開發人員指南

AWS AppSync

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

AWS AppSync: 開發人員指南

Copyright © 2024 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

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

Table of Contents

什麼是 AWS AppSync？

AWSAppSync使開發人員能夠使用安全、無伺服器和高效能 GraphQL 和 Pub/Sub API，將其應用程式和服務與資料和事件連接起來。您可以使用以下操作 AWSAppSync：

- 從單一 GraphQL API 端點存取一或多個資料來源的資料。
- 將多個來源 GraphQL API 合併為單一、合併的 GraphQL API。
- 將即時資料更新發佈至您的應用程式。
- 運用內建的安全性、監控、記錄和追蹤功能，搭配選用的快取功能，達到低延遲。
- 只需為 API 請求和傳遞的任何即時訊息付費。

主題

AWSAppSync功能

- 由 GraphQL 提供支援的簡化資料存取和查詢
- WebSockets適用於 GraphQL 訂閱和發布/訂閱通道的無伺服器
- 伺服器端快取可讓資料在高速記憶體內快取中使用，以達到低延遲
- JavaScript並TypeScript支持編寫業務邏輯
- 利用私有 API 來限制 API 存取和與之整合的企業安全性 AWS WAF
- 內建授權控制功能，並支援 API 金鑰、IAM、Amazon Cognito、OpenID Connect 供應商，以及自訂邏輯的 Lambda 授權。
- 合併 API 以支援聯合使用案例

如需這些功能的詳細資訊，請參閱AWSAppSync功能。

您是第一次使用 AWS AppSync 的新手嗎？

建議您一開AWS AppSync始先閱讀下列各節：

- 如果您不熟悉 GraphQL，請參閱. 開始使用：建立您的第一個圖形 SQL API
- 如果您正在建置使用 GraphQL API 的應用程式，請參閱建立用戶端應用程式和the section called “即時資料”。
- 如果您正在尋找 GraphQL 解析器資訊：

  JavaScript/TypeScript

  - 解析器教程（ ）JavaScript
  - 解析器參考 () JavaScript

  VTL

  - 解析器教程（VTL）
  - 解析器對映範本參考 (VTL)
- 如果您正在尋找AWS AppSync範例專案、更新等項目，請參閱部AppSync落格。

相關服務

如果您要從頭開始構建 Web 或移動應用程序，請考慮使用 AWS Amplify. Amplify 槓桿AWS AppSync和其他AWS服務可協助您以更少的工作量建置更強大、功能更強大的 Web 和行動應用程式。

AWS AppSync 的定價

AWS AppSync根據數百萬個請求和更新定價。緩存需要額外費用。如需詳細資訊，請參閱 AWS AppSync 定價。

下列各AWS AppSync節：

- 中的 API 快取AWS AppSync不符合AWS免費方案資格。
- 授權和驗證失敗的要求不會收取費用。
- API 金鑰遺失或無效時，不會收取呼叫需要 API 金鑰之方法的費用。

GraphQL 與架構 AWS AppSync

> **Note**
>
> 本指南假設用戶具有 REST 架構風格的工作知識。我們建議您在使用 GraphQL 和. 之前，先檢閱此主題和AWS AppSync其他前端主題。

GraphQL 是用於 API 的查詢和操作語言。GraphQL 提供了一種靈活且直觀的語法來描述數據需求和交互。它使開發人員能夠確切地詢問所需的內容，並取回可預測的結果。它還可以在單個請求中訪問許多來源，從而減少了網絡呼叫的數量和帶寬需求，從而節省了應用程序消耗的電池壽命和 CPU 週期。

通過突變可以簡化數據的更新，從而使開發人員可以描述數據應該如何更改。GraphQL 也有助於透過訂閱快速設定即時解決方案。所有這些功能結合，再加上強大的開發人員工具，使 GraphQL 對於管理應用程式資料至關重要。

GraphQL 是其餘的替代方案。RESTful 架構目前是用於客戶端-服務器通信的比較流行的解決方案之一。它以 URL 公開的資源（數據）的概念為中心。這些 URL 可以用來訪問和通過 CRUD（創建，讀取，更新，刪除）在 HTTP 方法的形式`GET`，如`POST`，和`DELETE`操作數據。REST 的優點是學習和實施相對簡單。您可以快速設定 RESTful API 來呼叫廣泛的服務。

但是，技術變得越來越複雜。隨著應用程式、工具和服務開始為全球使用者擴充，對快速、可擴充架構的需求非常重要。REST 在處理可擴展操作時存在許多缺點。請參閱此使用案例以取得範例。

在下面的章節中，我們將回顧一些圍繞 RESTful API 的概念。然後，我們將介紹 GraphQL 及其工作原理。

如需 GraphQL 以及移轉至的好處的詳細資訊AWS，請參閱 GraphQL 實作的決策指南。

主題

- 什麼是 API？
- 什麼是休息？
- 為什麼使用 GraphQL 而不是休息？
- GraphQL API 的元件
- GraphQL 的其他屬性

什麼是 API？

應用程式設計介面 (API) 定義了與其他軟體系統通訊時必須遵循的規則。開發人員公開或建立 API，讓其他應用程式可以透過程式設計方式與應用程式 例如，時間表應用程序公開了一個 API，要求員工的全名和日期範圍。當它收到此信息時，它會在內部處理員工的時間表，並返回該日期範圍內的工作時數。

您可以將 Web API 視為 Web 上客戶端和資源之間的網關。

用戶端

用戶端是想要從網路存取資訊的使用者。用戶端可以是使用 API 的個人或軟體系統。例如，開發人員可以撰寫從氣象系統存取天氣資料的程式。或者，當您直接訪問天氣網站時，您可以從瀏覽器訪問相同的數據。

資源

資源是不同的應用程序提供給他們的客戶的信息。資源可以是圖像，視頻，文本，數字或任何類型的數據。將資源提供給客戶端的機器也稱為服務器。Organizations 使用 API 來共用資源並提供 Web 服務，同時維護安全性、控制和驗證。此外，API 可協助他們判斷哪些用戶端可以存取特定的內部資源。

什麼是休息？

在高層次上，具象性狀態傳輸（REST）是一種軟件體系結構，對 API 應該如何工作施加條件。REST 最初是作為管理互聯網等複雜網絡上通信的指導原則而創建的。您可以使用 REST 架構來大規模支援高效能且可靠的通訊。您可以輕鬆實施和修改它，為任何 API 系統帶來可見性和跨平台可移植性。

API 開發人員可以使用多種不同的架構來設計 API。遵循其餘架構樣式的 API 稱為其餘 API。實作 REST 架構的 Web 服務稱為 REST 風格的網路服務。術語 REST 風格的 API 通常指的是 REST 風格的網絡 API。但是，您可以互換使用術語 REST API 和 REST 風格 API。

以下是 REST 建築風格的一些原則：

統一介面

統一的接口是任何 RESTful Web 服務設計的基礎。這表明服務器以標準格式傳輸信息。格式化的資源在 REST 中稱為表示。此格式可以不同於伺服器應用程式上資源的內部表示法。例如，伺服器可以將資料儲存為文字，但以 HTML 表示格式傳送。

統一的接口施加了四個架構約束：

1. 請求應識別資源。他們通過使用統一的資源標識符來做到這一點。
2. 客戶端在資源表示中有足夠的信息來修改或刪除資源（如果需要）。伺服器會傳送進一步描述資源的中繼資料，藉此符合此條件。
3. 客戶會收到有關如何進一步處理表示的資訊。服務器通過發送包含有關客戶端如何最好地使用它們的元數據的自我描述消息來實現這一目標。
4. 用戶端會收到完成工作所需的所有其他相關資源的相關資訊。服務器通過發送表示超鏈接來實現這一點，以便客戶端可以動態地發現更多資源。

無狀態

在 REST 體系結構中，無狀態是指服務器完成每個客戶端請求的通信方法，而不受所有先前請求的影響。客戶端可以按任何順序請求資源，並且每個請求都是無狀態的或與其他請求隔離的。這種 REST API 設計約束意味著服務器每次都可以完全理解和滿足請求。

分層系統

在分層系統架構中，客戶端可以連接到客戶端和服務器之間的其他授權中介機構，並且它仍然會接收來自服務器的響應。服務器還可以將請求傳遞給其他服務器。您可以將 RESTful Web 服務設計為在具有多層（例如安全性，應用程序和業務邏輯）的多個服務器上運行，共同工作以滿足客戶端請求。這些層仍然是不可見的客戶端。

可快取性

RESTful Web 服務支持緩存，這是在客戶端或中介存儲一些響應以提高服務器響應時間的過程。例如，假設您造訪的網站在每個頁面上都有共同的頁首和頁尾影像。每次訪問新網站頁面時，服務器都必須重新發送相同的圖像。為了避免這種情況，用戶端會在第一個回應之後快取或儲存這些影像，然後直接使用快取中的影像。RESTful Web 服務通過使用將自己定義為可緩存或不可緩存的 API 響應來控制緩存。

什麼是一個不穩定的 API？

RESTful API 是兩個計算機系統用於通過互聯網安全地交換信息的接口。大多數商務應用程式必須與其他內部和第三方應用程式通訊，才能執行各種工作 例如，要生成每月工資單，您的內部帳戶系統必須與客戶的銀行系統共享數據，以自動開立發票並與內部時間表應用程序進行通信。RESTful API 支持此信息交換，因為它們遵循安全，可靠和高效的軟件通信標準。

如何做瑞斯特的 API 工作？

一個 RESTful API 的基本功能與瀏覽互聯網相同。客戶端通過使用 API 聯繫服務器時，它需要一個資源。API 開發人員在伺服器應用程式 API 文件中說明用戶端應該如何使用 REST API。以下是任何 REST API 呼叫的一般步驟：

1. 客戶端向服務器發送請求。客戶端遵循 API 文檔以服務器理解的方式格式化請求。
2. 伺服器會驗證用戶端，並確認用戶端有權提出該要求。
3. 服務器接收請求並在內部進行處理。
4. 服務器返回到客戶端的響應。響應包含信息，告訴客戶端請求是否成功。響應還包括客戶請求的任何信息。

REST API 請求和響應詳細信息會根據 API 開發人員設計 API 的方式略有不同。

為什麼使用 GraphQL 而不是休息？

REST 是 Web API 的基礎架構風格之一。然而，隨著世界變得更加相互聯繫，開發強大且可擴展的應用程序的需求將成為一個更緊迫的問題。雖然 REST 目前是構建 Web API 的行業標準，但 RESTful 實現已經確定了幾個反復出現的缺點：

1. 數據請求：使用 RESTful API，您通常會通過端點請求所需的數據。當你有可能不那麼整齊地打包的數據時，就會出現問題。您需要的數據可能位於多層抽象的後面，獲取數據的唯一方法是使用多個端點，這意味著發出多個請求來提取所有數據。
2. 重載和底線：為了增加多個請求的問題，來自每個端點的數據是嚴格定義的，這意味著您將返回為該 API 定義的任何數據，即使您在技術上不需要它。

   這可能會導致過度獲取，這意味著我們的請求返回多餘的數據。例如，假設您要求公司人員資料，並且想要知道特定部門中員工的姓名。返回數據的端點將包含名稱，但它也可能包含其他數據，如工作職稱或出生日期。由於 API 是固定的，因此您不能單獨請求名稱；其餘的數據都隨之而來。

   在我們沒有返回足夠的數據相反的情況被稱為不足抓取。要獲取所有請求的數據，您可能必須向服務提出多個請求。根據數據的結構方式，您可能會遇到效率低下的查詢，導致諸如可怕的 n+1 問題之類的問題。
3. 緩慢的開發迭代：許多開發人員量身定制其 RESTful API 以適應其應用程序的流程。但是，隨著應用程式的成長，前端和後端都可能需要進行大量變更。因此，API 可能不再以有效或有影響力的方式符合資料的形式。由於需要修改 API，這會導致產品迭代速度較慢。

4. 大規模效能：由於這些複合問題，有許多可擴展性會受到影響的領域。應用程式端的效能可能會受到影響，因為您的要求會傳回太多資料或太少 (導致更多要求)。這兩種情況都會對網路造成不必要的壓力，導致效能不佳。在開發人員方面，開發速度可能會降低，因為您的 API 是固定的，不再適合他們請求的數據。

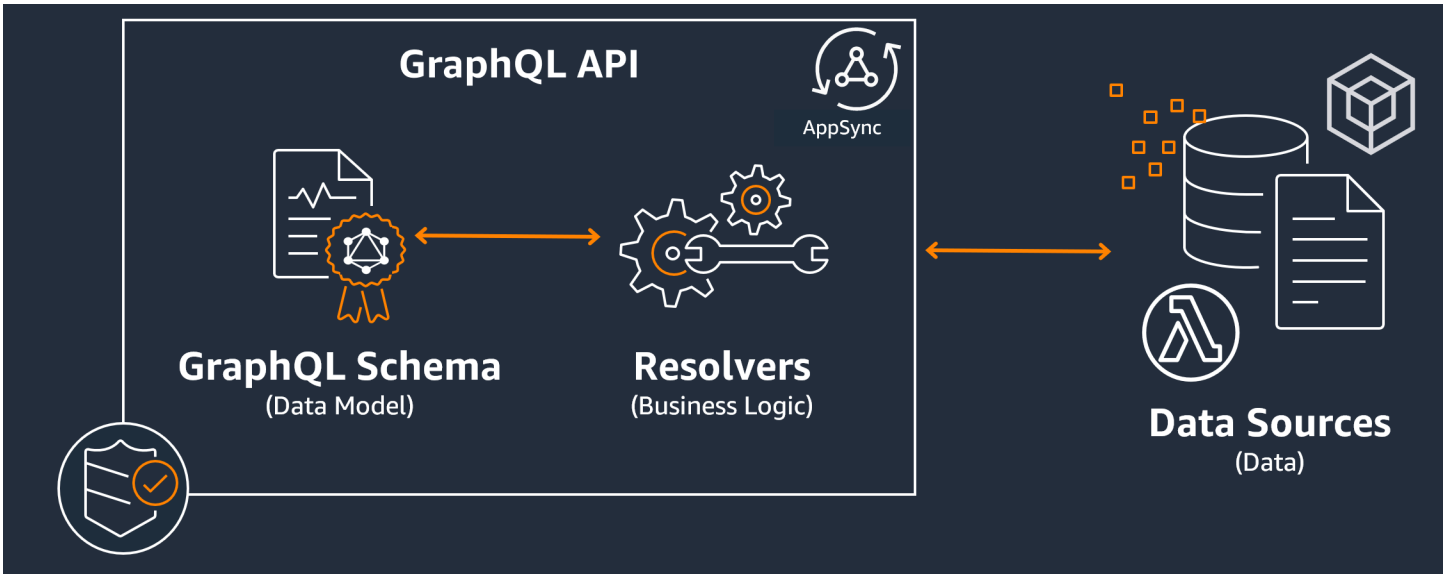
GraphQL 的賣點是克服 REST 的缺點。以下是 GraphQL 為開發人員提供的一些關鍵解決方案：

1. 單一端點：GraphQL 使用單一端點來查詢資料。不需要建置多個 API 來符合資料的形狀。這會導致通過網絡傳輸的請求更少。
2. 獲取：GraphQL 通過簡單地定義所需的數據來解決過度和不足抓取的常年問題。GraphQL 可讓您根據需求調整資料，因此您只會收到您要求的資料。
3. 抽象：GraphQL API 包含一些組件和系統，這些組件和系統使用與語言無關的標準描述數據。換句話說，數據的形狀和結構是標準化的，因此前端和後端都知道如何通過網絡發送數據。這使得兩端的開發人員可以使用 GraphQL 的系統，而不是圍繞它們。
4. 快速迭代：由於數據的標準化，另一端可能不需要在開發的一端進行更改。例如，前端簡報變更可能不會導致大量的後端變更，因為 GraphQL 允許資料規格隨時修改。您可以簡單地定義或修改數據的形狀，以適應應用程序的需求，因為它成長。這導致潛在的開發工作較少。

這些只是 GraphQL 的一些好處。在接下來的幾節中，您將了解 GraphQL 的構造方式以及使其成為 REST 獨特替代方案的屬性。

GraphQL API 的元件

標準 GraphQL API 由單一結構描述組成，該結構描述可處理將要查詢的資料的形狀。您的結構描述會連結至一或多個資料來源，例如資料庫或 Lambda 函數。在兩者之間，一個或多個解析器處理您的請求的業務邏輯。每個元件在 GraphQL 實作中都扮演著重要角色。以下各節將介紹這三個元件，以及它們在 GraphQL 服務中扮演的角色。



主題

綱要

圖 GraphQL 結構描述是 GraphQL API 的基礎。它可作為定義資料形式的藍圖。這也是您的客戶端和服務器之間的合同，它定義了如何檢索和/或修改數據。

GraphQL 結構描述是以結構描述定義語言 (SDL) 撰寫的。SDL 由具有既定結構的型別與欄位所組成：

- 類型：類型是 GraphQL 定義數據的形狀和行為的方式。GraphQL 支援多種類型，這些類型將在本節稍後加以說明。結構描述中定義的每個類型都會包含其自己的範圍。範圍內將是一個或多個可以包含將在 GraphQL 服務中使用的值或邏輯的欄位。類型填充許多不同的角色，最常見的是對象或標量（原始值類型）。
- 欄位：欄位存在於類型範圍內，並保留 GraphQL 服務要求的值。這些與其他編程語言中的變量非常相似。您在字段中定義的數據的形狀將決定數據在請求/響應操作中的構造方式。這使開發人員可以在不知道如何實現服務後端的情況下預測將返回的內容。

若要視覺化結構描述的外觀，讓我們檢閱簡單 GraphQL 結構描述的內容。在生產代碼中，您的模式通常位於名為`schema.graphql`或的文件中`schema.json`。假設我們正在對等於實作 GraphQL 服務的

專案。該項目正在存儲公司人事數據，該schema.graphql文件被用於檢索人事數據並將新人員添加到數據庫中。代碼可能如下所示：

schema.graphql

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

我們可以看到，在結構描述中定義了三種類型：PersonQuery、、和Mutation。看著Person，我們可以猜測，這是一個公司員工的實例的藍圖，這將使這種類型的對象。在它的範圍內，我們看到idname、、和age。這些是定義的屬性的字段Person。這意味著我們的數據源將每個Person數據源存儲name為String標量（原始）類型和ageInt標量（原始）類型。作id為每個標識符的特殊唯一標識符Person。這也是由!符號表示的必需值。

接下來的兩個物件類型的行為不同。GraphQL 會針對特殊物件類型保留幾個關鍵字，這些關鍵字會定義資料在結構描述中的填入方式。Query類型將從源中檢索數據。在我們的例子中，我們的查詢可能會從數據庫中檢索Person對象。這可能會提醒您 RESTful 術語中的GET操作。A Mutation 將修改數據。在我們的例子中，我們的突變可能會向數據庫添加更多Person對象。這可能會讓您想起或之類PUT的狀態變化操作。POST本節稍後將說明所有特殊物件類型的行為。

讓我們假設Query在我們的例子將從數據庫中檢索的東西。如果我們看一下的領域Query，我們看到一個名為字段people。其欄位值為[Person]。這意味著我們要檢索數據庫Person中的某些實例。但是，添加括號意味著我們要返回所有Person實例的列表，而不僅僅是一個特定的實例。

該Mutation類型負責執行狀態更改操作，如數據修改。突變負責對數據源執行某些狀態更改操作。在我們的例子中，我們的突變包含一個名為的操作，addPerson該操作將新Person對象添加到數據庫中。突變使用 a，Person並且需要idname、和age欄位的輸入。

在這一點上，您可能想知道這樣的操addPerson作如何在沒有代碼實現的情況下工作，因為它應該執行某些行為並且看起來很像具有函數名稱和參數的函數。目前，它將不起作用，因為模式僅用作聲明。為了實現的行為addPerson，我們將不得不向其添加一個解析器。解析器是每當調用其關聯字段（在

本例中為addPerson操作）時執行的代碼單元。如果你想使用一個操作，你必須在某個時候添加解析器實現。在某種程度上，您可以將模式操作視為函數聲明，並將解析器視為定義。解析器將在不同的部分中進行說明。

這個範例只會顯示結構描述可以操作資料的最簡單方法。您可以利用 GraphQL 和. 的功能，建置複雜、穩固且可擴充的應用程式。AWS AppSync在下一節中，我們將定義您可以在結構描述中使用的所有不同類型和欄位行為。

GraphQL 類型

GraphQL 支援許多不同的類型。正如您在上一節中所看到的，類型定義了數據的形狀或行為。它們是 GraphQL 結構描述的基本建置區塊。

類型可以分為輸入和輸出。輸入是允許作為特殊對象類型（等）的參數傳入的類型 `Query`、`Mutation`，而輸出類型則嚴格用於存儲和返回數據。下面列出了類型及其分類的列表：

- 物件：物件包含描述實體的欄位。例如，一個對象可能是像一個book描述其特徵的字段authorName，如publishingYear，等等。它們是嚴格的輸出類型。
- 標量：這些是原始類型，如 int，字符串等 它們通常會指派給欄位。使用該authorName字段作為示例，可以將String標量分配給存儲像「約翰·史密斯」這樣的名稱。標量可以是輸入和輸出類型。
- 輸入：輸入允許您將一組字段作為參數傳遞。它們的結構與對象非常相似，但它們可以作為參數傳遞給特殊對象。輸入允許您在其範圍內定義純量，枚舉和其他輸入。輸入只能是輸入類型。
- 特殊對象：特殊對象執行狀態變化操作，並完成服務的繁重工作。有三種特殊的物件類型：查詢、變異和訂閱。查詢通常會擷取資料；突變操作資料；訂閱會開啟並維護用戶端與伺服器之間的雙向連線，以便持續進行通訊。特殊對象既不是輸入也不輸出給定它們的功能。
- 枚舉：枚舉是預定義的合法值列表。如果您調用枚舉，則其值只能是其範圍中定義的值。例如，如果您有一個名為trafficLights描述流量信號列表的枚舉，則它可能具有類似redLightgreenLight而不purpleLight是這樣的值。一個真正的交通信號燈只會有這麼多的信號，所以你可以使用枚舉來定義它們，並強制它們成為引用時唯一的合法值trafficLight。枚舉可以是輸入和輸出類型。
- 聯集/介面：聯集可讓您根據用戶端要求的資料，在要求中傳回一或多個項目。例如，如果您有一個帶有title字段的類Book型和一個帶有name字段的類Author型，則可以在兩種類型之間創建聯集。如果您的客戶想查詢一個數據庫中的短語「凱撒大帝」，聯盟可以返回朱利葉斯·凱撒大帝 (由莎士比亞劇) 從Booktitle和朱利葉斯·凱撒 (評論家德貝洛加利科的作者) 從. Author name 聯集只能是輸出類型。

  接口是對象必須實現的字段集。這與 Java 等編程語言中的接口有點類似，您必須在其中實現接口中定義的字段。例如，假設您製作了一個名為包Book含title欄位的介面。假設您稍後創建了一個稱

為實現Novel的類型Book。你Novel將不得不包括一個title字段。但是，您也Novel可以包含不在界面中的其他字pageCount段ISBN。接口只能是輸出類型。

以下各節將說明每種類型在 GraphQL 中的工作原理。

物件

GraphQL 對象是您將在生產代碼中看到的主要類型。在 GraphQL 中，您可以將物件視為不同欄位的群組 (類似於其他語言中的變數)，每個欄位都是由可容納值的類型 (通常是純量或其他物件) 定義。對象表示可以從服務實現中檢索/操作的數據單元。

對象類型使用Type關鍵字聲明。讓我們稍微修改我們的模式示例：

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

type Occupation {
  title: String
}
```

這裡的物件類型是Person和Occupation。每個對象都有自己的字段和自己的類型。GraphQL 的一個特點是能夠將欄位設定為其他類型。您可以在中看到occupation欄位Person包含Occupation物件類型。我們可以進行此關聯，因為 GraphQL 僅描述數據，而不是服務的實現。

标量

標量基本上是保存值的原始類型。在中AWS AppSync，有兩種類型的純量：預設 GraphQL 純量和純量。AWS AppSync純量通常用於在對象類型中存儲字段值。預設 GraphQL 類型包括IntFloatString、Boolean、和ID。讓我們再次使用前面的例子：

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

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

挑選name和title字段，都保持一個String標量。 Name可以返回像 "John Smith" 這樣的字符串值，並且標題可以返回類似 "firefighter」的內容。某些 GraphQL 實作也支援使用Scalar關鍵字和實作型別行為的自訂純量。但是，AWS AppSync目前不支持自定義標量。如需純量清單，請參閱中的純量類型。AWS AppSync

輸入

由於輸入和輸出類型的概念，在傳入參數時存在某些限制。通常需要傳入的類型（尤其是對象）受到限制。您可以使用輸入類型來略過此規則。輸入是包含標量，枚舉和其他輸入類型的類型。

輸入使用input關鍵字定義：

```
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
}
```

正如你所看到的，我們可以有模仿原始類型的單獨輸入。這些輸入通常會在您的現場操作中使用，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
}
```

請注意我們如何仍然傳遞occupationInput的位置Occupation來創建一個Person.

這只是輸入的一種情況。他們不一定需要復制對象 1：1，在生產代碼中，你很可能不會像這樣使用它。最好利用 GraphQL 結構描述，方法是定義您需要輸入為引數的內容。

此外，相同的輸入可以在多個操作中使用，但我們不建議這樣做。理想情況下，每個操作都應該包含自己的唯一輸入副本，以防結構描述的需求發生變化。

特殊物件

GraphQL 為特殊對象保留了一些關鍵字，這些對象定義了模式如何檢索/操作數據的一些業務邏輯。在結構描述中，最多可以有這些關鍵字中的每個關鍵字之一。它們充當用戶端針對 GraphQL 服務執行的所有要求資料的入口點。

特殊對象也使用type關鍵字定義。儘管它們的使用方式與常規對像類型不同，但它們的實現非常相似。

Queries

查詢與GET作業非常相似，因為它們會執行唯讀擷取以從來源取得資料。在 GraphQL 中，定Query義了針對伺服器發出要求的用戶端的所有進入點。您的 GraphQL 實現Query中總會有一個。

以下是我們在前面的模式示例中使用的Query和修改的對象類型：

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

我們Query包含一個名為的字段people，該字段從數據源返回Person實例列表。比方說，我們需要改變我們的應用程序的行為，現在我們需要返回只有一些單獨的目的Occupation實例的列表。我們可以簡單地將其添加到查詢中：

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

在 GraphQL 中，我們可以將查詢視為請求的單一來源。正如你所看到的，這可能比 RESTful 實現簡單得多，可能使用不同的端點來實現相同的事情（.../api/1/people和.../api/1/occupations）。

假設我們有這個查詢的解析器實現，我們現在可以執行一個實際的查詢。雖然Query類型存在，我們必須明確地調用它，以便在應用程序的代碼中運行。這可以使用query關鍵字來完成：

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

正如你所看到的，這個查詢被調用getItems並返回people（Person對象列表）和occupations（Occupation對象列表）。在中people，我們只返回每個name領域Person，而我們正在返回每個title領域Occupation。回應可能如下所示：

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

範例回應顯示資料如何遵循查詢的形狀。擷取的每個項目都會列在欄位的範圍內。people並occupations將事物作為單獨的列表返回。儘管很有用，但修改查詢以返回人員姓名和職業列表可能會更方便：

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

這是一個法律修改，因為我們的Person類型包含一個類型的occupation字段Occupation。當在的範圍內列出時people，我們將返回每個name與Person它們相關聯Occupation的title。回應可能如下所示：

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

Mutations

突變類似於或之類的狀態變化操作。PUT POST他們執行寫操作來修改源中的數據，然後獲取響應。他們定義了數據修改請求的進入點。與查詢不同，突變可能會或可能不會包含在模式中，具體取決於項目的需求。以下是架構示例中的突變：

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

此addPerson欄位代表一個將資料來源新增Person至資料來源的進入點。 addPerson是欄位名稱；idname、和age是參數；且Person是傳回類型。回顧Person類型：

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

我們新增occupation欄位。但是，我們不能Occupation直接將此字段設置為，因為對象不能作為參數傳入; 它們是嚴格的輸出類型。我們應該傳遞一個與參數相同字段的輸入：

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

我們還可以輕鬆地更新我們的，以便在創建新Person實例時addPerson將其包含為參數：

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

以下是更新的結構描述：

```
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
}
```

請注意，occupation將從title欄位中傳入，occupationInput以完成建立Person而非原始Occupation物件。假設我們有一個解析器實現addPerson，我們現在可以執行一個實際的突變。雖然Mutation類型存在，我們必須明確地調用它，以便在應用程序的代碼中運行。這可以使用mutation關鍵字來完成：

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

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

這種突變被稱為createPerson，並且addPerson是操作。若要建立新的Person，我們可以輸入id、nameage、和的引數occupation。在範圍內addPerson，我們還可以看到其他領域name，如age，等等。這是您的回應；這些是addPerson作業完成後將傳回的欄位。以下是範例的最後一部分：

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

使用此突變，結果可能如下所示：

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

正如你所看到的，響應以我們的突變中定義的相同格式返回我們請求的值。最好是傳回修改過的所有值，以減少混淆和 future 需要更多查詢。突變允許您在其範圍內包含多個操作。它們將按照

突變中列出的順序順序運行。例如，如果我們創建另一個名addOccupation為將作業標題添加到數據源的操作，則可以在突變之後addPerson調用它。 addPerson將首先處理，然後進行處理addOccupation。

Subscriptions

訂閱用WebSockets於在伺服器及其用戶端之間開啟持久的雙向連線。一般而言，用戶端會訂閱或監聽伺服器。每當服務器進行服務器端更改或執行事件時，訂閱的客戶端將收到更新。當訂閱多個用戶端，而且需要收到伺服器或其他用戶端中發生變更的通知時，這種類型的通訊協定非常有用。例如，訂閱可用於更新社交媒體供稿。可能有兩個使用者，即使用者 A 和使用者 B，他們都訂閱了自動通知更新，每當他們收到直接訊息。用戶端 A 上的使用者 A 可以直接傳送訊息給用戶端 B 上的使用者 B。使用者 A 的用戶端會傳送直接訊息，伺服器會處理此訊息。接著，伺服器會傳送直接訊息給使用者 B 的帳戶，同時傳送自動通知給用戶端 B。

以下是我們可以添加到模式示例中的示例：Subscription

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

每當新增到資料來源時，該personAdded欄位Person就會向訂閱的用戶端傳送訊息。假設我們有一個解析器實現personAdded，我們現在可以使用訂閱。雖然Subscription類型存在，我們必須明確地調用它，以便在應用程序的代碼中運行。這可以使用subscription關鍵字來完成：

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

訂閱被調用personAddedOperation，操作是personAdded。 personAdded將返回新Person實例的id和name字段。看看突變示例，我們添加了一個 Person using 這個操作：

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

如果我們的客戶訂閱了新添加的更新Person，他們可能會在addPerson運行後看到以下內容：

```
{
```

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

以下是訂閱提供的摘要：

訂閱是雙向通道，可讓用戶端和伺服器接收快速但穩定的更新。他們通常使用 WebSocket 協議，該協議創建標準化和安全的連接。

訂閱非常靈活，因為它們可以減少連接設置開銷。一旦訂閱，用戶端就可以長時間在該訂閱上繼續執行。他們通常可以讓開發人員自訂訂閱的生命週期，並設定要求哪些資訊，藉此有效地使用運算資源。

一般而言，訂閱允許用戶端一次進行多個訂閱。就其相關而言AWS AppSync，訂閱僅用於從AWS AppSync服務接收即時更新。它們不能用於執行查詢或突變。

訂閱的主要替代方案是輪詢，它會以設定的間隔傳送查詢以要求資料。此過程通常比訂閱效率低，並且對客戶端和後端產生了很大的壓力。

在我們的模式示例中沒有提到的一件事是，您的特殊對像類型也必須在schema根中定義。因此，當您在中導出模式時AWS AppSync，它可能看起來像這樣：

schema.graphql

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

.
.
.

type Query {
  # code goes here
}
```

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

列舉

枚舉或枚舉是限制類型或字段可能具有的法律參數的特殊純量。這意味著每當在模式中定義枚舉時，其關聯的類型或字段將被限制為枚舉中的值。枚舉被序列化為字符串標量。請注意，不同的編程語言可能以不同的方式處理 GraphQL 枚舉。例如，沒 JavaScript 有本地枚舉支持，因此枚舉值可能會映射到 int 值。

枚舉是使用enum關鍵字定義的。範例如下：

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

調用trafficLights枚舉時，參數只能是 solidRed solidYellowsolidGreen，等等。通常使用枚舉來描述具有不同但有限數量選擇的東西。

聯會/介面

請參閱 GraphQL 中的介面和聯集。

GraphQL 欄位

欄位存在於類型的範圍內，並保留 GraphQL 服務要求的值。這些與其他編程語言中的變量非常相似。例如，這是一個Person對象類型：

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

在這種情況下，字段分別是nameageString和保持一個和Int值。像上面顯示的對象字段可以用作查詢和突變字段（操作）的輸入。例如，請參閱Query以下內容：

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

people欄位正在向資料來源要求Person的所有執行個體。當您在 GraphQL Server Person 中新增或擷取資料時，您可以期望資料遵循類型和欄位的格式，也就是說，結構描述中的資料結構決定了資料在回應中的結構化方式：

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

字段在結構化數據中起著重要作用。下面還有幾個額外的屬性可以應用於更多的自定義字段解釋。

清單

列表返回指定類型的所有項目。列表可以使用括號添加到字段的類型[]：

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

```
}
```

在中Query，周圍的括號Person表示您想要將資料來源的Person所有執行個體作為陣列傳回。在回應中，每個的name和age值Person將以單一分隔的清單傳回：

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

您不僅限於特殊物件類型。您也可以在一般物件類型的欄位中使用清單。

非空值

非空值表示響應中不能為空的字段。您可以通過使用 ! 符號將字段設置為非空：

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

該name字段不能明確為空。如果您要查詢資料來源並為此欄位提供 null 輸入，則會擲回錯誤。

您可以結合列表和非空值。比較這些查詢：

```
type Query {
```

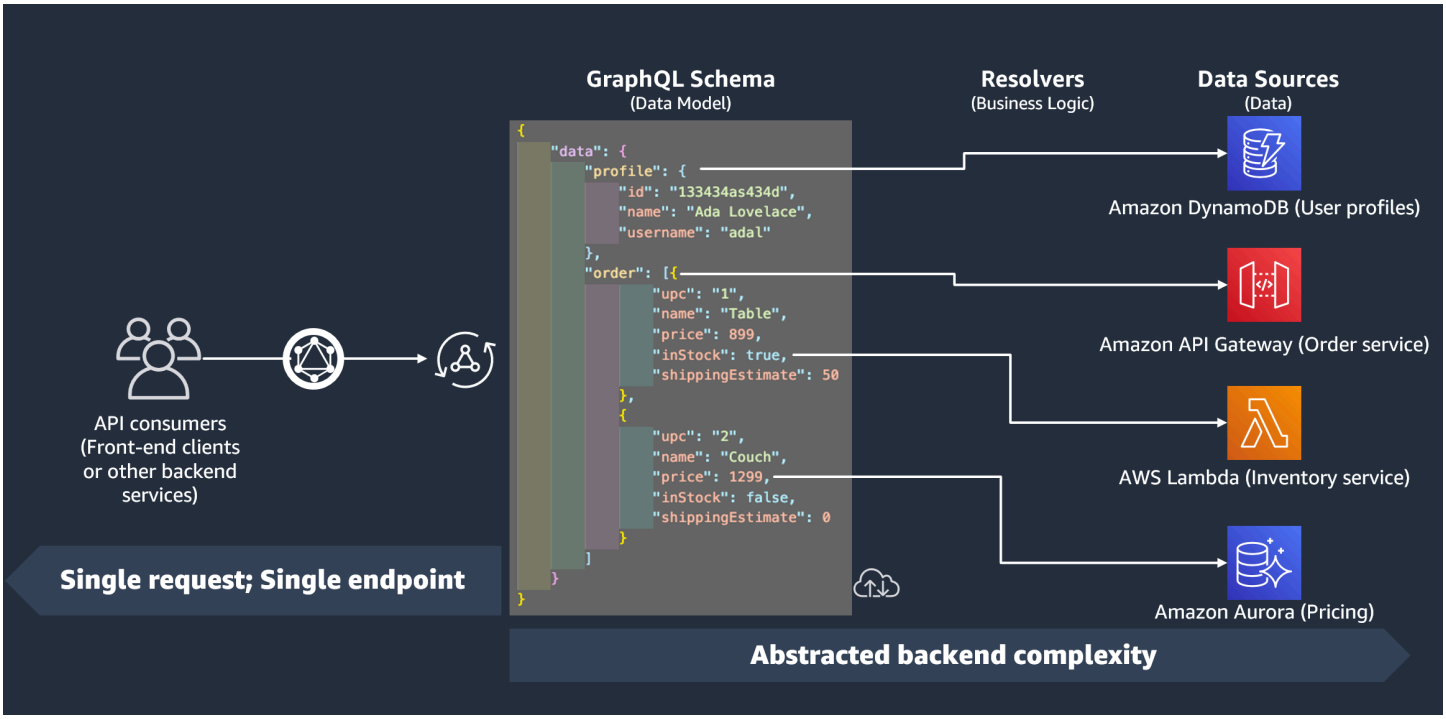
```
  people: [Person!]      # Use case 1
}

.
.
.

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

.
.
.

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

在使用案例 1 中，清單不能包含空項目。在使用案例 2 中，清單本身不能設定為 null。在使用案例 3 中，清單及其項目不能為空。但是，在任何情況下，您仍然可以返回空列表。

如您所見，GraphQL 中有許多移動元件。在本節中，我們展示了一個簡單的模式的結構以及模式支持的不同類型和字段。在下一節中，您將探索 GraphQL API 的其他元件，以及它們如何使用結構描述。

資料來源

在上一節中，我們了解到結構描述定義了資料的形狀。但是，我們從未解釋過這些數據來自何處。在實際專案中，您的結構描述就像是處理對伺服器發出的所有要求的閘道。發出請求時，結構描述充當與客戶端介面的單一端點。結構描述將存取、處理和轉送資料從資料來源回用戶端。請參閱下面的信息圖表：



AWS AppSync和 GraphQL 完美地實作前端後端 (BFF) 解決方案。它們通過抽象後端來大規模降低複雜性。如果您的服務使用不同的數據源和/或微服務，則基本上可以通過在單個模式（上標圖）中定義每個源（子圖）的數據的形狀來抽像一些複雜性。這表示您的 GraphQL API 不僅限於使用一個資料來源。您可以將任意數量的資料來源與 GraphQL API 建立關聯，並在程式碼中指定它們與服務互動的方式。

如您在資訊圖表中所見，GraphQL 結構描述包含用戶端請求資料所需的所有資訊。這意味著所有內容都可以在單個請求中處理，而不是像 REST 一樣處理多個請求。這些要求會經過結構描述，也就是服務的唯一端點。處理請求時，解析器（在下一節中說明）執行其代碼以處理來自相關數據源的數據。當返回響應時，綁定到數據源的子圖將與模式中的數據填充。

AWS AppSync支援許多不同的資料來源類型。在下表中，我們將描述每種類型，列出每種類型的一些優點，並為其他上下文提供有用的鏈接。

資料來源	描述	優勢	補充資訊

Amazon DynamoDB	「Amazon DynamoDB 是一種全受管的 NoSQL 資料庫服務，可提供快速且可預測的效能	• 大規模效能：DynamoDB 的設計是以任何規模的一致效能為基礎。這可以通過使用分	• 官方文件
			  • 分區
			  • 自動擴展
			  • 容錯能力
			  • 監控

資料來源	描述	優勢	補充資訊
	以及無縫的可擴展性。DynamoDB 是全受管的 NoSQL 資料庫服務，可讓您卸下操作及擴展分散式資料庫的管理負擔，不再需要煩惱硬體佈建、設定和組態、複寫、軟體修補或叢集擴展。DynamoDB 還提供靜態加密，從而消除了保護敏感資料所涉及的操作負擔和複雜性。」	區。DynamoDB 會自動將您的表格分割為數個配置，這些配置將儲存在多個節點的多個 SSD 中。這通常會增加網路輸送量並減少延遲。 • 大規模容量：DynamoDB 會監控您的流量，並允許您在網路長時間超載時自動調整輸送量。 • 可用性和容錯能力：數個實體隔離的區域支援 DynamoDB，每個區域都包含數個實體隔離的可用區域。DynamoDB 會在服務中斷時自動切換至備份區域。您也可以手動備份和複製資料，以確保資料安全。 • 記錄和監控：DynamoDB 為您的表格提供數種分析工具。您可以監視表格的效能並建立警示，以通知您服務的劇烈變更。	• 安全性 • GraphQL 和 DynamoDB • 適用於 DynamoDB 的解析程式作業 • 定價模式

資料來源	描述	優勢	補充資訊
		• 安全性：DynamoDB 遵循嚴格的通訊協定，以確保您的資料符合組織的安全性需求。 • 與整合 AWS AppSync：DynamoDB 與我們的服務無縫整合。您可以建立新的 DynamoDB 表格，並從中自動產生結構描述，以簡化開發程序。我們還提供完整的操作集合，以便從解析器中帳戶中的現有 DynamoDB 表格輕鬆請求資料。	

資料來源	描述	優勢	補充資訊

AWS Lambda	「AWS Lambda是一種運算服務，可讓您在不佈建或管理伺服器的情況下執行程式碼。
	
	Lambda 在高可用性的運算基礎設施上執行您的程式碼，並執行所有運算資源的管理，包括伺服器與作業系統維護、容量佈建與自動擴展以及記錄。有了 Lambda，您所需要做的就是以 Lambda 支援的其中一種語言執行階段提供程式碼。」
		• P ay-as-you-use 模型：Lambda 只會在您使用其資源時向您收取費用。它們還允許您擴展與應用程序需求使用的資源量。
		• 自動調整規模：有時您的應用程式可能需要額外的運算能力來執行特定程序 Lambda 可讓您自動擴充運算資源，以符合應用程式的需求。
		• 更快的部署時間：您可以透過部署套件簡化開發程序。使用套件將函數程式碼上傳至 Lambda 服務。然後，您可以使用它們的運行時環境來測試和執行您的函數。
		• 多功能性：Lambda 可用於多種使用案例。您可以將 Lambda 與第三方服務和服AWS務無縫整合。一些例子包括 CI/CD 管道和大量郵件服務。
		• 與整合 AWS AppSync：您可以
			• 官方文件
			  • 擴展
			  • 部署
			  • 執行階段
			• Lambda 解析器教程
			• 定價模式

資料來源	描述	優勢	補充資訊
		輕鬆地在解析器中叫用 Lambda 函數來處理請求。我們的服務提供簡化的請求操作來執行 Lambda 呼叫。我們允許單一呼叫和批次呼叫。	

資料來源	描述	優勢	補充資訊
OpenSearch	「Amazon OpenSearch 服務是一種受管服務，可讓您輕鬆在AWS雲端中部署、操作和擴展 OpenSearch 叢集。Amazon OpenSearch 服務支持 OpenSearch 和傳統的彈性搜索 OSS（最高 7.10，軟件的最終開源版本）。在您建立叢集時，您可選擇要使用的搜尋引擎。 OpenSearch是完全開放原始碼的搜尋和分析引擎，適用於日誌分析、即時應用程式監控和點擊流分析等使用案例。如需詳細資訊，請參閱 OpenSearch 文件。 Amazon OpenSearch 服務會為您的 OpenSearch 叢集佈建所有資源並啟動它。它也會自動偵測並取代故障的 OpenSearch 服務節點，減少與自我管理基礎架構相關的額外負荷。您可以通過單個 API 調用或	• 擴展：您可以透過 OpenSearch 無伺服器輕鬆擴展服務以符合您的服務需求。 • 資料擷取：您可以使用 OpenSearch 擷取來匯入、處理和分析資料。有許多用於數據獲取的應用程序，您可以在這裡找到。 • 安全性：OpenSearch 可以管理您的AWS安全配置，包括 IAM CloudTrail，VPC，身份驗證等。 • 可用性：在其服務中 OpenSearch 也支援不同的區域和可用區域。 • 與整合 AWS AppSync：在中 AWS AppSync，您可以使用 GraphQL API 來儲存和擷取帳戶中現有 OpenSearch 服務網域的資料。	• 官方文件 • 無伺服器 • 定價模式

資料來源	描述	優勢	補充資訊
	在控制台中單擊幾下來擴展集群。」		
HTTP 端點	您可以使用 HTTP 端點做為資料來源。AWS AppSync 可以使用參數和有效負載等相關信息將請求發送到端點。HTTP 響應將暴露給解析器，該解析器將在完成其操作後返回最終響應。	• 對於未與 Lambda 等服務整合的簡單應用程式非常有用。	• 解析器參考

資料來源	描述	優勢	補充資訊

Amazon EventBridge	「EventBridge 是一種使用事件將應用程式元件連接在一起的無伺服器服務，讓您更輕鬆地建置可擴充的事件驅動型應用程式。使用它可將事件從本土應用程式、AWS服務和協力廠商軟體等來源路由到整個組織的消費者應用程式。 EventBridge 提供簡單且一致的方式來擷取、篩選、轉換和傳遞事件，讓您可以快速建置新的應用程式。」	• 事件驅動架構：您可以利用事件驅動的架構。
		• 排程：您可以使用排程 EventBridge 器，使用 cron 運算式自動執行工作和規則，或設定時間間隔作為事件模式的替代方案。
		• 管道：使用 P EventBridge ipes，您可以使用管道取代事件匯流排，其中包括其他篩選事件模式，以及透過資料轉換進行擴充，然後再將事件傳送至目標。
		• 與集成 AWS AppSync：AWS AppSync允許您使用解析器將事件發送到事件總線。
			• 官方文件
			  • 管道
			  • 排程器
			• 解析器參考
			• 定價模式

資料來源	描述	優勢	補充資訊

關聯式資料庫	「Amazon Relational Database Service (Amazon RDS) 是一種 Web 服務，可讓您更輕鬆地在AWS雲端中設定、操作和擴展關聯式資料庫。它為業界標準的關聯式資料庫提供符合成本效益且可調整大小的容量，並管理常見的資料庫管理工作。」	• 輕鬆管理：RDS 會定期對其資源執行維護。維護通常需要更新資料庫執行個體的基礎硬體、基礎作業系統 (OS) 或資料庫引擎版本。在一般情況下，您可以決定何時執行更新 (例外包括安全性修補程式)。
		• 建議：RDS 的建議功能提供自動化建議，以修正執行個體中的潛在問題。
		• 可用性：RDS 在世界各地的不同實體區域提供。您可以輕鬆地將數據庫需求分配到不同的節點，以便為客戶提供更好的服務。
		• 定制：RDS 是為滿足大型企業的需求量身定制的。RDS 為運算、快速部署、可擴充性和儲存提供各種選項。
		• 安全性：RDS 與多種工具和服務集成在一起，以維護用戶，數據庫和網絡
			• 官方文件
			  • 功能
			  • Maintenance (維護)
			  • 建議
			  • 儲存選項
			  • 可用性
			  • 安全性
			• 定價模式

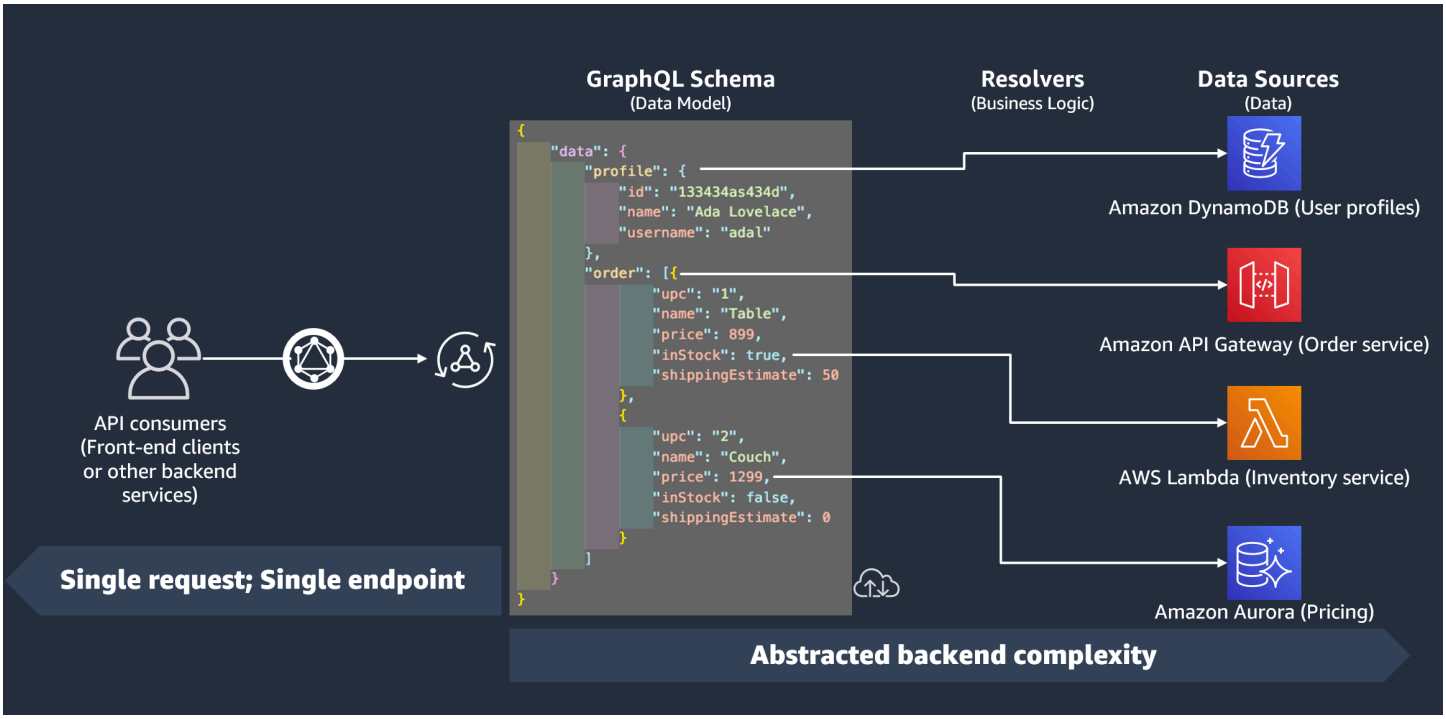
資料來源	描述	優勢	補充資訊
		級別的數據庫安全性。 • 與整合 AWS AppSync：如果您正在尋找成熟的後端解決方案，AWS AppSync 可讓您使用執行個體做為資料來源來傳送、處理、儲存和傳回資料。	
無資料來源	如果您不打算使用資料來源服務，可以將其設定為none。none資料來源雖然仍明確分類為資料來源，但不是儲存媒體。儘管如此，它在某些情況下對於數據操作和傳遞仍然很有用。	• 對於數據轉換等內容可能有用 • 在本地解決某些問題時很	• 解析器參考

> **Tip**
>
> 有關資料來源如何與之互動的詳情AWS AppSync，請參閱附加資料來源。

解析器

從前面的章節中，您瞭解結構描述和資料來源的元件。現在，我們需要解決結構描述和資料來源的互動方式。這一切都始於解析器。

解析器是一種代碼單元，用於處理向服務發出請求時如何解析該字段的數據。解析器附加到模式中類型中的特定字段。它們最常用於實現查詢，突變和訂閱字段操作的狀態更改操作。解析器將處理客戶端的請求，然後返回結果，它可以是一組輸出類型，如對象或標量：



解析器運行時

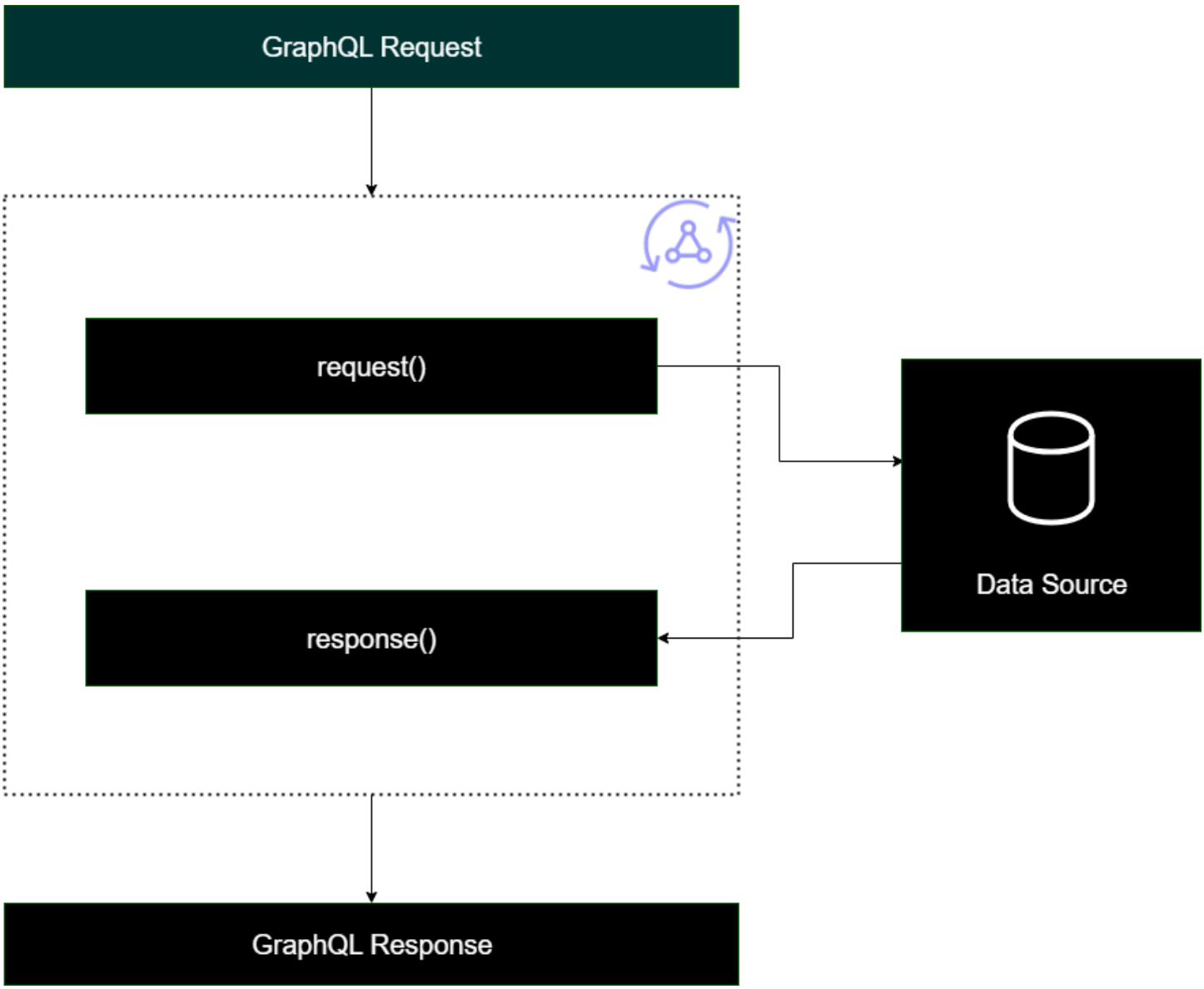
在中AWS AppSync，您必須先指定解析程式的執行階段。解析器運行時指示在其中執行解析器的環境。它還規定了您的解析器將使用的語言。 AWS AppSync目前支援用於 JavaScript 和速度範本語言 (VTL) 的應用程式 SYNC_JS。請參閱 VTL 的解析器和函數的JavaScript 運行時功能 JavaScript 或解析器映射模板實用程序參考。

解析器結構

代碼方面，解析器可以通過幾種方式進行構造。有單元和管道解析器。

單位解析器

單元解析器由定義針對數據源執行的單個請求和響應處理程序的代碼組成。請求處理程序將上下文對象作為參數，並返回用於調用數據源的請求有效負載。響應處理程序從數據源接收有效負載，並帶有執行請求的結果。回應處理常式會將有效負載轉換成 GraphQL 回應，以解析 GraphQL 欄位。



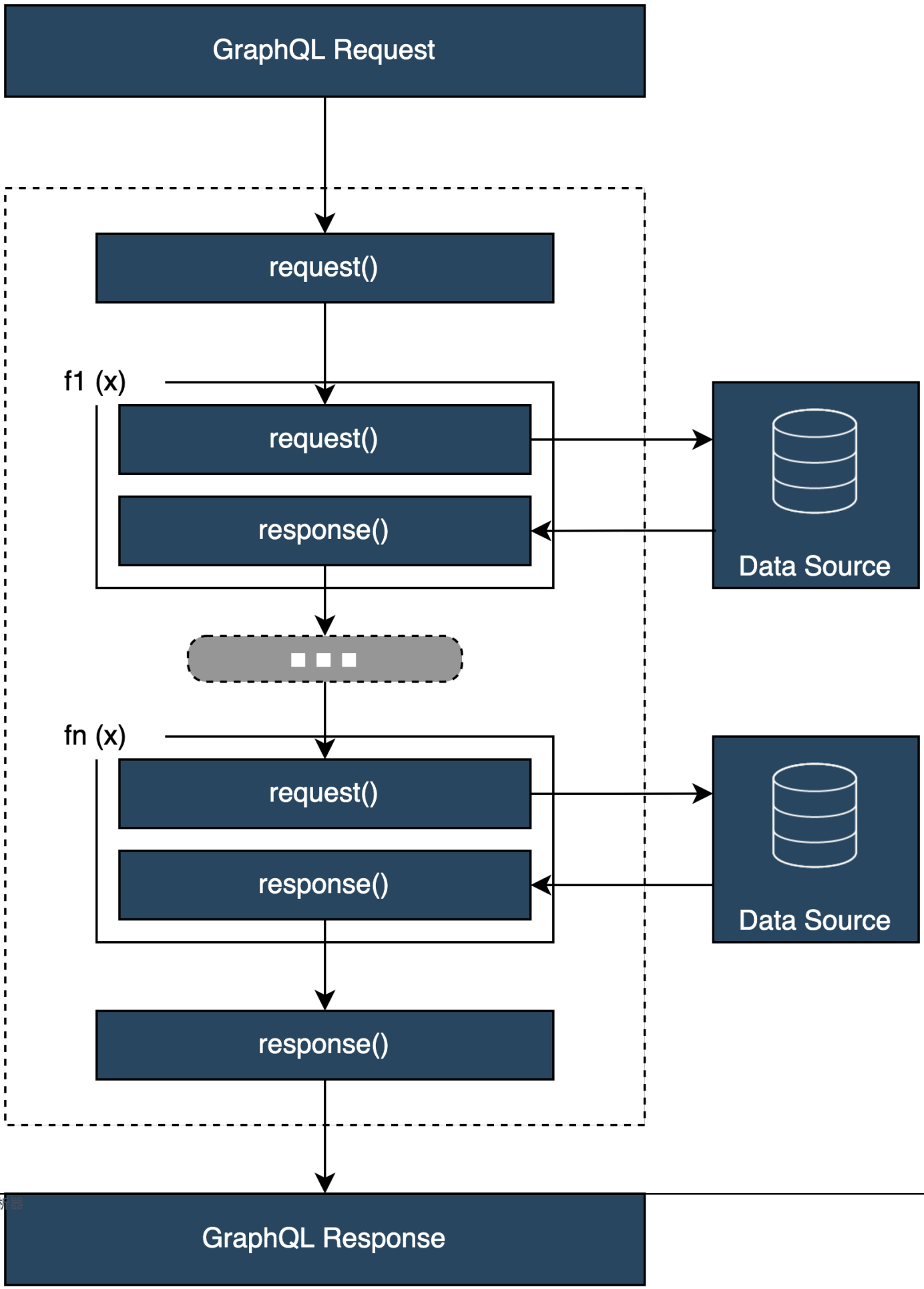
管道解析器

在實施管道解析器時，它們遵循一個通用的結構：

- 在步驟之前：當客戶端提出請求時，正在使用的模式字段（通常是您的查詢，突變，訂閱）的解析器將傳遞請求數據。解析器將開始使用 before step 處理程序處理請求數據，這允許在數據通過解析器移動之前執行一些預處理操作。
- 函數（S）：在步驟運行之前，請求被傳遞到函數列表。清單中的第一個函數會針對資料來源執行。函數是解析器代碼的子集，其中包含自己的請求和響應處理程序。請求處理器將採取請求數據並針對數據源執行操作。響應處理程序將數據源的響應傳遞回列表之前處理數據源的響應。如果有多個函

數，請求數據將被發送到要執行的列表中的下一個函數。列表中的函數將按照開發人員定義的順序進行連續執行。一旦所有的函數都被執行，最終的結果被傳遞給後一步。

- 之後步驟：之後的步驟是一個處理函數，允許您在將其傳遞給 GraphQL 響應之前對最終函數的響應執行一些最終操作。


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

解析程式處理程式結構

處理程序通常是調用Request和的函數Response：

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

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

在單元解析器中，只有一組這些功能。在管道解析器中，將有一組用於步驟之前和之後的步驟和每個函數的附加集合。為了可視化這看起來如何，讓我們回顧一個簡單的Query類型：

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

這是一個簡單的查詢，其中包含一個名為 type helloWorld 的字段String。讓我們假設我們總是希望這個字段返回字符串「Hello World」。為了實現這種行為，我們需要將解析器添加到此字段中。在單元解析器中，我們可以添加如下所示的內容：

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

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

request可以保留空白，因為我們沒有請求或處理數據。我們也可以假設我們的數據源是None，表明此代碼不需要執行任何調用。響應只是返回「你好世界」。為了測試這個解析器，我們需要使用查詢類型的請求：

```
query helloWorldTest {
  helloWorld
}
```

這是一個名為helloWorldTest返回helloWorld字段的查詢。執行時，字helloWorld段解析器也會執行並返回響應：

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

像這樣返回常量是你可以做的最簡單的事情。實際上，您將返回輸入，列表等。這是一個更複雜的例子：

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

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

在這裡，我們返回的列表Books。假設我們使用 DynamoDB 資料表來儲存書籍資料。我們的處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;
}
```

我們的請求使用內置的掃描操作來搜索表中的所有條目，將發現項目存儲在上下文中，然後將其傳遞給響應。響應採取的結果項，並在響應中返回它們：

```
{
  "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，處理程序鏈中的每個步驟都必須知道先前步驟中數據的狀態。來自一個處理程序的結果可以被存儲並作為參數傳遞給另一個處理程序。GraphQL 定義了四個基本的解析器參數：

解析器基本參數	描述
`obj`、`root`、`parent` 等等	父項的結果。
`args`	提供給 GraphQL 查詢中欄位的引數。
`context`	提供給每個解析器並保存重要的上下文信息，例如當前登錄的用戶或訪問數據庫的值。
`info`	保存與當前查詢相關的字段特定信息以及模式詳細信息的值。

在中AWS AppSync，`context`(ctx) 引數可以容納上述所有資料。它是每個請求創建的對象，包含諸如授權憑據，結果數據，錯誤，請求元數據等數據。上下文是程序員操縱來自請求其他部分的數據的簡單方法。再次採取這個片段：

```
/**
 * 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;
}
```

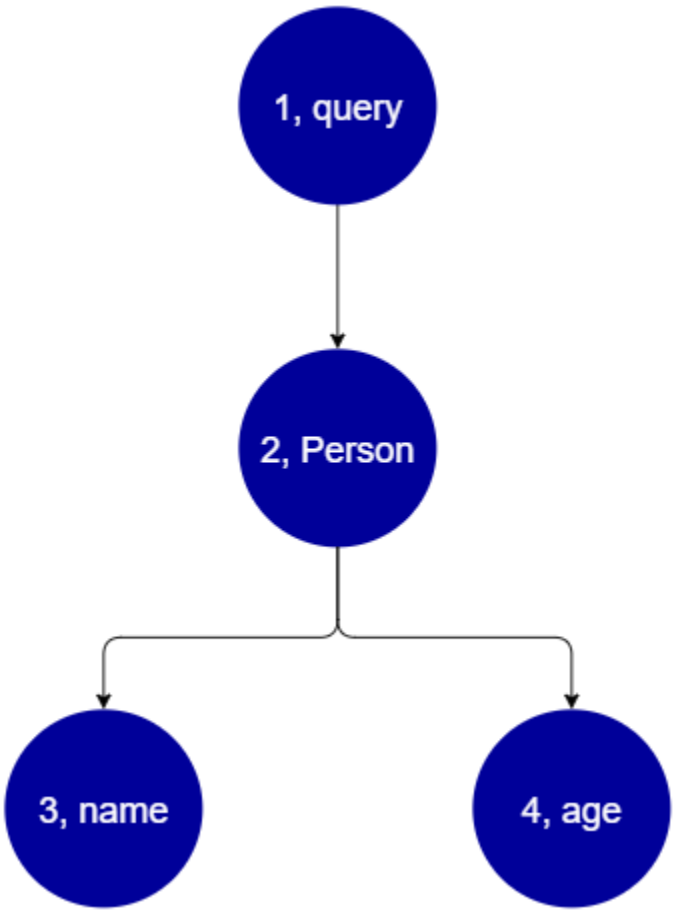
該請求被賦予上下文（ctx）作為參數; 這是請求的狀態。它對表中的所有項目執行掃描，然後將結果存儲回中的上下文中result。然後，上下文會傳遞至回應引數，該引數會存取result並傳回其內容。

請求和解析

當您對 GraphQL 服務進行查詢時，它必須先執行剖析和驗證程序，才能執行。您的請求將被解析並轉換為抽象語法樹。樹的內容是透過針對您的結構描述執行多種驗證演算法來驗證。驗證步驟後，樹的節點遍歷和處理。調用解析器，結果存儲在上下文中，並返回響應。例如，採取以下查詢：

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

我們正在返回Person與name和age欄位。運行此查詢時，樹將如下所示：



從樹狀結構中，這個要求會在結構描述Query中搜尋根目錄。在查詢內部，該Person字段將被解析。從前面的例子中，我們知道這可能是來自用戶的輸入，值列表等Person很可能與包含我們需要的字段的對象類型相關聯（name和age）。一旦找到這兩個子字段，它們將按給定的順序（name後跟age）進行解析。一旦樹狀結構完全解決，請求就會完成並傳送回用戶端。

GraphQL 的其他屬性

GraphQL 由幾個設計原則組成，以保持大規模的簡單性和耐用性。

宣告式

GraphQL 是聲明式的，這意味著用戶將通過僅聲明他們想要查詢的字段來描述（塑造）數據。回應只會傳回這些屬性的資料。*##################Book##### ISBN 13 id ## 9780199536061#*

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

```
    year
    author
  }
}
```

響應將返回有效負載（name，和author）中的字段year，並且沒有其他內容：

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

由於這種設計原則，GraphQL 消除了 REST API 在複雜系統中處理的過度和不足抓取問題。這樣可以更有效率地收集資料並改善網路效能。

階層

GraphQL 具有靈活性，因為請求的數據可以由用戶塑造以適應應用程序的需求。請求的數據始終遵循 GraphQL API 中定義的屬性的類型和語法。例如，下列程式碼片段會顯示新欄位範圍的getBook作業，quotes該作業會傳回連結至 Book *9780* 199536061 的所有已儲存引號字串和頁面：

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

執行此查詢會傳回下列結果：

```
{
```

```
  "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
         }
      ]
    }
  }
}
```

正如你所看到的，鏈接到所請求的書的quotes字段以我們的查詢描述的格式相同的數組返回。儘管此處未顯示，但 GraphQL 具有額外的優點，就是不要特別關注它所擷取的資料位置。 Books並且quotes可以單獨存儲，但只要關聯存在，GraphQL 仍然會檢索信息。這意味著您的查詢可以在單個請求中檢索大量的獨立數據。

內省

GraphQL 是自我記錄或內省性。它支持幾個內置操作，允許用戶查看模式中的基礎類型和字段。例如，下面是一個帶有date和description字段的Foo類型：

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

我們可以使用該_type操作來查找模式下的鍵入元數據：

```
{
  __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
      }
    }
  }
}
```

這將返回一個響應：

```
{
  "__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
      },
    ]
  }
}
```

此功能可用於找出特定 GraphQL 結構描述支援的類型和欄位。GraphQL 支援各種各樣的這些內省性作業。如需詳細資訊，請參閱內部檢查。

強大的打字

GraphQL 通過其類型和字段系統支持強大的打字。當您在模式中定義某些內容時，它必須具有可以在運行時之前驗證的類型。它還必須遵循 GraphQL 的語法規範。這個概念是從其他語言編程沒有什麼不同。例如，以下是較早的Foo類型：

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

```
  description: String
}
```

我們可以看到，Foo是將被創建的對象。在的實例中Foo，會有一個date和description字段，這兩個String原始類型（純量）。在語法上，我們看到這Foo是聲明的，並且它的領域存在於其範圍內。這種類型檢查和邏輯語法的組合可確保 GraphQL API 簡潔且不言而喻。GraphQL 的類型和語法規範可以在這裡找到。

開始使用：建立您的第一個圖形 SQL API

您可以使用AWS AppSync用於設定和啟動 GraphQL API 的主控台。圖形 SQL API 通常需要三個元件：

1. 图形 SQL 模式-您的 GraphQL 結構描述是 API 的藍圖。它定義了執行操作時，您可以請求的類型和字段。若要在結構描述中填入資料，您必須將資料來源連線到 GraphQL API。在本快速入門指南中，我們將使用預先定義的模型建立結構定義。
2. 資料來源-這些資源包含用於填入 GraphQL API 的資料。這可以是一個動態資料表、Lambda 函數等。AWSAppSync支援多種資料來源，以建置強大且可擴充的 GraphQL API。資料來源會連結至資料架構中的欄位。每當對字段執行請求時，源中的數據都會填充該字段。此機制由解析器控制。在本快速入門指南中，我們將使用預先定義的模型與結構描述一起建立資料來源。
3. 解析器-解析器負責將結構描述欄位連結至資料來源。他們從源中檢索數據，然後根據字段定義的內容返回結果。AWSAppSync支持兩者JavaScript和 VTL，用於為您的 GraphQL API 編寫解析器。在本快速入門指南中，將根據結構描述和資料來源自動產生解析器。我們不會在本節中深入研究這一點。

AWS AppSync支援建立和設定所有 GraphQL 元件。開啟主控台時，您可以使用下列方法建立 API：

1. 透過預先定義的模型產生自訂 GraphQL API，並設定新的 DynamoDB 表格 (資料來源) 來支援它，藉此設計一個自訂的 GraphQL API。
2. 設計具有空白結構描述且沒有資料來源或解析器的 GraphQL API。
3. 使用 DynamoDB 表格匯入資料並產生結構描述的類型和欄位。
4. 使用AWS AppSync的WebSocket可開發即時 API 的功能和發佈/訂閱架構。
5. 使用現有的 GraphQL API (來源 API) 連結至合併的 API。

> **Note**
>
> 我們建議您檢閱設計結構描述在使用更高級工具之前的部分。這些指南將解釋更簡單的示例，您可以在概念上使用它們來構建更複雜的應用程序AWS AppSync。

AWS AppSync也支援數個非主控台選項來建立 GraphQL API。其中包含：

1. AWS Amplify

2. AWS SAM
3. AWS CloudFormation
4. CDK

下列範例將示範如何使用預先定義的模型和 DynamoDB 建立 GraphQL API 的基本元件。

主題

步驟 1：啟動結構描述

在此範例中，您將建立Todo允許使用者建立的 APITodo日常瑣事提醒的項目，例如####或者####。此 API 將示範如何使用 GraphQL 作業，在 DynamoDB 資料表中的狀態持續存在。

從概念上講，建立您的第一個 GraphQL API 有三個主要步驟。您必須定義模式（類型和字段），將數據源附加到您的字段，然後編寫處理業務邏輯的解析器。不過，主控台體驗會變更此順序。我們將通過定義如何我們希望我們的數據源與我們的模式進行交互開始，然後在稍後定義模式和解析器。

若要建立您的圖形 SQL API

1. 登入 AWS Management Console 並開啟 AppSync主控台。
2. 在儀表板上，選擇 Create API (建立 API)。
3. 雖然圖形 SQL API已選取，選擇從頭開始設計。然後選擇 Next (下一步)。
4. 對於API 名稱，將預先填入的名稱變更為**`Todo API`**，然後選擇下一步。

   > **Note**
   >
   > 這裡還有其他選項，但我們不會在這個例子中使用它們。

5. 在指定圖形 SQL 資源區段中，執行下列動作：

   a. 選擇立即建立由 DynamoDB 表格支援的類型。

> **Note**
>
> 這表示我們將建立一個新的 DynamoDB 表格以作為資料來源連接。

b. 在型號名稱欄位中，輸入**Todo**。

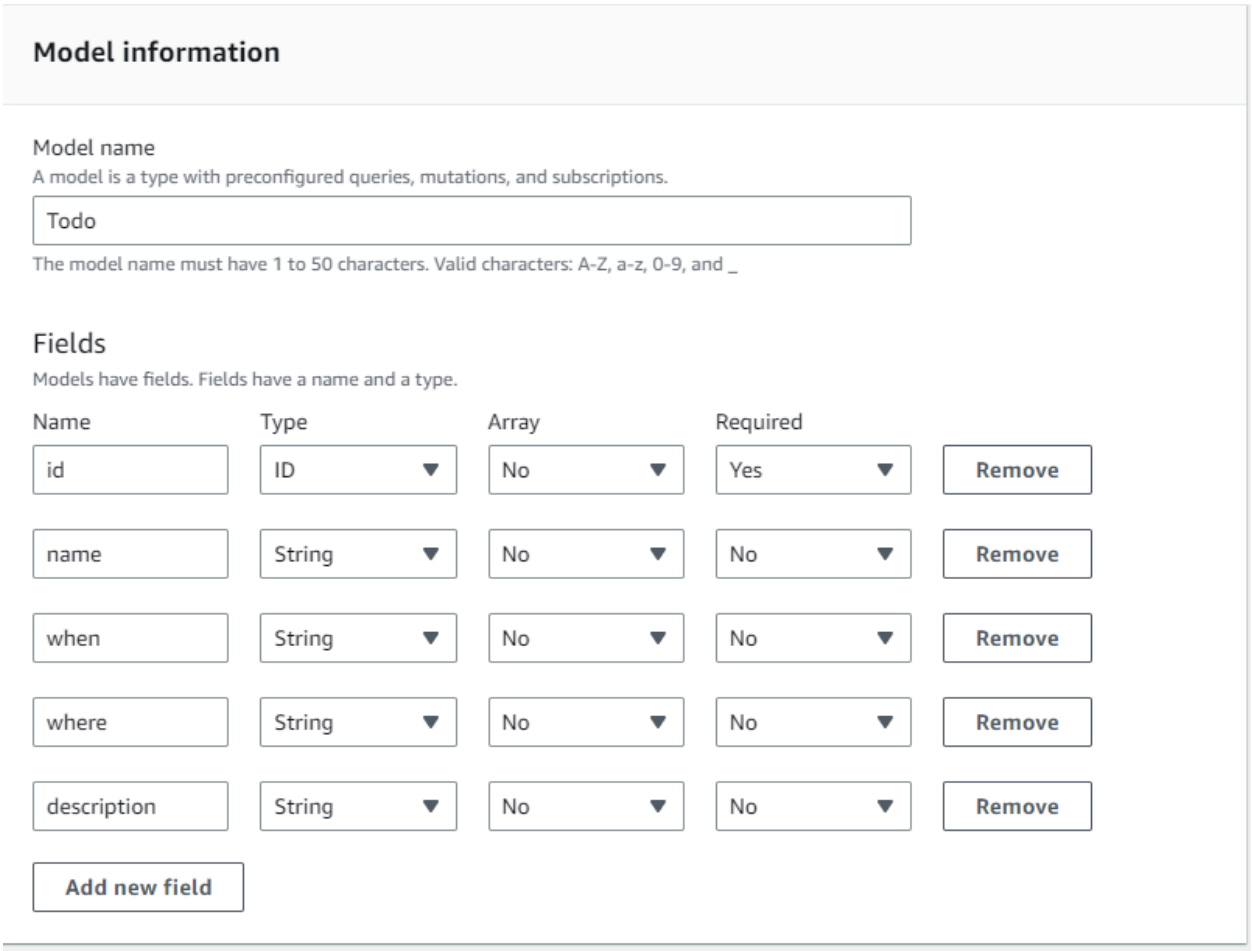
> **Note**
>
> 我們的第一個要求是定義我們的架構。這個型號名稱將是類型名稱，所以你真正做的是創建一個type呼叫Todo將存在於模式中：
>
> ```
> type Todo {}
> ```

c. 下欄位，請執行下列動作：

i. 建立名為的欄位**id**，與類型ID，且必要設定為Yes。

> **Note**
>
> 這些是將在您的範圍內存在的字段Todo類型。您的欄位名稱將會被稱為id與一個類型ID!:
>
> ```
> type Todo {
>  id: ID!
> }
> ```
>
> AWS AppSync支援不同使用案例的多個標量值。

ii. 使用新增欄位，建立四個其他欄位Name值設定為**name**,**when**,**where**，以及**description**。他們的Type值將是String，以及Array和Required值都將被設置為No。其會如下所示：



> **Note**
>
> 完整類型及其欄位如下所示：
>
> ```
> type Todo {
>  id: ID!
>  name: String
>  when: String
>  where: String
>  description: String
> }
> ```
>
> 因為我們正在使用此預定義模型創建一個模式，因此它還將根據類型填充幾個樣板突變，例如`create`,`delete`，以及`update`協助您輕鬆填入資料來源。

d. 下規劃模型表，輸入表格名稱，例如**`TodoAPITable`**。設定主要索引鍵至`id`。

> **Note**
>
> 我們基本上正在創建一個名為的新的 DynamoDB 表###將作為我們的主要數據源附加到 API。我們的主鍵設置為必需的`id`我們在此之前定義的字段。請注意，這個新表是空白的，除了分區鍵以外不包含任何內容。

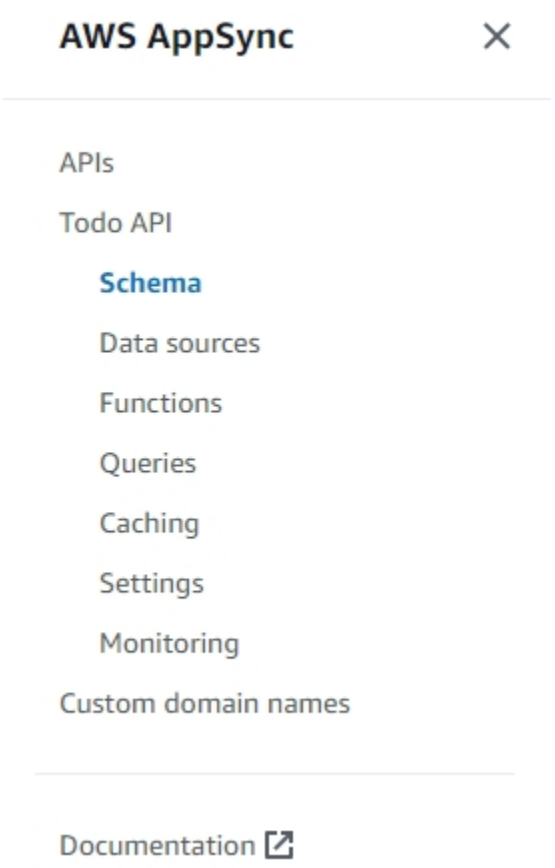
e. 選擇 下一步。

6. 檢閱您的變更並選擇建立 API。請稍等片刻，讓AWS AppSync服務完成建立您的 API。

您已成功建立具有結構描述和 DynamoDB 資料來源的 GraphQL API。為了總結上述步驟，我們選擇創建一個全新的 GraphQL API。我們定義了 API 的名稱，然後通過添加我們的第一個類型添加了我們的模式定義。我們定義了類型及其欄位，然後選擇將資料來源附加到其中一個欄位，方法是建立不含任何資料的新 DynamoDB 表格。

步驟 2：導覽主機

在將資料新增至 DynamoDB 表之前，我們應該先檢閱AWS AppSync主控台體驗。該AWS AppSync頁面左側的「主控台」索引標籤可讓使用者輕鬆瀏覽至任何主要元件或組態選項，AWS AppSync提供：



架構設計師

選擇綱要以檢視您剛建立的結構描述。如果您檢閱結構描述的內容，您會注意到它已載入許多協助程式作業，以簡化開發程序。在綱要編輯器，如果你滾動代碼，你最終會到達你在上一節中定義的模型：

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

您的模型成為整個結構描述中使用的基底類型。我們將開始使用從這種類型自動生成的突變將數據添加到我們的數據源。

這裡有一些額外的提示和事實綱要編輯器：

1. 程式碼編輯器具有連結和錯誤檢查功能，您可以在撰寫自己的應用程式時使用這些功能。

2. 主控台右側顯示則會顯示已建立的 GraphQL 類型，以及不同最上層類型 (例如查詢) 的解析程式。
3. 將新類型添加到模式時（例如，`type User {...}`），你可以有AWS AppSync為您佈建動態資源。這些包括最符合您 GraphQL 資料存取模式的適當主索引鍵、排序索引鍵和索引設計。如果您選擇最上方的 Create Resources (建立資源)，並從選單選取其中一個使用者定義的類型，您便可以在結構描述設計中選擇不同的欄位選項。我們將在設計資料架構部分。

解析器配置

在結構描述設計工具中，解析器區段包含結構描述中的所有類型和欄位。如果您捲動欄位清單，您會注意到您可以透過選擇將解析器附加到某些欄位貼附。這將打開一個代碼編輯器，您可以在其中編寫解析器代碼。AWSAppSync同時支援 VTL 和JavaScript執行階段，可透過選擇在頁面頂端進行變更動作，然後更新執行期。在頁面底部，您還可以創建將按順序運行多個操作的函數。但是，解析器是一個高級主題，我們將不會在本節中介紹該主題。

資料來源

選擇資料來源以檢視您的動態資料表。通過選擇`Resource`選項 (如果可用)，您可以檢視資料來源的組態。在我們的範例中，這會導致 DynamoDB 主控台。從那裡，您可以編輯數據。您也可以選擇資料來源，然後選擇來直接編輯某些資料編輯。如果您需要刪除資料來源，可以選擇資料來源，然後選取刪除。最後，您可以選擇建立新的資料來源建立資料來源，然後設定名稱和類型。請注意，此選項用於鏈接AWS AppSync服務到現有的資源。您仍然需要使用相關服務在您的帳戶中創建資源之前AWS AppSync承認它。

查詢

選擇查詢查看您的查詢和突變。當我們使用我們的模型創建我們的 GraphQL API 時，AWS AppSync 自動生成一些幫助突變和查詢用於測試目的。在查詢編輯器中，左側包含探險者。這是一個列表，顯示您的所有突變和查詢。您可以通過單擊名稱值輕鬆啟用要在此處使用的操作和字段。這將導致代碼自動出現在編輯器的中間部分。在這裡，您可以通過修改值來編輯突變和查詢。在編輯器的底部，你有查詢變數編輯器，允許您輸入操作的輸入變量的字段值。選擇執行在編輯器頂部會彈出一個下拉列表來選擇要運行的查詢/突變。此執行的輸出將顯示在頁面的右側。早在探險者在頂部的部分，您可以選擇一個操作（查詢，突變，訂閱），然後選擇+符號添加該特定操作的新實例。在頁面頂端，會有另一個下拉式清單，其中包含查詢執行的授權模式。但是，我們將不會在本節中介紹該功能（如需詳細資訊，請參閱安全。）。

設定

選擇設定以檢視您的圖形 SQL API 的一些設定選項。您可以在此啟用某些選項，例如記錄、追蹤和 Web 應用程式防火牆功能。您還可以添加新的授權模式，以保護您的數據免受不必要的洩漏給公眾。但是，這些選項更高級，本節將不會介紹。

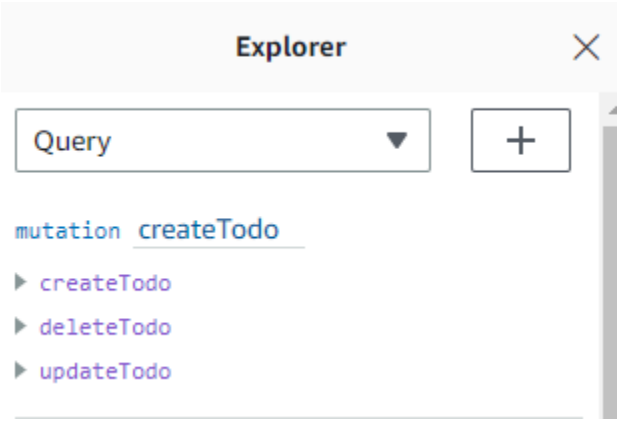
> **Note**
>
> 默認授權模式，API_KEY，使用 API 金鑰來測試應用程式。這是提供給所有新建立的 GraphQL API 的基本授權。我們建議您使用不同的生產方法。為了本節中的示例，我們將僅使用 API 密鑰。如需有關支援授權方法的詳細資訊，請參閱安全。

第 3 步：使用 GraphQL 突變添加數據

您的下一個步驟是使用 GraphQL 突變將資料新增至您的空白 DynamoDB 資料表。突變是 GraphQL 中的基本操作類型之一。它們在結構描述中定義，可讓您操作資料來源中的資料。就 REST API 而言，這些操作與類似的操作非常相似PUT或者POST。

若要將資料新增至資料來源

1. 如果您尚未這樣做，請登入AWS Management Console並打開AppSync安慰。
2. 從表格中選擇您的 API。
3. 在左側的索引標籤中，選擇查詢。
4. 在探險者標籤在表的左側，您可能會看到在查詢編輯器中已經定義了幾個突變和查詢：



> **Note**
>
> 這種突變實際上坐在你的模式中Mutation類型。它有代碼：
>
> ```
> type Mutation {
>  createTodo(input: CreateTodoInput!): Todo
>  updateTodo(input: UpdateTodoInput!): Todo
>  deleteTodo(input: DeleteTodoInput!): Todo
> }
> ```
>
> 如您所見，這裡的操作類似於查詢編輯器內部的操作。

AWS AppSync從我們之前定義的模型中自動生成這些。這個例子將使用createTodo突變添加條目到我們的###表。

5. 選擇合適的createTodo通過在下擴展操作createTodo突變：

啟用所有字段的複選框，如上圖所示。

> **Note**
>
> 您在這裡看到的屬性是突變的不同可修改元素。您的input可以被認為是參數createTodo。帶有複選框的各種選項是一旦執行操作，將在響應中返回的字段。

6. 在畫面中央的程式碼編輯器中，您會注意到該作業出現在createTodo突變：

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

Note

要正確解釋這個代碼片段，我們還必須查看模式代碼。該聲明`mutation createTodo($createtodoinput: CreateTodoInput!){}`是其操作之一的突變，`createTodo`。完整的突變位於模式中：

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

從編輯器返回突變聲明，該參數是一個名為的對象`$createtodoinput`具有必要的輸入類型`CreateTodoInput`。請注意`CreateTodoInput`（以及突變中的所有輸入）也在模式中定義。例如，這是樣板代碼`CreateTodoInput`:

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

它包含了我們在我們的模型，即定義的字段`name`,`when`,`where`，以及`description`。回到編輯器代碼，在`createTodo(input: $createtodoinput) {}`，我們將輸入聲明為`$createtodoinput`，這也是在突變聲明中使用的。我們這樣做是因為這允許 GraphQL 根據提供的類型驗證我們的輸入，並確保它們與正確的輸入一起使用。
編輯器程式碼的最後一部分會顯示執行作業後回應中傳回的欄位：

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

在查詢變數此編輯器下方的選項卡，將有一個通用createtodoinput可能具有以下數據的對象：

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

Note

這是我們為前面提到的輸入分配值的地方：

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

改變createtodoinput透過新增我們想要放入 DynamoDB 表格中的資訊。在這種情況下，我們想創建一些Todo作為提醒的項目：

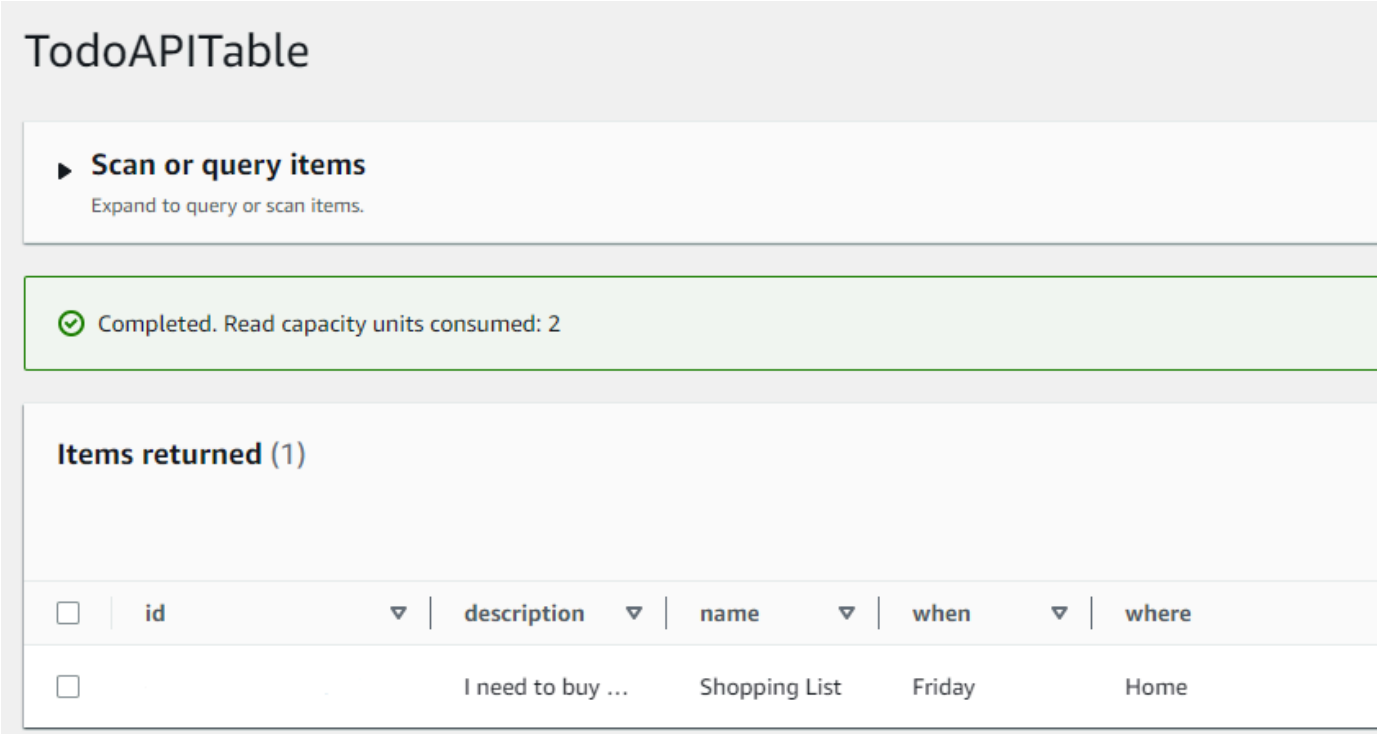
```
{
  "createtodoinput": {
    "name": "Shopping List",
```

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

7. 選擇執行在編輯器的頂部。選擇創建待辦事項在下拉列表中。在編輯器的右側，您應該會看到響應。這可能看起來如下：

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

如果您導覽至 DynamoDB 服務，您現在會在資料來源中看到包含以下資訊的項目：



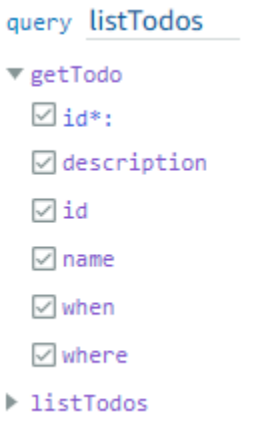
為了總結操作，GraphQL 引擎會剖析記錄，然後有一個解析器將其插入您的 Amazon DynamoDB 表格中。同樣地，您可以在 DynamoDB 主控台中進行驗證。請注意，您不需要傳遞一個`id`價值。一個`id`產生並在結果中傳回。這是因為這個例子使用了`autoId()`在 GraphQL 解析器中為 DynamoDB 資源上的磁碟分割金鑰集執行功能。我們將介紹如何在不同的部分構建解析器。注意返回的`id`值；您將在下一節中使用它，透過 GraphQL 查詢擷取資料。

步驟 4：使用圖形 SQL 查詢擷取資料

現在您的資料庫中存在記錄，您會在執行查詢時取得結果。查詢是 GraphQL 的其他基本操作之一。它用於解析和檢索數據源中的信息。就其餘 API 而言，這類似於`GET`操作。GraphQL 查詢的主要優點是能夠指定應用程式的確切資料需求，以便在正確的時間擷取相關資料。

若要查詢您的資料來源

1. 如果您尚未這樣做，請登入AWS Management Console並打開AppSync安慰。
2. 從表格中選擇您的 API。
3. 在左側的索引標籤中，選擇查詢。
4. 在探險者表格左側的標籤，下`query listTodos`，展開`getTodo`操作：



5. 在代碼編輯器中，您應該看到操作代碼：

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

在(id:"")」下方，填入您儲存在變異作業結果中的值。在我們的例子中，這將是：

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

6. 選擇執行，然後列表待辦事項。結果將顯示在編輯器的右側。我們的例子看起來像這樣：

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

> **Note**
>
> 查詢只會傳回您指定的欄位。您可以通過從返回字段中刪除它們來取消選擇不需要的字段：
>
> ```
> {
>     description
>     id
>     name
>     when
>     where
>   }
> ```
>
> 您也可以取消勾選探險者您要刪除的字段旁邊的標簽。

7. 您也可以嘗試listTodos重複這些步驟以在資料來源中建立項目，然後使用listTodos操作。以下是我們添加了第二個任務的示例：

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

通過調用listTodos操作，它返回舊的和新的條目：

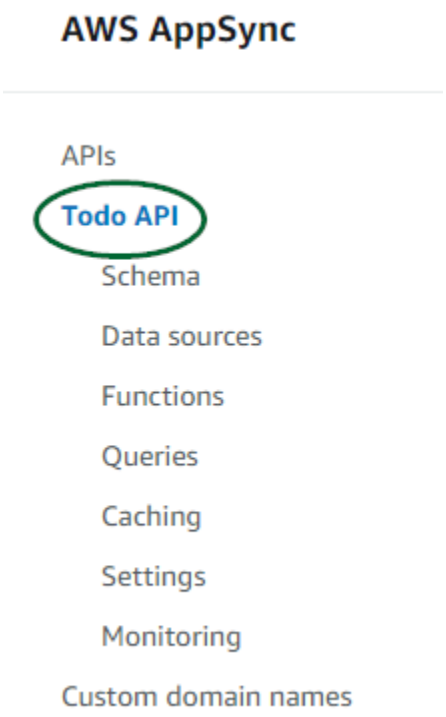
```
{
  "data": {
    "listTodos": {
      "items": [
        {
          "id": "abcdefgh-1234-1234-1234-abcdefghijkl",
          "name": "Shopping List",
          "when": "Friday",
          "where": "Home",
          "description": "I need to buy eggs"
        },
        {
          "id": "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee",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

補充區段

這些部分是更高級的參考AWS AppSync主題。我們建議遵循補充閱讀在做其他事情之前的部分。

整合

在主控台索引標籤中，如果您選擇 API 的名稱，整合頁面出現：



它總結了設定 API 的步驟，並概述建立用戶端應用程式的後續步驟。該與您的應用程式整合段落提供使用的詳細資訊AWS放大工具鏈自動化您的 API 與 iOS、安卓系統和連接的過程JavaScript通過配置和代碼生成應用程序。Amplify 工具鏈為從本機工作站建置專案提供完整支援，包括用於 CI/CD 的 GraphQL 佈建和工作流程。

該用戶端範例部分還列出了示例客戶端應用程序（例如，JavaScript，iOS，安卓），用於測試端到端體驗。您可以複製並下載這些範例，且組態檔案具有開始使用所需的必要資訊 (例如您的端點 URL)。按照上的說明進行操作AWS Amplify工具鏈頁面來運行您的應用程序。

補充閱讀

- 設計 GraphQL 的 API-這是使用沒有資料來源或解析器的空白結構描述建立 GraphQL 的綜合指南。

設計 GraphQL 的 API

AWS AppSync可讓您使用主控台體驗來建立 GraphQL API。您在 [啟動範例結構描述] 區段中看到了這一點。但是，該指南並未顯示您可以利用的選項和配置的整個目錄AWS AppSync。

當您選擇在主控台中建立 GraphQL API 時，有幾個選項可供探索。如果您遵循我們的啟動範例結構描述指南，我們將向您展示如何從預先定義的模型建立 API。在以下各節中，我們將引導您完成其餘的選項和配置，以便在中AWS AppSync創建 GraphQL API

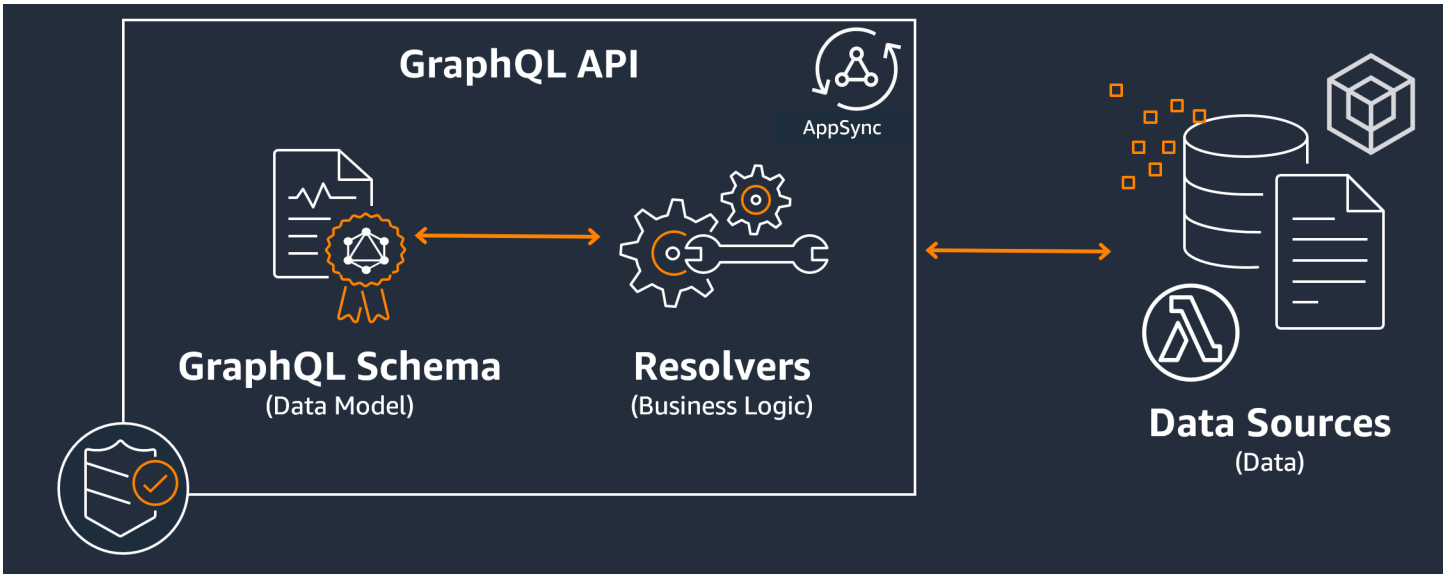
在本節中，您將檢閱下列概念：

1. Blank APIs or imports：本指南將執行建立 GraphQL API 的整個建立程序。您將學習如何從沒有模型的空白範本建立 GraphQL、為結構描述設定資料來源，以及將第一個解析器新增至欄位。
2. Real-time data：本指南將向您展示使用 WebSocket 引擎創建 API AWS AppSync 的潛在選項。
3. Merged APIs：本指南將說明如何透過關聯和合併來自多個現有 GraphQL API 的資料來建立新的 GraphQL API。
4. the section called “自省”：本指南將說明如何使用資料 API 整合 Amazon RDS 資料表。

建構圖形 SQL API (空白或匯入的 API)

在您從空白範本建立 GraphQL API 之前，這將有助於檢閱周圍 GraphQL 的概念。有一個圖形 SQL API 的三個基本組成部分：

1. 該模式是包含資料形狀和定義的檔案。當用戶端向 GraphQL 服務發出要求時，傳回的資料將遵循結構描述的規格。如需詳細資訊，請參閱綱要。
2. 該資料來源已附加至您的結構描述。當提出請求時，這是檢索和修改數據的地方。如需詳細資訊，請參閱Data sources。
3. 該解析器位於結構描述和資料來源之間。當發出請求時，解析器對來自源的數據執行操作，然後返回結果作為響應。如需詳細資訊，請參閱Resolvers。



AWS AppSync透過允許您建立、編輯和儲存結構描述和解析器的程式碼來管理 API。您的資料來源將來自外部儲存庫，例如資料庫、DynamoDB 資料表和 Lambda 函數。如果您使用AWS用於存儲您的數據或計劃這樣做的服務，AWS AppSync在關聯來自您的資料時，提供近乎無縫的體驗AWS您的圖形 SQL API 的帳戶。

在下一節中，您將學習如何使用AWS AppSync服務。

主題

- 步驟 1：設計結構描述
- 步驟 2：附加資料來源
- 步驟 3：配置解析器
- 第 4 步：使用 API：CDK 示例

步驟 1：設計結構描述

GraphQL 結構描述是任何 GraphQL 伺服器實作的基礎。每個圖形 SQL API 都是由一個定義單身包含描述如何填入要求資料的類型和欄位的結構描述。流經 API 的資料和執行的作業必須針對結構描述進行驗證。

在一般情況下，圖形 QL 類型系統說明 GraphQL 伺服器的功能，並用來判斷查詢是否有效。伺服器的型別系統通常稱為該伺服器的結構描述，而且可以由不同的物件類型、純量類型、輸入類型等組成。GraphQL 既是聲明式和強類型，這意味著類型將在運行時定義良好，並且只返回指定的內容。

AWS AppSync可讓您定義和設定 GraphQL 結構描述。下一節說明如何使用從頭開始建立 GraphQL 結構描述AWS AppSync的服務。

結構化圖形 SQL 結構描述

> **Tip**
>
> 我們建議您檢閱綱要繼續之前的部分。

GraphQL 是用於實作 API 服務的強大工具。根據圖形 SQL 的網站，圖形 SQL 如下所示：

「GraphQL 是 API 的查詢語言，也是用於使用現有資料完成這些查詢的執行階段。GraphQL 為您的 API 中的資料提供完整且易於理解的描述，讓客戶能夠準確地要求他們所需的內容，僅此而已，隨著時間的推移更容易發展 API，並啟用強大的開發人員工具。「

本節介紹 GraphQL 實作的第一部分，即結構描述。使用上面的引用，模式扮演「在 API 中提供完整且易於理解的數據描述」的作用。換句話說，GraphQL 結構描述是服務資料、作業以及它們之間關係的文字表示。結構描述被視為 GraphQL 服務實作的主要進入點。毫不奇怪，這通常是您在項目中製作的第一件事情之一。我們建議您檢閱綱要繼續之前的部分。

若要引用綱要區段中，圖形 SQL 結構描述會寫入結構定義語言(SDL). SDL 由具有既定結構的型別與欄位所組成：

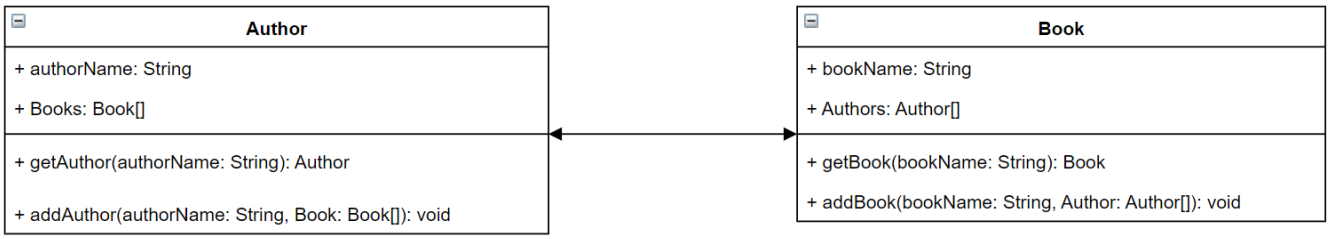
- 类型：類型是 GraphQL 定義資料形狀和行為的方式。GraphQL 支援多種類型，這些類型將在本節稍後加以說明。結構描述中定義的每個類型都會包含其自己的範圍。範圍內將是一個或多個可以包含將在 GraphQL 服務中使用的值或邏輯的欄位。類型填充許多不同的角色，最常見的是對象或標量（原始值類型）。
- 欄位：欄位存在於類型的範圍內，並保留 GraphQL 服務要求的值。這些與其他編程語言中的變量非常相似。您在欄位中定義的資料形狀將決定資料在要求/回應作業中的結構方式。這使開發人員可以在不知道如何實現服務後端的情況下預測將返回的內容。

最簡單的模式將包含三種不同的數據類別：

1. 綱要根：根定義綱要的進入點。它指向將對像添加，刪除或修改某些數據執行某些操作的字段。
2. 类型：這些是用於表示數據形狀的基本類型。您幾乎可以將這些視為對象或具有定義特徵的東西的抽象表示。例如，你可以做一個`Person`代表資料庫中人員的物件。每個人的特徵都將在內部定義`Person`作為字段。它們可以是任何人的姓名，年齡，工作，地址等。

3. 特殊物件類型：這些是定義模式中操作行為的類型。每個特殊物件類型會在每個綱要中定義一次。它們會先放置在結構描述根目錄中，然後在結構描述主體中定義。特殊物件類型中的每個欄位都會定義解析程式執行的單一作業。

想像一下，想像一下，您正在創建一個存儲作者和他們所寫的書籍的服務。每位作者都有一個名字和他們所撰寫的書籍陣列。每本書都有一個名字和相關作者的列表。我們還希望能夠添加或檢索書籍和作者。這種關係的一個簡單的 UML 表示可能如下所示：



在圖形 SQL 中，實體Author和Book在結構描述中代表兩種不同的物件類型：

```
type Author {
}

type Book {
}
```

Author包含authorName和Books，而Book包含bookName和Authors。這些可以表示為類型範圍內的字段：

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

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

正如你所看到的，類型表示非常接近圖表。但是，這些方法是變得有點棘手的地方。這些將作為欄位放置在幾種特殊物件類型之一中。他們的特殊對象分類取決於他們的行為。GraphQL 包含三種基本的特殊物件類型：查詢、突變和訂閱。如需詳細資訊，請參閱特殊物件。

因為getAuthor和getBook都在請求數據，它們將被放置在Query特殊物件類型：

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

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

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

這些作業會連結至查詢，查詢本身會連結至資料架構。添加模式根將定義特殊對象類型（Query在這種情況下）作為您的入口點之一。這可以使用schema關鍵字:

```
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
}
```

看看最後兩種方法，addAuthor和addBook正在將數據添加到您的數據庫中，因此它們將在Mutation特殊物件類型。然而，從类型頁面中，我們也知道直接引用 Objects 的輸入是不允許的，因為它們是嚴格的輸出類型。在這種情況下，我們不能使用Author或者Book，所以我們需要使用相同的字段進行輸入類型。在這個例子中，我們添加了AuthorInput和BookInput，這兩者都接受其各自類型的相同字段。然後，我們使用輸入作為參數來創建突變：

```
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
}
```

讓我們回顧一下我們剛剛做了什麼：

1. 我們建立了一個結構描述Book和Author類型來代表我們的實體。
2. 我們添加了包含我們實體特徵的字段。
3. 我們添加了一個查詢以從數據庫中檢索此信息。
4. 我們添加了一個突變來操作數據庫中的數據。
5. 我們添加了輸入類型來替換突變中的對象參數，以符合 GraphQL 的規則。
6. 我們將查詢和變異新增至根結構描述，以便 GraphQL 實作瞭解根類型的位置。

正如您所看到的，創建模式的過程通常需要數據建模（尤其是數據庫建模）中的許多概念。您可以將結構描述視為符合來源資料的形狀。它也作為解析器將實現的模型。在接下來的章節中，您將學習如何使用各種模式AWS支持的工具和服務。

> **Note**
>
> 以下各節中的範例並不意味著在真實的應用程式中執行。它們只在那裡展示命令，以便您可以構建自己的應用程序。

建立綱要

您的模式將位於名為的文件中`schema.graphql`。AWSAppSync允許使用者使用各種方法為其 GraphQL API 建立新結構描述。在這個例子中，我們將創建一個空白 API 以及一個空白的模式。

Console

1. 登入 AWS Management Console 並開啟 AppSync主控台。
   a. 在儀表板上，選擇 Create API (建立 API)。
   b. 下API 選項，選擇圖形 SQL API,從頭開始設計，然後下一步。
      i. 對於API 名稱，將預先填入的名稱變更為您的應用程式所需的名稱。
      ii. 對於聯絡資料，您可以輸入聯絡點以識別 API 的管理員。此為選用欄位。
      iii. 下私有 API 配置，您可以啟用私有 API 功能。私有 API 只能從已設定的 VPC 端點 (VPCE) 存取。如需詳細資訊，請參閱私有 API。

      我們不建議在此範例中啟用此功能。選擇下一步在查看您的輸入之後。
   c. 下建立圖形 SQL 類型，您可以選擇建立 DynamoDB 表作為資料來源使用，或略過此動作，稍後再執行。

在此範例中，選擇稍後建立圖形 SQL 資源。我們將在一個單獨的部分創建一個資源。

d. 檢查您的輸入，然後選擇建立 API。

2. 您將在特定 API 的儀表板中。您可以判斷，因為 API 的名稱將位於儀表板的頂部。如果不是這種情況，您可以選擇API在側邊欄，然後在「」中選擇您的 APIAPI 儀表板。

   - 在側邊欄在您的 API 名稱下，選擇綱要。

3. 在架構編輯器，您可以配置`schema.graphql`文件。它可以是空的，也可以填充從模型生成的類型。在右邊，你有解析器用於將解析器附加到模式字段的部分。我們將不會在本節中查看解析器。

CLI

> **Note**
>
> 使用 CLI 時，請確定您擁有在服務中存取和建立資源的正確權限。您可能想要設定最低權限適用於需要存取服務的非管理員使用者的原則。有關更多信息AWS AppSync策略，請參閱的身分識別與存取管理AWS AppSync。
> 此外，如果您尚未閱讀主控台版本，建議您先閱讀主控台版本。

1. 如果你還沒有這樣做，安裝該AWSCLI，然後添加您的配置。
2. 透過執行`create-graphql-api`指令。

   您需要為此特定命令輸入兩個參數：

   1. 該`name`你的 API 的。
   2. 該`authentication-type`，或用於存取 API 的登入資料類型 (IAM、OIDC 等)。

   > **Note**
   >
   > 其他參數，例如`Region`必須設定，但通常會預設為 CLI 組態值。

   範例命令可能如下所示：

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

輸出將在 CLI 中返回。範例如下：

```
{
    "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**
>
> 這是一個可選命令，它接受現有的模式並將其上傳到AWS AppSync服務使用一個基 64 塊。為了這個例子，我們不會使用這個命令。

執行 start-schema-creation 命令。

您需要為此特定命令輸入兩個參數：

1. 您的`api-id`從上一步。
2. 該模式`definition`是一個基於 64 編碼的二進制 Blob。

範例命令可能如下所示：

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

一個輸出將被返回：

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

此命令在處理後不會返回最終輸出。您必須使用單獨的命令，get-schema-creation-status，以查看結果。請注意，這兩個命令是異步的，因此即使模式仍在創建，您也可以檢查輸出狀態。

CDK

> **Tip**
>
> 在您使用 CDK 之前，我們建議您先查看 CDK 的官方文件隨著AWS AppSync的CDK 參考。
> 下面列出的步驟將僅顯示用於添加特定資源的代碼片段的一般示例。這是不意味著成為您的生產代碼中的工作解決方案。我們還假設您已經有一個可用的應用程序。

1. CDK 的起點有點不同。理想情況下，您schema.graphql檔案應該已經建立。您只需要創建一個新文件.graphql檔案副檔名。這可以是一個空文件。
2. 通常，您可能必須將 import 指令添加到您正在使用的服務中。例如，它可以遵循以下形式：

```
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'
```

若要新增 GraphQL API，您的堆疊檔案必須匯入AWS AppSync服務：

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

> **Note**
>
> 這意味著我們正在導入整個服務appsync關鍵字。要在您的應用中使用此功能，您的AWS AppSync建構將使用格式appsync.construct_name。例如，如果我們想製作

一個 GraphQL API，我們會說new appsync.GraphqlApi(args_go_here)。下面的步驟描述了這一點。

3. 最基本的圖形 SQL API 將包括一個name應用程式介面和schema路徑。

```
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**
>
> 讓我們回顧一下此程式碼片段的作用。在範圍內api中，我們正在通過調用創建一個新的 GraphQL APIappsync.GraphqlApi(scope: Construct, id: string, props: GraphqlApiProps)。範圍是this，它指的是當前對象。該識別碼是*API_ID*，這將是您的圖形 SQL API 的資源名稱AWS CloudFormation當它被創建。該GraphqlApiProps包含name您的圖形 SQL 應用程式介面和schema。該schema將生成一個模式（SchemaFile.fromAsset）通過搜索絕對路徑（__dirname) 為.graphql檔案 (*####. ##*). 在實際情況下，您的架構文件可能位於 CDK 應用程序中。
>
> 若要使用對 GraphQL API 所做的變更，您必須重新部署應用程式。

將類型新增至綱要

現在您已新增結構描述，您可以開始新增輸入和輸出類型。請注意，這裡的類型不應該在實際代碼中使用; 它們只是幫助您理解過程的示例。

首先，我們將創建一個對象類型。在實際程式碼中，您不必從這些類型開始。只要遵循 GraphQL 的規則和語法，就可以隨時創建任何類型。

> **Note**
>
> 接下來的幾節將使用架構編輯器，所以保持打開。

Console

- 您可以使用建立物件類型type關鍵字以及類型的名稱：

```
type Type_Name_Goes_Here {}
```

在類型的範圍內，您可以添加代表對象特徵的字段：

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

範例如下：

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

> **Note**
>
> 在此步驟中，我們新增了具有必要項目的泛型物件類型id欄位儲存為ID，一個title欄位儲存為String，以及date欄位儲存為AWSDateTime。若要查看類型和欄位的清單及其功能，請參閱綱要。要查看標量列表以及它們的作用，請參閱類型參考。

CLI

> **Note**
>
> 如果您尚未閱讀主控台版本，建議您先閱讀主控台版本。

- 您可以執行create-type指令。

  您需要為此特定命令輸入一些參數：

1. 該api-id你的 API 的。
2. 該definition，或您類型的內容。在控制台示例中，這是：

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

3. 該format你的輸入。在這個例子中，我們使用SDL。

範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
```

輸出將在 CLI 中返回。範例如下：

```
{
    "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**
>
> 在此步驟中，我們新增了具有必要項目的泛型物件類型id欄位儲存為ID，一個title欄位儲存為String，以及date欄位儲存為AWSDateTime。若要查看類型和欄位的清單及其功能，請參閱綱要。要查看標量列表及其作用，請參閱類型參考。另外，您可能已經意識到輸入定義直接適用於較小的類型，但對於添加較大或多個類型是不可行的。您可以選擇添加所有內容.graphql文件，然後將其作為輸入傳遞。

CDK

> **Tip**
>
> 在您使用 CDK 之前，我們建議您先查看 CDK 的官方文件隨著AWS AppSync的CDK 參考。
> 下面列出的步驟將僅顯示用於添加特定資源的代碼片段的一般示例。這是不意味著成為您的生產代碼中的工作解決方案。我們還假設您已經有一個可用的應用程序。

要添加類型，您需要將其添加到您的.graphql文件。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樣直接將類型添加到模式中。

> **Note**
>
> 若要使用對 GraphQL API 所做的變更，您必須重新部署應用程式。

該物件類型具有下列欄位標量類型例如字符串和整數。AWSAppSync還允許您使用增強的標量類型AWSDateTime除了基本的 GraphQL 標量。此外，任何以驚嘆號結尾的欄位都是必需的。

該ID特別是標量類型是一個唯一的標識符，可以是String或者Int。您可以在解析器代碼中控制它們以進行自動分配。

特殊物件類型之間存在相似之處，例如Query和像上面的例子一樣的「常規」對像類型，因為它們都使用type關鍵字和被認為是對象。但是，對於特殊對象類型（Query,Mutation，以及Subscription），它們的行為有很大的不同，因為它們被公開為您的 API 的入口點。他們也更多關於塑造操作而不是數據。如需詳細資訊，請參閱查詢和突變類型。

關於特殊物件類型的主題，下一個步驟可能是新增一個或多個物件類型，以對造型的資料執行作業。在實際案例中，每個 GraphQL 結構描述至少必須具有用於請求資料的根查詢類型。您可以將查詢視為 GraphQL 伺服器的其中一個進入點 (或端點)。讓我們添加一個查詢作為一個例子。

Console

- 要創建查詢，您可以像任何其他類型一樣將其添加到模式文件中。一個查詢將需要Query類型和根目錄中的條目，如下所示：

```
schema {
  query: Name_of_Query
}

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

請注意#### (__)在生產環境中將簡單地調用Query在大多數情況下。我們建議將其保持在此值。在查詢類型中，您可以添加字段。每個字段將在請求中執行操作。因此，這些字段中的大多數（如果不是全部）都將附加到解析器。但是，在本節中，我們不關心這一點。關於現場操作的格式，它可能看起來像這樣：

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

範例如下：

```
schema {
  query: Query
}

type Query {
  getObj: [Obj_Type_1]
}

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

> **Note**
>
> 在此步驟中，我們新增了Query鍵入並在我們的定義schema根。我們的Query類型定義的getObj返回列表的字段Obj_Type_1物件。請注意Obj_Type_1是上一個步驟的物件。在生產代碼中，您的現場操作通常會處理由像這樣的對象形成的數據Obj_Type_1。此外，像這樣的字段getObj通常會有一個解析器來執行業務邏輯。這將在不同的部分中覆蓋。
> 作為額外的注意事項,AWS AppSync在匯出期間會自動新增結構描述根目錄，因此從技術上講，您不必直接將其新增至結構描述。我們的服務將自動處理重複的架構。我們在這裡新增它做為最佳作法。

CLI

> **Note**
>
> 如果您尚未閱讀主控台版本，建議您先閱讀主控台版本。

1. 創建一個schema帶有根query定義 (透過執行)create-type指令。

   您需要為此特定命令輸入一些參數：

   1. 該api-id你的 API 的。
   2. 該definition，或您類型的內容。在控制台示例中，這是：

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

   3. 該format你的輸入。在這個例子中，我們使用SDL。

   範例命令可能如下所示：

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

輸出將在 CLI 中返回。範例如下：

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

> **Note**
>
> 請注意，如果您沒有在中輸入正確的內容`create-type`指令時，您可以執行[update-type](#)指令。在此範例中，我們將暫時變更結構描述根目錄，以包含`subscription`定義。
> 您需要為此特定命令輸入一些參數：
>
> 1. 該`api-id`你的 API 的。
> 2. 該`type-name`你的類型。在控制台示例中，這是`schema`。
> 3. 該`definition`，或您類型的內容。在控制台示例中，這是：
>
>    ```
>    schema {
>      query: Query
>    }
>    ```
>
>   添加一個後的模式`subscription`看起來像這樣：
>
>    ```
>    schema {
>      query: Query
>      subscription: Subscription
>    }
>    ```
> 4. 該`format`你的輸入。在這個例子中，我們使用`SDL`。
>
> 範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
```

輸出將在 CLI 中返回。範例如下：

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

在此範例中，新增預先格式化的檔案仍可運作。

2. 創建一個Query通過運行鍵入create-type指令。

您需要為此特定命令輸入一些參數：

1. 該api-id你的 API 的。
2. 該definition，或您類型的內容。在控制台示例中，這是：

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

3. 該format你的輸入。在這個例子中，我們使用SDL。

範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
```

輸出將在 CLI 中返回。範例如下：

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

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

> **Note**
>
> 在此步驟中，我們新增了Query鍵入並在您的schema根。我們的Query類型定義的getObj返回列表的字段Obj_Type_1物件。
> 在schema根代碼query: Query，該query:部分表示查詢已在您的模式中定義，而Querypart 表示實際的特殊對象名稱。

CDK

> **Tip**
>
> 在您使用 CDK 之前，我們建議您先查看 CDK 的官方文件隨著AWS AppSync的CDK 參考。
> 下面列出的步驟將僅顯示用於添加特定資源的代碼片段的一般示例。這是不意味著成為您的生產代碼中的工作解決方案。我們還假設您已經有一個可用的應用程序。

您需要將查詢和結構描述根目錄添加到.graphql文件。我們的例子看起來像下面的例子，但是您需要將其替換為實際的模式代碼：

```
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樣直接將類型添加到模式中。

> **Note**
>
> 更新架構根目錄是選擇性的。我們將其添加到此示例中作為最佳實踐。
> 若要使用對 GraphQL API 所做的變更，您必須重新部署應用程式。

您現在已經看到了一個創建對象和特殊對象（查詢）的示例。您還看到了如何將這些相互連接以描述數據和操作。您可以擁有僅包含資料描述和一或多個查詢的結構描述。但是，我們想添加另一個操作來將數據添加到數據源。我們將添加另一種稱為的特殊對象類型Mutation修改數據。

Console

- 突變將被調用Mutation。喜歡Query，內部的現場操作Mutation將描述一個操作，並將附加到解析器。另外，請注意，我們需要在schemaroot，因為它是一個特殊的對象類型。這是一個突變的例子：

  ```
  schema {
    mutation: Name_of_Mutation
  }

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

  典型的突變將像查詢一樣在根目錄中列出。突變是使用type關鍵字與名稱一起。*#### (__F)*通常會被稱為Mutation，所以我們建議保持這種方式。每個欄位也會執行一項作業。關於現場操作的格式，它可能看起來像這樣：

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

  範例如下：

  ```
  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

在此步驟中，我們新增了Mutation類型與addObj欄位。讓我們總結一下這個領域的作用：

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

addObj正在使用Obj_Type_1要執行作業的物件。由於字段，這很明顯，但語法證明了這一點： Obj_Type_1返回類型。裡面addObj，它正在接受id,title，以及date來自的欄位Obj_Type_1對象作為參數。正如你可能看到的，它看起來很像一個方法聲明。但是，我們還沒有描述我們方法的行為。如前所述，模式只能在那裡定義數據和操作將是什麼，而不是它們的操作方式。實現實際的業務邏輯將在稍後我們創建我們的第一個解析器時。

完成模式後，可以選擇將其導出為schema.graphql文件。在架構編輯器，你可以選擇匯出綱要以支援的格式下載檔案。

作為額外的注意事項,AWS AppSync在匯出期間會自動新增結構描述根目錄，因此從技術上講，您不必直接將其新增至結構描述。我們的服務將自動處理重複的架構。我們在這裡新增它做為最佳作法。

CLI

> **Note**
>
> 如果您尚未閱讀主控台版本，建議您先閱讀主控台版本。

1. 執行以更新根綱要update-type指令。

   您需要為此特定命令輸入一些參數：

   1. 該api-id你的 API 的。
   2. 該type-name你的類型。在控制台示例中，這是schema。
   3. 該definition，或您類型的內容。在控制台示例中，這是：

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

   4. 該format你的輸入。在這個例子中，我們使用SDL。

   範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
   ```

   輸出將在 CLI 中返回。範例如下：

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

2. 創建一個Mutation通過運行鍵入create-type指令。

您需要為此特定命令輸入一些參數：

1. 該`api-id`你的 API 的。
2. 該`definition`，或您類型的內容。在控制台示例中，這是

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

3. 該`format`你的輸入。在這個例子中，我們使用SDL。

範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
```

輸出將在 CLI 中返回。範例如下：

```
{
    "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**
>
> 在您使用 CDK 之前，我們建議您先查看 CDK 的官方文件隨著AWS AppSync的CDK 參考。

下面列出的步驟將僅顯示用於添加特定資源的代碼片段的一般示例。這是不意味著成為您的生產代碼中的工作解決方案。我們還假設您已經有一個可用的應用程序。

您需要將查詢和結構描述根目錄添加到.graphql文件。我們的例子看起來像下面的例子，但是您需要將其替換為實際的模式代碼：

```
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

更新架構根目錄是選擇性的。我們將其添加到此示例中作為最佳實踐。
若要使用對 GraphQL API 所做的變更，您必須重新部署應用程式。

可選考量事項-使用枚舉作為狀態

在這一點上，你知道如何製作一個基本的模式。但是，您可以添加許多內容來增加模式的功能。在應用程序中找到的一個常見的事情是使用枚舉作為狀態。您可以使用枚舉來強制調用時從一組值中選擇特定值。這對於您知道在長時間內不會發生巨大變化的事情來說是有益的。假設地說，我們可以添加一個枚舉來返回響應中的狀態代碼或字符串。

舉個例子，假設我們正在製作一個社交媒體應用程序，該應用程序將用戶的帖子數據存儲在後端。我們的架構包含Post代表單個帖子數據的類型：

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

我們的Post將包含一個唯一的id, 文章title,date發布，以及一個叫做的枚舉PostStatus表示該帖子的狀態，因為它是由應用程序處理。對於我們的操作，我們將有一個查詢返回所有發布數據：

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

我們還將有一個突變，將帖子添加到數據源中：

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

查看我們的架構，PostStatus枚舉可能有幾種狀態。我們可能想要稱為三個基本狀態success（後處理成功），pending（正在處理後），以及error（無法處理後）。要添加枚舉，我們可以這樣做：

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

完整的結構描述可能如下所示：

```
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
}
```

如果使用者新增Post在應用程式中，addPost操作將被調用來處理該數據。作為解析器附加到addPost處理數據，它將不斷更新poststatus與操作的狀態。查詢時，Post將包含數據的最終狀態。請記住，我們只描述了我們希望數據在模式中的工作方式。我們假設了很多關於我們解析器的實現，它將實現用於處理數據以滿足請求的實際業務邏輯。

選擇性考量-訂閱

中的訂閱AWS AppSync被調用為對突變的響應。您可利用結構描述之中的 Subscription 類型及 @aws_subscribe() 指令進行設定，表示哪些變動用於叫用一項以上的訂閱。如需有關設定訂閱的詳細資訊，請參閱即時資料。

可選的考慮因素-關係和分頁

假設你有一百萬Posts儲存在 DynamoDB 資料表中，而且您想要傳回其中一些資料。但是，上面給出的示例查詢僅返回所有帖子。您不希望每次提出請求時都獲取所有這些內容。相反，你會想分頁通過他們。請對結構描述進行下列變更：

- 在getPosts字段中，添加兩個輸入參數：nextToken（迭代器）和limit（迭代限制）。
- 添加一個新的PostIterator類型包含Posts（檢索列表Post物件) 和nextToken（迭代器）字段。
- 變更getPosts這樣它就會返回PostIterator而不是一個列表Post物件。

```
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
}
```

該PostIteratortype 允許您返回列表的一部分Post物件和nextToken為了獲得下一部分。裡面PostIterator，有一個列表Post項目 ([Post])，返回一個分頁令牌 (nextToken). 在AWS AppSync，這會透過解析器連線至 Amazon DynamoDB，並自動產生為加密權杖。這會將 limit 引數的值轉換成 maxResults 參數，並將 nextToken 引數轉換成 exclusiveStartKey 參數。對於示例和內置模板示例AWS AppSync主控台，請參閱解析器參考 (JavaScript)。

步驟 2：附加資料來源

資料來源是您的資源AWS圖形 SQL API 可以與之互動的帳戶。AWSAppSync支持多種數據源，例如AWS Lambda, 亞馬遜動態資料庫, 關聯式資料庫 (亞馬遜極光無伺服器), 亞馬遜OpenSearch服務和 HTTP 端點。一個AWS AppSyncAPI 可設定為與多個資料來源互動，讓您能夠在單一位置彙總資

料。AWSAppSync可以使用現有的AWS您帳戶中的資源，或代表您從結構描述定義佈建 DynamoDB 表格。

以下部分將向您展示如何將數據源附加到 GraphQL API。

資料來源的類型

現在，您已經在AWS AppSync控制台中，您可以將數據源附加到它。當您一開始建立 API 時，可以選擇在建立預先定義的結構描述期間佈建 Amazon DynamoDB 表格。但是，我們將不會在本節中介紹該選項。你可以看到這個例子啟動結構描述部分。

相反，我們將查看所有數據源AWS AppSync支持。為您的應用選擇正確的解決方案有很多因素。以下各節將為每個資料來源提供一些額外的上下文。如需有關資料來源的一般資訊，請參閱資料來源。

Amazon DynamoDB

亞馬遜是其中之一AWS'可擴展應用的主要存儲解決方案。動態支援的核心元件是表，這只是一個數據的集合。您通常會根據實體創建表Book或者Author。表格項目資訊儲存為項目，是每個項目唯一的欄位群組。完整項目代表資料庫中的列/記錄。例如，對於一個項目Book項目可能包括title和author連同它們的價值觀。各個字段title和author被稱為屬性，類似於關聯式資料庫中的欄值。

您可以猜到，表將用於存儲應用程序中的數據。AWS AppSync可讓您將 DynamoDB 資料表連結至您的圖形 SQL API 來操作資料。拿著這個使用案例從前端 Web 和移動博客。該應用程序允許用戶註冊社交媒體應用程序。使用者可以加入群組並上傳廣播給訂閱群組的其他使用者的貼文。其應用程式會將使用者、貼文和使用者群組資訊儲存在 DynamoDB 中。圖形 SQL 應用程式介面 (由管理AWS AppSync) 與動態資料表的介面。當使用者在將反映在前端的系統中進行變更時，GraphQL API 會擷取這些變更，並即時將其廣播給其他使用者。

AWS Lambda

Lambda 是一項事件驅動型服務，可自動建置必要的資源，以便執行程式碼作為回應事件。拉姆達使用功能，這是包含用於執行資源的程式碼、相依性和組態的群組陳述式。函數在偵測到時自動執行觸發器，一組調用您的函數的活動。觸發器可以是類似於進行 API 調用的應用程序，AWS您帳戶中的服務旋轉資源等 觸發時，函數將處理事件，這是包含要修改之資料的 JSON 文件。

Lambda 非常適合執行程式碼，而不必佈建資源來執行程式碼。拿著這個使用案例從前端 Web 和移動博客。此使用案例有點類似於 DynamoDB 一節中展示的使用案例。在此應用程序中，GraphQL API 負責定義諸如添加帖子（突變）和獲取數據（查詢）之類的操作。為了實現其操作的功能（例如，`getPost ( id: String ! ) : Post`,`getPostsByAuthor ( author: String ! ) : [ Post ]`），他們使用 Lambda 函數來處理輸入請求。下選項二：AWS AppSync使用拉姆達解析器，

他們使用AWS AppSync維護其結構描述並將 Lambda 資料來源連結至其中一項作業的服務。呼叫作業時，Lambda 會與 Amazon RDS 代理進行介面，以便在資料庫上執行商業邏輯。

Amazon RDS

Amazon RDS 可讓您快速建置和設定關聯式資料庫。在亞馬遜 RDS，你會創建一個通用資料庫實例這將作為雲中隔離的數據庫環境。在這種情況下，您將使用資料庫引擎這是實際的數據庫管理系統軟件（PostgreSQL，MySQL 等）。該服務通過使用提供可擴展性來卸載大部分後端工作AWS'基礎架構、修補和加密等安全服務，以及降低部署的管理成本。

採取相同的使用案例從拉姆達部分。下選項三：AWS AppSync與亞馬遜 RDS 解析器，提供的另一個選項是將 GraphQL API 連接到AWS AppSync直接到亞馬遜 RDS。使用資料 API，它們會將資料庫與 GraphQL API 建立關聯。解析器附加到字段（通常是查詢，突變或訂閱），並實現訪問數據庫所需的 SQL 語句。當調用字段的請求是由客戶端提出，解析器執行語句並返回響應。

Amazon EventBridge

在EventBridge，您將創建活動巴士，它們是從您附加的服務或應用程式接收事件的管道 (事件來源) 並根據一組規則處理它們。一個事件是執行環境中的某些狀態變化，而 a規則是一組用於事件的過濾器。規則遵循事件模式，或事件狀態變更的中繼資料 (ID、地區、帳號、ARN 等)。當事件與事件模式匹配時，EventBridge將通過管道將事件發送到目的地服務（目標) 並觸發規則中指定的處理行動。

EventBridge對於將狀態更改操作路由到其他服務有好處。拿著這個使用案例從前端 Web 和移動博客。這個例子描述了一個電子商務解決方案，該解決方案具有多個團隊維護 其中一項服務會在前端的每個交貨步驟 (下訂單、進行中、出貨、交付等)，為客戶提供訂單更新。但是，管理此服務的前端團隊無法直接存取訂購系統資料，因為這些資料是由個別的後端團隊所維護的。後端團隊的訂購系統也被描述為黑匣子，因此很難收集有關他們構建數據方式的信息。但是，後端團隊確實設置了一個系統，該系統通過管理的事件總線發布訂單數據EventBridge。為了訪問來自事件總線的數據並將其路由到前端，前端團隊創建了一個指向其 GraphQL API 的新目標AWS AppSync。他們也建立了一個規則，只傳送與訂單更新相關的資料。進行更新時，事件匯流排中的資料會傳送至 GraphQL API。API 中的結構描述會處理資料，然後將其傳遞至前端。

無資料來源

如果您不打算使用資料來源，可以將其設定為none。一個none資料來源雖然仍明確分類為資料來源，但不是儲存媒體。通常，解析器會在某個時候調用一個或多個數據源來處理請求。但是，在某些情況下，您可能不需要操作資料來源。將資料來源設定為none將運行請求，跳過數據調用步驟，然後運行響應。

採取相同的使用案例從EventBridge部分。在結構描述中，突變會處理狀態更新，然後將其傳送給訂閱者。回顧解析器如何工作，通常至少有一個數據源調用。不過，此案例中的資料已由事件匯流排自動傳

