前言

建構一個 API 開發者平台讓數百萬人使用，絕對是你軟體生涯中最有挑戰性、最刺激的工作之一，本書將教你具體的做法。

API 是現代軟體開發的核心，它可以解決基本的開發者難題。我，身為一位軟體工程師，如何將寫好的程式公開給其他開發者，讓他們用來創新？在現代世界中，建構軟體很像構築樂高積木。身為開發者的你可以透過 API 來使用付款、通訊、授權與身分驗證等服務。在建構新軟體時，身為軟體工程師的你可以用這些 API 來編寫新產品，重複使用別人建構的程式碼，以節省時間並且避免重新發明車輪。

很多小時候喜歡玩樂高的軟體工程師現在同樣喜歡玩這種玩具。誰不喜歡呢？樂高很有趣，你可以把神奇的彩色積木組裝起來，做出別緻的作品。但如果你也可以建構樂高本身呢？如果你不但可以發明新的樂高組法，也可以發明樂高積木本身，讓別人用它們來創新，豈不美哉？當你建構自己的 API 時，其實就是在建構自己的樂高積木讓別的開發者使用。

API 不是最近才有的電腦科學概念，早在 60 年代，開發者就開始為第一批程序語言建構標準的程式庫，並分享給別的開發者了，這些程式庫可讓別的開發者直接使用它的標準功能，而不需要瞭解內部的程式碼。

接著，在 70 與 80 年代，隨著網路電腦的興起，第一批網路 API 出現了，它們可讓開發者透過遠端程序呼叫（RPCs）來使用它們的服務。開發者可以透過 RPCs 以網路公開他們寫好的功能，也可以像呼叫本地程式庫一樣呼叫遠端的程式庫。Java 這類的程式語言更提供了進一步的抽象與複雜度，可用傳遞訊息的中間伺服器來列出與整合這些遠端服務。

在 90 年代，隨著網際網路的出現，許多公司希望將建構 API 與公開 API 的做法標準化。諸如 Common Object Request Broker Architecture（CORBA）、Microsoft 的 Component Object Model（COM）與 Distributed Component Object Model（DCOM），以及許多其他的標準都企圖成為用 web 公開服務的公認做法。問題在於，多數這類的標準管理起來都太麻煩了，它們要求網路兩端都要使用類似的程式語言，有時需要在本地安裝部分的遠端服務（通常稱為 _stub_）才能生效。在這個混亂的局面中，實現交互運作的美夢很快成為充滿一大堆設置與諸多限制的惡夢。

在 90 年代晚期與 00 年代早期，開發者可以用一些比較開放且標準的做法，透過 web 使用遠端服務（web API）。首先是 Simple Object Access Protocol（SOAP）與 Extensible Markup Language（XML），接著有 Representative State Transfer（REST）與 JavaScript Object Notation（JSON），它們讓服務更容易使用也更標準化，不需要依賴用戶端程式碼或特定的程式語言。本書將討論以上這些最熱門且最實用的方法。

科技公司早就開始透過 API 來公開好用的服務了，從早期的 Amazon Affiliate API（2002）到 Flickr API（2004）再到 Google Maps API（2005）以及 Yahoo! Pipes API（2007），現在有成千上萬個 API 公開了你想像得到的服務，其中包括圖像處理與人工智慧。開發者可以呼叫它們，並且組合多個 API 來創造新的產品，就像組裝樂高積木一般。

雖然 API 已經成為一種易用的日常用品了，但是建構 API 仍然是一門藝術。千萬不要小看這項挑戰，建構堅實的 API 並不容易！API 必須非常簡單，而且有高度的交互運作性——就像樂高，任何組合的每一個零件都可以換成其他組合的每一個零件。API 也必須附有開發者程式與資源，以協助開發者使用。

