推薦序

根據最近 IDC 的一份關於 API 的調查報告，七成五的受訪者將 API 視為數位轉型的重點，也有超過一半的人預期 API 的使用量與回應時間將會大幅增長，除此之外，大部分的組織承認他們面臨的挑戰也來自 API，因為當前的 API 難以滿足內部外部雙方面的需求，這些問題都再再指出，企業需要一個一致的、可靠的、可擴展的 API 制定計畫來協助他們通過數位轉型，如同作者 James Higginbotham 所說的：「當今最大的挑戰來自於如何設計出具有一致性、可擴展性的 API，並使開發者得以理解並加以運用。」

出於上面的原因，我很開心看到有這本書的出現，我曾經有幸與 James 一同工作，他有著絕佳的工作態度，我很開心知道他在寫一本關於 Web API 設計的書，在拜讀之後，我很樂意將此書推薦給各位讀者。

過去幾年來，Web API 與相關的設計工作迅速成長，人們也開始關心該如何跟上這種趨勢，API 已經成為一塊廣大的領域，當中包含了許多的面向，商業面上，API 該扮演怎樣的角色；設計面上，API 的需求彙整、文件撰寫該如何進行；技術面上，編程技術的演進對程式撰寫、測試、發布、監控等帶來的變化等等，這些各個面向的課題都使 API 難以掌握，而透過書中的 ADDR（Align-Define-Design-Refine，對齊 - 定義 - 設計 - 優化）流程，以及那些 James 提供的建議與範例，將可以幫助讀者有系統地走入 API 的世界，以因應未來更大的挑戰。

James 的長處之一是他能夠跨越技術面，從社會與商業面的觀點檢視組織的 API，他在銀行、保險、貨運、電腦硬體等客戶的豐富經驗也充分反映在本書中，成為教材的一部分，不同產業、規模大小的組織都可以運用書中提及的技術與流程，這本書集結了詹姆士數年來的精華，不論您是在尋求設計上的建議，或是追求商業與技術上的一致性，抑或是探求 REST、GraphQL 等技術面的實作細節，您都可以在書中找到適合且具體的內容。

書中我認為特別有價值的部分是關於如何在成長中的企業對 API 進行設計與實作上優化的章節，對公司內負責 API 的執行與管理的人士來說，你們的書架上也應該都要供奉一本《Web API 設計原則》。

如前述 IDC 報告所指出的，全球許多公司都面臨數位轉型帶來的挑戰，在如何滿足客戶的需求上，API 扮演著重要的角色，無論您關注的是 API 的設計、建置、部署、維護等的哪個面向，都可從本書找到實用的見解與建議。

我相信在我與各產業合作，協助他們開發 API 時，此書將會是我的知識庫中重要的一部分，希望您也有相同感受，書中點出了我們會遭遇到的各種機會與挑戰，而對此容我借用 James 的一句話作為回應：「這只是開始」。

—*Mike Amundsen*，*API* 策劃師

前言

已經很難認定這本書真正的起源，或許是十年前開始的——它是在經歷數千小時的訓練、數萬公里的旅程，以及數以萬計的文字與程式碼之後所累積的心血，它包含來自全球各地不同企業、組織對 API 設計的經歷與理解，這些組織有的才踏出第一步，有的已經走上 API 之路，我有幸見識過這些企業在 API 設計方面的見解與洞察，並將他們的智慧結晶彙整於此。

而也或許來自更早的二十五年前左右——當我初次進入軟體業時，來自書籍與文章給予的知識，以及在軟體架構方面前輩對我的影響，這些都奠定了我在實現軟體架構時的思維模式。

又或許是來自更久遠的四十年前左右——當時爺爺送了我一台 Commodore 64 電腦，愛好知識的爺爺白天是土木工程師與成本工程師，下班後還到夜校求學，他喜愛閱讀並吸收一切知識，當他在看到電腦運作時說的那句「真是太神了！」總是令人感到莞爾，也是他送了我這台神奇的電腦，他曾說：「電腦會越來越厲害，我的小孫子應該要了解它。」而這也開啟了我一生對軟體開發的熱愛。

實際上，真正的起源來自於七十多年以前——當時與我們年紀相仿的時代先驅們，建立了許多軟體建構的基本原則，這些原則直到今日依然為我們所用，即使技術隨著時代不停更迭，但在技術更迭之下的基石，依靠的是軟體產業無數人士辛勤工作之後的成果，有他們的昔日成果才有今日我們開闢前往未來的道路。