送。這意味著不需要突變來執行數據源調用; 訂單狀態可以簡單地在本地處理。突變設置為none，它充當沒有資料來源叫用的傳遞值。然後，結構描述會填入資料，資料會傳送給訂閱者。

OpenSearch

亞馬遜OpenSearch服務是一套工具，用於實現全文搜索，數據可視化和日誌記錄。您可以使用此服務查詢已上傳的結構化資料。

在此服務中，您將建立的執行個體OpenSearch。這些被稱為節點。在節點中，您將至少添加一個指數。在概念上，索引有點像關係數據庫中的表。(但是,OpenSearch不符合 ACID 標準，因此不應該以這種方式使用）。您將在索引中填入您上傳到OpenSearch服務。上傳數據後，它將在索引中存在的一個或多個分片中進行索引。一個碎片就像索引的一個分區，其中包含一些數據，並且可以與其他碎片分開查詢。上傳後，您的數據將結構為 JSON 文件，稱為文件。然後，您可以查詢文檔中的數據節點。

HTTP 端點

您可以使用 HTTP 端點做為資料來源。AWSAppSync可以使用參數和有效負載等相關信息將請求發送到端點。HTTP 響應將暴露給解析器，該解析器將在完成其操作後返回最終響應。

新增資料來源

如果您已建立資料來源，則可以將其連結至AWS AppSync服務，更具體地說，API。

Console

1. 登入 AWS Management Console 並開啟 AppSync主控台。
   a. 在中選擇您的 API儀表板。
   b. 在側邊欄，選擇資料來源。
2. 選擇 Create data source (建立資料來源)。
   a. 為您的資料來源命名。你也可以給它一個描述，但這是可選的。
   b. 選擇您的資料來源類型。
   c. 對於 DynamoDB，您必須選擇您的區域，然後選擇區域中的表格。您可以選擇設定新的泛用表格角色或匯入表格的現有角色，來指定與表格互動規則。您可以啟用版本化，當多個用戶端同時嘗試更新資料時，可自動為每個要求建立資料版本。版本控制用於保留和維護數據的多種變體，用於衝突檢測和解決目的。您還可以啟用自動模式生成，這將需要您的數據源並生成一些 CRUD，`List`，以及`Query`在模式中訪問它所需的操作。

對於OpenSearch，您必須選擇您的區域，然後選擇區域中的域（集群）。您可以選擇設定新的泛用表格角色或匯入表格的現有角色，來指定與網域的互動規則。

對於 Lambda，您必須選擇您的區域，然後選擇區域中 Lambda 函數的 ARN。您可以選擇建立新的泛用資料表角色或匯入表格的現有角色，來指定與 Lambda 函數的互動規則。

對於 HTTP，您必須輸入您的 HTTP 端點。

對於EventBridge，您必須選擇您的地區，然後選擇該地區的活動巴士。您可以選擇建立新的泛用表格角色或匯入表格的現有角色，來指定與事件匯流排的互動規則。

對於 RDS，您必須選擇您的區域，然後選擇秘密存儲區（用戶名和密碼），數據庫名稱和模式。

如果是 none，您將新增沒有實際資料來源的資料來源。這是用於在本地處理解析器，而不是通過實際的數據源處理。

> **Note**
>
> 如果您要匯入現有角色，則他們需要信任原則。如需詳細資訊，請參閱IAM 信任政策。

3. 選擇 建立 。

> **Note**
>
> 或者，如果您要建立 DynamoDB 資料來源，您可以移至綱要在控制台頁面中，選擇建立資源在頁面頂部，然後填寫預先定義的模型以轉換為表格。在此選項中，您將填寫或匯入基礎類型、設定包含分割區索引鍵的基本資料表資料，以及檢閱結構描述變更。

CLI

- 執行以建立資料來源`create-data-source`指令。

  您需要為此特定命令輸入一些參數：

  1. 該`api-id`你的 API。
  2. 該`name`你的桌子。

3. 該type的資料來源。根據您選擇的資料來源類型，您可能需要輸入service-role-arn和一個-config標籤。

範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**
>
> 在您使用 CDK 之前，我們建議您先查看 CDK 的官方文件隨著AWS AppSync的CDK 參考資料。
> 下面列出的步驟將僅顯示用於添加特定資源的代碼片段的一般示例。這是不意味著成為生產代碼中的工作解決方案。我們還假設您已經有一個可用的應用程序。

要添加特定的數據源，您需要將構造添加到堆棧文件中。您可以在此處找到資料來源類型的清單：

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

1. 通常，您可能必須將 import 指令添加到您正在使用的服務中。例如，它可以遵循以下形式：

```
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，以下是如何導入AWS AppSync和動態支援服務：

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

2. 某些服務 (例如 RDS) 需要在堆疊檔案中進行一些額外的設定，才能建立資料來源 (例如 VPC 建立、角色和存取認證)。如需詳細資訊，請參閱相關 CDK 頁面中的範例。
3. 對於大多數資料來源，尤其是AWS服務，您將在堆棧文件中創建數據源的新實例。通常情況下，這看起來像下面這樣：

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

例如，以下是亞馬遜動態資料表的範例：

```
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**
>
> 大多數資料來源至少會有一個必要 prop (將表示)無一個?符號）。請參閱 CDK 文件以瞭解需要哪些道具。

4. 接下來，您需要將資料來源連結至 GraphQL API。建議的方法是在為管線解析器創建函數時添加它。例如，下面的程式碼片段是掃描 DynamoDB 表中所有元素的函數：

```
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道具，你可以調用圖形 SQL API（add_api）並使用其中一種內置方法（addDynamoDbDataSource) 以建立資料表與 GraphQL API 之間的關聯性。參數是此鏈接的名稱，將存在於AWS AppSync控制台（data_source_name_in_console在這個例子中）和表方法（add_ddb_table). 當您開始製作解析器時，下一節將顯示有關此主題的更多信息。

有用於連結資料來源的替代方法。你可以在技術上添加api到表格函數中的 prop 列表。例如，這是步驟 3 中的代碼片段，但帶有api包含一個圖形 SQL API 的道具：

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

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

 ...

  api: add_api
});
```

或者，您可以呼叫GraphqlApi單獨構造：

```
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);
```

我們建議您只在函數的 prop 中建立關聯。否則，您必須在中手動將解析器函數鏈接到數據源AWS AppSync控制台（如果你想繼續使用控制台值`data_source_name_in_console`）或以另一個名稱在函數中創建一個單獨的關聯`data_source_name_in_console_2`。這是由於 prop 處理資訊的方式有所限制。

> **Note**
>
> 您必須重新部署應用程式才能查看變更。

IAM 信任政策

如果您為資料來源使用現有的 IAM 角色，則需要授予該角色適當的許可，才能在您的資料來源上執行操作AWS資源，例如`PutItem`在亞馬遜動態 B 表上。您也需要修改該角色的信任原則，以允許AWS AppSync以將其用於資源訪問，如以下示例策略所示：

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

您也可以在信任原則中新增條件，以視需要限制對資料來源的存取。目前，`SourceArn`和`SourceAccount`金鑰可以在這些條件下使用。例如，下列政策會限制對帳戶對資料來源的存取`123456789012`:

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

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

或者，您可以將資料來源的存取限制為特定 API，例如`abcdefghijklmnopq`，使用下列原則：

```
{
  "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"
        }
      }
    }
  ]
}
```

您可以限制對所有人的存取AWS AppSync來自特定區域的 API，例如`us-east-1`，使用下列原則：

```
{
  "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/*"
        }
      }
    }
  ]
}
```

在下一節中（配置解析器），我們將添加解析器業務邏輯並將其附加到模式中的字段，以處理數據源中的數據。

如需有關角色原則組態的詳細資訊，請參閱修改角色在IAM 使用者指南。

有關跨帳戶訪問的更多信息AWS Lambda解析器AWS AppSync，請參閱建立跨帳戶AWS Lambda解析器AWS AppSync。

步驟 3：配置解析器

在前面的章節中，您學會瞭如何建立 GraphQL 結構描述和資料來源，然後在 AWS AppSync 服務中將它們連結在一起。在您的模式中，您可能已經在查詢和突變中建立了一個或多個字段（操作）。雖然結構描述了操作將從數據源請求的數據類型，但它從未實現這些操作如何圍繞數據的行為。

一個操作的行為總是在解析器中實現，該解析器將鏈接到執行操作的字段。有關解析器一般如何工作的詳細資訊，請參閱解析器頁面。

在中 AWS AppSync，您的解析程式與執行階段相關聯，也就是解析程式執行的環境。運行時決定您的解析器將用來編寫的語言。目前有兩個支援的執行階段：APPSYNC_JS (JavaScript) 和阿帕奇速度範本語言 (VTL)。

在實施解析器時，它們遵循一個通用的結構：

- 在步驟之前：當客戶端提出請求時，正在使用的模式字段（通常是您的查詢，突變，訂閱）的解析器將傳遞請求數據。解析器將開始使用 before step 處理程序處理請求數據，這允許在數據通過解析器移動之前執行一些預處理操作。
- 函數（S）：在步驟運行之前，請求被傳遞到函數列表。清單中的第一個函數會針對資料來源執行。函數是解析器代碼的子集，其中包含自己的請求和響應處理程序。請求處理程序將採取請求數據並對數據源執行操作。響應處理程序將數據源的響應傳遞回列表之前處理數據源的響應。如果有多個函

數，請求數據將被發送到要執行的列表中的下一個函數。列表中的函數將按照開發人員定義的順序進行連續執行。一旦所有的函數都被執行，最終的結果被傳遞給後一步。

- 之後步驟：之後的步驟是一個處理函數，允許您在將其傳遞給 GraphQL 響應之前對最終函數的響應執行一些最終操作。

此流程是管線解析器的範例。這兩個執行階段都支援管線解析器。但是，這是對管道解析器可以執行的操作的簡化解釋。另外，我們只描述了一種可能的解析器配置。如需有關支援解析器組態的詳細資訊，請參閱 APPSYNC_JS 的JavaScript 解析器概觀或 VTL 的解析器對應範本概觀。

正如你所看到的，解析器是模塊化的。為了使解析器的組件正常工作，它們必須能夠從其他組件對等到執行狀態。從 Resolvers 部分，您知道解析器中的每個組件都可以作為一組參數（`args`等`context`）傳遞有關執行狀態的重要信息。在中 AWS AppSync，這是由嚴格處理`context`。這是一個容器，用於有關正在解析的字段的信息。這可以包括傳遞的參數，結果，授權數據，標題數據等的所有內容。如需有關前後關聯的詳細資訊，請參閱 APPSYNC_JS 的解析器前後關聯物件參考或 VTL 的解析器對應範本前後關聯參考。

上下文不是您可以用來實現解析器的唯一工具。 AWS AppSync 支持用於價值生成，錯誤處理，解析，轉換等各種實用程序。您可以在此處查看 APPSYNC_JS 的實用程序列表，或者在此處查看 VTL 的實用程序。

在以下各節中，您將學習如何在 GraphQL API 中配置解析器。

主題

- 配置解析器（JavaScript)
- 設定解析器 (VTL)

配置解析器（JavaScript)

GraphQL 解析程式將類型結構描述中的欄位連接到資料來源。解析器是通過其請求被滿足的機制。

解析器AWS AppSync使用JavaScript將 GraphQL 運算式轉換為資料來源可以使用的格式。或者，可以將對應範本寫入阿帕奇速度模板語言（VTL）將 GraphQL 運算式轉換為資料來源可以使用的格式。

本節介紹如何使用配置解析器JavaScript。該解析器教程（JavaScript)部分提供有關如何使用實現解析器的深入教程JavaScript。該解析器參考 (JavaScript)部分提供了可與之搭配使用的公用程式作業的說明JavaScript解析器。

我們建議您在嘗試使用上述任何教學課程之前遵循本指南。

在本節中，我們將逐步介紹如何創建和配置查詢和突變的解析器。

> **Note**
>
> 本指南假設您已建立結構描述，且至少有一個查詢或變異。如果您正在尋找訂閱（即時資料），請參閱這個指南。

在本節中，我們將提供一些配置解析器的一般步驟以及使用以下結構描述的示例：

```
// 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]
}
```

建立基本查詢解析器

本節將向您展示如何製作基本的查詢解析器。

Console

1. 登入 AWS Management Console 並開啟 AppSync主控台。
   a. 在API 儀表板」下方，選擇您的圖形 SQL API。
   b. 在側邊欄，選擇綱要。

2. 輸入結構描述和資料來源的詳細資訊。請參閱設計您的架構和貼附資料來源區段可取得更多資訊。
3. 旁邊的綱要編輯器，有一個窗口叫解析器。此方塊包含類型和欄位的清單，如綱要窗口。您可以將解析器附加到欄位。您很可能會將解析器附加到現場操作中。在本節中，我們將看看簡單的查詢配置。在下查詢類型，選擇貼附在查詢欄位旁邊。
4. 在「」附加解析器頁面，下解析器類型，您可以在管線或單位解析器之間進行選擇。如需這些類型的詳細資訊，請參閱解析器。本指南將利用`pipeline resolvers`。

> **Tip**
>
> 建立管線解析器時，您的資料來源將附加至管線函數。函數是在您建立管線解析器本身之後建立的，這就是為什麼在此頁面中沒有選項可以設定它的原因。如果您使用的是單位解析器，則數據源將直接綁定到解析器，因此您可以在此頁面中進行設置。

對於解析器運行時，選擇APPSYNC_JS以啟用JavaScript執行階段。

5. 您可以啟用高速緩存對於這個 API。我們建議您立即關閉此功能。選擇 建立 。
6. 在「」編輯解析器頁面中，有一個代碼編輯器叫解析器代碼它允許您實現解析器處理程序和響應的邏輯（步驟之前和之後）。如需詳細資訊，請參閱JavaScript解析器概述。

> **Note**
>
> 在我們的例子中，我們只是將請求留空，響應集以返回最後一個數據源結果上下文:
>
> ```
> import {util} from '@aws-appsync/utils';
>
> export function request(ctx) {
>     return {};
> }
>
> export function response(ctx) {
>     return ctx.prev.result;
> }
> ```

在本節下面，有一個名為的表函数。函數允許您實現可以在多個解析器中重複使用的代碼。您可以將源代碼存儲為函數，而不是經常重寫或複製代碼，以便在需要時添加到解析器中。

函數構成了管道操作列表的大部分。在解析器中使用多個函數時，您可以設置函數的順序，它們將按順序運行。它們在請求函數運行後和響應函數開始之前執行。

若要新增函數，請在函数，選擇新增功能，然後建立新函數。或者，您可能會看到建立函數按鈕來選擇。

a. 選擇資料來源。這將是解析器作用的數據源。

> **Note**
>
> 在我們的例子中，我們附加了一個解析器`getPost`，它會擷取一個`Post`物件依據`id`。假設我們已經為此結構描述設定了 DynamoDB 資料表。它的分割區索引鍵設定為`id`並且是空的。

b. 輸入一個`Function name`。

c. 下函數代碼，您需要實現該函數的行為。這可能會令人困惑，但是每個函數都有自己的本地請求和響應處理程序。請求運行，然後進行數據源調用來處理請求，然後數據源響應由響應處理程序處理。結果會儲存在上下文物件。之後，列表中的下一個函數將運行或將被傳遞到後步響應處理程序（如果它是最後一個）。

> **Note**
>
> 在我們的例子中，我們將解析器附加到`getPost`，它得到一個列表`Post`來自資料來源的物件。我們的 request 函數將從我們的表中請求數據，該表將其響應傳遞給上下文（ctx），然後響應將返回上下文中的結果。AWS AppSync的力量在於它與其他人的相互聯繫AWS服務。因為我們使用的是 DynamoDB，所以我們有一個作業套件簡化這樣的事情。我們也有其他數據源類型的一些樣板示例。
> 我們的代碼將如下所示：
>
> ```
> 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;
}
```

在此步驟中，我們添加了兩個函數：

- `request`：要求處理常式會針對資料來源執行擷取作業。該參數包含上下文對象 (`ctx`) 或某些資料，可供執行特定作業的所有解析器使用。例如，它可能包含授權數據，正在解析的字段名稱等。返回語句執行Scan作業 (請參閱這裡例如）。因為我們正在使用 DynamoDB，所以我們可以使用該服務中的某些作業。掃描執行我們表中所有項目的基本獲取。此操作的結果存儲在上下文對象作為`result`容器被傳遞給響應處理程序之前。該`request`在管道中的響應之前運行。
- `response`：傳回輸出的回應處理常式`request`。該參數是更新的上下文對象，返回語句是`ctx.prev.result`。此時在指南中，您可能不熟悉此值。`ctx`指的是上下文對象。`prev`指的是管道中的以前的操作，這是我們的`request`。該`result`包含解析器在管線中移動時的結果。如果你把它們放在一起`ctx.prev.result`正在返回執行的最後一個操作的結果，這是請求處理程序。

d. 選擇創建在你完成之後。

7. 回到解析器屏幕上，在函數，選擇新增功能下拉菜單並將您的功能添加到您的功能列表中。
8. 選擇儲存以更新解析器。

CLI

若要新增您的函數

- 使用建立管線解析程式的函數`create-function`指令。

  您需要為此特定命令輸入一些參數：

  1. 該`api-id`你的 API 的。
  2. 該`name`中的功能AWS AppSync控制台。
  3. 該`data-source-name`，或函數將使用的資料來源名稱。它必須已經建立並連結至您在AWS AppSync服務。

4. 該runtime，或函數的環境和語言。對於JavaScript，名稱必須是APPSYNC_JS，以及執行階段，1.0.0。
5. 該code，或函數的請求和響應處理程序。雖然您可以手動輸入它，但將其添加到 .txt 文件（或類似格式）中，然後將其作為參數傳遞要容易得多。

> **Note**
>
> 我們的查詢代碼將位於作為參數傳入的文件中：
>
> ```
> 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;
> }
> ```

範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}
```

輸出將在 CLI 中返回。範例如下：

```
{
    "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**
>
> 確保您記錄了functionId某處，因為這將被用於將函數附加到解析器。

若要建立您的解析程式

- 建立管線函數Query透過執行create-resolver指令。

  您需要為此特定命令輸入一些參數：

  1. 該api-id你的 API 的。
  2. 該type-name，或結構描述中的特殊物件類型 (查詢、變異、訂閱)。
  3. 該field-name，或您要附加解析器的特殊物件類型內的欄位作業。
  4. 該kind，它指定單位或配管解析器。將此設定為PIPELINE以啟用管線功能。
  5. 該pipeline-config，或附加到解析器的函數。請確定您知道functionId你的函數值。上市順序事宜。
  6. 該runtime，這是APPSYNC_JS(JavaScript). 該runtimeVersion目前是1.0.0。
  7. 該code，其中包含之前和之後的步驟處理程序。

  > **Note**
  >
  > 我們的查詢代碼將位於作為參數傳入的文件中：
  >
  > ```
  > 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;
}
```

範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}
```

輸出將在 CLI 中返回。範例如下：

```
{
    "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**
>
> 在您使用 CDK 之前，我們建議您先查看 CDK 的官方文件隨著AWS AppSync的CDK 參考。
> 下面列出的步驟將僅顯示用於添加特定資源的代碼片段的一般示例。這是不意味著成為您的生產代碼中的工作解決方案。我們還假設您已經有一個可用的應用程序。

一個基本的應用程序將需要以下內容：

1. 服務匯入指令
2. 綱要程式碼
3. 資料來源產生器
4. 函數程式碼
5. 解析器代碼

從設計您的架構和貼附資料來源部分中，我們知道堆棧文件將包含形式的導入指令：

```
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**
>
> 在前面的章節中，我們只說明瞭如何導入AWS AppSync構造。在實際代碼中，您必須導入更多服務才能運行該應用程序。在我們的例子中，如果我們要創建一個非常簡單的 CDK 應用程序，我們至少會導入AWS AppSync與我們的資料來源一起提供服務，這是 DynamoDB 資料表。我們還需要導入一些額外的構造來部署應用程序：
>
> ```
> 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';
> ```
>
> 總結這些中的每一個：
>
> - `import * as cdk from 'aws-cdk-lib';`：這允許您定義 CDK 應用程序和構造，例如堆棧。它還包含了一些有用的實用功能，例如操作元數據我們的應用程序。如果您熟悉此 import 指令，但想知道為什麼此處沒有使用 cdk 核心庫，請參閱移民頁面。
> - `import * as appsync from 'aws-cdk-lib/aws-appsync';`：這將導入AWS AppSync服務。
> - `import * as dynamodb from 'aws-cdk-lib/aws-dynamodb';`：這將導入動態支援服務。
> - `import { Construct } from 'constructs';`：我們需要這個來定義根建構。

匯入的類型取決於您呼叫的服務。我們建議您查看 CDK 文檔以獲取示例。頁面頂部的模式將是 CDK 應用程序中的單獨文件`.graphql`文件。在堆棧文件中，我們可以使用以下形式將其與新的 GraphQL 關聯起來：

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

> **Note**
>
> 在範圍內`add_api`中，我們正在添加一個新的 GraphQL API 使用`new`關鍵字後跟`appsync.GraphqlApi(scope: Construct, id: string , props: GraphqlApiProps)`。我們的範圍是`this`，該 CFN 識別碼是`graphQL-example`，我們

的道具是`my-first-api`（控制台中的 API 名稱）和`schema.graphql`(結構描述檔案的絕對路徑)。

若要新增資料來源，您必須先將資料來源新增至堆疊。然後，您需要使用特定於源的方法將其與 GraphQL API 相關聯。當你使你的解析器功能時，關聯會發生。在此期間，讓我們使用建立 DynamoDB 資料表來使用範例`dynamodb.Table`:

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

> **Note**
>
> 如果我們要在我們的範例中使用這個，我們會新增一個新的 DynamoDB 表格，其中含有 CFN 識別碼`posts-table`和一個分區鍵`id (S)`。

接下來，我們需要在堆棧文件中實現我們的解析器。以下是掃描 DynamoDB 表中所有項目的簡單查詢範例：

```
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，我們創建了一個名為的函數add_func。這種創建順序可能看起來有點違反直覺，但是在製作解析器本身之前，您必須在管道解析器中創建函數。函數的形式如下：

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

我們的範圍是this，我們的 CFN 識別碼是func-get-posts，我們的道具包含了實際的功能細節。裡面的道具，我們包括：

- 該name的功能將存在於AWS AppSync控制台（get_posts_func_1).
- 我們之前創建的圖形 SQL API（add_api).
- 資料來源；這是我們將資料來源連結至 GraphQL API 值，然後將其附加至函數的點。我們採取我們創建的表（add_ddb_table）並將其附加到圖形 SQL API（add_api) 使用其中一個GraphqlApi方法（addDynamoDbDataSource). 識別碼值 (table-for-posts) 是中資料來源的名稱AWS AppSync控制台。如需來源特定方法的清單，請參閱下列頁面：
  - DynamoDbDataSource
  - EventBridgeDataSource
  - HttpDataSource
  - LambdaDataSource
  - NoneDataSource

    - OpenSearchDataSource
    - RdsDataSource
  - 該代碼包含我們函數的請求和響應處理程序，這是一個簡單的掃描和返回。
  - 執行階段會指定我們要使用 APPSYNC_JS 執行階段版本 1.0.0。請注意，這是目前唯一可用於 APPSYNC_JS 的版本。

  接下來，我們需要將函數附加到管道解析器。我們使用以下表單創建了我們的解析器：

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

  我們的範圍是`this`，我們的 CFN 識別碼是`pipeline-resolver-get-posts`，我們的道具包含了實際的功能細節。在道具裡面，我們包括：

  - 我們之前創建的圖形 SQL API（`add_api`）。
  - 特殊物件類型名稱；這是查詢作業，所以我們只是加入值`Query`。
  - 欄位名稱 (`getPost`) 是結構描述中欄位的名稱`Query`類型。
  - 該代碼包含您的前後處理程序。我們的例子只是返回函數執行其操作後的上下文中的任何結果。
  - 執行階段會指定我們要使用 APPSYNC_JS 執行階段版本 1.0.0。請注意，這是目前唯一可用於 APPSYNC_JS 的版本。
  - 管道配置包含對我們創建的函數的引用（`add_func`）。

為了總結這個例子中發生的事情，你看到了AWS AppSync實現請求和響應處理程序的函數。該函數負責與您的數據源進行交互。請求處理程序發送了`Scan`作業至AWS AppSync，指示它針對 DynamoDB 資料來源執行的作業。響應處理程序返回的項目列表（`ctx.result.items`）. 然後將項目清單對應至`Post`圖形 QL 自動類型。

創建基本的突變解析器

本節將向您展示如何製作基本的突變解析器。

Console

1. 登入 AWS Management Console 並開啟 AppSync主控台。

   a. 在API 儀表板」下方，選擇您的圖形 SQL API。

b. 在側邊欄，選擇綱要。

2. 在下解析器部分和突變類型，選擇貼附在你的領域旁邊。

> **Note**
>
> 在我們的例子中，我們附加了一個解析器`createPost`，這增加了一個`Post`反對我們的表。假設我們使用的是上一節中相同的 DynamoDB 表。它的分割區索引鍵設定為`id`並且是空的。

3. 在「」附加解析器頁面，下解析器類型，選擇`pipeline resolvers`。提醒您，您可以找到有關解析器的更多信息這裡。對於解析器運行時，選擇APPSYNC_JS以啟用JavaScript執行階段。

4. 您可以啟用高速緩存對於這個 API。我們建議您立即關閉此功能。選擇 建立 。

5. 選擇新增功能，然後選擇建立新函數。或者，您可能會看到建立函數按鈕來選擇。

a. 選擇 資料來源。這應該是您將通過突變操作其數據的源。

b. 輸入一個`Function name`。

c. 下函數代碼，您需要實現該函數的行為。這是一個突變，因此請求理想情況下會在調用的數據源上執行一些狀態更改操作。結果將由響應函數進行處理。

> **Note**
>
> `createPost`正在增加或「推出」一個新的`Post`在表中，我們的參數作為數據。我們可以添加這樣的東西：

```
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;
}
```

在此步驟中，我們還添加了request和response功能：

- request：請求處理程序接受上下文作為參數。請求處理程序返回語句執行PutItem命令，這是一個內建的 DynamoDB 作業 (請參閱這裡或者這裡例如）。該PutItem指令會新增Post透過取得分割區來反對我們的 DynamoDB 表格key值 (自動產生util.autoid()) 及attributes從上下文參數輸入（這些是我們將在請求中傳遞的值）。該key是id和attributes是date和title字段參數。他們都預先格式化通過util.dynamodb.toMapValues協助程式與動態資料表一起使用。
- response：響應接受更新的上下文並返回請求處理程序的結果。

d. 選擇創建在你完成之後。

6. 回到解析器屏幕上，在函數，選擇新增功能下拉菜單並將您的功能添加到您的功能列表中。
7. 選擇儲存以更新解析器。

CLI

若要新增您的函數

- 使用建立管線解析程式的函數create-function指令。

  您需要為此特定命令輸入一些參數：

  1. 該api-id你的 API 的。
  2. 該name中的功能AWS AppSync控制台。
  3. 該data-source-name，或函數將使用的資料來源名稱。它必須已經建立並連結至您在AWS AppSync服務。
  4. 該runtime，或函數的環境和語言。對於JavaScript，名稱必須是APPSYNC_JS，以及執行階段，1.0.0。
  5. 該code，或函數的請求和響應處理程序。雖然您可以手動輸入它，但將其添加到 .txt 文件（或類似格式）中，然後將其作為參數傳遞要容易得多。

> **Note**
>
> 我們的查詢代碼將位於作為參數傳入的文件中：
>
> ```
> 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;
> }
> ```

範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}
```

輸出將在 CLI 中返回。範例如下：

```
{
    "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**
>
> 確保您記錄了functionId某處，因為這將被用於將函數附加到解析器。

若要建立您的解析程式

- 建立管線函數Mutation透過執行create-resolver指令。

  您需要為此特定命令輸入一些參數：

  1. 該api-id你的 API 的。
  2. 該type-name，或結構描述中的特殊物件類型 (查詢、變異、訂閱)。
  3. 該field-name，或您要附加解析器的特殊物件類型內的欄位作業。
  4. 該kind，它指定單位或配管解析器。將此設定為PIPELINE以啟用管線功能。
  5. 該pipeline-config，或附加到解析器的函數。請確定您知道functionId你的函數值。上市順序事宜。
  6. 該runtime，這是APPSYNC_JS(JavaScript). 該runtimeVersion目前是1.0.0。
  7. 該code，其中包含之前和之後的步驟。

  > **Note**
  >
  > 我們的查詢代碼將位於作為參數傳入的文件中：
  >
  > ```
  > 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;
}
```

範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}
```

輸出將在 CLI 中返回。範例如下：

```
{
    "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**
>
> 在您使用 CDK 之前，我們建議您先查看 CDK 的官方文件隨著AWS AppSync的CDK 參考。
> 下面列出的步驟將僅顯示用於添加特定資源的代碼片段的一般示例。這是不意味著成為您的生產代碼中的工作解決方案。我們還假設您已經有一個可用的應用程序。

- 為了進行突變，假設你在同一個項目中，你可以像查詢一樣將它添加到堆棧文件中。這是一個修改後的函數和突變的解析器，它增加了一個新的Post到表格：

```
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**
>
> 由於這種突變和查詢的結構相似，我們將只解釋我們為了進行突變所做的更改。
> 在函數中，我們將 CFN ID 更改為`func-add-post`和名稱`add_posts_func_1`以反映我們正在添加的事實`Posts`到桌子上。在數據源中，我們與我們的表建立了新的關聯（`add_ddb_table`) 中AWS AppSync控制台作為`table-for-posts-2`因為`addDynamoDbDataSource`方法需要它。請記住，這個新關聯仍然使用我們之前創建的相同表格，但我們現在在AWS AppSync控制台：一個用於查詢`table-for-posts`和一個用於突變`table-for-posts-2`。該代碼已更改為添加`Post`通過生成其`id`自動值，並接受客戶端對其餘字段的輸入。
> 在解析器中，我們將 id 值更改為`pipeline-resolver-create-posts`以反映我們正在添加的事實`Posts`到桌子上。為了反映模式中的突變，類型名稱已更改為`Mutation`，以及名稱，`createPost`。管道配置被設置為我們的新突變功能`add_func_2`。

為了總結這個例子中發生的事情，AWS AppSync自動轉換`createPost`從您的圖形 SQL 結構描述到 DynamoDB 作業中的欄位。此範例會使用的索引鍵將記錄儲存在 DynamoDB 中`id`，這是使用我們的自動創建`util.autoId()`幫手。您傳遞給上下文參數的所有其他字段（`ctx.args.input`）從提出

的請求AWS AppSync控制台或以其他方式將被存儲為表的屬性。金鑰和屬性都會自動對應至相容的 DynamoDB 格式，使用`util.dynamodb.toMapValues(values)`幫手。

AWS AppSync 也支援編輯解析程式的測試及偵錯工作流程。你可以使用模擬`context`object 可在調用模板之前查看模板的轉換值。或者，您可以在執行查詢時，以互動方式檢視對資料來源的完整要求。如需詳細資訊，請參閱測試和除錯解析器 (JavaScript)和監控和記錄。

進階解析器

如果您正在遵循可選的分頁部分設計您的架構，您仍然需要將您的解析器添加到您的請求中以利用分頁。我們的例子中使用了查詢分頁稱為`getPosts`一次只傳回部分要求的項目。我們在該字段上的解析器代碼可能如下所示：

```
/**
 * 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 };
}
```

在請求中，我們通過請求的上下文。我們的`limit`是*`20`*，這意味著我們返回到 20`Posts`在第一個查詢中。我們的`nextToken`游標固定在第一個`Post`資料來源中的項目。這些被傳遞給參數。然後，請求從第一個執行掃描`Post`最多可達掃描限制數量。數據源將結果存儲在上下文中，該上下文被傳遞給響應。回應會傳回`Posts`它檢索，然後設置`nextToken`設定為`Post`進入限制後的權利。下一個請求被發送出去做完全相同的事情，但從第一個查詢後的偏移量開始。請記住，這些類型的請求是按順序完成的，而不是並行完成的。

測試和除錯解析器 (JavaScript)

AWS AppSync針對資料來源，在 GraphQL 欄位上執行解析程式。使用管線解析器時，函數會與您的資料來源互動。如中所述JavaScript解析器概述，函數使用寫入的要求和回應處理常式與資料來源通訊

JavaScript並在上運行APPSYNC_JS執行階段。這可讓您在與資料來源通訊之前和之後提供自訂邏輯和條件。

為了幫助開發人員編寫，測試和調試這些解析器，AWS AppSync控制台還提供了用於創建 GraphQL 請求和響應的工具，並將數據模擬到單個字段解析器。此外，您可以執行查詢、變更和訂閱AWS AppSync控制台並查看來自亞馬遜的整個請求的詳細日誌流CloudWatch。這包括來自資料來源的結果。

用模擬數據進行測試

當調用 GraphQL 解析器時，它會包含一個`context`具有有關請求相關信息的對象。其中包括用戶端引數、身分資訊，以及父 GraphQL 欄位的資料。它還存儲來自數據源，其可以在響應處理程序中使用的結果。如需有關此結構和程式設計時所要使用之可用輔助程式公用程式的詳細資訊，請參閱解析器上下文對象引用。

編寫或編輯解析器函數時，您可以傳遞嘲笑或者測試上下文對象進入控制台編輯器。這使您可以查看請求和響應處理程序如何評估，而無需實際對數據源運行。例如您可傳送測試 `firstname: Shaggy` 引數，了解該引數在範本程式碼之中使用 `ctx.args.firstname` 時如何進行評估。您也可以測試任何公用程式協助程式的評估，例如 `util.autoId()` 或 `util.time.nowISO8601()`。

測試解析器

這個例子將使用AWS AppSync用於測試解析器的控制台。

1. 登入 AWS Management Console 並開啟 AppSync主控台。
   a. 在API 儀表板」下方，選擇您的圖形 SQL API。
   b. 在側邊欄，選擇函数。
2. 選擇現有功能。
3. 在頂部更新功能頁面上，選擇選擇測試上下文，然後選擇建立新上下文。
4. 選取範例內容物件或手動填入 JSON配置測試上下文下面的窗口。
5. 輸入一個文字上下文名稱。
6. 選擇 Save (儲存) 按鈕。
7. 若要使用此項模擬的內容物件評估解析程式，選擇 Run Test (執行測試) 按鈕。

對於一個更實際的例子，假設你有一個應用程序存儲一個 GraphQL 類型Dog它會針對物件使用自動識別碼產生，並將其儲存在 Amazon DynamoDB 中。您還希望從 GraphQL 突變的參數中寫入一些值，並且只允許特定用戶看到響應。下面的代碼片段顯示了模式可能是什麼樣子：

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

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

你可以寫一個AWS AppSync功能並將其添加到您的addDog解析器來處理突變。若要測試您的AWS AppSync函數中，您可以像下面的例子一樣填充一個上下文對象。下列引數來自 name 及 age，以及將 username 填入 identity 物件之中的用戶端：

```
{
    "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"
    }
}
```

你可以測試你的AWS AppSync函數使用下面的代碼：

```
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();
}
```

評估的請求和響應處理程序具有來自測試上下文對象的數據以及從生成的值`util.autoId()`。此外，若您將 `username` 變更為 `Nadia` 以外的值，將不會傳回結果，因為授權檢查將會失敗。如需有關精細存取控制的詳細資訊，請參閱授權使用案例。

測試請求和響應處理程序AWS AppSync的 API

您可以使用`EvaluateCode`用於使用模擬數據遠程測試代碼的 API 命令。若要開始使用指令，請確定您已新增`appsync:evaluateMappingCode`允許您的政策。例如：

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

您可以使用AWS CLI或者AWS開發套件。例如，採取Dog綱要及其AWS AppSync函數請求和響應處理程序從上一節。使用本機工作站上的 CLI，將程式碼儲存到名為的檔案`code.js`，然後儲存`context`對象到一個名為的文件`context.json`。從您的 shell 中運行以下命令：

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

響應包含一個evaluationResult包含處理程序返回的有效負載。它還包含一個logsobject，其中包含處理常式在評估期間所產生的記錄清單。這可讓您輕鬆偵錯程式碼執行，並查看評估相關資訊，以協助進行疑難排解。例如：

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

該evaluationResult可以解析為 JSON，這給出了：

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

使用 SDK，您可以輕鬆地從自己喜歡的測試套件中合併測試，以驗證處理程序的行為。我們建議您使用開玩笑測試框架，但任何測試套件都可以工作。下面的代碼片段顯示了一個假設的驗證運行。請注意，我們希望評估響應是有效的 JSON，因此我們使用JSON.parse從字符串響應中檢索 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)
})
```

這會產生以下結果：

```
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
```

偵錯即時查詢

端對端測試和記錄調試生產應用程序沒有替代品。AWSAppSync可讓您使用 Amazon 記錄錯誤和完整請求詳細資訊CloudWatch。此外，您可以使用AWS AppSync控制台，用於測試 GraphQL 查詢，突變和訂閱以及每個請求的實時流日誌數據返回查詢編輯器以實時調試。針對訂閱，日誌會顯示連線時間資訊。

要執行此操作，你需要有亞馬遜CloudWatch預先啟用記錄，如中所述監控和記錄。接下來，在AWS AppSync控制台，選擇查詢選項卡，然後輸入有效的 GraphQL 查詢。在右下角區段中，按一下並拖曳日誌開啟記錄檢視的視窗。在頁面頂端，選擇執行箭頭圖示來執行您的 GraphQL 查詢。稍後，系統會將作業的完整要求和回應記錄串流至此區段，您可以在主控台中檢視它們。

管道解析器 (JavaScript)

AWS AppSync在 GraphQL 欄位上執行解析程式。在某些情況下，應用程式需要執行多個操作，才能解析單一 GraphQL 欄位。使用管道解析器，開發人員現在可以撰寫稱為 Functions 的操作並按順序執行它們。在像是以需要先執行授權檢查才能擷取欄位資料的情況中，就很適合使用管道解析程式。

有關的體系結構的更多信息JavaScript管道解析器，請參閱JavaScript解析器概述。

建立配管解析器

在AWS AppSync主控台，移至綱要頁面。

儲存下列結構描述：

```
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
}
```

我們將一個管道解析器連接到注册欄位上突變類型。在突變在右側輸入，選擇貼附旁邊的signUp突變領域。將解析器設定為pipeline resolver和APPSYNC_JS執行階段，然後建立解析程式。

我們的管道解析程式會註冊使用者，此註冊的第一步是驗證電子郵件地址輸入，然後在系統中儲存此位使用者。我們將封裝在一個電子郵件驗證驗證電子郵件功能和內部用戶的保存儲存使用者功能。validateEmail 函數會先執行，而當電子郵件驗證有效時，saveUser 函數就會接著執行。

執行流程將如下所示：

1. 突變. 註冊解析器請求處理程序
2. validateEmail 函數
3. saveUser 函數
4. 突變. 註冊解析器響應處理程序

因為我們可能會重複使用驗證電子郵件在我們的 API 上的其他解析器中的功能，我們希望避免訪問ctx.args因為這些字段將從一個 GraphQL 字段更改為另一個字段。反之，我們可以使用 ctx.stash 來存放 signUp(input: Signup) 輸入欄位引數所傳遞的電子郵件屬性。

通過替換請求和響應函數來更新您的解析器代碼：

```
export function request(ctx) {
    ctx.stash.email = ctx.args.input.email
    return {};
```

```
}

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

選擇創建或者儲存以更新解析器。

建立函數

從配管解析器頁面的函数」區段中，按一下新增功能，然後建立新函數。也可以在不經過解析器頁面的情況下創建函數; 要做到這一點，在AWS AppSync主控台，移至函数頁面。選擇 Create function (建立函數) 按鈕。讓我們來建立可檢查電子郵件是否有效且來源是特定網域的函數。如果電子郵件無效，則該函數會引發錯誤。否則，它會轉送任何獲予的任何輸入。

請確定您已建立的資料來源沒有類型。選擇此資料來源資料來源名稱列表。對於函數名稱，請輸入`validateEmail`。在函數代碼區域，用這個代碼片段覆蓋所有內容：

```
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;
}
```

檢查您的輸入，然後選擇創建。我們已建立我們 validateEmail 函數。重複這些步驟來建立儲存使用者帶有下面的代碼的函數（為了簡單起見，我們使用沒有數據源並假裝用戶在函數執行後已保存在系統中。):

```
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;
}
```

我們剛剛創建了儲存使用者功能。

將函數添加到管線解析器

我們的函數應該已經自動添加到我們剛剛創建的管道解析器中。如果不是這種情況，或者您通過函数頁面上，您可以點擊新增功能回到signUp解析器頁面附加它們。添加兩個驗證電子郵件和儲存使用者解析器的函數。validateEmail 函數應該放在 saveUser 函數之前。當您新增更多函數時，您可以使用向上移動和向下移動用於重新組織函數執行順序的選項。檢視您的變更，然後選擇儲存。

執行查詢

在AWS AppSync主控台，移至查詢頁面。在資源管理器中，確保您正在使用突變。如果您不是，請選擇Mutation在下拉列表中，然後選擇+。輸入下列查詢：

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

這應該返回如下內容：

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

我們已使用管道解析程式成功註冊我們的使用者，並完成輸入電子郵件的驗證。

設定解析器 (VTL)

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

GraphQL 解析程式將類型結構描述中的欄位連接到資料來源。解析器是通過其請求被滿足的機制。AWS AppSync 可以從模式自動創建和連接解析器，或者創建模式並從現有表連接解析器，而無需編寫任何代碼。

解析器正在AWS AppSync 使用 GraphQL 表達式轉換為數據源可 JavaScript 以使用的格式。或者，可以使用 Apache 速度範本語言 (VTL) 撰寫對應範本，將 GraphQL 運算式轉換為資料來源可使用的格式。

本節將向您展示如何使用 VTL 配置解析器。可以在 Resolver 映射模板編程指南中找到用於編寫解析器的入門教程風格編程指南，以及可在解析器映射模板上下文引用中找到編程時可以使用的幫助器實用程序。AWS AppSync 此外，還具有內建的測試和偵錯流程，您可以在從頭開始編輯或製作時使用這些流程。如需詳細資訊，請參閱測試和除錯解析器。

我們建議您在嘗試使用上述任何教學課程之前遵循本指南。

在本節中，我們將逐步介紹如何創建解析器，為突變添加解析器以及使用高級配置。

建立您的第一個解析器

遵循前幾節中的範例，第一步是為您的類型建立解析器`Query`。

Console

1. 登入 AWS Management Console 並開啟 AppSync主控台。
   a. 在 API 儀表板中，選擇您的 GraphQL API。
   b. 在側邊欄中，選擇結構描述。
2. 在頁面的右側，有一個名為解析器的窗口。此方塊包含在頁面左側的「結構描述」視窗中定義的類型和欄位清單。您可以將解析器附加到字段。例如，在「查詢」類型下，選擇`getTodos`欄位旁邊的「附加」。

3. 在「建立解析器」頁面上，選擇您在「附加資料來源」指南中建立的資料來源。在「設定對應範本」視窗中，您可以使用右側的下拉式清單選擇一般請求與回應對應範本，或自行編寫範本。

> **Note**
>
> 請求映射模板與響應映射模板的配對稱為單位解析器。單位解析器通常用於執行 rote 操作；我們建議僅將其用於具有少量數據源的單數操作。對於更複雜的操作，我們建議使用管道解析器，該解析器可以按順序對多個數據源執行多個操作。
> 有關請求和響應映射模板之間差異的詳細信息，請參閱單位解析器。
> 如需有關使用管線解析器的詳細資訊，請參閱管線解析器。

4. 對於常見使用案例，AWS AppSync 主控台具有內建範本，可用來從資料來源取得項目 (例如，所有項目查詢、個別查詢等)。例如，在設計getTodos沒有分頁的結構描述的簡單版本上，列出項目的請求映射模板如下所示：

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

5. 您始終需要一個響應映射模板來伴隨請求。主控台為清單提供具有下列傳遞值的預設值：

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

在這個範例中，項目清單的 `context` 物件 (別名為 `$ctx`) 為 `$context.result.items` 格式。如果您的 GraphQL 操作返回一個項目，那就`$context.result`是這樣。AWS AppSync 提供常見操作的輔助函`$util.toJson`數，例如先前列出的功能，以正確格式化響應。有關函數的完整列表，請參閱解析器映射模板實用程序參考。

6. 選擇「儲存解析器」。

API

1. 通過調用 API 創建一個解析器對象。`CreateResolver`
2. 您可以通過調用 API 來修改解析器的字段。`UpdateResolver`

CLI

1. 透過執行指令建立解析程式。create-resolver

   您需要為此特定命令輸入 6 個參數：

   1. 您api-id的應用程式介面。
   2. 您要在type-name結構描述中修改的類型。在控制台示例中，這是Query。
   3. 您要在類型中修改的欄位。field-name在控制台示例中，這是getTodos。
   4. 您在「貼附資料來源」指南中建立data-source-name的資料來源。
   5. 的request-mapping-template，這是請求的主體。在控制台示例中，這是：

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

   6. 的response-mapping-template，這是響應的主體。在控制台示例中，這是：

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

   範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)"
   ```

   輸出將在 CLI 中返回。範例如下：

   ```
   {
       "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. 若要修改解析程式的欄位和/或對應範本，請執行指[update-resolver](#)令。

   除了api-id參數之外，命令中使用的參數將被create-resolver命令中的新值覆蓋。update-resolver

為突變添加解析器

下一步是為您的類型創建一個解析器Mutation。

Console

1. 登入 AWS Management Console 並開啟 [AppSync主控台](#)。
   a. 在 API 儀表板中，選擇您的 GraphQL API。
   b. 在側邊欄中，選擇結構描述。
2. 在「變異類型」下，選擇addTodo欄位旁邊的「附加」。
3. 在「建立解析器」頁面上，選擇您在「[附加資料來源」指南中建立的資料來源](#)。
4. 在 [設定對應範本] 視窗中，您需要修改請求範本，因為這是您將新項目新增至 DynamoDB 的變異。請使用下列要求映射範本：

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

5. AWS AppSync 會自動將addTodo欄位中定義的引數從 GraphQL 結構描述轉換為 DynamoDB 作業。上一個範例會使用的索引鍵將記錄儲存在 DynamoDB 中id，這個索引鍵會從突變引數傳遞為。$ctx.args.id您傳遞的所有其他欄位都會使用自動對應至 DynamoDB 屬性。$util.dynamodb.toMapValuesJson($ctx.args)

   在此解析程式中，使用下列的回應映射範本：

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

AWS AppSync 還支持用於編輯解析器的測試和調試工作流程。您可以使用模擬 context 物件，先查看範本轉換後的值，然後再叫用。或者，您可以在執行查詢時以互動方式檢視對資料來源的完整要求執行。如需詳細資訊，請參閱測試和偵錯解析器和監視和記錄。

6. 選擇「儲存解析器」。

API

您也可以利用 [建立您的第一個解析程式] 區段中的指令以及本節中的參數詳細資訊，使用 API 來執行此操作。

CLI

您也可以使用建立您的第一個解析器一節中的指令，以及本節中的參數詳細資訊，在 CLI 中執行此操作。

此時，如果您沒有使用進階解析器，您可以開始使用 GraphQL API，如使用您的 API 所述。

進階解析器

如果您正在遵循 [進階] 區段，而且要在 [設計結構描述] 中建立範例結構描述以進行分頁掃描，請改為使用下列getTodos欄位的要求範本：

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

對此分頁使用案例而言，回應映射不只是傳遞，因為其中必須包含游標 (讓用戶端了解接下來從哪個頁面開始) 和結果集。映射範本如下所示：

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

前述回應映射範本的欄位，應符合 TodoConnection 類型之中定義的欄位。

對於您有一個Comments表，並且您正在解析類型上的註釋字段（返回一個Todo類型[Comment]）的關係的情況下，您可以使用對第二個表運行查詢的映射模板。若要這麼做，您必須已經建立Comments表格的資料來源，如附加資料來源中所述。

> **Note**
>
> 我們對第二個表使用查詢操作僅用於說明目的。您可以改為對 DynamoDB 使用另一個作業。此外，您可以從其他資料來源 (例如AWS Lambda或 Amazon OpenSearch 服務) 提取資料，因為該關係是由 GraphQL 結構描述所控制。

Console

1. 登入 AWS Management Console 並開啟 AppSync主控台。
   a. 在 API 儀表板中，選擇您的 GraphQL API。
   b. 在側邊欄中，選擇結構描述。
2. 在 [待辦事項類型] 下方，選擇comments欄位旁邊的 [附加]。
3. 在「建立解析器」頁面上，選擇「註解」表格資料來源。快速入門指南中「註解」表格的預設名稱為AppSyncCommentTable，但可能會因您提供的名稱而有所不同。
4. 將下列程式碼片段新增至您的請求對應範本：

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

5. context.source 參照目前正在解析之欄位的父物件。在這個範例中，source.id 代表個別 Todo 物件，這個物件會在稍後用於查詢表達式。

您可使用傳遞回應映射範本，如下所示：

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

6. 選擇「儲存解析器」。
7. 最後，返回控制台中的「結構描述」頁面，將解析器附加到該addComment字段，然後指定表的數據源。Comments在這種情況下，要求映射範本是簡單的 PutItem，其中具有註解為引數的特定 todoid，但您需要使用 $utils.autoId() 公用程式來為註解建立唯一的排序索引鍵，如下所示：

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

使用傳遞回應範本，如下所示：

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

API

您也可以利用 [建立您的第一個解析程式] 區段中的指令以及本節中的參數詳細資訊，使用 API 來執行此操作。

CLI

您也可以使用建立您的第一個解析器一節中的指令，以及本節中的參數詳細資訊，在 CLI 中執行此操作。

直接 Lambda 解析器 (VTL)

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

使用直接 Lambda 解析器，您可以在使用資料來源時規避 VTL 對應範本的使用。AWS LambdaAWS AppSync 可以為 Lambda 函數提供預設承載，以及 Lambda 函數對 GraphQL 類型回應的預設轉譯。您可以選擇提供請求模板，響應模板或兩者都不提供，並且AWS AppSync 將相應地處理它。

若要進一步了解AWS AppSync 提供的預設要求承載和回應轉譯，請參閱直接 Lambda 解析器參考資料。如需有關設定資AWS Lambda料來源和設定 IAM 信任政策的詳細資訊，請參閱附加資料來源。

設定直接 Lambda 解析器

以下各節將說明如何連接 Lambda 資料來源，以及如何將 Lambda 解析器新增至欄位。

新增 Lambda 資料來源

您必須先新增 Lambda 資料來源，才能啟用直接 Lambda 解析器。

Console

1. 登入 AWS Management Console 並開啟 AppSync主控台。
   a. 在 API 儀表板中，選擇您的 GraphQL API。
   b. 在側邊欄中，選擇資料來源。
2. 選擇 Create data source (建立資料來源)。
   a. 對於資料來源名稱，請輸入資料來源的名稱，例如**`myFunction`**。
   b. 對於資料來源類型，請選擇AWS Lambda函數。
   c. 針對「區域」，選擇適當的區域。
   d. 對於函數 ARN，請從下拉式清單中選擇 Lambda 函數。您可以搜尋函數名稱，或手動輸入要使用之函數的 ARN。
   e. 建立新的 IAM 角色 (建議使用) 或選擇具有 `lambda:invokeFunction` IAM 權限的現有角色。現有角色需要信任原則，如附加資料來源一節所述。

      以下是具有對資源執行操作所需許可的 IAM 政策範例：

```
{
     "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. 選擇「建立」按鈕。

CLI

1. 透過執行[create-data-source](#)指令建立資料來源物件。

   您需要為此特定命令輸入 4 個參數：

   1. 您`api-id`的應用程式介面。
   2. 您`name`的資料來源。在主控台範例中，這是資料來源名稱。
   3. 資`type`料來源的。在控制台示例中，這是AWS Lambda函數。
   4. 的`lambda-config`，這是在控制台示例中的函數 ARN。

   > **Note**
   >
   > 必須設定其他參數`Region`，例如，但通常會預設為 CLI 組態值。

   範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
```

輸出將在 CLI 中返回。範例如下：

```
{
    "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. 若要修改資料來源的屬性，請執行update-data-source命令。

   除了api-id參數之外，命令中使用的參數將被create-data-source命令中的新值覆蓋。update-data-source

啟動直接 Lambda 解析器

建立 Lambda 資料來源並設定適當的 IAM 角色以允許AWS AppSync 呼叫函數之後，您可以將其連結至解析器或管線函數。

Console

1. 登入 AWS Management Console 並開啟 AppSync主控台。
   a. 在 API 儀表板中，選擇您的 GraphQL API。
   b. 在側邊欄中，選擇結構描述。
2. 在 [解析器] 視窗中，選擇欄位或作業，然後選取 [附加] 按鈕。
3. 在「建立新解析器」頁面中，從下拉式清單中選擇 Lambda 函數。
4. 若要利用直接 Lambda 解析器，請在 [設定對應範本] 區段中確認請求和回應對應範本已停用。
5. 選擇「儲存解析器」按鈕。

CLI

- 透過執行指令建立解析程式。[create-resolver]

  您需要為此特定命令輸入 6 個參數：

  1. 您api-id的應用程式介面。
  2. 結構描述中的類型。type-name
  3. 結構描述中的欄位。field-name
  4. 或您的 data-source-name Lambda 函數的名稱。
  5. 的request-mapping-template，這是請求的主體。在控制台示例中，這被禁用：

     ```
     " "
     ```

  6. 的response-mapping-template，這是響應的主體。在控制台示例中，這也被禁用：

     ```
     " "
     ```

  範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 " "
  ```

  輸出將在 CLI 中返回。範例如下：

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

當您停用對應範本時，會發生數個其他行為AWS AppSync：

- 透過停用對應範本，即表示您接受 Direct Lambda 解析器參考中指定的預設資料轉換。AWS AppSync
- 藉由停用請求對應範本，您的 Lambda 資料來源將收到由整個 Context 物件組成的承載。
- 透過停用回應對應範本，系統會根據請求對應範本的版本或請求對應範本是否也停用轉譯 Lambda 叫用的結果。

測試和除錯解析器 (VTL)

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync 針對資料來源，在 GraphQL 欄位上執行解析程式。如解析器映射模板概述中所述，解析器使用模板語言與數據源進行通信。這可讓您自訂行為，並在與資料來源通訊之前和之後套用邏輯和條件。有關編寫解析器的入門教程式編程指南，請參閱解析器映射模板編程指南。

為了幫助開發人員編寫，測試和調試這些解析器，AWS AppSync 控制台還提供了創建 GraphQL 請求和響應的工具，並將數據模擬到單個字段解析器。此外，您可以在AWS AppSync 主控台中執行查詢、突變和訂閱，並查看來自 Amazon CloudWatch 的整個請求的詳細日誌資料流。其中包括資料來源的結果。

用模擬數據進行測試

當調用 GraphQL 解析器時，它包含一個包含有關請求信息的`context`對象。其中包括用戶端引數、身分資訊，以及父 GraphQL 欄位的資料。它還包含從數據源，它可以在響應模板中使用的結果。如需詳細資訊來了解此項架構，以及程式設計時可用的各種協助公用程式，請參閱解析程式映射範本內容參考。

編寫或編輯解析器時，您可以將模擬或測試上下文對象傳遞給控制台編輯器。這可讓您在沒有實際依據資料來源執行的情況下，了解這兩者如何請求及回應範本評估。例如您可傳送測試 `firstname: Shaggy` 引數，了解該引數在範本程式碼之中使用 `$ctx.args.firstname` 時如何進行評估。您也可以測試任何公用程式協助程式的評估，例如 `$util.autoId()` 或 `util.time.nowISO8601()`。

測試解析器

這個例子將使用AWS AppSync 控制台來測試解析器。

1. 登入 AWS Management Console 並開啟 AppSync主控台。
   a. 在 API 儀表板中，選擇您的 GraphQL API。
   b. 在側邊欄中，選擇結構描述。
2. 如果您尚未執行此操作，請在類型下方並在欄位旁邊選擇 [附加] 以新增您的解析程式。

   如需有關如何建置完整解析器的詳細資訊，請參閱設定解析器。

   否則，請選取欄位中已存在的解析器。
3. 在「編輯解析器」頁面頂端，選擇選取測試相關資訊環境，然後選擇建立新的相關資訊環境。
4. 選取範例前後關聯物件，或在下方的 [執行] 內容視窗中手動填入 JSON。
5. 輸入文字前後關聯名稱。
6. 選擇 Save (儲存) 按鈕。
7. 在 「編輯解析器」頁面頂端，選擇「執行測試」。

如需更實用的範例，假設您有一個應用程式儲存 GraphQL 類型，Dog該應用程式使用物件的自動 ID 產生，並將它們儲存在 Amazon DynamoDB 中。您也希望寫入 GraphQL 變動引數的值，並且只讓特定使用者看見回應。結構描述看起來類似如下：

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

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

當您為addDog突變添加解析器時，可以像以下示例一樣填充上下文對象。下列引數來自 `name` 及 `age`，以及將 `username` 填入 `identity` 物件之中的用戶端：

```
{
    "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"
    }
}
```

您可利用下列要求及回應映射範本對此進行測試：

請求範本

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

回應範本

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

經過評估的範本擁有測試內容物件的資料，以及 `$util.autoId()` 產生的值。此外，若您將 `username` 變更為 `Nadia` 以外的值，將不會傳回結果，因為授權檢查將會失敗。如需有關精細存取控制的詳細資訊，請參閱授權使用案例。

使用AWS AppSync的 API 測試映射模板

您可以使用 `EvaluateMappingTemplate` API 命令從遠端測試具有模擬資料的對應範本。若要開始使用命令，請確定您已將`appsync:evaluateMappingTemplate`權限新增至您的原則。例如：

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

您可以使用AWS CLI或 AWSSDK 來利用命令。例如，取得上一節的Dog結構描述及其要求/回應對映範本。使用本機工作站上的 CLI，將要求範本儲存至名為的檔案`request.vtl`，然後將`context`物件儲存到名為的檔案中`context.json`。從您的 shell 中運行以下命令：

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

此命令會傳回下列回應：

```
{
  "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`包含使用提供的測試您提供的範本的結果`context`。您也可以使用 AWS SDK 測試範本。以下是使用適用於 JavaScript V2 AWS SDK 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))
```

使用 SDK，您可以輕鬆地從自己喜歡的測試套件中合併測試，以驗證模板的行為。我們建議使用 Jest 測試框架創建測試，但任何測試套件都可以正常工作。下面的代碼片段顯示了一個假設的驗證運行。請注意，我們希望評估響應是有效的 JSON，因此我們使JSON.parse用從字符串響應中檢索 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)
})
```

這會產生以下結果：

```
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
```

偵錯即時查詢

end-to-end 測試和記錄調試生產應用程序沒有替代品。AWS AppSync 可讓您使用 Amazon 記錄錯誤和完整請求詳細資訊 CloudWatch。此外您可使用 AWS AppSync 主控台測試 GraphQL 查詢、變動，以及訂閱和各個要求的即時串流日誌資料，傳回查詢編輯器進行即時偵錯。針對訂閱，日誌會顯示連線時間資訊。

若要執行此操作，您必須預先啟用 Amazon CloudWatch 日誌，如監控和記錄中所述。接下來，在 AWS AppSync 主控台中，選擇 Queries (查詢) 標籤，然後輸入有效的 GraphQL 查詢。在右下角區段中，按一下並拖曳「記錄檔」視窗，以開啟記錄檢視。在頁面頂端，選擇執行箭頭圖示來執行您的 GraphQL 查詢。幾分鐘後，操作的完整請求及回應日誌，將串流至此區段，然後您可在主控台中檢視。

管線解析器 (VTL)

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync 在 GraphQL 欄位上執行解析程式。在某些情況下，應用程式需要執行多個操作，才能解析單一 GraphQL 欄位。使用管道解析器，開發人員現在可以撰寫稱為 Functions 的操作並按順序執行它們。在像是以需要先執行授權檢查才能擷取欄位資料的情況中，就很適合使用管道解析程式。

管道解析程式包含 Before 映射範本和 After 映射範本，以及一份函數清單。每個函數都有一個請求和響應映射模板，它對數據源執行。由於管道解析程式是將執行委派到一份函數清單，所以不會連結到任何資料來源。單元解析器和函數是對數據源執行操作的基元。如需詳細資訊，請參閱解析器對映範本概觀。

建立管道解析程式

在 AWS AppSync 主控台中，移至Schema (結構描述) 頁面。

儲存下列結構描述：

```
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
}
```

我們將連接一個管道解析器到突變類型上的 sign U p 字段。在右側的「突變類型」中，選擇「signUp突變」欄位旁邊的「附加」。在「創建解析器」頁面上，單擊「操作」，然後單擊「更新運行時」。選擇Pipeline Resolver，然後選擇VTL，然後選擇更新。此頁面現在應該會顯示三個區段：「對映前範本」文字區域、「函數」區段和「後對映範本」文字區域。

我們的管道解析程式會註冊使用者，此註冊的第一步是驗證電子郵件地址輸入，然後在系統中儲存此位使用者。我們將在 validateEmail 函數中封裝該電子郵件驗證，並在 saveUser 函數中封裝使用者儲存步驟。validateEmail 函數會先執行，而當電子郵件驗證有效時，saveUser 函數就會接著執行。

執行流程將如下所示：

1. Mutation.signUp 解析程式要求映射範本
2. validateEmail 函數
3. saveUser 函數
4. Mutation.signUp 解析程式回應映射範本

因為我們可能會在我們的 API 上的其他解析器中重複使用 validateEmail 函數，所以我們希望避免訪問，$ctx.args因為這些將從一個 GraphQL 字段更改為另一個字段。反之，我們可以使用 $ctx.stash 來存放 signUp(input: Signup) 輸入欄位引數所傳遞的電子郵件屬性。

之前對映範本：

```
## store email input field into a generic email key
$util.qr($ctx.stash.put("email", $ctx.args.input.email))
{}
```

控制台提供了一個默認的傳遞 AF TER 映射模板，我們將使用它：

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

選擇「建立」或「儲存」以更新解析程式。

建立函數

在管線解析器頁面的函數區段中，按一下新增函數，然後按一下建立新函數。也可以在不經過解析器頁面的情況下創建函數; 要做到這一點，在AWS AppSync控制台中，轉到 Functions 頁面。選擇 Create function (建立函數) 按鈕。讓我們來建立可檢查電子郵件是否有效且來源是特定網域的函數。如果電子郵件無效，則該函數會引發錯誤。否則，它會轉送任何獲予的任何輸入。

在新功能頁面上，選擇動作，然後選擇更新執行階段。選擇VTL，然後選擇更新。請確定您已建立 NONE 類型的資料來源。在「資料來源名稱」清單中選擇此資料來源。對於函數名稱，請輸入 in `validateEmail`。在函數代碼區域中，用這個代碼片段覆蓋所有內容：

```
#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}) }
}
```

將其粘貼到響應映射模板中：

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

檢閱您的變更，然後選擇「建立」。我們已建立我們 validateEmail 函數。重複這些步驟，使用以下請求和響應映射模板創建 saveUser 函數（為了簡單起見，我們使用 NONE 數據源並假裝該用戶在函數執行後已保存在系統中。):

要求映射範本：

```
## $ctx.prev.result contains the signup input values. We could have also
## used $ctx.args.input.
{
    "payload": $util.toJson($ctx.prev.result)
}
```

回應映射範本：

```
## 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)
```

我們剛剛創建了我們的 saveUser 功能。

新增函數到管道解析程式

我們的函數應該已經自動添加到我們剛剛創建的管道解析器中。如果不是這種情況，或者您通過「函數」頁面創建了函數，則可以單擊解析器頁面上的添加函數以附加它們。兩個 validateEmail 和 saveUser 功能添加到解析器。validateEmail 函數應該放在 saveUser 函數之前。當您新增更多函數時，您可以使用「上移」和「下移」選項來重新組織函數的執行順序。檢閱您的變更，然後選擇 [儲存]。

執行查詢

在 AWS AppSync 主控台中，移至Queries (查詢) 頁面。在資源管理器中，確保您正在使用突變。如果不是，請Mutation在下拉式清單中選擇，然後選擇+。輸入下列查詢：

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

這應該返回如下內容：

```
{
  "data": {
    "signUp": {
      "id": "256b6cc2-4694-46f4-a55e-8cb14cc5d7fc",
      "email": "nadia@myvaliddomain.com"
    }
  }
}
```

我們已使用管道解析程式成功註冊我們的使用者，並完成輸入電子郵件的驗證。若要取得更多著重管道解析程式的全面教學課程，您可以移至教學課程：管道解析程式

第 4 步：使用 API：CDK 示例

> **Tip**
>
> 在您使用 CDK 之前，我們建議您先查看 CDK 的官方文件隨著AWS AppSync的CDK 參考資料。
> 我們還建議您確保AWSCLI和NPM安裝正在您的系統上運作。

在本節中，我們將建立一個簡單的 CDK 應用程式，該應用程式可以新增和擷取 DynamoDB 資料表中的項目。這是一個快速入門示例，使用來自設計您的架構,貼附資料來源，以及配置解析器（JavaScript)部分。

建立 CDK 專案

> **Warning**
>
> 根據您的環境，這些步驟可能不完全準確。我們假設您的系統安裝了必要的實用程序，這是一種與AWS服務和適當的配置。

第一步是安裝AWSCDK。在 CLI 中，您可以輸入以下命令：

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

接下來，您需要創建一個項目目錄，然後導航到它。建立並導覽至目錄的指令集範例為：

```
mkdir example-cdk-app
cd example-cdk-app
```

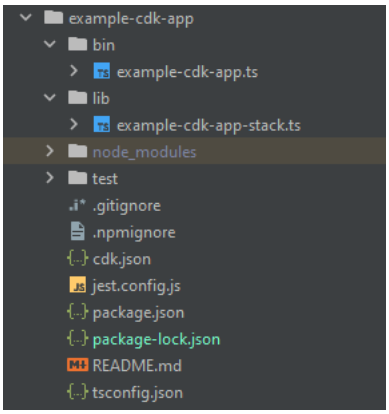
接下來，您需要創建一個應用程序。我們的服務主要使用TypeScript。在您的項目目錄中，輸入以下命令：

```
cdk init app --language typescript
```

當你這樣做時，CDK 應用程序及其初始化文件將被安裝：

```
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!
```

您的項目結構可能如下所示：



您會注意到我們有幾個重要的目錄：

- `bin`：初始 bin 文件將創建該應用程序。我們不會在本指南中觸及此內容。
- `lib`：lib 目錄包含您的堆棧文件。您可以將堆棧文件視為單獨的執行單元。構造將在我們的堆棧文件中。基本上，這些都是將要轉化的服務的資源AWS CloudFormation部署應用程式時。這是我們大多數編碼都會發生的地方。
- `node_modules`：此目錄由 NPM 建立，並包含您使用npm指令。

我們的初始堆棧文件可能包含這樣的內容：

```
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)
    // });
  }
}
```

這是在我們的應用程序中創建堆棧的樣板代碼。我們在這個例子中的大多數代碼都會在這個類的範圍內。

要驗證堆棧文件是否位於應用程序中，請在應用程序的目錄中，在終端中運行以下命令：

```
cdk ls
```

您的堆疊清單應會出現。如果沒有，那麼您可能需要再次執行這些步驟或查看官方文檔以獲取幫助。

如果您想在部署之前構建代碼更改，則始終可以在終端機中運行以下命令：

```
npm run build
```

並且，要在部署之前查看更改：

```
cdk diff
```

在我們將我們的代碼添加到堆棧文件之前，我們要執行一個引導。引導功能使我們能夠在應用程序部署之前為 CDK 佈建資源。可以找到有關此過程的更多信息這裡。要創建一個引導程序，命令是：

```
cdk bootstrap aws://ACCOUNT-NUMBER/REGION
```

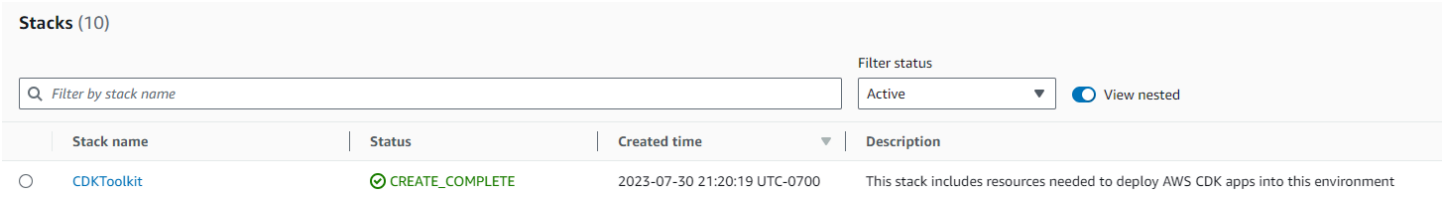
> **Tip**
>
> 此步驟需要您的帳戶中有多個 IAM 許可。如果您沒有引導程序，您的引導程序將被拒絕。如果發生這種情況，您可能必須刪除由啟動程序導致的不完整資源，例如其產生的 S3 儲存貯體。

引導將旋轉了幾個資源。最後的消息將如下所示：

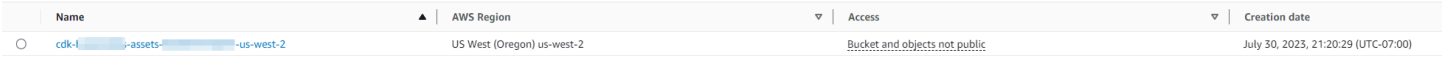
```
⏳ Bootstrapping environment
Trusted accounts for deployment: (none)
Trusted accounts for lookup: (none)
Using default execution policy of 'arn:aws:iam::aws:policy/AdministratorAccess'. Pass '--cloudformation-execution-policies' to customize.
CDKToolkit: creating CloudFormation changeset...
✅ Environment bootstrapped.
```

每個區域的每個帳戶只能完成一次，因此您不必經常這樣做。引導程序的主要資源是AWS CloudFormation堆棧和亞馬遜 S3 桶。

Amazon S3 儲存貯體用於存放檔案和 IAM 角色，這些角色授予執行部署所需的許可。所需的資源定義在AWS CloudFormation堆棧，稱為引導堆棧，該堆棧通常被命名CDKToolkit。像任何AWS CloudFormation堆棧，它出現在AWS CloudFormation控制台一旦部署：



對於桶也可以這樣說：



要導入我們需要在堆棧文件中的服務，我們可以使用以下命令：

```
npm install aws-cdk-lib # V2 command
```

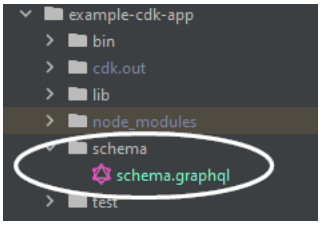
> **Tip**
>
> 如果您在使用 V2 時遇到問題，可以使用 V1 命令安裝單個庫：
>
> ```
> npm install @aws-cdk/aws-appsync @aws-cdk/aws-dynamodb
> ```
>
> 我們不建議這樣做，因為 V1 已被棄用。

實現 CDK 項目-模式

我們現在可以開始實作我們的程式碼。首先，我們必須創建我們的模式。你可以簡單地創建一個`.graphql`您的應用程序中的文件：

```
mkdir schema
touch schema.graphql
```

在我們的例子中，我們包括一個名為的頂級目錄schema包含我們的schema.graphql:



在我們的模式中，讓我們包括一個簡單的例子：

```
input CreatePostInput {
    title: String
    content: String
}

type Post {
    id: ID!
    title: String
    content: String
}

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

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

回到我們的堆棧文件中，我們需要確保定義以下導入指令：

```
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';
```

在類中，我們將添加代碼來製作我們的 GraphQL API 並將其連接到我們的schema.graphql檔案:

```
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'),
    });
  }
}
```

我們還將添加一些代碼來打印出 GraphQL 網址，API 密鑰和區域：

```
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
    });
  }
}
```

此時，我們將再次使用部署我們的應用程序：

```
cdk deploy
```

這是結果：

看來我們的例子是成功的，但讓我們檢查一下AWS AppSync控制台只是為了確認：

看來我們的 API 已建立。現在，我們將檢查附加到 API 的模式：

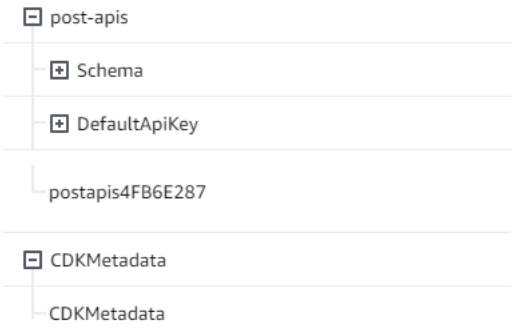
Schema

```
1  input CreatePostInput {
2    title: String
3    date: AWSDateTime
4  }
5
6  type Post {
7    id: ID!
8    title: String
9    date: AWSDateTime
10 }
11
12 type Mutation {
13   createPost(input: CreatePostInput!): Post
14 }
15
16 type Query {
17   getPost: [Post]
18 }
```

這似乎與我們的模式代碼匹配，所以它成功了。從元數據角度確認這一點的另一種方法是查看AWS CloudFormation堆疊：

○	ExampleCdkAppStack	UPDATE_COMPLETE	2023-07-30 22:13:31 UTC-0700	-
○	CDKToolkit	CREATE_COMPLETE	2023-07-30 21:20:19 UTC-0700	This stack includes resources needed to deploy AWS CDK apps into this environment

當我們部署我們的 CDK 應用程序時，它會經歷AWS CloudFormation旋轉像引導程序這樣的資源。我們應用程序中的每個堆棧都將 1：1 映射AWS CloudFormation堆疊。如果您返回堆棧代碼，則從類名中獲取堆棧名稱`ExampleCdkAppStack`。您可以看到它所建立的資源，這些資源也符合 GraphQL API 建構中的命名慣例：



實施 CDK 項目-數據源

接下來，我們需要添加我們的數據源。我們的範例將使用 DynamoDB 資料表。在堆棧類中，我們將添加一些代碼來創建一個新的表：

```
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", {
```

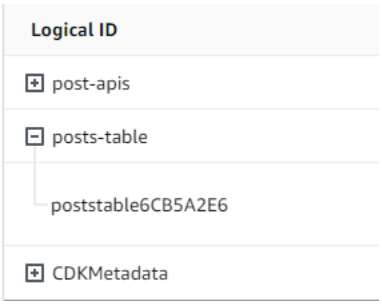
```
      value: api.graphqlUrl
    });

    // 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
    });
  }
}
```

在這一點上，讓我們再次部署：

```
cdk deploy
```

我們應該檢查 DynamoDB 主控台以取得新資料表：

我們的堆棧名稱是正確的，表名與我們的代碼匹配。如果我們檢查我們的AWS CloudFormation再次堆疊，我們現在將看到新表：



實施 CDK 項目-解析器

這個例子將使用兩個解析器：一個用於查詢表，另一個用於添加到該表中。由於我們使用管道解析器，因此我們需要聲明兩個管道解析器，每個管道解析器中都有一個函數。在查詢中，我們將添加以下代碼：

```
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],
  });
```

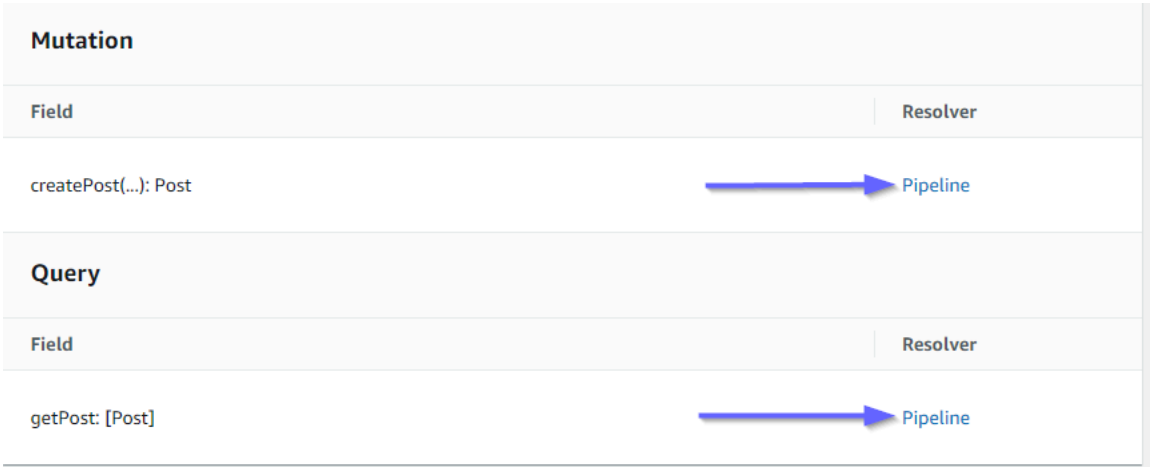
```
    // Prints out URL
    new cdk.CfnOutput(this, "GraphQLAPIURL", {
      value: api.graphqlUrl
    });

    // 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
    });
  }
}
```

在此代碼片段中，我們添加了一個名為的管道解析器pipeline-resolver-create-posts與一個名為的函數func-add-post附加到它。這是將添加的代碼Posts到表中。另一個管道解析器被稱為pipeline-resolver-get-posts與一個名為的函數func-get-post檢索Posts已新增至表格。

我們將部署它以將其添加到AWS AppSync服務：

```
cdk deploy
```

讓我們檢查一下AWS AppSync控制台以查看它們是否已連接到我們的 GraphQL API：



這似乎是正確的。在代碼中，這兩個解析器都附加到我們製作的 GraphQL API 上（由apiprop值存在於解析器和函數）。在 GraphQL API 中，我們附加解析器的欄位也在道具中指定(由typename和fieldname道具在每個解析器）。

讓我們看看解析器的內容是否正確pipeline-resolver-get-posts:

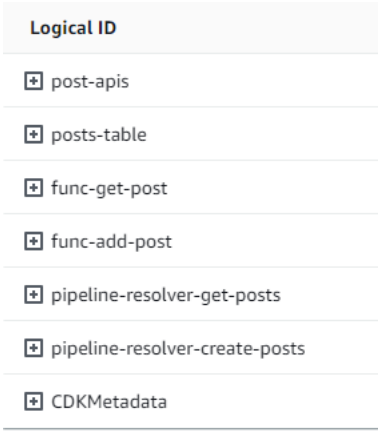
之前和之後的處理程序與我們的code道具值。我們也可以看到一個叫做的函數add_posts_func_1，它與我們在解析器中附加的函數的名稱相匹配。

讓我們來看看該函數的代碼內容：

這與匹配code的道具add_posts_func_1功能。我們的查詢已成功上傳，所以讓我們檢查查詢：

這些也符合程式碼。如果我們看一下get_posts_func_1:

一切似乎都到位了。要從元數據的角度確認這一點，我們可以檢查我們的堆棧AWS CloudFormation再次：



現在，我們需要通過執行一些請求來測試此代碼。

實施 CDK 項目-請求

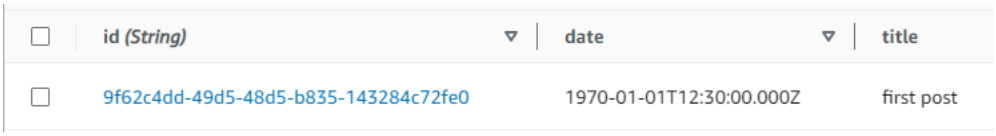
若要測試我們的應用程式AWS AppSync控制台，我們做了一個查詢和一個突變：

```
query MyQuery {
  getPost {
    id
    date
    title
  }
}

mutation MyMutation {
  createPost(input: {date: "1970-01-01T12:30:00.000Z", title: "first post"}) {
    date
    id
    title
  }
}

```

MyMutation包含一個createPost操作與參數1970-01-01T12:30:00.000Z和first post。它返回date和title我們傳入以及自動生成的id價值。運行突變產生的結果：

```
{
  "data": {
    "createPost": {
      "date": "1970-01-01T12:30:00.000Z",
      "id": "4dc1c2dd-0aa3-4055-9eca-7c140062ada2",
      "title": "first post"
    }
  }
}
```

如果我們快速檢查 DynamoDB 表，我們可以在掃描時在表格中看到我們的項目：



id *(String)*	date	title
9f62c4dd-49d5-48d5-b835-143284c72fe0	1970-01-01T12:30:00.000Z	first post

回到了AWS AppSync控制台，如果我們運行查詢來檢索這個Post，我們得到以下結果：

```
{
  "data": {
    "getPost": [
      {
        "id": "9f62c4dd-49d5-48d5-b835-143284c72fe0",
        "date": "1970-01-01T12:30:00.000Z",
        "title": "first post"
      }
    ]
  }
```

```
}
```

即時資料

AWS AppSync允許您使用訂閱來實現實時應用程序更新，推送通知等。當用戶端叫用 GraphQL 訂閱作業時，會由 AWS AppSync. WebSocket 然後，應用程式可以從資料來源即時將資料分發給訂閱者，同時AWS AppSync 持續管理應用程式的連線和擴展需求。以下部分將向您展示訂閱的AWS AppSync工作方式。

結 GraphQL 述訂閱指令

會呼叫 AWS AppSync 中的訂閱做為變動的回應。這表示您可以透過對變動指定 GraphQL 結構描述指令來在 AWS AppSync 中即時製作任何資料來源。

用AWS Amplify戶端程式庫會自動處理訂閱連線管理。這些程式庫使用純粹 WebSockets 做為用戶端與服務之間的網路通訊協定。

> **Note**
>
> 若要在連線到訂閱時控制授權，您可以使用 AWS Identity and Access Management (IAM) AWS Lambda、Amazon Cognito 身分集區或 Amazon Cognito 使用者集區進行欄位層級授權。若要對訂閱進行精細分級的存取控制，您可以將解析程式連接至訂閱欄位，並使用呼叫者的身分和 AWS AppSync 資料來源來執行邏輯。如需詳細資訊，請參閱授權與驗證。

會從變動觸發訂閱且會將變動選擇設定傳送給訂閱者。

以下範例示範如何使用 GraphQL 訂閱。它不會指定資料來源，因為資料來源可能是 Lambda、Amazon DynamoDB 或亞馬遜 OpenSearch 服務。

若要開始使用訂閱，您必須將訂閱進入點新增至結構描述，如下所示：

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

假設您有一個部落格文章網站，以及您想要訂閱新部落格和變更現有的部落格。若要執行此操作，請將以下 `Subscription` 定義新增至結構描述：

```
type Subscription {
    addedPost: Post
    updatedPost: Post
    deletedPost: Post
}
```

假設您有以下變動：

```
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!
}
```

您可以透過為您希望接收通知的每個訂閱新增 `@aws_subscribe(mutations: ["mutation_field_1", "mutation_field_2"])` 指示 (如下所示) 讓這些欄位變得即時：

```
type Subscription {
    addedPost: Post
    @aws_subscribe(mutations: ["addPost"])
    updatedPost: Post
    @aws_subscribe(mutations: ["updatePost"])
    deletedPost: Post
    @aws_subscribe(mutations: ["deletePost"])
}
```

由於`@aws_subscribe(mutations: ["",..,""])`需要一組突變輸入，因此您可以指定多個突變，從而啟動訂閱。如果您是從用戶端進行訂閱，您的 GraphQL 查詢可能如下所示：

```
subscription NewPostSub {
    addedPost {
        __typename
        version
        title
        content
        author
        url
    }
}
```

用戶端連線和工具需要此訂閱查詢。

使 WebSockets 用純用戶端時，選集篩選是針對每個用戶端進行，因為每個用戶端都可以定義自己的選集。在此情況下，訂閱選項集必須是變動選項集的子集。例如，連結至變動 `addPost(...){id author title url version}` 的訂閱 `addedPost{author title}` 僅接收文章的作者和標題。而沒有接收其他欄位。但是，如果變動缺少其選項集中的作者，則訂閱者將取得作者欄位的 `null` 值 (或者如果作者欄位在結構描述中被定義為必要/非 null，則會出現錯誤)。

使用 pure 時，訂閱選擇組是必不可少的 WebSockets。如果未在訂閱中明確定義欄位，則AWS AppSync不會傳回欄位。

在上述範例中，訂閱沒有引數。假設您的結構描述如下所示：

```
type Subscription {
    updatedPost(id:ID! author:String): Post
    @aws_subscribe(mutations: ["updatePost"])
}
```

在此情況下，您的用戶端定義訂閱如下：

```
subscription UpdatedPostSub {
    updatedPost(id:"XYZ", author:"ABC") {
        title
        content
    }
}
```

在結構描述 `subscription` 欄位的傳回類型必須與對應變動欄位的傳回類型相符。在上述範例中，這是以 `addPost` 類型形式傳回做為 `addedPost` 和 `Post` 而顯示。

若要在用戶端上設定訂閱，請參閱建立用戶端應用程式。

使用訂閱引數

使用 GraphQL 訂閱的重要組成部分是了解何時以及如何使用引數。您可以進行細微變更，以修改通知用戶端發生的突變的方式和時機。若要這麼做，請參閱快速入門章節中的範例結構描述，該章節會建立「Todos」。針對此範例結構描述，會定義下列突變：

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

```
}
```

在預設範例中，用戶端可以使用無引數來訂閱任何Todo更新：onUpdateTodosubscription

```
subscription OnUpdateTodo {
  onUpdateTodo {
    description
    id
    name
    when
  }
}
```

您可以使用其參數subscription來篩選您的。例如，要僅在更新具todo有特ID定功能的subscription時候觸發 a，請指定ID值：

```
subscription OnUpdateTodo {
  onUpdateTodo(id: "a-todo-id") {
    description
    id
    name
    when
  }
}
```

您也可以傳遞多個參數。例如，以下內容subscription示範如何在特定地點和時間取得任何Todo更新的通知：

```
subscription todosAtHome {
  onUpdateTodo(when: "tomorrow", where: "at home") {
    description
    id
    name
    when
    where
  }
}
```

請注意，所有參數都是可選的。如果您未在您的中指定任何引數subscription，您將會訂閱應用程式中發生的所有Todo更新。但是，您可以更新您subscription的字段定義以要求ID參數。這將強制特定todo而不是所有 todo s 的響應：

```
onUpdateTodo(
  id: ID!,
  name: String,
  when: String,
  where: String,
  description: String
): Todo
```

引數 Null 值具有意義

在 AWS AppSync 中進行訂閱查詢時，null 引數值與整個忽略該引數時的情況不同，會以不同方式篩選結果。

讓我們回到待辦事項 API 示例，我們可以創建待辦事項。請參閱快速入門一章中的範例結構描述。

讓我們修改我們的模式，以包括一個新的ownerTodo字段，描述誰是所有者。此owner欄位不是必填欄位，且只能在上設定UpdateTodoInput。請參閱以下簡化版本的模式：

```
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"])
}
```

下列訂閱會傳回所有Todo更新：

```
subscription MySubscription {
  onUpdateTodo {
    description
    id
    name
    when
    where
  }
}
```

如果您修改先前的訂閱以新增欄位引數owner: null，您現在會提出不同的問題。此訂閱現在會註冊用戶端，以取得尚未提供擁有者的所有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，MQTT 以上 WebSockets 版本已不再作為 API 中 GraphQL 訂閱的通訊協定提供。AWS AppSyncPure WebSockets 是中唯一支援的通訊協定AWS AppSync。

2019 年 11 月之後發行的 AWS AppSync SDK 或 Amplify 程式庫的用戶端會自動使用純 WebSockets 預設值。將客戶端升級到最新版本允許他們使用AWS AppSync的純 WebSockets 引擎。
Pure 具 WebSockets 有更大的承載大小（240 KB），更多種客戶端選項以及改進的 CloudWatch 指標。如需使用純用 WebSocket 戶端的詳細資訊，請參閱建立即時 WebSocket 用戶端。

建立由無伺服器提供支援的一般發佈/訂閱 API WebSockets

有些應用程式只需要簡單的 WebSocket API，用戶端可監聽特定的通道或主題。可以將沒有特定形狀或強類型要求的通用 JSON 資料推送至以純粹且簡單的發佈-訂閱 (pub/sub) 模式監聽其中一個通道的用戶端。

透過在 API 後端和用AWS AppSync 戶端自動產生 GraphQL 程 WebSocket 式碼，可在幾分鐘內實作簡單的發佈/訂閱 API，幾分鐘內完全沒有 GraphQL 知識。

建立和設定發佈訂閱 API

若要開始使用，請執行下列動作：

1. 登入 AWS Management Console 並開啟 AppSync 主控台。
   - 在儀表板上，選擇 Create API (建立 API)。
2. 在下一個畫面中，選擇「建立即時 API」，然後選擇「下一步」。
3. 為您的發布/訂閱 API 輸入一個易記的名稱。
4. 您可以啟用私有 API 功能，但我們建議您暫時關閉此功能。選擇下一步。
5. 您可以選擇使用自動生成工作的發布/訂閱 API。 WebSockets我們建議您立即關閉此功能。選擇下一步。
6. 選擇「建立 API」，然後等待幾分鐘。將在您的帳戶中創建一個新的預配置發AWS AppSync 布/訂閱 API。AWS

該 API 使用內置AWS AppSync的本地解析器（有關使用本地解析器的更多信息，請參閱AWS AppSync 開發人員指南中的教程：本地解析器）來管理多個臨時發布/訂閱頻道和 WebSocket 連接，它們僅根據頻道名稱自動傳遞和過濾數據到訂閱的客戶端。API 呼叫是使用 API 金鑰進行授權的。

部署 API 之後，您會看到幾個額外的步驟來產生用戶端程式碼，並將其與用戶端應用程式整合。有關如何快速集成客戶端的示例，本指南將使用一個簡單的 React Web 應用程序。

1. 首先在本地計算機上使用 NPM 創建一個樣板 React 應用程序：

```
$ npx create-react-app mypubsub-app
$ cd mypubsub-app
```

> **Note**
>
> 此範例使用 Amplify 程式庫將用戶端連線至後端 API。但是，沒有必要在本地創建一個 Amplify CLI 項目。雖然 React 是這個範例中首選的用戶端，但 Amplify 程式庫也支援 iOS、Android 和 Flutter 用戶端，在這些不同的執行階段中提供相同的功能。支援的 Amplify 用戶端提供簡單的抽象化，可以透過幾行程式碼與 AWS AppSync GraphQL API 後端進行互動，包括內建 WebSocket功能與即時通訊協定完全相容：AWS AppSync WebSocket
>
> ```
> $ npm install @aws-amplify/api
> ```

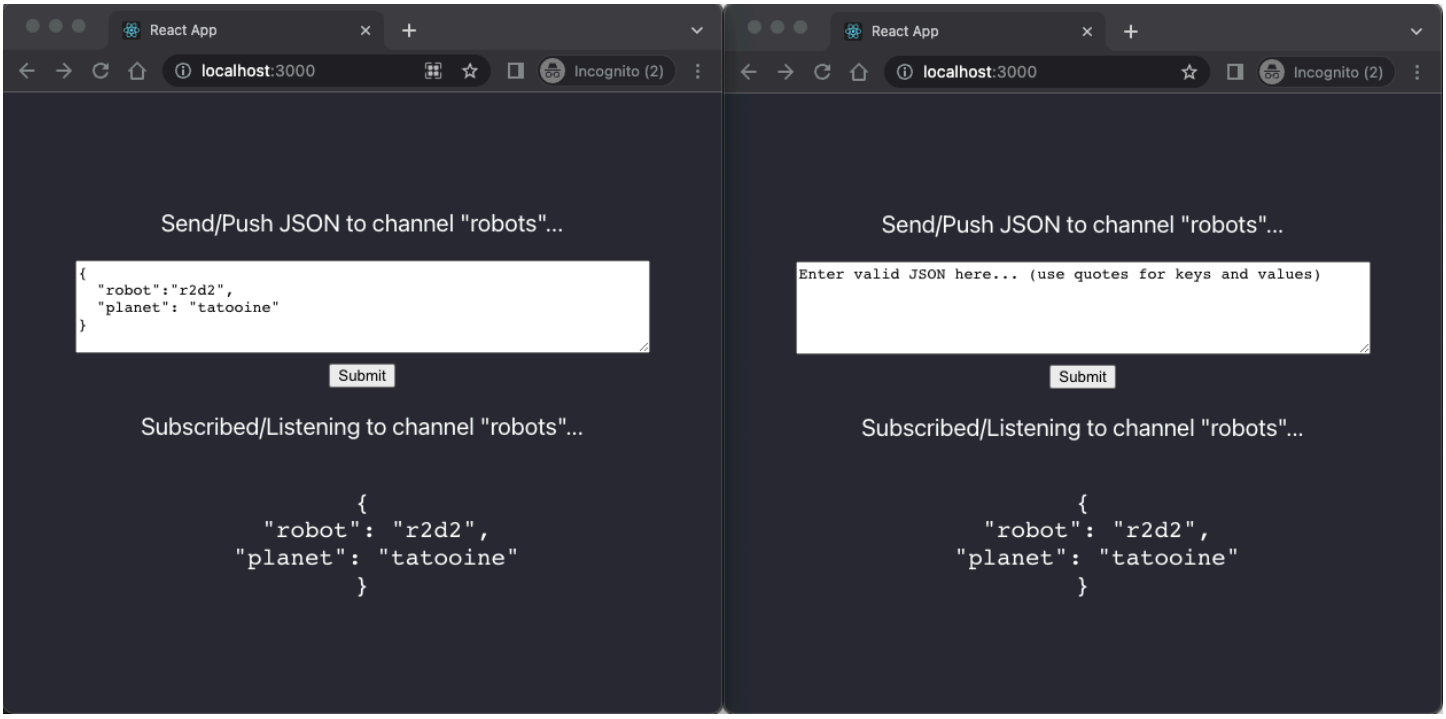
2. 在AWS AppSync 主控台中 JavaScript，選取「下載」以下載包含 API 設定詳細資訊和產生 GraphQL 作業程式碼的單一檔案。
3. 將下載的檔案複製到 React 專案中的`/src`資料夾。
4. 接下來，將現有樣板`src/App.js`檔案的內容取代為主控台中可用的範例用戶端程式碼。
5. 使用下列命令在本機啟動應用程式：

```
$ npm start
```

6. 若要測試傳送和接收即時資料，請開啟兩個瀏覽器視窗並存取 *`localhost:`* 3000。範例應用程式設定為將一般 JSON 資料傳送至名為 *`bot`* 的硬式編碼通道。
7. 在其中一個瀏覽器視窗中，在文字方塊中輸入下列 JSON blob，然後按一下「送出」：

```
{
  "robot":"r2d2",
  "planet": "tatooine"
}
```

兩個瀏覽器實例都訂閱了*`###`*頻道，並實時接收已發布的數據，顯示在 Web 應用程序的底部：



系統會自動產生所有必要的 GraphQL API 程式碼，包括結構描述、解析器和作業，以啟用一般的 pub/sub 使用案例。在後端，資料會發佈到具有 GraphQL 突變AWS AppSync的即時端點，如下所示：

```
mutation PublishData {
    publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
        data
        name
    }
}
```

訂閱者透過相關 GraphQL 訂閱存取傳送至特定臨時通道的已發佈資料：

```
subscription SubscribeToData {
    subscribe(name:"channel") {
        name
        data
    }
}
```

將發布訂閱 API 實施到現有的應用程序

如果您只需要在現有應用程序中實現實時功能，則可以將此通用的 pub/Sub API 配置輕鬆集成到任何應用程序或 API 技術中。雖然使用單一 API 端點透過 GraphQL 在單一網路呼叫中安全地存取、操作和

合併來自一或多個資料來源的資料具有優勢，但是不需要從頭開始轉換或重建現有的 REST 應用程式即可充分利用的AWS AppSync即時功能。例如，您可以在單獨的 API 端點中擁有現有的 CRUD 工作負載，客戶端將消息或事件從現有應用程序發送和接收到通用 Pub/sub API，僅用於實時和發布/訂閱目的。

增強的訂閱過濾

在中AWS AppSync，您可以使用支援其他邏輯運算子的篩選器，直接在 GraphQL API 訂閱解析器中定義和啟用後端資料篩選的商務邏輯。您可以設定這些篩選器，與用戶端中訂閱查詢上定義的訂閱引數不同。如需有關使用訂閱引數的詳細資訊，請參閱使用訂閱引數。如需運算子清單，請參閱解析程式對映範本公用程式參考。

針對本文件的目的，我們將即時資料篩選分為以下幾類：

- 基本篩選-根據訂閱查詢中用戶端定義的引數進行篩選。
- 增強過濾-根據在AWS AppSync服務後端集中定義的邏輯進行過濾。

下列各節說明如何設定增強型訂閱篩選器，並顯示其實際用途。

在您的 GraphQL 結構描述中定義訂閱

若要使用增強型訂閱篩選器，您可以在 GraphQL 結構描述中定義訂閱，然後使用篩選延伸模組定義增強型篩選器。若要說明增強型訂閱篩選的運作方式AWS AppSync，請使用下列 GraphQL 結構描述 (定義票證管理系統 API) 做為範例：

```
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
```

假設您為 API 創建了一個NONE數據源，然後使用此數據源將解析器附加到createTicket突變。您的處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**
>
> 在指定訂閱中，GraphQL 解析器的處理常式中會啟用增強型篩選器。如需詳細資訊，請參閱解析器參考。

若要實作增強型篩選器的行為，您必須使用`extensions.setSubscriptionFilter()`函數來定義篩選運算式，根據訂閱用戶端可能感興趣的 GraphQL 突變中的已發佈資料進行評估。如需篩選延伸模組的詳細資訊，請參閱延伸模組。

下一節說明如何使用篩選延伸模組來實作增強型篩選器。

使用篩選延伸模組建立增強的訂閱

增強型篩選器會在訂閱解析器的回應處理常式中以 JSON 撰寫。過濾器可以在名為 a 的列表中組合在一起`filterGroup`。篩選器至少使用一個規則定義，每個規則都有欄位、運算子和值。讓我們定義一個新的解析器`onSpecialTicketCreated`，用於設置一個增強的過濾器。您可以在使用 AND 邏輯評估的篩選器中設定多個規則，同時使用 OR 邏輯評估篩選群組中的多個篩選器：

```
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;
}
```

根據上述範例中定義的篩選器，如果使用下列項目建立票證，則會自動將重要票證推送至訂閱的 API 用戶端：

- priority水平high或 medium

  AND

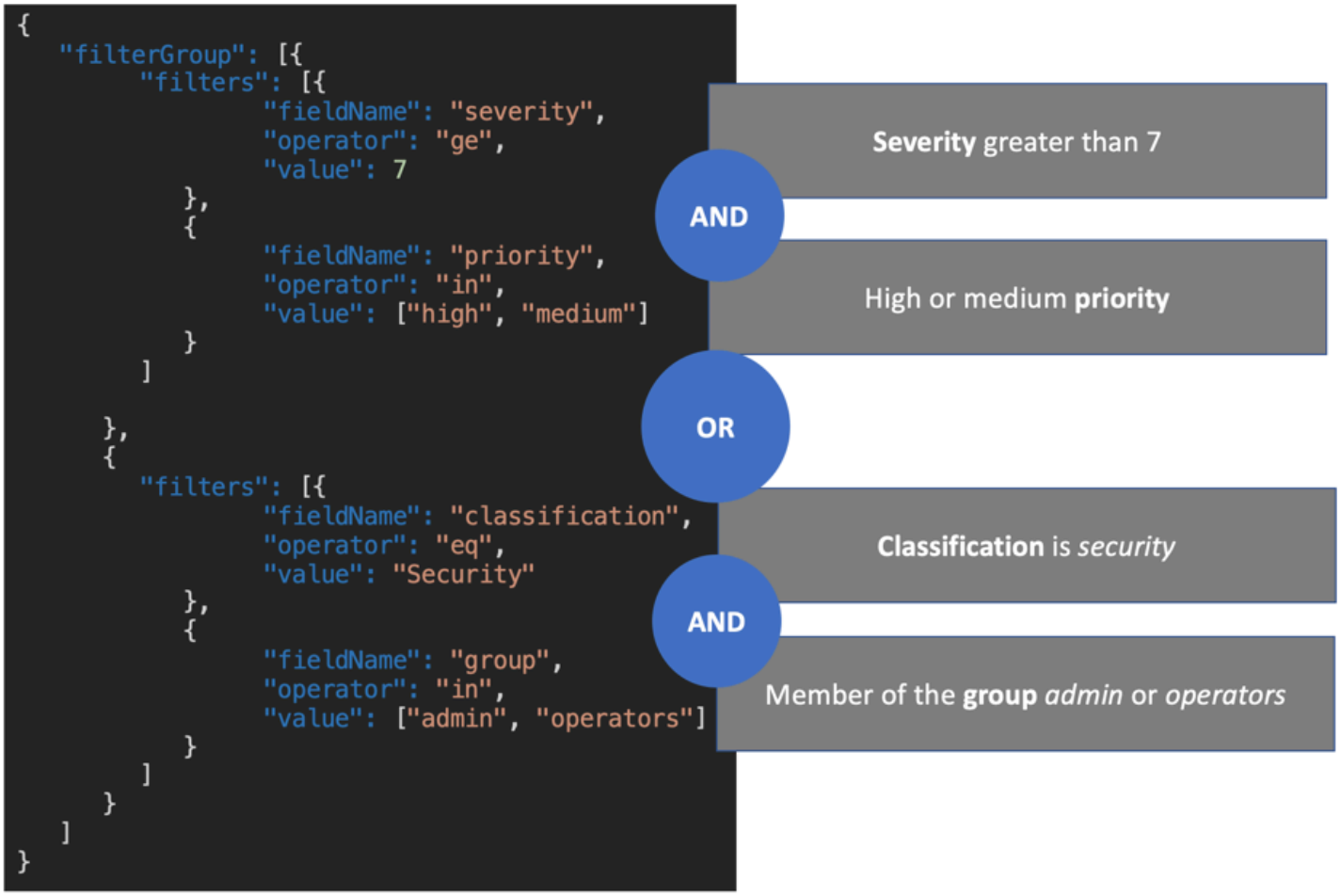
- severity層級大於或等於 7 (ge)

或

- classification票, 設置為 Security

  AND

- group指定集為admin或 operators



訂閱解析器 (增強型篩選) 中定義的篩選器優先於僅根據訂閱引數 (基本篩選) 的篩選。如需有關使用訂閱引數的詳細資訊，請參閱使用訂閱引數)。

如果在訂閱的 GraphQL 結構描述中定義並且需要引數，則僅當引數定義為解析器方法中的規則時，才會根據給定引數進行篩選。`extensions.setSubscriptionFilter()`但是，如果訂閱解析器中沒有`extensions`篩選方法，則用戶端中定義的引數僅用於基本篩選。您無法同時使用基本篩選和增強型篩選。

您可以使用訂閱的篩選延伸功能邏輯中的`context`變數來存取有關請求的內容資訊。例如，使用 Amazon Cognito 使用者集區、OIDC 或 Lambda 自訂授權器進行授權時，您可以在訂閱建立`context.identity`時擷取有關使用者的資訊。您可以使用該資訊根據使用者的身分建立篩選器。

現在假設您想要實作的增強型篩選器行為`onGroupTicketCreated`。`onGroupTicketCreated`訂閱需要強制性`group`名稱作為引數。建立時，工單會自動指派`pending`狀態。您可以設定訂閱篩選器，以僅接收屬於所提供群組的新建立工單：

```
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;
}
```

當使用突變發佈資料時，如下列範例所示：

```
mutation CreateTicket {
  createTicket(input: {priority: medium, severity: 2, group: "aws"}) {
    id
    priority
    severity
    status
    group
    createdAt
  }
}
```

訂閱的客戶端會在使用`createTicket`突變創建票證後立即監聽要自動推送的數據： WebSockets

```
subscription OnGroup {
  onGroupTicketCreated(group: "aws") {
    category
    status
    severity
    priority
    id
    group
    createdAt
    content
  }
}
```

因為篩選邏輯是透過增強型篩選功能來實作，因此可以在不含引數的AWS AppSync情況下訂閱用戶端，以簡化用戶端程式碼。只有在符合定義的篩選條件時，用戶端才會接收資料。

定義巢狀結構描述欄位的增強篩選

您可以使用增強的訂閱篩選來篩選巢狀結構描述欄位。假設我們修改了上一節中的結構描述，以包含位置和地址類型：

```
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
```

使用此模式，您可以使用分.隔符來表示嵌套。下列範例會在下為巢狀結構描述欄位新增篩選規則location.address.country。如果票證的地址設置為以下內容，則將觸發訂閱USA：

```
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;
}
```

在上面的範例中，location表示巢狀層級 1，address代表巢狀層級 2，並country代表巢狀層級三，所有這些都以分隔符號分.隔。

您可以通過使用createTicket突變來測試此訂閱：

```
mutation CreateTicketInUSA {
  createTicket(input: {location: "{\"address\":{\"country\":\"USA\"}}"}) {
    category
```

```
    content
    createdAt
    group
    id
    location {
      address {
        country
      }
    }
    priority
    severity
    status
  }
}
```

定義來自用戶端的增強過濾器

您可以在 GraphQL 中搭配訂閱引數使用基本篩選。在訂閱查詢中進行呼叫的用戶端會定義引數的值。在具有篩選的AWS AppSync訂閱解析器中啟用增強型篩選器時，在解析器中定義的後端`extensions`篩選器將具有優先順序和優先順序。

使用訂閱中的`filter`引數來設定用戶端定義的動態增強型篩選器。設定這些篩選器時，您必須更新 GraphQL 結構描述以反映新引數：

```
...
type Subscription {
    onSpecialTicketCreated(filter: String): Ticket
        @aws_subscribe(mutations: ["createTicket"])
}
...
```

然後，用戶端可以傳送訂閱查詢，如下列範例所示：

```
subscription onSpecialTicketCreated($filter: String) {
     onSpecialTicketCreated(filter: $filter) {
        id
        group
        description
        priority
        severity
     }
 }
```

您可以設定查詢變數，如下列範例所示：

```
{"filter" : "{\"severity\":{\"le\":2}}"}
```

您可以在訂閱回應對應範本中實作`util.transform.toSubscriptionFilter()`解析器公用程式，以針對每個用戶端套用在訂閱引數中定義的篩選器：

```
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;
}
```

透過此策略，用戶端可以定義自己的篩選器，以使用增強型篩選邏輯和其他運算子。當指定的用戶端呼叫安全 WebSocket 連線中的訂閱查詢時，就會指派篩選器。如需有關增強型篩選之轉換公用程式的詳細資訊，包括`filter`查詢變數承載的格式，請參閱JavaScript 解析器概觀。

其他增強的過濾限制

以下是一些針對增強型篩選器設定額外限制的使用案例：

- 增強型篩選器不支援頂層物件清單的篩選。在此使用案例中，增強型訂閱將忽略來自突變的已發佈資料。
- AWS AppSync 最多支援五個巢狀層級。超過巢狀層級 5 的結構描述欄位篩選器將會被忽略。採取下面的 GraphQL 響應。中的`continent`欄位`venue.address.country.metadata.continent`是允許的，因為它是五級巢穴。但是，`financial`in `venue.address.country.metadata.capital.financial` 是六級嵌套，因此過濾器不起作用：

```
{
    "data": {
        "onCreateFilterEvent": {
            "venue": {
```

```
                "address": {
                    "country": {
                        "metadata": {
                            "capital": {
                                "financial": "New York"
                            },
                            "continent" : "North America"
                        }
                    },
                    "state": "WA"
                },
                "builtYear": 2023
            },
            "private": false,
        }
    }
}
```

使用篩選器取消訂閱 WebSocket 連線

在中AWS AppSync，您可以根據特定篩選邏輯強制取消訂閱並關閉 WebSocket連線用戶端的連線 (無效)。這在授權相關案例中非常有用，例如當您從群組中移除使用者時。

訂閱失效是為了回應突變中定義的有效負載而發生。我們建議您將用於使訂閱連線失效的變異視為 API 中的系統管理作業，並相應地將權限範圍限制在管理員使用者、群組或後端服務。例如，使用結構描述授權指令，例如`@aws_auth(cognito_groups: ["Administrators"])`或`@aws_iam`。如需詳細資訊，請參閱使用其他授權模式。

無效驗證篩選器使用與增強型訂閱篩選器相同的語法和邏輯。使用下列公用程式定義這些篩選器：

- `extensions.invalidateSubscriptions()`— 在 GraphQL 解析器的突變回應處理常式中定義。
- `extensions.setSubscriptionInvalidationFilter()`— 在 GraphQL 解析器的連結至突變之訂閱的回應處理常式中定義。

如需有關無效驗證篩選延伸模組的詳細資訊，請參閱JavaScript 解析器概觀。

使用訂閱失效

若要查看訂閱失效的運作方式AWS AppSync，請使用下列 GraphQL 結構描述：

```
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突變解析器代碼中定義無效過濾器：

```
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 };
}
```

呼叫突變時，會使用payload物件中定義的資料來取消訂閱中subscriptionField定義的訂閱。onGroupMessageCreated訂閱的回應對應範本中也會定義無效驗證篩選器。

如果extensions.invalidateSubscriptions()承載包含與篩選器中定義的已訂閱用戶端 ID 相符的 ID，則會取消訂閱對應的訂閱。此外， WebSocket 連接已關閉。定義訂閱的訂閱解析程式碼：onGroupMessageCreated

```
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;
}
```

請注意，訂閱回應處理常式可同時定義訂閱篩選器和無效驗證篩選器。

例如，假設用戶端 A 使用下列訂閱要求，*user-1*將 ID 為具有 ID 的群組訂閱新*group-1*使用者：

```
onGroupMessageCreated(userId : "user-1", groupId: :"group-1"){...}
```

AWS AppSync執行訂閱解析程式，該解析程式會依照先前onGroupMessageCreated回應對應範本中的定義產生訂閱和無效驗證篩選器。對於用戶端 A，訂閱篩選器僅允許將資料傳送至*group-1*，而且會為*user-1*和*group-1*定義無效驗證篩選器。

現在假設用戶端 B 使用下列訂閱要求，*user-2*將具有 ID 的*group-2*使用者訂閱至具有 ID 的群組：

```
onGroupMessageCreated(userId : "user-2", groupId: :"group-2"){...}
```

AWS AppSync運行訂閱解析器，該解析器會生成訂閱和無效驗證過濾器。對於用戶端 B，訂閱篩選器只允許將資料傳送至*group-2*，而且會為*user-2*和*group-2*定義無效驗證篩選器。

接下來，假設使用突變請求創建具*message-1*有 ID 的新組消息，如以下示例所示：

```
createGroupMessage(id: "message-1", groupId :
      "group-1", message: "test message"){...}
```

已訂閱的用戶端符合定義的篩選器，會透過以下方式自動接收以 WebSockets下

```
{
  "data": {
    "onGroupMessageCreated": {
      "id": "message-1",
      "groupId": "group-1",
      "message": "test message",
    }
  }
}
```

用戶端 A 會收到郵件，因為篩選條件符合定義的訂閱篩選器。但是，用戶端 B 不會收到訊息，因為使用者不屬於*group-1*。此外，請求與訂閱解析器中定義的訂閱過濾器不匹配。

最後，假設從*group-1*使用以下突變請求中刪除：*user-1*

```
removeUserFromGroup(userId: "user-1", groupId : "group-1"){...}
```

突變啟動訂閱失效，如其`extensions.invalidateSubscriptions()`解析器響應處理程序代碼中定義的那樣。 AWS AppSync然後取消訂閱客戶端 A 並關閉其 WebSocket 連接。用戶端 B 不受影響，因為突變中定義的失效負載與其使用者或群組不相符。

使連線AWS AppSync無效時，用戶端會收到一則訊息，確認連線已取消訂閱：

```
{
  "message": "Subscription complete."
}
```

在訂閱失效篩選器中使用上下文變數

與增強的訂閱篩選器一樣，您可以使用訂閱失效篩選延伸模組中的`context`變數來存取特定資料。

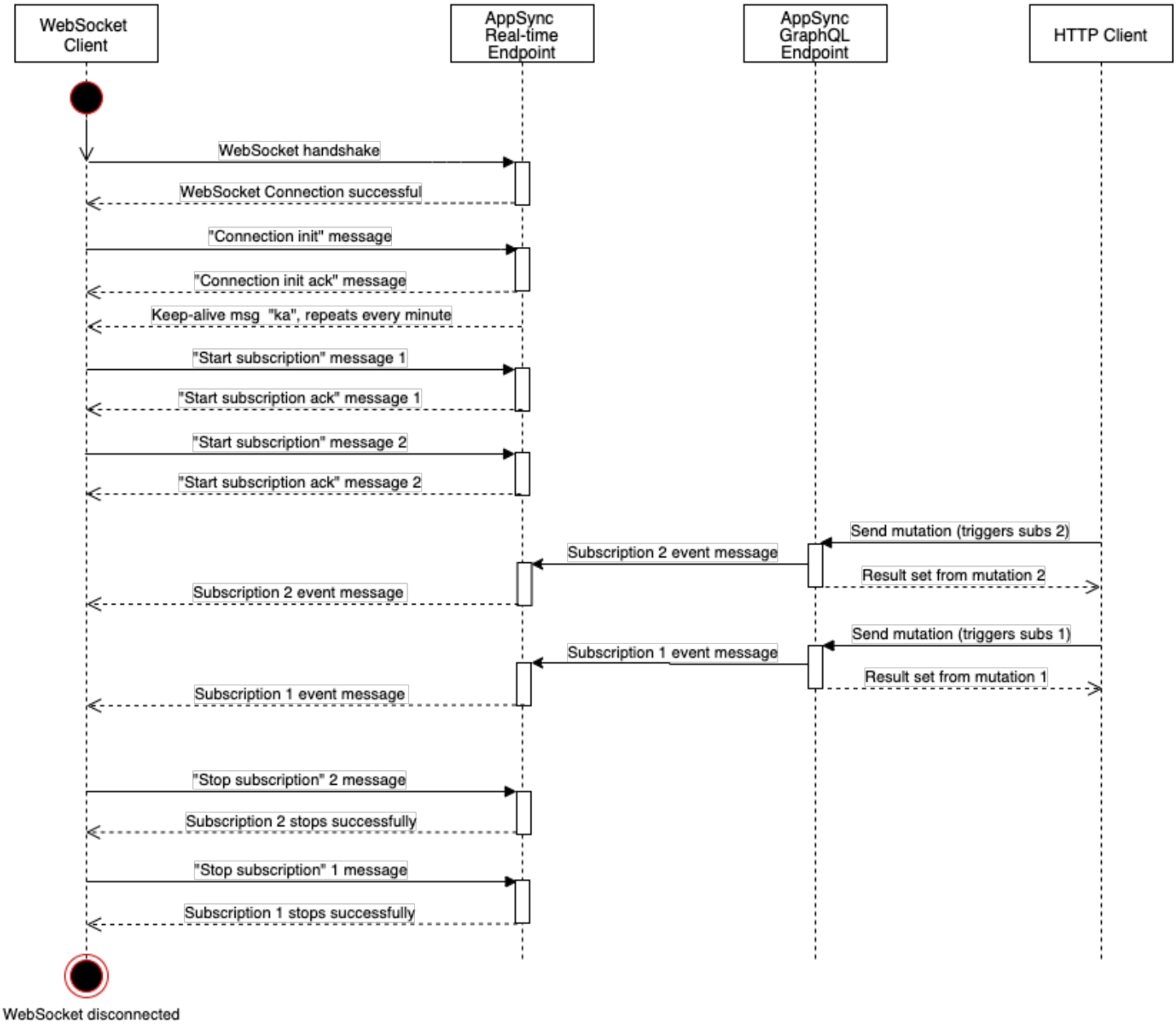
例如，可以將電子郵件地址設定為突變中的無效負載，然後將其與電子郵件屬性進行比對，或者從授權 Amazon Cognito 使用者集區或 OpenID Connect 的訂閱使用者宣告。`extensions.setSubscriptionInvalidationFilter()`訂閱無`extensions.invalidateSubscriptions()`效驗證器中定義的無效驗證過濾器會檢查突變的有效負載設置的電子郵件地址是否與從用戶的 JWT 令牌中`context.identity.claims.email`檢索到的電子郵件地址相匹配，從而啟動失效。

建立即時 WebSocket 用戶端

以下各節將向您展示實時功能背後AWS AppSync的架構。

GraphQL 訂閱的即時 WebSocket 用戶端實作

下列順序圖表和步驟顯示 WebSocket 用戶端、HTTP 用戶端和之間的即時訂閱工作流程AWS AppSync。



1. 用戶端與AWS AppSync 即時端點建立 WebSocket 連線。如果存在網路錯誤，用戶端應該執行抖動的指數輪詢。如需詳細資訊，請參閱架構部落格上的指數輪詢和抖動。AWS
2. 成功建立 WebSocket 連接後，客戶端發送一`connection_init`條消息。
3. 用戶端會等待來自AWS AppSync的`connection_ack`訊息。此訊息包含`connectionTimeoutMs`參數，這是 "ka" (保持活動) 訊息的等待時間上限 (以毫秒為單位)。
4. AWS AppSync 定期傳`"ka"`送訊息。客戶端跟踪它收到每`"ka"`條消息的時間。如果客戶端沒有在`connectionTimeoutMs`幾毫秒內收到`"ka"`消息，客戶端應該關閉連接。

5. 用戶端會透過傳送訂閱訊息來註冊 `start` 訂閱。單個 WebSocket 連接支持多個訂閱，即使它們處於不同的授權模式。
6. 用戶端會等待 AWS AppSync 傳送 `start_ack` 訊息以確認訂閱成功。如果發生錯誤，則會 AWS AppSync 傳回 `"type": "error"` 訊息。
7. 用戶端偵聽訂閱事件，這些事件會在呼叫對應的變異之後傳送。查詢和變動通常透過 `https://` 傳送到 AWS AppSync GraphQL 端點。訂閱會使用 secure WebSocket (`wss://`) 流經AWS AppSync 即時端點。
8. 用戶端會透過傳送訂閱訊息來取消註冊 `stop` 訂閱。
9. 取消註冊所有訂閱並檢查是否沒有透過傳輸的訊息之後 WebSocket，用戶端可以中斷 WebSocket 連線。

握手詳細信息以建立 WebSocket 連接

若要連線並啟動成功交握 AWS AppSync，用 WebSocket 戶端需要下列項目：

- AWS AppSync 即時端點
- 包含`header`和`payload`參數的查詢字串：
  - `header`：包含與 AWS AppSync 端點和授權相關的資訊。這是來自字串化 JSON 物件的 base64 編碼字串。JSON 對象內容取決於授權模式。
  - `payload`：以 BAS64 編碼的字串。`payload`

根據這些需求，用 WebSocket 戶端可以連線到 URL，該 URL 包含具有查詢字串的即時端點，並用`graphql-ws`作 WebSocket 通訊協定。

從 GraphQL 端點探索 即時端點

AWS AppSync GraphQL 端點和 AWS AppSync 即時端點在協議和網域中略有不同。您可以使用 AWS Command Line Interface (AWS CLI) 命令`aws appsync get-graphql-api`擷取 GraphQL 端點。

AWS AppSync GraphQL 端點：

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync 即時端點：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql
```

應用程式可以使用任何 HTTP 用戶端進行查詢和變動連線到 AWS AppSync GraphQL 端點 (`https://`)。應用程式可以使用任何用 WebSocket 戶端進行訂閱連線至AWS AppSync 即時端點 (`wss://`)。

使用自訂網域名稱，您可以使用單一網域與兩個端點互動。例如，如果您設定`api.example.com`為自訂網域，則可以使用下列 URL 與 GraphQL 和即時端點互動：

AWS AppSync 自訂網域 GraphQL 端點：

```
https://api.example.com/graphql
```

AWS AppSync 自訂網域即時端點：

```
wss://api.example.com/graphql/realtime
```

基於 AWS AppSync API 授權模式的標題參數格式

連接查詢字串中使用的`header`物件格式視 AWS AppSync API 授權模式而有所不同。在對象中的 `host` 欄位指的是 AWS AppSync GraphQL 端點，系統會用它來驗證，即使 `wss://` 呼叫是針對即時端點進行的連線。若要啟動交握並建立授權連線，`payload` 應該是空的 JSON 物件。

API 金鑰

API 金鑰標頭

標題內容

- `"host": <string>`：AWS AppSyncGraphQL 端點的主機或您的自訂網域名稱。
- `"x-api-key": <string>`：為 API 設定的 AWS AppSync API 金鑰。

範例

```
{
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com",
    "x-api-key":"da2-12345678901234567890123456"
}
```

有效載荷

```
{}
```

請求網址

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJob3N0IjoiZXhhbXBsZTEyMzQ1Njc4OTAwMDAuYXBwc3luYy1hcGkudXMtZWFzdC0xLmFtYXpvbmF3cy5jb20i
```

Amazon Cognito 使用者集區和 OpenID 連線 (OIDC)

亞馬遜認知和解決方案

標頭內容：

- `"Authorization": <string>`：一個 JWT ID 令牌。頭可以使用承載方案。
- `"host": <string>`：AWS AppSyncGraphQL 端點的主機或您的自訂網域名稱。

範例：

```
{

 "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"
}
```

承載內容：

```
{}
```

請求 URL：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

IAM

IAM 標頭

標題內容

- "accept": "application/json, text/javascript"：常數 <string> 參數。
- "content-encoding": "amz-1.0"：常數 <string> 參數。
- "content-type": "application/json; charset=UTF-8"：常數 <string> 參數。
- "host": <string>：這是 AWS AppSync GraphQL 端點的主機。
  - "x-amz-date": <string>: 時間戳記必須是世界標準時間，並採用下列 ISO 8601 格式：年月月日 'Z'。例如，20150830T123600Z 是有效的時間戳記。不包含時間戳記的毫秒數。如需詳細資訊，請參閱中的「處理簽名版本 4」中的日期AWS 一般參考。
  - "X-Amz-Security-Token": <string>：使用臨時安全憑據時需要的AWS會話令牌。如需詳細資訊，請參閱 IAM 使用者指南中的將臨時登入資料與AWS資源搭配使用。
  - "Authorization": <string>：AWS AppSync端點的簽章版本 4 (Sigv4) 簽署資訊。如需簽署程序的詳細資訊，請參閱工作 4：將簽章新增至 AWS 一般參考.

SigV4 簽署 HTTP 請求包含一個正式 URL，這是有 /connect 附加的 AWS AppSync GraphQL 端點。服務端點AWS區域與您使用 AWS AppSync API 的區域相同，服務名稱為「appsync」。要簽署的 HTTP 請求如下：

```
{
  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",
  }
}
```

範例

```
{
  "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"
}
```

有效載荷

```
{}
```

請求網址

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyEXAMPLEHQiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoE
```

若要使用自訂網域簽署要求：

```
{
  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",
```

```
  }
}
```

範例

```
{
  "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"
}
```

有效載荷

```
{}
```

請求網址

```
wss://api.example.com/graphql?
header=eyEXAMPLEHQiOiJhcHBsaWNhdGlvbi9qc29uLCB0ZXh0L2phdmFEXAMPLEQiLCJjb250ZW50LWVuY29kaW5nIjoE
```

> **Note**
>
> 一個 WebSocket 連線可以有多個訂閱 (即使使用不同的驗證模式)。實現這一點的一種方法是為第一個訂閱創建 WebSocket 連接，然後在最後一個訂閱未註冊時將其關閉。如果應用程序在最後一個訂閱未註冊後立即訂閱，則可以通過在關閉 WebSocket連接之前等待幾秒鐘來優化此功能。對於移動應用程序示例，當從一個屏幕切換到另一個屏幕時，卸載事件將停止訂閱，並在掛載事件時啟動不同的訂閱。

Lambda 授權

Lambda 權標頭

標題內容

- `"Authorization": <string>`：傳遞為的值`authorizationToken`。
- `"host": <string>`：AWS AppSyncGraphQL 端點的主機或您的自訂網域名稱。

範例

```
{

 "Authorization":"M0UzQzM1MkQtMkI0Ni00OTZCLUI1NkQtMUM0MTQ0QjVBRTczCkI1REEzRTIxLTk5NzItNDJENi1BQ
    "host":"example1234567890000.appsync-api.us-east-1.amazonaws.com"
}
```

有效載荷

```
{}
```

請求網址

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/graphql?
header=eyJBdXRob3JpemF0aW9uIjoiZXlKcmFXUWlPaUpqYkc1eGIzQTVlVzVNSzA5UVlYSXJNVEpIV0VGTFNYQmllVTVX
```

實時 WebSocket操作

在啟動成功 WebSocket 握手之後 AWS AppSync，用戶端必須傳送後續訊息以進行不同的AWS AppSync 作業連線。這些訊息需要下列資料：

- `type`：操作類型。
- `id`：訂閱的唯一識別碼。我們建議您使用 UUID 來達到此目的。
- `payload`：關聯的裝載，取決於操作類型。

此`type`欄位是唯一必填欄位；`id`和`payload`欄位為選擇性欄位。

事件順序

若要成功啟動、建立、註冊和處理訂閱要求，用戶端必須逐步執行下列順序：

1. 初始化連線 (`connection_init`)
2. 連線確認 (`connection_ack`)
3. 訂閱註冊 (`start`)
4. 訂閱確認 (`start_ack`)
5. 正在處理訂閱 (`data`)
6. 取消註冊訂閱 (`stop`)

連線初始化訊息

成功握手之後，用戶端必須傳送`connection_init`訊息以開始與AWS AppSync即時端點通訊。如果沒有此步驟，則忽略所有其他消息。該訊息是透過字串化以下 JSON 對象獲得的字串，如下所示：

```
{ "type": "connection_init" }
```

連線確認訊息

傳送 `connection_init` 訊息後，用戶端必須等待 `connection_ack` 訊息。接收前傳送的所有訊息`connection_ack`都會遭到忽略。該訊息應該如下所示：

```
{
```

```
  "type": "connection_ack",
  "payload": {
    // Time in milliseconds waiting for ka message before the client should terminate
 the WebSocket connection
    "connectionTimeoutMs": 300000
  }
}
```

保持啟用訊息

除了連線確認訊息之外，用戶端會定期接收保持作用中的訊息。如果用戶端在連線逾時期間內未收到保持連線的訊息，用戶端應該關閉連線。AWS AppSync 繼續發送這些消息並為註冊的訂閱提供服務，直到它自動關閉連接（24 小時後）。保持活動訊息是活動訊號，不需要用戶端對其進行認可。

```
{ "type": "ka" }
```

訂閱註冊訊息

用戶端收到connection_ack訊息後，用戶端可以將訂閱註冊訊息傳送至AWS AppSync。此類型的訊息是字串化的 JSON 物件，其中包含下列欄位：

- `"id": <string>`：訂閱的識別碼。此 ID 對於每個訂閱都必須是唯一的，否則伺服器會傳回錯誤，指出訂閱識別碼已重複。
- `"type": "start"`：常數 `<string>` 參數。
- `"payload": <Object>`：包含與訂閱相關資訊的物件。
  - `"data": <string>`：包含 GraphQL 查詢和變數的字串化 JSON 物件。
    - `"query": <string>`：一個 GraphQL 作業。
    - `"variables": <Object>`：包含查詢變數的物件。
  - `"extensions": <Object>`：包含授權物件的物件。
- `"authorization": <Object>`：包含授權所需字段的對象。

訂閱註冊的授權物件

該[基於 AWS AppSync API 授權模式的標題參數格式](#)部分中的相同規則適用於授權對象。唯一的例外是 IAM，其中 Sigv4 簽名信息略有不同。如需詳細資訊，請參閱 IAM 範例。

使用 Amazon Cognito 使用者集區的範例：

```
{
  "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 的範例：

```
{
  "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"
}
```

使用自訂網域名稱的範例：

```
{
  "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"
}
```

不需要將 Sigv4 簽章附加/connect至網址，而且會取代 JSON 字串化的 GraphQL 作業。data以下是 Sigv4 簽名請求的範例：

```
{
  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",
  }
}
```

訂閱確認訊息

發送訂閱開始消息後，客戶端應等AWS AppSync待發送start_ack消息。該start_ack消息表明訂閱成功。

訂閱確認範例：

```
{
  "type": "start_ack",
  "id": "eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

錯誤訊息

如果連線 init 或訂閱註冊失敗，或者訂閱從伺服器結束，伺服器會傳送錯誤訊息給用戶端：

- `"type": "error"`：常數 <string> 參數。
- `"id": <string>`：對應註冊訂閱的 ID (如果相關)。
- `"payload" <Object>`：包含對應錯誤資訊的物件。

範例：

```
{
  "type": "error",
  "payload": {
    "errors": [
      {
        "errorType": "LimitExceededError",
        "message": "Rate limit exceeded"
      }
    ]
  }
}
```

處理資料訊息

當客戶端提交突變時，AWS AppSync 識別所有感興趣的用戶，並使用訂閱操作中的相應訂閱向每個訂閱id者發送"type":"data"消息。"start"用戶端預期會追蹤其傳送的訂閱id，以便在收到資料訊息時，用戶端可以將其與對應的訂閱進行比對。

- `"type": "data"`：常數 <string> 參數。
- `"id": <string>`：對應註冊訂閱的 ID。
- `"payload" <Object>`：包含訂閱資訊的物件。

範例：

```
{
  "type": "data",
  "id": "ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69",
  "payload": {
    "data": {
      "onCreateMessage": {
        "__typename": "Message",
        "message": "test"
      }
    }
  }
}
```

訂閱取消註冊訊息

當應用程序想要停止監聽訂閱事件時，客戶端應該發送帶有以下字符串化 JSON 對象的消息：

- `"type": "stop"`：常數 <string> 參數。
- `"id": <string>`：要取消註冊的訂閱識別碼。

範例：

```
{
  "type":"stop",
  "id":"ee849ef0-cf23-4cb8-9fcb-152ae4fd1e69"
}
```

AWS AppSync 傳回含下列字串化 JSON 物件的確認訊息：

- `"type": "complete"`：常數 `<string>` 參數。
- `"id": <string>`：未註冊訂閱的 ID。

用戶端收到確認訊息後，就不會再收到此特定訂閱的訊息。

範例：

```
{
  "type":"complete",
  "id":"eEXAMPLE-cf23-1234-5678-152EXAMPLE69"
}
```

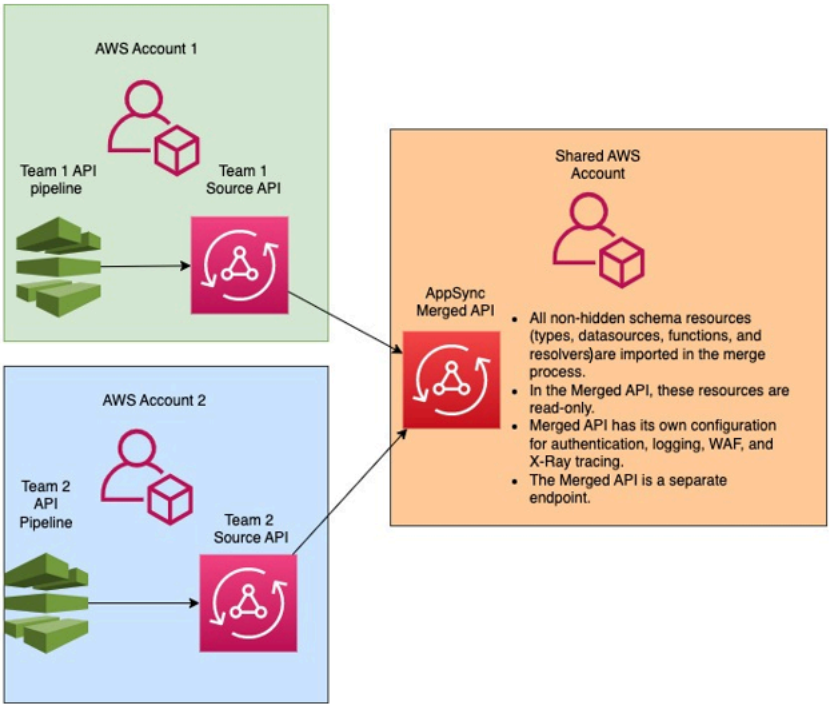
斷開連接 WebSocket

中斷連線之前，為避免資料遺失，用戶端應具備必要的邏輯，以檢查目前沒有透過 WebSocket連線執行的作業。在中斷連線之前，所有訂閱都應取消註冊。 WebSocket

已合併的 API

隨著 GraphQL 在組織內的使用擴展，可能會在 API ease-of-use 和 API 開發速度之間取捨。一方面，組織採AWS AppSync用 GraphQL 來簡化應用程式開發，方法是為開發人員提供彈性的 API，讓他們透過單一網路呼叫，安全地存取、操作和合併來自一或多個資料網域的資料。另一方面，組織內負責將不同資料網域合併為單一 GraphQL API 端點的團隊可能希望能夠彼此獨立建立、管理和部署 API 更新，以提高其開發速度。

為了解決這種緊張問題，AWS AppSync合併的 API 功能允許來自不同資料網域的團隊獨立建立和部署 AWS AppSync API (例如 GraphQL 結構描述、解析器、資料來源和函數)，然後將這些 API 合併為單一的合併 API。這使組織能夠維護簡單易用的跨域 API，並為不同團隊提供一種方式，這些團隊有助於該 API 快速且獨立地進行 API 更新的能力。



使用合併 API，組織可以將多個獨立來源 AWS AppSync API 的資源匯入單一AWS AppSync合併的 API 端點。為此，AWS AppSync允許您創建源代碼 API 的列表，然後將與AWS AppSync源 API（包括模式，類型，數據源，解析器和函數）關聯的所有元數據合併到新的合併 API 中。AWS AppSync

在合併期間，可能會因為來源 API 資料內容 (例如組合多個結構描述時類型命名衝突) 的不一致而發生合併衝突。對於來源 API 中沒有任何定義衝突的簡單使用案例，不需要修改來源 API 結構描述。生成的合併 API 只是從原始源 API 導入所有類型，解析器，數據源AWS AppSync和函數。對於發生衝突的複雜用例，用戶/團隊將不得不通過各種方式解決衝突。 AWS AppSync為使用者提供數種可減少合併衝突的工具和範例。

在中設定的後續合併AWS AppSync會將來源 API 中所做的變更傳播至關聯的合併 API。

合併的 API 和同盟

GraphQL 社群中有許多解決方案和模式，可用於結合 GraphQL 結構描述，並透過共用圖形實現團隊協同作業。 AWS AppSync合併的 API 採用構建時間方法來構成結構描述，其中源 API 合併為單獨的合併 API。另一種方法是跨多個源 API 或子圖層運行時路由器。在這種方法中，路由器會接收要求、參考它維護為中繼資料的組合結構描述、建構要求計劃，然後將要求元素散佈至其基礎子圖形/伺服器。下表將合AWS AppSync併的 API 建置階段方法與 GraphQL 結構描述構成的路由器型執行階段方法進行比較：

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 合併的 API 只能與AWS AppSync來源 API 建立關聯。如果您需要跨越AWS AppSync和非AWS AppSync子圖形的結構描述構成支援，您可以將一或多個 AWS AppSync GraphQL 和/或合併的API 連接到路由器型解決方案中。例如，請參閱參考部落格，瞭解如AWS AppSync何使用路由器架構搭配阿波羅聯合 v2：阿波羅 Graph QL 聯合使用. AWS AppSync

主題

合併的 API 衝突解決

發生合併衝突時，會為使用者AWS AppSync提供數種工具和範例，以協助疑難排解問題。

合併的 API 架構指令

AWS AppSync引入了幾個 GraphQL 指令，這些指令可用於減少或解決來源 API 之間的衝突：

- @canonical：此指令設置具有相似名稱和數據的類型/字段的優先級。如果兩個或多個源 API 具有相同的 GraphQL 類型或字段，則其中一個 API 可以將其類型或字段註釋為標準，這些 API 將在合併期間優先級。合併時，會忽略其他來源 API 中未使用此指示詞註解的衝突類型/欄位。
- @hidden：此指令封裝某些類型/字段以將其從合併過程中刪除。團隊可能想要移除或隱藏來源 API 中的特定類型或作業，以便只有內部用戶端可以存取特定類型的資料。附加此指令後，類型或欄位不會合併到合併的 API 中。
- @renamed：此指令更改類型/字段的名稱以減少命名衝突。在某些情況下，不同的 API 具有相同的類型或字段名稱。但是，它們都需要在合併的模式中可用。將它們全部包含在合併的 API 中的一種簡單方法是將字段重命名為類似但不同的字段。

要顯示實用程序模式指令提供，請考慮以下示例：

在此範例中，假設我們想要合併兩個來源 API。我們給出了兩個模式來創建和檢索帖子（例如，評論部分或社交媒體帖子）。假設類型和字段非常相似，在合併操作期間發生衝突的可能性很高。下面的片段顯示了每個模式的類型和字段。

第一個文件，稱為源 1.GraphQL，是一個 Gra phQL 模式，允許用戶使用突變創建。Posts putPost每個都Post包含一個標題和一個 ID。ID 用於引用User，或海報的信息（電子郵件和地址），以及Message，或有效負載（內容）。User類型會以 @canonical 標籤加上註解。

```
# 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
}
```

第二個文件，稱為源 2. 圖形 SQL，是一個 GraphQL 模式，它做非常相似的事情來源 1.GraphQL。但是，請注意，每種類型的欄位都不同。合併這兩個結構描述時，會因為這些差異而產生合併衝突。

還要注意源 2.GraphQL 還包含幾個指令來減少這些衝突。該Post類型用 @hidden 標籤註釋，以便在合併操作期間混淆自己。Message類型會以 @renamed 標籤註解，以便ChatMessage在與其他類型發生命名衝突時將Message類型名稱修改為。

```
# 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")
}
```

當合併發生時，結果將產生文MergedSchema.graphql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
}
```

合併中發生了幾件事情：

- 由於 @canonical 註釋，源 1.GraphQL 中的*User*類型優先於來源 2. 圖形 QL 的User優先級。
- 來自源 1.GraphQL 的Message類型已包含在合併中。但是，來*Message*自源 2.GraphQL 存在命名衝突。由於其 @renamed 註釋，它也包含在合併中，但具有替代名稱ChatMessage。
- 來自源 1.GraphQL 的*Post*類型已包含在內，但來自源 2. 圖形 QL 的*Post*類型不是。通常情況下，這種類型會發生衝突，但是由於 Source2.GraphQL 中的Post類型具有 @hidden 註釋，因此其數據被模糊化並且不包含在合併中。這會導致沒有衝突。
- Query類型已更新，以包含兩個檔案中的內容。但是，由於該指令的GetChatMessage原因，有一個GetMessage查詢被重新命名。這解決了具有相同名稱的兩個查詢之間的命名衝突。

還有沒有指令被添加到衝突的類型的情況。在這裡，合併的類型將包括該類型的所有源定義中所有字段的聯集。例如，請考量下列範例：

這個結構描述，稱為源 1.GraphQL，允許創建和檢索。Posts組態與前面的範例類似，但資訊較少。

```
# 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
}
```

這個模式稱為 Source2.GraphQL，允許創建和檢索Reviews（例如，電影評分或餐廳評論）。Reviews與相同Post的 ID 值相關聯。它們共同包含完整評論帖子的標題，帖子 ID 和有效負載消息。

合併時，兩Post種類型之間會發生衝突。因為沒有註釋可以解決此問題，因此預設行為是對衝突的類型執行聯集操作。

```
# 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
}
```

當合併發生時，結果將產生文MergedSchema.graphql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
}
```

合併中發生了幾件事情：

- 該Mutation類型沒有衝突並被合併。
- Post類型字段通過聯集操作進行組合。請注意兩者之間的聯集如何產生單個 idtitle，a 和單個reviews。
- 該Review類型沒有衝突並被合併。
- 該Query類型沒有衝突並被合併。

管理共用類型的解析器

在上述範例中，請考慮來源 1.GraphQL 已在上設定單位解析器的情況Query.getPost，該解析器使用名為的 DynamoDB 資料來源。PostDatasource這個Post解析器將返回一個類型title的id和。現在，考慮源 2.GraphQL 已經配置了一個管道解析器，它運行兩個函數。Post.reviews

Function1具有附加的None資料來源以執行自訂授權檢查。 Function2具有附加的 DynamoDB 資料來源可供查詢資料表reviews。

```
query GetPostQuery {
    getPost(id: "1") {
        id,
        title,
        reviews
    }
}
```

當用戶端對合併的 API 端點執行上述查詢時，AWS AppSync服務會先執行來源的單位解析程式Source1，然後Query.getPost從 DynamoDB 呼叫PostDatasource並傳回資料。然後，它會執行Post.reviews管線解析程式，其中Function1執行自訂授權邏輯，並Function2傳回中id找到的檢閱。$context.source服務會以單一 GraphQL 執行的方式處理要求，而且這個簡單的要求只需要單一要求權杖。

管理共用類型的解析程式衝突

考慮下面的情況下，我們還實現了一個解析器，以便Query.getPost在一次超出字段解析器中提供多個字段。Source2源 1.GraphQL 可能看起來像這樣：

```
# This snippet represents a file called Source1.graphql

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

type Query {
    getPost(id: ID!): Post
}
```

源 2. 圖形 QL 可能看起來像這樣：

```
# This snippet represents a file called Source2.graphql

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

```
  contentHash: String!
  author: String!
}

type Query {
    getPost(id: ID!): Post
}
```

嘗試合併這兩個結構描述將產生合併錯誤，因為合AWS AppSync併的 API 不允許將多個源解析器附加到同一個字段。為了解決這種衝突，您可以實現一個字段解析器模式，該模式需要 S ource2.GraphQL 添加一個單獨的類型，該類型將定義它擁有的類型的字段。Post在下面的例子中，我們添加了一個名為的類型PostInfo，其中包含將由 Source 2.GraphQL 解析的內容和作者字段。S@@ ource1.graphQL 將實現附加到的解析器Query.getPost，而 S ource2.GraphQL 現在將附加一個解析器，以確保可以成功檢索所有數據：Post.postInfo

```
type Post  {
  id: ID!
  postInfo: PostInfo
}

type PostInfo {
   content: String!
   contentHash: String!
   author: String!
}

type Query {
    getPost(id: ID!): Post
}
```

雖然解決此類衝突需要重寫源 API 結構描述，並且可能客戶更改其查詢，但這種方法的優點是，合併解析器的所有權在於跨來源團隊保持清晰。

配置模式

雙方負責配置結構描述以創建合併的 API：

- 合併的 API 擁有者-合併的 API 擁有者必須配置合併 API 的授權邏輯和進階設定，例如記錄、追蹤、快取和 WAF 支援。
- 關聯的來源 API 擁有者-相關聯的 API 擁有者必須設定組成合併 API 的結構描述、解析器和資料來源。

由於合併 API 的結構描述是從關聯來源 API 的結構描述建立的，因此它是唯讀的。這表示您必須在來源 API 中啟動結構描述的變更。在AWS AppSync主控台中，您可以使用 [結構描述] 視窗上方的下拉式清單，在 [已合併的結構描述] 和 [已合併 API] 中包含的來源 API 的個別結構描述之間切換。

設定授權模式

多種授權模式可用於保護您的合併 API。若要深入瞭解中的授權模式AWS AppSync，請參閱授權和驗證。

下列授權模式可與合併的 API 搭配使用：

- API 密鑰：最簡單的授權策略。所有請求都必須在請`x-api-key`求標題下包含一個 API 密鑰。過期的 API 金鑰會在到期日期後保留 60 天。
- AWSIdentity and Access Management (IAM)：AWSIAM 授權策略會授權所有 sigv4 簽署的請求。
- Amazon Cognito 使用者集區：透過 Amazon Cognito 使用者集區授權您的使用者，以實現更精細的控制。
- AWSLambda 授權者：一種無伺服器函數，可讓您使用自訂邏輯對 AWS AppSync API 進行驗證和授權存取。
- OpenID Connect：此授權類型強制執行 OIDC 兼容服務提供的 OpenID 連接（OIDC）令牌。您的應用程式可以運用由 OIDC 提供者定義的使用者與權限來控制存取權限。

合併 API 的授權模式由合併的 API 擁有者設定。在合併作業時，「已合併的 API」必須包含在來源 API 上設定的主要授權模式，作為其本身的主要授權模式或次要授權模式。否則，它將不兼容，並且合併操作將失敗並發生衝突。在源 API 中使用多重身份驗證指令時，合併過程能夠將這些指令自動合併到統一端點中。在源 API 的主授權模式與合併 API 的主授權模式不匹配的情況下，它將自動添加這些身份驗證指令，以確保源 API 中類型的授權模式一致。

設定執行角色

建立合併的 API 時，您需要定義服務角色。AWS服務角色是 I AWS dentity and Access Management (IAM) 角色，AWS服務會用來代表您執行工作。

在這種情況下，您的合併 API 必須運行解析器，以訪問源 API 中配置的數據源中的數據。必要的服務角色是`mergedApiExecutionRole`，它必須具有明確存取權，才能透過 `appsync:SourceGraphQL` IAM 許可在合併的 API 中包含的來源 API 上執行請求。在 GraphQL 要求執行期間，AWS AppSync服務會擔任此服務角色，並授權角色執行`appsync:SourceGraphQL`動作。

AWS AppSync支援在請求中的特定頂層欄位上允許或拒絕此權限，例如 IAM 授權模式對 IAM API 的運作方式。對於 non-top-level欄位，AWS AppSync需要您定義來源 API ARN 本身的權限。為了限制對合併 API 中特定 non-top-level 欄位的存取，我們建議您在 Lambda 中實作自訂邏輯，或使用 @hidden 指示詞在合併 API 中隱藏來源 API 欄位。如果您想要允許角色執行來源 API 中的所有資料作業，您可以在下方新增政策。請注意，第一個資源項目允許訪問所有頂級字段，第二個條目涵蓋對源 API 資源本身授權的子解析器：

```
{
    "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"]
    }]
}
```

如果您想限制只有特定頂層欄位的存取權限，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主控台 API 建立精靈來產生服務角色，以允許合併的 API 存取與合併 API 位於相同帳戶的來源 API 中設定的資源。如果您的來源 API 與合併的 API 不在相同的帳戶中，您必須先使用 AWS Resource Access Manager (AWS RAM) 共用資源。

使用設定跨帳戶合併 API AWS RAM

建立合併的 API 時，您可以選擇性地關聯已透過 AWS Resource Access Manager (AWS RAM) 共用的其他帳戶的來源 API。 AWS RAM協助您跨AWS帳戶、組織或組織單位 (OU) 內，以及 IAM 角色和使用者安全地共用資源。

AWS AppSync與AWS RAM整合，以支援透過單一合併 API 跨多個帳戶設定和存取來源 API。 AWS RAM可讓您建立資源共用或資源容器，以及將為每個資源共用的權限集。您可以將 AWS AppSync API 新增至中的資源共用AWS RAM。在資源共用中，AWS AppSync提供三個可與 RAM 中的 AWS AppSync API 相關聯的不同權限集：

1. `AWSRAMPermissionAppSyncSourceApiOperationAccess`：如果未指定其他權限，AWS RAM則在共享 AWS AppSync API 時添加的默認權限集。此權限集用於與合併的 AWS AppSync API 擁有者共用來源 API。此權限集包括來源 API `appsync:AssociateMergedGraphqlApi` 的權限，以及在執行階段存取來源 API 資源所需的`appsync:SourceGraphQL`權限。
2. `AWSRAMPermissionAppSyncMergedApiOperationAccess`：與來源 API 擁有者共用合併的 API 時，應設定此權限集。此權限集將使來源 API 能夠設定合併的 API，包括將目標主體擁有的任何來源 API 與合併 API 相關聯，以及讀取和更新合併 API 的來源 API 關聯。
3. `AWSRAMPermissionAppSyncAllowSourceGraphQLAccess`：此權限集允許`appsync:SourceGraphQL`權限與 AWS AppSync API 搭配使用。它旨在用於與合併的 API 所有者共享源 API。與為來源 API 作業存取設定的預設權限不同，此權限集僅包含執行階段權限`appsync:SourceGraphQL`。如果使用者選擇將「已合併的 API」作業存取權共用給來源 API 擁有者，他們還需要將來源 API 的此權限共用給「已合併的 API」擁有者，才能透過「合併的 API」端點擁有執行階段存取權。

AWS AppSync也支援客戶管理的權限。當其中一個提供的AWS受管理權限無法運作時，您可以建立自己的客戶管理權限。客戶管理的權限是您編寫和維護的受管理權限，方法是精確指定可以在哪些情況下與使用共用AWS RAM資源執行的動作。 AWS AppSync可讓您在建立自己的權限時從下列動作中進行選擇：

1. `appsync:AssociateSourceGraphqlApi`
2. `appsync:AssociateMergedGraphqlApi`
3. `appsync:GetSourceApiAssociation`
4. `appsync:UpdateSourceApiAssociation`
5. `appsync:StartSchemaMerge`
6. `appsync:ListTypesByAssociation`
7. `appsync:SourceGraphQL`

在中正確共用來源 API 或合併 API 之AWS RAM後，如有必要，資源共用邀請已被接受，當您在合併的 API 上建立或更新來源 API 關聯時，該邀請將會顯示在AWS AppSync主控台中。您也可以透過呼叫

所提供的ListGraphqlApis作業AWS AppSync並使用擁AWS RAM有OTHER_ACCOUNTS者篩選器，列出所有使用您帳戶共用的 AWS AppSync API，而不論權限設定為何。

> **Note**
>
> 透過AWS RAM共用時，呼叫者AWS RAM必須擁有對正在共用的任何 API 執行appsync:PutResourcePolicy動作的權限。

合併

管理合併

合併的 API 旨在支持統一AWS AppSync端點上的團隊協作。團隊可以在後端獨立發展自己的獨立來源 GraphQL API，同時該AWS AppSync服務將資源整合到單一合併 API 端點中，以減少協同作業的摩擦並縮短開發前置時間。

自動合併

您可以將與合AWS AppSync併 API 相關聯的來源 API 設定為在來源 API 進行任何變更後，自動合併 (自動合併) 到合併的 API。這樣可確保來自來源 API 的變更始終在背景傳播到合併的 API 端點。只要來源 API 結構描述中的任何變更不會與合併 API 中的現有定義產生合併衝突，就會在「合併的 API」中更新。如果來源 API 中的更新正在更新解析器、資料來源或函數，則匯入的資源也會更新。引入無法自動解決 (自動解決) 的新衝突時，由於合併作業期間發生不支援的衝突，導致「已合併的 API 結構描述更新」遭到拒絕。對於每個狀態為的來源 API 關聯，主控台中都會顯示錯誤訊息MERGE_FAILED。您還可以通過使用 AWS SDK 調用給定源 API 關聯的GetSourceApiAssociation操作或使用 AWS CLI 來檢查錯誤消息，如下所示：

```
aws appsync get-source-api-association --merged-api-identifier <Merged API ARN> --
association-id <SourceApiAssociation id>
```

這將產生以下格式的結果：

```
{
    "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."
    }
}
```

手動合併

來源 API 的預設設定為手動合併。若要合併自合併 API 上次更新後在來源 API 中發生的任何變更，來源 API 擁有者可以從AWS AppSync主控台或透過 AWS SDK 和 AWS CLI 中可用的`StartSchemaMerge`作業叫用手動合併。

對合併 API 的其他支援

設定訂閱

與以路由器為基礎的 GraphQL 結構描述構成方法不同，AWS AppSync合併的 API 提供 GraphQL 訂閱的內建支援。在您關聯的來源 API 中定義的所有訂閱作業都會自動合併並在您的合併 API 中運作，而無需修改。若要深入瞭解如何透過無伺服器 WebSockets 連線AWS AppSync支援訂閱，請參閱即時資料。

配置可觀測性

AWS AppSync合併的 API 透過 Amazon 提供內建的記錄、監控和指標 CloudWatch。 AWS AppSync還提供內置支持跟踪通過 AWS X-Ray.

