第一部分

機制

演化式架構包括兩個廣泛的研究領域：**機制**（*mechanics*）和**結構**（*structure*）。

演化式架構的**機制**涉及工程實踐和驗證，這些實踐和驗證使架構得以演進，並與架構治理（architectural governance）互相重疊。這包括工程實務做法、測試、衡量指標以及其他一系列使軟體的演化變得可能的組成要件。第一部分定義並舉例說明了演化式架構的運作細節。

《**建立演進式系統架構**》的另一個面向涉及軟體系統的結構或拓撲（topology）。某些架構風格（architecture styles）是否更有利於建置更易於演化的系統？為了使演化更加容易，是否應該避免架構中的一些結構性決策？我們將在第二部分回答這些問題和其他問題，該部分涉及為了便於演進而結構化的架構。

建立演化式架構的諸多面向都結合了機制和結構；本書第三部分的標題是「影響（Impact）」。該部分包含了許多案例研究，提供建議，涵蓋模式和反模式（antipatterns），以及架構師（architects）和團隊為使演化變得可能而需要注意的其他事項。

當一切都在不斷變化，如何實現長期規劃？

我們使用的程式設計平台就是持續演化的例證。新版本的程式語言會提供更好的應用程式介面（API），提高了因應新問題的靈活性或適用性；新的程式語言提供不同的典範和不同的構造集合（set of constructs）。舉例來說，Java 是作為 C++ 的替代品被引進的，試圖減輕編寫網路程式碼的難度、並改善記憶體管理的問題。回顧過去的 20 年，我們觀察到，許多語言仍在不斷改進其 API，而全新的程式語言似乎會定期出現，以解決較新的問題。圖 1-3 展示了程式語言的演化過程。

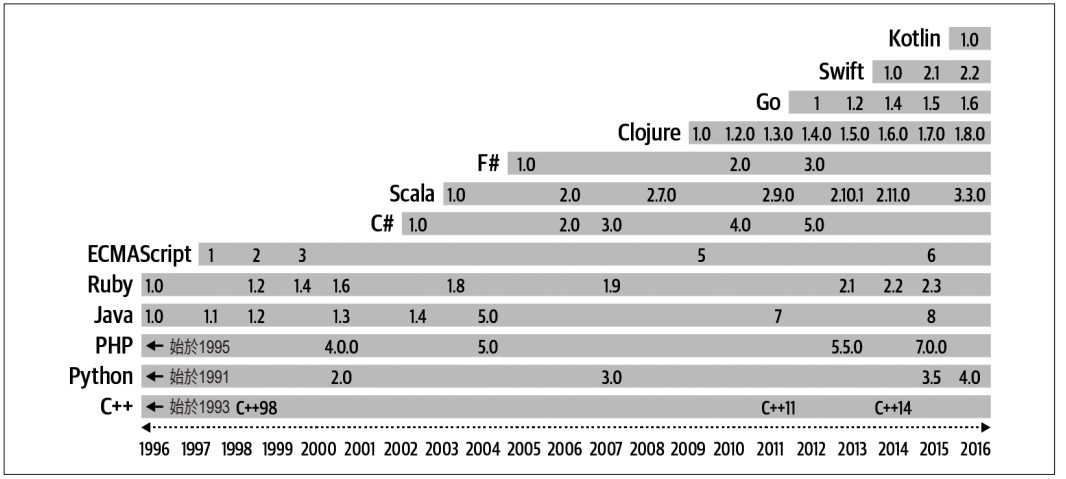

圖 1-3　熱門程式語言的演化

不管是軟體開發的哪個特定面向，諸如程式設計平台、語言、運營環境、續存技術、雲端產品等等，我們都預期不斷的變化。雖然我們無法預測技術或領域環境何時會發生改變，或者哪些變化的影響會持續存在，但我們知道變化是不可避免的。因此，我們在架構系統時，應該知曉技術環境會發生變化。

如果生態系統以意想不到的方式不斷變化，而且可預測性不可能達成，那麼除了固定的計畫之外還有什麼替代選擇呢？企業架構師和其他開發人員必須學會適應。制定長期計畫的部分傳統理由是財務因素；軟體的變更成本昂貴。然而，現代工程實踐卻使這一前提失效，因為透過將以前的人工過程自動化以及 DevOps 等其他進階技術，可以降低改變的成本。