建構堅實的 API 只是第一步而已，你也要建立、支援蓬勃的開發者生態系統。我們會在本書的最後一個部分討論這些挑戰。

我們這些作者寫這本書的原因，是我們在工作時都經歷了類似的過程，為許多 API 做出類似的決策、處理與優化，但是這些準則都沒有被寫成具備公信力的資源。我們每個人都可以介紹好幾篇與各種主題有關的部落格文章或文獻，但卻找不到專門探討如何規劃 web API 及其生態系統的文獻。我們希望用這本書告訴你，我們在建構 API 時曾經創造與發現的工具，這些技術相當寶貴，它可能攸關你的企業或技術的成敗，也將會成為你職涯的獨特優勢。

本書的架構

本書有三個主要的部分：

理論（第 1 至 4 章）

我們在此討論建構 API 的基本概念，回顧各種不同的 API 模式，並討論優良 API 的各種面向。

實踐（第 5 至 7 章）

在這些章節中，討論如何實際設計 API 並在產品中管理它的操作。

開發者粉絲（第 8 至 11 章）

在這個部分，我們跳出 API 的設計，向你展示如何建構一個蓬勃的開發者生態系統。

本書也收錄了一些案例研究（來自 Stripe、Slack、Twitch、Microsoft、Uber、GitHub、Facebook、Cloudinary、Oracle 等大公司的經驗！）、這個領域的專家所提供的建議與專業提示，以及真正發生過的經驗。附錄 A 有方便的工作表、模板與檢查清單供你使用。

第一章

API 是什麼？

“API 是什麼？”當程式設計新人問這個問題時，他們通常會得到這個答案“編寫應用程式的介面”。

但是 API 不是只有名稱上的意義而已，為了瞭解與發揮它們的價值，我們必須把焦點放在介面這個關鍵字上。

API 是軟體程式提供給別的程式或人類使用的介面，例如，web API 是透過網際網路提供給全世界的介面。API 這項設計相當程度地掩蓋了它底下的程式，包括商務模型、產品功能、偶發的 bug。雖然 API 的設計是為了與其他程式合作的，但它們大多是為了讓編寫那些程式的人類瞭解與使用的。

API 是讓 web 上的主要商業平台具備交互運作性的積木。API 是在各個雲端軟體帳號建立、維護身分的工具，包括你公司的 email 地址、合作設計的軟體、幫你預訂 pizza 外送的 web 應用程式。API 是將氣象預測資料從可信的來源（例如美國氣象局）送到專門顯示它的上百種 app 的手段。API 可以處理你的信用卡，並且讓各家公司無縫收取你的付款，而不需要操心金融技術的旁枝末節與相應的法規。

API 已經逐漸成為富擴展性且成功的網際網路公司的關鍵元素了，這些公司包括 Amazon、Stripe、Google 與 Facebook。對希望建立商業平台來將市場延伸到每一個人的公司來說，API 是很重要的成分。

設計你的第一個 API 只是工作的開端，這本書不僅僅討論你的第一個 API 的設計原則，也要深入介紹如何隨著商務來開發與發展 API。當你做出正確的選擇，你的 API 就經得起時間的考驗。

為什麼我們需要 API ？

API 最初的目的是為了與能夠解決特定問題的資料提供者交換資訊，讓不同公司的成員不需要自行花時間解決問題。例如，我們可能想要在網頁中嵌入一個互動式地圖，但不希望重新創造 Google Maps，也可能想要讓使用者登入，但不希望重新製作 Facebook Login，或者，我們可能想要建立一個偶爾可以和使用者互動的聊天機器人，但不希望建立即時通訊系統。

這些例子的輔助功能與產品都是用專門的平台提供的資料或互動來建立的。API 可讓業界快速開發獨特的產品，也可以讓新創者在踏入其他的生態系統時，只要利用現有的技術就可以讓產品與眾不同。

我們的使用者是誰？

> 如果你的重點不是幫正確的顧客製作正確的東西，那麼所有的理論都不重要。
>
> —*Bilal Aijazi，Polly 的 CTO*

當你製作任何一種產品時，最好先把焦點放在顧客身上，這一點在設計 API 時也很重要。在第 8 章，我們會討論各種類型的開發者與使用者案例，以及與他們接觸並賦予他們價值的策略。你必須瞭解你的開發者是誰、他們的需求是什麼，以及為什麼他們要使用你的 API。把焦點放在開發者身上，可以避免你做出沒人想要使用或不符合開發者需求的 API。

