前言

微服務（microservice）是一種分散式系統的做法，倡導使用可以獨立地變更、部署和發布的細微化服務。對於正在發展更鬆散的耦合系統之組織來說，微服務能透過自主團隊來提供使用者功能，其效果是非常好的。除此之外，微服務還提供了很多建置系統的選項，給我們很大的彈性，能確保系統可以修改以滿足使用者的需求。

然而，微服務並非沒有明顯的缺點。作為一種分散式系統，微服務非常複雜，即便是有經驗的開發者，也可能對其中大部分的內容感到陌生。

全球專業人士的實務經驗，加上新技術的出現，對於微服務的運用產生了極為深遠的影響。本書結合這些觀點與具體的實務範例，幫助你瞭解微服務是否適合你。

誰應該閱讀這本書

這本書的討論範圍甚廣，因為微服務架構的意涵也是相當寬闊，因此，本書應該會吸引各路人馬的關注，包括設計、開發、部署、測試、和系統維護等相關人員。那些已經踏上微服務架構之征途的人，不管是要新建應用程式，或者是要分解既有的單體式系統，將會從本書中找到許多有用的實務建議。另外，對於想要瞭解這一切是怎麼回事的人，這本書也是非常有幫助的，藉此可以自行判斷微服務是否適合你。

為什麼撰寫這本書

在某程度上，寫這本書是因為我想確保第一版書中的資訊仍是最新、準確和有用的。我寫第一版是因為我有非常有趣的想法想分享，很幸運的是，我在一個有時間和支援的地方完成了第一版，並且因為我沒有為任何大型技術供應商工作，我可以從一個相當公正

的角度來寫作。我不是在推銷解決方案，也希望不是在推銷微服務，我只是發現這些想法很吸引人，而我也樂於解讀微服務的概念並找到能廣泛分享它的方法。

撰寫第二版主要有兩個原因。首先，我覺得這次我可以做得更好，我學到了更多的東西，也希望能寫得更好。此外，在推廣微服務觀點進入主流的事上，我也貢獻了一點綿薄之力，因此我有責任確保這些觀點是以合理、平衡的方式介紹。對許多人而言，微服務已經成為預設的架構選項，但我認為這是很難證明的事，我想要藉此機會分享我的觀點。

這本書並非**支援**，也不是**反對**微服務。我只是想確保我有正確地探討了這些想法的背景，並分享可能導致的問題。

第一版後發生了哪些改變？

我花了一年左右的時間完成了《**建構微服務**》的第一版，從 2014 年初著手寫作，到 2015 年 2 月出版。這是微服務的早期階段，至少在廣大行業對這個名詞的認識上是如此。從那時起，微服務以一種我沒有預想到的方式成為主流，隨著這種成長，有許多經驗可以借鏡，也還有更多技術可以探索。

當我在撰寫第一版與更多團隊合作時，我開始精煉了我對於微服務概念的一些想法。在某些情況下，這意味著一些我原本不注重的想法（如資訊隱藏），開始變得更清楚，成為需要更加強調的基礎概念。在其他領域，新技術為我們的系統提供了新的解決方案也帶來了新的問題。看到這麼多人對 Kubernetes 寄予厚望，希望能解決他們在微服務架構上的所有問題，這著實讓我停下來思考。

此外，第一版的《**建構微服務**》不僅提供對微服務的解釋，還廣泛地介紹這種架構方法是如何改變軟體開發的各方面。因此，當我更深入研究安全性和韌性方面的問題時，我想回到過去，將這些對現代軟體開發越來越重要的議題有更展開的論述。

因此，在第二版中，我花了更多時間以明確的範例更清楚地解釋這些觀點。每一章節、每一句話都被重新審視。在具體的文章方面，第一版的內容雖保留得不多，但想法都還在。我努力使自己的觀點更加清晰，同時也認知到解決一個問題往往有多種方法。所以對於行程內溝通（inter-process communication）的討論展開成三個章節。我也花了許多時間研究像是容器、Kubernetes、無伺服器等技術可能的影響，因此現在有獨立的章節討論建置和部署。

原本，我希望能完成一本和第一版差不多厚的書，同時能找到方法裝得下更多的想法，然而正如你所見的，我沒能達成這個目標——這版本變得更厚了！但我想我已經成功地清楚表達我的想法了。

本書簡介

這本書主要根據主題（topic）組織而成，本書的結構和內容可以從頭讀到尾，但當然你可能會想要直接跳到最感興趣的特定主題。如果你決定直接進入某個章節，書末的詞彙表能解釋新的或不熟悉的術語，會對你很有幫助。關於術語的問題，我在書中交替地使用微服務和服務這兩個詞，除非我明確地指明，不然你可將這二者視為同一件事。我也在參考書目中彙整了本書的一些關鍵建議，因此如果你**真的**只想跳到最後面，記住，如果你這樣做，你將會錯過**很多**細節！

本書主要分為三個獨立的部分：基礎、實作和人。讓我們來看看各個部分的內容：

第一部分　基礎

在第一部分中，我詳細介紹一些微服務的核心觀念。

第一章　什麼是微服務？

這是對於微服務一般性的介紹，其中會簡要地討論一些在後續章節中會展開的主題。

第二章　如何對微服務塑模？

本章探討了諸如資訊隱藏、耦合、內聚和領域驅動設計（domain-driven design）等概念，幫助你為微服務找到正確邊界的重要性。

第三章　拆分單體

本章提供了一些指引，說明如何採用既有的單體式應用並將其拆解為微服務。

第四章　微服務的溝通風格

本部分的最後一章討論了不同類型的微服務溝通，包含了異步（asynchronous）與同步（synchronous）呼叫，以及請求 / 回應（request-response）和基於事件（event-driven）的協作風格。

第二部分　實作

從高階層的概念轉到實作的細節，在這一部分，我們將著眼於可以幫助你充分利用微服務的技巧和技術。

第五章　實作微服務溝通

在本章，我們將更深入地探討用於實現微服務間溝通的具體技術。

第六章　工作流程

本章比較了 saga 和分散式交易，並討論它們在對包含多個微服務的業務流程進行塑模時的用處。

第七章　建置

本章解釋從微服務到儲存庫的對映及其建置。

第八章　部署

在本章中你會看到對於部署微服務之眾多選項的討論，包含探討了容器、Kubernetes 和 FaaS（函式即服務）。

第九章　測試

在此，我們將討論測試微服務時所面臨的挑戰，包括由端到端測試所引起的問題，以及消費者驅動的契約測試（consumer-driven contract）、和生產中測試能如何提供幫助。

