第一章

可靠、可擴展與可維護的應用系統

> 網際網路發展得太好了，以至很多人把它當成像是太平洋一樣的自然資源，而非人造的。還記得上一次達到如此規模而又強壯的科技是什麼時候呢？
>
> —Alan Kay, 於 *Dr. Dobb Journal* 的採訪（2012）

今天的許多應用都是屬於**資料密集型**（*data-intensive*），而不是**計算密集型**（*compute-intensive*）。對於這些應用，CPU 本身的處理能力通常不會是限制因素，資料量級、資料複雜度以及資料的速變性才是問題所在。

資料密集型應用通常是由一些常用功能的標準模組所建構而成。例如，許多應用都有以下需求：

- **資料庫**（*databases*）：儲存資料，以便它們或另一個應用可以再次訪問
- **快取記憶體**（*caches*）：將昂貴操作的結果暫存起來，使讀取操作得以加速
- **搜尋索引**（*search indexes*）：使用者可以通過關鍵字來搜尋資料或支援各種過濾資料的方式
- **串流處理**（*stream processing*）：將訊息發送至另一個程序，且以非同步方式進行處理
- **批次處理**（*batch processing*）：週期性地處理大量累積的資料

上述這些需求看起來似乎再平常不過了，那是因為這些**資料處理系統**（*data systems*）有著很成功的抽象：讓我們拿來就用，不用想太多。當工程師需要建構一個全新應用時，多數人應該都不希望一切從頭開始、重新寫一個全新的資料儲存引擎吧，因為現成的資料庫方案已經可以很好地滿足我們的需求。

不過，現實並不總是那麼簡單。因為不同的資料庫各有所長，往往也是為了滿足各種應用的不同需求，而有各式各樣的快取方法以及多種不同的索引方法等等。當需求無法被單一工具獨立滿足時，要整合運用多種工具也是一項大工程。因此在建構應用時，我們仍需要弄清楚哪些工具和方法最適合手頭上的任務。

本書不只介紹原理，同時也闡述資料系統的實踐面向，如何使用它們來建構資料密集型應用。我們將探索不同工具的共通之處與差異所在，以們它們如何實現各自的特性。

本章首先探討實踐的基礎：可靠、可擴展與可維護的資料系統，闡述它們的含義以及思考之道，並且走一遍後續章節所需要的基礎知識。接下來我們將層層推進，看看在處理資料密集型應用時需要考量的不同設計決策。

資料系統思維

我們通常認為資料庫、佇列、快取記憶體等是不同類型的工具。儘管資料庫和訊息佇列在表面上有些相似性（例如兩者都會儲存資料一段時間），卻有著不同的存取模式，這說明了它們有不同的性能特徵，因此也有著截然不同的實作。

那麼，為什麼我們要將它們歸在「**資料系統**」這個大項底下呢？

近年出現了許多新的資料儲存和處理的工具。它們針對各種不同的用例進行了最佳化，所以無法再用傳統的方式加以歸類 [1]。例如，有些資料儲存也用作訊息佇列（Redis），而有些訊息佇列也具備類似資料庫的持久化儲存保證（Apache Kafka）。這些資料系統之間的分野界線日漸模糊。

其次，現今越來越多的應用有更高的要求和更廣泛的需求，以致單一工具往往無法滿足應用對所有資料的處理與儲存需求。取而代之的是，工作被進一步分解，分解成一個個可以由個別單一工具所高效執行的任務，而這些工具的組合運用則是通過應用層程式碼來進一步縫合。

例如，如果有某個應用程序所管理的快取層（使用 Memcached 或類似者），或是與主資料庫分離的全文檢索伺服器（像 Elasticsearch 或 Solr），那麼通常會由應用層程式碼負責保持這些快取和索引與主資料庫的同步。圖 1-1 給出了它可能的樣子，詳細技術將於後面的章節中再做介紹。

