

Table of Contents

簡介	1.1
序言	1.2
第一部分：資料系統基礎	1.3
第一章：可靠性、可伸縮性和可維護性	1.3.1
關於資料系統的思考	1.3.1.1
可靠性	1.3.1.2
可伸縮性	1.3.1.3
可維護性	1.3.1.4
本章小結	1.3.1.5
第二章：資料模型與查詢語言	1.3.2
關係模型與文件模型	1.3.2.1
資料查詢語言	1.3.2.2
圖資料模型	1.3.2.3
本章小結	1.3.2.4
第三章：儲存與檢索	1.3.3
驅動資料庫的資料結構	1.3.3.1
事務處理還是分析？	1.3.3.2
列式儲存	1.3.3.3
本章小結	1.3.3.4
第四章：編碼與演化	1.3.4
編碼資料的格式	1.3.4.1
資料流的型別	1.3.4.2
本章小結	1.3.4.3
第二部分：分散式資料	1.4
第五章：複製	1.4.1
領導者與追隨者	1.4.1.1
複製延遲問題	1.4.1.2
多主複製	1.4.1.3
無主複製	1.4.1.4
本章小結	1.4.1.5
第六章：分割槽	1.4.2
分割槽與複製	1.4.2.1
鍵值資料的分割槽	1.4.2.2
分割槽與次級索引	1.4.2.3
分割槽再平衡	1.4.2.4
請求路由	1.4.2.5
本章小結	1.4.2.6

第七章：事務	1.4.3
事務的棘手概念	1.4.3.1
弱隔離級別	1.4.3.2
可序列化	1.4.3.3
本章小結	1.4.3.4
第八章：分散式系統的麻煩	1.4.4
故障與部分失效	1.4.4.1
不可靠的網路	1.4.4.2
不可靠的時鐘	1.4.4.3
知識、真相與謊言	1.4.4.4
本章小結	1.4.4.5
第九章：一致性與共識	1.4.5
一致性保證	1.4.5.1
線性一致性	1.4.5.2
順序保證	1.4.5.3
分散式事務與共識	1.4.5.4
本章小結	1.4.5.5
第三部分：衍生資料	1.5
第十章：批處理	1.5.1
使用Unix工具的批處理	1.5.1.1
MapReduce和分散式檔案系統	1.5.1.2
MapReduce之後	1.5.1.3
本章小結	1.5.1.4
第十一章：流處理	1.5.2
傳遞事件流	1.5.2.1
資料庫與流	1.5.2.2
流處理	1.5.2.3
本章小結	1.5.2.4
第十二章：資料系統的未來	1.5.3
資料整合	1.5.3.1
分拆資料庫	1.5.3.2
將事情做正確	1.5.3.3
做正確的事情	1.5.3.4
本章小結	1.5.3.5
術語表	1.6
後記	1.7

設計資料密集型應用 - 中文翻譯

- 作者：Martin Kleppmann
- 原名：《Designing Data-Intensive Applications》
- 譯者：馮若航 (@Vonng)
- 校訂：@yingang
- 繁體：繁體中文版本 by @afunTW

使用 Typora、Gitbook 或 Github Pages 以獲取最佳閱讀體驗。

本地：你可在專案根目錄中執行 `make`，並透過瀏覽器閱讀（[線上預覽](#)）。

譯序

不懂資料庫的全棧工程師不是好架構師

—— Vonng

現今，尤其是在網際網路領域，大多數應用都屬於資料密集型應用。本書從底層資料結構到頂層架構設計，將資料系統設計中的精髓娓娓道來。其中的寶貴經驗無論是對架構師、DBA、還是後端工程師、甚至產品經理都會有幫助。

這是一本理論結合實踐的書，書中很多問題，譯者在實際場景中都曾遇到過，讀來讓人擊節扼腕。如果能早點讀到這本書，該少走多少彎路啊！

這也是一本深入淺出的書，講述概念的來龍去脈而不是賣弄定義，介紹事物發展演化歷程而不是事實堆砌，將複雜的概念講述的淺顯易懂，但又直擊本質不失深度。每章最後的引用質量非常好，是深入學習各個主題的絕佳索引。

本書為資料系統的設計、實現、與評價提供了很好的概念框架。讀完並理解本書內容後，讀者可以輕鬆看破大多數的技術忽悠，與技術磚家撕起來虎虎生風。

這是 2017 年譯者讀過最好的一本技術類書籍，這麼好的書沒有中文翻譯，實在是遺憾。某不才，願為先進技術文化的傳播貢獻一份力量。既可以深入學習有趣的技術主題，又可以鍛鍊中英文語言文字功底，何樂而不為？

前言

在我們的社會中，技術是一種強大的力量。資料、軟體、通訊可以用於壞的方面：不公平的階級固化，損害公民權利，保護既得利益集團。但也可以用於好的方面：讓底層人民發出自己的聲音，讓每個人都擁有機會，避免災難。本書獻給所有將技術用於善途的人們。

計算是一種流行文化，流行文化鄙視歷史。流行文化關乎個體身份和參與感，但與合作無關。流行文化活在當下，也與過去和未來無關。我認為大部分（為了錢）編寫程式碼的人就是這樣的，他們不知道自己的文化來自哪裡。

—— 阿蘭·凱接受 Dobb 博士的雜誌採訪時（2012 年）

目錄

序言

第一部分：資料系統基礎

- 第一章：可靠性、可伸縮性和可維護性
 - 關於資料系統的思考
 - 可靠性
 - 可伸縮性
 - 可維護性
 - 本章小結
- 第二章：資料模型與查詢語言
 - 關係模型與文件模型
 - 資料查詢語言
 - 圖資料模型
 - 本章小結
- 第三章：儲存與檢索
 - 驅動資料庫的資料結構
 - 事務處理還是分析？
 - 列式儲存
 - 本章小結
- 第四章：編碼與演化
 - 編碼資料的格式
 - 資料流的型別
 - 本章小結

第二部分：分散式資料

- 第五章：複製
 - 領導者與追隨者
 - 複製延遲問題
 - 多主複製
 - 無主複製
 - 本章小結
- 第六章：分割槽
 - 分割槽與複製
 - 鍵值資料的分割槽
 - 分割槽與次級索引
 - 分割槽再平衡
 - 請求路由
 - 本章小結
- 第七章：事務
 - 事務的棘手概念
 - 弱隔離級別
 - 可序列化
 - 本章小結
- 第八章：分散式系統的麻煩
 - 故障與部分失效
 - 不可靠的網路
 - 不可靠的時鐘
 - 知識、真相與謊言
 - 本章小結
- 第九章：一致性與共識
 - 一致性保證

- 線性一致性
- 順序保證
- 分散式事務與共識
- 本章小結

第三部分：衍生資料

- 第十章：批處理
 - 使用Unix工具的批處理
 - MapReduce和分散式檔案系統
 - MapReduce之後
 - 本章小結
- 第十一章：流處理
 - 傳遞事件流
 - 資料庫與流
 - 流處理
 - 本章小結
- 第十二章：資料系統的未來
 - 資料整合
 - 分拆資料庫
 - 將事情做正確
 - 做正確的事情
 - 本章小結

術語表

後記

法律宣告

從原作者處得知，已經有簡體中文的翻譯計劃，將於 2018 年末完成。[購買地址](#)

譯者純粹出於 學習目的 與 個人興趣 翻譯本書，不追求任何經濟利益。

譯者保留對此版本譯文的署名權，其他權利以原作者和出版社的主張為準。

本譯文只供學習研究參考之用，不得公開傳播發行或用於商業用途。有能力閱讀英文書籍者請購買正版支援。

貢獻

1. 全文校訂 by [@yingang](#)
2. 序言初翻修正 by [@seagullbird](#)
3. 第一章語法標點校正 by [@nevertiree](#)
4. 第六章部分校正 與第十章的初翻 by [@MuAlex](#)
5. 第一部分前言，ch2校正 by [@jiajiadebug](#)
6. 詞彙表、[後記](#)關於野豬的部分 by [@Chowss](#)
7. 繁體中文版本與轉換指令碼 by [@afunTW](#)
8. 多處翻譯修正 by [@songzhibin97](#) [@MamaShip](#) [@FangYuan33](#)
9. 感謝所有作出貢獻，提出意見的朋友們：

► [Pull Requests & Issues](#)

協議

[CC-BY 4.0](#)

序言

如果近幾年從業於軟體工程，特別是伺服器端和後端系統開發，那麼你很有可能已經被大量關於資料儲存和處理的時髦詞彙轟炸過了：NoSQL！大資料！Web-Scale！分片！最終一致性！ACID！CAP 定理！雲服務！MapReduce！實時！

在最近十年中，我們看到了很多有趣的進展，關於資料庫，分散式系統，以及在此基礎上構建應用程式的方式。這些進展有著各種各樣的驅動力：

- 谷歌、雅虎、亞馬遜、臉書、領英、微軟和推特等網際網路公司正在和巨大的流量 / 資料打交道，這迫使他們去創造能有效應對如此規模的新工具。
- 企業需要變得敏捷，需要低成本地檢驗假設，需要透過縮短開發週期和保持資料模型的靈活性，快速地響應新的市場洞察。
- 免費和開源軟體變得非常成功，在許多環境中比商業軟體和定製軟體更受歡迎。
- 處理器主頻幾乎沒有增長，但是多核處理器已經成為標配，網路也越來越快。這意味著並行化程度只增不減。
- 即使你在一個小團隊中工作，現在也可以構建分佈在多臺計算機甚至多個地理區域的系統，這要歸功於譬如亞馬遜網路服務（AWS）等基礎設施即服務（IaaS）概念的踐行者。
- 許多服務都要求高可用，因停電或維護導致的服務不可用，變得越來越難以接受。

資料密集型應用（data-intensive applications） 正在透過使用這些技術進步來推動可能性的邊界。一個應用被稱為**資料密集型**的，如果**資料是其主要挑戰**（資料量，資料複雜度或資料變化速度）——與之相對的是**計算密集型**，即處理器速度是其瓶頸。

幫助資料密集型應用儲存和處理資料的工具與技術，正迅速地適應這些變化。新型資料庫系統（“NoSQL”）已經備受關注，而訊息併列，快取，搜尋索引，批處理和流處理框架以及相關技術也非常重要。很多應用組合使用這些工具與技術。

這些生意盎然的時髦詞彙體現出人們對新的可能性的熱情，這是一件好事。但是作為軟體工程師和架構師，如果要開發優秀的應用，我們還需要對各種層出不窮的技術及其利弊權衡有精準的技術理解。為了獲得這種洞察，我們需要深挖時髦詞彙背後的內容。

幸運的是，在技術迅速變化的背後總是存在一些持續成立的原則，無論你使用了特定工具的哪個版本。如果你理解了這些原則，就可以領會這些工具的適用場景，如何充分利用它們，以及如何避免其中的陷阱。這正是本書的初衷。

本書的目標是幫助你在飛速變化的資料處理和資料儲存技術大觀園中找到方向。本書並不是某個特定工具的教程，也不是一本充滿枯燥理論的教科書。相反，我們將看到一些成功資料系統的樣例：許多流行應用每天都要在生產中滿足可伸縮性、效能、以及可靠性的要求，而這些技術構成了這些應用的基礎。

我們將深入這些系統的內部，理清它們的關鍵演算法，討論背後的原則和它們必須做出的權衡。在這個過程中，我們將嘗試尋找**思考**資料系統的有效方式——不僅關於它們**如何**工作，還包括它們**為什麼**以這種方式工作，以及哪些問題是我們需要問的。

閱讀本書後，你能很好地決定哪種技術適合哪種用途，並瞭解如何將工具組合起來，為一個良好應用架構奠定基礎。本書並不足以使你從頭開始構建自己的資料庫儲存引擎，不過幸運的是這基本上很少有必要。你將獲得對系統底層發生事情的敏銳直覺，這樣你就有能力推理它們的行為，做出優秀的設計決策，並追蹤任何可能出現的問題。

本書的目標讀者

如果你開發的應用具有用於儲存或處理資料的某種伺服器 / 後端系統，而且使用網路（例如，Web 應用、移動應用或連線到網際網路的感測器），那麼本書就是為你準備的。

本書是為軟體工程師，軟體架構師，以及喜歡寫程式碼的技術經理準備的。如果你需要對所從事系統的架構做出決策——例如你需要選擇解決某個特定問題的工具，並找出如何最好地使用這些工具，那麼這本書對你尤有價值。但即使你無法選擇你的工具，本書仍將幫助你更好地瞭解所使用工具的長處和短處。

你應當具有一些開發 Web 應用或網路服務的經驗，且應當熟悉關係型資料庫和 SQL。任何你瞭解的非關係型資料庫和其他與資料相關工具都會有所幫助，但不是必需的。對常見網路協議如 TCP 和 HTTP 的大概理解是有幫助的。程式語言或框架的選擇對閱讀本書沒有任何不同影響。

如果以下任意一條對你為真，你會發現這本書很有價值：

- 你想了解如何使資料系統可伸縮，例如，支援擁有數百萬使用者的 Web 或移動應用。
- 你需要提高應用程式的可用性（最大限度地減少停機時間），保持穩定執行。
- 你正在尋找使系統在長期執行過程易於維護的方法，即使系統規模增長，需求與技術也發生變化。
- 你對事物的運作方式有著天然的好奇心，並且希望知道一些主流網站和線上服務背後發生的事情。這本書打破了各種資料庫和資料處理系統的內幕，探索這些系統設計中的智慧是非常有趣的。

有時在討論可伸縮的資料系統時，人們會說：“你又不在谷歌或亞馬遜，別操心可伸縮性了，直接上關係型資料庫”。這個陳述有一定的道理：為了不必要的伸縮性而設計程式，不僅會浪費不必要的精力，並且可能會把你鎖死在一個不靈活的設計中。實際上這是一種“過早最佳化”的形式。不過，選擇合適的工具確實很重要，而不同的技術各有優缺點。我們將看到，關係資料庫雖然很重要，但絕不是資料處理的終章。

本書涉及的領域

本書並不會嘗試告訴讀者如何安裝或使用特定的軟體包或 API，因為已經有大量文件給出了詳細的使用說明。相反，我們會討論資料系統的基礎——各種原則與利弊權衡，並探討了不同產品所做出的不同設計決策。

在電子書中包含了線上資源全文的連結。所有連結在出版時都進行了驗證，但不幸的是，由於網路的自然規律，連結往往會頻繁地破損。如果你遇到連結斷開的情況，或者正在閱讀本書的列印副本，可以使用搜索引擎查詢參考文獻。對於學術論文，你可以在 Google 學術中搜索標題，查詢可以公開獲取的 PDF 檔案。或者，你也可以在 <https://github.com/ept/ddia-references> 中找到所有的參考資料，我們在那兒維護最新的連結。

我們主要關注的是資料系統的 架構 (**architecture**)，以及它們被整合到資料密集型應用中的方式。本書沒有足夠的空間覆蓋部署、運維、安全、管理等領域——這些都是複雜而重要的主題，僅僅在本書中用粗略的註解討論這些對它們很不公平。每個領域都值得用單獨的書去講。

本書中描述的許多技術都被涵蓋在 **大資料 (Big Data)** 這個時髦詞的範疇中。然而“大資料”這個術語被濫用，缺乏明確定義，以至於在嚴肅的工程討論中沒有用處。這本書使用歧義更小的術語，如“單節點”之於“分散式系統”，或“線上 / 互動式系統”之於“離線 / 批處理系統”。

本書對 **自由和開源軟體 (FOSS)** 有一定偏好，因為閱讀、修改和執行原始碼是瞭解某事物詳細工作原理的好方法。開放的平臺也可以降低供應商壟斷的風險。然而在適當的情況下，我們也會討論專利軟體（閉源軟體，軟體即服務 SaaS，或一些在文獻中描述過但未公開發行的公司內部軟體）。

本書綱要

本書分為三部分：

1. 在 **第一部分** 中，我們會討論設計資料密集型應用所賴的基本思想。我們從 **第一章** 開始，討論我們實際要達到的目標：可靠性、可伸縮性和可維護性；我們該如何思考這些概念；以及如何實現它們。在 **第二章** 中，我們比較了幾種不同的資料模型和查詢語言，看看它們如何適用於不同的場景。在 **第三章** 中將討論儲存引擎：資料庫如何在磁碟上擺放資料，以便能高效地再次找到它。**第四章** 轉向資料編碼（序列化），以及隨時間演化的模式。

2. 在 [第二部分](#) 中，我們從討論儲存在一臺機器上的資料轉向討論分佈在多臺機器上的資料。這對於可伸縮性通常是非常必要的，但帶來了各種獨特的挑戰。我們首先討論複製 ([第五章](#))、分割槽 / 分片 ([第六章](#)) 和事務 ([第七章](#))。然後我們將探索關於分散式系統問題的更多細節 ([第八章](#))，以及在分散式系統中實現一致性與共識意味著什麼 ([第九章](#))。
3. 在 [第三部分](#) 中，我們討論那些從其他資料集衍生出一些資料集的系統。衍生資料經常出現在異構系統中：當沒有單個數據庫可以把所有事情都做的很好時，應用需要整合幾種不同的資料庫、快取、索引等。在 [第十章](#) 中我們將從一種衍生資料的批處理方法開始，然後在此基礎上建立在 [第十一章](#) 中討論的流處理。最後，在 [第十二章](#) 中，我們將所有內容彙總，討論在將來構建可靠、可伸縮和可維護的應用程式的方法。

參考文獻與延伸閱讀

本書中討論的大部分內容已經在其它地方以某種形式出現過了——會議簡報、研究論文、部落格文章、程式碼、BUG 跟蹤器、郵件列表以及工程習慣中。本書總結了不同來源資料中最重要的想法，並在文字中包含了指向原始文獻的連結。如果你想更深入地探索一個領域，那麼每章末尾的參考文獻都是很好的資源，其中大部分可以免費線上獲取。

O'Reilly Safari

[Safari](#) (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit <http://oreilly.com/safari>.

致謝

本書融合了學術研究和工業實踐的經驗，融合並系統化了大量其他人的想法與知識。在計算領域，我們往往會被各種新鮮花樣所吸引，但我認為前人完成的工作中，有太多值得我們學習的地方了。本書有 800 多處引用：文章、部落格、講座、文件等，對我來說這些都是寶貴的學習資源。我非常感謝這些材料的作者分享他們的知識。

我也從與人交流中學到了很多東西，很多人花費了寶貴的時間與我討論想法並耐心解釋。特別感謝 Joe Adler, Ross Anderson, Peter Bailis, Márton Balassi, Alastair Beresford, Mark Callaghan, Mat Clayton, Patrick Collison, Sean Cribbs, Shirshanka Das, Niklas Ekström, Stephan Ewen, Alan Fekete, Gyula Fóra, Camille Fournier, Andres Freund, John Garbutt, Seth Gilbert, Tom Haggett, Pat Helland, Joe Hellerstein, Jakob Homan, Heidi Howard, John Hugg, Julian Hyde, Conrad Irwin, Evan Jones, Flavio Junqueira, Jessica Kerr, Kyle Kingsbury, Jay Kreps, Carl Lerche, Nicolas Liochon, Steve Loughran, Lee Mallabone, Nathan Marz, Caitie McCaffrey, Josie McLellan, Christopher Meiklejohn, Ian Meyers, Neha Narkhede, Neha Narula, Cathy O'Neil, Onora O'Neill, Ludovic Orban, Zoran Perkov, Julia Powles, Chris Riccomini, Henry Robinson, David Rosenthal, Jennifer Rullmann, Matthew Sackman, Martin Scholl, Amit Sela, Gwen Shapira, Greg Spurrier, Sam Stokes, Ben Stopford, Tom Stuart, Diana Vasile, Rahul Vohra, Pete Warden, 以及 Brett Wooldridge。

更多人透過審閱草稿並提供反饋意見在本書的創作過程中做出了無價的貢獻。我要特別感謝 Raul Agepati, Tyler Akidau, Mattias Andersson, Sasha Baranov, Veena Basavaraj, David Beyer, Jim Brikman, Paul Carey, Raul Castro Fernandez, Joseph Chow, Derek Elkins, Sam Elliott, Alexander Gallego, Mark Grover, Stu Halloway, Heidi Howard, Nicola Kleppmann, Stefan Kruppa, Bjorn Madsen, Sander Mak, Stefan Podkowinski, Phil Potter, Hamid Ramazani, Sam Stokes, 以及 Ben Summers。當然對於本書中的任何遺留錯誤或難以接受的見解，我都承擔全部責任。

為了幫助這本書落地，並且耐心地處理我緩慢的寫作和不尋常的要求，我要對編輯 Marie Beaugureau，Mike Loukides，Ann Spencer 和 O'Reilly 的所有團隊表示感謝。我要感謝 Rachel Head 幫我找到了合適的術語。我要感謝 Alastair Beresford，Susan Goodhue，Neha Narkhede 和 Kevin Scott，在其他工作事務之外給了我充分地創作時間和自由。

特別感謝 Shabbir Diwan 和 Edie Freedman，他們非常用心地為各章配了地圖。他們提出了不落俗套的靈感，創作了這些地圖，美麗而引人入勝，真是太棒了。

最後我要表達對家人和朋友們的愛，沒有他們，我將無法走完這個將近四年的寫作歷程。你們是最棒的。

第一部分：資料系統基礎

本書前四章介紹了資料系統底層的基礎概念，無論是在單臺機器上執行的單點資料系統，還是分佈在多臺機器上的分散式資料系統都適用。

1. 第一章 將介紹本書使用的術語和方法。可靠性，可伸縮性和可維護性，這些詞彙到底意味著什麼？如何實現這些目標？
2. 第二章 將對幾種不同的 資料模型和查詢語言 進行比較。從程式設計師的角度看，這是資料庫之間最明顯的區別。不同的資料模型適用於不同的應用場景。
3. 第三章 將深入 儲存引擎 內部，研究資料庫如何在磁碟上擺放資料。不同的儲存引擎針對不同的負載進行最佳化，選擇合適的儲存引擎對系統性能有巨大影響。
4. 第四章 將對幾種不同的 資料編碼 進行比較。特別研究了這些格式在應用需求經常變化、模式需要隨時間演變的環境中表現如何。

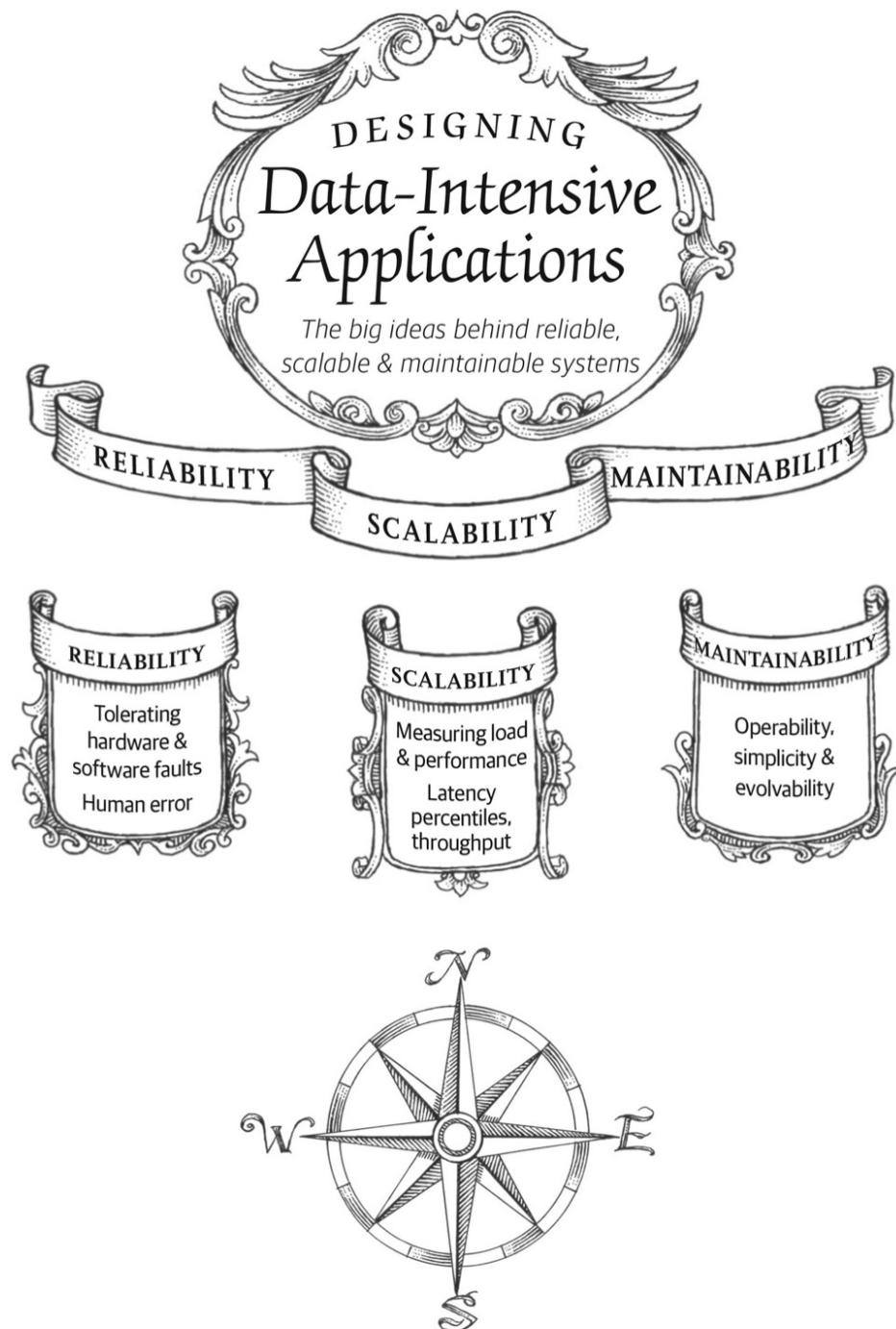
第二部分將專門討論在 分散式資料系統 中特有的問題。

目錄

1. 可靠性、可伸縮性和可維護性
2. 資料模型與查詢語言
3. 儲存與檢索
4. 編碼與演化

上一章	目錄	下一章
序言	設計資料密集型應用	第一章：可靠性、可伸縮性和可維護性

第一章：可靠性、可伸縮性和可維護性



網際網路做得太棒了，以至於大多數人將它看作像太平洋這樣的自然資源，而不是什麼人工產物。上一次出現這種大規模且無差錯的技術，你還記得是什麼時候嗎？

—— 艾倫·凱 在接受 Dobb 博士雜誌採訪時說（2012 年）

[TOC]

現今很多應用程式都是 **資料密集型（data-intensive）** 的，而非 **計算密集型（compute-intensive）** 的。因此 CPU 很少成為這類應用的瓶頸，更大的問題通常來自資料量、資料複雜性、以及資料的變更速度。

資料密集型應用通常由標準組件構建而成，標準組件提供了很多通用的功能；例如，許多應用程式都需要：

- 儲存資料，以便自己或其他應用程式之後能再次找到（資料庫，即 *databases*）
- 記住開銷昂貴操作的結果，加快讀取速度（快取，即 *caches*）
- 允許使用者按關鍵字搜尋資料，或以各種方式對資料進行過濾（搜尋索引，即 *search indexes*）
- 向其他程序傳送訊息，進行非同步處理（流處理，即 *stream processing*）
- 定期處理累積的大批次資料（批處理，即 *batch processing*）

如果這些功能聽上去平淡無奇，那是因為這些 **資料系統（data system）** 是非常成功的抽象：我們一直不假思索地使用它們並習以為常。絕大多數工程師不會幻想從零開始編寫儲存引擎，因為在開發應用時，資料庫已經是足夠完美的工具了。

但現實沒有這麼簡單。不同的應用有著不同的需求，因而資料庫系統也是百花齊放，有著各式各樣的特性。實現快取有很多種手段，建立搜尋索引也有好幾種方法，諸如此類。因此在開發應用前，我們依然有必要先弄清楚最適合手頭工作的工具和方法。而且當單個工具解決不了你的問題時，組合使用這些工具可能還是有些難度的。

本書將是一趟關於資料系統原理、實踐與應用的旅程，並講述了設計資料密集型應用的方法。我們將探索不同工具之間的共性與特性，以及各自的實現原理。

本章將從我們所要實現的基礎目標開始：可靠、可伸縮、可維護的資料系統。我們將澄清這些詞語的含義，概述考量這些目標的方法。並回顧一些後續章節所需的基礎知識。在接下來的章節中我們將抽絲剝繭，研究設計資料密集型應用時可能遇到的設計決策。

關於資料系統的思考

我們通常認為，資料庫、訊息佇列、快取等工具分屬於幾個差異顯著的類別。雖然資料庫和訊息隊列表面上有一些相似性——它們都會儲存一段時間的資料——但它們有迥然不同的訪問模式，這意味著迥異的效能特徵和實現手段。

那我們為什麼要把這些東西放在 **資料系統（data system）** 的總稱之下混為一談呢？

近些年來，出現了許多新的資料儲存工具與資料處理工具。它們針對不同應用場景進行最佳化，因此不再適合生硬地歸入傳統類別【1】。類別之間的界限變得越來越模糊，例如：資料儲存可以被當成訊息佇列用（Redis），訊息佇列則帶有類似資料庫的持久保證（Apache Kafka）。

其次，越來越多的應用程式有著各種嚴格而廣泛的要求，單個工具不足以滿足所有的資料處理和儲存需求。取而代之的是，總體工作被拆分成一系列能被單個工具高效完成的任務，並透過應用程式碼將它們縫合起來。

例如，如果將快取（應用管理的快取層，Memcached 或同類產品）和全文搜尋（全文搜尋伺服器，例如 Elasticsearch 或 Solr）功能從主資料庫剝離出來，那麼使快取 / 索引與主資料庫保持同步通常是應用程式碼的責任。圖 1-1 紹出了這種架構可能的樣子（細節將在後面的章節中詳細介紹）。

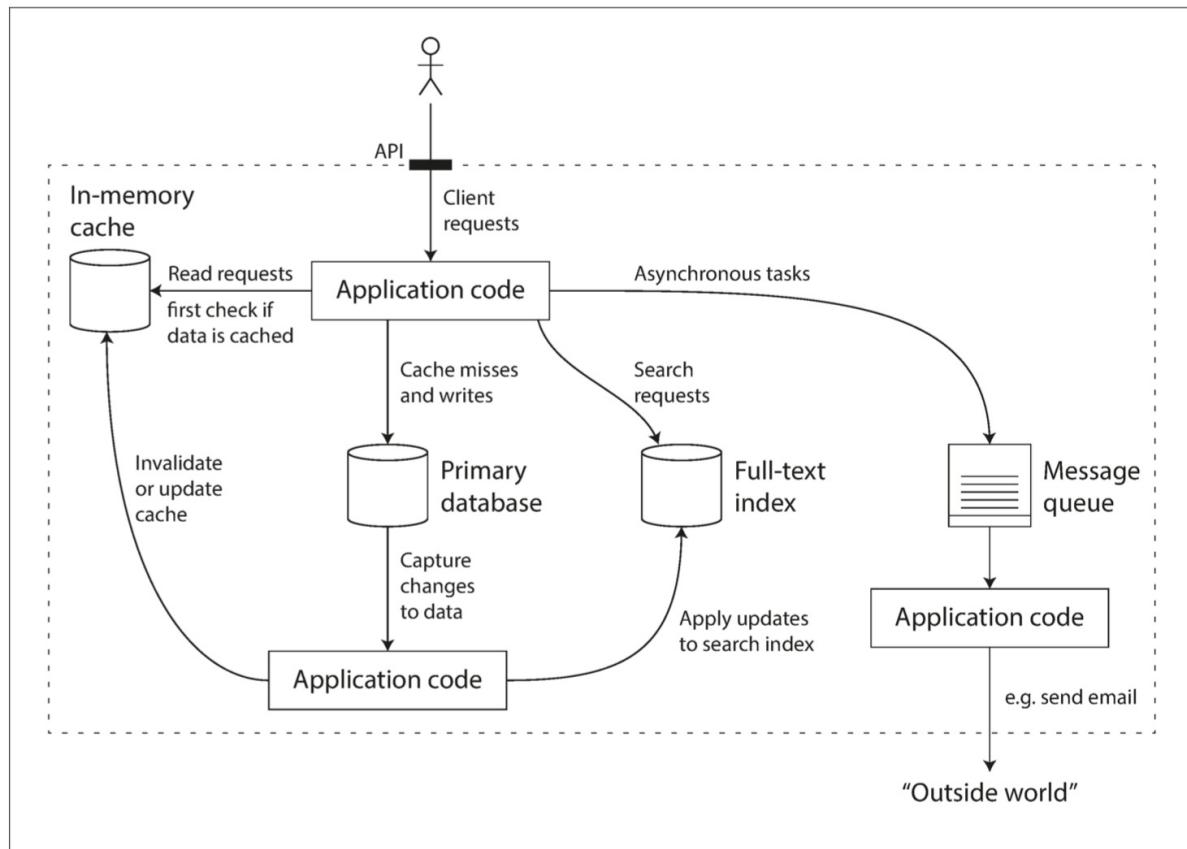


圖 1-1 一個可能的組合使用多個元件的資料系統架構

當你將多個工具組合在一起提供服務時，服務的介面或 **應用程式程式設計介面（API, Application Programming Interface）** 通常向客戶端隱藏這些實現細節。現在，你基本上已經使用較小的通用元件建立了一個全新的、專用的資料系統。這個新的複合資料系統可能會提供特定的保證，例如：快取在寫入時會作廢或更新，以便外部客戶端獲取一致的結果。現在你不僅是應用程式開發人員，還是資料系統設計人員了。

設計資料系統或服務時可能會遇到很多棘手的問題，例如：當系統出問題時，如何確保資料的正確性和完整性？當部分系統退化降級時，如何為客戶提供始終如一的良好效能？當負載增加時，如何擴容應對？什麼樣的 API 才是好的 API？影響資料系統設計的因素很多，包括參與人員的技能和經驗、歷史遺留問題、系統路徑依賴、交付時限、公司的風險容忍度、監管約束等，這些因素都需要具體問題具體分析。

