第一章

歡迎使用 GraphQL

Tim Berners-Lee 被英國女皇冊封為騎士之前是位程式員。他當時在瑞士的 CERN 歐洲粒子物理實驗室任職，周圍都是才華洋溢的研究人員。Berners-Lee 希望協助同事們分享心得，所以想要建立一個網路來讓科學家張貼與更新資訊。最終這個專案在 1990 年 12 月幫 CERN（*https://www.w3.org/People/Berners-Lee/Longer.html*）做出史上第一台網路伺服器與第一台網路用戶端，以及“WorldWideWeb”瀏覽器（之後稱為“Nexus”）。

Berners-Lee 的專案可讓研究人員在自己的電腦觀看與更新網頁內容。“WorldWideWeb”包括 HTML、URL、瀏覽器與用來更新內容的 WYSIWYG 介面。

現在的網際網路並非只是瀏覽器內的 HTML，它包括筆電、手錶、智慧型手機，也包括在滑雪纜椅券裡面的無線射頻辨識（RFID）晶片，甚至包括當你外出時幫忙餵貓的機器人。

現今的用戶端比以前多，但我們仍然希望做到同樣的事情：盡快載入資料。使用者的高標準讓應用程式必須保持高效。他們希望 app 在任何情況下都可良好運行——從功能手機的 2G 網路到大螢幕桌機的超快速光纖網際網路。快速的 app 可讓更多人與我們的內容輕鬆互動。快速的 app 可讓使用者開心，並且讓我們發大財。

將伺服器的資料快速且可靠地送到用戶端是網路的重點，無論過去、現在與未來都是如此。雖然本書經常提到從前的背景，但我們討論的重點是現代的解決方案。所以我們接下來要談談未來，談談 GraphQL。

什麼是 GraphQL ？

GraphQL（*https://www.graphql.org/*）是供 API 使用的查詢語言。它也是滿足資料查詢的 runtime。GraphQL 服務不規定使用哪種傳輸協定，但通常是透過 HTTP 來提供的。

為了展示 GraphQL query[譯註] 與它的回應，請看一下 SWAPI（*https://graphql.org/swapi-graphql/*），也就是 Star Wars API。SWAPI 是與 GraphQL 包在一起的表現層狀態轉換（Representational State Transfer，REST）API。我們可以用它來傳送 query 及接收資料。

GraphQL query 只會要求它需要的資料。圖 1-1 是個 GraphQL query 範例，query 在左邊，在此我們索取 Princess Leia 的個人資料。由於我們指明索取第五人的資料（`personID:5`），所以得到 Leia Organa 的紀錄。接下來，我們指定取得資料的三個欄位：`name`、`birthYear` 與 `created`。右邊是回應：它按照 query 的外形（shape）將 JSON 資料格式化了。這個回應只含有我們需要的資料。

```
1 query {
2   person(personID:5) {
3     name
4     birthYear
5     created
6   }
7 }
```

```
{
  "data": {
    "person": {
      "name": "Leia Organa",
      "birthYear": "19BBY",
      "created": "2014-12-10T15:20:09.791000Z"
    }
  }
}
```

圖 1-1　用 Star Wars API 來查詢個人資料

因為查詢的動作是互動的，所以我們接下來可以修改它，以查看新的結果。如果我們加入欄位 `filmConnection`，就可以索取有 Leia 的電影名稱，如圖 1-2 所示。

[譯註] 為了協助閱讀，本書將動詞的 query 譯為“查詢”，將名詞的 query 保持原狀，因為它在 GraphQL 中的關鍵字也是 query。GraphQL 的另兩項操作——mutation 和 subscription 亦同。

```
1  query {
2    person(personID:5) {
3      name
4      birthYear
5      created
6      filmConnection {
7        films {
8          title
9        }
10     }
11   }
12 }
```

```
{
  "data": {
    "person": {
      "name": "Leia Organa",
      "birthYear": "19BBY",
      "created": "2014-12-10T15:20:09.791000Z",
      "filmConnection": {
        "films": [
          {
            "title": "A New Hope"
          },
          {
            "title": "The Empire Strikes Back"
          },
          {
            "title": "Return of the Jedi"
          },
          {
            "title": "Revenge of the Sith"
          },
          {
            "title": "The Force Awakens"
          }
        ]
      }
    }
  }
}
```