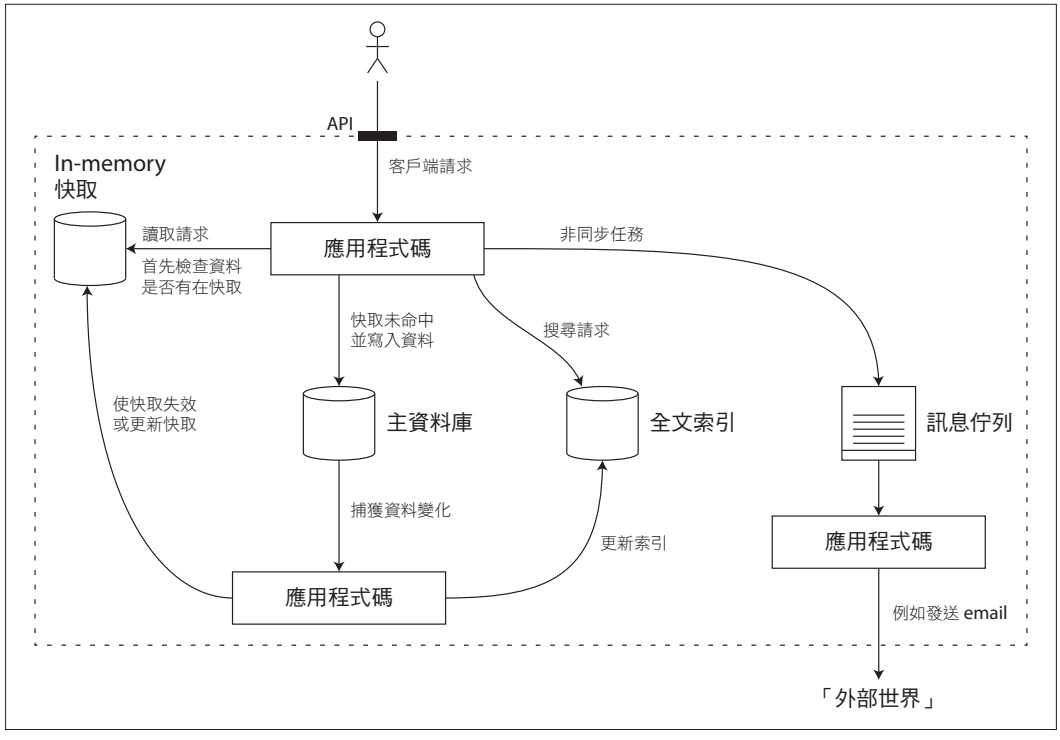

圖 1-1　一種組合了多個元件的資料系統架構

當服務是由多個元件組合建構而成時，服務的介面或應用程式介面（API）通常會對客戶端隱藏內部的實作細節。現在，假設你已經用一些較小的通用元件建構出一個全新的專用資料系統。這個整合起來的資料系統可能會提供某些保證，例如確保快取失效或者在寫入時正確刷新，好讓外部客戶端能看到一致的結果。這樣的你，已不只是一位應用開發者，同時也是一位資料系統設計師。

在設計資料系統或服務時，會出現很多棘手的問題。當系統內部出錯時，如何確保資料的正確性與完整性？當系統某些部分發生降級時，該如何向客戶提供良好如一的性能？系統應該如何擴展以因應增加的負載？一個具備友善 API 的服務又應該如何設計？

第二章

資料模型與查詢語言

> 語言之所束，世界之所限。
>
> —Ludwig Wittgenstein, 邏輯哲學論（1922）

資料模型可能是軟體開發當中最重要的部分，因為它們的影響深遠：不僅影響軟體的編寫方式，還有我們**對於待解問題的思考方式**。

大多數應用程式是通過不同的資料模型層層疊加所建構而成的。對於每一層，關鍵問題在於：它是如何以低它一層的資料模型來**表現**的？例如：

1. 身為一名應用程式開發者，觀察現實世界（其中有人員、組織、商品、行為、資金流動、感測器等），並根據物件、資料結構以及操作這些資料結構的 API 來進行建模。資料的結構往往是由具體的應用所決定。

2. 當需要儲存這些資料結構時，可採通用的資料模型來表示它們，例如 JSON 或 XML 文件（documents）、關聯式資料庫的表（tables）、或圖模型（graph model）。

3. 建構資料庫的工程師會決定採用 JSON/XML/relational/graph 資料來表示來自記憶體、磁碟或網路上的位元數據，因為具象化的資料可以更容易地支援查詢、檢索、操作等處理。