本書著重討論三個在大多數軟體系統中都很重要的問題：

- 可靠性（Reliability）

系統在 **困境**（adversity，比如硬體故障、軟體故障、人為錯誤）中仍可正常工作（正確完成功能，並能達到期望的效能水準）。請參閱“[可靠性](#)”。

- 可伸縮性（Scalability）

有合理的辦法應對系統的增長（資料量、流量、複雜性）。請參閱“[可伸縮性](#)”。

- 可維護性（Maintainability）

許多不同的人（工程師、運維）在不同的生命週期，都能高效地在系統上工作（使系統保持現有行為，並適應新的應用場景）。請參閱“[可維護性](#)”。

人們經常追求這些詞彙，卻沒有清楚理解它們到底意味著什麼。為了工程的嚴謹性，本章的剩餘部分將探討可靠性、可伸縮性和可維護性的含義。為實現這些目標而使用的各種技術，架構和演算法將在後續的章節中研究。

可靠性

人們對於一個東西是否可靠，都有一個直觀的想法。人們對可靠軟體的典型期望包括：

- 應用程式表現出使用者所期望的功能。
- 允許使用者犯錯，允許使用者以出乎意料的方式使用軟體。
- 在預期的負載和資料量下，效能滿足要求。
- 系統能防止未經授權的訪問和濫用。

如果所有這些在一起意味著“正確工作”，那麼可以把可靠性粗略理解為“即使出現問題，也能繼續正確工作”。

造成錯誤的原因叫做 **故障 (fault)**，能預料並應對故障的系統特性可稱為 **容錯 (fault-tolerant)** 或 **韌性 (resilient)**。“容錯”一詞可能會產生誤導，因為它暗示著系統可以容忍所有可能的錯誤，但在實際中這是不可能的。比方說，如果整個地球（及其上的所有伺服器）都被黑洞吞噬了，想要容忍這種錯誤，需要把網路託管到太空中——這種預算能不能批准就祝你好運了。所以在討論容錯時，只有談論特定型別的錯誤才有意義。

注意 **故障 (fault)** 不同於 **失效 (failure)** 【2】。故障通常定義為系統的一部分狀態偏離其標準，而失效則是系統作為一個整體停止向用戶提供服務。故障的機率不可能降到零，因此最好設計容錯機制以防因**故障**而導致**失效**。本書中我們將介紹幾種用不可靠的部件構建可靠系統的技術。

反直覺的是，在這類容錯系統中，透過故意觸發來提高故障率是有意義的，例如：在沒有警告的情況下隨機地殺死單個程序。許多高危漏洞實際上是由糟糕的錯誤處理導致的【3】，因此我們可以透過故意引發故障來確保容錯機制不斷執行並接受考驗，從而提高故障自然發生時系統能正確處理的信心。Netflix 公司的 *Chaos Monkey* 【4】就是這種方法的一個例子。

儘管比起**阻止錯誤 (prevent error)**，我們通常更傾向於**容忍錯誤**。但也有**預防勝於治療**的情況（比如不存在治療方法時）。安全問題就屬於這種情況。例如，如果攻擊者破壞了系統，並獲取了敏感資料，這種事是撤銷不了的。但本書主要討論的是可以恢復的故障種類，正如下面幾節所述。

硬體故障

當想到系統失效的原因時，**硬體故障 (hardware faults)** 總會第一個進入腦海。硬碟崩潰、記憶體出錯、機房斷電、有人拔錯網線……任何與大型資料中心打過交道的人都會告訴你：一旦你擁有很多機器，這些事情總會發生！

據報道稱，硬碟的**平均無故障時間 (MTTF, mean time to failure)** 約為 10 到 50 年【5】【6】。因此從數學期望上講，在擁有 10000 個磁碟的儲存叢集上，平均每天會有 1 個磁碟出故障。

為了減少系統的故障率，第一反應通常都是增加單個硬體的冗餘度，例如：磁碟可以組建 RAID，伺服器可能有雙路電源和熱插拔 CPU，資料中心可能有電池和柴油發電機作為後備電源，某個元件掛掉時冗餘元件可以立刻接管。這種方法雖然不能完全防止由硬體問題導致的系統失效，但它簡單易懂，通常也足以讓機器不間斷執行很多年。

直到最近，硬體冗餘對於大多數應用來說已經足夠了，它使單臺機器完全失效變得相當罕見。只要你能快速地把備份恢復到新機器上，故障停機時間對大多數應用而言都算不上災難性的。只有少量高可用性至關重要的應用才會要求有多套硬體冗餘。

但是隨著資料量和應用計算需求的增加，越來越多的應用開始大量使用機器，這會相應地增加硬體故障率。此外，在類似亞馬遜 AWS (Amazon Web Services) 的一些雲服務平臺上，虛擬機器例項不可用卻沒有任何警告也是很常見的【7】，因為雲平臺的設計就是優先考慮**靈活性 (flexibility)** 和**彈性 (elasticity)**ⁱ，而不是單機可靠性。

如果在硬體冗餘的基礎上進一步引入軟體容錯機制，那麼系統在容忍整個（單臺）機器故障的道路上就更進一步了。這樣的系統也有運維上的便利，例如：如果需要重啟機器（例如應用作業系統安全補丁），單伺服器系統就需要計劃停機。而允許機器失效的系統則可以一次修復一個節點，無需整個系統停機。

ⁱ 在**應對負載的方法**一節定義 ↪

軟體錯誤

我們通常認為硬體故障是隨機的、相互獨立的：一臺機器的磁碟失效並不意味著另一臺機器的磁碟也會失效。雖然大量硬體元件之間可能存在微弱的相關性（例如伺服器機架的溫度等共同的原因），但同時發生故障也是極為罕見的。

另一類錯誤是內部的 **系統性錯誤 (systematic error)** 【8】。這類錯誤難以預料，而且因為是跨節點相關的，所以比起不相關的硬體故障往往可能造成更多的 **系統失效** 【5】。例子包括：

- 接受特定的錯誤輸入，便導致所有應用伺服器例項崩潰的 BUG。例如 2012 年 6 月 30 日的閏秒，由於 Linux 核心中的一個錯誤【9】，許多應用同時掛掉了。
- 失控程序會用盡一些共享資源，包括 CPU 時間、記憶體、磁碟空間或網路頻寬。
- 系統依賴的服務變慢，沒有響應，或者開始返回錯誤的響應。
- 級聯故障，一個元件中的小故障觸發另一個元件中的故障，進而觸發更多的故障【10】。

導致這類軟體故障的 BUG 通常會潛伏很長時間，直到被異常情況觸發為止。這種情況意味著軟體對其環境做出了某種假設——雖然這種假設通常來說是正確的，但由於某種原因最後不再成立了【11】。

雖然軟體中的系統性故障沒有速效藥，但我們還是有很多小辦法，例如：仔細考慮系統中的假設和互動；徹底的測試；程序隔離；允許程序崩潰並重啟；測量、監控並分析生產環境中的系統行為。如果系統能夠提供一些保證（例如在一個訊息佇列中，進入與發出的訊息數量相等），那麼系統就可以在執行時不斷自檢，並在出現 **差異 (discrepancy)** 時報警【12】。

人為錯誤

設計並構建了軟體系統的工程師是人類，維持系統執行的運維也是人類。即使他們懷有最大的善意，人類也是不可靠的。舉個例子，一項關於大型網際網路服務的研究發現，運維配置錯誤是導致服務中斷的主要原因之一，而硬體故障（伺服器或網路）僅導致了 10-25% 的服務中斷【13】。

儘管人類不可靠，但怎麼做才能讓系統變得可靠？最好的系統會組合使用以下幾種辦法：

- 以最小化犯錯機會的方式設計系統。例如，精心設計的抽象、API 和管理後臺使做對事情更容易，搞砸事情更困難。但如果介面限制太多，人們就會忽略它們的好處而想辦法繞開。很難正確把握這種微妙的平衡。
- 將人們最容易犯錯的地方與可能導致失效的地方 **解耦 (decouple)**。特別是提供一個功能齊全的非生產環境 **沙箱 (sandbox)**，使人們可以在不影響真實使用者的情況下，使用真實資料安全地探索和實驗。
- 在各個層次進行徹底的測試【3】，從單元測試、全系統整合測試到手動測試。自動化測試易於理解，已經被廣泛使用，特別適合用來覆蓋正常情況中少見的 **邊緣場景 (corner case)**。
- 允許從人為錯誤中簡單快速地恢復，以最大限度地減少失效情況帶來的影響。例如，快速回滾配置變更，分批發布新程式碼（以便任何意外錯誤隻影響一小部分使用者），並提供資料重算工具（以備舊的計算出錯）。
- 配置詳細和明確的監控，比如效能指標和錯誤率。在其他工程學科中這指的是 **遙測 (telemetry)**（一旦火箭離開了地面，遙測技術對於跟蹤發生的事情和理解失敗是至關重要的）。監控可以向我們發出預警訊號，並允許我們檢查是否有任何地方違反了假設和約束。當出現問題時，指標資料對於問題診斷是非常寶貴的。
- 良好的管理實踐與充分的培訓——一個複雜而重要的方面，但超出了本書的範圍。

可靠性有多重要？

可靠性不僅僅是針對核電站和空中交通管制軟體而言，我們也期望更多平凡的應用能可靠地執行。商務應用中的錯誤會導致生產力損失（也許資料報告不完整還會有法律風險），而電商網站的中斷則可能會導致收入和聲譽的巨大損失。

即使在“非關鍵”應用中，我們也對使用者負有責任。試想一位家長把所有的照片和孩子的影片儲存在你的照片應用裡【15】。如果資料庫突然損壞，他們會感覺如何？他們可能會知道如何從備份恢復嗎？

在某些情況下，我們可能會選擇犧牲可靠性來降低開發成本（例如為未經證實的市場開發產品原型）或運營成本（例如利潤率極低的服務），但我們偷工減料時，應該清楚意識到自己在做什麼。

可伸縮性

系統今天能可靠執行，並不意味未來也能可靠執行。服務 **降級 (degradation)** 的一個常見原因是負載增加，例如：系統負載已經從一萬個併發使用者增長到十萬個併發使用者，或者從一百萬增長到一千萬。也許現在處理的資料量級要比過去大得多。

可伸縮性 (Scalability) 是用來描述系統應對負載增長能力的術語。但是請注意，這不是貼在系統上的一維標籤：說“X 可伸縮”或“Y 不可伸縮”是沒有任何意義的。相反，討論可伸縮性意味著考慮諸如“如果系統以特定方式增長，有什麼選項可以應對增長？”和“如何增加計算資源來處理額外的負載？”等問題。

描述負載

在討論增長問題（如果負載加倍會發生什麼？）前，首先要能簡要描述系統的當前負載。負載可以用一些稱為 **負載引數 (load parameters)** 的數字來描述。引數的最佳選擇取決於系統架構，它可能是每秒向 Web 啟服器發出的請求、資料庫中的讀寫比率、聊天室中同時活躍的使用者數量、快取命中率或其他東西。除此之外，也許平均情況對你很重要，也許你的瓶頸是少數極端場景。

為了使這個概念更加具體，我們以推特在 2012 年 11 月釋出的資料【16】為例。推特的兩個主要業務是：

- 釋出推文

使用者可以向其粉絲釋出新訊息（平均 4.6k 請求 / 秒，峰值超過 12k 請求 / 秒）。

- 主頁時間線

使用者可以查閱他們關注的人釋出的推文（300k 請求 / 秒）。

處理每秒 12,000 次寫入（發推文的速率峰值）還是很簡單的。然而推特的伸縮性挑戰並不是主要來自推特量，而是來自 **扇出 (fan-out)**ⁱⁱ——每個使用者關注了很多人，也被很多人關注。

ⁱⁱ. 扇出：從電子工程學中借用的術語，它描述了輸入連線到另一個門輸出的邏輯閘數量。輸出需要提供足夠的電流來驅動所有連線的輸入。在事務處理系統中，我們使用它來描述為了服務一個傳入請求而需要執行其他服務的請求數量。 ↪

大體上講，這一對操作有兩種實現方式。

1. 釋出推文時，只需將新推文插入全域性推文集合即可。當一個使用者請求自己的主頁時間線時，首先查詢他關注的所有人，查詢這些被關注使用者釋出的推文並按時間順序合併。在如 圖 1-2 所示的關係型資料庫中，可以編寫這樣的查詢：

```
SELECT tweets.*, users.*
  FROM tweets
  JOIN users ON tweets.sender_id = users.id
  JOIN follows ON follows.followee_id = users.id
 WHERE follows.follower_id = current_user
```

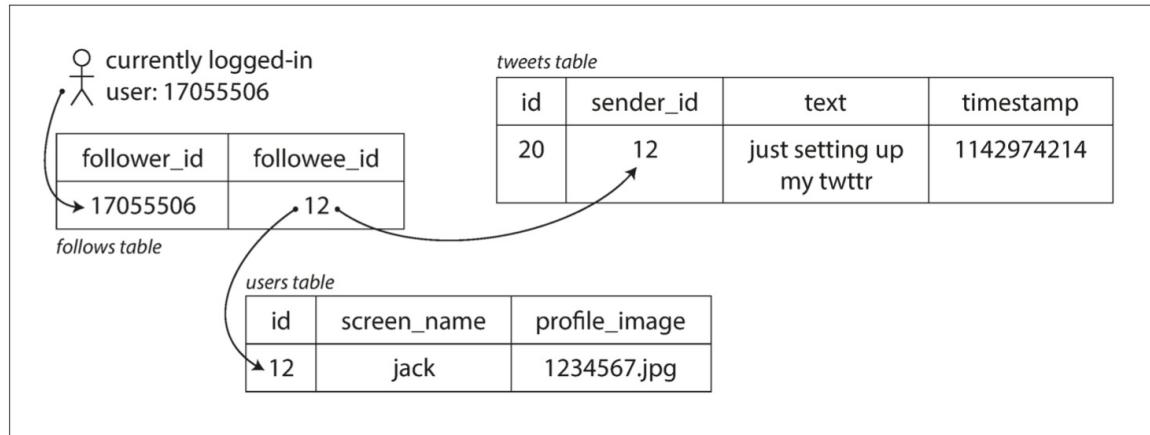
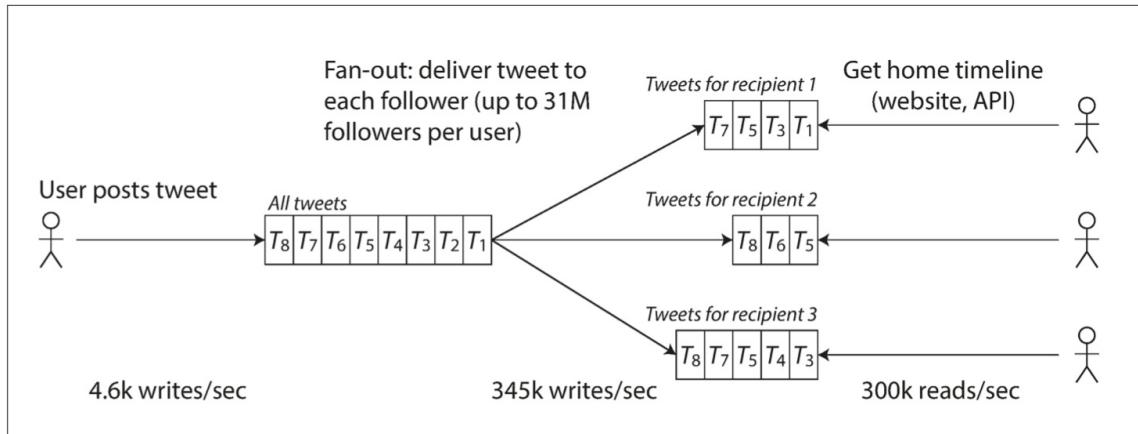


圖 1-2 推特主頁時間線的關係型模式簡單實現

2. 為每個使用者的主頁時間線維護一個快取，就像每個使用者的推文收件箱（[圖 1-3](#)）。當一個使用者釋出推文時，查詢所有關注該使用者的人，並將新的推文插入到每個主頁時間線快取中。因此讀取主頁時間線的請求開銷很小，因為結果已經提前計算好了。

**圖 1-3 用於分發推特至關注者的資料流水線，2012 年 11 月的負載引數【16】**

推特的第一個版本使用了方法 1，但系統很難跟上主頁時間線查詢的負載。所以公司轉向了方法 2，方法 2 的效果更好，因為發推頻率比查詢主頁時間線的頻率幾乎低了兩個數量級，所以在這種情況下，最好在寫入時做更多的工作，而在讀取時做更少的工作。

然而方法 2 的缺點是，發推現在需要大量的額外工作。平均來說，一條推文會發往約 75 個關注者，所以每秒 4.6k 的發推寫入，變成了對主頁時間線快取每秒 345k 的寫入。但這個平均值隱藏了使用者粉絲數差異巨大這一現實，一些使用者有超過 3000 萬的粉絲，這意味著一條推文就可能會導致主頁時間線快取的 3000 萬次寫入！及時完成這種操作是一個巨大的挑戰——推特嘗試在 5 秒內向粉絲傳送推文。

在推特的例子中，每個使用者粉絲數的分佈（可能按這些使用者的發推頻率來加權）是探討可伸縮性的一個關鍵負載引數，因為它決定了扇出負載。你的應用程式可能具有非常不同的特徵，但可以採用相似的原則來考慮它的負載。

推特軼事的最終轉折：現在已經穩健地實現了方法 2，推特逐步轉向了兩種方法的混合。大多數使用者發的推文會被扇出寫入其粉絲主頁時間線快取中。但是少數擁有海量粉絲的使用者（即名流）會被排除在外。當用戶讀取主頁時間線時，分別地獲取出該使用者所關注的每位名流的推文，再與使用者的主頁時間線快取合併，如方法 1 所示。這種混合方法能始終如一地提供良好效能。在 [第十二章](#) 中我們將重新討論這個例子，這在覆蓋更多技術層面之後。

描述效能

一旦系統的負載被描述好，就可以研究當負載增加會發生什麼。我們可以從兩種角度來看：

- 增加負載引數並保持系統資源（CPU、記憶體、網路頻寬等）不變時，系統性能將受到什麼影響？
- 增加負載引數並希望保持效能不變時，需要增加多少系統資源？

這兩個問題都需要效能資料，所以讓我們簡單地看一下如何描述系統性能。

對於 Hadoop 這樣的批處理系統，通常關心的是 **吞吐量 (throughput)**，即每秒可以處理的記錄數量，或者在特定規模資料集上執行作業的總時間ⁱⁱⁱ。對於線上系統，通常更重要的是服務的 **響應時間 (response time)**，即客戶端傳送請求到接收響應之間的時間。

ⁱⁱⁱ. 理想情況下，批次作業的執行時間是資料集的大小除以吞吐量。在實踐中由於資料傾斜（資料不是均勻分佈在每個工作程序中），需要等待最慢的任務完成，所以執行時間往往更長。 ↪

延遲和響應時間

延遲 (latency) 和響應時間 (response time) 經常用作同義詞，但實際上它們並不一樣。響應時間是客戶所看到的，除了實際處理請求的時間（服務時間 (service time)）之外，還包括網路延遲和排隊延遲。延遲是某個請求等待處理的持續時長，在此期間它處於休眠 (latent) 狀態，並等待服務【17】。

即使不斷重複傳送同樣的請求，每次得到的響應時間也都會略有不同。現實世界的系統會處理各式各樣的請求，響應時間可能會有很大差異。因此我們需要將響應時間視為一個可以測量的數值 分佈 (distribution)，而不是單個數值。

在 圖 1-4 中，每個灰條代表一次對服務的請求，其高度表示請求花費了多長時間。大多數請求是相當快的，但偶爾會出現需要更長的時間的異常值。這也許是因為緩慢的請求實質上開銷更大，例如它們可能會處理更多的資料。但即使（你認為）所有請求都花費相同時間的情況下，隨機的附加延遲也會導致結果變化，例如：上下文切換到後臺程序，網路資料包丟失與 TCP 重傳，垃圾收集暫停，強制從磁碟讀取的頁面錯誤，伺服器機架中的震動【18】，還有很多其他原因。

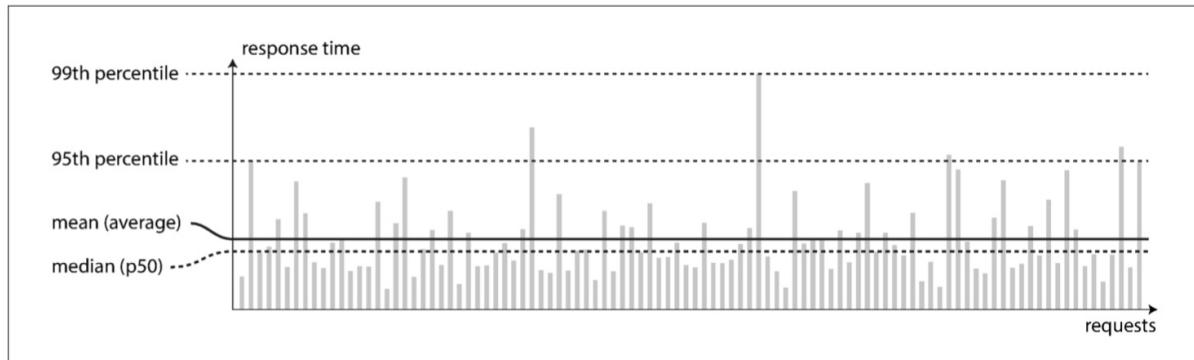


圖 1-4 展示了一個服務 100 次請求響應時間的均值與百分位數

通常報表都會展示服務的平均響應時間。（嚴格來講“平均”一詞並不指代任何特定公式，但實際上它通常被理解為 算術平均值 (arithmetic mean)：給定 n 個值，加起來除以 n ）。然而如果你想知道“典型 (typical)”響應時間，那麼平均值並不是一個非常好的指標，因為它不能告訴你有多少使用者實際上經歷了這個延遲。

通常使用 百分位點 (percentiles) 會更好。如果將響應時間列表按最快到最慢排序，那麼 中位數 (median) 就在正中間：舉個例子，如果你的響應時間中位數是 200 毫秒，這意味著一半請求的返回時間少於 200 毫秒，另一半比這個要長。

如果想知道典型場景下使用者需要等待多長時間，那麼中位數是一個好的度量標準：一半使用者請求的響應時間少於響應時間的中位數，另一半服務時間比中位數長。中位數也被稱為第 50 百分位點，有時縮寫為 p50。注意中位數是關於單個請求的；如果使用者同時發出幾個請求（在一個會話過程中，或者由於一個頁面中包含了多個資源），則至少一個請求比中位數慢的機率遠大於 50%。

為了弄清異常值有多糟糕，可以看看更高的百分位點，例如第 95、99 和 99.9 百分位點（縮寫為 p95, p99 和 p999）。它們意味著 95%、99% 或 99.9% 的請求響應時間要比該閾值快，例如：如果第 95 百分位點響應時間是 1.5 秒，則意味著 100 個請求中的 95 個響應時間快於 1.5 秒，而 100 個請求中的 5 個響應時間超過 1.5 秒。如 圖 1-4 所示。

響應時間的高百分位點（也稱為 尾部延遲，即 tail latencies）非常重要，因為它們直接影響使用者的服務體驗。例如亞馬遜在描述內部服務的響應時間要求時是以 99.9 百分位點為準，即使它隻影響一千個請求中的一個。這是因為請求響應最慢的客戶往往也是資料最多的客戶，也可以說是最有價值的客戶——因為他們掏錢了【19】。保證網站響應迅速對於保持客戶的滿意度非常重要，亞馬遜觀察到：響應時間增加 100 毫秒，銷售量就減少 1%【20】；而另一些報告說：慢 1 秒鐘會讓客戶滿意度指標減少 16%【21, 22】。

另一方面，最佳化第 99.99 百分位點（一萬個請求中最慢的一個）被認為太昂貴了，不能為亞馬遜的目標帶來足夠好處。減小高百分位點處的響應時間相當困難，因為它很容易受到隨機事件的影響，這超出了控制範圍，而且效益也很小。

百分位點通常用於 **服務級別目標 (SLO, service level objectives)** 和 **服務級別協議 (SLA, service level agreements)**，即定義服務預期效能和可用性的合同。SLA 可能會宣告，如果服務響應時間的中位數小於 200 毫秒，且 99.9 百分位點低於 1 秒，則認為服務工作正常（如果響應時間更長，就認為服務不達標）。這些指標為客戶設定了期望值，並允許客戶在 SLA 未達標的情況下要求退款。

排隊延遲 (queueing delay) 通常佔了高百分位點處響應時間的很大一部分。由於伺服器只能並行處理少量的事務（如受其 CPU 核數的限制），所以只要有少量緩慢的請求就能阻礙後續請求的處理，這種效應有時被稱為 **頭部阻塞 (head-of-line blocking)**。即使後續請求在伺服器上處理的非常迅速，由於需要等待先前請求完成，客戶端最終看到的是緩慢的總體響應時間。因為存在這種效應，測量客戶端的響應時間非常重要。

為測試系統的可伸縮性而人為產生負載時，產生負載的客戶端要獨立於響應時間不斷傳送請求。如果客戶端在傳送下一個請求之前等待先前的請求完成，這種行為會產生人為排隊的效果，使得測試時的佇列比現實情況更短，使測量結果產生偏差【23】。

實踐中的百分位點

在多重呼叫的後端服務裡，高百分位數變得特別重要。即使並行呼叫，終端使用者請求仍然需要等待最慢的並行呼叫完成。如 圖 1-5 所示，只需要一個緩慢的呼叫就可以使整個終端使用者請求變慢。即使只有一小部分後端呼叫速度較慢，如果終端使用者請求需要多個後端呼叫，則獲得較慢呼叫的機會也會增加，因此較高比例的終端使用者請求速度會變慢（效果稱為尾部延遲放大【24】）。

如果你想將響應時間百分點新增到你的服務的監視儀表板，則需要持續有效地計算它們。例如，你可能希望在最近 10 分鐘內保持請求響應時間的滾動視窗。每一分鐘，你都會計算出該視窗中的中值和各種百分數，並將這些度量值繪製在圖上。

簡單的實現是在時間視窗內儲存所有請求的響應時間列表，並且每分鐘對列表進行排序。如果對你來說效率太低，那麼有一些演算法能夠以最小的 CPU 和記憶體成本（如前向衰減【25】、t-digest【26】或 HdrHistogram【27】）來計算百分位數的近似值。請注意，平均百分比（例如，減少時間解析度或合併來自多臺機器的資料）在數學上沒有意義 - 聚合響應時間資料的正確方法是新增直方圖【28】。

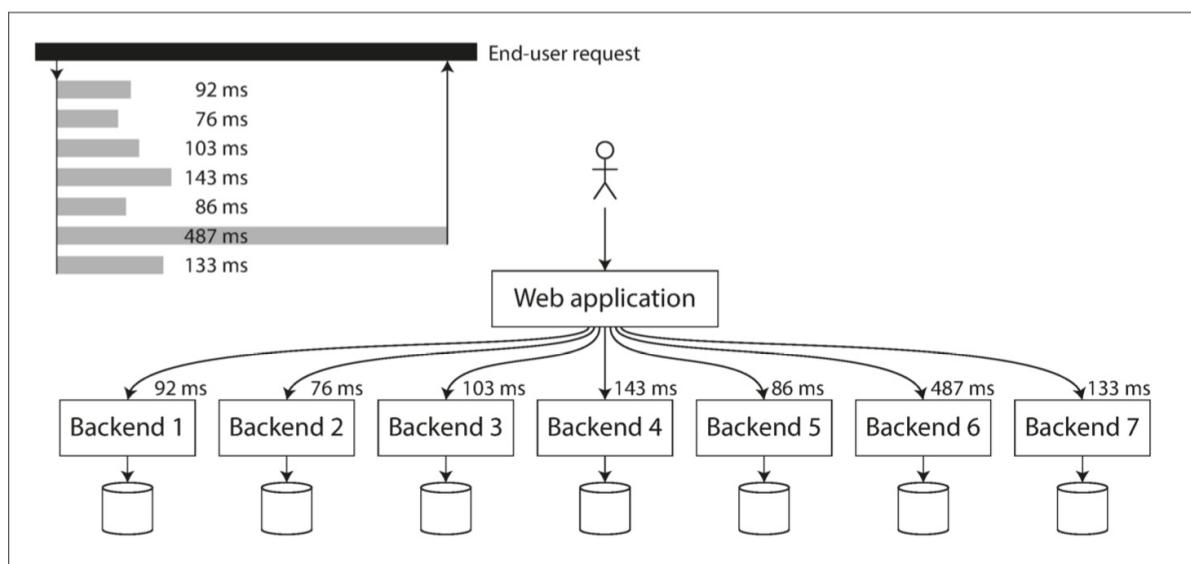


圖 1-5 當一個請求需要多個後端請求時，單個後端慢請求就會拖慢整個終端使用者的請求

應對負載的方法

現在我們已經討論了用於描述負載的引數和用於衡量效能的指標。可以開始認真討論可伸縮性了：當負載引數增加時，如何保持良好的效能？

適應某個級別負載的架構不太可能應付 10 倍於此的負載。如果你正在開發一個快速增長的服務，那麼每次負載發生數量級的增長時，你可能都需要重新考慮架構——或者更頻繁。

人們經常討論 **縱向伸縮** (scaling up, 也稱為垂直伸縮，即 vertical scaling，轉向更強大的機器) 和 **橫向伸縮** (scaling out, 也稱為水平伸縮，即 horizontal scaling，將負載分佈到多臺小機器上) 之間的對立。跨多臺機器分配負載也稱為“**無共享 (shared-nothing)**”架構。可以在單臺機器上執行的系統通常更簡單，但高階機器可能非常貴，所以非常密集的負載通常無法避免地需要橫向伸縮。現實世界中的優秀架構需要將這兩種方法務實地結合，因為使用幾臺足夠強大的機器可能比使用大量的小型虛擬機器更簡單也更便宜。

有些系統是 **彈性 (elastic)** 的，這意味著可以在檢測到負載增加時自動增加計算資源，而其他系統則是手動伸縮（人工分析容量並決定向系統新增更多的機器）。如果負載 **極難預測 (highly unpredictable)**，則彈性系統可能很有用，但手動伸縮系統更簡單，並且意外操作可能會更少（請參閱“[分割槽再平衡](#)”）。

跨多臺機器部署 **無狀態服務 (stateless services)** 非常簡單，但將帶狀態的資料系統從單節點變為分散式配置則可能引入許多額外複雜度。出於這個原因，常識告訴我們應該將資料庫放在單個節點上（縱向伸縮），直到伸縮成本或可用性需求迫使其改為分散式。

隨著分散式系統的工具和抽象越來越好，至少對於某些型別的應用而言，這種常識可能會改變。可以預見分散式資料系統將成為未來的預設設定，即使對不處理大量資料或流量的場景也如此。本書的其餘部分將介紹多種分散式資料系統，不僅討論它們在可伸縮性方面的表現，還包括易用性和可維護性。

大規模的系統架構通常是應用特定的——沒有一招鮮吃遍天的通用可伸縮架構（不正式的叫法：**萬金油 (magic scaling sauce)**）。應用的問題可能是讀取量、寫入量、要儲存的資料量、資料的複雜度、響應時間要求、訪問模式或者所有問題的大雜燴。

舉個例子，用於處理每秒十萬個請求（每個大小為 1 kB）的系統與用於處理每分鐘 3 個請求（每個大小為 2GB）的系統看上去會非常不一樣，儘管兩個系統有同樣的資料吞吐量。

一個良好適配應用的可伸縮架構，是圍繞著 **假設 (assumption)** 建立的：哪些操作是常見的？哪些操作是罕見的？這就是所謂負載引數。如果假設最終是錯誤的，那麼為伸縮所做的工程投入就白費了，最糟糕的是適得其反。在早期創業公司或非正式產品中，通常支援產品快速迭代的能力，要比可伸縮至未來的假想負載要重要的多。

儘管這些架構是應用程式特定的，但可伸縮的架構通常也是從通用的積木塊搭建而成的，並以常見的模式排列。在本書中，我們將討論這些構件和模式。

可維護性

眾所周知，軟體的大部分開銷並不在最初的開發階段，而是在持續的維護階段，包括修復漏洞、保持系統正常執行、調查失效、適配新的平臺、為新的場景進行修改、償還技術債和新增新的功能。

不幸的是，許多從事軟體系統行業的人不喜歡維護所謂的 **遺留 (legacy)** 系統，——也許因為涉及修復其他人的錯誤、和過時的平臺打交道，或者系統被迫使用於一些份外工作。每一個遺留系統都以自己的方式讓人不爽，所以很難給出一個通用的建議來和它們打交道。

但是我們可以，也應該以這樣一種方式來設計軟體：在設計之初就儘量考慮儘可能減少維護期間的痛苦，從而避免自己的軟體系統變成遺留系統。為此，我們將特別關注軟體系統的三個設計原則：

- 可操作性 (Operability)

便於運維團隊保持系統平穩執行。

- 簡單性 (Simplicity)

從系統中消除儘可能多的 **複雜度 (complexity)**，使新工程師也能輕鬆理解系統（注意這和使用者介面的簡單性不一樣）。

- 可演化性 (evolvability)

使工程師在未來能輕鬆地對系統進行更改，當需求變化時為新應用場景做適配。也稱為 **可擴充套件性 (extensibility)**、**可修改性 (modifiability)** 或 **可塑性 (plasticity)**。