圖 1-2　連結查詢

這個 query 是嵌套狀的，它被執行時可遍歷相關的物件，如此一來，我們就可以用單一 HTTP 請求來取得兩種類型的資料，而不需要為了查詢多個物件而反覆執行多次。我們不會收到與這些類型有關但不想要取得的資料。使用 GraphQL 時，用戶端可以用一個請求指令取得他們需要的所有資料。

當你對 GraphQL 伺服器執行 query 時，它會用一種型態系統（type system）來驗證 query。每一個 GraphQL 服務都會在 GraphQL schema（綱要）裡面定義許多型態。你可以將型態系統視為 API 的資料藍圖，這個藍圖的基礎是你所定義的一系列物件。例如，稍早的人物 query 的基礎是 Person 物件：

```
type Person {
    id: ID!
    name: String
    birthYear: String
    eyeColor: String
```

```
    gender: String
    hairColor: String
    height: Int
    mass: Float
    skinColor: String
    homeworld: Planet
    species: Species
    filmConnection: PersonFilmsConnection
    starshipConnection: PersonStarshipConnection
    vehicleConnection: PersonVehiclesConnection
    created: String
    edited: String
}
```

Person 型態定義了所有欄位及其型態，讓你可以在查詢 Princess Leia 時取得它們。第三章會更深入討論 schema 與 GraphQL 的型態系統。

GraphQL 通常被稱為**宣告式**（*declarative*）資料擷取語言，意思是開發者只要根據他們需要**哪些**資料來列出資料需求，而不需要把注意力放在該**如何**取得它。市面上有供各種語言使用的 GraphQL 伺服器程式庫，包括 C#、Clojure、Elixir、Erlang、Go、Groovy、Java、JavaScript、.NET、PHP、Python、Scala 與 Ruby[1]。

在本書中，我們的重點是如何用 JavaScript 來建構 GraphQL 服務。本書討論的所有技術都適用於任何語言寫成的 GraphQL。

GraphQL 規格

GraphQL 是一種用戶端 / 伺服器通訊規格（spec）。什麼是規格？規格描述了某種語言的功能與特性。我們都受益於語言規格，因為它提供了共通的詞彙與最佳用法，讓語言的使用社群得以依循。

ECMAScript 規格是一種著名的軟體規格。每隔一段時間，一群來自瀏覽器公司、科技公司和語言社群的代表都會聚在一起，討論應該在 ECMAScript 規格中加入（與移除）哪些東西。GraphQL 也是如此，有一群人會聚在一起討論應納入這種語言（與從中移除）的東西，這份規格是所有 GraphQL 實品的指南。

1 見 *https://graphql.org/code/* 的 GraphQL Server Libraries。

當規格發表時，GraphQL 的創造者也會分享一個以 JavaScript 寫成的參考成品——graphql.js（*https://github.com/graphql/graphql-js*）。你可以將這個參考成品當成藍圖，但它的目的不包括規定你一定要用某種語言來實作服務，它只是個指南。瞭解這種查詢語言與型態系統之後，你可以用你喜歡的任何一種語言來建立服務。

如果規格與實作是兩回事，那規格究竟規定些什麼？規格敘述編寫 query 時應使用的語言與語法，也規定一個型態系統及型態系統的執行和驗證引擎。除了以上的規定之外，規格不限制任何東西，GraphQL 未規定該使用哪種語言、如何儲存資料、該支援哪些用戶端。查詢語言有一份指南，但如何實際設計你的專案完全由你自己決定（如果你想要瞭解整個情況，可研究文件（*http://facebook.github.io/graphql/*））。

