开发人员指南

AWS AppSync

AWS AppSync: 开发人员指南

Table of Contents

什么是 AWS AppSync？

AWS AppSync 允许开发人员通过安全、无服务器且高性能的 GraphQL 和 pub/sub API 将其应用程序和服务连接到数据和事件。您可以使用 AWS AppSync 执行以下操作：

- 从单个 GraphQL API 终端节点中访问一个或多个数据源的数据。
- 将多个源 GraphQL API 合并为一个合并的 GraphQL API。
- 将实时数据更新发布到您的应用程序。
- 利用内置安全性、监控、日志记录和跟踪，并使用可选的缓存以实现低延迟。
- 只需为 API 请求和传送的任何实时消息付费。

主题

AWS AppSync 功能

- 由 GraphQL 提供支持的简化数据访问和查询
- 用于 GraphQL 订阅和 pub/sub 通道的无服务器 WebSocket
- 服务器端缓存；在高速内存缓存中提供数据以实现低延迟
- 支持使用 JavaScript 和 TypeScript 编写业务逻辑
- 使用私有 API 限制 API 访问并与 AWS WAF 集成以提高企业安全性
- 内置授权控制，支持 API 密钥、IAM、Amazon Cognito、OpenID Connect 提供程序以及用于自定义逻辑的 Lambda 授权。
- 支持联合使用案例的合并 API

有关其中的每个功能的更多详细信息，请参阅 AWS AppSync 功能。

您是 AWS AppSync 新用户吗？

我们建议新 AWS AppSync 用户先阅读以下小节：

- 如果您不熟悉 GraphQL，请参阅入门：创建您的第一个 GraphQL API。
- 如果您要构建使用 GraphQL API 的应用程序，请参阅构建客户端应用程序和the section called “实时数据”。
- 如果您要查找 GraphQL 解析器信息，请参阅以下内容：

  JavaScript/TypeScript

  - 解析器教程 (JavaScript)
  - 解析器参考 (JavaScript)

  VTL

  - 解析器教程 (VTL)
  - 解析器映射模板参考 (VTL)
- 如果您要查找 AWS AppSync 示例项目、更新等，请参阅 AppSync 博客。

相关服务

如果您从头开始构建 Web 或移动应用程序，请考虑使用 AWS Amplify。Amplify 利用 AWS AppSync 和其他 AWS 服务，以帮助您以更少的工作量构建更稳健、更强大的 Web 和移动应用程序。

AWS AppSync 定价

AWS AppSync 定价基于数百万次请求和更新。缓存需要额外付费。有关更多信息，请参阅 AWS AppSync 定价。

下面列出了一般 AWS AppSync 定价的例外情况：

- AWS AppSync 中的 API 缓存不符合 AWS 免费套餐的条件。
- 对于授权和身份验证失败，不对请求计费。
- 如果 API 密钥缺失或无效，系统不会向需要 API 密钥的调用方法收费。

GraphQL 和 AWS AppSync 架构

> **Note**
>
> 本指南假设用户具有 REST 架构风格的实际经验。我们建议在使用 GraphQL 和 AWS AppSync 之前查看该指南和其他前端主题。

GraphQL 是一种 API 查询和处理语言。GraphQL 提供了灵活直观的语法以描述数据要求和交互。它使开发人员能够准确询问所需的内容并获得可预测的结果。它还可以在单个请求中访问多个源，以减少网络调用次数和带宽要求，从而节省电池使用寿命和应用程序使用的 CPU 周期。

通过变更使数据更新变得更简单，以使开发人员能够描述数据应如何发生变化。GraphQL 还有助于通过订阅快速设置实时解决方案。所有这些功能相结合再加上强大的开发人员工具，使 GraphQL 在管理应用程序数据方面变得不可或缺。

GraphQL 是 REST 的替代方案。RESTful 架构是目前比较流行的客户端-服务器通信解决方案之一。它以通过 URL 公开资源（数据）的概念为中心。可以使用这些 URL 通过 CRUD（创建、读取、更新、删除）操作访问和处理数据，这些操作采用 HTTP 方法的形式，例如 `GET`、`POST` 和 `DELETE`。REST 的优点是学习和实施相对简单。您可以快速设置 RESTful API 以调用多种服务。

不过，技术正变得越来越复杂。随着应用程序、工具和服务开始扩展到全球受众，对快速且可扩展的架构的需求变得至关重要。REST 在处理可扩展的操作时存在很多缺点。有关示例，请参阅该使用案例。

在以下几节中，我们将介绍一些有关 RESTful API 的概念。然后，我们介绍 GraphQL 及其工作方式。

有关 GraphQL 以及迁移到 AWS 的好处的更多信息，请参阅 Decision guide to GraphQL implementations。

主题

什么是 API？

应用程序编程接口 (API) 定义与其他软件系统通信时必须遵循的规则。开发人员公开或创建 API，以使其他应用程序可以按编程方式与其应用程序进行通信。例如，工时表应用程序公开一个 API，要求提供员工的全名和日期范围。在它收到该信息时，它在内部处理员工的工时表，并返回该日期范围内的工作小时数。

您可以将 Web API 视为客户端和 Web 上的资源之间的网关。

客户端

客户端是希望从 Web 中访问信息的用户。客户端可以是一个人，也可以是一个使用 API 的软件系统。例如，开发人员可以编写程序，从天气系统中访问天气数据。或者，在您直接访问天气网站时，您可以从浏览器中访问相同的数据。

资源

资源是不同应用程序向其客户端提供的信息。资源可以是图像、视频、文本、数字或任何类型的数据。向客户端提供资源的计算机也称为服务器。组织使用 API 共享资源并提供 Web 服务，同时保持安全性、控制和身份验证。此外，API 还帮助他们确定哪些客户端可以访问特定的内部资源。

什么是 REST？

概括地说，表述性状态传输 (REST) 是一种软件架构，它对 API 的工作方式施加了条件。REST 最初是作为管理互联网等复杂网络上的通信的准则创建的。您可以使用基于 REST 的架构支持大规模的高性能且可靠的通信。您可以轻松进行实施和修改，从而为任何 API 系统提供可见性和跨平台可移植性。

API 开发人员可以使用多种不同的架构设计 API。遵循 REST 架构风格的 API 称为 REST API。实施 REST 架构的 Web 服务称为 RESTful Web 服务。RESTful API 术语通常是指 RESTful Web API。不过，您可以将 REST API 和 RESTful API 术语互换使用。

以下是 REST 架构风格的一些准则：

统一接口

统一接口是任何 RESTful Web 服务设计的基础。它表示服务器以标准格式传输信息。设置了格式的资源在 REST 中称为表示形式。该格式可能与服务器应用程序上的资源的内部表示形式不同。例如，服务器可以将数据存储为文本，但以 HTML 表示形式发送。

统一接口施加了 4 个架构限制：

1. 请求应标识资源。它们使用统一资源标识符以实现该目的。
2. 客户端在资源表示形式中具有足够多的信息，可以根据需要修改或删除资源。服务器发送进一步描述资源的元数据以满足该条件。
3. 客户端收到有关如何进一步处理表示形式的信息。服务器发送自描述消息以实现该目的，这些消息包含有关客户端如何以最佳方式使用它们的元数据。
4. 客户端收到完成任务所需的所有其他相关资源的信息。服务器在表示形式中发送超链接以实现该目的，以使客户端可以动态发现更多资源。

无状态

在 REST 架构中，无状态是指服务器独立于所有以前请求完成每个客户端请求的通信方法。客户端可以按任意顺序请求资源，并且每个请求都是无状态或与其他请求隔离。这种 REST API 设计限制意味着，服务器每次都能完全了解并完成请求。

分层系统

在分层系统架构中，客户端可以连接到客户端和服务器之间的其他授权中间设备，并且仍然会接收来自服务器的响应。服务器也可以将请求传送到其他服务器。您可以将 RESTful Web 服务设计为在具有多个层（例如安全性、应用程序和业务逻辑）的多个服务器上运行，这些层相互协作以完成客户端请求。这些层对客户端是不可见的。

可缓存性

RESTful Web 服务支持缓存，这是在客户端或中间设备上存储一些响应以缩短服务器响应时间的过程。例如，假设您访问的网站的每个页面上具有通用的页眉和页脚图像。每次您访问新的网站页面时，服务器都必须重新发送相同的图像。为了避免这种情况，客户端在第一次响应后缓存或存储这些图像，然后直接使用缓存中的图像。RESTful Web 服务使用将自身定义为可缓存或不可缓存的 API 响应以控制缓存。

什么是 RESTful API？

RESTful API 是两个计算机系统用于通过互联网安全地交换信息的接口。大多数业务应用程序必须与其他内部和第三方应用程序通信以执行各种任务。例如，要生成每月工资单，您的内部账户系统必须与客户的银行系统共享数据，以自动开具发票并与内部工时表应用程序进行通信。RESTful API 支持这种信息交换，因为它们遵循安全可靠且高效的软件通信标准。

RESTful API 是如何工作的？

RESTful API 的基本功能与浏览互联网相同。在客户端需要资源时，它使用 API 连接到服务器。API 开发人员在服务器应用程序 API 文档中解释了客户端应如何使用 REST API。以下是任何 REST API 调用的一般步骤：

1. 客户端向服务器发送请求。客户端按照 API 文档以服务器理解的方式设置请求格式。
2. 服务器对客户端进行身份验证，并确认客户端有权发出该请求。
3. 服务器接收该请求，并在内部进行处理。
4. 服务器向客户端返回响应。响应包含告诉客户端请求是否成功的信息。响应还包含客户端请求的任何信息。

根据 API 开发人员设计 API 的方式，REST API 请求和响应详细信息略有不同。

为什么使用 GraphQL 而不是 REST？

REST 是 Web API 的基石架构风格之一。不过，随着世界的互联程度变得越来越高，开发稳健且可扩展的应用程序的需求成为更紧迫的问题。虽然 REST 目前是构建 Web API 的行业标准，但已发现 RESTful 实施存在几个反复出现的缺点：

1. 数据请求：在使用 RESTful API 时，您通常通过终端节点请求所需的数据。在您的数据可能没有那么整齐打包时，就会出现问题。您需要的数据可能位于多个抽象层后面，获取数据的唯一方法是使用多个终端节点，这意味着发出多个请求以获取所有数据。
2. 过度获取和获取不足：除了多个请求的问题以外，来自每个终端节点的数据都是严格定义的，这意味着您返回为该 API 定义的任何数据，即使您在技术上并不需要这些数据。

   这可能会导致过度获取，这意味着我们的请求返回多余的数据。例如，假设您请求公司个人数据，并希望知道某个部门的员工姓名。返回数据的终端节点将包含姓名，但也可能包含其他数据，例如职位或出生日期。由于 API 是固定的，因此，您无法只请求姓名本身；其余数据也会随之而来。

   相反的情况是，我们没有返回足够的数据，这称为获取不足。要获取请求的所有数据，您可能需要向服务发出多个请求。根据设置数据结构的方式，您可能会遇到效率低下的查询，从而导致类似于令人头痛的 n+1 问题的情况。
3. 开发迭代速度缓慢：很多开发人员定制其 RESTful API 以适应他们的应用程序流程。不过，随着应用程序的扩展，前端和后端可能需要进行大量更改。因此，API 可能不再以高效或有影响力的方式适应数据的形状。由于需要修改 API，这会导致产品迭代速度变慢。

4. 大规模性能：由于这些复杂的问题，很多领域的可扩展性都会受到影响。应用程序端的性能可能会受到影响，因为您的请求返回太多的数据或太少的数据（导致更多请求）。这两种情况都会对网络造成不必要的压力，从而导致性能下降。在开发人员方面，开发速度可能会下降，因为您的 API 是固定的，不再适合他们请求的数据。

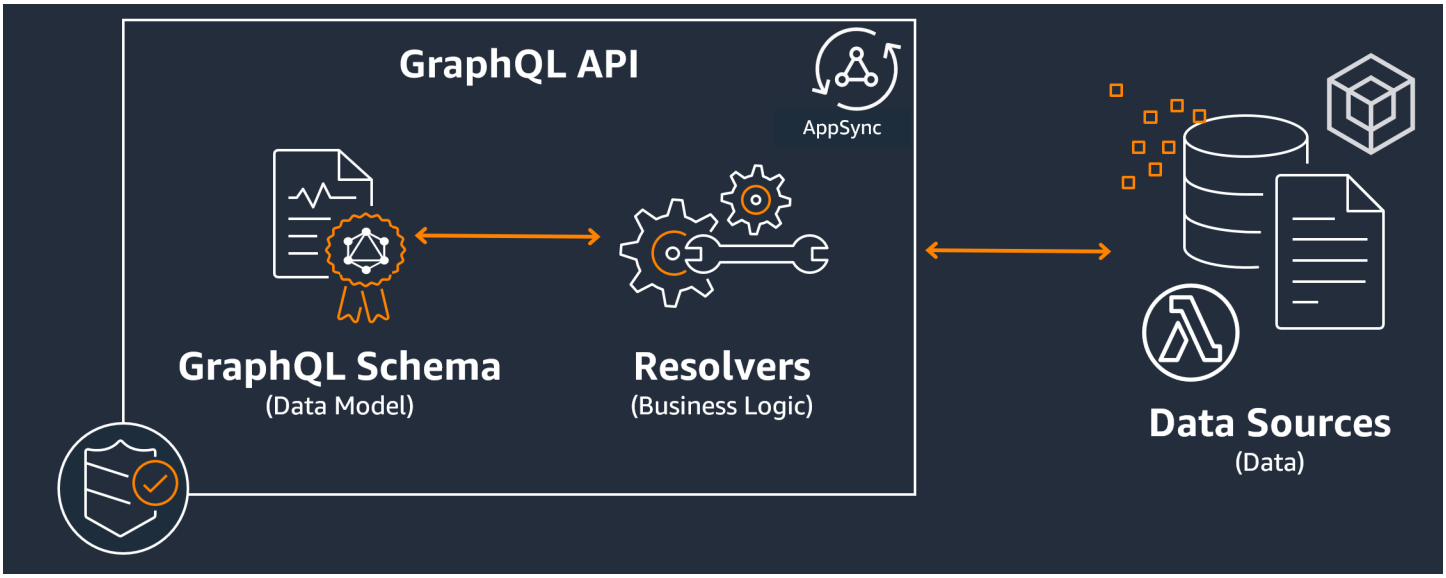
GraphQL 的卖点是克服 REST 的缺点。以下是 GraphQL 为开发人员提供的一些关键解决方案：

1. 单个终端节点：GraphQL 使用单个终端节点查询数据。无需构建多个 API 以适应数据的形状。这会导致通过网络传输的请求减少。
2. 获取：GraphQL 直接定义所需的数据以解决长期存在的过度获取和获取不足问题。GraphQL 允许您设置数据形状以满足您的需求，从而仅收到所要求的内容。
3. 抽象：GraphQL API 包含一些使用与语言无关的标准描述数据的组件和系统。换句话说，数据形状和结构是标准化的，因此，前端和后端知道如何通过网络发送数据。这使两端的开发人员可以使用 GraphQL 的系统，而不是围绕它们工作。
4. 快速迭代：由于数据标准化，在开发的一端进行更改时，可能不需要在另一端进行更改。例如，前端表示形式更改可能不会导致在后端进行大量更改，因为 GraphQL 可以轻松修改数据规范。您可以直接定义或修改数据的形状，以满足应用程序扩展时的需求。这会导致潜在的开发工作减少。

这些只是 GraphQL 的一部分好处。在接下来的几节中，您将了解如何设置 GraphQL 结构以及使其成为 REST 的独特替代方案的属性。

GraphQL API 组件

标准 GraphQL API 由单个架构组成，该架构处理查询的数据的形状。您的架构链接到一个或多个数据源，例如数据库或 Lambda 函数。在它们之间具有一个或多个解析器，用于处理您的请求的业务逻辑。每个组件在 GraphQL 实施中都发挥着重要作用。以下几节介绍了这三个组件以及它们在 GraphQL 服务中发挥的作用。



主题

架构

GraphQL 架构是 GraphQL API 的基础。它充当定义数据形状的蓝图。它也是客户端和服务器之间的合约，用于定义如何检索和/或修改您的数据。

GraphQL 架构是使用架构定义语言 (SDL) 编写的。SDL 由具有既定结构的类型和字段组成：

- 类型：类型是 GraphQL 定义数据形状和行为的方式。GraphQL 支持多种类型，本节后面将介绍这些类型。架构中定义的每种类型将包含自己的范围。在该范围内具有一个或多个字段，这些字段可以包含在 GraphQL 服务中使用的值或逻辑。类型扮演很多不同的角色，最常见的角色是对象或标量（基元值类型）。
- 字段：字段位于类型范围内，并保存从 GraphQL 服务中请求的值。它们与其他编程语言中的变量非常相似。您在字段中定义的数据的形状决定了如何在请求/响应操作中设置数据结构。这样，开发人员就可以在不知道服务后端实施方式的情况下预测返回的内容。

为了直观地了解架构的外观，让我们看一个简单 GraphQL 架构的内容。在生产代码中，您的架构通常位于名为 `schema.graphql` 或 `schema.json` 的文件中。假设我们正在研究一个实施 GraphQL 服务

的项目。该项目存储公司人员数据，并使用 schema.graphql 文件检索人员数据以及在数据库中添加新人员。代码可能如下所示：

schema.graphql

```
type Person {
   id: ID!
   name: String
   age: Int
}
type Query {
  people: [Person]
}
type Mutation {
  addPerson(id: ID!, name: String, age: Int): Person
}
```

我们可以看到在架构中定义了三种类型：Person、Query 和 Mutation。看一下 Person，我们可以猜到这是公司员工实例的蓝图，这会使该类型成为一个对象。在其范围内，我们看到 id、name 和 age。这些是定义 Person 属性的字段。这意味着我们的数据源将每个 Person 的 name 存储为 String 标量（基元）类型，并将 age 存储为 Int 标量（基元）类型。id 充当每个 Person 的特殊唯一标识符。它也是一个必需的值，由！符号表示。

接下来的两个对象类型的行为有所不同。GraphQL 为特殊对象类型保留一些关键字，这些关键字定义如何在架构中填充数据。Query 类型从源中检索数据。在我们的示例中，我们的查询可能从数据库中检索 Person 对象。这可能会让您想起 RESTful 术语中的 GET 操作。Mutation 修改数据。在我们的示例中，我们的变更可能会在数据库中添加更多 Person 对象。这可能会让您想起 PUT 或 POST 等状态更改操作。本节稍后将介绍所有特殊对象类型的行为。

让我们假设示例中的 Query 从数据库中检索一些内容。如果我们查看 Query 的字段，就会看到一个名为 people 的字段。其字段值为 [Person]。这意味着我们要检索数据库中的某个 Person 实例。不过，添加方括号意味着，我们希望返回所有 Person 实例的列表，而不仅仅是一个特定的实例。

Mutation 类型负责执行状态更改操作，例如数据修改。变更负责对数据源执行一些状态更改操作。在我们的示例中，变更包含一个名为 addPerson 的操作，它将新的 Person 对象添加到数据库中。该变更使用 Person 并需要使用 id、name 和 age 字段的输入。

此时，您可能想知道 addPerson 等操作在没有代码实施的情况下如何工作，因为它应该实施某种行为，并且看起来很像具有函数名称和参数的函数。目前，它不起作用，因为架构仅充当声明。要实施 addPerson 行为，我们必须为其添加一个解析器。解析器是一个代码单元，只要调用其关联字段（此

处的 addPerson 操作），就会执行该代码单元。如果要使用某个操作，您必须在某个时候添加解析器实施。从某种意义上说，您可以将架构操作视为函数声明，并将解析器视为定义。将在另一个小节中介绍解析器。

该示例仅显示架构处理数据的最简单方法。您可以使用 GraphQL 和 AWS AppSync 功能构建复杂、稳健且可扩展的应用程序。在下一节中，我们将定义您可以在架构中使用的所有不同类型和字段行为。

GraphQL 类型

GraphQL 支持很多不同的类型。正如您在上一节中看到的一样，类型定义数据的形状或行为。它们是 GraphQL 架构的基本构建块。

类型可以分为输入和输出。输入是允许作为特殊对象类型（Query、Mutation 等）的参数传入的类型，而输出类型严格用于存储和返回数据。下面列出了类型及其分类的列表：

- 对象：对象包含描述实体的字段。例如，一个对象可能类似于 book，其中包含描述其特性的字段，如 authorName、publishingYear 等。它们严格来说是输出类型。
- 标量：这些是基元类型，如整数、字符串等。它们通常分配给字段。以 authorName 字段为例，可以为其分配 String 标量以存储名称，例如“John Smith”。标量可以是输入类型和输出类型。
- 输入：输入允许您传递一组字段以作为参数。它们的结构与对象非常相似，但可以将其作为特殊对象的参数传递。输入允许您在其范围内定义标量、枚举和其他输入。输入只能是输入类型。
- 特殊对象：特殊对象执行状态更改操作，并完成服务的大部分繁重工作。共有三种特殊对象类型：查询、变更和订阅。查询通常获取数据；变更处理数据；订阅在客户端和服务器之间打开并保持双向连接以进行持续通信。鉴于其功能，特殊对象既不是输入，也不是输出。
- 枚举：枚举是预定义的合法值列表。如果调用枚举，则枚举值只能是其范围内定义的值。例如，如果您使用一个名为 trafficLights 的枚举以描述交通信号列表，它可能具有 redLight 和 greenLight 等值，但不能具有 purpleLight 值。真正的交通信号灯只有那么多信号，因此，您可以使用枚举定义它们，并在引用 trafficLight 时强制使它们成为唯一的合法值。枚举可以是输入类型和输出类型。
- 联合/接口：联合允许您根据客户端请求的数据在请求中返回一个或多个内容。例如，如果您使用具有 title 字段的 Book 类型以及具有 name 字段的 Author 类型，您可以在这两种类型之间创建联合。如果您的客户端希望在数据库中查询“Julius Caesar”，则联合可能会从 Book title 返回 Julius Caesar（威廉·莎士比亚的戏剧），并从 Author name 返回 Julius Caesar（Commentarii de Bello Gallico 的作者）。联合只能是输出类型。

  接口是对象必须实施的字段集。这有点类似于 Java 等编程语言中的接口，您必须实施接口中定义的字段。例如，假设您创建了一个名为 Book 的接口，其中包含一个 title 字段。假设您后来创建了

一个名为 Novel 的类型，该类型实施 Book。Novel 必须包含一个 title 字段。不过，Novel 可能还包含接口中不包含的其他字段，例如 ISBN 的 pageCount。接口只能是输出类型。

以下几节介绍了每种类型在 GraphQL 中的工作方式。

Objects

GraphQL 对象是您在生产代码中看到的主要类型。在 GraphQL 中，您可以将对象视为不同字段的分组（类似于其他语言中的变量），每个字段由可以保存值的类型（通常是标量或另一个对象）定义。对象表示可以从服务实施中检索/处理的数据单元。

对象类型是使用 Type 关键字声明的。让我们稍微修改一下架构示例：

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}
```

此处的对象类型是 Person 和 Occupation。每个对象具有自己的字段和自己的类型。GraphQL 的一项功能是，能够将字段设置为其他类型。您可以看到 Person 中的 occupation 字段包含 Occupation 对象类型。我们可以建立这种关联，因为 GraphQL 仅描述数据，而不描述服务实施。

标量

标量本质上是保存值的基元类型。在 AWS AppSync 中，具有两种类型的标量：默认 GraphQL 标量和 AWS AppSync 标量。标量通常用于存储对象类型中的字段值。默认 GraphQL 类型包括 Int、Float、String、Boolean 和 ID。让我们再次使用上一示例：

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}
```

```
type Occupation {
  title: String
}
```

挑出 name 和 title 字段，两者都包含 String 标量。Name 可能返回类似于 "John Smith" 的字符串值，title 可能返回类似于 "firefighter" 的值。一些 GraphQL 实施还支持使用 Scalar 关键字并实施类型行为的自定义标量。不过，AWS AppSync 目前不支持自定义标量。有关标量的列表，请参阅 AWS AppSync 中的标量类型。

输入

由于输入和输出类型概念，在传入参数时存在特定的限制。通常需要传入的类型（尤其是对象）受到限制。您可以使用输入类型绕过该规则。输入是包含标量、枚举和其他输入类型的类型。

输入是使用 input 关键字定义的：

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}

input personInput {
  id: ID!
  name: String
  age: Int
  occupation: occupationInput
}

input occupationInput {
  title: String
}
```

正如您看到的一样，我们可以使用模仿原始类型的单独输入。这些输入通常在您的字段操作中使用，如下所示：

```
type Person {
```

```
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}

input occupationInput {
  title: String
}

type Mutation {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person
}
```

请注意，我们仍然可以传递 occupationInput（替代 Occupation）以创建 Person。

这只是输入的一种场景。它们不一定需要以 1:1 的方式复制对象；在生产代码中，您很可能不会像这样使用该参数。一种利用 GraphQL 架构的很好做法是，仅定义需要作为参数输入的内容。

此外，可以在多个操作中使用相同的输入，但我们不建议这样做。理想情况下，每个操作应包含自己的唯一输入副本，以防架构的要求发生变化。

特殊对象

GraphQL 为特殊对象保留一些关键字，这些对象定义架构如何检索/处理数据的一些业务逻辑。最多可以在架构中包含其中的关键字之一。它们充当客户端对 GraphQL 服务运行的所有请求的数据的入口点。

也可以使用 type 关键字定义特殊对象。尽管它们的使用方式与常规对象类型不同，但它们的实施非常相似。

Queries

查询与 GET 操作非常相似，因为它们执行只读获取以从源中获取数据。在 GraphQL 中，Query 定义向您的服务器发出请求的客户端的所有入口点。在您的 GraphQL 实施中始终具有一个 Query。

以下是我们在以前的架构示例中使用的 Query 和修改的对象类型：

```
type Person {
```

```
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}
type Occupation {
  title: String
}
type Query {
  people: [Person]
}
```

我们的 Query 包含一个名为 people 的字段，该字段从数据源中返回 Person 实例列表。假设我们需要更改应用程序的行为，现在我们需要返回仅包含 Occupation 实例的列表以用于某些单独的用途。我们可以直接将其添加到查询中：

```
type Query {
  people: [Person]
  occupations: [Occupation]
}
```

在 GraphQL 中，我们可以将查询视为单一请求来源。正如您看到的一样，这可能比使用不同终端节点（.../api/1/people 和 .../api/1/occupations）实现相同目标的 RESTful 实施简单得多。

假设我们具有该查询的解析器实施，我们现在可以执行实际的查询。虽然 Query 类型存在，但我们必须明确调用该类型，才能在应用程序的代码中运行。可以使用 query 关键字完成该操作：

```
query getItems {
   people {
      name
   }
   occupations {
      title
   }
}
```

正如您看到的一样，该查询命名为 getItems 并返回 people（Person 对象列表）和 occupations（Occupation 对象列表）。在 people 中，我们仅返回每个 Person 的 name 字段，同时返回每个 Occupation 的 title 字段。响应可能如下所示：

```
{
  "data": {
    "people": [
      {
        "name": "John Smith"
      },
      {
        "name": "Andrew Miller"
      },
      .
      .
      .
    ],
    "occupations": [
      {
        "title": "Firefighter"
      },
      {
        "title": "Bookkeeper"
      },
      .
      .
      .
    ]
  }
}
```

该示例响应说明了数据如何遵循查询的形状。检索的每个条目在该字段的范围内列出。people 和 occupations 是作为单独的列表返回的。虽然这很有用，但修改查询以返回人员姓名和职业的列表可能会更方便：

```
query getItems {
   people {
      name
      occupation {
        title
      }
}
```

这是合法的修改，因为我们的 Person 类型包含 Occupation 类型的 occupation 字段。在 people 范围内列出时，我们将返回每个 Person 的 name 及其按 title 关联的 Occupation。响应可能如下所示：

```
}
  "data": {
    "people": [
      {
        "name": "John Smith",
        "occupation": {
          "title": "Firefighter"
        }
      },
      {
        "name": "Andrew Miller",
        "occupation": {
          "title": "Bookkeeper"
        }
      },
      .
      .
      .
    ]
  }
}
```

Mutations

变更类似于 PUT 或 POST 等状态更改操作。它们执行写入操作以修改源中的数据，然后获取响应。它们定义数据修改请求的入口点。与查询不同，根据项目的需求，变更可能会包含在架构中，也可能不会包含在架构中。以下是架构示例中的变更：

```
type Mutation {
  addPerson(id: ID!, name: String, age: Int): Person
}
```

addPerson 字段表示将 Person 添加到数据源的一个入口点。addPerson 是字段名称；id、name 和 age 是参数；Person 是返回类型。回顾一下 Person 类型：

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}
```

我们添加了 occupation 字段。不过，我们不能直接将该字段设置为 Occupation，因为不能将对象作为参数传入；它们严格来说是输出类型。我们应该将具有相同字段的输入作为参数传递：

```
input occupationInput {
  title: String
}
```

在创建新的 Person 实例时，我们也可以轻松更新 addPerson 以将其包含为参数：

```
type Mutation {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person
}
```

以下是更新的架构：

```
type Person {
  id: ID!
  name: String
  age: Int
  occupation: Occupation
}

type Occupation {
  title: String
}

input occupationInput {
  title: String
}

type Mutation {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput): Person
}
```

请注意，occupation 从 occupationInput 传入 title 字段，以完成创建 Person 而不是原始 Occupation 对象。假设我们具有 addPerson 的解析器实施，我们现在可以执行实际的变更。虽然 Mutation 类型存在，但我们必须明确调用该类型，才能在应用程序的代码中运行。可以使用 mutation 关键字完成该操作：

```
mutation createPerson {
  addPerson(id: ID!, name: String, age: Int, occupation: occupationInput) {
```

```
    name
    age
    occupation {
      title
    }
  }
}
```

该变更命名为 createPerson，addPerson 是操作。要创建新的 Person，我们可以输入 id、name、age 和 occupation 的参数。在 addPerson 的范围内，我们还可以看到其他字段，如 name、age 等。这是您的响应；这些是 addPerson 操作完成后返回的字段。以下是示例的最后一部分：

```
mutation createPerson {
  addPerson(id: "1", name: "Steve Powers", age: "50", occupation: "Miner") {
    id
    name
    age
    occupation {
      title
    }
  }
}
```

在使用该变更时，结果可能如下所示：

```
{
  "data": {
    "addPerson": {
      "id": "1",
      "name": "Steve Powers",
      "age": "50",
      "occupation": {
        "title": "Miner"
      }
    }
  }
}
```

正如您看到的一样，响应使用与变更中定义的相同格式返回我们请求的值。最好返回所有修改的值，以减少混乱以及将来需要进行更多查询。变更允许您在其范围内包含多个操作。它们按照变更

中列出的顺序依次运行。例如，如果我们创建另一个名为 addOccupation 的操作以将职务添加到数据源中，我们可以在变更中的 addPerson 之后调用该操作。先处理 addPerson，然后处理 addOccupation。

Subscriptions

订阅使用 WebSockets 在服务器与其客户端之间打开持久的双向连接。通常，客户端订阅或侦听服务器。每次服务器进行服务器端更改或执行事件时，订阅的客户端都会收到更新。如果订阅了多个客户端，并且需要向它们通知服务器或其他客户端中发生的更改，这种类型的协议是非常有用的。例如，可以使用订阅更新社交媒体源。可能具有两个用户（用户 A 和用户 B），他们订阅了自动通知更新，以在每次收到直接消息时收到通知。客户端 A 上的用户 A 可能向客户端 B 上的用户 B 发送直接消息。用户 A 的客户端将发送直接消息，而服务器处理该消息。然后，服务器向用户 B 的账户发送直接消息，同时向客户端 B 发送自动通知。

以下是我们可以添加到架构示例的 Subscription 示例：

```
type Subscription {
  personAdded: Person
}
```

每次将新的 Person 添加到数据源时，personAdded 字段都会向订阅的客户端发送消息。假设我们具有 personAdded 的解析器实施，我们现在可以使用订阅。虽然 Subscription 类型存在，但我们必须明确调用该类型，才能在应用程序的代码中运行。可以使用 subscription 关键字完成该操作：

```
subscription personAddedOperation {
  personAdded {
    id
    name
  }
}
```

订阅命名为 personAddedOperation，操作为 personAdded。personAdded 将返回新 Person 实例的 id 和 name 字段。看一下变更示例，我们使用该操作添加了一个 Person：

```
addPerson(id: "1", name: "Steve Powers", age: "50", occupation: "Miner")
```

如果我们的客户端订阅了新添加的 Person 的更新，它们可能会在 addPerson 运行后看到这一点：

```
{
  "data": {
    "personAdded": {
      "id": "1",
      "name": "Steve Powers"
    }
  }
}
```

以下是订阅提供的内容摘要：

订阅是双向通道，允许客户端和服务器接收快速但稳定的更新。它们通常使用 WebSocket 协议，该协议创建标准化的安全连接。

订阅非常灵活，因为它们减少了连接建立开销。在订阅后，客户端可以在较长时间内持续运行该订阅。它们通常允许开发人员定制订阅生命周期并配置请求哪些信息，从而高效地使用计算资源。

一般来说，订阅允许客户端一次进行多个订阅。由于与 AWS AppSync 相关，订阅仅用于从 AWS AppSync 服务接收实时更新。它们不能用于执行查询或变更。

订阅的主要替代方案是轮询，后者按设置的间隔发送查询以请求数据。该过程通常比订阅效率低，并且给客户端和后端带来很大压力。

我们的架构示例中没有提到的一件事是，还必须在 schema 根中定义您的特殊对象类型。因此，当您在 AWS AppSync 中导出架构时，它可能如下所示：

schema.graphql

```
schema {
  query: Query
  mutation: Mutation
  subscription: Subscription
}

.
.
.

type Query {
  # code goes here
}
type Mutation {
```

```
  # code goes here
}
type Subscription {
  # code goes here
}
```

枚举

枚举是特殊标量，它限制类型或字段可能具有的合法参数。这意味着每次在架构中定义枚举时，其关联类型或字段仅限于枚举中的值。枚举被序列化为字符串标量。请注意，不同的编程语言可能以不同的方式处理 GraphQL 枚举。例如，JavaScript 没有本机枚举支持，因此，枚举值可能会映射到整数值。

枚举是使用 enum 关键字定义的。示例如下：

```
enum trafficSignals {
  solidRed
  solidYellow
  solidGreen
  greenArrowLeft
  ...
}
```

在调用 trafficLights 枚举时，参数只能是 solidRed、solidYellow、solidGreen 等。通常使用枚举描述具有不同但有限数量的选项的内容。

联合/接口

请参阅 GraphQL 中的接口和联合。

GraphQL 字段

字段位于类型范围内，并保存从 GraphQL 服务中请求的值。它们与其他编程语言中的变量非常相似。例如，以下是一个 Person 对象类型：

```
type Person {
   name: String
   age: Int
}
```

此处的字段为 name 和 age，分别保存 String 和 Int 值。可以将像上面所示的对象字段作为查询和变更字段（操作）中的输入。例如，请参阅下面的 Query：

```
type Query {
  people: [Person]
}
```

people 字段从数据源中请求所有 Person 实例。在 GraphQL 服务器中添加或检索 Person 时，您可以要求数据采用您的类型和字段的格式，即，架构中的数据结构决定了如何在响应中设置其结构：

```
}
  "data": {
    "people": [
      {
        "name": "John Smith",
        "age": "50"
      },
      {
        "name": "Andrew Miller",
        "age": "60"
      },
      .
      .
      .
    ]
  }
}
```

字段在设置数据结构方面发挥着重要作用。下面介绍了几个额外的属性，可以将其应用于字段以进行更多自定义。

列表

列表返回指定类型的所有项目。可以使用方括号 [] 将列表添加到字段类型中：

```
type Person {
  name: String
  age: Int
}
type Query {
  people: [Person]
}
```

在 Query 中，Person 两侧的方括号表示您希望以数组形式从数据源返回所有 Person 实例。在响应中，每个 Person 的 name 和 age 值作为单个分隔列表返回：

```
}
  "data": {
    "people": [
      {
        "name": "John Smith",         # Data of Person 1
        "age": "50"
      },
      {
        "name": "Andrew Miller",      # Data of Person 2
        "age": "60"
      },
      .                               # Data of Person N
      .
      .
    ]
  }
}
```

您不限于使用特殊对象类型。您还可以在常规对象类型的字段中使用列表。

非 Null 值

非 Null 值表示在响应中不能为 Null 的字段。您可以使用 ！符号将字段设置为非 Null：

```
type Person {
  name: String!
  age: Int
}
type Query {
  people: [Person]
}
```

name 字段不能明确为 Null。如果您要查询数据源并为该字段提供 Null 输入，则会引发错误。

您可以组合使用列表和非 Null 值。比较以下查询：

```
type Query {
  people: [Person!]      # Use case 1
}

.
.
```

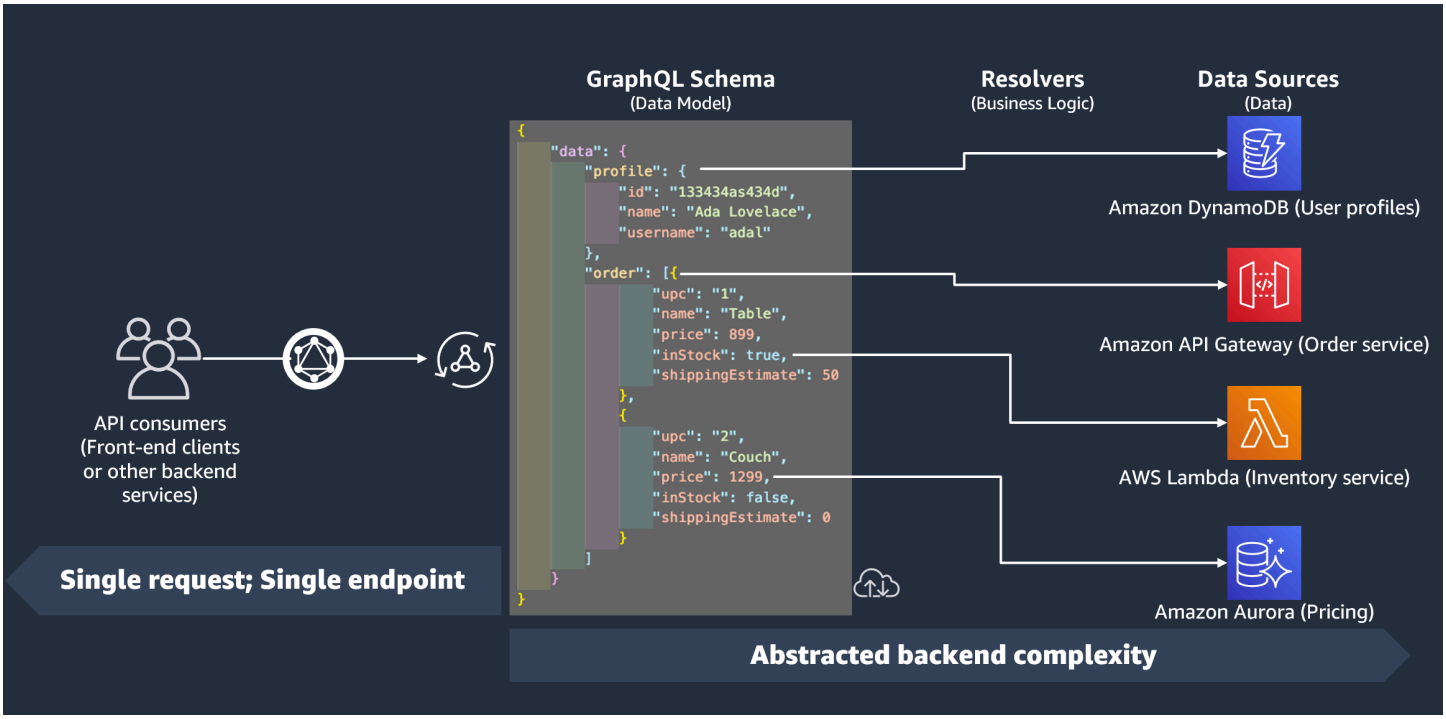
```
.

type Query {
  people: [Person]!      # Use case 2
}

.
.
.

type Query {
  people: [Person!]!     # Use case 3
}
```

在使用案例 1 中，列表不能包含 Null 项目。在使用案例 2 中，列表本身不能设置为 Null。在使用案例 3 中，列表及其项目不能为 Null。不过，在任何案例中，您仍然可能会返回空列表。

正如您看到的一样，在 GraphQL 中具有很多不断变化的组件。在本节中，我们说明了一个简单架构的结构以及架构支持的各种类型和字段。在下一节中，您将了解 GraphQL API 的其他组件以及它们如何与架构一起使用。

数据源

在上一节中，我们了解了架构定义数据的形状。不过，我们从未介绍这些数据来自哪里。在实际项目中，您的架构就像一个网关，它处理向服务器发出的所有请求。在发出请求时，架构充当与客户端交互的单个终端节点。架构访问和处理数据，并将其从数据源转发回客户端。请参阅下面的信息图：



AWS AppSync 和 GraphQL 出色地实施了服务于前端的后端 (BFF) 解决方案。它们协同工作，通过抽象后端来降低大规模系统的复杂性。如果您的服务使用不同的数据源和/或微服务，您实际上可以在单个架构（超图）中定义每个源（子图）的数据形状以抽象掉一些复杂性。这意味着您的 GraphQL API 不限于使用一个数据源。您可以将任意数量的数据源与 GraphQL API 关联，并在代码中指定它们如何与服务进行交互。

正如您在信息图中看到的一样，GraphQL 架构包含客户端请求数据所需的所有信息。这意味着可以在单个请求中处理所有内容，而不是像 REST 那样在多个请求中进行处理。这些请求经过架构，这是服务的唯一终端节点。在处理请求时，解析器（在下一节中介绍）执行其代码以处理来自相关数据源的数据。在返回响应时，将使用架构中的数据填充与数据源关联的子图。

AWS AppSync 支持很多不同的数据源类型。在下表中，我们将描述每种类型，列出每种类型的一些优点，并提供非常有用的链接以获取额外的背景信息。

数据来源	描述	优势	补充信息

Amazon DynamoDB	“Amazon DynamoDB 是一种完全托管的 NoSQL 数据库服务，可以提供快速且可预测的性能以及无缝可	• 大规模性能：DynamoDB 的设计理念是，在任何规模都能提供一致的性能。可以通过使	• DynamoDB 官方文档
			  • 分区
			  • 自动扩展
			  • 容错能力

数据来源	描述	优势	补充信息
	扩展性。DynamoDB 可以免除操作和扩展分布式数据库的管理工作负担，因而无需担心硬件预置、设置和预置、复制、软件修补或集群扩展等问题。DynamoDB 还提供静态加密，从而消除了保护敏感数据时涉及的运行负担和复杂性。”	用分区来实现该目的。DynamoDB 自动将您的表划分为多个分配，这些分配将存储在位于多个节点的多个 SSD 中。这通常会增加网络吞吐量并减少延迟。 • 大规模容量：DynamoDB 监控您的流量，并允许您在网络长时间处于过载状态时自动扩展吞吐量。 • 可用性和容错能力：DynamoDB 由多个物理隔离的区域支持，每个区域包含多个物理隔离的可用区。如果服务中断，DynamoDB 将自动切换到备份区域。您也可以手动备份和复制数据以保证数据可用性。 • 日志记录和监控：DynamoDB 为您的表提供了多种分析工具。您可以监控表的性能，并创建警报以向您通知服务的重大变化。	• 监控 • 安全性 • GraphQL 和 DynamoDB • DynamoDB 的解析器操作 • 定价模式

数据来源	描述	优势	补充信息
		• 安全性：DynamoDB 采用严格的协议，以确保您的数据符合组织的安全要求。 • 与 AWS AppSync 集成：DynamoDB 与我们的服务无缝集成在一起。您可以创建新的 DynamoDB 表，并自动从中生成架构以简化您的开发过程。我们还提供了一整套操作，以便在解析器中轻松从您的账户中的现有 DynamoDB 表请求数据。	

数据来源	描述	优势	补充信息

AWS Lambda	“AWS Lambda 是一种计算服务，可用于运行代码，而无需预置或管理服务器。
	
	Lambda 在可用性高的计算基础设施上运行您的代码，执行计算资源的所有管理工作，其中包括服务器和操作系统维护、容量预置和弹性伸缩和记录。在使用 Lambda 时，您只需在 Lambda 支持的语言运行时环境之一中提供代码。”
		• 按实际使用量付费模式：Lambda 仅在您使用其资源时向您收费。它们还允许您根据应用程序需求扩展使用的资源量。
		• 自动扩展：有时，您的应用程序可能需要在特定进程中使用额外的计算能力。Lambda 允许您自动扩展计算资源以满足您的应用程序需求。
		• 更快的部署时间：您可以通过部署包简化您的开发过程。可以使用包将函数代码上传到 Lambda 服务中。然后，您可以使用它们的运行时环境测试和执行您的函数。
		• 多功能性：Lambda 可用于多种使用案例。您可以将 Lambda 与第三方服务和 AWS 服务无缝集成在一起。一些示例包括 CI/CD 管道和群发邮件服务。
			• 官方文档
			  • 扩展
			  • 部署
			  • 运行时环境
			• Lambda 解析器教程
			• 定价模式

数据来源	描述	优势	补充信息
		• 与 AWS AppSync 集成：您可以轻松在解析器中调用 Lambda 函数以处理请求。我们的服务提供简化的请求操作以执行 Lambda 调用。我们允许单个调用和批量调用。	

数据来源	描述	优势	补充信息
OpenSearch	“Amazon OpenSearch Service 是一种托管服务，可以轻松在 AWS 云中部署、运行和扩展 OpenSearch 集群。Amazon OpenSearch Service 支持 OpenSearch 和旧 Elasticsearch OSS（最高 7.10，该软件的最终开源版本）。创建集群时，您可以选择使用哪种搜索引擎。 OpenSearch 是一个全面开源搜索和分析引擎，用例包括日志分析、实时应用程序监控、点击流分析等。有关更多信息，请参阅 OpenSearch 文档。 Amazon OpenSearch Service 可为 OpenSearch 集群预置所有资源，并启动集群。它还自动检测和替换失败的 OpenSearch Service 节点，减少与自管理基础设施相关的开销。您只需调用一次 API 或在控制台中单击几	• 扩展：您可以通过 OpenSearch Serverless 轻松扩展服务以满足您的服务要求。 • 数据摄取：您可以使用 OpenSearch 摄取导入、处理和分析数据。有很多数据摄取应用程序，您可以在此处找到这些应用程序。 • 安全性：OpenSearch 可以管理您的 AWS 安全配置，包括 IAM、CloudTrail、VPC、身份验证等。 • 可用性：OpenSearch 还在服务中支持不同的区域和可用区。 • 与 AWS AppSync 集成：在 AWS AppSync 中，可以使用 GraphQL API 在您的账户的现有 OpenSearch Service 域中存储和检索数据。	• 官方文档 • Serverless (无服务器) • 定价模式

数据来源	描述	优势	补充信息
	下鼠标按钮，即可扩展集群。”		
HTTP 终端节点	您可以将 HTTP 终端节点作为数据源。AWSAppSync 可以向终端节点发送具有参数和负载等相关信息的请求。将向解析器公开 HTTP 响应，解析器在完成操作后返回最终响应。	• 对于没有与 Lambda 等服务集成的简单应用程序非常有用。	• 解析器参考

数据来源	描述	优势	补充信息

Amazon EventBridge	“EventBridge 是一种无服务器服务，它使用事件将应用程序组件连接在一起，以使您更轻松地构建可扩展的事件驱动应用程序。可以使用该服务将事件从源（本地应用程序、AWS 服务和第三方软件等）路由到您的组织中的使用者应用程序。EventBridge 提供了一种简单且一致的方式以摄取、筛选、转换和交付事件，因此，您可以快速构建新的应用程序。”	• 事件驱动的架构：您可以利用事件驱动的架构。
		• 计划：您可以使用 EventBridge 调度器通过 cron 表达式自动执行任务和规则，或者设置时间间隔以作为事件模式的替代方案。
		• 管道：通过使用 EventBridge 管道，您可以将事件总线替换为管道，其中包括额外的筛选事件模式，并在将事件发送到目标之前通过数据转换扩充数据。
		• 与 AWS AppSync 集成：AWS AppSync 允许您使用解析器将事件发送到事件总线。
			• 官方文档
			  • 管道
			  • 调度器
			• 解析器参考
			• 定价模式

数据来源	描述	优势	补充信息

关系数据库	“Amazon Relational Database Service (Amazon RDS) 是一种 Web 服务，可以在 AWS 云中更轻松地设置、运行和扩展关系数据库。它为符合行业标准的关系数据库提供经济高效且可调整大小的容量，并管理常见的数据库管理任务。”	• 管理变得简单：RDS 定期对其资源进行维护。维护通常涉及对数据库实例的底层硬件、底层操作系统 (OS) 或数据库引擎版本进行更新。正常情况下，您可以决定何时执行更新（安全补丁除外）。
		• 建议：RDS 的建议功能提供自动建议以修复实例中的潜在问题。
		• 可用性：在全球的不同物理区域中提供了 RDS。您可以轻松将数据库需求分配给不同的节点，以更好地为您的客户提供服务。
		• 自定义：RDS 专为满足大型企业的要求而量身定制。RDS 提供了不同的计算、快速部署、扩展性和存储选项。
		• 安全性：RDS 与多种工具和服务集成在一起，以保持用户、数据库和网络级别的数据库安全性。
			• 官方文档
			  • 功能
			  • Maintenance
			  • 建议
			  • 存储选项
			  • 可用性
			  • 安全性
			• 定价模式

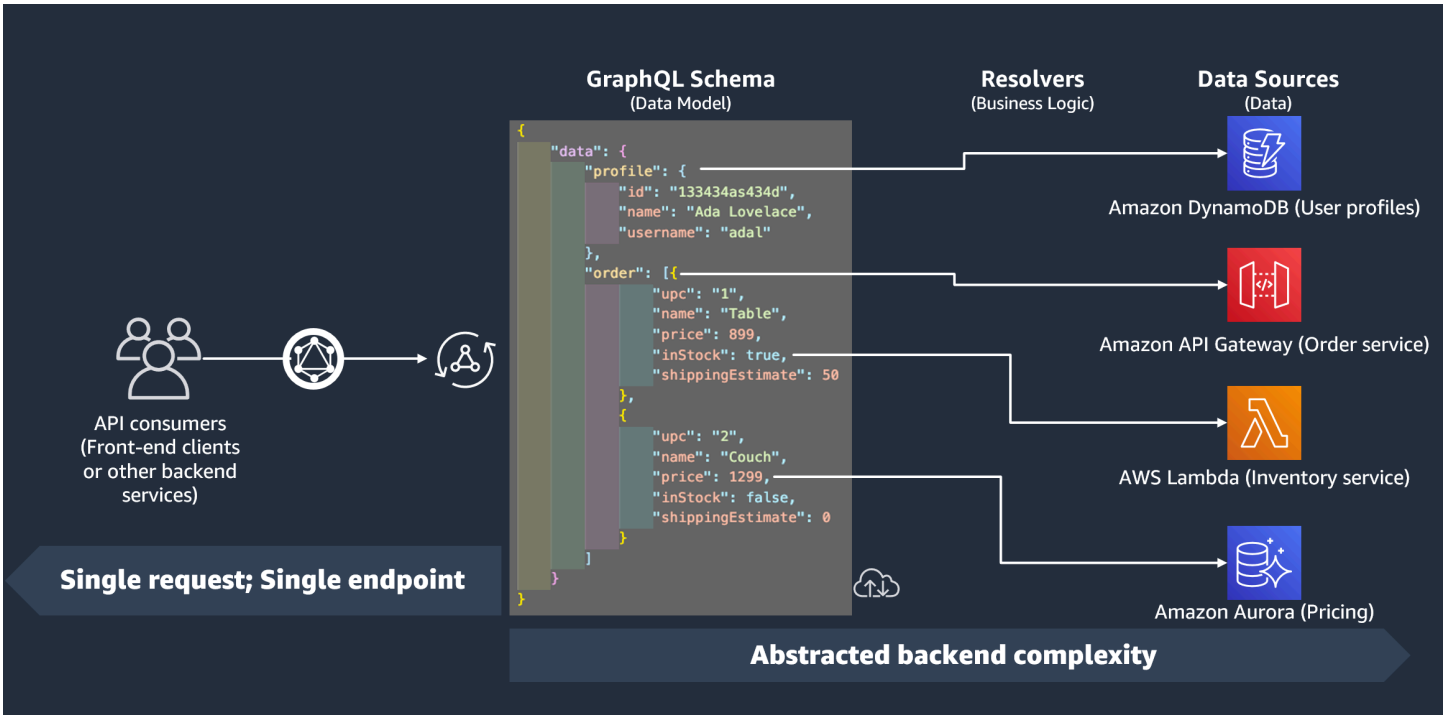
数据来源	描述	优势	补充信息
		• 与 AWS AppSync 集成：如果您正在寻找成熟的后端解决方案，AWS AppSync 是您的理想之选，它允许您将实例作为数据源以发送、处理、存储和返回数据。	
None 数据源	如果不打算使用数据源服务，您可以将其设置为 none。虽然 none 数据源仍明确归类为数据源，但并不是存储介质。尽管如此，它在某些情况下对于数据处理和传递仍然非常有用。	• 对于数据转换等可能非常有用 • 在本地解决问题时非常有用	• 解析器参考

> **Tip**
>
> 有关数据源如何与 AWS AppSync 交互的更多信息，请参阅附加数据源。

解析器

从前面的几节中，您了解了架构和数据源的组件。现在，我们需要解决架构和数据源如何交互的问题。这一切都始于解析器。

解析器是一个代码单元，可以处理向服务发出请求时如何解析该字段的数据的问题。解析器附加到架构中的您的类型内的特定字段。它们通常用于实施查询、变更和订阅字段操作的状态更改操作。解析器处理客户端的请求，然后返回结果，结果可能是一组输出类型，例如对象或标量：



解析器运行时环境

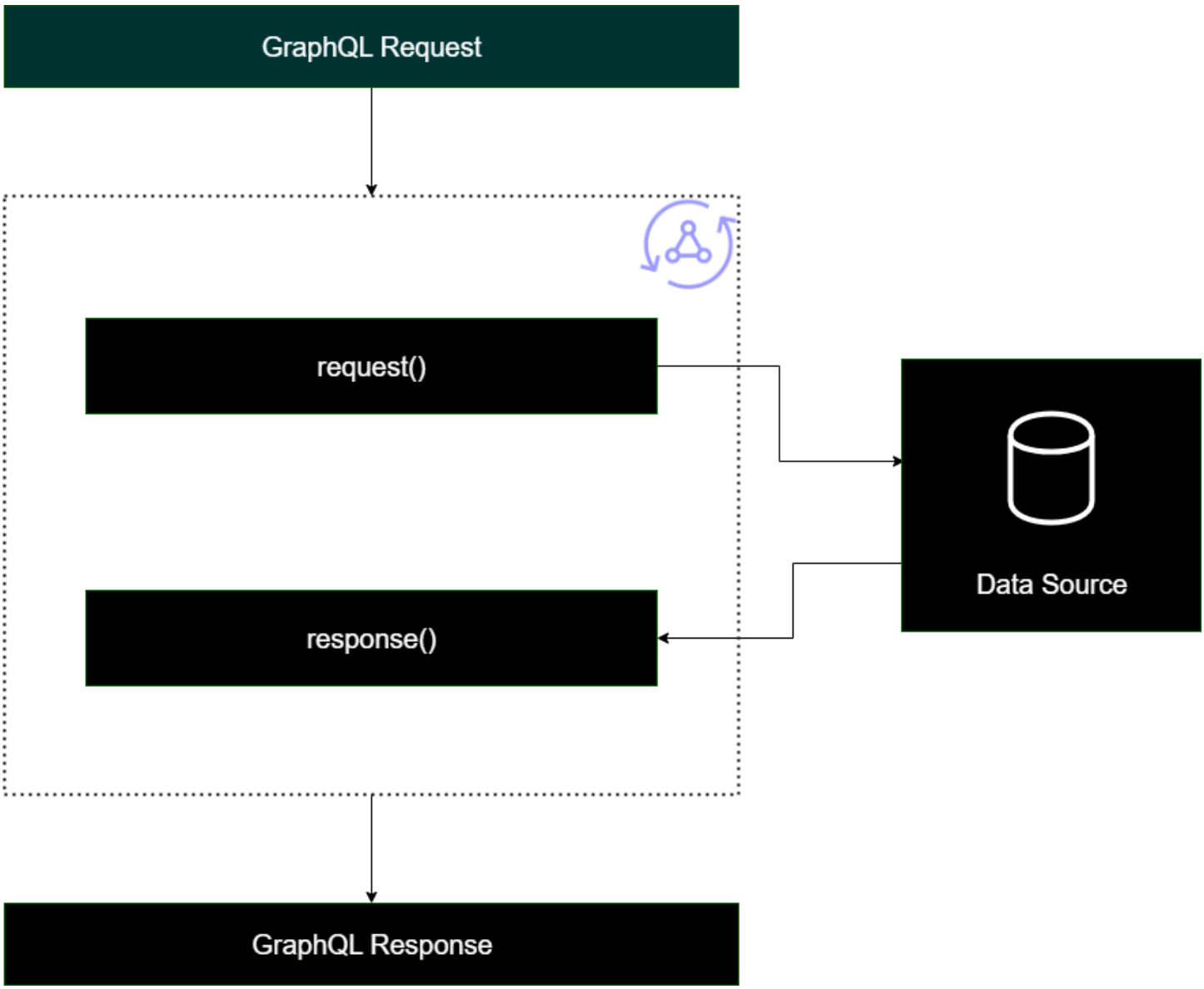
在 AWS AppSync 中，您必须先为解析器指定运行时环境。解析器运行时环境表示执行解析器的环境。它还规定了编写解析器时使用的语言。AWS AppSync 目前支持适用于 JavaScript 的 APPSYNC_JS 和 Velocity 模板语言 (VTL)。请参阅解析器和函数的 JavaScript 运行时功能 (JavaScript) 或解析器映射模板实用程序参考 (VTL)。

解析器结构

从代码角度看，可以通过多种方式设置解析器结构。具有单位解析器和管道解析器。

单位解析器

单位解析器由定义对数据源执行的单个请求和响应处理程序的代码组成。请求处理程序将上下文对象作为参数，并返回用于调用数据源的请求负载。响应处理程序接收从数据源返回的负载以及执行的请求结果。响应处理程序将负载转换为 GraphQL 响应以解析 GraphQL 字段。



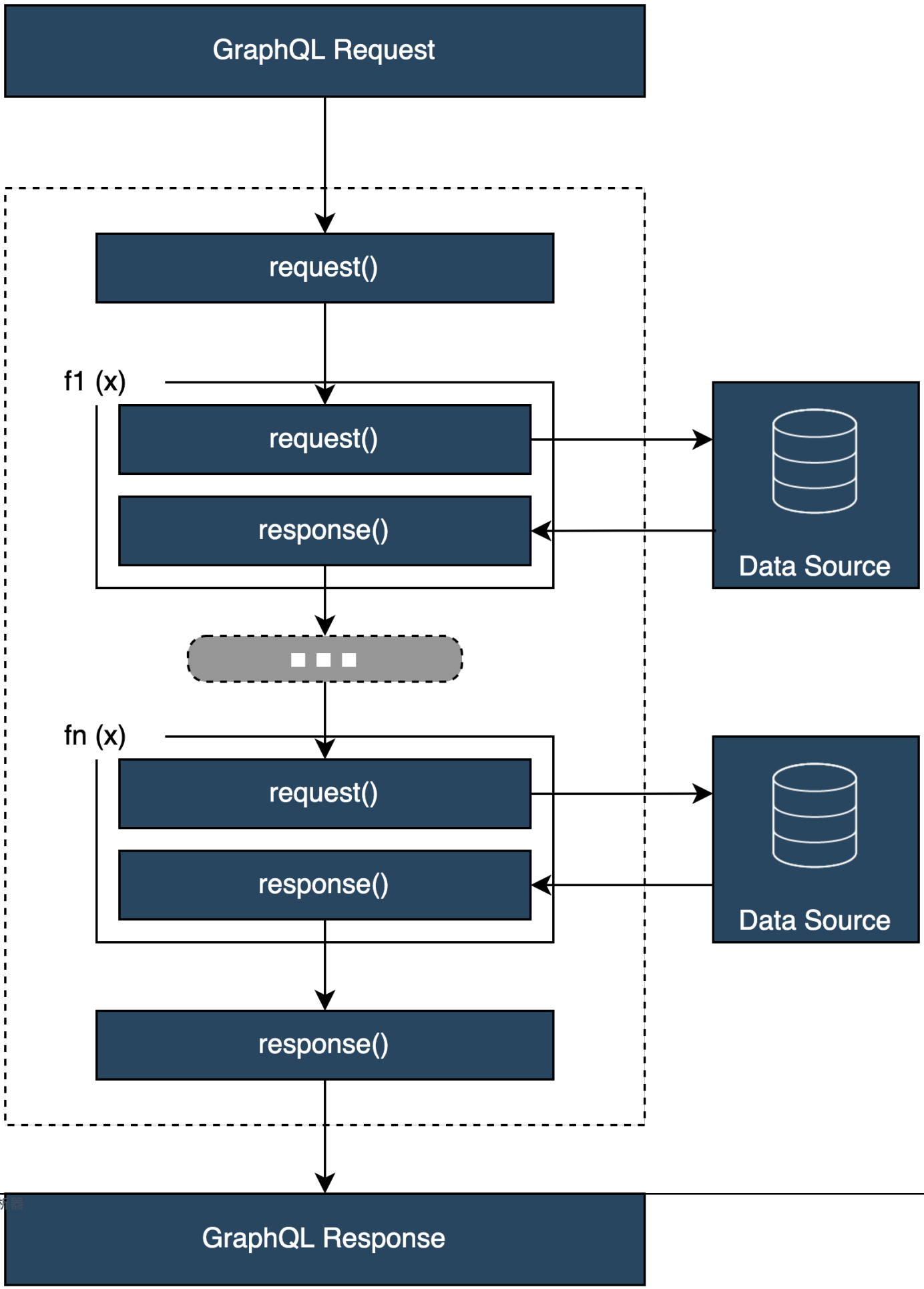
管道解析器

在实施管道解析器时，它们采用通用的结构：

- 预备步骤：在客户端发出请求时，将为使用的架构字段（通常是查询、变更、订阅）的解析器传送请求的数据。解析器开始使用预备步骤处理程序处理请求数据，该处理程序允许在数据传送到解析器之前执行一些预处理操作。
- 函数：在运行预备步骤后，请求传送到函数列表。将对数据源执行列表中的第一个函数。函数是解析器代码的子集，其中包含自己的请求和响应处理程序。请求处理程序获取请求数据，并对数据源执行操作。在将数据源的响应传回到列表之前，响应处理程序对其进行处理。如果具有多个函数，请求数

据将发送到列表中的下一个函数以进行执行。列表中的函数按照开发人员定义的顺序依次执行。在执行所有函数后，最终结果传送到后续步骤。

- 后续步骤：后续步骤是一个处理程序函数，允许您在将最终函数的响应传送到 GraphQL 响应之前对其执行一些最终操作。


GraphQL Request
request()
f1 (x)
request()
response()
Data Source
fn (x)
request()
response()
Data Source
response()
GraphQL Response

解析器处理程序结构

处理程序通常是名为 Request 和 Response 的函数：

```
export function request(ctx) {
    // Code goes here
}

export function response(ctx) {
    // Code goes here
}
```

在单位解析器中，只有一组这样的函数。在管道解析器中，预备步骤和后续步骤具有一组这样的函数，并为每个函数额外提供一组这样的函数。为了直观地了解它的外观，让我们看一个简单的 Query 类型：

```
type Query {
 helloWorld: String!
}
```

这是一个简单的查询，其中包含一个名为 helloWorld 且类型为 String 的字段。假设我们始终希望该字段返回字符串“Hello World”。为了实现该行为，我们需要将解析器添加到该字段中。在单位解析器中，我们可以添加如下内容：

```
export function request(ctx) {
    return {}
}

export function response(ctx) {
    return "Hello World"
}
```

request 可以直接保留空白，因为我们不会请求或处理数据。我们还可以假设数据源是 None，表明该代码不需要执行任何调用。响应仅返回“Hello World”。为了测试该解析器，我们需要使用该查询类型发出请求：

```
query helloWorldTest {
  helloWorld
}
```

以下是一个名为 helloWorldTest 的查询，它返回 helloWorld 字段。在执行时，helloWorld 字段解析器也会执行并返回响应：

```
{
  "data": {
    "helloWorld": "Hello World"
  }
}
```

像这样返回常量是您可以执行的最简单操作。实际上，您将返回输入、列表等。以下是一个更复杂的示例：

```
type Book {
  id: ID!
  title: String
}

type Query {
  getBooks: [Book]
}
```

此处，我们返回一个 Books 列表。假设我们使用 DynamoDB 表存储图书数据。我们的处理程序可能如下所示：

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

我们的请求使用内置扫描操作搜索表中的所有条目，将结果存储在上下文中，然后将其传送到响应。响应获取结果项目，并在响应中返回它们：

```
{
  "data": {
    "getBooks": {
      "items": [
        {
          "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
          "title": "book1"
        },
        {
          "id": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
          "title": "book2"
        },

        ...

      ]
    }
  }
}
```

解析器上下文

在解析器中，处理程序链中的每个步骤必须了解以前步骤中的数据状态。可以存储一个处理程序的结果，并将其作为参数传递给另一个处理程序。GraphQL 定义了 4 个基本解析器参数：

解析器基本参数	描述
`obj` `root`、`parent`、等	父项的结果。
`args`	为 GraphQL 查询中的字段提供的参数。
`context`	为每个解析器提供的值，其中保存重要上下文信息，例如当前登录的用户或数据库的访问权限。
`info`	该值保存与当前查询相关的字段特定信息以及架构详细信息。

在 AWS AppSync 中，`context` (ctx) 参数可以保存所有上述数据。它是根据请求创建的对象，并包含授权凭证、结果数据、错误、请求元数据等数据。上下文为程序员提供了一种简单方法，以处理来自请求的其他部分的数据。再看一下该代码片段：

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
 */
export function response(ctx) {
  return ctx.result.items;
}
```

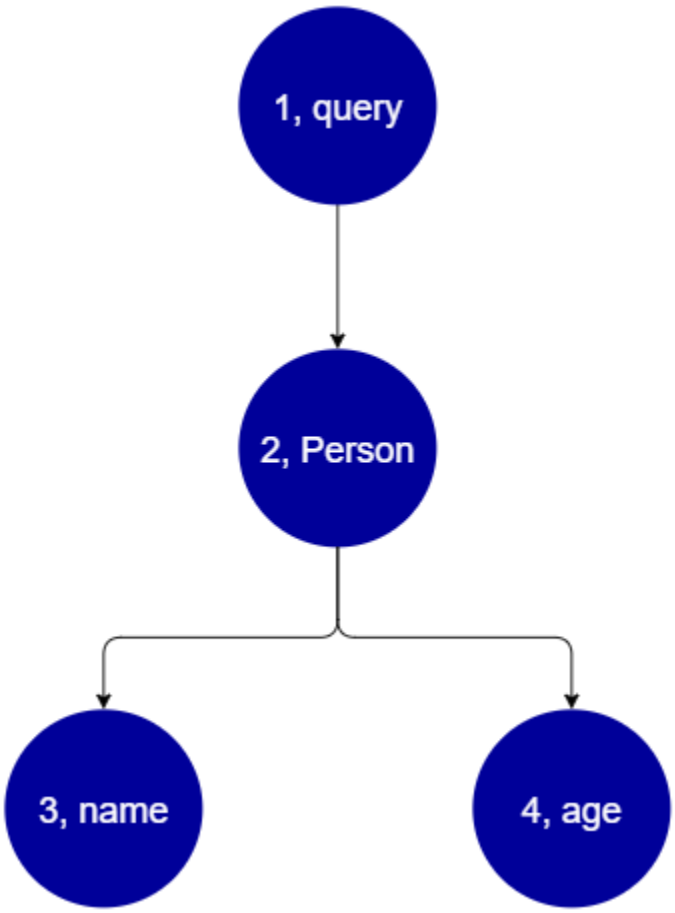
为请求提供了上下文 (ctx) 作为参数；这是请求的状态。它对表中的所有项目执行扫描，并将结果重新存储到 `result` 的上下文中。然后，将上下文传递给响应参数，该参数访问 `result` 并返回其内容。

请求和解析

在对 GraphQL 服务进行查询时，它必须在执行之前执行解析和验证过程。将解析您的请求并转换为抽象语法树，并对您的架构运行多种验证算法以验证树内容。在执行验证步骤后，将遍历并处理树的节点，调用解析器，将结果存储在上下文中，然后返回响应。例如，使用以下查询：

```
query {
  Person {  //object type
    name  //scalar
    age   //scalar
  }
}
```

我们返回具有 `name` 和 `age` 字段的 `Person`。在运行该查询时，树如下所示：



从树中可以看出，该请求在架构中搜索根以查找 Query。在查询中，将解析 Person 字段。从前面的示例中，我们知道这可能是来自用户的输入、值列表等。Person 很可能与包含我们所需的字段（name 和 age）的对象类型相关联。在找到这两个子字段后，它们按给定的顺序进行解析（先解析 name，然后解析 age）。在完全解析树后，将完成请求并将其发回到客户端。

GraphQL 的其他属性

GraphQL 由多种设计原则组成，以在大规模系统中保持简单性和稳健性。

声明性

GraphQL 是声明性的，这意味着用户仅声明他们希望查询的字段以描述数据（设置形状）。响应仅返回这些属性的数据。例如，以下操作检索 DynamoDB 表中的 Book 对象，其 ISBN 13 id 值为 *9780199536061*：

```
{
  getBook(id: "9780199536061") {
    name
```

```
    year
    author
  }
}
```

响应返回负载中的字段（name、year 和 author），而不返回任何其他内容：

```
{
  "data": {
    "getBook": {
      "name": "Anna Karenina",
      "year": "1878",
      "author": "Leo Tolstoy",
    }
  }
}
```

由于这一设计原则，GraphQL 消除了 REST API 在复杂系统中经常遇到的过度获取和获取不足问题。这可以提高数据收集效率并提高网络性能。

分层

GraphQL 非常灵活，用户可以设置请求的数据形状以满足应用程序需求。请求的数据始终遵循 GraphQL API 中定义的属性的类型和语法。例如，以下代码片段显示具有名为 quotes 的新字段范围的 getBook 操作，该操作返回与 Book *9780199536061* 关联的所有存储的引用字符串和页面：

```
{
  getBook(id: "9780199536061") {
    name
    year
    author
    quotes {
      description
      page
    }
  }
}
```

运行该查询将返回以下结果：

```
{
```

```
  "data": {
    "getBook": {
      "name": "Anna Karenina",
      "year": "1878",
      "author": "Leo Tolstoy",
      "quotes": [
         {
            "description": "The highest Petersburg society is essentially one: in it
 everyone knows everyone else, everyone even visits everyone else.",
            "page": 135
         },
         {
            "description": "Happy families are all alike; every unhappy family is
 unhappy in its own way.",
            "page": 1
         },
         {
            "description": "To Konstantin, the peasant was simply the chief partner in
 their common labor.",
            "page": 251
         }
      ]
    }
  }
}
```

正如您看到的一样，与请求的书籍关联的 quotes 字段以数组形式返回，其格式与查询描述的格式相同。尽管此处未显示，但 GraphQL 具有额外的优势，即不特别关注它检索的数据的位置。Books 和 quotes 可以是单独存储的，但只要存在关联，GraphQL 仍会检索该信息。这意味着，您的查询可以在单个请求中检索大量单独的数据。

内省

GraphQL 是自说明性的，或者说是内省的。它支持多种内置操作，以使用户能够查看架构中的基本类型和字段。例如，以下是具有 date 和 description 字段的 Foo 类型：

```
type Foo {
 date: String
 description: String
}
```

我们可以使用 _type 操作在架构下面查找类型元数据：

```
{
  __type(name: "Foo") {
    name                    # returns the name of the type
    fields {                # returns all fields in the type
      name                  # returns the name of each field
      type {                # returns all types for each field
        name                # returns the scalar type
      }
    }
  }
}
```

这会返回一个响应：

```
{
  "__type": {
    "name": "Foo",                    # The type name
    "fields": [
      {
        "name": "date",               # The date field
        "type": { "name": "String" }  # The date's type
      },
      {
        "name": "description",        # The description field
        "type": { "name": "String" }  # The description's type
      },
    ]
  }
}
```

该功能可用于找出特定 GraphQL 架构支持哪些类型和字段。GraphQL 支持各种这样的内省操作。有关更多信息，请参阅内省。

强类型

GraphQL 通过其类型和字段系统支持强类型。在架构中定义某些内容时，它必须具有可以在运行时之前验证的类型。它还必须遵循 GraphQL 的语法规范。这个概念与其他语言的编程没有什么不同。例如，以下是以前的 Foo 类型：

```
type Foo {
 date: String
```

```
  description: String
}
```

我们可以看到 Foo 是将创建的对象。在 Foo 实例中，具有一个 date 和 description 字段，它们都是 String 基元类型（标量）。从语法上讲，我们看到已声明 Foo，并且它的字段位于其范围内。这种类型检查和逻辑语法的组合确保您的 GraphQL API 简洁且不言自明。可以在此处找到 GraphQL 的类型和语法规范。

入门：创建您的第一个 GraphQL API

您可以使用 AWS AppSync 控制台配置和启动 GraphQL API。GraphQL API 通常需要三个组件：

1. GraphQL 架构 - 您的 GraphQL 架构是 API 的蓝图。它定义您可以在执行操作时请求的类型和字段。要使用数据填充架构，您必须将数据源连接到 GraphQL API。在该快速入门指南中，我们使用预定义的模型创建一个架构。
2. 数据源 - 这些资源包含用于填充 GraphQL API 的数据。它可以是 DynamoDB 表、Lambda 函数等。AWSAppSync 支持多种数据源以构建稳健且可扩展的 GraphQL API。数据源链接到架构中的字段。每次对字段执行请求时，来自源的数据将填充该字段。该机制是由解析器控制的。在该快速入门指南中，我们使用预定义的模型以及架构创建一个数据源。
3. 解析器 - 解析器负责将架构字段链接到数据源。它们从源中检索数据，然后根据字段定义的内容返回结果。AWSAppSync 支持使用 JavaScript 和 VTL 编写 GraphQL API 的解析器。在该快速入门指南中，将根据架构和数据源自动生成解析器。我们不会在本节中深入介绍该内容。

AWS AppSync 支持创建和配置所有 GraphQL 组件。在打开控制台时，您可以使用以下方法创建您的 API：

1. 设计自定义的 GraphQL API，即通过预定义的模型生成一个 GraphQL API，然后设置新的 DynamoDB 表（数据源）以支持该 API。
2. 使用空白架构设计 GraphQL API，没有数据源或解析器。
3. 使用 DynamoDB 表导入数据，并生成架构的类型和字段。
4. 使用 AWS AppSync 的 WebSocket 功能和 Pub/Sub 架构开发实时 API。
5. 使用现有的 GraphQL API（源 API）链接到合并的 API。

> **Note**
>
> 我们建议在使用更高级的工具之前查看设计架构一节。这些指南将介绍更简单的示例，从概念上讲，您可以使用这些示例在 AWS AppSync 中构建更复杂的应用程序。

AWS AppSync 还支持多种非控制台选项以创建 GraphQL API。其中包括：

1. AWS Amplify
2. AWS SAM

3. AWS CloudFormation
4. CDK

以下示例将说明如何使用预定义的模型和 DynamoDB 创建 GraphQL API 基本组件。

主题

步骤 1：启动架构

在该示例中，您创建一个 Todo API，以允许用户创建 Todo 项目以提供日常琐事提醒，例如 *`Finish task`* 或 *`Pick up groceries`*。该 API 将说明如何使用 GraphQL 操作，其中状态持久保留在 DynamoDB 表中。

从概念上讲，需要三个主要步骤以创建第一个 GraphQL API。您必须定义架构（类型和字段），将数据源附加到字段，然后编写处理业务逻辑的解析器。不过，控制台体验改变了该顺序。我们先定义希望数据源如何与架构交互，然后定义架构和解析器。

创建您的 GraphQL API

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
2. 在控制面板中，选择创建 API。
3. 在选择 GraphQL API 后，选择从头开始设计。然后选择下一步。
4. 对于 API 名称，将预填充的名称更改为 **`Todo API`**，然后选择下一步。

> **Note**
>
> 此处还提供了其他选项，但我们不会在该示例中使用这些选项。

5. 在指定 GraphQL 资源部分中，执行以下操作：

   a. 选择立即创建由 DynamoDB 表支持的类型。

> **Note**
>
> 这意味着，我们将创建一个新的 DynamoDB 表以附加为数据源。

b. 在模型名称字段中，输入 **Todo**。

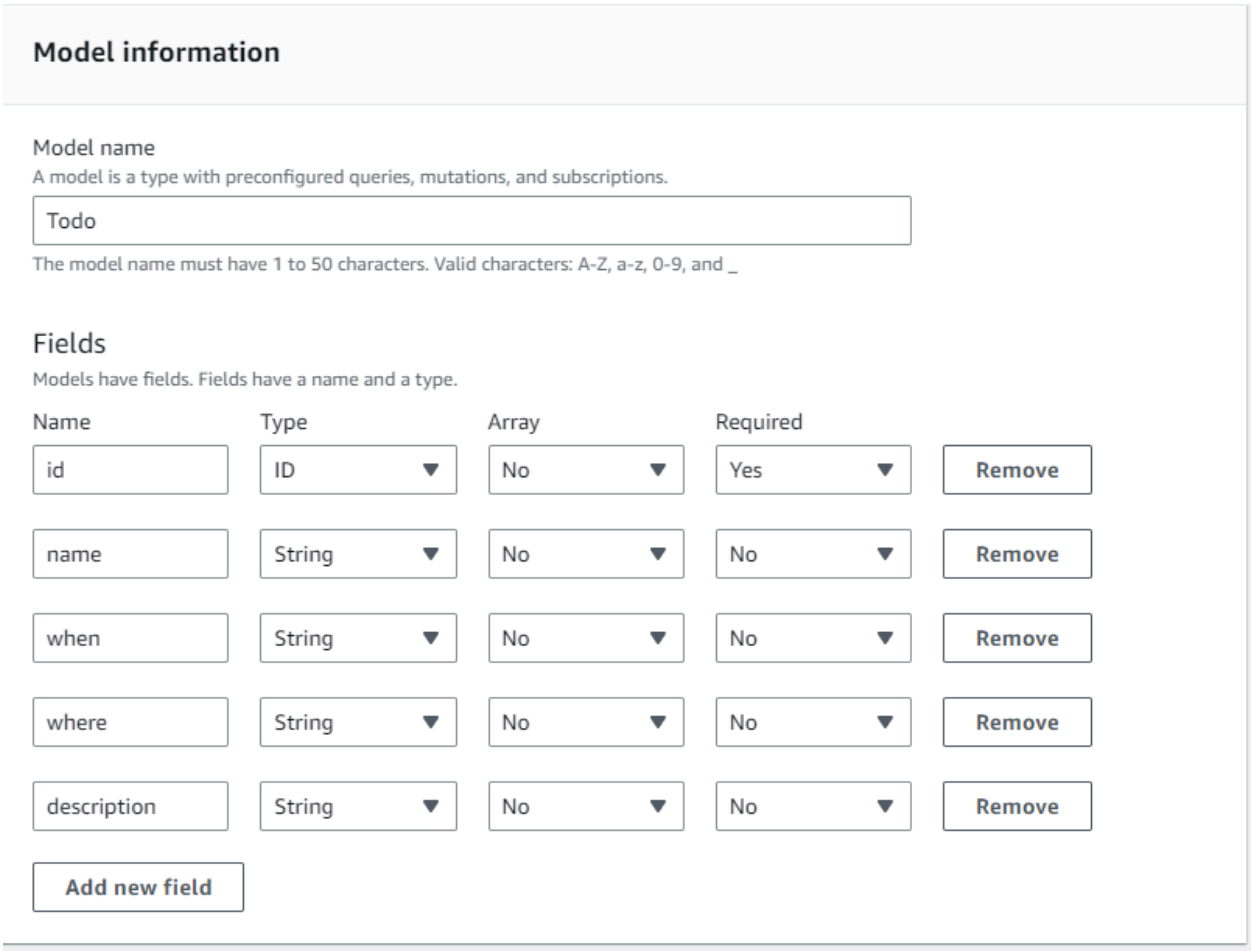
> **Note**
>
> 我们的第一个要求是，定义我们的架构。该模型名称是类型名称，因此，您真正要做的是创建一个名为 Todo 的 type，该类型将包含在架构中：
>
> ```
> type Todo {}
> ```

c. 在字段下面，执行以下操作：

i. 创建一个名为 **id** 的字段，“类型”为 ID，并且“必填”设置为 Yes。

> **Note**
>
> 这些字段将位于您的 Todo 类型范围内。此处的字段名称指定为 id，类型为 ID!：
>
> ```
> type Todo {
>  id: ID!
> }
> ```
>
> AWS AppSync 支持多个标量值以用于不同的使用案例。

ii. 通过使用添加新字段，创建 4 个额外的字段，并将Name值设置为 **name**、**when**、**where** 和 **description**。它们的Type值为 String，Array和Required值设置为 No。它将如下所示：



> **Note**
>
> 完整的类型及其字段如下所示：
>
> ```
> type Todo {
>  id: ID!
>  name: String
>  when: String
>  where: String
>  description: String
> }
> ```
>
> 由于我们使用该预定义的模型创建架构，因此，将根据 `create`、`delete` 和 `update` 等类型使用多个样板变更填充该架构，以帮助您轻松填充数据源。

d. 在配置模型表下面，输入表名称，例如 **`TodoAPITable`**。将主键设置为 `id`。

> **Note**
>
> 我们实际上创建一个名为 *TodoAPITable* 的新 DynamoDB 表，该表将作为主数据源附加到 API。我们的主键设置为以前定义的必填 `id` 字段。请注意，该新表是空白的，除了分区键以外，该表不包含任何内容。

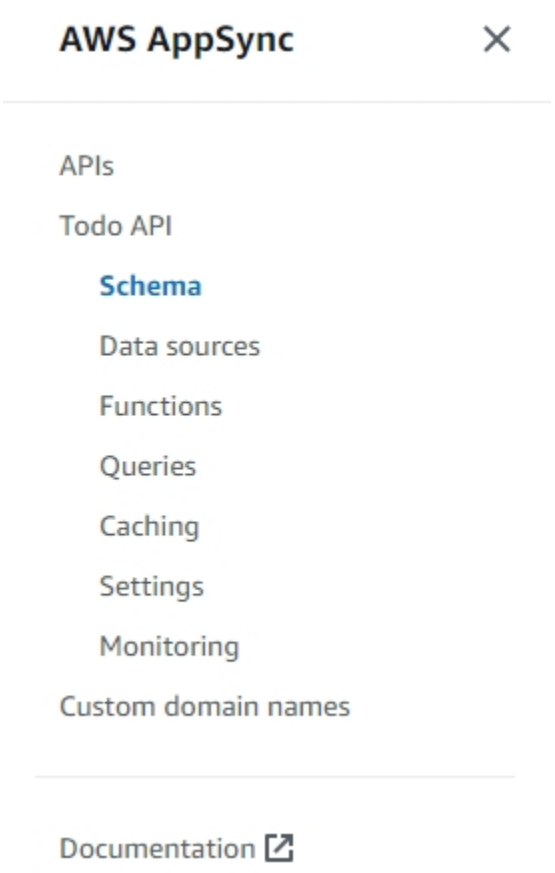
e. 选择下一步。

6. 检查您的更改，然后选择创建 API。稍等片刻，让 AWS AppSync 服务完成创建您的 API。

您已成功创建一个 GraphQL API 及其架构和 DynamoDB 数据源。简要说明一下上述步骤，我们选择创建一个全新的 GraphQL API。我们定义了 API 名称，然后添加第一个类型以添加架构定义。我们定义了类型及其字段，然后选择创建不包含数据的新 DynamoDB 表，以将该数据源附加到其中的一个字段。

步骤 2：浏览控制台

在将数据添加到 DynamoDB 表之前，我们应回顾一下 AWS AppSync 控制台体验的基本功能。通过使用 AWS AppSync 控制台页面左侧的选项卡，用户可以轻松导航到 AWS AppSync 提供的任何主要组件或配置选项：



架构设计器

选择架构以查看您刚刚创建的架构。如果您查看架构的内容，您会发现它已加载一些帮助程序操作以简化开发过程。在架构编辑器中，如果滚动查看代码，您最终会看到上一节中定义的模型：

```
type Todo {
 id: ID!
 name: String
 when: String
 where: String
 description: String
}
```

您的模型已成为在整个架构中使用的基本类型。我们将开始使用从该类型自动生成的变更，以将数据添加到我们的数据源中。

以下是有关架构编辑器的一些其他提示和信息：

1. 代码编辑器具有代码规范检查和错误检查功能，您可以在编写自己的应用程序时使用这些功能。

2. 在控制台的右侧显示已创建的 GraphQL 类型，以及不同顶级类型（例如查询）的解析器。
3. 在架构中添加新类型（例如 `type User {...}`）时，您可以让 AWS AppSync 为您预置 DynamoDB 资源。其中包括可与您的 GraphQL 数据访问模式进行最佳匹配的适当主键、排序键和索引。如果您选择顶部的 Create Resources (创建资源)，并从菜单中选择这些用户定义的类型之一，即可在架构设计中选择不同字段选项。我们将在设计架构一节中介绍该内容。

解析器配置

在架构设计器中，解析器部分包含架构中的所有类型和字段。如果滚动查看字段列表，您会发现可以选择附加以将解析器附加到某些字段。这会打开一个代码编辑器，您可以在其中编写解析器代码。AWSAppSync 支持 VTL 和 JavaScript 运行时环境，可以在页面顶部选择操作，然后选择更新运行时以更改运行时环境。您还可以在页面底部创建函数，它们将按顺序运行多个操作。不过，解析器是一个高级主题，我们不会在本节中介绍该内容。

数据源

选择数据源以查看您的 DynamoDB 表。通过选择`Resource`选项（如果可用），您可以查看数据源的配置。在我们的示例中，这会显示 DynamoDB 控制台。从该控制台中，可以编辑您的数据。您也可以选择数据源，然后选择编辑以直接编辑某些数据。如果需要删除数据源，您可以选择数据源，然后选择删除。最后，您可以选择创建数据源，然后配置名称和类型以创建新的数据源。请注意，该选项用于将 AWS AppSync 服务链接到现有的资源。您仍然需要使用相关服务在您的账户中创建资源，然后 AWS AppSync 才能识别该资源。

查询

选择查询以查看您的查询和变更。在使用我们的模型创建 GraphQL API 时，AWS AppSync 自动生成一些帮助程序变更和查询以用于测试目的。在查询编辑器中，左侧包含资源管理器。这是一个显示您的所有变更和查询的列表。您可以在此处单击名称值，以轻松启用要使用的操作和字段。这会使代码自动出现在编辑器的中心部分。在此处，您可以修改值以编辑变更和查询。在编辑器底部具有查询变量编辑器，可用于为操作的输入变量输入字段值。选择编辑器顶部的运行将打开一个下拉列表，可以从中选择要运行的查询/变更。该运行的输出显示在页面右侧。返回到顶部的资源管理器部分，您可以选择一个操作（查询、变更、订阅），然后选择 + 符号以添加该特定操作的新实例。在页面顶部还包含一个下拉列表，其中包含您的查询运行的授权模式。不过，我们不会在本节中介绍该功能（有关更多信息，请参阅安全性）。

设置

可以选择设置以查看 GraphQL API 的一些配置选项。在此处，您可以启用一些选项，例如日志记录、跟踪和 Web 应用程序防火墙功能。您也可以添加新的授权模式以保护您的数据，以免意外对外泄露数据。不过，这些选项是更高级的选项，不会在本节中介绍该内容。

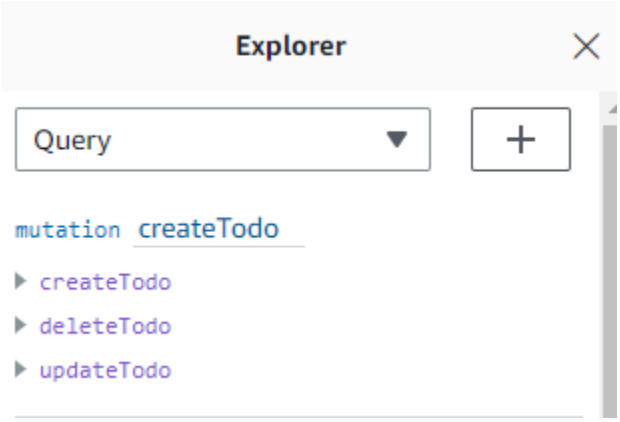
> **Note**
>
> 默认授权模式 API_KEY 使用 API 密钥测试应用程序。这是为所有新创建的 GraphQL API 提供的基本授权。我们建议您在生产环境中使用不同的方法。对于本节中的示例，我们仅使用 API 密钥。有关支持的授权方法的更多信息，请参阅安全性。

步骤 3：使用 GraphQL 变更添加数据

下一步是使用 GraphQL 变更将数据添加到空白 DynamoDB 表中。变更是 GraphQL 中的基本操作类型之一。它们是在架构中定义的，可用于处理数据源中的数据。就 REST API 而言，这些变更与 PUT 或 POST 等操作非常相似。

将数据添加到您的数据源

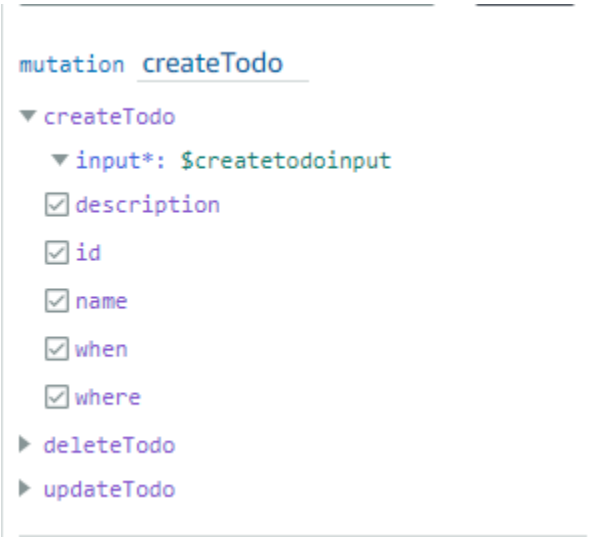
1. 如果您尚未登录到 AWS Management Console 并打开 AppSync 控制台，请执行以下操作。
2. 从表中选择您的 API。
3. 在左侧的选项卡中，选择查询。
4. 在表左侧的资源管理器选项卡中，您可能会看到在查询编辑器中已定义的多个变更和查询：



> **Note**
>
> 该变更实际作为 Mutation 类型包含在您的架构中。它具有以下代码：
>
> ```
> type Mutation {
>  createTodo(input: CreateTodoInput!): Todo
>  updateTodo(input: UpdateTodoInput!): Todo
>  deleteTodo(input: DeleteTodoInput!): Todo
> }
> ```
>
> 正如您看到的一样，此处的操作与查询编辑器中的操作类似。

AWS AppSync 根据我们以前定义的模型自动生成这些内容。该示例使用 createTodo 变更将条目添加到我们的 *TodoAPITable* 表中。

5. 在 createTodo 变更下面展开以选择 createTodo 操作：



启用所有字段的复选框，如上图所示。

> **Note**
>
> 您在此处看到的属性是变更的各种可修改元素。您的 input 可以被视为 createTodo 的参数。具有复选框的各种选项是一些字段，它们是在执行操作后在响应中返回的。

6. 在屏幕中央的代码编辑器中，您会注意到该操作显示在 createTodo 变更下面：

```
mutation createTodo($createtodoinput: CreateTodoInput!) {
  createTodo(input: $createtodoinput) {
    where
    when
    name
    id
    description
  }
}
```

Note

为了正确解释该代码片段，我们还必须查看架构代码。声明 mutation createTodo($createtodoinput: CreateTodoInput!){} 是具有操作之一 (createTodo) 的变更。完整的变更位于架构中：

```
type Mutation {
 createTodo(input: CreateTodoInput!): Todo
 updateTodo(input: UpdateTodoInput!): Todo
 deleteTodo(input: DeleteTodoInput!): Todo
}
```

返回到编辑器中的变更声明，参数是一个名为 $createtodoinput 的对象，所需的输入类型为 CreateTodoInput。请注意，CreateTodoInput（以及变更中的所有输入）也是在架构中定义的。例如，以下是 CreateTodoInput 的样板代码：

```
input CreateTodoInput {
 name: String
 when: String
 where: String
 description: String
}
```

它包含我们在模型中定义的字段，即 name、when、where 和 description。
返回到编辑器代码，在 createTodo(input: $createtodoinput) {} 中，我们将输入声明为 $createtodoinput，还会在变更声明中使用它。我们这样做是因为，这允许 GraphQL 根据提供的类型验证我们的输入，并确保它们与正确的输入一起使用。
编辑器代码的最后一部分显示执行操作后在响应中返回的字段：

```
{
    where
    when
    name
    id
    description
  }
```

在该编辑器下面的查询变量选项卡中，具有一个通用 createtodoinput 对象，它可能包含以下数据：

```
{
  "createtodoinput": {
    "name": "Hello, world!",
    "when": "Hello, world!",
    "where": "Hello, world!",
    "description": "Hello, world!"
  }
}
```

> **Note**
>
> 这是我们为前面提到的输入分配值的位置：
>
> ```
> input CreateTodoInput {
>  name: String
>  when: String
>  where: String
>  description: String
> }
> ```

添加我们希望放入 DynamoDB 表中的信息以更改 createtodoinput。在该示例中，我们希望创建一些 Todo 项目以作为提醒：

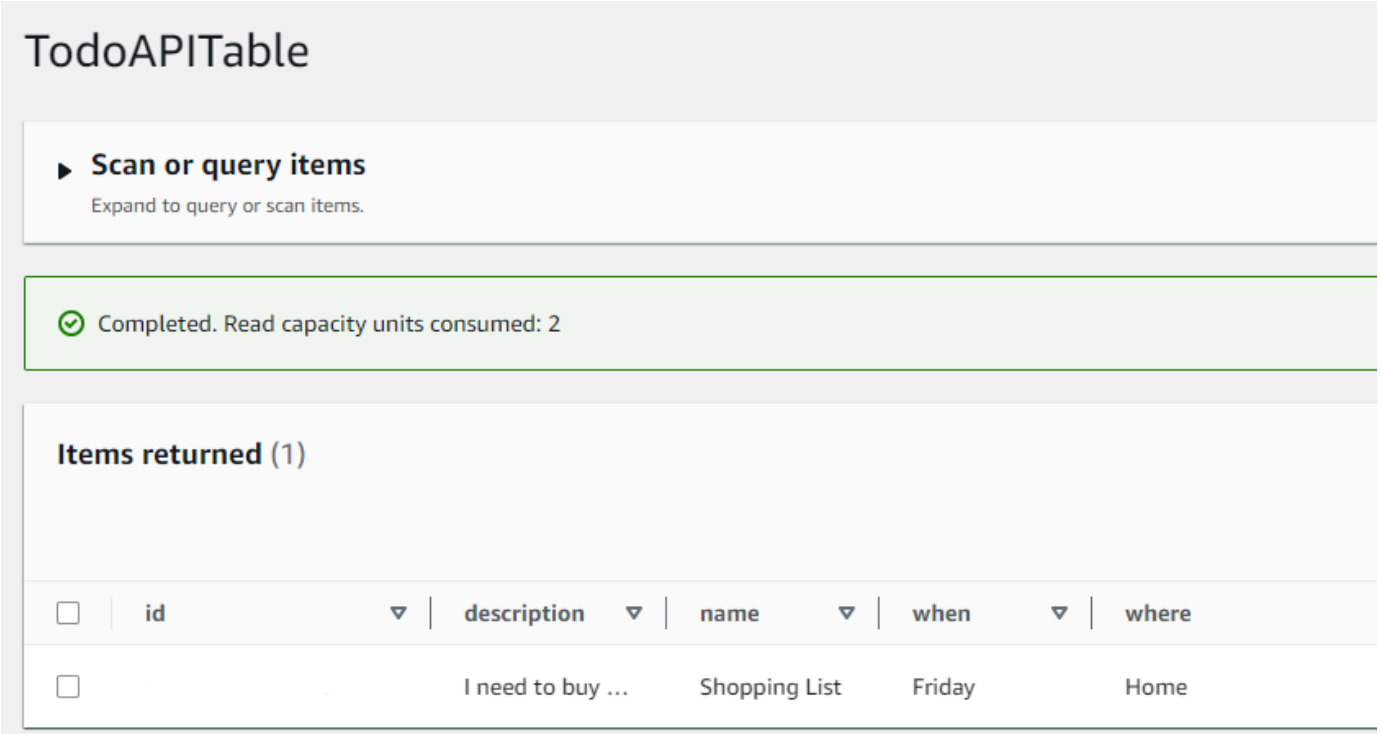
```
{
  "createtodoinput": {
```

```
    "name": "Shopping List",
    "when": "Friday",
    "where": "Home",
    "description": "I need to buy eggs"
  }
}
```

7. 选择编辑器顶部的运行。在下拉列表中选择 createTodo。在编辑器右侧，您应该会看到响应。它可能如下所示：

```
{
  "data": {
    "createTodo": {
      "where": "Home",
      "when": "Friday",
      "name": "Shopping List",
      "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
      "description": "I need to buy eggs"
    }
  }
}
```

如果您导航到 DynamoDB 服务，您现在会在数据源中看到一个包含以下信息的条目：



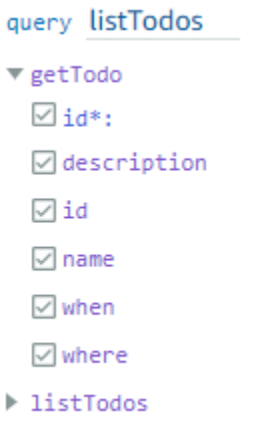
简要说明一下该操作，GraphQL 引擎解析记录，然后解析器将其插入到您的 Amazon DynamoDB 表中。同样，您可以在 DynamoDB 控制台中验证这一点。请注意，您不需要传入 `id` 值。将生成一个 `id` 并在结果中返回。这是因为，对于 DynamoDB 资源上设置的分区键，该示例在 GraphQL 解析器中使用 `autoId()` 函数。我们将在另一节中介绍如何构建解析器。记下返回的 `id` 值；您在下一节中使用该值通过 GraphQL 查询检索数据。

步骤 4：使用 GraphQL 查询检索数据

您的数据库现已包含一条记录，您将在运行查询时获得结果。查询是 GraphQL 的其他基本操作之一。它用于从数据源中解析和检索信息。就 REST API 而言，这与 `GET` 操作类似。GraphQL 查询的主要优点是，能够指定应用程序的确切数据要求，以使您在正确的时间获取相关的数据。

查询您的数据源

1. 如果您尚未登录到 AWS Management Console 并打开 AppSync 控制台，请执行以下操作。
2. 从表中选择您的 API。
3. 在左侧的选项卡中，选择查询。
4. 在表左侧的资源管理器选项卡中，在 `query listTodos` 下面展开 `getTodo` 操作：



5. 在代码编辑器中，您应该会看到操作代码：

```
query listTodos {
  getTodo(id: "") {
    description
    id
    name
    when
    where
  }
```

在 (id:"") 中，填写您在变更操作结果中保存的值。在我们的示例中，这是：

```
query listTodos {
  getTodo(id: "abcdefgh-1234-1234-1234-abcdefghijkl") {
    description
    id
    name
    when
    where
  }
```

6. 选择运行，然后选择 listTodos。结果将显示在编辑器右侧。我们的示例如下所示：

```
{
  "data": {
    "getTodo": {
      "description": "I need to buy eggs",
      "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
      "name": "Shopping List",
      "when": "Friday",
      "where": "Home"
    }
  }
}
```

> **Note**
>
> 查询仅返回您指定的字段。您可以从返回字段中删除不需要的字段，以取消选择这些字段：
>
> ```
> {
>     description
>     id
>     name
>     when
>     where
>   }
> ```
>
> 在资源管理器选项卡中，您也可以取消选中要删除的字段旁边的框。

7. 您还可以尝试 listTodos 操作，方法是重复在数据源中创建条目的步骤，然后使用 listTodos 操作重复查询步骤。以下是我们添加第二个任务的示例：

```
{
  "createtodoinput": {
    "name": "Second Task",
    "when": "Monday",
    "where": "Home",
    "description": "I need to mow the lawn"
  }
}
```

在调用 listTodos 操作时，它返回旧条目和新条目：

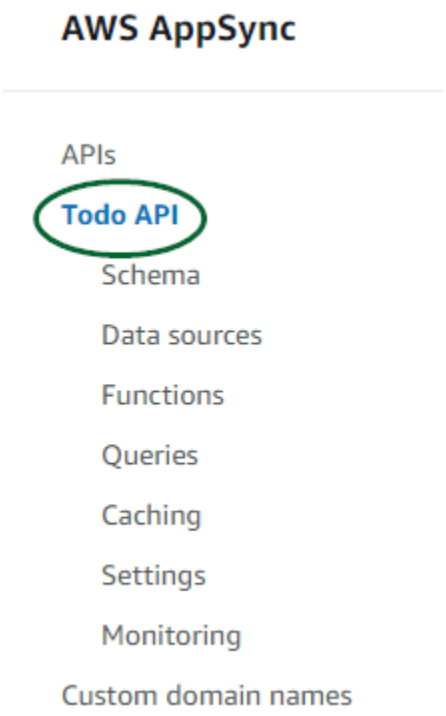
```
{
  "data": {
    "listTodos": {
      "items": [
        {
          "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
          "name": "Shopping List",
          "when": "Friday",
          "where": "Home",
          "description": "I need to buy eggs"
        },
        {
          "id": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",
          "name": "Second Task",
          "when": "Monday",
          "where": "Home",
          "description": "I need to mow the lawn"
        }
      ]
    }
  }
}
```

补充章节

这些小节是更高级 AWS AppSync 主题的参考。我们建议您在执行任何其他操作之前查看补充内容一节。

集成

在控制台选项卡中，如果您选择 API 的名称，则会显示集成页面：



它简要说明了设置 API 的步骤，以及构建客户端应用程序的后续步骤。与您的应用程序集成部分提供了以下详细信息：使用 AWS Amplify 工具链通过配置和代码生成功能自动完成将 API 连接到 iOS、Android 和 JavaScript 应用程序的过程。Amplify 工具链为从本地工作站中构建项目提供全面支持，包括 GraphQL 预置和 CI/CD 工作流。

客户端样本部分还列出用于测试端到端体验的示例客户端应用程序（例如 JavaScript、iOS、Android）。您可以克隆并下载这些示例，配置文件包含您开始使用这些示例所需的信息（例如您的终端节点 URL）。按照 AWS Amplify 工具链页面上的说明运行您的应用程序。

补充内容

- 设计 GraphQL API - 这是使用没有数据源或解析器的空白架构创建 GraphQL 的综合指南。

设计 GraphQL API

AWS AppSync 允许您使用控制台体验创建 GraphQL API。您可以在启动示例架构一节中大致了解该内容。不过，该指南没有显示您可以在 AWS AppSync 中使用的选项和配置的完整目录。

当您选择在控制台中创建 GraphQL API 时，有多个选项可供您使用。如果您按照我们的启动示例架构指南进行操作，我们已向您说明了如何通过预定义的模型创建 API。在以下几节中，我们将指导您在 AWS AppSync 中创建 GraphQL API 所需的其余选项和配置。

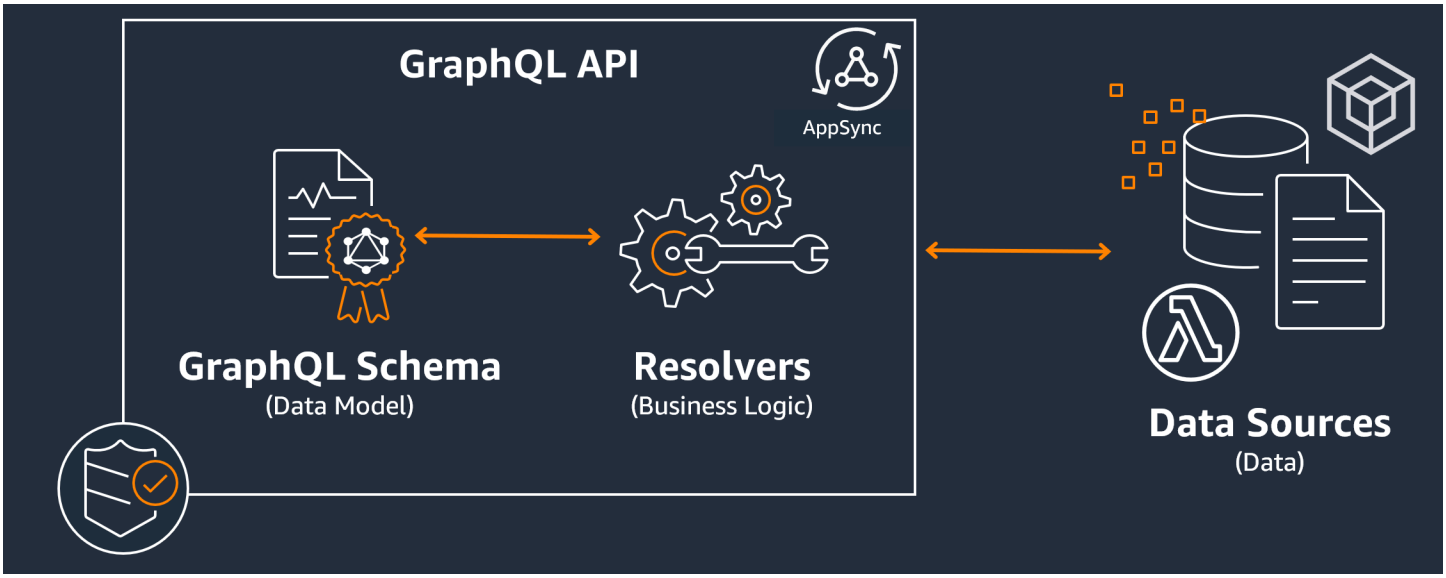
在本节中，您将回顾以下概念：

1. Blank APIs or imports：本指南详细介绍了创建 GraphQL API 的完整过程。您将了解如何通过没有模型的空白模板创建 GraphQL，为架构配置数据源以及将第一个解析器添加到字段中。
2. Real-time data：本指南说明使用 AWS AppSync 的 WebSocket 引擎创建 API 的可能选项。
3. Merged APIs：本指南说明如何关联和合并来自多个现有 GraphQL API 的数据以创建新的 GraphQL API。
4. the section called “RDS 自省”：本指南将向您展示如何使用数据 API 集成 Amazon RDS 表。

构建 GraphQL API（空白或导入的 API）

在通过空白模板创建 GraphQL API 之前，回顾一下有关 GraphQL 的概念会有所帮助。GraphQL API 具有三个基本组件：

1. 架构是包含数据形状和定义的文件。在客户端向您的 GraphQL 服务发出请求时，返回的数据将遵循架构规范。有关更多信息，请参阅架构。
2. 数据源附加到您的架构。在发出请求时，这是检索和修改数据的地方。有关更多信息，请参阅Data sources。
3. 解析器位于架构和数据源之间。在发出请求时，解析器对来自数据源的数据执行操作，然后返回结果以作为响应。有关更多信息，请参阅Resolvers。



AWS AppSync 允许您创建、编辑和存储架构和解析器代码以管理您的 API。您的数据源来自外部存储库，例如数据库、DynamoDB 表和 Lambda 函数。如果您使用 AWS 服务存储数据或计划这样做，在将您的 AWS 账户中的数据与 GraphQL API 关联时，AWS AppSync 可以提供近乎无缝的体验。

在下一节中，您将了解如何使用 AWS AppSync 服务创建其中的每个组件。

主题

步骤 1：设计您的架构

GraphQL 架构是任何 GraphQL 服务器实施的基础。每个 GraphQL API 由单个架构定义，其中包含描述如何填充请求中的数据的类型和字段。必须根据架构验证流经 API 的数据和执行的操作。

一般来说，GraphQL 类型系统描述 GraphQL 服务器的功能，并用于确定查询是否有效。服务器的类型系统通常称为该服务器的架构，可以由不同的对象类型、标量类型、输入类型等组成。GraphQL 既是声明性的，又是强类型的，这意味着将在运行时明确定义类型，并且仅返回指定的内容。

AWS AppSync 允许您定义和配置 GraphQL 架构。以下几节介绍了如何使用 AWS AppSync 的服务从头开始创建 GraphQL 架构。

构建 GraphQL 架构

> **Tip**
>
> 我们建议在继续之前查看架构一节。

GraphQL 是一种实施 API 服务的强大工具。根据 GraphQL 网站，GraphQL 定义如下：

GraphQL 是 API 的查询语言，也是使用现有数据完成这些查询的运行时环境。GraphQL 为 API 中的数据提供完整且易于理解的描述，使客户能够准确询问他们需要的内容而不包含多余信息，从而更轻松地随着时间的推移改进 API 并构建强大的开发工具。

本节介绍了 GraphQL 实施的第一部分，即架构。根据上面的引述，架构扮演“为 API 中的数据提供完整且易于理解的描述”角色。换句话说，GraphQL 架构是您的服务的数据、操作以及它们之间的关系的文本表示形式。架构被视为 GraphQL 服务实施的主要入口点。毫不奇怪，它通常是您在项目中首先实施的内容之一。我们建议在继续之前查看架构一节。

引用架构一节的内容，GraphQL 架构是使用架构定义语言 (SDL) 编写的。SDL 由具有既定结构的类型和字段组成：

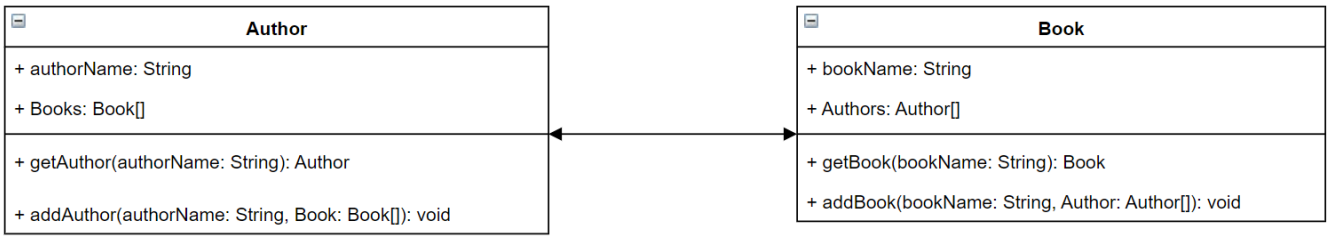
- 类型：类型是 GraphQL 定义数据形状和行为的方式。GraphQL 支持多种类型，本节后面将介绍这些类型。架构中定义的每种类型将包含自己的范围。在该范围内具有一个或多个字段，这些字段可以包含在 GraphQL 服务中使用的值或逻辑。类型扮演很多不同的角色，最常见的角色是对象或标量（基元值类型）。
- 字段：字段位于类型范围内，并保存从 GraphQL 服务中请求的值。它们与其他编程语言中的变量非常相似。您在字段中定义的数据的形状决定了如何在请求/响应操作中设置数据结构。这样，开发人员就可以在不知道服务后端实施方式的情况下预测返回的内容。

最简单的架构包含三种不同的数据类别：

1. 架构根：根定义架构的入口点。它指向对数据执行某种操作（例如添加、删除或修改某些内容）的字段。
2. 类型：这些是用于表示数据形状的基本类型。您几乎可以将它们视为具有定义的特征的事物的对象或抽象表示形式。例如，您可以创建 `Person` 对象以表示数据库中的某个人。每个人的特征将在 `Person` 中定义为字段。它们可能是这个人的姓名、年龄、工作、地址等任何内容。

3. 特殊对象类型：这些是在架构中定义操作行为的类型。每种特殊对象类型在每个架构中定义一次。它们先放置在架构根中，然后在架构正文中进行定义。特殊对象类型中的每个字段定义解析器实施的单个操作。

为了便于理解，假设您正在创建一个服务以存储作者及其所写的书籍。每个作者具有姓名和他们撰写的一系列书籍。每本书具有名称和相关的作者列表。我们还希望能够添加或检索书籍和作者。这种关系的简单 UML 表示形式可能如下所示：



在 GraphQL 中，Author 和 Book 实体表示架构中的两种不同的对象类型：

```
type Author {
}

type Book {
}
```

Author 包含 authorName 和 Books，而 Book 包含 bookName 和 Authors。这些可以表示为类型范围内的字段：

```
type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}
```

正如您看到的一样，类型表示形式与图表非常接近。不过，这些方法可能会变得有些复杂。它们将作为字段放置在几种特殊对象类型之一中。它们的特殊对象分类取决于它们的行为。GraphQL 包含三种基本的特殊对象类型：查询、变更和订阅。有关更多信息，请参阅特殊对象。

由于 getAuthor 和 getBook 都请求数据，因此，它们将放置在 Query 特殊对象类型中：

```
type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}

type Query {
  getAuthor(authorName: String): Author
  getBook(bookName: String): Book
}
```

这些操作链接到查询，而查询本身链接到架构。添加架构根会将特殊对象类型（该示例中的 Query）定义为入口点之一。可以使用 schema 关键字完成该操作：

```
schema {
  query: Query
}

type Author {
  authorName: String
  Books: [Book]
}

type Book {
  bookName: String
  Authors: [Author]
}

type Query {
  getAuthor(authorName: String): Author
  getBook(bookName: String): Book
}
```

看一下最后两个方法（addAuthor 和 addBook），它们在数据库中添加数据，因此，它们是在 Mutation 特殊对象类型中定义的。不过，从类型页面中，我们还知道不允许直接引用对象的输入，因为它们严格来说是输出类型。在这种情况下，我们不能使用 Author 或 Book，因此，我们需要创建一种具有相同字段的输入类型。在该示例中，我们添加了 AuthorInput 和 BookInput，它们接受相应类型的相同字段。然后，我们将输入作为参数以创建变更：

```
schema {
  query: Query
  mutation: Mutation
}

type Author {
  authorName: String
  Books: [Book]
}

input AuthorInput {
  authorName: String
  Books: [BookInput]
}

type Book {
  bookName: String
  Authors: [Author]
}

input BookInput {
  bookName: String
  Authors: [AuthorInput]
}

type Query {
  getAuthor(authorName: String): Author
  getBook(bookName: String): Book
}

type Mutation {
  addAuthor(input: [BookInput]): Author
  addBook(input: [AuthorInput]): Book
}
```

让我们回顾一下我们刚刚执行的操作：

1. 我们创建了一个具有 Book 和 Author 类型的架构以表示我们的实体。
2. 我们添加了包含实体特性的字段。
3. 我们添加了一个查询，以从数据库中检索该信息。
4. 我们添加了一个变更以处理数据库中的数据。
5. 我们添加了输入类型以在变更中替换对象参数，从而符合 GraphQL 的规则。
6. 我们将查询和变更添加到根架构中，以使 GraphQL 实施了解根类型位置。

正如您看到的一样，创建架构的过程通常采用数据建模（尤其是数据库建模）中的一些概念。您可以将架构视为适合源数据的形状。它还充当解析器实施的模型。在以下几节中，您将了解如何使用 AWS 支持的各种工具和服务创建架构。

> **Note**
>
> 以下几节中的示例并不表示在实际应用程序中运行。它们仅用于说明这些命令，以使您可以构建自己的应用程序。

创建架构

您的架构将位于名为 `schema.graphql` 的文件中。AWSAppSync 允许用户使用各种方法为其 GraphQL API 创建新架构。在该示例中，我们将创建一个空白 API 以及空白架构。

Console

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。

   a. 在控制面板中，选择创建 API。
   b. 在 API 选项下面，选择 GraphQL API，选择从头开始设计，然后选择下一步。

      i. 对于 API 名称，将预填充的名称更改为您的应用程序所需的名称。
      ii. 对于联系信息，您可以输入联系人以指定 API 的管理员。此为可选字段。
      iii. 在私有 API 配置下面，您可以启用私有 API 功能。只能从配置的 VPC 终端节点 (VPCE) 中访问私有 API。有关更多信息，请参阅私有 API。

      对于该示例，我们不建议启用该功能。在检查您的输入后，选择下一步。
   c. 在创建 GraphQL 类型下面，您可以选择创建 DynamoDB 表以用作数据源，或者跳过该步骤并稍后执行。

对于该示例，请选择稍后创建 GraphQL 资源。我们将在单独的章节中创建资源。

d. 检查您的输入，然后选择创建 API。

2. 将进入您的特定 API 的控制面板。由于该 API 的名称位于控制面板顶部，因此，您可以看出这一点。如果不是这样，您可以在侧边栏中选择 API，然后在 API 控制面板中选择您的 API。

   - 在侧边栏中，在您的 API 名称下面选择架构。

3. 在架构编辑器中，您可以配置您的 `schema.graphql` 文件。它可能是空的，也可能填充了通过模型生成的类型。右侧是解析器部分，用于将解析器附加到您的架构字段。我们不会在本节中介绍解析器内容。

CLI

> **Note**
>
> 在使用 CLI 时，请确保您具有正确权限以在该服务中访问和创建资源。您可能希望为需要访问该服务的非管理员用户设置最低权限策略。有关 AWS AppSync 策略的更多信息，请参阅适用于 AWS AppSync 的 Identity and Access Management。
> 此外，如果您还没有查看控制台版本，我们建议您先查看该版本。

1. 如果您尚未安装 AWS CLI 并添加您的配置，请执行该操作。
2. 运行 `create-graphql-api` 命令以创建 GraphQL API 对象。

   您需要为该特定命令键入两个参数：

   1. 您的 API 的 `name`。
   2. `authentication-type` 或用于访问该 API 的凭证类型（IAM、OIDC 等）。

   > **Note**
   >
   > 必须配置其他参数（例如 `Region`），但这些参数通常默认为您的 CLI 配置值。

   示例命令可能如下所示：

```
aws appsync create-graphql-api --name testAPI123 --authentication-type API_KEY
```

将在 CLI 中返回输出。示例如下：

```
{
    "graphqlApi": {
        "xrayEnabled": false,
        "name": "testAPI123",
        "authenticationType": "API_KEY",
        "tags": {},
        "apiId": "abcdefghijklmnopqrstuvwxyz",
        "uris": {
            "GRAPHQL": "https://zyxwvutsrqponmlkjihgfedcba.appsync-api.us-
west-2.amazonaws.com/graphql",
            "REALTIME": "wss://zyxwvutsrqponmlkjihgfedcba.appsync-realtime-
api.us-west-2.amazonaws.com/graphql"
        },
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz"
    }
}
```

3.

> **Note**
>
> 这是一个可选命令，它采用现有的架构并使用 Base64 Blob 将其上传到 AWS AppSync 服务中。对于该示例，我们不会使用该命令。

运行 start-schema-creation 命令。

您需要为该特定命令键入两个参数：

1. 上一步中的 `api-id`。
2. 架构 `definition` 是一个 Base64 编码的二进制 Blob。

示例命令可能如下所示：

```
 aws appsync start-schema-creation --api-id abcdefghijklmnopqrstuvwxyz --
definition "aa1111aa-123b-2bb2-c321-12hgg76cc33v"
```

将返回输出：

```
{
    "status": "PROCESSING"
}
```

该命令不会在处理后返回最终输出。您必须使用单独的命令 (get-schema-creation-status) 查看结果。请注意，这两个命令是异步的，因此，即使仍在创建架构，您也可以检查输出状态。

CDK

> **Tip**
>
> 在使用 CDK 之前，我们建议您查看 CDK 的官方文档以及 AWS AppSync 的 CDK 参考。下面列出的步骤仅显示用于添加特定资源的一般代码片段示例。这并不意味着，它是您的生产代码中的有效解决方案。我们还假设您已具有正常工作的应用程序。

1. CDK 的起点有所不同。理想情况下，应该已创建了您的 `schema.graphql` 文件。您只需创建一个具有 `.graphql` 文件扩展名的新文件。它可以是空文件。
2. 一般来说，您可能需要将 import 指令添加到您使用的服务中。例如，它可能采用以下形式：

```
import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'
import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
```

要添加 GraphQL API，您的堆栈文件需要导入 AWS AppSync 服务：

```
import * as appsync from 'aws-cdk-lib/aws-appsync';
```

> **Note**
>
> 这意味着我们使用 `appsync` 关键字导入整个服务。要在您的应用程序中使用该服务，您的 AWS AppSync 构造将使用 `appsync.construct_name` 格式。例如，如果我们要创建 GraphQL API，我们将使用 `new appsync.GraphqlApi(args_go_here)`。以下步骤介绍了这一点。

3. 最基本的 GraphQL API 将包括 API 的 name 和 schema 路径。

```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {
  name: 'name_of_API_in_console',
  schema: appsync.SchemaFile.fromAsset(path.join(__dirname,
 'schema_name.graphql')),
});
```

> **Note**
>
> 让我们回顾一下该代码片段执行的操作。在 api 范围内，我们调用 appsync.GraphqlApi(scope: Construct, id: string, props: GraphqlApiProps) 以创建一个新的 GraphQL API。范围是 this，它指的是当前对象。ID 是 *API_ID*，这是在创建您的 GraphQL API 时在 AWS CloudFormation 中表示该 API 的资源名称。GraphqlApiProps 包含 GraphQL API 的 name 和 schema。schema 搜索 .graphql 文件 (*schema_name.graphql*) 的绝对路径 (__dirname) 以生成架构 (SchemaFile.fromAsset)。在实际场景中，您的架构文件可能位于 CDK 应用程序内。
> 要使用对 GraphQL API 所做的更改，您必须重新部署该应用程序。

在架构中添加类型

现已添加了架构，您可以开始添加输入和输出类型。请注意，不应在实际代码中使用此处的类型；它们只是帮助您理解该过程的示例。

首先，我们创建一个对象类型。在实际代码中，您不必从这些类型开始。只要遵循 GraphQL 的规则和语法，您可以随时创建所需的任何类型。

> **Note**
>
> 接下来的几节将使用架构编辑器，因此，请将其保持打开状态。

Console

- 您可以使用 type 关键字和类型名称以创建对象类型：

```
type Type_Name_Goes_Here {}
```

在类型的范围内，您可以添加表示对象特性的字段：

```
type Type_Name_Goes_Here {
  # Add fields here
}
```

示例如下：

```
type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}
```

> **Note**
>
> 在该骤中，我们添加了一个通用对象类型，将 id 必填字段存储为 ID，将 title 字段存储为 String，并将 date 字段存储为 AWSDateTime。要查看类型和字段列表及其用途，请参阅架构。要查看标量列表及其用途，请参阅类型参考。

CLI

> **Note**
>
> 如果您还没有查看控制台版本，我们建议您先查看该版本。

- 您可以运行 create-type 命令以创建对象类型。

  您需要为该特定命令输入一些参数：

  1. 您的 API 的 api-id。
  2. definition 或您的类型内容。在控制台示例中，这是：

     ```
     type Obj_Type_1 {
     ```

```
  id: ID!
  title: String
  date: AWSDateTime
}
```

3. 您的输入的 format。在该示例中，我们使用 SDL。

示例命令可能如下所示：

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type
 Obj_Type_1{id: ID! title: String date: AWSDateTime}" --format SDL
```

将在 CLI 中返回输出。示例如下：

```
{
    "type": {
        "definition": "type Obj_Type_1{id: ID! title: String date:
 AWSDateTime}",
        "name": "Obj_Type_1",
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Obj_Type_1",
        "format": "SDL"
    }
}
```

> **Note**
>
> 在该骤中，我们添加了一个通用对象类型，将 id 必填字段存储为 ID，将 title 字段存储为 String，并将 date 字段存储为 AWSDateTime。要查看类型和字段列表及其用途，请参阅架构。要查看标量列表及其用途，请参阅类型参考。
> 此外，您可能已意识到，直接输入定义适用于较小的类型，但对于添加较大类型或多个类型是不可行的。您可以选择将所有内容添加到 .graphql 文件中，然后将其作为输入传递。

CDK

> **Tip**
>
> 在使用 CDK 之前，我们建议您查看 CDK 的官方文档以及 AWS AppSync 的 CDK 参考。下面列出的步骤仅显示用于添加特定资源的一般代码片段示例。这并不意味着，它是您的生产代码中的有效解决方案。我们还假设您已具有正常工作的应用程序。

要添加类型，您需要将其添加到您的 `.graphql` 文件中。例如，控制台示例是：

```
type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}
```

您可以像任何其他文件一样将类型直接添加到架构中。

> **Note**
>
> 要使用对 GraphQL API 所做的更改，您必须重新部署该应用程序。

对象类型具有标量类型的字段，例如字符串和整数。AWS除了基本 GraphQL 标量以外，AppSync 还允许您使用增强的标量类型，例如 `AWSDateTime`。此外，任何以感叹号结尾的字段都是必填字段。

特别是，`ID` 标量类型是唯一标识符，可以是 `String` 或 `Int`。您可以在解析器代码中控制这些内容以自动进行分配。

特殊对象类型（如 `Query`）和“常规”对象类型（如上面的示例）之间存在相似之处，因为它们都使用 `type` 关键字并被视为对象。不过，对于特殊对象类型（`Query`、`Mutation` 和 `Subscription`），它们的行为有很大不同，因为它们是作为 API 的入口点公开的。它们更多地涉及设置形状操作而不是数据。有关更多信息，请参阅查询和变更类型。

对于特殊对象类型主题，下一步可能是添加一个或多个对象类型以对设置形状的数据执行操作。在实际场景中，每个 GraphQL 架构必须至少具有一个根查询类型以请求数据。您可以将查询视为 GraphQL 服务器的入口点（或终端节点）之一。让我们添加一个查询以作为示例。

Console

- 要创建查询，您只需将其添加到架构文件中，就像任何其他类型一样。查询需要具有 Query 类型并在根中具有一个条目，如下所示：

```
schema {
  query: Name_of_Query
}

type Name_of_Query {
  # Add field operation here
}
```

请注意，在大多数情况下，生产环境中的 *Name_of_Query* 简称为 Query。我们建议保留该值。在查询类型中，您可以添加字段。每个字段都会在请求中执行一个操作。因此，大多数（即使不是全部）字段将附加到一个解析器。不过，我们在本节中并不关注这个问题。对于字段操作格式，它可能如下所示：

```
Name_of_Query(params): Return_Type # version with params
Name_of_Query: Return_Type # version without params
```

示例如下：

```
schema {
  query: Query
}

type Query {
  getObj: [Obj_Type_1]
}

type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}
```

> **Note**
>
> 在该步骤中，我们添加了一个 Query 类型，并在 schema 根中定义该类型。我们的 Query 类型定义了一个 getObj 字段，该字段返回 Obj_Type_1 对象列表。请注意，Obj_Type_1 是上一步中的对象。在生产代码中，您的字段操作通常处理由 Obj_Type_1 等对象设置形状的数据。此外，getObj 等字段通常具有一个解析器以执行业务逻辑。将在另一节中介绍该内容。
> 另外，AWS AppSync 在导出期间自动添加架构根，因此从技术上讲，您不必将其直接添加到架构中。我们的服务自动处理重复的架构。我们在此处添加架构根以作为最佳实践。

CLI

> **Note**
>
> 如果您还没有查看控制台版本，我们建议您先查看该版本。

1. 运行 create-type 命令以创建一个具有 query 定义的 schema 根。

   您需要为该特定命令输入一些参数：

   1. 您的 API 的 api-id。
   2. definition 或您的类型内容。在控制台示例中，这是：

      ```
      schema {
        query: Query
      }
      ```

   3. 您的输入的 format。在该示例中，我们使用 SDL。

   示例命令可能如下所示：

   ```
   aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "schema
    {query: Query}" --format SDL
   ```

将在 CLI 中返回输出。示例如下：

```
{
    "type": {
        "definition": "schema {query: Query}",
        "name": "schema",
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/schema",
        "format": "SDL"
    }
}
```

Note

请注意，如果您在 `create-type` 命令中未正确输入某些内容，您可以运行 [update-type](#) 命令以更新您的架构根（或架构中的任何类型）。在该示例中，我们暂时更改架构根以包含 `subscription` 定义。
您需要为该特定命令输入一些参数：

1. 您的 API 的 `api-id`。
2. 您的类型的 `type-name`。在控制台示例中，这是 `schema`。
3. `definition` 或您的类型内容。在控制台示例中，这是：

   ```
   schema {
     query: Query
   }
   ```

   添加 `subscription` 后的架构如下所示：

   ```
   schema {
     query: Query
     subscription: Subscription
   }
   ```

4. 您的输入的 `format`。在该示例中，我们使用 SDL。

示例命令可能如下所示：

```
aws appsync update-type --api-id abcdefghijklmnopqrstuvwxyz --type-name
 schema --definition "schema {query: Query subscription: Subscription}"
 --format SDL
```

将在 CLI 中返回输出。示例如下：

```
{
    "type": {
        "definition": "schema {query: Query subscription: Subscription}",
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/schema",
        "format": "SDL"
    }
}
```

在该示例中，添加预设置格式的文件仍然有效。

2. 运行 create-type 命令以创建一个 Query 类型。

您需要为该特定命令输入一些参数：

1. 您的 API 的 api-id。
2. definition 或您的类型内容。在控制台示例中，这是：

```
type Query {
  getObj: [Obj_Type_1]
}
```

3. 您的输入的 format。在该示例中，我们使用 SDL。

示例命令可能如下所示：

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type
 Query {getObj: [Obj_Type_1]}" --format SDL
```

将在 CLI 中返回输出。示例如下：

```
{
    "type": {
```

```
        "definition": "Query {getObj: [Obj_Type_1]}",
        "name": "Query",
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Query",
        "format": "SDL"
    }
}
```

> **Note**
>
> 在该步骤中，我们添加了一个 Query 类型，并在 schema 根中定义该类型。我们的 Query 类型定义了一个 getObj 字段，该字段返回 Obj_Type_1 对象列表。
> 在 schema 根代码 query: Query 中，query: 部分指示在您的架构中定义了一个查询，而 Query 部分指示实际的特殊对象名称。

CDK

> **Tip**
>
> 在使用 CDK 之前，我们建议您查看 CDK 的官方文档以及 AWS AppSync 的 CDK 参考。下面列出的步骤仅显示用于添加特定资源的一般代码片段示例。这并不意味着，它是您的生产代码中的有效解决方案。我们还假设您已具有正常工作的应用程序。

您需要将查询和架构根添加到 .graphql 文件中。我们的示例与以下示例类似，但您希望将其替换为实际的架构代码：

```
schema {
  query: Query
}

type Query {
  getObj: [Obj_Type_1]
}

type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
```

```
}
```

您可以像任何其他文件一样将类型直接添加到架构中。

> **Note**
>
> 更新架构根是可选的。我们在该示例中添加架构根以作为最佳实践。
> 要使用对 GraphQL API 所做的更改，您必须重新部署该应用程序。

您现已看到创建对象和特殊对象（查询）的示例。您还了解了这些对象如何相互关联以描述数据和操作。您可以具有仅包含数据描述以及一个或多个查询的架构。不过，我们希望添加另一个操作以将数据添加到数据源中。我们将添加另一个名为 `Mutation` 的特殊对象类型以修改数据。

Console

- 一个变更命名为 `Mutation`。与 `Query` 一样，`Mutation` 中的字段操作描述一个操作并附加到一个解析器。另请注意，我们需要在 `schema` 根中定义该变更，因为它是一个特殊对象类型。下面是一个变更示例：

  ```
  schema {
    mutation: Name_of_Mutation
  }

  type Name_of_Mutation {
    # Add field operation here
  }
  ```

  像查询一样，将在根中列出典型的变更。变更是使用 `type` 关键字以及名称定义的。*`Name_of_Mutation`* 通常命名为 `Mutation`，因此，我们建议保留这种命名方式。每个字段还会执行一个操作。对于字段操作格式，它可能如下所示：

  ```
  Name_of_Mutation(params): Return_Type # version with params
  Name_of_Mutation: Return_Type # version without params
  ```

  示例如下：

  ```
  schema {
    query: Query
  ```

```
  mutation: Mutation
}

type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}

type Query {
  getObj: [Obj_Type_1]
}

type Mutation {
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
}
```

Note

在该步骤中，我们添加了一个具有 addObj 字段的 Mutation 类型。让我们简要说明一下该字段的用途：

```
addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
```

addObj 使用 Obj_Type_1 对象以执行操作。由于这些字段，这是显而易见的，但语法在： Obj_Type_1 返回类型中证实了这一点。在 addObj 中，它接受 Obj_Type_1 对象中的 id、title 和 date 字段以作为参数。正如您看到的一样，它看起来很像方法声明。不过，我们还没有介绍我们的方法的行为。正如前面所述，架构仅用于定义数据和操作是什么，而不定义它们的工作方式。在我们稍后创建第一个解析器时，将实施实际的业务逻辑。

在完成架构后，可以选择将其导出为 schema.graphql 文件。在架构编辑器中，您可以选择导出架构以使用支持的格式下载该文件。

另外，AWS AppSync 在导出期间自动添加架构根，因此从技术上讲，您不必将其直接添加到架构中。我们的服务自动处理重复的架构。我们在此处添加架构根以作为最佳实践。

CLI

> **Note**
>
> 如果您还没有查看控制台版本，我们建议您先查看该版本。

1. 运行 update-type 命令以更新您的根架构。

   您需要为该特定命令输入一些参数：

   1. 您的 API 的 `api-id`。
   2. 您的类型的 `type-name`。在控制台示例中，这是 `schema`。
   3. `definition` 或您的类型内容。在控制台示例中，这是：

      ```
      schema {
        query: Query
        mutation: Mutation
      }
      ```

   4. 您的输入的 `format`。在该示例中，我们使用 SDL。

   示例命令可能如下所示：

   ```
   aws appsync update-type --api-id abcdefghijklmnopqrstuvwxyz --type-name schema
    --definition "schema {query: Query mutation: Mutation}" --format SDL
   ```

   将在 CLI 中返回输出。示例如下：

   ```
   {
       "type": {
           "definition": "schema {query: Query mutation: Mutation}",
           "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
   abcdefghijklmnopqrstuvwxyz/types/schema",
           "format": "SDL"
       }
   }
   ```

2. 运行 create-type 命令以创建一个 `Mutation` 类型。

您需要为该特定命令输入一些参数：

1. 您的 API 的 `api-id`。
2. `definition` 或您的类型内容。在控制台示例中，这是：

```
type Mutation {
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
}
```

3. 您的输入的 `format`。在该示例中，我们使用 SDL。

示例命令可能如下所示：

```
aws appsync create-type --api-id abcdefghijklmnopqrstuvwxyz --definition "type
 Mutation {addObj(id: ID! title: String date: AWSDateTime): Obj_Type_1}" --
format SDL
```

将在 CLI 中返回输出。示例如下：

```
{
    "type": {
        "definition": "type Mutation {addObj(id: ID! title: String date:
 AWSDateTime): Obj_Type_1}",
        "name": "Mutation",
        "arn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Mutation",
        "format": "SDL"
    }
}
```

CDK

> **Tip**
>
> 在使用 CDK 之前，我们建议您查看 CDK 的官方文档以及 AWS AppSync 的 CDK 参考。下面列出的步骤仅显示用于添加特定资源的一般代码片段示例。这并不意味着，它是您的生产代码中的有效解决方案。我们还假设您已具有正常工作的应用程序。

您需要将查询和架构根添加到 .graphql 文件中。我们的示例与以下示例类似，但您希望将其替换为实际的架构代码：

```
schema {
  query: Query
  mutation: Mutation
}

type Obj_Type_1 {
  id: ID!
  title: String
  date: AWSDateTime
}

type Query {
  getObj: [Obj_Type_1]
}

type Mutation {
  addObj(id: ID!, title: String, date: AWSDateTime): Obj_Type_1
}
```

> **Note**
>
> 更新架构根是可选的。我们在该示例中添加架构根以作为最佳实践。
> 要使用对 GraphQL API 所做的更改，您必须重新部署该应用程序。

可选注意事项 - 将枚举作为状态

至此，您知道如何创建基本架构了。不过，您可以添加很多内容以增加架构的功能。在应用程序中，一种常见情况是将枚举作为状态。您可以使用枚举在调用时强制从一组值中选择一个特定的值。对于您知道在很长一段时间内不会发生显著变化的内容，这是非常有用的。假设来说，我们可以添加一个枚举以在响应中返回状态代码或字符串。

例如，假设我们创建一个社交媒体应用程序，该应用程序在后端存储用户的文章数据。我们的架构包含一个 Post 类型，它表示一篇文章的数据：

```
type Post {
  id: ID!
  title: String
```

```
  date: AWSDateTime
  poststatus: PostStatus
}
```

我们的 Post 将包含唯一的 id、文章 title、发布 date 以及名为 PostStatus 的枚举，它表示应用程序处理时的文章状态。对于我们的操作，我们使用一个查询以返回所有文章数据：

```
type Query {
  getPosts: [Post]
}
```

我们还使用一个变更以将文章添加到数据源中：

```
type Mutation {
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post
}
```

看一下我们的架构，PostStatus 枚举可能具有多种状态。我们可能需要三种基本状态，分别命名为 success（已成功处理文章）、pending（正在处理文章）和 error（无法处理文章）。要添加枚举，我们可以编写以下代码：

```
enum PostStatus {
  success
  pending
  error
}
```

完整架构可能如下所示：

```
schema {
  query: Query
  mutation: Mutation
}

type Post {
  id: ID!
  title: String
  date: AWSDateTime
  poststatus: PostStatus
}
```

```
type Mutation {
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post
}

type Query {
  getPosts: [Post]
}

enum PostStatus {
  success
  pending
  error
}
```

如果用户在应用程序中添加 Post，将调用 addPost 操作以处理该数据。在附加到 addPost 的解析器处理数据时，它不断使用操作状态更新 poststatus。在查询时，Post 将包含数据的最终状态。请记住，我们只是介绍我们希望如何在架构中处理数据。我们对解析器实施进行了很多假设，这些解析器实施处理数据以完成请求的实际业务逻辑。

可选注意事项 - 订阅

AWS AppSync 中的订阅是作为变更响应调用的。您可使用架构中的Subscription 类型和 @aws_subscribe() 指令进行配置，以指定哪些变更会调用一个或多个订阅。有关配置订阅的更多信息，请参阅实时数据。

可选的注意事项 - 关系和分页

假设您在 DynamoDB 表中存储了一百万个 Posts，并且您希望返回其中的一些数据。不过，上面给出的示例查询仅返回所有文章。您不希望每次发出请求时获取所有这些文章。相反，您可能希望对它们进行分页。请对您的架构进行以下改动：

- 在 getPosts 字段中，添加两个输入参数：nextToken（迭代器）和 limit（迭代限制）。
- 添加一个新的 PostIterator 类型，其中包含 Posts（检索 Post 对象列表）和 nextToken（迭代器）字段。
- 更改 getPosts 以使其返回 PostIterator，而不是 Post 对象列表。

```
schema {
  query: Query
  mutation: Mutation
```

```
}

type Post {
  id: ID!
  title: String
  date: AWSDateTime
  poststatus: PostStatus
}

type Mutation {
  addPost(id: ID!, title: String, date: AWSDateTime, poststatus: PostStatus): Post
}

type Query {
  getPosts(limit: Int, nextToken: String): PostIterator
}

enum PostStatus {
  success
  pending
  error
}

type PostIterator {
  posts: [Post]
  nextToken: String
}
```

PostIterator 类型允许您返回 Post 对象列表的一部分，以及用于获取下一部分的 nextToken。在 PostIterator 中，具有一个 Post 项目 ([Post]) 列表，该列表与分页标记 (nextToken) 一起返回。在 AWS AppSync 中，它通过解析器连接到 Amazon DynamoDB，并自动生成为加密标记。它会将 limit 参数的值转换为 maxResults 参数；并将 nextToken 参数转换为 exclusiveStartKey 参数。有关 AWS AppSync 控制台中的示例和内置模板示例，请参阅解析器参考 (JavaScript)。

步骤 2：附加数据源

数据源是位于您的 AWS 账户中并且 GraphQL API 可以与其交互的资源。AWSAppSync 支持多种数据源，例如 AWS Lambda、Amazon DynamoDB、关系数据库 (Amazon Aurora Serverless)、Amazon OpenSearch Service 和 HTTP 终端节点。AWS AppSync API 可以配置为与多个数据源进行交互，以使您能够在单个位置中聚合数据。AWSAppSync 可以使用您的账户中的现有 AWS 资源，或代表您通过架构定义预置 DynamoDB 表。

以下几节说明了如何将数据源附加到 GraphQL API。

数据源类型

您现已在 AWS AppSync 控制台中创建架构，您可以将数据源附加到该架构。在您最初创建 API 时，可以选择在创建预定义的架构期间预置 Amazon DynamoDB 表。不过，我们不会在本节中介绍该选项。您可以在启动架构一节中查看该选项的示例。

相反，我们介绍 AWS AppSync 支持的所有数据源。在为您的应用程序选择正确的解决方案时，需要考虑很多因素。以下几节为每个数据源提供一些额外的上下文。有关数据源的一般信息，请参阅数据源。

Amazon DynamoDB

Amazon DynamoDB 是用于可扩展的应用程序的 AWS 主要存储解决方案之一。DynamoDB 的核心组件是表，它就是一个数据集合。您通常会根据 `Book` 或 `Author` 等实体创建表。表条目信息存储为项目，这些项目是每个条目的唯一字段组。完整项目表示数据库中的一行/一个记录。例如，`Book` 条目的项目可能包括 `title` 和 `author` 及其值。像 `title` 和 `author` 这样的单独字段称为属性，它们类似于关系数据库中的列值。

正如您猜测的一样，将使用表存储您的应用程序中的数据。AWS AppSync 允许您将 DynamoDB 表挂载到 GraphQL API 以处理数据。请从前端 Web 和移动博客中获取该使用案例。该应用程序允许用户注册社交媒体应用程序。用户可以加入组，并上传文章以向订阅该组的其他用户广播。他们的应用程序将用户、文章和用户组信息存储在 DynamoDB 中。GraphQL API（由 AWS AppSync 管理）与 DynamoDB 表进行交互。当用户在系统中进行更改并反映到前端时，GraphQL API 检索这些更改并向其他用户实时广播。

AWS Lambda

Lambda 是一种事件驱动的服务，它自动构建所需的资源以运行代码，从而响应事件。Lambda 使用函数，这些函数是包含用于执行资源的代码、依赖项和配置的组语句。在函数检测到触发器（一组调用函数的活动）时，将自动执行函数。触发器可能是任意内容，例如进行 API 调用的应用程序、您的账户中启动资源的 AWS 服务，等等。在触发时，函数将处理事件，这些事件是包含要修改的数据的 JSON 文档。

Lambda 非常适合运行代码，无需预置资源即可运行。请从前端 Web 和移动博客中获取该使用案例。该使用案例与 DynamoDB 一节中说明的使用案例有点相似。在该应用程序中，GraphQL API 负责定义操作，例如，添加文章（变更）和获取该数据（查询）。为了实施其操作的功能（例如 `getPost ( id: String ! ) : Post`、`getPostsByAuthor ( author: String ! ) : [ Post ]`），它们使用 Lambda 函数处理入站请求。在 Option 2: AWS AppSync with Lambda

resolver 下面，它们使用 AWS AppSync 服务维护其架构，并将 Lambda 数据源链接到其中的一个操作。在调用该操作时，Lambda 与 Amazon RDS 代理交互以对数据库执行业务逻辑。

Amazon RDS

通过使用 Amazon RDS，您可以快速构建和配置关系数据库。在 Amazon RDS 中，您将创建一个通用数据库实例，以作为云中的隔离数据库环境。在该示例中，您使用一个数据库引擎，它是实际的 RDBMS 软件（PostgreSQL、MySQL 等）。该服务使用 AWS 的基础设施和安全服务（如修补和加密）提供可扩展性，从而减少了大部分的后端工作并降低了部署的管理成本。

使用 Lambda 一节中的相同使用案例。在 Option 3: AWS AppSync with Amazon RDS resolver 下面，提供的另一个选项是将 AWS AppSync 中的 GraphQL API 直接链接到 Amazon RDS。通过使用数据 API，他们将数据库与 GraphQL API 相关联。一个解析器附加到字段（通常是查询、变更或订阅），并实施访问数据库所需的 SQL 语句。在客户端发出调用该字段的请求时，解析器执行这些语句并返回响应。

Amazon EventBridge

在 EventBridge 中，您将创建事件总线，它们是一些管道，用于从您附加的服务或应用程序（事件源）中接收事件，并根据一组规则处理事件。事件是执行环境中的某种状态变化，而规则是事件的一组筛选条件。规则遵循一种事件模式或事件状态变化元数据（ID、区域、账号、ARN 等）。在事件与事件模式匹配时，EventBridge 将事件通过管道发送到目标服务（目标），并触发规则中指定的操作。

EventBridge 适合将状态更改操作路由到某种其他服务。请从前端 Web 和移动博客中获取该使用案例。该示例介绍了一个电子商务解决方案，该解决方案具有多个团队以维护不同的服务。其中的一种服务在前端针对每个交付步骤（下订单、处理中、发货、交付等）向客户提供订单更新。不过，管理该服务的前端团队无法直接访问订购系统数据，因为该数据是由单独的后端团队维护的。后端团队的订购系统也被描述为黑匣子，因此，很难收集有关他们如何设置数据结构的信息。不过，后端团队确实设置了一个系统，以通过 EventBridge 管理的事件总线发布订单数据。为了访问来自该事件总线的数据并将其路由到前端，前端团队创建了一个新目标以指向 AWS AppSync 中的 GraphQL API。他们还创建一条规则，以仅发送与订单更新相关的数据。在进行更新时，来自该事件总线的数据将发送到 GraphQL API。API 中的架构处理数据，然后将其传送到前端。

None 数据源

如果不打算使用数据源，您可以将其设置为 none。虽然 none 数据源仍明确归类为数据源，但并不是存储介质。通常，解析器在某一时刻调用一个或多个数据源以处理请求。不过，在某些情况下，您可能不需要处理数据源。如果将数据源设置为 none，将运行请求，跳过数据调用步骤，然后运行响应。

使用 EventBridge 一节中的相同使用案例。在架构中，变更处理状态更新，然后将其发送给订阅者。回想一下解析器的工作方式，通常至少调用一次数据源。不过，事件总线已自动发送本场景中的数据。这

意味着变更不需要执行数据源调用；可以直接在本地处理订单状态。变更设置为 none，它充当传递值而不会调用数据源。然后，使用数据填充架构，该数据将发送给订阅者。

OpenSearch

Amazon OpenSearch Service 是一套用于实施全文搜索、数据可视化和日志记录的工具。可以使用该服务查询您上传的结构化数据。

在该服务中，您创建 OpenSearch 实例。这些实例称为节点。在节点中，您添加至少一个索引。从概念上讲，索引有点像关系数据库中的表。（不过，OpenSearch 不符合 ACID 标准，因此，不应以这种方式使用该服务）。您将使用上传到 OpenSearch Service 的数据填充索引。在上传您的数据时，将使用索引中存在的一个或多个分片对其编制索引。分片就像索引的一个分区，其中包含一些数据，并且可以与其他分片分开查询。在上传后，您的数据结构设置为称为文档的 JSON 文件。然后，您可以查询节点以获取文档中的数据。

HTTP 终端节点

您可以将 HTTP 终端节点作为数据源。AWSAppSync 可以向终端节点发送具有参数和负载等相关信息的请求。将向解析器公开 HTTP 响应，解析器在完成操作后返回最终响应。

添加数据源

如果创建了数据源，您可以将其链接到 AWS AppSync 服务，更具体地说，链接到 API。

Console

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   a. 在控制面板中选择您的 API。
   b. 在侧边栏中，选择数据源。
2. 选择创建数据源。
   a. 命名您的数据源。您也可以为其提供描述，但这是可选的。
   b. 选择您的数据源类型。
   c. 对于 DynamoDB，必须选择您的区域，然后选择该区域中的表。您可以选择创建新的通用表角色或导入表的现有角色，以规定与表交互的规则。您可以启用版本控制，在多个客户端同时尝试更新数据时，该功能可以自动为每个请求创建数据版本。版本控制用于保留和维护多个数据变体，以实现冲突检测和解决目的。您还可以启用自动架构生成功能，该功能获取您的数据源，并生成在架构中访问它所需的一些 CRUD、`List` 和 `Query` 操作。

对于 OpenSearch，必须选择您的区域，然后选择该区域中的域（集群）。您可以选择创建新的通用表角色或导入表的现有角色，以规定与域交互的规则。

对于 Lambda，必须选择您的区域，然后选择该区域中的 Lambda 函数的 ARN。您可以选择创建新的通用表角色或导入表的现有角色，以规定与 Lambda 函数交互的规则。

对于 HTTP，必须输入您的 HTTP 终端节点。

对于 EventBridge，必须选择您的区域，然后选择该区域中的事件总线。您可以选择创建新的通用表角色或导入表的现有角色，以规定与事件总线交互的规则。

对于 RDS，必须选择您的区域，然后选择密钥存储（用户名和密码）、数据库名称和架构。

对于“None”，您将添加一个没有实际数据源的数据源。这是为了在本地处理解析器，而不是通过实际数据源。

> **Note**
>
> 如果要导入现有的角色，它们需要使用信任策略。有关更多信息，请参阅 IAM 信任策略。

3. 选择创建。

> **Note**
>
> 或者，如果要创建 DynamoDB 数据源，您可以转到控制台中的架构页面，选择页面顶部的创建资源，然后填写一个预定义模型以转换为表。在该选项中，您填写或导入基本类型，配置包括分区键在内的基本表数据，并检查架构更改。

CLI

- 运行 `create-data-source` 命令以创建数据源。

  您需要为该特定命令输入一些参数：

  1. 您的 API 的 `api-id`。
  2. 您的表的 `name`。

3. 数据源的 `type`。根据您选择的数据源类型，您可能需要输入 `service-role-arn` 和 `-config` 标签。

示例命令可能如下所示：

```
aws appsync create-data-source --api-id abcdefghijklmnopqrstuvwxyz
--name data_source_name --type data_source_type --service-role-arn
arn:aws:iam::107289374856:role/role_name --[data_source_type]-config {params}
```

CDK

> **Tip**
>
> 在使用 CDK 之前，我们建议您查看 CDK 的官方文档以及 AWS AppSync 的 CDK 参考。下面列出的步骤仅显示用于添加特定资源的一般代码片段示例。这并不意味着，它是您的生产代码中的有效解决方案。我们还假设您已具有正常工作的应用程序。

要添加您的特定数据源，您需要将构造添加到堆栈文件中。可以在此处找到一个数据源类型列表：

- DynamoDbDataSource
- EventBridgeDataSource
- HttpDataSource
- LambdaDataSource
- NoneDataSource
- OpenSearchDataSource
- RdsDataSource

1. 一般来说，您可能需要将 import 指令添加到您使用的服务中。例如，它可能采用以下形式：

```
import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'
import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
```

例如，这是您导入 AWS AppSync 和 DynamoDB 服务的方法：

```
import * as appsync from 'aws-cdk-lib/aws-appsync';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
```

2. 某些服务（例如 RDS）要求在创建数据源之前在堆栈文件中进行一些额外的设置（例如，VPC 创建、角色和访问凭证）。有关更多信息，请参阅相关 CDK 页面中的示例。
3. 对于大多数数据源（尤其是 AWS 服务），您将在堆栈文件中创建新的数据源实例。通常，这会如下所示：

```
const add_data_source_func = new service_scope.resource_name(scope: Construct,
 id: string, props: data_source_props);
```

例如，以下是一个示例 Amazon DynamoDB 表：

```
const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {
  partitionKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
  sortKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
  tableClass: dynamodb.TableClass.STANDARD,
});
```

> **Note**
>
> 大多数数据源至少具有一个必需的属性（不使用？符号表示）。请参阅 CDK 文档以了解需要使用哪些属性。

4. 接下来，您需要将数据源链接到 GraphQL API。建议的方法是，在为管道解析器创建函数时添加数据源。例如，下面的代码片段是一个扫描 DynamoDB 表中的所有元素的函数：

```
const add_func = new appsync.AppsyncFunction(this, 'func_ID', {
  name: 'func_name_in_console',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('data_source_name_in_console',
 add_ddb_table),
  code: appsync.Code.fromInline(`
```

```
      export function request(ctx) {
        return { operation: 'Scan' };
      }

      export function response(ctx) {
        return ctx.result.items;
      }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});
```

在 dataSource 属性中，您可以调用 GraphQL API (add_api)，并使用其内置方法之一 (addDynamoDbDataSource) 在表和 GraphQL API 之间建立关联。参数是该链接在 AWS AppSync 控制台中的名称（该示例中为 data_source_name_in_console）以及在表方法 (add_ddb_table) 中的名称。您在下一节中开始创建解析器，此时，将介绍有关该主题的更多信息。

可以使用多种替代方法链接数据源。从技术上讲，您可以将 api 添加到表函数的属性列表中。例如，以下是步骤 3 中的代码片段，但具有包含 GraphQL API 的 api 属性：

```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {
  ...
});

const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {

 ...

  api: add_api
});
```

或者，您可以单独调用 GraphqlApi 构造：

```
const add_api = new appsync.GraphqlApi(this, 'API_ID', {
  ...
});

const add_ddb_table = new dynamodb.Table(this, 'Table_ID', {
  ...
});
```

```
const link_data_source =
 add_api.addDynamoDbDataSource('data_source_name_in_console', add_ddb_table);
```

我们建议仅在函数的属性中创建关联。否则，您必须在 AWS AppSync 控制台中手动将解析器函数链接到数据源（如果要继续使用控制台值 `data_source_name_in_console`），或者在函数中使用另一个名称（例如 `data_source_name_in_console_2`）以创建单独的关联。这是由于属性处理信息的方式的限制造成的。

> **Note**
>
> 您必须重新部署应用程序才能看到更改。

IAM 信任策略

如果您将现有的 IAM 角色用于数据源，则需要为该角色授予相应的权限，以对您的 AWS 资源执行操作，例如对 Amazon DynamoDB 表执行 `PutItem`。您还需要修改该角色的信任策略，以允许 AWS AppSync 使用该策略访问资源，如以下示例策略所示：

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

您也可以根据需要在信任策略中添加条件，以限制对数据源的访问。目前，可以在这些条件中使用 `SourceArn` 和 `SourceAccount` 键。例如，以下策略仅限账户 `123456789012` 访问数据源：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
```

```
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "123456789012"
        }
      }
    }
  ]
}
```

或者，您可以使用以下策略，仅限特定的 API（例如 abcdefghijklmnopq）访问数据源：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnEquals": {
          "aws:SourceArn": "arn:aws:appsync:us-west-2:123456789012:apis/
abcdefghijklmnopq"
        }
      }
    }
  ]
}
```

您可以使用以下策略，仅限特定区域（例如 us-east-1）中的所有 AWS AppSync API 进行访问：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "appsync.amazonaws.com"
      },
```

```
      "Action": "sts:AssumeRole",
      "Condition": {
        "ArnEquals": {
          "aws:SourceArn": "arn:aws:appsync:us-east-1:123456789012:apis/*"
        }
      }
    }
  ]
}
```

在下一节（配置解析器）中，我们将添加解析器业务逻辑，并将其附加到架构中的字段以处理数据源中的数据。

有关角色策略配置的更多信息，请参阅《IAM 用户指南》中的修改角色。

有关适用于 AWS AppSync 的 AWS Lambda 解析器的跨账户访问的更多信息，请参阅 Building cross-account AWS Lambda resolvers for AWS AppSync。

步骤 3：配置解析器

在前面的章节中，您学习了如何创建 GraphQL 架构和数据源，然后在服务中将它们链接在一起。AWS AppSync 在您的架构中，您可能已在查询和变更中创建了一个或多个字段（操作）。虽然该架构描述了操作从数据来源请求的数据类型，但从未实施这些操作如何处理数据的行为。

操作的行为始终是在解析器中实施的，解析器将链接到执行操作的字段。有关解析器一般如何工作的更多信息，请参阅解析器页面。

在中 AWS AppSync，您的解析器与运行时绑定，运行时是您的解析器执行的环境。运行时系统决定了编写解析器时使用的语言。目前支持两种运行时：APPSYNC_JS (JavaScript) 和 Apache Velocity 模板语言 (VTL)。

在实施解析器时，它们采用通用的结构：

- 预备步骤：在客户端发出请求时，将为使用的架构字段（通常是查询、变更、订阅）的解析器传送请求的数据。解析器开始使用预备步骤处理程序处理请求数据，该处理程序允许在数据传送到解析器之前执行一些预处理操作。
- 函数：在运行预备步骤后，请求传送到函数列表。将对数据来源执行列表中的第一个函数。函数是解析器代码的子集，其中包含自己的请求和响应处理程序。请求处理程序将获取请求数据，并对数据来源执行操作。在将数据来源的响应传回到列表之前，响应处理程序对其进行处理。如果具有多个函数，请求数据将发送到列表中的下一个函数以进行执行。列表中的函数按照开发人员定义的顺序依次执行。在执行所有函数后，最终结果传送到后续步骤。

- 后续步骤：后续步骤是一个处理程序函数，允许您在将最终函数的响应传送到 GraphQL 响应之前对其执行一些最终操作。

该流程是一个管道解析器示例。在两个运行时系统中都支持管道解析器。不过，这仅简要说明了管道解析器的用途。此外，我们仅介绍一种可能的解析器配置。有关支持的解析器配置的更多信息，请参阅 APPSYNC_JS 的JavaScript 解析器概述或 VTL 的解析器映射模板概述。

正如您看到的一样，解析器是模块化的。要使解析器的组件正常工作，它们必须能够从其他组件了解执行状态。从解析器一节中，您知道可以将有关执行状态的重要信息作为一组参数（`args`、`context` 等）传递给解析器中的每个组件。在中 AWS AppSync，这由严格处理`context`。它是一个容器，用于存放有关解析的字段的信息。这可能包括传递的参数、结果、授权数据、标头数据等所有内容。有关上下文的更多信息，请参阅适用于 APPSYNC_JS 的解析器上下文对象参考或适用于 VTL 的解析器映射模板上下文参考。

上下文并不是你可以用来实现解析器的唯一工具。 AWS AppSync 支持各种用于值生成、错误处理、解析、转换等的实用工具。您可以在此处查看 APPSYNC_JS 的实用程序列表，或者在此处查看 VTL 的实用程序列表。

在以下几节中，您将了解如何在 GraphQL API 中配置解析器。

主题

- 配置解析器 (JavaScript)
- 配置解析器 (VTL)

配置解析器 (JavaScript)

GraphQL 解析器将类型的架构中的字段连接到数据源。解析器是用于完成请求的机制。

AWS AppSync 中的解析器使用 JavaScript，将 GraphQL 表达式转换为数据源可使用的格式。或者，可以使用 Apache Velocity 模板语言 (VTL) 编写映射模板，以将 GraphQL 表达式转换为数据源可使用的格式。

本节介绍了如何使用 JavaScript 配置解析器。解析器教程 (JavaScript) 一节提供了有关如何使用 JavaScript 实施解析器的深入教程。解析器参考 (JavaScript) 一节说明了可以与 JavaScript 解析器一起使用的实用程序操作。

我们建议您在尝试使用任何上述教程之前先遵循本指南。

在本节中，我们将介绍如何为查询和变更创建和配置解析器。

> **Note**
>
> 本指南假设您已创建架构并至少具有一个查询或变更。如果您要获取订阅（实时数据），请参阅本指南。

在本节中，我们提供一些配置解析器的常规步骤以及一个使用以下架构的示例：

```
// schema.graphql file

input CreatePostInput {
  title: String
  date: AWSDateTime
}

type Post {
  id: ID!
  title: String
  date: AWSDateTime
}

type Mutation {
  createPost(input: CreatePostInput!): Post
}

type Query {
  getPost: [Post]
}
```

创建基本查询解析器

本节说明了如何创建基本查询解析器。

Console

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   a. 在 API 控制面板中，选择您的 GraphQL API。
   b. 在侧边栏中，选择架构。
2. 输入架构和数据源详细信息。有关更多信息，请参阅设计架构和附加数据源小节。

3. 在架构编辑器旁边，具有一个名为解析器的窗口。该框包含架构窗口中定义的类型和字段列表。您可以将解析器附加到字段。您很可能会将解析器附加到字段操作。在本节中，我们将了解简单的查询配置。在 Query 类型下面，选择您的查询字段旁边的附加。
4. 在附加解析器页面上的解析器类型下面，您可以在管道解析器和单位解析器之间进行选择。有关这些类型的更多信息，请参阅解析器。本指南将使用`pipeline resolvers`。

> **Tip**
>
> 在创建管道解析器时，您的数据源将附加到管道函数。函数是在您创建管道解析器本身之后创建的，这就是为什么在该页面中没有设置数据源的选项。如果您使用单位解析器，则数据源直接绑定到解析器，因此，您可以在该页面中设置数据源。

对于解析器运行时，选择 APPSYNC_JS 以启用 JavaScript 运行时环境。

5. 您可以为该 API 启用缓存。我们建议暂时关闭该功能。选择创建。
6. 在编辑解析器页面上，具有一个名为解析器代码的代码编辑器，可用于实施解析器处理程序和响应的逻辑（预备步骤和后续步骤）。有关更多信息，请参阅 JavaScript 解析器概述。

> **Note**
>
> 在我们的示例中，我们直接将请求保留空白，并将响应设置为从上下文返回最后的数据源结果：
>
> ```
> import {util} from '@aws-appsync/utils';
>
> export function request(ctx) {
>     return {};
> }
>
> export function response(ctx) {
>     return ctx.prev.result;
> }
> ```

在该部分下面，具有有一个名为函数的表。函数用于实施可以在多个解析器中重复使用的代码。您可以将源代码存储为函数以在需要时添加到解析器中，而不是不断重新编写或复制代码。

函数占据了管道的操作列表中的很大部分。在解析器中使用多个函数时，您可以设置函数顺序，将按该顺序运行它们。它们在请求函数运行之后和响应函数开始之前执行。

要添加新函数，请在函数下面选择添加函数，然后选择创建新函数。或者，您可能会看到一个创建函数按钮可供选择。

a. 选择一个数据源。这是解析器处理的数据源。

> **Note**
>
> 在我们的示例中，我们为 `getPost` 附加一个解析器，它按 `id` 检索 `Post` 对象。假设我们已为该架构设置一个 DynamoDB 表。其分区键设置为 `id` 并且为空。

b. 输入一个`Function name`。

c. 在函数代码下面，您需要实施函数的行为。这可能会令人困惑，但每个函数具有自己的本地请求和响应处理程序。先运行请求，然后调用数据源以处理请求，最后由响应处理程序处理数据源响应。结果存储在上下文对象中。然后，运行列表中的下一个函数；如果是最后一个函数，则将其传递给后续步骤响应处理程序。

> **Note**
>
> 在我们的示例中，我们将一个解析器附加到 `getPost`，它从数据源获取 `Post` 对象列表。我们的请求函数从表中请求数据，表将其响应传递给上下文 (ctx)，然后响应在上下文中返回结果。AWS AppSync 的优势在于它与其他 AWS 服务之间的互连。由于我们使用的是 DynamoDB，因此，我们具有一组操作以简化此类操作。我们还具有其他数据源类型的一些样板示例。
> 我们的代码将如下所示：

```
import { util } from '@aws-appsync/utils';

/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  return { operation: 'Scan' };
}

/**
 * return a list of scanned post items
```

```
 */
export function response(ctx) {
  return ctx.result.items;
}
```

在该步骤中，我们添加了两个函数：

- `request`：请求处理程序对数据源执行检索操作。参数包含上下文对象 (`ctx`) 或为执行特定操作的所有解析器提供的一些数据。例如，它可能包含授权数据、解析的字段名称等。返回语句执行 `Scan` 操作（请参阅此处的示例）。由于我们使用的是 DynamoDB，因此，我们可以使用该服务中的一些操作。扫描对表中的所有项目执行基本获取。该操作的结果作为 `result` 容器存储在上下文对象中，然后再传递给响应处理程序。`request` 在管道中的响应之前运行。
- `response`：返回 `request` 输出的响应处理程序。参数是更新的上下文对象，返回语句是 `ctx.prev.result`。在本指南的当前阶段，您可能还不熟悉该值。`ctx` 指的是上下文对象。`prev` 指的是管道中的以前操作，也就是 `request`。`result` 包含在管道中执行解析器的结果。如果将它们放在一起，`ctx.prev.result` 将返回最后执行的操作的结果，即请求处理程序。

d. 在完成后，选择创建。

7. 返回到解析器屏幕，在函数下面选择添加函数下拉列表，然后将您的函数添加到函数列表中。
8. 选择保存以更新解析器。

CLI

添加您的函数

- 使用 `create-function` 命令为管道解析器创建一个函数。

  您需要为该特定命令输入一些参数：

  1. 您的 API 的 `api-id`。
  2. AWS AppSync 控制台中的函数的 `name`。
  3. `data-source-name` 或函数使用的数据源名称。它必须已创建并链接到 AWS AppSync 服务中的 GraphQL API。
  4. `runtime` 或函数的环境和语言。对于 JavaScript，名称必须是 `APPSYNC_JS`，运行时环境必须是 `1.0.0`。

5. code 或函数的请求和响应处理程序。虽然您可以手动键入该内容，但将其添加到 .txt 文件（或类似格式）并作为参数传入要容易得多。

> **Note**
>
> 我们的查询代码将位于作为参数传入的文件中：
>
> ```
> import { util } from '@aws-appsync/utils';
>
> /**
>  * Performs a scan on the dynamodb data source
>  */
> export function request(ctx) {
>   return { operation: 'Scan' };
> }
>
> /**
>  * return a list of scanned post items
>  */
> export function response(ctx) {
>   return ctx.result.items;
> }
> ```

示例命令可能如下所示：

```
aws appsync create-function \
--api-id abcdefghijklmnopqrstuvwxyz \
--name get_posts_func_1 \
--data-source-name table-for-posts \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file://~/path/to/file/{filename}.{fileType}
```

将在 CLI 中返回输出。示例如下：

```
{
    "functionConfiguration": {
        "functionId": "ejglgvmcabdn7lx75ref4qeig4",
        "functionArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/functions/ejglgvmcabdn7lx75ref4qeig4",
```

```
        "name": "get_posts_func_1",
        "dataSourceName": "table-for-posts",
        "maxBatchSize": 0,
        "runtime": {
            "name": "APPSYNC_JS",
            "runtimeVersion": "1.0.0"
        },
        "code": "Code output goes here"
    }
}
```

> **Note**
>
> 确保将 functionId 记录在某处，因为它用于将函数附加到解析器。

创建您的解析器

- 运行 create-resolver 命令，为 Query 创建一个管道函数。

  您需要为该特定命令输入一些参数：

  1. 您的 API 的 api-id。
  2. type-name 或架构中的特殊对象类型（查询、变更、订阅）。
  3. field-name 或要将解析器附加到的特殊对象类型中的字段操作。
  4. kind，它指定单位解析器或管道解析器。请将其设置为 PIPELINE 以启用管道函数。
  5. pipeline-config 或附加到解析器的函数。确保您知道函数的 functionId 值。列表顺序很重要。
  6. runtime，它是 APPSYNC_JS (JavaScript)。runtimeVersion 目前是 1.0.0。
  7. code，它包含预备步骤和后续步骤处理程序。

  > **Note**
  >
  > 我们的查询代码将位于作为参数传入的文件中：

  ```
  import { util } from '@aws-appsync/utils';

  /**
   * Sends a request to `put` an item in the DynamoDB data source
  ```

```
 */
export function request(ctx) {
  const { id, ...values } = ctx.args;
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}

/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}
```

示例命令可能如下所示：

```
aws appsync create-resolver \
--api-id abcdefghijklmnopqrstuvwxyz \
--type-name Query \
--field-name getPost \
--kind PIPELINE \
--pipeline-config functions=ejglgvmcabdn7lx75ref4qeig4 \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}
```

将在 CLI 中返回输出。示例如下：

```
{
    "resolver": {
        "typeName": "Mutation",
        "fieldName": "getPost",
        "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Mutation/resolvers/getPost",
        "kind": "PIPELINE",
        "pipelineConfig": {
            "functions": [
                "ejglgvmcabdn7lx75ref4qeig4"
            ]
```

```
        },
        "maxBatchSize": 0,
        "runtime": {
            "name": "APPSYNC_JS",
            "runtimeVersion": "1.0.0"
        },
        "code": "Code output goes here"
    }
}
```

CDK

> **Tip**
>
> 在使用 CDK 之前，我们建议您查看 CDK 的官方文档以及 AWS AppSync 的 CDK 参考。下面列出的步骤仅显示用于添加特定资源的一般代码片段示例。这并不意味着，它是您的生产代码中的有效解决方案。我们还假设您已具有正常工作的应用程序。

基本应用程序需要使用以下内容：

1. 服务导入指令
2. 架构代码
3. 数据源生成器
4. 函数代码
5. 解析器代码

从设计您的架构和附加数据源小节中，我们知道堆栈文件将包含以下格式的 import 指令：

```
import * as x from 'x'; # import wildcard as the 'x' keyword from 'x-service'
import {a, b, ...} from 'c'; # import {specific constructs} from 'c-service'
```

> **Note**
>
> 在前面的几节中，我们仅说明了如何导入 AWS AppSync 构造。在实际代码中，您必须导入更多服务才能运行应用程序。在我们的示例中，如果我们要创建非常简单的 CDK 应用程

序，我们至少需要导入 AWS AppSync 服务以及我们的数据源（即 DynamoDB 表）。我们还需要导入一些额外的构造以部署应用程序：

```
import * as cdk from 'aws-cdk-lib';
import * as appsync from 'aws-cdk-lib/aws-appsync';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
import { Construct } from 'constructs';
```

简要说明一下每个指令：

- `import * as cdk from 'aws-cdk-lib';`：用于定义 CDK 应用程序和构造，例如堆栈。它还包含一些对我们的应用程序非常有用的实用程序函数，例如处理元数据。如果您熟悉该 import 指令，但想知道为什么此处没有使用 cdk 核心库，请参阅 Migration 页面。
- `import * as appsync from 'aws-cdk-lib/aws-appsync';`：它导入 AWS AppSync 服务。
- `import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';`：它导入 DynamoDB 服务。
- `import { Construct } from 'constructs';`：我们需要使用该指令以定义根构造。

导入类型取决于您调用的服务。我们建议查看 CDK 文档以获取示例。页面顶部的架构是 CDK 应用程序中的单独文件，显示为 `.graphql` 文件。在堆栈文件中，我们可以使用以下格式将其与新的 GraphQL 相关联：

```
const add_api = new appsync.GraphqlApi(this, 'graphQL-example', {
  name: 'my-first-api',
  schema: appsync.SchemaFile.fromAsset(path.join(__dirname, 'schema.graphql')),
});
```

> **Note**
>
> 在 `add_api` 范围内，我们使用 `new` 关键字和后面的 `appsync.GraphqlApi(scope: Construct, id: string , props: GraphqlApiProps)` 添加新的 GraphQL API。我们的范围是 `this`，CFN ID 是 `graphQL-example`，我们的属性是 `my-first-api`（控制台中的 API 的名称）和 `schema.graphql`（架构文件的绝对路径）。

要添加数据源，您必须先将数据源添加到堆栈中。然后，您需要使用源特定的方法将其与 GraphQL API 相关联。在您创建解析器函数时，将会发生关联。同时，让我们使用一个通过 `dynamodb.Table` 创建 DynamoDB 表的示例：

```
const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
  partitionKey: {
    name: 'id',
    type: dynamodb.AttributeType.STRING,
  },
});
```

> **Note**
>
> 如果我们要在示例中使用该表，我们将添加一个 CFN ID 为 `posts-table` 且分区键为 `id (S)` 的新 DynamoDB 表。

接下来，我们需要在堆栈文件中实施解析器。以下是扫描 DynamoDB 表中的所有项目的简单查询示例：

```
const add_func = new appsync.AppsyncFunction(this, 'func-get-posts', {
  name: 'get_posts_func_1',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('table-for-posts', add_ddb_table),
  code: appsync.Code.fromInline(`
      export function request(ctx) {
        return { operation: 'Scan' };
      }

      export function response(ctx) {
        return ctx.result.items;
      }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});

new appsync.Resolver(this, 'pipeline-resolver-get-posts', {
  add_api,
  typeName: 'Query',
  fieldName: 'getPost',
  code: appsync.Code.fromInline(`
```

```
      export function request(ctx) {
        return {};
      }

      export function response(ctx) {
        return ctx.prev.result;
      }
  `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
  pipelineConfig: [add_func],
});
```

