序

十多年前，當我在 *Financial Times*（金融時報）建立我的第一個 API 時，尚未存在有太多的 API。我們是在一種單體架構（monolithic architecture）上建置的，而該 API 的存在只是為了讓外部第三方得以存取我們的內容。

但現在，API 無處不在，它們是你建立一個系統時成功的核心所在。

這是因為，在過去十年中，有幾件事情結合起來改變了我們許多人進行軟體開發的方式。

首先，我們可取用的技術改變了。雲端計算（cloud computing）的興起帶給了我們自助服務（self-service）、視需要供應（on-demand provisioning）。自動化的建置和部署管線使我們能夠進行持續的整合和部署，而容器和相關技術（如協調）讓我們能作為一個分散式系統運行大量的獨立小型服務。

我們為什麼要這樣做？因為所發生的第二件事：研究表明，成功的軟體開發組織擁有鬆散耦合（loosely coupled）的架構，和有權做出決策的自主團隊。這裡的成功是以對業務的正面影響來衡量的，例如增加市場佔有率、生產力和獲利能力。

我們的架構現在傾向於更鬆散的耦合、更為分散，並以 API 為中心來建置。你會希望 API 是可發現（discoverable）並一致的，即使它們意外改變或消失，也不太可能給消費者帶來問題。其他的任何東西都會把工作耦合在一起，拖慢你團隊的腳步。

在本書中，James、Daniel 和 Matthew 為如何構建有效的 API 架構，提供了全面且實用的指南。他們涵蓋了很多主題，從如何建置和測試一個單獨的 API，到你在其中部署它們的生態系統、有效發佈和營運它們的方法，還有或許最重要的，如何使用 API 來發展你的架構。我在 *Financial Times* 建立的第一批 API 已經不復存在，而且我們是從頭開始建立那些系統，其成本非常昂貴。

James、Daniel 和 Matthew 提供了一個範本，說明如何利用 API 作為關鍵工具來處理無法避免的變化並發展你的系統。

軟體架構（software architecture）被定義為那些既重要又難以改變的決策。那些決定將左右你專案的成功或失敗。

作者的焦點並非抽象的架構，而是你如何在自己的組織內套用架構。決定採用 API 閘道（API gateway）還是服務網格（service mesh），以及採用哪一種，正是你應該謹慎對待並仔細評估的那種難以反悔的決策。James、Daniel 和 Matthew 在他們認為合適的地方給予強而有力且有所主張的指導，而在選項不那麼明確之處，他們提供了一個框架，來幫助你做出適合你情況的最佳選擇。

他們用一個實際的現實案例研究來闡明重點，這個案例研究採用了那些概念，並示範如何在實務上讓它們發揮作用。他們的案例研究在全書中不斷演進，與真實系統發展的方式相同。作者表明，你不必在事先就做好所有的事情，可以在發現需求的時候，逐塊發展你的架構，提取出服務並添加 API 閘道和服務網格等工具。

我建立第一個 API 時，犯了很多錯誤。多希望當時能有一本像這樣的書，幫助我了解可能被絆倒的地方，並引導我做出明智的抉擇。

只要 API 在你的系統中扮演了主要的角色，而你正在領導相關的工作，我就會向你推薦此書。有了它，你應該就能開發出前後一致的一套工具和標準，以支援你組織中正在建置 API 的每個團隊。

— *Sarah Wells*
QCon London 會議聯合主席、
獨立顧問與 *Financial Times* 前技術總監，
英國雷丁（*Reading, UK*），
2022 年 *9* 月

前言

我們為何撰寫這本書？

2020 年初，我們參加了在紐約舉行的 O'Reilly Software Architecture 會議，Jim 和 Matt 主持了一個關於 API 的專題研討會，並以 API 鬧道為主題發表了演說。Jim 和 Daniel 在倫敦的 Java Community 就認識了，而就像在許多軟體架構活動中一樣，我們聚在一起談論 API 架構相關的想法和理解。我們在走廊上交談時，幾位與會代表走過來和我們聊起了他們在 API 方面的經驗。人們紛紛詢問我們對他們 API 歷程的想法和指引。就在此時，我們認為寫一本以 API 為主題的書籍，將有助於與其他架構師分享我們在會議上探討的內容。

為什麼要閱讀這本書？

本書旨在提供關於設計、營運和發展 API 架構的全貌。透過我們的寫作和附帶的案例研究來分享經驗和建議，該案例模擬了現實生活中的一個活動管理會議系統（event-management conference system），讓與會者能夠查看和報名演講議程。案例研究貫穿全書，目的是讓你探索抽象概念有時是如何轉化為實際應用的。如果你想對此案例研究的演變有一個高階的概觀，你可以在第 10 章中找到。

我們也相信，讓你做出自己決定的重要性。為了支持這點，我們將會：

- 在我們有強烈的建議或指引時明確表達。
- 強調你可能遭遇問題而必須慎重的領域。

- 提供 Architecture Decision Record（ADR，架構決策紀錄）指導方針[1]，以幫助你在已知的架構環境之下做出可能的最佳決策，並提供指引來讓你知道要考慮些什麼（因為有時答案會是「視情況而定」）。
- 指出參考文獻和實用的文章，在那裡可以找到更深入的內容。