4. 在更底層，硬體工程師會有一些表示位元數據的方法，例如電流、光脈衝或磁場等等。

複雜的應用程式可能有更多的中間層，例如建構在某些 API 之上的 API。然而，基本思想是一樣的：每一層都通過提供一個簡潔的資料模型來隱藏其下各層的複雜性。這些抽象讓不同的團隊（例如資料庫供應商、應用開發者）得以有效地合力工作。

物件 - 關聯的不匹配

現今大多數的應用系統多採物件導向的程式語言來加以開發設計，這就引發了一個對 SQL 資料模型的批評：如果資料儲存在其中，那麼應用層的物件和資料庫模型（tables、rows 與 columns）之間就勢必需要一個笨拙的轉換層。模型間的關係斷連（disconnect）有時候稱為**阻抗不匹配**（*impedance mismatch*）[1]。

像 ActiveRecord 和 Hibernate 這種物件 - 關聯映射（Object-relational mapping, ORM）框架減少了轉換層所需要的模板程式碼（boilerplate code），但是它們還是不能完全隱藏兩個模型之間的差異。

例如，圖 2-1 展示了如何以 relational schema 表示一份履歷（一份 LinkedIn profile）。整份 profile 可以通過一個唯一的識別符 `user_id` 來標識。像 `first_name` 和 `last_name` 這種欄位在每個使用者的資料中只會出現一次，因此可以建模為 `users` table 中的行（columns）。然而，大多數人的職業生涯（職位）可能不只從事過一份工作，而且每個人的教育歷程也不盡相同，就連聯絡資訊恐怕也可能有好幾種。使用者與這些項目（items）之間存在著一對多（one-to-many）的關係，這可以用不同的方式加以表示：

- 傳統的 SQL 模型（SQL: 1999 以前），最常見的正規化表示是將職位 `positions`、教育程度 `education` 和聯絡資訊 `contact_info` 單獨放在各自的 table 中，其中帶有對 `users` table 的外鍵（foreign key）參照，如圖 2-1 所示。
- SQL 標準的後續版本增加了對結構化資料類型和 XML 資料的支援；這讓多值資料（multi-valued data）得以儲存在單一列中，並支援在這些 documents 中進行查詢和索引。Oracle、IBM DB2、MS SQL Server 和 PostgreSQL [6,7] 都在不同程度上支援了這些功能。一些資料庫也支援 JSON 資料類型，包括 IBMDB2、MySQL 和 PostgreSQL [8]。
- 第三個選項是將工作、教育和聯絡資訊進一步編成 JSON 或 XML document，將其用資料庫的文本行（text column）儲存下來，讓應用程式自行去解釋其結構和內容。不過，這種方式通常就沒辦法使用資料庫來直接查詢那些資料行中的值了。

1 從電子學借來的術語。每個電路的輸入和輸出都有一定的阻抗（對交流電流的抵抗能力）。當你把一個電路的輸出連接到另一個電路的輸入時，如果兩個電路的輸出阻抗和輸入阻抗是匹配的話，這樣就能使電路之間的功率傳輸得到最大化。阻抗不匹配會引起訊號反射和其他問題。