Note

首先，我们创建一个名为 add_func 的函数。这种创建顺序可能看起来有点违背常理，但您必须在创建管道解析器本身之前在解析器中创建函数。函数采用以下格式：

```
AppsyncFunction(scope: Construct, id: string, props: AppsyncFunctionProps)
```

我们的范围是 this，CFN ID 是 func-get-posts，属性包含实际函数详细信息。我们在属性中包含：

- 在 AWS AppSync 控制台 (get_posts_func_1) 中显示的函数的 name。
- 我们以前创建的 GraphQL API (add_api)。
- 数据源；我们在其中将数据源链接到 GraphQL API 值，然后将其附加到函数。我们获取创建的表 (add_ddb_table)，并使用 GraphqlApi 方法之一 (addDynamoDbDataSource) 将其附加到 GraphQL API (add_api)。ID 值 (table-for-posts) 是 AWS AppSync 控制台中的数据源的名称。有关源特定的方法列表，请参阅以下页面：
  - DynamoDbDataSource
  - EventBridgeDataSource
  - HttpDataSource
  - LambdaDataSource
  - NoneDataSource
  - OpenSearchDataSource
  - RdsDataSource
- 代码包含函数的请求和响应处理程序，这是简单的扫描和返回。

- 运行时环境指定我们要使用 APPSYNC_JS 运行时环境 1.0.0 版。请注意，这是目前唯一适用于 APPSYNC_JS 的版本。

接下来，我们需要将函数附加到管道解析器。我们使用以下格式创建解析器：

```
Resolver(scope: Construct, id: string, props: ResolverProps)
```

我们的范围是 `this`，CFN ID 是 `pipeline-resolver-get-posts`，属性包含实际函数详细信息。我们在属性中包含：

- 我们以前创建的 GraphQL API (`add_api`)。
- 特殊对象类型名称；这是一个查询操作，因此，我们直接添加了 `Query` 值。
- 字段名称 (`getPost`) 是架构中的 `Query` 类型下面的字段名称。
- 代码包含预备步骤处理程序和后续步骤处理程序。我们的示例仅返回函数执行操作后在上下文中包含的任何结果。
- 运行时环境指定我们要使用 APPSYNC_JS 运行时环境 1.0.0 版。请注意，这是目前唯一适用于 APPSYNC_JS 的版本。
- 管道配置包含对我们创建的函数 (`add_func`) 的引用。

简要说明一下该示例中发生的情况，您看到了一个实施请求和响应处理程序的 AWS AppSync 函数。该函数负责与您的数据源进行交互。请求处理程序向 AWS AppSync 发送 `Scan` 操作，指示它对 DynamoDB 数据源执行什么操作。响应处理程序返回项目列表 (`ctx.result.items`)。然后，将项目列表自动映射到 `Post` GraphQL 类型。

创建基本变更解析器

本节说明了如何创建基本变更解析器。

Console

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   a. 在 API 控制面板中，选择您的 GraphQL API。
   b. 在侧边栏中，选择架构。
2. 在解析器部分的 Mutation 类型下面，选择您的字段旁边的附加。

> **Note**
>
> 在我们的示例中，我们为 `createPost` 附加一个解析器，它将 `Post` 对象添加到我们的表中。假设我们使用上一节中的相同 DynamoDB 表。其分区键设置为 `id` 并且为空。

3. 在附加解析器页面上的解析器类型下面，选择`pipeline resolvers`。提醒一下，您可以在此处找到有关解析器的更多信息。对于解析器运行时，选择 APPSYNC_JS 以启用 JavaScript 运行时环境。
4. 您可以为该 API 启用缓存。我们建议暂时关闭该功能。选择创建。
5. 选择添加函数，然后选择创建新函数。或者，您可能会看到一个创建函数按钮可供选择。

   a. 选择您的 数据源。这应该是使用变更处理数据的源。

   b. 输入一个`Function name`。

   c. 在函数代码下面，您需要实施函数的行为。这是一个变更，因此，理想情况下，请求将对调用的数据源执行某种状态更改操作。结果由响应函数进行处理。

> **Note**
>
> `createPost` 在表中添加或“放置”新的 `Post`，并将我们的参数作为数据。我们可能会添加如下内容：

```
import { util } from '@aws-appsync/utils';

/**
 * Sends a request to `put` an item in the DynamoDB data source
 */
export function request(ctx) {
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({id: util.autoId()}),
    attributeValues: util.dynamodb.toMapValues(ctx.args.input),
  };
}

/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
```

```
  return ctx.result;
}
```

在该步骤中，我们还添加了 request 和 response 函数：

- request：请求处理程序接受上下文以作为参数。请求处理程序返回语句执行 PutItem 命令，这是内置的 DynamoDB 操作（请参阅此处或此处的示例）。PutItem 命令获取分区 key 值（由 util.autoid() 自动生成）以及来自上下文参数输入的 attributes（这些是在请求中传递的值），以将 Post 对象添加到我们的 DynamoDB 表中。key 是 id，attributes 是 date 和 title 字段参数。它们通过 util.dynamodb.toMapValues 帮助程序预先设置格式以与 DynamoDB 表一起使用。
- response：响应接受更新的上下文，并返回请求处理程序的结果。

d. 在完成后，选择创建。

6. 返回到解析器屏幕，在函数下面选择添加函数下拉列表，然后将您的函数添加到函数列表中。
7. 选择保存以更新解析器。

CLI

添加您的函数

- 使用 create-function 命令为管道解析器创建一个函数。

  您需要为该特定命令输入一些参数：

  1. 您的 API 的 api-id。
  2. AWS AppSync 控制台中的函数的 name。
  3. data-source-name 或函数使用的数据源名称。它必须已创建并链接到 AWS AppSync 服务中的 GraphQL API。
  4. runtime 或函数的环境和语言。对于 JavaScript，名称必须是 APPSYNC_JS，运行时环境必须是 1.0.0。
  5. code 或函数的请求和响应处理程序。虽然您可以手动键入该内容，但将其添加到 .txt 文件（或类似格式）并作为参数传入要容易得多。

> **Note**
>
> 我们的查询代码将位于作为参数传入的文件中：
>
> ```
> import { util } from '@aws-appsync/utils';
>
> /**
>  * Sends a request to `put` an item in the DynamoDB data source
>  */
> export function request(ctx) {
>   return {
>     operation: 'PutItem',
>     key: util.dynamodb.toMapValues({id: util.autoId()}),
>     attributeValues: util.dynamodb.toMapValues(ctx.args.input),
>   };
> }
>
> /**
>  * returns the result of the `put` operation
>  */
> export function response(ctx) {
>   return ctx.result;
> }
> ```

示例命令可能如下所示：

```
aws appsync create-function \
--api-id abcdefghijklmnopqrstuvwxyz \
--name add_posts_func_1 \
--data-source-name table-for-posts \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}
```

将在 CLI 中返回输出。示例如下：

```
{
    "functionConfiguration": {
        "functionId": "vulcmbfcxffiram63psb4dduoa",
```

```
        "functionArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/functions/vulcmbfcxffiram63psb4dduoa",
        "name": "add_posts_func_1",
        "dataSourceName": "table-for-posts",
        "maxBatchSize": 0,
        "runtime": {
            "name": "APPSYNC_JS",
            "runtimeVersion": "1.0.0"
        },
        "code": "Code output foes here"
    }
}
```

> **Note**
>
> 确保将 functionId 记录在某处，因为它用于将函数附加到解析器。

创建您的解析器

- 运行 create-resolver 命令，为 Mutation 创建一个管道函数。

  您需要为该特定命令输入一些参数：

  1. 您的 API 的 api-id。
  2. type-name 或架构中的特殊对象类型（查询、变更、订阅）。
  3. field-name 或要将解析器附加到的特殊对象类型中的字段操作。
  4. kind，它指定单位解析器或管道解析器。请将其设置为 PIPELINE 以启用管道函数。
  5. pipeline-config 或附加到解析器的函数。确保您知道函数的 functionId 值。列表顺序很重要。
  6. runtime，它是 APPSYNC_JS (JavaScript)。runtimeVersion 目前是 1.0.0。
  7. code，它包含预备步骤和后续步骤。

  > **Note**
  >
  > 我们的查询代码将位于作为参数传入的文件中：

  ```
  import { util } from '@aws-appsync/utils';
  ```

```
/**
 * Sends a request to `put` an item in the DynamoDB data source
 */
export function request(ctx) {
  const { id, ...values } = ctx.args;
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}

/**
 * returns the result of the `put` operation
 */
export function response(ctx) {
  return ctx.result;
}
```

示例命令可能如下所示：

```
aws appsync create-resolver \
--api-id abcdefghijklmnopqrstuvwxyz \
--type-name Mutation \
--field-name createPost \
--kind PIPELINE \
--pipeline-config functions=vulcmbfcxffiram63psb4dduoa \
--runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
--code file:///path/to/file/{filename}.{fileType}
```

将在 CLI 中返回输出。示例如下：

```
{
    "resolver": {
        "typeName": "Mutation",
        "fieldName": "createPost",
        "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Mutation/resolvers/createPost",
        "kind": "PIPELINE",
        "pipelineConfig": {
            "functions": [
```

```
                "vulcmbfcxffiram63psb4dduoa"
            ]
        },
        "maxBatchSize": 0,
        "runtime": {
            "name": "APPSYNC_JS",
            "runtimeVersion": "1.0.0"
        },
        "code": "Code output goes here"
    }
}
```

CDK

> **Tip**
>
> 在使用 CDK 之前，我们建议您查看 CDK 的官方文档以及 AWS AppSync 的 CDK 参考。下面列出的步骤仅显示用于添加特定资源的一般代码片段示例。这并不意味着，它是您的生产代码中的有效解决方案。我们还假设您已具有正常工作的应用程序。

- 要创建变更，假设您使用同一项目，您可以将其添加到堆栈文件中，就像查询一样。以下是修改的函数和解析器，用于在表中添加新 Post 的变更：

```
const add_func_2 = new appsync.AppsyncFunction(this, 'func-add-post', {
  name: 'add_posts_func_1',
  add_api,
  dataSource: add_api.addDynamoDbDataSource('table-for-posts-2', add_ddb_table),
      code: appsync.Code.fromInline(`
          export function request(ctx) {
            return {
              operation: 'PutItem',
              key: util.dynamodb.toMapValues({id: util.autoId()}),
              attributeValues: util.dynamodb.toMapValues(ctx.args.input),
            };
          }

          export function response(ctx) {
            return ctx.result;
          }
      `),
```

```
  runtime: appsync.FunctionRuntime.JS_1_0_0,
});

new appsync.Resolver(this, 'pipeline-resolver-create-posts', {
  add_api,
  typeName: 'Mutation',
  fieldName: 'createPost',
      code: appsync.Code.fromInline(`
          export function request(ctx) {
            return {};
          }

          export function response(ctx) {
            return ctx.prev.result;
          }
      `),
  runtime: appsync.FunctionRuntime.JS_1_0_0,
  pipelineConfig: [add_func_2],
});
```

> **Note**
>
> 由于该变更和查询的结构相似，因此，我们仅介绍为创建变更而进行的更改。
> 在该函数中，我们将 CFN ID 更改为 `func-add-post` 并将名称更改为 `add_posts_func_1`，以反映我们将 Posts 添加到表中的情况。在数据源中，我们在 AWS AppSync 控制台中为表 (`add_ddb_table`) 创建一个新关联 (`table-for-posts-2`)，因为 `addDynamoDbDataSource` 方法需要这样做。请记住，该新关联仍使用我们以前创建的同一个表，但现在我们在 AWS AppSync 控制台中具有该表的两个连接：一个连接 (`table-for-posts`) 用于查询，另一个连接 (`table-for-posts-2`) 用于变更。代码进行了更改以添加 Post，方法是自动生成其 `id` 值，并接受客户端输入以用于其余字段。
> 在解析器中，我们将 id 值更改为 `pipeline-resolver-create-posts` 以反映我们将 Posts 添加到表中的情况。为了反映架构中的变更，类型名称更改为 `Mutation`，名称更改为 `createPost`。管道配置设置为我们的新变更函数 `add_func_2`。

简要说明一下该示例中发生的情况：AWS AppSync 自动将 `createPost` 字段中定义的参数从 GraphQL 架构转换为 DynamoDB 操作。该示例使用 `id` 键将记录存储在 DynamoDB 中，该键是使用我们的 `util.autoId()` 帮助程序自动创建的。从请求（在 AWS AppSync 控制台中或以其他方

式发出）传递给上下文参数 (ctx.args.input) 的所有其他字段将存储为表的属性。键和属性使用 util.dynamodb.toMapValues(values) 帮助程序自动映射到兼容的 DynamoDB 格式。

AWS AppSync 还支持测试和调试编辑解析器所用的工作流。您可以在调用模板之前使用模拟 context 对象查看转换的模板值。或者，您可以选择在运行查询时以交互方式查看对数据源的完整请求。有关更多信息，请参阅测试和调试解析器 (JavaScript) 和监控和日志记录。

高级解析器

如果您按照设计您的架构中的可选分页一节进行操作，您仍然需要将解析器添加到请求才能使用分页。我们的示例使用名为 getPosts 的查询分页，每次仅返回一部分请求内容。该字段上的解析器代码可能如下所示：

```
/**
 * Performs a scan on the dynamodb data source
 */
export function request(ctx) {
  const { limit = 20, nextToken } = ctx.args;
  return { operation: 'Scan', limit, nextToken };
}

/**
 * @returns the result of the `put` operation
 */
export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

在请求中，我们传入请求的上下文。limit 是 *20*，这意味着我们在第一个查询中最多返回 20 个 Posts。nextToken 光标固定在数据源中的第一个 Post 条目。这些内容将传递给参数。然后，请求执行从第一个 Post 到扫描限制数的扫描。数据源将结果存储在上下文中，该结果将传递给响应。响应返回它检索的 Posts，然后将 nextToken 设置为紧靠限制后面的 Post 条目。发出的下一个请求执行完全相同的操作，但从紧靠第一个查询后面的偏移开始。请记住，这些类型的请求是按顺序完成的，而不是并行完成的。

测试和调试解析器 (JavaScript)

AWS AppSync 对数据源执行 GraphQL 字段上的解析器。在使用管道解析器时，函数与数据源交互。正如 JavaScript 解析器概述中所述，函数使用以 JavaScript 编写并在 APPSYNC_JS 运行时环境中运行

的请求和响应处理程序与数据源进行通信。这样，您就可以在与数据源通信之前和之后提供自定义逻辑和条件。

为了帮助开发人员编写、测试和调试这些解析器，AWS AppSync 控制台还提供了一些工具，以针对单个字段解析器使用模拟数据创建 GraphQL 请求和响应。此外，您可以在 AWS AppSync 控制台中执行查询、变更和订阅，并从 Amazon CloudWatch 中查看整个请求的详细日志流。这包括来自数据源的结果。

使用模拟数据进行测试

在调用 GraphQL 解析器时，它包含一个 `context` 对象，其中包含有关请求的相关信息。其中包括来自客户端的参数、身份信息以及 GraphQL 父字段的数据。它还存储来自数据源的结果，可以在响应处理程序中使用这些结果。有关该结构以及编程时使用的可用帮助程序实用程序的更多信息，请参阅解析器上下文对象参考。

在编写或编辑解析器函数时，您可以将模拟 或测试上下文 对象传递到控制台编辑器中。这样，您就可以了解请求和响应处理程序如何进行评估，而无需实际对数据源运行。例如，您可以传递测试 `firstname: Shaggy` 参数，观察在您的模板代码中使用 `ctx.args.firstname` 时如何评估。您还可以测试任何实用程序帮助程序（例如 `util.autoId()` 或 `util.time.nowISO8601()`）的评估。

测试解析器

该示例将使用 AWS AppSync 控制台测试解析器。

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   a. 在 API 控制面板中，选择您的 GraphQL API。
   b. 在侧边栏中，选择函数。
2. 选择一个现有的函数。
3. 在更新函数页面顶部，选择选择测试上下文，然后选择创建新上下文。
4. 选择一个示例上下文对象，或者在下面的配置测试上下文窗口中手动填充 JSON。
5. 输入测试上下文名称。
6. 选择保存按钮。
7. 要使用此模拟上下文对象评估解析器，请选择 Run Test (运行测试)。

举一个更实际的例子，假设您具有一个存储 GraphQL 类型 Dog 的应用程序，该应用程序自动为对象生成 ID 并将其存储在 Amazon DynamoDB 中。您还希望通过 GraphQL 变更参数写入一些值，并仅允许特定用户查看响应。以下代码片段显示了架构的外观：

```
type Dog {
  breed: String
  color: String
}

type Mutation {
  addDog(firstname: String, age: Int): Dog
}
```

您可以编写一个 AWS AppSync 函数，并将其添加到 addDog 解析器以处理变更。要测试您的 AWS AppSync 函数，您可以填充一个上下文对象，如以下示例中所示。以下代码拥有 name 和 age 客户端的参数，以及 identity 对象中填充的 username：

```
{
    "arguments" : {
        "firstname": "Shaggy",
        "age": 4
    },
    "source" : {},
    "result" : {
        "breed" : "Miniature Schnauzer",
        "color" : "black_grey"
    },
    "identity": {
        "sub" : "uuid",
        "issuer" : " https://cognito-idp.{region}.amazonaws.com/{userPoolId}",
        "username" : "Nadia",
        "claims" : { },
        "sourceIp" :[  "x.x.x.x" ],
        "defaultAuthStrategy" : "ALLOW"
    }
}
```

您可以使用以下代码测试您的 AWS AppSync 函数：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id: util.autoId() }),
    attributeValues: util.dynamodb.toMapValues(ctx.args),
```

```
  };
}

export function response(ctx) {
  if (ctx.identity.username === 'Nadia') {
    console.log("This request is allowed")
    return ctx.result;
  }
  util.unauthorized();
}
```

评估的请求和响应处理程序具有来自测试上下文对象的数据以及从 `util.autoId()` 中生成的值。此外，如果您要将 `username` 更改为除 `Nadia` 之外的值，将不会返回结果，因为授权检查将失败。有关精细访问控制的更多信息，请参阅授权使用案例。

使用 AWS AppSync 的 API 测试请求和响应处理程序

您可以通过 `EvaluateCode` API 命令使用模拟数据远程测试您的代码。要开始使用该命令，请确保您已将 `appsync:evaluateMappingCode` 权限添加到您的策略中。例如：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateCode",
            "Resource": "arn:aws:appsync:<region>:<account>:*"
        }
    ]
}
```

您可以通过 AWS CLI 或 AWS SDK 使用该命令。例如，采用上一节中的 Dog 架构及其 AWS AppSync 函数请求和响应处理程序。通过在本地工作站上使用 CLI，将代码保存到名为 `code.js` 的文件中，然后将 `context` 对象保存到名为 `context.json` 的文件中。从 Shell 中，运行以下命令：

```
$ aws appsync evaluate-code \
  --code file://code.js \
  --function response \
  --context file://context.json \
  --runtime name=APPSYNC_JS,runtimeVersion=1.0.0
```

响应包含一个 evaluationResult，其中包含处理程序返回的负载。它还包含一个 logs 对象，其中保存处理程序在评估期间生成的日志列表。这样，就可以轻松调试代码执行，并查看有关评估的信息以帮助排除故障。例如：

```
{
    "evaluationResult": "{\"breed\":\"Miniature Schnauzer\",\"color\":\"black_grey\"}",
    "logs": [
        "INFO - code.js:13:5: \"This request is allowed\""
    ]
}
```

可以将 evaluationResult 解析为 JSON，其中提供：

```
{
  "breed": "Miniature Schnauzer",
  "color": "black_grey"
}
```

通过使用 SDK，您可以轻松合并常用的测试套件中的测试以验证处理程序行为。我们建议使用 Jest 测试框架创建测试，但任何测试套件都有效。以下代码片段显示假设的验证运行。请注意，我们希望评估响应是有效的 JSON ，因此，我们使用 JSON.parse 从字符串响应中检索 JSON：

```
const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })
const runtime = {name:'APPSYNC_JS',runtimeVersion:'1.0.0')

test('request correctly calls DynamoDB', async () => {
  const code = fs.readFileSync('./code.js', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateCode({ code, context, runtime, function:
 'request' }).promise()
  const result = JSON.parse(response.evaluationResult)

  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})
```

这会产生以下结果：

```
Ran all test suites.
> jest

PASS ./index.test.js
# request correctly calls DynamoDB (543 ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 totalTime: 1.511 s, estimated 2 s
```

调试实时查询

在调试生产应用程序时，进行端到端测试和日志记录是无可替代的。AWSAppSync 允许您使用 Amazon CloudWatch 记录错误和完整请求详细信息。此外，您可以使用 AWS AppSync 控制台测试 GraphQL 查询、变更和订阅，并将每个请求的日志数据实时流式传输到查询编辑器以进行实时调试。对于订阅而言，日志显示连接时间信息。

要执行该操作，您需要提前启用 Amazon CloudWatch Logs，如监控和日志记录中所述。接下来，在 AWS AppSync 控制台中，选择查询选项卡，然后输入有效的 GraphQL 查询。在右下部分中，单击并拖动日志窗口以打开“日志”视图。在页面顶部，选择“播放”箭头图标运行您的 GraphQL 查询。稍后，操作的完整请求和响应日志将流式传输到该部分，您可以在控制台中查看这些日志。

管道解析器 (JavaScript)

AWS AppSync 对 GraphQL 字段执行解析器。在某些情况下，应用程序需要执行多个操作以解析单个 GraphQL 字段。通过使用管道解析器，开发人员现在可以编写称为“函数”的操作并按顺序执行它们。例如，管道解析器对于需要在获取字段数据之前执行授权检查的应用程序非常有用。

有关 JavaScript 管道解析器架构的更多信息，请参阅 JavaScript 解析器概述。

创建管道解析器

在 AWS AppSync 控制台中，转到架构页面。

保存以下架构：

```
schema {
    query: Query
    mutation: Mutation
}

type Mutation {
```

```
    signUp(input: Signup): User
}

type Query {
    getUser(id: ID!): User
}

input Signup {
    username: String!
    email: String!
}

type User {
    id: ID!
    username: String
    email: AWSEmail
}
```

我们将一个管道解析器连接到 Mutation 类型的 signUp 字段。在右侧的 Mutation 类型中，选择 `signUp` 变更字段旁边的附加。将解析器设置为`pipeline resolver`和 `APPSYNC_JS` 运行时环境，然后创建解析器。

我们的管道解析器通过先验证电子邮件地址输入，然后将用户保存在系统中来注册用户。我们将电子邮件验证封装在 validateEmail 函数中，并将用户保存在 saveUser 函数中。首先执行 validateEmail 函数，如果电子邮件有效，则执行 saveUser 函数。

执行流程如下所示：

1. Mutation.signUp 解析器请求处理程序
2. validateEmail 函数
3. saveUser 函数
4. Mutation.signUp 解析器响应处理程序

由于我们可能在 API 上的其他解析器中重复使用 validateEmail 函数，因此，我们希望避免访问 `ctx.args`，因为它们将从一个 GraphQL 字段更改为另一个字段。相反，我们可以使用 `ctx.stash` 存储 `signUp(input: Signup)` 输入字段参数中的电子邮件属性。

替换请求函数和响应函数以更新解析器代码：

```
export function request(ctx) {
```

```
    ctx.stash.email = ctx.args.input.email
    return {};
}

export function response(ctx) {
    return ctx.prev.result;
}
```

选择创建或保存以更新解析器。

创建函数

从管道解析器页面的函数部分中，单击添加函数，然后单击创建新函数。也可以不通过解析器页面创建函数；为此，请在 AWS AppSync 控制台中转到函数页面。选择创建函数按钮。让我们创建一个函数来检查电子邮件是否有效并来自特定域。如果电子邮件无效，该函数会引发一个错误。否则，它将转发接收到的任何输入。

确保您已创建一个 NONE 类型的数据源。在数据源名称列表中选择该数据源。对于函数名称，请输入 `validateEmail`。在函数代码区域中，使用以下代码片段覆盖所有内容：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { email } = ctx.stash;
  const valid = util.matches(
    '^[a-zA-Z0-9_.+-]+@(?:(?:[a-zA-Z0-9-]+\.)?[a-zA-Z]+\.)?(myvaliddomain)\.com',
    email
  );
  if (!valid) {
    util.error(`"${email}" is not a valid email.`);
  }

  return { payload: { email } };
}

export function response(ctx) {
  return ctx.result;
}
```

检查您的输入，然后选择创建。我们刚刚创建了 validateEmail 函数。重复这些步骤以创建具有以下代码的 saveUser 函数（为了简单起见，我们使用 NONE 数据源，并假设在函数执行后已将用户保存在系统中）：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  return ctx.prev.result;
}

export function response(ctx) {
  ctx.result.id = util.autoId();
  return ctx.result;
}
```

我们刚刚创建了 saveUser 函数。

将函数添加到管道解析器

我们的函数应该已自动添加到刚刚创建的管道解析器中。如果情况并非如此，或者您通过函数页面创建了函数，您可以再次单击 signUp 解析器页面上的添加函数以附加这些函数。将 validateEmail 和 saveUser 函数添加到解析器中。validateEmail 函数应放在 saveUser 函数之前。在添加更多函数时，您可以使用上移和下移选项重新排列函数的执行顺序。检查您的更改，然后选择保存。

运行查询

在 AWS AppSync 控制台中，转到查询页面。在资源管理器中，确保您正在使用变更。如果不是，请在下拉列表中选择Mutation，然后选择 +。输入以下查询：

```
mutation {
  signUp(input: {email: "nadia@myvaliddomain.com", username: "nadia"}) {
    id
    username
  }
}
```

它应返回如下内容：

```
{
  "data": {
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "username": "nadia"
    }
  }
}
```

我们已成功注册用户，并使用管道解析器验证了输入电子邮件。

配置解析器 (VTL)

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

GraphQL 解析器将类型的架构中的字段连接到数据源。解析器是用于完成请求的机制。AWSAppSync 可以自动从架构中创建并连接解析器，或者从现有表中创建架构并连接解析器，而无需编写任何代码。

AWS AppSync 中的解析器使用 JavaScript，将 GraphQL 表达式转换为数据源可使用的格式。或者，可以使用 Apache Velocity 模板语言 (VTL) 编写映射模板，以将 GraphQL 表达式转换为数据源可使用的格式。

本节说明了如何使用 VTL 配置解析器。可以在解析器映射模板编程指南中找到用于编写解析器的入门教程风格的编程指南，以及在解析器映射模板上下文参考中找到可以在编程时使用的帮助程序实用程序。AWSAppSync 还具有内置的测试和调试流程，您可以在编辑或从头开始编写时使用这些流程。有关更多信息，请参阅测试和调试解析器。

我们建议您在尝试使用任何上述教程之前先遵循本指南。

在本节中，我们将介绍如何创建解析器，为变更添加解析器以及使用高级配置。

创建您的第一个解析器

按照前面几节中的示例，第一步是为 `Query` 类型创建一个解析器。

Console

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   a. 在 API 控制面板中，选择您的 GraphQL API。
   b. 在侧边栏中，选择架构。
2. 在页面右侧，具有一个名为解析器的窗口。该框包含页面左侧的架构窗口中定义的类型和字段列表。您可以将解析器附加到字段。例如，在 Query 类型下面，选择 `getTodos` 字段旁边的附加。
3. 在创建解析器页面上，选择您在附加数据源指南中创建的数据源。在配置映射模板窗口中，您可以使用右侧的下拉列表选择通用请求和响应映射模板，也可以编写自己的映射模板。

> **Note**
>
> 请求映射模板与响应映射模板的配对称为单位解析器。单位解析器通常用于执行机械性的操作；我们建议仅将它们用于具有少量数据源的单一操作。对于更复杂的操作，我们建议使用管道解析器，该解析器按顺序对多个数据源执行多个操作。
> 有关请求映射模板和响应映射模板之间的差异的更多信息，请参阅单位解析器。
> 有关使用管道解析器的更多信息，请参阅管道解析器。

4. 对于常见使用案例，AWS AppSync 控制台具有内置的模板，可用于从数据源获取项目（例如，所有项目查询、单独查找等）。例如，对于设计您的架构中的简单架构版本（getTodos 没有分页），用于列出项目的请求映射模板如下所示：

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
}
```

5. 您始终需要为请求提供响应映射模板。控制台为列表提供的默认模板具有以下传递值：

```
$util.toJson($ctx.result.items)
```

此示例中，项目列表的 context 对象（别名为 $ctx）的格式为 $context.result.items。如果您的 GraphQL 操作返回单个项目，则它为 $context.result。AWSAppSync 为常见操作提供帮助程序函数（例如前面列出的 $util.toJson），以正确设置响应格式。有关完整的函数列表，请参阅解析器映射模板实用程序参考。

6. 选择保存解析器。

API

1. 调用 CreateResolver API 以创建一个解析器对象。
2. 您可以调用 UpdateResolver API 以修改解析器的字段。

CLI

1. 运行 create-resolver 命令以创建解析器。

您需要为该特定命令键入 6 个参数：

1. 您的 API 的 `api-id`。
2. 您要在架构中修改的类型的 `type-name`。在控制台示例中，这是 `Query`。
3. 您要在类型中修改的字段的 `field-name`。在控制台示例中，这是 `getTodos`。
4. 您在附加数据源指南中创建的数据源的 `data-source-name`。
5. `request-mapping-template`，这是请求的正文。在控制台示例中，这是：

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
}
```

6. `response-mapping-template`，这是响应的正文。在控制台示例中，这是：

```
$util.toJson($ctx.result.items)
```

示例命令可能如下所示：

```
aws appsync create-resolver --api-id abcdefghijklmnopqrstuvwxyz --type-name
 Query --field-name getTodos --data-source-name TodoTable --request-mapping-
template "{ "version" : "2017-02-28", "operation" : "Scan", }" --response-
mapping-template ""$"util.toJson("$"ctx.result.items)"
```

将在 CLI 中返回输出。示例如下：

```
{
    "resolver": {
        "kind": "UNIT",
        "dataSourceName": "TodoTable",
        "requestMappingTemplate": "{ version : 2017-02-28, operation : Scan, }",
        "resolverArn": "arn:aws:appsync:us-west-2:107289374856:apis/
abcdefghijklmnopqrstuvwxyz/types/Query/resolvers/getTodos",
        "typeName": "Query",
        "fieldName": "getTodos",
        "responseMappingTemplate": "$util.toJson($ctx.result.items)"
    }
}
```

2. 要修改解析器的字段和/或映射模板，请运行 `update-resolver` 命令。

   除了 `api-id` 参数以外，`create-resolver` 命令中使用的参数将由 `update-resolver` 命令中的新值覆盖。

为变更添加解析器

下一步是为您的 `Mutation` 类型创建一个解析器。

Console

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   a. 在 API 控制面板中，选择您的 GraphQL API。
   b. 在侧边栏中，选择架构。
2. 在 Mutation 类型下面，选择 addTodo 字段旁边的附加。
3. 在创建解析器页面上，选择您在附加数据源指南中创建的数据源。
4. 在配置映射模板窗口中，您需要修改请求模板，因为这是一个变更，该操作将新项目添加 DynamoDB 中。使用以下请求映射模板：

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

5. AWS AppSync 自动将 addTodo 字段中定义的参数从 GraphQL 架构转换为 DynamoDB 操作。上一示例使用 `id` 键将记录存储在 DynamoDB 中，该键是从变更参数中作为 `$ctx.args.id` 传递的。您传递的所有其他字段使用 `$util.dynamodb.toMapValuesJson($ctx.args)` 自动映射到 DynamoDB 属性。

   对于此解析器，使用以下响应映射模板：

```
$util.toJson($ctx.result)
```

AWS AppSync 还支持用于编辑解析器的测试和调试工作流程。您可在调用之前使用模拟 `context` 对象查看模板经过转换的值。还可在运行查询时选择以交互方式查看对数据源的完整请求执行过程。有关更多信息，请参阅测试和调试解析器和监控和日志记录。

6. 选择保存解析器。

API

您也可以使用 API 通过创建您的第一个解析器一节中的命令以及本节中的参数详细信息执行该操作。

CLI

您也可以在 CLI 中使用创建您的第一个解析器一节中的命令以及本节中的参数详细信息执行该操作。

此时，如果您不使用高级解析器，您可以开始使用 GraphQL API，如使用 API 中所述。

高级解析器

如果您按照设计您的架构的“高级”一节进行操作，并构建一个示例架构以执行分页扫描，请将以下请求模板用于 `getTodos` 字段：

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "limit": $util.defaultIfNull(${ctx.args.limit}, 20),
    "nextToken": $util.toJson($util.defaultIfNullOrBlank($ctx.args.nextToken, null))
}
```

对于这个分页使用案例，响应映射不只是简单的传递，因为它必须包含光标（这样客户端才知道下次从哪一页开始）以及结果集。映射模板如下所示：

```
{
    "todos": $util.toJson($context.result.items),
    "nextToken": $util.toJson($context.result.nextToken)
}
```

以上响应映射模板中的字段必须与 `TodoConnection` 类型中定义的字段匹配。

如果存在以下关系：您具有 Comments 表并且要解析 Todo 类型（返回 [Comment] 类型）的 Comments 字段，您可以使用映射模板以对第二个表运行查询。为此，您必须已为 Comments 表创建了一个数据源，如附加数据源中所述。

> **Note**
>
> 我们对第二个表使用查询操作仅用于说明目的。您可以对 DynamoDB 使用其他操作。此外，您可以从其他数据源（例如 AWS Lambda 或 Amazon OpenSearch Service）中提取数据，因为该关系是由您的 GraphQL 架构控制的。

Console

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   a. 在 API 控制面板中，选择您的 GraphQL API。
   b. 在侧边栏中，选择架构。
2. 在 Todo 类型下面，选择 comments 字段旁边的附加。
3. 在创建解析器页面上，选择您的 Comments 表数据源。快速入门指南中的 Comments 表的默认名称是 AppSyncCommentTable，但它可能会根据您指定的名称而有所不同。
4. 将以下代码片段添加到您的请求映射模板中：

```
{
    "version": "2017-02-28",
    "operation": "Query",
    "index": "todoid-index",
    "query": {
        "expression": "todoid = :todoid",
        "expressionValues": {
            ":todoid": {
                "S": $util.toJson($context.source.id)
            }
        }
    }
}
```

5. context.source 会引用当前被解析的字段的父对象。在本示例中，source.id 指单个 Todo 对象，之后它会被用于查询表达式。

   您可以如下所示使用传递响应映射模板：

```
$util.toJson($ctx.result.items)
```

6. 选择保存解析器。
7. 最后，返回到控制台中的架构页面，将一个解析器附加到 addComment 字段，并指定 Comments 表的数据源。在此例中请求映射模板只是简单的 PutItem，它具有作为参数注释的特定 todoid，但您可以使用 $utils.autoId() 实用程序来为如下所示注释创建唯一的排序键：

```
{
    "version": "2017-02-28",
    "operation": "PutItem",
    "key": {
        "todoid": { "S": $util.toJson($context.arguments.todoid) },
        "commentid": { "S": "$util.autoId()" }
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

如下所示使用传递响应模板：

```
$util.toJson($ctx.result)
```

API

您也可以使用 API 通过创建您的第一个解析器一节中的命令以及本节中的参数详细信息执行该操作。

CLI

您也可以在 CLI 中使用创建您的第一个解析器一节中的命令以及本节中的参数详细信息执行该操作。

直接 Lambda 解析器 (VTL)

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

通过使用直接 Lambda 解析器，您可以在使用 AWS Lambda 数据源时避免使用 VTL 映射模板。AWSAppSync 可以为您的 Lambda 函数提供默认负载，以及从 Lambda 函数响应到 GraphQL 类型的默认转换。您可以选择提供请求模板、响应模板或两者都不提供，AWS AppSync 将相应地进行处理。

要了解 AWS AppSync 提供的默认请求负载和响应转换的更多信息，请参阅直接 Lambda 解析器参考。有关设置 AWS Lambda 数据源和设置 IAM 信任策略的更多信息，请参阅附加数据源。

配置直接 Lambda 解析器

以下几节说明了如何附加 Lambda 数据源，并将 Lambda 解析器添加到您的字段中。

添加 Lambda 数据源

您必须先添加 Lambda 数据源，然后才能激活直接 Lambda 解析器。

Console

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   a. 在 API 控制面板中，选择您的 GraphQL API。
   b. 在侧边栏中，选择数据源。
2. 选择创建数据源。
   a. 对于数据源名称，输入您的数据源的名称，例如 **myFunction**。
   b. 对于数据源类型，选择 AWS Lambda 函数。
   c. 对于区域，选择相应的区域。
   d. 对于函数 ARN，从下拉列表中选择 Lambda 函数。您可以搜索函数名称，或手动输入要使用的函数的 ARN。
   e. 创建新的 IAM 角色（建议），或者选择具有 `lambda:invokeFunction` IAM 权限的现有角色。现有角色需要具有一个信任策略，如附加数据源一节中所述。

      以下是一个示例 IAM 策略，该策略具有对资源执行操作所需的权限：

```
{
     "Version": "2012-10-17",
     "Statement": [
         {
             "Effect": "Allow",
             "Action": [ "lambda:invokeFunction" ],
```

```
            "Resource": [
                "arn:aws:lambda:us-
west-2:123456789012:function:myFunction",
                "arn:aws:lambda:us-
west-2:123456789012:function:myFunction:*"
            ]
        }
    ]
 }
```

3. 选择创建按钮。

CLI

1. 运行 `create-data-source` 命令以创建数据源对象。

   您需要为该特定命令键入 4 个参数：

   1. 您的 API 的 `api-id`。
   2. 您的数据源的 `name`。在控制台示例中，这是数据源名称。
   3. 数据源的 `type`。在控制台示例中，这是 AWS Lambda 函数。
   4. `lambda-config`，即控制台示例中的函数 ARN。

   > **Note**
   >
   > 必须配置其他参数（例如 `Region`），但这些参数通常默认为您的 CLI 配置值。

   示例命令可能如下所示：

   ```
   aws appsync create-data-source --api-id abcdefghijklmnopqrstuvwxyz
    --name myFunction --type AWS_LAMBDA --lambda-config
    lambdaFunctionArn=arn:aws:lambda:us-west-2:102847592837:function:appsync-
   lambda-example
   ```

   将在 CLI 中返回输出。示例如下：

   ```
   {
       "dataSource": {
   ```

```
        "dataSourceArn": "arn:aws:appsync:us-west-2:102847592837:apis/
abcdefghijklmnopqrstuvwxyz/datasources/myFunction",
        "type": "AWS_LAMBDA",
        "name": "myFunction",
        "lambdaConfig": {
            "lambdaFunctionArn": "arn:aws:lambda:us-
west-2:102847592837:function:appsync-lambda-example"
        }
    }
}
```

2. 要修改数据源的属性，请运行 update-data-source 命令。

   除了 api-id 参数以外，create-data-source 命令中使用的参数将由 update-data-source 命令中的新值覆盖。

激活直接 Lambda 解析器

在创建 Lambda 数据源并设置相应 IAM 角色以允许 AWS AppSync 调用该函数后，您可以将其链接到解析器或管道函数。

Console

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   a. 在 API 控制面板中，选择您的 GraphQL API。
   b. 在侧边栏中，选择架构。
2. 在解析器窗口中，选择一个字段或操作，然后选择附加按钮。
3. 在创建新解析器页面中，从下拉列表中选择 Lambda 函数。
4. 要使用直接 Lambda 解析器，请确认在配置映射模板部分中禁用了请求和响应映射模板。
5. 选择保存解析器按钮。

CLI

- 运行 create-resolver 命令以创建解析器。

  您需要为该特定命令键入 6 个参数：

  1. 您的 API 的 api-id。

2. 架构中的类型的 `type-name`。
3. 架构中的字段的 `field-name`。
4. `data-source-name` 或您的 Lambda 函数名称。
5. `request-mapping-template`，这是请求的正文。在控制台示例中，已将其禁用：

```
" "
```

6. `response-mapping-template`，这是响应的正文。在控制台示例中，也已将其禁用：

```
" "
```

示例命令可能如下所示：

```
aws appsync create-resolver --api-id abcdefghijklmnopqrstuvwxyz --type-name
 Subscription --field-name onCreateTodo --data-source-name LambdaTest --request-
mapping-template " " --response-mapping-template " "
```

将在 CLI 中返回输出。示例如下：

```
{
    "resolver": {
        "resolverArn": "arn:aws:appsync:us-west-2:102847592837:apis/
abcdefghijklmnopqrstuvwxyz/types/Subscription/resolvers/onCreateTodo",
        "typeName": "Subscription",
        "kind": "UNIT",
        "fieldName": "onCreateTodo",
        "dataSourceName": "LambdaTest"
    }
}
```

在您禁用映射模板时，将在 AWS AppSync 中出现一些其他行为：

- 通过禁用映射模板，您向 AWS AppSync 发出通知，表示您接受直接 Lambda 解析器参考中指定的默认数据转换。
- 通过禁用请求映射模板，您的 Lambda 数据源将接收包含整个上下文对象的负载。
- 通过禁用响应映射模板，将转换您的 Lambda 调用结果，具体取决于请求映射模板版本或是否还禁用了请求映射模板。

测试和调试解析器 (VTL)

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

AWS AppSync 对数据源执行 GraphQL 字段上的解析器。正如解析器映射模板概述中所述，解析器使用模板语言与数据源进行通信。这样，您就可以在与数据源通信之前和之后自定义行为并应用逻辑和条件。有关编写解析器的入门教程风格的编程指南，请参阅解析器映射模板编程指南。

为了帮助开发人员编写、测试和调试这些解析器，AWS AppSync 控制台还提供了一些工具，以针对单个字段解析器使用模拟数据创建 GraphQL 请求和响应。此外，您可以在 AWS AppSync 控制台中执行查询、变更和订阅，并从 Amazon CloudWatch 中查看整个请求的详细日志流。其中包括从数据源获得的结果。

使用模拟数据进行测试

在调用 GraphQL 解析器时，它包含一个 `context` 对象，其中包含有关请求的信息。其中包括来自客户端的参数、身份信息以及 GraphQL 父字段的数据。它还包含来自数据源的结果，可以在响应模板中使用这些结果。有关此结构的更多信息以及在编程时可用的帮助程序实用程序，请参阅解析器映射模板上下文参考。

在编写或编辑解析器时，您可以将模拟 或测试上下文 对象传递到控制台编辑器中。这使您能够查看请求和响应模板如何评估，而不必针对数据源实际运行。例如，您可以传递测试 `firstname: Shaggy` 参数，观察在您的模板代码中使用 `$ctx.args.firstname` 时如何评估。您还可以测试任何实用程序帮助程序（例如 `$util.autoId()` 或 `util.time.nowISO8601()`）的评估。

测试解析器

该示例将使用 AWS AppSync 控制台测试解析器。

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   a. 在 API 控制面板中，选择您的 GraphQL API。
   b. 在侧边栏中，选择架构。
2. 如果您尚未在类型下面的字段旁边选择附加以添加解析器，请执行该操作。

   有关如何构建完整解析器的更多信息，请参阅配置解析器。

否则，选择已位于字段中的解析器。

3. 在编辑解析器页面顶部，选择选择测试上下文，然后选择创建新上下文。
4. 选择一个示例上下文对象，或者在下面的执行上下文窗口中手动填充 JSON。
5. 输入测试上下文名称。
6. 选择保存按钮。
7. 在编辑解析器页面顶部，选择运行测试。

举一个更实际的例子，假设您具有一个存储 GraphQL 类型 Dog 的应用程序，该应用程序自动为对象生成 ID 并将其存储在 Amazon DynamoDB 中。您还希望通过 GraphQL 变更参数写入一些值，并仅允许特定用户查看响应。下面显示了架构的外观：

```
type Dog {
  breed: String
  color: String
}

type Mutation {
  addDog(firstname: String, age: Int): Dog
}
```

在您为 addDog 变更添加解析器时，您可以填充上下文对象，如以下示例中所示。以下代码拥有 name 和 age 客户端的参数，以及 identity 对象中填充的 username：

```
{
    "arguments" : {
        "firstname": "Shaggy",
        "age": 4
    },
    "source" : {},
    "result" : {
        "breed" : "Miniature Schnauzer",
        "color" : "black_grey"
    },
    "identity": {
        "sub" : "uuid",
        "issuer" : " https://cognito-idp.{region}.amazonaws.com/{userPoolId}",
        "username" : "Nadia",
        "claims" : { },
```

```
        "sourceIp" :[  "x.x.x.x" ],
        "defaultAuthStrategy" : "ALLOW"
    }
}
```

您可以使用以下请求和响应映射模板测试此内容：

请求模板

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : { "S" : "$util.autoId()" }
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

响应模板

```
#if ($context.identity.username == "Nadia")
  $util.toJson($ctx.result)
#else
  $util.unauthorized()
#end
```

经过评估的模板拥有您的测试上下文对象的数据，以及 `$util.autoId()` 生成的值。此外，如果您要将 `username` 更改为除 `Nadia` 之外的值，将不会返回结果，因为授权检查将失败。有关精细访问控制的更多信息，请参阅[授权使用案例](#)。

使用 AWS AppSync 的 API 测试映射模板

您可以通过 `EvaluateMappingTemplate` API 命令使用模拟数据远程测试您的映射模板。要开始使用该命令，请确保您已将 `appsync:evaluateMappingTemplate` 权限添加到您的策略中。例如：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateMappingTemplate",
```

```
            "Resource": "arn:aws:appsync:<region>:<account>:*"
        }
    ]
}
```

您可以通过 AWS CLI 或 AWS SDK 使用该命令。例如，采用上一节中的 Dog 架构及其请求/响应映射模板。通过在本地工作站上使用 CLI，将请求模板保存到名为 `request.vtl` 的文件中，然后将 `context` 对象保存到名为 `context.json` 的文件中。从 Shell 中，运行以下命令：

```
aws appsync evaluate-mapping-template --template file://request.vtl --context file://
context.json
```

该命令返回以下响应：

```
{
  "evaluationResult": "{\n    \"version\" : \"2017-02-28\",\n
 \"operation\" : \"PutItem\",\n    \"key\" : {\n        \"id\" : { \"S\" :
 \"afcb4c85-49f8-40de-8f2b-248949176456\" }\n    },\n    \"attributeValues\" :
 {\"firstname\":{\"S\":\"Shaggy\"},\"age\":{\"N\":4}}\n}\n"
}
```

`evaluationResult` 包含使用提供的 `context` 测试您提供的模板的结果。您也可以使用 AWS SDK 测试您的模板。以下是使用 AWS SDK for JavaScript V2 的示例：

```
const AWS = require('aws-sdk')
const client = new AWS.AppSync({ region: 'us-east-2' })

const template = fs.readFileSync('./request.vtl', 'utf8')
const context = fs.readFileSync('./context.json', 'utf8')

client
  .evaluateMappingTemplate({ template, context })
  .promise()
  .then((data) => console.log(data))
```

通过使用 SDK，您可以轻松合并常用的测试套件中的测试以验证模板行为。我们建议使用 Jest 测试框架创建测试，但任何测试套件都有效。以下代码片段显示假设的验证运行。请注意，我们希望评估响应是有效的 JSON ，因此，我们使用 `JSON.parse` 从字符串响应中检索 JSON：

```
const AWS = require('aws-sdk')
```

```
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })

test('request correctly calls DynamoDB', async () => {
  const template = fs.readFileSync('./request.vtl', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateMappingTemplate({ template,
 context }).promise()
  const result = JSON.parse(response.evaluationResult)

  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})
```

这会产生以下结果：

```
Ran all test suites.
> jest

PASS ./index.test.js
# request correctly calls DynamoDB (543 ms)

Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 total
Time: 1.511 s, estimated 2 s
```

调试实时查询

在调试生产应用程序时，进行端到端测试和日志记录是无可替代的。AWSAppSync 允许您使用 Amazon CloudWatch 记录错误和完整请求详细信息。此外，您可以使用 AWS AppSync 控制台测试 GraphQL 查询、变更和订阅，并将每个请求的日志数据实时流式传输到查询编辑器以进行实时调试。对于订阅而言，日志显示连接时间信息。

要执行该操作，您需要提前启用 Amazon CloudWatch Logs，如监控和日志记录中所述。接下来，在 AWS AppSync 控制台中，选择查询选项卡，然后输入有效的 GraphQL 查询。在右下部分中，单击并拖动日志窗口以打开“日志”视图。在页面顶部，选择“播放”箭头图标运行您的 GraphQL 查询。片刻之后，此操作的完整请求和响应日志将流式传输到控制台的这一部分，之后您可以在控制台中进行查看。

管道解析器 (VTL)

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

AWS AppSync 对 GraphQL 字段执行解析器。在某些情况下，应用程序需要执行多个操作以解析单个 GraphQL 字段。通过使用管道解析器，开发人员现在可以编写称为“函数”的操作并按顺序执行它们。例如，管道解析器对于需要在获取字段数据之前执行授权检查的应用程序非常有用。

管道解析器由之前 映射模板、之后 映射模板和一组函数组成。每个函数具有对数据源执行的请求映射模板和响应映射模板。由于管道解析器将执行委托给函数列表，因此它不会链接到任何数据源。单位解析器和函数是对数据源执行操作的基元。有关更多信息，请参阅解析器映射模板概述。

创建管道解析器

在 AWS AppSync 控制台中，转到架构页面。

保存以下架构：

```
schema {
    query: Query
    mutation: Mutation
}

type Mutation {
    signUp(input: Signup): User
}

type Query {
    getUser(id: ID!): User
}

input Signup {
    username: String!
    email: String!
}

type User {
    id: ID!
```

```
    username: String
    email: AWSEmail
}
```

我们将一个管道解析器连接到 Mutation 类型的 signUp 字段。在右侧的 Mutation 类型中，选择 signUp 变更字段旁边的附加。在“创建解析器”页面上，单击操作，然后单击更新运行时。依次选择Pipeline Resolver、VTL 和更新。该页面现在应显示三个部分：之前映射模板文本区域、函数部分和之后映射模板文本区域。

我们的管道解析器通过先验证电子邮件地址输入，然后将用户保存在系统中来注册用户。我们将电子邮件验证封装在 validateEmail 函数中，并将用户保存在 saveUser 函数中。首先执行 validateEmail 函数，如果电子邮件有效，则执行 saveUser 函数。

执行流将如下所示：

1. Mutation.signUp 解析器请求映射模板
2. validateEmail 函数
3. saveUser 函数
4. Mutation.signUp 解析器响应映射模板

由于我们可能在 API 上的其他解析器中重复使用 validateEmail 函数，因此，我们希望避免访问 $ctx.args，因为它们将从一个 GraphQL 字段更改为另一个字段。相反，我们可以使用 $ctx.stash 存储 signUp(input: Signup) 输入字段参数中的电子邮件属性。

之前映射模板：

```
## store email input field into a generic email key
$util.qr($ctx.stash.put("email", $ctx.args.input.email))
{}
```

控制台提供一个我们将使用的默认传递之后映射模板：

```
$util.toJson($ctx.result)
```

选择创建或保存以更新解析器。

创建函数

从管道解析器页面的函数部分中，单击添加函数，然后单击创建新函数。也可以不通过解析器页面创建函数；为此，请在 AWS AppSync 控制台中转到函数页面。选择创建函数按钮。让我们创建一个函数

来检查电子邮件是否有效并来自特定域。如果电子邮件无效，该函数会引发一个错误。否则，它将转发接收到的任何输入。

在新函数页面上，选择操作，然后选择更新运行时。选择 VTL，然后选择更新。确保您已创建一个 NONE 类型的数据源。在数据源名称列表中选择该数据源。对于函数名称，请输入 `validateEmail`。在函数代码区域中，使用以下代码片段覆盖所有内容：

```
#set($valid = $util.matches("^[a-zA-Z0-9_.+-]+@(?:(?:[a-zA-Z0-9-]+\.)?[a-zA-Z]+\.)?
(myvaliddomain)\.com", $ctx.stash.email))
#if (!$valid)
    $util.error("$ctx.stash.email is not a valid email.")
#end
{
    "payload": { "email": $util.toJson(${ctx.stash.email}) }
}
```

将其粘贴到响应映射模板中：

```
$util.toJson($ctx.result)
```

检查您的更改，然后选择创建。我们刚刚创建了 validateEmail 函数。重复这些步骤以创建具有以下请求和响应映射模板的 saveUser 函数（为了简单起见，我们使用 NONE 数据源，并假设在函数执行后已将用户保存在系统中）：

请求映射模板：

```
## $ctx.prev.result contains the signup input values. We could have also
## used $ctx.args.input.
{
    "payload": $util.toJson($ctx.prev.result)
}
```

响应映射模板：

```
## an id is required so let's add a unique random identifier to the output
$util.qr($ctx.result.put("id", $util.autoId()))
$util.toJson($ctx.result)
```

我们刚刚创建了 saveUser 函数。

将函数添加到管道解析器

我们的函数应该已自动添加到刚刚创建的管道解析器中。如果情况并非如此，或者您通过函数页面创建了函数，您可以单击解析器页面上的添加函数以附加这些函数。将 validateEmail 和 saveUser 函数添加到解析器中。validateEmail 函数应放在 saveUser 函数之前。在添加更多函数时，您可以使用上移和下移选项重新排列函数的执行顺序。检查您的更改，然后选择保存。

执行查询

在 AWS AppSync 控制台中，转到查询页面。在资源管理器中，确保您正在使用变更。如果不是，请在下拉列表中选择Mutation，然后选择 +。输入以下查询：

```
mutation {
  signUp(input: {
    email: "nadia@myvaliddomain.com"
    username: "nadia"
  }) {
    id
    email
  }
}
```

它应返回如下内容：

```
{
  "data": {
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "email": "nadia@myvaliddomain.com"
    }
  }
}
```

我们已成功注册用户，并使用管道解析器验证了输入电子邮件。要遵循有关管道解析器的更完整的教程，您可以转到教程：管道解析器

步骤 4：使用 API：CDK 示例

> **Tip**
>
> 在使用 CDK 之前，我们建议您查看 CDK 的官方文档以及 AWS AppSync 的 CDK 参考。

我们还建议确保 AWS CLI 和 NPM 安装在您的系统上正常工作。

在本节中，我们将创建一个简单的 CDK 应用程序，该应用程序可以在 DynamoDB 表中添加和获取项目。这是一个快速入门示例，它使用设计您的架构、附加数据源和配置解析器 (JavaScript) 小节中的一些代码。

设置 CDK 项目

Warning

根据您的环境，这些步骤可能不完全准确。我们假设您的系统安装了所需的实用程序，提供了一种方法与 AWS 服务交互并设置了正确的配置。

第一步是安装 AWS CDK。在 CLI 中，您可以输入以下命令：

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

接下来，您需要创建一个项目目录，然后导航到该目录。用于创建和导航到目录的一组示例命令是：

```
mkdir example-cdk-app
cd example-cdk-app
```

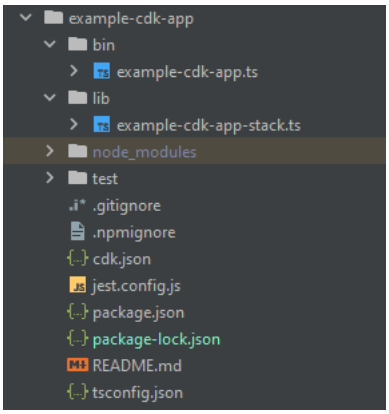
接下来，您需要创建一个应用程序。我们的服务主要使用 TypeScript。在您的项目目录中，输入以下命令：

```
cdk init app --language typescript
```

在您执行该操作时，将安装 CDK 应用程序及其初始化文件：

```
Initializing a new git repository...
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Executing npm install...
✅All done!
```

您的项目结构可能如下所示：



您会注意到我们具有几个重要目录：

- `bin`：初始 bin 文件将创建应用程序。我们在本指南中不会对其进行修改。
- `lib`：lib 目录包含您的堆栈文件。您可以将堆栈文件视为单独的执行单元。构造将位于我们的堆栈文件中。基本上，这些是部署应用程序时在 AWS CloudFormation 中启动的服务的资源。这是我们完成大部分编码的位置。
- `node_modules`：该目录由 NPM 创建，并包含您使用 `npm` 命令安装的所有包依赖项。

我们的初始堆栈文件可能包含如下内容：

```
import * as cdk from 'aws-cdk-lib';
import { Construct } from 'constructs';
// import * as sqs from 'aws-cdk-lib/aws-sqs';

export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);
```

```
    // The code that defines your stack goes here

    // example resource
    // const queue = new sqs.Queue(this, 'ExampleCdkAppQueue', {
    //   visibilityTimeout: cdk.Duration.seconds(300)
    // });
  }
}
```

这是在我们的应用程序中创建堆栈的样板代码。该示例中的大部分代码都位于该类的范围内。

要验证您的堆栈文件是否位于应用程序的目录中，请在终端中运行以下命令：

```
cdk ls
```

应该会显示您的堆栈列表。如果没有显示，您可能需要再次执行这些步骤，或者查看官方文档以获得帮助。

如果要在部署之前构建代码更改，您始终可以在终端中运行以下命令：

```
npm run build
```

要在部署之前查看更改，请运行以下命令：

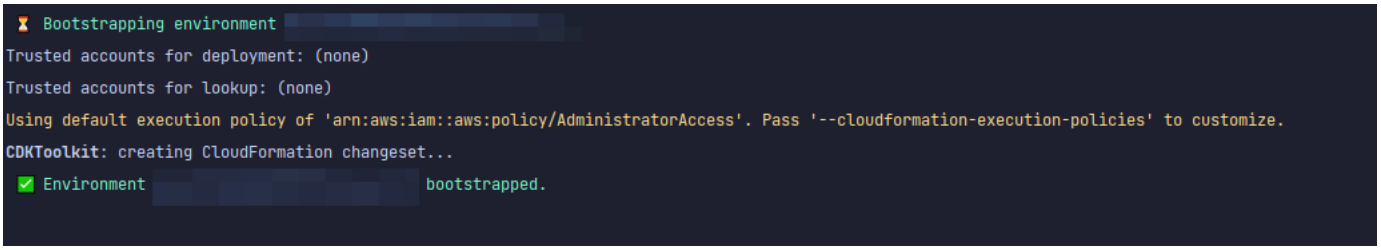
```
cdk diff
```

在将代码添加到堆栈文件之前，我们将执行引导。通过进行引导，我们可以在部署应用程序之前为 CDK 预置资源。可以在此处找到有关该过程的更多信息。要创建引导，请运行以下命令：

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

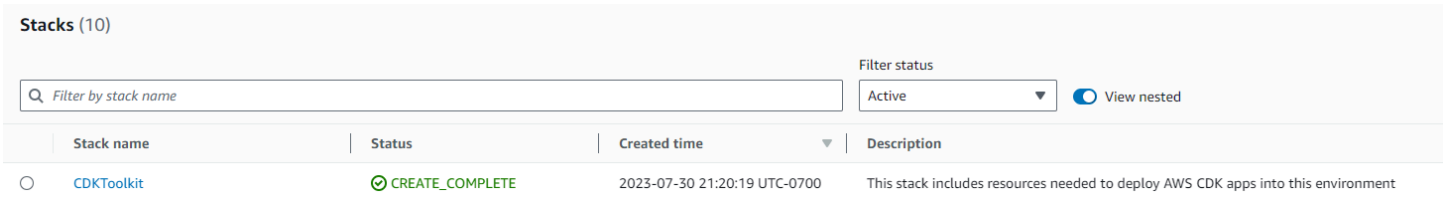
> **Tip**
>
> 该步骤要求在您的账户中具有多个 IAM 权限。如果您没有这些权限，将拒绝您的引导。如果发生这种情况，您可能必须删除引导导致的不完整资源，例如，引导生成的 S3 存储桶。

引导将启动多个资源。最终消息将如下所示：

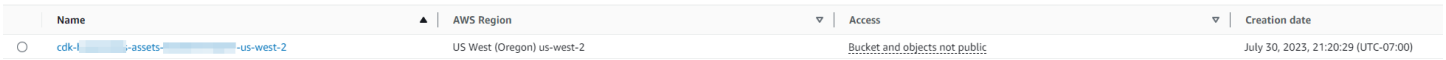


该操作对每个账户的每个区域执行一次，因此，您无需经常这样做。引导的主要资源是 AWS CloudFormation 堆栈和 Amazon S3 存储桶。

Amazon S3 存储桶用于存储文件和 IAM 角色，这些角色授予执行部署所需的权限。所需的资源是在 AWS CloudFormation 堆栈中定义的，该堆栈称为引导堆栈，它通常命名为 `CDKToolkit`。与任何 AWS CloudFormation 堆栈一样，在部署后，它显示在 AWS CloudFormation 控制台中：



存储桶也是如此：



要在堆栈文件中导入所需的服务，我们可以使用以下命令：

```
npm install aws-cdk-lib # V2 command
```

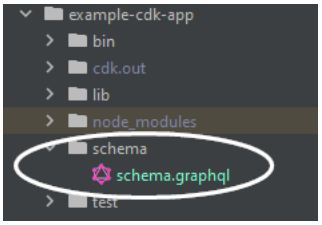
> **Tip**
>
> 如果您在使用 V2 时遇到问题，您可以使用 V1 命令安装各个库：
>
> ```
> npm install @aws-cdk/aws-appsync @aws-cdk/aws-dynamodb
> ```
>
> 我们不建议这样做，因为 V1 已被弃用。

实施 CDK 项目 - 架构

我们现在可以开始实施代码了。首先，我们必须创建架构。您只需在应用程序中创建一个 `.graphql` 文件即可：

```
mkdir schema
touch schema.graphql
```

在我们的示例中，我们包含一个名为 schema 的顶级目录，其中包含 schema.graphql：



在我们的架构中，让我们包含一个简单的示例：

```
input CreatePostInput {
    title: String
    content: String
}

type Post {
    id: ID!
    title: String
    content: String
}

type Mutation {
    createPost(input: CreatePostInput!): Post
}

type Query {
    getPost: [Post]
}
```

回到我们的堆栈文件，我们需要确保定义了以下 import 指令：

```
import * as cdk from 'aws-cdk-lib';
import * as appsync from 'aws-cdk-lib/aws-appsync';
import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';
import { Construct } from 'constructs';
```

在类中，我们将添加代码以创建 GraphQL API 并将其连接到 schema.graphql 文件：

```
export class ExampleCdkAppStack extends cdk.Stack {
```

```
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // makes a GraphQL API
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });
  }
}
```

我们还会添加一些代码以输出 GraphQL URL、API 密钥和区域：

```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Makes a GraphQL API construct
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });

    // Prints out URL
    new cdk.CfnOutput(this, "GraphQLAPIURL", {
      value: api.graphqlUrl
    });

    // Prints out the AppSync GraphQL API key to the terminal
    new cdk.CfnOutput(this, "GraphQLAPIKey", {
      value: api.apiKey || ''
    });

    // Prints out the stack region to the terminal
    new cdk.CfnOutput(this, "Stack Region", {
      value: this.region
    });
  }
}
```

此时，再次部署我们的应用程序：

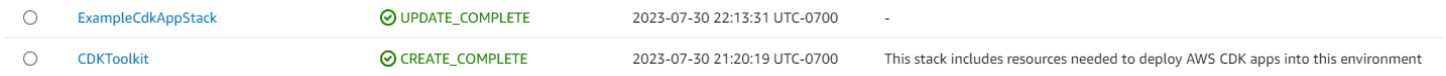
```
cdk deploy
```

结果如下所示：

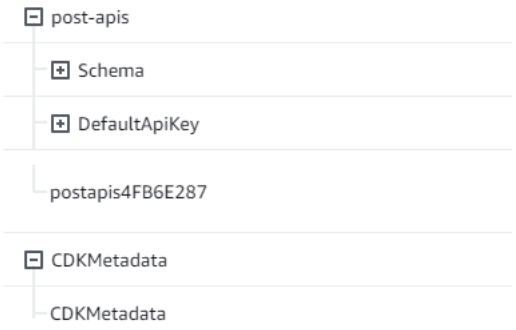
看来我们的示例成功了，但让我们直接检查 AWS AppSync 控制台以进行确认：

看来已创建了我们的 API。现在，我们将检查附加到 API 的架构：

这似乎与我们的架构代码匹配，因此，它是成功的。从元数据的角度，确认这一点的另一种方法是查看 AWS CloudFormation 堆栈：



在我们部署 CDK 应用程序时，它会通过 AWS CloudFormation 启动资源，例如引导。我们的应用程序中的每个堆栈都与 AWS CloudFormation 堆栈 1:1 映射。如果您回到堆栈代码，就会发现堆栈名称是从类名称 `ExampleCdkAppStack` 中获取的。您可以看到它创建的资源，这些资源也符合 GraphQL API 构造中的命名约定：



实施 CDK 项目 - 数据源

接下来，我们需要添加数据源。我们的示例使用 DynamoDB 表。在堆栈类中，我们添加一些代码以创建新的表：

```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    // Makes a GraphQL API construct
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });

    //creates a DDB table
    const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
      partitionKey: {
        name: 'id',
        type: dynamodb.AttributeType.STRING,
      },
    });

    // Prints out URL
    new cdk.CfnOutput(this, "GraphQLAPIURL", {
```

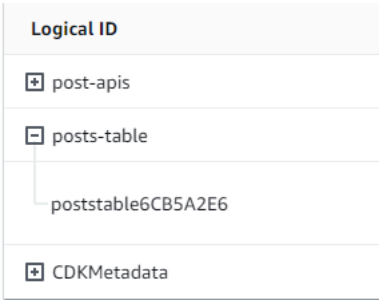
```
      value: api.graphqlUrl
    });

    // Prints out the AppSync GraphQL API key to the terminal
    new cdk.CfnOutput(this, "GraphQLAPIKey", {
      value: api.apiKey || ''
    });

    // Prints out the stack region to the terminal
    new cdk.CfnOutput(this, "Stack Region", {
      value: this.region
    });
  }
}
```

此时，让我们再次部署：

```
cdk deploy
```

我们应该检查 DynamoDB 控制台是否有新的表：

我们的堆栈名称是正确的，并且表名称与代码匹配。如果我们再次检查 AWS CloudFormation 堆栈，我们现在将看到新的表：



实施 CDK 项目 - 解析器

该示例将使用两个解析器：一个用于查询表，另一个用于在表中添加数据。由于我们使用管道解析器，因此，我们需要声明两个管道解析器，在每个解析器中具有一个函数。在查询中，我们添加以下代码：

```
export class ExampleCdkAppStack extends cdk.Stack {
  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);
```

```
    // Makes a GraphQL API construct
    const api = new appsync.GraphqlApi(this, 'post-apis', {
      name: 'api-to-process-posts',
      schema: appsync.SchemaFile.fromAsset('schema/schema.graphql'),
    });

    //creates a DDB table
    const add_ddb_table = new dynamodb.Table(this, 'posts-table', {
      partitionKey: {
        name: 'id',
        type: dynamodb.AttributeType.STRING,
      },
    });

    // Creates a function for query
    const add_func = new appsync.AppsyncFunction(this, 'func-get-post', {
      name: 'get_posts_func_1',
      api,
      dataSource: api.addDynamoDbDataSource('table-for-posts', add_ddb_table),
      code: appsync.Code.fromInline(`
          export function request(ctx) {
          return { operation: 'Scan' };
          }

          export function response(ctx) {
          return ctx.result.items;
          }
  `),
      runtime: appsync.FunctionRuntime.JS_1_0_0,
    });

    // Creates a function for mutation
    const add_func_2 = new appsync.AppsyncFunction(this, 'func-add-post', {
      name: 'add_posts_func_1',
      api,
      dataSource: api.addDynamoDbDataSource('table-for-posts-2', add_ddb_table),
      code: appsync.Code.fromInline(`
          export function request(ctx) {
            return {
            operation: 'PutItem',
            key: util.dynamodb.toMapValues({id: util.autoId()}),
            attributeValues: util.dynamodb.toMapValues(ctx.args.input),
            };
          }
```

```
        export function response(ctx) {
          return ctx.result;
        }
    `),
    runtime: appsync.FunctionRuntime.JS_1_0_0,
  });

  // Adds a pipeline resolver with the get function
  new appsync.Resolver(this, 'pipeline-resolver-get-posts', {
    api,
    typeName: 'Query',
    fieldName: 'getPost',
    code: appsync.Code.fromInline(`
        export function request(ctx) {
        return {};
        }

        export function response(ctx) {
        return ctx.prev.result;
        }
`),
    runtime: appsync.FunctionRuntime.JS_1_0_0,
    pipelineConfig: [add_func],
  });

  // Adds a pipeline resolver with the create function
  new appsync.Resolver(this, 'pipeline-resolver-create-posts', {
    api,
    typeName: 'Mutation',
    fieldName: 'createPost',
    code: appsync.Code.fromInline(`
        export function request(ctx) {
        return {};
        }

        export function response(ctx) {
        return ctx.prev.result;
        }
`),
    runtime: appsync.FunctionRuntime.JS_1_0_0,
    pipelineConfig: [add_func_2],
  });
```

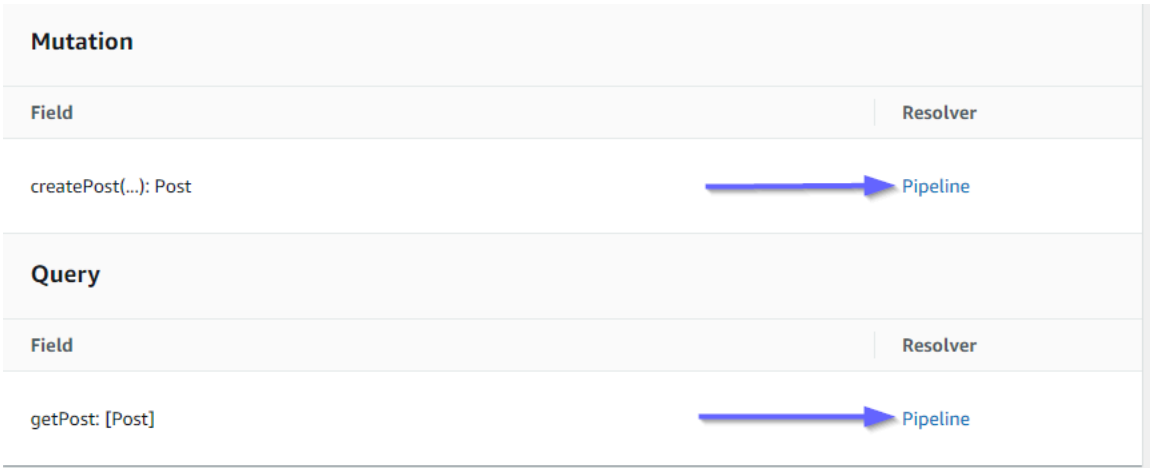
```
    // Prints out URL
    new cdk.CfnOutput(this, "GraphQLAPIURL", {
      value: api.graphqlUrl
    });

    // Prints out the AppSync GraphQL API key to the terminal
    new cdk.CfnOutput(this, "GraphQLAPIKey", {
      value: api.apiKey || ''
    });

    // Prints out the stack region to the terminal
    new cdk.CfnOutput(this, "Stack Region", {
      value: this.region
    });
  }
}
```

在该代码片段中，我们添加一个名为 `pipeline-resolver-create-posts` 的管道解析器，并将一个名为 `func-add-post` 的函数附加到该解析器。这是将 `Posts` 添加到表中的代码。另一个管道解析器命名为 `pipeline-resolver-get-posts`，并具有一个名为 `func-get-post` 的函数，该函数检索添加到表中的 `Posts`。

我们将部署该代码以将其添加到 AWS AppSync 服务中：

```
cdk deploy
```

让我们检查 AWS AppSync 控制台，看看它们是否附加到我们的 GraphQL API：



看来是正确的。在代码中，这两个解析器都附加到我们创建的 GraphQL API（由解析器和函数中存在的 `api` 属性值表示）。在 GraphQL API 中，我们将解析器附加到的字段也是在属性中指定的（由每个解析器中的 `typename` 和 `fieldname` 属性定义）。

让我们看看解析器内容是否正确，从 `pipeline-resolver-get-posts` 开始：

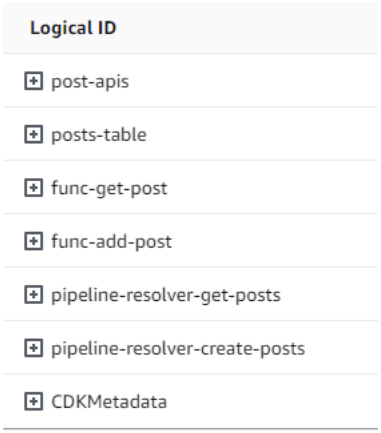
之前和之后处理程序与 `code` 属性值匹配。我们还可以看到一个名为 `add_posts_func_1` 的函数，它与我们在解析器中附加的函数的名称匹配。

我们看一下该函数的代码内容：

这与 add_posts_func_1 函数的 code 属性匹配。已成功上传我们的查询，因此，让我们检查一下该查询：

这些也与代码匹配。如果我们看一下 `get_posts_func_1`：

一切似乎准备就绪。要从元数据角度确认这一点，我们可以再次在 AWS CloudFormation 中检查堆栈：



现在，我们需要执行一些请求以测试该代码。

实施 CDK 项目 - 请求

要在 AWS AppSync 控制台中测试我们的应用程序，我们创建了一个查询和一个变更：

```
query MyQuery {
  getPost {
    id
    date
    title
  }
}

mutation MyMutation {
  createPost(input: {date: "1970-01-01T12:30:00.000Z", title: "first post"}) {
    date
    id
    title
  }
}

```

MyMutation 包含具有 createPost 和 1970-01-01T12:30:00.000Z 参数的 first post 操作。它返回我们传入的 date 和 title 以及自动生成的 id 值。运行该变更将产生以下结果：

```
{
  "data": {
    "createPost": {
      "date": "1970-01-01T12:30:00.000Z",
      "id": "4dc1c2dd-0aa3-4055-9eca-7c140062ada2",
      "title": "first post"
    }
  }
}
```

如果快速检查 DynamoDB 表，我们可以在扫描时在该表中看到我们的条目：

id *(String)*	date	title
9f62c4dd-49d5-48d5-b835-143284c72fe0	1970-01-01T12:30:00.000Z	first post

返回到 AWS AppSync 控制台，如果我们运行查询以检索该 Post，我们将获得以下结果：

```
{
  "data": {
    "getPost": [
      {
        "id": "9f62c4dd-49d5-48d5-b835-143284c72fe0",
        "date": "1970-01-01T12:30:00.000Z",
        "title": "first post"
      }
    ]
```

```
  }
}
```

实时数据

AWS AppSync 允许您使用订阅实施实时应用程序更新、推送通知等。在客户端调用 GraphQL 订阅操作时，AWS AppSync 自动建立并维护安全的 WebSocket 连接。然后，应用程序可以将数据从数据源实时分配给订阅者，同时 AWS AppSync 持续管理应用程序的连接和扩展要求。以下几节说明了订阅在 AWS AppSync 中的工作方式。

GraphQL 架构订阅指令

AWS AppSync 中的订阅是作为变更响应调用的。这意味着，您可以在变更上指定 GraphQL 架构指令，以在 AWS AppSync 中实时生成任何数据源。

AWS Amplify 客户端库自动处理订阅连接管理。这些库将纯 WebSockets 作为客户端和服务之间的网络协议。

> **Note**
>
> 要控制连接到订阅时的授权，您可以使用 AWS Identity and Access Management (IAM)、AWS Lambda、Amazon Cognito 身份池或 Amazon Cognito 用户池进行字段级授权。要对订阅进行精细的访问控制，您可以将解析器附加到您的订阅字段，并使用调用方身份和 AWS AppSync 数据源执行逻辑。有关更多信息，请参阅授权和身份验证。

订阅由变更触发，并将变更选择集发送给订阅者。

以下示例展示了如何使用 GraphQL 订阅。该示例未指定数据源，因为数据源可以是 Lambda、Amazon DynamoDB 或 Amazon OpenSearch Service。

要开始使用订阅，您必须将订阅入口点添加到您的架构中，如下所示：

```
schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
}
```

假设有一个博客站点，您希望订阅新博文和现有博客的变更。为此，请在架构中添加以下 Subscription 定义：

```
type Subscription {
    addedPost: Post
    updatedPost: Post
    deletedPost: Post
}
```

进一步假设有以下变更：

```
type Mutation {
    addPost(id: ID! author: String! title: String content: String url: String): Post!
    updatePost(id: ID! author: String! title: String content: String url: String ups:
 Int! downs: Int! expectedVersion: Int!): Post!
    deletePost(id: ID!): Post!
}
```

对于希望收到通知的每个订阅，您可以添加 @aws_subscribe(mutations: ["mutation_field_1", "mutation_field_2"]) 指令，使这些字段成为实时字段，如下所示：

```
type Subscription {
    addedPost: Post
    @aws_subscribe(mutations: ["addPost"])
    updatedPost: Post
    @aws_subscribe(mutations: ["updatePost"])
    deletedPost: Post
    @aws_subscribe(mutations: ["deletePost"])
}
```

由于 @aws_subscribe(mutations: ["",..,""]) 使用变更输入数组，因此，您可以指定多个启动订阅的变更。如果您从客户端订阅，您的 GraphQL 查询可能是下面的样子：

```
subscription NewPostSub {
    addedPost {
        __typename
        version
        title
        content
        author
        url
```

```
    }
}
```

客户端连接和工具需要使用该订阅查询。

对于纯 WebSockets 客户端，选择集筛选是按客户端完成的，因为每个客户端可以定义自己的选择集。在这种情况下，订阅选择集必须是变更选择集的子集。例如，链接到变更 `addPost(...){id author title url version}` 的订阅 `addedPost{author title}` 仅接收文章的作者和标题。它不会接收其他字段。但是，如果变更在其选择集中缺少作者，则订阅者将获得作者字段的 `null` 值（或者，如果在架构中将作者字段定义为必填/非 Null 的情况下，将得到错误）。

在使用纯 WebSockets 时，订阅选择集是至关重要的。如果在订阅中未明确定义某个字段，则 AWS AppSync 不会返回该字段。

在以上示例中，订阅没有参数。假设您的架构如下所示：

```
type Subscription {
    updatedPost(id:ID! author:String): Post
    @aws_subscribe(mutations: ["updatePost"])
}
```

在这种情况下，您的客户端定义了订阅，如下所示：

```
subscription UpdatedPostSub {
    updatedPost(id:"XYZ", author:"ABC") {
        title
        content
    }
}
```

您的架构中 `subscription` 字段的返回类型必须与相应的变更字段的返回类型匹配。在上一示例中，`addPost` 和 `addedPost` 返回的类型都是 `Post`。

要在客户端上设置订阅，请参阅构建客户端应用程序。

使用订阅参数

使用 GraphQL 订阅的一个重要部分是，了解何时以及如何使用参数。您可以进行细微的更改，以修改何时以及如何向客户端通知发生的变更。为此，请参阅快速入门章节中创建“Todo”的示例架构。对于该示例架构，定义了以下变更：

```
type Mutation {
    createTodo(input: CreateTodoInput!): Todo
    updateTodo(input: UpdateTodoInput!): Todo
    deleteTodo(input: DeleteTodoInput!): Todo
}
```

在默认示例中，客户端可以使用不带参数的 onUpdateTodo subscription 订阅任何 Todo 的更新：

```
subscription OnUpdateTodo {
  onUpdateTodo {
    description
    id
    name
    when
  }
}
```

您可以使用 subscription 的参数对其进行筛选。例如，要仅在更新具有特定 ID 的 todo 时触发 subscription，请指定 ID 值：

```
subscription OnUpdateTodo {
  onUpdateTodo(id: "a-todo-id") {
    description
    id
    name
    when
  }
}
```

您也可以传递多个参数。例如，以下 subscription 说明了如何在特定地点和时间获取任何 Todo 更新的通知：

```
subscription todosAtHome {
  onUpdateTodo(when: "tomorrow", where: "at home") {
    description
    id
    name
    when
    where
```

```
  }
}
```

请注意，所有参数都是可选的。如果未在 subscription 中指定任何参数，您将订阅应用程序中发生的所有 Todo 更新。不过，您可以更新 subscription 的字段定义以要求使用 ID 参数。这会强制提供特定 todo 的响应，而不是提供所有 todo 的响应：

```
onUpdateTodo(
  id: ID!,
  name: String,
  when: String,
  where: String,
  description: String
): Todo
```

参数 null 值具有含义

在 AWS AppSync 中进行订阅查询时，null 参数值筛选结果的方式与完全省略该参数不同。

让我们回到创建 Todo 的 Todo API 示例。请参阅快速入门章节中的示例架构。

让我们修改架构以在 Todo 类型上包含新的 owner 字段，该字段描述所有者是谁。owner 字段不是必填字段，只能在 UpdateTodoInput 上设置该字段。请参阅以下简化架构版本：

```
type Todo {
  id: ID!
  name: String!
  when: String!
  where: String!
  description: String!
  owner: String
}

input CreateTodoInput {
  name: String!
  when: String!
  where: String!
  description: String!
}

input UpdateTodoInput {
```

```
  id: ID!
  name: String
  when: String
  where: String
  description: String
  owner: String
}

type Subscription {
    onUpdateTodo(
        id: ID,
        name: String,
        when: String,
        where: String,
        description: String
    ): Todo @aws_subscribe(mutations: ["updateTodo"])
}
```

以下订阅返回所有 Todo 更新：

```
subscription MySubscription {
  onUpdateTodo {
    description
    id
    name
    when
    where
  }
}
```

如果您修改前面的订阅以添加字段参数 `owner: null`，您现在会问一个不同的问题。该订阅现在注册客户端，以获得所有未提供所有者的 Todo 更新的通知。

```
subscription MySubscription {
  onUpdateTodo(owner: null) {
    description
    id
    name
    when
    where
  }
}
```

> **Note**
>
> 自 2022 年 1 月 1 日起，基于 WebSockets 的 MQTT 不再作为 AWS AppSync API 中的 GraphQL 订阅协议提供。纯 WebSockets 是 AWS AppSync 中唯一支持的协议。
> 默认情况下，基于 AWS AppSync SDK 或 Amplify 库（在 2019 年 11 月后发布）的客户端自动使用纯 WebSockets。通过将客户端升级到最新版本，它们可以使用 AWS AppSync 的纯 WebSockets 引擎。
> 纯 WebSockets 具有更大的负载大小 (240 KB)、更广泛的客户端选项以及改进的 CloudWatch 指标。有关使用纯 WebSocket 客户端的更多信息，请参阅构建实时 WebSocket 客户端。

创建由无服务器 WebSockets 支持的通用 Pub/Sub API

某些应用程序仅需要使用简单的 WebSocket API，客户端可以在其中侦听特定的通道或主题。对于没有特定形状或强类型要求的通用 JSON 数据，可以将其推送到使用纯粹且简单的发布-订阅 (Pub/Sub) 模式侦听这些通道之一的客户端。

可以使用 AWS AppSync 在 API 后端和客户端自动生成 GraphQL 代码，在几分钟内即可实施简单的 Pub/Sub WebSocket API，几乎不需要了解任何 GraphQL 知识。

创建和配置 Pub-Sub API

首先，执行以下操作：

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   - 在控制面板中，选择创建 API。
2. 在下一个屏幕上，选择创建一个实时 API，然后选择下一步。
3. 输入您的 Pub/Sub API 的友好名称。
4. 您可以启用私有 API 功能，但我们建议暂时关闭该功能。选择下一步。
5. 您可以选择使用 WebSockets 自动生成正常工作的 Pub/Sub API。我们建议暂时也关闭该功能。选择下一步。
6. 选择创建 API，然后等待几分钟。将在您的 AWS 账户中创建一个新的预预置 AWS AppSync Pub/Sub API。

该 API 使用 AWS AppSync 的内置本地解析器（有关使用本地解析器的更多信息，请参阅《AWS AppSync 开发人员指南》中的教程：本地解析器）管理多个临时 Pub/Sub 通道和 WebSocket 连接，

这些连接自动向订阅的客户端传送数据，并仅根据通道名称进行筛选。API 调用通过 API 密钥进行授权。

在部署 API 后，您需要执行几个额外的步骤以生成客户端代码，并将其与客户端应用程序集成在一起。本指南使用一个简单的 React Web 应用程序，以提供如何快速集成客户端的示例。

1. 首先，在本地计算机上使用 NPM 创建一个样板 React 应用程序：

```
$ npx create-react-app mypubsub-app
$ cd mypubsub-app
```

> **Note**
>
> 该示例使用 Amplify 库将客户端连接到后端 API。不过，无需在本地创建 Amplify CLI 项目。虽然 React 是该示例中选择的客户端，但 Amplify 库还支持 iOS、Android 和 Flutter 客户端，并在这些不同的运行时环境中提供相同的功能。支持的 Amplify 客户端提供简单的抽象，只需编写几行代码，即可与 AWS AppSync GraphQL API 后端进行交互，包括与 AWS AppSync 实时 WebSocket 协议完全兼容的内置 WebSocket 功能：
>
> ```
> $ npm install @aws-amplify/api
> ```

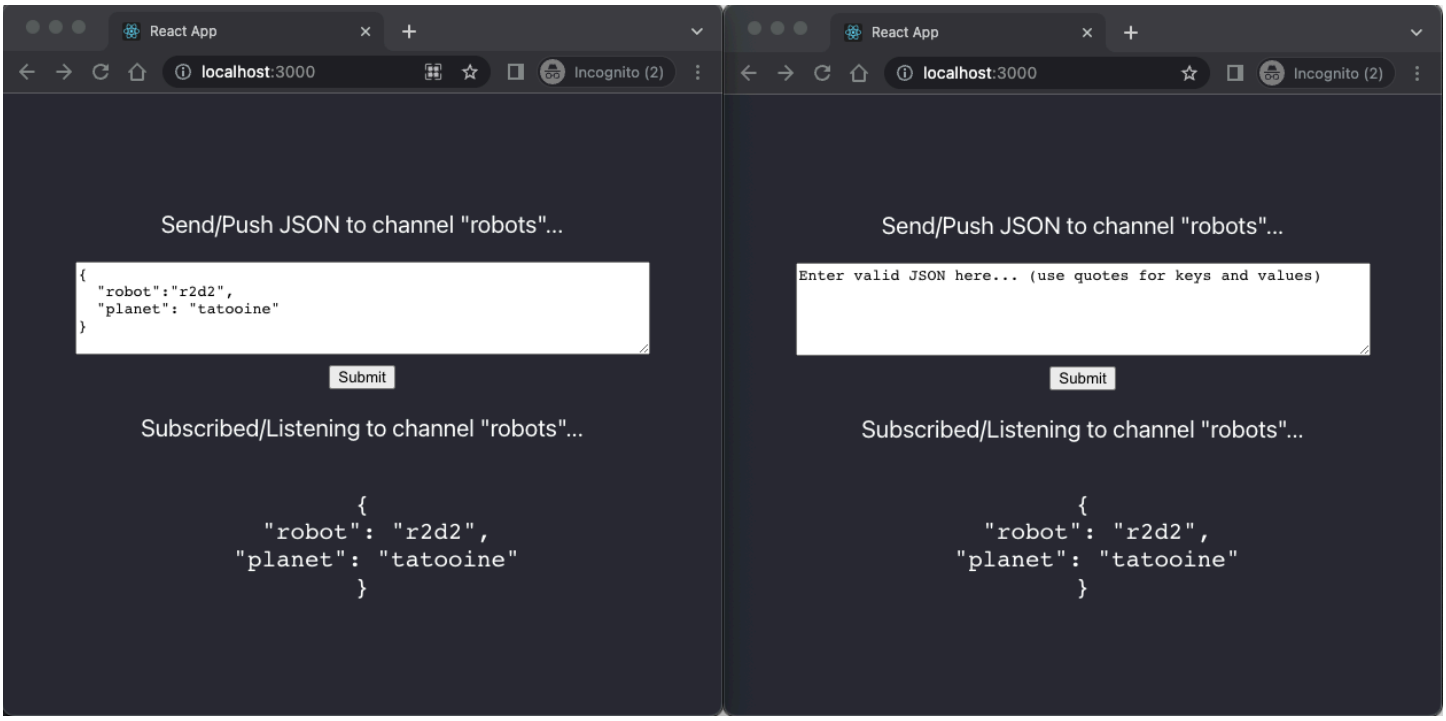
2. 在 AWS AppSync 控制台中，选择 JavaScript，然后选择下载以下载一个包含 API 配置详细信息和生成的 GraphQL 操作代码的文件。
3. 将下载的文件复制到 React 项目中的 `/src` 文件夹。
4. 接下来，将现有的样板 `src/App.js` 文件内容替换为控制台中提供的示例客户端代码。
5. 使用以下命令在本地启动该应用程序：

```
$ npm start
```

6. 要测试发送和接收实时数据的过程，请打开两个浏览器窗口并访问 *`localhost:3000`*。示例应用程序配置为将通用 JSON 数据发送到名为 *`robots`* 的硬编码通道。
7. 在其中的一个浏览器窗口中，在文本框中输入以下 JSON Blob，然后单击提交：

```
{
  "robot":"r2d2",
  "planet": "tatooine"
}
```

两个浏览器实例都订阅了 *robots* 通道并实时接收发布的数据（显示在 Web 应用程序底部）：



将自动生成所需的所有 GraphQL API 代码（包括架构、解析器和操作）以实现通用的 Pub/Sub 使用案例。在后端，数据通过 GraphQL 变更发布到 AWS AppSync 的实时终端节点，如下所示：

```
mutation PublishData {
    publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
        data
        name
    }
}
```

订阅者通过相关的 GraphQL 订阅访问发送到特定临时通道的发布数据：

```
subscription SubscribeToData {
    subscribe(name:"channel") {
        name
        data
    }
}
```

在现有应用程序中实施 Pub-Sub API

如果您只需在现有应用程序中实施实时功能，可以将这种通用的 Pub/Sub API 配置轻松集成到任何应用程序或 API 技术中。虽然使用单个 API 终端节点通过 GraphQL 在单个网络调用中安全地访问、处理和合并来自一个或多个数据源的数据具有优势，但无需转换现有的基于 REST 的应用程序或从头开始重建以利用 AWS AppSync 的实时功能。例如，您可能在单独的 API 终端节点中具有现有的 CRUD 工作负载，其中客户端在现有应用程序和通用 Pub/Sub API 之间发送和接收消息或事件以仅实现实时和 Pub/Sub 目的。

增强订阅筛选

在 AWS AppSync 中，您可以使用支持其他逻辑运算符的筛选条件，直接在 GraphQL API 订阅解析器中为后端数据筛选定义和启用业务逻辑。您可以配置这些筛选条件，这与在客户端的订阅查询上定义的订阅参数不同。有关使用订阅参数的更多信息，请参阅使用订阅参数。有关运算符列表，请参阅解析器映射模板实用程序参考。

就本文而言，我们将实时数据筛选分为以下几个类别：

- 基本筛选 - 根据订阅查询中的客户端定义的参数进行筛选。
- 增强筛选 - 根据 AWS AppSync 服务后端集中定义的逻辑进行筛选。

以下几节介绍了如何配置增强订阅筛选条件，并说明了它们的实际用途。

在 GraphQL 架构中定义订阅

要使用增强订阅筛选条件，您可以在 GraphQL 架构中定义订阅，然后使用筛选扩展定义增强筛选条件。要说明增强订阅筛选在 AWS AppSync 中的工作方式，请将以下 GraphQL 架构（定义票证管理系统 API）作为示例：

```
type Ticket {
 id: ID
 createdAt: AWSDateTime
 content: String
 severity: Int
 priority: Priority
 category: String
 group: String
 status: String

}
```

```
type Mutation {
 createTicket(input: TicketInput): Ticket
}

type Query {
 getTicket(id: ID!): Ticket
}

type Subscription {
 onSpecialTicketCreated: Ticket @aws_subscribe(mutations: ["createTicket"])
 onGroupTicketCreated(group: String!): Ticket @aws_subscribe(mutations:
 ["createTicket"])
}



enum Priority {
 none
 lowest
 low
 medium
 high
 highest
}

input TicketInput {
 content: String
 severity: Int
 priority: Priority
 category: String
 group: String
```

假设您为 API 创建一个 NONE 数据源，然后使用该数据源将一个解析器附加到 `createTicket` 变更。您的处理程序可能如下所示：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
 return {
  payload: {
   id: util.autoId(),
```

```
    createdAt: util.time.nowISO8601(),
    status: 'pending',
    ...ctx.args.input,
   },
  };
}

export function response(ctx) {
 return ctx.result;
}
```

> **Note**
>
> 增强筛选条件是在给定订阅的 GraphQL 解析器处理程序中启用的。有关更多信息，请参阅解析器参考。

要实施增强筛选条件的行为，您必须使用 `extensions.setSubscriptionFilter()` 函数定义一个筛选条件表达式，该表达式针对通过 GraphQL 变更发布并且订阅客户端可能感兴趣的数据进行评估。有关筛选扩展的更多信息，请参阅扩展。

以下几节介绍了如何使用筛选扩展实施增强筛选条件。

使用筛选扩展创建增强订阅筛选条件

增强筛选条件是在订阅解析器的响应处理程序中以 JSON 格式编写的。可以将筛选条件组合到一个名为 `filterGroup` 的列表中。筛选条件是使用至少一个规则定义的，每个规则包含字段、运算符和值。让我们为 `onSpecialTicketCreated` 定义一个新的解析器以设置增强筛选条件。您可以在一个筛选条件中配置多个使用 AND 逻辑进行评估的规则，而一个筛选条件组中的多个筛选条件使用 OR 逻辑进行评估：

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
 // simplfy return null for the payload
 return { payload: null };
}

export function response(ctx) {
 const filter = {
  or: [
```

```
    { severity: { ge: 7 }, priority: { in: ['high', 'medium'] } },
    { category: { eq: 'security' }, group: { in: ['admin', 'operators'] } },
   ],
  };
  extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

   // important: return null in the response
  return null;
}
```

根据前面示例中定义的筛选条件，如果使用以下条件创建了票证，则将重要票证自动推送到订阅的 API 客户端：

- `priority` 级别 `high` 或 `medium`

  AND

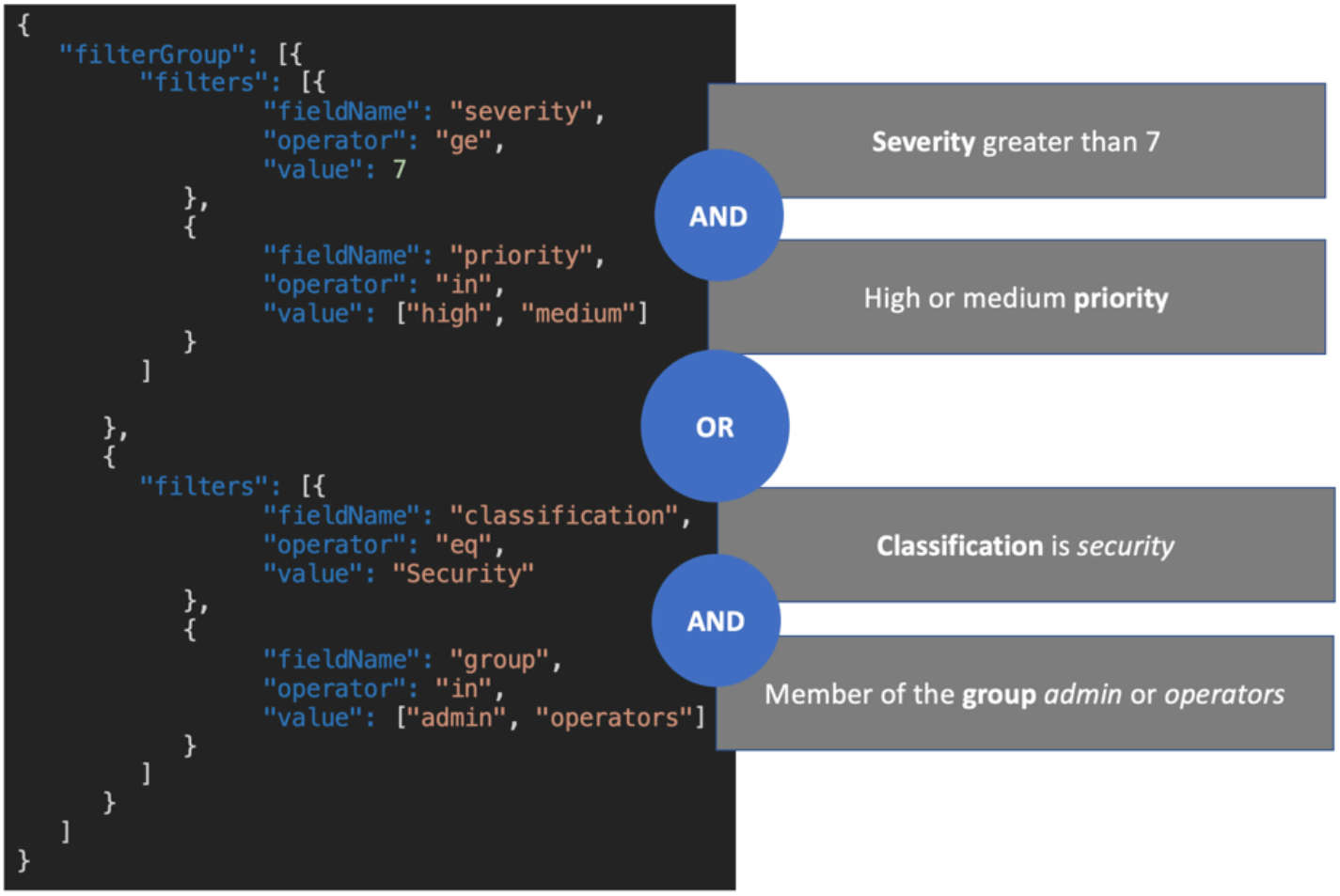
- 大于或等于 7 (ge) 的 `severity` 级别

或

- 设置为 `Security` 的 `classification` 票证

  AND

- 设置为 `admin` 或 `operators` 的 `group` 分配



订阅解析器中定义的筛选条件（增强筛选）优先于仅基于订阅参数的筛选（基本筛选）。有关使用订阅参数的更多信息，请参阅使用订阅参数。

如果在订阅的 GraphQL 架构中定义某个参数，并且该参数是必需的，只有在解析器的 `extensions.setSubscriptionFilter()` 方法中将给定参数定义为规则时，才会根据该参数进行筛选。不过，如果在订阅解析器中没有 `extensions` 筛选方法，则客户端中定义的参数仅用于基本筛选。您不能同时使用基本筛选和增强筛选。

您可以在订阅的筛选条件扩展逻辑中使用 `context` 变量，以访问有关请求的上下文信息。例如，在使用 Amazon Cognito 用户池、OIDC 或 Lambda 自定义授权者进行授权时，您可以在设置订阅后在 `context.identity` 中检索有关您的用户的信息。您可以使用该信息根据用户身份设置筛选条件。

现在假设您希望实施 `onGroupTicketCreated` 的增强筛选条件行为。`onGroupTicketCreated` 订阅要求将必需的 `group` 名称作为参数。在创建后，将自动为票证分配 `pending` 状态。您可以设置订阅筛选条件，以仅接收属于提供的组的新创建的票证：

```
import { util, extensions } from '@aws-appsync/utils';
```

```
export function request(ctx) {
 // simplfy return null for the payload
 return { payload: null };
}

export function response(ctx) {
 const filter = { group: { eq: ctx.args.group }, status: { eq: 'pending' } };
 extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

 return null;
}
```

在使用变更发布数据时，如以下示例中所示：

```
mutation CreateTicket {
  createTicket(input: {priority: medium, severity: 2, group: "aws"}) {
    id
    priority
    severity
    status
    group
    createdAt
  }
}
```

在使用 `createTicket` 变更创建票证时，订阅的客户端立即侦听通过 WebSockets 自动推送的数据：

```
subscription OnGroup {
  onGroupTicketCreated(group: "aws") {
    category
    status
    severity
    priority
    id
    group
    createdAt
    content
  }
}
```

客户端可以在没有参数的情况下进行订阅，因为筛选逻辑是在 AWS AppSync 服务中使用增强筛选实施的，这简化了客户端代码。只有在满足定义的筛选条件时，客户端才会收到数据。

为嵌套的架构字段定义增强筛选条件

您可以使用增强订阅筛选以筛选嵌套的架构字段。假设我们修改了上一节中的架构以包含位置和地址类型：

```
type Ticket {
 id: ID
 createdAt: AWSDateTime
 content: String
 severity: Int
 priority: Priority
 category: String
 group: String
 status: String
 location: ProblemLocation
}

type Mutation {
 createTicket(input: TicketInput): Ticket
}

type Query {
 getTicket(id: ID!): Ticket
}

type Subscription {
 onSpecialTicketCreated: Ticket @aws_subscribe(mutations: ["createTicket"])
 onGroupTicketCreated(group: String!): Ticket @aws_subscribe(mutations:
 ["createTicket"])
}

type ProblemLocation {
 address: Address
}

type Address {
 country: String
}

enum Priority {
```

```
 none
 lowest
 low
 medium
 high
 highest
}

input TicketInput {
 content: String
 severity: Int
 priority: Priority
 category: String
 group: String
 location: AWSJSON
```

对于该架构，您可以使用 . 分隔符表示嵌套。以下示例为 location.address.country 下面的嵌套架构字段添加筛选规则。如果票证的地址设置为 USA，则会触发订阅：

```
import { util, extensions } from '@aws-appsync/utils';

export const request = (ctx) => ({ payload: null });

export function response(ctx) {
 const filter = {
  or: [
   { severity: { ge: 7 }, priority: { in: ['high', 'medium'] } },
   { category: { eq: 'security' }, group: { in: ['admin', 'operators'] } },
   { 'location.address.country': { eq: 'USA' } },
  ],
 };
 extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));
 return null;
}
```

在上面的示例中，location 表示嵌套级别 1，address 表示嵌套级别 2，country 表示嵌套级别 3，所有这些字段以 . 分隔符分隔。

您可以使用 createTicket 变更测试该订阅：

```
mutation CreateTicketInUSA {
  createTicket(input: {location: "{\"address\":{\"country\":\"USA\"}}"}) {
    category
```

```
    content
    createdAt
    group
    id
    location {
      address {
        country
      }
    }
    priority
    severity
    status
  }
}
```

从客户端中定义增强筛选条件

您可以在 GraphQL 中通过订阅参数使用基本筛选。在订阅查询中进行调用的客户端定义参数的值。在具有 `extensions` 筛选的 AWS AppSync 订阅解析器中启用增强筛选条件时，解析器中定义的后端筛选条件优先。

使用 `filter` 参数在订阅中配置客户端定义的动态增强筛选条件。在配置这些筛选条件时，您必须更新 GraphQL 架构以反映新参数：

```
...
type Subscription {
    onSpecialTicketCreated(filter: String): Ticket
        @aws_subscribe(mutations: ["createTicket"])
}
...
```

然后，客户端可以发送订阅查询，如以下示例中所示：

```
subscription onSpecialTicketCreated($filter: String) {
     onSpecialTicketCreated(filter: $filter) {
        id
        group
        description
        priority
        severity
     }
 }
```

您可以像以下示例一样配置查询变量：

```
{"filter" : "{\"severity\":{\"le\":2}}"}
```

可以在订阅响应映射模板中实施 `util.transform.toSubscriptionFilter()` 解析器实用程序，以应用每个客户端的订阅参数中定义的筛选条件：

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
 // simplfy return null for the payload
 return { payload: null };
}

export function response(ctx) {
 const filter = ctx.args.filter;
 extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));
 return null;
}
```

通过使用该策略，客户端可以定义自己的筛选条件，以使用增强筛选逻辑和额外的运算符。当给定客户端在安全 WebSocket 连接中调用订阅查询时，将会分配筛选条件。有关用于增强筛选的转换实用程序的更多信息（包括 `filter` 查询变量负载格式），请参阅 JavaScript 解析器概述。

额外的增强筛选限制

以下是几个对增强筛选条件施加额外限制的使用案例：

- 增强筛选条件不支持顶级对象列表筛选。在该使用案例中，对于增强订阅，将忽略来自变更的发布数据。
- AWS AppSync 最多支持 5 个嵌套级别。将忽略超过嵌套级别 5 的架构字段的筛选条件。请考虑下面的 GraphQL 响应。允许使用 `venue.address.country.metadata.continent` 中的 `continent` 字段，因为它是 5 级嵌套。不过，`venue.address.country.metadata.capital.financial` 中的 `financial` 是 6 级嵌套，因此，筛选条件不起作用：

```
{
    "data": {
        "onCreateFilterEvent": {
            "venue": {
```

```
                "address": {
                    "country": {
                        "metadata": {
                            "capital": {
                                "financial": "New York"
                            },
                            "continent" : "North America"
                        }
                    },
                    "state": "WA"
                },
                "builtYear": 2023
            },
            "private": false,
        }
    }
}
```

使用筛选条件取消订阅 WebSocket 连接

在 AWS AppSync 中，您可以根据特定的筛选逻辑从连接的客户端强制取消订阅并关闭 WebSocket 连接（使其失效）。这在与授权相关的场景中非常有用，例如，在您从组中删除用户时。

订阅失效是针对变更中定义的负载做出的响应。我们建议您将用于使订阅连接失效的变更视为 API 中的管理操作，并仅限管理员用户、组或后端服务使用这些变更以相应地设置其权限范围。例如，使用架构授权指令，例如 `@aws_auth(cognito_groups: ["Administrators"])` 或 `@aws_iam`。有关更多信息，请参阅使用其他授权模式。

失效筛选条件使用与增强订阅筛选条件相同的语法和逻辑。请使用以下实用程序定义这些筛选条件：

- `extensions.invalidateSubscriptions()` - 在变更的 GraphQL 解析器响应处理程序中定义。
- `extensions.setSubscriptionInvalidationFilter()` - 在链接到变更的订阅的 GraphQL 解析器响应处理程序中定义。

有关失效筛选扩展的更多信息，请参阅 JavaScript 解析器概述。

使用订阅失效

要了解订阅失效在 AWS AppSync 中的工作方式，请使用以下 GraphQL 架构：

```
type User {
  userId: ID!
  groupId: ID!
}

type Group {
  groupId: ID!
  name: String!
  members: [ID!]!
}

type GroupMessage {
  userId: ID!
  groupId: ID!
  message: String!
}

type Mutation {
    createGroupMessage(userId: ID!, groupId : ID!, message: String!): GroupMessage
    removeUserFromGroup(userId: ID!, groupId : ID!) : User @aws_iam
}

type Subscription {
    onGroupMessageCreated(userId: ID!, groupId : ID!): GroupMessage
        @aws_subscribe(mutations: ["createGroupMessage"])
}

type Query {
 none: String
}
```

在 removeUserFromGroup 变更解析器代码中定义失效筛选条件：

```
import { extensions } from '@aws-appsync/utils';

export function request(ctx) {
 return { payload: null };
}

export function response(ctx) {
 const { userId, groupId } = ctx.args;
 extensions.invalidateSubscriptions({
  subscriptionField: 'onGroupMessageCreated',
```

```
  payload: { userId, groupId },
 });
 return { userId, groupId };
}
```

在调用变更时，payload 对象中定义的数据用于取消订阅 subscriptionField 中定义的订阅。还会在 onGroupMessageCreated 订阅的响应映射模板中定义一个失效筛选条件。

如果 extensions.invalidateSubscriptions() 负载包含的 ID 与筛选条件中定义的订阅客户端的 ID 匹配，则取消订阅相应的订阅。此外，还会关闭 WebSocket 连接。定义 onGroupMessageCreated 订阅的订阅解析器代码：

```
import { util, extensions } from '@aws-appsync/utils';

export function request(ctx) {
 // simplfy return null for the payload
 return { payload: null };
}

export function response(ctx) {
 const filter = { groupId: { eq: ctx.args.groupId } };
 extensions.setSubscriptionFilter(util.transform.toSubscriptionFilter(filter));

 const invalidation = { groupId: { eq: ctx.args.groupId }, userId: { eq:
 ctx.args.userId } };
 extensions.setSubscriptionInvalidationFilter(util.transform.toSubscriptionFilter(invalidation)

 return null;
}
```

请注意，订阅响应处理程序可以同时定义订阅筛选条件和失效筛选条件。

例如，假设客户端 A 使用以下订阅请求为具有 ID *user-1* 的新用户订阅具有 ID *group-1* 的组：

```
onGroupMessageCreated(userId : "user-1", groupId: :"group-1"){...}
```

AWS AppSync 运行订阅解析器，这会生成前面的 onGroupMessageCreated 响应映射模板中定义的订阅和失效筛选条件。对于客户端 A，订阅筛选条件仅允许将数据发送到 *group-1*，并为 *user-1* 和 *group-1* 定义了失效筛选条件。

现在假设客户端 B 使用以下订阅请求为具有 ID *user-2* 的用户订阅具有 ID *group-2* 的组：

```
onGroupMessageCreated(userId : "user-2", groupId: :"group-2"){...}
```

AWS AppSync 运行订阅解析器，这会生成订阅和失效筛选条件。对于客户端 B，订阅筛选条件仅允许将数据发送到 *group-2*，并为 *user-2* 和 *group-2* 定义了失效筛选条件。

接下来，假设使用变更请求创建 ID 为 *message-1* 的新组消息，如以下示例中所示：

```
createGroupMessage(id: "message-1", groupId :
      "group-1", message: "test message"){...}
```

与定义的筛选条件匹配的订阅客户端通过 WebSockets 自动收到以下数据负载：

```
{
  "data": {
    "onGroupMessageCreated": {
      "id": "message-1",
      "groupId": "group-1",
      "message": "test message",
    }
  }
}
```

客户端 A 收到该消息，因为筛选条件与定义的订阅筛选条件匹配。不过，客户端 B 不会收到该消息，因为该用户不属于 *group-1*。此外，请求与订阅解析器中定义的订阅筛选条件不匹配。

最后，假设使用以下变更请求从 *group-1* 中删除了 *user-1*：

```
removeUserFromGroup(userId: "user-1", groupId : "group-1"){...}
```

该变更启动 `extensions.invalidateSubscriptions()` 解析器响应处理程序代码中定义的订阅失效。然后，AWS AppSync 取消订阅客户端 A 并关闭其 WebSocket 连接。客户端 B 不受影响，因为变更中定义的失效负载与其用户或组不匹配。

在 AWS AppSync 使连接失效时，客户端收到一条消息，以确认已将它们取消订阅：

```
{
  "message": "Subscription complete."
}
```

在订阅失效筛选条件中使用上下文变量

与增强订阅筛选条件一样，您可以在订阅失效筛选条件扩展中使用 `context` 变量以访问某些数据。

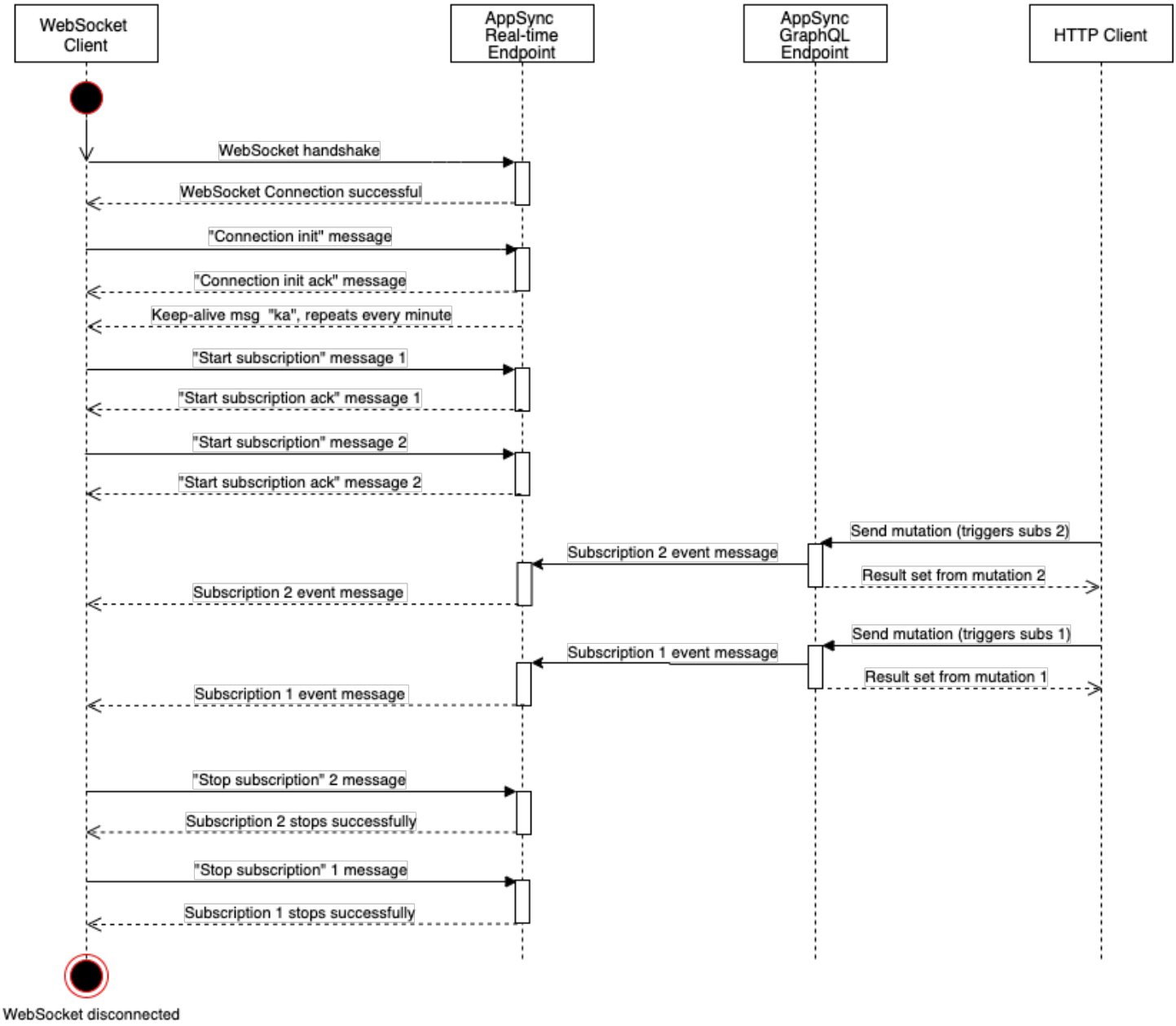
例如，可以在变更中将电子邮件地址配置为失效负载，然后将其与通过 Amazon Cognito 用户池或 OpenID Connect 授权的订阅用户的电子邮件属性或声明进行匹配。失效筛选条件（在 `extensions.setSubscriptionInvalidationFilter()` 订阅失效器中定义）检查变更的 `extensions.invalidateSubscriptions()` 负载设置的电子邮件地址是否与从用户 JWT 令牌的 `context.identity.claims.email` 中检索的电子邮件地址匹配，从而启动失效。

构建实时 WebSocket 客户端

以下几节说明了 AWS AppSync 实时功能背后的架构。

GraphQL 订阅的实时 WebSocket 客户端实施

以下序列图和步骤显示 WebSocket 客户端、HTTP 客户端和 AWS AppSync 之间的实时订阅工作流。



1. 客户端与 AWS AppSync 实时终端节点建立 WebSocket 连接。如果存在网络错误，则客户端应该执行抖动指数退避。有关更多信息，请参阅 AWS 架构博客上的 Exponential backoff and jitter。
2. 在成功建立 WebSocket 连接后，客户端发送 `connection_init` 消息。
3. 客户端等待来自 AWS AppSync 的 `connection_ack` 消息。该消息包含一个 `connectionTimeoutMs` 参数，它是 `"ka"`（保持活动）消息的最长等待时间（以毫秒为单位）。
4. AWS AppSync 定期发送 `"ka"` 消息。客户端跟踪它收到每条 `"ka"` 消息的时间。如果客户端在 `connectionTimeoutMs` 毫秒内未收到 `"ka"` 消息，客户端应关闭连接。

5. 客户端通过发送 `start` 订阅消息来注册订阅。单个 WebSocket 连接支持多个订阅，即使这些订阅处于不同的授权模式。
6. 客户端等待 AWS AppSync 发送 `start_ack` 消息以确认订阅成功。如果出现错误，AWS AppSync 将返回 `"type": "error"` 消息。
7. 客户端侦听订阅事件，这些事件是在调用相应变更后发送的。查询和变更通常通过 `https://` 发送到 AWS AppSync GraphQL 终端节点。订阅使用安全 WebSocket (`wss://`) 流经 AWS AppSync 实时终端节点。
8. 客户端通过发送 `stop` 订阅消息来取消注册订阅。
9. 取消注册所有订阅并检查没有通过 WebSocket 传输的消息后，客户端可以从 WebSocket 连接断开。

建立 WebSocket 连接的握手详细信息

要连接到 AWS AppSync 并启动成功握手，WebSocket 客户端需要满足以下要求：

- AWS AppSync 实时终端节点
- 包含 `header` 和 `payload` 参数的查询字符串：
  - `header`：包含与 AWS AppSync 终端节点和授权相关的信息。这是来自字符串化 JSON 对象的 Base64 编码字符串。JSON 对象内容因授权模式而异。
  - `payload`：`payload` 的 Base64 编码字符串。

在满足这些要求的情况下，WebSocket 客户端可以将 `graphql-ws` 作为 WebSocket 协议以连接到 URL，其中包含具有查询字符串的实时终端节点。

从 GraphQL 终端节点发现 实时终端节点

AWS AppSync GraphQL 终端节点和 AWS AppSync 实时终端节点在协议和域方面略有不同。您可以使用 AWS Command Line Interface (AWS CLI) 命令 `aws appsync get-graphql-api` 检索 GraphQL 终端节点。

AWS AppSync GraphQL 终端节点：

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync 实时终端节点：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

应用程序可以使用任何 HTTP 客户端连接到 AWS AppSync GraphQL 终端节点 (`https://`) 以进行查询和变更。应用程序可以使用任何 WebSocket 客户端连接到 AWS AppSync 实时终端节点 (`wss://`) 以进行订阅。

通过使用自定义域名，您可以使用单个域与两个终端节点进行交互。例如，如果将 `api.example.com` 配置为自定义域，您可以使用以下 URL 与 GraphQL 终端节点和实时终端节点进行交互：

AWS AppSync 自定义域 GraphQL 终端节点：

```
https://api.example.com/graphql
```

AWS AppSync 自定义域实时终端节点：

```
wss://api.example.com/graphql/realtime
```

基于 AWS AppSync API 授权模式的标头参数格式

连接查询字符串中使用的 `header` 对象的格式因 AWS AppSync API 授权模式而异。对象中的 `host` 字段引用 AWS AppSync GraphQL 终端节点，该终端节点用于验证连接，即使针对实时终端节点进行 `wss://` 调用。要启动握手并建立授权连接，`payload` 应为空 JSON 对象。

API 密钥

API 密钥标头

标头内容

- `"host": <string>`：AWS AppSync GraphQL 终端节点主机或您的自定义域名。
- `"x-api-key": <string>`：为 AWS AppSync API 配置的 API 密钥。

示例

```
{
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com",
    "x-api-key":"da2-12345678901234567890123456"
}
```

负载内容

```
{}
```

请求 URL

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJob3N0IjoiZXhhbXBsZTEyMzQ1Njc4OTAwMDAuYXBwc3luYy1hcGkudXMtZWFzdC0xLmFtYXpvbmF3cy5jb20i
```

Amazon Cognito 用户池和 OpenID Connect (OIDC)

Amazon Cognito 和 OIDCheader

标头内容：

- `"Authorization": <string>`：JWT ID 令牌。标头可以使用持有者方案。
- `"host": <string>`：AWS AppSync GraphQL 终端节点主机或您的自定义域名。

示例：

```
{

 "Authorization":"eyEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJHWEFLSXBieU5WNHhsQjEXAMPLEnM2WldvPSIsImFsZ
zEE2DJH7sHOl2zxYi7f-SmEGoh2AD8emxQRYajByz-rE4Jh0QOymN2Ys-ZIkMpVBTPgu-
TMWDyOHhDUmUj2OP82yeZ3wlZAtr_gM4LzjXUXmI_K2yGjuXfXTaa1mvQEBG0mQfVd7SfwXB-
jcv4RYVi6j25qgow9Ew52ufurPqaK-3WAKG32KpV8J4-Wejq8t0c-
yA7sb8EnB551b7TU93uKRiVVK3E55Nk5ADPoam_WYE45i3s5qVAP_-InW75NUoOCGTsS8YWMfb6ecHYJ-1j-
bzA27zaT9VjctXn9byNFZmEXAMPLExw",
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

负载内容：

```
{}
```

请求 URL：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

IAM

IAM 标头

标头内容

- `"accept": "application/json, text/javascript"`：常量 `<string>` 参数。
- `"content-encoding": "amz-1.0"`：常量 `<string>` 参数。
- `"content-type": "application/json; charset=UTF-8"`：常量 `<string>` 参数。
- `"host": <string>`：这是 AWS AppSync GraphQL 终端节点的主机。
  - `"x-amz-date": <string>`：时间戳必须以 UTC 表示并采用以下 ISO 8601 格式：YYYYMMDD'T'HHMMSS'Z'。例如，20150830T123600Z 是一个有效的时间戳。请勿在时间戳中包含毫秒。有关更多信息，请参阅《AWS 一般参考》中的处理签名版本 4 的日期。
  - `"X-Amz-Security-Token": <string>`：AWS 会话令牌，在使用临时安全凭证时，需要使用该令牌。有关更多信息，请参阅《IAM 用户指南》中的将临时凭证用于 AWS 资源。
  - `"Authorization": <string>`：AWS AppSync 终端节点的签名版本 4 (SigV4) 签名信息。有关签名过程的更多信息，请参阅《AWS 一般参考》中的任务 4：将签名添加到 HTTP 请求。

SigV4 签名 HTTP 请求包含一个规范 URL，它是附加了 `/connect` 的 AWS AppSync GraphQL 终端节点。服务终端节点 AWS 区域与您使用 AWS AppSync API 的区域相同，服务名称为“appsync”。要签名的 HTTP 请求如下所示：

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql/
connect",
  data: "{}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

示例

```
{
  "accept": "application/json, text/javascript",
  "content-encoding": "amz-1.0",
  "content-type": "application/json; charset=UTF-8",
  "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
  "x-amz-date": "20200401T001010Z",
  "X-Amz-Security-Token":
 "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXRoZWFEXAMPLEcwRQIgAh97Cljq7wOPL8KsxP3YtDuyc/9hAj8PhJ7Fvf38SgoC
```

```
+
+pEagWCveZUjKEn0zyUhBEXAMPLEjj//////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUdAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtwR+9zF7NaMMMse07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZOzlZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnGlSNVx7SMcfovkFDqQamm
+88y1OwwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVYlj9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEfwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCkcCGFnwBNFLmfnbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpgqJsPKt582caFKArIFIvO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+tOLnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOKtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtKOOweDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA==",
  "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
 Signature=83EXAMPLEbcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}
```

负载内容

```
{}
```

请求 URL

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyEXAMPLEHQiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoE
```

使用自定义域对请求进行签名：

```
{
  url: "https://api.example.com/graphql/connect",
  data: "{}",
  method: "POST",
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
```

```
  }
}
```

示例

```
{
  "accept": "application/json, text/javascript",
  "content-encoding": "amz-1.0",
  "content-type": "application/json; charset=UTF-8",
  "host": "api.example.com",
  "x-amz-date": "20200401T001010Z",
  "X-Amz-Security-Token":
 "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXRoZWFEXAMPLEcwRQIgAh97Cljq7wOPL8KsxP3YtDuyc/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKEn0zyUhBEXAMPLEjj//////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUdAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtwR+9zF7NaMMMse07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZOzlZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnGlSNVx7SMcfovkFDqQamm
+88y1OwwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVYlj9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEfwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
+GYTUyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCkcCGFnwBNFLmfnbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpgqJsPKt582caFKArIFIvO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+tOLnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOKtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtKOOweDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA==",
  "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
 Signature=83EXAMPLEbcc1fe3ee69f75cd5ebbf4cb4f150e4f99cec869f149c5EXAMPLEdc"
}
```

负载内容

```
{}
```

请求 URL

```
wss://api.example.com/graphql?
header=eyEXAMPLEHQiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoE
```

> **Note**
>
> 一个 WebSocket 连接可以具有多个订阅（即使使用不同的授权模式）。实施该功能的一种方法是，为第一个订阅创建 WebSocket 连接，并在取消注册最后一个订阅时关闭该连接。您可以在关闭 WebSocket 连接之前等待几秒钟以进行优化，以防应用程序在取消注册最后一个订阅后立即进行订阅。以移动应用程序为例，从一个屏幕切换到另一个屏幕时，它在发生卸载事件时停止订阅，并在发生挂载事件时启动另一个订阅。

Lambda 授权

Lambda 授权标头

标头内容

- `"Authorization": <string>`：作为 `authorizationToken` 传递的值。
- `"host": <string>`：AWS AppSync GraphQL 终端节点主机或您的自定义域名。

示例

```
{

 "Authorization":"M0UzQzM1MkQtMkI0Ni00OTZCLUI1NkQtMUM0MTQ0QjVBRTczCkI1REEzRTIxLTk5NzItNDJENi1BQ
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

负载内容

```
{}
```

请求 URL

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

实时 WebSocket 操作

在与 AWS AppSync 成功启动 WebSocket 握手后，客户端必须发送一条后续消息，才能连接到 AWS AppSync 以执行不同的操作。这些消息需要以下数据：

- `type`：操作的类型。
- `id`：订阅的唯一标识符。我们建议为此目的使用 UUID。
- `payload`：关联的负载，取决于操作类型。

`type` 字段是唯一的必填字段；`id` 和 `payload` 字段是可选的。

事件序列

要成功启动、建立、注册和处理订阅请求，客户端必须执行以下序列：

1. 初始化连接 (`connection_init`)
2. 连接确认 (`connection_ack`)
3. 订阅注册 (`start`)
4. 订阅确认 (`start_ack`)
5. 处理订阅 (`data`)
6. 订阅取消注册 (`stop`)

连接启动消息

在成功握手后，客户端必须发送 `connection_init` 消息，才能开始与 AWS AppSync 实时终端节点进行通信。如果没有执行该步骤，将忽略所有其他消息。该消息是通过将下列 JSON 对象字符串化获得的字符串，如下所示：

```
{ "type": "connection_init" }
```

连接确认消息

发送 `connection_init` 消息后，客户端必须等待 `connection_ack` 消息。将忽略在收到 `connection_ack` 之前发送的所有消息。该消息具体如下：

```
{
```

```
  "type": "connection_ack",
  "payload": {
    // Time in milliseconds waiting for ka message before the client should terminate
 the WebSocket connection
    "connectionTimeoutMs": 300000
  }
}
```

保持活动消息

除了连接确认消息以外，客户端还会定期收到保持活动消息。如果客户端在连接超时期限内没有收到保持活动消息，客户端应关闭连接。AWSAppSync 不断发送这些消息并为注册的订阅提供服务，直到它自动关闭连接（24 小时后）。保持活动消息是检测信号，客户端不需要确认这些消息。

```
{ "type": "ka" }
```

订阅注册消息

在客户端收到 connection_ack 消息后，客户端可以向 AWS AppSync 发送订阅注册消息。这种类型的消息是字符串化 JSON 对象，其中包含以下字段：

- `"id": <string>`：订阅的 ID。对于每个订阅，该 ID 必须是唯一的，否则，服务器返回错误，以指示订阅 ID 是重复的。
- `"type": "start"`：常量 `<string>` 参数。
- `"payload": <Object>`：包含与订阅相关的信息的对象。
  - `"data": <string>`：包含 GraphQL 查询和变量的字符串化 JSON 对象。
    - `"query": <string>`：GraphQL 操作。
    - `"variables": <Object>`：包含查询变量的对象。
  - `"extensions": <Object>`：包含授权对象的对象。
- `"authorization": <Object>`：包含授权所需的字段的对象。

订阅注册的授权对象

基于 AWS AppSync API 授权模式的标头参数格式一节中的相同规则适用于授权对象。唯一的例外是 IAM，其中的 SigV4 签名信息略有不同。有关详细信息，请参阅 IAM 示例。

使用 Amazon Cognito 用户池的示例：

```
{
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n
 __typename\\n message\\n }\\n }\",\"variables\":{}}",
      "extensions": {
        "authorization": {
          "Authorization":
 "eyEXAMPLEiJjbG5xb3A5eW5MK09QYXIrMTJEXAMPLEBieU5WNHhsQjhPVW9YMnM2WldvPSIsImFsZyI6IlEXAMPLEn0.e
qTCtrYeboUJ4luRSTPXaNewNeEXAMPLE14C6sfg05tO0fOMpiUwj9k19gtNCCMqoSsjtQoUweFnH4JYa5EXAMPLEVxOyQEQ
RWvW7yQU3sNQRLEXAMPLEcd0yufBiCYs3dfQxTTdvR1B6Wz6CD78lfNeKqfzzUn2beMoup2h6EXAMPLE4ow8cUPUPvG0DzR
          "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com"
         }
      }
  },
  "type": "start"
}
```

使用 IAM 的示例：

```
{
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n
 __typename\\n message\\n }\\n }\",\"variables\":{}}",
    "extensions": {
      "authorization": {
        "accept": "application/json, text/javascript",
        "content-type": "application/json; charset=UTF-8",
        "X-Amz-Security-Token":
 "AgEXAMPLEZ2luX2VjEAoaDmFwLXNvdXRoZWFEXAMPLEcwRQIgAh97Cljq7wOPL8KsxP3YtDuyc/9hAj8PhJ7Fvf38SgoC
+
+pEagWCveZUjKEn0zyUhBEXAMPLEjj/////////8BEXAMPLExODk2NDgyNzg1NSIMo1mWnpESWUoYw4BkKqEFSrm3DXuL8
+ZbVc4JKjDP4vUCKNR6Le9C9pZp9PsW0NoFy3vLBUdAXEXAMPLEOVG8feXfiEEA+1khgFK/
wEtwR+9zF7NaMMMse07wN2gG2tH0eKMEXAMPLEQX+sMbytQo8iepP9PZOzlZsSFb/
dP5Q8hk6YEXAMPLEYcKZsTkDAq2uKFQ8mYUVA9EtQnNRiFLEY83aKvG/tqLWNnGlSNVx7SMcfovkFDqQamm
+88y1OwwAEYK7qcoceX6Z7GGcaYuIfGpaX2MCCELeQvZ+8WxEgOnIfz7GYvsYNjLZSaRnV4G
+ILY1F0QNW64S9Nvj
+BwDg3ht2CrNvpwjVYlj9U3nmxE0UG5ne83LL5hhqMpm25kmL7enVgw2kQzmU2id4IKu0C/
WaoDRuO2F5zE63vJbxN8AYs7338+4B4HBb6BZ6OUgg96Q15RA41/
gIqxaVPxyTpDfTU5GfSLxocdYeniqqpFMtZG2n9d0u7GsQNcFkNcG3qDZm4tDo8tZbuym0a2VcF2E5hFEgXBa
+XLJCfXi/77OqAEjP0x7Qdk3B43p8KG/BaioP5RsV8zBGvH1zAgyPha2rN70/
tT13yrmPd5QYEfwzexjKrV4mWIuRg8NTHYSZJUaeyCwTom80VFUJXG
```

```
+GYTUyv5W22aBcnoRGiCiKEYTLOkgXecdKFTHmcIAejQ9Welr0a196Kq87w5KNMCkcCGFnwBNFLmfnbpNqT6rUBxxs3X5nt
aox0FtHX21eF6qIGT8j1z+l2opU+ggwUgkhUUgCH2TfqBj+MLMVVvpgqJsPKt582caFKArIFIvO
+9QupxLnEH2hz04TMTfnU6bQC6z1buVe7h
+tOLnh1YPFsLQ88anib/7TTC8k9DsBTq0ASe8R2GbSEsmO9qbbMwgEaYUhOKtGeyQsSJdhSk6XxXThrWL9EnwBCXDkICMqd
+WgtPtKOOweDlCaRs3R2qXcbNgVhleMk4IWnF8D1695AenU1LwHjOJLkCjxgNFiWAFEPH9aEXAMPLExA==",
        "Authorization": "AWS4-HMAC-SHA256 Credential=XXXXXXXXXXXXXXXXXXX/20200401/
us-east-1/appsync/aws4_request, SignedHeaders=accept;content-
encoding;content-type;host;x-amz-date;x-amz-security-token,
 Signature=b90131a61a7c4318e1c35ead5dbfdeb46339a7585bbdbeceeaff51f4022eb1fd",
        "content-encoding": "amz-1.0",
        "host": "example1234567890000.appsync-api.us-east-1.amazonaws.com",
        "x-amz-date": "20200401T001010Z"
      }
    }
  },
  "type": "start"
}
```

使用自定义域名的示例：

```
{
  "id": "key-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n
 __typename\\n message\\n }\\n }\",\"variables\":{}}",
      "extensions": {
        "authorization": {
          "x-api-key": "da2-12345678901234567890123456",
          "host": "api.example.com"
         }
      }
  },
  "type": "start"
}
```

SigV4 签名不需要将 /connect 附加到 URL 中，并使用 JSON 字符串化 GraphQL 操作替换 data。以下是一个 SigV4 签名请求示例：

```
{
  url: "https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql",
  data: "{\"query\":\"subscription onCreateMessage {\\n onCreateMessage {\\n __typename
\\n message\\n }\\n }\",\"variables\":{}}",
  method: "POST",
```

```
  headers: {
    "accept": "application/json, text/javascript",
    "content-encoding": "amz-1.0",
    "content-type": "application/json; charset=UTF-8",
  }
}
```

订阅确认消息

在发送订阅开始消息后，客户端应等待 AWS AppSync 发送 `start_ack` 消息。`start_ack` 消息表示订阅成功。

订阅确认示例：

```
{
  "type": "start_ack",
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

错误消息

如果连接启动或订阅注册失败，或者从服务器中终止订阅，服务器将向客户端发送错误消息：

- `"type": "error"`：常量 <string> 参数。
- `"id": <string>`：注册的相应订阅的 ID（如果相关）。
- `"payload" <Object>`：包含相应错误信息的对象。

示例：

```
{
  "type": "error",
  "payload": {
    "errors": [
      {
        "errorType": "LimitExceededError",
        "message": "Rate limit exceeded"
      }
    ]
  }
}
```

处理数据消息

在客户端提交变更时，AWS AppSync 确定对该变更感兴趣的所有订阅者，并使用 "start" 订阅操作中的相应订阅 id 向每个订阅者发送 "type":"data" 消息。客户端应跟踪它发送的订阅 id，这样，在它收到数据消息时，客户端可以将其与相应的订阅进行匹配。

- "type": "data"：常量 <string> 参数。
- "id": <string>：注册的相应订阅的 ID。
- "payload" <Object>：包含订阅信息的对象。

示例：

```
{
  "type": "data",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": {
      "onCreateMessage": {
        "__typename": "Message",
        "message": "test"
      }
    }
  }
}
```

订阅取消注册消息

在应用程序希望停止侦听订阅事件时，客户端应发送包含以下字符串化 JSON 对象的消息：

- "type": "stop"：常量 <string> 参数。
- "id": <string>：要取消注册的订阅的 ID。

示例：

```
{
  "type":"stop",
  "id":"ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

AWS AppSync 发回一条确认消息，其中包含以下字符串化 JSON 对象：

- `"type": "complete"`：常量 `<string>` 参数。
- `"id": <string>`：取消注册的订阅的 ID。

在客户端收到确认消息后，它不会接收该特定订阅的更多消息。

示例：

```
{
  "type":"complete",
  "id":"eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

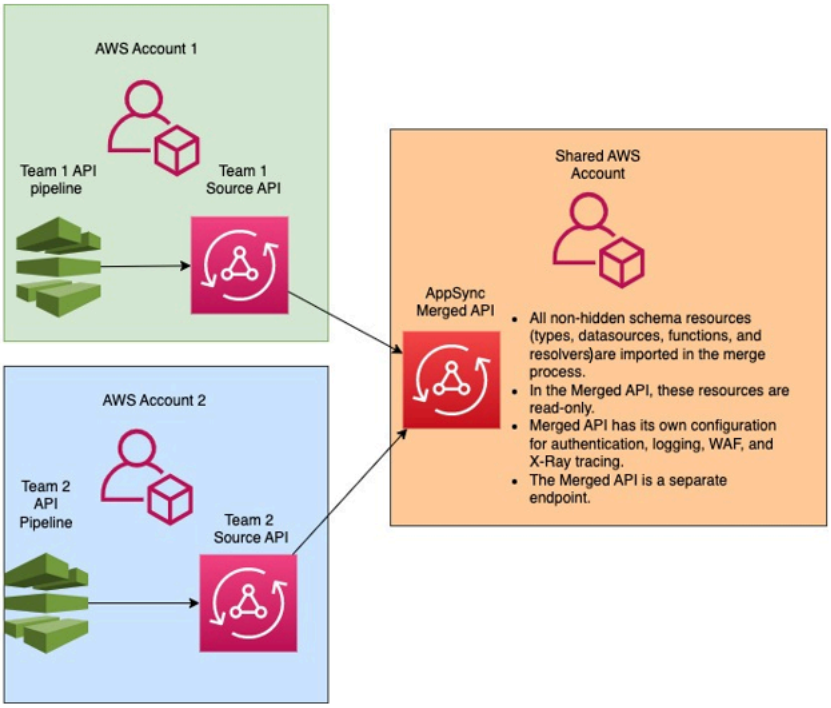
断开 WebSocket 连接

在断开连接之前，客户端应具有所需的逻辑以检查当前没有通过 WebSocket 连接执行任何操作，以避免数据丢失。在断开与 WebSocket 的连接之前，应先取消注册所有订阅。

合并的 API

随着在组织中越来越多地使用 GraphQL，可能需要在 API 易用性和 API 开发速度之间进行权衡。一方面，组织采用 AWS AppSync 和 GraphQL 为开发人员提供灵活的 API，他们可以通过单个网络调用安全地访问、处理和合并来自一个或多个数据域的数据，从而简化应用程序开发。另一方面，组织中负责将不同数据域合并为单个 GraphQL API 终端节点的团队可能希望能够相互独立地创建、管理和部署 API 更新，从而提高开发速度。

为了消除这种紧张关系，AWS AppSync 合并 API 功能允许来自不同数据域的团队独立创建和部署 AWS AppSync API（例如 GraphQL 架构、解析器、数据源和函数），然后将它们合并为一个合并的 API。这使组织能够维护简单易用的跨域 API，并为对该 API 做出贡献的不同团队提供一种方法，以快速独立地进行 API 更新。



通过使用合并的 API，组织可以将多个独立源 AWS AppSync API 的资源导入到单个 AWS AppSync 合并 API 终端节点中。为此，AWS AppSync 允许您创建一个 AWS AppSync 源 API 列表，然后将与源 API 关联的所有元数据（包括架构、类型、数据源、解析器和函数）合并为一个新的 AWS AppSync 合并 API。

在合并过程中，由于源 API 数据内容不一致（例如，在合并多个架构时发生的类型命名冲突），可能会发生合并冲突。对于源 API 中的定义不发生冲突的简单使用案例，无需修改源 API 架构。生成的合并 API 直接从原始源 AWS AppSync API 导入所有类型、解析器、数据源和函数。对于发生冲突的复杂使用案例，用户/团队必须通过各种方法解决冲突。AWS AppSync 为用户提供了一些可以减少合并冲突的工具和示例。

AWS AppSync 中配置的后续合并将源 API 中所做的更改传播到关联的合并 API。

合并的 API 和联合

在 GraphQL 社区中提供很多合并 GraphQL 架构的解决方案和模式，并通过共享图实现团队协作。AWS AppSync合并的 API 采用构建时方法进行架构组合，其中源 API 合并为一个单独的合并 API。另一种方法是在多个源 API 或子图之间添加运行时路由器。在这种方法中，路由器接收请求，引用它作为元数据维护的合并架构，构建请求计划，然后在其底层子图/服务器之间分配请求元素。下表将 AWS AppSync 合并 API 构建时方法与基于路由器的运行时 GraphQL 架构组合方法进行了比较：

Feature	AppSync Merged API	Router-based solutions
Sub-graphs managed independently	Yes	Yes
Sub-graphs addressable independently	Yes	Yes
Automated schema compositi on	Yes	Yes
Automated conflict detection	Yes	Yes
Conflict resolution via schema directives	Yes	Yes
Supported sub-graph servers	AWS AppSync*	Varies
Network complexity	Single, merged API means no extra network hops.	Multi-layer architecture requires query planning and delegation, sub-query parsing and serialization/deserializati on, and reference resolvers in sub-graphs to perform joins.
Observability support	Built-in monitoring, logging, and tracing. A single, Merged API server means simplified debugging.	Build-your-own observability across router and all associate d sub-graph servers. Complex debugging across distributed system.
Authorization support	Built in support for multiple authorization modes.	Build-your-own authorization rules.
Cross account security	Built-in support for cross-AWS cloud account associations.	Build-your-own security model.
Subscriptions support	Yes	No

* AWS AppSync 合并 API 只能与 AWS AppSync 源 API 相关联。如果您需要支持跨 AWS AppSync 和非 AWS AppSync 子图的架构组合，您可以将一个或多个 AWS AppSync GraphQL 和/或合并 API 连接到基于路由器的解决方案。例如，请参阅以下参考博客，以了解如何将基于路由器的架构与 Apollo Federation v2 一起使用以将 AWS AppSync API 添加为子图：Apollo GraphQL Federation with AWS AppSync。

主题

解决合并的 API 冲突

如果发生合并冲突，AWS AppSync 为用户提供了一些工具和示例以帮助解决问题。

合并的 API 架构指令

AWS AppSync 引入了一些 GraphQL 指令，可用于减少或解决源 API 之间的冲突：

- @canonical：该指令设置具有类似名称和数据的类型/字段的优先级。如果两个或更多源 API 具有相同的 GraphQL 类型或字段，其中的一个 API 可以将其类型或字段注释为 canonical，将在合并过程中优先考虑该 API。在合并时，忽略其他源 API 中未使用该指令注释的冲突类型/字段。
- @hidden：该指令封装某些类型/字段以将其从合并过程中删除。团队可能希望删除或隐藏源 API 中的特定类型或操作，以便仅内部客户端可以访问特定类型的数据。在附加该指令后，类型或字段不会合并到合并的 API 中。
- @renamed：该指令更改类型/字段名称以减少命名冲突。在某些情况下，不同的 API 具有相同的类型或字段名称。不过，需要在合并的架构中提供所有这些 API。要将它们全部包含在合并的 API 中，一个简单方法是将字段重命名为类似但不同的名称。

要显示架构指令提供的功能，请考虑以下示例：

在该示例中，假设我们要合并两个源 API。我们有两个创建和检索文章（例如，评论部分或社交媒体文章）的架构。假设这些类型和字段非常相似，在合并操作期间很可能会发生冲突。下面的代码片段显示每个架构的类型和字段。

第一个文件名为 Source1.graphql，它是一个 GraphQL 架构，允许用户使用 putPost 变更创建 Posts。每个 Post 包含一个标题和 ID。ID 用于引用 User 或发布者信息（电子邮件和地址）以及 Message 或负载（内容）。User 类型使用 @canonical 标签进行注释。

```
# This snippet represents a file called Source1.graphql

type Mutation {
    putPost(id: ID!, title: String!): Post
}

type Post {
    id: ID!
    title: String!
}

type Message {
   id: ID!
   content: String
}

type User @canonical {
   id: ID!
   email: String!
   address: String!
}

type Query {
    singlePost(id: ID!): Post
    getMessage(id: ID!): Message
}
```

第二个文件名为 Source2.graphql，它是一个 GraphQL 架构，其功能与 Source1.graphql 非常相似。但请注意，每种类型的字段是不同的。在合并这两个架构时，由于这些差异，将会发生合并冲突。

另请注意，Source2.graphql 还包含多个指令以减少这些冲突。Post 类型使用 @hidden 标签进行注释，以在合并操作期间对其自身进行模糊处理。Message 类型使用 @renamed 标签进行注释，以便在与另一个 Message 类型发生命名冲突时将类型名称修改为 ChatMessage。

```
# This snippet represents a file called Source2.graphql

type Post @hidden  {
    id: ID!
    title: String!
    internalSecret: String!
}

type Message @renamed(to: "ChatMessage") {
   id: ID!
   chatId: ID!
   from: User!
   to: User!
}

# Stub user so that we can link the canonical definition from Source1
type User {
   id: ID!
}

type Query {
    getPost(id: ID!): Post
    getMessage(id: ID!): Message @renamed(to: "getChatMessage")
}
```

在发生合并时，结果将生成 MergedSchema.graphql 文件：

```
# This snippet represents a file called MergedSchema.graphql

type Mutation {
    putPost(id: ID!, title: String!): Post
}

# Post from Source2 was hidden so only uses the Source1 definition.
type Post {
    id: ID!
    title: String!
}
```

```
# Renamed from Message to resolve the conflict
type ChatMessage {
   id: ID!
   chatId: ID!
   from: User!
   to: User!
}

type Message {
   id: ID!
   content: String
}

# Canonical definition from Source1
type User {
   id: ID!
   email: String!
   address: String!
}

type Query {
    singlePost(id: ID!): Post
    getMessage(id: ID!): Message

    # Renamed from getMessage
    getChatMessage(id: ID!): ChatMessage
}
```

在合并过程中发生了以下情况：

- 由于 @canonical 注释，Source1.graphql 中的 `User` 类型优先于 Source2.graphql 中的 `User` 类型。
- Source1.graphql 中的 `Message` 类型包含在合并中。不过，Source2.graphql 中的 `Message` 存在命名冲突。由于它具有 @renamed 注释，它也包含在合并中，但具有替代名称 `ChatMessage`。
- 将包含 Source1.graphql 中的 `Post` 类型，但不包含 Source2.graphql 中的 `Post` 类型。通常，该类型将会发生冲突，但由于 Source2.graphql 中的 `Post` 类型具有 @hidden 注释，将对其数据进行模糊处理而不会包含在合并中。这不会导致任何冲突。
- 更新了 `Query` 类型以包含两个文件中的内容。不过，由于该指令，一个 `GetMessage` 查询被命名为 `GetChatMessage`。这解决了两个同名查询之间的命名冲突。

还有一种情况是，不会将任何指令添加到冲突的类型中。此处，合并的类型包括该类型的所有源定义中的所有字段的联合。例如，请考虑以下示例：

该架构名为 Source1.graphql，允许创建和检索 Posts。配置与上一示例类似，但信息较少。

```
# This snippet represents a file called Source1.graphql

type Mutation {
    putPost(id: ID!, title: String!): Post
}

type Post  {
    id: ID!
    title: String!
}

type Query {
    getPost(id: ID!): Post
}
```

该架构名为 Source2.graphql，允许创建和检索 Reviews（例如，电影评级或餐厅评价）。Reviews 与相同 ID 值的 Post 相关联。它们放在一起以提供完整评价文章的标题、文章 ID 和负载消息。

在合并时，将在两种 Post 类型之间发生冲突。由于没有可以解决该问题的注释，因此，默认行为是对冲突类型执行联合操作。

```
# This snippet represents a file called Source2.graphql

type Mutation {
    putReview(id: ID!, postId: ID!, comment: String!): Review
}

type Post  {
    id: ID!
    reviews: [Review]
}

type Review {
   id: ID!
   postId: ID!
   comment: String!
}
```

```
type Query {
    getReview(id: ID!): Review
}
```

在发生合并时，结果将生成 MergedSchema.graphql 文件：

```
# This snippet represents a file called MergedSchema.graphql

type Mutation {
    putReview(id: ID!, postId: ID!, comment: String!): Review
    putPost(id: ID!, title: String!): Post
}

type Post  {
    id: ID!
    title: String!
    reviews: [Review]
}

type Review {
   id: ID!
   postId: ID!
   comment: String!
}

type Query {
    getPost(id: ID!): Post
    getReview(id: ID!): Review
}
```

在合并过程中发生了以下情况：

- Mutation 类型没有遇到冲突并进行了合并。
- Post 类型字段通过联合操作进行合并。请注意两者之间的联合如何生成一个 id、title 和 reviews。
- Review 类型没有遇到冲突并进行了合并。
- Query 类型没有遇到冲突并进行了合并。

管理共享类型上的解析器

在上面的示例中，考虑 Source1.graphql 在 `Query.getPost` 上配置了单位解析器的情况，该解析器使用名为 `PostDatasource` 的 DynamoDB 数据源。该解析器将返回 `Post` 类型的 `id` 和 `title`。现在，考虑 Source2.graphql 在 `Post.reviews` 上配置了管道解析器的情况，该解析器运行两个函数。`Function1` 附加了一个 None 数据源以执行自定义授权检查。`Function2` 附加了一个 DynamoDB 数据源以查询 `reviews` 表。

```
query GetPostQuery {
    getPost(id: "1") {
        id,
        title,
        reviews
    }
}
```

在客户端对合并的 API 终端节点运行上述查询时，AWS AppSync 服务先从 `Source1` 中运行 `Query.getPost` 的单位解析器，该解析器调用 `PostDatasource` 并从 DynamoDB 中返回数据。然后，它运行 `Post.reviews` 管道解析器，其中 `Function1` 执行自定义授权逻辑，并且 `Function2` 返回位于 `$context.source` 中的给定 `id` 的评价。该服务将请求作为单个 GraphQL 运行进行处理，并且该简单请求仅需要一个请求令牌。

管理共享类型上的解析器冲突

考虑以下情况，我们还在 `Query.getPost` 上实施了一个解析器，以便每次在 `Source2` 中的字段解析器以外提供多个字段。Source1.graphql 可能如下所示：

```
# This snippet represents a file called Source1.graphql

type Post  {
    id: ID!
    title: String!
    date: AWSDateTime!
}

type Query {
    getPost(id: ID!): Post
}
```

Source2.graphql 可能如下所示：

```
# This snippet represents a file called Source2.graphql

type Post  {
  id: ID!
  content: String!
  contentHash: String!
  author: String!
}

type Query {
    getPost(id: ID!): Post
}
```

尝试合并这两个架构将发生合并错误，因为 AWS AppSync 合并 API 不允许将多个源解析器附加到同一字段。为了解决该冲突，您可以实施一种字段解析器模式，该模式要求 Source2.graphql 添加一个单独的类型以定义它从 `Post` 类型中拥有的字段。在以下示例中，我们添加一个名为 `PostInfo` 的类型，其中包含由 Source2.graphql 解析的 content 和 author 字段。Source1.graphql 实施附加到 `Query.getPost` 的解析器，而 Source2.graphql 现在将一个解析器附加到 `Post.postInfo`，以确保可以成功检索所有数据：

```
type Post  {
  id: ID!
  postInfo: PostInfo
}

type PostInfo {
   content: String!
   contentHash: String!
   author: String!
}

type Query {
    getPost(id: ID!): Post
}
```

虽然解决此类冲突需要重新编写源 API 架构，并且客户端可能需要更改其查询，但这种方法的优点是，合并解析器的所有权在源团队中仍然是清晰的。

配置架构

双方负责配置架构以创建合并的 API：

- 合并的 API 所有者 - 合并的 API 所有者必须配置合并 API 的授权逻辑和高级设置，例如日志记录、跟踪、缓存和 WAF 支持。
- 关联的源 API 所有者 - 关联的 API 所有者必须配置构成合并 API 的架构、解析器和数据源。

由于合并 API 的架构是根据关联源 API 的架构创建的，因此，它是只读的。这意味着必须在源 API 中启动对架构的更改。在 AWS AppSync 控制台中，您可以使用架构窗口上面的下拉列表，在合并架构与合并 API 中包含的源 API 的各个架构之间进行切换。

配置授权模式

可以使用多种授权模式保护您的合并 API。要了解 AWS AppSync 中的授权模式的更多信息，请参阅授权和身份验证。

可以将以下授权模式与合并的 API 一起使用：

- API 密钥：最简单的授权策略。所有请求必须在 `x-api-key` 请求标头下面包含 API 密钥。过期的 API 密钥在过期日期之后保留 60 天。
- AWS Identity and Access Management (IAM)：AWS IAM 授权策略授权所有经过 SigV4 签名的请求。
- Amazon Cognito 用户池：通过 Amazon Cognito 用户池授权您的用户以实现更精细的控制。
- AWS Lambda 授权者：一种无服务器函数，允许您使用自定义逻辑对 AWS AppSync API 进行身份验证和授权访问。
- OpenID Connect：该授权类型强制实施 OIDC 兼容服务提供的 OpenID Connect (OIDC) 令牌。您的应用程序可以利用由 OIDC 提供程序定义的用户和权限来控制访问。

合并的 API 的授权模式是由合并的 API 所有者配置的。在执行合并操作时，合并的 API 必须包含在源 API 上配置的主要授权模式，以作为自己的主要授权模式或辅助授权模式。否则，将会不兼容，并且合并操作由于冲突而失败。在源 API 中使用多重授权指令时，合并过程能够自动将这些指令合并到统一的终端节点中。如果源 API 的主要授权模式与合并 API 的主要授权模式不匹配，它自动添加这些授权指令，以确保源 API 中的类型的授权模式一致。

配置执行角色

在创建合并的 API 时，您需要定义服务角色。AWS 服务角色是一个 AWS Identity and Access Management (IAM) 角色，AWS 服务使用该角色代表您执行任务。

在这种情况下，您的合并 API 需要运行解析器以访问源 API 中配置的数据源中的数据。这种情况所需的服务角色是 `mergedApiExecutionRole`，它必须具有明确的访问权限，以通过 `appsync:SourceGraphQL` IAM 权限对合并 API 中包含的源 API 运行请求。在 GraphQL 请求运行期间，AWS AppSync 服务担任该服务角色，并授权该角色执行 `appsync:SourceGraphQL` 操作。

AWS AppSync 支持为请求中的特定顶级字段允许或拒绝该权限，例如，如何将 IAM 授权模式用于 IAM API。对于非顶级字段，AWS AppSync 要求您定义源 API ARN 本身的权限。为了限制对合并 API 中的特定非顶级字段的访问，我们建议在 Lambda 中实施自定义逻辑，或使用 @hidden 指令从合并的 API 中隐藏源 API 字段。如果要允许角色在源 API 中执行所有数据操作，您可以添加以下策略。请注意，第一个资源条目允许访问所有顶级字段，第二个条目涵盖对源 API 资源本身进行授权的子解析器：

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [ "appsync:SourceGraphQL"],
        "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId/*",
            "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId"]
    }]
}
```

如果要仅限访问特定的顶级字段，您可以使用如下策略：

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [ "appsync:SourceGraphQL"],
        "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId/types/
Query/fields/<Field-1>",
            "arn:aws:appsync:us-west-2:123456789012:apis/YourSourceGraphQLApiId"]
    }]
}
```

您也可以使用 AWS AppSync 控制台 API 创建向导生成服务角色，以允许您的合并 API 访问源 API 中配置的资源，这些源 API 与合并的 API 位于同一账户中。如果您的源 API 没有位于与合并 API 相同的账户中，您必须先使用 AWS Resource Access Manager (AWS RAM) 共享您的资源。

使用 AWS RAM 配置跨账户的合并 API

在创建合并的 API 时，您可以选择关联已通过 AWS Resource Access Manager (AWS RAM) 共享的其他账户中的源 API。AWS RAM 帮助您在 AWS 账户之间、在您的组织或组织单位 (OU) 内以及与 IAM 角色和用户安全地共享资源。

AWS AppSync 与 AWS RAM 集成在一起，以支持从单个合并的 API 中配置和访问多个账户中的源 API。AWS RAM 允许您创建资源共享，即资源容器以及为每个资源共享的权限集。您可以将 AWS AppSync API 添加到 AWS RAM 中的资源共享。在资源共享中，AWS AppSync 提供了三种不同的权限集，它们可以与 RAM 中的 AWS AppSync API 相关联：

1. `AWSRAMPermissionAppSyncSourceApiOperationAccess`：默认权限集；在 AWS RAM 中共享 AWS AppSync API 时，如果未指定其他权限，将添加该权限集。该权限集用于与合并的 API 所有者共享源 AWS AppSync API。该权限集包括源 API 的 `appsync:AssociateMergedGraphqlApi` 权限以及在运行时访问源 API 资源所需的 `appsync:SourceGraphQL` 权限。
2. `AWSRAMPermissionAppSyncMergedApiOperationAccess`：在与源 API 所有者共享合并的 API 时，应配置该权限集。该权限集使源 API 能够配置合并的 API，包括能够将目标主体拥有的任何源 API 与合并的 API 关联，以及读取和更新合并 API 的源 API 关联。
3. `AWSRAMPermissionAppSyncAllowSourceGraphQLAccess`：该权限集允许将 `appsync:SourceGraphQL` 权限与 AWS AppSync API 一起使用。它旨在用于与合并的 API 所有者共享源 API。与源 API 操作访问权限的默认权限集相反，该权限集仅包括运行时权限 `appsync:SourceGraphQL`。如果用户选择与源 API 所有者共享合并的 API 操作访问权限，他们还需要从源 API 中将该权限与合并的 API 所有者共享，以便通过合并的 API 终端节点进行运行时访问。

AWS AppSync 还支持客户托管权限。在提供的 AWS 托管权限之一不起作用时，您可以创建自己的客户托管权限。客户托管权限是您编写和维护的托管权限，您精确指定可以在哪些条件下对使用 AWS RAM 共享的资源执行哪些操作。AWS AppSync 允许您在创建自己的权限时选择以下操作：

1. `appsync:AssociateSourceGraphqlApi`
2. `appsync:AssociateMergedGraphqlApi`
3. `appsync:GetSourceApiAssociation`
4. `appsync:UpdateSourceApiAssociation`
5. `appsync:StartSchemaMerge`
6. `appsync:ListTypesByAssociation`

7. appsync:SourceGraphQL

在 AWS RAM 中正确共享源 API 或合并的 API 并接受资源共享邀请（如有必要）后，在为合并的 API 创建或更新源 API 关联时，将在 AWS AppSync 控制台中显示该信息。您也可以调用 AWS AppSync 提供的 ListGraphqlApis 操作并使用 OTHER_ACCOUNTS 所有者筛选条件，以列出已使用 AWS RAM 与您的账户共享的所有 AWS AppSync API，而无论权限集如何。

> **Note**
>
> 要通过 AWS RAM 进行共享，AWS RAM 中的调用方需要有权对共享的任何 API 执行 appsync:PutResourcePolicy 操作。

合并

管理合并

合并的 API 旨在支持统一 AWS AppSync 终端节点上的团队协作。团队可以在后端独立开发自己的隔离源 GraphQL API，而 AWS AppSync 服务管理将资源集成到单个合并 API 终端节点的过程，以减少协作中的摩擦并缩短开发前期时间。

自动合并

可以配置与您的 AWS AppSync 合并 API 关联的源 API，以在对源 API 进行任何更改后自动合并到合并的 API 中。这会确保源 API 中的更改始终在后台传播到合并的 API 终端节点。将在合并的 API 中更新源 API 架构中的任何更改，只要它不会与合并 API 中的现有定义发生合并冲突。如果源 API 中的更新将更新解析器、数据源或函数，则也会更新导入的资源。在引入无法自动解决的新冲突时，将拒绝合并的 API 架构更新，因为在合并操作期间发生不支持的冲突。对于状态为 MERGE_FAILED 的每个源 API 关联，将在控制台中显示错误消息。您也可以使用 AWS SDK 或 AWS CLI 为给定源 API 关联调用 GetSourceApiAssociation 操作以检查错误消息，如下所示：

```
aws appsync get-source-api-association --merged-api-identifier <Merged API ARN> --
association-id <SourceApiAssociation id>
```

这会生成以下格式的结果：

```
{
    "sourceApiAssociation": {
        "associationId": "<association id>",
```

```
        "associationArn": "<association arn>",
        "sourceApiId": "<source api id>",
        "sourceApiArn": "<source api arn>",
        "mergedApiArn": "<merged api arn>",
        "mergedApiId": "<merged api id>",
        "sourceApiAssociationConfig": {
            "mergeType": "MANUAL_MERGE"
        },
        "sourceApiAssociationStatus": "MERGE_FAILED",
        "sourceApiAssociationStatusDetail": "Unable to resolve conflict on object with
 name title: Merging is not supported for fields with different types."
    }
}
```

手动合并

源 API 的默认设置是手动合并。要合并自上次更新合并 API 以来源 API 中发生的任何更改，源 API 所有者可以从 AWS AppSync 控制台中或通过 AWS SDK 和 AWS CLI 中提供的 `StartSchemaMerge` 操作调用手动合并。

对合并 API 的额外支持

配置订阅

与基于路由器的 GraphQL 架构组合方法不同，AWS AppSync 合并 API 为 GraphQL 订阅提供内置支持。关联的源 API 中定义的所有订阅操作将在合并 API 中自动进行合并和运行，而无需进行修改。要了解 AWS AppSync 如何通过无服务器 WebSockets 连接支持订阅的更多信息，请参阅实时数据。

配置可观测性

AWS AppSync 合并 API 通过 Amazon CloudWatch 提供内置日志记录、监控和指标。AWS AppSync 还通过 AWS X-Ray 提供内置的跟踪支持。

配置自定义域

AWS AppSync 合并 API 提供内置支持，以将自定义域与合并 API 的 GraphQL 和实时终端节点一起使用。

配置缓存

AWS AppSync 合并 API 提供内置支持，可以选择缓存请求级和/或解析器级响应以及响应压缩。要了解更多信息，请参阅缓存和压缩。

配置私有 API

AWS AppSync 合并 API 为私有 API 提供内置支持，这些 API 仅限来自您可以配置的 VPC 终端节点的流量访问合并 API 的 GraphQL 和实时终端节点。

配置防火墙规则

AWS AppSync 合并 API 为 AWS WAF 提供内置支持，以使您能够定义 Web 应用程序防火墙规则以保护您的 API。

配置审核日志

AWS AppSync 合并 API 为 AWS CloudTrail 提供内置支持，以使您能够配置和管理审核日志。

合并的 API 限制

在开发合并 API 时，请注意以下规则：

1. 合并的 API 不能是另一个合并 API 的源 API。
2. 一个源 API 不能与多个合并的 API 相关联。
3. 合并的 API 架构文档的默认大小限制为 10 MB。
4. 可以与合并 API 关联的源 API 的默认数量为 10 个。但是，如果您的合并 API 中需要 10 个以上的源 API，则可以请求增加限制。

创建合并的 API

在控制台中创建合并的 API

1. 登录 AWS Management Console并打开 AWS AppSync 控制台。
   - 在控制面板中，选择创建 API。
2. 选择 Merged API，然后选择下一步。
3. 在指定 API 详细信息页面中，输入以下信息：
   a. 在 API 详细信息下面，输入以下信息：
      i. 指定合并 API 的 API 名称。该字段是一种标记 GraphQL API 的方法，以方便地将其与其他 GraphQL API 区分开。

ii. 指定联系信息。该字段是可选的，并将名称或组附加到 GraphQL API。它不会链接到其他资源或由其他资源生成，其工作方式与 API 名称字段非常相似。

b. 在服务角色下面，您必须将一个 IAM 执行角色附加到合并的 API，以便 AWS AppSync 在运行时安全地导入和使用您的资源。您可以选择创建并使用新的服务角色，以使您能够指定 AWS AppSync 将使用的策略和资源。您也可以选择使用现有的服务角色，然后从下拉列表中选择现有 IAM 角色以将其导入。

c. 在私有 API 配置下面，您可以选择启用私有 API 功能。请注意，在创建合并的 API 后，无法更改该选项。有关私有 API 的更多信息，请参阅使用 AWS AppSync 私有 API。

在完成后，选择下一步。

4. 接下来，您必须添加将作为合并 API 基础的 GraphQL API。在选择源 API 页面中，输入以下信息：

a. 在来自您的 AWS 账户的 API 表中，选择添加源 API。在 GraphQL API 列表中，每个条目包含以下数据：

i. 名称：GraphQL API 的 API 名称字段。

ii. API ID：GraphQL API 的唯一 ID 值。

iii. 主要授权模式：GraphQL API 的默认授权模式。有关 AWS AppSync 中的授权模式的更多信息，请参阅授权和身份验证。

iv. 额外的授权模式：在 GraphQL API 中配置的辅助授权模式。

v. 选择将在合并 API 中使用的 API，方法是选中该 API 的名称字段旁边的复选框。然后，选择添加源 API。选定的 GraphQL API 将显示在来自您的 AWS 账户的 API 表中。

b. 在来自其他 AWS 账户的 API 表中，选择添加源 API。该列表中的 GraphQL API 来自通过 AWS Resource Access Manager (AWS RAM) 与您共享资源的其他账户。在该表中选择 GraphQL API 的过程与上一节中的过程相同。有关通过 AWS RAM 共享资源的更多信息，请参阅 What is AWS Resource Access Manager?。

在完成后，选择下一步。

c. 添加您的主要授权模式。有关更多信息，请参阅授权和身份验证。选择下一步。

d. 检查您的输入，然后选择创建 API。

RDS 自省

AWS AppSync 使从现有关系数据库构建 API 变得容易。它的自省实用程序可以从数据库表中发现模型并建议 GraphQL 类型。AWS AppSync 控制台的“创建 API”向导可以立即从 Aurora MySQL 或 PostgreSQL 数据库生成 API。它会自动创建用于读取和写入数据的类型和 JavaScript 解析器。

AWS AppSync 通过 Amazon RDS 数据 API 提供与 Amazon Aurora 数据库的直接集成。Amazon RDS 数据 API 不要求永久数据库连接，而是提供了一个安全 HTTP 端点，AWS AppSync 连接到该端点以运行 SQL 语句。您可以利用这一点在 Aurora 上为您的 MySQL 和 PostgreSQL 工作负载创建关系数据库 API。

使用 AWS AppSync 为关系数据库构建 API 有几个优点：

- 您的数据库不会直接暴露给客户端，从而使接入点与数据库本身分离。
- 您可以根据不同应用程序的需求构建专门构建的 API，无需在前端使用自定义业务逻辑。这与服务于前端的后端 (BFF) 模式一致。
- 可以使用各种授权模式在 AWS AppSync 层上实现授权和访问控制，以控制访问权限。无需额外的计算资源即可连接到数据库，例如托管 Web 服务器或代理连接。
- 可以通过订阅来添加实时功能，通过 AppSync 进行的数据突变会自动推送到连接的客户端。
- 客户端可以使用常用端口（例如 443）通过 HTTPS 连接到 API。

AWS AppSync 使从现有关系数据库构建 API 变得容易。它的自省实用程序可以从数据库表中发现模型并建议 GraphQL 类型。AWS AppSync 控制台的创建 API 向导可以立即从 Aurora MySQL 或 PostgreSQL 数据库生成 API。它会自动创建用于读取和写入数据的类型和 JavaScript 解析器。

AWS AppSync 提供了集成的 JavaScript 实用程序，以简化在解析器中编写 SQL 语句。您可以将 AWS AppSync 的 `sql` 标签模板用于具有动态值的静态语句，也可以使用 `rds` 模块实用程序以编程方式构建语句。有关更多信息，请参阅 RDS 的解析器函数参考数据来源和内置模块。

使用自省特征（控制台）

有关详细教程和入门指南，请参阅教程：带数据 API 的 Aurora PostgreSQL Serverless。

AWS AppSync 控制台可让您在短短几分钟内从配置了数据 API 的现有 Aurora 数据库创建 AWS AppSync GraphQL API。这会根据您的数据库配置快速生成操作架构。您可以按原样使用 API，也可以在它的基础之上构建来添加特征。

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。

- 在控制面板中，选择创建 API。

2. 在 API 选项下，选择 GraphQL API 和以 Amazon Aurora 集群开始，然后选择下一步。

   a. 输入 API 名称。这将在控制台中用作 API 的标识符。

   b. 对于联系信息，您可以输入联系人以指定 API 的管理员。此为可选字段。

   c. 在私有 API 配置下面，您可以启用私有 API 功能。只能从配置的 VPC 终端节点 (VPCE) 中访问私有 API。有关更多信息，请参阅私有 API。

      对于该示例，我们不建议启用该功能。在检查您的输入后，选择下一步。

3. 在数据库页面中，选择选择数据库。

   a. 您需要从集群中选择数据库。第一步是选择您的集群所在的区域。

   b. 从下拉列表中选择 Aurora 集群。请注意，在使用资源之前，您必须已创建并启用相应的数据 API。

   c. 接下来，您必须将数据库的凭证添加到服务中。这主要是使用 AWS Secrets Manager 来完成的。选择您的密钥所在的区域。有关如何检索密钥信息的更多信息，请参阅查找密钥或检索密钥。

   d. 从下拉列表中添加您的密钥。请注意，用户必须对您的数据库具有读取权限。

4. 选择导入。

   AWS AppSync 将开始自省您的数据库，同时发现表、列、主键和索引。它会检查 GraphQL API 是否支持所发现的表。请注意，为了支持创建新行，表需要一个可以使用多列的主键。AWS AppSync 将表列映射到类型字段，如下所示：

数据类型	字段类型
VARCHAR	String
CHAR	String
BINARY	String
VARBINARY	String
TINYBLOB	String
TINYTEXT	String

TEXT	String
BLOB	String
MEDIUMTEXT	String
MEDIUMBLOB	String
LONGTEXT	String
LONGBLOB	String
BOOL	Boolean
BOOLEAN	Boolean
BIT	Int
TINYINT	Int
SMALLINT	Int
MEDIUMINT	Int
INT	Int
INTEGER	Int
BIGINT	Int
YEAR	Int
FLOAT	Float
DOUBLE	Float
DECIMAL	Float
DEC	Float
NUMERIC	Float
DATE	AWSDate

TIMESTAMP	String
DATETIME	String
TIME	AWSTime
JSON	AWSJson
ENUM	ENUM

5. 表发现完成后，数据库部分将填充您的信息。在新的数据库表部分中，表中的数据可能已经填充并转换为架构的类型。如果您没有看到某些必填数据，则可以通过以下方式进行检查：选择添加表，在出现的模式中单击这些类型的复选框，然后选择添加。

   要从数据库表部分删除类型，请单击要删除的类型旁边的复选框，然后选择删除。如果您想稍后再次添加，删除的类型将置于添加表模式中。

   请注意，AWS AppSync 使用表名称作为类型名称，但您可以对其进行重命名，例如，将诸如 *movies* 之类的复数表名更改为类型名称 *Movie*。要重命名数据库表部分中的类型，请单击要重命名的类型的复选框，然后单击类型名称列中的铅笔图标。

   要根据您的选择预览架构的内容，请选择预览架构。请注意，此架构不能为空，因此必须至少有一个表转换为类型。此外，此架构的大小不能超过 1MB。

   - 在服务角色下，选择是专门为此导入创建新的服务角色，还是使用现有的角色。
6. 选择下一步。
7. 接下来，选择是创建只读 API（仅限查询），还是创建用于读取和写入数据（包含查询和突变）的 API。后者还支持由突变触发的实时订阅。
8. 选择下一步。
9. 查看您的选择，然后选择创建 API。AWS AppSync 将创建 API 并将解析器附加到查询和突变。生成的 API 可完全运行，可以根据需要进行扩展。

使用自省特征 (API)

您可以使用 `StartDataSourceIntrospection` 自省 API 以编程方式发现数据库中的模型。有关该命令的更多详细信息，请参阅“使用 `StartDataSourceIntrospection` API”。

要使用 `StartDataSourceIntrospection`，请提供您的 Aurora 集群 Amazon 资源名称 (ARN)、数据库名称和 AWS Secrets Manager 密钥 ARN。该命令启动自省过程。您可以使用

GetDataSourceIntrospection 命令检索结果。您可以指定该命令是否应返回已发现模型的存储定义语言 (SDL) 字符串。这对于直接从已发现的模型生成 SDL 架构定义非常有用。

例如，如果您对简单 Todos 表使用以下数据定义语言 (DDL) 语句：

```
create table if not exists public.todos
(
id serial constraint todos_pk primary key,
description text,
due timestamp,
"createdAt" timestamp default now()
);
```

您从以下内容开始自省。

```
aws appsync start-data-source-introspection \
  --rds-data-api-config resourceArn=<cluster-arn>,secretArn=<secret-
arn>,databaseName=database
```

接下来，使用 GetDataSourceIntrospection 命令检索结果。

```
aws appsync get-data-source-introspection \
  --introspection-id a1234567-8910-abcd-efgh-identifier \
  --include-models-sdl
```