多年來，許多聰明的開發人員都意識到，系統的某些部分比其他部分更難修改。這就是為什麼**軟體架構**被定義為「日後難以改變的部分」。這個便利的定義將你**可以**不費吹灰之力就修改的部分、與真正很難修改的部分切割開來。不幸的是，這個定義也演變成了架構思考的一個盲點：開發人員認為變革是困難的，這成為了一種自我實現的預言（self-fulfilling prophecy）。

幾年前，一些富有創新精神的軟體架構師重新審視了「日後難以改變」的問題：如果我們將可變性融入到架構中呢？換句話說，如果**易於改變**是架構的基本原則，那麼改變就不再會是困難的。而在架構中加入可演化性又會允許全新的行為突現（emerge），再次打破動態平衡。

即使生態系統不發生變化，架構特性也會逐漸受到侵蝕，這又是怎麼回事呢？架構師設計了架構，但隨後將其暴露於會在架構上**實作**（*implementing*）東西的混亂現實世界之中。架構師如何保護他們定義的重要部分呢？

架構建立好之後，如何避免其隨時間而衰退？

在許多組織中都會出現一種不幸的衰退，通常被稱為「**位元腐爛**（*bit rot*）」。架構師選擇特定的架構模式來處理業務需求和「能力（-ilities）」，但隨著時間的推移，那些特性往往會意外地退化。舉例來說，如果架構師建立了一個分層架構（layered architecture），最上層是表現（presentation）層，最下層是續存（persistence）層，中間還有數層，那麼出於效能考量，製作報表的開發人員通常會請求權限，希望直接從表現層存取續存層，而繞過中間的那幾層。架構師建立層級是為了隔離變化。然後，開發人員繞過那些分層，增加了耦合度，並使那些分層背後的理念失效。

一旦定義了重要的架構特性，架構師該如何**保護**那些特性，以確保它們不被削弱呢？將**可演化性**（*evolvability*）作為一項架構特性加入，就意味著在系統演化的過程中保護其他特性。舉例來說，如果架構師為了規模可擴充性（scalability）而設計一個架構，他們就會不希望這一特性隨著系統的演進而劣化。因此，**可演化性**是一種元特性（meta-characteristic），一種會保護所有其他架構特性的架構外殼（architectural wrapper）。

演化式架構的機制與架構治理（architectural governance）的關注點和目標有很大的重疊，而架構治理的原則是以設計、品質、安全性和其他品質考量來定義的。本書闡述了演化式架構做法實現架構治理自動化的多種方式。

第六章

演化式資料

關聯式（relational）和其他類型的資料儲存方式，在現代軟體專案中無處不在，這種形式的耦合往往比架構耦合更容易出問題。資料團隊通常更不習慣採用單元測試和重構等工程實務做法（這一點正在逐漸改善）。此外，資料庫通常會成為整合點，這使得資料團隊不願意進行更改，因為可能會產生副作用漣漪。

資料是建立可演化式架構時需要考慮的一個重要維度。像微服務這樣的架構需要更多的架構考量，例如資料分割、依存關係、交易性和其他一系列之前只屬於資料團隊範疇的問題。本書無法涵蓋演化式資料庫設計（evolutionary database design）的所有面向。幸運的是，我們的合著者 Pramod Sadalage 與 Scott Ambler 共同撰寫了《*Refactoring Databases*》一書（*http://databaserefactoring.com*），副標題即為 *Evolutionary Database Design*。我們只涵蓋資料庫設計中對演化式架構有影響的部分，並鼓勵讀者去閱讀那本書。

演化式資料庫設計

當開發人員可以根據需求的不斷變化來建置和演化資料庫的結構時，資料庫中的演化式設計就會出現。資料庫結構描述（database schemas）是一種抽象概念，類似於類別階層架構（class hierarchies）。隨著底層現實世界發生改變，那些變化必須反映在開發人員和資料團隊所建置的抽象層中。否則，那些抽象結構就會逐漸與現實世界脫節。