和之前提到的可靠性、可伸縮性一樣，實現這些目標也沒有簡單的解決方案。不過我們會試著想象具有可操作性，簡單性和可演化性的系統會是什麼樣子。

可操作性：人生苦短，關愛運維

有人認為，“良好的運維經常可以繞開垃圾（或不完整）軟體的侷限性，而再好的軟體攤上垃圾運維也沒法可靠執行”。儘管運維的某些方面可以，而且應該是自動化的，但在最初建立正確運作的自動化機制仍然取決於人。

運維團隊對於保持軟體系統順利執行至關重要。一個優秀運維團隊的典型職責如下（或者更多）【29】：

- 監控系統的執行狀況，並在服務狀態不佳時快速恢復服務。
- 跟蹤問題的原因，例如系統故障或效能下降。
- 及時更新軟體和平臺，比如安全補丁。
- 瞭解系統間的相互作用，以便在異常變更造成損失前進行規避。
- 預測未來的問題，並在問題出現之前加以解決（例如，容量規劃）。
- 建立部署、配置、管理方面的良好實踐，編寫相應工具。
- 執行複雜的維護任務，例如將應用程式從一個平臺遷移到另一個平臺。
- 當配置變更時，維持系統的安全性。
- 定義工作流程，使運維操作可預測，並保持生產環境穩定。
- 鐵打的營盤流水的兵，維持組織對系統的瞭解。

良好的可操作性意味著更輕鬆的日常工作，進而運維團隊能專注於高價值的事情。資料系統可以透過各種方式使日常任務更輕鬆：

- 透過良好的監控，提供對系統內部狀態和執行時行為的 **可見性 (visibility)**。
- 為自動化提供良好支援，將系統與標準化工具相整合。
- 避免依賴單臺機器（在整個系統繼續不間斷執行的情況下允許機器停機維護）。
- 提供良好的文件和易於理解的操作模型（“如果做 X，會發生 Y”）。
- 提供良好的預設行為，但需要時也允許管理員自由覆蓋預設值。
- 有條件時進行自我修復，但需要時也允許管理員手動控制系統狀態。
- 行為可預測，最大限度減少意外。

簡單性：管理複雜度

小型軟體專案可以使用簡單討喜的、富表現力的程式碼，但隨著專案越來越大，程式碼往往變得非常複雜，難以理解。這種複雜度拖慢了所有系統相關人員，進一步增加了維護成本。一個陷入複雜泥潭的軟體專案有時被描述為 **爛泥潭 (a big ball of mud)** 【30】。

複雜度 (complexity) 有各種可能的症狀，例如：狀態空間激增、模組間緊密耦合、糾結的依賴關係、不一致的命名和術語、解決效能問題的 Hack、需要繞開的特例等等，現在已經有很多關於這個話題的討論【31,32,33】。

因為複雜度導致維護困難時，預算和時間安排通常會超支。在複雜的軟體中進行變更，引入錯誤的風險也更大：當開發人員難以理解系統時，隱藏的假設、無意的後果和意外的互動就更容易被忽略。相反，降低複雜度能極大地提高軟體的可維護性，因此簡單性應該是構建系統的一個關鍵目標。

簡化系統並不一定意味著減少功能；它也可以意味著消除 **額外的 (accidental)** 的複雜度。Moseley 和 Marks 【32】把 **額外複雜度** 定義為：由具體實現中湧現，而非（從使用者視角看，系統所解決的）問題本身固有的複雜度。

用於消除 **額外複雜度** 的最好工具之一是 **抽象 (abstraction)**。一個好的抽象可以將大量實現細節隱藏在一個乾淨，簡單易懂的外觀下面。一個好的抽象也可以廣泛用於各類不同應用。比起重複造很多輪子，重用抽象不僅更有效率，而且有助於開發高質量的軟體。抽象元件的質量改進將使所有使用它的應用受益。

例如，高階程式語言是一種抽象，隱藏了機器碼、CPU 暫存器和系統呼叫。SQL 也是一種抽象，隱藏了複雜的磁碟 / 記憶體資料結構、來自其他客戶端的併發請求、崩潰後的不一致性。當然在用高階語言程式設計時，我們仍然用到了機器碼；只不過沒有 **直接（directly）** 使用罷了，正是因為程式語言的抽象，我們才不必去考慮這些實現細節。

抽象可以幫助我們將系統的複雜度控制在可管理的水平，不過，找到好的抽象是非常困難的。在分散式系統領域雖然有許多好的演算法，但我們並不清楚它們應該打包成什麼樣抽象。

本書將緊盯那些允許我們將大型系統的部分提取為定義明確的、可重用的元件的優秀抽象。

可演化性：擁抱變化

系統的需求永遠不變，基本是不可能的。更可能的情況是，它們處於常態的變化中，例如：你瞭解了新的事實、出現意想不到的應用場景、業務優先順序發生變化、使用者要求新功能、新平臺取代舊平臺、法律或監管要求發生變化、系統增長迫使架構變化等。

在組織流程方面，**敏捷（agile）** 工作模式為適應變化提供了一個框架。敏捷社群還開發了對在頻繁變化的環境中開發軟體很有幫助的技術工具和模式，如 **測試驅動開發（TDD, test-driven development）** 和 **重構（refactoring）** 。

這些敏捷技術的大部分討論都集中在相當小的規模（同一個應用中的幾個程式碼檔案）。本書將探索在更大資料系統層面上提高敏捷性的方法，可能由幾個不同的應用或服務組成。例如，為了將裝配主頁時間線的方法從方法 1 變為方法 2，你會如何“重構”推特的架構？

修改資料系統並使其適應不斷變化需求的容易程度，是與 **簡單性** 和 **抽象性** 密切相關的：簡單易懂的系統通常比複雜系統更容易修改。但由於這是一個非常重要的概念，我們將用一個不同的詞來指代資料系統層面的敏捷性：**可演化性（evolvability）** 【34】。

本章小結

本章探討了一些關於資料密集型應用的基本思考方式。這些原則將指導我們閱讀本書的其餘部分，那裡將會深入技術細節。

一個應用必須滿足各種需求才稱得上有用。有一些 **功能需求（functional requirements）**，即它應該做什麼，比如允許以各種方式儲存、檢索、搜尋和處理資料）以及一些 **非功能性需求（nonfunctional）**，即通用屬性，例如安全性、可靠性、合規性、可伸縮性、相容性和可維護性）。在本章詳細討論了可靠性、可伸縮性和可維護性。

可靠性（Reliability） 意味著即使發生故障，系統也能正常工作。故障可能發生在硬體（通常是隨機的和不相關的）、軟體（通常是系統性的 Bug，很難處理）和人類（不可避免地時不時出錯）。**容錯技術** 可以對終端使用者隱藏某些型別的故障。

可伸縮性（Scalability） 意味著即使在負載增加的情況下也有保持效能的策略。為了討論可伸縮性，我們首先需要定量描述負載和效能的方法。我們簡要了解了推特主頁時間線的例子，介紹描述負載的方法，並將響應時間百分位點作為衡量效能的一種方式。在可伸縮的系統中可以新增 **處理容量（processing capacity）** 以在高負載下保持可靠。

可維護性（Maintainability） 有許多方面，但實質上是關於工程師和運維團隊的生活質量的。良好的抽象可以幫助降低複雜度，並使系統易於修改和適應新的應用場景。良好的可操作性意味著對系統的健康狀態具有良好的可見性，並擁有有效的管理手段。

不幸的是，使應用可靠、可伸縮或可維護不容易。但是某些模式和技術會不斷重新出現在不同的應用中。在接下來的幾章中，我們將看到一些資料系統的例子，並分析它們如何實現這些目標。

在本書後面的 [第三部分](#) 中，我們將看到一種模式：幾個元件協同工作以構成一個完整的系統（如 [圖 1-1](#) 中的例子）

參考文獻

1. Michael Stonebraker and Uğur Çetintemel: “One Size Fits All”: An Idea Whose Time Has Come and Gone,” at

- 21st International Conference on Data Engineering (ICDE), April 2005.
- 2. Walter L. Heimerdinger and Charles B. Weinstock: “[A Conceptual Framework for System Fault Tolerance](#),” Technical Report CMU/SEI-92-TR-033, Software Engineering Institute, Carnegie Mellon University, October 1992.
 - 3. Ding Yuan, Yu Luo, Xin Zhuang, et al.: “[Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems](#),” at 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2014.
 - 4. Yury Izrailevsky and Ariel Tseitlin: “[The Netflix Simian Army](#),” techblog.netflix.com, July 19, 2011.
 - 5. Daniel Ford, François Labelle, Florentina I. Popovici, et al.: “[Availability in Globally Distributed Storage Systems](#),” at 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2010.
 - 6. Brian Beach: “[Hard Drive Reliability Update – Sep 2014](#),” backblaze.com, September 23, 2014.
 - 7. Laurie Voss: “[AWS: The Good, the Bad and the Ugly](#),” blog.ewe.sm, December 18, 2012.
 - 8. Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, et al.: “[What Bugs Live in the Cloud?](#),” at 5th ACM Symposium on Cloud Computing (SoCC), November 2014. doi:[10.1145/2670979.2670986](https://doi.org/10.1145/2670979.2670986)
 - 9. Nelson Minar: “[Leap Second Crashes Half the Internet](#),” somebits.com, July 3, 2012.
 - 10. Amazon Web Services: “[Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region](#),” aws.amazon.com, April 29, 2011.
 - 11. Richard I. Cook: “[How Complex Systems Fail](#),” Cognitive Technologies Laboratory, April 2000.
 - 12. Jay Kreps: “[Getting Real About Distributed System Reliability](#),” blog.empathybox.com, March 19, 2012.
 - 13. David Oppenheimer, Archana Ganapathi, and David A. Patterson: “[Why Do Internet Services Fail, and What Can Be Done About It?](#),” at 4th USENIX Symposium on Internet Technologies and Systems (USITS), March 2003.
 - 14. Nathan Marz: “[Principles of Software Engineering, Part 1](#),” nathanmarz.com, April 2, 2013.
 - 15. Michael Jurewitz: “[The Human Impact of Bugs](#),” jury.me, March 15, 2013.
 - 16. Raffi Krikorian: “[Timelines at Scale](#),” at QCon San Francisco, November 2012.
 - 17. Martin Fowler: *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002. ISBN: 978-0-321-12742-6
 - 18. Kelly Sommers: “[After all that run around, what caused 500ms disk latency even when we replaced physical server?](#)” twitter.com, November 13, 2014.
 - 19. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al.: “[Dynamo: Amazon's Highly Available Key-Value Store](#),” at 21st ACM Symposium on Operating Systems Principles (SOSP), October 2007.
 - 20. Greg Linden: “[Make Data Useful](#),” slides from presentation at Stanford University Data Mining class (CS345), December 2006.
 - 21. Tammy Everts: “[The Real Cost of Slow Time vs Downtime](#),” webperformancetoday.com, November 12, 2014.
 - 22. Jake Brutlag: “[Speed Matters for Google Web Search](#),” googleresearch.blogspot.co.uk, June 22, 2009.
 - 23. Tyler Treat: “[Everything You Know About Latency Is Wrong](#),” bravenewgeek.com, December 12, 2015.
 - 24. Jeffrey Dean and Luiz André Barroso: “[The Tail at Scale](#),” *Communications of the ACM*, volume 56, number 2, pages 74–80, February 2013. doi:[10.1145/2408776.2408794](https://doi.org/10.1145/2408776.2408794)
 - 25. Graham Cormode, Vladislav Shkapenyuk, Divesh Srivastava, and Bojian Xu: “[Forward Decay: A Practical Time Decay Model for Streaming Systems](#),” at 25th IEEE International Conference on Data Engineering (ICDE), March 2009.
 - 26. Ted Dunning and Otmar Ertl: “[Computing Extremely Accurate Quantiles Using t-Digests](#),” github.com, March 2014.
 - 27. Gil Tene: “[HdrHistogram](#),” hdrhistogram.org.
 - 28. Baron Schwartz: “[Why Percentiles Don't Work the Way You Think](#),” vividcortex.com, December 7, 2015.
 - 29. James Hamilton: “[On Designing and Deploying Internet-Scale Services](#),” at 21st Large Installation System Administration Conference (LISA), November 2007.
 - 30. Brian Foote and Joseph Yoder: “[Big Ball of Mud](#),” at 4th Conference on Pattern Languages of Programs (PLoP), September 1997.
 - 31. Frederick P Brooks: “No Silver Bullet – Essence and Accident in Software Engineering,” in *The Mythical Man-Month*, Anniversary edition, Addison-Wesley, 1995. ISBN: 978-0-201-83595-3
 - 32. Ben Moseley and Peter Marks: “[Out of the Tar Pit](#),” at BCS Software Practice Advancement (SPA), 2006.
 - 33. Rich Hickey: “[Simple Made Easy](#),” at Strange Loop, September 2011.

34. Hongyu Pei Breivold, Ivica Crnkovic, and Peter J. Eriksson: “[Analyzing Software Evolvability](#),” at *32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, July 2008.
[doi:10.1109/COMPSAC.2008.50](https://doi.org/10.1109/COMPSAC.2008.50)
-

上一章	目錄	下一章
第一部分：資料系統基礎	設計資料密集型應用	第二章：資料模型與查詢語言

第二章：資料模型與查詢語言



語言的邊界就是思想的邊界。

——路德維奇·維特根斯坦，《邏輯哲學》（1922）

[TOC]

資料模型可能是軟體開發中最重要的部分了，因為它們的影響如此深遠：不僅僅影響著軟體的編寫方式，而且影響著我們的解題思路。

多數應用使用層層疊加的資料模型構建。對於每層資料模型的關鍵問題是：它是如何用低一層資料模型來表示的？例如：

1. 作為一名應用開發人員，你觀察現實世界（裡面有人員、組織、貨物、行為、資金流向、感測器等），並採用物件或資料結構，以及操控那些資料結構的 API 來進行建模。那些結構通常是特定於應用程式的。
2. 當要儲存那些資料結構時，你可以利用通用資料模型來表示它們，如 JSON 或 XML 文件、關係資料庫中的表或圖模型。
3. 資料庫軟體的工程師選定如何以記憶體、磁碟或網路上的位元組來表示 JSON / XML / 關係 / 圖資料。這類表示形式使資料有可能以各種方式來查詢，搜尋，操縱和處理。
4. 在更低的層次上，硬體工程師已經想出了使用電流、光脈衝、磁場或者其他東西來表示位元組的方法。

一個複雜的應用程式可能會有更多的中間層次，比如基於 API 的 API，不過基本思想仍然是一樣的：每個層都透過提供一個明確的資料模型來隱藏更低層次中的複雜性。這些抽象允許不同的人群有效地協作（例如資料庫廠商的工程師和使用資料庫的應用程式開發人員）。

資料模型種類繁多，每個資料模型都帶有如何使用的設想。有些用法很容易，有些則不支援如此；有些操作執行很快，有些則表現很差；有些資料轉換非常自然，有些則很麻煩。

掌握一個數據模型需要花費很多精力（想想關係資料建模有多少本書）。即便只使用一個數據模型，不用操心其內部工作機制，構建軟體也是非常困難的。然而，因為資料模型對上層軟體的功能（能做什麼，不能做什麼）有著至深的影響，所以選擇一個適合的資料模型是非常重要的。

在本章中，我們將研究一系列用於資料儲存和查詢的通用資料模型（前面列表中的第 2 點）。特別地，我們將比較關係模型，文件模型和少量基於圖形的資料模型。我們還將檢視各種查詢語言並比較它們的用例。在 [第三章](#) 中，我們將討論儲存引擎是如何工作的。也就是說，這些資料模型實際上是如何實現的（列表中的第 3 點）。

關係模型與文件模型

現在最著名的資料模型可能是 SQL。它基於 Edgar Codd 在 1970 年提出的關係模型 [【1】](#)：資料被組織成 關係（SQL 中稱作 表），其中每個關係是 元組（SQL 中稱作 行）的無序集合。

關係模型曾是一個理論性的提議，當時很多人都懷疑是否能夠有效實現它。然而到了 20 世紀 80 年代中期，關係資料庫管理系統（RDBMSes）和 SQL 已成為大多數人們儲存和查詢某些常規結構的資料的首選工具。關係資料庫已經持續稱霸了大約 25~30 年——這對計算機史來說是極其漫長的時間。

關係資料庫起源於商業資料處理，在 20 世紀 60 年代和 70 年代用大型計算機來執行。從今天的角度來看，那些用例顯得很平常：典型的 事務處理（將銷售或銀行交易，航空公司預訂，庫存管理資訊記錄在庫）和 批處理（客戶發票，工資單，報告）。

當時的其他資料庫迫使應用程式開發人員必須考慮資料庫內部的資料表示形式。關係模型致力於將上述實現細節隱藏在更簡潔的介面之後。

多年來，在資料儲存和查詢方面存在著許多相互競爭的方法。在 20 世紀 70 年代和 80 年代初，網狀模型（network model）和層次模型（hierarchical model）曾是主要的選擇，但關係模型（relational model）隨後佔據了主導地位。物件資料庫在 20 世紀 80 年代末和 90 年代初來了又去。XML 資料庫在二十一世紀初出現，但只有小眾採用過。關係模型的每個競爭者都在其時代產生了大量的炒作，但從來沒有持續 [【2】](#)。

隨著電腦越來越強大和互聯，它們開始用於日益多樣化的目的。關係資料庫非常成功地被推廣到業務資料處理的原始範圍之外更為廣泛的用例上。你今天在網上看到的大部分內容依舊是由關係資料庫來提供支援，無論是線上釋出、討論、社交網路、電子商務、遊戲、軟體即服務生產力應用程式等內容。

NoSQL 的誕生

現在 - 2010 年代，NoSQL 開始了最新一輪嘗試，試圖推翻關係模型的統治地位。“NoSQL”這個名字讓人遺憾，因為實際上它並沒有涉及到任何特定的技術。最初它只是作為一個醒目的 Twitter 標籤，用在 2009 年一個關於分散式，非關係資料庫上的開源聚會上。無論如何，這個術語觸動了某些神經，並迅速在網路創業社群內外傳播開來。好些有趣的資料庫系統現在都與 #NoSQL 標籤相關聯，並且 NoSQL 被追溯性地重新解釋為 不僅是 SQL（Not Only SQL） [【4】](#)。

採用 NoSQL 資料庫的背後有幾個驅動因素，其中包括：

- 需要比關係資料庫更好的可伸縮性，包括非常大的資料集或非常高的寫入吞吐量
- 相比商業資料庫產品，免費和開源軟體更受偏愛
- 關係模型不能很好地支援一些特殊的查詢操作
- 受挫於關係模型的限制性，渴望一種更具多動態性與表現力的資料模型 [【5】](#)

不同的應用程式有不同的需求，一個用例的最佳技術選擇可能不同於另一個用例的最佳技術選擇。因此，在可預見的未來，關係資料庫似乎可能會繼續與各種非關係資料庫一起使用。這種想法有時也被稱為 **混合持久化**（polyglot persistence）。

物件關係不匹配

目前大多數應用程式開發都使用面向物件的程式語言來開發，這導致了對 SQL 資料模型的普遍批評：如果資料儲存在關係表中，那麼需要一個笨拙的轉換層，處於應用程式程式碼中的物件和表，行，列的資料庫模型之間。模型之間的不連貫時被稱為 **阻抗不匹配 (impedance mismatch)**ⁱ。

ⁱ 一個從電子學借用的術語。每個電路的輸入和輸出都有一定的阻抗（交流電阻）。當你將一個電路的輸出連線到另一個電路的輸入時，如果兩個電路的輸出和輸入阻抗匹配，則連線上的功率傳輸將被最大化。阻抗不匹配會導致訊號反射及其他問題。[←](#)

像 ActiveRecord 和 Hibernate 這樣的 **物件關係對映 (ORM object-relational mapping)** 框架可以減少這個轉換層所需的樣板程式碼的數量，但是它們不能完全隱藏這兩個模型之間的差異。

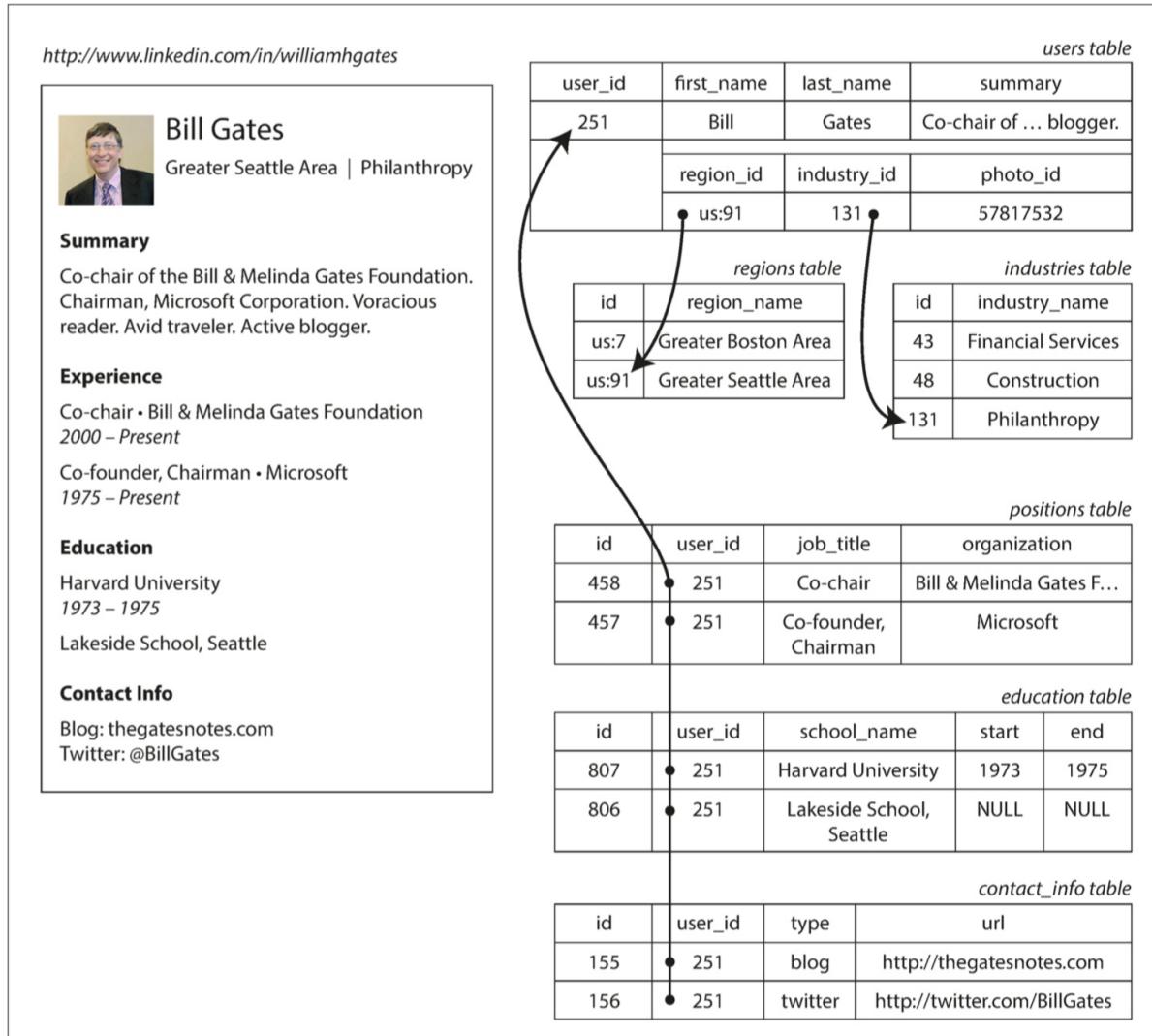


圖 2-1 使用關係型模式來表示領英簡介

例如，[圖 2-1](#) 展示瞭如何在關係模式中表示簡歷（一個 LinkedIn 簡介）。整個簡歷可以透過一個唯一的識別符號 `user_id` 來標識。像 `first_name` 和 `last_name` 這樣的欄位每個使用者只出現一次，所以可以在 User 表上將其建模為列。但是，大多數人在職業生涯中擁有多於一份的工作，人們可能有不同樣的教育階段和任意數量的聯絡資訊。從使用者到這些專案之間存在一對多的關係，可以用多種方式來表示：

- 傳統 SQL 模型（SQL：1999 之前）中，最常見的規範化表示形式是將職位，教育和聯絡資訊放在單獨的表中，對 User 表提供外來鍵引用，如 [圖 2-1](#) 所示。
- 後續的 SQL 標準增加了對結構化資料型別和 XML 資料的支援；這允許將多值資料儲存在單行內，並支援在這些文件內查詢和索引。這些功能在 Oracle，IBM DB2，MS SQL Server 和 PostgreSQL 中都有不同程度的支援
【6,7】。JSON 資料型別也得到多個數據庫的支援，包括 IBM DB2，MySQL 和 PostgreSQL 【8】。

- 第三種選擇是將職業，教育和聯絡資訊編碼為 JSON 或 XML 文件，將其儲存在資料庫的文字列中，並讓應用程式解析其結構和內容。這種配置下，通常不能使用資料庫來查詢該編碼列中的值。

對於一個像簡歷這樣自包含文件的資料結構而言，JSON 表示是非常合適的：請參閱 [例 2-1](#)。JSON 比 XML 更簡單。面向文件的資料庫（如 MongoDB 【9】，RethinkDB 【10】，CouchDB 【11】和 Espresso 【12】）支援這種資料模型。

例 2-1. 用 JSON 文件表示一個 LinkedIn 簡介

```
{
  "user_id": 251,
  "first_name": "Bill",
  "last_name": "Gates",
  "summary": "Co-chair of the Bill & Melinda Gates... Active blogger.",
  "region_id": "us:91",
  "industry_id": 131,
  "photo_url": "/p/7/000/253/05b/308dd6e.jpg",
  "positions": [
    {
      "job_title": "Co-chair",
      "organization": "Bill & Melinda Gates Foundation"
    },
    {
      "job_title": "Co-founder, Chairman",
      "organization": "Microsoft"
    }
  ],
  "education": [
    {
      "school_name": "Harvard University",
      "start": 1973,
      "end": 1975
    },
    {
      "school_name": "Lakeside School, Seattle",
      "start": null,
      "end": null
    }
  ],
  "contact_info": {
    "blog": "http://thegatesnotes.com",
    "twitter": "http://twitter.com/BillGates"
  }
}
```

有一些開發人員認為 JSON 模型減少了應用程式程式碼和儲存層之間的阻抗不匹配。不過，正如我們將在 [第四章](#) 中看到的那樣，JSON 作為資料編碼格式也存在問題。缺乏一個模式往往被認為是一個優勢；我們將在 “[文件模型中的模式靈活性](#)” 中討論這個問題。

JSON 表示比 [圖 2-1](#) 中的多表模式具有更好的 **區域性 (locality)**。如果在前面的關係型示例中獲取簡介，那需要執行多個查詢（透過 `user_id` 查詢每個表），或者在 User 表與其下屬表之間混亂地執行多路連線。而在 JSON 表示中，所有相關資訊都在同一個地方，一個查詢就足夠了。

從使用者簡介檔案到使用者職位，教育歷史和聯絡資訊，這種一對多關係隱含了資料中的一個樹狀結構，而 JSON 表示使得這個樹狀結構變得明確（見 [圖 2-2](#)）。

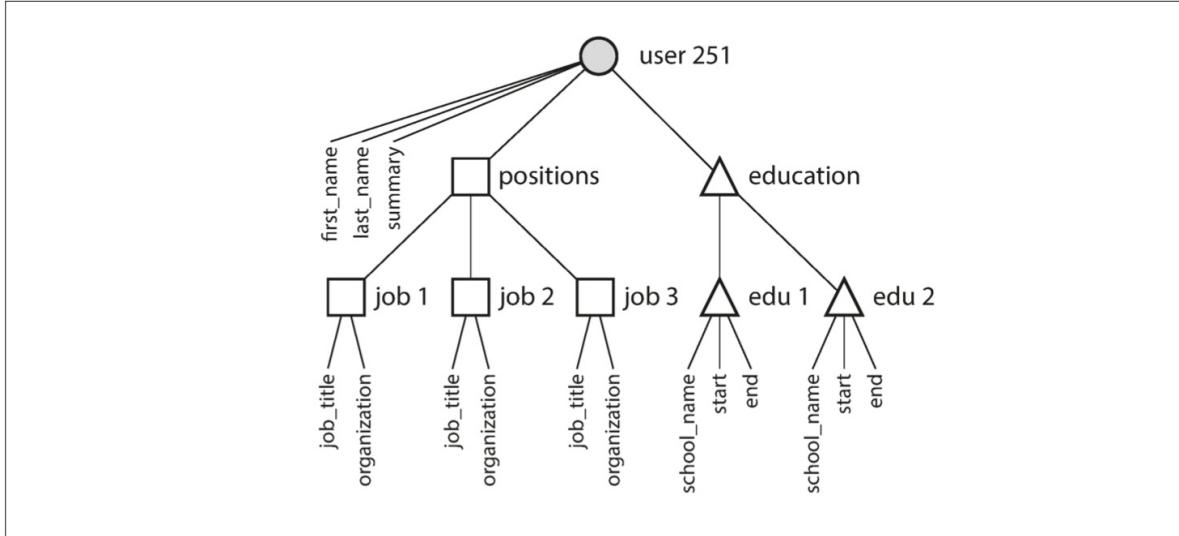


圖 2-2 一對多關係構建了一個樹結構

多對一和多對多的關係

在上一節的 [例 2-1](#) 中，`region_id` 和 `industry_id` 是以 ID，而不是純文字字串 “Greater Seattle Area” 和 “Philanthropy”的形式給出的。為什麼？

如果使用者介面用一個自由文字欄位來輸入區域和行業，那麼將他們儲存為純文字字串是合理的。另一方式是給出地理區域和行業的標準化的列表，並讓使用者從下拉列表或自動填充器中進行選擇，其優勢如下：

- 各個簡介之間樣式和拼寫統一
- 避免歧義（例如，如果有幾個同名的城市）
- 易於更新——名稱只儲存在一個地方，如果需要更改（例如，由於政治事件而改變城市名稱），很容易進行全面更新。
- 本地化支援——當網站翻譯成其他語言時，標準化的列表可以被本地化，使得地區和行業可以使用使用者的語言來顯示
- 更好的搜尋——例如，搜尋華盛頓州的慈善家就會匹配這份簡介，因為地區列表可以編碼記錄西雅圖在華盛頓這一事實（從 “Greater Seattle Area” 這個字串中看不出來）

儲存 ID 還是文字字串，這是個 **副本 (duplication)** 問題。當使用 ID 時，對人類有意義的資訊（比如單詞：`Philanthropy`）只儲存在一處，所有引用它的地方使用 ID（ID 只在資料庫中有意義）。當直接儲存文字時，對人類有意義的資訊會複製在每處使用記錄中。

使用 ID 的好處是，ID 對人類沒有任何意義，因而永遠不需要改變：ID 可以保持不變，即使它標識的資訊發生變化。任何對人類有意義的東西都可能需要在將來某個時候改變——如果這些資訊被複制，所有的冗餘副本都需要更新。這會導致寫入開銷，也存在不一致的風險（一些副本被更新了，還有些副本沒有被更新）。去除此類重複是資料庫 **規範化 (normalization)** 的關鍵思想。ⁱⁱ

ⁱⁱ. 關於關係模型的文獻區分了幾種不同的規範形式，但這些區別幾乎沒有實際意義。一個經驗法則是，如果重複儲存了可以儲存在一個地方的值，則模式就不是 **規範化 (normalized)** 的。 ↵

資料庫管理員和開發人員喜歡爭論規範化和非規範化，讓我們暫時保留判斷吧。在本書的 [第三部分](#)，我們將回到這個話題，探討系統的方法用以處理快取，非規範化和衍生資料。

不幸的是，對這些資料進行規範化需要多對一的關係（許多人生活在一個特定的地區，許多人在一個特定的行業工作），這與文件模型不太吻合。在關係資料庫中，透過 ID 來引用其他表中的行是正常的，因為連線很容易。在文件資料庫中，一對多樹結構沒有必要用連線，對連線的支援通常很弱ⁱⁱⁱ。

ⁱⁱⁱ. 在撰寫本文時，RethinkDB 支援連線，MongoDB 不支援連線，而 CouchDB 只支援預先宣告的檢視。 ↵

如果資料庫本身不支援連線，則必須在應用程式程式碼中透過對資料庫進行多個查詢來模擬連線。（在這種情況中，地區和行業的列表可能很小，改動很少，應用程式可以簡單地將其儲存在記憶體中。不過，執行連線的工作從資料庫被轉移到應用程式程式碼上。）

此外，即便應用程式的最初版本適合無連線的文件模型，隨著功能新增到應用程式中，資料會變得更加互聯。例如，考慮一下對簡歷例子進行的一些修改：

- 組織和學校作為實體

在前面的描述中，`organization`（使用者工作的公司）和`school_name`（他們學習的地方）只是字串。也許他們應該是對實體的引用呢？然後，每個組織、學校或大學都可以擁有自己的網頁（標識、新聞提要等）。每個簡歷可以連結到它所提到的組織和學校，並且包括他們的圖示和其他資訊（請參閱 [圖 2-3](#)，來自 LinkedIn 的一個例子）。

- 推薦

假設你想新增一個新的功能：一個使用者可以為另一個使用者寫一個推薦。在使用者的簡歷上顯示推薦，並附上推薦使用者的姓名和照片。如果推薦人更新他們的照片，那他們寫的任何推薦都需要顯示新的照片。因此，推薦應該擁有作者個人簡介的引用。