这会返回以下结果。

```
{
    "introspectionId": "a1234567-8910-abcd-efgh-identifier",
    "introspectionStatus": "SUCCESS",
    "introspectionStatusDetail": null,
    "introspectionResult": {
        "models": [
            {
                "name": "todos",
                "fields": [
                    {
                        "name": "description",
                        "type": {
                            "kind": "Scalar",
                            "name": "String",
                            "type": null,
```

```
                        "values": null
                    },
                    "length": 0
                },
                {
                    "name": "due",
                    "type": {
                        "kind": "Scalar",
                        "name": "AWSDateTime",
                        "type": null,
                        "values": null
                    },
                    "length": 0
                },
                {
                    "name": "id",
                    "type": {
                        "kind": "NonNull",
                        "name": null,
                        "type": {
                            "kind": "Scalar",
                            "name": "Int",
                            "type": null,
                            "values": null
                        },
                        "values": null
                    },
                    "length": 0
                },
                {
                    "name": "createdAt",
                    "type": {
                        "kind": "Scalar",
                        "name": "AWSDateTime",
                        "type": null,
                        "values": null
                    },
                    "length": 0
                }
            ],
            "primaryKey": {
                "name": "PRIMARY_KEY",
                "fields": [
                    "id"
```

```
                    ]
                },
                "indexes": [],
                "sdl": "type todos\n{\ndescription: String\n\ndue: AWSDateTime\n\nid:
 Int!\n\ncreatedAt: AW
SDateTime\n}\n"
            }
        ],
        "nextToken": null
    }
}
```

构建客户端应用程序

你可以使用任何 AWS AppSync GraphQL 客户端连接到你的 GraphQL API，但我们强烈建议使用 Amplify 客户端。Amplify 不仅可以为您的 GraphQL API 自动生成强类型客户端软件开发工具包，而且还支持客户端应用程序中的实时数据和增强的 GraphQL 查询功能。对于网络应用程序，Amplify 可以生成客户端。 JavaScript对于那些针对跨平台或移动环境的用户，Amplify 可以满足 Android、iOS 和 React Native 要求。要深入研究这些平台的客户端代码生成功能，请参阅 Amplify 文档。以下是开启 JavaScript React 应用程序之旅的指南：

> **Note**
>
> 在开始之前，您需要安装并配置 npm 和 Amazon CLI。如果您使用的是 Amplify v6 客户端，请遵循本指南。

开始使用：

1. 在本地计算机上，导航到您的项目的目录。使用以下命令安装 Amplify 库：

```
npm install aws-amplify
```

2. 下载您的配置文件并将其放在您的项目文件夹中。您的配置文件通常会包含一个定义了某些设置（终端节点、区域、授权模式等）的config变量。例如，它可能看起来像这样：

```
const config =  {
    API: {
        GraphQL: {
          endpoint: 'https://abcdefghijklmnopqrstuvwxyz.appsync-api.us-
west-2.amazonaws.com/graphql',
          region: 'us-west-2',
          defaultAuthMode: 'apiKey',
          apiKey: ''
        }
    }
};

export default config;
```

3. 在你的代码中，导入 Amplify 库和你的配置来设置 Amplify：

```
import { Amplify } from 'aws-amplify';
import config from './aws-exports.js';

Amplify.configure(config);
```

或者，使用 API 配置中的代码片段直接设置 Amplify：

```
import { Amplify } from 'aws-amplify';

Amplify.configure({
  API: {
    GraphQL: {
      endpoint: 'https://abcdefghijklmnopqrstuvwxyz.appsync-api.us-
west-2.amazonaws.com/graphql',
      region: 'us-west-2',
      defaultAuthMode: 'apiKey',
      apiKey: ''
    }
  }
});
```

4. 通过使用 Amplify 工具链，您可以选择根据您的架构自动生成操作，从而省去手动编写脚本的麻烦。在应用程序的根目录中，使用以下 CLI 命令：

```
npx @aws-amplify/cli codegen add --apiId <id goes here> --region <region goes here>
```

这将下载您的 API 架构，并在默认情况下将客户端帮助程序代码生成到该`src/graphql`文件夹中。每次部署 API 后，您可以重新运行以下命令来生成更新的 GraphQL 语句和类型：

```
npx @aws-amplify/cli codegen
```

5. 你现在可以生成适用于安卓、Swift、Flutter 和 JavaScript DataStore的模型。使用以下命令下载您的架构：

```
aws appsync get-introspection-schema --api-id <id goes here> --region <region goes
 here> --format SDL schema.graphql
```

然后，从应用程序的根目录运行以下命令：

```
npx @aws-amplify/cli codegen models \
--model-schema schema.graphql \
--target [android|ios|flutter|javascript|typescript] \
--output-dir ./
```

解析器教程 (JavaScript)

AWS AppSync 通过数据源和解析器转换 GraphQL 请求，并从 AWS 资源中获取信息。AWSAppSync 支持自动预置和连接到某些数据源类型。AWSAppSync 支持将 AWS Lambda、Amazon DynamoDB、关系数据库 (Amazon Aurora Serverless)、Amazon OpenSearch Service 和 HTTP 终端节点作为数据源。您可以将 GraphQL API 与现有的 AWS 资源一起使用，或者构建数据源和解析器。本节通过一系列教程演示了此过程，帮助您更好地理解详细的工作原理以及优化选项。

主题

教程：DynamoDB JavaScript 解析器

在本教程中，您将 Amazon DynamoDB 表导入到 AWS AppSync，并连接这些表以使用 JavaScript 管道解析器构建功能齐全的 GraphQL API，您可以在自己的应用程序中使用该 API。

您将使用 AWS AppSync 控制台预置 Amazon DynamoDB 资源，创建解析器并将其连接到您的数据源。您也可以通过 GraphQL 语句读取和写入 Amazon DynamoDB 数据库并订阅实时数据。

必须完成一些特定的步骤，才能将 GraphQL 语句转换为 Amazon DynamoDB 操作以及将响应转换回 GraphQL。本教程通过一些现实世界的场景和数据访问模式介绍了配置过程。

创建您的 GraphQL API

在 AWS AppSync 中创建 GraphQL API

1. 打开 AppSync 控制台，然后选择创建 API。
2. 选择从头开始设计，然后选择下一步。

3. 将您的 API 命名为 PostTutorialAPI，然后选择下一步。跳到检查页面，同时将其余选项设置为默认值，然后选择Create。

AWS AppSync 控制台将为您创建一个新的 GraphQL API。默认情况下，它使用 API 密钥身份验证方式。您可以根据本教程后面的说明，使用控制台设置 GraphQL API 的其余部分，并针对它运行查询。

定义基本文章 API

您现在具有 GraphQL API，您可以设置一个基本架构，以允许对文章数据执行基本创建、检索和删除操作。

将数据添加到您的架构

1. 在您的 API 中，选择架构选项卡。
2. 我们将创建一个架构，它定义 Post 类型和 addPost 操作以添加和获取 Post 对象。在架构窗格中，将内容替换为以下代码：

```
schema {
    query: Query
    mutation: Mutation
}

type Query {
    getPost(id: ID): Post
}

type Mutation {
    addPost(
        id: ID!
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}

type Post {
    id: ID!
    author: String
    title: String
    content: String
```

```
    url: String
    ups: Int!
    downs: Int!
    version: Int!
}
```

3. 选择 Save Schema (保存架构)。

设置您的 Amazon DynamoDB 表

AWS AppSync 控制台可以帮助预置在 Amazon DynamoDB 表中存储您自己的资源所需的 AWS 资源。在该步骤中，您创建一个 Amazon DynamoDB 表以存储您的文章。您还会设置我们稍后使用的二级索引。

创建您的 Amazon DynamoDB 表

1. 在架构页面上，选择创建资源。
2. 选择使用现有的类型，然后选择 `Post` 类型。
3. 在其他索引部分中，选择添加索引。
4. 将索引命名为 `author-index`。
5. 将 `Primary key` 设置为 `author`，并将 `Sort` 键设置为 `None`。
6. 禁用自动生成 GraphQL。在该示例中，我们将自行创建解析器。
7. 选择创建。

您现在具有一个名为 `PostTable` 的新数据源，您可以访问侧面选项卡中的数据源以查看该数据源。您使用该数据源将查询和变更链接到 Amazon DynamoDB 表。

设置 addPost 解析器 (Amazon DynamoDB PutItem)

现在 AWS AppSync 识别了 Amazon DynamoDB 表，您可以定义解析器以将其链接到各个查询和变更。您创建的第一个解析器使用 JavaScript 的 `addPost` 管道解析器，可用于在 Amazon DynamoDB 表中创建文章。管道解析器具有以下组件：

- GraphQL 架构中的位置，用于附加解析器。在本例中，您将设置 `createPost` 类型的 `Mutation` 字段的解析器。在调用方调用 `{ addPost(...){...} }` 变更时，将调用该解析器。
- 此解析器所用的数据源。在该示例中，您希望使用以前定义的 DynamoDB 数据源，因此，您可以在 `post-table-for-tutorial` DynamoDB 表中添加条目。

- 请求处理程序。请求处理程序是一个函数，用于处理来自调用方的传入请求，并将其转换为 AWS AppSync 指令以对 DynamoDB 执行。
- 响应处理程序。响应处理程序的任务是，处理来自 DynamoDB 的响应，并将其转换回 GraphQL 所需的内容。如果 DynamoDB 中的数据形态与 GraphQL 中的 `Post` 类型不同，此模板很有用。但在此例中它们的形态相同，所以只用于传递数据。

设置您的解析器

1. 在您的 API 中，选择架构选项卡。
2. 在解析器窗格中，找到 `Mutation` 类型下面的 `addPost` 字段，然后选择附加。
3. 选择您的数据源，然后选择创建。
4. 在代码编辑器中，将代码替换为以下代码片段：

```
import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 const item = { ...ctx.arguments, ups: 1, downs: 0, version: 1 }
 const key = { id: ctx.args.id ?? util.autoId() }
 return ddb.put({ key, item })
}

export function response(ctx) {
 return ctx.result
}
```

5. 选择保存。

> **Note**
>
> 在该代码中，您使用 DynamoDB 模块实用程序轻松创建 DynamoDB 请求。

AWS AppSync 附带一个名为 `util.autoId()` 的自动 ID 生成实用程序，用于为您的新文章生成 ID。如果您未指定 ID，该实用程序将自动为您生成 ID。

```
const key = { id: ctx.args.id ?? util.autoId() }
```

有关适用于 JavaScript 的实用程序的更多信息，请参阅解析器和函数的 JavaScript 运行时功能。

调用 API 以添加文章

解析器现已配置完毕，AWS AppSync 可以将传入的 addPost 变更转换为 Amazon DynamoDB PutItem 操作。现在，您可以运行一个变更，在表中添加内容。

运行操作

1. 在您的 API 中，选择查询选项卡。
2. 在查询窗格中，添加以下变更：

```
mutation addPost {
  addPost(
    id: 123,
    author: "AUTHORNAME"
    title: "Our first post!"
    content: "This is our first post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. 选择运行（橙色播放按钮），然后选择 addPost。新创建的文章的结果应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
```

```
      "downs": 0,
      "version": 1
    }
  }
}
```

以下解释说明了发生的情况：

1. AWS AppSync 收到 addPost 变更请求。
2. AWS AppSync 执行解析器的请求处理程序。ddb.put 函数创建一个 PutItem 请求，如下所示：

```
{
  operation: 'PutItem',
  key: { id: { S: '123' } },
  attributeValues: {
    downs: { N: 0 },
    author: { S: 'AUTHORNAME' },
    ups: { N: 1 },
    title: { S: 'Our first post!' },
    version: { N: 1 },
    content: { S: 'This is our first post.' },
    url: { S: 'https://aws.amazon.com/appsync/' }
  }
}
```

3. AWS AppSync 使用该值生成并执行 Amazon DynamoDB PutItem 请求。
4. AWS AppSync 将 PutItem 请求的结果转换回 GraphQL 类型。

```
{
    "id" : "123",
    "author": "AUTHORNAME",
    "title": "Our first post!",
    "content": "This is our first post.",
    "url": "https://aws.amazon.com/appsync/",
    "ups" : 1,
    "downs" : 0,
    "version" : 1
}
```

5. 响应处理程序立即返回结果 (return ctx.result)。
6. 最终结果显示在 GraphQL 响应中。

设置 getPost 解析器 (Amazon DynamoDB GetItem)

您现在能够将数据添加到 Amazon DynamoDB 表中，您需要设置 `getPost` 查询，以使其可以从表中检索该数据。为了实现此目的，您要设置另一解析器。

添加您的解析器

1. 在您的 API 中，选择架构选项卡。
2. 在右侧的解析器窗格中，找到 `Query` 类型上的 `getPost` 字段，然后选择附加。
3. 选择您的数据源，然后选择创建。
4. 在代码编辑器中，将代码替换为以下代码片段：

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 return ddb.get({ key: { id: ctx.args.id } })
}

export const response = (ctx) => ctx.result
```

5. 保存解析器。

> **Note**
>
> 在该解析器中，我们将箭头函数表达式用于响应处理程序。

调用 API 以获取文章

解析器现已设置完毕，AWS AppSync 知道如何将传入的 `getPost` 查询转换为 Amazon DynamoDB `GetItem` 操作。现在，您可以运行查询，检索之前创建的文章。

运行您的查询

1. 在您的 API 中，选择查询选项卡。
2. 在查询窗格中，添加以下代码，并使用您在创建文章后复制的 ID：

```
query getPost {
  getPost(id: "123") {
```

```
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. 选择运行（橙色播放按钮），然后选择 getPost。新创建的文章的结果应显示在查询窗格右侧的结果窗格中。
4. 从 Amazon DynamoDB 检索的文章应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "getPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

或者，采用以下示例：

```
query getPost {
  getPost(id: "123") {
    id
    author
    title
  }
}
```

如果您的 getPost 查询仅需要 id、author 和 title，您可以将请求函数更改为使用投影表达式仅指定您希望从 DynamoDB 表中获取的属性，以避免将不必要的数据从 DynamoDB 传输到 AWS AppSync。例如，请求函数可能类似于以下代码片段：

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 return ddb.get({
  key: { id: ctx.args.id },
  projection: ['author', 'id', 'title'],
 })
}

export const response = (ctx) => ctx.result
```

您也可以使用具有 getPost 的 selectionSetList 以表示 expression：

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 const projection = ctx.info.selectionSetList.map((field) => field.replace('/', '.'))
 return ddb.get({ key: { id: ctx.args.id }, projection })
}

export const response = (ctx) => ctx.result
```

创建 updatePost 变更 (Amazon DynamoDB UpdateItem)

到目前为止，您可以在 Amazon DynamoDB 中创建和检索 Post 对象。接下来，您设置一个新的变更以更新对象。与需要指定所有字段的 addPost 变更相比，该变更允许您仅指定要更改的字段。它还引入一个新的 expectedVersion 参数，以允许您指定要修改的版本。您设置一个条件，以确保您修改对象的最新版本。您将使用 UpdateItem Amazon DynamoDB 操作完成该操作。

更新您的解析器

1. 在您的 API 中，选择架构选项卡。
2. 在 Schema (架构) 窗格中修改 Mutation 类型，添加新的 updatePost 变更，如下所示：

```
type Mutation {
    updatePost(
```

```
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post

    addPost(
        id: ID
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}
```

3. 选择 Save Schema (保存架构)。
4. 在右侧的解析器窗格中，找到 Mutation 类型上的新创建的 updatePost 字段，然后选择附加。使用下面的代码片段创建新的解析器：

```
import { util } from '@aws-appsync/utils';
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { id, expectedVersion, ...rest } = ctx.args;
  const values = Object.entries(rest).reduce((obj, [key, value]) => {
    obj[key] = value ?? ddb.operations.remove();
    return obj;
  }, {});

  return ddb.update({
    key: { id },
    condition: { version: { eq: expectedVersion } },
    update: { ...values, version: ddb.operations.increment(1) },
  });
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type);
  }
```

```
  return result;
```

5. 保存您所做的任何更改。

该解析器使用 ddb.update 创建 Amazon DynamoDB UpdateItem 请求。您仅要求 Amazon DynamoDB 更新某些属性，而不是编写整个项目。这是使用 Amazon DynamoDB 更新表达式完成的。

ddb.update 函数将一个键和更新对象作为参数。然后，您检查传入的参数的值。在一个值设置为 null 时，使用 DynamoDB remove 操作指示应从 DynamoDB 项目中删除该值。

还有一个新的 condition 部分。通过使用条件表达式，您可以在执行操作之前根据 Amazon DynamoDB 中的已有对象的状态向 AWS AppSync 和 Amazon DynamoDB 通知请求是否会成功。在该示例中，只有在当前位于 Amazon DynamoDB 中的项目的 version 字段与 expectedVersion 参数完全匹配时，您才希望 UpdateItem 请求成功。在更新项目时，我们希望增加 version 的值。可以使用 increment 操作函数轻松完成该操作。

有关条件表达式的更多信息，请参阅条件表达式文档。

有关 UpdateItem 请求的更多信息，请参阅 UpdateItem 文档和 DynamoDB 模块文档。

有关如何编写更新表达式的更多信息，请参阅 DynamoDB UpdateExpressions 文档。

调用 API 以更新文章

让我们尝试使用新的解析器更新 Post 对象。

更新您的对象

1. 在您的 API 中，选择查询选项卡。
2. 在查询窗格中，添加以下变更。您还需要将 id 参数更新为您以前记下的值：

```
mutation updatePost {
  updatePost(
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 1
  ) {
    id
    author
    title
```

```
    content
    url
    ups
    downs
    version
  }
}
```

3. 选择运行（橙色播放按钮），然后选择 updatePost。

4. 在 Amazon DynamoDB 中更新的文章应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "updatePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 2
    }
  }
}
```

在该请求中，您要求 AWS AppSync 和 Amazon DynamoDB 仅更新 `title` 和 `content` 字段。所有其他字段保持不变（除了递增 `version` 字段以外）。您将 `title` 属性设置为新的值，并从文章中删除 `content` 属性。`author`、`url`、`ups` 和 `downs` 字段没有变化。再次尝试执行变更请求，同时将请求完全保持原样。您可以看到类似以下内容的响应：

```
{
  "data": {
    "updatePost": null
  },
  "errors": [
    {
      "path": [
        "updatePost"
      ],
      "data": null,
```

```
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "The conditional request failed (Service: DynamoDb, Status Code: 400,
  Request ID: 1RR3QN5F35CS8IV5VR4OQO9NNBVV4KQNSO5AEMVJF66Q9ASUAAJG)"
    }
  ]
}
```

请求失败，因为条件表达式的评估结果为 `false`：

1. 第一次运行请求时，Amazon DynamoDB 中的文章的 `version` 字段值为 1，它与 `expectedVersion` 参数匹配。请求成功，这意味着 Amazon DynamoDB 中的 `version` 字段已增加到 2。
2. 第二次运行请求时，Amazon DynamoDB 中的文章的 `version` 字段值为 2，它与 `expectedVersion` 参数不匹配。

这种模式通常被称为乐观锁。

创建评价变更 (Amazon DynamoDB UpdateItem)

`Post` 类型包含 `ups` 和 `downs` 字段，用于记录点赞和差评。不过，API 目前不允许我们进行任何评论。让我们添加一个变更，以对文章点赞和差评。

添加您的变更

1. 在您的 API 中，选择架构选项卡。
2. 在架构窗格中，修改 `Mutation` 类型并添加 `DIRECTION` 枚举以添加新的评论变更：

```
type Mutation {
    vote(id: ID!, direction: DIRECTION!): Post
    updatePost(
        id: ID!,
        author: String,
```

```
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        id: ID,
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post!
}

enum DIRECTION {
  UP
  DOWN
}
```

3. 选择 Save Schema (保存架构)。
4. 在右侧的解析器窗格中，找到 Mutation 类型上的新创建的 vote 字段，然后选择附加。创建代码并将其替换为以下代码片段，以创建新的解析器：

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const field = ctx.args.direction === 'UP' ? 'ups' : 'downs';
  return ddb.update({
    key: { id: ctx.args.id },
    update: {
      [field]: ddb.operations.increment(1),
      version: ddb.operations.increment(1),
    },
  });
}

export const response = (ctx) => ctx.result;
```

5. 保存您所做的任何更改。

调用 API 以对文章点赞或差评

新的解析器现已设置完毕，AWS AppSync 知道如何将传入的 upvotePost 或 downvote 变更转换为 Amazon DynamoDB UpdateItem 操作。现在您可以运行变更，为之前创建的文章点赞或差评。

运行您的变更

1. 在您的 API 中，选择查询选项卡。
2. 在查询窗格中，添加以下变更。您还需要将 id 参数更新为您以前记下的值：

```
mutation votePost {
  vote(id:123, direction: UP) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. 选择运行（橙色播放按钮），然后选择 votePost。
4. 在 Amazon DynamoDB 中更新的文章应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "vote": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 0,
      "version": 4
    }
  }
}
```

5. 再选择几次运行。每次执行查询时，您应该会看到 ups 和 version 字段增加 1。

6. 更改查询以使用不同的 DIRECTION 进行调用。

```
mutation votePost {
  vote(id:123, direction: DOWN) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

7. 选择运行（橙色播放按钮），然后选择 votePost。

   这次，每次运行查询时，您应该会看到 downs 和 version 字段增加 1。

设置 deletePost 解析器 (Amazon DynamoDB DeleteItem)

接下来，您希望创建一个变更以删除文章。您将使用 DeleteItem Amazon DynamoDB 操作完成该操作。

添加您的变更

1. 在您的架构中，选择架构选项卡。
2. 在架构窗格中，修改 Mutation 类型以添加新的 deletePost 变更：

```
type Mutation {
    deletePost(id: ID!, expectedVersion: Int): Post
    vote(id: ID!, direction: DIRECTION!): Post
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        id: ID
```

```
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post!
}
```

3. 这次，您将 expectedVersion 字段设置为可选。接下来，选择保存架构。
4. 在右侧的解析器窗格中，找到 Mutation 类型中的新创建的 delete 字段，然后选择附加。使用以下代码创建一个新的解析器：

```
import { util } from '@aws-appsync/utils'

import { util } from '@aws-appsync/utils';
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  let condition = null;
  if (ctx.args.expectedVersion) {
    condition = {
      or: [
        { id: { attributeExists: false } },
        { version: { eq: ctx.args.expectedVersion } },
      ],
    };
  }
  return ddb.remove({ key: { id: ctx.args.id }, condition });
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type);
  }
  return result;
}
```

> **Note**
>
> expectedVersion 参数是可选的参数。如果调用方在请求中设置 expectedVersion 参数，请求处理程序将添加一个条件，只有在已删除项目或 Amazon DynamoDB 中的文章的 version 属性与 expectedVersion 完全匹配时，才允许 DeleteItem 请求成功。如果

未设置此参数，则 DeleteItem 请求中不指定条件表达式。无论 version 值如何，或者项目在 Amazon DynamoDB 中是否存在，该请求都会成功。
即使您要删除一个项目，如果尚未删除，您也可以返回要删除的项目。

有关 DeleteItem 请求的更多信息，请参阅 DeleteItem 文档。

调用 API 以删除文章

解析器现已设置完毕，AWS AppSync 知道如何将传入的 delete 变更转换为 Amazon DynamoDB DeleteItem 操作。现在，您可以运行变更，从表中删除一些内容。

运行您的变更

1. 在您的 API 中，选择查询选项卡。
2. 在查询窗格中，添加以下变更。您还需要将 id 参数更新为您以前记下的值：

```
mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. 选择运行（橙色播放按钮），然后选择 deletePost。
4. 将从 Amazon DynamoDB 中删除该文章。请注意，AWS AppSync 返回从 Amazon DynamoDB 中删除的项目的值，它应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
```

```
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 4,
      "version": 12
    }
  }
}
```

5. 只有在对 deletePost 的该调用实际将其从 Amazon DynamoDB 中删除时，才会返回该值。再次选择运行。
6. 调用仍然成功，但没有返回任何值：

```
{
  "data": {
    "deletePost": null
  }
}
```

7. 现在，让我们尝试删除一篇文章，但这次指定 expectedValue。首先，您需要创建一个新文章，因为您刚刚删除了迄今为止一直使用的文章。
8. 在查询窗格中，添加以下变更：

```
mutation addPost {
  addPost(
    id:123
    author: "AUTHORNAME"
    title: "Our second post!"
    content: "A new post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

9. 选择运行（橙色播放按钮），然后选择 addPost。

10新创建的文章的结果应显示在查询窗格右侧的结果窗格中。记下新创建的对象的 id，因为您稍后需要使用该 ID。如下所示：

```
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

11现在，让我们尝试删除具有非法 expectedVersion 值的文章。在查询窗格中，添加以下变更。您还需要将 id 参数更新为您以前记下的值：

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

12选择运行（橙色播放按钮），然后选择 deletePost。将返回以下结果：

```
{
  "data": {
    "deletePost": null
  },
```

```
  "errors": [
    {
      "path": [
        "deletePost"
      ],
      "data": null,
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "The conditional request failed (Service: DynamoDb, Status Code:
 400, Request ID: 7083O037M1FTFRK038A4CI9H43VV4KQNSO5AEMVJF66Q9ASUAAJG)"
    }
  ]
}
```

13.请求失败，因为条件表达式的评估结果为 `false`。Amazon DynamoDB 中的文章 `version` 值与参数中指定的 `expectedValue` 不匹配。对象的当前值返回到 GraphQL 响应的 `data` 部分的 `errors` 字段中。重试请求，但更正 `expectedVersion`：

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

14.选择运行（橙色播放按钮），然后选择 `deletePost`。

这次请求成功，并返回从 Amazon DynamoDB 中删除的值：

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

15.再次选择运行。调用仍然成功，但这次没有返回任何值，因为已在 Amazon DynamoDB 中删除该文章。

```
{ "data": { "deletePost": null } }
```

设置 allPost 解析器 (Amazon DynamoDB Scan)

到目前为止，只有在您知道要查看的每篇文章的 `id` 时，才能使用该 API。让我们添加新的解析器，它可以返回表中的所有文章。

添加您的变更

1. 在您的 API 中，选择架构选项卡。
2. 在 Schema (架构) 窗格中修改 `Query` 类型，添加新的 `allPost` 查询，如下所示：

```
type Query {
    allPost(limit: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

3. 添加新 `PaginationPosts` 类型：

```
type PaginatedPosts {
```

```
    posts: [Post!]!
    nextToken: String
}
```

4. 选择 Save Schema (保存架构)。
5. 在右侧的解析器窗格中，找到 Query 类型中的新创建的 allPost 字段，然后选择附加。使用以下代码创建一个新的解析器：

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { limit = 20, nextToken } = ctx.arguments;
  return ddb.scan({ limit, nextToken });
}

export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

该解析器的请求处理程序需要使用两个可选的参数：

- limit - 指定单次调用中返回的最大项目数。
- nextToken - 用于检索下一组结果（我们稍后将显示 nextToken 值来自何处）。

6. 保存对您的解析器所做的任何更改。

有关 Scan 请求的更多信息，请参阅 Scan 参考文档。

调用 API 以扫描所有文章

解析器现已设置完毕，AWS AppSync 知道如何将传入的 allPost 查询转换为 Amazon DynamoDB Scan 操作。现在您可以扫描整个表，检索所有文章。在进行尝试之前，您需要在表中填充一些数据，因为您已经删除了之前使用的所有内容。

添加和查询数据

1. 在您的 API 中，选择查询选项卡。
2. 在查询窗格中，添加以下变更：

```
mutation addPost {
```

```
  post1: addPost(id:1 author: "AUTHORNAME" title: "A series of posts, Volume 1"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post2: addPost(id:2 author: "AUTHORNAME" title: "A series of posts, Volume 2"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post3: addPost(id:3 author: "AUTHORNAME" title: "A series of posts, Volume 3"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post4: addPost(id:4 author: "AUTHORNAME" title: "A series of posts, Volume 4"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post5: addPost(id:5 author: "AUTHORNAME" title: "A series of posts, Volume 5"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post6: addPost(id:6 author: "AUTHORNAME" title: "A series of posts, Volume 6"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post7: addPost(id:7 author: "AUTHORNAME" title: "A series of posts, Volume 7"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post8: addPost(id:8 author: "AUTHORNAME" title: "A series of posts, Volume 8"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post9: addPost(id:9 author: "AUTHORNAME" title: "A series of posts, Volume 9"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
}
```

3. 选择运行（橙色播放按钮）。

4. 现在，让我们扫描表，每次返回 5 个结果。在查询窗格中，添加以下查询：

```
query allPost {
  allPost(limit: 5) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

5. 选择运行（橙色播放按钮），然后选择 `allPost`。

   前 5 篇文章应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "5",
```

```
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        },
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        }
      ],
      "nextToken": "<token>"
    }
  }
}
```

6. 您收到 5 个结果和一个 nextToken（可用于获取下一组结果）。更新 allPost 查询，加入上一组结果的 nextToken：

```
query allPost {
  allPost(
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
      author
    }
    nextToken
  }
}
```

7. 选择运行（橙色播放按钮），然后选择 allPost。

其余 4 篇文章应显示在查询窗格右侧的结果窗格中。在这组结果中没有 nextToken，因为您已查看了所有 9 篇文章，没有其余文章了。如下所示：

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        }
      ],
      "nextToken": null
    }
  }
}
```

设置 allPostsByAuthor 解析器 (Amazon DynamoDB Query)

除了扫描 Amazon DynamoDB 以查找所有文章以外，您还可以查询 Amazon DynamoDB 以检索特定作者创建的文章。您以前创建的 Amazon DynamoDB 表已具有一个名为 author-index 的 GlobalSecondaryIndex，您可以将其与 Amazon DynamoDB Query 操作一起使用以检索特定作者创建的所有文章。

添加您的查询

1. 在您的 API 中，选择架构选项卡。
2. 在 Schema (架构) 窗格中修改 Query 类型，添加新的 allPostsByAuthor 查询，如下所示：

```
type Query {
    allPostsByAuthor(author: String!, limit: Int, nextToken: String): PaginatedPosts!
    allPost(limit: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

请注意，它使用与 allPost 查询相同的 PaginatedPosts 类型。

3. 选择 Save Schema (保存架构)。
4. 在右侧的解析器窗格中，找到 Query 类型上的新创建的 allPostsByAuthor 字段，然后选择附加。使用下面的代码片段创建一个解析器：

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { limit = 20, nextToken, author } = ctx.arguments;
  return ddb.query({
    index: 'author-index',
    query: { author: { eq: author } },
    limit,
    nextToken,
  });
}

export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

与 allPost 解析器一样，该解析器具有两个可选的参数：

- limit - 指定单次调用中返回的最大项目数。
- nextToken - 检索下一组结果（可以从以前的调用中获取 nextToken 值）。

5. 保存对您的解析器所做的任何更改。

有关 Query 请求的更多信息，请参阅 Query 参考文档。

调用 API 以查询作者的所有文章

解析器现已设置完毕，AWS AppSync 知道如何将传入的 `allPostsByAuthor` 变更转换为针对 `author-index` 索引的 DynamoDB `Query` 操作。现在，您可以查询表，检索某一作者的所有文章。

不过，在此之前，让我们在表中再填充一些文章，因为到目前为止每篇文章都是由同一作者撰写的。

添加数据和查询

1. 在您的 API 中，选择查询选项卡。
2. 在查询窗格中，添加以下变更：

```
mutation addPost {
  post1: addPost(id:10 author: "Nadia" title: "The cutest dog in the world" content:
 "So cute. So very, very cute." url: "https://aws.amazon.com/appsync/" ) { author,
 title }
  post2: addPost(id:11 author: "Nadia" title: "Did you know...?" content: "AppSync
 works offline?" url: "https://aws.amazon.com/appsync/" ) { author, title }
  post3: addPost(id:12 author: "Steve" title: "I like GraphQL" content: "It's great"
 url: "https://aws.amazon.com/appsync/" ) { author, title }
}
```

3. 选择运行（橙色播放按钮），然后选择 `addPost`。
4. 现在，让我们查询表，返回作者为 `Nadia` 的所有文章。在查询窗格中，添加以下查询：

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

5. 选择运行（橙色播放按钮），然后选择 `allPostsByAuthor`。Nadia 撰写的所有文章应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
```

```
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

6. Query 的分页方式与 Scan 相同。例如，如果我们查找作者为 AUTHORNAME 的所有文章，每次显示 5 个结果。
7. 在查询窗格中，添加以下查询：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

8. 选择运行（橙色播放按钮），然后选择 allPostsByAuthor。AUTHORNAME 撰写的所有文章应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
```

```
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        }
      ],
      "nextToken": "<token>"
    }
  }
}
```

9. 用上次查询返回的值更新 nextToken 参数，如下所示：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

10选择运行（橙色播放按钮），然后选择 allPostsByAuthor。AUTHORNAME 撰写的其余文章应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "allPostsByAuthor": {
```

```
      "posts": [
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        },
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        }
      ],
      "nextToken": null
    }
  }
}
```

使用集

到目前为止，Post 类型一直是平面键/值对象。您也可以使用解析器对复杂对象进行建模，例如集、列表和映射。让我们更新 Post 类型，加入标签。一篇文章可以具有 0 个或更多标签，这些标签作为字符串集存储在 DynamoDB 中。您还将设置一些变更，用于添加并删除标签；还要用一个新查询扫描具有特定标签的文章。

设置您的数据

1. 在您的 API 中，选择架构选项卡。
2. 在 Schema (架构) 窗格中修改 Post 类型，添加新的 tags 字段，如下所示：

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
```

```
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
}
```

3. 在 Schema (架构) 窗格中修改 Query 类型，添加新的 allPostsByTag 查询，如下所示：

```
type Query {
  allPostsByTag(tag: String!, limit: Int, nextToken: String): PaginatedPosts!
  allPostsByAuthor(author: String!, limit: Int, nextToken: String): PaginatedPosts!
  allPost(limit: Int, nextToken: String): PaginatedPosts!
  getPost(id: ID): Post
}
```

4. 在 Schema (架构) 窗格中修改 Mutation 类型，添加新的 addTag 和 removeTag 变更，如下所示：

```
type Mutation {
  addTag(id: ID!, tag: String!): Post
  removeTag(id: ID!, tag: String!): Post
  deletePost(id: ID!, expectedVersion: Int): Post
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}
```

5. 选择 Save Schema (保存架构)。
6. 在右侧的解析器窗格中，找到 Query 类型上的新创建的 allPostsByTag 字段，然后选择附加。使用下面的代码片段创建您的解析器：

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
  const { limit = 20, nextToken, tag } = ctx.arguments;
  return ddb.scan({ limit, nextToken, filter: { tags: { contains: tag } } });
}

export function response(ctx) {
  const { items: posts = [], nextToken } = ctx.result;
  return { posts, nextToken };
}
```

7. 保存对您的解析器所做的任何更改。
8. 现在，使用下面的代码片段为 Mutation 字段 addTag 执行相同的操作：

> **Note**
>
> 尽管 DynamoDB 实用程序当前不支持集操作，但您仍然可以自行构建请求以与集进行交互。

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { id, tag } = ctx.arguments
 const expressionValues = util.dynamodb.toMapValues({ ':plusOne': 1 })
 expressionValues[':tags'] = util.dynamodb.toStringSet([tag])

 return {
  operation: 'UpdateItem',
  key: util.dynamodb.toMapValues({ id }),
  update: {
   expression: `ADD tags :tags, version :plusOne`,
   expressionValues,
  },
 }
}

export const response = (ctx) => ctx.result
```

9. 保存对您的解析器所做的任何更改。

10.使用下面的代码片段为 Mutation 字段 removeTag 再重复一次该操作：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
   const { id, tag } = ctx.arguments;
   const expressionValues = util.dynamodb.toMapValues({ ':plusOne': 1 });
   expressionValues[':tags'] = util.dynamodb.toStringSet([tag]);

   return {
     operation: 'UpdateItem',
     key: util.dynamodb.toMapValues({ id }),
     update: {
       expression: `DELETE tags :tags ADD version :plusOne`,
       expressionValues,
     },
   };
 }

 export const response = (ctx) => ctx.resultexport
```

11.保存对您的解析器所做的任何更改。

调用 API 以处理标签

您现已设置解析器，AWS AppSync 知道如何将传入的 addTag、removeTag 和 allPostsByTag 请求转换为 DynamoDB UpdateItem 和 Scan 操作。我们选择您之前创建的一个文章进行尝试。例如，我们使用作者为 Nadia 的一篇文章。

使用标签

1. 在您的 API 中，选择查询选项卡。
2. 在查询窗格中，添加以下查询：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
```

```
    nextToken
  }
}
```

3. 选择运行（橙色播放按钮），然后选择 allPostsByAuthor。
4. Nadia 的所有文章应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

5. 让我们使用标题为 The cutest dog in the world 的文章。记下其 id，因为您稍后使用该 ID。现在，让我们尝试添加一个 dog 标签。
6. 在查询窗格中，添加以下变更。您还需要将 id 参数更新为您以前记下的值。

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

7. 选择运行（橙色播放按钮），然后选择 addTag。将使用新标签更新该文章：

```
{
  "data": {
    "addTag": {
```

```
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

8. 您可以添加更多标签。更新变更以将 tag 参数更改为 puppy：

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

9. 选择运行（橙色播放按钮），然后选择 addTag。将使用新标签更新该文章：

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

10您也可以删除标签。在查询窗格中，添加以下变更。您还需要将 id 参数更新为您以前记下的值：

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

11选择运行（橙色播放按钮），然后选择 removeTag。文章已更新，puppy 标签已删除。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

12您也可以搜索所有具有标签的文章。在查询窗格中，添加以下查询：

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

13选择运行（橙色播放按钮），然后选择 allPostsByTag。将返回具有 dog 标签的所有文章，如下所示：

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
            "dog",
            "puppy"
          ]
        }
      ],
```

```
      "nextToken": null
    }
  }
}
```

结论

在本教程中，您构建了一个 API，可用于通过 AWS AppSync 和 GraphQL 处理 DynamoDB 中的 `Post` 对象。

要进行清理，您可以从控制台中删除 AWS AppSync GraphQL API。

要删除与您的 DynamoDB 表关联的角色，请在数据源表中选择您的数据源，然后单击编辑。记下创建或使用现有角色下面的角色值。转到 IAM 控制台以删除该角色。

要删除您的 DynamoDB 表，请在数据源列表中单击该表名称。这会转到 DynamoDB 控制台，您可以在其中删除该表。

教程：Lambda 解析器

您可以将 AWS Lambda 与 AWS AppSync 一起使用以解析任何 GraphQL 字段。例如，GraphQL 查询可能会向 Amazon Relational Database Service (Amazon RDS) 实例发送调用，而 GraphQL 变更可能会写入到 Amazon Kinesis 流。在本节中，我们说明了如何编写 Lambda 函数，以根据 GraphQL 字段操作调用执行业务逻辑。

创建 Lambda 函数

以下示例显示了一个使用 `Node.js`（运行时环境：Node.js 18.x）编写的 Lambda 函数，该函数对博客文章应用程序包含的博客文章执行各种操作。请注意，代码应保存在具有 .mis 扩展名的文件中。

```
export const handler = async (event) => {
console.log('Received event {}', JSON.stringify(event, 3))

  const posts = {
1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
 content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
 AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
 content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
 '10', },
```

```
    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
 ups: null, downs: null },
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }

  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
case 'getPost':
      return posts[event.arguments.id]
    case 'allPosts':
      return Object.values(posts)
    case 'addPost':
      // return the arguments back
return event.arguments
    case 'addPostErrorWithData':
      result = posts[event.arguments.id]
      // attached additional error information to the post
      result.errorMessage = 'Error with the mutation, data has changed'
      result.errorType = 'MUTATION_ERROR'
return result
    case 'relatedPosts':
      return relatedPosts[event.source.id]
    default:
      throw new Error('Unknown field, unable to resolve ' + event.field)
  }
}
```

该 Lambda 函数按 ID 检索文章，添加文章，检索文章列表以及获取给定文章的相关文章。

> **Note**
>
> Lambda 函数对 `event.field` 执行 `switch` 语句以确定当前解析的字段。

使用 AWS 管理控制台创建该 Lambda 函数。

为 Lambda 配置数据源

在创建 Lambda 函数后，在 AWS AppSync 控制台中导航到您的 GraphQL API，然后选择数据源选项卡。

选择创建数据源，输入友好的数据源名称（例如 **Lambda**），然后为数据源类型选择 AWS Lambda 函数。对于区域，选择与您的函数相同的区域。对于函数 ARN，选择您的 Lambda 函数的 Amazon 资源名称 (ARN)。

在选择您的 Lambda 函数后，您可以创建新的 AWS Identity and Access Management (IAM) 角色（AWS AppSync 为其分配相应的权限），或选择具有以下内联策略的现有角色：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/LAMBDA_FUNCTION"
        }
    ]
}
```

您还必须为 IAM 角色建立与 AWS AppSync 信任关系，如下所示：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "appsync.amazonaws.com"
            },
```

```
            "Action": "sts:AssumeRole"
        }
    ]
}
```

创建 GraphQL 架构

数据源现已连接到您的 Lambda 函数，请创建 GraphQL 架构。

从 AWS AppSync 控制台的架构编辑器中，确保您的架构与以下架构匹配：

```
schema {
    query: Query
    mutation: Mutation
}
type Query {
    getPost(id:ID!): Post
    allPosts: [Post]
}
type Mutation {
    addPost(id: ID!, author: String!, title: String, content: String, url: String):
 Post!
}
type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    relatedPosts: [Post]
}
```

配置解析器

您现已注册了 Lambda 数据源和有效的 GraphQL 架构，您可以使用解析器将 GraphQL 字段连接到 Lambda 数据源。

您将创建一个解析器，它使用 AWS AppSync JavaScript (APPSYNC_JS) 运行时环境并与您的 Lambda 函数交互。要了解使用 JavaScript 编写 AWS AppSync 解析器和函数的更多信息，请参阅解析器和函数的 JavaScript 运行时功能。

有关 Lambda 映射模板的更多信息，请参阅 Lambda 的 JavaScript 解析器函数参考。

在该步骤中，您将一个解析器附加到以下字段的 Lambda 函数：`getPost(id:ID!): Post`、`allPosts: [Post]`、`addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` 和 `Post.relatedPosts: [Post]`。从 AWS AppSync 控制台的架构编辑器中，在解析器窗格中选择 `getPost(id:ID!): Post` 字段旁边的附加。选择您的 Lambda 数据源。接下来，提供以下代码：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  return ctx.result;
}
```

在调用 Lambda 函数时，该解析器代码将字段名称、参数列表以及有关源对象的上下文传递给该函数。选择保存。

您已成功附加了您的首个解析器。对于其余字段，重复该操作。

测试您的 GraphQL API

现在您的 Lambda 函数已与 GraphQL 解析器连接，您可以使用控制台或客户端应用程序运行一些变更和查询。

在 AWS AppSync 控制台左侧，选择查询，然后粘贴以下代码：

addPost 变更

```
mutation AddPost {
    addPost(
        id: 6
        author: "Author6"
```

```
        title: "Sixth book"
        url: "https://www.amazon.com/"
        content: "This is the book is a tutorial for using GraphQL with AWS AppSync."
    ) {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

getPost 查询

```
query GetPost {
    getPost(id: "2") {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

allPosts 查询

```
query AllPosts {
    allPosts {
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {
            id
            title
```

```
        }
    }
}
```

返回错误

解析任何给定的字段可能会导致错误。在使用 AWS AppSync 时，您可以从以下源中引发错误：

- 解析器响应处理程序
- Lambda 函数

从解析器响应处理程序中

要故意引发错误，您可以使用 `util.error` 实用程序方法。它将 `errorMessage`、`errorType` 和可选的 `data` 值作为参数。出现错误后，`data` 对于将额外的数据返回客户端很有用。在 GraphQL 最终响应中，`data` 对象将添加到 `errors`。

以下示例说明了如何在 `Post.relatedPosts: [Post]` 解析器响应处理程序中使用该方法。

```
// the Post.relatedPosts response handler
export function response(ctx) {
    util.error("Failed to fetch relatedPosts", "LambdaFailure", ctx.result)
    return ctx.result;
}
```

这将生成与以下内容类似的 GraphQL 响应：

```
{
    "data": {
        "allPosts": [
            {
                "id": "2",
                "title": "Second book",
                "relatedPosts": null
            },
            ...
        ]
    },
    "errors": [
        {
```

```
            "path": [
                "allPosts",
                0,
                "relatedPosts"
            ],
            "errorType": "LambdaFailure",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ],
            "message": "Failed to fetch relatedPosts",
            "data": [
                {
                  "id": "2",
                  "title": "Second book"
                },
                {
                  "id": "1",
                  "title": "First book"
                }
            ]
        }
    ]
}
```

错误导致 `allPosts[0].relatedPosts` 为 null，而 `errorMessage`、`errorType` 和 `data` 体现在 `data.errors[0]` 对象中。

从 Lambda 函数

AWS AppSync 还可以识别 Lambda 函数引发的错误。Lambda 编程模型允许您引发处理的 错误。如果 Lambda 函数引发错误，则 AWS AppSync 无法解析当前字段。仅在响应中设置从 Lambda 返回的错误消息。目前，您无法通过从 Lambda 函数中引发错误，将任何无关数据传回到客户端。

> **Note**
>
> 如果您的 Lambda 函数引发未处理的 错误，AWS AppSync 将使用 Lambda 设置的错误消息。

以下 Lambda 函数会引发错误：

```
export const handler = async (event) => {
  console.log('Received event {}', JSON.stringify(event, 3))
  throw new Error('I always fail.')
}
```

在您的响应处理程序中收到错误。您可以使用 `util.appendError` 将错误附加到 GraphQL 的响应，以将其发回到响应。为此，请将您的 AWS AppSync 函数响应处理程序更改为：

```
// the lambdaInvoke response handler
export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  }
  return result;
}
```

这将返回与以下内容类似的 GraphQL 响应：

```
{
  "data": {
    "allPosts": null
  },
  "errors": [
    {
      "path": [
        "allPosts"
      ],
      "data": null,
      "errorType": "Lambda:Unhandled",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "I fail. always"
    }
  ]
}
```

高级使用案例：批处理

该示例中的 Lambda 函数具有一个 `relatedPosts` 字段，它返回给定文章的相关文章列表。在示例查询中，从 Lambda 函数中调用 `allPosts` 字段将返回 5 篇文章。由于我们指定还希望为每个返回的文章解析 `relatedPosts`，因此，将 `relatedPosts` 字段操作调用 5 次。

```
query {
    allPosts {   // 1 Lambda invocation - yields 5 Posts
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {   // 5 Lambda invocations - each yields 5 posts
            id
            title
        }
    }
}
```

虽然这在该特定示例中听起来可能并不严重，但这种累积的过度获取可能会迅速降低应用程序的性能。

如果您要针对同一查询中返回的相关 Posts 再次提取 `relatedPosts`，调用数量将显著增加。

```
query {
    allPosts {   // 1 Lambda invocation - yields 5 Posts
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {   // 5 Lambda invocations - each yield 5 posts = 5 x 5 Posts
            id
            title
            relatedPosts {  // 5 x 5 Lambda invocations - each yield 5 posts = 25 x 5
 Posts
                id
                title
```

```
                author
            }
        }
    }
}
```

在这个相对简单的查询中，AWS AppSync 将调用 Lambda 函数 1 + 5 + 25 = 31 次。

这是相当常见的挑战，常被称为 N+1 问题（在本例中 N = 5），会导致延迟增加，以及应用程序费用提升。

我们解决此问题的方式是批处理类似的字段解析器请求。在该示例中，Lambda 函数可能会解析给定批次的文章的相关文章列表，而不是让 Lambda 函数解析单个给定文章的相关文章列表。

为了说明这一点，让我们更新 `relatedPosts` 的解析器以进行批处理。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: ctx.info.fieldName === 'relatedPosts' ? 'BatchInvoke' : 'Invoke',
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  }
  return result;
}
```

现在，在解析的 `fieldName` 为 `relatedPosts` 时，代码将操作从 `Invoke` 更改为 `BatchInvoke`。现在，在配置批处理部分中为函数启用批处理。将最大批处理大小设置为 5。选择保存。

通过进行该更改，在解析 `relatedPosts` 时，Lambda 函数收到以下内容以作为输入：

```
[
    {
        "field": "relatedPosts",
        "source": {
```

```
            "id": 1
        }
    },
    {
        "field": "relatedPosts",
        "source": {
            "id": 2
        }
    },
    ...
]
```

如果在请求中指定了 BatchInvoke，Lambda 函数将收到请求列表并返回结果列表。

具体来说，结果列表必须与请求负载条目的大小和顺序匹配，以使 AWS AppSync 可以相应地匹配结果。

在该批处理示例中，Lambda 函数返回一批结果，如下所示：

```
[
    [{"id":"2","title":"Second book"}, {"id":"3","title":"Third book"}],   //
 relatedPosts for id=1
    [{"id":"3","title":"Third book"}]                                      //
 relatedPosts for id=2
]
```

您可以更新 Lambda 代码以处理 relatedPosts 批处理：

```
export const handler = async (event) => {
  console.log('Received event {}', JSON.stringify(event, 3))
  //throw new Error('I fail. always')

  const posts = {
    1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
 content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
 AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
 content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
 '10', },
    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
 ups: null, downs: null },
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
```

```
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
    1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }

  if (!event.field && event.length){
    console.log(`Got a BatchInvoke Request. The payload has ${event.length} items to
 resolve.`);
    return event.map(e => relatedPosts[e.source.id])
  }

  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
    case 'getPost':
      return posts[event.arguments.id]
    case 'allPosts':
      return Object.values(posts)
    case 'addPost':
      // return the arguments back
      return event.arguments
    case 'addPostErrorWithData':
      result = posts[event.arguments.id]
      // attached additional error information to the post
      result.errorMessage = 'Error with the mutation, data has changed'
      result.errorType = 'MUTATION_ERROR'
      return result
    case 'relatedPosts':
      return relatedPosts[event.source.id]
    default:
      throw new Error('Unknown field, unable to resolve ' + event.field)
  }
}
```

返回单个错误

以前的示例表明，可以从 Lambda 函数中返回单个错误，或者从响应处理程序中引发错误。对于批处理调用，从 Lambda 函数中引发错误会将整个批次标记为失败。对于发生不可恢复错误的特定场景（例如，到数据存储的连接失败），这可能是可以接受的。不过，如果批次中的某些项目成功，而其他项目失败，则可能会同时返回错误和有效的数据。由于 AWS AppSync 要求批处理响应列出与批次的原始大小匹配的元素，因此，您必须定义一个可以区分有效数据和错误的数据结构。

例如，如果 Lambda 函数预计返回一批相关文章，您可以选择返回 `Response` 对象列表，其中每个对象具有可选的 data、errorMessage 和 errorType 字段。如果出现 errorMessage 字段，则表示出现错误。

以下代码说明了如何更新 Lambda 函数：

```
export const handler = async (event) => {
console.log('Received event {}', JSON.stringify(event, 3))
  // throw new Error('I fail. always')
const posts = {
1: { id: '1', title: 'First book', author: 'Author1', url: 'https://amazon.com/',
 content: 'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
 AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1', ups: '100', downs: '10', },
    2: { id: '2', title: 'Second book', author: 'Author2', url: 'https://amazon.com',
 content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT', ups: '100', downs:
 '10', },
    3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null,
 ups: null, downs: null },
    4: { id: '4', title: 'Fourth book', author: 'Author4', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4', ups: '1000', downs: '0', },
    5: { id: '5', title: 'Fifth book', author: 'Author5', url: 'https://
www.amazon.com/', content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT', ups: '50', downs: '0', },
  }

  const relatedPosts = {
1: [posts['4']],
    2: [posts['3'], posts['5']],
    3: [posts['2'], posts['1']],
    4: [posts['2'], posts['1']],
    5: [],
  }
```

```
  if (!event.field && event.length){
console.log(`Got a BatchInvoke Request. The payload has ${event.length} items to
 resolve.`);
    return event.map(e => {
// return an error for post 2
if (e.source.id === '2') {
return { 'data': null, 'errorMessage': 'Error Happened', 'errorType': 'ERROR' }
      }
      return {data: relatedPosts[e.source.id]}
      })
  }

  console.log('Got an Invoke Request.')
  let result
  switch (event.field) {
case 'getPost':
      return posts[event.arguments.id]
    case 'allPosts':
      return Object.values(posts)
    case 'addPost':
      // return the arguments back
return event.arguments
    case 'addPostErrorWithData':
      result = posts[event.arguments.id]
      // attached additional error information to the post
      result.errorMessage = 'Error with the mutation, data has changed'
      result.errorType = 'MUTATION_ERROR'
return result
    case 'relatedPosts':
      return relatedPosts[event.source.id]
    default:
      throw new Error('Unknown field, unable to resolve ' + event.field)
  }
}
```

更新 `relatedPosts` 解析器代码：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const {source, args} = ctx
  return {
    operation: ctx.info.fieldName === 'relatedPosts' ? 'BatchInvoke' : 'Invoke',
```

```
    payload: { field: ctx.info.fieldName, arguments: args, source },
  };
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  } else if (result.errorMessage) {
    util.appendError(result.errorMessage, result.errorType, result.data)
  } else if (ctx.info.fieldName === 'relatedPosts') {
      return result.data
  } else {
      return result
  }
}
```

响应处理程序现在检查 Lambda 函数在 Invoke 操作中返回的错误，检查 BatchInvoke 操作为各个项目返回的错误，最后检查 fieldName。对于 relatedPosts，该函数返回 result.data。对于所有其他字段，该函数仅返回 result。例如，请参阅下面的查询：

```
query AllPosts {
  allPosts {
    id
    title
    content
    url
    ups
    downs
    relatedPosts {
      id
    }
    author
  }
}
```

该查询返回类似下面的 GraphQL 响应：

```
{
  "data": {
    "allPosts": [
      {
```

```
        "id": "1",
        "relatedPosts": [
          {
            "id": "4"
          }
        ]
      },
      {
        "id": "2",
        "relatedPosts": null
      },
      {
        "id": "3",
        "relatedPosts": [
          {
            "id": "2"
          },
          {
            "id": "1"
          }
        ]
      },
      {
        "id": "4",
        "relatedPosts": [
          {
            "id": "2"
          },
          {
            "id": "1"
          }
        ]
      },
      {
        "id": "5",
        "relatedPosts": []
      }
    ]
  },
  "errors": [
    {
      "path": [
        "allPosts",
        1,
```

```
        "relatedPosts"
      ],
      "data": null,
      "errorType": "ERROR",
      "errorInfo": null,
      "locations": [
        {
          "line": 4,
          "column": 5,
          "sourceName": null
        }
      ],
      "message": "Error Happened"
    }
  ]
}
```

配置最大批处理大小

要配置解析器上的最大批处理大小，请在 AWS Command Line Interface (AWS CLI) 中使用以下命令：

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
 relatedPosts \
 --code "<code-goes-here>" \
 --runtime name=APPSYNC_JS,runtimeVersion=1.0.0 \
 --data-source-name "<lambda-datasource>" \
 --max-batch-size X
```

> **Note**
>
> 在提供请求映射模板时，您必须使用 `BatchInvoke` 操作才能使用批处理。

教程：本地解析器

AWS AppSync 允许您使用支持的数据源（AWS Lambda、Amazon DynamoDB 或 Amazon OpenSearch Service）执行各种操作。但在某些情况下，可能不必调用支持的数据源。

这时本地解析器就很方便。本地解析器仅将请求处理程序结果转发到响应处理程序，而不会调用远程数据源。字段解析是在 AWS AppSync 中完成的。

在很多情况下，本地解析器是非常有用的。最常用的使用案例是在不触发数据源调用的情况下发布通知。为了说明该使用案例，让我们构建一个 pub/sub 应用程序，用户可以在其中发布和订阅消息。此示例利用了订阅，如果您不熟悉订阅，可以参考实时数据教程。

创建 pub/sub 应用程序

首先，创建一个空白的 GraphQL API，方法是选择从头开始设计选项，并在创建 GraphQL API 时配置可选的详细信息。

在我们的 pub/sub 应用程序中，客户端可以订阅和发布消息。每条发布的消息包含名称和数据。将以下内容添加到架构中：

```
type Channel {
 name: String!
 data: AWSJSON!
}

type Mutation {
 publish(name: String!, data: AWSJSON!): Channel
}

type Query {
 getChannel: Channel
}

type Subscription {
 subscribe(name: String!): Channel
  @aws_subscribe(mutations: ["publish"])
}
```

接下来，让我们将一个解析器附加到 `Mutation.publish` 字段。在架构窗格旁边的解析器窗格中，找到 `Mutation` 类型，找到 `publish(...): Channel` 字段，然后单击附加。

创建一个 None 数据源，并将其命名为 PageDataSource。将其附加到您的解析器。

使用以下代码片段添加您的解析器实施：

```
export function request(ctx) {
  return { payload: ctx.args };
}

export function response(ctx) {
```

```
  return ctx.result;
}
```

确保您创建解析器并保存所做的更改。

发送和订阅消息

要让客户端接收消息，它们必须先订阅收件箱。

在查询窗格中，执行 SubscribeToData 订阅：

```
subscription SubscribeToData {
    subscribe(name:"channel") {
        name
        data
    }
}
```

每次调用 publish 变更时，只有在消息发送到 channel 订阅时，订阅者才会收到消息。让我们在查询窗格中试一下。当您的订阅仍在控制台中运行时，打开另一个控制台，并在查询窗格中运行以下请求：

> **Note**
>
> 我们在该示例中使用有效的 JSON 字符串。

```
mutation PublishData {
    publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
        data
        name
    }
}
```

结果类似下面这样：

```
{
  "data": {
    "publish": {
      "data": "{\"msg\":\"hello world!\"}",
      "name": "channel"
```

```
    }
  }
}
```

我们刚刚说明了如何使用本地解析器，即，在不离开 AWS AppSync 服务的情况下发布和接收消息。

教程：组合使用 GraphQL 解析器

GraphQL 架构中的解析器和字段具有 1:1 的关系，具有很高的灵活性。由于数据源是在解析器上独立于架构配置的，因此，您可以通过不同的数据源解析或处理 GraphQL 类型，从而允许您混合使用和匹配架构以最佳方式满足您的需求。

以下场景说明了如何在架构中混合使用和匹配数据源。在开始之前，您应该熟悉如何配置数据源以及 AWS Lambda、Amazon DynamoDB 和 Amazon OpenSearch Service 的解析器。

示例架构

以下架构具有一个名为 `Post` 的类型，其中包含三个 `Query` 和 `Mutation` 操作：

```
type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
}

type Query {
    allPost: [Post]
    getPost(id: ID!): Post
    searchPosts: [Post]
}

type Mutation {
    addPost(
        id: ID!,
        author: String!,
        title: String,
        content: String,
```

```
        url: String
    ): Post
    updatePost(
        id: ID!,
        author: String!,
        title: String,
        content: String,
        url: String,
        ups: Int!,
        downs: Int!,
        expectedVersion: Int!
    ): Post
    deletePost(id: ID!): Post
}
```

在该示例中，总共有 6 个解析器，每个解析器都需要一个数据源。解决该问题的一种方法是，将它们挂接到一个名为 Posts 的 Amazon DynamoDB 表，其中 AllPost 字段运行扫描，searchPosts 字段运行查询（请参阅 DynamoDB 的 JavaScript 解析器函数参考）。不过，您并不仅限于 Amazon DynamoDB；可以使用 Lambda 或 OpenSearch Service 等不同的数据源以满足您的业务要求。

通过解析器更改数据

您可能需要从 AWS AppSync 数据源不直接支持的第三方数据库返回结果。在将数据返回到 API 客户端之前，您可能还必须对数据执行复杂的修改。这可能是由于数据类型格式不正确造成的（例如，客户端上的时间戳差异），也可能是由于处理向后兼容性问题造成的。在这种情况下，将 AWS Lambda 函数作为数据源连接到 AWS AppSync API 是合适的解决方案。出于说明目的，在以下示例中，AWS Lambda 函数处理从第三方数据存储中获取的数据：

```
export const handler = (event, context, callback) => {
    // fetch data
    const result = fetcher()

    // apply complex business logic
    const data = transform(result)

    // return to AppSync
    return data
};
```

这是一个完全有效的 Lambda 函数，可以附加到 GraphQL 架构中的 AllPost 字段，以便返回所有结果的任何查询都可以对顶/踩操作获得随机数字。

DynamoDB 和 OpenSearch Service

对于某些应用程序，您可能会对 DynamoDB 执行变更或简单查找查询，并让后台进程将文档传输到 OpenSearch Service。您可以简单地将 `searchPosts` 解析器附加到 OpenSearch Service 数据源，并使用 GraphQL 查询返回搜索结果（从源自 DynamoDB 的数据）。在应用程序中添加高级搜索操作（例如关键字、模糊字词匹配，甚至地理空间查找）时，这可能是非常强大的。可以通过 ETL 流程完成从 DynamoDB 传输数据的过程，也可以使用 Lambda 从 DynamoDB 进行流式传输。

要开始使用这些特定数据源，请参阅我们的 DynamoDB 和 Lambda 教程。

例如，通过使用以前教程中的架构，以下变更将一个项目添加到 DynamoDB 中：

```
mutation addPost {
  addPost(
    id: 123
    author: "Nadia"
    title: "Our first post!"
    content: "This is our first post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

这会将数据写入到 DynamoDB 中，然后 DynamoDB 通过 Lambda 将数据流式传输到 Amazon OpenSearch Service，您可以使用该服务按不同字段搜索文章。例如，由于数据位于 Amazon OpenSearch Service 中，您可以使用自由格式文本（甚至包含空格）搜索作者或内容字段，如下所示：

```
query searchName{
    searchAuthor(name:"   Nadia   "){
        id
        title
        content
```

```
    }
}

---------- or ----------

query searchContent{
    searchContent(text:"test"){
        id
        title
        content
    }
}
```

由于数据直接写入到 DynamoDB 中，因此，您仍然可以使用 `allPost{...}` 和 `getPost{...}` 查询对表执行高效的列表或项目查找操作。该堆栈将以下示例代码用于 DynamoDB 流：

> **Note**
>
> 该 Python 代码是一个示例，并不适合在生产代码中使用。

```
import boto3
import requests
from requests_aws4auth import AWS4Auth

region = '' # e.g. us-east-1
service = 'es'
credentials = boto3.Session().get_credentials()
awsauth = AWS4Auth(credentials.access_key, credentials.secret_key, region, service,
 session_token=credentials.token)

host = '' # the OpenSearch Service domain, e.g. https://search-mydomain.us-
west-1.es.amazonaws.com
index = 'lambda-index'
datatype = '_doc'
url = host + '/' + index + '/' + datatype + '/'

headers = { "Content-Type": "application/json" }

def handler(event, context):
    count = 0
    for record in event['Records']:
```

```
        # Get the primary key for use as the OpenSearch ID
        id = record['dynamodb']['Keys']['id']['S']

        if record['eventName'] == 'REMOVE':
            r = requests.delete(url + id, auth=awsauth)
        else:
            document = record['dynamodb']['NewImage']
            r = requests.put(url + id, auth=awsauth, json=document, headers=headers)
        count += 1
    return str(count) + ' records processed.'
```

然后，您可以使用 DynamoDB 流将其附加到主键为 `id` 的 DynamoDB 表，并将对 DynamoDB 源的任何更改流式传输到您的 OpenSearch Service 域。有关配置上述功能的更多信息，请参阅 DynamoDB 流文档。

教程：Amazon OpenSearch Service 解析器

AWS AppSync 支持从您在自己的 AWS 账户中预置的域中使用 Amazon OpenSearch Service，但前提是这些域在 VPC 中不存在。预置域后，您可以使用数据源连接这些域，此时，您可以在架构中配置一个解析器以执行 GraphQL 操作（如查询、变更和订阅）。本教程将引导您了解一些常见示例。

有关更多信息，请参阅 OpenSearch 的 JavaScript 解析器函数参考。

创建新的 OpenSearch Service 域

要开始学习本教程，您需要具有一个现有的 OpenSearch Service 域。如果您没有域，可以使用以下示例。请注意，创建 OpenSearch Service 域最多可能需要 15 分钟，然后您才能继续将其与 AWS AppSync 数据源集成在一起。

```
aws cloudformation create-stack --stack-name AppSyncOpenSearch \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/elasticsearch/
ESResolverCFTemplate.yaml \
--parameters ParameterKey=OSDomainName,ParameterValue=ddtestdomain
 ParameterKey=Tier,ParameterValue=development \
--capabilities CAPABILITY_NAMED_IAM
```

可以在您的 AWS 账户的 US-West-2（俄勒冈州）区域中启动以下 AWS CloudFormation 堆栈：

为 OpenSearch Service 配置数据源

在创建 OpenSearch Service 域后，导航到 AWS AppSync GraphQL API 并选择数据源选项卡。选择创建数据源，并为数据源输入一个友好名称，例如“*oss*”。然后，为数据源类型选择 Amazon OpenSearch 域，选择相应的区域，您应该会看到列出了您的 OpenSearch Service 域。在选择该域后，您可以创建一个新角色，AWS AppSync 为其分配相应的角色权限，您也可以选择一个现有角色，该角色具有以下内联策略：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1234234",
            "Effect": "Allow",
            "Action": [
                "es:ESHttpDelete",
                "es:ESHttpHead",
                "es:ESHttpGet",
                "es:ESHttpPost",
                "es:ESHttpPut"
            ],
            "Resource": [
                "arn:aws:es:REGION:ACCOUNTNUMBER:domain/democluster/*"
            ]
        }
    ]
}
```

您还需要为该角色建立与 AWS AppSync 的信任关系：

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

此外，OpenSearch Service 域具有自己的访问策略，您可以通过 Amazon OpenSearch Service 控制台修改该策略。您必须添加一个类似于下面的策略，并具有 OpenSearch Service 域的相应操作和资源。请注意，主体是 AWS AppSync 数据源角色，如果您让 IAM 控制台创建该角色，则可以在所述控制台中找到该角色。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Effect": "Allow",
        "Principal": {
            "AWS": "arn:aws:iam::ACCOUNTNUMBER:role/service-role/
APPSYNC_DATASOURCE_ROLE"
        },
        "Action": [
            "es:ESHttpDelete",
            "es:ESHttpHead",
            "es:ESHttpGet",
            "es:ESHttpPost",
            "es:ESHttpPut"
        ],
        "Resource": "arn:aws:es:REGION:ACCOUNTNUMBER:domain/DOMAIN_NAME/*"
        }
    ]
}
```

连接解析器

数据源现已连接到您的 OpenSearch Service 域，您可以使用解析器将其连接到您的 GraphQL 架构，如以下示例中所示：

```
 type Query {
   getPost(id: ID!): Post
   allPosts: [Post]
 }

 type Mutation {
   addPost(id: ID!, author: String, title: String, url: String, ups: Int, downs: Int,
 content: String): AWSJSON
 }

type Post {
```

```
  id: ID!
  author: String
  title: String
  url: String
  ups: Int
  downs: Int
  content: String
}
```

请注意，有一个用户定义的 Post 类型（具有一个 id 字段）。在以下示例中，我们假设通过一个流程（可以自动完成）将该类型放入您的 OpenSearch Service 域中，这会映射到 /post/_doc 的路径根，其中 post 是索引。从该根路径中，您可以执行单独文档搜索、使用 /id/post* 的通配符搜索或使用 /post/_search 路径的多文档搜索。例如，如果您具有另一个名为 User 的类型，您可以使用名为 user 的新索引对文档编制索引，然后使用 /user/_search 路径执行搜索。

从 AWS AppSync 控制台的架构编辑器中，修改前面的 Posts 架构以包含 searchPosts 查询：

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

保存架构。在解析器窗格中，找到 searchPosts 并选择附加。选择您的 OpenSearch Service 数据源并保存解析器。使用下面的代码片段更新解析器的代码：

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by using an input term
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
 return {
  operation: 'GET',
  path: `/post/_search`,
  params: { body: { from: 0, size: 50 } },
 }
}

/**
```

```
 * Returns the fetched items
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type)
 }
 return ctx.result.hits.hits.map((hit) => hit._source)
}
```

这假设前面的架构的文档已在 OpenSearch Service 中使用 `post` 字段编制索引。如果您以不同方式设置数据结构，则需要相应地进行更新。

修改您的搜索

前面的解析器请求处理程序为所有记录执行简单的查询。假设您想要按某个特定作者进行搜索。此外，假设您希望将该作者指定为 GraphQL 查询中定义的参数。在 AWS AppSync 控制台的架构编辑器中，添加一个 `allPostsByAuthor` 查询：

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

在解析器窗格中，找到 `allPostsByAuthor` 并选择附加。选择 OpenSearch Service 数据源，并使用以下代码：

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
 return {
  operation: 'GET',
  path: '/post/_search',
```

```
  params: {
   body: {
    from: 0,
    size: 50,
    query: { match: { author: ctx.args.author } },
   },
  },
 }
}

/**
 * Returns the fetched items
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type)
 }
 return ctx.result.hits.hits.map((hit) => hit._source)
}
```

请注意，body 用一个针对 author 字段的术语查询填充，它将作为一个参数从客户端进行传递。或者，您可以选择使用预填充的信息，例如标准文本。

将数据添加到 OpenSearch Service

您可能希望将数据添加到您的 OpenSearch Service 域以作为 GraphQL 变更结果。这是一个用于搜索和其他用途的强大机制。由于您可以使用 GraphQL 订阅实时获取数据，因此，它可以作为一种机制向客户端通知 OpenSearch Service 域中的数据更新。

返回到 AWS AppSync 控制台中的架构页面，然后为 addPost() 变更选择附加。再次选择 OpenSearch Service 数据源，并使用以下代码：

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
```

```
 return {
  operation: 'PUT',
  path: `/post/_doc/${ctx.args.id}`,
  params: { body: ctx.args },
 }
}

/**
 * Returns the inserted post
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type)
 }
 return ctx.result
}
```

与之前一样，这是一个说明如何设置数据结构的示例。如果您具有不同的字段名称或索引，则需要更新 path 和 body。该示例还说明了如何在请求处理程序中使用 `context.arguments`（也可以写成 `ctx.args`）。

检索单个文档

最后，如果要在架构中使用 `getPost(id:ID)` 查询以返回单个文档，请在 AWS AppSync 控制台的架构编辑器中找到该查询，然后选择附加。再次选择 OpenSearch Service 数据源，并使用以下代码：

```
import { util } from '@aws-appsync/utils'

/**
 * Searches for documents by `author`
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the request
 */
export function request(ctx) {
 return {
  operation: 'GET',
  path: `/post/_doc/${ctx.args.id}`,
 }
}
```

```
/**
 * Returns the post
 * @param {import('@aws-appsync/utils').Context} ctx the context
 * @returns {*} the result
 */
export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type)
 }
 return ctx.result._source
}
```

执行查询和变更

您现在应该能够对 OpenSearch Service 域执行 GraphQL 操作。导航到 AWS AppSync 控制台的查询选项卡，并添加一个新记录：

```
mutation AddPost {
    addPost (
        id:"12345"
        author: "Fred"
        title: "My first book"
        content: "This will be fun to write!"
        url: "publisher website",
        ups: 100,
        downs:20
       )
}
```

您将会在右侧看到变更结果。同样，您现在可以对您的 OpenSearch Service 域运行 `searchPosts` 查询：

```
query search {
    searchPosts {
        id
        title
        author
        content
    }
}
```

最佳实践

- 应将 OpenSearch Service 用于查询数据，而不是作为主数据库。您可能希望将 OpenSearch Service 与 Amazon DynamoDB 一起使用，如组合使用 GraphQL 解析器中所述。
- 通过允许 AWS AppSync 服务角色访问集群，仅授予您的域的访问权限。
- 您可以通过最低成本的集群先开始小规模开发，然后随着您转向生产阶段，而转至具有高可用性 (HA) 的较大集群。

教程：DynamoDB 事务解析器

AWS AppSync 支持对单个区域中的一个或多个表执行 Amazon DynamoDB 事务操作。支持的操作为 `TransactGetItems` 和 `TransactWriteItems`。通过在 AWS AppSync 中使用这些功能，您可以执行如下任务：

- 在单个查询中传递键列表，并从表中返回结果
- 在单个查询中从一个或多个表中读取记录
- 以全有或全无方式将事务中的记录写入到一个或多个表
- 在满足某些条件时运行事务

权限

与其他解析器一样，您需要在 AWS AppSync 中创建数据源，并创建一个角色或使用现有的角色。由于事务操作需要具有 DynamoDB 表的不同权限，因此，您需要为配置的角色授予读取或写入操作权限：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
```

```
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:region:accountId:table/TABLENAME",
                "arn:aws:dynamodb:region:accountId:table/TABLENAME/*"
            ]
        }
    ]
}
```

> **Note**
>
> 角色与 AWS AppSync 中的数据源相关联，并对数据源调用字段上的解析器。配置为针对 DynamoDB 获取的数据源仅指定一个表，以使配置保持简单。因此，当在单个解析器中针对多个表执行事务操作（这是一项更高级的任务）时，您必须向该数据源的角色授予对将与解析器进行交互的任何表的访问权限。这项操作将在上面 IAM 策略中的 Resource (资源) 字段中执行。针对表的事务调用的配置是在解析器代码中完成的，我们将在下面进行介绍。

数据源

为简便起见，我们将为本教程中使用的所有解析器使用同一个数据源。

我们具有两个名为 savingAccounts 和 checkingAccounts 的表，它们将 `accountNumber` 作为分区键，还有一个 transactionHistory 表，它将 `transactionId` 作为分区键。您可以使用下面的 CLI 命令创建表。确保将 `region` 替换为您的区域。

使用 CLI

```
aws dynamodb create-table --table-name savingAccounts \
  --attribute-definitions AttributeName=accountNumber,AttributeType=S \
  --key-schema AttributeName=accountNumber,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region

aws dynamodb create-table --table-name checkingAccounts \
  --attribute-definitions AttributeName=accountNumber,AttributeType=S \
  --key-schema AttributeName=accountNumber,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region

aws dynamodb create-table --table-name transactionHistory \
```

```
  --attribute-definitions AttributeName=transactionId,AttributeType=S \
  --key-schema AttributeName=transactionId,KeyType=HASH \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5 \
  --table-class STANDARD --region region
```

在 AWS AppSync 控制台的数据源中，创建一个新的 DynamoDB 数据源，并将其命名为 TransactTutorial。选择 savingAccounts 以作为表（尽管在使用事务时具体的表并不重要）。选择创建新角色和数据源。您可以检查数据源配置以查看生成的角色的名称。在 IAM 控制台中，您可以添加一个允许数据源与所有表交互的内联策略。

将 `region` 和 `accountID` 替换为您的区域和账户 ID：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:region:accountId:table/savingAccounts",
                "arn:aws:dynamodb:region:accountId:table/savingAccounts/*",
                "arn:aws:dynamodb:region:accountId:table/checkingAccounts",
                "arn:aws:dynamodb:region:accountId:table/checkingAccounts/*",
                "arn:aws:dynamodb:region:accountId:table/transactionHistory",
                "arn:aws:dynamodb:region:accountId:table/transactionHistory/*"
            ]
        }
    ]
}
```

事务

对于本示例，上下文是一个典型的银行交易，我们将使用 `TransactWriteItems` 来执行以下操作：

- 从储蓄账户转账到支票账户

- 为每个交易生成新的交易记录

然后我们将使用 TransactGetItems 从储蓄账户和支票账户中检索详细信息。

我们按如下所示定义我们的 GraphQL 架构：

```
type SavingAccount {
    accountNumber: String!
    username: String
    balance: Float
}

type CheckingAccount {
    accountNumber: String!
    username: String
    balance: Float
}

type TransactionHistory {
    transactionId: ID!
    from: String
    to: String
    amount: Float
}

type TransactionResult {
    savingAccounts: [SavingAccount]
    checkingAccounts: [CheckingAccount]
    transactionHistory: [TransactionHistory]
}

input SavingAccountInput {
    accountNumber: String!
    username: String
    balance: Float
}

input CheckingAccountInput {
    accountNumber: String!
    username: String
    balance: Float
}
```

```
input TransactionInput {
    savingAccountNumber: String!
    checkingAccountNumber: String!
    amount: Float!
}

type Query {
    getAccounts(savingAccountNumbers: [String], checkingAccountNumbers: [String]):
 TransactionResult
}

type Mutation {
    populateAccounts(savingAccounts: [SavingAccountInput], checkingAccounts:
 [CheckingAccountInput]): TransactionResult
    transferMoney(transactions: [TransactionInput]): TransactionResult
}
```

TransactWriteItems - 填充账户

为了在账户之间转账，我们需要用详细信息填充表格。我们将使用 GraphQL 操作 `Mutation.populateAccounts` 来实现此目的。

在“架构”部分中，单击 `Mutation.populateAccounts` 操作旁边的附加。选择 `TransactTutorial` 数据源，然后选择创建。

现在使用以下代码：

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { savingAccounts, checkingAccounts } = ctx.args

 const savings = savingAccounts.map(({ accountNumber, ...rest }) => {
  return {
   table: 'savingAccounts',
   operation: 'PutItem',
   key: util.dynamodb.toMapValues({ accountNumber }),
   attributeValues: util.dynamodb.toMapValues(rest),
  }
 })

 const checkings = checkingAccounts.map(({ accountNumber, ...rest }) => {
  return {
```

```
    table: 'checkingAccounts',
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ accountNumber }),
    attributeValues: util.dynamodb.toMapValues(rest),
   }
 })
 return {
  version: '2018-05-29',
  operation: 'TransactWriteItems',
  transactItems: [...savings, ...checkings],
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
 }
 const { savingAccounts: sInput, checkingAccounts: cInput } = ctx.args
 const keys = ctx.result.keys
 const savingAccounts = sInput.map((_, i) => keys[i])
 const sLength = sInput.length
 const checkingAccounts = cInput.map((_, i) => keys[sLength + i])
 return { savingAccounts, checkingAccounts }
}
```

保存解析器，并导航到 AWS AppSync 控制台的查询部分以填充账户。

执行以下变更：

```
mutation populateAccounts {
  populateAccounts (
    savingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 100},
      {accountNumber: "2", username: "Amy", balance: 90},
      {accountNumber: "3", username: "Lily", balance: 80},
    ]
    checkingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 70},
      {accountNumber: "2", username: "Amy", balance: 60},
      {accountNumber: "3", username: "Lily", balance: 50},
    ]) {
    savingAccounts {
      accountNumber
    }
```

```
    checkingAccounts {
      accountNumber
    }
  }
}
```

我们在一个变更中填充了三个储蓄账户和三个支票账户。

使用 DynamoDB 控制台验证是否在 savingAccounts 和 checkingAccounts 表中显示数据。

TransactWriteItems - 转账

使用以下代码将一个解析器附加到 `transferMoney` 变更。对于每笔转账，我们都需要支票账户和储蓄账户具有成功修饰符，并且需要跟踪交易中的转账。

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const transactions = ctx.args.transactions

 const savings = []
 const checkings = []
 const history = []
 transactions.forEach((t) => {
  const { savingAccountNumber, checkingAccountNumber, amount } = t
  savings.push({
   table: 'savingAccounts',
   operation: 'UpdateItem',
   key: util.dynamodb.toMapValues({ accountNumber: savingAccountNumber }),
   update: {
    expression: 'SET balance = balance - :amount',
    expressionValues: util.dynamodb.toMapValues({ ':amount': amount }),
   },
  })
  checkings.push({
   table: 'checkingAccounts',
   operation: 'UpdateItem',
   key: util.dynamodb.toMapValues({ accountNumber: checkingAccountNumber }),
   update: {
    expression: 'SET balance = balance + :amount',
    expressionValues: util.dynamodb.toMapValues({ ':amount': amount }),
   },
  })
```

```
  history.push({
   table: 'transactionHistory',
   operation: 'PutItem',
   key: util.dynamodb.toMapValues({ transactionId: util.autoId() }),
   attributeValues: util.dynamodb.toMapValues({
    from: savingAccountNumber,
    to: checkingAccountNumber,
    amount,
   }),
  })
 })

 return {
  version: '2018-05-29',
  operation: 'TransactWriteItems',
  transactItems: [...savings, ...checkings, ...history],
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
 }
 const tInput = ctx.args.transactions
 const tLength = tInput.length
 const keys = ctx.result.keys
 const savingAccounts = tInput.map((_, i) => keys[tLength * 0 + i])
 const checkingAccounts = tInput.map((_, i) => keys[tLength * 1 + i])
 const transactionHistory = tInput.map((_, i) => keys[tLength * 2 + i])
 return { savingAccounts, checkingAccounts, transactionHistory }
}
```

现在，导航到 AWS AppSync 控制台的查询部分并执行 transferMoney 变更，如下所示：

```
mutation write {
  transferMoney(
    transactions: [
      {savingAccountNumber: "1", checkingAccountNumber: "1", amount: 7.5},
      {savingAccountNumber: "2", checkingAccountNumber: "2", amount: 6.0},
      {savingAccountNumber: "3", checkingAccountNumber: "3", amount: 3.3}
    ]) {
    savingAccounts {
      accountNumber
```

```
    }
    checkingAccounts {
      accountNumber
    }
    transactionHistory {
      transactionId
    }
  }
}
```

我们在一个变更中发送了三笔银行交易。使用 DynamoDB 控制台验证是否在 savingAccounts、checkingAccounts 和 transactionHistory 表中显示数据。

TransactGetItems - 检索账户

为了在单个事务请求中从储蓄和支票账户中检索详细信息，我们将一个解析器附加到架构上的 `Query.getAccounts` GraphQL 操作。选择附加，选择在教程开始时创建的相同 `TransactTutorial` 数据源。使用以下代码：

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { savingAccountNumbers, checkingAccountNumbers } = ctx.args

 const savings = savingAccountNumbers.map((accountNumber) => {
  return { table: 'savingAccounts', key: util.dynamodb.toMapValues({ accountNumber }) }
 })
 const checkings = checkingAccountNumbers.map((accountNumber) => {
  return { table: 'checkingAccounts', key:
 util.dynamodb.toMapValues({ accountNumber }) }
 })
 return {
  version: '2018-05-29',
  operation: 'TransactGetItems',
  transactItems: [...savings, ...checkings],
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.error(ctx.error.message, ctx.error.type, null, ctx.result.cancellationReasons)
 }
```

```
  const { savingAccountNumbers: sInput, checkingAccountNumbers: cInput } = ctx.args
  const items = ctx.result.items
  const savingAccounts = sInput.map((_, i) => items[i])
  const sLength = sInput.length
  const checkingAccounts = cInput.map((_, i) => items[sLength + i])
  return { savingAccounts, checkingAccounts }
}
```

保存解析器，并导航到 AWS AppSync 控制台的查询部分。为了检索储蓄和支票账户，执行以下查询：

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
      username
      balance
    }
  }
}
```

我们已成功说明了如何通过 AWS AppSync 使用 DynamoDB 事务。

教程：DynamoDB 批处理解析器

AWS AppSync 支持对单个区域中的一个或多个表执行 Amazon DynamoDB 批处理操作。支持的操作为 `BatchGetItem`、`BatchPutItem` 和 `BatchDeleteItem`。通过在 AWS AppSync 中使用这些功能，您可以执行如下任务：

- 在单个查询中传递键列表，并从表中返回结果
- 在单个查询中从一个或多个表中读取记录
- 将记录批量写入到一个或多个表

- 在可能存在关系的多个表中有条件地写入或删除记录

AWS AppSync 中的批处理操作与非批处理操作有两个主要区别：

- 数据源角色必须具有解析器访问的所有表的权限。
- 解析器的表规范是请求对象的一部分。

单个表批处理

首先，我们创建一个新的 GraphQL API。在 AWS AppSync 控制台中，选择创建 API、GraphQL API 和从头开始设计。将您的 API 命名为 `BatchTutorial API`，选择下一步，在指定 GraphQL 资源步骤中选择稍后创建 GraphQL 资源，然后单击下一步。检查您的详细信息并创建 API。转到架构页面并粘贴以下架构，请注意，对于查询，我们将传入一个 ID 列表：

```
type Post {
    id: ID!
    title: String
}

input PostInput {
    id: ID!
    title: String
}

type Query {
    batchGet(ids: [ID]): [Post]
}

type Mutation {
    batchAdd(posts: [PostInput]): [Post]
    batchDelete(ids: [ID]): [Post]
}
```

保存您的架构，并选择页面顶部的创建资源。选择使用现有的类型，并选择 `Post` 类型。将您的表命名为 `Posts`。确保主键设置为 `id`，取消选择自动生成 GraphQL（您将提供自己的代码），然后选择创建。首先，AWS AppSync 创建一个新的 DynamoDB 表以及一个使用相应角色连接到该表的数据源。不过，您仍然需要为该角色添加一些权限。转到数据源页面，并选择新的数据源。在选择现有角色下面，您会注意到已为该表自动创建了一个角色。记下该角色（应类似于 `appsync-ds-ddb-aaabbbcccddd-Posts`），然后转到 IAM 控制台 (https://console.aws.amazon.com/iam/)。在 IAM

控制台中，选择角色，然后从该表中选择您的角色。在您的角色中，在权限策略下面，单击策略旁边的“+”（名称应与角色名称相似）。在显示该策略时，选择折叠菜单顶部的编辑。您需要为您的策略添加批处理权限，具体来说是 dynamodb:BatchGetItem 和 dynamodb:BatchWriteItem。代码片段如下所示：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem",
                "dynamodb:BatchWriteItem",
                "dynamodb:BatchGetItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:…",
                "arn:aws:dynamodb:…"
            ]
        }
    ]
}
```

选择下一步，然后选择保存更改。您的策略现在应该允许进行批处理。

返回到 AWS AppSync 控制台，转到架构页面并选择 Mutation.batchAdd 字段旁边的附加。将 Posts 表作为数据源以创建解析器。在代码编辑器中，将处理程序替换为下面的代码片段。该代码片段自动获取 GraphQL input PostInput 类型中的每个项目，并构建 BatchPutItem 操作所需的映射：

```
import { util } from "@aws-appsync/utils";

export function request(ctx) {
  return {
    operation: "BatchPutItem",
    tables: {
      Posts: ctx.args.posts.map((post) => util.dynamodb.toMapValues(post)),
```

```
    },
  };
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
}
```

导航到 AWS AppSync 控制台的查询页面，并运行以下 batchAdd 变更：

```
mutation add {
    batchAdd(posts:[{
            id: 1 title: "Running in the Park"},{
            id: 2 title: "Playing fetch"
        }]){
            id
            title
    }
}
```

您应该会看到在屏幕上输出的结果；可以在 DynamoDB 控制台中扫描写入到 Posts 表的值以验证这一点。

接下来，重复附加解析器的过程，但对于 Query.batchGet 字段，将 Posts 表作为数据源。将处理程序替换为以下代码。这会自动接受 GraphQL ids:[] 类型的每个项目并构建 BatchGetItem 操作所需的一个映射：

```
import { util } from "@aws-appsync/utils";

export function request(ctx) {
  return {
    operation: "BatchGetItem",
    tables: {
      Posts: {
        keys: ctx.args.ids.map((id) => util.dynamodb.toMapValues({ id })),
        consistentRead: true,
      },
    },
  };
```

```
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
  return ctx.result.data.Posts;
}
```

现在，返回到 AWS AppSync 控制台的查询页面，并运行以下 batchGet 查询：

```
query get {
    batchGet(ids:[1,2,3]){
        id
        title
    }
}
```

这应返回您早前添加的两个 id 值的结果。请注意，对于值为 3 的 id，将返回 null 值。这是因为您的 Posts 表中还没有具有该值的记录。另请注意，AWS AppSync 使用与传递给查询的键相同的顺序返回结果，这是 AWS AppSync 代表您执行的一项额外功能。因此，如果切换到 batchGet(ids:[1,3,2])，您会看到顺序发生了变化。您还将了解哪个 id 返回了 null 值。

最后，将另一个解析器附加到 Mutation.batchDelete 字段，并将 Posts 表作为数据源。将处理程序替换为以下代码。这会自动接受 GraphQL ids:[] 类型的每个项目并构建 BatchGetItem 操作所需的一个映射：

```
import { util } from "@aws-appsync/utils";

export function request(ctx) {
  return {
    operation: "BatchDeleteItem",
    tables: {
      Posts: ctx.args.ids.map((id) => util.dynamodb.toMapValues({ id })),
    },
  };
}

export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type);
  }
```

```
  return ctx.result.data.Posts;
}
```

现在，返回到 AWS AppSync 控制台的查询页面，并运行以下 batchDelete 变更：

```
mutation delete {
    batchDelete(ids:[1,2]){ id }
}
```

现在应删除 id 为 1 和 2 的记录。如果您更早重新运行 batchGet() 查询，则应返回 null。

多个表批处理

AWS AppSync 还允许您在多个表中执行批处理操作。我们来构建更复杂的应用程序。想象一下，我们构建一个宠物健康应用程序，其中传感器报告宠物的位置和体温。传感器由电池供电，并每隔几分钟尝试连接到网络。在传感器建立连接时，它将读数发送到我们的 AWS AppSync API。然后，触发器分析数据，这样，就可以向宠物主人显示控制面板。我们重点关注如何表示传感器与后端数据存储之间的交互。

在 AWS AppSync 控制台中，选择创建 API、GraphQL API 和从头开始设计。将您的 API 命名为 MultiBatchTutorial API，选择下一步，在指定 GraphQL 资源步骤中选择稍后创建 GraphQL 资源，然后单击下一步。检查您的详细信息并创建 API。转到架构页面，并粘贴和保存以下架构：

```
type Mutation {
    # Register a batch of readings
    recordReadings(tempReadings: [TemperatureReadingInput], locReadings:
 [LocationReadingInput]): RecordResult
    # Delete a batch of readings
    deleteReadings(tempReadings: [TemperatureReadingInput], locReadings:
 [LocationReadingInput]): RecordResult
}

type Query {
    # Retrieve all possible readings recorded by a sensor at a specific time
    getReadings(sensorId: ID!, timestamp: String!): [SensorReading]
}

type RecordResult {
    temperatureReadings: [TemperatureReading]
    locationReadings: [LocationReading]
}
```

```
interface SensorReading {
    sensorId: ID!
    timestamp: String!
}

# Sensor reading representing the sensor temperature (in Fahrenheit)
type TemperatureReading implements SensorReading {
    sensorId: ID!
    timestamp: String!
    value: Float
}

# Sensor reading representing the sensor location (lat,long)
type LocationReading implements SensorReading {
    sensorId: ID!
    timestamp: String!
    lat: Float
    long: Float
}

input TemperatureReadingInput {
    sensorId: ID!
    timestamp: String
    value: Float
}

input LocationReadingInput {
    sensorId: ID!
    timestamp: String
    lat: Float
    long: Float
}
```

我们需要创建两个 DynamoDB 表：

- `locationReadings` 存储传感器位置读数。
- `temperatureReadings` 存储传感器温度读数。

这两个表具有相同的主键结构：将 `sensorId`（String）作为分区键，并将 `timestamp`（String）作为排序键。

选择页面顶部的创建资源。选择使用现有的类型，并选择 locationReadings 类型。将您的表命名为 locationReadings。确保将主键设置为 sensorId，并将排序键设置为 timestamp。取消选择自动生成 GraphQL（您将提供自己的代码），然后选择创建。为 temperatureReadings 重复该过程，并将 temperatureReadings 作为类型和表名称。使用与上面相同的键。

您的新表将包含自动生成的角色。您仍然需要为这些角色添加一些权限。转到数据源页面并选择 locationReadings。在选择现有角色下面，您可以看到该角色。记下该角色（应类似于 appsync-ds-ddb-aaabbbcccddd-locationReadings），然后转到 IAM 控制台 (https://console.aws.amazon.com/iam/)。在 IAM 控制台中，选择角色，然后从该表中选择您的角色。在您的角色中，在权限策略下面，单击策略旁边的“+”（名称应与角色名称相似）。在显示该策略时，选择折叠菜单顶部的编辑。您需要为该策略添加权限。代码片段如下所示：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem",
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:region:account:table/locationReadings",
                "arn:aws:dynamodb:region:account:table/locationReadings/*",
                "arn:aws:dynamodb:region:account:table/temperatureReadings",
                "arn:aws:dynamodb:region:account:table/temperatureReadings/*"
            ]
        }
    ]
}
```

选择下一步，然后选择保存更改。使用上面相同的策略片段，为 temperatureReadings 数据源重复该过程。

BatchPutItem - 记录传感器读数

我们的传感器需要能够在连接到 Internet 后立即发送其读数。GraphQL 字段 `Mutation.recordReadings` 是传感器将用来执行上述操作的 API。我们需要在该字段中添加一个解析器。

在 AWS AppSync 控制台的架构页面中，选择 `Mutation.recordReadings` 字段旁边的附加。在下一个屏幕上，将 `locationReadings` 表作为数据源以创建解析器。

在创建解析器后，在编辑器中将处理程序替换为以下代码。我们可以通过 `BatchPutItem` 操作指定多个表：

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { locReadings, tempReadings } = ctx.args
 const locationReadings = locReadings.map((loc) => util.dynamodb.toMapValues(loc))
 const temperatureReadings = tempReadings.map((tmp) => util.dynamodb.toMapValues(tmp))

 return {
  operation: 'BatchPutItem',
  tables: {
   locationReadings,
   temperatureReadings,
  },
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.appendError(ctx.error.message, ctx.error.type)
 }
 return ctx.result.data
}
```

使用批处理操作，可能会从调用中同时返回错误和结果。在这种情况下，我们可以自主执行一些额外的错误处理。

> **Note**
>
> 使用 `utils.appendError()` 与 `util.error()` 类似，主要区别在于，它不会中断请求或响应处理程序评估。相反，它指示字段存在错误，但允许评估处理程序，从而将数据返回到调用方。在您的应用程序需要返回部分结果时，我们建议您使用 `utils.appendError()`。

保存解析器，并导航到 AWS AppSync 控制台中的查询页面。我们现在可以发送一些传感器读数。

执行以下变更：

```
mutation sendReadings {
  recordReadings(
    tempReadings: [
      {sensorId: 1, value: 85.5, timestamp: "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, value: 85.7, timestamp: "2018-02-01T17:21:06.000+08:00"},
      {sensorId: 1, value: 85.8, timestamp: "2018-02-01T17:21:07.000+08:00"},
      {sensorId: 1, value: 84.2, timestamp: "2018-02-01T17:21:08.000+08:00"},
      {sensorId: 1, value: 81.5, timestamp: "2018-02-01T17:21:09.000+08:00"}
    ]
    locReadings: [
      {sensorId: 1, lat: 47.615063, long: -122.333551, timestamp:
 "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, lat: 47.615163, long: -122.333552, timestamp:
 "2018-02-01T17:21:06.000+08:00"},
      {sensorId: 1, lat: 47.615263, long: -122.333553, timestamp:
 "2018-02-01T17:21:07.000+08:00"},
      {sensorId: 1, lat: 47.615363, long: -122.333554, timestamp:
 "2018-02-01T17:21:08.000+08:00"},
      {sensorId: 1, lat: 47.615463, long: -122.333555, timestamp:
 "2018-02-01T17:21:09.000+08:00"}
    ]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
```

```
    }
  }
}
```

我们在一个变更中发送了 10 个传感器读数，这些读数拆分到两个表中。使用 DynamoDB 控制台验证是否在 `locationReadings` 和 `temperatureReadings` 表中显示数据。

BatchDeleteItem - 删除传感器读数

同样，我们还需要能够批量删除传感器读数。我们使用 `Mutation.deleteReadings` GraphQL 字段来实现此目的。在 AWS AppSync 控制台的架构页面中，选择 `Mutation.deleteReadings` 字段旁边的附加。在下一个屏幕上，将 `locationReadings` 表作为数据源以创建解析器。

在创建解析器后，在代码编辑器中将处理程序替换为下面的代码片段。在该解析器中，我们使用帮助程序函数映射器，该映射器从提供的输入中提取 `sensorId` 和 `timestamp`。

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 const { locReadings, tempReadings } = ctx.args
 const mapper = ({ sensorId, timestamp }) => util.dynamodb.toMapValues({ sensorId,
 timestamp })

 return {
  operation: 'BatchDeleteItem',
  tables: {
   locationReadings: locReadings.map(mapper),
   temperatureReadings: tempReadings.map(mapper),
  },
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.appendError(ctx.error.message, ctx.error.type)
 }
 return ctx.result.data
}
```

保存解析器，并导航到 AWS AppSync 控制台中的查询页面。现在，让我们删除几个传感器读数。

执行以下变更：

```
mutation deleteReadings {
  # Let's delete the first two readings we recorded
  deleteReadings(
    tempReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]
    locReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}
```

> **Note**
>
> 与 DeleteItem 操作相反，响应中不会返回完全删除的项目。只返回传递的键。要了解更多信息，请参阅 DynamoDB 的 JavaScript 解析器函数参考中的 BatchDeleteItem。

通过 DynamoDB 控制台验证是否从 locationReadings 和 temperatureReadings 表中删除了这两个读数。

BatchGetItem - 检索读数

我们的应用程序的另一个常见操作是，检索传感器在特定时间点的读数。我们将解析器附加到架构上的 Query.getReadings GraphQL 字段。在 AWS AppSync 控制台的架构页面中，选择 Query.getReadings 字段旁边的附加。在下一个屏幕上，将 locationReadings 表作为数据源以创建解析器。

让我们使用以下代码：

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
```

```
 const keys = [util.dynamodb.toMapValues(ctx.args)]
 const consistentRead = true
 return {
  operation: 'BatchGetItem',
  tables: {
   locationReadings: { keys, consistentRead },
   temperatureReadings: { keys, consistentRead },
  },
 }
}

export function response(ctx) {
 if (ctx.error) {
  util.appendError(ctx.error.message, ctx.error.type)
 }
 const { locationReadings: locs, temperatureReadings: temps } = ctx.result.data

 return [
  ...locs.map((l) => ({ ...l, __typename: 'LocationReading' })),
  ...temps.map((t) => ({ ...t, __typename: 'TemperatureReading' })),
 ]
}
```

保存解析器，并导航到 AWS AppSync 控制台中的查询页面。现在，让我们检索传感器读数。

执行以下查询：

```
query getReadingsForSensorAndTime {
  # Let's retrieve the very first two readings
  getReadings(sensorId: 1, timestamp: "2018-02-01T17:21:06.000+08:00") {
    sensorId
    timestamp
    ...on TemperatureReading {
      value
    }
    ...on LocationReading {
      lat
      long
    }
  }
}
```

我们已成功说明了如何通过 AWS AppSync 使用 DynamoDB 批处理操作。

错误处理

在 AWS AppSync 中，数据源操作有时可能会返回部分结果。部分结果是一个术语，我们用它来表示操作的输出中包含某些数据和一个错误。由于错误处理本质上是应用程序特定的，因此，AWS AppSync 允许您在响应处理程序中处理错误。上下文中的解析器调用错误（如果有）为 `ctx.error`。调用错误始终包含一条消息和一个类型，可作为属性 `ctx.error.message` 和 `ctx.error.type` 进行访问。在响应处理程序中，您可以使用三种方法处理部分结果：

1. 仅返回数据以忽略调用错误。
2. 停止处理程序评估以引发错误（使用 `util.error(...)`），这不会返回任何数据。
3. 附加一个错误（使用 `util.appendError(...)`）并且也返回数据。

让我们通过 DynamoDB 批处理操作分别说明上述三点。

DynamoDB 批处理操作

借助 DynamoDB 批处理操作，批处理可能会部分完成。也就是说，某些请求的项目或键未得到处理。如果 AWS AppSync 无法完成批处理，则会在上下文中设置未处理的项目和调用错误。

我们将使用本教程前一部分中 `Query.getReadings` 操作的 `BatchGetItem` 字段配置来实施错误处理。这一次，我们假定在执行 `Query.getReadings` 字段时，`temperatureReadings` DynamoDB 表耗尽了预置的吞吐量。在 AWS AppSync 第二次尝试处理批次中的剩余元素时，DynamoDB 引发了 `ProvisionedThroughputExceededException`。

以下 JSON 表示在 DynamoDB 批处理调用之后但在调用响应处理程序之前的序列化上下文：

```
{
  "arguments": {
    "sensorId": "1",
    "timestamp": "2018-02-01T17:21:05.000+08:00"
  },
  "source": null,
  "result": {
    "data": {
      "temperatureReadings": [
        null
      ],
      "locationReadings": [
        {
          "lat": 47.615063,
```

```
          "long": -122.333551,
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ]
    },
    "unprocessedKeys": {
      "temperatureReadings": [
        {
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ],
      "locationReadings": []
    }
  },
  "error": {
    "type": "DynamoDB:ProvisionedThroughputExceededException",
    "message": "You exceeded your maximum allowed provisioned throughput for a table or
 for one or more global secondary indexes. (...)"
  },
  "outErrors": []
}
```

关于上下文需要注意的几点：

- AWS AppSync 在上下文中的 `ctx.error` 处设置了调用错误，并且错误类型设置为 `DynamoDB:ProvisionedThroughputExceededException`。
- 即使存在错误，也会在 `ctx.result.data` 中为每个表映射结果。
- 在 `ctx.result.data.unprocessedKeys` 中提供了未处理的键。此处，由于表吞吐量不足，AWS AppSync 无法检索具有 (sensorId:1, timestamp:2018-02-01T17:21:05.000+08:00) 键的项目。

> **Note**
>
> 对于 `BatchPutItem`，它是 `ctx.result.data.unprocessedItems`。对于 `BatchDeleteItem`，它是 `ctx.result.data.unprocessedKeys`。

我们通过三种不同方式处理此错误。

1. 承受调用错误

返回数据而不处理调用错误：这会有效地承受此错误，同时使给定 GraphQL 字段的结果始终成功。

我们编写的代码是熟悉的，并且仅关注结果数据。

响应处理程序

```
export function response(ctx) {
  return ctx.result.data
}
```

GraphQL 响应

```
{
  "data": {
    "getReadings": [
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "lat": 47.615063,
        "long": -122.333551
      },
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  }
}
```

将不向错误响应中添加任何错误，因为只对数据执行了操作。

2. 引发错误以中止执行响应处理程序

从客户端角度看，在应将部分失败视为完全失败时，您可以中止执行响应处理程序以防止返回数据。`util.error(...)` 实用程序方法实现完全此行为。

响应处理程序代码

```
export function response(ctx) {
```

```
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type, null,
 ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}
```

GraphQL 响应

```
{
  "data": {
    "getReadings": null
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
        }
      ],
      "message": "You exceeded your maximum allowed provisioned throughput for a table
 or for one or more global secondary indexes. (...)"
    }
  ]
}
```

即使可能已从 DynamoDB 批处理操作返回了一些结果，我们也选择引发错误，这样，getReadings GraphQL 字段为 Null，并且此错误已添加到 GraphQL 响应的错误数据块中。

3. 追加错误以返回数据和错误

在某些情况下，为了提供更好的用户体验，应用程序可以返回部分结果并向其客户端通知未处理的项目。客户端可以决定是实施重试，还是将错误翻译出来并返回给最终用户。`util.appendError(...)` 是一种实现该行为的实用程序方法，它让应用程序设计者在上下文中附加错误，而不干扰响应处理程序评估。在评估响应处理程序后，AWS AppSync 将任何上下文错误附加到 GraphQL 响应的错误块以处理它们。

响应处理程序代码

```
export function response(ctx) {
  if (ctx.error) {
    util.appendError(ctx.error.message, ctx.error.type, null,
 ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}
```

我们在 GraphQL 响应的错误块中转发了调用错误和 `unprocessedKeys` 元素。`getReadings` 字段也从 `locationReadings` 表中返回部分数据，如下面的响应中所示。

GraphQL 响应

```
{
  "data": {
    "getReadings": [
      null,
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
```

```
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
        }
      ],
      "message": "You exceeded your maximum allowed provisioned throughput for a table
 or for one or more global secondary indexes. (...)"
    }
  ]
}
```

教程：HTTP 解析器

除了使用任意 HTTP 终端节点解析 GraphQL 字段以外，AWS AppSync 还允许您使用支持的数据源（即 AWS Lambda、Amazon DynamoDB、Amazon OpenSearch Service 或 Amazon Aurora）执行各种操作。在您的 HTTP 终端节点可用后，您可以使用数据源连接它们。然后，您可以在架构中配置一个解析器以执行 GraphQL 操作（如查询、变更和订阅）。本教程将引导您了解一些常见示例。

在本教程中，您在 AWS AppSync GraphQL 终端节点中使用 REST API（是使用 Amazon API Gateway 和 Lambda 创建的）。

创建 REST API

您可以使用以下 AWS CloudFormation 模板来设置适用于本教程的 REST 终端节点：

AWS CloudFormation 堆栈将执行以下步骤：

1. 设置 Lambda 函数，其中包含您的微服务的业务逻辑。
2. 使用以下终端节点/方法/内容类型组合设置一个 API Gateway REST API：

API 资源路径	HTTP 方法	支持的内容类型
/v1/users	POST	application/json
/v1/users	GET	application/json
/v1/users/1	GET	application/json
/v1/users/1	PUT	application/json
/v1/users/1	删除	application/json

创建您的 GraphQL API

要在 AWS AppSync 中创建 GraphQL API，请执行以下操作：

1. 打开 AWS AppSync 控制台，然后选择创建 API。
2. 选择 GraphQL API，然后选择从头开始设计。选择下一步。
3. 对于 API 名称，请键入 `UserData`。选择下一步。
4. 选择 `Create GraphQL resources later`。选择下一步。
5. 检查您的输入，然后选择创建 API。

AWS AppSync 控制台使用 API 密钥身份验证模式为您创建新的 GraphQL API。您可以使用控制台进一步配置 GraphQL API 并运行请求。

创建 GraphQL 架构

现在，您有一个 GraphQL API，让我们创建一个 GraphQL 架构。在 AWS AppSync 控制台的架构编辑器中，使用以下代码片段：

```
type Mutation {
    addUser(userInput: UserInput!): User
    deleteUser(id: ID!): User
}

type Query {
    getUser(id: ID): User
```

```
    listUser: [User!]!
}

type User {
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
}

input UserInput {
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
}
```

配置您的 HTTP 数据源

要配置 HTTP 数据源，请执行以下操作：

1. 在 AWS AppSync GraphQL API 的数据源页面中，选择创建数据源。
2. 输入数据源的名称，例如 `HTTP_Example`。
3. 在数据源类型中，选择 HTTP 终端节点。
4. 将终端节点设置为教程开始时创建的 API Gateway 终端节点。如果您导航到 Lambda 控制台并在应用程序下面找到您的应用程序，则可以找到堆栈生成的终端节点。在应用程序的设置中，您应该会看到一个 API 终端节点，它是您在 AWS AppSync 中的终端节点。确保不要将阶段名称作为终端节点的一部分包含在内。例如，如果您的终端节点是 `https://aaabbbcccd.execute-api.us-east-1.amazonaws.com/v1`，您将键入 `https://aaabbbcccd.execute-api.us-east-1.amazonaws.com`。

> **Note**
>
> 目前，AWS AppSync 仅支持公有终端节点。

有关 AWS AppSync 服务识别的认证机构的更多信息，请参阅 AWS AppSync 识别的 HTTPS 终端节点证书颁发机构 (CA)。

配置解析器

在该步骤中，您将 HTTP 数据源连接到 getUser 和 addUser 查询。

要设置 getUser 解析器，请执行以下操作：

1. 在 AWS AppSync GraphQL API 中，选择架构选项卡。
2. 在架构编辑器右侧的解析器窗格中的 Query 类型下面，找到 getUser 字段并选择附加。
3. 将解析器类型保留为 Unit，并将运行时保留为 APPSYNC_JS。
4. 在数据源名称中，选择您以前创建的 HTTP 终端节点。
5. 选择创建。
6. 在解析器代码编辑器中，添加以下代码片段以作为您的请求处理程序：

```
import { util } from '@aws-appsync/utils'

export function request(ctx) {
 return {
  version: '2018-05-29',
  method: 'GET',
  params: {
   headers: {
    'Content-Type': 'application/json',
   },
  },
  resourcePath: `/v1/users/${ctx.args.id}`,
 }
}
```

7. 添加以下代码片段以作为您的响应处理程序：

```
export function response(ctx) {
 const { statusCode, body } = ctx.result
 // if response is 200, return the response
 if (statusCode === 200) {
  return JSON.parse(body)
```

```
 }
 // if response is not 200, append the response to error block.
 util.appendError(body, statusCode)
}
```

8. 选择 Query (查询) 选项卡，然后运行以下查询：

```
query GetUser{
    getUser(id:1){
        id
        username
    }
}
```

此查询应返回以下响应：

```
{
    "data": {
        "getUser": {
            "id": "1",
            "username": "nadia"
        }
    }
}
```

要设置 addUser 解析器，请执行以下操作：

1. 选择架构选项卡。
2. 在架构编辑器右侧的解析器窗格中的 Query 类型下面，找到 addUser 字段并选择附加。
3. 将解析器类型保留为 Unit，并将运行时保留为 APPSYNC_JS。
4. 在数据源名称中，选择您以前创建的 HTTP 终端节点。
5. 选择创建。
6. 在解析器代码编辑器中，添加以下代码片段以作为您的请求处理程序：

```
export function request(ctx) {
    return {
        "version": "2018-05-29",
        "method": "POST",
        "resourcePath": "/v1/users",
```

```
        "params":{
            "headers":{
                "Content-Type": "application/json"
            },
        "body": ctx.args.userInput
        }
    }
}
```

7. 添加以下代码片段以作为您的响应处理程序：

```
export function response(ctx) {
    if(ctx.error) {
        return util.error(ctx.error.message, ctx.error.type)
    }
    if (ctx.result.statusCode == 200) {
        return ctx.result.body
    } else {
        return util.appendError(ctx.result.body, "ctx.result.statusCode")
    }
}
```

8. 选择 Query (查询) 选项卡，然后运行以下查询：

```
mutation addUser{
    addUser(userInput:{
        id:"2",
        username:"shaggy"
    }){
        id
        username
    }
}
```

如果再次运行 `getUser` 查询，它应该会返回以下响应：

```
{
    "data": {
        "getUser": {
        "id": "2",
        "username": "shaggy"
        }
    }
```

```
}
```

调用 AWS 服务

您可以使用 HTTP 解析器为 AWS 服务设置 GraphQL API 接口。发送到 AWS 的 HTTP 请求必须使用签名版本 4 流程进行签名，以使 AWS 可以识别发送者。AWS在将 IAM 角色与 HTTP 数据源关联时，AppSync 代表您计算签名。

您提供两个额外的组件以使用 HTTP 解析器调用 AWS 服务：

- 有权调用 AWS 服务 API 的 IAM 角色
- 数据源中的签名配置

例如，如果要使用 HTTP 解析器调用 ListGraphqlApis 操作，您需要先创建由 AWS AppSync 担任的 IAM 角色并附加以下策略：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "appsync:ListGraphqlApis"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

接下来，为 AWS AppSync 创建 HTTP 数据源。在该示例中，您在美国西部（俄勒冈州）区域中调用 AWS AppSync。在名为 `http.json` 文件中设置以下 HTTP 配置，其中包括签名区域和服务名称：

```
{
    "endpoint": "https://appsync.us-west-2.amazonaws.com/",
    "authorizationConfig": {
        "authorizationType": "AWS_IAM",
        "awsIamConfig": {
            "signingRegion": "us-west-2",
            "signingServiceName": "appsync"
```

```
        }
    }
}
```

然后，使用 AWS CLI 创建具有关联角色的数据源，如下所示：

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

在将一个解析器附加到架构中的字段时，请使用以下请求映射模板调用 AWS AppSync：

```
{
    "version": "2018-05-29",
    "method": "GET",
    "resourcePath": "/v1/apis"
}
```

在为该数据源运行 GraphQL 查询时，AWS AppSync 使用您提供的角色对请求进行签名，并将签名包含在请求中。该查询返回您的账户在该 AWS 区域中的 AWS AppSync GraphQL API 列表。

教程：带有数据 API 的 Aurora PostgreSQL

AWS AppSync 提供了一个数据来源，用于针对使用数据 API 启用的 Amazon Aurora 集群执行 SQL 语句。您可以使用 AWS AppSync 解析程序通过 GraphQL 查询、更改和订阅对数据 API 运行 SQL 语句。

> **Note**
>
> 本教程使用 US-EAST-1 区域。

创建集群

在将 Amazon RDS 数据来源添加到 AWS AppSync 之前，先在 Aurora Serverless 集群上启用数据 API。您还必须使用 AWS Secrets Manager 配置密钥。要创建 Aurora Serverless 集群，您可以使用 AWS CLI：

```
aws rds create-db-cluster \
    --db-cluster-identifier appsync-tutorial \
    --engine aurora-postgresql --engine-version 13.11 \
    --engine-mode serverless \
    --master-username USERNAME \
    --master-user-password COMPLEX_PASSWORD
```

这将为集群返回一个 ARN。您可以使用以下命令检查集群的状态：

```
aws rds describe-db-clusters \
    --db-cluster-identifier appsync-tutorial \
    --query "DBClusters[0].Status"
```

使用上一步中的 USERNAME 和 COMPLEX_PASSWORD 通过 AWS Secrets Manager 控制台创建一个密钥，或者通过 AWS CLI 使用如下输入文件创建密钥：

```
{
    "username": "USERNAME",
    "password": "COMPLEX_PASSWORD"
}
```

将其作为参数传递给 CLI：

```
aws secretsmanager create-secret \
    --name appsync-tutorial-rds-secret \
    --secret-string file://creds.json
```

这将为密钥返回 ARN。记下 Aurora Serverless 集群的 ARN 和密钥，稍后在 AWS AppSync 控制台中创建数据来源时需要使用。

启用数据 API

集群状态更改为 `available` 后，请按照 Amazon RDS 文档启用数据 API。在将数据 API 添加为 AWS AppSync 数据来源之前，必须先启用它。还可以使用 AWS CLI 启用数据 API。

```
aws rds modify-db-cluster \
    --db-cluster-identifier appsync-tutorial \
    --enable-http-endpoint \
    --apply-immediately
```

创建数据库和表

启用您的数据 API 后，请使用 AWS CLI 中的 `aws rds-data execute-statement` 命令来验证它是否正常运行。这可以确保在将 Aurora Serverless 集群添加到 AWS AppSync API 之前，对其进行正确配置。首先，使用 `--sql` 参数创建 TESTDB 数据库：

```
aws rds-data execute-statement \
    --resource-arn "arn:aws:rds:us-east-1:123456789012:cluster:appsync-tutorial" \
    --secret-arn "arn:aws:secretsmanager:us-east-1:123456789012:secret:appsync-
tutorial-rds-secret" \
    --sql "create DATABASE \"testdb\""
```

如果运行而未出现错误，请使用 `create table` 命令添加两个表：

```
 aws rds-data execute-statement \
    --resource-arn "arn:aws:rds:us-east-1:123456789012:cluster:appsync-tutorial" \
    --secret-arn "arn:aws:secretsmanager:us-east-1:123456789012:secret:appsync-
tutorial-rds-secret" \
    --database "testdb" \
    --sql 'create table public.todos (id serial constraint todos_pk primary key,
 description text not null, due date not null, "createdAt" timestamp default now());'

aws rds-data execute-statement \
    --resource-arn "arn:aws:rds:us-east-1:123456789012:cluster:appsync-tutorial" \
    --secret-arn "arn:aws:secretsmanager:us-east-1:123456789012:secret:appsync-
tutorial-rds-secret" \
    --database "testdb" \
    --sql 'create table public.tasks (id serial constraint tasks_pk primary key,
 description varchar, "todoId" integer not null constraint tasks_todos_id_fk references
 public.todos);'
```

如果一切运行正常，您现在可以将集群添加为 API 中的数据来源。

创建 GraphQL 架构

现在，您的 Aurora Serverless 数据 API 正在使用已配置的表运行，我们将创建一个 GraphQL 架构。您可以手动执行此操作，但 AWS AppSync 可让您通过使用 API 创建向导从现有数据库导入表配置，从而快速入门。

首先：

1. 在 AWS AppSync 控制台中，选择创建 API，然后选择以 Amazon Aurora 集群开始。
2. 指定 API 详细信息，例如 API 名称，然后选择要生成 API 的数据库。
3. 选择数据库。如果需要，请更新区域，然后选择您的 Aurora 集群和 TESTDB 数据库。
4. 选择您的密钥，然后选择导入。
5. 发现表后，更新类型名称。将 Todos 更改为 Todo，并将 Tasks 更改为 Task。
6. 通过选择预览架构来预览生成的架构。您的架构将如下所示：

```
type Todo {
  id: Int!
  description: String!
  due: AWSDate!
  createdAt: String
}

type Task {
  id: Int!
  todoId: Int!
  description: String
}
```

7. 对于角色，您可以让 AWS AppSync 创建新角色，也可以使用与以下策略类似的策略创建角色：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "rds-data:ExecuteStatement",
            ],
            "Resource": [
                "arn:aws:rds:us-east-1:123456789012:cluster:appsync-tutorial",
                "arn:aws:rds:us-east-1:123456789012:cluster:appsync-tutorial:*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": [
```

```
                "arn:aws:secretsmanager:us-
east-1:123456789012:secret:your:secret:arn:appsync-tutorial-rds-secret",
                "arn:aws:secretsmanager:us-
east-1:123456789012:secret:your:secret:arn:appsync-tutorial-rds-secret:*"
            ]
        }
    ]
}
```

请注意，此策略中有两条您要授予角色访问权限的语句。第一个资源是您的 Aurora 集群，第二个资源是您的 AWS Secrets Manager ARN。

选择下一步，查看配置详细信息，然后选择创建 API。您现在已拥有完全正常运行的 API。您可以在架构页面上查看 API 的完整详细信息。

RDS 的解析器

API 创建流程会自动创建解析器来与我们的类型进行交互。如果您查看架构页面，您会发现执行以下操作所需的解析器：

- 通过 `Mutation.createTodo` 字段创建 `todo`。
- 通过 `Mutation.updateTodo` 字段更新 `todo`。
- 通过 `Mutation.deleteTodo` 字段删除 `todo`。
- 通过 `Query.getTodo` 字段获取单个 `todo`。
- 通过 `Query.listTodos` 字段列出所有 `todos`。

您会发现该 `Task` 类型附带了类似的字段和解析器。让我们更仔细地看看一些解析器。

Mutation.createTodo

在 AWS AppSync 控制台的架构编辑器中，在右侧选择 `createTodo(...): Todo` 旁边的 `testdb`。解析器代码使用 `rds` 模块中的 `insert` 函数动态创建向 `todos` 表中添加数据的插入语句。因为我们正在使用 Postgres，所以我们可以利用 `returning` 语句来取回插入的数据。

让我们更新解析器以正确指定 due 字段的 DATE 类型：

```
import { util } from '@aws-appsync/utils';
import { insert, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/
rds';
```

```
export function request(ctx) {
    const { input } = ctx.args;
    // if a due date is provided, cast is as `DATE`
    if (input.due) {
        input.due = typeHint.DATE(input.due)
    }
    const insertStatement = insert({
        table: 'todos',
        values: input,
        returning: '*',
    });
    return createPgStatement(insertStatement)
}

export function response(ctx) {
    const { error, result } = ctx;
    if (error) {
        return util.appendError(
            error.message,
            error.type,
            result
        )
    }
    return toJsonObject(result)[0][0]
}
```

保存解析器。类型提示将输入对象中的 due 正确标记为 DATE 类型。这允许 Postgres 引擎正确解释该值。接下来，更新您的架构以从 `CreateTodo` 输入中删除 `id`。因为我们的 Postgres 数据库可以返回生成的 ID，所以我们可以依靠它来创建并以单个请求的形式返回结果：

```
input CreateTodoInput {
    due: AWSDate!
    createdAt: String
    description: String!
}
```

进行更改并更新您的架构。前往查询编辑器，以向数据库添加一个项目：

```
mutation CreateTodo {
  createTodo(input: {description: "Hello World!", due: "2023-12-31"}) {
    id
```

```
    due
    description
    createdAt
  }
}
```

您得到的结果是：

```
{
  "data": {
    "createTodo": {
      "id": 1,
      "due": "2023-12-31",
      "description": "Hello World!",
      "createdAt": "2023-11-14 20:47:11.875428"
    }
  }
}
```

Query.listTodos

在控制台的架构编辑器中，在右侧选择 `listTodos(id: ID!): Todo` 旁边的 `testdb`。请求处理程序使用 select 实用程序函数在运行时动态生成请求。

```
export function request(ctx) {
    const { filter = {}, limit = 100, nextToken } = ctx.args;
    const offset = nextToken ? +util.base64Decode(nextToken) : 0;
    const statement = select({
        table: 'todos',
        columns: '*',
        limit,
        offset,
        where: filter,
    });
    return createPgStatement(statement)
}
```

我们想根据 due 日期筛选 `todos`。让我们更新解析器以将 due 值强制转换为 `DATE`。更新导入列表和请求处理程序：

```
import { util } from '@aws-appsync/utils';
import * as rds from '@aws-appsync/utils/rds';
```

```
export function request(ctx) {
  const { filter: where = {}, limit = 100, nextToken } = ctx.args;
  const offset = nextToken ? +util.base64Decode(nextToken) : 0;

  // if `due` is used in a filter, CAST the values to DATE.
  if (where.due) {
    Object.entries(where.due).forEach(([k, v]) => {
      if (k === 'between') {
        where.due[k] = v.map((d) => rds.typeHint.DATE(d));
      } else {
        where.due[k] = rds.typeHint.DATE(v);
      }
    });
  }

  const statement = rds.select({
    table: 'todos',
    columns: '*',
    limit,
    offset,
    where,
  });
  return rds.createPgStatement(statement);
}

export function response(ctx) {
  const {
    args: { limit = 100, nextToken },
    error,
    result,
  } = ctx;
  if (error) {
    return util.appendError(error.message, error.type, result);
  }
  const offset = nextToken ? +util.base64Decode(nextToken) : 0;
  const items = rds.toJsonObject(result)[0];
  const endOfResults = items?.length < limit;
  const token = endOfResults ? null : util.base64Encode(`${offset + limit}`);
  return { items, nextToken: token };
}
```

让我们试试这个查询。在查询编辑器中：

```
query LIST {
  listTodos(limit: 10, filter: {due: {between: ["2021-01-01", "2025-01-02"]}}) {
    items {
      id
      due
      description
    }
  }
}
```

Mutation.updateTodo

您也可以对 Todo 执行 update。从查询编辑器中，我们来更新第一个 Todo 项目（ id 为 1 ）。

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits"}) {
    description
    due
    id
  }
}
```

请注意，您必须指定要更新的项目的 id。您也可以指定一个条件，以仅更新符合特定条件的项目。例如，如果描述以 edits 开头，我们可能只想编辑该项目：

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits: make a change"}, condition:
 {description: {beginsWith: "edits"}}) {
    description
    due
    id
  }
}
```

就像我们处理 create 和 list 操作的方式一样，我们可以更新解析器以将 due 字段强制转换为 DATE。将这些更改保存到 updateTodo：

```
import { util } from '@aws-appsync/utils';
import * as rds from '@aws-appsync/utils/rds';

export function request(ctx) {
```

```
  const { input: { id, ...values }, condition = {}, } = ctx.args;
  const where = { ...condition, id: { eq: id } };

  // if `due` is used in a condition, CAST the values to DATE.
  if (condition.due) {
    Object.entries(condition.due).forEach(([k, v]) => {
      if (k === 'between') {
        condition.due[k] = v.map((d) => rds.typeHint.DATE(d));
      } else {
        condition.due[k] = rds.typeHint.DATE(v);
      }
    });
  }

  // if a due date is provided, cast is as `DATE`
  if (values.due) {
    values.due = rds.typeHint.DATE(values.due);
  }

  const updateStatement = rds.update({
    table: 'todos',
    values,
    where,
    returning: '*',
  });
  return rds.createPgStatement(updateStatement);
}

export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    return util.appendError(error.message, error.type, result);
  }
  return rds.toJsonObject(result)[0][0];
}
```

现在尝试使用一个条件进行更新：

```
mutation UPDATE {
  updateTodo(
    input: {
        id: 1, description: "edits: make a change", due: "2023-12-12"},
    condition: {
```

```
        description: {beginsWith: "edits"}, due: {ge: "2023-11-08"}})
    {
          description
          due
          id
        }
}
```

Mutation.deleteTodo

您可以通过 deleteTodo 突变对 Todo 执行 delete。这就像 updateTodo 突变一样，您必须指定要删除的项目的 id：

```
mutation DELETE {
  deleteTodo(input: {id: 1}) {
    description
    due
    id
  }
}
```

编写自定义查询

我们已经使用 rds 模块实用程序来创建我们的 SQL 语句。我们还可以编写自己的自定义静态语句来与我们的数据库进行交互。首先，更新架构以从 CreateTask 输入中删除 id 字段。

```
input CreateTaskInput {
    todoId: Int!
    description: String
}
```

接下来，创建几个任务。任务与 Todo 有外键关系：

```
mutation TASKS {
  a: createTask(input: {todoId: 2, description: "my first sub task"}) { id }
  b:createTask(input: {todoId: 2, description: "another sub task"}) { id }
  c: createTask(input: {todoId: 2, description: "a final sub task"}) { id }
}
```

在您的 Query 类型中创建一个名为 getTodoAndTasks 的新字段：

```
getTodoAndTasks(id: Int!): Todo
```

向 Todo 类型添加 tasks 字段：

```
type Todo {
    due: AWSDate!
    id: Int!
    createdAt: String
    description: String!
    tasks:TaskConnection
}
```

保存架构。从控制台的架构编辑器中，在右侧为 getTodosAndTasks(id: Int!): Todo 选择附加解析器。选择您的 Amazon RDS 数据来源。使用以下代码更新您的解析器：

```
import { sql, createPgStatement,toJsonObject } from '@aws-appsync/utils/rds';

export function request(ctx) {
    return createPgStatement(
        sql`SELECT * from todos where id = ${ctx.args.id}`,
        sql`SELECT * from tasks where "todoId" = ${ctx.args.id}`);
}

export function response(ctx) {
    const result = toJsonObject(ctx.result);
    const todo = result[0][0];
    if (!todo) {
        return null;
    }
    todo.tasks = { items: result[1] };
    return todo;
}
```

在此代码中，我们使用 sql 标签模板编写了一条 SQL 语句，我们可以在运行时安全地将动态值传递给该语句。createPgStatement 一次最多可以处理两个 SQL 请求。我们用它来为 todo 发送一个查询，并为 tasks 发送另一个查询。您可以使用 JOIN 语句或任何其它方法来完成此操作。其想法是能够编写您自己的 SQL 语句来实现您的业务逻辑。要在查询编辑器中使用查询，我们可以试试下面的操作：

```
query TodoAndTasks {
```

```
  getTodosAndTasks(id: 2) {
    id
    due
    description
    tasks {
      items {
        id
        description
      }
    }
  }
}
```

删除集群

> **Important**
>
> 删除集群是永久性的。在执行此操作之前，请仔细检查您的项目。

删除集群：

```
$ aws rds delete-db-cluster \
    --db-cluster-identifier appsync-tutorial \
    --skip-final-snapshot
```

解析器教程 (VTL)

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

AWS AppSync 通过数据源和解析器转换 GraphQL 请求，并从 AWS 资源中获取信息。AWSAppSync 支持自动预置和连接到某些数据源类型。AWSAppSync 支持将 AWS Lambda、Amazon DynamoDB、关系数据库 (Amazon Aurora Serverless)、Amazon OpenSearch Service 和 HTTP 终端节点作为数据源。您可以将 GraphQL API 与现有的 AWS 资源一起使用，或者构建数据源和解析器。本节通过一系列教程演示了此过程，帮助您更好地理解详细的工作原理以及优化选项。

AWS AppSync 将以 Apache Velocity 模板语言 (VTL) 编写的映射模板用于解析器。有关使用映射模板的更多信息，请参阅解析器映射模板参考。有关使用 VTL 的更多信息，请参阅解析器映射模板编程指南。

AWS AppSync 支持从 GraphQL 架构中自动预置 DynamoDB 表，如“从架构中预置”（可选）和“启动示例架构”中所述。您也可以从现有 DynamoDB 表中导入，从而创建架构并连接解析器。在“从 Amazon DynamoDB 导入”（可选）中简要说明了该内容。

主题

- 教程：DynamoDB 解析器
- 教程：Lambda 解析器
- 教程：Amazon OpenSearch Service 解析器
- 教程：本地解析器
- 教程：组合使用 GraphQL 解析器
- 教程：DynamoDB 批处理解析器
- 教程：DynamoDB 事务解析器
- 教程：HTTP 解析器
- 教程：Aurora Serverless
- 教程：管道解析器
- 教程：增量同步

教程：DynamoDB 解析器

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

本教程说明了如何将您自己的 Amazon DynamoDB 表添加到 AWS AppSync 中，并将其连接到 GraphQL API。

您可以让 AWS AppSync 代表您预置 DynamoDB 资源。如果您愿意，也可以创建数据源和解析器，将现有的表连接到 GraphQL 架构。在这两种情况下，您都可以通过 GraphQL 语句读写您的 DynamoDB 数据库，并订阅实时数据。

要将 GraphQL 语句转换为 DynamoDB 操作，并将响应转换回 GraphQL，需要完成一些特定的配置步骤。本教程通过一些现实世界的场景和数据访问模式介绍了配置过程。

设置您的 DynamoDB 表

要开始本教程，您需要先按照以下步骤预置 AWS 资源。

1. 在 CLI 中使用以下 AWS CloudFormation 模板预置 AWS 资源：

   ```
   aws cloudformation create-stack \
       --stack-name AWSAppSyncTutorialForAmazonDynamoDB \
       --template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/
   dynamodb/AmazonDynamoDBCFTemplate.yaml \
       --capabilities CAPABILITY_NAMED_IAM
   ```

   或者，可以在您的 AWS 账户的 US-West-2（俄勒冈州）区域中启动以下 AWS CloudFormation 堆栈。

   这会创建以下内容：

   - 名为 `AppSyncTutorial-Post` 的 DynamoDB 表，用于保留 `Post` 数据。
   - 一个 IAM 角色和关联的 IAM 托管策略，旨在允许 AWS AppSync 与 `Post` 表交互。

2. 要了解堆栈和所创建资源的更多详细信息，请运行以下 CLI 命令：

```
aws cloudformation describe-stacks --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

3. 稍后要删除资源，您可以运行以下操作：

```
aws cloudformation delete-stack --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

创建您的 GraphQL API

要在 AWS AppSync 中创建 GraphQL API，请执行以下操作：

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   - 在 API 控制面板中，选择创建 API。
2. 在自定义您的 API 或从 Amazon DynamoDB 导入下面，选择从头开始构建。
   - 选择同一窗口右侧的开始。
3. 在 API 名称字段中，将 API 的名称设置为 `AWSAppSyncTutorial`。
4. 选择创建。

AWS AppSync 控制台使用 API 密钥身份验证模式为您创建新的 GraphQL API。您可以根据本教程后面的说明，使用控制台设置 GraphQL API 的其余部分，并针对它运行查询。

定义基本文章 API

您现已创建 AWS AppSync GraphQL API，您可以设置一个基本架构，以允许对文章数据执行基本创建、检索和删除操作。

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   - 在 API 控制面板中，选择您刚刚创建的 API。
2. 在侧边栏中，选择架构。
   - 在架构窗格中，将内容替换为以下代码：

```
schema {
    query: Query
```

```
    mutation: Mutation
}

type Query {
    getPost(id: ID): Post
}

type Mutation {
    addPost(
        id: ID!
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}

type Post {
    id: ID!
    author: String
    title: String
    content: String
    url: String
    ups: Int!
    downs: Int!
    version: Int!
}
```

3. 选择保存。

此架构定义 Post 类型，执行操作以添加并获取 Post 对象。

为 DynamoDB 表配置数据源

接下来，将架构中定义的查询和变更链接到 AppSyncTutorial-Post DynamoDB 表。

首先，AWS AppSync 需要识别您的表。您可以在 AWS AppSync 中设置数据源以完成该操作：

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   a. 在 API 控制面板中，选择您的 GraphQL API。
   b. 在侧边栏中，选择数据源。

2. 选择创建数据源。

   a. 对于数据源名称，输入 PostDynamoDBTable。

   b. 对于数据源类型，选择 Amazon DynamoDB 表。

   c. 对于区域，选择 US-WEST-2。

   d. 对于表名称，选择 AppSyncTutorial-Post DynamoDB 表。

   e. 创建新的 IAM 角色（建议），或者选择具有 lambda:invokeFunction IAM 权限的现有角色。现有角色需要具有一个信任策略，如附加数据源一节中所述。

      以下是一个示例 IAM 策略，该策略具有对资源执行操作所需的权限：

```
{
     "Version": "2012-10-17",
     "Statement": [
         {
             "Effect": "Allow",
             "Action": [ "lambda:invokeFunction" ],
             "Resource": [
                 "arn:aws:lambda:us-west-2:123456789012:function:myFunction",
                 "arn:aws:lambda:us-west-2:123456789012:function:myFunction:*"
             ]
         }
     ]
 }
```

3. 选择创建。

设置 addPost 解析器 (DynamoDB PutItem)

在 AWS AppSync 识别 DynamoDB 表后，您可以定义解析器以将其链接到各个查询和变更。您创建的第一个解析器是 addPost 解析器，可用于在 AppSyncTutorial-Post DynamoDB 表中创建文章。

解析器具有以下组件：

- GraphQL 架构中的位置，用于附加解析器。在本例中，您将设置 addPost 类型的 Mutation 字段的解析器。在调用方调用 mutation { addPost(...){...} } 时，将调用该解析器。
- 此解析器所用的数据源。在本例中，您要使用之前定义的 PostDynamoDBTable 数据源，这样您就可以在 AppSyncTutorial-Post DynamoDB 表中添加条目。

- 请求映射模板。请求映射模板的用途是，获取来自调用方的传入请求，并将其转换为 AWS AppSync 指令以对 DynamoDB 执行。
- 响应映射模板。响应映射模板的任务是将 DynamoDB 的响应转换回 GraphQL 期待获得的内容。如果 DynamoDB 中的数据形态与 GraphQL 中的 `Post` 类型不同，此模板很有用。但在此例中它们的形态相同，所以只用于传递数据。

设置解析器：

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   a. 在 API 控制面板中，选择您的 GraphQL API。
   b. 在侧边栏中，选择数据源。
2. 选择创建数据源。
   a. 对于数据源名称，输入 `PostDynamoDBTable`。
   b. 对于数据源类型，选择 Amazon DynamoDB 表。
   c. 对于区域，选择 US-WEST-2。
   d. 对于表名称，选择 AppSyncTutorial-Post DynamoDB 表。
   e. 创建新的 IAM 角色（建议），或者选择具有 `lambda:invokeFunction` IAM 权限的现有角色。现有角色需要具有一个信任策略，如附加数据源一节中所述。

      以下是一个示例 IAM 策略，该策略具有对资源执行操作所需的权限：

```
{
     "Version": "2012-10-17",
     "Statement": [
         {
             "Effect": "Allow",
             "Action": [ "lambda:invokeFunction" ],
             "Resource": [
                 "arn:aws:lambda:us-west-2:123456789012:function:myFunction",
                 "arn:aws:lambda:us-west-2:123456789012:function:myFunction:*"
             ]
         }
     ]
 }
```

3. 选择创建。

4. 选择架构选项卡。
5. 在右侧的数据类型窗格中，找到 Mutation 类型上的 addPost 字段，然后选择附加。
6. 在操作菜单中，选择更新运行时，然后选择单位解析器 (仅限 VTL)。
7. 在 Data source name (数据源名称) 中，选择 PostDynamoDBTable。
8. 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下内容：

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "attributeValues" : {
        "author" : $util.dynamodb.toDynamoDBJson($context.arguments.author),
        "title" : $util.dynamodb.toDynamoDBJson($context.arguments.title),
        "content" : $util.dynamodb.toDynamoDBJson($context.arguments.content),
        "url" : $util.dynamodb.toDynamoDBJson($context.arguments.url),
        "ups" : { "N" : 1 },
        "downs" : { "N" : 0 },
        "version" : { "N" : 1 }
    }
}
```

注意：为所有键和属性值指定了类型。例如，您将 `author` 字段设置为 `{ "S" : "${context.arguments.author}" }`。`S` 部分向 AWS AppSync 和 DynamoDB 指示该值是字符串值。实际的值由 `author` 参数填充。与此类似，`version` 字段是一个数字字段，因为它使用 `N` 作为类型。最后，您还将初始化 `ups`、`downs` 和 `version` 字段。

对于本教程，您已指定 GraphQL `ID!` 类型作为客户端参数的一部分，该类型对插入到 DynamoDB 的新项目编制索引。AWS AppSync 附带一个名为 `$utils.autoId()` 的自动 ID 生成实用程序，您也可以按 `"id" : { "S" : "${$utils.autoId()}" }` 形式使用该实用程序。然后，就可以在 `id: ID!` 的架构定义中省去 `addPost()`，因为它将自动插入。您不会在本教程中使用该技术，但在写入到 DynamoDB 表时，您应该将其视为一种很好的做法。

有关映射模板的更多信息，请参阅 解析器映射模板概述参考文档。有关 GetItem 请求映射的更多信息，请参阅 GetItem 参考文档。有关类型的更多信息，请参阅类型系统（请求映射）参考文档。

9. 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下内容：

```
$utils.toJson($context.result)
```

注意：由于 AppSyncTutorial-Post 表中的数据形态与 GraphQL 中 Post 类型的形态完全匹配，响应映射模板只会直接传递结果。还请注意，此教程中的所有示例均使用同一响应映射模板，所以您只需创建一个文件。

10. 选择保存。

调用 API 以添加文章

解析器现已设置完毕，AWS AppSync 可以将传入的 addPost 变更转换为 DynamoDB PutItem 操作。现在，您可以运行一个变更，在表中添加内容。

- 选择 Queries 选项卡。
- 在 Queries (查询) 窗格中，粘贴以下变更：

```
mutation addPost {
  addPost(
    id: 123
    author: "AUTHORNAME"
    title: "Our first post!"
    content: "This is our first post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 选择 Execute query (执行查询)（橙色播放按钮）。
- 新创建的文章的结果应出现在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
```

```
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

以下是具体过程：

- AWS AppSync 收到 addPost 变更请求。
- AWS AppSync 获取该请求和请求映射模板，并生成请求映射文档。如下所示：

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id" : { "S" : "123" }
    },
    "attributeValues" : {
        "author": { "S" : "AUTHORNAME" },
        "title": { "S" : "Our first post!" },
        "content": { "S" : "This is our first post." },
        "url": { "S" : "https://aws.amazon.com/appsync/" },
        "ups" : { "N" : 1 },
        "downs" : { "N" : 0 },
        "version" : { "N" : 1 }
    }
}
```

- AWS AppSync 使用请求映射文档生成并执行 DynamoDB PutItem 请求。
- AWS AppSync 获取 PutItem 请求结果，并将其转换回 GraphQL 类型。

```
{
    "id" : "123",
    "author": "AUTHORNAME",
```

```
    "title": "Our first post!",
    "content": "This is our first post.",
    "url": "https://aws.amazon.com/appsync/",
    "ups" : 1,
    "downs" : 0,
    "version" : 1
}
```

- 通过响应映射文档进行传递，没有变化。
- 在 GraphQL 响应中返回新创建的对象。

设置 getPost 解析器 (DynamoDB GetItem)

您现在能够将数据添加到 AppSyncTutorial-Post DynamoDB 表中，您需要设置 getPost 查询，以使其可以从 AppSyncTutorial-Post 表中检索该数据。为了实现此目的，您要设置另一解析器。

- 选择架构选项卡。
- 在右侧的数据类型窗格中，找到 Query 类型上的 getPost 字段，然后选择附加。
- 在操作菜单中，选择更新运行时，然后选择单位解析器 (仅限 VTL)。
- 在 Data source name (数据源名称) 中，选择 PostDynamoDBTable。
- 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下内容：

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

- 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下内容：

```
$utils.toJson($context.result)
```

- 选择保存。

调用 API 以获取文章

解析器现已设置完毕，AWS AppSync 知道如何将传入的 `getPost` 查询转换为 DynamoDB `GetItem` 操作。现在，您可以运行查询，检索之前创建的文章。

- 选择 Queries 选项卡。
- 在Queries (查询) 窗格中粘贴以下内容：

```
query getPost {
  getPost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 选择 Execute query (执行查询)（橙色播放按钮）。
- 从 DynamoDB 中检索的文章应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "getPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our first post!",
      "content": "This is our first post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

以下是具体过程：

- AWS AppSync 收到 `getPost` 查询请求。
- AWS AppSync 获取该请求和请求映射模板，并生成请求映射文档。如下所示：

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "123" }
    }
}
```

- AWS AppSync 使用请求映射文档生成并执行 DynamoDB GetItem 请求。
- AWS AppSync 获取 `GetItem` 请求结果，并将其转换回 GraphQL 类型。

```
{
    "id" : "123",
    "author": "AUTHORNAME",
    "title": "Our first post!",
    "content": "This is our first post.",
    "url": "https://aws.amazon.com/appsync/",
    "ups" : 1,
    "downs" : 0,
    "version" : 1
}
```

- 通过响应映射文档进行传递，没有变化。
- 在响应中返回检索到的对象。

或者，采用以下示例：

```
query getPost {
  getPost(id:123) {
    id
    author
    title
  }
}
```

如果您的 getPost 查询仅需要 id、author 和 title，您可以将请求映射模板更改为使用投影表达式仅指定您希望从 DynamoDB 表中获取的属性，以避免将不必要的数据从 DynamoDB 传输到 AWS AppSync。例如，请求映射模板可能类似于以下代码片段：

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "projection" : {
     "expression" : "#author, id, title",
     "expressionNames" : { "#author" : "author"}
    }
}
```

创建 updatePost 变更 (DynamoDB UpdateItem)

到目前为止，您可以在 DynamoDB 中创建和检索 Post 对象。现在，您要设置一项新的变更，以便更新对象。您将使用 UpdateItem DynamoDB 操作实现此目的。

- 选择架构选项卡。
- 在 Schema (架构) 窗格中修改 Mutation 类型，添加新的 updatePost 变更，如下所示：

```
type Mutation {
    updatePost(
        id: ID!,
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post
    addPost(
        author: String!
        title: String!
        content: String!
        url: String!
    ): Post!
}
```

- 选择保存。

- 在右侧的数据类型窗格中，找到 Mutation 类型上的新创建的 updatePost 字段，然后选择附加。
- 在操作菜单中，选择更新运行时，然后选择单位解析器 (仅限 VTL)。
- 在 Data source name (数据源名称) 中，选择 PostDynamoDBTable。
- 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下内容：

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "SET author = :author, title = :title, content = :content,
 #url = :url ADD version :one",
        "expressionNames": {
            "#url" : "url"
        },
        "expressionValues": {
            ":author" : $util.dynamodb.toDynamoDBJson($context.arguments.author),
            ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title),
            ":content" : $util.dynamodb.toDynamoDBJson($context.arguments.content),
            ":url" : $util.dynamodb.toDynamoDBJson($context.arguments.url),
            ":one" : { "N": 1 }
        }
    }
}
```

注意：该解析器使用 DynamoDB UpdateItem，这与 PutItem 操作存在明显的差异。您仅要求 DynamoDB 更新某些属性，而不是编写整个项目。这是使用 DynamoDB 更新表达式完成的。表达式本身是在 `expression` 部分的 `update` 字段中指定的。它会设置 `author`、`title`、`content` 和 URL 属性，还会递增 `version` 字段。要使用的值不会出现在表达式本身；表达式中的占位符名称以冒号打头，并在 `expressionValues` 字段中进行定义。最后，DynamoDB 具有一些保留字，它们不能出现在 `expression` 中。例如，`url` 是保留关键字，所以要更新 `url` 字段，您可使用名称占位符，并在 `expressionNames` 字段中定义它们。

有关 `UpdateItem` 请求映射的更多信息，请参阅 UpdateItem 参考文档。有关如何编写更新表达式的更多信息，请参阅 DynamoDB UpdateExpressions 文档。

- 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下内容：

```
$utils.toJson($context.result)
```

调用 API 以更新文章

解析器现已设置完毕，AWS AppSync 知道如何将传入的 update 变更转换为 DynamoDB Update 操作。现在，您可以运行变更，以更新您之前写入的项目。

- 选择 Queries 选项卡。
- 在 Queries (查询) 窗格中，粘贴以下变更。您还需要将 id 参数更新为您以前记下的值。

```
mutation updatePost {
  updatePost(
    id:"123"
    author: "A new author"
    title: "An updated author!"
    content: "Now with updated content!"
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 选择 Execute query (执行查询) (橙色播放按钮) 。
- 在 DynamoDB 中更新的文章应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "updatePost": {
      "id": "123",
      "author": "A new author",
      "title": "An updated author!",
      "content": "Now with updated content!",
      "url": "https://aws.amazon.com/appsync/",
```

```
      "ups": 1,
      "downs": 0,
      "version": 2
    }
  }
}
```

在该示例中，未修改 ups 和 downs 字段，因为请求映射模板不要求 AWS AppSync 和 DynamoDB 对这些字段执行任何操作。此外，version 字段增加 1，因为您要求 AWS AppSync 和 DynamoDB 在 version 字段中添加 1。

修改 updatePost 解析器 (DynamoDB UpdateItem)

updatePost 变更看上去不错，但它有两个主要问题：

- 如果您只希望更新一个字段，则必须更新所有字段。
- 如果两个人同时修改对象，您可能会丢失信息。

为了解决这些问题，您要修改 updatePost 变更，做到只修改请求中指定的参数，然后在 UpdateItem 操作中添加条件。

1. 选择架构选项卡。
2. 在 Schema (架构) 窗格中修改 Mutation 类型中的 updatePost 字段，删除 author、title、content 和 url 参数的感叹号，确保 id 字段不变。这样它们就会成为可选参数。还要新增一个必需 expectedVersion 参数。

```
type Mutation {
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        author: String!
        title: String!
        content: String!
```

```
        url: String!
    ): Post!
}
```

3. 选择保存。
4. 在右侧的数据类型窗格中，找到 Mutation 类型的 updatePost 字段。
5. 选择 PostDynamoDBTable 以打开现有解析器。
6. 在 Configure the request mapping template (配置请求映射模板) 中修改请求映射模板，如下所示：

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },

    ## Set up some space to keep track of things you're updating **
    #set( $expNames  = {} )
    #set( $expValues = {} )
    #set( $expSet = {} )
    #set( $expAdd = {} )
    #set( $expRemove = [] )

    ## Increment "version" by 1 **
    $!{expAdd.put("version", ":one")}
    $!{expValues.put(":one", { "N" : 1 })}

    ## Iterate through each argument, skipping "id" and "expectedVersion" **
    #foreach( $entry in $context.arguments.entrySet() )
        #if( $entry.key != "id" && $entry.key != "expectedVersion" )
            #if( (!$entry.value) && ("$!{entry.value}" == "") )
                ## If the argument is set to "null", then remove that attribute from
 the item in DynamoDB **

                #set( $discard = ${expRemove.add("#${entry.key}")} )
                $!{expNames.put("#${entry.key}", "$entry.key")}
            #else
                ## Otherwise set (or update) the attribute on the item in DynamoDB **

                $!{expSet.put("#${entry.key}", ":${entry.key}")}
                $!{expNames.put("#${entry.key}", "$entry.key")}
                $!{expValues.put(":${entry.key}", { "S" : "${entry.value}" })}
```

```
            #end
        #end
    #end

    ## Start building the update expression, starting with attributes you're going to
SET **
    #set( $expression = "" )
    #if( !${expSet.isEmpty()} )
        #set( $expression = "SET" )
        #foreach( $entry in $expSet.entrySet() )
            #set( $expression = "${expression} ${entry.key} = ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Continue building the update expression, adding attributes you're going to ADD
**
    #if( !${expAdd.isEmpty()} )
        #set( $expression = "${expression} ADD" )
        #foreach( $entry in $expAdd.entrySet() )
            #set( $expression = "${expression} ${entry.key} ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Continue building the update expression, adding attributes you're going to
REMOVE **
    #if( !${expRemove.isEmpty()} )
        #set( $expression = "${expression} REMOVE" )

        #foreach( $entry in $expRemove )
            #set( $expression = "${expression} ${entry}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Finally, write the update expression into the document, along with any
expressionNames and expressionValues **
```

```
    "update" : {
        "expression" : "${expression}"
        #if( !${expNames.isEmpty()} )
            ,"expressionNames" : $utils.toJson($expNames)
        #end
        #if( !${expValues.isEmpty()} )
            ,"expressionValues" : $utils.toJson($expValues)
        #end
    },

    "condition" : {
        "expression"       : "version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" :
 $util.dynamodb.toDynamoDBJson($context.arguments.expectedVersion)
        }
    }
}
```

7. 选择保存。

此模板是一个更复杂的示例。它演示了映射模板的强大功能和灵活性。它遍历所有参数，跳过 `id` 和 `expectedVersion`。如果参数设置为某个值，它要求 AWS AppSync 和 DynamoDB 为 DynamoDB 中的对象更新该属性。如果该属性设置为 Null，它要求 AWS AppSync 和 DynamoDB 从 Post 对象中删除该属性。如果未指定参数，该属性会保留原样。它还会递增 `version` 字段。

还有一个新的 `condition` 部分。通过使用条件表达式，您可以在执行操作之前根据 DynamoDB 中的已有对象的状态向 AWS AppSync 和 DynamoDB 通知请求是否会成功。在该示例中，只有在当前位于 DynamoDB 中的项目的 `version` 字段与 `expectedVersion` 参数完全匹配时，您才希望 `UpdateItem` 请求成功。

有关条件表达式的更多信息，请参阅条件表达式参考文档。

调用 API 以更新文章

让我们尝试使用新的解析器更新 `Post` 对象：

- 选择 Queries 选项卡。
- 在 Queries (查询) 窗格中，粘贴以下变更。您还需要将 `id` 参数更新为您以前记下的值。

```
mutation updatePost {
  updatePost(
```

```
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 2
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 选择 Execute query (执行查询)（橙色播放按钮）。
- 在 DynamoDB 中更新的文章应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "updatePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 3
    }
  }
}
```

在该请求中，您要求 AWS AppSync 和 DynamoDB 仅更新 `title` 和 `content` 字段。它不会处理所有其他字段（除了递增 `version` 字段）。您将 `title` 属性设置为新的值，并从文章中删除 `content` 属性。`author`、`url`、`ups` 和 `downs` 字段没有变化。

请尝试再次执行变更请求，保持请求完全不变。您可以看到类似以下内容的响应：

```
{
  "data": {
```

```
    "updatePost": null
  },
  "errors": [
    {
      "path": [
        "updatePost"
      ],
      "data": {
        "id": "123",
        "author": "A new author",
        "title": "An empty story",
        "content": null,
        "url": "https://aws.amazon.com/appsync/",
        "ups": 1,
        "downs": 0,
        "version": 3
      },
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
    }
  ]
}
```

请求失败，因为条件表达式的评估结果为 false：

- 第一次运行请求时，DynamoDB 中的文章的 `version` 字段的值为 2，它与 `expectedVersion` 参数匹配。请求成功，这意味着 DynamoDB 中的 `version` 字段已增加到 3。
- 第二次运行请求时，DynamoDB 中的文章的 `version` 字段的值为 3，它与 `expectedVersion` 参数不匹配。

这种模式通常被称为乐观锁。

AWS AppSync DynamoDB 解析器的一项功能是，它返回 DynamoDB 中的 Post 对象的当前值。您可以在 GraphQL 响应的 `data` 部分的 `errors` 字段中找到这个值。您的应用程序可以利用此信息决定应

如何继续。在该示例中，您可以看到 DynamoDB 中的对象的 `version` 字段设置为 `3`，因此，您只需将 `expectedVersion` 参数更新为 `3`，请求就会再次成功。

有关如何处理条件检查失败的更多信息，请参阅条件表达式映射模板参考文档。

创建 upvotePost 和 downvotePost 变更 (DynamoDB UpdateItem)

`Post` 类型有 `ups` 和 `downs` 字段，用于记录点赞和差评，但现在还无法通过 API 使用它们。让我们添加一些变更，对文章点赞和差评。

- 选择架构选项卡。
- 在 Schema (架构) 窗格中修改 `Mutation` 类型，添加新的 `upvotePost` 和 `downvotePost` 变更，如下所示：

```
type Mutation {
    upvotePost(id: ID!): Post
    downvotePost(id: ID!): Post
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post!
}
```

- 选择保存。
- 在右侧的数据类型窗格中，找到 Mutation 类型上的新创建的 upvotePost 字段，然后选择附加。
- 在操作菜单中，选择更新运行时，然后选择单位解析器 (仅限 VTL)。
- 在 Data source name (数据源名称) 中，选择 PostDynamoDBTable。
- 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下内容：

```
{
```

```
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "ADD ups :plusOne, version :plusOne",
        "expressionValues" : {
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下内容：

```
$utils.toJson($context.result)
```

- 选择保存。
- 在右侧的数据类型窗格中，找到 Mutation 类型上的新创建的 `downvotePost` 字段，然后选择附加。
- 在 Data source name (数据源名称) 中，选择 PostDynamoDBTable。
- 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下内容：

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "ADD downs :plusOne, version :plusOne",
        "expressionValues" : {
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下内容：

```
$utils.toJson($context.result)
```

- 选择保存。

调用 API，为文章点赞或差评

新的解析器现已设置完毕，AWS AppSync 知道如何将传入的 upvotePost 或 downvote 变更转换为 DynamoDB UpdateItem 操作。现在您可以运行变更，为之前创建的文章点赞或差评。

- 选择 Queries 选项卡。
- 在 Queries (查询) 窗格中，粘贴以下变更。您还需要将 id 参数更新为您以前记下的值。

```
mutation votePost {
  upvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 选择 Execute query (执行查询)（橙色播放按钮）。
- 将在 DynamoDB 中更新文章，并且应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "upvotePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 0,
      "version": 4
    }
  }
}
```

- 再选择几次 (执行查询) 按钮。您应看到，每次您执行查询时，ups 和 version 字段均会递增 1。
- 更改查询以调用 downvotePost 变更，如下所示：

```
mutation votePost {
  downvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 选择 Execute query (执行查询) (橙色播放按钮) 。这次您应看到，每次您执行查询时，downs 和 version 字段均会递增 1。

```
{
  "data": {
    "downvotePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 4,
      "version": 12
    }
  }
}
```

设置 deletePost 解析器 (DynamoDB DeleteItem)

接下来您要设置的变更是删除一个文章。您将使用 DeleteItem DynamoDB 操作完成该操作。

- 选择架构选项卡。
- 在 Schema (架构) 窗格中修改 Mutation 类型，添加新的 deletePost 变更，如下所示：

```
type Mutation {
    deletePost(id: ID!, expectedVersion: Int): Post
```

```
    upvotePost(id: ID!): Post
    downvotePost(id: ID!): Post
    updatePost(
        id: ID!,
        author: String,
        title: String,
        content: String,
        url: String,
        expectedVersion: Int!
    ): Post
    addPost(
        author: String!,
        title: String!,
        content: String!,
        url: String!
    ): Post!
}
```

这次您将 `expectedVersion` 字段设为可选，稍后在添加请求映射模板时将对此进行说明。

- 选择保存。
- 在右侧的数据类型窗格中，找到 Mutation 类型上的新创建的 delete 字段，然后选择附加。
- 在操作菜单中，选择更新运行时，然后选择单位解析器 (仅限 VTL)。
- 在 Data source name (数据源名称) 中，选择 PostDynamoDBTable。
- 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下内容：

```
{
    "version" : "2017-02-28",
    "operation" : "DeleteItem",
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($context.arguments.id)
    }
    #if( $context.arguments.containsKey("expectedVersion") )
        ,"condition" : {
            "expression"       : "attribute_not_exists(id) OR version
 = :expectedVersion",
            "expressionValues" : {
                ":expectedVersion" :
 $util.dynamodb.toDynamoDBJson($context.arguments.expectedVersion)
            }
        }
    #end
```

```
}
```

注意：expectedVersion 参数是可选的。如果调用方在请求中设置 expectedVersion 参数，模板将添加一个条件，只有在已删除项目或 DynamoDB 中的文章的 version 属性与 expectedVersion 完全匹配时，才允许 DeleteItem 请求成功。如果未设置此参数，则 DeleteItem 请求中不指定条件表达式。无论 version 值如何，或者项目在 DynamoDB 中是否存在，该请求都会成功。

- 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下内容：

```
$utils.toJson($context.result)
```

注意：即使您要删除一个项目，如果该项目不是已经删除，还是可以返回要删除的项目。

- 选择保存。

有关 DeleteItem 请求映射的更多信息，请参阅 DeleteItem 参考文档。

调用 API 以删除文章

解析器现已设置完毕，AWS AppSync 知道如何将传入的 delete 变更转换为 DynamoDB DeleteItem 操作。现在，您可以运行变更，从表中删除一些内容。

- 选择 Queries 选项卡。
- 在 Queries (查询) 窗格中，粘贴以下变更。您还需要将 id 参数更新为您以前记下的值。

```
mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 选择 Execute query (执行查询)（橙色播放按钮）。

- 该文章已从 DynamoDB 中删除。请注意，AWS AppSync 返回从 DynamoDB 中删除的项目的值，它应显示在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "A new author",
      "title": "An empty story",
      "content": null,
      "url": "https://aws.amazon.com/appsync/",
      "ups": 6,
      "downs": 4,
      "version": 12
    }
  }
}
```

只有调用的 `deletePost` 将项目从 DynamoDB 中实际删除，才会返回值。

- 再次选择 Execute query (执行查询)。
- 调用仍然成功，但没有返回任何值。

```
{
  "data": {
    "deletePost": null
  }
}
```

现在，让我们尝试删除一篇文章，但这次指定 `expectedValue`。但首先您需要创建一个新文章，因为您刚刚删除了一直在使用的文章。

- 在 Queries (查询) 窗格中，粘贴以下变更：

```
mutation addPost {
  addPost(
    id:123
    author: "AUTHORNAME"
    title: "Our second post!"
```

```
    content: "A new post."
    url: "https://aws.amazon.com/appsync/"
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 选择 Execute query (执行查询) (橙色播放按钮) 。
- 新创建的文章的结果应出现在查询窗格右侧的结果窗格中。记下新建对象的 `id`，因为一会您将用到它。如下所示：

```
{
  "data": {
    "addPost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

现在让我们尝试删除这个文章，但放入 `expectedVersion` 的错误值：

- 在 Queries (查询) 窗格中，粘贴以下变更。您还需要将 `id` 参数更新为您以前记下的值。

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
```

```
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 选择 Execute query (执行查询)（橙色播放按钮）。

```
{
  "data": {
    "deletePost": null
  },
  "errors": [
    {
      "path": [
        "deletePost"
      ],
      "data": {
        "id": "123",
        "author": "AUTHORNAME",
        "title": "Our second post!",
        "content": "A new post.",
        "url": "https://aws.amazon.com/appsync/",
        "ups": 1,
        "downs": 0,
        "version": 1
      },
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "locations": [
        {
          "line": 2,
          "column": 3
        }
      ],
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
    }
```

```
  ]
}
```

请求失败，因为条件表达式的评估结果为 false：DynamoDB 中的文章的 `version` 值与参数中指定的 `expectedValue` 不匹配。对象的当前值返回到 GraphQL 响应的 `data` 部分的 `errors` 字段中。

- 重试请求，但更正 `expectedVersion`：

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 选择 Execute query (执行查询)（橙色播放按钮）。
- 这次请求成功，并返回从 DynamoDB 中删除的值：

```
{
  "data": {
    "deletePost": {
      "id": "123",
      "author": "AUTHORNAME",
      "title": "Our second post!",
      "content": "A new post.",
      "url": "https://aws.amazon.com/appsync/",
      "ups": 1,
      "downs": 0,
      "version": 1
    }
  }
}
```

- 再次选择 Execute query (执行查询)。
- 调用仍然成功，但这次没有返回任何值，因为已在 DynamoDB 中删除该文章。

```
{
  "data": {
    "deletePost": null
  }
}
```

设置 allPost 解析器 (DynamoDB Scan)

到目前为止，只有在您知道要查看的每篇文章的 `id` 时，才能使用该 API。让我们添加新的解析器，它可以返回表中的所有文章。

- 选择架构选项卡。
- 在 Schema (架构) 窗格中修改 `Query` 类型，添加新的 `allPost` 查询，如下所示：

```
type Query {
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

- 添加新 `PaginationPosts` 类型：

```
type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}
```

- 选择保存。
- 在右侧的数据类型窗格中，找到 Query 类型上的新创建的 allPost 字段，然后选择附加。
- 在操作菜单中，选择更新运行时，然后选择单位解析器 (仅限 VTL)。
- 在 Data source name (数据源名称) 中，选择 PostDynamoDBTable。
- 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下内容：

```
{
    "version" : "2017-02-28",
```

```
    "operation" : "Scan"
    #if( ${context.arguments.count} )
        ,"limit": $util.toJson($context.arguments.count)
    #end
    #if( ${context.arguments.nextToken} )
        ,"nextToken": $util.toJson($context.arguments.nextToken)
    #end
}
```

此解析器有两个可选参数：count 指定单次调用可返回的项目数量上限；nextToken 可用于检索下一组结果（稍后您将展示 nextToken 的值来自何处）。

- 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下内容：

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

注意：此响应映射模板与目前我们所用到的其他模板均不同。allPost 查询的结果是 PaginatedPosts，其中包含一组文章和一个分页标记。该对象的形状与从 AWS AppSync DynamoDB 解析器返回的对象不同：文章列表在 items AppSync DynamoDB 解析器结果中命名为 AWS，但在 PaginatedPosts 中命名为 posts。

- 选择保存。

有关 Scan 请求映射的更多信息，请参阅 Scan 参考文档。

调用 API 以扫描所有文章

解析器现已设置完毕，AWS AppSync 知道如何将传入的 allPost 查询转换为 DynamoDB Scan 操作。现在您可以扫描整个表，检索所有文章。

在进行尝试之前，您需要在表中填充一些数据，因为您已经删除了之前使用的所有内容。

- 选择 Queries 选项卡。
- 在 Queries (查询) 窗格中，粘贴以下变更：

```
mutation addPost {
```

```
  post1: addPost(id:1 author: "AUTHORNAME" title: "A series of posts, Volume 1"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post2: addPost(id:2 author: "AUTHORNAME" title: "A series of posts, Volume 2"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post3: addPost(id:3 author: "AUTHORNAME" title: "A series of posts, Volume 3"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post4: addPost(id:4 author: "AUTHORNAME" title: "A series of posts, Volume 4"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post5: addPost(id:5 author: "AUTHORNAME" title: "A series of posts, Volume 5"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post6: addPost(id:6 author: "AUTHORNAME" title: "A series of posts, Volume 6"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post7: addPost(id:7 author: "AUTHORNAME" title: "A series of posts, Volume 7"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post8: addPost(id:8 author: "AUTHORNAME" title: "A series of posts, Volume 8"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
  post9: addPost(id:9 author: "AUTHORNAME" title: "A series of posts, Volume 9"
 content: "Some content" url: "https://aws.amazon.com/appsync/" ) { title }
}
```

- 选择 Execute query (执行查询) (橙色播放按钮) 。

现在，让我们扫描表，每次返回 5 个结果。

- 在Queries (查询) 窗格中粘贴以下查询：

```
query allPost {
  allPost(count: 5) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- 选择 Execute query (执行查询) (橙色播放按钮) 。
- 最前面的 5 个文章应出现在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "allPost": {
```

```
      "posts": [
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        },
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        }
      ],
      "nextToken":
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
    }
  }
}
```

您可以看到 5 个结果，还有一个 nextToken，可用于获得下一组结果。

- 更新 allPost 查询，加入上一组结果的 nextToken：

```
query allPost {
  allPost(
    count: 5
    nextToken:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
  ) {
    posts {
      id
      author
    }
```

```
    nextToken
  }
}
```

- 选择 Execute query (执行查询)（橙色播放按钮）。
- 其余的 4 个文章应出现在查询窗格右侧的结果窗格中。在这组结果中没有 nextToken，因为您已查看了所有 9 篇文章，没有其余文章了。如下所示：

```
{
  "data": {
    "allPost": {
      "posts": [
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        }
      ],
      "nextToken": null
    }
  }
}
```

设置 allPostsByAuthor 解析器 (DynamoDB Query)

除了扫描 DynamoDB 以查找所有文章以外，您还可以查询 DynamoDB 以检索特定作者创建的文章。您以前创建的 DynamoDB 表已具有一个名为 author-index 的 GlobalSecondaryIndex，您可以将其与 DynamoDB Query 操作一起使用以检索特定作者创建的所有文章。

- 选择架构选项卡。

- 在 Schema (架构) 窗格中修改 Query 类型，添加新的 allPostsByAuthor 查询，如下所示：

```
type Query {
    allPostsByAuthor(author: String!, count: Int, nextToken: String): PaginatedPosts!
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

  注意：这次使用与 allPost 查询相同的 PaginatedPosts 类型。

- 选择保存。
- 在右侧的数据类型窗格中，找到 Query 类型上的新创建的 allPostsByAuthor 字段，然后选择附加。
- 在操作菜单中，选择更新运行时，然后选择单位解析器 (仅限 VTL)。
- 在 Data source name (数据源名称) 中，选择 PostDynamoDBTable。
- 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下内容：

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "index" : "author-index",
    "query" : {
      "expression": "author = :author",
        "expressionValues" : {
          ":author" : $util.dynamodb.toDynamoDBJson($context.arguments.author)
        }
    }
    #if( ${context.arguments.count} )
        ,"limit": $util.toJson($context.arguments.count)
    #end
    #if( ${context.arguments.nextToken} )
        ,"nextToken": "${context.arguments.nextToken}"
    #end
}
```

  与 allPost 解析器相似，此解析器也有两个可选参数：count指定单次调用可返回的项目数量上限；nextToken 可用于检索下一组结果（nextToken 的值可从上次调用获得）。

- 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下内容：

```
{
    "posts": $utils.toJson($context.result.items)
```

```
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

注意：此处使用的响应映射模板与 `allPost` 解析器中所用的相同。

- 选择保存。

了解有关 `Query` 请求映射的更多信息，请参阅 Query 参考文档。

调用 API 以查询某一作者的所有文章

解析器现已设置完毕，AWS AppSync 知道如何将传入的 `allPostsByAuthor` 变更转换为针对 `author-index` 索引的 DynamoDB `Query` 操作。现在，您可以查询表，检索某一作者的所有文章。

但首先我们需要在表中再填充一些文章，因为目前所有文章都是同一作者。

- 选择 Queries 选项卡。
- 在 Queries (查询) 窗格中，粘贴以下变更：

```
mutation addPost {
  post1: addPost(id:10 author: "Nadia" title: "The cutest dog in the world" content:
 "So cute. So very, very cute." url: "https://aws.amazon.com/appsync/" ) { author,
 title }
  post2: addPost(id:11 author: "Nadia" title: "Did you know...?" content: "AppSync
 works offline?" url: "https://aws.amazon.com/appsync/" ) { author, title }
  post3: addPost(id:12 author: "Steve" title: "I like GraphQL" content: "It's great"
 url: "https://aws.amazon.com/appsync/" ) { author, title }
}
```

- 选择 Execute query (执行查询)（橙色播放按钮）。

现在，让我们查询表，返回作者为 `Nadia` 的所有文章。

- 在Queries (查询) 窗格中粘贴以下查询：

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
```

```
      title
    }
    nextToken
  }
}
```

- 选择 Execute query (执行查询) (橙色播放按钮) 。
- 作者为 Nadia 的全部文章应出现在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

Query 的分页方式与 Scan 相同。例如，如果我们查找作者为 AUTHORNAME 的所有文章，每次显示 5 个结果。

- 在Queries (查询) 窗格中粘贴以下查询：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
  ) {
    posts {
      id
      title
    }
```

```
    nextToken
  }
}
```

- 选择 Execute query (执行查询)（橙色播放按钮）。
- 作者为 AUTHORNAME 的全部文章应出现在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "6",
          "title": "A series of posts, Volume 6"
        },
        {
          "id": "4",
          "title": "A series of posts, Volume 4"
        },
        {
          "id": "2",
          "title": "A series of posts, Volume 2"
        },
        {
          "id": "7",
          "title": "A series of posts, Volume 7"
        },
        {
          "id": "1",
          "title": "A series of posts, Volume 1"
        }
      ],
      "nextToken":
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
    }
  }
}
```

- 用上次查询返回的值更新 nextToken 参数，如下所示：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
```

```
    count: 5
    nextToken:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- 选择 Execute query (执行查询) (橙色播放按钮) 。
- 作者为 AUTHORNAME 的剩余文章应出现在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "8",
          "title": "A series of posts, Volume 8"
        },
        {
          "id": "5",
          "title": "A series of posts, Volume 5"
        },
        {
          "id": "3",
          "title": "A series of posts, Volume 3"
        },
        {
          "id": "9",
          "title": "A series of posts, Volume 9"
        }
      ],
      "nextToken": null
    }
  }
}
```

使用集

到目前为止，Post 类型一直是平面键/值对象。您也可以使用 AWS AppSync DynamoDB 解析器对复杂对象进行建模，例如集、列表和映射。

让我们更新 Post 类型，加入标签。一篇文章可以具有 0 个或更多标签，这些标签作为字符串集存储在 DynamoDB 中。您还将设置一些变更，用于添加并删除标签；还要用一个新查询扫描具有特定标签的文章。

- 选择架构选项卡。
- 在 Schema (架构) 窗格中修改 Post 类型，添加新的 tags 字段，如下所示：

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
}
```

- 在 Schema (架构) 窗格中修改 Query 类型，添加新的 allPostsByTag 查询，如下所示：

```
type Query {
  allPostsByTag(tag: String!, count: Int, nextToken: String): PaginatedPosts!
  allPostsByAuthor(author: String!, count: Int, nextToken: String): PaginatedPosts!
  allPost(count: Int, nextToken: String): PaginatedPosts!
  getPost(id: ID): Post
}
```

- 在 Schema (架构) 窗格中修改 Mutation 类型，添加新的 addTag 和 removeTag 变更，如下所示：

```
type Mutation {
  addTag(id: ID!, tag: String!): Post
  removeTag(id: ID!, tag: String!): Post
  deletePost(id: ID!, expectedVersion: Int): Post
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
```

```
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}
```

- 选择保存。
- 在右侧的数据类型窗格中，找到 Query 类型上的新创建的 allPostsByTag 字段，然后选择附加。
- 在 Data source name (数据源名称) 中，选择 PostDynamoDBTable。
- 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下内容：

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "filter": {
      "expression": "contains (tags, :tag)",
        "expressionValues": {
          ":tag": $util.dynamodb.toDynamoDBJson($context.arguments.tag)
        }
    }
    #if( ${context.arguments.count} )
        ,"limit": $util.toJson($context.arguments.count)
    #end
    #if( ${context.arguments.nextToken} )
        ,"nextToken": $util.toJson($context.arguments.nextToken)
    #end
}
```

- 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下内容：

```
{
    "posts": $utils.toJson($context.result.items)
```

```
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

- 选择保存。
- 在右侧的数据类型窗格中，找到 Mutation 类型上的新创建的 addTag 字段，然后选择附加。
- 在 Data source name (数据源名称) 中，选择 PostDynamoDBTable。
- 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下内容：

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
        "expression" : "ADD tags :tags, version :plusOne",
        "expressionValues" : {
            ":tags" : { "SS": [ $util.toJson($context.arguments.tag) ] },
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下内容：

```
$utils.toJson($context.result)
```

- 选择保存。
- 在右侧的数据类型窗格中，找到 Mutation 类型上的新创建的 removeTag 字段，然后选择附加。
- 在 Data source name (数据源名称) 中，选择 PostDynamoDBTable。
- 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下内容：

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
    },
    "update" : {
```

```
        "expression" : "DELETE tags :tags ADD version :plusOne",
        "expressionValues" : {
            ":tags" : { "SS": [ $util.toJson($context.arguments.tag) ] },
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

- 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下内容：

```
$utils.toJson($context.result)
```

- 选择保存。

调用 API 以处理标签

您现已设置解析器，AWS AppSync 知道如何将传入的 `addTag`、`removeTag` 和 `allPostsByTag` 请求转换为 DynamoDB `UpdateItem` 和 `Scan` 操作。

我们选择您之前创建的一个文章进行尝试。例如，我们使用作者为 `Nadia` 的一篇文章。

- 选择 Queries 选项卡。
- 在Queries (查询) 窗格中粘贴以下查询：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- 选择 Execute query (执行查询)（橙色播放按钮）。
- Nadia 的全部文章应出现在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
```

```
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

- 让我们使用标题为 "The cutest dog in the world" 的文章。记下其 id，因为您稍后将用到它。

现在，让我们尝试添加一个 dog 标签。

- 在 Queries (查询) 窗格中，粘贴以下变更。您还需要将 id 参数更新为您以前记下的值。

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

- 选择 Execute query (执行查询)（橙色播放按钮）。
- 将使用新标签更新该文章。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
```

```
    }
  }
}
```

您可以添加更多标签，如下所示：

- 更新变更以将 tag 参数更改为 puppy。

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- 选择 Execute query (执行查询)（橙色播放按钮）。
- 将使用新标签更新该文章。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

您也可以删除标签：

- 在 Queries (查询) 窗格中，粘贴以下变更。您还需要将 id 参数更新为您以前记下的值。

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
    title
```

```
    tags
  }
}
```

