前言

多年前，我們當中有些人談論到微服務是一個有趣的想法。接下來您會知道它已經成為全球數百家公司的預設使用的架構（甚至有許多公司可能是為了解決微服務所引起的問題而設立的新創公司），每個人都為了能跟上這波潮流汲汲營營，不然會擔心消失在地平線上。

我必須承擔部分責任。自 2015 年我就此主題撰寫了自己的書《建構微服務》（*https://oreil.ly/building-microservices-2e*）以來，我便一直與人們一同工作，以幫助他們瞭解這類型的架構。我一直在嘗試做的事情就是消除炒作並幫助公司決定微服務是否適合他們。許多我的客戶所面臨的挑戰即是如何將微服務架構套用在他們現有的「非微服務導向」系統上。在不停止所有其他工作的前提下，您要如何採用目前的系統並重建它呢？這正是本書的來源。更重要的是，我的目標乃是對於微服務架構相關的挑戰進行誠實的評估，並幫助瞭解這趟遷移之旅是否適合您。

您將學習到

本書旨在深入探討如何思考、執行，將現有系統分解為微服務架構。我們將會涉及諸多與該架構相關的主題，但重點是在事物的分解方面。想要獲得微服務架構的一般性指南，我前一本著作《建構微服務》會是一個不錯的起頭，事實上，我強烈建議您將它當作本書的姊妹書。

第 1 章概述了什麼是微服務，並進一步探討這類架構的思想。這個章節適合第一次接觸微服務的人，但我也強烈建議那些有經驗的人不要跳過本章。在技術浪潮中，感覺有一些微服務的重要核心概念經常會被遺忘：這也正是本書將要一遍又一遍討論的觀念。

認識微服務是好事，但是要如何知道它適不適合您是另一件事。在第 2 章裡，我將引導您逐步評估微服務是否適合您，也會提供非常重要的指示來指導您如何管理從單體式系統到微服務架構之間的過渡期。我們將介紹從區域驅動的設計到組織變革模型的所有內容，即使您決定不採用微服務架構，這些重要的基礎也將為您帶來良好的發展。

在第 3 章和第 4 章中，我們會更深入探討與分解整體式系統技術相關的方面，探索現實世界中的例子並提取遷移模式。第 3 章著重在應用程序分解層面，而第 4 章則深究資料問題。若您真心想要從單體式系統移動到微服務架構下，那麼將會需要將一些資料庫分解來看！

最終，第 5 章會介紹隨著微服務架構的發展，將面臨的各種挑戰。這些系統雖然可以帶來極大的好處，但同時也帶來許多從未面對的複雜度和問題。本章旨在幫助您發現這些問題，並解決與微服務相關之不斷增長的難題。

本書編排慣例

本書使用下列的編排規則：

斜體字（*Italic*）

表示新術語、URL、電子郵件地址、檔案名稱和副檔名。中文用楷體表示。

定寬字（`Constant width`）

用於程式清單以及在段落中引用程式元素，例如變數或函式名稱、資料庫、資料類型、環境變數、語句和關鍵字。

定寬粗體字（**`Constant width bold`**）

表示應由使用者逐字輸入的指令或其他文字。

定寬斜體字（*`Constant width italic`*）

表示該文字應由使用者提供的值所取代，或者由上下文去判定。

這個圖案代表提示或建議。

這個圖案代表註解。

第一章

足夠的微服務

「好吧，那很快地就要升級了，接著就發展到一發不可收拾的地步了！」

—Ron Burgrundy, *Anchorman*

在深入探討如何使用微服務之前，重要的是我們需要對微服務架構有共識。我想說明一些常見的誤解和容易被忽略的細微差異，您需要扎實的知識基礎才能充分利用本書其餘部分的內容。因此，本章將對微服務架構進行說明，簡要介紹其開發方式（這也意味著需要研究整體單體式系統），並探討使用微服務的一些優勢和挑戰。

什麼是微服務？

微服務泛指圍繞業務領域建模的可獨立部署之服務。它們透過網路彼此溝通，並且作為一個架構體系的選項，它也提供了許多您可能面臨到的問題之解決方案。因此，微服務架構是基於多個共同協作的微服務。

它們是一種服務導向架構（SOA），儘管人們對於應該如何劃分服務邊界有各自的見解，但獨立的可部署性是關鍵。微服務還具有與技術無關的優勢。

從技術角度來看，微服務透過一個或多個網路端點公開了封裝的業務功能。微服務透過這些網絡相互溝通，使它們成為分佈式系統的形式。它們還封裝資料存儲及檢索，並透過定義明確的介面來公開資料，所以資料庫是隱匿在服務邊界內。

有許多東西需要解壓縮來瞭解，因此就讓我們更深入一點地探究其中吧。

獨立部署

獨立部署是指我們可以在不使用到其他任何服務的情形下，變更至微服務並將其部署到生產環境中。更重要的是，這不僅僅是我們「可以」作到，而是它「實際上」就是您在系統中管理部署的方式。這是您在大部分版本中實踐的一個訓練，且這看似簡單的主意，執行起來卻很複雜。

如果您在本書中只學到一件事，那應該是：確保您是接受微服務獨立部署性之概念。養成將單個微服務的變更發佈到生產環境中、且不需部署任何其他東西的習慣。如此，許多美好的事物將隨之而來。

為了保證獨立部署性，需要確認我們的服務是**鬆散耦合**的；換句話說，我們必須作到不更改其他任何內容即可變更一項服務。這意味著我們需要在服務之間建立清楚、定義明確且穩定的約定。某些實施項目就使這塊變得困難，例如共享資料庫就是個問題。對具有穩定介面的鬆散耦合服務的需求，引導我們首先思考如何找到服務邊界。

圍繞業務領域模式

跨程序邊界進行更改的成本是很高的，如果您需要對兩個服務進行變更以推出新功能，並編配這兩個變更的部署，比在單個服務（或稱作單體式系統）中進行相同的更改更費力。因此，我們想要找到一個能夠不經常性跨服務變更的方法。

遵循我在《**建構微服務**》書中所使用的相同方式，本書在無法提供真實案例的情況下，使用虛擬域名和公司來說明某些觀念。這間公司為大型跨國公司 Music Corp，儘管它幾乎以銷售 CD 為主體，但仍有其他營收來源。

我們已決定將 Music Corp 邁向第二十一世紀，並評估現有的系統架構。在圖 1-1 中，我們看到了一個簡易的三層架構。我們有一個基於網站的使用者介面，一個單體式後端形成的業務邏輯層，以及傳統資料庫中的資料存儲。這三層通常由不同的團隊所管理。

