前言

此處對本書的介紹涵蓋以下：

- 本書背景及目的：動機、目標及範圍。
- 誰應閱讀此書？目標讀者與相關使用案例和資訊需求。
- 以一本知識載體的模式而言，本書的組織架構。

動機

軟體和人類一樣使用許多不同的語言溝通。軟體不僅以多種程式語言撰寫，也透過各種協議如 HTTPS，和訊息交換格式如 JSON 溝通。每當有人更新社群檔案（social network profile）和在網路商店刷卡買東西時，HTTP、JSON 和其他技術都會運作：

- 應用程式前端（application frontends），如手機 App 對後端發出交易請求，及在線上購物下單。
- 應用程式部件（application parts）的相互交流，以及與業務夥伴、顧客和供應商的系統交換等長期資料，例如顧客檔案或產品目錄。
- 應用程式後端（application backends）會提供外部服務，例如支付閘道或包含資料和元資料的雲端儲存。

這些情境中的各種大小軟體元件會彼此溝通，以實現個別目標，同時共同服務終端使用者。軟體工程師對這種分散式系統的挑戰回應，是透過**應用程式軟體介面（application programming interface, API）**的應用程式整合。每個整合情境涉及至少兩個對話實體（communication parties）：API 客戶端（API client）及 API 提供者（API provider）；API 客戶端消費 API 提供者暴露的服務，API 文件則控制客戶端與提供者的互動。

軟體元件與人類一樣，會試著努力了解彼此。對設計人員來說，決定訊息內容的適當大小、結構及最適合的會話風格是一件很困難的事。在表達請求及回應的需求

時，會話雙方都不想過於沉默或太多話。有些應用程式整合和 API 設計運作良好，表示這些團體彼此互相了解，且實現它們的目標，讓這些參與者能有效地互動及操作。但也有些 API 會因為不夠明確，而讓參與者感到困惑和壓力；冗長且多餘的訊息可能使傳輸通道超載、引入不必要的技術風險，更會增加開發和營運上的額外工作。

因此，決定整合 API 設計的好壞為何？ API 設計人員如何帶來正面的客戶端開發者體驗？理想上，好的整合架構和 API 設計指南不會依賴任何特定科技或產品，技術和產品來來去去，但相關的設計建議必須能長期保持。若以真實世界來類比，就像是西塞羅（Cicero）的修辭及雄辯技巧，或 Rosenberg（2002）的《非暴力溝通：愛的語言》（Nonviolent Communication: A Language of Life，台灣光啟出版社）原則，不限於英文或其他特定自然語言，不會因語言的演進而退流行。本書旨在為整合人員和 API 設計人員建立一個類似的工具箱和詞彙，能將其知識片段呈現為 API 設計和演進的**模式**，以適用不同的會話典範和技術；範例將以 HTTP 及 JSON Web API 為主。

目標與範圍

我們的任務是透過已驗證、可重用的解決方案元素，去克服設計和演進 API 的複雜性。

> 如何從利益相關人員的目標、架構上的明顯需求和已驗證的設計元素出發，來打造可理解和持續的 API ？

雖然已經有許多 HTTP、Web API 和整合架構的討論及文章，包含服務導向；然而，個別 API 端點和訊息交換的設計至今收到的關注仍不算多：

- 應該暴露多少遠端 API 操作？應該交換請求與回應訊息中的哪些資料？
- 如何確保 API 操作和 API 客戶端與提供者互動的鬆耦合（loose coupling）？
- 何謂合適的訊息表現（message representations）：是扁平或巢狀階層訊息表現？如何就表現元素的意義達成共識，以便可以正確和有效地處理這些元素？

- API 提供者是否要負責處理客戶端提供的資料？這些資料可能會改變提供者端的狀態和連接到後端系統。或提供者是否應該僅對客戶端提供共享資料存儲？
- 如何以兼顧擴展與相容性的可控方式引入 API 變更？

本書會藉由描繪出對特定需求情境中重複發生的特定設計問題已驗證解決方案，來幫助回答這些問題；會將重心放在遠端 API 而非程式內部 API，旨在同時改善客戶端和提供者端的開發者體驗。

目標讀者

本書針對想要改善技能和設計的中級軟體專業人員。所介紹的模式主要針對那些對平台獨立架構知識（platform-independent architectural knowledge）感興趣的整合架構師、API 設計人員和 Web 開發人員；後端對後端整合人員，和支援前端應用程式的 API 開發人員，也能從這些模式所傳授的知識中獲益。因為我們專注在 API 端點粒度和訊息中的資料交換，所以 API 產品負責人、API 審查人員及雲端服務用戶及供應商也可說是本書目標讀者。