因為事後更改 API 的設計很困難，因此在開始實作 API 之前先確定並驗證它非常重要。對大多數的開發者來說，從一種 API 設計切換到另一種設計需要付出很高的代價。

以下是用來上傳與儲存圖像的 API 的開發者使用案例：

- Lisa 是一間藝術品銷售公司的 web 開發者，她需要一種可讓藝術家輕鬆上傳與展示照片的方法。
- Ben 是位後端企業開發者，他需要將支出系統的收據存入他的稽核原則解決專案之中。
- Jane 是位前端開發者，她想要在公司的網站中加入即時的顧客支援聊天功能。

以上只是少數的案例，它們都有獨特的潛在需求與要求。如果你無法滿足開發者的需求，你的 API 就無法成功。

在下一節，我們要討論一些影響 API 設計的高階使用案例，但只要你越仔細地處理使用案例，並且越深入瞭解你的開發者，你就可以提供越好的服務。

API 的業務案例

眾所周知，網路為現今的產品創新和技術市場提供了絕大部分的動力，因此，API 對開創事業的重要性超越過往任何時刻，坊間也有許多模型可將它們整合到產品之中。有時 API 可直接帶來利潤與盈收（透過利潤分享模型、訂閱費或使用費），但是我們也有許多其他建立 API 的理由。有可能 API 可以支援公司的整體產品策略，它們可能是整合第三方服務與公司的產品的重要環節。API 也有可能是整體策略的一部分，目的是促使別人製作產品的開發者不願意或無法投注精力的輔助產品。API 也有可能是吸引潛在客戶、建立新的產品分銷管道，或提高產品銷量的手段。要進一步瞭解這些策略，可參考 John Musser 的 API business models 演說（*https://www.slideshare.net/jmusser/j-musser-apibizmodels2013*）。

API 必須與核心的業務保持一致，許多軟體即服務（SaaS）公司都是如此，其中最值得注意的案例是 GitHub、Salesforce 與 Stripe。用這些 API 建立的產品有時稱為“服務整合產品”。如果你有許多由使用者產生的內容（例如使用 Facebook 與 Flickr 的照片分享功能），用戶 API 就可以發揮很大的作用。雖然建立 API 與推出開發者平台的原因很多，但是當 API 策略與核心業務不一致時，就是不建立開發者平台最重要的考量了。例如，若產

品的主要收入流量是廣告，那麼為產品選擇“用戶端”的 API 會將廣告的流量分散，造成收入的減少，就像 Twitter API 的情況。

撇開營利與商業動機不談，以下是幾種公司開發 API 的方式，我們仔細研究它們：

- 先讓內部開發者使用，再讓外部開發者使用的 API
- 先讓外部開發者使用，再讓內部開發者使用的 API
- API 即產品

先讓內部開發者使用，再讓外部開發者使用的 API

有些公司優先幫他們的內部開發者製作 API，再將 API 交給外部開發者使用。這種做法的動機有很多種，其中一種是公司看到加入外部 API 的潛在價值。這種做法可以建立開發者生態系統、驅動公司產品的新需求、或是促使其他公司製作原公司不想製作的產品。

舉個具體的例子，我們來看一下 Slack 的 API 的源起——它是 Slack 的 web、原生桌機與行動用戶端的 API，功能是顯示一個訊息傳遞介面。雖然 Slack 的 API 最初是為它的內部開發者建構的，但是這間公司在成長的過程中發生了一些事情：Slack 通訊軟體有一些與重要商業軟體的“整合”會大大地影響它的成長與發展，於是 Slack 決定不整合自家與別家的產品來製作訂製的 app，而是推出 Developer Platform（開發者平台）與一套可讓新舊公司建構它們的 app 的產品。

Slack 的這個舉措促進了“整合 Slack 傳訊平台的 app”的生態系統的發展。也意味著同時使用 Slack 與其他商業軟體的使用者可以無縫整合已經在 Slack 訊息傳遞用戶端中進行的通訊。

