前言

本書的論點是，軟體系統大規模運行的能力，會日益成為定義成功的關鍵因素。隨著我們的世界變得更緊密相連，這種特徵只會變得更加普遍。因此，本書的目標是提供讀者分散式和併發系統的核心知識，也介紹了一些可用於建構可擴展系統的軟體架構方法和分散式技術。

為什麼是可擴展性？

我們世界變化的速度令人望而生畏，每天都有創新出現，為我們所有人創造了互動、開展業務、自我娛樂……甚至終結流行病的新能力。大部分創新的動力來自於軟體，是由主要網際網路公司名副其實的開發者大軍、初創公司的精幹小團隊、以及介於兩者之間各種形式和規模的團隊所撰寫。

交付能夠回應使用者需求的軟體系統已經夠困難了，但對於大規模系統來說，更是難上加難。我們都知道當系統暴露在意外的高負荷時會突然失效——這種情況（在最好的情況下）對公司來說更是負面的宣傳，在最壞的情況下甚至可能會導致失業或公司毀於一旦。

軟體與實體系統的不同在於它沒有固定的形狀——它實體的形式（1 和 0）與它實際的功能毫無任何相似之處。我們永遠不會期望在一夜之間將一個 500 人的小村落，轉變成 1,000 萬人口的城市，但我們有時候會期望我們的軟體系統突然間能夠處理比其設計時多一千倍的請求數量；無庸置疑的，這很少會有好的結果。

誰應該讀這本書

本書主要的目標讀者是對分散式併發系統毫無經驗、或經驗有限的軟體工程師和架構師，他們需要加深對理論和實務設計的知識，以因應建構更大規模、通常是面向網際網路的應用程式所面臨的挑戰。

從本書你會學到什麼

本書透過可擴展性的角度涵蓋了併發和分散式系統的概況，雖然不可能完全將可擴展性從其他架構品質分開，但可擴展性是討論的主要焦點。當然，其他的品質也必然會發揮作用，性能、可用性和一致性經常會在書中出現。

建構分散式系統需要對分散和併發有一些基本的了解——這些知識是貫穿本書且反覆出現的主題，這些知識的理解是必需的，因為分散式系統中存在的兩個基本問題讓它們變得複雜，如同以下所述。

首先，雖然整體系統幾乎一直能完美正確地運行，但系統中的個別部分隨時都可能會失效。當一個元件失效（無論是由於硬體當機、網路中斷、伺服器錯誤等），我們必須採用使整體系統能繼續運行且從失效中復原的技術。每個分散式系統都會經歷元件失效，通常會以奇怪、神秘和意想不到的方式出現。

其次，創建可擴展的分散式系統需要協調多個同時運作的元件，系統的每個元件都必須履行自己的職責，並且盡可能快地處理請求；只要有任何一個元件延遲，整體效能就可能下降，嚴重時甚至會造成當機。

有大量的文獻可以協助你處理這些問題。幸運的是對我們工程師而言，還有大量為了協助我們建構容錯和可擴展分散式系統而設計的技術，這些技術實現了各種理論方法、和非常難正確建構的複雜演算法。使用這些平台等級、廣泛應用的技術，我們的應用程式可以站在巨人的肩膀上，使我們能夠建構複雜的商業解決方案。

具體來說，本書的讀者將會學到：

- 分散式系統的基本特徵，包括了狀態管理、時間協調、併發、通訊和協調
- 用於建構可擴展、強健服務的架構方法和支援技術
- 分散式資料庫如何運行以及能夠用於建構可擴展的分散式系統
- 用於建構像是 Apache Kafka 和 Flink 等串流、基於事件系統的架構和技術

規模如何影響商務系統

1990 年代網際網路存取使用者的遽增為企業帶來了新的線上賺錢機會，企業急於透過網路瀏覽器向使用者展示它們商業上的功能（銷售、服務等）。這預示著我們對於如何建構系統的思考方式發生了強烈的變化。

以一家零售銀行為例子，在提供線上服務之前，要準確預測銀行商務系統負荷是可能的。我們知道有多少人在銀行中工作並使用內部的系統、有多少台終端機 / PC 連接到銀行的網路、必須要支援多少台 ATM，以及與其他金融機構連接的數量和性質。有了這些資料，我們就可以建構像是最多能同時支援 3,000 個使用者的系統，並且有把握不會超過這個數量。增長也會相對地緩慢，而且大部分時間（即在營業時間以外）的負荷遠低於尖峰時刻，這使我們的軟體設計決策和硬體配置變得更為容易。

現在，假設我們的零售銀行決定讓所有客戶都可以使用網路銀行，而且這個銀行有 500 萬個客戶，那現在最大的負荷是多少？在營業日的一天內負荷該如何分散？什麼時候是尖峰時刻？如果我們進行限時促銷以試圖吸引新客戶註冊，那會發生什麼情況？突然間，我們原本相對簡單和受限制的商務系統環境，因為網路使用者帶來的更高平均與尖峰負荷、以及其不可預測性，而被徹底打亂。

可擴展性基本設計原則

擴展系統的基本目的是增加它在某些應用程式特定維度上的能力。一個常見的維度是增加系統在給定期間內可以處理的請求量，這被稱為系統的吞吐量。讓我們用一個比喻來探討我們可用於擴展系統和增加吞吐量的兩個基本原則：複製和優化。