我要說的是，如果少了前人的耕耘，就不會有今日 API 的存在，我們必須先了解行業的歷史，才能更深刻的理解今日所做之事背後的「如何」與「為何」，然後設法在往後運用那些前人的智慧，同時我們需要找到方法去激勵其他人也這麼做，這是我祖父和父親教給我的一課，而我也透過此書傳達同樣的理念，這本書反映了我迄今在生涯旅程中的所知所學，我希望您透過此書的內容能獲得一些新的見解，並以此為基礎，再向下一個世代傳播這些知識與理念。

誰適合閱讀本書

簡單的說，不論您是否從事程式開發，只要與 API 設計有關，都應該閱讀本書。產品經理能藉由本書深入了解團隊在設計 API 時所需的要素；軟體架構師與開發者將學習到相關的軟體架構，並依此原理來設計 API；技術寫作者將會知道，他們不僅能寫出清晰明瞭的 API 文件，更可以在 API 設計過程中提供更多有意義的價值。

關於本書

本書概述了設計 API 的一系列原則和過程，書中的「ADDR」（Align-Define-Design-Refine，對齊 - 定義 - 設計 - 優化）流程旨在幫助個人和跨職能團隊應對 API 設計的複雜性，此流程鼓勵以「從外向內」（outside-in）的視角來進行 API 設計，其中的概念包括聆聽客戶聲音、工作目標與流程映射（process mapping）等，儘管《Web API 設計原則》是用一個從零開始的全新 API 做為示例，但本書也適用於既有 API 的設計。

這本書涵蓋了 API 設計中，從需求探訪到最終交付的各個面向，也包括撰寫 API 設計文件的指南，以利您與團隊及 API 用戶間進行更有效的溝通，最後，本書也涉及了一些在 API 交付時可能會對 API 設計產生影響的議題。

本書分為五大篇：

- **Part I：初探 Web API 設計**——概述 API 的重要性，並介紹本書中使用的 API 設計流程。
- **Part II：尋求一致性**——確保 API 設計團隊與參與者及所有客戶具有一致性的目標。
- **Part III：定義 API**——釐清為滿足 API 作業及目標所必要的相關需求，並產出 API Profile。
- **Part IV：設計 API**——將 API Profile 依專案需求，以不同的 API 風格實現，包括 REST、gRPC、GraphQL，以及 event-based async API（事件式的異步 API）等。

- **Part V：優化 API 設計**——根據來自文檔、測試的反饋持續改善 API 設計，以及一個獨立的章節，談論將 API 解構成微服務，最後，本書結尾將會討論到，將此設計流程更廣泛的運用在大型組織內的議題。

對於那些需要復習 HTTP（Web 底層的通訊協議）的人，我們在附錄提供了很棒的入門篇章，幫助您走入 HTTP 的世界。

本書並不是…

除了一些用於表達 API 細節的結構化文件，本書沒有其他的程式碼，因此就算不是軟體開發人員，也還是可以運用本書介紹的流程與技術，因為那些流程或技術並不涉及特定的程式語言或特定的設計與開發方法。

完整的 API 設計與交付是範圍龐大的議題，雖然本書是以 API 設計為題，但不可能完整涵蓋這個龐大議題全部的情節與細節，本書聚焦的是如何解決團隊在設計 API 時的挑戰，特別是從單一想法發展成完整的商業需求，進而成為具體的 API 設計等這一連串的過程中要面臨到的各種挑戰。

讓我們開始吧！

第一章

API 設計原則

> 所有架構都來自設計，但並非所有設計都能成為架構，架構來自一系列重要的設計決策，這些決策塑造了一個系統的形式及功能。
>
> —Grady Booch

企業組織在發展 API 上，已有數十年的歷史，最原始的形式是那些販售的共用程式庫或元件，它們最終在架構上逐漸走向標準化，例如用於物件的 CORBA（通用物件請求代理架構）標準，以及用於服務的 SOAP（簡單物件存取協定）標準，這些標準為程式的互通定義了規範，但並未定義設計面上的規範，因此每次的 API 串接工作往往得耗時數個月方可完成。

在網路時代來臨後，Web API 成為主流，早期只有少量的 API 應用，團隊也有充足的時間來細細思量 API 的設計，然而今非昔比，現在的 API 生態呈爆發式成長，在市場的驅動下，企業組織也必須以更快的速度交付 API，API 不再只是幾個內部系統間的小遊戲。