當 Slack 的 Developer Platform 平台啟動時，它的 API 有一個優勢：這些 API 已經被內部的開發者充分測試和使用過了。但是隨著時間的推移，當外部開發者與內部開發者的需求漸漸不一致的時候，這種方法的缺點也開始浮現。內部開發者必須靈活地為傳

訊用戶端的最終使用者創造新體驗，包括新的公用頻道、檔案與訊息類型，以及日益複雜的通訊體驗。與此同時，第三方開發者再也不會幫 Slack 建立可供替換的用戶端使用者介面（UI）了——他們開始創造強大的工作流程商業 app 與工具，而非只限於顯示訊息。外部的開發者也需要穩定性，“API 的回溯相容性”與“為了新產品的功能而改變 API”之間的緊張關係讓 Slack 付出內部專案開發速度下降的代價。

先讓外部開發者使用，再給內部開發者使用的 API

有些公司先幫外部的專案關係人建立 API，再讓內部的專案關係人使用它們，這正是 GitHub 從一開始就採取的做法。我們來看一下 GitHub 如何與為何開發它的 API，以及它的使用者如何影響 API 的演變。

最初，GitHub API 的用戶主要是想要用自己的資料來設計程式的外部開發者。在 API 首次發表之後不久，有一些圍繞著 GitHub API 的小型公司開始成立。這些公司創造了開發工具，並將它們賣給 GitHub 的使用者。

此後，GitHub 大大地擴展它的 API 產品。它製作了一個 API，這個 API 既可以為想要創造自己的專案或工作流程的個人用戶提供服務，也可以為想要一起建構與 GitHub 整合的機器人腳本或工作流程工具的團隊提供服務。這些團隊，稱為整合者（integrator），建構了開發工具、用 GitHub 的平台來連接使用者，並且將那些工具賣給共同顧客。

當 GitHub 建構它的 GraphQL API 時，第三方開發者成為第一群顧客。GraphQL 是 web API 的查詢介面，雖然這種介面不是同類型的第一種，但是由於它是著名的 API 供應者 Facebook 製作的，而且被另一個著名的 API 供應者 GitHub 採用，所以在寫這本書時備受矚目。當第三方開發者開始使用 GitHub 的新 GraphQL API 之後，GitHub 內部開發者也使用它來支援 GitHub web UI 與用戶端 app 的功能。

在 GitHub 的例子中，這個 API 有一個明顯的目的，它先服務外部的專案關係人，最後也服務內部的開發者。採取這種做法的優點之一在於，它可以為外部開發者訂製 API，而不是腳踏兩條船，橫跨兩個使用族群。隨著 GitHub API 的演變，它能夠用越來越多開發者需要的資料來註解它的 JSON 回應。最終，由於負載（payload）太大了，所以 GitHub 製作了 GraphQL 讓開發者可以用查詢指令（query）來指定他們想要的欄位。GraphQL 的做法有一個缺點在於，因為 GraphQL 讓開發者擁有許多彈性，所以各種存取模式有許多效能瓶頸。與一次使用一個端點（例如 REST）的做法相較之下，這種做法會讓問題的排除更加麻煩。

第 2 章會更詳細說明 GraphQL。

API 即產品

對一些公司來說，API 就是產品，例如 Stripe 與 Twilio。Stripe 提供在網際網路處理付款的 API。Twilio 提供透過簡訊、語音與訊息來通訊的 API。在這兩間公司的案例中，API 的建構 100% 是為了單一產品的用戶，它們的 API 就是產品，整個企業的目的都是為顧客建構無縫的介面。就 API 的管理與滿足顧客需求而言，“API 即產品”是最直截了當的公司策略。

偉大的 API 有哪些特徵？

我們向業界專家提出這個問題，得到的答案可歸結為 API 究竟可否做它該做的事情。為了深入研究有助於提升 API 可用性的各個層面，我們不但會討論 API 的設計與擴展，也會說明為了促進開發者使用 API 而應該提供的支援與生態系統。