設定自訂網域

AWS AppSync合併的 API 提供內建支援，可將自訂網域與合併 API 的 GraphQL 和即時端點搭配使用。

配置緩存

AWS AppSync合併的 API 提供內建支援，可選擇快取要求層級和/或解析程式層級的回應，以及回應壓縮。若要深入瞭解，請參閱快取和壓縮。

設定私有 API

AWS AppSync合併的 API 為私有 API 提供內建支援，可限制對來自您可設定之 VPC 端點的流量存取合併 API 的 GraphQL 和即時端點的流量。

設定防火牆規則

AWS AppSync合併的 API 提供的內建支援AWS WAF，可讓您透過定義 Web 應用程式防火牆規則來保護 API。

設定稽核記錄

AWS AppSync合併的 API 提供內建支援 AWS CloudTrail，可讓您設定和管理稽核記錄。

合併的 API 限制

開發合併的 API 時，請注意以下規則：

1. 合併的 API 不能是另一個合併 API 的來源 API。
2. 來源 API 無法與一個以上的合併 API 相關聯。
3. 「合併的 API 結構描述」文件的預設大小限制為 10 MB。
4. 可與合併 API 相關聯的來源 API 預設數量為 10。不過，如果您在合併的 API 中需要 10 個以上的來源 API，則可以要求提高限制。

建立合併的 API

在主控台中建立合併的 API

1. 登入 AWS Management Console 並開啟 AWS AppSync主控台。
   - 在儀表板中，選擇「建立 API」。
2. 選擇合併的 API，然後選擇下一步。
3. 在「指定 API 詳細資訊」頁面中，輸入下列資訊：
   a. 在「API 詳細資訊」下，輸入下列資訊：
      i. 指定合併的 API 名稱。此欄位是為 GraphQL API 加上標籤的一種方式，以便於將其與其他 GraphQL API 區分開來。

ii. 指定聯絡人詳細資料。此欄位為選擇性欄位，並將名稱或群組附加至 GraphQL API。它不會連結至其他資源或由其他資源產生，而且運作方式與 API 名稱欄位非常相似。

b. 在「服務角色」下，您必須將 IAM 執行角色附加到合併的 API，AWS AppSync以便在執行時期安全地匯入和使用您的資源。您可以選擇建立和使用新的服務角色，這將允許您指定AWS AppSync將使用的策略和資源。您也可以選擇使用現有的服務角色，然後從下拉式清單中選取角色，以匯入現有的 IAM 角色。

c. 在私人 API 配置下，您可以選擇啟用私有 API 功能。請注意，建立合併的 API 後，無法變更此選項。如需有關私有 API 的詳細資訊，請參閱使用AWS AppSync私有 API。

完成後，請選擇 [下一步]。

4. 接下來，您必須新增 GraphQL API，這些 API 將用作合併 API 的基礎。在 [選取來源 API] 頁面中，輸入下列資訊：

a. 在AWS帳戶表格中的 API 中，選擇「新增來源 API」。在 GraphQL API 清單中，每個項目都會包含下列資料：

i. 名稱：圖 GraphQL API 的 API 名稱欄位。

ii. API 識別碼：圖 GraphQL API 的唯一識別碼值。

iii. 主要驗證模式：GraphQL API 的預設授權模式。如需中授權模式的詳細資訊AWS AppSync，請參閱授權與驗證。

iv. 額外的驗證模式：在 GraphQL API 中設定的次要授權模式。

v. 選取 API 名稱欄位旁邊的核取方塊，選擇您要在合併 API 中使用的 API。然後，選擇添加源 API。選取的 GraphQL API 會顯示在您AWS帳戶表格中的 API 中。

b. 在來自其他AWS帳戶的 API 表格中，選擇新增來源 API。此列表中的 GraphQL API 來自通過AWS Resource Access Manager（AWS RAM）將其資源共享給您的其他帳戶。在此表格中選取 GraphQL API 的程序與上一節中的程序相同。如需透過共用資源的詳細資訊AWS RAM，請參閱什麼是AWS Resource Access Manager? 。

完成後，請選擇 [下一步]。

c. 添加您的主要身份驗證模式。如需詳細資訊，請參閱授權和驗證。選擇下一步。

d. 檢閱您的輸入，然後選擇「建立 API」。

自省

AWS AppSync使從現有的關係數據庫構建 API 變得容易。它的內部檢查實用程序可以從數據庫表中發現模型並提出 GraphQL 類型。AWS AppSync控制台的「創建 API」嚮導可以立即從 Aurora MySQL 或 PostgreSQL 數據庫生成 API。它會自動創建類型和 JavaScript 解析器來讀取和寫入數據。

AWS AppSync透過 Amazon RDS 資料 API 與 Amazon Aurora 資料庫直接整合。Amazon RDS 資料 API 不需要持續性的資料庫連線，而是提供可AWS AppSync連接至執行SQL陳述式的安全 HTTP 端點。您可以使用此功能為 Aurora 上的 MySQL 和 PostgreSQL 工作負載建立關聯式資料庫 API。

為您的關係數據庫構建 API AWS AppSync 具有以下幾個優點：

- 您的數據庫不會直接暴露給客戶端，從數據庫本身解耦接接入點。
- 您可以根據不同應用程式的需求，建置專為特定用途建置的 API，不再需要前端自訂商務邏輯。這與前端後端（BFF）模式對齊。
- 可以使用各種授權模式來控制訪問的AWS AppSync層實現授權和訪問控制。無需額外的計算資源即可連接到數據庫，例如託管 Web 服務器或代理連接。
- 您可以透過訂閱新增即時功能，並透過 AppSync 自動推送至連線的用戶端進行資料突變。
- 客戶端可以使用像 443 這樣的通用端口通過 HTTPS 連接到 API。

AWS AppSync使從現有的關係數據庫構建 API 變得容易。它的內部檢查實用程序可以從數據庫表中發現模型並提出 GraphQL 類型。AWS AppSync控制台的「創建 API」嚮導可以立即從 Aurora MySQL 或 PostgreSQL 數據庫生成 API。它會自動創建類型和 JavaScript解析器來讀取和寫入數據。

AWS AppSync提供整合 JavaScript 式公用程式，可簡化在解析器中撰寫 SQL 陳述式的作業 您可以使用具有動態值AWS AppSync的靜態陳述式的`sql`標籤範本，或使用`rds`模組公用程式以程式設計方式建立陳述式。有關更多信息，請參閱 RDS 數據源和內置模塊的解析器函數參考。

使用內部檢查功能（控制台）

如需詳細教學課程和入門指南，請參閱教學課程：使用資料 API 的 Aurora PostgreSQL 無伺服器。

主AWS AppSync控台可讓您在幾分鐘內從使用資料 API 設定的現有 Aurora 資料庫建立 AWS AppSync GraphQL API。這會根據您的資料庫組態快速產生作業結構描述。您可以按原樣使用 API，也可以在其上構建以添加功能。

1. 登入 AWS Management Console 並開啟 AppSync主控台。

- 在儀表板上，選擇 Create API (建立 API)。

2. 在 API 選項下，選擇 GraphQL API，從 Amazon Aurora 叢集開始，然後選擇下一步。

   a. 輸入 API 名稱。這將被用作控制台中 API 的標識符。

   b. 如需聯絡詳細資料，您可以輸入聯絡窗口以識別 API 的管理員。此為選用欄位。

   c. 在私人 API 配置下，您可以啟用私有 API 功能。私有 API 只能從已設定的 VPC 端點 (VPCE) 存取。如需詳細資訊，請參閱私有 API。

      我們不建議在此範例中啟用此功能。檢閱輸入後，請選擇「下一步」。

3. 在「資料庫」頁面中，選擇選取資料庫。

   a. 您需要從叢集中選擇資料庫。第一個步驟是選擇叢集所在的區域。

   b. 從下拉式清單中選擇 Aurora 叢集。請注意，在使用資源之前，您必須先建立並啟用對應的資料 API。

   c. 接下來，您必須將資料庫的認證新增至服務。這主要是使用AWS Secrets Manager. 選擇您的密碼所在的地區。如需如何擷取機密資訊的詳細資訊，請參閱尋找機密或擷取機密。

   d. 從下拉式清單中新增您的密碼。請注意，使用者必須具有資料庫的讀取權限。

4. 選擇匯入。

AWS AppSync將開始檢查您的資料庫，探索資料表、資料行、主索引鍵和索引。它會檢查發現的資料表是否可以在 GraphQL API 中受到支援。請注意，為了支持創建新行，表需要一個主鍵，它可以使用多個列。 AWS AppSync將表格欄對映至類型欄位，如下所示：

資料類型	欄位類型
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. 資料表探索完成後，「資料庫」區段就會填入您的資訊。在新的 [資料庫資料表] 區段中，表格中的資料可能已經填入，並轉換為結構描述的類型。如果您沒有看到某些必要資料，可以選擇「新增表格」，在顯示的強制回應中按一下這些類型的核取方塊，然後選擇「新增」來檢查該資料。

   若要從「資料庫表格」區段移除類型，請按一下要移除之類型旁邊的核取方塊，然後選擇「移除」。如果您想稍後再次添加它們，則刪除的類型將被放置在「添加表」模式中。

   請注意，AWS AppSync使用表名作為類型名稱，但您可以重命名它們-例如，將複數表格名稱（如*##*）更改為類型名稱 *Movie*。若要重新命名「資料庫表格」區段中的類型，請按一下您要重新命名之類型的核取方塊，然後按一下「類型名稱」欄中的鉛筆圖示。

   若要根據您的選擇預覽結構描述的內容，請選擇 [預覽結構描述]。請注意，此結構定義不能為空，因此您必須至少將一個表轉換為類型。此外，此結構描述的大小不能超過 1 MB。

   - 在 [服務角色] 下，選擇是要專門針對此匯入建立新的服務角色，還是使用現有角色。
6. 選擇下一步。
7. 接下來，選擇創建只讀 API（僅查詢）還是用於讀取和寫入數據（包含查詢和突變）的 API。後者還支持突變觸發的實時訂閱。
8. 選擇下一步。
9. 檢閱您的選擇，然後選擇 [建立 API]。 AWS AppSync將創建 API 並將解析器附加到查詢和突變。產生的 API 可完全運作，並可視需要進行擴充。

使用內部檢查功能 (API)

您可以使用`StartDataSourceIntrospection`內部檢查 API 以程式設計方式探索資料庫中的模型。有關該命令的更多詳細信息，請參閱使用 `StartDataSourceIntrospection`API。

若要使用`StartDataSourceIntrospection`，請提供您的 Aurora 叢集 Amazon 資源名稱 (ARN)、資料庫名稱和AWS Secrets Manager秘密 ARN。此指令會啟動內部檢查程序。您可以使

用GetDataSourceIntrospection指令擷取結果。您可以指定命令是否應傳回探索到的模型的儲存定義語言 (SDL) 字串。這對於直接從發現的模型產生 SDL 結構描述定義非常有用。

例如，如果您有下列簡單資料Todos表的資料定義語言 (DDL) 陳述式：

```
create table if not exists public.todos
(
id serial constraint todos_pk primary key,
description text,
due timestamp,
"createdAt" timestamp default now()
);
```

您可以使用下列方式開始內省。

```
aws appsync start-data-source-introspection \
  --rds-data-api-config resourceArn=<cluster-arn>,secretArn=<secret-
arn>,databaseName=database
```

接下來，使用GetDataSourceIntrospection命令檢索結果。

```
aws appsync get-data-source-introspection \
  --introspection-id a1234567-8910-abcd-efgh-identifier \
  --include-models-sdl
```

這將返回以下結果。

```
{
    "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
    }
}
```

建立用戶端應用程式

您可以使用任何 AWS AppSync GraphQL 用戶端連線到 GraphQL API，但我們強烈建議您使用 Amplify 用戶端。Amplify 不僅為 GraphQL API 自動產生強型別用戶端 SDK，還可支援用戶端應用程式中的即時資料和增強的 GraphQL 查詢功能。對於 Web 應用程序，Amplify 可以生成一個 JavaScript客戶端。對於那些針對跨平台或移動環境，Amplify 迎合 Android，iOS 和反應本地。若要深入研究這些平台的用戶端程式碼產生，請參閱 Amplify 文件。以下是使用 JavaScript React 應用程式啟動旅程的指南：

> **Note**
>
> 在開始使用之前，您必須先安裝和設定 npm 和 Amazon CLI。如果您使用的是 Amplify v6 客戶端，請按照本指南進行操作。

開始使用：

1. 在本機電腦上，導覽至專案的目錄。使用以下命令安裝 Amplify 程式庫：

```
npm install aws-amplify
```

2. 下載您的配置文件並將其放置在項目文件夾中。您的配置文件通常包含一個已定義某些設置（端點，地區，授權模式等）的config變量。例如，它可能看起來像這樣：

```
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. 在您的程式碼中，匯入 Amplify 程式庫和設定以設定 Amplify：

```
import { Amplify } from 'aws-amplify';
import config from './aws-exports.js';

Amplify.configure(config);
```

或者，使用 API 配置中的代碼片段直接設置 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. 使用 Amplify 工具鏈，您可以選擇根據結構描述自動產生作業，這樣可以節省手動指令碼的工作量。在應用程式的根目錄中，使用下列 CLI 指令：

```
npx @aws-amplify/cli codegen add --apiId <id goes here> --region <region goes here>
```

這將下載您的 API 的模式，默認情況下，將客戶端幫助程序代碼生成到`src/graphql`文件夾中。每次部署 API 之後，您都可以重新執行下列命令，以產生更新的 GraphQL 陳述式和類型：

```
npx @aws-amplify/cli codegen
```

5. 您現在可以為 Android 生成模型, 斯威夫特, 撲, 和 JavaScript DataStore. 使用下列指令來下載結構描述：

```
aws appsync get-introspection-schema --api-id <id goes here> --region <region goes
 here> --format SDL schema.graphql
```

然後，從應用程序的根目錄運行以下命令：

```
npx @aws-amplify/cli codegen models \
--model-schema schema.graphql \
--target [android|ios|flutter|javascript|typescript] \
--output-dir ./
```

解析器教程（ ）JavaScript

資料來源和解析器是AWS AppSync 轉譯 GraphQL 要求以及從資源擷取資訊的方式。AWSAWS AppSync 支援自動佈建以及與特定資料來源類型的連線。AWS AppSync 支援AWS Lambda、Amazon DynamoDB、關聯式資料庫 (Amazon Aurora 無伺服器)、Amazon OpenSearch 服務和 HTTP 端點做為資料來源。您可以將 GraphQL API 與現有AWS資源搭配使用，也可以建置資料來源和解析器。本區段將帶您在一系列教學課程中完成此過程，以更好地了解詳細資訊如何運作和調校選項。

主題

- 教學課程：解析器 JavaScript
- 教學課程：Lambda 解析器
- 教學課程：本機解析器
- 教學課程：結合 GraphQL 解析器
- 教程：亞馬遜OpenSearch服務解析器
- 教學課程：交易解析器
- 教學課程：批次解析器
- 教學課程：HTTP 解析器
- 教學課程：使用資料 API 的 Aurora

教學課程：解析器 JavaScript

在本教學中，您將匯入 Amazon DynamoDB 表格AWS AppSync並將其連接，以使用 JavaScript 管道解析器建立功能齊全的 GraphQL API，您可以在自己的應用程式中運用這些解析器。

您將使用AWS AppSync主控台佈建 Amazon DynamoDB 資源、建立解析器，然後將它們連接到資料來源。您也可以透過 GraphQL 陳述式讀取和寫入 Amazon DynamoDB 資料庫，並訂閱即時資料。

必須完成一些特定步驟，才能將 GraphQL 陳述式轉換為 Amazon DynamoDB 作業，以及將回應翻譯回 GraphQL。本教學課程概述透過幾個真實世界案例和資料存取模式的組態程序。

建立您的 GraphQL API

若要在中建立 GraphQL API AWS AppSync

1. 開啟 AppSync 主控台，然後選擇 [建立 API]。

2. 選擇從頭開始設計，然後選擇下一步。
3. 為您的 API 命名PostTutorialAPI，然後選擇「下一步」。跳至檢閱頁面，同時將其餘選項設定為預設值並進行選擇Create。

主AWS AppSync控台會為您建立新的 GraphQL API。通過詳細說明，它使用 API 密鑰身份驗證模式。您可以使用主控台來設定其他 GraphQL API 和對其執行查詢，以進行此教學的其他部分。

定義一個基本的後期 API

現在您已經擁有 GraphQL API，您可以設定一個基本結構描述，以便基本建立、擷取和刪除貼文資料。

若要將資料加入至資料架構

1. 在您的 API 中，選擇「結構描述」標籤。
2. 我們將創建一個定義Post類型和操作addPost添加和獲取Post對象的模式。在「結構描述」窗格中，以下列程式碼取代內容：

```
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. 選擇 Save Schema (儲存結構描述)。

設定您的 Amazon DynamoDB 表

主AWS AppSync控台可協助佈建將您自己的AWS資源存放在 Amazon DynamoDB 表格中所需的資源。在此步驟中，您將建立一個 Amazon DynamoDB 表來存放您的貼文。您還將設置我們稍後將使用的次要索引。

若要建立您的 Amazon DynamoDB 資料表

1. 在 [架構] 頁面上，選擇 [建立資源]。
2. 選擇「使用現有類型」，然後選擇Post類型。
3. 在「其他索引」段落中，選擇「新增索引」。
4. 命名索引author-index。
5. Primary key將設定為author和Sort金鑰None。
6. 停用「自動產生 GraphQL」。在這個例子中，我們將自己創建解析器。
7. 選擇 建立 。

您現在有一個名為的新資料來源PostTable，您可以透過瀏覽側邊選項卡中的資料來源來查看該資料來源。您將使用此資料來源將查詢和突變連結至 Amazon DynamoDB 表格。

設定一個新的 addPost 解析器 (Amazon DynamoDB) PutItem

現在AWS AppSync已經知道 Amazon DynamoDB 表，您可以透過定義解析器將其連結至個別查詢和突變。您建立的第一個解析器是使用的addPost管道解析器 JavaScript，可讓您在 Amazon DynamoDB 表中建立貼文。配管解析器具有以下元件：

- 在 GraphQL 結構描述中要附加解析程式的位置。在這種情況下，您會對 createPost 類型上的 Mutation 欄位設定解析程式。當調用者調用突變時，將調用此解析器。{ addPost(...) {...} }

- 用於此解析程式的資料來源。在這種情況下，您想要使用先前定義的 DynamoDB 資料來源，以便將項目新增到 `post-table-for-tutorial` DynamoDB 表中。
- 要求處理常式。要求處理常式是一個函數，可處理來自呼叫者的傳入要求，並將其轉譯為AWS AppSync針對 DynamoDB 執行的指示。
- 響應處理程序。回應處理常式的工作是處理來自 DynamoDB 的回應，並將其轉換回 GraphQL 預期的項目。如果 DynamoDB 中的資料形狀與 GraphQL 中的 `Post` 類型不同，此功能會很有用，但如果他們的形狀一樣，您只需傳遞資料。

若要設定您的解析器

1. 在您的 API 中，選擇「結構描述」標籤。
2. 在「解析器」窗格中，找到**`Mutation`**類型下的**`addPost`**欄位，然後選擇「附加」。
3. 選擇您的資料來源，然後選擇「建立」。
4. 在程式碼編輯器中，以下列程式碼片段取代程式碼：

```
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. 選擇 儲存 。

> **Note**
>
> 在此程式碼中，您可以使用 DynamoDB 模組公用程式，讓您輕鬆建立 DynamoDB 請求。

AWS AppSync帶有一個稱為自動 ID 生成的實用程序`util.autoId()`，該實用程序用於為您的新帖子生成 ID。如果您未指定 ID，公用程式會自動為您產生 ID。

```
const key = { id: ctx.args.id ?? util.autoId() }
```

如需有關可用公用程式的詳細資訊 JavaScript，請參閱解析器和函數的JavaScript執行階段功能。

呼叫 API 以新增貼文

現在已設定解析器，AWS AppSync可以將傳入的addPost突變轉換為 Amazon DynamoDB 作業。PutItem您現在可以執行變動以將項目放置到資料表中。

若要執行作業

1. 在您的 API 中，選擇 [查詢] 索引標籤。
2. 在 [查詢] 窗格中，新增下列變更：

```
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. 選擇 [執行] (橘色播放按鈕)，然後選擇addPost。新建立的貼文的結果應該會出現在 [查詢] 窗格右側的 [結果] 窗格中。其看起來與下列類似：

```
{
  "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
    }
  }
}
```

下面的解釋顯示了發生了什麼：

1. AWS AppSync 收到 addPost 變動要求。
2. AWS AppSync執行解析器的請求處理程序。該ddb.put函數創建一個如下所示的PutItem請求：

```
{
  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使用此值來產生並執行 Amazon DynamoDB PutItem 請求。
4. AWS AppSync 接受 PutItem 要求的結果，並將其轉換回 GraphQL 類型。

```
{
    "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. 響應處理程序立即返回結果（`return ctx.result`）。
6. 最終結果會在 GraphQL 回應中可見。

設置 getPost 解析器（Amazon DynamoDB） GetItem

現在您可以將資料新增至 Amazon DynamoDB 資料表，您必須設定`getPost`查詢，才能從表格擷取該資料。若要執行此作業，您可以設定另一個解析程式。

若要新增您的解析程式

1. 在您的 API 中，選擇「結構描述」標籤。
2. 在右側的 [解析器] 窗格中，尋找**`Query`**類型上的**`getPost`**欄位，然後選擇 [附加]。
3. 選擇您的資料來源，然後選擇「建立」。
4. 在程式碼編輯器中，以下列程式碼片段取代程式碼：

```
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. 儲存您的解析程式。

> **Note**
>
> 在此解析器中，我們為響應處理程序使用箭頭函數表達式。

呼叫 API 以取得貼文

現在已設定解析程式，就AWS AppSync知道如何將傳入的`getPost`查詢轉譯成 Amazon DynamoDB `GetItem` 作業。您現在可以執行查詢，擷取您稍早建立的貼文。

若要執行您的查詢

1. 在您的 API 中，選擇 [查詢] 索引標籤。

2. 在「查詢」窗格中，新增下列程式碼，並使用您在建立貼文後複製的 ID：

```
query getPost {
  getPost(id: "123") {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. 選擇 [執行] (橘色播放按鈕)，然後選擇`getPost`。新建立的貼文的結果應該會出現在 [查詢] 窗格右側的 [結果] 窗格中。

4. 從 Amazon DynamoDB 擷取的貼文應會顯示在「查詢」窗格右側的「結果」窗格中。其看起來與下列類似：

```
{
  "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，採取以下示例：

```
query getPost {
  getPost(id: "123") {
    id
    author
```

```
    title
  }
}
```

如果您的getPost查詢只需要id、和 authortitle，您可以將請求函數變更為使用投影運算式，僅從 DynamoDB 表指定您想要的屬性，以避免不必要的資料從 DynamoDB 傳輸到。AWS AppSync例如，request 函數可能如下所示：

```
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selectionSetList與getPost來表示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
```

創建一個 updatePost 突變（Amazon DynamoDB）UpdateItem

到目前為止，您可以在 Amazon DynamoDB 中建立和擷取Post物件。接下來，您將設置一個新的突變來更新對象。與需要指定所有字段的addPost突變相比，此突變允許您僅指定要更改的字段。它還引入了一個新的expectedVersion參數，允許您指定要修改的版本。您將設定一個條件，以確保您正在修改物件的最新版本。您可以使用UpdateItem亞馬遜動態 B 操作來執行此操作。

更新您的解析器

1. 在您的 API 中，選擇「結構描述」標籤。

2. 修改 Schema (結構描述) 窗格中的 Mutation 類型來新增 updatePost 變動，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. 選擇 Save Schema (儲存結構描述)。
4. 在右側的「解析器」窗格中，在**Mutation**類型上找到新建立的**updatePost**欄位，然後選擇「附加」。使用下面的代碼片段創建新的解析器：

```
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. 儲存您所做的任何變更。

此解析程式用ddb.update來建立 Amazon DynamoDB UpdateItem 請求。您不需要撰寫整個項目，而是要求 Amazon DynamoDB 更新某些屬性。這是使用 Amazon DynamoDB 更新運算式來完成的。

該ddb.update函數需要一個鍵和一個更新對象作為參數。然後，檢查傳入參數的值。將值設定為時null，請使用 DynamoDB remove 作業來表示應該從 DynamoDB 項目中移除該值。

還有一個新的condition部分。條件運算式可讓AWS AppSync您根據 Amazon DynamoDB 中已有的物件狀態來判斷請求是否應該成功，然後再執行操作。在這種情況下，只有在 Amazon DynamoDB 中目前的項目version欄位與expectedVersion引數完全相符時，您才希望UpdateItem請求成功。更新項目時，我們想要增加的值version。這很容易與操作功能做到increment。

如需有關條件運算式的詳細資訊，請參閱條件運算式文件。

如需有關UpdateItem請求的詳細資訊，請參閱UpdateItem文件和 DynamoDB 模組文件。

如需如何撰寫更新運算式的詳細資訊，請參閱 DynamoDB UpdateExpressions 文件。

呼叫 API 以更新貼文

讓我們嘗試使用新的解析器更新Post對象。

更新物件的步驟

1. 在您的 API 中，選擇 [查詢] 索引標籤。
2. 在 [查詢] 窗格中，新增下列變異。您還需要將id參數更新為您之前記下的值：

```
mutation updatePost {
  updatePost(
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 1
  ) {
```

```
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. 選擇 [執行] (橘色播放按鈕)，然後選擇updatePost。
4. Amazon DynamoDB 中更新的貼文應顯示在「查詢」窗格右側的「結果」窗格中。其看起來與下列類似：

```
{
  "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
    }
  }
}
```

在此請求中，您要求 AWS AppSync Amazon DynamoDB 僅更新title和content欄位。所有其他字段都被單獨留下（除了增加字version段）。您將title屬性設定為新值，並從貼文中移除content屬性。author、url、ups 和 downs 欄位維持原狀。嘗試再次執行突變請求，同時保持請求完全按原樣。您應該會看到類似以下的回應：

```
{
  "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)"
    }
  ]
}
```

請求失敗，因為條件運算式評估為`false`：

1. 第一次執行請求時，Amazon DynamoDB 中貼文`version`欄位的值為`1`，此值與引數相`expectedVersion`符。請求成功，這表示該`version`欄位在 Amazon DynamoDB 中增加了。2
2. 第二次執行請求時，Amazon DynamoDB 中貼文`version`欄位的值為`2`，這與引`expectedVersion`數不符。

此模式通常稱為樂觀鎖定。

創建投票突變 (Amazon DynamoDB UpdateItem)

該`Post`類型包含`ups`和`downs`字段，用於啟用上票和下票的記錄。但是，此時 API 不允許我們對它們做任何事情。讓我們添加一個突變，讓我們對帖子進行讚揚和低票。

要添加你的突變

1. 在您的 API 中，選擇「結構描述」標籤。
2. 在「架構」窗格中，修改`Mutation`類型並添加`DIRECTION`枚舉以添加新的投票突變：

```
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. 選擇 Save Schema (儲存結構描述)。
4. 在右側的 [解析器] 窗格中，尋找**Mutation**類型上新建立的**vote**欄位，然後選擇 [附加]。建立新的解析程式，並以下列程式碼片段取代程式碼：

```
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. 儲存您所做的任何變更。

呼叫 API 以加票或下票貼文

現在已設定新的解析器，AWS AppSync知道如何將傳入upvotePost或downvote變異轉換為 Amazon DynamoDB 作業。UpdateItem您現在可以執行變動，對您之前建立的文章表達贊同或不贊同。

運行你的突變

1. 在您的 API 中，選擇 [查詢] 索引標籤。
2. 在 [查詢] 窗格中，新增下列變異。您還需要將id參數更新為您之前記下的值：

```
mutation votePost {
  vote(id:123, direction: UP) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. 選擇 [執行] (橘色播放按鈕)，然後選擇votePost。
4. Amazon DynamoDB 中更新的貼文應顯示在「查詢」窗格右側的「結果」窗格中。其看起來與下列類似：

```
{
  "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. 再選擇 [執行幾次]。1每次執行查詢時，您應該會看到ups和version欄位遞增。
6. 變更查詢，以不同的方式呼叫它DIRECTION。

```
mutation votePost {
  vote(id:123, direction: DOWN) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

7. 選擇 [執行] (橘色播放按鈕)，然後選擇votePost。

   這次，您應該會在1每次執行查詢時看到downs和version欄位遞增。

設定 deletePost 解析器 (Amazon DynamoDB) DeleteItem

接下來，您需要創建一個突變以刪除帖子。您將使用 DeleteItem Amazon DynamoDB 操作來執行此操作。

要添加你的突變

1. 在您的架構中，選擇「結構描述」頁籤。
2. 在「結構描述」窗格中，修改Mutation類型以添加新的deletePost突變：

```
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. 這次，您將expectedVersion欄位設定為選用。接下來，選擇「儲存綱要」。
4. 在右側的「解析器」窗格中，在**Mutation**類型中找到新建立的**delete**欄位，然後選擇「附加」。使用下面的代碼創建一個新的解析器：

```
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**
>
> 引expectedVersion數是選擇性引數。如果呼叫者在請求中設定expectedVersion引數，請求處理常式會新增條件，該條件僅在項目已刪除或 Amazon DynamoDB 中貼文的version屬性完全符合時，才允許DeleteItem請求成功。expectedVersion如果省略，將不會在 DeleteItem 要求中指定條件表達式。無論 Amazon DynamoDB 中是version否存在該項目的值為何，它都會成功。
> 即使您正在刪除某個項目，如果已刪除的項目尚未刪除，您也可以返回該項目。

如需有關DeleteItem要求的詳細資訊，請參閱DeleteItem文件。

呼叫 API 以刪除貼文

現在解析器已經設定好，就AWS AppSync知道如何將傳入的delete突變轉換成 Amazon DynamoDB 作業。DeleteItem您現在可以執行變動以在資料表中刪除項目。

運行你的突變

1. 在您的 API 中，選擇 [查詢] 索引標籤。
2. 在 [查詢] 窗格中，新增下列變異。您還需要將id參數更新為您之前記下的值：

```
mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

3. 選擇 [執行] (橘色播放按鈕)，然後選擇deletePost。
4. 該貼文已從 Amazon DynamoDB 刪除。請注意，會AWS AppSync傳回從 Amazon DynamoDB 刪除的項目值，該值應顯示在「查詢」窗格右側的「結果」窗格中。其看起來與下列類似：

```
{
  "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. 只有當這個呼叫是實際從 Amazon DynamoDB 中刪除的呼叫時，才會傳回該值。deletePost再次選擇 [執行]。
6. 呼叫仍然成功，但沒有返回任何值：

```
{
  "data": {
    "deletePost": null
  }
}
```

7. 現在，讓我們嘗試刪除一個帖子，但這次指定expectedValue. 首先，您需要創建一個新帖子，因為您剛剛刪除了迄今為止一直在使用的帖子。
8. 在 [查詢] 窗格中，新增下列變更：

```
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. 選擇 [執行] (橘色播放按鈕)，然後選擇addPost。

10新建立的貼文的結果應該會出現在 [查詢] 窗格右側的 [結果] 窗格中。記錄新創建id的對象的，因為您只需要一會兒。其看起來與下列類似：

```
{
  "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現在，讓我們嘗試刪除具有 expec ted Version 非法值的帖子。在 [查詢] 窗格中，新增下列變異。您還需要將id參數更新為您之前記下的值：

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
```

```
}
```

12選擇 [執行] (橘色播放按鈕)，然後選擇deletePost。傳回下列結果：

```
{
  "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要求失敗，因為條件運算式評估為false。Amazon DynamoDB version 中貼文的值與引數中expectedValue指定的值不符。會在 GraphQL 回應 data 區段中 errors 欄位傳回物件的目前值。重試要求，但更正 expectedVersion：

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
```

```
    ups
    downs
    version
  }
}
```

14.選擇 [執行] (橘色播放按鈕)，然後選擇deletePost。

這次請求成功，並傳回從 Amazon DynamoDB 刪除的值：

```
{
  "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.再次選擇 [執行]。呼叫仍然成功，但這次不會傳回任何值，因為在 Amazon DynamoDB 中已刪除貼文。

```
{ "data": { "deletePost": null } }
```

設定 AallPost 解析器 (Amazon DynamoDB 掃描)

到目前為止，只有當您知道要查看id的每篇文章的內容時，API 才有用。讓我們新增新的解析程式，傳回資料表中的所有文章。

要添加你的突變

1. 在您的 API 中，選擇「結構描述」標籤。
2. 修改 Schema (結構描述) 窗格中的 `Query` 類型以新增 `allPost` 查詢，如下所示：

```
type Query {
```

```
    allPost(limit: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

3. 新增 PaginationPosts 類型：

```
type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}
```

4. 選擇 Save Schema (儲存結構描述)。
5. 在右側的「解析器」窗格中，在**Query**類型中找到新建立的**allPost**欄位，然後選擇「附加」。使用以下代碼創建一個新的解析器：

```
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 };
}
```

   此解析器的請求處理程序需要兩個可選參數：

   - limit-指定單次呼叫中要傳回的最大項目數。
   - nextToken-用於檢索下一組結果（我們將顯示稍後的值nextToken來自哪裡）。

6. 儲存對解析程式所做的任何變更。

如需Scan要求的詳細資訊，請參閱 Scan 參考文件。

調用 API 掃描所有帖子

現在已設定解析程式，就AWS AppSync知道如何將傳入的allPost查詢轉譯成 Amazon DynamoDB Scan 作業。您現在可以掃描資料表來擷取所有文章。在您可以嘗試之前，您必須將一些資料填入資料表，因為您已刪除目前為止所使用的項目。

若要新增和查詢資料

1. 在您的 API 中，選擇 [查詢] 索引標籤。
2. 在 [查詢] 窗格中，新增下列變更：

```
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. 選擇 [執行] (橘色播放按鈕)。
4. 現在，讓我們來掃描資料表，一次會傳回五個結果。在「查詢」窗格中，新增下列查詢：

```
query allPost {
  allPost(limit: 5) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

5. 選擇 [執行] (橘色播放按鈕)，然後選擇`allPost`。

   前五個貼文應顯示在「查詢」窗格右側的「結果」窗格中。其看起來與下列類似：

```
{
  "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. 您收到五個結果nextToken，一個可用來取得下一組結果。更新 allPost 查詢以包括來自之前結果組的 nextToken：

```
query allPost {
  allPost(
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
      author
    }
```

```
    nextToken
  }
}
```

7. 選擇 [執行] (橘色播放按鈕)，然後選擇allPost。

   其餘四個貼文應顯示在「查詢」窗格右側的「結果」窗格中。這組結果nextToken中沒有，因為您已經通過所有九個帖子進行了分頁，而沒有剩餘任何帖子。其看起來與下列類似：

```
{
  "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
    }
  }
}
```

設定 allPostsBy作者解析程式 (Amazon DynamoDB 查詢)

除了針對所有貼文掃描 Amazon DynamoDB 之外，您也可以查詢 Amazon DynamoDB 以擷取特定作者建立的貼文。您之前建立的 Amazon DynamoDB 表格已經有一個GlobalSecondaryIndex呼叫author-index，您可以將其與 Amazon DynamoDB 操作搭配使用，以擷取特定Query作者建立的所有貼文。

若要新增您的查詢

1. 在您的 API 中，選擇「結構描述」標籤。
2. 修改 Schema (結構描述) 窗格中的 Query 類型以新增 allPostsByAuthor 查詢，如下所示：

```
type Query {
    allPostsByAuthor(author: String!, limit: Int, nextToken: String): PaginatedPosts!
    allPost(limit: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

請注意，這會使用與allPost查詢搭配使用的相同PaginatedPosts類型。

3. 選擇 Save Schema (儲存結構描述)。
4. 在右側的 [解析器] 窗格中，尋找**Query**類型上新建立的**allPostsByAuthor**欄位，然後選擇 [附加]。使用下面的代碼片段創建一個解析器：

```
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 };
}
```

像解析器一樣，此allPost解析器有兩個可選參數：

- limit-指定單次呼叫中要傳回的最大項目數。
- nextToken-擷取下一組結果 (nextToken可從上一個呼叫取得的值)。

5. 儲存對解析程式所做的任何變更。

如需有關Query要求的詳細資訊，請參閱查詢參考文件。

調用 API 以按作者查詢所有帖子

現在解析器已經設定好，AWS AppSync知道如何將傳入的allPostsByAuthor突變轉換為針對索引的 DynamoDB Query 作業。author-index您現在可以查詢資料表以擷取特定作者的所有文章。

然而，在此之前，讓我們在表中填充一些更多的帖子，因為到目前為止每個帖子都有相同的作者。

若要新增資料和查詢

1. 在您的 API 中，選擇 [查詢] 索引標籤。
2. 在 [查詢] 窗格中，新增下列變更：

```
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. 選擇 [執行] (橘色播放按鈕)，然後選擇addPost。
4. 現在，讓我們查詢資料表，傳回 Nadia 撰寫的所有文章。在「查詢」窗格中，新增下列查詢：

```
query allPostsByAuthor {
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

5. 選擇 [執行] (橘色播放按鈕)，然後選擇allPostsByAuthor。所有由撰寫的貼文都Nadia應顯示在「查詢」窗格右側的「結果」窗格中。其看起來與下列類似：

```
{
  "data": {
```

```
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

6. 分頁適用於 Query，如同它適用於 Scan。例如，讓我們查詢 AUTHORNAME 的所有文章，一次取得五篇。
7. 在「查詢」窗格中，新增下列查詢：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

8. 選擇 [執行] (橘色播放按鈕)，然後選擇allPostsByAuthor。所有由撰寫的貼文都AUTHORNAME應顯示在「查詢」窗格右側的「結果」窗格中。其看起來與下列類似：

```
{
  "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. 使用之前查詢傳回的值更新 nextToken 引數，如下所示：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    limit: 5
    nextToken: "<token>"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

10選擇 [執行] (橘色播放按鈕)，然後選擇allPostsByAuthor。其餘撰寫的貼文AUTHORNAME應顯示在「查詢」窗格右側的「結果」窗格中。其看起來與下列類似：

```
{
```

```
  "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

到目前為止，該Post類型已經是一個平鍵/值對象。您也可以使用解析器建模複雜物件，例如集合、清單和地圖。讓我們將 Post 類型更新為包含標籤。貼文可以有零個或多個標籤，這些標籤會以字串集形式儲存在 DynamoDB 中。您也會設定一些變動來新增和移除標籤，以及新查詢以掃描含特定標籤的文章。

若要設定您的資料

1. 在您的 API 中，選擇「結構描述」標籤。
2. 修改 Schema (結構描述) 窗格中的 Post 類型以新增 tags 欄位，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. 修改 Schema (結構描述) 窗格中的 Query 類型以新增 allPostsByTag 查詢，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. 修改 Schema (結構描述) 窗格中的 Mutation 類型以新增 addTag 和 removeTag 變動，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. 選擇 Save Schema (儲存結構描述)。

6. 在右側的 [解析器] 窗格中，尋找**Query**類型上新建立的**allPostsByTag**欄位，然後選擇 [附加]。使用下面的代碼片段創建您的解析器：

```
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. 儲存您對解析程式所做的任何變更。

8. 現在，addTag使用下面的代碼片段對Mutation字段執行相同的操作：

> **Note**
>
> 雖然 DynamoDB 公用程式目前不支援集合作業，但您仍然可以透過自行建立要求來與集合互動。

```
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. 儲存對解析程式所做的任何變更。

10removeTag使用下面的代碼片段為Mutation字段重複此操作一次：

```
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.儲存對解析程式所做的任何變更。

呼叫 API 來使用標籤

現在，您已經設定解析器，AWS AppSync知道如何將傳入addTag和allPostsByTag請求轉換為 DynamoDB UpdateItem 和作業。removeTag Scan為了嘗試看看，讓我們選擇您先前建立的其中一篇文章。例如，讓我們使用 Nadia 撰寫的一篇文章。

若要使用標籤

1. 在您的 API 中，選擇 [查詢] 索引標籤。
2. 在「查詢」窗格中，新增下列查詢：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
```

```
    posts {
      id
      title
    }
    nextToken
  }
}
```

3. 選擇 [執行] (橘色播放按鈕)，然後選擇allPostsByAuthor。
4. Nadia 的所有貼文都應顯示在「查詢」窗格右側的「結果」窗格中。其看起來與下列類似：

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

5. 讓我們使用標題為世界上最可愛的狗。記錄它，id因為你稍後會使用它。現在，讓我們嘗試添加一個dog標籤。
6. 在 [查詢] 窗格中，新增下列變異。您也必須將 id 引數更新為您先前記下的值。

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

7. 選擇 [執行] (橘色播放按鈕)，然後選擇addTag。該帖子與新的標籤更新：

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

8. 您可以新增更多標籤。更新突變以將tag參數更改為puppy：

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

9. 選擇 [執行] (橘色播放按鈕)，然後選擇addTag。該帖子與新的標籤更新：

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

10您也可以刪除標籤。在 [查詢] 窗格中，新增下列變異。您還需要將id參數更新為您之前記下的值：

```
mutation removeTag {
  removeTag(id:10 tag: "puppy") {
    id
```

```
    title
    tags
  }
}
```

11選擇 [執行] (橘色播放按鈕)，然後選擇removeTag。貼文將會更新，且 puppy 標籤將會刪除。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

12您也可以搜尋具有標籤的所有貼文。在「查詢」窗格中，新增下列查詢：

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

13選擇 [執行] (橘色播放按鈕)，然後選擇allPostsByTag。具有 dog 標籤的所有文章將會傳回，如下所示：

```
{
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
```

```
            "dog",
            "puppy"
          ]
        }
      ],
      "nextToken": null
    }
  }
}
```

結論

在本教學課程中，您已建置可讓您使用AWS AppSync和 GraphQL 操作 DynamoDB 中Post物件的 API。

若要清理，您可以從主控台刪除 AWS AppSync GraphQL API。

若要刪除與 DynamoDB 表關聯的角色，請在「資料來源」表格中選取您的資料來源，然後按一下編輯。請記下 [建立或使用現有角色] 底下角色的值。前往 IAM 主控台以刪除角色。

若要刪除 DynamoDB 表格，請在資料來源清單中按一下表格的名稱。這會帶您前往 DynamoDB 主控台，您可以在其中刪除表格。

教學課程：Lambda 解析器

您可以使用AWS Lambda與AWS AppSync以解析任何圖形 SQL 欄位。例如，GraphQL 查詢可能會傳送呼叫至亞馬遜關聯式資料庫服務 (Amazon RDS) 執行個體，而 GraphQL 突變可能會寫入 Amazon Kinesis 串流。在本節中，我們將示範如何撰寫 Lambda 函數，以根據 GraphQL 欄位作業的叫用來執行商務邏輯。

建立 Lambda 函數

以下示例顯示了寫入的 Lambda 函數Node.js（運行時：Node.js 18.x），作為博客文章應用程序的一部分，在博客文章上執行不同的操作。請注意，程式碼應儲存在副檔名為 .mis 的檔案名稱中。

```
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)
  }
}
```

此 Lambda 函數會依 ID 擷取貼文、新增貼文、擷取貼文清單，以及擷取給定貼文的相關貼文。

> **Note**
>
> 拉姆達函數使用`switch`上的聲明`event.field`以確定當前正在解析哪個字段。

使用建立此 Lambda 函數AWS管理主控台。

設定 Lambda 的資料來源

建立 Lambda 函數之後，請瀏覽至AWS AppSync控制台，然後選擇資料來源標籤。

選擇建立資料來源，輸入友好資料來源名稱（例如，**Lambda**），然後用於資料來源類型，選擇AWS Lambda功能。對於地區，選擇與您的功能相同的區域。對於函數 ARN，選擇你的 Lambda 函數的亞馬遜資源名稱（ARN）。

選擇您的 Lambda 函數後，您可以創建一個新的AWS Identity and Access Management（IAM）的角色（為其AWS AppSync指派適當的權限) 或選擇具有下列內嵌原則的現有角色：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/LAMBDA_FUNCTION"
        }
    ]
}
```

您還必須與以下方式建立信任關係AWS AppSync針對 IAM 角色，如下所示：

```
{
```

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

建立圖形 SQL 結構描述

現在，資料來源已連接至您的 Lambda 函數，請建立 GraphQL 結構描述。

從資料架構編輯器AWS AppSync控制台中，請確保您的模式與以下模式匹配：

```
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

現在您已經註冊了 Lambda 資料來源和有效的 GraphQL 結構描述，您可以使用解析器將 GraphQL 欄位連接到 Lambda 資料來源。

您將建立一個使用AWS AppSync JavaScript(APPSYNC_JS) 執行階段並與您的 Lambda 函數互動。了解有關寫作的更多信息AWS AppSync解析器和功能JavaScript，請參閱JavaScript解析器和函數的運行時功能。

如需 Lambda 對應範本的詳細資訊，請參閱JavaScriptLambda 的解析器函數參考。

在此步驟中，您可以為下列欄位附加解析程式至 Lambda 函數：`getPost(id:ID!): Post`,`allPosts: [Post]`,`addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`，以及`Post.relatedPosts: [Post]`。從網要中的編輯器AWS AppSync主控台，在解析器」窗格中，選擇貼附旁邊的`getPost(id:ID!): Post`欄位。選擇您的資料來源。接下來，提供以下代碼：

```
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;
}
```

此解析程式碼會在呼叫 Lambda 函數時，將欄位名稱、引數清單和來源物件的相關內容傳遞給 Lambda 函數。選擇 儲存 。

您已成功連接第一個解析程式。對其餘欄位重複此操作。

測試您的圖形 SQL API

現在，您的 Lambda 函式已經連接到 GraphQL 解析程式，您可以使用主控台或用戶端應用程式執行一些變動和查詢。

在左側的AWS AppSync控制台，選擇查詢，然後貼入下列程式碼：

addPost Mutation

```
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 Query

```
query GetPost {
    getPost(id: "2") {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

allPosts Query

```
query AllPosts {
    allPosts {
        id
        author
```

```
        title
        content
        url
        ups
        downs
        relatedPosts {
            id
            title
        }
    }
}
```

返回錯誤

任何給定的字段解析度都可能導致錯誤。同AWS AppSync，您可以從下列來源引發錯誤：

- 解析器響應處理程序
- Lambda 函數

從解析器響應處理程序

若要引發故意錯誤，您可以使用util.error實用方法。它需要一個參數 aerrorMessage，一個errorType，以及一個可選的data價值。發生錯誤時，data 可將額外的資料傳回給用戶端。data物件會新增到 GraphQL 最後回應中的 errors。

下面的例子演示了如何使用它Post.relatedPosts: [Post]解析器響應處理程序。

```
// the Post.relatedPosts response handler
export function response(ctx) {
    util.error("Failed to fetch relatedPosts", "LambdaFailure", ctx.result)
    return ctx.result;
}
```

這會產生類似下列的 GraphQL 回應：

```
{
    "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"
                }
            ]
        }
    ]
}
```

其中的 `allPosts[0].relatedPosts` 為 null，因為錯誤以及 `errorMessage`、`errorType` 和 `data` 出現在 `data.errors[0]` 物件中。

從 Lambda 函式

AWS AppSync也瞭解 Lambda 函數所擲回的錯誤。Lambda 程式設計模型可讓您提高處理錯誤。如果 Lambda 函數拋出錯誤，AWS AppSync無法解析目前欄位。回應中只會設定從 Lambda 傳回的錯誤訊息。目前，您無法透過從 Lambda 函數引發錯誤，將任何多餘的資料傳回用戶端。

> **Note**
>
> 如果你的 Lambda 函數引發一個未處理錯誤，AWS AppSync使用 Lambda 設定的錯誤訊息。

下列 Lambda 函式會引發錯誤：

```
export const handler = async (event) => {
  console.log('Received event {}', JSON.stringify(event, 3))
  throw new Error('I always fail.')
}
```

在您的回應處理常式中收到錯誤。您可以通過將錯誤附加到響應中將其發送回 GraphQL 響應`util.appendError`。若要這樣做，請變更您的AWS AppSync函數響應處理程序：

```
// the lambdaInvoke response handler
export function response(ctx) {
  const { error, result } = ctx;
  if (error) {
    util.appendError(error.message, error.type, result);
  }
  return result;
}
```

這會傳回類似下列的 GraphQL 回應：

```
{
  "data": {
    "allPosts": null
  },
  "errors": [
    {
      "path": [
        "allPosts"
      ],
      "data": null,
      "errorType": "Lambda:Unhandled",
      "errorInfo": null,
      "locations": [
        {
          "line": 2,
```

```
          "column": 3,
          "sourceName": null
        }
      ],
      "message": "I fail. always"
    }
  ]
}
```

進階使用案例：批次處理

此範例中的 Lambda 函數具有relatedPosts此欄位會傳回指定貼文的相關貼文清單。在範例查詢中，allPosts來自 Lambda 函數的欄位呼叫會傳回五個貼文。因為我們指定我們也想解決relatedPosts對於每個返回的帖子，relatedPosts現場操作被調用五次。

```
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
        }
    }
}
```

雖然這在這個特定的例子中聽起來可能不大，但這種複合的過度提取可能會迅速破壞應用程序。

如果您要在同一查詢中，再次擷取傳回之相關 Posts 的 relatedPosts，那麼叫用的次數將大幅增加。

```
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
            }
        }
    }
}
```

在這個相對簡單的查詢中，AWS AppSync將調用拉姆達函數 1 + 5 + 25 = 31 次。

這是一個相當常見的挑戰，通常稱為 N+1 問題 (在此例中，N = 5)，它可能會增加應用程式的延遲和成本。

解決此問題的方法之一，是將類似的欄位解析程式請求一起批次處理。在這個例子中，它不會讓 Lambda 函數解析單個給定帖子的相關帖子列表，而是可以解析給定批次帖子的相關帖子列表。

為了演示這一點，讓我們更新解析器relatedPosts處理批處理。

```
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;
}
```

代碼現在將操作更改為Invoke至BatchInvoke當fieldName正在解決是relatedPosts。現在，啟用批處理功能設定批次處理部分。將最大批次處理大小設定為5。選擇 儲存 。

通過此更改，在解決時relatedPosts，Lambda 函數接收以下內容作為輸入：

```
[
    {
        "field": "relatedPosts",
        "source": {
            "id": 1
        }
    },
    {
        "field": "relatedPosts",
        "source": {
            "id": 2
        }
    },
    ...
]
```

何時BatchInvoke在請求中指定，Lambda 函數會接收請求清單並傳回結果清單。

具體而言，結果清單必須符合要求裝載項目的大小和順序，以便AWS AppSync可以相應地匹配結果。

在此批次處理範例中，Lambda 函數會傳回一批結果，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 程式碼來處理批次處理relatedPosts:

```
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)
  }
}
```

返回個別錯誤

前面的範例顯示，可以從 Lambda 函數傳回單一錯誤，或是從回應處理常式引發錯誤。對於批次呼叫，從 Lambda 函數引發錯誤會將整個批次標記為失敗。對於發生無法恢復的錯誤的特定情況（例如與資料倉庫的連接失敗），這可能是可以接受的。但是，如果批次中的某些項目成功而其他項目失敗，則可能會同時傳回錯誤和有效資料。因為AWS AppSync需要批次回應符合批次原始大小的清單元素，您必須定義可以區分有效資料與錯誤的資料結構。

例如，如果 Lambda 函數預期會傳回一批相關貼文，您可以選擇傳回清單Response每個物件都有選擇性的物件資料,錯誤訊息，以及錯誤類型欄位。如果出現 errorMessage 欄位，表示發生錯誤。

下列程式碼會示範如何更新 Lambda 函數：

```
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解析器代碼：

```
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
  }
}
```

回應處理常式現在會檢查 Lambda 函數傳回的錯誤Invoke作業, 檢查針對個別項目傳回的錯誤BatchInvoke操作，最後檢查fieldName。對於relatedPosts，函數返回result.data。對於所有其他字段，該函數只返回result。例如，請參閱下面的查詢：

```
query AllPosts {
  allPosts {
    id
    title
    content
    url
    ups
    downs
    relatedPosts {
      id
    }
    author
  }
```

```
}
```

此查詢會傳回類似下列內容的 GraphQL 回應：

```
{
  "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"
    }
  ]
}
```

設定最大批次處理大小

若要設定解析器上的最大批次處理大小，請在AWS Command Line Interface(AWS CLI):

```
$ 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**
>
> 提供請求對應範本時，您必須使用`BatchInvoke`使用批次處理的作業。

教學課程：本機解析器

AWS AppSync可讓您使用支援的資料來源 (AWS Lambda、亞馬遜或亞馬遜OpenSearch服務）執行各種操作。不過，在某些情況下，可能不需要呼叫支援的資料來源。

這就是本機解析程式的實用之處。而不是調用遠程數據源，本地解析器將只是前進請求處理程序的響應處理程序的結果。字段分辨率不會離開AWS AppSync。

本地解析器在大量情況下很有用。最常見的使用案例是發佈通知而不觸發資料來源呼叫。為了演示這個用例，讓我們構建一個 pub/sub 應用程序，用戶可以在其中發布和訂閱消息。此範例使用訂閱，因此，如果您不熟悉訂閱，可以遵循即時資料教學課程。

創建發布/訂閱應用程序

首先，通過選擇創建一個空白的圖形 SQL API從頭開始設計選項並在創建 GraphQL API 時配置可選的詳細信息。

在我們的 pub/sub 應用程序中，客戶可以訂閱和發布消息。每個已發佈的訊息都包含一個名稱和資料。將其添加到模式中：

```
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"])
}
```

接下來，讓我們將解析器附加到Mutation.publish欄位。在解析器旁邊的窗格綱要」窗格中，找到Mutation鍵入，然後publish(...): Channel欄位，然後按一下貼附。

創建一個沒有資料來源並將其命名PageDataSource。將其附加到您的解析器。

使用以下代碼片段添加您的解析器實現：

```
export function request(ctx) {
  return { payload: ctx.args };
}

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

確保您創建了解析器並保存所做的更改。

傳送和訂閱訊息

若要讓用戶端接收訊息，必須先訂閱收件匣。

在查詢」窗格中，執行SubscribeToData訂閱：

```
subscription SubscribeToData {
    subscribe(name:"channel") {
        name
        data
    }
}
```

訂閱者將收到消息，每當publish突變被調用，但只有當消息被發送到channel訂閱。讓我們試試這個查詢窗格。當您的訂閱仍在主控台中執行時，請開啟另一個主控台，然後在查詢窗格：

> **Note**
>
> 我們在這個例子中使用有效的 JSON 字符串。

```
mutation PublishData {
    publish(data: "{\"msg\": \"hello world!\"}", name: "channel") {
        data
        name
    }
}
```

結果看起來像這樣：

```
{
  "data": {
    "publish": {
      "data": "{\"msg\":\"hello world!\"}",
      "name": "channel"
    }
  }
}
```

我們剛剛演示了使用本地解析器，通過發布消息並接收它而不離開AWS AppSync服務。

教學課程：結合 GraphQL 解析器

GraphQL 結構描述中的解析程式與欄位具有一對一的關係，以及高度的彈性。由於資料來源設定在獨立於結構描述的解析器上，因此您可以透過不同的資料來源來解析或操作 GraphQL 類型，從而使您能夠混合和比對結構描述以最符合您的需求。

下列案例示範如何混合和比對結構描述中的資料來源。在開始之前，您應該熟悉為以下項目配置資料來源和解析器AWS Lambda、亞馬遜和亞馬遜OpenSearch服務。

示例模式

下面的模式有一個類型`Post`與三`Query`和`Mutation`每個操作：

```
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
}
```

在此範例中，您總共有六個解析器，每個解析器都需要資料來源。解決此問題的一種方法是將這些問題連接到稱為的單個 Amazon DynamoDB 表格`Posts`，其中`AllPost`字段運行掃描並且`searchPosts`欄位執行查詢 (請參閱JavaScript解析器函數參考). 不過，您不僅限於 Amazon DynamoDB；不同的資料來源，例如 Lambda 或OpenSearch服務存在以滿足您的業務需求。

透過解析器變更資料

您可能需要從未直接支援的協力廠商資料庫傳回結果AWS AppSync資料來源。在將資料傳回給 API 用戶端之前，您可能還必須對資料執行複雜的修改。這可能是由於資料類型的格式不正確，例如用戶端上的時間戳記差異，或向後相容性問題的處理所致。在這種情況下，連接AWS Lambda作為您的資料來源的功能AWS AppSyncAPI 是適當的解決方案。為了說明目的，在下面的例子中，AWS Lambda函數操縱從第三方數據存儲獲取的數據：

```
export const handler = (event, context, callback) => {
    // fetch data
    const result = fetcher()

    // apply complex business logic
    const data = transform(result)

    // return to AppSync
```

```
    return data
};
```

這是一個非常有效的 Lambda 函式，可以連結至 GraphQL 結構描述中的 `AllPost` 欄位，如此，任何傳回所有結果的查詢，都會取得支持票和不支持票票數的隨機數量。

動態支援和OpenSearch服務

對於某些應用程式，您可以對 DynamoDB 執行突變或簡單的查閱查詢，並讓背景程序將文件傳輸到OpenSearch服務。你可以簡單地附上`searchPosts`解析器OpenSearch使用 GraphQL 查詢服務資料來源並傳回搜尋結果 (來自 DynamoDB 中產生的資料)。將高級搜索操作添加到應用程序（例如關鍵字，模糊單詞匹配甚至地理空間查找）時，這可能非常強大。從 DynamoDB 傳輸資料可透過 ETL 程序完成，或者，您也可以使用 Lambda 從 DynamoDB 串流。

若要開始使用這些特定資料來源，請參閱我們的DynamoDB和拉姆達教程。

例如，使用上一個教學課程中的結構描述，下列變異會將項目新增至 DynamoDB：

```
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
  }
}
```

這會將資料寫入 DynamoDB，然後透過 Lambda 將資料串流至亞馬遜OpenSearch服務，然後您可以使用該服務按不同字段搜索帖子。例如，由於數據在亞馬遜OpenSearch服務中，您可以使用自由格式的文本來搜索作者或內容字段，即使是空格，如下所示：

```
query searchName{
```

```
    searchAuthor(name:"   Nadia   "){
        id
        title
        content
    }
}

---------- or ----------

query searchContent{
    searchContent(text:"test"){
        id
        title
        content
    }
}
```

由於資料會直接寫入 DynamoDB，因此您仍然可以使用`allPost{...}`和`getPost{...}`查詢。此堆疊會針對 DynamoDB 串流使用下列範例程式碼：

> **Note**
>
> 這個 Python 代碼是一個例子，並不意味著在生產代碼中使用。

```
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.'
```

然後，您可以使用 DynamoDB 串流將其附加到主索引鍵為id，而對 DynamoDB 來源的任何變更都會串流到您的OpenSearch服務網域。如需進行這項設定的詳細資訊，請參閱 DynamoDB 串流文件。

教程：亞馬遜OpenSearch服務解析器

AWS AppSync支持使用亞馬遜OpenSearch從您自己佈建的網域提供服務AWS帳戶，前提是它們不存在於 VPC 內部。在佈建網域之後，您可以使用資料來源來連線到這些網域，此時您可以在結構描述中設定解析程式，來進行 GraphQL 操作，例如查詢、變動和訂閱。本教學課程將會逐步說明一些常見的範例。

如需詳細資訊，請參閱我們的JavaScript解析器函數參考OpenSearch。

創建一個新的OpenSearch服務網域

要開始使用本教程，您需要一個現有的OpenSearch服務網域。如果尚未擁有，您可以使用下列範例。請注意，最多可能需要 15 分鐘的時間OpenSearch要建立的服務網域，然後才能繼續整合AWS AppSync資料來源。

```
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
```

您可以啟動以下內容AWS CloudFormation在美國西部 2 (奧勒岡) 區域的堆疊AWS帳戶：

Launch Stack

設定資料來源OpenSearch服務

之後OpenSearch服務域已創建，導航到您的AWS AppSync圖形 SQL API 並選擇資料來源標籤。選擇建立資料來源並輸入資料來源的易記名稱，例如」*oss*」. 然後，選擇亞馬遜OpenSearch域為了資料來源類型，選擇適當的地區，您應該會看到您的OpenSearch列出的服務網域。選擇之后，您可以創建一個新角色，AWS AppSync將指派適當角色的權限，或者您可以選擇具有下列內嵌原則的現有角色：

```
{
    "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/*"
            ]
        }
    ]
}
```

您還需要與以下方式建立信任關係AWS AppSync對於該角色：

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

```
    ]
}
```

此外，OpenSearch服務域有自己的存取政策您可以通過亞馬遜修改OpenSearch服務主控台。您必須使用適當的動作和資源來新增與下列類似的策略OpenSearch服務網域。請注意，主要將是AWS AppSync數據源角色，如果您讓控制台創建它，則可以在 IAM 控制台中找到該角色。

```
{
    "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/*"
        }
    ]
}
```

連接解析器

現在，數據源已連接到您的OpenSearch服務域，您可以使用解析器將其連接到 GraphQL 架構，如以下示例所示：

```
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
}
```

請注意，有使用者定義的 Post 類型與 id 欄位。在以下示例中，我們假設有一個過程（可以自動化）將此類型放入您的OpenSearch服務網域，會對應至的路徑根目錄/post/_doc哪裡post是索引。從此根路徑，您可以執行個別文件搜尋，使用萬用字元搜尋/id/post*，或使用路徑進行多文件搜尋/post/_search。例如，如果您有另一種稱為的類型User，您可以在名為的新索引下為文件索引user，然後執行搜尋路徑的/user/_search。

從綱要中的編輯器AWS AppSync控制台，修改前面的Posts要包含的綱要searchPosts查詢：

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

儲存結構描述。在解析器窗格中，尋找searchPosts並選擇貼附。選擇您的OpenSearch服務數據源並保存解析器。使用下面的代碼片段更新解析器的代碼：

```
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)
}
```

這假設前面的結構描述具有已在中編製索引的文檔OpenSearch下的服務post欄位。如果您的資料結構不同，則需要相應地進行更新。

修改您的搜尋

前面的解析器請求處理程序執行所有記錄的簡單查詢。假設您想要根據特定的作者來進行搜尋。此外，假設您希望該作者成為 GraphQL 查詢中定義的參數。在綱要的編輯AWS AppSync主控台，新增allPostsByAuthor查詢：

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

在解析器窗格中，尋找allPostsByAuthor並選擇貼附。選擇合適的OpenSearch服務數據源並使用以下代碼：

```
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)
}
```

請注意，body 會填入 author 欄位的查詢詞語，此欄位會做為引數，從用戶端直接傳來。或者，您可以使用預先填入的資訊，例如標準文字。

將資料新增至OpenSearch服務

您可能需要將數據添加到OpenSearch由於 GraphQL 突變而產生的服務網域。這是一個具有搜尋和其他用途的強大機制。因為您可以使用 GraphQL 訂閱使您的數據實時，它可以作為一種機制，用於通知客戶更新數據OpenSearch服務網域。

返回綱要頁面中的AWS AppSync控制台並選擇貼附為addPost()突變。選擇OpenSearch服務數據源再次使用以下代碼：

```
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
}
```

像以前一樣，這是一個如何結構數據的示例。如果您有不同的欄位名稱或索引，則需要更新path和body。這個例子還顯示了如何使用context.arguments，也可以寫成ctx.args，在您的請求處理程序中。

擷取單一文件

最後，如果您想要使用getPost(id:ID)在結構描述中查詢以返回單個文檔，請在綱要的編輯AWS AppSync控制台和選擇貼附。選擇OpenSearch服務數據源再次使用以下代碼：

```
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
}
```

執行查詢和突變

您現在應該可以針對您的OpenSearch服務網域。導覽至查詢的索引標籤AWS AppSync控制台並添加新記錄：

```
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
       )
}
```

您將在右側看到突變的結果。同樣，您現在可以運行`searchPosts`查詢您的OpenSearch服務網域：

```
query search {
    searchPosts {
        id
        title
        author
        content
    }
}
```

最佳實務

- OpenSearch服務應該用於查詢數據，而不是作為您的主數據庫。您可能想要使用OpenSearch與亞馬遜動態 B 結合使用的服務，如下所述結合圖形 SQL 解析器。
- 只允許存取您的網域AWS AppSync存取叢集的服務角色。
- 您可以在開發時少量使用最低成本的叢集，然後在進入正式生產時，轉移到具有高可用性 (HA) 的較大型叢集。

教學課程：交易解析器

AWS AppSync支援在單一區域中的一或多個表格中使用 Amazon DynamoDB 交易操作。支援的操作包括 `TransactGetItems` 和 `TransactWriteItems`。透過使用這些功能AWS AppSync，您可以執行下列工作：

- 在單個查詢中傳遞密鑰列表並從表中返回結果
- 在單個查詢中讀取一個或多個表中的記錄
- 將交易中的記錄寫入中的一個或多個表all-or-nothing方式
- 滿足某些條件時執行作業事件

許可

像其他解析器一樣，您需要在AWS AppSync並創建一個角色或使用現有的角色。由於交易操作需要 DynamoDB 表上的不同權限，因此您需要授與已設定的角色權限以進行讀取或寫入動作：

```
{
    "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**
>
> 角色與中的資料來源相關聯AWS AppSync，並針對資料來源叫用欄位的解析器。設定為針對 DynamoDB 擷取的資料來源只會指定一個表格，以便簡化組態。因此，在單一解析程式中針對多個資料表執行交易操作時 (這是一項更為進階的工作)，您必須授予權限給該資料來源上的角色，以存取解析程式將會與其互動的所有資料表。這會在上述 IAM 政策的 Resource (資源) 欄位中完成。對表的事務調用的配置是在解析器代碼，我們在下面描述完成。

資料來源

為簡單起見，我們將針對本教學課程中使用的所有解析程式，使用相同的資料來源。

我們將有兩個表稱為儲蓄賬戶和檢查帳戶，兩者都與`accountNumber`作為一個分區鍵，和交易歷史具有表`transactionId`作為分區鍵。您可以使用下面的 CLI 命令來創建表格。確保更換`region`與您的區域。

使用 CLI 的功能

```
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主控台，在資料來源，建立新的 DynamoDB 資料來源並將其命名TransactTutorial。选择儲蓄賬戶作為表格（儘管使用事務時特定的表無關緊要）。選擇此選項可建立新角色和資料來源。您可以檢閱資料來源組態，以查看產生的角色名稱。在 IAM 主控台中，您可以新增內嵌政策，以允許資料來源與所有資料表互動。

取代region和accountID使用您的地區和帳戶 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/*"
            ]
        }
    ]
}
```

交易

在此範例中，內容是傳統的銀行交易，我們將使用 TransactWriteItems 以進行下列操作：

- 從存款帳戶轉帳至支票帳戶
- 為每筆交易產生新的交易記錄

然後，我們會使用 TransactGetItems 擷取存款帳戶和支票帳戶中的詳細資料。

定義 GraphQl 結構描述，如下所示：

```
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-填充帳戶

為了在帳戶之間轉帳，我們需要在表格中填入詳細資料。我們將使用 GraphQL 操作 `Mutation.populateAccounts` 來執行此作業。

在「架構」部分中，單擊貼附旁邊的`Mutation.populateAccounts`操作。選擇合適的`TransactTutorial`資料來源並選擇創建。

現在使用下面的代碼：

```
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 }
}
```

儲存解析程式並導覽至查詢的部分AWS AppSync控制台來填充帳戶。

執行下列的變動：

```
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
    }
```

```
  }
}
```

我們在一個突變中填充了三個儲蓄帳戶和三個支票帳戶。

使用 DynamoDB 主控台來驗證資料是否同時顯示在儲蓄賬戶和檢查帳戶表格。

TransactWriteItems-轉賬

將解析器附加到`transferMoney`突變與下面的代碼。對於每次轉賬，我們都需要支票帳戶和儲蓄帳戶的成功修飾符，並且我們需要跟踪交易中的轉賬。

```
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 }
}
```

現在，導航到查詢的部分AWS AppSync控制台並執行轉移貨幣突變如下：

```
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
    }
  }
}
```

我們在一個突變中發送了三筆銀行交易。使用 DynamoDB 主控台來驗證資料是否顯示在儲蓄賬戶,檢查帳戶，以及交易歷史表格。

TransactGetItems-檢索帳戶

為了在單個交易請求中從儲蓄和支票帳戶中檢索詳細信息，我們將在`Query.getAccounts`我們架構上的 GraphQL 作業。选择貼附，選擇相同`TransactTutorial`在自學課程開始時建立的資料來源。使用下列程式碼：

```
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 }
}
```

儲存解析程式並導覽至查詢的部分AWS AppSync控制台。為了檢索儲蓄和支票帳戶，請執行以下查詢：

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
      username
      balance
    }
  }
}
```

我們已經成功展示了使用 DynamoDB 交易的使用AWS AppSync。

教學課程：批次解析器

AWS AppSync支援在單一區域中的一個或多個表格中使用 Amazon DynamoDB 批次操作。支援的操作包括 `BatchGetItem`、`BatchPutItem` 和 `BatchDeleteItem`。透過使用這些功能AWS AppSync，您可以執行下列工作：

- 在單個查詢中傳遞鍵列表並從表中返回結果
- 在單個查詢中讀取一個或多個表中的記錄
- 將記錄批量寫入一個或多個表
- 有條件地寫入或刪除可能有關係的多個表中的記錄

批次處理作業AWS AppSync與非批處理操作有兩個關鍵區別：

- 資料來源角色必須具有解析程式將存取之所有資料表的權限。
- 解析器的表格規範是請求對象的一部分。

單表批次

若要開始使用，讓我們建立新的 GraphQL API。在AWS AppSync控制台，選擇建立 API,圖形 SQL API，以及從頭開始設計。為您的 API 命名BatchTutorial API，選擇下一步，並在指定圖形 SQL 資源步驟，選擇稍後建立圖形 SQL 資源然後按一下下一步。檢閱您的詳細資料並建立 API。前往綱要頁面並粘貼以下模式，注意到對於查詢，我們將在 ID 列表中傳遞：

```
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]
}
```

儲存您的結構描述並選擇建立資源在頁面頂部。選擇使用現有類型，然後選擇Post類型。命名您的表Posts。請確定主索引鍵設定為id，取消選取自動產生圖形 SQL（您將提供自己的代碼），然後選擇創建。為了讓你開始，AWS AppSync建立新的 DynamoDB 表格，以及使用適當角色連接至表格的資料來源。不過，您仍需要新增至角色的幾個權限。前往資料來源頁面並選擇新的資料來源。下選取現有角色，您會注意到該表格已自動建立角色。注意角色（應該看起來像appsync-ds-ddb-aaabbbcccddd-Posts），然後轉到 IAM 控制台（https://console.aws.amazon.com/iam/). 在 IAM 主控台中，選擇角色，然後從表格中選擇您的角色。在您的角色中，權限原則，點擊」+「在策略旁邊（應該具有與角色名稱相似的名稱）。選擇編輯出現策略時，位於可折疊的頂部。您需要為策略添加批處理權限，特別是dynamodb:BatchGetItem和dynamodb:BatchWriteItem。它看起來像這樣：

```
{
    "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:…"
            ]
        }
    ]
}
```

選擇下一步，然後儲存變更。您的政策現在應該允許批次處理。

回到了AWS AppSync主控台，移至綱要頁面並選擇貼附旁邊的Mutation.batchAdd欄位。使用建立您的解析程式Posts表格做為資料來源。在程式碼編輯器中，以下列程式碼片段取代處理常式。此程式碼片段會自動取得 GraphQL 中的每個項目input PostInput鍵入並構建一個地圖，這是需要的BatchPutItem操作：

```
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;
}
```

導覽至查詢的頁面AWS AppSync控制台並運行以下內容batchAdd突變：

```
mutation add {
    batchAdd(posts:[{
            id: 1 title: "Running in the Park"},{
            id: 2 title: "Playing fetch"
        }]){
            id
            title
    }
}
```

您應該會看到畫面上列印的結果；這可透過檢視 DynamoDB 主控台掃描寫入Posts表。

接下來，重複附加解析器的過程，但對於Query.batchGet欄位使用Posts表格做為資料來源。用下面的代碼替換處理程序。這會自動擷取 GraphQL ids:[] 類型的每個項目，並建置 BatchGetItem 作業所需的對應圖：

```
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;
```

```
}
```

現在，回到查詢的頁面AWS AppSync控制台並運行以下內容batchGet查詢：

```
query get {
    batchGet(ids:[1,2,3]){
        id
        title
    }
}
```

這應該會針對您先前所新增的兩個 id 值，傳回其結果。注意一個null已傳回的值id值為3。這是因為你沒有記錄Posts具有該值的表。還要注意的是AWS AppSync以與傳遞給查詢的鍵相同的順序返回結果，這是一個附加功能AWS AppSync代表您執行。所以，如果你切換到batchGet(ids:[1,3,2])，您會看到順序已變更。您也會知道哪個 id 傳回 null 值。

最後，再將一個解析器附加到Mutation.batchDelete欄位使用Posts表格做為資料來源。用下面的代碼替換處理程序。這會自動擷取 GraphQL ids:[] 類型的每個項目，並建置 BatchGetItem 作業所需的對應圖：

```
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;
}
```

現在，回到查詢的頁面AWS AppSync控制台並運行以下內容batchDelete突變：

```
mutation delete {
    batchDelete(ids:[1,2]){ id }
```

```
}
```

id 1 和 2 的記錄現在應已刪除。如果重新執行先前的 batchGet() 查詢，這些應該會傳回 null。

多表批次

AWS AppSync還可讓您跨表執行批次作業。來試試建置更複雜的應用程式。想像一下，我們正在構建一個寵物健康應用程序，其中傳感器報告寵物的位置和體溫。感應器是由電池供電，而且每隔幾分鐘就會試著連線到網路。當傳感器建立連接時，它將其讀數發送給我們的AWS AppSyncAPI。觸發條件接著就會分析資料，然後將儀表板呈現給寵物的主人。讓我們著重介紹感應器與後端資料存放區之間的互動。

在AWS AppSync控制台，選擇建立 API,圖形 SQL API，以及從頭開始設計。為您的 API 命名MultiBatchTutorial API，選擇下一步，並在指定圖形 SQL 資源步驟，選擇稍後建立圖形 SQL 資源然後按一下下一步。檢閱您的詳細資料並建立 API。前往綱要頁面並貼上並儲存下列綱要：

```
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
}
```

我們需要建立兩個表：

- locationReadings將存儲傳感器位置讀數。
- temperatureReadings將存儲傳感器溫度讀數。

這兩個表將共享相同的主鍵結構：sensorId (String)作為分區鍵和timestamp (String)作為排序關鍵字。

選擇建立資源在頁面頂部。選擇使用現有類型，然後選擇locationReadings類型。命名您的表locationReadings。請確定主索引鍵設定為sensorId和排序鍵timestamp。取消選取自動產生圖形 SQL（您將提供自己的代碼），然後選擇創建。重複此過程temperatureReadings使用temperatureReadings作為類型和表名。使用與上述相同的按鍵。

您的新表格將包含自動產生的角色。您仍需要將幾個權限新增至這些角色。前往資料來源頁面並選擇locationReadings。下選取現有角色，您可以看到該角色。注意角色（應該看起來

像appsync-ds-ddb-aaabbbcccddd-locationReadings），然後轉到 IAM 控制台（https://console.aws.amazon.com/iam/). 在 IAM 主控台中，選擇角色，然後從表格中選擇您的角色。在您的角色中，權限原則，點擊」+「在策略旁邊（應該具有與角色名稱相似的名稱）。選擇編輯出現策略時，位於可折疊的頂部。您必須將權限新增至此原則。它看起來像這樣：

```
{
    "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/*"
            ]
        }
    ]
}
```

選擇下一步，然後儲存變更。重複此過程temperatureReadings使用上述相同政策程式碼片段的資料來源。

BatchPutItem-記錄傳感器讀數

感應器必須能夠在連線到網際網路之後傳送其讀數。感應器將使用 GraphQL 欄位 Mutation.recordReadings 這個 API 來執行此項動作。我們需要添加一個解析器到這個字段。

在AWS AppSync控制台綱要頁面上，選取貼附旁邊的Mutation.recordReadings欄位。在下一個屏幕上，使用創建解析器locationReadings表格做為資料來源。

建立解析程式之後，請在編輯器中使用下列程式碼取代處理常式。這BatchPutItem操作允許我們指定多個表：

```
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
}
```

進行批次操作時，呼叫可能會同時傳回錯誤和結果。在這種情況中，我們可以隨意進行一些額外的錯誤處理。

> **Note**
>
> 的使用utils.appendError()類似於util.error()，主要區別在於它不會中斷請求或響應處理程序的評估。相反，它表示該字段出現錯誤，但允許評估處理程序，從而將數據返回給調用者。我們建議您使用utils.appendError()當您的應用程序需要返回部分結果時。

儲存解析程式並導覽至查詢頁面中的AWS AppSync控制台。我們現在可以發送一些傳感器讀數。

執行下列的變動：

```
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
    }
  }
}
```

我們在一個突變中發送了十個傳感器讀數，讀數分為兩個表。使用 DynamoDB 主控台來驗證資料是否同時顯示在`locationReadings`和`temperatureReadings`表。

BatchDeleteItem-刪除傳感器讀數

同樣，我們還需要能夠刪除一批傳感器讀數。讓我們使用 `Mutation.deleteReadings` GraphQL 欄位來進行這項動作。在AWS AppSync控制台綱要頁面上，選取貼附旁邊的`Mutation.deleteReadings`欄位。在下一個屏幕上，使用創建解析器`locationReadings`表格做為資料來源。

建立解析程式之後，請使用下列程式碼片段取代程式碼編輯器中的處理常式。在此解析器中，我們使用了一個輔助函數映射器來提取sensorId和timestamp從提供的輸入。

```
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
}
```

儲存解析程式並導覽至查詢頁面中的AWS AppSync控制台。現在，讓我們刪除幾個傳感器讀數。

執行下列的變動：

```
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**
>
> 不同於 DeleteItem 操作，回應中未傳回完整刪除的項目。只會傳回已傳遞的索引鍵。若要深入瞭解，請參閱BatchDeleteItem在JavaScript解析器函數參考。

透過 DynamoDB 主控台驗證這兩個讀數是否已
從locationReadings和temperatureReadings表。

BatchGetItem-檢索讀數

我們的應用程序的另一個常見操作是在特定時間點檢索傳感器的讀數。試試將解析程式連接到結構描述中的 Query.getReadings GraphQL 欄位。在AWS AppSync控制台綱要頁面上，選取貼附旁邊的Query.getReadings欄位。在下一個屏幕上，使用創建解析器locationReadings表格做為資料來源。

讓我們使用下面的代碼：

```
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' })),
 ]
}
```

儲存解析程式並導覽至查詢頁面中的AWS AppSync控制台。現在，讓我們檢索我們的傳感器讀數。

執行下列的查詢：

```
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
    }
  }
}
```

我們已經成功展示了使用 DynamoDB 批次作業的使用方式AWS AppSync。

錯誤處理

在AWS AppSync，資料來源作業有時會傳回部分結果。我們將會使用部分結果這個詞彙，來表示操作的輸出包含一些資料和錯誤的情況。因為錯誤處理本質上是應用程序特定的，AWS AppSync讓您有機會處理回應處理常式中的錯誤。如果發生解析程式呼叫錯誤，在文字內容中會出現 `ctx.error`。呼叫錯誤一律包含訊息和類型，可做為 `ctx.error.message` 和 `ctx.error.type` 屬性存取。在響應處理程序中，您可以通過三種方式處理部分結果：

1. 只需返回數據即可吞下調用錯誤。
2. 引發錯誤（使用`util.error(...)`）通過停止處理程序評估，該評估不會返回任何數據。
3. 附加錯誤（使用`util.appendError(...)`），並返回數據。

讓我們使用 DynamoDB 批次作業來示範上述三個要點中的每一個。

DynamoDB 批次操作

使用 DynamoDB 批次操作時，批次作業就有可能部分完成。也就是說，請求的項目或索引鍵可以有一些尚未處理完成。如果AWS AppSync無法完成批次、未處理的項目，且會在前後關聯上設定呼叫錯誤。

我們將會使用 `Query.getReadings` 欄位組態 (取自於本教學課程先前區段所說明的 `BatchGetItem` 操作) 來建置錯誤處理功能。這次，讓我們假設在執行 `Query.getReadings` 欄位時，`temperatureReadings` DynamoDB 資料表用盡了佈建的輸送量。提出了一個`ProvisionedThroughputExceededException`在第二次嘗試期間AWS AppSync以處理批次中的剩餘元素。

下列 JSON 代表 DynamoDB 批次叫用之後，但在呼叫回應處理常式之前的序列化內容：

```
{
  "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": []
}
```

對於內容的幾個注意事項：

- 調用錯誤已在上下文中設置`ctx.error`由AWS AppSync，並且錯誤類型已設定為`DynamoDB:ProvisionedThroughputExceededException`。
- 每個表格對應結果`ctx.result.data`即使存在錯誤。
- 保留未處理的金鑰可在以下位置取得：`ctx.result.data.unprocessedKeys`。在這裡，AWS AppSync由於表格輸送量不足，因此無法使用索引鍵擷取項目 (感測器:1，時間戳記：17:21:05.000 + 08:00)。

> **Note**
>
> 如果是 `BatchPutItem`，則為 `ctx.result.data.unprocessedItems`。如果是 `BatchDeleteItem`，則為 `ctx.result.data.unprocessedKeys`。

讓我們用三種不同的方式來處理這項錯誤。

1. 抑制呼叫錯誤

傳回資料而不處理呼叫錯誤，可有效地抑制錯誤，讓指定 GraphQL 欄位的結果一律成功。

我們編寫的代碼很熟悉，只關注結果數據。

響應處理

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

圖形 SQL 回應

```
{
  "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
      }
    ]
  }
}
```

錯誤回應中不會加入任何錯誤，因為只處理了資料。

2. 引發錯誤以中止響應處理程序執行

從用戶端的角度來看，當部分失敗應該被視為完全失敗時，您可以中止回應處理常式執行，以防止傳回資料。`util.error(...)` 公用程式方法可確實完成這項動作。

回應處理程式碼

```
export function response(ctx) {
  if (ctx.error) {
    util.error(ctx.error.message, ctx.error.type, null,
 ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}
```

圖形 SQL 回應

```
{
  "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 DynamoDB 批次作業可能已經傳回一些結果，我們還是選擇丟出錯誤，如此 `getReadings` GraphQL 欄位為 null，而錯誤也會加入到 GraphQL 回應的 errors (錯誤) 區塊。

3. 附加錯誤，以同時傳回資料和錯誤

在某些情況中，為提供更好的使用者體驗，應用程式可以傳回部分結果，並通知其用戶端有未處理的項目。用戶端可以決定是否要重試，或是轉譯錯誤後傳回給最終使用者。該`util.appendError(...)`是一種公用程式方法，可讓應用程式設計工具在內容上附加錯誤，而不會干擾回應處理常式的評估，藉此啟用此行為。評估響應處理程序後，AWS AppSync將通過將任何上下文錯誤附加到 GraphQL 響應的錯誤塊來處理它們。

回應處理程式碼

```
export function response(ctx) {
  if (ctx.error) {
```

```
    util.appendError(ctx.error.message, ctx.error.type, null,
 ctx.result.data.unprocessedKeys);
  }
  return ctx.result.data;
}
```

我們轉發了調用錯誤和unprocessedKeysGraphQL 回應的錯誤區塊內的元素。該getReadings字段也從中返回部分數據locationReadings表，你可以在下面的響應中看到。

圖形 SQL 回應

```
{
  "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. (...)"
    }
  ]
}
```

教學課程：HTTP 解析器

AWS AppSync可讓您使用支援的資料來源 (也就是AWS Lambda, 亞馬遜, 亞馬遜OpenSearch服務或亞馬遜極光) 除了用於解析 GraphQL 字段的任意 HTTP 端點之外，還可以執行各種操作。在您的 HTTP 端點可供使用之後，您可以使用資料來源連線到這些端點。然後，您可以在結構描述中設定解析程式以執行 GraphQL 操作，例如查詢、變動和訂閱。本教學將逐步說明一些常見的範例。

在本教學中，您可以使用 REST API (使用亞馬遜 API 閘道和 Lambda 建立) 搭配AWS AppSync圖形 SQL 端點。

建立 REST API

您可以使用以下 AWS CloudFormation 範本來設定適用於本教學的 REST 端點：

AWS CloudFormation 堆疊會執行下列步驟：

1. 設定 Lambda 函數，其中包含您微服務的商業邏輯。
2. 使用以下端點/方法/內容類型組合設置 API 閘道 REST API：

API 資源路徑	HTTP 方法	已支援的內容類型
/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	DELETE	application/json

建立您的圖形 SQL API

若要在中建立圖形 SQL APIAWS AppSync:

1. 打開AWS AppSync控制台和選擇建立 API。
2. 選擇圖形 SQL API然後選擇從頭開始設計。選擇 下一步。
3. 針對 API 名稱，輸入 `UserData`。選擇 下一步。
4. 選擇 `Create GraphQL resources later`。選擇 下一步。
5. 查看您的輸入並選擇建立 API。

該AWS AppSync主控台會使用 API 金鑰驗證模式，為您建立新的 GraphQL API。您可以使用主控台進一步設定 GraphQL API 並執行要求。

建立圖形 SQL 結構描述

現在您有一個 GraphQL API，讓我們來建立 GraphQL 結構描述吧。在綱要中的編輯器AWS AppSync控制台，使用下面的代碼片段：

```
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
}
```

設定您的 HTTP 資料來源

若要設定您的 HTTP 資料來源，請執行以下作業：

1. 在資料來源在您的頁面AWS AppSync圖形 SQL API, 選擇建立資料來源。
2. 輸入資料來源的名稱，例如HTTP_Example。
3. 在資料來源類型，選擇端點。
4. 將端點設定為在教學課程開始時建立的 API 閘道端點。如果您導覽至 Lambda 主控台並在下方找到您的應用程式，則可以找到堆疊產生的端點應用。在應用程序的設置中，您應該看到一個API 端點這將是你的終點AWS AppSync。請確定您沒有將階段名稱包含為端點的一部分。例如，如果您的端點是https://aaabbbcccd.execute-api.us-east-1.amazonaws.com/v1，您可以輸入https://aaabbbcccd.execute-api.us-east-1.amazonaws.com。

> **Note**
>
> 目前，僅支援公用端點AWS AppSync。
> 有關被認可的認證機構的更多信息AWS AppSync服務，請參閱憑證授權單位 (CA) 認可AWS AppSync適用於 HTTPS 端點。

設定解析程式

在此步驟中，您將 HTTP 資料來源連線到getUser和addUser查詢。

若要設定getUser解析器：

1. 在您的AWS AppSync圖形 SQL 應用程式介面，選擇綱要標籤。
2. 在右邊綱要編輯器，在解析器窗格和下查詢輸入，找到getUser欄位並選擇貼附。
3. 將解析器類型保持為Unit和運行時APPSYNC_JS。
4. 在資料來源名稱」中，選擇您之前建立的 HTTP 端點。

5. 選擇 建立 。
6. 在解析器代碼編輯器，添加以下代碼片段作為請求處理程序：

```
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. 添加以下代碼片段作為您的響應處理程序：

```
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. 選擇 Query (查詢) 標籤，然後執行以下查詢：

```
query GetUser{
    getUser(id:1){
        id
        username
    }
}
```

這應該會傳回以下回應：

```
{
```

```
    "data": {
        "getUser": {
            "id": "1",
            "username": "nadia"
        }
    }
}
```

若要設定addUser解析器：

1. 選擇 Schema (結構描述) 標籤。
2. 在右邊綱要編輯器，在解析器窗格和下查詢輸入，找到addUser欄位並選擇貼附。
3. 將解析器類型保持為Unit和運行時APPSYNC_JS。
4. 在資料來源名稱」中，選擇您之前建立的 HTTP 端點。
5. 選擇 建立 。
6. 在解析器代碼編輯器，添加以下代碼片段作為請求處理程序：

```
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. 添加以下代碼片段作為您的響應處理程序：

```
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. 選擇 Query (查詢) 標籤，然後執行以下查詢：

```
mutation addUser{
    addUser(userInput:{
        id:"2",
        username:"shaggy"
    }){
        id
        username
    }
}
```

如果您執行`getUser`再次查詢，它應該返回以下響應：

```
{
    "data": {
        "getUser": {
        "id": "2",
        "username": "shaggy"
        }
    }
}
```

調用AWS服務

您可以使用 HTTP 解析器來設置一個圖形 SQL API 接口AWS服務。將 HTTP 要求傳送至AWS必須簽署簽名版本 4 進程這樣AWS可以識別誰發送給他們。AWSAppSync當您將 IAM 角色與 HTTP 資料來源建立關聯時，會代表您計算簽名。

您提供兩個額外的組件來調用AWS使用 HTTP 解析器的服務：

- 具有呼叫權限的 IAM 角色AWS服務 API
- 資料來源中的簽署組態

例如，如果您想要呼叫ListGraphqlApis操作使用 HTTP 解析器，您首先建立 IAM 角色那個AWS AppSync假設附加了以下策略：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "appsync:ListGraphqlApis"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

接下來，為下列項目建立 HTTP 資料來源AWS AppSync。在這個例子中，你打電話AWS AppSync位於美國西部 (奧勒岡) 區域。在名為 `http.json` 的檔案中設定以下 HTTP 組態，包含簽署區域和服務名稱：

```
{
    "endpoint": "https://appsync.us-west-2.amazonaws.com/",
    "authorizationConfig": {
        "authorizationType": "AWS_IAM",
        "awsIamConfig": {
            "signingRegion": "us-west-2",
            "signingServiceName": "appsync"
        }
    }
}
```

然後，使用AWS CLI以建立具有關聯角色的資料來源，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>
```

將解析器附加到模式中的字段時，請使用以下請求映射模板調用AWS AppSync:

```
{
    "version": "2018-05-29",
    "method": "GET",
```

```
    "resourcePath": "/v1/apis"
}
```

當您針對此資料來源執行 GraphQL 查詢時，AWS AppSync使用您提供的角色簽署請求，並在請求中包含簽名。該查詢返回一個列表AWS AppSync您的帳戶中的圖形 SQL APIAWS區域。

教學課程：使用資料 API 的 Aurora

AWS AppSync提供資料來源，以針對已啟用資料 API 的 Amazon Aurora 叢集執行 SQL 陳述式。您可以使用AWS AppSync解析器針對具有 GraphQL 查詢、突變和訂閱的資料 API 執行 SQL 陳述式。

> **Note**
>
> 此教學會使用 US-EAST-1 區域。

建立叢集

將 Amazon RDS 資料來源新增至之前AWS AppSync，請先在 Aurora 無伺服器叢集上啟用資料 API。您還必須使用配置密碼AWS Secrets Manager。若要建立 Aurora 無伺服器叢集，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
```

這會傳回叢集的 ARN。您可以使用以下命令檢查叢集的狀態：

```
aws rds describe-db-clusters \
    --db-cluster-identifier appsync-tutorial \
    --query "DBClusters[0].Status"
```

通過AWS Secrets Manager控制台或使AWS CLI用輸入文件（例如使用USERNAME和COMPLEX_PASSWORD上一步）創建密鑰：

```
{
    "username": "USERNAME",
```

```
    "password": "COMPLEX_PASSWORD"
}
```

將其作為參數傳遞給 CLI：

```
aws secretsmanager create-secret \
    --name appsync-tutorial-rds-secret \
    --secret-string file://creds.json
```

這會傳回秘密的 ARN。請記下 Aurora 無伺服器叢集的 ARN，以及稍後在主控台中建立資料來源時使用的AWS AppSync密碼。

啟用資料 API

叢集狀態變更為後`available`，請遵循 Amazon RDS 文件啟用資料 API。在將資料 API 新增為資料來源之前，必須先啟用AWS AppSync資料 API。您也可以使用以下命令啟用資料 APIAWS CLI：

```
aws rds modify-db-cluster \
    --db-cluster-identifier appsync-tutorial \
    --enable-http-endpoint \
    --apply-immediately
```

創建數據庫和表

啟用資料 API 之後，請`aws rds-data execute-statement`使用AWS CLI. 這可確保您的 Aurora 無伺服器叢集在將叢集新增至 AWS AppSync API 之前已正確設定。首先，使用以下參數創建一個 TESTDB `--sql` 數據庫：

```
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\""
```

如果執行時沒有任何錯誤，請使用以下`create table`指令新增兩個資料表：

```
 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);'
```

如果一切都沒有問題執行，您現在可以將叢集新增為 API 中的資料來源。

建立 GraphQL 結構描述

現在您的 Aurora 無伺服器資料 API 已使用已設定的表格執行，我們將建立 GraphQL 結構描述。您可以手動執行此操作，但可AWS AppSync讓您使用 API 創建嚮導從現有數據庫導入表配置來快速開始。

若要開始：

1. 在AWS AppSync主控台中，選擇「建立 API」，然後選擇「從 Amazon Aurora 叢集開始」。
2. 指定 API 詳細信息，例如 API 名稱，然後選擇數據庫以生成 API。
3. 選擇您的資料庫。如有需要，請更新區域，然後選擇您的 Aurora 叢集和 TESTDB 資料庫。
4. 選擇您的密碼，然後選擇「輸入」。
5. 一旦發現資料表，請更新類型名稱。變Todos更Tasks為Todo和Task。
6. 選擇預覽結構描述來預覽產生的結構描述。你的模式看起來像這樣：

```
type Todo {
  id: Int!
  description: String!
  due: AWSDate!
  createdAt: String
}

type Task {
  id: Int!
  todoId: Int!
  description: String
}
```

7. 對於角色，您可以AWS AppSync建立新角色，也可以使用類似下列原則來建立角色：

```
{
    "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:*"
            ]
        }
    ]
}
```

請注意，此原則中有兩個陳述式可讓您授與角色存取權。第一個資源是您的 Aurora 叢集，第二個資源是您的 AWS Secrets Manager ARN。

選擇「下一步」，檢閱組態詳細資料，然後選擇「建立 API」。您現在有一個完全可操作的 API。您可以在「結構描述」頁面上查看 API 的完整詳細信息。

適用於 RDS 的解析器

API 創建流程會自動創建解析器以與我們的類型進行交互。如果您查看「結構描述」頁面，您會發現解析器必須執行以下操作：

- todo通過字Mutation.createTodo段創建一個。
- todo通過字Mutation.updateTodo段更新一個。
- 刪除todo通過字Mutation.deleteTodo段。
- todo通過Query.getTodo字段獲取單個。
- 列出所有todos通過Query.listTodos字段。

您會發現類型附加的Task類似欄位和解析器。讓我們仔細看看一些解析器。

突變. 創建待辦事項

從主AWS AppSync控台右側的結構描述編輯器中，選擇「testdb下一步」createTodo(...): Todo。解析器代碼使用rds模塊中的insert函數來動態創建將數據添加到表中的插入語句。todos因為我們正在使用 Postgres，所以我們可以利用returning語句來獲取插入的數據。

讓我們更新解析器以正確指定DATE字段的類型：due

```
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]
}
```

儲存解析器。類型提示將輸入對像中的due正確標記為DATE類型。這可讓 Postgres 引擎正確解譯值。接下來，更新您的結構描述以id從CreateTodo輸入中移除。由於 Postgres 數據庫可以返回生成的 ID，因此我們可以依靠它來創建並將結果作為單個請求返回：

```
input CreateTodoInput {
    due: AWSDate!
    createdAt: String
    description: String!
}
```

進行變更並更新您的結構描述。前往查詢編輯器將項目添加到數據庫中：

```
mutation CreateTodo {
  createTodo(input: {description: "Hello World!", due: "2023-12-31"}) {
    id
    due
    description
    createdAt
  }
}
```

你得到的結果：

```
{
  "data": {
    "createTodo": {
      "id": 1,
      "due": "2023-12-31",
      "description": "Hello World!",
      "createdAt": "2023-11-14 20:47:11.875428"
    }
  }
}
```

查詢. 列表待辦事項

從主控台右側的結構描述編輯器中，選擇「testdb下一步」listTodos(id: ID!): Todo。要求處理常式會使用 select 公用程式函數，在執行階段動態建立要求。

```
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)
}
```

我們想要todos根據due日期進行篩選。讓我們更新解析器以將due值轉換為。DATE更新匯入清單和要求處理常式：

```
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 };
}
```

讓我們試試查詢。在「查詢」編輯器中：

```
query LIST {
  listTodos(limit: 10, filter: {due: {between: ["2021-01-01", "2025-01-02"]}}) {
    items {
      id
      due
      description
    }
  }
}
```

突變. 更新待辦事項

你也可以update一個Todo. 從「查詢」編輯器中，讓我們更新的第一個Todo項目id1。

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits"}) {
    description
    due
    id
```

```
  }
}
```

請注意，您必須指id定要更新的項目。您也可以指定條件，以僅更新符合特定條件的料號。例如，我們可能只想編輯項目，如果描述開頭為edits：

```
mutation UPDATE {
  updateTodo(input: {id: 1, description: "edits: make a change"}, condition:
 {description: {beginsWith: "edits"}}) {
    description
    due
    id
  }
}
```

就像我們處理create和list操作的方式一樣，我們可以更新解析器以將due字段轉換為. DATE 將這些變更儲存到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];
}
```

現在嘗試使用條件更新：

```
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
        }
}
```

突變. 刪除待辦事項

你可delete以Todo與deleteTodo突變。這就像updateTodo突變一樣，您必須指定要刪除id的項目：

```
mutation DELETE {
  deleteTodo(input: {id: 1}) {
    description
    due
    id
  }
```

```
}
```

撰寫自訂查詢

我們已經使用rds模塊實用程序來創建我們的 SQL 語句。我們也可以編寫自己的自定義靜態語句與我們的數據庫進行交互。首先，更新結構描述以從CreateTask輸入中移除id欄位。

```
input CreateTaskInput {
    todoId: Int!
    description: String
}
```

接下來，創建幾個任務。工作與下列項目具有外部索引鍵關係Todo：

```
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類型中創建一個名為的新字段getTodoAndTasks：

```
getTodoAndTasks(id: Int!): Todo
```

添加tasks字段的Todo類型：

```
type Todo {
    due: AWSDate!
    id: Int!
    createdAt: String
    description: String!
    tasks:TaskConnection
}
```

儲存結構描述。在主控台的資料架構編輯器中，選擇右側的「貼附解析器」。getTodosAndTasks(id: Int!): Todo選擇您的 Amazon RDS 資料來源。使用以下代碼更新您的解析器：

```
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;
}
```

在這段程式碼中，我們使用sql標籤範本來撰寫 SQL 陳述式，我們可以在執行階段安全地傳遞動態值。 createPgStatement一次最多可以使用兩個 SQL 請求。我們使用它為我們發送一個查詢，另一個查詢為我們todo的tasks. 您可以通過JOIN語句或任何其他方法來完成此操作。這個想法是能夠編寫自己的 SQL 語句來實現您的業務邏輯。要在查詢編輯器中使用查詢，我們可以嘗試以下操作：

```
query TodoAndTasks {
  getTodosAndTasks(id: 2) {
    id
    due
    description
    tasks {
      items {
        id
        description
      }
    }
  }
}
```

刪除叢集

> **Important**
>
> 刪除叢集是永久性的。在執行此操作之前，請徹底檢查您的項目。

若要刪除叢集：

```
$ aws rds delete-db-cluster \
    --db-cluster-identifier appsync-tutorial \
    --skip-final-snapshot
```

解析器教程（VTL）

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

數據源和解析器是AWS AppSync 轉換 GraphQL 請求和從資源中獲取信息的方式。AWSAWS AppSync 支援自動佈建和與特定資料來源類型的連線。AWS AppSync 支援AWS Lambda、Amazon DynamoDB、關聯式資料庫 (Amazon Aurora 無伺服器)、亞馬遜 OpenSearch 服務和 HTTP 端點做為資料來源。您可以將 GraphQL API 與現有AWS資源搭配使用，也可以建置資料來源和解析器。本區段將帶您在一系列教學課程中完成此過程，以更好地了解詳細資訊如何運作和調校選項。

AWS AppSync 使用用 Apache 速度模板語言（VTL）編寫的映射模板進行解析器。如需有關使用對應範本的詳細資訊，請參閱解析器對映範本參考。有關使用 VTL 的更多資訊，請參閱解析器映射範本程式設計指南。

AWS AppSync 支援從 GraphQL 結構描述自動佈建 DynamoDB 表格，如從結構描述佈建 (選用) 和啟動範例結構描述中所述。您也可以從現有將建立結構描述和連接解析程式的 DynamoDB 資料表滙入。這會在從亞馬遜動態資料庫匯入 (選用) 中概述。

主題

- 教學課程：解析器
- 教學課程：Lambda 解析器
- 教程：Amazon OpenSearch 服務解析器
- 教學課程：本機解析程式
- 教學課程：合併 GraphQL 解析程式
- 教學課程：Batch 解析器
- 教學課程：交易解析器
- 教學課程：HTTP 解析器
- 教學課程：Aurora 無伺服器
- 教學課程：管線解析器
- 教學課程：增量同步

教學課程：解析器

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

本教學課程說明如何將自己的 Amazon DynamoDB 表格帶入AWS AppSync 並將其連接到 GraphQL API。

您可以讓您代表AWS AppSync 佈建 DynamoDB 資源。或者，如果您願意，您可以透過建立資料來源和解析程式來將現有的資料表連接到 GraphQL 結構描述。在這兩種情況下，您將能夠透過 GraphQL 陳述式讀取和寫入 DynamoDB 資料庫，以及訂閱即時資料。

有您必須完成的特定的組態步驟，才能讓 GraphQL 陳述式轉譯到 DynamoDB 操作，以及讓回應轉譯回 GraphQL。本教學課程概述透過幾個真實世界案例和資料存取模式的組態程序。

設定您的 DynamoDB 資料表

要開始本教程，首先您需要按照以下步驟佈建AWS資源。

1. 在 CLI 中使用下列AWS CloudFormation範本佈建AWS資源：

   ```
   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，您也可以在AWS帳戶中的美國西部 2 (奧勒岡) 區域啟動下列AWS CloudFormation堆疊。

   這將會建立下列項目：

   - 稱為將保Post存資料AppSyncTutorial-Post的 DynamoDB 資料表。
   - IAM 角色和關聯的 IAM 受管政策，可允許 AWS AppSync 與 Post 資料表互動。
2. 要查看有關堆疊和建立的資源的詳細資訊，請執行以下 CLI 命令：

```
aws cloudformation describe-stacks --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

3. 之後若要刪除資源，您可以執行以下項目：

```
aws cloudformation delete-stack --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

建立您的 GraphQL API

若要在 AWS AppSync 中建立 GraphQL API：

1. 登入 AWS Management Console 並開啟 AppSync 主控台。
   - 在 API 儀表板中，選擇「建立 API」。
2. 在「自訂您的 API 或從 Amazon DynamoDB 匯入」視窗下，選擇「從頭開始建立」。
   - 選擇開始在同一窗口的右側。
3. 在「API 名稱」欄位中，將 API 的名稱設定為AWSAppSyncTutorial。
4. 選擇 建立 。

AWS AppSync 主控台會使用 API 金鑰身分驗證模式為您建立新 GraphQL API。您可以使用主控台來設定其他 GraphQL API 和對其執行查詢，以進行此教學的其他部分。

定義一個基本的後期 API

現在您已經建立 AWS AppSync GraphQL API，您可以設定一個基本結構描述，以便基本建立、擷取和刪除貼文資料。

1. 登入 AWS Management Console 並開啟 AppSync 主控台。
   - 在 API 儀表板中，選擇您剛建立的 API。
2. 在側邊欄中，選擇結構描述。
   - 在「結構描述」窗格中，以下列程式碼取代內容：

```
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. 選擇 儲存 。

這個結構描述會定義 `Post` 類型和操作以新增和取得 `Post` 物件。

設定 DynamoDB 表格的資料來源

接下來，將結構描述中定義的查詢和突變連結至 `AppSyncTutorial-Post` DynamoDB 表格。

首先，AWS AppSync 需要注意到您的資料表。您需要在 AWS AppSync 設定資料來源來這樣做：

1. 登入 AWS Management Console 並開啟 AppSync 主控台。
   a. 在 API 儀表板中，選擇您的 GraphQL API。
   b. 在側邊欄中，選擇「資料來源」。
2. 選擇 Create data source (建立資料來源)。

a. 對於資料來源名稱，輸入 in PostDynamoDBTable。

b. 對於資料來源類型，請選擇 Amazon DynamoDB 資料表。

c. 在「區域」中，選擇「US-WEST-2」。

d. 對於表格名稱，請選擇 AppSyncTutorial-發佈 DynamoDB 表。

e. 建立新的 IAM 角色 (建議使用) 或選擇具有 lambda:invokeFunction IAM 權限的現有角色。現有角色需要信任原則，如附加資料來源一節所述。

以下是具有對資源執行操作所需許可的 IAM 政策範例：

```
{
     "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. 選擇 建立 。

設定新 addPost 解析器 (DynamoDB 器) PutItem

瞭解 DynamoDB 表後 AWS AppSync ，您可以透過定義解析器將其連結至個別查詢和突變。您建立的第一個解析器是addPost解析器，它可讓您在 DynamoDB 表格中建立貼文。AppSyncTutorial-Post

解析程式具有下列元件：

- 在 GraphQL 結構描述中要附加解析程式的位置。在這種情況下，您會對 addPost 類型上的 Mutation 欄位設定解析程式。發起人呼叫 mutation { addPost(...){...} } 時會叫用此解析程式。
- 用於此解析程式的資料來源。在這種情況下，您想要使用先前定義的 PostDynamoDBTable 資料來源，如此您可以將項目新增到 AppSyncTutorial-Post DynamoDB 資料表。

- 請求映射範本。要求映射範本的目的是從發起人取得傳入要求並將其轉譯至適合 AWS AppSync 的指示以針對 DynamoDB 執行。
- 回應映射範本。回應映射範本的任務即是從 DynamoDB 取得回應，並將其轉譯為 GraphQL 期望的項目。如果 DynamoDB 中的資料形狀與 GraphQL 中的 `Post` 類型不同，此功能會很有用，但如果他們的形狀一樣，您只需傳遞資料。

設定解析程式：

1. 登入 AWS Management Console 並開啟 AppSync 主控台。
   a. 在 API 儀表板中，選擇您的 GraphQL API。
   b. 在側邊欄中，選擇「資料來源」。
2. 選擇 Create data source (建立資料來源)。
   a. 對於資料來源名稱，輸入 in `PostDynamoDBTable`。
   b. 對於資料來源類型，請選擇 Amazon DynamoDB 資料表。
   c. 在「區域」中，選擇「US-WEST-2」。
   d. 對於表格名稱，請選擇 AppSyncTutorial-發佈 DynamoDB 表。
   e. 建立新的 IAM 角色 (建議使用) 或選擇具有 `lambda:invokeFunction` IAM 權限的現有角色。現有角色需要信任原則，如附加資料來源一節所述。

      以下是具有對資源執行操作所需許可的 IAM 政策範例：

```
{
     "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. 選擇 建立 。

4. 選擇 Schema (結構描述) 標籤。
5. 在右側的 [資料類型] 窗格中，尋找 [變異] 類型上的 [addPost] 欄位，然後選擇 [附加]。
6. 在「動作」功能表中，選擇「更新執行階段」，然後選擇「單位解析器（僅限 VTL）」。
7. 在 [資料來源名稱] 中，選擇 [資料PostDynamo表]。
8. 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
    "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：所有金鑰和屬性值皆有指定的類型。例如，您將 `author` 欄位設為 `{ "S" : "${context.arguments.author}" }`。該S部分向AWS AppSync 和 DynamoDB 表示該值將是一個字符串值。會將實際值填入 `author` 引數。同樣地，`version` 欄位是號碼欄位，因為它會將 `N` 用於類型。最後，您也可以初始化 `ups`、`downs` 和 `version` 欄位。

在本教學課程中，您已指定 GraphQL `ID!` 類型 (其為插入 DynamoDB 的新項目編製索引) 作為用戶端引數的一部分。 AWS AppSync帶有一個用於自動 ID 生成的實用程序，您也可以以的形式使用`$utils.autoId()`它`"id" : { "S" : "${$utils.autoId()}" }`。然後，您可以直接將 `id: ID!` 移出 `addPost()` 的結構描述定義，該 ID 便會自動插入。您不會在本教學課程中使用此技巧，但是在寫入 DynamoDB 表格時，應將其視為良好作法。

如需映射範本的詳細資訊，請參閱解析程式映射範本概觀參考文件。如需有關 GetItem 請求對應的詳細資訊，請參閱GetItem參考文件。如需有關類型的詳細資訊，請參閱類型系統 (要求映射) 參考文件。

9. 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

注意：由於 AppSyncTutorial-Post 資料表中的資料形狀正好與 GraphQL 中的 Post 類型形狀相符，回應映射範本只需直接傳遞結果。另請注意，此教學課程中的所有範例會使用相同的回應映射範本，因此您只需建立一個檔案。

10. 選擇 儲存 。

呼叫 API 以新增文章

現在已設定解析器，AWS AppSync 可以將傳入的addPost突變轉換為 DynamoDB 作業。 PutItem 您現在可以執行變動以將項目放置到資料表中。

- 選擇 Queries (查詢) 標籤。
- 將以下變動貼到 Queries (查詢) 窗格：

```
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
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 新建立文章的結果應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "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
    }
  }
}
```

以下是發生的情況：

- AWS AppSync 收到addPost突變請求。
- AWS AppSync 取得請求和請求對應範本，並產生了請求對應文件。此看起來如下：

```
{
    "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 使用請求對應文件來產生並執行 DynamoDB `PutItem` 請求。
- AWS AppSync 取得PutItem要求的結果，並將它們轉換回 GraphQL 類型。

```
{
    "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
}
```

- 透過回應映射文件來進行傳遞，這可讓其以原狀進行傳遞。
- 傳回在 GraphQL 回應中新建立的物件。

設定 getPost 解析器 (DynamoDB 器) GetItem

現在您可以將資料新增至 AppSyncTutorial-Post DynamoDB 資料表，您必須設定getPost查詢，才能從資料表擷取該資料。AppSyncTutorial-Post若要執行此作業，您可以設定另一個解析程式。

- 選擇 Schema (結構描述) 標籤。
- 在右側的 [資料類型] 窗格中，尋找 [查詢] 類型上的 getPost 欄位，然後選擇 [附加]。
- 在「動作」功能表中，選擇「更新執行階段」，然後選擇「單位解析器（僅限 VTL）」。
- 在 [資料來源名稱] 中，選擇 [資料PostDynamo表]。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

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

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

- 選擇 儲存 。

呼叫 API 以取得文章

現在解析器已經設置好了，AWS AppSync 知道如何將傳入的`getPost`查詢轉換為 DynamoDB `GetItem` 操作。您現在可以執行查詢，擷取您稍早建立的貼文。

- 選擇 Queries (查詢) 標籤。
- 在 Queries (查詢) 窗格中，貼入下面內容：

```
query getPost {
  getPost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 從 DynamoDB 擷取的貼文應會顯示在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "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
    }
  }
}
```

以下是發生的情況：

- AWS AppSync 收到getPost查詢請求。
- AWS AppSync 取得請求和請求對應範本，並產生了請求對應文件。此看起來如下：

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "123" }
    }
}
```

- AWS AppSync 使用請求對應文件來產生並執行 DynamoDB GetItem 請求。
- AWS AppSync 獲取GetItem請求的結果並將其轉換回 GraphQL 類型。

```
{
    "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
}
```

- 透過回應映射文件來進行傳遞，這可讓其以原狀進行傳遞。
- 在回應中傳回擷取的物件。

或者，採取以下示例：

```
query getPost {
  getPost(id:123) {
    id
    author
    title
  }
}
```

如果您的getPost查詢只需要id、和 authortitle，您可以將請求對應範本變更為使用投影運算式，僅從 DynamoDB 表指定您想要的屬性，以避免不必要的資料從 DynamoDB 傳輸到。AWS AppSync例如，請求映射模板可能如下所示：

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    },
    "projection" : {
     "expression" : "#author, id, title",
     "expressionNames" : { "#author" : "author"}
    }
}
```

創建 updatePost 突變（DynamoDB UpdateItem

到目前為止，您可以在 DynamoDB 中建立和擷取Post物件。接下來，您會設定新的變動，讓我們能夠更新物件。您可以使用 UpdateItem DynamoDB 作業來執行這項操作。

- 選擇 Schema (結構描述) 標籤。
- 修改 Schema (結構描述) 窗格中的 `Mutation` 類型來新增 `updatePost` 變動，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!
}
```

- 選擇 儲存 。

- 在右側的 [資料類型] 窗格中，在 [變更] 類型上尋找新建立的 updatePost 欄位，然後選擇 [附加]。
- 在「動作」功能表中，選擇「更新執行階段」，然後選擇「單位解析器（僅限 VTL）」。
- 在 [資料來源名稱] 中，選擇 [資料PostDynamo表]。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
    "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 }
        }
    }
}
```

附註：此解析程式使用的是 DynamoDB UpdateItem，這與作業有明顯的不同。 PutItem 您不需要撰寫整個項目，而是要求 DynamoDB 更新某些屬性。這是使用 DynamoDB 更新運算式來完成的。表達式本身會在 `expression` 區段中的 `update` 欄位中指定。其要求設定 `author`、`title`、`content` 以及 url 屬性，然後遞增 `version` 欄位。要使用的值不會出現在表達式本身；表達式的預留位置名稱開頭為冒號，其會在 `expressionValues` 欄位中加以定義。最後，DynamoDB 的保留字無法出現在. `expression` 例如，由於 `url` 是保留字，因此若要更新 `url` 欄位，您可以使用名稱預留位置，並在 `expressionNames` 欄位中定義他們。

如需有關`UpdateItem`請求對應的詳細資訊，請參閱UpdateItem參考文件。如需如何撰寫更新運算式的詳細資訊，請參閱 DynamoDB UpdateExpressions 文件。

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

呼叫 API 以更新文章

現在解析器已經設置好了，AWS AppSync 知道如何將傳入的update突變轉換為 DynamoDB 操作。Update您現在可以執行變動，以更新您之前撰寫的項目。

- 選擇 Queries (查詢) 標籤。
- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 id 引數更新為您先前記下的值。

```
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
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- DynamoDB 中更新的貼文應顯示在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "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
    }
  }
}
```

在此範例中，並未修改ups和downs欄位，因為請求對應範本AWS AppSync 並未要求 DynamoDB 對這些欄位執行任何動作。此外，由於您要求 DynamoDB 向version欄位新增 1 AWS AppSync ，因此欄位會增加 1。version

修改 updatePost 解析器 (DynamoDB) UpdateItem

這對於 updatePost 變動而言是一個好的開始，但有兩個主要的問題：

- 如果您只想更新單一欄位，您必須更新所有欄位。
- 如果兩個人同時修改物件，您便可能會失去資訊。

為了解決這些問題，您將會修改 updatePost 變動，使其僅修改請求中指定的引數，然後將條件新增到 UpdateItem 操作。

1. 選擇 Schema (結構描述) 標籤。
2. 在 Schema (結構描述) 窗格中，修改 Mutation 類型中的 updatePost 欄位，以便從 author、title、content 及 url 引數中移除驚嘆號，並請確保 id 欄位維持原狀。這會使它們變成可選引數。此外，新增必要 expectedVersion 引數。

```
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. 選擇 儲存 。
4. 在右側的 [資料類型] 窗格中，尋找 [變異] 類型上的 [updatePost] 欄位。
5. 選擇「PostDynamo資料庫表格」以開啟現有的解析程式。
6. 在 Configure the request mapping template (設定要求映射範本) 中修改要求映射範本，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. 選擇 儲存 。

此範本是更為複雜的範例。它示範映射範本的強大功能和靈活性。它會循環使用所有引數，但會略過 `id` 和 `expectedVersion`。如果引數設定為某個項目，它會要求AWS AppSync 和 DynamoDB 更新 DynamoDB 中物件上的該屬性。如果屬性設定為 null，它會要求AWS AppSync 和 DynamoDB 從貼文物件中移除該屬性。若未指定引數，它將會保留該屬性。它會增量 `version` 欄位。

另外，還有新的 `condition` 區段。條件運算式可讓您根據 DynamoDB 中已有的物件狀態告訴AWS AppSync 和 DynamoDB 是否應該成功執行作業。在這種情況下，只有當 DynamoDB 中目前項目的`version`欄位與引數完全相符時，您才希望`UpdateItem`請求成功。`expectedVersion`

如需條件表達式的詳細資訊，請參閱條件表達式參考文件。

呼叫 API 以更新文章

讓我們嘗試使用新解析程式來更新 `Post` 物件：

- 選擇 Queries (查詢) 標籤。
- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 `id` 引數更新為您先前記下的值。