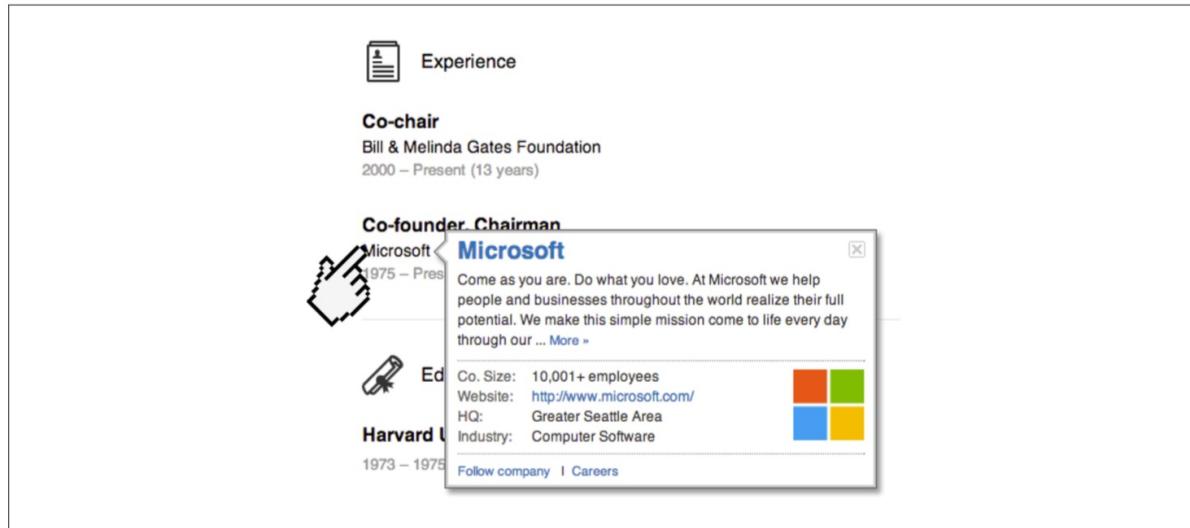


圖 2-3 公司名不僅是字串，還是一個指向公司實體的連結（LinkedIn 截圖）

圖 2-4 闡明瞭這些新功能需要如何使用多對多關係。每個虛線矩形內的資料可以分組成一個文件，但是對單位，學校和其他使用者的引用需要表示成引用，並且在查詢時需要連線。

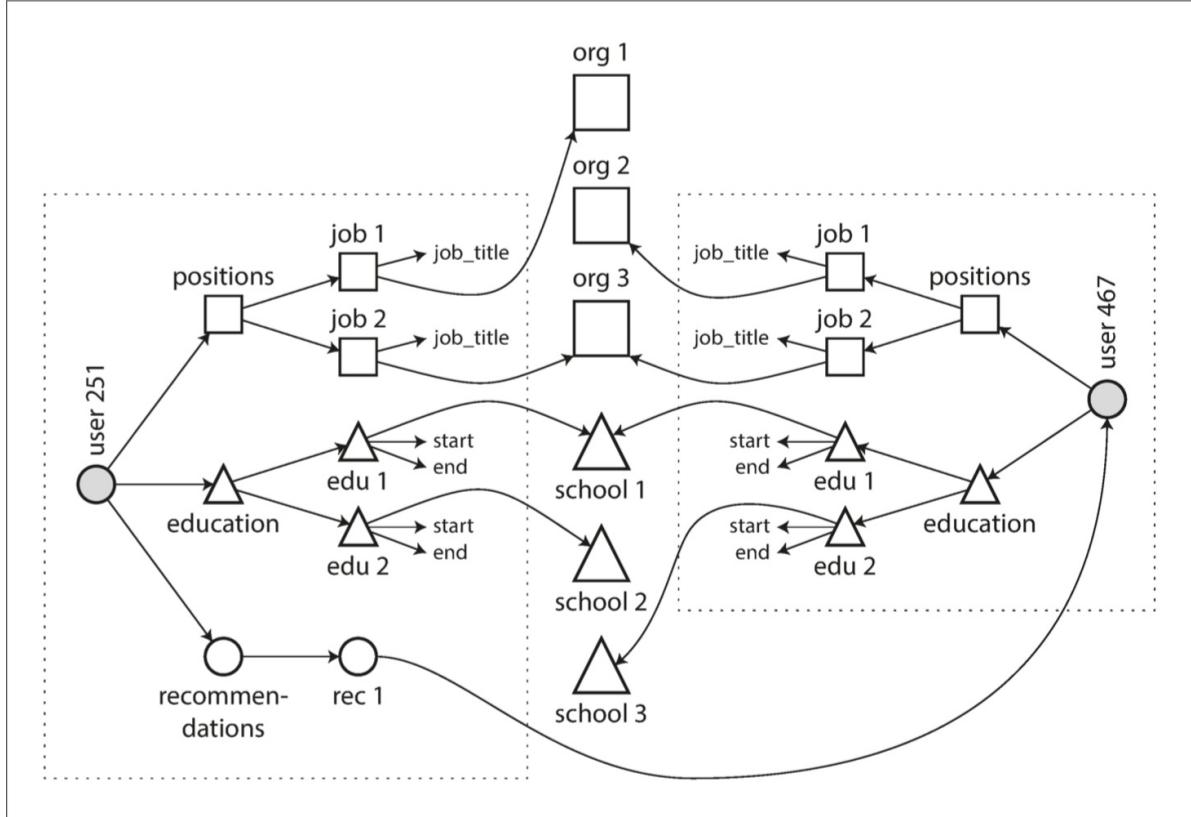


圖 2-4 使用多對多關係擴充套件簡歷

文件資料庫是否在重蹈覆轍？

在多對多的關係和連線已常規用在關係資料庫時，文件資料庫和 NoSQL 重啟了辯論：如何以最佳方式在資料庫中表示多對多關係。那場辯論可比 NoSQL 古老得多，事實上，最早可以追溯到計算機化資料庫系統。

20 世紀 70 年代最受歡迎的業務資料處理資料庫是 IBM 的資訊管理系統（IMS），最初是為了阿波羅太空計劃的庫存管理而開發的，並於 1968 年有了首次商業釋出【13】。目前它仍在使用和維護，執行在 IBM 大型機的 OS/390 上【14】。

IMS 的設計中使用了一個相當簡單的資料模型，稱為 **層次模型**（hierarchical model），它與文件資料庫使用的 JSON 模型有一些驚人的相似之處【2】。它將所有資料表示為巢狀在記錄中的記錄樹，這很像 圖 2-2 的 JSON 結構。

同文檔資料庫一樣，IMS 能良好處理一對多的關係，但是很難應對多對多的關係，並且不支援連線。開發人員必須決定是否複製（非規範化）資料或手動解決從一個記錄到另一個記錄的引用。這些二十世紀六七十年代的問題與現在開發人員遇到的文件資料庫問題非常相似【15】。

那時人們提出了各種不同的解決方案來解決層次模型的侷限性。其中最突出的兩個是 **關係模型**（relational model，它變成了 SQL，並統治了世界）和 **網狀模型**（network model，最初很受關注，但最終變得冷門）。這兩個陣營之間的“大辯論”在 70 年代持續了很久時間【2】。

那兩個模式解決的問題與當前的問題相關，因此值得簡要回顧一下那場辯論。

網狀模型

網狀模型由一個稱為資料系統語言會議（CODASYL）的委員會進行了標準化，並被數個不同的資料庫廠商實現；它也被稱為 CODASYL 模型【16】。

CODASYL 模型是層次模型的推廣。在層次模型的樹結構中，每條記錄只有一個父節點；在網路模式中，每條記錄可能有多個父節點。例如，“Greater Seattle Area” 地區可能是一條記錄，每個居住在該地區的使用者都可以與之相關聯。這允許對多對一和多對多的關係進行建模。

網狀模型中記錄之間的連結不是外來鍵，而更像程式語言中的指標（同時仍然儲存在磁碟上）。訪問記錄的唯一方法是跟隨從根記錄起沿這些鏈路所形成的路徑。這被稱為 **訪問路徑 (access path)**。

最簡單的情況下，訪問路徑類似遍歷連結串列：從列表頭開始，每次檢視一條記錄，直到找到所需的記錄。但在多對多關係的情況中，數條不同的路徑可以到達相同的記錄，網狀模型的程式設計師必須跟蹤這些不同的訪問路徑。

CODASYL 中的查詢是透過利用遍歷記錄列和跟隨訪問路徑表在資料庫中移動遊標來執行的。如果記錄有多個父結點（即多個來自其他記錄的傳入指標），則應用程式程式碼必須跟蹤所有的各種關係。甚至 CODASYL 委員會成員也承認，這就像在 n 維資料空間中進行導航【17】。

儘管手動選擇訪問路徑能夠最有效地利用 20 世紀 70 年代非常有限的硬體功能（如磁帶驅動器，其搜尋速度非常慢），但這使得查詢和更新資料庫的程式碼變得複雜不靈活。無論是分層還是網狀模型，如果你沒有所需資料的路徑，就會陷入困境。你可以改變訪問路徑，但是必須瀏覽大量手寫資料庫查詢程式碼，並重寫來處理新的訪問路徑。更改應用程式的資料模型是很難的。

關係模型

相比之下，關係模型做的就是將所有的資料放在光天化日之下：一個 **關係 (表)** 只是一個 **元組 (行)** 的集合，僅此而已。如果你想讀取資料，它沒有迷宮似的巢狀結構，也沒有複雜的訪問路徑。你可以選中符合任意條件的行，讀取表中的任何或所有行。你可以透過指定某些列作為匹配關鍵字來讀取特定行。你可以在任何表中插入一個新的行，而不必擔心與其他表的外來鍵關係^{iv}。

^{iv}. 外來鍵約束允許對修改進行限制，但對於關係模型這並不是必選項。即使有約束，外來鍵連線在查詢時執行，而在 CODASYL 中，連線在插入時高效完成。 ↪

在關係資料庫中，查詢最佳化器自動決定查詢的哪些部分以哪個順序執行，以及使用哪些索引。這些選擇實際上是“訪問路徑”，但最大的區別在於它們是由查詢最佳化器自動生成的，而不是由程式設計師生成，所以我們很少需要考慮它們。

如果想按新的方式查詢資料，你可以宣告一個新的索引，查詢會自動使用最合適的那些索引。無需更改查詢來利用新的索引（請參閱“[資料查詢語言](#)”）。關係模型因此使新增應用程式新功能變得更加容易。

關係資料庫的查詢最佳化器是複雜的，已耗費了多年的研究和開發精力【18】。關係模型的一個關鍵洞察是：只需構建一次查詢最佳化器，隨後使用該資料庫的所有應用程式都可以從中受益。如果你沒有查詢最佳化器的話，那麼為特定查詢手動編寫訪問路徑比編寫通用最佳化器更容易——不過從長期看通用解決方案更好。

與文件資料庫相比

在一個方面，文件資料庫還原為層次模型：在其父記錄中儲存巢狀記錄（[圖 2-1](#) 中的一對多關係，如 `positions`，`education` 和 `contact_info`），而不是在單獨的表中。

但是，在表示多對一和多對多的關係時，關係資料庫和文件資料庫並沒有根本的不同：在這兩種情況下，相關專案都被一個唯一的識別符號引用，這個識別符號在關係模型中被稱為 **外來鍵**，在文件模型中稱為 **文件引用**【9】。該識別符號在讀取時透過連線或後續查詢來解析。迄今為止，文件資料庫沒有走 CODASYL 的老路。

關係型資料庫與文件資料庫在今日的對比

將關係資料庫與文件資料庫進行比較時，可以考慮許多方面的差異，包括它們的容錯屬性（請參閱 [第五章](#)）和處理併發性（請參閱 [第七章](#)）。本章將只關注資料模型中的差異。

支援文件資料模型的主要論據是架構靈活性，因區域性而擁有更好的效能，以及對於某些應用程式而言更接近於應用程式使用的資料結構。關係模型透過為連線提供更好的支援以及支援多對一和多對多的關係來反擊。

哪種資料模型更有助於簡化應用程式碼？

如果應用程式中的資料具有類似文件的結構（即，一對多關係樹，通常一次性載入整個樹），那麼使用文件模型可能是一個好主意。將類似文件的結構分解成多個表（如 圖 2-1 中的 `positions`、`education` 和 `contact_info`）的關係技術可能導致繁瑣的模式和不必要的複雜的應用程式程式碼。

文件模型有一定的侷限性：例如，不能直接引用文件中的巢狀的專案，而是需要說“使用者 251 的位置列表中的第二項”（很像層次模型中的訪問路徑）。但是，只要檔案巢狀不太深，這通常不是問題。

文件資料庫對連線的糟糕支援可能是個問題，也可能不是問題，這取決於應用程式。例如，如果某分析型應用程式使用一個文件資料庫來記錄何時何地發生了何事，那麼多對多關係可能永遠也用不上。【19】。

但如果你的應用程式確實會用到多對多關係，那麼文件模型就沒有那麼誘人了。儘管可以透過反規範化來消除對連線的需求，但這需要應用程式程式碼來做額外的工作以確保資料一致性。儘管應用程式程式碼可以透過向資料庫發出多個請求的方式來模擬連線，但這也將複雜性轉移到應用程式中，而且通常也會比由資料庫內的專用程式碼更慢。在這種情況下，使用文件模型可能會導致更複雜的應用程式碼與更差的效能【15】。

我們沒有辦法說哪種資料模型更有助於簡化應用程式碼，因為它取決於資料項之間的關係種類。對高度關聯的資料而言，文件模型是極其糟糕的，關係模型是可以接受的，而選用圖形模型（請參閱“[圖資料模型](#)”）是最自然的。

文件模型中的模式靈活性

大多數文件資料庫以及關係資料庫中的 JSON 支援都不會強制文件中的資料採用何種模式。關係資料庫的 XML 支援通常帶有可選的模式驗證。沒有模式意味著可以將任意的鍵和值新增到文件中，並且當讀取時，客戶端無法保證文件可能包含的欄位。

文件資料庫有時稱為 **無模式（schemaless）**，但這具有誤導性，因為讀取資料的程式碼通常假定某種結構——即存在隱式模式，但不由資料庫強制執行【20】。一個更精確的術語是 **讀時模式**（即 schema-on-read，資料的結構是隱含的，只有在資料被讀取時才被解釋），相應的是 **寫時模式**（即 schema-on-write，傳統的關係資料庫方法中，模式明確，且資料庫確保所有的資料都符合其模式）【21】。

讀時模式類似於程式語言中的動態（執行時）型別檢查，而寫時模式類似於靜態（編譯時）型別檢查。就像靜態和動態型別檢查的相對優點具有很大的爭議性一樣【22】，資料庫中模式的強制性是一個具有爭議的話題，一般來說沒有正確或錯誤的答案。

在應用程式想要改變其資料格式的情況下，這些方法之間的區別尤其明顯。例如，假設你把每個使用者的全名儲存在一個欄位中，而現在想分別儲存名字和姓氏【23】。在文件資料庫中，只需開始寫入具有新欄位的新文件，並在應用程式中使用程式碼來處理讀取舊文件的情況。例如：

```
if (user && user.name && !user.first_name) {
    // Documents written before Dec 8, 2013 don't have first_name
    user.first_name = user.name.split(" ")[0];
}
```

另一方面，在“靜態型別”資料庫模式中，通常會執行以下 **遷移（migration）** 操作：

```
ALTER TABLE users ADD COLUMN first_name text;
UPDATE users SET first_name = split_part(name, ' ', 1);      -- PostgreSQL
UPDATE users SET first_name = substring_index(name, ' ', 1);   -- MySQL
```

模式變更的速度很慢，而且要求停運。它的這種壞名譽並不是完全應得的：大多數關係資料庫系統可在幾毫秒內執行 `ALTER TABLE` 語句。MySQL 是一個值得注意的例外，它執行 `ALTER TABLE` 時會複製整個表，這可能意味著在更改一個大型表時會花費幾分鐘甚至幾個小時的停機時間，儘管存在各種工具來解決這個限制【24,25,26】。

大型表上執行 `UPDATE` 語句在任何資料庫上都可能會很慢，因為每一行都需要重寫。要是不可接受的話，應用程式可以將 `first_name` 設定為預設值 `NULL`，並在讀取時再填充，就像使用文件資料庫一樣。

當由於某種原因（例如，資料是異構的）集合中的專案並不都具有相同的結構時，讀時模式更具優勢。例如，如果：

- 存在許多不同型別的物件，將每種型別的物件放在自己的表中是不現實的。
- 資料的結構由外部系統決定。你無法控制外部系統且它隨時可能變化。

在上述情況下，模式的壞處遠大於它的幫助，無模式文件可能是一個更加自然的資料模型。但是，要是所有記錄都具有相同的結構，那麼模式是記錄並強制這種結構的有效機制。第四章將更詳細地討論模式和模式演化。

查詢的資料區域性

文件通常以單個連續字串形式進行儲存，編碼為 JSON、XML 或其二進位制變體（如 MongoDB 的 BSON）。如果應用程式經常需要訪問整個文件（例如，將其渲染至網頁），那麼儲存區域性會帶來效能優勢。如果將資料分割到多個表中（如 圖 2-1 所示），則需要進行多次索引查詢才能將其全部檢索出來，這可能需要更多的磁碟查詢並花費更多的時間。

區域性僅僅適用於同時需要文件絕大部分內容的情況。資料庫通常需要載入整個文件，即使只訪問其中的一小部分，這對於大型文件來說是很浪費的。更新文件時，通常需要整個重寫。只有不改變文件大小的修改才可以容易地原地執行。因此，通常建議保持相對小的文件，並避免增加文件大小的寫入【9】。這些效能限制大大減少了文件資料庫的實用場景。

值得指出的是，為了區域性而分組集合相關資料的想法並不侷限於文件模型。例如，Google 的 Spanner 資料庫在關係資料模型中提供了同樣的區域性屬性，允許模式宣告一個表的行應該交錯（巢狀）在父表內【27】。Oracle 類似地允許使用一個稱為 **多表索引叢集表** (**multi-table index cluster tables**) 的類似特性【28】。Bigtable 資料模型（用於 Cassandra 和 HBase）中的 **列族** (**column-family**) 概念與管理區域性的目的類似【29】。

在 第三章 將還會看到更多關於區域性的內容。

文件和關係資料庫的融合

自 2000 年代中期以來，大多數關係資料庫系統（MySQL 除外）都已支援 XML。這包括對 XML 文件進行本地修改的功能，以及在 XML 文件中進行索引和查詢的功能。這允許應用程式使用那種與文件資料庫應當使用的非常類似的資料模型。

從 9.3 版本開始的 PostgreSQL 【8】，從 5.7 版本開始的 MySQL 以及從版本 10.5 開始的 IBM DB2 【30】也對 JSON 文件提供了類似的支援級別。鑑於用在 Web APIs 的 JSON 流行趨勢，其他關係資料庫很可能會跟隨他們的腳步並新增 JSON 支援。

在文件資料庫中，RethinkDB 在其查詢語言中支援類似關係的連線，一些 MongoDB 驅動程式可以自動解析資料庫引用（有效地執行客戶端連線，儘管這可能比在資料庫中執行的連線慢，需要額外的網路往返，並且最佳化更少）。

隨著時間的推移，關係資料庫和文件資料庫似乎變得越來越相似，這是一件好事：資料模型相互補充^v，如果一個數據庫能夠處理類似文件的資料，並能夠對其執行關係查詢，那麼應用程式就可以使用最符合其需求的功能組合。

關係模型和文件模型的混合是未來資料庫一條很好的路線。

^v. Codd 對關係模型【1】的原始描述實際上允許在關係模式中與 JSON 文件非常相似。他稱之為 **非簡單域** (**nonsimple domains**)。這個想法是，一行中的值不一定是一個像數字或字串一樣的原始資料型別，也可以是一個巢狀的關係（表），因此可以把一個任意巢狀的樹結構作為一個值，這很像 30 年後新增到 SQL 中的 JSON 或 XML 支援。 ↪

資料查詢語言

當引入關係模型時，關係模型包含了一種查詢資料的新方法：SQL 是一種 **宣告式** 查詢語言，而 IMS 和 CODASYL 使用 **命令式** 程式碼來查詢資料庫。那是什麼意思？

許多常用的程式語言是命令式的。例如，給定一個動物物種的列表，返回列表中的鯊魚可以這樣寫：

```

function getSharks() {
  var sharks = [];
  for (var i = 0; i < animals.length; i++) {
    if (animals[i].family === "Sharks") {
      sharks.push(animals[i]);
    }
  }
  return sharks;
}

```

而在關係代數中，你可以這樣寫：

```

$$ sharks = \sigma_{family = "sharks"}(animals)

$$

```

其中 σ (希臘字母西格瑪) 是選擇運算子，只返回符合 `family="shark"` 條件的動物。

定義 SQL 時，它緊密地遵循關係代數的結構：

```

SELECT * FROM animals WHERE family = 'Sharks';

```

命令式語言告訴計算機以特定順序執行某些操作。可以想象一下，逐行地遍歷程式碼，評估條件，更新變數，並決定是否再迴圈一遍。

在宣告式查詢語言（如 SQL 或關係代數）中，你只需指定所需資料的模式 - 結果必須符合哪些條件，以及如何將資料轉換（例如，排序，分組和集合） - 但不是如何實現這一目標。資料庫系統的查詢最佳化器決定使用哪些索引和哪些連線方法，以及以何種順序執行查詢的各個部分。

宣告式查詢語言是迷人的，因為它通常比命令式 API 更加簡潔和容易。但更重要的是，它還隱藏了資料庫引擎的實現細節，這使得資料庫系統可以在無需對查詢做任何更改的情況下進行效能提升。

例如，在本節開頭所示的命令程式碼中，動物列表以特定順序出現。如果資料庫想要在後臺回收未使用的磁碟空間，則可能需要移動記錄，這會改變動物出現的順序。資料庫能否安全地執行，而不會中斷查詢？

SQL 示例不確保任何特定的順序，因此不在意順序是否改變。但是如果查詢用命令式的程式碼來寫的話，那麼資料庫就永遠不可能確定程式碼是否依賴於排序。SQL 相當有限的功能性為資料庫提供了更多自動最佳化的空間。

最後，宣告式語言往往適合並行執行。現在，CPU 的速度透過核心（core）的增加變得更快，而不是以比以前更高的時鐘速度執行【31】。命令程式碼很難在多個核心和多個機器之間並行化，因為它指定了指令必須以特定順序執行。宣告式語言更具有並行執行的潛力，因為它們僅指定結果的模式，而不指定用於確定結果的演算法。在適當情況下，資料庫可以自由使用查詢語言的並行實現【32】。

Web 上的宣告式查詢

宣告式查詢語言的優勢不僅限於資料庫。為了說明這一點，讓我們在一個完全不同的環境中比較宣告式和命令式方法：一個 Web 瀏覽器。

假設你有一個關於海洋動物的網站。使用者當前正在檢視鯊魚頁面，因此你將當前所選的導航專案“鯊魚”標記為當前選中專案。

```

<ul>
  <li class="selected">
    <p>Sharks</p>
    <ul>
      <li>Great White Shark</li>
      <li>Tiger Shark</li>
      <li>Hammerhead Shark</li>
    </ul>
  </li>
</ul>

```

```

<li><p>Whales</p>
  <ul>
    <li>Blue Whale</li>
    <li>Humpback Whale</li>
    <li>Fin Whale</li>
  </ul>
</li>
</ul>

```

現在想讓當前所選頁面的標題具有一個藍色的背景，以便在視覺上突出顯示。使用 CSS 實現起來非常簡單：

```

li.selected > p {
  background-color: blue;
}

```

這裡的 CSS 選擇器 `li.selected > p` 聲明瞭我們想要應用藍色樣式的元素的模式：即其直接父元素是具有 CSS 類 `selected` 的 `` 元素的所有 `<p>` 元素。示例中的元素 `<p>Sharks</p>` 匹配此模式，但 `<p>Whales</p>` 不匹配，因為其 `` 父元素缺少 `class="selected"`。

如果使用 XSL 而不是 CSS，你可以做類似的事情：

```

<xsl:template match="li[@class='selected']/p">
  <fo:block background-color="blue">
    <xsl:apply-templates/>
  </fo:block>
</xsl:template>

```

這裡的 XPath 表達式 `li[@class='selected']/p` 相當於上例中的 CSS 選擇器 `li.selected > p`。CSS 和 XSL 的共同之處在於，它們都是用於指定文件樣式的宣告式語言。

想象一下，必須使用命令式方法的情況會是如何。在 Javascript 中，使用 **文件物件模型（DOM） API**，其結果可能如下所示：

```

var liElements = document.getElementsByTagName("li");
for (var i = 0; i < liElements.length; i++) {
  if (liElements[i].className === "selected") {
    var children = liElements[i].childNodes;
    for (var j = 0; j < children.length; j++) {
      var child = children[j];
      if (child.nodeType === Node.ELEMENT_NODE && child.tagName === "P") {
        child.setAttribute("style", "background-color: blue");
      }
    }
  }
}

```

這段 JavaScript 程式碼命令式地將元素設定為藍色背景，但是程式碼看起來很糟糕。不僅比 CSS 和 XSL 等價物更長，更難理解，而且還有一些嚴重的問題：

- 如果選定的類被移除（例如，因為使用者點選了不同的頁面），即使程式碼重新執行，藍色背景也不會被移除 - 因此該專案將保持突出顯示，直到整個頁面被重新載入。使用 CSS，瀏覽器會自動檢測 `li.selected > p` 規則何時不再適用，並在選定的類被移除後立即移除藍色背景。
- 如果你想要利用新的 API（例如 `document.getElementsByClassName("selected")` 甚至 `document.evaluate()`）來提高效能，則必須重寫程式碼。另一方面，瀏覽器供應商可以在不破壞相容性的情況下提高 CSS 和 XPath 的效能。

在 Web 瀏覽器中，使用宣告式 CSS 樣式比使用 JavaScript 命令式地操作樣式要好得多。類似地，在資料庫中，使用像 SQL 這樣的宣告式查詢語言比使用命令式查詢 API 要好得多^{vi}。

^{vi}. IMS 和 CODASYL 都使用命令式 API。應用程式通常使用 COBOL 程式碼遍歷資料庫中的記錄，一次一條記

錄【2,16】。 ↫

MapReduce查詢

MapReduce 是一個由 Google 推廣的程式設計模型，用於在多臺機器上批次處理大規模的資料【33】。一些 NoSQL 資料儲存（包括 MongoDB 和 CouchDB）支援有限形式的 MapReduce，作為在多個文件中執行只讀查詢的機制。

關於 MapReduce 更詳細的介紹在 [第十章](#)。現在我們只簡要討論一下 MongoDB 使用的模型。

MapReduce 既不是一個宣告式的查詢語言，也不是一個完全命令式的查詢 API，而是處於兩者之間：查詢的邏輯用程式碼片段來表示，這些程式碼片段會被處理框架重複性呼叫。它基於 `map`（也稱為 `collect`）和 `reduce`（也稱為 `fold` 或 `inject`）函式，兩個函式存在於許多函數語言程式設計語言中。

最好舉例來解釋 MapReduce 模型。假設你是一名海洋生物學家，每當你看到海洋中的動物時，你都會在資料庫中新增一條觀察記錄。現在你想生成一個報告，說明你每月看到多少鯊魚。

在 PostgreSQL 中，你可以像這樣表述這個查詢：

```
SELECT
    date_trunc('month', observation_timestamp) AS observation_month,
    sum(num_animals) AS total_animals
FROM observations
WHERE family = 'Sharks'
GROUP BY observation_month;
```

`date_trunc('month', timestamp)` 函式用於確定包含 `timestamp` 的日曆月份，並返回代表該月份開始的另一個時間戳。換句話說，它將時間戳舍入成最近的月份。

這個查詢首先過濾觀察記錄，以只顯示鯊魚家族的物種，然後根據它們發生的日曆月份對觀察記錄果進行分組，最後將在該月的所有觀察記錄中看到的動物數目加起來。

同樣的查詢用 MongoDB 的 MapReduce 功能可以按如下來表述：

```
db.observations.mapReduce(function map() {
    var year = this.observationTimestamp.getFullYear();
    var month = this.observationTimestamp.getMonth() + 1;
    emit(year + "-" + month, this.numAnimals);
},
function reduce(key, values) {
    return Array.sum(values);
},
{
    query: {
        family: "Sharks"
    },
    out: "monthlySharkReport"
});
```

- 可以宣告式地指定一個只考慮鯊魚種類的過濾器（這是 MongoDB 特定的 MapReduce 擴充套件）。
- 每個匹配查詢的文件都會呼叫一次 JavaScript 函式 `map`，將 `this` 設定為文件物件。
- `map` 函式發出一個鍵（包括年份和月份的字串，如 `"2013-12"` 或 `"2014-1"`）和一個值（該觀察記錄中的動物數量）。
- `map` 發出的鍵值對按鍵來分組。對於具有相同鍵（即，相同的月份和年份）的所有鍵值對，呼叫一次 `reduce` 函式。
- `reduce` 函式將特定月份內所有觀測記錄中的動物數量相加。
- 將最終的輸出寫入到 `monthlySharkReport` 集合中。

例如，假設 `observations` 集合包含這兩個文件：

```
{
  observationTimestamp: Date.parse("Mon, 25 Dec 1995 12:34:56 GMT"),
  family: "Sharks",
  species: "Carcharodon carcharias",
  numAnimals: 3
}
{
  observationTimestamp: Date.parse("Tue, 12 Dec 1995 16:17:18 GMT"),
  family: "Sharks",
  species: "Carcharias taurus",
  numAnimals: 4
}
```

對每個文件都會呼叫一次 `map` 函式，結果將是 `emit("1995-12", 3)` 和 `emit("1995-12", 4)`。隨後，以 `reduce("1995-12", [3,4])` 呼叫 `reduce` 函式，將返回 `7`。

`map` 和 `reduce` 函式在功能上有所限制：它們必須是 純 函式，這意味著它們只使用傳遞給它們的資料作為輸入，它們不能執行額外的資料庫查詢，也不能有任何副作用。這些限制允許資料庫以任何順序執行任何功能，並在失敗時重新執行它們。然而，`map` 和 `reduce` 函式仍然是強大的：它們可以解析字串、呼叫庫函式、執行計算等等。

`MapReduce` 是一個相當底層的程式設計模型，用於計算機叢集上的分散式執行。像 `SQL` 這樣的更高階的查詢語言可以用一系列的 `MapReduce` 操作來實現（見 [第十章](#)），但是也有很多不使用 `MapReduce` 的分散式 `SQL` 實現。請注意，`SQL` 中沒有任何內容限制它在單個機器上執行，而 `MapReduce` 在分散式查詢執行上沒有壟斷權。

能夠在查詢中使用 `JavaScript` 程式碼是高階查詢的一個重要特性，但這不限於 `MapReduce`，一些 `SQL` 資料庫也可以用 `JavaScript` 函式進行擴充套件 [【34】](#)。

`MapReduce` 的一個可用性問題是，必須編寫兩個密切合作的 `JavaScript` 函式，這通常比編寫單個查詢更困難。此外，宣告式查詢語言為查詢最佳化器提供了更多機會來提高查詢的效能。基於這些原因，`MongoDB 2.2` 添加了一種叫做 `聚合管道` 的宣告式查詢語言的支援 [【9】](#)。用這種語言表述鯊魚計數查詢如下所示：

```
db.observations.aggregate([
  { $match: { family: "Sharks" } },
  { $group: {
    _id: {
      year: { $year: "$observationTimestamp" },
      month: { $month: "$observationTimestamp" }
    },
    totalAnimals: { $sum: "$numAnimals" } }
  }]);

```

聚合管道語言的表現力與（前述 `PostgreSQL` 例子的）`SQL` 子集相當，但是它使用基於 `JSON` 的語法而不是 `SQL` 那種接近英文句式的語法；這種差異也許只是口味問題。這個故事的寓意是：`NoSQL` 系統可能會意外發現自己只是重新發明了一套經過喬裝改扮的 `SQL`。

圖資料模型

如我們之前所見，多對多關係是不同資料模型之間具有區別性的重要特徵。如果你的應用程式大多數的關係是一對多關係（樹狀結構化資料），或者大多數記錄之間不存在關係，那麼使用文件模型是合適的。

但是，要是多對多關係在你的資料中很常見呢？關係模型可以處理多對多關係的簡單情況，但是隨著資料之間的連線變得更加複雜，將資料建模為圖形顯得更加自然。

一個圖由兩種物件組成：`頂點` (`vertices`，也稱為 `節點`，即 `nodes`，或 `實體`，即 `entities`)，和 `邊` (`edges`，也稱為 `關係`，即 `relationships`，或 `弧`，即 `arcs`)。多種資料可以被建模為一個圖形。典型的例子包括：

- 社交圖譜

頂點是人，邊指示哪些人彼此認識。

- 網路圖譜

頂點是網頁，邊緣表示指向其他頁面的 HTML 連結。

- 公路或鐵路網路

頂點是交叉路口，邊線代表它們之間的道路或鐵路線。

可以將那些眾所周知的演算法運用到這些圖上：例如，汽車導航系統搜尋道路網路中兩點之間的最短路徑，PageRank 可以用在網絡圖上來確定網頁的流行程度，從而確定該網頁在搜尋結果中的排名。

在剛剛給出的例子中，圖中的所有頂點代表了相同型別的事物（人、網頁或交叉路口）。不過，圖並不侷限於這樣的同類資料：同樣強大地是，圖提供了一種一致的方式，用來在單個數據儲存中儲存完全不同型別的物件。例如，Facebook 維護一個包含許多不同型別的頂點和邊的單個圖：頂點表示人、地點、事件、簽到和使用者的評論；邊表示哪些人是好友、簽到發生在哪裡、誰評論了什麼帖子、誰參與了什麼事件等等【35】。

在本節中，我們將使用 [圖 2-5](#) 所示的示意。它可以從社交網路或系譜資料庫中獲得：它顯示了兩個人，來自愛達荷州的 Lucy 和來自法國 Beaune 的 Alain。他們已婚，住在倫敦。