> 如果你是一名具有一定經驗的軟體工程師，例如開發者、架構師或產品負責人，已熟悉 API 基礎且想改進 API 設計能力，包括訊息資料的規約設計和 API 演進，本書就是為你準備的。

學生、講師、軟體工程研究人員也可從本書的模式與表現找到有用之處。我們提供了 API 基礎介紹和 API 設計的領域模型，因此不必事先閱讀其他初學者入門書籍，也能夠理解本書及其模式。

了解可用模式的優缺點可改進 API 設計與演進效率。將本書中的模式套用在適用的特定情境時，API 及服務的開發、使用和演進會變得容易許多。

使用案例

本書目的是讓設計及使用 API 成為愉快的體驗，為此介紹 3 個主要使用案例及其模式：

1. **促進 API 設計討論及工作坊**，透過建立共同詞彙，指出需要的設計決策，並分享可用選項及相關取捨。有了這些知識，API 提供者能同時在短期和長期下，提供符合客戶品質及風格需求的 API。
2. **簡化 API 設計審查及加速 API 客觀比較**，以確保 API 品質，並以向後相容和可擴展的方式演進。
3. **以平台中性的設計資訊提升 API 文件品質**，如此 API 客戶端開發者便能快速掌握 API 功能及限制。模式設計為可嵌入到 API 規約中，且可在既有設計中觀察。

我們提供一個虛構的研究案例，和兩個真實世界的模式，以故事方式來說明和推動模式的使用。

讀者不需事先知道任何特定的建模方法、設計技巧或架構風格。然而以下概念就有其必要性，例如校正－定義－設計－改善（Align-Define-Design-Refine, ADDR）流程，領域驅動設計（DDD）及責任驅動設計（RDD）；附錄 A 也會簡短地討論這些概念。

現有設計探索（與知識鴻溝）

市面上有不少關於 API 深度見解的好書：《RESTful Web Services Cookbook》（Allamaraju 2010）解釋如何建立 HTTP 資源 API，例如要選用 POST 或 PUT 等哪一種 HTTP 方法。其他書以路由、轉換及傳遞保證，解釋非同步訊息的運作方式（Hohpe 2003），《Strategic DDD》（Evans 2003; Vernon 2013）幫助你了解 API 端點及服務識別。服務導向架構（Service-oriented architecture）、雲端計算及微服務基礎設施模式已發布，結構化資料存儲，如關聯資料庫、NoSQL 也已得到廣泛記錄，以及可用的分散式系統的整體模式語言（Buschmann 2007）；最後，《Release It!》（Nygard 2018a）廣泛地涵蓋營運和部署到生產環境的設計。

在現有書籍中也能找到 API 設計流程，包括目標驅動端點識別（goal-driven endpoint identification）和操作設計，例如《Web API 設計原則：API 與微服務傳遞價值之道》（Principles of Web API Design: Delivering Value with APIs and Microservices, Higginbotham 2021）的四階段及七步驟。《The Design of Web APIs》（Lauret 2019）提出 API 目標畫布（API goal canvas），《Design and Build Great Web APIs: Robust, Reliable, and Resilient》（Amundsen 2020）搭配 API 故事。

雖然有這些寶貴的設計建議資源，但對遠端 API 設計的討論還是不足，尤其是 API 客戶端和提供者之間往來的 API 請求與回應訊息結構。《Enterprise Integration Patterns》（Hohpe 2003）雖然提到三種模式的訊息類型：事件、命令及文件訊息，但並未提供其內部運作的更多細節。然而系統之間交換的「外部資料」與程式內處理的「內部資料」不同（Helland 2005）。兩種資料類型的可變性、生命週期、準確性、一致性及保護需求有明顯的差異。例如在庫存系統內增加存貨計數的架構設計通常比較簡單，而製造商與物流公司透過遠端 API 交換產品價格和出貨資訊的架構設計則比較複雜。

訊息表現設計：外部資料（Helland 2005），或 API 的「發布語言」（Published Language）模式（Evans 2003）為本書主要關注領域。這縮小了 API 端點、操作和訊息設計的知識鴻溝。

知識分享載體的模式

軟體模式是擁有超過 25 年歷史紀錄的複雜知識分享工具。我們決定用模式格式來分享 API 設計建議，因為模式名稱旨在形成一個領域詞彙，一個「共通語言」（Ubiquitous Language, Evans 2003）。例如，企業整合模式（enterprise integration pattern）已經成為佇列訊息（queue-based mesaging）的通用語；在訊息框架和工具中甚至已經實現了這些模式。

模式是從實務經驗挖掘出來而非創造出來的，並因同儕反饋而更加穩固。模式社群（patterns community）發展出一套回饋流程的實務；領導及寫作者工作坊（shepherding and writers' workshops）是特別重要的兩個實務（Coplien 1997）。

