'cphaseshift00',
 'cphaseshift01', 'cphaseshift10', 'cswap', 'cy', 'cz', 'ecr', 'h', 'i', 'iswap',
```

```
 'pswap', 'phaseshift', 'rx', 'ry', 'rz', 's', 'si', 'swap', 't', 'ti', 'unitary', 'v',
 'vi', 'x', 'xx', 'xy', 'y', 'yy', 'z', 'zz'])
('supportedResultTypes', [ResultType(name='Sample', observables=['x', 'y', 'z', 'h',
 'i', 'hermitian'], minShots=1, maxShots=100000), ResultType(name='Expectation',
 observables=['x', 'y', 'z', 'h', 'i', 'hermitian'], minShots=0, maxShots=100000),
 ResultType(name='Variance', observables=['x', 'y', 'z', 'h', 'i', 'hermitian'],
 minShots=0, maxShots=100000), ResultType(name='Probability', observables=None,
 minShots=1, maxShots=100000), ResultType(name='Amplitude', observables=None,
 minShots=0, maxShots=0)])
('disabledQubitRewiringSupported', None)
```

提交量子任务示例

提交要在按需模拟器上运行的量子任务示例。

```
from braket.circuits import Circuit, Observable

# Create a circuit with a result type
circ = Circuit().rx(0, 1).ry(1, 0.2).cnot(0, 2).variance(observable=Observable.Z(),
 target=0)
# Add another result type
circ.probability(target=[0, 2])

# Set up S3 bucket (where results are stored)
my_bucket = "amazon-braket-s3-demo-bucket"  # The name of the bucket
my_prefix = "your-folder-name"  # The name of the folder in the bucket
s3_location = (my_bucket, my_prefix)

# Submit the quantum task to run
my_task = device.run(circ, s3_location, shots=1000, poll_timeout_seconds=100,
 poll_interval_seconds=10)
# The positional argument for the S3 bucket is optional if you want to specify a bucket
 other than the default

# Get results of the quantum task
result = my_task.result()
```

该 `device.run()` 命令通过 `CreateQuantumTask` API 创建量子任务。在短暂的初始化时间后，量子任务将进行排队，直到有能力在设备上运行量子任务为止。在本例中，设备为 SV1。设备完成计算后，Amazon Braket 会将结果写入调用中指定的 Amazon S3 位置。除本地模拟器 `s3_location` 之外，所有设备都需要有位置参数。

> **Note**
>
> Braket 量子任务动作的大小限制在 3MB 以内。

提交参数化任务

Amazon Braket 按需模拟器和本地模拟器， QPUs 还支持在任务提交时指定免费参数的值。您可以通过使用 `device.run()` 的 `inputs` 参数来执行此操作，如以下示例所示。`inputs` 必须是字符串浮点对的字典，其中键是参数名。

参数化编译可以提高在某些情况下执行参数电路的性能。 QPUs将参数电路作为量子任务提交给支持的 QPU 时，Braket 将编译电路一次，然后缓存结果。对于同一电路的后续参数更新，无需重新编译，从而缩短了使用相同电路的任务的运行时。编译电路时，Braket 会自动使用硬件提供商提供的更新的校准数据，以确保获得高质量的结果。

> **Note**
>
> 除脉冲电平程序外，所有基于门的 QPUs 超导模式都支持参数化编译。Rigetti Computing

```
from braket.circuits import Circuit, FreeParameter, Observable

# Create the free parameters
alpha = FreeParameter('alpha')
beta = FreeParameter('beta')

# Create a circuit with a result type
circ = Circuit().rx(0, alpha).ry(1, alpha).cnot(0, 2).xx(0, 2, beta)
circ.variance(observable=Observable.Z(), target=0)

# Add another result type
circ.probability(target=[0, 2])

# Submit the quantum task to run
my_task = device.run(circ, inputs={'alpha': 0.1, 'beta': 0.2}, shots=100)
```

指定 shots

shots 参数指的是所需的测量 shots 的次数。诸如 SV1 之类的模拟器支持两种模拟模式。

- 对于 shots = 0，模拟器执行精确模拟，返回所有结果类型的真实值。（对 TN1 暂不可用。）
- 对于的非零值shots，仿真器会从输出分布中采样以模拟真实的shot噪声。 QPUsQPU 设备仅允许 shots > 0。

有关每个量子任务的最大拍摄次数的信息，请参阅 Braket 配额。

轮询结果

执行 `my_task.result()` 时，SDK 开始使用您在创建量子任务时定义的参数轮询结果：

- `poll_timeout_seconds` 是在按需模拟器和/或 QPU 设备上运行量子任务时，在量子任务超时之前对其进行轮询的秒数。默认值为 43.2 万秒（5 天）。
- 注意：对于 Rigetti 和 IonQ 之类的 QPU 设备，我们建议您留出几天时间。如果您的轮询超时时间太短，则可能无法在轮询时间内返回结果。例如，当 QPU 不可用时，会返回本地超时错误。
- `poll_interval_seconds` 是轮询量子任务的频率。它指定了在按需模拟器和 QPU 设备上运行量子任务时，您多久调用 Braket API 以获取状态。默认值为 1 秒。

这种异步执行便于与并非总是可用的 QPU 设备进行交互。例如，在常规维护时段内，设备可能不可用。

返回的结果包含一系列与量子任务相关的元数据。您可以使用以下命令检查测量结果：

```
print('Measurement results:\n', result.measurements)
print('Counts for collapsed states:\n', result.measurement_counts)
print('Probabilities for collapsed states:\n', result.measurement_probabilities)
```

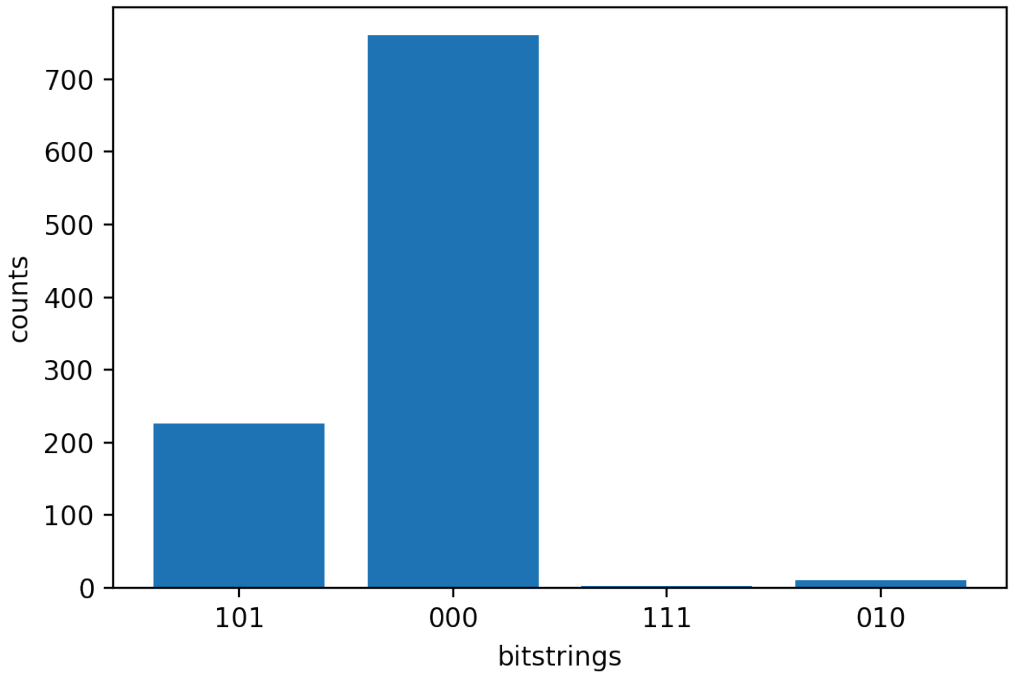
```
Measurement results:
 [[1 0 1]
 [0 0 0]
 [0 0 0]
 ...
 [0 0 0]
 [0 0 0]
 [1 0 1]]
Counts for collapsed states:
 Counter({'000': 766, '101': 220, '010': 11, '111': 3})
Probabilities for collapsed states:
 {'101': 0.22, '000': 0.766, '010': 0.011, '111': 0.003}
```

示例：查看结果

由于您还指定了 ResultType，您可以查看返回的结果。结果类型按添加到电路的添加顺序显示这些类型。

```
print('Result types include:\n', result.result_types)
print('Variance=', result.values[0])
print('Probability=', result.values[1])

# Plot the result and do some analysis
import matplotlib.pyplot as plt
plt.bar(result.measurement_counts.keys(), result.measurement_counts.values())
plt.xlabel('bitstrings')
plt.ylabel('counts')
```

```
Result types include:
 [ResultTypeValue(type=Variance(observable=['z'], targets=[0], type=<Type.variance:
 'variance'>), value=0.693084), ResultTypeValue(type=Probability(targets=[0, 2],
 type=<Type.probability: 'probability'>), value=array([0.777, 0.   , 0.   , 0.223]))]
Variance= 0.693084
Probability= [0.777 0.    0.    0.223]
Text(0, 0.5, 'counts')
```



使用本地模拟器测试量子任务

您可以将量子任务直接发送到本地模拟器，以进行快速原型设计和测试。此模拟器在本地环境中运行，因此您无需指定一个 Amazon S3 位置。结果是在您的会话中直接计算出来的。要在本地模拟器上运行量子任务，您只需指定 shots 参数即可。

> **Note**
>
> 本地模拟器可以处理的 qubits 的执行速度和最大数量取决于 Amazon Braket Notebook 实例类型或您的本地硬件规格。

以下命令完全相同，用于实例化状态向量（无噪声）本地模拟器。

```
# Import the LocalSimulator module
from braket.devices import LocalSimulator

# The following are identical commands
device = LocalSimulator()
device = LocalSimulator("default")
device = LocalSimulator(backend="default")
device = LocalSimulator(backend="braket_sv")
```

然后使用以下命令运行量子任务。

```
my_task = device.run(circ, shots=1000)
```

要实例化局部密度矩阵（噪声）模拟器，客户需要按如下方式更改后端。

```
# Import the LocalSimulator module
from braket.devices import LocalSimulator

device = LocalSimulator(backend="braket_dm")
```

在本地模拟器上测量特定的量子比特

局部状态向量模拟器和局部密度矩阵模拟器支持运行电路，在这些电路中，可以测量电路的量子比特子集，这通常称为部分测量。

例如，在以下代码中，您可以创建一个双量子比特电路，并且只能通过在电路末端添加一条带有目标量子比特的 measure 指令来测量第一个量子比特。

```
# Import the LocalSimulator module
from braket.devices import LocalSimulator

# Use the local simulator device
device = LocalSimulator()

# Define a bell circuit and only measure
circuit = Circuit().h(0).cnot(0, 1).measure(0)

# Run the circuit
task = device.run(circuit, shots=10)

# Get the results
result = task.result()

# Print the measurement counts for qubit 0
print(result.measurement_counts)
```

本地量子设备模拟器

借助 Amazon Braket 的本地模拟器工具，您可以在本地模拟逐字量子程序，然后再在实际的量子硬件上运行它们。模拟器使用设备校准数据来验证逐字电路，这样您就可以尽早发现兼容性问题了。

此外，本地模拟器通过以下过程模拟量子硬件噪声：

- 使用设备校准数据构造噪声模型
- 对电路中的每个门施加去极化噪声
- 在电路末端应用读出错误
- 使用局部密度矩阵模拟器模拟噪声电路

有关使用本地仿真器的更多信息，请参阅存储库中的 Amazon Braket 上的逐字电路本地仿真。amazon-braket-examples GitHub

本节内容：

本地模拟的好处

- 使用实时或历史校准数据根据设备限制对逐字电路进行验证。
- 在向量子硬件提交任务之前，请先调试问题。
- 将无噪声和噪声模拟与硬件结果进行比较，了解噪声效果。
- 简化开发噪声感知量子算法的工作流程。

创建本地模拟器

本地量子设备模拟器可以直接从量子设备或一组设备属性中创建。直接模拟设备时，模拟器使用实例化设备的最新校准数据。下面代码示例演示了如何直接仿真 Rigetti's Ankaa-3 设备。

```
from braket.aws.aws_device import AwsDevice

ankaa3 = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
ankaa3_emulator = ankaa3.emulator()
```

以下示例显示如何使用一组 JSON 格式的 Ankaa-3 设备属性创建本地设备模拟器。

```
from braket.aws import AwsDevice
from braket.emulation.local_emulator import LocalEmulator
import json

# Instantiate the device
ankaa3 = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")
ankaa3_properties = ankaa3.properties

# Put the Ankaa-3 properties in a file named ankaa3_device_properties.json
with open("ankaa3_device_properties.json", "w") as f:
    json.dump(ankaa3_properties.json(), f)

# Load the json into the ankaa3_data_json variable
with open("ankaa3_device_properties.json", "r") as json_file:
    ankaa3_data_json = json.load(json_file)

# Create the Ankaa-3 local emulator from the json file you created
ankaa3_emulator = LocalEmulator.from_json(ankaa3_data_json)
```

您可以通过以下方式自定义示例：

- 使用来自其他 QPU 设备的属性
- 为模拟器指定不同的 JSON 文件
- 在实例化模拟器之前更改设备属性的值

使用 Amazon Braket 运行您的量子任务

Braket 可实现对不同类型的量子计算机的安全按需访问。您可以从、、和Rigetti访问基于门的量子计算机 AQT IonQIQM，也可以访问来自的模拟哈密顿仿真器。 QuEra同时，您无需预先承诺，也不需要通过个别提供商购买访问权限。

- Amazon Braket 控制台提供设备信息和状态，帮助您创建、管理和监控资源及量子任务。
- 通过 Amazon Braket Python SDK 和控制台提交及运行量子任务。软件开发工具包可通过预配置的 Amazon Braket Notebook 访问。
- Amazon Braket API 可通过 Amazon Braket Python SDK 和 Notebook 访问。如果您正在以编程方式构建使用量子计算的应用程序，则可以直接调用 API。

本节中的示例演示了如何使用 Amazon Braket Python SDK 和适用于 AWS Braket 的 Python SDK（Boto3）直接使用 Amazon Braket API。

有关 Amazon Braket Python SDK 的更多信息

要使用 Amazon Braket Python SDK，请先安装适用于 Braket 的 AWS Python 开发工具包 (Boto3)，这样你就可以与之通信。 AWS API您可以将 Amazon Braket Python SDK 看作是适用于量子客户的便捷的 Boto3 包装程序。

- Boto3 包含您需要进入的接口。 AWS API（请注意，Boto3 是一个大型 Python SDK，可以与... 通信。 AWS API 大多数都 AWS 服务 支持 Boto3 接口。 ）
- Amazon Braket Python SDK 包含用于电路、门、设备、结果类型和量子任务其他部分的软件模块。每次创建程序时，都要导入该量子任务所需的模块。
- Amazon Braket Python SDK 可通过 Notebook 访问，这些 Notebook 预装了运行量子任务所需的所有模块和依赖项。
- 如果您不想使用 Notebook，您可以将模块从 Amazon Braket Python SDK 导入到任何 Python 脚本中。

安装 Boto3 后，通过 Amazon Braket Python SDK 创建量子任务的步骤概述如下所示：

1. （可选）打开 Notebook。
2. 导入电路所需的 SDK 模块。
3. 指定 QPU 或模拟器。

4. 实例化电路。
5. 运行电路。
6. 收集结果。

本节中的示例显示了每个步骤的详细信息。

有关更多示例，请参阅上的 Amazon Braket 示例存储库。 GitHub

本节内容：

将量子任务提交给 QPUs

Amazon Braket 允许访问多个可以运行量子任务的设备。您可以单独提交量子任务，也可以设置量子任务批处理。

量子处理单元 (QPUs)

您可以 QPUs 随时向提交量子任务，但该任务在 Amazon Braket 控制台的 “设备” 页面上显示的特定可用性窗口内运行。您可以使用量子任务 ID 检索量子任务的结果，下一节将对此进行介绍。

- AQT IBEX-Q1 ：`arn:aws:braket:eu-north-1::device/qpu/aqt/Ibex-Q1`
- IonQ Forte-1 ：`arn:aws:braket:us-east-1::device/qpu/ionq/Forte-1`
- IonQ Forte-Enterprise-1 ：`arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1`
- IQM Garnet ：`arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet`
- IQM Emerald ：`arn:aws:braket:eu-north-1::device/qpu/iqm/Emerald`
- QuEra Aquila ：`arn:aws:braket:us-east-1::device/qpu/quera/Aquila`
- Rigetti Ankaa-3 ：`arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3`

> **Note**
>
> 您可以取消按需模拟器CREATED状态下的 QPUs 量子任务。您可以尽最大努力取消该QUEUED州的量子任务，用于按需模拟器和. QPUs 请注意，在 QPU 可用性窗口期间，QPU QUEUED 量子任务不太可能成功取消。

本节内容：

AQT

AQT的 IBEX-Q1 QPU 基于位于超高真空室中的宏观射频陷阱中的 ^{40} 个 Ca^{+} 离子的晶体。该设备在室温下运行，可放入两个 19 英寸的数据中心兼容机架中。

陷阱的低加热速率以及使用直接光学过渡进行量子比特旋转，从而实现了高保真门。量子比特过渡由具有非常高的相对频率稳定性的窄线宽激光器驱动。量子比特还具有通过光学架进行高效的状态准备和读取的功能。 All-to-all连接是通过离子晶体中的远距离库仑相互作用实现的。单离子寻址和读出是通过使用高数值孔径镜头实现的。

该AQT器件支持以下量子门。

```
'ccnot', 'cnot', 'cphaseshift', 'cphaseshift00', 'cphaseshift01', 'cphaseshift10',
 'cswap', 'swap', 'iswap', 'pswap', 'ecr', 'cy', 'cz', 'xy', 'xx', 'yy', 'zz', 'h',
 'i', 'phaseshift', 'rx', 'ry', 'rz', 's', 'si', 't', 'ti', 'v', 'vi', 'x', 'y', 'z',
 'prx'
```

通过逐字编译，该AQT设备支持以下本机门。

```
'prx', 'xx', 'rz'
```

> **Note**
>
> 以下内容描述了AQT原生门和 Amazon Braket 之间的等效大门：
>
> - AQTMölmer-Sörensen（MS 或 RXX）大门对应于 Braket 的大门 `'xx'`
> - AQTR 门对应于 Braket 的`'prx'`门
> - 大`'rz'`门的命名是一样的

IonQ

IonQ基于离子阱技术提供 QPUs 基于栅极的产品。 IonQ's捕获的离 QPUs 子建立在一条捕获的 171Yb + 离子链上，这些离子通过真空室内的微型表面电极捕集器在空间上进行限制。

IonQ 设备支持以下量子门。

```
'x', 'y', 'z', 'rx', 'ry', 'rz', 'h', 'cnot', 's', 'si', 't', 'ti', 'v', 'vi', 'xx',
 'yy', 'zz', 'swap'
```

通过逐字编译，它们IonQ QPUs 支持以下本机门。

```
'gpi', 'gpi2', 'ms'
```

如果在使用原生 MS 门时仅指定两个相位参数，则会运行一个完全纠缠的 MS 门。完全纠缠的 MS 门始终执行 π/2 旋转。要指定不同的角度并运行部分纠缠的 MS 门，您可以通过添加第三个参数来指定所需的角度。有关更多信息，请参阅 braket.circuits.gate 模块。

这些原生门只能用于逐字编译。要了解有关逐字编译的更多信息，请参阅逐字编译。

IQM

IQM 量子处理器是基于超导 transmon 量子比特的通用门模型设备。IQM Garnet 是 20 量子比特设备，而 IQM Emerald 是 54 量子比特设备。这两种设备都使用方格拓扑，也称为晶格拓扑。

这些 IQM 设备支持以下量子门。

```
"ccnot", "cnot", "cphaseshift", "cphaseshift00", "cphaseshift01", "cphaseshift10",
 "cswap", "swap", "iswap", "pswap", "ecr", "cy", "cz", "xy", "xx", "yy", "zz", "h",
 "i", "phaseshift", "rx", "ry", "rz", "s", "si", "t", "ti", "v", "vi", "x", "y", "z"
```

通过逐字编译，这些 IQM 设备支持以下本机门。

```
'cz', 'prx'
```

Rigetti

Rigetti 量子处理器是基于全可调超导 qubits 的通用门控模型机器。

- 该 Ankaa-3 系统是一款利用可扩展多芯片技术的 84 量子比特设备。

该 Rigetti 设备支持以下量子门。

```
'cz', 'xy', 'ccnot', 'cnot', 'cphaseshift', 'cphaseshift00', 'cphaseshift01',
 'cphaseshift10', 'cswap', 'h', 'i', 'iswap', 'phaseshift', 'pswap', 'rx', 'ry', 'rz',
 's', 'si', 'swap', 't', 'ti', 'x', 'y', 'z'
```

通过逐字编译，Ankaa-3 支持以下原生门。

```
'rx', 'rz', 'iswap'
```

Rigetti 超导量子处理器只能以 ±π/2 或 ±π 的角度运行“rx”门。

Rigetti 设备提供脉冲电平控制，该设备支持 Ankaa-3 系统的一组以下类型的预定义帧。

```
`flux_tx`, `charge_tx`, `readout_rx`, `readout_tx`
```

该Ankaa-3器件每个电路的最大限制为 20,000 个门。超过此限制的电路会因验证错误而被拒绝。这是一个固定限额，不能提高。栅极计数是指编译后的电路，它可能与原始未编译电路的门数不同。要在提交给 QPU 之前估算已编译的门数，可以在本地使用逐字编译或将电路转换为原生门集（、、）。`rx` `rz` `iswap`

QuEra

QuEra 提供基于中性原子的设备，这些设备可以运行模拟哈密顿模拟（AHS）量子任务。这些专用设备真实再现了数百个同时相互作用的量子比特的时变量子动态。

人们可以通过规定量子比特寄存器的布局以及操纵场的时间和空间依赖性，在模拟哈密顿模拟的范式中对这些设备进行编程。Amazon Braket 提供了通过 Python SDK 的 AHS 模块 `braket.ahs` 构造此类程序的实用工具。

有关更多信息，请参阅模拟哈密顿仿真示例笔记本或使用的 Aquila 提交模拟程序 QuEra页面。

示例：向 QPU 提交量子任务

Amazon Braket 可在 QPU 设备上运行量子电路。以下示例说明如何向 Rigetti 或 IonQ 设备提交量子任务。

选择 Rigetti Ankaa-3 设备，然后查看相关的连接图

```
# import the QPU module
from braket.aws import AwsDevice
# choose the Rigetti device
device = AwsDevice("arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3")

# take a look at the device connectivity graph
device.properties.dict()['paradigm']['connectivity']
```

```
{'fullyConnected': False,
 'connectivityGraph': {'0': ['1', '7'],
  '1': ['0', '2', '8'],
  '2': ['1', '3', '9'],
  '3': ['2', '4', '10'],
  '4': ['3', '5', '11'],
  '5': ['4', '6', '12'],
  '6': ['5', '13'],
  '7': ['0', '8', '14'],
  '8': ['1', '7', '9', '15'],
  '9': ['2', '8', '10', '16'],
  '10': ['3', '9', '11', '17'],
  '11': ['4', '10', '12', '18'],
  '12': ['5', '11', '13', '19'],
  '13': ['6', '12', '20'],
  '14': ['7', '15', '21'],
  '15': ['8', '14', '22'],
  '16': ['9', '17', '23'],
  '17': ['10', '16', '18', '24'],
  '18': ['11', '17', '19', '25'],
  '19': ['12', '18', '20', '26'],
  '20': ['13', '19', '27'],
  '21': ['14', '22', '28'],
  '22': ['15', '21', '23', '29'],
  '23': ['16', '22', '24', '30'],
  '24': ['17', '23', '25', '31'],
```

```
    '25': ['18', '24', '26', '32'],
    '26': ['19', '25', '33'],
    '27': ['20', '34'],
    '28': ['21', '29', '35'],
    '29': ['22', '28', '30', '36'],
    '30': ['23', '29', '31', '37'],
    '31': ['24', '30', '32', '38'],
    '32': ['25', '31', '33', '39'],
    '33': ['26', '32', '34', '40'],
    '34': ['27', '33', '41'],
    '35': ['28', '36', '42'],
    '36': ['29', '35', '37', '43'],
    '37': ['30', '36', '38', '44'],
    '38': ['31', '37', '39', '45'],
    '39': ['32', '38', '40', '46'],
    '40': ['33', '39', '41', '47'],
    '41': ['34', '40', '48'],
    '42': ['35', '43', '49'],
    '43': ['36', '42', '44', '50'],
    '44': ['37', '43', '45', '51'],
    '45': ['38', '44', '46', '52'],
    '46': ['39', '45', '47', '53'],
    '47': ['40', '46', '48', '54'],
    '48': ['41', '47', '55'],
    '49': ['42', '56'],
    '50': ['43', '51', '57'],
    '51': ['44', '50', '52', '58'],
    '52': ['45', '51', '53', '59'],
    '53': ['46', '52', '54'],
    '54': ['47', '53', '55', '61'],
    '55': ['48', '54', '62'],
    '56': ['49', '57', '63'],
    '57': ['50', '56', '58', '64'],
    '58': ['51', '57', '59', '65'],
    '59': ['52', '58', '60', '66'],
    '60': ['59'],
    '61': ['54', '62', '68'],
    '62': ['55', '61', '69'],
    '63': ['56', '64', '70'],
    '64': ['57', '63', '65', '71'],
    '65': ['58', '64', '66', '72'],
    '66': ['59', '65', '67'],
    '67': ['66', '68'],
    '68': ['61', '67', '69', '75'],
```

```
  '69': ['62', '68', '76'],
  '70': ['63', '71', '77'],
  '71': ['64', '70', '72', '78'],
  '72': ['65', '71', '73', '79'],
  '73': ['72', '80'],
  '75': ['68', '76', '82'],
  '76': ['69', '75', '83'],
  '77': ['70', '78'],
  '78': ['71', '77', '79'],
  '79': ['72', '78', '80'],
  '80': ['73', '79', '81'],
  '81': ['80', '82'],
  '82': ['75', '81', '83'],
  '83': ['76', '82']}}
```

前面的字典 connectivityGraph 列出了 Rigetti 设备中每个量子比特的相邻量子比特。

选择 IonQ Forte-Enterprise-1 设备

对于该IonQ Forte-Enterprise-1设备，connectivityGraph为空，如以下示例所示，因为该设备提供all-to-all连接。因此，无需详细说明 connectivityGraph。

```
# or choose the IonQ Forte-Enterprise-1 device
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1")

# take a look at the device connectivity graph
device.properties.dict()['paradigm']['connectivity']
```

```
{'fullyConnected': True, 'connectivityGraph': {...}}
```

如以下示例所示，如果您选择指定默认存储桶以外的其他位置，则可以选择调整 shots（默认值 = 1000）、poll_timeout_seconds（默认值 = 432000 = 5 天）、poll_interval_seconds（默认值 = 1）和 S3 存储桶 (s3_location) 的位置。

```
my_task = device.run(circ, s3_location = 'amazon-braket-my-folder', shots=100,
 poll_timeout_seconds = 100, poll_interval_seconds = 10)
```

IonQ 和 Rigetti 设备会自动将提供的电路编译成各自的原生门集，并将抽象 qubit 索引映射到相应的 QPU 上的物理 qubits 索引。

> **Note**
>
> QPU 设备的容量有限。达到容量后，等待时间可能会更长。

Amazon Braket 可在特定的可用性窗口内运行 QPU 量子任务，但您仍然可以随时（全天候）提交量子任务，因为所有相应的数据和元数据都可靠地存储在相应的 S3 存储桶中。如下节所示，您可以使用 `AwsQuantumTask` 和您唯一的量子任务 ID 恢复量子任务。

检查编译后的电路

当量子电路需要在硬件设备 [如量子处理单元（QPU）] 上运行时，必须首先将该电路编译成设备可以理解和处理的可接受格式。例如，将高级量子电路向下转换到目标 QPU 硬件支持的特定原生门。检查量子电路的实际编译输出对实现调试和优化目的会非常有用。这些知识可以帮助识别潜在的问题、瓶颈或机会，以提高量子应用的性能和效率。您可以使用下面提供的代码查看和分析 Rigetti 和 IQM 量子计算设备量子电路的编译输出。

```
task = AwsQuantumTask(arn=task_id, aws_session=session)
# After the task has finished running
task_result = task.result()
compiled_circuit = task_result.get_compiled_circuit()
```

> **Note**
>
> 当前不支持查看 IonQ 设备的编译电路输出。

运行多个程序

Amazon Braket 提供了两种高效运行多个量子程序的方法，即程序集和量子任务批处理。

程序集是使用多个程序运行工作负载的首选方式。它们可将多个程序打包成单个 Amazon Braket 量子任务。与单独提交程序相比，程序集可以提高性能并节省成本，尤其是当程序执行数量接近 100 时。

目前，IQM 和 Rigetti 设备支持程序集。在向提交程序集之前 QPUs，建议先在 Amazon Braket 本地模拟器上进行测试。要检查设备是否支持程序集，您可以使用 Amazon Braket SDK 查看设备属性，或者在 Amazon Braket 控制台中查看设备页面。

以下示例显示了如何运行程序集。

```
from math import pi
from braket.devices import LocalSimulator
from braket.program_sets import ProgramSet
from braket.circuits import Circuit

program_set = ProgramSet([
    Circuit().h(0).cnot(0,1),
    Circuit().rx(0, pi/4).ry(1, pi/8).cnot(1,0),
    Circuit().t(0).t(1).cz(0,1).s(0).cz(1,2).s(1).s(2),
])

device = LocalSimulator()
result = device.run(program_set, shots=300).result()
print(result[0][0].counts)  # The result of the first program in the program set
```

要详细了解构造程序集（例如，使用单个程序从多个可观察对象或参数构造程序集）和检索程序集结果的不同方法，请参阅《Amazon Braket 开发人员指南》中的“程序集”部分和 Braket 示例 Github 存储库中的程序集文件夹。

每台 Amazon Braket 设备都支持量子任务批处理。批处理对于在按需模拟器（SV1、DM1 或 TN1）上运行的量子任务特别有用，因为它们可以并行处理多个量子任务。批处理可并行启动量子任务。例如，如果您希望进行需要 10 项量子任务的计算，并且这些量子任务中的程序相互独立，则建议使用任务批处理。在不支持程序集的设备上运行包含多个程序的工作负载时，请使用量子任务批处理。

以下示例显示了如何运行量子任务。

```
from braket.circuits import Circuit
from braket.devices import LocalSimulator

bell = Circuit().h(0).cnot(0, 1)
circuits = [bell for _ in range(5)]

device = LocalSimulator()
batch = device.run_batch(circuits, shots=100)
print(batch.results()[0].measurement_counts)  # The result of the first quantum task in
 the batch
```

有关批处理的更多具体信息，请参阅上的 Amazon Braket 示例。 GitHub

本节内容：

- 关于程序集和费用

- 关于量子任务批处理和成本
- 量子任务批处理和 PennyLane
- 任务批处理和参数化电路

关于程序集和费用

程序集通过将多达 100 个程序或参数集打包到单个量子任务中来高效运行多个量子程序。对于程序集，您只需支付每项任务的费用以及基于所有程序总拍摄次数的每次拍摄费用，这与单独提交程序相比，显著降低了成本。这种方法对于有许多程序且每个程序的拍摄次数较少的工作负载特别有用。目前，IQM 和 Rigetti 设备以及 Amazon Braket 本地模拟器都支持程序集。

有关更多信息，请参阅“程序集”部分，了解详细的实现步骤、最佳实践和代码示例。

关于量子任务批处理和成本

关于量子任务批处理和计费成本，请记住一些注意事项：

- 默认情况下，量子任务批处理重试全部超时或量子任务失败 3 次。
- 一批长时间运行的量子任务（例如：对于 SV1，为 34 个 qubits）可能会产生高昂的费用。在开始一批量子任务之前，请务必仔细检查 `run_batch` 分配值。我们不建议 TN1 搭配 `run_batch` 使用。
- TN1可能因排练阶段的任务失败而产生费用（有关更多信息，请参阅 TN1 说明）。自动重试可能会增加成本，因此我们建议在使用 TN1 时将批处理时的“max_retries”次数设置为 0（参见量子任务批处理，第 186 行）。

量子任务批处理和 PennyLane

在 Amazon Braket PennyLane 上使用时，通过`parallel = True`设置实例化 Amazon Braket 设备的时间，充分利用批处理功能，如以下示例所示。

```
import pennylane as qml

# Define the number of wires (qubits) you want to use
wires = 2  # For example, using 2 qubits

# Define your S3 bucket
my_bucket = "amazon-braket-s3-demo-bucket"
my_prefix = "pennylane-batch-output"
```

```
s3_folder = (my_bucket, my_prefix)

device = qml.device("braket.aws.qubit",
                    device_arn="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
                    wires=wires,
                    s3_destination_folder=s3_folder,
                    parallel=True)
```

有关批处理的更多信息 PennyLane，请参阅量子电路的并行优化。

任务批处理和参数化电路

在提交包含参数化电路的量子任务批处理时，您可以提供 `inputs` 字典（用于批次中的所有量子任务）或输入字典的 `list`，在第二种情况下，第 `i` 个字典与第 `i` 个任务配对，如以下示例所示。

```
from braket.circuits import Circuit, FreeParameter, Observable
from braket.aws import AwsQuantumTaskBatch, AwsDevice

# Define your quantum device
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")

# Create the free parameters
alpha = FreeParameter('alpha')
beta = FreeParameter('beta')

# Create two circuits
circ_a = Circuit().rx(0, alpha).ry(1, alpha).cnot(0, 2).xx(0, 2, beta)
circ_a.variance(observable=Observable.Z(), target=0)

circ_b = Circuit().rx(0, alpha).rz(1, alpha).cnot(0, 2).zz(0, 2, beta)
circ_b.expectation(observable=Observable.Z(), target=2)

# Use the same inputs for both circuits in one batch
tasks = device.run_batch([circ_a, circ_b], inputs={'alpha': 0.1, 'beta': 0.2})

# Or provide each task its own set of inputs
inputs_list = [{'alpha': 0.3, 'beta': 0.1}, {'alpha': 0.1, 'beta': 0.4}]

tasks = device.run_batch([circ_a, circ_b], inputs=inputs_list)
```

您还可以为单个参数电路准备输入字典列表，然后将其作为量子任务批量提交。如果列表中有 N 个输入字典，则该批次包含 N 个量子任务。第 `i` 个量子任务对应于使用第 `i` 个输入字典执行的电路。

```
from braket.circuits import Circuit, FreeParameter

# Create a parametric circuit
circ = Circuit().rx(0, FreeParameter('alpha'))

# Provide a list of inputs to execute with the circuit
inputs_list = [{'alpha': 0.1}, {'alpha': 0.2}, {'alpha': 0.3}]

tasks = device.run_batch(circ, inputs=inputs_list, shots=100)
```

我的量子任务什么时候能运行？

当您提交电路时，Amazon Braket 会将其发送到您指定的设备。量子处理单元（QPU）和按需模拟器量子任务按接收顺序排队处理。提交量子任务后处理该任务所需的时间会有所不同，具体取决于其他 Amazon Braket 客户提交任务的数量和复杂程度以及所选 QPU 的可用性。

本节内容：

QPU 可用性窗口和状态

QPU 可用性因设备而异。

在 Amazon Braket 控制台的“设备”页面中，您可以看到当前和即将推出的可用性窗口以及设备状态。此外，每个设备页面都显示量子任务和混合作业的单独队列深度。

无论可用性窗口如何，如果客户无法使用设备，则该设备将被视为离线。例如，它可能由于定期维护、升级或操作问题而处于离线状态。

队列可见性

在提交量子任务或混合作业之前，您可以通过检查设备队列深度来查看摆在您面前的量子任务或混合作业的数量。

队列深度

Queue depth 指排队等候特定设备的量子任务和混合作业的数量。可通过 Braket Software Development Kit (SDK) 或 Amazon Braket Management Console 访问设备的量子任务和混合作业队列数。

1. 任务队列深度是指当前等待以正常优先级运行的量子任务总数。
2. 优先任务队列深度是指等待通过 Amazon Braket Hybrid Jobs 运行的已提交量子任务的总数。这些任务在独立任务之前运行。
3. 混合作业队列深度是指当前在设备上排队的混合作业总数。Quantum tasks 作为混合作业的一部分提交，具有优先级，汇总在 Priority Task Queue 中。

希望通过 Braket SDK 查看队列深度的客户可以修改以下代码片段以获取其量子任务或混合作业的队列位置：