1932 年，世界代表性的工程奇蹟之一——Sydney Harbour Bridge（*https://oreil.ly/u7bOH*）正式啟用。現在，我們幾乎可以相當有把握地假設 2021 年的交通流量會比 1932 年高出一些。如果在過去 30 年中你有機會在尖峰時段開車通過這座橋，那麼你就會知道它每天都遠遠地超過了它的容量，那麼我們要如何增加像是橋梁這樣實體基礎設施的通行量呢？

這個問題於 1980 年代在 Sydney 變得非常明顯，當時大家意識到必須要增加海港渡口的容量。解決方案是不太具有代表性的 Sydney Harbout Tunnel（*https://oreil.ly/1VWm7*），它基本上沿著海港下方的相同路線建造。這方案額外提供了四條車道，因此增加了大約三分之

一海港渡口的容量。在不太遠的 Auckland，他們的海港大橋（*https://oreil.ly/E7yJz*）因為在 1959 年建造時只有四個車道，所以也存在容量的問題。本質上，他們採用了與 Sydney 相同增加容量的解決方案，但是他們不是建造隧道，而是巧妙地透過使用名為「Nippon clip-ons」（*https://oreil.ly/g7QBu*）的方法，把橋的兩側加寬，直接把車道數量翻倍。

這些例子說明了我們在軟體系統中增加能力的第一個策略。基本上我們複製了軟體處理資源，以提供更多的能力來處理請求，因而提高了吞吐量，如圖 1-1 所示。這些被複製的處理資源，就好比橋梁上的車道，為一連串到達的請求提供了大多彼此獨立的處理通道。

幸運的是，在基於雲端的軟體系統中，只需要單擊滑鼠就可以實現複製，我們甚至可以有效地複製我們的處理資源數千次。在這方面，我們比橋梁建設者要容易得多。我們的工作可以說比建造橋梁的工程師輕鬆許多。然而，我們仍然必須謹慎，確保複製資源是為了紓緩真正的瓶頸。如果在並未超載的處理路徑上增加容量，只會徒增成本，卻無法帶來任何可擴展性的效益。

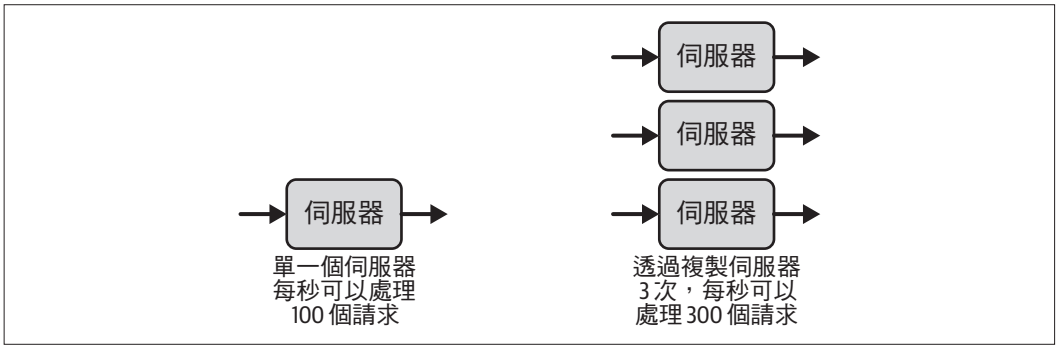

圖 1-1　透過複製增加能力

可擴展性的第二個策略也可以用我們橋梁的例子來說明。在 Sydney，一些細心的人意識到，早上從北向南過橋的車輛多了很多，而在下午我們看到了相反的模式。因此設計了一個聰明的解決方案——在早上將更多的車道分配給有高需求量的方向，而在下午的某個時候調換方向。這個方案在不需要分配任何新資源下，有效地增加了橋梁的容量——我們優化了已有的可用資源。

在軟體中我們也可以採用相同的方法來擴展我們的系統。如果我們可以透過使用更有效率的演算法、在資料庫中增加額外的索引來加快查詢速度，或甚至是用更快的程式語言重寫我們的伺服器等，以某種方式優化我們的處理，我們就可以在不增加資源的情況下增加我們的能力。這方面典型的例子是 Facebook 創建的（現在已經停產）HipHop for

我們永遠不會期望有人會嘗試將郊區住宅的容量，擴大成 50 層樓的辦公大樓。因為住宅不具備辦公大樓的結構、材料和地基，所以幾乎是不可能的，除非完全拆除和重建。同樣地，我們也不應該期望那些不採用可擴展的架構、機制和技術的軟體系統，能夠快速地演進以滿足更大的能力需求。規模的基礎需要從一開始就建立，並體會到元件會隨著時間的推移而演進。透過採用促進可擴展性的設計和開發原則，我們可以更迅速、更便宜地擴展系統以滿足快速增長的需求；我將在本書的 Part II 說明這些原則。

可以指數型擴展而成本是線性增長的軟體系統被稱為超大規模系統，我將其定義如下：「超大規模系統在計算和儲存能力上呈現出指數型的增長，同時在建構、運行、支援和演進所需要的軟體和硬體資源的成本上呈現出線性增長率」。你可以在這篇文章（*https://oreil.ly/WwHqX*）中讀到更多關於超大規模系統的訊息。