```
mutation updatePost {
  updatePost(
```

```
    id:123
    title: "An empty story"
    content: null
    expectedVersion: 2
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- DynamoDB 中更新的貼文應顯示在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "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
    }
  }
}
```

在此請求中，您要求AWS AppSync 和 DynamoDB 僅更新`title`和`content`欄位。它會將所有其他欄位單獨保留 (除了遞增 `version` 欄位)。您將 `title` 屬性設定為新的值，並從文章中移除 `content` 屬性。`author`、`url`、`ups` 和 `downs` 欄位維持原狀。

嘗試再次執行變動要求，讓要求維持原狀。您應該會看到類似以下的回應：

```
{
  "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)"
    }
  ]
}
```

條件表達式評估為「false」，因此要求失敗：

- 您第一次執行要求時，DynamoDB 中貼文`version`欄位的值為2與引數相`expectedVersion`符。要求成功，這表示 DynamoDB 中的`version`欄位已遞增至。3
- 第二次執行要求時，DynamoDB 中貼文`version`欄位的值為3，此值與引`expectedVersion`數不符。

此模式通常稱為樂觀鎖定。

AWS AppSync DynamoDB 解析器的一項功能是它會傳回 DynamoDB 中貼文物件的目前值。您可以在 GraphQL 回應的 `data` 區段 `errors` 欄位中找到此值。您的應用程式可以使用這些資訊來決定繼

續執行的方式。在這種情況下，您可以看到 DynamoDB 中物件的`version`欄位設定為`3`，因此您只需將`expectedVersion`引數更新為，要求就會再次成功。`3`

如需有關處理條件檢查錯誤的詳細資訊，請參閱條件表達式映射範本參考文件。

建立 upvotePost 和 downvotePost 突變 (DynamoDB) UpdateItem

`Post` 類型具有 `ups` 和 `downs` 欄位，可記錄贊同數和不贊同數，但是到目前為止，API 不會讓我們使用它們來執行任何動作。讓我們新增一些變動來對文章表示贊同與不贊同。

- 選擇 Schema (結構描述) 標籤。
- 修改 Schema (結構描述) 窗格中的 `Mutation` 類型以新增 `upvotePost` 和 `downvotePost` 變動，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!
}
```

- 選擇 儲存 。
- 在右側的 [資料類型] 窗格中，尋找 [突變] 類型上新建立的 UpvotePost 欄位，然後選擇 [附加]。
- 在「動作」功能表中，選擇「更新執行階段」，然後選擇「單位解析器（僅限 VTL）」。
- 在 [資料來源名稱] 中，選擇 [資料PostDynamo表]。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
```

```
    "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 }
        }
    }
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

- 選擇 儲存 。
- 在右側的 [資料類型] 窗格中，在 [變異] 類型上尋找新建立的downvotePost欄位，然後選擇 [附加]。
- 在 [資料來源名稱] 中，選擇 [資料PostDynamo表]。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
    "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 }
        }
    }
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

- 選擇 儲存 。

呼叫 API 來對文章表達贊同或不贊同

現在，新的解析器已經設置好了，AWS AppSync 知道如何將傳入upvotePost或downvote突變轉換為 DynamoDB 操作。 UpdateItem 您現在可以執行變動，對您之前建立的文章表達贊同或不贊同。

- 選擇 Queries (查詢) 標籤。
- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 id 引數更新為您先前記下的值。

```
mutation votePost {
  upvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 該貼文會在 DynamoDB 中更新，並應顯示在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "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
    }
  }
}
```

- 再選擇幾次 Execute query (執行查詢)。您應查看在每次執行查詢時以 1 為單位而遞增的 ups 和 version 欄位。

- 變更查詢以呼叫 downvotePost 變動，如下所示：

```
mutation votePost {
  downvotePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。此時，您應查看在每次執行查詢時以 1 為單位而遞增的 downs 和 version 欄位。

```
{
  "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
    }
  }
}
```

設定 deletePost 解析器 (DynamoDB) DeleteItem

您要設定的下一個變動是刪除貼文。您可以使用 DeleteItem DynamoDB 作業來執行這項操作。

- 選擇 Schema (結構描述) 標籤。
- 修改 Schema (結構描述) 窗格中的 Mutation 類型來新增 deletePost 變動，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!
}
```

這次您將 expectedVersion 欄位設為選用，稍後在您新增要求映射範本時，將對此進行說明。

- 選擇 儲存 。
- 在右側的 [資料類型] 窗格中，在 [變更] 類型上尋找新建立的刪除欄位，然後選擇 [附加]。
- 在「動作」功能表中，選擇「更新執行階段」，然後選擇「單位解析器（僅限 VTL）」。
- 在 [資料來源名稱] 中，選擇 [資料PostDynamo表]。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
    "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 引數為可選引數。如果呼叫者在請求中設定expectedVersion引數，則範本會新增條件，該條件僅在項目已刪除或 DynamoDB 中貼文的version屬性完全符合時，才允許DeleteItem要求成功。expectedVersion如果省略，將不會在 DeleteItem 要求中指定條件表達式。無論 DynamoDB 中的值為何version，或項目是否存在，它都會成功。

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

注意：即使您正在刪除某個項目，如果該項目尚未刪除，您仍然可以傳回已刪除的項目。

- 選擇 儲存 。

如需有關DeleteItem請求對應的詳細資訊，請參閱DeleteItem參考文件。

呼叫 API 以刪除文章

現在解析器已經設置好了，AWS AppSync 知道如何將傳入的delete突變轉換為 DynamoDB 操作。DeleteItem您現在可以執行變動以在資料表中刪除項目。

- 選擇 Queries (查詢) 標籤。
- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 id 引數更新為您先前記下的值。

```
mutation deletePost {
  deletePost(id:123) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。

- 此貼文即會從 DynamoDB 中刪除。請注意，AWS AppSync 會從 DynamoDB 傳回已刪除項目的值，其應會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "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
    }
  }
}
```

如果對 `deletePost` 的呼叫實際上是從 DynamoDB 遭到刪除的呼叫，則只會傳回值。

- 再次選擇 Execute query (執行查詢)。
- 此呼叫仍會成功，但不會傳回任何值。

```
{
  "data": {
    "deletePost": null
  }
}
```

現在讓我們嘗試刪除文章，但這次是指定 `expectedValue`。首先，您需要建立新文章，因為您刪除的正好是您到目前為止所用的文章。

- 將以下變動貼到 Queries (查詢) 窗格：

```
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
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 新建立文章的結果應該會出現在查詢窗格右側的結果窗格中。請記下新建立的物件的 `id`，因為您很快就會用到它。其看起來與下列類似：

```
{
  "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
    }
  }
}
```

現在讓我們嘗試刪除該文章，但我們將錯誤的值放入 `expectedVersion`：

- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 `id` 引數更新為您先前記下的值。

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 9999
```

```
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。

```
{
  "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)"
    }
```

```
  ]
}
```

請求失敗，因為條件運算式評估為 false：DynamoDB 中貼文`version`的值與引數中`expectedValue`指定的值不符。會在 GraphQL 回應 `data` 區段中 `errors` 欄位傳回物件的目前值。

- 重試要求，但更正 `expectedVersion`：

```
mutation deletePost {
  deletePost(
    id:123
    expectedVersion: 1
  ) {
    id
    author
    title
    content
    url
    ups
    downs
    version
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 這次請求成功，並傳回從 DynamoDB 刪除的值：

```
{
  "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
    }
  }
}
```

- 再次選擇 Execute query (執行查詢)。
- 呼叫仍然成功，但這次不會傳回任何值，因為已在 DynamoDB 中刪除貼文。

```
{
  "data": {
    "deletePost": null
  }
}
```

設定 allPost 解析程式 (DynamoDB Scan)

到目前為止，如果您知道要尋找之每個文章的 `id`，API 才有用。讓我們新增新的解析程式，傳回資料表中的所有文章。

- 選擇 Schema (結構描述) 標籤。
- 修改 Schema (結構描述) 窗格中的 `Query` 類型以新增 `allPost` 查詢，如下所示：

```
type Query {
    allPost(count: Int, nextToken: String): PaginatedPosts!
    getPost(id: ID): Post
}
```

- 新增 `PaginationPosts` 類型：

```
type PaginatedPosts {
    posts: [Post!]!
    nextToken: String
}
```

- 選擇 儲存 。
- 在右側的 [資料類型] 窗格中，在 [查詢] 類型上尋找新建立的 allPost 欄位，然後選擇 [附加]。
- 在「動作」功能表中，選擇「更新執行階段」，然後選擇「單位解析器（僅限 VTL）」。
- 在 [資料來源名稱] 中，選擇 [資料PostDynamo表]。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
    "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
}
```

此解析程式有兩個選用引數：count 會指定單一呼叫中傳回的項目上限數，nextToken 可用來擷取下一組結果 (稍後您將顯示 nextToken 值來自何處)。

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

注意：到目前為止，此回應映射範本與所有其他範本皆不同。allPost 查詢結果是 PaginatedPosts，其中包含文章清單和分頁字符。此物件形狀與從 AWS AppSync DynamoDB 解析程式傳回的物件不同：文章清單在 AWS AppSync DynamoDB 解析程式結果中稱為 items，但在 PaginatedPosts 中稱為 posts。

- 選擇 儲存 。

如需有關 Scan 要求映射的詳細資訊，請參閱掃描參考文件。

呼叫 API 以掃描所有文章

現在解析器已經設置好了，AWS AppSync 知道如何將傳入的allPost查詢轉換為 DynamoDB Scan 操作。您現在可以掃描資料表來擷取所有文章。

在您可以嘗試之前，您必須將一些資料填入資料表，因為您已刪除目前為止所使用的項目。

- 選擇 Queries (查詢) 標籤。
- 將以下變動貼到 Queries (查詢) 窗格：

```
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 }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。

現在，讓我們來掃描資料表，一次會傳回五個結果。

- 將以下查詢貼到 Queries (查詢) 窗格中：

```
query allPost {
  allPost(count: 5) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 前五篇文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "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
    }
  }
}
```

您取得五個結果以及 nextToken，您可以用它來取得下一組結果。

- 更新 allPost 查詢以包括來自之前結果組的 nextToken：

```
query allPost {
  allPost(
    count: 5
    nextToken:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
  ) {
    posts {
      id
      author
    }
```

```
    nextToken
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 剩餘的四篇文章應該會出現在查詢窗格右側的結果窗格中。在這組結果中沒有 `nextToken`，因為您已翻遍了所有九篇文章，已經沒有剩餘的文章。其看起來與下列類似：

```
{
  "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
    }
  }
}
```

設定 allPostsBy作者解析程式 (DynamoDB 查詢)

除了針對所有貼文掃描 DynamoDB 之外，您還可以查詢 DynamoDB 以擷取特定作者建立的貼文。您先前建立的 DynamoDB 表格已經具有`author-index`可與 DynamoDB 作業搭配使用的`GlobalSecondaryIndex`呼叫，以擷取特定`Query`作者建立的所有貼文。

- 選擇 Schema (結構描述) 標籤。

- 修改 Schema (結構描述) 窗格中的 `Query` 類型以新增 `allPostsByAuthor` 查詢，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：這會使用與您在 `allPost` 查詢中使用的相同 `PaginatedPosts` 類型。

- 選擇 儲存 。
- 在右側的 [資料類型] 窗格中，在 [查詢] 類型上尋找新建立的 [allPostsBy作者] 欄位，然後選擇 [附加]。
- 在「動作」功能表中，選擇「更新執行階段」，然後選擇「單位解析器（僅限 VTL）」。
- 在 [資料來源名稱] 中，選擇 [資料PostDynamo表]。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
    "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
}
```

  如同 `allPost` 解析程式，此解析程式有兩個可選引數：`count`，其會指定項目的上限數，以在單一呼叫中傳回，以及 `nextToken`，其可用來擷取接下來的結果組 (您可從先前的呼叫中取得 `nextToken` 值)。

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

注意：這是與您在 allPost 解析程式中所使用的相同回應映射範本。

- 選擇 儲存 。

如需有關 Query 要求映射的詳細資訊，請參閱查詢參考文件。

呼叫 API 來查詢某個作者的所有文章

現在解析器已經設置好了，AWS AppSync 知道如何將傳入的allPostsByAuthor突變轉換為針對索引的 DynamoDB Query 操作。author-index您現在可以查詢資料表以擷取特定作者的所有文章。

在這麼做之前，讓我們再將一些文章填入資料表，因為目前為止每篇文章的作者都一樣。

- 選擇 Queries (查詢) 標籤。
- 將以下變動貼到 Queries (查詢) 窗格：

```
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 }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。

現在，讓我們查詢資料表，傳回 Nadia 撰寫的所有文章。

- 將以下查詢貼到 Queries (查詢) 窗格中：

```
query allPostsByAuthor {
```

```
  allPostsByAuthor(author: "Nadia") {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- Nadia 撰寫的所有文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you know...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

分頁適用於 Query，如同它適用於 Scan。例如，讓我們查詢 AUTHORNAME 的所有文章，一次取得五篇。

- 將以下查詢貼到 Queries (查詢) 窗格中：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
  ) {
    posts {
```

```
      id
      title
    }
    nextToken
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- AUTHORNAME 撰寫的所有文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "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
    }
  }
}
```

- 使用之前查詢傳回的值更新 nextToken 引數，如下所示：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "AUTHORNAME"
    count: 5
    nextToken:
 "eyJ2ZXJzaW9uIjoxLCJ0b2tlbiI6IkFRSUNBSGo4eHR0RG0xWXhUa1F0cEhXMEp1R3B0M1B3eThOSmRvcG9ad2RHYjI
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- AUTHORNAME 撰寫的其餘文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "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

到目前為止，Post 類型為統一金鑰/值物件。您還可以使用AWS AppSyncDynamo數據庫解析器建模複雜的對象，例如集合，列表和映射。

讓我們將 Post 類型更新為包含標籤。貼文可以有 0 或多個標籤，這些標籤會以字串集形式儲存在 DynamoDB 中。您也會設定一些變動來新增和移除標籤，以及新查詢以掃描含特定標籤的文章。

- 選擇 Schema (結構描述) 標籤。
- 修改 Schema (結構描述) 窗格中的 Post 類型以新增 tags 欄位，如下所示：

```
type Post {
  id: ID!
  author: String
  title: String
  content: String
  url: String
  ups: Int!
  downs: Int!
  version: Int!
  tags: [String!]
}
```

- 修改 Schema (結構描述) 窗格中的 Query 類型以新增 allPostsByTag 查詢，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
}
```

- 修改 Schema (結構描述) 窗格中的 Mutation 類型以新增 addTag 和 removeTag 變動，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!
}
```

- 選擇 儲存 。
- 在右側的 [資料類型] 窗格中，在 [查詢] 類型上尋找新建立的 [allPostsBy標籤] 欄位，然後選擇 [附加]。
- 在 [資料來源名稱] 中，選擇 [資料PostDynamo表]。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
    "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
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
{
    "posts": $utils.toJson($context.result.items)
    #if( ${context.result.nextToken} )
        ,"nextToken": $util.toJson($context.result.nextToken)
    #end
}
```

- 選擇 儲存 。
- 在右側的 [資料類型] 窗格中，在 [變更] 類型上尋找新建立的 addTag 欄位，然後選擇 [附加]。
- 在 [資料來源名稱] 中，選擇 [資料PostDynamo表]。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
    "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 }
        }
    }
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

- 選擇 儲存 。
- 在右側的 [資料類型] 窗格中，在 [變更] 類型上尋找新建立的 removeTag 欄位，然後選擇 [連接]。
- 在 [資料來源名稱] 中，選擇 [資料PostDynamo表]。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
    "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 }
        }
    }
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

- 選擇 儲存 。

呼叫 API 來使用標籤

現在，您已經設定解析器，AWS AppSync 知道如何將傳入`addTag`和`allPostsByTag`請求轉換為 DynamoDB `UpdateItem` 和作業。`removeTag` `Scan`

為了嘗試看看，讓我們選擇您先前建立的其中一篇文章。例如，讓我們使用 `Nadia` 撰寫的一篇文章。

- 選擇 Queries (查詢) 標籤。
- 將以下查詢貼到 Queries (查詢) 窗格中：

```
query allPostsByAuthor {
  allPostsByAuthor(
    author: "Nadia"
  ) {
    posts {
      id
      title
    }
    nextToken
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- Nadia 的所有文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "data": {
    "allPostsByAuthor": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world"
        },
        {
          "id": "11",
          "title": "Did you known...?"
        }
      ],
      "nextToken": null
    }
  }
}
```

- 我們使用標題為 "The cutest dog in the world" 的文章。請記下其 id，因為您稍後將會用到它。

現在，讓我們嘗試新增 dog 標籤。

- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 id 引數更新為您先前記下的值。

```
mutation addTag {
  addTag(id:10 tag: "dog") {
    id
    title
    tags
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 使用新標籤更新的文章。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
```

```
      "tags": [
        "dog"
      ]
    }
  }
}
```

您可以新增更多的標籤，如下所示：

- 更新變動以將 tag 引數變更為 puppy。

```
mutation addTag {
  addTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 使用新標籤更新的文章。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog",
        "puppy"
      ]
    }
  }
}
```

您也可以刪除標籤：

- 將以下變動貼到 Queries (查詢) 窗格。您也必須將 id 引數更新為您先前記下的值。

```
mutation removeTag {
```

```
  removeTag(id:10 tag: "puppy") {
    id
    title
    tags
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 貼文將會更新，且 puppy 標籤將會刪除。

```
{
  "data": {
    "addTag": {
      "id": "10",
      "title": "The cutest dog in the world",
      "tags": [
        "dog"
      ]
    }
  }
}
```

您也可以搜尋所有包含標籤的貼文：

- 將以下查詢貼到 Queries (查詢) 窗格中：

```
query allPostsByTag {
  allPostsByTag(tag: "dog") {
    posts {
      id
      title
      tags
    }
    nextToken
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- 具有 dog 標籤的所有文章將會傳回，如下所示：

```
{
```

```
  "data": {
    "allPostsByTag": {
      "posts": [
        {
          "id": "10",
          "title": "The cutest dog in the world",
          "tags": [
            "dog",
            "puppy"
          ]
        }
      ],
      "nextToken": null
    }
  }
}
```

使用清單和映射

除了使用 DynamoDB 集之外，您還可以使用 DynamoDB 清單和對應，在單一物件中建立複雜資料的模型。

讓我們新增功能以將評論新增到文章。這將被建模為 DynamoDB 中物件上的對應物件清單。Post

注意：在實際應用程式中，您會在它們自己的資料表中塑造評論。在本教學課程中，您將只是將它們新增至 Post 資料表。

- 選擇 Schema (結構描述) 標籤。
- 修改 Schema (結構描述) 窗格中新增 Comment 類型，如下所示：

```
type Comment {
    author: String!
    comment: String!
}
```

- 修改 Schema (結構描述) 窗格中的 Post 類型以新增 comments 欄位，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!]
}
```

- 修改 Schema (結構描述) 窗格中的 `Mutation` 類型來新增 `addComment` 變動，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!
}
```

- 選擇 儲存 。
- 在右側的 [資料類型] 窗格中，在 [變更] 類型上尋找新建立的 addComment 欄位，然後選擇 [附加]。
- 在 [資料來源名稱] 中，選擇 [資料PostDynamo表]。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
  "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)
    }
  }
}
```

這個更新表達式將會將包含新評論的清單附加到現有的 `comments` 清單。如果清單不存在，則會建立一個。

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
$utils.toJson($context.result)
```

- 選擇 **儲存** 。

呼叫 API 以新增評論

現在您已設定解析器，AWS AppSync 知道如何將傳入的`addComment`請求轉譯為 DynamoDB `UpdateItem` 作業。

讓我們透過將評論新增至您已新增標籤的相同文章，以嘗試此操作。

- 選擇 Queries (查詢) 標籤。
- 將以下查詢貼到 Queries (查詢) 窗格中：

```
mutation addComment {
  addComment(
    id:10
```

```
    author: "Steve"
    comment: "Such a cute dog."
  ) {
    id
    comments {
      author
      comment
    }
  }
}
```

- 選擇 Execute query (執行查詢) (橘色播放按鈕)。
- Nadia 的所有文章應該會出現在查詢窗格右側的結果窗格中。其看起來與下列類似：

```
{
  "data": {
    "addComment": {
      "id": "10",
      "comments": [
        {
          "author": "Steve",
          "comment": "Such a cute dog."
        }
      ]
    }
  }
}
```

如果您多次執行要求，系統會將多個評論附加到清單。

結論

在本教學課程中，您已經建置了可讓我們使用AWS AppSync 和 GraphQL 操作 DynamoDB 中的貼文物件的 API。如需詳細資訊，請參閱解析程式映射範本參考。

若要進行清理，您可以從主控台刪除 AppSync GraphQL API。

若要刪除針對本教學課程建立的 DynamoDB 表和 IAM 角色，您可以執行以下指令刪除AWSAppSyncTutorialForAmazonDynamoDB堆疊，或造訪AWS CloudFormation主控台並刪除堆疊：

```
aws cloudformation delete-stack \
    --stack-name AWSAppSyncTutorialForAmazonDynamoDB
```

教學課程：Lambda 解析器

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

您可以使AWS Lambda用AWS AppSync 來解析任何 GraphQL 欄位。例如，GraphQL 查詢可能會傳送呼叫至 Amazon Relational Database Service 服務 (Amazon RDS) 執行個體，而 GraphQL 突變可能會寫入 Amazon Kinesis 串流。在本節中，我們將示範如何撰寫 Lambda 函數，以根據 GraphQL 欄位作業的叫用來執行商務邏輯。

建立 Lambda 函數

下列範例顯示撰寫的 Lambda 函數，Node.js該函數會在部落格貼文應用程式中，在部落格貼文上執行不同作業。

```
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;
    }
};
```

此 Lambda 函數會依 ID 擷取貼文、新增貼文、擷取貼文清單，以及擷取給定貼文的相關貼文。

注意：Lambda 函數會使用上的`switch`陳述式`event.field`來判斷目前正在解析哪個欄位。

使用AWS管理主控台或AWS CloudFormation堆疊建立此 Lambda 函數。要從 CloudFormation 堆棧創建函數，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
```

您也可以在您的AWS帳戶中啟動美國西部 (奧勒岡) AWS 區域的AWS CloudFormation堆疊，從這裡：

設定 Lambda 的資料來源

建立 Lambda 函數之後，在AWS AppSync主控台中導覽至 GraphQL API，然後選擇 [資料來源] 索引標籤。

選擇 [建立資料來源]，輸入易記的資料來源名稱 (例如，**Lambda**)，然後針對 [資料來源類型] 選擇 [AWS Lambda函數]。對於「地區」，請選擇與功能相同的「區域」。（如果您從提供的 CloudFormation 堆棧中創建了函數，則該函數可能在 US-WEST-2 中。）對於函數 ARN，選擇 Lambda 函數的亞馬遜資源名稱 (ARN)。

選擇 Lambda 函數後，您可以建立新的 AWS Identity and Access Management (IAM) 角色 (為其AWS AppSync 指派適當權限)，或選擇具有下列內嵌政策的現有角色：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "lambda:InvokeFunction"
            ],
            "Resource": "arn:aws:lambda:REGION:ACCOUNTNUMBER:function/LAMBDA_FUNCTION"
        }
    ]
```

```
}
```

您還必須為 IAM 角色設定信任關係，如下所示：AWS AppSync

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

建立 GraphQL 結構描述

現在，資料來源已連接至您的 Lambda 函數，請建立 GraphQL 結構描述。

從AWS AppSync 主控台的資料架構編輯器中，確定您的資料架構符合下列資料架構：

```
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

現在您已經註冊了 Lambda 資料來源和有效的 GraphQL 結構描述，您可以使用解析器將 GraphQL 欄位連接到 Lambda 資料來源。

要創建一個解析器，你需要映射模板。若要進一步瞭解對應範本，請參閱Resolver Mapping Template Overview。

如需 Lambda 對應範本的詳細資訊，請參閱Resolver mapping template reference for Lambda。

在此步驟中，您可以為下列欄位附加解析程式至 Lambda 函數：`getPost(id:ID!): Post`、`allPosts: [Post]`、`addPost(id: ID!, author: String!, title: String, content: String, url: String): Post!`、和。`Post.relatedPosts: [Post]`

在AWS AppSync 主控台的資料架構編輯器中，選擇右側的「貼附解析器」。`getPost(id:ID!): Post`

然後，在「動作」功能表中選擇「更新執行階段」，然後選擇「單位解析器」（僅限 VTL）。

之後，選擇您的 Lambda 資料來源。在 request mapping template (請求映射範本) 區段中，選擇 Invoke And Forward Arguments (叫用並轉發引數)。

修改 `payload` 物件以新增欄位名稱。您的範本看起來應該如下所示：

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments":  $utils.toJson($context.arguments)
    }
}
```

在 response mapping template (回應映射範本) 區段中，選擇 Return Lambda Result (傳回 Lambda 結果)。

在此案例中，使用原本的基礎範本。它看起來應該如下所示：

```
$utils.toJson($context.result)
```

選擇 儲存 。您已成功連接第一個解析程式。對剩餘欄位重複此操作，如下所示：

addPost(id: ID!, author: String!, title: String, content: String, url: String): Post! 請求映射範本：

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "addPost",
        "arguments":  $utils.toJson($context.arguments)
    }
}
```

addPost(id: ID!, author: String!, title: String, content: String, url: String): Post! 回應映射範本：

```
$utils.toJson($context.result)
```

allPosts: [Post] 請求映射範本：

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "allPosts"
    }
}
```

allPosts: [Post] 回應映射範本：

```
$utils.toJson($context.result)
```

Post.relatedPosts: [Post] 請求映射範本：

```
{
```

```
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "relatedPosts",
        "source":  $utils.toJson($context.source)
    }
}
```

`Post.relatedPosts: [Post]` 回應映射範本：

```
$utils.toJson($context.result)
```

測試您的 GraphQL API

現在，您的 Lambda 函式已經連接到 GraphQL 解析程式，您可以使用主控台或用戶端應用程式執行一些變動和查詢。

在AWS AppSync 主控台的左側，選擇 [查詢]，然後貼入下列程式碼：

addPost Mutation

```
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 Query

```
query getPost {
```

```
    getPost(id: "2") {
        id
        author
        title
        content
        url
        ups
        downs
    }
}
```

allPosts Query

```
query allPosts {
    allPosts {
        id
        author
        title
        content
        url
        ups
        downs
        relatedPosts {
            id
            title
        }
    }
}
```

返回錯誤

任何給定的字段解析度都可能導致錯誤。使用 AWS AppSync，您可以從下列來源引發錯誤：

- 請求或回應映射範本
- Lambda 函數

從映射範本

若要引發故意錯誤，您可以使用速度範本語言 (VTL) 範本中的`$utils.error`輔助程式方法。它使用`errorMessage`、`errorType`，及選用的 `data` 值做為引數。發生錯誤時，`data` 可將額外的資料傳回給用戶端。`data` 物件會新增到 GraphQL 最後回應中的 `errors`。

以下範例說明如何在 Post.relatedPosts: [Post] 回應映射範本中使用它：

```
$utils.error("Failed to fetch relatedPosts", "LambdaFailure", $context.result)
```

這會產生類似下列的 GraphQL 回應：

```
{
    "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"
                }
            ]
        }
```

```
    ]
}
```

其中的 `allPosts[0].relatedPosts` 為 null，因為錯誤以及 `errorMessage`、`errorType` 和 `data` 出現在 `data.errors[0]` 物件中。

從 Lambda 函式

AWS AppSync 也瞭解 Lambda 函數所擲回的錯誤。Lambda 程式設計模型可讓您引發處理的錯誤。如果 Lambda 函數擲回錯誤，則AWS AppSync 無法解析目前的欄位。回應中只會設定從 Lambda 傳回的錯誤訊息。目前，您無法透過從 Lambda 函數引發錯誤，將任何多餘的資料傳回用戶端。

注意：如果您的 Lambda 函數引發未處理的錯誤，請AWS AppSync 使用 Lambda 設定的錯誤訊息。

下列 Lambda 函式會引發錯誤：

```
exports.handler = (event, context, callback) => {
    console.log("Received event {}", JSON.stringify(event, 3));
    callback("I fail. Always.");
};
```

這會傳回類似下列的 GraphQL 回應：

```
{
    "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."
        }
    ]
}
```

進階使用案例：批次處理

此範例中的 Lambda 函數有一個relatedPosts欄位，該欄位會傳回給定貼文的相關貼文清單。在範例查詢中，從 Lambda 函數呼叫allPosts欄位會傳回五個貼文。因為我們指定我們也想為每個返回的帖子relatedPosts進行解析，因此relatedPosts字段操作被調用五次。

```
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
        }
    }
}
```

雖然這在這個特定的例子中聽起來可能不大，但這種複合的過度提取可能會迅速破壞應用程序。

如果您要在同一查詢中，再次擷取傳回之相關 Posts 的 relatedPosts，那麼叫用的次數將大幅增加。

```
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
            }
        }
    }
}
```

在這個相對簡單的查詢中，AWS AppSync 會叫用 Lambda 函數 1 + 5 + 25 = 31 次。

這是一個相當常見的挑戰，通常稱為 N+1 問題 (在此例中，N = 5)，它可能會增加應用程式的延遲和成本。

解決此問題的方法之一，是將類似的欄位解析程式請求一起批次處理。在這個例子中，它不會讓 Lambda 函數解析單個給定帖子的相關帖子列表，而是可以解析給定批次帖子的相關帖子列表。

為了示範，讓我們將 `Post.relatedPosts: [Post]` 解析程式切換為啟用批次的解析程式。

在 AWS AppSync 主控台的右側，選擇現有的 `Post.relatedPosts: [Post]` 解析程式。將請求映射範本變更為以下內容：

```
{
    "version": "2017-02-28",
    "operation": "BatchInvoke",
    "payload": {
        "field": "relatedPosts",
        "source":  $utils.toJson($context.source)
    }
}
```

只有 `operation` 欄位從 `Invoke` 變更為 `BatchInvoke`。有效負載欄位現在會變成範本中指定之任何內容的陣列。在此範例中，Lambda 函數會接收下列項目做為輸入：

```
[
    {
        "field": "relatedPosts",
        "source": {
            "id": 1
        }
    },
    {
        "field": "relatedPosts",
        "source": {
            "id": 2
        }
    },
    ...
]
```

在請求對應範本中指定時BatchInvoke，Lambda 函數會接收要求清單並傳回結果清單。

具體而言，結果清單必須符合要求裝載項目的大小和順序，AWS AppSync 以便相應地符合結果。

在此批次處理範例中，Lambda 函數會傳回一批結果，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
]
```

Node.js 中的下列 Lambda 函數會示範此Post.relatedPosts欄位的批次處理功能，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;
    }
};
```

返回個別錯誤

前面的範例顯示，可以從 Lambda 函數傳回單一錯誤，或從對應範本引發錯誤。對於批次呼叫，從 Lambda 函數引發錯誤會將整個批次標記為失敗。對於發生無法恢復的錯誤的特定情況（例如與資料倉庫的連接失敗），這可能是可以接受的。但是，如果批次中的某些項目成功而其他項目失敗，則可能會

同時傳回錯誤和有效資料。因為AWS AppSync 要求對符合批次原始大小的清單元素進行批次回應，因此您必須定義可以區分有效資料與錯誤的資料結構。

例如，如果 Lambda 函數預期會傳回一批相關貼文，您可以選擇傳回物件清單，其中每個Response物件都有選用的資料、errorMessage 和 errorType 欄位。如果出現 errorMessage 欄位，表示發生錯誤。

下列程式碼會示範如何更新 Lambda 函數：

```
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;
    }
};
```

在此範例中，下列回應對應範本會剖析 Lambda 函數的每個項目，並引發任何發生的錯誤：

```
#if( $context.result && $context.result.errorMessage )
    $utils.error($context.result.errorMessage, $context.result.errorType,
 $context.result.data)
#else
    $utils.toJson($context.result.data)
#end
```

此範例會傳回類似下列的 GraphQL 回應：

```
{
  "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"
    }
  ]
}
```

設定最大批次處理大小

依預設，使用時BatchInvoke，AWS AppSync 會將請求分批傳送至您的 Lambda 函數，最多五個項目。您可以設定 Lambda 解析器的最大批次大小。

若要設定解析器上的最大批次處理大小，請使用 () 中的下列指令：AWS Command Line InterfaceAWS CLI

```
$ 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**
>
> 提供請求對應範本時，您必須使用BatchInvoke作業來使用批次處理。

您也可以使用下列命令在直接 Lambda 解析器上啟用和設定批次處理：

```
$ aws appsync create-resolver --api-id <api-id> --type-name Query --field-name
 relatedPosts \
 --data-source-name "<lambda-datasource>" \
 --max-batch-size X
```

VTL 範本的最大批次處理大小組態

對於具有 VTL 要求範本的 Lambda 解析器，除非在 VTL 中直接將其指定為作業，否則最大批次大小不會有任何作BatchInvoke用。同樣地，如果您正在執行頂層突變，則不會針對突變進行批次處理，因為 GraphQL 規格需要依序執 parallel 突變。

例如，採取以下突變：

```
type Mutation {
    putItem(input: Item): Item
    putItems(inputs: [Item]): [Item]
}
```

使用第一個突變，我們可以創建 10 Items 如下面的代碼片段所示：

```
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
    }
}
```

在此範例中，即使 Lambda 解析器中的最大批次大小設定為 10，也不Items會以 10 的群組進行批次處理。相反，它們將根據 GraphQL 規範順序執行。

要執行實際的批處理突變，您可以使用第二個突變遵循以下示例：

```
mutation MyMutation {
    putItems([$someItem1, $someItem2, $someItem3,$someItem4, $someItem5, $someItem6,
    $someItem7, $someItem8, $someItem9, $someItem10]) {
    id,
    name
    }
}
```

如需將批次處理與直接 Lambda 解析器搭配使用的詳細資訊，請參閱。直接 Lambda 解析器

教程：Amazon OpenSearch 服務解析器

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync 支援從您自己AWS帳戶中佈建的網域使用 Amazon OpenSearch 服務，前提是這些網域不存在於 VPC 中。在佈建網域之後，您可以使用資料來源來連線到這些網域，此時您可以在結構描述中設定解析程式，來進行 GraphQL 操作，例如查詢、變動和訂閱。本教學課程將會逐步說明一些常見的範例。

如需詳細資訊，請參閱的解析程式對映範本參考。 OpenSearch

一鍵設定

若要在已設定 Amazon OpenSearch 服務的中AWS AppSync 自動設定 GraphQL 端點，您可以使用以下AWS CloudFormation範本：

在 AWS CloudFormation 部署完成之後，您可以直接跳到執行 GraphQL 查詢與變動。

建立新的 OpenSearch 服務網域

若要開始使用本教學課程，您需要現有的 OpenSearch Service 網域。如果尚未擁有，您可以使用下列範例。請注意，建立 OpenSearch Service 網域最多可能需要 15 分鐘的時間，才能繼續將其與AWS AppSync資料來源整合。

```
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
```

您可以在您的AWS帳戶中的美國西部 2 (奧勒岡) 區域啟動下列AWS CloudFormation堆疊：

Launch Stack

設定 OpenSearch 服務的資料來源

建立 OpenSearch 服務網域之後，瀏覽至 AWS AppSync GraphQL API 並選擇資料來源索引標籤。選擇 [新增]，然後輸入資料來源的易記名稱，例如「oss」。然後選擇 Amazon OpenSearch 網域做為資料來源類型，選擇適當的區域，您應該會看到列出您的 OpenSearch 服務網域。選擇網域之後，您可以建立新的角色 (AWS AppSync 將會指派角色適用的權限)，或是選擇現有的角色，後者具有下列的內嵌政策：

```
{
    "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/*"
            ]
        }
    ]
}
```

您同時需要設定該角色與 AWS AppSync 的信任關係：

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

```
        }
    ]
}
```

此外， OpenSearch 服務網域擁有自己的存取政策，您可以透過 Amazon OpenSearch 服務主控台進行修改。您將需要新增類似下列的原則，並針對 OpenSearch Service 網域採取適當的動作和資源。請注意，主體將是 AppSync 資料來源角色，如果您讓主控台建立此角色，您可以在 IAM 主控台中找到該角色。

```
{
    "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/*"
        }
    ]
}
```

連結解析程式

現在，資料來源已連線到您的 OpenSearch 服務網域，您可以使用解析器將其連線到 GraphQL 結構描述，如下列範例所示：

```
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
}
...
```

請注意，有使用者定義的 Post 類型與 id 欄位。在下面的例子中，我們假設有一個過程（可以自動化）將此類型放入您的 OpenSearch Service 域中，該過程將映射到的路徑根目錄/post/_doc，其中post是索引。從此根路徑中，您可以執行個別文件搜尋、使用萬用字元搜尋/id/post*，或使用路徑執行多文件搜尋。/post/_search例如，如果您有另一個名為的類型User，您可以在名為的新索引下為文件建立索引user，然後使用的路徑執行搜尋/user/_search。

從 AWS AppSync 主控台中的結構描述編輯器，修改先前的 Posts 結構描述，以加入 searchPosts 查詢：

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  searchPosts: [Post]
}
```

儲存結構描述。在右側的 searchPosts 中，選擇 Attach resolver (附加解析程式)。在「動作」功能表中，選擇「更新執行階段」，然後選擇「單位解析器（僅限 VTL）」。然後，選擇您的 OpenSearch 服務數據源。在 request mapping template (請求映射範本) 區段中，選取 Query posts (查詢文章) 的下拉式清單，來取得基本範本。將 path 修改為 /post/_search。它看起來應該如下所示：

```
{
    "version":"2017-02-28",
```

```
    "operation":"GET",
    "path":"/post/_search",
    "params":{
        "headers":{},
        "queryString":{},
        "body":{
            "from":0,
            "size":50
        }
    }
}
```

這假設前面的結構描述具有已在post欄位下的 OpenSearch Service 中編製索引的文件。如果您建立的資料結構不同，將會需要進行對應的更新。

在響應映射模板部分下，如果要從 OpenSearch 服務查詢獲取數據結果並轉換為 GraphQL，則需要指定適當的_source過濾器。使用下列的範本：

```
[
    #foreach($entry in $context.result.hits.hits)
    #if( $velocityCount > 1 ) , #end
    $utils.toJson($entry.get("_source"))
    #end
]
```

修改您的搜尋

前述的請求映射範本，會針對所有記錄執行簡單的查詢。假設您想要根據特定的作者來進行搜尋。此外，假設您希望該位作者成為 GraphQL 查詢中所定義的引數。請在 AWS AppSync 主控台的結構描述編輯器中，新增 allPostsByAuthor 查詢：

```
type Query {
  getPost(id: ID!): Post
  allPosts: [Post]
  allPostsByAuthor(author: String!): [Post]
  searchPosts: [Post]
}
```

現在選擇附加解析器並選擇 OpenSearch Service 數據源，但在響應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)
                }
            }
        }
    }
}
```

請注意，body 會填入 author 欄位的查詢詞語，此欄位會做為引數，從用戶端直接傳來。您可以選擇性地預先填入資訊 (例如標準的文字內容)，或甚至使用其他公用程式。

如果您正在使用此解析程式，請使用與先前範例相同的資訊，來填寫回應映射範本。

將資料新增至 OpenSearch 服務

您可能想要將資料新增至您的 OpenSearch 服務網域，因為 GraphQL 突變的結果。這是一個具有搜尋和其他用途的強大機制。因為您可以使用 GraphQL 訂閱讓資料即時化，因此它可以做為通知用戶端 OpenSearch 服務網域中資料更新的機制。

在 主控台中返回 SchemaAWS AppSync (結構描述) 頁面，然後針對 變動選擇 Attach resolveraddPost( ) (附加解析程式)。再次選取「 OpenSearch 服務」資料來源，然後針對**Posts**網要使用下列回應對應範本：

```
{
    "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)
        }
    }
}
```

和之前相同，這是如何建構資料結構的一個範例。如果有不同的欄位名稱或索引，您需要適當地更新 path 與 body。此範例也示範了如何利用 $context.arguments，以 GraphQL 變動引數來填寫範本。

在繼續之前，請使用以下響應映射模板，該模板將返回突變操作的結果或錯誤信息作為輸出：

```
#if($context.error)
    $util.toJson($ctx.error)
#else
    $util.toJson($context.result)
#end
```

擷取單一文件

最後，如果您想要在結構描述中使用 getPost(id:ID) 查詢，以傳回個別的文件，請在 AWS AppSync 主控台的結構描述編輯器中找出此查詢，然後選擇 Attach resolver (附加解析程式)。再次選取「 OpenSearch 服務」資料來源，然後使用下列對應範本：

```
{
    "version":"2017-02-28",
    "operation":"GET",
    "path": $util.toJson("post/_doc/$context.arguments.id"),
    "params":{
        "headers":{},
        "queryString":{},
        "body":{}
    }
}
```

由於上述的 path 使用 id 引數與空的內容，因此這會傳回空的單一文件。不過，因為現在傳回的是單一項目而非清單，您需要使用下列回應映射範本：

```
$utils.toJson($context.result.get("_source"))
```

執行查詢與變動

您現在應該可以針對 OpenSearch 服務網域執行 GraphQL 作業。請瀏覽至 主控台的 QueriesAWS AppSync (查詢) 標籤，然後新增記錄：

```
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
       )
}
```

您將在右側看到突變的結果。同樣地，您現在可以針對 OpenSearch 服務網域執行`searchPosts`查詢：

```
query searchPosts {
    searchPosts {
        id
        title
        author
        content
    }
}
```

最佳實務

- OpenSearch 服務應該用於查詢數據，而不是作為您的主數據庫。您可能想要將 OpenSearch 服務與 Amazon DynamoDB 搭配使用，如結合 Graph QL 解析器中所述。
- 請只透過允許 AWS AppSync 服務角色存取叢集，來提供對您網域的存取。
- 您可以在開發時少量使用最低成本的叢集，然後在進入正式生產時，轉移到具有高可用性 (HA) 的較大型叢集。

教學課程：本機解析程式

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync 可讓您使用支援的資料來源 (AWS LambdaAmazon DynamoDB 或亞馬遜 OpenSearch 服務) 來執行各種操作。不過，在某些情況下，可能不需要呼叫支援的資料來源。

這就是本機解析程式的實用之處。本機解析程式只是將請求映射範本的結果轉發到回應映射範本，而不是呼叫遠端資料來源。欄位解析度不會離開 AWS AppSync。

本機解析程式適用於多種使用案例。最常見的使用案例是發佈通知而不觸發資料來源呼叫。為了示範此使用案例，我們來建置一個分頁應用程式，讓使用者可以互相傳遞頁面。此範例使用訂閱，因此，如果您不熟悉訂閱，可以遵循即時資料教學課程。

建立分頁應用程式

在我們的分頁應用程式中，用戶端可以訂閱收件匣，並向其他用戶端傳送頁面。每個頁面皆包含訊息。結構描述如下：

```
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
}
```

現在將解析程式附加到 Mutation.page 欄位。在 Schema (結構描述) 窗格中，按一下右側窗格上欄位定義旁邊的 Attach Resolver (附加解析程式)。建立類型為 None 的新資料來源，並將其命名PageDataSource。

在請求映射範本部分，輸入：

```
{
  "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()"
  }
}
```

在回應映射範本部分，則選取預設的 Forward the result (轉發結果)。儲存您的解析程式。您的應用程式現已就緒，可以開始傳遞頁面！

傳送和訂閱頁面

用戶端必須先訂閱收件匣，才能接收頁面。

現在就讓我們在 Queries (查詢) 窗格中執行 inbox 訂閱：

```
subscription Inbox {
    inbox(to: "Nadia") {
        body
        to
        from
        sentAt
    }
}
```

每當調用Mutation.page突變時，Nadia 都會收到頁面。現在透過執行變動來叫用變動：

```
mutation Page {
    page(to: "Nadia", body: "Hello, World!") {
        body
        to
        from
        sentAt
    }
}
```

我們剛剛在不離開 AWS AppSync 之下傳送和接收頁面，藉此示範本機解析程式的用法。

教學課程：合併 GraphQL 解析程式

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

GraphQL 結構描述中的解析程式與欄位具有一對一的關係，以及高度的彈性。由於資料來源是在解析程式上與結構描述個別獨立設定，因此您可以透過不同的資料來源來解析或操縱 GraphQL 的類型，對結構描述進行混合和匹配，最大程度地滿足您的需求。

下列範例案例示範如何混合和比對結構描述中的資料來源。在開始之前，我們建議您先熟悉設定 Amazon DynamoDB 和 Amazon OpenSearch 服務的資料來源和解析器，如先前教學中所述。AWS Lambda

範例結構描述

下面的模式具有 3 個Query操作Post和 3 Mutation 個操作定義的類型：

```
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
}
```

在此範例中，總共要連結 6 個解析程式。有一種可能的方法是讓所有這些都來自稱為 Amazon DynamoDB 表`Posts`，其中執行掃描並`AllPosts``searchPosts`執行查詢，如 DynamoDB 解析器對映範本參考中所述。不過，有一些替代方案可以滿足您的業務需求，例如讓這些 GraphQL 查詢從 Lambda 或 OpenSearch 服務解析。

透過解析程式修改資料

您可能需要從 DynamoDB (或 Amazon Aurora) 等資料庫將結果傳回給已變更某些屬性的用戶端。這可能是資料類型的格式所造成 (例如用戶端上時間戳記的差異)，或是為了處理回溯相容性的問題。為了說明目的，在下列範例中，AWS Lambda函數會在每次叫用 GraphQL 解析器時為部落格貼文指派亂數，藉此操控部落格貼文的上票和下票：

```
'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);
    });
};
```

這是一個非常有效的 Lambda 函式，可以連結至 GraphQL 結構描述中的 `AllPosts` 欄位，如此，任何傳回所有結果的查詢，都會取得支持票和不支持票票數的隨機數量。

DynamoDB 和服務 OpenSearch

對於某些應用程式，您可能會對 DynamoDB 執行突變或簡單的查閱查詢，並讓背景程序將文件傳輸至服務。 OpenSearch 然後，您可以簡單地將`searchPosts`解析器附加到 OpenSearch 服務資料來源，並使用 GraphQL 查詢傳回搜尋結果 (來自 DynamoDB 中的資料)。在為您的應用程式加入進階的搜尋操作時 (例如關鍵字詞、模糊字詞比對或甚至地理空間查詢)，這可以是非常強大的功能。從 DynamoDB 傳輸資料可透過 ETL 程序完成，或者您也可以使用 Lambda 從 DynamoDB 進行串流。您可以使用帳戶中美國西部 2 (奧勒岡) 區域的下列AWS CloudFormation堆疊來啟動此項AWS目的完整範例：

此範例中的結構描述可讓您使用 DynamoDB 解析器新增貼文，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
    }
}
```

這會將資料寫入 DynamoDB，然後透過 Lambda 將資料串流至 Amazon OpenSearch 服務，您可以依不同欄位搜尋所有貼文。例如，由於資料位於 Amazon Ser OpenSearch vice 中，因此您可以使用自由格式的文字來搜尋作者或內容欄位，甚至包含空格，如下所示：

```
query searchName{
    searchAuthor(name:"   Nadia   "){
        id
        title
        content
    }
}

query searchContent{
    searchContent(text:"test"){
        id
        title
        content
    }
}
```

由於資料會直接寫入 DynamoDB，因此您仍然可以使用`allPosts{...}`和`singlePost{...}`查詢對資料表執行有效率的清單或項目查閱作業。此堆疊會針對 DynamoDB 串流使用下列範例程式碼：

注意：這些程式碼僅為示範。

```
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);
};
```

然後，您可以使用 DynamoDB 串流將其附加到主索引鍵的 DynamoDB 表格`id`，而對 DynamoDB 來源的任何變更都會串流至您的服務網域。 OpenSearch 如需進行這項設定的詳細資訊，請參閱 DynamoDB 串流文件。

教學課程：Batch 解析器

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync 支援在單一區域中的一個或多個表格中使用 Amazon DynamoDB 批次操作。支援的操作包括 `BatchGetItem`、`BatchPutItem` 和 `BatchDeleteItem`。您可以在 AWS AppSync 中使用這些功能來執行工作，例如：

- 在單次查詢中傳遞索引鍵清單，並從資料表傳回結果
- 在單次查詢中讀取一個或多個資料表的記錄
- 將大量的記錄寫入一個或多個資料表
- 在可能具有關聯的多個資料表中，有條件地寫入或刪除記錄

在 DynamoDB 中使用批次作業AWS AppSync 是一項進階技術，需要對後端作業和表格結構的額外思考和知識。此外，在 AWS AppSync 中的批次操作和非批次操作有兩項主要的差異：

- 對於解析程式將會存取的所有資料表，資料來源角色都必須具有權限。
- 解析程式資料表規格包含於映射範本中。

許可

如同其他的解析程式，您需要在 AWS AppSync 中建立資料來源，然後建立角色或使用現有的角色。由於批次操作需要 DynamoDB 表上的不同權限，因此您需要授與已設定的角色權限以進行讀取或寫入動作：

```
{
    "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：在 AWS AppSync 中，角色會連結至資料來源，而會針對資料來源叫用欄位上的解析程式。設定為針對 DynamoDB 擷取的資料來源只會指定一個表格，以保持組態簡單。因此，在單一解析程式中針對多個資料表執行批次作業時 (這是一項更為進階的工作)，您必須授予權限給該資料來源上的角色，以存取解析程式將會與其互動的所有資料表。這會在上述 IAM 政策的 Resource (資源) 欄位中完成。針對批次操作所要呼叫的資料表進行設定，這項動作是在解析程式範本中完成，我們將會在下列內容中說明此範本。

資料來源

為簡單起見，我們將針對本教學課程中使用的所有解析程式，使用相同的資料來源。在資料來源索引標籤上，建立新 DynamoDB 資料來源並為其命名。BatchTutorial資料表可以使用任何名稱，因為資料表名稱在批次作業中是指定為請求映射範本的一部分。我們會將資料表命名為 `empty`。

在本教學課程中，具備下列內嵌政策的任何角色皆可使用：

```
{
    "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/*"
            ]
        }
    ]
}
```

單一資料表批次

例如，假設有一個名為 Posts 的資料表，您想要利用批次操作來新增和移除此資料表的項目。使用下列的結構描述，請注意，我們會將 ID 清單傳入查詢：

```
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
}
```

使用下列「請求對應範本」將解析器附加至`batchAdd()`欄位。這會自動擷取 GraphQL `input PostInput` 類型的每個項目，並建置 `BatchPutItem` 作業所需的對應圖：

```
#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)
    }
}
```

在此情況中的回應映射範本只是簡單的直接傳遞，但是該資料表名稱會以 `..data.Posts` 的形式附加至內容物件，如下所示：

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

現在請瀏覽至 主控台的 QueriesAWS AppSync (查詢) 頁面，並執行下列的 batchAdd 變動動作：

```
mutation add {
    batchAdd(posts:[{
            id: 1 title: "Running in the Park"},{
            id: 2 title: "Playing fetch"
        }]){
            id
            title
    }
}
```

您應該會看到列印在畫面上的結果，並且可以透過 DynamoDB 主控台獨立驗證這兩個值都寫入貼文表。

接下來，使用下列「請求對應範本」將解析器附加至`batchGet()`欄位。這會自動擷取 GraphQL `ids:[]` 類型的每個項目，並建置 `BatchGetItem` 作業所需的對應圖：

```
#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"}
                }
        }
    }
}
```

回應映射範本仍然只是簡單的直接傳遞，該資料表名稱也會再次地以 `..data.Posts` 的形式附加至內容物件：

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

現在請返回 主控台的 QueriesAWS AppSync (查詢) 頁面，並執行下列的 batchGet 查詢動作：

```
query get {
    batchGet(ids:[1,2,3]){
        id
        title
    }
}
```

這應該會針對您先前所新增的兩個 `id` 值，傳回其結果。請注意，值為 `null` 的 `id` 傳回了 3 值。這是因為在 Posts 資料表中，尚未有任何記錄具有該值。另請注意，AWS AppSync 會依索引鍵傳入查詢的相同順序傳回結果，這是 AWS AppSync 代表您執行的一項額外功能。因此，如果切換到 `batchGet(ids:[1,3,2)`，順序將會變更。您也會知道哪個 `id` 傳回 `null` 值。

最後，使用以下「請求映射模板」將解析器附加到`batchDelete()`字段中。這會自動擷取 GraphQL `ids:[]` 類型的每個項目，並建置 `BatchGetItem` 作業所需的對應圖：

```
#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)
    }
}
```

回應映射範本仍然只是簡單的直接傳遞，該資料表名稱也會再次地以 `..data.Posts` 的形式附加至內容物件：

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

現在請返回 主控台的 QueriesAWS AppSync (查詢) 頁面，並執行下列的 batchDelete 變動動作：

```
mutation delete {
    batchDelete(ids:[1,2]){ id }
}
```

id 1 和 2 的記錄現在應已刪除。如果重新執行先前的 `batchGet()` 查詢，這些應該會傳回 `null`。

多重資料表批次

AWS AppSync 還可讓您跨表執行批次作業。來試試建置更複雜的應用程式。試想建置一個寵物健康狀態 (Pet Health) 應用程式，其中會有感應器呈報寵物的位置和體溫。感應器是由電池供電，而且每隔幾分鐘就會試著連線到網路。當感應器建立連線時，會將其讀數傳送到我們的 AWS AppSync API。觸發條件接著就會分析資料，然後將儀表板呈現給寵物的主人。讓我們著重介紹感應器與後端資料存放區之間的互動。

先決條件是，讓我們先建立兩個 DynamoDB 表；locationReadings 會儲存感應器位置讀數，而 temperatureReadings 則會儲存感測器溫度讀數。這兩個資料表碰巧擁有相同的主索引鍵結構：`sensorId (String)` 為分割區索引鍵、`timestamp (String)` 為排序索引鍵。

讓我們使用下列 GraphQL 結構描述：

```
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 -記錄傳感器讀數

感應器必須能夠在連線到網際網路之後傳送其讀數。感應器將使用 GraphQL 欄位 `Mutation.recordReadings` 這個 API 來執行此項動作。讓我們連結解析程式，讓 API 能夠開始運作。

選取欄位旁邊的「附加`Mutation.recordReadings`」。在下一個畫面中，選取在教學課程開始時所建立的同一個 `BatchTutorial` 資料來源。

讓我們來新增下列請求映射範本：

請求映射範本

```
## 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)
    }
}
```

如您所見，`BatchPutItem` 操作可讓我們指定多個資料表。

讓我們使用下列回應映射範本。

回應映射範本

```
## 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)
```

進行批次操作時，呼叫可能會同時傳回錯誤和結果。在這種情況中，我們可以隨意進行一些額外的錯誤處理。

注意：$utils.appendError() 的使用類似於 $util.error()，主要差別在於前者不會中斷映射範本的評估，而是在欄位出現錯誤時發出訊號，但允許範本的評估，進而將資料傳回給呼叫程式。如果應用程式需要傳回部分結果，建議您使用 $utils.appendError()。

儲存解析程式，然後瀏覽至 主控台的 QueriesAWS AppSync (查詢) 頁面。讓我們開始傳送一些感應器讀數吧！

執行下列的變動：

```
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
    }
  }
}
```

我們會在一次的變動中傳送 10 個感應器讀數，讀數的資料會分置於兩個資料表。使用 DynamoDB 主控台驗證資料是否同時顯示在「locationReadings」和「temperatureReadings」表中。

BatchDeleteItem -刪除傳感器讀數

同樣地，我們也需要刪除分批的感應器讀數。讓我們使用 `Mutation.deleteReadings` GraphQL 欄位來進行這項動作。選取欄位旁邊的「附加`Mutation.recordReadings`」。在下一個畫面中，選取在教學課程開始時所建立的同一個 `BatchTutorial` 資料來源。

讓我們使用下列請求映射範本。

請求映射範本

```
## 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)
    }
}
```

這個回應映射範本和我們用於 Mutation.recordReadings 的相同。

回應映射範本

```
## 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)
```

儲存解析程式，然後瀏覽至 主控台的 QueriesAWS AppSync (查詢) 頁面。現在，讓我們刪除一些感應器讀數吧！

執行下列的變動：

```
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
    }
  }
}
```

透過 DynamoDB 主控台驗證這兩個讀數是否已從「locationReadings」和「temperatureReadings 讀數」表格中刪除。

BatchGetItem -檢索讀數

我們寵物健康狀態 (Pet Health) 應用程式另一個常用的操作，就是在特定的時間點擷取感應器的讀數。試試將解析程式連接到結構描述中的 `Query.getReadings` GraphQL 欄位。選取 Attach (附加)，然後在下一個畫面上，選取在教學課程開始時所建立的同一個 `BatchTutorial` 資料來源。

讓我們來新增下列請求映射範本。

請求映射範本

```
## 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
        }
    }
}
```

請注意，我們現在正在使用該BatchGetItem操作。

我們的回應映射範本將會有些微的不同，因為我們選擇傳回 SensorReading 清單。讓我們將叫用結果映射到所需的結構。

回應映射範本

```
## 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)
```

儲存解析程式，然後瀏覽至 主控台的 QueriesAWS AppSync (查詢) 頁面。現在，讓我們擷取感應器讀數吧！

執行下列的查詢：

```
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
    }
  }
```

```
}
```

我們已經成功示範了使用. AWS AppSync

錯誤處理

在 AWS AppSync 中，資料來源的操作有時可能會傳回部分結果。我們將會使用部分結果這個詞彙，來表示操作的輸出包含一些資料和錯誤的情況。因為錯誤處理原本就是隨應用程式而有不同，AWS AppSync 提供了機會，讓您在回應映射範本中處理錯誤。如果發生解析程式呼叫錯誤，在文字內容中會出現 `$ctx.error`。呼叫錯誤一律包含訊息和類型，可做為 `$ctx.error.message` 和 `$ctx.error.type` 屬性存取。在回應映射範本呼叫期間，您可以用三種方式來處理部分結果：

1. 藉由只傳回資料來抑制呼叫錯誤
2. 藉由停止進行回應映射範本的評估 (這不會傳回任何資料)，來產生錯誤 (使用 `$util.error(...)`)。
3. 附加錯誤 (使用 `$util.appendError(...)`)，而且也傳回資料

讓我們透過 DynamoDB 的批次操作來示範上述三點！

DynamoDB 批次操作

使用 DynamoDB 批次操作時，批次作業就有可能部分完成。也就是說，請求的項目或索引鍵可以有一些尚未處理完成。如果 AWS AppSync 無法完成批次作業，在文字內容中將會顯示未處理的項目和呼叫錯誤。

我們將會使用 `Query.getReadings` 欄位組態 (取自於本教學課程先前區段所說明的 `BatchGetItem` 操作) 來建置錯誤處理功能。這次，讓我們假設在執行 `Query.getReadings` 欄位時，`temperatureReadings` DynamoDB 資料表用盡了佈建的輸送量。DynamoDB 在第二次嘗試處理批次中的AWS AppSync 剩餘元素ProvisionedThroughputExceededException時提出了一個。

下列的 JSON 呈現了在 DynamoDB 批次呼叫之後、回應映射範本評估之前的序列化內容。

```
{
  "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": []
}
```

對於內容的幾個注意事項：

- 呼叫錯誤已在`$ctx.error`上下文中設定 AWS AppSync，且錯誤類型已設定為 DynamoDB:。ProvisionedThroughputExceededException
- 即使出現錯誤，也會針對每個資料表對應結果 (`$ctx.result.data`)
- 未處理的索引鍵可透過 `$ctx.result.data.unprocessedKeys` 取得。在這裡，AWS AppSync 無法利用索引鍵 (sensorId:1、時間戳記：2018-02-01T17:21:05.000+08:00) 來擷取項目，因為資料表輸送量不足。

注意：對於 `BatchPutItem`，則為 `$ctx.result.data.unprocessedItems`。如果是 `BatchDeleteItem`，則為 `$ctx.result.data.unprocessedKeys`。

讓我們用三種不同的方式來處理這項錯誤。

1. 抑制呼叫錯誤

傳回資料而不處理呼叫錯誤，可有效地抑制錯誤，讓指定 GraphQL 欄位的結果一律成功。

我們所編寫的回應映射範本，只熟悉和著重於結果的資料。

回應映射範本：

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

GraphQL 回應：

```
{
  "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
      }
    ]
  }
}
```

錯誤回應中不會加入任何錯誤，因為只處理了資料。

2. 丟出錯誤來中止範本的執行

如果從用戶端的角度，部分故障應視為完全故障，則您可以中止範本的執行，來防止傳回資料。`$util.error(...)` 公用程式方法可確實完成這項動作。

回應映射範本：

```
## 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 回應：

```
{
  "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 DynamoDB 批次作業可能已經傳回一些結果，我們還是選擇丟出錯誤，如此 `getReadings` GraphQL 欄位為 null，而錯誤也會加入到 GraphQL 回應的 errors (錯誤) 區塊。

3. 附加錯誤，以同時傳回資料和錯誤

在某些情況中，為提供更好的使用者體驗，應用程式可以傳回部分結果，並通知其用戶端有未處理的項目。用戶端可以決定是否要重試，或是轉譯錯誤後傳回給最終使用者。`$util.appendError(...)` 是一項公用程式方法，可讓應用程式的設計人員將錯誤附加於文字內容，但不會干擾到範本的評估，進而完成前述的動作。在評估範本之後，AWS AppSync 會處理所有的內容錯誤，方式是將這些錯誤附加到 GraphQL 回應的錯誤區塊。

回應映射範本：

```
#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)
```

我們會同時轉傳呼叫錯誤，以及 GraphQL 回應錯誤區塊中的 unprocessedKeys (未處理的索引鍵) 項目。`getReadings` 欄位也會傳回來自 locationReadings 資料表的部分資料，如下列的回應中所示。

GraphQL 回應：

```
{
  "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. (...)"
    }
  ]
}
```

教學課程：交易解析器

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync 支援在單一區域中的一或多個表格中使用 Amazon DynamoDB 交易操作。支援的操作包括 `TransactGetItems` 和 `TransactWriteItems`。您可以在 AWS AppSync 中使用這些功能來執行工作，例如：

- 在單次查詢中傳遞索引鍵清單，並從資料表傳回結果
- 在單次查詢中讀取一個或多個資料表的記錄
- 以某 all-or-nothing 種方式將事務中的記錄寫入一個或多個表
- 符合某些條件時執行交易

許可

如同其他的解析程式，您需要在 AWS AppSync 中建立資料來源，然後建立角色或使用現有的角色。由於交易操作需要 DynamoDB 表上的不同權限，因此您需要授與已設定的角色權限以進行讀取或寫入動作：

```
{
    "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：在 AWS AppSync 中，角色會連結至資料來源，而會針對資料來源叫用欄位上的解析程式。設定為針對 DynamoDB 擷取的資料來源只會指定一個表格，以保持組態簡單。因此，在單一解析程式中針對多個資料表執行交易操作時 (這是一項更為進階的工作)，您必須授予權限給該資料來源上的角色，以存取解析程式將會與其互動的所有資料表。這會在上述 IAM 政策的 Resource (資源) 欄位中完成。針對資料表設定交易呼叫是在解析程式範本中完成，我們將會在下列內容中說明此範本。

資料來源

為簡單起見，我們將針對本教學課程中使用的所有解析程式，使用相同的資料來源。在資料來源索引標籤上，建立新 DynamoDB 資料來源並為其命名。TransactTutorial資料表可以使用任何名稱，因為資料表名稱在交易操作中是指定為請求映射範本的一部分。我們會將資料表命名為 `empty`。

我們將兩個資料表稱為 savingAccounts 和 checkingAccounts，兩者皆具有 `accountNumber` 做為分割區索引鍵，以及 transactionHistory 資料表，具有 `transactionId` 做為分割區索引鍵。

在本教學課程中，具備下列內嵌政策的任何角色皆可使用。將 region 和 accountId 更換為您的區域和帳戶 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/*"
            ]
        }
    ]
}
```

交易

在此範例中，內容是傳統的銀行交易，我們將使用 TransactWriteItems 以進行下列操作：

- 從存款帳戶轉帳至支票帳戶
- 為每筆交易產生新的交易記錄

然後，我們會使用 TransactGetItems 擷取存款帳戶和支票帳戶中的詳細資料。

定義 GraphQl 結構描述，如下所示：

```
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 -填入帳戶

為了在帳戶之間轉帳，我們需要在表格中填入詳細資料。我們將使用 GraphQL 操作 `Mutation.populateAccounts` 來執行此作業。

在「架構」部分中，單擊`Mutation.populateAccounts`操作旁邊的「附加」。前往 VTL 單位解析器，然後選擇相同`TransactTutorial`的資料來源。

現在使用下列請求映射範本：

請求映射範本

```
#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)
}
```

以及下列回應映射範本：

回應映射範本

```
#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)
```

儲存解析程式並導覽至AWS AppSync 主控台的 [查詢] 區段以填入帳戶。

執行下列的變動：

```
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
    }
  }
}
```

我們在一個變動中填入 3 個存款帳戶和 3 個支票帳戶。

使用 DynamoDB 主控台驗證資料是否顯示在「儲存帳戶」和「checkingAccounts」表格中。

TransactWriteItems -轉賬

使用以下請求映射模板將解析器附加到`transferMoney`突變。請注意，`amounts`、`savingAccountNumbers` 以及 `checkingAccountNumbers` 的值是相同的。

```
#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)
}
```

我們將會在單一 TransactWriteItems 操作中有 3 筆銀行交易。使用下列回應映射範本：

```
#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)
```

現在導航到AWS AppSync 控制台的「查詢」部分，然後執行 transferMoney 突變，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
    }
  }
}
```

我們已傳送單一變動中的 2 筆銀行交易。使用 DynamoDB 主控台驗證資料是否顯示在「儲存帳戶」、「checking Accounts」和「transactionHistory 記錄」表格中。

TransactGetItems -檢索帳戶

為了擷取單一交易請求中存款帳戶和支票帳戶的詳細資料，我們會將解析程式附加至結構模式中的 Query.getAccounts GraphQL 操作。選取「貼附」，移至「VTL 單位解析器」，然後在下一個畫面中，挑選自學課程開始時建立的相同TransactTutorial資料來源。設定範本，如下所示：

請求映射範本

```
#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)
}
```

回應映射範本

```
#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)
```

儲存解析程式並瀏覽至主控台的 [查詢] 區段。AWS AppSync 若要擷取存款帳戶和支票帳戶，請執行下列查詢：

```
query getAccounts {
  getAccounts(
    savingAccountNumbers: ["1", "2", "3"],
    checkingAccountNumbers: ["1", "2"]
  ) {
    savingAccounts {
      accountNumber
      username
      balance
    }
    checkingAccounts {
      accountNumber
      username
      balance
    }
  }
}
```

我們已經成功地示範了使用. AWS AppSync

教學課程：HTTP 解析器

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync 除了可解析 GraphQL 欄位的任意 HTTP 端點之外AWS Lambda，您還可以使用支援的資料來源 (即 Amazon DynamoDB、亞馬遜 OpenSearch 服務或 Amazon Aurora) 執行各種操作。在您的 HTTP 端點可供使用之後，您可以使用資料來源連線到這些端點。然後，您可以在結構描述中設定解析程式以執行 GraphQL 操作，例如查詢、變動和訂閱。本教學將逐步說明一些常見的範例。

在本教學課程中，您將 REST API (使用 Amazon API Gateway 和 Lambda 建立) 搭配 AWS AppSync GraphQL 端點使用。

一鍵設定

如果您想要在中自動設定 GraphQL 端點並AWS AppSync 設定 HTTP 端點 (使用 Amazon API Gateway 和 Lambda)，您可以使用下列AWS CloudFormation範本：

Launch Stack

建立 REST API

您可以使用以下 AWS CloudFormation 範本來設定適用於本教學的 REST 端點：

Launch Stack

AWS CloudFormation 堆疊會執行下列步驟：

1. 設定 Lambda 函數，其中包含您微服務的商業邏輯。
2. 使用以下端點/方法/內容類型組合設置 API Gateway REST API：

API 資源路徑	HTTP 方法	已支援的內容類型
/v1/users	POST	application/json

API 資源路徑	HTTP 方法	已支援的內容類型
/v1/users	GET	application/json
/v1/users/1	GET	application/json
/v1/users/1	PUT	application/json
/v1/users/1	DELETE	application/json

建立 GraphQL API

若要在 AWS AppSync 中建立 GraphQL API：

- 開啟 AWS AppSync 主控台，並選擇 Create API (建立 API)。
- 針對 API 名稱，輸入 `UserData`。
- 選擇 Custom schema (自訂結構描述)。
- 選擇 建立 。

AWS AppSync 主控台會使用 API 金鑰身分驗證模式為您建立新 GraphQL API。您可以使用主控台來設定 GraphQL API 的其餘部分，並在本教學的其餘部分中對其執行查詢。

建立 GraphQL 結構描述

現在您有一個 GraphQL API，讓我們來建立 GraphQL 結構描述吧。在 AWS AppSync 主控台的結構描述編輯器中，確認您的結構描述與以下結構描述相符：

```
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
}
```

設定您的 HTTP 資料來源

若要設定您的 HTTP 資料來源，請執行以下作業：

- 在DataSources索引標籤上，選擇 [新增]，然後輸入資料來源的易記名稱 (例如，HTTP)。
- 在 Data source type (資料來源類型) 中選擇 HTTP。
- 將端點設定為所建立的 API Gateway 端點。請確定您沒有在端點中包含階段名稱。

注意：目前 AWS AppSync 僅支援公有端點。

備註：如需有關AWS AppSync 服務辨識之認證授權單位的詳細資訊，請參閱 HTTPS 端點辨識的憑證授權單位 (CA)。AWS AppSync

設定解析程式

在此步驟中，您會將 http 資料來源連線到 getUser 查詢。

設定解析程式：

- 選擇 Schema (結構描述) 標籤。

- 在 [查詢類型] 下方右側的 [資料類型] 窗格中，找到 getUser 欄位並選擇 [附加]。
- 在 Data source name (資料來源名稱) 中，選擇 HTTP。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
    "version": "2018-05-29",
    "method": "GET",
    "params": {
        "headers": {
            "Content-Type": "application/json"
        }
    },
    "resourcePath": $util.toJson("/v1/users/${ctx.args.id}")
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
## 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
```

- 選擇 Query (查詢) 標籤，然後執行以下查詢：

```
query GetUser{
    getUser(id:1){
        id
        username
    }
}
```

這應該會傳回以下回應：

```
{
```

```
    "data": {
        "getUser": {
            "id": "1",
            "username": "nadia"
        }
    }
}
```

- 選擇 Schema (結構描述) 標籤。
- 在「變異」下方右側的「資料類型」窗格中，找到 AddU ser 欄位，然後選擇「連接」。
- 在 Data source name (資料來源名稱) 中，選擇 HTTP。
- 將以下內容貼到 Configure the request mapping template (設定請求映射範本) 區段：

```
{
    "version": "2018-05-29",
    "method": "POST",
    "resourcePath": "/v1/users",
    "params":{
      "headers":{
        "Content-Type": "application/json",
      },
      "body": $util.toJson($ctx.args.userInput)
    }
}
```

- 將以下內容貼到 Configure the response mapping template (設定回應映射範本) 區段：

```
## 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
```

- 選擇 Query (查詢) 標籤，然後執行以下查詢：

```
mutation addUser{
    addUser(userInput:{
        id:"2",
        username:"shaggy"
    }){
        id
        username
    }
}
```

這應該會傳回以下回應：

```
{
    "data": {
        "getUser": {
        "id": "2",
        "username": "shaggy"
        }
    }
}
```

叫用AWS服務

您可以使用 HTTP 解析器為服務設定 GraphQL API 介面。AWSHTTP 請求AWS必須使用簽名版本 4 進程進行簽名，AWS以便識別發送它們的人員。AWS AppSync 當您將 IAM 角色與 HTTP 資料來源建立關聯時，會代表您計算簽名。

您可以提供兩個額外的元件來叫用 HTTP 解析器的AWS服務：

- 具有呼叫AWS服務 API 之權限的 IAM 角色
- 資料來源中的簽署組態

例如，如果您想要使用 HTTP 解析器呼叫ListGraphqlApis 作業，首先建立AWS AppSync 假設具有下列政策的 IAM 角色：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Action": [
                "appsync:ListGraphqlApis"
            ],
            "Effect": "Allow",
            "Resource": "*"
        }
    ]
}
```

接下來，建立 AWS AppSync 的 HTTP 資料來源。在此範例中，您會在美國西部 (奧勒岡) 區域呼叫 AWS AppSync 。在名為 `http.json` 的檔案中設定以下 HTTP 組態，包含簽署區域和服務名稱：

```
{
    "endpoint": "https://appsync.us-west-2.amazonaws.com/",
    "authorizationConfig": {
        "authorizationType": "AWS_IAM",
        "awsIamConfig": {
            "signingRegion": "us-west-2",
            "signingServiceName": "appsync"
        }
    }
}
```

然後，使用建AWS CLI立具有關聯角色的資料來源，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>
```

當您將解析程式連接到結構描述中的欄位時，請使用以下請求映射範本呼叫 AWS AppSync：

```
{
    "version": "2018-05-29",
    "method": "GET",
    "resourcePath": "/v1/apis"
```

```
}
```

當您針對此資料來源執行 GraphQL 查詢時，AWS AppSync 會使用您提供的角色簽署請求，並在請求中包含簽章。查詢會傳回您在該AWS區域中帳戶中的 AWS AppSync GraphQL API 清單。

教學課程：Aurora 無伺服器

AWS AppSync提供資料來源，以針對已透過資料 API 啟用的 Amazon Aurora 無伺服器叢集執行 SQL 命令。您可以使用 AppSync 解析器針對具有 GraphQL 查詢、突變和訂閱的資料 API 執行 SQL 陳述式。

建立叢集

將 RDS 資料來源新增至之前， AppSync 您必須先在 Aurora 無伺服器叢集上啟用資料 API，並使AWS Secrets Manager用. 您可以先使AWS CLI用下列方式建立 Aurora 無伺服器叢集：

```
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
```

這會傳回叢集的 ARN。

通過AWS Secrets Manager控制台或也通過 CLI 創建一個密鑰與輸入文件，如使用用戶名和 COMPIC_PASSWORD 從上一步如下：

```
{
    "username": "USERNAME",
    "password": "COMPLEX_PASSWORD"
}
```

將此作為參數傳遞給AWS CLI：

```
aws secretsmanager create-secret --name HttpRDSSecret --secret-string file://creds.json
 --region us-east-1
```

這會傳回秘密的 ARN。

請注意 Aurora 無伺服器叢集的 ARN 和密碼，以便稍後在建立資料來源時在 AppSync 主控台中使用。

啟用 Data API

您可以依照 RDS 文件中的說明，在您的叢集上啟用 Data API。在新增為資料來源之前，必須先啟用 AppSync 資料 API。

建立資料庫及資料表

啟用資料 API 之後，您可以確保它可以使用AWS CLI. `aws rds-data execute-statement` 這樣可確保您的 Aurora 無伺服器叢集在將叢集新增至 AppSync API 之前已正確設定。首先使用像這樣的參數創建一個名為 TESTDB 的`--sql`數據庫：

```
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"
```

如果這次執行沒有錯誤，則搭配 create table 命令新增資料表：

```
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"
```

如果一切都沒有問題地運行，則可以繼續將叢集作為 AppSync API 中的數據源添加。

GraphQL 模式

現在您的 Aurora Serverless Data API 已經啟動並搭配資料表運作，我們會建立 GraphQL 結構描述，並連接可執行變動和訂閱的解析程式。在AWS AppSync主控台中建立新的 API 並導覽至「結構描述」頁面，然後輸入下列內容：

```
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
}
```

儲存您的結構描述並瀏覽到 Data Sources (資料來源) 頁面，並建立新的資料來源。選取資料來源類型為 Relational database (關聯式資料庫)，並提供易記名稱。使用您在上一個步驟中所建立的資料庫名

稱，以及您所建立叢集的叢集 ARN。對於角色，您可以 AppSync 創建一個新角色，也可以使用類似下面的策略創建角色：

```
{
    "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:*"
            ]
        }
    ]
}
```

請注意，在您授予角色存取權的這個政策中有 2 個陳述式 。第一個資源是您的 Aurora 無伺服器叢集，第二個資源是您的 AWS Secrets Manager ARN。按一下「建立」之前，您需要在 AppSync 資料來源組態中提供兩個 ARN。

設定解析程式

現在，我們已備妥有效的 GraphQL 結構描述和 RDS 資料來源，我們可以將解析程式連接到結構描述上的 GraphQL 欄位。我們的 API 提供下列功能：

1. 透過 Mutation.createPet 欄位來建立 pet
2. 透過 Mutation.updatePet 欄位來更新 pet
3. 透過 Mutation.deletePet 欄位來建立刪除 pet
4. 透過 Query.getPet 欄位來取得單一 pet
5. 透過 Query.listPets 欄位列出所有 pet
6. 通過查詢列出價格範圍內的寵物。 listPetsByPriceRange字段

Mutation.createPet

在AWS AppSync 主控台的資料架構編輯器中，選擇右側的「貼附解析器」。`createPet(input: CreatePetInput!): Pet`選擇 RDS 資料來源。在 request mapping template (要求映射範本) 區段中，新增下列範本：

```
#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 陳述式會根據陳述式陣列中的順序執行。結果將以相同順序傳回。由於這是一個突變，我們在插入後運行 select 語句來檢索提交的值，以填充 GraphQL 響應映射模板。

在 response mapping template (回應映射範本) 區段中，新增下列範本：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

由於陳述式有兩個 SQL 查詢，所以我們需要使用下列程式碼，指定自資料庫傳回矩陣中的第二個結果：`$utils.rds.toJsonString($ctx.result))[1][0]`)。

Mutation.updatePet

在AWS AppSync 主控台的資料架構編輯器中，選擇右側的「貼附解析器」。updatePet(input: UpdatePetInput!): Pet選擇 RDS 資料來源。在 request mapping template (要求映射範本) 區段中，新增下列範本：

```
{
"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 (回應映射範本) 區段中，新增下列範本：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[1][0])
```

Mutation.deletePet

在AWS AppSync 主控台的資料架構編輯器中，選擇右側的「貼附解析器」。deletePet(input: DeletePetInput!): Pet選擇 RDS 資料來源。在 request mapping template (要求映射範本) 區段中，新增下列範本：

```
{
"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 (回應映射範本) 區段中，新增下列範本：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.getPet

現在，您的結構描述的變動已經建立完成，接著我們要連結這三個查詢，展示如何取得個別、清單，以及套用 SQL 篩選。在AWS AppSync 主控台的資料架構編輯器中，選擇右側的「貼附解析器」。`getPet(id: ID!): Pet`選擇 RDS 資料來源。在 request mapping template (要求映射範本) 區段中，新增下列範本：

```
{
"version": "2018-05-29",
        "statements": [
            $util.toJson("select * from Pets WHERE id=:ID")
    ],
    "variableMap": {
        ":ID": "$ctx.args.id"
    }
}
```

在 response mapping template (回應映射範本) 區段中，新增下列範本：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0][0])
```

Query.listPets

在AWS AppSync 主控台的資料架構編輯器中，選擇右側的「貼附解析器」。`getPet(id: ID!): Pet`選擇 RDS 資料來源。在 request mapping template (要求映射範本) 區段中，新增下列範本：

```
{
    "version": "2018-05-29",
    "statements": [
        "select * from Pets"
    ]
}
```

在 response mapping template (回應映射範本) 區段中，新增下列範本：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

查詢。 listPetsByPriceRange

在AWS AppSync 主控台的資料架構編輯器中，選擇右側的「貼附解析器」。`getPet(id: ID!): Pet`選擇 RDS 資料來源。在 request mapping template (要求映射範本) 區段中，新增下列範本：

```
{
    "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 (回應映射範本) 區段中，新增下列範本：

```
$utils.toJson($utils.rds.toJsonObject($ctx.result)[0])
```

執行變動

現在，您已運用 SQL 陳述式設定您的所有解析程式，並已將 GraphQL API 連接到您的 Serverless Aurora Data API，您可以開始執行變動和查詢了。在 AWS AppSync 主控台中，選擇 Queries (查詢) 標籤，然後輸入下列項目來建立 Pet：

```
mutation add {
    createPet(input : { type:fish, price:10.0 }){
        id
        type
        price
    }
}
```

回應應該包含如下的 id、type 和 price：

```
{
  "data": {
    "createPet": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
```

```
      "type": "fish",
      "price": "10.0"
    }
  }
}
```

您可以執行 updatePet 變動，修改此項目：

```
mutation update {
    updatePet(input : {
        id: ID_PLACEHOLDER,
        type:bird,
        price:50.0
    }){
        id
        type
        price
    }
}
```

請注意，我們使用了之前從 createPet 操作返回的 id。這將是您的記錄在解析程式運用 `$util.autoId()` 時的唯一值。您可以用類似的方式刪除記錄：

```
mutation delete {
    deletePet(input : {id:ID_PLACEHOLDER}){
        id
        type
        price
    }
}
```

運用第一個變動，搭配 price 的幾個不同值，建立一些記錄，然後執行一些查詢。

執行查詢

繼續在主控台的 Queries (查詢) 標籤中，使用以下陳述式，列出您所建立的所有記錄：

```
query allpets {
    listPets {
        id
        type
        price
```

```
    }
}
```

這很好，但讓我們利用 SQL WHER E 謂詞`where price > :MIN and price < :MAX`在我們的查詢映射模板。 listPetsByPriceRange使用以下 GraphQL 查詢：

```
query petsByPriceRange {
    listPetsByPriceRange(min:1, max:11) {
        id
        type
        price
    }
}
```

您應該只會看到 price 超過 $1 元或不到 $10 元的記錄。最後，您可以執行查詢來擷取個別記錄，如下所示：

```
query onePet {
    getPet(id:ID_PLACEHOLDER){
        id
        type
        price
    }
}
```

輸入清理

我們建議開發人員使`variableMap`用來防止 SQL 插入攻擊。如果不使用變數對應，開發人員將負責清理其 GraphQL 作業的引數。達成這個淨化的一種方法是先在要求映射範本中提供輸入特定驗證步驟，接著再對 Data API 執行 SQL 陳述式。讓我們來看看如何修改 `listPetsByPriceRange` 範例的請求映射範本。您可以執行以下操作，而不再只是依賴使用者輸入：

```
#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)
    }
}
```

在對 Data API 執行解析程式時防堵惡意輸入的另外一種方法，就是同時使用預備的陳述式來搭配預存程序和參數化的輸入。例如，在 `listPets` 的解析程式中，定義下列執行 select 為預先準備陳述式的程序：

```
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
```

使用以下執行 SQL 命令，即可在您的 Aurora Serverless 執行個體中建立此陳述式：

```
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"
```

針對 listPets 產生的解析程式碼變得簡單，因為我們現在只要呼叫預存程序。至少，任何字串輸入都應將單引號逸出。

```
#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("'", "''"))
    }
}
```

逸出字串

單引號表示在 SQL 陳述式中字串常值的開始和結束，例如，`'some string value'`。若要允許在字串中使用具有一或多個單引號字元 (') 的字串值，必須以兩個單引號 ('') 取代每個字串值。例如，如果輸入字串是 `Nadia's dog`，則您會將其逸出為如下的 SQL 陳述式：

```
update Pets set type='Nadia''s dog' WHERE id='1'
```

教學課程：管線解析器

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync 提供了一種通過單元解析器將 GraphQL 字段連接到單個數據源的簡單方法。不過，執行單一操作可能還是不夠。管道解析程式的功能是對資料來源依序執行操作。在您的 API 中建立函數，並將其連接到管道解析程式。每個函數執行結果都會輸送到下一個函數，直到沒有要執行的函數。您現在可以使用管道解析程式，直接在 AWS AppSync 中建構更為複雜的工作流程。在此教學課程中，您會建置一個簡單的圖片檢視應用程式，而使用者可以透過此應用程式來張貼圖片和檢視朋友所張貼的圖片。

一鍵設定

如果您想要在中自動設定 GraphQL 端點，並設定所AWS AppSync 有解析器和必要的AWS資源，您可以使用下列範本：AWS CloudFormation

這個堆疊會在您的帳戶中建立下列資源：

- AWS AppSync 的 IAM 角色，可存取您帳戶中的資源
- 2 個 DynamoDB 資料表
- 1 個 Amazon Cognito 使用者工具
- 2 個 Amazon Cognito 使用者集區群組
- 3 個 Amazon Cognito 使用者集區使用者
- 1 個 AWS AppSync API

在AWS CloudFormation堆疊建立程序結束時，您會針對建立的三個 Amazon Cognito 使用者各收到一封電子郵件。每封電子郵件都會包含一個臨時密碼，供您用來以 Amazon Cognito 使用者身分登入 AWS AppSync 主控台。儲存這些密碼，以供教學課程其餘內容使用。

手動設定

如果您希望通過AWS AppSync控制台手動執行某個過 step-by-step 程，請按照以下設置過程進行操作。

設定您的非AWS AppSync資源

API 會與兩個 DynamoDB 表格進行通訊：一個儲存圖片的圖片表格，以及儲存使用者之間關係的好友資料表。API 已設定為使用 Amazon Cognito 使用者集區做為驗證類型。下列AWS CloudFormation堆疊會在帳號中設定這些資源。

在AWS CloudFormation堆疊建立程序結束時，您會針對建立的三個 Amazon Cognito 使用者各收到一封電子郵件。每封電子郵件都會包含一個臨時密碼，供您用來以 Amazon Cognito 使用者身分登入 AWS AppSync 主控台。儲存這些密碼，以供教學課程其餘內容使用。

建立 GraphQL API

若要在 AWS AppSync 中建立 GraphQL API：

1. 開啟 AWS AppSync 主控台，然後選擇 Build From Scratch (從頭開始建置)，然後選擇 Start (開始)。
2. 將 API 名稱設定為 `AppSyncTutorial-PicturesViewer`。
3. 選擇 建立 。

AWS AppSync 主控台會使用 API 金鑰身分驗證模式為您建立新 GraphQL API。您可以使用主控台來設定其他 GraphQL API 和對其執行查詢，以進行此教學的其他部分。

設定 GraphQL API

您需要搭配您先前所建立 Amazon Cognito 使用者集區來設定 AWS AppSync API。

1. 選擇 Settings (設定) 標籤。
2. 在 Authorization Type (授權類型) 區段下，選擇 Amazon Cognito User Pool (Amazon Cognito 使用者集區)。
3. 在 [使用者集區組態] 下，選擇 [AWS區域] 的 [US-WEST-2]。
4. 選擇 AppSyncTutorial-UserPool 使用者集區。
5. 選擇拒絕作為預設處理行動。
6. 將AppId 客戶端正則表達式字段留空。
7. 選擇 儲存 。

API 現在已設定為使用 Amazon Cognito 使用者集區做為驗證類型。

設定 DynamoDB 表格的資料來源

建立 DynamoDB 表格之後，在主控台中導覽至 AWS AppSync GraphQL API，然後選擇「資料來源」索引標籤。現在，您將在中AWS AppSync 為您剛才建立的每個 DynamoDB 資料表建立資料來源。

1. 選擇 Data source (資料來源) 標籤。
2. 選擇 New (新建) 來建立新資料來源。
3. 輸入 `PicturesDynamoDBTable` 做為資料來源名稱。
4. 選擇 Amazon DynamoDB table (Amazon DynamoDB 資料表) 做為資料來源類型。
5. 選擇 US-WEST-2 做為區域。
6. 從表格清單中，選擇 AppSyncTutorial-圖片 DynamoDB 表格。
7. 在 [建立或使用現有角色] 區段中，選擇 [現有角色]。

8. 選擇剛從 CloudFormation 範本建立的角色。如果您沒有變更 ResourceNamePrefix，角色的名稱應該是 AppSyncTutorial-Dynamo Brole。
9. 選擇 建立 。

對好友表重複相同的過程，如果您在創建堆棧時沒有更改ResourceNamePrefix參數，DynamoDB 表的名稱應該是 AppSyncTutorial-Friends。 CloudFormation

建立 GraphQL 結構描述

現在，資料來源已連接至您的 DynamoDB 資料表，讓我們建立 GraphQL 結構描述。在 AWS AppSync 主控台的結構描述編輯器中，確認您的結構描述與以下結構描述相符：

```
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!
}
```

選擇 Save Schema (儲存結構描述) 來儲存結構描述。

有些結構描述欄位已標註 @aws_auth 指令。由於 API 預設動作設定設為 DENY，所以 API 會拒絕不是 @aws_auth 指令提及群組成員的所有使用者。如需如何保護 API 的詳細資訊，您可以讀取 Security (安全性) 頁面。在此情況下，只有管理員使用者才能存取「變更. 建立圖片」和「變更. 建立友誼」欄位，而身為「管理員」或「檢視者」群組成員的使用者則可以存取查詢。 getPicturesBy擁有者欄位。其他所有使用者都無存取權。

設定解析程式

現在，您已備妥有效的 GraphQL 結構描述和兩個資料來源，您可以將解析程式連接到結構描述上的 GraphQL 欄位。API 提供下列功能：

- 透過 Mutation.createPicture 欄位來建立圖片
- 透過 Mutation.createFriendship 欄位來建立朋友關係
- 透過 Query.getPicture 欄位來擷取圖片

Mutation.createPicture

在AWS AppSync 主控台的資料架構編輯器中，選擇右側的「貼附解析器」。`createPicture(input: CreatePictureInput!): Picture!`選擇動 DynamoDB PicturesDynamo料庫表資料來源。在 request mapping template (要求映射範本) 區段中，新增下列範本：

```
#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 (回應映射範本) 區段中，新增下列範本：

```
#if($ctx.error)
```

```
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

建立圖片功能完成。您會使用隨機產生的 UUID 做為圖片 ID，並使用 Cognito 使用者名稱做為擁有者，將圖片儲存到 Pictures (圖片) 資料表。

Mutation.createFriendship

在AWS AppSync 主控台的資料架構編輯器中，選擇右側的「貼附解析器」。`createFriendship(id: ID!, target: ID!): Boolean`選擇動 DynamoDB FriendsDynamo料庫表資料來源。在 request mapping template (要求映射範本) 區段中，新增下列範本：

```
#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)
    }
}
```

重要事項：在BatchPutItem請求範本中，DynamoDB 表格的確切名稱應該會出現。預設的資料表名稱為 AppSyncTutorial-好友。如果您使用錯誤的資料表名稱，則會在 AppSync 嘗試假設提供的角色時收到錯誤。

為了簡化本自學課程，請像已核准友誼請求一樣繼續操作，並將關係項目直接儲存到AppSyncTutorialFriends表格中。

很快地，您就能為每個朋友關係儲存兩個項目，做為雙向關係。如需代表 many-to-many 關係之 Amazon DynamoDB 最佳實務的詳細資訊，請參閱 DynamoDB 最佳實務。

在 response mapping template (回應映射範本) 區段中，新增下列範本：

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end
true
```

注意：請確定您的要求範本包含適當的資料表名稱。預設名稱為 AppSyncTutorial-Friends，但如果您變更 CloudFormation ResourceNamePrefix參數，您的表格名稱可能會有所不同。

查詢。 getPicturesBy擁有者

現在您已經備妥朋友關係和圖片，您還需要提供讓使用者檢視其朋友圖片的權限。為了滿足這項要求，您必須先檢查申請者是內容擁有者的朋友，最後提供圖片查詢功能。

由於此功能需要兩個資料來源操作，所以您要建立兩個函數。第一種函數 isFriend 會檢查申請者和擁有者是否為朋友。第二個功能，getPicturesBy所有者，檢索給定所有者 ID 請求的圖片。讓我們來看看下面的查詢提議的解析器的執行流程。 getPicturesBy擁有者欄位：

1. Before 映射範本：準備內容和欄位輸入引數。
2. isFriend 函數：檢查申請者是否為圖片的擁有者。如果沒有，它會透過在好友資料表上執 GetItem 行 DynamoDB 作業來檢查要求者和擁有者使用者是否為好友。
3. getPicturesBy擁有者函數：使用擁有者索引全域次要索引上的 DynamoDB 查詢操作，從「圖片」表擷取圖片。
4. After 映射範本：映射圖片結果，讓 DynamoDB 屬性正確映射到預期的 GraphQL 類型欄位。

首先，讓我們來建立下列函數。

isFriend 函數

1. 選擇 Functions (函數) 索引標籤。
2. 選擇 Create Function (建立函數) 來建立函數。
3. 輸入 `FriendsDynamoDBTable` 做為資料來源名稱。
4. 對於函數名稱，輸入 isFriend。
5. 在要求映射範本文字區域中，貼上下列範本：

```
#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. 在回應映射範本文字區域中，貼上下列範本：

```
#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. 選擇 Create Function (建立函數)。

結果：您已建立 isFriend 函數。

getPicturesBy擁有者函數

1. 選擇 Functions (函數) 索引標籤。
2. 選擇 Create Function (建立函數) 來建立函數。
3. 輸入 `PicturesDynamoDBTable` 做為資料來源名稱。
4. 對於函數名稱，輸入 `getPicturesByOwner`。
5. 在要求映射範本文字區域中，貼上下列範本：

```
{
    "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. 在回應映射範本文字區域中，貼上下列範本：

```
#if($ctx.error)
    $util.error($ctx.error.message, $ctx.error.type)
#end

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

7. 選擇 Create Function (建立函數)。

結果：您已建立getPicturesBy擁有者函數。現在函數已經創建，請將管線解析器附加到 Query。getPicturesBy擁有者欄位。

在AWS AppSync 主控台的資料架構編輯器中，選擇右側的「貼附解析器」。`Query.getPicturesByOwner(id: ID!): [Picture]`在接下來的頁面，選擇會出現在資料來源下拉式清單中的 Convert to pipeline resolver (轉換成管道解析程式) 連結。使用下列程序來使用 Before 映射範本：

```
#set($result = { "owner": $ctx.args.id, "callerId": $ctx.identity.username })
$util.toJson($result)
```

在 after mapping template (After 映射範本) 區段中，使用下列範本：

```
#foreach($picture in $ctx.result.items)
    ## prepend "src://" to picture.src property
    #set($picture['src'] = "src://${picture['src']}")
#end
$util.toJson($ctx.result.items)
```

選擇 Create Resolver (建立解析程式)。您已成功連接第一個管道解析程式。在相同頁面上，新增您之前建立的兩個函數。在函數區段中，選擇 Add A Function (新增函數)，然後選擇或輸入第一個函數的名稱，isFriend 依照 getPicturesByOwner 函數的相同程序來新增第二個函數。確保 iSriend 功能首先出現在列表中，然後是所getPicturesBy有者功能。您可以使用上下箭頭，重新安排管道中的函數執行順序。

現在，管道解析程式已經建立完成，而且您已連接函數，讓我們來測試新建立的 GraphQL API。

測試您的 GraphQL API

首先，您必須透過您先前建立的管理員使用者，執行少量變動來填入圖片和朋友關係。在 AWS AppSync 主控台的左側，選擇 Queries (查詢) 標籤。

createPicture Mutation

1. 在 AWS AppSync 主控台中，選擇 Queries (查詢) 標籤。
2. 選擇 Login With User Pools (登入使用者集區)。
3. 在強制回應中，輸入 CloudFormation 堆疊所建立的 Cognito 範例用戶端識別碼，例如：
4. 輸入您作為參數傳遞給 CloudFormation 堆疊的使用者名稱。預設為 nadia。
5. 使用發送到您提供的電子郵件作為參數到 CloudFormation 堆棧的臨時密碼（例如，UserPoolUserEmail）。
6. 選擇 Login (登入)。您現在應該會看到重新命名為登出 nadia 的按鈕，或者您在建立 CloudFormation 堆疊時選擇的任何使用者名稱 (也就是說)。UserPoolUsername

讓我們傳送幾個 createPicture 變動來填入該圖片資料表。在主控台中，執行以下 GraphQL 查詢：

```
mutation {
  createPicture(input:{
    owner: "nadia"
    src: "nadia.jpg"
  }) {
    id
```

```
    owner
    src
  }
}
```

回應看起來應如下所示：

```
{
  "data": {
    "createPicture": {
      "id": "c6fedbbe-57ad-4da3-860a-ffe8d039882a",
      "owner": "nadia",
      "src": "nadia.jpg"
    }
  }
}
```

讓我們新增更多圖片：

```
mutation {
  createPicture(input:{
    owner: "shaggy"
    src: "shaggy.jpg"
  }) {
    id
    owner
    src
  }
}
```

```
mutation {
  createPicture(input:{
    owner: "rex"
    src: "rex.jpg"
  }) {
    id
    owner
    src
  }
}
```

您已透過管理使用者的身分，使用 nadia 新增三張圖片。

createFriendship 變動

讓我們來新增朋友關係項目。在主控台中，執行下列變動。

注意：您仍必須登入管理使用者 (預設管理使用者是 nadia)。

```
mutation {
  createFriendship(id: "nadia", target: "shaggy")
}
```

回應看起來應該會像這樣：

```
{
  "data": {
    "createFriendship": true
  }
}
```

nadia 和 shaggy 是朋友。rex 不是任何人的朋友。

getPicturesBy擁有者查詢

第一步是使用 Cognito 使用者集區，以及本教學課程一開始所設定的登入資料，登入成為 nadia 使用者。登入為 nadia 時，擷取 shaggy 擁有的圖片。

```
query {
    getPicturesByOwner(id: "shaggy") {
        id
        owner
        src
    }
}
```

由於 nadia 和 shaggy 是朋友，所以查詢應該會傳回相對應的圖片。

```
{
  "data": {
    "getPicturesByOwner": [
      {
        "id": "05a16fba-cc29-41ee-a8d5-4e791f4f1079",
        "owner": "shaggy",
```

```
        "src": "src://shaggy.jpg"
      }
    ]
  }
}
```

同樣地，嘗試擷取自己圖片的 nadia 也能順利擷取。管道解析器已經過優化，以避免在這種情況下運行 isFriend GetItem 操作。嘗試下列查詢：

```
query {
    getPicturesByOwner(id: "nadia") {
        id
        owner
        src
    }
}
```

如果您啟用登入您的 API (在 Settings 窗格)，則當偵錯層級設定為 ALL (全部)，並再次執行相同查詢時，就會傳回欄位執行的日誌。通過查看日誌，您可以確定 isFriend 函數是否在請求映射模板階段提前返回：

```
{
  "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
}
```

索earlyReturnedValue引鍵代表 #return 指示詞傳回的資料。

最後，即使雷克斯是觀眾 Cognito UserPool 集團的成員，因為雷克斯不是任何人的朋友，他將無法訪問任何毛茸茸或 nadia 擁有的圖片。如果您以 rex 登入主控台，並執行下列查詢時：

```
query {
    getPicturesByOwner(id: "nadia") {
        id
        owner
        src
    }
}
```

您會收到下列關於未經授權的錯誤：

```
{
  "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"
```

```
    }
  ]
}
```

這表示您已成功使用管道解析程式，實作複雜的授權程序。

教學課程：增量同步

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync 中的用戶端應用程式會使用行動/Web 應用程式，將本機快取的 GraphQL 回應儲存到磁碟。已建立版本的資料來源和 Sync 操作可讓客戶使用單一解析程式來執行同步處理程序。透過這些查詢，用戶端可以使用從可能包含大量記錄的一個 base 查詢結果補充其本機快取，然後僅接收從上次查詢後有所更動的資料 (delta 更新)。藉由允許用戶端透過初始請求和其他請求的遞增性更新來執行快取的基本填入，所以您可將用戶端應用程式的計算工作移到後端。對於經常在上線和離線狀態之間切換的用戶端應用程式而言，這種做法可讓效率大幅提升。

若要實作 Delta Sync，Sync 查詢會在已建立版本的資料來源上使用 Sync 操作。當 AWS AppSync 變動變更已建立版本資料來源中的項目時，該變更的記錄也將存放在 Delta 資料表中。您可以選擇為其他版本化資料來源使用不同的 Delta 表格 (例如，每個類型一個，每個網域區域一個)，或針對 API 使用單一 Delta 表格。AWS AppSync 建議不要對多個 API 使用單個 Delta 表，以避免主鍵的衝突。

此外，Delta Sync 用戶端也可以透過引數接收訂閱，然後用戶端協調訂閱和重新連線，並在離線和線上轉換之間進行寫入。Delta Sync 會自動恢復訂閱來執行這個程序，包括指數退避、透過不同的網路錯誤案例進行抖動，以及將事件存放在佇列中。接著會先執行適當的 delta 或 base 查詢，再合併佇列中的任何事件，最後依照一般方式處理訂閱。

用戶端設定選項 (包括 Amplify DataStore) 的說明文件可在 A mplify 架構網站上取得。這份文件概述如何設定已建立版本的 DynamoDB 資料來源和 Sync 操作，以便與 Delta Sync 用戶端搭配使用，提供最佳的資料存取權。

一鍵設定

若要在中自動設定 GraphQL 端點，並設定所AWS AppSync 有解析器和必要的AWS資源，請使用以下範本：AWS CloudFormation

這個堆疊會在您的帳戶中建立下列資源：

- 2 個 DynamoDB 表格 (基本和差異)
- 1 個 AWS AppSync API，含 API 金鑰
- 1 具有動態資料表政策的身分與存取權管理角色

兩個資料表會用來將同步查詢分割成第二個資料表，並做為用戶端離線時遺漏事件的日誌。為了讓 delta 資料表的查詢維持效率，您可以視需要使用 Amazon DynamoDB TTLs 來自動備妥這些事件。TTL 時間可根據您對資料來源的需求進行設定 (您可能希望將其設為 1 小時、1天等)。

結構描述

為了示範差異同步，範例應用程式會在 DynamoDB 中建立由基礎和差異資料表支援的 Post 結構描述。AWS AppSync 自動將突變寫入兩個表中。同步查詢會依適當情況從 Base 或 Delta 資料表提取記錄，並且定義一份訂閱，說明用戶端如何透過其重新連線邏輯來發揮變動功能。

```
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
}
```

這個 GraphQL 結構描述雖然是標準程序，但在繼續之前，有幾個事項值得我們提出說明。首先，所有變動會先自動寫入 Base 資料表，然後寫入 Delta 資料表。Base 資料表是狀態實況的中心來源，而 Delta 資料表是日誌記錄。如果您未傳入*lastSync: AWSTimestamp*，則*syncPosts*查詢會針對 Base 資料表執行，並加入快取，並在定期時執行，做為邊緣案例的全域追趕程序，當用戶端離線時間長於 Delta 資料表中設定的 TTL 時間時。如果您傳入 `lastSync: AWSTimestamp`，`syncPosts` 查詢會執行處理您的 Delta 資料表，並由用戶端用來擷取自其上次離線後有所變更的事件。Amplify 用戶端會自動傳遞 `lastSync: AWSTimestamp` 值，並依適當情況儲存到磁碟中。

在發布刪除字段用於刪除操作。當用戶端離線且記錄已從 Base 資料表移除時，此屬性會通知用戶端執行同步處理，將記錄自其本機快取中移除。如果用戶端離線時間較長，而且在用戶端可以配合 Delta Sync 查詢來擷取這個值之前，該項目已先移除，則 Base 查詢 (可在用戶端中設定) 的全域更新事件將會執行，並從快取中移除該項目。此欄位會標記為選用項目，因為只有在執行同步查詢而結果出現已刪除的項目時，才會傳回值。

變動

對於所有變動，AWS AppSync 會在 Base 資料表中執行標準的建立/更新/刪除操作，並自動記錄 Delta 資料表中的變更。您可以藉由在資料來源中修改 `DeltaSyncTableTTL` 值，以縮短或延長記錄保留時間。對於擁有高速資料的組織，應該會考慮縮短這段時間。或者，如果您的用戶端離線時間較長，最好設定較長的資料保留時間。

同步查詢

基本查詢是未指`lastSync`定值的 DynamoDB 同步作業。這種做法適合許多組織，因為 Base 查詢只會在一開始時執行，且後續是定期執行。

差異查詢是具有指`lastSync`定值的 DynamoDB 同步作業。每當離線用戶端恢復上線，就會執行 delta 查詢 (只要 base 查詢定期時間未觸發執行)。用戶端會自動追蹤上次成功執行查詢以同步資料的時間。

執行 delta 查詢時，查詢的解析程式會使用 `ds_pk` 和 `ds_sk` 僅查詢自上次用戶端執行同步以來變更的記錄。用戶端會存放適當的 GraphQL 回應。

如需有關執行同步查詢的詳細資訊，請參閱同步操作文件。

範例

首先，藉由呼叫 createPost 變動來建立一個項目：

```
mutation create {
  createPost(input: {author: "Nadia", title: "My First Post", content: "Hello World"})
 {
    id
    author
    title
    content
    _version
    _lastChangedAt
    _deleted
  }
}
```

此變動的傳回值將如下所示：

```
{
  "data": {
    "createPost": {
      "id": "81d36bbb-1579-4efe-92b8-2e3f679f628b",
      "author": "Nadia",
      "title": "My First Post",
      "content": "Hello World",
      "_version": 1,
      "_lastChangedAt": 1574469356331,
      "_deleted": null
    }
  }
}
```

如果您檢查 Base 資料表的內容，您會看到一筆記錄，如下所示：

```
{
  "_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"
  }
}
```

如果您檢查 Delta 資料表的內容，您將會看見一筆記錄，如下所示：

```
{
  "_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"
  }
}
```

現在，我們可以模擬 Base 查詢，客戶端將使用以下 syncPosts 查詢來執行該查詢以填入其本機資料存放區：

```
query baseQuery {
  syncPosts(limit: 100, lastSync: null, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
      _lastChangedAt
    }
    startedAt
    nextToken
  }
}
```

此 Base 查詢的傳回值將如下所示：

```
{
  "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
    }
  }
```

```
}
```

稍後，我們將儲存 startedAt 值，以模擬 Delta 查詢，但需要先對資料表進行變更。讓我們使用 updatePost 變動以修改現有的 Post：

```
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
  }
}
```

此變動的傳回值將如下所示：

```
{
  "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
    }
  }
}
```

如果您現在檢查 Base 資料表的內容，您應該會看見更新的項目：

```
{
  "_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"
  }
}
```

如果您現在檢查 Delta 資料表的內容，您應該會看到兩筆記錄：

1. 建立項目時的記錄
2. 更新項目時的記錄。

新項目如下所示：

```
{
  "_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"
  }
}
```

現在，我們可以模擬 Delta 查詢，以擷取用戶端離線時發生的修改。我們將使用從我們的 Base 查詢返回的`startedAt`值來提出請求：

```
query delta {
  syncPosts(limit: 100, lastSync: 1574469602238, nextToken: null) {
    items {
      id
      author
      title
      content
      _version
    }
    startedAt
    nextToken
  }
}
```

此 Delta 查詢的傳回值將如下所示：

```
{
  "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
```

```
    }
  }
}
```

配置和設置

AWS AppSync 讓您可以：

- 緩存經常請求但不太可能從請求更改為請求的數據。這可以減少解析器的負載。如需詳細資訊，請參閱 the section called “緩存和壓縮”。
- 版本 GraphQL 對象來處理和避免多個客戶端之間的衝突。如需詳細資訊，請參閱 the section called “衝突偵測與同步”。
- 使用自訂網域名稱來設定適用於 GraphQL 和即時 API 的單一易記網域。如需詳細資訊，請參閱設定自訂網域名稱。
- 允許透過虛擬私人 VPC 存取您的 GraphQL API。如需詳細資訊，請參閱使用AWS AppSync私有API。
- 啟用內部檢查，並設定每個查詢的查詢深度和解析器限制。如需詳細資訊，請參閱組態限制。

此外，還AWS AppSync包括下列用於記錄、監視和追蹤的標準AWS工具：

- 登入 AWS CloudTrail
- 使用 Amazon 監控 CloudWatch
- 使用追蹤 AWS X-Ray

緩存和壓縮

AWS AppSync伺服器端的資料快取功能可讓資料以高速記憶體內快取提供，進而改善效能並減少延遲。這樣可減少直接存取資料來源的需求。單元和管線解析器都可以使用緩存。

AWS AppSync還允許您壓縮 API 響應，以便更快地加載和下載承載內容。這可能會減少應用程式的壓力，同時也可能降低資料傳輸費用。壓縮行為是可配置的，並且可以自行決定進行設置。

如需定義 AWS AppSync API 中所需伺服器端快取和壓縮行為的說明，請參閱本節。

執行個體類型

AWS AppSync在與您的 AWS AppSync API 相同的AWS帳戶和AWS區域中託管 ElastiCache 適用於 Redis 執行個體的亞馬遜。

您可以使 ElastiCache 用下列 Redis 執行個體類型：

小型

1 個 vCPU、1.5 GiB 記憶體、低至中等網路效能

中型

2 個 vCPU、3 GiB 記憶體、低至中等網路效能

大型

2 個 vCPU、12.3 GiB 記憶體、高達 10 千兆網路效能

xlarge

4 個 vCPU、25.05 GiB 記憶體、高達 10 千兆網路效能

2xlarge

8 個 vCPU、50.47 GiB 記憶體、高達 10 千兆網路效能

4xlarge

16 個 vCPU、101.38 GiB 記憶體、高達 10 千兆網路效能

8xlarge

32 個 vCPU、203.26 GiB 記憶體、10 千兆位元網路效能 (並非所有區域皆提供)

12xlarge

48 個 vCPU、317.77 GiB 博記憶體、10 千兆網路效能

> **Note**
>
> 從歷史上看，您指定了特定的例證類型（例如`t2.medium`）。從 2020 年 7 月起，這些舊版執行個體類型仍然可用，但不建議使用它們。建議您使用此處所述的泛型執行個體類型。

快取行為

以下是與緩存相關的行為：

無

沒有伺服器端快取。

完整要求快取

如果資料不在快取中，則會從資料來源擷取資料並填入快取，直到存留時間 (TTL) 到期為止。對 API 的所有後續請求都會從緩存中返回。這表示除非 TTL 過期，否則不會直接連絡資料來源。在此設定中，我們使用`context.arguments`和 `context.identity` map 的內容做為快取金鑰。

每個解析程式快取

使用此設定時，必須明確選擇每個解析器，才能快取回應。您可以在解析器上指定 TTL 和快取金鑰。您可以指定的快取金鑰是這些對應中的頂層對應`context.arguments`、`context.source`、`context.identity`和/或字串欄位。TTL 值為必填，但快取索引鍵是可選的。如果您未指定任何快取金鑰，預設值為`context.arguments`、`context.source`和對`context.identity`應的內容。

例如，您可以使用下列組合：

- 上下文. 參數和上下文. 源
- 上下文. 參數和上下文. 身份
- 上下文. 參數 .id 或上下文. 參數。 InputType.id
- 上下文. 源. ID 和上下文. 身份.
- 上下文. 識別. 宣告. 使用者名稱

當您僅指定 TTL 且沒有快取金鑰時，解析程式的行為與完整要求快取相同。

快取存留時間

此設定定義在記憶體中儲存快取項目的時間量。TTL 的最大值為 3,600 秒 (1 小時)，之後會自動刪除項目。

快取加密

緩存加密有以下兩種方式。這些設定與 Redis 允許 ElastiCache 的設定類似。只有在第一次啟用 AWS AppSync API 的快取時，您才能啟用加密設定。

- 傳輸中加密 — 快取和資料來源之間AWS AppSync的要求 (不安全的 HTTP 資料來源除外) 會在網路層級加密。由於需要一些處理來加密和解密端點上的資料，因此傳輸中加密可能會影響效能。
- 靜態加密 — 在交換作業期間從記憶體儲存到磁碟的資料會在快取執行個體加密。此設定也會影響效能。

若要使快取項目無效，您可以使用AWS AppSync 主控台或 AWS Command Line Interface (AWS CLI) 進行清除快取 API 呼叫。

有關更多信息，請參閱 AWS AppSync API 參考中的ApiCache數據類型。

緩存驅逐

當您設AWS AppSync定伺服器端快取時，您可以設定最大 TTL。此值定義快取項目儲存在記憶體中的時間量。在必須從緩存中刪除特定條目的情況下，您可以在解析器AWS AppSync的請求或響應中使用的evictFromApiCache擴展實用程序。例如，當資料來源中的資料發生變更，而快取項目現在已過時。) 要從緩存中驅逐項目，您必須知道其密鑰。因此，如果您必須動態收回項目，我們建議您使用每個解析器快取，並明確定義要用來將項目新增至快取的金鑰。

收回快取項目

若要從快取中收回項目，請使用evictFromApiCache擴充功能公用程式。指定類型名稱和欄位名稱，然後提供索引鍵值項目的物件，以建立要收回之項目的索引鍵。在物件中，每個索引鍵都代表快取解析器清單中使用之context物件的cachingKey有效項目。每個值都是用來建構索引鍵值的實際值。您必須按照與緩存解析器列表中的緩存鍵相同的順序放置對象中的cachingKey項目。

例如，請參閱下列結構描述：

```
type Note {
  id: ID!
  title: String
  content: String!
}

type Query {
  getNote(id: ID!): Note
}

type Mutation {
  updateNote(id: ID!, content: String!): Note
}
```

在此範例中，您可以啟用每個解析器快取，然後將其啟用於查詢。getNote然後，您可以配置要包含的緩存密鑰[context.arguments.id]。

當您嘗試獲取時Note，要構建緩存密鑰，請使用查詢的id參數在其服務器端緩存中AWS AppSync 執行getNote查找。

當您更新時Note，您必須收回特定附註的項目，以確保下一個要求會從後端資料來源擷取該項目。若要這麼做，您必須建立要求處理常式。

下面的例子顯示了使用此方法處理驅逐的一種方法：

```
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，您也可以在回應處理常式中處理逐出。

處理updateNote突變時，AWS AppSync 嘗試驅逐條目。如果成功清除項目，回應會在extensions物件中包含一個apiCacheEntriesDeleted值，顯示已刪除的項目數目：

```
"extensions": {  "apiCacheEntriesDeleted": 1}
```

根據身份收回快取項目

您可以根據context物件中的多個值建立快取金鑰。

例如，採用下列使用 Amazon Cognito 使用者集區做為預設驗證模式的結構描述，並由 Amazon DynamoDB 資料來源提供支援：

```
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物件類型會儲存在 DynamoDB 表格中。此資料表有一個複合索引鍵，該索引鍵使用 Amazon Cognito 使用者名稱做為主索引鍵，並使用的 id (Slug) Note 做為分割區索引鍵。這是一個多租戶系統，允許多個用戶託管和更新他們的私有Note對象，這些對象永遠不會共享。

由於這是一個大量讀取的系統，因此getNote查詢會使用每個解析器快取來快取，快取金鑰由. [context.identity.username, context.arguments.id] 更新時Note，您可以收回該特定Note項目的項目。您必須按照在解析器列表中指定的順序在對象中添加組件。cachingKeys

下面的例子顯示了這一點：

```
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;
```

後端系統也可以更新Note和驅逐項目。例如，採取這種突變：

```
type Mutation {
  updateNoteFromBackend(id: ID!, content: String!, username: ID!): Note @aws_iam
}
```

您可以收回項目，但是將快取金鑰的元件新增至cachingKeys物件。

在下面的例子中，驅逐發生在解析器的響應：

```
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;
```

如果您的後端資料已在以外更新 AWS AppSync，您可以呼叫使用資NONE料來源的突變，從快取中收回項目。

壓縮 API 回應

AWS AppSync 允許用戶端要求壓縮承載。如果要求，API 回應會被壓縮並傳回，以回應要求指出偏好的壓縮內容。壓縮的 API 回應載入速度更快、下載內容的速度更快，而且您的資料傳輸費用也可能降低。

> **Note**
>
> 壓縮功能適用於 2020 年 6 月 1 日之後建立的所有新 API。
> AWS AppSync 以最大努力的方式壓縮物件。在極少數情況下，AWS AppSync 可能會根據各種因素（包括當前容量）略過壓縮。

AWS AppSync 可以將 GraphQL 查詢裝載大小壓縮在 1,000 到 10 億個位元組之間。若要啟用壓縮，用戶端必須傳送含有值的`Accept-Encoding`標頭`gzip`。壓縮可以通過檢查響應`Content-Encoding`頭的值進行驗證（`gzip`）。

依預設，AWS AppSync 主控台中的查詢總管會自動設定要求中的標頭值。如果您執行的查詢具有足夠大的回應，則可以使用瀏覽器開發人員工具來確認壓縮。

設定自訂網域名稱

同AWS AppSync，您可以使用自訂網域名稱來設定單一易記的網域，該網域同時適用於 GraphQL 和即時 API。

換句話說，您可以通過創建與AWS AppSync您帳戶中的 API。

當您配置AWS AppSyncAPI，會佈建兩個端點：

AWS AppSync圖形 SQL 端點：

```
https://example1234567890000.appsync-api.us-east-1.amazonaws.com/graphql
```

AWS AppSync即時端點：

```
wss://example1234567890000.appsync-realtime-api.us-east-1.amazonaws.com/
graphql
```

使用自訂網域名稱，您可以使用單一網域與兩個端點互動。例如，如果您設定`api.example.com`做為您的自訂網域，您可以使用下列 URL 與 GraphQL 和即時端點互動：

AWS AppSync自訂網域圖形 SQL 端點：

```
https://api.example.com/graphql
```

AWS AppSync自訂網域即時端點：

```
wss://api.example.com/graphql/realtime
```

> **Note**
>
> AWS AppSync對於自訂網域名稱，API 僅支援 TLS 1.2 和 TLS 1.3。

註冊和設定網域名稱

若要為您的網域設定自訂網域名稱AWS AppSyncAPI，您必須擁有註冊的互聯網域名稱。您可以使用以下方式註冊互聯網域Amazon Route 53 domain registration或您選擇的第三方域名註冊商。如需 53 號公路的詳細資訊，請參閱什麼是亞馬遜路線 53？在亞馬遜路線 53 開發者指南。

API 的自訂網域名稱可以是已註冊網際網路網域的子網域名稱或根網域名稱 (也稱為「區域頂點」)。在中建立自訂網域名稱之後AWS AppSync，您必須建立或更新 DNS 供應商的資源記錄，才能對應至您的 API 端點。如果沒有此映射，綁定自定義域名的 API 請求將無法到達AWS AppSync。

在中建立自訂網域名稱AWS AppSync

建立自訂網域名稱AWS AppSyncAPI 設置了一個Amazon CloudFront分佈。您必須設定 DNS 記錄，才能將自訂網域名稱對應至CloudFront分發網域名稱。需要此映射才能路由綁定到自定義域名的 API 請求AWS AppSync透過對應CloudFront分佈。您也必須提供自訂網域名稱的憑證。

若要設定自訂網域名稱或更新其憑證，您必須擁有更新權限CloudFront分佈和描述AWS Certificate Manager您計劃使用的 (ACM) 憑證。要授予這些權限，請附上以下內容AWS Identity and Access Management(IAM) 向您帳戶中的 IAM 使用者、群組或角色發出的政策聲明：

```
{
  "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透過在上利用伺服器名稱指示 (SNI) 來支援自訂網域名稱CloudFront分佈。如需有關在CloudFront散佈 (包括必要的憑證格式和最大憑證金鑰長度)，請參閱搭配使用 HTTPSCloudFront在亞馬遜CloudFront開發者指南。

若要將自訂網域名稱設定為 API 的主機名稱，API 擁有者必須為自訂網域名稱提供 SSL/TLS 憑證。若要提供憑證，請執行下列其中一個動作：

- 要求 ACM 中的新憑證，或將協力廠商憑證授權單位核發的憑證匯入 ACMus-east-1 AWS區域 (美國東部 (維吉尼亞北部))。如需 ACM 的更多資訊，請參閱什麼是AWS Certificate Manager?在AWS Certificate Manager使用者指南。
- 提供 IAM 伺服器憑證。如需詳細資訊，請參閱在 IAM 中管理伺服器憑證在IAM 使用者指南。

萬用字元自訂網域名稱AWS AppSync

AWS AppSync支援萬用字元自訂網域名稱。若要設定萬用字元自訂網域名稱，請指定萬用字元 (*) 做為自訂網域的第一個子網域。這代表根網域的所有可能的子網域。例如，萬用字元自訂網域名稱`*.example.com`會產生子網域，例如`a.example.com`,`b.example.com`，以及`c.example.com`。所有這些子網域都會路由到相同的網域。

若要在中使用萬用字元自訂網域名稱AWS AppSync，您必須提供 ACM 核發的憑證，其中包含萬用字元名稱，該憑證可以保護相同網域中的多個網站。如需詳細資訊，請參閱ACM 憑證特性在AWS Certificate Manager使用者指南。

衝突偵測與同步

已建立版本的資料來源

AWS AppSync 目前支援 DynamoDB 資料來源上的版本控制。衝突偵測、衝突解決和同步作業需要 `Versioned` 資料來源。當您在資料來源上啟用版本控制時，AWS AppSync 會自動：

- 使用物件版本控制中繼資料增強項目。
- 對於具有 AWS AppSync 變動之項目所做的變更，將其記錄到 Delta 資料表。
- 針對可設定的時間量，在具有「tombstone」的 Base 資料表中，維護已刪除的項目。

已建立版本的資料來源組態

當您在 DynamoDB 資料來源上啟用版本控制時，請指定以下欄位：

`BaseTableTTL`

在具有「tombstone」的 Base 資料表中，保留已刪除項目的分鐘數 - 指出已刪除項目的中繼資料欄位。如果您想要在刪除項目時立即移除項目，可以將此值設為 0。此欄位為必填。

`DeltaSyncTableName`

資料表名稱，其中存放對於具有 AWS AppSync 變動之項目所做的變更。此欄位為必填。

`DeltaSyncTableTTL`

保留 Delta 資料表中項目的分鐘數。此欄位為必填。

Delta Sync 資料表

AWS AppSync 目前支援使用`PutItem`、`UpdateItem`和`DeleteItem` DynamoDB 作業進行突變的差異同步記錄。

當 AWS AppSync 變動變更已建立版本資料來源中的項目時，該變更的記錄也將存放在針對累加式更新進行最佳化的 Delta 資料表中。您可以選擇為其他版本化資料來源使用不同的 Delta 表格 (例如，每個類型一個，每個網域區域一個)，或針對 API 使用單一 Delta 表格。AWS AppSync 建議不要對多個 API 使用單個 Delta 表，以避免主鍵的衝突。

此資料表所需的結構描述如下所示：

ds_pk

用來做為分割區索引鍵的字串值。它是通過連接基礎數據源名稱和發生更改日期的 ISO 8601 格式（例如Comments:2019-01-01）構建的。

將 VTL 對應範本中的customPartitionKey旗標設定為分區索引鍵的欄名稱時 (請參閱AWS AppSync 開發人員指南中的 DynamoDB 的解析程式對應範本參考)、ds_pk變更格式和字串是透過將分區索引鍵的值附加到基底資料表中的新記錄中來建構。例如，如果基底資料表中的記錄的分區索引鍵值為1a且排序索引鍵值為2b，則字串的新值將為：Comments:2019-01-01:1a。

ds_sk

用來做為排序索引鍵的字串值。它是透過串連發生變更時間的 ISO 8601 格式、項目的主索引鍵以及項目版本來建構的。這些字段的組合保證了 Delta 表中每個條目的唯一性（例如，對於的09:30:00一段時間1a和一個 ID 和版本2，這將是09:30:00:1a:2）。

當 VTL 對應範本中的customPartitionKey旗標設定為分區索引鍵的欄名稱時 (請參閱AWS AppSync 開發人員指南中 DynamoDB 的解析器對應範本參考)、ds_sk變更格式和字串是透過將組合鍵的值取代為基底資料表中排序鍵的值來建構。使用上述範例，如果「基底」資料表中的記錄具有的分割區索引鍵值1a且排序索引鍵值為2b，則字串的新值將是：09:30:00:2b:3。

_ttl

此數值用於存放應該從 Delta 資料表中移除項目時的時間戳記 (以 epoch 秒為單位)。此值是透過將資料來源上設定的 DeltaSyncTableTTL 值加入發生變更的時刻所決定的。此欄位應該設定為 DynamoDB TTL 屬性。

設定為與 Base 資料表搭配使用的 IAM 角色必須也包含在 Delta 資料表上進行操作的許可。在此範例中，會顯示名為的「基底」資料表Comments和名為的 Delta 資料表的權限原ChangeLog則：

```
{
    "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/*"
            ]
        }
    ]
}
```

已建立版本的資料來源中繼資料

AWS AppSync 代表您管理`Versioned`資料來源上的中繼資料欄位。自行修改這些欄位可能會導致應用程式發生錯誤或資料遺失。這些欄位包括：

`_version`

依序遞增的計數器，每當對項目進行變更時都會更新。

`_lastChangedAt`

此數值用於存放上次修改項目時的時間戳記 (以 epoch 毫秒為單位)。

`_deleted`

布林「tombstone」值，表示已刪除項目。應用程式可以使用此值，從本機資料存放區撤出已刪除的項目。

`_ttl`

此數值用於存放應該從基礎資料來源中移除項目時的時間戳記 (以 epoch 秒為單位)。

`ds_pk`

用來做為 Delta 資料表之分割區索引鍵的字串值。

`ds_sk`

用來做為 Delta 資料表之排序索引鍵的字串值。

`gsi_ds_pk`

為支援全域次要索引做為資料分割索引鍵而產生的字串值屬性。只有在 VTL 對應範本中同時啟用`customPartitionKey`和`populateIndexFields`旗標時，才會包含它 (請參閱AWS AppSync 開發人員指南中的 DynamoDB 的解析程式對應範本參考)。如果啟用，將透過連接基礎資料來源名

稱和發生變更日期的 ISO 8601 格式來建構值 (例如，如果「基底」資料表命名為「註解」，則此記錄將設定為`Comments:2019-01-01`)。

`gsi_ds_sk`

為支援全域次要索引做為排序索引鍵而產生的字串值屬性。只有在 VTL 對應範本中同時啟用`customPartitionKey`和`populateIndexFields`旗標時，才會包含它 (請參閱AWS AppSync 開發人員指南中的 DynamoDB 的解析程式對應範本參考)。如果啟用，則會將發生變更的時間 ISO 8601 格式、Base 資料表中項目的分割索引鍵、Base 資料表中項目的排序索引鍵，以及項目的版本 (例如，分割區索引鍵值為的*`09:30:00`*時間*`1a`*、排序索引鍵值*`2b`*，以及此項目的*`3`*版本*`09:30:00:1a#2b:3`*) 來建構。

基礎資料來源中基礎資料來源中這些中繼資料欄位會影響項目的整體大小。AWS AppSync 建議在設計應用程式時，為版本化的資料來源中繼資料保留 500 位元組 + 最大主索引鍵儲存大小。若要在用戶端應用程式中使用此中繼資料，請在 GraphQL 類型和變動選取範圍中包含 `_version`、`_lastChangedAt` 和 `_deleted` 欄位。

衝突偵測與解決方案

當 AWS AppSync 發生並行寫入時，您可以設定「衝突偵測」和「衝突解決」策略，以適當地處理更新。「衝突偵測」會判斷變動是否與資料來源中的實際寫入項目衝突。將`conflictDetection`欄位中的值設定 SyncConfig 為，即可啟用衝突偵測`VERSION`。

「衝突解決」是偵測到衝突時所採取的動作。這是透過在中設定「衝突處理程式」欄位來決定的 SyncConfig。有三種衝突解決策略：

- OPTIMISTIC_CONCURRENCY
- AUTOMERGE
- LAMBDA

每一種衝突解決策略的詳細介紹如下所示。

版本會 AppSync 在寫入作業期間自動遞增，而且不應由用戶端或在已啟用版本功能的資料來源設定的解析程式之外進行修改。這樣做會改變系統的一致性行為，並可能造成資料遺失。

開放式並行存取

開放式並行存取是 AWS AppSync 為已建立版本之資料來源提供的一種衝突解決策略。當衝突解析程式設為「開放式並行存取」時，如果偵測到傳入變動的版本與物件的實際版本不同，衝突處理常式將僅

拒絕傳入請求。在 GraphQL 回應中，將提供最新版本伺服器上的現有項目。然後，預期用戶端在本機處理此衝突，並使用該項目的更新版本重試變動。

自動合併

自動合併提供開發人員一種簡單的方法來設定衝突解決策略，而不需撰寫用戶端邏輯以手動合併其他策略無法處理的衝突。自動合併在合併資料來解決衝突時，遵守嚴格的規則集。自動合併的原則是以 GraphQL 欄位的基礎資料類型為主。如下所示：

- 標量字段上的衝突：GraphQL 標量或任何不是集合的字段（即列表，集合，地圖）。拒絕純量欄位的傳入值，並選取伺服器中的現有值。
- List 的衝突：GraphQL 類型和資料庫類型為 List。串連傳入清單與伺服器中的現有清單。傳入變動的清單值將附加到伺服器列表的末尾。將保留重複的值。
- Set 的衝突：GraphQL 類型為 List，資料庫類型為 Set。使用傳入集合和伺服器中的現有集合套用集合聯集。這符合 Set 的屬性，表示沒有重複的項目。
- 當傳入的變異將新欄位新增至項目或針對具有值的欄位建立時`null`，請將該欄位合併至現有項目。
- Map 的衝突：當資料庫中的基礎資料類型是 Map (即索引鍵-值文件) 時，會套用上述規則，因為其剖析和處理 Map 的每個屬性。

自動合併是為了自動偵測、合併和重試具有更新版本的請求而設計，讓用戶端無需手動合併任何衝突的資料。

為了展現自動合併如何處理純量類型衝突的範例。我們將使用以下記錄做為起點。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 4
}
```

現在，由於用戶端尚未與伺服器同步，傳入變動可能試圖使用舊版本來更新該項目。如下所示：

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 55,
  "_version" : 2
}
```

請注意，傳入請求中的過時版本 2。在此流程中，自動合併拒絕將 ‘jersey’ 欄位更新為 ‘55’ ，並保留值 ‘5’，導致以下項目影像儲存在伺服器中。

```
{
  "id" : 1,
  "name" : "Nadia",
  "jersey" : 5,
  "_version" : 5 # version is incremented every time automerge performs a merge that is
 stored on the server.
}
```

假設如上所示的項目狀態為版本 5，現在假設傳入變動嘗試使用以下影像對項目進行變動：

```
{
  "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
}
```

傳入變動中有三個參考資訊。名稱 (純量) 已變更，但增加兩個新欄位，“interests” (Set) 和 “points” (List)。在這種情況下，因為版本不符，會偵測到衝突。自動合併遵循其屬性，並拒絕名稱變更，因為它是純量，而且是非衝突的欄位。這將導致儲存在伺服器中的項目顯示如下。

```
{
  "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
}
```

現在，使用版本 6 的項目更新影像，現在假設傳入變動 (具有另一個不相符的版本) 嘗試將項目轉換為以下內容：

```
{
  "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
}
```

我們在這裡觀察到 “interests” 的傳入欄位具有伺服器中的一個重複值和兩個新值。在這種情況下，由於基礎資料類型是 Set，所以自動合併會將伺服器中的現有值與傳入請求中的值合併，並刪除任何重複項目。同樣地，“points” 欄位也有衝突，有一個重複值和一個新值。但由於這裡的基礎資料類型是 List，自動合併會簡單地將傳入請求中的所有值附加到伺服器中已存在值的結尾。存放在伺服器上的產生合併影像顯示如下：

```
{
  "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
  "_version" : 7
}
```

現在，假設存放在伺服器中的項目在版本 8 中如下所示。

```
{
  "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
  "stats": {
      "ppg": "35.4",
      "apg": "6.3"
  }
  "_version" : 8
}
```

但傳入的請求嘗試使用以下影像來更新項目，再次發生版本不相符：

```
{
  "id" : 1,
```

```
  "name" : "Nadia",
  "stats": {
      "ppg": "25.7",
      "rpg": "6.9"
  }
  "_version" : 3
}
```

在這種情況下，我們可以看到伺服器中已存在的欄位遺失 (interests、points、jersey)。此外，已編輯映射 “stats” 中的 “ppg” 值，已新增新值 “rpg”，而且已省略 “apg”。自動合併會保留已省略的欄位 (注意：如果要移除欄位，必須使用相符的版本再次嘗試該請求)，以便不會遺失欄位。它也會將相同的規則套用至映射中的欄位，因此將拒絕對 “ppg” 進行的變更，而保留 “apg” ，並新增欄位 “rpg”。存放在伺服器中的產生項目現在顯示如下：

```
{
  "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",
      "rpg": "6.9"
  }
  "_version" : 9
}
```

Lambda

衝突解決選項：

- `RESOLVE`：以回應承載中提供的新項目取代現有項目。您一次只能對單一項目重試相同的操作。目前支援 DynamoDB `PutItem` 和 `UpdateItem`。
- `REJECT`：拒絕變動，並透過 GraphQL 回應中的現有項目傳回錯誤。目前支援 DynamoDB `PutItem`、`UpdateItem` 和 `DeleteItem`。
- `REMOVE`：移除現有的項目。目前支援 DynamoDB `DeleteItem`。

Lambda 叫用請求

AWS AppSync DynamoDB 解析程式會叫用中指定的 Lambda 函數`LambdaConflictHandlerArn`。它會使用資料來源上所設定的相同 `service-role-arn`。叫用承載的結構如下：

```
{
    "newItem": { ... },
    "existingItem": {... },
    "arguments": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

欄位定義如下：

`newItem`

預覽項目，如果變動成功。

`existingItem`

目前位於 DynamoDB 資料表中的項目。

`arguments`

來自 GraphQL 變動的引數。

`resolver`

AWS AppSync 解析程式的相關資訊。

`identity`

發起人的相關資訊。如果使用 API 金鑰存取，則此欄位會設為 Null。

承載範例：