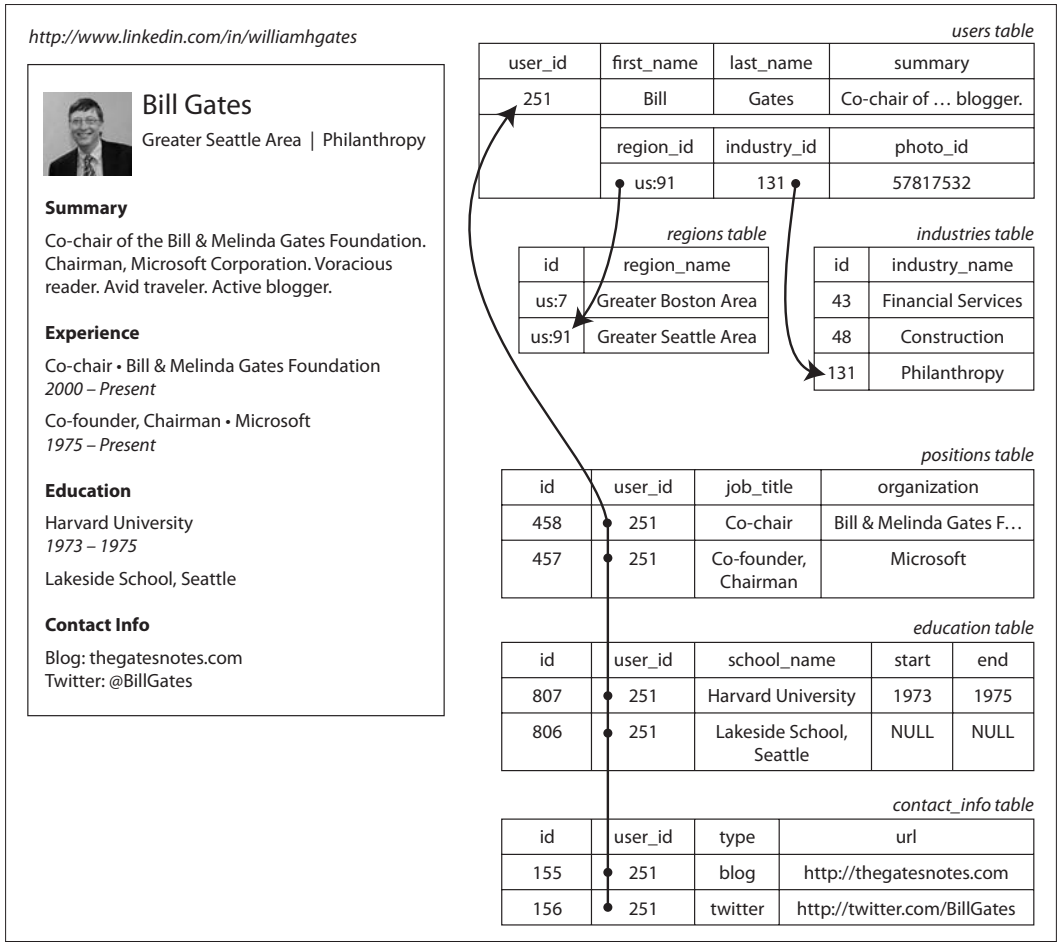

圖 2-1　使用 relational schema 來表示一份 LinkedIn profile（Bill Gates 的照片由 Wikimedia Commons 提供，來自 Ricardo Stuckert, Agência Brasil。）

對於類似履歷的資料結構，大部分都是一份獨立**文件**（self-contained *document*），因此很適合用 JSON 格式來儲存，參見範例 2-1。JSON 吸引人的地方在於，它比 XML 要簡單得多。文件導向的資料庫（document-oriented databases）如 MongoDB [9]、RethinkDB [10]、CouchDB [11] 和 Espresso [12] 都支援 JSON。

第三章

資料儲存與檢索

秩序讓你毫不費力。

—德國諺語

資料庫基本要做兩件事情：當你給它一些資料，它就應該儲存資料；稍後再次請求這些資料時，它應該將其傳回給你。

我們在第 2 章討論了資料模型和查詢語言，前者是應用程式開發者提供給資料庫的資料格式，後者是向資料庫查詢資料的機制。本章將從資料庫的角度來探討同樣的問題：如何儲存給定的資料，以及如何在需要這些資料的時候重新找到它們。

作為應用開發者，為什麼要關心資料庫內部是如何處理儲存和檢索的呢？因為多數人不太可能從頭開始實作自己的儲存引擎，取而代之的是從眾多的儲存引擎中選擇一個合適的來使用。不過，為了能夠針對應用的工作負載來對儲存引擎進行調校，你就得大概了解儲存引擎在幕後的工作原理才行。

我們的起手式是先討論儲存引擎，這些引擎用於你或許熟悉的兩種資料庫：傳統的關聯式資料庫和所謂的 NoSQL 資料庫。本章將研究兩大儲存引擎家族：**日誌結構**（*log-structured*）的儲存引擎和**分頁導向**（*page-oriented*）的儲存引擎，例如 B-trees。

雜湊索引

讓我們從建立 key-value data 的索引開始。Key-value data 很常見，它並不是可以索引的唯一資料類型，而且對於複雜度更高的索引來說，它也是一個很好用的建構方塊。