專家說

良好的 API 與你想要解決的問題以及解決它的價值有多大有很大的關係。如果有一個 API 可讓你使用獨特的資料集合或複雜的功能，就算它用起來讓人費解、不一致、沒有良好的文件，你也會使用它。良好的 API 往往提供了明確的意圖（當然，指的是設計與環境（context））、彈性（能夠用於各種不同的使用案例）、功能（所提供的解決方案的完整性）、hackability（能夠藉由反覆使用與試驗來快速掌握），以及文件。

—Chris Messina，Uber 的開發者體驗主管

易用性、可擴展性與效能都是製作優秀 API 的要素。我們會在第 2 到 4 章討論這些主題。文件與開發者資源對協助使用者成功而言也非常重要。我們會在第 7 到 9 章討論它們。因為你不可能優化所有的 API 要素，所以必須決定哪一種要素對最終使用者來說是最重要的。我們會在第 7 章教你如何擬定處理這種問題的策略。

此外還有一件事需要考慮：如何讓一個偉大的 API 經得起時間的考驗。變動是很艱巨且不可避免的工作。API 是將企業連接起來的靈活平台，它的改變率（rate of change）並不是固定的。大型企業環境的改變率比尚未找到產品市場契合（product–market fit）的小型新創公司緩慢。但有時小型新創公司可透過 API 提供大型企業不可或缺的寶貴服務。你會在第 5 章看到如何設計經得起時間考驗的 API。

總結

總之，API 是現代科技產品的重要元件，我們可以利用它們透過許多種方式來開創事業。在第 2 章，我們要回顧一些 API 設計模式。

第二章

API 模式

選擇正確的 API 模式非常重要。API 模式就是公開某項服務的後端資料給其他 app 的介面的定義。當一個機構開始創造 API 時，它們不一定知道讓 API 成功的所有因素，所以沒有建立充分的空間方便在之後加入新功能。當機構或產品隨著時間而改變時也會如此。不幸的是，當開發者開始使用 API 之後，API 就很難變動了（有時甚至不可能）。為了節省時間、精力與麻煩（並且為新的、很酷的功能留下空間），在你動手前，應該先參考一下各種協定、模式與一些最佳做法，以設計未來可以進行改變的 API。

多年來，坊間已出現許多 API 模式了，REST、RPC、GraphQL、WebHook 與 WebSockets 都是目前最熱門的標準，本章將討論這些典範。

請求 / 回應 API

請求 / 回應 API 通常透過 HTTP web 伺服器來公開介面。這些 API 定義了一組端點，讓用戶端對著這些端點發出 HTTP 請求來索取資料，伺服器則會做出回應。這些回應通常是用 JSON 或 XML 來回傳的。請求 / 回應 API 的服務通常使用三種模式來公開：REST、RPC 與 GraphQL。接下來的小節將一一討論它們。

表現層狀態轉換

表現層狀態轉換（REST）是近來最熱門的 API 開發選項。Google、Stripe、Twitter 與 GitHub 等提供者都採取 REST 模式。REST 與資源息息相關，資源是可在 web 上被識別、指名、定址或處理的實體。REST API 會將資料當成資源來公開，並使用標準的 HTTP 方法來代表對這些資源做的建立、讀取、更新與刪除（CRUD）等動作。例如，Stripe 的 API 用資源來表示顧客、費用、餘額、退款、事件、檔案與支出。

以下是 REST API 所遵循的一般規則：

- 資源是 URL 的一部分，例如 `/users`。
- 每一個資源通常有兩個 URL：一個代表群體，例如 `/users`，一個代表特定元素，例如 `/users/U123`。
- 使用資源名詞，而不是動詞。例如，使用 `/users/U123`，而不是 `/getUserInfo/U123`。
- 用 `GET`、`POST`、`UPDATE` 與 `DELETE` 等 HTTP 方法來告知伺服器將要執行的動作。用不同的 HTTP 方法對著同一個 URL 進行呼叫有不同的作用：