```
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1")

# returns the number of quantum tasks queued on the device
print(device.queue_depth().quantum_tasks)
{<QueueType.NORMAL: 'Normal'>: '0', <QueueType.PRIORITY: 'Priority'>: '0'}


# returns the number of hybrid jobs queued on the device
print(device.queue_depth().jobs)
'3'
```

向 QPU 提交量子任务或混合作业可能会导致您的工作负载处于 QUEUED 状态。Amazon Braket 可使客户看到他们的量子任务和混合任务队列的位置。

队列位置

Queue position 指您的量子任务或混合作业在相应设备队列中的当前位置。它可以通过 Braket Software Development Kit (SDK) 或 Amazon Braket Management Console 获得，用于量子任务或混合作业。

希望通过 Braket SDK 查看队列位置的客户可以通过修改以下代码片段来获取其量子任务或混合作业的队列位置：

```
# choose the device to run your circuit
device = AwsDevice("arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet")
```

```
#execute the circuit
task = device.run(bell, s3_folder, shots=100)

# retrieve the queue position information
print(task.queue_position().queue_position)

# Returns the number of Quantum Tasks queued ahead of you
'2'


from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    "arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet",
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    wait_until_complete=False
)

# retrieve the queue position information
print(job.queue_position().queue_position)
'3' # returns the number of hybrid jobs queued ahead of you
```

设置电子邮件或 SMS 通知

当 QPU 的可用性发生变化或量子任务状态发生变化 EventBridge 时，Amazon Braket 会向亚马逊发送事件。请按照以下步骤通过电子邮件或 SMS 消息接收设备和量子任务状态变更通知：

1. 创建 Amazon SNS 主题和订阅电子邮件或 SMS。电子邮件或 SMS 的可用性取决于您所在的区域。有关更多信息，请参阅 Amazon SNS 入门和发送 SMS 消息。
2. 在中创建一条规则 EventBridge ，用于触发您的 SNS 主题的通知。有关更多信息，请参阅使用亚马逊监控 Amazon Braket。 EventBridge

（可选）设置 SNS 通知

您可以通过 Amazon Simple Notification Service (SNS) 设置通知，以便在 Amazon Braket 量子任务完成时收到提醒。如果您预计等待时间很长，则活动通知很有用；例如，当您提交大型量子任务或在设备可用性窗口之外提交量子任务时。如果您不想等待量子任务完成，则可以设置 SNS 通知。

Amazon Braket Notebook 将引导您完成设置步骤。有关更多信息，请参阅上的 Amazon Braket 示例 GitHub，特别是用于设置通知的笔记本示例。

使用预留

通过预留，您可以独家使用自己选择的量子设备。您可以在方便时安排预留，这样就可以确切地知道工作负载何时开始和结束执行了。所有 Braket 设备均以 1 小时为增量进行预订，并且可以提前 48 小时取消预订，无需支付额外费用。我们建议使用您的 Braket Direct Reservation ARN 提前将量子任务和混合任务排队等候即将到来的预订，或者在预订期间提交工作负载。

无论您在量子处理单元 (QPU) 上运行多少量子任务和混合作业，访问专用设备的费用都取决于您的预留时间。可在我们的定价页面或通过 Amazon Braket 管理控制台找到可供预订的量子计算机的最新清单。

> **Note**
>
> 在预约中，没有门禁限制。此外，对于IonQ设备，错误缓解任务的最小镜头数量减少到500（而按需拍摄的最小镜头计数为 2500）。

何时使用预留

利用预留访问权限为您提供了便捷性和可预测性，让您能够准确了解量子工作负载何时开始和结束执行。与按需提交任务和混合作业相比，您无需排队等候其他客户任务。由于您在预留期间拥有对设备的独占访问权限，因此在整个预留期间，只有您的工作负载在设备上运行。

我们建议在研究的设计和原型设计阶段使用按需访问权限，从而实现算法的快速且经济有效的迭代。准备好得出最终实验结果后，可以考虑在方便的时候安排设备预留，以确保能够在项目或出版的最后期限之前完成。如果您希望在特定时间执行任务，如在量子计算机上运行现场演示或讲习会时，我们也建议您使用预留功能。

本节内容：

如何创建预留

要创建预留，请按照以下步骤联系 Braket 团队：

1. 打开 Amazon Braket 控制台。
2. 在左侧窗格中选择 Braket Direct，然后在“预留”部分选择“预留设备”。
3. 选择您要预留的设备。
4. 提供您的联系信息，包括姓名和电子邮件。请务必提供您定期检查的有效的电子邮件地址。
5. 在“告诉我们您的工作负载”下，提供有关使用您的预留运行的工作负载的所有详细信息。例如，期望的预留时长、相关限制条件或期望的时间表。

提交表格后，您将收到一封来自 Braket 团队的电子邮件，其中包含后续步骤。一旦您的预留得到确认，您将通过电子邮件收到预留 ARN。您需要使用预留 ARN 来创建预留任务。在没有预留 ARN 的情况下创建的任务将提交到常规按需队列，并且不会在您的预留期间运行。

> **Note**
>
> 只有在您收到预留 ARN 后，您的预留才会得以确认。

预留按至少 1 小时为增量提供，某些设备可能有额外的预留时长限制（包括最短和最长预留时长）。在确认预留之前，Braket 团队会与您共享所有相关信息。

Braket 团队将通过您的电子邮件与您联系，安排与 Braket 专家进行 30 分钟的会谈。

在预留期间运行量子任务

从“创建预留”中获取有效的预留 ARN 后，您可以创建要在预留期间运行的量子任务。使用预留 ARN 提交的量子任务和混合任务不会显示在设备队列中。在预约开始时间之前提交的任务将保持该QUEUED状态，直到您的预订开始。

> **Note**
>
> 预订视 AWS 账户和设备而定。只有创建预留的 AWS 账户才能使用您的预订 ARN。
> 在预约期间，既可以创建预留任务，也可以创建常规任务。要验证创建的 Braket 量子任务是否与预留关联，请在 Braket 控制台中查看量子任务页面上的 “预留 ARN” 字段，或者使用 SDK 查询任务元数据中的相同字段。本页的其余部分将介绍如何指定哪些任务与预留相关联。

您可以使用Python SDKs诸如 Brak et、、、之类的量子任务 CUDA-QPennyLaneQiskit，也可以直接使用 boto3（使用 Boto3）创建量子任务。要使用预留，您必须拥有 Amazon Braket Python SDK 版本

1.79.0 或更高版本。您可以使用以下代码更新到最新的 Braket SDK、Qiskit 提供程序和 PennyLane 插件。

```
pip install --upgrade amazon-braket-sdk amazon-braket-pennylane-plugin qiskit-braket-provider
```

使用 **DirectReservation** 上下文管理器运行任务

在计划预留中运行任务的推荐方法是使用 DirectReservation 上下文管理器。通过指定您的目标设备和预留 ARN，上下文管理器可确保在 Python with 语句中创建的所有任务均以独占访问该设备的方式运行。

首先，定义量子电路和设备。然后，使用预留上下文并运行任务。确保您的整个工作负载都在**with**区块内运行；在**with**区块范围之外运行的任何内容都不会与您的预留相关联！

```
from braket.aws import AwsDevice, DirectReservation
from braket.circuits import Circuit
from braket.devices import Devices

bell = Circuit().h(0).cnot(0, 1)
device = AwsDevice(Devices.IonQ.ForteEnterprise1)

# run the circuit in a reservation
with DirectReservation(device, reservation_arn="<my_reservation_arn>"):
    task = device.run(bell, shots=100)
```

只要创建量子任务时DirectReservation上下文处于活动状态 CUDA-QPennyLane，就可以使用、和Qiskit插件在预留中创建量子任务。例如，使用 Qiskit-Braket 提供程序，您可以按如下方式运行任务。

```
from braket.devices import Devices
from braket.aws import DirectReservation
from qiskit import QuantumCircuit
from qiskit_braket_provider import BraketProvider


qc = QuantumCircuit(2)
qc.h(0)
qc.cx(0, 1)

qpu = BraketProvider().get_backend("Forte Enterprise 1")
```

```
# run the circuit in a reservation
with DirectReservation(Devices.IonQ.ForteEnterprise1,
 reservation_arn="<my_reservation_arn>"):
    qpu_task = qpu.run(qc, shots=10)
```

同样，以下代码在预留期间使用 Braket-PennyLane 插件运行电路。

```
from braket.devices import Devices
from braket.aws import DirectReservation
import pennylane as qml


dev = qml.device("braket.aws.qubit", device_arn=Devices.IonQ.ForteEnterprise1.value,
 wires=2, shots=10)

@qml.qnode(dev)
def bell_state():
    qml.Hadamard(wires=0)
    qml.CNOT(wires=[0, 1])
    return qml.probs(wires=[0, 1])

# run the circuit in a reservation
with DirectReservation(Devices.IonQ.ForteEnterprise1,
 reservation_arn="<my_reservation_arn>"):
    probs = bell_state()
```

手动设置预留上下文

您还可以使用以下代码手动设置预留。

```
# set reservation context
reservation_context = DirectReservation(device,
 reservation_arn="<my_reservation_arn>").start()

# run circuit during reservation
task = device.run(bell, shots=100)
```

这非常适合 Jupyter Notebook，其中，上下文可以在第一个单元格中运行，所有后续任务都将在预留中运行。

> **Note**
>
> 包含 `.start()` 调用的单元只能运行一次。

要切换回按需模式，请执行以下操作：重新启动 Jupyter Notebook，或调用以下命令将上下文改回按需模式。

```
reservation_context.stop()  # unset reservation context
```

> **Note**
>
> 预订有预先确定的开始和结束时间（参见创建预订）。`reservation_context.start()` 和 `reservation_context.stop()` 方法不会开始或终止预留。相反，当上下文处于活动状态时，您创建的任何量子任务都将与您的预留相关联，并且仅在您的计划预留期间运行。预订上下文对预定预订时间没有影响。

创建任务时明确传递预留 ARN

在预留期间创建任务的另一种方法是在调用 `device.run()` 时明确传递预留 ARN。

```
task = device.run(bell, shots=100, reservation_arn="<my_reservation_arn>")
```

此方法直接将量子任务与预留 ARN 关联起来，确保其在预留期内运行。对于此选项，请将预留 ARN 添加到您计划在预留期间运行的每项任务中。但是，请注意，在使用第三方库（例如Qiskit或）时 PennyLane，可能很难确保提交的任务使用的是正确的预留 ARN。因此，建议使用 DirectReservation 上下文管理器。

直接使用 boto3 时，请在创建任务时将预留 ARN 作为关联传递。

```
import boto3

braket_client = boto3.client("braket")


kwargs["associations"] = [
    {
        "arn": "<my_reservation_arn>",
```

```
        "type": "RESERVATION_TIME_WINDOW_ARN",
    }
]

response = braket_client.create_quantum_task(**kwargs)
```

在预留期间运行混合作业

将 Python 函数作为混合作业运行后，您可以通过传递 `reservation_arn` 关键字参数在预留中运行混合作业。混合作业中的所有任务都使用预留 ARN。重要的是，只有在您的预留开始后，带 *`reservation_arn`* 的混合作业才会启动经典计算。

> **Note**
>
> 预留期间运行的混合作业只能成功运行预留设备上的量子任务。如果尝试使用其他按需 Braket 设备，将会导致错误。如果您需要在同一个混合作业中同时在按需模拟器和预留设备上运行任务，请改用 `DirectReservation`。

以下代码演示了如何在预留期间运行混合作业。

```
from braket.aws import AwsDevice
from braket.devices import Devices
from braket.jobs import get_job_device_arn, hybrid_job

@hybrid_job(device=Devices.IonQ.ForteEnterprise1,
 reservation_arn="<my_reservation_arn>")
def example_hybrid_job():
    # declare AwsDevice within the hybrid job
    device = AwsDevice(get_job_device_arn())
    bell = Circuit().h(0).cnot(0, 1)

    task = device.run(bell, shots=10)
```

对于使用 Python 脚本的混合作业（参见开发者指南中有关创建第一个混合作业的部分），您可以在创建作业时传递 `reservation_arn` 关键字参数，从而在预留中运行它们。

```
from braket.aws import AwsQuantumJob
from braket.devices import Devices

job = AwsQuantumJob.create(
```

```
    Devices.IonQ.ForteEnterprise1,
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    reservation_arn="<my_reservation_arn>"
)
```

预留结束后会发生什么

预留结束后，您将不再拥有设备的专门访问权限。通过此预留排队的所有剩余工作负载都将自动取消。

> **Note**
>
> 任何在预留结束时处于 RUNNING 状态的任务都将被取消。我们建议您在方便时使用保存和重新启动检查点作业。

正在进行的预留（例如预留开始后和结束前）无法延长，因为每项预留都代表着独立的专用设备访问权限。例如，两个 back-to-back预留被视为分开的，第一个预留中的任何待处理任务都会自动取消。它们不会在第二个预留中恢复。

> **Note**
>
> 预订代表您 AWS 账户的专用设备访问权限。即使设备处于闲置状态，其他客户也无法使用它。因此，无论使用时间长短，都要按预留时间长度收费。

取消或重新安排现有预留

您可以在预定预留开始时间前不少于 48 小时取消预留。要取消预留，请回复您收到的预留确认电子邮件及取消申请。

要重新安排，您必须取消现有预留，然后创建新的预留。

缓解错误技术

量子误差缓解是一套旨在降低量子计算机中错误的影响的技术。

量子设备会受到环境噪声的影响，这会降低所执行的计算的质量。尽管容错量子计算有望解决这一问题，但当前的量子设备会受到量子比特数量和相对较高的错误率的限制。为了在短期内解决这一问题，

研究人员正在研究提高噪声量子计算准确性的方法。这种方法被称为量子误差缓解，涉及使用各种技术从噪声测量数据中提取最佳信号。

本节内容：

IonQ 设备上的错误缓解技术

错误缓解包括运行多个物理电路并将它们的测量结果组合在一起以获得更好的结果。

> **Note**
>
> 对于所有 IonQ 设备：使用按需模式时，有 100 万次门拍摄限制，错误缓解任务至少有 2500 次拍摄。对于直接预留，没有门拍摄限制，错误缓解任务至少有 500 次拍摄。

去偏

IonQ 设备具有一种名为去偏的错误缓解方法。

去偏将电路映射成多个变体，这些变体作用于不同的量子比特排列或具有不同的门分解。这种做法通过使用不同的电路实现来降低系统误差（如门过旋转或单个错误的量子比特）的影响，否则这些电路可能会出现测量结果偏差。这是以校准多个量子比特和门的额外开销为代价的。

有关去偏的更多信息，请参阅通过对称化增强量子计算机性能。

> **Note**
>
> 使用去偏方法至少需要 2500 次拍摄。

您可以使用以下代码在 IonQ 设备上运行带去偏的量子任务：

```
from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.error_mitigation import Debias

# choose an IonQ device
device = AwsDevice("arn:aws:braket:us-east-1::device/qpu/ionq/Forte-Enterprise-1")
```

```
circuit = Circuit().h(0).cnot(0, 1)

task = device.run(circuit, shots=2500, device_parameters={"errorMitigation": Debias()})

result = task.result()
print(result.measurement_counts)
>>> {"00": 1245, "01": 5, "10": 10 "11": 1240} # result from debiasing
```

量子任务完成后，您可以看到量子任务的测量概率及任何结果类型。所有变体的测量概率和计数汇总到一个分布中。电路中指定的任何结果类型（如期望值）均使用汇总测量计数进行计算。

锐化

您还可以访问使用不同的后处理策略（称为锐化）计算的测量概率。锐化会比较每个变体的结果并丢弃不一致的拍摄，从而有利于各变体之间最有可能的测量结果。有关更多信息，请参阅通过对称化增强量子计算机性能。

重要的一点，锐化假设输出分布的形式是稀疏的，高概率状态很少，零概率状态很多。如果此假设无效，则会扭曲概率分布。

您可以在 Braket Python SDK 中访问 `additional_metadata` 字段中经过锐化分布的 `GateModelTaskResult` 概率。请注意，锐化不会返回测量计数，而是返回重新归一化的概率分布。以下代码片段展示了如何在锐化后访问发行版。

```
print(result.additional_metadata.ionqMetadata.sharpenedProbabilities)
>>> {"00": 0.51, "11": 0.549} # sharpened probabilities
```

使用 Amazon Braket Hybrid Jobs

Amazon Braket Hybrid Jobs 为您提供了一种运行混合量子经典算法的方法，需要经典 AWS 资源和量子处理单元（ ）。QPUsHybrid Jobs 旨在启动请求的经典资源，运行您的算法，并在完成后释放实例，因此您只需为实际使用的资源付费。

Hybrid Jobs 非常适合长期运行的迭代算法，这些算法涉及使用经典计算资源和量子计算资源。使用 Hybrid Jobs 提交算法运行后，Braket 将在可扩展的容器化环境中运行您的算法。算法完成后，您就可以检索结果了。

此外，通过混合作业创建的量子任务受益于更高的优先级排队到目标 QPU 设备。这种优先级划分可确保您的量子计算在队列中等待的其他任务之前得到处理和运行。这对于迭代混合算法尤其有利，在迭代混合算法中，一项量子任务的结果取决于先前量子任务的结果。此类算法的示例包括量子近似优化算法（QAOA）、变分量子特征求解器或量子机器学习。您还可以近乎实时地监控算法进度，从而跟踪成本、预算或自定义指标，如训练损失或期望值。

您可以使用以下方式在 Braket 中访问混合作业：

- Amazon Braket Python SDK。
- Amazon Braket 控制台。
- Amazon Braket API。

本节内容：

何时使用 Amazon Braket Hybrid Jobs

Amazon Braket Hybrid Jobs 可运行混合量子经典算法，如变分量子特征求解器（VQE）和量子近似优化算法（QAOA），它们将经典计算资源与量子计算设备相结合，以优化当今量子系统的性能。Amazon Braket Hybrid Jobs 有三个主要好处：

1. 性能：与在您自己的环境中运行混合算法相比，Amazon Braket Hybrid Jobs 的性能更佳。当您的作业正在运行时，它可以优先访问选定的目标 QPU。您的任务比设备上排队的其他任务之前运行的时间要早。由此，混合算法的运行时更短、更具可预测性。Amazon Braket Hybrid Jobs 还支持参数化编译。您可以使用免费参数提交电路。Braket 只需编译一次电路，无需重新编译即可对同一电路进行后续参数更新，从而实现更快的运行时。
2. 便利：Amazon Braket Hybrid Jobs 简化了计算环境的设置和管理，并在混合算法运行时保持其运行。您只需提供算法脚本，然后选择要运行的量子设备（量子处理单元或模拟器）即可。Amazon Braket 等待目标设备可用，启动传统资源，在预先构建的容器环境中运行工作负载，将结果返回到 Amazon Simple Storage Service (Amazon S3)，然后释放计算资源。
3. 指标：Amazon Braket Hybrid Job on-the-fly s 提供有关运行算法的见解，并近乎实时地向亚马逊 CloudWatch 和 Amazon Braket 控制台提供可自定义的算法指标，以便您可以跟踪算法的进度。

使用 Amazon Braket Hybrid Jobs 运行混合作业

要使用 Amazon Braket Hybrid Jobs 运行混合作业，您首先需要定义算法。您可以通过使用 Amazon Braket Python 软件开发工具包编写算法脚本以及其他依赖项文件来定义它，也可以使用 Amazon Braket Python SDK 或。PennyLane如果您想使用其他（开源或专有）库，则可以使用 Docker 定义自己的自定义容器映像，包括这些库。有关更多信息，请参阅自带容器（BYOC）。

无论哪种情况，接下来都要使用 Amazon Braket API 创建混合作业，在其中提供算法脚本或容器，选择混合作业要使用的目标量子设备，然后从各种可选设置中进行选择。为这些可选设置提供的默认值适用于大多数使用案例。对于运行混合作业的目标设备，您可以在 QPU、按需模拟器（如 SV1、DM1 或 TN1）或经典混合作业实例本身之间进行选择。使用按需模拟器或 QPU，您的混合作业容器可以对远程设备进行 API 调用。使用嵌入式模拟器，模拟器与算法脚本嵌入在同一个容器中。中的闪电模拟 PennyLane 器嵌入了默认的预建混合作业容器供您使用。如果您使用嵌入式 PennyLane 模拟器或自定义模拟器运行代码，则可以指定实例类型以及要使用的实例数量。有关每种选择的相关费用，请参阅 Amazon Braket 定价页面。

如果您的目标设备是按需模拟器或嵌入式模拟器，Amazon Braket 会立即开始运行混合任务。它启动混合作业实例（您可以在 API 调用中自定义实例类型），运行算法，将结果写入 Amazon S3，然后释放您的资源。此资源版本可确保您只需按实际使用量付费。

每个量子处理单元（QPU）的并发混合任务总数受到限制。如今，在任何给定时间，QPU 上只能运行一个混合作业。队列用于控制可运行的混合作业的数量，以免超过规定的限制。如果您的目标设备是 QPU，则混合作业将首先进入所选 QPU 的作业队列。Amazon Braket 启动所需的混合作业实例，并在设备上运行您的混合作业。在算法持续时间内，您的混合作业具有优先访问权限，这意味着混合作业中的量子任务优先于设备上排队的其他 Braket 量子任务，前提是该作业的量子任务每隔几分钟提交给 QPU 一次。混合作业完成后，资源就会被释放，这意味着您只需按实际使用量付费。

> **Note**
>
> 设备是区域性的，您的混合任务与主设备在 AWS 区域 同一设备上运行。

在模拟器和 QPU 目标场景中，您可以选择将自定义算法指标（如哈密顿量）定义为算法的一部分。这些指标会自动报告给亚马逊， CloudWatch 然后在亚马逊Braket控制台中以近乎实时的方式显示。

> **Note**
>
> 如果您想使用基于 GPU 的实例，请务必使用 Braket 上嵌入式模拟器中提供的基于 GPU 的模拟器（例如，`lightning.gpu`）。如果您选择基于 CPU 的嵌入式模拟器之一（例

如，lightning.qubit 或 braket:default-simulator），则不会使用 GPU，且可能会产生不必要的成本。

混合作业的关键概念

本节介绍了 Amazon Braket Python SDK 提供的 AwsQuantumJob.create 函数以及容器文件结构映射的关键概念。

除了构成完整算法脚本的一个或多个文件外，您的混合作业还可以有其他输入输出。当您的混合作业启动时，Amazon Braket 会将创建混合作业时提供的输入复制到运行算法脚本的容器中。混合任务完成后，算法期间定义的所有输出都将复制到指定的 Amazon S3 位置。

> **Note**
>
> 算法指标是实时报告的，不遵循此输出程序。

Amazon Braket 还提供了多个环境变量和辅助函数，以简化与容器输入和输出的交互。有关更多信息，请参阅 Amazon Braket 软件开发工具包中的 braket.jobs 软件包。

本节内容：

输入

输入数据：通过使用 input_data 参数指定设置为字典的输入数据文件，可将输入数据提供给混合算法。用户在 SDK 的 AwsQuantumJob.create 函数中定义 input_data 参数。这会将输入数据复制到环境变量所给位置的容器文件系统 "AMZN_BRAKET_INPUT_DIR"。有关如何在混合算法中使用输入数据的几个示例，请参阅带有 Amazon Braket Hybrid Jobs 的 QAOA PennyLane 和 Amazon Braket Hybrid Job s Jupyter 笔记本中的量子机器学习。

> **Note**
>
> 当输入数据很大 (>1GB) 时，混合作业需要很长时间才能提交。这是因为本地输入数据将首先上传到 S3 存储桶，然后将 S3 路径添加到混合任务请求中，最后将混合任务请求提交给 Braket 服务。

超参数：如果传入 `hyperparameters`，则它们在环境变量 "AMZN_BRAKET_HP_FILE" 下可用。

> **Note**
>
> 有关如何创建超参数和输入数据而后将这些信息传递给混合作业脚本的更多信息，请参阅使用超参数部分和此 github 页面。

检查点：要指定 `job-arn` 在新混合作业中使用哪个检查点，请使用 `copy_checkpoints_from_job` 命令。此命令将检查点数据复制到新混合作业的 `checkpoint_configs3Uri` 中，使其在作业运行时在环境变量 `AMZN_BRAKET_CHECKPOINT_DIR` 给出的路径上可用。默认值为 `None`，这意味着来自其他混合作业的检查点数据将不会用于新的混合作业。

输出

量子任务：量子任务结果存储在 S3 位置 `s3://amazon-braket-<region>-<accountID>/jobs/<job-name>/tasks`。

作业结果：您的算法脚本保存到环境变量给定目录的所有内容"AMZN_BRAKET_JOB_RESULTS_DIR"都将复制到 `output_data_config` 中指定的 S3 位置。如果未指定数值，将默认为 `s3://amazon-braket-<region>-<accountID>/jobs/<job-name>/<timestamp>/data`。我们提供 SDK 帮助器函数 **`save_job_result`**，当从算法脚本调用时，您可以使用该函数以字典的形式方便地存储结果。

检查点：如果要使用检查点，可将其保存在环境变量 "AMZN_BRAKET_CHECKPOINT_DIR" 给出的目录中。您也可以使用 `save_job_checkpoint` 调用函数。

算法指标：您可以将算法指标定义为算法脚本的一部分，这些指标将在混合作业运行时发送到 Amazon CloudWatch 并实时显示在 Amazon Braket 控制台中。有关如何使用算法指标的示例，请参阅使用 Amazon Braket Hybrid Jobs 运行 QAOA 算法。

有关保存作业输出的更多信息，请参阅 Hybrid Jobs 文档中的保存结果。

环境变量

Amazon Braket 提供了多个环境变量来简化与容器输入输出的交互。以下代码列出了 Braket 使用的环境变量。

- AMZN_BRAKET_INPUT_DIR— 输入数据目录opt/braket/input/data。
- AMZN_BRAKET_JOB_RESULTS_DIR— opt/braket/model 要将作业结果写入的输出目录。
- AMZN_BRAKET_JOB_NAME：作业的名称。
- AMZN_BRAKET_CHECKPOINT_DIR：检查点目录。
- AMZN_BRAKET_HP_FILE：包含超参数的文件。
- AMZN_BRAKET_DEVICE_ARN— 设备 ARN（AWS 资源名称）。
- AMZN_BRAKET_OUT_S3_BUCKET：CreateJob 请求 OutputDataConfig 中指定的输出 Amazon S3 存储桶。
- AMZN_BRAKET_SCRIPT_ENTRY_POINT：CreateJob 请求的 ScriptModeConfig 中指定的入口点。
- AMZN_BRAKET_SCRIPT_COMPRESSION_TYPE：CreateJob 请求的 ScriptModeConfig 中指定的压缩类型。
- AMZN_BRAKET_SCRIPT_S3_URI：CreateJob 请求的 ScriptModeConfig 中指定的用户脚本的 Amazon S3 位置。
- AMZN_BRAKET_TASK_RESULTS_S3_URI：SDK 默认存储任务的量子任务结果的 Amazon S3 位置。
- AMZN_BRAKET_JOB_RESULTS_S3_PATH：存储任务结果的 Amazon S3 位置，如 CreateJob 请求的 OutputDataConfig 中所述。
- AMZN_BRAKET_JOB_TOKEN：对于在作业容器中创建的量子任务，应传递给的 CreateQuantumTask 的 jobToken 参数的字符串。

辅助函数

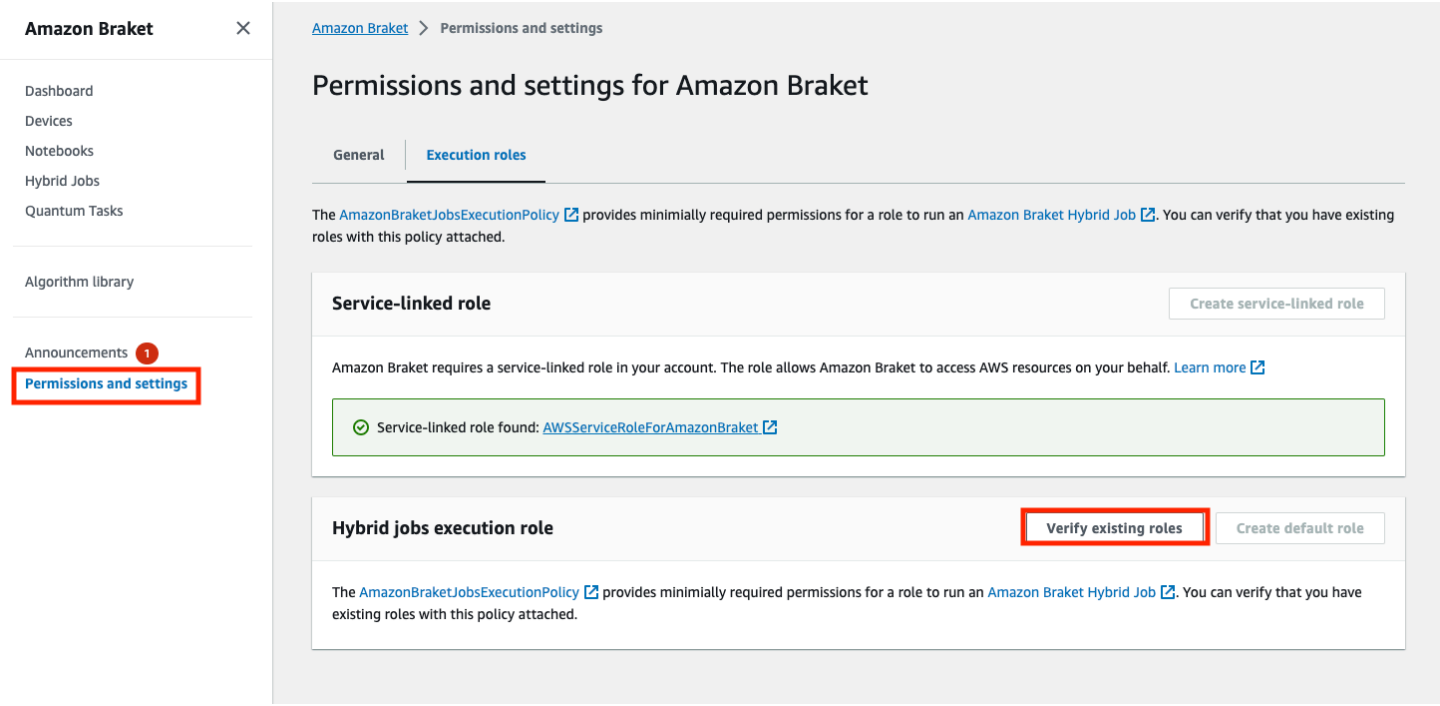
Amazon Braket 提供了多个辅助函数，以简化与容器输入和输出的交互。这些辅助函数将从用于运行 Hybrid Job 的算法脚本中调用。以下示例演示了如何执行此操作。