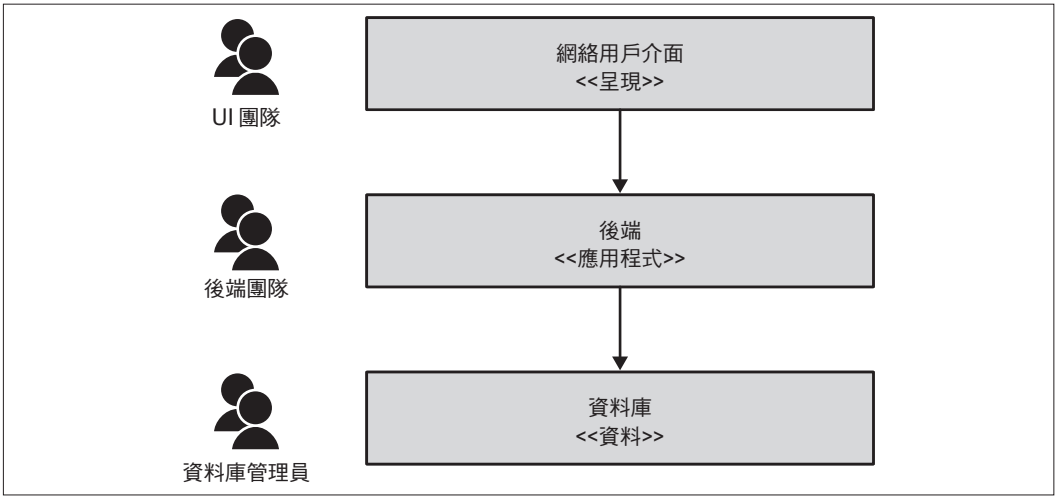

圖 1-1　Music Corp 傳統型三層架構

我們希望對功能進行簡單的變更：像是希望能夠讓客戶自定他們喜愛的音樂類型。而這個變更就需要我們更改使用者介面以顯示所有類型的選項，亦即後端服務需允許這些音樂類型項目顯示在使用者介面且是能夠編輯的，以及將此變更連動到資料庫。這些更改動作需要由每一個團隊依照正確的順序來管理部署，如圖 1-2 所示。

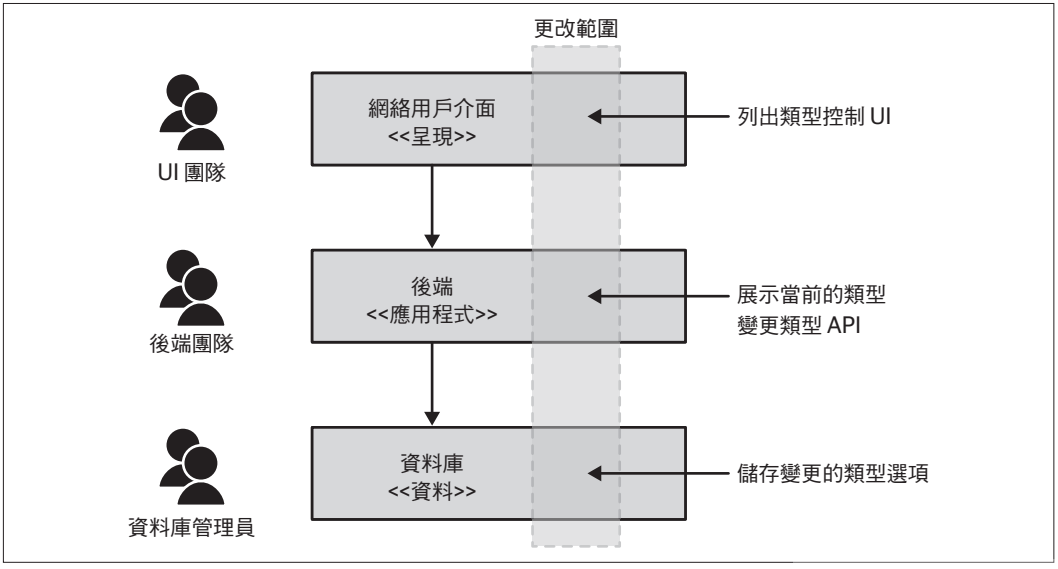

圖 1-2　較複雜的橫跨三層架構的變更

如今，這種架構還不差，所有的架構至終都會因應目標進行優化。三層架構之所以如此普遍，部分原因是它的通用性，且每個人都聽過。因此，選擇一個曾經在其他地方見過的架構也許是我們會不斷看到這種模式的原因之一。但是我認為，不斷看到該架構的最大原因是因為它基於我們組織的方式。

著名的康威定律

> 任何設計系統的組織…都將不可避免地產生以組織通訊架構為複本之設計。
>
> —Melvin Conway, *How Do Committees Invent?*

三層體系架構是實際應用上一個很好的例子。過去 IT 企業將人員根據他們的核心能力來分組：資料庫管理員和其相關人員為一個團隊，Java 開發人員和其相關人員組成一個團隊，前端開發人員（他們現今知道 JavaScript 和本機行動應用程式開發）為另一個團隊。我們根據人員的核心能力進行分組，為此創建了與團隊一致的 IT 資產。

這也解釋了為什麼該架構會如此常見的原因。它只是針對分組力量優化——傳統上人們是按著熟悉程度來編組，但這股力量發生了變化，對軟體的追求也改變了。現今我們會將人員分成多技能團隊以減少之間的交手。我們希望能比以前更快速的發佈軟體，這驅使我們在組織團隊及如何將系統分解的方面上，做出不同的選擇。

功能上的變化主要指的就是業務功能方面上的變化，但是在圖 1-1 中，我們的業務功能實際上分佈在所有三層架構中，從而增加了跨層級功能變化的機會。這樣的架構具有相關技術的高內聚性，但是業務功能的內聚性低。如果我們想要使更改變得容易些，就需要變更訊息碼的分組方式，選擇業務功能的內聚性而非技術。每個服務最終可能包含或不包含這混合的三層架構，但這就是本地服務實現的關注點。

讓我們將上述所說的架構和圖 1-3 揭示的潛在性替代架構進行比較。在這裡有一個專門的客戶服務，提供一個公開的使用者介面予客戶更新他們的資訊，且客戶的狀態也會儲存在該服務中。最喜歡的類型的選項與指定的客戶相關聯，讓此更改更加本地化。圖 1-3 顯示了從目錄服務中獲取的可能已存在之可用類型列表，也看到另一個新的推薦服務正訪問客戶最喜愛的類型資訊，以便在後續的更新版本中輕易實現。