  建立

  使用 `POST` 來建立新資源。

  讀取

  使用 `GET` 來讀取資源。`GET` 請求永遠不會改變資源的狀態。它們沒有副作用；`GET` 方法有唯讀的意思。`GET` 是冪等（idempotent）的，因此，你可以完美地快取呼叫。

  更新

  使用 `PUT` 來替換資源，以及使用 `PATCH` 來更新部分的既有資源。

  刪除

  使用 `DELETE` 來刪除既有資源。

- 伺服器回傳標準的 HTTP 回應狀態碼來指出成功或失敗。通常 2XX 範圍內的代碼代表成功，3XX 代碼代表資源已被移除，4XX 代碼代表用戶端錯誤（例如缺少必要的參數或太多請求）。5XX 代碼代表伺服器端錯誤。
- REST API 可回傳 JSON 或 XML 回應。儘管如此，由於 JSON 很簡單，而且很容易和 JavaScript 搭配使用，所以它已成為現代 API 的標準了。（為了讓已經透過類似的 API 來使用 XML 與其他格式的用戶端使用，這些格式仍然可能會被支援。）

表 2-1 列出 REST API 通常如何使用 HTTP 方法，範例 2-1 與 2-2 是一些 HTTP 請求範例。

表 2-1 CRUD 操作、HTTP 動詞與 REST 規範

操作	HTTP 動詞	URL: /users	URL: /users/U123
建立	POST	建立一位新使用者	不適用
讀取	GET	列出所有使用者	取得使用者 U123
更新	PUT 或 PATCH	批次更新使用者	更新使用者 U123
刪除	DELETE	刪除所有使用者	刪除使用者 U123

範例 2-1 用 Stripe API 取得費用的 HTTP 請求

```
GET /v1/charges/ch_CWyutlXs9pZyfD
HOST api.stripe.com
Authorization:Bearer YNoJ1Yq64iCBhzfL9HNO00fzVrsEjtVl
```

範例 2-2 用 Stripe API 建立費用的 HTTP 請求

```
POST /v1/charges/ch_CWyutlXs9pZyfD
HOST api.stripe.com
Content-Type: application/x-www-form-urlencoded
Authorization:Bearer YNoJ1Yq64iCBhzfL9HNO00fzVrsEjtVl

amount=2000&currency=usd
```

顯示關係

盡量用子資源來表示只屬於其他資源的資源，而不是用 URL 的頂層資源來表示它，這種做法可讓使用 API 的開發者清楚知道它們之間的關係。

例如，GitHub API 在各種 API 中使用子資源來表示關係：

`POST /repos/:owner/:repo/issues`

建立一個問題。

`GET /repos/:owner/:repo/issues/:number`

取得一個問題。

`GET /repos/:owner/:repo/issues`

列出所有問題。

`PATCH /repos/:owner/:repo/issues/:number`

編輯一個問題。

非 CRUD 操作

除了剛才看到的 CRUD 典型操作之外，REST API 有時需要表示非 CRUD 的操作，以下是常見的做法：

- 以資源的部分欄位來表示動作。如範例 2-3 所示，GitHub 的 API 使用存放區編輯 API 的輸入參數 `"archived"` 代表將存放區歸檔（archive）的動作。
- 將操作視為子資源。GitHub API 使用這個模式來鎖定與解鎖一個問題。`PUT /repos/:owner/:repo/issues/:number/lock` 可鎖定一個問題。
- 有些操作難以用 REST 模式實現，例如搜尋，此時，通常會在 API URL 中直接使用操作動詞。`GET /search/code?q=:query:` 會在 GitHub 中尋找匹配 query 的檔案。

範例 2-3 將 GitHub 存放區歸檔的 HTTP 請求

```
PATCH /repos/saurabhsahni/Hacks
HOST api.github.com
Content-Type: application/json
Authorization: token OAUTH-TOKEN

{
  "archived": true
}
```

遠端程序呼叫

遠端程序呼叫（Remote Procedure Call，RPC）是最簡單的 API 模式之一，它的用戶端會在另一個伺服器上執行一段程式碼。REST 與資源有密切的關係，RPC 則與動作有關。用戶端通常會傳遞方法名稱與引數給伺服器，以取回 JSON 或 XML。