本書不只是一本綠地技術（greenfield technology）書籍。我們認為涵蓋現有的架構，並以演化的方式朝向更合適的 API 架構前進，將為你提供最大的好處。我們也試著平衡這一點與 API 架構領域最新的技術展望和發展。

本書是為誰而寫的

雖然我們在創作本書時，心裡有一個最初的人物形象，但在寫作和審閱的過程中，出現了三個關鍵的角色：開發人員、意外的架構師和解決方案或企業架構師。我們在下面章節中概述這些角色，目的是讓你不僅能認同至少其中一個角色，而且還能透過這些角色提供的不同觀點來審視每一章。

開發人員（Developer）

你很可能已經有好幾年的專業程式設計經驗，對常見的軟體開發挑戰、模式和最佳實務做法有很好的理解。你越來越意識到，軟體產業朝著構建服務導向架構（service-oriented architecture，SOA）和採用雲端服務的方向發展，這意味著建置和營運 API 正迅速成為一項核心技能。你熱衷於學習設計有效 API 和測試它們的相關知識。你想探索各種實作選項（例如同步或非同步通訊）和技術，並學習如何提出正確的問題，評估哪種做法最適合特定的情境。

意外的架構師（Accidental Architect）

你很可能已經開發了多年的軟體，並經常領導團隊或作為常駐的軟體架構師（即使你沒有正式的頭銜）。你了解核心的架構概念，例如為高凝聚力（high cohesion）和鬆散耦合（loose coupling）而設計，並將這些應用於軟體開發的所有面向，包括系統的設計、測試和營運。

1 你將在「導論」中了解關於 ADR 的更多資訊及其對做出和記錄架構決策的重要性。

你意識到你的角色越來越專注在結合系統以滿足客戶的需求。這可能包括內部構建的應用程式和第三方的 SaaS 類型產品。API 在成功整合你們系統和外部系統方面扮演重要角色。你想學習關於支援技術（例如 API 閘道、服務網格等）的更多知識，也想了解如何營運和保護基於 API 的系統。

解決方案或企業架構師（Solutions/Enterprise Architect）

你已經設計並構建企業軟體系統好幾年了，而你的工作頭銜或角色描述中很可能有**架構師**（*architect*）這個詞。你負責軟體交付的整體大局，通常在一個大型組織或相互關聯的一系列組織之範圍內工作。

你意識到服務導向架構風格最新修訂版本，對於軟體之設計、整合和治理所帶來的變化，而你也了解到 API 對你組織的軟體策略之成功是至關重要的。你熱衷於學習關於演化模式（evolutionary patterns）的更多知識，並了解 API 設計和實作的抉擇將如何影響這一點。你還想關注跨功能的「性質（ilities）」：易用性（usability）、可維護性（maintainability）、規模可擴充性（scalability）和可得性（availability）；並了解如何建置基於 API 的系統，以展示這些特性，並提供安全性（security）。

你會學到什麼

讀完本書之後，你會學到：

- REST API 的基礎知識以及如何建置和測試 API 並做好它們的版本管理
- 建置 API 平台所涉及的架構模式
- 在入口處和在服務與服務之間的通訊中管理 API 訊務的差異，以及如何應用像 API 閘道和服務網格之類的模式和技術
- API 的威脅建模（threat modeling）和關鍵的安全性考量，例如認證（authentication）、授權（authorization）與加密（encryption）。
- 如何演進現有的系統，朝向 API 和不同的部署目標（例如雲端）發展

而你將會能夠：

- 設計、建置並測試以 API 為基礎的系統
- 從架構的角度幫助實作和推動企業的 API 計畫
- 部署、發佈和配置 API 平台的關鍵元件

- 根據案例研究來部署閘道和服務網格
- 識別出 API 架構中的弱點並實作慎重的安全性緩解措施
- 為新興的 API 趨勢和相關社群做出貢獻

本書不包括什麼

我們很清楚 API 包含了廣大的市場空間，所以希望明確指出本書不包括什麼。這並不意味著我們認為那些主題不重要，反過來說，如果我們試圖涵蓋所有內容，就無法與你有效分享我們的知識。

我們將涵蓋用於遷移（migration）和現代化（modernization）的應用模式，其中包括利用雲端平台的優勢，但**本書並沒有把焦點完全放在雲端技術**。你們中的許多人將擁有混合架構，甚至將所有的系統都託管在資料中心。我們想要確保支援這兩種做法的 API 架構之設計和營運要素都有涵蓋到。

本書不拘泥於特定的語言，但會使用一些輕量化的例子，來展示 API 建置與設計的做法及其相應的基礎設施。本書將更關注做法，程式碼範例可在隨書推出的 GitHub 儲存庫中取用（*https://github.com/masteringapi*）。