每一個模式中心都是一對問題－解決方案（problem-solution pairs），其力量與結果的討論支持了決策，例如期望和實現的品質特性，以及特定的設計缺點。替代方案的討論，以及相關模式的指標和可能的實現技術會讓全貌更為完整。

注意，模式非旨在提供一個完整的解決方案，而是作為用於特定情境的 API 設計草稿。換言之，模式的邊界是軟性的，能勾勒出可能解法而非盲目複製的藍圖。如何利用模式來滿足專案或產品需求，仍是 API 設計人員和負責人的責任。

業界及學界已使用模式很長一段時間，其中有些人以模式來撰寫程式、設計架構及整合分散式系統（Voelter 2004; Zimmermann 2009; Pautasso 2016）。

我們發現模式的概念相當契合之前「目標與範圍」與「目標讀者」兩節中的使用情境。

微服務 API 模式

微服務 API 模式（Microservice API Patterns, MAP）從訊息交換的角度，在暴露 API 和消費 API 時，提供了 API 設計和演進的綜合觀點。這些訊息與酬載（payload）會結構化為表現元素。由於 API 端點及操作有不同的架構**職責（responsibilities）**，而讓表現元素出現不同的**結構（structure）**和意義。訊息結構強烈影響 API 設計時間與運行**品質（qualities）**及其基礎的實現；例如，少量的大型訊息和多量的小訊息對網路和端點的工作量影響不同。最後，成功的 API 隨時間**演進（evolve）**，而隨時間發生的改變則需要有所管理。

我們選擇以 MAP 為縮寫，是因為「地圖」（map）這個字也有提供方位和指引的意義，正如同模式語言，能指導讀者在抽象解決方案空間中找到可用選擇。API 本身也有和地圖一樣的本質，負責將請求導引到背後的服務實現。

我們承認「微服務 API 模式」有點標題黨（click-bait）。為了避免微服務在本書出版後不再流行，我們保留重新命名的權利。例如改為「訊息 API 模式」（Message API Patterns），也可適切地畫出模式的範圍。但本書大多以 MAP 代稱「模式語言」或「我們的模式」。

本書的模式範圍

本書為一項自願者專案的結果，開始於 2016 年秋天，專注於網路 API 及其他遠端 API 的設計和演進，以及訊息職責、結構、品質與服務發展，本專案希望能回答以下問題：

- 每一個 API 端點在架構上扮演何種角色？端點角色及操作職責如何影響服務大小及粒度？
- 請求與回應中的合理元素數量是多少？元素如何結構化？如何分組及標記補充訊息？
- API 提供者如何在實現特定品質水準的同時，以有效方式使用其資源？如何傳遞和考慮品質取捨？
- API 專業人員如何處理生命週期問題，例如支援期間和版本控制？如何促進向後相容和通知破壞性變更（breaking changes）？

我們研究許多網路 API 和 API 相關規範，在撰寫任何模式以前搭配自身經驗，來搜集模式，也從業界的公開網路 API 和整合專案觀察到許多模式使用，許多模式的中間版本，經過 2017 至 2020 年於 EuroPLoP 舉辦的領導人及寫作者作坊程序[1]，之後以會議論文[2]形式出版。

閱讀切入點、閱讀順序及內容組織

要操作一個複雜的設計空間來解決棘手問題時（Wikipedia 2022a），通常很難見樹又見林；尤其 API 設計有時確實相當棘手（wicked），不可能也不期望將解決問題的活動順序標準化。因此，本書的模式語言有多個閱讀切入點，書中的每一部皆可，附錄 A 也有更多建議。