RPC API 通常遵循兩個簡單的規則：

- 端點含有準備執行的操作的名稱。
- API 呼叫是用最適合的 HTTP 動詞來執行的：`GET` 是唯讀請求，`POST` 是其他的。

當 API 公開的動作比 CRUD 封裝的還要細膩且複雜，或是存在與眼前的“資源”無關的副作用時，很適合使用 RPC。RPC 樣式的 API 也可以配合複雜的資源模型，或針對多種類型的資源執行的動作。

Slack 的 API 是一種明顯的 PRC 樣式 web API 案例。範例 2-4 是對著 Slack 的 `conversations.archive` RPC API 發出 `POST` 請求的例子。

範例 2-4 對著 Slack API 發出 HTTP 請求

```
POST /api/conversations.archive
HOST slack.com
Content-Type: application/x-www-form-urlencoded
Authorization:Bearer xoxp-1650112-jgc2asDae

channel=C01234
```

Slack 的 Conversations API（圖 2-1）可執行許多動作，例如歸檔、聯結、踢出、離開與重新命名。雖然這個例子有明確的“資源”，但並非所有的動作都很適合 REST 模式。此外，有些其他的動作（例如用 `chat.postMessage` post 訊息）與訊息資源、附件資源以及 web 用戶端內的畫面設定有複雜的關係。

對話 API 方法

方法	說明
conversations.archive	將一場對話歸檔。
conversations.close	關閉一個直接訊息或多人直接訊息。
conversations.create	開始一場公開或私人頻道的對話。
conversations.history	抓取一場對話的訊息與事件歷史紀錄。
conversations.info	取得一場對話的資訊。
conversations.invite	邀請使用者進入頻道。
conversations.join	加入既有的對話。
conversations.kick	將一位對話中的使用者移除。
conversations.leave	離開對話。
conversations.list	列出 Slack 團隊的所有頻道。
conversations.members	取得對話的成員。
conversations.open	打開或恢復一個直接訊息或多人直接訊息。

圖 2-1 RPC 樣式的 Slack API 方法

RPC 樣式的 API 並非只能使用 HTTP，它也可以使用其他高效能的協定，包括 Apache Thrift（*https://thrift.apache.org/*）與 gRPC（*https://grpc.io/docs/guides/index.html*）。雖然 gRPC 有 JSON 選項，但 Thrift 與 gRPC 請求都是被序列化的。結構化的資料與明確定義的介面促成了這種序列化。Thrift 與 gRPC 也有內建的資料結構編輯機制。本書不會討論太多 gRPC 與 Thrift 的範例，但還是要稍微介紹一下它們。

GraphQL

GraphQL（*http://graphql.org/*）是近來備受矚目的 API 查詢語言。它是 2012 年 Facebook 在內部開發的，隨後在 2015 年公開發表，並且被 GitHub、Yelp 與 Pinterest 等 API 提供者採用。GraphQL 可讓用戶端定義所需的資料結構，讓伺服器完全以那個結構回傳資料。範例 2-5 與 2-6 是送給 GitHub API 的 GraphQL query 及其回應。

範例 2-5　GraphQL query

```
{
  user(login: "saurabhsahni") {
    id
    name
    company
    createdAt
  }
}
```

範例 2-6　GitHub GraphQL API 的回應

```
{
  "data": {
    "user": {
      "id":"MDQ6VXNlcjY1MDI5",
      "name":"Saurabh Sahni",
      "company":"Slack",
      "createdAt":"2009-03-19T21:00:06Z"
    }
  }
}
```

與 REST 和 RPC API 不同的是，GraphQL API 只需要一個 URL 端點。與之前的情況類似，你不需要用不同的 HTTP 動詞來描述操作，只要在 JSON 內文中指明你究竟在執行查詢還是變動就可以了，如範例 2-7 所示。GraphQL API 支援 `GET` 與 `POST` 動詞。