```
from braket.jobs import get_checkpoint_dir, get_hyperparameters, get_input_data_dir,
 get_job_device_arn, get_job_name, get_results_dir, save_job_result,
 save_job_checkpoint, load_job_checkpoint
```

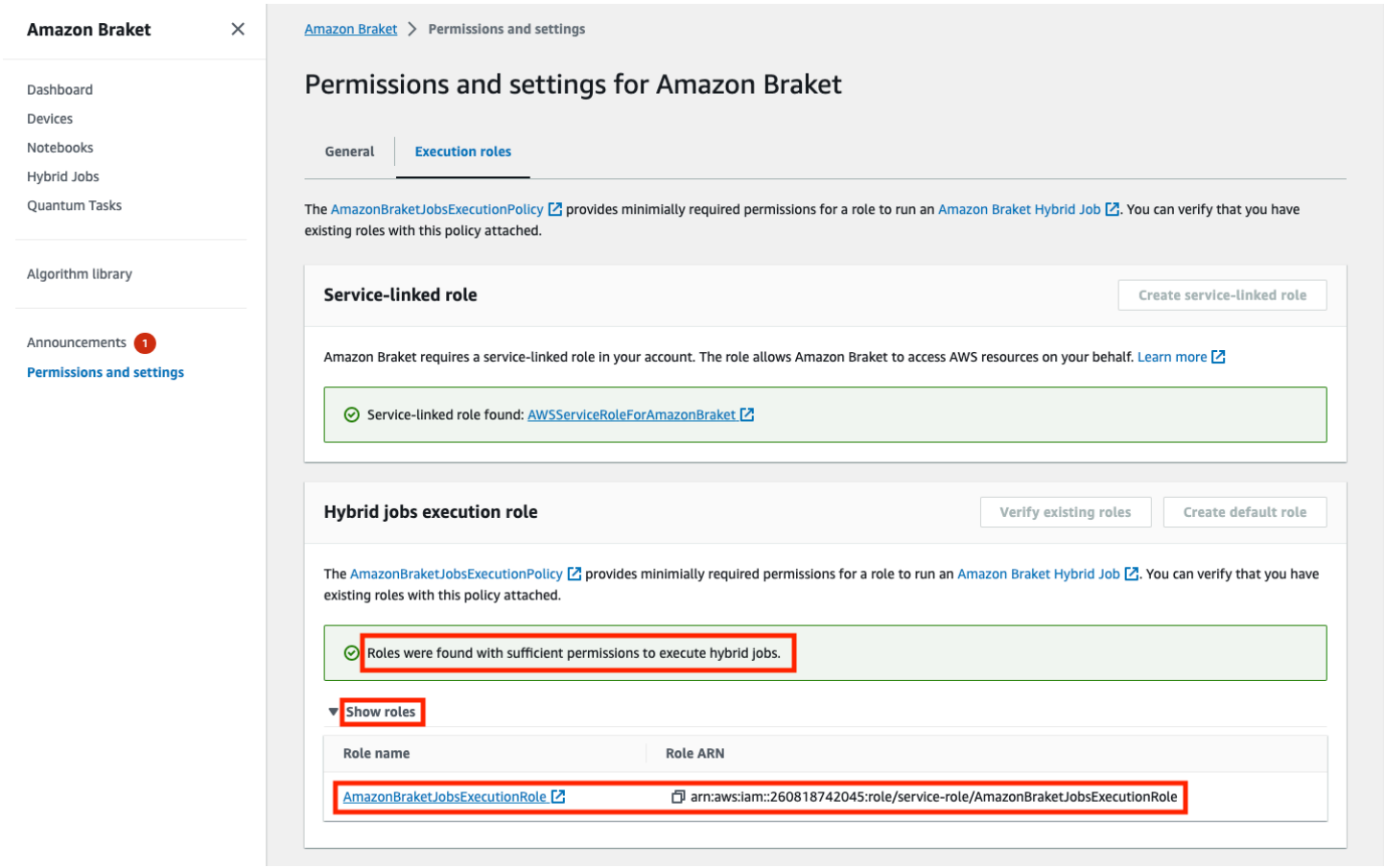
```
get_checkpoint_dir() # Get the checkpoint directory
get_hyperparameters() # Get the hyperparameters as strings
get_input_data_dir() # Get the input data directory
get_job_device_arn() # Get the device specified by the hybrid job
get_job_name() # Get the name of the hybrid job.
get_results_dir() # Get the path to a results directory
save_job_result(result_data='data') # Save hybrid job results
save_job_checkpoint(checkpoint_data={'key': 'value'}) # Save a checkpoint
load_job_checkpoint() # Load a previously saved checkpoint
```

先决条件

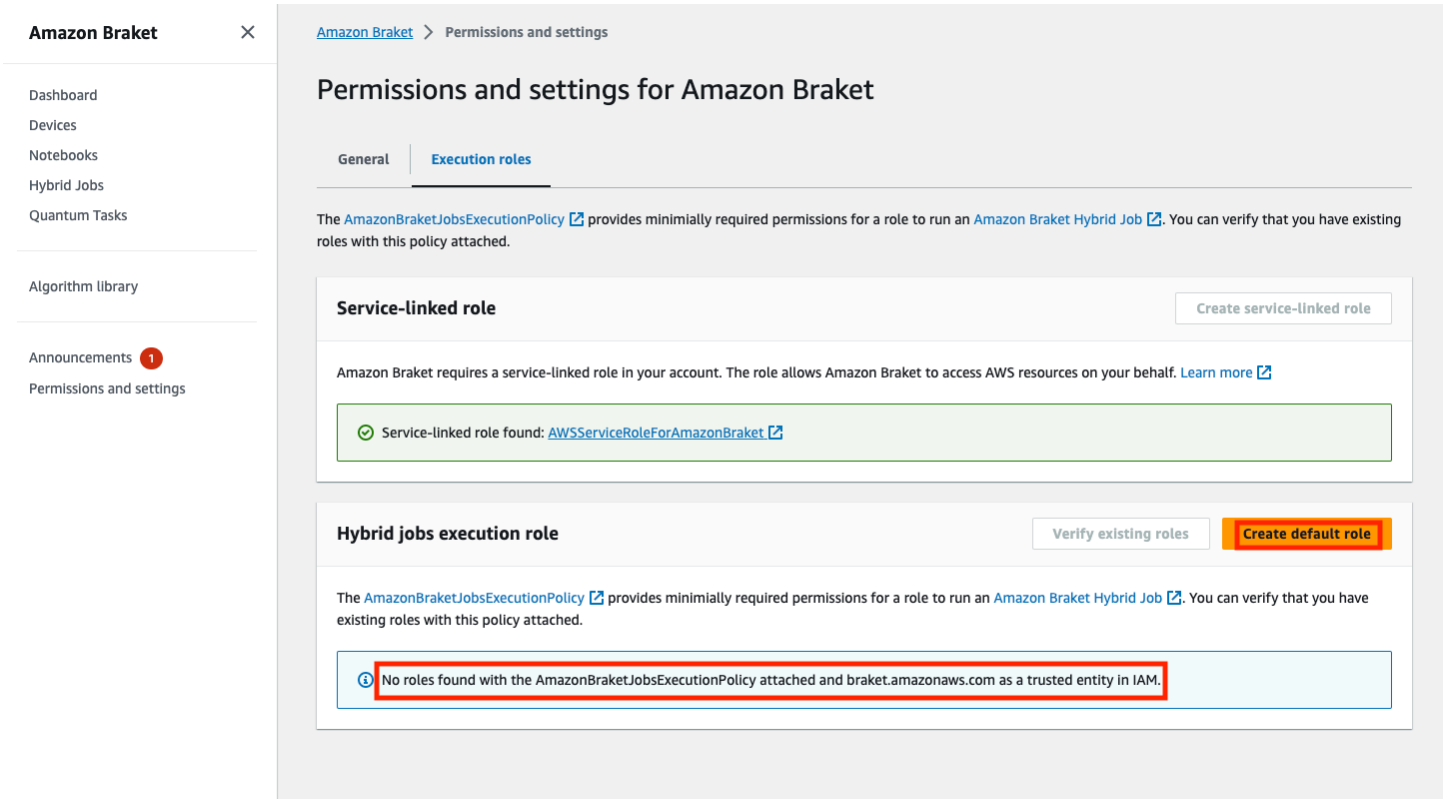
运行第一个混合作业之前，必须确保您具有足够的权限以继续执行此任务。要确定您的权限是否正确，请从 Braket 控制台左侧的菜单中选择“权限”。“Amazon Braket 的权限管理”页面可帮助您验证您的某个现有角色是否具有足以运行混合任务的权限，或者指导您创建可用于运行混合任务的默认角色（如果您还没有此类角色）。



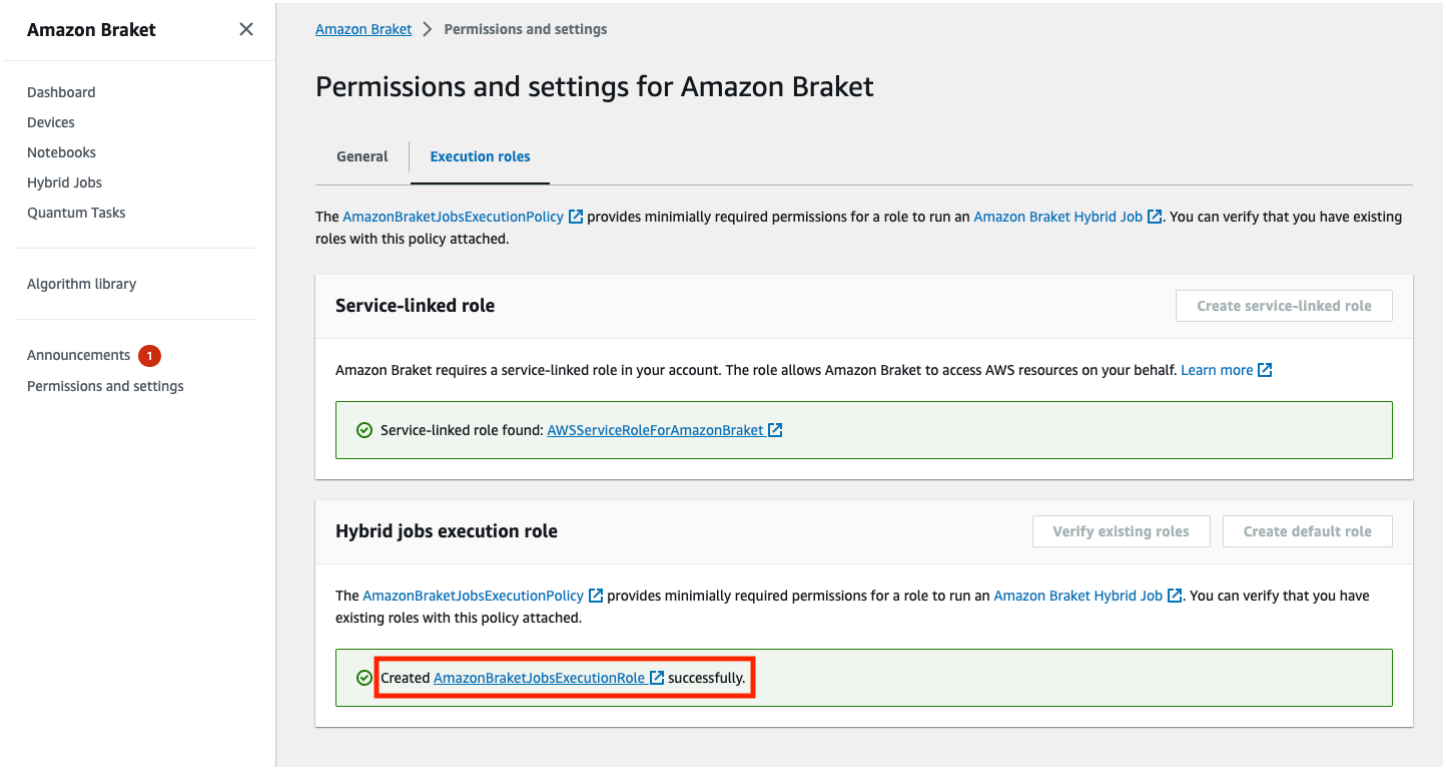
要验证您的角色是否有足够的权限来运行混合作业，请选择验证现有角色按钮。如果这样做，您会收到一条消息，说明角色已找到。要查看角色的名称及其角色 ARNs，请选择显示角色按钮。



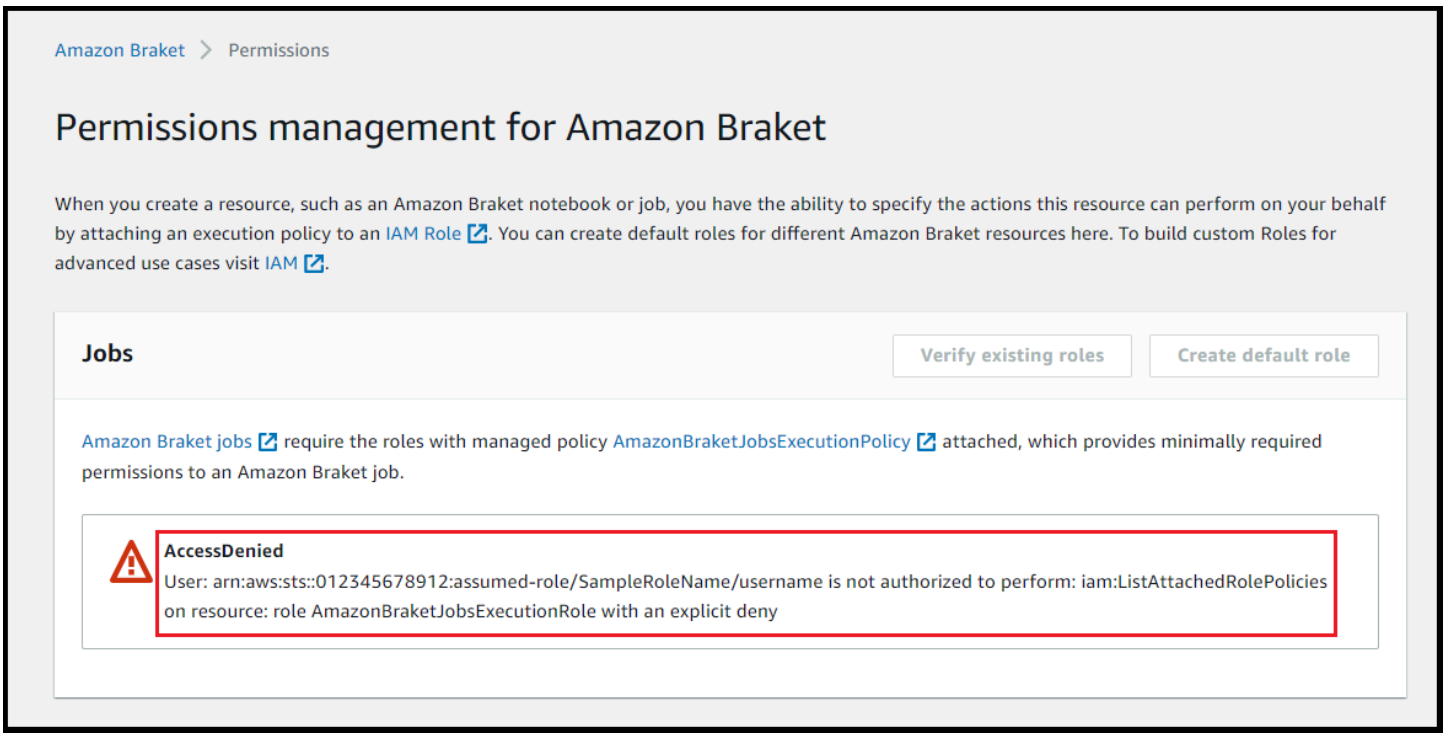
如果您的角色没有足够的权限来运行混合作业，则会收到一条消息，提示未找到此类角色。选择“创建默认角色”按钮，获取具有足够权限的角色。



如果角色已成功创建，您将收到一条确认消息。



如果您无权进行此查询，则将被拒绝访问。在这种情况下，请联系您的内部 AWS 管理员。



创建混合作业

本节将向您介绍如何使用 Python 脚本创建混合作业。或者，要使用本地 Python 代码 [例如您首选的集成式开发环境（IDE）或 Braket Notebook] 创建混合作业，请参阅 将本地代码作为混合作业运行。

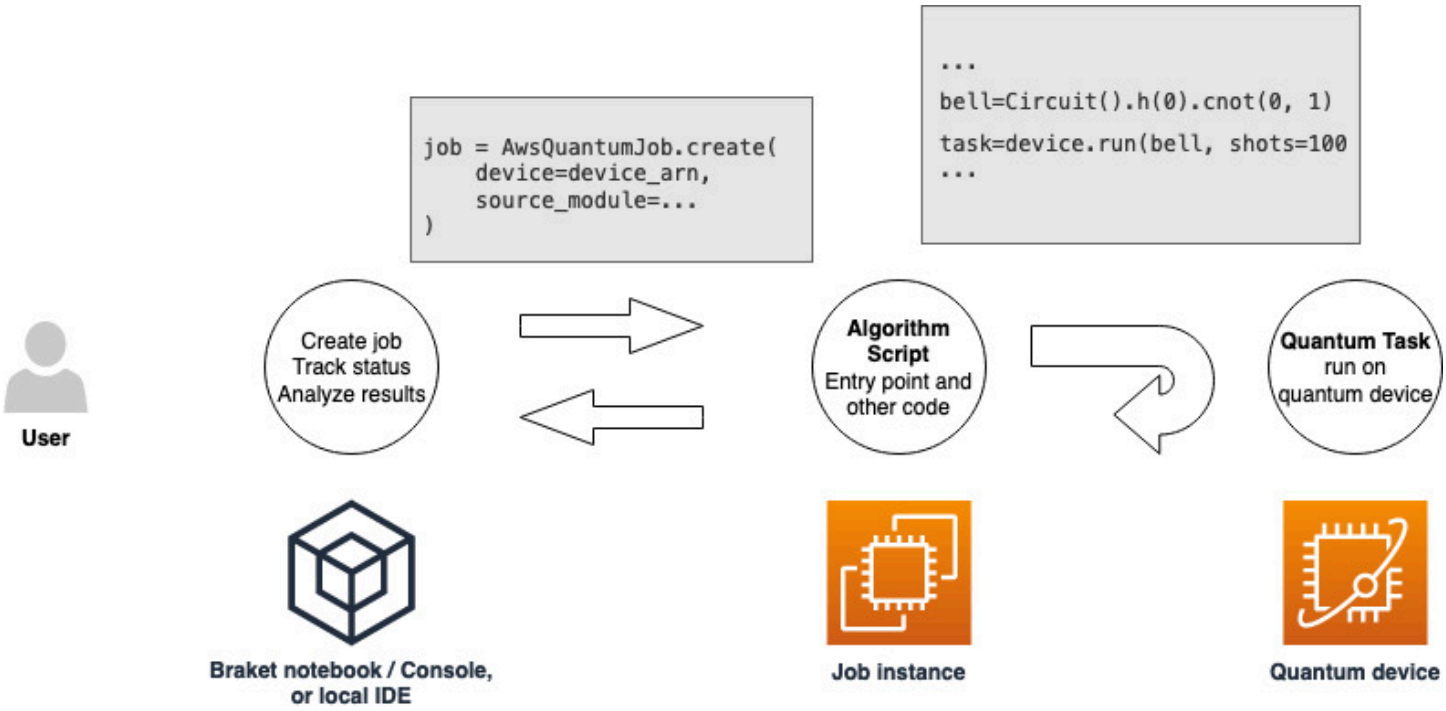
本节内容：

- 创建并运行
- 监控结果
- 保存结果
- 使用检查点
- 将本地代码作为混合作业运行
- 将 API 和 Hybrid Jobs 配合使用
- 使用本地模式创建和调试混合作业

创建并运行

拥有运行混合作业权限的角色后，就可以继续操作了。您的第一个 Braket 混合作业的关键部分是算法脚本。它定义了您要运行的算法，并包含作为算法一部分的经典逻辑和量子任务。除算法脚本外，还可

以提供其他依赖关系文件。算法脚本及其依赖关系被称为源模块。入口点定义了混合作业启动时要在源模块中运行的第一个文件或函数。



首先，考虑以下算法脚本的基本示例，该脚本创建了五个钟形状态并打印相应的测量结果。

```
import os

from braket.aws import AwsDevice
from braket.circuits import Circuit


def start_here():

    print("Test job started!")

    # Use the device declared in the job script
    device = AwsDevice(os.environ["AMZN_BRAKET_DEVICE_ARN"])

    bell = Circuit().h(0).cnot(0, 1)
    for count in range(5):
        task = device.run(bell, shots=100)
        print(task.result().measurement_counts)

    print("Test job completed!")
```

将此文件名为 algorithm_script.py 的文件保存在 Braket Notebook 或本地环境的当前工作目录中。algorithm_script.py 文件已将 `start_here()` 作为计划的入口点。

接下来，在与 algorithm_script.py 文件相同的目录下创建 Python 文件或 Python Notebook。此脚本启动混合作业并处理任何异步处理，例如打印我们感兴趣的状态或关键结果。此脚本至少需要指定您的混合作业脚本和主设备。

> **Note**
>
> 有关如何创建 Braket Notebook 或将文件（如 algorithm_script.py 文件）上传到与 Notebook 相同的目录中的更多信息，请参阅使用 Amazon Braket Python SDK 运行您的第一个电路

对于基本的第一种情况，您的目标是模拟器。无论您瞄准的是哪种类型的量子设备，无论是模拟器还是实际的量子处理单元（QPU），您在以下脚本 `device` 中指定的设备都用于调度混合作业，并且可以作为环境变量 `AMZN_BRAKET_DEVICE_ARN` 提供给算法脚本。

> **Note**
>
> 您只能使用混合作业中 AWS 区域 可用的设备。Amazon Braket SDK 会自动选择此 AWS 区域。例如，us-east-1 中的混合作业可以使用 IonQ、SV1、DM1 和 TN1 设备，但不能使用 Rigetti 设备。

如果您选择量子计算机而不是模拟器，Braket 会安排您的混合作业，以优先访问权限运行其所有量子任务。

```
from braket.aws import AwsQuantumJob
from braket.devices import Devices

job = AwsQuantumJob.create(
    Devices.Amazon.SV1,
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    wait_until_complete=True
)
```

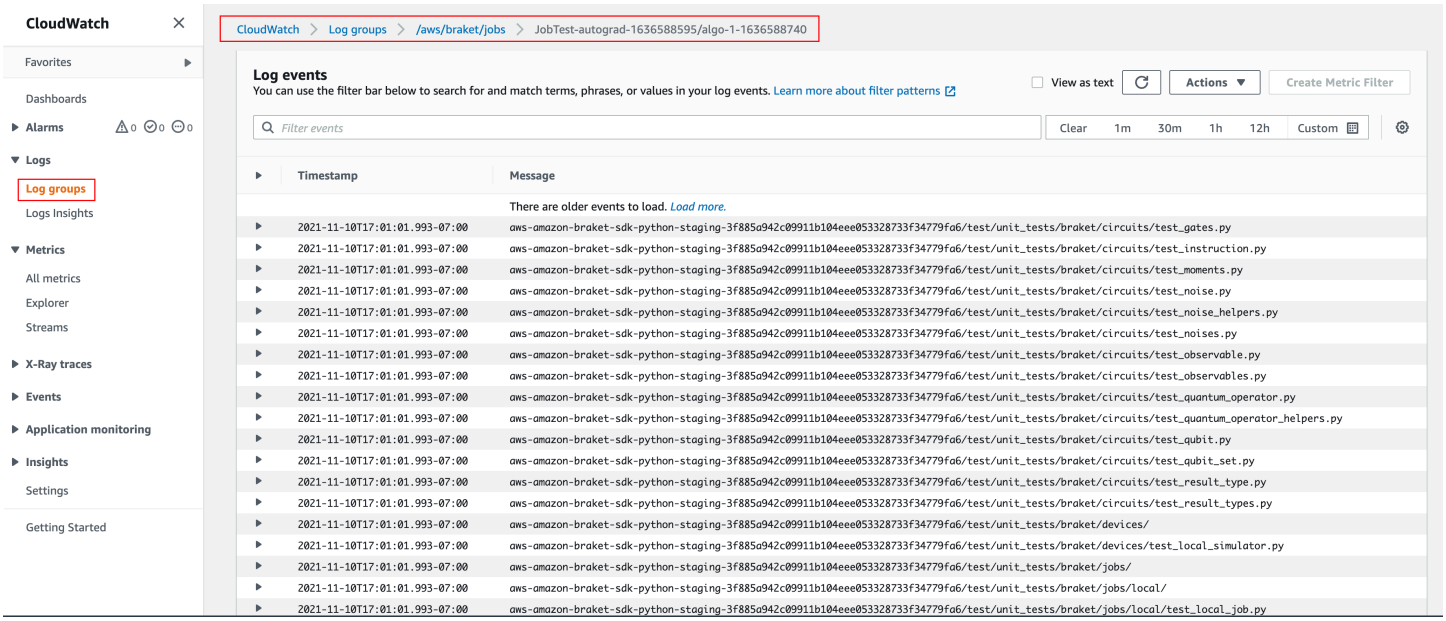
参数 `wait_until_complete=True` 设置了详细模式，以便您的作业在运行时打印实际作业的输出。您应该可以看到类似于以下示例的输出。

```
Initializing Braket Job: arn:aws:braket:us-west-2:111122223333:job/braket-job-
default-123456789012
Job queue position: 1
Job queue position: 1
Job queue position: 1
..............
.
.
.
Beginning Setup
Checking for Additional Requirements
Additional Requirements Check Finished
Running Code As Process
Test job started!
Counter({'00': 58, '11': 42})
Counter({'00': 55, '11': 45})
Counter({'11': 51, '00': 49})
Counter({'00': 56, '11': 44})
Counter({'11': 56, '00': 44})
Test job completed!
Code Run Finished
2025-09-24 23:13:40,962 sagemaker-training-toolkit INFO     Reporting training SUCCESS
```

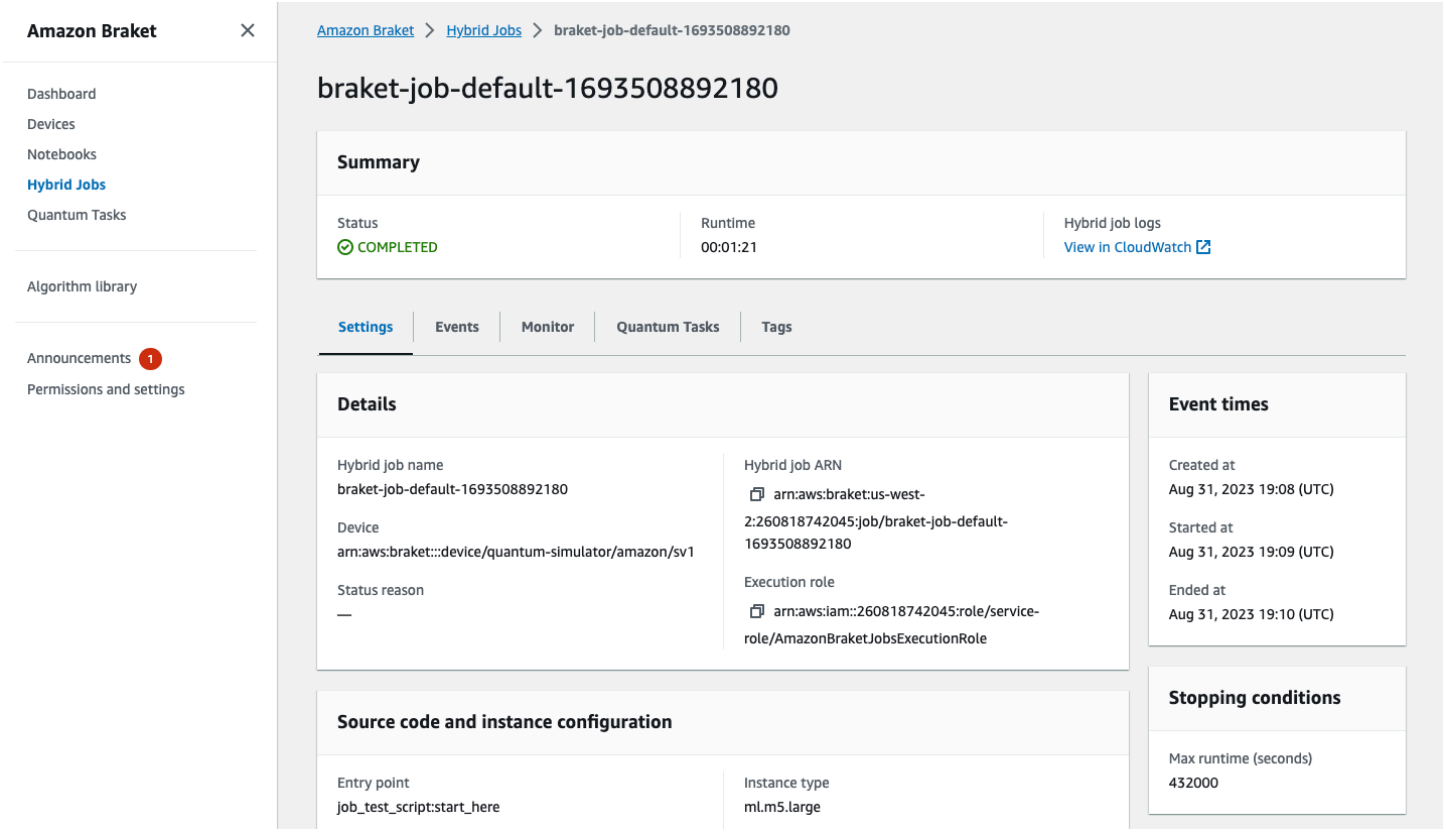
> **Note**
>
> 您还可以通过传递自定义模块的位置（本地目录或文件的路径或 tar.gz 文件的 S3 URI）来使用带有 AwsQuantumJob. create 方法的自定义模块。有关工作示例，请参阅 Amazon Braket 示例 Github 存储库中混合作业文件夹中的 Parallelize_training_for_QML.ipynb 文件。

监控结果

或者，您可以访问来自 Amazon 的日志输出 CloudWatch。为此，请转到作业详细信息页面左侧菜单上的“日志组”选项卡，选择“日志组 aws/braket/jobs”，然后选择包含该作业名称的日志流。在上述示例中，即为 braket-job-default-1631915042705/algo-1-1631915190。



您还可以在控制台中查看混合作业的状态，方法是选择“混合作业”页面，然后选择“设置”。



您的混合作业在运行时会在 Amazon S3 中生成一些构件。S3 存储桶的默认名称为 `amazon-braket-<region>-<accountid>` 且内容位于 `jobs/<jobname>/<timestamp>` 目录中。使用 Braket

Python SDK 创建混合作业时，您可以通过指定其他 code_location来配置存储这些构件的 S3 位置。

> **Note**
>
> 此 S3 存储桶必须与您的任务脚本位于同一 AWS 区域 位置。

该 jobs/<jobname>/<timestamp> 目录包含一个子文件夹，model.tar.gz 文件中包含入口点脚本的输出。还有一个名为 script 的目录，包含了 source.tar.gz 文件中您的算法脚本构件。实际量子任务的结果位于名为 jobs/<jobname>/tasks 的目录中。

保存结果

您可以保存算法脚本生成的结果，以便从混合作业脚本中的混合作业对象以及 Amazon S3 的输出文件夹（名为 model.tar.gz 的 tar 压缩文件中）获得这些结果。

必须使用 JavaScript 对象表示法 (JSON) 格式将输出保存在文件中。如果数据无法轻易序列化为文本（如 numpy 数组），则可以传入一个使用腌制数据格式进行序列化的选项。有关更多详细信息，请参阅 t.jobs.data_persistence 模块。

要保存混合作业的结果，请将以下用 #ADD 注释的行添加到 algorithm_script.py 文件中。

```
import os

from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.jobs import save_job_result  # ADD


def start_here():

    print("Test job started!")

    device = AwsDevice(os.environ['AMZN_BRAKET_DEVICE_ARN'])

    results = []  # ADD

    bell = Circuit().h(0).cnot(0, 1)
    for count in range(5):
        task = device.run(bell, shots=100)
        print(task.result().measurement_counts)
```

```
        results.append(task.result().measurement_counts)  # ADD

        save_job_result({"measurement_counts": results})  # ADD

    print("Test job completed!")
```

然后，您可以通过附加带有 #ADD 注释的行 **print(job.result())** 来显示作业脚本中的作业结果。

```
import time
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
    device="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
)

print(job.arn)
while job.state() not in AwsQuantumJob.TERMINAL_STATES:
    print(job.state())
    time.sleep(10)

print(job.state())
print(job.result())   # ADD
```

在此示例中，我们移除了 wait_until_complete=True，以抑制冗余输出。您可以将其重新添加以进行调试。当您运行该混合作业时，它会每隔 10 秒输出一次标识符和 job-arn，然后输出混合作业的状态，直到混合作业的状态为 COMPLETED 为止，然后它会向您显示钟形回路的结果。请参阅以下示例。

```
arn:aws:braket:us-west-2:111122223333:job/braket-job-default-123456789012
INITIALIZED
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
RUNNING
```

```
RUNNING
...
RUNNING
RUNNING
COMPLETED
{'measurement_counts': [{'11': 53, '00': 47},..., {'00': 51, '11': 49}]}
```

使用检查点

您可以使用检查点保存混合作业的中间迭代。在上一节的算法脚本示例中，您添加了以下用 #ADD 注释的行来创建检查点文件。

```
from braket.aws import AwsDevice
from braket.circuits import Circuit
from braket.jobs import save_job_checkpoint  # ADD
import os


def start_here():

    print("Test job starts!")

    device = AwsDevice(os.environ["AMZN_BRAKET_DEVICE_ARN"])

    # ADD the following code
    job_name = os.environ["AMZN_BRAKET_JOB_NAME"]
    save_job_checkpoint(checkpoint_data={"data": f"data for checkpoint from
 {job_name}"}, checkpoint_file_suffix="checkpoint-1")  # End of ADD

    bell = Circuit().h(0).cnot(0, 1)
    for count in range(5):
        task = device.run(bell, shots=100)
        print(task.result().measurement_counts)

    print("Test hybrid job completed!")
```

当您运行混合作业时，它会在检查点目录中的混合作业构件中使用默认 `/opt/jobs/checkpoints` 路径创建文件 <jobname>-checkpoint-1.json。除非您要更改此默认路径，否则混合作业脚本保持不变。

如果要从之前的混合作业生成的检查点加载混合作业，则算法脚本会使用 `from braket.jobs import load_job_checkpoint`。加载到算法脚本中的逻辑如下所示。

```
from braket.jobs import load_job_checkpoint

checkpoint_1 = load_job_checkpoint(
    "previous_job_name",
    checkpoint_file_suffix="checkpoint-1",
)
```

加载此检查点后，您可以根据加载到 checkpoint-1 的内容继续执行逻辑。

> **Note**
>
> checkpoint_file_suffix 必须与之前在创建检查点时指定的后缀匹配。

您的编排脚本需要指定前一个混合作业中的 job-arn，该行用 #ADD 注释。

```
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    source_module="source_dir",
    entry_point="source_dir.algorithm_script:start_here",
    device="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
    copy_checkpoints_from_job="<previous-job-ARN>", #ADD
    )
```

将本地代码作为混合作业运行

Amazon Braket Hybrid Jobs 提供了混合量子经典算法的完全托管编排，将 Amazon EC2 计算资源与 Amazon Braket 量子处理单元（QPU）访问权限相结合。在混合作业中创建的量子任务优先于单个量子任务，因此您的算法不会因量子任务队列的波动而中断。每个 QPU 都维护一个单独的混合作业队列，确保在任何给定时间只能运行一个混合作业。

本节内容：

- 使用本地 Python 代码创建混合作业
- 安装其他 Python 软件包和源代码
- 将数据保存并加载到混合作业实例中
- 混合作业修饰器最佳实践

使用本地 Python 代码创建混合作业

您可以将本地 Python 代码作为 Amazon Braket Hybrid Jobs 运行。您可以使用 @hybrid_job 装饰器为代码添加注释来做到这一点，如以下代码示例中所示。对于自定义环境，您可以选择使用 Amazon Elastic Registry（ECR）中的自定义容器。

> **Note**
>
> 默认情况下，仅支持 Python 3.12。

您可以使用 @hybrid_job 装饰器为函数添加注释。Braket 将装饰器内部的代码转换为 Braket 混合作业算法脚本。然后，混合作业在 Amazon EC2 实例上调用装饰器内部的函数。您可以使用 `job.state()` 或 Braket 控制台监控作业进度。以下代码示例显示如何在 State Vector Simulator (SV1) device 上运行由五个状态组成的序列。

```
from braket.aws import AwsDevice
from braket.circuits import Circuit, FreeParameter, Observable
from braket.devices import Devices
from braket.jobs.hybrid_job import hybrid_job
from braket.jobs.metrics import log_metric

device_arn = Devices.Amazon.SV1


@hybrid_job(device=device_arn)  # Choose priority device
def run_hybrid_job(num_tasks=1):
    device = AwsDevice(device_arn)  # Declare AwsDevice within the hybrid job

    # Create a parametric circuit
    circ = Circuit()
    circ.rx(0, FreeParameter("theta"))
    circ.cnot(0, 1)
    circ.expectation(observable=Observable.X(), target=0)

    theta = 0.0  # Initial parameter

    for i in range(num_tasks):
        task = device.run(circ, shots=100, inputs={"theta": theta})  # Input parameters
        exp_val = task.result().values[0]

        theta += exp_val  # Modify the parameter (possibly gradient descent)
```

```
        log_metric(metric_name="exp_val", value=exp_val, iteration_number=i)

    return {"final_theta": theta, "final_exp_val": exp_val}
```

您可以像普通 Python 函数一样通过调用函数来创建混合作业。但是，装饰器函数返回的是混合任务句柄而不是函数的结果。要在结果完成后检索结果，请使用 `job.result()`。

```
job = run_hybrid_job(num_tasks=1)
result = job.result()
```

@hybrid_job 装饰器中的 device 参数指定混合作业可以优先访问的设备，在本例中为 SV1 模拟器。要获得 QPU 优先级，必须确保函数中使用的设备 ARN 与装饰器中指定的设备 ARN 相匹配。为方便起见，您可以使用辅助函数 `get_job_device_arn()` 来捕获 @hybrid_job 中声明的设备 ARN。

> **Note**
>
> 自从在 Amazon EC2 上创建容器化环境以来，每个混合任务至少有一分钟的启动时间。因此，对于非常短的工作负载，例如单个电路或一批电路，使用量子任务可能就足够了。

超级参数

该 `run_hybrid_job()` 函数采用参数 `num_tasks` 来控制创建的量子任务的数量。混合作业会自动将其捕获为超参数。

> **Note**
>
> 超参数在 Braket 控制台上显示为字符串，限制为 2500 个字符。

指标和日志

在 `run_hybrid_job()` 函数中，使用 `log_metrics` 记录来自迭代算法的指标。相关指标会自动绘制在 Braket 控制台页面的“混合作业”选项卡下。在混合作业运行期间，您可以使用 Braket 成本跟踪器近乎实时地跟踪量子任务成本。上面的示例使用指标名称“概率”来记录结果类型的第一个概率。

检索结果

混合作业完成后，您可以使用 `job.result()` 检索混合作业的结果。Braket 会自动捕获返回语句中的任何对象。请注意，该函数返回的对象必须是一个元组，每个元素都是可序列化的。例如，以下代码显示了一个正在工作和一个失败的示例。

```
import numpy as np


# Working example
@hybrid_job(device=Devices.Amazon.SV1)
def passing():
    np_array = np.random.rand(5)
    return np_array  # Serializable

# # Failing example
# @hybrid_job(device=Devices.Amazon.SV1)
# def failing():
#     return MyObject() # Not serializable
```

作业名称

默认情况下，此混合作业的名称是根据函数名称推断出来的。您还可以指定长度最多为 50 个字符的名称。例如，在以下代码中，作业名称为 “my-job-name”。