本書分為三部分：**第一部分「基礎及敘事」，第二部分「模式」，第三部分「模式實戰」（現在和過去）**。圖 P.1 顯示各部分與其章節的邏輯依賴。

1 https://europlop.net/content/conference

2 本書不會納入過多模式選集；這些資訊可以從網路上及 2016 至 2020 的 EuroPLop 會議取得；「補充資源」也能找到額外實作提示。

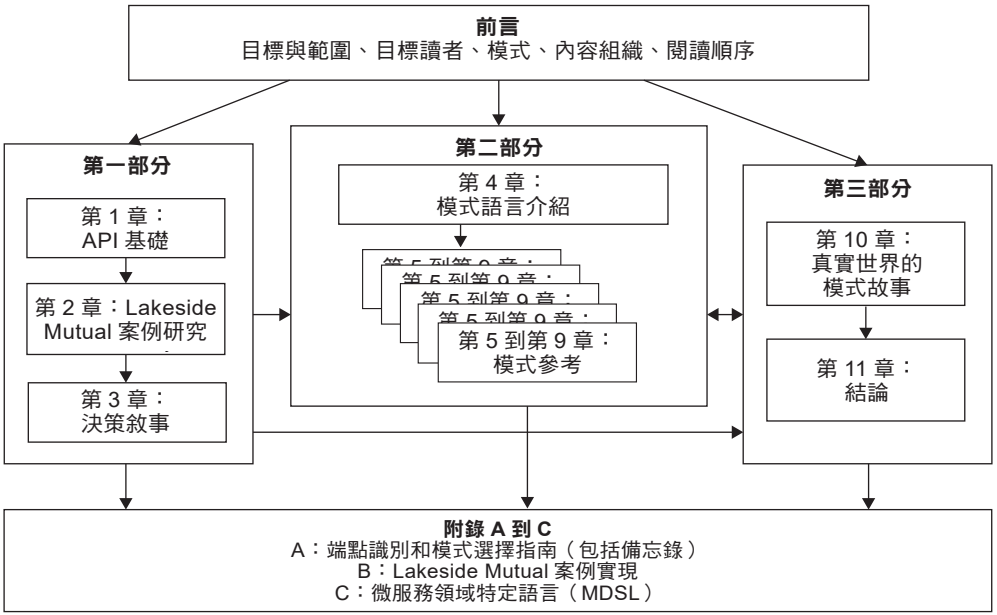

圖 P.1 本書各部分及依賴

第一部分介紹 API 領域概念，從**第 1 章〈應用程式介面（API）基礎〉**開始，**第 2 章〈Lakeside Mutual 案例研究〉**是本書主要範例來源 Lakeside Mutual 案例的初登場，包括其業務背景、需求、現有系統和初步 API 設計。**第 3 章〈API 決策敘事〉**中提供決策模型，展示語言中模式的互相關聯，這章也提供模式選擇準則和 Lakeside Mutual 案例中介紹的決策方式。在閱讀本書和在實務中套用這些模式時，這些決策模型可作為導覽幫助。

第二部分為模式參考，從**第 4 章〈模式語言介紹〉**開始，接續 5 章的全部模式：**第 5 章〈定義端點類型及操作〉，第 6 章〈設計請求與回應訊息表現〉，第 7 章〈改善訊息設計品質〉，第 8 章〈演進 API〉，及第 9 章〈API 規約文件與傳達〉**。

圖 P.2 說明本部各章的閱讀路徑；例如你可以學習基本結構模式，像是第 4 章的*原子參數*（*Atomic Parameter*）及*參數樹*（*Parameter Tree*），然後移往*元素刻板*（*element stereotypes*），像是第 6 章的 *ID 元素*（*ID Element*）與*元資料元素*（*Metadata Element*）。

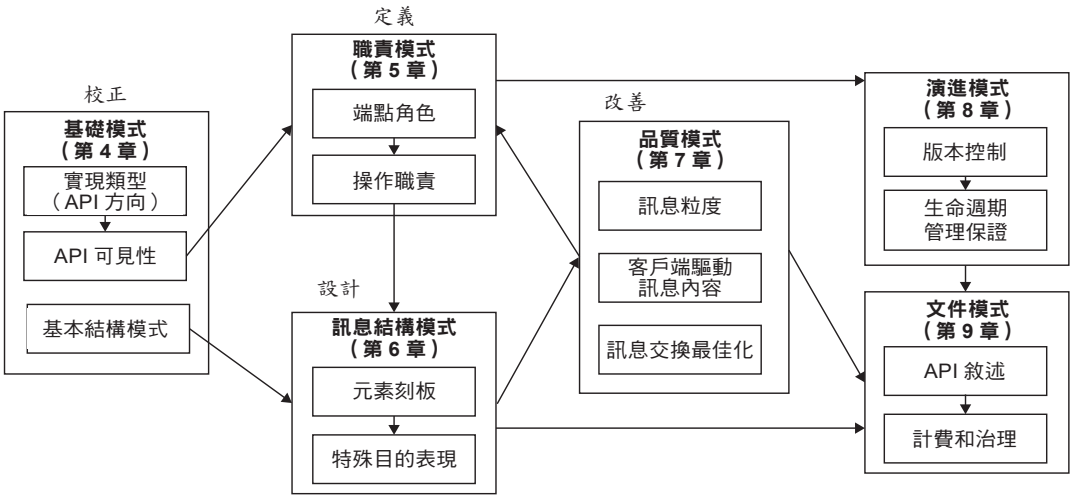

圖 P.2 Über 模式地圖：本書第二部分各章順序

每一個模式可看作是一小篇的獨立文章，通常只有幾頁。這些討論以相同的方式組織：首先介紹套用這個模式的時機及理由，接著解釋模式運作方式，並給出至少一個具體範例；接著討論套用模式的結果，並引導讀者至其他可套用模式。我們把模式的英文名稱設定為「小型大寫字母」（small caps），如：PROCESSING RESOURCE。第 4 章詳細介紹的模式模板來自 EuroPLoP 會議（Harrison 2003），我們把審核時的評論及建議也納入考慮，並稍微重構了一點（感謝 Gregor 與 Peter ！）模式處理架構需求時特別強調品質屬性及衝突，因此，從事 API 設計及演進決策時需要做出取捨。