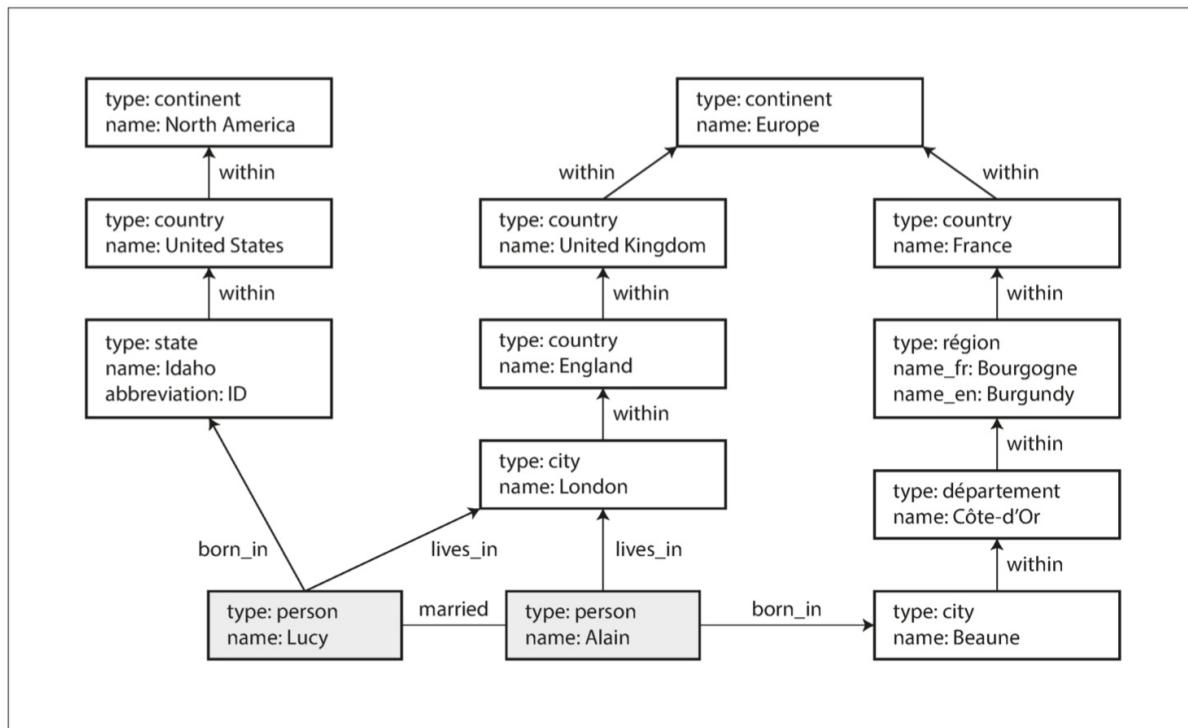


圖 2-5 圖資料結構示意（框代表頂點，箭頭代表邊）

有幾種不同但相關的方法用來構建和查詢圖表中的資料。在本節中，我們將討論屬性圖模型（由 Neo4j，Titan 和 InfiniteGraph 實現）和三元組儲存（triple-store）模型（由 Datomic、AllegroGraph 等實現）。我們將檢視圖的三種宣告式查詢語言：Cypher，SPARQL 和 Datalog。除此之外，還有像 Gremlin 【36】這樣的圖形查詢語言和像 Pregel 這樣的圖形處理框架（見 [第十章](#)）。

屬性圖

在屬性圖模型中，每個頂點（vertex）包括：

- 唯一的識別符號
- 一組出邊（outgoing edges）
- 一組入邊（ingoing edges）

- 一組屬性（鍵值對）

每條邊（edge）包括：

- 唯一識別符號
- 邊的起點（尾部頂點，即 tail vertex）
- 邊的終點（頭部頂點，即 head vertex）
- 描述兩個頂點之間關係型別的標籤
- 一組屬性（鍵值對）

可以將圖儲存看作由兩個關係表組成：一個儲存頂點，另一個儲存邊，如 [例 2-2](#) 所示（該模式使用 PostgreSQL JSON 資料型別來儲存每個頂點或每條邊的屬性）。頭部和尾部頂點用來儲存每條邊；如果你想要一組頂點的輸入或輸出邊，你可以分別透過 `head_vertex` 或 `tail_vertex` 來查詢 `edges` 表。

例 2-2 使用關係模式來表示屬性圖

```
CREATE TABLE vertices (
    vertex_id INTEGER PRIMARY KEY,
    properties JSON
);

CREATE TABLE edges (
    edge_id INTEGER PRIMARY KEY,
    tail_vertex INTEGER REFERENCES vertices (vertex_id),
    head_vertex INTEGER REFERENCES vertices (vertex_id),
    label TEXT,
    properties JSON
);

CREATE INDEX edges_tails ON edges (tail_vertex);
CREATE INDEX edges_heads ON edges (head_vertex);
```

關於這個模型的一些重要方面是：

1. 任何頂點都可以有一條邊連線到任何其他頂點。沒有模式限制哪種事物可不可以關聯。
2. 給定任何頂點，可以高效地找到它的入邊和出邊，從而遍歷圖，即沿著一系列頂點的路徑前後移動（這就是為什麼 [例 2-2](#) 在 `tail_vertex` 和 `head_vertex` 列上都有索引的原因）。
3. 透過對不同型別的關係使用不同的標籤，可以在一個圖中儲存幾種不同的資訊，同時仍然保持一個清晰的資料模型。

這些特性為資料建模提供了很大的靈活性，如 [圖 2-5](#) 所示。圖中顯示了一些傳統關係模式難以表達的事情，例如不同國家的不同地區結構（法國有省和大區，美國有縣和州），國中國的怪事（先忽略主權國家和民族錯綜複雜的爛攤子），不同的資料粒度（Lucy 現在的住所記錄具體到城市，而她的出生地點只是在一個州的級別）。

你可以想象該圖還能延伸出許多關於 Lucy 和 Alain 的事實，或其他人的其他更多的事實。例如，你可以用它來表示食物過敏（為每個過敏源增加一個頂點，並增加人與過敏源之間的一條邊來指示一種過敏情況），並連結到過敏源，每個過敏源具有一組頂點用來顯示哪些食物含有哪些物質。然後，你可以寫一個查詢，找出每個人吃什麼是安全的。圖在可演化性方面是富有優勢的：當你嚮應用程式新增功能時，可以輕鬆擴充套件圖以適應程式資料結構的變化。

Cypher 查詢語言

Cypher 是屬性圖的宣告式查詢語言，為 Neo4j 圖形資料庫而發明 [【37】](#)（它是以電影“駭客帝國”中的一個角色來命名的，而與密碼學中的加密演算法無關 [【38】](#)）。

[例 2-3](#) 顯示了將 [圖 2-5](#) 的左邊部分插入圖形資料庫的 Cypher 查詢。可以類似地新增圖的其餘部分，為了便於閱讀而省略。每個頂點都有一個像 `USA` 或 `Idaho` 這樣的符號名稱，查詢的其他部分可以使用這些名稱在頂點之間建立邊，使用箭頭符號：`(Idaho) - [:WITHIN] -> (USA)` 建立一條標記為 `WITHIN` 的邊，`Idaho` 為尾節點，`USA` 為頭節點。

例 2-3 將圖 2-5 中的資料子集表示為 Cypher 查詢

```

CREATE
(NAmerica:Location {name:'North America', type:'continent'}),
(USA:Location {name:'United States', type:'country' }),
(Idaho:Location {name:'Idaho', type:'state' }),
(Lucy:Person {name:'Lucy' }),
(Idaho) -[:WITHIN]-> (USA) -[:WITHIN]-> (NAmerica),
(Lucy) -[:BORN_IN]-> (Idaho)

```

當 **圖 2-5** 的所有頂點和邊被新增到資料庫後，讓我們提些有趣的問題：例如，找到所有從美國移民到歐洲的人的名字。更確切地說，這裡我們想要找到符合下麵條件的所有頂點，並且返回這些頂點的 `name` 屬性：該頂點擁有一條連到美國任一位置的 `BORN_IN` 邊，和一條連到歐洲的任一位置的 `LIVING_IN` 邊。

例 2-4 展示瞭如何在 Cypher 中表達這個查詢。在 MATCH 子句中使用相同的箭頭符號來查詢圖中的模式：`(person) -[:BORN_IN]-> ()` 可以匹配 `BORN_IN` 邊的任意兩個頂點。該邊的尾節點被綁定了變數 `person`，頭節點則未被繫結。

例 2-4 檢查所有從美國移民到歐洲的人的 Cypher 查詢：

```

MATCH
(person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (us:Location {name:'United States'}),
(person) -[:LIVES_IN]-> () -[:WITHIN*0..]-> (eu:Location {name:'Europe'})
RETURN person.name

```

查詢按如下來解讀：

找到滿足以下兩個條件的所有頂點（稱之為 `person` 頂點）：

- `person` 頂點擁有一條到某個頂點的 `BORN_IN` 出邊。從那個頂點開始，沿著一系列 `WITHIN` 出邊最終到達一個型別為 `Location`，`name` 屬性為 `United States` 的頂點。
- `person` 頂點還擁有一條 `LIVES_IN` 出邊。沿著這條邊，可以透過一系列 `WITHIN` 出邊最終到達一個型別為 `Location`，`name` 屬性為 `Europe` 的頂點。

對於這樣的 `Person` 頂點，返回其 `name` 屬性。

執行這條查詢可能會有幾種可行的查詢路徑。這裡給出的描述建議首先掃描資料庫中的所有人，檢查每個人的出生地和居住地，然後只返回符合條件的那些人。

等價地，也可以從兩個 `Location` 頂點開始反向地查詢。假如 `name` 屬性上有索引，則可以高效地找到代表美國和歐洲的兩個頂點。然後，沿著所有 `WITHIN` 入邊，可以繼續查找出所有在美國和歐洲的位置（州、地區、城市等）。最後，查找出那些可以由 `BORN_IN` 或 `LIVES_IN` 入邊到那些位置頂點的人。

通常對於宣告式查詢語言來說，在編寫查詢語句時，不需要指定執行細節：查詢最佳化程式會自動選擇預測效率最高的策略，因此你可以專注於編寫應用程式的其他部分。

SQL 中的圖查詢

例 2-2 指出，可以在關係資料庫中表示圖資料。但是，如果圖資料已經以關係結構儲存，我們是否也可以使用 SQL 查詢它？

答案是肯定的，但有些困難。在關係資料庫中，你通常會事先知道在查詢中需要哪些連線。在圖查詢中，你可能需要在找到待查詢的頂點之前，遍歷可變數量的邊。也就是說，連線的數量事先並不確定。

在我們的例子中，這發生在 Cypher 查詢中的 `(-) -[:WITHIN*0..]-> ()` 規則中。一個人的 `LIVES_IN` 邊可以指向任何型別的位置：街道、城市、地區、國家等。一個城市可以在 `(WITHIN)` 一個地區內，一個地區可以在 `(WITHIN)` 在一個州內，一個州可以在 `(WITHIN)` 一個國家內，等等。`LIVES_IN` 邊可以直接指向正在查詢的位置，或者一個在位置層次結構中隔了數層的位置。

在 Cypher 中，用 `WITHIN*0..` 非常簡潔地表述了上述事實：“沿著 `WITHIN` 邊，零次或多次”。它很像正則表示式中的 * 運算子。

自 SQL:1999，查詢可變長度遍歷路徑的思想可以使用稱為 **遞迴公用表表達式**（`WITH RECURSIVE` 語法）的東西來表示。

例 2-5 顯示了同樣的查詢 - 查詢從美國移民到歐洲的人的姓名 - 在 SQL 使用這種技術（PostgreSQL、IBM DB2、Oracle 和 SQL Server 均支援）來表述。但是，與 Cypher 相比，其語法非常笨拙。

例 2-5 與示例 2-4 同樣的查詢，在 SQL 中使用遞迴公用表表達式表示

```
WITH RECURSIVE
-- in_usa 包含所有的美國境內的位置 ID
in_usa(vertex_id) AS (
  SELECT vertex_id FROM vertices WHERE properties ->> 'name' = 'United States'
  UNION
  SELECT edges.tail_vertex FROM edges
    JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
    WHERE edges.label = 'within'
),
-- in_europe 包含所有的歐洲境內的位置 ID
in_europe(vertex_id) AS (
  SELECT vertex_id FROM vertices WHERE properties ->> 'name' = 'Europe'
  UNION
  SELECT edges.tail_vertex FROM edges
    JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
    WHERE edges.label = 'within' ),
-- born_in_usa 包含了所有型別為 Person，且出生在美國的頂點
born_in_usa(vertex_id) AS (
  SELECT edges.tail_vertex FROM edges
    JOIN in_usa ON edges.head_vertex = in_usa.vertex_id
    WHERE edges.label = 'born_in' ),
-- lives_in_europe 包含了所有型別為 Person，且居住在歐洲的頂點。
lives_in_europe(vertex_id) AS (
  SELECT edges.tail_vertex FROM edges
    JOIN in_europe ON edges.head_vertex = in_europe.vertex_id
    WHERE edges.label = 'lives_in' )

SELECT vertices.properties ->> 'name'
FROM vertices
  JOIN born_in_usa ON vertices.vertex_id = born_in_usa.vertex_id
  JOIN lives_in_europe ON vertices.vertex_id = lives_in_europe.vertex_id;
```

- 首先，查詢 `name` 屬性為 `United States` 的頂點，將其作為 `in_usa` 頂點的集合的第一個元素。
- 從 `in_usa` 集合的頂點出發，沿著所有的 `with_in` 入邊，將其尾頂點加入同一集合，不斷遞迴直到所有 `with_in` 入邊都被訪問完畢。
- 同理，從 `name` 屬性為 `Europe` 的頂點出發，建立 `in_europe` 頂點的集合。
- 對於 `in_usa` 集合中的每個頂點，根據 `born_in` 入邊來查找出生在美國某個地方的人。
- 同樣，對於 `in_europe` 集合中的每個頂點，根據 `lives_in` 入邊來查詢居住在歐洲的人。
- 最後，把在美國出生的人的集合與在歐洲居住的人的集合相交。

同一個查詢，用某一個查詢語言可以寫成 4 行，而用另一個查詢語言需要 29 行，這恰恰說明了不同的資料模型是為不同的應用場景而設計的。選擇適合應用程式的資料模型非常重要。

三元組儲存和 SPARQL

三元組儲存模式大體上與屬性圖模型相同，用不同的詞來描述相同的想法。不過仍然值得討論，因為三元組儲存有很多現成的工具和語言，這些工具和語言對於構建應用程式的工具箱可能是寶貴的補充。

在三元組儲存中，所有資訊都以非常簡單的三部分表示形式儲存（主語，謂語，賓語）。例如，三元組（吉姆，喜歡，香蕉）中，吉姆 是主語，喜歡 是謂語（動詞），香蕉 是物件。

三元組的主語相當於圖中的一個頂點。而賓語是下面兩者之一：

1. 原始資料型別中的值，例如字串或數字。在這種情況下，三元組的謂語和賓語相當於主語頂點上的屬性的鍵和值。例如，`(lucy, age, 33)` 就像屬性 `{"age": 33}` 的頂點 `lucy`。
2. 圖中的另一個頂點。在這種情況下，謂語是圖中的一條邊，主語是其尾部頂點，而賓語是其頭部頂點。例如，在 `(lucy, marriedTo, alain)` 中主語和賓語 `lucy` 和 `alain` 都是頂點，並且謂語 `marriedTo` 是連線他們的邊的標籤。

例 2-6 展示了與 **例 2-3** 相同的資料，以稱為 Turtle 的格式（Notation3（N3）【39】的一個子集）寫成三元組。

例 2-6 圖 2-5 中的資料子集，表示為 Turtle 三元組

```
@prefix : <urn:example:>.
_:lucy    a      :Person.
_:lucy    :name   "Lucy".
_:lucy    :bornIn _:idaho.
_:idaho   a      :Location.
_:idaho   :name   "Idaho".
_:idaho   :type   "state".
_:idaho   :within _:usa.
_:usa     a      :Location
_:usa     :name   "United States"
_:usa     :type   "country".
_:usa     :within _:namerica.
_:namerica a      :Location
_:namerica :name  "North America"
_:namerica :type   :"continent"
```

在這個例子中，圖的頂點被寫為：`_ :someName`。這個名字並不意味著這個檔案以外的任何東西。它的存在只是幫助我們明確哪些三元組引用了同一頂點。當謂語表示邊時，該賓語是一個頂點，如 `_:idaho :within _:usa`。當謂語是一個屬性時，該賓語是一個字串，如 `_:usa :name "United States"`

一遍又一遍地重複相同的主語看起來相當重複，但幸運的是，可以使用分號來說明關於同一主語的多個事情。這使得 Turtle 格式相當不錯，可讀性強：請參閱 **例 2-7**。

例 2-7 一種相對例 2-6 寫入資料的更為簡潔的方法。

```
@prefix : <urn:example:>.
_:lucy    a :Person;   :name "Lucy";           :bornIn _:idaho.
_:idaho   a :Location; :name "Idaho";          :type "state";   :within _:usa
_:usa     a :Location; :name "United States";  :type "country"; :within _:namerica.
_:namerica a :Location; :name "North America"; :type "continent".
```

語義網

如果你深入瞭解關於三元組儲存的資訊，可能會陷入關於語義網的討論漩渦中。三元組儲存模型其實是完全獨立於語義網存在的，例如，Datomic【40】作為一種三元組儲存資料庫^{vii}，從未被用於語義網中。但是，由於在很多人眼中這兩者緊密相連，我們應該簡要地討論一下。

^{vii} 從技術上講，Datomic 使用的是五元組而不是三元組，兩個額外的欄位是用於版本控制的元資料 ↩

從本質上講，語義網是一個簡單且合理的想法：網站已經將資訊釋出為文字和圖片供人類閱讀，為什麼不將資訊作為機器可讀的資料也釋出給計算機呢？（基於三元組模型的）資源描述框架（RDF）【41】，被用作不同網站以統一的格式釋出資料的一種機制，允許來自不同網站的資料自動合併成一個數據網路——成為一種網際網路範圍內的“通用語義網資料庫”。

不幸的是，語義網在二十一世紀初被過度炒作，但到目前為止沒有任何跡象表明已在實踐中應用，這使得許多人嗤之以鼻。它還飽受眼花繚亂的縮略詞、過於複雜的標準提案和狂妄自大的困擾。

然而，如果從過去的失敗中汲取教訓，語義網專案還是擁有很多優秀的成果。即使你沒有興趣在語義網上釋出 RDF 資料，三元組這種模型也是一種好的應用程式內部資料模型。

RDF 資料模型

例 2-7 中使用的 Turtle 語言是一種用於 RDF 資料的人類可讀格式。有時候，RDF 也可以以 XML 格式編寫，不過完成同樣的事情會相對囉嗦，請參閱 例 2-8。Turtle/N3 是更可取的，因為它更容易閱讀，像 Apache Jena 【42】這樣的工具可以根據需要在不同的 RDF 格式之間進行自動轉換。

例 2-8 用 RDF/XML 語法表示例 2-7 的資料

```

<rdf:RDF xmlns="urn:example:"  
          xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">  
  <Location rdf:nodeID="idaho">  
    <name>Idaho</name>  
    <type>state</type>  
    <within>  
      <Location rdf:nodeID="usa">  
        <name>United States</name>  
        <type>country</type>  
        <within>  
          <Location rdf:nodeID="namerica">  
            <name>North America</name>  
            <type>continent</type>  
          </Location>  
        </within>  
      </Location>  
    </within>  
  </Location>  
  <Person rdf:nodeID="lucy">  
    <name>Lucy</name>  
    <bornIn rdf:nodeID="idaho"/>  
  </Person>  
</rdf:RDF>

```

RDF 有一些奇怪之處，因為它是為了在網際網路上交換資料而設計的。三元組的主語，謂語和賓語通常是 URI。例如，謂語可能是一個 URI，如 `<http://my-company.com/namespace#within>` 或 `<http://my-company.com/namespace#lives_in>`，而不僅僅是 `WITHIN` 或 `LIVES_IN`。這個設計背後的原因為了讓你能夠把你的資料和其他人的資料結合起來，如果他們賦予單詞 `within` 或者 `lives_in` 不同的含義，兩者也不會衝突，因為它們的謂語實際上是 `<http://other.org/foo#within>` 和 `<http://other.org/foo#lives_in>`。

從 RDF 的角度來看，URL `<http://my-company.com/namespace>` 不一定需要能解析成什麼東西，它只是一個名稱空間。為避免與 `http://URL` 混淆，本節中的示例使用不可解析的 URI，如 `urn:example:within`。幸運的是，你只需在檔案頂部對這個字首做一次宣告，後續就不用再管了。

SPARQL 查詢語言

SPARQL 是一種用於三元組儲存的面向 RDF 資料模型的查詢語言【43】（它是 SPARQL 協議和 RDF 查詢語言的縮寫，發音為“sparkle”）。SPARQL 早於 Cypher，並且由於 Cypher 的模式匹配借鑑於 SPARQL，這使得它們看起來非常相似【37】。

與之前相同的查詢——查詢從美國移民到歐洲的人——使用 SPARQL 比使用 Cypher 甚至更為簡潔（請參閱 例 2-9）。

例 2-9 與示例 2-4 相同的查詢，用 SPARQL 表示

```

PREFIX : <urn:example:>  
SELECT ?personName WHERE {  
  ?person :name ?personName.  
  ?person :bornIn / :within* / :name "United States".  
  ?person :livesIn / :within* / :name "Europe".  
}

```

結構非常相似。以下兩個表示式是等價的（SPARQL 中的變數以問號開頭）：

```
(person) -[:BORN_IN]-> () -[:WITHIN*0..]-> (location) # Cypher
?person :bornIn / :within* ?location. # SPARQL
```

因為 RDF 不區分屬性和邊，而只是將它們作為謂語，所以可以使用相同的語法來匹配屬性。在下面的表示式中，變數 `usa` 被繫結到任意 `name` 屬性為字串值 "United States" 的頂點：

```
(usa {name:'United States'}) # Cypher
?usa :name "United States". # SPARQL
```

SPARQL 是一種很好的查詢語言——儘管它構想的語義網從未實現，但它仍然是一種可用於應用程式內部的強大工具。

圖形資料庫與網狀模型相比較

在“[文件資料庫是否在重蹈覆轍？](#)”中，我們討論了 CODASYL 和關係模型如何競相解決 IMS 中的多對多關係問題。乍一看，CODASYL 的網狀模型看起來與圖模型相似。CODASYL 是否是圖形資料庫的第二個變種？

不，他們在幾個重要方面有所不同：

- 在 CODASYL 中，資料庫有一個模式，用於指定哪種記錄型別可以巢狀在其他記錄型別中。在圖形資料庫中，不存在這樣的限制：任何頂點都可以具有到其他任何頂點的邊。這為應用程式適應不斷變化的需求提供了更大的靈活性。
- 在 CODASYL 中，達到特定記錄的唯一方法是遍歷其中的一個訪問路徑。在圖形資料庫中，可以透過其唯一 ID 直接引用任何頂點，也可以使用索引來查詢具有特定值的頂點。
- 在 CODASYL 中，記錄的子專案是一個有序集合，所以資料庫必須去管理它們的次序（這會影響儲存佈局），並且應用程式在插入新記錄到資料庫時必須關注新記錄在這些集合中的位置。在圖形資料庫中，頂點和邊是無序的（只能在查詢時對結果進行排序）。
- 在 CODASYL 中，所有查詢都是命令式的，難以編寫，並且很容易因架構變化而受到破壞。在圖形資料庫中，你可以在命令式程式碼中手寫遍歷過程，但大多數圖形資料庫都支援高階宣告式查詢，如 Cypher 或 SPARQL。

基礎：Datalog

Datalog 是比 SPARQL、Cypher 更古老的語言，在 20 世紀 80 年代被學者廣泛研究 [\[44,45,46\]](#)。它在軟體工程師中不太知名，但是它是重要的，因為它為以後的查詢語言提供了基礎。

實踐中，Datalog 在有限的幾個資料系統中使用：例如，它是 Datomic [\[40\]](#) 的查詢語言，Cascalog [\[47\]](#) 是一種用於查詢 Hadoop 大資料集的 Datalog 實現 ^{viii}。

^{viii}. Datomic 和 Cascalog 使用 Datalog 的 Clojure S 表示式語法。在下面的例子中使用了一個更容易閱讀的 Prolog 語法，但兩者沒有任何功能差異。 ↵

Datalog 的資料模型類似於三元組模式，但進行了一點泛化。把三元組寫成 謂語（主語，賓語），而不是寫三元語（主語，謂語，賓語）。[例 2-10](#) 顯示瞭如何用 Datalog 寫入我們的例子中的資料。

例 2-10 用 Datalog 來表示圖 2-5 中的資料子集

```
name(namerica, 'North America').
type(namerica, continent).

name(usa, 'United States').
type(usa, country).
within(usa, namerica).
```

```

name(idaho, 'Idaho').
type(idaho, state).
within(idaho, usa).

name(lucy, 'Lucy').
born_in(lucy, idaho).

```

既然已經定義了資料，我們可以像之前一樣編寫相同的查詢，如 [例 2-11](#) 所示。它看起來與 Cypher 或 SPARQL 的語法差異較大，但請不要抗拒它。Datalog 是 Prolog 的一個子集，如果你是計算機科學專業的學生，可能已經見過 Prolog。

例 2-11 與示例 2-4 相同的查詢，用 Datalog 表示

```

within_recursive(Location, Name) :- name(Location, Name). /* Rule 1 */

within_recursive(Location, Name) :- within(Location, Via), /* Rule 2 */
    within_recursive(Via, Name).

migrated(Name, BornIn, LivingIn) :- name(Person, Name), /* Rule 3 */
    born_in(Person, BornLoc),
    within_recursive(BornLoc, BornIn),
    lives_in(Person, LivingLoc),
    within_recursive(LivingLoc, LivingIn).

?- migrated(Who, 'United States', 'Europe'). /* Who = 'Lucy'. */

```

Cypher 和 SPARQL 使用 SELECT 立即跳轉，但是 Datalog 一次只進行一小步。我們定義 規則，以將新謂語告訴資料庫：在這裡，我們定義了兩個新的謂語，`within_recursive` 和 `migrated`。這些謂語不是儲存在資料庫中的三元組中，而是從資料或其他規則派生而來的。規則可以引用其他規則，就像函式可以呼叫其他函式或者遞迴地呼叫自己一樣。像這樣，複雜的查詢可以藉由小的磚瓦構建起來。

在規則中，以大寫字母開頭的單詞是變數，謂語則用 Cypher 和 SPARQL 的方式一樣來匹配。例如，`name(Location, Name)` 透過變數繫結 `Location = namerica` 和 `Name = 'North America'` 可以匹配三元組 `name(namerica, 'North America')`。

要是系統可以在 `:-` 運算子的右側找到與所有謂語的一個匹配，就運用該規則。當規則運用時，就好像透過 `:-` 的左側將其新增到資料庫（將變數替換成它們匹配的值）。

因此，一種可能的應用規則的方式是：

1. 資料庫存在 `name(namerica, 'North America')`，故運用規則 1。它生成 `within_recursive(namerica, 'North America')`。
2. 資料庫存在 `within(usa, namerica)`，在上一步驟中生成 `within_recursive(namerica, 'North America')`，故運用規則 2。它會產生 `within_recursive(usa, 'North America')`。
3. 資料庫存在 `within(idaho, usa)`，在上一步生成 `within_recursive(usa, 'North America')`，故運用規則 2。它產生 `within_recursive(idaho, 'North America')`。

透過重複應用規則 1 和 2，`within_recursive` 謂語可以告訴我們在資料庫中包含北美（或任何其他位置名稱）的所有位置。這個過程如 [圖 2-6](#) 所示。

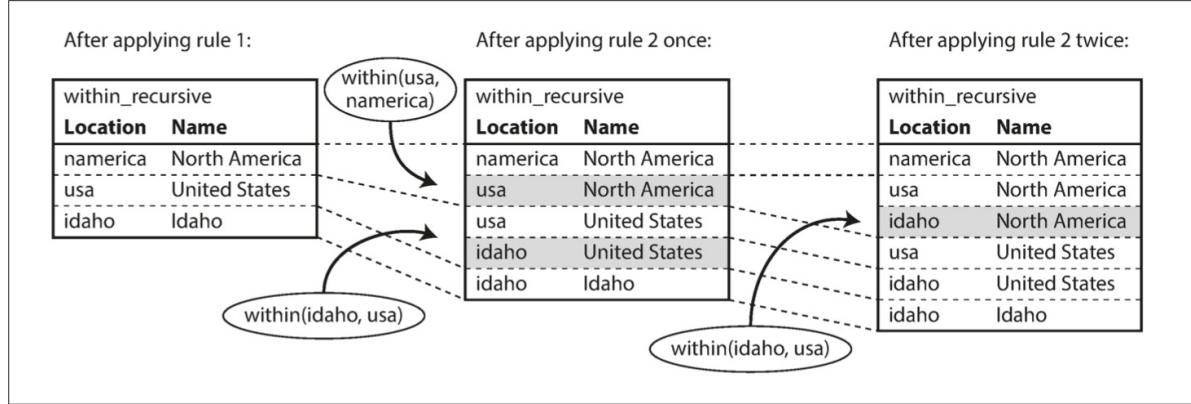


圖 2-6 使用示例 2-11 中的 Datalog 規則來確定愛達荷州在北美。

現在規則 3 可以找到出生在某個地方 `BornIn` 的人，並住在某個地方 `LivingIn`。透過查詢 `BornIn = 'United States'` 和 `LivingIn = 'Europe'`，並將此人作為變數 `Who`，讓 Datalog 系統找出變數 `Who` 會出現哪些值。因此，最後得到了與早先的 Cypher 和 SPARQL 查詢相同的答案。

相對於本章討論的其他查詢語言，我們需要採取不同的思維方式來思考 Datalog 方法，但這是一種非常強大的方法，因為規則可以在不同的查詢中進行組合和重用。雖然對於簡單的一次性查詢，顯得不太方便，但是它可以更好地處理資料很複雜的情況。

本章小結

資料模型是一個巨大的課題，在本章中，我們快速瀏覽了各種不同的模型。我們沒有足夠的篇幅來詳述每個模型的細節，但是希望這個概述足以激起你的興趣，以更多地瞭解最適合你的應用需求的模型。

在歷史上，資料最開始被表示為一棵大樹（層次資料模型），但是這不利於表示多對多的關係，所以發明了關係模型來解決這個問題。最近，開發人員發現一些應用程式也不適合採用關係模型。新的非關係型“NoSQL”資料儲存分化為兩個主要方向：

1. **文件資料庫** 主要關注自我包含的資料文件，而且文件之間的關係非常稀少。
2. **圖形資料庫** 用於相反的場景：任意事物之間都可能存在潛在的關聯。

這三種模型（文件，關係和圖形）在今天都被廣泛使用，並且在各自的領域都發揮很好。一個模型可以用另一個模型來模擬——例如，圖資料可以在關係資料庫中表示——但結果往往是糟糕的。這就是為什麼我們有著針對不同目的的不同系統，而不是一個單一的萬能解決方案。

文件資料庫和圖資料庫有一個共同點，那就是它們通常不會將儲存的資料強制約束為特定模式，這可以使應用程式更容易適應不斷變化的需求。但是應用程式很可能仍會假定資料具有一定的結構；區別僅在於模式是明確的（寫入時強制）還是隱含的（讀取時處理）。

每個資料模型都具有各自的查詢語言或框架，我們討論了幾個例子：SQL，MapReduce，MongoDB 的聚合管道，Cypher，SPARQL 和 Datalog。我們也談到了 CSS 和 XSL/XPath，它們不是資料庫查詢語言，而包含有趣的相似之處。

雖然我們已經覆蓋了很多層面，但仍然有許多資料模型沒有提到。舉幾個簡單的例子：

- 使用基因組資料的研究人員通常需要執行 **序列相似性搜尋**，這意味著需要一個很長的字串（代表一個 DNA 序列），並在一個擁有類似但不完全相同的字串的大型資料庫中尋找匹配。這裡所描述的資料庫都不能處理這種用法，這就是為什麼研究人員編寫了像 GenBank 這樣的專門的基因組資料庫軟體的原因【48】。
- 粒子物理學家數十年來一直在進行大資料型別的大規模資料分析，像大型強子對撞機（LHC）這樣的專案現在會處理數百 PB 的資料！在這樣的規模下，需要定製解決方案來阻止硬體成本的失控【49】。
- **全文搜尋** 可以說是一種經常與資料庫一起使用的資料模型。資訊檢索是一個很大的專業課題，我們不會在本書中詳細介紹，但是我們將在第三章和第三部分中介紹搜尋索引。

讓我們暫時將其放在一邊。在 [下一章](#) 中，我們將討論在 實現 本章描述的資料模型時會遇到的一些權衡。

參考文獻

1. Edgar F. Codd: “[A Relational Model of Data for Large Shared Data Banks](#),” *Communications of the ACM*, volume 13, number 6, pages 377–387, June 1970. doi:[10.1145/362384.362685](https://doi.org/10.1145/362384.362685)
2. Michael Stonebraker and Joseph M. Hellerstein: “[What Goes Around Comes Around](#),” in *Readings in Database Systems*, 4th edition, MIT Press, pages 2–41, 2005. ISBN: 978-0-262-69314-1
3. Pramod J. Sadalage and Martin Fowler: *NoSQL Distilled*. Addison-Wesley, August 2012. ISBN: 978-0-321-82662-6
4. Eric Evans: “[NoSQL: What's in a Name?](#),” *blog.sym-link.com*, October 30, 2009.
5. James Phillips: “[Surprises in Our NoSQL Adoption Survey](#),” *blog.couchbase.com*, February 8, 2012.
6. Michael Wagner: *SQL/XML:2006 – Evaluierung der Standardkonformität ausgewählter Datenbanksysteme*. Diplomica Verlag, Hamburg, 2010. ISBN: 978-3-836-64609-3
7. “[XML Data in SQL Server](#),” SQL Server 2012 documentation, *technet.microsoft.com*, 2013.
8. “[PostgreSQL 9.3.1 Documentation](#),” The PostgreSQL Global Development Group, 2013.
9. “[The MongoDB 2.4 Manual](#),” MongoDB, Inc., 2013.
10. “[RethinkDB 1.11 Documentation](#),” *rethinkdb.com*, 2013.
11. “[Apache CouchDB 1.6 Documentation](#),” *docs.couchdb.org*, 2014.
12. Lin Qiao, Kapil Surlaker, Shirshanka Das, et al.: “[On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform](#),” at *ACM International Conference on Management of Data* (SIGMOD), June 2013.
13. Rick Long, Mark Harrington, Robert Hain, and Geoff Nicholls: *IMS Primer*. IBM Redbook SG24-5352-00, IBM International Technical Support Organization, January 2000.
14. Stephen D. Bartlett: “[IBM's IMS—Myths, Realities, and Opportunities](#),” The Clipper Group Navigator, TCG2013015LI, July 2013.
15. Sarah Mei: “[Why You Should Never Use MongoDB](#),” *sarahmei.com*, November 11, 2013.
16. J. S. Knowles and D. M. R. Bell: “[The CODASYL Model](#),” in *Databases—Role and Structure: An Advanced Course*, edited by P. M. Stocker, P. M. D. Gray, and M. P. Atkinson, pages 19–56, Cambridge University Press, 1984. ISBN: 978-0-521-25430-4
17. Charles W. Bachman: “[The Programmer as Navigator](#),” *Communications of the ACM*, volume 16, number 11, pages 653–658, November 1973. doi:[10.1145/355611.362534](https://doi.org/10.1145/355611.362534)
18. Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton: “[Architecture of a Database System](#),” *Foundations and Trends in Databases*, volume 1, number 2, pages 141–259, November 2007. doi:[10.1561/1900000002](https://doi.org/10.1561/1900000002)
19. Sandeep Parikh and Kelly Stirman: “[Schema Design for Time Series Data in MongoDB](#),” *blog.mongodb.org*, October 30, 2013.
20. Martin Fowler: “[Schemaless Data Structures](#),” *martinfowler.com*, January 7, 2013.
21. Amr Awadallah: “[Schema-on-Read vs. Schema-on-Write](#),” at *Berkeley EECS RAD Lab Retreat*, Santa Cruz, CA, May 2009.
22. Martin Odersky: “[The Trouble with Types](#),” at *Strange Loop*, September 2013.
23. Conrad Irwin: “[MongoDB—Confessions of a PostgreSQL Lover](#),” at *HTML5DevConf*, October 2013.
24. “[Percona Toolkit Documentation: pt-online-schema-change](#),” Percona Ireland Ltd., 2013.
25. Rany Keddo, Tobias Bielohlawek, and Tobias Schmidt: “[Large Hadron Migrator](#),” SoundCloud, 2013.
26. Shlomi Noach: “[gh-ost: GitHub's Online Schema Migration Tool for MySQL](#),” *githubengineering.com*, August 1, 2016.
27. James C. Corbett, Jeffrey Dean, Michael Epstein, et al.: “[Spanner: Google's Globally-Distributed Database](#),” at *10th USENIX Symposium on Operating System Design and Implementation* (OSDI), October 2012.
28. Donald K. Burleson: “[Reduce I/O with Oracle Cluster Tables](#),” *dba-oracle.com*.
29. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al.: “[Bigtable: A Distributed Storage System for Structured Data](#),” at *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.

