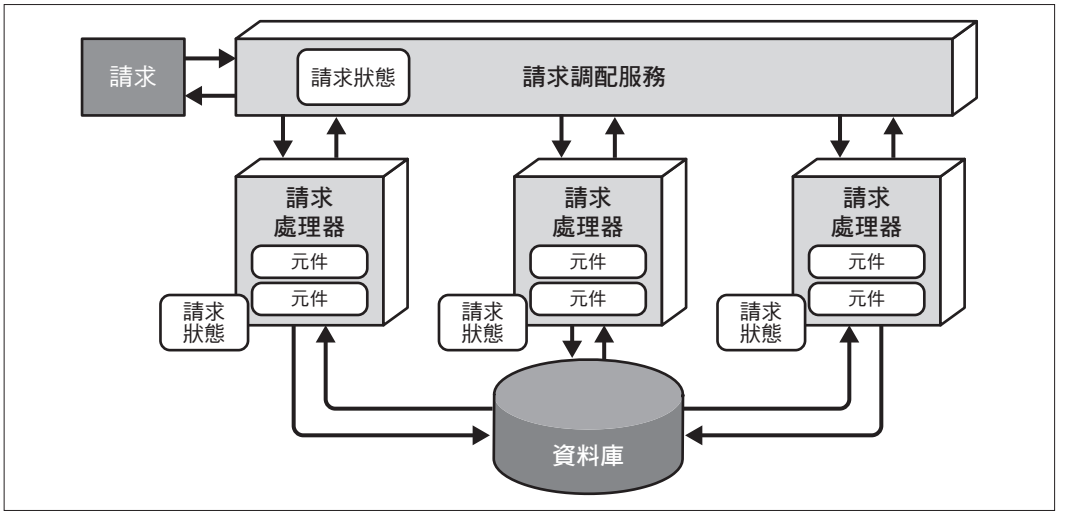


圖 15-1　請求模式

拓撲

事件驅動架構運用非同步的射後不理（*fire-and-forget*）通訊方式，也就是由一個服務觸發事件，再由其他服務回應該事件。這種拓撲有四個主要的架構元件：起始事件（*initiating event*）、事件中介器（*event broker*）、事件處理器（*event processor*）（通常簡稱為服務），以及衍生事件（*derived event*）。

起始事件是啟動整個事件流程的事件。這可以是一個簡單的事件，例如在線上拍賣會中出價，也可以是複雜的事件，例如在員工結婚時，更新健康福利系統。起始事件會被送往事件中介器內的事件通道進行處理。事件處理器會從事件中介器接收起始事件，並開始處理它。

接收起始事件的事件處理器會執行與該事件的處理有關的任務（例如為拍賣標的物出價），然後觸發所謂的衍生事件，並將此事件送至事件中介器，藉以非同步地向系統的其餘部分宣告它做過的事情。其他事件處理器會回應這個衍生事件，執行特定處理，然後透過新的衍生事件來宣告它們做了什麼事情。這個過程會持續進行，直到所有事件處理器都閒置，且所有衍生事件都處理完畢為止。圖 15-2 是這一個事件處理流程。