可擴展性和架構的權衡

可擴展性只是眾多品質屬性或非功能性需求之一，它是軟體架構領域的通用語言。軟體架構一個長期存在的複雜性是品質屬性權衡取捨的必要性。基本上，一個偏重某種品質屬性的設計，可能會正面或負面地影響其他品質屬性。例如，我們可能想要在我們的服務中發生某些事件時寫入日誌訊息，以便我們可以進行取證並支援程式碼除錯。然而，我們需要注意我們記錄了多少的事件，因為日誌記錄會引入開銷、並且對性能和成本產生負面的影響。

經驗豐富的軟體架構師總是小心翼翼地精心規劃他們的設計，以滿足高優先的品質屬性，同時將對其他品質屬性的負面影響最小化。

可擴展性也不例外，當我們將焦點放在系統的擴展能力上時，我們也必須仔細考慮我們的設計會如何影響其他非常想要的屬性，像是性能、可用性、安全性和經常被忽視的可管理性等，我將在接下來的章節中簡要地討論其中一些固有的權衡取捨。

性能

有一種簡單的方法可以思考性能和可擴展性之間的差異。當我們以性能為目標時，我們會試圖滿足個別請求的一些期望指標，這可能是少於 2 秒的平均反應時間，或是在最壞情況下的性能目標，像是 99 百分位的反應時間少於 3 秒等。

提升性能一般而言對可擴展性是一件好事，如果我們提升單個請求的性能，我們就能在系統中創造更多的能力，因為我們可以使用尚未用到的能力來處理更多的請求，所以這有助於提升我們的可擴展性。然而，事情並不總是那麼簡單。我們可以透過多種方式減

少反應時間，我們可能會仔細地優化我們的程式碼，例如刪除不必要的物件複製、使用更快的 JSON 序列化函數庫、甚至用更快的程式語言完全重寫程式碼等，這些方法可以在不增加資源使用的情況下優化性能。

另一種方法可能是藉由將經常存取的狀態保存在記憶體中，而不是在每個請求時寫入資料庫來優化單個請求，消除對資料庫的存取幾乎總是能加快速度，然而，如果我們的系統長時間在記憶體中維持大量的狀態資訊，我們可能（而且在高負荷的系統中必定會）需要仔細管理我們系統可以處理的請求數量。這可能會降低可擴展性，因為我們對單個請求的優化方法比原來的解決方案使用了更多的資源（在這情況下是記憶體），從而降低了系統的能力。

我們將在本書中看到性能和可擴展性之間的這種緊張關係一再出現。事實上，有時候讓個別請求稍微慢一些是明智的，這樣我們就可以利用額外的系統能力。當我在下一章討論負荷平衡時，將會描述一個很好的例子。

可用性

可用性和可擴展性通常是高度相容的伙伴。當我們透過複製資源來擴展我們的系統時，我們創建了多個服務的實例，可以用來處理來自任何使用者的請求。如果我們的一個實例失效，其他實例仍然可以使用。系統只會因為一個失效的、不能使用的資源，而使能力衰減。類似的想法也適用於複製網路鏈接、網路路由器、磁碟和計算系統中的幾乎所有資源。

當涉及到狀態時，事情會因可擴展性和可用性而變得複雜。想像一個資料庫，如果我們的單個資料庫伺服器過載，我們可以複製它並發送請求到任何一個實例。因為我們可以容忍一個實例的失效，所以這也增加了可用性。如果我們的資料庫是唯讀的，這個模式會運作得很好；但是一旦我們更新了一個實例，我們就必須弄清楚如何以及何時更新另一個實例，這就是複製一致性問題發生的地方。

事實上，每當為了可擴展性和可用性而複製狀態時，我們都必須處理一致性。這將會是本書 Part III 討論分散式資料庫時的重點主題。

安全性

安全性是一個複雜的、高度技術性的主題，值得單獨寫一本書。沒有人願意使用不安全的系統，而受到駭客攻擊和洩露使用者資料的系統會導致 CTO 離職，在極端情況下公司甚至會倒閉。