第三部分為兩個不同領域的真實專案模式應用，分別為電子政府及營造業報價及訂單管理專案，依此得出結論及觀點。

附錄 A〈端點識別與模式選擇指南〉提供一個問題導向的備忘錄，以作為另一個開始選項。其中也討論模式與 RDD（責任驅動設計）、DDD（領域驅動設計）及 ADDR（校正 - 定義 - 設計 - 改善）的關係。**附錄 B〈Lakeside Mutual 案例實現〉**分享更多書中案例的 API 設計成果，**附錄 C〈微服務特定領域語言（MDSL）〉**則提供 MDSL 的運作知識，MDSL 是一個以 `<<Pagination>>` 為修飾、支援模式的微服務規約語言，MDSL 支援對 OpenAPI、gRPC、GraphQL 及其他介面敘述及程式語言的綁定及生成器。

第 3 章

API 決策敘事

API 端點、操作及訊息設計是多方面的，所以並不簡單。彼此衝突的需求隨處可見，需要有所平衡；在有許多解決方案選項可選擇的情況下，也必須做出許多架構決策及實現選擇。一個 API 設計成功的關鍵是做出對的決策，有時開發者不曉得有哪些必要選項，或只知道可用選項的子部分。而且不是所有準則都顯而易見，例如效能及安全性等品質屬性，就會比持續性等其他屬性明顯。

本章將依照主題種類來分辨模式選擇決策，先走過 API 設計迭代，從 API 範圍開始，然後移往端點角色及操作職責的架構決策，也涵蓋品質相關的設計改善決策及 API 演進，說明需要的決策，第二部分模式所涵蓋的最普遍選項，以及已經在實務中看過的模式選擇準則。

序幕：以模式作為決策選項，力量作為決策準則

如同《Continuous Architecture in Practice》（Erder 2021）所提，選擇模式是一個做出架構決策及證明可行的過程，因此，以下敘述會強調 API 設計及演進時需要的架構決策，並討論每一個決策準則和其他設計方案，我們的模式也會提供這些替代方案選項，可見本書第二部分的深入探討。

為了識別需要的決策，將使用以下格式。

決策需要的決策範例

強調哪一個主題？

以下格式中接著呈現合格的模式。

	模式：模式名稱
問題	〔強調哪一個設計問題？〕
解決方案	〔處理問題的可能方式概覽〕

然後總結決策準則，對應第二部分的模式力量（pattern forces），並且給出一些良好的實務建議；但不用逐字遵從，而應該考量特定 API 設計工作的背景。

範例決策結果。呈現第 2 章〈Lakeside Mutual 案例研究〉的決策範例，使用以下架構決策紀錄（ADR）格式：

在〔功能或元件〕的背景中。

想要／面對〔需求或品質目標〕的需求

決定〔選擇選項〕

及忽略的「替代方案」

來達成「好處」

接受「負面結果」

這樣的格式稱為**陳述之因（why-statement）**（Zdun 2013），是一個架構決策紀錄範本，因 Michael Nygard 而盛行（Nygard 2011），這類決策日誌在研究及實務中流傳已久；簡而言之，它可持續追蹤決策成果及在給定背景中的合理性。[1]

一份 ADR 範本的實例可寫成：

在模式決策敘事背景，

面對在範例中描繪選項及準則的需要，

決定注入這類架構決策紀錄，以達到理論和實務的平衡，

接受相關內容會更長，讀者必須從頭閱讀到尾，才能從概念跳到應用。

1 見 https://ozimmer.ch/practices/2020/04/27/ArchitectureDecisionMaking.html。

每一個陳述之因皆設為**楷體**，所以能清楚地和本章概念性內容，即決策點、選項及準則有所區分。陳述之因的「被忽略」部分是可選擇的，且此範例不使用。

本章剩餘部分涵蓋以下決策主題：

- 「基礎 API 決策與模式」介紹 API 可見性、API 整合類型及 API 文件。
- 「API 角色及職責決策」討論端點的架構角色，改善資訊持有者的角色及定義操作職責。
- 「選擇訊息表現模式」涵蓋表現元素的扁平及巢狀結構之間的選擇及元素刻板。
- 「治理 API 品質」是多方面的：API 客戶端的識別和驗證、對 API 使用量的計量和收費、防止客戶端過度使用 API、明確的品質目標與罰則規範、錯誤溝通以及外部上下文表示。
- 「API 品質改善決策」處理分頁，避免不必要的資料傳輸其他手段，及處理訊息中的參照資料。
- 「API 演進決策」有兩部分：版本及相容性管理和啟用及停用策略。