```
{
    "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 叫用回應

適用於 PutItem 和 UpdateItem 衝突解決

RESOLVE 變動。回應必須採用下列格式。

```
{
    "action": "RESOLVE",
    "item": { ... }
}
```

item 欄位代表將用來取代基礎資料來源中現有項目的物件。如果包含在 item 中，則會忽略主索引鍵和同步中繼資料。

REJECT 變動。回應必須採用下列格式。

```
{
    "action": "REJECT"
}
```

適用於 DeleteItem 衝突解決

REMOVE 項目。回應必須採用下列格式。

```
{
    "action": "REMOVE"
}
```

REJECT 變動。回應必須採用下列格式。

```
{
    "action": "REJECT"
}
```

以下 Lambda 函數範例會檢查進行呼叫的對象和解析程式名稱。如果它是由製作REMOVE的jeffTheAdmin，則用於 DeletePost解析器的對象或RESOLVE與更新/放置解析器的新項目衝突。如果不是，則變動為 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;
}
```

錯誤

ConflictUnhandled

衝突偵測發現版本不相符，而且衝突處理常式拒絕變動。

範例：具有開放式並行存取衝突處理常式的衝突解決機制。或者，Lambda 衝突處理常式傳回 REJECT。

ConflictError

嘗試解決衝突時，發生內部錯誤。

範例：Lambda 衝突處理常式傳回格式錯誤的回應。或者，無法叫用 Lambda 衝突處理常式，因為找不到提供的資源 LambdaConflictHandlerArn。

MaxConflicts

已達到衝突解決的重試次數上限。

範例：同一個物件上有太多並行請求。解決衝突之前，另一個用戶端已將物件更新為新版本。

BadRequest

用戶端嘗試更新中繼資料欄位 (_version、_ttl、_lastChangedAt、_deleted)。

範例：用戶端嘗試更新具有更新變動之 _version 的物件。

DeltaSyncWriteError

無法寫入差異同步記錄。

範例：變動成功，但嘗試寫入差異同步資料表時發生內部錯誤。

InternalFailure

發生內部錯誤。

CloudWatch 日誌

如果AWS AppSync API 已啟用記錄 CloudWatch 檔，且將記錄設定設為 [欄位層級記錄檔]，enabled且 [欄位層級記錄檔] 的記錄層級設定為ALL，則AWS AppSync 會將衝突偵測和解決方案資訊發送至記錄群組。如需日誌訊息格式的相關資訊，請參閱衝突偵測與同步記錄的文件。

同步操作

版本化資料來源支Sync援可讓您從 DynamoDB 表擷取所有結果的作業，然後僅接收自上次查詢 (差異更新) 以來變更的資料。當 AWS AppSync 收到 Sync 操作的請求時，會使用請求中指定的欄位來決定是否應該存取 Base 資料表或 Delta 資料表。

- 如果未指定lastSync欄位，則會Scan在「基底」資料表上執行。
- 如果已指定lastSync欄位，但值在之前current moment - DeltaSyncTTL，則會執行「基底」資料表Scan上的 a。
- 如果已指定lastSync欄位，且值在上或之後current moment - DeltaSyncTTL，則會執行 Delta 資料表Query上的 a。

AWS AppSync 將該startedAt字段返回到所有Sync操作的響應映射模板。startedAt 欄位是開始進行 Sync 操作時，可以在本機存放並在另一個請求中使用的時間 (以 epoch 毫秒為單位)。如果請求中包含分頁字符，則該值將與請求針對第一頁結果傳回的值相同。

如需 Sync 映射範本的相關資訊，請參閱映射範本參考。

監控和記錄

若要監控 AWS AppSync GraphQL API 並協助偵錯與請求相關的問題，您可以開啟 Amazon CloudWatch 日誌的記錄功能。

設置和配置

若要在 GraphQL API 上開啟自動記錄功能，請使用 AWS AppSync 主控台。

1. 登入 AWS Management Console 並開啟AppSync主控台。
2. 在 [API] 頁面上，選擇 GraphQL API 的名稱。
3. 在 API 首頁的導覽窗格中，選擇 [設定]。
4. 在 Logging (記錄)，執行以下項目：

a. 開啟「啟用記錄檔」。

b. 如需詳細的要求層級記錄，請選取 [包含詳細內容] 下的核取方塊。 (選擇性)

c. 在 [欄位解析程式記錄層級] 下，選擇您偏好的欄位層級記錄層級 ([無]、[錯誤] 或 [全部])。 (選擇性)

d. 在 [建立或使用現有角色] 底下，選擇 [新增角色] 以建立可 AWS AppSync 將記錄寫入的新 AWS Identity and Access Management (IAM) CloudWatch。或者，選擇現有角色以選取 AWS 帳戶中現有 IAM 角色的 Amazon 資源名稱 (ARN)。

5. 選擇儲存。

手動 IAM 角色設定

如果您選擇使用現有的 IAM 角色，則該角色必須授與 AWS AppSync 寫入記錄的必要權限 CloudWatch。若要手動設定此項目，您必須提供服務角色 ARN， AWS AppSync 以便在寫入記錄檔時可以擔任該角色。

在 IAM 主控台中，建立名稱`AWSAppSyncPushToCloudWatchLogsPolicy`具有下列定義的新政策：

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
        }
    ]
}
```

接下來，使用名稱建立新角色 AWSAppSyncPushToCloudWatchLogsRole，並將新建立的原則附加至該角色。將此角色的信任關係編輯為下列項目：

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

```
        {
        "Effect": "Allow",
        "Principal": {
            "Service": "appsync.amazonaws.com"
        },
        "Action": "sts:AssumeRole"
        }
    ]
}
```

複製角色 ARN 並在設定 AWS AppSync GraphQL API 的記錄時使用它。

CloudWatch 度量

您可以使用 CloudWatch 指標來監控並提供有關可能導致 HTTP 狀態碼或延遲的特定事件的警示。系統會發出下列量度：

量度清單

4XXError

由於用戶端設定不正確而無效的要求所產生的錯誤。通常，這些錯誤發生在 GraphQL 處理之外的任何地方。例如，當要求包含不正確的 JSON 承載或不正確的查詢、服務限制或授權設定錯誤時，就會發生這些錯誤。

單位：計數。使用總和統計以取得這些錯誤的總次數。

5XXError

在執行 GraphQL 查詢期間遇到的錯誤。例如，當呼叫空白或不正確的結構描述的查詢時，就會發生這種情況。當 Amazon Cognito 使用者集區識別碼或 AWS 區域無效時，也可能會發生這種情況。或者，如果在處理請求期間 AWS AppSync 遇到問題，也可能發生這種情況。

單位：計數。使用總和統計以取得這些錯誤的總次數。

Latency

從客戶端接 AWS AppSync 收請求到返回響應到客戶端之間的時間。這不包含回應到達最終裝置所發生的網路延遲。

單位：毫秒。使用平均統計資料以評估預期的延遲。

Requests

您帳戶中所有 API 已處理的請求數量 (查詢 + 突變)，依區域排列。

單位：計數。在特定區域中處理的所有要求數目。

TokensConsumed

Requests根據 a Request 消耗的資源量 (處理時間和記憶體) 量來配置 Token。通常，每個都Request會消耗一個令牌。但是，消耗大量資源的消耗會根據需要分配額外的令牌。Request

單位：計數。分配給在特定區域中處理的請求的令牌數量。

NetworkBandwidthOutAllowanceExceeded

> **Note**
>
> 在 AWS AppSync 主控台的「快取設定值」頁面上，「快取 He alth 況測量結果」選項可讓您啟用此快取相關的健全狀況測量結果。

網路封包因為輸送量超過彙總頻寬限制而丟棄。這對於診斷快取組態中的瓶頸非常有用。透過API_Id在appsyncCacheNetworkBandwidthOutAllowanceExceeded測量結果中指定特定 API 來記錄資料。

單位：計數。超過 ID 指定之 API 的頻寬限制後丟棄的封包數目。

EngineCPUUtilization

> **Note**
>
> 在 AWS AppSync 主控台的「快取設定值」頁面上，「快取 He alth 況測量結果」選項可讓您啟用此快取相關的健全狀況測量結果。

分配給 Redis 處理序的 CPU 使用率 (百分比)。這對於診斷快取組態中的瓶頸非常有用。透過API_Id在appsyncCacheEngineCPUUtilization測量結果中指定特定 API 來記錄資料。

單位：百分比。Redis 處理序目前針對 ID 指定的 API 所使用的 CPU 百分比。

即時訂閱

所有指標都會在一個維度中發出：GraphQLAPIId。這意味著所有指標都與 GraphQL API ID 相結合。以下指標與純粹 WebSockets的 GraphQL 訂閱相關：

量度清單

ConnectRequests

發出的 WebSocket 連線要求數目 AWS AppSync，包括成功嘗試和失敗的嘗試次數。

單位：計數。使用「總和」統計資料來取得連線要求的總數。

ConnectSuccess

成功 WebSocket 連線的數目 AWS AppSync。您可以在沒有訂閱的情況下連線。

單位：計數。使用總和統計以取得這些成功連線的總次數。

ConnectClientError

因為用戶端錯誤而 AWS AppSync 遭拒絕的 WebSocket 連線數目。這可能表示服務受到限制，或授權設定設定錯誤。

單位：計數。使用總和統計以取得這些用戶端連線錯誤的總次數。

ConnectServerError

處理連線 AWS AppSync 時產生的錯誤數目。發生未預期的伺服器端問題時，通常會發生這種情況。

單位：計數。使用總和統計以取得這些伺服器端連線錯誤的總次數。

DisconnectSuccess

成功中 WebSocket AWS AppSync斷連線的數目。

單位：計數。使用總和統計以取得這些成功中斷連線的總次數。

DisconnectClientError

中斷 WebSocket連線 AWS AppSync 時產生的用戶端錯誤數目。

單位：計數。使用總和統計以取得這些中斷連線錯誤的總次數。

DisconnectServerError

中斷 WebSocket連線 AWS AppSync 時產生的伺服器錯誤數目。

單位：計數。使用總和統計以取得這些中斷連線錯誤的總次數。

SubscribeSuccess

AWS AppSync 透過順利註冊的訂閱數目 WebSocket。沒有訂閱的情況下可能有連接，但沒有連接就不可能有訂閱。

單位：計數。使用總和統計以取得這些成功訂閱的總次數。

SubscribeClientError

因為用戶端錯誤而 AWS AppSync 遭拒絕的訂閱數目。當 JSON 承載不正確、服務限制或授權設定錯誤時，就可能會發生這種情況。

單位：計數。使用總和統計以取得這些用戶端訂閱錯誤的總次數。

SubscribeServerError

處理訂閱 AWS AppSync 時產生的錯誤數目。發生未預期的伺服器端問題時，通常會發生這種情況。

單位：計數。使用總和統計以取得這些伺服器端訂閱錯誤的總次數。

UnsubscribeSuccess

已成功處理的取消訂閱要求數目。

單位：計數。使用「總和」統計資料來取得成功取消訂閱要求的總發生次數。

UnsubscribeClientError

AWS AppSync 因為用戶端錯誤而遭拒絕的取消訂閱要求數目。

單位：計數。使用「總和」統計資料來取得用戶端取消訂閱要求錯誤的總發生次數。

UnsubscribeServerError

處理取消訂閱要求 AWS AppSync 時產生的錯誤數目。發生未預期的伺服器端問題時，通常會發生這種情況。

單位：計數。使用「總和」統計資料來取得伺服器端取消訂閱要求錯誤的總發生次數。

PublishDataMessageSuccess

已成功發佈的訂閱事件訊息數目。

單位：計數。使用總和統計以取得已順利發佈之訂閱事件訊息的總計。

PublishDataMessageClientError

因為用戶端錯誤而無法發佈的訂閱事件訊息數目。

Unit：計數。使用總和統計以取得這些用戶端發佈訂閱事件錯誤的總次數。

PublishDataMessageServerError

發佈訂閱事件訊息 AWS AppSync 時產生的錯誤數目。發生未預期的伺服器端問題時，通常會發生這種情況。

單位：計數。使用總和統計以取得這些伺服器端發佈訂閱事件錯誤的總次數。

PublishDataMessageSize

已發佈的訂閱事件訊息大小。

單位：位元組。

ActiveConnections

1 分鐘 AWS AppSync 內從用戶端同時 WebSocket 連線的數目。

單位：計數。使用總和統計來取得開啟的連線總數。

ActiveSubscriptions

在 1 分鐘內來自用戶端同時訂閱的數量。

單位：計數。使用總和統計來取得使用中的訂閱總數。

ConnectionDuration

連線保持開啟的時間。

單位：毫秒。使用平均統計資料來評估連線持續時間。

OutboundMessages

成功發佈的計量付費訊息數目。一個計量訊息等於 5 kB 的已傳送資料。

單位：計數。使用「總和」統計資料可取得成功發佈的計量付費訊息總數。

InboundMessageSuccess

成功處理的輸入訊息數目。由變異叫用的每個訂閱類型都會產生一個輸入訊息。

單位：計數。使用「總和」統計資料可取得已成功處理之輸入訊息的總數。

InboundMessageError

因無效 API 要求 (例如超過 240 kB 訂閱承載資料大小限制) 而無法處理的輸入訊息數目。

單位：計數。您可以使用「總和」統計資料，取得發生 API 相關處理失敗的輸入訊息總數。

InboundMessageFailure

因發生錯誤而處理失敗的輸入訊息數目 AWS AppSync。

單位：計數。您可以使用「總和」統計資料，取得發生 AWS AppSync相關處理失敗的輸入訊息總數。

InboundMessageDelayed

延遲輸入訊息的數目。當輸入郵件速率配額或輸出郵件速率配額超出時，輸入郵件可能會延遲。

單位：計數。您可以使用「總和」統計資料來取得延遲的輸入訊息總數。

InboundMessageDropped

卸除的輸入訊息數目。如果超出輸入郵件速率配額或輸出郵件速率配額，則可以卸除輸入郵件。

單位：計數。您可以使用「總和」統計資料來取得捨棄的輸入訊息總數。

InvalidationSuccess

使用的變異成功使訂閱無效 (取消訂閱) 的數目。`$extensions.invalidateSubscriptions()`

單位：計數。使用「總和」統計資料可擷取已成功取消訂閱的訂閱總數。

InvalidationRequestSuccess

成功處理的無效驗證要求數目。

單位：計數。使用「總和」統計資料可取得已成功處理失效要求的總數。

InvalidationRequestError

因無效 API 要求而處理失敗的無效驗證要求數目。

單位：計數。使用「總和」統計資料可取得發生 API 相關處理失敗的無效驗證要求總數。

InvalidationRequestFailure

因發 AWS AppSync生錯誤而處理失敗的無效驗證要求數目。

單位：計數。使用「總和」統計資料可取得發生 AWS AppSync相關處理失敗的無效驗證要求總數。

InvalidationRequestDropped

超過無效驗證要求配額時，捨棄的無效驗證要求數目。

單位：計數。使用「總和」統計資料可取得捨棄的無效驗證要求總數。

比較輸入和輸出郵件

執行突變時，會叫用具有該變異 @aws_subscribe 指令的訂閱欄位。每個訂閱叫用都會產生一個輸入訊息。例如，如果兩個訂閱欄位在 @aws_subscribe 中指定相同的變異，則在呼叫該變異時會產生兩個輸入訊息。

一則輸出訊息等於傳送給 WebSocket 用戶端的 5 kB 資料。例如，將 15 kB 的資料傳送至 10 個用戶端會產生 30 則輸出訊息 (15 kB * 10 個用戶端/每則訊息 5 kB = 30 則訊息)。

您可以要求增加輸入或輸出郵件的配額。如需詳細資訊，請參閱AWS 一般參考指南中的AWS AppSync 端點和配額，以及《Service Quotas 使用指南》中有關要求增加配額的指示。

增強的指標

增強的指標會發出有關 API 使用情況和效能的精細資料，例如 AWS AppSync 請求和錯誤計數、延遲以及快取命中/未命中。所有增強型指標資料都會傳送至您的 CloudWatch 帳戶，您可以設定要傳送的資料類型。

> **Note**
>
> 使用增強型指標時會收取額外費用。如需詳細資訊，請參閱 Amazon 定價中詳細的監控 CloudWatch定價層。

您可以在 AWS AppSync 主控台的各種設定頁面上找到這些指標。在 [API 設定] 頁面上，[增強量度] 區段可讓您啟用或停用下列項目：

1. 解析器測量結果行為：這些選項控制如何收集解析器的其他測量結果。您可以選擇啟用完整要求解析程式測量結果 (請求中所有解析器啟用的測量結果) 或每個解析器測量結果 (僅針對組態設為已啟用的解析器啟用測量結果)。以下是可用的選項：

指標	公制維度	指標名稱	單位	Description
每個解析器的 GraphQL 錯誤	API 識別碼、解析器	圖形錯誤	Count (計數)	每個解析器發生的 GraphQL 錯誤數目。

每個解析器的請求	API 識別碼、解析器	請求	Count (計數)	要求期間發生的呼叫數目。這是在每個解析器的基礎上記錄的。
每個解析器的延遲	API 識別碼、解析器	Latency (延遲)	毫秒	完成解析器調用的時間。延遲以毫秒為單位，並以每個解析器為基礎進行記錄。
每個解析器的快取命中	API 識別碼、解析器	CacheHit	Count (計數)	要求期間快取命中的數目。如果使用了緩存，這將只被發射。以每個解析器為基礎記錄快取命中。
每個解析器的快取遺漏	API 識別碼、解析器	CacheMiss	Count (計數)	要求期間快取遺漏的數目。如果使用了緩存，這將只被發射。快取遺漏是以每個解析器為基礎記錄的。

2. 資料來源指標行為：這些選項可控制收集資料來源其他指標的方式。您可以選擇啟用完整請求資料來源指標 (針對請求中的所有資料來源啟用量度) 或每個資料來源度量 (僅針對設定設為啟用的資料來源啟用量度)。以下是可用的選項：

指標	公制維度	指標名稱	單位	Description
每個資料來源的要求	API 識別碼，資料來源	請求	Count (計數)	要求期間發生的呼叫數目。請求記錄在每個資料來源的基礎上。如果啟用完整

				請求，則每個資料來源都會在中具有自己的項目 CloudWatch。
每個資料來源的延遲	API 識別碼，資料來源	Latency (延遲)	毫秒	完成資料來源呼叫的時間。延遲記錄在每個資料來源的基礎上。
每個資料來源的錯誤	API 識別碼，資料來源	圖形錯誤	Count (計數)	資料來源叫用期間發生的錯誤數目。

3. 作業指標:啟用 GraphQL 作業層級指標。

指標	公制維度	指標名稱	單位	Description
每項作業的要求	API 識別碼，作業	請求	Count (計數)	呼叫指定 GraphQL 作業的次數。
每個作業的 GraphQL 錯誤	API 識別碼，作業	圖形錯誤	Count (計數)	指定的 GraphQL 作業期間發生的 GraphQL 錯誤數目。

CloudWatch 日誌

您可以在任何新的或現有的 GraphQL API 上設定兩種類型的登入：請求層級和欄位層級。

要求層級記錄

設定要求層級記錄 (包含詳細內容) 時，會記錄下列資訊：

- 消耗的令牌數量
- 請求和回應 HTTP 標頭
- 正在要求中執行的 GraphQL 查詢

- 整體操作總結
- 已註冊之新的和現有的 GraphQL 訂閱

欄位層級記錄

設定欄位層級記錄時，會記錄下列資訊：

- 使用每個字段的源和參數生成的請求映射
- 每個欄位的已轉換回應對應，其中包含解析該欄位所產生的資料
- 追蹤每個欄位的資訊

如果您開啟記錄，請 AWS AppSync 管理記 CloudWatch 錄。此程序包含建立日誌群組和日誌串流，並以這些日誌向日誌串流報告。

當您在 GraphQL API 上開啟記錄並提出要求時， AWS AppSync 會在記錄群組下建立記錄群組並記錄資料流。日誌群組的命名是遵循 `/aws/appsync/apis/{graphql_api_id}` 格式。在每個日誌群組中，日誌將進一步分為日誌串流。它們會依照記錄資料回報的上次事件時間進行排序。

每個記錄事件都會以該要求RequestId的 x-amzn- 標記。這可協助您篩選記錄事件， CloudWatch 以取得有關該要求的所有記錄資訊。您可以 RequestId 從每個 GraphQL AWS AppSync 請求的響應標頭中獲取。

欄位層級記錄是用以下日誌層級設定的：

- 無-不擷取欄位層級記錄。
- 錯誤-僅針對發生錯誤的欄位記錄下列資訊：
    - 伺服器回應的錯誤區段
    - 欄位層級錯誤
    - 產生的請求/回應功能已解析錯誤的欄位
- 全部-記錄查詢中所有欄位的下列資訊：
    - 欄位層級追蹤資訊
    - 產生的請求/回應功能已解析每個欄位

監控的好處

您可以使用記錄和指標來識別、故障診斷和最佳化您的 GraphQL 查詢。例如，這些將協助您使用查詢中每個欄位記錄的追蹤資訊以偵錯延遲問題。為了示範，假設您使用 GraphQL 查詢中巢狀的一或多個解析程式。 CloudWatch 記錄檔中的範例欄位作業看起來可能類似下列內容：

```
{
    "path": [
        "singlePost",
        "authors",
        0,
        "name"
    ],
    "parentType": "Post",
    "returnType": "String!",
    "fieldName": "name",
    "startOffset": 416563350,
    "duration": 11247
}
```

這可能對應到 GraphQL 結構描述，類似如下：

```
type Post {
  id: ID!
  name: String!
  authors: [Author]
}

type Author {
  id: ID!
  name: String!
}

type Query {
  singlePost(id:ID!): Post
}
```

在先前的記錄結果中，路徑會顯示執行名為的查詢傳回的資料中的單一項目`singlePost()`。在這個例子中，它代表在第一個索引（0）的名稱字段。該 start Offset 了從 GraphQL 查詢操作的開始的偏移量。持續時間是解析欄位的總時間。這些值非常有用，可以診斷為何來自特定資料來源的資料的執行速

度低於預期，或特定欄位是否降低了整個查詢的速度。例如，您可以選擇增加 Amazon DynamoDB 表的佈建輸送量，或從導致整體操作執行不佳的查詢中移除特定欄位。

自 2019 年 5 月 8 日起，會以完整結構化的 JSON 形式 AWS AppSync 產生記錄事件。這可協助您使用日誌分析服務 (例如 CloudWatch 日誌洞見和 Amazon OpenSearch 服務) 來瞭解 GraphQL 請求的效能和結構描述欄位的使用特性。例如，您可以輕鬆找出延遲時間很長，並且可能是效能問題根本原因的解析程式。您也可以找出結構描述中最常用和最不常用的欄位，並評估移除 GraphQL 欄位的影響。

衝突偵測與同步處理記錄

如果 AWS AppSync API 已將 「欄位解析程式」記錄層級設定為「全部」的「記錄 CloudWatch 檔」記錄，則會向記錄群組 AWS AppSync 發出衝突偵測和解決方案資訊。這提供了 AWS AppSync API 如何回應衝突的詳細見解。為了協助您解譯回應，記錄檔中提供下列資訊：

量度清單

`conflictType`

詳述發生的衝突原因是版本不符或客戶提供的條件所致。

`conflictHandlerConfigured`

說明請求時在解析程式中設定的衝突處理常式。

`message`

提供如何偵測和解決衝突的相關資訊。

`syncAttempt`

在最終拒絕請求之前，伺服器為了同步資料而嘗試的次數。

`data`

如果設定的衝突處理程式為`Automerge`，則會填入此欄位以顯示每個欄位所`Automerge`採取的決定。提供的動作可能是：

- 拒絕-當`Automerge`拒絕傳入的字段值有利於服務器中的值時。
- 添加-當`Automerge`由於在服務器中沒有預先存在的值對傳入字段添加。
- 附加-當傳入值`Automerge`追加到服務器中存在的列表的值。
- 合併-將傳入值合`Automerge`併到服務器中存在的集合的值時。

使用令牌計數優化您的請求

消耗小於或等於 1,500 KB 記憶體和 vCPU 時間的要求會分配一個權杖。資源耗用量超過 1,500 KB 秒的要求會收到額外的權杖。例如，如果請求消耗 3,350 KB 秒，則會將三個令牌（四捨五入到下一個整數值） AWS AppSync 分配給請求。根據預設，每秒最多會為帳戶中的 API AWS AppSync 配置 5,000 或 10,000 個要求權杖，視其部署所在的 AWS 區域而定。如果您的 API 每個平均每秒使用兩個令牌，則每秒最多只能使用 2,500 或 5,000 個請求。如果您每秒需要比分配數量更多的令牌，則可以提交請求以增加請求令牌速率的默認配額。如需詳細資訊，請參閱AWS 一般參考 指南中的AWS AppSync 端點和配額和 Service Quotas 使用者指南中的要求增加配額。

每個請求的 Token 計數很高可能表示有機會優化您的請求並提高 API 的性能。可以增加每個請求令牌計數的因素包括：

- 您的 GraphQL 結構描述的大小和複雜性。
- 請求和響應映射模板的複雜性。
- 每個要求的解析程式呼叫次數。
- 從解析器返回的數據量。
- 下游資料來源的延遲。
- 需要連續資料來源呼叫的結構描述和查詢設計 (與 parallel 或批次呼叫相反)。
- 記錄設定，特別是欄位層級和詳細記錄檔內容。

> **Note**
>
> 除了 AWS AppSync 指標和日誌之外，客戶端還可以通過響應標頭訪問請求中使用的令牌數量`x-amzn-appsync-TokensConsumed`。

記錄檔類型參考

RequestSummary

- requestId：請求的唯一識別符。
- graphQLAPIId：提出請求的 GraphQL API ID。
- statusCode：HTTP 狀態碼回應。
- 延遲：請求的 E nd-to-end 延遲（以納秒為單位）作為整數。

```
{
     "logType": "RequestSummary",
     "requestId": "dbe87af3-c114-4b32-ae79-8af11f3f96f1",
     "graphQLAPIId": "pmo28inf75eepg63qxq4ekoeg4",
     "statusCode": 200,
     "latency": 242000000
}
```

ExecutionSummary

- requestId：請求的唯一識別符。
- graphQLAPIId：提出請求的 GraphQL API ID。
- startTime：針對要求進行 GraphQL 處理的開始時間戳記，格式為 RFC 3339。
- endTime：處理要求的 GraphQL 結束時間戳記，格式為 RFC 3339。
- 持續時間：經過的 GraphQL 處理時間總計，以納秒為單位，以整數表示。
- 版本：的綱要版本 ExecutionSummary。
- parsing：
    - start@@ Offset：解析的開始偏移量，以納秒為單位，相對於調用，作為一個整數。
    - duration：剖析所花費的時間，以奈秒為單位，並且為整數。
- validation：
    - start@@ Offset：驗證的開始偏移量，以納秒為單位，相對於調用，作為一個整數。
    - duration：執行驗證所花費的時間，以奈秒為單位，並且為整數。

```
{
    "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"
}
```

追蹤

- requestId：請求的唯一識別符。
- graphQLAPIId：提出請求的 GraphQL API ID。
- start@@ Offset：字段分辨率的開始偏移量，以納秒為單位，相對於調用，作為一個整數。
- duration：解析欄位所花費的時間，以奈秒為單位，並且為整數。
- fieldName：正在解析的欄位名稱。
- parentType：正在解析欄位的父類別。
- returnType：正在解析欄位的傳回類型。
- path：路徑區段的清單，從回應的根開始，並結束於正在解析的欄位。
- resolverArn：用於解析欄位的解析程式 ARN。在巢狀欄位上可能不存在。

```
{
    "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"
}
```

使用日誌見解分析您的 CloudWatch 日誌

以下是您可以執行的查詢範例，您可以透過這些執行取得您 GraphQL 操作效能及運作狀態的可採取動作詳情。這些範例在 CloudWatch 記錄見解主控台中以範例查詢的形式提供。在CloudWatch主

控台中，選擇 [記錄深入解析]，選取 GraphQL API 的記 AWS AppSync 錄群組，然後在 [範例AWS AppSync 查詢] 下選擇 [查詢]。

下列查詢會傳回使用最多權杖的前 10 個 GraphQL 要求：

```
filter @message like "Tokens Consumed"
| parse @message "* Tokens Consumed: *" as requestId, tokens
| sort tokens desc
| display requestId, tokens
| limit 10
```

以下查詢會傳回前 10 個延遲最長的解析程式：

```
  fields resolverArn, duration
| filter logType = "Tracing"
| limit 10
| sort duration desc
```

以下查詢會傳回最常呼叫的解析程式：

```
  fields ispresent(resolverArn) as isRes
| stats count() as invocationCount by resolverArn
| filter isRes and logType = "Tracing"
| limit 10
| sort invocationCount desc
```

以下查詢會傳回映射範本中錯誤最多的解析程式：

```
  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
```

以下查詢會傳回解析程式的延遲統計資料：

```
  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
```

以下查詢會傳回欄位延遲統計資料：

```
  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
```

CloudWatch 日誌見解查詢的結果可以匯出到 CloudWatch 儀表板。

使用服務分析您的 OpenSearch 日誌

您可以使用 Amazon Ser OpenSearch vice 搜尋、分析和視覺化您的 AWS AppSync 日誌，以識別效能瓶頸和操作問題的根本原因。您可以找出延遲最長和錯誤最多的解析程式。此外，您可以使用 OpenSearch 儀表板來建立具有強大視覺效果的儀表板。 OpenSearch 儀表板是 OpenSearch 服務中提供的開源數據可視化和探索工具。您可以使用 OpenSearch 儀表板持續監控 GraphQL 作業的效能和健康狀態。例如，您可以建立儀表板以視覺化 GraphQL 請求的 P90 延遲，並深入研究每個解析器的 P90 延遲。

使用 OpenSearch 服務時，請使用 「cwl*」 作為過濾器模式來搜索索引。 OpenSearch OpenSearch 服務會為前置詞為 「cwl-」從 CloudWatch 記錄檔串流處理的記錄建立索引。若要區分 AWS AppSync API 記錄與傳送至 OpenSearch 服務的其他 CloudWatch 記錄檔，建議您在搜尋中新增額外的`graphQLAPIID.keyword=`*`YourGraphQLAPIID`*篩選器運算式。

記錄檔格式遷移

在 2019 年 5 月 8 日或之後 AWS AppSync 產生的記錄事件會格式化為完整結構化 JSON。若要在 2019 年 5 月 8 日之前分析 GraphQL 要求，您可以使用GitHub 範例中提供的指令碼，將較舊的記錄檔移轉至完全結構化的 JSON。若您需要使用 2019 年 5 月 8 日以前的日誌格式，請使用下列設定建立支援票證：將 Type (類型) 設為 Account Management (帳戶管理)，然後將 Category (類別) 設為 General Account Question (一般帳戶問題)。

您也可以在中使用指標篩選器，將記錄資料轉換 CloudWatch 為數字 CloudWatch 指標，以便您可以在其上繪製圖形或設定警示。

使用追蹤AWS X-Ray

您可以使用AWS X-Ray來追蹤在執行的請求AWSAppSync。您可以將 X-RayAWS在所有AppSyncAWS提供 X-Ray 的區域。X-Ray 為您提供整個 GraphQL 請求的詳細概述。這可讓您分析 API 及其基礎解析程式和資料來源中的延遲。您可以使用 X-Ray 服務對應來檢視請求的延遲，包括任何AWS與 X-Ray 集成的服務。您也可以設定取樣規則，以根據您指定的條件告知 X-Ray 要記錄哪些請求，以及使用何種取樣率。

如需 X-Ray 取樣的詳細資訊，請參見在AWS X-Ray主控台。

設定與組態

您可以 GraphQL 過AWSAppSync 主控台。

1. 前往登入AWSAppSync 主控台。
2. 在導覽窗格中選擇 Settings (設定)。
3. 在 X-Ray 下，開啟 Enable X-Ray (啟用 X-Ray)。
4. 選擇 Save (儲存)。現在已為您的 API 啟用 X-Ray 追蹤。

如果您使用AWS CLI或者AWS CloudFormation，您還 X-Ray 以在創建新AWS應用同步 API，或更新現有AWS應用同步 API，通過設置`xrayEnabled`屬性設置為`true`。

X-Ray 為AWSAppSync API，AWS Identity and Access Management 服務連結角色會利用適當的許可，在您的帳戶中自動建立。這允許AWSAppSync 以安全的方式將追蹤傳送給 X-Ray。

使用 X-Ray 跟蹤您的 API

抽樣

透過取樣規則，您可以控制AWSAppSync，並且可迅速修改取樣行為，而無需修改或重新部署程式碼。例如，此規則使用 API ID `3n572shhcpfokwhdnq1ogu59v6` 對 GraphQL API 的請求進行取樣。

- 規則名稱 — `test-sample`
- 優先順序 — `10`
- 儲槽大小 — `10`
- 固定頻率 — `10`

- 服務名稱 — *
- 服務類型 — AWS::AppSync::GraphQLAPI
- HTTP 方法 — *
- 資源 ARN — arn:aws:appsync:us-west-2:123456789012:apis/3n572shhcpfokwhdnq1ogu59v6
- 主機 — *

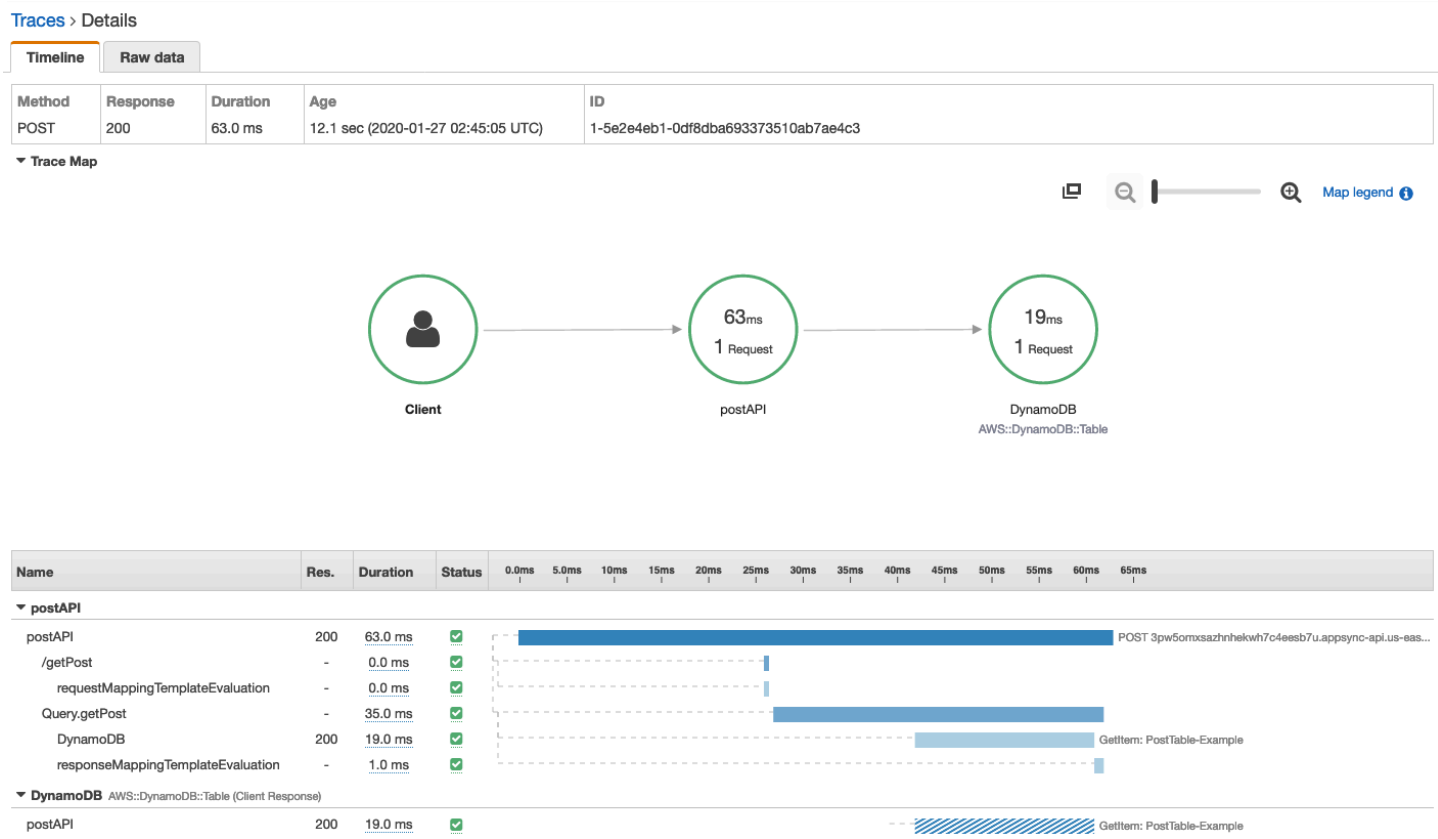
了解追蹤

為 GraphQL API 啟用 X-Ray 追蹤時，您可以使用 X-Ray 追蹤詳細資訊頁面檢視對 API 提出請求的詳細延遲資訊。以下範例顯示此特定請求的追蹤檢視以及服務對應。請求是針對名稱為postAPI，其資料包含 Amazon DynamoDB 稱為PostTable-Example。

下列追蹤影像對應下列 GraphQL 查詢：

```
query getPost {
    getPost(id: "1") {
      id
      title
    }
}
```

針對getPost查詢會使用底層 DynamoDB 資料源。以下追蹤檢視顯示對 DynamoDB 的調用，以及查詢執行各部分的延遲：



- 在上述影像中，/getPost 表示要解析之元素的完整路徑。在這種情況下，因為 getPost 是根 Query 類型的欄位，它直接出現在路徑的根之後。
- requestMappingTemplateEvaluation表示AWSAppSync 為查詢中此元素評估請求映射範本。
- Query.getPost 表示類型和欄位 (格式為 Type.field)。它可以包含多個子區段，取決於 API 的結構和要追蹤的請求而定。
  - DynamoDB 表示附加至此解析程式的資料來源。它包含對 DynamoDB 進行網路呼叫以解析字段的延遲。
  - responseMappingTemplateEvaluation表示AWSAppSync 為查詢中此元素評估回應映射範本。

當您在 X-Ray 中檢視追蹤時，您可以在AWSAppSync 區段，方法是選擇子段並瀏覽詳細視圖。

對於某些深度嵌套或複雜的查詢，請注意AWSAppSync 可以大於段文檔允許的最大大小，如AWS X-Ray區段文件。X-Ray 不顯示超過限制的段。

使用 AWS CloudTrail 記錄 AWS AppSync API 呼叫

AWS AppSync 與 AWS CloudTrail 整合，該服務提供由使用者、角色或 AWS 服務在 AWS AppSync 中所採取的動作記錄。CloudTrail 會擷取 AWS AppSync 的所有 API 呼叫當作事件。擷取的呼叫包括從 AWS AppSync 主控台的呼叫，以及對 AWS AppSync API 的程式碼呼叫。您可以使用收集的信息CloudTrail以確定提出的請求AWS AppSync、要求者的 IP 位址、提出要求的人、提出要求的時間，以及其他詳細資訊。

您可以建立線索啟用持續交付CloudTrail亞馬遜簡單儲存服務 (Amazon S3) 儲存貯體的事件，包括以下事件AWS AppSync。如果您未設定追蹤，您仍然可以在CloudTrail控制台。

> **Important**
>
> 目前並非所有 GraphQL 動作都會記錄下來。AppSync不會將查詢和變異動作記錄到 CloudTrail。

如需有關 CloudTrail 的詳細資訊，請參閱《使用者指南》AWS CloudTrail。

CloudTrail 中的 AWS AppSync 資訊

當您建立帳戶時，系統會在您的 AWS 帳戶中啟用 CloudTrail。在CloudTrail控制台事件歷史，您可以查看，搜索和下載最近的事件AWS帳戶。如需詳細資訊，請參閱檢視事件CloudTrail事件歷史在AWS CloudTrail使用者指南。

如需您 AWS 帳戶中正在進行事件的記錄 (包含 AWS AppSync 的事件)，請建立線索。根據預設，當您在主控台建立線索時，線索會套用到所有 AWS 區域。該追蹤會記錄來自 AWS 分割區中所有區域的事件，並將日誌檔案交付到您指定的 Amazon S3 儲存貯體。此外，您可以設定其他 AWS 服務，以進一步分析和處理 CloudTrail 日誌中所收集的事件資料。如需詳細資訊，請參閱AWS CloudTrail使用者指南:

- 為您的建立追蹤AWS帳號
- AWS服務整合CloudTrail日誌
- 設定 CloudTrail 的 Amazon SNS 通知
- 從多個區域接收 CloudTrail 日誌檔案
- 從多個帳戶接收 CloudTrail 日誌檔案

CloudTrail 會記錄所有 AWS AppSync API 操作。例如，呼叫`CreateGraphqlApi`,`CreateDataSource`，以及`ListResolvers`API 會在CloudTrail日誌文件。這些和其他操作記錄在AWS AppSyncAPI 參考資料。

每一筆事件或日誌項目都會包含產生請求者的資訊。身分資訊可協助您判斷：

- 該請求是否透過根或 AWS Identity and Access Management (IAM) 使用者憑證來提出。
- 提出該請求時，是否使用了特定角色或聯合身分使用者的暫時安全憑證。
- 該請求是否由另一項 AWS 服務提出。

如需詳細資訊，請參閱CloudTrail使用者識別元素在AWS CloudTrail使用者指南。

了解 AWS AppSync 日誌檔案項目

CloudTrail將事件傳送為包含一或多個記錄項目的記錄檔。事件代表來自任何來源的單一請求，並包括有關請求的操作，操作的日期和時間，請求參數等信息。因為這些記錄檔不是公用 API 呼叫的排序堆疊追蹤，因此不會以任何特定順序顯示。

下面的例子CloudTrail記錄項目示範`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"
    }
  ]
}
```

下面的例子CloudTrail記錄項目示範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"
    }
  ]
}
```

下面的例子CloudTrail記錄項目示範DeleteApiKey操作。

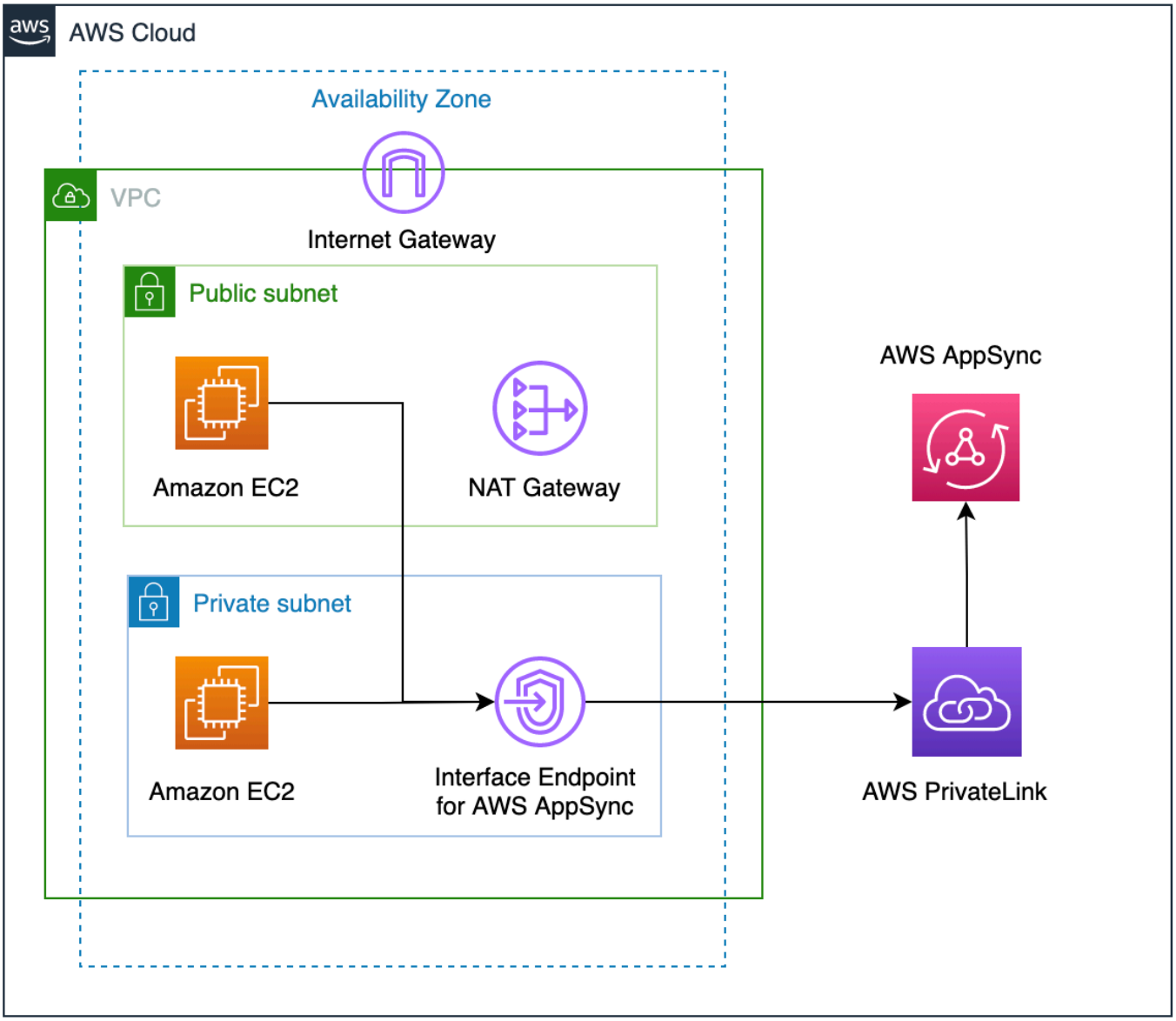
```
{
  "Records": [{
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "A1B2C3D4E5F6G7EXAMPLE",
      "arn": "arn:aws:iam::111122223333:user/Alice",
      "accountId": "111122223333",
      "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
      "userName": "Alice"
    },
    "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

如果您使用亞馬遜虛擬私有雲（亞馬遜 VPC），則可以創建AWS AppSync私有 API，這是只能從 VPC 存取的 API。使用私有 API，您可以限制對內部應用程式的 API 存取，並連線到 GraphQL 和即時端點，而不會公開資料。

若要在您的 VPC 和AWS AppSync服務，您必須建立VPC 端點介面。介面端點是由 AWS PrivateLink 供電，讓您不需要網際網路閘道、NAT 裝置、VPN 連線或 AWS Direct Connect 連線即可私密存取 AWS AppSync API。VPC 中的執行個體不需要公有 IP 地址，即能與 AWS AppSync API 通訊。 您的 VPC 和之間的流量AWS AppSync不離開AWS網絡。



在啟用私有 API 功能之前，還有一些其他因素需要考慮：

- 設定下列項目的 VPC 介面端點AWS AppSync啟用私有 DNS 功能將防止 VPC 中的資源叫用其他 AWS AppSync公用 API 使用AWS AppSync產生的 API 網址。這是由於對公共 API 的請求通過接口端點進行路由，這是公共 API 不允許的。若要在此案例中叫用公用 API，建議您在公用 API 上設定自訂網域名稱，然後 VPC 中的資源可使用這些網域名稱來叫用公用 API。
- 您的AWS AppSync只有您的 VPC 才能使用私有 API。該AWS AppSync主控台查詢編輯器只有在瀏覽器的網路設定可以將流量路由到您的 VPC (例如透過 VPN 或透過 VPN 連線) 時，才能存取您的APIAWS Direct Connect).

- 使用 VPC 介面端點AWS AppSync，您可以訪問相同的任何私有 APIAWS帳戶和地區。要進一步限制對私有 API 的訪問，您可以考慮以下選項：
  - 確保只有必要的管理員可以建立 VPC 端點介面AWS AppSync。
  - 使用 VPC 端點自訂原則來限制可從 VPC 中的資源叫用哪些 API。
  - 對於 VPC 中的資源，建議您使用 IAM 授權來叫用AWS AppSync通過確保將資源分配給 API 的範圍內角色來實現 API。
- 建立或使用限制 IAM 主體的政策時，您必須設定`authorizationType`的方法`AWS_IAM`或者`NONE`。

創建AWS AppSync私有 API

下面的步驟說明如何創建私有 APIAWS AppSync服務。

> **Warning**
>
> 您只能在建立 API 期間啟用私有 API 功能。無法修改此設定AWS AppSync應用程式介面或AWS AppSync私人 API 在創建之後。

1. 登入 AWS Management Console 並開啟 AppSync主控台。
   - 在儀表板上，選擇 Create API (建立 API)。
2. 選擇從頭開始設計 API，然後選擇下一步。
3. 在私有 API區段中，選擇使用私有 API 功能。
4. 設定其餘選項，檢閱 API 的資料，然後選擇創建。

在您可以使用AWS AppSync私有 API，您必須配置接口端點AWS AppSync在您的虛擬私人雲端。請注意，私有 API 和 VPC 必須位於相同AWS帳戶和地區。

為建立介面端點AWS AppSync

您可以為下列項目建立介面端點AWS AppSync使用亞馬遜 VPC 控制台或AWS Command Line Interface(AWS CLI). 如需詳細資訊，請參閱 Amazon VPC 使用者指南中的建立介面端點。

Console

1. 登入AWS Management Console並打開端點亞馬遜 VPC 控制台的頁面。

2. 選擇 Create endpoint (建立端點)。
   a. 在服務類別欄位中，確認AWS服務已選取。
   b. 在服務表格中，選擇`com.amazonaws.{region}.appsync-api`。驗證类型列值是`Interface`。
   c. 在VPC」欄位中，選擇 VPC 及其子網路。
   d. 若要為介面端點啟用私有 DNS 功能，請勾選啟用 DNS 名稱核取方塊。
   e. 在安全性群組」欄位中，選擇一或多個安全性群組。
3. 選擇 Create endpoint (建立端點)。

CLI

使用 `create-vpc-endpoint` 命令，並指定 VPC ID、VPC 端點 (介面) 的類型、服務名稱、要使用端點的子網路，以及要與端點網路介面建立關聯的安全群組。例如：

```
$ aws ec2 create-vpc-endpoint –vpc-id vpc-ec43eb89 \
  –vpc-endpoint-type Interface \
  –service-name com.amazonaws.{region}.appsync-api \
  –subnet-id subnet-abababab –security-group-id sg-1a2b3c4d
```

若要使用私有 DNS 選項，您必須設定`enableDnsHostnames`和`enableDnsSupportattributes`您的虛擬私人雲端的值。如需詳細資訊，請參閱 Amazon VPC 使用者指南中的檢視並更新 VPC 的 DNS 支援。如果您為介面端點啟用私有 DNS 功能，您可以向您的AWS AppSyncAPI GraphQL 和即時端點使用其預設公有 DNS 端點，格式如下：

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

如需服務端點的詳細資訊，請參閱服務端點和配額在AWS一般參考。

如需服務與介面端點互動的詳細資訊，請參閱透過介面端點存取服務在亞馬遜 VPC 用戶指南。

如需使用建立和設定端點的相關資訊AWS CloudFormation，請參閱AWS:: EC2:: 點點中的資源AWS CloudFormation使用者指南。

進階 範例

如果您為介面端點啟用私有 DNS 功能，您可以向您的AWS AppSyncAPI GraphQL 和即時端點使用其預設公有 DNS 端點，格式如下：

```
https://{api_url_identifier}.appsync-api.{region}.amazonaws.com/graphql
```

使用介面 VPC 端點公用 DNS 主機名稱，呼叫 API 的基本 URL 將採用下列格式：

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}.appsync-api.
{region}.vpce.amazonaws.com/graphql
```

如果您已在 AZ 中部署端點，也可以使用 AZ 特定的 DNS 主機名稱：

```
https://{vpc_endpoint_id}-{endpoint_dns_identifier}-{az_id}.appsync-api.
{region}.vpce.amazonaws.com/graphql.
```

使用 VPC 端點公用 DNS 名稱將需要AWS AppSync要傳遞的 API 端點主機名稱`Host`或作為 `x-appsync-domain`頭的請求。這些範例使用TodoAPI這是在啟動範例結構描述指南：

```
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"}}}'
```

在下面的實例中，我們將使用待辦事項在中產生的應用程式啟動範例結構描述指南。為了測試樣本待辦事項 API，我們將使用私有 DNS 來調用 API。您可以使用您選擇的任何命令行工具; 這個例子使用捲曲發送查詢和突變wscat以設定訂閱。為了模擬我們的例子，請替換括號中的值{ }在下面的命令中，包含來自您的相應值AWS帳戶。

測試突變操作 —**`createTodo`**請求

```
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"}}}'
```

測試突變操作 —**createTodo**回應

```
{
    "data": {
        "createTodo": {
            "id": "<todo-id>",
            "name": "My first GraphQL task",
            "where": "Day 1",
            "when": "Friday Night",
            "description": "Learn more about GraphQL"
        }
    }
}
```

測試查詢操作 —**listTodos**請求

```
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"}}}'
```

測試查詢操作 —**listTodos**請求

```
{
  "data": {
    "listTodos": {
      "items": [
        {
          "description": "Learn more about GraphQL",
          "id": "<todo-id>",
          "name": "My first GraphQL task",
          "when": "Friday night",
          "where": "Day 1"
        }
```

```
      ]
    }
  }
}
```

測試訂閱操作 — 訂閱**createTodo**突變

若要在中設定 GraphQL 訂閱AWS AppSync，請參閱建立即時WebSocket用戶端。從 VPC 中的 Amazon EC2 執行個體，您可以測試AWS AppSync私有 API 訂閱端點使用wscat。下面的例子使用API KEY用於授權。

```
$ 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，請使用 VPC 端點網域名稱，同時確保指定主持人中的標頭wscat命令來建立網絡套接字：

```
$ 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"}
```

運行以下突變代碼：

```
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"}}}'
```

之後，會觸發訂閱，並顯示訊息通知，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 政策限制公用 API 建立

AWS AppSync支援 IAMCondition聲明與私有 API 一起使用。該visibility欄位可包含在以下項目的 IAM 政策聲明中appsync:CreateGraphqlApi用於控制哪些 IAM 角色和使用者可以建立私有和公有 API 的操作。這讓 IAM 管理員能夠定義只允許使用者建立私有 GraphQL API 的 IAM 政策。嘗試建立公用 API 的使用者會收到未經授權的訊息。

例如，IAM 管理員可以建立下列 IAM 政策陳述式，以允許建立私有 API：

```
{
    "Sid": "AllowPrivateAppSyncApis",
    "Effect": "Allow",
    "Action": "appsync:CreateGraphqlApi",
    "Resource": "*",
    "Condition": {
        "ForAnyValue:StringEquals": {
            "appsync:Visibility": "PRIVATE"
        }
    }
}
```

IAM 管理員也可以新增下列項目服務控制政策封鎖中的所有使用者AWS組織, 從, 建立AWS AppSync私有 API 以外的 API：

```
{
    "Sid": "BlockNonPrivateAppSyncApis",
    "Effect": "Deny",
    "Action": "appsync:CreateGraphqlApi",
    "Resource": "*",
    "Condition": {
        "ForAnyValue:StringNotEquals": {
            "appsync:Visibility": "PRIVATE"
        }
    }
}
```

使用設定 GraphQL 執行複雜度、查詢深度和內部檢查 AWS AppSync

AWS AppSync可讓您啟用或停用內部檢查功能，並在單一查詢中設定巢狀層級和解析器數量的限制。

使用內部檢查功能

> **Tip**
>
> 如需 GraphQL 內部檢查的詳細資訊，請參閱 GraphQL 基金會網站上的這篇文章。

默認情況下，GraphQL 允許您使用內部檢查來查詢模式本身以發現其類型，字段，查詢，突變，訂閱等。這是學習 GraphQL 服務如何塑造和處理資料的重要功能。但是，在處理自省時，有一些事情需要考慮。您可能有一個可以受益於禁用內部檢查的用例，例如字段名稱可能是敏感或隱藏的情況，或者完整的 API 模式旨在為消費者保留未記錄。在這些情況下，透過內部檢查發佈結構描述資料可能會導致蓄意洩漏私有資料。

若要避免這種情況發生，您可以停用內部檢查。這樣可以防止未經授權的人員在您的結構描述中使用內部檢查欄位。不過，請務必注意，內部檢查對於開發團隊瞭解服務中資料的處理方式非常有用。在內部，在生產代碼中禁用它作為額外的安全層時，保持啟用內部檢查可能會有所幫助。處理這個問題的另一種方法是添加一個授權方法，該方法AWS AppSync也提供了。如需詳細資訊，請參閱授權。

AWS AppSync可讓您在 API 層級啟用或停用內部檢查。若要啟用或停用內部檢查，請執行下列動作：

1. 登入 AWS Management Console 並開啟 AppSync主控台。
2. 在 [API] 頁面上，選擇 GraphQL API 的名稱。

3. 在 API 首頁的導覽窗格中，選擇 [設定]。
4. 在 API 設定中，選擇編輯。
5. 在內部檢查查詢下，執行下列操作：
   - 開啟或關閉「啟用內部檢查查詢」。
6. 選擇儲存。

啟用內部檢查 (預設行為) 時，使用內部檢查系統將會正常運作。例如，下圖顯示了處理結構描述中所有可用類型的__schema欄位：

停用此功能時，回應中會出現驗證錯誤：

設定查詢深度限制

在某些時候，您可能需要更精細地控制 API 在操作期間的運作方式。其中一個控制項是為查詢可能處理的巢狀層級數量增加限制。根據預設，查詢能夠處理無限數量的巢狀層級。將查詢限制為特定數量的巢狀層級會對專案的效能和彈性造成潛在影響。採取下列查詢：

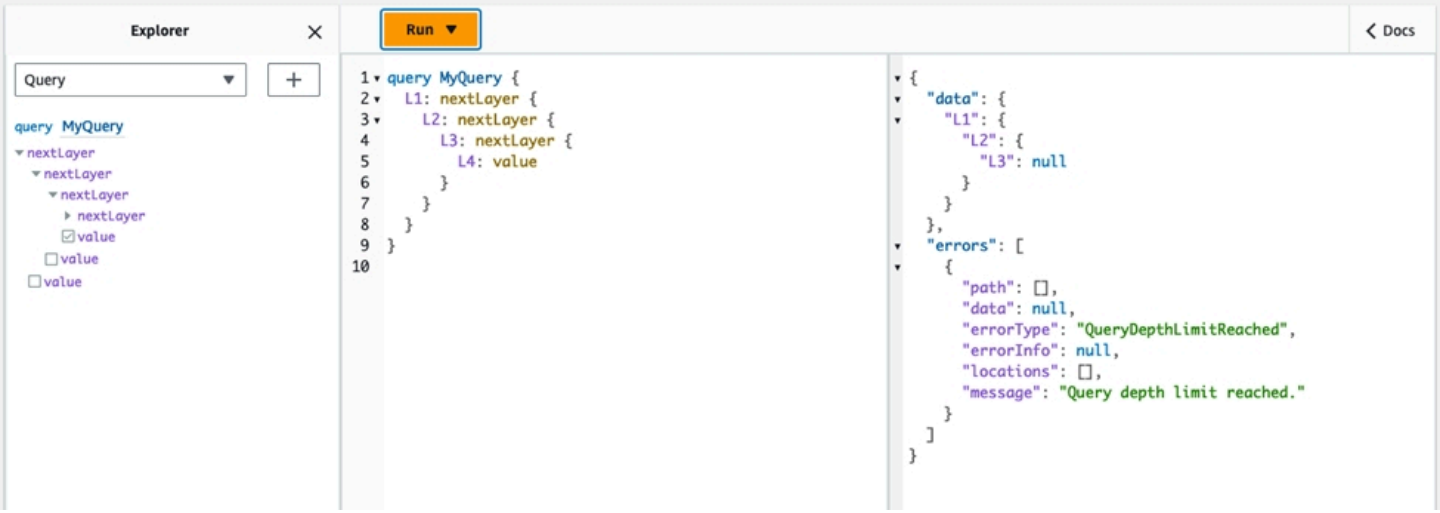
```
query MyQuery {
  L1: nextLayer {
    L2: nextLayer {
      L3: nextLayer {
        L4: value
      }
    }
  }
}
```

您的項目可能需要將查詢限制為某種目的L1或L2出於某種目的。默認情況下，從L1到的整個查詢L4將被處理，而無法控制。通過設置限制，您可以防止查詢訪問超過指定級別的任何內容。

若要新增查詢深度限制，請執行下列動作：

1. 登入 AWS Management Console 並開啟 AppSync主控台。
2. 在 [API] 頁面上，選擇 GraphQL API 的名稱。
3. 在 API 首頁的導覽窗格中，選擇 [設定]。
4. 在 API 設定中，選擇編輯。
5. 在查詢深度之下，執行下列操作：
   a. 打開或關閉「啟用查詢深度」。
   b. 在「最大深度」中，設定深度限制。這可以介於1和之間75。
6. 選擇儲存。

設置限制時，超過其上限將導致QueryDepthLimitReached錯誤。例如，下圖顯示了一個查詢，其深度限制超2過第三個 (L3) 和第四個 (L4) 層級的限制：



請注意，在結構描述中，欄位仍可標示為空或不可為空。如果一個不可為空的字段收到一個QueryDepthLimitReached錯誤，該錯誤將被拋出到第一個可為空的父字段。

設定解析器計數限制

您還可以控制每個查詢可以處理多少個解析器。就像查詢深度一樣，您可以設定此數量的限制。採取包含三個解析器的以下查詢：

```
query MyQuery {
  resolver1: resolver
  resolver2: resolver
  resolver3: resolver
}
```

依預設，每個查詢最多可處理 10000 個解析器。在上面的範例中resolver1，resolver2、和resolver3將被處理。但是，您的項目可能需要限制每個查詢總共處理一個或兩個解析器。通過設置限制，您可以告訴查詢不要處理超過某個數字的任何解析器，例如第一個（resolver1）或第二個（）解析器。resolver2

若要新增解析器計數限制，請執行下列操作：

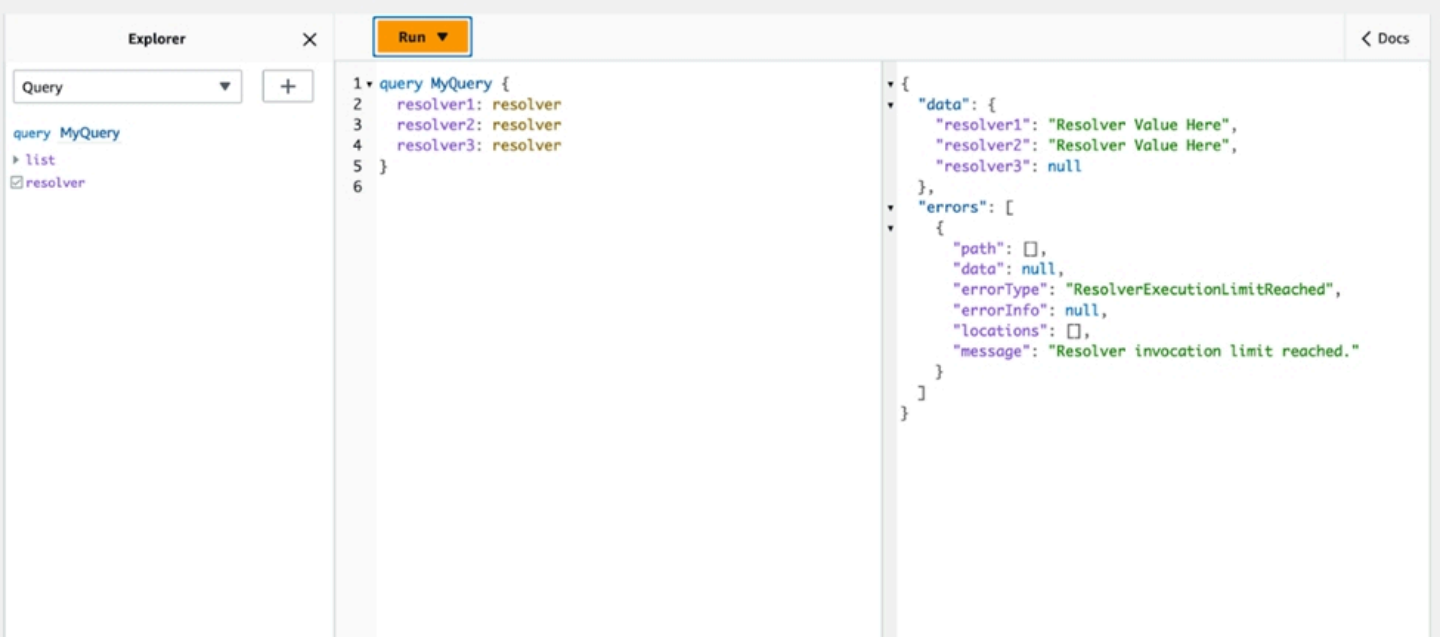
1. 登入 AWS Management Console 並開啟 AppSync主控台。
2. 在 [API] 頁面上，選擇 GraphQL API 的名稱。
3. 在 API 首頁的導覽窗格中，選擇 [設定]。
4. 在 API 設定中，選擇編輯。
5. 在解析器計數限制下，執行下列操作：

a. 開啟「啟用解析器計數」。

b. 在最大解析器計數中，設定計數限制。這可以介於1和之間10000。

6. 選擇儲存。

與查詢深度限制一樣，超過配置的解析器限制會導致查詢在其他解析器上以ResolverExecutionLimitReached錯誤結束。在下圖中，解析器計數限制為 2 的查詢會嘗試處理三個解析器。由於限制，第三個解析器拋出錯誤並且不運行。



使用環境變數 AWS AppSync

您可以使用環境變量來調整 AWS AppSync 解析器和函數的行為，而無需更新代碼。環境變量是與您的 API 配置一起存儲的字符串對，可供解析器和函數在運行時使用。對於您必須參考僅在初始設定期間可用的組態資料，但在執行期間需要由解析器和函數使用的組態資料時，這些資料特別有用。環境變數會在程式碼中公開組態資料，因此減少對這些值進行硬式編碼的需求。

> **Note**
>
> 若要提高資料庫安全性，建議您使用 Secrets Manager 或 AWS Systems Manager 參數存放區，而不要使用環境變數來儲存認證或機密資訊。若要利用此功能，請參閱使用 AWS AppSync HTTP 資料來源叫用 AWS 服務。

環境變量必須遵循幾個行為和規則才能正常運行：

- JavaScript 解析器/函數和 VTL 模板都支持環境變量。
- 在函數調用之前不會評估環境變量。
- 環境變量僅支持字符串值。
- 環境變數中的任何定義值都會被視為字串常值，而不會展開。
- 理想情況下，應該在函數代碼中執行變量評估。

配置環境變量（控制台）

您可以建立變數並定義其索引鍵值配對，以設定 AWS AppSync GraphQL API 的環境變數。您的解析器和函數將使用環境變量的密鑰名稱在運行時檢索值。若要在 AWS AppSync 主控台中設定環境變數：

1. 登入 AWS Management Console 並開啟AppSync主控台。
2. 在 [API] 頁面上，選擇 GraphQL API 的名稱。
3. 在 API 首頁的導覽窗格中，選擇 [設定]。
4. 在環境變數下，選擇新增環境變數。
5. 選擇 Add environment variable (新增環境變數)。
6. 輸入索引鍵和值。
7. 如有必要，請重複步驟 5 和 6 以加入更多關鍵值。如果您需要移除索引鍵值，請選擇 [移除] 選項和要移除的金鑰。
8. 選擇提交。

> **Tip**
>
> 創建鍵和值時必須遵循一些規則：
>
> - 金鑰必須以字母開頭。
> - 金鑰必須至少有兩個字元。
> - 鍵只能包含字母、數字和底線字元 (_)。
> - 值最多可以有 512 個字元。
> - 您可以在 GraphQL API 中設定多達 50 個鍵值組。

設定環境變數 (API)

若要使用 API 設定環境變數，您可以使用PutGraphqlApiEnvironmentVariables. 對應的 CLI 命令是put-graphql-api-environment-variables。

若要使用 API 擷取環境變數，您可以使用GetGraphqlApiEnvironmentVariables. 對應的 CLI 命令是get-graphql-api-environment-variables。

該命令必須包含 API ID 和環境變量列表：

```
aws appsync put-graphql-api-environment-variables \
  --api-id "<api-id>" \
  --environment-variables '{"key1":"value1","key2":"value2", …}'
```

下列範例會在 API 中設定兩個環境變數，其 ID 為abcdefghijklmnopqrstuvwxyz使用put-graphql-api-environment-variables指令：

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true"}'
```

請注意，當您使用put-graphql-api-environment-variables指令套用環境變數時，環境變數結構的內容會被覆寫；這表示現有的環境變數將會遺失。要在添加新環境變量時保留現有的環境變量，請在請求中包括所有現有的鍵值對以及新的鍵值對。使用上面的例子，如果你想添加"EMPTY":""，你可以做到以下幾點：

```
aws appsync put-graphql-api-environment-variables \
  --api-id "abcdefghijklmnopqrstuvwxyz" \
  --environment-variables '{"USER_TABLE":"users_prod","DEBUG":"true", "EMPTY":""}'
```

若要擷取目前的組態，請使用以下get-graphql-api-environment-variables指令：

```
aws appsync get-graphql-api-environment-variables --api-id "<api-id>"
```

使用上面的例子，您可以使用以下命令：

```
aws appsync get-graphql-api-environment-variables --api-id "abcdefghijklmnopqrstuvwxyz"
```

結果將顯示環境變量及其關鍵值的列表：

```
{
    "environmentVariables": {
        "USER_TABLE": "users_prod",
        "DEBUG": "true",
        "EMPTY": ""
    }
}
```

設定環境變數 (CFN)

您可以使用下面的模板創建環境變量：

```
AWSTemplateFormatVersion: 2010-09-09
Resources:
  GraphQLApiWithEnvVariables:
    Type: "AWS::AppSync::GraphQLApi"
    Properties:
      Name: "MyApiWithEnvVars"
      AuthenticationType: "AWS_IAM"
      EnvironmentVariables:
        EnvKey1: "non-empty"
        EnvKey2: ""
```

環境變數和合併的 API

在來源 API 中定義的環境變數也可以在合併的 API 中使用。合併 API 中的環境變數為唯讀且無法更新。請注意，您的環境變數金鑰在所有 Source API 中必須是唯一的，才能成功合併；重複的金鑰永遠會導致合併失敗。

擷取環境變數

要在函數代碼中檢索環境變量，請從解析器和函數中的`ctx.env`對象中檢索值。以下是一些實際操作的例子。

Publishing to Amazon SNS

在此範例中，我們的 HTTP 解析器會將訊息傳送至 Amazon SNS 主題。只有在定義 GraphQL API 的堆疊和主題部署完成後，才會知道主題的 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

在此範例中，如果針對測試部署 API 或已在生產環境中部署，DynamoDB 表的名稱會有所不同。解析器代碼不需要更改。環境變數的值會根據 API 的部署位置進行更新。

```
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
      },
    ],
  };
}
```

授權與驗證

本區段說明為您的應用程式設定安全性與資料保護的選項。

授權類型

您可以透過五種方式授權應用程式與 AWS AppSync GraphQL API 進行互動。您可以在 AWS AppSync API 或 CLI 呼叫中指定下列其中一個授權類型值，以指定所使用的授權類型：

- **API_KEY**

  使用 API 金鑰。

- **AWS_LAMBDA**

  用於使用 AWS Lambda 功能。

- **AWS_IAM**

  適用於使用 AWS Identity and Access Management (IAM) 許可。

- **OPENID_CONNECT**

  對於使用您的 OpenID Connect 提供者。

- **AMAZON_COGNITO_USER_POOLS**

  用於使用 Amazon Cognito 用戶池。

這些基本授權類型適用於大多數的開發人員。對於更進階的使用案例，您可以透過主控台、CLI 和 AWS CloudFormation. 對於其他授權模式， AWS AppSync 提供採用上述值的授權類型 (即API_KEYAWS_LAMBDA、AWS_IAM、OPENID_CONNECT、、和AMAZON_COGNITO_USER_POOLS)。

當您指定API_KEYAWS_LAMBDA、或AWS_IAM做為主要或預設授權類型時，您無法再將它們指定為其中一種額外的授權模式。同樣，您不能複製API_KEY，AWS_LAMBDA也不能在其他授權模式中AWS_IAM進行複製。您可以使用多個 Amazon Cognito 使用者集區和 OpenID Connect 供應商。但是，您無法在預設授權模式和任何其他授權模式之間使用複製的 Amazon Cognito 使用者集區或 OpenID Connect 提供者。您可以使用對應的組態規則運算式，為 Amazon Cognito 使用者集區或 OpenID Connect 提供者指定不同的用戶端。

API 金鑰授權

未驗證的 API 相較於已驗證 API 需要更嚴格的調節。其中一種控制未授權 GraphQL 端點調節的方法就是使用 API 金鑰。API 金鑰是應用程式中的硬式編碼值，由 AWS AppSync 服務在您建立未經驗證的 GraphQL 端點時產生。您可以從主控台、CLI 或 AWS AppSync API 參考輪換 API 金鑰。

Console

1. 登入 AWS Management Console 並開啟AppSync主控台。
   a. 在 API 儀表板中，選擇您的 GraphQL API。
   b. 在側邊欄中，選擇設定。
2. 在「預設授權模式」下，選擇「API 金鑰」。
3. 在「API 金鑰」表格中，選擇「新增 API 金鑰」。

   表格中會產生新的 API 金鑰。

   - 若要刪除舊的 API 金鑰，請在表格中選取 API 金鑰，然後選擇 [刪除]。
4. 選擇頁面底部的 Save (儲存)。

CLI

1. 如果您尚未這樣做，請設定您對 AWS CLI 的存取權限。如需詳細資訊，請參閱組態基本知識。
2. 透過執行`update-graphql-api`命令來建立 GraphQL API 物件。

   您需要為此特定命令輸入兩個參數：

   1. 您`api-id`的 GraphQL 應用程式介面。
   2. 您的 API `name` 的新功能。您可以使用相同的`name`。
   3. 的`authentication-type`，這將是`API_KEY`。

   > **Note**
   > 必須設定其他參數`Region`，例如，但通常會預設為 CLI 組態值。

範例命令可能如下所示：

```
aws appsync update-graphql-api --api-id abcdefghijklmnopqrstuvwxyz --name
 TestAPI --authentication-type API_KEY
```

輸出將在 CLI 中返回。以下是 JSON 中的一個示例：

```
{
    "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 金鑰可設定為最多 365 天，您可以從過期日開始再延展現有的過期日期的最多 365 天。用於開發目的或者需安全公開公用 API 時，建議使用 API 金鑰。

在用戶端上，由標題 `x-api-key` 指定 API 金鑰。

例如，如果您的 `API_KEY` 是 `'ABC123'`，您可以經由 `curl` 來傳送 GraphQL 查詢，如下所示：

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "x-api-key:ABC123" -d
 '{ "query": "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

AW_Lambda 授權

您可以使用 AWS Lambda 函數實現自己的 API 授權邏輯。您可以將 Lambda 函數用於主要或次要授權者，但每個 API 只能有一個 Lambda 授權函數。使用 Lambda 函數進行授權時，適用下列條件：

- 如果 API 啟用了`AWS_LAMBDA`和`AWS_IAM`授權模式，則無法將 Sigv4 簽名用作`AWS_LAMBDA`授權令牌。
- 如果 API 具有`AWS_LAMBDA`和`OPENID_CONNECT`授權模式或啟用了`AMAZON_COGNITO_USER_POOLS`授權模式，則 OIDC 令牌不能用作`AWS_LAMBDA`授權令牌。請注意，OIDC 令牌可以是承載計劃。
- Lambda 函數不得為解析器傳回超過 5MB 的上下文資料。

例如，如果您的授權令牌是`'ABC123'`，則可以通過 curl 發送 GraphQL 查詢，如下所示：

```
$ curl -XPOST -H "Content-Type:application/graphql" -H "Authorization:ABC123" -d
 '{ "query":
        "query { movies { id } }" }' https://YOURAPPSYNCENDPOINT/graphql
```

Lambda 函數在每個查詢或突變之前被調用。返回值可以根據 API ID 和身份驗證令牌進行緩存。默認情況下，緩存不打開，但可以在 API 級別或通過在函數的返回`ttlOverride`值中設置值來啟用緩存。

如果需要，可以指定在調用函數之前驗證授權令牌的正則表達式。這些正則表達式用於在調用函數之前驗證授權令牌是否具有正確的格式。使用與此正則表達式不匹配的令牌的任何請求都將被自動拒絕。

用於授權的 Lambda 函數需`appsync.amazonaws.com`要對其套用主體政策，以 AWS AppSync 允許呼叫它們。這個動作會在 AWS AppSync 主控台中自動完成； AWS AppSync 主控台不會移除原則。如需將政策附加至 Lambda 函數的詳細資訊，請參閱 AWS Lambda 開發人員指南中的資源型政策。

您指定的 Lambda 函數將收到具有以下形狀的事件：

```
{
    "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
    }
}
```

該event對象包含從應用程序客戶端發送到的請求中發送的標頭 AWS AppSync。

授權函數必須至少返回一個布爾值isAuthorized，指示請求是否被授權。 AWS AppSync 可辨識從 Lambda 授權函數傳回的下列金鑰：

函數清單

isAuthorized(布林值，必要)

布林值，指出中的值authorizationToken是否獲得對 GraphQL API 進行呼叫的授權。

如果此值為真，則會繼續執行 GraphQL API。如果這個值是假的，一個UnauthorizedException被提高

deniedFields（字符串列表，可選）

即使從解析器返回一個值null，其列表也會被強制更改為。

每個項目都是形式的完全合格欄位 ARN arn:aws:appsync:*us-east-1*:*111122223333*:apis/*GraphQLApiId*/types/*TypeName*/fields/*FieldName* 或簡短格式。*TypeName*.*FieldName*當兩個 API 共享一個 Lambda 函數授權器時，應使用完整的 ARN 表單，並且兩個 API 之間的常見類型和字段之間可能存在歧義。

resolverContext（JSON 對象，可選）

在解析器模板$ctx.identity.resolverContext中可見的 JSON 對象。例如，如果解析器返回以下結構：

```
{
  "isAuthorized":true
  "resolverContext": {
    "banana":"very yellow",
    "apple":"very green"
  }
```

```
}
```

解析程式範本`ctx.identity.resolverContext.apple`中的值將是 "" `very green`。該對`resolverContext`象僅支持鍵值對。不支援巢狀索引鍵。

> **Warning**
>
> 此 JSON 物件的總大小不得超過 5MB。

`ttlOverride`（整數，可選）

應快取回應的秒數。如果未傳回任何值，則會使用 API 的值。如果這是 0，則不會緩存響應。

Lambda 授權者的逾時時間為 10 秒。我們建議您設計函數，以盡可能在最短的時間內執行，以擴展 API 的效能。

多個 AWS AppSync API 可以共用單一驗證 Lambda 函數。不允許跨帳戶授權者使用。

在多個 API 之間共用授權函式時，請注意，短格式欄位名稱 (*typename*.*fieldname*) 可能會無意間隱藏欄位。若要消除中某個欄位的歧義`deniedFields`，您可以以的形式指定明確的欄位 ARN。`arn:aws:appsync:`*region*`:`*accountId*`:apis/`*GraphQLApiId*`/types/`*typeName*`/fields/`*fieldName*

若要在中新增 Lambda 函數做為預設授權模式 AWS AppSync：

Console

1. 登入主 AWS AppSync 控台並瀏覽至您要更新的 API。
2. 導航到 API 的「設置」頁面。

   將 API 層級授權變更為。AWS Lambda
3. 選擇要對其進行 API 呼叫授權的 AWS 區域 和 Lambda ARN。

   > **Note**
   >
   > 系統會自動新增適當的主體政策， AWS AppSync 讓您呼叫 Lambda 函數。
4. 選擇性地設定回應 TTL 和權杖驗證規則運算式。

AWS CLI

1. 將下列原則附加至正在使用的 Lambda 函數：

```
aws lambda add-permission --function-name "my-function" --statement-id "appsync"
 --principal appsync.amazonaws.com --action lambda:InvokeFunction --output text
```

> **Important**
>
> 如果您希望將函數的政策鎖定為單一 GraphQL API，則可以執行以下命令：
>
> ```
> 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 以使用給定的 Lambda 函數 ARN 作為授權者：

```
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**
>
> 您也可以包含其他組態選項，例如 Token 規則運算式。

下列範例說明 Lambda 函數，該函數示範 Lambda 函數用作 AWS AppSync 授權機制時可能具有的各種驗證和失敗狀態：

```
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 {}
```

規避簽名 V4 和 OIDC 令牌授權限制

當啟用某些授權模式時，可以使用以下方法來避免無法使用您的 SIGv4 簽名或 OIDC 權杖做為 Lambda 授權權杖的問題。

如果您想要在為 API 啟用`AWS_IAM`和授權模式時使用 Sigv4 簽章做為 Lambda `AWS_LAMBDA` 授權權杖，請執行下列動作： AWS AppSync

- 若要建立新的 Lambda 授權權杖，請在 Sigv4 簽章中新增隨機字尾和/或前置字元。
- 若要擷取原始 Sigv4 簽章，請移除 Lambda 授權權杖中的隨機前置詞和/或尾碼，以更新 Lambda 函數。然後，使用原始的 Sigv4 簽名進行身份驗證。

如果您想要在為 API 啟用授權模式或AMAZON_COGNITO_USER_POOLS和授權模式時使用 OIDC OPENID_CONNECT 權杖做為 Lambda AWS_LAMBDA 授權權杖，請執行以下操作： AWS AppSync

- 若要建立新的 Lambda 授權權杖，請在 OIDC 權杖中新增隨機字尾和/或前置字元。Lambda 授權令牌不應包含承載方案前綴。
- 若要擷取原始 OIDC 權杖，請移除 Lambda 授權權杖中的隨機字首和/或尾碼，以更新 Lambda 函數。然後，使用原始的 OIDC 令牌進行身份驗證。

IAM 授權

此授權類型會在 GraphQL API 上強制執行簽AWS 章版本 4 簽署程序。您可以將身分與存取管理 (IAM) 的存取政策與此授權類型建立關聯。您的應用程式可以使用存取金鑰 (由存取金鑰 ID 和秘密存取金鑰組成)，或使用 Amazon Cognito 聯合身分提供的短期暫時登入資料，利用此關聯性。

如果您希望角色擁有執行所有資料操作的存取權限：

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Effect": "Allow",
         "Action": [
            "appsync:GraphQL"
         ],
         "Resource": [
            "arn:aws:appsync:us-west-2:123456789012:apis/YourGraphQLApiId/*"
         ]
      }
   ]
}
```

您可以YourGraphQLApiId從主 AppSync控台的主要 API 清單頁面，直接在 API 名稱下找到。或者，您可以使用 CLI 來取得該資訊：`aws appsync list-graphql-apis`

如果您想要僅限制對於特定 GraphQL 操作的存取權限，您可以在根 `Query`、`Mutation` 以及 `Subscription` 欄位執行此操作。

```
{
   "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，假設您有以下結構描述且您想要限制取得所有文章的存取：

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

type Query {
   posts:[Post!]!
}

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

角色的對應 IAM 政策 (例如，您可以附加至 Amazon Cognito 身分集區) 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"
            ]
        }
    ]
}
```

開放連接授權

此授權類型會強制執行 OIDC 相容服務所提供的 OpenID 連線 (OIDC) 權杖。您的應用程式可以運用由 OIDC 提供者定義的使用者與權限來控制存取權限。

發行者 URL 是您提供給的唯一必要組態值 AWS AppSync (例如，`https://auth.example.com`)。此網址必須可透過 HTTPS 進行定址。 AWS AppSync 附加`/.well-known/openid-configuration`到發行者網址，並`https://auth.example.com/.well-known/openid-configuration`根據 OpenID Connect 發現規範找到 Open ID 配置。它預期會在此 URL 擷取相容於 RFC5785 的 JSON 文件。這個 JSON 文檔必須包含一個`jwks_uri`密鑰，該密鑰指向帶有簽名密鑰的 JSON Web 密鑰集（JWKS）文檔。 AWS AppSync 要求 JWKS 包含和的 `kty` JSON 欄位。`kid`

AWS AppSync 支持廣泛的簽名算法。

簽署演算法
RS256
RS384
RS512
PS256
PS384
PS512
HS256
HS384

簽署演算法
HS512
ES256
ES384
ES512

我們建議您使用 RSA 演算法。由提供者發行的字符必須包含字符發行的時間 (`iat`)，且可包含驗證時的時間 (`auth_time`)。您可以提供 OpenID Connect 組態中的發行時間的 TTL 值 (`iatTTL`) 和身分驗證時間 (`authTTL`)，以供額外驗證。如果您的提供者授權多個應用程式，您也可以提供用於根據用戶端 ID 授權的一般表達式 (`clientId`)。當 OpenID Connect 配置中存在時，通過要求與令牌中的`aud`或`azp`聲明匹配`clientId`來 AWS AppSync 驗證聲明。`clientId`

若要驗證多個用戶端 ID，請使用管線運算子 (「|」)，它是規則運算式中的「or」。例如，如果您的 OIDC 應用程式有四個具有用戶端識別碼 (例如 0A1S2D、1F4G9H、1J6L4B、6GS5MG) 的用戶端，以便僅驗證前三個用戶端識別碼，您可以在用戶端識別碼欄位中放置 1F4G9H|1J6L4B|6GS5MG。

亞馬遜用戶池授權

此授權類型會強制執行 Amazon Cognito 使用者集區提供的 OIDC 權杖。您的應用程式可以利用來自其他 AWS 帳戶的使用者集區和使用者集區中的使用者和群組，並將這些使用者和群組與 GraphQL 欄位關聯以控制存取。

使用 Amazon Cognito 使用者集區時，您可以建立使用者所屬的群組。此資訊會在傳送 GraphQL 作業時，以應用程式在授權標頭 AWS AppSync 中傳送至的 JWT 權杖編碼。您可以使用結構描述上的 GraphQL 指令來控制哪些群組可以叫用欄位上的哪些解析程式，進而為您的客戶提供更具控制性的存取權限。

例如，假設您有下列 GraphQL 結構描述：

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

type Query {
```

```
   posts:[Post!]!
}

type Mutation {
   addPost(id:ID!, title:String!):Post!
}
...
```

如果您在 Amazon Cognito 用戶池中有兩個組-博客作者和讀者-並且您想要限制讀者，以便他們無法添加新條目，那麼您的模式應該如下所示：

```
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"])
}
...
```

請注意，如果您想要預設使用特定存取策略，可以省 grant-or-deny 略該`@aws_auth`指令。當您透過主控台或透過下列 CLI 命令建立 GraphQL API 時，您可以在使用者集區設定中指定 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"}'
```

使用其他授權模式

當您新增其他授權模式時，您可以直接在 AWS AppSync GraphQL API 層級設定授權設定 (也就是您可以直接在`GraphqlApi`物件上設定的`authenticationType`欄位) 上設定授權設定，並做為結構描述的預設值。這表示沒有特定指示詞的任何類型必須傳遞 API 層級授權設定。

在結構描述層級上，您可以在結構描述上使用指示詞來指定其他授權模式。您可以在結構描述中的個別欄位上指定授權模式。例如，針對 API_KEY 授權，您可以在結構描述物件類型定義/欄位上使用 @aws_api_key。結構描述欄位和物件類型定義支援下列指示詞：

- @aws_api_key - 指定欄位會經由 API_KEY 授權。
- @aws_iam - 指定欄位會經由 AWS_IAM 授權。
- @aws_oidc - 指定欄位會經由 OPENID_CONNECT 授權。
- @aws_cognito_user_pools - 指定欄位會經由 AMAZON_COGNITO_USER_POOLS 授權。
- @aws_lambda - 指定欄位會經由 AWS_LAMBDA 授權。

您無法將 @aws_auth 指示詞搭配使用其他授權模式。@aws_auth 只能在沒有其他授權模式的 AMAZON_COGNITO_USER_POOLS 授權內容中運作。但是，您可以搭配相同的引數，使用 @aws_cognito_user_pools 指示詞來取代 @aws_auth 指示詞。這兩者的主要差異是您可以在任何欄位和物件類型定義上指定 @aws_cognito_user_pools。

為了了解其他授權模式如何運作以及如何在結構描述上指定這些模式，讓我們看看以下結構描述：

```
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!
}
...
```

對於此結構描述，假設這AWS_IAM是 AWS AppSync GraphQL API 上的預設授權類型。這表示沒有指示詞的欄位會使用 AWS_IAM 進行保護。例如，Query 類型的 getPost 欄位就是這種情況。結構描述指示詞可讓您使用超過一種授權模式。例如，您可以在 AWS AppSync GraphQL API 上API_KEY設定為其他授權模式，並且可以使用@aws_api_key指示詞來標記欄位 (例如，getAllPosts在本範例中)。指示詞會在欄位層級運作，因此您也需要將 API_KEY 存取權限提供給 Post 類型。您可以透過在 Post 類型中使用指示詞標記每個欄位，或是使用 @aws_api_key 指示詞標記 Post 類型，來執行此作業。

若要進一步限制對 Post 類型中欄位的存取，您可以針對 Post 類型中的個別欄位使用指示詞，如下所示。

例如，您可以將 restrictedContent 欄位新增到 Post 類型，並使用 @aws_iam 指示詞限制對該項目進行存取。AWS_IAM 已驗證請求可以存取 restrictedContent，但 API_KEY 請求將無法存取它。

```
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
}
...
```

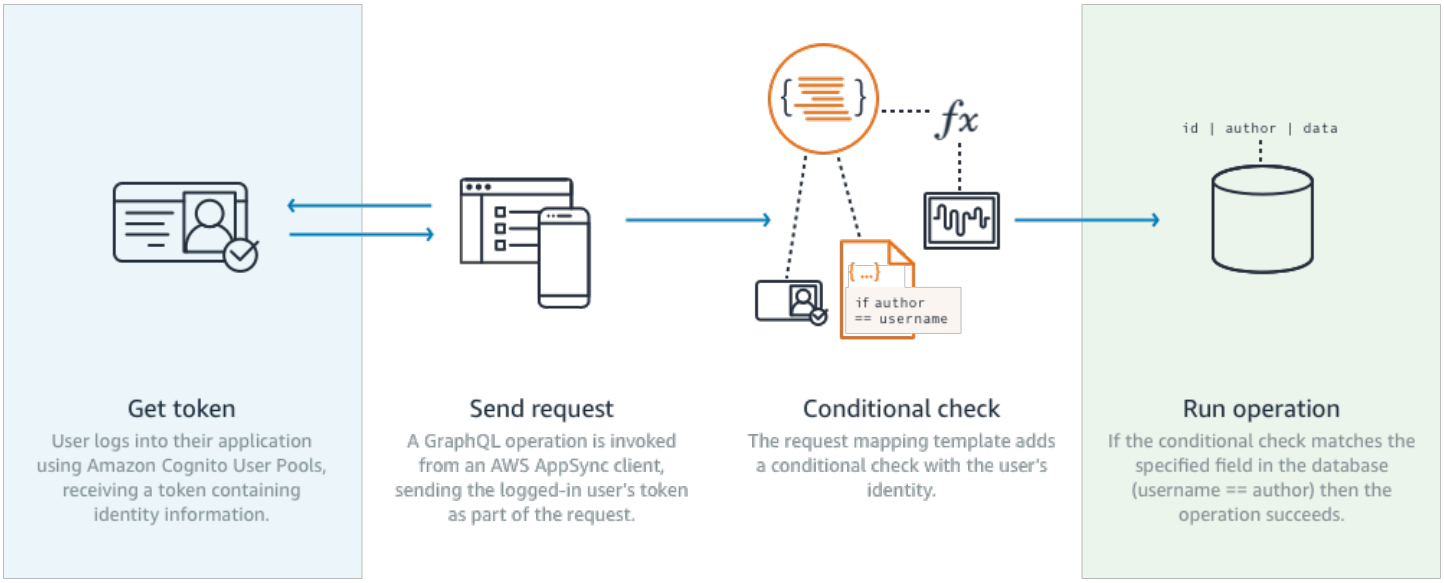
精細定義存取控制

上述資訊示範了如何限制存取權限，或將存取權限授予特定 GraphQL 欄位。若您希望根據特定條件設定資料的存取控制 (例如根據進行呼叫的使用者，以及使用者是否擁有資料)，您可以在解析程式中使用映射範本。您也可以執行更複雜的商業邏輯；我們會在篩選資訊中進行說明。

本節說明如何使用 DynamoDB 解析器對應範本設定資料的存取控制。

在繼續進行任何操作之前，如果您不熟悉中的對應範本 AWS AppSync，您可能需要檢閱 DynamoDB 的解析器對映範本參考和解析器對映範本參考。

在下列使用 DynamoDB 的範例中，假設您使用先前的部落格文章結構描述，且只有建立貼文的使用者才能編輯該貼文。評估程序需使用者獲得應用程式的登入資料，例如使用 Amazon Cognito 使用者集區，然後將這些登入資料做為 GraphQL 操作的一部分。映射範本接著將取代在條件式陳述式中的登入資料值 (例如使用者名稱)，此資訊接著將與資料庫中的值進行比較。



若要新增此功能，請新增 `editPost` 的 GraphQL 欄位，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 的解析程式映射範本 (顯示於本區段末端的範例中) 需要針對您的資料存放區執行邏輯檢查，只允許建立文章的使用者可對其進行編輯。由於這是一項編輯作業，因此它對應於 DynamoDB UpdateItem 中的一個。在執行此動作前可以使用透過使用者身分驗證傳遞的上下文來執行條件檢查。此資訊存放在 Identity 物件中，有以下值：

```
{
   "accountId" : "12321434323",
   "cognitoIdentityPoolId" : "",
   "cognitoIdentityId" : "",
   "sourceIP" : "",
   "caller" : "ThisistheprincipalARN",
   "username" : "username",
   "userArn" : "Sameasabove"
}
```

若要在 DynamoDB UpdateItem 呼叫中使用此物件，您需要將使用者識別資訊儲存在表格中以進行比較。首先，您的 addPost 變動需要儲存建立者。其次，您的 editPost 變動需在更新前執行條件式檢查。

以下是將使用者身分儲存為資料行的解析程式程式碼範例Author：addPost

```
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;
```

請注意，Author 屬性從來自於應用程式的 Identity 物件發布。

最後，以下是解析器代碼的示例editPost，如果請求來自創建帖子的用戶，該代碼僅更新博客文章的內容：

```
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;
```

這個範例會使用覆PutItem寫所有值而非覆寫值UpdateItem，但同樣的概念適用於condition陳述式區塊。

篩選資訊

在某些情況下，您無法控制資料來源的回應，但又不想在成功寫入或讀取資料來源時，將不必要的資訊傳送給用戶端。如果發生這些情況，您可以使用回應映射範本來篩選資訊。

例如，假設您的部落格文章 DynamoDB 表格上沒有適當的索引 (例如上Author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;
}
```

即使調用者不是創建帖子的作者，請求處理程序也會獲取該項目。為了防止傳回所有資料，回應處理常式會檢查以確定呼叫者與項目的作者相符。如果發起人不符合此檢查，只會傳回 nul在某些情況下，您無法控制數據源的響應，但又不想在成功寫入或讀取數據源時將不必要的信息發送給客戶端。l 回應。

資料來源存取

AWS AppSync 使用身分識別和存取管理 (IAM) 角色和存取政策與資料來源通訊。如果您使用現有的角色，則需要新增信任原則才能擔任該角色。 AWS AppSync 信任關係如下所示：

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

縮小角色的存取政策範圍，使其只能對必要的最少資源集合採取行動，是非常重要的。使用 AppSync 主控台建立資料來源並建立角色時，會自動為您完成此操作。不過，當使用 IAM 主控台的內建範例範本在主控台外部建立角色時，不會自動縮減資源的許可範圍，您應該在將應用程式移至生產環境之前執行此動作。 AWS AppSync

授權使用案例

您在安全章節了解保護 API 的各種授權模式，其中也簡介精細分級授權機制，以了解各項概念與流程。由於可AWS AppSync 讓您透過使用 GraphQL 解析器對應範本對資料執行邏輯完整作業，因此您可以結合使用者身分識別、條件式和資料注入，以非常靈活的方式保護讀取或寫入的資料。

如果您對於編輯 AWS AppSync 解析程式不熟悉，請檢閱程式設計指南。

概要

傳統上，授與系統中資料的存取權是透過存取控制矩陣來完成，其中列 (資源) 與欄 (使用者/角色) 的交集是授與的權限。

AWS AppSync 將您自己帳戶中的資源和線程身份（用戶/角色）信息用於 GraphQL 請求和響應作為上下文對象，您可以在解析器中使用該對象。這表示您可依據解析程式邏輯，將權限適當地授與寫入或讀取操作。如果此邏輯位於資源層級，例如只有特定的具名使用者或群組可以讀取/寫入特定資料庫資料列，則必須儲存該「授權中繼資料」。AWS AppSync 不存儲任何數據，因此您必須將此授權元數據與資源一起存儲，以便可以計算權限。授權中繼資料通常是 DynamoDB 表之中的屬性 (欄)，例如擁有者或使用者/群組清單。例如其中可能有讀取器及寫入器屬性。

大致而言，這表示若您由資料來源讀取個別項目，就會在解析程式由資料來源讀取之後，於回應範本執行條件式 `#if () ... #end` 陳述式。檢查通常會使用 `$context.identity` 之中的使用者或群組值，依據讀取操作傳回的授權中繼資料進行成員檢查。對於多筆記錄，例如由資料表 `Scan` 或 `Query` 傳回的清單，您將使用類似的使用者或群組值，將條件檢查做為操作的一部分傳送至資料來源。

同樣地，寫入資料時，您會將條件式陳述式套用至動作 (例如 `PutItem` 或 `UpdateItem`)，了解進行變動的使用者或群組是否具有許可。條件式將再次多次使用 `$context.identity` 之中的值，與該項資源的授權中繼資料進行比較。對於要求及回應範本，您也可使用用戶端的自訂標頭執行驗證檢查。

讀取資料

如前所述，執行檢查的授權中繼資料必須與資源儲存，或傳送至 GraphQL 要求 (身分、標頭等等)。為了示範，假設您有下列的 DynamoDB 表格：

ID	Data	PeopleCanAccess	GroupsCanAccess	Owner
123	{my: data,...}	[Mary, Joe]	[Admins, Editors]	Nadia

主索引鍵為 `id`，要存取的資料為 `Data`。其他資料行是您可以針對授權執行的檢查範例。 `Owner`會有一`String`段時間，`PeopleCanAccess`並且`GroupsCanAccess`將`String Sets`如 DynamoDB 的解析器映射模板參考中所述。

在解析程式映射範本概觀之中的圖會顯示回應範本不僅包含內容物件，也包含資料來源結果。對個別項目的 GraphQL 查詢而言，您可使用回應範本檢查使用者是否允許檢視這些結果，否則將傳回授權錯誤訊息。這有時稱為「授權篩選」。對使用掃描或查詢的 GraphQL 查詢傳回清單而言，效能較高的作法是在要求範本執行檢查，並僅於滿足授權條件時傳回資料。實作之後將：

1. GetItem -個人記錄的授權檢查。使用 `#if() ... #end` 陳述式完成。
2. 掃描/查詢操作 - 授權檢查為 `"filter":{"expression":...}` 陳述式。一般檢查為對等 (`attribute = :input`) 或檢查值是否在清單之中 (`contains(attribute, :input)`)。

在 #2 之中，兩種陳述式的 attribute 都代表表格之中記錄的欄名稱，例如以上範例中的 Owner。您可利用 # 簽署和使用 "expressionNames":{...} 為此設定別名，但並非必要。:input 是您要與資料庫屬性比較值的參考，該屬性將於 "expressionValues":{...} 定義。您可參考以下各項範例。

使用案例：所有者可以閱讀

使用以上表格，若您只想傳回資料，了解 Owner == Nadia 是否用於個別讀取操作 (GetItem)，您的範本可能會像這樣：

```
#if($context.result["Owner"] == $context.identity.username)
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

這裡提出兩點注意事項，在剩餘章節中會用到。首先，如果使用 Amazon Cognito 使用者集區，則檢查會使用$context.identity.username這個名稱是易於使用的使用者註冊名稱，如果使用 IAM (包括 Amazon Cognito 聯合身分)，則會成為使用者身分。要為擁有者存放其他值，例如唯一的「Amazon Cognito 身分」值，這在聯合多個位置登入時非常有用，因此您應該檢閱解析器對應範本內容參考中可用的選項。

第二，以 $util.unauthorized() 進行的條件式否則檢查回應為完全可選用的選項，但建議做為設計 GraphQL API 時的最佳實務。

用例：硬編碼特定訪問

```
// 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：篩選結果清單

在前一項範例之中，您可在 $context.result 傳回單一項目時對其執行檢查，不過像是掃描等部分操作，將在 $context.result.items 傳回多個項目；您在此需要執行驗證篩選，並僅傳回使用者允許檢視的結果。假設此Owner欄位在記錄上設定了 Amazon Cognito IdentityID，接著您可以使用下列回應對應範本進行篩選，以便僅顯示使用者擁有的記錄：

```
#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：多人可以閱讀

另一種熱門的授權選擇，就是允許一群人讀取資料。在以下範例中，"filter": {"expression":...} 僅傳回表格掃描的值 (若使用者執行的 GraphQL 查詢列於 PeopleCanAccess 集之中)。

```
{
    "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：群組可以讀取

類似於上一項使用案例，可能只有單一或更多群組的人，有權讀取資料庫之中的特定項目。"expression": "contains()" 操作的使用方式類似，不過這是所有群組的邏輯 OR，使用者可能是其中的一部分，需要在設定成員資格時加以考量。在此案例中，我們針對使用者所屬的各個群組建立以下 $expression 陳述式，然後將其傳送至篩選器：

```
#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)
    }
}
```

寫入資料

在變動寫入資料一定是在要求映射範本進行控制。就 DynamoDB 資料來源而言，關鍵在於使用適當的 `"condition":{"expression"...}"`，依據該表格之中的授權中繼資料執行驗證。在安全性中，我們提供範例讓您用來檢查資料表中的 `Author` 欄位。本節使用案例將探索更多使用案例。

使用案例：多個擁有者

使用之前的範例表格圖示，假設 `PeopleCanAccess` 清單

```
{
    "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)
        }
    }
}
```

用例：組可以創建新記錄

```
#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：群組可以更新現有記錄

```
#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)
    }
}
```

公有和私有日期

您也可以使用條件式篩選器，選擇將資料標示為公開、私人或其他布林檢查。並於之後合併為回應範本之中授權篩選的一部分。此項檢查適合用於暫時隱藏資料，或由檢視之中移除資料，不必嘗試控制群組成員資格。

例如，假設您在名為 public 的 DynamoDB 表格，以 yes 或 no 值在各個項目新增屬性。下列回應範本可於 GetItem 呼叫時使用；只有在使用者位於具有權限的群組之中，且該資料標示為公開的情況下，資料才會顯示：

```
#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
```

以上程式碼也可使用邏輯 OR (||)，在使用者擁有權限或記錄為公開的情況下能夠讀取記錄：

```
#if($hasPermission || $context.result.public == 'yes')
    $utils.toJson($context.result)
#else
    $utils.unauthorized()
#end
```

一般而言，您在執行授權檢查時，運算子 ==、!=、&& 及 || 都相當實用。

即時日日

您可在用戶端訂閱時，使用本文之前介紹的相同技巧，將精細分級的存取控制套用至 GraphQL 訂閱。您將解析程式附加至訂閱欄位，在此由資料來源查詢資料，並於要求或回應映射範本時執行條件式邏輯。只要資料結構符合 GraphQL 訂閱之中傳回的類型，您也可以傳回其他資料給用戶端，例如訂閱的初始結果。

使用案例：使用者只能訂閱特定的對話

這是常見的即時資料使用案例，其中 GraphQL 訂閱正在建構訊息或私人交談應用程式。建立具有多位使用者的交談應用程式時，可能在兩位或多位使用者之間產生對話。這些使用者可能會聚集成為私人或公開的「聊天室」。在這種情況下，您可能只想授權使用者訂閱其有權存取的對話 (一對一或群組之中)。以下範例示範單一使用者向他人傳送私人訊息的簡單使用案例。此安裝程式有兩個 Amazon DynamoDB 資料表：

- 訊息表格：(主索引鍵) `toUser`、(排序索引鍵) `id`
- 權限表格：(主索引鍵) `username`

訊息表格儲存透過 GraphQL 變動傳送的實際訊息。權限表格會由 GraphQL 訂閱檢查，在用戶端連線時進行授權。以下範例假設您使用下列 GraphQL 結構描述：

```
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
}
```

下面未涵蓋某些標準操作`createUserPermissions()`，例如，以說明訂閱解析器，而是 DynamoDB 解析器的標準實作。我們將重點放在訂閱授權流程與解析程式。使用下列要求範本，讓使用者將訊息傳送給其他使用者，將解析程式連接至 `sendMessage()` 欄位，然後選取 Messages (訊息) 資料表資料來源：

```
{
    "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),
    }
}
```

在此範例中，我們使用 `$context.identity.username`. 這會傳回AWS Identity and Access Management或 Amazon Cognito 使用者的使用者資訊。回應範本為簡單的 `$util.toJson($ctx.result)` 傳遞。儲存後回到結構描述頁面。然後連接解析程式以用於 `newMessage()` 訂閱，使用許可資料表做為資料來源，以及下列要求映射範本：

```
{
    "version": "2018-05-29",
    "operation": "GetItem",
    "key": {
        "username": $util.dynamodb.toDynamoDBJson($ctx.identity.username),
    },
}
```

之後請使用下列回應映射範本，以許可資料表的資料執行授權檢查：

```
#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
```

在此情況下，您會進行三項授權檢查。第一項檢查確保結果已返回。第二項檢查確保使用者並未訂閱其他人的訊息。第三個可確保允許使用者訂閱任何欄位，方法是檢查`isAuthorizedForSubscriptions`儲存為 a 的 DynamoDB 屬性`B00L`。

若要測試項目，您可以使用 Amazon Cognito 使用者集區和名為「Nadia」的使用者登入AWS AppSync 主控台，然後執行下列 GraphQL 訂閱：

```
subscription AuthorizedSubscription {
    newMessage(toUser: "Nadia") {
        id
        toUser
        fromUser
        content
    }
}
```

如果在許可資料表之中，存在 `Nadia` 的 `username` 索引鍵屬性記錄，且 `isAuthorizedForSubscriptions` 設定為 `true`，您就可以獲得成功回應。如果您在以上的 `username` 查詢嘗試不同 `newMessage()`，就會傳回錯誤。

使用 AWS WAF 來保護您的 API

AWS WAF 是一個 Web 應用程式防火牆，有助於保護 Web 應用程式和 API 免遭攻擊。它可讓您設定一組稱為 Web 存取控制清單 (Web ACL) 的規則，根據您定義的可自訂 Web 安全規則和條件來允許、封鎖或監視 (計數) Web 要求。當您整合AWS AppSync使用 APIAWS WAF，您可以對 API 接受的 HTTP 流量獲得更多控制和可見性。若要進一步瞭解AWS WAF，請參閱如何AWS WAF作品在AWS WAF開發人員指南。

您可以使用 AWS WAF 來保護 AppSync API 免於常見的網頁入侵，例如 SQL 插入和跨網站指令碼 (XSS) 攻擊。這些可能會影響 API 可用性和效能、危及安全性，或耗用過多的資源。例如，您可以建立規則，允許或封鎖來自指定 IP 地址、來自 CIDR 區塊的請求，或源自特定國家或區域，其中包含惡意 SQL 程式碼或惡意指令碼的請求。

您也可以建立規則，以符合在 HTTP 標頭、方法、查詢字串、URI 和請求本文 (限於前 8 KB) 的指定字串或常規表達式模式。此外，您可以建立規則以封鎖來自特定使用者代理程式、惡意機器人和內容抓取器的攻擊。例如，您可以使用以速率為基礎的規則，以指定每個用戶端 IP 在尾隨、持續更新的 5 分鐘期間，允許的 Web 請求數。

若要深入瞭解支援的規則類型及其他規則AWS WAF功能，請參閱AWS WAF開發者指南和AWS WAFAPI 參考資料。

> **Important**
>
> AWS WAF 是您對抗網路攻擊的第一條防線。何時AWS WAF已在 API 上啟用，AWS WAF規則會先於其他存取控制功能 (例如 API 金鑰授權、IAM 政策、OIDC 權杖和 Amazon Cognito 使用者集區) 進行評估。

整合AppSync使用 APIAWS WAF

您可以將應用程序同步 API 與AWS WAF使用AWS Management Console，該AWS CLI,AWS CloudFormation，或任何其他兼容的客戶端。

若要整合AWS AppSync使用 APIAWS WAF

1. 創建一個AWS WAF網絡交叉韌帶。有關使用的詳細步驟AWS WAF控制台，請參閱建立網路 ACL。
2. 定義網頁 ACL 的規則。一個或多個規則是在建立 Web ACL 的過程中定義的。如需有關如何建構規則的資訊，請參閱AWS WAF規則。有關您可以為您定義的有用規則的示例AWS AppSyncAPI，請參閱建立網頁 ACL 的規則。
3. 將網頁 ACL 與AWS AppSyncAPI。您可以在AWS WAF控制台或在AppSync控制台。
   - 將網 ACL 與AWS AppSync應用程式介面中AWS WAF控制台，請按照說明進行操作關聯或取消 Web ACL 與AWS資源在AWS WAF開發人員指南。
   - 將網 ACL 與AWS AppSync應用程式介面中AWS AppSync控制台
     a. 登入AWS Management Console並打開AppSync控制台。
     b. 選擇您要與網頁 ACL 建立關聯的 API。
     c. 在導覽窗格中，選擇 Settings (設定)。
     d. 在Web 應用防火牆區段，開啟啟用AWS WAF。
     e. 在十字韌帶下拉式清單中，選擇要與您的 API 建立關聯的網頁 ACL 名稱。
     f. 選擇儲存將網路 ACL 與您的 API 建立關聯。

> **Note**
>
> 在中建立網頁 ACL 之後AWS WAF主控台中，可能需要幾分鐘的時間才能使用新的 Web ACL。如果在中沒有看到新建立的網頁 ACLWeb 應用防火牆」功能表中，請等待幾分鐘，然後重試這些步驟，將 Web ACL 與您的 API 建立關聯。

> **Note**
>
> AWS WAF整合僅支援`Subscription registration message`即時端點的事件。AWS AppSync將回應錯誤訊息，而不是`start_ack`任何消息`Subscription registration message`被阻止AWS WAF。

將網頁 ACL 與AWS AppSyncAPI 中，您將會使用AWS WAFAPI。您不需要重新關聯網 ACL 與AWS AppSyncAPI，除非您想要關聯AWS AppSync具有不同網頁 ACL 的 API。

建立網頁 ACL 的規則

規則定義如何檢查 Web 請求，以及當 Web 請求符合檢查條件時該如何處理。規則不會自行存在於 AWS WAF。您可以在規則群組或定義規則的 Web ACL 中，依名稱存取規則。如需詳細資訊，請參閱AWS WAF規則。下列範例示範如何定義及關聯對於保護AppSyncAPI。

Example 用於限制請求主體大小的 Web ACL 規則

以下是限制要求主體大小的規則範例。這將被輸入到規則編輯器當在中建立網路 ACL 時AWS WAF控制台。

```
{
    "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
        }
}
```

使用上述範例規則建立 Web ACL 之後，您必須將其與您的AppSyncAPI。作為使用的替代方法AWS Management Console，您可以執行此步驟AWS CLI通過運行以下命令。

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

變更可能需要幾分鐘的時間才會傳播，但執行此命令之後，包含大於 1024 位元組之主體的要求將會遭到拒絕AWS AppSync。

> **Note**
>
> 在中建立新的網頁 ACL 之後AWS WAF主控台中，可能需要幾分鐘的時間才能讓 Web ACL 與 API 建立關聯。如果您執行 CLI 命令並取得`WAFUnavailableEntityException`錯誤，請等待幾分鐘，然後重試執行命令。

Example 用於限制來自單一 IP 位址的要求的 Web ACL 規則

以下是節流規則的範例AppSync從單一 IP 位址傳送 100 個要求的 API。這將被輸入到規則編輯器使用以速率為基礎的規則建立 Web ACL 時AWS WAF控制台。

```
{
  "Name": "Throttle",
  "Priority": 0,
  "Action": {
    "Block": {}
```

```
  },
  "VisibilityConfig": {
    "SampledRequestsEnabled": true,
    "CloudWatchMetricsEnabled": true,
    "MetricName": "Throttle"
  },
  "Statement": {
    "RateBasedStatement": {
      "Limit": 100,
      "AggregateKeyType": "IP"
    }
  }
}
```

使用上述範例規則建立 Web ACL 之後，您必須將其與您的AppSyncAPI。您可以在AWS CLI通過運行以下命令。

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

Example 網絡 ACL 規則，以防止對 API 進行圖形 QL __ 架構內省查詢

以下是防止對 API 進行 GraphQL __ 結構描述內部檢查查詢的規則範例。任何包含字串「__ 結構描述」的 HTTP 主體都會遭到封鎖。這將被輸入到規則編輯器當在中建立網路 ACL 時AWS WAF控制台。

```
{
  "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
        }
      ]
    }
  }
}
```

使用上述範例規則建立 Web ACL 之後，您必須將其與您的AppSyncAPI。您可以在AWS CLI通過運行以下命令。

```
aws waf associate-web-acl --web-acl-id waf-web-acl-arn --resource-arn appsync-api-arn
```

中的安全性 AWS AppSync

雲安全 AWS 是最高的優先級。身為 AWS 客戶，您可以從資料中心和網路架構中獲益，這些架構是為了滿足對安全性最敏感的組織的需求而建置的。

安全是 AWS 與您之間共同承擔的責任。共同責任模型將其描述為雲端的安全性和雲端中的安全性：

- 雲端的安全性 — AWS 負責保護在 AWS 雲端中執行 AWS 服務的基礎架構。 AWS 還為您提供可以安全使用的服務。若要深入瞭解適用於的規範遵循計劃 AWS AppSync，請參閱合規計劃的AWS 服務範圍範圍)。
- 雲端中的安全性 — 您的責任取決於您使用的 AWS 服務。您也必須對其他因素負責，包括資料的機密性、您公司的要求和適用法律和法規。

本文件可協助您瞭解如何在使用時套用共同責任模型 AWS AppSync。下列主題說明如何設定 AWS AppSync 以符合安全性與合規性目標。您還將學習如何使用其他 AWS 服務來幫助您監控和保護您的 AWS AppSync 資源。

主題

- 資料保護 AWS AppSync
- 符合性驗證 AWS AppSync
- 基礎架構安全性 AWS AppSync
- 韌性在 AWS AppSync
- 的身分識別與存取管理 AWS AppSync
- 使用記錄 AWS AppSync API 呼叫 AWS CloudTrail
- 安全性最佳做法 AWS AppSync

資料保護 AWS AppSync

AWS 共用責任模型適用於中的資料保護 AWS AppSync。如此模型中所述， AWS 負責保護執行所有 AWS 雲端. 您負責維護在此基礎設施上託管內容的控制權。您也同時負責所使用 AWS 服務 的安全組態和管理任務。如需資料隱私權的詳細資訊，請參閱資料隱私權常見問答集。如需有關歐洲資料保護的相關資訊，請參閱 AWS 安全性部落格上的 AWS 共同的責任模型和 GDPR 部落格文章。

基於資料保護目的，我們建議您使用 AWS IAM Identity Center 或 AWS Identity and Access Management (IAM) 保護 AWS 帳戶 登入資料並設定個別使用者。如此一來，每個使用者都只會獲得授與完成其任務所必須的許可。我們也建議您採用下列方式保護資料：

- 每個帳戶均要使用多重要素驗證 (MFA)。
- 使用 SSL/TLS 與 AWS 資源進行通訊。我們需要 TLS 1.2 並建議使用 TLS 1.3。
- 使用設定 API 和使用者活動記錄 AWS CloudTrail。
- 使用 AWS 加密解決方案，以及其中的所有默認安全控制 AWS 服務。
- 使用進階的受管安全服務 (例如 Amazon Macie)，協助探索和保護儲存在 Amazon S3 的敏感資料。
- 如果您在透過命令列介面或 API 存取時需要經 AWS 過 FIPS 140-2 驗證的加密模組，請使用 FIPS 端點。如需有關 FIPS 和 FIPS 端點的更多相關資訊，請參閱聯邦資訊處理標準 (FIPS) 140-2 概觀。

我們強烈建議您絕對不要將客戶的電子郵件地址等機密或敏感資訊，放在標籤或自由格式的文字欄位中，例如名稱欄位。這包括當您使用主控台、API AWS AppSync 或 AWS SDK 時 AWS 服務 使用或其他使用時。 AWS CLI您在標籤或自由格式文字欄位中輸入的任何資料都可能用於計費或診斷日誌。如果您提供外部伺服器的 URL，我們強烈建議請勿在驗證您對該伺服器請求的 URL 中包含憑證資訊。

運動中的加密

AWS AppSync與所有 AWS 服務一樣，在使用已發佈的 API 和 SDK 時，會使用 TLS1.2 及更高 AWS 版本進行通訊。

AWS AppSync 與 Amazon DynamoDB 等其他 AWS 服務搭配使用可確保傳輸過程中的加密：除非另有說明，否則所有 AWS 服務都使用 TLS 1.2 及更高版本互相通訊。對於使用 Amazon EC2 的解析器 CloudFront，或者您有責任確認 TLS (HTTPS) 已設定且安全無虞。如需在 Amazon EC2 中設定 HTTPS 的相關資訊，請參閱 Amazon EC2 使用者指南中的在亞馬遜 Linux 2 上設定 SSL/TLS。如需設定 HTTPS 的相關資訊 CloudFront，請參閱 CloudFront 使用者指南 CloudFront中的 Amazon 中的 HTTPS。

符合性驗證 AWS AppSync

協力廠商稽核人員會評估其安全性與合規性， AWS AppSync 做為多個 AWS 合規計畫的一部分。AWS AppSync 符合 SOC、PCI、HIPAA、HIPAA、IRAP、C5、ENS 高階、OSPAR 和 HITRUST CSF 計畫的規範。

若要瞭解 AWS 服務 是否屬於特定規範遵循方案的範圍內，請參閱AWS 服務 遵循規範計劃方案中的，並選擇您感興趣的合規方案。如需一般資訊，請參閱AWS 規範計劃AWS。

您可以使用下載第三方稽核報告 AWS Artifact。如需詳細資訊，請參閱下載中的報告中的 AWS Artifact。

您在使用時的合規責任取決 AWS 服務 於資料的敏感性、公司的合規目標以及適用的法律和法規。AWS 提供下列資源以協助遵循法規：

- 安全性與合規性快速入門指南 — 這些部署指南討論架構考量，並提供部署以安全性和合規性 AWS 為重點的基準環境的步驟。
- 在 Amazon Web Services 上架構 HIPAA 安全性與合規性 — 本白皮書說明公司如何使用建立符合 HIPAA 資格的應 AWS 用程式。

> **Note**
>
> 並非所有人 AWS 服務 都符合 HIPAA 資格。如需詳細資訊，請參閱 HIPAA 資格服務參照。

- AWS 合規資源AWS — 此工作簿和指南集合可能適用於您的產業和所在地。
- AWS 客戶合規指南 — 透過合規的角度瞭解共同的責任模式。這份指南總結了在多個架構 (包括美國國家標準技術研究所 (NIST)、支付卡產業安全標準委員會 (PCI) 和國際標準化組織 (ISO)) 中，保 AWS 服務 護指引並對應至安全控制的最佳實務。
- 使用AWS Config 開發人員指南中的規則評估資源 — 此 AWS Config 服務會評估您的資源組態符合內部實務、產業準則和法規的程度。
- AWS Security Hub— 這 AWS 服務 提供了內部安全狀態的全面視圖 AWS。Security Hub 使用安全控制，可評估您的 AWS 資源並檢查您的法規遵循是否符合安全業界標準和最佳實務。如需支援的服務和控制清單，請參閱 Security Hub controls reference。
- Amazon GuardDuty — 透過監控環境中的 AWS 帳戶可疑和惡意活動，藉此 AWS 服務 偵測您的工作負載、容器和資料的潛在威脅。 GuardDuty 可協助您滿足特定合規性架構所要求的入侵偵測需求，例如 PCI DSS 等各種合規性需求。
- AWS Audit Manager— 這 AWS 服務 有助於您持續稽核您的 AWS 使用情況，以簡化您管理風險的方式，以及遵守法規和業界標準的方式。

基礎架構安全性 AWS AppSync

作為託管服務， AWS AppSync 受到 AWS 全球網絡安全的保護。有關 AWS 安全服務以及如何 AWS 保護基礎結構的詳細資訊，請參閱AWS 雲端安全 若要使用基礎架構安全性的最佳做法來設計您的 AWS 環境，請參閱安全性支柱架構良 AWS 好的架構中的基礎結構保護。

您可以使用 AWS 已發佈的 API 呼叫透 AWS AppSync 過網路進行存取。使用者端必須支援下列專案：

- Transport Layer Security (TLS)。我們需要 TLS 1.2 並建議使用 TLS 1.3。
- 具備完美轉送私密(PFS)的密碼套件，例如 DHE (Ephemeral Diffie-Hellman)或 ECDHE (Elliptic Curve Ephemeral Diffie-Hellman)。現代系統(如 Java 7 和更新版本)大多會支援這些模式。

此外，請求必須使用存取金鑰 ID 和與 IAM 主體相關聯的私密存取金鑰來簽署。或者，您可以透過 AWS Security Token Service (AWS STS) 來產生暫時安全憑證來簽署請求。

韌性在 AWS AppSync

AWS 全球基礎架構是圍繞區 AWS 域和可用區域建立的。 AWS 區域提供多個實體分離和隔離的可用區域，這些區域透過低延遲、高輸送量和高度備援的網路連線。透過可用區域，您可以設計與操作的應用程式和資料庫，在可用區域之間自動容錯移轉而不會發生中斷。可用區域的可用性、容錯能力和擴展能力，均較單一或多個資料中心的傳統基礎設施還高。

如需區域和可用區域的相關 AWS 資訊，請參閱AWS 全域基礎結構。

除了 AWS 全域基礎結構之外，還 AWS AppSync 允許使用 AWS CloudFormation 範本定義大多數資源；如需使用範本宣告 AWS AppSync 資源的 AWS CloudFormation 範例，請參閱 AWS 部落格和使用AWS CloudFormation 者指南上的 AWS AppSync Pipeline Resolvers 實際使用案例。

的身分識別與存取管理 AWS AppSync

AWS Identity and Access Management (IAM) 可協助系統管理員安全地控制 AWS 資源存取權。 AWS 服務 IAM 管理員控制哪些人可以通過身份驗證 (登入) 和授權 (具有權限) 來使用 AWS AppSync 資源。IAM 是您可以使用的 AWS 服務 ，無需額外付費。

主題

- 物件
- 使用身分驗證
- 使用政策管理存取權
- 如何與 IAM AWS AppSync 搭配使用

- 適用於 AWS AppSync 的身分型政策
- 疑難排解 AWS AppSync 身分和存取

物件

您使用 AWS Identity and Access Management (IAM) 的方式會有所不同，具體取決於您在進行的工作 AWS AppSync。

服務使用者 — 如果您使用 AWS AppSync 服務執行工作，則管理員會為您提供所需的認證和權限。當您使用更多 AWS AppSync 功能來完成工作時，您可能需要其他權限。了解存取許可的管理方式可協助您向管理員請求正確的許可。如果無法存取中的圖徵 AWS AppSync，請參閱疑難排解 AWS AppSync 身分和存取。

服務管理員 — 如果您負責公司的 AWS AppSync 資源，您可能擁有完整的存取權 AWS AppSync。決定您的服務使用者應該存取哪些 AWS AppSync 功能和資源是您的工作。接著，您必須將請求提交給您的 IAM 管理員，來變更您服務使用者的許可。檢閱此頁面上的資訊，了解 IAM 的基本概念。若要進一步瞭解貴公司如何搭配使用 IAM AWS AppSync，請參閱如何與 IAM AWS AppSync 搭配使用。

IAM 管理員 — 如果您是 IAM 管理員，您可能想要瞭解如何撰寫政策來管理存取權限的詳細資訊 AWS AppSync。若要檢視可在 IAM 中使用的 AWS AppSync 基於身分的政策範例，請參閱。適用於 AWS AppSync 的身分型政策

使用身分驗證

驗證是您 AWS 使用身分認證登入的方式。您必須以 IAM 使用者身分或假設 IAM 角色進行驗證 (登入 AWS)。 AWS 帳戶根使用者

您可以使用透過 AWS 身分識別來源提供的認證，以聯合身分識別身分登入。 AWS IAM Identity Center (IAM 身分中心) 使用者、貴公司的單一登入身分驗證，以及您的 Google 或 Facebook 登入資料都是聯合身分識別的範例。您以聯合身分登入時，您的管理員先前已設定使用 IAM 角色的聯合身分。當您使 AWS 用同盟存取時，您會間接擔任角色。

根據您的使用者類型，您可以登入 AWS Management Console 或 AWS 存取入口網站。如需有關登入的詳細資訊 AWS，請參閱《AWS 登入 使用指南》 AWS 帳戶中的如何登入您的。

如果您 AWS 以程式設計方式存取，請 AWS 提供軟體開發套件 (SDK) 和命令列介面 (CLI)，以使用您的認證以加密方式簽署要求。如果您不使用 AWS 工具，則必須自行簽署要求。如需使用建議的方法自行簽署請求的詳細資訊，請參閱 IAM 使用者指南中的簽署 AWS API 請求。

無論您使用何種身分驗證方法，您可能都需要提供額外的安全性資訊。例如， AWS 建議您使用多重要素驗證 (MFA) 來增加帳戶的安全性。如需更多資訊，請參閱 AWS IAM Identity Center 使用者指南中的多重要素驗證和 IAM 使用者指南中的在 AWS中使用多重要素驗證 (MFA)。

AWS 帳戶 根使用者

當您建立時 AWS 帳戶，您會從一個登入身分開始，該身分可完整存取該帳戶中的所有資源 AWS 服務和資源。此身分稱為 AWS 帳戶 root 使用者，可透過使用您用來建立帳戶的電子郵件地址和密碼登入來存取。強烈建議您不要以根使用者處理日常任務。保護您的根使用者憑證，並將其用來執行只能由根使用者執行的任務。如需這些任務的完整清單，了解需以根使用者登入的任務，請參閱 IAM 使用者指南中的需要根使用者憑證的任務。

聯合身分

最佳作法是要求人類使用者 (包括需要系統管理員存取權的使用者) 使用與身分識別提供者的同盟，才能使用臨時登入資料進行存取 AWS 服務 。

聯合身分識別是來自企業使用者目錄的使用者、Web 身分識別提供者、Identi ty Center 目錄，或使用透過身分識別來源提供的認證進行存取 AWS 服務 的任何使用者。 AWS Directory Service同盟身分存取時 AWS 帳戶，他們會假設角色，而角色則提供臨時認證。

對於集中式存取權管理，我們建議您使用 AWS IAM Identity Center。您可以在 IAM Identity Center 中建立使用者和群組，也可以連線並同步到自己身分識別來源中的一組使用者和群組，以便在所有應用程式 AWS 帳戶 和應用程式中使用。如需 IAM Identity Center 的相關資訊，請參閱 AWS IAM Identity Center 使用者指南中的什麼是 IAM Identity Center？。

IAM 使用者和群組

IAM 使用者是您內部的身分，具 AWS 帳戶 有單一人員或應用程式的特定許可。建議您盡可能依賴暫時憑證，而不是擁有建立長期憑證 (例如密碼和存取金鑰) 的 IAM 使用者。但是如果特定使用案例需要擁有長期憑證的 IAM 使用者，建議您輪換存取金鑰。如需更多資訊，請參閱 IAM 使用者指南中的為需要長期憑證的使用案例定期輪換存取金鑰。

IAM 群組是一種指定 IAM 使用者集合的身分。您無法以群組身分簽署。您可以使用群組來一次為多名使用者指定許可。群組可讓管理大量使用者許可的程序變得更為容易。例如，您可以擁有一個名為 IAMAdmins 的群組，並給予該群組管理 IAM 資源的許可。

使用者與角色不同。使用者只會與單一人員或應用程式建立關聯，但角色的目的是在由任何需要它的人員取得。使用者擁有永久的長期憑證，但角色僅提供暫時憑證。如需進一步了解，請參閱 IAM 使用者指南中的建立 IAM 使用者 (而非角色) 的時機。

IAM 角色

IAM 角色是您 AWS 帳戶 內部具有特定許可的身分。它類似 IAM 使用者，但不與特定的人員相關聯。您可以切換角色，在中暫時擔任 IAM 角色。 AWS Management Console 您可以透過呼叫 AWS CLI 或 AWS API 作業或使用自訂 URL 來擔任角色。如需使用角色的方法更多相關資訊，請參閱 IAM 使用者指南中的使用 IAM 角色。

使用暫時憑證的 IAM 角色在下列情況中非常有用：

- 聯合身分使用者存取 – 若要向聯合身分指派許可，請建立角色，並為角色定義許可。當聯合身分進行身分驗證時，該身分會與角色建立關聯，並獲授予由角色定義的許可。如需有關聯合角色的相關資訊，請參閱 IAM 使用者指南中的為第三方身分提供者建立角色。如果您使用 IAM Identity Center，則需要設定許可集。為控制身分驗證後可以存取的內容，IAM Identity Center 將許可集與 IAM 中的角色相關聯。如需有關許可集的資訊，請參閱 AWS IAM Identity Center 使用者指南中的許可集。
- 暫時 IAM 使用者許可 – IAM 使用者或角色可以擔任 IAM 角色來暫時針對特定任務採用不同的許可。
- 跨帳戶存取權 – 您可以使用 IAM 角色，允許不同帳戶中的某人 (信任的委託人) 存取您帳戶中的資源。角色是授予跨帳戶存取權的主要方式。但是，對於某些策略 AWS 服務，您可以將策略直接附加到資源（而不是使用角色作為代理）。若要了解跨帳戶存取權角色和資源型政策間的差異，請參閱 IAM 使用者指南中的 IAM 角色與資源類型政策的差異。
- 跨服務訪問 — 有些 AWS 服務 使用其他 AWS 服務功能。例如，當您在服務中進行呼叫時，該服務通常會在 Amazon EC2 中執行應用程式或將物件儲存在 Amazon Simple Storage Service (Amazon S3) 中。服務可能會使用呼叫主體的許可、使用服務角色或使用服務連結角色來執行此作業。
  - 轉寄存取工作階段 (FAS) — 當您使用 IAM 使用者或角色在中執行動作時 AWS，您會被視為主體。使用某些服務時，您可能會執行某個動作，進而在不同服務中啟動另一個動作。FAS 會使用主體呼叫的權限 AWS 服務，並結合要求 AWS 服務 向下游服務發出要求。只有當服務收到需要與其 AWS 服務 他資源互動才能完成的請求時，才會發出 FAS 請求。在此情況下，您必須具有執行這兩個動作的許可。如需提出 FAS 請求時的政策詳細資訊，請參閱《轉發存取工作階段》。
  - 服務角色 – 服務角色是服務擔任的 IAM 角色，可代表您執行動作。IAM 管理員可以從 IAM 內建立、修改和刪除服務角色。如需更多資訊，請參閱 IAM 使用者指南中的建立角色以委派許可給 AWS 服務。
  - 服務連結角色 — 服務連結角色是連結至. AWS 服務服務可以擔任代表您執行動作的角色。服務連結角色會顯示在您的中， AWS 帳戶 且屬於服務所有。IAM 管理員可以檢視，但不能編輯服務連結角色的許可。
- 在 Amazon EC2 上執行的應用程式 — 您可以使用 IAM 角色來管理在 EC2 執行個體上執行的應用程式以及發出 AWS CLI 或 AWS API 請求的臨時登入資料。這是在 EC2 執行個體內儲存存取金鑰的較好方式。若要將 AWS 角色指派給 EC2 執行個體並提供給其所有應用程式，請建立連接至執行個體

的執行個體設定檔。執行個體設定檔包含該角色，並且可讓 EC2 執行個體上執行的程式取得暫時憑證。如需更多資訊，請參閱 IAM 使用者指南中的利用 IAM 角色來授予許可給 Amazon EC2 執行個體上執行的應用程式。

若要了解是否要使用 IAM 角色或 IAM 使用者，請參閱 IAM 使用者指南中的建立 IAM 角色 (而非使用者) 的時機。

使用政策管理存取權

您可以透 AWS 過建立原則並將其附加至 AWS 身分識別或資源來控制中的存取。原則是一個物件 AWS ，當與身分識別或資源相關聯時，會定義其權限。 AWS 當主參與者 (使用者、root 使用者或角色工作階段) 提出要求時，評估這些原則。政策中的許可決定是否允許或拒絕請求。大多數原則會 AWS 以 JSON 文件的形式儲存在中。如需 JSON 政策文件結構和內容的更多相關資訊，請參閱 IAM 使用者指南中的 JSON 政策概觀。

管理員可以使用 AWS JSON 政策來指定誰可以存取哪些內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

預設情況下，使用者和角色沒有許可。若要授予使用者對其所需資源執行動作的許可，IAM 管理員可以建立 IAM 政策。然後，管理員可以將 IAM 政策新增至角色，使用者便能擔任這些角色。

IAM 政策定義該動作的許可，無論您使用何種方法來執行操作。例如，假設您有一個允許 `iam:GetRole` 動作的政策。具有該原則的使用者可以從 AWS Management Console AWS CLI、或 AWS API 取得角色資訊。

身分型政策

身分型政策是可以附加到身分 (例如 IAM 使用者、使用者群組或角色) 的 JSON 許可政策文件。這些政策可控制身分在何種條件下能對哪些資源執行哪些動作。若要了解如何建立身分類型政策，請參閱 IAM 使用者指南中的建立 IAM 政策。

身分型政策可進一步分類成內嵌政策或受管政策。內嵌政策會直接內嵌到單一使用者、群組或角色。受管理的策略是獨立策略，您可以將其附加到您的 AWS 帳戶. 受管政策包括 AWS 受管政策和客戶管理的策略。若要了解如何在受管政策及內嵌政策間選擇，請參閱 IAM 使用者指南中的在受管政策和內嵌政策間選擇。

資源型政策

資源型政策是連接到資源的 JSON 政策文件。資源型政策的最常見範例是 IAM 角色信任政策和 Amazon S3 儲存貯體政策。在支援資源型政策的服務中，服務管理員可以使用它們來控制對特定資源

的存取權限。對於附加政策的資源，政策會定義指定的主體可以對該資源執行的動作以及在何種條件下執行的動作。您必須在資源型政策中指定主體。主參與者可以包括帳戶、使用者、角色、同盟使用者或。 AWS 服務

資源型政策是位於該服務中的內嵌政策。您無法在以資源為基礎的政策中使用 IAM 的 AWS 受管政策。

存取控制清單 (ACL)

存取控制清單 (ACL) 可控制哪些委託人 (帳戶成員、使用者或角色) 擁有存取某資源的許可。ACL 類似於資源型政策，但它們不使用 JSON 政策文件格式。

Amazon S3 和 Amazon VPC 是支援 ACL 的服務範例。 AWS WAF若要進一步了解 ACL，請參閱 Amazon Simple Storage Service 開發人員指南中的存取控制清單 (ACL) 概觀。

其他政策類型

AWS 支援其他較不常見的原則類型。這些政策類型可設定較常見政策類型授予您的最大許可。

- 許可界限 – 許可範圍是一種進階功能，可供您設定身分型政策能授予 IAM 實體 (IAM 使用者或角色) 的最大許可。您可以為實體設定許可界限。所產生的許可會是實體的身分型政策和其許可界限的交集。會在 `Principal` 欄位中指定使用者或角色的資源型政策則不會受到許可界限限制。所有這類政策中的明確拒絕都會覆寫該允許。如需許可範圍的更多相關資訊，請參閱 IAM 使用者指南中的 IAM 實體許可範圍。
- 服務控制策略（SCP）— SCP 是 JSON 策略，用於指定中組織或組織單位（OU）的最大權限。AWS Organizations AWS Organizations 是一種用於分組和集中管理您企業擁 AWS 帳戶 有的多個服務。若您啟用組織中的所有功能，您可以將服務控制政策 (SCP) 套用到任何或所有帳戶。SCP 限制成員帳戶中實體的權限，包括每個 AWS 帳戶根使用者帳戶。如需組織和 SCP 的更多相關資訊，請參閱 AWS Organizations 使用者指南中的 SCP 運作方式。
- 工作階段政策 – 工作階段政策是一種進階政策，您可以在透過編寫程式的方式建立角色或聯合使用者的暫時工作階段時，作為參數傳遞。所產生工作階段的許可會是使用者或角色的身分型政策和工作階段政策的交集。許可也可以來自資源型政策。所有這類政策中的明確拒絕都會覆寫該允許。如需更多資訊，請參閱 IAM 使用者指南中的工作階段政策。

多種政策類型

將多種政策類型套用到請求時，其結果形成的許可會更為複雜、更加難以理解。要了解如何在涉及多個政策類型時 AWS 確定是否允許請求，請參閱《IAM 使用者指南》中的政策評估邏輯。

如何與 IAM AWS AppSync 搭配使用

在您使用 IAM 管理存取權限之前 AWS AppSync，請先了解哪些 IAM 功能可搭配使用 AWS AppSync。

可搭配使用的 IAM 功能 AWS AppSync

IAM 功能	AWS AppSync 支持
身分型政策	是
資源型政策	否
政策動作	是
政策資源	是
政策條件索引鍵	否
ACL	否
ABAC(政策中的標籤)	部分
臨時憑證	是
轉送存取工作階段 (FAS)	部分
服務角色	否
服務連結角色	部分

若要深入瞭解如何以 AWS AppSync 及其他 AWS 服務如何使用大多數 IAM 功能，請參閱 IAM 使用者指南中的搭配 IAM 使用的AWS 服務。

以身分識別為基礎的原則 AWS AppSync

支援身分型政策	是

身分型政策是可以連接到身分 (例如 IAM 使用者、使用者群組或角色) 的 JSON 許可政策文件。這些政策可控制身分在何種條件下能對哪些資源執行哪些動作。若要了解如何建立身分類型政策，請參閱《IAM 使用者指南》中的建立 IAM 政策。

使用 IAM 身分型政策，您可以指定允許或拒絕的動作和資源，以及在何種條件下允許或拒絕動作。您無法在身分型政策中指定主體，因為這會套用至連接的使用者或角色。如要了解您在 JSON 政策中使用的所有元素，請參閱《IAM 使用者指南》中的 IAM JSON 政策元素參考。

以身分識別為基礎的原則範例 AWS AppSync

若要檢視以 AWS AppSync 身分為基礎的原則範例，請參閱。適用於 AWS AppSync 的身分型政策

以資源為基礎的政策 AWS AppSync

支援以資源基礎的政策	否

資源型政策是附加到資源的 JSON 政策文件。資源型政策的最常見範例是 IAM 角色信任政策和 Amazon S3 儲存貯體政策。在支援資源型政策的服務中，服務管理員可以使用它們來控制對特定資源的存取權限。對於附加政策的資源，政策會定義指定的主體可以對該資源執行的動作以及在何種條件下執行的動作。您必須在資源型政策中指定主體。主參與者可以包括帳戶、使用者、角色、同盟使用者或。 AWS 服務

若要啟用跨帳戶存取，您可以指定在其他帳戶內的所有帳戶或 IAM 實體，作為資源型政策的主體。新增跨帳戶主體至資源型政策，只是建立信任關係的一半。當主體和資源位於不同時 AWS 帳戶，受信任帳戶中的 IAM 管理員也必須授與主體實體 (使用者或角色) 權限，才能存取資源。其透過將身分型政策連接到實體來授與許可。不過，如果資源型政策會為相同帳戶中的主體授予存取，這時就不需要額外的身分型政策。如需詳細資訊，請參閱《IAM 使用者指南》中的 IAM 角色與資源型政策有何差異。

的政策動作 AWS AppSync

支援政策動作	是

管理員可以使用 AWS JSON 政策來指定誰可以存取哪些內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

JSON 政策的 Action 元素描述您可以用來允許或拒絕政策中存取的動作。原則動作通常與關聯的 AWS API 作業具有相同的名稱。有一些例外狀況，例如沒有相符的 API 操作的僅限許可動作。也有一些作業需要政策中的多個動作。這些額外的動作稱為相依動作。

政策會使用動作來授予執行相關聯動作的許可。

若要查看 AWS AppSync 動作清單，請參閱服務授權參考 AWS AppSync中所定義的動作。

中的策略動作在動作之前 AWS AppSync 使用下列前置詞：