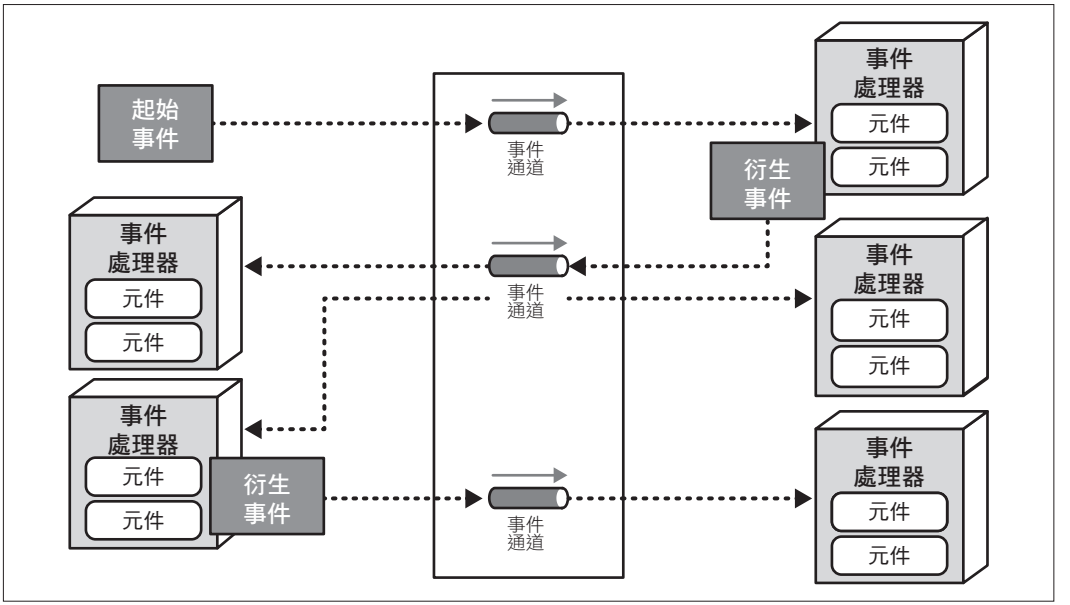


圖 15-2　事件驅動架構的基本拓撲

事件中介器元件通常是聯邦式的（意指它有多個按領域劃分的叢集實例）。每一個聯邦中介器都有該領域的事件流程（處理事件的整個工作流程）所使用的事件通道（例如佇列與主題）。由於這種架構風格有解耦、非同步、射後不理廣播的特性，中介器拓撲會使用主題、主題交換機制（在 Advanced Message Queuing Protocol〔AMQP〕〔*https://oreil.ly/TQDvA*〕的情境下），或是採用發布訂閱傳訊模式的串流。

為了說明 EDA 的整體處理方式，考慮圖 15-3 所示的典型零售訂單輸入系統的工作流程，其中，顧客可以下單購買商品（例如，像這本書一樣的書籍）。在這個範例中，`Order Placement` 事件處理器會接收起始事件（`place order`），將訂單寫入資料庫的資料表中，並將訂單 ID 回傳給顧客。然後，它用一個 `order placed` 衍生事件來向系統其餘部分宣告它已經建立訂單。請注意，有三個事件處理器對這個衍生事件有興趣：`Notification` 事件處理器、`Payment` 事件處理器與 `Inventory` 事件處理器，三者會平行執行各自的任務。

這種拓撲的缺點則取決於所採用的資料庫技術棧（technology stack），它可能是一種非常昂貴的選項。然而，最大的缺點也許是事件處理器之間的同步動態耦合。為了說明這項缺點，我們再次回到 `Order Placement` 事件處理器需要的兩項資訊：書籍庫存與配送選項。在這個拓撲裡，`Order Placement` 事件處理器必須向 `Inventory` 事件處理器以及 `Order Shipment` 事件處理器發出同步呼叫，以取得所需資訊，因而在整個架構中形成緊密的同步耦合點（如圖 15-38 所示）。如同領域資料庫拓撲，在選擇這種資料庫拓撲之前，你要先釐清每一個事件處理器與資料有關的所有需求。