GraphQL 的設計準則

雖然 GraphQL 不限制建構 API 的方式，但它提供了一些構思服務[2] 的指南：

階層式

GraphQL query 是階層式的，query 有一些欄位在其他欄位裡面，且 query 的外形類似它回傳的資料。

以產品為中心

GraphQL 的目的是提供用戶端需要的資料，以及支援用戶端的語言和 runtime。

強型態

GraphQL 伺服器採取 GraphQL 型態系統。在 schema 內，每一個資料點都有一種特定的型態，GraphQL 會拿它來驗證資料點。

由用戶端指定的 *query*

GraphQL 伺服器提供用戶端可以使用的功能。

自我查詢

GraphQL 語言可查詢 GraphQL 伺服器的型態系統。

初步瞭解 GraphQL 規格是什麼之後，接著來看一下它為什麼會問世。

2 見 2018 年 6 月的 GraphQL Spec（*http://facebook.github.io/graphql/June2018/#sec-Overview*）。

GraphQL 的緣起

在 2012 年，Facebook 決定重建 app 的原生行動 app，當時這間公司的 iOS 與 Android app 只不過是將行動網站的畫面稍微包裝起來而已。Facebook 有個 RESTful 與 FQL（Facebook 的 SQL 版本）資料表，當時它效能低下，且 app 經常當機，工程師認為他們需要改善將資料送給用戶端 app 的方式[3]。

Lee Byron、Nick Schrock 與 Dan Schafer 的團隊決定站在用戶端的角度重新考慮他們的資料。他們著手建構了 GraphQL，用這種查詢語言來描述其公司的用戶端 / 伺服器 app 的資料模型其功能與需求。

這個團隊在 2015 年 7 月發表第一版的 GraphQL 規格，以及以 JavaScript 寫成的 GraphQL 參考成品 —— graphql.js。GraphQL 在 2016 年 9 月正式脫離“技術預審”階段，這代表 GraphQL 已經是官方的準成品，雖然它已經在 Facebook 中實際使用多年了。GraphQL 現在負責處理幾乎所有 Facebook 的資料擷取工作，且 IBM、Intuit、Airbnb 與許多其他公司的產品也使用它。

資料傳輸歷史

GraphQL 代表一種極新穎的概念，但你也要瞭解資料傳輸的歷史背景。當我們談到資料傳輸時，都會試著釐清如何在電腦之間來回傳遞資料。我們會向遠端系統請求一些資料，並期望收到一個回應。

遠端程序呼叫

遠端程序呼叫（RPC）是在 1960 年代發明的。RPC 是由用戶端發起的，它會傳送一個請求訊息給遠端的電腦，要求它做某些事情。遠端電腦會傳送一個回應給用戶端。那些電腦與現今的用戶端和伺服器不一樣，但資訊的流動基本上是相同的：由用戶端請求一些資料，再從伺服器取得回應。

3 見 Dan Schafer 與 Jing Chen 一起演說的“Data Fetching for React Applications”，*https://www.youtube.com/watch?v=9sc8Pyc51uU*。

簡易物件存取通訊協定

Microsoft 在 1990 年代末期做出簡易物件存取協定（Simple Object Access Protocol，SOAP）。SOAP 使用 XML 來為傳輸用的訊息與 HTTP 編碼。SOAP 也使用一種型態系統，並引入“資源導向資料索取”的概念。SOAP 可提供可預測性極高的結果，但因為 SOAP 的做法相當複雜，所以造成一些麻煩。

REST

REST 應該是你目前最熟悉的 API 架構。REST 是 Roy Fielding 於 2000 年在加州大學爾灣分校發表的博士論文（*http://bit.ly/2j4SIKI*）中定義的。他描述了一個資源導向的架構，可讓使用者在這個架構中執行諸如 GET、PUT、POST 與 DELETE 等動作來處理網路資源。你可以將資源組成的網路當成一種**虛擬狀態機**，而那些動作（GET、PUT、POST、DELETE）是這個機制內的狀態改變。我們現在或許將它視為理所當然，但當時這是相當大的進步。（對了，Fielding 也拿到博士學位了。）

在 RESTful 架構中，路由（route）代表資訊。例如，向這些路由請求資訊可得到特定的回應：

```
/api/food/hot-dog
/api/sport/skiing
/api/city/Lisbon
```