範例 2-7　對著 GitHub 執行 GraphQL API 呼叫

```
POST /graphql
HOST api.github.com
Content-Type: application/json
Authorization: bearer 2332dg1acf9f502737d5e
```

```
    xoxp-16501860787-17163410960-113570727396-7051650

{
  "query": "query { viewer { login }}"
}
```

與 REST 和 RPC 相較之下，GraphQL 有一些重要的優勢：

節省多次的往返

GraphQL 可讓用戶端嵌套 query 並且用單一請求從多個資源取回資料。如果不使用 GraphQL 的話，做這件事可能需要對伺服器做多次 HTTP 呼叫，這意味著使用 GraphQL 的行動 app 運行的速度較快，即使在緩慢的網路上也是如此。

不需要管理版本

你可以在 GraphQL API 加入新的欄位與型態且不影響既有的查詢，同樣的，棄用既有的欄位也很方便。API 提供者可以用 log 來分析有哪些用戶端使用了某個欄位。你可以在工具中隱藏棄用的欄位，並且在沒有用戶端使用它們時移除它們。使用 REST 與 RPC API 時，較難以找出有哪些用戶端正在使用已被棄用的欄位，所以難以移除它們。

較小的負載

REST 與 RPC API 經常回傳用戶端永遠用不到的資料。使用 GraphQL 時，因為用戶端可以明確地指定他們需要什麼，所以有更小的負載。GraphQL query 會回傳可預測的結果，用戶端也能夠控制回傳的資料。

強型態

GraphQL 是強型態。在開發期，GraphQL 型態檢查可協助確保 query 的語法是正確且有效的，讓你更容易做法高品質、不易出錯的用戶端。

自我檢查

雖然有一些外部的解決專案（例如 Swagger）可以協助你輕鬆地瞭解 REST API，但 GraphQL 原本就很容易瞭解。它有一個可用來瞭解 GraphQL 的瀏覽器 IDE，GraphiQL（*https://*

github.com/graphql/graphiql），可讓使用者在瀏覽器內編寫、驗證與測試 GraphQL query。圖 2-2 是使用 GraphiQL 來瞭解 GitHub API 的情況。

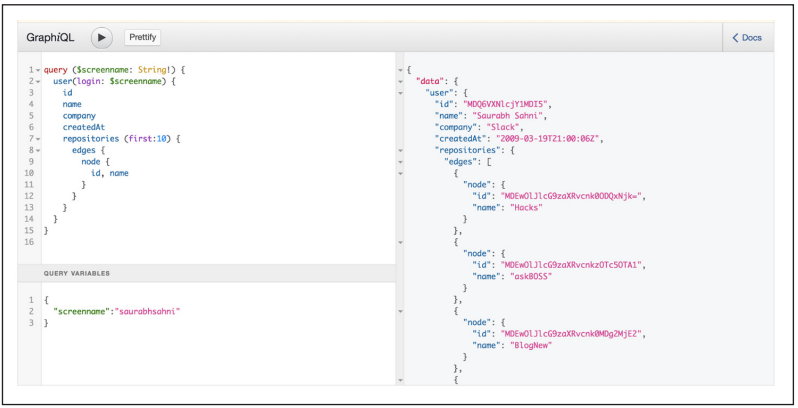

圖 2-2 GraphiQL：正在展示複雜的 query 的 GitHub GraphQL 探索器

專家說

REST 負載蠕變是 GitHub 的嚴重問題之一，隨著時間的推移，你會將（例如，存放區的）額外的資訊加入序列化器（serializer）。它一開始很小，但是當你加入額外的資料（或許是你已經加入新功能），原始程式就會產生越來越多資料，到最後，API 回應會變得非常龐大。

多年來，我們藉由建立更多端點來讓你指定更詳細的回應，以及加入越來越多快取來解決這個問題。但是隨著時間的推移，我們發現我們回傳了許多整合者不想要的資料，這就是我們開發 GraphQL API 的原因之一。使用 GraphQL 時，你可以設定只會取得想要的資料的 query，我們只會回傳那些資料。

—Kyle Daigle，GitHub 的生態工程總監