時至今日，用於連接彼此的 Web API 已經有了許多產業標準得以遵循，還有數以百計的套件和框架，讓我們得以用更簡單、更快、更省錢的方式建置出 API，更有 CI/CD 工具讓我們輕鬆地完成自動化作業，這些新玩具都讓我們能用更快也更有效率的方式交付我們的 API。

然而，當今面臨的挑戰是該如何設計出易於理解、使用，又有一致性與可擴展性的 API，面對這樣的挑戰，首先得認知到，Web API 不只是程式的玩意兒，如同好的藝術作品來自光線與色彩的完美調和，好的 API 設計也是來自商業能力、產品思維、開發體驗的完美平衡。

Web API 設計要素

企業提供的 API 反映了市場上該企業對外傳達的價值，開發者會從 API 設計品質的視角來檢視它，API 某些特性的存在與否，也反映了該企業對該特性的重視與否，一個高成效的 API 設計應該要考量到下面三個重要的元素：商業能力、產品思維、開發體驗。

商業能力

所謂的商業能力，指的是企業組織為維持市場競爭力與獲利所具備之能力，商業能力可以是組織面向外部的能力，例如獨特的產品設計能力、出色的客戶服務能力、產品交付能力等，商業能力也可以是組織面向內部的能力，例如銷售渠道管理能力、信用風險評估能力等。

組織可藉由三種模式對外傳達自身的商業能力：由組織自身傳達、由外包廠商傳達、或者兩者之混合。

以販售自家口味的咖啡館為例，咖啡豆來自豆商，烘焙在自家，而門市銷售的 POS 系統又是來自某個外部廠商，像這樣把部分事務外包，咖啡館就可專注在自身的核心商業能力，並且透過這樣的方式在市場上做出差異化。

我們將 API 視為數位化後的商業能力，當我們在設計 API 時，應該先理解 API 背後所代表的商業能力為何，並且該 API 的特性也應該反映出同樣的商業能力。

產品思維

在 Web API 尚不成熟的時代，企業與外部夥伴或用戶早已有所謂的資訊整合，但當時都是以客製化的方式進行的，而客製化也是當時面臨到的最大挑戰，每每有新的專案，都要派出一組專門的團隊負責該專案的客製化，而一次又一次的客製化之後，儘管付出了巨大的人力成本，獲得的成果卻難以為全體所共享，造成巨大的浪費。

而自從 SaaS 商業模式興起後，Web API 的需求也隨之增長，整合模式也從過往的單一用戶客製化模式，轉變為產品化思維導向模式。

在轉向產品思維導向之後，事務的焦點也就從單一用戶客製化轉移到 API 設計本身，優良的 API 設計減少了過往為每位用戶客製化的人力支出，並且還能為我們帶來新的用戶，自家員工、企業用戶都可以透過 API 轉向自助式的工作模式，自行實現所需要的資訊整合。

將 API 視為產品意味著更少的客製化，以及更多的投注在迎合市場需求上，一支 API 將會為多位客戶所用，應該在 API 設計階段就盡早納入用戶端的意見，才能更貼近市場需求，也才有機會迎來更多的潛在用戶。

開發體驗

用戶體驗（user experience）涉及的是在用戶與企業、用戶與產品、用戶與服務間的互動中，如何滿足客戶使用感受的課題，而開發體驗（developer experience）就像是 API 領域的用戶體驗，開發體驗重視的是開發者與 API 互動時的感受，它不只是那些 API 的技術細節，而是一個 API 產品的各個面向，從第一印象、實際使用的感受，延伸到技術支援這些層面上帶給開發者的體驗。

要成就一個好的 API，優秀的開發體驗是不可或缺的，優秀的開發體驗讓開發者能簡單快速上手，甚至能讓開發者主動對外推廣我們的 API，為我們在市場上吸引更多的用戶，也唯有優秀的開發體驗，開發者才能用更少的精力、更短的時間將他們的業務賦予更大的商業價值。

當設計團隊在思考如何展現出優秀的開發體驗時，別忘了開發體驗對內部開發者來說也是同樣重要，以文件為例，好的文件讓開發者能輕鬆上手，而差的文件讓開發者得先找到對的人，才能問到正確的使用方法，即便內部開發者是我們的同事，但這仍然是額外的溝通成本，優秀文件的價值在於它對內外部開發者都是有益的，他們都能透過文件輕易地為他們的事業增加更多的附加價值。