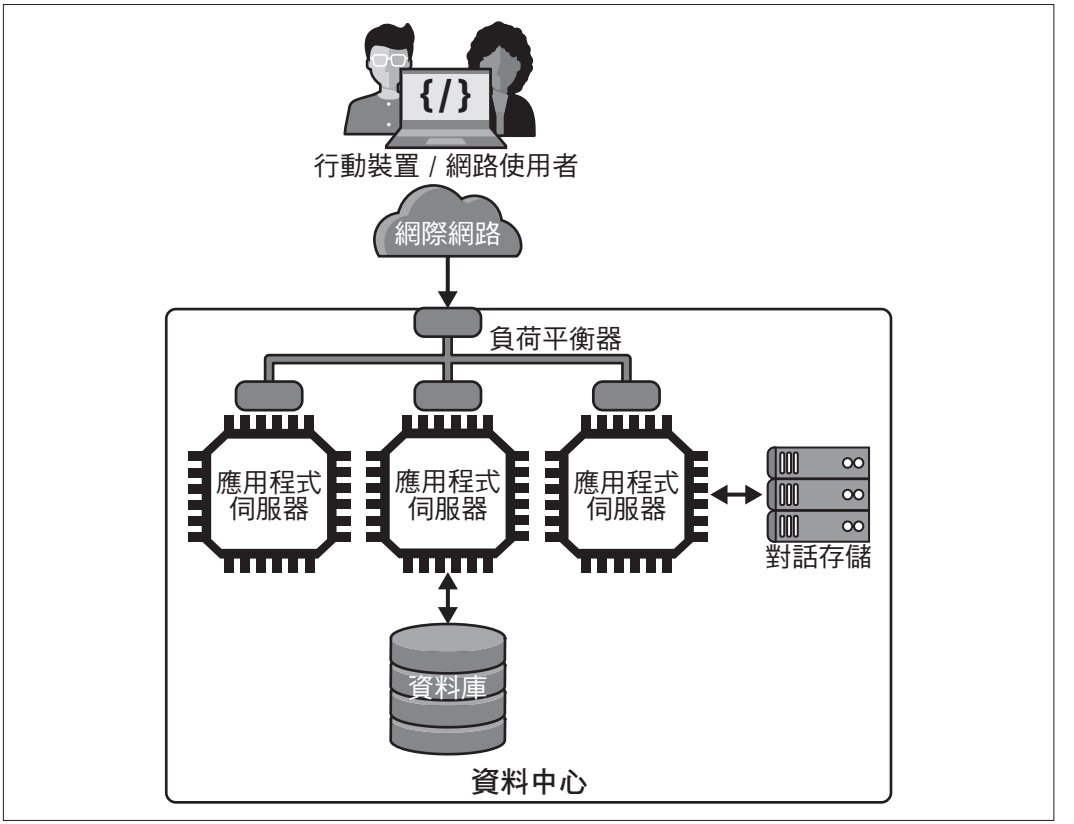

圖 2-2　向外擴大架構

用快取擴展資料庫

透過增加資料庫伺服器中的 CPU、記憶體和磁碟的數量來向外擴大，對於擴展系統有很大的作用。例如，在撰寫本文的時候，GCP 可以在 db-n1-highmem-96 節點上提供 SQL 資料庫，這個節點有 96 個虛擬 CPU（vCPU）、624 GB 的記憶體、30 TB 的磁碟，並且可以支援 4,000 個連接。這每年將花費 6,000 美元到 16,000 美元，這對我來說聽起來很划算！擴大是一種常見的資料庫可擴展性策略。

大型資料庫需要經驗豐富的資料庫管理者持續的維護和調校，以保持它們的調諧和快速執行。這項工作中有很多技巧——例如，查詢調校、磁碟分區、索引、節點快取等等——因此資料庫管理者是你想要善待的重要人物，因為他們可以使你的應用服務具有高度的反應性。

圖 2-4 顯示了我們的架構如何併入一個分散式資料庫。當資料量增長，分散式資料庫可以增加存儲節點的數量；當節點增加（或移除）時，跨所有節點管理的資料將重新平衡，以試圖確保每個節點的處理和儲存能力被均等的利用。