REST 可讓我們用各種端點來建構資料模型，這種做法比之前的架構簡單多了。它提供了可在日漸複雜的網路環境中處理資料的新方法，卻不強制規定使用特定的資料回應格式。最初，REST 是與 XML 一起使用的。AJAX 原本是 Asynchronous JavaScript And XML 的縮詞，因為當時發出 Ajax 請求後得到的回應資料會被格式化成 XML（它現在是個獨立的單字，寫成“Ajax”）。這為網路開發者平添一個痛苦的步驟：為了在 JavaScript 中使用資料，他們必須先解析 XML 回應。

不久之後，Douglas Crockford 開發出 JavaScript Object Notation（JSON）並且將它標準化。JSON 不強制使用特定語言，它提供一種優雅的資料格式，讓許多不同的語言都可以解析和使用。Crockford 也著作 *JavaScript: The Good Parts*（*http://bit.ly/js-good-parts*）（O’Reilly，2008），讓我們知道 JSON 是很棒的成員。

REST 的影響是不可忽視的，它已經被用來建構無數的 API 了，各個軟體層面的開發者都曾經獲得它的幫助，有一些粉絲甚至喜歡討論“什麼是 RESTful，什麼不是”，以致於被稱為 *RESTafarians*。既然如此，為何 Byron、Schrock 與 Schafer 要踏上開創新事物的旅程？我們可以在 REST 的缺陷中尋找答案。

REST 的缺陷

當 GraphQL 第一次發表時，有些人吹捧它是 REST 的替代品。早期的採用者大喊“REST 已死！”，鼓勵我們將 REST API 丟在一旁。這是提升部落格點擊率與會議開場的好標題，但將 GraphQL 描繪成 REST 的殺手太過簡化了。比較深層的原因是，隨著網路的發展，REST 在某些狀況下已經顯露一些疲態了。GraphQL 正是為了緩解這些疲態而造就的。

Overfetch

假如我們要建立一個 app，並讓它使用 REST 版的 SWAPI 提供的資料。我們要先載入 1 號角色 Luke Skywalker 的一些資料[4]，為了取得這項資訊，我們可以向 *https://swapi.co/api/people/1/* 發出 GET 請求，得到的回應是這筆 JSON 資料：

```
{
  "name": "Luke Skywalker",
  "height": "172",
  "mass": "77",
  "hair_color": "blond",
  "skin_color": "fair",
  "eye_color": "blue",
  "birth_year": "19BBY",
  "gender": "male",
  "homeworld": "https://swapi.co/api/planets/1/",
  "films": [
    "https://swapi.co/api/films/2/",
    "https://swapi.co/api/films/6/",
    "https://swapi.co/api/films/3/",
    "https://swapi.co/api/films/1/",
    "https://swapi.co/api/films/7/"
  ],
  "species": [
    "https://swapi.co/api/species/1/"
  ],
```

4 請留意，這項 SWAPI 資料並未包含最新的星際大戰電影。

```
    "vehicles": [
      "https://swapi.co/api/vehicles/14/",
      "https://swapi.co/api/vehicles/30/"
    ],
    "starships": [
      "https://swapi.co/api/starships/12/",
      "https://swapi.co/api/starships/22/"
    ],
    "created": "2014-12-09T13:50:51.644000Z",
    "edited": "2014-12-20T21:17:56.891000Z",
    "url": "https://swapi.co/api/people/1/"
  }
```

這是個龐大的回應，遠比 app 需要的資料還要多。我們只需要 name、mass 與 height 的資訊：

```
{
  "name": "Luke Skywalker",
  "height": "172",
  "mass": "77"
}
```

這是明顯的 *overfetch*（**過度擷取**）案例——取得許多不需要的資料。用戶端只需要三個資料點，卻得到包含 16 個鍵的物件，而且是透過網路傳送不需要的資訊。

在 GraphQL app 中，這個請求又是如何？我們仍然想要取得 Luke Skywalker 的 name、height 與 mass，如圖 1-3 所示。

```
query {
  person(personID:1) {
    name
    height
    mass
  }
}
```

```
{
  "data": {
    "person": {
      "name": "Luke Skywalker",
      "height": 172,
      "mass": 77
    }
  }
}
```