Key-value stores 與大多數程式語言中的**字典**（*dictionary*）型別非常類似，通常是以雜湊映射（hash map，或稱雜湊表 hash table）來實作。很多演算法教科書對 hash map 都有詳細介紹 [1, 2]，在此略過其原理細節。既然 in-memory 的資料結構都用了 hash map，為什麼不拿它們來索引磁碟上的資料呢？

假設我們的 data storage 就如前面的例子那樣，寫入內容的唯一方式就是追加進檔案。那麼，最簡單的一個索引策略是：在記憶體中保存一個 hash map，其中每個 key 都映射到資料檔案中的位元組偏移量（byte offset），此偏移量正是在檔案中可以找到其 value 的地方，圖 3-1 說明了這個概念。每當向檔案追加一個新的 key-value pair 時，還需要更新 hash map 來反映剛剛寫入資料的偏移量（包括插入新鍵和更新已存在的鍵）。當想要查找某筆資料時，便是通過 hash map 找到它在檔案中的偏移量，求得儲存位置，然後讀取其內容。

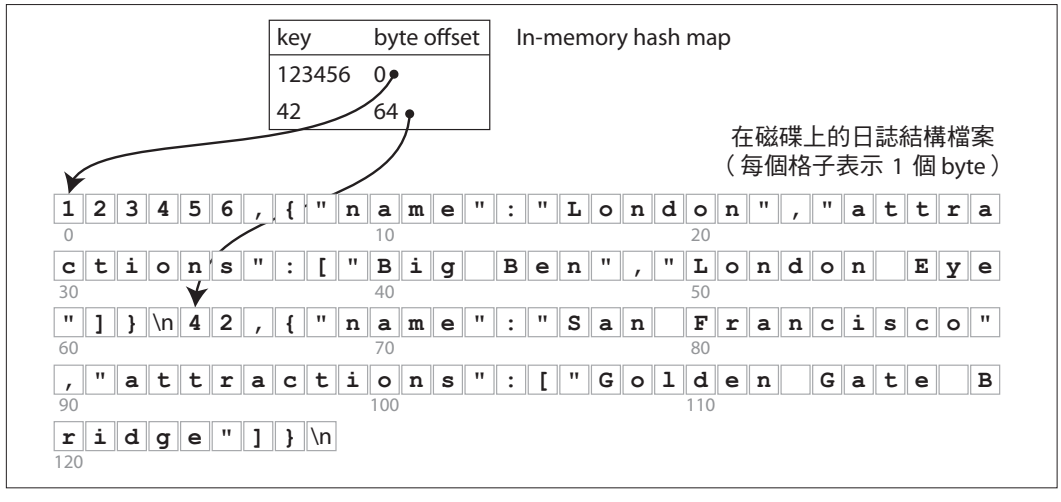

圖 3-1　以 CSV-like 格式儲存一筆 key-value data，並使用 in-memory hash map 作索引

這聽起來可能太過簡單，但確實是一種可行的方法。事實上，這就是 Bitcask（Riak 預設的儲存引擎）所採用的做法 [3]。因為 hash map 需要存在記憶體中，只要所有的 keys 都能放進記憶體，Bitcask 就能提供高性能的讀和寫。對於 values，它們不必全塞進記憶

第四章

資料編碼與演化

> 滄海復成桑田，變化從未止歇。
>
> —Heraclitus of Ephesus, 由柏拉圖在 *Cratylus* 中引用（西元前 360 年）

隨著時間推移，應用程式不可避免地會發生變化。隨著新產品推出、對使用者需求更好的理解、或業務環境變化時，系統也需要跟著增加或修改功能。第 1 章我們介紹了**可演化性**的概念，目標是建構出可以輕鬆適應變化的系統（參閱第 21 頁的「可演化性：易於求變」）。

大多數情況下，對應用程式功能的更改也會同時伴隨修改相應儲存資料的需求：可能需要增加新欄位或記錄類型，或需要以新的方式來呈現現有資料。

我們在第 2 章所討論的資料模型有不同的方法來應對這種變化。關聯式資料庫通常假設資料庫中的所有資料都符合某種 schema，雖然該 schema 可以更改（透過 schema migrations；即，`ALTER` 語句），但於任何時間點下都只有一個 schema 有效。相比之下，schema-on-read（"schemaless"）的資料庫並不強制使用 schema，所以資料庫可以包含不同時間點寫入的新舊資料格式（參閱第 2 章「文件模型中的基模靈活性」）。