演化結構描述

架構師如何才能建置支援演化的系統，但仍使用關聯式資料庫（relational databases）等傳統工具呢？演化資料庫設計的關鍵在於與程式碼一起演化結構描述（schemas）。Continuous Delivery（持續交付）解決了如何將傳統的資料孤島融入現代軟體專案的持續反饋迴路（continuous feedback loop）的問題。開發人員必須像對待原始碼一樣對待資料庫結構的變化：測試、版本控制和逐步進行：

經過測試

資料團隊和開發人員應嚴格測試對資料庫結構描述的更動，以確保穩定性。如果開發人員使用物件對關聯式映射器（object-relational mapper，ORM）等資料映射工具，則應考慮新增適應性函數，以確保映射與結構描述保持同步。

有版本控制

開發人員和資料團隊應將資料庫結構描述和使用資料庫結構描述的程式碼，一起進行版本控制。原始碼和資料庫結構描述有共生（symbiotic）關係，二者缺一不可。人為地將這兩種必然耦合的事物分開的工程實務做法，會造成沒必要的效率低落。

漸進式

對資料庫結構描述的修改應該像原始碼變更的累積一樣，隨著系統的演化而逐漸產生。現代工程實務做法摒棄了手動更新資料庫結構描述的做法，而更傾向於使用自動遷移工具（automated migration tools）。

資料庫遷移工具是允許開發人員（或資料團隊）對資料庫進行小規模漸進式更改的實用工具，這些變更將作為部署管線（deployment pipeline）的一部分自動套用。這些工具的功能範圍很廣，從簡單的命令列工具到複雜的原始 IDE 都有。當開發人員需要對某個結構描述進行修改時，他們會編寫小型的資料庫遷移（database migration，又稱 delta）指令稿，如範例 6-1 所示。

範例 6-1　簡單的資料庫遷移

```
CREATE TABLE customer (
       id BIGINT GENERATED BY DEFAULT AS IDENTITY (START WITH 1) PRIMARY KEY,
       firstname VARCHAR(60),
       lastname VARCHAR(60)
);
```

案例研究：演化 PenultimateWidgets 的路由功能

PenultimateWidgets 決定在頁面之間實作新的路由方案（routing scheme），為使用者提供巡覽軌跡（navigational breadcrumb trail）。這意味著要改變頁面之間的繞送方式（使用內部框架）。實作新路由機制的頁面需要更多的情境（來源頁面、工作流程狀態等），因此需要更多的資料。

在路由服務量子中，PenultimateWidgets 目前只有單一個表來處理路由。對於新版本，開發人員需要更多資訊，因此表格結構將更加複雜。請看圖 6-4 所示的起點。

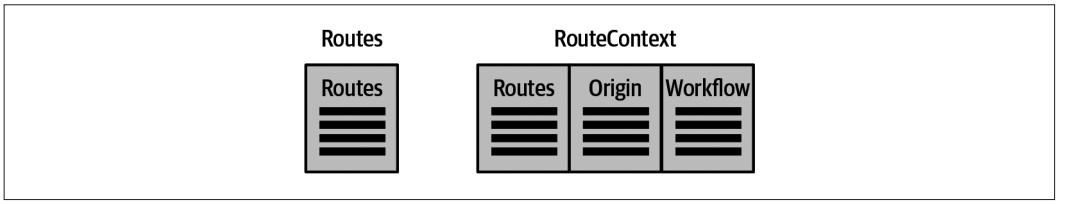

圖 6-4　新路由實作的起點

由於不同業務部門的工作速度不同，PenultimateWidgets 的所有頁面並不會同時實作新的路由。因此，路由服務必須同時支援新舊版本。我們將在第 7 章中了解如何透過路由進行處理。在這種情況下，我們必須在資料層面處理相同的情況。

開發人員可以使用 Expand/Contract 模式建立新的路由結構，並透過服務呼叫使其可用。在內部，兩個路由表都有一個與 `route` 欄相關聯的觸發器（trigger），因此對其中一個路由表的更改會自動複製到另一個路由表，如圖 6-5 所示。