另有兩個小節，包括「Lakeside Mutual 案例的職責及結構模式」，以及「Lakeside Mutual 案例的品質及演進模式」。

基礎 API 決策與模式

第 1 章〈應用程式介面（API）基礎〉介紹 API 是暴露計算或資訊管理服務的介面，同時解耦底層服務提供者的實現與 API 客戶端。本節將介紹基礎的架構設計決策，及作為決策選項的模式，詳細說明 API 提供者的服務實現與 API 客戶端之間的關係。本節的模式有管理及組織相關主題，而且將明顯影響重要的技術考量。

本節決策回答以下問題：

- API 應該從哪取得，或是 API 的**可見性**如何？
- API 應該支援何種**整合類型**？
- API 應該**文件化**嗎？如果是的話，應該怎麼寫成文件？

圖 3.1 顯示這些決策關係。

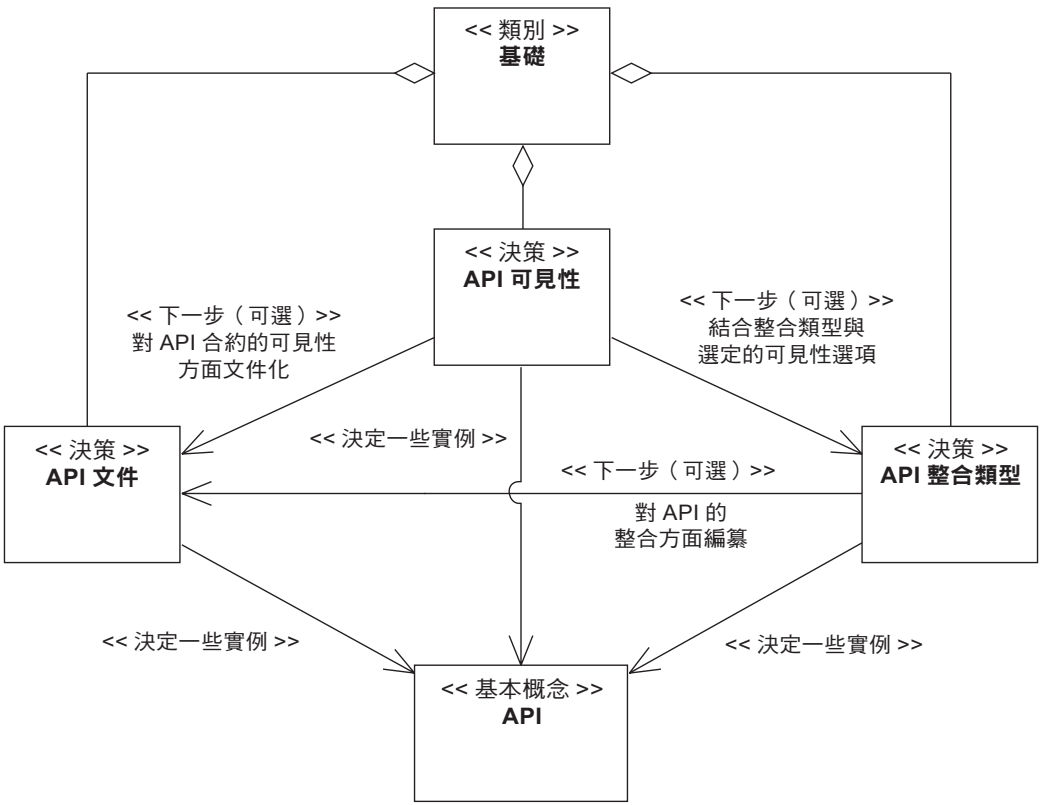

圖 3.1 基礎類別

這個類別中的第一個決策在於 API 可見性。不同的 API 種類中，預計使用 API 的 API 客戶端可能截然不同，範圍從不同組織及地點的大量 API 客戶端，到單一組織及／或同個軟體系統中的少數幾個知名的 API。

除此之外，必須決定物理層系統的組織如何與 API 關聯，進而導致不同的可能整合類型。負責顯示及控制終端使用者介面的前端，可能在物理上與負責資料處理與儲存的後端分離，這樣的後端可能會拆分並分散到多個系統和／或子系統中，例如服務導向架構。這兩種情況下，前端與後端都可能基於 API 而整合。