- 选择 Execute query (执行查询) (橙色播放按钮) 。
- 文章已更新，puppy 标签已删除。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

您也可以搜索所有具有标签的文章：

- 在Queries (查询) 窗格中粘贴以下查询：

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

- 选择 Execute query (执行查询) (橙色播放按钮) 。
- 将返回具有 dog 标签的所有文章，如下所示：

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
```

```
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
            "dog",
            "puppy"
          ]
        }
      ],
      "nextToken": null
    }
  }
}
```

使用列表和映射

除了使用 DynamoDB 集以外，您还可以使用 DynamoDB 列表和映射对单个对象中的复杂数据进行建模。

我们可以为文章添加评论功能。这会建模为 DynamoDB 中的 Post 对象上的映射对象列表。

注意：在真正的应用程序中，您会在评论自身的表中对评论进行建模。在本教程中，您仅将评论添加到 Post 表。

- 选择架构选项卡。
- 在 Schema (架构) 窗格中，添加新的 Comment 类型，如下所示：

```
type Comment {
    author: String!
    comment: String!
}
```

- 在 Schema (架构) 窗格中修改 Post 类型，添加新的 comments 字段，如下所示：

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
```

```
  downs: Int!
  version: Int!
  tags: [String!]
  comments: [Comment!]
}
```

- 在 Schema (架构) 窗格中修改 Mutation 类型，添加新的 addComment 变更，如下所示：

```
type Mutation {
  addComment(id: ID!, author: String!, comment: String!): Post
  addTag(id: ID!, tag: String!): Post
  removeTag(id: ID!, tag: String!): Post
  deletePost(id: ID!, expectedVersion: Int): Post
  upvotePost(id: ID!): Post
  downvotePost(id: ID!): Post
  updatePost(
    id: ID!,
    author: String,
    title: String,
    content: String,
    url: String,
    expectedVersion: Int!
  ): Post
  addPost(
    author: String!,
    title: String!,
    content: String!,
    url: String!
  ): Post!
}
```

- 选择保存。
- 在右侧的数据类型窗格中，找到 Mutation 类型上的新创建的 addComment 字段，然后选择附加。
- 在 Data source name (数据源名称) 中，选择 PostDynamoDBTable。
- 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下内容：

```
{
  "version" : "2017-02-28",
  "operation" : "UpdateItem",
  "key" : {
    "id" : $util.dynamodb.toDynamoDBJson($context.arguments.id)
  },
```

```
  "update" : {
    "expression" : "SET comments =
 list_append(if_not_exists(comments, :emptyList), :newComment) ADD version :plusOne",
    "expressionValues" : {
      ":emptyList": { "L" : [] },
      ":newComment" : { "L" : [
        { "M": {
          "author": $util.dynamodb.toDynamoDBJson($context.arguments.author),
          "comment": $util.dynamodb.toDynamoDBJson($context.arguments.comment)
          }
        }
      ] },
      ":plusOne" : $util.dynamodb.toDynamoDBJson(1)
    }
  }
}
```

此更新表达式将一个列表（包含新评论）追加到现有的 `comments` 列表中。如果这个列表不存在，将创建它。

- 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下内容：

```
$utils.toJson($context.result)
```

- 选择保存。

调用 API 以添加评论

您现已设置解析器，AWS AppSync 知道如何将传入的 `addComment` 请求转换为 DynamoDB `UpdateItem` 操作。

让我们尝试在您已添加标签的文章中添加评论。

- 选择 Queries 选项卡。
- 在Queries (查询) 窗格中粘贴以下查询：

```
mutation addComment {
  addComment(
    id:10
    author: "Steve"
    comment: "Such a cute dog."
  ) {
```

```
    id
    comments {
      author
      comment
    }
  }
}
```

- 选择 Execute query (执行查询) (橙色播放按钮) 。
- Nadia 的全部文章应出现在查询窗格右侧的结果窗格中。如下所示：

```
{
  "data": {
    "addComment": {
      "id": "10",
      "comments": [
        {
          "author": "Steve",
          "comment": "Such a cute dog."
        }
      ]
    }
  }
}
```

如果您多次执行该请求，列表中将追加多条评论。

结论

在本教程中，您构建了一个 API，可用于通过 AWS AppSync 和 GraphQL 处理 DynamoDB 中的 Post 对象。有关更多信息，请参阅解析器映射模板参考。

要进行清理，您可以从控制台中删除 AppSync GraphQL API。

要删除您为本教程创建的 DynamoDB 表和 IAM 角色，您可以运行以下命令以删除 AWSAppSyncTutorialForAmazonDynamoDB 堆栈，或访问 AWS CloudFormation 控制台并删除堆栈：

```
aws cloudformation delete-stack \
    --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

教程：Lambda 解析器

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

您可以将 AWS Lambda 与 AWS AppSync 一起使用以解析任何 GraphQL 字段。例如，GraphQL 查询可能会向 Amazon Relational Database Service (Amazon RDS) 实例发送调用，而 GraphQL 变更可能会写入到 Amazon Kinesis 流。在本节中，我们说明了如何编写 Lambda 函数，以根据 GraphQL 字段操作调用执行业务逻辑。

创建 Lambda 函数

以下示例显示了一个使用 Node.js 编写的 Lambda 函数，该函数对博客文章应用程序包含的博客文章执行各种操作。

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    var posts = {
         "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://
amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR
 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100",
 "downs": "10"},
         "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://
amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups":
 "100", "downs": "10"},
         "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null,
 "content": null, "ups": null, "downs": null },
         "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
         "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} };

    var relatedPosts = {
        "1": [posts['4']],
        "2": [posts['3'], posts['5']],
```

```
        "3": [posts['2'], posts['1']],
        "4": [posts['2'], posts['1']],
        "5": []
    };

    console.log("Got an Invoke Request.");
    switch(event.field) {
        case "getPost":
            var id = event.arguments.id;
            callback(null, posts[id]);
            break;
        case "allPosts":
            var values = [];
            for(var d in posts){
                values.push(posts[d]);
            }
            callback(null, values);
            break;
        case "addPost":
            // return the arguments back
            callback(null, event.arguments);
            break;
        case "addPostErrorWithData":
            var id = event.arguments.id;
            var result = posts[id];
            // attached additional error information to the post
            result.errorMessage = 'Error with the mutation, data has changed';
            result.errorType = 'MUTATION_ERROR';
            callback(null, result);
            break;
        case "relatedPosts":
            var id = event.source.id;
            callback(null, relatedPosts[id]);
            break;
        default:
            callback("Unknown field, unable to resolve" + event.field, null);
            break;
    }
};
```

该 Lambda 函数按 ID 检索文章，添加文章，检索文章列表以及获取给定文章的相关文章。

注意：Lambda 函数对 `event.field` 执行 `switch` 语句以确定当前解析的字段。

使用 AWS 管理控制台或 AWS CloudFormation 堆栈创建该 Lambda 函数。要从 CloudFormation 堆栈中创建函数，您可以使用以下 AWS Command Line Interface (AWS CLI) 命令：

```
aws cloudformation create-stack --stack-name AppSyncLambdaExample \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/lambda/
LambdaCFTemplate.yaml \
--capabilities CAPABILITY_NAMED_IAM
```

也可以从此处在您的 AWS 账户的美国西部（俄勒冈州）AWS 区域中启动 AWS CloudFormation 堆栈：

为 Lambda 配置数据源

在创建 Lambda 函数后，在 AWS AppSync 控制台中导航到您的 GraphQL API，然后选择数据源选项卡。

选择创建数据源，输入友好的数据源名称（例如 **Lambda**），然后为数据源类型选择 AWS Lambda 函数。对于区域，选择与您的函数相同的区域。（如果您从提供的 CloudFormation 堆栈中创建该函数，则该函数可能位于 US-WEST-2 中。）对于函数 ARN，选择您的 Lambda 函数的 Amazon 资源名称 (ARN)。

在选择您的 Lambda 函数后，您可以创建新的 AWS Identity and Access Management (IAM) 角色（AWS AppSync 为其分配相应的权限），或选择具有以下内联策略的现有角色：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/LAMBDA_FUNCTION"
        }
    ]
}
```

您还必须为 IAM 角色建立与 AWS AppSync 信任关系，如下所示：

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

创建 GraphQL 架构

数据源现已连接到您的 Lambda 函数，请创建 GraphQL 架构。

从 AWS AppSync 控制台的架构编辑器中，确保您的架构与以下架构匹配：

```
schema {
    query: Query
    mutation: Mutation
}

type Query {
    getPost(id:ID!): Post
    allPosts: [Post]
}

type Mutation {
    addPost(id: ID!, author: String!, title: String, content: String, url: String):
 Post!
}

type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    relatedPosts: [Post]
```

```
}
```

配置解析器

您现已注册了 Lambda 数据源和有效的 GraphQL 架构，您可以使用解析器将 GraphQL 字段连接到 Lambda 数据源。

要创建解析器，您需要使用映射模板。要了解映射模板的更多信息，请参阅Resolver Mapping Template Overview。

有关 Lambda 映射模板的更多信息，请参阅Resolver mapping template reference for Lambda。

在该步骤中，您将一个解析器附加到以下字段的 Lambda 函数：`getPost(id:ID!): Post`、`allPosts: [Post]`、`addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` 和 `Post.relatedPosts: [Post]`。

从 AWS AppSync 控制台的架构编辑器中，在右侧为 `getPost(id:ID!): Post` 选择附加解析器。

然后，在操作菜单中，选择更新运行时，然后选择单位解析器 (仅限 VTL)。

然后，选择您的 Lambda 数据源。在请求映射模板部分中，选择 Invoke And Forward Arguments (调用并转发参数)。

修改 `payload` 对象，添加字段名称。您的模板应该类似以下内容：

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments":  $utils.toJson($context.arguments)
    }
}
```

在响应映射模板部分中，选择 Return Lambda Result (返回 Lambda 结果)。

在本例中，按原样使用基本模板。它应该类似以下内容：

```
$utils.toJson($context.result)
```

选择保存。您已成功附加了您的首个解析器。针对其余字段重复此操作，如下所示：

对于 `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` 请求映射模板：

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "addPost",
        "arguments":  $utils.toJson($context.arguments)
    }
}
```

对于 `addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!` 响应映射模板：

```
$utils.toJson($context.result)
```

对于 `allPosts: [Post]` 请求映射模板：

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "allPosts"
    }
}
```

对于 `allPosts: [Post]` 响应映射模板：

```
$utils.toJson($context.result)
```

对于 `Post.relatedPosts: [Post]` 请求映射模板：

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
```

```
        "field": "relatedPosts",
        "source":  $utils.toJson($context.source)
    }
}
```

对于 `Post.relatedPosts: [Post]` 响应映射模板：

```
$utils.toJson($context.result)
```

测试您的 GraphQL API

现在您的 Lambda 函数已与 GraphQL 解析器连接，您可以使用控制台或客户端应用程序运行一些变更和查询。

在 AWS AppSync 控制台左侧，选择查询，然后粘贴以下代码：

addPost 变更

```
mutation addPost {
    addPost(
        id: 6
        author: "Author6"
        title: "Sixth book"
        url: "https://www.amazon.com/"
        content: "This is the book is a tutorial for using GraphQL with AWS AppSync."
    ) {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

getPost 查询

```
query getPost {
    getPost(id: "2") {
```

```
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

allPosts 查询

```
query allPosts {
    allPosts {
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {
            id
            title
        }
    }
}
```

返回错误

解析任何给定的字段可能会导致错误。在使用 AWS AppSync 时，您可以从以下源中引发错误：

- 请求或响应映射模板
- Lambda 函数

从映射模板

要故意引发错误，您可以使用 Velocity 模板语言 (VTL) 模板中的 `$utils.error` 帮助程序方法。它可接收 `errorMessage`、`errorType` 以及可选的 `data` 值作为参数。出现错误后，`data` 对于将额外的数据返回客户端很有用。在 GraphQL 最终响应中，`data` 对象将添加到 `errors`。

以下示例显示了如何在 Post.relatedPosts: [Post] 响应映射模板中使用它：

```
$utils.error("Failed to fetch relatedPosts", "LambdaFailure", $context.result)
```

这将生成与以下内容类似的 GraphQL 响应：

```
{
    "data": {
        "allPosts": [
            {
                "id": "2",
                "title": "Second book",
                "relatedPosts": null
            },
            ...
        ]
    },
    "errors": [
        {
            "path": [
                "allPosts",
                0,
                "relatedPosts"
            ],
            "errorType": "LambdaFailure",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ],
            "message": "Failed to fetch relatedPosts",
            "data": [
                {
                  "id": "2",
                  "title": "Second book"
                },
                {
                  "id": "1",
                  "title": "First book"
                }
            ]
        }
```

```
    ]
}
```

错误导致 `allPosts[0].relatedPosts` 为 null，而 `errorMessage`、`errorType` 和 `data` 体现在 `data.errors[0]` 对象中。

从 Lambda 函数

AWS AppSync 还可以识别 Lambda 函数引发的错误。Lambda 编程模型允许您引发处理的 错误。如果 Lambda 函数引发错误，则 AWS AppSync 无法解析当前字段。仅在响应中设置从 Lambda 返回的错误消息。目前，您无法通过从 Lambda 函数中引发错误，将任何无关数据传回到客户端。

注意：如果您的 Lambda 函数引发未处理的 错误，AWS AppSync 将使用 Lambda 设置的错误消息。

以下 Lambda 函数会引发错误：

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    callback("I fail. Always.");
};
```

这将返回与以下内容类似的 GraphQL 响应：

```
{
    "data": {
        "allPosts": [
            {
                "id": "2",
                "title": "Second book",
                "relatedPosts": null
            },
            ...
        ]
    },
    "errors": [
        {
            "path": [
                "allPosts",
                0,
                "relatedPosts"
            ],
            "errorType": "Lambda:Handled",
            "locations": [
```

```
                {
                    "line": 5,
                    "column": 5
                }
            ],
            "message": "I fail. Always."
        }
    ]
}
```

高级使用案例：批处理

该示例中的 Lambda 函数具有一个 `relatedPosts` 字段，它返回给定文章的相关文章列表。在示例查询中，从 Lambda 函数中调用 `allPosts` 字段将返回 5 篇文章。由于我们指定还希望为每个返回的文章解析 `relatedPosts`，因此，将 `relatedPosts` 字段操作调用 5 次。

```
query allPosts {
    allPosts {   // 1 Lambda invocation - yields 5 Posts
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {   // 5 Lambda invocations - each yields 5 posts
            id
            title
        }
    }
}
```

虽然这在该特定示例中听起来可能并不严重，但这种累积的过度获取可能会迅速降低应用程序的性能。

如果您要针对同一查询中返回的相关 Posts 再次提取 `relatedPosts`，调用数量将显著增加。

```
query allPosts {
    allPosts {   // 1 Lambda invocation - yields 5 Posts
        id
        author
        title
        content
```

```
        url
        ups
        downs
        relatedPosts {   // 5 Lambda invocations - each yield 5 posts = 5 x 5 Posts
            id
            title
            relatedPosts {  // 5 x 5 Lambda invocations - each yield 5 posts = 25 x 5
 Posts
                id
                title
                author
            }
        }
    }
}
```

在这个相对简单的查询中，AWS AppSync 将调用 Lambda 函数 1 + 5 + 25 = 31 次。

这是相当常见的挑战，常被称为 N+1 问题（在本例中 N = 5），会导致延迟增加，以及应用程序费用提升。

我们解决此问题的方式是批处理类似的字段解析器请求。在该示例中，Lambda 函数可能会解析给定批次的文章的相关文章列表，而不是让 Lambda 函数解析单个给定文章的相关文章列表。

为了演示此操作，让我们将 `Post.relatedPosts: [Post]` 解析器转换为启用批处理的解析器。

在 AWS AppSync 控制台右侧，选择现有的 `Post.relatedPosts: [Post]` 解析器。将请求映射模板改为以下内容：

```
{
    "version": "2017-02-28",
    "operation": "BatchInvoke",
    "payload": {
        "field": "relatedPosts",
        "source":  $utils.toJson($context.source)
    }
}
```

只有 `operation` 字段由 `Invoke` 改为了 `BatchInvoke`。负载字段现在成为模板中指定的任何内容的数组。在该示例中，Lambda 函数收到以下内容以作为输入：

```
[
```

```
    {
        "field": "relatedPosts",
        "source": {
            "id": 1
        }
    },
    {
        "field": "relatedPosts",
        "source": {
            "id": 2
        }
    },
    ...
]
```

如果在请求映射模板中指定了 `BatchInvoke`，Lambda 函数将收到请求列表并返回结果列表。

具体来说，结果列表必须与请求负载条目的大小和顺序匹配，以使 AWS AppSync 可以相应地匹配结果。

在该批处理示例中，Lambda 函数返回一批结果，如下所示：

```
[
    [{"id":"2","title":"Second book"}, {"id":"3","title":"Third book"}],   //
 relatedPosts for id=1
    [{"id":"3","title":"Third book"}]
           // relatedPosts for id=2
]
```

使用 Node.js 编写的以下 Lambda 函数说明了 `Post.relatedPosts` 字段的批处理功能，如下所示：

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    var posts = {
         "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://
amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR
 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100",
 "downs": "10"},
         "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://
amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups":
 "100", "downs": "10"},
         "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null,
 "content": null, "ups": null, "downs": null },
```

```
         "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
         "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} };

    var relatedPosts = {
        "1": [posts['4']],
        "2": [posts['3'], posts['5']],
        "3": [posts['2'], posts['1']],
        "4": [posts['2'], posts['1']],
        "5": []
    };

    console.log("Got a BatchInvoke Request. The payload has %d items to resolve.",
 event.length);
    // event is now an array
    var field = event[0].field;
    switch(field) {
        case "relatedPosts":
            var results = [];
            // the response MUST contain the same number
            // of entries as the payload array
            for (var i=0; i< event.length; i++) {
                console.log("post {}", JSON.stringify(event[i].source));
                results.push(relatedPosts[event[i].source.id]);
            }
            console.log("results {}", JSON.stringify(results));
            callback(null, results);
            break;
        default:
            callback("Unknown field, unable to resolve" + field, null);
            break;
    }
};
```

返回单个错误

以前的示例表明，可以从 Lambda 函数中返回单个错误，或者从映射模板中引发错误。对于批处理调用，从 Lambda 函数中引发错误会将整个批次标记为失败。对于发生不可恢复错误的特定场景（例如，到数据存储的连接失败），这可能是可以接受的。不过，如果批次中的某些项目成功，而其他项目

失败，则可能会同时返回错误和有效的数据。由于 AWS AppSync 要求批处理响应列出与批次的原始大小匹配的元素，因此，您必须定义一个可以区分有效数据和错误的数据结构。

例如，如果 Lambda 函数预计返回一批相关文章，您可以选择返回 Response 对象列表，其中每个对象具有可选的 data、errorMessage 和 errorType 字段。如果出现 errorMessage 字段，则表示出现错误。

以下代码说明了如何更新 Lambda 函数：

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    var posts = {
         "1": {"id": "1", "title": "First book", "author": "Author1", "url": "https://
amazon.com/", "content": "SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR
 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1", "ups": "100",
 "downs": "10"},
         "2": {"id": "2", "title": "Second book", "author": "Author2", "url": "https://
amazon.com", "content": "SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT", "ups":
 "100", "downs": "10"},
         "3": {"id": "3", "title": "Third book", "author": "Author3", "url": null,
 "content": null, "ups": null, "downs": null },
         "4": {"id": "4", "title": "Fourth book", "author": "Author4", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4", "ups": "1000", "downs": "0"},
         "5": {"id": "5", "title": "Fifth book", "author": "Author5", "url": "https://
www.amazon.com/", "content": "SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT
 AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT", "ups": "50", "downs": "0"} };

    var relatedPosts = {
        "1": [posts['4']],
        "2": [posts['3'], posts['5']],
        "3": [posts['2'], posts['1']],
        "4": [posts['2'], posts['1']],
        "5": []
    };

    console.log("Got a BatchInvoke Request. The payload has %d items to resolve.",
 event.length);
    // event is now an array
    var field = event[0].field;
    switch(field) {
        case "relatedPosts":
```

```
            var results = [];
            results.push({ 'data': relatedPosts['1'] });
            results.push({ 'data': relatedPosts['2'] });
            results.push({ 'data': null, 'errorMessage': 'Error Happened', 'errorType':
 'ERROR' });
            results.push(null);
            results.push({ 'data': relatedPosts['3'], 'errorMessage': 'Error Happened
 with last result', 'errorType': 'ERROR' });
            callback(null, results);
            break;
        default:
            callback("Unknown field, unable to resolve" + field, null);
            break;
    }
};
```

对于该示例，以下响应映射模板解析 Lambda 函数的每个项目，并引发发生的任何错误：

```
#if( $context.result && $context.result.errorMessage )
    $utils.error($context.result.errorMessage, $context.result.errorType,
 $context.result.data)
#else
    $utils.toJson($context.result.data)
#end
```

此示例返回与以下内容类似的 GraphQL 响应：

```
{
  "data": {
    "allPosts": [
      {
        "id": "1",
        "relatedPostsPartialErrors": [
          {
            "id": "4",
            "title": "Fourth book"
          }
        ]
      },
      {
        "id": "2",
        "relatedPostsPartialErrors": [
          {
```

```
            "id": "3",
            "title": "Third book"
          },
          {
            "id": "5",
            "title": "Fifth book"
          }
        ]
      },
      {
        "id": "3",
        "relatedPostsPartialErrors": null
      },
      {
        "id": "4",
        "relatedPostsPartialErrors": null
      },
      {
        "id": "5",
        "relatedPostsPartialErrors": null
      }
    ]
  },
  "errors": [
    {
      "path": [
        "allPosts",
        2,
        "relatedPostsPartialErrors"
      ],
      "errorType": "ERROR",
      "locations": [
        {
          "line": 4,
          "column": 9
        }
      ],
      "message": "Error Happened"
    },
    {
      "path": [
        "allPosts",
        4,
        "relatedPostsPartialErrors"
```

```
      ],
      "data": [
        {
          "id": "2",
          "title": "Second book"
        },
        {
          "id": "1",
          "title": "First book"
        }
      ],
      "errorType": "ERROR",
      "locations": [
        {
          "line": 4,
          "column": 9
        }
      ],
      "message": "Error Happened with last result"
    }
  ]
}
```

配置最大批处理大小

默认情况下，在使用 BatchInvoke 时，AWS AppSync 向您的 Lambda 函数批量发送请求（每个批次最多 5 个项目）。您可以配置 Lambda 解析器的最大批次大小。

要配置解析器上的最大批处理大小，请在 AWS Command Line Interface (AWS CLI) 中使用以下命令：

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
 relatedPosts \
 --request-mapping-template "<template>" --response-mapping-template "<template>" --
data-source-name "<lambda-datasource>" \
 --max-batch-size X
```

> **Note**
>
> 在提供请求映射模板时，您必须使用 BatchInvoke 操作才能使用批处理。

您也可以使用以下命令，在直接 Lambda 解析器上启用和配置批处理：

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
 relatedPosts \
 --data-source-name "<lambda-datasource>" \
 --max-batch-size X
```

使用 VTL 模板配置最大批处理大小

对于具有 VTL 请求中模板的 Lambda 解析器，最大批次大小无效，除非它们在 VTL 中直接将其指定为 `BatchInvoke` 操作。同样，如果您执行顶级变更，则不会为变更执行批处理，因为 GraphQL 规范要求按顺序执行并行变更。

例如，采用以下变更：

```
type Mutation {
    putItem(input: Item): Item
    putItems(inputs: [Item]): [Item]
}
```

通过使用第一个变更，我们可以创建 10 个 `Items`，如下面的代码片段所示：

```
mutation MyMutation {
    v1: putItem($someItem1) {
        id,
        name
    }
    v2: putItem($someItem2) {
        id,
        name
    }
    v3: putItem($someItem3) {
        id,
        name
    }
    v4: putItem($someItem4) {
        id,
        name
    }
    v5: putItem($someItem5) {
        id,
```

```
        name
    }
    v6: putItem($someItem6) {
        id,
        name
    }
    v7: putItem($someItem7) {
        id,
        name
    }
    v8: putItem($someItem8) {
        id,
        name
    }
    v9: putItem($someItem9) {
        id,
        name
    }
    v10: putItem($someItem10) {
        id,
        name
    }
}
```

在该示例中，即使在 Lambda 解析器中将最大批次大小设置为 10，也不会以 10 为一组对 `Items` 进行批处理，而是根据 GraphQL 规范按顺序执行它们。

要执行实际的批处理变更，您可以按照以下示例使用第二个变更：

```
mutation MyMutation {
    putItems([$someItem1, $someItem2, $someItem3,$someItem4, $someItem5, $someItem6,
    $someItem7, $someItem8, $someItem9, $someItem10]) {
    id,
    name
    }
}
```

有关使用直接 Lambda 解析器进行批处理的更多信息，请参阅直接 Lambda 解析器。

教程：Amazon OpenSearch Service 解析器

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

AWS AppSync 支持从您在自己的 AWS 账户中预置的域中使用 Amazon OpenSearch Service，但前提是这些域在 VPC 中不存在。预置域后，您可以使用数据源连接这些域，此时，您可以在架构中配置一个解析器以执行 GraphQL 操作（如查询、变更和订阅）。本教程将引导您了解一些常见示例。

有关更多信息，请参阅适用于 OpenSearch 的解析器映射模板参考。

一键设置

要在配置了 Amazon OpenSearch Service 的 AWS AppSync 中自动设置 GraphQL 终端节点，您可以使用以下 AWS CloudFormation 模板：

AWS CloudFormation 部署完成后，可以直接跳到运行 GraphQL 查询和变更。

创建新的 OpenSearch Service 域

要开始学习本教程，您需要具有一个现有的 OpenSearch Service 域。如果您没有域，可以使用以下示例。请注意，创建 OpenSearch Service 域最多可能需要 15 分钟，然后您才能继续将其与 AWS AppSync 数据源集成在一起。

```
aws cloudformation create-stack --stack-name AppSyncOpenSearch \
--template-url https://s3.us-west-2.amazonaws.com/awsappsync/resources/elasticsearch/
ESResolverCFTemplate.yaml \
--parameters ParameterKey=OSDomainName,ParameterValue=ddtestdomain
 ParameterKey=Tier,ParameterValue=development \
--capabilities CAPABILITY_NAMED_IAM
```

可以在您的 AWS 账户的 US West 2（俄勒冈州）区域中启动以下 AWS CloudFormation 堆栈：

为 OpenSearch Service 配置数据源

在创建 OpenSearch Service 域后，导航到 AWS AppSync GraphQL API 并选择数据源选项卡。选择新建，并为数据源输入一个友好名称，例如“oss”。然后，为数据源类型选择 Amazon OpenSearch 域，选择相应的区域，您应该会看到列出了您的 OpenSearch Service 域。在选择该域后，您可以创建一个新角色，AWS AppSync 为其分配相应的角色权限，您也可以选择一个现有角色，该角色具有以下内联策略：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Stmt1234234",
            "Effect": "Allow",
            "Action": [
                "es:ESHttpDelete",
                "es:ESHttpHead",
                "es:ESHttpGet",
                "es:ESHttpPost",
                "es:ESHttpPut"
            ],
            "Resource": [
                "arn:aws:es:REGION:ACCOUNTNUMBER:domain/democluster/*"
            ]
        }
    ]
}
```

您还需要为该角色建立与 AWS AppSync 的信任关系：

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

此外，OpenSearch Service 域具有自己的访问策略，您可以通过 Amazon OpenSearch Service 控制台修改该策略。您需要添加一个类似于下面的策略，并具有 OpenSearch Service 域的相应操作和资源。请注意，主体是 AppSync 数据源角色，如果您让控制台创建该角色，则可以在 IAM 控制台中找到该角色。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
        "Effect": "Allow",
        "Principal": {
            "AWS": "arn:aws:iam::ACCOUNTNUMBER:role/service-role/
APPSYNC_DATASOURCE_ROLE"
        },
        "Action": [
            "es:ESHttpDelete",
            "es:ESHttpHead",
            "es:ESHttpGet",
            "es:ESHttpPost",
            "es:ESHttpPut"
        ],
        "Resource": "arn:aws:es:REGION:ACCOUNTNUMBER:domain/DOMAIN_NAME/*"
        }
    ]
}
```

连接解析器

数据源现已连接到您的 OpenSearch Service 域，您可以使用解析器将其连接到您的 GraphQL 架构，如以下示例中所示：

```
 schema {
   query: Query
   mutation: Mutation
 }

 type Query {
   getPost(id: ID!): Post
   allPosts: [Post]
 }

 type Mutation {
```

```
   addPost(id: ID!, author: String, title: String, url: String, ups: Int, downs: Int,
 content: String): AWSJSON
 }

type Post {
  id: ID!
  author: String
  title: String
  url: String
  ups: Int
  downs: Int
  content: String
}
...
```

请注意，有一个用户定义的 Post 类型（具有一个 id 字段）。在以下示例中，我们假设通过一个流程（可以自动完成）将该类型放入您的 OpenSearch Service 域中，这会映射到 /post/_doc 的路径根，其中 post 是索引。从该根路径中，您可以执行单独文档搜索、使用 /id/post* 的通配符搜索或使用 /post/_search 路径的多文档搜索。例如，如果您具有另一个名为 User 的类型，您可以使用名为 user 的新索引对文档编制索引，然后使用 /user/_search 路径执行搜索。

从 AWS AppSync 控制台的架构编辑器中，修改前面的 Posts 架构以包含 searchPosts 查询：

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

保存架构。在右侧，对于 searchPosts，选择 Attach resolver (附加解析器)。在操作菜单中，选择更新运行时，然后选择单位解析器 (仅限 VTL)。然后，选择您的 OpenSearch Service 数据源。在 request mapping template (请求映射模板) 部分下，选择 Query posts (查询文章) 的下拉列表以获取基本模板。将 path 修改为 /post/_search。它应该类似以下内容：

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/post/_search",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
```

```
            "from":0,
            "size":50
        }
    }
}
```

这假设前面的架构的文档已在 OpenSearch Service 中使用 post 字段编制索引。如果您以不同方式设置数据的结构，将需要相应地进行更新。

在响应映射模板部分下面，如果要从 OpenSearch Service 查询中获取数据结果并将其转换为 GraphQL，则需要指定相应的 _source 筛选条件。使用以下模板：

```
[
    #foreach($entry in $context.result.hits.hits)
    #if( $velocityCount > 1 ) , #end
    $utils.toJson($entry.get("_source"))
    #end
]
```

修改您的搜索

上述的请求映射模板针对所有记录执行一个简单查询。假设您想要按某个特定作者进行搜索。此外，假设您希望该作者是在 GraphQL 查询中定义的一个参数。在 AWS AppSync 控制台的架构编辑器中，添加一个 allPostsByAuthor 查询：

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

现在选择附加解析器并选择 OpenSearch Service 数据源，但在响应映射模板中使用以下示例：

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/post/_search",
    "params":{
        "headers":{},
        "queryString":{},
```

```
        "body":{
            "from":0,
            "size":50,
            "query":{
                "match" :{
                    "author": $util.toJson($context.arguments.author)
                }
            }
        }
    }
}
```

请注意，body 用一个针对 author 字段的术语查询填充，它将作为一个参数从客户端进行传递。您可以选择已预填充信息（如标准文本），甚至使用其他实用程序。

如果您使用此解析器，则使用上例所示的相同信息填写响应映射模板。

将数据添加到 OpenSearch Service

您可能希望将数据添加到您的 OpenSearch Service 域以作为 GraphQL 变更结果。这是一个用于搜索和其他用途的强大机制。由于您可以使用 GraphQL 订阅实时获取数据，因此，它作为一种机制向客户端通知 OpenSearch Service 域中的数据更新。

返回到 AWS AppSync 控制台中的架构页面，然后为 addPost() 变更选择附加解析器。再次选择 OpenSearch Service 数据源，并将以下响应映射模板用于 Posts 架构：

```
{
    "version":"2017-02-28",
    "operation":"PUT",
    "path": $util.toJson("/post/_doc/$context.arguments.id"),
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "id": $util.toJson($context.arguments.id),
            "author": $util.toJson($context.arguments.author),
            "ups": $util.toJson($context.arguments.ups),
            "downs": $util.toJson($context.arguments.downs),
            "url": $util.toJson($context.arguments.url),
            "content": $util.toJson($context.arguments.content),
            "title": $util.toJson($context.arguments.title)
        }
    }
```

```
}
```

和之前一样，这是一个介绍如何设置数据的结构的示例。如果您有不同的字段名称或索引，则需要相应地更新 path 和 body。此示例还显示如何使用 $context.arguments 从您的 GraphQL 变更参数填充模板。

在继续之前，使用以下响应映射模板，这会返回变更操作结果或错误信息以作为输出：

```
#if($context.error)
    $util.toJson($ctx.error)
#else
    $util.toJson($context.result)
#end
```

检索单个文档

最后，如果要在架构中使用 getPost(id:ID) 查询以返回单个文档，请在 AWS AppSync 控制台的架构编辑器中找到该查询，然后选择附加解析器。再次选择 OpenSearch Service 数据源，并使用以下映射模板：

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path": $util.toJson("post/_doc/$context.arguments.id"),
    "params":{
        "headers":{},
        "queryString":{},
        "body":{}
    }
}
```

由于上面的 path 将 id 参数用于空正文，此命令将返回单个文档。但是，您需要使用以下响应映射模板，因为现在您返回的是单个项目而不是列表：

```
$utils.toJson($context.result.get("_source"))
```

执行查询和变更

您现在应该能够对 OpenSearch Service 域执行 GraphQL 操作。导航到 AWS AppSync 控制台的查询选项卡，并添加一个新记录：

```
mutation addPost {
    addPost (
        id:"12345"
        author: "Fred"
        title: "My first book"
        content: "This will be fun to write!"
        url: "publisher website",
        ups: 100,
        downs:20
       )
}
```

您将会在右侧看到变更结果。同样，您现在可以对您的 OpenSearch Service 域运行 `searchPosts` 查询：

```
query searchPosts {
    searchPosts {
        id
        title
        author
        content
    }
}
```

最佳实操

- 应将 OpenSearch Service 用于查询数据，而不是作为主数据库。您可能希望将 OpenSearch Service 与 Amazon DynamoDB 一起使用，如组合使用 GraphQL 解析器中所述。
- 通过允许 AWS AppSync 服务角色访问集群，仅授予您的域的访问权限。
- 您可以通过最低成本的集群先开始小规模开发，然后随着您转向生产阶段，而转至具有高可用性 (HA) 的较大集群。

教程：本地解析器

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

AWS AppSync 允许您使用支持的数据源（AWS Lambda、Amazon DynamoDB 或 Amazon OpenSearch Service）执行各种操作。但在某些情况下，可能不必调用支持的数据源。

这时本地解析器就很方便。本地解析器无需调用远程数据源，只需将请求映射模板的结果转发到响应映射模板。字段解析是在 AWS AppSync 中完成的。

在许多使用案例中本地解析器都很有用。最常用的使用案例是在不触发数据源调用的情况下发布通知。要演示此使用案例，让我们构建一个传呼应用程序，用户可互相呼叫。此示例利用了订阅，如果您不熟悉订阅，可以参考实时数据教程。

创建呼叫应用程序

在我们的呼叫应用程序中，客户端可以订阅收件箱，并向其他客户端发送呼叫。每个呼叫包含一条消息。以下是架构：

```
schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
}

type Subscription {
    inbox(to: String!): Page
    @aws_subscribe(mutations: ["page"])
}

type Mutation {
    page(body: String!, to: String!): Page!
}

type Page {
    from: String
    to: String!
    body: String!
    sentAt: String!
}

type Query {
    me: String
}
```

让我们在 Mutation.page 字段上附加一个解析器。在 Schema (架构) 窗格中，单击右侧面板中字段定义旁的 Attach Resolver (附加解析器)。创建类型为 None 的新数据源，并命名为 PageDataSource。

在请求映射模板中输入：

```
{
  "version": "2017-02-28",
  "payload": {
    "body": $util.toJson($context.arguments.body),
    "from": $util.toJson($context.identity.username),
    "to":  $util.toJson($context.arguments.to),
    "sentAt": "$util.time.nowISO8601()"
  }
}
```

对于响应映射模板，选择默认的 Forward the result (转发结果)。保存解析器。您的应用程序现在已经就绪，让我们开始呼叫吧！

发送呼叫和订阅呼叫

接收呼叫的客户端必须首先订阅收件箱。

让我们在 Queries (查询) 窗格中执行 inbox 订阅：

```
subscription Inbox {
    inbox(to: "Nadia") {
        body
        to
        from
        sentAt
    }
}
```

每次调用 Mutation.page 变更时，Nadia 都会收到页面。执行变更可进行调用：

```
mutation Page {
    page(to: "Nadia", body: "Hello, World!") {
        body
        to
        from
        sentAt
```

```
    }
}
```

我们刚刚说明了如何使用本地解析器，即，在不离开 AWS AppSync 的情况下发送和接收页面。

教程：组合使用 GraphQL 解析器

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

GraphQL 架构中的解析器和字段具有 1:1 的关系，具有很高的灵活性。由于数据源是独立于架构在解析器上配置的，因此您可以通过不同的数据源解析和操控 GraphQL 类型，同时在架构上混合和匹配数据源以最好地满足您的需要。

以下示例场景说明了如何在架构中混合使用和匹配数据源。在开始之前，我们建议您熟悉如何设置数据源以及 AWS Lambda、Amazon DynamoDB 和 Amazon OpenSearch Service 的解析器，如前面的教程中所述。

示例架构

以下架构具有一个名为 `Post` 的类型，它定义了 3 个 `Query` 操作和 3 个 `Mutation` 操作：

```
type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    version: Int!
}

type Query {
    allPost: [Post]
    getPost(id: ID!): Post
    searchPosts: [Post]
}
```

```
type Mutation {
    addPost(
        id: ID!,
        author: String!,
        title: String,
        content: String,
        url: String
    ): Post
    updatePost(
        id: ID!,
        author: String!,
        title: String,
        content: String,
        url: String,
        ups: Int!,
        downs: Int!,
        expectedVersion: Int!
    ): Post
    deletePost(id: ID!): Post
}
```

在本示例中，您共有 6 个可附加的解析器。一种可能的方法是，让所有这些解析器来自名为 `Posts` 的 Amazon DynamoDB 表，其中 `AllPosts` 运行扫描，`searchPosts` 运行查询，如 DynamoDB 解析器映射模板参考中所述。不过，可以使用替代方法满足您的业务需求，例如从 Lambda 或 OpenSearch Service 中解析这些 GraphQL 查询。

通过解析器更改数据

您可能需要将结果从数据库（例如 DynamoDB 或 Amazon Aurora）返回到客户端，并更改某些属性。这可能是由于数据类型的格式设置所致（例如，客户端上的时间戳差异），或者是为了处理向后兼容性问题。出于说明目的，在以下示例中，每次调用 GraphQL 解析器时，AWS Lambda 函数为其分配随机数以处理博客文章的点赞和差评操作：

```
'use strict';
const doc = require('dynamodb-doc');
const dynamo = new doc.DynamoDB();

exports.handler = (event, context, callback) => {
    const payload = {
        TableName: 'Posts',
        Limit: 50,
```

```
        Select: 'ALL_ATTRIBUTES',
    };

    dynamo.scan(payload, (err, data) => {
        const result = { data: data.Items.map(item =>{
            item.ups = parseInt(Math.random() * (50 - 10) + 10, 10);
            item.downs = parseInt(Math.random() * (20 - 0) + 0, 10);
            return item;
        }) };
        callback(err, result.data);
    });
};
```

这是一个完全有效的 Lambda 函数，可以附加到 GraphQL 架构中的 `AllPosts` 字段，以便返回所有结果的任何查询都可以对顶/踩操作获得随机数字。

DynamoDB 和 OpenSearch Service

对于某些应用程序，您可能会对 DynamoDB 执行变更或简单查找查询，并让后台进程将文档传输到 OpenSearch Service。然后，您可以简单地将 `searchPosts` 解析器附加到 OpenSearch Service 数据源，并使用 GraphQL 查询返回搜索结果（从源自 DynamoDB 的数据）。在应用程序中添加高级搜索操作（例如关键字、模糊字词匹配，甚至地理空间查找）时，这可能是非常强大的。可以通过 ETL 流程完成从 DynamoDB 传输数据的过程，也可以使用 Lambda 从 DynamoDB 进行流式传输。可以在您的 AWS 账户的 US West 2（俄勒冈州）区域中通过以下 AWS CloudFormation 堆栈启动完整的示例：

该示例中的架构允许您使用 DynamoDB 解析器添加文章，如下所示：

```
mutation add {
    putPost(author:"Nadia"
        title:"My first post"
        content:"This is some test content"
        url:"https://aws.amazon.com/appsync/"
    ){
        id
        title
    }
}
```

这会将数据写入到 DynamoDB 中，然后 DynamoDB 通过 Lambda 将数据流式传输到 Amazon OpenSearch Service，您可以使用该服务按不同字段搜索文章。例如，由于数据位于 Amazon OpenSearch Service 中，您可以使用自由格式文本（甚至包含空格）搜索作者或内容字段，如下所示：

```
query searchName{
    searchAuthor(name:"   Nadia   "){
        id
        title
        content
    }
}

query searchContent{
    searchContent(text:"test"){
        id
        title
        content
    }
}
```

由于数据直接写入到 DynamoDB 中，因此，您仍然可以使用 `allPosts{...}` 和 `singlePost{...}` 查询对表执行高效的列表或项目查找操作。该堆栈将以下示例代码用于 DynamoDB 流：

注意：此代码仅用于举例说明。

```
var AWS = require('aws-sdk');
var path = require('path');
var stream = require('stream');

var esDomain = {
    endpoint: 'https://opensearch-domain-name.REGION.es.amazonaws.com',
    region: 'REGION',
    index: 'id',
    doctype: 'post'
};

var endpoint = new AWS.Endpoint(esDomain.endpoint)
var creds = new AWS.EnvironmentCredentials('AWS');

function postDocumentToES(doc, context) {
```

```
    var req = new AWS.HttpRequest(endpoint);

    req.method = 'POST';
    req.path = '/_bulk';
    req.region = esDomain.region;
    req.body = doc;
    req.headers['presigned-expires'] = false;
    req.headers['Host'] = endpoint.host;

    // Sign the request (Sigv4)
    var signer = new AWS.Signers.V4(req, 'es');
    signer.addAuthorization(creds, new Date());

    // Post document to ES
    var send = new AWS.NodeHttpClient();
    send.handleRequest(req, null, function (httpResp) {
        var body = '';
        httpResp.on('data', function (chunk) {
            body += chunk;
        });
        httpResp.on('end', function (chunk) {
            console.log('Successful', body);
            context.succeed();
        });
    }, function (err) {
        console.log('Error: ' + err);
        context.fail();
    });
}

exports.handler = (event, context, callback) => {
    console.log("event => " + JSON.stringify(event));
    var posts = '';

    for (var i = 0; i < event.Records.length; i++) {
        var eventName = event.Records[i].eventName;
        var actionType = '';
        var image;
        var noDoc = false;
        switch (eventName) {
            case 'INSERT':
                actionType = 'create';
                image = event.Records[i].dynamodb.NewImage;
                break;
```

```
            case 'MODIFY':
                actionType = 'update';
                image = event.Records[i].dynamodb.NewImage;
                break;
            case 'REMOVE':
            actionType = 'delete';
                image = event.Records[i].dynamodb.OldImage;
                noDoc = true;
                break;
        }

        if (typeof image !== "undefined") {
            var postData = {};
            for (var key in image) {
                if (image.hasOwnProperty(key)) {
                    if (key === 'postId') {
                        postData['id'] = image[key].S;
                    } else {
                        var val = image[key];
                        if (val.hasOwnProperty('S')) {
                            postData[key] = val.S;
                        } else if (val.hasOwnProperty('N')) {
                            postData[key] = val.N;
                        }
                    }
                }
            }

            var action = {};
            action[actionType] = {};
            action[actionType]._index = 'id';
            action[actionType]._type = 'post';
            action[actionType]._id = postData['id'];
            posts += [
                JSON.stringify(action),
            ].concat(noDoc?[]:[JSON.stringify(postData)]).join('\n') + '\n';
        }
    }
    console.log('posts:',posts);
    postDocumentToES(posts, context);
};
```

然后，您可以使用 DynamoDB 流将其附加到主键为 `id` 的 DynamoDB 表，并将对 DynamoDB 源的任何更改流式传输到您的 OpenSearch Service 域。有关配置上述功能的更多信息，请参阅 DynamoDB 流文档。

教程：DynamoDB 批处理解析器

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

AWS AppSync 支持对单个区域中的一个或多个表执行 Amazon DynamoDB 批处理操作。支持的操作为 `BatchGetItem`、`BatchPutItem` 和 `BatchDeleteItem`。通过在 AWS AppSync 中使用这些功能，您可以执行如下任务：

- 在单个查询中传递键列表，并从表中返回结果
- 在单个查询中从一个或多个表读取记录
- 将记录批量写入一个或多个表
- 有条件写入或删除多个可能有关系的表中的记录

在 AWS AppSync 中将批处理操作用于 DynamoDB 是一项高级技术，需要对后端操作和表结构有所了解。此外，AWS AppSync 中的批处理操作与非批处理操作有两个主要区别：

- 数据源角色必须对解析器将访问的所有表具有权限。
- 解析器的表规范是映射模板的一部分。

权限

与其他解析器一样，您需要在 AWS AppSync 中创建数据源，并创建一个角色或使用现有的角色。由于批处理操作需要具有 DynamoDB 表的不同权限，因此，您需要为配置的角色授予读取或写入操作权限：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Action": [
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:region:account:table/TABLENAME",
                "arn:aws:dynamodb:region:account:table/TABLENAME/*"
            ]
        }
    ]
}
```

注意：角色与 AWS AppSync 中的数据源相关联，并对数据源调用字段上的解析器。配置为针对 DynamoDB 获取的数据源仅指定一个表，以使配置保持简单。因此，当在单个解析器中针对多个表执行批处理操作（这是一项更高级的任务）时，您必须向该数据源的角色授予对将与解析器进行交互的任何表的访问权限。这项操作将在上面 IAM 策略中的 Resource (资源) 字段中执行。对表进行配置以便对它们进行批处理调用是在解析器模板中实现的，下面将介绍相关内容。

数据源

为简便起见，我们将为本教程中使用的所有解析器使用同一个数据源。在数据源选项卡上，创建一个新的 DynamoDB 数据源，并将其命名为 BatchTutorial。表名称可以是任何内容，因为表名称被指定为批处理操作的请求映射模板的一部分。我们将提供表名称 `empty`。

对于本教程，具有以下内联策略的任何角色都有效：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:BatchGetItem",
                "dynamodb:BatchWriteItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:region:account:table/Posts",
                "arn:aws:dynamodb:region:account:table/Posts/*",
                "arn:aws:dynamodb:region:account:table/locationReadings",
                "arn:aws:dynamodb:region:account:table/locationReadings/*",
                "arn:aws:dynamodb:region:account:table/temperatureReadings",
```

```
                "arn:aws:dynamodb:region:account:table/temperatureReadings/*"
            ]
        }
    ]
}
```

单个表批处理

在此示例中，假设您有一个名为 Posts 的表，您要通过批处理操作在其中添加和删除项目。使用以下架构，注意对于此查询，我们传入一个 ID 列表：

```
type Post {
    id: ID!
    title: String
}

input PostInput {
    id: ID!
    title: String
}

type Query {
    batchGet(ids: [ID]): [Post]
}

type Mutation {
    batchAdd(posts: [PostInput]): [Post]
    batchDelete(ids: [ID]): [Post]
}

schema {
    query: Query
    mutation: Mutation
}
```

使用以下请求映射模板将一个解析器附加到 `batchAdd()` 字段。这会自动接受 GraphQL `input PostInput` 类型的每个项目并构建 `BatchPutItem` 操作所需的一个映射：

```
#set($postsdata = [])
#foreach($item in ${ctx.args.posts})
    $util.qr($postsdata.add($util.dynamodb.toMapValues($item)))
#end
```

```
{
    "version" : "2018-05-29",
    "operation" : "BatchPutItem",
    "tables" : {
        "Posts": $utils.toJson($postsdata)
    }
}
```

在这种情况下，响应映射模板是一个简单的传递，但表名称作为 `..data.Posts` 追加到上下文对象，如下所示：

```
$util.toJson($ctx.result.data.Posts)
```

现在，导航到 AWS AppSync 控制台的查询页面，并运行以下 batchAdd 变更：

```
mutation add {
    batchAdd(posts:[{
            id: 1 title: "Running in the Park"},{
            id: 2 title: "Playing fetch"
        }]){
            id
            title
    }
}
```

您应该会看到输出到屏幕的结果，并且可以通过 DynamoDB 控制台单独验证这两个值是否写入到 Posts 表。

接下来，使用以下请求映射模板将一个解析器附加到 `batchGet()` 字段。这会自动接受 GraphQL `ids:[]` 类型的每个项目并构建 `BatchGetItem` 操作所需的一个映射：

```
#set($ids = [])
#foreach($id in ${ctx.args.ids})
    #set($map = {})
    $util.qr($map.put("id", $util.dynamodb.toString($id)))
    $util.qr($ids.add($map))
#end

{
    "version" : "2018-05-29",
    "operation" : "BatchGetItem",
```

```
    "tables" : {
        "Posts": {
            "keys": $util.toJson($ids),
            "consistentRead": true,
            "projection" : {
                "expression" : "#id, title",
                "expressionNames" : { "#id" : "id"}
                }
        }
    }
}
```

响应映射模板还是一个简单的传递，且再一次，表名称作为 `..data.Posts` 追加到上下文对象：

```
$util.toJson($ctx.result.data.Posts)
```

现在，返回到 AWS AppSync 控制台的查询页面，并运行以下 batchGet 查询：

```
query get {
    batchGet(ids:[1,2,3]){
        id
        title
    }
}
```

这应返回您早前添加的两个 `id` 值的结果。请注意，对于值为 `null` 的 `id`，将返回 3 值。这是因为您的 Posts 表中还没有具有该值的记录。另请注意，AWS AppSync 使用与传递给查询的键相同的顺序返回结果，这是 AWS AppSync 代表您执行的一项额外功能。因此，如果您切换到 `batchGet(ids:[1,3,2)`，将看到订单已更改。您还将了解哪个 `id` 返回了 `null` 值。

最后，使用以下请求映射模板将一个解析器附加到 `batchDelete()` 字段。这会自动接受 GraphQL `ids:[]` 类型的每个项目并构建 `BatchGetItem` 操作所需的一个映射：

```
#set($ids = [])
#foreach($id in ${ctx.args.ids})
    #set($map = {})
    $util.qr($map.put("id", $util.dynamodb.toString($id)))
    $util.qr($ids.add($map))
#end

{
    "version" : "2018-05-29",
```

```
    "operation" : "BatchDeleteItem",
    "tables" : {
        "Posts": $util.toJson($ids)
    }
}
```

响应映射模板还是一个简单的传递，且再一次，表名称作为 `..data.Posts` 追加到上下文对象：

```
$util.toJson($ctx.result.data.Posts)
```

现在，返回到 AWS AppSync 控制台的查询页面，并运行以下 batchDelete 变更：

```
mutation delete {
    batchDelete(ids:[1,2]){ id }
}
```

现在应删除 `id` 为 1 和 2 的记录。如果您更早重新运行 `batchGet()` 查询，则应返回 `null`。

多个表批处理

AWS AppSync 还允许您在多个表中执行批处理操作。我们来构建更复杂的应用程序。想象一下，我们正在构建一个 Pet Health 应用程序，其中的传感器报告宠物位置和身体温度。传感器由电池供电，并每隔几分钟尝试连接到网络。在传感器建立连接时，它将读数发送到我们的 AWS AppSync API。然后，触发器分析数据，这样，就可以向宠物主人显示控制面板。我们重点关注如何表示传感器与后端数据存储之间的交互。

作为先决条件，让我们先创建两个 DynamoDB 表；locationReadings 存储传感器位置读数，temperatureReadings 存储传感器温度读数。这两个表正好共享相同的主键结构：`sensorId (String)` 是分区键，而 `timestamp (String)` 是排序键。

我们使用以下 GraphQL 架构：

```
type Mutation {
    # Register a batch of readings
    recordReadings(tempReadings: [TemperatureReadingInput], locReadings:
 [LocationReadingInput]): RecordResult
    # Delete a batch of readings
    deleteReadings(tempReadings: [TemperatureReadingInput], locReadings:
 [LocationReadingInput]): RecordResult
}
```

```
type Query {
    # Retrieve all possible readings recorded by a sensor at a specific time
    getReadings(sensorId: ID!, timestamp: String!): [SensorReading]
}

type RecordResult {
    temperatureReadings: [TemperatureReading]
    locationReadings: [LocationReading]
}

interface SensorReading {
    sensorId: ID!
    timestamp: String!
}

# Sensor reading representing the sensor temperature (in Fahrenheit)
type TemperatureReading implements SensorReading {
    sensorId: ID!
    timestamp: String!
    value: Float
}

# Sensor reading representing the sensor location (lat,long)
type LocationReading implements SensorReading {
    sensorId: ID!
    timestamp: String!
    lat: Float
    long: Float
}

input TemperatureReadingInput {
    sensorId: ID!
    timestamp: String
    value: Float
}

input LocationReadingInput {
    sensorId: ID!
    timestamp: String
    lat: Float
    long: Float
}
```

BatchPutItem - 记录传感器读数

我们的传感器需要能够在连接到 Internet 后立即发送其读数。GraphQL 字段 `Mutation.recordReadings` 是传感器将用来执行上述操作的 API。现在附加一个解析器以实际使用 API。

选择 `Mutation.recordReadings` 字段旁边的附加。在下一个屏幕上，选择本教程开始时创建的同一个 `BatchTutorial` 数据源。

我们添加以下请求映射模板。

请求映射模板

```
## Convert tempReadings arguments to DynamoDB objects
#set($tempReadings = [])
#foreach($reading in ${ctx.args.tempReadings})
    $util.qr($tempReadings.add($util.dynamodb.toMapValues($reading)))
#end

## Convert locReadings arguments to DynamoDB objects
#set($locReadings = [])
#foreach($reading in ${ctx.args.locReadings})
    $util.qr($locReadings.add($util.dynamodb.toMapValues($reading)))
#end

{
    "version" : "2018-05-29",
    "operation" : "BatchPutItem",
    "tables" : {
        "locationReadings": $utils.toJson($locReadings),
        "temperatureReadings": $utils.toJson($tempReadings)
    }
}
```

如您所见，`BatchPutItem` 操作使我们能够指定多个表。

我们使用以下响应映射模板。

响应映射模板

```
## If there was an error with the invocation
## there might have been partial results
```

```
#if($ctx.error)
    ## Append a GraphQL error for that field in the GraphQL response
    $utils.appendError($ctx.error.message, $ctx.error.message)
#end
## Also returns data for the field in the GraphQL response
$utils.toJson($ctx.result.data)
```

使用批处理操作，可能会从调用中同时返回错误和结果。在这种情况下，我们可以自主执行一些额外的错误处理。

注意：$utils.appendError() 的用法与 $util.error() 相似，主要区别是前者不中断对映射模板的评估。相反，它指示字段发生了错误，但允许评估模板，因此会将数据返回给调用方。我们建议，当您的应用程序需要返回部分结果时使用 $utils.appendError()。

保存解析器，并导航到 AWS AppSync 控制台的查询页面。现在发送一些传感器读数！

执行以下变更：

```
mutation sendReadings {
  recordReadings(
    tempReadings: [
      {sensorId: 1, value: 85.5, timestamp: "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, value: 85.7, timestamp: "2018-02-01T17:21:06.000+08:00"},
      {sensorId: 1, value: 85.8, timestamp: "2018-02-01T17:21:07.000+08:00"},
      {sensorId: 1, value: 84.2, timestamp: "2018-02-01T17:21:08.000+08:00"},
      {sensorId: 1, value: 81.5, timestamp: "2018-02-01T17:21:09.000+08:00"}
    ]
    locReadings: [
      {sensorId: 1, lat: 47.615063, long: -122.333551, timestamp:
 "2018-02-01T17:21:05.000+08:00"},
      {sensorId: 1, lat: 47.615163, long: -122.333552, timestamp:
 "2018-02-01T17:21:06.000+08:00"}
      {sensorId: 1, lat: 47.615263, long: -122.333553, timestamp:
 "2018-02-01T17:21:07.000+08:00"}
      {sensorId: 1, lat: 47.615363, long: -122.333554, timestamp:
 "2018-02-01T17:21:08.000+08:00"}
      {sensorId: 1, lat: 47.615463, long: -122.333555, timestamp:
 "2018-02-01T17:21:09.000+08:00"}
    ]) {
    locationReadings {
      sensorId
      timestamp
      lat
```

```
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}
```

我们在一个变更中发送了 10 个传感器读数，并在两个表之间拆分读数。使用 DynamoDB 控制台验证是否在 locationReadings 和 temperatureReadings 表中显示数据。

BatchDeleteItem - 删除传感器读数

同样，我们也需要删除批量传感器读数。我们使用 `Mutation.deleteReadings` GraphQL 字段来实现此目的。选择 `Mutation.recordReadings` 字段旁边的附加。在下一个屏幕上，选择本教程开始时创建的同一个 `BatchTutorial` 数据源。

我们使用以下请求映射模板。

请求映射模板

```
## Convert tempReadings arguments to DynamoDB primary keys
#set($tempReadings = [])
#foreach($reading in ${ctx.args.tempReadings})
    #set($pkey = {})
    $util.qr($pkey.put("sensorId", $reading.sensorId))
    $util.qr($pkey.put("timestamp", $reading.timestamp))
    $util.qr($tempReadings.add($util.dynamodb.toMapValues($pkey)))
#end

## Convert locReadings arguments to DynamoDB primary keys
#set($locReadings = [])
#foreach($reading in ${ctx.args.locReadings})
    #set($pkey = {})
    $util.qr($pkey.put("sensorId", $reading.sensorId))
    $util.qr($pkey.put("timestamp", $reading.timestamp))
    $util.qr($locReadings.add($util.dynamodb.toMapValues($pkey)))
#end

{
    "version" : "2018-05-29",
```

```
    "operation" : "BatchDeleteItem",
    "tables" : {
        "locationReadings": $utils.toJson($locReadings),
        "temperatureReadings": $utils.toJson($tempReadings)
    }
}
```

该响应映射模板与我们用于 Mutation.recordReadings 的模板相同。

响应映射模板

```
## If there was an error with the invocation
## there might have been partial results
#if($ctx.error)
    ## Append a GraphQL error for that field in the GraphQL response
    $utils.appendError($ctx.error.message, $ctx.error.message)
#end
## Also return data for the field in the GraphQL response
$utils.toJson($ctx.result.data)
```

保存解析器，并导航到 AWS AppSync 控制台的查询页面。现在，让我们删除几个传感器读数！

执行以下变更：

```
mutation deleteReadings {
  # Let's delete the first two readings we recorded
  deleteReadings(
    tempReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]
    locReadings: [{sensorId: 1, timestamp: "2018-02-01T17:21:05.000+08:00"}]) {
    locationReadings {
      sensorId
      timestamp
      lat
      long
    }
    temperatureReadings {
      sensorId
      timestamp
      value
    }
  }
}
```

通过 DynamoDB 控制台验证是否从 locationReadings 和 temperatureReadings 表中删除了这两个读数。

BatchGetItem - 检索读数

Pet Health 应用程序的另一个常见操作是检索特定时间点的传感器读数。我们将解析器附加到架构上的 `Query.getReadings` GraphQL 字段。选择 Attach (附加)，然后，在下一个屏幕上选择本教程开始时创建的同一个 `BatchTutorial` 数据源。

我们添加以下请求映射模板。

请求映射模板

```
## Build a single DynamoDB primary key,
## as both locationReadings and tempReadings tables
## share the same primary key structure
#set($pkey = {})
$util.qr($pkey.put("sensorId", $ctx.args.sensorId))
$util.qr($pkey.put("timestamp", $ctx.args.timestamp))

{
    "version" : "2018-05-29",
    "operation" : "BatchGetItem",
    "tables" : {
        "locationReadings": {
            "keys": [$util.dynamodb.toMapValuesJson($pkey)],
            "consistentRead": true
        },
        "temperatureReadings": {
            "keys": [$util.dynamodb.toMapValuesJson($pkey)],
            "consistentRead": true
        }
    }
}
```

请注意，我们现在使用 BatchGetItem 操作。

我们的响应映射模板会有点不同，因为我们选择返回 `SensorReading` 列表。我们将调用结果映射到所需的形状。

响应映射模板

```
## Merge locationReadings and temperatureReadings
```

```
## into a single list
## __typename needed as schema uses an interface
#set($sensorReadings = [])

#foreach($locReading in $ctx.result.data.locationReadings)
    $util.qr($locReading.put("__typename", "LocationReading"))
    $util.qr($sensorReadings.add($locReading))
#end

#foreach($tempReading in $ctx.result.data.temperatureReadings)
    $util.qr($tempReading.put("__typename", "TemperatureReading"))
    $util.qr($sensorReadings.add($tempReading))
#end

$util.toJson($sensorReadings)
```

保存解析器，并导航到 AWS AppSync 控制台的查询页面。现在，我们来检索传感器读数！

执行以下查询：

```
query getReadingsForSensorAndTime {
  # Let's retrieve the very first two readings
  getReadings(sensorId: 1, timestamp: "2018-02-01T17:21:06.000+08:00") {
    sensorId
    timestamp
    ...on TemperatureReading {
      value
    }
    ...on LocationReading {
      lat
      long
    }
  }
}
```

我们已成功说明了如何通过 AWS AppSync 使用 DynamoDB 批处理操作。

错误处理

在 AWS AppSync 中，数据源操作有时可能会返回部分结果。部分结果是一个术语，我们用它来表示操作的输出中包含某些数据和一个错误。由于错误处理本质上是应用程序特定的，因此，AWS AppSync 允许您在响应映射模板中处理错误。上下文中的解析器调用错误（如果有）为

$ctx.error。调用错误始终包含一条消息和一个类型，可作为属性 $ctx.error.message 和 $ctx.error.type 进行访问。在响应映射模板调用期间，您可以通过三种方式处理部分结果：

1. 仅返回数据以忽略调用错误
2. 通过停止响应映射模板评估（这不会返回任何数据）来引发错误（使用 $util.error(...)）。
3. 附加一个错误（使用 $util.appendError(...)）并且也返回数据

让我们通过 DynamoDB 批处理操作分别说明上述三点！

DynamoDB 批处理操作

借助 DynamoDB 批处理操作，批处理可能会部分完成。也就是说，某些请求的项目或键未得到处理。如果 AWS AppSync 无法完成批处理，则会在上下文中设置未处理的项目和调用错误。

我们将使用本教程前一部分中 Query.getReadings 操作的 BatchGetItem 字段配置来实施错误处理。这一次，我们假定在执行 Query.getReadings 字段时，temperatureReadings DynamoDB 表耗尽了预置的吞吐量。在 AWS AppSync 第二次尝试处理批次中的剩余元素时，DynamoDB 引发了 ProvisionedThroughputExceededException。

以下 JSON 表示在 DynamoDB 批处理调用之后但在评估响应映射模板之前的序列化上下文。

```
{
  "arguments": {
    "sensorId": "1",
    "timestamp": "2018-02-01T17:21:05.000+08:00"
  },
  "source": null,
  "result": {
    "data": {
      "temperatureReadings": [
        null
      ],
      "locationReadings": [
        {
          "lat": 47.615063,
          "long": -122.333551,
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ]
    },
```

```
    "unprocessedKeys": {
      "temperatureReadings": [
        {
          "sensorId": "1",
          "timestamp": "2018-02-01T17:21:05.000+08:00"
        }
      ],
      "locationReadings": []
    }
  },
  "error": {
    "type": "DynamoDB:ProvisionedThroughputExceededException",
    "message": "You exceeded your maximum allowed provisioned throughput for a table or
 for one or more global secondary indexes. (...)"
  },
  "outErrors": []
}
```

关于上下文需要注意的几点：

- AWS AppSync 在上下文中的 `$ctx.error` 处设置了调用错误，并且错误类型设置为 DynamoDB:ProvisionedThroughputExceededException。
- 即使存在错误，也会在 `$ctx.result.data` 中为每个表映射结果
- 在 `$ctx.result.data.unprocessedKeys` 中提供了未处理的键。此处，由于表吞吐量不足，AWS AppSync 无法检索具有 (sensorId:1, timestamp:2018-02-01T17:21:05.000+08:00) 键的项目。

注意：对于 `BatchPutItem`，它是 `$ctx.result.data.unprocessedItems`。对于 `BatchDeleteItem`，它是 `$ctx.result.data.unprocessedKeys`。

我们通过三种不同方式处理此错误。

1. 承受调用错误

返回数据而不处理调用错误：这会有效地承受此错误，同时使给定 GraphQL 字段的结果始终成功。

我们编写的响应映射模板是熟悉的，仅侧重于结果数据。

响应映射模板：

```
$util.toJson($ctx.result.data)
```

GraphQL 响应：

```
{
  "data": {
    "getReadings": [
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "lat": 47.615063,
        "long": -122.333551
      },
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  }
}
```

将不向错误响应中添加任何错误，因为只对数据执行了操作。

2. 引发错误以中止模板执行

从客户端角度看，在应将部分失败视为完全失败时，您可以中止执行模板以防止返回数据。`$util.error(...)` 实用程序方法实现完全此行为。

响应映射模板：

```
## there was an error let's mark the entire field
## as failed and do not return any data back in the response
#if ($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type, null,
 $ctx.result.data.unprocessedKeys)
#end

$util.toJson($ctx.result.data)
```

GraphQL 响应：

```
{
  "data": {
    "getReadings": null
```

```
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
        }
      ],
      "message": "You exceeded your maximum allowed provisioned throughput for a table
 or for one or more global secondary indexes. (...)"
    }
  ]
}
```

即使可能已从 DynamoDB 批处理操作返回了一些结果，我们也选择引发错误，这样，getReadings GraphQL 字段为 Null，并且此错误已添加到 GraphQL 响应的错误数据块中。

3. 追加错误以返回数据和错误

在某些情况下，为了提供更好的用户体验，应用程序可以返回部分结果并向其客户端通知未处理的项目。客户端可以决定是实施重试，还是将错误翻译出来并返回给最终用户。$util.appendError(...) 是一个可以启用此行为的实用程序方法，具体方式为：让应用程序设计人员在上下文中追加错误，而不干扰对模板的评估。在评估模板后，AWS AppSync 将任何上下文错误附加到 GraphQL 响应的错误块以处理它们。

响应映射模板：

```
#if ($ctx.error)
```

```
    ## pass the unprocessed keys back to the caller via the `errorInfo` field
    $util.appendError($ctx.error.message, $ctx.error.type, null,
 $ctx.result.data.unprocessedKeys)
#end

$util.toJson($ctx.result.data)
```

我们在 GraphQL 响应的错误块中转发了调用错误和 unprocessedKeys 元素。getReadings 字段也从 locationReadings 表中返回部分数据，如下面的响应中所示。

GraphQL 响应：

```
{
  "data": {
    "getReadings": [
      null,
      {
        "sensorId": "1",
        "timestamp": "2018-02-01T17:21:05.000+08:00",
        "value": 85.5
      }
    ]
  },
  "errors": [
    {
      "path": [
        "getReadings"
      ],
      "data": null,
      "errorType": "DynamoDB:ProvisionedThroughputExceededException",
      "errorInfo": {
        "temperatureReadings": [
          {
            "sensorId": "1",
            "timestamp": "2018-02-01T17:21:05.000+08:00"
          }
        ],
        "locationReadings": []
      },
      "locations": [
        {
          "line": 58,
          "column": 3
```

```
        }
      ],
      "message": "You exceeded your maximum allowed provisioned throughput for a table
 or for one or more global secondary indexes. (...)"
    }
  ]
}
```

教程：DynamoDB 事务解析器

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

AWS AppSync 支持对单个区域中的一个或多个表执行 Amazon DynamoDB 事务操作。支持的操作为 `TransactGetItems` 和 `TransactWriteItems`。通过在 AWS AppSync 中使用这些功能，您可以执行如下任务：

- 在单个查询中传递键列表，并从表中返回结果
- 在单个查询中从一个或多个表读取记录
- 以要么全部写入，要么全不写入的方式将事务记录写入一个或多个表
- 在满足某些条件时执行事务

权限

与其他解析器一样，您需要在 AWS AppSync 中创建数据源，并创建一个角色或使用现有的角色。由于事务操作需要具有 DynamoDB 表的不同权限，因此，您需要为配置的角色授予读取或写入操作权限：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
```

```
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:region:accountId:table/TABLENAME",
                "arn:aws:dynamodb:region:accountId:table/TABLENAME/*"
            ]
        }
    ]
}
```

注意：角色与 AWS AppSync 中的数据源相关联，并对数据源调用字段上的解析器。配置为针对 DynamoDB 获取的数据源仅指定一个表，以使配置保持简单。因此，当在单个解析器中针对多个表执行事务操作（这是一项更高级的任务）时，您必须向该数据源的角色授予对将与解析器进行交互的任何表的访问权限。这项操作将在上面 IAM 策略中的 Resource (资源) 字段中执行。针对表的事务调用配置在解析器模板中完成，下面将介绍相关内容。

数据源

为简便起见，我们将为本教程中使用的所有解析器使用同一个数据源。在数据源选项卡上，创建一个新的 DynamoDB 数据源，并将其命名为 TransactTutorial。表名称可以是任何内容，因为表名称被指定为事务操作的请求映射模板的一部分。我们将提供表名称 `empty`。

我们将有两个表，分别称为 savingAccounts 和 checkingAccounts，两个表都使用 `accountNumber` 作为分区键，还有一个 transactionHistory 表，该表使用 `transactionId` 作为分区键。

对于本教程，具有以下内联策略的任何角色都有效。将 `region` 和 `accountId` 替换为您的区域和账户 ID：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
```

```
                "dynamodb:UpdateItem"
            ],
            "Effect": "Allow",
            "Resource": [
                "arn:aws:dynamodb:region:accountId:table/savingAccounts",
                "arn:aws:dynamodb:region:accountId:table/savingAccounts/*",
                "arn:aws:dynamodb:region:accountId:table/checkingAccounts",
                "arn:aws:dynamodb:region:accountId:table/checkingAccounts/*",
                "arn:aws:dynamodb:region:accountId:table/transactionHistory",
                "arn:aws:dynamodb:region:accountId:table/transactionHistory/*"
            ]
        }
    ]
}
```

事务

对于本示例，上下文是一个典型的银行交易，我们将使用 TransactWriteItems 来执行以下操作：

- 从储蓄账户转账到支票账户
- 为每个交易生成新的交易记录

然后我们将使用 TransactGetItems 从储蓄账户和支票账户中检索详细信息。

我们按如下所示定义我们的 GraphQL 架构：

```
type SavingAccount {
    accountNumber: String!
    username: String
    balance: Float
}

type CheckingAccount {
    accountNumber: String!
    username: String
    balance: Float
}

type TransactionHistory {
    transactionId: ID!
    from: String
    to: String
```

```
    amount: Float
}

type TransactionResult {
    savingAccounts: [SavingAccount]
    checkingAccounts: [CheckingAccount]
    transactionHistory: [TransactionHistory]
}

input SavingAccountInput {
    accountNumber: String!
    username: String
    balance: Float
}

input CheckingAccountInput {
    accountNumber: String!
    username: String
    balance: Float
}

input TransactionInput {
    savingAccountNumber: String!
    checkingAccountNumber: String!
    amount: Float!
}

type Query {
    getAccounts(savingAccountNumbers: [String], checkingAccountNumbers: [String]):
 TransactionResult
}

type Mutation {
    populateAccounts(savingAccounts: [SavingAccountInput], checkingAccounts:
 [CheckingAccountInput]): TransactionResult
    transferMoney(transactions: [TransactionInput]): TransactionResult
}

schema {
    query: Query
    mutation: Mutation
}
```

TransactWriteItems - 填充账户

为了在账户之间转账，我们需要用详细信息填充表格。我们将使用 GraphQL 操作 `Mutation.populateAccounts` 来实现此目的。

在“架构”部分中，单击 `Mutation.populateAccounts` 操作旁边的附加。转到“VTL 单位解析器”，然后选择相同的 `TransactTutorial` 数据源。

现在使用以下请求映射模板：

请求映射模板

```
#set($savingAccountTransactPutItems = [])
#set($index = 0)
#foreach($savingAccount in ${ctx.args.savingAccounts})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
 $util.dynamodb.toString($savingAccount.accountNumber)))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("username",
 $util.dynamodb.toString($savingAccount.username)))
    $util.qr($attributeValues.put("balance",
 $util.dynamodb.toNumber($savingAccount.balance)))
    #set($index = $index + 1)
    #set($savingAccountTransactPutItem = {"table": "savingAccounts",
        "operation": "PutItem",
        "key": $keyMap,
        "attributeValues": $attributeValues})
    $util.qr($savingAccountTransactPutItems.add($savingAccountTransactPutItem))
#end

#set($checkingAccountTransactPutItems = [])
#set($index = 0)
#foreach($checkingAccount in ${ctx.args.checkingAccounts})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
 $util.dynamodb.toString($checkingAccount.accountNumber)))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("username",
 $util.dynamodb.toString($checkingAccount.username)))
    $util.qr($attributeValues.put("balance",
 $util.dynamodb.toNumber($checkingAccount.balance)))
    #set($index = $index + 1)
```

```
    #set($checkingAccountTransactPutItem = {"table": "checkingAccounts",
        "operation": "PutItem",
        "key": $keyMap,
        "attributeValues": $attributeValues})
    $util.qr($checkingAccountTransactPutItems.add($checkingAccountTransactPutItem))
#end

#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountTransactPutItems))
$util.qr($transactItems.addAll($checkingAccountTransactPutItems))

{
    "version" : "2018-05-29",
    "operation" : "TransactWriteItems",
    "transactItems" : $util.toJson($transactItems)
}
```

和以下响应映射模板：

响应映射模板

```
#if ($ctx.error)
    $util.appendError($ctx.error.message, $ctx.error.type, null,
 $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
    $util.qr($savingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($checkingAccounts = [])
#foreach($index in [3..5])
    $util.qr($checkingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))

$util.toJson($transactionResult)
```

保存解析器，并导航到 AWS AppSync 控制台的查询部分以填充账户。

执行以下变更：

```
mutation populateAccounts {
  populateAccounts (
    savingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 100},
      {accountNumber: "2", username: "Amy", balance: 90},
      {accountNumber: "3", username: "Lily", balance: 80},
    ]
    checkingAccounts: [
      {accountNumber: "1", username: "Tom", balance: 70},
      {accountNumber: "2", username: "Amy", balance: 60},
      {accountNumber: "3", username: "Lily", balance: 50},
    ]) {
    savingAccounts {
      accountNumber
    }
    checkingAccounts {
      accountNumber
    }
  }
}
```

我们在一个变更中填充了 3 个储蓄账户和 3 个支票账户。

使用 DynamoDB 控制台验证是否在 savingAccounts 和 checkingAccounts 表中显示数据。

TransactWriteItems - 转账

使用以下请求映射模板将一个解析器附加到 `transferMoney` 变更。请注意 `amounts`、`savingAccountNumbers` 和 `checkingAccountNumbers` 的值相同。

```
#set($amounts = [])
#foreach($transaction in ${ctx.args.transactions})
    #set($attributeValueMap = {})
    $util.qr($attributeValueMap.put(":amount",
 $util.dynamodb.toNumber($transaction.amount)))
    $util.qr($amounts.add($attributeValueMap))
#end

#set($savingAccountTransactUpdateItems = [])
#set($index = 0)
#foreach($transaction in ${ctx.args.transactions})
```

```
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
 $util.dynamodb.toString($transaction.savingAccountNumber)))
    #set($update = {})
    $util.qr($update.put("expression", "SET balance = balance - :amount"))
    $util.qr($update.put("expressionValues", $amounts[$index]))
    #set($index = $index + 1)
    #set($savingAccountTransactUpdateItem = {"table": "savingAccounts",
        "operation": "UpdateItem",
        "key": $keyMap,
        "update": $update})
    $util.qr($savingAccountTransactUpdateItems.add($savingAccountTransactUpdateItem))
#end

#set($checkingAccountTransactUpdateItems = [])
#set($index = 0)
#foreach($transaction in ${ctx.args.transactions})
    #set($keyMap = {})
    $util.qr($keyMap.put("accountNumber",
 $util.dynamodb.toString($transaction.checkingAccountNumber)))
    #set($update = {})
    $util.qr($update.put("expression", "SET balance = balance + :amount"))
    $util.qr($update.put("expressionValues", $amounts[$index]))
    #set($index = $index + 1)
    #set($checkingAccountTransactUpdateItem = {"table": "checkingAccounts",
        "operation": "UpdateItem",
        "key": $keyMap,
        "update": $update})

 $util.qr($checkingAccountTransactUpdateItems.add($checkingAccountTransactUpdateItem))
#end

#set($transactionHistoryTransactPutItems = [])
#foreach($transaction in ${ctx.args.transactions})
    #set($keyMap = {})
    $util.qr($keyMap.put("transactionId", $util.dynamodb.toString(${utils.autoId()})))
    #set($attributeValues = {})
    $util.qr($attributeValues.put("from",
 $util.dynamodb.toString($transaction.savingAccountNumber)))
    $util.qr($attributeValues.put("to",
 $util.dynamodb.toString($transaction.checkingAccountNumber)))
    $util.qr($attributeValues.put("amount",
 $util.dynamodb.toNumber($transaction.amount)))
    #set($transactionHistoryTransactPutItem = {"table": "transactionHistory",
```

```
        "operation": "PutItem",
        "key": $keyMap,
        "attributeValues": $attributeValues})

 $util.qr($transactionHistoryTransactPutItems.add($transactionHistoryTransactPutItem))
#end

#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountTransactUpdateItems))
$util.qr($transactItems.addAll($checkingAccountTransactUpdateItems))
$util.qr($transactItems.addAll($transactionHistoryTransactPutItems))

{
    "version" : "2018-05-29",
    "operation" : "TransactWriteItems",
    "transactItems" : $util.toJson($transactItems)
}
```

我们将在一个 TransactWriteItems 操作中进行 3 笔银行交易。使用以下响应映射模板：

```
#if ($ctx.error)
    $util.appendError($ctx.error.message, $ctx.error.type, null,
 $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
    $util.qr($savingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($checkingAccounts = [])
#foreach($index in [3..5])
    $util.qr($checkingAccounts.add(${ctx.result.keys[$index]}))
#end

#set($transactionHistory = [])
#foreach($index in [6..8])
    $util.qr($transactionHistory.add(${ctx.result.keys[$index]}))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))
```

```
$util.qr($transactionResult.put('transactionHistory', $transactionHistory))

$util.toJson($transactionResult)
```

现在，导航到 AWS AppSync 控制台的查询部分并执行 transferMoney 变更，如下所示：

```
mutation write {
  transferMoney(
    transactions: [
      {savingAccountNumber: "1", checkingAccountNumber: "1", amount: 7.5},
      {savingAccountNumber: "2", checkingAccountNumber: "2", amount: 6.0},
      {savingAccountNumber: "3", checkingAccountNumber: "3", amount: 3.3}
    ]) {
    savingAccounts {
      accountNumber
    }
    checkingAccounts {
      accountNumber
    }
    transactionHistory {
      transactionId
    }
  }
}
```

我们在一个变更中发送了 2 笔银行交易。使用 DynamoDB 控制台验证是否在 savingAccounts、checkingAccounts 和 transactionHistory 表中显示数据。

TransactGetItems - 检索账户

为了在单个交易请求中从储蓄账户和支票账户检索详细信息，我们需要将解析器附加到架构上的 `Query.getAccounts` GraphQL 操作。选择附加，转到“VTL 单位解析器”，然后在下一个屏幕上选择在教程开始时创建的相同 `TransactTutorial` 数据源。按如下所示配置模板：

请求映射模板

```
#set($savingAccountsTransactGets = [])
#foreach($savingAccountNumber in ${ctx.args.savingAccountNumbers})
    #set($savingAccountKey = {})
    $util.qr($savingAccountKey.put("accountNumber",
 $util.dynamodb.toString($savingAccountNumber)))
    #set($savingAccountTransactGet = {"table": "savingAccounts", "key":
 $savingAccountKey})
```

```
    $util.qr($savingAccountsTransactGets.add($savingAccountTransactGet))
#end

#set($checkingAccountsTransactGets = [])
#foreach($checkingAccountNumber in ${ctx.args.checkingAccountNumbers})
    #set($checkingAccountKey = {})
    $util.qr($checkingAccountKey.put("accountNumber",
 $util.dynamodb.toString($checkingAccountNumber)))
    #set($checkingAccountTransactGet = {"table": "checkingAccounts", "key":
 $checkingAccountKey})
    $util.qr($checkingAccountsTransactGets.add($checkingAccountTransactGet))
#end

#set($transactItems = [])
$util.qr($transactItems.addAll($savingAccountsTransactGets))
$util.qr($transactItems.addAll($checkingAccountsTransactGets))

{
    "version" : "2018-05-29",
    "operation" : "TransactGetItems",
    "transactItems" : $util.toJson($transactItems)
}
```

响应映射模板

```
#if ($ctx.error)
    $util.appendError($ctx.error.message, $ctx.error.type, null,
 $ctx.result.cancellationReasons)
#end

#set($savingAccounts = [])
#foreach($index in [0..2])
    $util.qr($savingAccounts.add(${ctx.result.items[$index]}))
#end

#set($checkingAccounts = [])
#foreach($index in [3..4])
    $util.qr($checkingAccounts.add($ctx.result.items[$index]))
#end

#set($transactionResult = {})
$util.qr($transactionResult.put('savingAccounts', $savingAccounts))
$util.qr($transactionResult.put('checkingAccounts', $checkingAccounts))
```

```
$util.toJson($transactionResult)
```

保存解析器，并导航到 AWS AppSync 控制台的查询部分。为了检索储蓄账户和支票账户，请执行以下查询：

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
      username
      balance
    }
  }
}
```

我们已成功说明了如何通过 AWS AppSync 使用 DynamoDB 事务。

教程：HTTP 解析器

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

除了使用任意 HTTP 终端节点解析 GraphQL 字段以外，AWS AppSync 还允许您使用支持的数据源（即 AWS Lambda、Amazon DynamoDB、Amazon OpenSearch Service 或 Amazon Aurora）执行各种操作。在您的 HTTP 终端节点可用后，您可以使用数据源连接它们。然后，您可以在架构中配置一个解析器以执行 GraphQL 操作（如查询、变更和订阅）。本教程将引导您了解一些常见示例。

在本教程中，您在 AWS AppSync GraphQL 终端节点中使用 REST API（是使用 Amazon API Gateway 和 Lambda 创建的）。

一键设置

如果要在配置了 HTTP 终端节点的 AWS AppSync 中自动设置 GraphQL 终端节点（使用 Amazon API Gateway 和 Lambda），您可以使用以下 AWS CloudFormation 模板：

创建 REST API

您可以使用以下 AWS CloudFormation 模板来设置适用于本教程的 REST 终端节点：

AWS CloudFormation 堆栈将执行以下步骤：

1. 设置 Lambda 函数，其中包含您的微服务的业务逻辑。
2. 使用以下终端节点/方法/内容类型组合设置一个 API Gateway REST API：

API 资源路径	HTTP 方法	支持的内容类型
/v1/users	POST	application/json
/v1/users	GET	application/json
/v1/users/1	GET	application/json
/v1/users/1	PUT	application/json
/v1/users/1	删除	application/json

创建您的 GraphQL API

要在 AWS AppSync 中创建 GraphQL API，请执行以下操作：

- 打开 AWS AppSync 控制台，然后选择创建 API。
- 对于 API 名称，请键入 `UserData`。
- 选择自定义架构。

- 选择创建。

AWS AppSync 控制台使用 API 密钥身份验证模式为您创建新的 GraphQL API。您可以根据本教程后面的说明，使用控制台设置 GraphQL API 的其余部分，并针对它运行查询。

创建 GraphQL 架构

现在，您有一个 GraphQL API，让我们创建一个 GraphQL 架构。从 AWS AppSync 控制台的架构编辑器中，确保您的架构与以下架构匹配：

```
schema {
    query: Query
    mutation: Mutation
}

type Mutation {
    addUser(userInput: UserInput!): User
    deleteUser(id: ID!): User
}

type Query {
    getUser(id: ID): User
    listUser: [User!]!
}

type User {
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
}

input UserInput {
    id: ID!
    username: String!
    firstname: String
    lastname: String
    phone: String
    email: String
}
```

配置您的 HTTP 数据源

要配置 HTTP 数据源，请执行以下操作：

- 在 DataSources (数据源) 选项卡上，选择 New (新建)，然后为数据源键入一个友好名称（例如，HTTP）。
- 在 Data source type (数据源类型) 中，选择 HTTP。
- 将终端节点设置为创建的 API 网关终端节点。确保不将阶段名称作为终端节点的一部分包含在内。

注意：AWS AppSync 目前仅支持公有终端节点。

注意：有关 AWS AppSync 服务识别的认证机构的更多信息，请参阅 AWS AppSync 识别的 HTTPS 终端节点证书颁发机构 (CA)。

配置解析器

在此步骤中，您将 HTTP 数据源连接到 getUser 查询。

设置解析器：

- 选择架构选项卡。
- 在右侧的数据类型窗格中，在 Query 类型下面找到 getUser 字段，然后选择附加。
- 在 Data source name (数据源名称) 中，选择 HTTP。
- 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下代码：

```
{
    "version": "2018-05-29",
    "method": "GET",
    "params": {
        "headers": {
            "Content-Type": "application/json"
        }
    },
    "resourcePath": $util.toJson("/v1/users/${ctx.args.id}")
}
```

- 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下代码：

```
## return the body
#if($ctx.result.statusCode == 200)
    ##if response is 200
    $ctx.result.body
#else
    ##if response is not 200, append the response to error block.
    $utils.appendError($ctx.result.body, "$ctx.result.statusCode")
#end
```

- 选择 Query (查询) 选项卡，然后运行以下查询：

```
query GetUser{
    getUser(id:1){
        id
        username
    }
}
```

此查询应返回以下响应：

```
{
    "data": {
        "getUser": {
            "id": "1",
            "username": "nadia"
        }
    }
}
```

- 选择架构选项卡。
- 在右侧的数据类型窗格中，在 Mutation 下面找到 addUser 字段，然后选择附加。
- 在 Data source name (数据源名称) 中，选择 HTTP。
- 在 Configure the request mapping template (配置请求映射模板) 中，粘贴以下代码：

```
{
    "version": "2018-05-29",
    "method": "POST",
    "resourcePath": "/v1/users",
```

```
    "params":{
      "headers":{
        "Content-Type": "application/json",
      },
      "body": $util.toJson($ctx.args.userInput)
    }
}
```

- 在 Configure the response mapping template (配置响应映射模板) 中，粘贴以下代码：

```
## Raise a GraphQL field error in case of a datasource invocation error
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
## if the response status code is not 200, then return an error. Else return the body
 **
#if($ctx.result.statusCode == 200)
    ## If response is 200, return the body.
    $ctx.result.body
#else
    ## If response is not 200, append the response to error block.
    $utils.appendError($ctx.result.body, "$ctx.result.statusCode")
#end
```

- 选择 Query (查询) 选项卡，然后运行以下查询：

```
mutation addUser{
    addUser(userInput:{
        id:"2",
        username:"shaggy"
    }){
        id
        username
    }
}
```

此查询应返回以下响应：

```
{
    "data": {
```

```
        "getUser": {
        "id": "2",
        "username": "shaggy"
        }
    }
}
```

调用 AWS 服务

您可以使用 HTTP 解析器为 AWS 服务设置 GraphQL API 接口。发送到 AWS 的 HTTP 请求必须使用签名版本 4 流程进行签名，以使 AWS 可以识别发送者。AWS在将 IAM 角色与 HTTP 数据源关联时，AppSync 代表您计算签名。

您提供两个额外的组件以使用 HTTP 解析器调用 AWS 服务：

- 有权调用 AWS 服务 API 的 IAM 角色
- 数据源中的签名配置

例如，如果要使用 HTTP 解析器调用 ListGraphqlApis 操作，您需要先创建由 AWS AppSync 担任的 IAM 角色并附加以下策略：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "appsync:ListGraphqlApis"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

接下来，为 AWS AppSync 创建 HTTP 数据源。在该示例中，您在美国西部（俄勒冈州）区域中调用 AWS AppSync。在名为 `http.json` 文件中设置以下 HTTP 配置，其中包括签名区域和服务名称：

```
{
    "endpoint": "https://appsync.us-west-2.amazonaws.com/",
    "authorizationConfig": {
```

```
        "authorizationType": "AWS_IAM",
        "awsIamConfig": {
            "signingRegion": "us-west-2",
            "signingServiceName": "appsync"
        }
    }
}
```

然后，使用 AWS CLI 创建具有关联角色的数据源，如下所示：

```
aws appsync create-data-source --api-id <API-ID> \
                               --name AWSAppSync \
                               --type HTTP \
                               --http-config file:///http.json \
                               --service-role-arn <ROLE-ARN>
```

在将一个解析器附加到架构中的字段时，请使用以下请求映射模板调用 AWS AppSync：

```
{
    "version": "2018-05-29",
    "method": "GET",
    "resourcePath": "/v1/apis"
}
```

在为该数据源运行 GraphQL 查询时，AWS AppSync 使用您提供的角色对请求进行签名，并将签名包含在请求中。该查询返回您的账户在该 AWS 区域中的 AWS AppSync GraphQL API 列表。

教程：Aurora Serverless

AWS AppSync 提供了一个数据源，用于对已启用数据 API 的 Amazon Aurora Serverless 集群执行 SQL 命令。您可以使用 AppSync 解析器通过 GraphQL 查询、变更和订阅对数据 API 执行 SQL 语句。

创建集群

在将 RDS 数据源添加到 AppSync 之前，您必须先在 Aurora Serverless 集群上启用数据 API 并使用 AWS Secrets Manager 配置密钥。您可以先使用 AWS CLI 创建一个 Aurora Serverless 集群：

```
aws rds create-db-cluster --db-cluster-identifier http-endpoint-test  --master-username
 USERNAME \
```

```
--master-user-password COMPLEX_PASSWORD --engine aurora --engine-mode serverless \
--region us-east-1
```

这将为集群返回一个 ARN。

使用上一步中的 USERNAME 和 COMPLEX_PASSWORD 通过 AWS Secrets Manager 控制台创建一个密钥，也可以通过 CLI 使用如下输入文件创建密钥：

```
{
    "username": "USERNAME",
    "password": "COMPLEX_PASSWORD"
}
```

将其作为参数传递给 AWS CLI：

```
aws secretsmanager create-secret --name HttpRDSSecret --secret-string file://creds.json
 --region us-east-1
```

这将为密钥返回 ARN。

记下稍后在创建数据源时在 AppSync 控制台中使用的 Aurora Serverless 集群和密钥的 ARN。

启用数据 API

您可以通过按照 RDS 文档中的说明操作来在您的集群上启用数据 API。在将数据 API 作为 AppSync 数据源添加之前，必须先启用它。

创建数据库和表

在启用数据 API 后，您可以确保它在 AWS CLI 中使用 `aws rds-data execute-statement` 命令。这将确保在将 Aurora Serverless 集群添加到 AppSync API 之前已进行正确配置。首先，使用 `--sql` 参数创建一个名为 TESTDB 的数据库，如下所示：

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789000:cluster:http-endpoint-test" \
--schema "mysql"  --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789000:secret:testHttp2-AmNvc1"  \
--region us-east-1 --sql "create DATABASE TESTDB"
```

如果运行无误，请使用创建表 命令添加一个表：

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:123456789000:cluster:http-endpoint-test" \
 --schema "mysql"  --secret-arn "arn:aws:secretsmanager:us-
east-1:123456789000:secret:testHttp2-AmNvc1" \
 --region us-east-1 \
 --sql "create table Pets(id varchar(200), type varchar(200), price float)" --database
 "TESTDB"
```

如果一切运行正常，您可以继续在 AppSync API 中将集群添加为数据源。

GraphQL 架构

现在，您的 Aurora Serverless 数据 API 已通过表启动并运行，我们将创建一个 GraphQL 架构并附加用于执行变更和订阅的解析器。在 AWS AppSync 控制台中创建一个新的 API，并导航到架构页面，然后输入以下内容：

```
type Mutation {
    createPet(input: CreatePetInput!): Pet
    updatePet(input: UpdatePetInput!): Pet
    deletePet(input: DeletePetInput!): Pet
}

input CreatePetInput {
    type: PetType
    price: Float!
}

input UpdatePetInput {
id: ID!
    type: PetType
    price: Float!
}

input DeletePetInput {
    id: ID!
}

type Pet {
    id: ID!
    type: PetType
    price: Float
}
```

```
enum PetType {
    dog
    cat
    fish
    bird
    gecko
}

type Query {
    getPet(id: ID!): Pet
    listPets: [Pet]
    listPetsByPriceRange(min: Float, max: Float): [Pet]
}

schema {
    query: Query
    mutation: Mutation
}
```

保存您的架构并导航到数据源页面，然后创建新的数据源。为数据源类型选择关系数据库，并提供一个友好名称。使用您在上一步中创建的数据库名称以及用来创建它的集群 ARN。对于角色，您可以让 AppSync 创建新角色，也可以使用与以下内容类似的策略创建角色：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "rds-data:DeleteItems",
                "rds-data:ExecuteSql",
                "rds-data:ExecuteStatement",
                "rds-data:GetItems",
                "rds-data:InsertItems",
                "rds-data:UpdateItems"
            ],
            "Resource": [
                "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster",
                "arn:aws:rds:us-east-1:123456789012:cluster:mydbcluster:*"
            ]
        },
        {
```

```
            "Effect": "Allow",
            "Action": [
                "secretsmanager:GetSecretValue"
            ],
            "Resource": [
                "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret",
                "arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret:*"
            ]
        }
    ]
}
```

请注意，此策略中有两个需要获得角色访问权的语句。第一个资源是您的 Aurora Serverless 集群，第二个资源是您的 AWS Secrets Manager ARN。在单击创建之前，您需要在 AppSync 数据源配置中提供这两个 ARN。

配置解析器

现在，我们有一个有效的 GraphQL 架构和一个 RDS 数据源，我们可以将解析器附加到架构上的 GraphQL 字段。我们的 API 将提供以下功能：

1. 通过 Mutation.createPet 字段创建宠物
2. 通过 Mutation.updatePet 字段更新宠物
3. 通过 Mutation.deletePet 字段删除宠物
4. 通过 Query.getPet 字段获取单个宠物
5. 通过 Query.listPets 字段列出所有宠物
6. 通过 Query.listPetsByPriceRange 字段列出价格范围内的宠物

Mutation.createPet

从 AWS AppSync 控制台的架构编辑器中，在右侧为 `createPet(input: CreatePetInput!): Pet` 选择附加解析器。选择您的 RDS 数据源。在 request mapping template (请求映射模板) 部分中，添加以下模板：

```
#set($id=$utils.autoId())
{
"version": "2018-05-29",
    "statements": [
        "insert into Pets VALUES (:ID, :TYPE, :PRICE)",
```

```
        "select * from Pets WHERE id = :ID"
    ],
    "variableMap": {
        ":ID": "$ctx.args.input.id",
        ":TYPE": $util.toJson($ctx.args.input.type),
        ":PRICE": $util.toJson($ctx.args.input.price)
    }
}
```

SQL 语句将根据语句数组中的顺序依次执行。结果将以相同的顺序返回。由于这是一个变更，我们在 insert 后面运行 select 语句以检索提交的值，以便填充 GraphQL 响应映射模板。

在 response mapping template (响应映射模板) 部分中，添加以下模板：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

由于语句有两个 SQL 查询，我们需要指定从具有 `$utils.rds.toJsonString($ctx.result))[1][0])` 的数据库返回的矩阵中的第二个结果。

Mutation.updatePet

从 AWS AppSync 控制台的架构编辑器中，在右侧为 `updatePet(input: UpdatePetInput!): Pet` 选择附加解析器。选择您的 RDS 数据源。在 request mapping template (请求映射模板) 部分中，添加以下模板：

```
{
"version": "2018-05-29",
    "statements": [
        $util.toJson("update Pets set type=:TYPE, price=:PRICE WHERE id=:ID"),
        $util.toJson("select * from Pets WHERE id = :ID")
    ],
    "variableMap": {
        ":ID": "$ctx.args.input.id",
        ":TYPE": $util.toJson($ctx.args.input.type),
        ":PRICE": $util.toJson($ctx.args.input.price)
    }
}
```

在 response mapping template (响应映射模板) 部分中，添加以下模板：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

Mutation.deletePet

从 AWS AppSync 控制台的架构编辑器中，在右侧为 `deletePet(input: DeletePetInput!): Pet` 选择附加解析器。选择您的 RDS 数据源。在 request mapping template (请求映射模板) 部分中，添加以下模板：

```
{
"version": "2018-05-29",
    "statements": [
        $util.toJson("select * from Pets WHERE id=:ID"),
        $util.toJson("delete from Pets WHERE id=:ID")
    ],
    "variableMap": {
        ":ID": "$ctx.args.input.id"
    }
}
```

在 response mapping template (响应映射模板) 部分中，添加以下模板：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.getPet

现在，已为您的架构创建变更，我们将连接三个查询以展示如何获取各个项目和列表以及应用 SQL 筛选。从 AWS AppSync 控制台的架构编辑器中，在右侧为 `getPet(id: ID!): Pet` 选择附加解析器。选择您的 RDS 数据源。在 request mapping template (请求映射模板) 部分中，添加以下模板：

```
{
"version": "2018-05-29",
        "statements": [
            $util.toJson("select * from Pets WHERE id=:ID")
    ],
    "variableMap": {
        ":ID": "$ctx.args.id"
    }
}
```

在 response mapping template (响应映射模板) 部分中，添加以下模板：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.listPets

从 AWS AppSync 控制台的架构编辑器中，在右侧为 `getPet(id: ID!): Pet` 选择附加解析器。选择您的 RDS 数据源。在 request mapping template (请求映射模板) 部分中，添加以下模板：

```
{
    "version": "2018-05-29",
    "statements": [
        "select * from Pets"
    ]
}
```

在 response mapping template (响应映射模板) 部分中，添加以下模板：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

Query.listPetsByPriceRange

从 AWS AppSync 控制台的架构编辑器中，在右侧为 `getPet(id: ID!): Pet` 选择附加解析器。选择您的 RDS 数据源。在 request mapping template (请求映射模板) 部分中，添加以下模板：

```
{
    "version": "2018-05-29",
    "statements": [
            "select * from Pets where price > :MIN and price < :MAX"
    ],

    "variableMap": {
        ":MAX": $util.toJson($ctx.args.max),
        ":MIN": $util.toJson($ctx.args.min)
    }
}
```

在 response mapping template (响应映射模板) 部分中，添加以下模板：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

运行变更

现在，您已使用 SQL 语句配置所有解析器并将 GraphQL API 连接到 Serverless Aurora 数据 API，可以开始执行变更和查询。在 AWS AppSync 控制台中，选择查询选项卡，并输入以下内容以创建一个宠物：

```
mutation add {
    createPet(input : { type:fish, price:10.0 }){
        id
        type
        price
    }
}
```

该响应包含 id、类型 和价格，如下所示：

```
{
  "data": {
    "createPet": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "type": "fish",
      "price": "10.0"
    }
  }
}
```

您可以通过运行 updatePet 变更来修改此项目：

```
mutation update {
    updatePet(input : {
        id: ID_PLACEHOLDER,
        type:bird,
        price:50.0
    }){
        id
        type
        price
    }
}
```

请注意，我们使用了以前从 createPet 操作返回的 id。当解析器利用 `$util.autoId()` 时，这将是您的记录的唯一值。您可以通过类似的方式删除记录：

```
mutation delete {
    deletePet(input : {id:ID_PLACEHOLDER}){
        id
        type
        price
    }
}
```

使用包含不同的价格 值的第一个变更创建几条记录，然后运行几个查询。

运行查询

同样，在控制台的查询选项卡中，使用以下语句列出您创建的所有记录：

```
query allpets {
    listPets {
        id
        type
        price
    }
}
```

这很好，但让我们在以下 GraphQL 查询中使用在 Query.listPetsByPriceRange 映射模板中包含 `where price > :MIN and price < :MAX` 的 SQL WHERE 谓词：

```
query petsByPriceRange {
    listPetsByPriceRange(min:1, max:11) {
        id
        type
        price
    }
}
```

您应仅看到包含高于 1 美元或低于 10 美元的价格 的记录。最后，您可以执行查询以检索各个记录，如下所示：

```
query onePet {
```

```
    getPet(id:ID_PLACEHOLDER){
        id
        type
        price
    }
}
```

输入净化

我们建议开发人员使用 variableMap 以防范 SQL 注入攻击。如果不使用变量映射，则由开发人员负责清理其 GraphQL 操作的参数。执行此操作的一种方法是在对数据 API 执行 SQL 语句之前，在请求映射模板中提供特定于输入的验证步骤。让我们看看如何修改 listPetsByPriceRange 示例的请求映射模板。您可以执行以下操作，而不是仅依赖于用户输入：

```
#set($validMaxPrice = $util.matches("\d{1,3}[,\\.]?(\\d{1,2})?",$ctx.args.maxPrice))

#set($validMinPrice = $util.matches("\d{1,3}[,\\.]?(\\d{1,2})?",$ctx.args.minPrice))


#if (!$validMaxPrice || !$validMinPrice)
    $util.error("Provided price input is not valid.")
#end
{
    "version": "2018-05-29",
    "statements": [
            "select * from Pets where price > :MIN and price < :MAX"
    ],

    "variableMap": {
        ":MAX": $util.toJson($ctx.args.maxPrice),
        ":MIN": $util.toJson($ctx.args.minPrice)
    }
}
```

在对数据 API 执行解析器时防止恶意输入的另一种方法是，将预编译语句与存储过程和参数化输入一起使用。例如，在 listPets 的解析器中，定义以下将 select 作为预编译语句执行的过程：

```
CREATE PROCEDURE listPets (IN type_param VARCHAR(200))
  BEGIN
     PREPARE stmt FROM 'SELECT * FROM Pets where type=?';
     SET @type = type_param;
     EXECUTE stmt USING @type;
```

```
    DEALLOCATE PREPARE stmt;
  END
```

可以使用以下 execute sql 命令在 Aurora Serverless 实例中创建此内容：

```
aws rds-data execute-statement --resource-arn "arn:aws:rds:us-
east-1:xxxxxxxxxxxx:cluster:http-endpoint-test" \
--schema "mysql"  --secret-arn "arn:aws:secretsmanager:us-
east-1:xxxxxxxxxxxx:secret:httpendpoint-xxxxxx"  \
--region us-east-1  --database "DB_NAME" \
--sql "CREATE PROCEDURE listPets (IN type_param VARCHAR(200)) BEGIN PREPARE stmt FROM
 'SELECT * FROM Pets where type=?'; SET @type = type_param; EXECUTE stmt USING @type;
 DEALLOCATE PREPARE stmt; END"
```

由于我们现在只是调用了存储过程，因此简化了为 listPets 生成的解析器代码。至少，任何字符串输入都应该有单引号进行转义。

```
#set ($validType = $util.isString($ctx.args.type) && !
$util.isNullOrBlank($ctx.args.type))
#if (!$validType)
    $util.error("Input for 'type' is not valid.", "ValidationError")
#end

{
    "version": "2018-05-29",
    "statements": [
        "CALL listPets(:type)"
    ]
    "variableMap": {
        ":type": $util.toJson($ctx.args.type.replace("'", "''"))
    }
}
```

转义字符串

单引号表示 SQL 语句中字符串文字的开始和结束，例如，`'some string value'`。要允许在字符串中使用具有一个或多个单引号字符 (') 的字符串值，必须用两个单引号 ('') 替换每个字符串值。例如，如果输入字符串是 `Nadia's dog`，您可以针对 SQL 语句将其转义，例如

```
update Pets set type='Nadia''s dog' WHERE id='1'
```

教程：管道解析器

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

AWS AppSync 提供了一种简单方法，以通过单位解析器将 GraphQL 字段连接到单个数据源。但是，执行单个操作可能还不够。管道解析器提供了对数据源连续执行操作的能力。在 API 中创建函数并将这些函数附加到管道解析器。每个函数执行结果将通过管道传输到下一个函数，直到没有要执行的函数为止。通过使用管道解析器，您现在可以直接在 AWS AppSync 中构建更复杂的工作流。在本教程中，您将构建一个简单的图片查看应用程序，用户可以在其中发布图片和查看其好友发布的图片。

一键设置

如果要在 AWS AppSync 中自动设置 GraphQL 终端节点并配置所有解析器和所需的 AWS 资源，您可以使用以下 AWS CloudFormation 模板：

此堆栈在您的账户中创建以下资源：

- AWS AppSync 的 IAM 角色，用于访问您的账户中的资源
- 2 个 DynamoDB 表
- 1 个 Amazon Cognito 用户池
- 2 个 Amazon Cognito 用户池组
- 3 个 Amazon Cognito 用户池用户
- 1 个 AWS AppSync API

在 AWS CloudFormation 堆栈创建过程结束时，对于创建的三个 Amazon Cognito 用户，您分别收到一封电子邮件。每封电子邮件包含一个临时密码，您可以使用该密码以 Amazon Cognito 用户身份登录到 AWS AppSync 控制台。保存密码完成本教程的剩余部分。

手动设置

如果您希望通过 AWS AppSync 控制台手动执行分步过程，请按照下面的设置过程进行操作。

设置您的非 AWS AppSync 资源

该 API 与两个 DynamoDB 表进行通信：pictures 表存储图片，friends 表存储用户之间的关系。此 API 配置为使用 Amazon Cognito 用户池作为身份验证类型。以下 AWS CloudFormation 堆栈在账户中设置这些资源。

在 AWS CloudFormation 堆栈创建过程结束时，对于创建的三个 Amazon Cognito 用户，您分别收到一封电子邮件。每封电子邮件包含一个临时密码，您可以使用该密码以 Amazon Cognito 用户身份登录到 AWS AppSync 控制台。保存密码完成本教程的剩余部分。

创建您的 GraphQL API

要在 AWS AppSync 中创建 GraphQL API，请执行以下操作：

1. 打开 AWS AppSync 控制台并选择从头开始构建，然后选择开始。
2. 将 API 的名称设置为 `AppSyncTutorial-PicturesViewer`。
3. 选择创建。

AWS AppSync 控制台使用 API 密钥身份验证模式为您创建新的 GraphQL API。您可以根据本教程后面的说明，使用控制台设置 GraphQL API 的其余部分，并针对它运行查询。

配置 GraphQL API

您需要为 AWS AppSync API 配置刚创建的 Amazon Cognito 用户池。

1. 选择 Settings 选项卡。
2. 在 Authorization Type (授权类型) 部分下，选择 Amazon Cognito User Pool (Amazon Cognito 用户池)。
3. 在用户池配置下面，为 AWS 区域选择 US-WEST-2。
4. 选择 AppSyncTutorial-UserPool 用户池。
5. 选择 DENY 作为默认操作。
6. 将 AppId client regex (AppId 客户端正则表达式) 字段保留为空。
7. 选择保存。

此 API 现在设置为使用 Amazon Cognito 用户池作为其授权类型。

为 DynamoDB 表配置数据源

在创建 DynamoDB 表后，在控制台中导航到您的 AWS AppSync GraphQL API 并选择数据源选项卡。现在，您将在 AWS AppSync 中为刚创建的每个 DynamoDB 表创建一个数据源。

1. 选择 Data source (数据源) 选项卡。
2. 选择 New (新建) 创建新的数据源。
3. 对于数据源名称，输入 `PicturesDynamoDBTable`。
4. 对于数据源类型，选择 Amazon DynamoDB 表。
5. 对于区域，选择 US-WEST-2。
6. 从表列表中，选择 AppSyncTutorial-PicturesDynamoDB 表。
7. 在创建或使用现有角色部分中，选择现有角色。
8. 选择刚刚通过 CloudFormation 模板创建的角色。如果您未更改 ResourceNamePrefix，则角色的名称应为 AppSyncTutorial-DynamoDBRole。
9. 选择创建。

对好友表重复相同的过程，DynamoDB 表的名称应为 AppSyncTutorial-Friends，前提是您在创建 CloudFormation 堆栈时未更改 ResourceNamePrefix 参数。

创建 GraphQL 架构

数据源现已连接到您的 DynamoDB 表，让我们创建一个 GraphQL 架构。从 AWS AppSync 控制台的架构编辑器中，确保您的架构与以下架构匹配：

```
schema {
    query: Query
    mutation: Mutation
}

type Mutation {
    createPicture(input: CreatePictureInput!): Picture!
    @aws_auth(cognito_groups: ["Admins"])
    createFriendship(id: ID!, target: ID!): Boolean
    @aws_auth(cognito_groups: ["Admins"])
}
```

```
type Query {
    getPicturesByOwner(id: ID!): [Picture]
    @aws_auth(cognito_groups: ["Admins", "Viewers"])
}

type Picture {
    id: ID!
    owner: ID!
    src: String
}

input CreatePictureInput {
    owner: ID!
    src: String!
}
```

选择 Save Schema (保存架构) 以保存您的架构。

已使用 @aws_auth 指令对一些架构字段进行注释。由于 API 默认操作配置设置为 DENY，因此此 API 将拒绝不属于 @aws_auth 指令中提及的组成员的所有用户。有关如何保护您的 API 的更多信息，您可以阅读安全性页面。在此情况下，仅管理员用户有权访问 Mutation.createPicture 和 Mutation.createFriendship 字段，而作为 Admins 或 Viewers 组成员的用户可访问 Query.getPicturesByOwner 字段。所有其他用户都没有访问权限。

配置解析器

现在，您有一个有效的 GraphQL 架构和两个数据源，可以将解析器附加到架构上的 GraphQL 字段。此 API 提供以下功能：

- 通过 Mutation.createPicture 字段创建图片
- 通过 Mutation.createFriendship 字段创建友好关系
- 通过 Query.getPicture 字段检索图片

Mutation.createPicture

从 AWS AppSync 控制台的架构编辑器中，在右侧为 `createPicture(input: CreatePictureInput!): Picture!` 选择附加解析器。选择 DynamoDB PicturesDynamoDBTable 数据源。在 request mapping template (请求映射模板) 部分中，添加以下模板：

```
#set($id = $util.autoId())

{
    "version" : "2018-05-29",

    "operation" : "PutItem",

    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($id),
        "owner": $util.dynamodb.toDynamoDBJson($ctx.args.input.owner)
    },

    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args.input)
}
```

在 response mapping template (响应映射模板) 部分中，添加以下模板：

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

创建图片功能已完成。将图片保存在图片表中，使用随机生成的 UUID 作为图片的 ID 并使用 Cognito 用户名作为图片拥有者。

Mutation.createFriendship

从 AWS AppSync 控制台的架构编辑器中，在右侧为 `createFriendship(id: ID!, target: ID!): Boolean` 选择附加解析器。选择 DynamoDB FriendsDynamoDBTable 数据源。在 request mapping template (请求映射模板) 部分中，添加以下模板：

```
#set($userToFriendFriendship = { "userId" : "$ctx.args.id", "friendId":
 "$ctx.args.target" })
#set($friendToUserFriendship = { "userId" : "$ctx.args.target", "friendId":
 "$ctx.args.id" })
#set($friendsItems = [$util.dynamodb.toMapValues($userToFriendFriendship),
 $util.dynamodb.toMapValues($friendToUserFriendship)])

{
    "version" : "2018-05-29",
    "operation" : "BatchPutItem",
    "tables" : {
```

```
        ## Replace 'AppSyncTutorial-' default below with the ResourceNamePrefix you
 provided in the CloudFormation template
        "AppSyncTutorial-Friends": $util.toJson($friendsItems)
    }
}
```

重要提示：在 BatchPutItem 请求模板中，应该存在 DynamoDB 表的确切名称。默认表名称为 AppSyncTutorial-Friends。如果您使用错误的表名称，则将在 AppSync 尝试代入提供的角色时收到错误。

为了简化本教程，请将友好关系请求视为已批准，并将友好关系条目直接保存到 AppSyncTutorialFriends 表中。

实际上，您将为每个友好关系存储两个项目，因为此关系是双向的。有关表示多对多关系的 Amazon DynamoDB 最佳实践的更多详细信息，请参阅 DynamoDB 最佳实践。

在 response mapping template (响应映射模板) 部分中，添加以下模板：

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
true
```

注意：请确保请求模板包含正确的表名称。默认名称为 AppSyncTutorial-Friends，但如果您更改了 CloudFormation ResourceNamePrefix 参数，则表名称可能不同。

Query.getPicturesByOwner

现在，您已具有友好关系和图片，需要为用户提供查看其好友图片的功能。要满足此要求，您需要先确认请求者是拥有者的好友，最后查询图片。

由于此功能需要两个数据源操作，因此您将创建两个函数。第一个函数 isFriend 将检查请求者和拥有者是否为好友。第二个函数 getPicturesByOwner 在给定所有者 ID 的情况下检索所请求的图片。让我们看看 Query.getPicturesByOwner 字段中有关建议的解析器的以下执行流：

1. 之前映射模板：准备上下文和字段输入参数。
2. isFriend 函数：检查请求者是否为图片的拥有者。如果不是，它对 friends 表执行 DynamoDB GetItem 操作，以检查请求者和所有者用户是否为好友。
3. getPicturesByOwner 函数：对 owner-index 全局二级索引执行 DynamoDB 查询操作以从 Pictures 表中检索图片。

4. 之后映射模板：映射图片结果，以便 DynamoDB 属性能够正确地映射到所需的 GraphQL 类型字段。

让我们先创建函数。

isFriend 函数

1. 选择 Functions (函数) 选项卡。
2. 选择 Create Function (创建函数) 以创建函数。
3. 对于数据源名称，输入 `FriendsDynamoDBTable`。
4. 对于函数名称，请输入 isFriend。
5. 在请求映射模板文本区域内，粘贴以下模板：

```
#set($ownerId = $ctx.prev.result.owner)
#set($callerId = $ctx.prev.result.callerId)

## if the owner is the caller, no need to make the check
#if($ownerId == $callerId)
    #return($ctx.prev.result)
#end

{
    "version" : "2018-05-29",

    "operation" : "GetItem",

    "key" : {
        "userId" : $util.dynamodb.toDynamoDBJson($callerId),
        "friendId" : $util.dynamodb.toDynamoDBJson($ownerId)
    }
}
```

6. 在响应映射模板文本区域内，粘贴以下模板：

```
#if($ctx.error)
    $util.error("Unable to retrieve friend mapping message: ${ctx.error.message}",
 $ctx.error.type)
#end

## if the users aren't friends
#if(!$ctx.result)
```

```
    $util.unauthorized()
#end

$util.toJson($ctx.prev.result)
```

7. 选择创建函数。

结果：您创建了 isFriend 函数。

getPicturesByOwner 函数

1. 选择 Functions (函数) 选项卡。
2. 选择 Create Function (创建函数) 以创建函数。
3. 对于数据源名称，输入 `PicturesDynamoDBTable`。
4. 对于函数名称，输入 `getPicturesByOwner`。
5. 在请求映射模板文本区域内，粘贴以下模板：

```
{
    "version" : "2018-05-29",

    "operation" : "Query",

    "query" : {
        "expression": "#owner = :owner",
        "expressionNames": {
            "#owner" : "owner"
        },
        "expressionValues" : {
            ":owner" : $util.dynamodb.toDynamoDBJson($ctx.prev.result.owner)
        }
    },

    "index": "owner-index"
}
```

6. 在响应映射模板文本区域内，粘贴以下模板：

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
```

```
$util.toJson($ctx.result)
```

7. 选择创建函数。

结果：您创建了 getPicturesByOwner 函数。现在已创建函数，请将管道解析器附加到 Query.getPicturesByOwner 字段。

从 AWS AppSync 控制台的架构编辑器中，在右侧为 `Query.getPicturesByOwner(id: ID!): [Picture]` 选择附加解析器。在以下页面上，选择数据源下拉列表下显示的 Convert to pipeline resolver (转换为管道解析器) 链接。对之前映射模板使用以下过程：

```
#set($result = { "owner": $ctx.args.id, "callerId": $ctx.identity.username })
$util.toJson($result)
```

在 after mapping template (之后映射模板) 部分中，使用以下过程：

```
#foreach($picture in $ctx.result.items)
    ## prepend "src://" to picture.src property
    #set($picture['src'] = "src://${picture['src']}")
#end
$util.toJson($ctx.result.items)
```

选择 Create Resolver (创建解析器)。您已成功附加您的首个管道解析器。在同一页上，添加您之前创建的两个函数。在函数部分中，选择 Add A Function (添加函数)，然后选择或键入第一个函数的名称 isFriend。通过对 getPicturesByOwner 函数执行相同的过程来添加第二个函数。确保 isFriend 函数在列表中先于 getPicturesByOwner 函数显示。您可以使用向上和向下箭头在管道中重新排列函数的执行顺序。

现在，已创建管道解析器并且您已附加函数，下面让我们测试新创建的 GraphQL API。

测试您的 GraphQL API

首先，您需要通过使用创建的管理员用户执行一些变更来填充图片和友好关系。在 AWS AppSync 控制台左侧，选择查询选项卡。

createPicture 变更

1. 在 AWS AppSync 控制台中，选择查询选项卡。
2. 选择 Login With User Pools (使用用户池登录)。

3. 在模态中，输入 CloudFormation 堆栈创建的 Cognito 示例客户端 ID，例如 37solo6mmhh7k4v63cqdfgdg5d。
4. 输入您作为参数传递到 CloudFormation 堆栈的用户名。默认值为 nadia。
5. 使用发送至您作为参数传递到 CloudFormation 堆栈的电子邮件（例如，UserPoolUserEmail）的临时密码。
6. 选择登录。现在，您应看到重命名为 Logout nadia 的按钮，或您创建 CloudFormation 堆栈时选择的任何用户名（即 UserPoolUsername）。

让我们发送一些 createPicture 变更来填充“图片”表。在控制台中执行以下 GraphQL 查询：

```
mutation {
  createPicture(input:{
    owner: "nadia"
    src: "nadia.jpg"
  }) {
    id
    owner
    src
  }
}
```

响应看上去应与下内容类似：

```
{
  "data": {
    "createPicture": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "owner": "nadia",
      "src": "nadia.jpg"
    }
  }
}
```

让我们再添加几张图片：

```
mutation {
  createPicture(input:{
    owner: "shaggy"
    src: "shaggy.jpg"
```

```
  }) {
    id
    owner
    src
  }
}
```

```
mutation {
  createPicture(input:{
    owner: "rex"
    src: "rex.jpg"
  }) {
    id
    owner
    src
  }
}
```

您已以管理员用户身份使用 nadia 添加三张图片。

createFriendship 变更

让我们添加友好关系条目。在控制台中执行以下变更。

注意：您仍必须以管理员用户身份（默认管理员用户为 nadia）登录。

```
mutation {
  createFriendship(id: "nadia", target: "shaggy")
}
```

响应应该类似于：

```
{
  "data": {
    "createFriendship": true
  }
}
```

nadia 和 shaggy 是好友。rex 与任何人都不是好友。

getPicturesByOwner 查询

在此步骤中，以 nadia 用户身份使用 Cognito 用户池和本教程开头设置的凭证登录。以 nadia 身份检索 shaggy 拥有的图片。

```
query {
    getPicturesByOwner(id: "shaggy") {
        id
        owner
        src
    }
}
```

由于 nadia 和 shaggy 是好友，因此查询应返回对应的图片。

```
{
  "data": {
    "getPicturesByOwner": [
      {
        "id": "05a16fba-cc29-41ee-a8d5-4e791f4f1079",
        "owner": "shaggy",
        "src": "src://shaggy.jpg"
      }
    ]
  }
}
```

同样，如果 nadia 尝试检索自己的图片，也会成功。管道解析器已经过优化来避免在此情况下运行 isFriend GetItem 操作。尝试以下查询：

```
query {
    getPicturesByOwner(id: "nadia") {
        id
        owner
        src
    }
}
```

如果您在 API 中启用日志记录（在 Settings (设置) 窗格中），将调试级别设置为 ALL (所有)，并再次运行相同的查询，则查询将返回字段执行的日志。通过查看日志，您可以确定 isFriend 函数是否在请求映射模板阶段提前返回：

```
{
  "errors": [],
  "mappingTemplateType": "Request Mapping",
  "path": "[getPicturesByOwner]",
  "resolverArn": "arn:aws:appsync:us-west-2:XXXX:apis/XXXX/types/Query/fields/
getPicturesByOwner",
  "functionArn": "arn:aws:appsync:us-west-2:XXXX:apis/XXXX/functions/
o2f42p2jrfdl3dw7s6xub2csdfs",
  "functionName": "isFriend",
  "earlyReturnedValue": {
    "owner": "nadia",
    "callerId": "nadia"
  },
  "context": {
    "arguments": {
      "id": "nadia"
    },
    "prev": {
      "result": {
        "owner": "nadia",
        "callerId": "nadia"
      }
    },
    "stash": {},
    "outErrors": []
  },
  "fieldInError": false
}
```

earlyReturnedValue 键表示 #return 指令所返回的数据。

最后，即使 rex 是 Viewers Cognito UserPool 组的成员，但由于 rex 不是任何人的好友，因此他无法访问 shaggy 或 nadia 拥有的任何图片。如果您以 rex 身份登录控制台并执行以下查询：

```
query {
    getPicturesByOwner(id: "nadia") {
        id
        owner
        src
    }
}
```

您将收到以下未经授权错误：

```
{
  "data": {
    "getPicturesByOwner": null
  },
  "errors": [
    {
      "path": [
        "getPicturesByOwner"
      ],
      "data": null,
      "errorType": "Unauthorized",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
          "column": 9,
          "sourceName": null
        }
      ],
      "message": "Not Authorized to access getPicturesByOwner on type Query"
    }
  ]
}
```

您已使用管道解析器成功实现复杂的授权。

教程：增量同步

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

AWS AppSync 中的客户端应用程序在移动/Web 应用程序中将 GraphQL 响应缓存到本地磁盘以存储数据。版本化的数据源和 Sync 操作使客户能够使用单个解析器执行同步过程。这使客户端能够将其本地缓存与来自一个基本查询的结果（可能包含大量记录）混合，然后仅接收自上次查询以来更改的数据（增量更新）。通过允许客户端使用初始请求和另一个请求中的增量更新来执行缓存的基本组合，您可

以将计算工作从客户端应用程序移至后端。这对于经常在联机状态和脱机状态之间切换的客户端应用程序来说效率更高。

为了实现增量同步，Sync 查询会对版本化数据源使用 Sync 操作。在 AWS AppSync 变更更改版本控制的数据源中的项目时，该更改的记录也会存储在增量 表中。您可以选择对其他版本控制的数据源使用不同的增量 表（例如，每种类型一个增量表，或者每个域区域一个增量表），或者对您的 API 使用单个增量 表。AWSAppSync 建议不要对多个 API 使用单个增量 表，以避免主键冲突。

此外，增量同步客户端还可以接收订阅作为参数，然后客户端协调订阅在脱机和联机转换之间重新连接和写入。增量同步通过自动恢复订阅（包括指数回退和通过各种网络错误情况的抖动重试）并将事件存储在队列中来执行此操作。然后，在合并队列中的任何事件之前运行适当的增量或基本查询，最后正常处理订阅。

客户端配置选项（包括 Amplify 数据存储）的文档可在Amplify 框架网站上查阅。本文档概述了如何设置版本化的 DynamoDB 数据源和 Sync 操作，以便与增量同步客户端一起使用，实现最佳的数据访问。

一键设置

要在 AWS AppSync 中自动设置 GraphQL 终端节点并配置所有解析器和所需的 AWS 资源，请使用以下 AWS CloudFormation 模板：

此堆栈在您的账户中创建以下资源：

- 2 个 DynamoDB 表（基表和增量表）
- 1 个 AWS AppSync API（具有 API 密钥）
- 1 个 IAM 角色（具有 DynamoDB 表策略）

两个表用于将您的同步查询分区到另一个表中，此表在客户端处于脱机状态时充当错过事件的日志。为了使查询在增量表上保持高效，将使用 Amazon DynamoDB TTL 来根据需要自动整理事件。可以根据您对数据源的需求配置 TTL 时间（您可能希望这是 1 小时，1 天等）。

架构

为了说明增量同步，示例应用程序创建了一个由 DynamoDB 中的基 表和增量 表支持的 Posts 架构。AWSAppSync 自动将变更写入到两个表中。同步查询将根据情况从基本 或增量 表中拉取记录，并定义单个订阅，以说明客户端如何在其重新连接逻辑中利用它。

```
input CreatePostInput {
    author: String!
    title: String!
    content: String!
    url: String
    ups: Int
    downs: Int
    _version: Int
}

interface Connection {
  nextToken: String
  startedAt: AWSTimestamp!
}

type Mutation {
    createPost(input: CreatePostInput!): Post
    updatePost(input: UpdatePostInput!): Post
    deletePost(input: DeletePostInput!): Post
}

type Post {
    id: ID!
    author: String!
    title: String!
    content: String!
    url: AWSURL
    ups: Int
    downs: Int
    _version: Int
    _deleted: Boolean
    _lastChangedAt: AWSTimestamp!
}

type PostConnection implements Connection {
    items: [Post!]!
    nextToken: String
    startedAt: AWSTimestamp!
}

type Query {
    getPost(id: ID!): Post
    syncPosts(limit: Int, nextToken: String, lastSync: AWSTimestamp): PostConnection!
```

```
}

type Subscription {
    onCreatePost: Post
        @aws_subscribe(mutations: ["createPost"])
    onUpdatePost: Post
        @aws_subscribe(mutations: ["updatePost"])
    onDeletePost: Post
        @aws_subscribe(mutations: ["deletePost"])
}

input DeletePostInput {
    id: ID!
    _version: Int!
}

input UpdatePostInput {
    id: ID!
    author: String
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    _version: Int!
}

schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
}
```

GraphQL 架构是标准的，但在继续之前需注意以下几点。首先，所有变更都将先自动写入基本 表，然后写入增量 表。基本 表是状态的可信中央来源，而 增量 表是您的日志。如果未传入 `lastSync: AWSTimestamp`，将对基 表运行 `syncPosts` 查询将数据加载到缓存中，以及作为全局同步过程定期运行以处理边缘情况，即，客户端离线时间超过您在增量 表中配置的 TTL 时间。如果您传入 `lastSync: AWSTimestamp`，则此 `syncPosts` 查询将针对增量 表运行并由客户端用于检索自它们上次脱机以后更改的事件。Amplify 客户端会自动传递该 `lastSync: AWSTimestamp` 值，并相应地保存到磁盘。

Post 上的 _deleted 字段用于 DELETE 操作。当客户端处于脱机状态并且从基本 表中删除记录时，此属性将通知客户端执行同步以移出其本地缓存中的项目。如果客户端脱机较长时间并且在客户端可以使用增量同步查询检索此值之前已删除该项目，则基本查询中的全局捕获事件（可在客户端中配置）将运行，并且会从缓存中删除该项目。此字段标记为可选，因为它仅在运行包含已删除项目的同步查询时返回值。

变更

对于所有变更，AWS AppSync 在基 表中执行标准创建/更新/删除操作，并自动在增量 表中记录更改。您可以通过修改数据源上的 DeltaSyncTableTTL 值来减少或延长保留记录的时间。对于拥有高速数据的组织，短时间保留记录可能是有意义的。另外，如果您的客户端长时间处于脱机状态，则最好是将记录保留较长时间。

同步查询

基本查询 是未指定 lastSync 值的 DynamoDB 同步操作。对于许多组织而言，这是有效的，因为基本查询仅在启动时运行，之后将定期运行。

增量查询 是指定了 lastSync 值的 DynamoDB 同步操作。每当客户端从脱机状态恢复联机状态时，就会执行增量查询（只要基本查询周期时间未触发运行）。客户端会自动跟踪上次成功运行查询以同步数据的时间。

运行增量查询时，查询的解析器使用 ds_pk 和 ds_sk，仅查询自客户端上次执行同步以来发生更改的记录。客户端将存储相应的 GraphQL 响应。

有关执行同步查询的详细信息，请参阅 同步操作文档。

示例

让我们首先调用一个 createPost 变更来创建一个项目：

```
mutation create {
  createPost(input: {author: "Nadia", title: "My First Post", content: "Hello World"})
 {
    id
    author
    title
    content
    _version
    _lastChangedAt
```

```
    _deleted
  }
}
```

此变更的返回值如下所示：

```
{
  "data": {
    "createPost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
      "author": "Nadia",
      "title": "My First Post",
      "content": "Hello World",
      "_version": 1,
      "_lastChangedAt": 1574469356331,
      "_deleted": null
    }
  }
}
```

如果您检查基本 表的内容，将看到一条如下所示的记录：

```
{
  "_lastChangedAt": {
    "N": "1574469356331"
  },
  "_version": {
    "N": "1"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "My First Post"
  }
}
```

如果您检查增量 表的内容，将看到一条如下所示的记录：

```
{
  "_lastChangedAt": {
    "N": "1574469356331"
  },
  "_ttl": {
    "N": "1574472956"
  },
  "_version": {
    "N": "1"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "ds_pk": {
    "S": "AppSync-delta-sync-post:2019-11-23"
  },
  "ds_sk": {
    "S": "00:35:56.331:81d36bbb-1579-4efe-92b8-2e3f679f628b:1"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "My First Post"
  }
}
```

现在我们可以模拟一个基本 查询，客户端将运行该查询来组合其本地数据存储，并使用如下所示的 syncPosts 查询：

```
query baseQuery {
  syncPosts(limit: 100, lastSync: null, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
```

```
      _lastChangedAt
    }
    startedAt
    nextToken
  }
}
```

此基本 查询的返回值如下所示：

```
{
  "data": {
    "syncPosts": {
      "items": [
        {
          "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
          "author": "Nadia",
          "title": "My First Post",
          "content": "Hello World",
          "_version": 1,
          "_lastChangedAt": 1574469356331
        }
      ],
      "startedAt": 1574469602238,
      "nextToken": null
    }
  }
}
```

我们稍后会保存 startedAt 值来模拟增量 查询，但首先我们需要对表进行更改。让我们使用 updatePost 变更来修改我们现有的文章：

```
mutation updatePost {
  updatePost(input: {id: "81d36bbb-1579-4efe-92b8-2e3f679f628b", _version: 1, title:
 "Actually this is my Second Post"}) {
    id
    author
    title
    content
    _version
    _lastChangedAt
    _deleted
  }
```

```
}
```

此变更的返回值如下所示：

```
{
  "data": {
    "updatePost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
      "author": "Nadia",
      "title": "Actually this is my Second Post",
      "content": "Hello World",
      "_version": 2,
      "_lastChangedAt": 1574469851417,
      "_deleted": null
    }
  }
}
```

如果您现在检查基本 表的内容，则应该看到更新后的项目：

```
{
  "_lastChangedAt": {
    "N": "1574469851417"
  },
  "_version": {
    "N": "2"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "Actually this is my Second Post"
  }
}
```

如果您现在检查增量 表的内容，则应看到两条记录：

1. 创建项目时的记录
2. 项目更新时间的记录。

新项目将如下所示：

```
{
  "_lastChangedAt": {
    "N": "1574469851417"
  },
  "_ttl": {
    "N": "1574473451"
  },
  "_version": {
    "N": "2"
  },
  "author": {
    "S": "Nadia"
  },
  "content": {
    "S": "Hello World"
  },
  "ds_pk": {
    "S": "AppSync-delta-sync-post:2019-11-23"
  },
  "ds_sk": {
    "S": "00:44:11.417:81d36bbb-1579-4efe-92b8-2e3f679f628b:2"
  },
  "id": {
    "S": "81d36bbb-1579-4efe-92b8-2e3f679f628b"
  },
  "title": {
    "S": "Actually this is my Second Post"
  }
}
```

现在，我们可以模拟增量 查询来检索客户端脱机时发生的修改。我们将使用从基本 查询返回的 `startedAt` 值发出请求：

```
query delta {
  syncPosts(limit: 100, lastSync: 1574469602238, nextToken: null) {
    items {
      id
```

```
      author
      title
      content
      _version
    }
    startedAt
    nextToken
  }
}
```

此增量 查询的返回值如下所示：

```
{
  "data": {
    "syncPosts": {
      "items": [
        {
          "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
          "author": "Nadia",
          "title": "Actually this is my Second Post",
          "content": "Hello World",
          "_version": 2
        }
      ],
      "startedAt": 1574470400808,
      "nextToken": null
    }
  }
}
```

配置和设置

AWS AppSync 让您能够：

- 缓存经常请求但不太可能随请求变化的数据。这可以减少解析器上的负载。有关更多信息，请参阅the section called “缓存和压缩”。
- 对 GraphQL 对象进行版本控制，以处理和避免多个客户端之间的冲突。有关更多信息，请参阅the section called “冲突检测和同步”。
- 使用自定义域名配置一个同时适用于 GraphQL 和实时 API 的简单、易于记忆的域。有关更多信息，请参阅配置自定义域名。
- 允许通过 VPC 访问您的 GraphQL API。有关更多信息，请参阅使用 AWS AppSync 私有 API。
- 启用自省并设置每个查询的查询深度和解析器限制。有关更多信息，请参阅配置限制。

此外，AWS AppSync 包括以下用于日志记录、监控和跟踪的标准 AWS 工具：

- AWS CloudTrail 中的日志记录
- 使用 Amazon 进行监控 CloudWatch
- 使用 AWS X-Ray 进行跟踪

缓存和压缩

AWS AppSync 的服务器端数据缓存功能在内存缓存中高速提供数据，从而提高性能并减少延迟。这减少了直接访问数据源的需求。缓存适用于单位解析器和管道解析器。

AWS AppSync 还允许您压缩 API 响应，以便更快地加载和下载负载内容。这可能会减轻应用程序的压力，同时还可能会降低数据传输费用。压缩行为是可配置的，您可以自行决定进行设置。

请参阅本节以帮助在 AWS AppSync API 中定义所需的服务器端缓存和压缩行为。

实例类型

AWS AppSync 在与您的 AWS AppSync API 相同的 AWS 账户和 AWS 区域中托管 Amazon ElastiCache for Redis 实例。

可以使用以下 ElastiCache for Redis 实例类型：

small

1 个 vCPU、1.5 GiB RAM、低到中网络性能

medium

2 个 vCPU、3 GiB RAM、低到中网络性能

large

2 个 vCPU、12.3 GiB RAM，高达 10 Gb 网络性能

xlarge

4 个 vCPU、25.05 GiB RAM，高达 10 Gb 网络性能

2xlarge

8 个 vCPU、50.47 GiB RAM，高达 10 Gb 网络性能

4xlarge

16 个 vCPU、101.38 GiB RAM，高达 10 Gb 网络性能

8xlarge

32 个 vCPU、203.26 GiB RAM、10 Gb 网络性能（并非在所有区域中都提供）

12xlarge

48 个 vCPU、317.77 GiB RAM、10 Gb 网络性能

> **Note**
>
> 过去，您指定了特定的实例类型（例如 `t2.medium`）。自 2020 年 7 月起，这些旧实例类型可以继续使用，但将弃用这些实例类型而不建议使用。我们建议您使用此处描述的通用实例类型。

缓存行为

以下是与缓存相关的行为：

无

没有服务器端缓存。

完整请求缓存

如果数据没有位于缓存中，则会从数据源中检索数据并填充缓存，直到生存时间 (TTL) 到期。对您的 API 的所有后续请求都从缓存中返回。这意味着除非 TTL 到期，否则，不会直接连接到数据源。在该设置中，我们将 `context.arguments` 和 `context.identity` 映射的内容作为缓存键。

每个解析器的缓存

对于该设置，必须明确选择每个解析器以缓存响应。您可以在解析器上指定 TTL 和缓存键。您可以指定的缓存键是顶级映射 `context.arguments`、`context.source` 和 `context.identity` 以及/或者这些映射中的字符串字段。TTL 值是必需的，但缓存键是可选的。如果您未指定任何缓存键，则默认值是 `context.arguments`、`context.source` 和 `context.identity` 映射内容。

例如，您可以使用以下组合：

- context.arguments 和 context.source
- context.arguments 和 context.identity.sub
- context.arguments.id 或 context.arguments.InputType.id
- context.source.id 和 context.identity.sub
- context.identity.claims.username

在您仅指定 TTL 而没有指定缓存键时，解析器的行为与完整请求缓存相同。

缓存生存时间

该设置定义在内存中存储缓存条目的时间。最大 TTL 为 3,600 秒（1 小时），之后自动删除条目。

缓存加密

缓存加密具有以下两种形式。这些与 ElastiCache for Redis 允许的设置类似。只有在首次为 AWS AppSync API 启用缓存时，您才能启用加密设置。

- 传输中加密 - AWS AppSync、缓存和数据源（不安全的 HTTP 数据源除外）之间的请求在网络级别进行加密。由于在终端节点中加密和解密数据需要进行一些处理，因此，传输中加密可能会影响性能。
- 静态加密 - 在交换操作期间从内存保存到磁盘的数据在缓存实例中进行加密。该设置也会影响性能。

要使缓存条目无效，您可以使用 AWS AppSync 控制台或 AWS Command Line Interface (AWS CLI) 进行刷新缓存 API 调用。

有关更多信息，请参阅 AWS AppSync API 参考中的 ApiCache 数据类型。

缓存逐出

在您设置 AWS AppSync 的服务器端缓存时，您可以配置最大 TTL。该值定义在内存中存储缓存条目的时间。在必须从缓存中删除特定条目的情况下，您可以在解析器的请求或响应中使用 AWS AppSync 的 `evictFromApiCache` 扩展实用程序。（例如，如果数据源中的数据发生变化，并且缓存条目现已过时。）要从缓存中逐出某个项目，您必须知道该项目的键。因此，如果您必须动态逐出项目，我们建议使用每个解析器的缓存，并明确定义一个用于将条目添加到缓存的键。

逐出缓存条目

要从缓存中逐出项目，请使用 `evictFromApiCache` 扩展实用程序。指定类型名称和字段名称，然后提供一个键值项目对象以构建要逐出的条目的键。在该对象中，每个键表示 `context` 对象中的一个有效条目，该条目在缓存解析器的 `cachingKey` 列表中使用。每个值是用于构建键值的实际值。您必须按照与缓存解析器的 `cachingKey` 列表中的缓存键相同的顺序，将项目放入对象中。

例如，请参阅以下架构：

```
type Note {
  id: ID!
  title: String
  content: String!
}

type Query {
  getNote(id: ID!): Note
}

type Mutation {
  updateNote(id: ID!, content: String!): Note
}
```

在该示例中，您可以启用每个解析器的缓存，然后为 `getNote` 查询启用该功能。接下来，您可以将缓存键配置为由 `[context.arguments.id]` 组成。

在您尝试获取 `Note` 以构建缓存键时，AWS AppSync 使用 `getNote` 查询的 `id` 参数在其服务器端缓存中执行查找。

在您更新 `Note` 时，您必须逐出特定注释的条目，以确保下一个请求从后端数据源中获取该注释。为此，您必须创建一个请求处理程序。

以下示例说明了一种使用该方法处理逐出的方法：

```
import { util, Context } from '@aws-appsync/utils';
import { update } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 extensions.evictFromApiCache('Query', 'getNote', { 'ctx.args.id': ctx.args.id });
 return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });
}

export const response = (ctx) => ctx.result;
```

或者，您也可以在响应处理程序中处理逐出。

在处理 updateNote 变更时，AWS AppSync 尝试逐出条目。如果成功清除条目，响应将在 `extensions` 对象中包含一个 `apiCacheEntriesDeleted` 值，以显示删除的条目数：

```
"extensions": {  "apiCacheEntriesDeleted": 1}
```

根据身份逐出缓存条目

您可以根据 `context` 对象中的多个值创建缓存键。

例如，采用以下架构，该架构将 Amazon Cognito 用户池作为默认身份验证模式并由 Amazon DynamoDB 数据源提供支持：

```
type Note {
  id: ID! # a slug; e.g.: "my-first-note-on-graphql"
  title: String
  content: String!
}

type Query {
  getNote(id: ID!): Note
}

type Mutation {
  updateNote(id: ID!, content: String!): Note
}
```

Note 对象类型保存在 DynamoDB 表中。该表具有一个组合键，该组合键将 Amazon Cognito 用户名作为主键，并将 Note 的 id（短标签）作为分区键。这是一个多租户系统，允许多个用户托管和更新他们的私有 Note 对象，这些对象从不进行共享。

由于这是一个读取密集型系统，因此，getNote 查询使用每个解析器的缓存进行缓存，缓存键由 [context.identity.username, context.arguments.id] 组成。在更新 Note 后，您可以逐出该特定 Note 的条目。您必须按照解析器的 cachingKeys 列表中指定的相同顺序将组件添加到对象中。

以下示例说明了这一点：

```
import { util, Context } from '@aws-appsync/utils';
import { update } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 extensions.evictFromApiCache('Query', 'getNote', {
  'ctx.identity.username': ctx.identity.username,
  'ctx.args.id': ctx.args.id,
 });
 return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });
}

export const response = (ctx) => ctx.result;
```

后端系统也可以更新 Note 并逐出条目。例如，采用以下变更：

```
type Mutation {
  updateNoteFromBackend(id: ID!, content: String!, username: ID!): Note @aws_iam
}
```

您可以逐出条目，但将缓存键的组件添加到 cachingKeys 对象中。

在以下示例中，逐出是在解析器响应中进行的：

```
import { util, Context } from '@aws-appsync/utils';
import { update } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 extensions.evictFromApiCache('Query', 'getNote', {
  'ctx.identity.username': ctx.args.username,
  'ctx.args.id': ctx.args.id,
 });
```

```
 return update({ key: { id: ctx.args.id }, update: { context: ctx.args.content } });
}

export const response = (ctx) => ctx.result;
```

如果已在 AWS AppSync 外部更新您的后端数据，您可以调用使用 NONE 数据源的变量以从缓存中逐出项目。

压缩 API 响应

AWS AppSync 允许客户端请求压缩的负载。如果请求，将压缩并返回 API 响应，以响应指示希望压缩内容的请求。压缩的 API 响应加载速度更快，内容下载速度更快，并且您的数据传输费用也可能会下降。

> **Note**
>
> 压缩适用于在 2020 年 6 月 1 日之后创建的所有新 API。
> AWS AppSync 尽力压缩对象。在极少数情况下，AWS AppSync 可能会根据多种因素（包括当前容量）跳过压缩。

AWS AppSync 可以将 GraphQL 查询负载大小压缩到 1,000 到 10,000,000 字节之间。要启用压缩，客户端必须发送具有 `gzip` 值的 `Accept-Encoding` 标头。可以检查响应 (`gzip`) 中的 `Content-Encoding` 标头值以验证压缩。

默认情况下，AWS AppSync 控制台中的查询浏览器自动设置请求中的标头值。如果您执行的查询具有足够大的响应，可以使用浏览器开发人员工具确认压缩。

配置自定义域名

对于 AWS AppSync，您可以使用自定义域名配置一个适用于 GraphQL 和实时 API 的易于记忆的域。

换句话说，您可以创建与您的账户中的 AWS AppSync API 关联的自定义域名，以将简单且易于记忆的终端节点 URL 与所选的域名一起使用。

在配置 AWS AppSync API 时，将预置两个终端节点：

AWS AppSync GraphQL 终端节点：

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync 实时终端节点：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/
graphql
```

通过使用自定义域名，您可以使用单个域与两个终端节点进行交互。例如，如果将 `api.example.com` 配置为自定义域，您可以使用以下 URL 与 GraphQL 终端节点和实时终端节点进行交互：

AWS AppSync 自定义域 GraphQL 终端节点：

```
https://api.example.com/graphql
```

AWS AppSync 自定义域实时终端节点：

```
wss://api.example.com/graphql/realtime
```

> **Note**
>
> 对于自定义域名，AWS AppSync API 仅支持 TLS 1.2 和 TLS 1.3。

注册和配置域名

要为您的 AWS AppSync API 设置自定义域名，您必须具有注册的互联网域名。您可以使用 Amazon Route 53 domain registration 或所选的第三方域名注册商注册互联网域。有关 Route 53 的更多信息，请参阅《Amazon Route 53 开发人员指南》中的什么是 Amazon Route 53？

API 的自定义域名可以是注册的互联网域的子域或根域（也称为“机构根网域”）名称。在 AWS AppSync 中创建自定义域名后，您必须创建或更新 DNS 提供程序的资源记录以映射到您的 API 终端节点。如果没有该映射，发送到自定义域名的 API 请求将无法到达 AWS AppSync。

在 AWS AppSync 中创建自定义域名

在为 AWS AppSync API 创建自定义域名时，将会设置一个 Amazon CloudFront 分配。您必须设置 DNS 记录，以将自定义域名映射到 CloudFront 分配域名。需要具有该映射，才能通过映射的 CloudFront 分配路由发送到 AWS AppSync 中的自定义域名的 API 请求。您还必须为自定义域名提供证书。

要设置自定义域名或更新其证书，您必须有权更新 CloudFront 分配并描述您计划使用的 AWS Certificate Manager (ACM) 证书。要授予这些权限，请将以下 AWS Identity and Access Management (IAM) 策略语句附加到您的账户中的 IAM 用户、组或角色：

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowUpdateDistributionForAppSyncCustomDomainName",
      "Effect": "Allow",
      "Action": ["cloudfront:updateDistribution"],
      "Resource": ["*"]
    },
    {
      "Sid": "AllowDescribeCertificateForAppSyncCustomDomainName",
      "Effect": "Allow",
      "Action": "acm:DescribeCertificate",
      "Resource": "arn:aws:acm:<region>:<account-id>:certificate/<certificate-id>"
    }
  ]
}
```

AWS AppSync 在 CloudFront 分配上使用服务器名称指示 (SNI) 以支持自定义域名。有关在 CloudFront 分配上使用自定义域名的更多信息（包括所需的证书格式和最大证书密钥长度），请参阅《Amazon CloudFront 开发人员指南》中的将 HTTPS 与 CloudFront 结合使用。

要将自定义域名设置为 API 的主机名，API 所有者必须为自定义域名提供 SSL/TLS 证书。要提供证书，请执行以下操作之一：

- 在 `us-east-1` AWS 区域（美国东部（弗吉尼亚州北部））中，在 ACM 中请求新证书，或者将第三方证书颁发机构颁发的证书导入到 ACM 中。有关 ACM 的更多信息，请参阅《AWS Certificate Manager 用户指南》中的什么是 AWS Certificate Manager？
- 提供 IAM 服务器证书。有关更多信息，请参阅《IAM 用户指南》中的在 IAM 中管理服务器证书。

AWS AppSync 中的通配符自定义域名

AWS AppSync 支持通配符自定义域名。要配置通配符自定义域名，请将一个通配符 (*) 指定为自定义域的第一个子域。它表示根域的所有可能的子域。例如，通配符自定义域名 `*.example.com` 生成 `a.example.com`、`b.example.com` 和 `c.example.com` 等子域。所有这些子域路由到同一个域。

要在 AWS AppSync 中使用通配符自定义域名，您必须提供一个由 ACM 颁发的证书，其中包含可以保护同一域中的多个站点的通配符名称。有关更多信息，请参阅《AWS Certificate Manager 用户指南》中的 ACM 证书特征。

冲突检测和同步

版本化数据源

AWS AppSync 目前支持 DynamoDB 数据源版本控制。冲突检测、冲突解决和同步操作需要 `Versioned` 数据源。在为数据源启用版本控制时，AWS AppSync 自动执行以下操作：

- 使用对象版本化元数据增强项目。
- 将使用 AWS AppSync 变更对项目所做的更改记录到增量 表中。
- 使用“逻辑删除”将基本 表中的已删除项目保留可配置的时间。

版本化数据源配置

对 DynamoDB 数据源启用版本控制时，可以指定以下字段：

`BaseTableTTL`

使用“逻辑删除”（一个元数据字段，指示项目已被删除）保留基本 表中已删除项目的分钟数。如果希望删除项目时立即将其移除，则可以将此值设置为 0。该字段为必填。

`DeltaSyncTableName`

存储使用 AWS AppSync 变更对项目所做的更改的表名称。该字段为必填。

`DeltaSyncTableTTL`

在增量 表中保留项目的分钟数。该字段为必填。

增量同步表

AWS AppSync 目前为使用 `PutItem`、`UpdateItem` 和 `DeleteItem` DynamoDB 操作的变更提供增量同步日志记录支持。

在 AWS AppSync 变更更改版本控制的数据源中的项目时，将在针对增量更新优化的增量 表中存储该更改的记录。您可以选择对其他版本控制的数据源使用不同的增量 表（例如，每种类型一个增量表，

或者每个域区域一个增量表），或者对您的 API 使用单个增量 表。AWSAppSync 建议不要对多个 API 使用单个增量 表，以避免主键冲突。

此表所需的架构如下所示：

ds_pk

用作分区键的字符串值。这是将基本数据源名称和更改发生日期的 ISO 8601 格式连接在一起而构建的（例如 `Comments:2019-01-01`）。

在 VTL 映射模板中的 `customPartitionKey` 标记设置为分区键的列名称时（请参阅《AWS AppSync 开发人员指南》中的 DynamoDB 解析器映射模板参考），`ds_pk` 格式将发生变化，字符串是通过附加基 表中的新记录的分区键值构建的。例如，如果基 表中的记录的分区键值为 `1a`，排序键值为 `2b`，则字符串的新值为：`Comments:2019-01-01:1a`。

ds_sk

用作排序键的字符串值。这是将更改发生时间的 ISO 8601 格式、项目的主键和项目的版本连接在一起而构建的。这些字段的组合保证了增量 表中的每个条目的唯一性（例如，如果时间为 `09:30:00`，ID 为`1a`，版本为 `2`，则该值为 `09:30:00:1a:2`）。

在 VTL 映射模板中的 `customPartitionKey` 标记设置为分区键的列名称时（请参阅《AWS AppSync 开发人员指南》中的 DynamoDB 解析器映射模板参考），`ds_sk` 格式将发生变化，字符串是通过将组合键的值替换为基 表中的排序键值构建的。使用上面的示例，如果基 表中的记录的分区键值为 `1a`，排序键值为 `2b`，则字符串的新值为：`09:30:00:2b:3`。

_ttl

一个数值，用于存储应从增量 表中移除项目时的时间戳（以纪元秒为单位）。此值是通过将数据源上配置的 `DeltaSyncTableTTL` 值添加到发生更改那一刻的时间来确定的。该字段应配置为 DynamoDB TTL 属性。

配置为与基本 表一起使用的 IAM 角色还必须包含对增量 表进行操作的权限。在该示例中，显示了名为 `Comments` 的基 表和名为 `ChangeLog` 的增量 表的权限策略：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
```

```
                "dynamodb:DeleteItem",
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:UpdateItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:us-east-1:000000000000:table/Comments",
                "arn:aws:dynamodb:us-east-1:000000000000:table/Comments/*",
                "arn:aws:dynamodb:us-east-1:000000000000:table/ChangeLog",
                "arn:aws:dynamodb:us-east-1:000000000000:table/ChangeLog/*"
            ]
        }
    ]
}
```

版本化数据源元数据

AWS AppSync 代表您管理 `Versioned` 数据源上的元数据字段。自行修改这些字段可能会导致应用程序中的错误或数据丢失。这些字段包括：

`_version`

一个单调递增的计数器，在项目发生更改时随时更新。

`_lastChangedAt`

一个数值，用于存储上次修改项目时的时间戳（以纪元毫秒为单位）。

`_deleted`

一个布尔型“逻辑删除”值，指示项目已被删除。应用程序可以使用此功能从本地数据存储中移除已删除的项目。

`_ttl`

一个数值，用于存储应从底层数据源中移除项目时的时间戳（以纪元秒为单位）。

`ds_pk`

用作增量 表的分区键的字符串值。

`ds_sk`

用作增量 表的排序键的字符串值。

gsi_ds_pk

生成的字符串值属性，用于支持将全局二级索引作为分区键。只有在 VTL 映射模板中启用 customPartitionKey 和 populateIndexFields 标记时，才会包含该属性（请参阅《AWS AppSync 开发人员指南》中的 DynamoDB 解析器映射模板参考）。如果启用，则将基本 数据源名称和更改发生日期的 ISO 8601 格式连接在一起以构建该值（例如，如果基表名为 Comments，该记录将设置为 Comments:2019-01-01）。

gsi_ds_sk

生成的字符串值属性，用于支持将全局二级索引作为排序键。只有在 VTL 映射模板中启用 customPartitionKey 和 populateIndexFields 标记时，才会包含该属性（请参阅《AWS AppSync 开发人员指南》中的 DynamoDB 解析器映射模板参考）。如果启用，则将更改发生时间的 ISO 8601 格式、基 表中的项目的分区键、基表中的项目的排序键以及项目的版本连接在一起以构建该值（例如，如果时间为 09:30:00，分区键值为 1a，排序键值为 2b，版本为 3，则该值为 09:30:00:1a#2b:3）。

这些元数据字段将影响基本数据源中的项目的总体大小。AWS在设计应用程序时，AppSync 建议为版本控制的数据源元数据保留 500 字节 + 最大主键大小的存储空间。要在客户端应用程序中使用此元数据，请在您的 GraphQL 类型和变更的选择集中包括 _version、_lastChangedAt 和 _deleted 字段。

冲突检测和解决

在 AWS AppSync 中发生并发写入时，您可以配置冲突检测和冲突解决策略以相应地处理更新。冲突检测确定变更是否与数据源中的实际写入项目发生冲突。通过将 conflictDetection 字段的 SyncConfig 中的值设置为 VERSION 来启用冲突检测。

冲突解决是在检测到冲突时执行的操作。这是通过在 SyncConfig 中设置冲突处理程序字段来确定的。有三种冲突解决策略：

- OPTIMISTIC_CONCURRENCY
- AUTOMERGE
- LAMBDA

下文分别详细介绍了这些解决冲突策略。

在写入操作期间，版本将由 AppSync 自动递增，而不应由客户端或在配置了版本化数据源的解析器之外进行修改。否则会改变系统的一致性行为，并可能导致数据丢失。

乐观并发

乐观并发是 AWS AppSync 为版本控制的数据源提供的一种冲突解决策略。当冲突解决程序设置为乐观并发时，如果检测到传入的变更具有与对象的实际版本不同的版本，则冲突处理程序将简单地拒绝传入请求。在 GraphQL 响应中，将提供服务器上具有最新版本的现有项目。然后，客户端需要在本地处理此冲突，并使用项目的更新版本重试变更。

Automerge

Automerge 为开发人员提供了配置冲突解决策略的简单方法，而无需编写客户端逻辑来手动合并其他策略无法处理的冲突。Automerge 在合并数据以解决冲突时遵守严格的规则集。Automerge 的原则围绕 GraphQL 字段的底层数据类型。这些原则如下所示：

- 标量字段上的冲突：GraphQL 标量或不是集合（即 List、Set、Map）的任何字段。拒绝标量字段的传入值并选择服务器中现有的值。
- 列表上的冲突：GraphQL 类型和数据库类型是列表。将传入列表与服务器中的现有列表连接起来。传入变更中的列表值将附加到服务器中列表的末尾。将保留重复的值。
- 集合上的冲突：GraphQL 类型是一个列表，数据库类型是一个集合。使用传入集合和服务器中的现有集合应用集合合并。这符合集合的属性，意味着没有重复的条目。
- 当传入变更在项目中添加新字段或针对值为 `null` 的字段进行变更时，请将其合并到现有项目中。
- 映射上的冲突：当数据库中的底层数据类型是映射（即键值文档）时，在解析和处理映射的每个属性时应用上述规则。

Automerge 旨在使用更新版本自动检测、合并和重试请求，从而使客户端无需手动合并任何冲突的数据。

为了显示 Automerge 如何处理标量类型上的冲突的示例，我们将使用以下记录作为起点。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 4
}
```

现在，传入的变更可能正在尝试更新该项目，但使用的是较旧版本，因为客户端尚未与服务器同步。它类似于以下内容：

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 55,
  "_version" : 2
}
```

请注意传入请求中的过时版本 2。在此流程中，Automerge 将通过拒绝“球衣”字段更新为“55”来合并数据，并将其值保持在“5”，从而导致下面的项目图像保存在服务器中。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 5 # version is incremented every time automerge performs a merge that is
 stored on the server.
}
```

鉴于上面显示的具有版本 5 的项目状态，现在假设一个传入的变更尝试使用以下图像改变项目：

```
{
  "id" : 1,
  "name" : "Shaggy",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner"] # underlying data type is a Set
  "points": [24, 30, 27] # underlying data type is a List
  "_version" : 3
}
```

传入的变更中有三个兴趣点。名称（一个标量）已被更改，但添加了两个新字段“兴趣”（集合）和“分数”（列表）。在这种情况下，将检测到由于版本不匹配而导致的冲突。Automerge 将遵循其属性并拒绝名称更改，因为它是一个标量并添加到非冲突字段。这将导致保存在服务器中的项目显示如下。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner"] # underlying data type is a Set
  "points": [24, 30, 27] # underlying data type is a List
  "_version" : 6
}
```

使用具有版本 6 的项目的更新图像，现在假设传入的变更（版本不匹配）尝试将项目转换为以下内容：

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "brunch"] # underlying data type is a Set
  "points": [30, 35] # underlying data type is a List
  "_version" : 5
}
```

在这里，我们观察到“兴趣”的传入字段有一个存在于服务器中的重复值和两个新值。在这种情况下，由于底层数据类型是集合，Automerge 会将服务器中现有的值与传入请求中的值合并，并剔除任何重复项。同样，“分数”字段上也存在冲突，其中有一个重复值和一个新值。但是，由于这里的底层数据类型是列表，Automerge 会简单地将传入请求中的所有值附加到服务器中已存在的值的末尾。存储在服务器上的结果合并图像如下所示：

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a
 Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "_version" : 7
}
```

现在，让我们假设存储在服务器中的项目（版本 8）显示如下。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a
 Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "stats": {
      "ppg": "35.4",
      "apg": "6.3"
  }
  "_version" : 8
```

```
}
```

但是传入的请求尝试使用以下图像更新该项目，从而再次出现版本不匹配的情况：

```
{
  "id" : 1,
  "name" : "Nadia",
  "stats": {
      "ppg": "25.7",
      "rpg": "6.9"
  }
  "_version" : 3
}
```

现在，在这种情况下，我们可以看到服务器中已经存在的字段丢失（兴趣、分数、球衣）。此外，正在编辑映射“统计”中的“ppg”值，添加了一个新值“rpg”，并省略了“apg”。Automerge 保留已省略的字段（注意：如果打算删除字段，则必须使用匹配版本再次尝试请求），以便它们不会丢失。它也会将相同的规则应用于映射中的字段，因此对“ppg”的更改将被拒绝，而“apg”被保留，并添加一个新字段“rpg”。存储在服务器中的结果项目现在将显示为：

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "interests" : ["breakfast", "lunch", "dinner", "brunch"] # underlying data type is a
 Set
  "points": [24, 30, 27, 30, 35] # underlying data type is a List
  "stats": {
      "ppg": "35.4",
      "apg": "6.3",
      "rpg": "6.9"
  }
  "_version" : 9
}
```

Lambda

冲突解决选项：

- RESOLVE：将现有项替换为响应有效负载中提供的新项。您一次只能对单个项目重试同一操作。当前受 DynamoDB `PutItem` 和 `UpdateItem` 支持。

- REJECT：拒绝变更并返回错误以及 GraphQL 响应中的现有项目。当前受 DynamoDB PutItem、UpdateItem 和 DeleteItem 支持。
- REMOVE：删除现有项目。当前受 DynamoDB DeleteItem 支持。

Lambda 调用请求

AWS AppSync DynamoDB 解析器调用 LambdaConflictHandlerArn 中指定的 Lambda 函数。它将使用在数据源上配置的相同 service-role-arn。调用的负载具有以下结构：

```
{
    "newItem": { ... },
    "existingItem": {... },
    "arguments": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

字段定义如下：

newItem

预览项目（如果变更成功）。

existingItem

该项目当前位于 DynamoDB 表中。

arguments

来自 GraphQL 变更的参数。

resolver

有关 AWS AppSync 解析器的信息。

identity

有关调用方的信息。如果使用 API 密钥进行访问，则该字段设置为 null。

有效负载示例：

```
{
```

```
    "newItem": {
        "id": "1",
        "author": "Jeff",
        "title": "Foo Bar",
        "rating": 5,
        "comments": ["hello world"],
    },
    "existingItem": {
        "id": "1",
        "author": "Foo",
        "rating": 5,
        "comments": ["old comment"]
    },
    "arguments": {
        "id": "1",
        "author": "Jeff",
        "title": "Foo Bar",
        "comments": ["hello world"]
    },
    "resolver": {
        "tableName": "post-table",
        "awsRegion": "us-west-2",
        "parentType": "Mutation",
        "field": "updatePost"
    },
    "identity": {
         "accountId": "123456789012",
         "sourceIp": "x.x.x.x",
         "username": "AIDAAAAAAAAAAAAAAAAA",
         "userArn": "arn:aws:iam::123456789012:user/appsync"
    }
}
```

Lambda 调用响应

对于 PutItem 和 UpdateItem 解决冲突

RESOLVE 变更。响应必须采用以下格式。

```
{
    "action": "RESOLVE",
    "item": { ... }
}
```

item 字段表示将用于替换底层数据源中的现有项目的对象。如果在 item 中包含主键和同步元数据，则将被忽略。

REJECT 变更。响应必须采用以下格式。

```
{
    "action": "REJECT"
}
```

对于 DeleteItem 冲突解决

REMOVE 项目。响应必须采用以下格式。

```
{
    "action": "REMOVE"
}
```

REJECT 变更。响应必须采用以下格式。

```
{
    "action": "REJECT"
}
```

下面的示例 Lambda 函数检查谁进行调用以及解析器名称。如果由 jeffTheAdmin 调用，则 REMOVE DeletePost 解析器的对象，或 RESOLVE 与更新/放置解析器的新项目冲突。如果不是，则变更是 REJECT。

```
exports.handler = async (event, context, callback) => {
    console.log("Event: "+ JSON.stringify(event));

    // Business logic goes here.
    var response;
    if ( event.identity.user == "jeffTheAdmin" ) {
        let resolver = event.resolver.field;

        switch(resolver) {
            case "deletePost":
                response = {
                    "action" : "REMOVE"
                }
                break;
```

```
            case "updatePost":
            case "createPost":
                response = {
                    "action" : "RESOLVE",
                    "item": event.newItem
                }
                break;
            default:
                response = { "action" : "REJECT" };
        }
    } else {
        response = { "action" : "REJECT" };
    }

    console.log("Response: "+ JSON.stringify(response));
    return response;
}
```

错误

ConflictUnhandled

冲突检测发现版本不匹配情况，并且冲突处理程序拒绝变更。

示例：使用乐观并发冲突处理程序解决冲突。或者，Lambda 冲突处理程序返回 REJECT。

ConflictError

尝试解决冲突时发生内部错误。

示例：Lambda 冲突处理程序返回格式错误的响应。或者，无法调用 Lambda 冲突处理程序，因为找不到提供的资源 LambdaConflictHandlerArn。

MaxConflicts

已达到解决冲突的最大重试次数。

示例：同一对象上的并发请求太多。在解决冲突之前，另一个客户端将对象更新为新版本。

BadRequest

客户端尝试更新元数据字段（_version、_ttl、_lastChangedAt、_deleted）。

示例：客户端尝试使用更新变更来更新对象的版本。

`DeltaSyncWriteError`

写入增量同步记录失败。

示例：变更成功，但尝试写入增量同步表时出现内部错误。

`InternalFailure`

出现内部错误。

CloudWatch Logs

如果 AWS AppSync API 已启用 CloudWatch Logs，将日志记录设置设为字段级日志 `enabled`，并将字段级日志的日志级别设置为 `ALL`，AWS AppSync 将向日志组发出冲突检测和解决信息。有关日志消息格式的信息，请参阅冲突检测和同步日志记录的文档。

同步操作

版本控制的数据源支持 `Sync` 操作，以允许您从 DynamoDB 表中检索所有结果，然后仅接收自上次查询以来更改的数据（增量更新）。在 AWS AppSync 收到 `Sync` 操作请求时，它使用请求中指定的字段确定是否应访问基 表或增量 表。

- 如果未指定 `lastSync` 字段，则对基 表执行 `Scan`。
- 如果指定了 `lastSync` 字段，但该值在 `current moment - DeltaSyncTTL` 之前，则对基 表执行 `Scan`。
- 如果指定了 `lastSync` 字段，并且该值在 `current moment - DeltaSyncTTL` 或之后，则对增量 表执行 `Query`。

AWS AppSync 将 `startedAt` 字段返回到所有 `Sync` 操作的响应映射模板。`startedAt` 字段是 `Sync` 操作开始的时刻，以纪元毫秒为单位，您可以在本地存储并在其他请求中使用该值。如果请求中包含分页令牌，则该值将与请求针对第一页结果返回的值相同。

有关 `Sync` 映射模板格式的信息，请参阅映射模板参考。

监控和日志记录

要监控您的 AWS AppSync GraphQL API 并帮助调试与请求相关的问题，您可以开启对 Ama CloudWatch zon Logs 的登录功能。

设置和配置

要在 GraphQL API 上开启自动日志功能，请使用控制台。 AWS AppSync

1. 登录 AWS Management Console 并打开AppSync控制台。
2. 在 API 页面上，选择一个 GraphQL API 的名称。
3. 在您的 API 主页的导航窗格中，选择设置。
4. 在日志记录下面，执行以下操作：
   a. 开启启用日志记录。
   b. 要获取详细的请求级日志记录，请选中包含详细内容下面的复选框（可选）。
   c. 在字段解析器日志级别下面，选择所需的字段级日志记录级别（无、错误或全部）（可选）。
   d. 在 “创建或使用现有角色” 下，选择 “新建角色” 以创建允许 AWS AppSync 向其写入日志的新 AWS Identity and Access Management (IAM) CloudWatch。或者，选择现有角色以选择您的 AWS 账户中的现有 IAM 角色的 Amazon 资源名称 (ARN)。
5. 选择保存。

手动配置 IAM 角色

如果您选择使用现有 IAM 角色，则该角色必须授予 AWS AppSync 写入日志所需的权限 CloudWatch。要手动配置，您必须提供服务角色 ARN，以便在写入日志时 AWS AppSync 可以代入该角色。

在 IAM 控制台中，创建一个名为 `AWSAppSyncPushToCloudWatchLogsPolicy` 的新策略，该策略具有以下定义：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
```

```
    ]
}
```

接下来，使用名称创建一个新角色 AWSAppSyncPushToCloudWatchLogsRole，并将新创建的策略附加到该角色。编辑该角色的信任关系以包含以下内容：

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

复制角色 ARN，并在为 GraphQL AP AWS AppSync I 设置日志时使用它。

CloudWatch 指标

您可以使用 CloudWatch 指标来监控可能导致 HTTP 状态代码或延迟的特定事件并提供警报。将发出以下指标：

指标列表

4XXError

由于客户端配置不正确导致请求无效而产生的错误。通常，这些错误是在 GraphQL 处理以外的任何地方发生的。例如，如果请求包含不正确的 JSON 负载或不正确的查询，服务受到限制或未正确配置授权设置，则可能会出现这些错误。

单位：计数。使用 Sum 统计数据以得出这些错误的总出现次数。

5XXError

在运行 GraphQL 查询期间遇到的错误。例如，在为空架构或不正确的架构调用查询时，可能会出现该错误。当 Amazon Cognito 用户池 ID 或 AWS 区域无效时，也可能发生这种情况。或者，如果在处理请求时 AWS AppSync 遇到问题，也可能发生这种情况。

单位：计数。使用 Sum 统计数据以得出这些错误的总出现次数。

Latency

从 AWS AppSync 收到来自客户端的请求到它向客户端返回响应之间的时间。这不包括响应进入终端设备所遇到的网络延迟。

单位：毫秒。使用平均值统计数据可评估预期延迟。

Requests

您的账户中的所有 API 已处理的请求（查询 + 变量）数量（按区域划分）。

单位：计数。特定区域中处理的所有请求的数量。

TokensConsumed

根据 Request 使用的资源量（处理时间和使用的内存），为 Requests 分配令牌。通常，每个 Request 使用一个令牌。不过，根据需要，将为使用大量资源的 Request 分配额外的令牌。

单位：计数。为特定区域中处理的请求分配的令牌数量。

NetworkBandwidthOutAllowanceExceeded

> **Note**
>
> 在 AWS AppSync 控制台的缓存设置页面上，Cache Healt h Metrics 选项允许您启用此与缓存相关的运行状况指标。

由于吞吐量超过聚合带宽限制，网络数据包被丢弃。这对于诊断缓存配置中的瓶颈很有用。通过在appsyncCacheNetworkBandwidthOutAllowanceExceeded指标API_Id中指定来记录特定 API 的数据。

单位：计数。超过 ID 指定的 API 的带宽限制后丢弃的数据包数。

EngineCPUUtilization

> **Note**
>
> 在 AWS AppSync 控制台的缓存设置页面上，Cache Healt h Metrics 选项允许您启用此与缓存相关的运行状况指标。

分配给 Redis 进程的 CPU 使用率（百分比）。这对于诊断缓存配置中的瓶颈很有用。通过在appsyncCacheEngineCPUUtilization指标API_Id中指定来记录特定 API 的数据。

单位：百分比。由 ID 指定的 API 的 Redis 进程当前使用的 CPU 百分比。

实时订阅

所有指标都在一个维度中发出：GraphQLAPIId。这意味着所有指标都与 GraphQL API ID 相结合。以下指标与纯粹的 GraphQL 订阅有关： WebSockets