本書並不偏重於任一種架構風格，但是我們將討論在哪些情況下，架構的做法可能導致所提供的 API 受到限制。

本書編排慣例

本書使用下列排版慣例：

斜體字（*Italic*）

代表新名詞、URL、電子郵件位址、檔名和延伸檔名。中文以楷體表示。

定寬字（`Constant width`）

用於程式碼列表，還有正文段落裡參照到程式元素的地方，例如變數或函式名稱、資料庫、資料型別、環境變數、述句和關鍵字。

定寬粗體字（**`Constant width bold`**）

顯示應該逐字由使用者輸入的命令或其他文字。

導論

在這個導論中，你將探索 API 的基礎知識以及它們成為架構之旅一部分的潛力。我們會介紹 API 的輕量化定義以及它們在行程內外的使用方式。為了展現 API 的重要性，我們將介紹會議系統（conference system）案例研究，這是一個貫穿全書的可執行範例。行程外的 API（out-of-process API）允許你超越簡單的三層架構（three-tiered architecture），我們將介紹訊務模式（traffic patterns）及其重要性來展示這一點。我們將概述案例研究的步驟，如果你對某個領域感興趣，可以直接跳到後面閱讀。

為了介紹 API 及其相關的生態系統，我們將使用一系列重要的人造物（artifacts）。我們將以 C4 模型圖（C4 model diagrams，*https://c4model.com*）介紹案例研究，並重新審視這種做法背後的具體細節和邏輯。你還會學到 Architecture Decision Records（ADR，架構決策紀錄）的使用，以及清楚定義跨越軟體生命週期之決策所帶來的價值。隨著導論的結束，我們將概述 ADR 指導方針：當答案是「視情況而定」時，我們的做法可以幫助你做出決定。

架構之旅

經歷長途旅行的任何人無疑都會遇到這樣的問題（而且可能是持續的）：「我們到了嗎？」。對於最初的幾次詢問，你會看一下 GPS 或路線規劃器，並提供一個估計值，並期望你在路上不會遇到任何延誤。同樣地，對於開發者和架構師來說，構建基於 API 的架構之旅程可能路途複雜，即使有一個架構 *GPS* 存在，你的目的地會是什麼呢？

架構是沒有目的地的旅程，你無法預測技術和架構做法將如何變化。舉例來說，你可能無法預測服務網格（service mesh）技術會得到如此廣泛的應用，但只要你了解其能力所在，它可能會使你考慮演進發展現有的架構。影響架構變化的不僅僅是技術，新的業務需求和約束也會推動架構方向的改變。

交付增量價值（incremental value）與新興技術相結合的最終效果，導致了演化架構（evolutionary architecture）的概念出現。演化架構是一種逐步改變架構的做法，焦點放在快速改變的能力，以及減少負面衝擊的風險。在這一過程中，我們請你在對待 API 架構時牢記以下建議：

> 儘管架構師們希望能對未來進行策略性的規劃，但不斷變化的軟體開發生態系統使之難以實現。既然我們無法避免變化，我們就得加以利用它。
>
> — Neal Ford、Rebecca Parsons 和 Patrick Kua 所著的《*Building Evolutionary Architectures*》（O'Reilly）

在許多專案中，API 本身是演化式的，隨著更多系統和服務的整合，需要改變以適應。大多數開發者建立的服務都集中在單一功能上，而沒有從消費者（consumer）的角度考慮更廣泛的 API 重複使用。

在 API-First（API 優先）設計這種做法中，開發者和架構師考慮其服務的功能性，以消費者為中心的方式設計 API。API 的消費者可以是一個行動應用程式、另一個服務，甚至是一個外部客戶。在第 1 章中，我們會回顧支援 API-First 做法的設計技巧，及探討如何構建對於改變有耐久性並能為廣大消費者提供價值的 API。

好消息是，你可以在任何時候開始由 API 驅動的架構之旅。如果你負責維護既有的技術庫存，我們將向你展示演進你架構的技巧，以促進 API 在你的平台中的使用。另一方面，如果你很幸運，有一張空白的畫布可以發揮，我們將根據多年的經驗與你分享採用 API 架構的好處，同時也會強調決策的關鍵要素。

API 簡介

在軟體架構（software architecture）領域，有一些術語是非常難以定義的。API 這個術語，也就是 application programming interface（應用程式介面），就屬於這一類，因為這個概念最早出現在非常久遠的 80 年前。已經存在很長時間的術語最終會被過度使用，並且在不同的問題空間有多種含義。我們認為 API 代表的是：

- API 代表底層實作的一個抽象層（abstraction）。
- 一個 API 由一個引入型別（types）的規格（specification）來表示。開發人員可以理解這些規格，並使用工具生成多種語言的程式碼來實作 API 消費者（消耗 API 的軟體）。
- API 定義了語意（semantics）或行為，以有效地為資訊的交換建立模型。
- 有效的 API 設計能夠擴充到客戶或第三方以進行業務整合。

廣義上來講，API 可以分為兩大類，取決於 API 的呼叫是在**行程內**（*in process*）還是在**行程外**（*out of process*）。這裡所指的**行程**（*process*）是指作業系統（OS）的行程。舉例來說，從一個類別到另一個類別的 Java 方法調用（method invocation）是**行程內**的 API 調用，因為該呼叫是由進行呼叫的同一行程來處理的。一個 .NET 應用程式使用 HTTP 程式庫呼叫外部的類 REST API 則是**行程外**的 API 調用，因為該呼叫是由一個額外的外部行程所處理，而非由進行呼叫的行程處理。典型情況下，行程外的 API 呼叫將涉及穿越網路的資料，那可能是區域網路、虛擬私有雲（virtual private cloud，VPC）網路或網際網路（internet）。我們將專注於後一種風格的 API。然而，架構師經常會遇到要把行程內 API 改造為行程外 API 的需求。為了展示這一概念（以及其他概念），我們將創建一個可運作的案例研究，該案例將在本書中不斷發展。