```
appsync
```

若要在單一陳述式中指定多個動作，請用逗號分隔。

```
"Action": [
      "appsync:action1",
      "appsync:action2"
         ]
```

若要檢視以 AWS AppSync 身分為基礎的原則範例，請參閱。適用於 AWS AppSync 的身分型政策

的政策資源 AWS AppSync

支援政策資源	是

管理員可以使用 AWS JSON 政策來指定誰可以存取哪些內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

Resource JSON 政策元素可指定要套用動作的物件。陳述式必須包含 Resource 或 NotResource 元素。最佳實務是使用其 Amazon Resource Name (ARN) 來指定資源。您可以針對支援特定資源類型的動作 (稱為資源層級許可) 來這麼做。

對於不支援資源層級許可的動作 (例如列出操作)，請使用萬用字元 (*) 來表示陳述式適用於所有資源。

```
"Resource": "*"
```

若要查看 AWS AppSync 資源類型及其 ARN 的清單，請參閱服務授權參考 AWS AppSync中所定義的資源。若要瞭解您可以使用哪些動作指定每個資源的 ARN，請參閱由 AWS AppSync定義的動作。

若要檢視以 AWS AppSync 身分為基礎的原則範例，請參閱。適用於 AWS AppSync 的身分型政策

的政策條件索引鍵 AWS AppSync

支援服務特定政策條件金鑰	否

管理員可以使用 AWS JSON 政策來指定誰可以存取哪些內容。也就是說，哪個主體在什麼條件下可以對什麼資源執行哪些動作。

Condition 元素 (或 Condition 區塊) 可讓您指定使陳述式生效的條件。Condition 元素是選用項目。您可以建立使用條件運算子的條件運算式 (例如等於或小於)，來比對政策中的條件和請求中的值。

若您在陳述式中指定多個 Condition 元素，或是在單一 Condition 元素中指定多個索引鍵， AWS 會使用邏輯 AND 操作評估他們。如果您為單一條件索引鍵指定多個值，請使用邏輯OR運算來 AWS 評估條件。必須符合所有條件，才會授與陳述式的許可。

您也可以在指定條件時使用預留位置變數。例如，您可以只在使用者使用其 IAM 使用者名稱標記時，將存取資源的許可授予該 IAM 使用者。如需更多資訊，請參閱 IAM 使用者指南中的 IAM 政策元素：變數和標籤。

AWS 支援全域條件金鑰和服務特定條件金鑰。若要查看所有 AWS 全域條件金鑰，請參閱《IAM 使用者指南》中的AWS 全域條件內容金鑰。

若要查看 AWS AppSync 條件索引鍵清單，請參閱服務授權參考 AWS AppSync中的條件金鑰。若要瞭解您可以使用條件索引鍵的動作和資源，請參閱定義的動作 AWS AppSync。

若要檢視以 AWS AppSync 身分為基礎的原則範例，請參閱。適用於 AWS AppSync 的身分型政策

AWS AppSync 中的存取控制清單 (ACL)

支援 ACL	否

存取控制清單 (ACL) 可控制哪些主體 (帳戶成員、使用者或角色) 擁有存取某資源的許可。ACL 類似於資源型政策，但它們不使用 JSON 政策文件格式。

以屬性為基礎的存取控制 (ABAC) 搭配 AWS AppSync

支援 ABAC (政策中的標籤)	部分

屬性型存取控制 (ABAC) 是一種授權策略，可根據屬性來定義許可。在中 AWS，這些屬性稱為標籤。您可以將標籤附加到 IAM 實體 (使用者或角色) 和許多 AWS 資源。為實體和資源加上標籤是 ABAC 的第一步。您接著要設計 ABAC 政策，允許在主體的標籤與其嘗試存取的資源標籤相符時操作。

ABAC 在成長快速的環境中相當有幫助，並能在政策管理變得繁瑣時提供協助。

若要根據標籤控制存取，請使用 `aws:ResourceTag/key-name`、`aws:RequestTag/key-name` 或 `aws:TagKeys` 條件金鑰，在政策的條件元素中，提供標籤資訊。

如果服務支援每個資源類型的全部三個條件金鑰，則對該服務而言，值為 Yes。如果服務僅支援某些資源類型的全部三個條件金鑰，則值為 Partial。

如需 ABAC 的詳細資訊，請參閱《IAM 使用者指南》中的什麼是 ABAC?。如要查看含有設定 ABAC 步驟的教學課程，請參閱《IAM 使用者指南》中的使用屬性型存取控制 (ABAC)。

使用臨時登入資料 AWS AppSync

支援臨時憑證	是

當您使用臨時憑據登錄時，某些 AWS 服務 不起作用。如需其他資訊，包括哪些 AWS 服務 與臨時登入資料搭配AWS 服務 使用，請參閱 IAM 使用者指南中的 IAM。

如果您使用除了使用者名稱和密碼以外的任何方法登入，則您正在 AWS Management Console 使用臨時認證。例如，當您 AWS 使用公司的單一登入 (SSO) 連結存取時，該程序會自動建立暫時認證。當您以使用者身分登入主控台，然後切換角色時，也會自動建立臨時憑證。如需切換角色的詳細資訊，請參閱《IAM 使用者指南》中的切換至角色 (主控台)。

您可以使用 AWS CLI 或 AWS API 手動建立臨時登入資料。然後，您可以使用這些臨時登入資料來存取 AWS。 AWS 建議您動態產生臨時登入資料，而不是使用長期存取金鑰。如需詳細資訊，請參閱 IAM 中的暫時性安全憑證。

轉寄存取工作階段 AWS AppSync

支援轉寄存取工作階段 (FAS)	部分

當您使用 IAM 使用者或角色在中執行動作時 AWS，您會被視為主體。使用某些服務時，您可能會執行某個動作，進而在不同服務中啟動另一個動作。FAS 會使用主體呼叫的權限 AWS 服務，並結合要求 AWS 服務 向下游服務發出要求。只有當服務收到需要與其 AWS 服務 他資源互動才能完成的請求時，才會發出 FAS 請求。在此情況下，您必須具有執行這兩個動作的許可。如需提出 FAS 請求時的政策詳細資訊，請參閱《轉發存取工作階段》。

AWS AppSync 的服務角色

支援服務角色	否

服務角色是服務擔任的 IAM 角色，可代您執行動作。IAM 管理員可以從 IAM 內建立、修改和刪除服務角色。如需更多資訊，請參閱 IAM 使用者指南中的建立角色以委派許可給 AWS 服務。

> **Warning**
>
> 變更服務角色的權限可能會中斷 AWS AppSync 功能。只有在 AWS AppSync 提供指引時才編輯服務角色。

服務連結角色 AWS AppSync

支援服務連結角色	部分

服務連結角色是一種連結至. AWS 服務服務可以擔任代表您執行動作的角色。服務連結角色會顯示在您的中， AWS 帳戶 且屬於服務所有。IAM 管理員可以檢視，但不能編輯服務連結角色的許可。

如需建立或管理服AWS 務連結角色的詳細資訊，請參閱 IAM 使用者指南中的與 IAM 搭配使用的服務。在表格中尋找服務，其中包含服務連結角色欄中的 Yes。選擇是連結，以檢視該服務的服務連結角色文件。

適用於 AWS AppSync 的身分型政策

依預設，使用者和角色沒有建立或修改 AWS AppSync資源的權限。他們也無法使用 AWS Management Console、 AWS Command Line Interface (AWS CLI) 或 AWS API 來執行工作。若要授予使用者對其所需資源執行動作的許可，IAM 管理員可以建立 IAM 政策。然後，管理員可以將 IAM 政策新增至角色，使用者便能擔任這些角色。

若要了解如何使用這些範例 JSON 政策文件建立 IAM 身分型政策，請參閱《IAM 使用者指南》中的建立 IAM 政策。

如需有關由定義的動作和資源類型的詳細資訊 AWS AppSync，包括每個資源類型的 ARN 格式，請參閱服務授權參考 AWS AppSync中的動作、資源和條件索引鍵。

若要了解建立和設定 IAM 身分型政策的最佳做法，請參閱。the section called “IAM 政策最佳做法”

如需的 IAM 身分型政策清單 AWS AppSync，請參閱。AWS 受管理的政策 AWS AppSync

主題

- 使用 AWS AppSync 主控台
- 允許使用者檢視他們自己的許可
- 存取一個 Amazon S3 儲存貯體
- 根據標籤檢視 AWS AppSync 小工具
- AWS 受管理的政策 AWS AppSync

使用 AWS AppSync 主控台

若要存取 AWS AppSync 主控台，您必須擁有最少一組權限。這些權限必須允許您列出和檢視有關 AWS 帳戶. AWS AppSync 如果您建立比最基本必要許可更嚴格的身分型政策，則對於具有該政策的實體 (使用者或角色) 而言，主控台就無法如預期運作。

您不需要為僅對 AWS CLI 或 AWS API 進行呼叫的使用者允許最低主控台權限。反之，只需允許存取符合他們嘗試執行之 API 操作的動作就可以了。

為確保 IAM 使用者和角色仍可使用 AWS AppSync 主控台，請同時將 AWS AppSync `ConsoleAccess`或`ReadOnly` AWS 受管政策附加到實體。如需詳細資訊，請參閱《IAM 使用者指南》中的新增許可到使用者。

允許使用者檢視他們自己的許可

此範例會示範如何建立政策，允許 IAM 使用者檢視附加到他們使用者身分的內嵌及受管政策。此原則包含在主控台上或以程式設計方式使用 AWS CLI 或 AWS API 完成此動作的權限。

```
{
    "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": "*"
        }
    ]
}
```

存取一個 Amazon S3 儲存貯體

在此範例中，您想要授與 AWS 帳戶中的 IAM 使用者存取其中一個 Amazon S3 儲存貯體的存取權`examplebucket`。您也希望允許使用者新增、更新和刪除物件。

除了授予使用者 `s3:PutObject`、`s3:GetObject` 與 `s3:DeleteObject` 許可之外，政策也會授予 `s3:ListAllMyBuckets`、`s3:GetBucketLocation` 與 `s3:ListBucket` 許可。這些是主控台需要的額外許可。還需要 `s3:PutObjectAcl` 與 `s3:GetObjectAcl` 動作才能在主控台中複製、剪下與貼上物件。如需授與使用者權限並使用主控台測試權限的範例逐步解說，請參閱[逐步解說範例：使用使用者政策控制值區的存取權](#)。

```
{
   "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/*"
      }
   ]
}
```

根據標籤檢視 AWS AppSync

您可以使用以身分識別為基礎的原則中的條件，根據標籤來控制 AWS AppSync資源的存取。此範例顯示如何建立允許檢視 *Widget* 的策略。不過，只有在 *Widget* ###Owner有該使用者名稱的值時，才會授與權限。此政策也會授予在主控台完成此動作的必要許可。

```
{
    "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}"}
            }
        }
    ]
}
```

您可以將此政策連接到您帳戶中的 IAM 使用者。如果名為的使用者`richard-roe`嘗試檢視 AWS AppSync`###`，則必須標記`#`器具`Owner=richard-roe`或`owner=richard-roe`。否則他便會被拒絕存取。條件標籤鍵 `Owner` 符合 `Owner` 和 `owner`，因為條件索引鍵名稱不區分大小寫。如需更多資訊，請參閱 IAM 使用者指南中的 IAM JSON 政策元素：條件。

AWS 受管理的政策 AWS AppSync

若要新增使用者、群組和角色的權限，使用 AWS 受管理的原則比自己撰寫原則更容易。建立 IAM 客戶受管政策需要時間和專業知識，而受管政策可為您的團隊提供其所需的許可。若要快速開始使用，您可以使用我們的 AWS 受管政策。這些政策涵蓋常見的使用案例，並可在您的 AWS 帳戶中使用。如需 AWS 受管政策的詳細資訊，請參閱 IAM 使用者指南中的AWS 受管政策。

AWS 服務會維護和更新 AWS 受管理的策略。您無法變更 AWS 受管理原則中的權限。服務有時會將其他權限新增至受 AWS 管理的策略，以支援新功能。此類型的更新會影響已連接政策的所有身分識別 (使用者、群組和角色)。當新功能啟動或新作業可用時，服務最有可能更新 AWS 受管理的策略。服務不會從 AWS 受管理的政策移除權限，因此政策更新不會破壞您現有的權限。

此外，還 AWS 支援跨多個服務之工作職能的受管理原則。例如，ReadOnlyAccess AWS 受管理的策略提供對所有 AWS 服務和資源的唯讀存取權。當服務啟動新功能時，會為新作業和資源新 AWS 增唯讀權限。如需任務職能政策的清單和說明，請參閱 IAM 使用者指南中有關任務職能的AWS 受管政策。

AWS 受管理策略： AWSAppSyncInvokeFullAccess

使用受AWSAppSyncInvokeFullAccess AWS 管理原則可讓您的系統管理員透過主控台或獨立存取 AWS AppSync 服務。

您可將 AWSAppSyncInvokeFullAccess 政策連接到 IAM 身分。

許可詳細資訊

此政策包含以下許可。

- AWS AppSync— 允許對中的所有資源進行完全管理訪問 AWS AppSync

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "appsync:GraphQL",
                "appsync:GetGraphqlApi",
                "appsync:ListGraphqlApis",
                "appsync:ListApiKeys"
            ],
            "Resource": "*"
        }
    ]
}
```

AWS 受管理策略： AWSAppSyncSchemaAuthor

使用受AWSAppSyncSchemaAuthor AWS 管政策允許 IAM 使用者存取以建立、更新和查詢其 GraphQL 結構描述。如需使用者可以使用這些權限執行哪些動作的資訊，請參閱設計 GraphQL 的 API。

您可將 AWSAppSyncSchemaAuthor 政策連接到 IAM 身分。

許可詳細資訊

此政策包含以下許可。

- AWS AppSync— 允許執行下列動作：
  - 建立 GraphQL 結構描述
  - 允許建立、修改和刪除 GraphQL 類型、解析器和函數
  - 評估請求和響應模板邏輯
  - 使用運行時和上下文評估代碼
  - 將 GraphQL 查詢傳送至 GraphQL API
  - 擷取 GraphQL 資料

```
{
    "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 受管理策略： AWSAppSyncPushToCloudWatchLogs

AWS AppSync 使用 Amazon CloudWatch 產生可用於疑難排解和優化 GraphQL 請求的日誌，藉此監控應用程式的效能。如需詳細資訊，請參閱 監控和記錄。

使用受AWSAppSyncPushToCloudWatchLogs AWS 管政策允許 AWS AppSync 將日誌推送到 IAM 使用者的 CloudWatch 帳戶。

您可將 AWSAppSyncPushToCloudWatchLogs 政策連接到 IAM 身分。

許可詳細資訊

此政策包含以下許可。

- CloudWatch Logs— 允許使 AWS AppSync 用指定名稱建立記錄群組和串流。 AWS AppSync將記錄事件推送至指定的記錄資料流。

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogGroup",
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "*"
```

```
        }
    ]
}
```

AWS 受管理策略： AWSAppSyncAdministrator

使用受AWSAppSyncAdministrator AWS 管理原則可讓您的系統管理員存取 AWS AppSync除 AWS 主控台以外的所有內容。

您可以將 AWSAppSyncAdministrator 連接到 IAM 實體。 AWS AppSync 也會將此原則附加至可讓其代表您執行動作的服務角色。

許可詳細資訊

此政策包含以下許可。

- AWS AppSync— 允許對中的所有資源進行完全管理訪問 AWS AppSync
- IAM— 允許執行下列動作：
  - 建立服務連結角色，以 AWS AppSync 便代表您分析其他服務中的資源
  - 刪除服務連結角色
  - 將服務連結角色傳遞給其他 AWS 服務，以便稍後擔任該角色，並代表您執行動作

```
{
    "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 受管理策略： AWSAppSyncServiceRolePolicy

使用AWSAppSyncServiceRolePolicy AWS 受管理的策略可允許存取 AWS AppSync使用或管理的 AWS 服務和資源。

您不得將 AWSAppSyncServiceRolePolicy 連接到 IAM 實體。此原則附加至服務連結角色，可 AWS AppSync 代表您執行動作。如需詳細資訊，請參閱 服務連結角色 AWS AppSync。

許可詳細資訊

此政策包含以下許可。

- X-Ray— AWS AppSync 用 AWS X-Ray 於收集有關在您的應用程序中提出的請求的數據。如需詳細資訊，請參閱 使用追蹤AWS X-Ray。

  此原則允許執行下列動作：

  - 擷取取樣規則及其結果
  - 將追蹤資料傳送至 X-Ray 精靈

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "xray:PutTraceSegments",
                "xray:PutTelemetryRecords",
                "xray:GetSamplingTargets",
                "xray:GetSamplingRules",
                "xray:GetSamplingStatisticSummaries"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

AWS AppSync AWS 受管理策略的更新

檢視 AWS AppSync 自此服務開始追蹤這些變更以來的 AWS 受管理策略更新詳細資料。如需有關此頁面變更的自動警示，請訂閱「 AWS AppSync 文件歷史記錄」頁面上的 RSS 摘要。

變更	描述	日期
AWSAppSyncSchemaAuthor - 更新現有政策	已新增EvaluateCode 原則動作，以允許使用者使用執行階段和內容評估程式碼。	2023 年 2 月 7 日

變更	描述	日期
AWSAppSyncSchemaAuthor - 更新現有政策	已新增政策動作，以允許 API 的清單、取得、建立、更新和刪除功能。 已新增EvaluateMappingTemplate 原則動作，以允許使用者評估要求與回應解析程式對應範本邏輯。 新增政策動作以允許資源標記。	2022 年 8 月 25 日
AWS AppSync 開始追蹤變更	AWS AppSync 開始追蹤其 AWS 受管理策略的變更。	2022 年 8 月 25 日

疑難排解 AWS AppSync 身分和存取

使用下列資訊可協助您診斷和修正使用和 IAM 時可能會遇到的 AWS AppSync 常見問題。

我沒有執行操作的授權 AWS AppSync

如果 AWS Management Console 告訴您您沒有執行動作的授權，則您必須聯絡您的管理員以尋求協助。您的管理員是提供您使用者名稱和密碼的人員。

當 IAM 使用者mateojackson嘗試使用主控台來檢視虛構*my-example-widget*資源的詳細資料，但他沒有虛構appsync:*GetWidget*許可時，就會發生下列範例錯誤。

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 appsync:GetWidget on resource: my-example-widget
```

在此情況下，Mateo 會請求管理員更新他的政策，允許他使用 *my-example-widget* 動作存取 appsync:*GetWidget* 資源。

我沒有授權執行 iam：PassRole

如果您收到未獲授權執行iam:PassRole動作的錯誤訊息，則必須更新您的原則以允許您將角色傳遞給 AWS AppSync。

有些 AWS 服務 允許您將現有角色傳遞給該服務，而不是建立新的服務角色或服務連結角色。如需執行此作業，您必須擁有將角色傳遞至該服務的許可。

當名為的 IAM 使用者marymajor嘗試使用主控台執行中的動作時，會發生下列範例錯誤 AWS AppSync。但是，動作請求服務具備服務角色授予的許可。Mary 沒有將角色傳遞至該服務的許可。

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

在這種情況下，Mary 的政策必須更新，允許她執行 iam:PassRole 動作。

如果您需要協助，請聯絡您的 AWS 管理員。您的管理員提供您的登入憑證。

我想要檢視我的存取金鑰

在您建立 IAM 使用者存取金鑰後，您可以隨時檢視您的存取金鑰 ID。但是，您無法再次檢視您的私密存取金鑰。若您遺失了密碼金鑰，您必須建立新的存取金鑰對。

存取金鑰包含兩個部分：存取金鑰 ID (例如 AKIAIOSFODNN7EXAMPLE) 和私密存取金鑰 (例如 wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY)。如同使用者名稱和密碼，您必須一起使用存取金鑰 ID 和私密存取金鑰來驗證您的請求。就如對您的使用者名稱和密碼一樣，安全地管理您的存取金鑰。

> **Important**
>
> 請勿將您的存取金鑰提供給第三方，甚至是協助尋找您的標準使用者 ID。通過這樣做，您可能會讓某人永久訪問您的 AWS 帳戶.

建立存取金鑰對時，您會收到提示，要求您將存取金鑰 ID 和私密存取金鑰儲存在安全位置。私密存取金鑰只會在您建立它的時候顯示一次。若您遺失了私密存取金鑰，您必須將新的存取金鑰新增到您的 IAM 使用者。您最多可以擁有兩個存取金鑰。若您已有兩個存取金鑰，您必須先刪除其中一個金鑰對，才能建立新的金鑰對。若要檢視說明，請參閱《IAM 使用者指南》中的管理存取金鑰。

我是系統管理員，想要允許其他人存取 AWS AppSync

若要允許其他人存取 AWS AppSync，您必須為需要存取的人員或應用程式建立 IAM 實體 (使用者或角色)。他們將使用該實體的憑證來存取 AWS。然後，您必須將原則附加至實體，以便在中授與其正確權限 AWS AppSync。

若要立即開始使用，請參閱《IAM 使用者指南》中的建立您的第一個 IAM 委派使用者及群組。

我想允許 AWS 帳戶以外的人員存取我的 AWS AppSync 資源

您可以建立一個角色，讓其他帳戶中的使用者或您組織外部的人員存取您的資源。您可以指定要允許哪些信任物件取得該角色。針對支援基於資源的政策或存取控制清單 (ACL) 的服務，您可以使用那些政策來授予人員存取您的資源的許可。

如需進一步了解，請參閱以下內容：

- 若要瞭解是否 AWS AppSync 支援這些功能，請參閱如何與 IAM AWS AppSync 搭配使用。
- 若要了解如何提供對您所擁有資源 AWS 帳戶 的存取權，請參閱《IAM 使用者指南》中您擁有的另一 AWS 帳戶 個 IAM 使用者提供存取權限。
- 若要了解如何將資源存取權提供給第三方 AWS 帳戶，請參閱 IAM 使用者指南中的提供第三方 AWS 帳戶 擁有的存取權。
- 若要了解如何透過聯合身分提供存取權，請參閱 IAM 使用者指南中的將存取權提供給在外部進行身分驗證的使用者 (聯合身分)。
- 若要了解使用角色和資源型政策進行跨帳戶存取之間的差異，請參閱 IAM 使用者指南中的 IAM 角色與資源型政策的差異。

使用記錄 AWS AppSync API 呼叫 AWS CloudTrail

AWS AppSync 與 (提供中的使用者 AWS CloudTrail、角色或服務所採取的動作記錄) 的 AWS 服務整合 AWS AppSync。 CloudTrail 擷取 AWS AppSync 做為事件的 API 呼叫。擷取的呼叫包括來自 AWS AppSync 主控台的呼叫和 AWS AppSync API 作業的程式碼呼叫。如果您建立追蹤，您可以啟用持續交付 CloudTrail 事件到 Amazon S3 儲存貯體，包括 AWS AppSync. 如果您未設定追蹤，您仍然可以在 [事件歷程記錄] 中檢視 CloudTrail 主控台中最近的事件。使用收集的資訊 CloudTrail，您可以判斷提出的要求 AWS AppSync、提出要求的 IP 位址、提出要求的人員、提出要求的時間，以及其他詳細資訊。

若要進一步了解 CloudTrail，請參閱使AWS CloudTrail 用者指南。

AWS AppSync 中的資訊 CloudTrail

CloudTrail 在您創建 AWS 帳戶時，您的帳戶已啟用。當活動發生在中時 AWS AppSync，該活動會與事件歷史記錄中的其他 AWS 服務 CloudTrail 事件一起記錄在事件中。您可以在帳戶中查看，搜索和下載最近的事 AWS 件。如需詳細資訊，請參閱使用 CloudTrail 事件歷程記錄檢視事件。

如需 AWS 帳戶中持續記錄事件 (包括的事件) AWS AppSync，請建立追蹤。追蹤可 CloudTrail 將日誌檔交付到 Amazon S3 儲存貯體。根據預設，當您在主控台中建立追蹤時，追蹤會套用至所有 AWS 區域。追蹤記錄來自 AWS 分區中所有區域的事件，並將日誌檔傳送到您指定的 Amazon S3 儲存貯體。此外，您還可以設定其他 AWS 服務，以進一步分析 CloudTrail 記錄中收集的事件資料並採取行動。如需詳細資訊，請參閱下列內容：

- 建立追蹤的概觀
- CloudTrail 支援的服務與整合
- 設定 Amazon SNS 通知 CloudTrail
- 從多個區域接收 CloudTrail 日誌文件並從多個帳戶接收 CloudTrail 日誌文件

AWS AppSync 支持記錄通過 AWS AppSync API 進行的調用。目前，對您的 API 的調用以及對解析器進行的調用都不會被 AWS AppSync 登錄。 CloudTrail

每一筆事件或日誌專案都會包含產生請求者的資訊。身分資訊可協助您判斷下列事項：

- 要求是使用根使用者登入資料還是 AWS Identity and Access Management (IAM) 使用者登入資料提出。
- 提出該請求時，是否使用了特定角色或聯合身分使用者的暫時安全憑證。
- 請求是否由其他 AWS 服務提出。

如需詳細資訊，請參閱CloudTrail 使 userIdentity 元素。

瞭解 AWS AppSync 記錄檔項目

追蹤是一種組態，可讓事件以日誌檔的形式傳遞到您指定的 Amazon S3 儲存貯體。 CloudTrail 記錄檔包含一或多個記錄項目。事件代表來自任何來源的單一請求，包括有關請求的操作，動作的日期和時間，請求參數等信息。 CloudTrail 日誌文件不是公共 API 調用的有序堆棧跟踪，因此它們不會以任何特定順序顯示。

下列範例顯示示範透過 AWS AppSync 主控台`GetGraphqlApi`執行動作的 CloudTrail 記錄項目：

```
{
    "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"
}
```

下列範例顯示示範透過以下方式`CreateApikey`執行之動作的 CloudTrail 記錄項目 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"
}
```


安全性最佳做法 AWS AppSync

保護安全不僅僅 AWS AppSync 是打開幾個槓桿或設置日誌記錄。以下各節討論的安全性最佳作法會因您使用服務的方式而有所不同。

瞭解驗證方法

AWS AppSync 提供多種方式來驗證您的使用者對您的 GraphQL API。每種方法在安全性、可稽核性和可用性方面都有權衡。

以下是可用的一般驗證方法：

- Amazon Cognito 使用者集區可讓您的 GraphQL API 使用使用者屬性進行更精細的存取控制和篩選。
- API 令牌的生命週期有限，適用於自動化系統，例如持續集成系統以及與外部 API 的集成。
- AWS Identity and Access Management (IAM) 適用於在您管理的 AWS 帳戶.
- OpenID Connect 允許您使用 OpenID Connect 協議控制和聯合訪問。

如需中的驗證和授權的詳細資訊 AWS AppSync，請參閱授權與驗證。

針對 HTTP 解析器使用 TLS

使用 HTTP 解析器時，請務必盡可能使用 TLS 安全 (HTTPS) 連線。如需 AWS AppSync 信任之 TLS 憑證的完整清單，請參閱憑證授權機構 (CA)AWS AppSyncHTTPS 端點。

盡可能使用權限最少的角色

使用解析器 (例如 DynamoDB 解析器) 時，請使用可為您的資源提供最嚴格檢視的角色，例如 Amazon DynamoDB 表格。

IAM 政策最佳做法

以身分識別為基礎的政策會決定某人是否可以建立、存取或刪除您帳戶中的 AWS AppSync 資源。這些動作可能會讓您的 AWS 帳戶產生費用。當您建立或編輯身分型政策時，請遵循下列準則及建議事項：

- 開始使用 AWS 受管原則並邁向最低權限權限 — 若要開始授與使用者和工作負載的權限，請使用可授與許多常見使用案例權限的AWS 受管理原則。它們在您的 AWS 帳戶. 建議您透過定義特定於您使用案例的 AWS 客戶管理政策，進一步降低使用權限。如需更多資訊，請參閱 IAM 使用者指南中的 AWS 受管政策或任務職能的AWS 受管政策。
- 套用最低許可許可 – 設定 IAM 政策的許可時，請僅授予執行任務所需的權限。為實現此目的，您可以定義在特定條件下可以對特定資源採取的動作，這也稱為最低權限許可。如需使用 IAM 套用許可的更多相關資訊，請參閱 IAM 使用者指南中的 IAM 中的政策和許可。
- 使用 IAM 政策中的條件進一步限制存取權 – 您可以將條件新增至政策，以限制動作和資源的存取。例如，您可以撰寫政策條件，指定必須使用 SSL 傳送所有請求。您也可以使用條件來授與對服務動作的存取權 (如透過特定) 使用這些動作 AWS 服務，例如 AWS CloudFormation。如需更多資訊，請參閱 IAM 使用者指南中的 IAM JSON 政策元素：條件。

- 使用 IAM Access Analyzer 驗證 IAM 政策，確保許可安全且可正常運作 – IAM Access Analyzer 驗證新政策和現有政策，確保這些政策遵從 IAM 政策語言 (JSON) 和 IAM 最佳實務。IAM Access Analyzer 提供 100 多項政策檢查及切實可行的建議，可協助您編寫安全且實用的政策。如需更多資訊，請參閱 IAM 使用者指南中的 IAM Access Analyzer 政策驗證。
- 需要多因素身份驗證 (MFA) — 如果您的案例需要 IAM 使用者或根使用者 AWS 帳戶，請開啟 MFA 以獲得額外的安全性。若要在呼叫 API 作業時請求 MFA，請將 MFA 條件新增至您的政策。如需更多資訊，請參閱 IAM 使用者指南中的設定 MFA 保護的 API 存取。

如需 IAM 中最佳實務的相關資訊，請參閱 IAM 使用者指南中的 IAM 安全最佳實務。

解析器參考 () JavaScript

以下各節說明APPSYNC_JS執行階段和 JavaScript 解析器。

主題

JavaScript 解析器概述

AWS AppSync 可讓您透過對資料來源執行作業來回應 GraphQL 要求。對於您想要執行查詢、變異或訂閱的每個 GraphQL 欄位，都必須附加解析器。

解析器是 GraphQL 和數據源之間的連接器。他們會說明AWS AppSync 如何將傳入的 GraphQL 要求轉譯成後端資料來源的指示，以及如何將來自該資料來源的回應轉換回 GraphQL 回應。使用 AWS AppSync，您可以使用編寫解析器 JavaScript並在 AWS AppSync （APPSYNC_JS）環境中運行它們。

AWS AppSync 可讓您撰寫由管線中多個AWS AppSync 函數組成的單位解析器或管線解析器。

支援的執行階段

AWS AppSync JavaScript 執行階段提供程式 JavaScript 庫、公用程式和功能的子集。如需APPSYNC_JS執行階段支援的特性和功能的完整清單，請參閱解析器和函數的JavaScript 執行階段功能。

單位解析器

單元解析器由定義對數據源執行的請求和響應處理程序的代碼組成。請求處理程序將上下文對象作為參數，並返回用於調用數據源的請求有效負載。響應處理程序從數據源接收有效負載，並帶有執行請求的結果。回應處理常式會將有效負載轉換成 GraphQL 回應，以解析 GraphQL 欄位。在以下範例中，解析器會從 DynamoDB 資料來源擷取項目：

```
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 管道解析器的剖析

管道解析器由定義請求和響應處理程序的代碼以及函數列表組成。每個函數都有一個請求和響應處理程序，它對數據源執行。由於管線解析程式委派執行至函數清單，因此不會連結至任何資料來源。單位解析程式和函數是對資料來源執行操作的基本元素。

管線解析程式要求處理常式

管線解析器的請求處理程序（之前的步驟）允許您在運行定義的函數之前執行一些準備邏輯。

函數清單

管道解析程式將會依序執行的函數清單。管道解析器請求處理程序評估結果可用於第一個函數。`ctx.prev.result`每個函數評估結果可用於下一個函數`ctx.prev.result`。

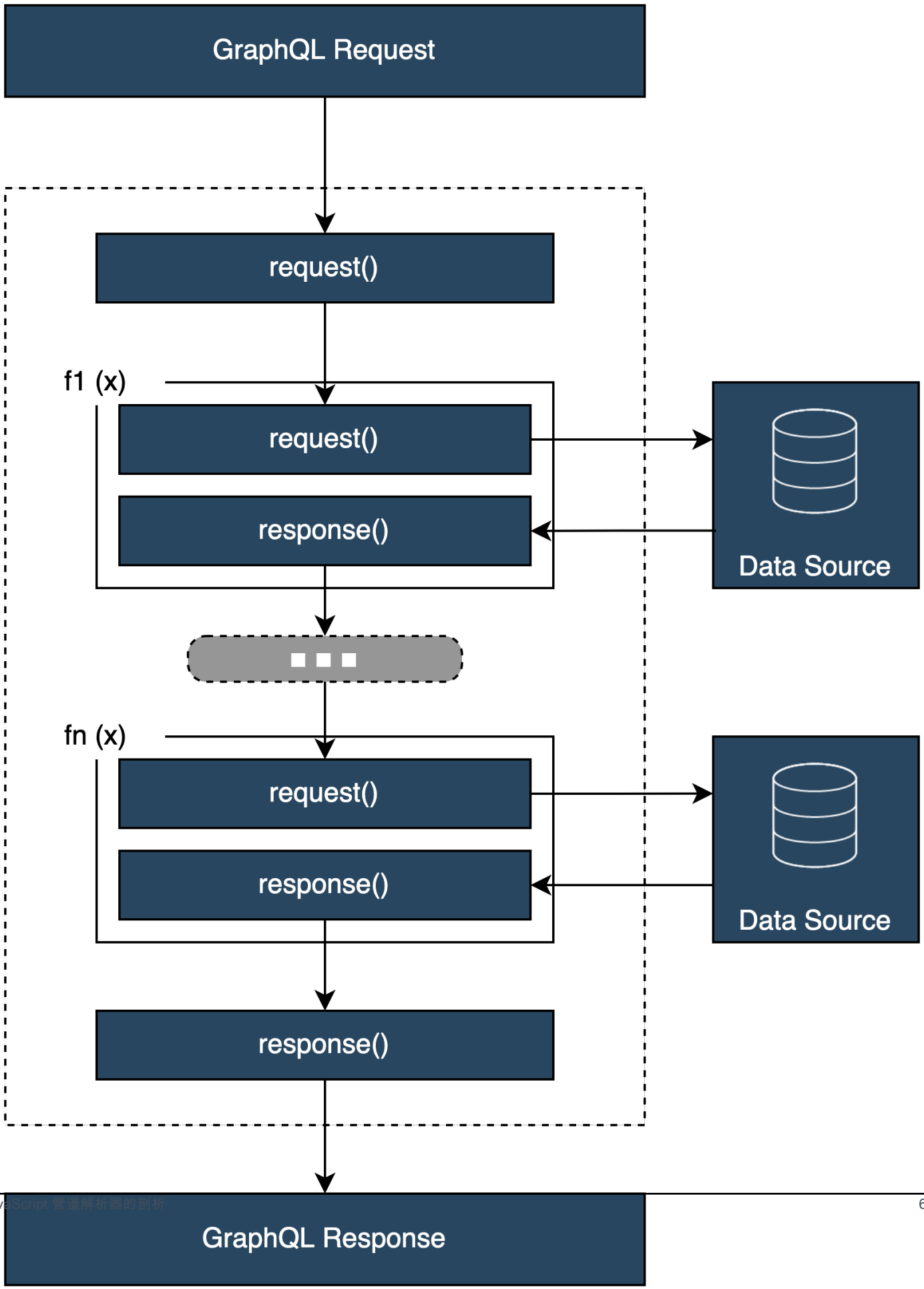
管道解析器響應處理程序

管線解析器的響應處理程序允許您執行從最後一個函數的輸出到預期的 GraphQL 字段類型的一些最終邏輯。函數清單中最後一個函數的輸出可在管線解析程式回應處理常式中以或形`ctx.prev.result`式取得。`ctx.result`

執行流程

給定由兩個函數組成的管線解析器，下面的列表代表調用解析器時的執行流程：

1. 管線解析程式要求處理常式
2. 函數 1：函數要求處理常式
3. 第 1 個函數：資料來源呼叫
4. 函數 1：函數響應處理程序
5. 函數 2：函數要求處理常式
6. 第 2 個函數：資料來源呼叫
7. 函數 2：函數響應處理程序
8. 管道解析器響應處理程序


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

實用的APPSYNC_JS執行階段內建

下列公用程式可在您使用管道解析程式時提供協助。

存儲

存儲是每個解析器和函數請求和響應處理程序中可用的對象。相同的存儲實例通過單個解析器運行存在。這意味著您可以使用存儲在請求和響應處理程序之間以及管道解析器中的函數之間傳遞任意數據。您可以像常規 JavaScript 對象一樣測試存儲。

上一頁結果

ctx.prev.result 會顯示管道先前執行操作的結果。如果先前的作業是管線解析程式要求處理常式，ctx.prev.result則可供鏈結中的第一個函數使用。如果先前操作是第一個函數，則 ctx.prev.result 會顯示第一個函數的輸出，並將資料提供給管道中的第二個函數。如果上一個操作是最後一個函數，則ctx.prev.result表示最後一個函數的輸出，並可供管線解析器響應處理程序使用。

實用程序錯誤

util.error 公用程式非常適合用來擲出欄位錯誤。在函數請求或響應處理程序中使用util.error會立即引發字段錯誤，從而防止後續函數被執行。有關更多詳細信息和其他util.error簽名，請訪問解析器和功能的JavaScript運行時功能。

附加工具

util.appendError類似於util.error()，主要區別在於它不會中斷處理程序的評估。相反，它表示該字段出現錯誤，但允許評估處理程序並因此返回數據。在函數中使用 util.appendError 並不會中斷管道的執行流程。有關更多詳細信息和其他util.error簽名，請訪問解析器和功能的 JavaScript 運行時功能。

運行時間. 早返回

該runtime.earlyReturn函數允許您從任何請求函數過早返回。在解析器請求處理程序中使runtime.earlyReturn用將從解析器返回。從AWS AppSync函數請求處理程序調用它將從函數返回，並將繼續運行到管道中的下一個函數或解析器響應處理程序。

撰寫管線解析器

管道解析器還具有一個請求和一個響應處理程序，圍繞管道中的函數運行：其請求處理程序在第一個函數的請求之前運行，並且其響應處理程序在最後一個函數的響應之後運行。解析器請求處理程序可以設置數據，以供管道中的函數使用。解析程式回應處理常式負責傳回對應至 GraphQL 欄位輸出類型的資

料。在以下範例中，解析器要求處理常式會定義allowedGroups；傳回的資料應屬於其中一個群組。解析器的函數可以使用此值來請求數據。解析器的響應處理程序進行最終檢查並過濾結果，以確保僅返回屬於允許組的項目。

```
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;
}
```

寫入AWS AppSync函數

AWS AppSync 函數使您能夠編寫可以在模式中跨多個解析器重複使用的通用邏輯。例如，您可以有一個名為的AWS AppSync 函數QUERY_ITEMS，負責查詢 Amazon DynamoDB 資料來源中的項目。對於您想要查詢項目的解析器，只需將函數添加到解析器的管道中，並提供要使用的查詢索引即可。邏輯不必重新實現。

撰寫程式碼

假設您想要在名為的欄位上附加管線解析器，getPost(id:ID!)該欄位會使用下列 GraphQL 查詢從 Amazon DynamoDB 資料來源傳回Post類型：

```
getPost(id:1){
```

```
    id
    title
    content
}
```

首先，Query.getPost使用下面的代碼附加一個簡單的解析器。這是簡單的解析器代碼的一個例子。沒有在請求處理程序中定義的邏輯，和響應處理程序只是返回最後一個函數的結果。

```
/**
 * 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
}
```

接下來，定義從GET_ITEM資料來源擷取網站項目的函數：

```
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
}
```

如果要求期間發生錯誤，函式的回應處理常式會附加錯誤，該錯誤會傳回至 GraphQL 回應中的呼叫用戶端。將GET_ITEM函數添加到您的解析器函數列表中。當您執行查詢時，GET_ITEM函數的要求處理常式會使用 DynamoDB 模組提供AWS AppSync的應用程式來建立使用id做為金鑰的DynamoDBGetItem要求。 ddb.get({ key: { id } })生成適當的GetItem操作：

```
{
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "1" }
    }
}
```

AWS AppSync使用請求從 Amazon DynamoDB 擷取資料。一旦返回數據，它是由GET_ITEM函數的響應處理程序，該處理程序檢查錯誤，然後返回結果。

```
{
  "result" : {
    "id": 1,
    "title": "hello world",
    "content": "<long story>"
  }
}
```

最後，解析器的響應處理程序直接返回結果。

使用錯誤

如果在請求期間在函數中發生錯誤，則該錯誤將在中的函數響應處理程序中提供`ctx.error`。您可以使用公用`util.appendError`程式將錯誤附加到 GraphQL 回應中。您可以使用存儲器將錯誤提供給管道中的其他函數。請參閱以下範例：

```
/**
 * 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;
}
```

公用程式

AWS AppSync提供了兩個庫，以幫助解析器與運行時的開發：APPSYNC_JS

- `@aws-appsync/eslint-plugin`-在開發過程中快速捕獲和修復問題。
- `@aws-appsync/utils`-在程式碼編輯器中提供型別驗證和自動完成功能。

配置外掛程式

eSlint 是一種工具，可以靜態分析您的代碼以快速找到問題。您可以將 eSLint 作為持續集成管道的一部分運行。 `@aws-appsync/eslint-plugin`是一個 eSlint 插件，在利用運行時捕獲代碼中的無效語法。APPSYNC_JS該插件使您可以在開發過程中快速獲得有關代碼的反饋，而無需將更改推送到雲中。

`@aws-appsync/eslint-plugin`提供兩個您可以在開發期間使用的規則集。

「插件：@aws-appsync/base」 配置了一組可以在項目中利用的基本規則：

規則	描述
無異步	不支援非同步處理程序和承諾。
不等待	不支援非同步處理程序和承諾。
沒有類別	不支援類別。
不適用	for不支援 (支援的for-in和for-of除外)
不繼續	不支援 continue。
無發電機	不支援產生器。
無收益率	不支援 yield。
無標籤	不支援標示。
沒有這個	this不支援關鍵字。
不嘗試	不支持嘗試/catch 結構。
沒有時間	雖然不支持循環。
no-disallowed-unary-operators	++--、和一~元運算子是不允許的。
no-disallowed-binary-operators	不允許instanceof 運算子。
不承諾	不支援非同步處理程序和承諾。

「插件：@aws-appsync /推薦」 提供了一些額外的規則，但也需要您將 TypeScript 配置添加到項目中。

規則	描述
不遞歸	遞歸函數調用是不允許的。
no-disallowed-methods	某些方法是不允許的。如需完整支援的內建函數集，請參閱參考資料。

規則	描述
no-function-passing	不允許將函數作為函數參數傳遞給函數。
no-function-reassign	無法重新指派函數。
no-function-return	函數不能是函數的返回值。

若要將外掛程式新增至您的專案，請按照 eSLint 入門中的安裝和使用步驟進行操作。然後，使用項目包管理器（例如 npm，yarn 或 pnpm）在項目中安裝插件：

```
$ npm install @aws-appsync/eslint-plugin
```

在您的.eslintrc.{js,yml,json}文件中，添加 「插件：@aws-appsync/base」 或 「插件：@aws-應用程序同步/推薦」的屬性。extends下面的代碼片段是一個基本的示例.eslintrc配置 JavaScript：

```
{
  "extends": ["plugin:@aws-appsync/base"]
}
```

若要使用 「外掛程式：@aws-appsync/建議設定」 規則集，請安裝必要的相依性：

```
$ npm install -D @typescript-eslint/parser
```

然後，創建一個.eslintrc.js文件：

```
{
  "parser": "@typescript-eslint/parser",
  "parserOptions": {
    "ecmaVersion": 2018,
    "project": "./tsconfig.json"
  },
  "extends": ["plugin:@aws-appsync/recommended"]
}
```

捆綁和源映射 TypeScript

利用程式庫並捆綁您的程式碼

在您的解析器和函數代碼中，只要它們符合要求，您就可以利用自定義庫和外部庫。APPSYNC_JS這使得可以在應用程序中重複使用現有代碼。要使用由多個文件定義的庫，您必須使用捆綁工具（例如 esbuild ）將代碼合併到單個文件中，然後將其保存到AWS AppSync解析器或函數中。

捆綁代碼時，請記住以下幾點：

- APPSYNC_JS僅支持電子印刷稿模塊（ESM）。
- `@aws-appsync/*`模組已整合到程式碼中，APPSYNC_JS且不應與程式碼捆綁在一起。
- 執行APPSYNC_JS階段環境與 NodeJS 類似，因為該程式碼不會在瀏覽器環境中執行。
- 您可以包括一個可選的源映射。但是，請勿包含來源內容。

  若要深入瞭解來源對應，請參閱使用來源對應。

例如，若要捆綁位於的解析器代碼`src/appsync/getPost.resolver.js`，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
```

建立您的程式碼並使用 TypeScript

TypeScript是由 Microsoft 開發的編程語言，提供了所有 JavaScript的功能與 TypeScript 打字系統一起。您可以使用編寫類型安全的代碼，並在構建時 catch 錯誤和錯誤，然後再 TypeScript 將代碼保存到。AWS AppSync該`@aws-appsync/utils`軟件包是完全類型的。

APPSYNC_JS執行階段不 TypeScript 直接支援。您必須先將 TypeScript 程式碼轉譯成APPSYNC_JS執行階段支援的 JavaScript 程式碼，然後再將程式碼儲存至AWS AppSync。您可 TypeScript 以使

用在本機整合式開發環境 (IDE) 中撰寫程式碼，但請注意，您無法在AWS AppSync主控台中建立 TypeScript 程式碼。

要開始使用，請確保您已TypeScript安裝在項目中。然後，設定您的 TypeScript 轉譯設定，以使用 TS Config 搭配APPSYNC_JS執行階段使用。以下是您可以使用的基本`tsconfig.json`文件的示例：

```
// tsconfig.json
{
  "compilerOptions": {
    "target": "esnext",
    "module": "esnext",
   "noEmit": true,
   "moduleResolution": "node",
  }
}
```

然後，您可以使用 esbuild 之類的捆綁工具來編譯和捆綁代碼。例如，假設您的AWS AppSync程式碼位於的專案`src/appsync`，您可以使用下列命令來編譯和捆綁您的程式碼：

```
$ 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 代碼

您可以使用 Amplify CLI 產生結構描述的類型。從`schema.graphql`文件所在的目錄中，運行以下命令並查看提示以配置代碼代碼：

```
$  npx @aws-amplify/cli codegen add
```

要在某些情況下（例如，當您的模式更新時）重新生成代碼，請運行以下命令：

```
$ npx @aws-amplify/cli codegen
```

然後，您可以在解析器代碼中使用生成的類型。例如，給定以下模式：

```
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函數中使用生成的類型：

```
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
}
```

在中使用泛型 TypeScript

您可以將泛型與幾種提供的類型一起使用。例如，下面的代碼片段是一個Todo類型：

```
export type Todo = {
  __typename: "Todo",
  id: string,
  title: string,
  description?: string | null,
};
```

您可以為使用的訂閱編寫解析器。Todo在 IDE 中，類型定義和自動完成提示將引導您正確使用toSubscriptionFilter轉換公用程式：

```
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
}
```

襯你的捆綁包

您可以通過導入esbuild-plugin-eslint插件自動 lint 您的捆綁包。然後，您可以提供啟用 eslint 功能的plugins值來啟用它。以下是在名build.mjs為的文件中使用 esbuild JavaScript API 的代碼片段：

```
/* 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 })],
})
```

使用來源地圖

您可以提供一個內聯源映射（sourcemap）與您的 JavaScript 代碼。當您捆綁 JavaScript 或 TypeScript 代碼並希望在日誌和運行時 JavaScript 錯誤消息中查看對輸入源文件的引用時，源映射非常有用。

您sourcemap必須出現在代碼的末尾。它由遵循以下格式的單一註釋行定義：

```
//# sourceMappingURL=data:application/json;base64,<base64 encoded string>
```

範例如下：

```
//# sourceMappingURL=data:application/
json;base64,ewogICJ2ZXJzaW9uIjogMywKICAic291cmNlcyI6IFsibGliLmpzIiwgImNvZGUuanMiXSwKICAibWFwcGl
```

可以使用 esbuild 創建源映射。下面的示例顯示了如何在構建和捆綁代碼時使用 esbuild JavaScript API 包含內聯源映射：

```
/* 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 })],
})
```

特別是，sourcemap和選sourcesContent項指定源映射應該在每個構建的末尾行添加，但不應包括源內容。作為慣例，我們建議不要在您的sourcemap. 您可以通過sources-content將false設置為在 esbuild 中禁用此功能。

為了說明源映射如何工作，請查看以下示例，其中解析器代碼引用幫助程序庫中的幫助程序函數。該代碼在解析器代碼和幫助程序庫中包含日誌語句：

。 /src /默認.解決器.ts（你的解析器）

```
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 /幫助者 .ts（一個輔助文件）

```
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..')
}
```

當您構建並捆綁解析器文件時，解析器代碼將包含內聯源映射。當您的解析程式執行時，記錄檔中會顯示下列項目： CloudWatch

查看 CloudWatch 日誌中的條目，您會注意到這兩個文件的功能已捆綁在一起，並且正在同時運行。每個檔案的原始檔案名稱也會清楚地反映在記錄中。

測試

在將代碼保存到解析器或函數之前，您可以使用 EvaluateCode API 命令使用模擬數據遠程測試解析器和函數處理程序。若要開始使用命令，請確定您已將appsync:evaluatecode權限新增至原則。例如：

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

您可以使用 AWSCLI 或 AWSSDK 來利用命令。例如，若要使用 CLI 測試程式碼，只要指向您的檔案、提供內容，然後指定您要評估的處理常式即可：

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

響應包含一個包evaluationResult含處理程序返回的有效負載。它還包含一個logs對象，用於保存您的處理程序在評估期間生成的日誌列表。這可讓您輕鬆偵錯程式碼執行，並查看評估相關資訊，以協助進行疑難排解。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\""
```

```
    ]
}
```

評估結果可以解析為 JSON，這給出了：

```
{
  "operation": "PutItem",
  "key": {
    "id": {
      "S": "record-id"
    }
  },
  "attributeValues": {
    "owner": {
      "S": "John doe"
    },
    "expectedVersion": {
      "N": 2
    },
    "authorId": {
      "S": "Sammy Davis"
    }
  }
}
```

使用 SDK，您可以輕鬆地從測試套件中合併測試以驗證代碼的行為。我們這裡的例子使用 Jest 測試框架，但任何測試套件都可以工作。下面的代碼片段顯示了一個假設的驗證運行。請注意，我們希望評估響應是有效的 JSON，因此我們使JSON.parse用從字符串響應中檢索 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)
})
```

這會產生以下結果：

```
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
```

從 VTL 移轉到 JavaScript

AWS AppSync 允許您使用 VTL 或為解析器和函數編寫業務邏輯。 JavaScript使用這兩種語言，您可以編寫邏輯來指示AWS AppSync 服務如何與資料來源互動。使用 VTL，您可以撰寫必須評估為有效 JSON 編碼字串的對應範本。使用時 JavaScript，您可以撰寫傳回物件的要求和回應處理常式。您不會傳回 JSON 編碼的字串。

例如，使用下列 VTL 對應範本來取得 Amazon DynamoDB 項目：

```
{
    "operation": "GetItem",
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
    }
}
```

該實用程序`$util.dynamodb.toDynamoDBJson`返回 JSON 編碼的字符串。如果設定`$ctx.args.id`為`<id>`，則範本會評估為有效的 JSON 編碼字串：

```
{
    "operation": "GetItem",
    "key": {
        "id": {"S": "<id>"},
    }
```

```
}
```

使用時 JavaScript，您不需要在代碼中打印出原始 JSON 編碼的字符串，並且使用不需要的實toDynamoDBJson用程序。上面的映射模板的一個等效例子是：

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: {id: util.dynamodb.toDynamoDB(ctx.args.id)}
  };
}
```

另一種方法是使用util.dynamodb.toMapValues，這是處理值對象的推薦方法：

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'GetItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
}
```

這評估為：

```
{
  "operation": "GetItem",
  "key": {
    "id": {
      "S": "<id>"
    }
  }
}
```

> **Note**
>
> 我們建議您搭配使用 DynamoDB 資料來源模組：
>
> ```
> import * as ddb from '@aws-appsync/utils/dynamodb'
> ```

```
export function request(ctx) {
 ddb.get({ key: { id: ctx.args.id } })
}
```

另一個範例是，使用下列對應範本將項目放入 Amazon DynamoDB 資料來源中：

```
{
    "operation" : "PutItem",
    "key" : {
        "id": $util.dynamodb.toDynamoDBJson($util.autoId()),
    },
    "attributeValues" : $util.dynamodb.toMapValuesJson($ctx.args)
}
```

評估時，此對應範本字串必須產生有效的 JSON 編碼字串。使用時 JavaScript，您的代碼直接返回請求對象：

```
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),
  };
}
```

其評估為：

```
{
  "operation": "PutItem",
  "key": {
    "id": { "S": "2bff3f05-ff8c-4ed8-92b4-767e29fc4e63" }
  },
  "attributeValues": {
    "firstname": { "S": "Shaggy" },
    "age": { "N": 4 }
  }
}
```

> **Note**
>
> 我們建議您搭配使用 DynamoDB 資料來源模組：
>
> ```
> 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 })
> }
> ```

透過 Lambda 資料來源在直接資料來源存取和代理之間進行選擇

透過AWS AppSync 和APPSYNC_JS執行階段，您可以撰寫自己的程式碼，藉由使用AWS AppSync 函數存取資料來源來實作自訂商務邏輯。這可讓您輕鬆地直接與 Amazon DynamoDB、Aurora 無伺服器、服務、HTTP API 和其他 OpenSearch 服務等資料來源互動，而無需部署其他運算AWS服務或基礎設施。AWS AppSync 也可以透過設定 Lambda 資料來源，輕鬆與AWS Lambda函數互動。Lambda 資料來源可讓您使用AWS Lambda的完整功能來執行複雜的商業邏輯，以解決 GraphQL 請求。在大多數情況下，直接連接到其目標數據源的AWS AppSync 函數將提供您需要的所有功能。如果您需要實作APPSYNC_JS執行階段不支援的複雜商業邏輯，您可以使用 Lambda 資料來源做為 Proxy 來與目標資料來源互動。

	直接資料來源整合	作為代理的 Lambda 資料來源
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)	任何支援的執 Lambda 階段
Maximum size of code	每個函數 32,000 個字元 AWS AppSync	每個 Lambda 50 MB (壓縮，用於直接上傳)
External modules	有限-僅支援的功能	是
Call any AWS service	是-使用 AWS AppSync HTTP 資料來源	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-使用佈建並行
Auto-scaling	是的-透明地 AWS AppSync	是-如 Lambda 中的設定
Pricing	不收取其他費用	按使用 Lambda 收費

AWS AppSync 直接與其目標資料來源整合的函數非常適合下列使用案例：

- 與 Amazon DynamoDB、Aurora 無伺服器和服務互動 OpenSearch
- 與 HTTP API 進行交互並傳遞傳入的標頭
- 使用 HTTP 資料來源與AWS服務互AWS AppSync動 (使用提供的資料來源角色自動簽署要求)
- 在存取資料來源之前實作存取控制
- 在完成請求之前實現檢索到的數據的過濾
- 通過在解析器管道中順序執行AWS AppSync功能來實現簡單的協調流程
- 控制查詢和突變中的快取和訂閱連線。

AWS AppSync 使用 Lambda 資料來源做為代理的函數非常適合下列使用案例：

- 使用速度範本語 JavaScript 言 (VTL) 以外的語言
- 調整和控制 CPU 或記憶體以最佳化效能
- 匯入第三方程式庫或需要不支援的功能 APPSYNC_JS
- 發出多個網絡請求和/或獲取文件系統訪問以完成查詢
- 使用批次設定批次處理請求。

解析器上下文對象引用

AWS AppSync定義了一組用於處理請求和響應處理程序的變量和函數。這樣可讓使用 GraphQL 對資料進行邏輯運算更簡單。本文件說明這些功能並提供範例。

正在存取 `context`

該context請求和響應處理程序的參數是一個對象，用於保存解析器調用的所有上下文信息。其結構如下：

```
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**
>
> 你會經常發現，context對象被稱為ctx。

中的每個欄位context對象的定義如下：

`context` 欄位

`arguments`

包含此欄位所有 GraphQL 引數的映射。

`identity`

包含有關發起人資訊的物件。如需此欄位結構的詳細資訊，請參閱身分。

source

包含父欄位解析度的映射。

stash

存儲是在每個解析器和函數處理程序中可用的對象。通過單個解析器運行，同樣的存儲對象存在。這意味著您可以使用存儲在請求和響應處理程序之間以及管道解析器中的函數之間傳遞任意數據。

> **Note**
>
> 您無法刪除或取代整個儲存區，但您可以新增、更新、刪除和讀取儲存區的屬性。

您可以修改下列其中一個程式碼範例，將項目新增至儲存庫：

```
//Example 1
ctx.stash.newItem = { key: "something" }

//Example 2
Object.assign(ctx.stash, {key1: value1, key2: value})
```

您可以通過修改以下代碼從存儲中刪除項目：

```
delete ctx.stash.key
```

result

此解析程式結果的容器。此欄位僅適用於回應處理常式。

例如，如果您正在解決author下列查詢欄位：

```
query {
    getPost(id: 1234) {
        postId
        title
        content
        author {
            id
            name
        }
```

```
    }
}
```

然後全部context在評估響應處理程序時可用變量：

```
{
  "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

在管道解析器中執行任何先前操作的結果。

如果之前的操作是管道解析器的請求處理程序，則ctx.prev.result代表評估結果，並可供管線中的第一個函數使用。

如果先前的操作是第一個功能，那麼ctx.prev.result代表第一個函數回應處理常式的評估結果，並可供管線中的第二個函數使用。

如果上一個操作是最後一個功能，那麼ctx.prev.result代表最後一個函數的評估結果，並可供管線解析程式的回應處理常式使用。

info

包含有關 GraphQL 請求資訊的物件。如需此欄位的結構，請參閱資訊。

Identity

包含有關發起人資訊的 `identity` 區段。此區段的形態取決於您 AWS AppSync API 的授權類型。

有關更多信息AWS AppSync安全性選項，請參閱授權與驗證。

`API_KEY` 授權

該`identity`未填入欄位。

`AWS_LAMBDA` 授權

該`identity`具有以下形式：

```
type AppSyncIdentityLambda = {
  resolverContext: any;
};
```

該`identity`包含`resolverContext`索引鍵，包含相同的`resolverContext`由 Lambda 函數授權請求所傳回的內容。

`AWS_IAM` 授權

該`identity`具有以下形式：

```
type AppSyncIdentityIAM = {
  accountId: string;
  cognitoIdentityPoolId: string;
  cognitoIdentityId: string;
  sourceIp: string[];
  username: string;
  userArn: string;
  cognitoIdentityAuthType: string;
  cognitoIdentityAuthProvider: string;
};
```

`AMAZON_COGNITO_USER_POOLS` 授權

該`identity`具有以下形式：

```
type AppSyncIdentityCognito = {
  sourceIp: string[];
  username: string;
```

```
  groups: string[] | null;
  sub: string;
  issuer: string;
  claims: any;
  defaultAuthStrategy: string;
};
```

每個欄位的定義如下：

accountId

該AWS來電者的帳戶 ID。

claims

使用者擁有的宣告。

cognitoIdentityAuthType

經身分驗證或未經身分驗證 (根據身分類型)。

cognitoIdentityAuthProvider

以逗號分隔的外部身分識別提供者資訊清單，用來取得用來簽署要求的認證。

cognitoIdentityId

來電者的亞馬遜認可身份識別碼。

cognitoIdentityPoolId

與呼叫者相關聯的 Amazon Cognito 身分集區識別碼。

defaultAuthStrategy

此發起人 (ALLOW 或 DENY) 的預設授權策略。

issuer

字符發行者。

sourceIp

來電者的來源 IP 位址AWS AppSync接收。如果請求不包含`x-forwarded-for`標頭，來源 IP 值只包含來自 TCP 連線的單一 IP 位址。如果要求包含 `x-forwarded-for` 標頭，則來源 IP 除了有 TCP 連線的 IP 地址外，也將有 `x-forwarded-for` 標頭中 IP 地址的清單。

sub

已驗證使用者的 UUID。

user

IAM 使用者。

userArn

IAM 使用者的亞馬遜資源名稱 (ARN)。

username

已驗證使用者的使用者名稱。如果是 AMAZON_COGNITO_USER_POOLS 授權，使用者名稱的值是屬性 cognito:username 的值。在的情況下AWS_IAM授權，值用戶名是的價值AWS使用者主體。如果您使用 IAM 授權搭配從 Amazon Cognito 身分集區提供的登入資料，建議您使用cognitoIdentityId。

存取要求標頭

AWS AppSync支持從客戶端傳遞自定義標題，並通過使用 GraphQL 解析器訪問它們ctx.request.headers。然後，您可以將標頭值用於動作，例如將資料插入資料來源或授權檢查。您可以使用單個或多個請求標頭$curl使用命令列中的 API 金鑰，如下列範例所示：

單頭示例

假設您設定使用 custom 值的 nadia 標頭，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
```

可使用 ctx.request.headers.custom 對此進行存取。例如，它可能位於 DynamoDB 的下列程式碼中：

```
"custom": util.dynamodb.toDynamoDB(ctx.request.headers.custom)
```

多個頭的例子

您還可以在單個請求中傳遞多個標頭，並在解析器處理程序中訪問這些標題。例如，如果custom標題設置有兩個值：

```
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
```

然後，您可以將這些存取為陣列，例如 `ctx.request.headers.custom[1]`。

> **Note**
>
> AWS AppSync不會暴露餅乾標題`ctx.request.headers`。

存取要求自訂網域名稱

AWS AppSync支援設定可用來存取您的 GraphQL 和 API 的即時端點的自訂網域。使用自訂網域名稱提出要求時，您可以使用`ctx.request.domainName`。

使用預設 GraphQL 端點網域名稱時，值為`null`。

Info

`info` 區段包含有關 GraphQL 請求的資訊。本節具有以下形式：

```
type Info = {
  fieldName: string;
  parentTypeName: string;
  variables: any;
  selectionSetList: string[];
  selectionSetGraphQL: string;
};
```

每個欄位的定義如下：

`fieldName`

目前正在解析的欄位名稱。

`parentTypeName`

目前正在解析的欄位父類型名稱。

variables

包含傳遞給 GraphQL 請求之所有變數的映射。

selectionSetList

此清單表示 GraphQL 選取範圍中的欄位。別名的欄位僅由別名參考，而不是欄位名稱參考。以下範例詳細說明這一點。

selectionSetGraphQL

此字串表示格式為 GraphQL 結構描述定義語言 (SDL) 的選取範圍。雖然片段不會合併到選集中，但會保留內嵌片段，如下列範例所示。

> **Note**
>
> `JSON.stringify`將不包括`selectionSetGraphQL`和`selectionSetList`在字符串序列化中。您必須直接參考這些屬性。

例如，如果您解析下列查詢的 `getPost` 欄位：

```
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
  }
}
```

然後全部ctx.info處理處理常式時可用的變數可能是：

```
{
  "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僅公開屬於目前類型的欄位。如果目前類型是介面或聯集，則只會顯示屬於該介面的選取欄位。例如，給定以下模式：

```
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
}
```

以下查詢：

```
query {
    node(id: "post1") {
        id
        ... on Post {
            title
        }

        ... on Blog {
            title
        }
    }
}
```

打電話時`ctx.info.selectionSetList`在`Query.node`欄位解析度，僅`id`暴露：

```
"selectionSetList": [
    "id"
]
```

JavaScript 解析器和函數的運行時功能

APPSYNC_JS執行階段環境提供類似於 ECMAScript (ES) 6.0 版的功能。它支持其功能的一個子集，並提供了一些不屬於 ES 規範的其他方法（實用程序）。下列主題列出所有支援的語言功能。

> **Note**
>
> 目前，此參考僅適用於執行階段版本 1.0.0。

主題

支援的執行階段

以下各節說明 APPSYNC_JS 執行階段支援的功能集。

核心功能

支持以下核心功能。

類型

支援下列類型：

- 數字
- 字串
- 布爾值
- objects
- 陣列

- 函數

運營商

支援運算子，包括：

- 標準數學運算子 (+-/%、*、、等)
- 空合併運算符 ( ) ??
- 可選鏈接 ( ) ?.
- 位運算符
- void和typeof運營商

不支援下列運算子：

- 一元運算子 (++--、和~)
- in運算子

> **Note**
>
> 使用Object.hasOwn運算子檢查指定的屬性是否在指定的物件中。

陳述

支援下列陳述式：

- const
- let
- var
- break
- else
- for-in
- for-of
- if
- return

- switch
- 傳播語法

不支援下列項目：

- catch
- continue
- do-while
- finally
- for(initialization; condition; afterthought)

> **Note**
>
> 例外是for-in和for-of表達式，這是支持的。

- throw
- try
- while
- 標記陳述式

文字

支持以下 ES 6 模板文字：

- 多行字符串
- 表達式插值
- 巢狀範本

函數

支援下列函數語法：

- 支持函數聲明。
- ES 6 支持箭頭功能。
- ES 6 其餘參數語法支持。

嚴格模式

函數按預設會在嚴格模式下運作，因此您不需要在函數程式碼中新增 `use_strict` 陳述式。無法對此進行變更。

基本物件

支持 ES 及其功能的以下原始對象。

物件

支援下列物件：

- `Object.assign()`
- `Object.entries()`
- `Object.hasOwn()`
- `Object.keys()`
- `Object.values()`
- `delete`

：字串

支援下列字串：

- `String.prototype.length()`
- `String.prototype.charAt()`
- `String.prototype.concat()`
- `String.prototype.endsWith()`
- `String.prototype.indexOf()`
- `String.prototype.lastIndexOf()`
- `String.raw()`
- `String.prototype.replace()`

> **Note**
>
> 不支援規則運算式。

- `String.prototype.replaceAll()`

> **Note**
>
> 不支援規則運算式。

- `String.prototype.slice()`
- `String.prototype.split()`
- `String.prototype.startsWith()`
- `String.prototype.toLowerCase()`
- `String.prototype.toUpperCase()`
- `String.prototype.trim()`
- `String.prototype.trimEnd()`
- `String.prototype.trimStart()`

數字

支援下列數字：

- `Number.isFinite`
- `Number.isNaN`

內置對象和功能

支持以下功能和對象。

Math (數學)

支援下列數學函數：

- `Math.random()`
- `Math.min()`
- `Math.max()`
- `Math.round()`
- `Math.floor()`

- Math.ceil()

Array (陣列)

支援下列陣列方法：

- 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()不支持參數。

- 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`

主控台

控制台對象可用於調試。在即時查詢執行期間，主控台記錄/錯誤陳述式會傳送至 Amazon Lo CloudWatch gs (如果啟用記錄功能)。在程式碼評估期間`evaluateCode`，會在命令回應中傳回 log 陳述式。

- `console.error()`
- `console.log()`

JSON

支援下列 JSON 方法：

- `JSON.parse()`

> **Note**
> 返回一個空白字符串，如果解析的字符串是無效的 JSON。

- `JSON.stringify()`

函數

- 不支援`apply`、`bind`、和`call`方法。
- 不支援函數建構子。
- 不支援將函數作為引數傳遞。
- 不支援遞迴函數呼叫。

Promise

不支援非同步處理程序，並且不支援承諾。

> **Note**
>
> 中的APPSYNC_JS執行階段不支援網路和檔案系統存取AWS AppSync。 AWS AppSync根據AWS AppSync解析器或AWS AppSync函數發出的請求來處理所有 I/O 操作。

全域變數

支援以下全局常數：

- `NaN`
- `Infinity`
- `undefined`
- `util`
- `extensions`
- `runtime`

錯誤類型

不支援擲回錯誤。`throw`您可以通過使用`util.error()`函數返回一個錯誤。您可以使用`util.appendError`函數在 GraphQL 回應中包含錯誤。

如需詳細資訊，請參閱錯誤公用程式。

內建公用程

此`util`變數包含可協助您處理資料的一般公用程式方法。除非另行指定，否則所有公用程式皆使用 UTF-8 字元集。

編碼公用程式

編碼實用程序列表

`util.urlEncode(String)`

以 `application/x-www-form-urlencoded` 編碼字串的形式傳回輸入字串。

`util.urlDecode(String)`

將 application/x-www-form-urlencoded 編碼的字串解碼回非編碼格式。

`util.base64Encode(string) : string`

將輸入編碼為 base64 編碼字串。

`util.base64Decode(string) : string`

解碼 base64 編碼字串中的資料。

ID 產生公用程式

ID 生成實用程序列表

`util.autoId()`

傳回 128 位元隨機產生的 UUID。

`util.autoUlid()`

返回一個 128 位隨機生成的 ULID（通用唯一的字典排序標識符）。

`util.autoKsuid()`

返回 128 位隨機生成的 KSUID（K 可排序的唯一標識符）基於 62 編碼為長度為 27 的字符串。

錯誤實用程序

錯誤實用程序列表

`util.error(String, String?, Object?, Object?)`

擲回自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。此外`errorType`，還可以指定`data`欄位、`errorInfo`欄位和欄位。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

> **Note**
>
> `data`將根據查詢選集進行篩選。`errorInfo` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

errorInfo將不會根據查詢選集進行篩選。

`util.appendError(String, String?, Object?, Object?)`

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。此外errorType，還可以指定data欄位、errorInfo欄位和欄位。與 `util.error(String, String?, Object?, Object?)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

> **Note**
>
> data將根據查詢選集進行篩選。`errorInfo` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。
> errorInfo將不會根據查詢選集進行篩選。

類型和模式匹配實用程序

類型和模式匹配實用程序列表

`util.matches(String, String) : Boolean`

如果第一個引數中指定的模式與第二個引數中提供的資料相符，則傳回真。模式必須為規則表達式，例如 `util.matches("a*b", "aaaaab")`。此功能是根據模式，您可以參考以取得更詳細的文件。

`util.authType()`

返回一個字符串，描述請求正在使用的多身份驗證類型，返回「IAM 授權」，「用戶池授權」，「打開 ID Connect 授權」或「API 密鑰授權」。

返回值行為實用程序

返回值行為實用程序列表

`util.escapeJavaScript(String)`

返回輸入字符串作為 JavaScript 轉義字符串。

解析器授權實用程序

解析器授權實用程序列表

`util.unauthorized()`

擲回欲解析之欄位的 Unauthorized。在請求或響應映射模板中使用此選項，以確定是否允許調用者解析字段。

內建模組

模塊是APPSYNC_JS運行時的一部分，並提供幫助編寫 JavaScript解析器和函數的實用程序。

DynamoDB 功能

DynamoDB 模組函數可在與 DynamoDB 資料來源互動時提供增強的體驗。您可以使用函數向 DynamoDB 資料來源發出請求，而無需新增類型對應。

使用以下方式匯入模組`@aws-appsync/utils/dynamodb`：

```
// Modules are imported using @aws-appsync/utils/dynamodb
import * as ddb from '@aws-appsync/utils/dynamodb';
```

函數

函數清單

`get<T>(payload: GetInput): DynamoDBGetItemRequest`

> **Tip**
>
> 如需相關the section called “輸入”資訊，請參閱 GetInput。

產生一個`DynamoDBGetItemRequest`物件以向 DynamoDB 發出GetItem請求。

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

export function request(ctx) {
 return get({ key: { id: ctx.args.id } });
}
```

`put<T>(payload): DynamoDBPutItemRequest`

產生一個DynamoDBPutItemRequest物件以向 DynamoDB 發出PutItem請求。

```
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`

產生一個DynamoDBDeleteItemRequest物件以向 DynamoDB 發出DeleteItem請求。

```
import * as ddb from '@aws-appsync/utils/dynamodb';

export function request(ctx) {
 return ddb.remove({ key: { id: ctx.args.id } });
}
```

`scan<T>(payload): DynamoDBScanRequest`

產生一個DynamoDBScanRequest以向 DynamoDB 發出掃描請求。

```
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`

生成一個DynamoDBSyncRequest對象以發出同步請求。請求僅接收自上次查詢（增量更新）以來更改的數據。只能對已建立版本化的 DynamoDB 資料來源發出請求。

```
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`

產生一個`DynamoDBUpdateItemRequest`物件以向 DynamoDB 發出UpdateItem請求。

操作

操作助手允許您在更新期間對部分數據採取特定操作。要開始使用，請`operations`從以下位置導入`@aws-appsync/utils/dynamodb`：

```
// Modules are imported using operations
import {operations} from '@aws-appsync/utils/dynamodb';
```

作業清單

`add<T>(payload)`

在更新 DynamoDB 時新增新屬性項目的協助程式函數。

範例

若要使用 ID 值將地址 (街道、城市和郵遞區號) 新增至現有 DynamoDB 項目：

```
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)`

可將裝載附加至 DynamoDB 中現有清單的協助程式函數。

範例

要在更新期間將新添加的好友 ID（`newFriendIds`）添加到現有的好友列表（`friendsIds`）：

```
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?)

此協助函數會在更新 DynamoDB 時遞減項目中的現有屬性值。

範例

要將朋友計數器（`friendsCount`）遞減 10：

```
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 時，可增加項目中現有屬性值的協助函數。

範例

要增加一個朋友計數器（`friendsCount`）10：

```
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)

附加在 DynamoDB 中現有清單之前的輔助函數。

範例

要在更新期間將新添加的好友 ID (`newFriendIds`) 添加到現有的好友列表 (`friendsIds`) ：

```
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)

在 DynamoDB 中更新項目時取代現有屬性的協助函數。當您想要更新屬性中的整個物件或子物件，而不僅僅是有效負載中的索引鍵時，此功能非常有用。

範例

若要取代`info`物件中的地址 (街道、城市和郵遞區號)，請執行下列動作：

```
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)`

替換列表中的項目的輔助函數。

範例

在 update (`newFriendIds`) 的範圍中，這個範例用`updateListItem`來更新清單 () 中第二個項目 (索引:1、新 ID:102) 和第三個項目 (索引:2、新 ID: `112friendsIds`) 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 });
}
```

輸入

輸入清單

`Type GetInput<T>`

```
GetInput<T>: {
    consistentRead?: boolean;
    key: DynamoDBKey<T>;
}
```

類型宣告

- `consistentRead?: boolean` (選用)

  選用的布林值，用來指定是否要與 DynamoDB 執行強烈一致性讀取。

- `key: DynamoDBKey<T>` (必要)

  指定 DynamoDB 中項目索引鍵的必要參數。DynamoDB 項目可能具有單一雜湊鍵或雜湊和排序索引鍵。

`Type PutInput<T>`

```
PutInput<T>: {
```

```
    _version?: number;
    condition?: DynamoDBFilterObject<T> | null;
    customPartitionKey?: string;
    item: Partial<T>;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
}
```

類型宣告

- `_version?: number` (選用)
- `condition?: DynamoDBFilterObject<T> | null` (選用)

  將物件放入 DynamoDB 表時，您可以選擇性地指定條件運算式，以根據執行作業之前已在 DynamoDB 中的物件狀態來控制要求是否成功。

- `customPartitionKey?: string` (選用)

  啟用時，此字串值會在啟用版本控制時修改 delta 同步表所使用的`ds_sk`和`ds_pk`記錄的格式。啟用時，也會啟用`populateIndexFields`項目的處理。

- `item: Partial<T>` (必要)

  要放置在 DynamoDB 中之項目的其餘屬性。

- `key: DynamoDBKey<T>` (必要)

  必要參數，指定要在 DynamoDB 中執行置入的項目索引鍵。DynamoDB 項目可能具有單一雜湊鍵或雜湊和排序索引鍵。

- `populateIndexFields?: boolean` (選用)

  布林值，當與啟用時，會為差異同步表中的每筆記錄建立新項目，特別是在`gsi_ds_pk`和`gsi_ds_sk`欄中。`customPartitionKey`如需詳細資訊，請參閱AWS AppSync開發人員指南中的衝突偵測與同步處理。

`Type QueryInput<T>`

```
QueryInput<T>: ScanInput<T> & {
    query: DynamoDBKeyCondition<Required<T>>;
}
```

類型宣告

- `query: DynamoDBKeyCondition<Required<T>>` (必要)

指定描述要查詢項目的索引鍵條件。對於給定的索引，分區索引鍵的條件應該是相等，排序索引鍵應為比較或 a beginsWith (當它是字串時)。分割區和排序索引鍵僅支援數字和字串類型。

範例

採取以下User類型：

```
type User = {
  id: string;
  name: string;
  age: number;
  isVerified: boolean;
  friendsIds: string[]
}
```

查詢只能包含下列欄位：idname、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;
}
```

類型宣告

- _version?: number (選用)
- condition?: DynamoDBFilterObject<T> (選用)

當您移除 DynamoDB 中的物件時，您可以選擇性地指定條件運算式，以根據執行作業之前 DynamoDB 中已存在的物件狀態來控制要求是否成功。

範例

下列範例是包含條件的運DeleteItem算式，只有在文件的擁有者符合提出要求的使用者時，才允許作業成功。

```
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 (選用)

  啟用時，customPartitionKey值會在啟用版本控制時修改差異同步表所使用的ds_sk和ds_pk記錄的格式。啟用時，也會啟用populateIndexFields項目的處理。

- key: DynamoDBKey<T> (必要)

  必要參數，指定要移除之 DynamoDB 中項目的索引鍵。DynamoDB 項目可能具有單一雜湊鍵或雜湊和排序索引鍵。

  範例

  如果User只有用戶的散列鍵id，那麼密鑰將如下所示：

```
type User = {
 id: number
 name: string
 age: number
 isVerified: boolean
}
const key: DynamoDBKey<User> = {
```

```
 id: 1,
}
```

如果表用戶有一個哈希鍵 (id) 和 sort key (name)，那麼鍵將如下所示：

```
type User = {
 id: number
 name: string
 age: number
 isVerified: boolean
 friendsIds: string[]
}

const key: DynamoDBKey<User> = {
 id: 1,
 name: 'XXXXXXXXXX',
}
```

- populateIndexFields?: boolean (選用)

  布林值，當與啟用時，會為差異同步表中的每筆記錄建立新項目，特別是在gsi_ds_pk和gsi_ds_sk欄中。customPartitionKey

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;
}
```

類型宣告

- consistentRead?: boolean | null (選用)

  選用的布林值，用於在查詢 DynamoDB 時指出一致的讀取。預設值為 false。

- filter?: DynamoDBFilterObject<T> | null (選用)

從表格擷取結果後，要套用至結果的選用篩選器。

- `index?: string | null` (選用)

  要掃描的索引的選擇性名稱。

- `limit?: number | null` (選用)

  可選的最大結果返回數。

- `nextToken?: string | null` (選用)

  一個可選的分頁令牌，以繼續以前的查詢。這會是從先前查詢所取得的。

- `scanIndexForward?: boolean | null` (選用)

  一個可選的布爾值，用於指示查詢是以升序還是遞減順序執行。依預設，此值是設為 `true`。

- `segment?: number` (選用)
- `select?: DynamoDBSelectAttributes` (選用)

  要從 DynamoDB 傳回的屬性。根據預設，AWS AppSyncDynamoDB 解析器只會傳回投影到索引中的屬性。支援的值是：

  - `ALL_ATTRIBUTES`

    返回指定表或索引中的所有項目屬性。如果您查詢本機次要索引，DynamoDB 會針對索引中每個相符項目，從父資料表擷取整個項目。如果索引設定為投射所有項目屬性，所有資料都可從本機次要索引取得，不需進行任何擷取。

  - `ALL_PROJECTED_ATTRIBUTES`

    傳回已投影到索引中的所有屬性。如果索引設定為投射所有屬性，此傳回值相當於指定 `ALL_ATTRIBUTES`。

  - `SPECIFIC_ATTRIBUTES`

    僅傳回中列出的屬性`ProjectionExpression`。此傳回值相當於指定`ProjectionExpression`而不指定任何值`AttributesToGet`。

- `totalSegments?: number` (選用)

`Type DynamoDBSyncInput<T>`

```
DynamoDBSyncInput<T>: {
    basePartitionKey?: string;
```

```
    deltaIndexName?: string;
    filter?: DynamoDBFilterObject<T> | null;
    lastSync?: number;
    limit?: number | null;
    nextToken?: string | null;
}
```

類型宣告

- basePartitionKey?: string (選用)

  執行同步作業時要使用的基底資料表的分割索引鍵。此欄位允許在資料表使用自訂分割區索引鍵時執行同步作業。

- deltaIndexName?: string (選用)

  用於同步作業的索引。當資料表使用自訂分割區索引鍵時，需要此索引才能在整個增量存放區資料表上啟用同步作業。同步操作將在 GSI 上執行（在gsi_ds_pk和上創建gsi_ds_sk）。

- filter?: DynamoDBFilterObject<T> | null (選用)

  從表格擷取結果後，要套用至結果的選用篩選器。

- lastSync?: number (選用)

  最後一次成功的同步作業開始的時刻 (以紀元毫秒為單位)。如果指定此值，只會傳回 lastSync 之後變更的項目。只有在從初始同步操作中擷取所有頁面後，才應填入此欄位。如果省略，從基表的結果將被返回。否則，從增量表的結果將被返回。

- limit?: number | null (選用)

  一次評估的可選項目數目上限。如果省略此值，預設限制將設為 100 個項目。此欄位的最大值為 1000 個項目。

- nextToken?: string | null (選用)

Type DynamoDBUpdateInput<T>

```
DynamoDBUpdateInput<T>: {
    _version?: number;
    condition?: DynamoDBFilterObject<T>;
    customPartitionKey?: string;
    key: DynamoDBKey<T>;
    populateIndexFields?: boolean;
    update: DynamoDBUpdateObject<T>;
```

```
}
```

類型宣告

- `_version?: number` (選用)
- `condition?: DynamoDBFilterObject<T>` (選用)

  當您更新 DynamoDB 中的物件時，您可以選擇性地指定條件運算式，以根據執行作業之前已在 DynamoDB 中的物件狀態來控制要求是否成功。

- `customPartitionKey?: string` (選用)

  啟用時，`customPartitionKey`值會在啟用版本控制時修改差異同步表所使用的`ds_sk`和`ds_pk`記錄的格式。啟用時，也會啟用`populateIndexFields`項目的處理。

- `key: DynamoDBKey<T>` (必要)

  必要參數，指定要更新的 DynamoDB 中項目的索引鍵。DynamoDB 項目可能具有單一雜湊鍵或雜湊和排序索引鍵。

- `populateIndexFields?: boolean` (選用)

  布林值，當與啟用時，會為差異同步表中的每筆記錄建立新項目，特別是在`gsi_ds_pk`和`gsi_ds_sk`欄中。`customPartitionKey`

- `update: DynamoDBUpdateObject<T>`

  一個對象，指定要更新的屬性以及它們的新值。更新物件可與`add`、、、、`remove`、`replace`、`increment`、`decrement`、`append`、`prepend`、搭配使用`updateListItem`。

Amazon RDS 模塊功能

Amazon RDS 模組功能可在與使用 Amazon RDS 資料 API 設定的資料庫進行互動時，提供增強的體驗。使用以下方式匯入模組`@aws-appsync/utils/rds`：

```
import * as rds from '@aws-appsync/utils/rds';
```

函數也可以單獨導入。例如，下面的導入使用`sql`：

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

函數

您可以使用 AWS AppSync RDS 模組的公用程式助手與資料庫互動。

Select

此`select`公用程式會建立`SELECT`陳述式來查詢您的關聯式資料庫。

基本使用

在其基本形式中，您可以指定要查詢的表格：

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // "SELECT * FROM "persons"
    return createPgStatement(select({table: 'persons'}));
}
```

請注意，您還可以在表標識符中指定模式：

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // SELECT * FROM "private"."persons"
    return createPgStatement(select({table: 'private.persons'}));
}
```

指定欄

您可以使用`columns`屬性指定欄。如果未將其設置為值，則默認為`*`：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    return createPgStatement(select({
```

```
        table: 'persons',
        columns: ['id', 'name']
    }));
}
```

您也可以指定資料欄的表格：

```
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

您可以套用limit和offset至查詢：

```
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
    }));
}
```

排序方式

您可以使用orderBy屬性對結果進行排序。提供指定列和可選dir屬性的對象的數組：

```
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'}]
    }));
}
```

篩選條件

您可以使用特殊條件物件來建置篩選器：

```
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'}}
    }));
}
```

您還可以組合過濾器：

```
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}}
    }));
}
```

您也可以建立OR陳述式：

```
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not用以下方式否定條件：

```
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 } }
        ]}
    }));
}
```

您也可以使用下列運算子來比較值：

運算子	描述	可能的值類型
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

該insert實用程序提供了一種通過INSERT操作在數據庫中插入單行項目的直接方法。

單一項目插入

若要插入項目，請指定表格，然後傳入值的物件。物件索引鍵會對應至您的資料表資料行。列名稱會自動轉義，並使用變量映射將值發送到數據庫：

```
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 用案例

您可以將`insert`後跟 a 組合起`select`來檢索插入的行：

```
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)
}
```

郵政使用案例

使用 Postgres，您可以使[returning](#)用從插入的列中取得資料。它接受`*`或列名的數組：

```
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

該update實用程序允許您更新現有行。您可以使用 condition 物件，將變更套用至符合條件的所有資料列中的指定資料行。例如，假設我們有一個模式，允許我們進行這種突變。我們希望更新namePerson的id值，3但前提是我們已經知道他們（known_since）自一年以來2000：

```
mutation Update {
    updatePerson(
        input: {id: 3, name: "Jon"},
        condition: {known_since: {ge: "2000"}}
    ) {
    id
    name
  }
}
```

我們的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)
}
```

我們可以在條件中添加檢查，以確保僅更新主鍵id等3於的行。同樣，對於 Postgresinserts，您可以使用returning返回修改後的數據。

Remove (移除)

該remove實用程序允許您刪除現有行。您可以在滿足條件的所有行上使用條件對象。請注意，這delete是中的保留關鍵字 JavaScript。 remove應該使用：

```
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)
}
```

轉換

在某些情況下，您可能需要更多有關在語句中使用的正確對象類型的特殊性。您可以使用提供的類型提示來指定參數的類型。 AWS AppSync支援與資料 API 相同的類型提示。您可以使用AWS AppSyncrds模塊中的typeHint函數來轉換參數。

下列範例可讓您傳送陣列做為 JSON 物件轉換的值。我們使用->運算子來擷取 JSON 陣列index2中的元素：

```
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
}
```

在處理和比較時，鑄造也很有用 DATETIME，和TIMESTAMP：

```
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)
}
```

以下是另一個示例，顯示如何發送當前日期和時間：

```
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)}`)
}
```

可用的類型提示

- typeHint.DATE-對應的參數作為DATE類型的對象發送到數據庫。接受的格式為 YYYY-MM-DD。
- typeHint.DECIMAL-對應的參數作為DECIMAL類型的對象發送到數據庫。
- typeHint.JSON-對應的參數作為JSON類型的對象發送到數據庫。
- typeHint.TIME-對應的字串參數值會作為TIME類型的物件傳送至資料庫。接受的格式為 HH:MM:SS[.FFF]。

- typeHint.TIMESTAMP-對應的字串參數值會作為TIMESTAMP類型的物件傳送至資料庫。接受的格式為 YYYY-MM-DD HH:MM:SS[.FFF]。
- typeHint.UUID-對應的字串參數值會作為UUID類型的物件傳送至資料庫。

運行時實用

該runtime庫提供了用於控制或修改解析器和函數的運行時屬性的實用程序。

運行時實用程序列表

runtime.earlyReturn(obj?: unknown): never

調用此函數將停止當前AWS AppSync函數或解析器（單位或管道解析器）的執行取決於當前上下文。指定的對象作為結果返回。

- 在AWS AppSync函數要求處理常式中呼叫時，會略過資料來源和回應處理常式，並呼叫下一個函數要求處理常式 (或管線解析程式回應處理常式，如果這是最後一個AWS AppSync函數)。
- 在AWS AppSync管線解析程式要求處理常式中呼叫時，會略過管線執行，並立即呼叫管線解析程式回應處理常式。

範例

```
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
}
```

實用程序中的時間助手

util.time 變數包含日期時間方法，可協助產生時間戳記、在日期時間格式之間轉換，以及剖析日期時間字串。日期時間格式的語法基於您可以參考DateTimeFormatter以獲取進一步的文檔。我們在下面提供了一些示例，以及可用的方法和描述的列表。

時間實用程序

時間實用程序列表

`util.time.nowISO8601()`

以 ISO8601 格式傳回 UTC 的字串表示方式。

`util.time.nowEpochSeconds()`

傳回從 1970-01-01T00:00:00Z 的 epoch 到現在的秒數。

`util.time.nowEpochMilliSeconds()`

傳回從 1970-01-01T00:00:00Z 的 epoch 到現在的毫秒數。

`util.time.nowFormatted(String)`

使用字串輸入類型的指定格式，傳回 UTC 中目前時間戳記的字串。

`util.time.nowFormatted(String, String)`

使用字串輸入類型的指定格式和時區，傳回時區目前時間戳記的字串。

`util.time.parseFormattedToEpochMilliSeconds(String, String)`

解析作為字符串傳遞的時間戳以及格式，然後返回自 epoch 以毫秒為單位的時間戳。

`util.time.parseFormattedToEpochMilliSeconds(String, String, String)`

解析作為字符串傳遞的時間戳以及格式和時區，然後返回自 epoch 以毫秒為單位的時間戳。

`util.time.parseISO8601ToEpochMilliSeconds(String)`

解析作為字符串傳遞的 ISO8601 時間戳，然後返回自紀元以毫秒為單位的時間戳。

`util.time.epochMilliSecondsToSeconds(long)`

將 epoch 毫秒時間戳記轉換為 epoch 秒時間戳記。

`util.time.epochMilliSecondsToISO8601(long)`

將紀元毫秒時間戳記轉換為 ISO8601 時間戳記。

`util.time.epochMilliSecondsToFormatted(long, String)`

將紀元毫秒時間戳記 (只要傳遞) 轉換為根據提供的 UTC 格式格式化的時間戳記。

`util.time.epochMilliSecondsToFormatted(long, String, String)`

將紀元毫秒時間戳記 (長時間傳遞) 轉換為根據提供的時區提供的格式格式化的時間戳記。

實用程序中的動態 DynamoDB 助手

`util.dynamodb`包含協助程式方法，可讓您更輕鬆地將資料寫入和讀取 Amazon DynamoDB，例如自動類型對應和格式化。

到 DynamoDB

到實用程序列表

`util.dynamodb.toDynamoDB(Object)`

DynamoDB 的一般物件轉換工具，可將輸入物件轉換為適當的 DynamoDB 表示法。它在代表一些類型的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「SS」、「NS」、「BS」)。這會傳回描述 DynamoDB 屬性值的物件。

字符串示例

```
Input:      util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

數字示例

```
Input:      util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

布爾示例

```
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" }
                        }
                    }
                ]
            }
```

地圖示例

```
Input:      util.dynamodb.toDynamoDB({ "foo": "bar", "baz" : 1234, "beep":
 [ "boop"] })
Output:     {
               "M" : {
                   "foo"  : { "S" : "bar" },
                   "baz"  : { "N" : 1234 },
                   "beep" : {
                       "L" : [
                           { "S" : "boop" }
                       ]
                   }
               }
            }
```

。toString 實用程序。

toString 實用程序列表

`util.dynamodb.toString(String)`

將輸入字串轉換為 DynamoDB 字串格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

`util.dynamodb.toStringSet(List<String>)`

將包含字串的清單轉換為 DynamoDB 字串集格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
```

```
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

音量号实用程序

音量號碼實用程序列表

`util.dynamodb.toNumber(Number)`

將數字轉換 DynamoDB 數字格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

`util.dynamodb.toNumberSet(List<Number>)`

將數字清單轉換為 DynamoDB 數字集格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

二元實用程序

二進制實用程序列表

`util.dynamodb.toBinary(String)`

將編碼為 base64 字串的二進位資料轉換為 DynamoDB 二進位格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

`util.dynamodb.toBinarySet(List<String>)`

將編碼為 base64 字串的二進位資料清單轉換為 DynamoDB 二進位集格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

到布爾實用程序

到布爾實用程序列表

`util.dynamodb.toBoolean(Boolean)`

將布林值轉換為適當的 DynamoDB 布林格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

音效应用程序

音效應用程式清單

`util.dynamodb.toNull()`

以空值格 DynamoDB 回空值。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toNull()
Output:     { "NULL" : null }
```

到列表實用程序

。主列表實用程序列表。

`util.dynamodb.toList(List)`

將物件清單轉換為 DynamoDB 清單格式。清單中的每個項目也會轉換為其適當的 DynamoDB 格式。它在代表一些巢狀物件的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「SS」、「NS」、「BS」)。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
                           "bar" : { "S" : "baz" }
                       }
```

```
                }
            ]
        }
```

toMap 用程序

。toMap 實用程序列表。

`util.dynamodb.toMap(Map)`

將地圖轉換為 DynamoDB 對映格式。地圖中的每個值也會轉換為其適當的 DynamoDB 格式。它在代表一些巢狀物件的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「SS」、「NS」、「BS」)。這會傳回描述 DynamoDB 屬性值的物件。

```
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)`

建立對映的副本，其中每個值都已轉換為適當的 DynamoDB 格式。它在代表一些巢狀物件的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「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**
>
> 這與稍有不同，`util.dynamodb.toMap(Map)`因為它只會傳回 DynamoDB 屬性值的內容，但不會傳回整個屬性值本身。例如，下列陳述式完全相同：
>
> ```
> util.dynamodb.toMapValues(<map>)
> util.dynamodb.toMap(<map>)("M")
> ```

S3 对象实用程序

S3 對象實用程序列表

`util.dynamodb.toS3Object(String key, String bucket, String region)`

將金鑰、儲存貯體和區域轉換為 DynamoDB S3 物件表示法。這會傳回描述 DynamoDB 屬性值的物件。

```
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)`

將金鑰、儲存貯體、區域和選用版本轉換為 DynamoDB S3 物件表示法。這會傳回描述 DynamoDB 屬性值的物件。

```
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 物件的字串值，並傳回包含金鑰、儲存貯體、區域和選用版本的對映。

```
Input:      util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
 \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
```

```
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :
 "beep" }
```

HTTP 助手在實用程序。

該實用util.http程序提供了幫助程序方法，您可以使用這些方法來管理 HTTP 請求參數和添加響應標頭。

實用程序列表

`util.http.copyHeaders(headers)`

從地圖複製標頭，而不受限制的 HTTP 標頭集。您可以使用它將請求標頭轉發到下游 HTTP 端點。

`util.http.addResponseHeader(String, Object)`

添加一個單一的自定義標題與名稱（String）和 value（Object）的響應。有下列限制：

- 標頭名稱不能匹配任何現有的或受限制的AWS或AWS AppSync標題。
- 標頭名稱不能以受限制的前綴開頭，例如x-amzn-或x-amz-。
- 自訂回應標頭的大小不得超過 4 KB。這包括標題名稱和值。
- 您應該在每個 GraphQL 作業中定義每個回應標頭一次。但是，如果您多次定義具有相同名稱的自訂標頭，則回應中會顯示最新的定義。無論命名如何，所有標題都計入標題大小限制。

`util.http.addResponseHeaders(Map)`

將多個響應頭添加到指定映射的名稱（String）和 value（Object）的響應。該addResponseHeader(String, Object)方法列出的相同限制也適用於此方法。

變換中的轉換助手

util.transform包含協助程式方法，可讓您更輕鬆地對資料來源執行複雜的作業。

轉換助手實用程序列表

`util.transform.toDynamoDBFilterExpression(filterObject: DynamoDBFilterObject) : string`

將輸入字串轉換為篩選器運算式，以與 DynamoDB 搭配使用。我們建議toDynamoDBFilterExpression與內置模塊功能一起使用。

`util.transform.toElasticsearchQueryDSL(object: OpenSearchQueryObject) : string`

將指定的輸入轉換為其對等的 OpenSearch 查詢 DSL 運算式，並以 JSON 字串的形式傳回。

示例輸入：

```
util.transform.toElasticsearchQueryDSL({
    "upvotes":{
        "ne":15,
        "range":[
            10,
            20
        ]
    },
    "title":{
        "eq":"hihihi",
        "wildcard":"h*i"
    }
  })
```

示例輸出：

```
{
    "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**
>
> 預設運算子假設為 AND。

`util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter`

將Map輸入物件轉換為SubscriptionFilter運算式物件。
該util.transform.toSubscriptionFilter方法被用
作extensions.setSubscriptionFilter()擴展的輸入。如需詳細資訊，請參閱擴充功能。

> **Note**
>
> 參數和返回語句列出如下：
> 參數

- objFilter: SubscriptionFilterObject

  轉換為SubscriptionFilter運算式物件的Map輸入物件。

- ignoredFields: SubscriptionFilterExcludeKeysType (選擇性)

  第一List個將被忽略的物件中的欄位名稱。

- rules: SubscriptionFilterRuleObject (選擇性)

  具有嚴格規則的Map輸入對象，當您構建SubscriptionFilter表達式對象時包含該對象。這些嚴格規則將包含在SubscriptionFilter運算式物件中，以便至少滿足其中一個規則以通過訂閱篩選器。

回應

傳回 SubscriptionFilter。

`util.transform.toSubscriptionFilter(Map, List)`

將Map輸入物件轉換為SubscriptionFilter運算式物件。該util.transform.toSubscriptionFilter方法被用作extensions.setSubscriptionFilter()擴展的輸入。如需詳細資訊，請參閱擴充功能。

第一個引數是轉換為SubscriptionFilter運算式物件的Map輸入物件。第二個參數是在構建SubscriptionFilter表達式對象時在第一個Map輸入對象中忽略List的字段名稱。

`util.transform.toSubscriptionFilter(Map, List, Map)`

將Map輸入物件轉換為SubscriptionFilter運算式物件。該util.transform.toSubscriptionFilter方法被用作extensions.setSubscriptionFilter()擴展的輸入。如需詳細資訊，請參閱擴充功能。

`util.transform.toDynamoDBConditionExpression(conditionObject)`

建立 DynamoDB 件運算式。

訂閱篩選引數

下表說明如何定義下列公用程式的引數：

- `Util.transform.toSubscriptionFilter(objFilter, ignoredFields?, rules?): SubscriptionFilter`

Argument 1: Map

引數 1 是具有下列索引鍵值的Map物件：

- 欄位名稱
- 「和」
- 「或」

對於作為鍵的欄位名稱，這些欄位項目中的條件格式為。"operator" : "value"

下列範例顯示如何將項目新增至Map：

```
"field_name" : {
                    "operator1" : value
               }

## We can have multiple conditions for the same field_name:

"field_name" : {
                    "operator1" : value
                    "operator2" : value
                    .
                    .
                    .
               }
```

當欄位上有兩個或多個條件時，所有這些條件都會被視為使用 OR 作業。

輸入也Map可以有「and」和「or」作為鍵，這意味著應該使用 AND 或 OR 邏輯連接這些內的所有條目，具體取決於鍵。鍵值「and」和「or」需要一組條件。

```
"and" : [

           {
               "field_name1" : {
                   "operator1" : value
               }
            },

            {
               "field_name2" : {
                   "operator1" : value
```

```
                }
            },
            .
            .
        ].
```

請注意，您可以嵌套「和」和「or」。也就是說，您可以在另一個「和」/「或」塊中嵌套「和」/「或」塊。但是，這不適用於簡單的字段。

```
"and" : [

            {
                "field_name1" : {
                    "operator" : value
                }
             },

             {
                "or" : [
                            {
                                "field_name2" : {
                                    "operator" : value
                                }
                            },

                            {
                                "field_name3" : {
                                    "operator" : value
                                }
                            }

                        ].
```

下面的例子顯示了使用參數 1 的輸入`util.transform.toSubscriptionFilter(Map) : Map`。

輸入

參數 1：地圖：

```
{
  "percentageUp": {
    "lte": 50,
```

```
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

輸出

結果是一個Map對象：

```
{
  "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

參數 2 包含在構建SubscriptionFilter表達式對象時不應在輸入Map（參數 1）中考慮List的字段名稱。也List可以是空的。

下面的例子顯示了使用參數 1 和參數 2 的輸入util.transform.toSubscriptionFilter(Map, List) : Map。

輸入

參數 1：地圖：

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
```

```
}
```

引數 2：列表：

```
["percentageUp", "author"]
```

輸出

結果是一個Map對象：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 3: Map

引數 3 是以欄位名稱作為鍵值的Map物件 (不能有「and」或「or」)。對於作為鍵的欄位名稱，這些欄位上的條件是形式的項目"operator" : "value"。與引數 1 不同，參數 3 在同一個鍵中不能有多個條件。此外，參數 3 沒有「and」或「or」子句，因此也不涉及嵌套。

引數 3 代表嚴格規則的清單，這些規則會新增至SubscriptionFilter運算式物件，以便至少符合這些條件中的一個以傳遞篩選器。

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.
```

下面的例子顯示了參數 1，參數 2 和參數 3 使用的輸入`util.transform.toSubscriptionFilter(Map, List, Map) : Map`。

輸入

參數 1：地圖：

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
```

```
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

引數 2：列表：

```
["percentageUp", "author"]
```

參數 3：地圖：

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
}
```

輸出

結果是一個Map對象：

```
{
  "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"
        }
      ]
    }
  ]
}
```

字符串助手在實用程序。STR

util.str包含幫助常見字符串操作的方法。

實用程序列表

`util.str.normalize(String, String)`

使用以下四種 Unicode 標準化形式之一來標準化字符串：NFC，NFD，NFKC 或 NFKD。第一個參數是要規範化的字符串。第二個參數是「nfc」，「nfd」，「nfkc」或「nfkd」，指定用於正規化過程的標準化類型。

擴充

`extensions`包含一組在解析器中執行其他動作的方法。

緩存擴展

`extensions.evictFromApiCache(typeName: string, fieldName: string, keyValuePair: Record<string, string>) : Object`

從AWS AppSync 伺服器端快取中收回項目。第一個參數是類型名稱。第二個引數是欄位名稱。第三個參數是包含指定緩存鍵值的鍵-值對項目的對象。您必須按照與緩存解析器中的緩存密鑰相同的順序放置對象中的項目。`cachingKey`如需快取的詳細資訊，請參閱快取行為。

範例 1：

此範例會逐出針對呼叫的解析器快取的項目，該項目使用了呼叫`Query.allClasses`的快取金鑰。`context.arguments.semester`當調用突變並運行解析器時，如果一個條目被成功清除，則響應包含在擴展對象中的一個`apiCacheEntriesDeleted`值，該值顯示了刪除了多少條目。

```
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**
>
> 此函數僅適用於突變，而不適用於查詢。

訂閱擴展

`extensions.setSubscriptionFilter(filterJsonObject)`

定義增強的訂閱篩選器。系統會根據提供的訂閱篩選器評估每個訂閱通知事件，並在所有篩選器評估為時向`true`用戶端傳送通知 引數為 `filterJsonObject` (有關此引數的更多資訊可在下方參數: filterJsonObject區段中找到。)。請參閱增強的訂閱篩選。

> **Note**
>
> 您只能在訂閱解析程式的回應處理常式中使用此延伸功能。此外，我們建議您使用`util.transform.toSubscriptionFilter`來建立篩選器。

`extensions.setSubscriptionInvalidationFilter(filterJsonObject)`

定義訂閱失效篩選器。會根據無效承載評估訂閱篩選器，然後如果篩選器評估為，則會使指定的訂閱失效。`true`引數為 `filterJsonObject` (有關此引數的更多資訊可在下方參數: filterJsonObject區段中找到。)。請參閱增強的訂閱篩選。

> **Note**
>
> 您只能在訂閱解析程式的回應處理常式中使用此延伸功能。此外，我們建議您使用`util.transform.toSubscriptionFilter`來建立篩選器。

`extensions.invalidateSubscriptions(invalidationJsonObject)`

用於從突變啟動訂閱失效。引數為 `invalidationJsonObject` (有關此引數的更多資訊可在下方參數: invalidationJsonObject區段中找到。)。

> **Note**
>
> 此擴充功能只能用於突變解析器的回應對應範本。

您最多只能在任何單個請求中使用五個唯一的`extensions.invalidateSubscriptions()`方法調用。如果超過此限制，您將收到 GraphQL 錯誤。

引數： filterJsonObject

JSON 物件會定義訂閱或無效驗證篩選器。它是一個篩選器陣列中的`filterGroup`. 每個濾鏡都是個別篩選器的集合。

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

每個篩選器都有三個屬性：

- `fieldName`— [圖 GraphQL 結構描述] 欄位。
- `operator`— 運算子類型。
- `value`— 要與訂閱通知值進行比較的`fieldName`值。

以下是這些屬性的範例指派：

```
{
 "fieldName" : "severity",
 "operator" : "le",
 "value" : context.result.severity
}
```

引數： invalidationJsonObject

定invalidationJsonObject義以下內容：

- subscriptionField— 要失效的 GraphQL 結構描述訂閱。單一訂閱 (在中定義為字串) 會subscriptionField被視為無效驗證。
- payload— 鍵值配對清單，如果無效篩選器根據其值進行評估，則用作使訂閱無效的輸入。true

  下列範例會在onUserDelete訂閱解析程式中定義的無效驗證篩選器對值進行評估時，使用訂閱將已訂閱和連線的用戶端失效。true payload

  ```
  export const request = (ctx) => ({ payload: null });

  export function response(ctx) {
   extensions.invalidateSubscriptions({
    subscriptionField: 'onUserDelete',
    payload: { group: 'Developer', type: 'Full-Time' },
   });
   return ctx.result;
  }
  ```

在 util.xml 中的 XML 助手

util.xml包含協助 XML 字串轉換的方法。

util.xml 實用程序列表

util.xml.toMap(String) : Object

將 XML 字串轉換為字典。

範例 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"
      }
    }
}
```

範例 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`

將 XML 字串轉換為 JSON 字串。這類似於`toMap`，不同之處在於輸出是一個字符串。如果要將來自 HTTP 物件的 XML 回應直接轉換並傳回給 JSON，這非常實用。您可以設定選用的布林參數，以決定是否要對 JSON 進行字串編碼。

JavaScript解析器函數參考

該AWS AppSync使用 DynamoDB 功能可讓您使用图形 QL以便在您帳戶的現有 Amazon DynamoDB 表中存放和擷取資料。此解析器的運作方式是讓您將傳入的 GraphQL 請求對應至 DynamoDB 呼叫，然後將 DynamoDB 回應對應回應至 GraphQL。本節說明支援 DynamoDB 作業的請求和回應處理常式。

GetItem

該`GetItem`請求可讓您告訴AWS AppSync用來建立一個`GetItem`向 DynamoDB 發出請求，並可讓您指定：

- 項目在動態支援中的索引鍵
- 是否使用一致性讀取

該`GetItem`請求具有以下結構：

```
type DynamoDBGetItem = {
  operation: 'GetItem';
  key: { [key: string]: any };
  consistentRead?: ConsistentRead;
  projection?: {
```

```
    expression: string;
    expressionNames?: { [key: string]: string };
  };
};
```

欄位定義如下：

GetItem 欄位

GetItem欄位清單

`operation`

要執行的動態支援作業。若要執行 `GetItem` DynamoDB 操作，這必須設為 `GetItem`。此值為必填。

`key`

項目在動態支援中的索引鍵。DynamoDB 項目可能具有單一雜湊鍵，或是雜湊鍵和排序索引鍵，具體取決於資料表結構。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。此值為必填。

`consistentRead`

是否要對 DynamoDB 執行強烈一致的讀取。此為選用，預設值為 `false`。

`projection`

用來指定要從 DynamoDB 作業傳回之屬性的投影。若要取得有關投影的更多資訊，請參閱投影。此欄位為選用欄位。

從 DynamoDB 傳回的項目會自動轉換為 GraphQL 和 JSON 原始類型，並可在內容結果中使用 (`context.result`).

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統（響應映射）。

有關更多信息JavaScript解析器，請參閱JavaScript解析器概述。

範例

下列範例為 GraphQL 查詢的函數要求處理常式`getThing(foo: String!, bar: String!)`:

```
export function request(ctx) {
  const {foo, bar} = ctx.args
```

```
  return {
    operation : "GetItem",
    key : util.dynamodb.toMapValues({foo, bar}),
    consistentRead : true
  }
}
```

如需 DynamoDB `GetItem` API 的詳細資訊，請參閱 DynamoDB API 文件。

PutItem

該`PutItem`請求對應文件可讓您告訴AWS AppSync用來建立一個`PutItem`向 DynamoDB 發出請求，並可讓您指定下列項目：

- 項目在動態支援中的索引鍵
- 項目 (由 `key` 和 `attributeValues` 組成) 的完整內容
- 操作成功的條件

該`PutItem`請求具有以下結構：

```
type DynamoDBPutItemRequest = {
  operation: 'PutItem';
  key: { [key: string]: any };
  attributeValues: { [key: string]: any};
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

欄位定義如下：

PutItem 欄位

PutItem欄位清單

`operation`

要執行的動態支援作業。若要執行 `PutItem` DynamoDB 操作，這必須設為 `PutItem`。此值為必填。

`key`

項目在動態支援中的索引鍵。DynamoDB 項目可能具有單一雜湊鍵，或是雜湊鍵和排序索引鍵，具體取決於資料表結構。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。此值為必填。

`attributeValues`

將放入 DynamoDB 的項目其餘屬性。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。此欄位為選用欄位。

`condition`

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，`PutItem` 要求會覆寫該項目的任何現有資料項目。如需條件的更多資訊，請參閱條件運算式。此值是選用的。

`_version`

代表項目之最新已知版本的數值。此值是選用的。此欄位用於衝突偵測，而且僅支援已建立版本的資料來源。

`customPartitionKey`

啟用時，此字串值會修改`ds_sk`和`ds_pk`啟用版本控制時增量同步表所使用的記錄(如需詳細資訊，請參閱衝突偵測與同步在AWS AppSync開發者指南). 啟用時，處理`populateIndexFields`條目也被啟用。此欄位為選用欄位。

`populateIndexFields`

啟用時的布爾值隨著**`customPartitionKey`**，為增量同步表中的每個記錄創建新條目，特別是`gsi_ds_pk`和`gsi_ds_sk`列。如需詳細資訊，請參閱衝突偵測與同步在AWS AppSync開發者指南。此欄位為選用欄位。

寫入 DynamoDB 的項目會自動轉換為 GraphQL 和 JSON 原始類型，並可在內容結果中使用 (`context.result`).

寫入 DynamoDB 的項目會自動轉換為 GraphQL 和 JSON 原始類型，並可在內容結果中使用 (`context.result`).

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統（響應映射）。

有關更多信息JavaScript解析器，請參閱JavaScript解析器概述。

範例 1

下面的例子是一個 GraphQL 突變的函數請求處理程序`updateThing(foo: String!, bar: String!, name: String!, version: Int!)`。

若無指定索引鍵的項目，則會建立該項目。若已有指定索引鍵的項目，則會覆寫該項目。

```
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),
  };
}
```

範例 2

下面的例子是一個 GraphQL 突變的函數請求處理程序`updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)`。

此範例會驗證目前在 DynamoDB 中的項目具有`version`欄位設定為`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,
  };
}
```

如需 DynamoDB `PutItem` API 的詳細資訊，請參閱 DynamoDB API 文件。

UpdateItem

該UpdateItem請求使您能夠告訴AWS AppSync用來建立一個UpdateItem向 DynamoDB 發出請求，並可讓您指定下列項目：

- 項目在動態支援中的索引鍵
- 說明如何在 DynamoDB 中更新項目的更新運算式
- 操作成功的條件

該UpdateItem請求具有以下結構：

```
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;
};
```

欄位定義如下：

UpdateItem 欄位

UpdateItem欄位清單

`operation`

要執行的動態支援作業。若要執行 UpdateItem DynamoDB 操作，這必須設為 UpdateItem。此值為必填。

`key`

項目在動態支援中的索引鍵。DynamoDB 項目可能具有單一雜湊鍵，或是雜湊鍵和排序索引鍵，具體取決於資料表結構。如需有關指定「類型值」的詳細資訊，請參閱類型系統 (請求對應)。此值為必填。

`update`

該update區段可讓您指定描述如何更新 DynamoDB 中項目的更新運算式。如需如何撰寫更新運算式的詳細資訊，請參閱DynamoDBUpdateExpressions文件。此區段是必須的。

update 區段有三個元件：

expression

更新表達式。此值為必填。

expressionNames

表達式屬性 name 預留位置的替代，形式為鍵值組。該鍵對應於中使用的名稱佔位符expression，而且值必須是對應至 DynamoDB 中項目的屬性名稱的字串。此欄位為選用的，應只能填入用於 expression 中表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性 value 預留位置的替代，形式為鍵值組。鍵對應用於 expression 的值預留位置，值必須是類型值。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。此必須指定。此欄位為選用的，應只能填入用於 expression 中表達式屬性值預留位置的替代。

`condition`

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，UpdateItem 要求會更新現有的資料項目，無論項目的目前狀態為何。如需條件的更多資訊，請參閱條件運算式。此值是選用的。

`_version`

代表項目之最新已知版本的數值。此值是選用的。此欄位用於衝突偵測，而且僅支援已建立版本的資料來源。

`customPartitionKey`

啟用時，此字串值會修改ds_sk和ds_pk啟用版本控制時增量同步表所使用的記錄(如需詳細資訊，請參閱衝突偵測與同步在AWS AppSync開發者指南). 啟用時，處理populateIndexFields條目也被啟用。此欄位為選用欄位。

`populateIndexFields`

啟用時的布爾值隨著**customPartitionKey**，為增量同步表中的每個記錄創建新條目，特別是gsi_ds_pk和gsi_ds_sk列。如需詳細資訊，請參閱衝突偵測與同步在AWS AppSync開發者指南。此欄位為選用欄位。

在 DynamoDB 中更新的項目會自動轉換為 GraphQL 和 JSON 原始類型，並可在內容結果中使用 (`context.result`).

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統（響應映射）。

有關更多信息JavaScript解析器，請參閱JavaScript解析器概述。

範例 1

下面的例子是用於 GraphQL 突變的函數請求處理程序`upvote(id: ID!)`。

在此範例中，DynamoDB 中的某個項目具有其`upvotes`和`version`以 1 遞增的欄位。

```
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 } },
    },
  };
}
```

範例 2

下面的例子是一個 GraphQL 突變的函數請求處理程序`updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)`。

這個複雜的範例會檢查引數，並持續產生更新表達式，其只包含由用戶端提供的引數。例如，如果 `title` 和 `author` 遭到省略，則不會更新。如果指定了一個參數，但它的值是`null`，則會從 DynamoDB 中的物件中刪除該欄位。最後，作業有一個條件，可驗證 DynamoDB 中目前的項目是否具有`version`欄位設定為`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),
    },
  };
}
```

如需 DynamoDB `UpdateItem` API 的詳細資訊，請參閱 DynamoDB API 文件。

DeleteItem

該`DeleteItem`請求可讓您告訴AWS AppSync用來建立一個`DeleteItem`向 DynamoDB 發出請求，並可讓您指定下列項目：

- 項目在動態支援中的索引鍵
- 操作成功的條件

該`DeleteItem`請求具有以下結構：

```
type DynamoDBDeleteItemRequest = {
  operation: 'DeleteItem';
  key: { [key: string]: any };
  condition?: ConditionCheckExpression;
  customPartitionKey?: string;
  populateIndexFields?: boolean;
  _version?: number;
};
```

欄位定義如下：

DeleteItem 欄位

DeleteItem欄位清單

`operation`

要執行的動態支援作業。若要執行 `DeleteItem` DynamoDB 操作，這必須設為 `DeleteItem`。此值為必填。

`key`

項目在動態支援中的索引鍵。DynamoDB 項目可能具有單一雜湊鍵，或是雜湊鍵和排序索引鍵，具體取決於資料表結構。如需有關指定「類型值」的詳細資訊，請參閱類型系統 (請求對應)。此值為必填。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，DeleteItem 要求會刪除項目，無論該項目的目前狀態為何。如需條件的更多資訊，請參閱條件運算式。此值是選用的。

_version

代表項目之最新已知版本的數值。此值是選用的。此欄位用於衝突偵測，而且僅支援已建立版本的資料來源。

customPartitionKey

啟用時，此字串值會修改ds_sk和ds_pk啟用版本控制時增量同步表所使用的記錄(如需詳細資訊，請參閱衝突偵測與同步在AWS AppSync開發者指南). 啟用時，處理populateIndexFields條目也被啟用。此欄位為選用欄位。

populateIndexFields

啟用時的布爾值隨著**customPartitionKey**，為增量同步表中的每個記錄創建新條目，特別是gsi_ds_pk和gsi_ds_sk列。如需詳細資訊，請參閱衝突偵測與同步在AWS AppSync開發者指南。此欄位為選用欄位。

從 DynamoDB 刪除的項目會自動轉換為 GraphQL 和 JSON 原始類型，並可在內容結果中使用(context.result).

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統（響應映射）。

有關更多信息JavaScript解析器，請參閱JavaScript解析器概述。

範例 1

下面的例子是一個 GraphQL 突變的函數請求處理程序deleteItem(id: ID!)。如果已存在此 ID 的項目，則該項目將會遭到刪除。

```
import { util } from '@aws-appsync/utils';
export function request(ctx) {
  return {
    operation: 'DeleteItem',
    key: util.dynamodb.toMapValues({ id: ctx.args.id }),
  };
```

```
}
```

範例 2

下面的例子是一個 GraphQL 突變的函數請求處理程序`deleteItem(id: ID!, expectedVersion: Int!)`。如果已存在此 ID 的項目，則該項目將會遭到刪除，但其 `version` 欄位必須已設為 `expectedVersion`：

```
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),
  };
}
```

如需 DynamoDB `DeleteItem` API 的詳細資訊，請參閱 DynamoDB API 文件。

Query

該`Query`請求對象可以讓你告訴AWS AppSync用於製作一個解析器`Query`向 DynamoDB 發出請求，並可讓您指定下列項目：

- 索引鍵表達式
- 要使用哪些索引
- 任何額外篩選條件
- 要傳回多少項目
- 是否使用一致性讀取
- 查詢方向 (向前或向後)
- 分頁字符

該`Query`請求對象具有以下結構：

```
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 };
  };
};
```

欄位定義如下：

查詢欄位

查詢欄位清單

operation

要執行的動態支援作業。若要執行 Query DynamoDB 操作，這必須設為 Query。此值為必填。

query

該query區段可讓您指定索引鍵條件運算式，以說明要從 DynamoDB 擷取哪些項目。如需如何撰寫索引鍵條件運算式的詳細資訊，請參閱DynamoDBKeyConditions文件。必須指定此區段。

expression

查詢表達式。必須指定此欄位。

expressionNames

表達式屬性 name 預留位置的替代，形式為鍵值組。該鍵對應於中使用的名稱佔位符`expression`，而且值必須是對應至 DynamoDB 中項目的屬性名稱的字串。此欄位為選用的，應只能填入用於 `expression` 中表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性 value 預留位置的替代，形式為鍵值組。鍵對應用於 `expression` 的值預留位置，值必須是類型值。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。此值為必填。此欄位為選用的，應只能填入用於 `expression` 中表達式屬性值預留位置的替代。

filter

額外的篩選條件可用來在傳回 DynamoDB 的結果之前先篩選結果。如需篩選條件的詳細資訊，請參閱篩選條件。此欄位為選用欄位。

index

要查詢的索引名稱。DynamoDB 查詢作業可讓您掃描本機次要索引和全域次要索引，以及雜湊鍵的主索引索引。如果有指定，這會告知 DynamoDB 查詢指定的索引。若省略，則會查詢主索引鍵索引。

nextToken

分頁字符將繼續先前的查詢。這會是從先前查詢所取得的。此欄位為選用欄位。

limit

要評估的項目數上限 (不一定是相符的項目數)。此欄位為選用欄位。

scanIndexForward

指出是否向前或向後查詢的布林值。此欄位為選用，預設值為 `true`。

consistentRead

布林值，指出在查詢 DynamoDB 時是否使用一致讀取。此欄位為選用，預設值為 `false`。

select

默認情況下，AWS AppSyncDynamoDB 解析程式只會傳回投影到索引中的屬性。如果需要更多屬性，請設定這個欄位。此欄位為選用欄位。支援的值是：

ALL_ATTRIBUTES

傳回所有指定資料表或索引的項目屬性。如果您查詢本機次要索引，DynamoDB 會針對索引中每個相符項目，從父資料表擷取整個項目。如果索引設定為投射所有項目屬性，所有資料都可從本機次要索引取得，不需進行任何擷取。

ALL_PROJECTED_ATTRIBUTES

只在查詢索引時才允許。擷取所有已投射到索引的屬性。如果索引設定為投射所有屬性，此傳回值相當於指定 ALL_ATTRIBUTES。

SPECIFIC_ATTRIBUTES

僅返回中列出的屬性projection的expression。此傳回值等同於指定projection的expression而不指定任何值Select。

projection

用來指定要從 DynamoDB 作業傳回之屬性的投影。若要取得有關投影的更多資訊，請參閱投影。此欄位為選用欄位。

DynamoDB 的結果會自動轉換為 GraphQL 和 JSON 原始類型，並可在內容結果中使用 (context.result).

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統（響應映射）。

有關更多信息JavaScript解析器，請參閱JavaScript解析器概述。

結果的結構如下：

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

欄位定義如下：

items

包含 DynamoDB 查詢傳回之項目的清單。

nextToken

如果可能有一個以上的結果，nextToken 會包含可用於另一個要求的分頁字符。請注意AWS AppSync加密和混淆從 DynamoDB 傳回的分頁權杖。這樣可確保資料表的資料不會不慎洩漏給發起人。另請注意，這些分頁令牌不能在不同的函數或解析器中使用。

scannedCount

套用篩選條件表達式 (若有的話) 以前符合查詢條件表達式的項目數。

範例

下列範例為 GraphQL 查詢的函數要求處理常式getPosts(owner: ID!)。

在此範例中，資料表上的全域次要索引受到查詢，以傳回指定 ID 擁有的所有文章。

```
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',
  };
}
```

如需 DynamoDB Query API 的詳細資訊，請參閱 DynamoDB API 文件。

Scan

該Scan請求可讓您告訴AWS AppSync用來建立一個Scan向 DynamoDB 發出請求，並可讓您指定下列項目：

- 排除結果的篩選結果
- 要使用哪些索引
- 要傳回多少項目

- 是否使用一致性讀取
- 分頁字符
- 平行掃描

該Scan請求對象具有以下結構：

```
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 };
  };
};
```

欄位定義如下：

掃描欄位

掃描欄位清單

`operation`

要執行的動態支援作業。若要執行 Scan DynamoDB 操作，這必須設為 Scan。此值為必填。

`filter`

可用於在傳回 DynamoDB 結果之前篩選結果的篩選器。如需篩選條件的詳細資訊，請參閱篩選條件。此欄位為選用欄位。

index

要查詢的索引名稱。DynamoDB 查詢作業可讓您掃描本機次要索引和全域次要索引，以及雜湊鍵的主索引索引。如果有指定，這會告知 DynamoDB 查詢指定的索引。若省略，則會查詢主索引鍵索引。

limit

單次可評估的項目數量上限。此欄位為選用欄位。

consistentRead

布林值，指出在查詢 DynamoDB 時是否使用一致讀取。此欄位為選用，預設值為 `false`。

nextToken

分頁字符將繼續先前的查詢。這會是從先前查詢所取得的。此欄位為選用欄位。

select

默認情況下，AWS AppSyncDynamoDB 函數只會傳回任何投影到索引中的屬性。如果需要更多屬性，則此欄位可以設定。此欄位為選用欄位。支援的值是：

ALL_ATTRIBUTES

傳回所有指定資料表或索引的項目屬性。如果您查詢本機次要索引，DynamoDB 會針對索引中每個相符項目，從父資料表擷取整個項目。如果索引設定為投射所有項目屬性，所有資料都可從本機次要索引取得，不需進行任何擷取。

ALL_PROJECTED_ATTRIBUTES

只在查詢索引時才允許。擷取所有已投射到索引的屬性。如果索引設定為投射所有屬性，此傳回值相當於指定 `ALL_ATTRIBUTES`。

SPECIFIC_ATTRIBUTES

僅返回中列出的屬性`projection`的`expression`。此傳回值等同於指定`projection`的`expression`而不指定任何值`Select`。

totalSegments

執行平行掃描時分割資料表的區段數。此欄位為選用的，但若指定 `segment`，則此欄位必須指定。

segment

此操作中執行平行掃描時的資料表區段。此欄位為選用的，但若指定 `totalSegments`，則此欄位必須指定。

`projection`

用來指定要從 DynamoDB 作業傳回之屬性的投影。若要取得有關投影的更多資訊，請參閱投影。此欄位為選用欄位。

DynamoDB 掃描傳回的結果會自動轉換為 GraphQL 和 JSON 原始類型，並可在內容結果中使用 (`context.result`).

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統（響應映射）。

有關更多信息JavaScript解析器，請參閱JavaScript解析器概述。

結果的結構如下：

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

欄位定義如下：

`items`

包含 DynamoDB 掃描傳回之項目的清單。

`nextToken`

如果可能有更多結果，`nextToken`包含您可以在另一個請求中使用的分頁標記。AWSAppSync 加密和混淆從 DynamoDB 傳回的分頁權杖。這樣可確保資料表的資料不會不慎洩漏給發起人。此外，這些分頁令牌不能在不同的函數或解析器中使用。

`scannedCount`

DynamoDB 在套用篩選器運算式 (如果存在) 之前擷取的項目數。

範例 1

下列範例是 GraphQL 查詢的函數要求處理常式：`allPosts`。

在此範例中，資料表中的所有項目都會傳回。

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

範例 2

下列範例是 GraphQL 查詢的函數要求處理常式：`postsMatching(title: String!)`。

在此範例中，資料表中開頭為 `title` 引數的所有項目都會傳回。

```
export function request(ctx) {
  const { title } = ctx.args;
  const filter = { filter: { beginsWith: title } };
  return {
    operation: 'Scan',
    filter: JSON.parse(util.transform.toDynamoDBFilterExpression(filter)),
  };
}
```

如需 DynamoDB `Scan` API 的詳細資訊，請參閱 DynamoDB API 文件。

Sync

該`Sync`請求物件可讓您從 DynamoDB 表擷取所有結果，然後僅接收自上次查詢後變更的資料 (差異更新)。`Sync`請求只能對已建立版本化的 DynamoDB 資料來源發出。您可以指定下列選項：

- 排除結果的篩選結果
- 要傳回多少項目
- 分頁字符
- 上次起始 Sync 操作時

該`Sync`請求對象具有以下結構：

```
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 };
  };
};
```

欄位定義如下：

同步欄位

同步欄位清單

`operation`

要執行的動態支援作業。若要執行 Sync 操作，這必須設定為 Sync。此值為必填。

`filter`

可用於在傳回 DynamoDB 結果之前篩選結果的篩選器。如需篩選條件的詳細資訊，請參閱篩選條件。此欄位為選用欄位。

`limit`

單次可評估的項目數量上限。此欄位為選用欄位。如果省略此值，預設限制將設為 100 個項目。此欄位的最大值為 1000 個項目。

`nextToken`

分頁字符將繼續先前的查詢。這會是從先前查詢所取得的。此欄位為選用欄位。

`lastSync`

上次成功啟動 Sync 操作的時間 (以毫秒為單位)。如果指定此值，只會傳回 `lastSync` 之後變更的項目。這個欄位是選用的，而且只有在初始 Sync 操作擷取所有頁面之後才能填入。如果省略此值，將傳回 Base 資料表的結果，否則會傳回 Delta 資料表的結果。

`basePartitionKey`

的磁碟分割索引鍵基地執行時使用的表Sync操作。此欄位允許Sync當表格使用自訂分割區索引鍵時要執行的作業。此為選用欄位。

`deltaIndexName`

使用的索引Sync操作。需要此索引才能啟用Sync當表使用自定義分區鍵時，對整個增量存儲表進行操作。該Sync作業將在 GSI 上執行 (建立於`gsi_ds_pk`和`gsi_ds_sk`). 此欄位為選用欄位。

DynamoDB 同步傳回的結果會自動轉換為 GraphQL 和 JSON 原始類型，並可在內容結果中使用 (`context.result`).