最後是關於 API 文件需要的決策。當服務提供者決定暴露一個或多個 API 端點時，客戶端必須能夠找出位置，及呼叫 API 操作的方式。這包括技術上的 API 存取資訊，像是 API 端點位置或訊息表現中的參數，以及操作行為的文件，包括前置及後置條件，及相關的服務品質保證。

API 可見性

你可能想以暴露一或多個端點的遠端 API，來提供應用程式的一部分。在這樣的情境下，每個 API 的早期決策都與其可見性相關。從技術觀點來看，API 可見性決定於部署位置及網路連線，例如網際網路、外部網路、公司內部網路或單一個資料中心；而從組織角度來看，API 客戶端服務的終端使用者，將影響可見度的需求等級。

可見性決策主要不是技術上的決策，而是一個管理或組織上的決策，通常與預算和資金考量有關。有時 API 開發、營運和維護是由單一個專案或產品所資助，當然也有多個組織或組織中多個單位共同資助的情況。

然而，可見性決策對技術方面有重要影響。比較在網際網路上由任意數量且部分未知的 API 客戶端使用的公開 API，與一個少數且穩定數量的其他組織系統及／或子系統使用的解決方案內部 API，公開 API 要承受的工作量可能相當高，且包含數個尖峰用量；而只有少數已知客戶端使用的解決方案內部 API，其工作量通常非常低。因此，這兩種 API 的可見性對效能和可擴充性的需求就會非常不一樣。

要採取的核心決策如下：

> **決策：*API 可見性***
>
> 要從哪裡存取 API：網路、存取受控制的網路例如內部或外部網路，或是只有特定解決方案的資料中心？

圖 3.2 顯示這個決策的三個決策選項，描述為模式。

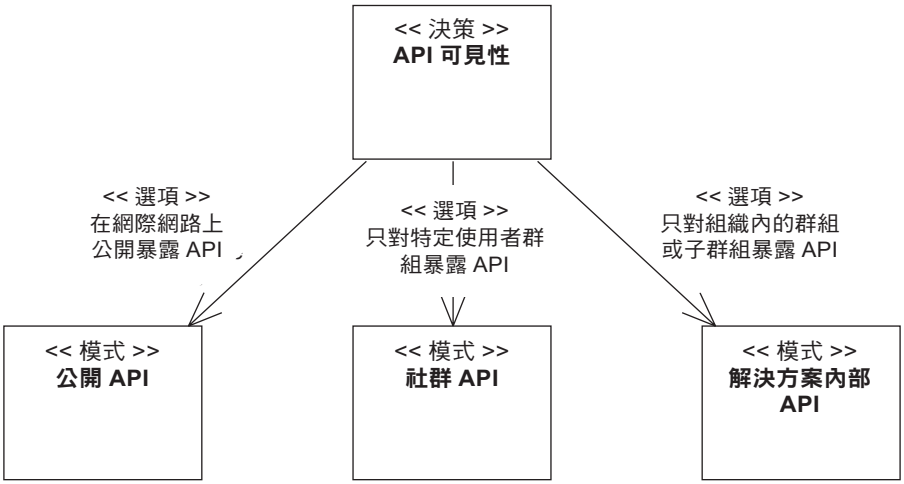

圖 3.2 API 可見性決策

第一個選項是**公開** *API*（*Public API*）模式。

模式：公開 *API*（*Public API*）

問題	如何讓組織外部無限且／或未知數量的 API 客戶端取用 API ？且這些客戶端分散於全球、國際和區域面。
解決方案	在公開網際網路上暴露 API 及詳細的 *API 敘述*（API Description），描述 API 的功能及非功能特性。

對於**公開** *API* 來說，更要考量目標聽眾的規模、位置及多樣性。目標受眾的期望及需求、可能使用的開發和中間件平台，以及其他類似考量，都有助於決定是否以及如何公開提供 API。例如，相較於在伺服器上渲染的動態網站，透過瀏覽器存取 API 的單頁應用程式趨勢，會導致可透過網際網路存取的 API 數量增加。

高可見性的**公開** *API* 經常必須應付持續的高工作負載及／或尖峰用量，這樣會增加後端系統和資料存儲的複雜度，和對高成熟度的要求。API 要承受的可能負載取決於目標受眾的規模，而目標受眾的位置，則決定網際網路存取等級及頻寬需要。

可見性高的 API，可能比可見性低的 API 有更高的安全需求。使用 *API 金鑰*（*API Key*）或驗證協議，通常代表一般**公開** *API* 與開放 API（Open API）變體的不同：一個真正的**開放** API，是沒有 *API 金鑰*或其他驗證方式的公開 API。當然，*API 金鑰*及驗證協議也可用於所有其他決策選項。