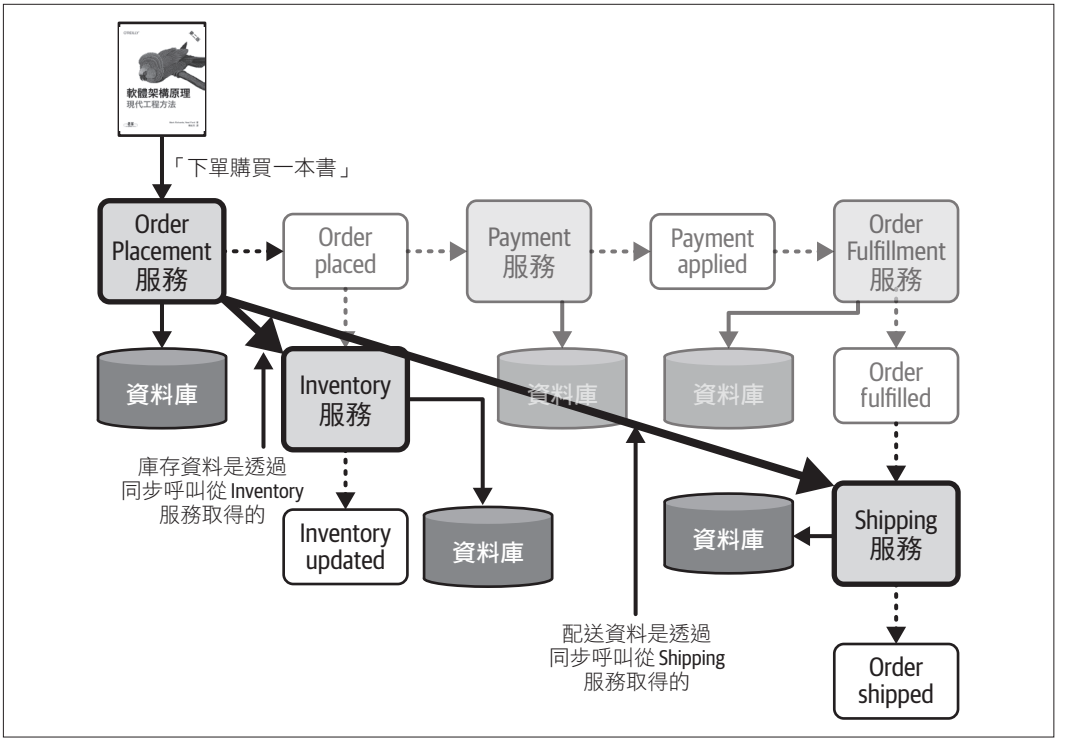
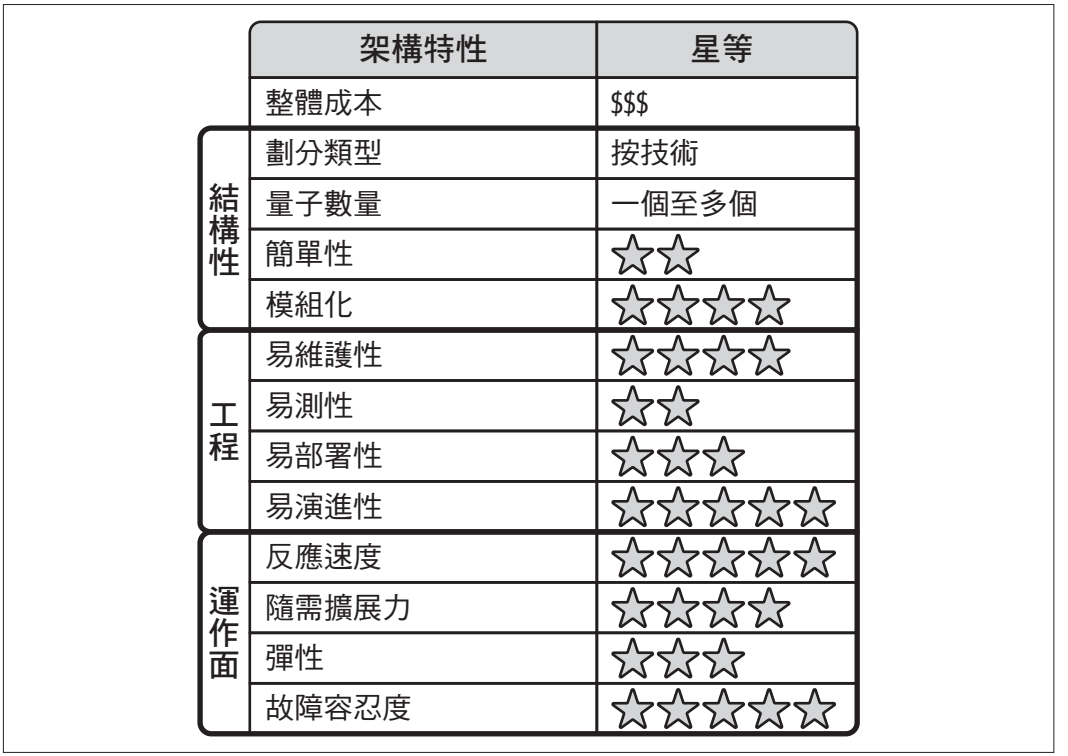