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統（響應映射）。

有關更多信息JavaScript解析器，請參閱JavaScript解析器概述。

結果的結構如下：

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10,
    startedAt = 1550000000000
}
```

欄位定義如下：

`items`

包含同步傳回項目的清單。

`nextToken`

如果可能有更多結果，`nextToken`包含您可以在另一個請求中使用的分頁標記。AWSAppSync 加密和混淆從 DynamoDB 傳回的分頁權杖。這樣可確保資料表的資料不會不慎洩漏給發起人。此外，這些分頁令牌不能在不同的函數或解析器中使用。

`scannedCount`

DynamoDB 在套用篩選器運算式 (如果存在) 之前擷取的項目數。

`startedAt`

開始同步操作時，可以在本機存放並在另一個請求中做為 `lastSync` 引數的時間 (以 epoch 毫秒為單位)。如果請求中包含分頁字符，則該值將與請求針對第一頁結果傳回的值相同。

範例 1

下列範例是 GraphQL 查詢的函數要求處理常式：`syncPosts(nextToken: String, lastSync: AWSTimestamp)`。

在此範例中，如果省略 `lastSync`，則會傳回 Base 資料表中的所有項目。如果提供 `lastSync`，只會傳回自 `lastSync` 變更之 Delta Sync 資料表中的項目。

```
export function request(ctx) {
  const { nextToken, lastSync } = ctx.args;
  return { operation: 'Sync', limit: 100, nextToken, lastSync };
}
```

BatchGetItem

該BatchGetItem請求對象可以讓你告訴AWS AppSync用來建立一個BatchGetItem向 DynamoDB 請求以擷取可能跨多個表格的多個項目。對於此請求對象，您必須指定以下內容：

- 將從其中擷取項目的資料表名稱
- 將從個別資料表擷取項目的索引鍵

此時會套用 DynamoDB BatchGetItem 限制，且可能不會提供任何條件表達式。

該BatchGetItem請求對象具有以下結構：

```
type DynamoDBBatchGetItemRequest = {
  operation: 'BatchGetItem';
  tables: {
    [tableName: string]: {
      keys: { [key: string]: any }[];
      consistentRead?: boolean;
      projection?: {
        expression: string;
        expressionNames?: { [key: string]: string };
      };
    };
  };
};
```

欄位定義如下：

BatchGetItem 欄位

BatchGetItem欄位清單

operation

要執行的動態支援作業。若要執行 BatchGetItem DynamoDB 操作，這必須設為 BatchGetItem。此值為必填。

tables

要從中擷取項目的動態資料表。此值是其中將資料表名稱指定為該映射索引鍵的映射。至少必須提供一個資料表。tables 值為必填。

keys

表示要擷取之項目之主索引鍵的 DynamoDB 索引鍵清單。DynamoDB 項目可能具有單一雜湊鍵，或是雜湊鍵和排序索引鍵，具體取決於資料表結構。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。

consistentRead

是否在執行時使用一致的讀取GetItem操作。此值為選用值，且預設值為 false。

projection

用來指定要從 DynamoDB 作業傳回之屬性的投影。若要取得有關投影的更多資訊，請參閱投影。此欄位為選用欄位。

注意事項：

- 如果未從資料表中擷取任何項目，該資料表的 data 區塊中會顯示 null 元素。
- 調用結果是根據請求對象內提供的順序，每個表進行排序。
- 每個Get命令, 裡面, aBatchGetItem是原子的，但是，批次可以部分處理。如果因錯誤而部分批次處理，則未處理的索引鍵會透過 unprocessedKeys 區塊傳回為部分的呼叫結果。
- BatchGetItem 限制為 100 個索引鍵。

對於以下示例函數請求處理程序：

```
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 })],
    },
  };
```

```
}
```

可透過 ctx.result 提供的呼叫結果如下所示：

```
{
   "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 包含錯誤的詳細資訊。鑰匙資料,未處理的金鑰，並且在函數請求對像中的結果中提供的每個表鍵都保證存在於調用結果中。已刪除的項目會出現在 data 區塊中。尚未處理的項目在該 data 區塊中會標示為 null，並置於 unprocessedKeys 區塊。

BatchDeleteItem

該BatchDeleteItem請求對象可以讓你告訴AWS AppSync用來建立一個BatchWriteItem請求 DynamoDB 刪除可能跨多個表格的多個項目。對於此請求對象，您必須指定以下內容：

- 將從其中刪除項目的資料表名稱
- 將從個別資料表刪除項目的索引鍵

此時會套用 DynamoDB BatchWriteItem 限制，且可能不會提供任何條件表達式。

該BatchDeleteItem請求對象具有以下結構：

```
type DynamoDBBatchDeleteItemRequest = {
  operation: 'BatchDeleteItem';
  tables: {
    [tableName: string]: { [key: string]: any }[];
  };
};
```

欄位定義如下：

BatchDeleteItem 欄位

BatchDeleteItem欄位清單

`operation`

要執行的動態支援作業。若要執行 BatchDeleteItem DynamoDB 操作，這必須設為 BatchDeleteItem。此值為必填。

`tables`

要從中刪除項目的 DynamoDB 表。每個資料表都是 DynamoDB 索引鍵清單，代表要刪除之項目的主索引鍵。DynamoDB 項目可能具有單一雜湊鍵，或是雜湊鍵和排序索引鍵，具體取決於資料表結構。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。至少必須提供一個資料表。該tables值是必需的。

注意事項：

- 不同於 DeleteItem 操作，回應中未傳回完整刪除的項目。只會傳回已傳遞的索引鍵。
- 如果資料表中未刪除任何項目，該資料表的 data 區塊中就會顯示 null 元素。
- 調用結果是根據請求對象內提供的順序，每個表進行排序。
- 每個Delete命令, 裡面, aBatchDeleteItem是原子的。但是，批次可以部分處理。如果因錯誤而部分批次處理，則未處理的索引鍵會透過 unprocessedKeys 區塊傳回為部分的呼叫結果。
- BatchDeleteItem 限制為 25 個索引鍵。

對於以下示例函數請求處理程序：

```
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 })],
    },
  };
}
```

可透過 ctx.result 提供的呼叫結果如下所示：

```
{
   "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 包含錯誤的詳細資訊。鑰匙資料,未處理的金鑰，並且函數請求對像中提供的每個表鍵都保證存在於調用結果中。已刪除的項目會出現在 data 區塊中。尚未處理的項目在該 data 區塊中會標示為 null，並置於 unprocessedKeys 區塊。

BatchPutItem

該BatchPutItem請求對象可以讓你告訴AWS AppSync用來建立一個BatchWriteItem請求 DynamoDB 放置多個項目，可能跨越多個表格。對於此請求對象，您必須指定以下內容：

- 項目將放入其中的資料表名稱
- 將放入每個資料表的所有項目

此時會套用 DynamoDB BatchWriteItem 限制，且可能不會提供任何條件表達式。

該BatchPutItem請求對象具有以下結構：

```
type DynamoDBBatchPutItemRequest = {
  operation: 'BatchPutItem';
  tables: {
    [tableName: string]: { [key: string]: any}[];
  };
};
```

欄位定義如下：

BatchPutItem 欄位

BatchPutItem欄位清單

operation

要執行的動態支援作業。若要執行 BatchPutItem DynamoDB 操作，這必須設為 BatchPutItem。此值為必填。

tables

要放置項目的動態資料表。每個表格項目代表要為此特定表格插入的 DynamoDB 項目清單。至少必須提供一個資料表。此值為必填。

注意事項：

- 若插入成功，回應會傳回完全插入的項目。
- 如果資料表中並未插入任何項目，該資料表的 data 區塊中就會顯示 null 元素。

- 插入的項目是每個表，根據它們在請求對象中提供的順序進行排序。
- 每個Put命令, 裡面, aBatchPutItem是原子的，但是，批次可以部分處理。如果因錯誤而部分批次處理，則未處理的索引鍵會透過 unprocessedKeys 區塊傳回為部分的呼叫結果。
- BatchPutItem 限制為 25 個項目。

對於以下示例函數請求處理程序：

```
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 })],
    },
  };
}
```

可透過 ctx.result 提供的呼叫結果如下所示：

```
{
   "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 包含錯誤的詳細資訊。鑰匙資料,非处理程序分析，並且請求對像中提供的每個表鍵都保證存在於調用結果中。已插入的項目會出現在 data 區塊中。尚未處理的項目在該 data 區塊中會標示為 null，並置於 unprocessedItems 區塊。

TransactGetItems

該TransactGetItems請求對象可以讓你告訴AWS AppSync用來建立一個TransactGetItems向 DynamoDB 請求以擷取可能跨多個表格的多個項目。對於此請求對象，您必須指定以下內容：

- 從中擷取項目之每個請求項目的資料表名稱
- 從每個資料表擷取之每個請求項目的索引鍵

此時會套用 DynamoDB TransactGetItems 限制，且可能不會提供任何條件表達式。

該TransactGetItems請求對象具有以下結構：

```
type DynamoDBTransactGetItemsRequest = {
  operation: 'TransactGetItems';
  transactItems: { table: string; key: { [key: string]: any }; projection?:
 { expression: string; expressionNames?: { [key: string]: string }; }[];
  };
};
```

欄位定義如下：

TransactGetItems 欄位

TransactGetItems欄位清單

operation

要執行的動態支援作業。若要執行 TransactGetItems DynamoDB 操作，這必須設為 TransactGetItems。此值為必填。

`transactItems`

要包含的請求項目。此值是請求項目的陣列。必須提供至少一個請求項目。`transactItems` 值為必填。

`table`

要從中擷取項目的動態資料表。此值是資料表名稱的字串。`table` 值為必填。

`key`

代表要擷取之項目的主索引鍵的 DynamoDB 索引鍵。DynamoDB 項目可能具有單一雜湊鍵，或是雜湊鍵和排序索引鍵，具體取決於資料表結構。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。

`projection`

用來指定要從 DynamoDB 作業傳回之屬性的投影。若要取得有關投影的更多資訊，請參閱投影。此欄位為選用欄位。

注意事項：

- 如果交易成功，`items` 區塊中擷取項目的順序會與請求項目的順序相同。
- 交易是在all-or-nothing方式。如果任何請求項目造成錯誤，將不執行整個交易，而且將傳回錯誤詳細資料。
- 無法擷取的請求項目不是錯誤。相反地，null 元素會出現在對應位置的項目區塊。
- 如果交易的錯誤是TransactionCanceledException，該`cancellationReasons`圖塊將被填充。`cancellationReasons` 區塊中取消原因的順序將與請求項目的順序相同。
- 將 `TransactGetItems` 限制為 25 個請求項目。

對於以下示例函數請求處理程序：

```
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 }),
      },
    ],
  };
}
```

如果交易成功，而且僅擷取第一個請求項目，則 ctx.result 中的可用叫用結果如下所示：

```
{
    "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
}
```

如果交易失敗的原因TransactionCanceledException由第一個請求項引起的，調用結果可用於ctx.result如下所示：

```
{
    "items": null,
    "cancellationReasons": [
       {
           "type":"Sample error type",
           "message":"Sample error message"
       },
       {
           "type":"None",
           "message":"None"
       }
    ]
}
```

ctx.error 包含錯誤的詳細資訊。索引鍵 items 和 cancellationReasons 保證出現在 ctx.result 中。

TransactWriteItems

該TransactWriteItems請求對象可以讓你告訴AWS AppSync用來建立一個TransactWriteItems請求 DynamoDB 寫入多個項目，可能會寫入多個資料表。對於此請求對象，您必須指定以下內容：

- 每個請求項目的目標資料表名稱
- 每個請求項目要執行的操作。支援的作業類型有四種：PutItem,UpdateItem,DeleteItem，以及ConditionCheck
- 每個要寫入之請求項目的索引鍵

套用 DynamoDB TransactWriteItems 限制。

該TransactWriteItems請求對象具有以下結構：

```
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 欄位

TransactWriteItems欄位清單

欄位定義如下：

operation

要執行的動態支援作業。若要執行 TransactWriteItems DynamoDB 操作，這必須設為 TransactWriteItems。此值為必填。

transactItems

要包含的請求項目。此值是請求項目的陣列。必須提供至少一個請求項目。transactItems 值為必填。

對於 PutItem，欄位定義如下：

table

目的地動態資料表。此值是資料表名稱的字串。table 值為必填。

operation

要執行的動態支援作業。若要執行 PutItem DynamoDB 操作，這必須設為 PutItem。此值為必填。

key

代表要放置之項目的主索引鍵的 DynamoDB 索引鍵。DynamoDB 項目可能具有單一雜湊鍵，或是雜湊鍵和排序索引鍵，具體取決於資料表結構。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。此值為必填。

attributeValues

將放入 DynamoDB 的項目其餘屬性。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。此欄位為選用欄位。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，PutItem 要求會覆寫該項目的任何現有資料項目。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱交易條件運算式。此值是選用的。

對於 UpdateItem，欄位定義如下：

table

要更新的動態資料表。此值是資料表名稱的字串。table 值為必填。

operation

要執行的動態支援作業。若要執行 UpdateItem DynamoDB 操作，這必須設為 UpdateItem。此值為必填。

key

代表要更新之項目的主索引鍵的 DynamoDB 索引鍵。DynamoDB 項目可能具有單一雜湊鍵，或是雜湊鍵和排序索引鍵，具體取決於資料表結構。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。此值為必填。

update

該update區段可讓您指定描述如何更新 DynamoDB 中項目的更新運算式。如需如何撰寫更新運算式的詳細資訊，請參閱DynamoDBUpdateExpressions文件。此區段是必須的。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，UpdateItem 要求會更新現有的資料項目，無論項目的目前狀態為何。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱交易條件運算式。此值是選用的。

對於 DeleteItem，欄位定義如下：

table

要在其中刪除項目的 DynamoDB 表格。此值是資料表名稱的字串。table 值為必填。

operation

要執行的動態支援作業。若要執行 DeleteItem DynamoDB 操作，這必須設為 DeleteItem。此值為必填。

key

代表要刪除之項目的主索引鍵的 DynamoDB 鍵。DynamoDB 項目可能具有單一雜湊鍵，或是雜湊鍵和排序索引鍵，具體取決於資料表結構。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。此值為必填。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，DeleteItem 要求會刪除項目，無論該項目的目前狀態為何。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱交易條件運算式。此值是選用的。

對於 ConditionCheck，欄位定義如下：

table

要在其中檢查條件的 DynamoDB 表格。此值是資料表名稱的字串。table 值為必填。

operation

要執行的動態支援作業。若要執行 ConditionCheck DynamoDB 操作，這必須設為 ConditionCheck。此值為必填。

key

DynamoDB 鍵，代表要進行條件檢查之項目的主索引鍵。DynamoDB 項目可能具有單一雜湊鍵，或是雜湊鍵和排序索引鍵，具體取決於資料表結構。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。此值為必填。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱交易條件運算式。此值為必填。

注意事項：

- 如果成功，只會在回應中傳回請求項目的索引鍵。索引鍵的順序將與請求項目的順序相同。
- 交易是在all-or-nothing方式。如果任何請求項目造成錯誤，將不執行整個交易，而且將傳回錯誤詳細資料。
- 沒有兩個請求項目可以定位到相同項目。否則，他們會導致TransactionCanceledException錯誤。
- 如果交易的錯誤是TransactionCanceledException，該cancellationReasons圖塊將被填充。如果請求項目的條件檢查失敗，而且您未將 `returnValuesOnConditionCheckFailure` 指定為 `false`，將擷取資料表中存在的項目，並存放在 `cancellationReasons` 區塊之對應位置的 `item` 中。
- 將 `TransactWriteItems` 限制為 25 個請求項目。

對於以下示例函數請求處理程序：

```
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 }),
        },
      },
    ],
```

```
  };
}
```

如果交易成功，ctx.result 中可用的叫用結果如下所示：

```
{
    "keys": [
       // Key of the PutItem request
       {
           "post_id": "p1",
       },
       // Key of the UpdateItem request
       {
           "author_id": "a1"
       }
    ],
    "cancellationReasons": null
}
```

如果交易因條件檢查失敗而失敗PutItem請求，調用結果ctx.result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 包含錯誤的詳細資訊。索引鍵 keys 和 cancellationReasons 保證出現在 ctx.result 中。

類型系統 (請求對應)

使用時AWS AppSync用來呼叫您的動態資料表的動態資料表、AWS AppSync需要知道在該調用中使用的每個值的類型。這是因為 DynamoDB 支援的型別原語比 GraphQL 或 JSON 更多 (例如集合和二進位資料)。AWSAppSync在 GraphQL 和 DynamoDB 之間進行轉換時需要一些提示，否則必須對資料表中的結構方式做出一些假設。

如需有關資料類型的詳細資訊，請參閱 DynamoDB資料類型描述元和資料類型文件。

DynamoDB 值由包含單一索引鍵值組的 JSON 物件表示。此索引鍵會指定 DynamoDB 類型，而值則會指定值本身。在下列範例中，S 鍵代表值是字串，`identifier` 值則是字串值本身。

```
{ "S" : "identifier" }
```

請注意，JSON 物件不能有一個以上的索引值對。如果指定了多個鍵值對，則不會解析請求對象。

DynamoDB 值可在您需要指定值的要求物件中的任何位置使用。您需要執行這項操作的一些地方包括：`key` 和 `attributeValue` 區段，以及表達式區段的 `expressionValues` 區段。在下列範例中，DynamoDB 字串值`identifier`正在指派給`id`在一個字段`key`部分（也許在`GetItem`請求對象）。

```
"key" : {
   "id" : { "S" : "identifier" }
}
```

支援的類型

AWS AppSync支援下列 DynamoDB 純量、文件和集合類型：

字串類型 **S**

單一字串值。一個字符串值表示為：

```
{ "S" : "some string" }
```

使用範例：

```
"key" : {
   "id" : { "S" : "some string" }
}
```

字串集類型 SS

字串值的集合。DynamoDB 字串集的值由下列方式表示：

```
{ "SS" : [ "first value", "second value", ... ] }
```

使用範例：

```
"attributeValues" : {
   "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
}
```

數字類型 N

單一數值。動態資料庫編號值的表示方式為：

```
{ "N" : 1234 }
```

使用範例：

```
"expressionValues" : {
   ":expectedVersion" : { "N" : 1 }
}
```

數字集類型 NS

數值的集合。動態數字集的值由下列方式表示：

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

使用範例：

```
"attributeValues" : {
   "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }
}
```

二進位類型 B

二進位值。二進位值的表示方式為：

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

請注意，該值實際上是一個字符串，其中字符串是二進制數據的 base64 編碼表示。AWSAppSync 在將此字串傳送至 DynamoDB 之前，將此字串解碼回其二進位值。AWSAppSync使用 RFC 2045 所定義的 base64 解碼方案：忽略不在 base64 字母中的任何字元。

使用範例：

```
"attributeValues" : {
   "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

二進位集類型 **BS**

二進位值的集合。二進位集的值表示為：

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

請注意，該值實際上是一個字符串，其中字符串是二進制數據的 base64 編碼表示。AWSAppSync 在將此字串傳送至 DynamoDB 之前，將此字串解碼回其二進位值。AWSAppSync使用 RFC 2045 所定義的 base64 解碼方案：忽略不在 base64 字母中的任何字元。

使用範例：

```
"attributeValues" : {
   "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }
}
```

布林類型 **BOOL**

布林值。一個布林值表示為：

```
{ "BOOL" : true }
```

請注意，只有 `true` 和 `false` 為有效值。

使用範例：

```
"attributeValues" : {
   "orderComplete" : { "BOOL" : false }
}
```

清單類型 **L**

任何其他受支援的 DynamoDB 值的清單。動態 B 清單值的表示方式為：

```
{ "L" : [ ... ] }
```

請注意，該值是複合值，其中清單可以包含零個或多個任何受支援的 DynamoDB 值 (包括其他清單)。清單也可包含混合的不同類型。

使用範例：

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

映射類型 **M**

代表其他受支援 DynamoDB 值之索引鍵值組的無序集合。一個地圖值表示為：

```
{ "M" : { ... } }
```

請注意，映射可包含零或多個索引值對。索引鍵必須是字串，且該值可以是任何支援的 DynamoDB 值 (包括其他對應)。映射也可包含混合的不同類型。

使用範例：

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

Null 類型 **NULL**

null 值。一個空值表示為：

```
{ "NULL" : null }
```

使用範例：

```
"attributeValues" : {
   "phoneNumbers" : { "NULL" : null }
}
```

如需每種類型的詳細資訊，請參閱 DynamoDB 文件。

類型系統（響應映射）

當收到來自動態支援的回應時，AWS AppSync自動將其轉換為 GraphQL 和 JSON 原始類型。DynamoDB 中的每個屬性都會在回應處理常式的內容中解碼並傳回。

例如，如果 DynamoDB 傳回下列內容：

```
{
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

從管道解析器返回結果時，AWS AppSync將其轉換為圖形 SQL 和 JSON 類型，如下所示：

```
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

本節說明如何AWS AppSync會轉換下列 DynamoDB 純量、文件和集合類型：

字串類型 S

單一字串值。DynamoDB 字串值會以字串形式傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 字串值：

```
{ "S" : "some string" }
```

AWS AppSync它轉換為一個字符串：

```
"some string"
```

字串集類型 SS

字串值的集合。DynamoDB 字串集的值會以字串清單的形式傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 字串集值：

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync將其轉換為字符串列表：

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

數字類型 N

單一數值。DynamoDB 編號值會以數字形式傳回。

例如，如果 DynamoDB 傳回下列動態資料庫編號值：

```
{ "N" : 1234 }
```

AWS AppSync將其轉換為一個數字：

```
1234
```

數字集類型 NS

數值的集合。DynamoDB 數字集的值會以數字清單的形式傳回。

例如，如果 DynamoDB 傳回下列數字集值：

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync將其轉換為數字列表：

```
[ 67.8, 12.2, 70 ]
```

二進位類型 B

二進位值。DynamoDB 二進位值會以包含該值 base64 表示法的字串形式傳回。

例如，如果 DynamoDB 傳回下列二進位值：

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync將其轉換為包含值 base64 表示法的字串：

```
"SGVsbG8sIFdvcmxkIQo="
```

請注意，二進位資料以 base64 編碼配置編碼，如 RFC 4648 和 RFC 2045 中所定義。

二進位集類型 **BS**

二進位值的集合。DynamoDB 二進位集值會傳回為包含值 base64 表示法的字串清單。

例如，如果 DynamoDB 傳回下列二進位集值：

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync將其轉換為包含值 base64 表示的字符串列表：

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

請注意，二進位資料以 base64 編碼配置編碼，如 RFC 4648 和 RFC 2045 中所定義。

布林類型 **BOOL**

布林值。DynamoDB 布林值會以布林值的形式傳回。

例如，如果 DynamoDB 傳回下列布林值：

```
{ "BOOL" : true }
```

AWS AppSync將其轉換為布爾值：

```
true
```

清單類型 **L**

任何其他受支援的 DynamoDB 值的清單。DynamoDB 清單值會以值清單的形式傳回，其中每個內部值也會轉換。

例如，如果 DynamoDB 傳回下列動態資料 B 清單值：

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

AWS AppSync將其轉換為轉換值的列表：

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

映射類型 **M**

任何其他受支援的 DynamoDB 值的索引鍵/值集合。DynamoDB 對應值會以 JSON 物件的形式傳回，其中每個索引鍵/值也會轉換。

例如，如果 DynamoDB 傳回下列 DynamoDB 對應值：

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

AWS AppSync將其轉換為一個 JSON 對象：

```
{
   "someString" : "A string value",
   "someNumber" : 1,
   "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

Null 類型 **NULL**

null 值。

例如，如果 DynamoDB 傳回下列空值：

```
{ "NULL" : null }
```

AWS AppSync將其轉換為空：

```
null
```

篩選條件

使用查詢動態資料庫中的物件時Query和Scan作業時，您可以選擇性地指定filter評估結果並僅返回所需的值。

的過濾器屬性Query或者Scan請求具有以下結構：

```
type DynamoDBExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
};
```

欄位定義如下：

expression

查詢表達式。如需如何撰寫篩選運算式的詳細資訊，請參閱DynamoDBQueryFilter和DynamoDBScanFilter文件。必須指定此欄位。

expressionNames

表達式屬性 name 預留位置的替代，形式為鍵值組。索引鍵對應至用於 expression 的名稱預留位置。該值必須是對應至 DynamoDB 中項目的屬性名稱的字串。此欄位為選用的，應只能填入用於 expression 中表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性 value 預留位置的替代，形式為鍵值組。鍵對應用於 expression 的值預留位置，值必須是類型值。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。此必須指定。此欄位為選用的，應只能填入用於 expression 中表達式屬性值預留位置的替代。

範例

下列範例是請求的篩選器區段，其中從 DynamoDB 擷取的項目只有在標題以title參數。

在這裡，我們使用util.transform.toDynamoDBFilterExpression從物件自動建立濾鏡：

```
const filter = util.transform.toDynamoDBFilterExpression({
  title: { beginsWith: 'far away' },
});

const request = {};
request.filter = JSON.parse(filter);
```

這將生成以下過濾器：

```
{
  "filter": {
    "expression": "(begins_with(#title,:title_beginsWith))",
    "expressionNames": { "#title": "title" },
    "expressionValues": {
      ":title_beginsWith": { "S": "far away" }
    }
  }
}
```

條件表達式

當您在 DynamoDB 中使用PutItem,UpdateItem，以及DeleteItemDynamoDB 作業時，您可以選擇性地指定條件運算式，以根據在執行作業之前 DynamoDB 中已存在的物件狀態來控制請求是否成功。

該AWS AppSyncDynamoDB 函數允許在中指定條件運算式PutItem,UpdateItem，以及DeleteItem要求物件，以及條件失敗且物件未更新時應遵循的策略。

範例 1

以下PutItem請求對象沒有條件表達式。因此，即使已存在具有相同金鑰的項目，它也會將項目置於 DynamoDB 中，進而覆寫現有項目。

```
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),
  };
}
```

範例 2

以下PutItem對象確實有一個條件表達式，只有當具有相同鍵的項目執行時才允許操作成功不存在於動態資料庫中。

```
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)" }
  };
}
```

依預設，如果條件檢查失敗，AWS AppSync動態支援函式在中提供錯誤ctx.error。您可以在 DynamoDB 中返回突變的錯誤和對象的當前值data欄位中的error部分的圖形 SQL 響應。

然而，AWS AppSyncDynamoDB 函數提供一些其他功能，可協助開發人員處理一些常見的邊緣案例：

- 如果AWS AppSyncDynamoDB 函數可以判斷 DynamoDB 中的目前值是否符合所需的結果，並將作業視為無論如何都成功。
- 您可以將函數設定為叫用自訂 Lambda 函數，以決定如何AWS AppSyncDynamoDB 函數應該處理失敗。

這些在中更詳細地描述處理條件檢查失敗部分。

如需 DynamoDB 條件運算式的詳細資訊，請參閱DynamoDBConditionExpressions文件。

指定條件

該PutItem,UpdateItem，以及DeleteItem請求對象都允許一個可選condition要指定的部分。若省略，則不會有條件檢查。若指定，條件必須為 true，操作才會成功。

condition 區段的結構如下：

```
type ConditionCheckExpression = {
  expression: string;
  expressionNames?: { [key: string]: string};
  expressionValues?: { [key: string]: any};
  equalsIgnore?: string[];
  consistentRead?: boolean;
  conditionalCheckFailedHandler?: {
    strategy: 'Custom' | 'Reject';
    lambdaArn?: string;
  };
};
```

下列欄位指定條件：

expression

更新表達式本身。如需如何撰寫條件運算式的詳細資訊，請參閱DynamoDBConditionExpressions文件。必須指定此欄位。

expressionNames

表達式屬性名稱預留位置的替代，形式為索引鍵值對。該鍵對應於中使用的名稱佔位符表達，而且值必須是對應至 DynamoDB 中項目的屬性名稱的字串。此欄位為選用的，應只能填入於表達式中所用表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性值預留位置的替代，形式為索引值對。鍵對應用於表達式的值預留位置，值必須是類型值。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。此必須指定。此欄位為選用的，應只能填入用於表達式中表達式屬性值預留位置的替代。

剩下的字段告訴AWS AppSyncDynamoDB 函數如何處理條件檢查失敗：

equalsIgnore

如果條件檢查失敗，則使用PutItem操作，該AWS AppSyncDynamoDB 函式會將目前在 DynamoDB 中的項目與它嘗試寫入的項目進行比較。如果兩者相同，則操作視為成功。您可以使用equalsIgnore用於指定屬性清單的欄位AWS AppSync執行該比較時應忽略。例如，如果唯一的區別是version屬性，它將操作視為成功。此欄位為選用欄位。

consistentRead

當條件檢查失敗時，AWS AppSync使用強式一致性讀取，從 DynamoDB 取得項目的目前值。您可以使用此欄位告訴AWS AppSyncDynamoDB 函數可改為使用最終一致性讀取。此欄位為選用，預設值為 `true`。

conditionalCheckFailedHandler

本節允許您指定如何AWS AppSyncDynamoDB 函數將 DynamoDB 中的目前值與預期的結果進行比較後，會處理條件檢查失敗。此區段為選用。若省略，則會預設為 `Reject` 策略。

strategy

該策略AWS AppSyncDynamoDB 函數會在將 DynamoDB 中的目前值與預期的結果進行比較之後才會採用。此欄位為必填，且採用下列可能值：

Reject

突變失敗，並且 DynamoDB 中的突變和對象的當前值出現錯誤`data`欄位中的`error`部分的圖形 SQL 響應。

Custom

該AWS AppSyncDynamoDB 函數會叫用自訂 Lambda 函數，以決定如何處理條件檢查失敗。當 `strategy` 設定為 `Custom`，`lambdaArn` 欄位必須包含要叫用 Lambda 函數的 ARN。

lambdaArn

要叫用的 Lambda 函數的 ARN，可決定如何AWS AppSyncDynamoDB 函數應該處理條件檢查失敗。只有在 `strategy` 設定為 `Custom` 時，此欄位才必須指定。如需如何使用此功能的詳細資訊，請參閱處理條件檢查失敗。

處理條件檢查失敗

當條件檢查失敗時，AWS AppSyncDynamoDB 函數可以傳遞突變的錯誤以及物件的目前值，方法是使用`util.appendError`實用程序。這增加了`data`欄位中的`error`部分的圖形 SQL 響應。然而，AWS AppSyncDynamoDB 函數提供一些其他功能，可協助開發人員處理一些常見的邊緣案例：

- 如果AWS AppSyncDynamoDB 函數可以判斷 DynamoDB 中的目前值是否符合所需的結果，並將作業視為無論如何都成功。
- 您可以將函數設定為叫用自訂 Lambda 函數，以決定如何AWS AppSyncDynamoDB 函數應該處理失敗。

此程序的流程圖為：

檢查所需的結果

當條件檢查失敗時，AWS AppSync動態支援函數會執行GetItem要求從 DynamoDB 取得項目目前的值。在預設情況下，它會使用強式一致性讀取，但這可使用 condition 區塊中的 consistentRead 欄位來設定，並和預期結果進行比較：

- 對於PutItem操作，該AWS AppSyncDynamoDB 函數將目前值與其嘗試寫入的值進行比較，排除中列出的任何屬性equalsIgnore從比較。如果項目相同，則會將作業視為成功，並傳回從 DynamoDB 擷取的項目。否則，其將按照設定的策略。

  例如，如果PutItem請求對象如下所示：

```
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']
    }
  };
}
```

  而目前在 DynamoDB 中的項目外觀如下：

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

  該AWS AppSyncDynamoDB 函數會將它嘗試寫入的項目與當前值進行比較，看到唯一的區別是version字段，但因為它被配置為忽略version欄位中，它會將作業視為成功，並傳回從 DynamoDB 擷取的項目。

- 對於DeleteItem操作，AWS AppSyncDynamoDB 函數會檢查以確認某個項目是否已從 DynamoDB 傳回。如果沒有項目傳回，則操作視為成功。否則，其將按照設定的策略。
- 對於UpdateItem操作，AWS AppSyncDynamoDB 函數沒有足夠的資訊來判斷 DynamoDB 中目前的項目是否符合預期的結果，因此會遵循設定的策略。

如果 DynamoDB 中物件的目前狀態與預期的結果不同，AWS AppSyncDynamoDB 函數會遵循設定的策略，以拒絕突變，或叫用 Lambda 函數來決定下一步要執行的動作。

遵循「拒絕」策略

當遵循Reject策略, 該AWS AppSyncDynamoDB 函數會針對變異傳回錯誤，而 DynamoDB 中物件的目前值也會以data欄位中的error部分的圖形 SQL 響應。從 DynamoDB 傳回的項目會透過函數回應處理常式，將其轉譯成用戶端預期的格式，並依選集進行篩選。

例如，假設變動要求如下：

```
mutation {
    updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
        Name
        theVersion
    }
}
```

如果從 DynamoDB 傳回的項目如下所示：

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

函數響應處理程序如下所示：

```
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 回應的外觀如下所示：

```
{
  "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
      },
      ...
    }
  ]
}
```

此外，如果傳回物件中的任何欄位皆由其他解析程式填入，且變動成功，則在 `error` 區段中傳回物件時，不會解析這些物件。

遵循「自定義」策略

當遵循Custom策略, 該AWS AppSyncDynamoDB 函數會叫用 Lambda 函數來決定下一步該做什麼。Lambda 函數會選擇下列其中一個選項：

- `reject` 變動。這告訴AWS AppSyncDynamoDB 函數的行為就像設定的策略一樣Reject，傳回 DynamoDB 中物件的突變和目前值的錯誤，如前一節所述。
- `discard` 變動。這告訴AWS AppSyncDynamoDB 函數會以無訊息方式忽略條件檢查失敗，並傳回 DynamoDB 中的值。
- `retry` 變動。這告訴AWS AppSyncDynamoDB 函數用於使用新的請求對象重試突變。

Lambda 叫用要求

該AWS AppSyncDynamoDB 函數會叫用中指定的 Lambda 函數lambdaArn。它會使用資料來源上所設定的相同 `service-role-arn`。叫用承載的結構如下：

```
{
    "arguments": { ... },
    "requestMapping": {... },
    "currentValue": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

欄位定義如下：

arguments

來自 GraphQL 變動的引數。這與中的請求對象可用的參數相同`context.arguments`。

requestMapping

此操作的請求對象。

currentValue

物件在 DynamoDB 中的目前值。

resolver

有關的信息AWS AppSync解析器或功能。

identity

發起人的相關資訊。這與中的請求對象可用的身份信息相同`context.identity`。

承載的完整範例：

```
{
    "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 叫用回應

Lambda 函數可以檢查叫用承載，並套用任何商業邏輯來決定如何AWS AppSyncDynamoDB 函數應該處理失敗。處理條件檢查失敗有三個選項：

- `reject` 變動。此選項的回應承載必須具有此架構：

```
{
    "action": "reject"
}
```

這告訴AWS AppSyncDynamoDB 函數的行為就像設定的策略一樣`Reject`，傳回 DynamoDB 中物件的突變和目前值的錯誤，如上節所述。

- `discard` 變動。此選項的回應承載必須具有此架構：

```
{
    "action": "discard"
}
```

這告訴AWS AppSyncDynamoDB 函數會以無訊息方式忽略條件檢查失敗，並傳回 DynamoDB 中的值。

- `retry` 變動。此選項的回應承載必須具有此架構：

```
{
    "action": "retry",
    "retryMapping": { ... }
}
```

這告訴AWS AppSyncDynamoDB 函數用於使用新的請求對象重試突變。的結構`retryMapping`區段取決於 DynamoDB 作業，而且是該作業之完整要求物件的子集。

若是 `PutItem`，`retryMapping` 區段的結構如下。若要取得的說明`attributeValues`欄位中，請參閱PutItem。

```
{
    "attributeValues": { ... },
    "condition": {
        "equalsIgnore" = [ ... ],
        "consistentRead" = true
    }
}
```

若是 `UpdateItem`，`retryMapping` 區段的結構如下。若要取得的說明`update`區段，請參閱UpdateItem。

```
{
    "update" : {
        "expression" : "someExpression"
        "expressionNames" : {
```

```
            "#foo" : "foo"
        },
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "condition": {
        "consistentRead" = true
    }
}
```

若是 DeleteItem，retryMapping 區段的結構如下。

```
{
    "condition": {
        "consistentRead" = true
    }
}
```

無法指定使用不同的操作或索引鍵。該AWS AppSyncDynamoDB 函數只允許在相同物件上重試相同作業。此外，condition 區段不允許指定 conditionalCheckFailedHandler。如果重試失敗，AWS AppSync動態支援函數遵循Reject策略。

此為 Lambda 函數處理失敗 PutItem 要求的範例。商業邏輯的重點是發起人為何。如果它是由jeffTheAdmin，它會重試請求，更新version和expectedVersion從目前在 DynamoDB 中的項目。否則，它會拒絕變動。

```
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)
};
```

交易條件運算式

交易條件運算式可在下列四種作業類型的要求中使用TransactWriteItems,即,PutItem,DeleteItem,UpdateItem，以及ConditionCheck。

對於PutItem,DeleteItem，以及UpdateItem，交易條件運算式是選用的。對於ConditionCheck，需要交易條件運算式。

範例 1

以下交易DeleteItem函數請求處理程序沒有條件表達式。因此，它會刪除 DynamoDB 中的項目。

```
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 }),
      }
    ],
```

```
  };
}
```

範例 2

以下交易DeleteItem函數請求處理程序確實有一個事務條件表達式，只有當該職位的作者等於特定名稱時才允許操作成功。

```
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 },
        }),
      }
    ],
  };
}
```

如果條件檢查失敗，會導致 TransactionCanceledException，並且會在 ctx.result.cancellationReasons 中傳回錯誤詳細資料。請注意，根據預設，DynamoDB 中使條件檢查失敗的舊項目將傳回ctx.result.cancellationReasons。

指定條件

該PutItem,UpdateItem，以及DeleteItem請求對象都允許一個可選condition要指定的部分。若省略，則不會有條件檢查。若指定，條件必須為 true，操作才會成功。ConditionCheck 必須有待指定的 condition 區段。條件必須為 true，整個交易才會成功。

condition 區段的結構如下：

```
type TransactConditionCheckExpression = {
  expression: string;
```

```
  expressionNames?: { [key: string]: string };
  expressionValues?: { [key: string]: string };
  returnValuesOnConditionCheckFailure: boolean;
};
```

下列欄位指定條件：

expression

更新表達式本身。如需如何撰寫條件運算式的詳細資訊，請參閱DynamoDBConditionExpressions文件。必須指定此欄位。

expressionNames

表達式屬性名稱預留位置的替代，形式為索引鍵值對。該鍵對應於中使用的名稱佔位符表達，而且值必須是對應至 DynamoDB 中項目的屬性名稱的字串。此欄位為選用的，應只能填入於表達式中所用表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性值預留位置的替代，形式為索引值對。鍵對應用於表達式的值預留位置，值必須是類型值。如需如何指定「輸入值」的詳細資訊，請參閱類型系統 (請求對應)。此必須指定。此欄位為選用的，應只能填入用於表達式中表達式屬性值預留位置的替代。

returnValuesOnConditionCheckFailure

指定當條件檢查失敗時，是否要將 DynamoDB 中的項目擷取回來。擷取的項目將位於 `ctx.result.cancellationReasons[<index>].item` 中，其中 `<index>` 是條件檢查失敗之請求項目的索引。此值預設為 true。

投影

讀取動態資料庫中的物件時，`GetItem`,`Scan`,`Query`,`BatchGetItem`，以及`TransactGetItems`作業時，您可以選擇性地指定識別所需屬性的投影。投影性質具有下列結構，類似於篩選器：

```
type DynamoDBExpression = {
  expression: string;
  expressionNames?: { [key: string]: string}
};
```

欄位定義如下：

expression

投影表達式，它是一個字符串。若要擷取單一屬性，請指定其名稱。對於多個屬性，名稱必須是逗號分隔值。如需撰寫投影運算式的詳細資訊，請參閱投影運算式文件。此欄位為必填。

expressionNames

表達式屬性的替換名稱鍵值對形式的預留位置。索引鍵對應至用於 expression 的名稱預留位置。該值必須是對應至 DynamoDB 中項目的屬性名稱的字串。此欄位為選擇性欄位，且只應填入中使用的運算式屬性名稱預留位置符號的替代expression。有關更多信息expressionNames，請參閱說明文件。

範例 1

下面的例子是一個投影部分JavaScript函數，其中只有屬性author和id會從下列項目傳回：

```
projection : {
    expression : "#author, id",
    expressionNames : {
        "#author" : "author"
    }
}
```

> **Tip**
>
> 您可以使用以下方式存取 GraphQL 請求選擇集selectionSetList。此欄位可讓您根據需求動態構建投影運算式的框架。

> **Note**
>
> 搭配使用投影表示式時Query和Scan作業, 值select必須是SPECIFIC_ATTRIBUTES。如需詳細資訊，請參閱說明文件。

JavaScript 解析器函數參考 OpenSearch

Amazon OpenSearch 服務的AWS AppSync 解析器可讓您使用 GraphQL 存放和擷取帳戶中現有 OpenSearch 服務網域中的資料。此解析器的工作原理是允許您將傳入的 GraphQL 請求映射

到 OpenSearch 服務請求中，然後將 OpenSearch 服務響應映射回 GraphQL。本節說明支援之 OpenSearch Service 作業的函數要求和回應處理常式。

要求

大多數的 OpenSearch 服務請求對象都有一個共同的結構，其中只有幾個部分改變。下列範例會針對 OpenSearch Service 網域執行搜尋，其中文件的類型為，`post`且會在其中編製索引`id`。搜尋參數定義於 `body` 區段，許多常見的查詢子句定義於 `query` 欄位。此範例將搜尋在 `"Nadia"` 欄位中包含 `"Bailey"` 或 `author` (或兩者) 的文件：

```
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' } },
            ],
          },
        },
      },
    },
  };
}
```

回應

與其他資料來源一樣， OpenSearch 服務會傳送需AWS AppSync 要轉換為 GraphQL 的回應。

大多數 GraphQL 查詢都在從 OpenSearch 服務響應中查找`_source`字段。因為您可以執行搜尋以傳回個別文件或文件清單，因此在 OpenSearch Service 中使用兩種常見的回應模式：

結果清單

```
export function response(ctx) {
  const entries = [];
  for (const entry of ctx.result.hits.hits) {
    entries.push(entry['_source']);
  }
  return entries;
}
```

個別項目

```
export function response(ctx) {
  return ctx.result['_source']
}
```

operation 欄位

（僅限請求處理程序）

發送到 OpenSearch 服務域的 HTTP 方法或動詞（獲取，AWS AppSync 發布，放置，頭或刪除）。金鑰與值皆必須為字串。

```
"operation" : "PUT"
```

path 欄位

（僅限請求處理程序）

來源 OpenSearch 服務要求的搜尋路徑AWS AppSync。這會形成操作的 HTTP 動詞的 URL。金鑰與值皆必須為字串。

```
"path" : "/indexname/type"

"path" : "/indexname/type/_search"
```

評估要求處理常式時，此路徑會做為 HTTP 要求的一部分傳送，包括 OpenSearch 服務網域。例如，之前的範例可轉譯為：

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

params 欄位

（僅限請求處理程序）

用來指定搜尋執行的動作，最常見的方式是在主體內部設定查詢值。不過，有多項其他功能可設定，例如回應的格式。

- 標頭

  標頭資訊，以金鑰值對形式。金鑰與值皆必須為字串。例如：

  ```
  "headers" : {
      "Content-Type" : "application/json"
  }
  ```

  > **Note**
  >
  > AWS AppSync 目前僅支援 JSON 作為Content-Type.

- queryString

  金鑰值對，指定常見的選項，例如 JSON 回應的程式碼格式。金鑰與值皆必須為字串。例如，如果您希望獲得非常完整格式的 JSON，請使用：

  ```
  "queryString" : {
      "pretty" : "true"
  }
  ```

- 本文

  這是請求的主要部分，允許AWS AppSync 向您的 OpenSearch 服務域創建格式良好的搜索請求。金鑰必須是由物件組成的字串。以下顯示幾個示範。

範例 1

傳回城市符合「seattle」的所有文件：

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

```
    operation: 'GET',
    path: '/id/post/_search',
    params: {
      headers: {},
      queryString: {},
      body: { from: 0, size: 50, query: { match: { city: 'seattle' } } },
    },
  };
}
```

範例 2

傳回所有符合「washington」做為城市或州的文件：

```
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'] },
        },
      },
    },
  };
}
```

傳遞變數

（僅限請求處理程序）

您也可以在要求處理常式中傳遞變數做為評估的一部分。例如，假設您有一個 GraphQL 查詢如下：

```
query {
    searchForState(state: "washington"){
        ...
    }
```

```
}
```

函數請求處理程序可以是以下內容：

```
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 的解析器函數參考

您可以使用 AWS AppSync 函數和解析器來叫用帳戶中的 Lambda 函數。在將 Lambda 函數傳回給客戶之前，您可以調整請求承載和來自 Lambda 函數的回應。您也可以指定要在要求物件中執行的作業類型。本節說明支援 Lambda 作業的要求。

請求物件

Lambda 要求物件會處理與 Lambda 函數相關的欄位：

```
export type LambdaRequest = {
  operation: 'Invoke' | 'BatchInvoke';
  invocationType?: 'RequestResponse' | 'Event';
  payload: unknown;
};
```

以下是一個使用`invoke`操作的示例，其有效負載數據是 GraphQL 架構中的`getPost`字段及其來自上下文的參數：

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

整個對應文件會作為輸入傳遞至 Lambda 函數，以便前面的範例現在看起來像這樣：

```
{
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "input": {
        "id": "postId1",
      }
    }
  }
}
```

作業

Lambda 資料來源可讓您在operation欄位中定義兩項作業：Invoke和BatchInvoke。該Invoke操作讓 AWS AppSync 知道為每個 GraphQL 字段解析器調用您的 Lambda 函數。BatchInvoke指示對 AWS AppSync 目前 GraphQL 欄位進行批次處理要求。operation 欄位是必要的。

對於Invoke，已解析的請求與 Lambda 函數的輸入有效負載相符。讓我們修改上面的例子：

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

這被解決並傳遞給 Lambda 函數，它可能看起來像這樣：

```
{
```

```
  "operation": "Invoke",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

對於BatchInvoke，請求會套用至批次中的每個欄位解析程式。為了簡潔起見，將所有請求payload值合 AWS AppSync 併到匹配請求對象的單個對象下的列表中。以下示例請求處理程序顯示了合併：

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    payload: ctx,
  };
}
```

此請求會評估並解析為下列對映文件：

```
{
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

清單中的每個元素都對應於payload單一批次項目。Lambda 函數還預計將返回與請求中發送的項目順序匹配的列表形狀響應：

```
[
  { "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
]
```

承載

該payload字段是用於將任何數據傳遞給 Lambda 函數的容器。如果operation欄位設定為BatchInvoke，則將現有payload值 AWS AppSync 包裝到清單中。此 payload 欄位為選用。

調用類型

Lambda 資料來源可讓您定義兩種叫用類型：RequestResponse和Event. 呼叫型別與 Lambda API 中定義的叫用型別同義。RequestResponse叫用類型可讓您同步 AWS AppSync 呼叫 Lambda 函數以等待回應。呼Event叫可讓您以非同步方式叫用 Lambda 函數。如需 Lambda 如何處理Event叫用類型要求的詳細資訊，請參閱非同步叫用。此 invocationType 欄位為選用。如果此欄位未包含在要求中， AWS AppSync 將會預設為RequestResponse呼叫類型。

對於任何invocationType欄位，已解析的請求都會與 Lambda 函數的輸入有效負載相符。讓我們修改上面的例子：

```
export function request(ctx) {
  return {
    operation: 'Invoke',
    invocationType: 'Event',
    payload: { field: 'getPost', arguments: ctx.args },
  };
}
```

這被解決並傳遞給 Lambda 函數，它可能看起來像這樣：

```
{
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

當BatchInvoke作業與Event叫用類型欄位搭配使用時，會以上述相同方式合 AWS AppSync 併欄位解析程式，並將要求作為非同步事件傳遞至 Lambda 函數，並將其作為值清單傳遞至您的 Lambda 函數。payload來自Event調用類型請求的響應會導致沒有響應處理程序的null值：

```
{
```

```
  "data": {
    "field": null
  }
}
```

我們建議您停用Event叫用類型解析器的解析器快取，因為如果有快取命中，就不會將這些解析程式傳送至 Lambda。

回應物件

與其他資料來源一樣，您的 Lambda 函數會傳送回應， AWS AppSync 該回應必須轉換為 GraphQL 類型。Lambda 函數的結果包含在context結果屬性 (context.result) 中。

如果 Lambda 函數回應的形狀符合 GraphQL 類型的形狀，您可以使用下列函數回應處理常式轉寄回應：

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

響應對象沒有必需字段或形狀限制。但是，由於 GraphQL 是強類型的，因此已解析的回應必須符合預期的 GraphQL 類型。

Lambda 函數批次回應

如果operation欄位設定為BatchInvoke，則 AWS AppSync需要 Lambda 函數回來的項目清單。為了 AWS AppSync 將每個結果映射回原始請求項，響應列表必須在大小和順序匹配。在響應列表中有null項目ctx.result是有效的; 相應地設置為 null。

JavaScript 資料來源的解析器函數參考 EventBridge

與 EventBridge 資料來源搭配使用的 AWS AppSync 解析器函數請求和回應可讓您將自訂事件傳送到 Amazon EventBridge 匯流排。

請求

要求處理常式可讓您將多個自訂事件傳送至 EventBridge 事件匯流排：

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

```
    "operation" : "PutEvents",
    "events" : [{}]
  }
}
```

請 EventBridge PutEvents求具有以下類型定義：

```
type PutEventsRequest = {
  operation: 'PutEvents'
  events: {
    source: string
    detail: { [key: string]: any }
    detailType: string
    resources?: string[]
    time?: string // RFC3339 Timestamp format
  }[]
}
```

回應

如果PutEvents作業成功，則來自 EventBridge 的回應會包含在ctx.result：

```
export function response(ctx) {
  if(ctx.error)
    util.error(ctx.error.message, ctx.error.type, ctx.result)
  else
    return ctx.result
}
```

在中執行InternalExceptions或等PutEvents作業時發生Timeouts的錯誤ctx.error。如需常見錯誤 EventBridge的清單，請參閱EventBridge 常見錯誤參考資料。

result將具有以下類型定義：

```
type PutEventsResult = {
  Entries: {
    ErrorCode: string
    ErrorMessage: string
    EventId: string
  }[]
  FailedEntryCount: number
```

```
}
```

- 作品

  攝入的事件結果，無論是成功還是不成功。如果擷取成功，則項目EventID中有。否則，您可以使用ErrorCode和ErrorMessage來識別項目的問題。

  對於每個記錄，響應元素的索引是相同的請求數組中的索引。

- FailedEntryCount

  失敗項目的數目。此值表示為整數。

如需有關的回應的詳細資訊PutEvents，請參閱PutEvents。

範例回應範例 1

下列範例是具有兩個成功事件的PutEvents作業：

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ],
    "FailedEntryCount" : 0
}
```

範例回應範例 2

下列範例是包含三個事件、兩個成功和一個失敗的PutEvents作業：

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        },
```

```
        {
            "ErrorCode" : "SampleErrorCode",
            "ErrorMessage" : "Sample Error Message"
        }
    ],
    "FailedEntryCount" : 1
}
```

PutEvents 欄位

- 版本

  對於所有請求對應範本而言，此version欄位會定義範本使用的版本。此欄位為必填。此值2018-05-29是 EventBridge 對應範本支援的唯一版本。

- 操作

  唯一支援的作業是PutEvents。此作業可讓您將自訂事件新增至事件匯流排。

- 事件

  將新增至事件匯流排的事件陣列。這個數組應該有 1-10 個項目的分配。

  Event 物件包含以下欄位：

  - "source": 定義事件來源的字串。
  - "detail"：可用來附加事件相關資訊的 JSON 物件。此欄位可以是空的 map ({ })。
  - "detailType：識別事件類型的字串。
  - "resources"：識別事件中涉及的資源的 JSON 字符串數組。此欄位可以是空陣列。
  - "time": 以字串形式提供的事件時間戳記。這應該遵循 RFC3339 時間戳記格式。

下面的片段是一些有效Event對象的例子：

範例 1

```
{
    "source" : "source1",
    "detail" : {
        "key1" : [1,2,3,4],
        "key2" : "strval"
    },
    "detailType" : "sampleDetailType",
```

```
    "resources" : ["Resouce1", "Resource2"],
    "time" : "2022-01-10T05:00:10Z"
}
```

範例 2

```
{
    "source" : "source1",
    "detail" : {},
    "detailType" : "sampleDetailType"
}
```

範例 3

```
{
    "source" : "source1",
    "detail" : {
        "key1" : 1200
    },
    "detailType" : "sampleDetailType",
    "resources" : []
}
```

JavaScript 無資料來源的解析器函數參考

AWS AppSync 解析器函數請求和響應類型為 None 的數據源使您能夠塑造AWS AppSync 本地操作的請求。

要求

要求處理常式很簡單，可讓您透過payload欄位傳遞盡可能多的內容資訊。

```
type NONERequest = {
  payload: any;
};
```

以下是將字段參數傳遞給有效負載的示例：

```
export function request(ctx) {
  return {
    payload: context.args
```

```
  };
}
```

該payload字段的值將被轉發給函數響應處理程序，並在中提供context.result。

承載

該payload字段是一個容器，可用於傳遞任何數據，然後可供函數響應處理程序使用。

此 payload 欄位為選用。

回應

因為沒有資料來源，所以payload欄位的值會轉送至函數回應處理常式，並在context.result屬性上設定。

如果payload欄位值的形狀完全符合 GraphQL 類型的形狀，您可以使用下列回應處理常式轉寄回應：

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

沒有適用於退貨響應的必填字段或形狀限制。但是，由於 GraphQL 是強類型的，因此已解析的回應必須符合預期的 GraphQL 類型。

JavaScript HTTP 的解析器函數參考

AWS AppSync HTTP 解析器函數使您可以將請求從任何 HTTP 端點發送AWS AppSync 到，並將響應從 HTTP 端點發送回。AWS AppSync使用要求處理常式，您可以提供有AWS AppSync 關要叫用之作業性質的提示。本節說明支援的 HTTP 解析器的不同組態。

請求

```
type HTTPRequest = {
  method: 'PUT' | 'POST' | 'GET' | 'DELETE' | 'PATCH';
  params?: {
    query?: { [key: string]: any };
    headers?: { [key: string]: string };
    body?: any;
  };
  resourcePath: string;
```

```
};
```

下面的代碼片段是一個 HTTP POST 請求的一個例子，其中包含一個text/plain正文：

```
export function request(ctx) {
  return {
    method: 'POST',
    params: {
      headers: { 'Content-Type': 'text/plain' },
      body: 'this is an example of text body',
    },
    resourcePath: '/',
  };
}
```

方法

僅請求處理程序

AWS AppSync 傳送到 HTTP 端點的 HTTP 方法或動詞 (GET、POST、PUT、PATCH 或 DELETE)。

```
"method": "PUT"
```

ResourcePath

僅請求處理程序

您要存取的資源路徑。資源路徑與 HTTP 資料來源中的端點會組合成 AWS AppSync 服務發出請求的目標 URL。

```
"resourcePath": "/v1/users"
```

評估要求時，此路徑會做為 HTTP 要求的一部分傳送，包括 HTTP 端點。例如，之前的範例可轉譯如下：

```
PUT <endpoint>/v1/users
```

參數欄位

僅請求處理程序

用來指定搜尋執行的動作，最常見的方式是在主體內設定查詢值。不過，有多項其他功能可設定，例如回應的格式。

標頭

標頭資訊，以金鑰值對形式。金鑰與值皆必須為字串。

例如：

```
"headers" : {
    "Content-Type" : "application/json"
}
```

目前支援的 Content-Type 標頭包括：

```
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
```

您無法設定下列 HTTP 標頭：

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

金鑰值對，指定常見的選項，例如 JSON 回應的程式碼格式。金鑰與值皆必須為字串。以下範例顯示如何傳送 ?type=json 查詢字串：

```
"query" : {
    "type" : "json"
}
```

本文

此本文包含您選擇設定的 HTTP 請求本文。除非內容類型指定字元集，否則請求本文一律是 UTF-8 編碼的字串。

```
"body":"body string"
```

回應

請見此處範例。

JavaScript Amazon RDS 的解析器函數參考

AWS AppSync RDS 功能和解析器可讓開發人員使用 RDS 資料 API 將SQL查詢傳送至 Amazon Aurora 叢集資料庫，並取回這些查詢的結果。您可以使用的`rds`模組 `sql`-tagged 範本或使用模組 AWS AppSync的、、和`remove`輔助函式`select`，`insert`撰寫傳送至資料 API 的SQL陳述式。`rds` `update` AWS AppSync 利用 RDS 資料服務的`ExecuteStatement`動作，針對資料庫執行 SQL 陳述式。

主題

- SQL 標記的範本
- 建立陳述式
- 擷取資料
- 實用功能
- SQL 選取
- 插入
- SQL 更新
- SQL 刪除
- 轉換

SQL 標記的範本

AWS AppSync的`sql`標記範本可讓您建立靜態陳述式，以便透過使用範本運算式在執行階段接收動態值。 AWS AppSync 從運算式值建立變數對應，以建構傳送至 Amazon Aurora 無伺服器資料 API

的SqlParameterized查詢。使用此方法，在運行時傳遞的動態值不可能修改原始語句，這可能會導致意外執行。所有動態值都作為參數傳遞，不能修改原始語句，並且不會由數據庫執行。這會使您的查詢不易受到SQL注入攻擊的攻擊。

> **Note**
>
> 在任何情況下，在撰寫SQL聲明時，您都應遵循保安指引，妥善處理您接收作為輸入的資料。

> **Note**
>
> 加上sql標籤的範本僅支援傳遞變數值。您無法使用運算式來動態指定資料行或資料表名稱。不過，您可以使用公用程式函數來建置動態陳述式。

在下列範例中，我們建立一個查詢，該查詢會根據在 GraphQL 查詢中動態設定的col引數值進行篩選。該值只能使用標籤表達式添加到語句中：

```
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);
}
```

透過變數對應傳遞所有動態值，我們依靠資料庫引擎來安全地處理和清理值。

建立陳述式

函數和解析器可以與 MySQL 和 PostgreSQL 數據庫進行交互。使用createMySQLStatement和createPgStatement分別建立陳述式。例如，createMySQLStatement可以創建一個 MySQL 查詢。這些函數最多可接受兩個陳述式，當要求應立即擷取結果時很有用。使用 MySQL，你可以這樣做：

```
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並且createMySQLStatement不會逸出或引用使用sql標記範本建置的陳述式。

擷取資料

您執行的 SQL 陳述式的結果可在物件的回應處理常式中context.result取得。結果是一個 JSON 字符串，其中包含ExecuteStatement動作的響應元素。解析後，結果具有以下形狀：

```
type SQLStatementResults = {
    sqlStatementResults: {
        records: any[];
        columnMetadata: any[];
        numberOfRecordsUpdated: number;
        generatedFields?: any[]
    }[]
}
```

您可以使用此toJsonObject公用程式將結果轉換為代表傳回資料列的 JSON 物件清單。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]
```

```
}
```

請注意，toJsonObject返回一個數組的語句結果。如果您提供了一個語句，則數組長度為1。如果您提供了兩個語句，則數組長度為2。陣列中的每個結果都包含0或多個資料列。toJsonObjectnull如果結果值無效或意外，則返回。

實用功能

您可以使用 AWS AppSync RDS 模組的公用程式助手與資料庫互動。

SQL 選取

此select公用程式會建立SELECT陳述式來查詢您的關聯式資料庫。

基本使用

在其基本形式中，您可以指定要查詢的表格：

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // "SELECT * FROM "persons"
    return createPgStatement(select({table: 'persons'}));
}
```

請注意，您也可以在資料表識別碼中指定結構定義：

```
import { select, createPgStatement } from '@aws-appsync/utils/rds';

export function request(ctx) {

    // Generates statement:
    // SELECT * FROM "private"."persons"
    return createPgStatement(select({table: 'private.persons'}));
}
```

指定欄

您可以使用columns屬性指定欄。如果未將其設置為值，則默認為*：

```
export function request(ctx) {

    // Generates statement:
    // SELECT "id", "name"
    // FROM "persons"
    return createPgStatement(select({
        table: 'persons',
        columns: ['id', 'name']
    }));
}
```

您也可以指定資料欄的表格：

```
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

您可以套用`limit`和`offset`至查詢：

```
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
    }));
}
```

排序方式

您可以使用orderBy屬性對結果進行排序。提供指定列和可選dir屬性的對象的數組：

```
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'}]
    }));
}
```

篩選條件

您可以使用特殊條件物件來建置篩選器：

```
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'}}
    }));
}
```

您還可以組合過濾器：

```
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}}
    }));
}
```

您也可以建立OR陳述式：

```
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not用以下方式否定條件：

```
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 } }
        ]}
    }));
}
```

您也可以使用下列運算子來比較值：

運算子	描述	可能的值類型
eq	等於	數字，字符串，布爾
氖	不等於	數字，字符串，布爾
樂	小於或等於	數字，字符串
lt	小於	數字，字符串
GE	大於或等於	數字，字符串
gt	大於	數字，字符串
contains	喜歡	string
不包含	不喜歡	string
開始與	以前綴開始	string
介於	在兩個值之間	數字，字符串
屬性存在	該屬性不為空	數字，字符串，布爾
size	檢查元素的長度	string

插入

該`insert`實用程序提供了一種通過`INSERT`操作在數據庫中插入單行項目的直接方法。

單一項目插入

若要插入項目，請指定表格，然後傳入值的物件。物件索引鍵會對應至您的資料表資料行。列名稱會自動轉義，並使用變量映射將值發送到數據庫：

```
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 用案例

您可以將`insert`後跟 a 組合起`select`來檢索插入的行：

```
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)
}
```

郵政使用案例

使用 Postgres，您可以使[returning](#)用從插入的列中取得資料。它接受`*`或列名的數組：

```
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

該update實用程序允許您更新現有行。您可以使用 condition 物件，將變更套用至符合條件的所有資料列中的指定資料行。例如，假設我們有一個模式，允許我們進行這種突變。我們希望更新namePerson的id值，3但前提是我們已經知道他們（known_since）自一年以來2000：

```
mutation Update {
    updatePerson(
        input: {id: 3, name: "Jon"},
        condition: {known_since: {ge: "2000"}}
    ) {
    id
    name
  }
}
```

我們的更新解析器看起來像這樣：

```
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)
}
```

我們可以在條件中添加檢查，以確保僅更新主鍵id等3於的行。同樣，對於 Postgresinserts，您可以使用returning返回修改後的數據。

SQL 刪除

該remove實用程序允許您刪除現有行。您可以在滿足條件的所有行上使用條件對象。請注意，這delete是中的保留關鍵字 JavaScript。 remove應該用來代替：

```
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)
}
```

轉換

在某些情況下，您可能需要更多有關在語句中使用的正確對象類型的特殊性。您可以使用提供的類型提示來指定參數的類型。 AWS AppSync 支援與資料 API 相同的類型提示。您可以使用 AWS AppSync rds模塊中的typeHint函數來轉換參數。

下面的例子允許你發送一個數組作為一個值被轉換為一個 JSON 對象。我們使用->運算子來擷取 JSON 陣列index2中的元素：

```
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
}
```

在處理和比較時，鑄造也很有用 DATETIME，和TIMESTAMP：

```
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)
}
```

以下是另一個示例，顯示如何發送當前日期和時間：

```
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)}`)
}
```

可用的類型提示

- typeHint.DATE-對應的參數作為DATE類型的對象發送到數據庫。接受的格式為 YYYY-MM-DD。
- typeHint.DECIMAL-對應的參數作為DECIMAL類型的對象發送到數據庫。
- typeHint.JSON-對應的參數作為JSON類型的對象發送到數據庫。

- typeHint.TIME-對應的字串參數值會以TIME類型的物件形式傳送至資料庫。接受的格式為 HH:MM:SS[.FFF]。
- typeHint.TIMESTAMP-對應的字串參數值會以TIMESTAMP類型的物件形式傳送至資料庫。接受的格式為 YYYY-MM-DD HH:MM:SS[.FFF]。
- typeHint.UUID-對應的字串參數值會以UUID類型的物件形式傳送至資料庫。

解析器對映範本參考 (VTL)

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

以下幾節將說明如何在對應範本中使用公用程式作業。

主題

解析器映射模板概述

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync 可讓您透過對資源執行作業來回應 GraphQL 要求。對於您希望在其上執行查詢或變異的每個 GraphQL 欄位，都必須附加解析器，以便與資料來源通訊。通訊通常是透過資料來源唯一的參數或作業進行。

解析器是 GraphQL 和數據源之間的連接器。他們會說明AWS AppSync 如何將傳入的 GraphQL 要求轉譯成後端資料來源的指示，以及如何將來自該資料來源的回應轉換回 GraphQL 回應。它們是以 Apache 速度範本語言 (VTL) 編寫的，它會將您的請求作為輸入，並輸出包含解析程式指示的 JSON 文件。您可以使用對應範本來執行簡單指示，例如從 GraphQL 欄位傳入引數，或用於較複雜的指示，例如在將項目插入 DynamoDB 之前，循環瀏覽引數以建立項目。

有兩種類型的解析器，AWS AppSync 它們以略有不同的方式利用映射模板：

- 單位解析器
- 管道解析器

單位解析器

單位解析器是獨立的實體，其中只包括一個請求和響應模板。這種類型可以用於簡易、單一的操作，例如列出來自單一資料來源的清單項目。

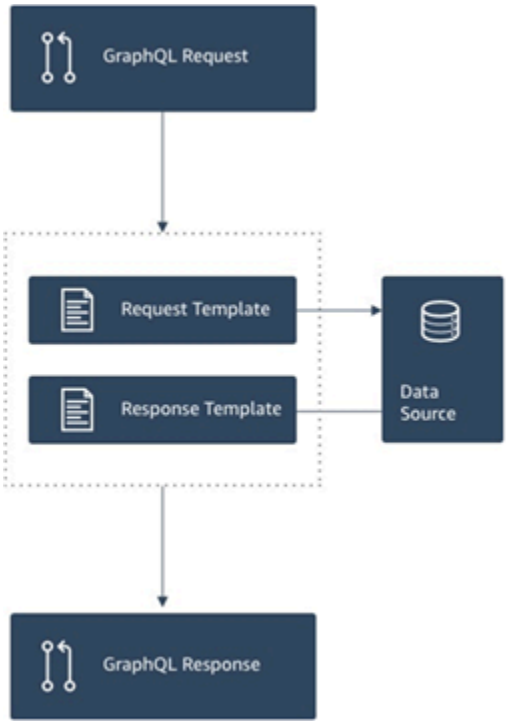
- 要求範本：剖析 GraphQL 作業後取得傳入的要求，並將其轉換為所選資料來源作業的要求組態。
- 回應範本：解譯來自資料來源的回應，並將其對應至 GraphQL 欄位輸出類型的形狀。

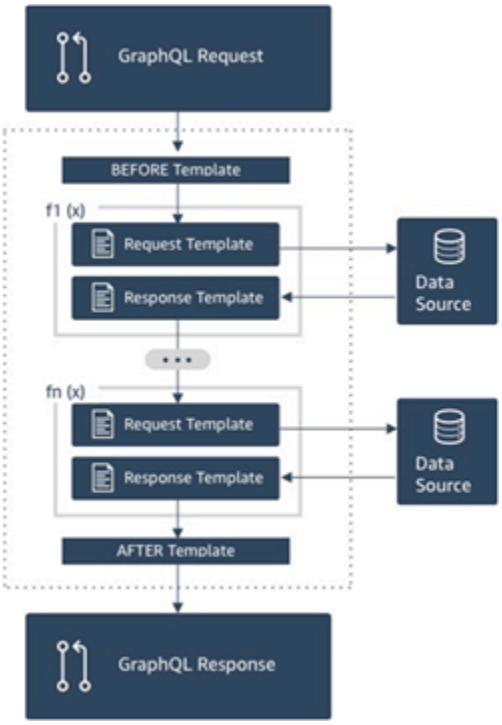
管道解析器

管線解析器包含按順序執行的一個或多個功能。每個函數都包括一個請求模板和響應模板。管道解析器也有一個之前的模板和一個後模板，圍繞模板包含的函數序列。之後範本會對應至 GraphQL 欄位輸出類型。管道解析器在響應模板映射輸出的方式中與單元解析器不同。管線解析器可以映射到所需的任何輸出，包括另一個函數的輸入或管線解析器的 after 模板。

管線解析器函數可讓您撰寫可在結構描述中跨多個解析器重複使用的通用邏輯。函數會直接附加至資料來源，與單位解析程式一樣，包含相同的請求與回應對映範本格式。

下圖說明左側單元解析器和右側配管解析器的流程。





管道解析器包含了單元解析器支持的功能的超集，以及更多功能，但代價要複雜一些。

管道解析程式剖析

配管解析器由「之前」對映範本、A fter 對映範本和函數清單組成。每個函數都有一個請求和響應映射模板，它對數據源執行。由於管道解析程式是將執行委派到一份函數清單，所以不會連結到任何資料來源。單位解析程式和函數是對資料來源執行操作的基本元素。如需詳細資訊，請參閱解析器對映範本概觀。

對映範本之前

管線解析器的請求對應範本或「之前」步驟可讓您在執行定義的函數之前執行一些準備邏輯。

函數清單

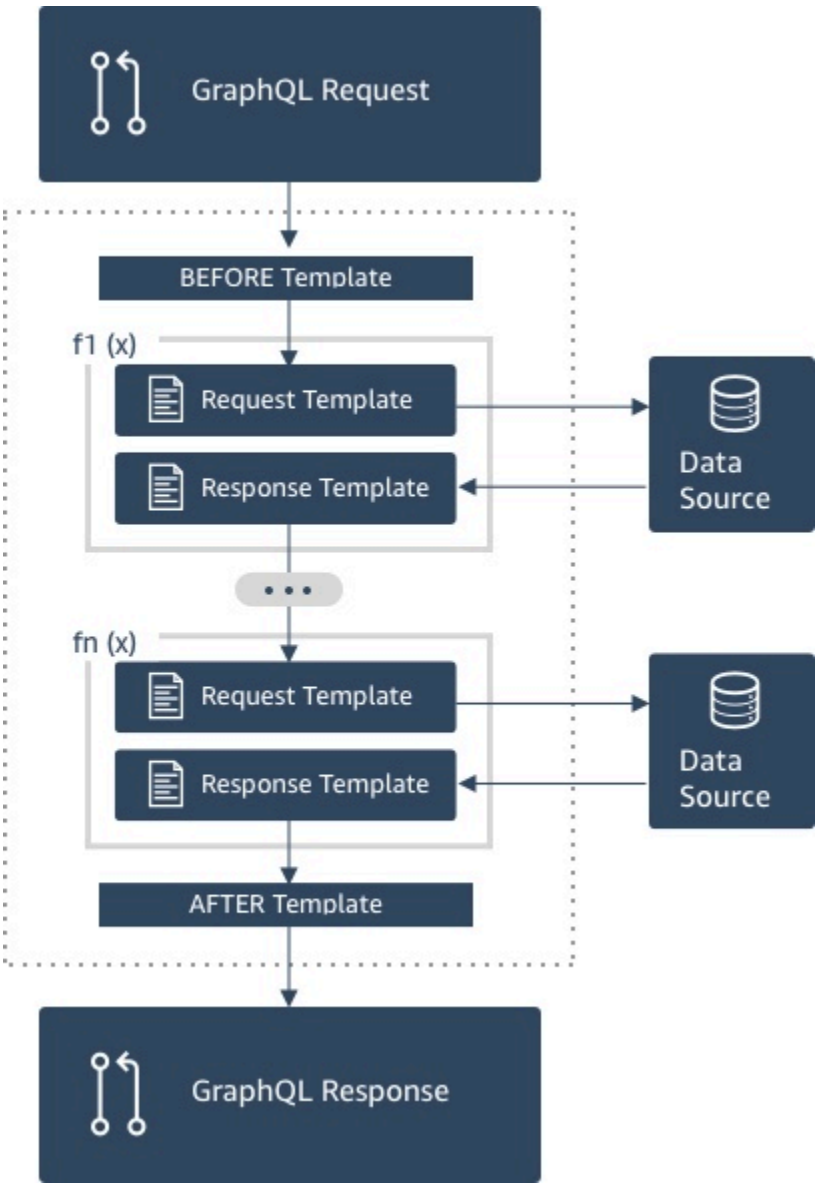
管道解析程式將會依序執行的函數清單。管道解析程式要求映射範本評估的結果，會依 `$ctx.prev.result` 提供給第一個函數。每個函數輸出都會依 `$ctx.prev.result` 提供給下一個函數。

After 映射範本

管線解析器的回應對應範本或 A fter 步驟可讓您執行從最後一個函式輸出到預期 GraphQL 欄位類型的一些最終對應邏輯。在函數清單的最後一個函數的輸出，將依 `$ctx.prev.result` 或 `$ctx.result` 提供給管道解析程式映射範本。

執行流程

給定由兩個函數組成的管線解析器，下面的列表代表調用解析器時的執行流程：



1. 配管解析器映射模板之前
2. 第 1 個函數：函數要求映射範本
3. 第 1 個函數：資料來源呼叫
4. 第 1 個函數：函數回應映射範本
5. 第 2 個函數：函數要求映射範本
6. 第 2 個函數：資料來源呼叫
7. 第 2 個函數：函數回應映射範本

8. 管道解析器映射模板後

> **Note**
>
> 管道解析程式執行流程為單向，並已在解析程式靜態定義。

有用的阿帕奇速度模板語言（VTL）實用程序

隨著應用程式複雜性提高，VTL 公用程式和此處的指示詞可加速開發生產力。下列公用程式可在您使用管道解析程式時提供協助。

$ctx.stash

存儲是在每個解析器和函數映射模板中提供的。Map在單一個解析程式執行期間，則會存在相同的 stash 執行個體。這表示您可以使用 stash 在管道解析程式中的所有要求和回應映射範本、以及全部函數中，傳遞任意資料。存儲公開與 Java 映射數據結構相同的方法。

$ctx.prev.result

`$ctx.prev.result`表示先前在管線解析器中執行的作業結果。

如果先前的作業是管線解析程式的「之前」對映範本，則`$ctx.prev.result`代表範本評估的輸出，並可供管線中的第一個函數使用。如果先前操作是第一個函數，則 `$ctx.prev.result` 會顯示第一個函數的輸出，並將資料提供給管道中的第二個函數。如果上一個操作是最後一個函數，則`$ctx.prev.result`代表最後一個函數的輸出，並可供管線解析器的 After 映射模板使用。

#return(data: Object)

當您需要從任何映射範本提前傳回時，這時使用 `#return(data: Object)` 指令就能完成。`#return(data: Object)` 類似於程式設計語言中的 return 關鍵字，因為它會從最靠近範圍的邏輯區塊傳回。這表示在解析程式映射範本中使用 `#return` 時，將從該解析程式傳回。在解析程式映射範本中使用 `#return(data: Object)`，將會設定 GraphQL 欄位上的 `data`。此外，使用函數映射範本的 `#return(data: Object)` 時會從該函數傳回，且執行將持續到該管道或是解析程式回應映射範本中的下一個函數。

#return

這與相同`#return(data: Object)`，但`null`將返回。

$util.error

`$util.error` 公用程式非常適合用來擲出欄位錯誤。在函數映射範本內使用 `$util.error` 會立即擲出錯誤，其可阻止後續函數執行。有關更多詳細信息和其他`$util.error`簽名，請訪問解析器映射模板實用程序參考。

$util.appendError

`$util.appendError` 的功能類似於 `$util.error()`，主要的差別在於前者不會中斷映射範本的評估，而是在欄位出現錯誤時發出訊號，但允許範本的評估，進而將資料傳回。在函數中使用 `$util.appendError` 並不會中斷管道的執行流程。有關更多詳細信息和其他`$util.error`簽名，請訪問解析器映射模板實用程序參考。

範例 範本

假設您在名為的欄位上有一個 DynamoDB 資料來源和一個單位解析器，`getPost(id:ID!)`該欄位會傳回具有下列 GraphQL 查詢的`Post`類型：

```
getPost(id:1){
    id
    title
    content
}
```

解析程式範本看起來可能會類似於下列的範例：

```
     {
    "version" : "2018-05-29",
    "operation" : "GetItem",
    "key" : {
        "id" : $util.dynamodb.toDynamoDBJson($ctx.args.id)
    }
}
```

這會用 `id` 輸入的參數值 1 來取代 `${ctx.args.id}`，並產生下列 JSON：

```
     {
    "version" : "2018-05-29",
    "operation" : "GetItem",
    "key" : {
        "id" : { "S" : "1" }
```

```
    }
}
```

AWS AppSync 使用此範本產生與 DynamoDB 通訊和取得資料 (或視需要執行其他作業) 的指示。在資料傳回之後，AWS AppSync 會透過選用的回應映射範本來處理資料 (您可以使用此範本來進行資料管理或執行邏輯)。例如，當我們從 DynamoDB 取回結果時，結果可能如下所示：

```
{
        "id" : 1,
        "theTitle" : "AWS AppSync works offline!",
        "theContent-part1" : "It also has realtime functionality",
        "theContent-part2" : "using GraphQL"
}
```

您可以利用下列的回應映射範本，來選擇將其中兩個欄位合併為單一欄位：

```
{
        "id" : $util.toJson($context.data.id),
        "title" : $util.toJson($context.data.theTitle),
        "content" : $util.toJson("${context.data.theContent-part1}
 ${context.data.theContent-part2}")
}
```

在將範本套用到資料之後，資料的格式如下所示：

```
{
        "id" : 1,
        "title" : "AWS AppSync works offline!",
        "content" : "It also has realtime functionality using GraphQL"
}
```

這些資料會再做為回應傳回給用戶端，如下所示：

```
{
        "data": {
                "getPost":      {
                        "id" : 1,
                        "title" : "AWS AppSync works offline!",
                        "content" : "It also has realtime functionality using GraphQL"
                }
        }
```

```
}
```

請注意，在大部分情況下，回應映射範本是簡單的資料傳遞，主要差別在於您是傳回個別項目或項目清單。若是個別項目，則傳遞是：

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

若是清單，則傳遞通常是：

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

要查看單元和管道解析器的更多示例，請參閱解析器教程。

評估的對映範本還原序列化規則

對於一個字串的對應範本評估。在 AWS AppSync 中，輸出字串必須遵循 JSON 結構才有效。

此外，系統會強制執行下列還原序列化規則。

JSON 物件中不允許重複金鑰

如果已評估的對應範本字串代表 JSON 物件，或包含具有重複金鑰的物件，則對應範本會傳回下列錯誤訊息：

```
Duplicate field 'aField' detected on Object. Duplicate JSON keys are not allowed.
```

已評估要求對應範本中重複金鑰的範例：

```
{
    "version": "2018-05-29",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "1",
        "field": "getPost" ## key 'field' has been redefined
    }
}
```

若要修正此錯誤，請勿在 JSON 物件中重新定義金鑰。

JSON 物件中不允許結尾字元

如果已評估的對應範本字串代表 JSON 物件，且包含額外的結尾字元，則對應範本會傳回下列錯誤訊息：

```
Trailing characters at the end of the JSON string are not allowed.
```

已評估請求映射範本中結尾字元的範例：

```
{
    "version": "2018-05-29",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": "1",
    }
}extraneouschars
```

若要修正此錯誤，請確定評估的範本嚴格評估為 JSON。

解析器映射模板編程指南

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

這是一個在 AWS AppSync 中使用 Apache Velocity 範本語言 (VTL) 進行程式設計的說明書樣式教學。如果您熟悉其他編程語言（例如 JavaScript C 或 Java），則應該非常簡單。

AWS AppSync 使用 VTL 將來自用戶端的 GraphQL 要求轉換為資料來源的要求。然後，它會反轉程序，來將資料來源轉換回 GraphQL 回應。VTL 是一種邏輯模板語言，它使您能夠使用以下技術在 Web 應用程序的標準請求/響應流中操作請求和響應：

- 新項目的預設值
- 輸入驗證和格式化
- 轉換與打造資料
- 逐一查看清單、映射和陣列以移出或更改值

- 根據使用者身分篩選條件/變更回應
- 複雜的授權檢查

例如，您可能想要在 GraphQL 引數上在服務中執行電話號碼驗證，或將輸入參數轉換為大寫，然後再將輸入參數儲存在 DynamoDB 中。或者，您希望用戶端系統能以 GraphQL 引數、JWT 字符或 HTTP 標頭的形式提供程式碼，並在程式碼符合清單中的特定字串時才回應資料。這些都是您可以在AWS AppSync中使用 VTL 執行的邏輯檢查。

VTL 可讓您使用熟悉的程式設計技巧，來套用邏輯。不過，其受到限制，必須在標準請求/回應流程內執行，以確保 GraphQL API 可隨著您的使用者成長而擴展。由於AWS AppSync 還支持AWS Lambda 作為解析器，因此如果您需要更多靈活性，可以使用您選擇的編程語言（Node.js，Python，Go，Java 等）編寫 Lambda 函數。

設定

學習語言時的一種常見技巧是打印出結果（例如，`console.log(variable)`in JavaScript）以查看會發生什麼。在本教學中，我們透過建立簡單的 GraphQL 結構描述並將對應值傳遞到 Lambda 函式來示範此技巧。Lambda 函式會列印出值，然後加以回應。這可讓您了解請求/回應流程，並查看不同的程式設計技巧。

開始建立以下 GraphQL 結構描述：

```
type Query {
    get(id: ID, meta: String): Thing
}

type Thing {
    id: ID!
    title: String!
    meta: String
}

schema {
    query: Query
}
```

現在，使用 Node.js 做為語言來建立以下 AWS Lambda 函式：

```
exports.handler = (event, context, callback) => {
```

```
    console.log('VTL details: ', event);
    callback(null, event);
};
```

在 主控台的 Data SourcesAWS AppSync (資料來源) 窗格中，新增此 Lambda 函式做為新的資料來源。瀏覽回主控台的 [結構描述] 頁面，然後按一下`get(...):Thing`查詢旁邊右側的 [ATT ACH] 按鈕。如需請求範本，從 Invoke and forward arguments (叫用和轉送引數) 功能表中選擇現有的範本。如需回應範本，請選擇 Return Lambda result (傳回 Lambda 結果)。

在一個位置為您的 Lambda 函數開啟 Amazon CloudWatch 日誌，然後從AWS AppSync 主控台的 [查詢] 索引標籤執行下列 GraphQL 查詢：

```
query test {
  get(id:123 meta:"testing"){
    id
    meta
  }
}
```

GraphQL 回應應包含 `id:123` 和 `meta:testing`，因為 Lambda 函式會參考這兩者。幾秒鐘後，您應該會在記錄 CloudWatch 檔中看到包含這些詳細資料的記錄。

Variables

VTL 使用參考，供您用來存放或操作資料。在 VTL 中有三種類型的參考：變數、屬性和方法。變數的前面會有 $ 符號且這些變數的建立是透過 `#set` 指令：

```
#set($var = "a string")
```

變數會使用其他語言存放您所熟悉的類似類型，例如數字、字串、陣列、清單和映射。您可能已經注意到傳送至預設請求範本以供 Lambda 解析程式使用的 JSON 承載：

```
"payload": $util.toJson($context.arguments)
```

在此有幾個需注意的事項；首先，AWS AppSync 提供多種便利性功能，適用於常見的操作。在這個範例中，`$util.toJson` 會將變數轉換為 JSON。其次，會自動從 GraphQL 請求填入變數 `$context.arguments` 做為映射物件。您可以建立新的映射，如下所示：

```
#set( $myMap = {
```

```
  "id": $context.arguments.id,
  "meta": "stuff",
  "upperMeta" : $context.arguments.meta.toUpperCase()
} )
```

現在，您可以建立名為 $myMap 的變數，其擁有 id、meta 以及 upperMeta 的金鑰。這也展現了以下幾點：

- 從 GraphQL 引數將金鑰填入 id。在 VTL 中從用戶端擷取引數是很常見的做法。
- 是以值來將 meta 硬式編碼，以展現預設值。
- upperMeta 會使用方法 meta 來轉換 .toUpperCase() 引數。

將之前的程式碼放在請求範本的頂部並變更 payload 以使用新的 $myMap 變數：

```
"payload": $util.toJson($myMap)
```

執行 Lambda 函數，您可以在 CloudWatch 記錄中查看回應變更以及此資料。在您逐步完成本教學的其餘部分，我們將保持填入 $myMap 如此您就可以執行類似的測試。

您也可以對變數設定 properties_。它們可以是簡單的字串、陣列或 JSON：

```
#set($myMap.myProperty = "ABC")
#set($myMap.arrProperty = ["Write", "Some", "GraphQL"])
#set($myMap.jsonProperty = {
    "AppSync" : "Offline and Realtime",
    "Cognito" : "AuthN and AuthZ"
})
```

安靜參考

由於 VTL 是一種範本化語言，在預設情況下，您給予的每個參考將會執行 .toString()。如果參考是未定義的，它會列印實際參考呈現做為字串。例如：

```
#set($myValue = 5)
##Prints '5'
$myValue

##Prints '$somethingelse'
$somethingelse
```

為了解決這個問題，VTL 有安靜參考或無提示參考語法，可告訴範本引擎抑制這種行為。此語法是 $!{}。例如，如果我們稍微變更之前的程式碼以使用 $!{somethingelse}，系統會抑制列印：

```
#set($myValue = 5)
##Prints '5'
$myValue

##Nothing prints out
$!{somethingelse}
```

呼叫方法

在之前的範例中，我們示範了如何建立變數並同時設定值。您也可以如下所示將資料新增到映射，以兩個步驟執行此操作：

```
#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")}
```

然而，有一些需了解這種行為的原因。雖然安靜參考表示法可讓您 $!{} 如以上方式呼叫方法，它將不會抑制執行方法的傳回值。這也是為什麼我們會說明以上的 ##Prints "first value" 及 ##Prints true。您逐一查看映射或清單可能會導致錯誤，例如插入索引鍵已存在的值，因為輸出會在評估時將未預期的字串加入範本中。

此問題的解決方法有時會使用 #set 指令來呼叫方法並忽略變數。例如：

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

您可以在模板中使用此技術，因為它可以防止在模板中打印意外的字符串。AWS AppSync 提供了一個替代的便利功能，以更簡潔的表示法提供相同的行為。這可讓您不需要考慮這些實作特性。您可以透過 $util.quiet() 或其別名 $util.qr() 存取此函式。例如：

```
#set ($myMap = {})
```

```
#set ($myList = [])

##Nothing prints out
$util.quiet($myMap.put("id", "first value"))
##Nothing prints out
$util.qr($myList.add("something"))
```

Strings

隨著使用多種程式設計語言，字串可能會變得難以處理，尤其是當您想要透過變數建構字串時。有一些與 VTL 相關的常見事件。

假設您要以字串形式將資料插入 DynamoDB 等資料來源，但是會從變數填入，例如 GraphQL 引數。一個字符串將有雙引號，並且只需要引用字符串中的變量"${}"（所以不要！像安靜的參考符號那樣）。這與以下內容中的模板文字類似 JavaScript：https://developer.mozilla.org/en-US/docs/Web/JavaScript /參考文字/ 模板文字

```
#set($firstname = "Jeff")
$!{myMap.put("Firstname", "${firstname}")}
```

您可以在 DynamoDB 要求範本中看到這一點，例如使用 GraphQL 用戶端的引數`"author": { "S" : "${context.arguments.author}"}`時，或是用於自動識別碼產生類似的。`"id" : { "S" : "$util.autoId()"}`這表示您可以參考變數或字串內部的方法結果來填入資料。

您也可以使用 Java 字串類別的公有方法 (例如取出子字串)：

```
#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}"))
```

字串連接也是非常常見的任務。您可以單獨使用變數參考或使用變數參考搭配靜態值來這樣做：

```
#set($s1 = "Hello")
#set($s2 = " World")

$util.qr($myMap.put("concat","$s1$s2"))
$util.qr($myMap.put("concat2","Second $s1 World"))
```

迴圈

現在您已建立變數和呼叫方法，您可以將一些邏輯到新增到程式碼。與其他語言不同，VTL 只允許迴圈，其中反覆次數是預先決定的。在速度中沒有 do..while。此設計可確保評估處理一律終止，並提供 GraphQL 操作執行時的擴充界限。

迴圈是使用 #foreach 所建立並需要您輸入迴圈變數和 iterable 物件 (例如陣列、清單、映射或集合)。#foreach 迴圈的典型程式設計範例，即是循環執行集合中的項目並將他們印出，因此我們的案例中，我們將這些項目移出，並將他們新增至映射：

```
#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
```

此範例顯示一些考量。第一種是使用變數與範圍 [..] 運算子來建立 iterable 物件。然後，您可以操作的變數 $i 會參考每個項目。在上述範例中，您還可以查看註解，註解會加上雙井字號 ## 來表示。這也展示在金鑰或值兩者中使用迴圈變數，以及使用字串的不同串連方法。

請注意，$i 是一個整數，因此您可以呼叫 .toString() 方法。若是 INT 的 GraphQL 類型，此技巧非常實用。

您也可以直接使用各種操作，例如：

```
#foreach($item in [1..5])
    ...
#end
```

陣列

您到目前為止已完成映射的操控，但在 VTL 中陣列也相當常見。您也可以使用陣列存取一些基本方法，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)))
```

前面的例子使用數組索引表示法來檢索具有的元素`arr2[$idx]`。您可以透過類似的方式從 Map/字典中來查詢名稱：

```
#set($result = {
    "Author" : "Nadia",
    "Topic" : "GraphQL"
})

$util.qr($myMap.put("Author", $result["Author"]))
```

在使用條件時，在回應範本中篩選來自資料來源的結果是很常見的。

條件式檢查

`#foreach` 的舊區段說明使用邏輯來透過 VTL 轉換資料的部分範例。您也可以套用條件檢查，在執行階段評估資料：

```
#if(!$array.isEmpty())
    $util.qr($myMap.put("ifCheck", "Array not empty"))
#else
    $util.qr($myMap.put("ifCheck", "Your array is empty"))
#end
```

上述布林值表達式 `#if()` 的檢查是精細的，但您也可以使用運算子和 `#elseif()` 來進行分支：

```
#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
```

這兩個範例示範 negation(!) 和 equality (==)。我們也可以使用 ||、&&、>、<、>=、<= 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：在條件中只有 `Boolean.FALSE` 和 `null` 會被視為 false。零 (0) 和空白字串 ("") 不等同於 false。

電信業者

程式設計語言需要運算子來執行數學動作才得以完整。有幾種簡單的入門方式：

```
#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))
```

迴圈和條件

這在 VTL 中轉換資料很常見 (例如在從資料來源寫入或讀取前)，以循環執行物件，然後執行檢查，再執行動作。將之前區段的一些工具進行整合，讓您有更多功能可使用。一個便利的工具是知道 #foreach 會自動提供您每個項目的 .count：

```
#foreach ($item in $arr2)
  #set($idx = "item" + $foreach.count)
  $util.qr($myMap.put($idx, $item))
#end
```

例如，如果值小於特定大小，則也許您會希望將其從映射移出。使用計數以及條件和 #break 陳述式可讓您執行此操作：

```
#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
```

先前 #foreach 已透過 .keySet() (您可以在映射上使用此項目) 來重複使用。這讓您能夠取得 $key 並使用 .get($key) 來參考該值。從用戶端在 AWS AppSync 中的 GraphQL 引數會存放為映射。也可以透過 .entrySet() (您可以同時存取金鑰和值做為 Set) 來重複使用這些項目，並填入其他變數或執行複雜條件式檢查 (例如驗證或輸入的轉換)：

```
#foreach( $entry in $context.arguments.entrySet() )
#if ($entry.key == "XYZ" && $entry.value == "BAD")
    #set($myvar = "...")
  #else
    #break
  #end
#end
```

其他常見的範例會自動填入預設資訊，例如同步處理資料時的初始物件版本 (在衝突解決中非常重要) 或物件的預設擁有者進行授權檢查-Mary 建立了這篇部落格文章，因此：

```
#set($myMap.owner ="Mary")
#set($myMap.defaultOwners = ["Admins", "Editors"])
```

Context

現在您更熟悉使用 VTL 在AWS AppSync解析器中執行邏輯檢查，請查看前後關聯物件：

```
$util.qr($myMap.put("context", $context))
```

這包含所有資訊，您可以在 GraphQL 請求中存取這些資訊。有關詳細說明，請參閱內容參考。

篩選

此教學到目前為止，會透過非常簡單的 JSON 轉換，將 Lambda 函式的所有資訊傳回到 GraphQL 查詢：

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

VTL 邏輯就跟您從資料來源取得回應時一樣強大，尤其是在對回應進行授權檢查時。讓我們逐步介紹一些範例。首先，嘗試如下變更回應範本：

```
#set($data = {
    "id" : "456",
    "meta" : "Valid Response"
})

$util.toJson($data)
```

無論您對 GraphQL 操作進行何種操作，系統會將硬式編碼值傳回用戶端。稍微變更此項目，以從 Lambda 回應填入 `meta` 欄位，如需了解條件時則在教學課程中以 `elseIfCheck` 值來較早設立此值：

```
#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 是一種映射，因此您可以使用 entrySet() 來對金鑰或傳回的值執行邏輯。由於 $context.identity 包含使用者執行 GraphQL 操作的相關資訊，如果您從資料來源傳回授權資訊，則可以根據邏輯決定是否向使用傳回全部、部分資料或不傳回資料。將您的回應範本變更為如下所示：

```
#if($context.result["id"] == 123)
    $util.toJson($context.result)
  #else
    $util.unauthorized()
#end
```

如果您執行 GraphQL 查詢，系統會以一般的方式傳回。不過，如果您將 id 引數變更 123 以外的值 (query test { get(id:456 meta:"badrequest"){} })，您會收到授權失敗的訊息。

您可以在授權使用案例小節中找到更多授權案例的範例。

附錄 - 範本範例

若您遵循此教學至今，您可以逐步建立此範本。若您並未遵循此教學，我們已在下方提供，可讓您複製並進行測試。

請求範本

```
#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)
}
```

回應範本

```
#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
```

解析器對映範本前後關聯參考

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync定義了一組用於使用解析器映射模板的變量和函數。這樣可讓使用 GraphQL 對資料進行邏輯運算更簡單。本文件說明這些函式，並提供範本使用範例。

正在存取 `$context`

`$context` 變數是一種映射，保存您的解析程式叫用的所有情境資訊。其結構如下：

```
{
   "arguments" : { ... },
   "source" : { ... },
   "result" : { ... },
   "identity" : { ... },
   "request" : { ... },
   "info": { ... }
}
```

> **Note**
>
> 如果您嘗試透過其索引鍵存取字典/對映項目 (例如中的項目context) 以擷取值，Velocity 範本語言 (VTL) 可讓您直接使用符號。<dictionary-element>.<key-name>不過，這可能不是所有案例皆適用，例如當索引鍵名稱有特殊字元 (例如，底線「_」)。建議您一律使用 <dictionary-element>.get("<key-name>") 表示法。

$context 映射中每個欄位的定義如下：

$context 欄位

arguments

包含此欄位所有 GraphQL 引數的映射。

identity

包含有關發起人資訊的物件。如需此欄位結構的詳細資訊，請參閱身分。

source

包含父欄位解析度的映射。

stash

此 stash 是每個解析程式和函數映射範本中都會提供的「映射」。在單一個解析程式執行期間，則會存在相同的 stash 執行個體。這表示您可以使用 stash 在管道解析程式中的所有要求和回應映射範本、以及全部函數中，傳遞任意資料。此 stash 會公開與 Java 映射資料結構相同的方法。

result

此解析程式結果的容器。此欄位僅適用於回應對應範本。

例如，如果您要解析下列查詢的author欄位：

```
query {
    getPost(id: 1234) {
        postId
        title
        content
        author {
```

```
            id
            name
        }
    }
}
```

那麼，在處理回應映射範本時，可用的完整 $context 變數可能是：

```
{
  "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

在管道解析器中執行任何先前操作的結果。

如果先前的作業是管線解析程式的「之前」對映範本，則$ctx.prev.result代表範本評估的輸出，並可供管線中的第一個函數使用。

如果先前操作是第一個函數，則 $ctx.prev.result 會顯示第一個函數的輸出，並將資料提供給管道中的第二個函數。

如果上一個操作是最後一個函數，則$ctx.prev.result代表最後一個函數的輸出，並可供管線解析器的 After 映射模板使用。

info

包含有關 GraphQL 請求資訊的物件。如需此欄位的結構，請參閱資訊。

Identity

包含有關發起人資訊的 identity 區段。此區段的形態取決於您 AWS AppSync API 的授權類型。

如需有關AWS AppSync 安全性選項的詳細資訊，請參閱授權和驗證。

API_KEY 授權

未填入identity欄位。

AWS_LAMBDA 授權

identity包含索resolverContext引鍵，其中包含授權請求的 Lambda 函數傳回的相同resolverContext內容。

AWS_IAM 授權

具identity有以下形式：

```
{
    "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 授權

具identity有以下形式：

```
{
    "sub" : "uuid",
    "issuer" : "string",
    "username" : "string"
```

```
    "claims" : { ... },
    "sourceIp" : ["x.x.x.x"],
    "defaultAuthStrategy" : "string"
}
```

每個欄位的定義如下：

accountId

來電者的AWS帳戶 ID。

claims

使用者擁有的宣告。

cognitoIdentityAuthType

經身分驗證或未經身分驗證 (根據身分類型)。

cognitoIdentityAuthProvider

以逗號分隔的外部身分識別提供者資訊清單，用來取得用來簽署要求的認證。

cognitoIdentityId

來電者的 Amazon Cognito 身份識別碼。

cognitoIdentityPoolId

與呼叫者相關聯的 Amazon Cognito 身分集區識別碼。

defaultAuthStrategy

此發起人 (ALLOW 或 DENY) 的預設授權策略。

issuer

字符發行者。

sourceIp

AWS AppSync接收來電者的來源 IP 位址。如果要求不包含標`x-forwarded-for`頭，則來源 IP 值僅包含來自 TCP 連線的單一 IP 位址。如果要求包含 `x-forwarded-for` 標頭，則來源 IP 除了有 TCP 連線的 IP 地址外，也將有 `x-forwarded-for` 標頭中 IP 地址的清單。

sub

已驗證使用者的 UUID。

user

IAM 使用者。

userArn

IAM 使用者的亞馬遜資源名稱 (ARN)。

username

已驗證使用者的使用者名稱。如果是 AMAZON_COGNITO_USER_POOLS 授權，使用者名稱的值是屬性 cognito:username 的值。在AWS_IAM授權的情況下，使用者名稱的值是AWS使用者主體的值。如果您使用 IAM 授權搭配從 Amazon Cognito 身分識別集區提供的登入資料，建議您使用。cognitoIdentityId

存取要求標頭

AWS AppSync 支持從客戶端傳遞自定義標題 GraphQL 並通過使用. $context.request.headers 然後，您可以將標頭值用於動作，例如將資料插入資料來源或授權檢查。您可以使用命令列中的 API 金鑰$curl搭配使用單一或多個要求標頭，如下列範例所示：

單頭的例子

假設您設定使用 custom 值的 nadia 標頭，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
```

可使用 $context.request.headers.custom 對此進行存取。例如，它可能位於 DynamoDB 的下列 VTL 中：

```
"custom": $util.dynamodb.toDynamoDBJson($context.request.headers.custom)
```

多個頭的例子

您也可以將多個標頭傳遞到單一要求，並在解析程式映射範本中存取這些要求。例如，如果標custom頭設定有兩個值：

```
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
```

然後，您可以將這些存取為陣列，例如 `$context.request.headers.custom[1]`。

> **Note**
>
> AWS AppSync不會公開中的餅乾標題`$context.request.headers`。

存取要求自訂網域名稱

AWS AppSync支援設定可用來存取您的 GraphQL 和 API 的即時端點的自訂網域。使用自訂網域名稱提出要求時，您可以使用來取得網域名稱`$context.request.domainName`。

使用預設 GraphQL 端點網域名稱時，值為`null`。

Info

`info` 區段包含有關 GraphQL 請求的資訊。本節具有以下形式：

```
{
    "fieldName": "string",
    "parentTypeName": "string",
    "variables": { ... },
    "selectionSetList": ["string"],
    "selectionSetGraphQL": "string"
}
```

每個欄位的定義如下：

`fieldName`

目前正在解析的欄位名稱。

`parentTypeName`

目前正在解析的欄位父類型名稱。

`variables`

包含傳遞給 GraphQL 請求之所有變數的映射。

selectionSetList

此清單表示 GraphQL 選取範圍中的欄位。別名的欄位僅由別名參考，而不是欄位名稱參考。以下範例詳細說明這一點。

selectionSetGraphQL

此字串表示格式為 GraphQL 結構描述定義語言 (SDL) 的選取範圍。雖然片段不會合併到選集中，但會保留內嵌片段，如下列範例所示。

> **Note**
>
> 使用 `$utils.toJson()` on 時`context.info`，默認情況下不會序列化`selectionSetGraphQL`和`selectionSetList`返回的值。

例如，如果您解析下列查詢的 `getPost` 欄位：

```
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
  }
}
```

那麼，在處理映射範本時，可用的完整 $context.info 變數可能是：

```
{
  "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僅公開屬於目前類型的欄位。如果目前類型是介面或聯集，則只會顯示屬於該介面的選取欄位。例如，給定以下模式：

```
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
}
```

以下查詢：

```
query {
    node(id: "post1") {
        id
        ... on Post {
            title
        }

        ... on Blog {
            title
        }
    }
}
```

$ctx.info.selectionSetList在調用Query.node字段分辨率時，id僅暴露：

```
"selectionSetList": [
    "id"
]
```

處理輸入

應用程式必須處理不受信任的輸入，以防止任何外部方使用其預期用途之外的應用程式。由於$context包含屬性（例如$context.arguments、$context.identity、$context.result和）中

的用戶輸入$context.request.headers，因此必須小心在映射模板中清理其值。$context.info.variables

由於對應範本代表 JSON，輸入處理採用從表示使用者輸入的字串逸出 JSON 保留字元的形式。將敏感字串值放入對應範本時，最佳做法是使用 $util.toJson() 公用程式將 JSON 保留字元從敏感字串值逸出。

例如，在下列 Lambda 要求對應範本中，因為我們存取了不安全的客戶輸入字串 ($context.arguments.id)，所以我們將其包裝起$util.toJson()來，以防止未逸出的 JSON 字元破壞 JSON 範本。

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "postId": $util.toJson($context.arguments.id)
    }
}
```

與下面的映射模板相反，我們直接插入而$context.arguments.id無需消毒。這不適用於包含未轉義引號或其他 JSON 保留字符的字符串，並且可能導致模板打開失敗。

```
## 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.
    }
}
```

解析程式對映範本公用程式參考

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync定義了一組公用程式，您可以在 GraphQL 解析器中使用這些公用程式，以簡化與資料來源的互動。其中一些公用程式適用於任何資料來源，例如產生 ID 或時間戳記。其他專用於某種類型的資料來源。

主題

在 $ 實用助手

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

此`$util`變數包含可協助您處理資料的一般公用程式方法。除非另行指定，否則所有公用程式皆使用 UTF-8 字元集。

解析工具

JSON 解析實用程序列表

```
$util.parseJson(String) : Object
```

採用「字串化」JSON，並傳回結果的物件呈現。

```
$util.toJson(Object) : String
```

採用物件，並傳回該物件的「字串化」JSON 呈現。

編碼公用程式

編碼實用程序列表

```
$util.urlEncode(String) : String
```

以 application/x-www-form-urlencoded 編碼字串的形式傳回輸入字串。

```
$util.urlDecode(String) : String
```

將 application/x-www-form-urlencoded 編碼的字串解碼回非編碼格式。

```
$util.base64Encode( byte[] ) : String
```

將輸入編碼為 base64 編碼字串。

```
$util.base64Decode(String) : byte[]
```

解碼 base64 編碼字串中的資料。

ID 產生公用程式

ID 生成實用程序列表

```
$util.autoId() : String
```

傳回 128 位元隨機產生的 UUID。

```
$util.autoUlid() : String
```

返回一個 128 位隨機生成的 ULID（通用唯一的字典排序標識符）。

```
$util.autoKsuid() : String
```

返回 128 位隨機生成的 KSUID（K 可排序的唯一標識符）基於 62 編碼為長度為 27 的字符串。

錯誤實用程序

錯誤實用程序列表

`$util.error(String)`

擲回自訂錯誤。在請求或響應映射模板中使用此選項，以檢測請求或調用結果的錯誤。

`$util.error(String, String)`

擲回自訂錯誤。在請求或響應映射模板中使用此選項，以檢測請求或調用結果的錯誤。您也可以指定`errorType`.

`$util.error(String, String, Object)`

擲回自訂錯誤。在請求或響應映射模板中使用此選項，以檢測請求或調用結果的錯誤。您也可以指定`errorType`和`data`欄位。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

> **Note**
>
> `data`將根據查詢選集進行篩選。

`$util.error(String, String, Object, Object)`

擲回自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。此外`errorType`，還可以指定`data`欄位、`errorInfo`欄位和欄位。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

> **Note**
>
> `data`將根據查詢選集進行篩選。`errorInfo` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。
> `errorInfo`將不會根據查詢選集進行篩選。

`$util.appendError(String)`

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。與 `$util.error(String)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。

`$util.appendError(String, String)`

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。您同時可以指定 `errorType`。與 `$util.error(String, String)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。

`$util.appendError(String, String, Object)`

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。您同時可以指定 `errorType` 和 `data` 欄位。與 `$util.error(String, String, Object)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

> **Note**
>
> `data`將根據查詢選集進行篩選。

`$util.appendError(String, String, Object, Object)`

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。此外`errorType`，還可以指定`data`欄位、`errorInfo`欄位和欄位。與 `$util.error(String, String, Object, Object)` 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。`data` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。

> **Note**
>
> `data`將根據查詢選集進行篩選。`errorInfo` 值將新增到 GraphQL 回應中，`error` 內對應的 `errors` 區塊。
> `errorInfo`將不會根據查詢選集進行篩選。

條件驗證實用程序

條件驗證實用程序列表

`$util.validate(Boolean, String) : void`

如果條件為 false，則拋出 CustomTemplateException 帶有指定消息的一個。

```
$util.validate(Boolean, String, String) : void
```

如果條件為 false，則拋出 CustomTemplateException 具有指定消息和錯誤類型的一個。

```
$util.validate(Boolean, String, String, Object) : void
```

如果條件為 false，則拋出 CustomTemplateException 帶有指定消息和錯誤類型的一個，以及要在響應中返回的數據。

空行为实用程序

空行為實用程序列表

```
$util.isNull(Object) : Boolean
```

如果提供物件為 null，則傳回真。

```
$util.isNullOrEmpty(String) : Boolean
```

如果提供資料為 null 或空白字串，則傳回真。否則即傳回 false。

```
$util.isNullOrBlank(String) : Boolean
```

如果提供資料為 null 或空白字串，則傳回真。否則即傳回 false。

```
$util.defaultIfNull(Object, Object) : Object
```

如果不是 null，則傳回第一個物件。否則，會傳回第二個物件做為「預設物件」。

```
$util.defaultIfNullOrEmpty(String, String) : String
```

如果不是 null 或空白，則傳回第一個字串。否則，會傳回第二個字串做為「預設字串」。

```
$util.defaultIfNullOrBlank(String, String) : String
```

如果不是 null 或空白，則傳回第一個字串。否則，會傳回第二個字串做為「預設字串」。

模式匹配實用程序

類型和模式匹配實用程序列表

```
$util.typeOf(Object) : String
```

傳回描述物件類型的字串。支援的類型識別為：「Null」、「數字」、「字串」、「映射」、「清單」、「布林值」。如果無法識別類型，則傳回類型為「物件」。

```
$util.matches(String, String) : Boolean
```

如果第一個引數中指定的模式與第二個引數中提供的資料相符，則傳回真。模式必須為規則表達式，例如 `$util.matches("a*b", "aaaaab")`。此功能是根據模式，您可以參考以取得更詳細的文件。

```
$util.authType() : String
```

返回一個字符串，描述請求正在使用的多身份驗證類型，返回「IAM 授權」，「用戶池授權」，「打開 ID Connect 授權」或「API 密鑰授權」。

對象驗證實用程序

對象驗證實用程序列表

```
$util.isString(Object) : Boolean
```

如果對象是一個字符串，則返回 true。

```
$util.isNumber(Object) : Boolean
```

如果對象是一個數字，則返回 true。

```
$util.isBoolean(Object) : Boolean
```

如果對象是布爾值，則返回 true。

```
$util.isList(Object) : Boolean
```

如果對象是一個列表，則返回 true。

```
$util.isMap(Object) : Boolean
```

如果對象是一個地圖返回 true。

CloudWatch 記錄公用程式

CloudWatch 記錄實用程序列表

```
$util.log.info(Object) : Void
```

當 API ALL 上的日誌級別啟用請求級別和字段級別的日誌記錄時，將提供對象的字符串表示 CloudWatch 記錄到請求的日誌流。

`$util.log.info(String, Object...) : Void`

當 API ALL 上的日誌級別啟用請求級別和字段級別日誌記錄時，將所提供對象的字符串表示 CloudWatch 記錄到請求的日誌流。此實用程序將按順序替換所提供對象的字符串表示的第一個輸入格式字符串中由「{}」表示的所有變量。

`$util.log.error(Object) : Void`

當欄位層級記錄啟用 API 上的記錄層級ERROR或 CloudWatch 記錄層級時，將所提供物件的字串表示法記錄到要求的記錄資料流。ALL

`$util.log.error(String, Object...) : Void`

當 API ALL 上的記錄層級ERROR或記錄層級啟用欄位層級 CloudWatch 記錄時，將所提供物件的字串表示法記錄到要求的記錄資料流。此實用程序將按順序替換所提供對象的字符串表示的第一個輸入格式字符串中由「{}」表示的所有變量。

返回值行為實用程序

返回值行為實用程序列表

`$util.qr()` 和 `$util.quiet()`

在隱藏傳回值的同時執行 VTL 陳述式。這對於在不使用臨時佔位符的情況下運行方法非常有用，例如將項目添加到地圖中。例如：

```
#set ($myMap = {})
#set($discard = $myMap.put("id", "first value"))
```

變成：

```
#set ($myMap = {})
$util.qr($myMap.put("id", "first value"))
```

`$util.escapeJavaScript(String) : String`

返回輸入字符串作為 JavaScript 轉義字符串。

`$util.urlEncode(String) : String`

以 `application/x-www-form-urlencoded` 編碼字串的形式傳回輸入字串。

$util.urlDecode(String) : String

將 application/x-www-form-urlencoded 編碼的字串解碼回非編碼格式。

$util.base64Encode(byte[]) : String

將輸入編碼為 base64 編碼字串。

$util.base64Decode(String) : byte[]

解碼 base64 編碼字串中的資料。

$util.parseJson(String) : Object

採用「字串化」JSON，並傳回結果的物件呈現。

$util.toJson(Object) : String

採用物件，並傳回該物件的「字串化」JSON 呈現。

$util.autoId() : String

傳回 128 位元隨機產生的 UUID。

$util.autoUlid() : String

返回一個 128 位隨機生成的 ULID（通用唯一的字典排序標識符）。

$util.autoKsuid() : String

返回 128 位隨機生成的 KSUID（K 可排序的唯一標識符）基於 62 編碼為長度為 27 的字符串。

$util.unauthorized()

擲回欲解析之欄位的 Unauthorized。在請求或響應映射模板中使用此選項，以確定是否允許調用者解析字段。

$util.error(String)

擲回自訂錯誤。在請求或響應映射模板中使用此選項，以檢測請求或調用結果的錯誤。

$util.error(String, String)

擲回自訂錯誤。在請求或響應映射模板中使用此選項，以檢測請求或調用結果的錯誤。您也可以指定errorType.

$util.error(String, String, Object)

擲回自訂錯誤。在請求或響應映射模板中使用此選項，以檢測請求或調用結果的錯誤。您也可以指定errorType和data欄位。data 值將新增到 GraphQL 回應中，error 內對應的 errors 區塊。注意：data 將根據查詢選取範圍集進行篩選。

$util.error(String, String, Object, Object)

擲回自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。您同時可以指定 errorType 欄位、data 欄位和 errorInfo 欄位。data 值將新增到 GraphQL 回應中，error 內對應的 errors 區塊。注意：data 將根據查詢選取範圍集進行篩選。errorInfo 值將新增到 GraphQL 回應中，error 內對應的 errors 區塊。注意：errorInfo 將不會根據查詢選取範圍集進行篩選。

$util.appendError(String)

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。與 $util.error(String) 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。

$util.appendError(String, String)

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。您同時可以指定 errorType。與 $util.error(String, String) 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。

$util.appendError(String, String, Object)

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。您同時可以指定 errorType 和 data 欄位。與 $util.error(String, String, Object) 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。data 值將新增到 GraphQL 回應中，error 內對應的 errors 區塊。注意：data 將根據查詢選取範圍集進行篩選。

$util.appendError(String, String, Object, Object)

附加自訂錯誤。如果範本偵測到要求或呼叫結果的錯誤，您可以將此用於要求或回應映射範本。您同時可以指定 errorType 欄位、data 欄位和 errorInfo 欄位。與 $util.error(String, String, Object, Object) 不同的是，範本評估不會受中斷，因此可以將資料傳回給發起人。data 值將新增到 GraphQL 回應中，error 內對應的 errors 區塊。注意：data 將根據查詢選取範圍集進行篩選。errorInfo 值將新增到 GraphQL 回應中，error 內對應的 errors 區塊。注意：errorInfo 將不會根據查詢選取範圍集進行篩選。

$util.validate(Boolean, String) : void

如果條件為 false，則拋出 CustomTemplateException 帶有指定消息的一個。

$util.validate(Boolean, String, String) : void

如果條件為 false，則拋出 CustomTemplateException 具有指定消息和錯誤類型的一個。

$util.validate(Boolean, String, String, Object) : void

如果條件為 false，則拋出 CustomTemplateException 帶有指定消息和錯誤類型的一個，以及要在響應中返回的數據。

$util.isNull(Object) : Boolean

如果提供物件為 null，則傳回真。

$util.isNullOrEmpty(String) : Boolean

如果提供資料為 null 或空白字串，則傳回真。否則即傳回 false。

$util.isNullOrBlank(String) : Boolean

如果提供資料為 null 或空白字串，則傳回真。否則即傳回 false。

$util.defaultIfNull(Object, Object) : Object

如果不是 null，則傳回第一個物件。否則，會傳回第二個物件做為「預設物件」。

$util.defaultIfNullOrEmpty(String, String) : String

如果不是 null 或空白，則傳回第一個字串。否則，會傳回第二個字串做為「預設字串」。

$util.defaultIfNullOrBlank(String, String) : String

如果不是 null 或空白，則傳回第一個字串。否則，會傳回第二個字串做為「預設字串」。

$util.isString(Object) : Boolean

如果物件是字串，則傳回真。

$util.isNumber(Object) : Boolean

如果物件是數字，則傳回真。

$util.isBoolean(Object) : Boolean

如果物件是布林值，則傳回真。

$util.isList(Object) : Boolean

如果物件是清單，則傳回真。

$util.isMap(Object) : Boolean

如果物件是映射，則傳回真。

$util.typeOf(Object) : String

傳回描述物件類型的字串。支援的類型識別為：「Null」、「數字」、「字串」、「映射」、「清單」、「布林值」。如果無法識別類型，則傳回類型為「物件」。

$util.matches(String, String) : Boolean

如果第一個引數中指定的模式與第二個引數中提供的資料相符，則傳回真。模式必須為規則表達式，例如 `$util.matches("a*b", "aaaaab")`。此功能是根據模式，您可以參考以取得更詳細的文件。

$util.authType() : String

返回一個字符串，描述請求正在使用的多身份驗證類型，返回「IAM 授權」，「用戶池授權」，「打開 ID Connect 授權」或「API 密鑰授權」。

$util.log.info(Object) : Void

當 API ALL 上的日誌級別啟用請求級別和字段級別的日誌記錄時，將提供對象的字符串表示 CloudWatch 記錄到請求的日誌流。

$util.log.info(String, Object...) : Void

當 API ALL 上的日誌級別啟用請求級別和字段級別日誌記錄時，將所提供對象的字符串表示 CloudWatch 記錄到請求的日誌流。此實用程序將按順序替換所提供對象的字符串表示的第一個輸入格式字符串中由「{}」表示的所有變量。

$util.log.error(Object) : Void

當欄位層級記錄啟用 API 上的記錄層級ERROR或 CloudWatch 記錄層級時，將所提供物件的字串表示法記錄到要求的記錄資料流。ALL

$util.log.error(String, Object...) : Void

當 API ALL 上的記錄層級ERROR或記錄層級啟用欄位層級 CloudWatch 記錄時，將所提供物件的字串表示法記錄到要求的記錄資料流。此實用程序將按順序替換所提供對象的字符串表示的第一個輸入格式字符串中由「{}」表示的所有變量。

`$util.escapeJavaScript(String) : String`

返回輸入字符串作為 JavaScript 轉義字符串。

解析器授權

解析器授權列表

`$util.unauthorized()`

擲回欲解析之欄位的 Unauthorized。在請求或響應映射模板中使用此選項，以確定是否允許調用者解析字段。

AWS AppSync 指令

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync 在使用 VTL 編寫時，公開指令以提高開發人員的生產力。

指令實用程序

`#return(Object)`

`#return(Object)`可讓您提前從任何對應範本傳回。 `#return(Object)`類似於編程語言中的 return 關鍵字，因為它將從最接近的邏輯範圍塊返回。使用解析器映射模板`#return(Object)`內部將從解析器返回。此外，`#return(Object)`從函數映射模板中使用將從函數返回，並將繼續運行到管道中的下一個函數或解析器響應映射模板。

`#return`

該`#return`指令表現出與相同的行為`#return(Object)`，但`null`將返回。

時間助手在 $ 實用時間

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

$util.time 變數包含日期時間方法，可協助產生時間戳記、在日期時間格式之間轉換，以及剖析日期時間字串。日期時間格式的語法基於您可以參考DateTimeFormatter以獲取進一步的文檔。我們在下面提供了一些示例，以及可用的方法和描述的列表。

时间实用程序

時間實用程序列表

```
$util.time.nowISO8601() : String
```

以 ISO8601 格式傳回 UTC 的字串表示方式。

```
$util.time.nowEpochSeconds() : long
```

傳回從 1970-01-01T00:00:00Z 的 epoch 到現在的秒數。

```
$util.time.nowEpochMilliSeconds() : long
```

傳回從 1970-01-01T00:00:00Z 的 epoch 到現在的毫秒數。

```
$util.time.nowFormatted(String) : String
```

使用字串輸入類型的指定格式，傳回 UTC 中目前時間戳記的字串。

```
$util.time.nowFormatted(String, String) : String
```

使用字串輸入類型的指定格式和時區，傳回時區目前時間戳記的字串。

```
$util.time.parseFormattedToEpochMilliSeconds(String, String) : Long
```

解析作為字符串傳遞的時間戳以及格式，然後返回自 epoch 以毫秒為單位的時間戳。

```
$util.time.parseFormattedToEpochMilliSeconds(String, String, String) : Long
```

解析作為字符串傳遞的時間戳以及格式和時區，然後返回自 epoch 以毫秒為單位的時間戳。

```
$util.time.parseISO8601ToEpochMilliSeconds(String) : Long
```

解析作為字符串傳遞的 ISO8601 時間戳，然後返回自紀元以毫秒為單位的時間戳。

```
$util.time.epochMilliSecondsToSeconds(long) : long
```

將 epoch 毫秒時間戳記轉換為 epoch 秒時間戳記。

```
$util.time.epochMilliSecondsToISO8601(long) : String
```

將紀元毫秒時間戳記轉換為 ISO8601 時間戳記。

```
$util.time.epochMilliSecondsToFormatted(long, String) : String
```

將紀元毫秒時間戳記 (只要傳遞) 轉換為根據提供的 UTC 格式格式化的時間戳記。

`$util.time.epochMilliSecondsToFormatted(long, String, String) : String`

將紀元毫秒時間戳記 (長時間傳遞) 轉換為根據提供的時區提供的格式格式化的時間戳記。

獨立功能示例

```
$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
```

轉換範例

```
#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
```

剖析範例

```
$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
```

AWS AppSync 定義標量的用法

以下格式與 AWSDate、AWSDateTime 和 AWSTime 相容。

```
$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
```

列表中的助手列表

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

`$util.list`包含一般 List 作業 (例如移除或保留清單中的項目以篩選使用案例) 的方法。

列出實用程序

`$util.list.copyAndRetainAll(List, List) : List`

在第一個參數中製作提供的列表的淺表副本，同時僅保留在第二個參數中指定的項目（如果存在）。所有其他項目將從副本中刪除。

`$util.list.copyAndRemoveAll(List, List) : List`

在第一個參數中製作提供的列表的淺表副本，同時刪除在第二個參數中指定該項目的任何項目（如果存在）。所有其他項目將保留在副本中。

`$util.list.sortList(List, Boolean, String) : List`

排序對象，這是在第一個參數中提供的列表。如果第二個參數為 true，則列表以降序方式排序; 如果第二個參數為 false，則列表以升序排序。第三個引數是用於排序自訂物件清單的屬性的字串名稱。如果它只是字符串，整數，浮點數或雙精度的列表，第三個參數可以是任何隨機字符串。如果所有對象都不是來自同一個類，則返回原始列表。僅支援包含最多 1000 個物件的清單。以下是此公用程式用法的範例：

```
 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}]
```

地圖助手在 $ 公用程序地圖

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

`$util.map`包含方法來幫助常見的 Map 操作，例如從 Map 中刪除或保留項目以過濾用例。

地圖實用程序

`$util.map.copyAndRetainAllKeys(Map, List) : Map`

使第一個映射的淺表副本，同時只保留列表中指定的鍵（如果它們存在）。所有其他金鑰將從副本中刪除。

`$util.map.copyAndRemoveAllKeys(Map, List) : Map`

使第一個映射的淺表副本，同時刪除列表中指定鍵的任何條目（如果它們存在）。所有其他金鑰將保留在副本中。

$util.dynamodb 中的 DynamoDB 協助程式

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

`$util.dynamodb`包含協助程式方法，可讓您更輕鬆地將資料寫入和讀取 Amazon DynamoDB，例如自動類型對應和格式化。這些方法的設計目的是讓原始類型和清單自動對應至適當的 DynamoDB 輸入格式，這是一`Map`種格式。`{ "TYPE" : VALUE }`

例如，先前在 DynamoDB 中建立新項目的請求對應範本可能如下所示：

```
{
    "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) }
    }
}
```

如果我們想要將欄位新增到物件，必須在結構描述中更新 GraphQL 查詢和要求映射範本。不過，我們現在可以重新架構請求對應範本，以便自動挑選結構描述中新增的新欄位，並使用正確的類型將這些欄位新增至 DynamoDB：

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

在前面的範例中，我們使用`$util.dynamodb.toDynamoDBJson(...)`輔助程式自動取得產生的 id，並將其轉換為字串屬性的 DynamoDB 表示法。然後，我們將所有引數轉換為其 DynamoDB 表示法，並將其輸出到範本中的`attributeValues`欄位。

每個協助程式有兩個版本：傳回物件的版本 (例如，`$util.dynamodb.toString(...)`)，以及傳回物件做為 JSON 字串的版本 (例如，`$util.dynamodb.toStringJson(...)`)。在前述範例中，我們使用傳回資料做為 JSON 字串的版本。如果您想要在範本使用物件之前，對物件進行操作，可以選擇傳回物件，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)
}
```

在前述範例中，我們傳回轉換引數做為映射而非 JSON 字串，然後在使用 `version` 將其輸出到範本的 `timestamp` 欄位之前，新增 `attributeValues` 和 `$util.toJson(...)` 欄位。

每個 JSON 版本的協助程式等同於以 `$util.toJson(...)` 包裝非 JSON 版本。例如，下列陳述式完全相同：

```
$util.toStringJson("Hello, World!")
$util.toJson($util.toString("Hello, World!"))
```

到 DynamoDB

到實用程序列表

`$util.dynamodb.toDynamoDB(Object) : Map`

DynamoDB 的一般物件轉換工具，可將輸入物件轉換為適當的 DynamoDB 表示法。它在代表一些類型的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「SS」、「NS」、「BS」)。這會傳回描述 DynamoDB 屬性值的物件。

字符串示例

```
Input:      $util.dynamodb.toDynamoDB("foo")
Output:     { "S" : "foo" }
```

數字示例

```
Input:      $util.dynamodb.toDynamoDB(12345)
Output:     { "N" : 12345 }
```

布爾示例

```
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" }
                       }
                   }
               ]
           }
```

地圖示例

```
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`

與，相同`$util.dynamodb.toDynamoDB(Object) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

。toString 實用程序。

toString 實用程序列表

`$util.dynamodb.toString(String) : String`

將輸入字串轉換為 DynamoDB 字串格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toString("foo")
Output:     { "S" : "foo" }
```

`$util.dynamodb.toStringJson(String) : Map`

與，相同`$util.dynamodb.toString(String) : String`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

`$util.dynamodb.toStringSet(List<String>) : Map`

將包含字串的清單轉換為 DynamoDB 字串集格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toStringSet([ "foo", "bar", "baz" ])
Output:     { "SS" : [ "foo", "bar", "baz" ] }
```

`$util.dynamodb.toStringSetJson(List<String>) : String`

與，相同`$util.dynamodb.toStringSet(List<String>) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

音量号实用程序

音量號碼實用程序列表

`$util.dynamodb.toNumber(Number) : Map`

將數字轉換 DynamoDB 數字格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toNumber(12345)
Output:     { "N" : 12345 }
```

`$util.dynamodb.toNumberJson(Number) : String`

與，相同`$util.dynamodb.toNumber(Number) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

`$util.dynamodb.toNumberSet(List<Number>) : Map`

將數字清單轉換為 DynamoDB 數字集格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toNumberSet([ 1, 23, 4.56 ])
Output:     { "NS" : [ 1, 23, 4.56 ] }
```

`$util.dynamodb.toNumberSetJson(List<Number>) : String`

與，相同`$util.dynamodb.toNumberSet(List<Number>) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

二元實用程序

二進制實用程序列表

`$util.dynamodb.toBinary(String) : Map`

將編碼為 base64 字串的二進位資料轉換為 DynamoDB 二進位格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toBinary("foo")
Output:     { "B" : "foo" }
```

`$util.dynamodb.toBinaryJson(String) : String`

與，相同`$util.dynamodb.toBinary(String) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

`$util.dynamodb.toBinarySet(List<String>) : Map`

將編碼為 base64 字串的二進位資料清單轉換為 DynamoDB 二進位集格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toBinarySet([ "foo", "bar", "baz" ])
Output:     { "BS" : [ "foo", "bar", "baz" ] }
```

`$util.dynamodb.toBinarySetJson(List<String>) : String`

與，相同`$util.dynamodb.toBinarySet(List<String>) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

到布爾實用程序

到布爾實用程序列表

`$util.dynamodb.toBoolean(Boolean) : Map`

將布林值轉換為適當的 DynamoDB 布林格式。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toBoolean(true)
Output:     { "BOOL" : true }
```

`$util.dynamodb.toBooleanJson(Boolean) : String`

與，相同`$util.dynamodb.toBoolean(Boolean) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

音拉實用程序

音效應用程式清單

`$util.dynamodb.toNull() : Map`

以空值格 DynamoDB 回空值。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toNull()
Output:     { "NULL" : null }
```

`$util.dynamodb.toNullJson() : String`

與，相同`$util.dynamodb.toNull() : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

到列表實用程序

。主列表實用程序列表。

`$util.dynamodb.toList(List) : Map`

將物件清單轉換為 DynamoDB 清單格式。清單中的每個項目也會轉換為其適當的 DynamoDB 格式。它在代表一些巢狀物件的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「SS」、「NS」、「BS」)。這會傳回描述 DynamoDB 屬性值的物件。

```
Input:      $util.dynamodb.toList([ "foo", 123, { "bar" : "baz" } ])
Output:     {
               "L" : [
                   { "S" : "foo" },
                   { "N" : 123 },
                   {
                       "M" : {
```

```
                        "bar" : { "S" : "baz" }
                    }
                }
            ]
        }
```

`$util.dynamodb.toListJson(List) : String`

與，相同`$util.dynamodb.toList(List) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

toMap 應用程序

。toMap 實用程序列表。

`$util.dynamodb.toMap(Map) : Map`

將地圖轉換為 DynamoDB 對映格式。地圖中的每個值也會轉換為其適當的 DynamoDB 格式。它在代表一些巢狀物件的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「SS」、「NS」、「BS」)。這會傳回描述 DynamoDB 屬性值的物件。

```
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`

與，相同`$util.dynamodb.toMap(Map) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

`$util.dynamodb.toMapValues(Map) : Map`

建立對映的副本，其中每個值都已轉換為適當的 DynamoDB 格式。它在代表一些巢狀物件的方式上是固定的：例如，它會使用清單 (「L」) 而不是集合 (「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**
>
> 這與稍有不同，`$util.dynamodb.toMap(Map) : Map`因為它只會傳回 DynamoDB 屬性值的內容，但不會傳回整個屬性值本身。例如，下列陳述式完全相同：
>
> ```
> $util.dynamodb.toMapValues($map)
> $util.dynamodb.toMap($map).get("M")
> ```

`$util.dynamodb.toMapValuesJson(Map) : String`

與，相同`$util.dynamodb.toMapValues(Map) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

S3 对象实用程序

S3 對象實用程序列表

`$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`

將金鑰、儲存貯體和區域轉換為 DynamoDB S3 物件表示法。這會傳回描述 DynamoDB 屬性值的物件。

```
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`

與，相同`$util.dynamodb.toS3Object(String key, String bucket, String region) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

`$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map`

將金鑰、儲存貯體、區域和選用版本轉換為 DynamoDB S3 物件表示法。這會傳回描述 DynamoDB 屬性值的物件。

```
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`

與，相同`$util.dynamodb.toS3Object(String key, String bucket, String region, String version) : Map`，但會以 JSON 編碼字串的形式傳回 DynamoDB 屬性值。

`$util.dynamodb.fromS3ObjectJson(String) : Map`

接受 DynamoDB S3 物件的字串值，並傳回包含金鑰、儲存貯體、區域和選用版本的對映。

```
Input:      $util.dynamodb.fromS3ObjectJson({ "S" : "{ \"s3\" : { \"key\" : \"foo\",
 \"bucket\" : \"bar\", \"region\" : \"baz\", \"version\" = \"beep\" } }" })
Output:     { "key" : "foo", "bucket" : "bar", "region" : "baz", "version" :
 "beep" }
```

Amazon RDS 助手在 $ 實用程序

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

`$util.rds`包含透過移除結果輸出中的無關資料來格式化 Amazon RDS 操作的協助程式方法

$ 實用程序列表

$util.rds.toJsonString(String serializedSQLResult): String

傳回的方String式是將字串化的原始 Amazon Relational Database Service (Amazon RDS) 資料 API 作業結果格式轉換為更簡潔的字串。傳回的字串是結果集 SQL 記錄的序列化清單。每筆記錄都會顯示為鍵值對集合。索引鍵是對應的欄位名稱。

如果輸入中的對應陳述式是導致變異的 SQL 查詢 (例如 INSERT、UPDATE、DELETE)，則會傳回空白清單。例如，查詢會select * from Books limit 2提供 Amazon RDS 資料操作的原始結果：

```
{
    "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`

這是相同的`util.rds.toJsonString`，但結果是一個 JSON `Object`。

HTTP 助手在美元工具

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

該實用`$util.http`程序提供了幫助程序方法，您可以使用這些方法來管理 HTTP 請求參數和添加響應標頭。

$ 實用程序列表

$util.http.copyHeaders(Map) : Map

從地圖複製標頭，而不受限制的 HTTP 標頭集。您可以使用它將請求標頭轉發到下游 HTTP 端點。

```
{
    ...
    "params": {
        ...
        "headers": $util.http.copyHeaders($ctx.request.headers),
        ...
    },
    ...
}
```

$util.http.addResponseHeader(String, Object)

添加一個單一的自定義標題與名稱（String）和 value（Object）的響應。有下列限制：

- 標頭名稱不能匹配任何現有的或受限制的 AWS 或 AWS AppSync標題。
- 標頭名稱不能以受限制的前綴開頭，例如x-amzn-或x-amz-。
- 自訂回應標頭的大小不得超過 4 KB。這包括標題名稱和值。
- 您應該在每個 GraphQL 作業中定義每個回應標頭一次。但是，如果您多次定義具有相同名稱的自訂標頭，則回應中會顯示最新的定義。無論命名如何，所有標題都計入標題大小限制。

```
...
$util.http.addResponseHeader("itemsCount", 7)
$util.http.addResponseHeader("render", $ctx.args.render)
...
```

$util.http.addResponseHeaders(Map)

將多個響應頭添加到指定映射的名稱（String）和 value（Object）的響應。
該addResponseHeader(String, Object)方法列出的相同限制也適用於此方法。

```
...
#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)
...
```

在美元實用程序的 XML 助手

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

`$util.xml`包含輔助方法，可以更輕鬆地將 XML 響應轉換為 JSON 或字典。

。實用程序列表。

`$util.xml.toMap(String) : Map`

將 XML 字串轉換為字典。

```
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

將 XML 字串轉換為 JSON 字串。這與 ToMap 類似，不同之處在於輸出是一個字符串。如果要將來自 HTTP 物件的 XML 回應直接轉換並傳回給 JSON，這非常實用。

$util.xml.toJsonString(String, Boolean) : String

使用選用的布林參數將 XML 字串轉換為 JSON 字串，以判斷是否要對 JSON 進行字串編碼。

轉換助手在 $ 實用變換

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

`$util.transform`包含協助程式方法，可讓您更輕鬆地對資料來源 (例如 Amazon DynamoDB 篩選器操作) 執行複雜的操作。

轉型助手

轉換助手實用程序列表

`$util.transform.toDynamoDBFilterExpression(Map) : Map`

將輸入字串轉換為篩選器運算式，以與 DynamoDB 搭配使用。

```
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`

將指定的輸入轉換為其對等的 OpenSearch 查詢 DSL 運算式，並將其傳回為 JSON 字串。