可運作的範例：Conference System 案例研究

我們選擇為一個會議系統（conference system）建立模型以作為案例研究，是因為該領域很容易識別，也提供了足夠的複雜性來進行演化架構的建模（modeling）。圖 I-1 直觀地展示了該會議系統的最頂層，讓我們為要討論的架構設下背景。外部客戶（customer）使用該系統來建立他們的出席者帳號（attendee account）、查看可參加的會議議程，並預訂他們的出席。

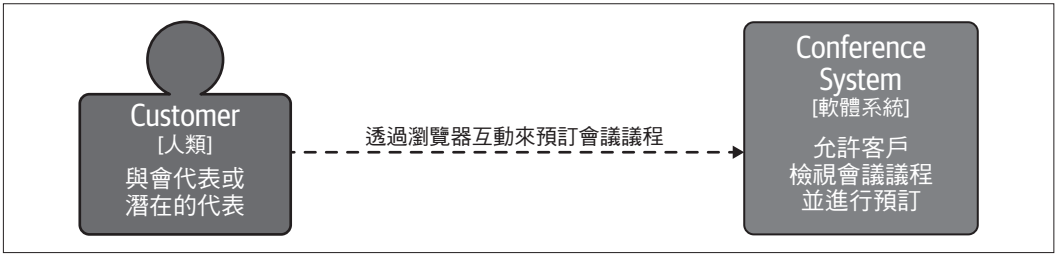

圖 I-1　C4 會議系統的情境圖（context diagram）

讓我們在圖 I-2 中放大 Conference System 方框。展開此會議系統為我們提供了關於其主要技術構件的更多細節。客戶與這個 Web 應用程式進行互動，後者會調用會議應用程式上的 API。會議應用程式使用 SQL 來查詢支援的資料存放區。

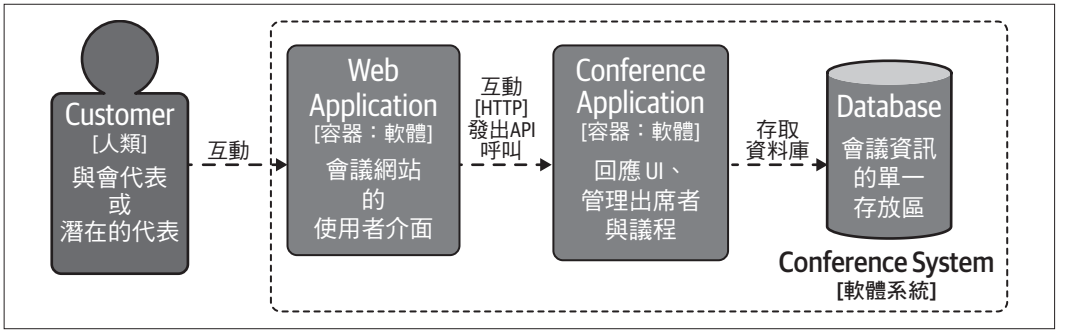

圖 I-2　C4 會議系統的容器圖（container diagram）

圖 I-2 顯示，從 API 的角度來看，最有趣的功能位在會議應用程式容器（conference application container）中。圖 I-3 放大了這個特定的容器，使你能夠探索其結構和互動情況。

在目前的系統中，有四個主要的元件和資料庫（database）。*API Controller*（控制器）面對來自 UI 的所有入站訊務，並判斷要把請求繞送（route）到系統中的何處。這個元件也負責將線路上網路層次的表示方式整列（marshaling）為程式碼中的物件或表徵（representation）。從行程內路由（routing）的角度來看，API Controller 元件是很有趣的，它充當了一個連接點（junction point）或前端控制器（*front controller*）模式。對於 API 的請求和處理來說，這是一個重要的模式，所有的請求都得通過控制器，由它來決定請求的去向。在第 3 章中，我們將探討將控制器從行程中取出的可能性。

Attendee（出席者）、*Booking*（預訂）和 *Session*（議程）元件參與將請求轉化為查詢，並對行程外資料庫執行 SQL 的過程。在現有的架構中，資料庫是一個重要的元件，可能會強制施加關係，例如預訂（bookings）和議程（sessions）之間的約束。