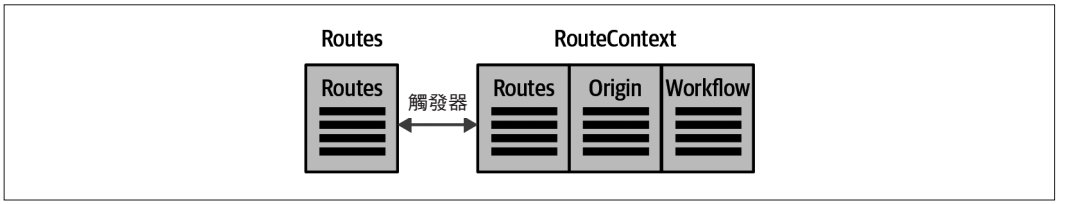

圖 6-5　過渡狀態，服務同時支援兩個版本的路由

如圖 6-5 所示，只要開發人員需要舊的路由服務，該服務就能同時支援兩種 API。實質上，應用程式現在等同於支援兩個版本的路由資訊。

不再需要舊服務時，路由服務開發人員可以刪除舊表和觸發器，如圖 6-6 所示。

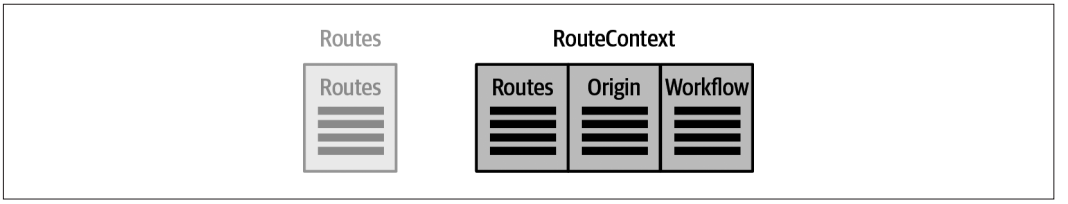

圖 6-6　路由表的結束狀態

在圖 6-6 中，所有服務都已遷移到新的路由功能，從而可以刪除舊服務。這與圖 6-2 所示的工作流程一致。

只要開發人員採用適當的工程實務做法，如持續整合、原始碼控制等，資料庫就能與架構一起演化。這種輕鬆更改資料庫結構描述的能力至關重要：資料庫是基於現實世界的抽象層，而現實世界可能會發生意想不到的變化。雖然資料抽象比行為更能抵禦改變，但它們仍必須不斷演化。在建置演化式架構時，架構師必須將資料作為首要關注點。

重構資料庫是資料團隊和開發人員需要磨練的一項重要技能和手藝。資料是許多應用程式的基礎。要建置可演化的系統，開發人員和資料團隊必須將有效的資料實務做法與其他現代工程實務做法結合起來。

從原生到適應性函數

有時，軟體架構的選擇會給生態系統的其他部分帶來問題。當架構師採用微服務架構時，這種架構建議每個有界情境使用一個資料庫，這改變了資料團隊對資料庫的傳統看法：他們更慣於使用單一的關聯式資料庫，以及那些工具和模型所提供的便利。舉例來說，資料團隊會密切關注參考完整性（referential integrity），以確保資料結構連接點的正確性。

但是，當架構師希望將資料庫分解成微服務之類，由各個服務分別提供資料的架構時，該如何說服持懷疑態度的資料團隊，讓他們相信為了微服務的優勢，值得去放棄一些他們信賴的機制？

由於這是一種治理（governance）形式，架構師可以透過在建置過程中加入持續型的適應性函數，來確保重要元件保持完整性並解決其他問題，從而讓資料團隊放心。

參考完整性

參考完整性是資料結構描述層面的一種治理形式，而不是架構耦合。然而，對於架構師來說，兩者都會因為增加耦合度而衝擊應用程式的演化能力。舉例來說，在很多情況下，資料團隊因為參考完整性而不願意將資料表拆分到單獨的資料庫中，但這種耦合會妨礙與之耦合的兩個服務發生改變。