30. Bobbie J. Cochrane and Kathy A. McKnight: “[DB2 JSON Capabilities, Part 1: Introduction to DB2 JSON](#),” IBM developerWorks, June 20, 2013.
31. Herb Sutter: “[The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software](#),” *Dr. Dobb's Journal*, volume 30, number 3, pages 202–210, March 2005.
32. Joseph M. Hellerstein: “[The Declarative Imperative: Experiences and Conjectures in Distributed Logic](#),” Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech report UCB/EECS-2010-90, June 2010.
33. Jeffrey Dean and Sanjay Ghemawat: “[MapReduce: Simplified Data Processing on Large Clusters](#),” at *6th USENIX Symposium on Operating System Design and Implementation* (OSDI), December 2004.
34. Craig Kerstiens: “[JavaScript in Your Postgres](#),” *blog.heroku.com*, June 5, 2013.
35. Nathan Bronson, Zach Amsden, George Cabrera, et al.: “[TAO: Facebook's Distributed Data Store for the Social Graph](#),” at *USENIX Annual Technical Conference* (USENIX ATC), June 2013.
36. “[Apache TinkerPop3.2.3 Documentation](#),” *tinkerpop.apache.org*, October 2016.
37. “[The Neo4j Manual v2.0.0](#),” Neo Technology, 2013.
38. Emil Eifrem: [Twitter correspondence](#), January 3, 2014.
39. David Beckett and Tim Berners-Lee: “[Turtle – Terse RDF Triple Language](#),” W3C Team Submission, March 28, 2011.
40. “[Datomic Development Resources](#),” Metadata Partners, LLC, 2013.
41. W3C RDF Working Group: “[Resource Description Framework \(RDF\)](#),” *w3.org*, 10 February 2004.
42. “[Apache Jena](#),” Apache Software Foundation.
43. Steve Harris, Andy Seaborne, and Eric Prud'hommeaux: “[SPARQL 1.1 Query Language](#),” W3C Recommendation, March 2013.
44. Todd J. Green, Shan Shan Huang, Boon Thau Loo, and Wenchao Zhou: “[Datalog and Recursive Query Processing](#),” *Foundations and Trends in Databases*, volume 5, number 2, pages 105–195, November 2013. doi:[10.1561/1900000017](#)
45. Stefano Ceri, Georg Gottlob, and Letizia Tanca: “[What You Always Wanted to Know About Datalog \(And Never Dared to Ask\)](#),” *IEEE Transactions on Knowledge and Data Engineering*, volume 1, number 1, pages 146–166, March 1989. doi:[10.1109/69.43410](#)
46. Serge Abiteboul, Richard Hull, and Victor Vianu: *Foundations of Databases*. Addison-Wesley, 1995. ISBN: 978-0-201-53771-0, available online at [webdam.inria.fr/Alice](#)
47. Nathan Marz: “[Cascalog](#),” *cascalog.org*.
48. Dennis A. Benson, Ilene Karsch-Mizrachi, David J. Lipman, et al.: “[GenBank](#),” *Nucleic Acids Research*, volume 36, Database issue, pages D25–D30, December 2007. doi:[10.1093/nar/gkm929](#)
49. Fons Rademakers: “[ROOT for Big Data Analysis](#),” at *Workshop on the Future of Big Data Management*, London, UK, June 2013.

上一章	目錄	下一章
第一章：可靠性、可伸縮性和可維護性	設計資料密集型應用	第三章：儲存與檢索

第三章：儲存與檢索



建立秩序，省卻搜尋

—— 德國諺語

[TOC]

一個數據庫在最基礎的層次上需要完成兩件事情：當你把資料交給資料庫時，它應當把資料儲存起來；而後當你向資料庫要資料時，它應當把資料返回給你。

在 [第二章](#) 中，我們討論了資料模型和查詢語言，即程式設計師將資料錄入資料庫的格式，以及再次要回資料的機制。在本章中我們會從資料庫的視角來討論同樣的問題：資料庫如何儲存我們提供的資料，以及如何在我們需要時重新找到資料。

作為程式設計師，為什麼要關心資料庫內部儲存與檢索的機理？你可能不會去從頭開始實現自己的儲存引擎，但是你確實需要從許多可用的儲存引擎中選擇一個合適的。而且為了讓儲存引擎能在你的工作負載型別上執行良好，你也需要大致瞭解儲存引擎在底層究竟做了什麼。

特別需要注意，針對 **事務性** 負載最佳化的和針對 **分析性** 負載最佳化的儲存引擎之間存在巨大差異。稍後我們將在“[事務處理還是分析？](#)”一節中探討這一區別，並在“[列式儲存](#)”中討論一系列針對分析性負載而最佳化的儲存引擎。

但首先，我們將從你可能已經很熟悉的兩大類資料庫（傳統的關係型資料庫和很多所謂的“NoSQL”資料庫）中使用的儲存引擎來開始本章的內容。我們將研究兩大類儲存引擎：日誌結構（**log-structured**）的儲存引擎，以及面向頁面（**page-oriented**）的儲存引擎（例如 B 樹）。

驅動資料庫的資料結構

世界上最簡單的資料庫可以用兩個 Bash 函式實現：

```
#!/bin/bash
db_set () {
    echo "$1,$2" >> database
}

db_get () {
    grep "^$1," database | sed -e "s/^$1,://" | tail -n 1
}
```

這兩個函式實現了鍵值儲存的功能。執行 `db_set key value` 會將 **鍵 (key)** 和 **值 (value)** 儲存在資料庫中。鍵和值（幾乎）可以是你喜歡的任何東西，例如，值可以是 JSON 文件。然後呼叫 `db_get key` 會查詢與該鍵關聯的最新值並將其返回。

麻雀雖小，五臟俱全：

```
$ db_set 123456 '{"name":"London","attractions":["Big Ben","London Eye"]}'  
$ db_set 42 '{"name":"San Francisco","attractions":["Golden Gate Bridge"]}'  
$ db_get 42  
{ "name": "San Francisco", "attractions": [ "Golden Gate Bridge" ] }
```

底層的儲存格式非常簡單：一個文字檔案，每行包含一條逗號分隔的鍵值對（忽略轉義問題的話，大致與 CSV 檔案類似）。每次對 `db_set` 的呼叫都會向檔案末尾追加記錄，所以更新鍵的時候舊版本的值不會被覆蓋——因而查詢最新值的時候，需要找到檔案中鍵最後一次出現的位置（因此 `db_get` 中使用了 `tail -n 1`）。

```
$ db_set 42 '{"name":"San Francisco","attractions":["Exploratorium"]}'  
$ db_get 42  
{ "name": "San Francisco", "attractions": [ "Exploratorium" ] }  
  
$ cat database  
123456,{"name":"London","attractions":["Big Ben","London Eye"]}  
42,{"name":"San Francisco","attractions":["Golden Gate Bridge"]}  
42,{ "name": "San Francisco", "attractions": [ "Exploratorium" ] }
```

`db_set` 函式對於極其簡單的場景其實有非常好的效能，因為在檔案尾部追加寫入通常是非常高效的。與 `db_set` 做的事情類似，許多資料庫在內部使用了 **日誌 (log)**，也就是一個 **僅追加 (append-only)** 的資料檔案。真正的資料庫有更多的問題需要處理（如併發控制，回收硬碟空間以避免日誌無限增長，處理錯誤與部分寫入的記錄），但基本原理是一樣的。日誌極其有用，我們還將在本書的其它部分重複見到它好幾次。

日誌 (log) 這個詞通常指應用日誌：即應用程式輸出的描述正在發生的事情的文字。本書在更普遍的意義下使用 **日誌** 這一詞：一個僅追加的記錄序列。它可能壓根就不是給人類看的，它可以使用二進位制格式，並僅能由其他程式讀取。

另一方面，如果這個資料庫中有著大量記錄，則這個 `db_get` 函式的效能會非常糟糕。每次你想查詢一個鍵時，`db_get` 必須從頭到尾掃描整個資料庫檔案來查詢鍵的出現。用演算法的語言來說，查詢的開銷是 $O(n)$ ：如果資料庫記錄數量 n 翻了一倍，查詢時間也要翻一倍。這就不好了。

為了高效查詢資料庫中特定鍵的值，我們需要一個數據結構：**索引 (index)**。本章將介紹一系列的索引結構，並在它們之間進行比較。索引背後的大致思想是透過儲存一些額外的元資料作為路標來幫助你找到想要的資料。如果你想以幾種不同的方式搜尋同一份資料，那麼你也許需要在資料的不同部分上建立多個索引。

索引是從主資料衍生的額外的（**additional**）結構。許多資料庫允許新增與刪除索引，這不會影響資料的內容，而只會影響查詢的效能。維護額外的結構會產生開銷，特別是在寫入時。寫入效能很難超過簡單地追加寫入檔案，因為追加寫入是最簡單的寫入操作。任何型別的索引通常都會減慢寫入速度，因為每次寫入資料時都需要更新索引。

這是儲存系統中一個重要的權衡：精心選擇的索引加快了讀查詢的速度，但是每個索引都會拖慢寫入速度。因為這個原因，資料庫預設並不會索引所有的內容，而需要你，也就是程式設計師或資料庫管理員（DBA），基於對應用的典型查詢模式的瞭解來手動選擇索引。你可以選擇那些能為應用帶來最大收益而且又不會引入超出必要開銷的索引。

雜湊索引

讓我們從 **鍵值資料**（key-value Data）的索引開始。這不是你可以索引的唯一資料型別，但鍵值資料是很常見的。在引入更複雜的索引之前，它是重要的第一步。

鍵值儲存與在大多數程式語言中可以找到的 **字典**（dictionary）型別非常相似，通常字典都是用 **雜湊對映**（hash map）或 **散列表**（hash table）實現的。雜湊對映在許多演算法教科書中都有描述【1,2】，所以這裡我們不會討論它的工作細節。既然我們已經可以用雜湊對映來表示 **記憶體** 中的資料結構，為什麼不使用它來索引 **硬碟上** 的資料呢？

假設我們的資料儲存只是一個追加寫入的檔案，就像前面的例子一樣，那麼最簡單的索引策略就是：保留一個記憶體中的雜湊對映，其中每個鍵都對映到資料檔案中的一個位元組偏移量，指明瞭可以找到對應值的位置，如 圖 3-1 所示。當你將新的鍵值對追加寫入檔案中時，還要更新雜湊對映，以反映剛剛寫入的資料的偏移量（這同時適用於插入新鍵與更新現有鍵）。當你想查詢一個值時，使用雜湊對映來查詢資料檔案中的偏移量，尋找（seek）該位置並讀取該值即可。

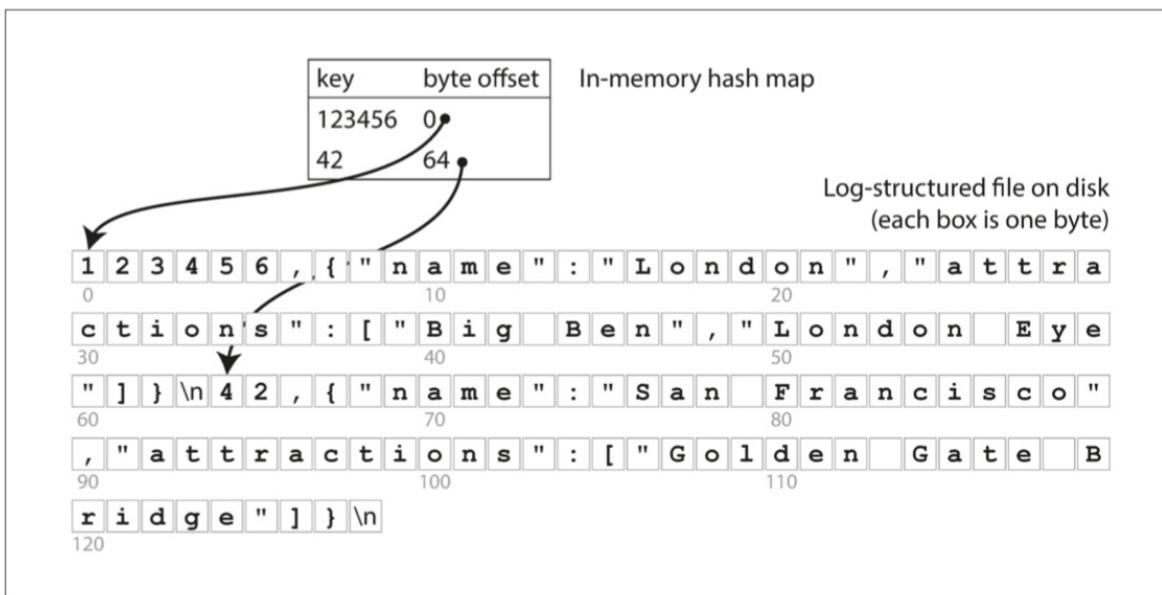


圖 3-1 以類 CSV 格式儲存鍵值對的日誌，並使用記憶體雜湊對映進行索引。

聽上去簡單，但這是一個可行的方法。現實中，Bitcask 實際上就是這麼做的（Riak 中預設的儲存引擎）【3】。Bitcask 提供高效能的讀取和寫入操作，但要求所有的鍵必須能放入可用記憶體中，因為雜湊對映完全保留在記憶體中。而資料值可以使用比可用記憶體更多的空間，因為可以在硬碟上透過一次硬碟查詢操作來載入所需部分，如果資料檔案的那部分已經在檔案系統快取中，則讀取根本不需要任何硬碟 I/O。

像 Bitcask 這樣的儲存引擎非常適合每個鍵的值經常更新的情況。例如，鍵可能是某個貓咪影片的網址（URL），而值可能是該影片被播放的次數（每次有人點選播放按鈕時遞增）。在這種型別的工作負載中，有很多寫操作，但是沒有太多不同的鍵——每個鍵有很多的寫操作，但是將所有鍵儲存在記憶體中是可行的。

到目前為止，我們只是在追加寫入一個檔案——所以如何避免最終用完硬碟空間？一種好的解決方案是，將日誌分為特定大小的 **段**（segment），當日志增長到特定尺寸時關閉當前段檔案，並開始寫入一個新的段檔案。然後，我們就可以對這些段進行 **壓縮**（compaction），如 圖 3-2 所示。這裡的壓縮意味著在日誌中丟棄重複的鍵，只保留每個鍵的

最近更新。

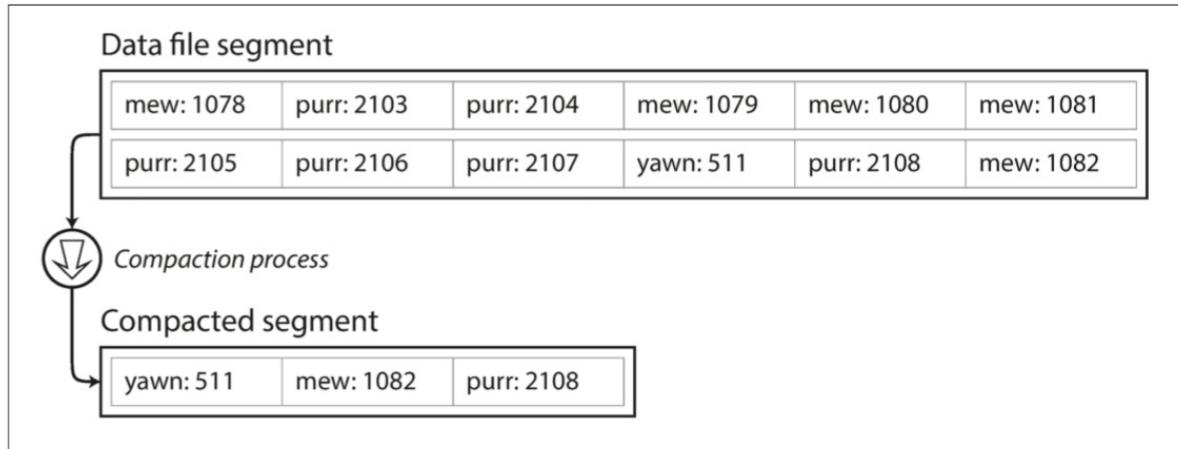


圖 3-2 鍵值更新日誌（統計貓咪影片的播放次數）的壓縮，只保留每個鍵的最近值

而且，由於壓縮經常會使得段變得很小（假設在一個段內鍵被平均重寫了好幾次），我們也可以在執行壓縮的同時將多個段合併在一起，如 圖 3-3 所示。段被寫入後永遠不會被修改，所以合併的段被寫入一個新的檔案。凍結段的合併和壓縮可以在後臺執行緒中完成，這個過程進行的同時，我們仍然可以繼續使用舊的段檔案來正常提供讀寫請求。合併過程完成後，我們將讀取請求轉換為使用新合併的段而不是舊的段——然後舊的段檔案就可以簡單地刪除掉了。

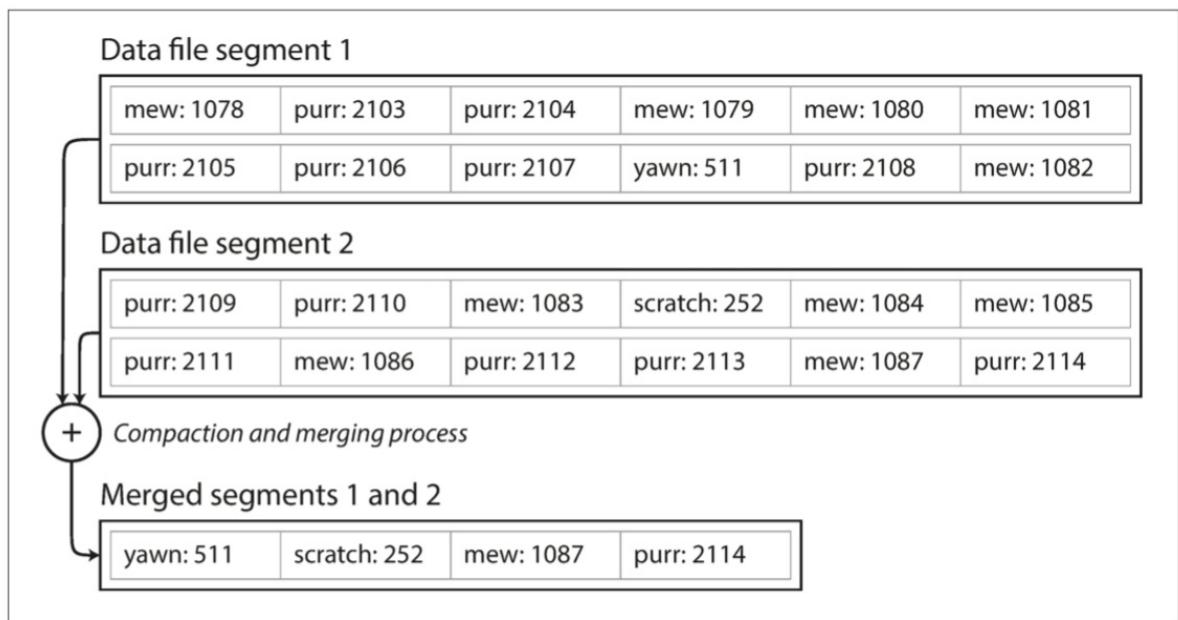


圖 3-3 同時執行壓縮和分段合併

每個段現在都有自己的記憶體散列表，將鍵對映到檔案偏移量。為了找到一個鍵的值，我們首先檢查最近的段的雜湊對映；如果鍵不存在，我們就檢查第二個最近的段，依此類推。合併過程將保持段的數量足夠小，所以查詢過程不需要檢查太多的雜湊對映。

要讓這個簡單的想法在實際中能工作會涉及到大量的細節。簡單來說，下面幾點都是實現過程中需要認真考慮的問題：

- 檔案格式

CSV 不是日誌的最佳格式。使用二進位制格式更快，更簡單：首先以位元組為單位對字串的長度進行編碼，然後是原始的字串（不需要轉義）。

- 刪除記錄

如果要刪除一個鍵及其關聯的值，則必須在資料檔案中追加一個特殊的刪除記錄（邏輯刪除，有時被稱為墓碑，即 tombstone） 。當日志段被合併時，合併過程會透過這個墓碑知道要將被刪除鍵的所有歷史值都丟棄掉。

- 崩潰恢復

如果資料庫重新啟動，則記憶體雜湊對映將丟失。原則上，你可以透過從頭到尾讀取整個段檔案並記錄下來每個鍵的最近值來恢復每個段的雜湊對映。但是，如果段檔案很大，可能需要很長時間，這會使服務的重啟比較痛苦。

Bitcask 透過將每個段的雜湊對映的快照儲存在硬碟上來加速恢復，可以使雜湊對映更快地載入到記憶體中。

- 部分寫入記錄

資料庫隨時可能崩潰，包括在將記錄追加到日誌的過程中。Bitcask 檔案包含校驗和，允許檢測和忽略日誌中的這些損壞部分。

- 併發控制

由於寫操作是以嚴格的順序追加到日誌中的，所以常見的實現是隻有一個寫入執行緒。也因為資料檔案段是僅追加的或者說是不可變的，所以它們可以被多個執行緒同時讀取。

乍一看，僅追加日誌似乎很浪費：為什麼不直接在檔案裡更新，用新值覆蓋舊值？僅追加的設計之所以是個好的設計，有如下幾個原因：

- 追加和分段合併都是順序寫入操作，通常比隨機寫入快得多，尤其是在磁性機械硬碟上。在某種程度上，順序寫入在基於快閃記憶體的 **固態硬碟 (SSD)** 上也是好的選擇【4】。我們將在“[比較 B 樹和 LSM 樹](#)”中進一步討論這個問題。
- 如果段檔案是僅追加的或不可變的，併發和崩潰恢復就簡單多了。例如，當一個數據值被更新的時候發生崩潰，你不用擔心檔案裡將會同時包含舊值和新值各自的一部分。
- 合併舊段的處理也可以避免資料檔案隨著時間的推移而碎片化的問題。

但是，散列表索引也有其侷限性：

- 散列表必須能放進記憶體。如果你有非常多的鍵，那真是倒楣。原則上可以在硬碟上維護一個雜湊對映，不幸的是硬碟雜湊對映很難表現優秀。它需要大量的隨機訪問 I/O，而後者耗盡時想要再擴充是很昂貴的，並且需要很煩瑣的邏輯去解決雜湊衝突【5】。
- 範圍查詢效率不高。例如，你無法輕鬆掃描 kitty00000 和 kitty99999 之間的所有鍵——你必須在雜湊對映中單獨查詢每個鍵。

在下一節中，我們將看到一個沒有這些限制的索引結構。

SSTables和LSM樹

在 [圖 3-3](#) 中，每個日誌結構儲存段都是一系列鍵值對。這些鍵值對按照它們寫入的順序排列，日誌中稍後的值優先於日誌中較早的相同鍵的值。除此之外，檔案中鍵值對的順序並不重要。

現在我們可以對段檔案的格式做一個簡單的改變：要求鍵值對的序列按鍵排序。乍一看，這個要求似乎打破了我們使用順序寫入的能力，我們將稍後再回到這個問題。

我們把這個格式稱為 **排序字串表 (Sorted String Table)**，簡稱 SSTable。我們還要求每個鍵只在每個合併的段檔案中出現一次（壓縮過程已經保證）。與使用雜湊索引的日誌段相比，SSTable 有幾個大的優勢：

- 即使檔案大於可用記憶體，合併段的操作仍然是簡單而高效的。這種方法就像歸併排序演算法中使用的方法一樣，如 [圖 3-4](#) 所示：你開始並排讀取多個輸入檔案，檢視每個檔案中的第一個鍵，複製最低的鍵（根據排序順序）到輸出檔案，不斷重複此步驟，將產生一個新的合併段檔案，而且它也是按鍵排序的。

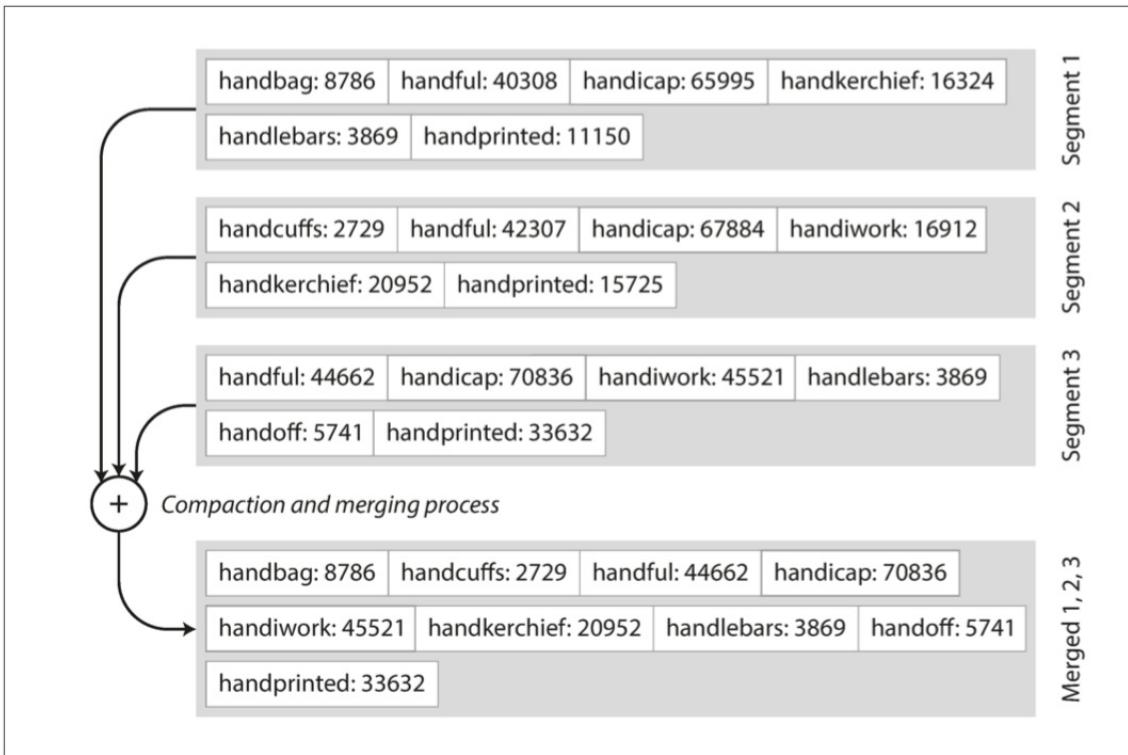


圖 3-4 合併幾個 SSTable 段，只保留每個鍵的最新值

如果在幾個輸入段中出現相同的鍵，該怎麼辦？請記住，每個段都包含在一段時間內寫入資料庫的所有值。這意味著一個輸入段中的所有值一定比另一個段中的所有值都更近（假設我們總是合併相鄰的段）。當多個段包含相同的鍵時，我們可以保留最近段的值，並丟棄舊段中的值。

- 為了在檔案中找到一個特定的鍵，你不再需要在記憶體中儲存所有鍵的索引。以 圖 3-5 為例：假設你正在記憶體中尋找鍵 `handiwork`，但是你不知道這個鍵在段檔案中的確切偏移量。然而，你知道 `handbag` 和 `handsome` 的偏移，而且由於排序特性，你知道 `handiwork` 必須出現在這兩者之間。這意味著你可以跳到 `handbag` 的偏移位置並從那裡掃描，直到你找到 `handiwork`（或沒找到，如果該檔案中沒有該鍵）。

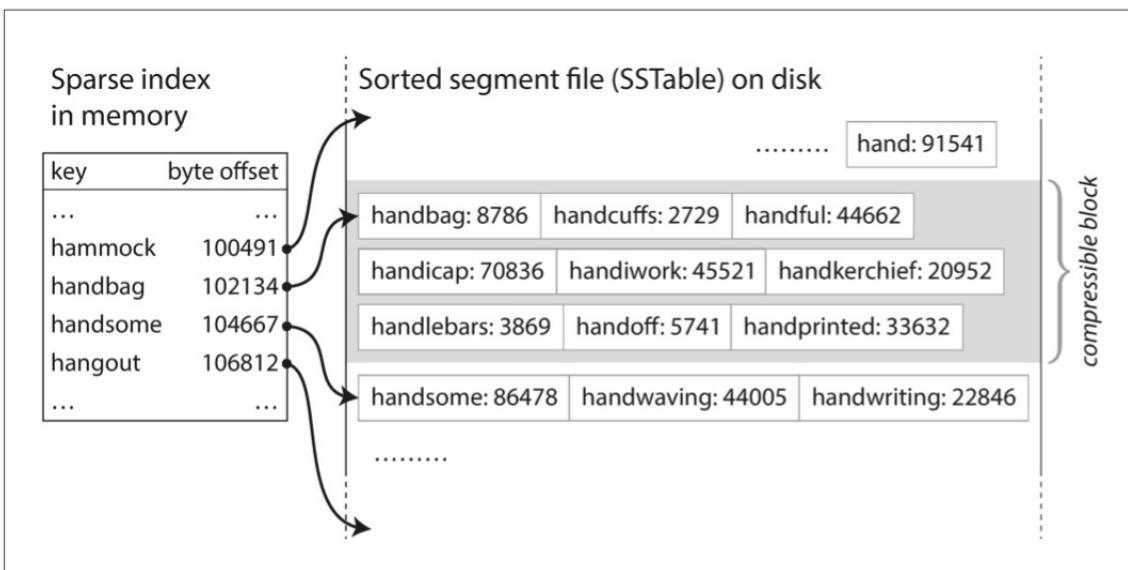


圖 3-5 具有記憶體索引的 SSTable

你仍然需要一個記憶體中的索引來告訴你一些鍵的偏移量，但它可以是稀疏的：每幾千位元組的段檔案有一個鍵就足夠了，因為幾千位元組可以很快地被掃描完ⁱ。

ⁱ. 如果所有的鍵與值都是定長的，你可以使用段檔案上的二分查詢並完全避免使用記憶體索引。然而實踐中的鍵和值通常都是變長的，因此如果沒有索引，就很難知道記錄的分界點（前一條記錄結束以及後一條記錄開始的地方）。 ↪

1. 由於讀取請求無論如何都需要掃描所請求範圍內的多個鍵值對，因此可以將這些記錄分組為塊（block），並在將其寫入硬碟之前對其進行壓縮（如 圖 3-5 中的陰影區域所示）^{譯註 i}。稀疏記憶體索引中的每個條目都指向壓縮塊的開始處。除了節省硬碟空間之外，壓縮還可以減少對 I/O 頻寬的使用。