圖 1-3　Luke Skywalker query

左邊是我們發出的 GraphQL query，它只索取需要的欄位。右邊是收到的 JSON 回應，這一次它只有我們索取的資料，完全沒有需要經過基地台傳到手機的其他 13 個欄位。我們索取指定外形的資料，並收到那種外形的資料，不多也不少。這是比較宣告式的做法，因為不會收到無關的額外資料，所以可以更快速地得到回應。

Underfetch

我們的專案經理想要在 Star Wars app 中加入另一個功能，除了 name、height 與 mass 之外，也要顯示有 Luke Skywalker 這位角色的電影名單。當我們向 *https://swapi.co/api/people/1/* 請求資料之後，仍然需要發出其他的請求來取得更多資料，這代表我們 *underfetch*（擷取不足）。

為了取得每部電影的名稱，我們必須向 films 陣列內的每一個路由索取資料：

```
"films": [
  "https://swapi.co/api/films/2/",
  "https://swapi.co/api/films/6/",
  "https://swapi.co/api/films/3/",
  "https://swapi.co/api/films/1/",
  "https://swapi.co/api/films/7/"
]
```

我們要發出一個請求來索取 Luke Skywalker（`https://swapi.co/api/people/1/`），接著還要發出五個請求來取得每一部電影才能得到這一筆資料。索取每部電影時，我們也會得到一個大型的物件，而且這一次我們只想要一個值。

```
{
"title": "The Empire Strikes Back",
"episode_id": 5,
"opening_crawl": "...",
"director": "Irvin Kershner",
"producer": "Gary Kurtz, Rick McCallum",
"release_date": "1980-05-17",
"characters": [
  "https://swapi.co/api/people/1/",
  "https://swapi.co/api/people/2/",
  "https://swapi.co/api/people/3/",
  "https://swapi.co/api/people/4/",
  "https://swapi.co/api/people/5/",
  "https://swapi.co/api/people/10/",
  "https://swapi.co/api/people/13/",
  "https://swapi.co/api/people/14/",
  "https://swapi.co/api/people/18/",
```

```
    "https://swapi.co/api/people/20/",
    "https://swapi.co/api/people/21/",
    "https://swapi.co/api/people/22/",
    "https://swapi.co/api/people/23/",
    "https://swapi.co/api/people/24/",
    "https://swapi.co/api/people/25/",
    "https://swapi.co/api/people/26/"
  ],
  "planets": [
        //... 長串的路由
    ],
  "starships": [
        //... 長串的路由
    ],
  "vehicles": [
        //... 長串的路由
    ],
  "species": [
        //... 長串的路由
    ],
  "created": "2014-12-12T11:26:24.656000Z",
  "edited": "2017-04-19T10:57:29.544256Z",
  "url": "https://swapi.co/api/films/2/"
  }
```

如果我們想要列出電影的每位角色，就要發出更多的請求，在這個例子中，我們還要接觸 16 個路由，並產生 16 次的用戶端往返。每一個 HTTP 請求都會用到用戶端的資源，並過度擷取資料，造成更緩慢的使用者體驗，特別是使用慢速網路或裝置的使用者可能會完全無法看到內容。

GraphQL 處理擷取不足的方法是定義一個嵌套式的 query，接著一次請求所有的資料，如圖 1-4 所示。

```
1  query {
2    person(personID:1) {
3      name
4      height
5      mass
6      filmConnection {
7        films {
8          title
9        }
10     }
11   }
12 }
```

```
{
  "data": {
    "person": {
      "name": "Luke Skywalker",
      "height": 172,
      "mass": 77,
      "filmConnection": {
        "films": [
          {
            "title": "A New Hope"
          },
          {
            "title": "The Empire Strikes Back"
          },
          {
            "title": "Return of the Jedi"
          },
          {
            "title": "Revenge of the Sith"
          },
          {
            "title": "The Force Awakens"
          }
        ]
      }
    }
  }
}
```