資料庫中的參考完整性指的是主鍵（primary keys）及其連結。在分散式架構中，團隊也有實體（entities）的唯一識別碼（unique identifiers），通常表示為 GUID 或其他隨機序列。因此，架構師必須編寫適應性函數，以確保在資訊所有者刪除特定項目時，這個刪除動作會傳播到可能仍然參考著已被刪除的實體的其他服務。事件驅動（event-driven）架構中的許多模式都能處理這類背景任務；圖 6-7 就是其中一個例子。

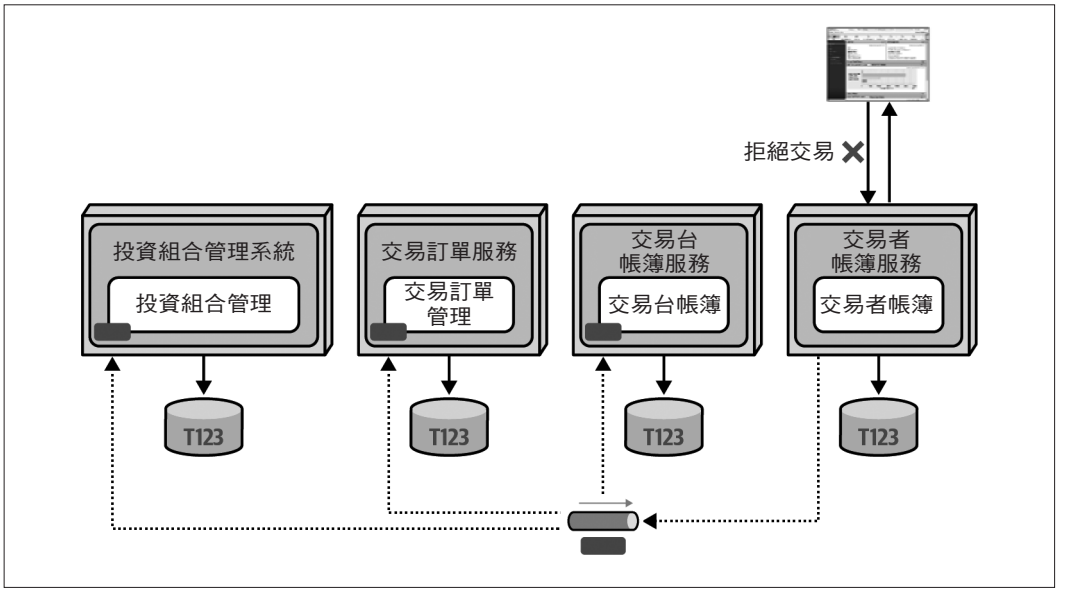

圖 6-7　使用基於事件的資料同步來處理參考完整性

在圖 6-7 中，當使用者介面透過 `Trader Blotter`（交易者帳簿）服務拒絕交易時，它會在持續存在的訊息佇列中傳播一條訊息，由對此感興趣的所有服務監控，並根據需要更新或刪除變更。

雖然資料庫中的參考完整性功能強大，但有時也會產生不必要的耦合，因此必須權衡利弊。

資料重複

如果團隊慣於使用單一關聯式資料庫，他們通常不會將讀取（*read*）和寫入（*write*）這兩種運算分開考慮。然而，微服務架構會迫使團隊更仔細地考慮哪些服務可以更新資訊，而哪些服務只能進行讀取。考慮一下許多剛接觸微服務的團隊所面臨的常見情況，如圖 6-8 所示。

一些服務需要存取系統的幾個關鍵部分，如 `Reference`、`Audit`、`Configuration` 和 `Customer`。團隊應如何處理這一需求？圖 6-8 所示的解決方案與所有感興趣的服務共用這些資料表，這樣做雖然方便，但卻違反了微服務架構的原則之一，即避免將服務耦合到共通的資料庫。如果其中任何一個表的結構描述發生變化，都會波及耦合的服務，可能會要求它們進行更改。