案例研究
當 API 與產品思維出現在銀行

自 2013 年起，Capital One 開始打造他們的企業 API 平台，起初的目標是為公司內部提供自動化機制，藉此提高效率，並打破組織內的壁壘。

當他們的 API 平台成長到一定規模後，他們的目標從原本的僅供內用放大到外部市場，計畫透過開放 API 建立更多的產品與可能性，於是他們在 SXSW [1] 大會發布了開發者入口網站 DevExchange 以及公開的 API 產品，範圍涵蓋了銀行級的授權認證系統、客戶獎勵計畫、信用卡審核等，甚至有一支 API 可以直接開立儲蓄帳戶。

之後 Capital One 將此概念進一步延伸，利用自身的數位能力發展全通路（omnichannel），以既有的 Web API 為基礎，利用 Amazon 的 Alex 平台搭建了一套語音交談機器人 Eno（就是倒著寫的 *one*），為客戶提供新穎的互動體驗 [2]。

藉由 API 的產品化，以及自身強大的數位能力在 API 上的展現，讓 Capital One 能與用戶及伙伴共同開拓新的商業機會，這些機會不會在一夜之間突然到來，但在 Capital One 全體共同的願景與意志貫徹之下，確實成真了。

API 設計＝溝通與交流

當一位開發者在思考軟體設計時，腦中迸出的會是一堆類別、方法、函式、模組、資料庫等技術概念；而 UML、流程圖、一堆框框和箭頭則用於協助我們理解程式的邏輯，不論是技術概念還是流程圖，我們在使用這些工具的同時實際上也是在進行一場彼此的溝通與交流。

同樣的，API 的設計也是一種溝通，但不只是團隊內成員彼此的溝通，而是更大範圍的對外溝通，這些溝通我們又分成三個面向：

1. **網路面的溝通：**API 的設計過程中必然會涉及到通訊協議的選擇，而通訊協議的選擇對 API 的溝通效率有著顯著的影響，例如 HTTP 較適用於粗粒度的訊息交換，較不適合用於高密度通訊的場景，而像是 MQTT（Message Queuing Telemetry Transport，訊息佇列遙測傳輸）或 AMQP（Advanced Message Queuing Protocol，高階訊息佇列協議）則更適合細粒度的訊息交換，也就是高

1 “Capital One DevExchange at SxSW 2017,” March 27, 2017, https://www.youtube.com/watch?v=4Cg9B4yaNVk

2 “Capital One Demo of Alexa Integration at SXSW 2016,” September 6, 2016, https://www.youtube.com/watch?v=KgVcVDUSvU4&t=36s

密度通訊場景。通訊協議的決定，影響了 API 的使用場景，我們在決定通訊協議時，應該要把應用場景中網路或設備可能會發生的效率問題也納入考量。

2. **開發者面溝通：**對開發者而言，API 的設計與文件就像是給他們的用戶介面，開發者透過文件才有辦法知道一支 API 的使用方式與時機，文件還能告訴開發者如何混搭不同的 API 操作，來達成更複雜的目的，對於開發者溝通，我們建議在 API 設計流程中盡早納入來自開發者的意見，才能設計出真正符合他們需求的 API。
3. **市場面的溝通：**透過 API 設計與文件，市場上的用戶、夥伴，以及開發人員才有辦法得知 API 具備那些數位能力，以及能幫他們達成什麼樣的目的，一個好的設計有助於 API 對市場面的溝通並促使人們善加利用他們。

如何建立有效的溝通，是從事 API 設計時的重要課題，而 API 設計流程能幫助我們去思考不同角度的溝通需求。

檢視軟體設計原則

所謂的軟體設計，關注的是軟體元件間的組成架構與通訊，而程式碼註解、時序圖（sequence diagram）、設計模式等，則是人們用於彼此溝通的工具，程式的通訊與人際的溝通，他們具有相同的本質。

API 設計的基礎原則來自於軟體設計，差異在於 API 的交流對象更廣，不只侷限在組織內部，也擴及到外部的開發者，因此儘管模組化、封裝、低耦合、高內聚等這些軟體設計的概念也適用於 API 設計，開發者也早已熟悉，但下面我們會一一檢視將它們用於 API 設計時應額外考量的點。