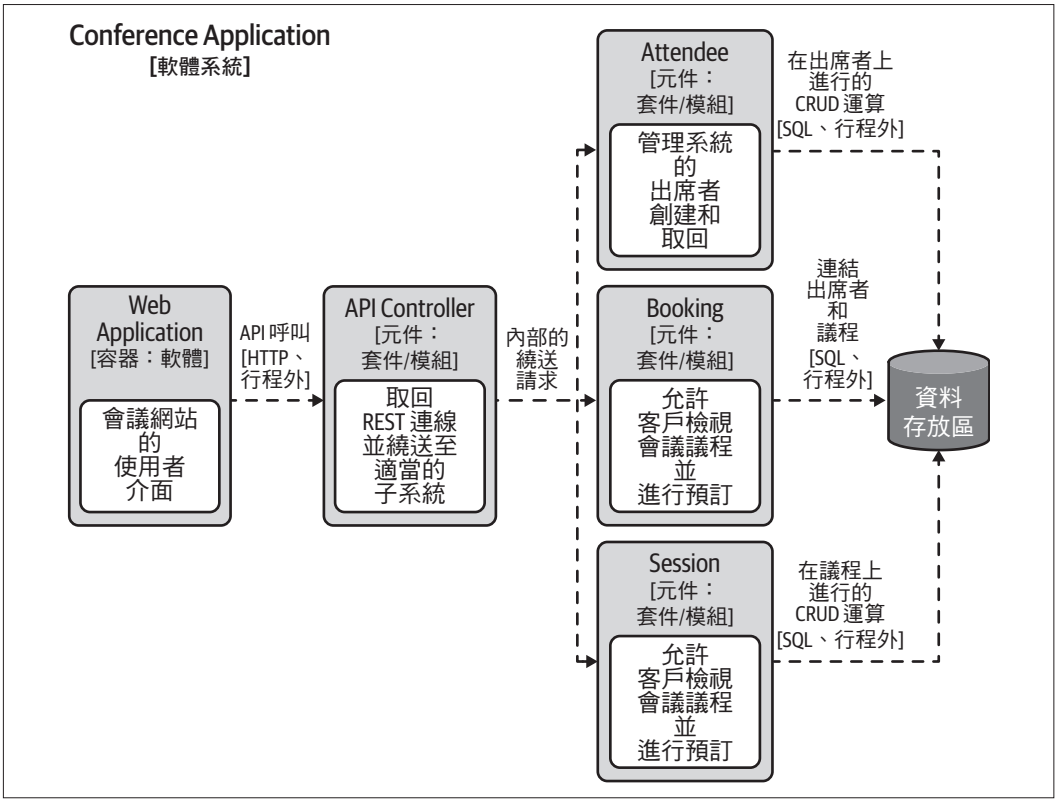

圖 I-3　C4 會議系統的元件圖（component diagram）

既然我們已經鑽研到適當程度的細節，就讓我們重新審視一下此時案例研究中的 API 互動類型。

Conference 案例研究中的 API 類型

在圖 I-3 中，*Web Application* 到 *API Controller* 的箭頭是行程外呼叫，而 *API Controller* 到 *Attendee Component* 的箭頭則是行程內呼叫的例子。在 Conference Application 邊界內的所有互動都是行程內呼叫的例子。行程內的調用（in-process invocation）由實作 Conference Application 的程式語言所明確定義和限制。這種調用具有編譯時期安全性（在這種情況下，交換機制會在編寫程式碼時強制施行）。

改變 Conference System 的理由

目前的架構做法已經用於此會議系統很多年了，然而，會議主辦人要求進行三項改善，這就推動了架構的變更。

- 會議組織者希望建立一個行動應用程式。
- 會議組織者計畫將他們的系統推向全球，每年舉辦數十個會議而不是一個。為了促進這種擴展，他們希望與外部的 Call for Papers（CFP，論文徵集）系統整合，以管理演講者和他們在會議上的演講議程申請。
- 會議組織者想讓他們的私人資料中心退役，轉而在一個具有全球影響力的雲端平台上運行會議系統。

我們的目標是遷移會議系統，使其能夠支援新的需求，而不影響現有的生產系統、或不需要一次性改寫所有內容。

從分層架構到 API 建模

本案例研究的起點是一個典型的三層架構（three-tier architecture），由 UI、伺服端處理層和資料存放區所組成。為了開始討論一個演化架構，我們需要一個模型（model）來思考 API 請求被各元件處理的方式。我們需要一個模型或抽象層，可以適用於公共雲、資料中心的虛擬機器和混合做法。

訊務（traffic）的抽象化使我們能夠考慮 API 消費者（API consumer）和 API 服務（有時被稱為 API 生產者，「API producer」）之間的行程外互動。在服務導向架構（SOA）和基於微服務的架構（microservices-based architecture）之類的架構做法下，對 API 互動進行建模的重要性很關鍵。了解 API 訊務和元件之間的通訊風格將左右著你是實現增加解耦程度所帶來的優勢，或是創造出維護噩夢。

訊務模式（traffic patterns）被資料中心工程師用來描述資料中心內和低階應用程式之間的網路交換。在 API 層次，我們使用訊務模式來描述應用程式組之間的資料流。就本書而言，我們指的是應用程式和 API 層級的訊務模式。

案例研究：一個演化步驟

為了開始考慮訊務模式的類型，在我們的案例研究架構中採取一個小型的演化步驟將是有益的。在圖 I-4 中，我們採取了一個步驟，將 *Attendee* 元件重構為一個獨立的服務，而非傳統會議系統（*legacy conference system*）中的一個套件（package）或模組（module）。會議系統現在有兩個訊務流量（traffic flows）：客戶和傳統會議系統之間的互動，以及傳統系統和出席者系統（attendee system）之間的互動。