```
@hybrid_job(device=Devices.Amazon.SV1, job_name="my-job-name")
def function():
    pass
```

本地模式

通过向装饰器添加参数 `local=True` 来创建本地作业。这样可以在本地计算环境（如 Notebook）的容器化环境中运行混合作业。本地作业没有优先排队等候量子任务。对于多节点或 MPI 等高级案例，本地作业可以访问所需的 Braket 环境变量。以下代码使用设备作为 SV1 模拟器创建本地混合作业。

```
@hybrid_job(device=Devices.Amazon.SV1, local=True)
def run_hybrid_job(num_tasks=1):
    return ...
```

支持所有其他混合作业选项。有关选项列表，请参阅 braket.jobs.quantum_job_creation 模块。

安装其他 Python 软件包和源代码

您可以自定义运行时环境以使用首选 Python 包。您可以使用 `requirements.txt` 文件、软件包名称列表或自带容器（BYOC）。例如，`requirements.txt` 文件可能包含其他要安装的软件包。

```
qiskit
pennylane >= 0.31
mitiq == 0.29
```

要使用 `requirements.txt` 文件自定义运行时系统环境，请参阅以下代码示例。

```
@hybrid_job(device=Devices.Amazon.SV1, dependencies="requirements.txt")
def run_hybrid_job(num_tasks=1):
    return ...
```

您还可以按如下方式以 Python 列表的形式提供软件包名称。

```
@hybrid_job(device=Devices.Amazon.SV1, dependencies=["qiskit", "pennylane>=0.31",
 "mitiq==0.29"])
def run_hybrid_job(num_tasks=1):
    return ...
```

其他源代码可以指定为模块列表，也可以指定为单个模块，如以下代码示例所示。

```
@hybrid_job(device=Devices.Amazon.SV1, include_modules=["my_module1", "my_module2"])
def run_hybrid_job(num_tasks=1):
    return ...
```

将数据保存并加载到混合作业实例中

指定输入训练数据

创建混合作业时，您可以通过指定 Amazon Simple Storage Service (Amazon S3) 存储桶，提供输入训练数据集。您也可以指定本地路径。然后，Braket 会自动将数据上传到 Amazon S3，URL 为 `s3://<default_bucket_name>/jobs/<job_name>/<timestamp>/data/<channel_name>`。如果您指定本地路径，则频道名称默认为“input”。以下代码显示了来自本地路径 `data/file.npy` 的 numpy 文件。

```
import numpy as np
```

```
@hybrid_job(device=Devices.Amazon.SV1, input_data="data/file.npy")
def run_hybrid_job(num_tasks=1):
    data = np.load("data/file.npy")
    return ...
```

对于 S3，必须使用 `get_input_data_dir()` 辅助函数。

```
import numpy as np
from braket.jobs import get_input_data_dir

s3_path = "s3://amazon-braket-us-east-1-123456789012/job-data/file.npy"


@hybrid_job(device=None, input_data=s3_path)
def job_s3_input():
    np.load(get_input_data_dir() + "/file.npy")


@hybrid_job(device=None, input_data={"channel": s3_path})
def job_s3_input_channel():
    np.load(get_input_data_dir("channel") + "/file.npy")
```

您可以通过提供通道值和 S3 URIs 或本地路径的字典来指定多个输入数据源。

```
import numpy as np
from braket.jobs import get_input_data_dir

input_data = {
    "input": "data/file.npy",
    "input_2": "s3://amzn-s3-demo-bucket/data.json"
}


@hybrid_job(device=None, input_data=input_data)
def multiple_input_job():
    np.load(get_input_data_dir("input") + "/file.npy")
    np.load(get_input_data_dir("input_2") + "/data.json")
```

> **Note**
>
> 当输入数据很大 (>1GB) 时，需要等待很长时间才能创建作业。这是由于本地输入数据首次上传到 S3 存储桶，然后将 S3 路径添加到任务请求中。最后，工作请求将提交给 Braket 服务。

将结果保存到 S3

要保存未包含在装饰函数的返回语句中的结果，必须将正确的目录附加到所有文件写入操作中。以下示例显示了如何保存一个 numpy 数组和 matplotlib 图。

```
import matplotlib.pyplot as plt
import numpy as np


@hybrid_job(device=Devices.Amazon.SV1)
def run_hybrid_job(num_tasks=1):
    result = np.random.rand(5)

    # Save a numpy array
    np.save("result.npy", result)

    # Save a matplotlib figure
    plt.plot(result)
    plt.savefig("fig.png")
    return ...
```

所有结果都压缩到名为 `model.tar.gz` 的文件中。您可以使用 Python 函数 `job.result()` 下载结果，也可以从 Braket 管理控制台的混合作业页面导航到结果文件夹。

保存并从检查点恢复

对于长时间运行的混合作业，建议定期保存算法的中间状态。您可以使用内置的 `save_job_checkpoint()` 辅助函数，也可以将文件保存到 `AMZN_BRAKET_JOB_RESULTS_DIR` 路径中。后者可通过辅助函数 `get_job_results_dir()` 获得。

以下是使用混合作业装饰器保存和加载检查点的最小工作示例：

```
from braket.jobs import save_job_checkpoint, load_job_checkpoint, hybrid_job


@hybrid_job(device=None, wait_until_complete=True)
```

```
def function():
    save_job_checkpoint({"a": 1})


job = function()
job_name = job.name
job_arn = job.arn


@hybrid_job(device=None, wait_until_complete=True, copy_checkpoints_from_job=job_arn)
def continued_function():
    load_job_checkpoint(job_name)


continued_job = continued_function()
```

在第一个混合作业中，使用包含我们要保存的数据的字典调用 `save_job_checkpoint()`。默认情况下，每个值都必须可序列化为文本。对于检查点更为复杂的 Python 对象，如 numpy 数组，您可以设置 `data_format = PersistedJobDataFormat.PICKLED_V4`。此代码在名为 `<jobname>.json` 的子文件夹下的混合作业构件中创建并覆盖具有默认名称的检查点文件。

要创建一个新的混合作业以从检查点继续，我们需要传递 `copy_checkpoints_from_job=job_arn` 到 `job_arn` 作为前一个作业的混合作业 ARN 的位置。然后，我们使用 `load_job_checkpoint(job_name)` 从检查点加载。

混合作业修饰器最佳实践

接受异步性

使用装饰器注释创建的混合作业是异步的，它们将在经典资源和量子资源可用后运行。您可以使用Braket Management Console或 Amazon 监控算法的进度 CloudWatch。当您提交算法以供运行时，Braket 会在可扩展的容器化环境中运行您的算法，并在算法完成后检索结果。

运行迭代变分算法

混合作业为您提供了运行迭代量子经典算法的工具。对于纯粹的量子问题，请使用量子任务或一批量子任务。优先访问某些 QPUs 算法对于需要多次迭代调用并在两者之间进行经典处理的长时间运行的变分算法最 QPUs 有利。

使用本地模式进行调试

在 QPU 上运行混合作业之前，建议先在模拟器上运行 SV1 以确认其按预期运行。对于小规模测试，可以在本地模式下运行，以实现快速迭代和调试。

使用自带容器（BYOC）提升可重复性

将您的软件及其依赖项封装在容器化环境中，从而创建可重现的实验。通过将所有代码、依赖项和设置打包到容器中，可以防止潜在的冲突和版本控制问题。

多实例分布式模拟器

要运行大量电路，可以考虑使用内置的 MPI 支持，在单个混合作业中的多个实例上运行本地模拟器。有关更多信息，请参阅嵌入式模拟器。

使用参数电路

您从混合作业中提交的参数电路会自动 QPUs 使用参数化编译进行编译，以改善算法的运行时间。

定期检查点

对于长时间运行的混合作业，建议定期保存算法的中间状态。

有关更多示例、用例和最佳实践，请参阅 Amazon Bra GitHub ket 示例。

将 API 和 Hybrid Jobs 配合使用

您可以直接使用 API Amazon Braket Hybrid Jobs 访问并与之交互。但是，直接使用 API 时，默认方法和便捷方法不可用。

> **Note**
>
> 强烈建议您使用 Amazon Braket Python SDK 与 Amazon Braket Hybrid Jobs 互动。它提供便捷的默认设置和保护功能，可帮助您的混合作业成功运行。

本主题涵盖了关于使用 API 的基础知识。如果您选择使用 API，请记住这种方法可能更为复杂，并且需要为几次迭代做好准备，以使您的混合作业得以运行。

要使用 API，您的账户应具有 `AmazonBraketFullAccess` 托管式策略的角色。

> **Note**
>
> 有关如何使用 `AmazonBraketFullAccess` 托管式策略获取角色的更多信息，请参阅启用 Amazon Braket 页面。

此外，您还需要一个执行角色。此角色将被传递给服务。您可以使用 Amazon Braket 控制台来创建角色。使用“权限和设置”页面上的“执行角色”选项卡为混合作业创建默认角色。

CreateJob API 要求您为混合作业指定所有必需的参数。要使用 Python，请将算法脚本文件压缩为 tar 包，如 input.tar.gz 文件，然后运行以下脚本。更新尖括号 (<>) 内的代码部分，以匹配您的账户信息和指定混合作业开始路径、文件和方法的入口点。

```
from braket.aws import AwsDevice, AwsSession
import boto3
from datetime import datetime

s3_client = boto3.client("s3")
client = boto3.client("braket")

project_name = "job-test"
job_name = project_name + "-" + datetime.strftime(datetime.now(), "%Y%m%d%H%M%S")
bucket = "amazon-braket-<your_bucket>"
s3_prefix = job_name

job_script = "input.tar.gz"
job_object = f"{s3_prefix}/script/{job_script}"
s3_client.upload_file(job_script, bucket, job_object)

input_data = "inputdata.csv"
input_object = f"{s3_prefix}/input/{input_data}"
s3_client.upload_file(input_data, bucket, input_object)

job = client.create_job(
    jobName=job_name,
    roleArn="arn:aws:iam::<your_account>:role/service-role/
AmazonBraketJobsExecutionRole",  # https://docs.aws.amazon.com/braket/latest/
developerguide/braket-manage-access.html#about-amazonbraketjobsexecution
    algorithmSpecification={
        "scriptModeConfig": {
            "entryPoint": "<your_execution_module>:<your_execution_method>",
            "containerImage": {"uri": "292282985366.dkr.ecr.us-west-1.amazonaws.com/
amazon-braket-base-jobs:1.0-cpu-py37-ubuntu18.04"},   # Change to the specific region
 you are using
            "s3Uri": f"s3://{bucket}/{job_object}",
            "compressionType": "GZIP"
        }
    },
    inputDataConfig=[
```

```
        {
            "channelName": "hellothere",
            "compressionType": "NONE",
            "dataSource": {
                "s3DataSource": {
                    "s3Uri": f"s3://{bucket}/{s3_prefix}/input",
                    "s3DataType": "S3_PREFIX"
                }
            }
        }
    ],
    outputDataConfig={
        "s3Path": f"s3://{bucket}/{s3_prefix}/output"
    },
    instanceConfig={
        "instanceType": "ml.m5.large",
        "instanceCount": 1,
        "volumeSizeInGb": 1
    },
    checkpointConfig={
        "s3Uri":  f"s3://{bucket}/{s3_prefix}/checkpoints",
        "localPath": "/opt/omega/checkpoints"
    },
    deviceConfig={
        "priorityAccess": {
            "devices": [
                "arn:aws:braket:us-west-1::device/qpu/rigetti/Ankaa-3"
            ]
        }
    },
    hyperParameters={
        "hyperparameter key you wish to pass": "<hyperparameter value you wish to
 pass>",
    },
    stoppingCondition={
        "maxRuntimeInSeconds": 1200,
        "maximumTaskLimit": 10
    },
)
```

创建混合作业后，您可以通过 GetJob API 或控制台访问混合作业的详细信息。要从运行 createJob 代码的 Python 会话中获取混合作业的详细信息，请使用以下 Python 命令。

```
getJob = client.get_job(jobArn=job["jobArn"])
```

要取消混合作业，请使用该作业 ('JobArn') Amazon Resource Name 调用 `CancelJob` API。

```
cancelJob = client.cancel_job(jobArn=job["jobArn"])
```

您可以使用 `checkpointConfig` 参数将检查点指定为 `createJob` API 的一部分。

```
    checkpointConfig = {
        "localPath" : "/opt/omega/checkpoints",
        "s3Uri": f"s3://{bucket}/{s3_prefix}/checkpoints"
    },
```

> **Note**
>
> `checkpointConfig` 的 localPath 不能以下列任何保留路径开头：`/opt/ml`、`/opt/braket`、`/tmp` 或 `/usr/local/nvidia`。

使用本地模式创建和调试混合作业

在构建新的混合算法时，本地模式可帮助您调试和测试算法脚本。本地模式是一项功能，可运行计划在 Amazon Braket Hybrid Jobs 中使用的代码，但不需要 Braket 来管理运行混合作业的基础设施。相反，可以在您的 Amazon Braket Notebook 实例或首选客户端（例如 Notebook 或台式电脑）上本地运行混合作业。

在本地模式下，您仍然可以将量子任务发送到实际设备，但是在本地模式下，在实际的量子处理单元（QPU）上运行时，您将无法获得性能优势。

要使用本地模式，当该模式在程序内部出现时，请将 `AwsQuantumJob` 修改为 `LocalQuantumJob`。例如，要运行创建第一个混合作业中的示例，请按如下方式编辑代码中的混合作业脚本。

```
from braket.jobs.local import LocalQuantumJob

job = LocalQuantumJob.create(
    device="arn:aws:braket:::device/quantum-simulator/amazon/sv1",
    source_module="algorithm_script.py",
    entry_point="algorithm_script:start_here",
```

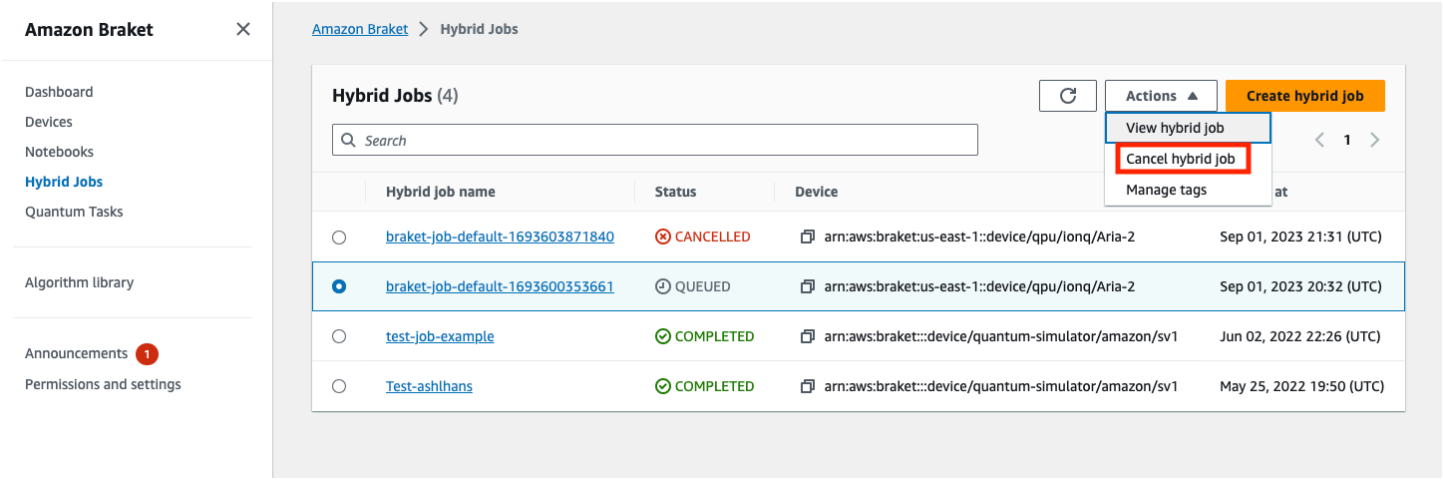
)

> **Note**
>
> Docker 已经预装在 Amazon Braket Notebook 中，需要安装在本地环境中才能使用此功能。Docker 的安装的说明可以在“获取 Docker”页面上找到。此外，并非所有参数在本地模式下都受支持。

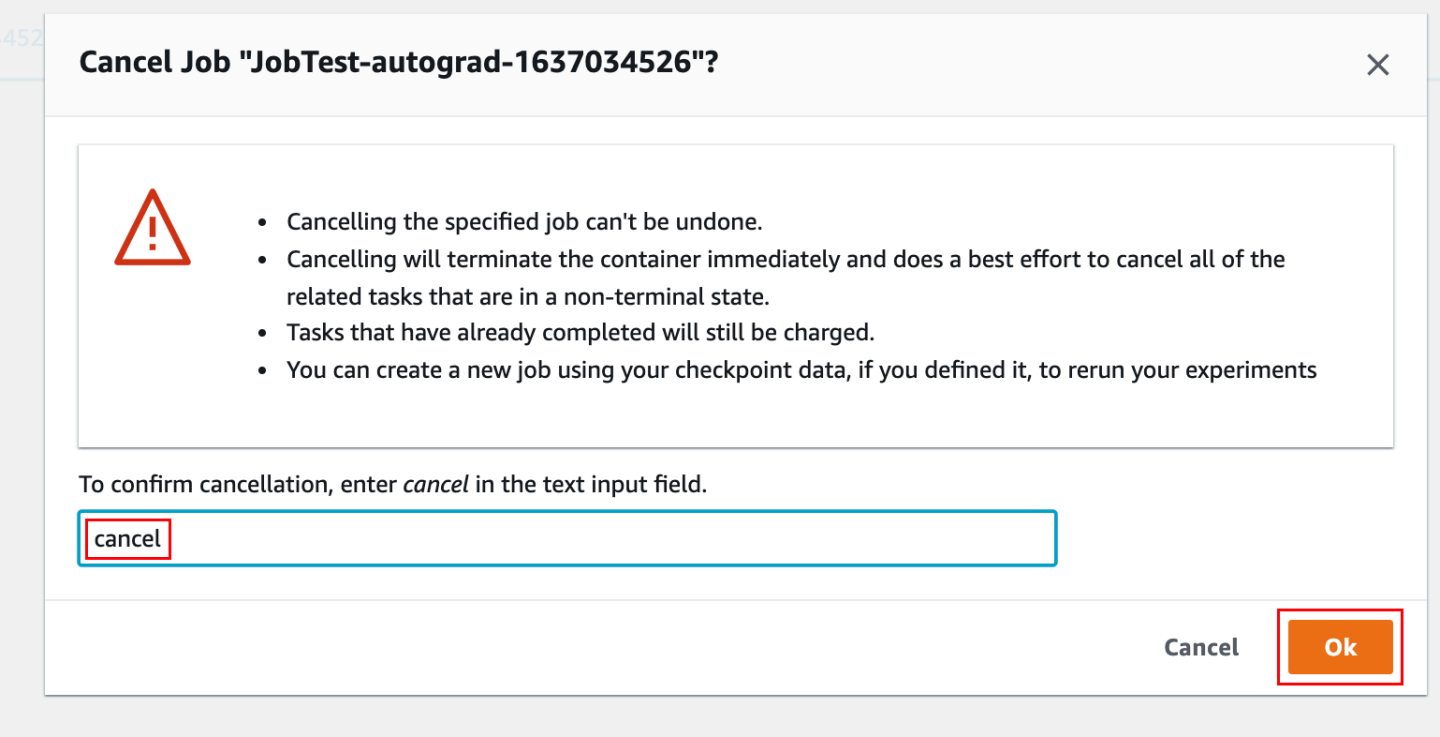
取消混合作业

您可能需要取消处于非终端状态的混合作业。这可以在控制台中完成，也可以使用代码完成。

要在控制台中取消混合作业，请从“混合作业”页面中选择要取消的混合作业，然后从“操作”下拉菜单中选择“取消混合作业”。



要确认取消，请在出现提示时在输入字段中输入“取消”，然后选择“确定”。



要使用 Braket Python SDK 中的代码取消混合作业，请使用 `job_arn` 识别混合作业，然后对其调用 `cancel` 命令，如以下代码所示。

```
job = AwsQuantumJob(arn=job_arn)
job.cancel()
```

`cancel` 命令会立即终止经典的混合作业容器，并尽最大努力取消所有仍处于非终端状态的相关量子任务。

自定义混合作业

Amazon Braket 提供了多种自定义混合任务运行方式的方法，可帮助您根据自己的特定需求定制环境。本节探讨了自定义混合作业的选项，从定义算法脚本环境到自带容器。您将学习到如何使用超参数优化工作流程、配置作业实例以及如何利用参数化编译来提高性能。这些自定义技术可帮助您最大限度地发挥在 Amazon Braket 上进行混合量子计算的潜力。

本节内容：

- 为算法脚本定义环境
- 查看超参数
- 配置您的混合作业实例

- 使用参数化编译加快混合作业的速度

为算法脚本定义环境

Amazon Braket 支持由容器为算法脚本定义的环境：

- 基础容器（如果未指定，则 `image_uri` 为默认容器）
- 装有 CUDA-Q 的容器
- 一个装有 Tensorflow 的容器和 PennyLane
- 一个带有 PyTorch PennyLane、和 CUDA-Q 的容器

下表详细介绍了有关容器及其包含的库的详细信息。

Amazon Braket 容器

Type	Base	CUDA-Q	TensorFlow	PyTorch
映像 URI	292282985366.dkr。ecr.us-west-2.amazonaws.com/:latest amazon-braket-base-jobs	292282985366.dkr。ecr.us-west-2.amazonaws.com/:latest amazon-braket-cudaq-jobs	292282985366.dkr。ecr.us-east-1.amazonaws.com/:latest amazon-braket-tensorflow-jobs	292282985366.dkr。ecr.us-west-2.amazonaws.com/:latest amazon-braket-pytorch-jobs
继承的库		• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk • awscli	• awscli • numpy • pandas • scipy	• awscli • numpy • pandas • scipy

Type	Base	CUDA-Q	TensorFlow	PyTorch
		• botocore • boto3 • dask • matplotlib • numpy • pandas • PennyLane • PennyLane-闪电 • qiskit-braket-provider • 请求 • sagemaker-training • scikit-learn • scipy		

Type	Base	CUDA-Q	TensorFlow	PyTorch
其他库	• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk • awscli • boto3 • ipykernel • matplotlib • networkx • numpy • openbabel • pandas • PennyLane • Protobuf • psi4 • rsa • scipy	• cudaq • cudaq-qec • cudaq-solvers	• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk • ipykernel • keras • matplotlib • networkx • openbabel • PennyLane • Protobuf • psi4 • rsa • PennyLane-闪电般的 GPU • cuQuantum	• amazon-braket-default-simulator • amazon-braket-pennylane-plugin • amazon-braket-schemas • amazon-braket-sdk • ipykernel • keras • matplotlib • networkx • openbabel • PennyLane • Protobuf • psi4 • rsa • PennyLane-闪电般的 GPU • cuQuantum • cudaq • cudaq-qec • cudaq-solvers

您可以在 aws/ amazon-braket-containers 上查看和访问开源容器定义。选择与您的使用案例匹配的容器。你可以使用 Braket 中的任何可用 AWS 区域（us-east-1、us-west-1、us-west-2、eu-north-1、eu-west-2），但容器区域必须与混合作业的区域相匹配。通过在混合作业脚本中的 `create(…)` 调用中添加以下三个参数之一，在创建混合作业时指定容器映像。由于 Amazon Braket

容器有互联网连接，因此您可以在运行时将其他依赖项安装到您选择的容器中（以启动或运行时为代价）。此示例使用 us-west-2 区域。

- 基本图片：image_uri= “292282985366.dkr。ecr.us-west-2.amazonaws.com/: latest” amazon-braket-base-jobs
- CUDA-Q 图片：image_uri= “292282985366.dkr。ecr.us-west-2.amazonaws.com/: latest” amazon-braket-cudaq-jobs
- Tensorflow 图片：image_uri= “292282985366.dkr。ecr.us-west-2.amazonaws.com/: latest” amazon-braket-tensorflow-jobs
- PyTorch 图片：image_uri= “292282985366.dkr。ecr.us-west-2.amazonaws.com/: latest” amazon-braket-pytorch-jobs

`image-uris` 也可以使用 Amazon Braket SDK 中的 `retrieve_image()` 函数进行检索。以下示例说明如何从 us-west- AWS 区域 2 中检索它们。

```
from braket.jobs.image_uris import retrieve_image, Framework

image_uri_base = retrieve_image(Framework.BASE, "us-west-2")
image_uri_cudaq = retrieve_image(Framework.CUDAQ, "us-west-2")
image_uri_tf = retrieve_image(Framework.PL_TENSORFLOW, "us-west-2")
image_uri_pytorch = retrieve_image(Framework.PL_PYTORCH, "us-west-2")
```

使用自己的容器（BYOC）

Amazon Braket Hybrid Jobs 提供了三个预先构建的容器，用于在不同的环境中运行代码。如果其中一个容器支持您的使用案例，则只需在创建混合作业时提供算法脚本即可。可以使用 `pip` 从算法脚本或 `requirements.txt` 文件中添加少量缺失的依赖关系。

如果这些容器都不支持您的使用案例，或者您想对其进行扩展，Braket Hybrid Jobs 支持使用您自己的自定义 Docker 容器映像运行混合作业，或者由您自带容器（BYOC）。请确保它是适合您使用案例的功能。

本节内容：

- 什么时候自带容器才是正确的决定？
- 自带容器指南
- 在自己的容器中运行 Braket 混合作业

什么时候自带容器才是正确的决定？

将自己的容器（BYOC）带到 Braket Hybrid Jobs 中，通过将自己的软件安装在打包环境中，可以灵活地使用自己的软件。根据您的具体需求，一些方法可能能够实现同样的灵活性，而不必经历完整的 BYOC Docker 构建 - Amazon ECR 上传 - 自定义映像 URI 周期。

> **Note**
>
> 如果您想添加少量公开发布的其他 Python 包（通常少于 10 个），BYOC 可能不是正确的选择。例如，如果你正在使用 PyPi.

在这种情况下，您可以使用其中一个预先构建的 Braket 映像，然后在作业提交时在源目录中加入一个 `requirements.txt` 文件。该文件可被自动读取，`pip` 将照常安装具有指定版本的软件包。如果您要安装大量软件包，则作业的运行时可能会大大增加。检查您要用来测试软件能否运行的预构建容器的 Python 和 CUDA 版本（如适用）。

当您想在作业脚本中使用非 Python 语言（如 C++ 或 Rust），或者您想使用无法通过 Braket 预建容器提供的 Python 版本时，BYOC 是必需的。在以下情况下，这也是一个不错的选择：

- 您正在使用带有许可证密钥的软件，需要通过许可服务器对该密钥进行身份验证才能运行该软件。使用 BYOC，您可以将许可证密钥嵌入 Docker 映像中，并包含用于对其进行身份验证的代码。
- 您使用的软件不可公开访问。例如，该软件托管在私有 GitHub 存储库 GitLab 或存储库上，您需要使用特定 SSH 密钥才能访问该存储库。
- 您需要安装一套未打包在 Braket 提供的容器中的大型软件。BYOC 可消除由于安装软件而导致混合作业容器启动时间过长。

BYOC 还使用您的软件构建 Docker 容器并将其提供给用户，从而使您的自定义 SDK 或算法可供客户使用。您可以通过在 Amazon ECR 中设置适当的权限来实现此目的。

> **Note**
>
> 您必须遵守所有适用的软件许可证。

自带容器指南

在本节中，我们提供了 Braket Hybrid Jobs 所需内容的 step-by-step指南，包括脚本、文件和将它们组合在一起以启动和运行自定义Docker映像的步骤。bring your own container (BYOC)两种常见案例的诀窍：

1. 在 Docker 映像中安装其他软件，并在作业中仅使用 Python 算法脚本。
2. 使用非 Python 语言编写的算法脚本与 Hybrid Jobs 或 x86 之外的 CPU 架构搭配使用。

对于案例 2，定义容器入口脚本更为复杂。

当 Braket 运行您的混合任务时，它会启动所请求的数量和类型的 Amazon EC2 实例，然后按照映像 URI 中的输入内容运行指定的 Docker 映像，从而在这些实例上创建任务。使用 BYOC 功能时，您可以指定托管在您拥有读取权限的私有 Amazon ECR 存储库中的映像 URI。Braket Hybrid Jobs 使用该自定义映像来运行作业。

构建可用于 Hybrid Jobs 的 Docker 映像所需的特定组件。如果您对编写和构建 `Dockerfiles` 不熟悉，请参阅 Dockerfile 文档和 Amazon ECR CLI 文档。

要求：

- 您的 Dockerfile 的基础映像
- （可选）修改后的容器入口点脚本
- 使用 Dockerfile 安装所需的软件和容器脚本

您的 Dockerfile 的基础映像

如果您使用的是 Python，并且想要在 Braket 提供的容器中提供的内容之上安装软件，那么基础映像的一个选项是托管在我们的GitHub 存储库和 Amazon ECR 上的 Brake t 容器镜像。您需要向 Amazon ECR 进行身份验证才能提取映像并在其上进行构建。例如，您的 BYOC Docker 文件的第一行可能是：`FROM [IMAGE_URI_HERE]`

接下来，填写要安装的 Dockerfile 的其余部分，然后设置要添加到容器中的软件。预先构建的 Braket 映像已经包含了相应的容器入口点脚本，因此您无需担心包含该脚本。

如果您想使用非 Python 语言，如 C++、Rust 或 Julia，或者您想为非 x86 CPU 架构（比如 ARM）构建映像，则可能需要在基本公共映像的基础上构建。您可在 Amazon Elastic Container Registry 公共图库中找到许多这样的图片。请务必选择适合 CPU 架构的 GPU，必要时还要选择要使用的 GPU。

（可选）修改后的容器入口点脚本

> **Note**
>
> 如果您只是在预先构建的 Braket 映像中添加其他软件，则可以跳过本节。

要在混合作业中运行非 Python 代码，请修改定义容器入口点的 Python 脚本。例如，Amazon Braket Githubbraket_container.py 上的 python 脚本。这是 Braket 预先构建的映像用来启动算法脚本和设置相应环境变量的脚本。容器入口点脚本自身必须使用 Python，但可以启动非 Python 脚本。在预先构建的示例中，您可以看到 Python 算法脚本要么作为 Python 子进程启动，要么作为全新的进程启动。通过修改此逻辑，您可以启用入口点脚本来启动非 Python 算法脚本。例如，您可以修改 thekick_off_customer_script() 函数来启动 Rust 进程，具体取决于文件扩展名的结尾。

您还可以选择写入一个全新 braket_container.py。它应将输入数据、源档案和其他必要文件从 Amazon S3 复制到容器中，并定义相应的环境变量。

使用 **Dockerfile** 安装所需的软件和容器脚本

> **Note**
>
> 如果您使用预先构建的 Braket 映像作为 Docker 基础映像，则容器脚本已经存在。

如果您在上一步中创建了修改后的容器脚本，则需要将其复制到容器中，然后将环境变量 SAGEMAKER_PROGRAM 定义为 braket_container.py，或者定义您为新容器入口点脚本命名的变量。

以下是可在 GPU 加速任务实例上使用 Julia 的 Dockerfile 的示例：

```
FROM nvidia/cuda:12.2.0-devel-ubuntu22.04


 ARG DEBIAN_FRONTEND=noninteractive
 ARG JULIA_RELEASE=1.8
 ARG JULIA_VERSION=1.8.3


 ARG PYTHON=python3.11
 ARG PYTHON_PIP=python3-pip
 ARG PIP=pip
```

```
ARG JULIA_URL = https://julialang-s3.julialang.org/bin/linux/x64/${JULIA_RELEASE}/
ARG TAR_NAME = julia-${JULIA_VERSION}-linux-x86_64.tar.gz


ARG PYTHON_PKGS = # list your Python packages and versions here


RUN curl -s -L ${JULIA_URL}/${TAR_NAME} | tar -C /usr/local -x -z --strip-components=1
-f -


RUN apt-get update \

   && apt-get install -y --no-install-recommends \

   build-essential \

   tzdata \

   openssh-client \

   openssh-server \

   ca-certificates \

   curl \

   git \

   libtemplate-perl \

   libssl1.1 \

   openssl \

   unzip \

   wget \

   zlib1g-dev \

   ${PYTHON_PIP} \
```

```
    ${PYTHON}-dev \




 RUN ${PIP} install --no-cache --upgrade ${PYTHON_PKGS}


 RUN ${PIP} install --no-cache --upgrade sagemaker-training==4.1.3


 # Add EFA and SMDDP to LD library path
 ENV LD_LIBRARY_PATH="/opt/conda/lib/python${PYTHON_SHORT_VERSION}/site-packages/
smdistributed/dataparallel/lib:$LD_LIBRARY_PATH"
 ENV LD_LIBRARY_PATH=/opt/amazon/efa/lib/:$LD_LIBRARY_PATH


 # Julia specific installation instructions
 COPY Project.toml /usr/local/share/julia/environments/v${JULIA_RELEASE}/
 RUN JULIA_DEPOT_PATH=/usr/local/share/julia \

    julia -e 'using Pkg; Pkg.instantiate(); Pkg.API.precompile()'
 # generate the device runtime library for all known and supported devices
 RUN JULIA_DEPOT_PATH=/usr/local/share/julia \

    julia -e 'using CUDA; CUDA.precompile_runtime()'


 # Open source compliance scripts
 RUN HOME_DIR=/root \

 && curl -o ${HOME_DIR}/oss_compliance.zip https://aws-dlinfra-
utilities.s3.amazonaws.com/oss_compliance.zip \

 && unzip ${HOME_DIR}/oss_compliance.zip -d ${HOME_DIR}/ \

 && cp ${HOME_DIR}/oss_compliance/test/testOSSCompliance /usr/local/bin/
testOSSCompliance \

 && chmod +x /usr/local/bin/testOSSCompliance \

 && chmod +x ${HOME_DIR}/oss_compliance/generate_oss_compliance.sh \
```

```
 && ${HOME_DIR}/oss_compliance/generate_oss_compliance.sh ${HOME_DIR} ${PYTHON} \

 && rm -rf ${HOME_DIR}/oss_compliance*


 # Copying the container entry point script
 COPY braket_container.py /opt/ml/code/braket_container.py
 ENV SAGEMAKER_PROGRAM braket_container.py
```

此示例下载并运行提供的脚本 AWS ，以确保符合所有相关的开源许可证。例如，通过正确归因任何受 MIT license 控制的已安装代码。

如果您需要包含非公开代码，例如托管在私有代码 GitHub 或 GitLab 存储库中的代码，请不要在 Docker映像中嵌入 SSH 密钥来访问它。相反，请在构建时使用 Docker Compose，以帮助 Docker 在其构建的主机上访问 SSH。有关更多信息，请参阅 Securely using SSH keys in Docker to access private Github repositories 指南。

创建和上传 Docker 映像

有了正确定义的 `Dockerfile`，您现在就可以按照步骤创建私有 Amazon ECR 存储库了（如果尚不存在）。您也可以构建、标记容器图片并将其上传到存储库。

您已准备好构建、标记和推送映像。有关 `docker build` 选项的完整说明和一些示例，请参阅 Docker 构建文档。

对于上面定义的示例文件，您可以运行：

```
aws ecr get-login-password --region ${your_region} | docker login --username AWS --
password-stdin ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com
 docker build -t braket-julia .
 docker tag braket-julia:latest ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com/
braket-julia:latest
 docker push ${aws_account_id}.dkr.ecr.${your_region}.amazonaws.com/braket-julia:latest
```

分配相应的 Amazon ECR 权限

Braket Hybrid Jobs Docker 映像必须托管在私有 Amazon ECR 存储库中。默认情况下，私有 Amazon ECR 存储库不向 Braket Hybrid Jobs IAM role 或任何其他想要使用您的映像的用户（如合作者或学生）提供读取权限。您必须将存储库策略设置为授予相应的权限。通常，仅向您想要访问映像的特定用户和 IAM 角色授予权限，而不是帮助使用 image URI 提取映像。

在自己的容器中运行 Braket 混合作业

要使用自己的容器创建混合作业，请使用指定的参数 `image_uri` 调用 `AwsQuantumJob.create()`。您可以使用 QPU、按需模拟器，也可以在 Braket Hybrid Jobs 提供的经典处理器上对代码进行本地运行。我们建议在真正的 QPU 上运行 TN1 之前 SV1 DM1，在模拟器上测试你的代码，比如、或。

要在经典处理器上运行代码，请通过更新 `InstanceConfig` 来指定您使用的 `instanceType` 和 `instanceCount`。请注意，如果您指定 `instance_count` > 1，则需要确保您的代码可以在多个主机上运行。您可以选择的实例数量上限为 5。例如：