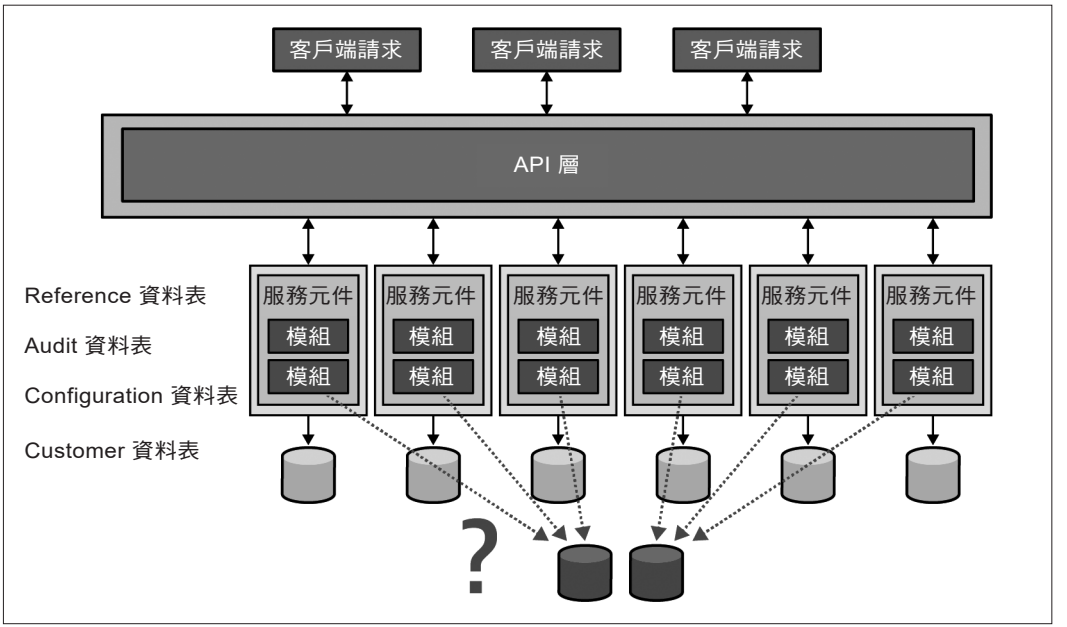

圖 6-8　管理分散式架構中的共享資訊

另一種做法請參閱圖 6-9。

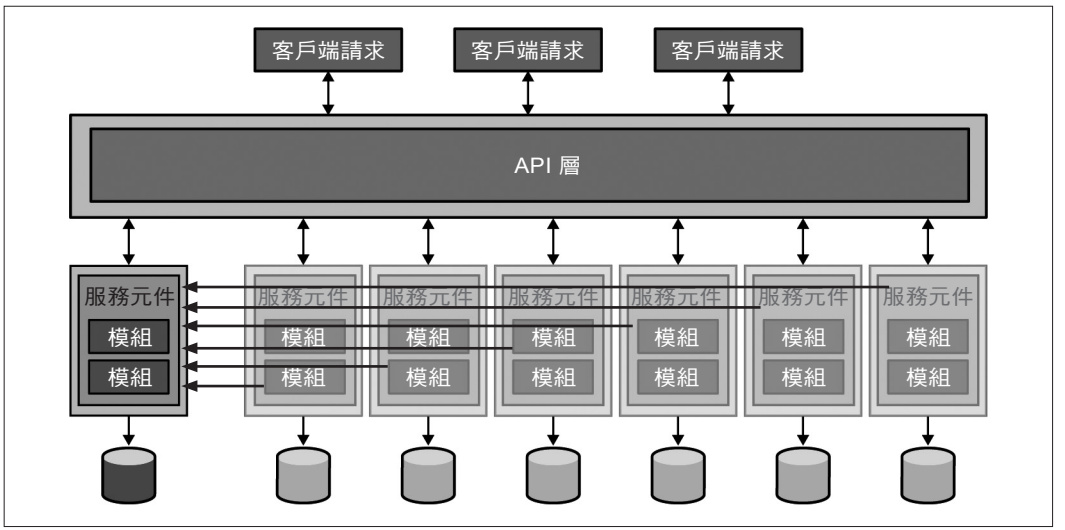

圖 6-9　將共享資訊建模為一個服務

在圖 6-9 中，依循微服務背後的理念，我們將共享的每筆資訊都作為一個獨立的服務來建模。然而，這揭露了微服務的一個問題：服務間通訊過多，會影響效能。

許多團隊常用的做法是仔細考慮誰應該**擁有**（*own*）資料（即誰可以更新資料），而誰可以**讀取**（*read*）資料的某個版本。圖 6-10 所示的解決方案使用行程內快取（in-process caching）進行讀取。