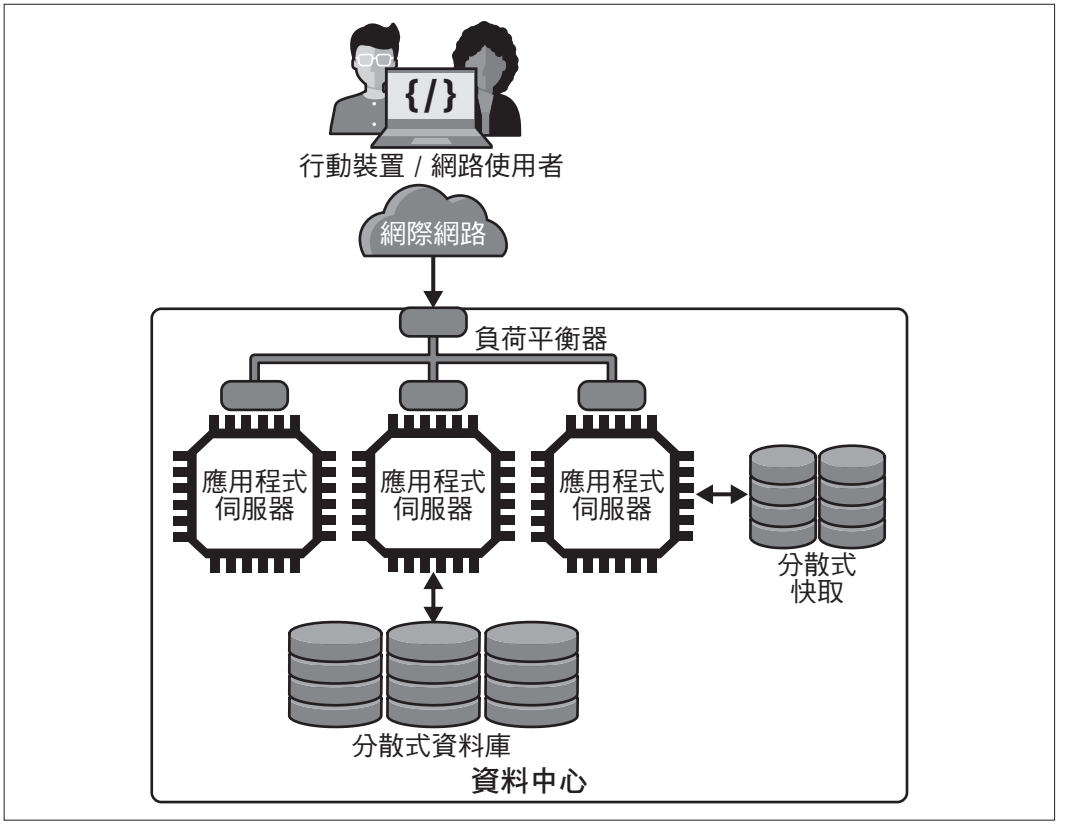

圖 2-4　用分散式資料庫擴展資料層

戶端提供服務和為行動客戶端提供服務，每個服務都可以根據它們所經歷的負荷獨立擴展，它通常被稱作為前端而開發的後端（BFF）模式[4]。

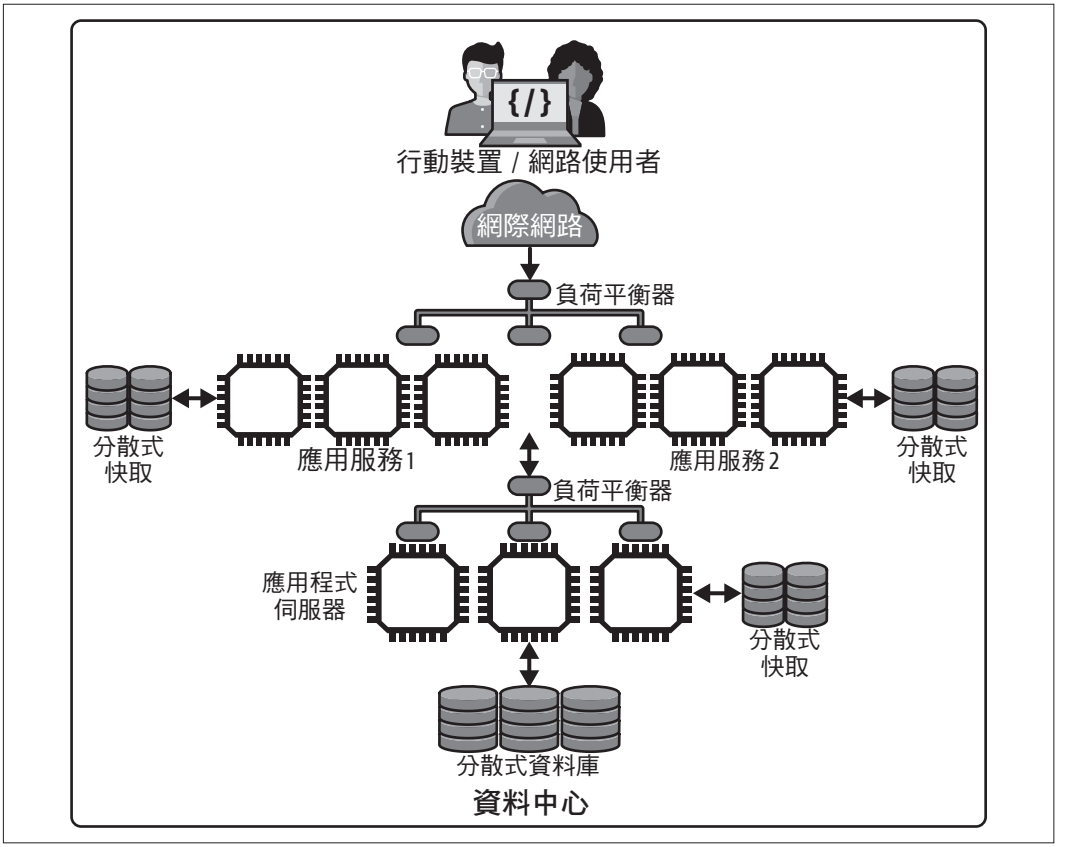

圖 2-6　有多種服務的可擴展架構

另外，透過將應用程式分解成多個獨立的服務，你可以根據服務的需求擴展每個服務。例如，如果你看到來自行動使用者的請求量增加、而來自網路使用者的請求量減少，則可以為每種服務提供不同數量的實例以滿足需求。這是將整體式應用程式重構為多個獨立服務的主要優勢，這些服務可以單獨地建構、測試、部署和擴展。我將在第 9 章探討設計此類以服務為基礎的系統（稱為微服務）時所面臨的一些主要問題。