第十章　從監控到可觀察性

本章涵蓋了從關注靜態監控活動，到廣泛地思考如何提高微服務架構可觀察性的轉變，以及一些關於工具的具體建議。

第十一章　資訊安全

微服務架構增加了被攻擊的範圍，但也給了我們更多深度防禦的機會；在本章中，我們將探討這種平衡。

第十二章　彈性

本章廣泛地探討什麼是彈性，以及微服務在改善應用的彈性方面可以發揮的作用。

第十三章　擴展

在本章中，我概述了關於擴展的四軸，並展示它們是如何組合起來以擴展微服務的架構。

第三部分　人

如果沒有人和組織的支援，想法和技術就毫無意義。

第十四章　使用者介面

從脫離專門的前端團隊，到使用 BFF 和 GraphQL，本章探討了微服務和使用者介面是如何合作的。

第十五章　組織結構

倒數第二章重點介紹 steam-aligned 團隊和 enabling 團隊如何在微服務架構環境中工作。

第十六章　進化的架構師

微服務架構不是一成不變的，所以你對於系統架構的看法可能需要改變，這是本章將深入探討的主題。

本書編排慣例

本書使用的字型、字體慣例，如下所示：

斜體字（*Italic*）

用來代表新的術語、URL、電子郵件地址、檔案名稱及副檔名。對於初次提到或重要的詞彙，中文以楷體字呈現，其對應的英文則以斜體表示。

定寬字（`Constant width`）

用來列舉程式碼，以及在文章中表示程式的元素，例如變數、函式、資料庫、資料型別、環境變數、程式語句和關鍵字等。

這個圖示代表一個小技巧或建議。

第一章

什麼是微服務？

自本書第一版完成以來，微服務已經成為一種越來越流行的軟體架構選項。我不能說這隨後而來的流行是我的功勞，但使用微服務架構的風潮意味著，雖然我之前提出的許多想法現在都經過了嘗試和測試，但在早期的作法已經不受歡迎的同時，新的想法也出現了。因此，是時候再次提煉微服務架構的精髓，同時強調能使微服務發揮作用的核心概念。

整體而言，本書旨在廣泛地介紹微服務架構對於軟體交付之各方面的影響。首先，本章將介紹微服務背後的核心思想，就是把我們帶到這裡的既有技術，以及這些架構之所以被廣為使用的一些原因。

微服務概覽

微服務（*Microservices*）是圍繞業務領域塑模而成、可獨立發布的服務。一個服務將功能封裝起來，並使其可透過網路來訪問其他服務（你可以從這些建置中，建立一個更複雜的系統）。一個微服務可能代表庫存、另一個代表訂單管理、還有一個代表出貨管理，但它們組合一起可能構成一個完整的電子商務系統。微服務是一種架構選擇，專注於為你提供多種選擇來解決可能面臨的問題。

儘管微服務對於服務邊界的劃分有自己的定義，而且獨立部署性是其關鍵，微服務是一種服務導向（service-oriented）的架構，而技術中立（technology agnostic）是它其中一個優點。

從外面看，單個微服務看起來是一個黑盒子，它在一個或多個網路端點（例如一個佇列（queue）或一個 REST API，如圖 1-1 所示）上透過任何最適合的通訊協定來託管業務功能。消費者，無論是其他的微服務還是其他類型的程式，都透過這些網路端點來訪問

這些功能。內部的實作細節（如服務所使用的撰寫技術或資料儲存的方式）對外界是完全隱藏的，這表示微服務架構在大部分的情況下避免使用共享的資料庫；相反地，每個微服務在必要時都會封裝自己的資料庫。

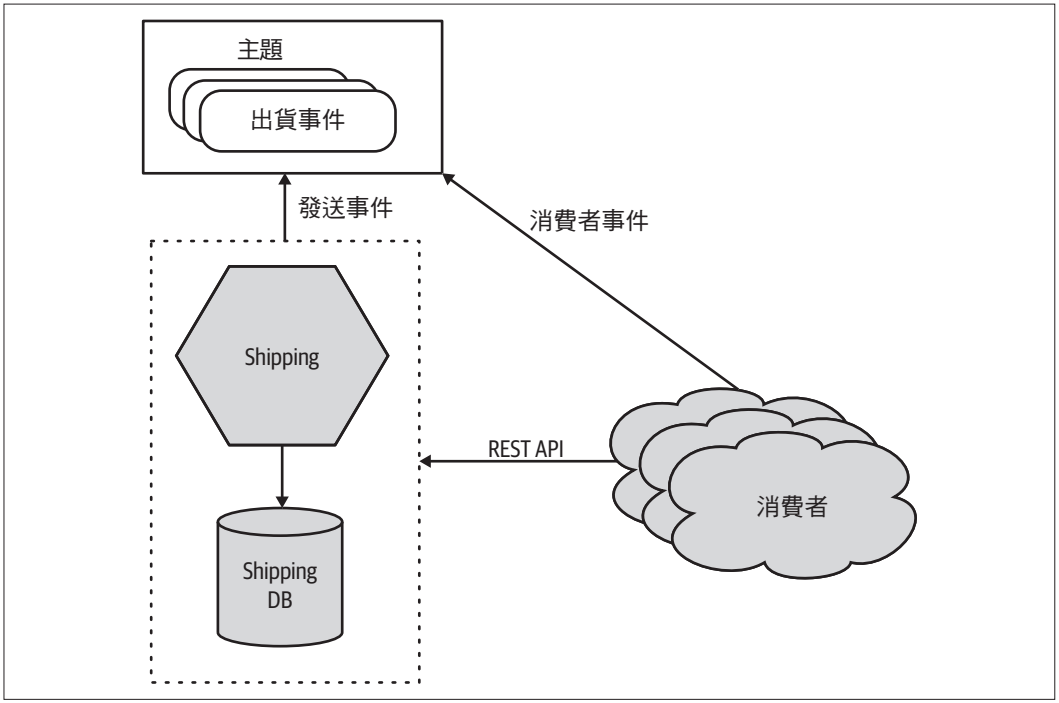

圖 1-1　一個透過 REST API 和主題展示其功能的微服務

微服務包含了資訊隱藏的概念[1]。資訊隱藏（*Information hiding*）意思是盡可能地將越多的資訊隱藏在一個元件之中，並且透過外部介面暴露在外的資訊越少越好。這使我們能明確區分出哪些是易於改變，哪些是較難改變的。只要微服務所暴露的網路介面不以無法向後相容的方式作變更，就可以自由改變對外部隱藏的實作部分。而在微服務邊界內的變化（如圖 1-1 所示）不應該影響到上游的消費者，從而實現功能的獨立發布。這對於能使我們的微服務可以獨立工作並按需求發布是非常重要的。擁有清楚、穩定，且不會隨著內部的實作變化而改變的服務邊界，能使系統具有更鬆散的耦合和更強的內聚性。

1　此概念由 David Parnas 首次概述於刊登在 *Information Processing: Proceedings of the IFIP Congress 1971* 的論文《Information Distribution Aspects of Design Methodology》中（*https://oreil.ly/rDPWA*）（Amsterdam: North-Holland，1972），1:339–44。

當我們在談論隱藏內部實作細節的時候，如果沒有提到最早由 Alistair Cockburn 提出的**六角形架構**（*Hexagonal Architecture*）模式的話，那我就失職了[2]。這個模式描述了將內部實作與外部介面分開的重要性，其理念是希望能透過不同類型的介面與相同的功能互動。我把我的微服務畫成六邊形，部分原因是為了將它們與「一般」服務區分開來，同時也是為了向這先前技術表達敬意。