在圖 6-10 中，左邊的服務元件「擁有」資料。不過，在啟動時，每個感興趣的服務都會讀取並快取想要的資料，並以適當的頻率更新快取的資訊。如果右側的某個服務需要更新共享值，它會透過向擁有資料的服務發出請求的方式進行更新，而擁有資料的服務就可以發佈該變更。

在現代架構中，架構師使用各種做法來管理資料的存取和更新。這方面的例子包括變更控制、連線管理的規模可擴充性、容錯、架構量子、資料庫型別最佳化、資料庫交易和資料關係，這在《**軟體架構：困難部分**》（O'Reilly 出版）一書中有更詳細的介紹。

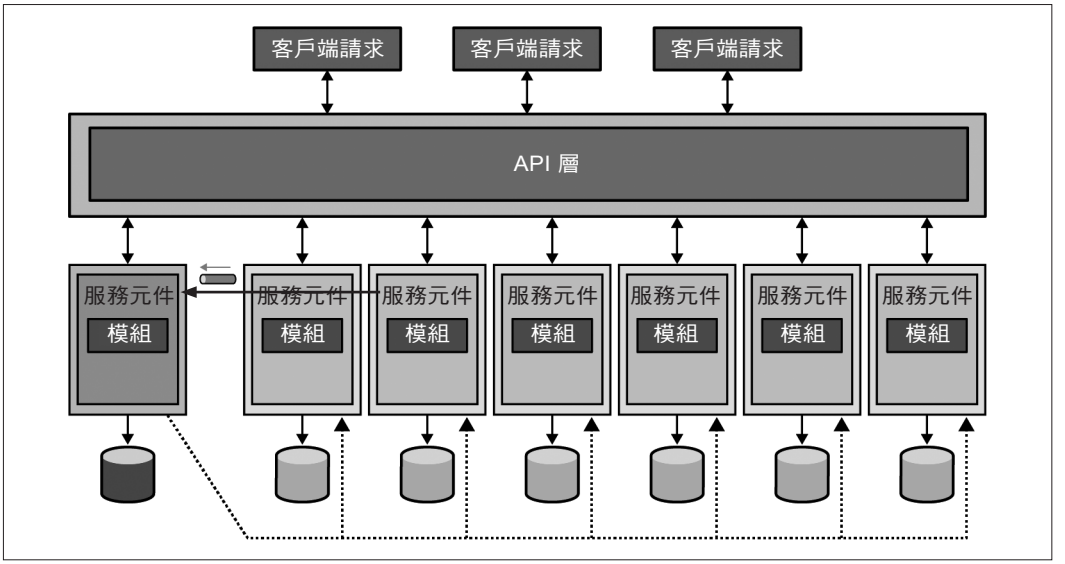

圖 6-10　使用快取進行唯讀存取

取代觸發器和預存程序

資料團隊仰賴的另一個常見機制，是以資料庫原生的 SQL 所編寫的預存程序（stored procedures）。雖然這是一種功能強大且效能卓越的資料操作方式，但它在現代軟體工程實務上卻面臨著一些挑戰。舉例來說，預存程序很難進行單元測試，對重構的支援往往很差，而且會將行為與原始碼中的其他行為分離開來。

遷移到微服務經常會導致資料團隊需要重構預存程序，因為相關資料不再位於單一資料庫中。在這種情況下，行為必須轉移到程式碼中，團隊必須解決資料量和傳輸等問題。在現代 NoSQL 資料庫中，觸發器（triggers）或無伺服器函式（serverless functions）可能會根據某些資料變化而觸發。所有的資料庫程式碼都必須重構。

如圖 6-11 所示，架構師可以使用同樣的 Expand/Contract 模式，將當前預存程序中的行為提取到應用程式碼中，使用 Migrate Method from Database（從資料庫遷移方法）模式（*https://oreil.ly/afabK*）。