指标列表

ConnectRequests

向发出的 WebSocket 连接请求数 AWS AppSync，包括成功和失败的尝试。

单位：计数。使用总和统计数据可获取连接请求总数。

ConnectSuccess

成功 WebSocket 连接的次数 AWS AppSync。可以在没有订阅的情况下进行连接。

单位：计数。使用 Sum 统计数据可得出成功连接的总出现次数。

ConnectClientError

AWS AppSync 由于客户端错误而被拒绝的 WebSocket 连接数。这可能意味着，服务受到限制或未正确配置授权设置。

单位：计数。使用 Sum 统计数据以得出客户端连接错误的总出现次数。

ConnectServerError

处理连接 AWS AppSync 时产生的错误数。当出现意外的服务器端问题时，通常会发生这种错误。

单位：计数。使用 Sum 统计数据以得出服务器端连接错误的总出现次数。

DisconnectSuccess

成功 WebSocket 断开连接的次数。 AWS AppSync

单位：计数。使用 Sum 统计数据可得出成功断开连接的总出现次数。

DisconnectClientError

断开 WebSocket连接 AWS AppSync 时产生的客户端错误数。

单位：计数。使用 Sum 统计数据可得出断开连接错误的总出现次数。

DisconnectServerError

断开 WebSocket连接 AWS AppSync 时产生的服务器错误数。

单位：计数。使用 Sum 统计数据可得出断开连接错误的总出现次数。

SubscribeSuccess

AWS AppSync 通过成功注册的订阅数量 WebSocket。可以在没有订阅的情况下建立连接，但无法在没有连接的情况下进行订阅。

单位：计数。使用 Sum 统计数据以得出成功订阅的总发生次数。

SubscribeClientError

AWS AppSync 由于客户端错误而被拒绝的订阅数量。如果 JSON 负载不正确，服务受到限制或未正确配置授权设置，则可能会出现该错误。

单位：计数。使用 Sum 统计数据以得出客户端订阅错误的总出现次数。

SubscribeServerError

处理订阅 AWS AppSync 时产生的错误数。当出现意外的服务器端问题时，通常会发生这种错误。

单位：计数。使用 Sum 统计数据以得出服务器端订阅错误的总出现次数。

UnsubscribeSuccess

已成功处理的取消订阅请求数。

单位：计数。可以使用 Sum 统计数据获取成功取消订阅请求的总发生次数。

UnsubscribeClientError

AWS AppSync 由于客户端错误而被拒绝的取消订阅请求的数量。

单位：计数。可以使用 Sum 统计数据获取客户端取消订阅请求错误的总出现次数。

UnsubscribeServerError

处理取消订阅请求 AWS AppSync 时产生的错误数。当出现意外的服务器端问题时，通常会发生这种错误。

单位：计数。可以使用 Sum 统计数据获取服务器端取消订阅请求错误的总出现次数。

PublishDataMessageSuccess

已成功发布的订阅事件消息的数量。

单位：计数。使用 Sum 统计数据以得出成功发布的订阅事件消息的总数。

PublishDataMessageClientError

由于客户端错误而无法发布的订阅事件消息的数量。

Unit：计数。使用 Sum 统计数据以得出客户端发布订阅事件错误的总出现次数。

PublishDataMessageServerError

发布订阅事件消息 AWS AppSync 时产生的错误数。当出现意外的服务器端问题时，通常会发生这种错误。

单位：计数。使用 Sum 统计数据以得出服务器端发布订阅事件错误的总出现次数。

PublishDataMessageSize

已发布的订阅事件消息的大小。

单位：字节。

ActiveConnections

1 分钟内从客户端到 AWS AppSync 的并发 WebSocket 连接数。

单位：计数。使用 Sum 统计数据以得出建立的连接总数。

ActiveSubscriptions

1 分钟内来自客户端的并发订阅数。

单位：计数。使用 Sum 统计数据以得出有效订阅的总数。

ConnectionDuration

连接保持打开状态的时间量。

单位：毫秒。使用 Average 统计数据可评估连接持续时间。

OutboundMessages

成功发布的按流量计费消息的数量。一条按流量计费的消息等于 5 KB 的传送数据。

单位：计数。使用总和统计数据可获取成功发布的计量消息数。

InboundMessageSuccess

成功处理的入站消息数量。突变调用的每种订阅类型都会生成一条入站消息。

单位：计数。使用总和统计数据可获取成功处理的入站消息总数。

InboundMessageError

由于 API 请求无效（例如超过 240 kB 的订阅有效负载大小限制）而导致无法处理的入站消息的数量。

单位：计数。使用总和统计数据可获取与 API 相关的处理失败的入站消息总数。

InboundMessageFailure

由于来自的错误而导致无法处理的入站消息的数量 AWS AppSync。

单位：计数。使用 Sum 统计数据获取出现 AWS AppSync相关处理失败的入站邮件的总数。

InboundMessageDelayed

延迟入站消息的数量。当超过入站邮件速率配额或出站邮件速率配额时，入站邮件可能会延迟。

单位：计数。使用 Sum 统计数据获取延迟的入站消息总数。

InboundMessageDropped

丢弃的入站消息的数量。当超过入站邮件速率配额或出站邮件速率配额时，入站邮件可能会被丢弃。

单位：计数。使用 Sum 统计数据获取丢弃的入站邮件总数。

InvalidationSuccess

变更通过 `$extensions.invalidateSubscriptions()` 成功使订阅失效（取消订阅）的次数。

单位：计数。可以使用 Sum 统计数据检索成功取消订阅的总订阅数。

InvalidationRequestSuccess

已成功处理的失效请求数。

单位：计数。使用总和统计数据可获取成功处理的失效请求总数。

InvalidationRequestError

由于 API 请求无效而导致处理失败的失效请求的数量。

单位：计数。使用总和统计数据可获取与 API 相关的处理失败的失效请求总数。

InvalidationRequestFailure

由于来自 AWS AppSync的错误而导致处理失败的失效请求的数量。

单位：计数。使用 Sum 统计数据来获取具有 AWS AppSync相关处理失败的失效请求的总数。

InvalidationRequestDropped

当超过失效请求限额时丢弃的失效请求的数量。

单位：计数。使用总和统计数据可获取丢弃的失效请求的总数。

比较入站消息和出站消息

执行突变时，将调用带有该突变的 @aws_subscribe 指令的订阅字段。每次订阅调用都会生成一条入站消息。例如，如果两个订阅字段在 @aws_subscribe 中指定了相同的突变，则在调用该变异时会生成两条入站消息。

一封出站消息等于向 WebSocket 客户端传送的 5 KB 数据。例如，向 10 个客户机发送 15 kB 的数据会产生 30 条出站消息（15 kB * 10 个客户机/每条消息 5 kB = 30 条消息）。

您可以申请增加入站或出站邮件的配额。有关更多信息，请参阅《AWS 通用参考指南》中的AWS AppSync 终端节点和配额以及《S ervice Qu otas 用户指南》中有关请求增加配额的说明。

增强的指标

增强的指标会生成有关 API 使用情况和性能的精细数据，例如 AWS AppSync 请求和错误计数、延迟以及缓存命中/未命中。所有增强型指标数据都将发送到您的 CloudWatch 账户，您可以配置要发送的数据类型。

> **Note**
>
> 使用增强型指标时会收取额外费用。有关更多信息，请参阅 Amazon 定价中的详细监控 CloudWatch定价等级。

这些指标可以在 AWS AppSync 控制台的各个设置页面上找到。在 API 设置页面上，增强指标部分允许您启用或禁用以下项目：

1. 解析器指标行为：这些选项控制如何收集解析者的其他指标。您可以选择启用完整的请求解析器指标（为请求中的所有解析器启用的指标）或每个解析器指标（仅在配置设置为启用的解析器中启用指标）。以下选项可用：

指标	指标维度	指标名称	单位	描述

每个解析器的 GraphQL 错误数	api_id，解析器	GraphQLErro	计数	每个解析器发生的 GraphQL 错误数。
每个解析者的请求数	api_id，解析器	请求	计数	请求期间发生的调用次数。这是按每个解析者记录的。
每个解析器的延迟	api_id，解析器	延迟	毫秒	完成解析器调用的时间。延迟以毫秒为单位进行测量，并以每个解析器为单位进行记录。
每个解析器的缓存命中数	api_id，解析器	CacheHit	计数	请求期间的缓存命中次数。只有在使用缓存时才会发出此消息。缓存命中率是按每个解析器记录的。
每个解析器的缓存未命中次数	api_id，解析器	CacheMiss	计数	请求期间的缓存丢失次数。只有在使用缓存时才会发出此消息。缓存失误是按每个解析器记录的。

2. 数据源指标行为：这些选项控制如何收集数据源的其他指标。您可以选择启用完整请求数据源指标（为请求中的所有数据源启用的指标）或每个数据源的指标（仅对配置设置为启用的数据源启用指标）。以下选项可用：

指标	指标维度	指标名称	单位	描述

每个数据源的请求数	api_id，数据源	请求	计数	请求期间发生的调用次数。请求是按数据源记录的。如果启用了完整请求，则每个数据源都将在中拥有自己的条目 CloudWatch。
每个数据源的延迟	api_id，数据源	延迟	毫秒	完成数据源调用的时间。延迟是按每个数据源记录的。
每个数据源的错误数	api_id，数据源	GraphQLErro	计数	调用数据源期间发生的错误数。

3. 操作指标：启用 GraphQL 操作级指标。

指标	指标维度	指标名称	单位	描述
每次操作的请求数	api_id，操作	请求	计数	指定 GraphQL 操作被调用的次数。
每次操作的 GraphQL 错误数	api_id，操作	GraphQLErro	计数	在指定的 GraphQL 操作期间发生的 GraphQL 错误数。

CloudWatch 日志

您可以对任何新的或现有的 GraphQL API 配置两种类型的日志记录：请求级别和字段级别。

请求级日志

如果配置了请求级日志记录（包含详细内容），将记录以下信息：

- 使用的令牌数
- 请求和响应 HTTP 标头
- 请求中运行的 GraphQL 查询
- 总体操作摘要
- 已注册的新的和现有的 GraphQL 订阅

字段级日志

如果配置了字段级日志记录，将记录以下信息：

- 生成的请求映射，其中包含每个字段的源和参数
- 每个字段的转换响应映射，其中包括作为该字段解析结果的数据
- 每个字段的跟踪信息

如果您开启日志记录，则 AWS AppSync 管理日 CloudWatch 志。该过程包括创建日志组和日志流，以及向日志流报告这些日志。

当您打开 GraphQL API 的日志记录并发出请求时， AWS AppSync 将在该日志组下创建一个日志组和日志流。日志组以 `/aws/appsync/apis/{graphql_api_id}` 格式命名。在每个日志组内，日志会进一步分成多个日志流。当报告已记录的数据时，这些日志按上次事件时间排序。

每个日志事件都标有该请求的 x-amzn-RequestId。这可以帮助您筛选日志事件 CloudWatch ，以获取有关该请求的所有已记录信息。你可以 RequestId 从每个 GraphQL AWS AppSync 请求的响应标头中获取。

字段级别日志记录配置为使用以下日志级别：

- 无 - 不捕获任何字段级日志。
- 错误 - 仅记录出现错误的字段的以下信息：
    - 服务器响应中的错误部分
    - 字段级别错误
    - 所生成的请求/响应函数，已针对错误字段获得解析

- 全部 - 记录查询中的所有字段的以下信息：
    - 字段级别跟踪信息
    - 所生成的请求/响应函数，已针对每个字段获得解析

监控优势

您可以使用日志记录和指标来识别、优化 GraphQL 查询和排除其问题。例如，这些将帮助您使用针对查询中的每个字段记录的跟踪信息调试延迟问题。为了对此进行演示，假设您正在使用一个或多个嵌套在 GraphQL 查询中的解析器。L CloudWatch ogs 中的示例字段操作可能与以下内容类似：

```
{
    "path": [
        "singlePost",
        "authors",
        0,
        "name"
    ],
    "parentType": "Post",
    "returnType": "String!",
    "fieldName": "name",
    "startOffset": 416563350,
    "duration": 11247
}
```

这可能对应于 GraphQL 架构，类似于以下内容：

```
type Post {
  id: ID!
  name: String!
  authors: [Author]
}

type Author {
  id: ID!
  name: String!
}

type Query {
  singlePost(id:ID!): Post
}
```

在前面的日志结果中，path 显示运行名为 `singlePost()` 的查询而返回的数据中的单个项目。在该示例中，它表示第一个索引 (0) 处的 name 字段。startOffset 提供相对于 GraphQL 查询操作开始位置的偏移。duration 是解析该字段的总时间。这些值可用来排除下面这类问题：为何来自特定数据来源的数据的运行速度可能低于预期，或者特定字段是否降低了整个查询的速度等。例如，您可以选择增加 Amazon DynamoDB 表的预置吞吐量，或从查询中删除导致整体操作性能不佳的特定字段。

自 2019 年 5 月 8 日起， AWS AppSync 将日志事件生成为完全结构化的 JSON。这可以帮助您使用 Logs Insights 和 Amazon OpenSearch 服务等 CloudWatch 日志分析服务来了解 GraphQL 请求的性能和架构字段的使用特征。例如，您可以轻松识别具有较大延迟的解析器，这可能是导致性能问题的根本原因。您还可以识别架构中最常用和最不常用的字段，并评估弃用 GraphQL 字段的影响。

冲突检测和同步日志记录

如果 AWS AppSync API 的 CloudWatch 日志记录配置为字段解析器日志级别设置为 “全部”，则会向日志 AWS AppSync 组发送冲突检测和解决信息。这提供了对 AWS AppSync API 如何应对冲突的详细见解。为了帮助您解释响应，在日志中提供了以下信息：

指标列表

`conflictType`

详细说明是否由于版本不匹配或由于客户提供的状况而发生冲突。

`conflictHandlerConfigured`

说明请求时在解析器上配置了冲突处理程序。

`message`

提供有关如何检测和解决冲突的信息。

`syncAttempt`

服务器在最终拒绝请求之前尝试同步数据的次数。

`data`

如果配置的冲突处理程序为 `Automerge`，则填充该字段以显示 `Automerge` 为每个字段做出的决策。提供的操作可以是：

- REJECTED - `Automerge` 拒绝传入的字段值，而采用服务器中的值。
- ADDED - 由于服务器中没有预先存在的值，`Automerge` 添加传入的字段。
- APPENDED - `Automerge` 将传入的值附加到服务器中存在的列表值。
- MERGED - `Automerge` 将传入的值合并到服务器中存在的集合值。

使用令牌计数优化您的请求

对于使用少于或等于 1,500 KB 秒内存和 vCPU 时间的请求，将分配一个令牌。对于使用超过 1,500 KB 秒资源的请求，将收到额外的令牌。例如，如果请求消耗 3,350 KB 秒，则向该请求 AWS AppSync 分配三个令牌（向上舍入到下一个整数值）。默认情况下，每秒最多向账户中的 API AWS AppSync 分配 5,000 或 10,000 个请求令牌，具体取决于其部署所在的 AWS 区域。如果您的 API 平均每秒使用两个令牌，则每秒将分别限制为 2,500 或 5,000 个请求。如果您每秒需要的令牌数多于分配的数量，您可以提交请求以增加请求令牌速率的默认配额。有关更多信息，请参阅AWS 一般参考 指南中的AWS AppSync 终端节点和配额以及 Servic e Quotas 用户指南中的请求增加配额。

如果每个请求的令牌数较高，则可能表明需要优化您的请求并提高 API 性能。可能增加每个请求的令牌数的因素包括：

- GraphQL 架构的大小和复杂性。
- 请求映射模板和响应映射模板的复杂性。
- 每个请求的解析器调用次数。
- 从解析器返回的数据量。
- 下游数据来源的延迟。
- 需要连续调用数据来源（而不是并行或批量调用）的架构和查询设计。
- 日志记录配置，特别是字段级和详细日志内容。

> **Note**
>
> 除了 AWS AppSync 指标和日志外，客户端还可以通过响应标头访问请求中消耗的令牌数量`x-amzn-appsync-TokensConsumed`。

日志类型参考

RequestSummary

- requestId：请求的唯一标识符。
- graphQLAPIId：发出请求的 GraphQL API 的 ID。
- statusCode：HTTP 状态代码响应。
- 延迟：E 请求的nd-to-end 延迟，以纳秒为单位，以整数表示。

```
{
     "logType": "RequestSummary",
     "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
     "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4",
     "statusCode": 200,
     "latency": 242000000
}
```

ExecutionSummary

- requestId：请求的唯一标识符。
- graphQLAPIId：发出请求的 GraphQL API 的 ID。
- startTime：GraphQL 处理请求的开始时间戳，采用 RFC 3339 格式。
- endTime：GraphQL 处理请求的结束时间戳，采用 RFC 3339 格式。
- duration：GraphQL 总处理时间，以纳秒为单位，整数值。
- 版本：的架构版本 ExecutionSummary。
- parsing：
    - startOffset：解析的起始偏移，以纳秒为单位，相对于调用，整数值。
    - duration：解析所花费的时间，以纳秒为单位，整数值。
- validation：
    - startOffset：验证的起始偏移，以纳秒为单位，相对于调用，整数值。
    - duration：执行验证所花费的时间，以纳秒为单位，整数值。

```
{
    "duration": 217406145,
    "logType": "ExecutionSummary",
    "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
    "startTime": "2019-01-01T06:06:18.956Z",
    "endTime": "2019-01-01T06:06:19.174Z",
    "parsing": {
        "startOffset": 49033,
        "duration": 34784
    },
    "version": 1,
    "validation": {
        "startOffset": 129048,
        "duration": 69126
```

```
    },
    "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4"
}
```

跟踪

- requestId：请求的唯一标识符。
- graphQLAPIId：发出请求的 GraphQL API 的 ID。
- startOffset：字段解析的起始偏移，以纳秒为单位，相对于调用，整数值。
- duration：解析字段所花费的时间，以纳秒为单位，整数值。
- fieldName：所解析字段的名称。
- parentType：所解析字段的父类型。
- returnType：所解析字段的返回类型。
- path：路径分段列表，从响应的根开始，以所解析字段结束。
- resolverArn：用于字段解析的解析器的 ARN。可能不会出现在嵌套字段中。

```
{
    "duration": 216820346,
    "logType": "Tracing",
    "path": [
        "putItem"
    ],
    "fieldName": "putItem",
    "startOffset": 178156,
    "resolverArn": "arn:aws:appsync:us-east-1:111111111111:apis/
pmo28inf75eepg63qxq4ekoeg4/types/Mutation/fields/putItem",
    "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
    "parentType": "Mutation",
    "returnType": "Item",
    "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4"
}
```

使用 “日志见解” 分析您的 CloudWatch 日志

以下是您可以运行的查询示例，以获取有关 GraphQL 操作的性能和运行状况的可行的见解。这些示例作为示例查询在 Logs Insight CloudWatch s 控制台中提供。在CloudWatch控制台中，选择 Lo gs Insights，为 GraphQL API 选择 AWS AppSync 日志组，然后在示例AWS AppSync 查询下选择查询。

以下查询返回使用令牌数最多的前 10 个 GraphQL 请求：

```
filter @message like "Tokens Consumed"
| parse @message "* Tokens Consumed: *" as requestId, tokens
| sort tokens desc
| display requestId, tokens
| limit 10
```

以下查询返回具有最大延迟的前 10 个解析器：

```
  fields resolverArn, duration
| filter logType = "Tracing"
| limit 10
| sort duration desc
```

以下查询返回最常调用的解析器：

```
  fields ispresent(resolverArn) as isRes
| stats count() as invocationCount by resolverArn
| filter isRes and logType = "Tracing"
| limit 10
| sort invocationCount desc
```

以下查询返回映射模板中具有最多错误的解析器：

```
  fields ispresent(resolverArn) as isRes
| stats count() as errorCount by resolverArn, logType
| filter isRes and (logType = "RequestMapping" or logType = "ResponseMapping") and
 fieldInError
| limit 10
| sort errorCount desc
```

以下查询返回解析器延迟统计数据：

```
  fields ispresent(resolverArn) as isRes
| stats min(duration), max(duration), avg(duration) as avg_dur by resolverArn
| filter isRes and logType = "Tracing"
| limit 10
| sort avg_dur desc
```

以下查询返回字段延迟统计数据：

```
  stats min(duration), max(duration), avg(duration) as avg_dur
  by concat(parentType, '/', fieldName) as fieldKey
| filter logType = "Tracing"
| limit 10
| sort avg_dur desc
```

可以将 CloudWatch Logs Insights 查询的结果导出到 CloudWatch 仪表板。

使用 OpenSearch 服务分析您的日志

您可以使用 Amazon S OpenSearch ervice 搜索、分析和可视化您的 AWS AppSync 日志，以确定性能瓶颈和操作问题的根本原因。您可以识别具有最大延迟和最多错误的解析器。此外，您还可以使用 OpenSearch 仪表板创建具有强大可视化效果的仪表板。 OpenSearch 仪表板是 S OpenSearch ervice 中提供的开源数据可视化和探索工具。使用 OpenSearch 控制面板，您可以持续监控 GraphQL 操作的性能和运行状况。例如，您可以创建控制面板以可视化 GraphQL 请求的 P90 延迟，并深入了解每个解析器的 P90 延迟。

使用 S OpenSearch ervice 时，使用 “cwl*” 作为筛选器模式来搜索索 OpenSearch 引。 OpenSearch 服务使用前缀为 “cwl-” 对从 CloudWatch 日志流式传输的日志进行索引。为了将 AWS AppSync API 日志与发送到 OpenSearch 服务的其他 CloudWatch 日志区分开来，我们建议在搜索中`graphQLAPIID.keyword=`*`YourGraphQLAPIID`*添加额外的筛选表达式。

日志格式迁移

2019 年 5 月 8 日当天或之后 AWS AppSync 生成的日志事件采用完全结构化的 JSON 格式。要分析 2019 年 5 月 8 日之前的 GraphQL 请求，您可以使用示例中提供的脚本将较旧的日志迁移到完全结构化的 JSON。GitHub 如果需要在 2019 年 5 月 8 日之前使用日志格式，请使用以下设置创建支持服务单：将 Type (类型) 设置为 Account Management (账户管理)，然后将 Category (类别) 设置为 General Account Question (一般账户问题)。

您还可以使用指标筛选器将日志数据转换为数字 CloudWatch 指标，以便您可以对它们绘制图表或设置警报。 CloudWatch

使用 AWS X-Ray 进行跟踪

您可以使用 AWS X-Ray 跟踪在 AWS AppSync 中执行的请求。在提供 X-Ray 的所有 AWS 区域中，您可以将 X-Ray 与 AWS AppSync 一起使用。X-Ray 为您提供整个 GraphQL 请求的详细概述。这使您能够分析 API 及其基础解析器和数据源中的延迟。您可以使用 X-Ray 服务地图查看请求延迟，包括

与 X-Ray 集成的任何 AWS 服务。您也可以配置采样规则，以根据您指定的标准指示 X-Ray 记录哪些请求以及以哪种采样率进行记录。

有关 X-Ray 中的采样的更多信息，请参阅 Configuring Sampling Rules in the AWS X-Ray Console。

设置和配置

您可以通过 AWS AppSync 控制台为 GraphQL API 启用 X-Ray 跟踪。

1. 登录到 AWS AppSync 控制台。
2. 从导航面板中选择 Settings (设置)。
3. 在 X-Ray 下，开启 Enable X-Ray (启用 X-Ray)。
4. 选择保存。您的 API 现已启用 X-Ray 跟踪。

如果使用 AWS CLI 或 AWS CloudFormation，您也可以在创建新的 AWS AppSync API 或更新现有的 AWS AppSync API 时启用 X-Ray 跟踪，方法是将 `xrayEnabled` 属性设置为 `true`。

如果为 AWS AppSync API 启用了 X-Ray 跟踪，则会在您的账户中自动创建一个具有相应权限的 AWS Identity and Access Management 服务相关角色。这允许 AWS AppSync 以安全方式将跟踪发送到 X-Ray。

使用 X-Ray 跟踪您的 API

采样

通过使用采样规则，您可以控制在 AWS AppSync 中记录的数据量，并且可以动态修改采样行为，而无需修改或重新部署代码。例如，此规则对 API ID 为 `3n572shhcpfokwhdnq1ogu59v6` 的 GraphQL API 的请求进行采样。

- 规则名称 - `test-sample`
- 优先级 - `10`
- 容器大小 - `10`
- 固定速率 - `10`
- 服务名称 - `*`
- 服务类型 - `AWS::AppSync::GraphQLAPI`
- HTTP 方法 - `*`

- 资源 ARN - `arn:aws:appsync:us-west-2:123456789012:apis/3n572shhcpfokwhdnq1ogu59v6`
- 主机 - *

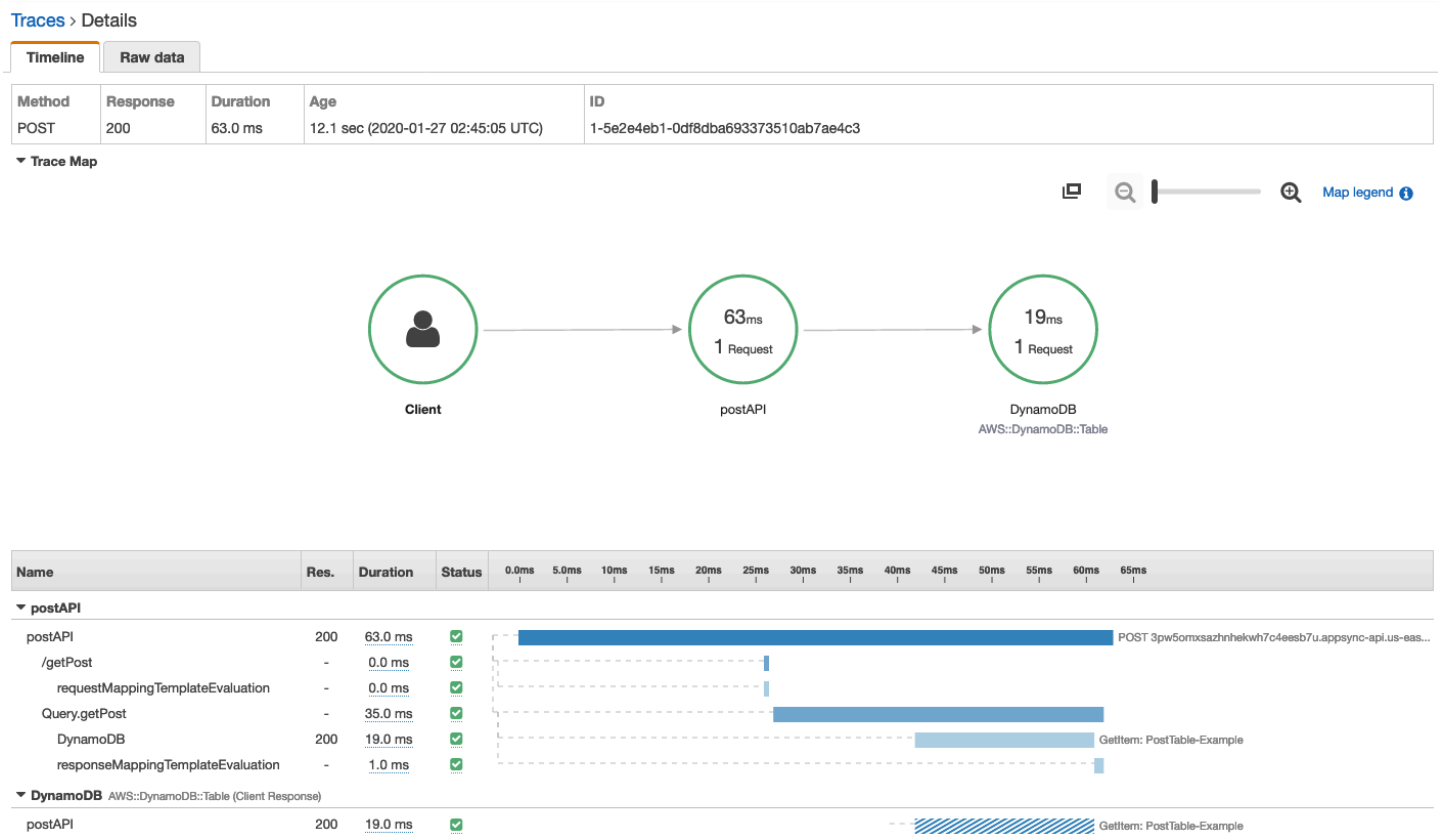
了解跟踪

在为 GraphQL API 启用 X-Ray 跟踪时，您可以使用 X-Ray 跟踪详细信息页面检查有关向您的 API 发出的请求的详细延迟信息。以下示例显示了此特定请求的跟踪视图以及服务地图。该请求是对名为 `postAPI` 且具有 Post 类型的 API 发出的，其数据包含在名为 `PostTable-Example` 的 Amazon DynamoDB 表中。

下面的跟踪映像对应于以下 GraphQL 查询：

```
query getPost {
    getPost(id: "1") {
      id
      title
    }
}
```

`getPost` 查询的解析器使用底层 DynamoDB 数据源。以下跟踪视图显示对 DynamoDB 的调用，以及查询执行的各个部分的延迟：



- 在上图中，/getPost 表示指向正在解析的元素的完整路径。在这种情况下，由于 getPost 是根 Query 类型上的字段，因此它直接显示在路径的根目录之后。
- requestMappingTemplateEvaluation 表示 AWS AppSync 在查询中评估该元素的请求映射模板所花的时间。
- Query.getPost 表示类型和字段（采用 Type.field 格式）。它可以包含多个子段，具体取决于 API 的结构和被跟踪的请求。
  - DynamoDB 表示附加到此解析器的数据源。它包含对 DynamoDB 进行网络调用以解析该字段的延迟。
  - responseMappingTemplateEvaluation 表示 AWS AppSync 在查询中评估该元素的响应映射模板所花的时间。

在 X-Ray 中查看跟踪时，您可以选择 AWS AppSync 分段中的子段并浏览详细视图，以获取有关这些子段的其他上下文和元数据信息。

对于某些深度嵌套查询或复杂查询，请注意，AWS AppSync 传送到 X-Ray 的分段可能超过分段文档允许的最大大小（在 AWS X-Ray Segment Documents 中定义了该大小）。X-Ray 不会显示超过限制的分段。

使用 AWS CloudTrail 记录 AWS AppSync API 调用

AWS AppSync 与 AWS CloudTrail 集成，后者是在 AWS AppSync 中记录用户、角色或 AWS 服务所执行操作的服务。CloudTrail 将 AWS AppSync 的所有 API 调用作为事件捕获。捕获的调用包含来自 AWS AppSync 控制台和来自对 AWS AppSync API 的代码调用的调用。您可以使用 CloudTrail 收集的信息确定向 AWS AppSync 发出的请求、请求者的 IP 地址、发出请求的人、发出请求的时间以及其他详细信息。

您可以创建跟踪以将 CloudTrail 事件持续传送到 Amazon Simple Storage Service (Amazon S3) 存储桶，包括 AWS AppSync 事件。如果您未配置跟踪，您仍然可以在 CloudTrail 控制台中查看最新事件。

> **Important**
>
> 当前不会记录所有 GraphQL 操作。AppSync 不会将查询和变量操作记录到 CloudTrail 中。

有关 CloudTrail 的更多信息，请参阅《AWS CloudTrail 用户指南》。

CloudTrail 中的 AWS AppSync 信息

在您创建 AWS 账户时，将在该账户上启用 CloudTrail。在 CloudTrail 控制台的事件历史记录中，可以查看、搜索和下载您的 AWS 账户中的最近事件。有关更多信息，请参阅 AWS CloudTrail 用户指南中的使用 CloudTrail 事件历史记录查看事件。

要持续记录 AWS 账户中的事件（包括 AWS AppSync 的事件），请创建跟踪。预设情况下，在控制台中创建跟踪时，此跟踪应用于所有 AWS 区域。此跟踪记录在 AWS 分区中记录所有区域中的事件，并将日志文件传送至您指定的 Amazon S3 存储桶。此外，您可以配置其他AWS服务，进一步分析在 CloudTrail 日志中收集的事件数据并采取行动。有关更多信息，请参阅 AWS CloudTrail 用户指南中的以下内容：

- 为您的 AWS 账户创建跟踪
- AWS 服务与 CloudTrail 日志的集成
- 为 CloudTrail 配置 Amazon SNS 通知
- 从多个区域接收 CloudTrail 日志文件
- 从多个账户接收 CloudTrail 日志文件

CloudTrail 记录所有 AWS AppSync API 操作。例如，对 `CreateGraphqlApi`、`CreateDataSource` 和 `ListResolvers` API 的调用在 CloudTrail 日志文件中生成条目。在 AWS AppSync API Reference 中介绍了这些操作和其他操作。

每个事件或日志条目都包含有关生成请求的人员信息。身份信息可以帮助您确定：

- 请求是使用根用户凭证还是 AWS Identity and Access Management（IAM）用户凭证发出的。
- 请求是使用角色还是联合身份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 AWS CloudTrail 用户指南中的 CloudTrail userIdentity 元素。

了解 AWS AppSync 日志文件条目

CloudTrail 将事件作为包含一个或多个日志条目的日志文件提供。一个事件表示来自任何源的单个请求，并包括有关请求的操作、操作日期和时间、请求参数等信息。由于这些日志文件不是公有 API 调用的有序堆栈跟踪，因此，它们不会按任何特定顺序显示。

以下示例 CloudTrail 日志条目说明了 `CreateApiKey` 操作。

```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/Alice",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "Alice"
    },
    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "CreateApiKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
    "requestParameters": {
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": {
```

```
      "apiKey": {
      "id": "***",
      "expires": 1518037200000
      }
    },
    "requestID": "99999999-9999-9999-9999-999999999999",
    "eventID": "99999999-9999-9999-9999-999999999999",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
    }
  ]
}
```

以下示例 CloudTrail 日志条目说明了 `ListApiKeys` 操作。

```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/Alice",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "Alice"
    },
    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "ListApiKeys",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
    "requestParameters": {
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": {
      "apiKeys": [
              {
                    "id": "***",
                    "expires": 1517954400000
              },
              {
```

```
                    "id": "***",
                    "expires": 1518037200000
                },
            ]
    },
    "requestID": "99999999-9999-9999-9999-999999999999",
    "eventID": "99999999-9999-9999-9999-999999999999",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
    }
  ]
}
```

以下示例 CloudTrail 日志条目说明了 DeleteApiKey 操作。

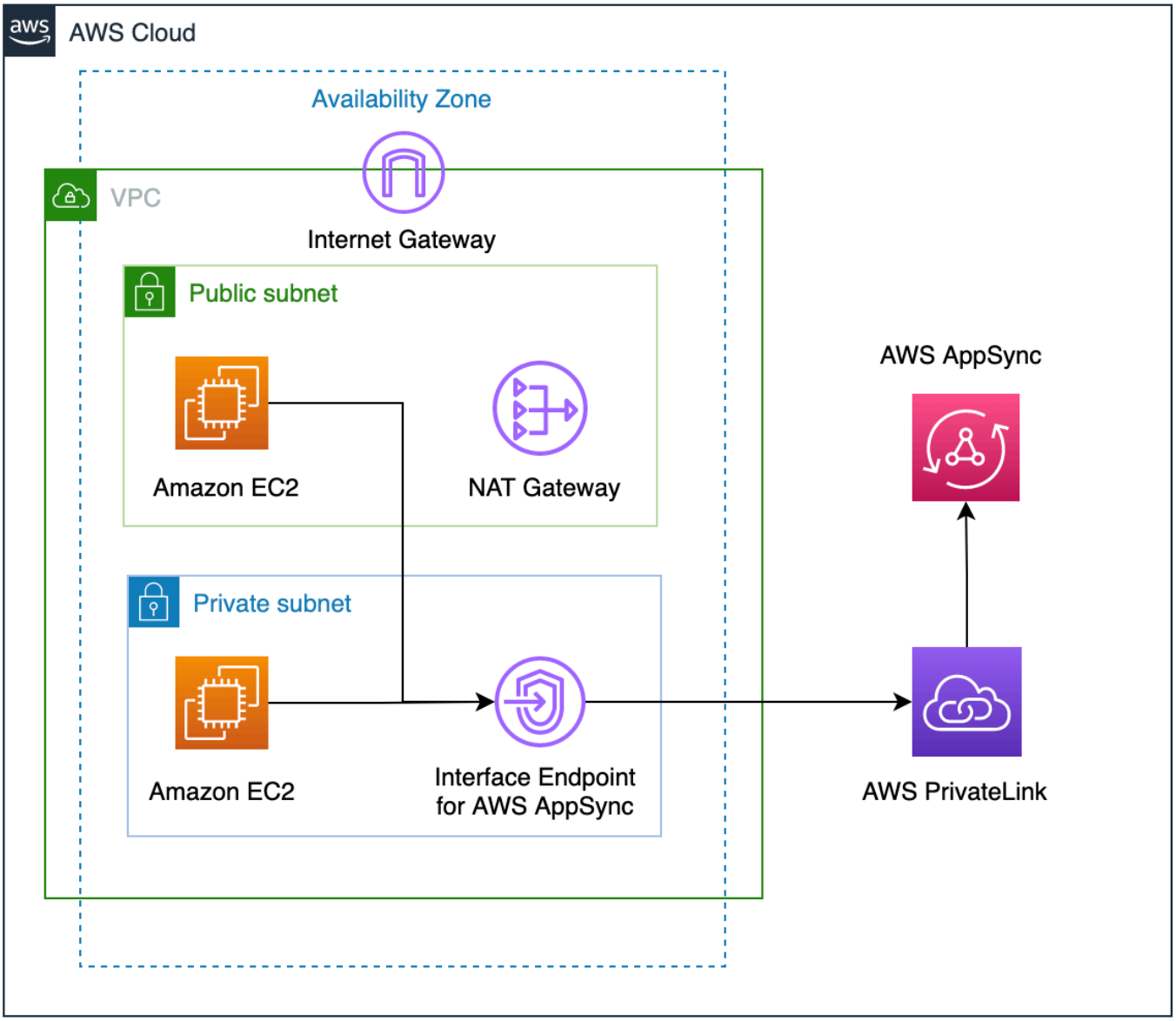
```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/Alice",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "Alice"
    },
    "eventTime": "2018-01-31T21:49:09Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "DeleteApiKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.2.0.1",
    "userAgent": "aws-cli/1.11.72 Python/2.7.11 Darwin/16.7.0 botocore/1.5.35",
    "requestParameters": {
      "id": "***",
      "apiId": "a1b2c3d4e5f6g7h8i9jexample"
    },
    "responseElements": null,
    "requestID": "99999999-9999-9999-9999-999999999999",
    "eventID": "99999999-9999-9999-9999-999999999999",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "recipientAccountId": "111122223333"
```

```
    }
  ]
}
```

使用 AWS AppSync 私有 API

如果使用 Amazon Virtual Private Cloud (Amazon VPC)，您可以创建 AWS AppSync 私有 API，只能从 VPC 中访问这些 API。通过使用私有 API，您可以限制对内部应用程序的 API 访问并连接到 GraphQL 和实时终端节点，而不会公开暴露数据。

要在您的 VPC 和 AWS AppSync 服务之间建立私有连接，您必须创建一个接口 VPC 终端节点。接口终端节点由 AWS PrivateLink 提供支持，该技术支持您通过私有连接访问 AWS AppSync API，而无需采用互联网网关、NAT 设备、VPN 连接或 AWS Direct Connect 连接。VPC 中的实例即使没有公有 IP 地址也可与 AWS AppSync API 进行通信。您的 VPC 和 AWS AppSync 之间的流量不会离开 AWS 网络。



在启用私有 API 功能之前，需要考虑一些其他因素：

- 通过为启用了私有 DNS 功能的 AWS AppSync 设置 VPC 接口终端节点，将禁止 VPC 中的资源使用 AWS AppSync 生成的 API URL 调用其他 AWS AppSync 公有 API。这是因为对公有 API 的请求是通过接口终端节点路由的，而公有 API 不允许这样做。要在该场景中调用公有 API，建议在公有 API 上配置自定义域名，VPC 中的资源可以使用这些域名调用公有 API。
- 您的 AWS AppSync 私有 API 只能从您的 VPC 中使用。只有在您的浏览器的网络配置可以将流量路由到您的 VPC（例如，通过 VPN 或 AWS Direct Connect 连接）时，AWS AppSync 控制台查询编辑器才能够访问您的 API。

- 通过使用 AWS AppSync 的 VPC 接口终端节点，您可以访问同一 AWS 账户和区域中的任何私有 API。要进一步限制对私有 API 的访问，您可以考虑使用以下选项：
  - 确保仅所需的管理员可以为 AWS AppSync 创建 VPC 终端节点接口。
  - 使用 VPC 终端节点自定义策略限制可以从 VPC 中的资源调用哪些 API。
  - 对于 VPC 中的资源，我们建议您使用 IAM 授权调用 AWS AppSync API，以确保为资源分配 API 的范围缩小角色。
- 在创建或使用限制 IAM 主体的策略时，您必须将方法的 `authorizationType` 设置为 `AWS_IAM` 或 `NONE`。

创建 AWS AppSync 私有 API

以下步骤说明了如何在 AWS AppSync 服务中创建私有 API。

> **Warning**
>
> 您只能在创建 API 期间启用私有 API 功能。在创建 AWS AppSync API 或 AWS AppSync 私有 API 后，无法在这些 API 上修改该设置。

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
   - 在控制面板中，选择创建 API。
2. 选择从头开始设计 API，然后选择下一步。
3. 在私有 API 部分中，选择使用私有 API 功能。
4. 配置其余选项，检查您的 API 的数据，然后选择创建。

您必须先在 VPC 中为 AWS AppSync 配置一个接口终端节点，然后才能使用 AWS AppSync 私有 API。请注意，私有 API 和 VPC 必须位于同一 AWS 账户和区域中。

为 AWS AppSync 创建接口终端节点

您可以使用 Amazon VPC 控制台或 AWS Command Line Interface (AWS CLI) 为 AWS AppSync 创建接口终端节点。有关更多信息，请参阅 Amazon VPC 用户指南中的创建接口端点

Console

1. 登录到 AWS Management Console，然后打开 Amazon VPC 控制台的终端节点页面。

2. 选择 Create endpoint（创建端点）。
   a. 在服务类别字段中，验证是否选择了 AWS 服务。
   b. 在服务表中，选择 `com.amazonaws.{region}.appsync-api`。验证类型列值是否为 `Interface`。
   c. 在 VPC 字段中，选择一个 VPC 及其子网。
   d. 要为接口终端节点启用私有 DNS 功能，请选中启用 DNS 名称复选框。
   e. 在安全组字段中，选择一个或多个安全组。
3. 选择 Create endpoint（创建端点）。

CLI

使用 `create-vpc-endpoint` 命令并指定 VPC ID、VPC 端点类型（接口）、服务名称、将使用端点的子网以及要与端点的网络接口关联的安全组。例如：

```
$ aws ec2 create-vpc-endpoint –vpc-id vpc-ec43eb89 \
  –vpc-endpoint-type Interface \
  –service-name com.amazonaws.{region}.appsync-api \
  –subnet-id subnet-abababab –security-group-id sg-1a2b3c4d
```

要使用私有 DNS 选项，您必须设置 VPC 的 `enableDnsHostnames` 和 `enableDnsSupportattributes` 值。有关更多信息，请参阅 Amazon VPC 用户指南中的查看和更新 VPC 的 DNS 支持。如果您为接口终端节点启用私有 DNS 功能，您可以使用其默认公有 DNS 终端节点向 AWS AppSync API GraphQL 和实时终端节点发出请求（使用以下格式）：

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

有关服务终端节点的更多信息，请参阅 AWS General Reference 中的 Service endpoints and quotas。

有关与服务接口终端节点的交互的更多信息，请参阅《Amazon VPC 用户指南》中的使用接口 VPC 端点访问服务。

有关使用 AWS CloudFormation 创建和配置终端节点的信息，请参阅 AWS CloudFormation User Guide 中的 AWS::EC2::VPCEndpoint。

高级 示例

如果您为接口终端节点启用私有 DNS 功能，您可以使用其默认公有 DNS 终端节点向 AWS AppSync API GraphQL 和实时终端节点发出请求（使用以下格式）：

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

在使用接口 VPC 终端节点公有 DNS 主机名时，调用 API 的基本 URL 将采用以下格式：

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.
{region}.vpce.amazonaws.com/graphql
```

如果已在可用区中部署终端节点，您也可以使用可用区特定的 DNS 主机名：

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}-{az_id}.appsync-api.
{region}.vpce.amazonaws.com/graphql.
```

使用 VPC 终端节点公有 DNS 名称需要将 AWS AppSync API 终端节点主机名作为 `Host` 或 `x-appsync-domain` 标头传递给请求。这些示例使用在启动示例架构指南中创建的 `TodoAPI`：

```
curl https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.
{region}.vpce.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-H "Host:{api_url_identifier}.appsync-api.{region}.amazonaws.com" \
-d '{"query":"mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input:
 $createtodoinput) {\n id\n name\n where\n when\n description\n }\n}","variables":
{"createtodoinput":{"name":"My first GraphQL task","when":"Friday Night","where":"Day
 1","description":"Learn more about GraphQL"}}}'
```

在以下示例中，我们使用在启动示例架构指南中生成的 Todo 应用程序。为了测试示例 Todo API，我们将使用私有 DNS 调用该 API。您可以使用所选的任何命令行工具；该示例使用 curl 发送查询和变更，并使用 wscat 设置订阅。要模拟我们的示例，请将下面命令中的花括号 ({ }) 中的值替换为您的 AWS 账户中的相应值。

测试变更操作 - **`createTodo`** 请求

```
curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}" \
```

```
-d '{"query":"mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input:
 $createtodoinput) {\n id\n name\n where\n when\n description\n }\n}","variables":
{"createtodoinput":{"name":"My first GraphQL task","when":"Friday Night","where":"Day
 1","description":"Learn more about GraphQL"}}}'
```

测试变更操作 - **createTodo** 响应

```
{
    "data": {
        "createTodo": {
            "id": "<todo-id>",
            "name": "My first GraphQL task",
            "where": "Day 1",
            "when": "Friday Night",
            "description": "Learn more about GraphQL"
        }
    }
}
```

测试查询操作 - **listTodos** 请求

```
curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query":"query ListTodos {\n listTodos {\n items {\n description\n id\n name
\n when\n where\n }\n }\n}\n","variables":{"createtodoinput":{"name":"My first
 GraphQL task","when":"Friday Night","where":"Day 1","description":"Learn more about
 GraphQL"}}}'
```

测试查询操作 - **listTodos** 请求

```
{
  "data": {
    "listTodos": {
      "items": [
        {
          "description": "Learn more about GraphQL",
          "id": "<todo-id>",
          "name": "My first GraphQL task",
          "when": "Friday night",
          "where": "Day 1"
        }
```

```
      ]
    }
  }
}
```

测试订阅操作 - 订阅 **createTodo** 变更

要在 AWS AppSync 中设置 GraphQL 订阅，请参阅构建实时 WebSocket 客户端。从 VPC 上的 Amazon EC2 实例中，您可以使用 wscat 测试您的 AWS AppSync 私有 API 订阅终端节点。以下示例使用 API KEY 进行授权。

```
$ header=`echo '{"host":"{api_url_identifier}.appsync-api.{region}.amazonaws.com","x-
api-key":"da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}"}' | base64 | tr -d '\n'`
$ wscat -p 13 -s graphql-ws -c  "wss://{api_url_identifier}.appsync-realtime-api.us-
west-2.amazonaws.com/graphql?header=$header&payload=e30="
Connected (press CTRL+C to quit)
> {"type": "connection_init"}
< {"type":"connection_ack","payload":{"connectionTimeoutMs":300000}}
< {"type":"ka"}
> {"id":"f7a49717","payload":{"data":"{\"query\":\"subscription
 onCreateTodo {onCreateTodo {description id name where when}}\",
\"variables\":{}}","extensions":{"authorization":{"x-api-key":"da2-
{xxxxxxxxxxxxxxxxxxxxxxxxxx}","host":"{api_url_identifier}.appsync-api.
{region}.amazonaws.com"}}},"type":"start"}
< {"id":"f7a49717","type":"start_ack"}
```

或者，使用 VPC 终端节点域名，同时确保在 wscat 命令中指定 Host 标头以建立 WebSocket 连接：

```
$ header=`echo '{"host":"{api_url_identifier}.appsync-api.{region}.amazonaws.com","x-
api-key":"da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}"}' | base64 | tr -d '\n'`
$ wscat -p 13 -s graphql-ws -c  "wss://{vpc_endpoint_id}-
{endpoint_dns_identifier}.appsync-api.{region}.vpce.amazonaws.com/graphql?header=
$header&payload=e30=" --header Host:{api_url_identifier}.appsync-realtime-api.us-
west-2.amazonaws.com
Connected (press CTRL+C to quit)
> {"type": "connection_init"}
< {"type":"connection_ack","payload":{"connectionTimeoutMs":300000}}
< {"type":"ka"}
> {"id":"f7a49717","payload":{"data":"{\"query\":\"subscription
 onCreateTodo {onCreateTodo {description id priority title}}\",
\"variables\":{}}","extensions":{"authorization":{"x-api-key":"da2-
{xxxxxxxxxxxxxxxxxxxxxxxxxx}","host":"{api_url_identifier}.appsync-api.
{region}.amazonaws.com"}}},"type":"start"}
```

```
< {"id":"f7a49717","type":"start_ack"}
```

运行下面的变更代码：

```
curl https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql \
-H "Content-Type:application/graphql" \
-H "x-api-key:da2-{xxxxxxxxxxxxxxxxxxxxxxxxxx}" \
-d '{"query":"mutation add($createtodoinput: CreateTodoInput!) {\n createTodo(input:
 $createtodoinput) {\n id\n name\n where\n when\n description\n }\n}","variables":
{"createtodoinput":{"name":"My first GraphQL task","when":"Friday Night","where":"Day
 1","description":"Learn more about GraphQL"}}}'
```

然后，将触发订阅并显示消息通知，如下所示：

```
< {"id":"f7a49717","type":"data","payload":{"data":{"onCreateTodo":{"description":"Go
 to the shops","id":"169ce516-b7e8-4a6a-88c1-ab840184359f","priority":5,"title":"Go to
 the shops"}}}}
```

使用 IAM 策略限制创建公有 API

AWS AppSync支持将 IAM Condition 语句与私有 API 一起使用。可以在 `appsync:CreateGraphqlApi` 操作的 IAM 策略语句中包含 `visibility` 字段，以控制哪些 IAM 角色和用户可以创建私有和公有 API。这样，IAM 管理员就能够定义仅允许用户创建私有 GraphQL API 的 IAM 策略。尝试创建公有 API 的用户将收到“未经授权”消息。

例如，IAM 管理员可以创建以下 IAM 策略语句以允许创建私有 API：

```
{
    "Sid": "AllowPrivateAppSyncApis",
    "Effect": "Allow",
    "Action": "appsync:CreateGraphqlApi",
    "Resource": "*",
    "Condition": {
        "ForAnyValue:StringEquals": {
            "appsync:Visibility": "PRIVATE"
        }
    }
}
```

IAM 管理员还可以添加以下服务控制策略，以阻止 AWS 组织中的所有用户创建私有 API 以外的 AWS AppSync API：

```
{
    "Sid": "BlockNonPrivateAppSyncApis",
    "Effect": "Deny",
    "Action": "appsync:CreateGraphqlApi",
    "Resource": "*",
    "Condition": {
        "ForAnyValue:StringNotEquals": {
            "appsync:Visibility": "PRIVATE"
        }
    }
}
```

使用 AWS AppSync 配置 GraphQL 运行复杂度、查询深度和自省

AWS AppSync 允许您启用或禁用自省特征，并对单个查询中的嵌套级别和解析器数量设置限制。

使用自省特征

> **Tip**
>
> 有关 GraphQL 中的自省的更多信息，请参阅 GraphQL 基金会网站上的这篇文章。

默认情况下，GraphQL 允许您使用自省来查询架构本身，以发现其类型、字段、查询、突变、订阅等。这是一项用于了解 GraphQL 服务如何塑造和处理数据的重要特征。然而，在处理自省时，有一些事情需要考虑。您可能有一个用例，该用例受益于禁用自省，例如字段名称可能是敏感的或隐藏的，或者不为使用者记录整个 API 架构。在这些情况下，通过自省发布架构数据可能会导致泄露特意保密的数据。

为了防止这种情况发生，您可以禁用自省。这将防止未经授权的各方在您的架构上使用自省字段。但值得注意的是，自省对于开发团队了解其服务中数据的处理方式非常有用。在内部将自省保持为启用状态，而在生产代码中将其禁用，这可能会有所帮助，可作为一层额外的安全措施。处理这个问题的另一种方式是添加授权方法，AWS AppSync 也提供了这一方法。有关更多信息，请参阅授权。

AWS AppSync 允许您在 API 级别启用或禁用自省。要启用或禁用自省，请执行以下操作：

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
2. 在 API 页面上，选择一个 GraphQL API 的名称。

3. 在您的 API 主页的导航窗格中，选择设置。
4. 在 API 配置中，选择编辑。
5. 在自省查询下，执行以下操作：
   - 开启或关闭启用自省查询。
6. 选择 Save（保存）。

启用自省（默认行为）后，自省系统将正常工作。例如，下图显示了一个处理架构中所有可用类型的 __schema 字段：

禁用此特征时，响应中将出现验证错误：

配置查询深度限制

有时您可能需要更精细地控制 API 在操作期间的运行方式。一个此类控制措施是对查询可以处理的嵌套级别数量添加限制。默认情况下，查询能够处理不限数量的嵌套级别。将查询限制为指定数量的嵌套级别可能会影响项目的性能和灵活性。执行以下查询：

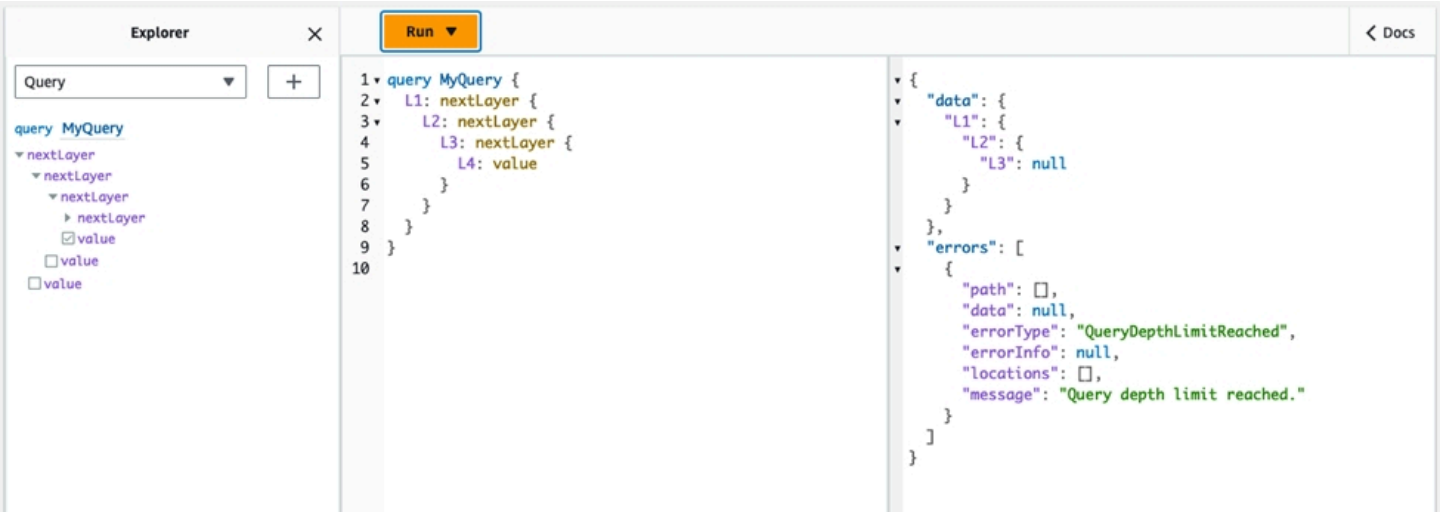
```
query MyQuery {
  L1: nextLayer {
    L2: nextLayer {
      L3: nextLayer {
        L4: value
      }
    }
  }
}
```

您的项目可能出于某种目的而要求将查询限制为 L1 或 L2。默认情况下，从 L1 到 L4 的整个查询都将在无法控制的情况下进行处理。通过设置限制，可以防止查询访问超过指定级别的任何内容。

要添加查询深度限制，请执行以下操作：

1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
2. 在 API 页面上，选择一个 GraphQL API 的名称。
3. 在您的 API 主页的导航窗格中，选择设置。
4. 在 API 配置中，选择编辑。
5. 在查询深度下，执行以下操作：
   a. 开启或关闭启用查询深度。
   b. 在最大深度中，设置深度限制。可以介于 1 和 75 之间。
6. 选择 Save（保存）。

设置限制后，超过其上限将导致 QueryDepthLimitReached 错误。例如，下图显示了一个查询，其深度限制 2 超过第三级 (L3) 和第四级 (L4) 的限制：



请注意，在架构中，字段仍然可以标记为可为 null 或不可为 null。如果不可为 null 的字段收到 `QueryDepthLimitReached` 错误，则会对第一个可为 null 的父字段引发该错误。

配置解析器计数限制

您还可以控制每个查询可以处理多少个解析器。与查询深度一样，您可以为此数量设置限制。采用以下包含三个解析器的查询：

```
query MyQuery {
  resolver1: resolver
  resolver2: resolver
  resolver3: resolver
}
```

默认情况下，每个查询最多可以处理 10000 个解析器。在上面的示例中，将处理 `resolver1`、`resolver2` 和 `resolver3`。但是，您的项目可能需要将每个查询限制为总共处理一个或两个解析器。通过设置限制，您可以告诉查询不要处理任何超过特定数量的解析器，比如第一个 (`resolver1`) 或第二个 (`resolver2`) 解析器。

要添加解析器计数限制，请执行以下操作：

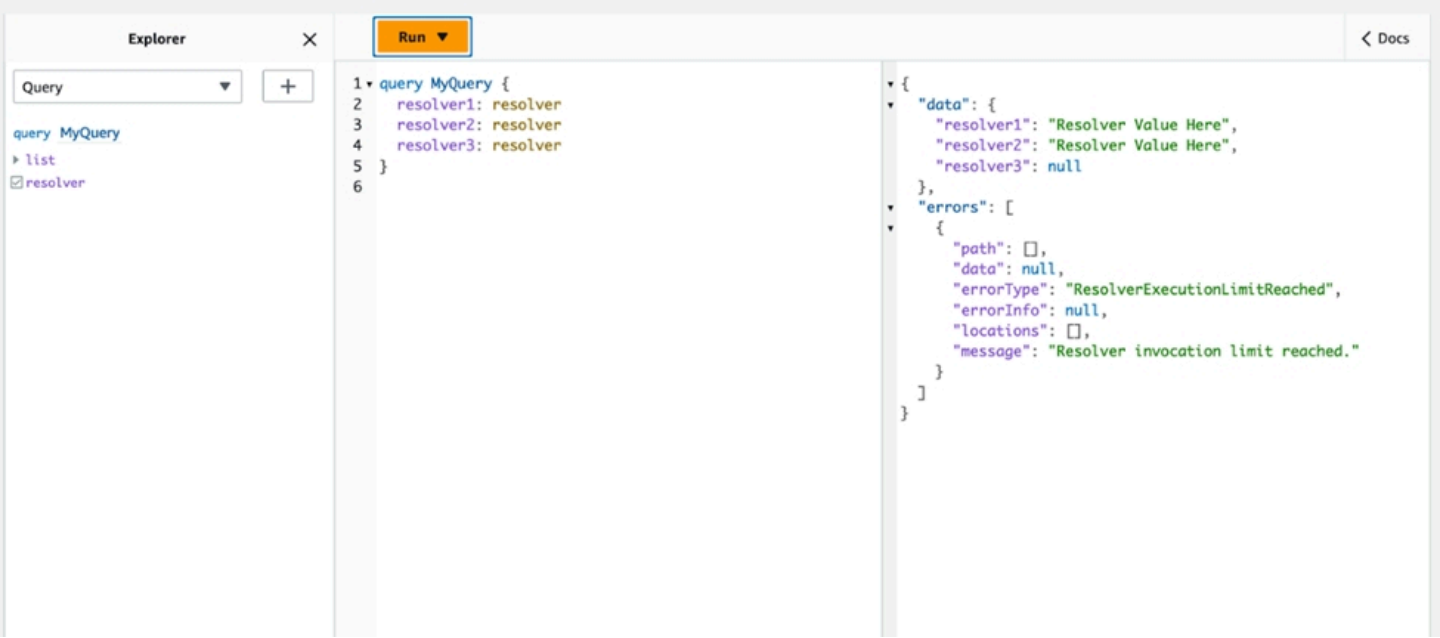
1. 登录到 AWS Management Console，然后打开 AppSync 控制台。
2. 在 API 页面上，选择一个 GraphQL API 的名称。
3. 在您的 API 主页的导航窗格中，选择设置。
4. 在 API 配置中，选择编辑。
5. 在解析器计数限制下，执行以下操作：

a. 开启启用解析器计数。

b. 在最大解析器数量中，设置计数限制。可以介于 1 和 10000 之间。

6. 选择 Save（保存）。

与查询深度限制一样，超过配置的解析器限制会导致查询在其它解析器上结束并出现 `ResolverExecutionLimitReached` 错误。在下图中，解析器计数限制为 2 的查询尝试处理三个解析器。由于限制，第三个解析器会引发错误并且不会运行。



在中使用环境变量 AWS AppSync

您可以使用环境变量来调整 AWS AppSync 解析器和函数的行为，而无需更新代码。环境变量是存储在您的 API 配置中的成对字符串，可供您的解析器和函数在运行时使用。对于必须引用仅在初始设置期间可用、但需要在运行期间由解析器和函数使用的配置数据，它们特别有用。环境变量在您的代码中公开配置数据，从而减少了对这些值进行硬编码的需求。

> **Note**
>
> 为了提高数据库的安全性，我们建议您使用 S ecrets Manager 或 S AWS ystems Manager 参数存储来代替环境变量来存储凭据或敏感信息。要利用此功能，请参阅使用 AWS AppSync HTTP 数据源调用 AWS 服务。

环境变量必须遵循多种行为和规则才能正常运行：

- JavaScript 解析器/函数和 VTL 模板都支持环境变量。
- 在调用函数之前，不会评估环境变量。
- 环境变量仅支持字符串值。
- 环境变量中任何已定义的值都被视为字符串字面值，不会被扩展。
- 理想情况下，应在函数代码中执行变量求值。

配置环境变量（控制台）

您可以通过创建变量并定义其键值对来为 AWS AppSync GraphQL API 配置环境变量。您的解析器和函数将在运行时使用环境变量的键名来检索该值。要在 AWS AppSync 控制台中设置环境变量，请执行以下操作：

1. 登录 AWS Management Console 并打开AppSync控制台。
2. 在 API 页面上，选择一个 GraphQL API 的名称。
3. 在您的 API 主页的导航窗格中，选择设置。
4. 在环境变量下，选择添加环境变量。
5. 选择 Add environment variable (添加环境变量)。
6. 输入密钥和值。
7. 如有必要，重复步骤 5 和 6 以添加更多键值。如果需要删除密钥值，请选择移除选项和要删除的密钥。
8. 选择提交。

> **Tip**
>
> 创建键和值时必须遵循以下几条规则：
>
> - 密钥必须以字母开头。
> - 密钥长度必须至少为两个字符。
> - 密钥只能包含字母、数字和下划线字符 (_)。
> - 值的长度最多可为 512 个字符。
> - 您最多可以在 GraphQL API 中配置 50 个键值对。

配置环境变量（API）

要使用 API 设置环境变量，可以使用PutGraphqlApiEnvironmentVariables。相应的 CLI 命令是put-graphql-api-environment-variables。

要使用 API 检索环境变量，可以使用GetGraphqlApiEnvironmentVariables。相应的 CLI 命令是get-graphql-api-environment-variables。

该命令必须包含 API ID 和环境变量列表：

```
aws appsync put-graphql-api-environment-variables \
  --api-id "<api-id>" \
  --environment-variables '{"key1":"value1","key2":"value2", …}'
```

以下示例在 API 中设置了两个环境变量，其ID为abcdefghijklmnopqrstuvwxyz使用put-graphql-api-environment-variables命令：

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true"}'
```

请注意，使用put-graphql-api-environment-variables命令应用环境变量时，环境变量结构的内容将被覆盖；这意味着现有的环境变量将丢失。要在添加新环境变量时保留现有的环境变量，请在请求中包括所有现有的键值对以及新的键值对。使用上面的示例，如果你想添加"EMPTY":""，你可以执行以下操作：

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true", "EMPTY":""}'
```

要检索当前配置，请使用以下get-graphql-api-environment-variables命令：

```
aws appsync get-graphql-api-environment-variables --api-id "<api-id>"
```

使用上面的示例，你可以使用以下命令：

```
aws appsync get-graphql-api-environment-variables --api-id "abcdefghijklmnopqrstuvwxyz"
```

结果将显示环境变量列表及其键值：

```
{
    "environmentVariables": {
        "USER_TABLE": "users_prod",
        "DEBUG": "true",
        "EMPTY": ""
    }
}
```

配置环境变量 (CFN)

您可以使用下面的模板来创建环境变量：

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  GraphQLApiWithEnvVariables:
    Type: "AWS::AppSync::GraphQLApi"
    Properties:
      Name: "MyApiWithEnvVars"
      AuthenticationType: "AWS_IAM"
      EnvironmentVariables:
        EnvKey1: "non-empty"
        EnvKey2: ""
```

环境变量和合并的 API

在源 API 中定义的环境变量也可在您的合并的 API 中使用。Merged API 中的环境变量是只读的，无法更新。请注意，您的环境变量密钥在所有 Source API 中必须是唯一的，合并才会成功；重复的密钥总是会导致合并失败。

检索环境变量

要检索函数代码中的环境变量，请从解析器和函数中的`ctx.env`对象中检索该值。以下是一些实际应用示例。

Publishing to Amazon SNS

在本示例中，我们的 HTTP 解析器向 Amazon SNS 主题发送了一条消息。只有在定义 GraphQL API 的堆栈和主题部署完毕后，才会知道该主题的 ARN。

```
/**
```

```
 * Sends a publish request to the SNS topic
 */
export function request(ctx) {
    const TOPIC_ARN = ctx.env.TOPIC_ARN;
    const { input: values } = ctx.args;
    // this custom function sends values to the SNS topic
    return publishToSNSRequest(TOPIC_ARN, values);
}
```

Transactions with DynamoDB

在此示例中，如果 API 是为暂存部署的，或者已经在生产中，则 DynamoDB 表的名称会有所不同。无需更改解析器代码。环境变量的值会根据 API 的部署位置进行更新。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: ctx.env.POST_TABLE,
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({ postId }),
        // rest of the configuration
      },
      {
        table: ctx.env.AUTHOR_TABLE,
        operation: 'UpdateItem',
        key: util.dynamodb.toMapValues({ authorId }),
        // rest of the configuration
      },
    ],
  };
}
```

授权和身份验证

本节介绍用于配置应用程序的安全性和数据保护的选项。

授权类型

您可以通过五种方式授权应用程序与您的 AWS AppSync GraphQL API 进行交互。您可以通过在 AWS AppSync API 或 CLI 调用中指定以下授权类型值之一来指定所使用的授权类型：

- **API_KEY**

  适用于使用 API 密钥。

- **AWS_LAMBDA**

  用于使用 AWS Lambda 函数。

- **AWS_IAM**

  用于使用 AWS Identity and Access Management (IAM) 权限。

- **OPENID_CONNECT**

  适用于使用 OpenID Connect 提供程序。

- **AMAZON_COGNITO_USER_POOLS**

  适用于使用 Amazon Cognito 用户池。

这些基本授权类型适用于大多数开发人员。对于更高级的使用案例，您可以通过控制台、CLI 和 AWS CloudFormation添加其他授权模式。对于其他授权模式， AWS AppSync 提供一种采用上面列出的值的授权类型（即API_KEY、AWS_LAMBDA、AWS_IAM、OPENID_CONNECT、和AMAZON_COGNITO_USER_POOLS）。

在将 API_KEY、AWS_LAMBDA 或 AWS_IAM 指定为主要或默认授权类型时，您不能再次将其指定为其他授权模式之一。同样，您不能在其他授权模式中重复使用 API_KEY、AWS_LAMBDA 或 AWS_IAM。您可以使用多个 Amazon Cognito 用户池和 OpenID Connect 提供程序。不过，您不能在默认授权模式和任何其他授权模式之间使用重复的 Amazon Cognito 用户池或 OpenID Connect 提供程序。您可以使用相应的配置正则表达式，为 Amazon Cognito 用户池或 OpenID Connect 提供程序指定不同的客户端。

API_KEY 授权

未经身份验证的 API 比经过身份验证的 API 需要更严格的限制。一种对未经身份验证的 GraphQL 终端节点的限制进行控制的方法是使用 API 密钥。API 密钥是应用程序中的硬编码值，在您创建未经身份验证的 GraphQL 端点时由 AWS AppSync 服务生成。您可以通过控制台或 CLI 轮换 API 密钥，或者按照 AWS AppSync API reference 轮换 API 密钥。

Console

1. 登录 AWS Management Console 并打开AppSync控制台。
   a. 在 API 控制面板中，选择您的 GraphQL API。
   b. 在侧边栏中，选择设置。
2. 在默认授权模式下面，选择 API 密钥。
3. 在 API 密钥表中，选择添加 API 密钥。

   将在表中生成一个新的 API 密钥。

   - 要删除旧 API 密钥，请在表中选择该 API 密钥，然后选择删除。
4. 在页面底部选择 Save (保存)。

CLI

1. 如果您尚未这样做，请配置您对 AWS CLI 的访问权限。有关更多信息，请参阅配置基础知识。
2. 运行 `update-graphql-api` 命令以创建 GraphQL API 对象。

   您需要为该特定命令键入两个参数：

   1. 您的 GraphQL API 的 `api-id`。
   2. 您的 API 的新 `name`。您可以使用相同的 `name`。
   3. `authentication-type`，它是 `API_KEY`。

   > **Note**
   > 必须配置其他参数（例如 `Region`），但这些参数通常默认为您的 CLI 配置值。

示例命令可能如下所示：

```
aws appsync update-graphql-api --api-id abcdefghijklmnopqrstuvwxyz --name
 TestAPI --authentication-type API_KEY
```

将在 CLI 中返回输出。以下是 JSON 格式的示例：

```
{
    "graphqlApi": {
        "xrayEnabled": false,
        "name": "TestAPI",
        "authenticationType": "API_KEY",
        "tags": {},
        "apiId": "abcdefghijklmnopqrstuvwxyz",
        "uris": {
            "GRAPHQL": "https://s8i3kk3ufhe9034ujnv73r513e.appsync-api.us-
west-2.amazonaws.com/graphql",
            "REALTIME": "wss://s8i3kk3ufhe9034ujnv73r513e.appsync-realtime-
api.us-west-2.amazonaws.com/graphql"
        },
        "arn": "arn:aws:appsync:us-west-2:348581070237:apis/
abcdefghijklmnopqrstuvwxyz"
    }
}
```

API 密钥可配置为最多 365 天，并且您可以将现有到期日期再延长多达 365 天（从到期日期当天开始）。建议将 API 密钥用于开发目的或可以安全公开公有 API 的使用案例。

在客户端，通过标头 `x-api-key` 指定 API 密钥。

例如，如果您的 `API_KEY` 为 `'ABC123'`，则您可以通过 `curl` 发送 GraphQL 查询，如下所示：

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "x-api-key:ABC123" -d
 '{ "query": "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

AWS_LAMBDA 授权

您可以使用 AWS Lambda 函数实现自己的 API 授权逻辑。您可以将 Lambda 函数用于主要或辅助授权者，但每个 API 只能有一个 Lambda 授权函数。在使用 Lambda 函数进行授权时，以下限制适用：

- 如果 API 启用了 `AWS_LAMBDA` 和 `AWS_IAM` 授权模式，则不能将 SigV4 签名作为 `AWS_LAMBDA` 授权令牌。
- 如果 API 启用了 `AWS_LAMBDA` 和 `OPENID_CONNECT` 授权模式或 `AMAZON_COGNITO_USER_POOLS` 授权模式，则不能将 OIDC 令牌作为 `AWS_LAMBDA` 授权令牌。请注意，OIDC 令牌可以采用持有者方案。
- Lambda 函数不能为解析器返回超过 5MB 的上下文数据。

例如，如果您的授权令牌是 `'ABC123'`，您可以通过 curl 发送 GraphQL 查询，如下所示：

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "Authorization:ABC123" -d
 '{ "query":
        "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

Lambda 函数是在每个查询或变更之前调用的。可以根据 API ID 和身份验证令牌缓存返回值。默认情况下，不会开启缓存，但可以在 API 级别启用该功能，或者在函数的返回值中设置 `ttlOverride` 值以启用该功能。

如果需要，可以指定一个正则表达式，以在调用函数之前验证授权令牌。这些正则表达式用于在调用函数之前验证授权令牌格式是否正确。将自动拒绝使用的令牌与该正则表达式不匹配的任何请求。

用于授权的 Lambda 函数需要对其应用委托人策略 AWS AppSync 才能允许调用它们。`appsync.amazonaws.com`此操作在 AWS AppSync 控制台中自动完成； AWS AppSync 控制台不会删除策略。有关向 Lambda 函数附加策略的更多信息，请参阅开发人员指南中的基于资源的策略。 AWS Lambda

您指定的 Lambda 函数将收到具有以下形状的事件：

```
{
    "authorizationToken": "ExampleAUTHtoken123123123",
    "requestContext": {
        "apiId": "aaaaaa123123123example123",
        "accountId": "111122223333",
```

```
        "requestId": "f4081827-1111-4444-5555-5cf4695f339f",
        "queryString": "mutation CreateEvent {...}\n\nquery MyQuery {...}\n",
        "operationName": "MyQuery",
        "variables": {}
    }
    "requestHeaders": {
        application request headers
    }
}
```

该event对象包含应用程序客户端在请求中发送到的标头 AWS AppSync。

授权函数必须至少返回一个布尔值isAuthorized，表示请求是否获得授权。 AWS AppSync 识别从 Lambda 授权函数返回的以下密钥：

函数列表

isAuthorized（布尔值，必需）

一个布尔值，用于指示是否授权 authorizationToken 中的值调用 GraphQL API。

如果该值为 true，则继续执行 GraphQL API。如果该值为 false，则会引发 UnauthorizedException。

deniedFields（字符串列表，可选）

强制更改为 null 的值列表，即使从解析器中返回值也是如此。

每个项目是 arn:aws:appsync:*us-east-1*:*111122223333*:apis/*GraphQLApiId*/types/*TypeName*/fields/*FieldName* 格式的完全限定字段 ARN，或者是 *TypeName*.*FieldName* 缩写格式。当两个 API 共享一个 Lambda 函数授权方并且这两个 API 之间的常见类型和字段之间可能存在歧义时，应使用完整的 ARN 表单。

resolverContext（JSON 对象，可选）

在解析器模板中显示为 $ctx.identity.resolverContext 的 JSON 对象。例如，如果解析器返回以下结构：

```
{
  "isAuthorized":true
  "resolverContext": {
    "banana":"very yellow",
    "apple":"very green"
```

```
  }
}
```

解析器模板中的 `ctx.identity.resolverContext.apple` 值将为 "very green"。`resolverContext` 对象仅支持键值对。不支持嵌套的键。

> **Warning**
>
> 该 JSON 对象的总大小不能超过 5MB。

`ttlOverride`（整数，可选）

应缓存响应的秒数。如果未返回任何值，则使用 API 中的值。如果为 0，则不会缓存响应。

Lambda 授权者的超时为 10 秒。我们建议设计的函数尽可能在最短的时间内执行，以提高 API 的性能。

多个 AWS AppSync API 可以共享一个身份验证 Lambda 函数。不允许跨账户授权者使用。

在多个 API 之间共享授权函数时，请注意缩写的字段名称 (*typename*.*fieldname*) 可能会无意中隐藏字段。要消除 `deniedFields` 中的字段的歧义，您可以使用 `arn:aws:appsync:`*region*`:`*accountId*`:apis/`*GraphQLApiId*`/types/`*typeName*`/fields/`*fieldName* 格式指定明确的字段 ARN。

要在 AWS AppSync中添加 Lambda 函数以作为默认授权模式，请执行以下操作：

Console

1. 登录 AWS AppSync 控制台并导航到您要更新的 API。
2. 导航到您的 API 的“设置”页面。

   将 API 级别授权更改为 AWS Lambda。
3. 选择 AWS 区域 和 Lambda ARN 来授权 API 调用。

> **Note**
>
> 将自动添加相应的主体策略，从而允许 AWS AppSync 调用您的 Lambda 函数。

4. （可选）设置响应 TTL 和令牌验证正则表达式。

AWS CLI

1. 将以下策略附加到使用的 Lambda 函数：

```
aws lambda add-permission --function-name "my-function" --statement-id "appsync"
 --principal appsync.amazonaws.com --action lambda:InvokeFunction --output text
```

> **Important**
>
> 如果您希望将函数策略锁定到单个 GraphQL API，您可以运行以下命令：
>
> ```
> aws lambda add-permission --function-name “my-function” --
> statement-id “appsync” --principal appsync.amazonaws.com --action
>  lambda:InvokeFunction --source-arn “<my AppSync API ARN>” --output text
> ```

2. 更新您的 AWS AppSync API 以使用给定的 Lambda 函数 ARN 作为授权方：

```
aws appsync update-graphql-api --api-id example2f0ur2oid7acexample --
name exampleAPI --authentication-type AWS_LAMBDA --lambda-authorizer-config
 authorizerUri="arn:aws:lambda:us-east-2:111122223333:function:my-function"
```

> **Note**
>
> 您还可以包含其他配置选项，例如令牌正则表达式。

以下示例介绍了一个 Lambda 函数，以说明 Lambda 函数在作为 AWS AppSync 授权机制时可能具有的各种身份验证和失败状态：

```
def handler(event, context):
  # This is the authorization token passed by the client
  token = event.get('authorizationToken')
  # If a lambda authorizer throws an exception, it will be treated as unauthorized.
  if 'Fail' in token:
    raise Exception('Purposefully thrown exception in Lambda Authorizer.')
```

```
  if 'Authorized' in token and 'ReturnContext' in token:
    return {
      'isAuthorized': True,
      'resolverContext': {
        'key': 'value'
      }
    }

  # Authorized with no f
  if 'Authorized' in token:
    return {
      'isAuthorized': True
    }
  # Partial authorization
  if 'Partial' in token:
    return {
      'isAuthorized': True,
      'deniedFields':['user.favoriteColor']
    }
  if 'NeverCache' in token:
    return {
      'isAuthorized': True,
      'ttlOverride': 0
    }
  if 'Unauthorized' in token:
    return {
      'isAuthorized': False
    }
  # if nothing is returned, then the authorization fails.
  return {}
```

规避 SigV4 和 OIDC 令牌授权限制

可以使用以下方法规避启用某些授权模式时无法将 SigV4 签名或 OIDC 令牌作为 Lambda 授权令牌的问题。

如果要在为 AWS AppSync API 启用了 `AWS_IAM` 和 `AWS_LAMBDA` 授权模式时将 SigV4 签名作为 Lambda 授权令牌，请执行以下操作：

- 要创建新的 Lambda 授权令牌，请在 SigV4 签名中添加随机后缀和/或前缀。
- 要检索原始 SigV4 签名，请从 Lambda 授权令牌中删除随机前缀和/或后缀以更新您的 Lambda 函数。然后，使用原始 SigV4 签名进行身份验证。

如果要在的 API 启用授权模式或AMAZON_COGNITO_USER_POOLS和授权模式时使用 OIDC 令牌作为 Lambda AWS_LAMBDA 授权令牌，请执行以下操作：OPENID_CONNECT AWS AppSync

- 要创建新的 Lambda 授权令牌，请在 OIDC 令牌中添加随机后缀和/或前缀。Lambda 授权令牌不应包含持有者方案前缀。
- 要检索原始 OIDC 令牌，请从 Lambda 授权令牌中删除随机前缀和/或后缀以更新您的 Lambda 函数。然后，使用原始 OIDC 令牌进行身份验证。

AWS_IAM 授权

该授权类型对 GraphQL API 强制执行 AWS 签名版本 4 签名过程。您可以将 Identity and Access Management (IAM) 访问策略与此授权类型关联。您的应用程序可以使用访问密钥（由访问密钥 ID 和秘密访问密钥组成）或 Amazon Cognito 联合身份提供的短期临时凭证以利用该关联。

如果您希望有访问权限的角色执行所有数据操作：

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "appsync:GraphQL"
         ],
         "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/*"
         ]
      }
   ]
}
```

你可以在 AppSync控制YourGraphQLApiId台的主API列表页面找到你的 API 名称正下方。或者，您也可以使用 CLI 检索它：aws appsync list-graphql-apis

如果您只想限制对某些 GraphQL 操作的访问，您可以对根 Query、Mutation 和 Subscription 字段执行此操作。

```
{
   "Version": "2012-10-17",
```

```
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "appsync:GraphQL"
         ],
         "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Query/
fields/<Field-1>",
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/Query/
fields/<Field-2>",
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/
Mutation/fields/<Field-1>",
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/
Subscription/fields/<Field-1>"
         ]
     }
   ]
}
```

例如，假设您具有以下架构，并且您想要限制能够获取所有文章的访问权限：

```
schema {
   query: Query
   mutation: Mutation
}

type Query {
   posts:[Post!]!
}

type Mutation {
   addPost(id:ID!, title:String!):Post!
}
```

角色的相应 IAM 策略（例如，您可以将其附加到 Amazon Cognito 身份池）将如下所示：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
```

```
            "appsync:GraphQL"
            ],
            "Resource": [
                "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/types/
Query/fields/posts"
            ]
        }
    ]
}
```

OPENID_CONNECT 授权

该授权类型强制实施 OIDC 兼容服务提供的 OpenID Connect (OIDC) 令牌。您的应用程序可以利用由 OIDC 提供程序定义的用户和权限来控制访问。

发行者 URL 是您向其提供的唯一必需配置值 AWS AppSync （ 例如`https://auth.example.com` ）。此网址必须可通过 HTTPS 进行寻址。 AWS AppSync 附加`/.well-known/openid-configuration`到颁发者网址并按照 OpenID Connect Discovery 规范找到`https://auth.example.com/.well-known/openid-configuration` OpenID 配置。它预期会在此 URL 处检索 RFC5785 兼容的 JSON 文档。此 JSON 文档必须包含一个`jwks_uri`密钥，该密钥指向带有签名密钥的 JSON Web 密钥集 (JWKS) 文档。 AWS AppSync 要求 JWKS 包含和的 JSON 字段。`kty` `kid`

AWS AppSync 支持多种签名算法。

签名算法
RS256
RS384
RS512
PS256
PS384
PS512
HS256

签名算法
HS384
HS512
ES256
ES384
ES512

我们建议您使用 RSA 算法。提供程序颁发的令牌必须包括颁发令牌的时间 (`iat`)，并可能包含对其进行身份验证的时间 (`auth_time`)。您可以在 OpenID Connect 配置中为颁发时间 (`iat`TTL) 和身份验证时间 (`auth`TTL) 提供 TTL 值以进行额外的验证。如果您的提供程序授权多个应用程序，还可以提供一个用于按客户端 ID 进行授权的正则表达式 (`clientId`)。当您`clientId`的 OpenID Connect 配置中存在时，通过要求与令牌中的`aud`或`azp`声明匹配`clientId`来 AWS AppSync 验证声明。

要验证多个客户端 ID，请使用管道运算符（“|”），它是正则表达式中的“or”。例如，如果您的 OIDC 应用程序有四个客户端，其客户端 ID 例如 0A1S2D、1F4G9H、1J6L4B、6GS5MG，仅验证前三个客户端 ID，则应在 “客户端 ID” 字段中放置 1F4G9H|1J6L4B|6GS5MG。

AMAZON_COGNITO_USER_POOLS 授权

该授权类型强制实施 Amazon Cognito 用户池提供的 OIDC 令牌。您的应用程序可以利用您的用户池和其他 AWS 账户的用户池中的用户和群组，并将它们与 GraphQL 字段关联以控制访问权限。

在使用 Amazon Cognito 用户池时，您可以创建用户所属的组。此信息以 JWT 令牌编码，您的应用程序在发送 GraphQL 操作时将其发送到授权标头 AWS AppSync 中。您可以在架构上使用 GraphQL 指令以控制哪些组可以对字段调用哪些解析器，从而向客户提供更受控制的访问。

例如，假设您具有以下 GraphQL 架构：

```
schema {
   query: Query
   mutation: Mutation
}

type Query {
```

```
   posts:[Post!]!
}

type Mutation {
   addPost(id:ID!, title:String!):Post!
}
...
```

如果您在 Amazon Cognito 用户池中具有两个组（bloggers 和 readers），并希望限制读者以使他们无法添加新条目，则架构应如下所示：

```
schema {
   query: Query
   mutation: Mutation
}
```

```
type Query {
   posts:[Post!]!
   @aws_auth(cognito_groups: ["Bloggers", "Readers"])
}

type Mutation {
   addPost(id:ID!, title:String!):Post!
   @aws_auth(cognito_groups: ["Bloggers"])
}
...
```

请注意，如果您想默认使用特定的访问 grant-or-deny 策略，则可以省略该`@aws_auth`指令。通过控制台或以下 CLI 命令创建 GraphQL API 时，可以在用户池配置中指定 grant-or-deny 策略：

```
$ aws appsync --region us-west-2 create-graphql-api --authentication-
type AMAZON_COGNITO_USER_POOLS  --name userpoolstest --user-pool-config
 '{ "userPoolId":"test", "defaultEffect":"ALLOW", "awsRegion":"us-west-2"}'
```

使用其他授权模式

添加其他授权模式时，可以直接在 AWS AppSync GraphQL API 级别配置授权设置（即可以直接在`GraphqlApi`对象上配置的`authenticationType`字段），它充当架构上的默认设置。这意味着任何没有特定指令的类型都必须通过 API 级别授权设置。

在架构级别，您可以使用架构上的指令指定其他授权模式。您可以在架构中的各个字段上指定授权模式。例如，对于 API_KEY 授权，您将在架构对象类型定义/字段上使用 @aws_api_key。架构字段和对象类型定义支持以下指令：

- @aws_api_key - 指定字段是 API_KEY 授权的。
- @aws_iam - 指定字段是 AWS_IAM 授权的。
- @aws_oidc - 指定字段是 OPENID_CONNECT 授权的。
- @aws_cognito_user_pools - 指定字段是 AMAZON_COGNITO_USER_POOLS 授权的。
- @aws_lambda - 指定字段是 AWS_LAMBDA 授权的。

您不能将 @aws_auth 指令与其他授权模式一起使用。@aws_auth 仅适用于 AMAZON_COGNITO_USER_POOLS 授权的上下文，没有其他授权模式。但是，您可以使用 @aws_cognito_user_pools 指令代替 @aws_auth 指令，使用相同的参数。两者之间的主要区别在于您可以在任何字段和对象类型定义上指定 @aws_cognito_user_pools。

要了解其他授权模式如何工作以及如何在架构上指定它们，我们来看一下以下架构：

```
schema {
   query: Query
   mutation: Mutation
}

type Query {
   getPost(id: ID): Post
   getAllPosts(): [Post]
   @aws_api_key
}

type Mutation {
   addPost(
      id: ID!
      author: String!
      title: String!
      content: String!
      url: String!
   ): Post!
}

type Post @aws_api_key @aws_iam {
   id: ID!
```

```
   author: String
   title: String
   content: String
   url: String
   ups: Int!
   downs: Int!
   version: Int!
}
...
```

对于此架构，假设这AWS_IAM是 AWS AppSync GraphQL API 上的默认授权类型。这意味着使用 AWS_IAM 保护没有指令的字段。例如，Query 类型上的 getPost 字段就是这种情况。架构指令使您可以使用多种授权模式。例如，您可以在 AWS AppSync GraphQL API 上API_KEY配置为额外的授权模式，也可以使用@aws_api_key指令标记字段（例如，在本示例getAllPosts中）。指令在字段级别工作，因此您也需要授予 API_KEY 访问 Post 类型的权限。您可以通过使用指令标记 Post 类型中的每个字段，或使用 @aws_api_key 指令标记 Post 类型来执行此操作。

要进一步限制对 Post 类型中的字段的访问，可以对 Post 类型中的各个字段使用指令，如下所示。

例如，您可以将 restrictedContent 字段添加到 Post 类型并使用 @aws_iam 指令限制对它的访问。 AWS_IAM 经过身份验证的请求可以访问 restrictedContent，但是，API_KEY 请求将无法访问它。

```
type Post @aws_api_key @aws_iam{
   id: ID!
   author: String
   title: String
   content: String
   url: String
   ups: Int!
   downs: Int!
   version: Int!
   restrictedContent: String!
   @aws_iam
}
...
```

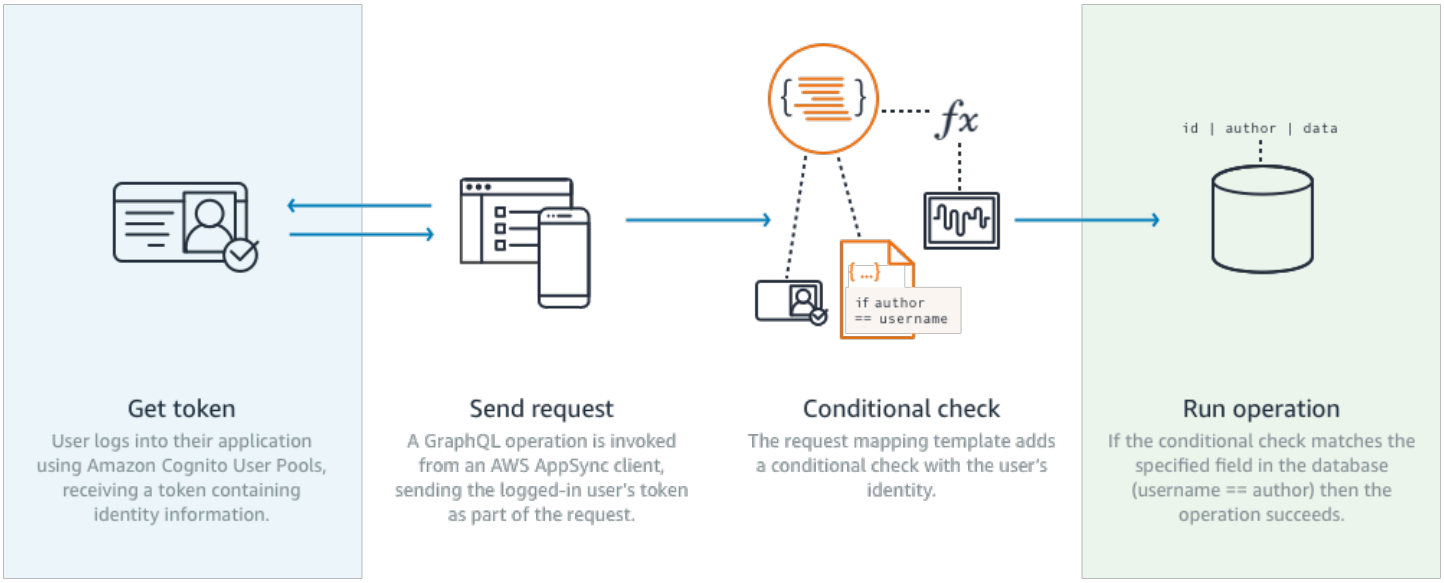
精细访问控制

上述信息演示了如何限制或授权对某些 GraphQL 字段的访问权限。如果您想根据某些条件（例如，根据进行调用的用户以及用户是否拥有数据）来设置对数据的访问控制，则可以使用解析器中的映射模板。您还可以执行更复杂的业务逻辑，筛选信息中介绍了相关内容。

本节介绍了如何使用 DynamoDB 解析器映射模板设置数据的访问控制。

在继续操作之前，如果您不熟悉中的 AWS AppSync映射模板，则可能需要查看 DynamoDB 的解析器映射模板参考和 DynamoDB 的解析器映射模板参考。

在使用 DynamoDB 的以下示例中，假设您使用前面的博客文章架构，并且仅允许创建博客的用户对其进行编辑。评估过程将是用户在应用程序中获得凭证（例如，使用 Amazon Cognito 用户池），然后将这些凭证作为 GraphQL 操作的一部分进行传递。然后，映射模板将替换条件语句中凭证（如用户名）的值，该值随后将与数据库中的值进行比较。



要添加此功能，请添加 GraphQL 字段 `editPost`，如下所示：

```
schema {
   query: Query
   mutation: Mutation
}

type Query {
   posts:[Post!]!
}
```

```
type Mutation {
   editPost(id:ID!, title:String, content:String):Post
   addPost(id:ID!, title:String!):Post!
}
...
```

editPost 的解析器映射模板（本节末尾的示例中所示）需要针对您的数据存储执行逻辑检查，以仅允许创建文章的用户对文章进行编辑。由于这是一个编辑操作，因此，它对应于 DynamoDB 中的 UpdateItem。您可以先执行条件检查，然后再执行此操作，同时使用传递的上下文进行用户身份验证。这些信息存储在一个 Identity 对象中，该对象具有以下值：

```
{
   "accountId" : "12321434323",
   "cognitoIdentityPoolId" : "",
   "cognitoIdentityId" : "",
   "sourceIP" : "",
   "caller" : "ThisistheprincipalARN",
   "username" : "username",
   "userArn" : "Sameasabove"
}
```

要在 DynamoDB UpdateItem 调用中使用该对象，您需要在表中存储用户身份信息以进行比较。首先，您的 addPost 变更需要存储创建者。其次，您的 editPost 变更需要先执行条件检查，然后才能更新。

下面是 addPost 的解析器代码示例，用于将用户身份存储为 Author 列：

```
import { util, Context } from '@aws-appsync/utils';
import { put } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const { id: postId, ...item } = ctx.args;
 return put({
  key: { postId },
  item: { ...item, Author: ctx.identity.username },
  condition: { postId: { attributeExists: false } },
 });
}

export const response = (ctx) => ctx.result;
```

请注意，Author 属性通过 Identity 对象填充，该对象来自应用程序。

最后，此处的示例介绍了 editPost 的解析器代码，此代码仅当请求来自创建博客文章的用户时，才更新该博客文章的内容。

```
import { util, Context } from '@aws-appsync/utils';
import { put } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const { id, ...item } = ctx.args;
 return put({
  key: { id },
  item,
  condition: { author: { contains: ctx.identity.username } },
 });
}

export const response = (ctx) => ctx.result;
```

此示例使用 PutItem 覆盖所有值，而不是使用 UpdateItem，但同样的概念适用于 condition 语句块。

筛选信息

有时，在成功写入或读取数据来源时，您可能无法控制来自数据来源的响应，但又不想向客户端发送不必要的信息。在这些情况下，您可以使用响应映射模板筛选信息。

例如，假设您的博客文章 DynamoDB 表上没有相应的索引（例如 Author 上的索引），您可以使用以下解析器：

```
import { util, Context } from '@aws-appsync/utils';
import { get } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 return get({ key: { ctx.args.id } });
}

export function response(ctx) {
 if (ctx.result.author === ctx.identity.username) {
  return ctx.result;
 }
 return null;
}
```

即使调用方不是创建博客文章的作者，请求处理程序也会获取该项目。为了防止它返回所有数据，响应处理程序会进行检查，以确保调用方与项目的作者匹配。如果调用方与此检查不匹配，则只返回 Null 响应。

数据来源访问

AWS AppSync 使用身份和访问管理 (IAM) 角色和访问策略与数据源通信。如果您使用的是现有角色，则需要添加信任策略才能代 AWS AppSync 入该角色。该信任关系将类似于下面内容：

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

需要将角色的访问策略缩小为仅具有对必要的最少资源集进行操作的权限，这一点非常重要。使用 AppSync 控制台创建数据源和创建角色时，系统会自动为您完成此操作。但是，当使用 IAM 控制台中的内置示例模板在控制台之外创建角色时，权限不会自动缩小到资源的范围，您应该在将应用程序移至生产环境之前执行此操作。 AWS AppSync

授权使用案例

在安全部分，您了解了保护 API 的不同授权模式，这一部分还介绍了精细授权机制，以便于您理解概念和流程。由于 AWS AppSync 允许您通过使用 GraphQL 解析器映射模板对数据执行逻辑完整操作，因此，您可以组合使用用户身份、条件和数据注入，以非常灵活的方式在读取或写入时保护数据。

如果您不熟悉如何编辑 AWS AppSync 解析器，请查看编程指南。

概述

在系统中授予数据访问权限的传统方法是通过访问控制矩阵，其中行（资源）和列（用户/角色）的交叉点就是授予的权限。

AWS AppSync 使用您自己的账户中的资源，并将身份（用户/角色）信息作为上下文对象嵌入到 GraphQL 请求和响应中，以便在解析器中使用该对象。也就是说，可以根据解析器的逻辑对读取或写入操作授予适当的权限。如果该逻辑处于资源级别，例如，仅某些指定的用户或组可以读取/写入特定的数据库行，您必须存储“授权元数据”。AWSAppSync 不会存储任何数据，因此，您必须将该授权元数据与资源一起存储，以便可以计算权限。授权元数据通常是 DynamoDB 表中的一个属性（列），例如 owner 或用户/组的列表。例如，可能有读取者和写入者属性。

从宏观角度而言，这就意味着如果您要从数据源中读取单个项目，即在解析器从数据源中读取内容之后在响应模板中执行一个条件 `#if () ... #end` 语句。此项检查通常会针对读取操作返回的授权元数据使用 `$context.identity` 中的用户或组值进行成员资格检查。如有多条记录，例如 `Scan` 或 `Query` 表返回的列表，您将使用类似的用户或组值将条件检查作为操作的一部分发送到数据源。

同样，在写入数据时，您将对操作（如 `PutItem` 或 `UpdateItem`）应用条件语句，以便了解进行变更的用户或组是否拥有权限。在许多情况下，条件将使用 `$context.identity` 中的值与资源的授权元数据进行比较。对于请求和响应模板，您还可使用客户端的自定义标头进行验证检查。

读取数据

如上所述，执行检查的授权元数据必须随资源存储，或传递到 GraphQL 请求中（身份、标头等）。为了说明这一点，假设您有如下 DynamoDB 表：

ID	Data	PeopleCanAccess	GroupsCanAccess	Owner
123	{my: data,...}	[Mary, Joe]	[Admins, Editors]	Nadia

主键是 `id`，要访问的数据是 `Data`。其他列是您可以执行的授权检查示例。`Owner` 是 `String`，而 `PeopleCanAccess` 和 `GroupsCanAccess` 是 `String Sets`，如 DynamoDB 解析器映射模板参考中所述。

解析器映射模板概述中的示意图展示了响应模板中不仅包含上下文对象，还包含数据源的结果。对于个别项目的 GraphQL 查询，您可以使用响应模板检查是否允许用户查看这些结果，或返回授权错误消息。这种方法有时称为”授权筛选“。对于 GraphQL 查询返回列表，更高效的方式是使用 Scan 或 Query 针对请求模板执行检查，只在满足授权条件的情况下返回数据。实施方法是：

1. GetItem - 针对个别记录进行授权检查。使用 `#if() ... #end` 语句实现。
2. Scan/Query 操作 - 授权检查是 `"filter":{"expression":...}` 语句。常用的检查方式是等式 (`attribute = :input`) 或者检查某个值是否在列表中 (`contains(attribute, :input)`)。

在 #2 中，两条语句中的 attribute 表示表中记录的列名，例如上例中的 Owner。您可以借助 # 符号并使用 "expressionNames":{...} 设置别名，但这不是必需的。:input 可引用与数据库属性进行比较的值，该属性在 "expressionValues":{...} 中定义。您将在下文中看到这些示例。

使用案例：所有者可以读取

以上表为例，对于一个读取操作 (Owner == Nadia)，如果您希望只在 GetItem 的情况下返回数据，您的模板将如下所示：

```
#if($context.result["Owner"] == $context.identity.username)
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

这里要提醒您几件事，在接下来的各节也会用到。首先，检查使用 $context.identity.username，如果使用 Amazon Cognito 用户池，这是友好的用户注册名称；如果使用 IAM，这是用户身份（包括 Amazon Cognito 联合身份）。还可以为所有者存储其他值，例如唯一的“Amazon Cognito 身份”值，这在从多个位置进行联合登录时是非常有用的，您应该查看解析器映射模板上下文参考中提供的选项。

第二，利用 $util.unauthorized() 进行响应的条件 else 检查完全是可选的，但作为最佳实践，建议您在设计 GraphQL API 时使用。

使用案例：硬编码特定的访问权限

```
// This checks if the user is part of the Admin group and makes the call
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #if($group == "Admin")
        #set($inCognitoGroup = true)
    #end
#end
#if($inCognitoGroup)
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "attributeValues" : {
        "owner" : $util.dynamodb.toDynamoDBJson($context.identity.username)
        #foreach( $entry in $context.arguments.entrySet() )
```

```
            ,"${entry.key}" : $util.dynamodb.toDynamoDBJson($entry.value)
        #end
    }
}
#else
    $utils.unauthorized()
#end
```

使用案例：筛选结果列表

在上一示例中，您能够直接针对 $context.result 执行检查，因为它只会返回一个项目；但有些操作（如扫描）将在 $context.result.items 中返回多个项目，您需要执行授权筛选，仅返回允许用户看到的结果。假设 Owner 字段这次在记录上设置了 Amazon Cognito IdentityID，则可以使用以下响应映射模板进行筛选，以仅显示用户拥有的记录：

```
#set($myResults = [])
#foreach($item in $context.result.items)
    ##For userpools use $context.identity.username instead
    #if($item.Owner == $context.identity.cognitoIdentityId)
        #set($added = $myResults.add($item))
    #end
#end
$utils.toJson($myResults)
```

使用案例：多人可以读取

另一种常用的授权选项是允许一组人员读取数据。在以下示例中，只有在运行 GraphQL 查询的用户属于 "filter":{"expression":...} 集的情况下 PeopleCanAccess 才会返回扫描表的值。

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "limit": #if(${context.arguments.count}) $util.toJson($context.arguments.count)
 #else 20 #end,
    "nextToken": #if(${context.arguments.nextToken})
 $util.toJson($context.arguments.nextToken) #else null #end,
    "filter":{
        "expression": "contains(#peopleCanAccess, :value)",
        "expressionNames": {
                "#peopleCanAccess": "peopleCanAccess"
        },
        "expressionValues": {
```

```
                ":value": $util.dynamodb.toDynamoDBJson($context.identity.username)
        }
    }
}
```

使用案例：组可以读取

与上一使用案例类似，可能只有一个或多个组中的人员才有权读取数据库中的某些项目。使用 `"expression": "contains()"` 操作具有类似效果，但在集的成员资格中，需考虑用户所在的所有组之间是逻辑或的关系。在本例中，我们构建一个 `$expression` 语句，包含用户所在的每个组，然后传递给筛选器：

```
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
    #set( $val = {})
    #set( $test = $val.put("S", $group))
    #set( $values = $expressionValues.put(":var$foreach.count", $val))
    #if ( $foreach.hasNext )
    #set( $expression = "${expression} OR" )
    #end
#end
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "limit": #if(${context.arguments.count}) $util.toJson($context.arguments.count)
 #else 20 #end,
    "nextToken": #if(${context.arguments.nextToken})
 $util.toJson($context.arguments.nextToken) #else null #end,
    "filter":{
        "expression": "$expression",
        "expressionValues": $utils.toJson($expressionValues)
    }
}
```

写入数据

将数据写入变更始终是通过请求映射模板进行控制的。对于 DynamoDB 数据源而言，关键是要使用适当的 `"condition":{"expression"...}"`，针对表中的授权元数据执行验证。安全部分提供了一个示例，您可以使用它检查表中的 Author 字段。本节中将探索更多使用案例。

使用案例：多个所有者

使用之前的表示意图示例，假设 PeopleCanAccess 列表

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "update" : {
        "expression" : "SET meta = :meta",
        "expressionValues": {
            ":meta" : $util.dynamodb.toDynamoDBJson($ctx.args.meta)
        }
    },
    "condition" : {
        "expression"       : "contains(Owner,:expectedOwner)",
        "expressionValues" : {
            ":expectedOwner" :
 $util.dynamodb.toDynamoDBJson($context.identity.username)
        }
    }
}
```

使用案例：组可以创建新记录

```
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
    #set( $val = {})
    #set( $test = $val.put("S", $group))
    #set( $values = $expressionValues.put(":var$foreach.count", $val))
    #if ( $foreach.hasNext )
    #set( $expression = "${expression} OR" )
    #end
#end
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
```

```
        ## If your table's hash key is not named 'id', update it here. **
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
        ## If your table has a sort key, add it as an item here. **
    },
    "attributeValues" : {
        ## Add an item for each field you would like to store to Amazon DynamoDB. **
        "title" : $util.dynamodb.toDynamoDBJson($ctx.args.title),
        "content": $util.dynamodb.toDynamoDBJson($ctx.args.content),
        "owner": $util.dynamodb.toDynamoDBJson($context.identity.username)
    },
    "condition" : {
        "expression": $util.toJson("attribute_not_exists(id) AND $expression"),
        "expressionValues": $utils.toJson($expressionValues)
    }
}
```

使用案例：组可以更新现有记录

```
#set($expression = "")
#set($expressionValues = {})
#foreach($group in $context.identity.claims.get("cognito:groups"))
    #set( $expression = "${expression} contains(groupsCanAccess, :var
$foreach.count )" )
    #set( $val = {})
    #set( $test = $val.put("S", $group))
    #set( $values = $expressionValues.put(":var$foreach.count", $val))
    #if ( $foreach.hasNext )
    #set( $expression = "${expression} OR" )
    #end
#end
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "update":{
                "expression" : "SET title = :title, content = :content",
        "expressionValues": {
            ":title" : $util.dynamodb.toDynamoDBJson($ctx.args.title),
            ":content" : $util.dynamodb.toDynamoDBJson($ctx.args.content)
        }
    },
```

```
    "condition" : {
        "expression": $util.toJson($expression),
        "expressionValues": $utils.toJson($expressionValues)
    }
}
```

公有记录和私有记录

您还可利用条件筛选器选择将数据标记为私有、公开，或进行其他布尔检查。还可在响应模板中进行组合，作为授权筛选的一部分。使用此检查是一种临时隐藏数据，使之不可见的好方法，而无需尝试控制组成员资格。

例如，假设您为 DynamoDB 表中的每个项目添加了一个称为 `public` 的属性，其值可为 `yes` 或 `no`。以下响应模板可用于 `GetItem` 调用，只在用户属于具有权限的组，且数据标为公共时才会显示数据：

```
#set($permissions = $context.result.GroupsCanAccess)
#set($claimPermissions = $context.identity.claims.get("cognito:groups"))

#foreach($per in $permissions)
    #foreach($cgroups in $claimPermissions)
        #if($cgroups == $per)
            #set($hasPermission = true)
        #end
    #end
#end

#if($hasPermission && $context.result.public == 'yes')
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

以上代码还可使用逻辑或 (`||`) 允许有权限的人员读取记录，或允许读取公共记录：

```
#if($hasPermission || $context.result.public == 'yes')
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

通常，在执行授权检查时，您会发现标准运算符 `==`、`!=`、`&&` 和 `||` 很有用。

实时数据

客户端进行订阅时，您可使用本文档之前介绍的方法，针对 GraphQL 订阅应用精细访问控制。您可将解析器附加到订阅字段，然后您就可以查询数据源的数据，并在请求或响应映射模板中执行条件逻辑。您也可以将其他数据返回到客户端，例如订阅的最初结果，条件是数据结构与 GraphQL 订阅中返回类型的结构相匹配。

使用案例：用户只能订阅特定对话

GraphQL 订阅实时数据的一种常见使用案例是构建消息收发或私人聊天应用程序。如果创建具有多个用户的聊天应用程序，可以在两人或多人之间发生对话。这些对话可以用私有或公共“房间”进行组织。因此，您可能只希望授权用户订阅他们有权访问的对话（可能是一对一或小组对话）。出于演示目的，以下示例展示一个简单的使用案例：一名用户向另一名用户发送私人消息。该设置具有两个 Amazon DynamoDB 表：

- 消息表：（主键）`toUser`，（排序键）`id`
- 权限表：（主键）`username`

消息表存储通过 GraphQL 变更发送的实际消息。权限表用于 GraphQL 订阅在客户端连接时检查授权。以下示例假设您使用以下 GraphQL 架构：

```
input CreateUserPermissionsInput {
    user: String!
    isAuthorizedForSubscriptions: Boolean
}

type Message {
    id: ID
    toUser: String
    fromUser: String
    content: String
}

type MessageConnection {
    items: [Message]
    nextToken: String
}

type Mutation {
    sendMessage(toUser: String!, content: String!): Message
```

```
    createUserPermissions(input: CreateUserPermissionsInput!): UserPermissions
    updateUserPermissions(input: UpdateUserPermissionInput!): UserPermissions
}

type Query {
    getMyMessages(first: Int, after: String): MessageConnection
    getUserPermissions(user: String!): UserPermissions
}

type Subscription {
    newMessage(toUser: String!): Message
        @aws_subscribe(mutations: ["sendMessage"])
}

input UpdateUserPermissionInput {
    user: String!
    isAuthorizedForSubscriptions: Boolean
}

type UserPermissions {
    user: String
    isAuthorizedForSubscriptions: Boolean
}

schema {
    query: Query
    mutation: Mutation
    subscription: Subscription
}
```

下面未介绍某些标准操作（例如 `createUserPermissions()`）以说明订阅解析器，但它们是 DynamoDB 解析器的标准实施。我们关注的是利用解析器进行订阅授权的流程。要从一个用户向另一个用户发送消息，可将解析器附加到 `sendMessage()` 字段，并利用以下请求模板选择消息表数据源：

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "toUser" : $util.dynamodb.toDynamoDBJson($ctx.args.toUser),
        "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
    },
    "attributeValues" : {
```

```
        "fromUser" : $util.dynamodb.toDynamoDBJson($context.identity.username),
        "content" : $util.dynamodb.toDynamoDBJson($ctx.args.content),
    }
}
```

在此示例中，我们使用的是 $context.identity.username。这会返回 AWS Identity and Access Management 或 Amazon Cognito 用户的用户信息。响应模板只是简单地传递 $util.toJson($ctx.result)。保存并返回架构页面。然后为 newMessage() 订阅附加解析器，使用权限表作为数据源，使用以下请求映射模板：

```
{
    "version": "2018-05-29",
    "operation": "GetItem",
    "key": {
        "username": $util.dynamodb.toDynamoDBJson($ctx.identity.username),
    },
}
```

然后利用权限表中的数据，通过以下响应映射模板执行授权检查：

```
#if(! ${context.result})
    $utils.unauthorized()
#elseif(${context.identity.username} != ${context.arguments.toUser})
    $utils.unauthorized()
#elseif(! ${context.result.isAuthorizedForSubscriptions})
    $utils.unauthorized()
#else
##User is authorized, but we return null to continue
    null
#end
```

在本例中，您进行三次授权检查。第一次确保返回结果。第二次确保用户未订阅面向其他人的消息。通过检查存储为 BOOL 的 isAuthorizedForSubscriptions DynamoDB 属性，第三次检查确保允许用户订阅任何字段。

要进行测试，您可以使用 Amazon Cognito 用户池和名为“Nadia”的用户登录到 AWS AppSync 控制台，然后运行以下 GraphQL 订阅：

```
subscription AuthorizedSubscription {
    newMessage(toUser: "Nadia") {
```

```
        id
        toUser
        fromUser
        content
    }
}
```

如果权限 表中有一条 username 的记录，其键属性为 Nadia，isAuthorizedForSubscriptions 设置为 true，您将看到成功响应。如果您在以上 username 查询中尝试其他 newMessage()，将返回错误。

使用 AWS WAF 保护 API

AWS WAF 是一个 Web 应用程序防火墙，可帮助保护 Web 应用程序和 API 免受攻击。通过使用该功能，您可以配置一组规则（称为 Web 访问控制列表 (Web ACL)），这些规则根据您定义的可自定义 Web 安全规则和条件允许、阻止或监控（统计）Web 请求。在将 AWS AppSync API 与 AWS WAF 集成时，您可以更好地控制和了解 API 接受的 HTTP 流量。要了解 AWS WAF 的更多信息，请参阅 AWS WAF Developer Guide 中的 How AWS WAF Works。

您可以使用 AWS WAF 保护 AppSync API 以免受到常见的 Web 攻击，例如 SQL 注入和跨站脚本 (XSS) 攻击。这些威胁可能会影响 API 的可用性和性能、损害安全性或消耗过多的资源。例如，您可以创建规则以允许或阻止以下请求：来自指定 IP 地址范围的请求；来自 CIDR 块的请求；源自特定国家/地区或区域的请求；包含恶意 SQL 代码的请求；或者包含恶意脚本的请求。

您还可以创建与 HTTP 标头、方法、查询字符串、URI 和请求正文中的指定字符串或正则表达式模式匹配的规则（限制为前 8 KB）。此外，您可以创建规则来阻止来自特定用户代理、恶意机器人和内容抓取程序的攻击。例如，您可以使用基于速率的规则来指定每个客户端 IP 在尾随的、不断更新的 5 分钟期间内允许的 Web 请求数。

要了解支持的规则类型和其他 AWS WAF 功能的更多信息，请参阅 AWS WAF Developer Guide 和 AWS WAF API Reference。

> **Important**
>
> AWS WAF 是抵御 Web 漏洞攻击的第一道防线。如果在 API 上启用了 AWS WAF，将在其他访问控制功能（例如 API 密钥授权、IAM 策略、OIDC 令牌和 Amazon Cognito 用户池）之前评估 AWS WAF 规则。

将 AppSync API 与 AWS WAF 集成

您可以使用 AWS Management Console、AWS CLI、AWS CloudFormation 或任何其他兼容客户端，将 AppSync API 与 AWS WAF 集成在一起。

将 AWS AppSync API 与 AWS WAF 集成

1. 创建一个 AWS WAF Web ACL。有关使用 AWS WAF 控制台的详细步骤，请参阅 Creating a web ACL。
2. 定义 Web ACL 的规则。在创建 Web ACL 的过程中定义了一个或多个规则。有关如何构建规则的信息，请参阅 AWS WAF rules。有关您可以为 AWS AppSync API 定义的有用规则的示例，请参阅为 Web ACL 创建规则。
3. 将 Web ACL 与一个 AWS AppSync API 相关联。您可以在 AWS WAF 控制台或 AppSync 控制台中执行该步骤。
   - 要在 AWS WAF 控制台中将 Web ACL 与一个 AWS AppSync API 关联，请按照 AWS WAF Developer Guide 的 Associating or disassociating a Web ACL with an AWS resource 中的说明进行操作。
   - 在 AWS AppSync 控制台中将 Web ACL 与 AWS AppSync API 关联
     a. 登录到 AWS Management Console，然后打开 AppSync 控制台。
     b. 选择您希望与 Web ACL 关联的 API。
     c. 在导航窗格中，选择 Settings (设置)。
     d. 在 Web 应用程序防火墙部分中，开启启用 AWS WAF。
     e. 在 Web ACL 下拉列表中，选择要与您的 API 关联的 Web ACL 名称。
     f. 选择保存以将该 Web ACL 与您的 API 关联。

> **Note**
>
> 在 AWS WAF 控制台中创建 Web ACL 后，新的 Web ACL 可能需要几分钟才能使用。如果您在 Web 应用程序防火墙菜单中没有看到新创建的 Web ACL，请等待几分钟，然后重试将 Web ACL 与您的 API 关联的步骤。

> **Note**
>
> AWS WAF 集成仅支持实时终端节点的 `Subscription registration message` 事件。对于 AWS WAF 阻止的任何 `Subscription registration message`，AWS AppSync 使用一条错误消息进行响应，而不是 `start_ack` 消息。

在将 Web ACL 与 AWS AppSync API 关联后，您将使用 AWS WAF API 管理 Web ACL。您不需要将 Web ACL 与 AWS AppSync API 重新关联，除非您希望将 AWS AppSync API 与不同的 Web ACL 关联。

为 Web ACL 创建规则

规则定义如何检查 Web 请求以及在 Web 请求符合检查条件时执行的操作。规则在 AWS WAF 中不是单独存在的。您可以在规则组或定义规则的 Web ACL 中按名称访问规则。有关更多信息，请参阅 AWS WAF rules。以下示例说明了如何定义和关联用于保护 AppSync API 的规则。

Example 用于限制请求正文大小的 Web ACL 规则

以下是限制请求正文大小的规则示例。在 AWS WAF 控制台中创建 Web ACL 时，将在规则 JSON 编辑器中输入该规则。

```
{
    "Name": "BodySizeRule",
    "Priority": 1,
    "Action": {
        "Block": {}
    },
    "Statement": {
        "SizeConstraintStatement": {
            "ComparisonOperator": "GE",
            "FieldToMatch": {
                "Body": {}
            },
            "Size": 1024,
            "TextTransformations": [
                {
                    "Priority": 0,
                    "Type": "NONE"
                }
             ]
```

```
          }
       },
       "VisibilityConfig": {
           "CloudWatchMetricsEnabled": true,
           "MetricName": "BodySizeRule",
           "SampledRequestsEnabled": true
        }
}
```

在使用上一示例规则创建 Web ACL 后，您必须将其与 AppSync API 相关联。作为使用 AWS Management Console 的替代方法，您可以在 AWS CLI 中运行以下命令以执行该步骤。

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

可能需要几分钟才能传播更改，但在运行该命令后，包含的正文大于 1024 字节的请求将被 AWS AppSync 拒绝。

> **Note**
>
> 在 AWS WAF 控制台中创建新的 Web ACL 后，可能需要几分钟才能将 Web ACL 与 API 关联。如果您运行 CLI 命令并出现 `WAFUnavailableEntityException` 错误，请等待几分钟，然后再次尝试运行该命令。

Example 限制来自单个 IP 地址的请求的 Web ACL 规则

以下是一个规则示例，它将 AppSync API 限制为来自单个 IP 地址的 100 个请求。在 AWS WAF 控制台中创建具有基于速率的规则的 Web ACL 时，将在规则 JSON 编辑器中输入该规则。

```
{
  "Name": "Throttle",
  "Priority": 0,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "Throttle"
  },
  "Statement": {
```

```
    "RateBasedStatement": {
      "Limit": 100,
      "AggregateKeyType": "IP"
    }
  }
}
```

在使用上一示例规则创建 Web ACL 后，您必须将其与 AppSync API 相关联。您可以在 AWS CLI 中运行以下命令以执行该步骤。

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

Example 禁止对 API 运行 GraphQL __schema 自省查询的 Web ACL 规则

以下是一个禁止对 API 运行 GraphQL __schema 自省查询的规则示例。将阻止任何包含字符串“__schema”的 HTTP 正文。在 AWS WAF 控制台中创建 Web ACL 时，将在规则 JSON 编辑器中输入该规则。

```
{
  "Name": "BodyRule",
  "Priority": 5,
  "Action": {
    "Block": {}
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "BodyRule"
  },
  "Statement": {
    "ByteMatchStatement": {
      "FieldToMatch": {
        "Body": {}
      },
      "PositionalConstraint": "CONTAINS",
      "SearchString": "__schema",
      "TextTransformations": [
        {
          "Type": "NONE",
          "Priority": 0
        }
      ]
```

```
    }
  }
}
```

在使用上一示例规则创建 Web ACL 后，您必须将其与 AppSync API 相关联。您可以在 AWS CLI 中运行以下命令以执行该步骤。

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

安全性 AWS AppSync

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 AWS 云中运行 AWS 服务的基础架构。 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。要了解适用的合规计划 AWS AppSync，请参阅按合规计划划分的范围内的AWSAWS 服务按合规计划。
- 云端安全-您的责任由您使用的 AWS 服务决定。您还需要对其他因素负责，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。

本文档可帮助您了解在使用时如何应用分担责任模型 AWS AppSync。以下主题向您介绍如何进行配置 AWS AppSync 以满足您的安全和合规性目标。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 AWS AppSync 资源。

主题

- 中的数据保护 AWS AppSync
- 合规性验证 AWS AppSync
- 中的基础设施安全 AWS AppSync
- 韧性在 AWS AppSync
- 的身份和访问管理 AWS AppSync
- 使用记录 AWS AppSync API 调用 AWS CloudTrail
- 以下方面的安全最佳实践 AWS AppSync

中的数据保护 AWS AppSync

分 AWS 担责任模型适用于中的数据保护 AWS AppSync。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设施上的内容的控制。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 安全性博客 上的 AWS 责任共担模式和 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。我们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)。
- 使用 SSL/TLS 与资源通信。 AWS 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。
- 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 140-2 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 140-2 版》。

我们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）。这包括您使用控制台、API AWS AppSync 或 SDK 或以其他 AWS 服务 方式使用控制台 AWS CLI、API 或 AWS SDK 的情况。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网址，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。

动态加密

AWS AppSync与所有 AWS 服务一样，在使用 AWS 已发布的 API 和 SDK 时，使用 TLS1.2 及更高版本进行通信。

AWS AppSync 与 Amazon DynamoDB 等其他 AWS 服务一起使用可确保传输过程中的加密：除非另有说明，否则 AWS 所有服务都使用 TLS 1.2 及更高版本相互通信。对于使用 Amazon EC2 或的解析人员 CloudFront，您有责任验证 TLS (HTTPS) 是否已配置且安全。有关在 Amazon EC2 中配置 HTTPS 的信息，请参阅《Amazon EC2 用户指南》中的在 Amazon Linux 2 上配置 SSL/TLS。有关在上配置 HTTPS 的信息 CloudFront，请参阅 CloudFront 用户指南中的亚马逊 CloudFront中的 HTTPS。

合规性验证 AWS AppSync

AWS AppSync 作为多个合规计划的一部分，第三方审计师对安全性和 AWS 合规性进行评估。 AWS AppSync 符合 SOC、PCI、HIPAA/HIPAA BAA、IRAP、C5、ENS High、OSPAR 和 HITRUST CSF 计划。

要了解是否属于特定合规计划的范围，请参阅AWS 服务 “按合规计划划分的范围” ”，然后选择您感兴趣的合规计划。 AWS 服务 有关一般信息，请参阅AWS 合规计划AWS。

您可以使用下载第三方审计报告 AWS Artifact。有关更多信息，请参阅中的 “下载报告” 中的 “ AWS Artifact。

您在使用 AWS 服务 时的合规责任取决于您的数据的敏感性、贵公司的合规目标以及适用的法律和法规。 AWS 提供了以下资源来帮助实现合规性：

- 安全与合规性快速入门指南 — 这些部署指南讨论了架构注意事项，并提供了在这些基础上 AWS 部署以安全性和合规性为重点的基准环境的步骤。
- 在 A@@ mazon Web Services 上构建 HIPAA 安全与合规架构 — 本白皮书描述了各公司如何使用 AWS 来创建符合 HIPAA 资格的应用程序。

> **Note**
>
> 并非所有 AWS 服务 人都符合 HIPAA 资格。有关更多信息，请参阅符合 HIPAA 要求的服务参考。

- AWS 合规资源AWS — 此工作簿和指南集可能适用于您所在的行业和所在地区。
- AWS 客户合规指南 — 从合规角度了解责任共担模式。这些指南总结了保护的最佳实践， AWS 服务并将指南映射到跨多个框架（包括美国国家标准与技术研究院 (NIST)、支付卡行业安全标准委员会 (PCI) 和国际标准化组织 (ISO)）的安全控制。
- 使用AWS Config 开发人员指南中的规则评估资源 — 该 AWS Config 服务评估您的资源配置在多大程度上符合内部实践、行业指导方针和法规。
- AWS Security Hub— 这 AWS 服务 提供了您内部安全状态的全面视图 AWS。Security Hub 通过安全控件评估您的 AWS 资源并检查其是否符合安全行业标准和最佳实践。有关受支持服务及控件的列表，请参阅 Security Hub 控件参考。
- Amazon GuardDuty — 它通过监控您的 AWS 账户环境中是否存在可疑和恶意活动，来 AWS 服务检测您的工作负载、容器和数据面临的潜在威胁。 GuardDuty 通过满足某些合规性框架规定的入侵检测要求，可以帮助您满足各种合规性要求，例如 PCI DSS。
- AWS Audit Manager— 这 AWS 服务 可以帮助您持续审计 AWS 使用情况，从而简化风险管理以及对法规和行业标准的合规性。

中的基础设施安全 AWS AppSync

作为一项托管服务 AWS AppSync ，受 AWS 全球网络安全的保护。有关 AWS 安全服务以及如何 AWS 保护基础设施的信息，请参阅AWS 云安全。要使用基础设施安全的最佳实践来设计您的 AWS 环境，请参阅 S AWS ecurity Pillar Well-Architected Fram ework 中的基础设施保护。

您可以使用 AWS 已发布的 API 调用 AWS AppSync 通过网络进行访问。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。

此外，必须使用访问密钥 ID 和与 IAM 委托人关联的秘密访问密钥来对请求进行签名。或者，您可以使用 AWS Security Token Service（AWS STS）生成临时安全凭证来对请求进行签名。

韧性在 AWS AppSync

AWS 全球基础设施是围绕 AWS 区域和可用区构建的。 AWS 区域提供多个物理隔离和隔离的可用区，这些可用区通过低延迟、高吞吐量和高度冗余的网络相连。利用可用区，您可以设计和操作在可用区之间无中断地自动实现失效转移的应用程序和数据库。与传统的单个或多个数据中心基础设施相比，可用区具有更高的可用性、容错性和可扩展性。

有关 AWS 区域和可用区的更多信息，请参阅AWS 全球基础设施。

除了 AWS 全球基础架构外，还 AWS AppSync 允许使用模板定义大多数资源；有关使用 AWS CloudFormation 模板声明 AWS AppSync 资源的示例，请参阅 AWS 博客和AWS CloudFormation 用户指南上的 Pip AWS CloudFormation AWS AppSync eline Resolvers 实用用例。

的身份和访问管理 AWS AppSync

AWS Identity and Access Management (IAM) AWS 服务 可帮助管理员安全地控制对 AWS 资源的访问权限。IAM 管理员控制谁可以进行身份验证（登录）和授权（有权限）使用 AWS AppSync 资源。您可以使用 IAM AWS 服务 ，无需支付额外费用。

主题

- 受众

- 使用身份进行身份验证
- 使用策略管理访问
- 如何 AWS AppSync 与 IAM 配合使用
- 适用于 AWS AppSync 的基于身份的策略
- 对 AWS AppSync 身份和访问进行故障排除

受众

您的使用方式 AWS Identity and Access Management (IAM) 会有所不同，具体取决于您所做的工作 AWS AppSync。

服务用户-如果您使用 AWS AppSync 服务完成工作，则管理员会为您提供所需的凭证和权限。当你使用更多 AWS AppSync 功能来完成工作时，你可能需要额外的权限。了解如何管理访问权限有助于您向管理员请求适合的权限。如果您无法访问中的功能 AWS AppSync，请参阅对 AWS AppSync 身份和访问进行故障排除。

服务管理员-如果您负责公司的 AWS AppSync 资源，则可能拥有完全访问权限 AWS AppSync。您的工作是确定您的服务用户应访问哪些 AWS AppSync 功能和资源。然后，您必须向 IAM 管理员提交请求以更改服务用户的权限。请查看该页面上的信息以了解 IAM 的基本概念。要详细了解您的公司如何将 IAM 与配合使用 AWS AppSync，请参阅如何 AWS AppSync 与 IAM 配合使用。

IAM 管理员 — 如果您是 IAM 管理员，则可能需要详细了解如何编写策略来管理访问权限 AWS AppSync。要查看您可以在 IAM 中使用的 AWS AppSync 基于身份的策略示例，请参阅。适用于 AWS AppSync 的基于身份的策略

使用身份进行身份验证

身份验证是您 AWS 使用身份凭证登录的方式。您必须以 IAM 用户身份或通过担 AWS 账户根用户任 IAM 角色进行身份验证（登录 AWS）。

您可以使用通过身份源提供的凭据以 AWS 联合身份登录。 AWS IAM Identity Center （ IAM Identity Center ）用户、贵公司的单点登录身份验证以及您的 Google 或 Facebook 凭据就是联合身份的示例。当您以联合身份登录时，您的管理员以前使用 IAM 角色设置了身份联合验证。当你使用联合访问 AWS 时，你就是在间接扮演一个角色。

根据您的用户类型，您可以登录 AWS Management Console 或 AWS 访问门户。有关登录的更多信息 AWS，请参阅《AWS 登录 用户指南》中的如何登录到您 AWS 账户的。

如果您 AWS 以编程方式访问，则会 AWS 提供软件开发套件 (SDK) 和命令行接口 (CLI)，以便使用您的凭据对请求进行加密签名。如果您不使用 AWS 工具，则必须自己签署请求。有关使用推荐的方法自行签署请求的更多信息，请参阅 IAM 用户指南中的签署 AWS API 请求。

无论使用何种身份验证方法，您可能需要提供其他安全信息。例如， AWS 建议您使用多重身份验证 (MFA) 来提高账户的安全性。要了解更多信息，请参阅《AWS IAM Identity Center 用户指南》中的多重身份验证和《IAM 用户指南》中的在 AWS中使用多重身份验证（MFA）。

AWS 账户 root 用户

创建时 AWS 账户，首先要有一个登录身份，该身份可以完全访问账户中的所有资源 AWS 服务 和资源。此身份被称为 AWS 账户 root 用户，使用您创建账户时使用的电子邮件地址和密码登录即可访问该身份。强烈建议您不要使用根用户执行日常任务。保护好根用户凭证，并使用这些凭证来执行仅根用户可以执行的任务。有关要求您以根用户身份登录的任务的完整列表，请参阅《IAM 用户指南》中的需要根用户凭证的任务。

联合身份

作为最佳实践，要求人类用户（包括需要管理员访问权限的用户）使用与身份提供商的联合身份验证 AWS 服务 通过临时证书进行访问。

联合身份是指您的企业用户目录、Web 身份提供商、Identity C enter 目录中的用户，或者任何使用 AWS 服务 通过身份源提供的凭据进行访问的用户。 AWS Directory Service当联合身份访问时 AWS 账户，他们将扮演角色，角色提供临时证书。

要集中管理访问权限，建议您使用 AWS IAM Identity Center。您可以在 IAM Identity Center 中创建用户和群组，也可以连接并同步到您自己的身份源中的一组用户和群组，以便在您的所有 AWS 账户 和应用程序中使用。有关 IAM Identity Center 的信息，请参阅《AWS IAM Identity Center 用户指南》中的什么是 IAM Identity Center？。

IAM 用户和群组

I AM 用户是您 AWS 账户 内部对个人或应用程序具有特定权限的身份。在可能的情况下，我们建议使用临时凭证，而不是创建具有长期凭证（如密码和访问密钥）的 IAM 用户。但是，如果您有一些特定的使用场景需要长期凭证以及 IAM 用户，建议您轮换访问密钥。有关更多信息，请参阅《IAM 用户指南》中的对于需要长期凭证的使用场景定期轮换访问密钥。

IAM 组是一个指定一组 IAM 用户的身份。您不能使用组的身份登录。您可以使用组来一次性为多个用户指定权限。如果有大量用户，使用组可以更轻松地管理用户权限。例如，您可能具有一个名为 IAMAdmins 的组，并为该组授予权限以管理 IAM 资源。

用户与角色不同。用户唯一地与某个人员或应用程序关联，而角色旨在让需要它的任何人代入。用户具有永久的长期凭证，而角色提供临时凭证。要了解更多信息，请参阅《IAM 用户指南》中的何时创建 IAM 用户（而不是角色）。

IAM 角色

I AM 角色是您内部具有特定权限 AWS 账户 的身份。它类似于 IAM 用户，但与特定人员不关联。您可以 AWS Management Console 通过切换角色在中临时担任 IAM 角色。您可以通过调用 AWS CLI 或 AWS API 操作或使用自定义 URL 来代入角色。有关使用角色的方法的更多信息，请参阅《IAM 用户指南》中的 使用 IAM 角色。

具有临时凭证的 IAM 角色在以下情况下很有用：

- 联合用户访问 – 要向联合身份分配权限，请创建角色并为角色定义权限。当联合身份进行身份验证时，该身份将与角色相关联并被授予由此角色定义的权限。有关联合身份验证的角色的信息，请参阅《IAM 用户指南》中的 为第三方身份提供商创建角色。如果您使用 IAM Identity Center，则需要配置权限集。为控制您的身份在进行身份验证后可以访问的内容，IAM Identity Center 将权限集与 IAM 中的角色相关联。有关权限集的信息，请参阅《AWS IAM Identity Center 用户指南》中的权限集。
- 临时 IAM 用户权限 – IAM 用户可代入 IAM 用户或角色，以暂时获得针对特定任务的不同权限。
- 跨账户存取 – 您可以使用 IAM 角色以允许不同账户中的某个人（可信主体）访问您的账户中的资源。角色是授予跨账户访问权限的主要方式。但是，对于某些资源 AWS 服务，您可以将策略直接附加到资源（而不是使用角色作为代理）。要了解用于跨账户访问的角色和基于资源的策略之间的差别，请参阅《IAM 用户指南》中的 IAM 角色与基于资源的策略有何不同。
- 跨服务访问 — 有些 AWS 服务 使用其他 AWS 服务服务中的功能。例如，当您在某个服务中进行调用时，该服务通常会在 Amazon EC2 中运行应用程序或在 Amazon S3 中存储对象。服务可能会使用发出调用的主体的权限、使用服务角色或使用服务相关角色来执行此操作。
  - 转发访问会话 (FAS) — 当您使用 IAM 用户或角色在中执行操作时 AWS，您被视为委托人。使用某些服务时，您可能会执行一个操作，然后此操作在其他服务中启动另一个操作。FAS 使用调用委托人的权限以及 AWS 服务 向下游服务发出请求的请求。 AWS 服务只有当服务收到需要与其他 AWS 服务 或资源交互才能完成的请求时，才会发出 FAS 请求。在这种情况下，您必须具有执行这两个操作的权限。有关发出 FAS 请求时的策略详情，请参阅转发访问会话。
  - 服务角色 - 服务角色是服务代表您在您的账户中执行操作而分派的 IAM 角色。IAM 管理员可以在 IAM 中创建、修改和删除服务角色。有关更多信息，请参阅《IAM 用户指南》中的创建向 AWS 服务委派权限的角色。
  - 服务相关角色-服务相关角色是一种与服务相关联的服务角色。 AWS 服务服务可以代入代表您执行操作的角色。服务相关角色出现在您的中 AWS 账户 ，并且归服务所有。IAM 管理员可以查看但不能编辑服务相关角色的权限。

- 在 A@@ mazon EC2 上运行的应用程序 — 您可以使用 IAM 角色管理在 EC2 实例上运行并发出 AWS CLI 或 AWS API 请求的应用程序的临时证书。这优先于在 EC2 实例中存储访问密钥。要向 EC2 实例分配 AWS 角色并使其可供其所有应用程序使用，您需要创建附加到该实例的实例配置文件。实例配置文件包含角色，并使 EC2 实例上运行的程序能够获得临时凭证。有关更多信息，请参阅《IAM 用户指南》中的 使用 IAM 角色为 Amazon EC2 实例上运行的应用程序授予权限。

要了解是使用 IAM 角色还是 IAM 用户，请参阅《IAM 用户指南》中的何时创建 IAM 角色（而不是用户）。

使用策略管理访问

您可以 AWS 通过创建策略并将其附加到 AWS 身份或资源来控制中的访问权限。策略是其中的一个对象 AWS ，当与身份或资源关联时，它会定义其权限。 AWS 在委托人（用户、root 用户或角色会话）发出请求时评估这些策略。策略中的权限确定是允许还是拒绝请求。大多数策略都以 JSON 文档的 AWS 形式存储在中。有关 JSON 策略文档的结构和内容的更多信息，请参阅《IAM 用户指南》中的 JSON 策略概览。

管理员可以使用 AWS JSON 策略来指定谁有权访问什么。也就是说，哪个主体 可以对什么资源执行操作，以及在什么条件下执行。

默认情况下，用户和角色没有权限。要授予用户对所需资源执行操作的权限，IAM 管理员可以创建 IAM 策略。管理员随后可以向角色添加 IAM 策略，用户可以代入角色。

IAM 策略定义操作的权限，无关乎您使用哪种方法执行操作。例如，假设您有一个允许 `iam:GetRole` 操作的策略。拥有该策略的用户可以从 AWS Management Console AWS CLI、或 AWS API 获取角色信息。

基于身份的策略

基于身份的策略是可附加到身份（如 IAM 用户、用户组或角色）的 JSON 权限策略文档。这些策略控制用户和角色可在何种条件下对哪些资源执行哪些操作。要了解如何创建基于身份的策略，请参阅《IAM 用户指南》中的创建 IAM 策略。

基于身份的策略可以进一步归类为内联策略或托管式策略。内联策略直接嵌入单个用户、组或角色中。托管策略是独立的策略，您可以将其附加到中的多个用户、群组和角色 AWS 账户。托管策略包括 AWS 托管策略和客户托管策略。要了解如何在托管式策略和内联策略之间进行选择，请参阅《IAM 用户指南》中的在托管式策略与内联策略之间进行选择。

基于资源的策略

基于资源的策略是附加到资源的 JSON 策略文档。基于资源的策略的示例包括 IAM 角色信任策略和 Simple Storage Service（Amazon S3）存储桶策略。在支持基于资源的策略的服务中，服务管理员可以使用它们来控制对特定资源的访问。对于在其中附加策略的资源，策略定义指定主体可以对该资源执行哪些操作以及在什么条件下执行。您必须在基于资源的策略中指定主体。委托人可以包括账户、用户、角色、联合用户或 AWS 服务。

基于资源的策略是位于该服务中的内联策略。您不能在基于资源的策略中使用 IAM 中的 AWS 托管策略。

访问控制列表 (ACL)

访问控制列表（ACL）控制哪些主体（账户成员、用户或角色）有权访问资源。ACL 与基于资源的策略类似，尽管它们不使用 JSON 策略文档格式。

Amazon S3 和 Amazon VPC 就是支持 ACL 的服务示例。 AWS WAF要了解有关 ACL 的更多信息，请参阅《Amazon Simple Storage Service 开发人员指南》 中的访问控制列表（ACL）概览。

其他策略类型

AWS 支持其他不太常见的策略类型。这些策略类型可以设置更常用的策略类型向您授予的最大权限。

- 权限边界 - 权限边界是一个高级功能，用于设置基于身份的策略可以为 IAM 实体（IAM 用户或角色）授予的最大权限。您可为实体设置权限边界。这些结果权限是实体基于身份的策略及其权限边界的交集。在 `Principal` 中指定用户或角色的基于资源的策略不受权限边界限制。任一项策略中的显式拒绝将覆盖允许。有关权限边界的更多信息，请参阅《IAM 用户指南》中的 IAM 实体的权限边界。
- 服务控制策略 (SCP)-SCP 是 JSON 策略，用于指定组织或组织单位 (OU) 的最大权限。 AWS Organizations AWS Organizations 是一项用于对您的企业拥有的多 AWS 账户 项进行分组和集中管理的服务。如果在组织内启用了所有功能，则可对任意或全部账户应用服务控制策略 (SCP)。SCP 限制成员账户中的实体（包括每个 AWS 账户根用户实体）的权限。有关 Organizations 和 SCP 的更多信息，请参阅《AWS Organizations 用户指南》中的 SCP 的工作原理。
- 会话策略 – 会话策略是当您以编程方式为角色或联合用户创建临时会话时作为参数传递的高级策略。结果会话的权限是用户或角色的基于身份的策略和会话策略的交集。权限也可以来自基于资源的策略。任一项策略中的显式拒绝将覆盖允许。有关更多信息，请参阅《IAM 用户指南》中的会话策略。

多个策略类型

当多个类型的策略应用于一个请求时，生成的权限更加复杂和难以理解。要了解在涉及多种策略类型时如何 AWS 确定是否允许请求，请参阅 IAM 用户指南中的策略评估逻辑。

如何 AWS AppSync 与 IAM 配合使用

在使用 IAM 管理访问权限之前 AWS AppSync，请先了解有哪些 IAM 功能可供使用 AWS AppSync。

您可以与之配合使用的 IAM 功能 AWS AppSync

IAM 功能	AWS AppSync 支持
基于身份的策略	是
基于资源的策略	否
策略操作	是
策略资源	是
策略条件密钥	否
ACL	否
ABAC（策略中的标签）	部分
临时凭证	是
转发访问会话（FAS）	部分
服务角色	否
服务相关角色	部分

要全面了解 AWS AppSync 以及其他 AWS 服务如何与大多数 IAM 功能配合使用，请参阅 IAM 用户指南中的与 IAM 配合使用的AWS 服务。

基于身份的策略 AWS AppSync

支持基于身份的策略	是

基于身份的策略是可附加到身份（如 IAM 用户、用户组或角色）的 JSON 权限策略文档。这些策略控制用户和角色可在何种条件下对哪些资源执行哪些操作。要了解如何创建基于身份的策略，请参阅 IAM 用户指南中的创建 IAM 策略。

通过使用 IAM 基于身份的策略，您可以指定允许或拒绝的操作和资源以及允许或拒绝操作的条件。您无法在基于身份的策略中指定主体，因为它适用于其附加的用户或角色。要了解可在 JSON 策略中使用的所有元素，请参阅《IAM 用户指南》中的 IAM JSON 策略元素引用。

基于身份的策略示例 AWS AppSync

要查看 AWS AppSync 基于身份的策略的示例，请参阅。适用于 AWS AppSync 的基于身份的策略

内部基于资源的政策 AWS AppSync

支持基于资源的策略	否

基于资源的策略是附加到资源的 JSON 策略文档。基于资源的策略的示例包括 IAM 角色信任策略和 Simple Storage Service（Amazon S3）存储桶策略。在支持基于资源的策略的服务中，服务管理员可以使用它们来控制对特定资源的访问。对于在其中附加策略的资源，策略定义指定主体可以对该资源执行哪些操作以及在什么条件下执行。您必须在基于资源的策略中指定主体。委托人可以包括账户、用户、角色、联合用户或 AWS 服务。

要启用跨账户存取，您可以将整个账户或其他账户中的 IAM 实体指定为基于资源的策略中的主体。将跨账户主体添加到基于资源的策略只是建立信任关系工作的一半而已。当委托人和资源处于不同位置时 AWS 账户，可信账户中的 IAM 管理员还必须向委托人实体（用户或角色）授予访问资源的权限。他们通过将基于身份的策略附加到实体以授予权限。但是，如果基于资源的策略向同一个账户中的主体授予访问权限，则不需要额外的基于身份的策略。有关更多信息，请参阅《IAM 用户指南》中的 IAM 角色与基于资源的策略有何不同。

的政策行动 AWS AppSync

支持策略操作	是

管理员可以使用 AWS JSON 策略来指定谁有权访问什么。也就是说，哪个主体 可以对什么资源 执行操作，以及在什么 条件 下执行。

JSON 策略的 `Action` 元素描述可用于在策略中允许或拒绝访问的操作。策略操作通常与关联的 AWS API 操作同名。有一些例外情况，例如没有匹配 API 操作的仅限权限 操作。还有一些操作需要在策略中执行多个操作。这些附加操作称为相关操作。

在策略中包含操作以授予执行关联操作的权限。

要查看 AWS AppSync 操作列表，请参阅《服务授权参考》 AWS AppSync中定义的操作。

正在执行的策略操作在操作前 AWS AppSync 使用以下前缀：

```
appsync
```

要在单个语句中指定多项操作，请使用逗号将它们隔开。

```
"Action": [
      "appsync:action1",
      "appsync:action2"
         ]
```

要查看 AWS AppSync 基于身份的策略的示例，请参阅。适用于 AWS AppSync 的基于身份的策略

的政策资源 AWS AppSync

支持策略资源	是

管理员可以使用 AWS JSON 策略来指定谁有权访问什么。也就是说，哪个主体 可以对什么资源 执行操作，以及在什么条件 下执行。

`Resource` JSON 策略元素指定要向其应用操作的一个或多个对象。语句必须包含 `Resource` 或 `NotResource` 元素。作为最佳实践，请使用其 Amazon 资源名称（ARN）指定资源。对于支持特定资源类型（称为资源级权限）的操作，您可以执行此操作。

对于不支持资源级权限的操作（如列出操作），请使用通配符 (*) 指示语句应用于所有资源。

```
"Resource": "*"
```

要查看 AWS AppSync 资源类型及其 ARN 的列表，请参阅《服务授权参考》 AWS AppSync中定义的资源。要了解您可以使用哪些操作来指定每种资源的 ARN，请参阅由定义的操作。 AWS AppSync

要查看 AWS AppSync 基于身份的策略的示例，请参阅。适用于 AWS AppSync 的基于身份的策略

的策略条件密钥 AWS AppSync

支持特定于服务的策略条件密钥	否

管理员可以使用 AWS JSON 策略来指定谁有权访问什么。也就是说，哪个主体可以对什么资源执行操作，以及在什么条件下执行。

在 Condition 元素（或 Condition 块）中，可以指定语句生效的条件。Condition 元素是可选的。您可以创建使用条件运算符（例如，等于或小于）的条件表达式，以使策略中的条件与请求中的值相匹配。

如果您在一个语句中指定多个 Condition 元素，或在单个 Condition 元素中指定多个键，则 AWS 使用逻辑 AND 运算评估它们。如果您为单个条件键指定多个值，则使用逻辑OR运算来 AWS 评估条件。在授予语句的权限之前必须满足所有的条件。

在指定条件时，您也可以使用占位符变量。例如，只有在使用 IAM 用户名标记 IAM 用户时，您才能为其授予访问资源的权限。有关更多信息，请参阅《IAM 用户指南》中的 IAM 策略元素：变量和标签。

AWS 支持全局条件密钥和特定于服务的条件密钥。要查看所有 AWS 全局条件键，请参阅 IAM 用户指南中的AWS 全局条件上下文密钥。

要查看 AWS AppSync 条件密钥列表，请参阅《服务授权参考》 AWS AppSync中的条件密钥。要了解可以使用条件键的操作和资源，请参阅由定义的操作 AWS AppSync。

要查看 AWS AppSync 基于身份的策略的示例，请参阅。适用于 AWS AppSync 的基于身份的策略

AWS AppSync 中的访问控制列表 (ACL)

支持 ACL	否

访问控制列表（ACL）控制哪些主体（账户成员、用户或角色）有权访问资源。ACL 与基于资源的策略类似，尽管它们不使用 JSON 策略文档格式。

基于属性的访问控制 (ABAC) AWS AppSync

支持 ABAC（策略中的标签）	部分

基于属性的访问权限控制（ABAC）是一种授权策略，该策略基于属性来定义权限。在中 AWS，这些属性称为标签。您可以将标签附加到 IAM 实体（用户或角色）和许多 AWS 资源。标记实体和资源是 ABAC 的第一步。然后设计 ABAC 策略，以在主体的标签与他们尝试访问的资源标签匹配时允许操作。

ABAC 在快速增长的环境中非常有用，并在策略管理变得繁琐的情况下可以提供帮助。

要基于标签控制访问，您需要使用 `aws:ResourceTag/key-name`、`aws:RequestTag/key-name` 或 `aws:TagKeys` 条件键在策略的条件元素中提供标签信息。

如果某个服务对于每种资源类型都支持所有这三个条件键，则对于该服务，该值为是。如果某个服务仅对于部分资源类型支持所有这三个条件键，则该值为部分。

有关 ABAC 的更多信息,请参阅《IAM 用户指南》中的什么是 ABAC？。要查看设置 ABAC 步骤的教程，请参阅《IAM 用户指南》中的使用基于属性的访问权限控制（ABAC）。

将临时凭证与 AWS AppSync

支持临时凭证	是

当你使用临时证书登录时，有些 AWS 服务 不起作用。有关更多信息，包括哪些 AWS 服务 适用于临时证书，请参阅 IAM 用户指南中的AWS 服务 与 IA M 配合使用的信息。

如果您使用除用户名和密码之外的任何方法登录，则 AWS Management Console 使用的是临时证书。例如，当您 AWS 使用公司的单点登录 (SSO) 链接进行访问时，该过程会自动创建临时证书。当您以用户身份登录控制台，然后切换角色时，您还会自动创建临时凭证。有关切换角色的更多信息，请参阅《IAM 用户指南》中的 切换到角色（控制台）。

您可以使用 AWS CLI 或 AWS API 手动创建临时证书。然后，您可以使用这些临时证书进行访问 AWS。 AWS 建议您动态生成临时证书，而不是使用长期访问密钥。有关更多信息，请参阅 IAM 中的临时安全凭证。

转发访问会话 AWS AppSync

支持转发访问会话 (FAS)	部分

当您使用 IAM 用户或角色在中执行操作时 AWS，您被视为委托人。使用某些服务时，您可能会执行一个操作，然后此操作在其他服务中启动另一个操作。FAS 使用调用委托人的权限以及 AWS 服务 向下游服务发出请求的请求。 AWS 服务只有当服务收到需要与其他 AWS 服务 或资源交互才能完成的请求时，才会发出 FAS 请求。在这种情况下，您必须具有执行这两个操作的权限。有关发出 FAS 请求时的策略详情，请参阅转发访问会话。

AWS AppSync 的服务角色

支持服务角色	否

服务角色是由一项服务担任、代表您执行操作的 IAM 角色。IAM 管理员可以在 IAM 中创建、修改和删除服务角色。有关更多信息，请参阅《IAM 用户指南》中的创建向 AWS 服务委派权限的角色。

> **Warning**
>
> 更改服务角色的权限可能会中断 AWS AppSync 功能。只有在 AWS AppSync 提供操作指导时才编辑服务角色。

的服务相关角色 AWS AppSync

支持服务相关角色	部分

服务相关角色是一种与服务相关联的 AWS 服务服务角色。服务可以代入代表您执行操作的角色。服务相关角色出现在您的中 AWS 账户 ，并且归服务所有。IAM 管理员可以查看但不能编辑服务相关角色的权限。

有关创建或管理服务相关角色的详细信息，请参阅 IAM 用户指南中的能够与 IAM 搭配使用的AWS 服务。在表中查找服务相关角色列中包含 Yes 额表。选择是链接以查看该服务的服务相关角色文档。

适用于 AWS AppSync 的基于身份的策略

默认情况下，用户和角色无权创建或修改 AWS AppSync资源。他们也无法使用 AWS Management Console、 AWS Command Line Interface (AWS CLI) 或 AWS API 执行任务。要授予用户对所需资源执行操作的权限，IAM 管理员可以创建 IAM 策略。管理员随后可以向角色添加 IAM 策略，用户可以代入角色。

要了解如何使用这些示例 JSON 策略文档创建基于 IAM 身份的策略，请参阅 IAM 用户指南中的 创建 IAM 策略。

有关由定义的操作和资源类型的详细信息 AWS AppSync，包括每种资源类型的 ARN 格式，请参阅《服务授权参考》 AWS AppSync中的操作、资源和条件密钥。

要了解创建和配置 IAM 基于身份的策略的最佳实践，请参阅the section called “IAM 策略最佳实践”。

有关的 IAM 基于身份的策略列表 AWS AppSync，请参阅。AWS 的托管策略 AWS AppSync

主题

- 使用 AWS AppSync控制台
- 允许用户查看他们自己的权限
- 访问一个 Amazon S3 存储桶
- 根据标签查看 AWS AppSync 微件
- AWS 的托管策略 AWS AppSync

使用 AWS AppSync控制台

要访问 AWS AppSync 控制台，您必须拥有一组最低权限。这些权限必须允许您列出和查看有关您的 AWS AppSync 资源的详细信息 AWS 账户。如果创建比必需的最低权限更为严格的基于身份的策略，对于附加了该策略的实体（用户或角色），控制台将无法按预期正常运行。

对于仅调用 AWS CLI 或 AWS API 的用户，您无需为其设置最低控制台权限。相反，只允许访问与其尝试执行的 API 操作相匹配的操作。

为确保 IAM 用户和角色仍然可以使用 AWS AppSync 控制台，还需要将 AWS AppSync `ConsoleAccess`或`ReadOnly` AWS 托管策略附加到实体。有关更多信息，请参阅《IAM 用户指南》中的为用户添加权限。

允许用户查看他们自己的权限

该示例说明了您如何创建策略，以允许 IAM 用户查看附加到其用户身份的内联和托管式策略。此策略包括在控制台上或使用 AWS CLI 或 AWS API 以编程方式完成此操作的权限。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

访问一个 Amazon S3 存储桶

在本示例中，您想向 AWS 账户中的 IAM 用户授予访问您的 Amazon S3 存储桶的权限。`examplebucket`您还想要允许该用户添加、更新和删除对象。

除了授予该用户 s3:PutObject、s3:GetObject 和 s3:DeleteObject 权限外，此策略还授予 s3:ListAllMyBuckets、s3:GetBucketLocation 和 s3:ListBucket 权限。这些是控制台所需的其他权限。此外，s3:PutObjectAcl 和 s3:GetObjectAcl 操作需要能够在控制台中复制、剪切和粘贴对象。有关为用户授予权限并使用控制台测试用户的示例演练，请参阅[示例演练：使用用户策略控制对桶的访问]。

```
{
   "Version":"2012-10-17",
   "Statement":[
      {
         "Sid":"ListBucketsInConsole",
         "Effect":"Allow",
         "Action":[
            "s3:ListAllMyBuckets"
         ],
         "Resource":"arn:aws:s3:::*"
      },
      {
         "Sid":"ViewSpecificBucketInfo",
         "Effect":"Allow",
         "Action":[
            "s3:ListBucket",
            "s3:GetBucketLocation"
         ],
         "Resource":"arn:aws:s3:::examplebucket"
      },
      {
         "Sid":"ManageBucketContents",
         "Effect":"Allow",
         "Action":[
            "s3:PutObject",
            "s3:PutObjectAcl",
            "s3:GetObject",
            "s3:GetObjectAcl",
            "s3:DeleteObject"
         ],
         "Resource":"arn:aws:s3:::examplebucket/*"
      }
   ]
}
```

根据标签查看 AWS AppSync

您可以使用基于身份的策略中的条件根据标签控制对 AWS AppSync资源的访问权限。该示例说明了如何创建允许查看###的策略。不过，只有在###标签 Owner 具有该用户的用户名值时，才会授予权限。此策略还授予在控制台上完成此操作的必要权限。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ListWidgetsInConsole",
            "Effect": "Allow",
            "Action": "appsync:ListWidgets",
            "Resource": "*"
        },
        {
            "Sid": "ViewWidgetIfOwner",
            "Effect": "Allow",
            "Action": "appsync:GetWidget",
            "Resource": "arn:aws:appsync:*:*:widget/*",
            "Condition": {
                "StringEquals": {"aws:ResourceTag/Owner": "${aws:username}"}
            }
        }
    ]
}
```

您可以将该策略附加到您账户中的 IAM 用户。如果名为的用户richard-roe尝试查看 AWS AppSync##，则必须为该###加上标签Owner=richard-roe或owner=richard-roe。否则，将拒绝其访问。条件标签键 Owner 匹配 Owner 和 owner，因为条件键名称不区分大小写。有关更多信息，请参阅《IAM 用户指南》中的 IAM JSON 策略元素：条件。

AWS 的托管策略 AWS AppSync

要向用户、群组和角色添加权限，使用 AWS 托管策略比自己编写策略要容易得多。创建 IAM 客户托管策略需要时间和专业知识，这些策略仅为您的团队提供所需的权限。要快速入门，您可以使用我们的 AWS 托管策略。这些策略涵盖常见使用案例，可在您的 AWS 账户中使用。有关 AWS 托管策略的更多信息，请参阅 IAM 用户指南中的AWS 托管策略。

AWS 服务维护和更新 AWS 托管策略。您无法更改 AWS 托管策略中的权限。服务偶尔会向 AWS 托管策略添加其他权限以支持新功能。此类更新会影响附加策略的所有身份（用户、组和角色）。当推出新功能或有新操作可用时，服务最有可能更新 AWS 托管策略。服务不会从 AWS 托管策略中移除权限，因此策略更新不会破坏您的现有权限。

此外，还 AWS 支持跨多个服务的工作职能的托管策略。例如，ReadOnlyAccess AWS 托管策略提供对所有 AWS 服务和资源的只读访问权限。当服务启动一项新功能时， AWS 会为新操作和资源添加只读权限。有关工作职能策略的列表和说明，请参阅《IAM 用户指南》中的适用于工作职能的AWS 托管式策略。

AWS 托管策略： AWSAppSyncInvokeFullAccess

使用AWSAppSyncInvokeFullAccess AWS 托管策略允许您的管理员通过控制台或独立访问 AWS AppSync 服务。

您可以将 AWSAppSyncInvokeFullAccess 策略附加到 IAM 身份。

权限详细信息

该策略包含以下权限。

- AWS AppSync— 允许对中的所有资源具有完全的管理访问权限 AWS AppSync

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "appsync:GraphQL",
                "appsync:GetGraphqlApi",
                "appsync:ListGraphqlApis",
                "appsync:ListApiKeys"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS 托管策略： AWSAppSyncSchemaAuthor

使用AWSAppSyncSchemaAuthor AWS 托管策略允许 IAM 用户访问创建、更新和查询其 GraphQL 架构。有关用户可以使用这些权限执行哪些操作的信息，请参阅设计 GraphQL API。

您可以将 AWSAppSyncSchemaAuthor 策略附加到 IAM 身份。

权限详细信息

该策略包含以下权限。

- AWS AppSync - 允许执行以下操作：
  - 创建 GraphQL 架构
  - 允许创建、修改和删除 GraphQL 类型、解析器和函数
  - 评估请求和响应模板逻辑
  - 使用运行时系统和上下文评估代码
  - 将 GraphQL 查询发送到 GraphQL API
  - 检索 GraphQL 数据

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "appsync:GraphQL",
                "appsync:CreateResolver",
                "appsync:CreateType",
                "appsync:DeleteResolver",
                "appsync:DeleteType",
                "appsync:GetResolver",
                "appsync:GetType",
                "appsync:GetDataSource",
                "appsync:GetSchemaCreationStatus",
                "appsync:GetIntrospectionSchema",
                "appsync:GetGraphqlApi",
                "appsync:ListTypes",
```

```
                "appsync:ListApiKeys",
                "appsync:ListResolvers",
                "appsync:ListDataSources",
                "appsync:ListGraphqlApis",
                "appsync:StartSchemaCreation",
                "appsync:UpdateResolver",
                "appsync:UpdateType",
                "appsync:TagResource",
                "appsync:UntagResource",
                "appsync:ListTagsForResource",
                "appsync:CreateFunction",
                "appsync:UpdateFunction",
                "appsync:GetFunction",
                "appsync:DeleteFunction",
                "appsync:ListFunctions",
                "appsync:ListResolversByFunction",
                "appsync:EvaluateMappingTemplate",
                "appsync:EvaluateCode"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS 托管策略： AWSAppSyncPushToCloudWatchLogs

AWS AppSync 使用 Amazon CloudWatch 通过生成可用于排查和优化 GraphQL 请求的日志来监控应用程序的性能。有关更多信息，请参阅 监控和日志记录。

使用AWSAppSyncPushToCloudWatchLogs AWS 托管策略允许将日志推送 AWS AppSync 到 IAM 用户的 CloudWatch 账户。

您可以将 AWSAppSyncPushToCloudWatchLogs 策略附加到 IAM 身份。

权限详细信息

该策略包含以下权限。

- CloudWatch Logs— AWS AppSync 允许创建具有指定名称的日志组和流。 AWS AppSync将日志事件推送到指定的日志流。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS 托管策略： AWSAppSyncAdministrator

使用AWSAppSyncAdministrator AWS 托管策略允许您的管理员访问 AWS AppSync除 AWS 控制台之外的所有内容。

您可以将 AWSAppSyncAdministrator 附加到 IAM 实体。 AWS AppSync 还将此策略附加到允许其代表您执行操作的服务角色。

权限详细信息

该策略包含以下权限。

- AWS AppSync— 允许对中的所有资源具有完全的管理访问权限 AWS AppSync
- IAM - 允许执行以下操作：
  - 创建服务相关角色 AWS AppSync 以允许代表您分析其他服务中的资源
  - 删除服务相关角色
  - 将服务相关角色传递给其他 AWS 服务，以便日后代入该角色并代表您执行操作

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Effect": "Allow",
            "Action": [
                "appsync:*"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": [
                        "appsync.amazonaws.com"
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": "iam:CreateServiceLinkedRole",
            "Resource": "*",
            "Condition": {
                "StringEquals": {
                    "iam:AWSServiceName": "appsync.amazonaws.com"
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "iam:DeleteServiceLinkedRole",
                "iam:GetServiceLinkedRoleDeletionStatus"
            ],
            "Resource": "arn:aws:iam::*:role/aws-service-role/appsync.amazonaws.com/
AWSServiceRoleForAppSync*"
        }
    ]
}
```

AWS 托管策略： AWSAppSyncServiceRolePolicy

使用AWSAppSyncServiceRolePolicy AWS 托管策略允许访问 AWS AppSync使用或管理的 AWS 服务和资源。

您不能将 AWSAppSyncServiceRolePolicy 附加到自己的 IAM 实体。此策略附加到允许代表您执行操作 AWS AppSync 的服务相关角色。有关更多信息，请参阅 的服务相关角色 AWS AppSync。

权限详细信息

该策略包含以下权限。

- X-Ray— AWS AppSync AWS X-Ray 用于收集有关在您的应用程序中提出的请求的数据。有关更多信息，请参阅 使用 AWS X-Ray 进行跟踪。

  该策略允许执行以下操作：

  - 检索采样规则及其结果
  - 将跟踪数据发送到 X-Ray 进程守护程序

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:PutTraceSegments",
                "xray:PutTelemetryRecords",
                "xray:GetSamplingTargets",
                "xray:GetSamplingRules",
                "xray:GetSamplingStatisticSummaries"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

对 AWS 托管式策略的AWS AppSync 更新

查看 AWS AppSync 自该服务开始跟踪这些更改以来 AWS 托管策略更新的详细信息。要获得有关此页面变更的自动提醒，请订阅 “ AWS AppSync 文档历史记录” 页面上的 RSS feed。

更改	描述	日期
AWSAppSyncSchemaAuthor – 对现有策略的更新	添加了 `EvaluateCode` 策略操作，以允许用户使用运行时系统和上下文评估代码。	2023 年 2 月 7 日
AWSAppSyncSchemaAuthor – 对现有策略的更新	添加了策略操作以允许 API 的 list、get、create、update 和 delete 函数。 添加了 `EvaluateMappingTemplate` 策略操作，以允许用户评估请求和响应解析器映射模板逻辑。 添加了策略操作以允许资源标记。	2022 年 8 月 25 日
AWS AppSync 已开始跟踪更改	AWS AppSync 开始跟踪其 AWS 托管策略的更改。	2022 年 8 月 25 日

对 AWS AppSync 身份和访问进行故障排除

使用以下信息来帮助您诊断和修复在使用 AWS AppSync 和 IAM 时可能遇到的常见问题。

我无权在以下位置执行操作 AWS AppSync

如果 AWS Management Console 告诉您您无权执行某项操作，则必须联系管理员寻求帮助。管理员是指提供用户名和密码的人员。

在 IAM 用户 `mateojackson` 尝试使用控制台查看有关虚构 *`my-example-widget`* 资源的详细信息时，由于他没有虚构 `appsync:`*`GetWidget`* 权限，出现以下示例错误。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 appsync:GetWidget on resource: my-example-widget
```

在这种情况下，Mateo 请求他的管理员更新其策略，以允许他使用 `appsync:GetWidget` 操作访问 `my-example-widget` 资源。

我无权执行 iam：PassRole

如果您收到错误消息，提示您无权执行`iam:PassRole`操作，则必须更新您的策略以允许您将角色传递给 AWS AppSync。

有些 AWS 服务 允许您将现有角色传递给该服务，而不是创建新的服务角色或服务相关角色。为此，您必须具有将角色传递到服务的权限。

当名为的 IAM 用户`marymajor`尝试使用控制台在中执行操作时，会出现以下示例错误 AWS AppSync。但是，服务必须具有服务角色所授予的权限才可执行此操作。Mary 不具有将角色传递到服务的权限。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

在这种情况下，必须更新 Mary 的策略以允许她执行 `iam:PassRole` 操作。

如果您需要帮助，请联系您的 AWS 管理员。您的管理员是提供登录凭证的人。

我想要查看我的访问密钥

在创建 IAM 用户访问密钥后，您可以随时查看您的访问密钥 ID。但是，您无法再查看您的秘密访问密钥。如果您丢失了私有密钥，则必须创建一个新的访问密钥对。

访问密钥包含两部分：访问密钥 ID（例如 `AKIAIOSFODNN7EXAMPLE`）和秘密访问密钥（例如 `wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY`）。与用户名和密码一样，您必须同时使用访问密钥 ID 和秘密访问密钥对请求执行身份验证。像对用户名和密码一样，安全地管理访问密钥。

> **Important**
>
> 请不要向第三方提供访问密钥，即便是为了帮助找到您的规范用户 ID 也不行。通过这样做，您可以授予他人永久访问您的权限 AWS 账户。

当您创建访问密钥对时，系统会提示您将访问密钥 ID 和秘密访问密钥保存在一个安全位置。秘密访问密钥仅在您创建它时可用。如果丢失了您的秘密访问密钥，您必须为 IAM 用户添加新的访问密钥。您最多可拥有两个访问密钥。如果您已有两个密钥，则必须删除一个密钥对，然后再创建新的密钥。要查看说明，请参阅 IAM 用户指南中的管理访问密钥。

我是一名管理员，想允许其他人访问 AWS AppSync

要允许其他人访问 AWS AppSync，您必须为需要访问的人员或应用程序创建 IAM 实体（用户或角色）。它们将使用该实体的凭证访问 AWS。然后，您必须将策略附加到授予他们在中的正确权限的实体 AWS AppSync。

要立即开始使用，请参阅《IAM 用户指南》中的创建您的第一个 IAM 委派用户和组。

我想允许 AWS 账户之外的人访问我的 AWS AppSync 资源

您可以创建一个角色，以便其他账户中的用户或您组织外的人员可以使用该角色来访问您的资源。您可以指定谁值得信赖，可以担任角色。对于支持基于资源的策略或访问控制列表（ACL）的服务，您可以使用这些策略向人员授予对您的资源的访问权。

要了解更多信息，请参阅以下内容：

- 要了解是否 AWS AppSync 支持这些功能，请参阅如何 AWS AppSync 与 IAM 配合使用。
- 要了解如何提供对您拥有的资源的访问权限 AWS 账户 ，请参阅 IAM 用户指南中的向您拥有 AWS 账户 的另一个 IAM 用户提供访问权限。
- 要了解如何向第三方提供对您的资源的访问权限 AWS 账户，请参阅 IAM 用户指南中的向第三方提供访问权限。 AWS 账户
- 要了解如何通过身份联合验证提供访问权限，请参阅《IAM 用户指南》中的为经过外部身份验证的用户（身份联合验证）提供访问权限。
- 要了解使用角色和基于资源的策略进行跨账户存取之间的差别，请参阅《IAM 用户指南》中的 IAM 角色与基于资源的策略有何不同。

使用记录 AWS AppSync API 调用 AWS CloudTrail

AWS AppSync 与 AWS CloudTrail一项服务集成，该服务提供用户、角色或 AWS 服务在中执行的操作的记录 AWS AppSync。 CloudTrail 将发出的 API 调用捕获 AWS AppSync 为事件。捕获的调用包括来自 AWS AppSync 控制台的调用和对 AWS AppSync API 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，包括的事件 AWS AppSync。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。使用收集的信息 CloudTrail，您可以确定向哪个请求发出 AWS AppSync、发出请求的 IP 地址、谁发出了请求、何时发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》。

AWS AppSync 信息在 CloudTrail

CloudTrail 在您创建 AWS 账户时已在您的账户上启用。当活动发生在中时 AWS AppSync，该活动会与其他 AWS 服务 CloudTrail 事件一起记录在事件历史记录中。您可以在自己的 AWS 账户中查看、搜索和下载最近发生的事件。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 AWS 账户中的事件，包括的事件 AWS AppSync，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，当您在控制台中创建跟踪时，该跟踪将应用于所有 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：

- 创建跟踪记录概述
- 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

AWS AppSync 支持记录通过 AWS AppSync API 发出的呼叫。目前，对你的 API 的调用以及对解析器的调用都不会被 AWS AppSync 登录到 CloudTrail。

每个事件或日记账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：

- 请求是使用根证书还是 AWS Identity and Access Management (IAM) 用户凭证发出。
- 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 用户身份元素。

了解 AWS AppSync 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。

以下示例显示了一个 CloudTrail 日志条目，该条目演示了通过 AWS AppSync 控制台执行的`GetGraphqlApi`操作：

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "AssumedRole",
        "principalId": "ABCDEFXAMPLEPRINCIPAL:nikkiwolf",
        "arn": "arn:aws:sts::111122223333:assumed-role/admin/nikkiwolf",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "sessionContext": {
            "sessionIssuer": {
                "type": "Role",
                "principalId": "AIDAJ45Q7YFFAREXAMPLE",
                "arn": "arn:aws:iam::111122223333:role/admin",
                "accountId": "111122223333",
                "userName": "admin"
            },
            "webIdFederationData": {},
            "attributes": {
                "mfaAuthenticated": "false",
                "creationDate": "2021-03-12T22:41:48Z"
            }
        }
    },
    "eventTime": "2021-03-12T22:46:18Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "GetGraphqlApi",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "203.0.113.69",
    "userAgent": "aws-internal/3 aws-sdk-java/1.11.942
 Linux/4.9.230-0.1.ac.223.84.332.metal1.x86_64 OpenJDK_64-Bit_Server_VM/25.282-b08
 java/1.8.0_282 vendor/Oracle_Corporation",
    "requestParameters": {
        "apiId": "xhxt3typtfnmidkhcexampleid"
    },
    "responseElements": null,
    "requestID": "2fc43a35-a552-4b5d-be6e-12553a03dd12",
    "eventID": "b95b0ad9-8c71-4252-a2ec-5dc2fe5f8ae8",
    "readOnly": true,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "eventCategory": "Management",
    "recipientAccountId": "111122223333"
}
```

以下示例显示了一个 CloudTrail 日志条目，该条目演示了通过执行的`CreateApikey`操作 AWS CLI：

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "ABCDEFXAMPLEPRINCIPAL",
        "arn": "arn:aws:iam::111122223333:user/nikkiwolf",
        "accountId": "111122223333",
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "nikkiwolf"
    },
    "eventTime": "2021-03-12T22:49:10Z",
    "eventSource": "appsync.amazonaws.com",
    "eventName": "CreateApiKey",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "203.0.113.69",
    "userAgent": "aws-cli/2.0.11 Python/3.7.4 Darwin/18.7.0 botocore/2.0.0dev15",
    "requestParameters": {
        "apiId": "xhxt3typtfnmidkhcexampleid"
    },
    "responseElements": {
        "apiKey": {
            "id": "***",
            "expires": 1616191200,
            "deletes": 1621375200
        }
    },
    "requestID": "e152190e-04ba-4d0a-ae7b-6bfc0bcea6af",
    "eventID": "ba3f39e0-9d87-41c5-abbb-2000abcb6013",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "eventCategory": "Management",
    "recipientAccountId": "111122223333"
}
```

以下方面的安全最佳实践 AWS AppSync

保护不仅仅 AWS AppSync 是简单地打开几个杠杆或设置日志记录。以下几节介绍了一些安全最佳实践，这些最佳实践因您使用该服务的方式而有所不同。

了解身份验证方法

AWS AppSync 提供了多种方法来向 GraphQL API 对用户进行身份验证。每种方法在安全性、可审核性和可用性方面进行了权衡和取舍。

可以使用以下常见的身份验证方法：

- Amazon Cognito 用户池允许 GraphQL API 使用用户属性进行精细的访问控制和筛选。
- API 令牌具有有限的生命周期，并适用于自动化系统，例如持续集成系统以及与外部 API 的集成。
- AWS Identity and Access Management (IAM) 适用于在您中管理的内部应用程序 AWS 账户。
- OpenID Connect 允许您通过 OpenID Connect 协议控制和联合访问。

有关中身份验证和授权的更多信息 AWS AppSync，请参阅授权和身份验证。

在 HTTP 解析器中使用 TLS

在使用 HTTP 解析器时，请确保尽可能使用 TLS 保护的 (HTTPS) 连接。有关 AWS AppSync 信任的 TLS 证书的完整列表，请参阅AWS AppSync 识别的 HTTPS 终端节点证书颁发机构 (CA)。

尽可能使用具有最低权限的角色

在使用解析器（例如 DynamoDB 解析器）时，请使用为您的资源（例如 Amazon DynamoDB 表）提供最严格限制的角色。

IAM 策略最佳实践

基于身份的策略决定了某人是否可以在您的账户中创建、访问或删除 AWS AppSync 资源。这些操作可能会使 AWS 账户产生成本。创建或编辑基于身份的策略时，请遵循以下准则和建议：

- 开始使用 AWS 托管策略并转向最低权限权限 — 要开始向用户和工作负载授予权限，请使用为许多常见用例授予权限的AWS 托管策略。它们在你的版本中可用 AWS 账户。我们建议您通过定义针对您的用例的 AWS 客户托管策略来进一步减少权限。有关更多信息，请参阅《IAM 用户指南》中的AWS 托管策略或工作职能的AWS 托管策略。
- 应用最低权限 – 在使用 IAM 策略设置权限时，请仅授予执行任务所需的权限。为此，您可以定义在特定条件下可以对特定资源执行的操作，也称为最低权限许可。有关使用 IAM 应用权限的更多信息，请参阅《IAM 用户指南》中的 IAM 中的策略和权限。
- 使用 IAM 策略中的条件进一步限制访问权限 – 您可以向策略添加条件来限制对操作和资源的访问。例如，您可以编写策略条件来指定必须使用 SSL 发送所有请求。如果服务操作是通过特定的方式使

用的，则也可以使用条件来授予对服务操作的访问权限 AWS 服务，例如 AWS CloudFormation。有关更多信息，请参阅《IAM 用户指南》中的 IAM JSON 策略元素：条件。

- 使用 IAM Access Analyzer 验证您的 IAM 策略，以确保权限的安全性和功能性 – IAM Access Analyzer 会验证新策略和现有策略，以确保策略符合 IAM 策略语言 (JSON) 和 IAM 最佳实践。IAM Access Analyzer 提供 100 多项策略检查和可操作的建议，以帮助您制定安全且功能性强的策略。有关更多信息，请参阅《IAM 用户指南》中的 IAM Acess Analyzer 策略验证。
- 需要多重身份验证 (MFA)-如果 AWS 账户您的场景需要 IAM 用户或根用户，请启用 MFA 以提高安全性。若要在调用 API 操作时需要 MFA，请将 MFA 条件添加到您的策略中。有关更多信息，请参阅《IAM 用户指南》中的 配置受 MFA 保护的 API 访问。

有关 IAM 中的最佳实操的更多信息，请参阅《IAM 用户指南》中的 IAM 中的安全最佳实操。

解析器参考 (JavaScript)

以下几节介绍了 APPSYNC_JS 运行时环境和 JavaScript 解析器。

主题

JavaScript 解析器概述

在 AWS AppSync 中，您可以对数据源执行操作以响应 GraphQL 请求。对于您希望运行查询、变更或订阅的每个 GraphQL 字段，必须附加一个解析器。

解析器是 GraphQL 和数据源之间的连接器。它们指示 AWS AppSync 如何将传入的 GraphQL 请求转换为后端数据源的指令，以及如何将该数据源的响应转换回 GraphQL 响应。对于 AWS AppSync，您可以使用 JavaScript 编写解析器，并在 AWS AppSync (APPSYNC_JS) 环境中运行它们。

AWS AppSync 允许您编写单位解析器或由管道中的多个 AWS AppSync 函数组成的管道解析器。

支持的运行时功能

AWS AppSync JavaScript 运行时环境提供一部分 JavaScript 库、实用程序和功能。有关 APPSYNC_JS 运行时环境支持的功能的完整列表，请参阅解析器和函数的 JavaScript 运行时功能。

单位解析器

单位解析器由定义对数据源执行的请求和响应处理程序的代码组成。请求处理程序将上下文对象作为参数，并返回用于调用数据源的请求负载。响应处理程序接收从数据源返回的负载以及执行的请求

结果。响应处理程序将负载转换为 GraphQL 响应以解析 GraphQL 字段。在以下示例中，解析器从 DynamoDB 数据源中检索项目：

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
  return ddb.get({ key: { id: ctx.args.id } });
}

export const response = (ctx) => ctx.result;
```