圖 1-4　電影連結

我們只用一個請求來取得想要的資料。而且一如既往，query 的外形符合所收到資料的外形。

管理 REST 端點

眾人經常抱怨 REST API 的另一個地方是它缺乏彈性。當用戶端的需求改變時，你通常要建立新的端點，而且新端點可能會開始快速倍增。套句 Oprah 說過的名言：“You get a route! You get a route! Every! Body! Gets! A! Route!”。

使用 SWAPI REST API 時，我們必須對許多路由發出請求，大型的 app 通常會使用自訂的端點來盡量減少 HTTP 請求，你可能會開始看到 `/api/character-with-movie-title` 這類的端點。設置新端點通常意味著前端與後端團隊必須進一步擬定計畫與進行更多的溝通，開發速度或許會因而減緩。

使用 GraphQL 時，典型的架構只有一個端點。單一端點可扮演閘道的角色並協調多個資料來源，但就算只有一個端點，我們仍然可以更輕鬆地組織資料。

在討論 REST 的缺陷時，有一個需要特別注意的地方在於許多機構都會同時使用 GraphQL 與 REST。設置 GraphQL 端點來從 REST 端點擷取資料是完全有效的 GraphQL 使用方式。在你的機構中逐漸漸進使用 GraphQL 是一種很好的做法。

現實世界的 GraphQL

許多公司都用 GraphQL 來改善它們的 app、網站與 API，GitHub 是早期就大量採用 GraphQL 的公司之一，它的 REST API 經歷三次變動，在公用 API 第 4 版時開始使用 GraphQL，GitHub 在官網（*https://developer.github.com/v4/*）提到，他們發現“能夠準確定義你需要的資料（而且只有你需要的）是它比 REST API v3 端點還要好的地方”。

其他的公司，例如 *The New York Times*、IBM、Twitter 與 Yelp 也信任 GraphQL，這些團隊的開發者也經常在會議上宣揚 GraphQL 的好處。

目前至少有三個專門以 GraphQL 為主題的會議：舊金山的 GraphQL Summit、赫爾辛基的 GraphQL Finland 與柏林的 GraphQL Europe。GraphQL 社群目前仍然藉由區域性的聚會與各種軟體會議持續成長中。

GraphQL 用戶端

我們已經多次談到，GraphQL 只是一種規格。它不在乎與它一起使用的究竟是 React 還是 Vue、或是 JavaScript，甚至是瀏覽器。GraphQL 對一些特定的主題有一些意見，但除此之外，如何設計架構由你自行決定。這導致一些工具的出現，讓你在規格之外的領域有一些選擇。接著討論 GraphQL 用戶端。

GraphQL 用戶端旨在加速開發團隊的工作流程，並改善 app 的效率與性能。它們可處理諸如網路請求、資料快取，以及將資料注入使用者介面等工作。市面上有許多 GraphQL 用戶端選項，但這個領域的領導者是 Relay（*https://facebook.github.io/relay/*）與 Apollo（*https://www.apollographql.com/*）。

Relay 是 Facebook 製作的用戶端，它是與 React 和 React Native 一起運作的。Relay 是 React 元件與 GraphQL 伺服器回傳的資料之間的連接組織。Facebook、GitHub、Twitch 以及許多其他公司都使用 Relay。

Apollo Client 是社群團體 Meteor Development Group 為了建立更全面的 GraphQL 周邊工具而開發的。Apollo Client 支援所有主要的前端開發平台，不限定任何特定的框架。Apollo 也開發了一些協助建構 GraphQL 服務和改善後端服務效能的開發工具，以及監控 GraphQL API 性能的工具組。Airbnb、CNBC、*The New York Times* 與 Ticketmaster 等公司都在產品中使用 Apollo Client。

這是個龐大且正在持續成長的生態系統，不過好消息是 GraphQL 已經是個相當穩定的標準了。接下來的章節將討論如何編寫 schema 與建立 GraphQL 伺服器。本書的 GitHub 存放區有一些學習資源可以在過程中協助你：*https://github.com/moonhighway/learning-graphql/*。你可以在那裡找到實用的連結、範例，以及按章整理的專案檔案。

在深入探討如何使用 GraphQL 之前，我們要討論一些關於圖論（graph theory）的知識，以及 GraphQL 底層概念的豐富歷史。