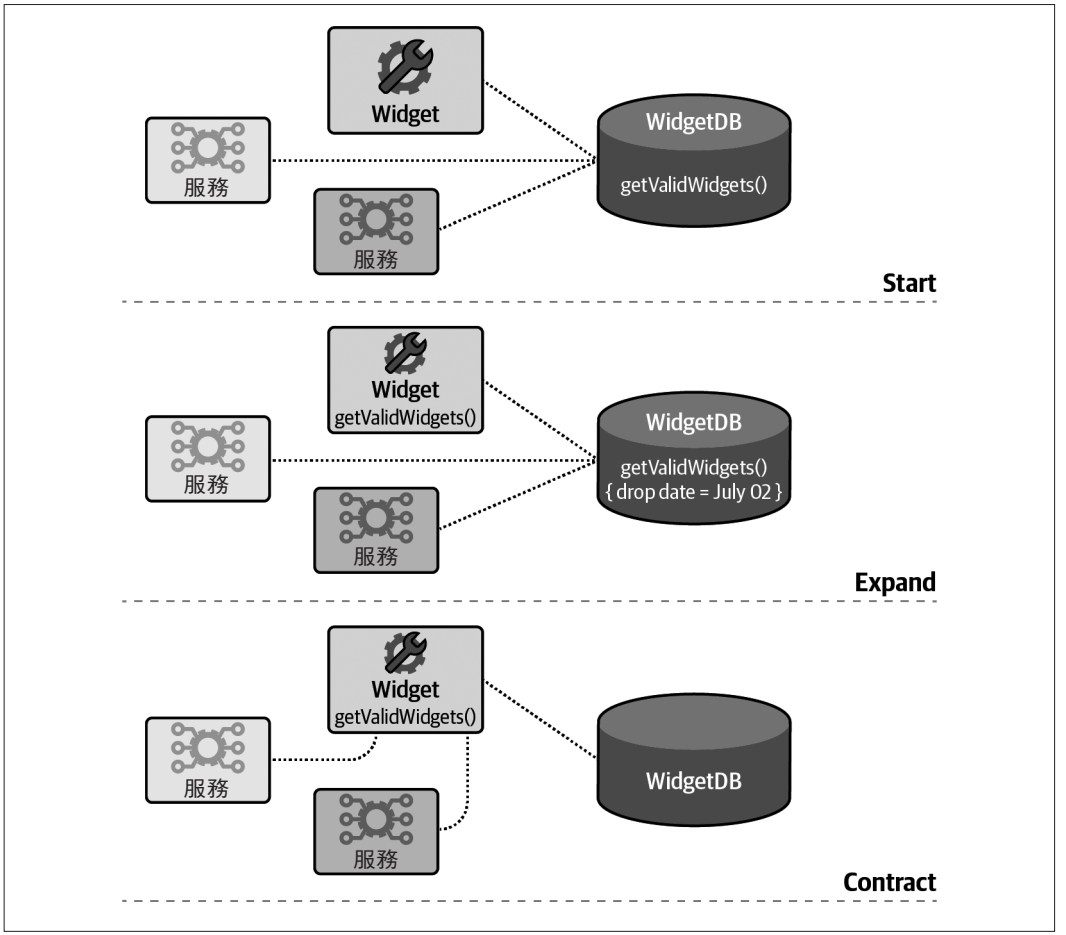

圖 6-11　將資料庫程式碼提取到服務中

在展開（Expand）階段，開發人員在 *Widgets Administration* 服務中新增替換方法，並重構其他服務以呼叫 *Widgets Administration* 服務。最初，新方法將作為預存程序的通道（pass-through），直到團隊能夠在經過良好測試的程式碼中無形地替換功能性為止。在此期間，應用程式支援對服務或預存程序的呼叫。在收縮（*Contract*）階段，架構師可以使用適應性函數來確保所有依存關係都已遷移為呼叫服務，並隨後捨棄預存程序。這是資料庫版的 Strangler Fig（絞殺者無花果樹）模式（*https://oreil.ly/BhDNV*）。