服務導向的架構和微服務是不同的東西嗎？

服務導向架構（*Service-oriented architecture*，SOA）是一種設計方法，其中多個服務協同合作以提供一組功能。這裡的**服務**通常是指一個完全獨立的作業系統行程（process），這些服務之間通常是透過跨網路呼叫進行溝通，而不是用行程邊界之內的方法呼叫（method call）。

SOA 的出現是為了應對大型單體式應用程式的挑戰，目標是促進軟體的可重利用性（reusability）；例如，兩個以上的終端使用者應用程式能夠使用相同的服務。SOA 的目標是讓我們更容易維護或重寫軟體，理論上，只要服務的語意（semantics）沒有太大的變動，我們可以悄悄地使用一個服務替換另一個服務。

SOA 在根本上是非常明智的想法，然而，儘管經過許多努力，但對 SOA 的**合適**實作上還是缺乏良好的共識。在我看來，許多業界人士都未能全盤思考這個問題，並且提出令人折服的解決方案來取代業內各個供應商的一家之言。

事實上，SOA 面臨的諸多問題包括通訊協定（如 SOAP）、供應商中間件（vendor middleware）、缺乏服務細粒度（granularity）的指導方針、或者缺乏要挑選何處進行系統分割的正確指導。憤世嫉俗的人可能認為供應商們吸納（甚至驅使）SOA 作為一種推廣業務或促銷產品的機制，而這些完全相同的產品最終卻損害了 SOA 的目標。

我見過很多 SOA 的範例，在這些例子中的團隊正在努力使服務變得更小，但他們仍然把所有的東西都耦合到資料庫上，並且不得不把所有東西都部署在一起。這算是服務導向架構，但這不是微服務。

2 Alistair Cockburn 於 2005 年 1 月 4 日發表的《Hexagonal Architecture》，*https://oreil.ly/NfvTP*。

微服務方法衍生自現實世界的實際運用，讓我們更清楚地理解有助於妥善建置 SOA 的系統與架構，因此，你應該把微服務看作是一種 SOA 的特定解法，就像 XP 或 Scrum 被視為敏捷軟體開發的特定做法那樣。

微服務的核心概念

在探索微服務時，必須了解一些核心想法。鑑於某些方面經常被忽視，進一步探索這些概念是非常重要的，以幫助確保你了解到底是什麼讓微服務發揮作用。

可獨立部署

可獨立部署（*Independent deployability*）是指我們可以對一個微服務進行改變、部署，並向我們的使用者發布這個改變，而不需要部署任何其他的微服務。更重要的是，這不僅是我們可以這樣做而已，這**事實上**是我們在系統中管理部署的方式，是一種預設的發布準則。雖然這是一個簡單的想法，但在執行上卻很複雜。

如果你只從本書或一般的微服務概念中得到一件事，那必須是：確保你接受微服務可獨立部署的概念。養成將單個微服務的變更部署和發布上線，而不需要部署其他東西的習慣。由此，許多美好的事物將接踵而來。

為了確保可獨立部署，我們需要確保微服務是**鬆散耦合**的：我們必須能夠在不改動任何其他服務的條件下，變更一項服務；這意味著我們需要服務之間有明確、定義清楚和穩定的契約。有一些實作上的選擇使得這變得困難，例如資料庫的共享尤其成問題。

可獨立部署本身顯然非常有價值，然而你還有很多事情需要做對，而這些事也帶來其好處。因此，你還可以將關注獨立部署視為一種強制功能——透過關注這個結果，你將獲得許多輔助效益。對於具有穩定介面的鬆散耦合服務之預期，幫助我們思考如何先找到微服務的邊界。

圍繞著業務領域塑模

領域驅動設計之類的技術可以讓你建置程式碼，以更好地代表軟體程式所運行的現實領域[3]。有了微服務架構，我們可以用相同的想法來定義我們的服務邊界，透過圍繞著業務領域對服務進行塑模，我們可以更輕鬆地推出新功能，並能以不同的方式重新組合微服務，提供新功能給使用者。

推出一項需要變更多個微服務的功能，其成本是很高的，你需要協調每個服務之間的工作（可能會跨不同的團隊），並且要仔細地管理這些新版本部署的順序。這比起在單個服務（或在單體式應用）中作同樣的改變需要更多的工作，因此可以得出，我們希望能找到方法盡可能地減少跨服務變更的頻率。

我經常看到分層架構，如圖 1-2 中的三層架構所示，架構中的每一層代表一個不同的服務邊界，而每個服務邊界是基於其相關的技術功能。在這個範例中，如果我只需要對表示層（presentation layer）進行修改，那會相當有效率；然而，經驗告訴我們，在這類型的架構中，功能的變化通常會跨多個分層——需要在表示層、應用（application）層和資料（data）層中進行變更。如果架構比圖 1-2 中的簡單例子還更多層（通常每一層會被分成更多層），則這個問題會更嚴重。

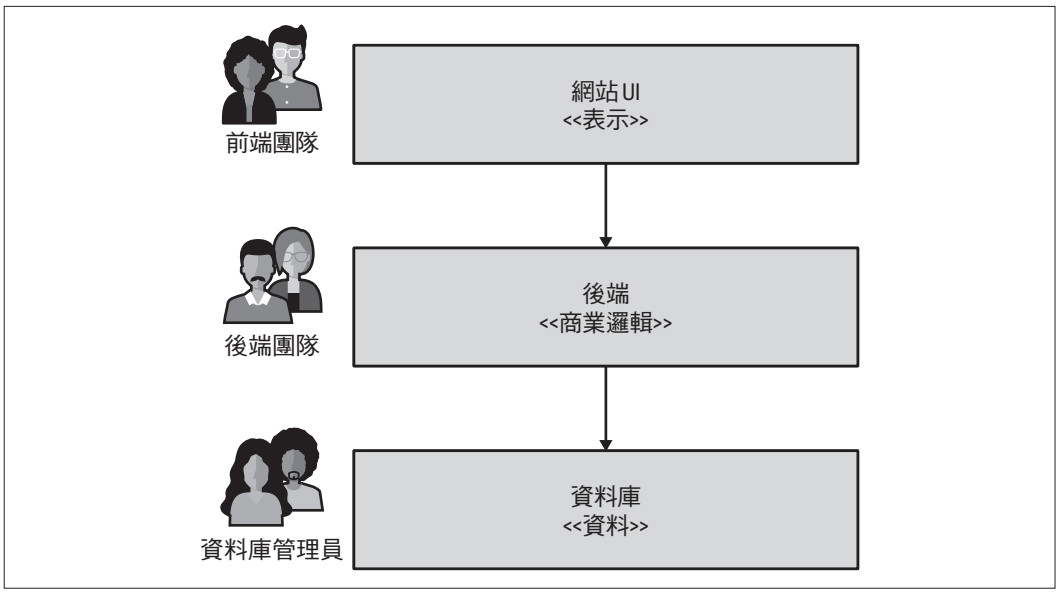

圖 1-2　傳統的三層架構

3　有關領域驅動設計的深入介紹，請參閱 Eric Evans 寫的《*Domain-Driven Design*》（Addison-Wesley），或者是簡潔的概述，請參閱 Vaughn Vernon 寫的《*Domain-Driven Design Distilled*》（Addison-Wesley）。