JavaScript 管道解析器剖析

管道解析器由定义请求和响应处理程序以及函数列表的代码组成。每个函数具有一个对数据源执行的请求和响应处理程序。由于管道解析器将运行委派给一组函数，因此，它不会链接到任何数据源。单位解析器和函数是对数据源执行操作的基元。

管道解析器请求处理程序

管道解析器的请求处理程序（预备步骤）允许您在运行定义的函数之前执行一些准备逻辑。

函数列表

管道解析器将按顺序运行的函数的列表。管道解析器请求处理程序评估结果作为 `ctx.prev.result` 提供给第一个函数。每个函数评估结果作为 `ctx.prev.result` 提供给下一个函数。

管道解析器响应处理程序

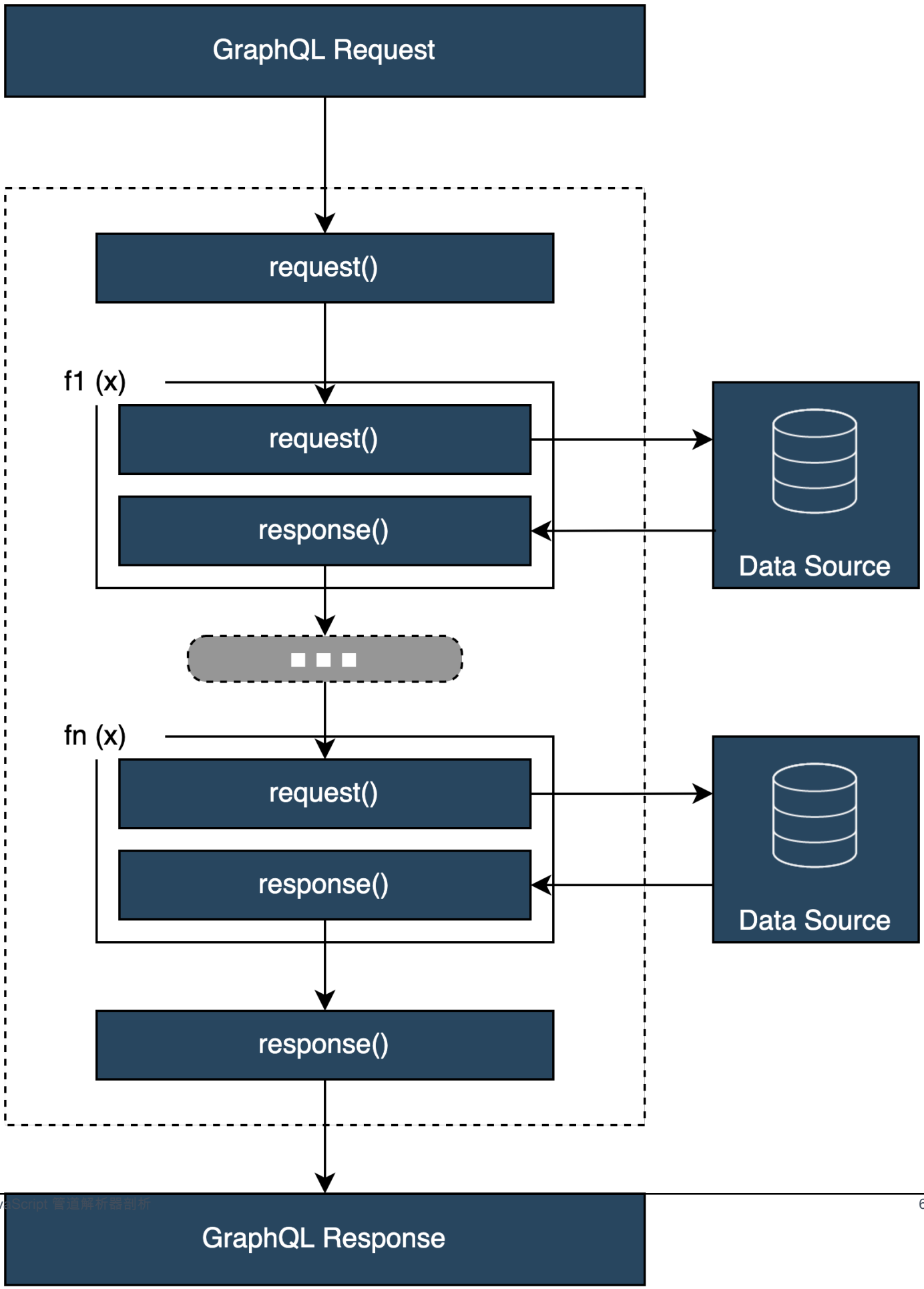
管道解析器的响应处理程序允许您执行从最后一个函数的输出到预期 GraphQL 字段类型的一些最终逻辑。函数列表中的最后一个函数的输出在管道解析器响应处理程序中作为 `ctx.prev.result` 或 `ctx.result` 提供。

执行流程

假定一个管道解析器由两个函数组成，下面的列表表示调用解析器时的执行流程：

1. 管道解析器请求处理程序
2. 函数 1：函数请求处理程序
3. 函数 1：数据源调用

4. 函数 1：函数响应处理程序
5. 函数 2：函数请求处理程序
6. 函数 2：数据源调用
7. 函数 2：函数响应处理程序
8. 管道解析器响应处理程序


GraphQL Request
request()
f1 (x)
request()
response()
Data Source
fn (x)
request()
response()
Data Source
response()
GraphQL Response

非常有用的 APPSYNC_JS 运行时环境内置实用程序

在使用管道解析器时，以下实用工具可为您提供帮助。

ctx.stash

存储区是一个在每个解析器以及函数请求和响应处理程序中提供的对象。相同的存储区实例在单次解析器运行时间内有效。这意味着，您可以使用存储区在请求和响应处理程序之间以及管道解析器中的函数之间传送任意数据。您可以像常规 JavaScript 对象一样测试存储区。

ctx.prev.result

`ctx.prev.result` 表示已在管道中执行的上一个操作的结果。如果上一个操作是管道解析器请求处理程序，则将 `ctx.prev.result` 提供给链中的第一个函数。如果上一个操作是第一个函数，则 `ctx.prev.result` 表示第一个函数的输出，并且可供管道中的第二个函数使用。如果上一个操作是最后一个函数，则 `ctx.prev.result` 表示最后一个函数的输出，并提供给管道解析器响应处理程序。

util.error

`util.error` 实用工具对于引发字段错误很有用。在函数请求或响应处理程序中使用 `util.error` 将立即引发字段错误，这会禁止执行后续的函数。有关更多详细信息和其他 `util.error` 签名，请访问[解析器和函数的 JavaScript 运行时功能](#)。

util.appendError

`util.appendError` 与 `util.error()` 类似，主要区别在于，它不会中断处理程序评估。相反，它指示字段存在错误，但允许评估处理程序并因而返回数据。在函数中使用 `util.appendError` 将不会中断管道的执行流。有关更多详细信息和其他 `util.error` 签名，请访问[解析器和函数的 JavaScript 运行时功能](#)。

runtime.earlyReturn

`runtime.earlyReturn` 函数允许您从任何请求函数中提前返回。在解析器请求处理程序中使用 `runtime.earlyReturn` 将从解析器中返回。从 AWS AppSync 函数请求处理程序中调用它将从该函数中返回，并继续运行到管道中的下一个函数或解析器响应处理程序。

编写管道解析器

管道解析器还具有运行管道中的函数之前和之后的请求和响应处理程序：其请求处理程序在第一个函数的请求之前运行，其响应处理程序在最后一个函数的响应之后运行。解析器请求处理程序可以设置管道

中的函数使用的数据。解析器响应处理程序负责返回映射到 GraphQL 字段输出类型的数据。在以下示例中，解析器请求处理程序定义 `allowedGroups`；返回的数据应属于这些组之一。解析器的函数可以使用该值以请求数据。解析器的响应处理程序进行最终检查并筛选结果，以确保仅返回属于允许的组的项目。

```
import { util } from '@aws-appsync/utils';

/**
 * Called before the request function of the first AppSync function in the pipeline.
 *  @param ctx the context object holds contextual information about the function
 invocation.
 */
export function request(ctx) {
  ctx.stash.allowedGroups = ['admin'];
  ctx.stash.startedAt = util.time.nowISO8601();
  return {};
}
/**
 * Called after the response function of the last AppSync function in the pipeline.
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function response(ctx) {
  const result = [];
  for (const item of ctx.prev.result) {
    if (ctx.stash.allowedGroups.indexOf(item.group) > -1) result.push(item);
  }
  return result;
}
```

编写 AWS AppSync 函数

AWS AppSync 函数允许您编写可在架构中的多个解析器之间重复使用的通用逻辑。例如，您具有一个名为 `QUERY_ITEMS` 的 AWS AppSync 函数，负责从 Amazon DynamoDB 数据源中查询项目。对于要用于查询项目的解析器，只需将函数添加到解析器的管道并提供要使用的查询索引即可。不必重新实施该逻辑。

编写代码

假设您希望在名为 `getPost(id:ID!)` 的字段上附加一个管道解析器，该解析器使用以下 GraphQL 查询从 Amazon DynamoDB 数据源中返回 `Post` 类型：

```
getPost(id:1){
    id
    title
    content
}
```

首先，使用下面的代码将一个简单的解析器附加到 Query.getPost 中。这是一个简单的解析器代码示例。在请求处理程序中没有定义任何逻辑，响应处理程序只是返回最后一个函数的结果。

```
/**
 * Invoked **before** the request handler of the first AppSync function in the
 pipeline.
 * The resolver `request` handler allows to perform some preparation logic
 * before executing the defined functions in your pipeline.
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function request(ctx) {
  return {}
}

/**
 * Invoked **after** the response handler of the last AppSync function in the pipeline.
 * The resolver `response` handler allows to perform some final evaluation logic
 * from the output of the last function to the expected GraphQL field type.
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function response(ctx) {
  return ctx.prev.result
}
```

接下来，定义从数据源中检索 postitem 的 GET_ITEM 函数：

```
import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'

/**
 * Request a single item from the attached DynamoDB table datasource
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
```

```
export function request(ctx) {
 const { id } = ctx.args
 return ddb.get({ key: { id } })
}

/**
 * Returns the result
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function response(ctx) {
 const { error, result } = ctx
 if (error) {
  return util.appendError(error.message, error.type, result)
 }
 return ctx.result
}
```

如果在请求期间出现错误，则该函数的响应处理程序附加一个错误，该错误将在 GraphQL 响应中返回到调用客户端。将 GET_ITEM 函数添加到您的解析器函数列表中。在您执行查询时，GET_ITEM 函数的请求处理程序使用 AWS AppSync 的 DynamoDB 模块提供的实用程序创建 DynamoDBGetItem 请求并将 id 作为键。ddb.get({ key: { id } }) 生成相应的 GetItem 操作：

```
{
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "1" }
    }
}
```

AWS AppSync 使用请求从 Amazon DynamoDB 中获取数据。在返回数据后，将由 GET_ITEM 函数的响应处理程序进行处理，响应处理程序检查错误，然后返回结果。

```
{
  "result" : {
    "id": 1,
    "title": "hello world",
    "content": "<long story>"
  }
}
```

最后，解析器的响应处理程序直接返回结果。

处理错误

如果您的函数在请求期间出现错误，将在函数响应处理程序的 `ctx.error` 中提供该错误。您可以使用 `util.appendError` 实用程序将错误附加到 GraphQL 响应中。您可以使用存储区将错误提供给管道中的其他函数。请参见以下示例：

```
/**
 * Returns the result
 * @param ctx the context object holds contextual information about the function
 invocation.
 */
export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    if (!ctx.stash.errors) ctx.stash.errors = []
    ctx.stash.errors.push(ctx.error)
    return util.appendError(error.message, error.type, result);
  }
  return ctx.result;
}
```

实用程序

AWS AppSync 提供了两个库，可以帮助使用 APPSYNC_JS 运行时环境开发解析器：

- `@aws-appsync/eslint-plugin` - 在开发过程中快速捕获并修复问题。
- `@aws-appsync/utils` - 在代码编辑器中提供类型验证和自动完成功能。

配置 ESLint 插件

ESLint 是一个静态分析代码以快速发现问题的工具。您可以将 ESLint 作为持续集成管道的一部分运行。`@aws-appsync/eslint-plugin` 是一个 ESLint 插件，可以在使用 APPSYNC_JS 运行时环境时捕获代码中的无效语法。通过使用该插件，您可以在开发过程中快速获得有关代码的反馈，而无需将更改推送到云端。

`@aws-appsync/eslint-plugin` 提供了两个可以在开发过程中使用的规则集。

"plugin:@aws-appsync/base" 配置您可以在项目中使用的一组基本规则：

规则	描述
no-async	不支持异步过程和 Promise。
no-await	不支持异步过程和 Promise。
no-classes	不支持类。
no-for	不支持 `for`（支持的 `for-in` 和 `for-of` 除外）
no-continue	不支持 `continue`。
no-generators	不支持生成器。
no-yield	不支持 `yield`。
no-labels	不支持标签。
no-this	不支持 `this` 关键字。
no-try	不支持 try/catch 结构。
no-while	不支持 while 循环。
no-disallowed-unary-operators	不允许使用 `++`、`--` 和 `~` 一元运算符。
no-disallowed-binary-operators	不允许使用 `instanceof` 运算符。
no-promise	不支持异步过程和 Promise。

"plugin:@aws-appsync/recommended" 提供一些额外的规则，但还要求您将 TypeScript 配置添加到项目中。

规则	描述
no-recursion	不允许使用递归函数调用。
no-disallowed-methods	不允许使用某些方法。请参阅参考以了解支持的全套内置函数。

规则	描述
no-function-passing	不允许将函数作为函数参数传递给函数。
no-function-reassign	无法重新分配函数。
no-function-return	函数不能是函数的返回值。

要将插件添加到您的项目中，请按照 ESLint 入门中的安装和使用步骤进行操作。然后，使用项目包管理器（例如 npm、yarn 或 pnpm）在项目中安装插件：

```
$ npm install @aws-appsync/eslint-plugin
```

在 `.eslintrc.{js,yml,json}` 文件中，将 "plugin:@aws-appsync/base" 或 "plugin:@aws-appsync/recommended" 添加到 `extends` 属性中。下面的代码片段是 JavaScript 的基本示例 `.eslintrc` 配置：

```
{
  "extends": ["plugin:@aws-appsync/base"]
}
```

要使用 "plugin:@aws-appsync/recommended" 规则集，请安装所需的依赖项：

```
$ npm install -D @typescript-eslint/parser
```

然后，创建一个 `.eslintrc.js` 文件：

```
{
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaVersion": 2018,
    "project": "./tsconfig.json"
  },
  "extends": ["plugin:@aws-appsync/recommended"]
}
```

捆绑、TypeScript 和源映射

使用库并捆绑您的代码

在您的解析器和函数代码中，您可以使用自定义库和外部库，只要它们符合 APPSYNC_JS 要求即可。这样，就可以在应用程序中重复使用现有的代码。要使用由多个文件定义的库，您必须使用捆绑工具（例如 esbuild）将代码合并到单个文件中，然后可以将该文件保存到您的 AWS AppSync 解析器或函数中。

在捆绑代码时，请记住以下几点：

- APPSYNC_JS 仅支持 ECMAScript 模块 (ESM)。
- @aws-appsync/* 模块集成到 APPSYNC_JS 中，不应将其与您的代码捆绑在一起。
- APPSYNC_JS 运行时环境与 NodeJS 类似，即，不会在浏览器环境中运行代码。
- 您可以包含可选的源映射。不过，不要包含源内容。

  要了解源映射的更多信息，请参阅使用源映射。

例如，要捆绑位于 src/appsync/getPost.resolver.js 中的解析器代码，您可以使用以下 esbuild CLI 命令：

```
$ esbuild --bundle \
--sourcemap=inline \
--sources-content=false \
--target=esnext \
--platform=node \
--format=esm \
--external:@aws-appsync/utils \
--outdir=out/appsync \
 src/appsync/getPost.resolver.js
```

构建代码并使用 TypeScript

TypeScript 是 Microsoft 开发的一种编程语言，它提供 JavaScript 的所有功能以及 TypeScript 类型系统。您可以使用 TypeScript 编写类型安全的代码，并在构建时捕获错误和缺陷，然后再将代码保存到 AWS AppSync 中。@aws-appsync/utils 包是完全类型化的。

APPSYNC_JS 运行时环境不直接支持 TypeScript。在将代码保存到 AWS AppSync 之前，您必须先将 TypeScript 代码转译为 APPSYNC_JS 运行时环境支持的 JavaScript 代码。您可以使用 TypeScript 在

本地集成开发环境 (IDE) 中编写代码，但请注意，您无法在 AWS AppSync 控制台中创建 TypeScript 代码。

首先，请确保在您的项目中安装了 TypeScript。然后，使用 TSConfig 配置 TypeScript 转译设置以与 APPSYNC_JS 运行时环境一起使用。以下是您可以使用的基本 `tsconfig.json` 文件示例：

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "esnext",
    "module": "esnext",
   "noEmit": true,
   "moduleResolution": "node",
  }
}
```

然后，您可以使用 esbuild 等捆绑工具编译和捆绑代码。例如，假定一个项目的 AWS AppSync 代码位于 `src/appsync` 中，您可以使用以下命令编译和捆绑代码：

```
$ esbuild --bundle \
--sourcemap=inline \
--sources-content=false \
--target=esnext \
--platform=node \
--format=esm \
--external:@aws-appsync/utils \
--outdir=out/appsync \
 src/appsync/**/*.ts
```

使用 Amplify codegen

您可以使用 Amplify CLI 生成架构的类型。从 `schema.graphql` 文件所在的目录中，运行以下命令并查看提示以配置 codegen：

```
$  npx @aws-amplify/cli codegen add
```

要在某些情况下（例如，更新架构时）重新生成 codegen，请运行以下命令：

```
$ npx @aws-amplify/cli codegen
```

然后，您可以在解析器代码中使用生成的类型。例如，给定以下架构：

```
type Todo {
  id: ID!
  title: String!
  description: String
}

type Mutation {
  createTodo(title: String!, description: String): Todo
}

type Query {
  listTodos: Todo
}
```

您可以在以下示例 AWS AppSync 函数中使用生成的类型：

```
import { Context, util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'
import { CreateTodoMutationVariables, Todo } from './API' // codegen

export function request(ctx: Context<CreateTodoMutationVariables>) {
 ctx.args.description = ctx.args.description ?? 'created on ' + util.time.nowISO8601()
 return ddb.put<Todo>({ key: { id: util.autoId() }, item: ctx.args })
}

export function response(ctx) {
 return ctx.result as Todo
}
```

在 TypeScript 中使用泛型

您可以将泛型与提供的多种类型一起使用。例如，下面的代码片段是 Todo 类型：

```
export type Todo = {
  __typename: "Todo",
  id: string,
  title: string,
  description?: string | null,
};
```

您可以为使用 Todo 的订阅编写解析器。在您的 IDE 中，类型定义和自动完成提示将指导您正确使用 toSubscriptionFilter 转换实用程序：

```
import { util, Context, extensions } from '@aws-appsync/utils'
import { Todo } from './API'

export function request(ctx: Context) {
  return {}
}

export function response(ctx: Context) {
  const filter = util.transform.toSubscriptionFilter<Todo>({
    title: { beginsWith: 'hello' },
    description: { contains: 'created' },
  })
  extensions.setSubscriptionFilter(filter)
  return null
}
```

检查您的包

您可以导入 esbuild-plugin-eslint 插件以自动检查您的包。然后，您可以提供启用 ESLint 功能的 plugins 值以启用该插件。下面是在名为 build.mjs 的文件中使用 esbuild JavaScript API 的片段：

```
/* eslint-disable */
import { build } from 'esbuild'
import eslint from 'esbuild-plugin-eslint'
import glob from 'glob'
const files = await glob('src/**/*.ts')

await build({
  format: 'esm',
  target: 'esnext',
  platform: 'node',
  external: ['@aws-appsync/utils'],
  outdir: 'dist/',
  entryPoints: files,
  bundle: true,
  plugins: [eslint({ useEslintrc: true })],
})
```

使用源映射

您可以为 JavaScript 代码提供内联源映射 (sourcemap)。如果您捆绑 JavaScript 或 TypeScript 代码，并希望在日志和运行时 JavaScript 错误消息中查看对输入源文件的引用，源映射是非常有用的。

您的 sourcemap 必须出现在代码末尾。它是由采用以下格式的单个注释行定义的：

```
//# sourceMappingURL=data:application/json;base64,<base64 encoded string>
```

示例如下：

```
//# sourceMappingURL=data:application/
json;base64,ewogICJ2ZXJzaW9uIjogMywKICAic291cmNlcyI6IFsibGliLmpzIiwgImNvZGUuanMiXSwKICAibWFwcGl
```

可以使用 esbuild 创建源映射。以下示例说明了如何使用 esbuild JavaScript API 在构建和捆绑代码时包含内联源映射：

```
/* eslint-disable */
import { build } from 'esbuild'
import eslint from 'esbuild-plugin-eslint'
import glob from 'glob'
const files = await glob('src/**/*.ts')

await build({
  sourcemap: 'inline',
  sourcesContent: false,

  format: 'esm',
  target: 'esnext',
  platform: 'node',
  external: ['@aws-appsync/utils'],
  outdir: 'dist/',
  entryPoints: files,
  bundle: true,
  plugins: [eslint({ useEslintrc: true })],
})
```

特别是，sourcemap 和 sourcesContent 选项指定应在每个构建末尾的行中添加源映射，但源映射不应包含源内容。作为惯例，我们建议不要在您的 sourcemap 中包含源内容。您可以将 sources-content 设置为 false 以在 esbuild 中禁用该功能。

要说明源映射的工作方式，请查看以下示例，其中解析器代码引用帮助程序库中的帮助程序函数。该代码在解析器代码和帮助程序库中包含日志语句：

./src/default.resolver.ts（您的解析器）

```
import { Context } from '@aws-appsync/utils'
import { hello, logit } from './helper'

export function request(ctx: Context) {
  console.log('start >')
  logit('hello world', 42, true)
  console.log('< end')
  return 'test'
}

export function response(ctx: Context): boolean {
  hello()
  return ctx.prev.result
}
```

.src/helper.ts（帮助程序文件）

```
export const logit = (...rest: any[]) => {
  // a special logger
  console.log('[logger]', ...rest.map((r) => `<${r}>`))
}

export const hello = () => {
  // This just returns a simple sentence, but it could do more.
  console.log('i just say hello..')
}
```

在您构建并捆绑解析器文件时，您的解析器代码将包含内联源映射。在您的解析器运行时，将在 CloudWatch 日志中出现以下条目：

看一下 CloudWatch 日志中的条目，您会注意到这两个文件的功能已捆绑在一起并且同时运行。每个文件的原始文件名也清晰地反映在日志中。

测试

在将代码保存到解析器或函数之前，您可以通过 EvaluateCode API 命令使用模拟数据远程测试解析器和函数处理程序。要开始使用该命令，请确保您已将 appsync:evaluatecode 权限添加到您的策略中。例如：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "appsync:evaluateCode",
            "Resource": "arn:aws:appsync:<region>:<account>:*"
        }
    ]
}
```

您可以通过 AWS CLI 或 AWS SDK 使用该命令。例如，要使用 CLI 测试代码，只需指向您的文件，提供上下文并指定要评估的处理程序：

```
aws appsync evaluate-code \
  --code file://code.js \
  --function request \
  --context file://context.json \
  --runtime name=APPSYNC_JS,runtimeVersion=1.0.0
```

响应包含一个 evaluationResult，其中包含处理程序返回的负载。它还包含一个 logs 对象，其中保存处理程序在评估期间生成的日志列表。这样，就可以轻松调试代码执行，并查看有关评估的信息以帮助排除故障。例如：

```
{
    "evaluationResult": "{\"operation\":\"PutItem\",\"key\":{\"id\":{\"S\":\"record-id
\"}},\"attributeValues\":{\"owner\":{\"S\":\"John doe\"},\"expectedVersion\":{\"N\":2},
\"authorId\":{\"S\":\"Sammy Davis\"}}}",
    "logs": [
        "INFO - code.js:5:3: \"current id\" \"record-id\"",
        "INFO - code.js:9:3: \"request evaluated\""
```

```
    ]
}
```

可以将评估结果解析为 JSON，其中提供：

```
{
  "operation": "PutItem",
  "key": {
    "id": {
      "S": "record-id"
    }
  },
  "attributeValues": {
    "owner": {
      "S": "John doe"
    },
    "expectedVersion": {
      "N": 2
    },
    "authorId": {
      "S": "Sammy Davis"
    }
  }
}
```

通过使用 SDK，您可以轻松合并测试套件中的测试以验证代码行为。此处的示例使用 Jest 测试框架，但任何测试套件都有效。以下代码片段显示假设的验证运行。请注意，我们希望评估响应是有效的 JSON ，因此，我们使用 `JSON.parse` 从字符串响应中检索 JSON：

```
const AWS = require('aws-sdk')
const fs = require('fs')
const client = new AWS.AppSync({ region: 'us-east-2' })
const runtime = {name:'APPSYNC_JS',runtimeVersion:'1.0.0')

test('request correctly calls DynamoDB', async () => {
  const code = fs.readFileSync('./code.js', 'utf8')
  const context = fs.readFileSync('./context.json', 'utf8')
  const contextJSON = JSON.parse(context)

  const response = await client.evaluateCode({ code, context, runtime, function:
 'request' }).promise()
  const result = JSON.parse(response.evaluationResult)
```

```
  expect(result.key.id.S).toBeDefined()
  expect(result.attributeValues.firstname.S).toEqual(contextJSON.arguments.firstname)
})
```

这会产生以下结果：

```
Ran all test suites.
> jest

PASS ./index.test.js
# request correctly calls DynamoDB (543 ms)
Test Suites: 1 passed, 1 total
Tests: 1 passed, 1 total
Snapshots: 0 totalTime: 1.511 s, estimated 2 s
```

从 VTL 迁移到 JavaScript

AWS AppSync 允许您使用 VTL 或 JavaScript 为解析器和函数编写业务逻辑。通过使用这两种语言，您可以编写逻辑以指示 AWS AppSync 服务如何与数据源进行交互。在使用 VTL 时，您可以编写评估结果必须为有效 JSON 编码字符串的映射模板。在使用 JavaScript 时，您可以编写返回对象的请求和响应处理程序。您不会返回 JSON 编码的字符串。

例如，采用以下 VTL 映射模板以获取 Amazon DynamoDB 项目：

```
{
    "operation": "GetItem",
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
    }
}
```

`$util.dynamodb.toDynamoDBJson` 实用程序返回 JSON 编码的字符串。如果 `$ctx.args.id` 设置为 `<id>`，则模板评估结果为有效的 JSON 编码字符串：

```
{
    "operation": "GetItem",
    "key": {
        "id": {"S": "<id>"},
    }
```

```
}
```

在使用 JavaScript 时，您不需要在代码中输出原始 JSON 编码字符串，并且不需要使用像 `toDynamoDBJson` 这样的实用程序。上述映射模板的等效示例是：

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: {id: util.dynamodb.toDynamoDB(ctx.args.id)}
  };
}
```

另一种方法是使用 `util.dynamodb.toMapValues`，这是处理值对象的建议方法：

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}
```

它的评估结果为：

```
{
  "operation": "GetItem",
  "key": {
    "id": {
      "S": "<id>"
    }
  }
}
```

> **Note**
>
> 我们建议将 DynamoDB 模块与 DynamoDB 数据源一起使用：
>
> ```
> import * as ddb from '@aws-appsync/utils/dynamodb'
> ```

```
export function request(ctx) {
 ddb.get({ key: { id: ctx.args.id } })
}
```

再举一个例子，采用以下映射模板将项目放入 Amazon DynamoDB 数据源中：

```
{
    "operation" : "PutItem",
    "key" : {
        "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

在评估后，该映射模板字符串必须生成有效的 JSON 编码字符串。在使用 JavaScript 时，您的代码直接返回请求对象：

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { id = util.autoId(), ...values } = ctx.args;
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ id }),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}
```

它的评估结果为：

```
{
  "operation": "PutItem",
  "key": {
    "id": { "S": "2bff3f05-ff8c-4ed8-92b4-767e29fc4e63" }
  },
  "attributeValues": {
    "firstname": { "S": "Shaggy" },
    "age": { "N": 4 }
  }
}
```

Note

我们建议将 DynamoDB 模块与 DynamoDB 数据源一起使用：

```
import { util } from '@aws-appsync/utils'
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 const { id = util.autoId(), ...item } = ctx.args
 return ddb.put({ key: { id }, item })
}
```

在直接数据源访问和通过 Lambda 数据源代理之间进行选择

通过使用 AWS AppSync 和 APPSYNC_JS 运行时环境，您可以编写自己的代码，以通过 AWS AppSync 函数访问数据源来实施自定义业务逻辑。这样，您可以轻松直接与数据源（如 Amazon DynamoDB、Aurora Serverless、OpenSearch Service、HTTP API 和其他 AWS 服务）交互，而无需部署额外的计算服务或基础设施。AWSAppSync 还可以配置 Lambda 数据源以轻松与 AWS Lambda 函数进行交互。通过使用 Lambda 数据源，您可以使用 AWS Lambda 的全套功能运行复杂的业务逻辑以解析 GraphQL 请求。在大多数情况下，直接连接到目标数据源的 AWS AppSync 函数将提供您所需的所有功能。在您需要实施 APPSYNC_JS 运行时环境不支持的复杂业务逻辑时，您可以将 Lambda 数据源作为代理以与目标数据源进行交互。

	直接数据源集成	将 Lambda 数据源作为代理
使用案例	AWS AppSync functions interact directly with API data sources.	AWS AppSync functions call Lambdas that interact with API data sources.
Runtime	*APPSYNC_JS* (JavaScript)	任何支持的 Lambda 运行时环境
Maximum size of code	每个 AWS AppSync 函数 32,000 个字符	每个 Lambda 50 MB（压缩，用于直接上传）
External modules	有限 - 仅 APPSYNC_JS 支持的功能	是

Call any AWS service	是 - 使用 AWS AppSync HTTP 数据源	是 - 使用 AWS SDK
Access to the request header	是	是
Network access	否	是
File system access	否	是
Logging and metrics	是	是
Build and test entirely within AppSync	是	否
Cold start	否	否 - 使用预置并发
Auto-scaling	是 - 由 AWS AppSync 透明实施	是 - 根据 Lambda 中的配置
Pricing	无额外费用	收取 Lambda 使用费用

直接与目标数据源集成的 AWS AppSync 函数非常适合以下使用案例：

- 与 Amazon DynamoDB、Aurora Serverless 和 OpenSearch Service 交互
- 与 HTTP API 交互并传递传入标头
- 使用 HTTP 数据源与 AWS 服务交互（AWS AppSync 使用提供的数据源角色自动对请求进行签名）
- 在访问数据源之前实施访问控制
- 在完成请求之前对检索的数据实施筛选
- 按顺序执行解析器管道中的 AWS AppSync 函数以实施简单编排
- 控制查询和变更中的缓存和订阅连接。

将 Lambda 数据源作为代理的 AWS AppSync 函数非常适合以下使用案例：

- 使用 JavaScript 或 Velocity 模板语言 (VTL) 以外的语言
- 调整和控制 CPU 或内存以优化性能
- 导入第三方库或要求使用 APPSYNC_JS 中不支持的功能
- 发出多个网络请求和/或获取文件系统访问权限以完成查询

- 使用批处理配置对请求进行批处理。

解析器上下文对象引用

AWS AppSync 定义了一组用于处理请求和响应处理程序的变量和函数。这样，就可以更轻松地通过 GraphQL 对数据进行逻辑操作。本文档介绍了这些函数并提供了示例。

使用 `context`

请求和响应处理程序的 `context` 参数是一个对象，它保存解析器调用的所有上下文信息。它具有以下结构：

```
type Context = {
  arguments: any;
  args: any;
  identity: Identity;
  source: any;
  error?: {
    message: string;
    type: string;
  };
  stash: any;
  result: any;
  prev: any;
  request: Request;
  info: Info;
};
```

> **Note**
>
> 您经常会发现 `context` 对象是作为 `ctx` 引用的。

`context` 对象中的每个字段定义如下：

`context` 字段

`arguments`

包含该字段的所有 GraphQL 参数的映射。

identity

包含有关调用方的信息的对象。有关该字段结构的更多信息，请参阅身份。

source

包含父字段解析的映射。

stash

存储区是一个在每个解析器以及函数处理程序中提供的对象。相同的存储区对象在单次解析器运行时间内有效。这意味着，您可以使用存储区在请求和响应处理程序之间以及管道解析器中的函数之间传送任意数据。

> **Note**
>
> 您无法删除或替换整个存储区，但可以添加、更新、删除和读取存储区属性。

您可以修改以下代码示例之一，以将项目添加到存储区中：

```
//Example 1
ctx.stash.newItem = { key: "something" }

//Example 2
Object.assign(ctx.stash, {key1: value1, key2: value})
```

您可以修改以下代码，以从存储区中删除项目：

```
delete ctx.stash.key
```

result

此解析器结果的容器。该字段仅适用于响应处理程序。

例如，如果要解析以下查询的 author 字段：

```
query {
    getPost(id: 1234) {
        postId
```

```
        title
        content
        author {
            id
            name
        }
    }
}
```

然后，在评估响应处理程序时，可以使用完整的 context 变量：

```
{
  "arguments" : {
    id: "1234"
  },
  "source": {},
  "result" : {
      "postId": "1234",
      "title": "Some title",
      "content": "Some content",
      "author": {
        "id": "5678",
        "name": "Author Name"
      }
  },
  "identity" : {
      "sourceIp" : ["x.x.x.x"],
      "userArn" : "arn:aws:iam::123456789012:user/appsync",
      "accountId" : "666666666666",
      "user" : "AIDAAAAAAAAAAAAAAAAA"
  }
}
```

prev.result

在管道解析器中执行的任何以前操作的结果。

如果上一个操作是管道解析器的请求处理程序，则 ctx.prev.result 表示该评估结果，并提供给管道中的第一个函数。

如果上一个操作是第一个函数，则 ctx.prev.result 表示第一个函数响应处理程序的评估结果，并提供给管道中的第二个函数。

如果上一个操作是最后一个函数，则 `ctx.prev.result` 表示最后一个函数的评估结果，并提供给管道解析器的响应处理程序。

`info`

包含有关 GraphQL 请求的信息的对象。有关该字段的结构，请参阅信息。

求同

`identity` 部分包含调用方的相关信息。此部分的形态取决于您的 AWS AppSync API 的授权类型。

有关 AWS AppSync 安全选项的更多信息，请参阅授权和身份验证。

`API_KEY` 授权

不填充 `identity` 字段。

`AWS_LAMBDA` 授权

`identity` 采用以下格式：

```
type AppSyncIdentityLambda = {
  resolverContext: any;
};
```

`identity` 包含 `resolverContext` 密钥，其中包含为请求授权的 Lambda 函数返回的相同 `resolverContext` 内容。

`AWS_IAM` 授权

`identity` 采用以下格式：

```
type AppSyncIdentityIAM = {
  accountId: string;
  cognitoIdentityPoolId: string;
  cognitoIdentityId: string;
  sourceIp: string[];
  username: string;
  userArn: string;
  cognitoIdentityAuthType: string;
  cognitoIdentityAuthProvider: string;
};
```

AMAZON_COGNITO_USER_POOLS 授权

identity 采用以下格式：

```
type AppSyncIdentityCognito = {
  sourceIp: string[];
  username: string;
  groups: string[] | null;
  sub: string;
  issuer: string;
  claims: any;
  defaultAuthStrategy: string;
};
```

每个字段的定义如下所示：

accountId

调用方的 AWS 账户 ID。

claims

用户拥有的声明。

cognitoIdentityAuthType

根据身份类型确定经过身份验证或未经身份验证。

cognitoIdentityAuthProvider

外部身份提供程序信息的逗号分隔列表，用于获取对请求进行签名时使用的凭证。

cognitoIdentityId

调用方的 Amazon Cognito 身份 ID。

cognitoIdentityPoolId

与调用方关联的 Amazon Cognito 身份池 ID。

defaultAuthStrategy

此调用方的默认授权策略（ALLOW 或 DENY）。

issuer

令牌发布者。

sourceIp

AWS AppSync 收到的调用方的源 IP 地址。如果请求不包含 x-forwarded-for 标头，则源 IP 值仅包含来自 TCP 连接的单个 IP 地址。如果请求中包含 x-forwarded-for 标头，那么源 IP 将是来自 x-forwarded-for 标头的 IP 地址列表，以及来自 TCP 连接的 IP 地址。

sub

经过验证的用户的 UUID。

user

IAM 用户。

userArn

IAM 用户的 Amazon 资源名称 (ARN)。

username

已验证的用户的用户名。对于 AMAZON_COGNITO_USER_POOLS 授权，用户名的值是 cognito:username 属性的值。对于 AWS_IAM 授权，username 值是 AWS 用户主体的值。如果您将 IAM 授权与从 Amazon Cognito 身份池提供的凭证一起使用，我们建议您使用 cognitoIdentityId。

访问请求标头

AWS AppSync 支持从客户端传递自定义标头，并使用 ctx.request.headers 在 GraphQL 解析器中访问这些标头。然后，您可以使用标头值执行操作，例如，将数据插入到数据源或进行授权检查。您可以在命令行中使用 $curl 将一个或多个请求标头与 API 密钥一起使用，如以下示例中所示：

单标头示例

假设您设置一个 custom 标头，值为 nadia，如下所示：

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:nadia" -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}"}' https://<ENDPOINT>/graphql
```

然后可使用 ctx.request.headers.custom 访问它。例如，对于 DynamoDB，它可能位于以下代码中：

```
"custom": util.dynamodb.toDynamoDB(ctx.request.headers.custom)
```

多标头示例

您也可以在单个请求中传递多个标头，并在解析器处理程序中访问这些标头。例如，如果为 custom 标头设置了两个值：

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:bailey" -H "custom:nadia"
 -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo
\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}"}'
 https://<ENDPOINT>/graphql
```

它们可以作为一个数组访问，例如 ctx.request.headers.custom[1]。

> **Note**
>
> AWS AppSync 不会在 ctx.request.headers 中公开 Cookie 标头。

访问请求自定义域名

AWS AppSync 支持配置自定义域，您可以使用该域访问 API 的 GraphQL 和实时终端节点。在使用自定义域名发出请求时，您可以使用 ctx.request.domainName 获取域名。

在使用默认 GraphQL 终端节点域名时，值为 null。

Info

info 部分包含有关 GraphQL 请求的信息。该部分采用以下格式：

```
type Info = {
  fieldName: string;
  parentTypeName: string;
  variables: any;
  selectionSetList: string[];
  selectionSetGraphQL: string;
};
```

每个字段的定义如下所示：

fieldName

当前正在解析的字段的名称。

parentTypeName

当前正在解析的字段的父类型的名称。

variables

保留传递到 GraphQL 请求的所有变量的映射。

selectionSetList

GraphQL 选择集中字段的列表表示形式。具有别名的字段仅按别名进行引用，而不按字段名称进行引用。以下示例对此详细进行了介绍。

selectionSetGraphQL

选择集的字符串表示形式，格式为 GraphQL 架构定义语言 (SDL)。尽管片段不会合并到选择集中，但会保留内联片段，如以下示例中所示。

> **Note**
>
> `JSON.stringify` 不会在字符串序列化中包含 `selectionSetGraphQL` 和 `selectionSetList`。您必须直接引用这些属性。

例如，如果您要解析以下查询的 `getPost` 字段：

```
query {
  getPost(id: $postId) {
    postId
    title
    secondTitle: title
    content
    author(id: $authorId) {
      authorId
      name
    }
    secondAuthor(id: "789") {
      authorId
    }
    ... on Post {
```

```
      inlineFrag: comments: {
        id
      }
    }
    ... postFrag
  }
}

fragment postFrag on Post {
  postFrag: comments: {
    id
  }
}
```

在处理处理程序时可使用的完整 ctx.info 变量可能是：

```
{
  "fieldName": "getPost",
  "parentTypeName": "Query",
  "variables": {
    "postId": "123",
    "authorId": "456"
  },
  "selectionSetList": [
    "postId",
    "title",
    "secondTitle"
    "content",
    "author",
    "author/authorId",
    "author/name",
    "secondAuthor",
    "secondAuthor/authorId",
    "inlineFragComments",
    "inlineFragComments/id",
    "postFragComments",
    "postFragComments/id"
  ],
  "selectionSetGraphQL": "{\n  getPost(id: $postId) {\n    postId\n    title\n
 secondTitle: title\n    content\n    author(id: $authorId) {\n      authorId\n
 name\n    }\n    secondAuthor(id: \"789\") {\n      authorId\n    }\n    ... on Post
 {\n      inlineFrag: comments {\n        id\n      }\n    }\n    ... postFrag\n  }\n}"
}
```

selectionSetList 仅公开属于当前类型的字段。如果当前类型是接口或联合，则仅公开属于该接口的选定字段。例如，给定以下架构：

```
type Query {
    node(id: ID!): Node
}

interface Node {
    id: ID
}

type Post implements Node {
    id: ID
    title: String
    author: String
}

type Blog implements Node {
    id: ID
    title: String
    category: String
}
```

以及以下查询：

```
query {
    node(id: "post1") {
        id
        ... on Post {
            title
        }

        ... on Blog {
            title
        }
    }
}
```

如果在进行 Query.node 字段解析时调用 ctx.info.selectionSetList，则仅公开 id：

```
"selectionSetList": [
    "id"
```

```
]
```

解析器和函数的 JavaScript 运行时功能

APPSYNC_JS 运行时环境提供与 ECMAScript (ES) 6.0 版类似的功能。它支持该版本的一部分功能，并提供一些不属于 ES 规范的额外方法（实用程序）。以下主题列出了所有支持的语言功能。

> **Note**
>
> 目前，该参考仅适用于运行时环境版本 1.0.0。

主题

支持的运行时功能

以下几节介绍了支持的 APPSYNC_JS 运行时功能集。

核心功能

支持以下核心功能。

类型

支持以下类型：

- 数字
- 字符串
- 布尔值
- objects
- 数组
- 函数

运算符

支持的运算符，其中包括：

- 标准数学运算符（+、-、/、%、* 等）
- Null 合并运算符 (??)
- 可选链 (?.)
- 按位运算符
- void 和 typeof 运算符

不支持以下运算符：

- 一元运算符（++、-- 和 ~）
- in 运算符

> **Note**
>
> 可以使用 Object.hasOwn 运算符检查指定的属性是否位于指定的对象中。

语句

支持以下语句：

- const
- let
- var
- break

- `else`
- `for-in`
- `for-of`
- `if`
- `return`
- `switch`
- spread syntax

不支持以下语句：

- `catch`
- `continue`
- `do-while`
- `finally`
- `for(initialization; condition; afterthought)`

> **Note**
>
> 例外情况是 `for-in` 和 `for-of` 表达式，支持这些表达式。

- `throw`
- `try`
- `while`
- 带标签的语句

文本

支持以下 ES 6 模板文本：

- 多行字符串
- 表达式插值
- 嵌套模板

函数

支持以下函数语法：

- 支持函数声明。
- 支持 ES 6 箭头函数。
- 支持 ES 6 剩余参数语法。

严格模式

默认情况下，函数在严格模式下运行，因此您无需在函数代码中添加 `use_strict` 语句。无法对其进行更改。

原语对象

支持以下 ES 基元对象及其函数。

Object

支持以下对象：

- `Object.assign()`
- `Object.entries()`
- `Object.hasOwn()`
- `Object.keys()`
- `Object.values()`
- `delete`

字符串

支持以下字符串：

- `String.prototype.length()`
- `String.prototype.charAt()`
- `String.prototype.concat()`
- `String.prototype.endsWith()`
- `String.prototype.indexOf()`
- `String.prototype.lastIndexOf()`
- `String.raw()`

- String.prototype.replace()

> **Note**
>
> 不支持正则表达式。

- String.prototype.replaceAll()

> **Note**
>
> 不支持正则表达式。

- String.prototype.slice()
- String.prototype.split()
- String.prototype.startsWith()
- String.prototype.toLowerCase()
- String.prototype.toUpperCase()
- String.prototype.trim()
- String.prototype.trimEnd()
- String.prototype.trimStart()

数字

支持以下数字：

- Number.isFinite
- Number.isNaN

内置对象和函数

支持以下函数和对象。

数学

支持以下数学函数：

- Math.random()

- Math.min()
- Math.max()
- Math.round()
- Math.floor()
- Math.ceil()

数组

支持以下数组方法：

- Array.prototype.length
- Array.prototype.concat()
- Array.prototype.fill()
- Array.prototype.flat()
- Array.prototype.indexOf()
- Array.prototype.join()
- Array.prototype.lastIndexOf()
- Array.prototype.pop()
- Array.prototype.push()
- Array.prototype.reverse()
- Array.prototype.shift()
- Array.prototype.slice()
- Array.prototype.sort()

> **Note**
>
> Array.prototype.sort() 不支持参数。

- Array.prototype.splice()
- Array.prototype.unshift()
- Array.prototype.forEach()
- Array.prototype.map()
- Array.prototype.flatMap()

- `Array.prototype.filter()`
- `Array.prototype.reduce()`
- `Array.prototype.reduceRight()`
- `Array.prototype.find()`
- `Array.prototype.some()`
- `Array.prototype.every()`
- `Array.prototype.findIndex()`
- `Array.prototype.findLast()`
- `Array.prototype.findLastIndex()`
- `delete`

控制台

可以使用控制台对象进行调试。在实时执行查询期间，控制台日志/错误语句将发送到 Amazon CloudWatch Logs（如果启用了日志记录）。在使用 `evaluateCode` 评估代码期间，将在命令响应中返回日志语句。

- `console.error()`
- `console.log()`

JSON

支持以下 JSON 方法：

- `JSON.parse()`

> **Note**
>
> 如果解析的字符串不是有效的 JSON，则返回空字符串。

- `JSON.stringify()`

函数

- 不支持 `apply`、`bind` 和 `call` 方法。
- 不支持函数构造函数。

- 不支持将函数作为参数传递。
- 不支持递归函数调用。

Promise

不支持异步过程，也不支持 Promise。

> **Note**
>
> 在 AWS AppSync 上的 APPSYNC_JS 运行时环境中不支持网络和文件系统访问。AWS AppSync 根据 AWS AppSync 解析器或 AWS AppSync 函数发出的请求处理所有 I/O 操作。

全局变量

支持以下全局约束：

- `NaN`
- `Infinity`
- `undefined`
- `util`
- `extensions`
- `runtime`

错误类型

不支持使用 `throw` 引发错误。您可以使用 `util.error()` 函数返回错误。您可以使用 `util.appendError` 函数在 GraphQL 响应中包含错误。

有关更多信息，请参阅错误实用程序。

内置实用程序

`util` 变量包含帮助您处理数据的常规实用程序方法。除非另行指定，否则所有实用程序均使用 UTF-8 字符集。

编码实用程序

编码实用程序列表

`util.urlEncode(String)`

将输入字符串作为 application/x-www-form-urlencoded 编码字符串返回。

`util.urlDecode(String)`

将 application/x-www-form-urlencoded 编码的字符串解码回未编码的形式。

`util.base64Encode(string) : string`

将输入编码为 base64 编码字符串。

`util.base64Decode(string) : string`

对 base64 编码字符串中的数据进行解码。

ID 生成实用程序

ID 生成实用程序列表

`util.autoId()`

返回 128 位随机生成的 UUID。

`util.autoUlid()`

返回一个 128 位随机生成的 ULID（可按字典排序的通用唯一标识符）。

`util.autoKsuid()`

返回一个 128 位随机生成的 KSUID（K 可排序唯一标识符），它使用 Base62 编码为长度为 27 的字符串。

错误实用程序

错误实用程序列表

`util.error(String, String?, Object?, Object?)`

引发自定义错误。如果模板检测到请求或调用结果的错误，可用于请求或响应映射模板中。此外，还可以指定 `errorType`、`data` 和 `errorInfo` 字段。将在 GraphQL 响应中 `data` 内部对应的 `error` 块中添加 `errors` 值。

> **Note**
>
> `data` 将根据查询选择集进行筛选。将在 GraphQL 响应中 `errorInfo` 内部对应的 `error` 块中添加 `errors` 值。
> `errorInfo` 不会根据查询选择集进行筛选。

`util.appendError(String, String?, Object?, Object?)`

追加自定义错误。如果模板检测到请求或调用结果的错误，可用于请求或响应映射模板中。此外，还可以指定 `errorType`、`data` 和 `errorInfo` 字段。与 `util.error(String, String?, Object?, Object?)` 不同，不会中断模板评估，因此，可以向调用方返回数据。将在 GraphQL 响应中 `data` 内部对应的 `error` 块中添加 `errors` 值。

> **Note**
>
> `data` 将根据查询选择集进行筛选。将在 GraphQL 响应中 `errorInfo` 内部对应的 `error` 块中添加 `errors` 值。
> `errorInfo` 不会根据查询选择集进行筛选。

类型和模式匹配实用程序

类型和模式匹配实用程序列表

`util.matches(String, String) : Boolean`

如果在第一个参数中指定的模式与第二个参数中提供的数据匹配，则返回 true。模式必须为正则表达式，例如 `util.matches("a*b", "aaaaab")`。此功能以模式为基础，您可参考其他文档，进一步了解此内容。

`util.authType()`

返回描述请求使用的多重身份验证类型的字符串，即，返回“IAM Authorization”、“User Pool Authorization”、“Open ID Connect Authorization”或“API Key Authorization”。

返回值行为实用程序

返回值行为实用程序列表

`util.escapeJavaScript(String)`

将输入字符串作为 JavaScript 转义字符串返回。

解析器授权实用程序

解析器授权实用程序列表

`util.unauthorized()`

针对被解析的字段引发 `Unauthorized`。可以在请求或响应映射模板中使用该实用程序，以确定是否允许调用方解析该字段。

内置模块

模块是 `APPSYNC_JS` 运行时环境的一部分，模块提供实用程序以帮助编写 JavaScript 解析器和函数。

DynamoDB 模块函数

在与 DynamoDB 数据源交互时，DynamoDB 模块函数提供增强的体验。您可以使用这些函数向 DynamoDB 数据源发出请求，而无需添加类型映射。

模块是使用 `@aws-appsync/utils/dynamodb` 导入的：

```
// Modules are imported using @aws-appsync/utils/dynamodb
import * as ddb from '@aws-appsync/utils/dynamodb';
```

函数

函数列表

`get<T>(payload: GetInput): DynamoDBGetItemRequest`

> **Tip**
>
> 有关 GetInput 的信息，请参阅the section called “输入”。

生成一个 `DynamoDBGetItemRequest` 对象以向 DynamoDB 发出 GetItem 请求。

```
import { get } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 return get({ key: { id: ctx.args.id } });
}
```

`put<T>(payload): DynamoDBPutItemRequest`

生成一个 `DynamoDBPutItemRequest` 对象以向 DynamoDB 发出 PutItem 请求。

```
import * as ddb from '@aws-appsync/utils/dynamodb'

export function request(ctx) {
 return ddb.put({ key: { id: util.autoId() }, item: ctx.args });
}
```

`remove<T>(payload): DynamoDBDeleteItemRequest`

生成一个 `DynamoDBDeleteItemRequest` 对象以向 DynamoDB 发出 DeleteItem 请求。

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 return ddb.remove({ key: { id: ctx.args.id } });
}
```

`scan<T>(payload): DynamoDBScanRequest`

生成一个 `DynamoDBScanRequest` 以向 DynamoDB 发出 Scan 请求。

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const { limit = 10, nextToken } = ctx.args;
 return ddb.scan({ limit, nextToken });
}
```

`sync<T>(payload): DynamoDBSyncRequest`

生成一个 DynamoDBSyncRequest 对象以发出 Sync 请求。该请求仅接收自上次查询以来更改的数据（增量更新）。只能向版本控制的 DynamoDB 数据源发出请求。

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const { limit = 10, nextToken, lastSync } = ctx.args;
 return ddb.sync({ limit, nextToken, lastSync });
}
```

`update<T>(payload): DynamoDBUpdateItemRequest`

生成一个 DynamoDBUpdateItemRequest 对象以向 DynamoDB 发出 UpdateItem 请求。

操作

通过使用操作帮助程序，您可以在更新期间对部分数据执行特定的操作。要开始使用，请从 `@aws-appsync/utils/dynamodb` 中导入 `operations`：

```
// Modules are imported using operations
import {operations} from '@aws-appsync/utils/dynamodb';
```

操作列表

`add<T>(payload)`

在更新 DynamoDB 时添加新属性项目的帮助程序函数。

示例

要使用 ID 值将地址（街道、城市和邮政编码）添加到现有 DynamoDB 项目中，请运行以下命令：

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const updateObj = {
  address: operations.add({
   street1: '123 Main St',
   city: 'New York',
   zip: '10001',
  }),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

append <T>(payload)

将负载附加到 DynamoDB 中的现有列表的帮助程序函数。

示例

要在更新期间将新添加的好友 ID (newFriendIds) 添加到现有好友列表 (friendsIds) 后面，请运行以下命令：

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const newFriendIds = [101, 104, 111];
 const updateObj = {
  friendsIds: operations.append(newFriendIds),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

decrement (by?)

在更新 DynamoDB 时减少项目中的现有属性值的帮助程序函数。

示例

要将好友计数器 (friendsCount) 减少 10，请运行以下命令：

```
import { update, operations } from '@aws-appsync/utils/dynamodb';
```

```
export function request(ctx) {
 const updateObj = {
  friendsCount: operations.decrement(10),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

`increment (by?)`

在更新 DynamoDB 时增加项目中的现有属性值的帮助程序函数。

示例

要将好友计数器 (`friendsCount`) 增加 10，请运行以下命令：

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const updateObj = {
  friendsCount: operations.increment(10),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

`prepend <T>(payload)`

在 DynamoDB 中的现有列表前面添加内容的帮助程序函数。

示例

要在更新期间将新添加的好友 ID (`newFriendIds`) 添加到现有好友列表 (`friendsIds`) 前面，请运行以下命令：

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const newFriendIds = [101, 104, 111];
 const updateObj = {
  friendsIds: operations.prepend(newFriendIds),
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

`replace <T>(payload)`

更新 DynamoDB 中的项目时替换现有属性的帮助程序函数。在您希望更新属性中的整个对象或子对象而不仅仅是负载中的键时，这是非常有用的。

示例

要替换 `info` 对象中的地址（街道、城市和邮政编码），请运行以下命令：

```
import { update, operations } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const updateObj = {
  info: {
   address: operations.replace({
    street1: '123 Main St',
    city: 'New York',
    zip: '10001',
   }),
  },
 };
 return update({ key: { id: 1 }, update: updateObj });
}
```

`updateListItem <T>(payload, index)`

替换列表中的项目的帮助程序函数。

示例

在更新范围 (`newFriendIds`) 中，该示例使用 `updateListItem` 更新列表 (`friendsIds`) 中的第二项（索引 1，新 ID 102）和第三项（索引 2，新 ID 112）的 ID 值。

```
import { update, operations as ops } from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 const newFriendIds = [
  ops.updateListItem('102', 1), ops.updateListItem('112', 2)
 ];
 const updateObj = { friendsIds: newFriendIds };
 return update({ key: { id: 1 }, update: updateObj });
}
```

输入

输入列表

Type GetInput<T>

```
GetInput<T>: {
    consistentRead?: boolean;
    key: DynamoDBKey<T>;
}
```

类型声明

- consistentRead?: boolean（可选）

  一个可选的布尔值，用于指定您是否要在 DynamoDB 中执行强一致性读取。

- key: DynamoDBKey<T>（必需）

  一个必需的参数，用于指定 DynamoDB 中的项目的键。DynamoDB 项目可能具有单个哈希键或者哈希键和排序键。

Type PutInput<T>

```
PutInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T> | null;
    customPartitionKey?: string;
    item: Partial<T>;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
}
```

类型声明

- _version?: number（可选）
- condition?: DynamoDBFilterObject<T> | null（可选）

  在将对象放入 DynamoDB 表时，您可以选择指定一个条件表达式，以根据执行操作之前 DynamoDB 中的已有对象状态控制请求是否应成功。

- customPartitionKey?: string（可选）

  如果启用，该字符串值修改启用了版本控制时增量同步表使用的 ds_sk 和 ds_pk 记录的格式。如果启用，还会启用 populateIndexFields 条目处理。

- `item: Partial<T>`（必需）

  要放入 DynamoDB 的项目的其余属性。

- `key: DynamoDBKey<T>`（必需）

  一个必需的参数，用于指定 DynamoDB 中执行放置的项目的键。DynamoDB 项目可能具有单个哈希键或者哈希键和排序键。

- `populateIndexFields?: boolean`（可选）

  一个布尔值，在与 `customPartitionKey` 一起启用时，它为增量同步表中的每个记录创建新条目，具体来说是在 `gsi_ds_pk` 和 `gsi_ds_sk` 列中创建新条目。有关更多信息，请参阅《AWS AppSync 开发人员指南》中的冲突检测和同步。

`Type QueryInput<T>`

```
QueryInput<T>: ScanInput<T> & {
    query: DynamoDBKeyCondition<Required<T>>;
}
```

类型声明

- `query: DynamoDBKeyCondition<Required<T>>`（必需）

  指定描述要查询的项目的键条件。对于给定索引，分区键条件应该是相等，排序键的条件应该是比较或 `beginsWith`（在它是字符串时）。分区键和排序键仅支持数字和字符串类型。

  示例

  采用下面的 `User` 类型：

  ```
  type User = {
    id: string;
    name: string;
    age: number;
    isVerified: boolean;
    friendsIds: string[]
  }
  ```

  查询只能包含以下字段：`id`、`name` 和 `age`：

  ```
  const query: QueryInput<User> = {
  ```

```
    name: { eq: 'John' },
    age: { gt: 20 },
}
```

Type RemoveInput<T>

```
RemoveInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T>;
    customPartitionKey?: string;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
}
```

类型声明

- _version?: number（可选）
- condition?: DynamoDBFilterObject<T>（可选）

  在您删除 DynamoDB 中的对象时，您可以选择指定一个条件表达式，以根据执行操作之前 DynamoDB 中的已有对象状态控制请求是否应成功。

  示例

  以下示例是包含一个条件的 DeleteItem 表达式，只有在文档所有者与发出请求的用户匹配时，该条件才允许操作成功。

```
type Task = {
  id: string;
  title: string;
  description: string;
  owner: string;
  isComplete: boolean;
}
const condition: DynamoDBFilterObject<Task> = {
  owner: { eq: 'XXXXXXXXXXXXXXXX' },
}

remove<Task>({
   key: {
     id: 'XXXXXXXXXXXXXXXX',
  },
  condition,
```

```
});
```

- customPartitionKey?: string（可选）

  如果启用，customPartitionKey 值修改启用了版本控制时增量同步表使用的 ds_sk 和 ds_pk 记录的格式。如果启用，还会启用 populateIndexFields 条目处理。

- key: DynamoDBKey<T>（必需）

  一个必需的参数，用于指定在 DynamoDB 中删除的项目的键。DynamoDB 项目可能具有单个哈希键或者哈希键和排序键。

  示例

  如果 User 只有哈希键和用户 id，则该键如下所示：

```
type User = {
 id: number
 name: string
 age: number
 isVerified: boolean
}
const key: DynamoDBKey<User> = {
 id: 1,
}
```

  如果表用户具有哈希键 (id) 和排序键 (name)，则该键如下所示：

```
type User = {
 id: number
 name: string
 age: number
 isVerified: boolean
 friendsIds: string[]
}

const key: DynamoDBKey<User> = {
 id: 1,
 name: 'XXXXXXXXXX',
}
```

- populateIndexFields?: boolean（可选）

一个布尔值，在与 customPartitionKey 一起启用时，它为增量同步表中的每个记录创建新条目，具体来说是在 gsi_ds_pk 和 gsi_ds_sk 列中创建新条目。

Type ScanInput<T>

```
ScanInput<T>: {
    consistentRead?: boolean | null;
    filter?: DynamoDBFilterObject<T> | null;
    index?: string | null;
    limit?: number | null;
    nextToken?: string | null;
    scanIndexForward?: boolean | null;
    segment?: number;
    select?: DynamoDBSelectAttributes;
    totalSegments?: number;
}
```

类型声明

- consistentRead?: boolean | null（可选）

  一个可选的布尔值，用于指示查询 DynamoDB 时的一致性读取。默认值为 false。

- filter?: DynamoDBFilterObject<T> | null（可选）

  从表中检索结果后为结果应用的可选筛选条件。

- index?: string | null（可选）

  要扫描的索引名称（可选）。

- limit?: number | null（可选）

  要返回的最大结果数（可选）。

- nextToken?: string | null（可选）

  一个可选的分页标记，用于在以前查询之后继续执行。这应已从之前查询中获得。

- scanIndexForward?: boolean | null（可选）

  一个可选的布尔值，用于指示查询是按升序还是降序执行的。默认情况下，该值设置为 true。

- segment?: number（可选）
- select?: DynamoDBSelectAttributes（可选）

从 DynamoDB 中返回的属性。默认情况下，AWS AppSync DynamoDB 解析器仅返回投影到索引的属性。支持的值为：

- ALL_ATTRIBUTES

  返回指定的表或索引中的所有项目属性。如果您查询本地二级索引，则 DynamoDB 从父表中为索引中的每个匹配项目获取整个项目。如果索引配置为投影所有项目属性，则可以从本地二级索引中获得所有数据，而不要求提取。

- ALL_PROJECTED_ATTRIBUTES

  返回已投影到索引的所有属性。如果索引配置为投影所有属性，则此返回值等同于指定 ALL_ATTRIBUTES。

- SPECIFIC_ATTRIBUTES

  仅返回 ProjectionExpression 中列出的属性。该返回值相当于指定 ProjectionExpression 而不指定 AttributesToGet 的任何值。

- totalSegments?: number（可选）

Type DynamoDBSyncInput<T>

```
DynamoDBSyncInput<T>: {
    basePartitionKey?: string;
    deltaIndexName?: string;
    filter?: DynamoDBFilterObject<T> | null;
    lastSync?: number;
    limit?: number | null;
    nextToken?: string | null;
}
```

类型声明

- basePartitionKey?: string（可选）

  执行 Sync 操作时使用的基表的分区键。在表使用自定义分区键时，该字段允许执行 Sync 操作。

- deltaIndexName?: string（可选）

  用于 Sync 操作的索引。在表使用自定义分区键时，需要使用该索引才能对整个增量存储表启用 Sync 操作。Sync 操作是对 GSI（在 gsi_ds_pk 和 gsi_ds_sk 上创建）执行的。

- filter?: DynamoDBFilterObject<T> | null（可选）

从表中检索结果后为结果应用的可选筛选条件。

- `lastSync?: number`（可选）

  上次成功执行的 Sync 操作的启动时间（以纪元毫秒为单位）。如果指定，则仅返回 `lastSync` 之后更改的项目。只有在从初始 Sync 操作中检索所有页面后，才会填充该字段。如果省略，将返回基表的结果。否则，将返回增量表的结果。

- `limit?: number | null`（可选）

  一次评估的最大项目数（可选）。如果省略，则默认限制将设置为 100 个项目。该字段的最大值为 1000 个项目。

- `nextToken?: string | null`（可选）

`Type DynamoDBUpdateInput<T>`

```
DynamoDBUpdateInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T>;
    customPartitionKey?: string;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
    update: DynamoDBUpdateObject<T>;
}
```

类型声明

- `_version?: number`（可选）
- `condition?: DynamoDBFilterObject<T>`（可选）

  在您更新 DynamoDB 中的对象时，您可以选择指定一个条件表达式，以根据执行操作之前 DynamoDB 中的已有对象状态控制请求是否应成功。

- `customPartitionKey?: string`（可选）

  如果启用，`customPartitionKey` 值修改启用了版本控制时增量同步表使用的 `ds_sk` 和 `ds_pk` 记录的格式。如果启用，还会启用 `populateIndexFields` 条目处理。

- `key: DynamoDBKey<T>`（必需）

  一个必需的参数，用于指定在 DynamoDB 中更新的项目的键。DynamoDB 项目可能具有单个哈希键或者哈希键和排序键。

- `populateIndexFields?: boolean`（可选）

一个布尔值，在与 customPartitionKey 一起启用时，它为增量同步表中的每个记录创建新条目，具体来说是在 gsi_ds_pk 和 gsi_ds_sk 列中创建新条目。

- update: DynamoDBUpdateObject<T>

  一个对象，它指定要更新的属性及其新值。可以将更新对象与 add、remove、replace、increment、decrement、append、prepend 和 updateListItem 一起使用。

Amazon RDS 模块函数

在与使用 Amazon RDS 数据 API 配置的数据库进行交互时，Amazon RDS 模块函数可提供增强的体验。使用 @aws-appsync/utils/rds 导入模块：

```
import * as rds from '@aws-appsync/utils/rds';
```

也可以单独导入函数。例如，下面的导入使用 sql：

```
import { sql } from '@aws-appsync/utils/rds';
```

函数

您可以使用 AWS AppSync RDS 模块的实用程序帮助程序与您的数据库进行交互。

Select

select 实用程序会创建一条 SELECT 语句来查询您的关系数据库。

基本用法

在其基本形式中，您可以指定要查询的表：

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // "SELECT * FROM "persons"
    return createPgStatement(select({table: 'persons'}));
}
```

请注意，您也可以在表标识符中指定架构：

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // SELECT * FROM "private"."persons"
    return createPgStatement(select({table: 'private.persons'}));
}
```

指定列

您可以使用 columns 属性指定列。如果未将其设置为某个值，则它默认为 *：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name']
    }));
}
```

您也可以指定列的表：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "persons"."name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'persons.name']
    }));
}
```

限制和偏移

您可以将 limit 和 offset 应用于查询：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // LIMIT :limit
    // OFFSET :offset
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        limit: 10,
        offset: 40
    }));
}
```

排序依据

您可以使用 orderBy 属性对结果进行排序。提供指定列和可选 dir 属性的对象数组：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name" FROM "persons"
    // ORDER BY "name", "id" DESC
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        orderBy: [{column: 'name'}, {column: 'id', dir: 'DESC'}]
    }));
}
```

筛选器

您可以使用特殊条件对象来构建筛选条件：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME
    return createPgStatement(select({
        table: 'persons',
```

```
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}}
    }));
}
```

您也可以组合筛选条件：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME and "id" > :ID
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}, id: {gt: 10}}
    }));
}
```

您也可以创建 OR 语句：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME OR "id" > :ID
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: { or: [
            { name: { eq: 'Stephane'} },
            { id: { gt: 10 } }
        ]}
    }));
}
```

您也可以使用 not 来否定条件：

```
export function request(ctx) {
```

```
    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE NOT ("name" = :NAME AND "id" > :ID)
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: { not: [
            { name: { eq: 'Stephane'} },
            { id: { gt: 10 } }
        ]}
    }));
}
```

您也可以使用以下运算符来比较值：

运算符	描述	可能的值类型
eq	Equal	number, string, boolean
ne	Not equal	number, string, boolean
le	Less than or equal	number, string
lt	Less than	number, string
ge	Greater than or equal	number, string
gt	Greater than	number, string
contains	Like	string
notContains	Not like	string
beginsWith	Starts with prefix	string
between	Between two values	number, string
attributeExists	The attribute is not null	number, string, boolean
size	checks the length of the element	string

Insert

`insert` 实用程序提供了一种通过 INSERT 操作在数据库中插入单行项目的简单方法。

单个项目插入

要插入项目，请指定表，然后传入您的值对象。对象键映射到您的表列。列名称会自动转义，并使用变量映射将值发送到数据库：

```
import { insert, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({ table: 'persons', values });

    // Generates statement:
    // INSERT INTO `persons`(`name`)
    // VALUES(:NAME)
    return createMySQLStatement(insertStatement)
}
```

MySQL 用例

您可以组合 `insert` 后跟 `select` 来检索您插入的行：

```
import { insert, select, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({  table: 'persons', values });
    const selectStatement = select({
        table: 'persons',
        columns: '*',
        where: { id: { eq: values.id } },
        limit: 1,
    });

    // Generates statement:
    // INSERT INTO `persons`(`name`)
    // VALUES(:NAME)
    // and
    // SELECT *
    // FROM `persons`
```

```
    // WHERE `id` = :ID
    return createMySQLStatement(insertStatement, selectStatement)
}
```

Postgres 用例

借助 Postgres，您可以使用 returning 从插入的行中获取数据。它接受 * 或列名称数组：

```
import { insert, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({
        table: 'persons',
        values,
        returning: '*'
    });

    // Generates statement:
    // INSERT INTO "persons"("name")
    // VALUES(:NAME)
    // RETURNING *
    return createPgStatement(insertStatement)
}
```

更新

update 实用程序允许您更新现有行。您可以使用条件对象将更改应用于满足条件的所有行中的指定列。例如，假设我们有一个允许我们进行这种突变的架构。我们要将 Person 的 name 更新为 id 值 3，但仅限我们自 2000 年开始就已知道它们 (known_since)：

```
mutation Update {
    updatePerson(
        input: {id: 3, name: "Jon"},
        condition: {known_since: {ge: "2000"}}
    ) {
    id
    name
  }
}
```

更新解析器如下所示：

```
import { update, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: { id, ...values }, condition } = ctx.args;
    const where = {
        ...condition,
        id: { eq: id },
    };
    const updateStatement = update({
        table: 'persons',
        values,
        where,
        returning: ['id', 'name'],
    });

    // Generates statement:
    // UPDATE "persons"
    // SET "name" = :NAME, "birthday" = :BDAY, "country" = :COUNTRY
    // WHERE "id" = :ID
    // RETURNING "id", "name"
    return createPgStatement(updateStatement)
}
```

我们可以在条件中添加一项检查，以确保只更新主键 `id` 等于 3 的行。同样，对于 Postgres `inserts`，您可以使用 `returning` 返回修改后的数据。

删除

`remove` 实用程序允许您删除现有行。您可以在满足条件的所有行上使用条件对象。请注意，`delete` 是 JavaScript 中的保留关键字。应改用 `remove`：

```
import { remove, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: { id }, condition } = ctx.args;
    const where = { ...condition, id: { eq: id } };
    const deleteStatement = remove({
        table: 'persons',
        where,
        returning: ['id', 'name'],
    });

    // Generates statement:
```

```
    // DELETE "persons"
    // WHERE "id" = :ID
    // RETURNING "id", "name"
    return createPgStatement(updateStatement)
}
```

转换

在某些情况下，您可能希望在语句中使用更具体的正确对象类型。您可以使用提供的类型提示来指定参数的类型。AWS AppSync 支持与数据 API 相同的类型提示。您可以使用 AWS AppSync `rds` 模块中的 `typeHint` 函数来强制转换参数。

以下示例允许您将数组作为强制转换为 JSON 对象的值发送。我们使用 `->` 运算符来检索 JSON 数组中 `index` 为 2 的元素：

```
import { sql, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/
rds';

export function request(ctx) {
    const arr = ctx.args.list_of_ids
    const statement = sql`select ${typeHint.JSON(arr)}->2 as value`
    return createPgStatement(statement)
}

export function response(ctx) {
    return toJsonObject(ctx.result)[0][0].value
}
```

在处理和比较 `DATE`、`TIME` 和 `TIMESTAMP` 时，强制转换也很有用：

```
import { select, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const when = ctx.args.when
    const statement = select({
        table: 'persons',
        where: { createdAt : { gt: typeHint.DATETIME(when) } }
    })
    return createPgStatement(statement)
}
```

下面是另一个示例，显示如何发送当前日期和时间：

```
import { sql, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const now = util.time.nowFormatted('YYYY-MM-dd HH:mm:ss')
    return createPgStatement(sql`select ${typeHint.TIMESTAMP(now)}`)
}
```

可用的类型提示

- typeHint.DATE – 相应的参数作为 DATE 类型的对象发送到数据库。接受的格式为 YYYY-MM-DD。
- typeHint.DECIMAL – 相应的参数作为 DECIMAL 类型的对象发送到数据库。
- typeHint.JSON – 相应的参数作为 JSON 类型的对象发送到数据库。
- typeHint.TIME – 相应的字符串参数值作为 TIME 类型的对象发送到数据库。接受的格式为 HH:MM:SS[.FFF]。
- typeHint.TIMESTAMP – 相应的字符串参数值作为 TIMESTAMP 类型的对象发送到数据库。接受的格式为 YYYY-MM-DD HH:MM:SS[.FFF]。
- typeHint.UUID – 相应的字符串参数值作为 UUID 类型的对象发送到数据库。

运行时实用程序

runtime 库提供一些实用程序以控制或修改解析器和函数的运行时属性。

运行时实用程序列表

runtime.earlyReturn(obj?: unknown): never

调用该函数将停止执行当前的 AWS AppSync 函数或解析器（单位解析器或管道解析器），具体取决于当前上下文。将返回指定的对象以作为结果。

- 在 AWS AppSync 函数请求处理程序中调用时，将跳过数据源和响应处理程序，并调用下一个函数请求处理程序（如果这是最后一个 AWS AppSync 函数，则调用管道解析器响应处理程序）。
- 在 AWS AppSync 管道解析器请求处理程序中调用时，将跳过管道执行，并立即调用管道解析器响应处理程序。

示例

```
import { runtime } from '@aws-appsync/utils'
```

```
export function request(ctx) {
  runtime.earlyReturn({ hello: 'world' })
  // code below is not executed
  return ctx.args
}

// never called because request returned early
export function response(ctx) {
  return ctx.result
}
```

util.time 中的时间帮助程序

util.time 变量包含的日期时间方法有助于生成时间截，在不同的日期时间格式之间进行转换，并解析日期时间字符串。日期时间格式的语法基于 DateTimeFormatter，您可参考其他文档，进一步了解此内容。我们在下面提供了一些示例以及可用方法和描述列表。

时间实用程序

时间实用程序列表

`util.time.nowISO8601()`

返回 UTC 的 ISO8601 格式字符串表示形式。

`util.time.nowEpochSeconds()`

返回从 1970-01-01T00:00:00Z 纪元到现在的秒数。

`util.time.nowEpochMilliSeconds()`

返回从 1970-01-01T00:00:00Z 纪元到现在的毫秒数。

`util.time.nowFormatted(String)`

使用字符串输入类型指定的格式返回当前 UTC 时间戳的字符串。

`util.time.nowFormatted(String, String)`

使用字符串输入类型指定的格式和时区返回该时区当前时间戳的字符串。

`util.time.parseFormattedToEpochMilliSeconds(String, String)`

解析作为字符串传递的时间戳以及格式，然后将时间戳作为自纪元以来的毫秒数返回。

`util.time.parseFormattedToEpochMilliSeconds(String, String, String)`

解析作为字符串传递的时间戳以及格式和时区，然后将时间戳作为自纪元以来的毫秒数返回。

`util.time.parseISO8601ToEpochMilliSeconds(String)`

解析作为字符串传递的 ISO8601 时间戳，然后将时间戳作为自纪元以来的毫秒数返回。

`util.time.epochMilliSecondsToSeconds(long)`

将纪元毫秒数时间戳转换为纪元秒数时间戳。

`util.time.epochMilliSecondsToISO8601(long)`

将纪元毫秒数时间戳转换为 ISO8601 时间戳。

`util.time.epochMilliSecondsToFormatted(long, String)`

将以长型形式传递的纪元毫秒数时间戳转换为根据提供的 UTC 格式设置的时间戳。

`util.time.epochMilliSecondsToFormatted(long, String, String)`

将以长型形式传递的纪元毫秒数时间戳转换为根据提供的时区和格式设置的时间戳。

util.dynamodb 中的 DynamoDB 帮助程序

`util.dynamodb` 包含一些帮助程序方法，可以更轻松地在 Amazon DynamoDB 中写入和读取数据，例如自动类型映射和格式设置。

toDynamoDB

toDynamoDB 实用程序列表

`util.dynamodb.toDynamoDB(Object)`

DynamoDB 的常规对象转换工具，可以将输入对象转换为相应的 DynamoDB 表示形式。表示某些类型的方式是自主的：例如，使用列表 ("L") 而不使用集 ("SS", "NS", "BS")。这会返回一个描述 DynamoDB 属性值的对象。

字符串示例

```
Input:      util.dynamodb.toDynamoDB("foo")
```

```
Output:     { "S" : "foo" }
```

数字示例

```
Input:      util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

布尔值示例

```
Input:      util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

列表示例

```
Input:      util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

映射示例

```
Input:      util.dynamodb.toDynamoDB({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
                           { "S" : "boop" }
                       ]
                   }
               }
```

```
        }
```

toString 实用程序

toString 实用程序列表

`util.dynamodb.toString(String)`

将输入字符串转换为 DynamoDB 字符串格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

`util.dynamodb.toStringSet(List<String>)`

将包含字符串的列表转换为 DynamoDB 字符串集格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

toNumber 实用程序

toNumber 实用程序列表

`util.dynamodb.toNumber(Number)`

将数字转换为 DynamoDB 数字格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

`util.dynamodb.toNumberSet(List<Number>)`

将数字列表转换为 DynamoDB 数字集格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

toBinary 实用程序

toBinary 实用程序列表

`util.dynamodb.toBinary(String)`

将编码为 Base64 字符串的二进制数据转换为 DynamoDB 二进制格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

`util.dynamodb.toBinarySet(List<String>)`

将编码为 Base64 字符串的二进制数据列表转换为 DynamoDB 二进制集格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

toBoolean 实用程序

toBoolean 实用程序列表

`util.dynamodb.toBoolean(Boolean)`

将布尔值转换为相应的 DynamoDB 布尔值格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

toNull 实用程序

toNull 实用程序列表

`util.dynamodb.toNull()`

使用 DynamoDB Null 格式返回 Null。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      util.dynamodb.toNull()
```

```
Output:     { "NULL" : null }
```

toList 实用程序

toList 实用程序列表

`util.dynamodb.toList(List)`

将对象列表转换为 DynamoDB 列表格式。列表中的每个项目也会转换为相应的 DynamoDB 格式。表示某些嵌套对象的方式是自主的：例如，使用列表 ("L") 而不使用集 ("SS", "NS", "BS")。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

toMap 实用程序

toMap 实用程序列表

`util.dynamodb.toMap(Map)`

将映射转换为 DynamoDB 映射格式。映射中的每个值也会转换为相应的 DynamoDB 格式。表示某些嵌套对象的方式是自主的：例如，使用列表 ("L") 而不使用集 ("SS", "NS", "BS")。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      util.dynamodb.toMap({ "foo": "bar", "baz" : 1234, "beep": [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
```

```
                "beep" : {
                    "L" : [
                        { "S" : "boop" }
                    ]
                }
            }
        }
```

util.dynamodb.toMapValues(Map)

创建映射的副本，其中每个值都已转换为相应的 DynamoDB 格式。表示某些嵌套对象的方式是自主的：例如，使用列表 ("L") 而不使用集 ("SS", "NS", "BS")。

```
Input:      util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "foo"  : { "S" : "bar" },
               "baz"  : { "N" : 1234 },
               "beep" : {
                   "L" : [
                       { "S" : "boop" }
                   ]
               }
           }
```

> **Note**
>
> 这与 `util.dynamodb.toMap(Map)` 略有不同，因为它仅返回 DynamoDB 属性值内容，而不返回整个属性值本身。例如，以下语句是完全相同的：
>
> ```
> util.dynamodb.toMapValues(<map>)
> util.dynamodb.toMap(<map>)("M")
> ```

S3Object 实用程序

S3Object 实用程序列表

`util.dynamodb.toS3Object(String key, String bucket, String region)`

将键、存储桶和区域转换为 DynamoDB S3 对象表示形式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      util.dynamodb.toS3Object("foo", "bar", region = "baz")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo", \"bucket\" : \"bar", \"region
\" : \"baz" } }" }
```

`util.dynamodb.toS3Object(String key, String bucket, String region, String version)`

将键、存储桶、区域和可选版本转换为 DynamoDB S3 对象表示形式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      util.dynamodb.toS3Object("foo", "bar", "baz", "beep")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region
\" : \"baz\", \"version\" = \"beep\" } }" }
```

`util.dynamodb.fromS3ObjectJson(String)`

接受 DynamoDB S3 对象的字符串值，并返回包含键、存储桶、区域和可选版本的映射。

```
Input:      util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
 \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :
 "beep" }
```

util.http 中的 HTTP 帮助程序

`util.http` 实用程序提供一些帮助程序方法，可用于管理 HTTP 请求参数和添加响应标头。

util.http 实用程序列表

`util.http.copyHeaders(headers)`

从映射中复制标头，不包含受限制的 HTTP 标头集。您可以使用该方法将请求标头转发到下游 HTTP 终端节点。

`util.http.addResponseHeader(String, Object)`

添加单个自定义标头，其中包含响应的名称 (`String`) 和值 (`Object`)。适用以下限制：

- 标头名称不能与任何现有或受限制的 AWS 或 AWS AppSync 标头匹配。
- 标头名称不能以受限制的前缀开头，例如 `x-amzn-` 或 `x-amz-`。
- 自定义响应标头大小不能超过 4 KB。这包括标头名称和值。
- 对于每个 GraphQL 操作，您应该定义一次每个响应标头。不过，如果您多次定义具有相同名称的自定义标头，将在响应中显示最新的定义。无论命名如何，所有标头都会计入标头大小限制。

`util.http.addResponseHeaders(Map)`

将多个响应标头添加到来自指定的名称 (`String`) 和值 (`Object`) 映射的响应中。为 `addResponseHeader(String, Object)` 方法列出的相同限制也适用于该方法。

util.transform 中的转换帮助程序

`util.transform` 包含一些帮助程序方法，可以更轻松地对数据源执行复杂的操作。

转换帮助程序实用程序列表

`util.transform.toDynamoDBFilterExpression(filterObject: DynamoDBFilterObject) : string`

将输入字符串转换为筛选条件表达式以用于 DynamoDB。我们建议将 `toDynamoDBFilterExpression` 与内置模块函数一起使用。

`util.transform.toElasticsearchQueryDSL(object: OpenSearchQueryObject) : string`

将给定输入转换为等效的 OpenSearch 查询 DSL 表达式，以将其作为 JSON 字符串返回。

示例输入：

```
util.transform.toElasticsearchQueryDSL({
```

```
    "upvotes":{
        "ne":15,
        "range":[
            10,
            20
        ]
    },
    "title":{
        "eq":"hihihi",
        "wildcard":"h*i"
    }
  })
```

示例输出：

```
{
    "bool":{
      "must":[
          {
            "bool":{
              "must":[
                  {
                    "bool":{
                      "must_not":{
                        "term":{
                          "upvotes":15
                        }
                      }
                    }
                  },
                  {
                    "range":{
                      "upvotes":{
                        "gte":10,
                        "lte":20
                      }
                    }
                  }
              ]
            }
          },
          {
            "bool":{
```

```
            "must":[
                {
                  "term":{
                    "title":"hihihi"
                  }
                },
                {
                "wildcard":{
                    "title":"h*i"
                  }
                }
              ]
            }
          }
      ]
    }
}
```

> **Note**
>
> 默认运算符假定为 AND。

`util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter`

将 Map 输入对象转换为 SubscriptionFilter 表达式对象。util.transform.toSubscriptionFilter 方法用作 extensions.setSubscriptionFilter() 扩展的输入。有关更多信息，请参阅扩展。

> **Note**
>
> 下面列出了参数和返回语句：
> 参数
>
> - objFilter: SubscriptionFilterObject
>
>   转换为 SubscriptionFilter 表达式对象的 Map 输入对象。
>
> - ignoredFields：SubscriptionFilterExcludeKeysType（可选）
>
>   第一个对象中将忽略的字段名称 List。

- rules：SubscriptionFilterRuleObject（可选）

  在构建 SubscriptionFilter 表达式对象时包含的具有严格规则的 Map 输入对象。这些严格规则将包含在 SubscriptionFilter 表达式对象中，以便至少满足其中的一个规则才能通过订阅筛选条件。

响应

返回 SubscriptionFilter。

util.transform.toSubscriptionFilter(Map, List)

将 Map 输入对象转换为 SubscriptionFilter 表达式对象。util.transform.toSubscriptionFilter 方法用作 extensions.setSubscriptionFilter() 扩展的输入。有关更多信息，请参阅扩展。

第一个参数是转换为 SubscriptionFilter 表达式对象的 Map 输入对象。第二个参数是字段名称 List，在构建 SubscriptionFilter 表达式对象时，将在第一个 Map 输入对象中忽略这些字段名称。

util.transform.toSubscriptionFilter(Map, List, Map)

将 Map 输入对象转换为 SubscriptionFilter 表达式对象。util.transform.toSubscriptionFilter 方法用作 extensions.setSubscriptionFilter() 扩展的输入。有关更多信息，请参阅扩展。

util.transform.toDynamoDBConditionExpression(conditionObject)

创建 DynamoDB 条件表达式。

订阅筛选条件参数

下表介绍了如何定义以下实用程序的参数：

- Util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter

Argument 1: Map

参数 1 是一个 Map 对象，它具有以下键值：

- 字段名称

- "and"
- "or"

对于作为键的字段名称，这些字段的条目条件采用 "operator" : "value" 格式。

以下示例说明了如何将条目添加到 Map 中：

```
"field_name" : {
                    "operator1" : value
               }

## We can have multiple conditions for the same field_name:

"field_name" : {
                    "operator1" : value
                    "operator2" : value
                    .
                    .
                    .
               }
```

在一个字段具有两个或更多条件时，所有这些条件被视为使用 OR 运算。

输入 Map 也可以将 "and" 和 "or" 作为键，这意味着这些键中的所有条目应根据键使用 AND 或 OR 逻辑进行连接。键值 "and" 和 "or" 需要使用一个条件数组。

```
"and" : [

            {
                "field_name1" : {
                    "operator1" : value
                }
             },

             {
                "field_name2" : {
                    "operator1" : value
                }
             },
             .
             .
```

```
        ].
```

请注意，您可以嵌套 "and" 和 "or"。也就是说，您可以将 "and"/"or" 嵌套在另一个 "and"/"or" 块中。不过，这不适用于简单字段。

```
"and" : [

            {
                "field_name1" : {
                    "operator" : value
                }
             },

             {
                "or" : [
                            {
                                "field_name2" : {
                                    "operator" : value
                                }
                            },

                            {
                                "field_name3" : {
                                    "operator" : value
                                }
                            }

                        ].
```

以下示例显示使用 `util.transform.toSubscriptionFilter(Map) : Map` 的参数 1 的输入。

输入

参数 1：Map：

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
```

```
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

输出

结果是一个 Map 对象：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
```

```
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Admin"
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "gte",
          "value": 20
        },
        {
          "fieldName": "title",
          "operator": "ne",
```

```
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Admin"
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "gte",
          "value": 20
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 2: List

参数 2 包含字段名称 `List`，在构建 `SubscriptionFilter` 表达式对象时，不应在输入 `Map`（参数 1）中考虑使用这些字段名称。`List` 也可以是空的。

以下示例显示使用 `util.transform.toSubscriptionFilter(Map, List)` : `Map` 的参数 1 和参数 2 的输入。

输入

参数 1：Map：

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

参数 2：List：

```
["percentageUp", "author"]
```

输出

结果是一个 Map 对象：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 3: Map

参数 3 是一个将字段名称作为键值的 Map 对象（不能具有 "and" 或 "or"）。对于作为键的字段名称，这些字段的条件是采用 "operator" : "value" 格式的条目。与参数 1 不同，参数 3 不能在同一键中具有多个条件。此外，参数 3 没有 "and" 或 "or" 子句，因此，也不涉及嵌套。

参数 3 表示一组严格规则，这些规则将添加到 SubscriptionFilter 表达式对象中，以便至少满足其中的一个条件才能通过筛选条件。

```
{
```

```
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.
```

以下示例显示使用 util.transform.toSubscriptionFilter(Map, List, Map) : Map 的参数 1、参数 2 和参数 3 的输入。

输入

参数 1：Map：

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
```

```
        "eq": false
      }
    }
  ]
}
```

参数 2：List：

```
["percentageUp", "author"]
```

参数 3：Map：

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
}
```

输出

结果是一个 Map 对象：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
```

```
          "value": false
        },
        {
          "fieldName": "upvotes",
          "operator": "gte",
          "value": 250
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Person1"
        }
      ]
    }
  ]
}
```

util.str 中的字符串帮助程序

`util.str` 包含一些方法以帮助执行常见的字符串操作。

util.str 实用程序列表

`util.str.normalize(String, String)`

使用以下 4 种 unicode 规范化形式之一规范化字符串：NFC、NFD、NFKC 或 NFKD。第一个参数是要规范化的字符串。第二个参数是 "nfc"、"nfd"、"nfkc" 或 "nfkd"，它指定用于规范化过程的规范化类型。

扩展程序

`extensions` 包含一组在解析器中执行额外操作的方法。

缓存扩展

`extensions.evictFromApiCache(typeName: string, fieldName: string, keyValuePair: Record<string, string>) : Object`

从 AWS AppSync 服务器端缓存中逐出项目。第一个参数是类型名称。第二个参数是字段名称。第三个参数是一个对象，其中包含指定缓存键值的键值对项目。您必须按照与缓存解析器的 `cachingKey` 中的缓存键相同的顺序，将项目放入对象中。有关缓存的更多信息，请参阅缓存行为。

示例 1：

该示例逐出为名为 `Query.allClasses` 的解析器缓存的项目，在该解析器上使用了名为 `context.arguments.semester` 的缓存键。在调用变更并运行解析器时，如果成功清除条目，则响应在扩展对象中包含一个 `apiCacheEntriesDeleted` 值以显示删除了多少条目。

```
import { util, extensions } from '@aws-appsync/utils';

export const request = (ctx) => ({ payload: null });

export function response(ctx) {
 extensions.evictFromApiCache('Query', 'allClasses', {
  'context.arguments.semester': ctx.args.semester,
 });
 return null;
}
```

> **Note**
>
> 该函数仅适用于变更，而不适用于查询。

订阅扩展

`extensions.setSubscriptionFilter(filterJsonObject)`

定义增强的订阅筛选条件。每个订阅通知事件都会根据提供的订阅筛选条件进行评估，如果所有筛选条件的评估结果均为 `true`，则向客户端发送通知。参数是 `filterJsonObject`（可以在下面的参数：filterJsonObject 部分中找到有关该参数的更多信息）。请参阅增强订阅筛选。

> **Note**
>
> 您只能在订阅解析器的响应处理程序中使用该扩展函数。此外，我们建议使用 `util.transform.toSubscriptionFilter` 创建筛选条件。

`extensions.setSubscriptionInvalidationFilter(filterJsonObject)`

定义订阅失效筛选条件。根据失效负载评估订阅筛选条件，如果筛选条件的评估结果为 `true`，则使给定订阅失效。参数是 `filterJsonObject`（可以在下面的参数：filterJsonObject 部分中找到有关该参数的更多信息）。请参阅增强订阅筛选。

> **Note**
>
> 您只能在订阅解析器的响应处理程序中使用该扩展函数。此外，我们建议使用 `util.transform.toSubscriptionFilter` 创建筛选条件。

`extensions.invalidateSubscriptions(invalidationJsonObject)`

用于启动变更导致的订阅失效。参数是 `invalidationJsonObject`（可以在下面的参数：invalidationJsonObject 部分中找到有关该参数的更多信息）。

> **Note**
>
> 只能在变更解析器的响应映射模板中使用该扩展。

您最多只能在任何单个请求中使用 5 个唯一的 `extensions.invalidateSubscriptions()` 方法调用。如果超过该限制，您将收到 GraphQL 错误。

参数：filterJsonObject

JSON 对象定义订阅或失效筛选条件。它是 `filterGroup` 中的筛选条件数组。每个筛选条件是单独筛选条件的集合。

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

每个筛选条件具有三个属性：

- `fieldName` - GraphQL 架构字段。
- `operator` - 运算符类型。
- `value` - 与订阅通知 `fieldName` 值进行比较的值。

以下是这些属性的分配示例：

```
{
 "fieldName" : "severity",
 "operator" : "le",
 "value" : context.result.severity
}
```

参数：invalidationJsonObject

`invalidationJsonObject` 定义以下内容：

- `subscriptionField` - 要失效的 GraphQL 架构订阅。单个订阅（在 `subscriptionField` 中定义为字符串）被视为失效。
- `payload` - 一个键值对列表，如果失效筛选条件根据其值评估的结果为 `true`，则将该列表作为使订阅失效的输入。

  在订阅解析器中定义的失效筛选条件根据 `payload` 值评估的结果为 `true` 时，以下示例导致使用 `onUserDelete` 订阅的订阅和连接的客户端失效。

  ```
  export const request = (ctx) => ({ payload: null });

  export function response(ctx) {
   extensions.invalidateSubscriptions({
    subscriptionField: 'onUserDelete',
    payload: { group: 'Developer', type: 'Full-Time' },
   });
   return ctx.result;
  }
  ```

util.xml 中的 XML 帮助程序

`util.xml` 包含帮助进行 XML 字符串转换的方法。

util.xml 实用程序列表

`util.xml.toMap(String) : Object`

将 XML 字符串转换为字典。

示例 1：

```
Input:

<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
    <id>1</id>
    <title>Getting started with GraphQL</title>
</post>
</posts>

Output (object):

{
    "posts":{
      "post":{
        "id":1,
        "title":"Getting started with GraphQL"
      }
    }
}
```

示例 2：

```
Input:

<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
<post>
  <id>2</id>
  <title>Getting started with AppSync</title>
</post>
</posts>

Output (JavaScript object):

{
    "posts":{
    "post":[
        {
```

```
            "id":1,
            "title":"Getting started with GraphQL"
        },
        {
            "id":2,
            "title":"Getting started with AppSync"
        }
    ]
    }
}
```

`util.xml.toJsonString(String, Boolean?) : String`

将 XML 字符串转换为 JSON 字符串。这与 `toMap` 类似，只不过输出是字符串。如果您要直接转换 XML 响应并将其从 HTTP 对象返回到 JSON，这非常有用。您可以设置一个可选的布尔值参数，以确定是否要对 JSON 进行字符串编码。

DynamoDB 的 JavaScript 解析器函数参考

通过使用 AWS AppSync DynamoDB 函数，您可以使用 GraphQL 在您的账户的现有 Amazon DynamoDB 表中存储和检索数据。该解析器的工作方式是，允许您将传入的 GraphQL 请求映射到 DynamoDB 调用，然后将 DynamoDB 响应映射回 GraphQL。本节介绍了支持的 DynamoDB 操作的请求和响应处理程序。

GetItem

通过使用 `GetItem` 请求，您可以指示 AWS AppSync DynamoDB 函数向 DynamoDB 发出 `GetItem` 请求，并允许您指定：

- DynamoDB 中的项目的键
- 是否使用一致性读取

`GetItem` 请求具有以下结构：

```
type DynamoDBGetItem = {
  operation: 'GetItem';
  key: { [key: string]: any };
  consistentRead?: ConsistentRead;
  projection?: {
```

```
    expression: string;
    expressionNames?: { [key: string]: string };
  };
};
```

字段定义如下：

GetItem 字段

GetItem 字段列表

`operation`

要执行的 DynamoDB 操作。要执行 `GetItem` DynamoDB 操作，该字段必须设置为 `GetItem`。该值为必填项。

`key`

DynamoDB 中的项目的键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

`consistentRead`

是否对 DynamoDB 执行强一致性读取。这是可选的，默认值为 `false`。

`projection`

用于指定从 DynamoDB 操作返回的属性的投影。有关投影的更多信息，请参阅投影。该字段是可选的。

从 DynamoDB 返回的项目自动转换为 GraphQL 和 JSON 基元类型，并在上下文结果 (`context.result`) 中提供。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关 JavaScript 解析器的更多信息，请参阅 JavaScript 解析器概述。

示例

以下示例是 GraphQL 查询 `getThing(foo: String!, bar: String!)` 的函数请求处理程序：

```
export function request(ctx) {
  const {foo, bar} = ctx.args
```

```
  return {
    operation : "GetItem",
    key : util.dynamodb.toMapValues({foo, bar}),
    consistentRead : true
  }
}
```

有关 DynamoDB `GetItem` API 的更多信息，请参阅 DynamoDB API 文档。

PutItem

通过使用 `PutItem` 请求映射文档，您可以指示 AWS AppSync DynamoDB 函数向 DynamoDB 发出 `PutItem` 请求，并允许您指定以下内容：

- DynamoDB 中的项目的键
- 项目的完整内容（包括 `key` 和 `attributeValues`）
- 操作成功执行的条件

`PutItem` 请求具有以下结构：

```
type DynamoDBPutItemRequest = {
  operation: 'PutItem';
  key: { [key: string]: any };
  attributeValues: { [key: string]: any};
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

字段定义如下：

PutItem 字段

PutItem 字段列表

`operation`

要执行的 DynamoDB 操作。要执行 `PutItem` DynamoDB 操作，该字段必须设置为 `PutItem`。该值为必填项。

`key`

DynamoDB 中的项目的键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

`attributeValues`

要放入 DynamoDB 中的项目的其余属性。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该字段是可选的。

`condition`

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。如果未指定条件，则 `PutItem` 请求将覆盖该项目的任何现有条目。有关条件的更多信息，请参阅条件表达式。该值为可选项。

`_version`

表示项目的最新已知版本的数值。该值为可选项。该字段用于冲突检测 ，仅受版本化数据源支持。

`customPartitionKey`

如果启用，该字符串值修改启用了版本控制时增量同步表使用的 `ds_sk` 和 `ds_pk` 记录的格式（有关更多信息，请参阅《AWS AppSync 开发人员指南》中的冲突检测和同步）。如果启用，还会启用 `populateIndexFields` 条目处理。该字段是可选的。

`populateIndexFields`

一个布尔值，在与 **`customPartitionKey`** 一起启用时，它为增量同步表中的每个记录创建新条目，具体来说是在 `gsi_ds_pk` 和 `gsi_ds_sk` 列中创建新条目。有关更多信息，请参阅《AWS AppSync 开发人员指南》中的冲突检测和同步。该字段是可选的。

写入到 DynamoDB 的项目自动转换为 GraphQL 和 JSON 基元类型，并在上下文结果 (`context.result`) 中提供。

写入到 DynamoDB 的项目自动转换为 GraphQL 和 JSON 基元类型，并在上下文结果 (`context.result`) 中提供。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关 JavaScript 解析器的更多信息，请参阅 JavaScript 解析器概述。

示例 1

以下示例是 GraphQL 变更 `updateThing(foo: String!, bar: String!, name: String!, version: Int!)` 的函数请求处理程序。

如果带指定键的项目不存在，则会创建它。如果带指定键的项已存在，则会覆盖它。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, ...values} = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}
```

示例 2

以下示例是 GraphQL 变更 `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)` 的函数请求处理程序。

该示例验证当前位于 DynamoDB 中的项目的 `version` 字段是否设置为 `expectedVersion`。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, name, expectedVersion } = ctx.args;
  const values = { name, version: expectedVersion + 1 };
  let condition = util.transform.toDynamoDBConditionExpression({
    version: { eq: expectedVersion },
  });

  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({ foo, bar }),
    attributeValues: util.dynamodb.toMapValues(values),
    condition,
  };
}
```

有关 DynamoDB `PutItem` API 的更多信息，请参阅 DynamoDB API 文档。

UpdateItem

通过使用 `UpdateItem` 请求，您可以指示 AWS AppSync DynamoDB 函数向 DynamoDB 发出 `UpdateItem` 请求，并允许您指定以下内容：

- DynamoDB 中的项目的键
- 描述如何更新 DynamoDB 中的项目的更新表达式
- 操作成功执行的条件

`UpdateItem` 请求具有以下结构：

```
type DynamoDBUpdateItemRequest = {
  operation: 'UpdateItem';
  key: { [key: string]: any };
  update: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

字段定义如下：

UpdateItem 字段

UpdateItem 字段列表

`operation`

要执行的 DynamoDB 操作。要执行 `UpdateItem` DynamoDB 操作，该字段必须设置为 `UpdateItem`。该值为必填项。

`key`

DynamoDB 中的项目的键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

`update`

`update` 部分用于指定一个更新表达式，以描述如何更新 DynamoDB 中的项目。有关如何编写更新表达式的更多信息，请参阅 DynamoDB UpdateExpressions 文档。此部分是必需的。

`update` 部分有三个组成部分：

`expression`

更新表达式。该值为必填项。

`expressionNames`

以键值对形式替换表达式属性名称 占位符。键对应于 `expression` 中使用的名称占位符，值必须是与 DynamoDB 中的项目的属性名称对应的字符串。该字段是可选的，只应填充 `expression` 中使用的表达式属性名称占位符的替换内容。

`expressionValues`

以键值对形式替换表达式属性值 占位符。键对应于 `expression` 中使用的值占位符，而值必须为类型化值。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。必须指定此值。该字段是可选的，只应填充 `expression` 中使用的表达式属性值占位符的替换内容。

`condition`

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。如果未指定条件，则 `UpdateItem` 请求将更新现有条目，而不考虑其当前状态。有关条件的更多信息，请参阅条件表达式。该值为可选项。

`_version`

表示项目的最新已知版本的数值。该值为可选项。该字段用于冲突检测 ，仅受版本化数据源支持。

`customPartitionKey`

如果启用，该字符串值修改启用了版本控制时增量同步表使用的 `ds_sk` 和 `ds_pk` 记录的格式（有关更多信息，请参阅《AWS AppSync 开发人员指南》中的冲突检测和同步）。如果启用，还会启用 `populateIndexFields` 条目处理。该字段是可选的。

`populateIndexFields`

一个布尔值，在与 **`customPartitionKey`** 一起启用时，它为增量同步表中的每个记录创建新条目，具体来说是在 `gsi_ds_pk` 和 `gsi_ds_sk` 列中创建新条目。有关更多信息，请参阅《AWS AppSync 开发人员指南》中的冲突检测和同步。该字段是可选的。

在 DynamoDB 中更新的项目自动转换为 GraphQL 和 JSON 基元类型，并在上下文结果 (`context.result`) 中提供。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关 JavaScript 解析器的更多信息，请参阅 JavaScript 解析器概述。

示例 1

以下示例是 GraphQL 变更 `upvote(id: ID!)` 的函数请求处理程序。

在该示例中，DynamoDB 中的项目的 `upvotes` 和 `version` 字段递增 1。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { id } = ctx.args;
  return {
    operation: 'UpdateItem',
    key: util.dynamodb.toMapValues({ id }),
    update: {
      expression: 'ADD #votefield :plusOne, version :plusOne',
      expressionNames: { '#votefield': 'upvotes' },
      expressionValues: { ':plusOne': { N: 1 } },
    },
  };
}
```

示例 2

以下示例是 GraphQL 变更 `updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)` 的函数请求处理程序。

这是一个复杂的示例，用于检查参数并动态生成更新表达式，此表达式仅包含由客户端提供的参数。例如，如果省略了 `title` 和 `author`，则不会更新它们。如果指定了一个参数，但该参数的值为 `null`，则会从 DynamoDB 上的对象中删除该字段。最后，该操作具有一个条件，用于验证目前位于 DynamoDB 中的项目的 `version` 字段是否设置为 `expectedVersion`：

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { args: { input: { id, ...values } } } = ctx;

  const condition = {
    id: { attributeExists: true },
```

```
    version: { eq: values.expectedVersion },
  };
  values.expectedVersion += 1;
  return dynamodbUpdateRequest({ keys: { id }, values, condition });
}


/**
 * Helper function to update an item
 * @returns an UpdateItem request
 */
function dynamodbUpdateRequest(params) {
  const { keys, values, condition: inCondObj } = params;

  const sets = [];
  const removes = [];
  const expressionNames = {};
  const expValues = {};

  // Iterate through the keys of the values
  for (const [key, value] of Object.entries(values)) {
    expressionNames[`#${key}`] = key;
    if (value) {
      sets.push(`#${key} = :${key}`);
      expValues[`:${key}`] = value;
    } else {
      removes.push(`#${key}`);
    }
  }

  let expression = sets.length ? `SET ${sets.join(', ')}` : '';
  expression += removes.length ? ` REMOVE ${removes.join(', ')}` : '';

  const condition = JSON.parse(
    util.transform.toDynamoDBConditionExpression(inCondObj)
  );

  return {
    operation: 'UpdateItem',
    key: util.dynamodb.toMapValues(keys),
    condition,
    update: {
      expression,
      expressionNames,
```

```
      expressionValues: util.dynamodb.toMapValues(expValues),
    },
  };
}
```

有关 DynamoDB `UpdateItem` API 的更多信息，请参阅 DynamoDB API 文档。

DeleteItem

通过使用 `DeleteItem` 请求，您可以指示 AWS AppSync DynamoDB 函数向 DynamoDB 发出 `DeleteItem` 请求，并允许您指定以下内容：

- DynamoDB 中的项目的键
- 操作成功执行的条件

`DeleteItem` 请求具有以下结构：

```
type DynamoDBDeleteItemRequest = {
  operation: 'DeleteItem';
  key: { [key: string]: any };
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

字段定义如下：

DeleteItem 字段

DeleteItem 字段列表

`operation`

要执行的 DynamoDB 操作。要执行 `DeleteItem` DynamoDB 操作，该字段必须设置为 `DeleteItem`。该值为必填项。

`key`

DynamoDB 中的项目的键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

condition

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。如果未指定条件，则 `DeleteItem` 请求将删除项目，而不考虑其当前状态。有关条件的更多信息，请参阅条件表达式。该值为可选项。

_version

表示项目的最新已知版本的数值。该值为可选项。该字段用于冲突检测 ，仅受版本化数据源支持。

customPartitionKey

如果启用，该字符串值修改启用了版本控制时增量同步表使用的 `ds_sk` 和 `ds_pk` 记录的格式（有关更多信息，请参阅《AWS AppSync 开发人员指南》中的冲突检测和同步）。如果启用，还会启用 `populateIndexFields` 条目处理。该字段是可选的。

populateIndexFields

一个布尔值，在与 **customPartitionKey** 一起启用时，它为增量同步表中的每个记录创建新条目，具体来说是在 `gsi_ds_pk` 和 `gsi_ds_sk` 列中创建新条目。有关更多信息，请参阅《AWS AppSync 开发人员指南》中的冲突检测和同步。该字段是可选的。

从 DynamoDB 中删除的项目自动转换为 GraphQL 和 JSON 基元类型，并在上下文结果 (`context.result`) 中提供。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关 JavaScript 解析器的更多信息，请参阅 JavaScript 解析器概述。

示例 1

以下示例是 GraphQL 变更 `deleteItem(id: ID!)` 的函数请求处理程序。如果具有此 ID 的项目存在，它将被删除。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'DeleteItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}
```

示例 2

以下示例是 GraphQL 变更 `deleteItem(id: ID!, expectedVersion: Int!)` 的函数请求处理程序。如果具有此 ID 的项目存在，它将被删除，但仅当其 `version` 字段设置为 `expectedVersion` 时：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { id, expectedVersion } = ctx.args;
  const condition = {
    id: { attributeExists: true },
    version: { eq: expectedVersion },
  };
  return {
    operation: 'DeleteItem',
    key: util.dynamodb.toMapValues({ id }),
    condition: util.transform.toDynamoDBConditionExpression(condition),
  };
}
```

有关 DynamoDB `DeleteItem` API 的更多信息，请参阅 DynamoDB API 文档。

Query

通过使用 `Query` 请求对象，您可以指示 AWS AppSync DynamoDB 解析器向 DynamoDB 发出 `Query` 请求，并允许您指定以下内容：

- 键表达式
- 要使用的索引
- 任何额外的筛选条件
- 要返回多少个项目
- 是否使用一致性读取
- 查询方向（向前或向后）
- 分页标记

`Query` 请求对象具有以下结构：

```
type DynamoDBQueryRequest = {
  operation: 'Query';
  query: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
  index?: string;
  nextToken?: string;
  limit?: number;
  scanIndexForward?: boolean;
  consistentRead?: boolean;
  select?: 'ALL_ATTRIBUTES' | 'ALL_PROJECTED_ATTRIBUTES' | 'SPECIFIC_ATTRIBUTES';
  filter?: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
  projection?: {
    expression: string;
    expressionNames?: { [key: string]: string };
  };
};
```

字段定义如下：

Query 字段

Query 字段列表

`operation`

要执行的 DynamoDB 操作。要执行 `Query` DynamoDB 操作，该字段必须设置为 `Query`。该值为必填项。

`query`

`query` 部分用于指定一个键条件表达式，用于描述要从 DynamoDB 中检索哪些项目。有关如何编写键条件表达式的更多信息，请参阅 DynamoDB KeyConditions 文档。必须指定此部分。

`expression`

查询表达式。必须指定该字段。

expressionNames

以键值对形式替换表达式属性名称 占位符。键对应于 expression 中使用的名称占位符，值必须是与 DynamoDB 中的项目的属性名称对应的字符串。该字段是可选的，只应填充 expression 中使用的表达式属性名称占位符的替换内容。

expressionValues

以键值对形式替换表达式属性值 占位符。键对应于 expression 中使用的值占位符，而值必须为类型化值。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。该字段是可选的，只应填充 expression 中使用的表达式属性值占位符的替换内容。

filter

另一个筛选条件，该筛选条件可用于在从 DynamoDB 返回结果之前先筛选结果。有关筛选条件的更多信息，请参阅筛选条件。该字段是可选的。

index

要查询的索引的名称。除了哈希键的主键索引以外，您还可以通过 DynamoDB 查询操作扫描本地二级索引和全局二级索引。如果指定，这会指示 DynamoDB 查询指定的索引。如果省略，则将查询主键索引。

nextToken

继续之前查询的分页标记。这应已从之前查询中获得。该字段是可选的。

limit

要评估的最大项目数（不一定是匹配项目数）。该字段是可选的。

scanIndexForward

一个布尔值，指示是向前还是向后查询。该字段是可选的，默认值为 true。

consistentRead

一个布尔值，用于指示在查询 DynamoDB 时是否使用一致性读取。该字段是可选的，默认值为 false。

select

默认情况下，AWS AppSync DynamoDB 解析器仅返回投影到索引的属性。如果需要更多属性，则可以设置该字段。该字段是可选的。支持的值为：

ALL_ATTRIBUTES

返回指定的表或索引中的所有项目属性。如果您查询本地二级索引，则 DynamoDB 从父表中为索引中的每个匹配项目获取整个项目。如果索引配置为投影所有项目属性，则可以从本地二级索引中获得所有数据，而不要求提取。

ALL_PROJECTED_ATTRIBUTES

仅在查询索引时才允许。检索已投影到索引的所有属性。如果索引配置为投影所有属性，则此返回值等同于指定 ALL_ATTRIBUTES。

SPECIFIC_ATTRIBUTES

仅返回 projection 的 expression 中列出的属性。该返回值相当于指定 projection 的 Select，而不指定 expression 的任何值。

projection

用于指定从 DynamoDB 操作返回的属性的投影。有关投影的更多信息，请参阅投影。该字段是可选的。

来自 DynamoDB 的结果自动转换为 GraphQL 和 JSON 基元类型，并在上下文结果 (context.result) 中提供。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关 JavaScript 解析器的更多信息，请参阅 JavaScript 解析器概述。

结果的结构如下所示：

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

字段定义如下：

items

包含 DynamoDB 查询返回的项目的列表。

nextToken

如果可能有更多结果，则 nextToken 包含一个分页标记，您可以在另一个请求中使用该标记。请注意，AWS AppSync 对从 DynamoDB 返回的分页标记进行加密和模糊处理。这可防止您的表数据无意中泄露给调用方。另请注意，这些分页标记不能在不同的函数或解析器中使用。

scannedCount

在应用筛选表达式（如果有）之前与查询条件表达式匹配的项目的数量。

示例

以下示例是 GraphQL 查询 getPosts(owner: ID!) 的函数请求处理程序。

在本示例中，将对表的全局二级索引执行查询，以返回指定的 ID 拥有的所有文章。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { owner } = ctx.args;
  return {
    operation: 'Query',
    query: {
      expression: 'ownerId = :ownerId',
      expressionValues: util.dynamodb.toMapValues({ ':ownerId': owner }),
    },
    index: 'owner-index',
  };
}
```

有关 DynamoDB Query API 的更多信息，请参阅 DynamoDB API 文档。

Scan

通过使用 Scan 请求，您可以指示 AWS AppSync DynamoDB 函数向 DynamoDB 发出 Scan 请求，并允许您指定以下内容：

- 排除结果的筛选条件
- 要使用的索引
- 要返回多少个项目

- 是否使用一致性读取
- 分页标记
- 并行扫描

Scan 请求对象具有以下结构：

```
type DynamoDBScanRequest = {
  operation: 'Scan';
  index?: string;
  limit?: number;
  consistentRead?: boolean;
  nextToken?: string;
  totalSegments?: number;
  segment?: number;
  filter?: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
  projection?: {
    expression: string;
    expressionNames?: { [key: string]: string };
  };
};
```

字段定义如下：

Scan 字段

Scan 字段列表

`operation`

要执行的 DynamoDB 操作。要执行 Scan DynamoDB 操作，该字段必须设置为 Scan。该值为必填项。

`filter`

一个筛选条件，可用于在返回来自 DynamoDB 的结果之前对其进行筛选。有关筛选条件的更多信息，请参阅筛选条件。该字段是可选的。

index

要查询的索引的名称。除了哈希键的主键索引以外，您还可以通过 DynamoDB 查询操作扫描本地二级索引和全局二级索引。如果指定，这会指示 DynamoDB 查询指定的索引。如果省略，则将查询主键索引。

limit

单次评估的最大项目数。该字段是可选的。

consistentRead

一个布尔值，用于指示在查询 DynamoDB 时是否使用一致性读取。该字段是可选的，默认值为 `false`。

nextToken

继续之前查询的分页标记。这应已从之前查询中获得。该字段是可选的。

select

默认情况下，AWS AppSync DynamoDB 函数仅返回投影到索引中的任何属性。如果需要更多属性，则可以设置该字段。该字段是可选的。支持的值为：

ALL_ATTRIBUTES

返回指定的表或索引中的所有项目属性。如果您查询本地二级索引，则 DynamoDB 从父表中为索引中的每个匹配项目获取整个项目。如果索引配置为投影所有项目属性，则可以从本地二级索引中获得所有数据，而不要求提取。

ALL_PROJECTED_ATTRIBUTES

仅在查询索引时才允许。检索已投影到索引的所有属性。如果索引配置为投影所有属性，则此返回值等同于指定 `ALL_ATTRIBUTES`。

SPECIFIC_ATTRIBUTES

仅返回 `projection` 的 `expression` 中列出的属性。该返回值相当于指定 `projection` 的 `Select`，而不指定 `expression` 的任何值。

totalSegments

执行并行扫描时对表进行分区的分段数。该字段是可选的，但如果指定 `segment`，则必须指定该字段。

segment

执行并行扫描时，此操作中的表分段。该字段是可选的，但如果指定 `totalSegments`，则必须指定该字段。

projection

用于指定从 DynamoDB 操作返回的属性的投影。有关投影的更多信息，请参阅投影。该字段是可选的。

DynamoDB 扫描返回的结果自动转换为 GraphQL 和 JSON 基元类型，并在上下文结果 (`context.result`) 中提供。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关 JavaScript 解析器的更多信息，请参阅 JavaScript 解析器概述。

结果的结构如下所示：

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

字段定义如下：

items

包含 DynamoDB 扫描返回的项目的列表。

nextToken

如果可能有更多结果，则 `nextToken` 包含一个分页标记，您可以在另一个请求中使用该标记。AWSAppSync 对从 DynamoDB 返回的分页标记进行加密和模糊处理。这可防止您的表数据无意中泄露给调用方。此外，这些分页标记不能在不同的函数或解析器中使用。

scannedCount

在应用筛选条件表达式（如果有）之前 DynamoDB 检索的项目数。

示例 1

以下示例是 GraphQL 查询 `allPosts` 的函数请求处理程序。

在本示例中，将返回表中的所有条目。

```
export function request(ctx) {
  return { operation: 'Scan' };
}
```

示例 2

以下示例是 GraphQL 查询 `postsMatching(title: String!)` 的函数请求处理程序。

在本示例中，将返回表中标题以 `title` 参数开头的所有条目。

```
export function request(ctx) {
  const { title } = ctx.args;
  const filter = { filter: { beginsWith: title } };
  return {
    operation: 'Scan',
    filter: JSON.parse(util.transform.toDynamoDBFilterExpression(filter)),
  };
}
```

有关 DynamoDB Scan API 的更多信息，请参阅 DynamoDB API 文档。

Sync（同步）

通过使用 Sync 请求对象，您可以从 DynamoDB 表中检索所有结果，然后仅接收自上次查询以来更改的数据（增量更新）。只能对版本控制的 DynamoDB 数据源发出 Sync 请求。您可以指定：

- 排除结果的筛选条件
- 要返回多少个项目
- 分页标记
- 上次 Sync 操作开始时间

Sync 请求对象具有以下结构：

```
type DynamoDBSyncRequest = {
  operation: 'Sync';
  basePartitionKey?: string;
  deltaIndexName?: string;
  limit?: number;
  nextToken?: string;
  lastSync?: number;
  filter?: {
    expression: string;
    expressionNames?: { [key: string]: string };
    expressionValues?: { [key: string]: any };
  };
};
```

字段定义如下：

Sync 字段

Sync 字段列表

`operation`

要执行的 DynamoDB 操作。要执行 Sync 操作，该字段必须设置为 Sync。该值为必填项。

`filter`

一个筛选条件，可用于在返回来自 DynamoDB 的结果之前对其进行筛选。有关筛选条件的更多信息，请参阅筛选条件。该字段是可选的。

`limit`

单次评估的最大项目数。该字段是可选的。如果省略，则默认限制将设置为 `100` 个项目。该字段的最大值为 `1000` 个项目。

`nextToken`

继续之前查询的分页标记。这应已从之前查询中获得。该字段是可选的。

`lastSync`

最后一次成功 Sync 操作开始的时刻（以纪元毫秒为单位）。如果指定，则仅返回 `lastSync` 之后更改的项目。该字段是可选的，只有在从初始 Sync 操作中检索所有页面后才能填充。如果省略，将返回基本 表中的结果，否则将返回增量 表中的结果。

basePartitionKey

执行 Sync 操作时使用的基 表的分区键。在表使用自定义分区键时，该字段允许执行 Sync 操作。此为 可选字段。

deltaIndexName

用于 Sync 操作的索引。在表使用自定义分区键时，需要使用该索引才能对整个增量存储表启用 Sync 操作。Sync 操作是对 GSI（在 `gsi_ds_pk` 和 `gsi_ds_sk` 上创建）执行的。该字段是可选的。

DynamoDB 同步返回的结果自动转换为 GraphQL 和 JSON 基元类型，并在上下文结果 (`context.result`) 中提供。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关 JavaScript 解析器的更多信息，请参阅 JavaScript 解析器概述。

结果的结构如下所示：

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10,
    startedAt = 1550000000000
}
```

字段定义如下：

items

包含同步操作返回的项目的列表。

nextToken

如果可能有更多结果，则 `nextToken` 包含一个分页标记，您可以在另一个请求中使用该标记。AWSAppSync 对从 DynamoDB 返回的分页标记进行加密和模糊处理。这可防止您的表数据无意中泄露给调用方。此外，这些分页标记不能在不同的函数或解析器中使用。

scannedCount

在应用筛选条件表达式（如果有）之前 DynamoDB 检索的项目数。

startedAt

同步操作开始的时刻，以纪元毫秒为单位，您可以在本地存储该值并在其他请求中将其用作 lastSync 参数。如果请求中包含分页令牌，则该值将与请求针对第一页结果返回的值相同。

示例 1

以下示例是 GraphQL 查询 syncPosts(nextToken: String, lastSync: AWSTimestamp) 的函数请求处理程序。

在此示例中，如果省略 lastSync，则返回基表中的所有条目。如果提供了 lastSync，则只返回增量同步表中自 lastSync 以来发生更改的条目。

```
export function request(ctx) {
  const { nextToken, lastSync } = ctx.args;
  return { operation: 'Sync', limit: 100, nextToken, lastSync };
}
```

BatchGetItem

通过使用 BatchGetItem 请求对象，您可以指示 AWS AppSync DynamoDB 函数对 DynamoDB 发出 BatchGetItem 请求以检索多个项目（可能位于多个表中）。对于该请求对象，您必须指定以下内容：

- 要从中检索项目的表名称
- 要从每个表中检索的项目的键

DynamoDB BatchGetItem 限制适用，并且无法提供任何条件表达式。

BatchGetItem 请求对象具有以下结构：

```
type DynamoDBBatchGetItemRequest = {
  operation: 'BatchGetItem';
  tables: {
    [tableName: string]: {
      keys: { [key: string]: any }[];
      consistentRead?: boolean;
      projection?: {
        expression: string;
```

```
        expressionNames?: { [key: string]: string };
      };
    };
  };
};
```

字段定义如下：

BatchGetItem 字段

BatchGetItem 字段列表

`operation`

要执行的 DynamoDB 操作。要执行 `BatchGetItem` DynamoDB 操作，该字段必须设置为 `BatchGetItem`。该值为必填项。

`tables`

要从中检索项目的 DynamoDB 表。该值是一个映射，其中表名称被指定为映射的键。必须提供至少一个表。该 `tables` 值为必填项。

`keys`

DynamoDB 键列表，表示要检索的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。

`consistentRead`

是否在执行 GetItem 操作时使用一致性读取。此值是可选的，默认为 false。

`projection`

用于指定从 DynamoDB 操作返回的属性的投影。有关投影的更多信息，请参阅投影。该字段是可选的。

要记住的事项：

- 如果尚未从表中检索某个项目，则 null 元素将显示在该表的数据块中。
- 根据在请求对象中提供调用结果的顺序，将按表对这些结果进行排序。
- `BatchGetItem` 中的每个 `Get` 命令都是原子性的，但可以部分处理一个批次。如果由于错误而部分处理一个批处理，则未处理的键将作为 unprocessedKeys 块内的调用结果的一部分返回。

- BatchGetItem 限制为 100 个键。

对于以下示例函数请求处理程序：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'BatchGetItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId })],
      posts: [util.dynamodb.toMapValues({ authorId, postId })],
    },
  };
}
```

ctx.result 中可用的调用结果如下所示：

```
{
   "data": {
     "authors": [null],
     "posts": [
        // Was retrieved
        {
          "authorId": "a1",
          "postId": "p2",
          "postTitle": "title",
          "postDescription": "description",
        }
     ]
   },
   "unprocessedKeys": {
     "authors": [
        // This item was not processed due to an error
        {
          "authorId": "a1"
        }
      ],
     "posts": []
   }
}
```

ctx.error 包含有关该错误的详细信息。在调用结果中，确保会出现 data 和 unprocessedKeys 键以及在函数请求对象结果中提供的每个表键。已删除的项目显示在数据块中。尚未处理的项目将在数据块中标记为 null 并置于 unprocessedKeys 块中。

BatchDeleteItem

通过使用 BatchDeleteItem 请求对象，您可以指示 AWS AppSync DynamoDB 函数对 DynamoDB 发出 BatchWriteItem 请求以删除多个项目（可能位于多个表中）。对于该请求对象，您必须指定以下内容：

- 要从中删除项目的表名称
- 要从每个表中删除的项目的键

DynamoDB BatchWriteItem 限制适用，并且无法提供任何条件表达式。

BatchDeleteItem 请求对象具有以下结构：

```
type DynamoDBBatchDeleteItemRequest = {
  operation: 'BatchDeleteItem';
  tables: {
    [tableName: string]: { [key: string]: any }[];
  };
};
```

字段定义如下：

BatchDeleteItem 字段

BatchDeleteItem 字段列表

operation

要执行的 DynamoDB 操作。要执行 BatchDeleteItem DynamoDB 操作，该字段必须设置为 BatchDeleteItem。该值为必填项。

tables

要从中删除项目的 DynamoDB 表。每个表是 DynamoDB 键列表，表示要删除的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。必须提供至少一个表。tables 值是必需的。

要记住的事项：

- 与 DeleteItem 操作相反，响应中不会返回完全删除的项目。只返回传递的键。
- 如果尚未从表中删除某个项目，null 元素将显示在该表的数据块中。
- 根据在请求对象中提供调用结果的顺序，将按表对这些结果进行排序。
- BatchDeleteItem 中的每个 Delete 命令都是原子性的。但是，可以部分处理一个批处理。如果由于错误而部分处理一个批处理，则未处理的键将作为 unprocessedKeys 块内的调用结果的一部分返回。
- BatchDeleteItem 限制为 25 个键。

对于以下示例函数请求处理程序：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'BatchDeleteItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId })],
      posts: [util.dynamodb.toMapValues({ authorId, postId })],
    },
  };
}
```

ctx.result 中可用的调用结果如下所示：

```
{
   "data": {
     "authors": [null],
     "posts": [
        // Was deleted
        {
          "authorId": "a1",
          "postId": "p2"
        }
     ]
   },
   "unprocessedKeys": {
     "authors": [
```

```
        // This key was not processed due to an error
        {
          "authorId": "a1"
        }
      ],
     "posts": []
   }
}
```

ctx.error 包含有关该错误的详细信息。在调用结果中，确保会出现 data 和 unprocessedKeys 键以及在函数请求对象中提供的每个表键。已删除的项目存在于数据块中。尚未处理的项目将在数据块中标记为 null 并置于 unprocessedKeys 块中。

BatchPutItem

通过使用 BatchPutItem 请求对象，您可以指示 AWS AppSync DynamoDB 函数对 DynamoDB 发出 BatchWriteItem 请求以放置多个项目（可能位于多个表中）。对于该请求对象，您必须指定以下内容：

- 要将项目放置在其中的表名称
- 要放置在每个表中的完整项目

DynamoDB BatchWriteItem 限制适用，并且无法提供任何条件表达式。

BatchPutItem 请求对象具有以下结构：

```
type DynamoDBBatchPutItemRequest = {
  operation: 'BatchPutItem';
  tables: {
    [tableName: string]: { [key: string]: any}[];
  };
};
```

字段定义如下：

BatchPutItem 字段

BatchPutItem 字段列表

`operation`

要执行的 DynamoDB 操作。要执行 `BatchPutItem` DynamoDB 操作，该字段必须设置为 `BatchPutItem`。该值为必填项。

`tables`

要在其中放置项目的 DynamoDB 表。每个表条目表示要为该特定表插入的 DynamoDB 项目列表。必须提供至少一个表。该值为必填项。

要记住的事项：

- 如果成功，响应中将返回完全插入的项目。
- 如果尚未向表中插入项目，null 元素将显示在该表的数据块中。
- 根据在请求对象中提供插入的项目的顺序，将按表对这些结果进行排序。
- `BatchPutItem` 中的每个 `Put` 命令都是原子性的，但可以部分处理一个批次。如果由于错误而部分处理一个批处理，则未处理的键将作为 unprocessedKeys 块内的调用结果的一部分返回。
- `BatchPutItem` 限制为 25 个项目。

对于以下示例函数请求处理程序：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId, name, title } = ctx.args;
  return {
    operation: 'BatchPutItem',
    tables: {
      authors: [util.dynamodb.toMapValues({ authorId, name })],
      posts: [util.dynamodb.toMapValues({ authorId, postId, title })],
    },
  };
}
```

`ctx.result` 中可用的调用结果如下所示：

```
{
   "data": {
     "authors": [
         null
     ],
     "posts": [
        // Was inserted
        {
          "authorId": "a1",
          "postId": "p2",
          "title": "title"
        }
     ]
   },
   "unprocessedItems": {
     "authors": [
        // This item was not processed due to an error
        {
          "authorId": "a1",
          "name": "a1_name"
        }
      ],
     "posts": []
   }
}
```

ctx.error 包含有关该错误的详细信息。在调用结果中，确保会出现 data 和 unprocessedItems 键以及在请求对象中提供的每个表键。已插入的项目存在于数据块中。尚未处理的项目将在数据块中标记为 null 并置于 unprocessedItems 块中。

TransactGetItems

通过使用 TransactGetItems 请求对象，您可以指示 AWS AppSync DynamoDB 函数对 DynamoDB 发出 TransactGetItems 请求以检索多个项目（可能位于多个表中）。对于该请求对象，您必须指定以下内容：

- 从中检索项目的每个请求项目的表名称
- 要从每个表中检索的每个请求项的键

DynamoDB TransactGetItems 限制适用，并且无法提供任何条件表达式。

TransactGetItems 请求对象具有以下结构：

```
type DynamoDBTransactGetItemsRequest = {
  operation: 'TransactGetItems';
  transactItems: { table: string; key: { [key: string]: any }; projection?:
 { expression: string; expressionNames?: { [key: string]: string }; }[];
  };
};
```

字段定义如下：

TransactGetItems 字段

TransactGetItems 字段列表

operation

要执行的 DynamoDB 操作。要执行 TransactGetItems DynamoDB 操作，该字段必须设置为 TransactGetItems。该值为必填项。

transactItems

要包含的请求项目。该值是请求项目的数组。必须提供至少一个请求项目。该 transactItems 值为必填项。

table

要从中检索项目的 DynamoDB 表。该值是表名的字符串。该 table 值为必填项。

key

DynamoDB 键，表示要检索的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。

projection

用于指定从 DynamoDB 操作返回的属性的投影。有关投影的更多信息，请参阅投影。该字段是可选的。

要记住的事项：

- 如果事务成功，items 块中检索项目的顺序将与请求项目的顺序相同。

- 事务以全部或全不的方式执行。如果任何请求项目导致错误，则整个交易都不会执行，并返回错误详细信息。
- 无法检索的请求项目不是错误。相反，null 元素会出现在相应位置的 items 块中。
- 如果一个事务的错误是 TransactionCanceledException，则 `cancellationReasons` 块将被填充。`cancellationReasons` 块中取消原因的顺序将与请求项目的顺序相同。
- `TransactGetItems` 仅限于 25 个请求项目。

对于以下示例函数请求处理程序：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId } = ctx.args;
  return {
    operation: 'TransactGetItems',
    transactItems: [
      {
        table: 'posts',
        key: util.dynamodb.toMapValues({ postId }),
      },
      {
        table: 'authors',
        key: util.dynamodb.toMapValues({ authorId }),
      },
    ],
  };
}
```

如果事务成功并且只检索第一个请求的项目，则 `ctx.result` 中可用的调用结果如下所示：

```
{
    "items": [
       {
           // Attributes of the first requested item
           "post_id": "p1",
           "post_title": "title",
           "post_description": "description"
       },
       // Could not retrieve the second requested item
       null,
    ],
```

```
    "cancellationReasons": null
}
```

如果事务由于第一个请求项目引起的 TransactionCanceledException 而失败，则 `ctx.result` 中可用的调用结果如下所示：

```
{
    "items": null,
    "cancellationReasons": [
       {
           "type":"Sample error type",
           "message":"Sample error message"
       },
       {
           "type":"None",
           "message":"None"
       }
    ]
}
```

`ctx.error` 包含有关该错误的详细信息。键 items 和 cancellationReasons 保证出现在 `ctx.result` 中。

TransactWriteItems

通过使用 `TransactWriteItems` 请求对象，您可以指示 AWS AppSync DynamoDB 函数对 DynamoDB 发出 `TransactWriteItems` 请求以写入多个项目（可能写入到多个表中）。对于该请求对象，您必须指定以下内容：

- 每个请求项目的目标表名称
- 要执行的每个请求项目的操作。支持四种类型的操作：PutItem、UpdateItem、DeleteItem 和 ConditionCheck
- 要写入的每个请求项目的键

DynamoDB `TransactWriteItems` 限制适用。

`TransactWriteItems` 请求对象具有以下结构：

```
type DynamoDBTransactWriteItemsRequest = {
  operation: 'TransactWriteItems';
```

```
  transactItems: TransactItem[];
};
type TransactItem =
  | TransactWritePutItem
  | TransactWriteUpdateItem
  | TransactWriteDeleteItem
  | TransactWriteConditionCheckItem;
type TransactWritePutItem = {
  table: string;
  operation: 'PutItem';
  key: { [key: string]: any };
  attributeValues: { [key: string]: string};
  condition?: TransactConditionCheckExpression;
};
type TransactWriteUpdateItem = {
  table: string;
  operation: 'UpdateItem';
  key: { [key: string]: any };
  update: DynamoDBExpression;
  condition?: TransactConditionCheckExpression;
};
type TransactWriteDeleteItem = {
  table: string;
  operation: 'DeleteItem';
  key: { [key: string]: any };
  condition?: TransactConditionCheckExpression;
};
type TransactWriteConditionCheckItem = {
  table: string;
  operation: 'ConditionCheck';
  key: { [key: string]: any };
  condition?: TransactConditionCheckExpression;
};
type TransactConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
  returnValuesOnConditionCheckFailure: boolean;
};
```

TransactWriteItems 字段

TransactWriteItems 字段列表

字段定义如下：

`operation`

要执行的 DynamoDB 操作。要执行 `TransactWriteItems` DynamoDB 操作，该字段必须设置为 `TransactWriteItems`。该值为必填项。

`transactItems`

要包含的请求项目。该值是请求项目的数组。必须提供至少一个请求项目。该 `transactItems` 值为必填项。

对于 `PutItem`，字段定义如下：

`table`

目标 DynamoDB 表。该值是表名的字符串。该 `table` 值为必填项。

`operation`

要执行的 DynamoDB 操作。要执行 `PutItem` DynamoDB 操作，该字段必须设置为 `PutItem`。该值为必填项。

`key`

DynamoDB 键，表示要放置的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

`attributeValues`

要放入 DynamoDB 中的项目的其余属性。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该字段是可选的。

`condition`

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。如果未指定条件，则 `PutItem` 请求将覆盖该项目的任何现有条目。您可以指定在状况检查失败时是否重新检索现有项目。有关事务条件的更多信息，请参阅事务条件表达式。该值为可选项。

对于 `UpdateItem`，字段定义如下：

table

要更新的 DynamoDB 表。该值是表名的字符串。该 `table` 值为必填项。

operation

要执行的 DynamoDB 操作。要执行 `UpdateItem` DynamoDB 操作，该字段必须设置为 `UpdateItem`。该值为必填项。

key

DynamoDB 键，表示要更新的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

update

`update` 部分用于指定一个更新表达式，以描述如何更新 DynamoDB 中的项目。有关如何编写更新表达式的更多信息，请参阅 DynamoDB UpdateExpressions 文档。此部分是必需的。

condition

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。如果未指定条件，则 `UpdateItem` 请求将更新现有条目，而不考虑其当前状态。您可以指定在状况检查失败时是否重新检索现有项目。有关事务条件的更多信息，请参阅事务条件表达式。该值为可选项。

对于 `DeleteItem`，字段定义如下：

table

要在其中删除项目的 DynamoDB 表。该值是表名的字符串。该 `table` 值为必填项。

operation

要执行的 DynamoDB 操作。要执行 `DeleteItem` DynamoDB 操作，该字段必须设置为 `DeleteItem`。该值为必填项。

key

DynamoDB 键，表示要删除的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

condition

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。如果未指定条件，则 DeleteItem 请求将删除项目，而不考虑其当前状态。您可以指定在状况检查失败时是否重新检索现有项目。有关事务条件的更多信息，请参阅事务条件表达式。该值为可选项。

对于 ConditionCheck，字段定义如下：

table

要在其中检查条件的 DynamoDB 表。该值是表名的字符串。该 table 值为必填项。

operation

要执行的 DynamoDB 操作。要执行 ConditionCheck DynamoDB 操作，该字段必须设置为 ConditionCheck。该值为必填项。

key

DynamoDB 键，表示要检查条件的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

condition

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。您可以指定在状况检查失败时是否重新检索现有项目。有关事务条件的更多信息，请参阅事务条件表达式。该值为必填项。

要记住的事项：

- 如果成功，响应中只返回请求项目的键。键的顺序将与请求项目的顺序相同。
- 事务以全部或全不的方式执行。如果任何请求项目导致错误，则整个交易都不会执行，并返回错误详细信息。
- 不能有两个请求项目针对同一个项目。否则，它们将导致 TransactionCanceledException 错误。
- 如果一个事务的错误是 TransactionCanceledException，则 cancellationReasons 块将被填充。如果请求项目的条件检查失败且 您没有将 returnValuesOnConditionCheckFailure 指定为 false，则表中存在的项目将被检索并存储在 cancellationReasons 块的相应位置的 item 中。
- TransactWriteItems 仅限于 25 个请求项目。

对于以下示例函数请求处理程序：

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { authorId, postId, title, description, oldTitle, authorName } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'PutItem',
        key: util.dynamodb.toMapValues({ postId }),
        attributeValues: util.dynamodb.toMapValues({ title, description }),
        condition: util.transform.toDynamoDBConditionExpression({
          title: { eq: oldTitle },
        }),
      },
      {
        table: 'authors',
        operation: 'UpdateItem',
        key: util.dynamodb.toMapValues({ authorId }),
        update: {
          expression: 'SET authorName = :name',
          expressionValues: util.dynamodb.toMapValues({ ':name': authorName }),
        },
      },
    ],
  };
}
```

如果事务成功，ctx.result 中可用的调用结果如下所示：

```
{
    "keys": [
       // Key of the PutItem request
       {
           "post_id": "p1",
       },
       // Key of the UpdateItem request
       {
           "author_id": "a1"
       }
```

```
    ],
    "cancellationReasons": null
}
```

如果由于 PutItem 请求的条件检查失败而导致事务失败，则 ctx.result 中提供的调用结果如下所示：

```
{
    "keys": null,
    "cancellationReasons": [
       {
           "item": {
               "post_id": "p1",
               "post_title": "Actual old title",
               "post_description": "Old description"
           },
           "type": "ConditionCheckFailed",
           "message": "The condition check failed."
       },
       {
           "type": "None",
           "message": "None"
       }
    ]
}
```

ctx.error 包含有关该错误的详细信息。键 keys 和 cancellationReasons 保证出现在 ctx.result 中。

类型系统（请求映射）

在使用 AWS AppSync DynamoDB 函数调用 DynamoDB 表时，AWS AppSync 需要知道该调用中使用的每个值的类型。这是因为 DynamoDB 支持的基元类型比 GraphQL 或 JSON 多（例如集和二进制数据）。AWS在 GraphQL 和 DynamoDB 之间转换时，AppSync 需要一些提示，否则，它必须对如何在表中设置数据结构做出一些假设。

有关 DynamoDB 数据类型的更多信息，请参阅 DynamoDB 数据类型描述符和数据类型文档。

DynamoDB 值由包含单个键值对的 JSON 对象表示。键指定 DynamoDB 类型，值指定值本身。在下面的示例中，键 S 表示值是一个字符串，值 identifier 是字符串值本身。

```
{ "S" : "identifier" }
```

请注意，JSON 对象不能具有多于一个键值对。如果指定了多个键值对，则不会解析请求对象。

DynamoDB 值用于请求对象中您需要指定值的任何位置。您需要这样做的一些地方包括：key 和 attributeValue 部分以及表达式部分的 expressionValues 部分。在以下示例中，DynamoDB 字符串值 identifier 分配给 key 部分中的 id 字段（可能位于 GetItem 请求对象中）。

```
"key" : {
   "id" : { "S" : "identifier" }
}
```

支持的类型

AWS AppSync 支持以下 DynamoDB 标量、文档和集类型：

字符串类型 **S**

单个字符串值。DynamoDB 字符串值表示为：

```
{ "S" : "some string" }
```

示例用法如下：

```
"key" : {
   "id" : { "S" : "some string" }
}
```

字符串集类型 **SS**

一组字符串值。DynamoDB 字符串集值表示为：

```
{ "SS" : [ "first value", "second value", ... ] }
```

示例用法如下：

```
"attributeValues" : {
   "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
}
```

数字类型 **N**

单个数字值。DynamoDB 数字值表示为：

```
{ "N" : 1234 }
```

示例用法如下：

```
"expressionValues" : {
   ":expectedVersion" : { "N" : 1 }
}
```

数字集类型 **NS**

一组数字值。DynamoDB 数字集值表示为：

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

示例用法如下：

```
"attributeValues" : {
   "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }
}
```

二进制类型 **B**

二进制值。DynamoDB 二进制值表示为：

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

请注意，该值实际上是一个字符串，其中，字符串是二进制数据的 Base64 编码表示形式。AWSAppSync 将该字符串解码回二进制值，然后再将其发送到 DynamoDB。AWSAppSync 使用 RFC 2045 定义的 Base64 解码方案：忽略没有位于 Base64 字母表中的任何字符。

示例用法如下：

```
"attributeValues" : {
   "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

二进制集类型 **BS**

一组二进制值。DynamoDB 二进制集值表示为：

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

请注意，该值实际上是一个字符串，其中，字符串是二进制数据的 Base64 编码表示形式。AWSAppSync 将该字符串解码回二进制值，然后再将其发送到 DynamoDB。AWSAppSync 使用 RFC 2045 定义的 Base64 解码方案：忽略没有位于 Base64 字母表中的任何字符。

示例用法如下：

```
"attributeValues" : {
   "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }
}
```

布尔值类型 **BOOL**

布尔值。DynamoDB 布尔值表示为：

```
{ "BOOL" : true }
```

请注意，只有 `true` 和 `false` 是有效值。

示例用法如下：

```
"attributeValues" : {
   "orderComplete" : { "BOOL" : false }
}
```

列表类型 **L**

任何其他支持的 DynamoDB 值列表。DynamoDB 列表值表示为：

```
{ "L" : [ ... ] }
```

请注意，该值是一个复合值，其中，列表可以包含零个或更多任何支持的 DynamoDB 值（包括其他列表）。此列表还可以包含不同类型的混合。

示例用法如下：

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

映射类型 **M**

表示其他支持的 DynamoDB 值的键值对的无序集合。DynamoDB 映射值表示为：

```
{ "M" : { ... } }
```

请注意，一个映射可以包含零个或零个以上的键值对。键必须是一个字符串，值可以是任何支持的 DynamoDB 值（包括其他映射）。此映射还可以包含不同类型的混合。

示例用法如下：

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

Null 类型 **NULL**

Null 值。DynamoDB Null 值表示为：

```
{ "NULL" : null }
```

示例用法如下：

```
"attributeValues" : {
   "phoneNumbers" : { "NULL" : null }
}
```

有关每个类型的更多信息，请参阅 DynamoDB 文档。

类型系统（响应映射）

在从 DynamoDB 收到响应时，AWS AppSync 自动将其转换为 GraphQL 和 JSON 基元类型。将解码 DynamoDB 中的每个属性，并在响应处理程序的上下文中返回。

例如，如果 DynamoDB 返回以下内容：

```
{
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

从管道解析器中返回结果时，AWS AppSync 将其转换为 GraphQL 和 JSON 类型，如下所示：

```
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

本节介绍了 AWS AppSync 如何转换以下 DynamoDB 标量、文档和集类型：

字符串类型 S

单个字符串值。DynamoDB 字符串值以字符串形式返回。

例如，如果 DynamoDB 返回以下 DynamoDB 字符串值：

```
{ "S" : "some string" }
```

AWS AppSync 将其转换为字符串：

```
"some string"
```

字符串集类型 SS

一组字符串值。DynamoDB 字符串集值以字符串列表形式返回。

例如，如果 DynamoDB 返回以下 DynamoDB 字符串集值：

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync 将其转换为字符串列表：

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

数字类型 **N**

单个数字值。DynamoDB 数字值以数字形式返回。

例如，如果 DynamoDB 返回以下 DynamoDB 数字值：

```
{ "N" : 1234 }
```

AWS AppSync 将其转换为数字：

```
1234
```

数字集类型 **NS**

一组数字值。DynamoDB 数字集值以数字列表形式返回。

例如，如果 DynamoDB 返回以下 DynamoDB 数字集值：

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync 将其转换为数字列表：

```
[ 67.8, 12.2, 70 ]
```

二进制类型 **B**

二进制值。DynamoDB 二进制值以字符串形式返回，其中包含该值的 Base64 表示形式。

例如，如果 DynamoDB 返回以下 DynamoDB 二进制值：

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync 将其转换为包含该值的 Base64 表示形式的字符串：

```
"SGVsbG8sIFdvcmxkIQo="
```

请注意，二进制数据按照 RFC 4648 和 RFC 2045 中指定的方式根据 base64 编码方案进行编码。

二进制集类型 **BS**

一组二进制值。DynamoDB 二进制集值以字符串列表形式返回，其中包含这些值的 Base64 表示形式。

例如，如果 DynamoDB 返回以下 DynamoDB 二进制集值：

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync 将其转换为包含这些值的 Base64 表示形式的字符串列表：

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

请注意，二进制数据按照 RFC 4648 和 RFC 2045 中指定的方式根据 base64 编码方案进行编码。

布尔值类型 **BOOL**

布尔值。DynamoDB 布尔值以布尔值形式返回。

例如，如果 DynamoDB 返回以下 DynamoDB 布尔值：

```
{ "BOOL" : true }
```

AWS AppSync 将其转换为布尔值：

```
true
```

列表类型 **L**

任何其他支持的 DynamoDB 值列表。DynamoDB 列表值以值列表形式返回，其中还会转换每个内部值。

例如，如果 DynamoDB 返回以下 DynamoDB 列表值：

```
{ "L" : [
```

```
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

AWS AppSync 将其转换为转换的值列表：

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

映射类型 **M**

任何其他支持的 DynamoDB 值的键/值集合。DynamoDB 映射值以 JSON 对象形式返回，其中还会转换每个键/值。

例如，如果 DynamoDB 返回以下 DynamoDB 映射值：

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

AWS AppSync 将其转换为 JSON 对象：

```
{
   "someString" : "A string value",
   "someNumber" : 1,
   "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

Null 类型 **NULL**

Null 值。

例如，如果 DynamoDB 返回以下 DynamoDB Null 值：

```
{ "NULL" : null }
```

AWS AppSync 将其转换为 Null：

```
null
```

Filters

在使用 Query 和 Scan 操作查询 DynamoDB 中的对象时，您可以选择指定 filter 以评估结果并仅返回所需的值。

Query 或 Scan 请求的 filter 属性具有以下结构：

```
type DynamoDBExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
};
```

字段定义如下：

expression

查询表达式。有关如何编写筛选条件表达式的更多信息，请参阅 DynamoDB QueryFilter 和 DynamoDB ScanFilter 文档。必须指定该字段。

expressionNames

以键值对形式替换表达式属性名称 占位符。键对应于 expression 中使用的名称占位符。该值必须是与 DynamoDB 中的项目的属性名称对应的字符串。该字段是可选的，只应填充 expression 中使用的表达式属性名称占位符的替换内容。

expressionValues

以键值对形式替换表达式属性值 占位符。键对应于 expression 中使用的值占位符，而值必须为类型化值。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。必须指定此值。该字段是可选的，只应填充 expression 中使用的表达式属性值占位符的替换内容。

示例

以下示例是请求的筛选条件部分，其中，只有在标题以 title 参数开头时，才会返回从 DynamoDB 中检索的条目。

此处，我们使用 util.transform.toDynamoDBFilterExpression 从对象中自动创建筛选条件：

```
const filter = util.transform.toDynamoDBFilterExpression({
  title: { beginsWith: 'far away' },
});

const request = {};
request.filter = JSON.parse(filter);
```

这会生成以下筛选条件：

```
{
  "filter": {
    "expression": "(begins_with(#title,:title_beginsWith))",
    "expressionNames": { "#title": "title" },
    "expressionValues": {
      ":title_beginsWith": { "S": "far away" }
    }
  }
}
```

条件表达式

在您使用 PutItem、UpdateItem 和 DeleteItem DynamoDB 操作变更 DynamoDB 中的对象时，您可以选择指定一个条件表达式，以根据执行操作之前 DynamoDB 中的已有对象状态控制请求是否应成功。

AWS AppSync DynamoDB 函数允许在 PutItem、UpdateItem 和 DeleteItem 请求对象中指定条件表达式，并提供在条件失败并且未更新对象时遵循的策略。

示例 1

以下 PutItem 请求对象没有条件表达式。因此，即使已存在具有相同键的项目，它也会将项目放置在 DynamoDB 中，从而覆盖现有的项目。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, ...values} = ctx.args
  return {
    operation: 'PutItem',
```

```
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues(values),
  };
}
```

示例 2

以下 PutItem 对象确实具有一个条件表达式，只有在 DynamoDB 中不 存在具有相同键的项目时，该操作才会成功。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { foo, bar, ...values} = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues(values),
    condition: { expression: "attribute_not_exists(id)" }
  };
}
```

默认情况下，如果条件检查失败，则 AWS AppSync DynamoDB 函数在 ctx.error 中提供一个错误。您可以返回变更错误，并在 GraphQL 响应 error 部分的 data 字段中返回 DynamoDB 中的对象的当前值。

不过，AWS AppSync DynamoDB 函数提供了一些额外的功能，以帮助开发人员处理一些常见的边缘情况：

- 如果 AWS AppSync DynamoDB 函数可以确定 DynamoDB 中的当前值与所需的结果匹配，则会将该操作视为成功。
- 您可以将函数配置为调用自定义 Lambda 函数以决定 AWS AppSync DynamoDB 函数应如何处理失败，而不是返回错误。

在处理条件检查失败一节中更详细地介绍了这些内容。

有关 DynamoDB 条件表达式的更多信息，请参阅 DynamoDB 条件表达式文档。

指定条件

PutItem、UpdateItem 和 DeleteItem 请求对象都允许指定可选的 condition 部分。如果省略，则不会进行条件检查。如果指定，条件必须为 true，操作才能成功。

condition 部分具有以下结构：

```
type ConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
  equalsIgnore?: string[];
  consistentRead?: boolean;
  conditionalCheckFailedHandler?: {
    strategy: 'Custom' | 'Reject';
    lambdaArn?: string;
  };
};
```

下列字段指定条件：

expression

更新表达式本身。有关如何编写条件表达式的更多信息，请参阅 DynamoDB ConditionExpressions 文档。必须指定该字段。

expressionNames

以键值对形式替换表达式属性名称占位符。键对应于 expression 中使用的名称占位符，值必须是与 DynamoDB 中的项目的属性名称对应的字符串。该字段是可选的，只应填充 expression 中使用的表达式属性名称占位符的替换内容。

expressionValues

以键值对形式替换表达式属性值占位符。键对应于 expression 中使用的值占位符，而值必须为类型化值。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。必须指定此值。该字段是可选的，只应填充 expression 中使用的表达式属性值占位符的替换内容。

其余字段指示 AWS AppSync DynamoDB 函数如何处理条件检查失败：

equalsIgnore

如果在使用 PutItem 操作时条件检查失败，则 AWS AppSync DynamoDB 函数将当前位于 DynamoDB 中的项目与它尝试写入的项目进行比较。如果两者相同，则它会将操作视为已成功。您可以使用 equalsIgnore 字段指定 AWS AppSync 在执行该比较时应忽略的属性列表。例如，如果唯一的区别是 version 属性，则会将操作视为成功。该字段是可选的。

consistentRead

在条件检查失败时，AWS AppSync 使用强一致性读取从 DynamoDB 中获取项目的当前值。您可以使用该字段指示 AWS AppSync DynamoDB 函数改用最终一致性读取。该字段是可选的，默认值为 `true`。

conditionalCheckFailedHandler

在该部分中，您可以指定 AWS AppSync DynamoDB 函数在将 DynamoDB 中的当前值与预期结果进行比较后如何处理条件检查失败。此部分是可选的。如果省略，它默认为 `Reject` 策略。

strategy

AWS AppSync DynamoDB 函数在将 DynamoDB 中的当前值与预期结果进行比较后采取的策略。该字段为必填字段，有以下可能的值：

Reject

变更失败，将返回变更错误，并在 GraphQL 响应 `error` 部分的 `data` 字段中返回 DynamoDB 中的对象的当前值。

Custom

AWS AppSync DynamoDB 函数调用自定义 Lambda 函数，以决定如何处理条件检查失败。当 `strategy` 设置为 `Custom` 时，`lambdaArn` 字段必须包含要调用的 Lambda 函数的 ARN。

lambdaArn

要调用的 Lambda 函数的 ARN，用于确定 AWS AppSync DynamoDB 函数应如何处理条件检查失败。当 `strategy` 设置为 `Custom` 时，必须指定该字段。有关如何使用该功能的更多信息，请参阅处理条件检查失败。

处理条件检查失败

在条件检查失败时，AWS AppSync DynamoDB 函数可以使用 `util.appendError` 实用程序传递变更错误和对象的当前值。这会在 GraphQL 响应的 `error` 部分中添加 `data` 字段。不过，AWS AppSync DynamoDB 函数提供了一些额外的功能，以帮助开发人员处理一些常见的边缘情况：

- 如果 AWS AppSync DynamoDB 函数可以确定 DynamoDB 中的当前值与所需的结果匹配，则会将该操作视为成功。
- 您可以将函数配置为调用自定义 Lambda 函数以决定 AWS AppSync DynamoDB 函数应如何处理失败，而不是返回错误。

此过程的流程图为：

检查所需的结果

在条件检查失败时，AWS AppSync DynamoDB 函数执行 `GetItem` DynamoDB 请求，以从 DynamoDB 中获取项目的当前值。默认情况下，它将使用强一致性读取，但这可以使用 `condition` 数据块中的 `consistentRead` 字段进行配置，并将当前值与预期结果进行比较：

- 对于 `PutItem` 操作，AWS AppSync DynamoDB 函数将当前值与它尝试写入的值进行比较，以从比较中排除 `equalsIgnore` 中列出的任何属性。如果项目相同，则将操作视为成功并返回从 DynamoDB 中检索的项目。否则，它将遵循所配置的策略。

  例如，如果 `PutItem` 请求对象如下所示：

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  const { id, name, version} = ctx.args
  return {
    operation: 'PutItem',
    key: util.dynamodb.toMapValues({foo, bar}),
    attributeValues: util.dynamodb.toMapValues({ name, version: version+1 }),
    condition: {
      expression: "version = :expectedVersion",
      expressionValues: util.dynamodb.toMapValues({':expectedVersion': version}),
      equalsIgnore: ['version']
    }
  };
}
```

  当前位于 DynamoDB 中的项目如下所示：

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

  AWS AppSync DynamoDB 函数将它尝试写入的项目与当前值进行比较，发现唯一的区别是 `version` 字段，但由于它配置为忽略 `version` 字段，因此，将操作视为成功并返回从 DynamoDB 中检索的项目。

- 对于 DeleteItem 操作，AWS AppSync DynamoDB 函数检查以验证是否从 DynamoDB 返回了项目。如果没有返回项目，它会将该操作视为已成功。否则，它将遵循所配置的策略。
- 对于 UpdateItem 操作，AWS AppSync DynamoDB 函数没有足够的信息，无法确定当前位于 DynamoDB 中的项目是否与预期结果匹配，因此，将遵循配置的策略。

如果 DynamoDB 中的对象的当前状态与预期结果不同，则 AWS AppSync DynamoDB 函数按照配置的策略拒绝变更或调用 Lambda 函数以确定后续操作。

遵循“reject”策略

在遵循 Reject 策略时，AWS AppSync DynamoDB 函数返回变更错误，并且还会在 GraphQL 响应 error 部分的 data 字段中返回 DynamoDB 中的对象的当前值。从 DynamoDB 返回的项目通过函数响应处理程序转换为客户端的预期格式，并按选择集进行筛选。

例如，给定以下变更请求：

```
mutation {
    updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
        Name
        theVersion
    }
}
```

如果从 DynamoDB 返回的项目如下所示：

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

函数响应处理程序如下所示：

```
import { util } from '@aws-appsync/utils';
export function response(ctx) {
  const { version, ...values } = ctx.result;
  const result = { ...values, theVersion: version };
  if (ctx.error) {
    if (error) {
      return util.appendError(error.message, error.type, result, null);
    }
```

```
  }
  return result
}
```

GraphQL 响应如下所示：

```
{
  "data": null,
  "errors": [
    {
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "data": {
        "Name": "Steve",
        "theVersion": 8
      },
      ...
    }
  ]
}
```

另外，如果返回的对象中的任何字段在变更成功后已由其他解析器填充，则当在 `error` 部分中返回此对象时，将不会对这些字段进行解析。

遵循“custom”策略

在遵循 `Custom` 策略时，AWS AppSync DynamoDB 函数调用 Lambda 函数以决定后续操作。Lambda 函数选择下列选项之一：

- `reject` 变更。这会指示 AWS AppSync DynamoDB 函数的行为就像配置的策略是 `Reject` 一样，返回变更错误和 DynamoDB 中的对象的当前值，如上一节所述。
- `discard` 变更。这会指示 AWS AppSync DynamoDB 函数静默忽略条件检查失败并返回 DynamoDB 中的值。
- `retry` 变更。这会指示 AWS AppSync DynamoDB 函数使用新的请求对象重试变更。

Lambda 调用请求

AWS AppSync DynamoDB 函数调用 `lambdaArn` 中指定的 Lambda 函数。它将使用在数据源上配置的相同 `service-role-arn`。调用的负载具有以下结构：

```
{
    "arguments": { ... },
    "requestMapping": {... },
    "currentValue": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

字段定义如下：

arguments

来自 GraphQL 变更的参数。这与 context.arguments 中的请求对象的可用参数相同。

requestMapping

该操作的请求对象。

currentValue

DynamoDB 中的对象的当前值。

resolver

有关 AWS AppSync 解析器或函数的信息。

identity

有关调用方的信息。这与 context.identity 中的请求对象的可用身份信息相同。

负载的完整示例：

```
{
    "arguments": {
        "id": "1",
        "name": "Steve",
        "expectedVersion": 1
    },
    "requestMapping": {
        "version" : "2017-02-28",
        "operation" : "PutItem",
        "key" : {
           "id" : { "S" : "1" }
```

```
        },
        "attributeValues" : {
           "name" : { "S" : "Steve" },
           "version" : { "N" : 2 }
        },
        "condition" : {
           "expression" : "version = :expectedVersion",
           "expressionValues" : {
               ":expectedVersion" : { "N" : 1 }
           },
           "equalsIgnore": [ "version" ]
        }
    },
    "currentValue": {
        "id" : { "S" : "1" },
        "name" : { "S" : "Steve" },
        "version" : { "N" : 8 }
    },
    "resolver": {
        "tableName": "People",
        "awsRegion": "us-west-2",
        "parentType": "Mutation",
        "field": "updatePerson",
        "outputType": "Person"
    },
    "identity": {
        "accountId": "123456789012",
        "sourceIp": "x.x.x.x",
        "user": "AIDAAAAAAAAAAAAAAAAA",
        "userArn": "arn:aws:iam::123456789012:user/appsync"
    }
}
```

Lambda 调用响应

Lambda 函数可以检查调用负载，并应用任何业务逻辑以决定 AWS AppSync DynamoDB 函数应如何处理失败。有三种选项可用于处理条件检查失败：

- `reject` 变更。此选项的响应负载必须具有此结构：

```
{
    "action": "reject"
}
```

这会指示 AWS AppSync DynamoDB 解析器的行为就像配置的策略是 `Reject` 一样，返回变更错误和 DynamoDB 中的对象的当前值，如上一节所述。

- `discard` 变更。此选项的响应负载必须具有此结构：

```
{
    "action": "discard"
}
```

这会指示 AWS AppSync DynamoDB 函数静默忽略条件检查失败并返回 DynamoDB 中的值。

- `retry` 变更。此选项的响应负载必须具有此结构：

```
{
    "action": "retry",
    "retryMapping": { ... }
}
```

这会指示 AWS AppSync DynamoDB 函数使用新的请求对象重试变更。`retryMapping` 部分的结构取决于 DynamoDB 操作，并且是该操作的完整请求对象的子集。

对于 `PutItem`，`retryMapping` 部分具有以下结构。有关 `attributeValues` 字段的描述，请参阅 PutItem。

```
{
    "attributeValues": { ... },
    "condition": {
        "equalsIgnore" = [ ... ],
        "consistentRead" = true
    }
}
```

对于 `UpdateItem`，`retryMapping` 部分具有以下结构。有关 `update` 部分的说明，请参阅 UpdateItem。

```
{
    "update" : {
        "expression" : "someExpression"
        "expressionNames" : {
            "#foo" : "foo"
        },
```

```
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "condition": {
        "consistentRead" = true
    }
}
```

对于 DeleteItem，retryMapping 部分具有以下结构。

```
{
    "condition": {
        "consistentRead" = true
    }
}
```

无法指定要使用的不同的操作或键。AWS AppSync DynamoDB 函数仅允许对同一对象重试相同操作。另外，condition 部分不允许指定 conditionalCheckFailedHandler。如果重试失败，AWS AppSync DynamoDB 函数将遵循 Reject 策略。

在下面的示例中，Lambda 函数处理失败的 PutItem 请求。业务逻辑将查看进行调用的用户。如果调用是由 jeffTheAdmin 进行的，则会重试该请求，并从当前位于 DynamoDB 的项目中更新 version 和 expectedVersion。否则，业务逻辑将拒绝变更。

```
exports.handler = (event, context, callback) => {
    console.log("Event: "+ JSON.stringify(event));

    // Business logic goes here.

    var response;
    if ( event.identity.user == "jeffTheAdmin" ) {
        response = {
            "action" : "retry",
            "retryMapping" : {
                "attributeValues" : event.requestMapping.attributeValues,
                "condition" : {
                    "expression" : event.requestMapping.condition.expression,
                    "expressionValues" :
 event.requestMapping.condition.expressionValues
                }
```

```
            }
        }
        response.retryMapping.attributeValues.version = { "N" :
 event.currentValue.version.N + 1 }
        response.retryMapping.condition.expressionValues[':expectedVersion'] =
 event.currentValue.version

    } else {
        response = { "action" : "reject" }
    }

    console.log("Response: "+ JSON.stringify(response))
    callback(null, response)
};
```

事务条件表达式

可以在 TransactWriteItems 中的所有 4 种类型的操作（PutItem、DeleteItem、UpdateItem 和 ConditionCheck）的请求中使用事务条件表达式。

对于 PutItem、DeleteItem 和 UpdateItem，事务条件表达式是可选的。对于 ConditionCheck，事务条件表达式是必需的。

示例 1

以下事务 DeleteItem 函数请求处理程序没有条件表达式。因此，它删除 DynamoDB 中的项目。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { postId } = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'DeleteItem',
        key: util.dynamodb.toMapValues({ postId }),
      }
    ],
  };
}
```

示例 2

以下事务 DeleteItem 函数请求处理程序确实具有一个事务条件表达式，只有在该文章的作者等于特定姓名时，操作才会成功。

```
import { util } from '@aws-appsync/utils';

export function request(ctx) {
  const { postId, authorName} = ctx.args;
  return {
    operation: 'TransactWriteItems',
    transactItems: [
      {
        table: 'posts',
        operation: 'DeleteItem',
        key: util.dynamodb.toMapValues({ postId }),
        condition: util.transform.toDynamoDBConditionExpression({
          authorName: { eq: authorName },
        }),
      }
    ],
  };
}
```

如果条件检查失败，则会导致 TransactionCanceledException，错误详细信息将在 ctx.result.cancellationReasons 中返回。请注意，默认情况下，DynamoDB 中导致条件检查失败的旧项目将在 ctx.result.cancellationReasons 中返回。

指定条件

PutItem、UpdateItem 和 DeleteItem 请求对象都允许指定可选的 condition 部分。如果省略，则不会进行条件检查。如果指定，条件必须为 true，操作才能成功。ConditionCheck 必须具有要指定的 condition 部分。条件必须为 true，整个事务才能成功。

condition 部分具有以下结构：

```
type TransactConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string };
  expressionValues?: { [key: string]: string };
  returnValuesOnConditionCheckFailure: boolean;
};
```

下列字段指定条件：

expression

更新表达式本身。有关如何编写条件表达式的更多信息，请参阅 DynamoDB ConditionExpressions 文档。必须指定该字段。

expressionNames

以键值对形式替换表达式属性名称占位符。键对应于 expression 中使用的名称占位符，值必须是与 DynamoDB 中的项目的属性名称对应的字符串。该字段是可选的，只应填充 expression 中使用的表达式属性名称占位符的替换内容。

expressionValues

以键值对形式替换表达式属性值占位符。键对应于 expression 中使用的值占位符，而值必须为类型化值。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。必须指定此值。该字段是可选的，只应填充 expression 中使用的表达式属性值占位符的替换内容。

returnValuesOnConditionCheckFailure

指定在条件检查失败时是否重新检索 DynamoDB 中的项目。检索到的项目将位于 `ctx.result.cancellationReasons[<index>].item` 中，其中 `<index>` 是未通过条件检查的请求项目的索引。该值默认为 true。

投影

在使用 `GetItem`、`Scan`、`Query`、`BatchGetItem` 和 `TransactGetItems` 操作读取 DynamoDB 中的对象时，您可以选择指定一个投影以指定所需的属性。投影属性具有以下结构，与筛选条件类似：

```
type DynamoDBExpression = {
  expression: string;
  expressionNames?: { [key: string]: string}
};
```

字段定义如下：

expression

投影表达式，它是一个字符串。要检索单个属性，请指定其名称。对于多个属性，名称必须是逗号分隔值。有关编写投影表达式的更多信息，请参阅 DynamoDB 投影表达式文档。该字段为必填。

`expressionNames`

以键值对形式替换表达式属性名称占位符。键对应于 `expression` 中使用的名称占位符。该值必须是与 DynamoDB 中的项目的属性名称对应的字符串。该字段是可选的，只应填充 `expression` 中使用的表达式属性名称占位符的替换内容。有关 `expressionNames` 的更多信息，请参阅 DynamoDB 文档。

示例 1

以下示例是 JavaScript 函数的投影部分，其中仅从 DynamoDB 返回 `author` 和 `id` 属性：

```
projection : {
    expression : "#author, id",
    expressionNames : {
        "#author" : "author"
    }
}
```

> **Tip**
>
> 您可以使用 selectionSetList 访问 GraphQL 请求选择集。可以通过该字段根据您的要求动态构建投影表达式。

> **Note**
>
> 在将投影表达式与 `Query` 和 `Scan` 运算一起使用时，`select` 的值必须为 `SPECIFIC_ATTRIBUTES`。有关更多信息，请参阅 DynamoDB 文档。

OpenSearch 的 JavaScript 解析器函数参考

通过使用适用于 Amazon OpenSearch Service 的 AWS AppSync 解析器，您可以使用 GraphQL 在您的账户的现有 OpenSearch Service 域中存储和检索数据。该解析器的工作方式是，允许您将传入的 GraphQL 请求映射到 OpenSearch Service 请求，然后将 OpenSearch Service 响应映射回 GraphQL。本节介绍了支持的 OpenSearch Service 操作的函数请求和响应处理程序。

请求

大多数 OpenSearch Service 请求对象具有通用结构，其中仅几个部分发生变化。以下示例对 OpenSearch Service 域运行搜索，其中文档的类型为 `post` 并按照 `id` 编制索引。搜索参数在 `body` 部分定义，而许多常用查询子句在 `query` 字段中定义。此示例将搜索在文档的 `"Nadia"` 字段中包含 `"Bailey"` 和/或 `author` 的文档。

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: {
        from: 0,
        size: 50,
        query: {
          bool: {
            should: [
              { match: { author: 'Nadia' } },
              { match: { author: 'Bailey' } },
            ],
          },
        },
      },
    },
  };
}
```

响应

与其他数据源一样，OpenSearch Service 向 AWS AppSync 发送响应，该响应需要转换为 GraphQL。

大多数 GraphQL 查询从 OpenSearch Service 响应中查找 `_source` 字段。由于您可以搜索以返回单个文档或文档列表，因此，可以在 OpenSearch Service 中使用两种常见的响应模式：

结果列表

```
export function response(ctx) {
  const entries = [];
  for (const entry of ctx.result.hits.hits) {
```

```
    entries.push(entry['_source']);
  }
  return entries;
}
```

单个项目

```
export function response(ctx) {
  return ctx.result['_source']
}
```

operation 字段

（仅请求处理程序）

AWS AppSync 发送到 OpenSearch Service 域的 HTTP 方法或动词（GET、POST、PUT、HEAD 或 DELETE）。键和值都必须是字符串。

```
"operation" : "PUT"
```

path 字段

（仅请求处理程序）

来自 AWS AppSync 的 OpenSearch Service 请求的搜索路径。这构成了操作的 HTTP 谓词的 URL。键和值都必须是字符串。

```
"path" : "/indexname/type"

"path" : "/indexname/type/_search"
```

在评估请求处理程序时，该路径将作为 HTTP 请求的一部分发送，包括 OpenSearch Service 域。例如，上一个示例可能会转换为：

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

params 字段

（仅请求处理程序）

用于指定搜索执行的操作，最常见的是在正文中设置查询值。但是，可以配置若干其他功能，如响应的格式设置。

- headers

  标头信息（为键值对）。键和值都必须是字符串。例如：

```
"headers" : {
    "Content-Type" : "application/json"
}
```

> **Note**
>
> AWS AppSync 目前仅支持将 JSON 作为 `Content-Type`。

- queryString

  指定常用选项的键值对，如 JSON 响应的代码格式设置。键和值都必须是字符串。例如，如果您要获得格式正确的 JSON，应使用：

```
"queryString" : {
    "pretty" : "true"
}
```

- body

  这是请求的主要部分，它允许 AWS AppSync 为您的 OpenSearch Service 域创建正确格式的搜索请求。键必须是组成对象的一个字符串。下面介绍了几个演示。

示例 1

返回城市与“seattle”匹配的所有文档：

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
```

```
      body: { from: 0, size: 50, query: { match: { city: 'seattle' } } },
    },
  };
}
```

示例 2

返回将“washington”作为城市或州匹配的所有文档。

```
export function request(ctx) {
  return {
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: {
        from: 0,
        size: 50,
        query: {
          multi_match: { query: 'washington', fields: ['city', 'state'] },
        },
      },
    },
  };
}
```

传递变量

（仅请求处理程序）

您也可以在评估请求处理程序期间传递变量。例如，假设您具有以下 GraphQL 查询，如下所示：

```
query {
    searchForState(state: "washington"){
        ...
    }
}
```

函数请求处理程序可能如下所示：

```
export function request(ctx) {
  return {
```

```
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: {
        from: 0,
        size: 50,
        query: {
          multi_match: { query: ctx.args.state, fields: ['city', 'state'] },
        },
      },
    },
  };
}
```

JavaScript Lambda 的解析器函数参考

您可以使用 AWS AppSync 函数和解析器来调用位于您账户中的 Lambda 函数。在将请求负载和 Lambda 函数返回给客户之前，您可以调整请求负载和 Lambda 函数的响应。您还可以指定在请求对象中执行的操作类型。本节介绍了支持的 Lambda 操作的请求。

请求对象

Lambda 请求对象处理与您的 Lambda 函数相关的字段：

```
export type LambdaRequest = {
  operation: 'Invoke' | 'BatchInvoke';
  invocationType?: 'RequestResponse' | 'Event';
  payload: unknown;
};
```

以下示例使用了一个`invoke`操作，其有效载荷数据是 GraphQL 架构中的`getPost`字段，其参数来自上下文：

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

整个映射文档将作为输入传递给您的 Lambda 函数，因此前面的示例现在如下所示：

```
{
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "input": {
        "id": "postId1",
      }
    }
  }
}
```

操作

Lambda 数据源允许您在operation字段中定义两个操作：Invoke和。BatchInvoke该Invoke操作允许 AWS AppSync 你为每个 GraphQL 字段解析器调用你的 Lambda 函数。 BatchInvoke指示 AWS AppSync 对当前 GraphQL 字段进行批量请求。operation 字段为必填项。

对于Invoke，已解析的请求与 Lambda 函数的输入有效负载相匹配。让我们修改上面的示例：

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

此问题已解决并传递给 Lambda 函数，该函数可能如下所示：

```
{
  "operation": "Invoke",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

对于BatchInvoke，请求将应用于批次中的每个字段解析器。为简洁起见，将所有请求payload值 AWS AppSync 合并到与请求对象匹配的单个对象下的列表中。以下示例请求处理程序显示了合并：