4　Sam Newman，「Pattern: Backends For Frontends」，Sam Newman & Associates，2015 年 11 月 18 日，*https://oreil.ly/1KR1z*。

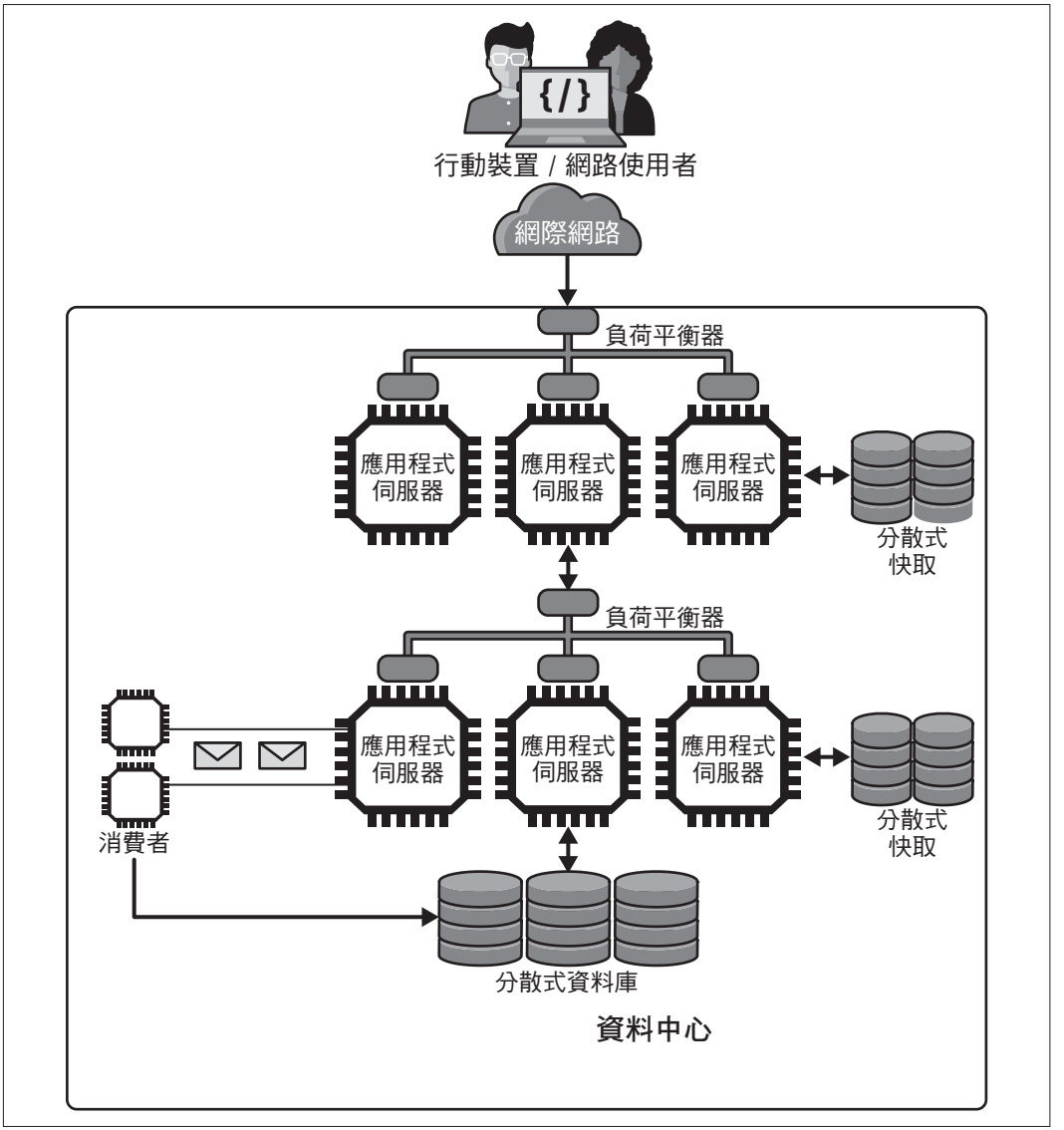

圖 2-7　用佇列增加反應能力

網路快取

網站回應速度如此快的原因之一，是因為網際網路上布滿了網路快取。網路快取會在定義的期間內，保存特定資源的副本，例如一個網頁或一張圖片。快取會攔截客戶端請求，若本地已有請求的資源快取，就直接回傳該副本，而不是將請求轉發到目標服務。因此，可以在不帶給服務負擔下滿足許多請求。另外，由於快取在實體上更接近客戶端，所以請求將有較低的延遲。

圖 6-2 展示了網路快取架構的概覽。存在多個層次的快取，從客戶端的網路瀏覽器快取和基於本地組織的快取開始，ISP 也會實現一般的網路代理快取，而且反向代理快取可以部署在應用程式服務執行領域內。網路瀏覽器快取也被稱為私有快取（針對單一使用者），組織的和 ISP 代理快取是支援來自多個使用者請求的共享快取。

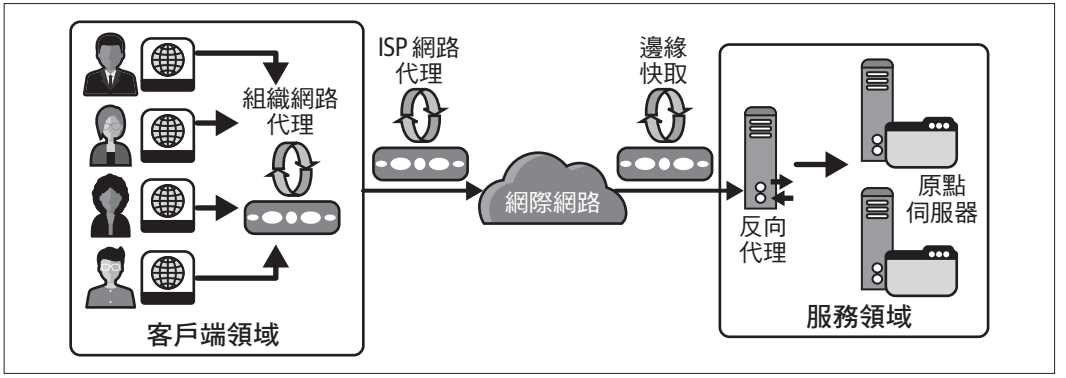

圖 6-2　網際網路中的網路快取

邊緣快取，也稱為內容傳遞網路（CDN），部署在全球各個策略性地理位置，使經常被存取的資料能在更接近客戶端的地方進行快取。例如，一家影音串流供應商可能會在澳洲雪梨設置邊緣快取，向大洋洲用戶提供影片內容，而不是從位於美國的來源伺服器跨越太平洋進行串流傳輸。邊緣快取由 CDN 供應商在全球部署，最初的 CDN 供應商 Akamai，擁有超過 2,000 個地點，並傳遞全球高達 30% 的網際網路流量（*https://oreil.ly/s73lC*）。對於擁有全球使用者的富媒體網站，邊緣快取是必不可少的。