模組化

「模組」是軟體程式中最小的構成單元，模組由一系列的類別、方法、函式所構成，模組可對外提供局部（local）、公開（public）的 API，透過 API 實現某些特定的機能或商業能力，模組有時也被稱為**元件**或**函式庫**。

大部分的程式語言都有提供模組化的特性，他們多半是用套件（package）或命名空間（namespace）來定義模組，當我們將那些彼此相關的程式碼歸納在同一個命名空間之下，這就是所謂的高內聚性（high cohesion），再透過程式語言的存取修飾器（access modifier）將模組內的實作細節隱藏起來。以 Java 為例，它可以用 `public`、`protected`、`package`、`private` 這些關鍵字來設定一個模組的存取範圍，實現模組間的去耦合化（loose coopling）。

當許多的模組整合在一起，即成為一個系統，而子系統則是介於系統與模組間的單位，它也被視為一個整合了幾個相關模組的大模組，參見圖 1.1。

模組與封裝的概念也同樣適用於 Web API，它能幫助我們劃分每個 API 所擔負的功能，一方面確保了每支 API 的角色分明，一方面展現了 API 的數位能力，並且隱藏了內部細節，開發者將能更容易理解 API，並且更快的對其上手。

封裝

封裝用於隱藏元件的內部細節，我們會用範圍修飾器（scope modifier）來限定一個模組可被存取的範圍，把一個有對外 API 的模組將內部細節隱藏起來，就算內部實作被修改，但只要不改變對外的 API 特性，任何依賴此 API 的外部程式就不會受到任何影響，有時我們將封裝稱為**資訊隱藏**，這是一個來自 1970 年代由 David Parnas 所提出的軟體開發概念。

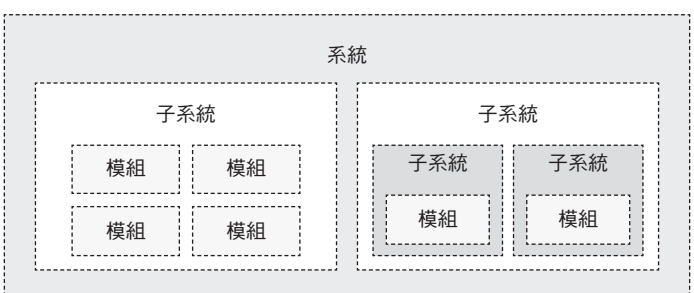

圖 1.1 模組與模組結合成更大的單元，最終成為完整的系統。

封裝的概念在 Web API 被進一步的延伸，不僅是程式碼的實作細節被隱藏，包括 Web 框架、系統內的類別 / 物件、資料庫的設計等等也都被隱藏，形成更進一步的去耦合化，用戶只需要關心如何與 API 進行通訊，而不用去管 API 內部的那些實作

細節，像是付款閘道內部是怎麼運作之類的，藉由封裝的機制，API 用戶可以忽略那些內部細節，把心力投注在他們真正該重視的商業目標上。

高內聚低耦合

內聚性（cohesion）指的是一個模組中的程式碼只與該模組內的其他程式碼有關，不與模組外的程式碼相關，以避免產生出俗稱「義大利麵」式盤根錯節、低內聚性的程式碼。

耦合性（coupling）是我們用來衡量模組間彼此獨立性的詞彙，高耦合的意思是兩個模組內的實作細節彼此高度依賴，反之低耦合指的是兩個元件的內部細節不互相依賴，彼此只透過外部的公開介面或程式語言的 API 溝通。