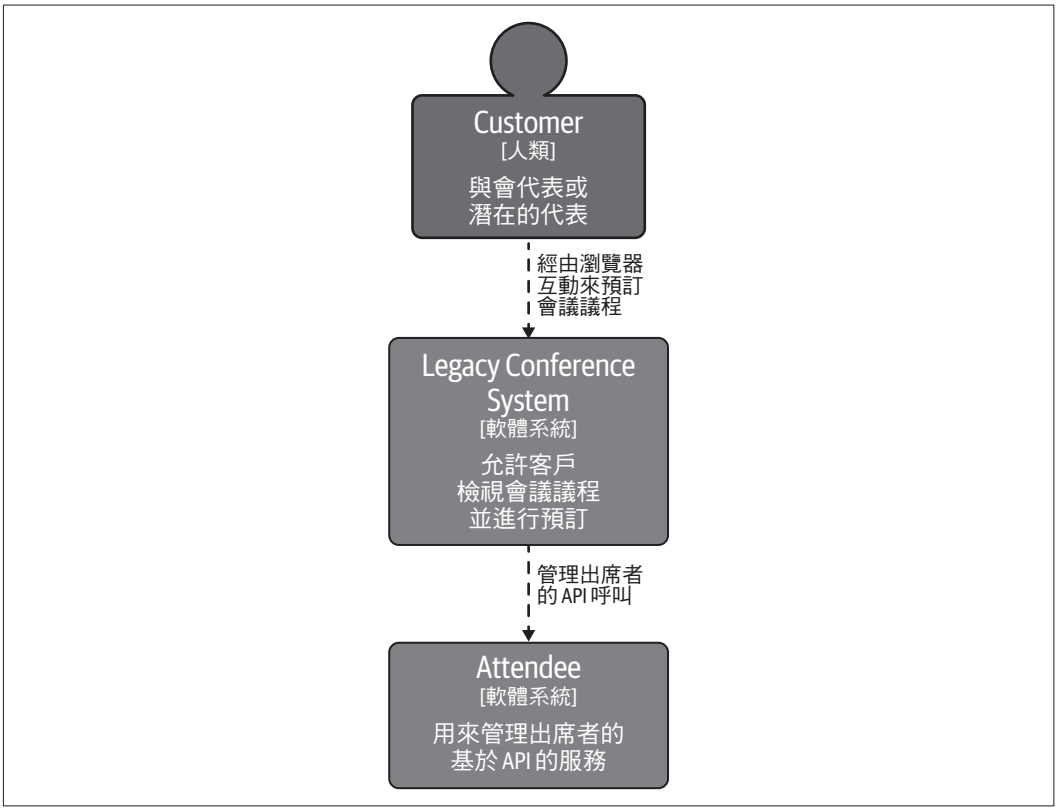

圖 I-4　C4 會議系統的情境：演化步驟

南北訊務

在圖 I-4 中，客戶和傳統會議系統之間的互動被稱為南北訊務（north–south traffic），它代表了一個進入訊務（ingress flow）。客戶使用 UI，它透過網際網路向傳統的會議系統發送請求。這代表了我們網路中的一個公開對外開放的點，並將由 UI 來存取[1]。這意味著處理南北訊務的任何元件都必須對客戶的身分進行具體檢查，並在允許訊務前進到系統中之前包括適當的盤問。第 7 章將詳細介紹南北向 API 訊務的安全問題。

東西訊務

傳統會議系統和 Attendee 服務之間的新互動為我們的系統引入了東西向的訊務。東西訊務（east–west traffic）可以被認為是一組應用程式內**服務對服務**（*service-to-service*）式的通訊。大多數的東西向訊務，特別是當來源是在你更廣泛的基礎設施內之時，都可以在某種程度上被信任。儘管我們可以信任訊務的來源，但仍有必要考慮保護東西訊務的安全性。

API 基礎設施和訊務模式

以 API 為基礎的架構中存在兩個關鍵的基礎設施元件（infrastructure components），它們是控制訊務的關鍵。控制和協調訊務通常被描述為**訊務管理**（*traffic management*）。一般來說，南北訊務將由 API 閘道控制（API gateways），那是第 3 章的關鍵主題。

東西訊務通常由 Kubernetes 或服務網格（service mesh）等基礎設施元件處理，那是第 4 章的關鍵主題。像 Kubernetes 和服務網格這樣的基礎設施元件使用網路抽象化（network abstractions）來進行服務的路由，要求服務在一個受管理的環境（managed environment）中執行。在一些系統中，東西訊務由應用程式本身管理，並實作有服務探索（service discovery）技巧來定位其他系統。

Conference 研究案例的發展路線圖

在這整本書中，你將觀察到此案例研究的以下變化或技術在其中的應用：

- 在第 1 章中，你將探索 Attendee API 的設計和規格。我們還將介紹版本管理（versioning）和對交換建模（modeling exchanges）在 Attendee API 效能上的重要性。
- 在第 2 章中，你將探索契約（contract）和元件測試，以驗證 Attendee 服務的行為。你還將看到 Testcontainers 如何幫助進行整合測試（integration testing）。