圖 15-38　`Order Placement` 事件處理器需要的資料可能要透過同步通訊，從其他事件處理器取得

如果你的事件處理器幾乎都是自成一體的，只需要存取它自己的有界範疇裡的資料以及相應的資料庫，那麼專屬資料庫拓撲會是很好的選項。如果事件處理器之間的通訊太過頻繁（參見第 274 頁的「治理」），架構師應考慮改採領域甚至是單體資料庫拓撲，以提升整體效能與隨需擴展力。然而，如果在特定情況下需要經常變更資料庫的結構，為了盡量減少受影響的事件處理器，你可能要在這些運作面特性之間做出取捨。



	架構特性	星等
	整體成本	$$$
結構性	劃分類型	按技術
	量子數量	一個至多個
	簡單性	☆☆
	模組化	☆☆☆☆
工程	易維護性	☆☆☆☆
	易測性	☆☆
	易部署性	☆☆☆
	易演進性	☆☆☆☆☆
運作面	反應速度	☆☆☆☆☆
	隨需擴展力	☆☆☆☆
	彈性	☆☆☆
	故障容忍度	☆☆☆☆☆

圖 15-39　EDA 架構的特性評分

在 EDA 裡的量子數量可能從一個到多個不等，取決於每一個事件處理器與資料庫的互動方式，以及系統是否使用 request-reply 處理流程。即使 EDA 的通訊依賴非同步呼叫，如果有多個事件處理器使用同一個資料庫實例，它們也屬於同一個架構量子。在使用 request-reply 處理流程時也一樣，即使事件處理器之間的通訊仍然是非同步的，只要有事件使用端需要立刻取得回應，這些事件處理器就會以同步的方式綁在一起，形成單一量子。

舉例來說，假設有一個事件處理器向另一個事件處理器送出請求，想要建立訂單。第一個事件處理器必須收到另一個事件處理器傳來的訂單 ID 才能繼續工作。如果第二個事件處理器（負責建立訂單並產生訂單 ID 的那個）故障了，第一個事件處理器就無法做下去。這意味著，即使這兩個事件處理器都傳送與接收非同步訊息，它們依然屬於同一個架構量子，並且有相同的架構特性。

例如其他有界範疇的資料庫或類別定義。這讓每一個範疇只定義自己需要的東西，不必迎合其他部分，因此在不同有界範疇之間很難重複使用元件。

雖然重複使用通常是有益的，但還記得軟體架構第一法則嗎？一切都是取捨的結果。重複使用的缺點在於實現它通常會增加系統的耦合度，不論是透過繼承還是組合。

如果架構師的目標是設計高度解耦的系統（這也是微服務的主要目標），那麼他們會偏好複製，而非重複使用，將有界範疇的邏輯概念設計成一個服務及其對應的資料。

拓撲

微服務的基本拓撲如圖 18-1 所示。由於這種架構風格的單一用途特性，它裡面的服務比其他分散式架構，例如 orchestration-driven SOA（第 17 章）、事件驅動架構（第 15 章）、service-based 架構（第 14 章）要小得多。架構師想讓每一個服務具備獨立運作所需的所有部分，包括資料庫與其他依賴元件。