```
job = AwsQuantumJob.create(
    source_module="source_dir",
    entry_point="source_dir.algorithm_script:start_here",
    image_uri="111122223333.dkr.ecr.us-west-2.amazonaws.com/my-byoc-container:latest",
    instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge", instanceCount=3),
    device="local:braket/braket.local.qubit",
    # ...)
```

> **Note**
>
> 使用设备 ARN 跟踪您用作混合作业元数据的模拟器。可接受的值必须遵循格式 `device = "local:<provider>/<simulator_name>"`。请记住 `<provider>` 和 `<simulator_name>` 必须仅包含字母、数字、_、- 和 .。字符串大小限制为 256 个字符。如果您计划使用 BYOC，但不使用 Braket SDK 创建量子任务，则应将环境变量 `AMZN_BRAKET_JOB_TOKEN` 的值传递给 `CreateQuantumTask` 请求中的 `jobToken` 参数。如果您不这样做，量子任务就不会获得优先级，而是作为常规的独立量子任务计费。

查看超参数

创建混合作业时，您可以定义算法所需的超参数，如学习率或步长。超参数值通常用于控制算法的各个方面，并且通常可以对其进行调整以优化算法的性能。要在 Braket 混合作业中使用超参数，您需要将其名称和值明确指定为字典。指定搜索最优值集时要测试的超参数值。使用超参数的第一步是设置超参数并将其定义为字典，这可以在以下代码中看到。

```
from braket.devices import Devices

device_arn = Devices.Amazon.SV1
```

```
hyperparameters = {"shots": 1_000}
```

然后传递上面给出的代码片段中定义的超参数，以便在您选择的算法中使用。要运行以下代码示例，请在与超参数文件相同的路径中创建一个名为“src”的目录。在“src”目录中，添加 0_getting_started_papermill.ipynb、notebook_runner.py 和 requirements.txt 代码文件。

```
import time
from braket.aws import AwsQuantumJob

job = AwsQuantumJob.create(
    device=device_arn,
    source_module="src",
    entry_point="src.notebook_runner:run_notebook",
    input_data="src/0_Getting_started_papermill.ipynb",
    hyperparameters=hyperparameters,
    job_name=f"papermill-job-demo-{int(time.time())}",
)

# Print job to record the ARN
print(job)
```

要从混合作业脚本中访问您的超参数，请参阅 notebook_runner.py python 文件中的 `load_jobs_hyperparams()` 函数。要在混合作业脚本之外访问您的超参数，请运行以下代码。

```
from braket.aws import AwsQuantumJob

# Get the job using the ARN
job_arn = "arn:aws:braket:us-east-1:111122223333:job/5eabb790-d3ff-47cc-98ed-
b4025e9e296f"  # Replace with your job ARN
job = AwsQuantumJob(arn=job_arn)

# Access the hyperparameters
job_metadata = job.metadata()
hyperparameters = job_metadata.get("hyperParameters", {})
print(hyperparameters)
```

有关学习如何使用超参数的更多信息，请参阅 Amazon Braket Hybrid Jobs 教程中的 QAOA 和 Amazon Braket Hybrid Jobs 教程中的 QAOA PennyLane 和 Quantum 机器学习。

配置您的混合作业实例

根据您的算法，您可能有不同的要求。默认情况下，Amazon Braket 会在 `ml.m5.large` 实例上运行您的算法脚本。但是，在创建混合作业时，您可以使用以下导入和配置参数自定义此实例类型。

```
from braket.jobs.config import InstanceConfig

job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge"), # Use NVIDIA T4
 instance with 4 GPUs.
    ...
    ),
```

如果您正在运行嵌入式模拟并在设备配置中指定了本地设备，则还可以通过 `InstanceConfig` 指定 `instanceCount` 并将其设置为大于一台来请求多个实例。上限为 5。例如，您可以按如下方式选择 3 个实例。

```
from braket.jobs.config import InstanceConfig
job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge", instanceCount=3), #
 Use 3 NVIDIA T4 instances
    ...
    ),
```

当您使用多个实例时，请考虑使用数据 parallel 功能分配混合作业。有关如何查看 QML 的 Parallelize 训练示例的更多详细信息，请参阅以下示例 Notebook。

以下三个表格列出了标准、高性能和 GPU 加速实例的可用实例类型及规格。

> **Note**
>
> 要查看混合任务的默认传统计算实例配额，请参阅 Amazon Braket 配额页面。

标准实例	vCPU	内存（GiB）
ml.m5.large（默认）	4	16

标准实例	vCPU	内存（GiB）
ml.m5.xlarge	4	16
ml.m5.2xlarge	8	32
ml.m5.4xlarge	16	64
ml.m5.12xlarge	48	192
ml.m5.24xlarge	96	384

高性能实例	vCPU	内存（GiB）
ml.c5.xlarge	4	8
ml.c5.2xlarge	8	16
ml.c5.4xlarge	16	32
ml.c5.9xlarge	36	72
ml.c5.18xlarge	72	144
ml.c5n.xlarge	4	10.5
ml.c5n.2xlarge	8	21
ml.c5n.4xlarge	16	32
ml.c5n.9xlarge	36	72
ml.c5n.18xlarge	72	192

GPU 加速实例	GPUs	vCPU	内存（GiB）	GPU 内存 (GiB)
ml.p4d.24xlarge	8	96	1152	320

GPU 加速实例	GPUs	vCPU	内存（GiB）	GPU 内存 (GiB)
ml.g4dn.xlarge	1	4	16	16
ml.g4dn.2xlarge	1	8	32	16
ml.g4dn.4xlarge	1	16	64	16
ml.g4dn.8xlarge	1	32	128	16
ml.g4dn.12xlarge	4	48	192	64
ml.g4dn.16xlarge	1	64	256	16

每个实例使用 30 GB 的数据存储（SSD）的默认配置。但是，您可以按照与配置 `instanceType` 相同的方式调整存储。以下示例说明如何将总存储空间增加到 50 GB。

```
from braket.jobs.config import InstanceConfig

job = AwsQuantumJob.create(
    ...
    instance_config=InstanceConfig(
        instanceType="ml.g4dn.xlarge",
        volumeSizeInGb=50,
    ),
    ...
    ),
```

在 **AwsSession** 中配置默认存储桶

使用自己的 `AwsSession` 实例可以提高灵活性，比如能够为默认 Amazon S3 存储桶指定自定义位置。默认情况下，预先配置的 `AwsSession` 的 Amazon S3 存储桶位置为 `"amazon-braket-{id}-{region}"`。但是，在创建 `AwsSession` 时，您可以选择覆盖默认 Amazon S3 的存储桶位置。用户可以选择将 `AwsSession` 对象传递到 `AwsQuantumJob.create()` 方法中，提供 `aws_session` 参数，如以下代码示例所示。

```
aws_session = AwsSession(default_bucket="amazon-braket-s3-demo-bucket")

# Then you can use that AwsSession when creating a hybrid job
job = AwsQuantumJob.create(
```

```
    ...
    aws_session=aws_session
)
```

使用参数化编译加快混合作业的速度

Amazon Braket 在某些方面支持参数化编译。 QPUs这样，您可以仅编译一次电路，而不是为混合算法中的每次迭代编译一次，从而减少与计算成本高昂的编译步骤相关的开销。这样可以明显缩短 Hybrid Jobs 的运行时，因为您无需在每一步都重新编译电路。只需将参数化电路提交给我们支持的 Braket Hybrid QPUs Job 即可。对于长时间运行的混合作业，Braket 在编译电路时会自动使用硬件提供商提供的最新校准数据，以确保获得最高质量的结果。

要创建参数化电路，首先需要在算法脚本中提供参数作为输入。在该例子中，我们使用了一个小型参数化电路，忽略了每次迭代之间的任何经典处理。对于典型的工作负载，您需要批量提交许多电路并执行经典处理，例如在每次迭代中更新参数。

```
import os

from braket.aws import AwsDevice
from braket.circuits import Circuit, FreeParameter

def start_here():

    print("Test job started.")

    # Use the device declared in the job script
    device = AwsDevice(os.environ["AMZN_BRAKET_DEVICE_ARN"])

    circuit = Circuit().rx(0, FreeParameter("theta"))
    parameter_list = [0.1, 0.2, 0.3]

    for parameter in parameter_list:
        result = device.run(circuit, shots=1000, inputs={"theta": parameter})

    print("Test job completed.")
```

您可以使用以下作业脚本提交算法脚本以混合作业形式运行。在支持参数化编译的 QPU 上运行 Hybrid Jobs 时，仅在第一次运行时才编译电路。在接下来的运行中，重复使用编译后的电路，无需任何额外的代码行即可提高 Hybrid Jobs 的运行时性能。

```
from braket.aws import AwsQuantumJob
```

```
job = AwsQuantumJob.create(
    device=device_arn,
    source_module="algorithm_script.py",
)
```

> **Note**
>
> 除脉冲电平程序外，所有基于门的 QPUs 超导模式都支持参数化编译。Rigetti Computing

PennyLane 与 Amazon Braket 一起使用

混合算法是同时包含经典指令和量子指令的算法。经典指令在经典硬件（EC2 实例或 Notebook）上运行，量子指令要么在模拟器上运行，要么在量子计算机上运行。我们建议您使用混合作业功能运行混合算法。有关更多信息，请参阅何时使用 Amazon Braket 任务。

Amazon Braket 使您能够在 Amazon Braket 插件的帮助下，或者使用 Amazon PennyLane Braket Python SDK 和示例笔记本存储库来设置和运行混合量子算法。基于软件开发工具包的 Amazon Braket 示例笔记本使您无需插件即可设置和运行某些混合算法。 PennyLane 但是，我们 PennyLane 之所以推荐，是因为它提供了更丰富的体验。

关于混合量子算法

混合量子算法对当今的行业很重要，因为当代量子计算设备通常会产生噪声，从而产生错误。计算中添加的每个量子门都会增加噪声增大的机会；因此，长期运行的算法可能会被噪声所淹没，从而导致计算错误。

诸如肖尔的（量子相位估计示例）或格罗弗的（格罗弗的例子）之类的纯量子算法需要数千或数百万次运算。出于这个原因，它们对于现有的量子设备来说可能不切实际，这些设备被称为噪声中级量子（NISQ）设备。

在混合量子算法中，量子处理单元 (QPUs) 充当经典算法的协处理器 CPUs，专门用于加快经典算法中的某些计算。电路执行时间大大缩短，触手可及。

本节内容：

- 带有 Amazon Braket PennyLane
- Amazon Braket 中的混合算法示例 Notebook
- 带有嵌入式 PennyLane 仿真器的混合算法

- PennyLane 使用 Amazon Braket 模拟器开启伴随渐变
- 使用混合作业和 PennyLane 运行 QAOA 算法
- 使用 PennyLane 嵌入式仿真器运行混合工作负载

带有 Amazon Braket PennyLane

Amazon Braket 为PennyLane围绕量子微分编程概念构建的开源软件框架提供支持。您可以使用该框架来训练量子电路，就像训练神经网络来寻找量子化学、量子机器学习和优化中的计算问题的解决方案一样。

该 PennyLane 库为熟悉的机器学习工具（包括 PyTorch 和）提供了接口 TensorFlow，使量子电路训练变得快速而直观。

- PennyLane 库 —— PennyLane 已预先安装在 Amazon Braket 笔记本电脑中。要从中访问 Amazon Braket 设备 PennyLane，请打开笔记本并使用以下命令导入 PennyLane 库。

```
import pennylane as qml
```

教程 Notebook 可帮助您快速入门。或者，您可以通过自己选择的 IDE PennyLane 在 Amazon Braket 上使用。

- AmazonBraket PennyLane 插件 — 要使用你自己的 IDE，你可以手动安装 Amazon Braket PennyLane 插件。该插件 PennyLane 与 Amazon Braket Python SDK 连接，因此你可以在 Amazon Braket 设备 PennyLane 上运行电路。要安装该 PennyLane 插件，请使用以下命令。

```
pip install amazon-braket-pennylane-plugin
```

以下示例演示了如何在中 PennyLane设置对 Amazon Braket 设备的访问权限：

```
# to use SV1
import pennylane as qml
sv1 = qml.device("braket.aws.qubit", device_arn="arn:aws:braket:::device/quantum-
simulator/amazon/sv1", wires=2)

# to run a circuit:
@qml.qnode(sv1)
def circuit(x):
```

```
    qml.RZ(x, wires=0)
    qml.CNOT(wires=[0,1])
    qml.RY(x, wires=1)
    return qml.expval(qml.PauliZ(1))

result = circuit(0.543)


#To use the local sim:
local = qml.device("braket.local.qubit", wires=2)
```

有关教程示例和更多信息 PennyLane，请参阅 Amazon Braket 示例存储库。

AmazonBraket PennyLane 插件使您只需一行代码即可在 Amazon PennyLane Braket QPU 和嵌入式仿真器设备之间切换。它提供了两个 Amazon Braket 量子器件可供使用 PennyLane：

- `braket.aws.qubit`用于与 Amazon Braket 服务的量子设备一起运行，包括 QPUs 和模拟器
- `braket.local.qubit` 用于使用 Amazon Braket SDK 的本地模拟器运行

AmazonBraket PennyLane 插件是开源的。你可以从PennyLane 插件 GitHub 存储库中安装它。

有关的更多信息 PennyLane，请参阅PennyLane 网站上的文档。

Amazon Braket 中的混合算法示例 Notebook

Amazon Braket 确实提供了各种不依赖 PennyLane 插件来运行混合算法的示例笔记本。您可以开始使用这些说明变分方法的 Amazon Braket 混合示例 Notebook 中的任何一个，如量子近似优化算法 (QAOA) 或变分量子特征求解器 (VQE)。

Amazon Braket 示例 Notebook 依赖于 Amazon Braket Python SDK。SDK 提供了一个通过 Amazon Braket 与量子计算硬件设备进行交互的框架。它是一个开源库，旨在帮助您完成混合工作流程的量子部分。

您可以使用我们的示例 Notebook 进一步探索 Amazon Braket。

带有嵌入式 PennyLane 仿真器的混合算法

Amazon Braket Hybrid Jobs 现在配备了基于 CPU 和 GPU 的高性能嵌入式模拟器。PennyLane该系列嵌入式模拟器可以直接嵌入到您的混合作业容器中，包括快速状态向量 `lightning.qubit` 模拟器、使用 NVIDIA 的 cuQuantum 库加速的 `lightning.gpu` 模拟器等。这些嵌入式模拟器非常适合变

分算法，如量子机器学习，这些算法可以从高级方法（如伴随微分法）中受益。您可以在一个或多个 CPU 或 GPU 实例上运行这些嵌入式模拟器。

借助 Hybrid Jobs，您现在可以使用经典协处理器和 QPU 的组合、Amazon Braket 按需模拟器（例如）或直接使用中的嵌入式仿真器来运行变分算法代码。SV1 PennyLane

嵌入式模拟器已经在 Hybrid Jobs 容器中可用，您需要用 @hybrid_job 装饰器来装饰您的主 Python 函数。要使用 PennyLane `lightning.gpu`模拟器，您还需要在中指定 GPU 实例，`InstanceConfig`如以下代码片段所示：

```
import pennylane as qml
from braket.jobs import hybrid_job
from braket.jobs.config import InstanceConfig


@hybrid_job(device="local:pennylane/lightning.gpu",
 instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge"))
def function(wires):
    dev = qml.device("lightning.gpu", wires=wires)
    ...
```

要开始使用带有 Hybrid Jobs 的 PennyLane 嵌入式模拟器，请参阅示例笔记本。

PennyLane 使用 Amazon Braket 模拟器开启伴随渐变

借助 Amazon Braket 的PennyLane插件，您可以在本地状态向量模拟器上运行时使用伴随微分法计算梯度，或者。 SV1

注意：要使用伴随微分法，必须在您的 qnode 而非 `diff_method='adjoint'` 中指定 `diff_method='device'`。请参阅以下示例。

```
device_arn = "arn:aws:braket:::device/quantum-simulator/amazon/sv1"
dev = qml.device("braket.aws.qubit", wires=wires, shots=0, device_arn=device_arn)

@qml.qnode(dev, diff_method="device")
def cost_function(params):
    circuit(params)
    return qml.expval(cost_h)

gradient = qml.grad(circuit)
initial_gradient = gradient(params0)
```

> **Note**
>
> 目前，PennyLane 将计算 QAOA 哈密顿量的分组指数，并使用它们将哈密顿函数拆分为多个期望值。如果要在从中运行 QAOA 时使用 SV1伴随微分能力 PennyLane，则需要通过移除分组指数来重建成本哈密顿模型，如下所示：`cost_h, mixer_h = qml.qaoa.max_clique(g, constrained=False) cost_h = qml.Hamiltonian(cost_h.coeffs, cost_h.ops)`

使用混合作业和 PennyLane 运行 QAOA 算法

在本节中，您将使用所学知识与参数化编译 PennyLane 一起编写实际的混合程序。您可以使用算法脚本来解决量子近似优化算法（QAOA）问题。该程序创建了一个与经典的 Max Cut 优化问题相对应的成本函数，指定了参数化量子电路，并使用梯度下降法来优化参数，从而使成本函数达到最小化。在此示例中，为简单起见，我们在算法脚本中生成问题图，但对于更典型的使用案例，最佳做法是通过输入数据配置中的专用通道提供问题规范。该标志`parametrize_differentiable`默认为，`True`因此您可以自动从支持的 QPUs参数化编译中获得提高运行时性能的好处。

```
import os
import json
import time

from braket.jobs import save_job_result
from braket.jobs.metrics import log_metric

import networkx as nx
import pennylane as qml
from pennylane import numpy as np
from matplotlib import pyplot as plt

def init_pl_device(device_arn, num_nodes, shots, max_parallel):
    return qml.device(
        "braket.aws.qubit",
        device_arn=device_arn,
        wires=num_nodes,
        shots=shots,
        # Set s3_destination_folder=None to output task results to a default folder
        s3_destination_folder=None,
        parallel=True,
        max_parallel=max_parallel,
        parametrize_differentiable=True, # This flag is True by default.
```

```
    )

def start_here():
    input_dir = os.environ["AMZN_BRAKET_INPUT_DIR"]
    output_dir = os.environ["AMZN_BRAKET_JOB_RESULTS_DIR"]
    job_name = os.environ["AMZN_BRAKET_JOB_NAME"]
    checkpoint_dir = os.environ["AMZN_BRAKET_CHECKPOINT_DIR"]
    hp_file = os.environ["AMZN_BRAKET_HP_FILE"]
    device_arn = os.environ["AMZN_BRAKET_DEVICE_ARN"]

    # Read the hyperparameters
    with open(hp_file, "r") as f:
        hyperparams = json.load(f)

    p = int(hyperparams["p"])
    seed = int(hyperparams["seed"])
    max_parallel = int(hyperparams["max_parallel"])
    num_iterations = int(hyperparams["num_iterations"])
    stepsize = float(hyperparams["stepsize"])
    shots = int(hyperparams["shots"])

    # Generate random graph
    num_nodes = 6
    num_edges = 8
    graph_seed = 1967
    g = nx.gnm_random_graph(num_nodes, num_edges, seed=graph_seed)

    # Output figure to file
    positions = nx.spring_layout(g, seed=seed)
    nx.draw(g, with_labels=True, pos=positions, node_size=600)
    plt.savefig(f"{output_dir}/graph.png")

    # Set up the QAOA problem
    cost_h, mixer_h = qml.qaoa.maxcut(g)

    def qaoa_layer(gamma, alpha):
        qml.qaoa.cost_layer(gamma, cost_h)
        qml.qaoa.mixer_layer(alpha, mixer_h)

    def circuit(params, **kwargs):
        for i in range(num_nodes):
            qml.Hadamard(wires=i)
        qml.layer(qaoa_layer, p, params[0], params[1])
```

```
    dev = init_pl_device(device_arn, num_nodes, shots, max_parallel)

    np.random.seed(seed)
    cost_function = qml.ExpvalCost(circuit, cost_h, dev, optimize=True)
    params = 0.01 * np.random.uniform(size=[2, p])

    optimizer = qml.GradientDescentOptimizer(stepsize=stepsize)
    print("Optimization start")

    for iteration in range(num_iterations):
        t0 = time.time()

        # Evaluates the cost, then does a gradient step to new params
        params, cost_before = optimizer.step_and_cost(cost_function, params)
        # Convert cost_before to a float so it's easier to handle
        cost_before = float(cost_before)

        t1 = time.time()

        if iteration == 0:
            print("Initial cost:", cost_before)
        else:
            print(f"Cost at step {iteration}:", cost_before)

        # Log the current loss as a metric
        log_metric(
            metric_name="Cost",
            value=cost_before,
            iteration_number=iteration,
        )

        print(f"Completed iteration {iteration + 1}")
        print(f"Time to complete iteration: {t1 - t0} seconds")

    final_cost = float(cost_function(params))
    log_metric(
        metric_name="Cost",
        value=final_cost,
        iteration_number=num_iterations,
    )

    # We're done with the hybrid job, so save the result.
    # This will be returned in job.result()
```

```
    save_job_result({"params": params.numpy().tolist(), "cost": final_cost})
```

> **Note**
>
> 除脉冲电平程序外，所有基于门的 QPUs 超导模式都支持参数化编译。Rigetti Computing

使用 PennyLane 嵌入式仿真器运行混合工作负载

让我们来看看如何使用 Amazon Braket Hybrid Jobs PennyLane 上的嵌入式模拟器来运行混合工作负载。Pennylane 基于 GPU 的嵌入式模拟器 `lightning.gpu` 使用 Nvidia cuQuantum 库来加快电路模拟速度。嵌入式 GPU 模拟器已在所有 Braket 作业容器中进行了预配置，用户可以开箱即用。在本页中，我们将介绍如何使用 `lightning.gpu` 来加速混合工作负载。

使用适用于 QAOA 工作负载的 `lightning.gpu`

考虑本 Notebook 中的量子近似优化算法（QAOA）示例。要选择嵌入式模拟器，请将 `device` 参数指定为以下形式的字符串：`"local:<provider>/<simulator_name>"`。例如，您可以设置 `lightning.gpu` 的 `"local:pennylane/lightning.gpu"`。启动时提供给 Hybrid Jobs 的设备字符串将作为环境变量 `"AMZN_BRAKET_DEVICE_ARN"` 传递给该作业。

```
device_string = os.environ["AMZN_BRAKET_DEVICE_ARN"]
prefix, device_name = device_string.split("/")
device = qml.device(simulator_name, wires=n_wires)
```

在本页中，比较两个嵌入式 PennyLane 状态矢量模拟器`lightning.qubit`（基于 CPU）和`lightning.gpu`（基于 GPU）。为模拟器提供自定义门分解以计算各种梯度。

现在，您已准备好混合作业启动脚本。使用两种实例类型运行 QAOA 算法：`ml.m5.2xlarge` 和 `ml.g4dn.xlarge`。`ml.m5.2xlarge` 实例类型相当于标准的开发人员 Notebook。`ml.g4dn.xlarge`是一个加速计算实例，它有一个 NVIDIA T4 GPU 和 16GB 内存。

要运行 GPU，我们首先需要指定兼容的映像和正确的实例（默认为`ml.m5.2xlarge`实例）。

```
from braket.aws import AwsSession
from braket.jobs.image_uris import Framework, retrieve_image

image_uri = retrieve_image(Framework.PL_PYTORCH, AwsSession().region)
instance_config = InstanceConfig(instanceType="ml.g4dn.xlarge")
```

然后，我们需要将这些参数以及系统和混合作业参数中更新的设备参数输入到混合作业装饰器。

```
@hybrid_job(
        device="local:pennylane/lightning.gpu",
        input_data=input_file_path,
        image_uri=image_uri,
        instance_config=instance_config)
def run_qaoa_hybrid_job_gpu(p=1, steps=10):
    params = np.random.rand(2, p)

    braket_task_tracker = Tracker()

    graph = nx.read_adjlist(input_file_path, nodetype=int)
    wires = list(graph.nodes)
    cost_h, _mixer_h = qaoa.maxcut(graph)

    device_string = os.environ["AMZN_BRAKET_DEVICE_ARN"]
    prefix, device_name = device_string.split("/")
    dev= qml.device(simulator_name, wires=len(wires))
    ...
```

> **Note**
>
> 如果您将指定`instance_config`为使用基于 GPU 的实例，但选择作为基于 CPU 的嵌入式模拟器 (`lightning.qubit`)，则不会使用 GPU。`device`如果您想瞄准 GPU，一定要使用基于 GPU 的嵌入式模拟器！

`m5.2xlarge`实例的平均迭代时间约为 73 秒，而`ml.g4dn.xlarge`实例的平均迭代时间约为 0.6 秒。对于这个 21 量子比特的工作流程，GPU 实例为我们提供了 100 倍的加速。如果你查看 Amazon Braket Hybrid Jobs 定价页面，你可以看到实例的每分钟费用为 0.00768 美元，而`m5.2xlarge`实例的每分钟费用为 0.01227 `ml.g4dn.xlarge` 美元。在这种情况下，在 GPU 实例上运行更快、更便宜。

量子机器学习和数据并行性

如果您的工作负载类型是基于数据集训练的量子机器学习（QML），则可以使用数据并行性进一步加快工作负载。在 QML 中，模型包含一个或多个量子电路。该模型可能还包含或者不包含经典神经网络。使用数据集训练模型时，会更新模型中的参数以最小化损失函数。损失函数通常是针对单个数据点以及整个数据集的平均损失的总损失定义的。在 QML 中，通常先串行计算损耗，然后再求平均为梯度计算的总损耗。此过程非常耗时，尤其是在有数百个数据点的情况下。

由于一个数据点的损失不依赖于其他数据点，因此可以并行评估损失！可以同时评估与不同数据点相关的损失和梯度。这就是所谓的数据并行性。借助 SageMaker分布式数据并行库，Amazon Braket Hybrid Jobs 可让您更轻松地使用数据并行性来加速训练。

考虑以下 QML 数据并行工作负载，该工作负载使用知名 UCI 存储库中的声纳数据集作为二元分类的示例。该声纳数据集有 208 个数据点，每个数据点有 60 个特征，这些特征是从材料上反弹的声纳信号中收集的。每个数据点要么被标记为“M”（代表地雷），要么标记为“R”（表示岩石）。我们的 QML 模型由输入层、作为隐藏层的量子电路和输出层组成。输入层和输出层是中实现的经典神经网络 PyTorch。量子电路使用的 qml.q PennyLane nn 模块与 PyTorch 神经网络集成。有关工作负载的更多详细信息，请参阅我们的示例 Notebook。就像上面的 QAOA 示例一样，你可以利用基于 GPU 的嵌入式模拟器（比如）`lightning.gpu`来提高基于 CPU PennyLane 的嵌入式模拟器的性能。

要创建混合作业，您可以通过其关键字参数调用 `AwsQuantumJob.create` 和指定算法脚本、设备和其他配置。

```
instance_config = InstanceConfig(instanceType='ml.g4dn.xlarge')

hyperparameters={"nwires": "10",
                 "ndata": "32",
                 ...
}

job = AwsQuantumJob.create(
    device="local:pennylane/lightning.gpu",
    source_module="qml_source",
    entry_point="qml_source.train_single",
    hyperparameters=hyperparameters,
    instance_config=instance_config,
    ...
)
```

要使用数据并行性，您需要修改 SageMaker 分布式库算法脚本中的几行代码，以正确并行化训练。首先，您导入`smdistributed`软件包，该软件包负责在多个 GPUs 和多个实例之间分配工作负载，从而完成大部分繁重的工作。此软件包已在 Braket PyTorch 和 TensorFlow容器中预先配置。该`dist`模块告诉我们的算法脚本训练 GPUs 的总数 (`world_size`) 以及 GPU 内核`local_rank`的`rank`和。`rank`是 GPU 在所有实例中的绝对索引，而`local_rank`是 GPU 在实例中的索引。例如，如果有四个实例，每个实例 GPUs 分配了八个用于训练，则`rank`范围为 0 到 31，`local_rank`范围为 0 到 7。

```
import smdistributed.dataparallel.torch.distributed as dist
```

```
dp_info = {
    "world_size": dist.get_world_size(),
    "rank": dist.get_rank(),
    "local_rank": dist.get_local_rank(),
}
batch_size //= dp_info["world_size"] // 8
batch_size = max(batch_size, 1)
```

接下来，您根据 world_size 和 rank 定义一个 DistributedSampler，然后将其传递到数据加载器中。此采样器可避免 GPUs 访问数据集的同一片段。

```
train_sampler = torch.utils.data.distributed.DistributedSampler(
    train_dataset,
    num_replicas=dp_info["world_size"],
    rank=dp_info["rank"]
)
train_loader = torch.utils.data.DataLoader(
    train_dataset,
    batch_size=batch_size,
    shuffle=False,
    num_workers=0,
    pin_memory=True,
    sampler=train_sampler,
)
```

接下来，使用 DistributedDataParallel 类来启用数据并行性。

```
from smdistributed.dataparallel.torch.parallel.distributed import
 DistributedDataParallel as DDP

model = DressedQNN(qc_dev).to(device)
model = DDP(model)
torch.cuda.set_device(dp_info["local_rank"])
model.cuda(dp_info["local_rank"])
```

以上是使用数据并行性所需的更改。在 QML 中，您通常希望保存结果并打印训练进度。如果每个 GPU 都运行保存和打印命令，则日志中将会充斥重复信息，结果将相互覆盖。为避免这种情况发生，您只能使用具有 rank 0 的 GPU 进行保存和打印。

```
if dp_info["rank"]==0:
    print('elapsed time: ', elapsed)
    torch.save(model.state_dict(), f"{output_dir}/test_local.pt")
```

```
    save_job_result({"last loss": loss_before})
```

Amazon Braket 混合任务支持 SageMaker 分布式数据并行库的ml.g4dn.12xlarge实例类型。您可以通过 Hybrid Jobs InstanceConfig 中的参数配置实例类型。要使 SageMaker 分布式数据并行库知道数据并行性已启用，您需要再添加两个超参数，即"sagemaker_distributed_dataparallel_enabled"设置为正在使用的实例类型"true"和"sagemaker_instance_type"设置。这两个超参数由 smdistributed 软件包使用。您的算法脚本无需明确使用它们。在 Amazon Braket SDK 中，它提供了一个方便的关键字参数 distribution。有了混合任务创建中的 distribution="data_parallel"，Amazon Braket SDK 会自动为您插入两个超参数。如果您使用 Amazon Braket API，则需要包含这两个超参数。

配置好实例和数据并行度后，您现在可以提交混合作业了。一个ml.g4dn.12xlarge实例 GPUs 中有 4 个。设置后instanceCount=1，工作负载将分布在实例 GPUs 中的 8 个中。当您设置instanceCount大于 1 时，工作负载将分布在所有 GPUs 可用实例中。使用多个实例时，每个实例会根据您的使用时间产生费用。例如，当您使用四个实例时，计费时间是每个实例运行时的四倍，因为有四个实例同时运行您的工作负载。

```
instance_config = InstanceConfig(instanceType='ml.g4dn.12xlarge',
                                 instanceCount=1,
)

hyperparameters={"nwires": "10",
                 "ndata": "32",
                 ...,
}

job = AwsQuantumJob.create(
    device="local:pennylane/lightning.gpu",
    source_module="qml_source",
    entry_point="qml_source.train_dp",
    hyperparameters=hyperparameters,
    instance_config=instance_config,
    distribution="data_parallel",
    ...
)
```

> **Note**
>
> 在上面的混合作业创建中，train_dp.py 是修改后用于使用数据并行性的算法脚本。请记住，只有当您根据上述部分修改算法脚本时，数据并行性才能正常工作。如果在未正确修改算

法脚本的情况下启用数据并行度选项，则混合作业可能会引发错误，或者每个 GPU 可能会重复处理相同的数据切片，从而造成效率低下。

如果使用得当，使用多个实例可以使时间和成本减少几个数量级。有关更多详细信息，请参阅示例笔记本。

在 Amazon Braket 上使用 CUDA-Q

NVIDIA's CUDA-Q是一个软件库，专为编程组合了 CPUs GPUs、和量子处理单元 (QPUs) 的混合量子算法而设计。它提供了统一的编程模型，允许开发人员在单个程序中表达经典指令和量子指令，从而简化工作流程。 CUDA-Q利用其内置 CPU 和 GPU 模拟器加速量子程序仿真和运行时间。 CUDA-Q适用于原生 Braket 笔记本实例 (NBIs) 和 Amazon Braket 混合任务。

本节内容：

- CUDA-Q in NBIs
- 混合工作中的 CUDA-Q

CUDA-Q in NBIs

CUDA-Q 默认安装在 Braket NBI 环境中。您可以通过前往 Jupyter 启动器页面并选择 CUDA-Q 和 Braket 图块来打开 CUDA-Q 示例 Notebook。这将在主窗口 `0_Getting_started_with_CUDA-Q.ipynb` 中打开示例 Notebook。有关更多 CUDA-Q 示例，请参阅 `nvidia_cuda_q/` 目录中的左侧面板。

您还可以验证 NBI 中安装的 CUDA-Q 或任何其他第三方软件包的版本。例如，您可以在笔记本代码单元中运行以下命令来验证环境中安装的CUDA-Q、Qiskit 和 Braket 软件包的版本。 PennyLane

```
%pip freeze | grep -i -e cudaq -e qiskit -e pennylane -e braket
```

混合工作中的 CUDA-Q

在 Amazon Braket Hybrid Jobs 上使用 CUDA-Q 可提供灵活的按需计算环境。计算实例仅在您的工作负载持续时间内运行，确保您只需按实际使用量付费。Amazon Braket Hybrid Jobs 还提供了可扩展的体验。用户可以从较小的实例开始进行原型设计和测试，然后纵向扩展到能够处理更多工作负载以进行完整实验的大型实例。

Amazon Braket 混合工作支持 GPUs 对于最大限度地发挥潜CUDA-Q力至关重要。 GPUs 与基于 CPU 的模拟器相比，可显著加快量子程序仿真速度，尤其是在使用高量子比特数电路时。在 Amazon Braket Hybrid Jobs 上使用 CUDA-Q 时，并行化变得简单明了。Hybrid Jobs 简化了电路采样和可观察评估在多个计算节点上的分布。借助这种 CUDA-Q 工作负载的无缝并行化，用户能够将更多精力放在开发工作负载上，而不是为大规模实验设置基础架构。

要开始使用 Braket 提供的 CUDA-Q 混合作业容器，请参阅 Github 上的 Amazon Braket 示例 Github 上的 CUDA-Q 入门示例。

以下代码片段是使用 Amazon Braket Hybrid Jobs 运行 CUDA-Q 程序的 `hello-world` 示例。

```
image_uri = retrieve_image(Framework.CUDAQ, AwsSession().region)

@hybrid_job(device='local:nvidia/qpp-cpu', image_uri=image_uri)
def hello_quantum():
    import cudaq

    # define the backend
    device=get_job_device_arn()
    cudaq.set_target(device.split('/')[-1])

    # define the Bell circuit
    kernel = cudaq.make_kernel()
    qubits = kernel.qalloc(2)
    kernel.h(qubits[0])
    kernel.cx(qubits[0], qubits[1])