1 其目的是讓 UI 存取進入點。然而，它是開放的，有可能被惡意利用。

- 在第 3 章中，你將看到如何使用 API 閘道將 Attendee 服務對外開放給消費者。我們還將演示如何使用 Kubernetes 上的 API 閘道來發展會議系統。
- 在第 4 章中，我們將使用服務網格將議程功能（sessions functionality）從傳統會議系統中重構出來。你還將了解到服務網格如何幫助實作路由、可觀察性和安全性。
- 在第 5 章中，我們將討論功能旗標（feature flagging），以及這如何有助於會議系統的演化發展，避免部署和發佈的耦合。你還將探索在會議系統為發佈（releases）建模的做法，我們將展示 Argo Rollouts 在 Attendee 服務中的應用。
- 在第 6 章中，你將探索如何在 Attendee 服務中應用威脅建模（threat modeling）並減輕 OWASP 安全疑慮。
- 在第 7 章中，你將研究認證（authentication）和授權（authorization），以及如何為 Attendee 服務實作這一目標。
- 在第 8 章中，你將研究如何確立 Attendee 服務的領域邊界（domain boundaries），以及不同的服務模式如何幫上忙。
- 在第 9 章中，你將研究雲端平台的採用，以及如何將 Attendee 服務轉移到雲端中並考慮平台重建（replatforming，或稱「環境遷移」）。

此案例研究和所規劃的路線圖要求我們將架構變更視覺化並記錄決策。這些都是重要的跡證，有助於解釋和規劃軟體專案的變化。我們相信，C4 圖（C4 diagrams）和 Architecture Decision Records（ADR，架構決策紀錄）代表了記錄變化的一種清晰方式。

使用 C4 圖

作為介紹案例研究的一部分，我們揭示了來自 C4 模型（*https://c4model.com*）的三種 C4 圖。我們相信 C4 是與不同的利害關係者溝通架構、背景和互動的最佳說明文件標準（documentation standard）。你可能想知道那 UML 又如何呢？Unified Modeling Language（UML，統一塑模語言）提供了一種廣泛的方言，用於軟體架構的交流。一個主要的挑戰是，UML 所提供的大部分內容並沒有被架構師和開發人員牢記在心，人們很快就會回歸到方框、圓圈或鑽石圖形來進行說明。在進入討論的技術內容之前，如何理解圖表的結構成為一項真正的挑戰。很多圖表之所以會進入專案歷史，只是因為有人不小心錯用了永久性馬克筆而不是可擦式馬克筆。C4 模型提供了一套簡化過的圖表，可作為你的專案架構在不同細節層次的指南。

C4 情境圖

圖 I-1 是用 C4 模型中的 C4 情境圖（context diagram）來表示。這種圖的目的是為技術和非技術觀眾設定背景資訊。許多架構對話都是直接深入到底層的細節，而忽略了高階互動的背景設定。考慮到弄錯系統情境圖的後果，總結一下這種做法的好處，可能得以節省糾正誤解的幾個月工作量。

C4 容器情境圖

圖 I-1 提供了會議系統的全貌，而容器圖（container diagram）則有助於描述架構中主要參與者的技術分工。C4 中的容器（container）被定義為「**為了讓整個系統得以運行而需要執行的東西**」（例如會議資料庫）。容器圖的本質是技術性的，建立在更高階的系統情境圖之上。圖 I-2 是一個容器圖，記錄了客戶與會議系統互動的細節。

圖 I-2 中的會議應用程式容器被單純標示為軟體（*software*）。一般情況下，C4 容器會提供關於容器類型的更多細節（例如 *Java Spring Application*）。然而，在本書中，我們將避免技術細節，除非那有助於展示一個具體的解決方案。API 和現代應用程式的優勢在於，解決方案的空間裡有很大的靈活性。

C4 元件圖

圖 I-3 中的 C4 元件圖（component diagram）有助於定義每個容器中的角色和責任，以及內部的互動。若要查詢一個容器的細節，這種圖就很有用，它還為源碼庫（codebase）提供了一個非常實用的地圖。想想第一次在一個新專案上開始作業的時候：瀏覽**自我說明**（*self-documenting*）的源碼庫是一種方法，但要把所有東西拼湊起來可能很困難。一個元件圖揭露了你用來構建軟體的語言和技術堆疊之細節。為了保持技術的不可知性，我們使用了**套件**（*package*）或**模組**（*module*）這些術語。

使用 Architecture Decision Records

身為開發人員、架構師，甚至是人類，一定都曾遇到過這樣的情況：我們會問「**他們在想什麼？**」。如果你曾經在英國（United Kingdom）的里茲（Leeds）和曼徹斯特（Manchester）之間的 M62 公路上開過車，你可能對其高速公路的建設方式感到困惑。

當你在三車道的高速公路上爬坡時，它開始偏離逆流車群，直到最後，斯科特霍爾農場（Scott Hall Farm）出現在眼前，周圍是大約 15 英畝的農田，貼靠在車輛之間。當地關於此事的傳說指稱，土地的主人很頑固，拒絕搬遷或交出土地，因此工程師們乾脆繞著他的土地興建[2]。五十年後，一部紀錄片浮出水面，揭示了這一事件的真正起因是土地下的地質斷層，這意味著高速公路必須以那種方式修建。當人們猜測為什麼要以特定的方式做某件事時，請預期出現的會是謠言、幽默和批評。

在軟體架構中，會有許多我們必須繞過它們來建置的約束條件，所以很重要的是，確保我們的決策有被記錄下來並且透明。ADR 有助於在軟體架構中使決策更為清晰可見。

> 在一個專案的生命週期中，最難追蹤的事情之一是某些決定背後的動機。新加入專案的人可能會對過去的一些決策感到困惑、不解、高興或憤怒。
>
> —Michael Nygard，ADR 概念的創始人