圖 1.2 展示了模組內高內聚、模組間低耦合的示意圖。

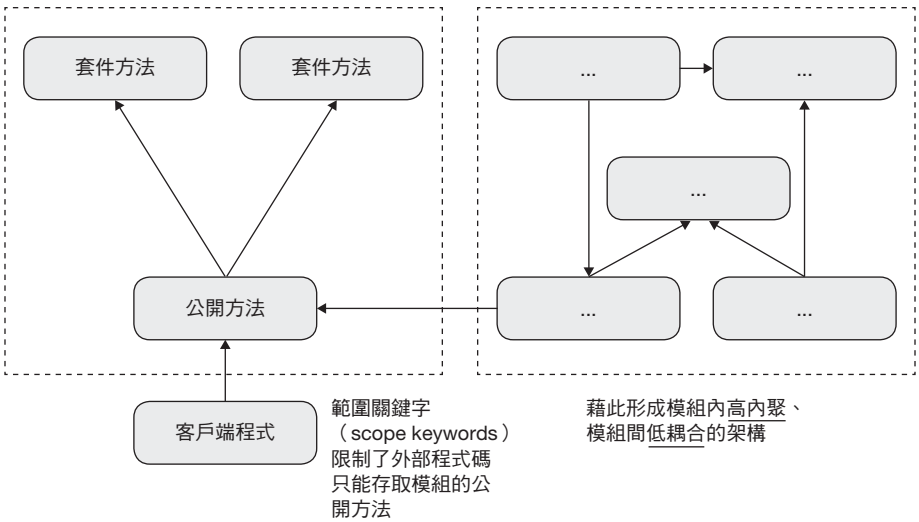

圖 1.2　API 模組化設計的基礎高內聚與低耦合

Web API 延續了這樣的概念，在 API 設計上我們也會將相關性高的 API 操作群集起來，並把 API 與 API 間的內部細節封裝起來，形成高內聚、低耦合的架構。

Resource-Based API 設計

數位世界的文件、圖像，或者實體世界的人員、物件，我們都可統稱為實體，而資源可以用於表示一個實體或是一系列實體的集合，每個資源都會被賦予特有的名稱或識別代號，而有時候資源也指涉商業流程、工作流程等更抽象的概念。

Resource-based API（以資源為基礎的 API）關注的是網路兩端的資源互動行為，與內部的資料庫結構或程式物件無關，resource-based API 為每種資源提供特定的資源行為，以及提供特定的格式，例如 JSON 或 XML，讓外部程式，不論是 Web app 或手機 app 都能與資源進行互動。

資源≠資料模型

必須意識到的是，資源並不全然等於資料庫中的資料模型，資料庫的資料模型更著重在表達資料庫中的欄位、型別等資料結構上的規劃，以及讀寫效能、報表生產等的效率。

儘管資料也是 API 的一部分，但 API 設計中的資料不應該全盤照抄資料庫的資料規劃，資料庫的資料有它自己的特殊需求，例如讀寫效能、儲存最佳化、查詢效能等，著重的是更底層的效能考量。

如同人們對程式語言及框架的愛好不斷改變一般，人們對資料庫的選擇也不斷在變，一旦把 API 的資料直接取自底層的程式物件或資料庫，那麼一旦內部實作或欄位設計變更，那 API 的資料格式也必然得跟著變更，而這很有可能破壞既有的串接。

在 Web API 設計上，我們追求的是交付正確的結果、好的使用體驗、選擇適當的通訊協定、避免被特定語言綁定等，因為 API 涉及的是跨系統間的資料交換，它應該盡可能地使用固定的資料格式，而底層的資料模型缺不具備這樣的特性，他們很可能會隨著不同的取用需求而不斷增減。

雖然底層數據模型的決定的確可能會影響 API 的資料格式，但 API 在設計之時應盡量避免被特定的資料庫特性綁定。

如果直接在 API 暴露底層資料庫的資料模型會有哪些問題？

程式碼變動問題：一旦資料庫的欄位設計改動，那 API 層的資料格式也必須跟著改動，表示 API 總是得跟著資料庫一起異動，而與 API 串接的那些苦主也得被迫跟著變動，造成牽一髮動全身的問題，儘管可以用一些防損毀層（anticorruption layer）手法讓負責資料的程式碼獨立出來，然而這無法從根本解決問題，一直改來改去只會增加無謂的維護成本。

冗餘的互動問題：直接將資料表對應到 API 層，那麼客端就必須自行向 API 發送多個請求，才能組出它想要的資料，而這樣多次的查詢造成 n + 1 的效率問題，最終拖垮資料庫，對兩方都沒益處。

資料不一致問題：承上，客戶端得發出多次請求才能自己組出想要的資料，而資料庫的多個不同資料表可能因為種種的因素而有著些微的差異，導致客戶端無法組出正確的資料。

回傳值不明確問題：在資料庫我們有時會為了性能而規劃出特別的欄位，例如以某個型別為 `CHAR(1)` 的欄來表示一筆紀錄的狀態，但這種設計對收到資料的客戶端來說沒有語意，難以理解。

機敏資料外洩問題：直接在 API 暴露資料庫層將導致資料外洩，可能一句 `SELECT * FROM [table name]` 就讓外人取得全部的資料，包括那些原本不應該公開的資料，例如會員的個資，另外也有可能讓駭客摸進系統內部。