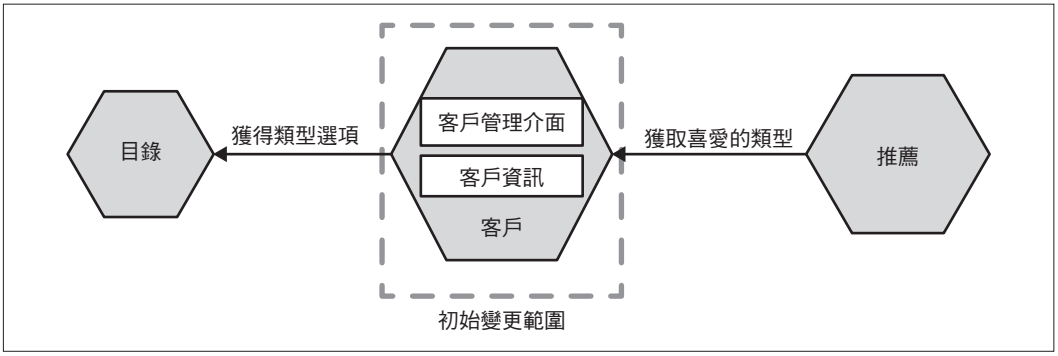

圖 1-3　專用的客戶服務可以容易的記錄客戶喜愛的音樂類型

在此種情形下，我們的客戶服務將三層中的每一層都封裝一個薄片——它具有 UI，邏輯應用程序和資料儲存——但這些層都封裝在單一服務中。

我們的業務領域成為推動系統架構的主力，期許使更改變得更容易，能更輕鬆地管理圍繞業務領域的團隊。這點非常重要，以致於在完成本章之前，我們將重新探訪圍繞領域建模軟體的觀念，因此我可以就圍繞領域驅動的設計分享一些想法，這些想法將影響對微服務架構的看法。

自身擁有的資料

我看到人們感到最困難的事情之一是微服務不應該共享資料庫的想法。如果一個服務想要存取由另外一個服務擁有的資料，它應該要向該服務詢問。這使得服務有能力去決定要共享或隱藏哪些資料，也能讓服務從內部實現細節（可能由於各種原因變化）對應到更穩定的公共契約，以確保穩定的服務介面。如果我們期望達到獨立部署，則服務之間介面的穩定性顯得至關重要。如果服務公開的介面不斷改變，這會產生連鎖反應進而導致其他服務也需更動。

除非需要，否則不要共享資料庫，甚至要盡可能地避免這樣作。如果您要達成獨立部署，在我看來，共享資料庫會是最糟糕的事情之一。

如同前一章節討論到的，我們希望將服務視為端到端的業務功能，在適當的地方封裝 UI、應用程序邏輯和資料存儲，這是因為我們希望減少變更和業務相關功能所需的精力。以這種方式對資料和行為進行封裝能帶來業務功能的高內聚性。透過隱藏支援我們服務的資料庫還可以確保減少耦合的發生，後面會再回來討論關於耦合與內聚。

這很難做到，尤其當您擁有一個現成且具有龐大資料庫的單體式系統時，就必須加以處理。很幸運地，第 4 章整章會說到擺脫單體式資料庫。

微服務帶來的好處？

微服務帶來的好處是多且廣的。部署的獨立性為改善系統規模及強健性開闢了新模式允許混合搭配技術。由於服務能並行運作，因此您可以讓多個開發人員擔負一個問題而不使他們之間相互干擾。如此可以使開發人員更簡易地理解他們在系統中負責的部分，他們也僅需要將注意力集中在其中的一部分。流程隔離還使我們能夠改變所作出的技術選擇，或許可以混合使用不同的程式語言、程式樣式、部署平台或資料庫來找出正確的組合。

也許最重要的是微服務架構帶來的靈活性，為將來解決問題提供更多選擇。

但是請務必注意這些好處不是理所當然的。系統分解有很多種方法，從根本上來講，您嘗試達成的目標使分解朝不同的方向前進。因此，瞭解您試圖從微服務架構獲得什麼變得很重要。

遇到的問題？

服務導向架構之所以成為一個問題，部分原因為計算機變得較便宜因此擁有的就更多了。比起在單一巨大的大型機上部署，更合理的是使用多個便宜計算機。服務導向架構試圖以最好的方法建構出跨多台計算機的應用程序。主要的挑戰項目之一在於計算機之間的相互通訊方式：網路。

計算機之間的網路通訊不是即時的（顯然與物理學有關）。這意味著我們需要在意訊號的延遲性，尤其是那遠遠超過我們在本地所見過以及運作過程中的延遲。這在某些情況會變得更糟，例如延遲產生變化導致系統行為變得難以預測。我們也必須解決以下事實：網路有時會異常，像是封包丟失或斷開的網路電纜。

這些挑戰使得像交易這種單進程之單體式系統的活動變得更加困難；事實上隨著系統複雜性的增加，您將不得不放棄交易及其安全性，然後以其他種類的技術（不幸的是這些技術的取捨大相逕庭）換取回報。

有時會遇到令人頭疼的事件，像是處理任何的網路呼叫失敗、您正在交談的服務沒來由的離線或者是行為開始變得異常。除了上述說的之外，也需要試著找出如何在多台計算機上獲取一致的資料視圖。

當然，我們也要考慮到大量的微服務友好型新技術，萬一使用不當，新技術反而會讓您以更快速、有趣及昂貴的方式出差錯。說實話，微服務除了有它的好處之外，也可能是個可怕的點子。

值得注意的是我們所歸納為「單體式系統」的都是分散式系統。單進程應用程序可能會讀取運行在另一台計算機的資料庫資料，然後將資料呈現到網頁瀏覽器，從此可見至少有三台計算機在這裡，且它們之間是透過網路通訊；與微服務架構相比之下，差異處在於單體式系統的分散程度。當更多的計算機透過網路進行通訊時，就越可能遇到與分散式系統相關且令人討厭的問題。我簡要討論的這些問題可能起初不會出現，但隨著時間和系統的發展，您可能就會遇到絕大多數甚至是全部。

如同我的一位老同事、朋友、微服務專家——James Lewis 所說的「微服務為您購買選擇」。James 不停的思考自己說的那句話——它們為您**購買選擇**。這的確有成本產生，而您要確認的是該選擇是否值得所花費的成本。這部分我們將會在第 2 章有更詳細地探討。

使用者介面

我經常看到的是人們的工作重點僅將微服務應用在伺服器端，讓使用者介面仍為單進程單體式。如果想要能更輕鬆迅速得部署新功能的架構體系，則單體式的使用者介面可能會是一大錯誤。我們可以也應該考慮分散使用者介面，這將會在第 3 章中進行探討。

技術

用一整套新技術來搭配閃亮的新微服務架構可能很誘人，但是我會強烈建議您不要掉入這陷阱。引用任何的新技術都會消耗成本，甚至會產生劇變。希望這些花費是值得的（當然，如果是選擇正確的技術！），但是當初次採用微服務架構時，已經足夠您承受了。

釐清如何正確地發展和管理微服務架構涉及解決與分散式系統相關的眾多挑戰，而這些挑戰可能是前所未見的。我認為，當遇到問題時應即時處理，善用熟悉的技術堆疊，然後考慮是否要變更現有的技術來幫助解決問題，也將更為有用。