還必須考慮 API 開發、營運及維護成本。通常 API 必須有一個用來產生資金的商業模型，對於**公開** *API* 來說，付費訂閱及依呼叫次數付費是常見的選項，可見**計價方案**（*Pricing Plan*）模式；另一個選項是交叉出資，例如透過廣告。而這些考量都必須配合預算，雖然提供第一版 API 的初始開發資金可能很容易，但長期營運、維護和演進的籌資也可能較為困難，尤其是有大量客戶的成功**公開** *API*。

可見性限制更多的**社群** *API*（*Community API*）是替代的決策選項。

模式：社群 *API*（*Community API*）

問題	如何將 API 的可見性和存取限制在封閉的使用者群組之下？這個群組不為單一組織單位工作，而是為多個法律實體，如公司、非營利／非政府組織和政府工作
解決方案	將 API 與其實現資源安全地部署在存取受限的位置，以便只有期望的使用者群組，例如外部網路（extranet）可以存取 API，或只與受限的目標聽眾分享 *API 敘述*。

如同**公開** *API*，**社群** *API* 的開發、營運及維護也一樣需要資金支持。因此預算同樣扮演重要角色，但社群特性及所需的解決方案決定資金獲得方式。例如，在產品使用者社群中，許可證費用可能涵蓋預算，政府及非營利組織可能為特定且受限的使用者群組提供 API 資金，以實現特定社群目標。簡單說，與**解決方案內部** *API*（*Solution-Internal APIs*）的主要差別是，**社群** *API* 通常沒有一個專案或產品預算為 API 支付費用，API 出資者的利益百百種。

通常在公司情境中可觀察到更多模式變體。企業 API（Enterprise API）是只能在公司內部網路使用的 API，產品 API 與購買的軟體或開源軟體一起提供。最後，透過雲端供應商和在雲端環境託管的應用程式服務暴露的服務 API（Service API），若其存取是受限且受保護的，也算是**社群** *API* 的變體。

目標聽眾大小、地點和技術偏好也有影響，這通常和預算考量有關，社群成員可能為 API 支付費用，這些社群特性可能比個別團隊或公開特性更具挑戰性和多樣性。與**公開** *API* 相比，API 的開發組織通常可以輕易地設立標準，因為使用者在政治上相對較弱勢，而在有限界的社群中，利害關係者的考量通常多樣且要求較高，例如，應用程式負責人、DevOps 職員、IT 安全官等角色的考量可能不同且彼此衝突。這些考量也可能使 API 生命週期管理有更高的要求，舉例來說，**社群** *API* 的付費顧客就可能對 API 版本維持運作有相當高的要求。

最後，可見性最受限的決策選項，是**解決方案內部** *API*。

模式：解決方案內部 *API*（*Solution-Internal API*）

問題	如何把 API 存取及使用限制在應用程式內？例如，在同個或另一個邏輯層及／或物理層中的元件。
解決方案	將應用程式依邏輯分解成元件，讓這些元件暴露本地或遠端 API。這些 API 只提供給系統內部通訊夥伴，例如應用程式後端中的其他服務。

就前面兩個模式而言，必須考慮**解決方案內部** *API* 開發、營運及維護的資金預算。相比其他兩個 API 可見性類型，即較多的暴露，**解決方案內部** *API* 通常問題較少，因為單一專案或產品的預算會包含 API 的成本，專案因此也可以決定生命週期考量，及支援的目標聽眾大小、地點和技術偏好，這些考量的重要性自然取決於專案目標。例如，開立線上購物發票的 API 開發，可以預期 API 開發團隊已經知道產品及出帳需求；而這些需求會隨時間改變。如果團隊推出新的 API 版本，相關改變也會通知依賴團隊，讓這些團隊可以在同一個購物應用程式從事開發工作。

之前提到的其他技術考量也有類似特色，它的工作量通常比**公開** *API* 清楚，除非**解決方案內部** *API* 接收來自**公開** *API* 的呼叫。例如在出帳的情境中，如果公司所有產品本身都透過**公開** *API* 提供，那出帳的**解決方案內部** *API* 就必須應付來自這些公開 API 的負載。同樣地，後端系統及資料存儲的複雜性與成熟度，以及安全需求也必須滿足解決方案內部的需求，並且能遵循組織中使用的最佳實踐（best practice）。

注意，有時**解決方案內部** *API* 會演進成為**社群** *API*，或甚至是**公開** *API*。這種轉變不應該以範疇蔓延的形式發生，而應該是有意識地決策和計畫；發生這樣的轉變時，有些 API 設計決策，例如 API 安全性決策，就有可能必須重新審視。