資源≠物件或領域模型

資源也並不等於程式中的物件，物件一般用於表達商業邏輯中的特定資料結構，或映射來自資料庫的資料結構，然而，將程式中的物件或資料結構直接對外暴露會也會帶來與上面相同的問題：程式碼變動、冗餘的互動，以及資料不一致等，這些都會對 API 的使用性造成負面的影響。

與之類似的，領域模型（domain model）通常由一系列物件所組成，用於表示商業邏輯中某種特定領域的資料模型，他們根據系統設計上的不同而有不同的使用模式，有時可能會在多筆交易中使用相同的領域模型，在 Web API 方面，考慮到 API 的效能，我們並不建議直接對外暴露內部的物件或領域模型，而最好是在不同的交易事務間劃出適當的區隔。

切記，API 用戶無法看到 API 後面的程式與資料模型，也不可能參與過 API 的設計會議，他們不會理解 API 設計過程中的那些如何與為何，因此好的 API 應該在設計上避免暴露這些只有內部人士才知曉的細節，而要著重在 API 對外訊息交換的設計上。

在 Resource-Based API 交換資訊

用戶或外部系統可透過 Resource-based API 與企業建立起對話的機制，例如某個專案管理應用的用戶可對 API 服務發起像圖 1.3 那樣的對話。

API 用擬人式的對話看起來很荒唐？這與我們一般認知的**物件導向**的觀念差距很大？其實物件導向的發明者 Alan Kay 當年在創建這個詞彙時，並不只談及物件的繼承與多型特性，他也說過物件導向程式就好比在元件中傳遞訊息：

> 我很抱歉一直以來都過分強調「物件」的概念，這使得很多人都只關注到整體概念中的一小部分，而其實真正的精神在於「交換資訊」[3]。

如同 Kay 對物件導向所給出的願景，Web API 就是以訊息為基礎的，服務接受外部的請求訊息，並用另一則訊息做出回應，大部分的 Web API 發送請求後，便開始等待回應，如此以同步（synchronously）的模式來做這樣的訊息交換。

API 設計要考量的是系統與系統間的訊息交換機制，這樣的機制必須要能滿足用戶、夥伴、員工的需求，而優秀的 API 設計還要額外考量到為未來的改版留下空間。

3 Alan Kay, “Prototypes vs Classes was: Re: Sun’s HotSpot,” Squeak Developer’s List, October 10, 1998, http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html.

圖 1.3 API 客戶端與服務端互動的示例，如同用戶在與服務員對話。

Web API 設計原則

API 設計必須在功能強大與簡單易用中間取得一定的平衡，而這來自於一些堅實的基礎，我們將介紹能達成此目標的五個基礎原則，詳述如下：

原則一：設計 API 千萬不要孤軍奮戰，要成就霸業必然得靠眾志成城。（見第 2 章）

原則二：API 設計是目標導向的，聚焦在目標並確保 API 對他人是有價值的。（見第 3 至 6 章）

原則三：根據需求決定 API 設計，完美的 API 風格是不存在的，應該根據需求來決定適合的 API 風格，不論是 REST、GraphQL、gRPC，或任何一種新風格、新玩具，都應該先了解需求與風格的特性，再選用最適合的方案。（見第 7 至 12 章）

原則四：API 最重要的 UI 叫文件，它應該被擺在第一順位，而不是拖到開學前一天才開始寫。（見第 13 章）

原則五：API 是永存的，設計時仔細規劃，而後加以迭代改進，才能讓 API 既穩定又保有彈性。（見第 14 章）

總結

成功的 Web API 設計必須考量三個面向：數位能力、產品思維、開發體驗，來自三方面的觀點與需求對 API 設計構築出了一系列複雜的課題，企業組織必然得加以重視，開發者、架構師、外部領域專家、產品經理，不同的成員們必須互相合作，才能設計出符合市場需求的 API。

此外，Web API 設計繼承了那些軟體設計的基本概念，如模組、封裝、低耦合、高內聚等，API 應該要隱藏內部實作細節與資料結構，聚焦於系統間的資料交換機制，才能設計出靈活可變的 API。

該如何將商業需求轉換成 API 設計？如何使 API 為客戶、夥伴、員工所用？這將是下一章的主題，在第 2 章我們會介紹一種能將商業需求與產品需求轉換成 API 設計的方法，請前往第 2 章了解更多細節。