透過將我們的服務作為業務功能端到端的各部分，我們能確保我們的架構在業務功能的變更上盡可能地有效率。可以說，對於微服務，我們決定優先考慮業務功能的高內聚性，而不是技術功能的高內聚性。

在本章之後的內容，我們將會回到領域驅動設計的互動作用，以及它是如何與組織設計互相影響。

擁有各自的狀態

我看到人們最難接受的一件事，就是微服務應該要避免使用共享資料庫的想法；如果一個微服務想要訪問另一個微服務的資料，它應該要向第二個微服務詢問資料。微服務能夠決定什麼可以共享、什麼要隱藏，這使我們能夠清楚地將可以自由變更的功能（我們內部的實作）、與我們不會經常變更的功能（消費者使用的外部契約）區分開來。

如果我們想要實現可獨立部署性，我們需要限制微服務中向後不相容的變更，如果我們破壞與上游消費者的相容性，我們也會迫使他們作出改變。在內部實作和外部契約中間，有清楚輪廓的微服務可以幫助減少對向後不相容變更的需求。

在微服務中隱藏內部狀態是類似於物件導向（object-oriented，OO）程式的封裝作法；在 OO 系統中資料的封裝是一種將資訊隱藏在動作中的例子。

除非真的必要，否則不要共享資料庫；而且即便如此，也要盡可能避免這件事。在我看來，如果你想要實現可獨立部署性，共享資料庫是一件最糟糕的事情。

正如在前一節所討論的，我們希望把我們的服務作為業務功能端到端的各部分，在合適的時機點，將使用者介面、業務邏輯和資料封裝起來。這是因為我們想要減少改變業務相關功能所需要的工作量，以這種方式封裝資料和行為，使我們的業務功能具有高度的內聚性，並且透過隱藏服務背後的資料庫，還能確保我們減少耦合。我們將會在第 2 章再提到耦合和內聚。

大小

「一個微服務應該要多大？」是我最常聽到的問題之一。考慮到「微」這個形容詞在其中，這個問題並不奇怪。然而，當你了解微服務作為一種架構是如何運作的，大小這個概念其實是最不有趣的部分。

你是如何測量大小的？是透過計算程式碼有幾行嗎？這對我而言沒有多大的意義。有些東西可能需要 25 行 Java 程式碼，但可以用 10 行 Clojure 完成，這不是說 Clojure 就比 Java 好或差，只是有些程式語言更具有表現力而已。

Thoughtworks 的技術總監 James Lewis 曾說：「一個微服務應該要和我的腦袋一樣大」。這乍看之下，似乎沒什麼用，畢竟我們不知道 James 的頭到底有多大，但這句話背後的基本原理是，微服務應該保持在易於理解的大小。當然，這難在每個人的理解能力並不相同，因此你需要自己判斷什麼大小最合適自己。比起其他團隊，一個有經驗的團隊可能可以更好的管理更大的程式碼基礎（codebase），所以或許最好將 James 的這番話，解讀為「微服務應該要和**你的**腦袋一樣大」。

《*Microservice Patterns*》（Manning Publications）的作者 Chris Richardson 曾經說過：「微服務的目標是擁有『越小越好』的介面」，這是我認為最接近微服務「大小」的意義，也再次符合資訊隱藏的概念，但它確實代表了嘗試在「微服務」一詞中尋找起初並不存在的含義，至少在當初這個術語第一次被用來定義這些架構時，重點並不是特別關注於介面的大小。

大小的概念終究因人而異。和在一系統工作了 15 年的人談話，他們會覺得他們那擁有 100,000 行程式碼的系統是很好理解的，但向剛接觸專案的新人詢問意見，他們卻會覺得那系統太大了；同樣地，詢問一家才剛開始轉型，可能只有少於 10 個微服務的公司，和另一間規模相同但已經將微服務作為常態，已有數百個微服務的公司，你將會得到完全不同的答案。

我勸大家不要那麼擔心大小，在剛開始，更重要的是要專注於兩個關鍵的事情。首先，你可以處理多少個微服務？隨著你有越來越多服務，你系統的複雜度會增加，並且你會需要學習新的技能（或許還需要採用新技術）來因應這種情況。轉型成微服務會帶來新的複雜度，以及隨之而來的所有挑戰，正因為這個原因，我強烈主張要漸進式的遷移到微服務架構。第二，要如何定義邊界以充分利用微服務，而不會使一切變成可怕的耦合混亂？當你開始了微服務之旅時，需要更注重這些主題。

彈性

James Lewis 另外說到：「微服務為你買到了選擇權」，這話說得很有意思，微服務是用買的、是有成本的，你必須決定這個成本是否值得你想要的選擇，由此產生了組織、技術、規模、穩定性等方面的彈性，可能是滿吸引人的。

我們不曉得未來會如何，所以會想要有一個架構在理論上能幫助我們解決未來可能面臨的任何問題。在保持選擇的開放性和承擔這樣的架構成本之間找到一個平衡，可能是一門藝術。

想像一下，採用微服務不像是打開一個開關，而更像是轉動一個轉盤；當你打開轉盤，若你有更多的微服務，你就增加了彈性，但也可能會增加痛點。這是我強烈主張逐步採用微服務的另一個原因，透過逐步增加轉盤，你可以更好地隨時評估你的影響，並且能在必要時停止。

架構與組織的調校

一間線上銷售 CD 的電商公司 MusicCorp，使用簡單的三層架構如圖 1-2 所示。我們決定強迫 MusicCorp 搬到 21 世紀，並且我們正在評估既有的系統架構。我們有一個網頁 UI、一個單體式後端的業務邏輯層，及傳統資料庫的資料倉儲。這些分層通常是由不同團隊所擁有。我們將在整本書中來看 MusicCorp 面臨的考驗和磨難。

我們想對我們的功能做一個簡單的更新：讓我們的客戶來指定他們最喜歡的音樂類型。這個更新需求需要我們改變 UI 來呈現曲風（genre）選項的 UI、改變後端服務讓曲風能呈現在 UI 上並且能變更值，以及改變資料庫能接受此變更。這些變更需要由每個團隊管理並按照正確的順序進行部署，如圖 1-3 所示。

現在這種架構還不錯。所有架構最終都會圍繞著一組目標進行最佳化。三層架構之所以如此普遍，部分原因在於它具有普遍性——每個人都曾聽過它。因此，傾向選擇一種可能在某處見過的常見架構，往往是我們不斷看到這種模式的原因之一。但我認為我們一再看到這種架構的最大原因是，它是基於我們團隊是如何組織的。