    # sample the Bell circuit
    result = cudaq.sample(kernel, shots_count=1000)
    measurement_probabilities = dict(result.items())

    return measurement_probabilities
```

上面的示例模拟了 CPU 模拟器上的贝尔电路。此示例在您的 Notebook 或 Braket Jupyter Notebook 上本地运行。由于 `local=True` 设置的原因，当您运行此脚本时，将在您的本地环境中启动一个容器来运行 CUDA-Q 程序以进行测试和调试。完成测试后，您可以移除 `local=True` 标志并继续在 AWS 上运行作业。要了解更多信息，请参阅使用 Amazon Braket Hybrid Jobs。

如果您的工作负载具有较高的量子比特数、大量的电路或大量的迭代，则可以通过指定 `instance_config` 设置来使用更强大的 CPU 计算资源。以下代码段演示如何在 `hybrid_job` 装饰

器中配置 instance_config 设置。有关所支持实例类型的更多信息，请参阅实例类型。有关实例类型的列表，请参阅 Amazon EC2 实例类型。

```
@hybrid_job(
    device="local:nvidia/qpp-cpu",
    image_uri=image_uri,
    instance_config=InstanceConfig(instanceType="ml.c5.2xlarge"),
)
def my_job_script():
    ...
```

对于要求更高的工作负载，您可以在 CUDA-Q GPU 模拟器上运行工作负载。要启用 GPU 模拟器，请使用后端名称 nvidia。nvidia 后端作为 CUDA-Q GPU 模拟器运行。接下来，选择支持 NVIDIA GPU 的 Amazon EC2 实例类型。以下代码片段显示了 GPU 配置的 hybrid_job 装饰器。

```
@hybrid_job(
    device="local:nvidia/nvidia",
    image_uri=image_uri,
    instance_config=InstanceConfig(instanceType="ml.g4dn.xlarge"),
)
def my_job_script():
    ...
```

Amazon Braket Hybrid Jobs NBIs 并支持并行 GPU 模拟。CUDA-Q您可以并行计算多个可观测值或多个电路，以提升工作负载的性能。要并行化多个可观测值，请对算法脚本做出以下更改。

设置 nvidia 后端的 mgpu 选项。这是并行化可观测值所必需的。并行化使用 MPI 进行相互通信 GPUs，因此 MPI 需要在执行前初始化，并在执行之后完成。

接下来，通过设置 execution=cudaq.parallel.mpi 指定执行模式。以下代码片段显示了这些更改。

```
cudaq.set_target("nvidia", option="mqpu")
cudaq.mpi.initialize()
result = cudaq.observe(
    kernel, hamiltonian, shots_count=n_shots, execution=cudaq.parallel.mpi
)
cudaq.mpi.finalize()
```

在hybrid_job装饰器中指定托管多个实例类型， GPUs 如以下代码片段所示。

```
@hybrid_job(
    device="local:nvidia/nvidia-mqpu",
    instance_config=InstanceConfig(instanceType="ml.g4dn.12xlarge", instanceCount=1),
    image_uri=image_uri,
)
def parallel_observables_gpu_job(sagemaker_mpi_enabled=True):
    ...
```

Github 上的 Amazon Braket 示例中的并行仿真笔记本 end-to-end提供了演示如何在 GPU 后端运行量子程序仿真以及如何对可观测物和电路批量进行并行仿真的示例。

在量子计算机上运行工作负载

完成模拟器测试后，您可以过渡到在上运行实验 QPUs。只需将目标切换到 Amazon Braket QPU，如 IQM、IonQ 或 Rigetti 设备即可。以下代码片段说明了如何将目标设置为 IQM Garnet 设备。有关可用列表 QPUs，请参阅 Amazon Braket 控制台。

```
device_arn = "arn:aws:braket:eu-north-1::device/qpu/iqm/Garnet"
cudaq.set_target("braket", machine=device_arn)
```

有关混合任务的更多信息，请参阅开发者指南中的使用 Amazon Braket Hybrid Jobs。要了解有关 CUDA-Q 的更多信息，请参阅 NVIDIA CUDA-Q 文档。

Amazon Braket 故障排除

使用本节中的故障排除信息和解决方案来帮助解决 Amazon Braket 的问题。

本节内容：

AccessDeniedException

如果您AccessDeniedException在启用或使用 Braket 时收到 Braket，则您可能正在尝试在您的受限角色无权访问的区域启用或使用 Braket。

在这种情况下，请联系您的内部 AWS 管理员，了解以下哪些条件适用：

- 是否存在角色限制，导致无法访问某个区域。
- 是否允许您尝试使用的角色使用 Braket。

如果您的角色在使用 Braket 时无法访问给定区域，则您将无法使用该特定区域的设备。

调用 CreateQuantumTask 操作时出错 (ValidationException)

如果您收到类似于 `An error occurred (ValidationException) when calling the CreateQuantumTask operation: Caller doesn't have access to amazon-braket-…` 的错误，请检查您是否引用了现有的 s3_folder。Braket 不会自动为您创建新的 Amazon S3 存储桶和前缀。

如果您直接访问 API 并收到类似于 `Failed to create quantum task: Caller doesn't have access to s3://MY_BUCKET` 的错误，请检查您是否未将 `s3://` 包含在 Amazon S3 存储桶路径中。

某个 SDK 功能无法使用

你的 Python 版本必须是 3.10 或更高版本。对于 Amazon Braket 混合任务，我们推荐 Python 3.12。

验证您的 SDK 和架构是否正确。 up-to-date要从 Notebook 或 Python 编辑器更新 SDK，请运行以下命令：

```
pip install amazon-braket-sdk --upgrade --upgrade-strategy eager
```

要更新架构，请运行以下命令。

```
pip install amazon-braket-schemas --upgrade
```

如果您通过自己的客户访问 Amazon Braket，请确认您的AWS 区域是否已设置为 Amazon Braket 支持的区域。

由于以下原因，混合作业失败 ServiceQuotaExceededException

如果您超出目标模拟器设备的并发量子任务限制，则可能无法创建针对 Amazon Braket 模拟器运行量子任务的混合作业。有关服务限制的更多信息，请参阅配额主题。

如果您在账户中的多个混合作业中对模拟器设备运行并发任务，则可能会遇到此错误。

要查看针对特定模拟器设备的并发量子任务数，请使用 `search-quantum-tasks` API，如以下代码示例所示。

```
DEVICE_ARN=arn:aws:braket:::device/quantum-simulator/amazon/sv1
task_list=""
for status_value in "CREATED" "QUEUED" "RUNNING" "CANCELLING"; do
    tasks=$(aws braket search-quantum-tasks --filters
 name=status,operator=EQUAL,values=${status_value}
 name=deviceArn,operator=EQUAL,values=$DEVICE_ARN --max-results 100 --query
 'quantumTasks[*].quantumTaskArn' --output text)
    task_list="$task_list $tasks"
done;
echo "$task_list" | tr -s ' \t' '[\n*]' | sort | uniq
```

您还可以使用亚马逊 CloudWatch 指标查看针对设备创建的量子任务：Braket > B y Device。

为避免遇到这些错误，请执行以下操作：

1. 请求增加模拟器设备并发量子任务数量的服务限额。这仅适用于 SV1 设备。
2. 处理代码中的 `ServiceQuotaExceeded` 异常并重试。

组件在 Notebook 实例中停止工作

如果 Notebook 的某些组件停止工作，请尝试以下操作。

1. 将您创建或修改的所有 Notebook 下载到本地驱动器。
2. 停用您的 Notebook 实例。
3. 删除您的 Notebook 实例。
4. 使用其他名称创建新的 Notebook 实例。
5. 将 Notebook 上传到新实例。

Python 3.12 升级疑难解答

生效日期：2026年1月21日

概述

自 2026 年 1 月 21 日起，Amazon Braket 将所有笔记本实例和托管容器镜像（Base、CU TensorFlow DA-Q 和 PyTorch）的 Python 运行时从 3.10 升级到 3.12。本指南提供了常见兼容性问题的解决方案。

本节内容：

- 常见错误消息
- Braket 托管笔记本电脑
- Hybrid Job 装饰器
- Bring-Your-Own-Container (BYOC)
- Braket 笔记本实例升级

常见错误消息

SDK Python 版本不匹配错误

错误：

```
RuntimeError: Python version must match between local environment and container. Client
 is running Python 3.10 locally, but container uses Python 3.12.
```

原因：Braket SDK 检测到你的笔记本正在运行 Python 3.10，但是 Hybrid Job 容器正在运行 Python 3.12。

解决方案：要么将你的笔记本升级到 Python 3.12，要么固定到 Python 3.10 容器。

Cloudpickle 序列化错误

错误：

```
TypeError: code() argument 13 must be str, not int
```

原因：如果绕过 SDK 验证，则由于 Python 3.12 中的构造函数 CodeType 发生了变化，cloudpickle 将无法序列化 Python 3.10 和 3.12 之间的代码。

解决方案：确保您的笔记本和容器使用相同的 Python 版本。

Braket 托管笔记本电脑

如果你在 Python 3.10 上运行 Braket Notebook 实例并提交混合作业，则会遇到版本不匹配错误，因为任务容器现在默认使用 Python 3.12。

你有两个选择：

1. [推荐] 使用 Python 3.12 创建新的笔记本实例——参见 Braket 笔记本实例升级
2. 固定到 Python 3.10 容器——参见 Hy brid J ob Decorator

Hybrid Job 装饰器

要使用`@hybrid_job`装饰器，您的环境的 Python 版本必须与容器的 Python 版本相匹配。

选项 1：使用 Python 3.12 容器（推荐）

如果您已将环境升级到 Python 3.12，它将使用最新标签（默认行为）。

选项 2：使用 Python 3.10 容器

如果你必须继续使用 Python 3.10，请在`@hybrid_job`装饰器中明确指定`image_uri`参数。

Python 3.10 容器标签：

映像名称	Tag
Base	1.0-cpu-py310-ubuntu22.04
CUDA-Q	0.12.0-cpu-py310-0.12.0
PyTorch	2.2.0-gpu-py310-cu121-ubuntu20.04
TensorFlow	2.14.1-gpu-py310-cu118-ubuntu20.04

此示例使用 us-west-2 区域。

完整图片 URIs：

```
Base:       292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:1.0-
cpu-py310-ubuntu22.04
CUDA-Q:     292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-cudaq-
jobs:0.12.0-cpu-py310-0.12.0
PyTorch:    292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-
jobs:2.2.0-gpu-py310-cu121-ubuntu20.04
TensorFlow: 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-tensorflow-
jobs:2.14.1-gpu-py310-cu118-ubuntu20.04
```

示例：

```
from braket.jobs.hybrid_job import hybrid_job
from braket.devices import Devices

device_arn = Devices.Amazon.SV1

@hybrid_job(
    device=device_arn,
```

```
    image_uri="292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-
jobs:1.0-cpu-py310-ubuntu22.04"
)
def my_job():
    pass
```

> **Note**
>
> - Python 3.10 容器将保持可用状态，但不会接收更新。
> - 请参阅为您的算法脚本定义环境。

Bring-Your-Own-Container (BYOC)

如果你的 Dockerfile 使用带有最新标签的 Braket 托管镜像，那么在 2026 年 1 月 21 日之后进行重建将提取支持 Python 3.12 的镜像。

要继续使用支持 Python 3.10 的 Braket 托管镜像，请更新你的 Dockerfile：

之前（升级后获得 Python 3.12）：

```
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:latest
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-cudaq-jobs:latest
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-tensorflow-jobs:latest
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:latest
```

之后（停留在 Python 3.10 上）：

```
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-base-jobs:1.0-cpu-
py310-ubuntu22.04
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-cudaq-jobs:0.12.0-cpu-
py310-0.12.0
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-pytorch-jobs:2.2.0-gpu-
py310-cu121-ubuntu20.04
FROM 292282985366.dkr.ecr.us-west-2.amazonaws.com/amazon-braket-tensorflow-jobs:2.14.1-
gpu-py310-cu118-ubuntu20.04
```

Braket 笔记本实例升级

按照以下步骤升级到 Python 3.12：

> **Important**
>
> 在删除笔记本实例之前，请确保已下载所有要保留的笔记本和文件。这些数据在删除后无法恢复。

1. 将您创建或修改的所有 Notebook 下载到本地驱动器。
2. 停用您的 Notebook 实例。
3. 删除您的 Notebook 实例。
4. 使用不同的名称创建一个新的笔记本实例。
5. 将您的笔记本上传到新实例。

OpenQASM 故障排除

本节提供了在使用 OpenQASM 3.0 的过程中遇到错误时可能有用的疑难解答指引。

本节内容：

包含语句错误

Braket 目前没有标准的门库文件可以包含在 OpenQASM 程序中。例如，以下示例会引发解析器错误。

```
OPENQASM 3;
include "standardlib.inc";
```

该代码生成错误消息：`No terminal matches '"' in the current parser context, at line 2 col 17.`

非连续 qubits 错误

如果在设备功能中将 `requiresContiguousQubitIndices` 设置为 `true` 的设备上使用非连续 qubits，将会导致错误。

在模拟器和 IonQ 上运行量子任务时，以下程序会触发错误。

```
OPENQASM 3;

qubit[4] q;

h q[0];
cnot q[0], q[2];
cnot q[0], q[3];
```

该代码生成错误消息：`Device requires contiguous qubits. Qubit register q has unused qubits q[1], q[4].`

物理 qubits 与虚拟 qubits 混搭错误

不得在同一个程序中将物理 qubits 和虚拟 qubits 混搭，这会导致错误。以下代码会生成错误。

```
OPENQASM 3;

qubit[2] q;
cnot q[0], $1;
```

该代码生成错误消息：`[line 4] mixes physical qubits and qubits registers.`

请求结果类型并在同一程序测量 qubits 错误

如果请求结果类型并在同一程序中明确测量 qubits，将会导致错误。以下代码会生成错误。

```
OPENQASM 3;

qubit[2] q;

h q[0];
cnot q[0], q[1];
measure q;

#pragma braket result expectation x(q[0]) @ z(q[1])
```

该代码生成错误消息：`Qubits should not be explicitly measured when result types are requested.`

超出经典寄存器和 qubit 寄存器限值误差

只允许使用一个经典寄存器和一个 qubit 寄存器。以下代码会生成错误。

```
OPENQASM 3;

qubit[2] q0;
qubit[2] q1;
```

该代码生成错误消息：`[line 4] cannot declare a qubit register. Only 1 qubit register is supported.`

方框前面没有逐字编译指示错误

所有方框前面都必须有逐字编译指示。以下代码会生成错误。

```
box{
rx(0.5) $0;
}
```

该代码生成错误消息：`In verbatim boxes, native gates are required. x is not a device native gate.`

逐字记录框缺少原生门错误

逐字记录框应有原生门和物理 qubits。以下代码会生成原生门错误。

```
#pragma braket verbatim
box{
x $0;
}
```

该代码生成错误消息：`In verbatim boxes, native gates are required. x is not a device native gate.`

逐字记录框缺少物理 qubits 错误

逐字记录框必须有物理 qubits。以下代码会生成物理 qubits 缺失错误。

```
qubit[2] q;

#pragma braket verbatim
box{
rx(0.1) q[0];
}
```

该代码生成错误消息：`Physical qubits are required in verbatim box.`

逐字编译指示缺少“braket”错误

您必须在逐字记录编译指示中包含“braket”。以下代码会生成错误。

```
#pragma braket verbatim          // Correct
#pragma verbatim                 // wrong
```

该代码生成错误消息：`You must include “braket” in the verbatim pragma`

无法为单个 qubits 编制索引错误

无法为单个 qubits 编制索引。以下代码会生成错误。

```
OPENQASM 3;
```

```
qubit q;
h q[0];
```

该代码生成错误：`[line 4] single qubit cannot be indexed.`

但是，可按如下方式对单个 qubit 数组编制索引：

```
OPENQASM 3;

qubit[1] q;
h q[0];   // This is valid
```

双 qubit 门中的物理 qubits 未连接错误

要使用物理 qubits，请首先通过检查 `device.properties.action[DeviceActionType.OPENQASM].supportPhysicalQubits` 来确认设备使用的是物理 qubits，然后通过选中 `device.properties.paradigm.connectivity.connectivityGraph` 或 `device.properties.paradigm.connectivity.fullyConnected` 来验证连接图。

```
OPENQASM 3;

cnot $0, $14;
```

该代码生成错误消息：`[line 3] has disconnected qubits 0 and 14`

本地模拟器支持警告

`LocalSimulator`支持 OpenQasm 中的高级功能，这些功能可能无法在按需模拟 QPUs 器上使用。如果您的程序仅包含对 `LocalSimulator` 特定的语言功能（如以下示例所示），您将收到一条警告。

```
qasm_string = """
qubit[2] q;

h q[0];
ctrl @ x q[0], q[1];
"""
qasm_program = Program(source=qasm_string)
```

此代码生成警告：`此程序使用仅支持的 OpenQasm 语言功能。 LocalSimulator QPUs 或按需模拟器可能不支持其中一些功能。

有关受支持的 OpenQASM 功能的更多信息，请浏览本地模拟器上对 OpenQASM 的高级功能支持页面。

Amazon Braket 中的安全性

云安全 AWS 是重中之重。作为 AWS 客户，您可以受益于专为满足大多数安全敏感型组织的要求而构建的数据中心和网络架构。

安全是双方共同承担 AWS 的责任。责任共担模式将其描述为云的安全性和云中的安全性：

- 云安全 — AWS 负责保护在云中运行 AWS 服务的基础架构 AWS Cloud。 AWS 还为您提供可以安全使用的服务。作为AWS 合规计划合规计划合规计划合的一部分，第三方审计师定期测试和验证我们安全的有效性。要了解适用于 Amazon Braket 的合规计划，请参阅按合规计划提供的范围内的 AWS 服务按合规计划划。
- 云端安全-您的责任由您使用的 AWS 服务决定。您还需要对其它因素负责，包括您的数据的敏感性、您公司的要求以及适用的法律法规。

此文档有助于您了解如何在使用 Braket 时应用责任共担模式。以下主题说明如何配置 Braket 以实现安全性和合规性目标。您还将学习如何使用其他 AWS 服务来帮助您监控和保护您的 Braket 资源。

本节内容：

共同承担安全责任

安全是双方共同承担 AWS 的责任。责任共担模式将其描述为云的 安全性和云中 的安全性：

- 云安全 — AWS 负责保护在云 AWS 服务 中运行的基础架构 AWS Cloud。 AWS 还为您提供可以安全使用的服务。作为 AWS 合规性计划的一部分，第三方审核人员将定期测试和验证安全性的有效性。要了解适用于 Amazon Braket 的合规性计划，请参阅 AWS 按合规性计划提供的范围内服务。

- 云端安全 — 您负责保持对托管在此 AWS 基础架构上的内容的控制。此内容包括您 AWS 服务 使用的的安全配置和管理任务。

数据保护

AWS 分担责任模型分担责任模型适用于 Amazon Braket 中的数据保护。如本模型所述 AWS ，负责保护运行所有内容的全球基础架构 AWS Cloud。您负责维护对托管在此基础结构上的内容的控制。您还负责您所使用的 AWS 服务 的安全配置和管理任务。有关数据隐私的更多信息，请参阅数据隐私常见问题。有关欧洲数据保护的信息，请参阅 AWS Security Blog 上的 AWS Shared Responsibility Model and GDPR 博客文章。

出于数据保护目的，我们建议您保护 AWS 账户 凭证并使用 AWS IAM Identity Center 或 AWS Identity and Access Management (IAM) 设置个人用户。这样，每个用户只获得履行其工作职责所需的权限。还建议您通过以下方式保护数据：

- 对每个账户使用多重身份验证（MFA）。
- 用于 SSL/TLS 与 AWS 资源通信。我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 使用设置 API 和用户活动日志 AWS CloudTrail。有关使用 CloudTrail 跟踪捕获 AWS 活动的信息，请参阅《AWS CloudTrail 用户指南》中的使用跟 CloudTrail 踪。
- 使用 AWS 加密解决方案以及其中的所有默认安全控件 AWS 服务。
- 使用高级托管安全服务（例如 Amazon Macie），它有助于发现和保护存储在 Amazon S3 中的敏感数据。
- 如果您在 AWS 通过命令行界面或 API 进行访问时需要经过 FIPS 140-3 验证的加密模块，请使用 FIPS 端点。有关可用的 FIPS 端点的更多信息，请参阅《美国联邦信息处理标准（FIPS）第 140-3 版》https://aws.amazon.com/compliance/fips/。

强烈建议您切勿将机密信息或敏感信息（如您客户的电子邮件地址）放入标签或自由格式文本字段（如名称字段）。这包括您 AWS 服务 使用控制台、API 或与 Amazon Braket 或其他人合作时。 AWS CLI AWS SDKs在用于名称的标签或自由格式文本字段中输入的任何数据都可能会用于计费或诊断日志。如果您向外部服务器提供 URL，强烈建议您不要在网址中包含凭证信息来验证对该服务器的请求。

数据留存

90 天后，Amazon Braket 会自动删除与您的量子任务 IDs 相关的所有量子任务和其他元数据。由于该数据留存策略，尽管这些任务和结果仍存储在您的 S3 存储桶中，但无法再通过从 Amazon Braket 控制台进行搜索来检索。

如果您需要访问在 S3 存储桶中存储超过 90 天的历史量子任务和结果，则必须单独记录任务 ID 以及与该数据关联的其他元数据。请务必在 90 天之前保存信息。您可以使用保存的信息来检索历史数据。

管理对 Amazon Braket 的访问权限

本章介绍运行 Amazon Braket 或限制特定用户和角色访问所需的权限。您可以向账户中的任何用户或角色授予（或拒绝）所需的权限。为此，请将相应的 Amazon Braket 策略附加到您账户中的该用户或角色，如以下各节所述。

作为先决条件，您必须启用 Amazon Braket。要启用 Braket，请务必以拥有 (1) 管理员权限或 (2) 已分配AmazonBraketFullAccess策略并有权创建亚马逊简单存储服务 (Amazon S3) 存储桶的用户或角色登录。

本节内容：

- Amazon Braket 资源
- Notebook 和角色
- AWS 亚马逊 Braket 的托管政策
- 限制用户访问某些设备
- 限制用户访问某些 Notebook 实例
- 限制用户访问某些 S3 存储桶

Amazon Braket 资源

Braket 创建了一种资源：量子任务资源。此 AWS 资源类型的资源名称 (ARN) 如下所示：

- 资源名称：AWS::Service::Braket
- ARN Regex：arn：$ {Partition}: b raket：$ {Region}: $ {Region}: $ {Account}: quantum-task/$ {} RandomId

Notebook 和角色

您可以在 Braket 中使用 Notebook 资源类型。笔记本是 Braket 能够共享的 SageMaker Amazon 人工智能资源。要将 Notebook 与 Braket 配合使用，必须指定名称以 `AmazonBraketServiceSageMakerNotebook` 开头的 IAM 角色。

要创建 Notebook，必须使用具有管理员权限或附加了以下内联策略的角色。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "CreateTheRole",
      "Effect": "Allow",
      "Action": "iam:CreateRole",
      "Resource": "arn:aws:iam::*:role/service-role/
AmazonBraketServiceSageMakerNotebookRole*"
    },
    {
      "Sid": "CreateThePolicy",
      "Effect": "Allow",
      "Action": "iam:CreatePolicy",
      "Resource": [
        "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookAccess*",
        "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookRole*"
      ]
    },
    {
      "Sid": "AttachTheRolePolicy",
      "Effect": "Allow",
      "Action": "iam:AttachRolePolicy",
      "Resource": "arn:aws:iam::*:role/service-role/
AmazonBraketServiceSageMakerNotebookRole*",
      "Condition": {
        "ArnLike": {
          "iam:PolicyARN": [
            "arn:aws:iam::aws:policy/AmazonBraketFullAccess",
```

```
            "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookAccess*",
            "arn:aws:iam::*:policy/service-role/
AmazonBraketServiceSageMakerNotebookRole*"
          ]
        }
      }
    }
  ]
}
```

要创建该角色，请按照“创建 Notebook ”页面中给出的步骤进行操作，或者让管理员为您创建该角色。确保已附上该AmazonBraketFullAccess政策。

创建角色后，您可以将该角色重复用于将来启动的所有 Notebook。

AWS 亚马逊 Braket 的托管政策

AWS 托管策略是由创建和管理的独立策略 AWS。 AWS 托管策略旨在为许多常见用例提供权限，以便您可以开始为用户、组和角色分配权限。

请记住， AWS 托管策略可能不会为您的特定用例授予最低权限权限，因为它们可供所有 AWS 客户使用。我们建议通过定义特定于使用案例的客户管理型策略来进一步减少权限。

您无法更改 AWS 托管策略中定义的权限。如果 AWS 更新 AWS 托管策略中定义的权限，则更新会影响该策略所关联的所有委托人身份（用户、组和角色）。 AWS 最有可能在启动新的 API 或现有服务可以使用新 AWS 服务 的 API 操作时更新 AWS 托管策略。

有关更多信息，请参阅《IAM 用户指南》中的 AWS 托管式策略。

主题

- AWS 托管策略： AmazonBraketFullAccess
- AWS 托管策略： AmazonBraketJobsExecutionPolicy
- AWS 托管策略： AmazonBraketServiceRolePolicy
- Amazon Braket 更新了托管 AWS 政策

AWS 托管策略： AmazonBraketFullAccess

该AmazonBraketFullAccess政策授予 Amazon Braket 操作权限，包括执行以下任务的权限：

- 从 Amazon Elastic Container Registry 下载容器：读取和下载用于 Amazon Braket Hybrid Jobs 功能的容器映像。容器必须符合“arn:aws:ecr:::repository/amazon-braket”的格式。
- 保留 AWS CloudTrail 日志-除了启动和停止查询、测试指标筛选器和筛选日志事件外，还适用于所有描述、获取和列出操作。该 AWS CloudTrail 日志文件包含您的账户中发生的所有 Amazon Braket API 活动的记录。
- 利用角色控制资源：在您的账户中创建服务相关角色。服务相关角色可以代表您访问 AWS 资源。它只能由 Amazon Braket 服务使用。此外，还可以将 IAM 角色传递给 Amazon Braket `CreateJob`API，创建角色并将范围限定为的策略附加 AmazonBraketFullAccess 到该角色。
- 创建日志组、日志事件和查询日志组，以维护您账户的使用情况日志文件：创建、存储和查看账户中有关 Amazon Braket 使用情况的日志信息。查询混合作业日志组的指标。包含正确的 Braket 路径并允许放置日志数据。将指标数据放入 CloudWatch。
- 在 Amazon S3 存储桶中创建和存储数据，并列出所有存储桶：要创建 S3 存储桶，请列出您账户中的 S3 存储桶，然后将对象放入账户中名称以 amazon-braket- 开头的任何存储桶并从中获取对象。Braket 需要这些权限才能将包含已处理量子任务结果的文件放入存储桶并从存储桶中检索这些文件。
- 传递 IAM 角色：将 IAM 角色传递给 `CreateJob` API。
- Amazon SageMaker AI Notebook — 创建和管理范围为 “arn: aws: sagemaker:: notebook-instance/amazon-braket-” 中资源的SageMaker笔记本实例。
- 验证服务配额 — 要创建 SageMaker AI 笔记本和 Amazon Braket Hybrid 任务，您的资源数量不能超过账户的配额。
- 查看产品定价：在提交工作负载之前，请查看并计划量子硬件成本。

要查看此策略的权限，请参阅 AWS 托管策略参考AmazonBraketFullAccess中的。

AWS 托管策略： AmazonBraketJobsExecutionPolicy

该AmazonBraketJobsExecutionPolicy策略授予在 Amazon Braket 混合任务中使用的执行角色的权限，如下所示：

- 从 Amazon Elastic Container Registry 下载容器：读取和下载用于 Amazon Braket Hybrid Jobs 功能的容器映像的权限。容器必须符合“arn:aws:ecr:*:*:repository/amazon-braket*”格式。

- 创建日志组、日志事件和查询日志组，以便维护您账户的使用情况日志文件：在您的账户中创建、存储和查看有关 Amazon Braket 使用情况的日志信息。查询混合作业日志组的指标。包含正确的 Braket 路径并允许放置日志数据。将指标数据放入 CloudWatch。
- 将数据存储在 Amazon S3 存储桶中：列出您账户中的 S3 存储桶，将对象放入账户名称中以 amazon-braket-开头的任何存储桶并从中获取对象。Braket 需要这些权限才能将包含已处理量子任务结果的文件放入存储桶并从存储桶中检索这些文件。
- 传递 IAM 角色 — 将 IAM 角色传递给 CreateJob API。角色必须符合 arn:aws:iam::*:role/service-role/AmazonBraketJobsExecutionRole* 的格式。

要查看此策略的权限，请参阅 AWS 托管策略参考AmazonBraketJobsExecutionPolicy中的。

AWS 托管策略： AmazonBraketServiceRolePolicy

该AmazonBraketServiceRolePolicy政策授予 Amazon Braket 操作权限，包括执行以下任务的权限：

- Amazon S3：列出您账户中的存储桶，以及将对象放入账户中、名称以 `amazon-braket-` 开头的任何存储桶并从中获取对象的权限。
- Amazon Lo CloudWatch gs — 列出和创建日志组、创建关联日志流以及将事件放入为 Amazon Braket 创建的日志组的权限。

有关服务相关角色的更多信息，请参阅和 Amazon Braket 服务相关的角色。

要查看此策略的权限，请参阅 AWS 托管策略参考AmazonBraketServiceRolePolicy中的。

Amazon Braket 更新了托管 AWS 政策

下表详细介绍了 Amazon Braket AWS 托管政策自该服务开始跟踪这些更改之时起的更新。

更改	描述	日期
AmazonBraketServiceRolePolicy-资源管理政策	在 Amazon S3 中添加了 “a ResourceAccount ws:”: “${aws:PrincipalAccount}” 条件范围并 CloudWatch 记录操作。	2025 年 7 月 11 日
AmazonBraketFullAccess-Braket 的完全访问政策	添加了 “定价：GetProducts” 操作。	2025 年 4 月 14 日

更改	描述	日期
AmazonBraketFullAccess-Braket 的完全访问政策	在 S3 操作中添加ResourceAccount了 “aws:”: “${aws:PrincipalAccount}” 条件范围。	2025 年 3 月 7 日
AmazonBraketFullAccess-Braket 的完全访问政策	添加了 s ervicequotas: GetServiceQuota 和 cloudwatch: 操作。GetMetricData	2023 年 3 月 24 日
AmazonBraketFullAccess-Braket 的完全访问政策	添加了 s3: ListAllMyBuckets 权限以查看和检查已使用的 Amazon S3 存储桶。	2022 年 3 月 31 日
AmazonBraketFullAccess-Braket 的完全访问政策	Braket 调整了 ia PassRole m: 包含`service-role/` 路径的权限。 AmazonBraketFullAccess	2021 年 11 月 29 日
AmazonBraketJobsExecutionPolicy-Amazon Braket Hybrid Jobs 的混合作业执行政策	Braket 更新了混合作业执行角色 ARN，使其包含 `service-role/` 路径。	2021 年 11 月 29 日
Braket 已开启跟踪更改	Braket 开始跟踪其 AWS 托管策略的变更。	2021 年 11 月 29 日

限制用户访问某些设备

要限制用户访问某些 Braket 设备，您可以向特定 IAM 角色添加拒绝权限策略。

可以限制以下操作：

- `CreateQuantumTask`：拒绝在指定设备上创建量子任务。
- `CreateJob`：拒绝在指定设备上创建混合作业。
- `GetDevice`：拒绝获取指定设备的详细信息。

以下示例限制了所有 QPUs 人的访问权限。 AWS 账户 `123456789012`

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
    "Effect": "Deny",
    "Action": [
      "braket:CreateQuantumTask",
      "braket:CreateJob",
      "braket:GetDevice"
    ],
    "Resource": [
    "arn:aws:braket:*:*:device/qpu/*"
    ],
    "Condition": {
             "StringEquals": {
                 "aws:PrincipalAccount": "123456789012"
             }
      }
    }
  ]
}
```

> **Note**
>
> 从策略中排除该 `braket:GetDevice` 操作，以启用用户通过 Braket 控制台对设备属性（例如设备可用性、校准数据和定价）的读取权限。

要改编此代码，请用受限设备的 Amazon 资源号（ARN）代替上一个示例中显示的字符串。此字符串提供资源值。在 Braket 中，设备代表一个 QPU 或模拟器，您可以调用它来运行量子任务。可用设备列在设备页面上。有两个架构用于指定对这些设备的访问权限：

- `arn:aws:braket:<region>:*:device/qpu/<provider>/<device_id>`
- `arn:aws:braket:<region>:*:device/quantum-simulator/<provider>/<device_id>`

以下是各种设备访问类型的示例

- 要在所有区域中选择 QPUs 全部，请执行以下操作：`arn:aws:braket:*:*:device/qpu/*`
- 要仅选择 us-west-2 区域 QPUs 中的所有内容，请执行以下操作：`arn:aws:braket:us-west-2:*:device/qpu/*`
- 同样，要仅选择 us-west-2 区域 QPUs 中的所有内容（因为设备是一种服务资源，而不是客户资源），请执行以下操作：`arn:aws:braket:us-west-2:*:device/qpu/*`
- 要限制对所有按需模拟器设备的访问权限，请执行以下操作：`arn:aws:braket:*:*:device/quantum-simulator/*`
- 要限制特定提供商对设备的访问权限（例如 Rigetti QPU 设备），请执行以下操作：`arn:aws:braket:*:*:device/qpu/rigetti/*`
- 要限制对 TN1 设备的访问，请执行以下操作：`arn:aws:braket:*:*:device/quantum-simulator/amazon/tn1`
- 要限制对所有 `Create` 操作的访问，请执行以下操作：`braket:Create*`

限制用户访问某些 Notebook 实例

要限制某些用户对特定 Braket Notebook 实例的访问，您可以向特定角色、用户或组添加拒绝权限策略。

以下示例使用策略变量有效地限制启动、停止和访问中特定笔记本实例的权限 AWS 账户 `123456789012`，该实例根据应具有访问权限的用户命名（例如，用户`Alice`将有权访问名为的笔记本实例`amazon-braket-Alice`）。

JSON

```
{
  "Version":"2012-10-17",
  "Statement": [
    {
      "Sid": "DenyCreateDeleteUpdateNotebookInstances",
      "Effect": "Deny",
      "Action": [
        "sagemaker:CreateNotebookInstance",
        "sagemaker:DeleteNotebookInstance",
        "sagemaker:UpdateNotebookInstance",
        "sagemaker:CreateNotebookInstanceLifecycleConfig",
        "sagemaker:DeleteNotebookInstanceLifecycleConfig",
        "sagemaker:UpdateNotebookInstanceLifecycleConfig"
```

```
      ],
      "Resource": "*"
    },
    {
      "Sid": "DenyDescribeStartStopNotebookInstances",
      "Effect": "Deny",
      "Action": [
        "sagemaker:DescribeNotebookInstance",
        "sagemaker:StartNotebookInstance",
        "sagemaker:StopNotebookInstance"
      ],
      "NotResource": [
        "arn:aws:sagemaker:*:123456789012:notebook-instance/amazon-braket-
${aws:username}"
      ]
    },
    {
      "Sid": "DenyNotebookInstanceUrl",
      "Effect": "Deny",
      "Action": [
        "sagemaker:CreatePresignedNotebookInstanceUrl"
      ],
      "NotResource": [
        "arn:aws:sagemaker:*:123456789012:notebook-instance/amazon-braket-
${aws:username}*"
      ]
    }
  ]
}
```

限制用户访问某些 S3 存储桶

要限制某些用户访问特定 Amazon S3 存储桶，您可以向特定角色、用户或组添加拒绝策略。

以下示例限制了检索对象并将其放入特定 S3 存储桶 (`arn:aws:s3:::amazon-braket-us-east-1-123456789012-Alice`) 的权限，还限制了这些对象的列表。

JSON