當資料格式或 schema 發生變化時，通常需要對應用程式碼進行相應的調整（例如記錄增加了新欄位，應用程式欲讀寫該欄位）。然而，在大型的應用系統中，時常修改程式碼往往不能被忍受：

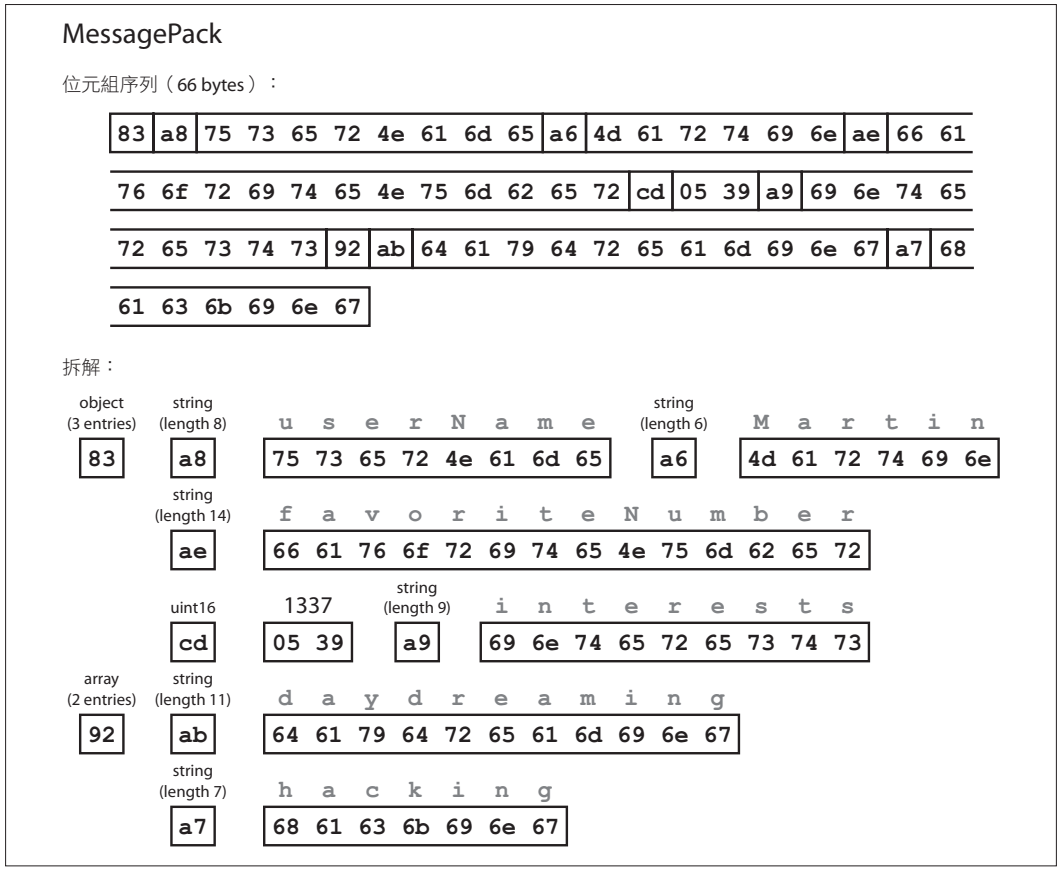

圖 4-1　使用 MessagePack 編碼後的示範記錄（範例 4-1）

Thrift 與 Protocol Buffers

Apache Thrift [15] 與 Protocol Buffers（protobuf）[16] 兩者都是基於相同原理的二進位編碼函式庫。Protocol Buffers 最初是 Google 開發的，而 Thrift 則是 Facebook 開發的，兩者都在 2007~2008 年左右開源出來 [17]。

Thrift 和 Protocol Buffers 對資料的編碼需要配合 schema。如果要用 Thrift 對範例 4-1 中的資料進行編碼，可以用 Thrift 的介面定義語言（interface definition language, IDL）來描述 schema，如下：