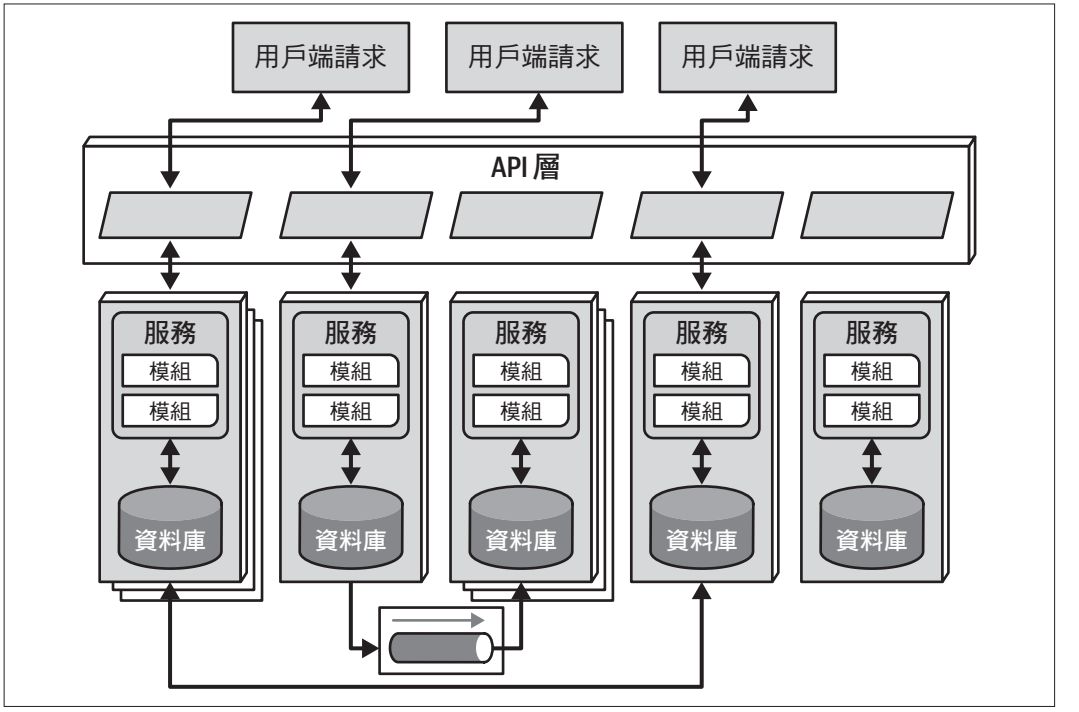


圖 18-1　微服務架構風格的拓撲

編排與調配

編排（*choreography*）採用與 EDA 一樣的通訊風格。運用編排的架構沒有中央協調器，且遵循有界範疇的理念，讓開發者可以在服務之間自然地實作解耦的事件。

在編排中，每一個服務都會在需要時自行呼叫其他服務，而不依賴中央中介者。例如，考慮圖 18-7 所示的情境。使用者想要取得另一位使用者的願望清單細節。由於 `CustomerWishList` 服務沒有它需要的所有資訊，它會呼叫 `CustomerDemographics` 以取得缺少的資訊，然後將結果回傳給使用者。

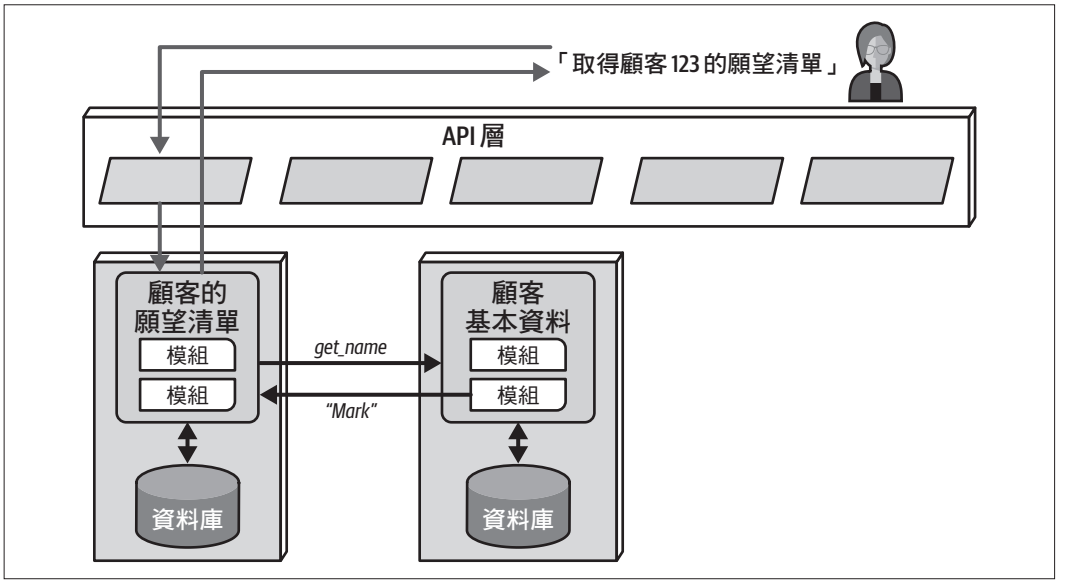


圖 18-7　在微服務中使用編排來管理協作

由於微服務架構不像其他服務導向架構一樣具備全域中介者，如果架構師需要在多個服務之間進行協調，他們可以建立自己的在地中介者（通常稱為調配服務〔*orchestration service*〕。