著名的康威（Conway）定律說到：

> 負責設計系統的組織所產生的「設計」，將無可避免地複製該組織的「溝通結構」。
>
> —Melvin Conway，《How Do Committees Invent?》（*https://oreil.ly/NhE86*）

三層架構是該定律的一個很好的例子。在過去，IT 組織人員進行分組的主要方式是根據他們的核心能力：資料庫管理員與其他資料庫管理員組成一個團隊；Java 開發人員與其他 Java 開發人員組成一個團隊；而前端開發人員（他們會 JavaScript 和原生行動裝置應用程式開發等新奇的事物）在另一個團隊中。我們根據人員的核心能力來分組，因此我們創建可以與這些團隊保持一致的 IT 資產。

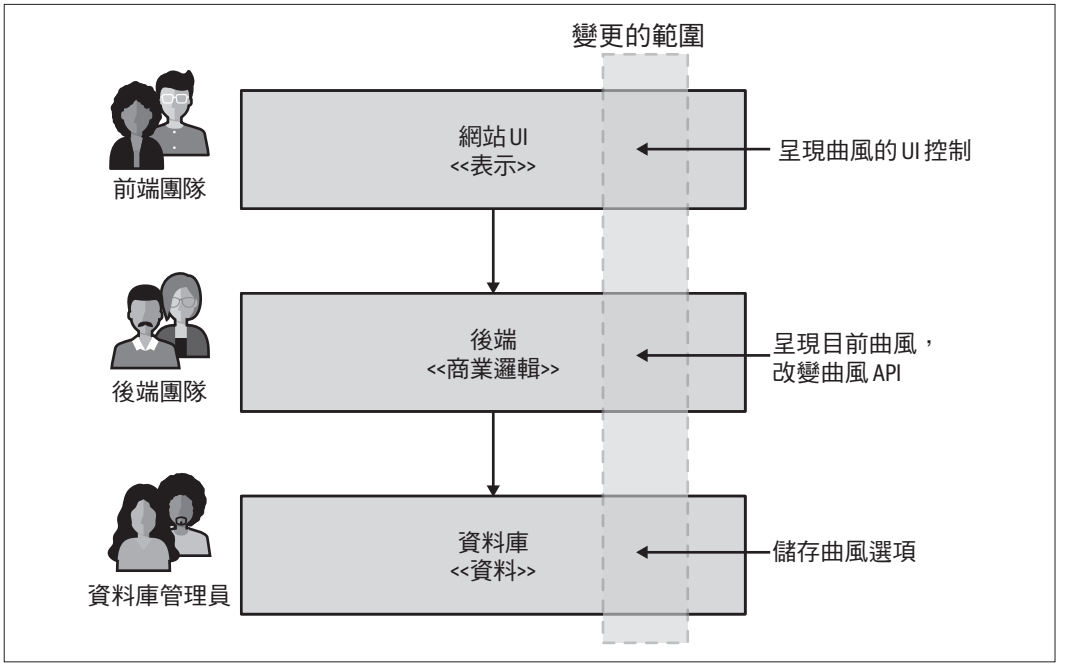

圖 1-3　在所有分層上進行變更，涉及的問題更複雜

這解釋了為什麼這種架構如此普遍。它不算太差；只是它是圍繞著一組力量進行了最佳化，這力量就是傳統上我們如何圍繞熟悉程度對人進行分組。然而，這力量已經發生了變化，我們對軟體的預期已經改變。我們現在將人員分組到多技能團隊中，以減少交接和孤立的穀倉（silo）。我們希望能在任何時候都更快地發布軟體。這促使我們對組織團隊的方式做出不同的選擇，以便我們能以打破系統的方式來組織團隊。

我們被要求對系統進行的變更大部分都與業務功能的變化有關。但是在圖 1-3 中，我們的業務功能實際上分佈在每一層中，如此便增加了功能變更跨越分層的可能性。這是一種「相關技術內聚性高，但業務功能內聚性低」的架構。如果我們想更容易地變更，我們需要改變我們程式碼分組的方式，選擇業務功能的內聚而非技術的內聚。每個服務最終可能包含或可能不包含這三個分層，但這是本地服務實作的問題。

讓我們將其與可能的替代架構比較，如圖 1-4 所示。我們不是採用水平的分層架構和組織，而是沿著垂直的業務線來拆解我們的組織和架構。在此，我們看到一個專門的團隊負責對使用者資料的各個方面進行變更，從而確保變更範圍僅限於一個團隊。

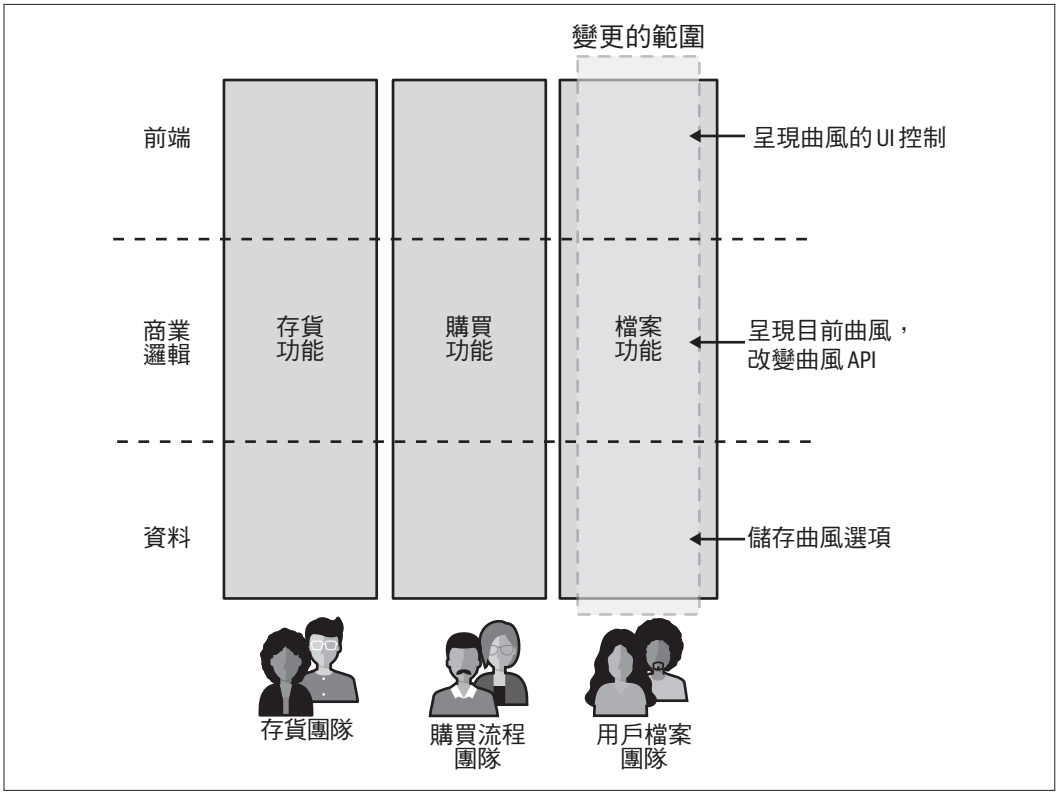

圖 1-4　UI 被拆分並歸一個團隊所有，該團隊還管理支援 UI 的伺服器端功能

這樣的實作可以透過一個使用者資料團隊所擁有的微服務來實現，該微服務呈現一種 UI 可允許使用者更新他們的資訊，使用者的狀態也儲存在此微服務中。最喜歡的曲風之選擇與特定的使用者有關聯，因此這種變更是更加在地化。在圖 1-5 中，我們還顯示了從 `Catalog` 微服務中獲取的可用曲風列表，這些曲風可能已經存在。我們還看到一個新的 `Recommendation` 微服務訪問了我們最喜歡的曲風資訊，這些資訊在後續版本中可以容易地遵循。