正如我們先前談論到的，微服務本質上與技術關係不大，只要服務可以透過網路相互通訊，其他東西都可以隨意擷取。這可說是一項巨大的優勢——允許您隨意混合搭配技術堆疊。

您不必使用 Kubernetes、Docker、軟體容器或雲端，也不必在 Go、Rust 或其他任何語言中進行編碼；實際上，在微服務架構的領域中用何種語言編碼相當不重要，除了某些程式語言具有豐富的內建函式庫和框架以外。假如您很擅長 PHP，那麼請開始以 PHP[1] 來建構服務！某些技術堆疊的技術勢利太多了，這些技術堆疊很不幸地可能會輕視使用特定工具的人[2]。不要成為問題的一部分，選擇適合您的方法，在遇到問題時進行改變以解決問題。

大小

「微服務應該要多大？」大概是我遇過最常見的問題。顧名思義，「微」這個字在這裡不足為奇；但是當您認識到微服務是以一種架構方式運作時，大小的關念實際上是最不有趣的事情之一。

如何測量大小呢？根據程式碼多少行嗎？對我來說那沒有多大意義。某事情用 Java 編碼可能需要 25 行程式碼，但用 Clojure 也許 10 行就完成了。這並不是說 Clojure 比 Java 好或差，就只是有些語言具有較好的表達力罷了。

就微服務「大小」的方面來說，我認為最接近、具有意義的一句話是一位微服務專家 Chris Richardson 曾說的「微服務的目標是具有『盡可能小的介面』。」這與信息隱藏的概念很相似（我們將在稍後討論），但背後的確代表著尋找意義的嘗試，當我們第一次談論到這些東西時，至少最初主要關注的是很容易替換的這些事物。

1 關於此主題的更多資訊，我推薦由 Lorna Jane Mitchell 撰寫的《*PHP* 網路服務》（O'Reilly）。

2 在閱讀 Aurynn Shaw 的一篇部落格文章「Contempt Culture」（*http://bit.ly/2oeICgL*）後，我意識到過去我曾對不同的技術以及所延伸的周圍社群因表現出輕視的態度而感到內疚。

「大小」的概念滿取決於上下文的。曾與一位在系統領域上有長達十五年工作經驗的人交談，他們認為十萬行的程式碼是很容易理解的；但對於案子相關的新成員來說，他們可能會覺得多到難懂。同樣地，詢問一間剛著手於遷移至微服務的公司（也許減少了十個微服務），和一間相似規模但多年以來一直有許多微服務的公司（現今可能數百個），您將會得到不同的答案。

我勸告大家不要擔心大小的問題，剛開始時更重要的是要注意兩個關鍵：首先，您可以處理多少個微服務？隨著服務變多，系統也會變得更複雜，您就需要去學習新技能（或者是採用新的技術）來化解，因此我大力倡導逐步遷移至微服務架構。其次，要如何定義微服務邊界，以及在不把事情搞得一團糟的前提下要如何充分利用它們？ 這部分將會在本章後段介紹。

「微服務」一詞的由來

追溯到 2011 年，當我還在一家名為 ThoughtWorks 的諮詢公司工作時，我的朋友兼同事 James Lewis 就對他所謂的「微型應用程式」非常感興趣。他發現這種模式已經為一些使用服務導向之架構的公司所採用，他們正在優化此架構以易於替換服務。有些遇到問題的公司對快速部署特定功能感興趣，但他們認為若有需要擴展所有功能的時候，在其他的技術堆疊中是可以被重寫的。

在當時浮現的想法是這些服務的範圍有多小，可能其中有些服務在幾天內被編寫（或重寫）；James 接著說道「服務不應該比我頭腦還大」。這個關於功能範圍的想法易於理解也易於改變。

後來在 2012 年，James 在一場架構的高峰會議上分享了這些想法，我們部分的人也在場。在那場會議上，我們討論後發現到這些東西並不是獨立的應用程序，所以用「微型應用程式」來表達不夠貼切；相反地，「微服務」聽起來似乎更合適[3]。

3 我不記得我們第一次真正寫下這個詞是什麼時候了，但我努力回顧了一下當初在考量到所有的邏輯與語法下，我堅持這個詞不應該使用連字號。事後看來，儘管難以辯解但我很堅持。我堅持了看似不合理卻最終贏得勝利的選擇。

所有權

透過圍繞業務領域模式的微服務，我們能夠在 IT 產物（可獨立部署的微服務）與業務領域之間看到一致性。當我們考慮打破「業務」和「IT」之間的鴻溝對技術公司做出轉變之時，引起了廣大的共鳴。在傳統的 IT 組織中，軟體開發的行為通常非由業務部門負責，實際上主要聯繫客戶並定義需求乃是由業務部門負責，如圖 1-4 所示。但這類組織架構所衍生出來的問題繁多，在此就不多加敘述了。

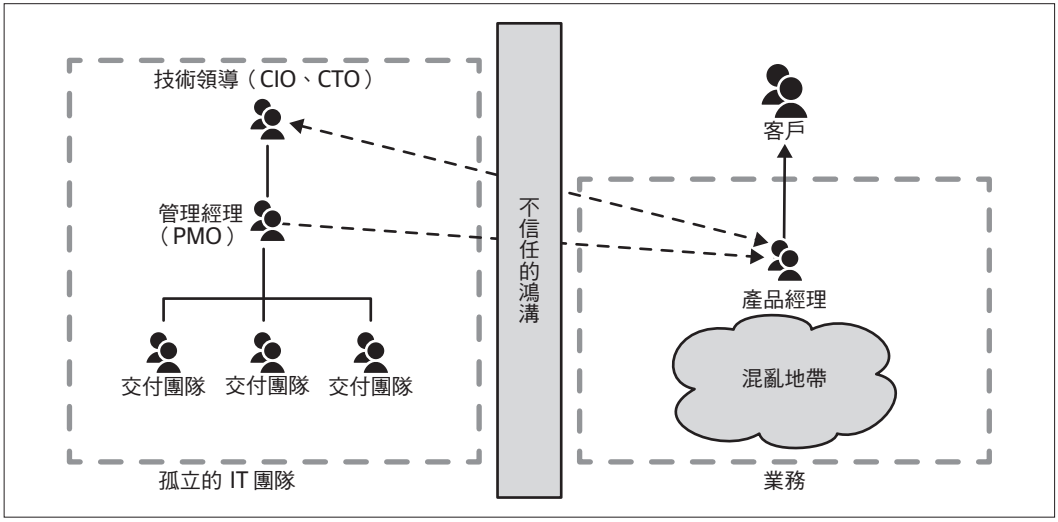

圖 1-4　傳統 IT 和公司業務間鴻溝的組織觀點

我們反而看見正統的科技公司實際上是把前面提到不同的孤立團隊結合在一起，如圖 1-5 所示。產品經理人直接擔負交付團隊的工作，這些團隊一致地面對客戶產品線，而不是隨意的技術分組。集中式 IT 功能雖還不是常態，但他們的存在都是為了支持這些以客戶為中心的交付團隊。