^{譯註 i}. 這裡的壓縮是 compression，不是前文的 compaction，請注意區分。 ↪

構建和維護SSTables

到目前為止還不錯，但是如何讓你的資料能夠預先排好序呢？畢竟我們接收到的寫入請求可能以任何順序發生。

雖然在硬碟上維護有序結構也是可能的（請參閱“B 樹”），但在記憶體儲存則要容易得多。有許多可以使用的眾所周知的樹形資料結構，例如紅黑樹或 AVL 樹【2】。使用這些資料結構，你可以按任何順序插入鍵，並按排序順序讀取它們。

現在我們可以讓我們的儲存引擎以如下方式工作：

- 有新寫入時，將其新增到記憶體中的平衡樹資料結構（例如紅黑樹）。這個記憶體樹有時被稱為 記憶體表（memtable）。
- 當 記憶體表 大於某個閾值（通常為幾兆位元組）時，將其作為 SSTable 檔案寫入硬碟。這可以高效地完成，因為樹已經維護了按鍵排序的鍵值對。新的 SSTable 檔案將成為資料庫中最新的段。當該 SSTable 被寫入硬碟時，新的寫入可以在一個新的記憶體表例項上繼續進行。
- 收到讀取請求時，首先嘗試在記憶體表中找到對應的鍵，如果沒有就在最近的硬碟段中尋找，如果還沒有就在下一個較舊的段中繼續尋找，以此類推。
- 時不時地，在後臺執行一個合併和壓縮過程，以合併段檔案並將已覆蓋或已刪除的值丟棄掉。

這個方案效果很好。它只會遇到一個問題：如果資料庫崩潰，則最近的寫入（在記憶體表中，但尚未寫入硬碟）將丟失。為了避免這個問題，我們可以在硬碟上儲存一個單獨的日誌，每個寫入都會立即被追加到這個日誌上，就像在前面的章節中所描述的那樣。這個日誌沒有按排序順序，但這並不重要，因為它的唯一目的是在崩潰後恢復記憶體表。每當記憶體表寫出到 SSTable 時，相應的日誌都可以被丟棄。

用SSTables製作LSM樹

這裡描述的演算法本質上是 LevelDB 【6】 和 RocksDB 【7】 這些鍵值儲存引擎庫所使用的技術，這些儲存引擎被設計嵌入到其他應用程式中。除此之外，LevelDB 可以在 Riak 中用作 Bitcask 的替代品。在 Cassandra 和 HBase 中也使用了類似的儲存引擎【8】，而且他們都受到了 Google 的 Bigtable 論文【9】（引入了術語 SSTable 和 memtable）的啟發。

這種索引結構最早由 Patrick O'Neil 等人發明，且被命名為日誌結構合併樹（或 LSM 樹）【10】，它是基於更早之前的日誌結構檔案系統【11】來構建的。基於這種合併和壓縮排序檔案原理的儲存引擎通常被稱為 LSM 儲存引擎。

Lucene，是一種全文搜尋的索引引擎，在 Elasticsearch 和 Solr 被使用，它使用類似的方法來儲存它的關鍵詞詞典【12,13】。全文索引比鍵值索引複雜得多，但是基於類似的想法：在搜尋查詢中，由一個給定的單詞，找到提及單詞的所有文件（網頁、產品描述等）。這也是透過鍵值結構實現的：其中鍵是 單詞（term），值是所有包含該單詞的文件的 ID 列表（postings list）。在 Lucene 中，從詞語到記錄列表的這種對映儲存在類似於 SSTable 的有序檔案中，並根據需要在後臺執行合併【14】。

效能最佳化

與往常一樣，要讓儲存引擎在實踐中表現良好涉及到大量設計細節。例如，當查詢資料庫中不存在的鍵時，LSM 樹演算法可能會很慢：你必須先檢查記憶體表，然後檢視從最近的到最舊的所有段（可能還必須從硬碟讀取每一個段檔案），然後才能確定這個鍵不存在。為了最佳化這種訪問，儲存引擎通常使用額外的布隆過濾器（Bloom filters）

【15】。（布隆過濾器是一種節省記憶體的資料結構，用於近似表達集合的內容，它可以告訴你資料庫中是否存在某個鍵，從而為不存在的鍵節省掉許多不必要的硬碟讀取操作。）

還有一些不同的策略來確定 SSTables 被壓縮和合並的順序和時間。最常見的選擇是 size-tiered 和 leveled compaction。LevelDB 和 RocksDB 使用 leveled compaction（LevelDB 因此得名），HBase 使用 size-tiered，Cassandra 同時支援這兩種【16】。對於 sized-tiered，較新和較小的 SSTables 相繼被合併到較舊的和較大的 SSTable 中。對於 leveled compaction，key（按照分佈範圍）被拆分到較小的 SSTables，而較舊的資料被移動到單獨的層級（level），這使得壓縮（compaction）能夠更加增量地進行，並且使用較少的硬碟空間。

即使有許多微妙的東西，LSM 樹的基本思想——儲存一系列在後臺合併的 SSTables——簡單而有效。即使資料集比可用記憶體大得多，它仍能繼續正常工作。由於資料按排序順序儲存，你可以高效地執行範圍查詢（掃描所有從某個最小值到某個最大值之間的所有鍵），並且因為硬碟寫入是連續的，所以 LSM 樹可以支援非常高的寫入吞吐量。

B樹

前面討論的日誌結構索引看起來已經相當可用了，但它們卻不是最常見的索引型別。使用最廣泛的索引結構和日誌結構索引相當不同，它就是我們接下來要討論的 B 樹。

從 1970 年被引入【17】，僅不到 10 年後就變得“無處不在”【18】，B 樹很好地經受了時間的考驗。在幾乎所有的關係資料庫中，它們仍然是標準的索引實現，許多非關係資料庫也會使用到 B 樹。

像 SSTables 一樣，B 樹保持按鍵排序的鍵值對，這允許高效的鍵值查詢和範圍查詢。但這也就是僅有的相似之處了：B 樹有著非常不同的設計理念。

我們前面看到的日誌結構索引將資料庫分解為可變大小的段，通常是幾兆位元組或更大的大小，並且總是按順序寫入段。相比之下，B 樹將資料庫分解成固定大小的塊（block）或分頁（page），傳統上大小為 4KB（有時會更大），並且一次只能讀取或寫入一個頁面。這種設計更接近於底層硬體，因為硬碟空間也是按固定大小的塊來組織的。

每個頁面都可以使用地址或位置來標識，這允許一個頁面引用另一個頁面——類似於指標，但在硬碟而不是在記憶體中。我們可以使用這些頁面引用來構建一個頁面樹，如 圖 3-6 所示。

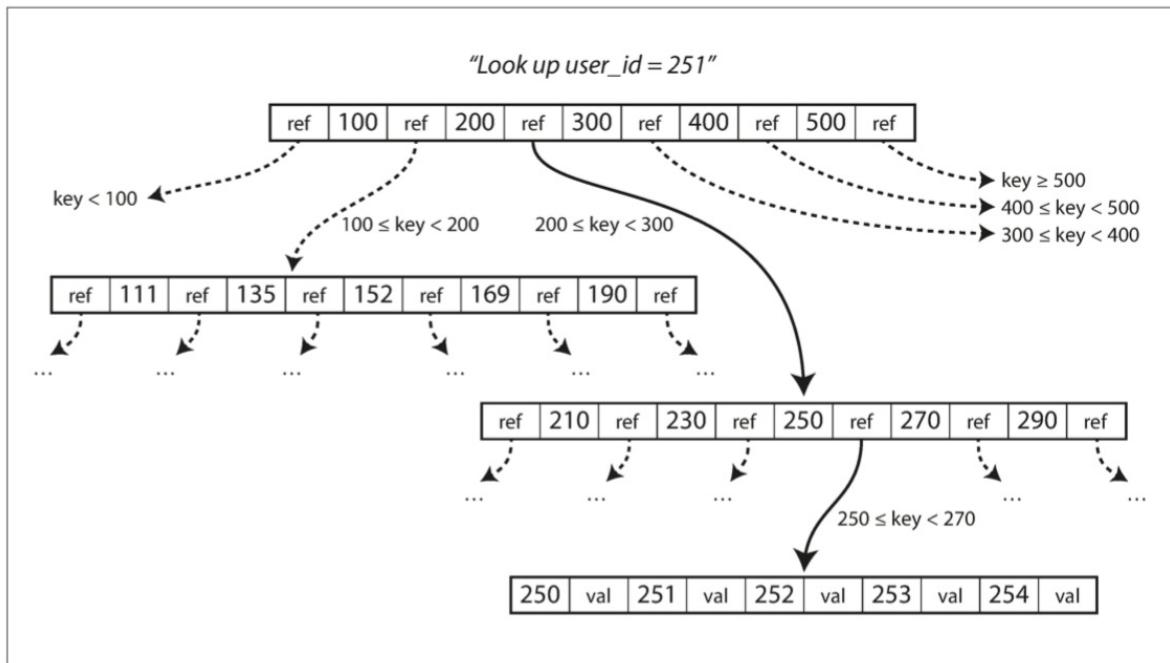


圖 3-6 使用 B 樹索引查詢一個鍵

一個頁面會被指定為 B 樹的根；在索引中查詢一個鍵時，就從這裡開始。該頁面包含幾個鍵和對子頁面的引用。每個子頁面負責一段連續範圍的鍵，根頁面上每兩個引用之間的鍵，表示相鄰子頁面管理的鍵的範圍（邊界）。

在 圖 3-6 的例子中，我們正在尋找鍵 251，所以我們知道我們需要跟蹤邊界 200 和 300 之間的頁面引用。這將我們帶到一個類似的頁面，進一步將 200 到 300 的範圍拆分到子範圍。

最終，我們將到達某個包含單個鍵的頁面（葉子頁面，leaf page），該頁面或者直接包含每個鍵的值，或者包含了對可以找到值的頁面的引用。

在 B 樹的一個頁面中對子頁面的引用的數量稱為 **分支因子 (branching factor)**。例如，在 圖 3-6 中，分支因子是 6。在實踐中，分支因子的大小取決於儲存頁面引用和範圍邊界所需的空間，但這個值通常是幾百。

如果要更新 B 樹中現有鍵的值，需要搜尋包含該鍵的葉子頁面，更改該頁面中的值，並將該頁面寫回到硬碟（對該頁面的任何引用都將保持有效）。如果你想新增一個新的鍵，你需要找到其範圍能包含新鍵的頁面，並將其新增到該頁面。如果頁面中沒有足夠的可用空間容納新鍵，則將其分成兩個半滿頁面，並更新父頁面以反映新的鍵範圍分割槽，如 圖 3-7 所示ⁱⁱ。

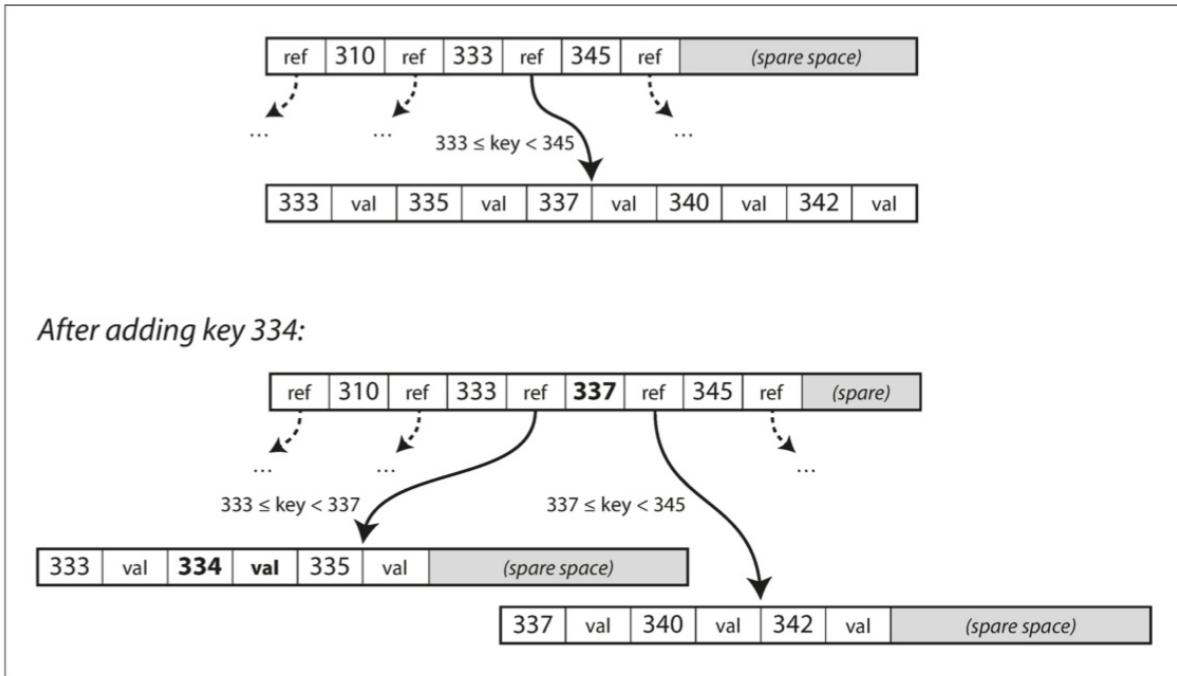


圖 3-7 透過分割頁面來生長 B 樹

ⁱⁱ. 向 B 樹中插入一個新的鍵是相當符合直覺的，但刪除一個鍵（同時保持樹平衡）就會牽扯很多其他東西了
【2】。 ↪

這個演算法可以確保樹保持平衡：具有 n 個鍵的 B 樹總是具有 $\$O(\log n)\$$ 的深度。大多數資料庫可以放入一個三到四層的 B 樹，所以你不需要追蹤多個頁面引用來找到你正在查詢的頁面（分支因子為 500 的 4KB 頁面的四層樹可以儲存多達 256TB 的資料）。

讓B樹更可靠

B 樹的基本底層寫操作是用新資料覆寫硬碟上的頁面，並假定覆寫不改變頁面的位置：即，當頁面被覆寫時，對該頁面的所有引用保持完整。這與日誌結構索引（如 LSM 樹）形成鮮明對比，後者只追加到檔案（並最終刪除過時的檔案），但從不修改檔案中已有的內容。

你可以把覆寫硬碟上的頁面對應為實際的硬體操作。在磁性硬碟驅動器上，這意味著將磁頭移動到正確的位置，等待旋轉盤上的正確位置出現，然後用新的資料覆寫適當的扇區。在固態硬碟上，由於 SSD 必須一次擦除和重寫相當大的儲存晶片塊，所以會發生更複雜的事情【19】。

而且，一些操作需要覆寫幾個不同的頁面。例如，如果因為插入導致頁面過滿而拆分頁面，則需要寫入新拆分的兩個頁面，並覆寫其父頁面以更新對兩個子頁面的引用。這是一個危險的操作，因為如果資料庫在系列操作進行到一半時崩潰，那麼最終將導致一個損壞的索引（例如，可能有一個孤兒頁面沒有被任何頁面引用）。

為了使資料庫能處理異常崩潰的場景，B 樹實現通常會帶有一個額外的硬碟資料結構：預寫式日誌（WAL，即 write-ahead log，也稱為 重做日誌，即 redo log）。這是一個僅追加的檔案，每個 B 樹的修改在其能被應用到樹本身的頁面之前都必須先寫入到該檔案。當資料庫在崩潰後恢復時，這個日誌將被用來使 B 樹恢復到一致的狀態【5,20】。

另外還有一個更新頁面的複雜情況是，如果多個執行緒要同時訪問 B 樹，則需要仔細的併發控制——否則執行緒可能會看到樹處於不一致的狀態。這通常是透過使用 鎖存器（latches，輕量級鎖）保護樹的資料結構來完成。日誌結構化的方法在這方面更簡單，因為它們在後臺進行所有的合併，而不會干擾新接收到的查詢，並且能夠時不時地將段檔案切換為新的（該切換是原子操作）。

B樹的最佳化

由於 B 樹已經存在了很久，所以並不奇怪這麼多年下來有很多最佳化的設計被開發出來，僅舉幾例：

- 不同於覆寫頁面並維護 WAL 以支援崩潰恢復，一些資料庫（如 LMDB）使用寫時複製方案【21】。經過修改的頁面被寫入到不同的位置，並且還在樹中建立了父頁面的新版本，以指向新的位置。這種方法對於併發控制也很有用，我們將在“[快照隔離和可重複讀](#)”中看到。
- 我們可以透過不儲存整個鍵，而是縮短其大小，來節省頁面空間。特別是在樹內部的頁面上，鍵只需要提供足夠的資訊來充當鍵範圍之間的邊界。在頁面中包含更多的鍵允許樹具有更高的分支因子，因此也就允許更少的層級ⁱⁱⁱ。
- 通常，頁面可以放置在硬碟上的任何位置；沒有什麼要求相鄰鍵範圍的頁面也放在硬碟上相鄰的區域。如果某個查詢需要按照排序順序掃描大部分的鍵範圍，那麼這種按頁面儲存的佈局可能會效率低下，因為每個頁面的讀取都需要執行一次硬碟查詢。因此，許多 B 樹的實現在佈局樹時會盡量使葉子頁面按順序出現在硬碟上。但是，隨著樹的增長，要維持這個順序是很困難的。相比之下，由於 LSM 樹在合併過程中一次性重寫一大段儲存，所以它們更容易使順序鍵在硬碟上連續儲存。
- 額外的指標被新增到樹中。例如，每個葉子頁面可以引用其左邊和右邊的兄弟頁面，使得不用跳回父頁面就能按順序對鍵進行掃描。
- B 樹的變體如 [分形樹（fractal trees）](#)【22】借用了一些日誌結構的思想來減少硬碟查詢（而且它們與分形無關）。

ⁱⁱⁱ. 這個變種有時被稱為 B+ 樹，但因為這個最佳化已被廣泛使用，所以經常無法區分於其它的 B 樹變種。 ↪

比較B樹和LSM樹

儘管 B 樹實現通常比 LSM 樹實現更成熟，但 LSM 樹由於效能特徵也非常有趣。根據經驗，通常 LSM 樹的寫入速度更快，而 B 樹的讀取速度更快【23】。LSM 樹上的讀取通常比較慢，因為它們必須檢查幾種不同的資料結構和不同壓縮（Compaction）層級的 SSTables。

然而，基準測試的結果通常和工作負載的細節相關。你需要用你特有的工作負載來測試系統，以便進行有效的比較。在本節中，我們將簡要討論一些在衡量儲存引擎效能時值得考慮的事情。

LSM樹的優點

B 樹索引中的每塊資料都必須至少寫入兩次：一次寫入預先寫入日誌（WAL），一次寫入樹頁面本身（如果有分頁還需要再寫入一次）。即使在該頁面中只有幾個位元組發生了變化，也需要接受寫入整個頁面的開銷。有些儲存引擎甚至會覆寫同一個頁面兩次，以免在電源故障的情況下頁面未完整更新【24,25】。

由於反覆壓縮和合並 SSTables，日誌結構索引也會多次重寫資料。這種影響——在資料庫的生命週期中每筆資料導致對硬碟的多次寫入——被稱為 [寫入放大（write amplification）](#)。使用固態硬碟的機器需要額外關注這點，固態硬碟的快閃記憶體壽命在覆寫有限次數後就會耗盡。

在寫入繁重的應用程式中，效能瓶頸可能是資料庫可以寫入硬碟的速度。在這種情況下，寫放大會導致直接的效能代價：儲存引擎寫入硬碟的次數越多，可用硬碟頻寬內它能處理的每秒寫入次數就越少。

進而，LSM 樹通常能夠比 B 樹支援更高的寫入吞吐量，部分原因是它們有時具有較低的寫放大（儘管這取決於儲存引擎的配置和工作負載），部分是因為它們順序地寫入緊湊的 SSTable 檔案而不是必須覆寫樹中的幾個頁面【26】。這種差異在機械硬碟上尤其重要，其順序寫入比隨機寫入要快得多。

LSM 樹可以被壓縮得更好，因此通常能比 B 樹在硬碟上產生更小的檔案。B 樹儲存引擎會由於碎片化（fragmentation）而留下一些未使用的硬碟空間：當頁面被拆分或某行不能放入現有頁面時，頁面中的某些空間仍未被使用。由於 LSM 樹不是面向頁面的，並且會透過定期重寫 SSTables 以去除碎片，所以它們具有較低的儲存開銷，特別是當使用分層壓縮（leveled compaction）時【27】。

在許多固態硬碟上，韌體內部使用了日誌結構化演算法，以將隨機寫入轉變為順序寫入底層儲存晶片，因此儲存引擎寫入模式的影響不太明顯【19】。但是，較低的寫入放大率和減少的碎片仍然對固態硬碟更有利：更緊湊地表示資料允許在可用的 I/O 頻寬內處理更多的讀取和寫入請求。

LSM樹的缺點

日誌結構儲存的缺點是壓縮過程有時會干擾正在進行的讀寫操作。儘管儲存引擎嘗試增量地執行壓縮以儘量不影響併發訪問，但是硬碟資源有限，所以很容易發生某個請求需要等待硬碟先完成昂貴的壓縮操作。對吞吐量和平均響應時間的影響通常很小，但是日誌結構化儲存引擎在更高百分位的響應時間（請參閱“[描述效能](#)”）有時會相當長，而 B 樹的行為則相對更具有可預測性【28】。

壓縮的另一個問題出現在高寫入吞吐量時：硬碟的有限寫入頻寬需要在初始寫入（記錄日誌和重新整理記憶體表到硬碟）和在後臺執行的壓縮執行緒之間共享。寫入空資料庫時，可以使用全硬碟頻寬進行初始寫入，但資料庫越大，壓縮所需的硬碟頻寬就越多。

如果寫入吞吐量很高，並且壓縮沒有仔細配置好，有可能導致壓縮跟不上寫入速率。在這種情況下，硬碟上未合併段的數量不斷增加，直到硬碟空間用完，讀取速度也會減慢，因為它們需要檢查更多的段檔案。通常情況下，即使壓縮無法跟上，基於 SSTable 的儲存引擎也不會限制傳入寫入的速率，所以你需要進行明確的監控來檢測這種情況【29,30】。

B 樹的一個優點是每個鍵只存在於索引中的一個位置，而日誌結構化的儲存引擎可能在不同的段中有相同鍵的多個副本。這個方面使得 B 樹在想要提供強大的事務語義的資料庫中很有吸引力：在許多關係資料庫中，事務隔離是透過在鍵範圍上使用鎖來實現的，在 B 樹索引中，這些鎖可以直接附加到樹上【5】。在 [第七章](#) 中，我們將更詳細地討論這一點。

B 樹在資料庫架構中是非常根深蒂固的，為許多工作負載都提供了始終如一的良好效能，所以它們不可能在短期內消失。在新的資料庫中，日誌結構化索引變得越來越流行。沒有簡單易行的辦法來判斷哪種型別的儲存引擎對你的使用場景更好，所以需要透過一些測試來得到相關經驗。

其他索引結構

到目前為止，我們只討論了鍵值索引，它們就像關係模型中的 **主鍵**（primary key）索引。主鍵唯一標識關係表中的一行，或文件資料庫中的一個文件或圖形資料庫中的一個頂點。資料庫中的其他記錄可以透過其主鍵（或 ID）引用該行 / 文件 / 頂點，索引就被用於解析這樣的引用。

次級索引（secondary indexes）也很常見。在關係資料庫中，你可以使用 `CREATE INDEX` 命令在同一個表上建立多個次級索引，而且這些索引通常對於有效地執行聯接（join）而言至關重要。例如，在 [第二章](#) 中的 [圖 2-1](#) 中，很可能在 `user_id` 列上有一個次級索引，以便你可以在每個表中找到屬於同一使用者的所有行。

次級索引可以很容易地從鍵值索引構建。次級索引主要的不同是鍵不是唯一的，即可能有許多行（文件，頂點）具有相同的鍵。這可以透過兩種方式來解決：將匹配行識別符號的列表作為索引裡的值（就像全文索引中的記錄列表），或者透過向每個鍵新增行識別符號來使鍵唯一。無論哪種方式，B 樹和日誌結構索引都可以用作次級索引。

將值儲存在索引中

索引中的鍵是查詢要搜尋的內容，而其值可以是以下兩種情況之一：它可以是實際的行（文件，頂點），也可以是對儲存在別處的行的引用。在後一種情況下，行被儲存的地方被稱為 **堆檔案 (heap file)**，並且儲存的資料沒有特定的順序（它可以是僅追加的，或者它可以跟蹤被刪除的行以便後續可以用新的資料進行覆蓋）。堆檔案方法很常見，因為它避免了在存在多個次級索引時對資料的複製：每個索引只引用堆檔案中的一個位置，實際的資料都儲存在一個地方。

在不更改鍵的情況下更新值時，堆檔案方法可以非常高效：只要新值的位元組數不大於舊值，就可以覆蓋該記錄。如果新值更大，情況會更複雜，因為它可能需要移到堆中有足夠空間的新位置。在這種情況下，要麼所有的索引都需要更新，以指向記錄的新堆位置，或者在舊堆位置留下一個轉發指標【5】。

在某些情況下，從索引到堆檔案的額外跳躍對讀取來說效能損失太大，因此可能希望將被索引的行直接儲存在索引中。這被稱為聚集索引（clustered index）。例如，在 MySQL 的 InnoDB 儲存引擎中，表的主鍵總是一個聚集索引，次級索引則引用主鍵（而不是堆檔案中的位置）【31】。在 SQL Server 中，可以為每個表指定一個聚集索引【32】。

在 **聚集索引**（在索引中儲存所有的行資料）和 **非聚集索引**（僅在索引中儲存對資料的引用）之間的折衷被稱為 **覆蓋索引 (covering index)** 或 **包含列的索引 (index with included columns)**，其在索引記憶體儲表的一部分列【33】。這允許透過單獨使用索引來處理一些查詢（這種情況下，可以說索引 **覆蓋 (cover)** 了查詢）【32】。

與任何型別的資料重複一樣，聚集索引和覆蓋索引可以加快讀取速度，但是它們需要額外的儲存空間，並且會增加寫入開銷。資料庫還需要額外的努力來執行事務保證，因為應用程式不應看到任何因為使用副本而導致的不一致。

多列索引

至今討論的索引只是將一個鍵對映到一個值。如果我們需要同時查詢一個表中的多個列（或文件中的多個欄位），這顯然是不夠的。

最常見的多列索引被稱為 **連線索引 (concatenated index)**，它透過將一列的值追加到另一列後面，簡單地將多個欄位組合成一個鍵（索引定義中指定了欄位的連線順序）。這就像一個老式的紙質電話簿，它提供了一個從（姓氏，名字）到電話號碼的索引。由於排序順序，索引可以用來查詢所有具有特定姓氏的人，或所有具有特定姓氏 - 名字組合的人。但如果你想找到所有具有特定名字的人，這個索引是沒有用的。

多維索引 (multi-dimensional index) 是一種查詢多個列的更一般的方法，這對於地理空間資料尤為重要。例如，餐廳搜尋網站可能有一個數據庫，其中包含每個餐廳的經度和緯度。當用戶在地圖上檢視餐館時，網站需要搜尋使用者正在檢視的矩形地圖區域內的所有餐館。這需要一個二維範圍查詢，如下所示：

```
SELECT * FROM restaurants WHERE latitude > 51.4946 AND latitude < 51.5079
    AND longitude > -0.1162 AND longitude < -0.1004;
```

一個標準的 B 樹或者 LSM 樹索引不能夠高效地處理這種查詢：它可以返回一個緯度範圍內的所有餐館（但經度可能是任意值），或者返回在同一個經度範圍內的所有餐館（但緯度可能是北極和南極之間的任意地方），但不能同時滿足兩個條件。

一種選擇是使用 **空間填充曲線 (space-filling curve)** 將二維位置轉換為單個數字，然後使用常規 B 樹索引【34】。更普遍的是，使用特殊化的空間索引，例如 R 樹。例如，PostGIS 使用 PostgreSQL 的通用 GiST 工具【35】將地理空間索引實現為 R 樹。這裡我們沒有足夠的地方來描述 R 樹，但是有大量的文獻可供參考。

有趣的是，多維索引不僅可以用於地理位置。例如，在電子商務網站上可以使用建立在（紅，綠，藍）維度上的三維索引來搜尋特定顏色範圍內的產品，也可以在天氣觀測資料庫中建立（日期，溫度）的二維索引，以便有效地搜尋 2013 年內的溫度在 25 至 30°C 之間的所有觀測資料。如果使用一維索引，你將不得不掃描 2013 年的所有記錄（不管溫度如何），然後透過溫度進行過濾，或者反之亦然。二維索引可以同時透過時間戳和溫度來收窄資料集。這個技術被 HyperDex 所使用【36】。

全文搜尋和模糊索引

到目前為止所討論的所有索引都假定你有確切的資料，並允許你查詢鍵的確切值或具有排序順序的鍵的值範圍。他們不允許你做的是搜尋類似的鍵，如拼寫錯誤的單詞。這種模糊的查詢需要不同的技術。

例如，全文搜尋引擎通常允許搜尋目標從一個單詞擴充套件為包括該單詞的同義詞，忽略單詞的語法變體，搜尋在相同文件中的近義詞，並且支援各種其他取決於文字的語言分析功能。為了處理文件或查詢中的拼寫錯誤，Lucene 能夠在一定的編輯距離內搜尋文字【37】（編輯距離 1 意味著單詞內發生了 1 個字母的新增、刪除或替換）。

正如“[用 SSTables 製作 LSM 樹](#)”中所提到的，Lucene 為其詞典使用了一個類似於 SSTable 的結構。這個結構需要一個小的記憶體索引，告訴查詢需要在排序檔案中哪個偏移量查詢鍵。在 LevelDB 中，這個記憶體中的索引是一些鍵的稀疏集合，但在 Lucene 中，記憶體中的索引是鍵中字元的有限狀態自動機，類似於 trie 【38】。這個自動機可以轉換成 Levenshtein 自動機，它支援在給定的編輯距離內有效地搜尋單詞【39】。

其他的模糊搜尋技術正朝著文件分類和機器學習的方向發展。更多詳細資訊請參閱資訊檢索教科書，例如【40】。

在記憶體中儲存一切

本章到目前為止討論的資料結構都是對硬碟限制的應對。與主記憶體相比，硬碟處理起來很麻煩。對於磁性硬碟和固態硬碟，如果要在讀取和寫入時獲得良好效能，則需要仔細地佈置硬碟上的資料。但是，我們能容忍這種麻煩，因為硬碟有兩個顯著的優點：它們是持久的（它們的內容在電源關閉時不會丟失），並且每 GB 的成本比 RAM 低。

隨著 RAM 變得更便宜，每 GB 成本的論據被侵蝕了。許多資料集不是那麼大，所以將它們全部儲存在記憶體中是非常可行的，包括可能分佈在多個機器上。這導致了記憶體資料庫的發展。

某些記憶體中的鍵值儲存（如 Memcached）僅用於快取，在重新啟動計算機時丟失的資料是可以接受的。但其他記憶體資料庫的目標是永續性，可以透過特殊的硬體（例如電池供電的 RAM）來實現，也可以將更改日誌寫入硬碟，還可以將定時快照寫入硬碟或者將記憶體中的狀態複製到其他機器上。

記憶體資料庫重新啟動時，需要從硬碟或透過網路從副本重新載入其狀態（除非使用特殊的硬體）。儘管寫入硬碟，它仍然是一個記憶體資料庫，因為硬碟僅出於永續性目的進行日誌追加，讀取請求完全由記憶體來處理。寫入硬碟同時還有運維上的好處：硬碟上的檔案可以很容易地由外部程式進行備份、檢查和分析。

諸如 VoltDB、MemSQL 和 Oracle TimesTen 等產品是具有關係模型的記憶體資料庫，供應商聲稱，透過消除與管理硬碟上的資料結構相關的所有開銷，他們可以提供巨大的效能改進【41,42】。RAM Cloud 是一個開源的記憶體鍵值儲存器，具有永續性（對記憶體和硬碟上的資料都使用日誌結構化方法）【43】。Redis 和 Couchbase 透過非同步寫入硬碟提供了較弱的永續性。

反直覺的是，記憶體資料庫的效能優勢並不是因為它們不需要從硬碟讀取的事實。只要有足夠的記憶體即使是基於硬碟的儲存引擎也可能永遠不需要從硬碟讀取，因為作業系統在記憶體中快取了最近使用的硬碟塊。相反，它們更快的原因在於省去了將記憶體資料結構編碼為硬碟資料結構的開銷【44】。

除了效能，記憶體資料庫的另一個有趣的地方是提供了難以用基於硬碟的索引實現的資料模型。例如，Redis 為各種資料結構（如優先順序佇列和集合）提供了類似資料庫的介面。因為它將所有資料儲存在記憶體中，所以它的實現相對簡單。

最近的研究表明，記憶體資料庫體系結構可以擴充套件到支援比可用記憶體更大的資料集，而不必重新採用以硬碟為中心的體系結構【45】。所謂的 **反快取 (anti-caching)** 方法透過在記憶體不足的情況下將最近最少使用的資料從記憶體轉移到硬碟，並在將來再次訪問時將其重新載入到記憶體中。這與作業系統對虛擬記憶體和交換檔案的操作類似，但資料庫可以比作業系統更有效地管理記憶體，因為它可以按單個記錄的粒度工作，而不是整個記憶體頁面。儘管如此，這種方法仍然需要索引能完全放入記憶體中（就像本章開頭的 Bitcask 例子）。

如果 **非易失性儲存器 (non-volatile memory, NVM)** 技術得到更廣泛的應用，可能還需要進一步改變儲存引擎設計【46】。目前這是一個新的研究領域，值得關注。

事務處理還是分析？

在早期的業務資料處理過程中，一次典型的資料庫寫入通常與一筆商業交易（commercial transaction）相對應：賣個貨、向供應商下訂單、支付員工工資等等。但隨著資料庫開始應用到那些不涉及到錢的領域，術語 **交易 / 事務 (transaction)** 仍留了下來，用於指代一組讀寫操作構成的邏輯單元。