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: ctx,
  };
}
```

将评估该请求并解析为以下映射文档：

```
{
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

payload列表中的每个元素对应一个批次项目。Lambda 函数还应返回与请求中发送的项目顺序相匹配的列表形响应：

```
[
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
 1
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
 2
  { "data": {...}, "errorMessage": null, "errorType": null }  // result for batch item
 3
]
```

有效负载

该payload字段是一个容器，用于将任何数据传递给 Lambda 函数。如果该operation字段设置为BatchInvoke，则 AWS AppSync 将现有payload值换成一个列表。payload 字段为可选项。

调用类型

Lambda 数据源允许您定义两种调用类型：和。RequestResponse Event调用类型与 Lambda API 中定义的调用类型同义。RequestResponse调用类型允许 AWS AppSync 同步调用您的 Lambda 函数以等待响应。该Event调用允许您异步调用 Lambda 函数。有关 Lambda 如何处理Event调用类型

请求的更多信息，请参阅异步调用。invocationType 字段为可选项。如果请求中未包含此字段，AWS AppSync 则默认为RequestResponse调用类型。

对于任何invocationType字段，已解析的请求都与 Lambda 函数的输入负载相匹配。让我们修改上面的示例：

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    invocationType: 'Event',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

此问题已解决并传递给 Lambda 函数，该函数可能如下所示：

```
{
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

当该BatchInvoke操作与Event调用类型字段结合使用时，将以与上述相同的方式 AWS AppSync 合并字段解析器，然后将请求作为异步事件传递给您的 Lambda 函数，并以值列表的形式传送给您的 Lambda 函数。payload来自Event调用类型请求的响应会生成一个没有响应处理程序的null值：

```
{
  "data": {
    "field": null
  }
}
```

我们建议您禁用Event调用类型解析器的解析器缓存，因为如果出现缓存命中，这些解析器不会发送到 Lambda。

响应对象

与其他数据源一样，您的 Lambda 函数会向其发送一个必须转换为 GraphQL 类型的响应。 AWS AppSync Lambda 函数的结果包含在`context`结果属性 () `context.result` 中。

如果您的 Lambda 函数响应的形状与 GraphQL 类型的形状相匹配，则可以使用以下函数响应处理程序转发响应：

```
export function response(ctx) {
  return ctx.result
}
```

没有适用于响应对象的必填字段或形状限制。不过，由于 GraphQL 是强类型的，因此，解析的响应必须与预期的 GraphQL 类型匹配。

Lambda 函数批处理的响应

如果该`operation`字段设置为`BatchInvoke`， AWS AppSync则需要从 Lambda 函数返回的项目列表。 AWS AppSync 为了将每个结果映射回原始请求项，响应列表的大小和顺序必须匹配。在响应列表中包含`null`项目是有效`ctx.result`的；相应地设置为 null。

JavaScript EventBridge 数据源的解析器函数参考

与 EventBridge 数据源一起使用的 AWS AppSync 解析器函数请求和响应允许您向 Amazon EventBridge 总线发送自定义事件。

请求

请求处理程序允许您将多个自定义事件发送到 EventBridge 事件总线：

```
export function request(ctx) {
  return {
    "operation" : "PutEvents",
    "events" : [{}]
  }
}
```

EventBridge `PutEvents`请求的类型定义如下：

```
type PutEventsRequest = {
  operation: 'PutEvents'
  events: {
    source: string
    detail: { [key: string]: any }
    detailType: string
    resources?: string[]
    time?: string // RFC3339 Timestamp format
  }[]
}
```

响应

如果PutEvents操作成功， EventBridge 则来自的响应将包含在ctx.result：

```
export function response(ctx) {
  if(ctx.error)
    util.error(ctx.error.message, ctx.error.type, ctx.result)
  else
    return ctx.result
}
```

执行 PutEvents 操作时发生的错误（例如 InternalExceptions 或 Timeouts）将出现在 ctx.error 中。有关常见错误 EventBridge的列表，请参阅EventBridge 常见错误参考。

result 将具有以下类型定义：

```
type PutEventsResult = {
  Entries: {
    ErrorCode: string
    ErrorMessage: string
    EventId: string
  }[]
  FailedEntryCount: number
}
```

- Entries

  摄取的事件结果（成功和失败）。如果摄取成功，则在该条目中包含 EventID。否则，您可以使用 ErrorCode 和 ErrorMessage 找出条目的问题。

对于每条记录，响应元素的索引与请求数组中的索引相同。

- FailedEntryCount

  失败条目数。该值表示为一个整数。

有关响应的更多信息PutEvents，请参阅PutEvents。

示例响应 1

以下示例是具有两个成功事件的 PutEvents 操作：

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ],
    "FailedEntryCount" : 0
}
```

示例响应 2

以下示例是具有三个事件的 PutEvents 操作，其中的两个事件是成功事件，一个是失败事件：

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        },
        {
            "ErrorCode" : "SampleErrorCode",
            "ErrorMessage" : "Sample Error Message"
        }
    ],
    "FailedEntryCount" : 1
```

```
}
```

PutEvents 字段

- 版本

  version 字段是所有请求映射模板通用的，用于定义模板使用的版本。该字段为必填。该值2018-05-29是 EventBridge 映射模板支持的唯一版本。

- 操作

  唯一支持的操作是 PutEvents。通过执行该操作，您可以将自定义事件添加到事件总线中。

- 事件

  将添加到事件总线的事件数组。应该为该数组分配 1-10 个项目。

  Event 对象具有以下字段：

  - "source"：定义事件源的字符串。
  - "detail"：可用于附加事件相关信息的 JSON 对象。该字段可以是空映射 ({ })。
  - "detailType：指定事件类型的字符串。
  - "resources"：指定事件中涉及的资源的 JSON 字符串数组。该字段可以是空数组。
  - "time"：以字符串形式提供的事件时间戳。它应该采用 RFC3339 时间戳格式。

以下片段是一些有效 Event 对象示例：

示例 1

```
{
    "source" : "source1",
    "detail" : {
        "key1" : [1,2,3,4],
        "key2" : "strval"
    },
    "detailType" : "sampleDetailType",
    "resources" : ["Resouce1", "Resource2"],
    "time" : "2022-01-10T05:00:10Z"
}
```

示例 2

```
{
    "source" : "source1",
    "detail" : {},
    "detailType" : "sampleDetailType"
}
```

示例 3

```
{
    "source" : "source1",
    "detail" : {
        "key1" : 1200
    },
    "detailType" : "sampleDetailType",
    "resources" : []
}
```

None 数据源的 JavaScript 解析器函数参考

通过使用 None 类型的数据源的 AWS AppSync 解析器函数请求和响应，您可以设置 AWS AppSync 本地操作的请求形状。

请求

请求处理程序可能很简单，并允许您通过 payload 字段传递尽可能多的上下文信息。

```
type NONERequest = {
  payload: any;
};
```

以下是一个将字段参数传递给负载的示例：

```
export function request(ctx) {
  return {
    payload: context.args
  };
}
```

payload 字段的值将转发到函数响应处理程序，并且可以在 context.result 中使用。

有效负载

payload 字段是一个容器，可用于传递任何数据以供函数响应处理程序使用。

payload 字段为可选项。

响应

由于没有数据源，payload 字段的值将转发到函数响应处理程序，并在 context.result 属性上设置该值。

如果 payload 字段值的形状与 GraphQL 类型的形状完全匹配，您可以使用以下响应处理程序转发响应：

```
export function request(ctx) {
  return ctx.result;
}
```

没有适用于返回响应的必填字段或形状限制。不过，由于 GraphQL 是强类型的，因此，解析的响应必须与预期的 GraphQL 类型匹配。

JavaScript HTTP 的解析器函数参考

使用 AWS AppSync HTTP 解析器函数，您可以将来自的请求发送AWS AppSync 到任何 HTTP 端点，并将来自您的 HTTP 端点的响应发送回到AWS AppSync。使用您的请求处理程序，您可以提供AWS AppSync 有关要调用的操作性质的提示。本节介绍了支持的 HTTP 解析器的各种配置。

请求

```
type HTTPRequest = {
  method: 'PUT' | 'POST' | 'GET' | 'DELETE' | 'PATCH';
  params?: {
    query?: { [key: string]: any };
    headers?: { [key: string]: string };
    body?: any;
  };
  resourcePath: string;
};
```

以下代码片段是一个 HTTP POST 请求示例，其正文为 `text/plain`：

```
export function request(ctx) {
  return {
    method: 'POST',
    params: {
      headers: { 'Content-Type': 'text/plain' },
      body: 'this is an example of text body',
    },
    resourcePath: '/',
  };
}
```

方法

仅请求处理程序

AWS AppSync 发送到 HTTP 终端节点的 HTTP 方法或动词（GET、POST、PUT、PATCH 或 DELETE）。

```
"method": "PUT"
```

ResourcePath

仅请求处理程序

您要访问的资源路径。资源路径与 HTTP 数据源中更多终端节点一起，构成 AWS AppSync 发出请求到的 URL。

```
"resourcePath": "/v1/users"
```

在评估请求时，该路径将作为 HTTP 请求的一部分发送，包括 HTTP 终端节点。例如，上一个示例可能会转换为如下所示：

```
PUT <endpoint>/v1/users
```

Params 字段

仅请求处理程序

用于指定搜索执行的操作，最常见的是在正文中设置查询值。但是，可以配置若干其他功能，如响应的格式设置。

headers

标头信息（为键值对）。键和值都必须是字符串。

例如：

```
"headers" : {
    "Content-Type" : "application/json"
}
```

目前支持的 Content-Type 标头包括：

```
text/*
application/xml
application/json
application/soap+xml
application/x-amz-json-1.0
application/x-amz-json-1.1
application/vnd.api+json
application/x-ndjson
```

您无法设置以下 HTTP 标头：

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

指定常用选项的键值对，如 JSON 响应的代码格式设置。键和值都必须是字符串。以下示例显示如何发送 ?type=json 格式的请求字符串：

```
"query" : {
    "type" : "json"
}
```

body

正文包含您选择要设置的 HTTP 请求正文。请求正文始终是 UTF-8 编码格式的字符串，除非内容类型指定字符集。

```
"body":"body string"
```

响应

请参见此处的示例。

JavaScript Amazon RDS 的解析器函数参考

AWS AppSync RDS 函数和解析器允许开发人员使用 RDS 数据 API 向 Amazon Aurora 集群数据库发送SQL查询，并获取这些查询的结果。您可以使用带有模块`sql`标签的模板或使用`rds`模块AWS AppSync的`select`、`insert`、和`remove`帮助函数来编写发送到 Data API 的SQL语句。`rds` `update` AWS AppSync 利用 RDS 数据服务的`ExecuteStatement`操作对数据库运行 SQL 语句。

主题

- 带 SQL 标签的模板
- 创建语句
- 检索数据
- 实用程序函数
- SQL 选择
- SQL 插入
- SQL 更新
- SQL 删除
- 转换

带 SQL 标签的模板

AWS AppSync的`sql`标记模板使您能够使用模板表达式创建可在运行时接收动态值的静态语句。 AWS AppSync 根据表达式值构建变量映射以构造发送到 Amazon Aurora 无服务器数据 API

的SqlParameterized查询。使用此方法，运行时传递的动态值不可能修改原始语句，这可能会导致意外执行。所有动态值都作为参数传递，不能修改原始语句，也不会由数据库执行。这使您的查询不太容易受到 SQL 注入攻击。

> **Note**
>
> 在所有情况下，在编写 SQL 语句时，都应遵循安全准则，以正确处理作为输入收到的数据。

> **Note**
>
> 带 `sql` 标签的模板仅支持传递变量值。不能使用表达式来动态指定列名称或表名称。但是，您可以使用实用程序函数来构建动态语句。

在以下示例中，我们创建了一个查询，该查询根据运行时在 GraphQL 查询中动态设置的 `col` 参数值进行筛选。只能使用标签表达式将该值添加到语句中：

```
import { sql, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
  const query = sql`
SELECT * FROM table
WHERE column = ${ctx.args.col}`
  ;
  return createMySQLStatement(query);
}
```

借助于通过变量映射传递所有动态值，我们依靠数据库引擎来安全地处理和清理值。

创建语句

函数和解析器可以与 MySQL 和 PostgreSQL 数据库进行交互。分别使用 `createMySQLStatement` 和 `createPgStatement` 来构建语句。例如，`createMySQLStatement` 可以创建 MySQL 查询。这些函数最多可接受两条语句，在请求应立即检索结果时很有用。使用 MySQL，您可以执行：

```
import { sql, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { id, text } = ctx.args;
```

```
    const s1 = sql`insert into Post(id, text) values(${id}, ${text})`;
    const s2 = sql`select * from Post where id = ${id}`;
    return createMySQLStatement(s1, s2);
}
```

> **Note**
>
> `createPgStatement` 和 `createMySQLStatement` 不转义或引用使用带 `sql` 标签的模板构建的语句。

检索数据

您执行的 SQL 语句的结果可在响应处理程序的 `context.result` 对象中提供。结果是一个 JSON 字符串，其中包含来自 `ExecuteStatement` 操作的响应元素。解析后，结果具有以下形状：

```
type SQLStatementResults = {
    sqlStatementResults: {
        records: any[];
        columnMetadata: any[];
        numberOfRecordsUpdated: number;
        generatedFields?: any[]
    }[]
}
```

您可以使用 `toJsonObject` 实用程序将结果转换为表示返回行的 JSON 对象列表。例如：

```
import { toJsonObject } from '@aws-appsync/utils/rds';

export function response(ctx) {
    const { error, result } = ctx;
    if (error) {
        return util.appendError(
            error.message,
            error.type,
            result
        )
    }
    return toJsonObject(result)[1][0]
}
```

请注意，toJsonObject 返回语句结果数组。如果您提供了一条语句，则数组长度为 1。如果您提供了两条语句，则数组长度为 2。数组中的每个结果都包含 0 行或多行。如果结果值无效或不符合预期，则 toJsonObject 返回 null。

实用程序函数

您可以使用 AWS AppSync RDS 模块的实用工具帮助程序与您的数据库进行交互。

SQL 选择

select 实用程序会创建一条 SELECT 语句来查询您的关系数据库。

基本用法

在其基本形式中，您可以指定要查询的表：

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // "SELECT * FROM "persons"
    return createPgStatement(select({table: 'persons'}));
}
```

请注意，您也可以在表标识符中指定架构：

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // SELECT * FROM "private"."persons"
    return createPgStatement(select({table: 'private.persons'}));
}
```

指定列

您可以使用 columns 属性指定列。如果未将其设置为某个值，则它默认为 *：

```
export function request(ctx) {

    // Generates statement:
```

```
    // SELECT "id", "name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name']
    }));
}
```

您也可以指定列的表：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "persons"."name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'persons.name']
    }));
}
```

限制和偏移

您可以将 `limit` 和 `offset` 应用于查询：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // LIMIT :limit
    // OFFSET :offset
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        limit: 10,
        offset: 40
    }));
}
```

排序依据

您可以使用 `orderBy` 属性对结果进行排序。提供指定列和可选 `dir` 属性的对象数组：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name" FROM "persons"
    // ORDER BY "name", "id" DESC
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        orderBy: [{column: 'name'}, {column: 'id', dir: 'DESC'}]
    }));
}
```

筛选器

您可以使用特殊条件对象来构建筛选条件：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}}
    }));
}
```

您也可以组合筛选条件：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME and "id" > :ID
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: {name: {eq: 'Stephane'}, id: {gt: 10}}
    }));
```

```
}
```

您也可以创建 OR 语句：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE "name" = :NAME OR "id" > :ID
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: { or: [
            { name: { eq: 'Stephane'} },
            { id: { gt: 10 } }
        ]}
    }));
}
```

您也可以使用 not 来否定条件：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    // WHERE NOT ("name" = :NAME AND "id" > :ID)
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name'],
        where: { not: [
            { name: { eq: 'Stephane'} },
            { id: { gt: 10 } }
        ]}
    }));
}
```

您也可以使用以下运算符来比较值：

运算符	描述	可能的值类型

eq	Equal	数字、字符串、布尔值
没有	Not equal	数字、字符串、布尔值
le	Less than or equal	数字，字符串
lt	Less than	数字，字符串
ge	Greater than or equal	数字，字符串
gt	Greater than	数字，字符串
contains	喜欢	字符串
不包含	不像	字符串
开始于	以前缀开头	字符串
介于	在两个值之间	数字，字符串
属性存在	该属性不为空	数字、字符串、布尔值
size	检查元素的长度	字符串

SQL 插入

`insert` 实用程序提供了一种通过 `INSERT` 操作在数据库中插入单行项目的简单方法。

单个项目插入

要插入项目，请指定表，然后传入您的值对象。对象键映射到您的表列。列名称会自动转义，并使用变量映射将值发送到数据库：

```
import { insert, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({ table: 'persons', values });

    // Generates statement:
    // INSERT INTO `persons`(`name`)
    // VALUES(:NAME)
```

```
    return createMySQLStatement(insertStatement)
}
```

MySQL 用例

您可以组合 `insert` 后跟 `select` 来检索您插入的行：

```
import { insert, select, createMySQLStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({  table: 'persons', values });
    const selectStatement = select({
        table: 'persons',
        columns: '*',
        where: { id: { eq: values.id } },
        limit: 1,
    });

    // Generates statement:
    // INSERT INTO `persons`(`name`)
    // VALUES(:NAME)
    // and
    // SELECT *
    // FROM `persons`
    // WHERE `id` = :ID
    return createMySQLStatement(insertStatement, selectStatement)
}
```

Postgres 用例

借助 Postgres，您可以使用 `returning` 从插入的行中获取数据。它接受 `*` 或列名称数组：

```
import { insert, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: values } = ctx.args;
    const insertStatement = insert({
        table: 'persons',
        values,
        returning: '*'
    });
```

```
    // Generates statement:
    // INSERT INTO "persons"("name")
    // VALUES(:NAME)
    // RETURNING *
    return createPgStatement(insertStatement)
}
```

SQL 更新

update 实用程序允许您更新现有行。您可以使用条件对象将更改应用于满足条件的所有行中的指定列。例如，假设我们有一个允许我们进行这种突变的架构。我们要将 Person 的 name 更新为 id 值 3，但仅限我们自 2000 年开始就已知道它们 (known_since)：

```
mutation Update {
    updatePerson(
        input: {id: 3, name: "Jon"},
        condition: {known_since: {ge: "2000"}}
    ) {
    id
    name
  }
}
```

更新解析器如下所示：

```
import { update, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: { id, ...values }, condition } = ctx.args;
    const where = {
        ...condition,
        id: { eq: id },
    };
    const updateStatement = update({
        table: 'persons',
        values,
        where,
        returning: ['id', 'name'],
    });

    // Generates statement:
```

```
    // UPDATE "persons"
    // SET "name" = :NAME, "birthday" = :BDAY, "country" = :COUNTRY
    // WHERE "id" = :ID
    // RETURNING "id", "name"
    return createPgStatement(updateStatement)
}
```

我们可以在条件中添加一项检查，以确保只更新主键 `id` 等于 3 的行。同样，对于 Postgres `inserts`，您可以使用 `returning` 返回修改后的数据。

SQL 删除

`remove` 实用程序允许您删除现有行。您可以在满足条件的所有行上使用条件对象。请注意，`delete`这是中的保留关键字 JavaScript。 `remove`应该改用：

```
import { remove, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const { input: { id }, condition } = ctx.args;
    const where = { ...condition, id: { eq: id } };
    const deleteStatement = remove({
        table: 'persons',
        where,
        returning: ['id', 'name'],
    });

    // Generates statement:
    // DELETE "persons"
    // WHERE "id" = :ID
    // RETURNING "id", "name"
    return createPgStatement(updateStatement)
}
```

转换

在某些情况下，您可能希望在语句中使用更具体的正确对象类型。您可以使用提供的类型提示来指定参数的类型。 AWS AppSync 支持与数据 API 相同的类型提示。你可以使用 AWS AppSync `rds`模块中的`typeHint`函数来转换参数。

以下示例允许您将数组作为强制转换为 JSON 对象的值发送。我们使用 `->` 运算符来检索 JSON 数组中 `index` 为 2 的元素：

```
import { sql, createPgStatement, toJsonObject, typeHint } from '@aws-appsync/utils/
rds';

export function request(ctx) {
    const arr = ctx.args.list_of_ids
    const statement = sql`select ${typeHint.JSON(arr)}->2 as value`
    return createPgStatement(statement)
}

export function response(ctx) {
    return toJsonObject(ctx.result)[0][0].value
}
```

在处理和比较 DATE、TIME 和 TIMESTAMP 时，强制转换也很有用：

```
import { select, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const when = ctx.args.when
    const statement = select({
        table: 'persons',
        where: { createdAt : { gt: typeHint.DATETIME(when) } }
    })
    return createPgStatement(statement)
}
```

下面是另一个示例，显示如何发送当前日期和时间：

```
import { sql, createPgStatement, typeHint } from '@aws-appsync/utils/rds';

export function request(ctx) {
    const now = util.time.nowFormatted('YYYY-MM-dd HH:mm:ss')
    return createPgStatement(sql`select ${typeHint.TIMESTAMP(now)}`)
}
```

可用的类型提示

- typeHint.DATE – 相应的参数作为 DATE 类型的对象发送到数据库。接受的格式为 YYYY-MM-DD。
- typeHint.DECIMAL – 相应的参数作为 DECIMAL 类型的对象发送到数据库。
- typeHint.JSON – 相应的参数作为 JSON 类型的对象发送到数据库。

- typeHint.TIME – 相应的字符串参数值作为 TIME 类型的对象发送到数据库。接受的格式为 HH:MM:SS[.FFF]。
- typeHint.TIMESTAMP – 相应的字符串参数值作为 TIMESTAMP 类型的对象发送到数据库。接受的格式为 YYYY-MM-DD HH:MM:SS[.FFF]。
- typeHint.UUID – 相应的字符串参数值作为 UUID 类型的对象发送到数据库。

解析器映射模板参考 (VTL)

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

以下几节介绍了如何在映射模板中使用实用程序操作。

主题

解析器映射模板概述

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

在 AWS AppSync 中，您可以对资源执行操作以响应 GraphQL 请求。对于您希望运行查询或变更的每个 GraphQL 字段，必须附加解析器才能与数据源通信。通常，通信是通过数据源特有的参数或操作完成的。

解析器是 GraphQL 和数据源之间的连接器。它们指示 AWS AppSync 如何将传入的 GraphQL 请求转换为后端数据源的指令，以及如何将该数据源的响应转换回 GraphQL 响应。它们是使用 Apache Velocity 模板语言 (VTL) 编写的，该语言将您的请求作为输入，并输出包含解析器指令的 JSON 文档。您可以使用映射模板执行简单的指令（例如，从 GraphQL 字段中传入参数），也可以执行更复杂的指令（例如，循环访问各个参数以构建项目，然后将该项目插入到 DynamoDB 中）。

在 AWS AppSync 中具有两种类型的解析器，它们以略微不同的方式使用映射模板：

- 单位解析器
- 管道解析器

单位解析器

单位解析器是独立的实体，仅包含请求和响应模板。将它们用于简单、单一的操作（例如，列出来自单个数据源的项目）。

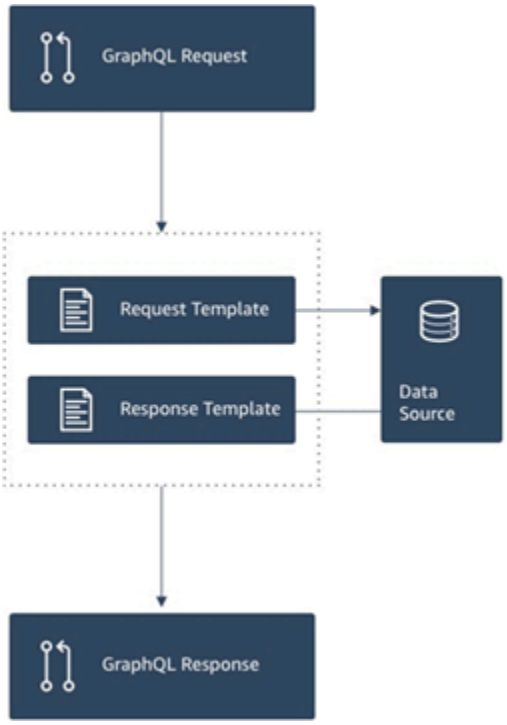
- 请求模板：在解析 GraphQL 操作后获取传入请求，并将其转换为选定数据源操作的请求配置。
- 响应模板：解释来自数据源的响应，并将其映射到 GraphQL 字段输出类型的形状。

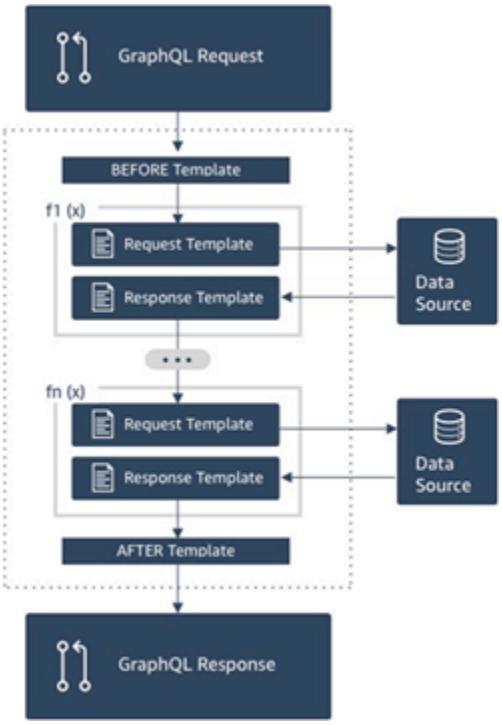
管道解析器

管道解析器包含一个或多个按顺序执行的函数。每个函数包含一个请求模板和一个响应模板。管道解析器还具有一个之前 模板和一个之后 模板，它们位于模板包含的一系列函数两侧。之后 模板映射到 GraphQL 字段输出类型。管道解析器与单位解析器的不同之处在于响应模板映射输出的方式。管道解析器可以映射到所需的任何输出，包括另一个函数的输入或管道解析器的之后 模板。

管道解析器函数 允许您编写可在架构中的多个解析器之间重复使用的通用逻辑。函数直接附加到数据源，并且像单位解析器一样，包含相同的请求和响应映射模板格式。

下图说明了左侧单位解析器和右侧管道解析器的流程。





管道解析器包含单位解析器支持的功能的超集以及更多功能，但代价是稍微复杂一些。

管道解析器剖析

管道解析器由之前映射模板、之后映射模板和一组函数组成。每个函数具有对数据源执行的请求映射模板和响应映射模板。由于管道解析器将执行委托给函数列表，因此它不会链接到任何数据源。单位解析器和函数是对数据源执行操作的基元。有关更多信息，请参阅解析器映射模板概述。

之前映射模板

管道解析器的请求映射模板或预备步骤允许您在执行定义的函数之前执行一些准备逻辑。

函数列表

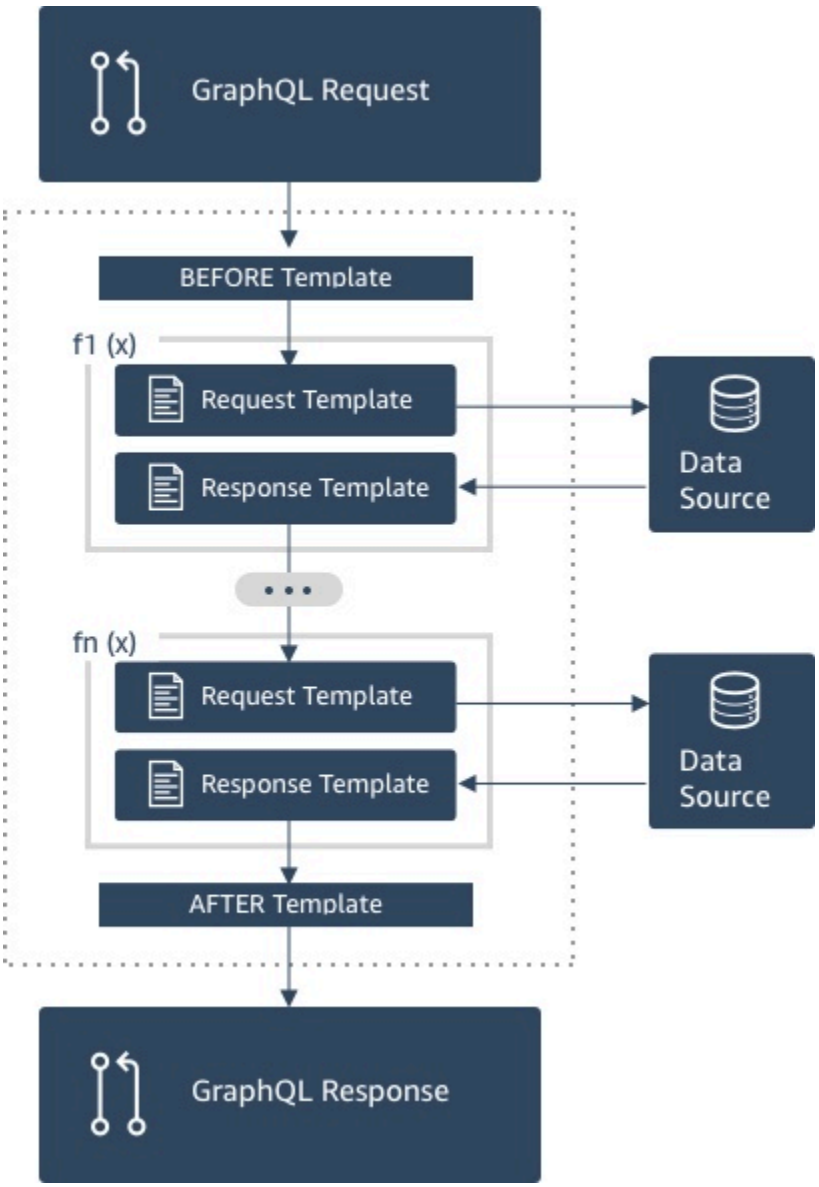
管道解析器将按顺序运行的函数的列表。管道解析器请求映射模板评估的结果可供第一个函数用作 `$ctx.prev.result`。每个函数输出可供下一个函数用作 `$ctx.prev.result`。

之后映射模板

管道解析器的响应映射模板或后续步骤允许您执行从最后一个函数的输出到预期 GraphQL 字段类型的一些最终映射逻辑。函数列表中最后一个函数的输出可在管道解析器映射模板中用作 `$ctx.prev.result` 或 `$ctx.result`。

执行流程

假定一个管道解析器由两个函数组成，下面的列表表示调用解析器时的执行流程：



1. 管道解析器的之前映射模板
2. 函数 1：函数请求映射模板
3. 函数 1：数据源调用
4. 函数 1：函数响应映射模板
5. 函数 2：函数请求映射模板
6. 函数 2：数据源调用
7. 函数 2：函数响应映射模板

8. 管道解析器的之后映射模板

> **Note**
>
> 管道解析器执行流是单向的，并在解析器上静态定义。

非常有用的 Apache Velocity 模板语言 (VTL) 实用程序

随着应用程序复杂性的增加，VTL 实用工具和指令在这里有助于提高开发效率。在使用管道解析器时，以下实用工具可为您提供帮助。

$ctx.stash

存储区是一个在每个解析器和函数映射模板中提供的Map。同一存储区实例通过单个解析器生效。这意味着，您可以使用存储区来跨请求和响应映射模板以及管道解析器中的函数传递任意数据。存储区公开与 Java 映射数据结构相同的方法。

$ctx.prev.result

`$ctx.prev.result` 表示在管道解析器中执行的上一个操作的结果。

如果上一个操作是管道解析器的之前映射模板，则 `$ctx.prev.result` 表示模板评估的输出，并提供给管道中的第一个函数。如果上一个操作是第一个函数，则 `$ctx.prev.result` 表示第一个函数的输出，并且可供管道中的第二个函数使用。如果上一个操作是最后一个函数，则 `$ctx.prev.result` 表示最后一个函数的输出，并提供给管道解析器的之后映射模板。

#return(data: Object)

如果您需要从任何映射模板提前返回，`#return(data: Object)` 指令会很有用。`#return(data: Object)` 类似于编程语言中的 return 关键字，因为它会从最近的逻辑范围块返回。这意味着在解析器映射模板中使用 `#return` 会从解析器返回。在解析器映射模板中使用 `#return(data: Object)` 会在 GraphQL 字段上设置 `data`。此外，从函数映射模板使用 `#return(data: Object)` 会从函数返回，并继续执行到管道中的下一个函数或解析器响应映射模板。

#return

这与 `#return(data: Object)` 相同，但返回 `null`。

$util.error

`$util.error` 实用工具对于引发字段错误很有用。在函数映射模板中使用 `$util.error` 会立即引发字段错误，从而阻止执行后续函数。有关更多详细信息和其他 `$util.error` 签名，请访问解析器映射模板实用程序参考。

$util.appendError

`$util.appendError` 类似于 `$util.error()`，主要区别在于前者不会中断映射模板的评估。相反，它指示该字段存在错误，但允许评估模板并因此会返回数据。在函数中使用 `$util.appendError` 将不会中断管道的执行流。有关更多详细信息和其他 `$util.error` 签名，请访问解析器映射模板实用程序参考。

示例 模板

假设您在名为 `getPost(id:ID!)` 的字段上具有 DynamoDB 数据源和单位解析器，该解析器使用以下 GraphQL 查询返回 `Post` 类型：

```
getPost(id:1){
    id
    title
    content
}
```

解析器模板应如下所示：

```
    {
    "version" : "2018-05-29",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

这会使用 `id` 输入参数值 1 替换 `${ctx.args.id}` 并生成以下 JSON：

```
    {
    "version" : "2018-05-29",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "1" }
```

```
    }
}
```

AWS AppSync 使用该模板生成指令，以与 DynamoDB 通信并获取数据（或执行其他相应的操作）。在返回数据后，AWS AppSync 通过可选的响应映射模板运行数据，您可以使用该模板执行数据形状设置或逻辑。例如，在我们从 DynamoDB 中获取结果时，它们可能如下所示：

```
{
        "id" : 1,
        "theTitle" : "AWS AppSync works offline!",
        "theContent-part1" : "It also has realtime functionality",
        "theContent-part2" : "using GraphQL"
}
```

您可以选择使用以下响应映射模板将两个字段联接成单一字段：

```
{
        "id" : $util.toJson($context.data.id),
        "title" : $util.toJson($context.data.theTitle),
        "content" : $util.toJson("${context.data.theContent-part1}
 ${context.data.theContent-part2}")
}
```

以下是将模板应用到数据后对设置数据形状的方式：

```
{
        "id" : 1,
        "title" : "AWS AppSync works offline!",
        "content" : "It also has realtime functionality using GraphQL"
}
```

此数据作为响应返回给客户端，如下所示：

```
{
        "data": {
                "getPost":      {
                        "id" : 1,
                        "title" : "AWS AppSync works offline!",
                        "content" : "It also has realtime functionality using GraphQL"
                }
        }
```

```
}
```

请注意，在大多数情况下，响应映射模板是简单的数据传递，主要由于您返回的是单个项目还是项目列表而不同。对于单个项目，传递：

```
$util.toJson($context.result)
```

对于列表，通常传递的是：

```
$util.toJson($context.result.items)
```

要查看单位解析器和管道解析器的更多示例，请参阅解析器教程。

评估的映射模板反序列化规则

映射模板评估结果为字符串。在 AWS AppSync 中，输出字符串必须采用 JSON 结构才有效。

此外，将强制执行以下反序列化规则。

JSON 对象中不允许使用重复的键

如果评估得出的映射模板字符串表示 JSON 对象或包含具有重复键的对象，则映射模板会返回以下错误消息：

```
Duplicate field 'aField' detected on Object. Duplicate JSON keys are not allowed.
```

已评估的请求映射模板中的重复键示例：

```
{
    "version": "2018-05-29",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "1",
        "field": "getPost" ## key 'field' has been redefined
    }
}
```

要修复此错误，请不要重新定义 JSON 对象中的键。

JSON 对象中不允许使用尾随字符

如果评估得出的映射模板字符串表示 JSON 对象并包含尾随的无关字符，则映射模板会返回以下错误消息：

```
Trailing characters at the end of the JSON string are not allowed.
```

已评估的请求映射模板中的尾随字符示例：

```
{
    "version": "2018-05-29",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "1",
    }
}extraneouschars
```

要修复该错误，请确保评估的模板的评估结果严格为 JSON。

解析器映射模板编程指南

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

此说明书风格的教程介绍如何在 AWS AppSync 中利用 Apache Velocity 模板语言 (VTL) 进行编程。如果您熟悉 JavaScript、C 或 Java 等其他编程语言，可能会比较容易理解本书内容。

AWS AppSync 使用 VTL 将来自客户端的 GraphQL 请求转换为数据源请求。然后，它将这一过程反转，将数据源响应转换回 GraphQL 响应。VTL 是一种逻辑模板语言，允许您使用以下技术处理 Web 应用程序的标准请求/响应流程中的请求和响应：

- 新项目的默认值
- 输入验证和格式化
- 转换数据和设置数据形状
- 遍历列表、映射和数组，从而提取值或更改值

- 根据用户身份筛选/更改响应
- 复杂的授权检查

例如，您可能希望在该服务中对 GraphQL 参数执行电话号码验证，或者在将输入参数存储到 DynamoDB 之前将其转换为大写。或者您可能希望客户端系统提供一个代码，作为 GraphQL 参数、JWT 令牌声明或 HTTP 标头的一部分，并且仅在该代码与列表中的特定字符串匹配时才利用数据做出响应。这些都是您可以在 AWS AppSync 中使用 VTL 执行的逻辑检查。

您可以通过 VTL 使用可能已很熟悉的编程技术应用逻辑。但是，它仅限于在标准请求/响应流程之内运行，以确保您的 GraphQL API 可随用户群的增长而扩展。由于 AWS AppSync 还支持将 AWS Lambda 作为解析器，如果您需要更大的灵活性，可以使用所选的编程语言（Node.js、Python、Go、Java 等）编写 Lambda 函数。

设置

学习一种语言时，一种常用方法是输出结果（例如 JavaScript 中的 `console.log(variable)`），看看会发生什么。在本教程中，我们将演示这一方法：创建一个简单的 GraphQL 架构，并将值的映射传递到 Lambda 函数中。Lambda 函数会输出这些值，并用这些值进行响应。这样可帮助您理解请求/响应流，并了解不同的编程技术。

首先要创建以下 GraphQL 架构：

```
type Query {
    get(id: ID, meta: String): Thing
}

type Thing {
    id: ID!
    title: String!
    meta: String
}

schema {
    query: Query
}
```

现在使用 Node.js 语言创建以下 AWS Lambda 函数：

```
exports.handler = (event, context, callback) => {
    console.log('VTL details: ', event);
```

```
    callback(null, event);
};
```

在 AWS AppSync 控制台的数据源窗格中，将该 Lambda 函数添加为新数据源。导航回控制台的架构页面，然后单击右侧的 `get(...):Thing` 查询旁边的附加按钮。对于请求模板，从 Invoke and forward arguments (调用并转发参数) 菜单中选择现有模板。对于响应模板，选择 Return Lambda result (返回 Lambda 结果)。

在一个位置中为您的 Lambda 函数打开 Amazon CloudWatch Logs，然后从 AWS AppSync 控制台的查询选项卡运行以下 GraphQL 查询：

```
query test {
  get(id:123 meta:"testing"){
    id
    meta
  }
}
```

GraphQL 响应应包含 `id:123` 和 `meta:testing`，因为 Lambda 函数将它们重复发送回来了。几秒钟后，您应该会在 CloudWatch Logs 中看到一条记录，其中包含这些详细信息。

Variables

VTL 使用引用存储或处理数据。VTL 中有三类引用：变量、属性和方法。变量前面有一个 $ 符号，由 `#set` 指令创建：

```
#set($var = "a string")
```

变量存储的类型与您熟悉的其他语言类似，例如数字、字符串、数组、列表和映射。您可能已经注意到了，在 Lambda 解析器的默认请求模板中发送了 JSON 负载：

```
"payload": $util.toJson($context.arguments)
```

此处需要注意一些事项 - 首先，AWS AppSync 提供一些便利函数以执行常见的操作。在此示例中，`$util.toJson` 将一个变量转换为 JSON。第二，变量 `$context.arguments` 作为映射对象由一个 GraphQL 请求自动填充。您可以创建新映射，方法如下：

```
#set( $myMap = {
  "id": $context.arguments.id,
  "meta": "stuff",
```

```
  "upperMeta" : $context.arguments.meta.toUpperCase()
} )
```

现在您已创建一个名为 $myMap 的变量，它的键有 id、meta 和 upperMeta。这一操作也说明了几件事：

- id 由 GraphQL 参数的键填充。这是 VTL 从客户端获取参数的常用方法。
- meta 利用一个值进行硬编码，展示默认值。
- upperMeta 使用 meta 方法转换 .toUpperCase() 参数。

将之前的代码加到请求模板的最上方，并更改 payload 以使用新的 $myMap 变量：

```
"payload": $util.toJson($myMap)
```

运行您的 Lambda 函数，您可以在 CloudWatch 日志中看到响应的变化以及此数据。当您演练此教程的其余部分时，我们将继续填充 $myMap，这样您就可以运行类似的测试。

您也可以在变量中设置 properties_。它们可以是简单的字符串、数组或 JSON：

```
#set($myMap.myProperty = "ABC")
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])
#set($myMap.jsonProperty = {
    "AppSync" : "Offline and Realtime",
    "Cognito" : "AuthN and AuthZ"
})
```

无提示引用

由于 VTL 是一种模板化的语言，默认情况下，您进行的每次引用都会执行 .toString()。如果未定义引用，它会以字符串输出实际的引用表示形式。例如：

```
#set($myValue = 5)
##Prints '5'
$myValue

##Prints '$somethingelse'
$somethingelse
```

为了应对这一问题，VTL 有一种无提示引用 或静默引用 语法，告知模板引擎禁止此行为。该语法为 $!{}。例如，如果我们稍微改动一下之前的代码，使用 $!{somethingelse}，输出就会禁止：

```
#set($myValue = 5)
##Prints '5'
$myValue

##Nothing prints out
$!{somethingelse}
```

调用方法

在之前的示例中，我们向您演示了如何在创建变量的同时设置值。您还可以通过两个步骤在映射中添加数据，实现相同的目的，如下所示：

```
#set ($myMap = {})
#set ($myList = [])

##Nothing prints out
$!{myMap.put("id", "first value")}
##Prints "first value"
$!{myMap.put("id", "another value")}
##Prints true
$!{myList.add("something")}
```

但是对于这种行为，您需要了解以下内容。虽然您可使用无提示引用表示法 `$!{}` 调用方法（如上所示），但它不会禁止所执行方法的返回值。因此在以上示例中我们会标注 `##Prints "first value"` 和 `##Prints true`。如果您要循环访问映射或列表，例如在已存在键的位置插入值，这样会引发错误。因为在评估时输出会在模板中添加意外字符串。

有时，可使用 `#set` 指令调用方法，并忽略变量来解决这一问题。例如：

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

您可以在模板中使用该技术，因为它可以防止在模板中输出意外的字符串。AWSAppSync 提供了另一种便利函数，它以更简洁的表示法提供相同的行为。使用这一函数不必考虑这些具体的实施规范。您可以通过 `$util.quiet()` 或它的别名 `$util.qr()` 使用此函数。例如：

```
#set ($myMap = {})
#set ($myList = [])

##Nothing prints out
```

```
$util.quiet($myMap.put("id", "first value"))
##Nothing prints out
$util.qr($myList.add("something"))
```

字符串

与许多编程语言一样，字符串有时可能较难处理，特别是当您希望通过变量生成字符串，情况更是如此。VTL 包含了许多处理字符串的常用功能。

假设您将数据作为字符串插入到 DynamoDB 等数据源中，但它是通过变量（如 GraphQL 参数）填充的。字符串具有双引号，要在字符串中引用变量，您只需使用 "${}"（没有 !，就像 quiet reference notation 中一样）。这与 JavaScript 中的模板文本类似：https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals

```
#set($firstname = "Jeff")
$!{myMap.put("Firstname", "${firstname}")}
```

您可以在 DynamoDB 请求模板中看到这种用法，例如，在使用来自 GraphQL 客户端的参数时的 `"author": { "S" : "${context.arguments.author}"}`，或者自动生成 ID 时的 `"id" : { "S" : "$util.autoId()"}`。这就意味着您可以引用变量，或方法的结果，在字符串内部填充数据。

您还可以使用 Java String class 的公共方法，例如提取子字符串：

```
#set($bigstring = "This is a long string, I want to pull out everything after the
 comma")
#set ($comma = $bigstring.indexOf(','))
#set ($comma = $comma +2)
#set ($substring = $bigstring.substring($comma))

$util.qr($myMap.put("substring", "${substring}"))
```

字符串联接也是一项常见的任务。您可以单独利用变量引用，或与静态值共同实现这一目的：

```
#set($s1 = "Hello")
#set($s2 = " World")

$util.qr($myMap.put("concat","$s1$s2"))
$util.qr($myMap.put("concat2","Second $s1 World"))
```

Loops

您已了解了如何创建变量及调用方法，现在可以在代码中添加一些逻辑了。与其他语言不同，VTL 只允许循环，迭代次数是提前确定的。Velocity 中没有 do..while。这一设计确保了评估过程始终会终止，并在执行 GraphQL 操作时提供了扩展边界。

使用 #foreach 可创建循环，需要您应用循环变量和可遍历对象，例如数组、列表、映射或集合。#foreach 循环的经典编程示例是遍历集合中的项目并将它们输出，因此在以下示例中，我们要把它们提取出来，并添加到映射中：

```
#set($start = 0)
#set($end = 5)
#set($range = [$start..$end])

#foreach($i in $range)
   ##$util.qr($myMap.put($i, "abc"))
   ##$util.qr($myMap.put($i, $i.toString()+"foo")) ##Concat variable with string
   $util.qr($myMap.put($i, "${i}foo"))     ##Reference a variable in a string with
 "${varname}"
#end
```

此示例展示了一些要点。第一，使用具有范围 [..] 运算符的变量，创建可遍历的对象。然后，每个项目由您可操作的 $i 变量引用。在上一示例中，您还会看到以双井号 ## 表示的注释。该示例还展示了如何在键或值中使用循环变量，以及联接字符串的不同方法。

请注意，$i 是整数，因此您可以调用 .toString() 方法。对于 GraphQL 的 INT 类型，此方法很方便。

您还可以直接使用范围运算符，例如：

```
#foreach($item in [1..5])
    ...
#end
```

数组

目前，您已经可以处理映射了，但在 VTL 中数组也很常用。您也可以通过数组访问一些底层方法，例如 .isEmpty()、.size()、.set()、.get() 和 .add()，如下所示：

```
#set($array = [])
```

```
#set($idx = 0)

##adding elements
$util.qr($array.add("element in array"))
$util.qr($myMap.put("array", $array[$idx]))

##initialize array vals on create
#set($arr2 = [42, "a string", 21, "test"])

$util.qr($myMap.put("arr2", $arr2[$idx]))
$util.qr($myMap.put("isEmpty", $array.isEmpty()))  ##isEmpty == false
$util.qr($myMap.put("size", $array.size()))

##Get and set items in an array
$util.qr($myMap.put("set", $array.set(0, 'changing array value')))
$util.qr($myMap.put("get", $array.get(0)))
```

上一示例使用数组索引表示法检索具有 arr2[$idx] 的元素。您可以从映射/字典中按名称进行查找，方法类似：

```
#set($result = {
    "Author" : "Nadia",
    "Topic" : "GraphQL"
})

$util.qr($myMap.put("Author", $result["Author"]))
```

如果使用条件在响应模板中筛选数据源中的结果，这种方法非常常用。

条件检查

之前介绍 #foreach 的部分展示了一些示例，说明了如何利用 VTL 使用逻辑转换数据。您也可以应用条件检查以在运行时评估数据：

```
#if(!$array.isEmpty())
    $util.qr($myMap.put("ifCheck", "Array not empty"))
#else
    $util.qr($myMap.put("ifCheck", "Your array is empty"))
#end
```

以上对布尔表达式进行 #if() 检查的示例很棒，但您也可以将运算符和 #elseif() 用于分支：

```
#if ($arr2.size() == 0)
    $util.qr($myMap.put("elseIfCheck", "You forgot to put anything into this array!"))
#elseif ($arr2.size() == 1)
    $util.qr($myMap.put("elseIfCheck", "Good start but please add more stuff"))
#else
    $util.qr($myMap.put("elseIfCheck", "Good job!"))
#end
```

这两个示例展示了否定 (!) 和相等 (==)。我们还可以使用 ||、&&、>、<、>=、<= 和 !=。

```
#set($T = true)
#set($F = false)

#if ($T || $F)
  $util.qr($myMap.put("OR", "TRUE"))
#end

#if ($T && $F)
  $util.qr($myMap.put("AND", "TRUE"))
#end
```

注意：在条件中只有 `Boolean.FALSE` 和 `null` 被视为 false。零 (0) 和空字符串 ("") 并不等同于 false。

运算符

如果没有一些执行数学运算的运算符，那么编程语言就不完整。以下是一些入门示例：

```
#set($x = 5)
#set($y = 7)
#set($z = $x + $y)
#set($x-y = $x - $y)
#set($xy = $x * $y)
#set($xDIVy = $x / $y)
#set($xMODy = $x % $y)

$util.qr($myMap.put("z", $z))
$util.qr($myMap.put("x-y", $x-y))
$util.qr($myMap.put("x*y", $xy))
$util.qr($myMap.put("x/y", $xDIVy))
$util.qr($myMap.put("x|y", $xMODy))
```

共用循环和条件

在 VTL 中转换数据时，这非常常用。例如在从数据源进行读或写的操作之前遍历对象，然后在执行操作之前进行检查。将之前的各部分中介绍的工具结合起来，您可以获得许多功能。一种非常好用的工具是，#foreach 会自动为每个项目提供 .count：

```
#foreach ($item in $arr2)
  #set($idx = "item" + $foreach.count)
  $util.qr($myMap.put($idx, $item))
#end
```

例如，可能您希望将不超过某一大小的映射中的值提取出来。结合使用计数、条件和 #break 语句即可实现这一目的：

```
#set($hashmap = {
  "DynamoDB" : "https://aws.amazon.com/dynamodb/",
  "Amplify" : "https://github.com/aws/aws-amplify",
  "DynamoDB2" : "https://aws.amazon.com/dynamodb/",
  "Amplify2" : "https://github.com/aws/aws-amplify"
})

#foreach ($key in $hashmap.keySet())
    #if($foreach.count > 2)
    #break
  #end
    $util.qr($myMap.put($key, $hashmap.get($key)))
#end
```

上一 #foreach 利用 .keySet() 进行遍历，您可将它用于映射。这样您就有权限获取 $key，并通过 .get($key) 引用值。在 AWS AppSync 中，来自客户端的 GraphQL 参数存储为映射。也可以通过 .entrySet() 循环访问这些参数，可通过它将键和值作为集进行访问，从而填充其他变量或执行复杂的条件检查，例如验证或转换输入：

```
#foreach( $entry in $context.arguments.entrySet() )
#if ($entry.key == "XYZ" && $entry.value == "BAD")
    #set($myvar = "...")
  #else
    #break
  #end
#end
```

其他常见的示例自动填充默认信息，例如，同步数据时的初始对象版本（在解决冲突时非常重要）或用于授权检查的对象的默认所有者 - Mary 创建了该博客文章，因此，代码为：

```
#set($myMap.owner ="Mary")
#set($myMap.defaultOwners = ["Admins", "Editors"])
```

上下文

您现在充分了解了如何在 AWS AppSync 解析器中使用 VTL 执行逻辑检查，让我们了解一下上下文对象：

```
$util.qr($myMap.put("context", $context))
```

其中包含您可以在 GraphQL 请求中访问的所有信息。有关详细解释，请参阅上下文参考。

过滤

在此教程中，目前来自 Lambda 函数的所有信息已返回 GraphQL 查询，并进行了非常简单的 JSON 转换：

```
$util.toJson($context.result)
```

如果您要从数据源获得响应，VTL 逻辑也同样强大，特别是对资源进行授权检查时更是如此。让我们演练一些示例。首先，尝试按如下方式更改响应模板：

```
#set($data = {
    "id" : "456",
    "meta" : "Valid Response"
})

$util.toJson($data)
```

无论您的 GraphQL 操作结果如何，硬编码值将返回客户端。把它稍做调整，用 Lambda 响应填充 `meta` 字段，我们在本教程前面学习条件时在 `elseIfCheck` 值中进行了相关设置：

```
#set($data = {
    "id" : "456"
})

#foreach($item in $context.result.entrySet())
```

```
    #if($item.key == "elseIfCheck")
        $util.qr($data.put("meta", $item.value))
    #end
#end

$util.toJson($data)
```

$context.result 是映射，因此您可以使用 entrySet() 针对返回的键或值执行逻辑。由于 $context.identity 包含执行 GraphQL 操作的用户的相关信息，如果您从数据源返回授权信息，那么您可以根据逻辑决定为用户返回全部、部分数据，或不返回数据。更改您的响应模板，使它与如下示例类似：

```
#if($context.result["id"] == 123)
    $util.toJson($context.result)
  #else
    $util.unauthorized()
#end
```

如果您运行 GraphQL 查询，数据将按正常情况返回。但如果您将 id 参数更改为 123 之外的值 (query test { get(id:456 meta:"badrequest"){} })，将收到授权失败的消息。

您可以在授权使用案例部分找到有关授权场景的更多示例。

附录 - 模板示例

如果您按照此教程的步骤执行，那么可能已逐步构建了此模板。如果您还没有这么做，我们将在下面提供模板，供您复制用于测试。

请求模板

```
#set( $myMap = {
  "id": $context.arguments.id,
  "meta": "stuff",
  "upperMeta" : "$context.arguments.meta.toUpperCase()"
} )

##This is how you would do it in two steps with a "quiet reference" and you can use it
 for invoking methods, such as .put() to add items to a Map
#set ($myMap2 = {})
$util.qr($myMap2.put("id", "first value"))

## Properties are created with a dot notation
```

```
#set($myMap.myProperty = "ABC")
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])
#set($myMap.jsonProperty = {
    "AppSync" : "Offline and Realtime",
    "Cognito" : "AuthN and AuthZ"
})

##When you are inside a string and just have ${} without ! it means stuff inside curly
 braces are a reference
#set($firstname = "Jeff")
$util.qr($myMap.put("Firstname", "${firstname}"))

#set($bigstring = "This is a long string, I want to pull out everything after the
 comma")
#set ($comma = $bigstring.indexOf(','))
#set ($comma = $comma +2)
#set ($substring = $bigstring.substring($comma))
$util.qr($myMap.put("substring", "${substring}"))

##Classic for-each loop over N items:
#set($start = 0)
#set($end = 5)
#set($range = [$start..$end])
#foreach($i in $range)          ##Can also use range operator directly like
 #foreach($item in [1...5])
   ##$util.qr($myMap.put($i, "abc"))
   ##$util.qr($myMap.put($i, $i.toString()+"foo")) ##Concat variable with string
   $util.qr($myMap.put($i, "${i}foo"))     ##Reference a variable in a string with
 "${varname)"
#end

##Operators don't work
#set($x = 5)
#set($y = 7)
#set($z = $x + $y)
#set($x-y = $x - $y)
#set($xy = $x * $y)
#set($xDIVy = $x / $y)
#set($xMODy = $x % $y)
$util.qr($myMap.put("z", $z))
$util.qr($myMap.put("x-y", $x-y))
$util.qr($myMap.put("x*y", $xy))
$util.qr($myMap.put("x/y", $xDIVy))
$util.qr($myMap.put("x|y", $xMODy))
```

```
##arrays
#set($array = ["first"])
#set($idx = 0)
$util.qr($myMap.put("array", $array[$idx]))
##initialize array vals on create
#set($arr2 = [42, "a string", 21, "test"])
$util.qr($myMap.put("arr2", $arr2[$idx]))
$util.qr($myMap.put("isEmpty", $array.isEmpty()))  ##Returns false
$util.qr($myMap.put("size", $array.size()))
##Get and set items in an array
$util.qr($myMap.put("set", $array.set(0, 'changing array value')))
$util.qr($myMap.put("get", $array.get(0)))

##Lookup by name from a Map/dictionary in a similar way:
#set($result = {
    "Author" : "Nadia",
    "Topic" : "GraphQL"
})
$util.qr($myMap.put("Author", $result["Author"]))


##Conditional examples
#if(!$array.isEmpty())
$util.qr($myMap.put("ifCheck", "Array not empty"))
#else
$util.qr($myMap.put("ifCheck", "Your array is empty"))
#end

#if ($arr2.size() == 0)
$util.qr($myMap.put("elseIfCheck", "You forgot to put anything into this array!"))
#elseif ($arr2.size() == 1)
$util.qr($myMap.put("elseIfCheck", "Good start but please add more stuff"))
#else
$util.qr($myMap.put("elseIfCheck", "Good job!"))
#end

##Above showed negation(!) and equality (==), we can also use OR, AND, >, <, >=, <=,
 and !=
#set($T = true)
#set($F = false)
#if ($T || $F)
  $util.qr($myMap.put("OR", "TRUE"))
#end
```

```
#if ($T && $F)
  $util.qr($myMap.put("AND", "TRUE"))
#end

##Using the foreach loop counter - $foreach.count
#foreach ($item in $arr2)
  #set($idx = "item" + $foreach.count)
  $util.qr($myMap.put($idx, $item))
#end

##Using a Map and plucking out keys/vals
#set($hashmap = {
    "DynamoDB" : "https://aws.amazon.com/dynamodb/",
    "Amplify" : "https://github.com/aws/aws-amplify",
    "DynamoDB2" : "https://aws.amazon.com/dynamodb/",
    "Amplify2" : "https://github.com/aws/aws-amplify"
})

#foreach ($key in $hashmap.keySet())
    #if($foreach.count > 2)
        #break
    #end
    $util.qr($myMap.put($key, $hashmap.get($key)))
#end

##concatenate strings
#set($s1 = "Hello")
#set($s2 = " World")
$util.qr($myMap.put("concat","$s1$s2"))
$util.qr($myMap.put("concat2","Second $s1 World"))

$util.qr($myMap.put("context", $context))

{
    "version" : "2017-02-28",
    "operation": "Invoke",
    "payload": $util.toJson($myMap)
}
```

响应模板

```
#set($data = {
```

```
"id" : "456"
})
#foreach($item in $context.result.entrySet())   ##$context.result is a MAP so we use
 entrySet()
    #if($item.key == "ifCheck")
        $util.qr($data.put("meta", "$item.value"))
    #end
#end

##Uncomment this out if you want to test and remove the below #if check
##$util.toJson($data)

#if($context.result["id"] == 123)
    $util.toJson($context.result)
  #else
    $util.unauthorized()
#end
```

解析器映射模板上下文参考

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

AWS AppSync 定义了一组用于处理解析器映射模板的变量和函数。这样，就可以更轻松地通过 GraphQL 对数据进行逻辑操作。本文档将介绍这些函数，并提供使用模板的示例。

使用 `$context`

`$context` 变量是一个映射，它保留进行解析器调用的所有上下文信息。它具有以下结构：

```
{
   "arguments" : { ... },
   "source" : { ... },
   "result" : { ... },
   "identity" : { ... },
   "request" : { ... },
   "info": { ... }
}
```

> **Note**
>
> 如果您尝试按字典/映射条目（例如 context 中的条目）键访问该条目以检索值，Velocity 模板语言 (VTL) 允许您直接使用 <dictionary-element>.<key-name> 表示法。但是，这可能不适用于所有情况，例如当键名称具有特殊字符时（例如，下划线“_”）。建议您始终使用 <dictionary-element>.get("<key-name>") 表示法。

$context 映射中每个字段的定义如下所示：

$context 字段

arguments

包含该字段的所有 GraphQL 参数的映射。

identity

包含有关调用方的信息的对象。有关该字段结构的更多信息，请参阅身份。

source

包含父字段解析的映射。

stash

存储区是一个在每个解析器和函数映射模板中提供的映射。同一存储区实例通过单个解析器生效。这意味着，您可以使用存储区来跨请求和响应映射模板以及管道解析器中的函数传递任意数据。存储区公开与 Java 映射数据结构相同的方法。

result

此解析器结果的容器。该字段仅适用于响应映射模板。

例如，如果要解析以下查询的 author 字段：

```
query {
    getPost(id: 1234) {
        postId
        title
        content
        author {
```

```
            id
            name
        }
    }
}
```

那么在处理响应映射模板时，完整的 $context 变量可以是：

```
{
  "arguments" : {
    id: "1234"
  },
  "source": {},
  "result" : {
      "postId": "1234",
      "title": "Some title",
      "content": "Some content",
      "author": {
        "id": "5678",
        "name": "Author Name"
      }
  },
  "identity" : {
      "sourceIp" : ["x.x.x.x"],
      "userArn" : "arn:aws:iam::123456789012:user/appsync",
      "accountId" : "666666666666",
      "user" : "AIDAAAAAAAAAAAAAAAAA"
  }
}
```

prev.result

在管道解析器中执行的任何以前操作的结果。

如果上一个操作是管道解析器的之前映射模板，则 $ctx.prev.result 表示模板评估的输出，并提供给管道中的第一个函数。

如果上一个操作是第一个函数，则 $ctx.prev.result 表示第一个函数的输出，并且可供管道中的第二个函数使用。

如果上一个操作是最后一个函数，则 $ctx.prev.result 表示最后一个函数的输出，并提供给管道解析器的之后映射模板。

info

包含有关 GraphQL 请求的信息的对象。有关该字段的结构，请参阅信息。

求同

identity 部分包含调用方的相关信息。此部分的形态取决于您的 AWS AppSync API 的授权类型。

有关 AWS AppSync 安全选项的更多信息，请参阅授权和身份验证。

API_KEY 授权

不填充 identity 字段。

AWS_LAMBDA 授权

identity 包含 resolverContext 密钥，其中包含为请求授权的 Lambda 函数返回的相同 resolverContext 内容。

AWS_IAM 授权

identity 采用以下格式：

```
{
    "accountId" : "string",
    "cognitoIdentityPoolId" : "string",
    "cognitoIdentityId" : "string",
    "sourceIp" : ["string"],
    "username" : "string", // IAM user principal
    "userArn" : "string",
    "cognitoIdentityAuthType" : "string", // authenticated/unauthenticated based on
 the identity type
    "cognitoIdentityAuthProvider" : "string" // the auth provider that was used to
 obtain the credentials
}
```

AMAZON_COGNITO_USER_POOLS 授权

identity 采用以下格式：

```
{
    "sub" : "uuid",
    "issuer" : "string",
    "username" : "string"
```

```
    "claims" : { ... },
    "sourceIp" : ["x.x.x.x"],
    "defaultAuthStrategy" : "string"
}
```

每个字段的定义如下所示：

accountId

调用方的 AWS 账户 ID。

claims

用户拥有的声明。

cognitoIdentityAuthType

根据身份类型确定经过身份验证或未经身份验证。

cognitoIdentityAuthProvider

外部身份提供程序信息的逗号分隔列表，用于获取对请求进行签名时使用的凭证。

cognitoIdentityId

调用方的 Amazon Cognito 身份 ID。

cognitoIdentityPoolId

与调用方关联的 Amazon Cognito 身份池 ID。

defaultAuthStrategy

此调用方的默认授权策略（ALLOW 或 DENY）。

issuer

令牌发布者。

sourceIp

AWS AppSync 收到的调用方的源 IP 地址。如果请求不包含 `x-forwarded-for` 标头，则源 IP 值仅包含来自 TCP 连接的单个 IP 地址。如果请求中包含 `x-forwarded-for` 标头，那么源 IP 将是来自 `x-forwarded-for` 标头的 IP 地址列表，以及来自 TCP 连接的 IP 地址。

sub

经过验证的用户的 UUID。

user

IAM 用户。

userArn

IAM 用户的 Amazon 资源名称 (ARN)。

username

已验证的用户的用户名。对于 AMAZON_COGNITO_USER_POOLS 授权，用户名的值是 cognito:username 属性的值。对于 AWS_IAM 授权，username 值是 AWS 用户主体的值。如果您将 IAM 授权与从 Amazon Cognito 身份池提供的凭证一起使用，我们建议您使用 cognitoIdentityId。

访问请求标头

AWS AppSync 支持从客户端传递自定义标头，并使用 $context.request.headers 在 GraphQL 解析器中访问这些标头。然后，您可以使用标头值执行操作，例如，将数据插入到数据源或进行授权检查。您可以在命令行中使用 $curl 将一个或多个请求标头与 API 密钥一起使用，如以下示例中所示：

单标头示例

假设您设置一个 custom 标头，值为 nadia，如下所示：

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:nadia" -H "x-api-key:<API-
KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo\", when: \"Next Friday!
\", where: \"Here!\") {id name when where description}}"}' https://<ENDPOINT>/graphql
```

然后可使用 $context.request.headers.custom 访问它。例如，对于 DynamoDB，它可能位于以下 VTL 中：

```
"custom": $util.dynamodb.toDynamoDBJson($context.request.headers.custom)
```

多标头示例

您还可以在一个请求中传递多个标头，并在解析器映射模板中访问它们。例如，如果为 custom 标头设置了两个值：

```
curl -XPOST -H "Content-Type:application/graphql" -H "custom:bailey" -H "custom:nadia"
 -H "x-api-key:<API-KEY-VALUE>" -d '{"query":"mutation { createEvent(name: \"demo
```

```
\", when: \"Next Friday!\", where: \"Here!\") {id name when where description}}"}'
 https://<ENDPOINT>/graphql
```

它们可以作为一个数组访问，例如 `$context.request.headers.custom[1]`。

> **Note**
>
> AWS AppSync 不会在 `$context.request.headers` 中公开 Cookie 标头。

访问请求自定义域名

AWS AppSync 支持配置自定义域，您可以使用该域访问 API 的 GraphQL 和实时终端节点。在使用自定义域名发出请求时，您可以使用 `$context.request.domainName` 获取域名。

在使用默认 GraphQL 终端节点域名时，值为 `null`。

Info

`info` 部分包含有关 GraphQL 请求的信息。该部分采用以下格式：

```
{
    "fieldName": "string",
    "parentTypeName": "string",
    "variables": { ... },
    "selectionSetList": ["string"],
    "selectionSetGraphQL": "string"
}
```

每个字段的定义如下所示：

`fieldName`

当前正在解析的字段的名称。

`parentTypeName`

当前正在解析的字段的父类型的名称。

`variables`

保留传递到 GraphQL 请求的所有变量的映射。

selectionSetList

GraphQL 选择集中字段的列表表示形式。具有别名的字段仅按别名进行引用，而不按字段名称进行引用。以下示例对此详细进行了介绍。

selectionSetGraphQL

选择集的字符串表示形式，格式为 GraphQL 架构定义语言 (SDL)。尽管片段不会合并到选择集中，但会保留内联片段，如以下示例中所示。

> **Note**
>
> 默认情况下，在 `context.info` 上使用 `$utils.toJson()` 时，不会序列化 `selectionSetGraphQL` 和 `selectionSetList` 返回的值。

例如，如果您要解析以下查询的 `getPost` 字段：

```
query {
  getPost(id: $postId) {
    postId
    title
    secondTitle: title
    content
    author(id: $authorId) {
      authorId
      name
    }
    secondAuthor(id: "789") {
      authorId
    }
    ... on Post {
      inlineFrag: comments: {
        id
      }
    }
    ... postFrag
  }
}

fragment postFrag on Post {
  postFrag: comments: {
```

```
    id
  }
}
```

那么，在处理映射模板时可用的完整 $context.info 变量可以是：

```
{
  "fieldName": "getPost",
  "parentTypeName": "Query",
  "variables": {
    "postId": "123",
    "authorId": "456"
  },
  "selectionSetList": [
    "postId",
    "title",
    "secondTitle"
    "content",
    "author",
    "author/authorId",
    "author/name",
    "secondAuthor",
    "secondAuthor/authorId",
    "inlineFragComments",
    "inlineFragComments/id",
    "postFragComments",
    "postFragComments/id"
  ],
  "selectionSetGraphQL": "{\n  getPost(id: $postId) {\n    postId\n    title\n
 secondTitle: title\n    content\n    author(id: $authorId) {\n      authorId\n
 name\n    }\n    secondAuthor(id: \"789\") {\n      authorId\n    }\n    ... on Post
 {\n      inlineFrag: comments {\n        id\n      }\n    }\n    ... postFrag\n  }\n}"
}
```

selectionSetList 仅公开属于当前类型的字段。如果当前类型是接口或联合，则仅公开属于该接口的选定字段。例如，给定以下架构：

```
type Query {
    node(id: ID!): Node
}

interface Node {
    id: ID
```

```
}

type Post implements Node {
    id: ID
    title: String
    author: String
}

type Blog implements Node {
    id: ID
    title: String
    category: String
}
```

以及以下查询：

```
query {
    node(id: "post1") {
        id
        ... on Post {
            title
        }

        ... on Blog {
            title
        }
    }
}
```

如果在进行 Query.node 字段解析时调用 $ctx.info.selectionSetList，则仅公开 id：

```
"selectionSetList": [
    "id"
]
```

清理输入

应用程序必须对不可信的输入进行清理，以防止任何外部方在应用程序的预期用途之外使用该应用程序。由于 $context 包含 $context.arguments、$context.identity、$context.result、$context.info.variables

和 $context.request.headers 等属性中的用户输入，因此，务必谨慎在映射模板中清理它们的值。

由于映射模板代表 JSON，因此输入清理采用从表示用户输入的字符串转义 JSON 保留字符的形式。将 JSON 保留字符放入映射模板时，最佳做法是使用 $util.toJson() 实用程序从敏感字符串值转义 JSON 保留字符。

例如，在以下 Lambda 请求映射模板中，由于我们访问了不安全的客户输入字符串 ($context.arguments.id)，因此，我们使用 $util.toJson() 将其包装起来，以防止未转义的 JSON 字符破坏 JSON 模板。

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": $util.toJson($context.arguments.id)
    }
}
```

下面的映射模板与之相反，我们直接插入 $context.arguments.id 而不进行清理。这不适用于包含未转义引号或其他 JSON 保留字符的字符串，并且可能使您的模板很容易失败。

```
## DO NOT DO THIS
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "$context.arguments.id" ## Unsafe! Do not insert $context string
 values without escaping JSON characters.
    }
}
```

解析器映射模板实用程序参考

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

AWS AppSync定义了一组实用程序，您可以在 GraphQL 解析器中使用这些实用程序来简化与数据源的交互。其中的一些实用程序可以广泛用于任何数据来源，例如生成 ID 或时间戳。其他实用程序是某种类型的数据来源特定的。

主题

$util 中的实用程序帮助程序

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

`$util` 变量包含帮助您处理数据的常规实用程序方法。除非另行指定，否则所有实用程序均使用 UTF-8 字符集。

JSON 解析实用程序

JSON 解析实用程序列表

```
$util.parseJson(String) : Object
```

获取“字符串化的”JSON 并返回结果的对象表示形式。

```
$util.toJson(Object) : String
```

获取对象并返回该对象“字符串化的”JSON 表示形式。

编码实用程序

编码实用程序列表

```
$util.urlEncode(String) : String
```

将输入字符串作为 application/x-www-form-urlencoded 编码字符串返回。

```
$util.urlDecode(String) : String
```

将 application/x-www-form-urlencoded 编码的字符串解码回未编码的形式。

```
$util.base64Encode( byte[] ) : String
```

将输入编码为 base64 编码字符串。

```
$util.base64Decode(String) : byte[]
```

对 base64 编码字符串中的数据进行解码。

ID 生成实用程序

ID 生成实用程序列表

```
$util.autoId() : String
```

返回 128 位随机生成的 UUID。

```
$util.autoUlid() : String
```

返回一个 128 位随机生成的 ULID（可按字典排序的通用唯一标识符）。

`$util.autoKsuid() : String`

返回一个 128 位随机生成的 KSUID（K 可排序唯一标识符），它使用 Base62 编码为长度为 27 的字符串。

错误实用程序

错误实用程序列表

`$util.error(String)`

引发自定义错误。可以在请求或响应映射模板中使用该实用程序，以检测请求或调用结果的错误。

`$util.error(String, String)`

引发自定义错误。可以在请求或响应映射模板中使用该实用程序，以检测请求或调用结果的错误。您还可以指定 `errorType`。

`$util.error(String, String, Object)`

引发自定义错误。可以在请求或响应映射模板中使用该实用程序，以检测请求或调用结果的错误。您还可以指定 `errorType` 和 `data` 字段。将在 GraphQL 响应中 `data` 内部对应的 `error` 块中添加 `errors` 值。

> **Note**
>
> `data` 将根据查询选择集进行筛选。

`$util.error(String, String, Object, Object)`

引发自定义错误。如果模板检测到请求或调用结果的错误，可用于请求或响应映射模板中。此外，还可以指定 `errorType`、`data` 和 `errorInfo` 字段。将在 GraphQL 响应中 `data` 内部对应的 `error` 块中添加 `errors` 值。

> **Note**
>
> `data` 将根据查询选择集进行筛选。将在 GraphQL 响应中 `errorInfo` 内部对应的 `error` 块中添加 `errors` 值。
> `errorInfo` 不会根据查询选择集进行筛选。

`$util.appendError(String)`

追加自定义错误。如果模板检测到请求或调用结果的错误，可用于请求或响应映射模板中。与 `$util.error(String)` 不同，不会中断模板评估，因此，可以向调用方返回数据。

`$util.appendError(String, String)`

追加自定义错误。如果模板检测到请求或调用结果的错误，可用于请求或响应映射模板中。此外，还可指定 `errorType`。与 `$util.error(String, String)` 不同，不会中断模板评估，因此，可以向调用方返回数据。

`$util.appendError(String, String, Object)`

追加自定义错误。如果模板检测到请求或调用结果的错误，可用于请求或响应映射模板中。此外，还可指定 `errorType` 和 `data` 字段。与 `$util.error(String, String, Object)` 不同，不会中断模板评估，因此，可以向调用方返回数据。将在 GraphQL 响应中 `data` 内部对应的 `error` 块中添加 `errors` 值。

> **Note**
>
> `data` 将根据查询选择集进行筛选。

`$util.appendError(String, String, Object, Object)`

追加自定义错误。如果模板检测到请求或调用结果的错误，可用于请求或响应映射模板中。此外，还可以指定 `errorType`、`data` 和 `errorInfo` 字段。与 `$util.error(String, String, Object, Object)` 不同，不会中断模板评估，因此，可以向调用方返回数据。将在 GraphQL 响应中 `data` 内部对应的 `error` 块中添加 `errors` 值。

> **Note**
>
> `data` 将根据查询选择集进行筛选。将在 GraphQL 响应中 `errorInfo` 内部对应的 `error` 块中添加 `errors` 值。
> `errorInfo` 不会根据查询选择集进行筛选。

条件验证实用程序

条件验证实用程序列表

```
$util.validate(Boolean, String) : void
```

如果条件为假，则 CustomTemplateException 使用指定的消息抛出 a。

```
$util.validate(Boolean, String, String) : void
```

如果条件为假，则 CustomTemplateException 使用指定的消息和错误类型抛出。

```
$util.validate(Boolean, String, String, Object) : void
```

如果条件为 false，则抛出 a， CustomTemplateException 其中包含指定的消息和错误类型，以及要在响应中返回的数据。

Null 行为实用程序

Null 行为实用程序列表

```
$util.isNull(Object) : Boolean
```

如果提供的对象为 null 则返回 true。

```
$util.isNullOrEmpty(String) : Boolean
```

如果提供的数据为 null 或空字符串，则返回 true。否则返回 false。

```
$util.isNullOrBlank(String) : Boolean
```

如果提供的数据为 null 或空白字符串，则返回 true。否则返回 false。

```
$util.defaultIfNull(Object, Object) : Object
```

如果首个对象非 null，则返回它。否则返回第二个对象，作为“默认对象”。

```
$util.defaultIfNullOrEmpty(String, String) : String
```

如果首个字符串非 null 也非空，则返回它。否则返回第二个字符串，作为“默认字符串”。

```
$util.defaultIfNullOrBlank(String, String) : String
```

如果首个字符串非 null 也非空白，则返回它。否则返回第二个字符串，作为“默认字符串”。

模式匹配实用程序

类型和模式匹配实用程序列表

```
$util.typeOf(Object) : String
```

返回字符串，描述对象的类型。支持的类型标识为："Null"、"Number"、"String"、"Map"、"List"、"Boolean"。如果无法识别类型，则返回 "Object" 类型。

```
$util.matches(String, String) : Boolean
```

如果在第一个参数中指定的模式与第二个参数中提供的数据匹配，则返回 true。模式必须为正则表达式，例如 `$util.matches("a*b", "aaaaab")`。此功能以模式为基础，您可参考其他文档，进一步了解此内容。

```
$util.authType() : String
```

返回描述请求使用的多重身份验证类型的字符串，即，返回“IAM Authorization”、“User Pool Authorization”、“Open ID Connect Authorization”或“API Key Authorization”。

对象验证实用程序

对象验证实用程序列表

```
$util.isString(Object) : Boolean
```

如果对象是字符串，则返回 true。

```
$util.isNumber(Object) : Boolean
```

如果对象是数字，则返回 true。

```
$util.isBoolean(Object) : Boolean
```

如果对象是布尔值，则返回 true。

```
$util.isList(Object) : Boolean
```

如果对象是列表，则返回 true。

```
$util.isMap(Object) : Boolean
```

如果对象是映射，则返回 true。

CloudWatch 日志工具

CloudWatch 日志实用程序列表

`$util.log.info(Object) : Void`

在 API 上启用请求级和字段级日志记录时，将所提供对象的字符串表示形式 CloudWatch 记录到请求的日志流中。ALL

`$util.log.info(String, Object...) : Void`

在 API 上启用请求级和字段级日志记录时，将所提供对象的字符串表示形式 CloudWatch 记录到请求的日志流中。ALL该实用程序按顺序将第一个输入格式字符串中由“{}”指示的所有变量替换为提供的对象的字符串表示形式。

`$util.log.error(Object) : Void`

在 API 上使用日志级别ERROR或日志级别启用字段级日志 CloudWatch 记录时，将所提供对象的字符串表示形式记录到请求的日志流中。ALL

`$util.log.error(String, Object...) : Void`

在 API 上使用日志级别ERROR或日志级别启用字段级日志 CloudWatch 记录时，将所提供对象的字符串表示形式记录到请求的日志流中。ALL该实用程序按顺序将第一个输入格式字符串中由“{}”指示的所有变量替换为提供的对象的字符串表示形式。

返回值行为实用程序

返回值行为实用程序列表

`$util.qr()` 和 `$util.quiet()`

运行 VTL 语句，同时禁止返回值。这对于在不使用临时占位符的情况下运行方法非常有用，例如，在映射中添加项目。例如：

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

变为：

```
#set ($myMap = {})
$util.qr($myMap.put("id", "first value"))
```

$util.escapeJavaScript(String) : String

将输入字符串作为 JavaScript 转义字符串返回。

$util.urlEncode(String) : String

将输入字符串作为 application/x-www-form-urlencoded 编码字符串返回。

$util.urlDecode(String) : String

将 application/x-www-form-urlencoded 编码的字符串解码回未编码的形式。

$util.base64Encode(byte[]) : String

将输入编码为 base64 编码字符串。

$util.base64Decode(String) : byte[]

对 base64 编码字符串中的数据进行解码。

$util.parseJson(String) : Object

获取“字符串化的”JSON 并返回结果的对象表示形式。

$util.toJson(Object) : String

获取对象并返回该对象“字符串化的”JSON 表示形式。

$util.autoId() : String

返回 128 位随机生成的 UUID。

$util.autoUlid() : String

返回一个 128 位随机生成的 ULID（可按字典排序的通用唯一标识符）。

$util.autoKsuid() : String

返回一个 128 位随机生成的 KSUID（K 可排序唯一标识符），它使用 Base62 编码为长度为 27 的字符串。

$util.unauthorized()

针对被解析的字段引发 Unauthorized。可以在请求或响应映射模板中使用该实用程序，以确定是否允许调用方解析该字段。

$util.error(String)

引发自定义错误。可以在请求或响应映射模板中使用该实用程序，以检测请求或调用结果的错误。

$util.error(String, String)

引发自定义错误。可以在请求或响应映射模板中使用该实用程序，以检测请求或调用结果的错误。您还可以指定 errorType。

$util.error(String, String, Object)

引发自定义错误。可以在请求或响应映射模板中使用该实用程序，以检测请求或调用结果的错误。您还可以指定 errorType 和 data 字段。将在 GraphQL 响应中 data 内部对应的 error 块中添加 errors 值。注意：将根据查询选择集筛选 data。

$util.error(String, String, Object, Object)

引发自定义错误。如果模板检测到请求或调用结果的错误，可用于请求或响应映射模板中。此外，还可指定 errorType、data 和 errorInfo 字段。将在 GraphQL 响应中 data 内部对应的 error 块中添加 errors 值。注意：将根据查询选择集筛选 data。将在 GraphQL 响应中 errorInfo 内部对应的 error 块中添加 errors 值。注意：errorInfo不会根据查询选择集筛选 。

$util.appendError(String)

追加自定义错误。如果模板检测到请求或调用结果的错误，可用于请求或响应映射模板中。与 $util.error(String) 不同，不会中断模板评估，因此，可以向调用方返回数据。

$util.appendError(String, String)

追加自定义错误。如果模板检测到请求或调用结果的错误，可用于请求或响应映射模板中。此外，还可指定 errorType。与 $util.error(String, String) 不同，不会中断模板评估，因此，可以向调用方返回数据。

$util.appendError(String, String, Object)

追加自定义错误。如果模板检测到请求或调用结果的错误，可用于请求或响应映射模板中。此外，还可指定 errorType 和 data 字段。与 $util.error(String, String, Object) 不同，不会中断模板评估，因此，可以向调用方返回数据。将在 GraphQL 响应中 data 内部对应的 error 块中添加 errors 值。注意：将根据查询选择集筛选 data。

$util.appendError(String, String, Object, Object)

追加自定义错误。如果模板检测到请求或调用结果的错误，可用于请求或响应映射模板中。此外，还可指定 errorType、data 和 errorInfo 字段。与 $util.error(String, String, Object, Object) 不同，不会中断模板评估，因此，可以向调用方返回数据。将在 GraphQL 响应中 data 内部对应的 error 块中添加 errors 值。注意：将根据查询选择集

筛选 data。将在 GraphQL 响应中 errorInfo 内部对应的 error 块中添加 errors 值。注意：errorInfo不会根据查询选择集筛选 。

$util.validate(Boolean, String) : void

如果条件为假，则 CustomTemplateException 使用指定的消息抛出 a。

$util.validate(Boolean, String, String) : void

如果条件为假，则 CustomTemplateException 使用指定的消息和错误类型抛出。

$util.validate(Boolean, String, String, Object) : void

如果条件为 false，则抛出 a， CustomTemplateException 其中包含指定的消息和错误类型，以及要在响应中返回的数据。

$util.isNull(Object) : Boolean

如果提供的对象为 null 则返回 true。

$util.isNullOrEmpty(String) : Boolean

如果提供的数据为 null 或空字符串，则返回 true。否则返回 false。

$util.isNullOrBlank(String) : Boolean

如果提供的数据为 null 或空白字符串，则返回 true。否则返回 false。

$util.defaultIfNull(Object, Object) : Object

如果首个对象非 null，则返回它。否则返回第二个对象，作为“默认对象”。

$util.defaultIfNullOrEmpty(String, String) : String

如果首个字符串非 null 也非空，则返回它。否则返回第二个字符串，作为“默认字符串”。

$util.defaultIfNullOrBlank(String, String) : String

如果首个字符串非 null 也非空白，则返回它。否则返回第二个字符串，作为“默认字符串”。

$util.isString(Object) : Boolean

如果对象是字符串，则返回 true。

$util.isNumber(Object) : Boolean

如果对象是数字，则返回 true。

$util.isBoolean(Object) : Boolean

如果对象是布尔值，则返回 true。

$util.isList(Object) : Boolean

如果对象是列表，则返回 true。

$util.isMap(Object) : Boolean

如果对象是映射，则返回 true。

$util.typeOf(Object) : String

返回字符串，描述对象的类型。支持的类型标识为："Null"、"Number"、"String"、"Map"、"List"、"Boolean"。如果无法识别类型，则返回 "Object" 类型。

$util.matches(String, String) : Boolean

如果在第一个参数中指定的模式与第二个参数中提供的数据匹配，则返回 true。模式必须为正则表达式，例如 `$util.matches("a*b", "aaaaab")`。此功能以模式为基础，您可参考其他文档，进一步了解此内容。

$util.authType() : String

返回描述请求使用的多重身份验证类型的字符串，即，返回“IAM Authorization”、“User Pool Authorization”、“Open ID Connect Authorization”或“API Key Authorization”。

$util.log.info(Object) : Void

在 API 上启用请求级和字段级日志记录时，将所提供对象的字符串表示形式 CloudWatch 记录到请求的日志流中。ALL

$util.log.info(String, Object...) : Void

在 API 上启用请求级和字段级日志记录时，将所提供对象的字符串表示形式 CloudWatch 记录到请求的日志流中。ALL该实用程序按顺序将第一个输入格式字符串中由“{}”指示的所有变量替换为提供的对象的字符串表示形式。

$util.log.error(Object) : Void

在 API 上使用日志级别ERROR或日志级别启用字段级日志 CloudWatch 记录时，将所提供对象的字符串表示形式记录到请求的日志流中。ALL

$util.log.error(String, Object...) : Void

在 API 上使用日志级别ERROR或日志级别启用字段级日志 CloudWatch 记录时，将所提供对象的字符串表示形式记录到请求的日志流中。ALL该实用程序按顺序将第一个输入格式字符串中由“{}”指示的所有变量替换为提供的对象的字符串表示形式。

`$util.escapeJavaScript(String) : String`

将输入字符串作为 JavaScript 转义字符串返回。

解析器授权

解析器授权列表

`$util.unauthorized()`

针对被解析的字段引发 `Unauthorized`。可以在请求或响应映射模板中使用该实用程序，以确定是否允许调用方解析该字段。

AWS AppSync 指令

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

AWS AppSync 公开指令以提高开发人员在 VTL 中编写时的生产力。

指令实用程序

`#return(Object)`

`#return(Object)` 允许您提前从任何映射模板返回。`#return(Object)` 类似于编程语言中的 return 关键字，因为它从最近的逻辑范围块返回。在解析器映射模板内使用 `#return(Object)` 将从解析器返回。此外，从函数映射模板中使用 `#return(Object)` 将从该函数返回，并继续运行到管道中的下一个函数或解析器响应映射模板。

`#return`

`#return` 指令具有与 `#return(Object)` 相同的行为，但返回 `null`。

$util.time 中的时间帮助程序

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

$util.time 变量包含的日期时间方法有助于生成时间截，在不同的日期时间格式之间进行转换，并解析日期时间字符串。日期时间格式的语法基于该语法 DateTimeFormatter，您可以参考该语法以获取更多文档。我们在下面提供了一些示例以及可用方法和描述列表。

时间实用程序

时间实用程序列表

```
$util.time.nowISO8601() : String
```

返回 UTC 的 ISO8601 格式字符串表示形式。

```
$util.time.nowEpochSeconds() : long
```

返回从 1970-01-01T00:00:00Z 纪元到现在的秒数。

```
$util.time.nowEpochMilliSeconds() : long
```

返回从 1970-01-01T00:00:00Z 纪元到现在的毫秒数。

```
$util.time.nowFormatted(String) : String
```

使用字符串输入类型指定的格式返回当前 UTC 时间戳的字符串。

```
$util.time.nowFormatted(String, String) : String
```

使用字符串输入类型指定的格式和时区返回该时区当前时间戳的字符串。

```
$util.time.parseFormattedToEpochMilliSeconds(String, String) : Long
```

解析作为字符串传递的时间戳以及格式，然后将时间戳作为自纪元以来的毫秒数返回。

```
$util.time.parseFormattedToEpochMilliSeconds(String, String, String) : Long
```

解析作为字符串传递的时间戳以及格式和时区，然后将时间戳作为自纪元以来的毫秒数返回。

```
$util.time.parseISO8601ToEpochMilliSeconds(String) : Long
```

解析作为字符串传递的 ISO8601 时间戳，然后将时间戳作为自纪元以来的毫秒数返回。

```
$util.time.epochMilliSecondsToSeconds(long) : long
```

将纪元毫秒数时间戳转换为纪元秒数时间戳。

```
$util.time.epochMilliSecondsToISO8601(long) : String
```

将纪元毫秒数时间戳转换为 ISO8601 时间戳。

```
$util.time.epochMilliSecondsToFormatted(long, String) : String
```

将以长型形式传递的纪元毫秒数时间戳转换为根据提供的 UTC 格式设置的时间戳。

```
$util.time.epochMilliSecondsToFormatted(long, String, String) : String
```

将以长型形式传递的纪元毫秒数时间戳转换为根据提供的时区和格式设置的时间戳。

单独函数示例

```
$util.time.nowISO8601()                                              :
 2018-02-06T19:01:35.749Z
$util.time.nowEpochSeconds()                                         : 1517943695
$util.time.nowEpochMilliSeconds()                                    : 1517943695750
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ")                      : 2018-02-06
 19:01:35+0000
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ", "+08:00")            : 2018-02-07
 03:01:35+0800
$util.time.nowFormatted("yyyy-MM-dd HH:mm:ssZ", "Australia/Perth") : 2018-02-07
 03:01:35+0800
```

转换示例

```
#set( $nowEpochMillis = 1517943695758 )
$util.time.epochMilliSecondsToSeconds($nowEpochMillis)
    : 1517943695
$util.time.epochMilliSecondsToISO8601($nowEpochMillis)
    : 2018-02-06T19:01:35.758Z
$util.time.epochMilliSecondsToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ")
    : 2018-02-06 19:01:35+0000
$util.time.epochMilliSecondsToFormatted($nowEpochMillis, "yyyy-MM-dd HH:mm:ssZ",
 "+08:00") : 2018-02-07 03:01:35+0800
```

解析示例

```
$util.time.parseISO8601ToEpochMilliSeconds("2018-02-01T17:21:05.180+08:00")
              : 1517476865180
$util.time.parseFormattedToEpochMilliSeconds("2018-02-02 01:19:22+0800", "yyyy-MM-dd
 HH:mm:ssZ")     : 1517505562000
$util.time.parseFormattedToEpochMilliSeconds("2018-02-02 01:19:22", "yyyy-MM-dd
 HH:mm:ss", "+08:00") : 1517505562000
```

使用 AWS AppSync 已定义的标量

以下格式与 AWSDate、AWSDateTime 和 AWSTime 兼容。

```
$util.time.nowFormatted("yyyy-MM-dd[XXX]", "-07:00:30")               :
 2018-07-11-07:00
$util.time.nowFormatted("yyyy-MM-dd'T'HH:mm:ss[XXXXX]", "-07:00:30")  :
 2018-07-11T15:14:15-07:00:30
```

$util.list 中的列表帮助程序

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

$util.list 包含一些方法以帮助执行常见的列表操作，例如在列表中删除或保留项目以用于筛选使用案例。

列表实用程序

`$util.list.copyAndRetainAll(List, List) : List`

制作第一个参数中提供的列表的浅副本，同时仅保留第二个参数中指定的项目（如果存在）。所有其他项目将从副本中移除。

`$util.list.copyAndRemoveAll(List, List) : List`

制作第一个参数中提供的列表的浅副本，同时删除在第二个参数中指定项目的任何项目（如果存在）。所有其他项目将保留在副本中。

`$util.list.sortList(List, Boolean, String) : List`

对第一个参数中提供的对象列表进行排序。如果第二个参数为 true，则列表按降序进行排序；如果第二个参数为 false，则列表按升序进行排序。第三个参数是用于对自定义对象列表进行排序的属性的字符串名称。如果它只是字符串、整数、浮点数或双精度数列表，则第三个参数可能是任何随机字符串。如果所有对象不是来自同一个类，则返回原始列表。仅支持最多包含 1000 个对象的列表。以下是该实用程序的用法示例：

```
 INPUT:      $util.list.sortList([{"description":"youngest", "age":5},
{"description":"middle", "age":45}, {"description":"oldest", "age":85}], false,
 "description")
 OUTPUT:     [{"description":"middle", "age":45}, {"description":"oldest",
 "age":85}, {"description":"youngest", "age":5}]
```

$util.map 中的映射帮助程序

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

`$util.map` 包含一些方法以帮助执行常见的映射操作，例如在映射中删除或保留项目以用于筛选使用案例。

映射实用程序

`$util.map.copyAndRetainAllKeys(Map, List) : Map`

制作第一个映射的浅副本，同时仅保留列表中指定的键（如果存在）。所有其他键将从副本中移除。

`$util.map.copyAndRemoveAllKeys(Map, List) : Map`

制作第一个映射的浅副本，同时删除在列表中指定键的任何条目（如果存在）。所有其他键将保留在副本中。

$util.dynamodb 中的 DynamoDB 帮助程序

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

`$util.dynamodb` 包含一些帮助程序方法，可以更轻松地在 Amazon DynamoDB 中写入和读取数据，例如自动类型映射和格式设置。这些方法旨在自动将基元类型和列表映射到正确的 DynamoDB 输入格式，即格式 `{ "TYPE" : VALUE }` 的 Map。

例如，以前用于在 DynamoDB 中创建新项目的请求映射模板可能如下所示：

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "id" : { "S" : "$util.autoId()" }
    },
    "attributeValues" : {
         "title" : { "S" : $util.toJson($ctx.args.title) },
         "author" : { "S" : $util.toJson($ctx.args.author) },
         "version" : { "N", $util.toJson($ctx.args.version) }
    }
}
```

如果我们想为对象添加字段，则必须在架构中更新 GraphQL 查询，还要更新请求映射模板。不过，我们现在可以重构请求映射模板，以使其自动选取在架构中添加的新字段，并使用正确的类型将其添加到 DynamoDB 中：

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

在上一示例中，我们使用 `$util.dynamodb.toDynamoDBJson(...)` 帮助程序自动获取生成的 ID，并将其转换为字符串属性的 DynamoDB 表示形式。然后，我们获取所有参数，将其转换为其 DynamoDB 表示形式，并输出到模板中的 `attributeValues` 字段。

每个帮助程序均有两个版本：一个版本返回对象（例如 `$util.dynamodb.toString(...)`）；一个版本将对象返回为 JSON 字符串（例如 `$util.dynamodb.toStringJson(...)`）。在上一示例中，我们使用了将数据返回为 JSON 字符串的版本。如果您希望在模板中使用对象之前处理对象，可以选择返回对象，如下所示：

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "id" : $util.dynamodb.toDynamoDBJson($util.autoId())
    },

    #set( $myFoo = $util.dynamodb.toMapValues($ctx.args) )
    #set( $myFoo.version = $util.dynamodb.toNumber(1) )
    #set( $myFoo.timestamp = $util.dynamodb.toString($util.time.nowISO8601()))

    "attributeValues" : $util.toJson($myFoo)
}
```

在上一示例中，我们将转换的参数作为映射返回，而不是 JSON 字符串；并在使用 `version` 最终输出到模板中的 `timestamp` 字段之前添加了 `attributeValues` 和 `$util.toJson(...)` 字段。

每个帮助程序的 JSON 版本等效于在 `$util.toJson(...)` 中包装非 JSON 版本。例如，以下语句是完全相同的：

```
$util.toStringJson("Hello, World!")
$util.toJson($util.toString("Hello, World!"))
```

toDynamoDB

toDynamoDB 实用程序列表

`$util.dynamodb.toDynamoDB(Object) : Map`

DynamoDB 的常规对象转换工具，可以将输入对象转换为相应的 DynamoDB 表示形式。表示某些类型的方式是自主的：例如，使用列表 ("L") 而不使用集 ("SS", "NS", "BS")。这会返回一个描述 DynamoDB 属性值的对象。

字符串示例

```
Input:      $util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

数字示例

```
Input:      $util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

布尔值示例

```
Input:      $util.dynamodb.toDynamoDB(true)
Output:     { "BOOL" : true }
```

列表示例

```
Input:      $util.dynamodb.toDynamoDB([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

映射示例

```
Input:      $util.dynamodb.toDynamoDB({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
```

```
                        { "S" : "boop" }
                    ]
                }
            }
        }
```

`$util.dynamodb.toDynamoDBJson(Object) : String`

与 `$util.dynamodb.toDynamoDB(Object) : Map` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

toString 实用程序

toString 实用程序列表

`$util.dynamodb.toString(String) : String`

将输入字符串转换为 DynamoDB 字符串格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      $util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

`$util.dynamodb.toStringJson(String) : Map`

与 `$util.dynamodb.toString(String) : String` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

`$util.dynamodb.toStringSet(List<String>) : Map`

将包含字符串的列表转换为 DynamoDB 字符串集格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      $util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

`$util.dynamodb.toStringSetJson(List<String>) : String`

与 `$util.dynamodb.toStringSet(List<String>) : Map` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

toNumber 实用程序

toNumber 实用程序列表

`$util.dynamodb.toNumber(Number) : Map`

将数字转换为 DynamoDB 数字格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      $util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

`$util.dynamodb.toNumberJson(Number) : String`

与 `$util.dynamodb.toNumber(Number) : Map` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

`$util.dynamodb.toNumberSet(List<Number>) : Map`

将数字列表转换为 DynamoDB 数字集格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      $util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

`$util.dynamodb.toNumberSetJson(List<Number>) : String`

与 `$util.dynamodb.toNumberSet(List<Number>) : Map` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

toBinary 实用程序

toBinary 实用程序列表

`$util.dynamodb.toBinary(String) : Map`

将编码为 Base64 字符串的二进制数据转换为 DynamoDB 二进制格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      $util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

`$util.dynamodb.toBinaryJson(String) : String`

与 `$util.dynamodb.toBinary(String) : Map` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

`$util.dynamodb.toBinarySet(List<String>) : Map`

将编码为 Base64 字符串的二进制数据列表转换为 DynamoDB 二进制集格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      $util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

`$util.dynamodb.toBinarySetJson(List<String>) : String`

与 `$util.dynamodb.toBinarySet(List<String>) : Map` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

toBoolean 实用程序

toBoolean 实用程序列表

`$util.dynamodb.toBoolean(Boolean) : Map`

将布尔值转换为相应的 DynamoDB 布尔值格式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      $util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

`$util.dynamodb.toBooleanJson(Boolean) : String`

与 `$util.dynamodb.toBoolean(Boolean) : Map` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

toNull 实用程序

toNull 实用程序列表

`$util.dynamodb.toNull() : Map`

使用 DynamoDB Null 格式返回 Null。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      $util.dynamodb.toNull()
Output:     { "NULL" : null }
```

`$util.dynamodb.toNullJson() : String`

与 `$util.dynamodb.toNull() : Map` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

toList 实用程序

toList 实用程序列表

`$util.dynamodb.toList(List) : Map`

将对象列表转换为 DynamoDB 列表格式。列表中的每个项目也会转换为相应的 DynamoDB 格式。表示某些嵌套对象的方式是自主的：例如，使用列表 ("L") 而不使用集 ("SS", "NS", "BS")。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      $util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
                   }
               ]
           }
```

`$util.dynamodb.toListJson(List) : String`

与 `$util.dynamodb.toList(List) : Map` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

toMap 实用程序

toMap 实用程序列表

`$util.dynamodb.toMap(Map) : Map`

将映射转换为 DynamoDB 映射格式。映射中的每个值也会转换为相应的 DynamoDB 格式。表示某些嵌套对象的方式是自主的：例如，使用列表 ("L") 而不使用集 ("SS", "NS", "BS")。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      $util.dynamodb.toMap({ "foo": "bar", "baz" : 1234, "beep": [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
                           { "S" : "boop" }
                       ]
                   }
               }
           }
```

`$util.dynamodb.toMapJson(Map) : String`

与 `$util.dynamodb.toMap(Map) : Map` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

`$util.dynamodb.toMapValues(Map) : Map`

创建映射的副本，其中每个值都已转换为相应的 DynamoDB 格式。表示某些嵌套对象的方式是自主的：例如，使用列表 ("L") 而不使用集 ("SS", "NS", "BS")。

```
Input:      $util.dynamodb.toMapValues({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "foo"  : { "S" : "bar" },
               "baz"  : { "N" : 1234 },
               "beep" : {
                   "L" : [
                       { "S" : "boop" }
                   ]
               }
```

```
        }
```

> **Note**
>
> 这与 `$util.dynamodb.toMap(Map) : Map` 略有不同，因为它仅返回 DynamoDB 属性值内容，而不返回整个属性值本身。例如，以下语句是完全相同的：
>
> ```
> $util.dynamodb.toMapValues($map)
> $util.dynamodb.toMap($map).get("M")
> ```

`$util.dynamodb.toMapValuesJson(Map) : String`

与 `$util.dynamodb.toMapValues(Map) : Map` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

S3Object 实用程序

S3Object 实用程序列表

`$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`

将键、存储桶和区域转换为 DynamoDB S3 对象表示形式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      $util.dynamodb.toS3Object("foo", "bar", region = "baz")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo", \"bucket\" : \"bar", \"region
\" : \"baz" } }" }
```

`$util.dynamodb.toS3ObjectJson(String key, String bucket, String region) : String`

与 `$util.dynamodb.toS3Object(String key, String bucket, String region) : Map` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

`$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map`

将键、存储桶、区域和可选版本转换为 DynamoDB S3 对象表示形式。这会返回一个描述 DynamoDB 属性值的对象。

```
Input:      $util.dynamodb.toS3Object("foo", "bar", "baz", "beep")
Output:     { "S" : "{ \"s3\" : { \"key\" : \"foo\", \"bucket\" : \"bar\", \"region
\" : \"baz\", \"version\" = \"beep\" } }" }
```

`$util.dynamodb.toS3ObjectJson(String key, String bucket, String region, String version) : String`

与 `$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map` 相同，但以 JSON 编码字符串形式返回 DynamoDB 属性值。

`$util.dynamodb.fromS3ObjectJson(String) : Map`

接受 DynamoDB S3 对象的字符串值，并返回包含键、存储桶、区域和可选版本的映射。

```
Input:      $util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
 \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :
 "beep" }
```

$util.rds 中的 Amazon RDS 帮助程序

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

`$util.rds` 包含一些帮助程序方法，可以删除结果输出中的无关数据以设置 Amazon RDS 操作格式。

$util.rds 实用程序列表

`$util.rds.toJsonString(String serializedSQLResult): String`

将字符串化的原始 Amazon Relational Database Service (Amazon RDS) 数据 API 操作结果格式转换为更简洁的字符串以返回 `String`。返回的字符串是结果集的序列化 SQL 记录列表。每条记录均表示为一个键-值对集合。键是对应的列名称。

如果输入中的相应语句是导致变更的 SQL 查询（例如 INSERT、UPDATE、DELETE），则返回空列表。例如，查询 `select * from Books limit 2` 提供 Amazon RDS 数据操作的原始结果：

```
{
    "sqlStatementResults": [
        {
            "numberOfRecordsUpdated": 0,
            "records": [
                [
                    {
                        "stringValue": "Mark Twain"
                    },
                    {
                        "stringValue": "Adventures of Huckleberry Finn"
                    },
                    {
                        "stringValue": "978-1948132817"
                    }
                ],
                [
                    {
                        "stringValue": "Jack London"
                    },
                    {
                        "stringValue": "The Call of the Wild"
                    },
                    {
                        "stringValue": "978-1948132275"
                    }
                  ]
            ],
            "columnMetadata": [
                {
                    "isSigned": false,
                    "isCurrency": false,
                    "label": "author",
                    "precision": 200,
                    "typeName": "VARCHAR",
                    "scale": 0,
                    "isAutoIncrement": false,
                    "isCaseSensitive": false,
                    "schemaName": "",
                    "tableName": "Books",
                    "type": 12,
                    "nullable": 0,
                    "arrayBaseColumnType": 0,
```

```
                    "name": "author"
                },
                {
                    "isSigned": false,
                    "isCurrency": false,
                    "label": "title",
                    "precision": 200,
                    "typeName": "VARCHAR",
                    "scale": 0,
                    "isAutoIncrement": false,
                    "isCaseSensitive": false,
                    "schemaName": "",
                    "tableName": "Books",
                    "type": 12,
                    "nullable": 0,
                    "arrayBaseColumnType": 0,
                    "name": "title"
                },
                {
                    "isSigned": false,
                    "isCurrency": false,
                    "label": "ISBN-13",
                    "precision": 15,
                    "typeName": "VARCHAR",
                    "scale": 0,
                    "isAutoIncrement": false,
                    "isCaseSensitive": false,
                    "schemaName": "",
                    "tableName": "Books",
                    "type": 12,
                    "nullable": 0,
                    "arrayBaseColumnType": 0,
                    "name": "ISBN-13"
                }
            ]
        }
    ]
}
```

util.rds.toJsonString 是：

```
[
  {
```

```
    "author": "Mark Twain",
    "title": "Adventures of Huckleberry Finn",
    "ISBN-13": "978-1948132817"
  },
  {
    "author": "Jack London",
    "title": "The Call of the Wild",
    "ISBN-13": "978-1948132275"
  },
]
```

`$util.rds.toJsonObject(String serializedSQLResult): Object`

这与 `util.rds.toJsonString` 相同，但结果是 JSON `Object`。

$util.http 中的 HTTP 帮助程序

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

`$util.http` 实用程序提供一些帮助程序方法，可用于管理 HTTP 请求参数和添加响应标头。

$util.http 实用程序列表

`$util.http.copyHeaders(Map) : Map`

从映射中复制标头，不包含受限制的 HTTP 标头集。您可以使用该方法将请求标头转发到下游 HTTP 终端节点。

```
{
    ...
    "params": {
        ...
        "headers": $util.http.copyHeaders($ctx.request.headers),
        ...
    },
    ...
}
```

`$util.http.addResponseHeader(String, Object)`

添加单个自定义标头，其中包含响应的名称 (`String`) 和值 (`Object`)。适用以下限制：

- 标头名称不能与任何现有 AWS 或受限制的 AWS AppSync标题相匹配。
- 标头名称不能以受限制的前缀开头，例如 `x-amzn-` 或 `x-amz-`。
- 自定义响应标头大小不能超过 4 KB。这包括标头名称和值。
- 对于每个 GraphQL 操作，您应该定义一次每个响应标头。不过，如果您多次定义具有相同名称的自定义标头，将在响应中显示最新的定义。无论命名如何，所有标头都会计入标头大小限制。

```
...
$util.http.addResponseHeader("itemsCount", 7)
$util.http.addResponseHeader("render", $ctx.args.render)
...
```

`$util.http.addResponseHeaders(Map)`

将多个响应标头添加到来自指定的名称 (`String`) 和值 (`Object`) 映射的响应中。为 `addResponseHeader(String, Object)` 方法列出的相同限制也适用于该方法。

```
...
#set($headersMap = {})
$util.qr($headersMap.put("headerInt", 12))
$util.qr($headersMap.put("headerString", "stringValue"))
$util.qr($headersMap.put("headerObject", {"field1": 7, "field2": "string"}))
$util.http.addResponseHeaders($headersMap)
...
```

$util.xml 中的 XML 帮助程序

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

`$util.xml` 包含一些帮助程序方法，可以更轻松地将 XML 响应转换为 JSON 或字典。

$util.xml 实用程序列表

`$util.xml.toMap(String) : Map`

将 XML 字符串转换为字典。

```
Input:

<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
</posts>

Output (JSON representation):

{
  "posts":{
    "post":{
      "id":1,
      "title":"Getting started with GraphQL"
    }
  }
}


Input:

<?xml version="1.0" encoding="UTF-8"?>
<posts>
<post>
  <id>1</id>
  <title>Getting started with GraphQL</title>
</post>
<post>
  <id>2</id>
  <title>Getting started with AWS AppSync</title>
</post>
</posts>

Output (JSON representation):
```

```
{
  "posts":{
    "post":[
        {
          "id":1,
          "title":"Getting started with GraphQL"
        },
        {
          "id":2,
          "title":"Getting started with AWS AppSync"
        }
    ]
  }
}
```

`$util.xml.toJsonString(String) : String`

将 XML 字符串转换为 JSON 字符串。这与 toMap 类似，只不过输出是字符串。如果您要直接转换 XML 响应并将其从 HTTP 对象返回到 JSON，这非常有用。

`$util.xml.toJsonString(String, Boolean) : String`

使用可选的布尔值参数将 XML 字符串转换为 JSON 字符串，以确定您是否要对 JSON 进行字符串编码。

$util.transform 中的转换帮助程序

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

`$util.transform` 包含一些帮助程序方法，可以更轻松地对数据来源执行复杂的操作，例如 Amazon DynamoDB 筛选操作。

转换帮助程序

转换帮助程序实用程序列表

`$util.transform.toDynamoDBFilterExpression(Map) : Map`

将输入字符串转换为筛选条件表达式以用于 DynamoDB。

```
Input:

$util.transform.toDynamoDBFilterExpression({
    "title":{
      "contains":"Hello World"
    }
  })

Output:

{
    "expression" : "contains(#title, :title_contains)"
    "expressionNames" : {
        "#title" : "title",
    },
    "expressionValues" : {
        ":title_contains" : { "S" : "Hello World" }
    },
}
```

`$util.transform.toElasticsearchQueryDSL(Map) : Map`

将给定输入转换为其等效的 OpenSearch Query DSL 表达式，将其作为 JSON 字符串返回。

```
Input:

$util.transform.toElasticsearchQueryDSL({
    "upvotes":{
        "ne":15,
        "range":[
            10,
            20
        ]
    },
    "title":{
```

```
        "eq":"hihihi",
        "wildcard":"h*i"
    }
  })

Output:
{
    "bool":{
      "must":[
          {
            "bool":{
              "must":[
                  {
                    "bool":{
                      "must_not":{
                        "term":{
                          "upvotes":15
                        }
                      }
                    }
                  },
                  {
                    "range":{
                      "upvotes":{
                        "gte":10,
                        "lte":20
                      }
                    }
                  }
              ]
            }
          },
          {
            "bool":{
              "must":[
                  {
                    "term":{
                      "title":"hihihi"
                    }
                  },
                  {
                  "wildcard":{
                      "title":"h*i"
                    }
```

```
                    }
                ]
            }
        }
    ]
  }
}
```

默认运算符假定为 AND。

转换帮助程序订阅筛选条件

转换帮助程序订阅筛选条件实用程序列表

`$util.transform.toSubscriptionFilter(Map) : Map`

将 Map 输入对象转换为 SubscriptionFilter 表达式对象。$util.transform.toSubscriptionFilter 方法用作 $extensions.setSubscriptionFilter() 扩展的输入。有关更多信息，请参阅扩展。

`$util.transform.toSubscriptionFilter(Map, List) : Map`

将 Map 输入对象转换为 SubscriptionFilter 表达式对象。$util.transform.toSubscriptionFilter 方法用作 $extensions.setSubscriptionFilter() 扩展的输入。有关更多信息，请参阅扩展。

第一个参数是转换为 SubscriptionFilter 表达式对象的 Map 输入对象。第二个参数是字段名称 List，在构建 SubscriptionFilter 表达式对象时，将在第一个 Map 输入对象中忽略这些字段名称。

`$util.transform.toSubscriptionFilter(Map, List, Map) : Map`

将 Map 输入对象转换为 SubscriptionFilter 表达式对象。$util.transform.toSubscriptionFilter 方法用作 $extensions.setSubscriptionFilter() 扩展的输入。有关更多信息，请参阅扩展。

第一个参数是转换为 SubscriptionFilter 表达式对象的 Map 输入对象，第二个参数是在第一个 Map 输入对象中忽略的字段名称 List，第三个参数是构建 SubscriptionFilter 表达式对象时包含的严格规则 Map 输入对象。这些严格规则包含在 SubscriptionFilter 表达式对象中，以便至少满足其中的一个规则才能通过订阅筛选条件。

订阅筛选条件参数

下表介绍了如何定义以下实用程序的参数：

- `$util.transform.toSubscriptionFilter(Map) : Map`
- `$util.transform.toSubscriptionFilter(Map, List) : Map`
- `$util.transform.toSubscriptionFilter(Map, List, Map) : Map`

Argument 1: Map

参数 1 是一个 Map 对象，它具有以下键值：

- 字段名称
- "and"
- "or"

对于作为键的字段名称，这些字段的条目条件采用 `"operator" : "value"` 格式。

以下示例说明了如何将条目添加到 Map 中：

```
"field_name" : {
                    "operator1" : value
               }

## We can have multiple conditions for the same field_name:

"field_name" : {
                    "operator1" : value
                    "operator2" : value
                    .
                    .
                    .
               }
```

在一个字段具有两个或更多条件时，所有这些条件被视为使用 OR 运算。

输入 Map 也可以将 "and" 和 "or" 作为键，这意味着这些键中的所有条目应根据键使用 AND 或 OR 逻辑进行连接。键值 "and" 和 "or" 需要使用一个条件数组。

```
"and" : [
```

```
            {
                "field_name1" : {
                    "operator1" : value
                }
             },

             {
                "field_name2" : {
                    "operator1" : value
                }
             },
             .
             .
        ].
```

请注意，您可以嵌套 "and" 和 "or"。也就是说，您可以将 "and"/"or" 嵌套在另一个 "and"/"or" 块中。不过，这不适用于简单字段。

```
"and" : [

            {
                "field_name1" : {
                    "operator" : value
                }
             },

             {
                "or" : [
                            {
                                "field_name2" : {
                                    "operator" : value
                                }
                            },

                            {
                                "field_name3" : {
                                    "operator" : value
                                }
                            }

                       ].
```

以下示例显示使用 $util.transform.toSubscriptionFilter(Map) : Map 的参数 1 的输入。

输入

参数 1：Map：

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

输出

结果是一个 Map 对象：

```
{
```

```
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Admin"
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "lte",
          "value": 50
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "isPublished",
```

```
          "operator": "eq",
          "value": false
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "gte",
          "value": 20
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 2000
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Admin"
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "percentageUp",
          "operator": "gte",
          "value": 20
        },
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
```

```
          "value": 2000
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 2: List

参数 2 包含字段名称 `List`，在构建 `SubscriptionFilter` 表达式对象时，不应在输入 `Map`（参数 1）中考虑使用这些字段名称。`List` 也可以是空的。

以下示例显示使用 `$util.transform.toSubscriptionFilter(Map, List) : Map` 的参数 1 和参数 2 的输入。

输入

参数 1：Map：

```
{

  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {
```

```
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

参数 2：List：

```
["percentageUp", "author"]
```

输出

结果是一个 Map 对象：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 3: Map

参数 3 是一个将字段名称作为键值的 Map 对象（不能具有 "and" 或 "or"）。对于作为键的字段名称，这些字段的条件是采用 "operator" : "value" 格式的条目。与参数 1 不同，参数 3 不能在同一键中具有多个条件。此外，参数 3 没有 "and" 或 "or" 子句，因此，也不涉及嵌套。

参数 3 表示一组严格规则，这些规则将添加到 SubscriptionFilter 表达式对象中，以便至少满足其中的一个条件才能通过筛选条件。

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.
```

以下示例显示使用 $util.transform.toSubscriptionFilter(Map, List, Map) : Map 的参数 1、参数 2 和参数 3 的输入。

输入

参数 1：Map：

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
```

```
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

参数 2：List：

```
["percentageUp", "author"]
```

参数 3：Map：

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
}
```

输出

结果是一个 Map 对象：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
```

```
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        },
        {
          "fieldName": "upvotes",
          "operator": "gte",
          "value": 250
        }
      ]
    },
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        },
        {
          "fieldName": "author",
          "operator": "eq",
          "value": "Person1"
        }
      ]
    }
  ]
```

```
}
```

$util.math 中的数学帮助程序

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

`$util.math` 包含一些方法以帮助进行常见的数学运算。

$util.math 实用程序列表

`$util.math.roundNum(Double) : Integer`

获取一个双精度值，并将其四舍五入到最接近的整数。

`$util.math.minVal(Double, Double) : Double`

获取两个双精度值，并返回两个双精度值之间的最小值。

`$util.math.maxVal(Double, Double) : Double`

获取两个双精度值，并返回两个双精度值之间的最大值。

`$util.math.randomDouble() : Double`

返回 0 到 1 之间的随机双精度值。

> **Important**
>
> 不应将该函数用于任何需要高熵随机性的场合（例如加密）。

`$util.math.randomWithinRange(Integer, Integer) : Integer`

返回指定范围内的随机整数值，第一个参数指定范围的下限值，第二个参数指定范围的上限值。

> **Important**
>
> 不应将该函数用于任何需要高熵随机性的场合（例如加密）。

$util.str 中的字符串帮助程序

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

`$util.str` 包含一些方法以帮助执行常见的字符串操作。

$util.str 实用程序列表

`$util.str.toUpper(String) : String`

获取一个字符串，并将其转换为完全大写。

`$util.str.toLower(String) : String`

获取一个字符串，并将其转换为完全小写。

`$util.str.toReplace(String, String, String) : String`

将字符串中的子字符串替换为另一个字符串。第一个参数指定要执行替换操作的字符串。第二个参数指定要替换的子字符串。第三个参数指定要替换第二个参数的字符串。以下是该实用程序的用法示例：

```
INPUT:      $util.str.toReplace("hello world", "hello", "mellow")
OUTPUT:     "mellow world"
```

`$util.str.normalize(String, String) : String`

使用以下 4 种 unicode 规范化形式之一规范化字符串：NFC、NFD、NFKC 或 NFKD。第一个参数是要规范化的字符串。第二个参数是 "nfc"、"nfd"、"nfkc" 或 "nfkd"，它指定用于规范化过程的规范化类型。

扩展程序

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

`$extensions` 包含一组在解析器中执行额外操作的方法。

$扩展名。 evictFromApi缓存（字符串、字符串、对象）：对象

从 AWS AppSync 服务器端缓存中移出一个项目。第一个参数是类型名称。第二个参数是字段名称。第三个参数是一个对象，其中包含指定缓存键值的键值对项目。您必须按照与缓存解析器的 `cachingKey` 中的缓存键相同的顺序，将项目放入对象中。

> **Note**
>
> 该实用程序仅适用于变更，而不适用于查询。

$扩展名。 setSubscriptionFilter(filterJsonObject)

定义增强的订阅筛选条件。每个订阅通知事件都会根据提供的订阅筛选条件进行评估，如果所有筛选条件的评估结果均为 `true`，则向客户端发送通知。参数是 `filterJsonObject`，如下所述。

> **Note**
>
> 您只能在订阅解析器的响应映射模板中使用该扩展方法。

$扩展名。 setSubscriptionInvalidation过滤器 (filterJsonObject)

定义订阅失效筛选条件。根据失效负载评估订阅筛选条件，如果筛选条件的评估结果为 `true`，则使给定订阅失效。参数是 `filterJsonObject`，如下所述。

> **Note**
>
> 您只能在订阅解析器的响应映射模板中使用该扩展方法。

论点： filterJsonObject

JSON 对象定义订阅或失效筛选条件。它是 filterGroup 中的筛选条件数组。每个筛选条件是单独筛选条件的集合。

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

每个筛选条件具有三个属性：

- fieldName - GraphQL 架构字段。
- operator - 运算符类型。
- value - 与订阅通知 fieldName 值进行比较的值。

以下是这些属性的分配示例：

```
{
 "fieldName" : "severity",
 "operator" : "le",
```

```
 "value" : $context.result.severity
}
```

字段：fieldName

字符串类型 `fieldName` 指的是 GraphQL 架构中定义的一个字段，它与订阅通知负载中的 `fieldName` 匹配。在找到匹配项时，GraphQL 架构字段的 `value` 与订阅通知筛选条件的 `value` 进行比较。在以下示例中，`fieldName` 筛选条件与给定 GraphQL 类型中定义的 `service` 字段匹配。如果通知负载包含 `service` 字段，并且 `value` 等于 `AWS AppSync`，则筛选条件的评估结果为 `true`：

```
{
 "fieldName" : "service",
 "operator" : "eq",
 "value" : "AWS AppSync"
}
```

字段：value

根据运算符，该值可能具有不同的类型：

- 单个数字或布尔值
  - 字符串示例：`"test"`、`"service"`
  - 数字示例：`1`、`2`、`45.75`
  - 布尔值示例：`true`、`false`
- 数字或字符串对
  - 字符串对示例：`["test1","test2"]`、`["start","end"]`
  - 数字对示例：`[1,4]`、`[67,89]`、`[12.45, 95.45]`
- 数字或字符串数组
  - 字符串数组示例：`["test1","test2","test3","test4","test5"]`
  - 数字数组示例：`[1,2,3,4,5]`、`[12.11,46.13,45.09,12.54,13.89]`

字段：operator

一个区分大小写的字符串，可能具有以下值：

运算符	描述	可能的值类型

eq	Equal	整数、浮点数、字符串、布尔值
没有	Not equal	整数、浮点数、字符串、布尔值
le	Less than or equal	整数、浮点数、字符串
lt	Less than	整数、浮点数、字符串
ge	Greater than or equal	整数、浮点数、字符串
gt	Greater than	整数、浮点数、字符串
contains	检查集合中的子序列或值。	整数、浮点数、字符串
不包含	检查集合中是否缺少子序列或缺少值。	整数、浮点数、字符串
开始于	检查前缀。	字符串
in	检查列表中是否有匹配的元素。	整数、浮点数或字符串数组
notIn	检查不在列表中的匹配元素。	整数、浮点数或字符串数组
介于	在两个值之间	整数、浮点数、字符串
包含任何	包含常见元素	整数、浮点数、字符串

下表介绍了如何在订阅通知中使用每个运算符。

eq (equal)

如果订阅通知字段值匹配并且严格等于筛选条件的值，则 eq 运算符的评估结果为 true。在以下示例中，如果订阅通知的 service 字段值等于 AWS AppSync，则筛选条件的评估结果为 true。

可能的值类型：整数、浮点数、字符串、布尔值

```
{
 "fieldName" : "service",
```

```
 "operator" : "eq",
 "value" : "AWS AppSync"
}
```

ne (not equal)

如果订阅通知字段值与筛选条件的值不同，则 ne 运算符的评估结果为 true。在以下示例中，如果订阅通知的 service 字段值不同于 AWS AppSync，则筛选条件的评估结果为 true。

可能的值类型：整数、浮点数、字符串、布尔值

```
{
 "fieldName" : "service",
 "operator" : "ne",
 "value" : "AWS AppSync"
}
```

le (less or equal)

如果订阅通知字段值小于或等于筛选条件的值，则 le 运算符的评估结果为 true。在以下示例中，如果订阅通知的 size 字段值小于或等于 5，则筛选条件的评估结果为 true。

可能的值类型：整数、浮点数、字符串

```
{
 "fieldName" : "size",
 "operator" : "le",
 "value" : 5
}
```

lt (less than)

如果订阅通知字段值小于筛选条件的值，则 lt 运算符的评估结果为 true。在以下示例中，如果订阅通知的 size 字段值小于 5，则筛选条件的评估结果为 true。

可能的值类型：整数、浮点数、字符串

```
{
 "fieldName" : "size",
 "operator" : "lt",
 "value" : 5
}
```

ge (greater or equal)

如果订阅通知字段值大于或等于筛选条件的值，则 ge 运算符的评估结果为 true。在以下示例中，如果订阅通知的 size 字段值大于或等于 5，则筛选条件的评估结果为 true。

可能的值类型：整数、浮点数、字符串

```
{
 "fieldName" : "size",
 "operator" : "ge",
 "value" : 5
}
```

gt (greater than)

如果订阅通知字段值大于筛选条件的值，则 gt 运算符的评估结果为 true。在以下示例中，如果订阅通知的 size 字段值大于 5，则筛选条件的评估结果为 true。

可能的值类型：整数、浮点数、字符串

```
{
 "fieldName" : "size",
 "operator" : "gt",
 "value" : 5
}
```

contains

contains 运算符检查集合或单个项目中的子字符串、子序列或值。如果订阅通知字段值包含筛选条件值，则具有 contains 运算符的筛选条件的评估结果为 true。在以下示例中，如果订阅通知的 seats 字段具有包含值 10 的数组值，则筛选条件的评估结果为 true。

可能的值类型：整数、浮点数、字符串

```
{
 "fieldName" : "seats",
 "operator" : "contains",
 "value" : 10
}
```

在另一个示例中，如果订阅通知的 event 字段将 launch 作为子字符串，则筛选条件的评估结果为 true。

```
{
 "fieldName" : "event",
 "operator" : "contains",
 "value" : "launch"
}
```

notContains

notContains 运算符检查在集合或单个项目中是否缺少子字符串、子序列或值。如果订阅通知字段值不包含筛选条件值，则具有 notContains 运算符的筛选条件的评估结果为 true。在以下示例中，如果订阅通知的 seats 字段具有不包含值 10 的数组值，则筛选条件的评估结果为 true。

可能的值类型：整数、浮点数、字符串

```
{
 "fieldName" : "seats",
 "operator" : "notContains",
 "value" : 10
}
```

在另一个示例中，如果订阅通知的 event 字段值没有将 launch 作为其子序列，则筛选条件的评估结果为 true。

```
{
 "fieldName" : "event",
 "operator" : "notContains",
 "value" : "launch"
}
```

beginsWith

beginsWith 运算符检查字符串中的前缀。如果订阅通知字段值以筛选条件的值开头，则包含 beginsWith 运算符的筛选条件的评估结果为 true。在以下示例中，如果订阅通知的 service 字段值以 AWS 开头，则筛选条件的评估结果为 true。

可能的值类型：字符串

```
{
 "fieldName" : "service",
 "operator" : "beginsWith",
 "value" : "AWS"
```

```
}
```

in

in 运算符在数组中检查匹配元素。如果订阅通知字段值在数组中存在，则包含 in 运算符的筛选条件的评估结果为 true。在以下示例中，如果订阅通知的 severity 字段具有数组 [1,2,3] 中存在的值之一，则筛选条件的评估结果为 true。

可能的值类型：整数、浮点数或字符串数组

```
{
 "fieldName" : "severity",
 "operator" : "in",
 "value" : [1,2,3]
}
```

notIn

notIn 运算符在数组中检查缺失元素。如果订阅通知字段值在数组中不存在，则包含 notIn 运算符的筛选条件的评估结果为 true。在以下示例中，如果订阅通知的 severity 字段具有数组 [1,2,3] 中不存在的值之一，则筛选条件的评估结果为 true。

可能的值类型：整数、浮点数或字符串数组

```
{
 "fieldName" : "severity",
 "operator" : "notIn",
 "value" : [1,2,3]
}
```

between

between 运算符检查两个数字或字符串之间的值。如果订阅通知字段值位于筛选条件的值对之间，则包含 between 运算符的筛选条件的评估结果为 true。在以下示例中，如果订阅通知的 severity 字段值为 2、3、4，则筛选条件的评估结果为 true。

可能的值类型：整数、浮点数或字符串对

```
{
 "fieldName" : "severity",
 "operator" : "between",
 "value" : [1,5]
```

```
}
```

containsAny

containsAny 运算符检查数组中的相同元素。如果订阅通知字段设置值和筛选条件设置值的交集不为空，则具有 containsAny 运算符的筛选条件的评估结果为 true。在以下示例中，如果订阅通知的 seats 字段具有包含 10 或 15 的数组值，则筛选条件的评估结果为 true。这意味着如果订阅通知的 seats 字段值为 [10,11] 或 [15,20,30]，则筛选条件的评估结果为 true。

可能的值类型：整数、浮点数或字符串

```
{
 "fieldName" : "seats",
 "operator" : "containsAny",
 "value" : [10, 15]
}
```

AND 逻辑

您可以在 filterGroup 数组的 filters 对象中定义多个条目，以使用 AND 逻辑组合多个筛选条件。在以下示例中，如果订阅通知的 userId 字段值等于 1 并且 group 字段的值为 Admin 或 Developer，则筛选条件的评估结果为 true。

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
```

```
}
```

OR 逻辑

您可以在 filterGroup 数组中定义多个筛选条件对象，以使用 OR 逻辑组合多个筛选条件。在以下示例中，如果订阅通知的 userId 字段值等于 1 或者 group 字段的值为 Admin 或 Developer，则筛选条件的评估结果为 true。

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

异常

请注意，使用筛选条件存在一些限制：

- 在 filters 对象中，每个筛选条件最多可以具有 5 个唯一的 fieldName 项目。这意味着，您可以使用 AND 逻辑组合最多 5 个单独的 fieldName 对象。
- containsAny 运算符最多可以具有 20 个值。
- in 和 notIn 运算符最多可以具有 5 个值。

- 每个字符串最多可以包含 256 个字符。
- 每个字符串比较都区分大小写。
- 嵌套对象筛选最多允许 5 个嵌套筛选级别。
- 每个 filterGroup 最多可以具有 10 个 filters。这意味着，您可以使用 OR 逻辑最多组合 10 个 filters。
  - in 运算符是 OR 逻辑的特例。在以下示例中，具有两个 filters：

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]
        }
    ]
}
```

前面的筛选条件组按以下方式进行评估，并计入最大筛选条件限制：

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "eq",
                    "value" : "Admin"
```

```
                }
            ]
        },
        {
            "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "eq",
                    "value" : "Developer"
                }
            ]
        }
    ]
}
```

$extensions.invalidateSubsion invalidationJsonObject

用于启动变更导致的订阅失效。参数是 `invalidationJsonObject`，如下所述。

> **Note**
>
> 只能在变更解析器的响应映射模板中使用该扩展。
> 您最多只能在任何单个请求中使用 5 个唯一的 `$extensions.invalidateSubscriptions()` 方法调用。如果超过该限制，您将收到 GraphQL 错误。

论点： invalidationJsonObject

`invalidationJsonObject` 定义以下内容：

- `subscriptionField` - 要失效的 GraphQL 架构订阅。单个订阅（在 `subscriptionField` 中定义为字符串）被视为失效。
- `payload` - 一个键值对列表，如果失效筛选条件根据其值评估的结果为 `true`，则将该列表作为使订阅失效的输入。

在订阅解析器中定义的失效筛选条件根据 payload 值评估的结果为 true 时，以下示例导致使用 onUserDelete 订阅的订阅和连接的客户端失效。

```
$extensions.invalidateSubscriptions({
        "subscriptionField": "onUserDelete",
        "payload": {
                "group": "Developer"
                "type" : "Full-Time"
      }
    })
```

DynamoDB 的解析器映射模板参考

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

通过使用 AWS AppSync DynamoDB 解析器，您可以使用 GraphQL 在您的账户的现有 Amazon DynamoDB 表中存储和检索数据。该解析器的工作方式是，允许您将传入的 GraphQL 请求映射到 DynamoDB 调用，然后将 DynamoDB 响应映射回 GraphQL。本节介绍了支持的 DynamoDB 操作的映射模板。

GetItem

通过使用 GetItem 请求映射文档，您可以指示 AWS AppSync DynamoDB 解析器向 DynamoDB 发出 GetItem 请求，并允许您指定：

- DynamoDB 中的项目的键
- 是否使用一致性读取

GetItem 映射文档具有以下结构：

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
```

```
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true,
    "projection" : {
        ...
    }
}
```

字段定义如下：

GetItem 字段

GetItem 字段列表

`version`

模板定义版本。目前支持 `2017-02-28` 和 `2018-05-29`。该值为必填项。

`operation`

要执行的 DynamoDB 操作。要执行 `GetItem` DynamoDB 操作，该字段必须设置为 `GetItem`。该值为必填项。

`key`

DynamoDB 中的项目的键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

`consistentRead`

是否对 DynamoDB 执行强一致性读取。这是可选的，默认值为 `false`。

`projection`

用于指定从 DynamoDB 操作返回的属性的投影。有关投影的更多信息，请参阅投影。该字段是可选的。

从 DynamoDB 返回的项目将自动转换为 GraphQL 和 JSON 基元类型，并且可以在映射上下文 (`$context.result`) 中使用。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关响应映射模板的更多信息，请参阅解析器映射模板概述。

示例

以下示例是 GraphQL 查询 `getThing(foo: String!, bar: String!)` 的映射模板：

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
        "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
    },
    "consistentRead" : true
}
```

有关 DynamoDB `GetItem` API 的更多信息，请参阅 DynamoDB API 文档。

PutItem

通过使用 `PutItem` 请求映射文档，您可以指示 AWS AppSync DynamoDB 解析器向 DynamoDB 发出 `PutItem` 请求，并允许您指定以下内容：

- DynamoDB 中的项目的键
- 项目的完整内容（包括 `key` 和 `attributeValues`）
- 操作成功执行的条件

`PutItem` 映射文档具有以下结构：

```
{
    "version" : "2018-05-29",
    "operation" : "PutItem",
    "customPartitionKey" : "foo",
    "populateIndexFields" : boolean value,
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "attributeValues" : {
        "baz" : ... typed value
    },
```

```
    "condition" : {
       ...
    },
    "_version" : 1
}
```

字段定义如下：

PutItem 字段

PutItem 字段列表

`version`

模板定义版本。目前支持 `2017-02-28` 和 `2018-05-29`。该值为必填项。

`operation`

要执行的 DynamoDB 操作。要执行 `PutItem` DynamoDB 操作，该字段必须设置为 `PutItem`。该值为必填项。

`key`

DynamoDB 中的项目的键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

`attributeValues`

要放入 DynamoDB 中的项目的其余属性。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该字段是可选的。

`condition`

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。如果未指定条件，则 `PutItem` 请求将覆盖该项目的任何现有条目。有关条件的更多信息，请参阅条件表达式。该值为可选项。

`_version`

表示项目的最新已知版本的数值。该值为可选项。该字段用于冲突检测 ，仅受版本化数据源支持。

`customPartitionKey`

如果启用，该字符串值修改启用了版本控制时增量同步表使用的 `ds_sk` 和 `ds_pk` 记录的格式（有关更多信息，请参阅《AWS AppSync 开发人员指南》中的冲突检测和同步）。如果启用，还会启用 `populateIndexFields` 条目处理。该字段是可选的。

`populateIndexFields`

一个布尔值，在与 **`customPartitionKey`** 一起启用时，它为增量同步表中的每个记录创建新条目，具体来说是在 `gsi_ds_pk` 和 `gsi_ds_sk` 列中创建新条目。有关更多信息，请参阅《AWS AppSync 开发人员指南》中的冲突检测和同步。该字段是可选的。

写入到 DynamoDB 的项目将自动转换为 GraphQL 和 JSON 基元类型，并且可以在映射上下文 (`$context.result`) 中使用。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关响应映射模板的更多信息，请参阅解析器映射模板概述。

示例 1

以下示例是 GraphQL 变更 `updateThing(foo: String!, bar: String!, name: String!, version: Int!)` 的映射模板。

如果带指定键的项目不存在，则会创建它。如果带指定键的项已存在，则会覆盖它。

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
        "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
    },
    "attributeValues" : {
        "name"    : $util.dynamodb.toDynamoDBJson($ctx.args.name),
        "version" : $util.dynamodb.toDynamoDBJson($ctx.args.version)
    }
}
```

示例 2

以下示例是 GraphQL 变更 `updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)` 的映射模板。

该示例检查以确保当前位于 DynamoDB 中的项目的 `version` 字段设置为 `expectedVersion`。

```
{
    "version" : "2017-02-28",
```

```
    "operation" : "PutItem",
    "key": {
        "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
        "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
    },
    "attributeValues" : {
        "name"    : $util.dynamodb.toDynamoDBJson($ctx.args.name),
        #set( $newVersion = $context.arguments.expectedVersion + 1 )
        "version" : $util.dynamodb.toDynamoDBJson($newVersion)
    },
    "condition" : {
        "expression" : "version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" : $util.dynamodb.toDynamoDBJson($expectedVersion)
        }
    }
}
```

有关 DynamoDB `PutItem` API 的更多信息，请参阅 DynamoDB API 文档。

UpdateItem

通过使用 `UpdateItem` 请求映射文档，您可以指示 AWS AppSync DynamoDB 解析器向 DynamoDB 发出 `UpdateItem` 请求，并允许您指定以下内容：

- DynamoDB 中的项目的键
- 描述如何更新 DynamoDB 中的项目的更新表达式
- 操作成功执行的条件

`UpdateItem` 映射文档具有以下结构：

```
{
    "version" : "2018-05-29",
    "operation" : "UpdateItem",
    "customPartitionKey" : "foo",
    "populateIndexFields" : boolean value,
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "update" : {
```

```
        "expression" : "someExpression",
        "expressionNames" : {
           "#foo" : "foo"
       },
       "expressionValues" : {
           ":bar" : ... typed value
       }
    },
    "condition" : {
        ...
    },
    "_version" : 1
}
```

字段定义如下：

UpdateItem 字段

UpdateItem 字段列表

`version`

模板定义版本。目前支持 `2017-02-28` 和 `2018-05-29`。该值为必填项。

`operation`

要执行的 DynamoDB 操作。要执行 `UpdateItem` DynamoDB 操作，该字段必须设置为 `UpdateItem`。该值为必填项。

`key`

DynamoDB 中的项目的键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

`update`

`update` 部分用于指定一个更新表达式，以描述如何更新 DynamoDB 中的项目。有关如何编写更新表达式的更多信息，请参阅 DynamoDB UpdateExpressions 文档。此部分是必需的。

`update` 部分有三个组成部分：

`expression`

更新表达式。该值为必填项。

expressionNames

以键值对形式替换表达式属性名称 占位符。键对应于 expression 中使用的名称占位符，值必须是与 DynamoDB 中的项目的属性名称对应的字符串。该字段是可选的，只应填充 expression 中使用的表达式属性名称占位符的替换内容。

expressionValues

以键值对形式替换表达式属性值 占位符。键对应于 expression 中使用的值占位符，而值必须为类型化值。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。必须指定此值。该字段是可选的，只应填充 expression 中使用的表达式属性值占位符的替换内容。

condition

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。如果未指定条件，则 UpdateItem 请求将更新现有条目，而不考虑其当前状态。有关条件的更多信息，请参阅条件表达式。该值为可选项。

_version

表示项目的最新已知版本的数值。该值为可选项。该字段用于冲突检测 ，仅受版本化数据源支持。

customPartitionKey

如果启用，该字符串值修改启用了版本控制时增量同步表使用的 ds_sk 和 ds_pk 记录的格式（有关更多信息，请参阅《AWS AppSync 开发人员指南》中的冲突检测和同步）。如果启用，还会启用 populateIndexFields 条目处理。该字段是可选的。

populateIndexFields

一个布尔值，在与 **customPartitionKey** 一起启用时，它为增量同步表中的每个记录创建新条目，具体来说是在 gsi_ds_pk 和 gsi_ds_sk 列中创建新条目。有关更多信息，请参阅《AWS AppSync 开发人员指南》中的冲突检测和同步。该字段是可选的。

在 DynamoDB 中更新的项目将自动转换为 GraphQL 和 JSON 基元类型，并且可以在映射上下文 ($context.result) 中使用。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关响应映射模板的更多信息，请参阅解析器映射模板概述。

示例 1

以下示例是 GraphQL 变更 upvote(id: ID!) 的映射模板。

在该示例中，DynamoDB 中的项目的 upvotes 和 version 字段递增 1。

```
{
    "version" : "2017-02-28",
    "operation" : "UpdateItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "update" : {
        "expression" : "ADD #votefield :plusOne, version :plusOne",
        "expressionNames" : {
            "#votefield" : "upvotes"
        },
        "expressionValues" : {
            ":plusOne" : { "N" : 1 }
        }
    }
}
```

示例 2

以下示例是 GraphQL 变更 updateItem(id: ID!, title: String, author: String, expectedVersion: Int!) 的映射模板。

这是一个复杂的示例，用于检查参数并动态生成更新表达式，此表达式仅包含由客户端提供的参数。例如，如果省略了 title 和 author，则不会更新它们。如果指定了一个参数，但该参数的值为 null，则会从 DynamoDB 上的对象中删除该字段。最后，该操作具有一个条件，用于验证目前位于 DynamoDB 中的项目的 version 字段是否设置为 expectedVersion：

```
{
    "version" : "2017-02-28",

    "operation" : "UpdateItem",

    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },

    ## Set up some space to keep track of things we're updating **
    #set( $expNames  = {} )
    #set( $expValues = {} )
    #set( $expSet = {} )
```

```
    #set( $expAdd = {} )
    #set( $expRemove = [] )

    ## Increment "version" by 1 **
    $!{expAdd.put("version", ":newVersion")}
    $!{expValues.put(":newVersion", { "N" : 1 })}

    ## Iterate through each argument, skipping "id" and "expectedVersion" **
    #foreach( $entry in $context.arguments.entrySet() )
        #if( $entry.key != "id" && $entry.key != "expectedVersion" )
            #if( (!$entry.value) && ("$!{entry.value}" == "") )
                ## If the argument is set to "null", then remove that attribute from
the item in DynamoDB **

                #set( $discard = ${expRemove.add("#${entry.key}")} )
                $!{expNames.put("#${entry.key}", "$entry.key")}
            #else
                ## Otherwise set (or update) the attribute on the item in DynamoDB **

                $!{expSet.put("#${entry.key}", ":${entry.key}")}
                $!{expNames.put("#${entry.key}", "$entry.key")}

                #if( $entry.key == "ups" || $entry.key == "downs" )
                    $!{expValues.put(":${entry.key}", { "N" : $entry.value })}
                #else
                    $!{expValues.put(":${entry.key}", { "S" : "${entry.value}" })}
                #end
            #end
        #end
    #end

    ## Start building the update expression, starting with attributes we're going to
SET **
    #set( $expression = "" )
    #if( !${expSet.isEmpty()} )
        #set( $expression = "SET" )
        #foreach( $entry in $expSet.entrySet() )
            #set( $expression = "${expression} ${entry.key} = ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end
```

```
    ## Continue building the update expression, adding attributes we're going to ADD **
    #if( !${expAdd.isEmpty()} )
        #set( $expression = "${expression} ADD" )
        #foreach( $entry in $expAdd.entrySet() )
            #set( $expression = "${expression} ${entry.key} ${entry.value}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Continue building the update expression, adding attributes we're going to REMOVE
**
    #if( !${expRemove.isEmpty()} )
        #set( $expression = "${expression} REMOVE" )

        #foreach( $entry in $expRemove )
            #set( $expression = "${expression} ${entry}" )
            #if ( $foreach.hasNext )
                #set( $expression = "${expression}," )
            #end
        #end
    #end

    ## Finally, write the update expression into the document, along with any
expressionNames and expressionValues **
    "update" : {
        "expression" : "${expression}"
        #if( !${expNames.isEmpty()} )
            ,"expressionNames" : $utils.toJson($expNames)
        #end
        #if( !${expValues.isEmpty()} )
            ,"expressionValues" : $utils.toJson($expValues)
        #end
    },

    "condition" : {
        "expression"       : "version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" :
$util.dynamodb.toDynamoDBJson($ctx.args.expectedVersion)
        }
    }
```

```
}
```

有关 DynamoDB `UpdateItem` API 的更多信息，请参阅 DynamoDB API 文档。

DeleteItem

通过使用 `DeleteItem` 请求映射文档，您可以指示 AWS AppSync DynamoDB 解析器向 DynamoDB 发出 `DeleteItem` 请求，并允许您指定以下内容：

- DynamoDB 中的项目的键
- 操作成功执行的条件

`DeleteItem` 映射文档具有以下结构：

```
{
    "version" : "2018-05-29",
    "operation" : "DeleteItem",
    "customPartitionKey" : "foo",
    "populateIndexFields" : boolean value,
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "condition" : {
        ...
    },
    "_version" : 1
}
```

字段定义如下：

DeleteItem 字段

DeleteItem 字段列表

`version`

模板定义版本。目前支持 `2017-02-28` 和 `2018-05-29`。该值为必填项。

operation

要执行的 DynamoDB 操作。要执行 DeleteItem DynamoDB 操作，该字段必须设置为 DeleteItem。该值为必填项。

key

DynamoDB 中的项目的键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

condition

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。如果未指定条件，则 DeleteItem 请求将删除项目，而不考虑其当前状态。有关条件的更多信息，请参阅条件表达式。该值为可选项。

_version

表示项目的最新已知版本的数值。该值为可选项。该字段用于冲突检测 ，仅受版本化数据源支持。

customPartitionKey

如果启用，该字符串值修改启用了版本控制时增量同步表使用的 ds_sk 和 ds_pk 记录的格式（有关更多信息，请参阅《AWS AppSync 开发人员指南》中的冲突检测和同步）。如果启用，还会启用 populateIndexFields 条目处理。该字段是可选的。

populateIndexFields

一个布尔值，在与 **customPartitionKey** 一起启用时，它为增量同步表中的每个记录创建新条目，具体来说是在 gsi_ds_pk 和 gsi_ds_sk 列中创建新条目。有关更多信息，请参阅《AWS AppSync 开发人员指南》中的冲突检测和同步。该字段是可选的。

从 DynamoDB 中删除的项目将自动转换为 GraphQL 和 JSON 基元类型，并且可以在映射上下文 ($context.result) 中使用。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关响应映射模板的更多信息，请参阅解析器映射模板概述。

示例 1

以下示例是 GraphQL 变更 deleteItem(id: ID!) 的映射模板。如果具有此 ID 的项目存在，它将被删除。

```
{
    "version" : "2017-02-28",
    "operation" : "DeleteItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

示例 2

以下示例是 GraphQL 变更 `deleteItem(id: ID!, expectedVersion: Int!)` 的映射模板。如果具有此 ID 的项目存在，它将被删除，但仅当其 `version` 字段设置为 `expectedVersion` 时：

```
{
    "version" : "2017-02-28",
    "operation" : "DeleteItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "condition" : {
        "expression"       : "attribute_not_exists(id) OR version = :expectedVersion",
        "expressionValues" : {
            ":expectedVersion" : $util.dynamodb.toDynamoDBJson($expectedVersion)
        }
    }
}
```

有关 DynamoDB `DeleteItem` API 的更多信息，请参阅 DynamoDB API 文档。

Query

通过使用 `Query` 请求映射文档，您可以指示 AWS AppSync DynamoDB 解析器向 DynamoDB 发出 `Query` 请求，并允许您指定以下内容：

- 键表达式
- 要使用的索引
- 任何额外的筛选条件
- 要返回多少个项目
- 是否使用一致性读取

- 查询方向（向前或向后）
- 分页标记

Query 映射文档具有以下结构：

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "query" : {
        "expression" : "some expression",
        "expressionNames" : {
            "#foo" : "foo"
        },
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "index" : "fooIndex",
    "nextToken" : "a pagination token",
    "limit" : 10,
    "scanIndexForward" : true,
    "consistentRead" : false,
    "select" : "ALL_ATTRIBUTES" | "ALL_PROJECTED_ATTRIBUTES" | "SPECIFIC_ATTRIBUTES",
    "filter" : {
        ...
    },
    "projection" : {
        ...
    }
}
```

字段定义如下：

Query 字段

Query 字段列表

`version`

模板定义版本。目前支持 2017-02-28 和 2018-05-29。该值为必填项。

operation

要执行的 DynamoDB 操作。要执行 Query DynamoDB 操作，该字段必须设置为 Query。该值为必填项。

query

query 部分用于指定一个键条件表达式，用于描述要从 DynamoDB 中检索哪些项目。有关如何编写键条件表达式的更多信息，请参阅 DynamoDB KeyConditions 文档。必须指定此部分。

expression

查询表达式。必须指定该字段。

expressionNames

以键值对形式替换表达式属性名称 占位符。键对应于 expression 中使用的名称占位符，值必须是与 DynamoDB 中的项目的属性名称对应的字符串。该字段是可选的，只应填充 expression 中使用的表达式属性名称占位符的替换内容。

expressionValues

以键值对形式替换表达式属性值 占位符。键对应于 expression 中使用的值占位符，而值必须为类型化值。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。该字段是可选的，只应填充 expression 中使用的表达式属性值占位符的替换内容。

filter

另一个筛选条件，该筛选条件可用于在从 DynamoDB 返回结果之前先筛选结果。有关筛选条件的更多信息，请参阅筛选条件。该字段是可选的。

index

要查询的索引的名称。除了哈希键的主键索引以外，您还可以通过 DynamoDB 查询操作扫描本地二级索引和全局二级索引。如果指定，这会指示 DynamoDB 查询指定的索引。如果省略，则将查询主键索引。

nextToken

继续之前查询的分页标记。这应已从之前查询中获得。该字段是可选的。

limit

要评估的最大项目数（不一定是匹配项目数）。该字段是可选的。

scanIndexForward

一个布尔值，指示是向前还是向后查询。该字段是可选的，默认值为 true。

consistentRead

一个布尔值，用于指示在查询 DynamoDB 时是否使用一致性读取。该字段是可选的，默认值为 `false`。

select

默认情况下，AWS AppSync DynamoDB 解析器仅返回投影到索引的属性。如果需要更多属性，则可以设置该字段。该字段是可选的。支持的值为：

ALL_ATTRIBUTES

返回指定的表或索引中的所有项目属性。如果您查询本地二级索引，则 DynamoDB 从父表中为索引中的每个匹配项目获取整个项目。如果索引配置为投影所有项目属性，则可以从本地二级索引中获得所有数据，而不要求提取。

ALL_PROJECTED_ATTRIBUTES

仅在查询索引时才允许。检索已投影到索引的所有属性。如果索引配置为投影所有属性，则此返回值等同于指定 `ALL_ATTRIBUTES`。

SPECIFIC_ATTRIBUTES

仅返回 `projection` 的 `expression` 中列出的属性。该返回值相当于指定 `projection` 的 `Select`，而不指定 `expression` 的任何值。

projection

用于指定从 DynamoDB 操作返回的属性的投影。有关投影的更多信息，请参阅投影。该字段是可选的。

来自 DynamoDB 的结果将自动转换为 GraphQL 和 JSON 基元类型，并且可以在映射上下文 (`$context.result`) 中使用。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关响应映射模板的更多信息，请参阅解析器映射模板概述。

结果的结构如下所示：

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
```

```
}
```

字段定义如下：

`items`

包含 DynamoDB 查询返回的项目的列表。

`nextToken`

如果可能有更多结果，则 `nextToken` 包含一个分页标记，您可以在另一个请求中使用该标记。请注意，AWS AppSync 对从 DynamoDB 返回的分页标记进行加密和模糊处理。这可防止您的表数据无意中泄露给调用方。另请注意，无法跨不同解析器使用这些分页标记。

`scannedCount`

在应用筛选表达式（如果有）之前与查询条件表达式匹配的项目的数量。

示例

以下示例是 GraphQL 查询 `getPosts(owner: ID!)` 的映射模板。

在本示例中，将对表的全局二级索引执行查询，以返回指定的 ID 拥有的所有文章。

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "query" : {
        "expression" : "ownerId = :ownerId",
        "expressionValues" : {
            ":ownerId" : $util.dynamodb.toDynamoDBJson($context.arguments.owner)
        }
    },
    "index" : "owner-index"
}
```

有关 DynamoDB `Query` API 的更多信息，请参阅 DynamoDB API 文档。

Scan

通过使用 `Scan` 请求映射文档，您可以指示 AWS AppSync DynamoDB 解析器向 DynamoDB 发出 `Scan` 请求，并允许您指定以下内容：

- 排除结果的筛选条件
- 要使用的索引
- 要返回多少个项目
- 是否使用一致性读取
- 分页标记
- 并行扫描

Scan 映射文档具有以下结构：

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "index" : "fooIndex",
    "limit" : 10,
    "consistentRead" : false,
    "nextToken" : "aPaginationToken",
    "totalSegments" : 10,
    "segment" : 1,
    "filter" : {
        ...
    },
    "projection" : {
        ...
    }
}
```

字段定义如下：

Scan 字段

Scan 字段列表

version

模板定义版本。目前支持 2017-02-28 和 2018-05-29。该值为必填项。

operation

要执行的 DynamoDB 操作。要执行 Scan DynamoDB 操作，该字段必须设置为 Scan。该值为必填项。

filter

一个筛选条件，可用于在返回来自 DynamoDB 的结果之前对其进行筛选。有关筛选条件的更多信息，请参阅筛选条件。该字段是可选的。

index

要查询的索引的名称。除了哈希键的主键索引以外，您还可以通过 DynamoDB 查询操作扫描本地二级索引和全局二级索引。如果指定，这会指示 DynamoDB 查询指定的索引。如果省略，则将查询主键索引。

limit

单次评估的最大项目数。该字段是可选的。

consistentRead

一个布尔值，用于指示在查询 DynamoDB 时是否使用一致性读取。该字段是可选的，默认值为 `false`。

nextToken

继续之前查询的分页标记。这应已从之前查询中获得。该字段是可选的。

select

默认情况下，AWS AppSync DynamoDB 解析器仅返回投影到索引的任何属性。如果需要更多属性，则可以设置该字段。该字段是可选的。支持的值为：

ALL_ATTRIBUTES

返回指定的表或索引中的所有项目属性。如果您查询本地二级索引，则 DynamoDB 从父表中为索引中的每个匹配项目获取整个项目。如果索引配置为投影所有项目属性，则可以从本地二级索引中获得所有数据，而不要求提取。

ALL_PROJECTED_ATTRIBUTES

仅在查询索引时才允许。检索已投影到索引的所有属性。如果索引配置为投影所有属性，则此返回值等同于指定 `ALL_ATTRIBUTES`。

SPECIFIC_ATTRIBUTES

仅返回 `projection` 的 `expression` 中列出的属性。该返回值相当于指定 `projection` 的 `Select`，而不指定 `expression` 的任何值。

totalSegments

执行并行扫描时对表进行分区的分段数。该字段是可选的，但如果指定 segment，则必须指定该字段。

segment

执行并行扫描时，此操作中的表分段。该字段是可选的，但如果指定 totalSegments，则必须指定该字段。

projection

用于指定从 DynamoDB 操作返回的属性的投影。有关投影的更多信息，请参阅投影。该字段是可选的。

DynamoDB 扫描返回的结果将自动转换为 GraphQL 和 JSON 基元类型，并且可以在映射上下文 ($context.result) 中使用。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关响应映射模板的更多信息，请参阅解析器映射模板概述。

结果的结构如下所示：

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

字段定义如下：

items

包含 DynamoDB 扫描返回的项目的列表。

nextToken

如果可能有更多结果，则 nextToken 包含一个分页标记，您可以在另一个请求中使用该标记。AWSAppSync 对从 DynamoDB 返回的分页标记进行加密和模糊处理。这可防止您的表数据无意中泄露给调用方。另外，无法跨不同的解析器使用这些分页标记。

scannedCount

在应用筛选条件表达式（如果有）之前 DynamoDB 检索的项目数。

示例 1

以下示例是 GraphQL 查询 `allPosts` 的映射模板。

在本示例中，将返回表中的所有条目。

```
{
    "version" : "2017-02-28",
    "operation" : "Scan"
}
```

示例 2

以下示例是 GraphQL 查询 `postsMatching(title: String!)` 的映射模板。

在本示例中，将返回表中标题以 `title` 参数开头的所有条目。

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "filter" : {
        "expression" : "begins_with(title, :title)",
        "expressionValues" : {
            ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
        },
    }
}
```

有关 DynamoDB `Scan` API 的更多信息，请参阅 DynamoDB API 文档。

Sync（同步）

通过使用 Sync 请求映射文档，您可以从 DynamoDB 表中检索所有结果，然后仅接收自上次查询以来更改的数据（增量更新）。只能对版本控制的 DynamoDB 数据源发出 Sync 请求。您可以指定：

- 排除结果的筛选条件
- 要返回多少个项目

- 分页标记
- 上次 Sync 操作开始时间

Sync 映射文档具有以下结构：

```
{
    "version" : "2018-05-29",
    "operation" : "Sync",
    "basePartitionKey": "Base Tables PartitionKey",
    "deltaIndexName": "delta-index-name",
    "limit" : 10,
    "nextToken" : "aPaginationToken",
    "lastSync" :  1550000000000,
    "filter" : {
        ...
    }
}
```

字段定义如下：

Sync 字段

Sync 字段列表

`version`

模板定义版本。当前仅支持 2018-05-29。该值为必填项。

`operation`

要执行的 DynamoDB 操作。要执行 Sync 操作，该字段必须设置为 Sync。该值为必填项。

`filter`

一个筛选条件，可用于在返回来自 DynamoDB 的结果之前对其进行筛选。有关筛选条件的更多信息，请参阅筛选条件。该字段是可选的。

`limit`

单次评估的最大项目数。该字段是可选的。如果省略，则默认限制将设置为 100 个项目。该字段的最大值为 1000 个项目。

`nextToken`

继续之前查询的分页标记。这应已从之前查询中获得。该字段是可选的。

lastSync

最后一次成功 Sync 操作开始的时刻（以纪元毫秒为单位）。如果指定，则仅返回 lastSync 之后更改的项目。该字段是可选的，只有在从初始 Sync 操作中检索所有页面后才能填充。如果省略，将返回基本 表中的结果，否则将返回增量 表中的结果。

basePartitionKey

执行 Sync 操作时使用的基 表的分区键。在表使用自定义分区键时，该字段允许执行 Sync 操作。此为可选字段。

deltaIndexName

用于 Sync 操作的索引。在表使用自定义分区键时，需要使用该索引才能对整个增量存储表启用 Sync 操作。Sync 操作是对 GSI（在 gsi_ds_pk 和 gsi_ds_sk 上创建）执行的。该字段是可选的。

DynamoDB 同步返回的结果将自动转换为 GraphQL 和 JSON 基元类型，并且可以在映射上下文 ($context.result) 中使用。

有关 DynamoDB 类型转换的更多信息，请参阅类型系统（响应映射）。

有关响应映射模板的更多信息，请参阅解析器映射模板概述。

结果的结构如下所示：

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10,
    startedAt = 1550000000000
}
```

字段定义如下：

items

包含同步操作返回的项目的列表。

nextToken

如果可能有更多结果，则 nextToken 包含一个分页标记，您可以在另一个请求中使用该标记。AWSAppSync 对从 DynamoDB 返回的分页标记进行加密和模糊处理。这可防止您的表数据无意中泄露给调用方。另外，无法跨不同的解析器使用这些分页标记。

scannedCount

在应用筛选条件表达式（如果有）之前 DynamoDB 检索的项目数。

startedAt

同步操作开始的时刻，以纪元毫秒为单位，您可以在本地存储该值并在其他请求中将其用作 `lastSync` 参数。如果请求中包含分页令牌，则该值将与请求针对第一页结果返回的值相同。

示例 1

以下示例是 GraphQL 查询 `syncPosts(nextToken: String, lastSync: AWSTimestamp)` 的映射模板。

在此示例中，如果省略 `lastSync`，则返回基表中的所有条目。如果提供了 `lastSync`，则只返回增量同步表中自 `lastSync` 以来发生更改的条目。

```
{
    "version" : "2018-05-29",
    "operation" : "Sync",
    "limit": 100,
    "nextToken": $util.toJson($util.defaultIfNull($ctx.args.nextToken, null)),
    "lastSync": $util.toJson($util.defaultIfNull($ctx.args.lastSync, null))
}
```

BatchGetItem

通过使用 `BatchGetItem` 请求映射文档，您可以指示 AWS AppSync DynamoDB 解析器对 DynamoDB 发出 `BatchGetItem` 请求以检索多个项目（可能位于多个表中）。对于此请求模板，您必须指定以下各项：

- 要从中检索项目的表名称
- 要从每个表中检索的项目的键

DynamoDB `BatchGetItem` 限制适用，并且无法提供任何条件表达式。

`BatchGetItem` 映射文档具有以下结构：

```
{
```

```
    "version" : "2018-05-29",
    "operation" : "BatchGetItem",
    "tables" : {
        "table1": {
           "keys": [
              ## Item to retrieve Key
              {
                   "foo" : ... typed value,
                   "bar" : ... typed value
              },
              ## Item2 to retrieve Key
              {
                   "foo" : ... typed value,
                   "bar" : ... typed value
              }
            ],
            "consistentRead": true|false,
            "projection" : {
                ...
            }
        },
        "table2": {
           "keys": [
              ## Item3 to retrieve Key
              {
                   "foo" : ... typed value,
                   "bar" : ... typed value
              },
              ## Item4 to retrieve Key
              {
                   "foo" : ... typed value,
                   "bar" : ... typed value
              }
            ],
            "consistentRead": true|false,
            "projection" : {
                ...
            }
        }
    }
}
```

字段定义如下：

BatchGetItem 字段

BatchGetItem 字段列表

`version`

模板定义版本。仅支持 `2018-05-29`。该值为必填项。

`operation`

要执行的 DynamoDB 操作。要执行 `BatchGetItem` DynamoDB 操作，该字段必须设置为 `BatchGetItem`。该值为必填项。

`tables`

要从中检索项目的 DynamoDB 表。该值是一个映射，其中表名称被指定为映射的键。必须提供至少一个表。该 `tables` 值为必填项。

`keys`

DynamoDB 键列表，表示要检索的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。

`consistentRead`

是否在执行 GetItem 操作时使用一致性读取。此值是可选的，默认为 false。

`projection`

用于指定从 DynamoDB 操作返回的属性的投影。有关投影的更多信息，请参阅投影。该字段是可选的。

要记住的事项：

- 如果尚未从表中检索某个项目，则 null 元素将显示在该表的数据块中。
- 根据在请求映射模板中提供调用结果的顺序，将按表对这些结果进行排序。
- `BatchGetItem` 中的每个 `Get` 命令都是原子性的，但可以部分处理一个批次。如果由于错误而部分处理一个批处理，则未处理的键将作为 unprocessedKeys 块内的调用结果的一部分返回。
- `BatchGetItem` 限制为 100 个键。

对于以下示例请求映射模板：

```
{
  "version": "2018-05-29",
  "operation": "BatchGetItem",
  "tables": {
    "authors": [
        {
          "author_id": {
            "S": "a1"
          }
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          }
        }
    ],
  }
}
```

$ctx.result 中可用的调用结果如下所示：

```
{
   "data": {
     "authors": [null],
     "posts": [
        # Was retrieved
        {
          "author_id": "a1",
          "post_id": "p2",
          "post_title": "title",
          "post_description": "description",
        }
     ]
   },
   "unprocessedKeys": {
     "authors": [
        # This item was not processed due to an error
        {
```

```
          "author_id": "a1"
        }
      ],
     "posts": []
   }
}
```

$ctx.error 包含有关该错误的详细信息。保证键数据、unprocessedKeys 和请求映射模板中提供的每个表键都出现在调用结果中。已删除的项目显示在数据块中。尚未处理的项目将在数据块中标记为 null 并置于 unprocessedKeys 块中。

有关更完整的示例，请按照此处适用于 AppSync 的 DynamoDB 批处理教程“教程：DynamoDB 批处理解析器”进行操作。

BatchDeleteItem

通过使用 BatchDeleteItem 请求映射文档，您可以指示 AWS AppSync DynamoDB 解析器对 DynamoDB 发出 BatchWriteItem 请求以删除多个项目（可能位于多个表中）。对于此请求模板，您必须指定以下各项：

- 要从中删除项目的表名称
- 要从每个表中删除的项目的键

DynamoDB BatchWriteItem 限制适用，并且无法提供任何条件表达式。

BatchDeleteItem 映射文档具有以下结构：

```
{
    "version" : "2018-05-29",
    "operation" : "BatchDeleteItem",
    "tables" : {
        "table1": [
        ## Item to delete Key
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        },
        ## Item2 to delete Key
        {
             "foo" : ... typed value,
```

```
            "bar" : ... typed value
        }],
        "table2": [
        ## Item3 to delete Key
        {
            "foo" : ... typed value,
            "bar" : ... typed value
        },
        ## Item4 to delete Key
        {
            "foo" : ... typed value,
            "bar" : ... typed value
        }],
    }
}
```

字段定义如下：

BatchDeleteItem 字段

BatchDeleteItem 字段列表

`version`

模板定义版本。仅支持 `2018-05-29`。该值为必填项。

`operation`

要执行的 DynamoDB 操作。要执行 `BatchDeleteItem` DynamoDB 操作，该字段必须设置为 `BatchDeleteItem`。该值为必填项。

`tables`

要从中删除项目的 DynamoDB 表。每个表是 DynamoDB 键列表，表示要删除的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。必须提供至少一个表。`tables` 值是必需的。

要记住的事项：

- 与 `DeleteItem` 操作相反，响应中不会返回完全删除的项目。只返回传递的键。
- 如果尚未从表中删除某个项目，null 元素将显示在该表的数据块中。

- 根据在请求映射模板中提供调用结果的顺序，将按表对这些结果进行排序。
- BatchDeleteItem 中的每个 Delete 命令都是原子性的。但是，可以部分处理一个批处理。如果由于错误而部分处理一个批处理，则未处理的键将作为 unprocessedKeys 块内的调用结果的一部分返回。
- BatchDeleteItem 限制为 25 个键。

对于以下示例请求映射模板：

```
{
  "version": "2018-05-29",
  "operation": "BatchDeleteItem",
  "tables": {
    "authors": [
        {
          "author_id": {
            "S": "a1"
          }
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          }
        }
    ],
  }
}
```

$ctx.result 中可用的调用结果如下所示：

```
{
   "data": {
     "authors": [null],
     "posts": [
        # Was deleted
        {
          "author_id": "a1",
```

```
          "post_id": "p2"
        }
      ]
    },
    "unprocessedKeys": {
      "authors": [
        # This key was not processed due to an error
        {
          "author_id": "a1"
        }
       ],
      "posts": []
    }
}
```

`$ctx.error` 包含有关该错误的详细信息。保证键数据、unprocessedKeys 和请求映射模板中提供的每个表键都出现在调用结果中。已删除的项目存在于数据块中。尚未处理的项目将在数据块中标记为 null 并置于 unprocessedKeys 块中。

有关更完整的示例，请按照此处适用于 AppSync 的 DynamoDB 批处理教程“教程：DynamoDB 批处理解析器”进行操作。

BatchPutItem

通过使用 `BatchPutItem` 请求映射文档，您可以指示 AWS AppSync DynamoDB 解析器对 DynamoDB 发出 `BatchWriteItem` 请求以放置多个项目（可能位于多个表中）。对于此请求模板，您必须指定以下各项：

- 要将项目放置在其中的表名称
- 要放置在每个表中的完整项目

DynamoDB `BatchWriteItem` 限制适用，并且无法提供任何条件表达式。

`BatchPutItem` 映射文档具有以下结构：

```
{
    "version" : "2018-05-29",
    "operation" : "BatchPutItem",
    "tables" : {
        "table1": [
        ## Item to put
```

```
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        },
        ## Item2 to put
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        }],
        "table2": [
        ## Item3 to put
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        },
        ## Item4 to put
        {
             "foo" : ... typed value,
             "bar" : ... typed value
        }],
    }
}
```

字段定义如下：

BatchPutItem 字段

BatchPutItem 字段列表

`version`

模板定义版本。仅支持 `2018-05-29`。该值为必填项。

`operation`

要执行的 DynamoDB 操作。要执行 `BatchPutItem` DynamoDB 操作，该字段必须设置为 `BatchPutItem`。该值为必填项。

`tables`

要在其中放置项目的 DynamoDB 表。每个表条目表示要为该特定表插入的 DynamoDB 项目列表。必须提供至少一个表。该值为必填项。

要记住的事项：

- 如果成功，响应中将返回完全插入的项目。
- 如果尚未向表中插入项目，null 元素将显示在该表的数据块中。
- 根据在请求映射模板中提供插入的项目的顺序，按表对这些项目进行排序。
- BatchPutItem 中的每个 Put 命令都是原子性的，但可以部分处理一个批次。如果由于错误而部分处理一个批处理，则未处理的键将作为 unprocessedKeys 块内的调用结果的一部分返回。
- BatchPutItem 限制为 25 个项目。

对于以下示例请求映射模板：

```
{
  "version": "2018-05-29",
  "operation": "BatchPutItem",
  "tables": {
    "authors": [
        {
          "author_id": {
            "S": "a1"
          },
          "author_name": {
            "S": "a1_name"
          }
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          },
          "post_title": {
            "S": "title"
          }
        }
    ],
  }
}
```

$ctx.result 中可用的调用结果如下所示：

```
{
   "data": {
     "authors": [
         null
     ],
     "posts": [
        # Was inserted
        {
          "author_id": "a1",
          "post_id": "p2",
          "post_title": "title"
        }
     ]
   },
   "unprocessedItems": {
     "authors": [
        # This item was not processed due to an error
        {
          "author_id": "a1",
          "author_name": "a1_name"
        }
      ],
     "posts": []
   }
}
```

`$ctx.error` 包含有关该错误的详细信息。保证键数据、unprocessedItems 和请求映射模板中提供的每个表键都出现在调用结果中。已插入的项目存在于数据块中。尚未处理的项目将在数据块中标记为 null 并置于 unprocessedItems 块中。

有关更完整的示例，请按照此处适用于 AppSync 的 DynamoDB 批处理教程“教程：DynamoDB 批处理解析器”进行操作。

TransactGetItems

通过使用 `TransactGetItems` 请求映射文档，您可以指示 AWS AppSync DynamoDB 解析器对 DynamoDB 发出 `TransactGetItems` 请求以检索多个项目（可能位于多个表中）。对于此请求模板，您必须指定以下各项：

- 从中检索项目的每个请求项目的表名称
- 要从每个表中检索的每个请求项的键

DynamoDB TransactGetItems 限制适用，并且无法提供任何条件表达式。

TransactGetItems 映射文档具有以下结构：

```
{
    "version": "2018-05-29",
    "operation": "TransactGetItems",
    "transactItems": [
       ## First request item
       {
           "table": "table1",
           "key": {
               "foo": ... typed value,
               "bar": ... typed value
           },
           "projection" : {
                ...
           }
       },
       ## Second request item
       {
           "table": "table2",
           "key": {
               "foo": ... typed value,
               "bar": ... typed value
           },
           "projection" : {
                ...
           }
       }
    ]
}
```

字段定义如下：

TransactGetItems 字段

TransactGetItems 字段列表

`version`

模板定义版本。仅支持 2018-05-29。该值为必填项。

operation

要执行的 DynamoDB 操作。要执行 TransactGetItems DynamoDB 操作，该字段必须设置为 TransactGetItems。该值为必填项。

transactItems

要包含的请求项目。该值是请求项目的数组。必须提供至少一个请求项目。该 transactItems 值为必填项。

table

要从中检索项目的 DynamoDB 表。该值是表名的字符串。该 table 值为必填项。

key

DynamoDB 键，表示要检索的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。

projection

用于指定从 DynamoDB 操作返回的属性的投影。有关投影的更多信息，请参阅投影。该字段是可选的。

要记住的事项：

- 如果事务成功，items 块中检索项目的顺序将与请求项目的顺序相同。
- 事务以全部或全不的方式执行。如果任何请求项目导致错误，则整个交易都不会执行，并返回错误详细信息。
- 无法检索的请求项目不是错误。相反，null 元素会出现在相应位置的 items 块中。
- 如果一个事务的错误是 TransactionCanceledException，则 cancellationReasons 块将被填充。cancellationReasons 块中取消原因的顺序将与请求项目的顺序相同。
- TransactGetItems 仅限于 25 个请求项目。

对于以下示例请求映射模板：

```
{
    "version": "2018-05-29",
    "operation": "TransactGetItems",
    "transactItems": [
       ## First request item
```

```
        {
            "table": "posts",
            "key": {
                "post_id": {
                  "S": "p1"
                }
            }
        },
        ## Second request item
        {
            "table": "authors",
            "key": {
                "author_id": {
                  "S": a1
                }
            }
        }
    ]
}
```

如果事务成功并且只检索第一个请求的项目，则 `$ctx.result` 中可用的调用结果如下所示：

```
{
    "items": [
        {
            // Attributes of the first requested item
            "post_id": "p1",
            "post_title": "title",
            "post_description": "description"
        },
        // Could not retrieve the second requested item
        null,
    ],
    "cancellationReasons": null
}
```

如果事务由于第一个请求项目引起的 TransactionCanceledException 而失败，则 `$ctx.result` 中可用的调用结果如下所示：

```
{
    "items": null,
    "cancellationReasons": [
        {
```

```
            "type":"Sample error type",
            "message":"Sample error message"
        },
        {
            "type":"None",
            "message":"None"
        }
    ]
}
```

`$ctx.error` 包含有关该错误的详细信息。键 items 和 cancellationReasons 保证出现在 `$ctx.result` 中。

有关更完整的示例，请按照此处适用于 AppSync 的 DynamoDB 事务教程“教程：DynamoDB 事务解析器”进行操作。

TransactWriteItems

通过使用 `TransactWriteItems` 请求映射文档，您可以指示 AWS AppSync DynamoDB 解析器对 DynamoDB 发出 `TransactWriteItems` 请求以写入多个项目（可能写入到多个表）。对于此请求模板，您必须指定以下各项：