一個 ADR 中有四個關鍵部分：狀態（status）、情境（context）、決定（decision）和後果（consequences）。一個 ADR 是以提案的狀態（proposed status）創建的，而根據討論，通常會被接受或拒絕。也有可能該決定後來被一個新的 ADR 所取代。情境有助於設定場景，並描述問題或將做出決定的範圍。雖然在 ADR 之前建立一篇部落格文章，然後從 ADR 中連結過去，有助於讓社群關注你的工作，但情境本來就並不是要成為一篇部落格文章或詳盡的描述。決定清楚地闡述了你打算做什麼以及你打算如何做。架構中，所有的決定都帶有後果或取捨（trade-offs），而這些決定有時會因為誤判而付出難以置信的代價。

審查 ADR 時，重要的是看你是否同意 ADR 中的決定，或者是否有替代做法。一個沒有被考慮到的替代做法可能導致 ADR 被駁回。被拒絕的 ADR 有很多價值，大多數團隊選擇保持 ADR 的不可變性，以捕捉觀點的變化。當 ADR 被展示在一個主要參與者可以查看、評論並幫助 ADR 被接受的地方時，其效果最好。

我們經常被問到的一個問題是，團隊應該在什麼時候建立一個 ADR？請確保有在 ADR 建立之前就進行了討論，而且紀錄是團隊集體思考的結果，這是很有用的。向更廣泛的社群發佈 ADR，就能有機會獲得直接團隊以外的回饋意見。

2 當地人的頑固特質助長了這種可能解釋的流傳。

Attendee 的演化 ADR

在圖 I-4 中，我們決定在會議系統架構中採取一個演化步驟。這是一個重大變化，需要有一個 ADR。表 I-1 是擁有該會議系統的工程團隊可能提出的一個範例 ADR。

表 I-1　ADR001 從傳統會議系統中分離出 attendees（出席者）

狀態	提案
情境	會議所有人要求為當前的會議系統新增兩個主要功能，並且得在不破壞當前系統的情況下實作。會議系統需要演化以支援一個行動應用程式並與外部 CFP 系統整合。行動應用程式和外部 CFP 系統都需要能夠存取出席者，才能為使用者登入第三方服務。
決定	我們將採取如圖 I-4 所示的演化步驟，將 Attendee 元件分割出來作為一個獨立的服務。這將允許針對 Attendee 服務的 API-First 開發，並允許從傳統會議服務中調用 API。這也將支援直接存取 Attendee 服務的能力，以便向外部 CFP 系統提供使用者資訊。
後果	對 Attendee 服務的呼叫不會是行程外（*out of process*）的，可能會引入一個延遲，需要進行測試。Attendee 服務可能成為架構中的單一故障點（single point of failure），我們可能需要採取措施來減輕執行單一 Attendee 服務的潛在衝擊。由於規劃中的 Attendee 服務是多消費者模型（multiple consumer model），我們將需要確保有良好的設計、版本控制和測試，以減少意外的破壞性變更。

這個 ADR 中的一些後果是相當重大的，肯定需要進一步討論。我們將把一些後果推遲到後面的章節中討論。

精通 API：ADR 指導方針

在本書中，我們將提供 *ADR 指導方針*（*ADR Guidelines*），以幫忙蒐集對我們所涵蓋的主題進行決策時，要問的重要問題。對基於 API 的架構進行決策可能真的很困難，而且在很多情況下，答案都是「視情況而定」。與其在沒有背景的情況下說「視情況而定」，ADR 指導方針將幫忙描述「取決於什麼」，並協助你做出明智決定。ADR 指導方針可以作為一個參考點，當你面臨特定的挑戰時，可以回過頭來查看，或者事先閱讀。表 I-2 概述了 ADR 指導方針的格式以及你可以預期從中得到什麼。

表 I-2　ADR 指導方針：格式

決定（Decision）	描述你在考慮本書某個面向時可能需要做出的決定。
討論重點（Discussion Points）	本節有助於識別出對你的 API 架構做出決定時應該進行的關鍵討論。在這節中，我們將揭示一些可能影響決策的經驗。我們將幫助你找出關鍵資訊，為你的決策過程提供參考。
建議（Recommendations）	我們將提出你在建立 ADR 時應該考慮的具體建議，並解釋我們提出具體建議的依據。

總結

在這篇導論中，我們提供了一個基礎，談到了案例研究和討論 API 驅動的架構時，我們會採取的做法：

- 架構是一個無止境的旅程，而 API 可以在幫助它演化的過程中發揮重要作用。
- API 是實作的一個抽象層，而且可以是行程內（in process）的，或是行程外（out of process）的。架構師常常發現自己處於向行程外 API 演進的位置，那也是本書的重點所在。
- 會議案例研究是為了描述和解釋概念。在這個導論中，你已經看到了一個小型的演化步驟，拆解出了 Attendee 服務以解決即將到來的業務需求。
- 你已經看到了 C4 圖的前三個層以及它們在分享和交流架構方面的重要性。
- ADR 為決策提供了有價值的紀錄，在專案的生命週期中同時具備現有價值和歷史價值。
- 你已經看到了 ADR 指導方針的結構，它將貫穿全書，以幫忙做出良好決策。

在做了將 Attendee 服務從會議系統中分離出來的決定後，我們現在將探索設計和描述 Attendee API 規格的選擇。