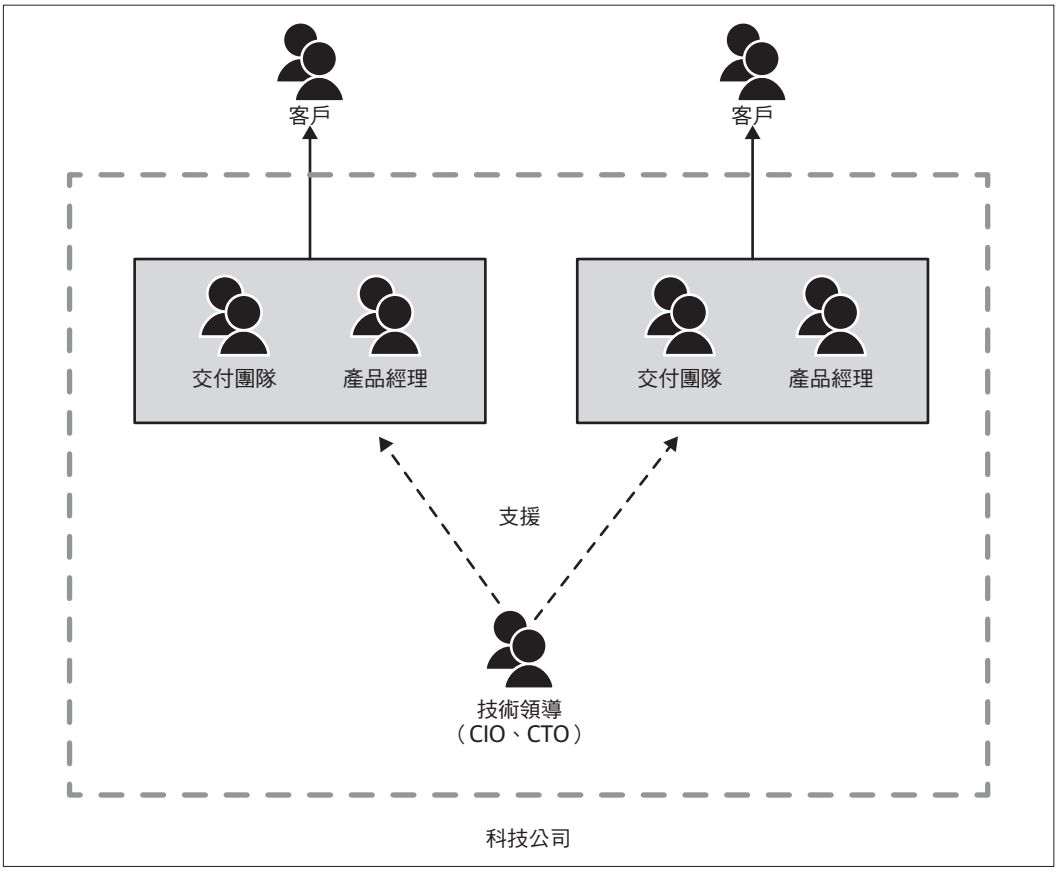

圖 1-5　正統的技術公司如何整合軟體交付團隊的範例

雖然不是所有組織都做了這項轉變，但是微服務架構使改變更加容易。如果您希望交付團隊圍繞產品線進行調整，而服務圍繞業務領域來調整，就明確分配所有權給這些產品導向的交付團隊。

單體式系統

我們已經講完微服務了，但是本書主要是說明從單體式系統遷移到微服務，因此我們還需要瞭解**單體式系統**這詞的含義。

在這本書裡當談論到單體式系統時，我主要是指部署單元。當系統上的所有功能必須一起部署時，我們將其視為單體式系統。至少有三種類型的單體式系統符合要求：單一程序系統、分散式單體系統以及第三方黑盒子系統。

單一程序的單體式系統

當討論到單體式系統時，最容易讓人想到的範例如圖 1-6 所示，即所有程式碼皆是以**單一程序**部署成的系統。考量到系統的強健性和擴展性您可能會有多個類似程序，但事實是所有的程式碼都被包到單一程序裡了。這些單體式系統本身實際上就是簡單的分散式系統，因為它們幾乎總是結束於從資料庫讀取或寫入資料。

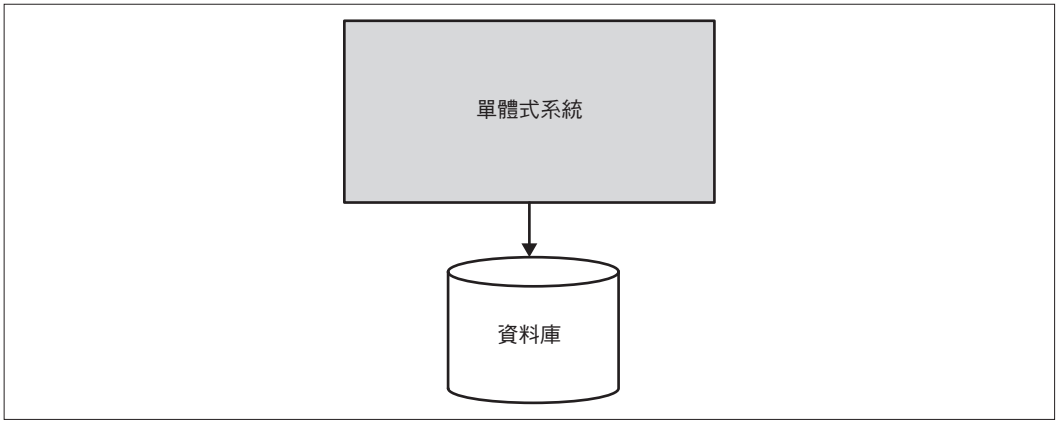

圖 1-6　單一程序的單體式系統：所有的程式碼都被包到單一程序中

這些單體式系統可能就是我所見過的人他們所苦苦掙扎的系統，也因此我們將會把大部分的時間花在此上。從現在開始，當我講到「單體式」一詞時，除非特別說明，否則就是指這類的單體式系統。

模組化單體式

模組化單體式是作為單程序單體式系統的子集合的一個變體：單一程序由個別的模組組成，每個模組都能獨立運作，但為了部署仍需集結起來，如圖 1-7 所示。將軟體分解為模組的概念並不新奇，本章節後段會再回來看有關這方面的一些歷史。