- 每个请求项目的目标表名称
- 要执行的每个请求项目的操作。支持四种类型的操作：PutItem、UpdateItem、DeleteItem 和 ConditionCheck
- 要写入的每个请求项目的键

DynamoDB `TransactWriteItems` 限制适用。

`TransactWriteItems` 映射文档具有以下结构：

```
{
    "version": "2018-05-29",
    "operation": "TransactWriteItems",
    "transactItems": [
        {
            "table": "table1",
            "operation": "PutItem",
            "key": {
                "foo": ... typed value,
                "bar": ... typed value
```

```
            },
            "attributeValues": {
                "baz": ... typed value
            },
            "condition": {
                "expression": "someExpression",
                "expressionNames": {
                    "#foo": "foo"
                },
                "expressionValues": {
                    ":bar": ... typed value
                },
                "returnValuesOnConditionCheckFailure": true|false
            }
        },
        {
            "table":"table2",
            "operation": "UpdateItem",
            "key": {
                "foo": ... typed value,
                "bar": ... typed value
            },
            "update": {
                "expression": "someExpression",
                "expressionNames": {
                    "#foo": "foo"
                },
                "expressionValues": {
                    ":bar": ... typed value
                }
            },
            "condition": {
                "expression": "someExpression",
                "expressionNames": {
                    "#foo":"foo"
                },
                "expressionValues": {
                    ":bar": ... typed value
                },
                "returnValuesOnConditionCheckFailure": true|false
            }
        },
        {
            "table": "table3",
```

```
            "operation": "DeleteItem",
            "key":{
                "foo": ... typed value,
                "bar": ... typed value
            },
            "condition":{
                "expression": "someExpression",
                "expressionNames": {
                    "#foo": "foo"
                },
                "expressionValues": {
                    ":bar": ... typed value
                },
                "returnValuesOnConditionCheckFailure": true|false
            }
        },
        {
            "table": "table4",
            "operation": "ConditionCheck",
            "key":{
                "foo": ... typed value,
                "bar": ... typed value
            },
            "condition":{
                "expression": "someExpression",
                "expressionNames": {
                    "#foo": "foo"
                },
                "expressionValues": {
                    ":bar": ... typed value
                },
                "returnValuesOnConditionCheckFailure": true|false
            }
        }
    ]
}
```

TransactWriteItems 字段

TransactWriteItems 字段列表

字段定义如下：

`version`

模板定义版本。仅支持 `2018-05-29`。该值为必填项。

`operation`

要执行的 DynamoDB 操作。要执行 `TransactWriteItems` DynamoDB 操作，该字段必须设置为 `TransactWriteItems`。该值为必填项。

`transactItems`

要包含的请求项目。该值是请求项目的数组。必须提供至少一个请求项目。该 `transactItems` 值为必填项。

对于 `PutItem`，字段定义如下：

`table`

目标 DynamoDB 表。该值是表名的字符串。该 `table` 值为必填项。

`operation`

要执行的 DynamoDB 操作。要执行 `PutItem` DynamoDB 操作，该字段必须设置为 `PutItem`。该值为必填项。

`key`

DynamoDB 键，表示要放置的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

`attributeValues`

要放入 DynamoDB 中的项目的其余属性。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该字段是可选的。

`condition`

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。如果未指定条件，则 `PutItem` 请求将覆盖该项目的任何现有条目。您可以指定在状况检查失败时是否重新检索现有项目。有关事务条件的更多信息，请参阅事务条件表达式。该值为可选项。

对于 UpdateItem，字段定义如下：

table

要更新的 DynamoDB 表。该值是表名的字符串。该 table 值为必填项。

operation

要执行的 DynamoDB 操作。要执行 UpdateItem DynamoDB 操作，该字段必须设置为 UpdateItem。该值为必填项。

key

DynamoDB 键，表示要更新的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

update

update 部分用于指定一个更新表达式，以描述如何更新 DynamoDB 中的项目。有关如何编写更新表达式的更多信息，请参阅 DynamoDB UpdateExpressions 文档。此部分是必需的。

condition

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。如果未指定条件，则 UpdateItem 请求将更新现有条目，而不考虑其当前状态。您可以指定在状况检查失败时是否重新检索现有项目。有关事务条件的更多信息，请参阅事务条件表达式。该值为可选项。

对于 DeleteItem，字段定义如下：

table

要在其中删除项目的 DynamoDB 表。该值是表名的字符串。该 table 值为必填项。

operation

要执行的 DynamoDB 操作。要执行 DeleteItem DynamoDB 操作，该字段必须设置为 DeleteItem。该值为必填项。

key

DynamoDB 键，表示要删除的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

condition

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。如果未指定条件，则 DeleteItem 请求将删除项目，而不考虑其当前状态。您可以指定在状况检查失败时是否重新检索现有项目。有关事务条件的更多信息，请参阅事务条件表达式。该值为可选项。

对于 ConditionCheck，字段定义如下：

table

要在其中检查条件的 DynamoDB 表。该值是表名的字符串。该 table 值为必填项。

operation

要执行的 DynamoDB 操作。要执行 ConditionCheck DynamoDB 操作，该字段必须设置为 ConditionCheck。该值为必填项。

key

DynamoDB 键，表示要检查条件的项目的主键。DynamoDB 项目可能具有单个哈希键，也可能具有哈希键和排序键，具体取决于表结构。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。该值为必填项。

condition

根据 DynamoDB 中已有的对象状态，确定请求是否应成功的条件。您可以指定在状况检查失败时是否重新检索现有项目。有关事务条件的更多信息，请参阅事务条件表达式。该值为必填项。

要记住的事项：

- 如果成功，响应中只返回请求项目的键。键的顺序将与请求项目的顺序相同。
- 事务以全部或全不的方式执行。如果任何请求项目导致错误，则整个交易都不会执行，并返回错误详细信息。
- 不能有两个请求项目针对同一个项目。否则，它们将导致 TransactionCanceledException 错误。
- 如果一个事务的错误是 TransactionCanceledException，则 cancellationReasons 块将被填充。如果请求项目的条件检查失败且 您没有将 returnValuesOnConditionCheckFailure 指定为 false，则表中存在的项目将被检索并存储在 cancellationReasons 块的相应位置的 item 中。
- TransactWriteItems 仅限于 25 个请求项目。

对于以下示例请求映射模板：

```
{
    "version": "2018-05-29",
    "operation": "TransactWriteItems",
    "transactItems": [
       {
           "table": "posts",
           "operation": "PutItem",
           "key": {
               "post_id": {
                   "S": "p1"
               }
           },
           "attributeValues": {
               "post_title": {
                   "S": "New title"
               },
               "post_description": {
                   "S": "New description"
               }
           },
           "condition": {
               "expression": "post_title = :post_title",
               "expressionValues": {
                   ":post_title": {
                       "S": "Expected old title"
                   }
               }
           }
       },
       {
           "table":"authors",
           "operation": "UpdateItem",
           "key": {
               "author_id": {
                   "S": "a1"
               },
           },
           "update": {
               "expression": "SET author_name = :author_name",
               "expressionValues": {
                   ":author_name": {
                       "S": "New name"
```

```
                    }
                }
            },
        }
    ]
}
```

如果事务成功，$ctx.result 中可用的调用结果如下所示：

```
{
    "keys": [
       // Key of the PutItem request
       {
           "post_id": "p1",
       },
       // Key of the UpdateItem request
       {
           "author_id": "a1"
       }
    ],
    "cancellationReasons": null
}
```

如果由于 PutItem 请求的条件检查失败而导致事务失败，则 $ctx.result 中提供的调用结果如下所示：

```
{
    "keys": null,
    "cancellationReasons": [
       {
           "item": {
               "post_id": "p1",
               "post_title": "Actual old title",
               "post_description": "Old description"
           },
           "type": "ConditionCheckFailed",
           "message": "The condition check failed."
       },
       {
           "type": "None",
           "message": "None"
       }
    ]
```

```
}
```

`$ctx.error` 包含有关该错误的详细信息。键 keys 和 cancellationReasons 保证出现在 `$ctx.result` 中。

有关更完整的示例，请按照此处适用于 AppSync 的 DynamoDB 事务教程“教程：DynamoDB 事务解析器”进行操作。

类型系统（请求映射）

在使用 AWS AppSync DynamoDB 解析器调用 DynamoDB 表时，AWS AppSync 需要知道该调用中使用的每个值的类型。这是因为 DynamoDB 支持的基元类型比 GraphQL 或 JSON 多（例如集和二进制数据）。AWS在 GraphQL 和 DynamoDB 之间转换时，AppSync 需要一些提示，否则，它必须对如何在表中设置数据结构做出一些假设。

有关 DynamoDB 数据类型的更多信息，请参阅 DynamoDB 数据类型描述符和数据类型文档。

DynamoDB 值由包含单个键值对的 JSON 对象表示。键指定 DynamoDB 类型，值指定值本身。在下面的示例中，键 S 表示值是一个字符串，值 `identifier` 是字符串值本身。

```
{ "S" : "identifier" }
```

请注意，JSON 对象不能具有多于一个键值对。如果指定了多个键值对，将不会解析请求映射文档。

DynamoDB 值用于请求映射文档中您需要指定值的任何位置。您需要这样做的一些地方包括：`key` 和 `attributeValue` 部分以及表达式部分的 `expressionValues` 部分。在以下示例中，DynamoDB 字符串值 `identifier` 分配给 `key` 部分中的 `id` 字段（可能位于 `GetItem` 请求映射文档中）。

```
"key" : {
   "id" : { "S" : "identifier" }
}
```

支持的类型

AWS AppSync 支持以下 DynamoDB 标量、文档和集类型：

字符串类型 **S**

单个字符串值。DynamoDB 字符串值表示为：

```
{ "S" : "some string" }
```

示例用法如下：

```
"key" : {
   "id" : { "S" : "some string" }
}
```

字符串集类型 SS

一组字符串值。DynamoDB 字符串集值表示为：

```
{ "SS" : [ "first value", "second value", ... ] }
```

示例用法如下：

```
"attributeValues" : {
   "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
}
```

数字类型 N

单个数字值。DynamoDB 数字值表示为：

```
{ "N" : 1234 }
```

示例用法如下：

```
"expressionValues" : {
   ":expectedVersion" : { "N" : 1 }
}
```

数字集类型 NS

一组数字值。DynamoDB 数字集值表示为：

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

示例用法如下：

```
"attributeValues" : {
   "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }
}
```

二进制类型 B

二进制值。DynamoDB 二进制值表示为：

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

请注意，该值实际上是一个字符串，其中，字符串是二进制数据的 Base64 编码表示形式。AWSAppSync 将该字符串解码回二进制值，然后再将其发送到 DynamoDB。AWSAppSync 使用 RFC 2045 定义的 Base64 解码方案：忽略没有位于 Base64 字母表中的任何字符。

示例用法如下：

```
"attributeValues" : {
   "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

二进制集类型 BS

一组二进制值。DynamoDB 二进制集值表示为：

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

请注意，该值实际上是一个字符串，其中，字符串是二进制数据的 Base64 编码表示形式。AWSAppSync 将该字符串解码回二进制值，然后再将其发送到 DynamoDB。AWSAppSync 使用 RFC 2045 定义的 Base64 解码方案：忽略没有位于 Base64 字母表中的任何字符。

示例用法如下：

```
"attributeValues" : {
   "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }
}
```

布尔值类型 BOOL

布尔值。DynamoDB 布尔值表示为：

```
{ "BOOL" : true }
```

请注意，只有 `true` 和 `false` 是有效值。

示例用法如下：

```
"attributeValues" : {
   "orderComplete" : { "BOOL" : false }
}
```

列表类型 **L**

任何其他支持的 DynamoDB 值列表。DynamoDB 列表值表示为：

```
{ "L" : [ ... ] }
```

请注意，该值是一个复合值，其中，列表可以包含零个或更多任何支持的 DynamoDB 值（包括其他列表）。此列表还可以包含不同类型的混合。

示例用法如下：

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

映射类型 **M**

表示其他支持的 DynamoDB 值的键值对的无序集合。DynamoDB 映射值表示为：

```
{ "M" : { ... } }
```

请注意，一个映射可以包含零个或零个以上的键值对。键必须是一个字符串，值可以是任何支持的 DynamoDB 值（包括其他映射）。此映射还可以包含不同类型的混合。

示例用法如下：

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

Null 类型 **NULL**

Null 值。DynamoDB Null 值表示为：

```
{ "NULL" : null }
```

示例用法如下：

```
"attributeValues" : {
   "phoneNumbers" : { "NULL" : null }
}
```

有关每个类型的更多信息，请参阅 DynamoDB 文档。

类型系统（响应映射）

在从 DynamoDB 收到响应时，AWS AppSync 自动将其转换为 GraphQL 和 JSON 基元类型。将解码 DynamoDB 中的每个属性，并在响应映射上下文中返回。

例如，如果 DynamoDB 返回以下内容：

```
{
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

则 AWS AppSync DynamoDB 解析器将其转换为 GraphQL 和 JSON 类型，如下所示：

```
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

本节介绍了 AWS AppSync 如何转换以下 DynamoDB 标量、文档和集类型：

字符串类型 **S**

单个字符串值。DynamoDB 字符串值以字符串形式返回。

例如，如果 DynamoDB 返回以下 DynamoDB 字符串值：

```
{ "S" : "some string" }
```

AWS AppSync 将其转换为字符串：

```
"some string"
```

字符串集类型 **SS**

一组字符串值。DynamoDB 字符串集值以字符串列表形式返回。

例如，如果 DynamoDB 返回以下 DynamoDB 字符串集值：

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync 将其转换为字符串列表：

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

数字类型 **N**

单个数字值。DynamoDB 数字值以数字形式返回。

例如，如果 DynamoDB 返回以下 DynamoDB 数字值：

```
{ "N" : 1234 }
```

AWS AppSync 将其转换为数字：

```
1234
```

数字集类型 **NS**

一组数字值。DynamoDB 数字集值以数字列表形式返回。

例如，如果 DynamoDB 返回以下 DynamoDB 数字集值：

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync 将其转换为数字列表：

```
[ 67.8, 12.2, 70 ]
```

二进制类型 **B**

二进制值。DynamoDB 二进制值以字符串形式返回，其中包含该值的 Base64 表示形式。

例如，如果 DynamoDB 返回以下 DynamoDB 二进制值：

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync 将其转换为包含该值的 Base64 表示形式的字符串：

```
"SGVsbG8sIFdvcmxkIQo="
```

请注意，二进制数据按照 RFC 4648 和 RFC 2045 中指定的方式根据 base64 编码方案进行编码。

二进制集类型 **BS**

一组二进制值。DynamoDB 二进制集值以字符串列表形式返回，其中包含这些值的 Base64 表示形式。

例如，如果 DynamoDB 返回以下 DynamoDB 二进制集值：

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync 将其转换为包含这些值的 Base64 表示形式的字符串列表：

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

请注意，二进制数据按照 RFC 4648 和 RFC 2045 中指定的方式根据 base64 编码方案进行编码。

布尔值类型 **BOOL**

布尔值。DynamoDB 布尔值以布尔值形式返回。

例如，如果 DynamoDB 返回以下 DynamoDB 布尔值：

```
{ "BOOL" : true }
```

AWS AppSync 将其转换为布尔值：

```
true
```

列表类型 **L**

任何其他支持的 DynamoDB 值列表。DynamoDB 列表值以值列表形式返回，其中还会转换每个内部值。

例如，如果 DynamoDB 返回以下 DynamoDB 列表值：

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

AWS AppSync 将其转换为转换的值列表：

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

映射类型 **M**

任何其他支持的 DynamoDB 值的键/值集合。DynamoDB 映射值以 JSON 对象形式返回，其中还会转换每个键/值。

例如，如果 DynamoDB 返回以下 DynamoDB 映射值：

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

AWS AppSync 将其转换为 JSON 对象：

```
{
   "someString" : "A string value",
   "someNumber" : 1,
   "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

Null 类型 **NULL**

Null 值。

例如，如果 DynamoDB 返回以下 DynamoDB Null 值：

```
{ "NULL" : null }
```

AWS AppSync 将其转换为 Null：

```
null
```

Filters

在使用 Query 和 Scan 操作查询 DynamoDB 中的对象时，您可以选择指定 filter 以评估结果并仅返回所需的值。

Query 或 Scan 映射文档的筛选映射部分具有以下结构：

```
"filter" : {
    "expression" : "filter expression"
    "expressionNames" : {
        "#name" : "name",
    },
    "expressionValues" : {
        ":value" : ... typed value
    },
}
```

字段定义如下：

expression

查询表达式。有关如何编写筛选条件表达式的更多信息，请参阅 DynamoDB QueryFilter 和 DynamoDB ScanFilter 文档。必须指定该字段。

expressionNames

以键值对形式替换表达式属性名称 占位符。键对应于 expression 中使用的名称占位符。该值必须是与 DynamoDB 中的项目的属性名称对应的字符串。该字段是可选的，只应填充 expression 中使用的表达式属性名称占位符的替换内容。

expressionValues

以键值对形式替换表达式属性值 占位符。键对应于 expression 中使用的值占位符，而值必须为类型化值。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。必须指定此值。该字段是可选的，只应填充 expression 中使用的表达式属性值占位符的替换内容。

示例

以下示例是映射模板的筛选条件部分，其中，只有在标题以 title 参数开头时，才会返回从 DynamoDB 中检索的条目。

```
"filter" : {
    "expression" : "begins_with(#title, :title)",
    "expressionNames" : {
        "#title" : "title"
    },
    "expressionValues" : {
        ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
    }
}
```

条件表达式

在您使用 PutItem、UpdateItem 和 DeleteItem DynamoDB 操作变更 DynamoDB 中的对象时，您可以选择指定一个条件表达式，以根据执行操作之前 DynamoDB 中的已有对象状态控制请求是否应成功。

AWS AppSync DynamoDB 解析器允许在 PutItem、UpdateItem 和 DeleteItem 请求映射文档中指定条件表达式，并提供在条件失败并且未更新对象时遵循的策略。

示例 1

以下 PutItem 映射文档没有条件表达式。因此，即使已存在具有相同键的项目，它也会将项目放置在 DynamoDB 中，从而覆盖现有的项目。

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
```

```
    }
}
```

示例 2

以下 PutItem 映射文档确实具有一个条件表达式，只有在 DynamoDB 中不 存在具有相同键的项目时，该操作才会成功。

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   },
   "condition" : {
      "expression" : "attribute_not_exists(id)"
   }
}
```

默认情况下，如果条件检查失败，则 AWS AppSync DynamoDB 解析器返回变更错误，并在 GraphQL 响应 error 部分的 data 字段中返回 DynamoDB 中的对象的当前值。不过，AWS AppSync DynamoDB 解析器提供了一些额外的功能，以帮助开发人员处理一些常见的边缘情况：

- 如果 AWS AppSync DynamoDB 解析器可以确定 DynamoDB 中的当前值与所需的结果匹配，则会将该操作视为成功。
- 您可以将解析器配置为调用自定义 Lambda 函数以决定 AWS AppSync DynamoDB 解析器应如何处理失败，而不是返回错误。

在处理条件检查失败一节中更详细地介绍了这些内容。

有关 DynamoDB 条件表达式的更多信息，请参阅 DynamoDB 条件表达式文档。

指定条件

PutItem、UpdateItem 和 DeleteItem 请求映射文档都允许指定可选的 condition 部分。如果省略，则不会进行条件检查。如果指定，条件必须为 true，操作才能成功。

condition 部分具有以下结构：

```
"condition" : {
```

```
    "expression" : "someExpression"
    "expressionNames" : {
        "#foo" : "foo"
    },
    "expressionValues" : {
        ":bar" : ... typed value
    },
    "equalsIgnore" : [ "version" ],
    "consistentRead" : true,
    "conditionalCheckFailedHandler" : {
        "strategy" : "Custom",
        "lambdaArn" : "arn:..."
    }
}
```

下列字段指定条件：

`expression`

更新表达式本身。有关如何编写条件表达式的更多信息，请参阅 DynamoDB ConditionExpressions 文档。必须指定该字段。

`expressionNames`

以键值对形式替换表达式属性名称占位符。键对应于 expression 中使用的名称占位符，值必须是与 DynamoDB 中的项目的属性名称对应的字符串。该字段是可选的，只应填充 expression 中使用的表达式属性名称占位符的替换内容。

`expressionValues`

以键值对形式替换表达式属性值占位符。键对应于 expression 中使用的值占位符，而值必须为类型化值。有关如何指定“类型化值”的更多信息，请参阅类型系统（请求映射）。必须指定此值。该字段是可选的，只应填充 expression 中使用的表达式属性值占位符的替换内容。

其余字段指示 AWS AppSync DynamoDB 解析器如何处理条件检查失败：

`equalsIgnore`

如果在使用 `PutItem` 操作时条件检查失败，则 AWS AppSync DynamoDB 解析器将当前位于 DynamoDB 中的项目与它尝试写入的项目进行比较。如果两者相同，则它会将操作视为已成功。您可以使用 `equalsIgnore` 字段指定 AWS AppSync 在执行该比较时应忽略的属性列表。例如，如果唯一的区别是 `version` 属性，则会将操作视为成功。该字段是可选的。

consistentRead

在条件检查失败时，AWS AppSync 使用强一致性读取从 DynamoDB 中获取项目的当前值。您可以使用该字段指示 AWS AppSync DynamoDB 解析器改用最终一致性读取。该字段是可选的，默认值为 `true`。

conditionalCheckFailedHandler

在该部分中，您可以指定 AWS AppSync DynamoDB 解析器在将 DynamoDB 中的当前值与预期结果进行比较后如何处理条件检查失败。此部分是可选的。如果省略，它默认为 `Reject` 策略。

strategy

AWS AppSync DynamoDB 解析器在将 DynamoDB 中的当前值与预期结果进行比较后采取的策略。该字段为必填字段，有以下可能的值：

Reject

变更失败，将返回变更错误，并在 GraphQL 响应 `error` 部分的 `data` 字段中返回 DynamoDB 中的对象的当前值。

Custom

AWS AppSync DynamoDB 解析器调用自定义 Lambda 函数，以决定如何处理条件检查失败。当 `strategy` 设置为 `Custom` 时，`lambdaArn` 字段必须包含要调用的 Lambda 函数的 ARN。

lambdaArn

要调用的 Lambda 函数的 ARN，用于确定 AWS AppSync DynamoDB 解析器应如何处理条件检查失败。当 `strategy` 设置为 `Custom` 时，必须指定该字段。有关如何使用该功能的更多信息，请参阅处理条件检查失败。

处理条件检查失败

默认情况下，如果条件检查失败，则 AWS AppSync DynamoDB 解析器返回变更错误，并在 GraphQL 响应 `error` 部分的 `data` 字段中返回 DynamoDB 中的对象的当前值。不过，AWS AppSync DynamoDB 解析器提供了一些额外的功能，以帮助开发人员处理一些常见的边缘情况：

- 如果 AWS AppSync DynamoDB 解析器可以确定 DynamoDB 中的当前值与所需的结果匹配，则会将该操作视为成功。
- 您可以将解析器配置为调用自定义 Lambda 函数以决定 AWS AppSync DynamoDB 解析器应如何处理失败，而不是返回错误。

此过程的流程图为：

检查所需的结果

在条件检查失败时，AWS AppSync DynamoDB 解析器执行 `GetItem` DynamoDB 请求，以从 DynamoDB 中获取项目的当前值。默认情况下，它将使用强一致性读取，但这可以使用 `condition` 数据块中的 `consistentRead` 字段进行配置，并将当前值与预期结果进行比较：

- 对于 `PutItem` 操作，AWS AppSync DynamoDB 解析器将当前值与它尝试写入的值进行比较，以从比较中排除 `equalsIgnore` 中列出的任何属性。如果项目相同，则将操作视为成功并返回从 DynamoDB 中检索的项目。否则，它将遵循所配置的策略。

  例如，如果 `PutItem` 请求映射文档如下所示：

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   },
   "attributeValues" : {
      "name" : { "S" : "Steve" },
      "version" : { "N" : 2 }
   },
   "condition" : {
      "expression" : "version = :expectedVersion",
      "expressionValues" : {
          ":expectedVersion" : { "N" : 1 }
      },
      "equalsIgnore": [ "version" ]
   }
}
```

  当前位于 DynamoDB 中的项目如下所示：

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

AWS AppSync DynamoDB 解析器将它尝试写入的项目与当前值进行比较，发现唯一的区别是 `version` 字段，但由于它配置为忽略 `version` 字段，因此，将操作视为成功并返回从 DynamoDB 中检索的项目。

- 对于 `DeleteItem` 操作，AWS AppSync DynamoDB 解析器检查以验证是否从 DynamoDB 返回了项目。如果没有返回项目，它会将该操作视为已成功。否则，它将遵循所配置的策略。
- 对于 `UpdateItem` 操作，AWS AppSync DynamoDB 解析器没有足够的信息，无法确定当前位于 DynamoDB 中的项目是否与预期结果匹配，因此，将遵循配置的策略。

如果 DynamoDB 中的对象的当前状态与预期结果不同，则 AWS AppSync DynamoDB 解析器按照配置的策略拒绝变更或调用 Lambda 函数以确定后续操作。

遵循“reject”策略

在遵循 `Reject` 策略时，AWS AppSync DynamoDB 解析器返回变更错误，并且还会在 GraphQL 响应 `error` 部分的 `data` 字段中返回 DynamoDB 中的对象的当前值。从 DynamoDB 返回的项目通过响应映射模板转换为客户端的预期格式，并按选择集进行筛选。

例如，给定以下变更请求：

```
mutation {
    updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
        Name
        theVersion
    }
}
```

如果从 DynamoDB 返回的项目如下所示：

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

并且响应映射模板如下所示：

```
{
   "id" : $util.toJson($context.result.id),
   "Name" : $util.toJson($context.result.name),
```

```
    "theVersion" : $util.toJson($context.result.version)
}
```

GraphQL 响应如下所示：

```
{
  "data": null,
  "errors": [
    {
      "message": "The conditional request failed (Service: AmazonDynamoDBv2;
 Status Code: 400; Error Code: ConditionalCheckFailedException; Request ID:
 ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ)"
      "errorType": "DynamoDB:ConditionalCheckFailedException",
      "data": {
        "Name": "Steve",
        "theVersion": 8
      },
      ...
    }
  ]
}
```

另外，如果返回的对象中的任何字段在变更成功后已由其他解析器填充，则当在 `error` 部分中返回此对象时，将不会对这些字段进行解析。

遵循“custom”策略

在遵循 `Custom` 策略时，AWS AppSync DynamoDB 解析器调用 Lambda 函数以决定后续操作。Lambda 函数选择下列选项之一：

- `reject` 变更。这会指示 AWS AppSync DynamoDB 解析器的行为就像配置的策略是 `Reject` 一样，返回变更错误和 DynamoDB 中的对象的当前值，如上一节所述。
- `discard` 变更。这会指示 AWS AppSync DynamoDB 解析器静默忽略条件检查失败并返回 DynamoDB 中的值。
- `retry` 变更。这会指示 AWS AppSync DynamoDB 解析器使用新的请求映射文档重试变更。

Lambda 调用请求

AWS AppSync DynamoDB 解析器调用 `lambdaArn` 中指定的 Lambda 函数。它将使用在数据源上配置的相同 `service-role-arn`。调用的负载具有以下结构：

```
{
    "arguments": { ... },
    "requestMapping": {... },
    "currentValue": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

字段定义如下：

arguments

来自 GraphQL 变更的参数。这与可用于 $context.arguments 中的请求映射文档的参数相同。

requestMapping

用于此操作的请求映射文档。

currentValue

DynamoDB 中的对象的当前值。

resolver

有关 AWS AppSync 解析器的信息。

identity

有关调用方的信息。这与可用于 $context.identity 中的请求映射文档的身份信息相同。

负载的完整示例：

```
{
    "arguments": {
        "id": "1",
        "name": "Steve",
        "expectedVersion": 1
    },
    "requestMapping": {
        "version" : "2017-02-28",
        "operation" : "PutItem",
        "key" : {
           "id" : { "S" : "1" }
```

```
        },
        "attributeValues" : {
           "name" : { "S" : "Steve" },
           "version" : { "N" : 2 }
        },
        "condition" : {
           "expression" : "version = :expectedVersion",
           "expressionValues" : {
               ":expectedVersion" : { "N" : 1 }
           },
           "equalsIgnore": [ "version" ]
        }
    },
    "currentValue": {
        "id" : { "S" : "1" },
        "name" : { "S" : "Steve" },
        "version" : { "N" : 8 }
    },
    "resolver": {
        "tableName": "People",
        "awsRegion": "us-west-2",
        "parentType": "Mutation",
        "field": "updatePerson",
        "outputType": "Person"
    },
    "identity": {
        "accountId": "123456789012",
        "sourceIp": "x.x.x.x",
        "user": "AIDAAAAAAAAAAAAAAAAA",
        "userArn": "arn:aws:iam::123456789012:user/appsync"
    }
}
```

Lambda 调用响应

Lambda 函数可以检查调用负载，并应用任何业务逻辑以决定 AWS AppSync DynamoDB 解析器应如何处理失败。有三种选项可用于处理条件检查失败：

- `reject` 变更。此选项的响应负载必须具有此结构：

```
{
    "action": "reject"
}
```

这会指示 AWS AppSync DynamoDB 解析器的行为就像配置的策略是 `Reject` 一样，返回变更错误和 DynamoDB 中的对象的当前值，如上一节所述。

- `discard` 变更。此选项的响应负载必须具有此结构：

```
{
    "action": "discard"
}
```

这会指示 AWS AppSync DynamoDB 解析器静默忽略条件检查失败并返回 DynamoDB 中的值。

- `retry` 变更。此选项的响应负载必须具有此结构：

```
{
    "action": "retry",
    "retryMapping": { ... }
}
```

这会指示 AWS AppSync DynamoDB 解析器使用新的请求映射文档重试变更。`retryMapping` 部分的结构取决于 DynamoDB 操作，并且是该操作的完整请求映射文档的子集。

对于 `PutItem`，`retryMapping` 部分具有以下结构。有关 `attributeValues` 字段的描述，请参阅 PutItem。

```
{
    "attributeValues": { ... },
    "condition": {
        "equalsIgnore" = [ ... ],
        "consistentRead" = true
    }
}
```

对于 `UpdateItem`，`retryMapping` 部分具有以下结构。有关 `update` 部分的说明，请参阅 UpdateItem。

```
{
    "update" : {
        "expression" : "someExpression"
        "expressionNames" : {
            "#foo" : "foo"
        },
```

```
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "condition": {
        "consistentRead" = true
    }
}
```

对于 DeleteItem，retryMapping 部分具有以下结构。

```
{
    "condition": {
        "consistentRead" = true
    }
}
```

无法指定要使用的不同的操作或键。AWS AppSync DynamoDB 解析器仅允许对同一对象重试相同操作。另外，condition 部分不允许指定 conditionalCheckFailedHandler。如果重试失败，AWS AppSync DynamoDB 解析器将遵循 Reject 策略。

在下面的示例中，Lambda 函数处理失败的 PutItem 请求。业务逻辑将查看进行调用的用户。如果调用是由 jeffTheAdmin 进行的，则会重试该请求，并从当前位于 DynamoDB 的项目中更新 version 和 expectedVersion。否则，业务逻辑将拒绝变更。

```
exports.handler = (event, context, callback) => {
    console.log("Event: "+ JSON.stringify(event));

    // Business logic goes here.

    var response;
    if ( event.identity.user == "jeffTheAdmin" ) {
        response = {
            "action" : "retry",
            "retryMapping" : {
                "attributeValues" : event.requestMapping.attributeValues,
                "condition" : {
                    "expression" : event.requestMapping.condition.expression,
                    "expressionValues" :
 event.requestMapping.condition.expressionValues
                }
```

```
            }
        }
        response.retryMapping.attributeValues.version = { "N" :
 event.currentValue.version.N + 1 }
        response.retryMapping.condition.expressionValues[':expectedVersion'] =
 event.currentValue.version

    } else {
        response = { "action" : "reject" }
    }

    console.log("Response: "+ JSON.stringify(response))
    callback(null, response)
};
```

事务条件表达式

事务条件表达式可用于 TransactWriteItems 中所有四种类型的操作的请求映射模板，即 PutItem、DeleteItem、UpdateItem 和 ConditionCheck。

对于 PutItem、DeleteItem 和 UpdateItem，事务条件表达式是可选的。对于 ConditionCheck，事务条件表达式是必需的。

示例 1

以下事务 DeleteItem 映射文档没有条件表达式。因此，它删除 DynamoDB 中的项目。

```
{
   "version": "2018-05-29",
   "operation": "TransactWriteItems",
   "transactItems": [
      {
         "table": "posts",
         "operation": "DeleteItem",
         "key": {
            "id": { "S" : "1" }
         }
      }
   ]
}
```

示例 2

以下事务 DeleteItem 映射文档确实具有一个事务条件表达式，只有在该文章的作者等于特定姓名时，操作才会成功。

```
{
   "version": "2018-05-29",
   "operation": "TransactWriteItems",
   "transactItems": [
      {
         "table": "posts",
         "operation": "DeleteItem",
         "key": {
            "id": { "S" : "1" }
         }
         "condition": {
            "expression": "author = :author",
            "expressionValues": {
               ":author": { "S" : "Chunyan" }
            }
         }
      }
   ]
}
```

如果条件检查失败，则会导致 TransactionCanceledException，错误详细信息将在 $ctx.result.cancellationReasons 中返回。请注意，默认情况下，DynamoDB 中导致条件检查失败的旧项目将在 $ctx.result.cancellationReasons 中返回。

指定条件

PutItem、UpdateItem 和 DeleteItem 请求映射文档都允许指定可选的 condition 部分。如果省略，则不会进行条件检查。如果指定，条件必须为 true，操作才能成功。ConditionCheck 必须具有要指定的 condition 部分。条件必须为 true，整个事务才能成功。

condition 部分具有以下结构：

```
"condition": {
    "expression": "someExpression",
    "expressionNames": {
        "#foo": "foo"
```

```
    },
    "expressionValues": {
        ":bar": ... typed value
    },
    "returnValuesOnConditionCheckFailure": false
}
```

下列字段指定条件：

expression

更新表达式本身。有关如何编写条件表达式的更多信息，请参阅 DynamoDB ConditionExpressions 文档。必须指定该字段。

expressionNames

以键值对形式替换表达式属性名称占位符。键对应于 expression 中使用的名称占位符，值必须是与 DynamoDB 中的项目的属性名称对应的字符串。该字段是可选的，只应填充 expression 中使用的表达式属性名称占位符的替换内容。

expressionValues

以键值对形式替换表达式属性值占位符。键对应于 expression 中使用的值占位符，而值必须为类型化值。有关如何指定“类型化值”的更多信息，请参阅“类型系统（请求映射）”。必须指定此值。该字段是可选的，只应填充 expression 中使用的表达式属性值占位符的替换内容。

returnValuesOnConditionCheckFailure

指定在条件检查失败时是否重新检索 DynamoDB 中的项目。检索到的项目将位于 `$ctx.result.cancellationReasons[$index].item` 中，其中 `$index` 是未通过条件检查的请求项目的索引。该值默认为 true。

投影

在使用 `GetItem`、`Scan`、`Query`、`BatchGetItem` 和 `TransactGetItems` 操作读取 DynamoDB 中的对象时，您可以选择指定一个投影以指定所需的属性。投影具有以下结构，与筛选条件类似：

```
"projection" : {
    "expression" : "projection expression"
    "expressionNames" : {
        "#name" : "name",
```

```
    }
}
```

字段定义如下：

`expression`

投影表达式，它是一个字符串。要检索单个属性，请指定其名称。对于多个属性，名称必须是逗号分隔值。有关编写投影表达式的更多信息，请参阅 DynamoDB 投影表达式文档。该字段为必填。

`expressionNames`

以键值对形式替换表达式属性名称占位符。键对应于 `expression` 中使用的名称占位符。该值必须是与 DynamoDB 中的项目的属性名称对应的字符串。该字段是可选的，只应填充 `expression` 中使用的表达式属性名称占位符的替换内容。有关 `expressionNames` 的更多信息，请参阅 DynamoDB 文档。

示例 1

以下示例是 VTL 映射模板的投影部分，其中仅从 DynamoDB 返回 `author` 和 `id` 属性：

```
"projection" : {
    "expression" : "#author, id",
    "expressionNames" : {
        "#author" : "author"
    }
}
```

> **Tip**
>
> 您可以使用 $context.info.selectionSetList 访问 GraphQL 请求选择集。可以通过该字段根据您的要求动态构建投影表达式。

> **Note**
>
> 在将投影表达式与 `Query` 和 `Scan` 运算一起使用时，`select` 的值必须为 `SPECIFIC_ATTRIBUTES`。有关更多信息，请参阅 DynamoDB 文档。

RDS 的解析器映射模板参考

通过使用 AWS AppSync RDS 解析器映射模板，开发人员可以将 SQL 查询发送到 Amazon Aurora Serverless 数据 API，并获取这些查询的结果。

请求映射模板

RDS 请求映射模板相当简单：

```
{
    "version": "2018-05-29",
    "statements": [],
    "variableMap": {},
    "variableTypeHintMap": {}
}
```

下面是 RDS 请求映射模板的 JSON 架构表示形式（解析后）：

```
{
    "definitions": {},
    "$schema": "https://json-schema.org/draft-07/schema#",
    "$id": "https://example.com/root.json",
    "type": "object",
    "title": "The Root Schema",
    "required": [
        "version",
        "statements",
        "variableMap"
    ],
    "properties": {
        "version": {
            "$id": "#/properties/version",
            "type": "string",
            "title": "The Version Schema",
            "default": "",
            "examples": [
                "2018-05-29"
            ],
            "enum": [
                "2018-05-29"
            ],
            "pattern": "^(.*)$"
```

```
        },
        "statements": {
            "$id": "#/properties/statements",
            "type": "array",
            "title": "The Statements Schema",
            "items": {
                "$id": "#/properties/statements/items",
                "type": "string",
                "title": "The Items Schema",
                "default": "",
                "examples": [
                    "SELECT * from BOOKS"
                ],
                "pattern": "^(.*)$"
            }
        },
        "variableMap": {
            "$id": "#/properties/variableMap",
            "type": "object",
            "title": "The Variablemap Schema"
        },
        "variableTypeHintMap": {
            "$id": "#/properties/variableTypeHintMap",
            "type": "object",
            "title": "The variableTypeHintMap Schema"
        }
    }
}
```

以下是一个具有静态查询的请求映射模板示例：

```
{
    "version": "2018-05-29",
    "statements": [
        "select title, isbn13 from BOOKS where author = 'Mark Twain'"
    ]
}
```

版本

version 字段是所有请求映射模板通用的，用于定义模板使用的版本。version 字段为必填项。“2018-05-29”值是 Amazon RDS 映射模板支持的唯一版本。

```
"version": "2018-05-29"
```

statements 和 variableMap

statements 数组是开发人员提供的查询的占位符。目前，每个请求映射模板最多支持两个查询。variableMap 是一个可选字段，它包含可用于使 SQL 语句更短且更易读的别名。例如，可以编写以下代码：

```
{
"version": "2018-05-29",
    "statements": [
        "insert into BOOKS VALUES (:AUTHOR, :TITLE, :ISBN13)",
        "select * from BOOKS WHERE isbn13 = :ISBN13"
    ],
    "variableMap": {
        ":AUTHOR": $util.toJson($ctx.args.newBook.author),
        ":TITLE": $util.toJson($ctx.args.newBook.title),
        ":ISBN13": $util.toJson($ctx.args.newBook.isbn13)
    }
}
```

AWS AppSync 使用变量映射值构建 SqlParameterized 查询，这些查询将发送到 Amazon Aurora Serverless 数据 API。SQL 语句是使用变量映射中提供的参数执行的，从而消除了 SQL 注入风险。

VariableTypeHintMap

variableTypeHintMap 是一个可选字段，它包含可用于发送 SQL 参数类型提示的别名类型。这些类型提示避免了 SQL 语句中的显式转换，从而使这些语句变得更短。例如，可以编写以下代码：

```
{
    "version": "2018-05-29",
    "statements": [
        "insert into LOGINDATA VALUES (:ID, :TIME)",
        "select * from LOGINDATA WHERE id = :ID"
     ],
     "variableMap": {
        ":ID": $util.toJson($ctx.args.id),
        ":TIME": $util.toJson($ctx.args.time)
     },
     "variableTypeHintMap": {
```

```
        ":id": "UUID",
        ":time": "TIME"
    }
}
```

AWS AppSync 使用变量映射值构建查询，这些查询将发送到 Amazon Aurora Serverless 数据 API。它还使用 `variableTypeHintMap` 数据，并将类型的信息发送到 RDS。可以在此处找到 RDS 支持的 `typeHints`。

OpenSearch 的解析器映射模板参考

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

通过使用适用于 Amazon OpenSearch Service 的 AWS AppSync 解析器，您可以使用 GraphQL 在您的账户的现有 OpenSearch Service 域中存储和检索数据。该解析器的工作方式是，允许您将传入的 GraphQL 请求映射到 OpenSearch Service 请求，然后将 OpenSearch Service 响应映射回 GraphQL。本节介绍了支持的 OpenSearch Service 操作的映射模板。

请求映射模板

大多数 OpenSearch Service 请求映射模板具有通用结构，其中仅几个部分发生变化。以下示例对 OpenSearch Service 域运行搜索，其中文档是按照名为 `post` 的索引组织的。搜索参数在 `body` 部分定义，而许多常用查询子句在 `query` 字段中定义。此示例将搜索在文档的 `"Nadia"` 字段中包含 `"Bailey"` 和/或 `author` 的文档。

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path":"/post/_search",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "from":0,
            "size":50,
```

```
            "query" : {
                "bool" : {
                    "should" : [
                        {"match" : { "author" : "Nadia" }},
                        {"match" : { "author" : "Bailey" }}
                    ]
                }
            }
        }
    }
}
```

响应映射模板

与其他数据源一样，OpenSearch Service 向 AWS AppSync 发送响应，该响应需要转换为 GraphQL。

大多数 GraphQL 查询从 OpenSearch Service 响应中查找 `_source` 字段。由于您可以搜索以返回单个文档或文档列表，因此，可以在 OpenSearch Service 中使用两种常见的响应映射模板：

结果列表

```
[
    #foreach($entry in $context.result.hits.hits)
      #if( $velocityCount > 1 ) , #end
        $utils.toJson($entry.get("_source"))
    #end
]
```

单个项目

```
$utils.toJson($context.result.get("_source"))
```

`operation` 字段

（仅限请求映射模板）

AWS AppSync 发送到 OpenSearch Service 域的 HTTP 方法或动词（GET、POST、PUT、HEAD 或 DELETE）。键和值都必须是字符串。

```
"operation" : "PUT"
```

path 字段

（仅限请求映射模板）

来自 AWS AppSync 的 OpenSearch Service 请求的搜索路径。这构成了操作的 HTTP 谓词的 URL。键和值都必须是字符串。

```
"path" : "/<indexname>/_doc/<_id>"
"path" : "/<indexname>/_doc"
"path" : "/<indexname>/_search"
"path" : "/<indexname>/_update/<_id>
```

在评估映射模板时，该路径作为 HTTP 请求的一部分发送，包括 OpenSearch Service 域。例如，上一个示例可能会转换为：

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

params 字段

（仅限请求映射模板）

用于指定搜索执行的操作，最常见的是在正文中设置查询值。但是，可以配置若干其他功能，如响应的格式设置。

- headers

  标头信息（为键值对）。键和值都必须是字符串。例如：

  ```
  "headers" : {
      "Content-Type" : "application/json"
  }
  ```

  > **Note**
  >
  > AWS AppSync 目前仅支持将 JSON 作为 `Content-Type`。

- queryString

  指定常用选项的键值对，如 JSON 响应的代码格式设置。键和值都必须是字符串。例如，如果您要获得格式正确的 JSON，应使用：

```
"queryString" : {
    "pretty" : "true"
}
```

- body

  这是请求的主要部分，它允许 AWS AppSync 为您的 OpenSearch Service 域创建正确格式的搜索请求。键必须是组成对象的一个字符串。下面介绍了几个演示。

示例 1

返回城市与“seattle”匹配的所有文档：

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "match" : {
            "city" : "seattle"
        }
    }
}
```

示例 2

返回将“washington”作为城市或州匹配的所有文档。

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "multi_match" : {
            "query" : "washington",
            "fields" : ["city", "state"]
        }
    }
}
```

传递变量

（仅限请求映射模板）

还可以在 VTL 语句中作为评估的一部分传递变量。例如，假设您具有以下 GraphQL 查询，如下所示：

```
query {
    searchForState(state: "washington"){
        ...
    }
}
```

映射模板可能将状态作为参数：

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "multi_match" : {
            "query" : "$context.arguments.state",
            "fields" : ["city", "state"]
        }
    }
}
```

有关可以包含在 VTL 中的实用工具的列表，请参阅访问请求标头。

Lambda 的解析器映射模板参考

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

您可以使用 AWS AppSync 函数和解析器来调用位于您账户中的 Lambda 函数。在将请求负载和 Lambda 函数返回给客户之前，您可以调整请求负载和 Lambda 函数的响应。您还可以使用映射模板来提示 AWS AppSync 要调用的操作的性质。本节介绍了支持的 Lambda 操作的各种映射模板。

请求映射模板

Lambda 请求映射模板可以处理与您的 Lambda 函数相关的字段：

```
{
  "version": string,
  "operation": Invoke|BatchInvoke,
  "payload": any type,
  "invocationType": RequestResponse|Event
}
```

以下是 Lambda 请求映射模板解析后的 JSON 架构表示形式：

```
{
  "definitions": {},
  "$schema": "https://json-schema.org/draft-06/schema#",
  "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
  "type": "object",
  "properties": {
    "version": {
      "$id": "/properties/version",
      "type": "string",
      "enum": [
        "2018-05-29"
      ],
      "title": "The Mapping template version.",
      "default": "2018-05-29"
    },
    "operation": {
      "$id": "/properties/operation",
      "type": "string",
      "enum": [
        "Invoke",
        "BatchInvoke"
      ],
      "title": "The Mapping template operation.",
      "description": "What operation to execute.",
      "default": "Invoke"
    },
    "payload": {},
    "invocationType": {
      "$id": "/properties/invocationType",
      "type": "string",
      "enum": [
        "RequestResponse",
        "Event"
      ],
```

```
      "title": "The Mapping template invocation type.",
      "description": "What invocation type to execute.",
      "default": "RequestResponse"
    }
  },
  "required": [
    "version",
    "operation"
  ],
  "additionalProperties": false
}
```

以下示例使用了一个invoke操作，其有效载荷数据是 GraphQL 架构中的getPost字段，其参数来自上下文：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $util.toJson($context.arguments)
  }
}
```

整个映射文档作为输入传递给您的 Lambda 函数，因此前面的示例现在如下所示：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "id": "postId1"
    }
  }
}
```

版本

所有请求映射模板都通用，version定义了模板使用的版本。version是必填的，是一个静态值：

```
"version": "2018-05-29"
```

操作

Lambda 数据源允许您在operation字段中定义两个操作：Invoke和。BatchInvoke该Invoke操作允许 AWS AppSync 你为每个 GraphQL 字段解析器调用你的 Lambda 函数。 BatchInvoke指示 AWS AppSync 对当前 GraphQL 字段进行批量请求。operation 字段为必填项。

对于Invoke，已解析的请求映射模板与 Lambda 函数的输入负载相匹配。让我们修改上面的示例：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
    "payload": {
      "arguments": $util.toJson($context.arguments)
    }
}
```

此问题已解决并传递给 Lambda 函数，该函数可能如下所示：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
    "payload": {
      "arguments": {
        "id": "postId1"
      }
    }
}
```

对于BatchInvoke，映射模板将应用于批次中的每个字段解析器。为简洁起见，将所有已解析的映射模板payload值 AWS AppSync 合并到与映射模板匹配的单个对象下的列表中。以下示例模板显示合并：

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": $util.toJson($context)
}
```

此模板解析为以下映射文档：

```
{
```

```
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

payload列表中的每个元素对应一个批次项目。Lambda 函数还应返回与请求中发送的项目顺序相匹配的列表形响应：

```
[
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
 1
  { "data": {...}, "errorMessage": null, "errorType": null }, // result for batch item
 2
  { "data": {...}, "errorMessage": null, "errorType": null }  // result for batch item
 3
]
```

有效负载

该payload字段是一个容器，用于将任何格式正确的 JSON 传递给 Lambda 函数。如果该operation字段设置为BatchInvoke，则 AWS AppSync 将现有payload值换成一个列表。payload 字段为可选项。

调用类型

Lambda 数据源允许您定义两种调用类型：和。RequestResponse Event调用类型与 Lambda API 中定义的调用类型同义。RequestResponse调用类型允许 AWS AppSync 同步调用您的 Lambda 函数以等待响应。该Event调用允许您异步调用 Lambda 函数。有关 Lambda 如何处理Event调用类型请求的更多信息，请参阅异步调用。invocationType 字段为可选项。如果请求中未包含此字段，AWS AppSync 则默认为RequestResponse调用类型。

对于任何invocationType字段，已解析的请求都与 Lambda 函数的输入负载相匹配。让我们修改上面的示例：

```
{
  "version": "2018-05-29",
```

```
  "operation": "Invoke",
  "invocationType": "Event"
  "payload": {
    "arguments": $util.toJson($context.arguments)
  }
}
```

此问题已解决并传递给 Lambda 函数，该函数可能如下所示：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

当该BatchInvoke操作与Event调用类型字段结合使用时，将以与上述相同的方式 AWS AppSync 合并字段解析器，然后将请求作为异步事件传递给您的 Lambda 函数，其为值列表payload。我们建议您禁用Event调用类型解析器的解析器缓存，因为如果出现缓存命中，这些解析器不会发送到 Lambda。

响应映射模板

与其他数据源一样，您的 Lambda 函数发送的响应必须转换为 GraphQL 类型。 AWS AppSync

Lambda 函数的结果是在通过 Velocity 模板语言 (VTL) `$context.result` 属性提供的 `context` 对象上设置的。

如果 Lambda 函数响应的形状与 GraphQL 类型的形状完全匹配，您可以使用以下响应映射模板转发响应：

```
$util.toJson($context.result)
```

没有必填字段，也没有形状限制应用于响应映射模板。但是，由于 GraphQL 是强类型化的，因此解析的映射模板必须与预期的 GraphQL 类型匹配。

Lambda 函数批处理的响应

如果该`operation`字段设置为`BatchInvoke`， AWS AppSync则需要从 Lambda 函数返回的项目列表。 AWS AppSync 为了将每个结果映射回原始请求项，响应列表的大小和顺序必须匹配。在响应列表中包含`null`项目是有效`$ctx.result`的；相应地设置为 null。

直接 Lambda 解析器

如果您想完全避免使用映射模板， AWS AppSync 可以为您的 Lambda 函数提供默认负载，并为 GraphQL 类型提供默认 Lambda 函数响应。您可以选择提供请求模板、响应模板或两者都不提供，然后进行相应的 AWS AppSync 处理。

直接 Lambda 请求映射模板

如果未提供请求映射模板，则 AWS AppSync 会将该`Context`对象作为操作直接发送到您的 Lambda 函数。`Invoke`有关 `Context` 对象结构的更多信息，请参阅 解析器映射模板上下文参考。

直接 Lambda 响应映射模板

如果未提供响应映射模板，则在收到 Lambda 函数的响应后 AWS AppSync 执行以下两项操作之一。如果您没有提供请求映射模板，或者您在版本中提供了请求映射模板`2018-05-29`，则响应将等同于以下响应映射模板：

```
#if($ctx.error)
     $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
 #end
 $util.toJson($ctx.result)
```

如果您提供了带有版本的模板`2017-02-28`，则响应逻辑的功能等同于以下响应映射模板：

```
$util.toJson($ctx.result)
```

从表面上看，绕过映射模板的操作方式与使用某些映射模板类似，如前面的示例所示。但在幕后，完全绕过了映射模板评估。由于绕过了模板评估步骤，因此与带有需要评估的响应映射模板的 Lambda 函数相比，在某些情况下，应用程序在响应期间遇到的开销和延迟可能会更少。

直接 Lambda 解析器响应中的自定义错误处理

您可以通过引发自定义异常，自定义直接 Lambda 解析器调用的 Lambda 函数的错误响应。以下示例演示如何使用创建自定义异常 JavaScript：

```
class CustomException extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomException";
  }
}

throw new CustomException("Custom message");
```

在引发异常时，errorType 和 errorMessage 分别是引发的自定义错误的 name 和 message。

如果errorType是UnauthorizedException，则 AWS AppSync返回默认消息 ("You are not authorized to make this call.") 而不是自定义消息。

以下片段是演示自定义的 GraphQL 响应示例：errorType

```
{
  "data": {
    "query": null
  },
  "errors": [
    {
      "path": [
        "query"
      ],
      "data": null,
      "errorType": "CustomException",
      "errorInfo": null,
      "locations": [
        {
          "line": 5,
          "column": 10,
          "sourceName": null
        }
      ],
      "message": "Custom Message"
    }
  ]
}
```

直接 Lambda 解析器：已启用批处理

您可以通过在解析器上配置 `maxBatchSize`，为直接 Lambda 解析器启用批处理。如果设置`0`为大于 D `maxBatchSize` irect Lambda 解析器的值，则会分批向您的 Lambda 函数 AWS AppSync 发送大小不超过的请求。`maxBatchSize`

在 Direct Lambda 解析器`0`上设置`maxBatchSize`为会关闭批处理。

有关 Lambda 解析器上的批处理的工作方式的更多信息，请参阅高级使用案例：批处理。

请求映射模板

启用批处理但未提供请求映射模板时， AWS AppSync 会将`Context`对象列表作为`BatchInvoke`操作直接发送到您的 Lambda 函数。

响应映射模板

如果启用了批处理并且未提供响应映射模板，响应逻辑相当于以下响应映射模板：

```
#if( $context.result && $context.result.errorMessage )
      $utils.error($context.result.errorMessage, $context.result.errorType,
      $context.result.data)
#else
      $utils.toJson($context.result.data)
#end
```

Lambda 函数返回结果列表的顺序必须与发送的 `Context` 对象列表相同。您可以为特定结果提供 `errorMessage` 和 `errorType` 以返回各个错误。列表中的每个结果采用以下格式：

```
{
   "data" : { ... }, // your data
   "errorMessage" : { ... }, // optional, if included an error entry is added to the
 "errors" object in the AppSync response
   "errorType" : { ... } // optional, the error type
}
```

> **Note**
>
> 目前忽略结果对象中的其他字段。

处理来自 Lambda 的错误

您可以通过在 Lambda 函数中引发异常或错误，为所有结果返回错误。如果批处理请求的负载请求或响应太大，Lambda 将返回错误。在这种情况下，您应该考虑减少 `maxBatchSize` 或减少响应负载大小。

有关处理各个错误的信息，请参阅返回单个错误。

示例 Lambda 函数

使用以下架构，您可以为`Post.relatedPosts`字段解析器创建 Direct Lambda 解析器，并通过上述设置启用批处理：`maxBatchSize0`

```
schema {
    query: Query
    mutation: Mutation
}

type Query {
    getPost(id:ID!): Post
    allPosts: [Post]
}

type Mutation {
    addPost(id: ID!, author: String!, title: String, content: String, url: String):
 Post!
}

type Post {
    id: ID!
    author: String!
    title: String
    content: String
    url: String
    ups: Int
    downs: Int
    relatedPosts: [Post]
}
```

在以下查询中，将为批量请求调用 Lambda 函数以解析 `relatedPosts`：

```
query getAllPosts {
  allPosts {
```

```
    id
    relatedPosts {
      id
    }
  }
}
```

下面提供了 Lambda 函数的简单实施：

```
const posts = {
  1: {
    id: '1',
    title: 'First book',
    author: 'Author1',
    url: 'https://amazon.com/',
    content:
      'SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT
 AUTHOR 1 SAMPLE TEXT AUTHOR 1 SAMPLE TEXT AUTHOR 1',
    ups: '100',
    downs: '10',
  },
  2: {
    id: '2',
    title: 'Second book',
    author: 'Author2',
    url: 'https://amazon.com',
    content: 'SAMPLE TEXT AUTHOR 2 SAMPLE TEXT AUTHOR 2 SAMPLE TEXT',
    ups: '100',
    downs: '10',
  },
  3: { id: '3', title: 'Third book', author: 'Author3', url: null, content: null, ups:
 null, downs: null },
  4: {
    id: '4',
    title: 'Fourth book',
    author: 'Author4',
    url: 'https://www.amazon.com/',
    content:
      'SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT AUTHOR 4 SAMPLE TEXT
 AUTHOR 4',
    ups: '1000',
    downs: '0',
```

```
  },
  5: {
    id: '5',
    title: 'Fifth book',
    author: 'Author5',
    url: 'https://www.amazon.com/',
    content: 'SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE TEXT AUTHOR 5 SAMPLE
 TEXT AUTHOR 5 SAMPLE TEXT',
    ups: '50',
    downs: '0',
  },
}

const relatedPosts = {
  1: [posts['4']],
  2: [posts['3'], posts['5']],
  3: [posts['2'], posts['1']],
  4: [posts['2'], posts['1']],
  5: [],
}
exports.handler = async (event) => {
  console.log('event ->', event)
  // retrieve the ID of each post
  const ids = event.map((context) => context.source.id)
  // fetch the related posts for each post id
  const related = ids.map((id) => relatedPosts[id])

  // return the related posts; or an error if none were found
  return related.map((r) => {
    if (r.length > 0) {
      return { data: r }
    } else {
      return { data: null, errorMessage: 'Not found', errorType: 'ERROR' }
    }
  })
}
```

的解析器映射模板参考 EventBridge

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

与 EventBridge 数据源一起使用的 AWS AppSync 解析器映射模板允许您向 Amazon EventBridge 总线发送自定义事件。

请求映射模板

PutEvents请求映射模板允许您将多个自定义事件发送到 EventBridge事件总线。 映射文档具有以下结构：

```
{
    "version" : "2018-05-29",
    "operation" : "PutEvents",
    "events" : [{}]
}
```

以下是用于的请求映射模板的示例 EventBridge：

```
{
    "version": "2018-05-29",
    "operation": "PutEvents",
    "events": [{
        "source": "com.mycompany.myapp",
        "detail": {
            "key1" : "value1",
            "key2" : "value2"
        },
        "detailType": "myDetailType1"
    },
    {
        "source": "com.mycompany.myapp",
        "detail": {
            "key3" : "value3",
            "key4" : "value4"
```

```
        },
        "detailType": "myDetailType2",
        "resources" : ["Resource1", "Resource2"],
        "time" : "2023-01-01T00:30:00.000Z"
    }

    ]
}
```

响应映射模板

如果PutEvents操作成功， EventBridge 则来自的响应将包含在$ctx.result：

```
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end
  $util.toJson($ctx.result)
```

执行 PutEvents 操作时发生的错误（例如 InternalExceptions 或 Timeouts）将出现在 $ctx.error 中。有关常见错误 EventBridge的列表，请参阅EventBridge 常见错误参考。

result 将采用以下格式：

```
{
    "Entries" [
        {
            "ErrorCode" : String,
            "ErrorMessage" : String,
            "EventId" : String
        }
    ],
    "FailedEntryCount" : number
}
```

- Entries

  摄取的事件结果（成功和失败）。如果摄取成功，则在该条目中包含 EventID。否则，您可以使用 ErrorCode 和 ErrorMessage 找出条目的问题。

  对于每条记录，响应元素的索引与请求数组中的索引相同。
- FailedEntryCount

失败条目数。该值表示为一个整数。

有关响应的更多信息PutEvents，请参阅PutEvents。

示例响应 1

以下示例是具有两个成功事件的 PutEvents 操作：

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ],
    "FailedEntryCount" : 0
}
```

示例响应 2

以下示例是具有三个事件的 PutEvents 操作，其中的两个事件是成功事件，一个是失败事件：

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        },
        {
            "ErrorCode" : "SampleErrorCode",
            "ErrorMessage" : "Sample Error Message"
        }
    ],
    "FailedEntryCount" : 1
}
```

PutEvents 字段

- 版本

  version 字段是所有请求映射模板通用的，用于定义模板使用的版本。该字段为必填。该值2018-05-29是 EventBridge 映射模板支持的唯一版本。

- 操作

  唯一支持的操作是 PutEvents。通过执行该操作，您可以将自定义事件添加到事件总线中。

- 事件

  将添加到事件总线的事件数组。应该为该数组分配 1-10 个项目。

  Event 对象是一个有效的 JSON 对象，它具有以下字段：

  - "source"：定义事件源的字符串。
  - "detail"：可用于附加事件相关信息的 JSON 对象。该字段可以是空映射 ({ })。
  - "detailType：指定事件类型的字符串。
  - "resources"：指定事件中涉及的资源的 JSON 字符串数组。该字段可以是空数组。
  - "time"：以字符串形式提供的事件时间戳。它应该采用 RFC3339 时间戳格式。

以下片段是一些有效 Event 对象示例：

示例 1

```
{
    "source" : "source1",
    "detail" : {
        "key1" : [1,2,3,4],
        "key2" : "strval"
    },
    "detailType" : "sampleDetailType",
    "resources" : ["Resouce1", "Resource2"],
    "time" : "2022-01-10T05:00:10Z"
}
```

示例 2

```
{
```

```
    "source" : "source1",
    "detail" : {},
    "detailType" : "sampleDetailType"
}
```

示例 3

```
{
    "source" : "source1",
    "detail" : {
        "key1" : 1200
    },
    "detailType" : "sampleDetailType",
    "resources" : []
}
```

None 数据源的解析器映射模板参考

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

通过使用与 None 类型的数据源一起使用的 AWS AppSync 解析器映射模板，您可以设置 AWS AppSync 本地操作的请求形状。

请求映射模板

映射模板很简单，使您能够通过 `payload` 字段传递尽可能多的上下文信息。

```
{
   "version": string,
   "payload": any type
}
```

下面是请求映射模板的 JSON 架构表示形式（解析后）：

```
{
```

```
    "definitions": {},
    "$schema": "https://json-schema.org/draft-06/schema#",
    "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
    "type": "object",
    "properties": {
        "version": {
            "$id": "/properties/version",
            "type": "string",
            "enum": [
                "2018-05-29"
            ],
            "title": "The Mapping template version.",
            "default": "2018-05-29"
        },
        "payload": {}
    },
    "required": [
        "version"
    ],
    "additionalProperties": false
}
```

以下是通过 VTL 上下文属性 `$context.arguments` 传递字段参数的示例：

```
{
    "version": "2018-05-29",
    "payload": $util.toJson($context.arguments)
}
```

payload 字段的值将转发到响应映射模板并可用于 VTL 上下文属性 (`$context.result`) 上。

这是一个表示 payload 字段的内插值的示例：

```
{
    "id": "postId1"
}
```

版本

version 字段是所有请求映射模板通用的，用于定义模板使用的版本。

version 字段为必填项。

示例：

```
"version": "2018-05-29"
```

有效负载

payload 字段是一个容器，可用于将任何格式正确的 JSON 传递到响应映射模板。

payload 字段为可选项。

响应映射模板

由于不存在任何数据源，payload 字段的值将转发到响应映射模板并在 context 对象上设置（此对象可通过 VTL $context.result 属性提供）。

如果 payload 字段值的形状与 GraphQL 类型的形状完全相符，您可以使用以下响应映射模板转发响应：

```
$util.toJson($context.result)
```

没有必填字段，也没有形状限制应用于响应映射模板。但是，由于 GraphQL 是强类型化的，因此解析的映射模板必须与预期的 GraphQL 类型匹配。

HTTP 的解析器映射模板参考

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时系统及其文档。请考虑使用 APPSYNC_JS 运行时系统和此处的指南。

借助 AWS AppSync HTTP 解析程序映射模板，您能够结束从 AWS AppSync 到任何 HTTP 终端节点的请求，以及从 HTTP 终端节点返回 AWS AppSync 的响应的形状。通过使用映射模板，您还可以向 AWS AppSync 提供提示以指出要调用的操作的性质。本节介绍用于支持的 HTTP 解析器的不同映射模板。

请求映射模板

```
{
```

```
    "version": "2018-05-29",
    "method": "PUT|POST|GET|DELETE|PATCH",
    "params": {
        "query": Map,
        "headers": Map,
        "body": any
    },
    "resourcePath": string
}
```

解析 HTTP 请求映射模板后，请求映射模板的 JSON 架构表现形式如下所示：

```
{
    "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
    "type": "object",
    "properties": {
        "version": {
        "$id": "/properties/version",
        "type": "string",
        "title": "The Version Schema ",
        "default": "",
        "examples": [
            "2018-05-29"
        ],
        "enum": [
            "2018-05-29"
        ]
        },
        "method": {
        "$id": "/properties/method",
        "type": "string",
        "title": "The Method Schema ",
        "default": "",
        "examples": [
            "PUT|POST|GET|DELETE|PATCH"
        ],
        "enum": [
            "PUT",
            "PATCH",
            "POST",
            "DELETE",
            "GET"
        ]
```

```
        },
        "params": {
        "$id": "/properties/params",
        "type": "object",
        "properties": {
            "query": {
            "$id": "/properties/params/properties/query",
            "type": "object"
            },
            "headers": {
            "$id": "/properties/params/properties/headers",
            "type": "object"
            },
            "body": {
            "$id": "/properties/params/properties/body",
            "type": "string",
            "title": "The Body Schema ",
            "default": "",
            "examples": [
                ""
            ]
            }
        }
        },
        "resourcePath": {
        "$id": "/properties/resourcePath",
        "type": "string",
        "title": "The Resourcepath Schema ",
        "default": "",
        "examples": [
            ""
        ]
        }
    },
    "required": [
        "version",
        "method",
        "resourcePath"
    ]
}
```

以下是 HTTP POST 请求的示例，具有 `text/plain` 正文：

```
{
    "version": "2018-05-29",
    "method": "POST",
    "params": {
        "headers":{
        "Content-Type":"text/plain"
        },
        "body":"this is an example of text body"
    },
    "resourcePath": "/"
}
```

版本

仅限请求映射模板

定义模板使用的版本。version 对于所有请求映射模板都相同且是必需的。

```
"version": "2018-05-29"
```

方法

仅限请求映射模板

AWS AppSync 发送到 HTTP 终端节点的 HTTP 方法或动词（GET、POST、PUT、PATCH 或 DELETE）。

```
"method": "PUT"
```

ResourcePath

仅限请求映射模板

您要访问的资源路径。资源路径与 HTTP 数据源中更多终端节点一起，构成 AWS AppSync 发出请求到的 URL。

```
"resourcePath": "/v1/users"
```

评估映射模板时，此路径作为 HTTP 请求的一部分发送，包括 HTTP 终端节点。例如，上一个示例可能会转换为如下所示：

```
PUT <endpoint>/v1/users
```

Params 字段

仅限请求映射模板

用于指定搜索执行的操作，最常见的是在正文中设置查询值。但是，可以配置若干其他功能，如响应的格式设置。

headers

标头信息（为键值对）。键和值都必须是字符串。

例如：

```
"headers" : {
    "Content-Type" : "application/json"
}
```

目前支持的 Content-Type 标头包括：

```
text/*
application/xml
application/json
application/soap+xml
application/x-amz-json-1.0
application/x-amz-json-1.1
application/vnd.api+json
application/x-ndjson
```

注意：您不能设置以下 HTTP 标头：

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

指定常用选项的键值对，如 JSON 响应的代码格式设置。键和值都必须是字符串。以下示例显示如何发送 ?type=json 格式的请求字符串：

```
"query" : {
    "type" : "json"
}
```

body

正文包含您选择要设置的 HTTP 请求正文。请求正文始终是 UTF-8 编码格式的字符串，除非内容类型指定字符集。

```
"body":"body string"
```

AWS AppSync 识别的 HTTPS 终端节点证书颁发机构 (CA)

> **Note**
>
> Let's Encrypt 通过 identrust and isrgrootx1 证书进行验证。如果您使用 Let's Encrypt，则无需采取任何措施。

目前，在使用 HTTPS 时，HTTP 解析器不支持自签名证书。AWS在解析 HTTPS 的 SSL/TLS 证书时，AppSync 识别以下证书颁发机构：

AWS AppSync 中的已知根证书

名称	日期	SHA1 指纹
digicertassuredidrootca	2018 年 4 月 21 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
trustcenterclass2caii	2018 年 4 月 21 日	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
thawtepremiumserverca	2018 年 4 月 21 日	E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:66:66

名称	日期	SHA1 指纹
cia-crt-g3-02-ca	2016 年 11 月 23 日	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
swisssignplatinumg2ca	2018 年 4 月 21 日	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66
swisssignsilverg2ca	2018 年 4 月 21 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
thawteserverca	2018 年 4 月 21 日	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
equifaxsecureebusinessca1	2018 年 4 月 21 日	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
securetrustca	2018 年 4 月 21 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
utnuserfirstclientauthemailca	2018 年 4 月 21 日	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
thawtepersonalfreemailca	2018 年 4 月 21 日	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
affirmtrustnetworkingca	2018 年 4 月 21 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
entrustevca	2018 年 4 月 21 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
utnuserfirsthardwareca	2018 年 4 月 21 日	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
certumca	2018 年 4 月 21 日	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
addtrustclass1ca	2018 年 4 月 21 日	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D

名称	日期	SHA1 指纹
entrustrootcag2	2018 年 4 月 21 日	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
equifaxsecureca	2018 年 4 月 21 日	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
quovadisrootca3	2018 年 4 月 21 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	2018 年 4 月 21 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
digicertglobalrootg2	2018 年 4 月 21 日	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
digicerthighassuranceevrootca	2018 年 4 月 21 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
secomvalicertclass1ca	2018 年 4 月 21 日	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
equifaxsecureglobalebusinessca1	2018 年 4 月 21 日	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
geotrustuniversalca	2018 年 4 月 21 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
deprecateditsecca	2012 年 1 月 27 日	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
verisignclass3ca	2018 年 4 月 21 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
thawteprimaryrootcag3	2018 年 4 月 21 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
thawteprimaryrootcag2	2018 年 4 月 21 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12

名称	日期	SHA1 指纹
deutschetelekomroo tca2	2018 年 4 月 21 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD: D6:13:30:FD:8C:DE:37:BF
buypassclass3ca	2018 年 4 月 21 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD: C7:C2:81:A5:BC:A9:64:57
utnuserfirstobject ca	2018 年 4 月 21 日	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B: AC:1D:81:D8:38:5E:2D:46
geotrustprimaryca	2018 年 4 月 21 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2: 10:0D:D6:02:90:37:F0:96
buypassclass2ca	2018 年 4 月 21 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6: C7:6B:EB:C6:0B:12:40:99
baltimorecodesigni ngca	2018 年 4 月 21 日	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD: 49:27:08:7C:60:56:7B:0D
verisignclass1ca	2018 年 4 月 21 日	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45: 3E:64:09:EA:E8:7D:60:F1
baltimorecybertrus tca	2018 年 4 月 21 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88: 2C:78:DB:28:52:CA:E4:74
starfieldclass2ca	2018 年 4 月 21 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20: 14:C3:D0:E3:37:0E:B5:8A
camerfirmachambers commerceca	2018 年 4 月 21 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0: DB:72:2E:31:30:61:F0:B1
ttelesecglobalroot class3ca	2018 年 4 月 21 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70: 19:9D:2A:BE:11:E3:81:D1
verisignclass3g5ca	2018 年 4 月 21 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD: 56:BE:3D:9B:67:44:A5:E5
ttelesecglobalroot class2ca	2018 年 4 月 21 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62: 32:17:65:CF:17:D8:94:E9

名称	日期	SHA1 指纹
trustcenteruniversalcai	2018 年 4 月 21 日	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
verisignclass3g4ca	2018 年 4 月 21 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3g3ca	2018 年 4 月 21 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
xrampglobalca	2018 年 4 月 21 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
amzninternalrootca	2008 年 12 月 12 日	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06
certplusclass3pprimaryca	2018 年 4 月 21 日	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
certumtrustednetworkca	2018 年 4 月 21 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
verisignclass3g2ca	2018 年 4 月 21 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
globalsignr3ca	2018 年 4 月 21 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
utndatacorpsgcca	2018 年 4 月 21 日	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
secomscrootca2	2018 年 4 月 21 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
gtecybertrustglobalca	2018 年 4 月 21 日	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
secomscrootca1	2018 年 4 月 21 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7

名称	日期	SHA1 指纹
affirmtrustcommercialca	2018 年 4 月 21 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
trustcenterclass4caii	2018 年 4 月 21 日	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
verisignuniversalrootca	2018 年 4 月 21 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
globalsignr2ca	2018 年 4 月 21 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
certplusclass2primaryca	2018 年 4 月 21 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
digicertglobalrootca	2018 年 4 月 21 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
globalsignca	2018 年 4 月 21 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
thawteprimaryrootca	2018 年 4 月 21 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
starfieldrootg2ca	2018 年 4 月 21 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
geotrustglobalca	2018 年 4 月 21 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
soneraclass2ca	2018 年 4 月 21 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
verisigntsaca	2018 年 4 月 21 日	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
soneraclass1ca	2018 年 4 月 21 日	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF

名称	日期	SHA1 指纹
quovadisrootca	2018 年 4 月 21 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
affirmtrustpremiumeccca	2018 年 4 月 21 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
starfieldservicesrootg2ca	2018 年 4 月 21 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
valicertclass2ca	2018 年 4 月 21 日	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
comodoaaaca	2018 年 4 月 21 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
aolrootca2	2018 年 4 月 21 日	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
keynectisrootca	2018 年 4 月 21 日	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
addtrustqualifiedca	2018 年 4 月 21 日	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
aolrootca1	2018 年 4 月 21 日	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignclass2g3ca	2018 年 4 月 21 日	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
addtrustexternalca	2018 年 4 月 21 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
verisignclass2g2ca	2018 年 4 月 21 日	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
geotrustprimarycag3	2018 年 4 月 21 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD

名称	日期	SHA1 指纹
geotrustprimarycag2	2018 年 4 月 21 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
swisssigngoldg2ca	2018 年 4 月 21 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
entrust2048ca	2018 年 4 月 21 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
chunghwaepkirootca	2018 年 4 月 21 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
camerfirmachambersignca	2018 年 4 月 21 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
camerfirmachambersca	2018 年 4 月 21 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
godaddyclass2ca	2018 年 4 月 21 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
affirmtrustpremiumca	2018 年 4 月 21 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
verisignclass1g3ca	2018 年 4 月 21 日	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
secomevrootca1	2018 年 4 月 21 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
verisignclass1g2ca	2018 年 4 月 21 日	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
amzninternalinfosecag3	2015 年 2 月 27 日	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
cia-crt-g3-01-ca	2016 年 11 月 23 日	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2

名称	日期	SHA1 指纹
godaddyrootg2ca	2018 年 4 月 21 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
digicertassuredidrootca	2018 年 4 月 21 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
microseceszignorootca2009	2018 年 4 月 21 日	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
affirmtrustcommercial	2018 年 4 月 21 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
comodoecccertificationauthority	2018 年 4 月 21 日	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
cadisigrootr2	2018 年 4 月 21 日	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
swisssignsilvercag2	2018 年 4 月 21 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
securetrustca	2018 年 4 月 21 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
cadisigrootr1	2018 年 4 月 21 日	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
accvraiz1	2018 年 4 月 21 日	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
entrustrootcertificationauthority	2018 年 4 月 21 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
camerfirmaglobalchambersignroot	2018 年 4 月 21 日	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
dstacescax6	2018 年 4 月 21 日	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D

名称	日期	SHA1 指纹
identrustpublicsectorrootca1	2018 年 4 月 21 日	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
starfieldrootcertificateauthorityg2	2018 年 4 月 21 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
secureglobalca	2018 年 4 月 21 日	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
eecertificationcentrerootca	2018 年 4 月 21 日	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
opentrustrootcag3	2018 年 4 月 21 日	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8A:A6
teliasonerarootcav1	2018 年 4 月 21 日	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
autoridaddecertificacionfirmaprofesionalcifa62634068	2018 年 4 月 21 日	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
opentrustrootcag2	2018 年 4 月 21 日	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	2018 年 4 月 21 日	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:D5:7E
globalsigneccrootcar5	2018 年 4 月 21 日	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
globalsigneccrootcar4	2018 年 4 月 21 日	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
izenpecom	2018 年 4 月 21 日	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19

名称	日期	SHA1 指纹
turktrustelektroniksertifikahizmetsaglayicisih5	2018 年 4 月 21 日	C4:18:F6:4D:46:D1:DF:00:3D:27:30:13:72:43:A9:12:11:C6:75:FB
gdcatrustauthr5root	2018 年 4 月 21 日	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
dtrustrootclass3ca22009	2018 年 4 月 21 日	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
quovadisrootca3	2018 年 4 月 21 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	2018 年 4 月 21 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
geotrustprimarycertificationauthorityg3	2018 年 4 月 21 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycertificationauthorityg2	2018 年 4 月 21 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
oistewisekeyglobalrootgbca	2018 年 4 月 21 日	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
addtrustexternalroot	2018 年 4 月 21 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
chambersofcommerceroot2008	2018 年 4 月 21 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
digicertglobalrootg3	2018 年 4 月 21 日	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E

名称	日期	SHA1 指纹
comodoaaaservicesroot	2018 年 4 月 21 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
digicertglobalrootg2	2018 年 4 月 21 日	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73:FE:06:D1:CC:8D:4F:82:A4
certinomisrootca	2018 年 4 月 21 日	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:88:A8
oistewisekeyglobalrootgaca	2018 年 4 月 21 日	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
dstrootcax3	2018 年 4 月 21 日	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
certigna	2018 年 4 月 21 日	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
digicerthighassuranceevrootca	2018 年 4 月 21 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
soneraclass2rootca	2018 年 4 月 21 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
trustcorrootcertca2	2018 年 4 月 21 日	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
usertrustrsacertificationauthority	2018 年 4 月 21 日	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
trustcorrootcertca1	2018 年 4 月 21 日	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
geotrustuniversalca	2018 年 4 月 21 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
certsignrootca	2018 年 4 月 21 日	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B

名称	日期	SHA1 指纹
amazonrootca4	2018 年 4 月 21 日	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
amazonrootca3	2018 年 4 月 21 日	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
amazonrootca2	2018 年 4 月 21 日	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
verisignuniversalrootcertificationauthority	2018 年 4 月 21 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
amazonrootca1	2018 年 4 月 21 日	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
networksolutionscertificateauthority	2018 年 4 月 21 日	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
thawteprimaryrootcag3	2018 年 4 月 21 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
affirmtrustnetworking	2018 年 4 月 21 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
thawteprimaryrootcag2	2018 年 4 月 21 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
trustcoreca1	2018 年 4 月 21 日	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
deutschetelekomrootca2	2018 年 4 月 21 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
godaddyrootcertificateauthorityg2	2018 年 4 月 21 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B

名称	日期	SHA1 指纹
entrustrootcertificationauthorityec1	2018 年 4 月 21 日	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
szafirrootca2	2018 年 4 月 21 日	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28:F3:9C:CC:CF:5E:B3:3F:DE
tubitakkamusmsslkoksertifikasisurum1	2018 年 4 月 21 日	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
buypassclass3rootca	2018 年 4 月 21 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
comodorsacertificationauthority	2018 年 4 月 21 日	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
netlockaranyclassgoldfotanusitvany	2018 年 4 月 21 日	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
securitycommunicationrootca2	2018 年 4 月 21 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
dtrustrootclass3ca2ev2009	2018 年 4 月 21 日	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
starfieldclass2ca	2018 年 4 月 21 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
pscprocert	2018 年 4 月 21 日	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
actalisauthenticationrootca	2018 年 4 月 21 日	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
staatdernederlandenrootcag3	2018 年 4 月 21 日	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC

名称	日期	SHA1 指纹
cfcaevroot	2018 年 4 月 21 日	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
digicerttrustedrootg4	2018 年 4 月 21 日	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
staatdernederlandenrootcag2	2018 年 4 月 21 日	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
securitycommunicationevrootca1	2018 年 4 月 21 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
globalsignrootcar3	2018 年 4 月 21 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
globalsignrootcar2	2018 年 4 月 21 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
certumtrustednetworkca2	2018 年 4 月 21 日	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
acraizfnmtrcm	2018 年 4 月 21 日	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
hellenicacademicandresearchinstitutionseccrootca2015	2018 年 4 月 21 日	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
certplusrootcag2	2018 年 4 月 21 日	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:CC:1A
twcarootcertificationauthority	2018 年 4 月 21 日	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
twcaglobalrootca	2018 年 4 月 21 日	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65

名称	日期	SHA1 指纹
certplusrootcag1	2018 年 4 月 21 日	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:F7:66
geotrustuniversalca2	2018 年 4 月 21 日	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
baltimorecybertrustroot	2018 年 4 月 21 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
buypassclass2rootca	2018 年 4 月 21 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
certumtrustednetworkca	2018 年 4 月 21 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
digicertassuredidrootg3	2018 年 4 月 21 日	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
digicertassuredidrootg2	2018 年 4 月 21 日	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F
isrgrootx1	2018 年 4 月 21 日	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36:35:CB:03:9D:43:29:A5:E8
entrustnetpremium2048secureserverca	2018 年 4 月 21 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
certplusclass2primaryca	2018 年 4 月 21 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
digicertglobalrootca	2018 年 4 月 21 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
entrustrootcertificationauthorityg2	2018 年 4 月 21 日	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4

名称	日期	SHA1 指纹
starfieldservicesrootcertificateauthorityg2	2018 年 4 月 21 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
thawteprimaryrootca	2018 年 4 月 21 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
atostrustedroot2011	2018 年 4 月 21 日	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
geotrustglobalca	2018 年 4 月 21 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
luxtrustglobalroot2	2018 年 4 月 21 日	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5F:3F
etugracertificationauthority	2018 年 4 月 21 日	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
visaecommerceroot	2018 年 4 月 21 日	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
quovadisrootca	2018 年 4 月 21 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
identrustcommercialrootca1	2018 年 4 月 21 日	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
staatdernederlandenevrootca	2018 年 4 月 21 日	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
ttelesecglobalrootclass3	2018 年 4 月 21 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
ttelesecglobalrootclass2	2018 年 4 月 21 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9

名称	日期	SHA1 指纹
comodocertificationauthority	2018 年 4 月 21 日	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
securitycommunicationrootca	2018 年 4 月 21 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
quovadisrootca3g3	2018 年 4 月 21 日	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D
xrampglobalcaroot	2018 年 4 月 21 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
securesignrootca11	2018 年 4 月 21 日	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
affirmtrustpremium	2018 年 4 月 21 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
globalsignrootca	2018 年 4 月 21 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
swisssigngoldcag2	2018 年 4 月 21 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
quovadisrootca2g3	2018 年 4 月 21 日	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
affirmtrustpremiumecc	2018 年 4 月 21 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
geotrustprimarycertificationauthority	2018 年 4 月 21 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
quovadisrootca1g3	2018 年 4 月 21 日	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67

名称	日期	SHA1 指纹
hongkongpostrootca1	2018 年 4 月 21 日	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
usertrustecccertificationauthority	2018 年 4 月 21 日	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0
cybertrustglobalroot	2018 年 4 月 21 日	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
godaddyclass2ca	2018 年 4 月 21 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
hellenicacademicandresearchinstitutionsrootca2015	2018 年 4 月 21 日	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
ecacc	2018 年 4 月 21 日	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B:6D:A7:D6:BA:A6:4A:F2:E8
hellenicacademicandresearchinstitutionsrootca2011	2018 年 4 月 21 日	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
verisignclass3publicprimarycertificationauthorityg5	2018 年 4 月 21 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
verisignclass3publicprimarycertificationauthorityg4	2018 年 4 月 21 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A

名称	日期	SHA1 指纹
verisignclass3publicprimarycertificationauthorityg3	2018 年 4 月 21 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
trustisfpsrootca	2018 年 4 月 21 日	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
epkirootcertificationauthority	2018 年 4 月 21 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
globalchambersignroot2008	2018 年 4 月 21 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
camerfirmachambersofcommerceroot	2018 年 4 月 21 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert81.pem	2014 年 3 月 13 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28:A2:59:3A:19:A7:0F:06:9E
mozillacert99.pem	2014 年 3 月 13 日	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
mozillacert145.pem	2014 年 3 月 13 日	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:3A:04
mozillacert37.pem	2014 年 3 月 13 日	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert4.pem	2014 年 3 月 13 日	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B
mozillacert70.pem	2014 年 3 月 13 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
mozillacert88.pem	2014 年 3 月 13 日	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D

名称	日期	SHA1 指纹
mozillacert134.pem	2014 年 3 月 13 日	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
mozillacert26.pem	2014 年 3 月 13 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
mozillacert77.pem	2014 年 3 月 13 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
mozillacert123.pem	2014 年 3 月 13 日	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10:DD:6B:DF:99:72:2C:96:E5
mozillacert15.pem	2014 年 3 月 13 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
mozillacert66.pem	2014 年 3 月 13 日	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34
mozillacert112.pem	2014 年 3 月 13 日	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
mozillacert55.pem	2014 年 3 月 13 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
mozillacert101.pem	2014 年 3 月 13 日	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
mozillacert119.pem	2014 年 3 月 13 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
mozillacert44.pem	2014 年 3 月 13 日	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
mozillacert108.pem	2014 年 3 月 13 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
mozillacert95.pem	2014 年 3 月 13 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57

名称	日期	SHA1 指纹
mozillacert141.pem	2014 年 3 月 13 日	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
mozillacert33.pem	2014 年 3 月 13 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
mozillacert0.pem	2014 年 3 月 13 日	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
mozillacert84.pem	2014 年 3 月 13 日	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2
mozillacert130.pem	2014 年 3 月 13 日	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert148.pem	2014 年 3 月 13 日	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
mozillacert22.pem	2014 年 3 月 13 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
mozillacert7.pem	2014 年 3 月 13 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
mozillacert73.pem	2014 年 3 月 13 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
mozillacert137.pem	2014 年 3 月 13 日	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A1:D1
mozillacert11.pem	2014 年 3 月 13 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
mozillacert29.pem	2014 年 3 月 13 日	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
mozillacert62.pem	2014 年 3 月 13 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B

名称	日期	SHA1 指纹
mozillacert126.pem	2014 年 3 月 13 日	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
mozillacert18.pem	2014 年 3 月 13 日	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
mozillacert51.pem	2014 年 3 月 13 日	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
mozillacert69.pem	2014 年 3 月 13 日	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
mozillacert115.pem	2014 年 3 月 13 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert40.pem	2014 年 3 月 13 日	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
mozillacert58.pem	2014 年 3 月 13 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
mozillacert104.pem	2014 年 3 月 13 日	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E
mozillacert91.pem	2014 年 3 月 13 日	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
mozillacert47.pem	2014 年 3 月 13 日	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
mozillacert80.pem	2014 年 3 月 13 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert98.pem	2014 年 3 月 13 日	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
mozillacert144.pem	2014 年 3 月 13 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27

名称	日期	SHA1 指纹
mozillacert36.pem	2014 年 3 月 13 日	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D
mozillacert3.pem	2014 年 3 月 13 日	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
mozillacert87.pem	2014 年 3 月 13 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert133.pem	2014 年 3 月 13 日	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
mozillacert25.pem	2014 年 3 月 13 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
mozillacert76.pem	2014 年 3 月 13 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert122.pem	2014 年 3 月 13 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
mozillacert14.pem	2014 年 3 月 13 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
mozillacert65.pem	2014 年 3 月 13 日	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93:CA:55:6A:F3:EC:AA:35:FB
mozillacert111.pem	2014 年 3 月 13 日	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
mozillacert129.pem	2014 年 3 月 13 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
mozillacert54.pem	2014 年 3 月 13 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
mozillacert100.pem	2014 年 3 月 13 日	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0

名称	日期	SHA1 指纹
mozillacert118.pem	2014 年 3 月 13 日	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45
mozillacert151.pem	2014 年 3 月 13 日	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
mozillacert43.pem	2014 年 3 月 13 日	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
mozillacert107.pem	2014 年 3 月 13 日	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
mozillacert94.pem	2014 年 3 月 13 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
mozillacert140.pem	2014 年 3 月 13 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert32.pem	2014 年 3 月 13 日	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C
mozillacert83.pem	2014 年 3 月 13 日	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:EA:46
mozillacert147.pem	2014 年 3 月 13 日	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
mozillacert21.pem	2014 年 3 月 13 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
mozillacert39.pem	2014 年 3 月 13 日	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
mozillacert6.pem	2014 年 3 月 13 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
mozillacert72.pem	2014 年 3 月 13 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B

名称	日期	SHA1 指纹
mozillacert136.pem	2014 年 3 月 13 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert10.pem	2014 年 3 月 13 日	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert28.pem	2014 年 3 月 13 日	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
mozillacert61.pem	2014 年 3 月 13 日	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
mozillacert79.pem	2014 年 3 月 13 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert125.pem	2014 年 3 月 13 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert17.pem	2014 年 3 月 13 日	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert50.pem	2014 年 3 月 13 日	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
mozillacert68.pem	2014 年 3 月 13 日	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
mozillacert114.pem	2014 年 3 月 13 日	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert57.pem	2014 年 3 月 13 日	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
mozillacert103.pem	2014 年 3 月 13 日	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert90.pem	2014 年 3 月 13 日	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC

名称	日期	SHA1 指纹
mozillacert46.pem	2014 年 3 月 13 日	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
mozillacert97.pem	2014 年 3 月 13 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
mozillacert143.pem	2014 年 3 月 13 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
mozillacert35.pem	2014 年 3 月 13 日	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
mozillacert2.pem	2014 年 3 月 13 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
mozillacert86.pem	2014 年 3 月 13 日	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
mozillacert132.pem	2014 年 3 月 13 日	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert24.pem	2014 年 3 月 13 日	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
mozillacert9.pem	2014 年 3 月 13 日	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
mozillacert75.pem	2014 年 3 月 13 日	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
mozillacert121.pem	2014 年 3 月 13 日	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
mozillacert139.pem	2014 年 3 月 13 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert13.pem	2014 年 3 月 13 日	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91

名称	日期	SHA1 指纹
mozillacert64.pem	2014 年 3 月 13 日	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
mozillacert110.pem	2014 年 3 月 13 日	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert128.pem	2014 年 3 月 13 日	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D
mozillacert53.pem	2014 年 3 月 13 日	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
mozillacert117.pem	2014 年 3 月 13 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
mozillacert150.pem	2014 年 3 月 13 日	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
mozillacert42.pem	2014 年 3 月 13 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
mozillacert106.pem	2014 年 3 月 13 日	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
mozillacert93.pem	2014 年 3 月 13 日	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17
mozillacert31.pem	2014 年 3 月 13 日	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
mozillacert49.pem	2014 年 3 月 13 日	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71
mozillacert82.pem	2014 年 3 月 13 日	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
mozillacert146.pem	2014 年 3 月 13 日	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A

名称	日期	SHA1 指纹
mozillacert20.pem	2014 年 3 月 13 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert38.pem	2014 年 3 月 13 日	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36
mozillacert5.pem	2014 年 3 月 13 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
mozillacert71.pem	2014 年 3 月 13 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert89.pem	2014 年 3 月 13 日	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
mozillacert135.pem	2014 年 3 月 13 日	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert27.pem	2014 年 3 月 13 日	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
mozillacert60.pem	2014 年 3 月 13 日	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert78.pem	2014 年 3 月 13 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert124.pem	2014 年 3 月 13 日	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert16.pem	2014 年 3 月 13 日	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
mozillacert67.pem	2014 年 3 月 13 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
mozillacert113.pem	2014 年 3 月 13 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31

名称	日期	SHA1 指纹
mozillacert56.pem	2014 年 3 月 13 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
mozillacert102.pem	2014 年 3 月 13 日	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
mozillacert45.pem	2014 年 3 月 13 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert109.pem	2014 年 3 月 13 日	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
mozillacert96.pem	2014 年 3 月 13 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
mozillacert142.pem	2014 年 3 月 13 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
mozillacert34.pem	2014 年 3 月 13 日	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
mozillacert1.pem	2014 年 3 月 13 日	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
mozillacert85.pem	2014 年 3 月 13 日	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
mozillacert131.pem	2014 年 3 月 13 日	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
mozillacert149.pem	2014 年 3 月 13 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert23.pem	2014 年 3 月 13 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
mozillacert8.pem	2014 年 3 月 13 日	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:6A:0F

名称	日期	SHA1 指纹
mozillacert74.pem	2014 年 3 月 13 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert120.pem	2014 年 3 月 13 日	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
mozillacert138.pem	2014 年 3 月 13 日	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
mozillacert12.pem	2014 年 3 月 13 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
mozillacert63.pem	2014 年 3 月 13 日	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
mozillacert127.pem	2014 年 3 月 13 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert19.pem	2014 年 3 月 13 日	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
mozillacert52.pem	2014 年 3 月 13 日	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F
mozillacert116.pem	2014 年 3 月 13 日	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert41.pem	2014 年 3 月 13 日	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
mozillacert59.pem	2014 年 3 月 13 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
mozillacert105.pem	2014 年 3 月 13 日	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC
mozillacert92.pem	2014 年 3 月 13 日	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0

名称	日期	SHA1 指纹
mozillacert30.pem	2014 年 3 月 13 日	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
mozillacert48.pem	2014 年 3 月 13 日	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC
verisignc4g2.pem	2014 年 3 月 20 日	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52
verisignc2g3.pem	2014 年 3 月 20 日	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
verisignc1g6.pem	2014 年 12 月 31 日	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
verisignc2g2.pem	2014 年 3 月 20 日	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
verisignroot.pem	2014 年 3 月 20 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
verisignc2g1.pem	2014 年 3 月 20 日	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:14:2A
verisignc3g5.pem	2014 年 3 月 20 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
verisignc1g3.pem	2014 年 3 月 20 日	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
verisignc3g4.pem	2014 年 3 月 20 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignc1g2.pem	2014 年 3 月 20 日	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
verisignc3g3.pem	2014 年 3 月 20 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6

名称	日期	SHA1 指纹
verisignc1g1.pem	2014 年 3 月 20 日	90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:55:6F
verisignc3g2.pem	2014 年 3 月 20 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
verisignc3g1.pem	2014 年 3 月 20 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
verisignc2g6.pem	2014 年 12 月 31 日	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F
verisignc4g3.pem	2014 年 3 月 20 日	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
gdroot-g2.pem	2014 年 12 月 31 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
gd-class2-root.pem	2014 年 12 月 31 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
gd_bundle-g2.pem	2014 年 12 月 31 日	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8
dstacescax6	2018 年 6 月 18 日	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
gd_bundle-g2.pem	2018 年 6 月 18 日	27:AC:93:69:FA:F2:52:07:BB:26:27:CE:FA:CC:BE:4E:F9:C3:19:B8
verisignc4g3.pem	2018 年 6 月 18 日	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
swisssignplatinumg2ca	2018 年 4 月 21 日	56:E0:FA:C0:3B:8F:18:23:55:18:E5:D3:11:CA:E8:C2:43:31:AB:66

名称	日期	SHA1 指纹
geotrustprimarycertificationauthorityg3	2018 年 6 月 18 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
geotrustprimarycertificationauthorityg2	2018 年 6 月 18 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
buypassclass2rootca	2018 年 6 月 18 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
camerfirmachambersofcommerceroot	2018 年 6 月 18 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1
mozillacert20.pem	2018 年 6 月 18 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert12.pem	2018 年 6 月 18 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
mozillacert90.pem	2018 年 6 月 18 日	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A:CE:19:2B:DD:C7:8E:9C:AC
mozillacert82.pem	2018 年 6 月 18 日	2E:14:DA:EC:28:F0:FA:1E:8E:38:9A:4E:AB:EB:26:C0:0A:D3:83:C3
mozillacert140.pem	2018 年 6 月 18 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert74.pem	2018 年 6 月 18 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
mozillacert132.pem	2018 年 6 月 18 日	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
mozillacert66.pem	2018 年 6 月 18 日	DD:E1:D2:A9:01:80:2E:1D:87:5E:84:B3:80:7E:4B:B1:FD:99:41:34

名称	日期	SHA1 指纹
mozillacert124.pem	2018 年 6 月 18 日	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert58.pem	2018 年 6 月 18 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
securitycommunicationrootca2	2018 年 6 月 18 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert116.pem	2018 年 6 月 18 日	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert108.pem	2018 年 6 月 18 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
certigna	2018 年 6 月 18 日	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert3.pem	2018 年 6 月 18 日	87:9F:4B:EE:05:DF:98:58:3B:E3:60:D6:33:E7:0D:3F:FE:98:71:AF
verisignc1g1.pem	2018 年 6 月 18 日	90:AE:A2:69:85:FF:14:80:4C:43:49:52:EC:E9:60:84:77:AF:55:6F
verisignc4g2.pem	2018 年 6 月 18 日	0B:77:BE:BB:CB:7A:A2:47:05:DE:CC:0F:BD:6A:02:FC:7A:BD:9B:52
deutschetelekomrootca2	2018 年 6 月 18 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
starfieldrootg2ca	2018 年 4 月 21 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
comodoecccertificationauthority	2018 年 6 月 18 日	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
digicertglobalrootg3	2018 年 6 月 18 日	7E:04:DE:89:6A:3E:66:6D:00:E6:87:D3:3F:FA:D9:3B:E8:3D:34:9E

名称	日期	SHA1 指纹
digicertglobalroot g2	2018 年 6 月 18 日	DF:3C:24:F9:BF:D6:66:76:1B:26:80:73: FE:06:D1:CC:8D:4F:82:A4
mozillacert11.pem	2018 年 6 月 18 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E: 4B:DF:B5:A8:99:B2:4D:43
mozillacert81.pem	2018 年 6 月 18 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28: A2:59:3A:19:A7:0F:06:9E
mozillacert73.pem	2018 年 6 月 18 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D: 92:F4:FE:39:D4:E7:0F:0E
szafirrootca2	2018 年 6 月 18 日	E2:52:FA:95:3F:ED:DB:24:60:BD:6E:28: F3:9C:CC:CF:5E:B3:3F:DE
mozillacert131.pem	2018 年 6 月 18 日	37:9A:19:7B:41:85:45:35:0C:A6:03:69: F3:3C:2E:AF:47:4F:20:79
ecacc	2018 年 6 月 18 日	28:90:3A:63:5B:52:80:FA:E6:77:4C:0B: 6D:A7:D6:BA:A6:4A:F2:E8
mozillacert65.pem	2018 年 6 月 18 日	69:BD:8C:F4:9C:D3:00:FB:59:2E:17:93: CA:55:6A:F3:EC:AA:35:FB
turktrustelektroni ksertifik ahizmetsa glayicisih5	2018 年 6 月 18 日	C4:18:F6:4D:46:D1:DF:00:3D:27:30:13: 72:43:A9:12:11:C6:75:FB
mozillacert123.pem	2018 年 6 月 18 日	2A:B6:28:48:5E:78:FB:F3:AD:9E:79:10: DD:6B:DF:99:72:2C:96:E5
mozillacert57.pem	2018 年 6 月 18 日	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F: CB:34:6E:B2:58:B2:8A:58
mozillacert115.pem	2018 年 6 月 18 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62: 32:17:65:CF:17:D8:94:E9

名称	日期	SHA1 指纹
mozillacert49.pem	2018 年 6 月 18 日	61:57:3A:11:DF:0E:D8:7E:D5:92:65:22:EA:D0:56:D7:44:B3:23:71
mozillacert107.pem	2018 年 6 月 18 日	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
verisignclass3g4ca	2018 年 4 月 21 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
securetrustca	2018 年 6 月 18 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
mozillacert2.pem	2018 年 6 月 18 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
buypassclass2ca	2018 年 4 月 21 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
secomscrootca2	2018 年 4 月 21 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
secomscrootca1	2018 年 4 月 21 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
trustisfpsrootca	2018 年 6 月 18 日	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
hongkongpostrootca1	2018 年 6 月 18 日	D6:DA:A8:20:8D:09:D2:15:4D:24:B5:2F:CB:34:6E:B2:58:B2:8A:58
certsignrootca	2018 年 6 月 18 日	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
geotrustprimaryca	2018 年 4 月 21 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
twcaglobalrootca	2018 年 6 月 18 日	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65

名称	日期	SHA1 指纹
camerfirmachambersca	2018 年 4 月 21 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
mozillacert10.pem	2018 年 6 月 18 日	5F:3A:FC:0A:8B:64:F6:86:67:34:74:DF:7E:A9:A2:FE:F9:FA:7A:51
mozillacert80.pem	2018 年 6 月 18 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert72.pem	2018 年 6 月 18 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
comodoaaaca	2018 年 4 月 21 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert130.pem	2018 年 6 月 18 日	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E
mozillacert64.pem	2018 年 6 月 18 日	62:7F:8D:78:27:65:63:99:D2:7D:7F:90:44:C9:FE:B3:F3:3E:FA:9A
mozillacert122.pem	2018 年 6 月 18 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68
mozillacert56.pem	2018 年 6 月 18 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
equifaxsecureebusinessca1	2018 年 4 月 21 日	AE:E6:3D:70:E3:76:FB:C7:3A:EB:B0:A1:C1:D4:C4:7A:A7:40:B3:F4
camerfirmachambersignca	2018 年 4 月 21 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert114.pem	2018 年 6 月 18 日	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert48.pem	2018 年 6 月 18 日	A0:A1:AB:90:C9:FC:84:7B:3B:12:61:E8:97:7D:5F:D3:22:61:D3:CC

名称	日期	SHA1 指纹
pscprocert	2018 年 6 月 18 日	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert106.pem	2018 年 6 月 18 日	E7:A1:90:29:D3:D5:52:DC:0D:0F:C6:92:D3:EA:88:0D:15:2E:1A:6B
mozillacert1.pem	2018 年 6 月 18 日	23:E5:94:94:51:95:F2:41:48:03:B4:D5:64:D2:A3:A3:F5:D8:8B:8C
eecertificationcentrerootca	2018 年 6 月 18 日	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
digicertglobalrootca	2018 年 6 月 18 日	A8:98:5D:3A:65:E5:E5:C4:B2:D7:D6:6D:40:C6:DD:2F:B1:9C:54:36
thawteprimaryrootcag3	2018 年 6 月 18 日	F1:8B:53:8D:1B:E9:03:B6:A6:F0:56:43:5B:17:15:89:CA:F3:6B:F2
thawteprimaryrootcag2	2018 年 6 月 18 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12
entrustrootcertificationauthorityec1	2018 年 6 月 18 日	20:D8:06:40:DF:9B:25:F5:12:25:3A:11:EA:F7:59:8A:EB:14:B5:47
valicertclass2ca	2018 年 4 月 21 日	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
globalchambersignroot2008	2018 年 6 月 18 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
amazonrootca4	2018 年 6 月 18 日	F6:10:84:07:D6:F8:BB:67:98:0C:C2:E2:44:C2:EB:AE:1C:EF:63:BE
gd-class2-root.pem	2018 年 6 月 18 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4

名称	日期	SHA1 指纹
amazonrootca3	2018 年 6 月 18 日	0D:44:DD:8C:3C:8C:1A:1A:58:75:64:81:E9:0F:2E:2A:FF:B3:D2:6E
amazonrootca2	2018 年 6 月 18 日	5A:8C:EF:45:D7:A6:98:59:76:7A:8C:8B:44:96:B5:78:CF:47:4B:1A
securitycommunicationrootca	2018 年 6 月 18 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
amazonrootca1	2018 年 6 月 18 日	8D:A7:F9:65:EC:5E:FC:37:91:0F:1C:6E:59:FD:C1:CC:6A:6E:DE:16
acraizfnmtrcm	2018 年 6 月 18 日	EC:50:35:07:B2:15:C4:95:62:19:E2:A8:9A:5B:42:99:2C:4C:2C:20
quovadisrootca3g3	2018 年 6 月 18 日	48:12:BD:92:3C:A8:C4:39:06:E7:30:6D:27:96:E6:A4:CF:22:2E:7D
certplusrootcag2	2018 年 6 月 18 日	4F:65:8E:1F:E9:06:D8:28:02:E9:54:47:41:C9:54:25:5D:69:CC:1A
certplusrootcag1	2018 年 6 月 18 日	22:FD:D0:B7:FD:A2:4E:0D:AC:49:2C:A0:AC:A6:7B:6A:1F:E3:F7:66
mozillacert71.pem	2018 年 6 月 18 日	4A:BD:EE:EC:95:0D:35:9C:89:AE:C7:52:A1:2C:5B:29:F6:D6:AA:0C
mozillacert63.pem	2018 年 6 月 18 日	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
mozillacert121.pem	2018 年 6 月 18 日	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
ttelesecglobalrootclass3ca	2018 年 4 月 21 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
mozillacert55.pem	2018 年 6 月 18 日	AA:DB:BC:22:23:8F:C4:01:A1:27:BB:38:DD:F4:1D:DB:08:9E:F0:12

名称	日期	SHA1 指纹
mozillacert113.pem	2018 年 6 月 18 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
baltimorecybertrustca	2018 年 4 月 21 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
mozillacert47.pem	2018 年 6 月 18 日	1B:4B:39:61:26:27:6B:64:91:A2:68:6D:D7:02:43:21:2D:1F:1D:96
mozillacert105.pem	2018 年 6 月 18 日	77:47:4F:C6:30:E4:0F:4C:47:64:3F:84:BA:B8:C6:95:4A:8A:41:EC
mozillacert39.pem	2018 年 6 月 18 日	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
usertrustecccertificationauthority	2018 年 6 月 18 日	D1:CB:CA:5D:B2:D5:2A:7F:69:3B:67:4D:E5:F0:5A:1D:0C:95:7D:F0
mozillacert0.pem	2018 年 6 月 18 日	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
securitycommunicationevrootca1	2018 年 6 月 18 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
verisignc3g5.pem	2018 年 6 月 18 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
globalsignr3ca	2018 年 4 月 21 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
trustcoreca1	2018 年 6 月 18 日	58:D1:DF:95:95:67:6B:63:C0:F0:5B:1C:17:4D:8B:84:0B:C8:78:BD
equifaxsecureglobalebusinessca1	2018 年 4 月 21 日	3A:74:CB:7A:47:DB:70:DE:89:1F:24:35:98:64:B8:2D:82:BD:1A:36
geotrustuniversalca	2018 年 6 月 18 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79

名称	日期	SHA1 指纹
affirmtrustpremiumca	2018 年 4 月 21 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
staatdernederlandenrootcag3	2018 年 6 月 18 日	D8:EB:6B:41:51:92:59:E0:F3:E7:85:00:C0:3D:B6:88:97:C9:EE:FC
staatdernederlandenrootcag2	2018 年 6 月 18 日	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
mozillacert70.pem	2018 年 6 月 18 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
secomevrootca1	2018 年 4 月 21 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D
geotrustglobalca	2018 年 6 月 18 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert62.pem	2018 年 6 月 18 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert120.pem	2018 年 6 月 18 日	DA:40:18:8B:91:89:A3:ED:EE:AE:DA:97:FE:2F:9D:F5:B7:D1:8A:41
mozillacert54.pem	2018 年 6 月 18 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
mozillacert112.pem	2018 年 6 月 18 日	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
mozillacert46.pem	2018 年 6 月 18 日	40:9D:4B:D9:17:B5:5C:27:B6:9B:64:CB:98:22:44:0D:CD:09:B8:89
swisssigngoldcag2	2018 年 6 月 18 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert104.pem	2018 年 6 月 18 日	4F:99:AA:93:FB:2B:D1:37:26:A1:99:4A:CE:7F:F0:05:F2:93:5D:1E

名称	日期	SHA1 指纹
mozillacert38.pem	2018 年 6 月 18 日	CB:A1:C5:F8:B0:E3:5E:B8:B9:45:12:D3:F9:34:A2:E9:06:10:D3:36
certplusclass3pprimaryca	2018 年 4 月 21 日	21:6B:2A:29:E6:2A:00:CE:82:01:46:D8:24:41:41:B9:25:11:B2:79
entrustrootcertificationauthorityg2	2018 年 6 月 18 日	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
godaddyrootg2ca	2018 年 4 月 21 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B
cfcaevroot	2018 年 6 月 18 日	E2:B8:29:4B:55:84:AB:6B:58:C2:90:46:6C:AC:3F:B8:39:8F:84:83
verisignc3g4.pem	2018 年 6 月 18 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
geotrustuniversalca2	2018 年 6 月 18 日	37:9A:19:7B:41:85:45:35:0C:A6:03:69:F3:3C:2E:AF:47:4F:20:79
starfieldservicesrootg2ca	2018 年 4 月 21 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F
digicerthighassuranceevrootca	2018 年 6 月 18 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
entrustnetpremium2048secureserverca	2018 年 6 月 18 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31
camerfirmaglobalchambersignroot	2018 年 6 月 18 日	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
verisignclass3g3ca	2018 年 4 月 21 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
godaddyclass2ca	2018 年 6 月 18 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4

名称	日期	SHA1 指纹
mozillacert61.pem	2018 年 6 月 18 日	E0:B4:32:2E:B2:F6:A5:68:B6:54:53:84:48:18:4A:50:36:87:43:84
mozillacert53.pem	2018 年 6 月 18 日	7F:8A:B0:CF:D0:51:87:6A:66:F3:36:0F:47:C8:8D:8C:D3:35:FC:74
atostrustedroot2011	2018 年 6 月 18 日	2B:B1:F5:3E:55:0C:1D:C5:F1:D4:E6:B7:6A:46:4B:55:06:02:AC:21
mozillacert111.pem	2018 年 6 月 18 日	9C:BB:48:53:F6:A4:F6:D3:52:A4:E8:32:52:55:60:13:F5:AD:AF:65
staatdernederlandenevrootca	2018 年 6 月 18 日	76:E2:7E:C1:4F:DB:82:C1:C0:A6:75:B5:05:BE:3D:29:B4:ED:DB:BB
mozillacert45.pem	2018 年 6 月 18 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert103.pem	2018 年 6 月 18 日	70:C1:8D:74:B4:28:81:0A:E4:FD:A5:75:D7:01:9F:99:B0:3D:50:74
mozillacert37.pem	2018 年 6 月 18 日	B1:2E:13:63:45:86:A4:6F:1A:B2:60:68:37:58:2D:C4:AC:FD:94:97
mozillacert29.pem	2018 年 6 月 18 日	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
izenpecom	2018 年 6 月 18 日	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
comodorsacertificationauthority	2018 年 6 月 18 日	AF:E5:D2:44:A8:D1:19:42:30:FF:47:9F:E2:F8:97:BB:CD:7A:8C:B4
mozillacert99.pem	2018 年 6 月 18 日	F1:7F:6F:B6:31:DC:99:E3:A3:C8:7F:FE:1C:F1:81:10:88:D9:60:33
mozillacert149.pem	2018 年 6 月 18 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0:DB:72:2E:31:30:61:F0:B1

名称	日期	SHA1 指纹
utnuserfirstobject ca	2018 年 4 月 21 日	E1:2D:FB:4B:41:D7:D9:C3:2B:30:51:4B: AC:1D:81:D8:38:5E:2D:46
verisignc3g3.pem	2018 年 6 月 18 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3: 39:E2:55:76:60:9B:5C:C6
dstrootcax3	2018 年 6 月 18 日	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1: 73:26:38:CA:6A:D7:7C:13
addtrustexternalro ot	2018 年 6 月 18 日	02:FA:F3:E2:91:43:54:68:60:78:57:69: 4D:F5:E4:5B:68:85:18:68
certumtrustednetwo rkca	2018 年 6 月 18 日	07:E0:32:E0:20:B7:2C:3F:19:2F:06:28: A2:59:3A:19:A7:0F:06:9E
affirmtrustpremium ecc	2018 年 6 月 18 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C: 11:A4:37:CA:EB:FF:C3:BB
starfieldclass2ca	2018 年 6 月 18 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20: 14:C3:D0:E3:37:0E:B5:8A
actalisauthenticat ionrootca	2018 年 6 月 18 日	F3:73:B3:87:06:5A:28:84:8A:F2:F3:4A: CE:19:2B:DD:C7:8E:9C:AC
verisignclass2g3ca	2018 年 4 月 21 日	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0: C3:59:12:AF:9F:EB:63:11
isrgrootx1	2018 年 6 月 18 日	CA:BD:2A:79:A1:07:6A:31:F2:1D:25:36: 35:CB:03:9D:43:29:A5:E8
godaddyrootcertifi cateauthorityg2	2018 年 6 月 18 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62: A7:9F:45:C2:54:FD:E6:8B
mozillacert60.pem	2018 年 6 月 18 日	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8: 5B:B1:C3:65:C7:D8:11:B3
chunghwaepkirootca	2018 年 4 月 21 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4: 56:4B:CF:E2:3D:69:C6:F0

名称	日期	SHA1 指纹
mozillacert52.pem	2018 年 6 月 18 日	8B:AF:4C:9B:1D:F0:2A:92:F7:DA:12:8E:B9:1B:AC:F4:98:60:4B:6F
microseceszignorootca2009	2018 年 6 月 18 日	89:DF:74:FE:5C:F4:0F:4A:80:F9:E3:37:7D:54:DA:91:E1:01:31:8E
securesignrootca11	2018 年 6 月 18 日	3B:C4:9F:48:F8:F3:73:A0:9C:1E:BD:F8:5B:B1:C3:65:C7:D8:11:B3
mozillacert110.pem	2018 年 6 月 18 日	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert44.pem	2018 年 6 月 18 日	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
mozillacert102.pem	2018 年 6 月 18 日	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
mozillacert36.pem	2018 年 6 月 18 日	23:88:C9:D3:71:CC:9E:96:3D:FF:7D:3C:A7:CE:FC:D6:25:EC:19:0D
mozillacert28.pem	2018 年 6 月 18 日	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
baltimorecybertrustroot	2018 年 6 月 18 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
amzninternalrootca	2008 年 12 月 12 日	A7:B7:F6:15:8A:FF:1E:C8:85:13:38:BC:93:EB:A2:AB:A4:09:EF:06
mozillacert98.pem	2018 年 6 月 18 日	C9:A8:B9:E7:55:80:5E:58:E3:53:77:A7:25:EB:AF:C3:7B:27:CC:D7
mozillacert148.pem	2018 年 6 月 18 日	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
verisignc3g2.pem	2018 年 6 月 18 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F

名称	日期	SHA1 指纹
quovadisrootca2g3	2018 年 6 月 18 日	09:3C:61:F3:8B:8B:DC:7D:55:DF:75:38:02:05:00:E1:25:F5:C8:36
geotrustprimarycertificatio nauthority	2018 年 6 月 18 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
opentrustrootcag3	2018 年 6 月 18 日	6E:26:64:F3:56:BF:34:55:BF:D1:93:3F:7C:01:DE:D8:13:DA:8A:A6
opentrustrootcag2	2018 年 6 月 18 日	79:5F:88:60:C5:AB:7C:3D:92:E6:CB:F4:8D:E1:45:CD:11:EF:60:0B
opentrustrootcag1	2018 年 6 月 18 日	79:91:E8:34:F7:E2:EE:DD:08:95:01:52:E9:55:2D:14:E9:58:D5:7E
verisignclass3ca	2018 年 4 月 21 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
globalsignca	2018 年 4 月 21 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
ttelesecglobalrootclass2ca	2018 年 4 月 21 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
verisignclass1g3ca	2018 年 4 月 21 日	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
verisignuniversalrootca	2018 年 4 月 21 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
soneraclass2ca	2018 年 4 月 21 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
starfieldservicesrootcertificateauthorityg2	2018 年 6 月 18 日	92:5A:8F:8D:2C:6D:04:E0:66:5F:59:6A:FF:22:D8:63:E8:25:6F:3F

名称	日期	SHA1 指纹
mozillacert51.pem	2018 年 6 月 18 日	FA:B7:EE:36:97:26:62:FB:2D:B0:2A:F6:BF:03:FD:E8:7C:4B:2F:9B
mozillacert43.pem	2018 年 6 月 18 日	F9:CD:0E:2C:DA:76:24:C1:8F:BD:F0:F0:AB:B6:45:B8:F7:FE:D5:7A
mozillacert101.pem	2018 年 6 月 18 日	99:A6:9B:E6:1A:FE:88:6B:4D:2B:82:00:7C:B8:54:FC:31:7E:15:39
mozillacert35.pem	2018 年 6 月 18 日	2A:C8:D5:8B:57:CE:BF:2F:49:AF:F2:FC:76:8F:51:14:62:90:7A:41
globalsignr2ca	2018 年 4 月 21 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
mozillacert27.pem	2018 年 6 月 18 日	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
affirmtrustpremium	2018 年 6 月 18 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert19.pem	2018 年 6 月 18 日	B4:35:D4:E1:11:9D:1C:66:90:A7:49:EB:B3:94:BD:63:7B:A7:82:B7
mozillacert97.pem	2018 年 6 月 18 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
netlockaranyclassgoldfotanusitvany	2018 年 6 月 18 日	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
mozillacert89.pem	2018 年 6 月 18 日	C8:EC:8C:87:92:69:CB:4B:AB:39:E9:8D:7E:57:67:F3:14:95:73:9D
verisignroot.pem	2018 年 6 月 18 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
mozillacert147.pem	2018 年 6 月 18 日	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4

名称	日期	SHA1 指纹
aolrootca2	2018 年 4 月 21 日	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
cia-crt-g3-01-ca	2016 年 11 月 23 日	2B:EE:2C:BA:A3:1D:B5:FE:60:40:41:95:08:ED:46:82:39:4D:ED:E2
aolrootca1	2018 年 4 月 21 日	39:21:C1:15:C1:5D:0E:CA:5C:CB:5B:C4:F0:7D:21:D8:05:0B:56:6A
verisignc3g1.pem	2018 年 6 月 18 日	A1:DB:63:93:91:6F:17:E4:18:55:09:40:04:15:C7:02:40:B0:AE:6B
mozillacert139.pem	2018 年 6 月 18 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
soneraclass2rootca	2018 年 6 月 18 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
swisssignsilverg2ca	2018 年 4 月 21 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
thawteprimaryrootca	2018 年 6 月 18 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
gdcatrustauthr5root	2018 年 6 月 18 日	0F:36:38:5B:81:1A:25:C3:9B:31:4E:83:CA:E9:34:66:70:CC:74:B4
trustcenterclass4caii	2018 年 4 月 21 日	A6:9A:91:FD:05:7F:13:6A:42:63:0B:B1:76:0D:2D:51:12:0C:16:50
usertrustrsacertificationauthority	2018 年 6 月 18 日	2B:8F:1B:57:33:0D:BB:A2:D0:7A:6C:51:F7:0E:E9:0D:DA:B9:AD:8E
digicertassuredidrootg3	2018 年 6 月 18 日	F5:17:A2:4F:9A:48:C6:C9:F8:A2:00:26:9F:DC:0F:48:2C:AB:30:89
digicertassuredidrootg2	2018 年 6 月 18 日	A1:4B:48:D9:43:EE:0A:0E:40:90:4F:3C:E0:A4:C0:91:93:51:5D:3F

名称	日期	SHA1 指纹
mozillacert50.pem	2018 年 6 月 18 日	8C:96:BA:EB:DD:2B:07:07:48:EE:30:32:66:A0:F3:98:6E:7C:AE:58
mozillacert42.pem	2018 年 6 月 18 日	85:A4:08:C0:9C:19:3E:5D:51:58:7D:CD:D6:13:30:FD:8C:DE:37:BF
mozillacert100.pem	2018 年 6 月 18 日	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
mozillacert34.pem	2018 年 6 月 18 日	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
affirmtrustcommercialca	2018 年 4 月 21 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert26.pem	2018 年 6 月 18 日	87:82:C6:C3:04:35:3B:CF:D2:96:92:D2:59:3E:7D:44:D9:34:FF:11
globalsigneccrootcar5	2018 年 6 月 18 日	1F:24:C6:30:CD:A4:18:EF:20:69:FF:AD:4F:DD:5F:46:3A:1B:69:AA
globalsigneccrootcar4	2018 年 6 月 18 日	69:69:56:2E:40:80:F4:24:A1:E7:19:9F:14:BA:F3:EE:58:AB:6A:BB
buypassclass3rootca	2018 年 6 月 18 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
mozillacert18.pem	2018 年 6 月 18 日	79:98:A3:08:E1:4D:65:85:E6:C2:1E:15:3A:71:9F:BA:5A:D3:4A:D9
mozillacert96.pem	2018 年 6 月 18 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
verisignc2g6.pem	2018 年 6 月 18 日	40:B3:31:A0:E9:BF:E8:55:BC:39:93:CA:70:4F:4E:C2:51:D4:1D:8F
secomvalicertclass1ca	2018 年 4 月 21 日	E5:DF:74:3C:B6:01:C4:9B:98:43:DC:AB:8C:E8:6A:81:10:9F:E4:8E

名称	日期	SHA1 指纹
mozillacert88.pem	2018 年 6 月 18 日	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
accvraiz1	2018 年 6 月 18 日	93:05:7A:88:15:C6:4F:CE:88:2F:FA:91:16:52:28:78:BC:53:64:17
mozillacert146.pem	2018 年 6 月 18 日	21:FC:BD:8E:7F:6C:AF:05:1B:D1:B3:43:EC:A8:E7:61:47:F2:0F:8A
mozillacert138.pem	2018 年 6 月 18 日	E1:9F:E3:0E:8B:84:60:9E:80:9B:17:0D:72:A8:C5:BA:6E:14:09:BD
verisignclass3g2ca	2018 年 4 月 21 日	85:37:1C:A6:E5:50:14:3D:CE:28:03:47:1B:DE:3A:09:E8:F8:77:0F
dtrustrootclass3ca2ev2009	2018 年 6 月 18 日	96:C9:1B:0B:95:B4:10:98:42:FA:D0:D8:22:79:FE:60:FA:B9:16:83
xrampglobalca	2018 年 4 月 21 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
mozillacert9.pem	2018 年 6 月 18 日	F4:8B:11:BF:DE:AB:BE:94:54:20:71:E6:41:DE:6B:BE:88:2B:40:B9
verisignuniversalrootcertificationauthority	2018 年 6 月 18 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
tubitakkamusmsslkoksertifikasisurum1	2018 年 6 月 18 日	31:43:64:9B:EC:CE:27:EC:ED:3A:3F:0B:8F:0D:E4:E8:91:DD:EE:CA
mozillacert41.pem	2018 年 6 月 18 日	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3
mozillacert33.pem	2018 年 6 月 18 日	FE:B8:C4:32:DC:F9:76:9A:CE:AE:3D:D8:90:8F:FD:28:86:65:64:7D

名称	日期	SHA1 指纹
mozillacert25.pem	2018 年 6 月 18 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
mozillacert17.pem	2018 年 6 月 18 日	40:54:DA:6F:1C:3F:40:74:AC:ED:0F:EC:CD:DB:79:D1:53:FB:90:1D
mozillacert95.pem	2018 年 6 月 18 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57
affirmtrustpremiumeccca	2018 年 4 月 21 日	B8:23:6B:00:2F:1D:16:86:53:01:55:6C:11:A4:37:CA:EB:FF:C3:BB
mozillacert87.pem	2018 年 6 月 18 日	5F:3B:8C:F2:F8:10:B3:7D:78:B4:CE:EC:19:19:C3:73:34:B9:C7:74
mozillacert145.pem	2018 年 6 月 18 日	10:1D:FA:3F:D5:0B:CB:BB:9B:B5:60:0C:19:55:A4:1A:F4:73:3A:04
mozillacert79.pem	2018 年 6 月 18 日	D8:A6:33:2C:E0:03:6F:B1:85:F6:63:4F:7D:6A:06:65:26:32:28:27
mozillacert137.pem	2018 年 6 月 18 日	4A:65:D5:F4:1D:EF:39:B8:B8:90:4A:4A:D3:64:81:33:CF:C7:A1:D1
digicertassuredidrootca	2018 年 6 月 18 日	05:63:B8:63:0D:62:D7:5A:BB:C8:AB:1E:4B:DF:B5:A8:99:B2:4D:43
addtrustqualifiedca	2018 年 4 月 21 日	4D:23:78:EC:91:95:39:B5:00:7F:75:8F:03:3B:21:1E:C5:4D:8B:CF
mozillacert129.pem	2018 年 6 月 18 日	E6:21:F3:35:43:79:05:9A:4B:68:30:9D:8A:2F:74:22:15:87:EC:79
verisignclass2g2ca	2018 年 4 月 21 日	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
baltimorecodesigningca	2018 年 4 月 21 日	30:46:D8:C8:88:FF:69:30:C3:4A:FC:CD:49:27:08:7C:60:56:7B:0D

名称	日期	SHA1 指纹
luxtrustglobalroot2	2018 年 6 月 18 日	1E:0E:56:19:0A:D1:8B:25:98:B2:04:44:FF:66:8A:04:17:99:5F:3F
visaecommerceroot	2018 年 6 月 18 日	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62
oistewisekeyglobalrootgbca	2018 年 6 月 18 日	0F:F9:40:76:18:D3:D7:6A:4B:98:F0:A8:35:9E:0C:FD:27:AC:CC:ED
mozillacert8.pem	2018 年 6 月 18 日	3E:2B:F7:F2:03:1B:96:F3:8C:E6:C4:D8:A8:5D:3E:2D:58:47:6A:0F
comodocertificationauthority	2018 年 6 月 18 日	66:31:BF:9E:F7:4F:9E:B6:C9:D5:A6:0C:BA:6A:BE:D1:F7:BD:EF:7B
cia-crt-g3-02-ca	2016 年 11 月 23 日	96:4A:BB:A7:BD:DA:FC:97:34:C0:0A:2D:F0:05:98:F7:E6:C6:6F:09
verisignc1g6.pem	2018 年 6 月 18 日	51:7F:61:1E:29:91:6B:53:82:FB:72:E7:44:D9:8D:C3:CC:53:6D:64
trustcenterclass2caii	2018 年 4 月 21 日	AE:50:83:ED:7C:F4:5C:BC:8F:61:C6:21:FE:68:5D:79:42:21:15:6E
quovadisrootca1g3	2018 年 6 月 18 日	1B:8E:EA:57:96:29:1A:C9:39:EA:B8:0A:81:1A:73:73:C0:93:79:67
mozillacert40.pem	2018 年 6 月 18 日	80:25:EF:F4:6E:70:C8:D4:72:24:65:84:FE:40:3B:8A:8D:6A:DB:F5
cadisigrootr2	2018 年 6 月 18 日	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
cadisigrootr1	2018 年 6 月 18 日	8E:1C:74:F8:A6:20:B9:E5:8A:F4:61:FA:EC:2B:47:56:51:1A:52:C6
mozillacert32.pem	2018 年 6 月 18 日	60:D6:89:74:B5:C2:65:9E:8A:0F:C1:88:7C:88:D2:46:69:1B:18:2C

名称	日期	SHA1 指纹
utndatacorpsgcca	2018 年 4 月 21 日	58:11:9F:0E:12:82:87:EA:50:FD:D9:87:45:6F:4F:78:DC:FA:D6:D4
mozillacert24.pem	2018 年 6 月 18 日	59:AF:82:79:91:86:C7:B4:75:07:CB:CF:03:57:46:EB:04:DD:B7:16
addtrustclass1ca	2018 年 4 月 21 日	CC:AB:0E:A0:4C:23:01:D6:69:7B:DD:37:9F:CD:12:EB:24:E3:94:9D
mozillacert16.pem	2018 年 6 月 18 日	DA:C9:02:4F:54:D8:F6:DF:94:93:5F:B1:73:26:38:CA:6A:D7:7C:13
affirmtrustnetworkingca	2018 年 4 月 21 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert94.pem	2018 年 6 月 18 日	49:0A:75:74:DE:87:0A:47:FE:58:EE:F6:C7:6B:EB:C6:0B:12:40:99
mozillacert86.pem	2018 年 6 月 18 日	74:2C:31:92:E6:07:E4:24:EB:45:49:54:2B:E1:BB:C5:3E:61:74:E2
mozillacert144.pem	2018 年 6 月 18 日	37:F7:6D:E6:07:7C:90:C5:B1:3E:93:1A:B7:41:10:B4:F2:E4:9A:27
mozillacert78.pem	2018 年 6 月 18 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
mozillacert136.pem	2018 年 6 月 18 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert128.pem	2018 年 6 月 18 日	A9:E9:78:08:14:37:58:88:F2:05:19:B0:6D:2B:0D:2B:60:16:90:7D
verisignclass1g2ca	2018 年 4 月 21 日	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47

名称	日期	SHA1 指纹
hellenicacademicandresearchinstitutionsrootca2015	2018 年 6 月 18 日	01:0C:06:95:A6:98:19:14:FF:BF:5F:C6:B0:B6:95:EA:29:E9:12:A6
soneraclass1ca	2018 年 4 月 21 日	07:47:22:01:99:CE:74:B9:7C:B0:3D:79:B2:64:A2:C8:55:E9:33:FF
hellenicacademicandresearchinstitutionsrootca2011	2018 年 6 月 18 日	FE:45:65:9B:79:03:5B:98:A1:61:B5:51:2E:AC:DA:58:09:48:22:4D
certumtrustednetworkca2	2018 年 6 月 18 日	D3:DD:48:3E:2B:BF:4C:05:E8:AF:10:F5:FA:76:26:CF:D3:DC:30:92
equifaxsecureca	2018 年 4 月 21 日	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
thawteserverca	2018 年 4 月 21 日	9F:AD:91:A6:CE:6A:C6:C5:00:47:C4:4E:C9:D4:A5:0D:92:D8:49:79
mozillacert7.pem	2018 年 6 月 18 日	AD:7E:1C:28:B0:64:EF:8F:60:03:40:20:14:C3:D0:E3:37:0E:B5:8A
affirmtrustnetworking	2018 年 6 月 18 日	29:36:21:02:8B:20:ED:02:F5:66:C5:32:D1:D6:ED:90:9F:45:00:2F
deprecateditsecca	2012 年 1 月 27 日	12:12:0B:03:0E:15:14:54:F4:DD:B3:F5:DE:13:6E:83:5A:29:72:9D
globalsignrootcar3	2018 年 6 月 18 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
globalsignrootcar2	2018 年 6 月 18 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE

名称	日期	SHA1 指纹
quovadisrootca	2018 年 6 月 18 日	DE:3F:40:BD:50:93:D3:9B:6C:60:F6:DA:BC:07:62:01:00:89:76:C9
mozillacert31.pem	2018 年 6 月 18 日	9F:74:4E:9F:2B:4D:BA:EC:0F:31:2C:50:B6:56:3B:8E:2D:93:C3:11
entrustrootcertificationauthority	2018 年 6 月 18 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert23.pem	2018 年 6 月 18 日	91:C6:D6:EE:3E:8A:C8:63:84:E5:48:C2:99:29:5C:75:6C:81:7B:81
mozillacert15.pem	2018 年 6 月 18 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
verisignc2g3.pem	2018 年 6 月 18 日	61:EF:43:D7:7F:CA:D4:61:51:BC:98:E0:C3:59:12:AF:9F:EB:63:11
mozillacert93.pem	2018 年 6 月 18 日	31:F1:FD:68:22:63:20:EE:C6:3B:3F:9D:EA:4A:3E:53:7C:7C:39:17
mozillacert151.pem	2018 年 6 月 18 日	AC:ED:5F:65:53:FD:25:CE:01:5F:1F:7A:48:3B:6A:74:9F:61:78:C6
mozillacert85.pem	2018 年 6 月 18 日	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
certplusclass2primaryca	2018 年 6 月 18 日	74:20:74:41:72:9C:DD:92:EC:79:31:D8:23:10:8D:C2:81:92:E2:BB
mozillacert143.pem	2018 年 6 月 18 日	36:B1:2B:49:F9:81:9E:D7:4C:9E:BC:38:0F:C6:56:8F:5D:AC:B2:F7
mozillacert77.pem	2018 年 6 月 18 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
mozillacert135.pem	2018 年 6 月 18 日	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18

名称	日期	SHA1 指纹
mozillacert69.pem	2018 年 6 月 18 日	2F:78:3D:25:52:18:A7:4A:65:39:71:B5:2C:A2:9C:45:15:6F:E9:19
mozillacert127.pem	2018 年 6 月 18 日	DE:28:F4:A4:FF:E5:B9:2F:A3:C5:03:D1:A3:49:A7:F9:96:2A:82:12
mozillacert119.pem	2018 年 6 月 18 日	75:E0:AB:B6:13:85:12:27:1C:04:F8:5F:DD:DE:38:E4:B7:24:2E:FE
geotrustprimarycag3	2018 年 4 月 21 日	03:9E:ED:B8:0B:E7:A0:3C:69:53:89:3B:20:D2:D9:32:3A:4C:2A:FD
identrustpublicsectorrootca1	2018 年 6 月 18 日	BA:29:41:60:77:98:3F:F4:F3:EF:F2:31:05:3B:2E:EA:6D:4D:45:FD
geotrustprimarycag2	2018 年 4 月 21 日	8D:17:84:D5:37:F3:03:7D:EC:70:FE:57:8B:51:9A:99:E6:10:D7:B0
trustcorrootcertca2	2018 年 6 月 18 日	B8:BE:6D:CB:56:F1:55:B9:63:D4:12:CA:4E:06:34:C7:94:B2:1C:C0
mozillacert6.pem	2018 年 6 月 18 日	27:96:BA:E6:3F:18:01:E2:77:26:1B:A0:D7:77:70:02:8F:20:EE:E4
trustcorrootcertca1	2018 年 6 月 18 日	FF:BD:CD:E7:82:C8:43:5E:3C:6F:26:86:5C:CA:A8:3A:45:5B:C3:0A
networksolutionscertificate authority	2018 年 6 月 18 日	74:F8:A3:C3:EF:E7:B3:90:06:4B:83:90:3C:21:64:60:20:E5:DF:CE
twcarootcertificationauthority	2018 年 6 月 18 日	CF:9E:87:6D:D3:EB:FC:42:26:97:A3:B5:A3:7A:A0:76:A9:06:23:48
addtrustexternalca	2018 年 4 月 21 日	02:FA:F3:E2:91:43:54:68:60:78:57:69:4D:F5:E4:5B:68:85:18:68

名称	日期	SHA1 指纹
verisignclass3g5ca	2018 年 4 月 21 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
autoridaddecertificacionfirmaprofesionalcifa62634068	2018 年 6 月 18 日	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
hellenicacademicandresearchinstitutionseccrootca2015	2018 年 6 月 18 日	9F:F1:71:8D:92:D5:9A:F3:7D:74:97:B4:BC:6F:84:68:0B:BA:B6:66
verisigntsaca	2018 年 4 月 21 日	20:CE:B1:F0:F5:1C:0E:19:A9:F3:8D:B1:AA:8E:03:8C:AA:7A:C7:01
utnuserfirsthardwareca	2018 年 4 月 21 日	04:83:ED:33:99:AC:36:08:05:87:22:ED:BC:5E:46:00:E3:BE:F9:D7
identrustcommercialrootca1	2018 年 6 月 18 日	DF:71:7E:AA:4A:D9:4E:C9:55:84:99:60:2D:48:DE:5F:BC:F0:3A:25
dtrustrootclass3ca22009	2018 年 6 月 18 日	58:E8:AB:B0:36:15:33:FB:80:F7:9B:1B:6D:29:D3:FF:8D:5F:00:F0
epkirootcertificationauthority	2018 年 6 月 18 日	67:65:0D:F1:7E:8E:7E:5B:82:40:A4:F4:56:4B:CF:E2:3D:69:C6:F0
mozillacert30.pem	2018 年 6 月 18 日	E7:B4:F6:9D:61:EC:90:69:DB:7E:90:A7:40:1A:3C:F4:7D:4F:E8:EE
teliasonerarootcav1	2018 年 6 月 18 日	43:13:BB:96:F1:D5:86:9B:C1:4E:6A:92:F6:CF:F6:34:69:87:82:37
buypassclass3ca	2018 年 4 月 21 日	DA:FA:F7:FA:66:84:EC:06:8F:14:50:BD:C7:C2:81:A5:BC:A9:64:57

名称	日期	SHA1 指纹
mozillacert22.pem	2018 年 6 月 18 日	32:3C:11:8E:1B:F7:B8:B6:52:54:E2:E2:10:0D:D6:02:90:37:F0:96
mozillacert14.pem	2018 年 6 月 18 日	5F:B7:EE:06:33:E2:59:DB:AD:0C:4C:9A:E6:D3:8F:1A:61:C7:DC:25
verisignc2g2.pem	2018 年 6 月 18 日	B3:EA:C4:47:76:C9:C8:1C:EA:F2:9D:95:B6:CC:A0:08:1B:67:EC:9D
certumca	2018 年 4 月 21 日	62:52:DC:40:F7:11:43:A2:2F:DE:9E:F7:34:8E:06:42:51:B1:81:18
mozillacert92.pem	2018 年 6 月 18 日	A3:F1:33:3F:E2:42:BF:CF:C5:D1:4E:8F:39:42:98:40:68:10:D1:A0
mozillacert150.pem	2018 年 6 月 18 日	33:9B:6B:14:50:24:9B:55:7A:01:87:72:84:D9:E0:2F:C3:D2:D8:E9
mozillacert84.pem	2018 年 6 月 18 日	D3:C0:63:F2:19:ED:07:3E:34:AD:5D:75:0B:32:76:29:FF:D5:9A:F2
ttelesecglobalrootclass3	2018 年 6 月 18 日	55:A6:72:3E:CB:F2:EC:CD:C3:23:74:70:19:9D:2A:BE:11:E3:81:D1
globalsignrootca	2018 年 6 月 18 日	B1:BC:96:8B:D4:F4:9D:62:2A:A8:9A:81:F2:15:01:52:A4:1D:82:9C
ttelesecglobalrootclass2	2018 年 6 月 18 日	59:0D:2D:7D:88:4F:40:2E:61:7E:A5:62:32:17:65:CF:17:D8:94:E9
mozillacert142.pem	2018 年 6 月 18 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
mozillacert76.pem	2018 年 6 月 18 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
mozillacert134.pem	2018 年 6 月 18 日	70:17:9B:86:8C:00:A4:FA:60:91:52:22:3F:9F:3E:32:BD:E0:05:62

名称	日期	SHA1 指纹
mozillacert68.pem	2018 年 6 月 18 日	AE:C5:FB:3F:C8:E1:BF:C4:E5:4F:03:07:5A:9A:E8:00:B7:F7:B6:FA
etugracertificationauthority	2018 年 6 月 18 日	51:C6:E7:08:49:06:6E:F3:92:D4:5C:A0:0D:6D:A3:62:8F:C3:52:39
mozillacert126.pem	2018 年 6 月 18 日	25:01:90:19:CF:FB:D9:99:1C:B7:68:25:74:8D:94:5F:30:93:95:42
keynectisrootca	2018 年 4 月 21 日	9C:61:5C:4D:4D:85:10:3A:53:26:C2:4D:BA:EA:E4:A2:D2:D5:CC:97
mozillacert118.pem	2018 年 6 月 18 日	7E:78:4A:10:1C:82:65:CC:2D:E1:F1:6D:47:B4:40:CA:D9:0A:19:45
quovadisrootca3	2018 年 6 月 18 日	1F:49:14:F7:D8:74:95:1D:DD:AE:02:C0:BE:FD:3A:2D:82:75:51:85
quovadisrootca2	2018 年 6 月 18 日	CA:3A:FB:CF:12:40:36:4B:44:B2:16:20:88:80:48:39:19:93:7C:F7
mozillacert5.pem	2018 年 6 月 18 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
verisignc1g3.pem	2018 年 6 月 18 日	20:42:85:DC:F7:EB:76:41:95:57:8E:13:6B:D4:B7:D1:E9:8E:46:A5
cybertrustglobalroot	2018 年 6 月 18 日	5F:43:E5:B1:BF:F8:78:8C:AC:1C:C7:CA:4A:9A:C6:22:2B:CC:34:C6
amzninternalinfosecag3	2015 年 2 月 27 日	B9:B1:CA:38:F7:BF:9C:D2:D4:95:E7:B6:5E:75:32:9B:A8:78:2E:F6
starfieldrootcertificateauthorityg2	2018 年 6 月 18 日	B5:1C:06:7C:EE:2B:0C:3D:F8:55:AB:2D:92:F4:FE:39:D4:E7:0F:0E
entrust2048ca	2018 年 4 月 21 日	50:30:06:09:1D:97:D4:F5:AE:39:F7:CB:E7:92:7D:7D:65:2D:34:31

名称	日期	SHA1 指纹
swisssignsilvercag2	2018 年 6 月 18 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
affirmtrustcommercial	2018 年 6 月 18 日	F9:B5:B6:32:45:5F:9C:BE:EC:57:5F:80:DC:E9:6E:2C:C7:B2:78:B7
certinomisrootca	2018 年 6 月 18 日	9D:70:BB:01:A5:A4:A0:18:11:2E:F7:1C:01:B9:32:C5:34:E7:88:A8
xrampglobalcaroot	2018 年 6 月 18 日	B8:01:86:D1:EB:9C:86:A5:41:04:CF:30:54:F3:4C:52:B7:E5:58:C6
secureglobalca	2018 年 6 月 18 日	3A:44:73:5A:E5:81:90:1F:24:86:61:46:1E:3B:9C:C4:5F:F5:3A:1B
swisssigngoldg2ca	2018 年 4 月 21 日	D8:C5:38:8A:B7:30:1B:1B:6E:D4:7A:E6:45:25:3A:6F:9F:1A:27:61
mozillacert21.pem	2018 年 6 月 18 日	9B:AA:E5:9F:56:EE:21:CB:43:5A:BE:25:93:DF:A7:F0:40:D1:1D:CB
mozillacert13.pem	2018 年 6 月 18 日	06:08:3F:59:3F:15:A1:04:A0:69:A4:6B:A9:03:D0:06:B7:97:09:91
verisignc2g1.pem	2018 年 6 月 18 日	67:82:AA:E0:ED:EE:E2:1A:58:39:D3:C0:CD:14:68:0A:4F:60:14:2A
mozillacert91.pem	2018 年 6 月 18 日	3B:C0:38:0B:33:C3:F6:A6:0C:86:15:22:93:D9:DF:F5:4B:81:C0:04
oistewisekeyglobalrootgaca	2018 年 6 月 18 日	59:22:A1:E1:5A:EA:16:35:21:F8:98:39:6A:46:46:B0:44:1B:0F:A9
mozillacert83.pem	2018 年 6 月 18 日	A0:73:E5:C5:BD:43:61:0D:86:4C:21:13:0A:85:58:57:CC:9C:EA:46
entrustevca	2018 年 4 月 21 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9

名称	日期	SHA1 指纹
mozillacert141.pem	2018 年 6 月 18 日	31:7A:2A:D0:7F:2B:33:5E:F5:A1:C3:4E:4B:57:E8:B7:D8:F1:FC:A6
mozillacert75.pem	2018 年 6 月 18 日	D2:32:09:AD:23:D3:14:23:21:74:E4:0D:7F:9D:62:13:97:86:63:3A
mozillacert133.pem	2018 年 6 月 18 日	85:B5:FF:67:9B:0C:79:96:1F:C8:6E:44:22:00:46:13:DB:17:92:84
mozillacert67.pem	2018 年 6 月 18 日	D6:9B:56:11:48:F0:1C:77:C5:45:78:C1:09:26:DF:5B:85:69:76:AD
mozillacert125.pem	2018 年 6 月 18 日	B3:1E:B1:B7:40:E3:6C:84:02:DA:DC:37:D4:4D:F5:D4:67:49:52:F9
mozillacert59.pem	2018 年 6 月 18 日	36:79:CA:35:66:87:72:30:4D:30:A5:FB:87:3B:0F:A7:7B:B7:0D:54
thawtepremiumserverca	2018 年 4 月 21 日	E0:AB:05:94:20:72:54:93:05:60:62:02:36:70:F7:CD:2E:FC:66:66
mozillacert117.pem	2018 年 6 月 18 日	D4:DE:20:D0:5E:66:FC:53:FE:1A:50:88:2C:78:DB:28:52:CA:E4:74
utnuserfirstclientauthemailca	2018 年 4 月 21 日	B1:72:B1:A5:6D:95:F9:1F:E5:02:87:E1:4D:37:EA:6A:44:63:76:8A
entrustrootcag2	2018 年 4 月 21 日	8C:F4:27:FD:79:0C:3A:D1:66:06:8D:E8:1E:57:EF:BB:93:22:72:D4
mozillacert109.pem	2018 年 6 月 18 日	B5:61:EB:EA:A4:DE:E4:25:4B:69:1A:98:A5:57:47:C2:34:C7:D9:71
digicerttrustedrootg4	2018 年 6 月 18 日	DD:FB:16:CD:49:31:C9:73:A2:03:7D:3F:C8:3A:4D:7D:77:5D:05:E4
gdroot-g2.pem	2018 年 6 月 18 日	47:BE:AB:C9:22:EA:E8:0E:78:78:34:62:A7:9F:45:C2:54:FD:E6:8B

名称	日期	SHA1 指纹
comodoaaaservicesroot	2018 年 6 月 18 日	D1:EB:23:A4:6D:17:D6:8F:D9:25:64:C2:F1:F1:60:17:64:D8:E3:49
mozillacert4.pem	2018 年 6 月 18 日	E3:92:51:2F:0A:CF:F5:05:DF:F6:DE:06:7F:75:37:E1:65:EA:57:4B
verisignclass3publicprimarycertificationauthorityg5	2018 年 6 月 18 日	4E:B6:D5:78:49:9B:1C:CF:5F:58:1E:AD:56:BE:3D:9B:67:44:A5:E5
chambersofcommerceroot2008	2018 年 6 月 18 日	78:6A:74:AC:76:AB:14:7F:9C:6A:30:50:BA:9E:A8:7E:FE:9A:CE:3C
verisignclass3publicprimarycertificationauthorityg4	2018 年 6 月 18 日	22:D5:D8:DF:8F:02:31:D1:8D:F7:9D:B7:CF:8A:2D:64:C9:3F:6C:3A
verisignclass3publicprimarycertificationauthorityg3	2018 年 6 月 18 日	13:2D:0D:45:53:4B:69:97:CD:B2:D5:C3:39:E2:55:76:60:9B:5C:C6
thawtepersonalfreemailca	2018 年 4 月 21 日	E6:18:83:AE:84:CA:C1:C1:CD:52:AD:E8:E9:25:2B:45:A6:4F:B7:E2
verisignc1g2.pem	2018 年 6 月 18 日	27:3E:E1:24:57:FD:C4:F9:0C:55:E8:2B:56:16:7F:62:F5:32:E5:47
gtecybertrustglobalca	2018 年 4 月 21 日	97:81:79:50:D8:1C:96:70:CC:34:D8:09:CF:79:44:31:36:7E:F4:74
trustcenteruniversalcai	2018 年 4 月 21 日	6B:2F:34:AD:89:58:BE:62:FD:B0:6B:5C:CE:BB:9D:D9:4F:4E:39:F3

名称	日期	SHA1 指纹
camerfirmachambers commerceca	2018 年 4 月 21 日	6E:3A:55:A4:19:0C:19:5C:93:84:3C:C0: DB:72:2E:31:30:61:F0:B1
verisignclass1ca	2018 年 4 月 21 日	CE:6A:64:A3:09:E4:2F:BB:D9:85:1C:45: 3E:64:09:EA:E8:7D:60:F1

解析器映射模板更改日志

> **Note**
>
> 我们现在主要支持 APPSYNC_JS 运行时环境及其文档。请考虑使用 APPSYNC_JS 运行时环境和此处的指南。

解析器和函数映射模板已版本化。映射模板版本（例如 2018-05-29）规定了以下内容：* 请求模板提供的数据源请求配置的预期形状 * 请求映射模板和响应映射模板的执行行为

版本使用 YYYY-MM-DD 格式表示，较晚的日期对应于更新版本。该页面列出了 AWS AppSync 中当前支持的映射模板版本之间的差异。

主题

每个版本矩阵的数据源操作可用性

支持的操作/版本	2017-02-28	2018-05-29
AWS Lambda Invoke	是	是

支持的操作/版本	2017-02-28	2018-05-29
AWS Lambda BatchInvoke	是	是
None Datasource	是	是
Amazon OpenSearch GET	是	是
Amazon OpenSearch POST	是	是
Amazon OpenSearch PUT	是	是
Amazon OpenSearch DELETE	是	是
Amazon OpenSearch GET	是	是
DynamoDB GetItem	是	是
DynamoDB Scan	是	是
DynamoDB Query	是	是
DynamoDB DeleteItem	是	是
DynamoDB PutItem	是	是
DynamoDB BatchGetItem	否	是
DynamoDB BatchPutItem	否	是
DynamoDB BatchDeleteItem	否	是
HTTP	否	是
Amazon RDS	否	是

注：函数中目前仅支持 2018-05-29 版本。

更改单位解析器映射模板上的版本

对于单位解析器，将版本指定为请求映射模板正文的一部分。要更新版本，只需将 version 字段更新为新版本。

例如，要更新 AWS Lambda 模板中的版本：

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

您需要将版本字段从 2017-02-28 更新为 2018-05-29，如下所示：

```
{
    "version": "2018-05-29", ## Note the version
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

更改函数上的版本

对于函数，将版本指定为函数对象上的 functionVersion 字段。要更新版本，只需更新 functionVersion。注意：目前，函数仅支持 2018-05-29。

以下是用于更新现有函数版本的 CLI 命令的示例：

```
aws appsync update-function \
--api-id REPLACE_WITH_API_ID \
--function-id REPLACE_WITH_FUNCTION_ID \
--data-source-name "PostTable" \
--function-version "2018-05-29" \
--request-mapping-template "{...}" \
--response-mapping-template "\$util.toJson(\$ctx.result)"
```

注意：建议从函数请求映射模板中省略版本字段，因为它不会被接受。如果您在函数请求映射模板中指定了一个版本，则该版本值将由 functionVersion 字段的值覆盖。

2018-05-29

行为更改

- 如果数据源调用结果为 null，则执行响应映射模板。
- 如果数据源调用产生了错误，现在由您来处理该错误，响应映射模板评估结果将始终放置在 GraphQL 响应 data 块中。

Reasoning

- null 调用结果具有意义，在某些应用程序使用案例中，我们可能希望以自定义方式处理 null 结果。例如，应用程序可能会检查 Amazon DynamoDB 表中是否存在记录以执行某些授权检查。在这种情况下，null 调用结果意味着用户可能未获得授权。现在，执行响应映射模板可以引发未经授权的错误。此行为向 API 设计器提供了更大的控制。

给定以下响应映射模板：

```
$util.toJson($ctx.result)
```

之前使用 2017-02-28，如果 $ctx.result 返回 null，则未执行响应映射模板。利用 2018-05-29，我们现在可以处理这种情况。例如，我们可以选择引发授权错误，如下所示：

```
# throw an unauthorized error if the result is null
#if ( $util.isNull($ctx.result) )
    $util.unauthorized()
#end
$util.toJson($ctx.result)
```

注：从数据源返回的错误有时并不是致命的，或者甚至是预期的，这就是应向响应映射模板提供处理调用错误并决定是否忽略它、重新引发它或引发不同的错误的灵活性。

给定以下响应映射模板：

```
$util.toJson($ctx.result)
```

以前，使用 2017-02-28，如果出现调用错误，则会评估响应映射模板，并自动将结果放入 GraphQL 响应的 errors 块中。利用 2018-05-29，我们现在可以选择如何处理错误，重新引发错误、引发不同的错误，或者在返回数据时追加错误。

重新引发调用错误

在以下响应模板中，我们引发了从数据源返回的同一错误。

```
#if ( $ctx.error )
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

如果出现调用错误（例如，出现 $ctx.error），则响应如下所示：

```
{
    "data": {
        "getPost": null
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "DynamoDB:ConditionalCheckFailedException",
            "message": "Conditional check failed exception...",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

引发不同的错误

在以下响应模板中，我们在处理从数据源返回的错误之后引发了自己的自定义错误。

```
#if ( $ctx.error )
    #if ( $ctx.error.type.equals("ConditionalCheckFailedException") )
```

```
        ## we choose here to change the type and message of the error for
 ConditionalCheckFailedExceptions
        $util.error("Error while updating the post, try again. Error:
 $ctx.error.message", "UpdateError")
    #else
        $util.error($ctx.error.message, $ctx.error.type)
    #end
#end
$util.toJson($ctx.result)
```

如果出现调用错误（例如，出现 $ctx.error），则响应如下所示：

```
{
    "data": {
        "getPost": null
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "UpdateError",
            "message": "Error while updating the post, try again. Error: Conditional
 check failed exception...",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

追加错误以返回数据

在以下响应模板中，我们追加从数据源返回的同一错误，同时将数据返回到响应中。这也称为部分响应。

```
#if ( $ctx.error )
    $util.appendError($ctx.error.message, $ctx.error.type)
    #set($defaultPost = {id: "1", title: 'default post'})
```

```
    $util.toJson($defaultPost)
#else
    $util.toJson($ctx.result)
#end
```

如果出现调用错误（例如，出现 $ctx.error），则响应如下所示：

```
{
    "data": {
        "getPost": {
            "id": "1",
            "title: "A post"
        }
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "ConditionalCheckFailedException",
            "message": "Conditional check failed exception...",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

从 2017-02-28 迁移到 2018-05-29

从 2017-02-28 迁移到 2018-05-29 非常简单。更改解析器请求映射模板或函数版本对象上的版本字段。但请注意，2018-05-29 执行的行为与 2017-02-28 的不同，此处概述了更改。

从 2017-02-28 到 2018-05-29 保留相同的执行行为

在某些情况下，在执行 2018-05-29 版本的模板时，可以保留与 2017-02-28 版本相同的执行行为。

示例：DynamoDB PutItem

给定以下 2017-02-28 DynamoDB PutItem 请求模板：

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "attributeValues" : {
        "baz" : ... typed value
    },
    "condition" : {
       ...
    }
}
```

以及以下响应模板：

```
$util.toJson($ctx.result)
```

迁移到 2018-05-29 会更改这些模板，如下所示：

```
{
    "version" : "2018-05-29", ## Note the new 2018-05-29 version
    "operation" : "PutItem",
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "attributeValues" : {
        "baz" : ... typed value
    },
    "condition" : {
       ...
    }
}
```

更改响应模板，如下所示：

```
## If there is a datasource invocation error, we choose to raise the same error
## the field data will be set to null.
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
```

```
#end

## If the data source invocation is null, we return null.
#if($util.isNull($ctx.result))
  #return
#end

$util.toJson($ctx.result)
```

现在，您有责任处理错误，我们选择使用从 DynamoDB 返回的 `$util.error()` 来引发同一错误。您可以调整此代码段以将映射模板转换为 2018-05-29，请注意，如果您的响应模板不同，则必须考虑执行行为更改。

示例：DynamoDB GetItem

给定以下 2017-02-28 DynamoDB GetItem 请求模板：

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true
}
```

以及以下响应模板：

```
## map table attribute postId to field Post.id
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))

$util.toJson($ctx.result)
```

迁移到 2018-05-29 会更改这些模板，如下所示：

```
{
    "version" : "2018-05-29", ## Note the new 2018-05-29 version
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
```

```
        "bar" : ... typed value
    },
    "consistentRead" : true
}
```

更改响应模板，如下所示：

```
## If there is a datasource invocation error, we choose to raise the same error
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type)
#end

## If the data source invocation is null, we return null.
#if($util.isNull($ctx.result))
  #return
#end

## map table attribute postId to field Post.id
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))

$util.toJson($ctx.result)
```

在 2017-02-28 版本中，如果数据源调用是 `null`，则表示 DynamoDB 表中没有与我们的键匹配的项目，响应映射模板将不会执行。在大多数情况下，这都是可行的，但如果您期望 `$ctx.result` 不是 `null`，则现在必须处理该情况。

2017-02-28

特性

- 如果数据源调用结果为 `null`，则不会执行响应映射模板。
- 如果数据源调用产生了错误，则将执行响应映射模板，并将评估结果放入 GraphQL 响应 `errors.data` 块中。

类型参考

本节用作架构类型参考。

AWS AppSync 中的标量类型

GraphQL 对象类型具有名称和字段，并且这些字段可以具有子字段。最终，对象类型的字段必须解析为标量类型，这些类型表示查询的叶节点。有关对象类型和标量的更多信息，请参阅 GraphQL 网站上的 Schemas and types。

除了默认 GraphQL 标量集以外，AWS AppSync 还允许您使用以 AWS 前缀开头的服务定义的标量。AWS AppSync 不支持创建用户定义的（自定义）标量。您必须使用默认标量或 AWS 标量。

您不能将 AWS 作为自定义对象类型的前缀。

下一节是架构类型参考。

默认标量

GraphQL 定义了以下默认标量：

默认标量列表

`ID`

对象的唯一标识符。该标量像 `String` 一样序列化，但并不意味着用户可读。

`String`

UTF-8 字符序列。

`Int`

$-(2^{31})$ 和 $2^{31}-1$ 之间的整数值。

`Float`

IEEE 754 浮点值。

`Boolean`

一个布尔值，可以是 `true` 或 `false`。

AWS AppSync 标量

AWS AppSync 定义了以下标量：

AWS AppSync 标量列表

`AWSDate`

格式为 `YYYY-MM-DD` 的扩展 ISO 8601 日期字符串。

`AWSTime`

格式为 `hh:mm:ss.sss` 的扩展 ISO 8601 时间字符串。

`AWSDateTime`

格式为 `YYYY-MM-DDThh:mm:ss.sssZ` 的扩展 ISO 8601 日期和时间字符串。

> **Note**
>
> `AWSDate`、`AWSTime` 和 `AWSDateTime` 标量可以选择包含时区偏移。例如，值 `1970-01-01Z`、`1970-01-01-07:00` 和 `1970-01-01+05:30` 对于 `AWSDate` 均有效。时区偏移必须是 `Z` (UTC) 或以小时和分钟以及秒（可选）为单位的偏移。例如，`±hh:mm:ss`。时区偏移中的秒字段被认为有效，即使它不是 ISO 8601 标准的一部分。

`AWSTimestamp`

表示 `1970-01-01-T00:00Z` 之前或之后的秒数的整数值。

`AWSEmail`

采用 RFC 822 定义的格式 `local-part@domain-part` 的电子邮件地址。

`AWSJSON`

JSON 字符串。任何有效的 JSON 构造自动作为映射、列表或标量值解析并加载到解析器代码中，而不是作为文本输入字符串。不带引号的字符串或其他无效的 JSON 将导致 GraphQL 验证错误。

`AWSPhone`

电话号码。该值存储为字符串。电话号码可以包含空格或连字符以分隔数字组。没有国家/地区代码的电话号码假定为符合北美编号计划 (NANP) 的美国/北美号码。

AWSURL

RFC 1738 定义的 URL。例如，https://www.amazon.com/dp/B000NZW3KC/ 或 mailto:example@example.com。URL 必须包含模式（http、mailto），并且不能在路径部分中包含两个正斜杠 (//)。

AWSIPAddress

有效的 IPv4 或 IPv6 地址。IPv4 地址应采用四点表示法 (123.12.34.56)。IPv6 地址应采用以冒号分隔的无括号格式 (1a2b:3c4b::1234:4567)。您可以包含可选的 CIDR 后缀 (123.45.67.89/16) 以指示子网掩码。

架构用法示例

以下示例 GraphQL 架构将所有自定义标量作为“对象”，并显示基本 put、get 和 list 操作的解析器请求和响应模板。最后，该示例说明了在运行查询和变更时如何使用该架构。

```
type Mutation {
    putObject(
        email: AWSEmail,
        json: AWSJSON,
        date: AWSDate,
        time: AWSTime,
        datetime: AWSDateTime,
        timestamp: AWSTimestamp,
        url: AWSURL,
        phoneno: AWSPhone,
        ip: AWSIPAddress
    ): Object
}

type Object {
    id: ID!
    email: AWSEmail
    json: AWSJSON
    date: AWSDate
    time: AWSTime
    datetime: AWSDateTime
    timestamp: AWSTimestamp
    url: AWSURL
    phoneno: AWSPhone
    ip: AWSIPAddress
```

```
}

type Query {
    getObject(id: ID!): Object
    listObjects: [Object]
}

schema {
    query: Query
    mutation: Mutation
}
```

putObject 的请求模板可能如下所示。putObject 使用 PutItem 操作在 Amazon DynamoDB 表中创建或更新项目。请注意，该代码片段没有配置 Amazon DynamoDB 表以作为数据源。这仅用作一个示例：

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key" : {
        "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

putObject 的响应模板返回以下结果：

```
$util.toJson($ctx.result)
```

getObject 的请求模板可能如下所示。getObject 使用 GetItem 操作为给定主键的项目返回一组属性。请注意，该代码片段没有配置 Amazon DynamoDB 表以作为数据源。这仅用作一个示例：

```
{
    "version": "2017-02-28",
    "operation": "GetItem",
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
    }
}
```

getObject 的响应模板返回以下结果：

```
$util.toJson($ctx.result)
```

listObjects 的请求模板可能如下所示。listObjects 使用 Scan 操作返回一个或多个项目和属性。请注意，该代码片段没有配置 Amazon DynamoDB 表以作为数据源。这仅用作一个示例：

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
}
```

listObjects 的响应模板返回以下结果：

```
$util.toJson($ctx.result.items)
```

以下是将该架构与 GraphQL 查询一起使用的一些示例：

```
mutation CreateObject {
    putObject(email: "example@example.com"
        json: "{\"a\":1, \"b\":3, \"string\": 234}"
        date: "1970-01-01Z"
        time: "12:00:34."
        datetime: "1930-01-01T16:00:00-07:00"
        timestamp: -123123
        url:"https://amazon.com"
        phoneno: "+1 555 764 4377"
        ip: "127.0.0.1/8"
    ) {
        id
        email
        json
        date
        time
        datetime
        url
        timestamp
        phoneno
        ip
    }
}

query getObject {
    getObject(id:"0d97daf0-48e6-4ffc-8d48-0537e8a843d2"){
```

```
        email
        url
        timestamp
        phoneno
        ip
    }
}

query listObjects {
    listObjects {
        json
        date
        time
        datetime
    }
}
```

GraphQL 中的接口和联合

GraphQL 类型系统支持接口。接口会公开特定的字段组合，类型要实施接口，必须包含这些字段。

GraphQL 类型系统还支持联合。联合与接口相同，只是联合未定义一组通用字段。如果可能的类型没有共享的逻辑层次结构，联合通常比接口更加常用。

下一节是架构类型参考。

接口示例

我们可以表示一个 Event 接口，它表示任何种类的活动或人群聚集。一些可能的事件类型是 Concert、Conference 和 Festival。这些类型具有共同特征，包括名称、举行活动的地点以及开始和结束日期。这些类型也存在差异；Conference 提供演讲者和研讨会列表，而 Concert 包含表演乐队。

在架构定义语言 (SDL) 中，Event 接口定义如下所示：

```
interface Event {
        id: ID!
        name : String!
        startsAt: String
        endsAt: String
        venue: Venue
        minAgeRestriction: Int
```

```
}
```

每个类型都会实施 Event 接口，如下所示：

```
type Concert implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performingBand: String
}

type Festival implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performers: [String]
}

type Conference implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    speakers: [String]
    workshops: [String]
}
```

如果要表示多种类型的元素，接口就很有用。例如，我们可以搜索特定地点发生的所有事件。让我们在架构中添加 findEventsByVenue 字段，如下所示：

```
schema {
    query: Query
}

type Query {
```

```
    # Retrieve Events at a specific Venue
    findEventsAtVenue(venueId: ID!): [Event]
}

type Venue {
    id: ID!
    name: String
    address: String
    maxOccupancy: Int
}

type Concert implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performingBand: String
}

interface Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
}

type Festival implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performers: [String]
}

type Conference implements Event {
    id: ID!
    name: String!
    startsAt: String
```

```
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    speakers: [String]
    workshops: [String]
}
```

findEventsByVenue 返回 Event 列表。由于 GraphQL 接口字段对于所有实施类型都是通用的，所以可以选择 Event 接口的任何字段（id、name、startsAt、endsAt、venue 和 minAgeRestriction）。此外，只要您指定了类型，就可以使用 GraphQL 片段访问任何实施类型的字段。

让我们看一下使用该接口的 GraphQL 查询示例。

```
query {
  findEventsAtVenue(venueId: "Madison Square Garden") {
    id
    name
    minAgeRestriction
    startsAt

    ... on Festival {
      performers
    }

    ... on Concert {
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

上一查询生成一个结果列表，默认情况下，服务器会根据开始日期将事件排序。

```
{
  "data": {
    "findEventsAtVenue": [
      {
```

```
        "id": "Festival-2",
        "name": "Festival 2",
        "minAgeRestriction": 21,
        "startsAt": "2018-10-05T14:48:00.000Z",
        "performers": [
          "The Singers",
          "The Screamers"
        ]
      },
      {
        "id": "Concert-3",
        "name": "Concert 3",
        "minAgeRestriction": 18,
        "startsAt": "2018-10-07T14:48:00.000Z",
        "performingBand": "The Jumpers"
      },
      {
        "id": "Conference-4",
        "name": "Conference 4",
        "minAgeRestriction": null,
        "startsAt": "2018-10-09T14:48:00.000Z",
        "speakers": [
          "The Storytellers"
        ],
        "workshops": [
          "Writing",
          "Reading"
        ]
      }
    ]
  }
}
```

由于结果是作为单个事件集合返回的，因此，使用接口表示相同特性对结果排序非常有帮助。

联合示例

正如前面所述，联合不定义相同的字段集。搜索结果可能表示很多不同的类型。使用 Event 架构，您可以定义一个 SearchResult 联合，如下所示：

```
type Query {
    # Retrieve Events at a specific Venue
    findEventsAtVenue(venueId: ID!): [Event]
```

```
    # Search across all content
    search(query: String!): [SearchResult]
}

union SearchResult = Conference | Festival | Concert | Venue
```

此处，要查询 SearchResult 联合上的任何字段，您必须使用片段：

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
      id
      name
      performers
    }

    ... on Concert {
      id
      name
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

在 AWS AppSync 中解析类型

类型解析是一种机制，GraphQL 引擎通过这一机制将解析后的值识别为一种特定的对象类型。

回到联合搜索示例，假设我们的查询生成结果，结果列表中的每个项目必须将自身表示为 SearchResult 联合定义的可能类型之一（即，Conference、Festival、Concert 或 Venue）。

由于根据 Venue 或 Conference 识别 Festival 的逻辑取决于应用程序要求，必须给 GraphQL 引擎一点提示，这样它才能从原始结果中识别可能的类型。

在 AWS AppSync 中，该提示由名为 __typename 的元字段表示，其值对应于指定的对象类型名称。如果返回类型是接口或联合，则需要使用 __typename。

类型解析示例

让我们重复使用之前的架构。如果您要跟随操作，可以导航到控制台，并在 Schema (架构) 页面之下添加以下内容：

```
schema {
    query: Query
}

type Query {
    # Retrieve Events at a specific Venue
    findEventsAtVenue(venueId: ID!): [Event]
    # Search across all content
    search(query: String!): [SearchResult]
}

union SearchResult = Conference | Festival | Concert | Venue

type Venue {
    id: ID!
    name: String!
    address: String
    maxOccupancy: Int
}

interface Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
}

type Festival implements Event {
    id: ID!
    name: String!
```

```
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performers: [String]
}

type Conference implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    speakers: [String]
    workshops: [String]
}

type Concert implements Event {
    id: ID!
    name: String!
    startsAt: String
    endsAt: String
    venue: Venue
    minAgeRestriction: Int
    performingBand: String
}
```

让我们为 Query.search 字段附加解析器。在Resolvers部分中，选择附加，创建一个 NONE 类型的新数据源，然后将其命名为 StubDataSource。在此示例中，我们假设已从外部源提取了结果，并将提取的结果硬编码到请求映射模板中。

在请求映射模板窗格中，输入以下内容：

```
{
    "version" : "2018-05-29",
    "payload":
    ## We are effectively mocking our search results for this example
    [
        {
            "id": "Venue-1",
            "name": "Venue 1",
            "address": "2121 7th Ave, Seattle, WA 98121",
```

```
            "maxOccupancy": 1000
        },
        {
            "id": "Festival-2",
            "name": "Festival 2",
            "performers": ["The Singers", "The Screamers"]
        },
        {
            "id": "Concert-3",
            "name": "Concert 3",
            "performingBand": "The Jumpers"
        },
        {
            "id": "Conference-4",
            "name": "Conference 4",
            "speakers": ["The Storytellers"],
            "workshops": ["Writing", "Reading"]
        }
    ]
}
```

如果应用程序返回类型名称以作为 `id` 字段的一部分，则类型解析逻辑必须解析 `id` 字段以提取类型名称，然后将 `__typename` 字段添加到每个结果中。您可以在响应映射模板中执行该逻辑，如下所示：

> **Note**
>
> 如果使用 Lambda 数据源，您也可以将该任务作为 Lambda 函数的一部分执行。

```
#foreach ($result in $context.result)
    ## Extract type name from the id field.
    #set( $typeName = $result.id.split("-")[0] )
    #set( $ignore = $result.put("__typename", $typeName))
#end
$util.toJson($context.result)
```

运行以下查询：

```
query {
  search(query: "Madison") {
    ... on Venue {
```

```
      id
      name
      address
    }

    ... on Festival {
        id
      name
      performers
    }

    ... on Concert {
      id
      name
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

查询生成以下结果：

```
{
  "data": {
    "search": [
      {
        "id": "Venue-1",
        "name": "Venue 1",
        "address": "2121 7th Ave, Seattle, WA 98121"
      },
      {
        "id": "Festival-2",
        "name": "Festival 2",
        "performers": [
          "The Singers",
          "The Screamers"
        ]
      },
      {
```

```
        "id": "Concert-3",
        "name": "Concert 3",
        "performingBand": "The Jumpers"
      },
      {
        "speakers": [
          "The Storytellers"
        ],
        "workshops": [
          "Writing",
          "Reading"
        ]
      }
    ]
  }
}
```

类型解析逻辑会随应用程序而改变。例如，您可以使用另一种识别逻辑，检查某些字段或字段的组合是否存在。也就是说，您可以检测是否存在 performers 字段，以识别 Festival；或利用 speakers 和 workshops 字段的组合来识别 Conference。最终，由您定义要使用的逻辑。

问题排查和常见错误

此部分将讨论一些常见的错误以及如何排查这些错误。

DynamoDB 键映射不正确

如果您的 GraphQL 操作返回以下错误消息，可能是因为您的请求映射模板结构与 Amazon DynamoDB 键结构不匹配：

```
The provided key element does not match the schema (Service: AmazonDynamoDBv2; Status
 Code: 400; Error Code
```

例如，如果您的 DynamoDB 表具有一个名为 "id" 的哈希键，并且您的模板包含 "PostID"（如以下示例中所示），则会导致上述错误，因为 "id" 与 "PostID" 不匹配。

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "PostID" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

缺少解析器

如果您执行 GraphQL 操作（如查询）并获得 Null 响应，这可能是因为您没有配置解析器。

例如，如果您导入的架构定义了 getCustomer(userId: ID!): 字段，但您尚未为此字段配置解析器，则当执行查询时（例如 getCustomer(userId:"ID123"){...}），您将获得如下所示的响应：

```
{
    "data": {
    "getCustomer": null
    }
}
```

映射模板错误

如果您的映射模板未正确配置，您将收到其 `errorType` 为 `MappingTemplate` 的 GraphQL 响应。`message` 字段应指出问题出在映射模板中的何处。

例如，如果在您的请求映射模板中没有 `operation` 字段，或者 `operation` 字段名称不正确，您将收到类似以下内容的响应：

```
{
    "data": {
        "searchPosts": null
    },
    "errors": [
        {
        "path": [
            "searchPosts"
        ],
        "errorType": "MappingTemplate",
        "locations": [
            {
            "line": 2,
            "column": 3
            }
        ],
        "message": "Value for field '$[operation]' not found."
        }
    ]
}
```

返回类型不正确

数据来源的返回类型必须与架构中对象的已定义类型匹配，否则您可能会看到如下所示的 GraphQL 错误：

```
"errors": [
    {
    "path": [
        "posts"
    ],
    "locations": null,
```

```
    "message": "Can't resolve value (/posts) : type mismatch error, expected type LIST,
 got OBJECT"
    }
]
```

例如，对于以下查询定义，可能会发生上述错误：

```
type Query {
    posts: [Post]
}
```

这预计是 [Posts] 对象的 LIST。例如，如果 Node.JS 中有一个 Lambda 函数（与以下内容类似）：

```
const result = { data: data.Items.map(item => { return item ; }) };
callback(err, result);
```

这将引发错误，因为 result 是一个对象。您需要将回调更改为 result.data，或更改架构以不返回 LIST。

正在处理无效的请求

AWS AppSync当无法处理请求并将其发送到字段解析器时（由于数据不正确，例如语法无效），响应负载将返回值设置为的字段数据null以及任何相关错误。

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。