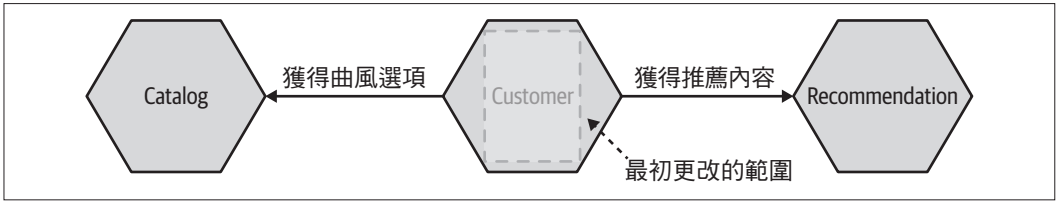

圖 1-5　專用的 `Customer` 微服務可以更輕鬆地記錄使用者最喜歡的曲風

在這種情況下，我們的 `Customer` 微服務封裝了三層中各層的一小部分 —— 它有一些 UI、一些應用程式邏輯和一些資料儲存。業務領域成為驅動我們系統架構的主要力量，希望可以更輕鬆地進行變更，並使我們的團隊更容易與組織內的業務線保持一致。

通常，微服務不會直接提供 UI；但即使有提供，我們也希望與此功能相關的 UI 部分仍由使用者檔案團隊所有，如圖 1-4 所示。團隊擁有端到端針對使用者的功能，這種概念越來越受到關注。《*Team Topologies*》[4] 一書介紹了流式團隊（stream-aligned team）的觀點，體現了這個概念：

> 流式團隊是一個與單一、有價值的工作流一致的團隊…該團隊有權盡可能地快速、安全並獨立地建立、交付顧客和使用者價值，而無需交給其他團隊執行部分的工作。

圖 1-4 中顯示的團隊就是流式團隊，我們將在第 14 章和第 15 章更深入探討這個概念，包括這些類型的組織結構在實作中是如何運作，以及它們如何與微服務保持一致。

關於「假」公司的說明

在整本書中，我們將在不同階段遇到 MusicCorp、FinanceCo、FoodCo、AdvertCo 和 PaymentCo。

FoodCo、AdvertCo 和 PaymentCo 是真實的公司，出於保密原因我變更了名字。此外，在分享有關這些公司的資訊時，我經常省略某些細節以提供更清晰的資訊。現實世界往往是混亂的，不過，我努力只刪除無用的細節，並同時確保仍能保留基本的現實狀況。

另一方面，MusicCorp 是一間虛構的公司，由我合作過的許多組織所組成。我所分享關於 MusicCorp 的故事反映了我所看到的真實事物，但它們並非都發生在同一家公司！

4 Matthew Skelton 和 Manuel Pais 所著的《*Team Topologies*》（land, OR: IT Revolution, 2019）

單體式系統

我們已經談到了微服務，但微服務最常以一種架構方法被討論，它是單體式架構的替代方案。為了更清楚地區分微服務架構，並幫助你更好地了解微服務是否值得考慮，我還是應該解釋一下我所說的**單體式系統**（*monolith*）到底是什麼。

在整本書中當我談論到單體時，主要指的是一個部署單位。當系統中的所有功能必須一起部署時，我認為它是一個單體。多種架構可以說都符合這個定義，但我將討論最常看到的幾種：單行程（single-process）單體、模組（modular）單體和分散式（distributed）單體。

單行程單體

討論到單體式應用時，最常見的例子是一個系統，其中所有程式碼都部署為**單一行程**，如圖 1-6 所示。出於強健性或擴展性的原因，你可能有多個此流程的實例（instance），但基本上所有程式碼都打包在一個行程中。實際上，這些單行程系統本身可以是簡單的分散式系統，因為它們最終幾乎總是從資料庫讀取資料，或將資料儲存到資料庫中，或者將資訊呈現於網頁或行動應用程式中。

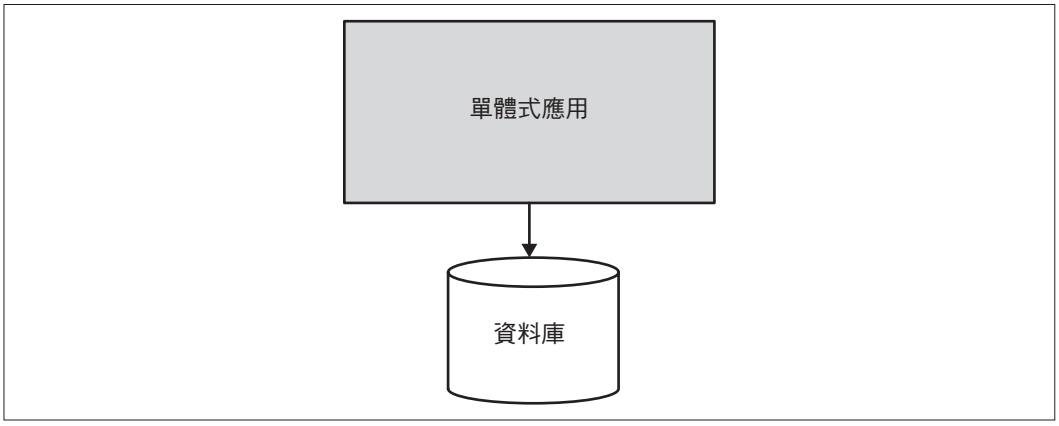

圖 1-6　在一單行程單體式應用中，所有的程式碼會被封裝在單個行程裡

雖然這符合大多數人對經典單體式應用的理解，但我遇到的大部分的系統都比這個更複雜，你可能有兩個或多個彼此緊密耦合的單體式應用，其中可能混合了一些供應商軟體。

經典的單行程單體部署對許多組織而言都有意義。Ruby on Rails 的發明者 David Heinemeier Hansson 有效證明了這種架構對於較小的組織是有意義的[5]。然而，當組織在成長，單體式應用可能會隨之增長，這將我們帶到模組化單體式應用。