	架構特性	星等
	整體成本	$$$$$
結構性	劃分類型	領域
	量子數量	一個至多個
	簡單性	☆
	模組化	☆☆☆☆☆
工程	易維護性	☆☆☆☆☆
	易測性	☆☆☆☆☆
	易部署性	☆☆☆☆☆
	易演進性	☆☆☆☆☆
運作面	反應速度	☆☆
	隨需擴展力	☆☆☆☆☆
	彈性	☆☆☆☆
	故障容忍度	☆☆☆☆☆

圖 18-16　微服務架構的特性評分

在這種架構風格裡，獨立、單一用途，因而粒度細小的服務通常帶來高故障容忍度，因此這項特性的分數很高。

這種架構的其他高分特性還有隨需擴展力、彈性與易演進性。在有史以來最具隨需擴展性的系統中，有些系統都成功地採用了微服務。同理，由於這種風格高度依賴自動化，以及與運作（operation）之間的巧妙整合，架構師可以在設計中支援彈性。由於這種架構在演進時的每一步都傾向高度解耦，它也支援現代企業的演進式變更法，甚至可在架構層面上支援。現代企業動作迅速，軟體開發一直賣力地追上腳步。由極小且高度解耦的部署單元來組成的架構能夠支援更快的變更速度。

效能經常是微服務的問題。分散式架構必須進行許多網路呼叫才能完成工作，這會帶來很高的效能成本。

將生成式 AI 納入架構

如果你要將 Gen AI 納入架構，我們建議採用抽象化與模組化。你必須有能力將目前的 LLM 迅速地換成另一個 LLM，並且讓系統能夠設置防護機制（rails），以及評估（evals）不同 LLM 的結果。

例如，假設有一家求職平台想要利用 Gen AI 來將履歷匿名化，他們的目的是降低偏見，讓雇主把重點放在求職者的技能，而非人口統計或其他條件上。這通常是人力工作，但 LLM 也能輕鬆做到。不過 LLM 的結果是否精確？它會不會把履歷中的太多資訊刪掉了？又或者，它會不會保留太多人口統計資料？這種系統必須能夠蒐集樣本與指標，並比較各種 LLM 引擎。Langfuse（*https://langfuse.com*）等工具能幫你在架構中建立這類的觀察機制。

將生成式 AI 當成架構師助理

只要開發者提供合適的提示詞，LLM（例如 Copilot〔*https://oreil.ly/kaEdv*〕）就能產生程式碼，幫他們節省許多時間與心力。LLM 特別適合用來處理非常明確的問題，例如「寫出一段 C# 程式，讓它可以產生不含重複數字且未曾出現過的四位數 PIN 碼」。但是 LLM 技術能夠協助軟體架構師處理常見任務嗎？以下是一些與架構有關的提示詞範例：

- 風險評估：「在這個架構裡有沒有哪個區域存在風險？」
- 風險緩解：「我該如何處理這個風險？」
- 反模式：「這個架構有沒有常見的反模式？」
- 決策：「這個工作流程應該採用調配還是編排？」

截至本書第二版撰稿時（2025 年初），我們還沒有獲得太大的成功。詢問 LLM 在特定情境下應該選擇微服務還是 space-based 架構幾乎（甚至根本）不會獲得正確的答案。為什麼？因為正如本書所述，*在軟體架構中的一切都是取捨的結果*。LLM 雖然擅長理解知識，但是到目前為止，它們仍然缺乏做出適當決策的智慧。這種智慧涉及大量脈絡，因此對架構師來說，自己解決商業問題比教導 LLM 認識該問題以及所牽涉的廣泛環境及脈絡要快得多。光是從我們列出八項需要額外關注的交會處，就足以看出這是一項艱鉅的任務了。