第五章

複製副本

> 事情可能出錯與不可能出錯的差別在於，不可出錯的事情一旦出錯，就意味著事情無法挽回了。
>
> —Douglas Adams, 基本無害（1992）

複製（*Replication*）是指在網路中的多台機器上保存相同資料的副本。正如第二部分開頭所說，需要複製資料的原因有以下幾個：

- 讓資料在地理位置上靠近使用者，進而減少存取延遲。
- 讓系統在某部分失效的情況下繼續工作，進而提升可用性。
- 擴大提供讀取查詢的機器數量，進而增加讀取的吞吐量。

在本章，我們會以「dataset 夠小」的前提來進行討論，這樣就能假設每台機器都有足夠的容量可以保存完整的 dataset 副本。之後在第 6 章會放寬這個假設，當 dataset 大到無法由單台機器儲存時，如何對 dataset 做**分區**（*partitioning*），或稱為**分片**（*sharding*）。後面的章節還會討論複製資料系統可能出現的各種故障，以及如何處理它們。

如果複製的資料不會隨時間變化，只需將資料複製到每個節點一次，事情很容易就完成了。Replication 最困難的地方其實是處理會變化的複製資料，這正是本章要討論的重點。我們會討論三種在節點之間複製**變動**資料的演算法：*single-leader* replication（**單領導**複製）、*multi-leader* replication（**多領導**複製）和 *leaderless* replication（**無領導**複製）。幾乎所有分散式資料庫都採用了上述方法的其中一種。它們各有優缺點，稍後將會詳細討論。

3. 當客戶端想從資料庫讀取資料時，它可以向 leader 或任何 follower 發起查詢。如果是想寫入資料庫的話，就只有 leader 才會接受寫請求，這也表示 followers 從客戶的角度來講是唯讀的副本。

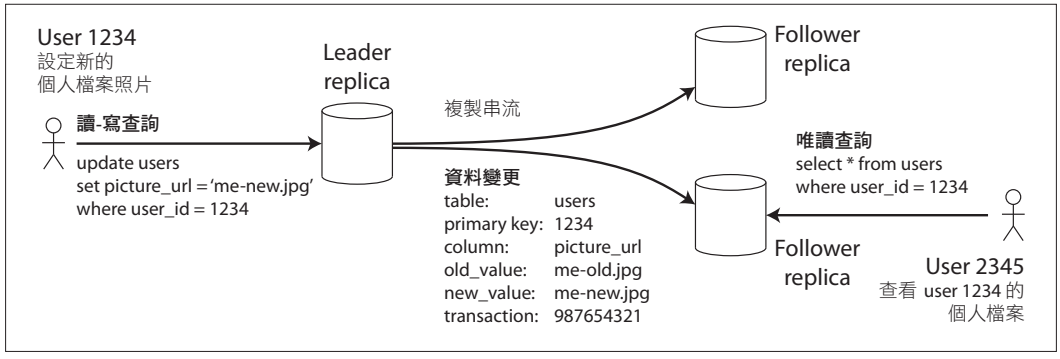

圖 5-1　Leader-base（master-slave）複製系統

許多關聯式資料庫都有內建這種複製模式，例如 PostgreSQL（自 9.0 版本後）、MySQL、Oracle DataGuard [2] 和 SQL Server 的 AlwaysOn Availability Groups [3]。一些非關聯式資料庫，包括 MongoDB、RethinkDB 和 Espresso [4] 也有支援。最後，leader-based replication 並不僅限於資料庫使用，分散式訊息代理如 Kafka [5] 和 RabbitMQ [6] 也有使用它。一些網路檔案系統和資料塊複製設備（如 DRBD）也有類似的作法。

同步複製與非同步複製

複製系統有一個重要細節，那就是 replication 的發生是**同步**（*synchronous*）還是**非同步的**（*asynchronous*）。對於關聯式資料庫，通常有一個可組態的選項用來決定同步或非同步複製；其他的系統則大多已經硬性設計為兩者的其中一種。

讓我們來看圖 5-1 的例子，當網站使用者更新個人資料照片時，想想看會發生的事情。在某個時刻，客戶端向 leader 發送更新請求；隨後，leader 收到了該請求。接著在後續某個時刻，leader 再將資料更新轉發給 followers。最終，由 leader 負責通知客戶端更新成功。

圖 5-2 顯示了系統各元件間的通訊：客戶端、leader 和兩個 followers，時間從左往右流動，其中粗箭頭用來表示請求或回應訊息。