模組化單體

作為單行程單體的子集，**模組化單體**是一種變形，其中單行程是由不同的模組所構成。每個模組可以獨立工作，但仍需要全部組合在一起進行部署，如圖 1-7 所示。將軟體分解成模組的概念並不是什麼新鮮事，模組化軟體起源於 1970 年代結構化程式設計相關的工作，甚至更早。儘管如此，我還沒有看到夠多的組織正確地使用這種方法。

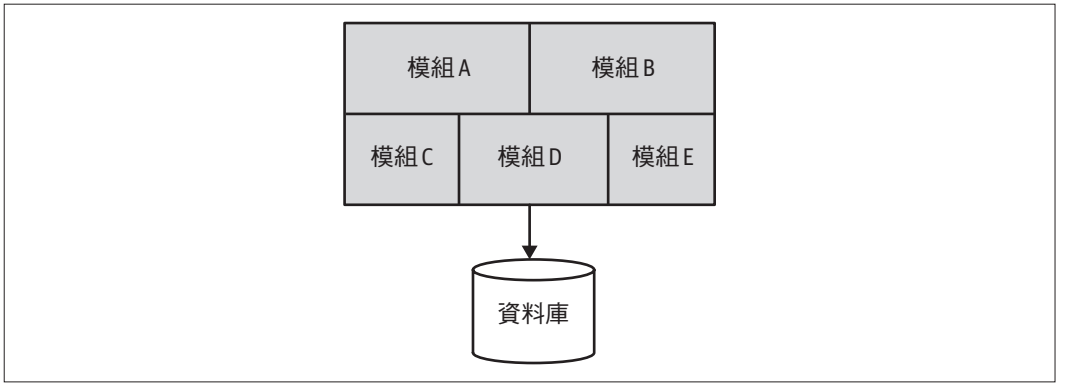

圖 1-7　在模組化單體式應用中，行程中的程式碼分別在各模組中

對於許多組織來說，模組化單體式應用是很好的選擇。如果模組邊界定義得好，它可以允許高度並行的工作，同時透過更簡單的部署拓撲（deployment topology）避免太多分散式微服務架構帶來的挑戰。Shopify 是一個很好的例子，使用這種技術作為微服務分解（decomposition）的替代方案，對其非常有用[6]。

其中一個模組化單體式應用所帶來的挑戰是，資料庫往往缺少我們在程式碼可見的分解，這將會在未來想要拆分單體式應用時，帶來重大的困難。我看到一些團隊試圖透過按照與模組相同的方式來分解資料庫，以進一步推動模組化單體的想法，如圖 1-8 所示。

5　David Heinemeier Hansson 刊登於 Signal v. Noise 期刊的論文〈The Majestic Monolith〉（February 29, 2016，*https://oreil.ly/WwG1C*）。

6　有關 Shopify 使用模組化單體式系統，而非使用微服務背後的想法，請觀看 Kirsten Westeinde 的〈Deconstructing the Monolith〉影片。

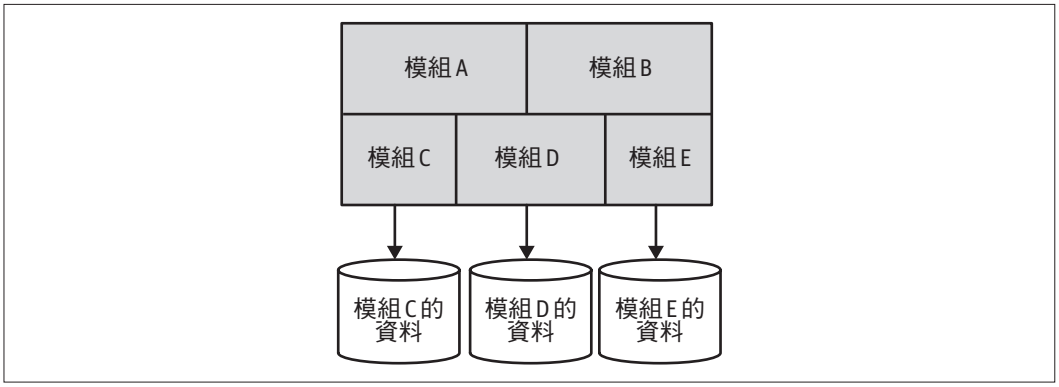

圖 1-8　具有分解資料庫的模組化單體式應用

分散式單體

> 在分散式系統中，未知的故障問題可能導致你的電腦無法使用。[7]
>
> —Leslie Lamport

分散式單體是由多個服務組成的系統，但無論如何，整個系統都必須在一起部署。分散式單體很可能符合 SOA 的定義，但它常常無法兌現 SOA 的承諾。根據我的經驗，分散式單體式應用具有分散式系統的所有缺點和單行程單體的缺點，但沒有足夠的優點。在我工作中遇到許多分散式單體，大大地影響了我對微服務架構的興趣。

分散式單體系統通常出現在對於資訊隱藏和業務功能內聚等概念不夠注重的環境中；取而代之地，高度耦合的架構會導致服務邊界的變化，而這些看似無害、局部範圍內的變化，將會破壞系統的其他部分。

單體式系統與交付競爭

隨著越來越多人在同個地方工作，他們會彼此妨礙；例如，不同的開發人員想要變更同一段程式碼、不同的團隊想要在不同時間即時推送功能（或延遲部署），以及對於誰擁有什麼和誰能做決定的困惑。許多研究顯示，所有權界限混亂會帶來許多挑戰[8]。我將這個問題稱為交付競爭（*delivery contention*）。

7　Leslie Lamport 在 1987 年 5 月 28 日 12:23:29 PDT 發送給 DEC SRC 公告板的 email 訊息（*https://oreil.ly/2nHF1*）。

8　Microsoft Research 在這領域有所研究，我推薦其所有的研究，但若作為一個起頭，我推薦由 Christian Bird 等人所寫的〈Don't Touch My Code! Examining the Effects of Ownership on Software Quality〉（*https://oreil.ly/0ahXX*）。

擁有單體式應用並不意味著你一定會遇到交付競爭的問題，就像擁有微服務架構不表示永遠不會面臨問題一樣。但是微服務架構確實提供了更具體的邊界，讓你在系統中畫出所有權界線，在減少此問題時提供你更大的彈性。

單體式系統的優點

有些單體系統，如單行程或模組化單體，也有許多優點。它們更簡單地部署拓撲可以避免許多與分散式系統相關的陷阱，這可以大大地簡化開發人員的工作流程，並且也可以簡化監控、故障排除和端到端測試等活動。

單體式系統還可以用於簡化單體內部的程式碼重利用（code reuse）。如果我們想在分散式系統中重利用程式碼，我們需要決定是要複製程式碼、拆分程式庫，還是將共用功能推送到服務中。對於單體式應用，我們的選擇來得簡單許多，而且很多人喜歡這種簡單——所有程式碼都在那裡——只要用就好！