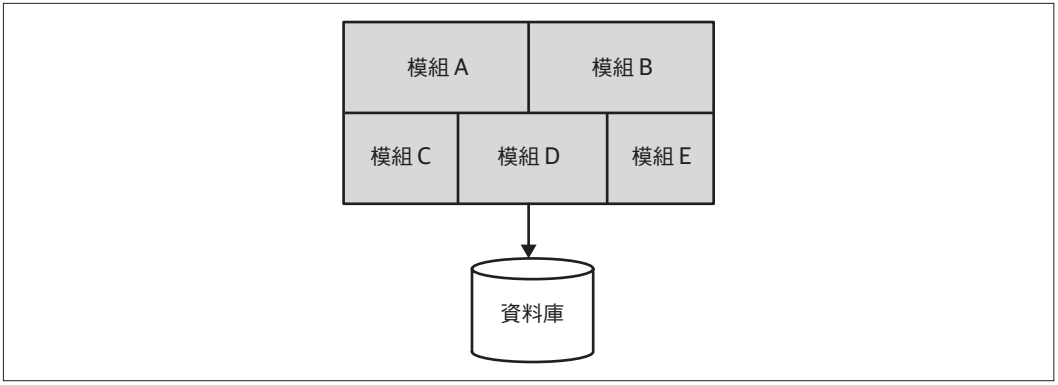

圖 1-7　模組化單體式：程序中的程式碼被分解到模組裡

對於許多組織機構來說，模組化單體式會是個絕佳的選擇。如果模組的邊界定義得宜，它能促進高度的平行運作，還可以避免分散式微服務架構面臨的挑戰以及簡易的部署問題。Shopify 公司即為一個很好的例子，它即是使用此技術來代替微服務分解，且效果似乎很不錯[4]。

如果將來要將單體式系統拉出，模組化單體式可能會面臨的重大挑戰之一即為資料庫缺乏了程式碼級別所採用的分解技術。我已經見過一些團隊試圖進一步推動模組化單體式的想法，使資料庫照著與之相同的方式進行分解，如圖 1-8 所示。即便如此，就算只剩下程式碼，對現有的單體式系統進行改變仍然具有相當的挑戰性；或是若您想要自我嘗試做出類似的改變，第 4 章探討的一些模型或許能提供幫助。

4　關於 Shopify 公司使用模組化單體式系統取代微服務其背後的思想，Kirsten Westeinde 在 YouTube 上的演講（*http://bit.ly/2oauZ29*）提供獨特的見解。

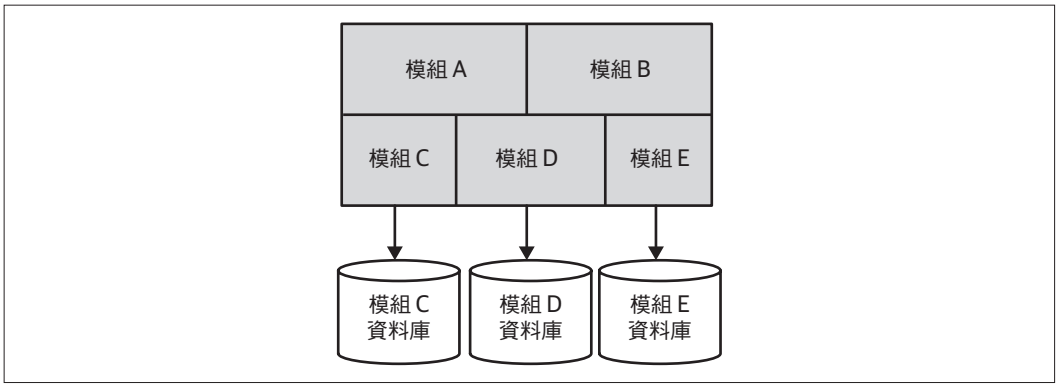

圖 1-8　具有分解資料庫的模組化單體式系統

分散式單體系統

「分散式單體系統是一種計算機存在著未察覺到的故障，可能會導致您的計算機無法使用的系統[5]。」

—Leslie Lamport

分散式單體系統是由多個服務所組成，不管是為了什麼緣故都必須將整個系統部署一起。分散式單體系統可以說是蠻符合服務導向架構的定義，但往往無法兌現 SOA（服務導向架構）的承諾。以我的經驗來說，分散式單體系統集結了所有分散式及單一程序單體式系統的缺點，且沒什麼足夠的進步空間。我曾在工作中遇到分散式單體系統，這在很大的程度上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一個，缺乏注意力於資訊隱藏和業務功能內聚性之類概念的環境，因而造成高度耦合架構；在這些架構中看似無害的變更服務邊界之舉動，實則可能破壞了本機系統的其他部分。

5　於 1987 年 5 月 28 日 12:23:29 PDT，一封寄給 DEC SRC 之公告欄的電子郵件（更多資訊請參考 *https://www.microsoft.com/en-us/research/publication/distribution/*）。

第三方黑盒子系統

我們可以將某些**第三方軟體**視為，欲在遷移工作中對其「分解」的單體式系統。其中可能包括薪資系統、客戶關係管理系統以及人力資源系統。它們的共同特徵是這些軟體是由他人所開發的，您沒有權限更改程式碼。它可能是在自己的基礎架構下部署的軟體，也可能是一個目前正在使用的軟體即服務（SaaS）產品。即便您無法更改底層的程式碼，也能使用到接下來在本書中討論的分解技術。

單體式系統的挑戰

無論是單一程序或分散式的單體式系統，通常較容易受到耦合風險的影響，特別是在實施和部署耦合上，這部分稍後會討論到。

當越來越多人在同一個地方工作時，他們就會開始互相妨礙。不同的開發人員想要更改同一段的程式碼，不同團隊希望在不一樣的時間（或延遲部署）來發布功能，常會有誰負責什麼和誰下決定的困惑。多數研究表明了所有權[6]模糊界線的挑戰，我將此問題稱為**交付爭議**。

擁有單體式系統並不代表您就一定會遇到交付爭議的挑戰；但擁有微服務架構意味著您將永遠不會面臨此問題。微服務架構確實為您在系統中提供了更具體的所有權界線範圍，從而減少此問題並加大了靈活性。

單體式系統的優點

單體式系統也具有許多優點。它的簡易部署拓樸可以避免許多與分散式系統相關的陷阱，以簡化開發流程、監控、疑難排解及端到端測試之類的活動。

單體式系統也可以簡化內部的程式碼重用。如果要在分散式系統中重複使用程式碼，我們必須決定是否要複製、分解函式庫或是將共享功能推入服務中。然而有了單體式系統，這一切選項都變得簡單了且廣受人們喜愛——所有程式碼都在那裡，盡情享用吧！

6 我推薦 Microsoft Research 在這個領域所做的所有相關研究。建議可以 Christian Bird 等人的「Don't Touch My Code! Examining the Effects of Ownership on Software Quality」作為起點去了解（*http://bit.ly/2p5RlT1*）。