```
{
  "Version":"2012-10-17",
```

```
  "Statement": [
    {
      "Effect": "Deny",
      "Action": [
        "s3:ListBucket"
      ],
      "NotResource": [
        "arn:aws:s3:::amazon-braket-us-east-1-123456789012-Alice"
      ]
    },
    {
      "Effect": "Deny",
      "Action": [
        "s3:GetObject"
      ],
      "NotResource": [
        "arn:aws:s3:::amazon-braket-us-east-1-123456789012-Alice/*"
      ]
    }
  ]
}
```

要限制某个 Notebook 实例对存储桶的访问，您可以将上述策略添加到 Notebook 执行角色中。

Amazon Braket 服务相关角色

启用 Amazon Braket 时，会在您的账户中创建一个服务相关角色。

服务相关角色是一种独特类型的 IAM 角色，它与 Amazon Braket 直接相关。Amazon Braket 服务相关角色是预定义的，包含服务代表您调用其他 AWS 服务 服务时 Braket 所需的所有权限。

服务相关角色可让您更轻松地设置 Amazon Braket，因为您不必手动添加必要的权限。Amazon Braket 可定义其服务相关角色的权限。除非您更改这些定义，否则只有 Amazon Braket 才能承担其角色。定义的权限包括信任策略和权限策略。不能将该权限策略附加到任何其他 IAM 实体。

Amazon Braket 设置的服务相关角色是 AWS Identity and Access Management（IAM）服务相关角色功能的一部分。有关支持服务相关角色的其他 AWS 服务 信息，请参阅使用 IAM 的 AWS 服务，并查找服务相关角色列中显示为“是”的服务。请选择是与查看该服务的服务相关角色文档的链接。

有关服务相关角色的 AWS 托管式策略的更多信息，请参阅 AmazonBraketServiceRolePolicy。

Amazon Braket 的合规性验证

> **Note**
>
> AWS 合规性报告不涵盖第三方硬件提供商提供的 QPU，他们可以选择实施自己的独立审计。

要了解某个 AWS 服务 是否在特定合规性计划范围内，请参阅合规性计划范围内的 AWS 服务，然后选择您感兴趣的合规性计划。有关常规信息，请参阅 AWS 合规性计划、、。

您可以使用 AWS Artifact 下载第三方审计报告。有关更多信息，请参阅在 AWS Artifact 中下载报告、。

您在使用 AWS 服务 时的合规性责任由您的数据的敏感性、您公司的合规性目标以及适用的法律法规决定。有关您在使用 AWS 服务 时的合规责任的更多信息，请参阅 AWS 安全性文档。

Amazon Braket 中的基础设施安全性

作为一项托管式服务，Amazon Braket 受 AWS 全球网络安全保护。有关 AWS 安全服务以及 AWS 如何保护基础设施的信息，请参阅 AWS 云安全性。要按照基础设施安全最佳实践设计您的 AWS 环境，请参阅《安全性支柱 AWS Well-Architected Framework》中的基础设施保护。

您可以使用 AWS 发布的 API 调用通过网络访问 Amazon Braket。客户端必须支持以下内容：

- 传输层安全性协议（TLS）。我们要求使用 TLS 1.2，建议使用 TLS 1.3。
- 具有完全向前保密（PFS）的密码套件，例如 DHE（临时 Diffie-Hellman）或 ECDHE（临时椭圆曲线 Diffie-Hellman）。大多数现代系统（如 Java 7 及更高版本）都支持这些模式。

您可以从任何网络位置调用这些 API 操作，但 Braket 不支持基于资源的访问策略，其中可以包含基于源 IP 地址的限制。您还可以使用 Braket 策略控制来自特定 Amazon Virtual Private Cloud (Amazon VPC) 端点或特定 VPC 的访问。事实上，这隔离了在 AWS 网络中仅从特定 VPC 到给定 Braket 资源的网络访问。

Amazon Braket 硬件提供商的安全性

Amazon Braket 上的 QPU 由第三方硬件提供商托管。当您在 QPU 上运行量子任务时，Amazon Braket 会使用 DeviceARN 作为标识符，将电路发送到指定的 QPU 进行处理。

如果您使用 Amazon Braket 访问由第三方硬件提供商运营的量子计算硬件，您的电路及其相关数据将由不在 AWS 运营设施内的硬件提供商处理。有关每个 QPU 可用的物理位置和 AWS 区域的信息，可在 Amazon Braket 控制台的“设备详情”部分找到。

您的内容会被匿名处理。只有处理电路所需的内容才会发送给第三方。AWS 账户 信息不会传输给第三方。

静态和传输中的所有数据均会被加密。数据解密仅用于处理之用。Amazon Braket 的第三方提供商不得存储您的内容，也不得将其用于电路处理以外的其他目的。电路完成后，结果将返回到 Amazon Braket 并存储在您的 S3 存储桶中。

定期审计 Amazon Braket 第三方量子硬件提供商的安全性，以确保符合网络安全、访问控制、数据保护和物理安全标准。

适用于 Amazon Braket 的 Amazon VPC 端点

您可以通过创建接口 VPC 端点，在 Amazon VPC 与 Amazon Braket 之间建立私有连接。接口端点由一种技术提供支持 AWS PrivateLink，该技术 APIs 无需互联网网关、NAT 设备、VPN 连接或 Direct Connect 连接即可访问 Braket。您的 VPC 中的实例不需要公有 IP 地址即可与 Braket APIs 通信。

每个接口端点均由子网中的一个或多个弹性网络接口表示。

使用 AWS PrivateLink，您的 VPC 和 Braket 之间的流量不会离开Amazon网络，这可以提高您与基于云的应用程序共享的数据的安全性，因为它可以减少您的数据暴露在公共互联网上的风险。有关更多信息，请参阅 Amazon VPC 用户指南中的使用接口 VPC 终端节点访问 AWS 服务。

本节内容：

- Amazon Braket VPC 端点注意事项
- 设置 Braket 然后 PrivateLink
- 有关创建端点的其他信息
- 使用 Amazon VPC 端点策略控制访问

Amazon Braket VPC 端点注意事项

请务必先查看《Amazon VPC 用户指南》中的接口端点先决条件，然后再为 Braket 连接设置接口 VPC 端点。

Braket 支持从 VPC 调用它的所有 API 操作。

默认情况下，允许通过 VPC 端点对 Braket 进行完全访问。如果您指定 VPC 端点策略，则可以控制访问。有关更多信息，请参阅《Amazon VPC 用户指南》中的使用端点策略控制对 VPC 端点的访问。

设置 Braket 然后 PrivateLink

要 AWS PrivateLink 与 Amazon Braket 一起使用，您必须创建一个亚马逊虚拟私有云（亚马逊 VPC）终端节点作为接口，然后通过 Amaz API on Braket 服务连接到该终端节点。

以下是该过程的一般步骤，将在后面的章节中详细介绍。

- 配置并启动 Amazon VPC 来托管您的 AWS 资源。如果您已有 VPC，请跳过此步骤。
- 为 Braket 创建 Amazon VPC 端点
- 通过您的端点连接并运行 Braket 量子任务

步骤 1：如果需要，启动 Amazon VPC

请记住，如果您的账户已有 VPC 在运行，则可以跳过此步骤。

使用 VPC 控制您的网络设置，例如 IP 地址范围、子网、路由表和网络网关。本质上，您是在自定义虚拟网络中启动 AWS 资源。有关更多信息 VPCs，请参阅 Amazon VPC 用户指南。

打开 Amazon VPC 控制台，创建一个包含子网、安全组和网络网关的新 VPC。

第 2 步：创建 Braket 接口 VPC 端点

您可以使用 Amazon VPC 控制台或 AWS Command Line Interface (AWS CLI) 为 Braket 服务创建 VPC 终端节点。有关更多信息，请参阅 Amazon VPC 用户指南中的创建 VPC 端点。

要在控制台中创建 VPC 端点，请打开 Amazon VPC 控制台，打开端点页面，然后继续创建新的端点。记下端点 ID 以供日后参考。当您对 Braket API 进行某些调用时，它必须作为 `–endpoint-url` 标志的一部分。

使用以下服务名称为 Braket 创建 VPC 端点：

- `com.amazonaws.substitute_your_region.braket`

有关更多信息，请参阅 Amazon VPC 用户指南中的使用接口 VPC 终端节点访问 AWS 服务。

第 3 步：通过端点连接并运行 Braket 量子任务

在创建 VPC 端点后，您可以使用以下示例之类的 CLI 命令，包括指定 API 或运行时接口端点的 `endpoint-url` 参数：

```
aws braket search-quantum-tasks --endpoint-url
 VPC_Endpoint_ID.braket.substituteYourRegionHere.vpce.amazonaws.com
```

如果为 VPC 端点启用专用 DNS 主机名，您不需要指定端点作为 CLI 命令的 URL。相反，CLI 和 Braket SDK 默认使用的 Amazon Braket API DNS 主机名会解析到您的 VPC 端点。它的格式如下例所示：

```
https://braket.substituteYourRegionHere.amazonaws.com
```

这篇名为 “使用 AWS PrivateLink 终端节点从 Amazon VPC 直接访问 Amazon A SageMaker I 笔记本电脑” 的博客文章提供了一个示例，说明如何设置终端节点以与 SageMaker 笔记本建立安全连接，与 Amazon Braket 笔记本类似。

如果你正在按照博客文章中的步骤进行操作，请记得用 AmazonBrak et 这个名字代替 Amazon A SageMaker I。如果您的地区不是 us-ea st- 1，请在服务 AWS 区域 名称中输入您的正确名称`com.amazonaws.us-east-1.braket`或将其替换为该字符串。

有关创建端点的其他信息

- 有关创建具有私有子网的 VPC 的信息，请参阅创建具有私有子网的 VPC。
- 有关使用 Amazon VPC 控制台或创建和配置终端节点的信息 AWS CLI，请参阅 Amazon VPC 用户指南中的创建 VPC 终端节点。
- 有关使用创建和配置终端节点的信息 CloudFormation，请参阅《CloudFormation 用户指南》中的 AWS:: EC2:: VPCEndpoint 资源。

使用 Amazon VPC 端点策略控制访问

要控制对 Amazon Braket 的连接访问，您可以在创建 Amazon VPC 端点时附加 AWS Identity and Access Management （IAM）端点策略。该策略指定以下信息：

- 可以执行操作的主体（用户或角色）。
- 可执行的操作。

- 可对其执行操作的资源。

有关更多信息，请参阅《Amazon VPC 用户指南》中的使用端点策略控制对 VPC 端点的访问。

示例：操作的 VPC 端点策略

下面是用于 Braket 的端点策略的示例。当附加到端点时，此策略会向所有资源上的所有主体授予对列出的 Braket 操作的访问权限。

```
{
 "Statement":[
 {
   "Principal":"*",
   "Effect":"Allow",
   "Action":[
     “braket:action-1",
     “braket:action-2",
     “braket:action-3”
     ],
   "Resource":"*"
   }
  ]
}
```

您可以通过附加多个端点策略来创建复杂的 IAM 规则。有关更多信息以及示例，请参阅：

- Step Functions 的亚马逊虚拟私有云端点策略
- 为非管理员用户创建精细的 IAM 权限
- 使用终端节点策略控制对 VPC 终端节点的访问

日志记录和监控

通过 Amazon Braket 服务提交量子任务后，您可以通过 Amazon Braket SDK 和控制台密切监控该任务的状态和进度。这为您提供了一个集中式界面，用于跟踪工作负载的实施，识别任何潜在的瓶颈或问题，并采取适当的措施来优化量子应用程序的性能和可靠性。量子任务完成后，Braket 会将结果保存在您指定的 Amazon S3 位置。量子任务的完成时间可能会有所不同，特别是对于在量子处理单元（QPU）设备上运行的任务而言。这在很大程度上是由于执行队列的长度，因为量子硬件资源在多个用户之间共享。

状态类型列表：

- `CREATED`：Amazon Braket 收到了您的量子任务。
- `QUEUED`：Amazon Braket 处理了您的量子任务，现在它正在等待在设备上运行。
- `RUNNING`：您的量子任务在 QPU 或按需模拟器上运行。
- `COMPLETED`：您的量子任务在 QPU 或按需模拟器上运行完毕。
- `FAILED`：您的量子任务尝试运行但失败了。根据您的量子任务失败的原因，请尝试再次提交量子任务。
- `CANCELLED`：您取消了量子任务。量子任务没有运行。

本节内容：

通过 Amazon Braket SDK 跟踪量子任务

命令 `device.run(…)` 定义具有唯一量子任务 ID 的量子任务。您可以使用 `task.state()` 查询和跟踪状态，如以下示例所示。

注意：task = device.run() 是一种异步操作，这意味着当系统在后台处理量子任务时，您可以继续工作。

检索结果

当您调用 task.result() 时，SDK 开始轮询 Amazon Braket，查看量子任务是否完成。SDK 使用您在 .run() 中定义的轮询参数。量子任务完成后，SDK 会从 S3 存储桶中检索结果并将其作为 QuantumTaskResult 对象返回。

```
# create a circuit, specify the device and run the circuit
circ = Circuit().rx(0, 0.15).ry(1, 0.2).cnot(0,2)
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
task = device.run(circ, s3_location, shots=1000)

# get ID and status of submitted task
task_id = task.id
status = task.state()
print('ID of task:', task_id)
print('Status of task:', status)
# wait for job to complete
while status != 'COMPLETED':
    status = task.state()
    print('Status:', status)
```

```
ID of task:
arn:aws:braket:us-west-2:123412341234:quantum-task/b68ae94b-1547-4d1d-aa92-1500b82c300d
Status of task: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: QUEUED
Status: RUNNING
Status: RUNNING
Status: COMPLETED
```

取消量子任务

要取消量子任务，请调用 cancel() 方法，如以下示例所示。

```
# cancel quantum task
task.cancel()
status = task.state()
print('Status of task:', status)
```

```
Status of task: CANCELLING
```

检查元数据

您可以检查已完成的量子任务的元数据，如以下示例所示。

```
# get the metadata of the quantum task
metadata = task.metadata()
# example of metadata
shots = metadata['shots']
date = metadata['ResponseMetadata']['HTTPHeaders']['date']
# print example metadata
print("{} shots taken on {}.".format(shots, date))

# print name of the s3 bucket where the result is saved
results_bucket = metadata['outputS3Bucket']
print('Bucket where results are stored:', results_bucket)
# print the s3 object key (folder name)
results_object_key = metadata['outputS3Directory']
print('S3 object key:', results_object_key)

# the entire look-up string of the saved result data
look_up = 's3://'+results_bucket+'/'+results_object_key
print('S3 URI:', look_up)
```

```
1000 shots taken on Wed, 05 Aug 2020 14:44:22 GMT.
Bucket where results are stored: amazon-braket-123412341234
S3 object key: simulation-output/b68ae94b-1547-4d1d-aa92-1500b82c300d
S3 URI: s3://amazon-braket-123412341234/simulation-output/b68ae94b-1547-4d1d-
aa92-1500b82c300d
```

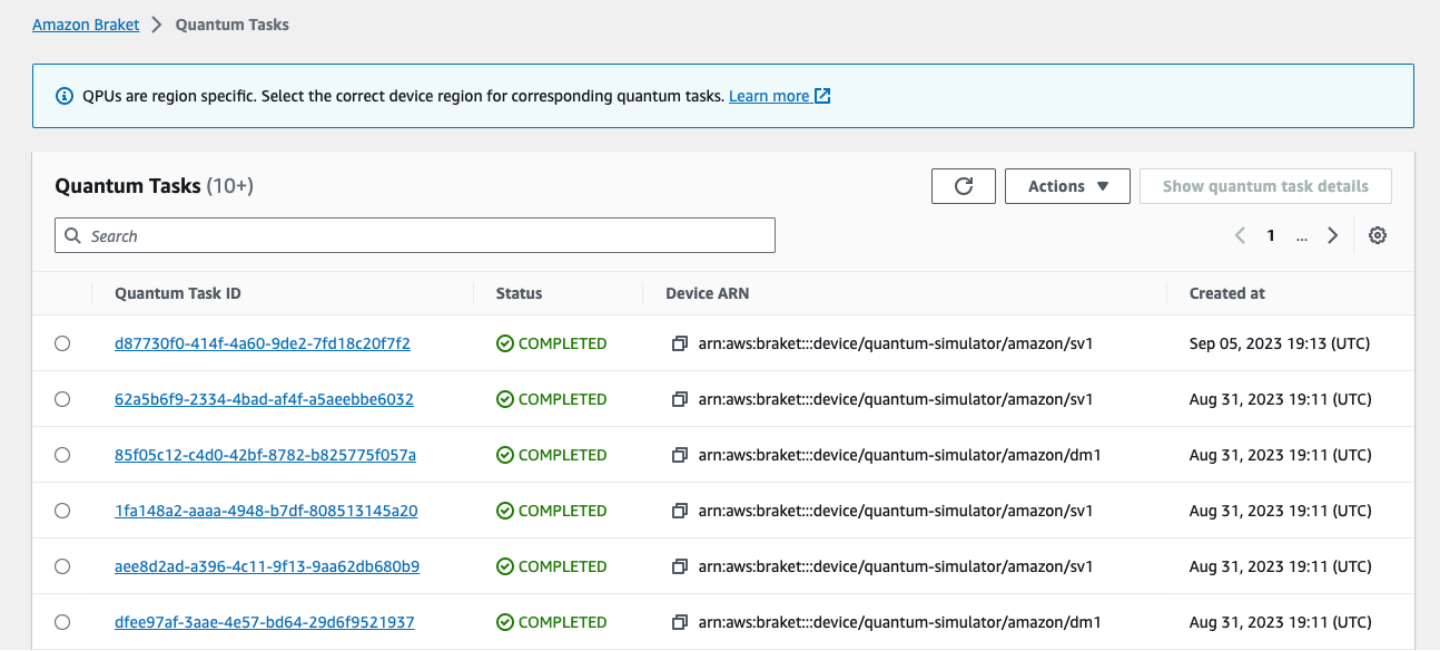
检索量子任务或结果

如果您的内核在您提交量子任务后死亡，或者您关闭了 Notebook 或电脑，您可以用其唯一的 ARN（量子任务 ID）重建 `task` 对象。然后，您可以调用 `task.result()`，从存储结果的 S3 存储桶中获取结果。

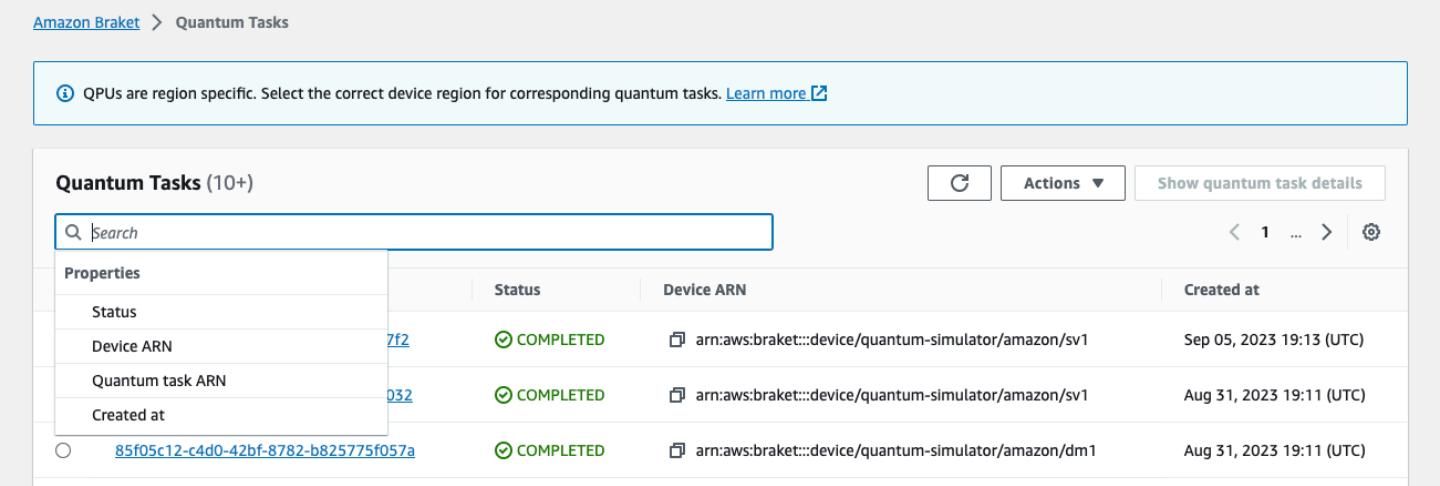
```
from braket.aws import AwsSession, AwsQuantumTask

# restore task with unique arn
task_load = AwsQuantumTask(arn=task_id)
# retrieve the result of the task
result = task_load.result()
```

通过 Amazon Braket 控制台监控量子任务

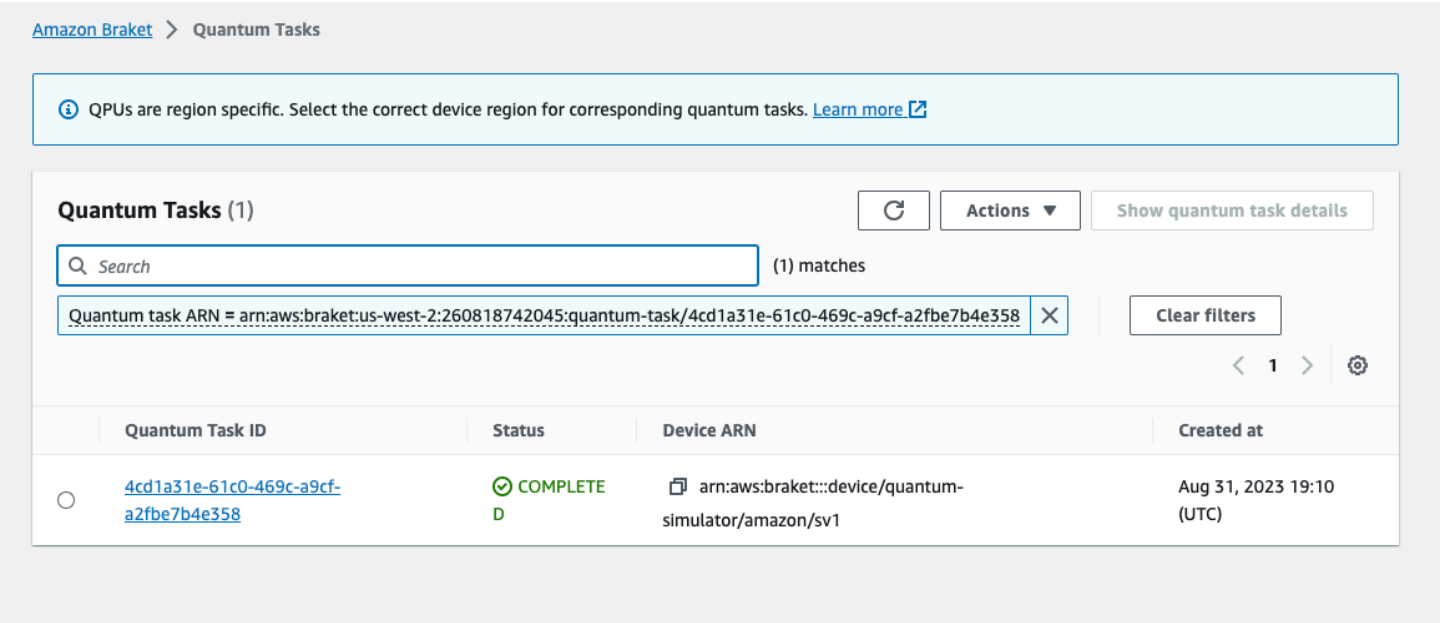
Amazon Braket 提供了一种通过 Amazon Braket 控制台监控量子任务的便捷方法。所有提交的量子任务都列在“量子任务”字段中，如下图所示。该服务是特定于区域的，这意味着您只能查看在特定区域中创建的量子任务。 AWS 区域



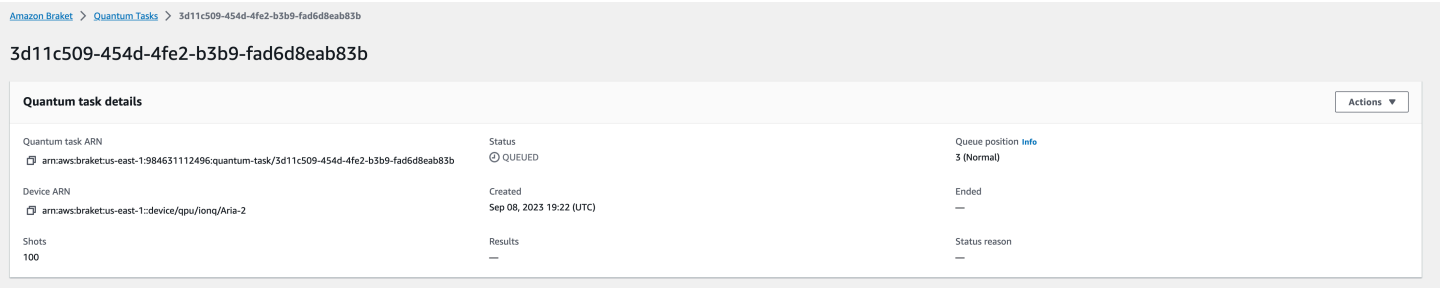
您可以通过导航栏搜索特定的量子任务。搜索可以基于量子任务 ARN（ID）、状态、设备和创建时间。当您选择导航栏时，选项会自动出现，如以下示例所示。



下图显示了基于量子任务的唯一量子任务 ID 搜索量子任务的示例，该任务可通过调用 `task.id` 获得。



此外，如下图所示，可以在量子任务处于 QUEUED 状态时对其状态进行监控。点击量子任务 ID 会显示详情页面。此页面显示量子任务相对于要处理的设备的动态队列位置。



作为混合作业的一部分提交的量子任务在排队时将具有优先级。在混合作业之外提交的量子任务将具有正常的排队优先级。

想要查询 Braket SDK 的客户可以通过编程方式获取其量子任务和混合作业队列位置。有关更多信息，请参阅“我的任务何时运行”页面。

标注 Amazon Braket 资源

标签是您分配或分配给 AWS 资源的自定义属性标签。 AWS 标签是元数据，可详细介绍您的资源。每个标签均包含一个键 和一个值。这些被统称为键/值对。对于您分配的标签，需要定义键和值。

在 Amazon Braket 控制台中，您可以导航到量子任务或 Notebook 并查看与之相关的标签列表。您可以添加标签、移除标签或修改标签。可以在创建量子任务或笔记本时对其进行标记，然后通过控制台 AWS CLI、或管理关联的标签API。

更多关于 AWS 和标签

- 有关标签的一般信息，包括命名和使用惯例，请参阅什么是标签编辑器？ 在《标签 AWS 资源和标签编辑器用户指南》中。
- 有关标签限制的信息，请参阅《标签 AWS 资源和标签编辑器用户指南》中的标签命名限制和要求。
- 有关最佳实践和标签策略，请参阅标记 AWS 资源的最佳实践。
- 有关支持使用标签的服务的列表，请参阅Resource Groups 标记 API 参考。

以下各部分提供了有关 Amazon Braket 标签的更多信息。

本节内容：

使用标签

标签可以将您的资源组织成对您有用的类别。例如，您可以分配一个指定拥有该资源的部门的“Department”标签。

每个 标签具有两个部分：

- 标签密钥（例如 CostCenter，“环境” 或 “项目”）。标签密钥区分大小写。
- 一个称为标签值的可选字段（例如，111122223333 或 Production）。省略标签值与使用空字符串效果相同。与标签键一样，标签值区分大小写。

标签可帮助您：

- 识别和整理您的 AWS 资源。许多都 AWS 服务 支持标记，因此您可以为来自不同服务的资源分配相同的标签，以表明这些资源是相关的。
- 追踪您的 AWS 成本。您可以在 AWS 账单与成本管理 控制面板上激活这些标签。 AWS 使用标签对您的成本进行分类，并向您提供每月成本分配报告。有关更多信息，请参阅AWS 账单与成本管理 用户指南中的使用成本分配标签。
- 控制对 AWS 资源的访问权限。有关更多信息，请参阅使用标签控制访问。

Amazon Braket 中支持的标注资源

Amazon Braket 中的下列资源类型支持标注：

- `quantum-task` 资源
- 资源名称：`AWS::Service::Braket`
- ARN 正则表达式：`arn:${Partition}:braket:${Region}:${Account}:quantum-task/${RandomId}`

注意：尽管笔记本实际上是 Amazon A SageMaker I 资源，但您可以在 Amazon Braket 控制台中为自己Amazon的 Braket 笔记本电脑应用和管理，方法是使用控制台导航到笔记本资源。有关更多信息，请参阅 SageMaker 文档中的笔记本实例元数据。

使用 Amazon Braket API 进行标注

- 如果您使用 Amazon Braket API 在资源上设置标签，请调用 `TagResource`API。

```
aws braket tag-resource --resource-arn $YOUR_TASK_ARN --tags {\"city\":
\"Seattle\"}
```

- 要从资源中移除标签，请调用 UntagResourceAPI。

```
aws braket list-tags-for-resource --resource-arn $YOUR_TASK_ARN
```

- 要列出附加到特定资源的所有标签，请调用 ListTagsForResourceAPI。

```
aws braket tag-resource --resource-arn $YOUR_TASK_ARN --tag-keys "[\"city
\",\"state\"]"
```

标注限制

以下基本限制适用于 Amazon Braket 资源上的标签：

- 您可以分配给资源的最大标签数量：50
- 最大键长度：128 个 Unicode 字符
- 最大值长度：256 个 Unicode 字符
- 键和值的有效字符：`a-z, A-Z, 0-9, space`，以及以下字符：`_ . : / = + -` 和 `@`
- 键和值区分大小写。
- 请勿`aws`用作密钥的前缀；它是保留供 AWS 使用的。

在 Amazon Braket 中管理标签

您可在资源上将标签设置为属性。您可以通过 Amazon Braket 控制台、Amazon Braket API 或 AWS CLI，查看、添加、修改、列出和删除标签。有关更多信息，请参阅 Amazon Braket API 参考。

本节内容：

添加 标签

您可以在以下时间将标签添加到可标注资源中：

- 创建资源时：使用控制台，或者在 AWS API 中将 Tags 参数包含在 Create 操作中。
- 创建资源后：使用控制台导航到量子任务或 Notebook 资源，或者在 AWS API 中调用 TagResource 操作。

要在创建资源时向资源添加标签，您还需要创建指定类型的资源的权限。

查看标签

您可以使用控制台导航到任务或笔记本资源，或者通过调用操作来查看 Amazon Braket 中任何可标记资源的标签。 AWS ListTagsForResource API

您可以使用以下 AWS API命令查看资源上的标签：

- AWS API: ListTagsForResource

编辑标签

您可以使用控制台导航到量子任务或 Notebook 资源来编辑标签，也可以使用以下命令修改附加到可标注资源的标签的值。当您指定已存在的标签键时，该键的值将被覆盖：

- AWS API: TagResource

移除标签

您可以通过指定要移除的键，或者使用控制台导航到量子任务或 Notebook 资源或在调用 UntagResource 操作时，从资源中移除标签。

- AWS API: UntagResource

在 Amazon Brake AWS CLI t 中添加标签的示例

当您使用 AWS Command Line Interface (AWS CLI) 与 Amazon Braket 交互时，以下代码是一个示例命令，用于演示如何创建适用于您创建的量子任务的标签。在此示例中，任务是在 SV1 量子模拟器上执行的，其参数设置为 Rigetti 量子处理单元（QPU）。重要的是，在示例命令中，标签在所有其他必

需参数之后最后指定。在本例中，标签的键为 state，值为 Washington。这些标签可以用来帮助对这个特定的量子任务进行分类或识别。

```
aws braket create-quantum-task --action /
"{\"braketSchemaHeader\": {\"name\": \"braket.ir.jaqcd.program\", /
    \"version\": \"1\"}, /
    \"instructions\": [{\"angle\": 0.15, \"target\": 0, \"type\": \"rz\"}], /
    \"results\": null, /
    \"basis_rotation_instructions\": null}" /
  --device-arn "arn:aws:braket:::device/quantum-simulator/amazon/sv1" /
  --output-s3-bucket  "my-example-braket-bucket-name" /
  --output-s3-key-prefix "my-example-username"  /
  --shots 100   /
  --device-parameters /
  "{\"braketSchemaHeader\": /
     {\"name\": \"braket.device_schema.rigetti.rigetti_device_parameters\", /
      \"version\": \"1\"}, \"paradigmParameters\": /
       {\"braketSchemaHeader\": /
         {\"name\": \"braket.device_schema.gate_model_parameters\", /
          \"version\": \"1\"}, /
          \"qubitCount\": 2}}" /
          --tags {\"state\":\”Washington\"}
```

此示例演示了在通过 AWS CLI运行时，如何将标签应用于您的量子任务，这对于组织和跟踪您的 Braket 资源很有帮助。

使用以下方法监控您的量子任务 EventBridge

亚马逊 EventBridge 监控 Amazon Braket 量子任务中的状态变更事件。来自 Amazon Braket 的 EventBridge活动几乎是实时的。您可以通过编写规则来指示所关注的事件，包括要在事件匹配规则时执行的自动化操作。可触发的自动操作包括以下操作：

- 调用函数 AWS Lambda
- 激活 AWS Step Functions 状态机
- 向 Amazon SNS 主题发送通知

EventBridge 监控以下 Amazon Braket 状态更改事件：

- 量子任务的状态发生变化

Amazon Braket 保证交付量子任务状态变更事件。这些事件至少传送一次，但可能会出现故障。

有关更多信息，请参阅 Amazon 中的活动 EventBridge。

本节内容：

使用监控量子任务状态 EventBridge

借助 EventBridge，您可以创建规则，定义在 Amazon Braket 发送有关 Braket 量子任务的状态变更通知时要采取的操作。例如，您可以创建一个规则，每当量子任务状态发生变化时，就向您发送电子邮件。

1. AWS 使用有权使用 EventBridge 和 Amazon Braket 的账户登录。
2. 打开 Amazon EventBridge 控制台。
3. 使用以下值创建 EventBridge 规则：
   - 对于规则类型，选择具有事件模式的规则。
   - 对于事件源，选择其他。
   - 在事件模式部分，选择自定义模式(JSON 编辑器)，然后将以下事件模式粘贴到文本区域：

```
{
  "source": [
    "aws.braket"
  ],
  "detail-type": [
    "Braket Task State Change"
  ]
}
```

要从 Amazon Braket 中捕获所有事件，请排除该 `detail-type` 部分，如以下代码所示：

```
{
  "source": [
    "aws.braket"
  ]
}
```

- 对于目标类型，选择 AWS 服务，在选择目标中，选择目标，例如 Amazon SNS 主题或 AWS Lambda 函数。当收到来自 Amazon Braket 的量子任务状态变化事件时，就会触发目标。

  例如，使用 Amazon Simple Notification Service（SNS）主题在事件发生时发送电子邮件或短信。您首先需要使用 Amazon SNS 控制台创建 Amazon SNS 主题。要了解更多信息，请参阅使用 Amazon SNS 发送用户通知。

有关创建规则的详细信息，请参阅创建对事件做出反应的 Amazon EventBridge 规则。

亚马逊 Braket 活动 EventBridge 示例

有关 Amazon Braket Quantum 任务状态更改事件字段的信息，请参阅亚马逊中的事件。 EventBridge

以下属性显示在 JSON 的“详细信息”字段中。

- **quantumTaskArn** (str)：生成此事件的量子任务。
- **status** (Optional[str])：量子任务过渡到的状态。
- **deviceArn** (str)：为其创建此量子任务的用户指定的设备。
- shots (int)：用户 shots 请求的数量。
- **outputS3Bucket** (str)：用户指定的输出存储桶。
- **outputS3Directory** (str)：用户指定的输出键前缀。
- **createdAt** (str)：以 ISO-8601 字符串表示的量子任务创建时间。
- **endedAt** (Optional[str])：量子任务达到终端状态的时间。只有当量子任务过渡到终端状态时，该字段才会出现。

以下 JSON 代码显示了 Amazon Braket Quantum 任务状态变更事件的示例。