但不幸的是，人們已經開始將單體式系統視為本質上有問題、需要避免的東西。我遇過很多人，他們認為**單體式系統**是**遺產**的同義詞。這是個問題。單體式架構是一種選擇，而且是一種有效的選擇。更進一步來說，在我看來，這是一個作為架構風格合理的預設選項。換句話說，我正在尋找一個能被說服使用微服務的理由，而非尋找一個不使用的理由。

如果我們落入了系統上削弱單體式作為交付軟體時預設選項的陷阱中，我們可能對自己或對軟體使用者做錯事情了。

賦能技術

正如我先前提到的，當你第一次開始使用微服務時，我認為你不需要採用很多新技術；實際上，這可能會適得其反。相反地，當你提升微服務架構時，你應該不斷找出由日益分散的系統所引起的問題，然後找到可能有助於解決問題的技術。

也就是說，技術在採用微服務這概念上扮演了很重要的角色。了解可以幫助我們充分利用此架構的工具將成為任何微服務實作能成功的關鍵。事實上，我想說的是，微服務需要在一定程度上了解可支援的技術，以致於先前邏輯架構和物理架構之間的區別可能會產生問題——若你參與幫助打造微服務架構，你將需要對這兩者有廣泛的理解。

我們將在後續章節中詳細探討這項技術，在此之前，讓我們簡要地介紹一些可能對你決定使用微服務有幫助的賦能技術。

日誌匯總與分散式追蹤

隨著需要管理的行程數量不斷增加，你可能會很難了解你的系統在正式環境中的表現，這會使故障排除更加困難。我們將在第 10 章更深入地探討這些想法，但至少，我強烈主張將日誌匯總系統的實作當作採用微服務架構的先決條件。

當你開始使用微服務時，要慎用過多新技術，話雖如此，日誌匯總的工具是非常重要的，你需要將其作為採用微服務架構的先決條件。

這些系統能讓你收集並匯總所有服務的日誌，提供你一個可以分析日誌的中心位置，甚至可以成為主動警報機制的一部分。有許多選項可滿足多種情況。由於諸多原因，我是 Humio（*https://www.humio.com*）的忠實粉絲，但主要公有雲供應商所提供的簡易日誌服務可能已夠入門使用。

透過關聯 ID 的實作可以使這些日誌匯總工具更加有用，其中單個 ID 可用於一組相關的服務呼叫；例如，可能由於使用者互動作用而觸發一連串的呼叫。透過將此 ID 記錄在每個日誌項目中，就能更容易地將特定呼叫流程相關的日誌區別出來，從而使故障排除更加容易。

隨著你的系統變得越來越複雜，能讓你更好地探索系統狀態的工具變得非常重要，使你能分析多個服務的追蹤、檢測瓶頸，以及能發現一開始你不知道的系統問題。開源工具可以提供其中一些功能，例如著重於等式之分散式追蹤方面的 Jaeger（*https://www.jaegertracing.io*）。

但像是 Lightstep（*https://lightstep.com*）和 Honeycomb（*https://honeycomb.io*）（如圖 1-9）的產品則更進一步地推進這些想法。它們代表了超越傳統監控方法的新一代工具，可以更輕鬆地探索正在運行的系統狀態。你可能已經在使用比較常見的工具，但你實在應該看看這些產品所提供的功能，它們從頭到腳都是為了解決微服務架構的運營者必須處理的各種問題。

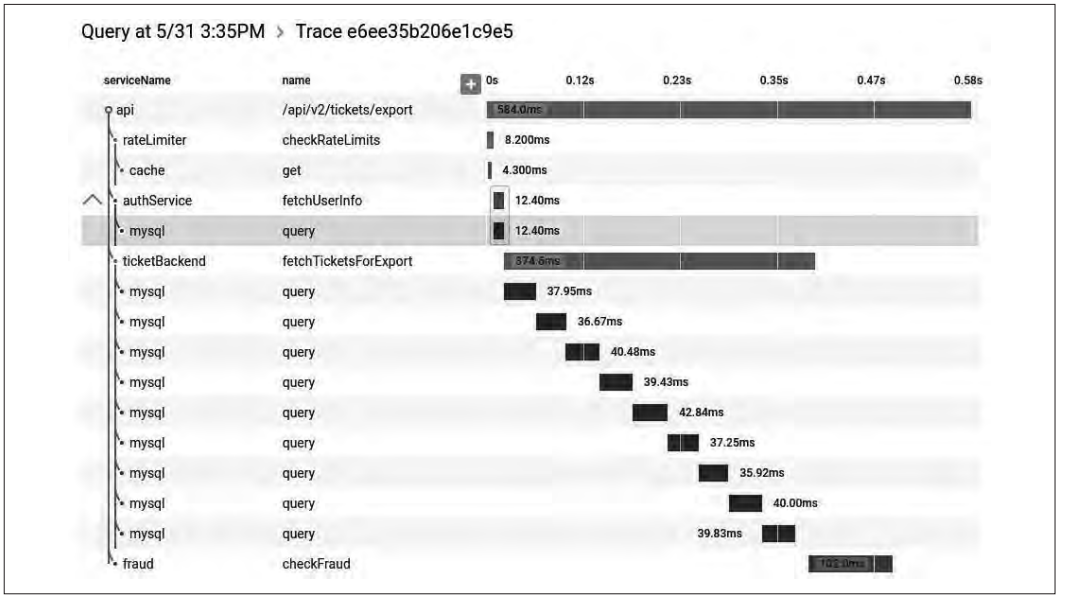

圖 1-9　Honeycomb 中顯示的分散式追蹤，使你能識別跨多個微服務的操作所花費之時間

容器與 Kubernetes

理想上，你會希望在隔離的狀態下運行各個微服務實例，以確保一個微服務的問題不會影響到另一個服務，像是所有的 CPU 被用光了。虛擬化是在既有硬體上創建隔離執行環境的一種方法，但當我們考慮到微服務的大小時，普通的虛擬化技術可能會非常繁重；另一方面，容器提供服務實例一種更輕量的隔離執行方式，從而加快新容器實例的啟動時間，同時對許多架構而言也更具成本效益。

在開始使用容器後，你還會意識到你需要一些東西使你能跨許多底層機器地管理這些容器。像 Kubernetes 這樣的容器編排平台正能這樣做，允許你以提供服務所需之強健性和流通量（throughput）的方式來分配容器實例，同時允許你有效利用底層機器。在第 8 章中，我們將探討操作隔離（operational isolation）、容器和 Kubernetes 的概念。

不要覺得有急著採用 Kubernetes 甚至是容器的必要性。與更傳統的部署技術相比，Kubernetes 和容器具有顯著優勢，但若你只有少量的微服務，則很難證明採用它們是特別好的。等到管理部署的開銷開始成為讓你頭痛的問題時，再開始考慮容器化你的服務和考慮使用 Kubernetes。但是，如果你最終這樣做了，請盡可能確保有其他人為你運行 Kubernetes 叢集（cluster），也許是透過使用公有雲供應商上的託管服務，因為運行自己的 Kubernetes 叢集可能會需要非常多的工作！