很不幸的是，人們將單體式系統視為應避免的對象，這是長久以來固有的問題。我曾遇過許多人，他們認為**單體式系統**是**傳統模式**的同義詞，這是有問題的。單體式系統架構是一種有效的選擇。雖並非在所有情況下都是正確的選擇，但它仍是一種選項。如果我們陷入了系統性破壞單體式的陷阱，以此為可行的方式去發佈軟體，那麼我們會有使軟體用戶或是我們自己無法正確執行操作的風險。我們將在第 3 章進一步探討單體式系統和微服務之間的權衡，以及能幫助您評估適合自身情況的工具。

耦合與內聚

當要定義微服務邊界時，很重要的一點是需要了解耦合與內聚之間的平衡力。**耦合**是說明要更改一個東西時需要改變另一個；**內聚**則是說明如何分類相關的程式碼。這些觀念彼此相連，而 Constantine 定律將此關係闡述得當：

> 「如果內聚性高而耦合度低，那此結構是穩定的。」
>
> —Larry Constantine

這句話似乎是很貼切的觀察，假如有兩段關係緊密的程式碼，內聚性因為相關的功能分布在兩段程式碼中而顯得較低；但它們也有緊耦合，因為當相關的程式碼更動時，這兩者皆需變更。

如果程式碼系統的結構產生變化，處理起來會是一筆昂貴的費用，因此變更橫跨了分散式系統中的服務邊界。必須在一個或多個可獨立部署的服務上進行更改，這也許在應對由服務契約變更衍生的影響上，是一大阻力。

單體式系統的問題在於這兩者經常是相反的。並非傾向於內聚性，而是趨向耦合，獲取各種無相關的程式碼並將欲變更的部分緊連在一起。同樣地，鬆散耦合並不存在：要對一行程式碼進行更改可能很容易，但是要部署此更改且不影響單體式系統中的其餘部分似乎不太可能，因此我必須重新部署整個系統。

另外，系統穩定性也是需要的，因為我們的目標是盡可能套用獨立部署的概念；也就是說，我們期望以**無須做任何其他更改**的方式來將變更過的服務部署到生產中。為此，我們需要的是所使用的服務之穩定性，及為使用我們服務的提供穩定契約。

關於這些術語的大量信息，我在這裡過多重複說明會顯得有點愚蠢，但我認為應該要總結一下，尤其要將這些想法放到微服務架構的內文中。最後要提的是內聚和耦合的觀念會深深影響我們對微服務架構的想法。不足為奇的是內聚和耦合是模組化軟體的關鍵，除了透過網路通訊且可獨立部署的模組以外，微服務架構還可以是什麼？

內聚與耦合的簡歷

內聚與耦合的概念在計算領域已經存在很長時間了，最初是由 Larry Constantine 在 1968 年所提出。此雙重的耦合與內聚觀念後來成了如何編寫計算機程式的基礎。像是由 Larry Constantine 和 Edward Yourdon（Prentice Hall, 1979）合著的書籍《*Structured Design*》隨後影響了好幾代的程式設計員（這是我大學時期必讀的書，距離首次出版已將近 20 年）。

Larry 於 1968 年（對計算機來說是特別吉祥的一年）在全國模組化編程研討會上初次概述他對內聚與耦合的概念，該會議亦為康威法則此名首次誕生的會議。同一年，還有兩場由惡名昭彰的 NATO 贊助的會議，在期間軟體工程的概念也得到重視（此術語以前由 Margaret H. Hamilton 所創造）。

內聚

我聽過用來描述內聚性最簡潔的定義之一是：「程式碼同變動，共存留。」對我們而言這是一個非常好的定義。如同先前討論到的，我們正努力做到簡單改變業務功能以優化微服務架構——所以我們希望將功能分類，盡可能地縮小更改範圍。

假設今天我想要更改銷貨單批准的管理方式，我不需要跨多重服務將其列出來，然後去協調並發佈更新的服務以推出新功能；相反地，我僅需要確保涉及到的相關服務並對其修改，以降低變動成本。

耦合

> 「就像節食一樣，資訊隱藏說的比做的容易。」
>
> —David Parnas，*The Secret History Of Information Hiding*

我們喜歡內聚但對其很謹慎。「耦合」的事物越多，需要一起改變的也越多。不過有不同形態的耦合，而每種類型可能需要不同的解決方案。

當提到要對耦合類型進行分類時，可利用許多現有的技術，其中最著名的是由 Meyer，Yourdan 以及 Constantine 所作的。在此提出我的觀點但不是要否定以前的工作，這個分類法在幫助人們理解與分散式系統耦合相關的方面有莫大的幫助。因此它不是要對不同類型的耦合進行詳盡分類。

資訊隱藏

耦合的討論中反覆出現的概念就是稱為**資訊隱藏**的技術。這個概念最早是由 David Parnas 於 1971 年，在研究要如何定義模組邊界[7]時提出的。

資訊隱藏的核心概念是將經常性變更的部分程式碼與靜態的分開。我們希望有穩定的模組邊界，所以應該要把模組中預期會經常變更的部分隱藏起來。這個理念是基於保持模組相容性的前提下，可以安全地進行內部變更。

就我個人而言，我採用的是盡量不公開模組（或微服務）邊界的方法。一旦有些東西成為模組介面的一部分，之後就很難回頭了。不過如果現在先隱藏它，之後可以隨時決定共享與否。

封裝作為物件導向（OO）的軟體中的概念是相關的，但是根據您接受的定義不同而不盡相同。OO 編碼中的封裝指的是將一或多項事物一起綁到容器中——類別同時包含區域和作用於其上的方法。然後您可以在類別定義中的可見性來隱藏其中某些部分。

若想要探索更多資訊隱藏的歷史，我推薦 Parnas 的《The Secret History of Information Hiding》[8]。

實現耦合

實現耦合是我見過最有害的耦合形式，但很慶幸的是，對我們而言它是輕易削減的一種。在實現耦合中，A 與 B 在實現方式上是耦合的——當 B 的實現發生改變時，A 也會跟著變。

這裡的問題點是，實現細節通常是開發人員的隨意選項。解決問題的方式有許多種，我們今天選擇了一項，但之後也許會改變心意。當我們決定改變心意時，我們不希望會影響到消費者（還記得獨立的部署性嗎？）

7 儘管經常以 1972 年 Parnas 著名的論文「On the Criteria to be Used in Decomposing Systems into Modules」作為來源，但他在 1971 年第 71 期的 IFIP 大會上，首次在「Information Distributions Aspects of Design Methodology」分享了這個概念。

8 請參閱 Parnas 及 David 所著的《The Secret History of Information Hiding》，*Software Pioneers* 出版，M. Broy 和 E. Denert 編輯（Berlin Heidelberg: Springer, 2002）。

實現耦合的經典及常見的案例形式是共享資料庫。在圖 1-9 中，「訂購服務」內含所有置入系統的訂購紀錄；「推薦服務」則根據消費者的歷史購買紀錄來推薦他們可能會想要買的東西，而「推薦服務」可以直接從資料庫存取資料。