快取通常儲存 `GET` 請求的結果，且快取鍵是關聯到 `GET` 的 URI。當客戶端發送 `GET` 請求時，它可能會被請求路徑上的一個或多個快取攔截。任何具有所請求資源的新副本的快取都可以回應這個請求。如果未找到快取的內容，這請求將由服務端點提供服務，它在網路技術術語中也稱為原點伺服器。

第七章

異步訊息傳遞

對於一本關於分散式系統的書來說，討論通訊議題幾乎是不可避免的。在前幾章中，我已經花了相當多的篇幅探討通訊相關問題。通訊是分散式系統的基礎，也是系統架構師在設計系統時必須納入的重要課題。

到目前為止，這些討論都假定採用同步訊息傳遞的方式，客戶端發送一個請求並等待伺服器的回應。這就是大多數分散式通訊設計的方式，因為客戶端需要即時回應以繼續進行。

不是所有系統都有這個要求。例如，當我要退回一些我在線上購買的物品時，我會將它們帶到我當地的 UPS 或 FedEx 門市。他們掃描我的 QR code，然後我將包裹交給他們處理。我不會待在店裡等待廠商確認已經成功收到商品並退還我的款項，這麼做既無聊又沒效率。我相信運輸服務會將我不需要的物品送到賣家手中，並預期在幾天後處理完成時收到通知。

我們可以使用異步通訊方式設計我們的分散式系統來模擬這種行為，稱為生產者的客戶端將他們的請求發送到中間的訊息服務，它充當一種傳遞機制，將請求轉發到稱為消費者的預期目的地處理，生產者對他們發送的請求「發送後不理」。請求一旦傳遞給訊息服務，生產者會移到他們邏輯中的下一步，確信它發送的請求最後會得到處理。因為生產者不需要等到請求處理完成，所以提高了系統的回應能力。

在這一章，我將描述異步訊息系統支援的基本通訊機制。我也將討論吞吐量和資料安全之間固有的權衡取捨——基本上，確保你的系統不會遺失訊息。我還會涉及到通常部署在高度可擴展分散式系統中的三種關鍵訊息傳遞模式。

表 7-1　交換機類型

交換機類型	訊息傳送行為
直接	根據與每條訊息一起發布的路由選擇鍵的值匹配，將訊息傳遞到佇列
主題	根據匹配的路由選擇鍵、和用於將佇列綁定到交換機的模式，將訊息傳遞到一個或多個佇列
扇出	傳遞訊息給綁定到交換機的所有佇列，並忽略路由選擇鍵

直接交換機通常用於根據匹配的路由選擇鍵，將每條訊息傳遞到一個目標佇列[4]。主題交換機是根據模式匹配而更靈活的機制，可用於實現複雜的發布 - 訂閱訊息傳遞拓撲。扇出交換機提供了一種簡單的一對多廣播機制，其中每條訊息都被發送到所有附隨的佇列。

圖 7-5 描述了直接交換機是如何運作的，佇列被消費者用三個值綁定到交換機，這三個值分別是「France」、「Spain」和「Portugal」。當訊息從發布者到達時，交換機使用附隨的路由選擇鍵將訊息傳遞到三個附隨的佇列之一。

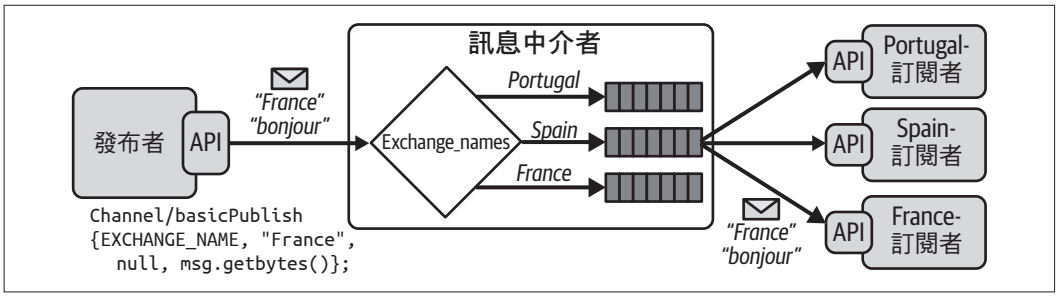

圖 7-5　RabbitMQ 直接交換機範例

以下程式碼展示了在 Java 中如何配置和使用直接交換機的片段。RabbitMQ 客戶端，即生產者和消費者的過程，使用通道抽象化來與中介者建立通訊（下一節有更多關於通道的內容）。生產者在中介者中建立交換機，並將路由選擇鍵設定為「France」對交換機發布訊息。一個消費者在中介者建立一個匿名的佇列，將佇列綁定到由發布者建立的交換機，並指定用路由選擇鍵「France」發布的訊息應該被傳送到這個佇列。

4　消費者可以多次呼叫 queueBind()，以指定他們的目的地應該接收一個以上的路由選擇鍵值的訊息，這種方法可用於建立一對多的訊息分發；主題交換機對於一對多的訊息傳遞更為強大。

生產者：

```
channel.exchangeDeclare(EXCHANGE_NAME, "direct");
channel.basicPublish(EXCHANGE_NAME, "France", null, message.getBytes());
```