```
{
    "version":"0",
    "id":"6101452d-8caf-062b-6dbc-ceb5421334c5",
    "detail-type":"Braket Task State Change",
    "source":"aws.braket",
    "account":"012345678901",
    "time":"2021-10-28T01:17:45Z",
    "region":"us-east-1",
    "resources":[
        "arn:aws:braket:us-east-1:012345678901:quantum-task/834b21ed-77a7-4b36-a90c-
c776afc9a71e"
```

```
    ],
    "detail":{
        "quantumTaskArn":"arn:aws:braket:us-east-1:012345678901:quantum-
task/834b21ed-77a7-4b36-a90c-c776afc9a71e",
        "status":"COMPLETED",
        "deviceArn":"arn:aws:braket:::device/quantum-simulator/amazon/sv1",
        "shots":"100",
        "outputS3Bucket":"amazon-braket-0260a8bc871e",
        "outputS3Directory":"sns-testing/834b21ed-77a7-4b36-a90c-c776afc9a71e",
        "createdAt":"2021-10-28T01:17:42.898Z",
        "eventName":"MODIFY",
        "endedAt":"2021-10-28T01:17:44.735Z"
      }
}
```

使用监控您的指标 CloudWatch

您可以使用 Amazon 监控 Amazon Braket CloudWatch，亚马逊会收集原始数据并将其处理为可读的近乎实时的指标。您可以在亚马逊 CloudWatch 控制台中查看 15 个月前生成的历史信息或最近 2 周内更新的搜索指标，以便更好地了解 Amazon Braket 的表现。要了解更多信息，请参阅使用 CloudWatch 指标。

> **Note**
>
> 您可以通过导航到 Amazon AI 控制台上的笔记本详情页面来查看 Ama SageMaker zon Braket 笔记本的 CloudWatch 日志流。其他 Amazon Braket 笔记本设置可通过控制台获得。SageMaker

本节内容：

- Amazon Braket 指标与维度

Amazon Braket 指标与维度

指标是中的基本概念 CloudWatch。指标表示发布到的一组按时间顺序排列的数据点。 CloudWatch每个指标都以一组维度为特征。要详细了解中的指标维度 CloudWatch，请参阅CloudWatch 维度。

Amazon Braket 将以下特定于 Amazon Braket 的指标数据发送到亚马逊指标中： CloudWatch

量子任务指标

如果存在量子任务，则可以使用指标。它们显示在控制台的 AWS/Braket/By Device 下。 CloudWatch

指标	说明
计数	量子任务数。
延迟	量子任务完成时，将发出此指标。它表示从量子任务初始化到完成的总时间。

量子任务指标的维度

量子任务指标以基于 `deviceArn` 参数的维度发布，其格式为 arn:aws:braket:::device/xxx。

使用记录你的量子任务 CloudTrail

Amazon Braket 与 AWS CloudTrail一项服务集成，该服务提供用户、角色或用户在 Amazon Braket AWS 服务 中采取的操作的记录。 CloudTrail 将所有对 Amazon Braket 的API通话记录为事件。捕获的调用包括通过 Amazon Braket 控制台的调用以及对 Amazon Braket 操作的代码调用。如果您创建了跟踪，则可以允许将 CloudTrail 事件持续传输到 Amazon S3 存储桶，包括 Amazon Braket 的事件。如果您未配置跟踪，您仍然可以在 CloudTrail 控制台的 “事件历史记录” 中查看最新的事件。通过收集的信息 CloudTrail，您可以确定向 Amazon Braket 发出的请求、发出请求的 IP 地址、谁提出了请求、何时提出请求以及其他详细信息。

要了解更多信息 CloudTrail，请参阅《AWS CloudTrail 用户指南》。

本节内容：

- Amazon Braket 中的信息 CloudTrail
- 了解 Amazon Braket 日志文件条目

Amazon Braket 中的信息 CloudTrail

CloudTrail 在您创建账户 AWS 账户 时已在您的账户上启用。当 Amazon Braket 中发生活动时，该活动会与其他 CloudTrail 事件一起记录在 AWS 服务 事件历史记录中。您可以在中查看、搜索和下载最近发生的事件 AWS 账户。有关更多信息，请参阅使用事件历史记录查看 CloudTrail 事件。

要持续记录您的事件 AWS 账户，包括 Amazon Braket 的事件，请创建跟踪。跟踪允许 CloudTrail 将日志文件传输到 Amazon S3 存储桶。默认情况下，在控制台中创建跟踪记录时，此跟踪记录应用于所有 AWS 区域。跟踪记录 AWS 分区中所有区域的事件，并将日志文件传送到您指定的 Amazon S3 存

储桶。此外，您可以配置其他 AWS 服务 ，以进一步分析和处理 CloudTrail 日志中收集的事件数据。有关更多信息，请参阅下列内容：

- 创建跟踪概述
- CloudTrail 支持的服务和集成
- 配置 Amazon SNS 通知 CloudTrail
- 接收来自多个区域的 CloudTrail 日志文件和接收来自多个账户的 CloudTrail 日志文件

所有 Amazon Braket 操作都由记录。 CloudTrail例如，对`GetQuantumTask`或`GetDevice`操作的调用会在 CloudTrail 日志文件中生成条目。

每个事件或日志条目都包含有关生成请求的人员信息。身份信息有助于您确定以下内容：

- 请求是使用角色还是联合用户的临时安全凭证发出的。
- 请求是否由其他 AWS 服务发出。

有关更多信息，请参阅 CloudTrail userIdentity 元素。

了解 Amazon Braket 日志文件条目

跟踪是一种配置，允许将事件作为日志文件传输到您指定的 Amazon S3 存储桶。 CloudTrail 日志文件包含一个或多个日志条目。事件代表来自任何来源的单个请求，包括有关请求的操作、操作的日期和时间、请求参数等的信息。 CloudTrail 日志文件不是公共API调用的有序堆栈跟踪，因此它们不会按任何特定的顺序出现。

以下示例是 `GetQuantumTask` 操作的日志条目，该条目获取了量子任务的详细信息。

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "foobar",
    "arn": "foobar",
    "accountId": "foobar",
    "accessKeyId": "foobar",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "foobar",
```

```
        "arn": "foobar",
        "accountId": "foobar",
        "userName": "foobar"
      },
      "webIdFederationData": {},
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2020-08-07T00:56:57Z"
      }
    }
  },
  "eventTime": "2020-08-07T01:00:08Z",
  "eventSource": "braket.amazonaws.com",
  "eventName": "GetQuantumTask",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "foobar",
  "userAgent": "aws-cli/1.18.110 Python/3.6.10
 Linux/4.9.184-0.1.ac.235.83.329.metal1.x86_64 botocore/1.17.33",
  "requestParameters": {
    "quantumTaskArn": "foobar"
  },
  "responseElements": null,
  "requestID": "20e8000c-29b8-4137-9cbc-af77d1dd12f7",
  "eventID": "4a2fdb22-a73d-414a-b30f-c0797c088f7c",
  "readOnly": true,
  "eventType": "AwsApiCall",
  "recipientAccountId": "foobar"
}
```

下图显示了 GetDevice 操作的日志条目，该条目返回了设备事件的详细信息。

```
{
  "eventVersion": "1.05",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "foobar",
    "arn": "foobar",
    "accountId": "foobar",
    "accessKeyId": "foobar",
    "sessionContext": {
      "sessionIssuer": {
        "type": "Role",
        "principalId": "foobar",
```

```
        "arn": "foobar",
        "accountId": "foobar",
        "userName": "foobar"
      },
      "webIdFederationData": {},
      "attributes": {
        "mfaAuthenticated": "false",
        "creationDate": "2020-08-07T00:46:29Z"
      }
    }
  },
  "eventTime": "2020-08-07T00:46:32Z",
  "eventSource": "braket.amazonaws.com",
  "eventName": "GetDevice",
  "awsRegion": "us-east-1",
  "sourceIPAddress": "foobar",
  "userAgent": "Boto3/1.14.33 Python/3.7.6 Linux/4.14.158-129.185.amzn2.x86_64 exec-
env/AWS_ECS_FARGATE Botocore/1.17.33",
  "errorCode": "404",
  "requestParameters": {
    "deviceArn": "foobar"
  },
  "responseElements": null,
  "requestID": "c614858b-4dcf-43bd-83c9-bcf9f17f522e",
  "eventID": "9642512a-478b-4e7b-9f34-75ba5a3408eb",
  "readOnly": true,
  "eventType": "AwsApiCall",
  "recipientAccountId": "foobar"
}
```

使用 Amazon Braket 进行高级日志记录

您可以使用记录器记录整个任务处理过程。借助这些高级日志记录技术，您可以查看后台轮询并创建记录供日后调试。

要使用记录器，我们建议更改 `poll_timeout_seconds` 和 `poll_interval_seconds` 参数，以便量子任务可以长时间运行，持续记录量子任务状态，并将结果保存到文件中。您可以将此代码传输到 Python 脚本而不是 Jupyter Notebook，这样该脚本就可以在后台作为进程运行了。

配置记录器

首先，配置记录器，使所有日志自动写入文本文件，如以下示例行所示。

```
# import the module
import logging
from datetime import datetime

# set filename for logs
log_file = 'device_logs-'+datetime.strftime(datetime.now(), '%Y%m%d%H%M%S')+'.txt'
print('Task info will be logged in:', log_file)

# create new logger object
logger = logging.getLogger("newLogger")

# configure to log to file device_logs.txt in the appending mode
logger.addHandler(logging.FileHandler(filename=log_file, mode='a'))

# add to file all log messages with level DEBUG or above
logger.setLevel(logging.DEBUG)
```

```
Task info will be logged in: device_logs-20200803203309.txt
```

创建并运行电路

现在，您可以创建一个电路，将其提交给设备运行，然后看看会发生什么，如本例所示。

```
# define circuit
circ_log = Circuit().rx(0, 0.15).ry(1, 0.2).rz(2, 0.25).h(3).cnot(control=0,
 target=2).zz(1, 3, 0.15).x(4)
print(circ_log)
# define backend
device = AwsDevice("arn:aws:braket:::device/quantum-simulator/amazon/sv1")
# define what info to log
logger.info(
    device.run(circ_log, s3_location,
               poll_timeout_seconds=1200, poll_interval_seconds=0.25, logger=logger,
 shots=1000)
    .result().measurement_counts
)
```

检查日志文件

您可以通过输入以下命令来检查写入文件的内容。

```
# print logs
```

```
! cat {log_file}
```

```
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: start polling for completion
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status CREATED
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status CREATED
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status QUEUED
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status RUNNING
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status RUNNING
Task arn:aws:braket:us-west-2:123412341234:quantum-
task/5088ec6c-89cf-4338-9750-9f5bb12a0dc4: task status COMPLETED
Counter({'00001': 493, '00011': 493, '01001': 5, '10111': 4, '01011': 3, '10101': 2})
```

从日志文件中获取 ARN

您可以从返回的日志文件输出中获取 ARN 信息，如上一个示例所示。使用 ARN ID，您可以检索已完成的量子任务的结果。

```
# parse log file for arn
with open(log_file) as openfile:
    for line in openfile:
        for part in line.split():
            if "arn:" in part:
                arn = part
                break
# remove final semicolon in logs
arn = arn[:-1]

# with this arn you can restore again task from unique arn
task_load = AwsQuantumTask(arn=arn, aws_session=AwsSession())

# get results of task
result = task_load.result()
```

Amazon Braket 限额

下表列出了 Amazon Braket 的服务配额。服务限额（也称为限制）是AWS 账户使用的服务资源或操作的最大数量。

某些配额可以增加。有关更多信息，请参阅 AWS 服务配额。

- 突发速率配额不能增加。
- 可调整配额（无法调整的突发速率除外）的最大速率增加是指定默认速率限值的 2 倍。例如，可将默认配额 60 调整为最大值 120。
- 并发 SV1 (DM1) 量子任务的可调整配额可使每个任务达到最多 60 个AWS 区域。
- 混合作业计算实例的最大允许数量为 1，配额是可以调整的。

资源	说明	限制	可调整
API 请求速率	在当前区域内的此账户中，您每秒可以发送的请求的最大数量。	140	是
API 请求的突发速率	在当前区域内的此账户中，您每秒可以在一次突增中发送的额外 请求的最大数量（RPS）。	600	否
CreateQuantumTask 请求速率	在每个区域的此账户中，您每秒可以发送的 CreateQuantumTask 请求的最大数量。	每秒 20 个	是
CreateQuantumTask 请求的突发速率	在当前区域内的此账户中，您每秒可以在一次突增中发送的额外 CreateQua	40	否

资源	说明	限制	可调整
	ntumTask 请求的最大数量（RPS）。		
SearchQuantumTasks 请求速率	在每个区域的此账户中，您每秒可以发送的 SearchQuantumTasks 请求的最大数量。	每秒 5 个	是
SearchQuantumTasks 请求的突发速率	在当前区域内的此账户中，您每秒可以在一次突增中发送的额外 SearchQuantumTasks 请求的最大数量（RPS）。	50	不可以
GetQuantumTask 请求速率	在每个区域的此账户中，您每秒可以发送的 GetQuantumTask 请求的最大数量。	每秒 100 个	是
GetQuantumTask 请求的突发速率	在当前区域内的此账户中，您每秒可以在一次突增中发送的额外 GetQuantumTask 请求的最大数量（RPS）。	500	否
CancelQuantumTask 请求速率	在每个区域的此账户中，您每秒可以发送的 CancelQuantumTask 请求的最大数量。	每秒 2 个	是

资源	说明	限制	可调整
CancelQuantumTask 请求的突发速率	在当前区域内的此账户中，您每秒可以在一次突增中发送的额外 CancelQuantumTask 请求的最大数量（RPS）。	20	否
GetDevice 请求速率	在每个区域的此账户中，您每秒可以发送的 GetDevice 请求的最大数量。	每秒 5 个	是
GetDevice 请求的突发速率	在当前区域内的此账户中，您每秒可以在一次突增中发送的额外 GetDevice 请求的最大数量（RPS）。	50	不可以
SearchDevices 请求速率	在每个区域的此账户中，您每秒可以发送的 SearchDevices 请求的最大数量。	每秒 5 个	是
SearchDevices 请求的突发速率	在当前区域内的此账户中，您每秒可以在一次突增中发送的额外 SearchDevices 请求的最大数量（RPS）。	50	不可以
CreateJob 请求速率	在每个区域的此账户中，您每秒可以发送的 CreateJob 请求的最大数量。	每秒 1 个	是

资源	说明	限制	可调整
CreateJob 请求的突发速率	在当前区域内的此账户中，您每秒可以在一次突增中发送的额外 CreateJob 请求的最大数量（RPS）。	5	否
SearchJobs 请求速率	在每个区域的此账户中，您每秒可以发送的 SearchJob 请求的最大数量。	每秒 5 个	是
SearchJobs 请求的突发速率	在当前区域内的此账户中，您每秒可以在一次突增中发送的额外 SearchJob 请求的最大数量（RPS）。	50	不可以
GetJob 请求速率	在每个区域的此账户中，您每秒可以发送的 GetJob 请求的最大数量。	每秒 5 个	是
GetJob 请求的突发速率	在当前区域内的此账户中，您每秒可以在一次突增中发送的额外 GetJob 请求的最大数量（RPS）。	25	否
CancelJob 请求速率	在每个区域的此账户中，您每秒可以发送的 CancelJob 请求的最大数量。	每秒 2 个	是

资源	说明	限制	可调整
CancelJob 请求的突发速率	在当前区域内的此账户中，您每秒可以在一次突增中发送的额外 CancelJob 请求的最大数量（RPS）。	5	否
并发 SV1 量子任务的数量	在当前区域内的状态向量模拟器（SV1）上运行的并发量子任务的最大数量。	100 us-east-1， 50 us-west-1， 100 us-west-2， 50 eu-west-2，	否
并发 DM1 任务的数量	在当前区域内的密度矩阵模拟器 (DM1) 上运行的并发量子任务的最大数量。	100 us-east-1， 50 us-west-1， 100 us-west-2， 50 eu-west-2，	否
并发 TN1 任务的数量	在当前区域内的张量网络模拟器（TN1）上运行的并发量子任务的最大数量。	10 us-east-1， 10 us-west-2， 5 eu-west-2，	是
并发混合作业的数量	当前区域内并发作业的最大数量。	3	是
混合作业运行时限制	混合作业可以运行的最长时间（以天为单位）。	5	否

下面是混合作业的默认经典计算实例限额。要提高这些配额，请联系 支持。此外，还会为每个实例指定可用区域。

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.c4.xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.c4.xlarge 类型实例数。	5	支持	是	是	是	是	否
用于混合任务的最大 ml.c4.2xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.c4.2xlarge 类型实例数。	5	支持	是	是	是	是	否
用于混合任务的最大 ml.c4.4xl	对于此账户和区域中的所有	5	支持	是	是	是	是	否

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
arge 实例数	Amazon Braket Hybrid Jobs，允许的最大 ml.c4.4xlarge 类型实例数。							
用于混合任务的最大 ml.c4.8xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.c4.8xlarge 类型实例数。	5	支持	是	是	是	否	否

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.c5.xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.c5.xlarge 类型实例数。	5	支持	是	是	是	是	是
用于混合任务的最大 ml.c5.2xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.c5.2xlarge 类型实例数。	5	支持	是	是	是	是	是

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.c5.4xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.c5.4xlarge 类型实例数。	1	是	是	是	是	是	是
用于混合任务的最大 ml.c5.9xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.c5.9xlarge 类型实例数。	1	是	是	是	是	是	是

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.c5.18xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.c5.18xlarge 类型实例数。	0	支持	是	是	是	是	是
用于混合任务的最大 ml.c5n.xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.c5n.xlarge 类型实例数。	0	支持	是	是	是	否	否

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.c5n.2xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.c5n.2xlarge 类型实例数。	0	支持	是	是	是	否	否
用于混合任务的最大 ml.c5n.4xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.c5n.4xlarge 类型实例数。	0	支持	是	是	是	否	否

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.c5n.9xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.c5n.9xlarge 类型实例数。	0	支持	是	是	是	否	否
用于混合任务的最大 ml.c5n.18xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.c5n.18xlarge 类型实例数。	0	支持	是	是	是	否	否

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.g4dn.xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.g4dn.xlarge 类型实例数。	0	支持	是	是	是	是	是
用于混合任务的最大 ml.g4dn.2xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.g4dn.2xlarge 类型实例数。	0	支持	是	是	是	是	是

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.g4dn.4xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.g4dn.4xlarge 类型实例数。	0	支持	是	是	是	是	是
用于混合任务的最大 ml.g4dn.8xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.g4dn.8xlarge 类型实例数。	0	支持	是	是	是	是	是

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.g4dn.12xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.g4dn.12xlarge 类型实例数。	0	支持	是	是	是	是	是
用于混合任务的最大 ml.g4dn.16xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.g4dn.16xlarge 类型实例数。	0	支持	是	是	是	是	是

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.m4.xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.m4.xlarge 类型实例数。	5	支持	是	是	是	是	否
用于混合任务的最大 ml.m4.2xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.m4.2xlarge 类型实例数。	5	支持	是	是	是	是	否

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.m4.4xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.m4.4xlarge 类型实例数。	2	是	是	是	是	是	否
用于混合任务的最大 ml.m4.10xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.m4.10xlarge 类型实例数。	0	支持	是	是	是	是	否

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.m4.16› large 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.m4.16› large 类型实例数。	0	支持	是	是	是	是	否
用于混合任务的最大 ml.m5.large 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.m5.large 类型实例数。	5	支持	是	是	是	是	是

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.m5.xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.m5.xlarge 类型实例数。	5	支持	是	是	是	是	是
用于混合任务的最大 ml.m5.2xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.m5.2xlarge 类型实例数。	5	支持	是	是	是	是	是

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.m5.4xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.m5.4xlarge 类型实例数。	5	支持	是	是	是	是	是
用于混合任务的最大 ml.m5.12xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.m5.12xlarge 类型实例数。	0	支持	是	是	是	是	是

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.m5.24› large 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.m5.24› large 类型实例数。	0	支持	是	是	是	是	是
用于混合任务的最大 ml.p2.xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.p2.xlarge 类型实例数。	0	支持	是	否	是	否	否

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.p2.8xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.p2.8xlarge 类型实例数。	0	支持	是	否	是	否	否
用于混合任务的最大 ml.p2.16xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.p2.16xlarge 类型实例数。	0	支持	是	否	是	否	否

资源	说明	限制	可调整	us-east-1	us-west-1	us-west-2	eu-west-2	eu-north-1
用于混合任务的最大 ml.p4d.24xlarge 实例数	对于此账户和区域中的所有 Amazon Braket Hybrid Jobs，允许的最大 ml.p4d.24xlarge 类型实例数。	0	支持	是	否	是	否	否

请求限额更新

如果您收到实例类型的 ServiceQuotaExceeded 例外情况，但没有足够的可用实例，则可以从AWS控制台的 “服务配额” 页面请求提高限制，然后在 “服务” 下AWS搜索 Amazon Braket。

> **Note**
>
> 如果您的混合作业无法预调配请求的 ML 计算容量，请使用其他区域。此外，如果您在表中看不到实例，则该实例不可用于混合作业。

其他配额和限制

- Amazon Braket 量子任务操作的大小限制在 3MB 以内。
- 对于 SV1，对于不超过 31 个量子比特的电路，最大运行时间为 3 小时，对于超过 31 个量子比特的电路，最大运行时间为 11 小时。
- SV1、DM1 和 Rigetti 设备允许的每个任务的最大拍摄次数为 5 万次。
- 每项任务允许的最大拍摄次数 TN1 为 1000。

- 对于AQT的IBEX-Q1设备，每项任务最多可拍摄 2000 张照片。
- 对于所有IonQ设备：每项任务的最低拍摄次数为 100。使用按需模式时，有 100 万次射门限制，错误缓解任务至少有 2500 次镜头。对于直接预留，没有门拍摄限制，错误缓解任务至少有 500 次拍摄。
- 对于 QuEra 的 Aquila 设备，每项任务的最大拍摄次数为 1000 次。
- 对于IQMGarnet和Emerald设备，每项任务最多可拍摄 20,000 张照片。
- 对于TN1和 QPU 设备，每项任务的镜头数必须大于 0。

《Amazon Braket 开发人员指南》文档历史记录

下表描述了 Amazon Braket 的文档版本。

- 最新API参考更新：2025 年 11 月 20 日
- 最新文档更新：2026 年 3 月 2 日

更改	描述	日期
ionQ Aria-1 设备停用	移除了对 ionQ Aria-1 设备的支持。	2026 年 3 月 2 日
更新 “处理预订” 页面	提高 “处理预订” 页面的清晰度	2026 年 2 月 3 日
支持 Braket 笔记本和托管容器的 Python 版本 3.12	为 Amazon Braket 笔记本和托管容器（Base、CUDA-Q PennyLane 和 Tensorflow）添加了 Python 版本 3.12 支持。包括 Python 3.12 升级的疑难解答指南。	2026年1月21日
移除 P3 示例	SageMaker 正在停用他们的`ml.p3`实例系列。将示例替换为推荐的`ml.g4dn`实例系列。	2025 年 12 月 19 日
新的支出限制功能	增加了对 Amazon Braket 支出限制功能的支持，该功能允许为自动验证和拒绝超过配置支出阈值 QPUs 的任务的个人设置可选的预算上限。	2025 年 11 月 20 日
全新 Braket 设备 AQT IBEX-Q1	增加了对AQT IBEX-Q1设备的支持。该设备基于位于超高真空室中的宏观射频陷阱中的 $^{40}Ca^{+}$ 个离子的晶体。	2025 年 11 月 18 日

亚马逊 Braket CUDA-Q 上提供了新的原生支持 NBIs	在 Amazon Braket Notebook 实例中添加了对 CUDA-Q 的原生支持。有关更多信息，请参阅中的 CUDA-Q。 NBIs	2025 年 11 月 10 日
IonQ Aria-2 设备停用	移除了对 IonQ Aria-2 设备的支持。	2025 年 10 月 27 日
合并了混合作业文档	合并了“混合作业”部分，显示在“使用 Amazon Braket Hybrid Jobs”下。	2025 年 10 月 21 日
新的 Braket 提供了 CUDA-Q 容器	添加了对所提供的 CUDA-Q 混合作业容器的支持。有关更多信息，请参阅为算法脚本定义环境。	2025 年 9 月 2 日
新的本地设备仿真器功能	添加了对本地量子设备仿真器工具的支持，可在将逐字记录程序提交给量子设备之前对其进行仿真。	2025 年 8 月 25 日
将 Pennylane 和 CUDA-Q 页面移到了“构建”部分	将 Pennylane 和 CUDA-Q 页面移到了目录的“构建”部分。	2025 年 8 月 15 日
新 `ProgramSet` 功能	添加了对程序集的支持，即在单个量子任务中运行多个量子电路的操作。	2025 年 8 月 14 日
新设备 IQM Emerald	添加了对 IQM Emerald 设备的支持。一款采用方形（水晶）晶格拓扑结构的 54 量子比特设备。	2025 年 7 月 21 日

更新了 AmazonBraketServiceRolePolicy政策	AmazonBraketServiceRolePolicy现在仅向 a ws: 提供 s3: * 和日志:* 操作PrincipalAccount。这限制了对请求者的存储桶和日志组的访问权限。	2025 年 7 月 11 日
新的实验能力特征：动态电路	中间电路测量和前馈操作可作为实验能力提供，参阅访问 IQM 设备上的动态电路。	2025 年 6 月 26 日
更新了 AmazonBraketFullAccess政策	AmazonBraketFullAccess现在包括定价：GetProducts在主机上显示硬件成本。	2025 年 4 月 14 日
新设备 IonQ Forte-Enterprise-1	添加了对 IonQ Forte-Enterprise-1 设备的支持。一种利用捕获离子技术的 36 量子比特设备。	2025 年 3 月 17 日
改进了 S3 条件权限	为提高安全性，`AmazonBraketFullAccess` 现在仅为 `aws:PrincipalAccount` 提供 `s3:*` 操作。这仅限制了请求者对自己存储桶的访问权限。	2025 年 3 月 7 日
新设备 Rigetti Ankaa-3	添加了对 Rigetti Ankaa-3 设备的支持。一款 84 量子比特设备，采用可扩展的多芯片技术。	2025 年 1 月 14 日
Rigetti Ankaa-2 设备停用	移除了对 Rigetti Ankaa-2 设备的支持。	2025 年 1 月 14 日
Support 对 IPv6 流量的支持	Amazon Braket 现在支持使用双堆栈终端节点的 IPv6 流量。`braket.{region}.api.aws`	2024 年 12 月 12 日

在 Amazon Braket 上为 NVIDIA's CUDA-Q 提供支持	客户现在可以在 Amazon Braket 上使用 NVIDIA's CUDA-Q 开发人员框架运行量子程序。	2024 年 12 月 6 日
IonQ Forte-1 设备随时可用	IonQ Forte-1 设备不再仅限于预留，现在可供我们的客户随时使用。	2024 年 11 月 22 日
Rigetti Aspen-M-3 设备停用	移除了对 Rigetti Aspen-M-3 设备的支持。	2024 年 9 月 27 日
IonQ Harmony 设备停用	移除了对 IonQ Harmony 设备的支持。	2024 年 8 月 29 日
新设备 Rigetti Ankaa-2	添加了对 Rigetti Ankaa-2 设备的支持。一款 84 量子比特设备，采用可扩展的多芯片技术。	2024 年 8 月 26 日
开发人员指南的重新组织	新的开发人员指南采用了现有的构建、测试、运行客户旅程，并引导用户通过 Amazon Braket 沿着这条路径前进。	2024 年 8 月 23 日
OQC Lucy 设备停用	移除了对 OQC Lucy 设备的支持。	2024 年 6 月 28 日
新设备 IQM Garnet 和区域 Europe North 1	添加了对 IQM Garnet 设备的支持。一款 20 量子比特设备，采用方格拓扑结构。将 Braket 支持的区域扩展到欧洲北部 1（斯德哥尔摩）。	2024 年 5 月 22 日
本地停机功能发布	实验功能现在包括 Aquila QPU QuEra 的本地失谐功能。	2024 年 4 月 11 日

Notebook 非活动管理器发布	创建 Notebook 实例时，启用不活动管理器并设置空闲持续时间以自动重置 Braket Notebook 实例。	2024 年 3 月 27 日
目录修订	重组了 Amazon Braket 目录，以遵守 AWS 风格指南要求并改善内容流以提高客户体验。	2023 年 12 月 12 日
Braket Direct 发布	添加了对 Braket Direct 功能的支持，包括： • 使用预留 • 获取专家建议 • 探索实验能力	2023 年 11 月 27 日
更新了 创建 Amazon Braket Notebook 实例	更新了文档，添加了为新老 Amazon Braket 客户创建 Notebook 实例的信息。	2023 年 11 月 27 日
更新了 使用自己的容器（BYOC）	更新了文档，添加了有关何时加入 BYOC、BYOC 配方以及在容器上运行 Braket Hybrid Jobs 的信息。	2023 年 10 月 18 日
混合作业修饰器发布	添加了 将本地代码作为混合作业运行 页面。包含以下示例： • 使用本地 Python 代码创建混合作业 • 安装其他 Python 软件包和源代码 • 将数据保存并加载到混合作业实例中 • 混合作业修饰器最佳实践	2023 年 10 月 16 日

添加了队列可见性	更新了《开发人员指南》文档，使其包含 queue depth 和 queue position。 更新了 API 文档，以反映关于队列可见性的新 API 更改。	2023 年 9 月 25 日
标准化文档中的命名	更新了文档，将“作业”的所有实例更改为“混合作业”，将“任务”更改为“量子任务”	2023 年 9 月 11 日
新设备 IonQ Aria 2	添加了对 IonQ Aria 2 设备的支持	2023 年 9 月 8 日
更新了原生门	更新了文档，添加了有关以编程方式从 Rigetti 对原生门进行访问的信息。	2023 年 8 月 16 日
Xanadu 离开	更新了文档，以移除所有 Xanadu 设备	2023 年 6 月 2 日
新设备 IonQ Aria	添加了对 IonQ Aria 设备的支持	2023 年 5 月 16 日
停用了 Rigetti 设备	停止了对 Rigetti Aspen-M-2 的支持	2023 年 5 月 2 日
更新的AmazonBraketFullAccess政策信息	更新了定义AmazonBraketFullAccess策略内容的脚本，使其包括 s ervicequotas: GetServiceQuota 和 cloudwatch: GetMetricData 操作以及有关配额限制的信息。	2023 年 4 月 19 日
导游旅程发布	更改了文档，以反映最新简化的 Braket 注册方法。	2023 年 4 月 5 日
新设备 Rigetti Aspen-M-3	添加了对 Rigetti Aspen-M-3 设备的支持	2023 年 1 月 17 日

新的伴随渐变功能	添加了有关 SV1 提供的伴随渐变功能的信息	2022 年 12 月 7 日
新的算法库功能	添加了有关 Braket 算法库的信息，该库提供了预先构建的量子算法目录	2022 年 11 月 28 日
D-Wave 离开	更新了文档，以适应所有 D-Wave 设备的移除情况	2022 年 11 月 17 日
新设备 QuEra Aquila	添加了对 QuEra Aquila 设备的支持	2022 年 10 月 31 日
支持 Braket Pulse	添加了对 Braket Pulse 的支持，可实现在 Rigetti 和 OQC 设备上使用脉冲控制	2022 年 10 月 20 日
支持 IonQ 原生门	添加了对 IonQ 设备提供的原生门设置的支持	2022 年 9 月 13 日
新实例配额	更新了与混合作业关联的默认经典计算实例配额	2022 年 8 月 22 日
新服务控制面板	更新了控制台屏幕截图，以包含服务控制面板	2022 年 8 月 17 日
新设备 Rigetti Aspen-M-2	添加了对 Rigetti Aspen-M-2 设备的支持	2022 年 8 月 12 日
OpenQASM 的新功能	添加了 OpenQASM 功能对本地模拟器（braket_sv 和 braket_dm）的支持	2022 年 8 月 4 日
新的成本跟踪程序	添加了如何获得模拟器和硬件工作负载的近乎实时的最大成本估算值	2022 年 7 月 18 日
新 Xanadu Borealis 设备	添加了对 Xanadu Borealis 设备的支持	2022 年 6 月 2 日

新的注册简化程序	添加了有关新的简化版注册程序如何运作的信息	2022 年 5 月 16 日
新设备 D-Wave Advantage _system6.1	添加了对 D-Wave Advantage _system6.1 设备的支持	2022 年 5 月 12 日
支持嵌入式模拟器	添加了如何使用混合作业运行嵌入式仿真以及如何使用 PennyLane 闪电模拟器	2022 年 5 月 4 日
AmazonBraketFullAccess -亚马逊 Braket 的完整访问政策	新增 s3：允许用户查看和检查为 Amazon Braket 创建和使用的存储桶的ListAllMyBuckets 权限	2022 年 3 月 31 日
支持 OpenQASM	为基于门的量子设备和模拟器添加了 OpenQASM 3.0 支持	2022 年 3 月 7 日
新的量子硬件提供商 Oxford Quantum Circuits 和新区域 eu-west-2	添加了对 OQC 和 eu-west-2 的支持	2022 年 2 月 28 日
新 Rigetti 设备	增加了对 Rigetti Aspen M-1 的支持	2022 年 2 月 15 日
新资源限制	将并发 DM1 和 SV1 任务的最大数量从 55 增加到 100	2022 年 1 月 5 日
新 Rigetti 设备	增加了对 Rigetti Aspen-11 的支持	2021 年 12 月 20 日
停用了 Rigetti 设备	已停止对 Rigetti Aspen-10 设备的支持	2021 年 12 月 20 日
新增结果类型	局部密度矩阵模拟器和 DM1 设备支持的低密度矩阵结果类型	2021 年 12 月 20 日

更新后的策略描述	Amazon Braket 更新了角色 ARN，使其包含 `servicerole/` 路径。有关策略更新的信息，请参阅 Amazon Braket AWS 托管式策略表更新信息。	2021 年 11 月 29 日
Amazon Braket 作业	添加了 Amazon Braket Hybrid Jobs 和 API 的用户指南	2021 年 11 月 29 日
新 Rigetti 设备	增加了对 Rigetti Aspen-10 的支持	2021 年 11 月 20 日
停用了 D-Wave 设备	停用了对 D-Wave QPU，Advantage_system1，的支持	2021 年 11 月 4 日
新 D-Wave 设备	添加了对额外 D-Wave QPU，Advantage_system4 的支持	2021 年 10 月 5 日
新噪声模拟器	添加了对密度矩阵模拟器 (DM1) 的支持，该模拟器最多可模拟多达 17 个 qubits 的电路及本地噪声模拟器 braket_dm	2021 年 5 月 25 日
PennyLane 支持	在 Amazon Brak PennyLane et 上增加了对的支持	2020 年 12 月 8 日
新模拟器	添加了对张量网络模拟器 (TN1) 的支持，它能实现更大的电路	2020 年 12 月 8 日
任务批处理	Braket 支持客户任务批处理	2020 年 11 月 24 日
手动 qubit 分配	Braket 支持在 Rigetti 设备上进行手动 qubit 分配	2020 年 11 月 24 日

可调整的配额	Braket 支持任务资源自助服务的可调整配额	2020 年 10 月 30 日
Support PrivateLink	您可以为 Braket 任务设置私有 VPC 端点	2020 年 10 月 30 日
支持标签	Braket 支持基于 API 的量子任务资源标签	2020 年 10 月 30 日
新 D-Wave 设备	添加了对额外 D-Wave QPU，Advantage_system1，的支持	2020 年 9 月 29 日
初始版本	Amazon Braket 文档首次发布	2020 年 8 月 12 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。