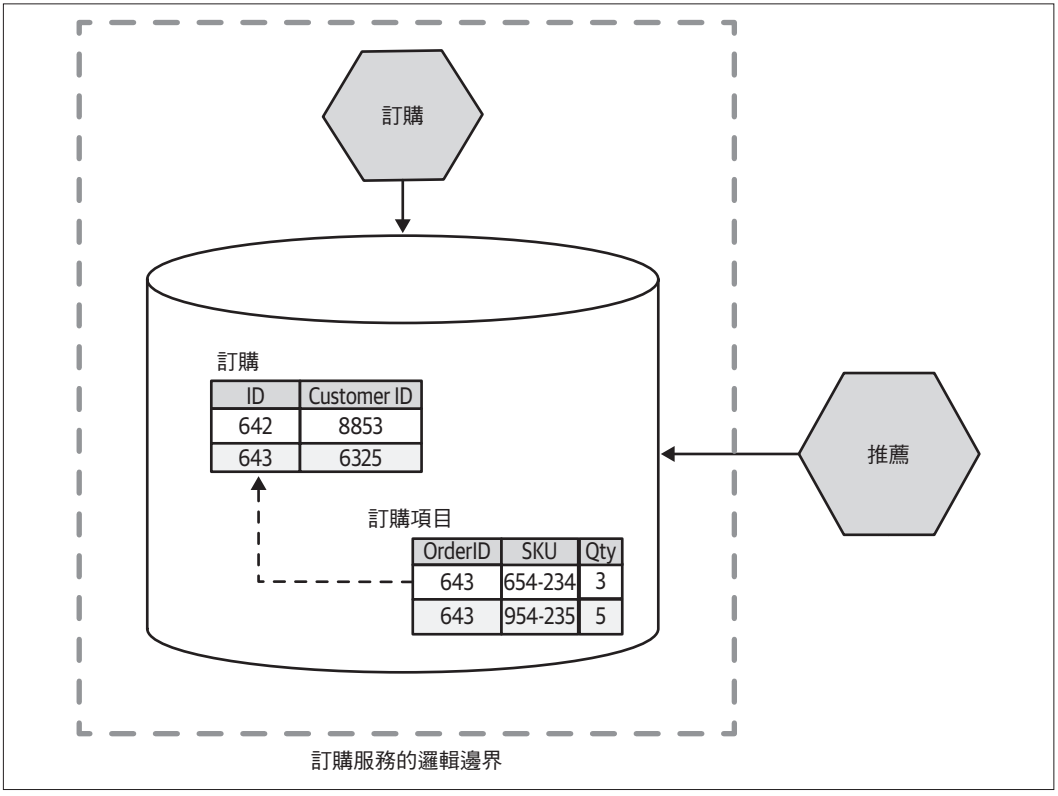

圖 1-9　推薦服務直接存取儲存在訂購服務中的資料

推薦服務需要知道哪些訂購已被置入，以某種程度來說，這是不可避免的域耦合，我們將在稍後討論。但在此特殊情形下，我們要耦合到特定的模式結構、SQL 語言，甚至是行的內容。如果「訂購服務」更改了某一欄位的名稱，將「客戶訂購」資料表拆開或是其他動作，理論上該服務仍然含有訂購訊息，但我們將會改變「推薦服務」取得此訊息的方式。更好的選擇是隱藏實現細節，如圖 1-10 所示，讓「推薦服務」透過呼叫 API 的方式來存取需要的資料。

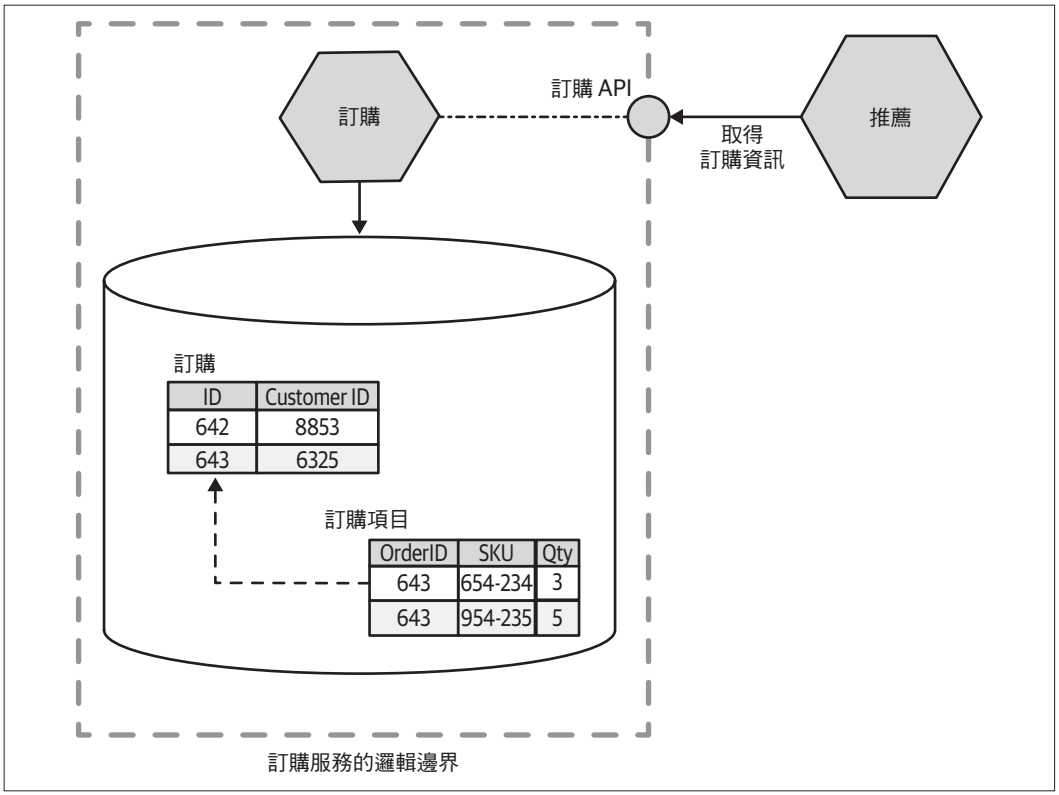

圖 1-10　推薦服務透過 API 的方式來存取訂購資訊以隱藏內部的實現細節

我們還可以讓「訂購服務」以資料庫的形式發佈資料集，用來讓消費者進行批量存取，如圖 1-11 所示。只要「訂購服務」可以相應地發佈資料，任何服務內部的更改對於消費者來說都是不可見的，因為它維護公共合約。這也為公開給客戶的資料模型提供了改善的機會，將根據他們的需求來調整。我們將會在第 3 和 4 章更詳盡探討。