消費者：

```
String queueName = channel.queueDeclare().getQueue();
channel.queueBind(queueName, EXCHANGE_NAME, "France");
```

分散和併發

若要在性能與可擴展性方面完全徹底發揮 RabbitMQ 的優勢，你必須了解該平台在底層的運作方式。相關的重要議題包括客戶端和中介者如何進行通訊，以及執行緒的管理方式。

每個 RabbitMQ 客戶端使用 RabbitMQ 連線連接到中介者，這基本上是 TCP/IP 上的抽象化，而且可以用用戶憑據或 TLS 保護。建立連線是一個重量級的操作，需要在客戶端和伺服器之間多次往返，因此每個客戶端保持一個單一的長期連線是常見的使用模式。

為了發送或接收訊息，客戶端使用連線建立 RabbitMQ 通道。通道是客戶端和中介者之間的邏輯連線，而且只存在於 RabbitMQ 連線的環境下，如以下程式碼片段所示：

```
ConnectionFactory connFactory = new ConnectionFactory();
Connection rmqConn = connFactory.createConnection();
Channel channel = rmqConn.createChannel();
```

在同一個客戶端可以建立多個通道，以建立多個邏輯中介者連線。在這些通道上的所有通訊都在相同的 RabbitMQ（TCP）連線上多路傳遞，如圖 7-6 所示。建立一個通道需要一次與中介者的網路往返。因此，基於性能考量，通道在理想情況下應該保持長時間存續，並且應避免頻繁地建立與銷毀通道，也就是所謂的通道攪動。

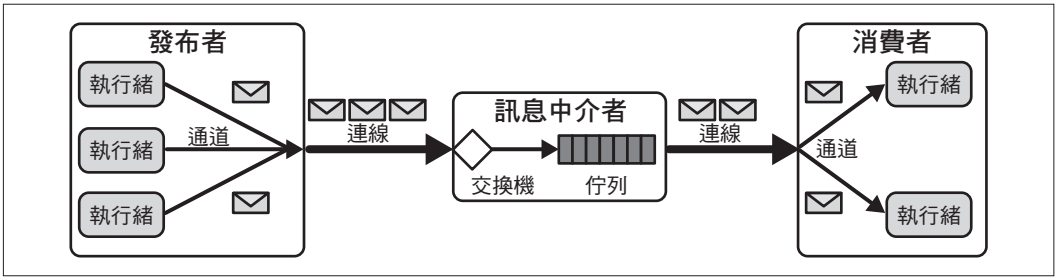

圖 7-6　RabbitMQ 連接和通道

拆散整體式應用程式

微服務架構將應用程式功能分解成多個獨立的服務，這些服務在必要時會進行通訊和協調。圖 9-2 顯示如何用微服務設計圖 9-1 中的大學管理系統，每個微服務都是完全獨立的，在需要的地方封裝自己的資料存儲，並提供通訊的 API。

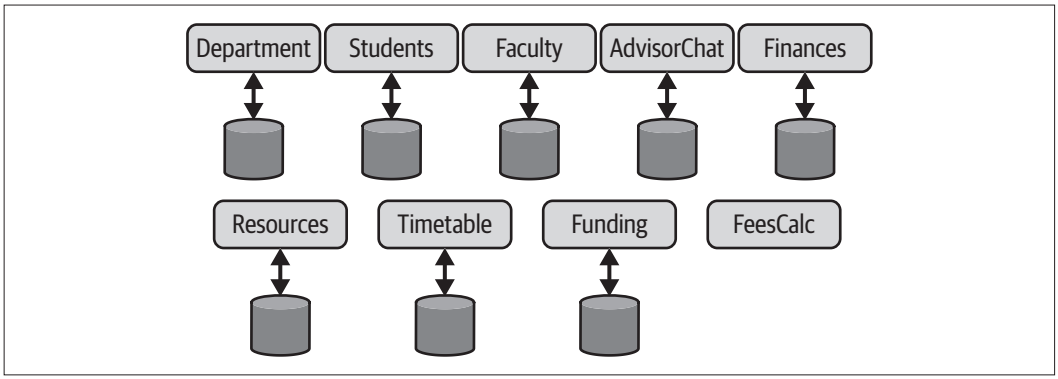

圖 9-2　微服務架構的例子

隨著系統的程式碼大小和請求負荷增長，微服務提供了以下優勢：

程式碼庫

依據兩個披薩規則，個別服務的複雜性不應超過小型團隊可以建構、發展和管理的程度。因為微服務是一個黑盒子，團隊有充分的自主權來選擇自己的開發堆疊和資料管理平台[1]。考量到一個設計良好的微服務，其功能範疇較小且高度內聚，這通常會帶來較低的程式碼複雜度，以及在新增功能時更快的開發節奏。另外，當需要時也可以獨立地部署微服務的修訂版。如果微服務支援的 API 是穩定的，那這種改變對依賴的服務是透明的。

向外擴大

個別的微服務可以向外擴大以滿足請求量和延遲要求。例如，為了滿足不斷要求和聊天的學生，`AdvisorChat` 微服務可以根據需要在它自己的負荷平衡器後面複製，以提供較短的回應時間，如圖 9-3 所示。其他經歷輕負荷的服務可以簡單地在單一節點上執行，或在低成本下複製以消除單點故障並增強可用性。

1　如 Susan Fowler 在《*Production-Ready Microservices*》（O'Reilly，2016 年）中所說明的，跨微服務的開發堆疊標準化當然具有好處。