```
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"
                    }
                  }
              ]
            }
          }
      ]
    }
}
```

預設運算子假設為 AND。

轉換助手訂閱過濾器

轉換助手訂閱過濾器實用程序列表

`$util.transform.toSubscriptionFilter(Map) : Map`

將Map輸入物件轉換為SubscriptionFilter運算式物件。
該$util.transform.toSubscriptionFilter方法被用
作$extensions.setSubscriptionFilter()擴展的輸入。如需詳細資訊，請參閱擴充功能。

`$util.transform.toSubscriptionFilter(Map, List) : Map`

將Map輸入物件轉換為SubscriptionFilter運算式物件。
該$util.transform.toSubscriptionFilter方法被用
作$extensions.setSubscriptionFilter()擴展的輸入。如需詳細資訊，請參閱擴充功能。

第一個引數是轉換為SubscriptionFilter運算式物件的Map輸入物件。第二個參數是在構
建SubscriptionFilter表達式對象時在第一個Map輸入對象中忽略List的字段名稱。

$util.transform.toSubscriptionFilter(Map, List, Map) : Map

將Map輸入物件轉換為SubscriptionFilter運算式物件。
該$util.transform.toSubscriptionFilter方法被用
作$extensions.setSubscriptionFilter()擴展的輸入。如需詳細資訊，請參閱擴充功能。

第一個參數是轉換為SubscriptionFilter表達式對象的Map輸入對象，第二個參數是將
在List第一個Map輸入對象中忽略的字段名稱，第三個參數是構建SubscriptionFilter表達式
對象時包含的嚴格規則的Map輸入對象。這些嚴格規則包含在SubscriptionFilter運算式物件
中，以便至少滿足其中一個規則以通過訂閱篩選器。

訂閱篩選引數

下表說明如何定義下列公用程式的引數：

- $util.transform.toSubscriptionFilter(Map) : Map
- $util.transform.toSubscriptionFilter(Map, List) : Map
- $util.transform.toSubscriptionFilter(Map, List, Map) : Map

Argument 1: Map

引數 1 是具有下列索引鍵值的Map物件：

- 欄位名稱
- 「和」
- 「或」

對於作為鍵的欄位名稱，這些欄位項目中的條件格式為。"operator" : "value"

下列範例顯示如何將項目新增至Map：

```
"field_name" : {
                    "operator1" : value
               }

## We can have multiple conditions for the same field_name:

"field_name" : {
                    "operator1" : value
```

```
                    "operator2" : value
                    .
                    .
                    .
                }
```

當欄位上有兩個或多個條件時，所有這些條件都會被視為使用 OR 作業。

輸入也Map可以有「and」和「or」作為鍵，這意味著應該使用 AND 或 OR 邏輯連接這些內的所有條目，具體取決於密鑰。索引鍵值「and」和「or」需要一組條件。

```
"and" : [

            {
                "field_name1" : {
                    "operator1" : value
                }
             },

             {
                "field_name2" : {
                    "operator1" : value
                }
             },
             .
             .
        ].
```

請注意，您可以嵌套「和」和「or」。也就是說，您可以在另一個「和」/「或」塊中嵌套「和」/「或」塊。但是，這不適用於簡單的字段。

```
"and" : [

            {
                "field_name1" : {
                    "operator" : value
                }
             },

             {
                "or" : [
                            {
                                "field_name2" : {
```

```
                                    "operator" : value
                                }
                            },

                            {
                                "field_name3" : {
                                    "operator" : value
                                }
                            }

                        ].
```

下面的例子顯示了使用參數 1 的輸入$util.transform.toSubscriptionFilter(Map) : Map。

輸入

參數 1：地圖：

```
{
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 2000
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
```

```
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

輸出

結果是一個Map對象：

```
{
  "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

參數 2 包含在構建SubscriptionFilter表達式對象時不應在輸入Map（參數 1）中考慮List的字段名稱。也List可以是空的。

下面的例子顯示了使用參數 1 和參數 2 的輸入$util.transform.toSubscriptionFilter(Map, List) : Map。

輸入

參數 1：地圖：

```
{

  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
```

```
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "gt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

引數 2：列表：

```
["percentageUp", "author"]
```

輸出

結果是一個Map對象：

```
{
  "filterGroup": [
    {
      "filters": [
        {
          "fieldName": "title",
          "operator": "ne",
          "value": "Book1"
        },
        {
```

```
          "fieldName": "downvotes",
          "operator": "gt",
          "value": 20
        },
        {
          "fieldName": "isPublished",
          "operator": "eq",
          "value": false
        }
      ]
    }
  ]
}
```

Argument 3: Map

引數 3 是以欄位名稱作為鍵值的Map物件 (不能有「and」或「or」)。對於作為鍵的欄位名稱，這些欄位上的條件是形式的項目"operator" : "value"。與引數 1 不同，參數 3 在同一個鍵中不能有多個條件。此外，參數 3 沒有「and」或「or」子句，因此也不涉及嵌套。

引數 3 代表嚴格規則的清單，這些規則會新增至SubscriptionFilter運算式物件，以便至少符合這些條件中的一個以傳遞篩選器。

```
{
  "fieldname1": {
    "operator": value
  },
  "fieldname2": {
    "operator": value
  }
}
.
.
.
```

下面的例子顯示了參數 1，參數 2 和參數 3 使用的輸入$util.transform.toSubscriptionFilter(Map, List, Map) : Map。

輸入

參數 1：地圖：

```
{
```

```
  "percentageUp": {
    "lte": 50,
    "gte": 20
  },
  "and": [
    {
      "title": {
        "ne": "Book1"
      }
    },
    {
      "downvotes": {
        "lt": 20
      }
    }
  ],
  "or": [
    {
      "author": {
        "eq": "Admin"
      }
    },
    {
      "isPublished": {
        "eq": false
      }
    }
  ]
}
```

引數 2：列表：

```
["percentageUp", "author"]
```

參數 3：地圖：

```
{
  "upvotes": {
    "gte": 250
  },
  "author": {
    "eq": "Person1"
  }
```

```
}
```

輸出

結果是一個Map對象：

```
{
  "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"
        }
      ]
    }
  ]
}
```

數學助手中的數學助手

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

`$util.math`包含幫助進行常見數學運算的方法。

數學實用程序列表

`$util.math.roundNum(Double) : Integer`

取一個 double 並將其四捨五入為最接近的整數。

`$util.math.minVal(Double, Double) : Double`

需要兩個雙打，並返回兩個雙打之間的最小值。

`$util.math.maxVal(Double, Double) : Double`

需要兩個雙打，並返回兩個雙打之間的最大值。

`$util.math.randomDouble() : Double`

返回 0 和 1 之間的隨機雙。

> **Important**
>
> 這個函數不應該用於任何需要高熵隨機性的東西（例如，密碼學）。

`$util.math.randomWithinRange(Integer, Integer) : Integer`

傳回指定範圍內的隨機整數值，第一個引數會指定範圍的較低值，而第二個引數則指定範圍的上限值。

> **Important**
>
> 這個函數不應該用於任何需要高熵隨機性的東西（例如，密碼學）。

字符串助手在 $ 實用程序

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

`$util.str`包含幫助常見字符串操作的方法。

$util.str 實用程序列表

`$util.str.toUpper(String) : String`

接受一個字符串並將其轉換為完全大寫。

`$util.str.toLower(String) : String`

需要一個字符串並將其轉換為完全小寫。

`$util.str.toReplace(String, String, String) : String`

用另一個字符串替換字符串中的子字符串。第一個引數指定要在其上執行替換操作的字符串。第二個參數指定要替換的子字符串。第三個參數指定用來替換第二個參數的字符串。以下是此公用程式用法的範例：

```
INPUT:      $util.str.toReplace("hello world", "hello", "mellow")
OUTPUT:     "mellow world"
```

`$util.str.normalize(String, String) : String`

使用以下四種 Unicode 標準化形式之一來標準化字符串：NFC，NFD，NFKC 或 NFKD。第一個參數是要規範化的字符串。第二個參數是「nfc」，「nfd」，「nfkc」或「nfkd」，指定用於正規化過程的標準化類型。

延伸模組

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

`$extensions`包含一組在解析器中執行其他動作的方法。

$ 擴展。 evictFromApi緩存（字符串，字符串，對象）：對象

從 AWS AppSync 伺服器端快取中收回項目。第一個參數是類型名稱。第二個引數是欄位名稱。第三個參數是包含指定緩存鍵值的鍵-值對項目的對象。您必須按照與緩存解析器中的緩存密鑰相同的順序放置對象中的項目。`cachingKey`

> **Note**
>
> 此實用程序僅適用於突變，而不適用於查詢。

$ 擴展。 setSubscriptionFilter(filterJsonObject)

定義增強的訂閱篩選器。系統會根據提供的訂閱篩選器評估每個訂閱通知事件，並在所有篩選器評估為時向`true`用戶端傳送通知 引數`filterJsonObject`如下所述。

> **Note**
>
> 您只能在訂閱解析器的回應對應範本中使用此延伸方法。

$ 擴展。 setSubscriptionInvalidation過濾器 (filterJsonObject)

定義訂閱失效篩選器。會根據無效承載評估訂閱篩選器，然後如果篩選器評估為，則會使指定的訂閱失效。`true`引數`filterJsonObject`如下所述。

> **Note**
>
> 您只能在訂閱解析器的回應對應範本中使用此延伸方法。

引數： filterJsonObject

JSON 物件會定義訂閱或無效驗證篩選器。它是一個篩選器陣列中的`filterGroup`. 每個濾鏡都是個別篩選器的集合。

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

每個篩選器都有三個屬性：

- fieldName— [圖 GraphQL 結構描述] 欄位。
- operator— 運算子類型。
- value— 要與訂閱通知值進行比較的fieldName值。

以下是這些屬性的範例指派：

```
{
 "fieldName" : "severity",
 "operator" : "le",
 "value" : $context.result.severity
}
```

欄位：fieldName

字串類型fieldName是指 GraphQL 結構描述中定義的欄位，該欄位與訂閱通知承載fieldName中的相符。找到相符項目時，GraphQL 結構描述欄位會與訂閱通知篩選器value的比較。value在下列範例中，fieldName篩選器會與指定 GraphQL 類型中定義的service欄位相符。如果通知有效負載包含value相當於的service欄位AWS AppSync，則篩選器評估為true：

```
{
 "fieldName" : "service",
 "operator" : "eq",
 "value" : "AWS AppSync"
}
```

欄位：值

該值可以是基於運算符的不同類型：

- 單個數字或布爾
  - 字符串示例："test"，"service"
  - 數字示例:1,2, 45.75
  - 布林值範例：true、false
- 成對的數字或字符串
  - 字符串對示例：["test1","test2"]，["start","end"]
  - 數字對示例：[1,4]，[67,89]，[12.45, 95.45]
- 數字或字符串的數組

- 字符串數組示例：`["test1","test2","test3","test4","test5"]`
- 數組的例子：`[1,2,3,4,5]`，`[12.11,46.13,45.09,12.54,13.89]`

欄位：運算子

具有以下可能值的區分大小寫的字符串：

運算子	描述	可能的值類型
eq	等於	整數，浮點數，字符串，布爾
氖	不等於	整數，浮點數，字符串，布爾
樂	小於或等於	整數，浮點數，字符串
lt	小於	整數，浮點數，字符串
GE	大於或等於	整數，浮點數，字符串
gt	大於	整數，浮點數，字符串
contains	檢查集合中的子序列或值。	整數，浮點數，字符串
不包含	檢查集合中是否存在子序列或不存在值。	整數，浮點數，字符串
開始與	檢查前置詞。	string
in	檢查是否符合清單中的元素。	整數，浮點數或字符串的數組
notIn	檢查不在列表中的匹配元素。	整數，浮點數或字符串的數組
介於	兩個值之間	整數，浮點數，字符串
包含任何	包含共同元素	整數，浮點數，字符串

下表說明如何在訂閱通知中使用每個運算子。

eq (equal)

eq運算子會評估訂閱通知欄位值是true否符合且嚴格等於篩選器的值。在下列範例中，篩選器會評估訂閱通知是true否具有等於值的service欄位AWS AppSync。

可能的值類型：整數，浮點數，字符串，布爾

```
{
 "fieldName" : "service",
 "operator" : "eq",
 "value" : "AWS AppSync"
}
```

ne (not equal)

ne運算子會評估訂閱通知欄位值是true否與篩選器的值不同。在下列範例中，篩選器會評估訂閱通知是true否具有與值不同的service欄位AWS AppSync。

可能的值類型：整數，浮點數，字符串，布爾

```
{
 "fieldName" : "service",
 "operator" : "ne",
 "value" : "AWS AppSync"
}
```

le (less or equal)

le運算子會評估訂閱通知欄位值是true否小於或等於篩選器的值。在下列範例中，篩選器會評估訂閱通知是true否具有小於或等於值的size欄位5。

可能的值類型：整數，浮點數，字符串

```
{
 "fieldName" : "size",
 "operator" : "le",
 "value" : 5
}
```

lt (less than)

lt運算子會評估訂閱通知欄位值是true否低於篩選器的值。在下列範例中，篩選器會評估訂閱通知是true否有值小於的size欄位5。

可能的值類型：整數，浮點數，字符串

```
{
 "fieldName" : "size",
 "operator" : "lt",
 "value" : 5
}
```

ge (greater or equal)

ge運算子會評估訂閱通知欄位值是true否大於或等於篩選器的值。在下列範例中，篩選器會評估訂閱通知是true否具有大於或等於值的size欄位5。

可能的值類型：整數，浮點數，字符串

```
{
 "fieldName" : "size",
 "operator" : "ge",
 "value" : 5
}
```

gt (greater than)

gt運算子會評估訂閱通知欄位值是true否大於篩選器的值。在下列範例中，篩選器會評估訂閱通知是true否具有大於值的size欄位5。

可能的值類型：整數，浮點數，字符串

```
{
 "fieldName" : "size",
 "operator" : "gt",
 "value" : 5
}
```

contains

contains運算子會檢查集合或單一項目中的子字串、子序列或值。含有contains運算子的篩選器會評估訂閱通知欄位值是true否包含篩選器值。在下列範例中，篩選器會評估訂閱通知是true否具有包含該值之陣列值的seats欄位10。

可能的值類型：整數，浮點數，字符串

```
{
 "fieldName" : "seats",
 "operator" : "contains",
 "value" : 10
}
```

在另一個範例中，篩選器會評估訂閱通知是true否具有launch作為子字串的event欄位。

```
{
 "fieldName" : "event",
 "operator" : "contains",
 "value" : "launch"
}
```

notContains

notContains運算子會檢查集合或單一項目中是否存在子字串、子序列或值。含有notContains運算子的篩選器會評估訂閱通知欄位值是true否不包含篩選器值。在下列範例中，篩選器會評估訂閱通知是true否具有不包含值的陣列值的seats欄位10。

可能的值類型：整數，浮點數，字符串

```
{
 "fieldName" : "seats",
 "operator" : "notContains",
 "value" : 10
}
```

在另一個範例中，篩選器會評估訂閱通知是true否有event欄位值而不launch做為其子序列。

```
{
 "fieldName" : "event",
 "operator" : "notContains",
 "value" : "launch"
}
```

beginsWith

beginsWith運算符檢查字符串中的前綴。包含beginsWith運算子的篩選器會評估訂閱通知欄位值是true否以篩選器的值開頭。在下列範例中，篩選器會評估訂閱通知是true否具有開頭值的service欄位AWS。

可能的值類型：字符串

```
{
 "fieldName" : "service",
 "operator" : "beginsWith",
 "value" : "AWS"
}
```

in

in運營商檢查數組中匹配的元素。包含in運算子的篩選器會評估訂閱通知欄位值是否存在於陣列中。true在下列範例中，篩選器會評估訂閱通知是true否具有包含陣列中存在值之一的severity欄位：[1,2,3]。

可能的值類型：整數，浮點數或字符串的數組

```
{
 "fieldName" : "severity",
 "operator" : "in",
 "value" : [1,2,3]
}
```

notIn

notIn運營商檢查數組中缺少的元素。包含notIn運算子的篩選器會評估為訂閱通知欄位值是true否不存在於陣列中。在下列範例中，篩選器會評估訂閱通知是true否有severity欄位，其中一個值不存在於陣列中：[1,2,3]。

可能的值類型：整數，浮點數或字符串的數組

```
{
 "fieldName" : "severity",
 "operator" : "notIn",
 "value" : [1,2,3]
}
```

between

between運算子會檢查兩個數字或字串之間的值。包含between運算子的篩選器會評估訂閱通知欄位值是true否介於篩選器的值配對之間。在下列範例中，篩選器會評估訂閱通知是true否有值為23、的severity欄位4。

可能的值類型：一對整數，浮點數或字符串

```
{
 "fieldName" : "severity",
 "operator" : "between",
 "value" : [1,5]
}
```

containsAny

containsAny運算子會檢查陣列中的共同元素。含有containsAny運算子的篩選器會評估訂閱通知欄位設定值與篩選器集值的交集是true否非空白。在下列範例中，篩選器會評估訂閱通知是true否具有包含10或之陣列值的seats欄位15。這表示篩選器會評估訂閱通知的seats欄位值是true否為[10,11]或[15,20,30]。

可能的值類型：整數，浮點數或字符串

```
{
 "fieldName" : "seats",
 "operator" : "containsAny",
 "value" : [10, 15]
}
```

AND 邏輯

透過在filterGroup陣列中的filters物件內定義多個項目，您可以使用 AND 邏輯合併多個濾鏡。在下列範例中，篩選器會評估訂閱通知是trueuserId否具有相當於 1 AND group 欄位值的欄位值為Admin或Developer。

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
```

```
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
            ]

        }
    ]
}
```

OR 邏輯

您可以透過在filterGroup陣列中定義多個濾鏡物件，使用 OR 邏輯合併多個濾鏡。在下列範例中，篩選器會評估訂閱通知的userId欄位是true否具有等於 1 OR group 欄位值Admin或的欄位值Developer。

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                }
           ]

        },
        {
           "filters" : [
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]

        }
    ]
}
```

例外狀況

請注意，使用篩選器有幾種限制：

- 在filters物件中，每個篩選器最多可有五個唯一fieldName項目。這意味著您最多可以使用 AND 邏輯合併五個單獨的fieldName對象。
- 運算子最多可以有二十個containsAny值。
- in和notIn運算子最多可以有五個值。
- 每個字串最多可以有 256 個字元。
- 每個字符串比較都區分大小寫。
- 巢狀物件篩選允許最多五個巢狀層級的篩選。
- 每個最多filterGroup可以有 10 個filters。這意味著您可以filters使用 OR 邏輯最多組合 10 個。
  - in運算子是 OR 邏輯的特殊情況。在下面的例子中，有兩個filters：

```
{
    "filterGroup": [
        {
           "filters" : [
                 {
                    "fieldName" : "userId",
                    "operator" : "eq",
                    "value" : 1
                },
                {
                    "fieldName" : "group",
                    "operator" : "in",
                    "value" : ["Admin", "Developer"]
                }
           ]
        }
    ]
}
```

上一個篩選器群組的評估方式如下，並計入最大篩選器限制：

```
{
    "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"
                }
            ]
        }
    ]
}
```

$ 擴展無效訂閱（ ）invalidationJsonObject

用於從突變啟動訂閱失效。引數`invalidationJsonObject`如下所述。

> **Note**
>
> 此擴充功能只能用於突變解析器的回應對應範本。
> 您最多只能在任何單個請求中使用五個唯一
> 的`$extensions.invalidateSubscriptions()`方法調用。如果超過此限制，您將收到 GraphQL 錯誤。

引數： invalidationJsonObject

定`invalidationJsonObject`義以下內容：

- `subscriptionField`— 要使用的 GraphQL 結構描述訂閱使其無效。單一訂閱 (在中定義為字串) 會`subscriptionField`被視為無效驗證。
- `payload`— 鍵值配對清單，如果無效篩選器根據其值進行評估，則用作使訂閱無效的輸入。`true`

  下列範例會在`onUserDelete`訂閱解析程式中定義的無效驗證篩選器對值進行評估時，使用訂閱將已訂閱和連線的用戶端失效。`true` `payload`

```
$extensions.invalidateSubscriptions({
        "subscriptionField": "onUserDelete",
        "payload": {
                "group": "Developer"
                "type" : "Full-Time"
      }
    })
```

DynamoDB 的解析程式對應範本參考

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync DynamoDB 解析器可讓您使用 GraphQL 存放和擷取帳戶中現有 Amazon DynamoDB 表格中的資料。此解析器的運作方式是讓您將傳入的 GraphQL 請求對應至 DynamoDB 呼叫，然後將 DynamoDB 回應對應回應至 GraphQL。本節說明支援 DynamoDB 作業的對應範本。

GetItem

`GetItem`請求對應文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 發出`GetItem`請求，並可讓您指定：

- 項目在動態支援中的索引鍵
- 是否使用一致性讀取

`GetItem` 映射文件結構如下：

```
{
```

```
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true,
    "projection" : {
        ...
    }
}
```

欄位定義如下：

GetItem 欄位

GetItem 欄位清單

`version`

範本定義版本。目前支援 `2017-02-28` 和 `2018-05-29`。此值為必填。

`operation`

要執行 DynamoDB 支援作業。若要執行 `GetItem` DynamoDB 操作，這必須設為 `GetItem`。此值為必填。

`key`

項目在 DynamoDB 支援中的索引鍵。根據資料表結構，DynamoDB 項目可能具有單一雜湊鍵或雜湊鍵和排序索引鍵。如需如何指定「類型值」的詳細資訊，請參閱型別系統 (請求對應)。此值為必填。

`consistentRead`

是否要對 DynamoDB 執行強烈一致的讀取。此為選用，預設值為 `false`。

`projection`

用來指定要從 DynamoDB 作業傳回之屬性的投影。若要取得有關投影的更多資訊，請參閱投影。此欄位為選用欄位。

從 DynamoDB 傳回的項目會自動轉換為 GraphQL 和 JSON 原始類型，並可在對應內容 () 中使用。`$context.result`

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統 (回應對映)。

如需有關回應對映範本的詳細資訊，請參閱解析器對映範本概觀。

範例

下列範例是 GraphQL 查詢`getThing(foo: String!, bar: String!)`的對應範本：

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : $util.dynamodb.toDynamoDBJson($ctx.args.foo),
        "bar" : $util.dynamodb.toDynamoDBJson($ctx.args.bar)
    },
    "consistentRead" : true
}
```

如需 DynamoDB `GetItem` API 的詳細資訊，請參閱 DynamoDB API 文件。

PutItem

`PutItem`請求對應文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 發出`PutItem`請求，並可讓您指定下列項目：

- 項目在動態支援中的索引鍵
- 項目 (由 `key` 和 `attributeValues` 組成) 的完整內容
- 操作成功的條件

`PutItem` 映射文件結構如下：

```
{
    "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
}
```

欄位定義如下：

PutItem 欄位

PutItem 欄位清單

`version`

範本定義版本。目前支援 `2017-02-28` 和 `2018-05-29`。此值為必填。

`operation`

要執行 DynamoDB 支援作業。若要執行 `PutItem` DynamoDB 操作，這必須設為 `PutItem`。此值為必填。

`key`

項目在 DynamoDB 支援中的索引鍵。根據資料表結構，DynamoDB 項目可能具有單一雜湊鍵或雜湊鍵和排序索引鍵。如需如何指定「類型值」的詳細資訊，請參閱型別系統 (請求對應)。此值為必填。

`attributeValues`

將放入 DynamoDB 的項目其餘屬性。如需如何指定「類型值」的詳細資訊，請參閱型別系統 (請求對應)。此欄位為選用欄位。

`condition`

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，`PutItem` 要求會覆寫該項目的任何現有資料項目。如需有關條件的詳細資訊，請參閱條件運算式。此值是選用的。

`_version`

代表項目之最新已知版本的數值。此值是選用的。此欄位用於衝突偵測，而且僅支援已建立版本的資料來源。

`customPartitionKey`

啟用後，此字串值會在啟用版本控制時修改 delta 同步表所使用的`ds_sk`和`ds_pk`記錄的格式 (如需詳細資訊，請參閱AWS AppSync 開發人員指南中的衝突偵測與同步處理)。啟用時，也會啟用`populateIndexFields`項目的處理。此欄位為選用欄位。

`populateIndexFields`

布林值，當與啟用時，會為差異同步表中的每筆記錄建立新項目，特別是在`gsi_ds_pk`和`gsi_ds_sk`欄中。`customPartitionKey`如需詳細資訊，請參閱AWS AppSync 開發人員指南中的衝突偵測與同步處理。此欄位為選用欄位。

寫入 DynamoDB 的項目會自動轉換為 GraphQL 和 JSON 原始類型，並可在對應內容 () 中使用。`$context.result`

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統 (回應對映)。

如需有關回應對映範本的詳細資訊，請參閱解析器對映範本概觀。

範例 1

下列範例是 GraphQL 突變`updateThing(foo: String!, bar: String!, name: String!, version: Int!)`的對應範本。

若無指定索引鍵的項目，則會建立該項目。若已有指定索引鍵的項目，則會覆寫該項目。

```
{
    "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)
    }
}
```

範例 2

下列範例是 GraphQL 突變`updateThing(foo: String!, bar: String!, name: String!, expectedVersion: Int!)`的對應範本。

此範例會檢查以確定目前在 DynamoDB 中的項目已將version欄位設定為。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)
        }
    }
}
```

如需 DynamoDB PutItem API 的詳細資訊，請參閱 DynamoDB API 文件。

UpdateItem

UpdateItem請求對應文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 發出UpdateItem請求，並可讓您指定下列項目：

- 項目在動態支援中的索引鍵
- 說明如何在 DynamoDB 中更新項目的更新運算式
- 操作成功的條件

UpdateItem 映射文件結構如下：

```
{
    "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
}
```

欄位定義如下：

UpdateItem 欄位

UpdateItem 欄位清單

`version`

範本定義版本。目前支援 `2017-02-28` 和 `2018-05-29`。此值為必填。

`operation`

要執行 DynamoDB 支援作業。若要執行 `UpdateItem` DynamoDB 操作，這必須設為 `UpdateItem`。此值為必填。

`key`

項目在 DynamoDB 支援中的索引鍵。根據資料表結構，DynamoDB 項目可能具有單一雜湊鍵或雜湊鍵和排序索引鍵。如需有關指定「類型值」的詳細資訊，請參閱型別系統 (要求對應)。此值為必填。

`update`

本`update`節可讓您指定更新運算式，說明如何更新 DynamoDB 中的項目。如需如何撰寫更新運算式的詳細資訊，請參閱 DynamoDB UpdateExpressions 文件。此區段是必須的。

`update` 區段有三個元件：

expression

更新表達式。此值為必填。

expressionNames

表達式屬性 name 預留位置的替代，形式為鍵值組。該鍵對應於中使用的名稱預留位置expression，而且值必須是對應至 DynamoDB 中項目的屬性名稱的字串。此欄位為選用的，應只能填入用於 expression 中表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性 value 預留位置的替代，形式為鍵值組。鍵對應用於 expression 的值預留位置，值必須是類型值。如需如何指定「類型值」的詳細資訊，請參閱型別系統 (請求對應)。此必須指定。此欄位為選用的，應只能填入用於 expression 中表達式屬性值預留位置的替代。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，UpdateItem 要求會更新現有的資料項目，無論項目的目前狀態為何。如需有關條件的詳細資訊，請參閱條件運算式。此值是選用的。

_version

代表項目之最新已知版本的數值。此值是選用的。此欄位用於衝突偵測，而且僅支援已建立版本的資料來源。

customPartitionKey

啟用後，此字串值會在啟用版本控制時修改 delta 同步表所使用的ds_sk和ds_pk記錄的格式 (如需詳細資訊，請參閱AWS AppSync 開發人員指南中的衝突偵測與同步處理)。啟用時，也會啟用populateIndexFields項目的處理。此欄位為選用欄位。

populateIndexFields

布林值，當與啟用時，會為差異同步表中的每筆記錄建立新項目，特別是在gsi_ds_pk和gsi_ds_sk欄中。customPartitionKey如需詳細資訊，請參閱AWS AppSync 開發人員指南中的衝突偵測與同步處理。此欄位為選用欄位。

DynamoDB 中更新的項目會自動轉換為 GraphQL 和 JSON 原始類型，並可在對應內容 () 中使用。$context.result

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統 (回應對映)。

如需有關回應對映範本的詳細資訊，請參閱解析器對映範本概觀。

範例 1

下列範例是 GraphQL 突變`upvote(id: ID!)`的對應範本。

在此範例中，DynamoDB 中的項目`upvotes`和`version`欄位會以 1 遞增。

```
{
    "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 }
        }
    }
}
```

範例 2

下列範例是 GraphQL 突變`updateItem(id: ID!, title: String, author: String, expectedVersion: Int!)`的對應範本。

這個複雜的範例會檢查引數，並持續產生更新表達式，其只包含由用戶端提供的引數。例如，如果 `title` 和 `author` 遭到省略，則不會更新。如果指定了引數，但其值為`null`，則會從 DynamoDB 中的物件中刪除該欄位。最後，作業有一個條件，可驗證 DynamoDB 中目前的項目是否已將`version`欄位設定為：`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)
        }
    }
}
```

如需 DynamoDB UpdateItem API 的詳細資訊，請參閱 DynamoDB API 文件。

DeleteItem

DeleteItem請求對應文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 發出DeleteItem請求，並可讓您指定下列項目：

- 項目在動態支援中的索引鍵
- 操作成功的條件

DeleteItem 映射文件結構如下：

```
{
    "version" : "2018-05-29",
    "operation" : "DeleteItem",
    "customPartitionKey" : "foo",
    "populateIndexFields" : boolean value,
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "condition" : {
        ...
    },
    "_version" : 1
}
```

欄位定義如下：

DeleteItem 欄位

DeleteItem欄位清單

`version`

範本定義版本。目前支援 2017-02-28 和 2018-05-29。此值為必填。

operation

要執行 DynamoDB 支援作業。若要執行 DeleteItem DynamoDB 操作，這必須設為 DeleteItem。此值為必填。

key

項目在 DynamoDB 支援中的索引鍵。根據資料表結構，DynamoDB 項目可能具有單一雜湊鍵或雜湊鍵和排序索引鍵。如需有關指定「類型值」的詳細資訊，請參閱型別系統 (要求對應)。此值為必填。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，DeleteItem 要求會刪除項目，無論該項目的目前狀態為何。如需有關條件的詳細資訊，請參閱條件運算式。此值是選用的。

_version

代表項目之最新已知版本的數值。此值是選用的。此欄位用於衝突偵測，而且僅支援已建立版本的資料來源。

customPartitionKey

啟用後，此字串值會在啟用版本控制時修改 delta 同步表所使用的ds_sk和ds_pk記錄的格式 (如需詳細資訊，請參閱AWS AppSync 開發人員指南中的衝突偵測與同步處理)。啟用時，也會啟用populateIndexFields項目的處理。此欄位為選用欄位。

populateIndexFields

布林值，當與啟用時，會為差異同步表中的每筆記錄建立新項目，特別是在gsi_ds_pk和gsi_ds_sk欄中。customPartitionKey如需詳細資訊，請參閱AWS AppSync 開發人員指南中的衝突偵測與同步處理。此欄位為選用欄位。

從 DynamoDB 刪除的項目會自動轉換為 GraphQL 和 JSON 原始類型，並可在對應內容 () 中使用。$context.result

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統 (回應對映)。

如需有關回應對映範本的詳細資訊，請參閱解析器對映範本概觀。

範例 1

下列範例是 GraphQL 突變deleteItem(id: ID!)的對應範本。如果已存在此 ID 的項目，則該項目將會遭到刪除。

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

範例 2

下列範例是 GraphQL 突變`deleteItem(id: ID!, expectedVersion: Int!)`的對應範本。如果已存在此 ID 的項目，則該項目將會遭到刪除，但其 `version` 欄位必須已設為 `expectedVersion`：

```
{
    "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)
        }
    }
}
```

如需 DynamoDB `DeleteItem` API 的詳細資訊，請參閱 DynamoDB API 文件。

Query

`Query`請求對應文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 發出`Query`請求，並可讓您指定下列項目：

- 索引鍵表達式
- 要使用哪些索引
- 任何額外篩選條件
- 要傳回多少項目
- 是否使用一致性讀取

- 查詢方向 (向前或向後)
- 分頁字符

Query 映射文件結構如下：

```
{
    "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" : {
        ...
    }
}
```

欄位定義如下：

查詢欄位

查詢欄位清單

version

範本定義版本。目前支援 2017-02-28 和 2018-05-29。此值為必填。

operation

要執行 DynamoDB 支援作業。若要執行 `Query` DynamoDB 操作，這必須設為 `Query`。此值為必填。

query

此`query`區段可讓您指定索引鍵條件運算式，以說明要從 DynamoDB 擷取哪些項目。如需如何撰寫金鑰條件運算式的詳細資訊，請參閱 DynamoDB KeyConditions 文件。必須指定此區段。

expression

查詢表達式。必須指定此欄位。

expressionNames

表達式屬性 name 預留位置的替代，形式為鍵值組。該鍵對應於中使用的名稱預留位置`expression`，而且值必須是對應至 DynamoDB 中項目的屬性名稱的字串。此欄位為選用的，應只能填入用於 `expression` 中表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性 value 預留位置的替代，形式為鍵值組。鍵對應用於 `expression` 的值預留位置，值必須是類型值。如需如何指定「類型值」的詳細資訊，請參閱型別系統 (請求對應)。此值為必填。此欄位為選用的，應只能填入用於 `expression` 中表達式屬性值預留位置的替代。

filter

額外的篩選條件可用來在傳回 DynamoDB 的結果之前先篩選結果。如需篩選條件的詳細資訊，請參閱篩選條件。此欄位為選用欄位。

index

要查詢的索引名稱。DynamoDB 查詢作業可讓您掃描本機次要索引和全域次要索引，以及雜湊鍵的主索引索引。如果有指定，這會告知 DynamoDB 查詢指定的索引。若省略，則會查詢主索引鍵索引。

nextToken

分頁字符將繼續先前的查詢。這會是從先前查詢所取得的。此欄位為選用欄位。

limit

要評估的項目數上限 (不一定是相符的項目數)。此欄位為選用欄位。

scanIndexForward

指出是否向前或向後查詢的布林值。此欄位為選用，預設值為 `true`。

consistentRead

布林值，指出在查詢 DynamoDB 時是否使用一致讀取。此欄位為選用，預設值為 `false`。

select

根據預設，AWS AppSync DynamoDB 解析器只會傳回投影到索引中的屬性。如果需要更多屬性，請設定這個欄位。此欄位為選用欄位。支援的值是：

ALL_ATTRIBUTES

傳回所有指定資料表或索引的項目屬性。如果您查詢本機次要索引，DynamoDB 會針對索引中每個相符項目，從父資料表擷取整個項目。如果索引設定為投射所有項目屬性，所有資料都可從本機次要索引取得，不需進行任何擷取。

ALL_PROJECTED_ATTRIBUTES

只在查詢索引時才允許。擷取所有已投射到索引的屬性。如果索引設定為投射所有屬性，此傳回值相當於指定 `ALL_ATTRIBUTES`。

SPECIFIC_ATTRIBUTES

僅返回的中列出`projection`的屬性`expression`。此傳回值等同於指定`projection`的，`expression`而不指定任何值`Select`。

projection

用來指定要從 DynamoDB 作業傳回之屬性的投影。若要取得有關投影的更多資訊，請參閱投影。此欄位為選用欄位。

DynamoDB 的結果會自動轉換為 GraphQL 和 JSON 原始類型，並可在對應內容 () 中使用。`$context.result`

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統 (回應對映)。

如需有關回應對映範本的詳細資訊，請參閱解析器對映範本概觀。

結果的結構如下：

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
```

```
}
```

欄位定義如下：

`items`

包含 DynamoDB 查詢傳回之項目的清單。

`nextToken`

如果可能有一個以上的結果，`nextToken` 會包含可用於另一個要求的分頁字符。請注意，會AWS AppSync 加密並混淆 DynamoDB 傳回的分頁權杖。這樣可確保資料表的資料不會不慎洩漏給發起人。另請注意，這些分頁字符在不同解析程式之間無法使用。

`scannedCount`

套用篩選條件表達式 (若有的話) 以前符合查詢條件表達式的項目數。

範例

下列範例是 GraphQL 查詢`getPosts(owner: ID!)`的對應範本。

在此範例中，資料表上的全域次要索引受到查詢，以傳回指定 ID 擁有的所有文章。

```
{
    "version" : "2017-02-28",
    "operation" : "Query",
    "query" : {
        "expression" : "ownerId = :ownerId",
        "expressionValues" : {
            ":ownerId" : $util.dynamodb.toDynamoDBJson($context.arguments.owner)
        }
    },
    "index" : "owner-index"
}
```

如需 DynamoDB `Query` API 的詳細資訊，請參閱 DynamoDB API 文件。

Scan

`Scan`請求對應文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 發出`Scan`請求，並可讓您指定下列項目：

- 排除結果的篩選結果
- 要使用哪些索引
- 要傳回多少項目
- 是否使用一致性讀取
- 分頁字符
- 平行掃描

Scan 映射文件結構如下：

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "index" : "fooIndex",
    "limit" : 10,
    "consistentRead" : false,
    "nextToken" : "aPaginationToken",
    "totalSegments" : 10,
    "segment" : 1,
    "filter" : {
        ...
    },
    "projection" : {
        ...
    }
}
```

欄位定義如下：

掃描欄位

掃描欄位清單

version

範本定義版本。目前支援 2017-02-28 和 2018-05-29。此值為必填。

operation

要執行 DynamoDB 支援作業。若要執行 Scan DynamoDB 操作，這必須設為 Scan。此值為必填。

filter

可用於在傳回 DynamoDB 結果之前篩選結果的篩選器。如需篩選條件的詳細資訊，請參閱篩選條件。此欄位為選用欄位。

index

要查詢的索引名稱。DynamoDB 查詢作業可讓您掃描本機次要索引和全域次要索引，以及雜湊鍵的主索引索引。如果有指定，這會告知 DynamoDB 查詢指定的索引。若省略，則會查詢主索引鍵索引。

limit

單次可評估的項目數量上限。此欄位為選用欄位。

consistentRead

布林值，指出在查詢 DynamoDB 時是否使用一致讀取。此欄位為選用，預設值為 `false`。

nextToken

分頁字符將繼續先前的查詢。這會是從先前查詢所取得的。此欄位為選用欄位。

select

根據預設，AWS AppSync DynamoDB 解析器只會傳回任何投影到索引中的屬性。如果需要更多屬性，則此欄位可以設定。此欄位為選用欄位。支援的值是：

ALL_ATTRIBUTES

傳回所有指定資料表或索引的項目屬性。如果您查詢本機次要索引，DynamoDB 會針對索引中每個相符項目，從父資料表擷取整個項目。如果索引設定為投射所有項目屬性，所有資料都可從本機次要索引取得，不需進行任何擷取。

ALL_PROJECTED_ATTRIBUTES

只在查詢索引時才允許。擷取所有已投射到索引的屬性。如果索引設定為投射所有屬性，此傳回值相當於指定 `ALL_ATTRIBUTES`。

SPECIFIC_ATTRIBUTES

僅返回的中列出`projection`的屬性`expression`。此傳回值等同於指定`projection`的，`expression`而不指定任何值`Select`。

totalSegments

執行平行掃描時分割資料表的區段數。此欄位為選用的，但若指定 `segment`，則此欄位必須指定。

segment

此操作中執行平行掃描時的資料表區段。此欄位為選用的，但若指定 `totalSegments`，則此欄位必須指定。

projection

用來指定要從 DynamoDB 作業傳回之屬性的投影。若要取得有關投影的更多資訊，請參閱投影。此欄位為選用欄位。

DynamoDB 掃描傳回的結果會自動轉換為 GraphQL 和 JSON 原始類型，並可在對應內容 () 中使用。`$context.result`

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統 (回應對映)。

如需有關回應對映範本的詳細資訊，請參閱解析器對映範本概觀。

結果的結構如下：

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10
}
```

欄位定義如下：

items

包含 DynamoDB 掃描傳回之項目的清單。

nextToken

如果可能有一個以上的結果，`nextToken` 會包含可用於另一個要求的分頁字符。AWS AppSync 加密和混淆從 DynamoDB 傳回的分頁權杖。這樣可確保資料表的資料不會不慎洩漏給發起人。此外，這些分頁字符在不同解析程式之間無法使用。

scannedCount

DynamoDB 在套用篩選器運算式 (如果存在) 之前擷取的項目數。

範例 1

下列範例是 GraphQL 查詢的對應範本：`allPosts`

在此範例中，資料表中的所有項目都會傳回。

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

範例 2

下列範例是 GraphQL 查詢的對應範本：`postsMatching(title: String!)`

在此範例中，資料表中開頭為 `title` 引數的所有項目都會傳回。

```
{
    "version" : "2017-02-28",
    "operation" : "Scan",
    "filter" : {
        "expression" : "begins_with(title, :title)",
        "expressionValues" : {
            ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
        },
    }
}
```

如需 DynamoDB `Scan` API 的詳細資訊，請參閱 DynamoDB API 文件。

Sync

Sync請求對應文件可讓您從 DynamoDB 表擷取所有結果，然後僅接收自上次查詢後變更的資料 (差異更新)。 Sync請求只能對已建立版本化的 DynamoDB 資料來源發出。您可以指定下列選項：

- 排除結果的篩選結果
- 要傳回多少項目
- 分頁字符
- 上次起始 Sync 操作時

Sync 映射文件結構如下：

```
{
    "version" : "2018-05-29",
    "operation" : "Sync",
    "basePartitionKey": "Base Tables PartitionKey",
    "deltaIndexName": "delta-index-name",
    "limit" : 10,
    "nextToken" : "aPaginationToken",
    "lastSync" :  1550000000000,
    "filter" : {
        ...
    }
}
```

欄位定義如下：

同步欄位

同步欄位清單

`version`

範本定義的版本。目前僅支援 2018-05-29。此值為必填。

`operation`

要執行 DynamoDB 支援作業。若要執行 Sync 操作，這必須設定為 Sync。此值為必填。

`filter`

可用於在傳回 DynamoDB 結果之前篩選結果的篩選器。如需篩選條件的詳細資訊，請參閱篩選條件。此欄位為選用欄位。

`limit`

單次可評估的項目數量上限。此欄位為選用欄位。如果省略此值，預設限制將設為 100 個項目。此欄位的最大值為 1000 個項目。

`nextToken`

分頁字符將繼續先前的查詢。這會是從先前查詢所取得的。此欄位為選用欄位。

lastSync

上次成功啟動 Sync 操作的時間 (以毫秒為單位)。如果指定此值，只會傳回 `lastSync` 之後變更的項目。這個欄位是選用的，而且只有在初始 Sync 操作擷取所有頁面之後才能填入。如果省略此值，將傳回 Base 資料表的結果，否則會傳回 Delta 資料表的結果。

basePartitionKey

執行Sync作業時所使用的 Base 資料表的資料分割索引鍵。此欄位允許在表格使用自訂分割區索引鍵時執行Sync作業。此為選用欄位。

deltaIndexName

用於Sync作業的索引。當資料表使用自訂資料分割索引鍵時，需要此索引才能在整個 delta 儲存資料表上啟用Sync作業。該Sync操作將在 GSI 上執行（在`gsi_ds_pk`和上創建`gsi_ds_sk`）。此欄位為選用欄位。

DynamoDB 同步傳回的結果會自動轉換為 GraphQL 和 JSON 原始類型，並可在對應內容 () 中使用。`$context.result`

如需 DynamoDB 類型轉換的詳細資訊，請參閱類型系統 (回應對映)。

如需有關回應對映範本的詳細資訊，請參閱解析器對映範本概觀。

結果的結構如下：

```
{
    items = [ ... ],
    nextToken = "a pagination token",
    scannedCount = 10,
    startedAt = 1550000000000
}
```

欄位定義如下：

items

包含同步傳回項目的清單。

nextToken

如果可能有一個以上的結果，`nextToken` 會包含可用於另一個要求的分頁字符。AWS AppSync 加密和混淆從 DynamoDB 傳回的分頁權杖。這樣可確保資料表的資料不會不慎洩漏給發起人。此外，這些分頁字符在不同解析程式之間無法使用。

scannedCount

DynamoDB 在套用篩選器運算式 (如果存在) 之前擷取的項目數。

startedAt

開始同步操作時，可以在本機存放並在另一個請求中做為 lastSync 引數的時間 (以 epoch 毫秒為單位)。如果請求中包含分頁字符，則該值將與請求針對第一頁結果傳回的值相同。

範例 1

下列範例是 GraphQL 查詢的對應範本：syncPosts(nextToken: String, lastSync: AWSTimestamp)

在此範例中，如果省略 lastSync，則會傳回 Base 資料表中的所有項目。如果提供 lastSync，只會傳回自 lastSync 變更之 Delta Sync 資料表中的項目。

```
{
    "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

BatchGetItem請求對應文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 發出BatchGetItem請求，以擷取可能跨多個表格的多個項目。使用此要求範本時，您必須指定下列項目：

- 將從其中擷取項目的資料表名稱
- 將從個別資料表擷取項目的索引鍵

此時會套用 DynamoDB BatchGetItem 限制，且可能不會提供任何條件表達式。

BatchGetItem 映射文件結構如下：

```
{
```

```
    "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" : {
                ...
            }
        }
    }
}
```

欄位定義如下：

BatchGetItem 欄位

BatchGetItem欄位清單

`version`

範本定義的版本。僅支援 `2018-05-29`。此值為必填。

`operation`

要執行 DynamoDB 支援作業。若要執行 `BatchGetItem` DynamoDB 操作，這必須設為 `BatchGetItem`。此值為必填。

`tables`

要從中擷取項目的 DynamoDB 資料表。此值是其中將資料表名稱指定為該映射索引鍵的映射。至少必須提供一個資料表。`tables` 值為必填。

`keys`

表示要擷取之項目之主索引鍵的 DynamoDB 索引鍵清單。根據資料表結構，DynamoDB 項目可能具有單一雜湊鍵或雜湊鍵和排序索引鍵。如需如何指定「類型值」的詳細資訊，請參閱型別系統 (請求對應)。

`consistentRead`

是否在執行GetItem操作時使用一致的讀取。此值為選用值，且預設值為 false。

`projection`

用來指定要從 DynamoDB 作業傳回之屬性的投影。若要取得有關投影的更多資訊，請參閱投影。此欄位為選用欄位。

注意事項：

- 如果未從資料表中擷取任何項目，該資料表的 data 區塊中會顯示 null 元素。
- 呼叫結果會根據要求對應範本內提供的順序，依據資料表排序。
- a 中的每個`Get`命令都`BatchGetItem`是原子的，但是，批處理可以部分處理。如果因錯誤而部分批次處理，則未處理的索引鍵會透過 unprocessedKeys 區塊傳回為部分的呼叫結果。
- `BatchGetItem` 限制為 100 個索引鍵。

使用下列範例要求映射範本時：

```
{
  "version": "2018-05-29",
  "operation": "BatchGetItem",
  "tables": {
    "authors": [
        {
          "author_id": {
            "S": "a1"
          }
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          }
        }
    ],
  }
}
```

可透過 $ctx.result 提供的呼叫結果如下所示：

```
{
   "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` 包含錯誤的詳細資訊。在要求映射範本中所提供的索引鍵 data、unprocessedKeys 及個別資料表索引鍵，都確定會出現在呼叫結果中。已刪除的項目會出現在 data 區塊中。尚未處理的項目在該 data 區塊中會標示為 null，並置於 unprocessedKeys 區塊。

如需更完整的範例，請遵循 DynamoDB Batch 教學課程的教學課程：DynamoDB 批次解析器。AppSync

BatchDeleteItem

`BatchDeleteItem`請求對應文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 發出`BatchWriteItem`請求，以刪除可能跨多個表格的多個項目。使用此要求範本時，您必須指定下列項目：

- 將從其中刪除項目的資料表名稱
- 將從個別資料表刪除項目的索引鍵

此時會套用 DynamoDB `BatchWriteItem` 限制，且可能不會提供任何條件表達式。

`BatchDeleteItem` 映射文件結構如下：

```
{
    "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
        }],
    }
}
```

欄位定義如下：

BatchDeleteItem 欄位

BatchDeleteItem欄位清單

`version`

範本定義的版本。僅支援 `2018-05-29`。此值為必填。

`operation`

要執行 DynamoDB 支援作業。若要執行 `BatchDeleteItem` DynamoDB 操作，這必須設為 `BatchDeleteItem`。此值為必填。

`tables`

要從中刪除項目的 DynamoDB 表。每個資料表都是 DynamoDB 索引鍵清單，代表要刪除之項目的主索引鍵。根據資料表結構，DynamoDB 項目可能具有單一雜湊鍵或雜湊鍵和排序索引鍵。如需如何指定「類型值」的詳細資訊，請參閱型別系統 (請求對應)。至少必須提供一個資料表。該`tables`值是必需的。

注意事項：

- 不同於 `DeleteItem` 操作，回應中未傳回完整刪除的項目。只會傳回已傳遞的索引鍵。
- 如果資料表中未刪除任何項目，該資料表的 data 區塊中就會顯示 null 元素。

- 呼叫結果會根據要求對應範本內提供的順序，依據資料表排序。
- a 中的每個Delete命令都BatchDeleteItem是原子的。但是，批次可以部分處理。如果因錯誤而部分批次處理，則未處理的索引鍵會透過 unprocessedKeys 區塊傳回為部分的呼叫結果。
- BatchDeleteItem 限制為 25 個索引鍵。

使用下列範例要求映射範本時：

```
{
  "version": "2018-05-29",
  "operation": "BatchDeleteItem",
  "tables": {
    "authors": [
        {
          "author_id": {
            "S": "a1"
          }
        },
    ],
    "posts": [
        {
          "author_id": {
            "S": "a1"
          },
          "post_id": {
            "S": "p2"
          }
        }
    ],
  }
}
```

可透過 $ctx.result 提供的呼叫結果如下所示：

```
{
   "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` 包含錯誤的詳細資訊。在要求映射範本中所提供的索引鍵 data、unprocessedKeys 及個別資料表索引鍵，都確定會出現在呼叫結果中。已刪除的項目會出現在 data 區塊中。尚未處理的項目在該 data 區塊中會標示為 null，並置於 unprocessedKeys 區塊。

如需更完整的範例，請遵循 DynamoDB Batch 教學課程的教學課程：DynamoDB 批次解析器。AppSync

BatchPutItem

`BatchPutItem`請求對應文件可讓您告知 DynamoDB 解析程式向 AWS AppSync DynamoDB 發出`BatchWriteItem`請求，以便放置多個項目，這些項目可能會跨越多個表格。使用此要求範本時，您必須指定下列項目：

- 項目將放入其中的資料表名稱
- 將放入每個資料表的所有項目

此時會套用 DynamoDB `BatchWriteItem` 限制，且可能不會提供任何條件表達式。

`BatchPutItem` 映射文件結構如下：

```
{
    "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
        }],
    }
}
```

欄位定義如下：

BatchPutItem 欄位

BatchPutItem欄位清單

`version`

範本定義的版本。僅支援 2018-05-29。此值為必填。

`operation`

要執行 DynamoDB 支援作業。若要執行 `BatchPutItem` DynamoDB 操作，這必須設為 `BatchPutItem`。此值為必填。

`tables`

要放置項目的 DynamoDB 資料表。每個表格項目代表要為此特定表格插入的 DynamoDB 項目清單。至少必須提供一個資料表。此值為必填。

注意事項：

- 若插入成功，回應會傳回完全插入的項目。
- 如果資料表中並未插入任何項目，該資料表的 data 區塊中就會顯示 null 元素。
- 插入的項目會根據要求對應範本內提供的順序，依據資料表排序。
- a 中的每個Put命令都BatchPutItem是原子的，但是，批處理可以部分處理。如果因錯誤而部分批次處理，則未處理的索引鍵會透過 unprocessedKeys 區塊傳回為部分的呼叫結果。
- BatchPutItem 限制為 25 個項目。

使用下列範例要求映射範本時：

```
{
  "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"
          }
        }
    ],
  }
}
```

可透過 $ctx.result 提供的呼叫結果如下所示：

```
{
   "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` 包含錯誤的詳細資訊。在要求映射範本中所提供的索引鍵 data、unprocessedItems 及個別資料表索引鍵，都確定會出現在呼叫結果中。已插入的項目會出現在 data 區塊中。尚未處理的項目在該 data 區塊中會標示為 null，並置於 unprocessedItems 區塊。

如需更完整的範例，請遵循 DynamoDB Batch 教學課程的教學課程：DynamoDB 批次解析器。AppSync

TransactGetItems

`TransactGetItems`請求對應文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 發出`TransactGetItems`請求，以擷取可能跨多個表格的多個項目。使用此要求範本時，您必須指定下列項目：

- 從中擷取項目之每個請求項目的資料表名稱
- 從每個資料表擷取之每個請求項目的索引鍵

此時會套用 DynamoDB `TransactGetItems` 限制，且可能不會提供任何條件表達式。

`TransactGetItems` 映射文件結構如下：

```
{
    "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" : {
                ...
           }
       }
    ]
}
```

欄位定義如下：

TransactGetItems 欄位

TransactGetItems欄位清單

`version`

範本定義的版本。僅支援 `2018-05-29`。此值為必填。

operation

要執行 DynamoDB 支援作業。若要執行 TransactGetItems DynamoDB 操作，這必須設為 TransactGetItems。此值為必填。

transactItems

要包含的請求項目。此值是請求項目的陣列。必須提供至少一個請求項目。transactItems 值為必填。

table

要從中擷取項目的 DynamoDB 資料表。此值是資料表名稱的字串。table 值為必填。

key

代表要擷取之項目的主索引鍵的 DynamoDB 索引鍵。根據資料表結構，DynamoDB 項目可能具有單一雜湊鍵或雜湊鍵和排序索引鍵。如需如何指定「類型值」的詳細資訊，請參閱型別系統 (請求對應)。

projection

用來指定要從 DynamoDB 作業傳回之屬性的投影。若要取得有關投影的更多資訊，請參閱投影。此欄位為選用欄位。

注意事項：

- 如果交易成功，items 區塊中擷取項目的順序會與請求項目的順序相同。
- 交易以某 all-or-nothing 種方式執行。如果任何請求項目造成錯誤，將不執行整個交易，而且將傳回錯誤詳細資料。
- 無法擷取的請求項目不是錯誤。相反地，null 元素會出現在對應位置的項目區塊。
- 如果事務的錯誤是 TransactionCanceledException，cancellationReasons塊將被填充。cancellationReasons 區塊中取消原因的順序將與請求項目的順序相同。
- 將 TransactGetItems 限制為 25 個請求項目。

使用下列範例要求映射範本時：

```
{
    "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
                }
            }
        }
    ]
}
```

如果交易成功，而且僅擷取第一個請求項目，則 `$ctx.result` 中的可用叫用結果如下所示：

```
{
    "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
}
```

如果交易失敗`$ctx.result`是TransactionCanceledException因為第一個要求項目所造成，可用的呼叫結果如下：

```
{
    "items": null,
    "cancellationReasons": [
        {
```

```
            "type":"Sample error type",
            "message":"Sample error message"
        },
        {
            "type":"None",
            "message":"None"
        }
    ]
}
```

`$ctx.error` 包含錯誤的詳細資訊。索引鍵 items 和 cancellationReasons 保證出現在 `$ctx.result` 中。

如需更完整的範例，請按照 AppSync 此處的 DynamoDB 交易教學課程進行操作：DynamoDB 交易解析器。

TransactWriteItems

`TransactWriteItems`請求對應文件可讓您告知 AWS AppSync DynamoDB 解析程式向 DynamoDB 發出`TransactWriteItems`請求，以便將多個項目寫入多個資料表。使用此要求範本時，您必須指定下列項目：

- 每個請求項目的目標資料表名稱
- 每個請求項目要執行的操作。支援的作業類型有四種：PutItemUpdateItem、DeleteItem、和 ConditionCheck
- 每個要寫入之請求項目的索引鍵

套用 DynamoDB `TransactWriteItems` 限制。

`TransactWriteItems` 映射文件結構如下：

```
{
    "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 欄位

TransactWriteItems欄位清單

欄位定義如下：

`version`

範本定義的版本。僅支援 `2018-05-29`。此值為必填。

`operation`

要執行 DynamoDB 支援作業。若要執行 `TransactWriteItems` DynamoDB 操作，這必須設為 `TransactWriteItems`。此值為必填。

`transactItems`

要包含的請求項目。此值是請求項目的陣列。必須提供至少一個請求項目。`transactItems` 值為必填。

對於 `PutItem`，欄位定義如下：

`table`

目的地 DynamoDB 資料表。此值是資料表名稱的字串。`table` 值為必填。

`operation`

要執行 DynamoDB 支援作業。若要執行 `PutItem` DynamoDB 操作，這必須設為 `PutItem`。此值為必填。

`key`

代表要放置之項目的主索引鍵的 DynamoDB 索引鍵。根據資料表結構，DynamoDB 項目可能具有單一雜湊鍵或雜湊鍵和排序索引鍵。如需如何指定「類型值」的詳細資訊，請參閱型別系統 (請求對應)。此值為必填。

`attributeValues`

將放入 DynamoDB 的項目其餘屬性。如需如何指定「類型值」的詳細資訊，請參閱型別系統 (請求對應)。此欄位為選用欄位。

`condition`

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，`PutItem` 要求會覆寫該項目的任何現有資料項目。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱交易條件運算式。此值是選用的。

對於 UpdateItem，欄位定義如下：

table

要更新 DynamoDB 資料表。此值是資料表名稱的字串。table 值為必填。

operation

要執行 DynamoDB 支援作業。若要執行 UpdateItem DynamoDB 操作，這必須設為 UpdateItem。此值為必填。

key

代表要更新之項目的主索引鍵的 DynamoDB 索引鍵。根據資料表結構，DynamoDB 項目可能具有單一雜湊鍵或雜湊鍵和排序索引鍵。如需如何指定「類型值」的詳細資訊，請參閱型別系統 (請求對應)。此值為必填。

update

本update節可讓您指定更新運算式，說明如何更新 DynamoDB 中的項目。如需如何撰寫更新運算式的詳細資訊，請參閱 DynamoDB UpdateExpressions 文件。此區段是必須的。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，UpdateItem 要求會更新現有的資料項目，無論項目的目前狀態為何。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱交易條件運算式。此值是選用的。

對於 DeleteItem，欄位定義如下：

table

要在其中刪除項目的 DynamoDB 資料表。此值是資料表名稱的字串。table 值為必填。

operation

要執行 DynamoDB 支援作業。若要執行 DeleteItem DynamoDB 操作，這必須設為 DeleteItem。此值為必填。

key

代表要刪除之項目的主索引鍵的 DynamoDB 鍵。根據資料表結構，DynamoDB 項目可能具有單一雜湊鍵或雜湊鍵和排序索引鍵。如需如何指定「類型值」的詳細資訊，請參閱型別系統 (請求對應)。此值為必填。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。如果沒有指定條件，DeleteItem 要求會刪除項目，無論該項目的目前狀態為何。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱交易條件運算式。此值是選用的。

對於 ConditionCheck，欄位定義如下：

table

要在其中檢查條件的 DynamoDB 表格。此值是資料表名稱的字串。table 值為必填。

operation

要執行 DynamoDB 支援作業。若要執行 ConditionCheck DynamoDB 操作，這必須設為 ConditionCheck。此值為必填。

key

DynamoDB 鍵，代表要進行條件檢查之項目的主索引鍵。根據資料表結構，DynamoDB 項目可能具有單一雜湊鍵或雜湊鍵和排序索引鍵。如需如何指定「類型值」的詳細資訊，請參閱型別系統 (請求對應)。此值為必填。

condition

決定要求是否成功的條件，可根據已存在於 DynamoDB 的物件狀態。您可以指定在條件檢查失敗時是否擷取現有項目。如需交易條件的詳細資訊，請參閱交易條件運算式。此值為必填。

注意事項：

- 如果成功，只會在回應中傳回請求項目的索引鍵。索引鍵的順序將與請求項目的順序相同。
- 交易以某 all-or-nothing 種方式執行。如果任何請求項目造成錯誤，將不執行整個交易，而且將傳回錯誤詳細資料。
- 沒有兩個請求項目可以定位到相同項目。否則，他們會導致TransactionCanceledException錯誤。
- 如果事務的錯誤是 TransactionCanceledException，cancellationReasons塊將被填充。如果請求項目的條件檢查失敗，而且您未將 returnValuesOnConditionCheckFailure 指定為 false，將擷取資料表中存在的項目，並存放在 cancellationReasons 區塊之對應位置的 item 中。
- 將 TransactWriteItems 限制為 25 個請求項目。

使用下列範例要求映射範本時：

```
{
    "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"
```

```
                    }
                }
            },
        }
    ]
}
```

如果交易成功，$ctx.result 中可用的叫用結果如下所示：

```
{
    "keys": [
       // Key of the PutItem request
       {
           "post_id": "p1",
       },
       // Key of the UpdateItem request
       {
           "author_id": "a1"
       }
    ],
    "cancellationReasons": null
}
```

如果交易因為PutItem要求的條件檢查失敗而失敗，則中$ctx.result可用的呼叫結果如下：

```
{
    "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"
       }
    ]
```

```
}
```

`$ctx.error` 包含錯誤的詳細資訊。索引鍵 keys 和 cancellationReasons 保證出現在 `$ctx.result` 中。

如需更完整的範例，請按照 AppSync 此處的 DynamoDB 交易教學課程進行操作：DynamoDB 交易解析器。

類型系統 (請求對應)

使用 AWS AppSync DynamoDB 解析器呼叫 DynamoDB 表時，AWS AppSync 需要知道該呼叫中要使用的每個值的類型。這是因為 DynamoDB 支援的型別原語比 GraphQL 或 JSON 更多 (例如集合和二進位資料)。AWS AppSync 在 GraphQL 和 DynamoDB 之間進行轉換時需要一些提示，否則必須對資料表中的結構方式做出一些假設。

如需 DynamoDB 資料類型的詳細資訊，請參閱 DynamoDB 資料類型描述元和資料類型說明文件。

DynamoDB 值由包含單一索引鍵值組的 JSON 物件表示。此索引鍵會指定 DynamoDB 類型，而值則會指定值本身。在下列範例中，`S` 鍵代表值是字串，`identifier` 值則是字串值本身。

```
{ "S" : "identifier" }
```

請注意，JSON 物件不能有一個以上的索引值對。如果指定一個以上的索引值對，要求映射文件就不會進行剖析。

DynamoDB 值可在您需要指定值的要求對應文件中的任何位置使用。您需要執行這項操作的一些地方包括：`key` 和 `attributeValue` 區段，以及表達式區段的 `expressionValues` 區段。在下列範例中，DynamoDB 字串值`identifier`已指派給`key`區段中的`id`欄位 (可能在`GetItem`請求對應文件中)。

```
"key" : {
   "id" : { "S" : "identifier" }
}
```

支援的類型

AWS AppSync 支援下列 DynamoDB 純量、文件和集合類型：

字串類型 S

單一字串值。一 DynamoDB 符串值表示為：

```
{ "S" : "some string" }
```

使用範例：

```
"key" : {
   "id" : { "S" : "some string" }
}
```

字串集類型 SS

字串值的集合。DynamoDB 字串集的值由下列方式表示：

```
{ "SS" : [ "first value", "second value", ... ] }
```

使用範例：

```
"attributeValues" : {
   "phoneNumbers" : { "SS" : [ "+1 555 123 4567", "+1 555 234 5678" ] }
}
```

數字類型 N

單一數值。動 DynamoDB 編號值的表示方式為：

```
{ "N" : 1234 }
```

使用範例：

```
"expressionValues" : {
   ":expectedVersion" : { "N" : 1 }
}
```

數字集類型 NS

數值的集合。動 DynamoDB 數字集的值由下列方式表示：

```
{ "NS" : [ 1, 2.3, 4 ... ] }
```

使用範例：

```
"attributeValues" : {
```

```
   "sensorReadings" : { "NS" : [ 67.8, 12.2, 70 ] }
}
```

二進位類型 **B**

二進位值。DynamoDB 進位值的表示方式為：

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

請注意，該值實際上是一個字符串，其中字符串是二進制數據的 base64 編碼表示。AWS AppSync 在將此字串傳送至 DynamoDB 之前，將此字串解碼回其二進位值。AWS AppSync 使用 RFC 2045 所定義的 base64 解碼方案：忽略不在 base64 字母中的任何字元。

使用範例：

```
"attributeValues" : {
   "binaryMessage" : { "B" : "SGVsbG8sIFdvcmxkIQo=" }
}
```

二進位集類型 **BS**

二進位值的集合。DynamoDB 進位集值的表示方式為：

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

請注意，該值實際上是一個字符串，其中字符串是二進制數據的 base64 編碼表示。AWS AppSync 在將此字串傳送至 DynamoDB 之前，將此字串解碼回其二進位值。AWS AppSync 使用 RFC 2045 所定義的 base64 解碼方案：忽略不在 base64 字母中的任何字元。

使用範例：

```
"attributeValues" : {
   "binaryMessages" : { "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ] }
}
```

布林類型 **BOOL**

布林值。一 DynamoDB 林值表示為：

```
{ "BOOL" : true }
```

請注意，只有 `true` 和 `false` 為有效值。

使用範例：

```
"attributeValues" : {
   "orderComplete" : { "BOOL" : false }
}
```

清單類型 L

任何其他受支援的 DynamoDB 值的清單。動 DynamoDB 單值的表示方式為：

```
{ "L" : [ ... ] }
```

請注意，該值是複合值，其中清單可以包含零個或多個任何受支援的 DynamoDB 值 (包括其他清單)。清單也可包含混合的不同類型。

使用範例：

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

映射類型 M

代表其他受支援 DynamoDB 值之索引鍵值組的無序集合。一 DynamoDB 圖值表示為：

```
{ "M" : { ... } }
```

請注意，映射可包含零或多個索引值對。索引鍵必須是字串，且該值可以是任何支援的 DynamoDB 值 (包括其他對應)。映射也可包含混合的不同類型。

使用範例：

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
```

```
    }
}
```

Null 類型 **NULL**

null 值。DynamoDB 空值表示為：

```
{ "NULL" : null }
```

使用範例：

```
"attributeValues" : {
   "phoneNumbers" : { "NULL" : null }
}
```

如需每種類型的詳細資訊，請參閱 DynamoDB 文件。

類型系統（響應映射）

從 DynamoDB 收到回應時，AWS AppSync 會自動將其轉換為 GraphQL 和 JSON 原始類型。DynamoDB 中的每個屬性都會在回應對應內容中解碼並傳回。

例如，如果 DynamoDB 傳回下列內容：

```
{
    "id" : { "S" : "1234" },
    "name" : { "S" : "Nadia" },
    "age" : { "N" : 25 }
}
```

然後，AWS AppSync DynamoDB 解析器將其轉換為 GraphQL 和 JSON 類型，如下所示：

```
{
    "id" : "1234",
    "name" : "Nadia",
    "age" : 25
}
```

本節說明 AWS AppSync 如何轉換下列的 DynamoDB 純量、文件和集合類型：

字串類型 S

單一字串值。DynamoDB 字串值會以字串形式傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 字串值：

```
{ "S" : "some string" }
```

AWS AppSync 它轉換為一個字符串：

```
"some string"
```

字串集類型 SS

字串值的集合。DynamoDB 字串集的值會以字串清單的形式傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 字串集值：

```
{ "SS" : [ "first value", "second value", ... ] }
```

AWS AppSync 將其轉換為字符串列表：

```
[ "+1 555 123 4567", "+1 555 234 5678" ]
```

數字類型 N

單一數值。DynamoDB 編號值會以數字形式傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 資料庫編號值：

```
{ "N" : 1234 }
```

AWS AppSync 將其轉換為一個數字：

```
1234
```

數字集類型 NS

數值的集合。DynamoDB 數字集的值會以數字清單的形式傳回。

例如，如果 DynamoDB 傳回下列數字集值：

```
{ "NS" : [ 67.8, 12.2, 70 ] }
```

AWS AppSync 將其轉換為數字列表：

```
[ 67.8, 12.2, 70 ]
```

二進位類型 **B**

二進位值。DynamoDB 二進位值會以包含該值 base64 表示法的字串形式傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 進位值：

```
{ "B" : "SGVsbG8sIFdvcmxkIQo=" }
```

AWS AppSync 將其轉換為包含值 base64 表示法的字串：

```
"SGVsbG8sIFdvcmxkIQo="
```

請注意，二進位資料以 base64 編碼配置編碼，如 RFC 4648 和 RFC 2045 中所定義。

二進位集類型 **BS**

二進位值的集合。DynamoDB 二進位集值會傳回為包含值 base64 表示法的字串清單。

例如，如果 DynamoDB 傳回下列 DynamoDB 進位集值：

```
{ "BS" : [ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ] }
```

AWS AppSync 將其轉換為包含值 base64 表示的字符串列表：

```
[ "SGVsbG8sIFdvcmxkIQo=", "SG93IGFyZSB5b3U/Cg==" ... ]
```

請注意，二進位資料以 base64 編碼配置編碼，如 RFC 4648 和 RFC 2045 中所定義。

布林類型 **BOOL**

布林值。DynamoDB 布林值會以布林值的形式傳回。

例如，如果 DynamoDB 傳回下列 DynamoDB 林值：

```
{ "BOOL" : true }
```

AWS AppSync 將其轉換為布爾值：

```
true
```

清單類型 L

任何其他受支援的 DynamoDB 值的清單。DynamoDB 清單值會以值清單的形式傳回，其中每個內部值也會轉換。

例如，如果 DynamoDB 傳回下列 DynamoDB 清單值：

```
{ "L" : [
      { "S"  : "A string value" },
      { "N"  : 1 },
      { "SS" : [ "Another string value", "Even more string values!" ] }
   ]
}
```

AWS AppSync 將其轉換為轉換值的列表：

```
[ "A string value", 1, [ "Another string value", "Even more string values!" ] ]
```

映射類型 M

任何其他受支援的 DynamoDB 值的索引鍵/值集合。DynamoDB 對應值會以 JSON 物件的形式傳回，其中每個索引鍵/值也會轉換。

例如，如果 DynamoDB 傳回下列 DynamoDB 對應值：

```
{ "M" : {
      "someString" : { "S"  : "A string value" },
      "someNumber" : { "N"  : 1 },
      "stringSet"  : { "SS" : [ "Another string value", "Even more string
 values!" ] }
   }
}
```

AWS AppSync 將其轉換為一個 JSON 對象：

```
{
```

```
    "someString" : "A string value",
    "someNumber" : 1,
    "stringSet"  : [ "Another string value", "Even more string values!" ]
}
```

Null 類型 **NULL**

null 值。

例如，如果 DynamoDB 傳回下列空值：

```
{ "NULL" : null }
```

AWS AppSync 將其轉換為空：

```
null
```

篩選條件

使用Query和Scan作業查詢 DynamoDB 中的物件時，您可以選擇性地指filter定評估結果並僅傳回所需值的一個。

Query 或 Scan 映射文件的篩選條件映射區段之結構如下：

```
"filter" : {
    "expression" : "filter expression"
    "expressionNames" : {
        "#name" : "name",
    },
    "expressionValues" : {
        ":value" : ... typed value
    },
}
```

欄位定義如下：

expression

查詢表達式。如需如何撰寫篩選器運算式的詳細資訊，請參閱 DynamoDB QueryFilter 和 Dynam ScanFilter o DB 文件。必須指定此欄位。

expressionNames

表達式屬性 name 預留位置的替代，形式為鍵值組。索引鍵對應至用於 expression 的名稱預留位置。此值必須是對應至 DynamoDB 中項目的屬性名稱的字串。此欄位為選用的，應只能填入用於 expression 中表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性 value 預留位置的替代，形式為鍵值組。鍵對應用於 expression 的值預留位置，值必須是類型值。如需如何指定「類型值」的詳細資訊，請參閱類型系統 (請求映射)。此必須指定。此欄位為選用的，應只能填入用於 expression 中表達式屬性值預留位置的替代。

範例

下列範例是對應範本的篩選器區段，其中只有在標題以引數開頭時，才會傳回從 DynamoDB 擷取的title項目。

```
"filter" : {
    "expression" : "begins_with(#title, :title)",
    "expressionNames" : {
        "#title" : "title"
    },
    "expressionValues" : {
        ":title" : $util.dynamodb.toDynamoDBJson($context.arguments.title)
    }
}
```

條件表達式

當您使用PutItem、UpdateItem和 DeleteItem DynamoDB 作業在 DynamoDB 中變更物件時，您可以選擇性地指定條件運算式，以根據執行作業前已在 DynamoDB 中的物件狀態控制請求是否成功。

AWS AppSync DynamoDB 解析器允許在、和DeleteItem請求對應文件中指定條件運算式 PutItemUpdateItem，以及在條件失敗且物件未更新時應遵循的策略。

範例 1

以下 PutItem 映射文件沒有條件表達式。因此，即使已存在具有相同金鑰的項目，它也會將項目置於 DynamoDB 中，進而覆寫現有項目。

```
{
```

```
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   }
}
```

範例 2

下列PutItem對應文件具有條件運算式，只有在 DynamoDB 中不存在具有相同索引鍵的項目時，才允許作業成功。

```
{
   "version" : "2017-02-28",
   "operation" : "PutItem",
   "key" : {
      "id" : { "S" : "1" }
   },
   "condition" : {
      "expression" : "attribute_not_exists(id)"
   }
}
```

依預設，如果條件檢查失敗，AWS AppSync DynamoDB 解析器會針對 GraphQL 回應區段中的data欄位中的突變和 DynamoDB 中物件的目前值傳回錯誤。error不過，AWS AppSync DynamoDB 解析器提供一些額外的功能，可協助開發人員處理一些常見的邊緣案例：

- 如果 AWS AppSync DynamoDB 解析器可以判斷 DynamoDB 中的目前值是否符合所需的結果，它會將作業視為無論如何都成功。
- 您可以將解析器設定為叫用自訂 Lambda 函數，以決定 AWS AppSync DynamoDB 解析器應如何處理失敗，而不會傳回錯誤。

處理條件檢查失敗一節將更詳細地說明這些內容。

如需有關 DynamoDB 條件運算式的詳細資訊，請參閱 DynamoDB ConditionExpressions 文件。

指定條件

PutItem、UpdateItem 和 DeleteItem 要求映射文件都允許可指定選用的 condition 區段。若省略，則不會有條件檢查。若指定，條件必須為 true，操作才會成功。

condition 區段的結構如下：

```
"condition" : {
    "expression" : "someExpression"
    "expressionNames" : {
        "#foo" : "foo"
    },
    "expressionValues" : {
        ":bar" : ... typed value
    },
    "equalsIgnore" : [ "version" ],
    "consistentRead" : true,
    "conditionalCheckFailedHandler" : {
        "strategy" : "Custom",
        "lambdaArn" : "arn:..."
    }
}
```

下列欄位指定條件：

expression

更新表達式本身。如需如何撰寫條件運算式的詳細資訊，請參閱 DynamoDB ConditionExpressions 文件。必須指定此欄位。

expressionNames

表達式屬性名稱預留位置的替代，形式為索引鍵值對。索引鍵對應至運算式中使用的名稱預留位置，而且值必須是對應至 DynamoDB 中項目的屬性名稱的字串。此欄位為選用的，應只能填入於表達式中所用表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性值預留位置的替代，形式為索引值對。鍵對應用於表達式的值預留位置，值必須是類型值。如需如何指定「類型值」的詳細資訊，請參閱類型系統 (請求映射)。此必須指定。此欄位為選用的，應只能填入用於表達式中表達式屬性值預留位置的替代。

其餘欄位會告訴 AWS AppSync DynamoDB 解析程式如何處理條件檢查失敗：

equalsIgnore

使用PutItem作業時，如果條件檢查失敗，AWS AppSync DynamoDB 解析器會將目前在 DynamoDB 中的項目與嘗試寫入的項目進行比較。如果兩者相同，則操作視為成功。您可以使用

equalsIgnore 欄位，指定 AWS AppSync 執行該比較時應忽略的屬性清單。例如，如果唯一的區別是version屬性，它會將作業視為成功。此欄位為選用欄位。

consistentRead

當條件檢查失敗時，AWS AppSync 會使用強烈一致性讀取從 DynamoDB 取得項目的目前值。您可以使用此欄位告知 AWS AppSync DynamoDB 解析程式改為使用最終一致性讀取。此欄位為選用，預設值為 true。

conditionalCheckFailedHandler

本節可讓您指定 DynamoDB 解析器在將 AWS AppSync DynamoDB 中的目前值與預期結果進行比較後，如何處理條件檢查失敗。此區段為選用。若省略，則會預設為 Reject 策略。

strategy

DynamoDB 解析程式在將 AWS AppSync DynamoDB 中的目前值與預期結果進行比較之後所採用的策略。此欄位為必填，且採用下列可能值：

Reject

突變失敗，並且 GraphQL 回應error區段中某個data欄位中 DynamoDB 中物件的突變和目前值出現錯誤。

Custom

AWS AppSync DynamoDB 解析器會叫用自訂 Lambda 函數來決定如何處理條件檢查失敗。當 strategy 設定為 Custom，lambdaArn 欄位必須包含要叫用 Lambda 函數的 ARN。

lambdaArn

要叫用的 Lambda 函數的 ARN，可決定 AWS AppSync DynamoDB 解析程式應如何處理條件檢查失敗。只有在 strategy 設定為 Custom 時，此欄位才必須指定。如需如何使用此功能的詳細資訊，請參閱處理條件檢查失敗。

處理條件檢查失敗

依預設，當條件檢查失敗時，AWS AppSync DynamoDB 解析器會針對 GraphQL 回應區段中的data欄位中的突變和 DynamoDB 中物件的目前值傳回錯誤。error不過，AWS AppSync DynamoDB 解析器提供一些額外的功能，可協助開發人員處理一些常見的邊緣案例：

- 如果 AWS AppSync DynamoDB 解析器可以判斷 DynamoDB 中的目前值是否符合所需的結果，它會將作業視為無論如何都成功。

- 您可以將解析器設定為叫用自訂 Lambda 函數，以決定 AWS AppSync DynamoDB 解析器應如何處理失敗，而不會傳回錯誤。

此程序的流程圖為：

檢查所需的結果

當條件檢查失敗時，AWS AppSync DynamoDB 解析器會執行 DynamoDB 請求，以從 `GetItem` DynamoDB 取得項目的目前值。在預設情況下，它會使用強式一致性讀取，但這可使用 `condition` 區塊中的 `consistentRead` 欄位來設定，並和預期結果進行比較：

- 針對此`PutItem`作業，AWS AppSync DynamoDB 解析器會將目前值與其嘗試寫入的值進行比較，排除比較中`equalsIgnore`列出的任何屬性。如果項目相同，則會將作業視為成功，並傳回從 DynamoDB 擷取的項目。否則，其將按照設定的策略。

  例如，如果 `PutItem` 要求映射文件外觀如下：

```
{
   "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" ]
   }
}
```

  而目前在 DynamoDB 中的項目外觀如下：

```
{
   "id" : { "S" : "1" },
```

```
    "name" : { "S" : "Steve" },
    "version" : { "N" : 8 }
}
```

AWS AppSync DynamoDB 解析器會將嘗試寫入的項目與目前值進行比較，請參閱唯一的區別是`version`欄位，但由於設定為忽略欄`version`位，因此會將作業視為成功，並傳回從 DynamoDB 擷取的項目。

- 針對此`DeleteItem`作業，AWS AppSync DynamoDB 解析程式會檢查以確認是否已從 DynamoDB 傳回項目。如果沒有項目傳回，則操作視為成功。否則，其將按照設定的策略。
- 針對此`UpdateItem`作業，AWS AppSync DynamoDB 解析器沒有足夠的資訊來判斷 DynamoDB 中目前的項目是否符合預期的結果，因此遵循設定的策略。

如果 DynamoDB 中物件的目前狀態與預期結果不同，AWS AppSync DynamoDB 解析器會遵循設定的策略，以拒絕突變或叫用 Lambda 函數來決定下一步要執行的動作。

遵循「拒絕」策略

遵循`Reject`策略時，AWS AppSync DynamoDB 解析器會針對突變傳回錯誤，而 DynamoDB 中物件的目前值也會傳回 GraphQL 回應區段中的`data`欄位中。`error`從 DynamoDB 傳回的項目會透過回應對應範本放置，以將其轉譯成用戶端預期的格式，並依選集進行篩選。

例如，假設變動要求如下：

```
mutation {
    updatePerson(id: 1, name: "Steve", expectedVersion: 1) {
        Name
        theVersion
    }
}
```

如果從 DynamoDB 傳回的項目如下所示：

```
{
   "id" : { "S" : "1" },
   "name" : { "S" : "Steve" },
   "version" : { "N" : 8 }
}
```

而回應映射範本如下所示：

```
{
   "id" : $util.toJson($context.result.id),
   "Name" : $util.toJson($context.result.name),
   "theVersion" : $util.toJson($context.result.version)
}
```

GraphQL 回應的外觀如下所示：

```
{
  "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
      },
      ...
    }
  ]
}
```

此外，如果傳回物件中的任何欄位皆由其他解析程式填入，且變動成功，則在 `error` 區段中傳回物件時，不會解析這些物件。

遵循「自定義」策略

遵循`Custom`策略時，AWS AppSync DynamoDB 解析程式會叫用 Lambda 函數來決定下一步該做什麼。Lambda 函數會選擇下列其中一個選項：

- `reject` 變動。這會告訴 AWS AppSync DynamoDB 解析器的行為如同已設定策略一樣`Reject`，傳回 DynamoDB 中物件的突變錯誤和目前值，如上一節所述。
- `discard` 變動。這會告知 AWS AppSync DynamoDB 解析程式以無訊息方式忽略條件檢查失敗，並傳回 DynamoDB 中的值。
- `retry` 變動。這會告訴 AWS AppSync DynamoDB 解析器使用新的請求對應文件重試突變。

Lambda 叫用要求

AWS AppSync DynamoDB 解析程式會叫用中指定的 Lambda 函數。`lambdaArn`它會使用資料來源上所設定的相同 `service-role-arn`。叫用承載的結構如下：

```
{
    "arguments": { ... },
    "requestMapping": {... },
    "currentValue": { ... },
    "resolver": { ... },
    "identity": { ... }
}
```

欄位定義如下：

`arguments`

來自 GraphQL 變動的引數。這與可在 `$context.arguments` 中要求映射文件取得的引數相同。

`requestMapping`

此操作的要求映射文件。

`currentValue`

物件在 DynamoDB 中的目前值。

`resolver`

AWS AppSync 解析程式的相關資訊。

`identity`

發起人的相關資訊。這與可在 `$context.identity` 中要求映射文件取得的身分資訊相同。

承載的完整範例：

```
{
    "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": "AIDAAAAAAAAAAAAAAAAAA",
        "userArn": "arn:aws:iam::123456789012:user/appsync"
    }
}
```

Lambda 叫用回應

Lambda 函數可以檢查叫用承載，並套用任何商業邏輯來決定 AWS AppSync DynamoDB 解析器應如何處理失敗。處理條件檢查失敗有三個選項：

- `reject` 變動。此選項的回應承載必須具有此架構：

```
{
```

```
    "action": "reject"
}
```

這會告訴 AWS AppSync DynamoDB 解析器的行為如同已設定策略一樣`Reject`，傳回 DynamoDB 中物件的突變和目前值的錯誤，如上節所述。

- `discard` 變動。此選項的回應承載必須具有此架構：

```
{
    "action": "discard"
}
```

這會告知 AWS AppSync DynamoDB 解析程式以無訊息方式忽略條件檢查失敗，並傳回 DynamoDB 中的值。

- `retry` 變動。此選項的回應承載必須具有此架構：

```
{
    "action": "retry",
    "retryMapping": { ... }
}
```

這會告訴 AWS AppSync DynamoDB 解析器使用新的請求對應文件重試突變。`retryMapping`區段的結構取決於 DynamoDB 作業，而且是該作業之完整請求對應文件的子集。

若是 `PutItem`，`retryMapping` 區段的結構如下。如需`attributeValues`欄位的描述，請參閱PutItem。

```
{
    "attributeValues": { ... },
    "condition": {
        "equalsIgnore" = [ ... ],
        "consistentRead" = true
    }
}
```

若是 `UpdateItem`，`retryMapping` 區段的結構如下。如需`update`區段的描述，請參閱UpdateItem。

```
{
    "update" : {
```

```
        "expression" : "someExpression"
        "expressionNames" : {
            "#foo" : "foo"
        },
        "expressionValues" : {
            ":bar" : ... typed value
        }
    },
    "condition": {
        "consistentRead" = true
    }
}
```

若是 DeleteItem，retryMapping 區段的結構如下。

```
{
    "condition": {
        "consistentRead" = true
    }
}
```

無法指定使用不同的操作或索引鍵。AWS AppSync DynamoDB 解析器僅允許在同一物件上重試相同作業。此外，condition 區段不允許指定 conditionalCheckFailedHandler。如果重試失敗，AWS AppSync DynamoDB 解析器會遵循策略。Reject

此為 Lambda 函數處理失敗 PutItem 要求的範例。商業邏輯的重點是發起人為何。如果它是由提出的jeffTheAdmin，它會重試請求，expectedVersion從 DynamoDB 中目前的項目更新version和。否則，它會拒絕變動。

```
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)
};
```

交易條件運算式

交易條件表達式可在 TransactWriteItems 中四種類型操作的請求映射範本中使用，即 PutItem、DeleteItem、UpdateItem 以及 ConditionCheck。

對於PutItem、和 DeleteItemUpdateItem，交易條件運算式是選擇性的。對於ConditionCheck，需要交易條件運算式。

範例 1

以下交易 DeleteItem 映射文件沒有條件表達式。因此，它會刪除 DynamoDB 中的項目。

```
{
   "version": "2018-05-29",
   "operation": "TransactWriteItems",
   "transactItems": [
      {
         "table": "posts",
         "operation": "DeleteItem",
         "key": {
            "id": { "S" : "1" }
         }
      }
   ]
```

```
}
```

範例 2

下列交易DeleteItem對應文件確實具有交易條件運算式，只有在該文章的作者等於特定名稱時，才允許作業成功。

```
{
   "version": "2018-05-29",
   "operation": "TransactWriteItems",
   "transactItems": [
      {
         "table": "posts",
         "operation": "DeleteItem",
         "key": {
            "id": { "S" : "1" }
         }
         "condition": {
            "expression": "author = :author",
            "expressionValues": {
               ":author": { "S" : "Chunyan" }
            }
         }
      }
   ]
}
```

如果條件檢查失敗，會導致 TransactionCanceledException，並且會在 $ctx.result.cancellationReasons 中傳回錯誤詳細資料。請注意，根據預設，DynamoDB 中使條件檢查失敗的舊項目將傳回中。$ctx.result.cancellationReasons

指定條件

PutItem、UpdateItem 和 DeleteItem 要求映射文件都允許可指定選用的 condition 區段。若省略，則不會有條件檢查。若指定，條件必須為 true，操作才會成功。ConditionCheck 必須有待指定的 condition 區段。條件必須為 true，整個交易才會成功。

condition 區段的結構如下：

```
"condition": {
    "expression": "someExpression",
    "expressionNames": {
```

```
        "#foo": "foo"
    },
    "expressionValues": {
        ":bar": ... typed value
    },
    "returnValuesOnConditionCheckFailure": false
}
```

下列欄位指定條件：

expression

更新表達式本身。如需如何撰寫條件運算式的詳細資訊，請參閱 DynamoDB ConditionExpressions 文件。必須指定此欄位。

expressionNames

表達式屬性名稱預留位置的替代，形式為索引鍵值對。索引鍵對應至運算式中使用的名稱預留位置，而且值必須是對應至 DynamoDB 中項目的屬性名稱的字串。此欄位為選用的，應只能填入於表達式中所用表達式屬性名稱預留位置的替代。

expressionValues

表達式屬性值預留位置的替代，形式為索引值對。鍵對應用於表達式的值預留位置，值必須是類型值。如需如何指定「類型值」的詳細資訊，請參閱〈類型系統 (請求映射)〉。此必須指定。此欄位為選用的，應只能填入用於表達式中表達式屬性值預留位置的替代。

returnValuesOnConditionCheckFailure

指定當條件檢查失敗時，是否要將 DynamoDB 中的項目擷取回來。擷取的項目將位於 `$ctx.result.cancellationReasons[$index].item` 中，其中 `$index` 是條件檢查失敗之請求項目的索引。此值預設為 true。

投影

使用、、和`TransactGetItems`操作讀取 DynamoDB 中的物件時 `GetItem` `Scan` `QueryBatchGetItem`，您可以選擇性地指定識別所需屬性的投影。投影具有以下結構，類似於篩選器：

```
"projection" : {
    "expression" : "projection expression"
    "expressionNames" : {
        "#name" : "name",
```

```
    }
}
```

欄位定義如下：

`expression`

投影表達式，它是一個字符串。若要擷取單一屬性，請指定其名稱。對於多個屬性，名稱必須是逗號分隔值。如需撰寫投影運算式的詳細資訊，請參閱 DynamoDB 投影運算式文件。此欄位為必填。

`expressionNames`

以鍵值對的形式替換表達式屬性名稱佔位符。索引鍵對應至用於 `expression` 的名稱預留位置。此值必須是對應至 DynamoDB 中項目的屬性名稱的字串。此欄位為選擇性欄位，且只應填入中使用的表示式屬性名稱預留位置符號的替代。`expression`如需有關的詳細資訊`expressionNames`，請參閱 DynamoDB 文件。

範例 1

下列範例是 VTL 對應範本的投影區段，其中僅從 DynamoDB 傳回屬性`author`和`id`：

```
"projection" : {
    "expression" : "#author, id",
    "expressionNames" : {
        "#author" : "author"
    }
}
```

> **Tip**
>
> 您可以使用 $ 上下文 .info 來存取您的 GraphQL 請求選擇集。 selectionSetList。此欄位可讓您根據需求動態構建投影運算式的框架。

> **Note**
>
> 將投影運算式與`Query`和`Scan`運算搭配使用時，的值`select`必須是`SPECIFIC_ATTRIBUTES`。如需詳細資訊，請參閱 DynamoDB 文件。

RDS 的解析程式對應範本參考

AWS AppSyncRDS 解析器對應範本可讓開發人員將 SQL 查詢傳送至適用於 Amazon Aurora 無伺服器的資料 API，並取回這些查詢的結果。

請求對應範本

RDS 要求映射範本非常簡單：

```
{
    "version": "2018-05-29",
    "statements": [],
    "variableMap": {},
    "variableTypeHintMap": {}
}
```

以下是 RDS 請求映射範本在解析後呈現的 JSON 結構描述。

```
{
    "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"
        }
    }
}
```

以下是具有靜態查詢的請求映射模板的示例：

```
{
    "version": "2018-05-29",
    "statements": [
        "select title, isbn13 from BOOKS where author = 'Mark Twain'"
    ]
}
```

版本

版本欄位對所有請求對應範本都通用，可定義範本使用的版本。版本欄位為必填欄位。該值「2018-05-29」是亞馬遜 RDS 映射範本支援的唯一版本。

```
"version": "2018-05-29"
```

聲明和 VariableMap

語句數組是開發人員提供的查詢的佔位符。目前，每個請求對應範本最多支援兩個查詢。這variableMap是選擇性欄位，其中包含別名，可用來讓 SQL 陳述式更短且更具可讀性。例如，以下是可能的：

```
{
"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將使用變數對應值建構將傳送至 Amazon Aurora 無伺服器資料 API 的SqlParameterized查詢。SQL 陳述式會使用變數對映中提供的參數來執行，從而消除 SQL 插入的風險。

VariableTypeHintMap

variableTypeHintMap是包含別名類型的選擇性欄位，可用來傳送 SQL 參數類型提示。這些類型提示可避免在 SQL 陳述式中明確轉換，從而縮短它們。例如，以下是可能的：

```
{
    "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將使用變數對應值來建構傳送至 Amazon Aurora 無伺服器資料 API 的查詢。它還使用數`variableTypeHintMap`據並將類型的信息發送到 RDS。支持 RDS `typeHints` 可以在這裡找到。

的解析程式對映範本參考 OpenSearch

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

Amazon OpenSearch 服務的AWS AppSync 解析器可讓您使用 GraphQL 存放和擷取帳戶中現有 OpenSearch 服務網域中的資料。此解析器的工作原理是允許您將傳入的 GraphQL 請求映射到服務請求中，然後將 OpenSearch 服務響應映射回 GraphQL。 OpenSearch 本節說明支援之 OpenSearch 服務作業的對應範本。

請求映射範本

大多數的 OpenSearch 服務請求對應範本都有一個共同的結構，其中只有幾個部分變更。下列範例會針對 OpenSearch Service 網域執行搜尋，其中文件會根據名為的索引進行組織`post`。搜尋參數定義於 `body` 區段，許多常見的查詢子句定義於 `query` 欄位。此範例將搜尋在 `"Nadia"` 欄位中包含 `"Bailey"` 或 `author` (或兩者) 的文件：

```
{
    "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" }}
                    ]
                }
            }
        }
    }
}
```

回應映射範本

與其他資料來源一樣， OpenSearch 服務會傳送需AWS AppSync 要轉換為 GraphQL 的回應。

大多數 GraphQL 查詢都在從 OpenSearch 服務響應中查找`_source`字段。因為您可以執行搜尋以傳回個別文件或文件清單，因此 OpenSearch Service 中使用了兩種常見的回應對應範本：

結果清單

```
[
    #foreach($entry in $context.result.hits.hits)
      #if( $velocityCount > 1 ) , #end
        $utils.toJson($entry.get("_source"))
    #end
]
```

個別項目

```
$utils.toJson($context.result.get("_source"))
```

`operation` 欄位

(僅請求映射範本)

發送到 OpenSearch 服務域的 HTTP 方法或動詞（獲取，AWS AppSync 發布，放置，頭或刪除）。金鑰與值皆必須為字串。

```
"operation" : "PUT"
```

path 欄位

(僅請求映射範本)

來源 OpenSearch 服務要求的搜尋路徑AWS AppSync。這會形成操作的 HTTP 動詞的 URL。金鑰與值皆必須為字串。

```
"path" : "/<indexname>/_doc/<_id>"
"path" : "/<indexname>/_doc"
"path" : "/<indexname>/_search"
"path" : "/<indexname>/_update/<_id>
```

評估對應範本時，此路徑會作為 HTTP 要求的一部分傳送，包括 OpenSearch 服務網域。例如，之前的範例可轉譯為：

```
GET https://opensearch-domain-name.REGION.es.amazonaws.com/indexname/type/_search
```

params 欄位

(僅請求映射範本)

用來指定搜尋執行的動作，最常見的方式是在主體內部設定查詢值。不過，有多項其他功能可設定，例如回應的格式。

- 標頭

  標頭資訊，以金鑰值對形式。金鑰與值皆必須為字串。例如：

  ```
  "headers" : {
      "Content-Type" : "application/json"
  }
  ```

  > **Note**
  >
  > AWS AppSync 目前僅支援 JSON 作為`Content-Type`.

- queryString

  金鑰值對，指定常見的選項，例如 JSON 回應的程式碼格式。金鑰與值皆必須為字串。例如，如果您希望獲得非常完整格式的 JSON，請使用：

```
"queryString" : {
    "pretty" : "true"
}
```

- 本文

  這是請求的主要部分，允許AWS AppSync 向您的 OpenSearch 服務域創建格式良好的搜索請求。金鑰必須是由物件組成的字串。以下顯示幾個示範。

範例 1

傳回城市符合「seattle」的所有文件：

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "match" : {
            "city" : "seattle"
        }
    }
}
```

範例 2

傳回所有符合「washington」做為城市或州的文件：

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "multi_match" : {
            "query" : "washington",
            "fields" : ["city", "state"]
        }
    }
}
```

傳遞變數

(僅請求映射範本)

您也可以將變數做為 VTL 陳述式中評估的一部分進行傳遞。例如，假設您有一個 GraphQL 查詢如下：

```
query {
    searchForState(state: "washington"){
        ...
    }
}
```

此映射範本可採取狀態做為引數：

```
"body":{
    "from":0,
    "size":50,
    "query" : {
        "multi_match" : {
            "query" : "$context.arguments.state",
            "fields" : ["city", "state"]
        }
    }
}
```

如需您可包含在 VTL 中的公用程式清單，請參閱存取請求標頭。

Lambda 的解析程式對應範本參考

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

您可以使用 AWS AppSync 函數和解析器來叫用帳戶中的 Lambda 函數。在將 Lambda 函數傳回給客戶之前，您可以調整請求承載和來自 Lambda 函數的回應。您也可以使用對應範本，提供 AWS AppSync 有關要叫用之作業性質的提示。本節說明支援 Lambda 作業的不同對應範本。

請求對應範本

Lambda 要求對應範本會處理與 Lambda 函數相關的欄位：

```
{
  "version": string,
  "operation": Invoke|BatchInvoke,
  "payload": any type,
  "invocationType": RequestResponse|Event
}
```

這是解決時 Lambda 請求映射模板的 JSON 結構描述表示：

```
{
  "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
}
```

以下是一個使用invoke操作的示例，其有效負載數據是 GraphQL 架構中的getPost字段及其來自上下文的參數：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": $util.toJson($context.arguments)
  }
}
```

整個對應文件會作為輸入傳遞至 Lambda 函數，以便前面的範例現在看起來像這樣：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "payload": {
    "field": "getPost",
    "arguments": {
      "id": "postId1"
    }
  }
}
```

版本

所有請求對應範本都通用，會version定義範本使用的版本。version是必要的，是靜態值：

```
"version": "2018-05-29"
```

作業

Lambda 資料來源可讓您在operation欄位中定義兩項作業：Invoke和BatchInvoke。該Invoke操作讓 AWS AppSync 知道為每個 GraphQL 字段解析器調用您的 Lambda 函數。BatchInvoke指示對 AWS AppSync 目前 GraphQL 欄位進行批次處理要求。operation 欄位是必要的。

對於Invoke，已解析的請求對應範本與 Lambda 函數的輸入有效負載相符。讓我們修改上面的例子：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
    "payload": {
      "arguments": $util.toJson($context.arguments)
    }
}
```

這被解決並傳遞給 Lambda 函數，它可能看起來像這樣：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
    "payload": {
      "arguments": {
        "id": "postId1"
      }
    }
}
```

對於BatchInvoke，對應範本會套用至批次中的每個欄位解析程式。為了簡潔起見，將所有已解析的映射模板payload值合 AWS AppSync 併到與映射模板匹配的單個對象下的列表中。以下範例範本顯示此合併：

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": $util.toJson($context)
}
```

此範本已解析為以下映射文件：

```
{
  "version": "2018-05-29",
  "operation": "BatchInvoke",
  "payload": [
    {...}, // context for batch item 1
    {...}, // context for batch item 2
    {...}  // context for batch item 3
  ]
}
```

清單中的每個元素都對應於payload單一批次項目。Lambda 函數還預計將返回與請求中發送的項目順序匹配的列表形狀響應：

```
[
  { "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
]
```

承載

該payload字段是用於將任何格式良好的 JSON 傳遞給 Lambda 函數的容器。如果operation欄位設定為BatchInvoke，則將現有payload值 AWS AppSync 包裝到清單中。此 payload 欄位為選用。

調用類型

Lambda 資料來源可讓您定義兩種叫用類型：RequestResponse和Event. 呼叫型別與 Lambda API 中定義的叫用型別同義。RequestResponse叫用類型可讓您同步 AWS AppSync 呼叫 Lambda 函數以等待回應。呼Event叫可讓您以非同步方式叫用 Lambda 函數。如需 Lambda 如何處理Event叫用類型要求的詳細資訊，請參閱非同步叫用。此 invocationType 欄位為選用。如果此欄位未包含在要求中， AWS AppSync 將會預設為RequestResponse呼叫類型。

對於任何invocationType欄位，已解析的請求都會與 Lambda 函數的輸入有效負載相符。讓我們修改上面的例子：

```
{
  "version": "2018-05-29",
```

```
  "operation": "Invoke",
  "invocationType": "Event"
  "payload": {
    "arguments": $util.toJson($context.arguments)
  }
}
```

這被解決並傳遞給 Lambda 函數，它可能看起來像這樣：

```
{
  "version": "2018-05-29",
  "operation": "Invoke",
  "invocationType": "Event",
  "payload": {
    "arguments": {
      "id": "postId1"
    }
  }
}
```

當BatchInvoke作業與Event叫用類型欄位搭配使用時，會以上述相同方式合 AWS AppSync 併欄位解析程式，並將要求作為非同步事件傳遞至 Lambda 函數，並將其作為值清單傳遞至您的 Lambda 函數。payload我們建議您停用Event叫用類型解析器的解析器快取，因為如果有快取命中，就不會將這些解析程式傳送至 Lambda。

回應對映範本

與其他資料來源一樣，您的 Lambda 函數會傳送回應， AWS AppSync 該回應必須轉換為 GraphQL 類型。

Lambda 函數的結果是在可透過「速度範本語言」(VTL) $context.result 屬性取得的context物件上設定。

如果您的 Lambda 函式回應形狀完全匹配 GraphQL 類型形狀時，您可以使用以下回應映射範本轉送回應：

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

沒有任何必要欄位或形狀限制適用於回應映射範本。不過，由於 GraphQL 是強類型，解析後的映射範本必須符合預期的 GraphQL 類型。

Lambda 函數批次回應

如果operation欄位設定為BatchInvoke，則 AWS AppSync需要 Lambda 函數回來的項目清單。為了 AWS AppSync 將每個結果映射回原始請求項，響應列表必須在大小和順序匹配。在響應列表中有null項目$ctx.result是有效的; 相應地設置為 null。

直接 Lambda 解析器

如果您想要完全規避對應範本的使用， AWS AppSync 可以為 Lambda 函數提供預設承載，並提供 GraphQL 類型的預設 Lambda 函數回應。您可以選擇提供請求範本、回應範本或兩者皆不提供，並據此 AWS AppSync 處理。

直接 Lambda 求映射模板

如果未提供請求對應範本，則 AWS AppSync 會將Context物件作為Invoke作業直接傳送至您的 Lambda 函數。如需 Context 物件結構的詳細資訊，請參閱 解析器對映範本前後關聯參考。

直接 Lambda 回應對應範本

未提供回應對應範本時， AWS AppSync 會在收到 Lambda 函數的回應時執行下列其中一項作業。如果您未提供請求對應範本，或者您已提供具有版本的請求對應範本2018-05-29，則回應將等同於下列回應對應範本：

```
#if($ctx.error)
     $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
 #end
 $util.toJson($ctx.result)
```

如果您提供了具有版本的範本2017-02-28，則回應邏輯的功能會等同於下列回應對應範本：

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

從表面上看，對應範本略過的運作方式與使用上述範例中所示的某些對應範本類似。但是，在幕後，完全規避了對映模板的評估。由於範本評估步驟已略過，因此在某些情況下，與需要評估之回應對應範本的 Lambda 函數相比，在某些情況下，應用程式在回應期間可能會遇到較少的額外負荷和延遲。

直接 Lambda 解析器回應中的自訂錯誤處理

您可以透過引發自訂例外狀況，從直接 Lambda 解析器呼叫的 Lambda 函數自訂錯誤回應。下面的例子演示了如何使用創建自定義異常 JavaScript：

```
class CustomException extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomException";
  }
}

throw new CustomException("Custom message");
```

引發例外狀況時，errorType和分別是errorMessage所擲回的自訂錯誤的name和message。

如果errorType是UnauthorizedException，則 AWS AppSync傳回預設訊息 ("You are not authorized to make this call.") 而非自訂訊息。

下列程式碼片段是示範自訂errorType的 GraphQL 回應範例：

```
{
  "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：已啟用批次處理

您可以透過maxBatchSize在解析器上設定，為直接 Lambda 解析器啟用批次處理。當設定maxBatchSize為大0於直接 Lambda 解析器的值時， AWS AppSync 會將請求批次傳送至您的 Lambda 函數，大小最大為。maxBatchSize

在直接 Lambda 解析器0上設定maxBatchSize為會關閉批次處理。

如需使用 Lambda 解析器進行批次處理方式的詳細資訊，請參閱。進階使用案例：批次處理

請求對應範本

啟用批次處理且未提供請求對應範本時， AWS AppSync 會將Context物件清單做為BatchInvoke作業直接傳送至 Lambda 函數。

回應對映範本

啟用批次處理且未提供回應對應範本時，回應邏輯等同於下列回應對應範本：

```
#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 函數必須以與已傳送Context物件清單相同的順序傳回結果清單。您可以errorType針對特定結果提供errorMessage和來傳回個別錯誤。清單中的每個結果具有下列格式：

```
{
   "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**
>
> 目前忽略結果物件中的其他欄位。

處理來自 Lambda 錯誤

您可以在 Lambda 函數中擲回例外或錯誤，以針對所有結果傳回錯誤。如果批次要求的承載要求或回應大小太大，Lambda 會傳回錯誤。在這種情況下，您應該考慮減少`maxBatchSize`或減少響應有效負載的大小。

如需處理個別錯誤的資訊，請參閱返回個別錯誤。

範例 Lambda 數

使用下面的結構描述，您可以為`Post.relatedPosts`欄位解析器建立直接 Lambda 解析器，並透過上述設定啟用批次處理：`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]
}
```

在下面的查詢中，Lambda 函數將被調用與一批請求來解決`relatedPosts`：

```
query getAllPosts {
  allPosts {
```

```
    id
    relatedPosts {
      id
    }
  }
}
```

下面提供了 Lambda 函數的簡單實現：

```
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' }
    }
  })
}
```

的解析程式對映範本參考 EventBridge

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

與 EventBridge 資料來源搭配使用的 AWS AppSync 解析器對應範本可讓您將自訂事件傳送到 Amazon EventBridge 匯流排。

請求對應範本

`PutEvents`請求對應範本可讓您將多個自訂事件傳送至 EventBridge事件匯流排。 映射文件結構如下：

```
{
    "version" : "2018-05-29",
    "operation" : "PutEvents",
    "events" : [{}]
}
```

以下是請求對應範本的範例 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"
    }

    ]
}
```

回應對映範本

如果PutEvents作業成功，則來自 EventBridge 的回應會包含在$ctx.result：

```
#if($ctx.error)
  $util.error($ctx.error.message, $ctx.error.type, $ctx.result)
#end
  $util.toJson($ctx.result)
```

執行PutEvents作業 (例如InternalExceptions或) 時發生Timeouts的錯誤$ctx.error。如需常見錯誤 EventBridge的清單，請參閱EventBridge 常見錯誤參考資料。

result將採用以下格式：

```
{
    "Entries" [
        {
            "ErrorCode" : String,
            "ErrorMessage" : String,
            "EventId" : String
        }
    ],
    "FailedEntryCount" : number
}
```

- 作品

  攝入的事件結果，無論是成功還是不成功。如果擷取成功，則項目EventID中有。否則，您可以使用ErrorCode和ErrorMessage來識別項目的問題。

  對於每個記錄，響應元素的索引是相同的請求數組中的索引。
- FailedEntryCount

失敗項目的數目。此值表示為整數。

如需有關的回應的詳細資訊PutEvents，請參閱PutEvents。

範例回應範例 1

下列範例是具有兩個成功事件的PutEvents作業：

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        }
    ],
    "FailedEntryCount" : 0
}
```

範例回應範例 2

下列範例是包含三個事件、兩個成功和一個失敗的PutEvents作業：

```
{
    "Entries" : [
        {
            "EventId": "11710aed-b79e-4468-a20b-bb3c0c3b4860"
        },
        {
            "EventId": "d804d26a-88db-4b66-9eaf-9a11c708ae82"
        },
        {
            "ErrorCode" : "SampleErrorCode",
            "ErrorMessage" : "Sample Error Message"
        }
    ],
    "FailedEntryCount" : 1
}
```

PutEvents 欄位

- 版本

  對於所有請求對應範本而言，此version欄位會定義範本使用的版本。此欄位為必填。此值2018-05-29是 EventBridge 對應範本支援的唯一版本。

- 操作

  唯一支援的作業是PutEvents。此作業可讓您將自訂事件新增至事件匯流排。

- 事件

  將新增至事件匯流排的事件陣列。這個數組應該有 1-10 個項目的分配。

  Event物件是有效的 JSON 物件，其中包含下列欄位：

  - "source": 定義事件來源的字串。
  - "detail"：可用來附加事件相關資訊的 JSON 物件。此欄位可以是空的 map ({ })。
  - "detailType：識別事件類型的字串。
  - "resources"：識別事件中涉及的資源的 JSON 字符串數組。此欄位可以是空陣列。
  - "time": 以字串形式提供的事件時間戳記。這應該遵循 RFC3339 時間戳記格式。

下面的片段是一些有效Event對象的例子：

範例 1

```
{
    "source" : "source1",
    "detail" : {
        "key1" : [1,2,3,4],
        "key2" : "strval"
    },
    "detailType" : "sampleDetailType",
    "resources" : ["Resouce1", "Resource2"],
    "time" : "2022-01-10T05:00:10Z"
}
```

範例 2

```
{
```

```
    "source" : "source1",
    "detail" : {},
    "detailType" : "sampleDetailType"
}
```

範例 3

```
{
    "source" : "source1",
    "detail" : {
        "key1" : 1200
    },
    "detailType" : "sampleDetailType",
    "resources" : []
}
```

無資料來源的解析程式對映範本參考

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

與 None 類型的資料來源搭配使用的AWS AppSync 解析程式對映範本，可讓您塑造AWS AppSync 本機作業的要求。

請求對應範本

映射範本非常簡單，可讓您透過 `payload` 欄位來傳遞任意數量的內容資訊。

```
{
   "version": string,
   "payload": any type
}
```

以下是請求映射範本在解析後呈現的 JSON 結構描述：

```
{
```

```
    "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
}
```

以下是透過 VTL 內容屬性$context.arguments傳遞欄位引數的範例：

```
{
    "version": "2018-05-29",
    "payload": $util.toJson($context.arguments)
}
```

payload 欄位的值將轉發到回應映射範本，並可在 VTL 內容屬性 ($context.result) 上使用。

這是代表 payload 欄位插入值的範例：

```
{
    "id": "postId1"
}
```

版本

對於所有請求對應範本而言，此version欄位會定義範本使用的版本。

version 欄位是必要的。

範例：

```
"version": "2018-05-29"
```

承載

payload 欄位是一種容器，您可以用來將任何格式正確的 JSON 傳遞到回應映射範本。

此 payload 欄位為選用。

回應對映範本

由於沒有資料來源，payload 欄位的值將轉發到回應映射範本，並在可透過 VTL context 屬性使用的 $context.result 物件上設定。

如果您的 payload 欄位值形狀完全匹配 GraphQL 類型形狀時，您可以使用以下回應映射範本轉送回應：

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

沒有任何必要欄位或形狀限制適用於回應映射範本。不過，由於 GraphQL 是強類型，解析後的映射範本必須符合預期的 GraphQL 類型。

HTTP 的解析程式對映範本參考

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

AWS AppSync HTTP 解析程式映射範本可讓您從 AWS AppSync 傳送請求至任何 HTTP 端點，並從您的 HTTP 端點傳送回應至 AWS AppSync。使用映射範本，您可以提供要呼叫操作的相關特性提示給 AWS AppSync 。本節說明支援的 HTTP 解析程式的不同映射範本。

請求映射範本

```
{
```

```
    "version": "2018-05-29",
    "method": "PUT|POST|GET|DELETE|PATCH",
    "params": {
        "query": Map,
        "headers": Map,
        "body": any
    },
    "resourcePath": string
}
```

在解析 HTTP 請求映射範本後，請求映射範本的 JSON 結構描述表示看起來會如下：

```
{
    "$id": "https://aws.amazon.com/appsync/request-mapping-template.json",
    "type": "object",
    "properties": {
        "version": {
        "$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 請求的範例，其中包括 `text/plain` 本文：

```
{
    "version": "2018-05-29",
    "method": "POST",
    "params": {
        "headers":{
        "Content-Type":"text/plain"
        },
        "body":"this is an example of text body"
    },
    "resourcePath": "/"
}
```

版本

僅請求映射範本

定義範本使用的版本。version 是所有請求映射範本的共用項目，並且是必要項目。

```
"version": "2018-05-29"
```

方法

僅請求映射範本

AWS AppSync 傳送到 HTTP 端點的 HTTP 方法或動詞 (GET、POST、PUT、PATCH 或 DELETE)。

```
"method": "PUT"
```

ResourcePath

僅請求映射範本

您要存取的資源路徑。資源路徑與 HTTP 資料來源中的端點會組合成 AWS AppSync 服務發出請求的目標 URL。

```
"resourcePath": "/v1/users"
```

評估映射範本時，此路徑會做為 HTTP 請求的一部分進行傳送，包括 HTTP 端點。例如，之前的範例可轉譯如下：

```
PUT <endpoint>/v1/users
```

參數欄位

僅請求映射範本

用來指定搜尋執行的動作，最常見的方式是在主體內設定查詢值。不過，有多項其他功能可設定，例如回應的格式。

標頭

標頭資訊，以金鑰值對形式。金鑰與值皆必須為字串。

例如：

```
"headers" : {
    "Content-Type" : "application/json"
}
```

目前支援的 Content-Type 標頭包括：

```
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：您不能設定以下 HTTP 標頭：

```
HOST
CONNECTION
USER-AGENT
EXPECTATION
TRANSFER_ENCODING
CONTENT_LENGTH
```

query

金鑰值對，指定常見的選項，例如 JSON 回應的程式碼格式。金鑰與值皆必須為字串。以下範例顯示如何傳送 `?type=json` 查詢字串：

```
"query" : {
    "type" : "json"
}
```

本文

此本文包含您選擇設定的 HTTP 請求本文。除非內容類型指定字元集，否則請求本文一律是 UTF-8 編碼的字串。

```
"body":"body string"
```

憑證授權機構 (CA)AWS AppSyncHTTPS 端點

> **Note**
>
> 讓我們加密通過同性信託和異性格魯特 x1憑證。如果您使用「讓我們加密」，則無需執行任何操作。

此時，HTTP 解析器在使用 HTTPS 時不支持自簽名證書。AWS在解析 HTTPS 的 SSL/TLS 證書時，應用同步會識別以下證書頒發機構：

中的已知根憑證AWS AppSync

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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
amzninternalinfoseccag3	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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
geotrustprimarycertificatio nauthorityg3	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
geotrustprimarycertificatio nauthorityg2	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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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
ttelesecglobalroot class3	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
ttelesecglobalroot class2	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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

名稱	日期	SHA1 指紋
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

解析器映射模板更新日誌

> **Note**
>
> 我們現在主要支援 APPSYNC_JS 執行階段及其說明文件。請考慮在此處使用 APPSYNC_JS 執行階段及其指南。

解析程式和函數映射範本已採取版本控制。對應範本版本 (例如2018-05-29) 會指定下列項目：* 要求範本所提供之資料來源要求組態的預期形狀 * 請求對應範本與回應對應範本的執行行為

版本呈現格式為 YYYY-MM-DD，較晚日期代表較新版本。此頁面列出目前支援的對應範本版本之間的差異AWS AppSync。

主題

- 每種版本陣列的可用資料來源操作
- 變更單位解析程式映射範本的版本
- 變更函數的版本
- 2018-05-29
- 2017-02-28

每種版本陣列的可用資料來源操作

支援的操作/版本	2017-02-28	2018-05-29
AWS Lambda調用	是	是

支援的操作/版本	2017-02-28	2018-05-29
AWS Lambda BatchInvoke	是	是
None Datasource	是	是
亞馬遜 OpenSearch 獲取	是	是
亞馬遜 OpenSearch 郵政	是	是
亞馬遜 OpenSearch 認沽	是	是
亞馬遜 OpenSearch 刪除	是	是
亞馬遜 OpenSearch 獲取	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

注意：函數目前僅支援 2018-05-29 版本。

變更單位解析程式映射範本的版本

對於單位解析程式，版本會指定為要求映射範本的內容部分。若要更新版本，只要直接將 version 欄位更新成新版本。

例如，若要更新範本上的版AWS Lambda本：

```
{
    "version": "2017-02-28",
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

您需要將 2017-02-28 的版本欄位更新成 2018-05-29，如下所示：

```
{
    "version": "2018-05-29", ## Note the version
    "operation": "Invoke",
    "payload": {
        "field": "getPost",
        "arguments": $utils.toJson($context.arguments)
    }
}
```

變更函數的版本

對於函數，版本會透過函數物件上的 functionVersion 欄位指定。若要更新版本，可直接更新 functionVersion。注意：目前只有 2018-05-29 支援函數。

下面是可更新現有函數版本的 CLI 命令範例：

```
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：建議省略要求映射範本的版本欄位，因為其不會獲得接收。如果您在函數要求映射範本內指定版本，functionVersion 欄位的值將覆寫此版本值。

2018-05-29

行為變更

- 如果資料來源呼叫結果是 null，回應映射範本將會執行。
- 如果資料來源呼叫產生錯誤，您現在要負責處理錯誤，而回應映射範本評估結果將一律置放在 GraphQL 回應 data 區塊中。

推理

- null 呼叫結果具有意義，而且在某些應用程式使用案例中，我們可能希望以自訂方式處理 null 結果。例如，應用程式可以檢查記錄是否存在於 Amazon DynamoDB 資料表，以便執行一些授權檢查。在這種情況下，null 呼叫結果就能表示使用者應未獲得授權。執行回應映射範本現在會提供引發關於未經授權錯誤的能力。這個行為能為 API 設計師提供更好的控制功能。

假設我們使用下列的回應映射範本：

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

過去使用 2017-02-28 時，如果 $ctx.result 傳回 Null，回應映射範本將不會執行。現在使用 2018-05-29 時，我們可以處理此案例。例如，我們可以選擇引發授權錯誤如下：

```
# throw an unauthorized error if the result is null
#if ( $util.isNull($ctx.result) )
    $util.unauthorized()
#end
$util.toJson($ctx.result)
```

注意：從資料來源發出的錯誤有時並不嚴重，或甚至早有預期，這說明了為什麼回應映射範本應該可靈活處理呼叫錯誤，並決定是否予以忽略、重新引發，或是擲出不同的錯誤。

假設我們使用下列的回應映射範本：

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

過去使用 2017-02-28 時，如果發生呼叫錯誤，回應映射範本將會進行評估，而且結果會自動放置在 GraphQL 回應的 errors 區塊。現在使用 2018-05-29 時，我們可以選擇如何處理錯誤、重新引發錯誤、引發不同的錯誤，或在傳回資料時附加該錯誤。

再次引發呼叫錯誤

使用下列回應範本時，我們會引發資料來源發出的相同錯誤。

```
#if ( $ctx.error )
    $util.error($ctx.error.message, $ctx.error.type)
#end
$util.toJson($ctx.result)
```

如果發生呼叫錯誤 (例如，$ctx.error 存在時) 回應看起來將如下所示：

```
{
    "data": {
        "getPost": null
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "DynamoDB:ConditionalCheckFailedException",
            "message": "Conditional check failed exception...",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

引發不同的錯誤

使用下列回應範本時，我們會在處理資料來源發出錯誤之後引發自訂的錯誤。

```
#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)
```

如果發生呼叫錯誤 (例如，$ctx.error 存在時) 回應看起來將如下所示：

```
{
    "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
                }
            ]
        }
    ]
}
```

附加錯誤至傳回資料

使用下列回應範本時，當透過回應傳回資料時，我們會同時附加由資料來源發出的相同錯誤。這也稱為部分回應。

```
#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
```

如果發生呼叫錯誤 (例如，$ctx.error 存在時) 回應看起來將如下所示：

```
{
    "data": {
        "getPost": {
            "id": "1",
            "title: "A post"
        }
    },
    "errors": [
        {
            "path": [
                "getPost"
            ],
            "errorType": "ConditionalCheckFailedException",
            "message": "Conditional check failed exception...",
            "locations": [
                {
                    "line": 5,
                    "column": 5
                }
            ]
        }
    ]
}
```

從 2017-02-28 遷移到 2018-05-29

從 2017-02-28 遷移到 2018-05-29 的過程相當簡單。變更在解析程式要求映射範本或函數版本物件上的版本欄位。不過，請注意，2018-05-29 的執行行為不同於 2017-02-28，其中變更已概述於此處。

保留 2017-02-28 到 2018-05-29 的相同執行行為

在某些情況下，在執行 2018-05- 29 版本化範本時，可以保留與 2017-02-28 版本相同的執行行為。

範例：DynamoDB PutItem

提供下列 Dynam PutItem oDB 範本：

```
{
    "version" : "2017-02-28",
    "operation" : "PutItem",
    "key": {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "attributeValues" : {
        "baz" : ... typed value
    },
    "condition" : {
       ...
    }
}
```

並使用下列回應範本：

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

遷移到 2018-05-29，這些範本變更如下：

```
{
    "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" : {
       ...
    }
}
```

而且回應範本變更如下：

```
## 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)
```

現在，錯誤處理將由您負責，我們選擇使用 `$util.error()` 引發從 DynamoDB 傳回的相同錯誤。您可以調整這個片段，將您的映射範本轉換成 2018-05-29，請注意，如果您使用不同的回應範本，則您必須處理執行行為變更。

範例：DynamoDB GetItem

提供下列 Dynam GetItem oDB 範本：

```
{
    "version" : "2017-02-28",
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
        "bar" : ... typed value
    },
    "consistentRead" : true
}
```

並使用下列回應範本：

```
## map table attribute postId to field Post.id
$util.qr($ctx.result.put("id", $ctx.result.get("postId")))

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

遷移到 2018-05-29，這些範本變更如下：

```
{
    "version" : "2018-05-29", ## Note the new 2018-05-29 version
    "operation" : "GetItem",
    "key" : {
        "foo" : ... typed value,
```

```
        "bar" : ... typed value
    },
    "consistentRead" : true
}
```

而且回應範本變更如下：

```
## 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)
```

使用 2017-02-28 版本時，如果資料來源呼叫為 `null`，表示 DynamoDB 資料表中沒有符合索引鍵的項目，而回應映射範本將不會執行。這種做法應該適用於大部分的案例，但是如果您希望 `$ctx.result` 不要是 `null`，現在您必須負責處理該案例。

2017-02-28

特性

- 如果資料來源呼叫結果是 `null`，回應映射範本將不會執行。
- 如果資料來源呼叫產生錯誤，您現在要負責處理錯誤，而回應映射範本會執行，且評估結果會置放在 GraphQL 回應 `errors.data` 區塊中。

類型參考

本節用作結構描述類型的參考。

標量類型AWS AppSync

GraphQL 物件類型具有名稱和欄位，而這些欄位可以有子欄位。最終，對象類型的字段必須解析為純量類型，代表查詢的葉子。如需有關物件類型和純量的詳細資訊，請參閱結構描述和類型在圖形 SQL 網站上。

除了預設的 GraphQL 純量集之外，AWS AppSync也可以讓您使用服務定義以開頭的標量AWS前綴。AWS AppSync不支持創建使用者（自定義）標量。您必須使用預設值或AWS標量。

您不能使用AWS做為自訂物件類型的字首。

下一節是結構描述類型的參考。

默認純量

GraphQL 定義了下列預設純量：

默認純量列表

`ID`

物件的唯一識別元。這個標量是序列化的`String`但並不意味著是人類可讀的。

`String`

一個 UTF-8 字元序列。

`Int`

介於-(2) 之間的整數值^{31}) 及^{31}-1.

`Float`

一個浮點數值。

`Boolean`

布林值，`true` 或 `false`。

AWS AppSync純量

AWS AppSync定義了以下純量：

AWS AppSync純量列表

`AWSDate`

一個擴展日期字符串的格式`YYYY-MM-DD`。

`AWSTime`

一個擴展国际标准时间字符串的格式`hh:mm:ss.sss`。

`AWSDateTime`

一個擴展ISO 8601 日期和時間字符串的格式`YYYY-MM-DDThh:mm:ss.sssZ`。

> **Note**
>
> 該`AWSDate`,`AWSTime`，以及`AWSDateTime`標量可以選擇包括時區偏移。例如，值`1970-01-01Z`,`1970-01-01-07:00`，以及`1970-01-01+05:30`都是有效的`AWSDate`。時區偏移量必須是`Z`（UTC）或以小時和分鐘為單位的偏移量（以及可選的秒）。例如：`±hh:mm:ss`。即使不是 ISO 8601 標準的一部分，時區偏移量中的秒數字段也被認為是有效的。

`AWSTimestamp`

表示之前或之後的秒數的整數值`1970-01-01-T00:00Z`。

`AWSEmail`

格式為的電子郵件地址`local-part@domain-part`定義為RFC。

`AWSJSON`

一個 JSON 字符串。任何有效的 JSON 結構都會在解析器程式碼中自動剖析並載入為映射、清單或純量值，而不是作為文字輸入字串。未加引號的字串或其他無效的 JSON 會導致 GraphQL 驗證錯誤。

AWSPhone

電話號碼。此值會儲存為字串。電話號碼可以包含空格或連字號，以分隔數字群組。沒有國家/地區代碼的電話號碼被假定為堅持美國/北美號碼[北美編號計劃](#)。

AWSURL

由定義的網址[RFC 1738](#)。例如 `https://www.amazon.com/dp/B000NZW3KC/` 或 `mailto:example@example.com`。URL 必須包含結構描述 (`http`,`mailto`) 且不能包含兩個正斜線 (`//`) 中的路徑部分。

AWSIPAddress

有效的 IPv4 或 IPv6 位址。IPv4 位址預期會以四點符號 (`123.12.34.56`). IPv6 位址應採用非括號、冒號分隔的格式 (`1a2b:3c4b::1234:4567`). 您可以包含選用的 CIDR 字尾 (`123.45.67.89/16`) 表示子網路遮罩。

綱要用法範例

下列範例 GraphQL 結構描述會使用所有自訂純量做為「物件」，並顯示基本 put、get 和清單作業的解析器要求和回應範本。最後，這個例子顯示了如何在運行查詢和突變時使用它。

```
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
}
```

這是一個請求模板putObject可能看起來像。一個putObject使用一個PutItem在您的 Amazon DynamoDB 表格中建立或更新項目的作業。請注意，此程式碼片段沒有已設定的 Amazon DynamoDB 表格做為資料來源。這僅用作示例：

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

的回應範本putObject返回結果：

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

這是一個請求模板getObject可能看起來像。一個getObject使用一個GetItem操作返回給定主鍵的項目的一組屬性。請注意，此程式碼片段沒有已設定的 Amazon DynamoDB 表格做為資料來源。這僅用作示例：

```
{
    "version": "2017-02-28",
    "operation": "GetItem",
```

```
    "key": {
        "id": $util.dynamodb.toDynamoDBJson($ctx.args.id),
    }
}
```

的回應範本getObject返回結果：

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

這是一個請求模板listObjects可能看起來像。一個listObjects使用一個Scan操作返回一個或多個項目和屬性。請注意，此程式碼片段沒有已設定的 Amazon DynamoDB 表格做為資料來源。這僅用作示例：

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

的回應範本listObjects返回結果：

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

以下是將此結構描述與 GraphQL 查詢搭配使用的一些範例：

```
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
    }
}
```

圖形 QL 中的介面和聯集

圖形 QL 類型系統支援介面。界面會公開特定的欄位組合，類型必須包含該組合以實作界面。

圖形 QL 類型系統也支援工会。聯集與界面相同，但它們不會定義一組常用的欄位。當可能的類型不共用邏輯階層時，使用者通常偏好使用聯集而非界面。

下一節是結構描述類型的參考。

介面範例

我們可以代表Event界面，代表任何類型的活動或聚會的人。一些可能的事件類型包括Concert,Conference，以及Festival。這些類型都具有常見的特性，包含名稱、事件進行的地點，以及開始和結束日期。這些類型也有差異; 和Conference提供演講者和研討會的列表，而Concert設有一個表演樂隊。

在結構定義語言 (SDL) 中，Event接口定義如下：

```
interface Event {
        id: ID!
        name : String!
        startsAt: String
        endsAt: String
        venue: Venue
        minAgeRestriction: Int
}
```

並且每個類型都實現了Event接口如下：

```
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]
}
```

界面可用來代表元素，而該元素可能有幾個類型。例如，我們可以搜尋在特定會場發生的所有事件。現在讓我們在結構描述上新增 findEventsByVenue 欄位，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]
}
```

該findEventsByVenue返回一個列表Event。由於 GraphQL 界面欄位對於所有實作類型而言都是共通的，因此您可以在 Event 界面 (id、name、startsAt、endsAt、venue 和 minAgeRestriction) 選取任何欄位。此外，您可以使用 GraphQL 片段存取任何實作類型上的欄位，只是您必須指定該類型。

讓我們來看看使用介面的 GraphQL 查詢範例。

```
query {
  findEventsAtVenue(venueId: "Madison Square Garden") {
    id
    name
    minAgeRestriction
    startsAt

    ... on Festival {
      performers
    }

    ... on Concert {
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
```

```
}
```

上述查詢會產生一個結果清單，伺服器預設會依據開始日期排序事件。

```
{
  "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"
        ]
      }
    ]
  }
}
```

由於結果會以單一事件集合的形式傳回，因此使用介面來表示共同特性對於排序結果非常有幫助。

聯盟範例

如前所述，工會不定義常見的字段集。搜尋結果可能代表許多不同的類型。使用 Event 結構描述，您可以定義 SearchResult 聯集，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
```

在這種情況下，要查詢我們的任何字段SearchResult聯合，你必須使用片段：

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
      id
      name
      performers
    }

    ... on Concert {
      id
      name
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

輸入解析度AWS AppSync

類型解析是一種機制，GraphQL 引擎會透過這個機制將解析值識別為特定物件類型。

回到聯合搜索示例，如果我們的查詢產生了結果，則結果列表中的每個項目都必須將自己顯示為可能的類型之一SearchResult聯集定義（也就是說，Conference,Festival,Concert，或Venue).

由於識別 Festival、Venue 或 Conference 的邏輯取決於應用程式要求，因此必須為 GraphQL 引擎提供提示，以便從原始結果中識別我們的類型。

同AWS AppSync，這個提示由一個名為的元字段表示__typename，其值對應於已識別的物件類型名稱。__typename對於接口或聯集的返回類型是必需的。

類型解析度範例

讓我們重複使用之前的結構描述。您可以透過瀏覽到主控台並將以下項目新增至 Schema (結構描述) 頁面下來執行此動作：

```
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
}
```

現在將解析程式附加到 Query.search 欄位。在Resolvers區段中，選擇貼附，建立新的資料來源的類型沒有，然後命名StubDataSource。在此範例中，我們將假設從外部來源擷取結果，並將擷取結果硬式編碼至請求映射範本中。

在請求映射範本窗格中，輸入以下內容：

```
{
```

```
    "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"]
        }
    ]
}
```

如果應用程式傳回類型名稱做為id欄位中，型別解析邏輯必須剖析id用於擷取類型名稱的欄位，然後新增__typename每個結果的字段。您可以在回應映射範本中執行該邏輯，如下所示：

> **Note**
>
> 如果您使用的是 Lambda 資料來源，也可以將此工作做為 Lambda 函數的一部分執行。

```
#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)
```

執行下列查詢：

```
query {
  search(query: "Madison") {
    ... on Venue {
      id
      name
      address
    }

    ... on Festival {
        id
      name
      performers
    }

    ... on Concert {
      id
      name
      performingBand
    }

    ... on Conference {
      speakers
      workshops
    }
  }
}
```

此查詢會產生下列結果：

```
{
  "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"
        ]
      }
    ]
  }
}
```

類型解析邏輯會依據應用程式而有不同。例如，您可以有不同的識別邏輯，以檢查特定欄位或甚至結合欄位是否存在。也就是說，您可以偵測 performers 欄位是否存在以識別 Festival，或偵測 speakers 與 workshops 欄位的組合是否存在以識別 Conference。最終，它是由你來定義你想要使用的邏輯。

故障診斷與常見錯誤

本區段說明一些常見的錯誤，以及如何排除這些問題。

不正確的 DynamoDB 索引鍵映射

如果 GraphQL 作業傳回下列錯誤訊息，可能是因為您的請求對應範本結構與 Amazon DynamoDB 金鑰結構不符：

```
The provided key element does not match the schema (Service: AmazonDynamoDBv2; Status
 Code: 400; Error Code
```

例如，如果您的 DynamoDB 資料表具有名為的雜湊索引鍵`"PostID"`，`"id"`且範本如下列範例所示，則會導致上述錯誤，因為`"id"`不符合。`"PostID"`

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

缺少解析程式

如果您執行查詢等 GraphQL 操作，但是獲得 null 回應，這可能是因為您並未設定解析程式。

如果匯入的結構描述定義了 `getCustomer(userId: ID!):` 欄位，但並未針對此欄位設定解析程式，則當您執行像是 `getCustomer(userId:"ID123"){...}` 的查詢時，將會獲得類似於下列的回應：

```
{
    "data": {
    "getCustomer": null
    }
}
```

映射範本錯誤

如果未正確設定映射範本，您將會收到 GraphQL 回應，其中的 `errorType` 為 `MappingTemplate`。`message` 欄位應該會指出映射範本的問題所在。

例如，如果請求映射範本未包含 `operation` 欄位，或是 `operation` 欄位的名稱不正確，則您將會收到類似下列的回應：

```
{
    "data": {
        "searchPosts": null
    },
    "errors": [
        {
        "path": [
            "searchPosts"
        ],
        "errorType": "MappingTemplate",
        "locations": [
            {
            "line": 2,
            "column": 3
            }
        ],
        "message": "Value for field '$[operation]' not found."
        }
    ]
}
```

不正確的傳回類型

資料來源所傳來的傳回類型，必須符合結構描述中所定義的物件類型，否則將會出現 GraphQL 錯誤，例如：

```
"errors": [
    {
    "path": [
        "posts"
    ],
    "locations": null,
```

```
    "message": "Can't resolve value (/posts) : type mismatch error, expected type LIST,
 got OBJECT"
    }
]
```

例如，下列的查詢定義可能會產生這種錯誤：

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

查詢式期望的結果是 [Posts] 物件的清單 (LIST)。例如，如果有使用 Node.JS 的 Lambda 函式，包含類似於下列的程式碼：

```
const result = { data: data.Items.map(item => { return item ; }) };
callback(err, result);
```

這將會丟出錯誤，因為 result 是一個物件。您需要將回呼變更為 result.data，或是將結構描述改為不傳回清單 (LIST)。

處理無效請求

當AWS AppSync無法處理並將請求（由於語法無效等不正確的數據）發送到字段解析器時，響應有效負載將返回值設置為的字段數據以null及任何相關錯誤。

本文為英文版的機器翻譯版本，如內容有任何歧義或不一致之處，概以英文版為準。