事務不一定具有 ACID（原子性，一致性，隔離性和永續性）屬性。事務處理只是意味著允許客戶端進行低延遲的讀取和寫入——而不是隻能定期執行（例如每天一次）的批處理作業。我們在 [第七章](#) 中討論 ACID 屬性，在 [第十章](#) 中討論批處理。

即使資料庫開始被用於許多不同型別的資料，比如部落格文章的評論、遊戲中的動作、地址簿中的聯絡人等等，基本的訪問模式仍然類似於處理商業交易。應用程式通常使用索引透過某個鍵找少量記錄。根據使用者的輸入來插入或更新記錄。由於這些應用程式是互動式的，這種訪問模式被稱為 **線上事務處理 (OLTP, OnLine Transaction Processing)**。

但是，資料庫也開始越來越多地用於資料分析，這些資料分析具有非常不同的訪問模式。通常，分析查詢需要掃描大量記錄，每個記錄只讀取幾列，並計算彙總統計資訊（如計數、總和或平均值），而不是將原始資料返回給使用者。例如，如果你的資料是一個銷售交易表，那麼分析查詢可能是：

- 一月份每個商店的總收入是多少？
- 在最近的推廣活動中多賣了多少香蕉？
- 哪個牌子的嬰兒食品最常與 X 品牌的尿布同時購買？

這些查詢通常由業務分析師編寫，並提供報告以幫助公司管理層做出更好的決策（商業智慧）。為了將這種使用資料庫的模式和事務處理區分開，它被稱為 **線上分析處理 (OLAP, OnLine Analytic Processing)** [\[47\]](#)^{iv}。OLTP 和 OLAP 之間的區別並不總是清晰的，但是一些典型的特徵在 [表 3-1](#) 中列出。

表 3-1 比較事務處理和分析系統的特點

屬性	事務處理系統 OLTP	分析系統 OLAP
主要讀取模式	查詢少量記錄，按鍵讀取	在大批次記錄上聚合
主要寫入模式	隨機訪問，寫入要求低延時	批次匯入 (ETL) 或者事件流
主要使用者	終端使用者，透過 Web 應用	內部資料分析師，用於決策支援
處理的資料	資料的最新狀態（當前時間點）	隨時間推移的歷史事件
資料集尺寸	GB ~ TB	TB ~ PB

^{iv}. OLAP 中的首字母 O (online) 的含義並不明確，它可能是指查詢並不是用來生成預定義好的報告的事實，也可能是指分析師通常是互動式地使用 OLAP 系統來進行探索式的查詢。 ↩

起初，事務處理和分析查詢使用了相同的資料庫。SQL 在這方面已證明是非常靈活的：對於 OLTP 型別的查詢以及 OLAP 型別的查詢來說效果都很好。儘管如此，在二十世紀八十年代末和九十年代初期，企業有停止使用 OLTP 系統進行分析的趨勢，轉而在單獨的資料庫上執行分析。這個單獨的資料庫被稱為 **資料倉庫 (data warehouse)**。

資料倉庫

一個企業可能有幾十個不同的交易處理系統：面向終端客戶的網站、控制實體商店的收銀系統、倉庫庫存跟蹤、車輛路線規劃、供應鏈管理、員工管理等。這些系統中每一個都很複雜，需要專人維護，所以最終這些系統互相之間都是獨立執行的。

這些 OLTP 系統往往對業務運作至關重要，因而通常會要求 **高可用 與 低延遲**。所以 DBA 會密切關注他們的 OLTP 資料庫，他們通常不願意讓業務分析人員在 OLTP 資料庫上執行臨時的分析查詢，因為這些查詢通常開銷巨大，會掃描大部分資料集，這會損害同時在執行的事務的效能。

相比之下，資料倉庫是一個獨立的資料庫，分析人員可以查詢他們想要的內容而不影響 OLTP 操作 [\[48\]](#)。資料倉庫包含公司各種 OLTP 系統中所有的只讀資料副本。從 OLTP 資料庫中提取資料（使用定期的資料轉儲或連續的更新流），轉換成適合分析的模式，清理並載入到資料倉庫中。將資料存入倉庫的過程稱為 **“抽取 - 轉換 - 載入 (ETL)”**，如 [圖 3-8](#) 所示。

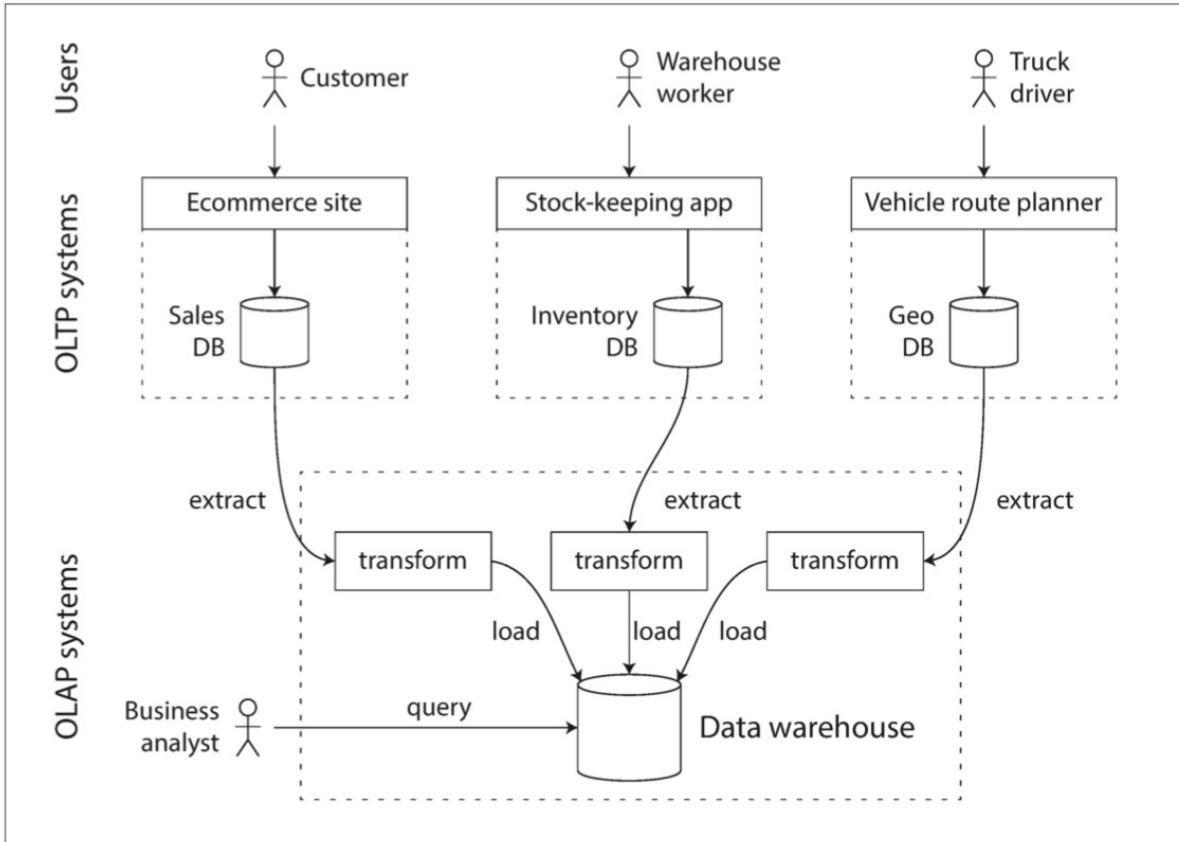


圖 3-8 ETL 至資料倉庫的簡化提綱

幾乎所有的大型企業都有資料倉庫，但在小型企業中幾乎聞所未聞。這可能是因為大多數小公司沒有這麼多不同的 OLTP 系統，大多數小公司只有少量的資料——可以在傳統的 SQL 資料庫中查詢，甚至可以在電子表格中分析。在一家大公司裡，要做一些在一家小公司很簡單的事情，需要很多繁重的工作。

使用單獨的資料倉庫，而不是直接查詢 OLTP 系統進行分析的一大優勢是資料倉庫可針對分析類的訪問模式進行最佳化。事實證明，本章前半部分討論的索引演算法對於 OLTP 來說工作得很好，但對於處理分析查詢並不是很好。在本章的其餘部分中，我們將研究為分析而最佳化的儲存引擎。

OLTP資料庫和資料倉庫之間的分歧

資料倉庫的資料模型通常是關係型的，因為 SQL 通常很適合分析查詢。有許多圖形資料分析工具可以生成 SQL 查詢，視覺化結果，並允許分析人員探索資料（透過下鑽、切片和切塊等操作）。

表面上，一個數據倉庫和一個關係型 OLTP 資料庫看起來很相似，因為它們都有一個 SQL 查詢介面。然而，系統的內部看起來可能完全不同，因為它們針對非常不同的查詢模式進行了最佳化。現在許多資料庫供應商都只是重點支援事務處理負載和分析工作負載這兩者中的一個，而不是都支援。

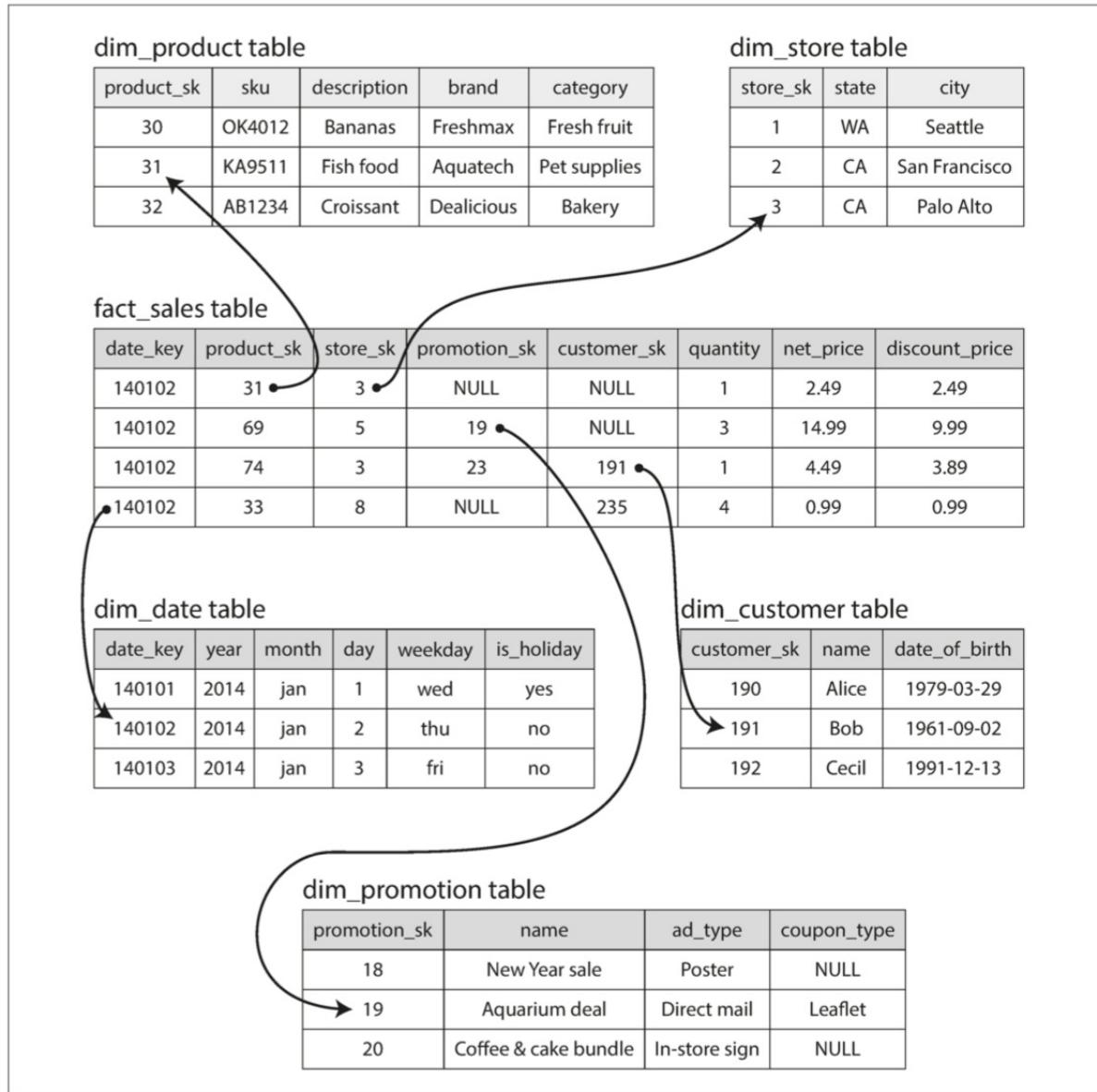
一些資料庫（例如 Microsoft SQL Server 和 SAP HANA）支援在同一產品中進行事務處理和資料倉庫。但是，它們也正日益發展為兩套獨立的儲存和查詢引擎，只是這些引擎正好可以透過一個通用的 SQL 介面訪問【49,50,51】。

Teradata、Vertica、SAP HANA 和 ParAccel 等資料倉庫供應商通常使用昂貴的商業許可證銷售他們的系統。Amazon RedShift 是 ParAccel 的託管版本。最近，大量的開源 SQL-on-Hadoop 專案已經出現，它們還很年輕，但是正在與商業資料倉庫系統競爭，包括 Apache Hive、Spark SQL、Cloudera Impala、Facebook Presto、Apache Tajo 和 Apache Drill【52,53】。其中一些基於了谷歌 Dremel 的想法【54】。

星型和雪花型：分析的模式

正如 [第二章](#) 所探討的，根據應用程式的需要，在事務處理領域中使用了大量不同的資料模型。另一方面，在分析型業務中，資料模型的多樣性則少得多。許多資料倉庫都以相當公式化的方式使用，被稱為星型模式（也稱為維度建模 [\[55\]](#)）。

[圖 3-9](#) 中的示意模式顯示了可能在食品零售商處找到的資料倉庫。在模式的中心是一個所謂的事實表（在這個例子中，它被稱為 `fact_sales`）。事實表的每一行代表在特定時間發生的事件（這裡，每一行代表客戶購買的產品）。如果我們分析的是網站流量而不是零售量，則每行可能代表一個使用者的頁面瀏覽或點選。



[圖 3-9](#) 用於資料倉庫的星型模式的示意

通常情況下，事實被視為單獨的事件，因為這樣可以在以後分析中獲得最大的靈活性。但是，這意味著事實表可以變得非常大。像蘋果、沃爾瑪或 eBay 這樣的大企業在其資料倉庫中可能有幾十 PB 的交易歷史，其中大部分儲存在事實表中 [\[56\]](#)。

事實表中的一些列是屬性，例如產品銷售的價格和從供應商那裡購買的成本（可以用來計算利潤率）。事實表中的其他列是對其他表（稱為維度表）的外來鍵引用。由於事實表中的每一行都表示一個事件，因此這些維度代表事件發生的物件、內容、地點、時間、方式和原因。

例如，在 [圖 3-9](#) 中，其中一個維度是已售出的產品。`dim_product` 表中的每一行代表一種待售產品，包括庫存單位（SKU）、產品描述、品牌名稱、類別、脂肪含量、包裝尺寸等。`fact_sales` 表中的每一行都使用外來鍵表明在特定交易中銷售了什麼產品。（簡單起見，如果客戶一次購買了幾種不同的產品，則它們在事實表中被表示為單獨的行）。

甚至日期和時間也通常使用維度表來表示，因為這允許對日期的附加資訊（諸如公共假期）進行編碼，從而允許區分假期和非假期的銷售查詢。

“星型模式”這個名字來源於這樣一個事實，即當我們對錶之間的關係進行視覺化時，事實表在中間，被維度表包圍；與這些表的連線就像星星的光芒。

這個模板的變體被稱為雪花模式，其中維度被進一步分解為子維度。例如，品牌和產品類別可能有單獨的表格，並且 `dim_product` 表格中的每一行都可以將品牌和類別作為外來鍵引用，而不是將它們作為字串儲存在 `dim_product` 表格中。雪花模式比星形模式更規範化，但是星形模式通常是首選，因為分析師使用它更簡單【55】。

在典型的資料倉庫中，表格通常非常寬：事實表通常有 100 列以上，有時甚至有數百列【51】。維度表也可以是非常寬的，因為它們包括了所有可能與分析相關的元資料——例如，`dim_store` 表可以包括在每個商店提供哪些服務的細節、它是否具有店內麵包房、店面面積、商店第一次開張的日期、最近一次改造的時間、離最近的高速公路的距離等等。

列式儲存

如果事實表中有萬億行和數 PB 的資料，那麼高效地儲存和查詢它們就成為一個具有挑戰性的問題。維度表通常要小得多（數百萬行），所以在本節中我們將主要關注事實表的儲存。

儘管事實表通常超過 100 列，但典型的資料倉庫查詢一次只會訪問其中 4 個或 5 個列（“`SELECT *`”查詢很少用於分析）【51】。以 [例 3-1](#) 中的查詢為例：它訪問了大量的行（在 2013 年中所有購買了水果或糖果的記錄），但只需訪問 `fact_sales` 表的三列：`date_key`, `product_sk`, `quantity`。該查詢忽略了所有其他的列。

例 3-1 分析人們是否更傾向於在一週的某一天購買新鮮水果或糖果

```

SELECT
    dim_date.weekday,
    dim_product.category,
    SUM(fact_sales.quantity) AS quantity_sold
FROM fact_sales
JOIN dim_date ON fact_sales.date_key = dim_date.date_key
JOIN dim_product ON fact_sales.product_sk = dim_product.product_sk
WHERE
    dim_date.year = 2013 AND
    dim_product.category IN ('Fresh fruit', 'Candy')
GROUP BY
    dim_date.weekday, dim_product.category;

```

我們如何有效地執行這個查詢？

在大多數 OLTP 資料庫中，儲存都是以面向行的方式進行佈局的：表格的一行中的所有值都相鄰儲存。文件資料庫也是相似的：整個文件通常儲存為一個連續的位元組序列。你可以在 [圖 3-1](#) 的 CSV 例子中看到這個。

為了處理像 [例 3-1](#) 這樣的查詢，你可能在 `fact_sales.date_key` 和 `fact_sales.product_sk` 上有索引，它們告訴儲存引擎在哪裡查詢特定日期或特定產品的所有銷售情況。但是，面向行的儲存引擎仍然需要將所有這些行（每個包含超過 100 個屬性）從硬碟載入到記憶體中，解析它們，並過濾掉那些不符合要求的屬性。這可能需要很長時間。

列式儲存背後的想法很簡單：不要將所有來自一行的值儲存在一起，而是將來自每一列的所有值儲存在一起。如果每個列式儲存在一個單獨的檔案中，查詢只需要讀取和解析查詢中使用的那些列，這可以節省大量的工作。這個原理如 [圖 3-10](#) 所示。

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	69	4	NULL	NULL	1	13.99	13.99
140102	69	5	19	NULL	3	14.99	9.99
140102	69	5	NULL	191	1	14.99	14.99
140102	74	3	23	202	5	0.99	0.89
140103	31	2	NULL	NULL	1	2.49	2.49
140103	31	3	NULL	NULL	3	14.99	9.99
140103	31	3	21	123	1	49.99	39.99
140103	31	8	NULL	233	1	0.99	0.99

Columnar storage layout:

date_key file contents:	140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103
product_sk file contents:	69, 69, 69, 74, 31, 31, 31, 31
store_sk file contents:	4, 5, 5, 3, 2, 3, 3, 8
promotion_sk file contents:	NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk file contents:	NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity file contents:	1, 3, 1, 5, 1, 3, 1, 1
net_price file contents:	13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price file contents:	13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

圖 3-10 按列儲存關係型資料，而不是行

列式儲存在關係資料模型中是最容易理解的，但它同樣適用於非關係資料。例如，Parquet【57】是一種列式儲存格式，支援基於 Google 的 Dremel 的文件資料模型【54】。

列式儲存佈局依賴於每個列檔案包含相同順序的行。因此，如果你需要重新組裝完整的行，你可以從每個單獨的列檔案中獲取第 23 項，並將它們放在一起形成表的第 23 行。

列壓縮

除了僅從硬碟載入查詢所需的列以外，我們還可以透過壓縮資料來進一步降低對硬碟吞吐量的需求。幸運的是，列式儲存通常很適合壓縮。

看看 圖 3-10 中每一列的值序列：它們通常看起來是相當重複的，這是壓縮的好兆頭。根據列中的資料，可以使用不同的壓縮技術。在資料倉庫中特別有效的一種技術是點陣圖編碼，如 圖 3-11 所示。

Column values:	
product_sk:	69 69 69 69 74 31 31 31 31 29 30 30 31 31 31 68 69 69
Bitmap for each possible value:	
product_sk = 29:	0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
product_sk = 30:	0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
product_sk = 31:	0 0 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
product_sk = 68:	0 1 0 0 0 0 0 0 0
product_sk = 69:	1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
product_sk = 74:	0 0 0 0 1 0
Run-length encoding:	
product_sk = 29:	9, 1 (9 zeros, 1 one, rest zeros)
product_sk = 30:	10, 2 (10 zeros, 2 ones, rest zeros)
product_sk = 31:	5, 4, 3, 3 (5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)
product_sk = 68:	15, 1 (15 zeros, 1 one, rest zeros)
product_sk = 69:	0, 4, 12, 2 (0 zeros, 4 ones, 12 zeros, 2 ones)
product_sk = 74:	4, 1 (4 zeros, 1 one, rest zeros)

圖 3-11 壓縮的點陣圖索引儲存佈局

通常情況下，一列中不同值的數量與行數相比要小得多（例如，零售商可能有數十億的銷售交易，但只有 100,000 個不同的產品）。現在我們可以拿一個有 n 個不同值的列，並把它轉換成 n 個獨立的點陣圖：每個不同值對應一個位圖，每行對應一個位元位。如果該行具有該值，則該位為 1，否則為 0。

如果 n 非常小（例如，國家 / 地區列可能有大約 200 個不同的值），則這些點陣圖可以將每行儲存成一個位元位。但是，如果 n 更大，大部分點陣圖中將會有很多的零（我們說它們是稀疏的）。在這種情況下，點陣圖可以另外再進行遊程編碼（run-length encoding，一種無損資料壓縮技術），如 [圖 3-11](#) 底部所示。這可以使列的編碼非常緊湊。

這些點陣圖索引非常適合資料倉庫中常見的各種查詢。例如：

```
WHERE product_sk IN (30, 68, 69)
```

載入 `product_sk = 30`、`product_sk = 68` 和 `product_sk = 69` 這三個點陣圖，並計算三個點陣圖的按位或（OR），這可以非常有效地完成。

```
WHERE product_sk = 31 AND store_sk = 3
```

載入 `product_sk = 31` 和 `store_sk = 3` 的點陣圖，並計算按位與（AND）。這是因為列按照相同的順序包含行，因此一列的點陣圖中的第 k 位和另一列的點陣圖中的第 k 位對應相同的行。

對於不同種類的資料，也有各種不同的壓縮方案，但我們不會詳細討論它們，請參閱 [【58】](#) 的概述。

列式儲存和列族

Cassandra 和 HBase 有一個列族 (column families) 的概念，他們從 Bigtable 繼承【9】。然而，把它們稱為列式 (column-oriented) 是非常具有誤導性的：在每個列族中，它們將一行中的所有列與行鍵一起儲存，並且不使用列壓縮。因此，Bigtable 模型仍然主要是面向行的。

記憶體頻寬和向量化處理

對於需要掃描數百萬行的資料倉庫查詢來說，一個巨大的瓶頸是從硬盤獲取資料到記憶體的頻寬。但是，這不是唯一的瓶頸。分析型資料庫的開發人員還需要有效地利用記憶體到 CPU 快取的頻寬，避免 CPU 指令處理流水線中的分支預測錯誤和閒置等待，以及在現代 CPU 上使用單指令多資料 (SIMD) 指令來加速運算【59,60】。

除了減少需要從硬碟載入的資料量以外，列式儲存佈局也可以有效利用 CPU 週期。例如，查詢引擎可以將一整塊壓縮好的列資料放進 CPU 的 L1 快取中，然後在緊密的迴圈（即沒有函式呼叫）中遍歷。相比於每條記錄的處理都需要大量函式呼叫和條件判斷的程式碼，CPU 執行這樣一個迴圈要快得多。列壓縮允許列中的更多行被同時放進容量有限的 L1 快取。前面描述的按位“與”和“或”運算子可以被設計為直接在這樣的壓縮列資料塊上操作。這種技術被稱為向量化處理 (vectorized processing) 【58,49】。

列式儲存中的排序順序

在列式儲存中，儲存行的順序並不關鍵。按插入順序儲存它們是最簡單的，因為插入一個新行只需要追加到每個列檔案。但是，我們也可以選擇按某種順序來排列資料，就像我們之前對 SSTables 所做的那樣，並將其用作索引機制。

注意，對每列分別執行排序是沒有意義的，因為那樣就沒法知道不同列中的哪些項屬於同一行。我們只能在明確一列中的第 k 項與另一列中的第 k 項屬於同一行的情況下，才能重建出完整的行。

相反，資料的排序需要對一整行統一操作，即使它們的儲存方式是按列的。資料庫管理員可以根據他們對常用查詢的瞭解，來選擇表格中用來排序的列。例如，如果查詢通常以日期範圍為目標，例如“上個月”，則可以將 `date_key` 作為第一個排序鍵。這樣查詢最佳化器就可以只掃描近1個月範圍的行了，這比掃描所有行要快得多。

對於第一排序列中具有相同值的行，可以用第二排序列來進一步排序。例如，如果 `date_key` 是 圖 3-10 中的第一個排序關鍵字，那麼 `product_sk` 可能是第二個排序關鍵字，以便同一天的同一產品的所有銷售資料都被儲存在相鄰位置。這將有助於需要在特定日期範圍內按產品對銷售進行分組或過濾的查詢。

按順序排序的另一個好處是它可以幫助壓縮列。如果主要排序列沒有太多個不同的值，那麼在排序之後，將會得到一個相同的值連續重複多次的序列。一個簡單的遊程編碼（就像我們用於 圖 3-11 中的點陣圖一樣）可以將該列壓縮到幾 KB —— 即使表中有數十億行。

第一個排序鍵的壓縮效果最強。第二和第三個排序鍵會更混亂，因此不會有這麼長的連續的重複值。排序優先順序更低的列以幾乎隨機的順序出現，所以可能不會被壓縮。但對前幾列做排序在整體上仍然是有好處的。

幾個不同的排序順序

對這個想法，有一個巧妙的擴充套件被 C-Store 發現，並在商業資料倉庫 Vertica 中被採用【61,62】：既然不同的查詢受益於不同的排序順序，為什麼不以幾種不同的方式來儲存相同的資料呢？反正資料都需要做備份，以防單點故障時丟失資料。因此你可以用不同排序方式來儲存冗餘資料，以便在處理查詢時，呼叫最適合查詢模式的版本。

在一個列式儲存中有多個排序順序有點類似於在一個面向行的儲存中有多個次級索引。但最大的區別在於面向行的儲存將每一行儲存在一個地方（在堆檔案或聚集索引中），次級索引只包含指向匹配行的指標。在列式儲存中，通常在其他地方沒有任何指向資料的指標，只有包含值的列。

寫入列式儲存

這些最佳化在資料倉庫中是有意義的，因為其負載主要由分析人員執行的大型只讀查詢組成。列式儲存、壓縮和排序都有助於更快地讀取這些查詢。然而，他們的缺點是寫入更加困難。

使用 B 樹的就地更新方法對於壓縮的列是不可能的。如果你想在排序表的中間插入一行，你很可能不得不重寫所有的列檔案。由於行由列中的位置標識，因此插入必須對所有列進行一致地更新。

幸運的是，本章前面已經看到了一個很好的解決方案：LSM 樹。所有的寫操作首先進入一個記憶體中的儲存，在這裡它們被新增到一個已排序的結構中，並準備寫入硬碟。記憶體中的儲存是面向行還是列的並不重要。當已經積累了足夠的寫入資料時，它們將與硬碟上的列檔案合併，並批次寫入新檔案。這基本上是 Vertica 所做的【62】。

查詢操作需要檢查硬碟上的列資料和記憶體中的最近寫入，並將兩者的結果合併起來。但是，查詢最佳化器對使用者隱藏了這個細節。從分析師的角度來看，透過插入、更新或刪除操作進行修改的資料會立即反映在後續的查詢中。

聚合：資料立方體和物化檢視

並非所有資料倉庫都需要採用列式儲存：傳統的面向行的資料庫和其他一些架構也被使用。然而，列式儲存可以顯著加快專門的分析查詢，所以它正在迅速變得流行起來【51,63】。

資料倉庫的另一個值得一提的方面是物化聚合（materialized aggregates）。如前所述，資料倉庫查詢通常涉及一個聚合函式，如 SQL 中的 COUNT、SUM、AVG、MIN 或 MAX。如果相同的聚合被許多不同的查詢使用，那麼每次都透過原始資料來處理可能太浪費了。為什麼不將一些查詢使用最頻繁的計數或總和快取起來？

建立這種快取的一種方式是物化檢視（Materialized View）。在關係資料模型中，它通常被定義為一個標準（虛擬）檢視：一個類似於表的物件，其內容是一些查詢的結果。不同的是，物化檢視是查詢結果的實際副本，會被寫入硬碟，而虛擬檢視只是編寫查詢的一個捷徑。從虛擬檢視讀取時，SQL 引擎會將其展開到檢視的底層查詢中，然後再處理展開的查詢。

當底層資料發生變化時，物化檢視需要更新，因為它是資料的非規範化副本。資料庫可以自動完成該操作，但是這樣的更新使得寫入成本更高，這就是在 OLTP 資料庫中不經常使用物化檢視的原因。在讀取繁重的資料倉庫中，它們可能更有意義（它們是否實際上改善了讀取效能取決於使用場景）。

物化檢視的常見特例稱為資料立方體或 OLAP 立方【64】。它是按不同維度分組的聚合網格。圖 3-12 顯示了一個例子。

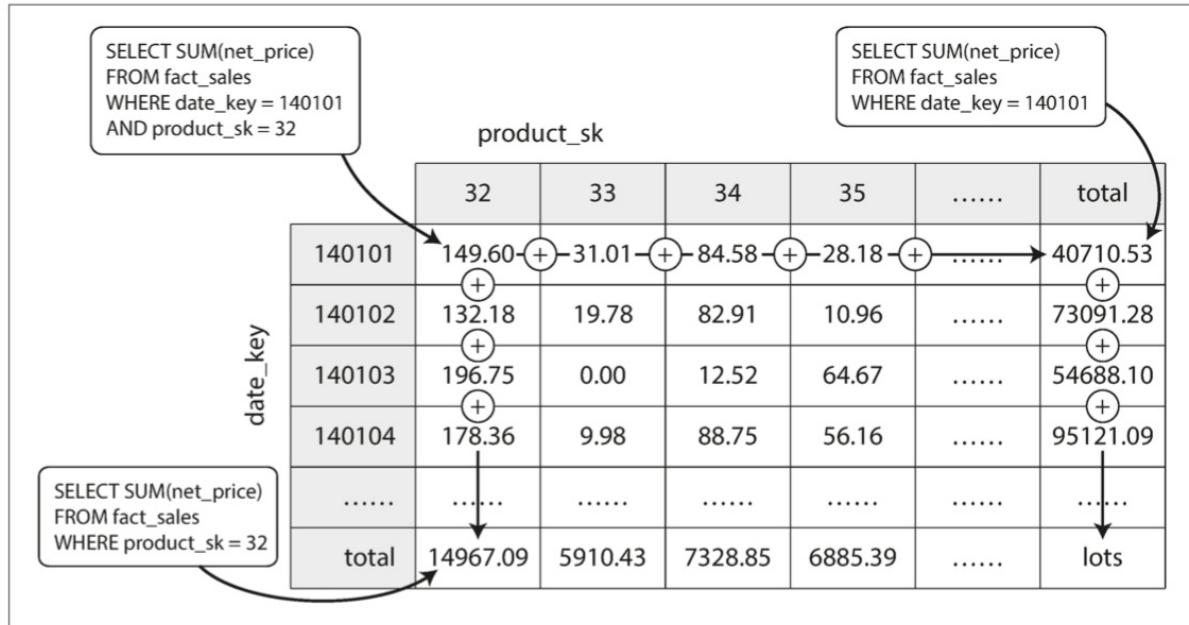


圖 3-12 資料立方的兩個維度，透過求和聚合

想像一下，現在每個事實都只有兩個維度表的外來鍵——在 圖 3-12 中分別是日期和產品。你現在可以繪製一個二維表格，一個軸線上是日期，另一個軸線上是產品。每個單元格包含具有該日期 - 產品組合的所有事實的屬性（例如 `net_price`）的聚合（例如 `SUM`）。然後，你可以沿著每行或每列應用相同的彙總，並獲得減少了一個維度的彙總（按產品的銷售額，無論日期，或者按日期的銷售額，無論產品）。

一般來說，事實往往有兩個以上的維度。在圖 3-9 中有五個維度：日期、產品、商店、促銷和客戶。要想象一個五維超立方體是什麼樣子是很困難的，但是原理是一樣的：每個單元格都包含特定日期 - 產品 - 商店 - 促銷 - 客戶組合的銷售額。這些值可以在每個維度上求和彙總。

物化資料立方體的優點是可以讓某些查詢變得非常快，因為它們已經被有效地預先計算了。例如，如果你想知道每個商店的總銷售額，則只需檢視合適維度的總計，而無需掃描數百萬行的原始資料。

資料立方體的缺點是不具有查詢原始資料的靈活性。例如，沒有辦法計算有多少比例的銷售來自成本超過 100 美元的專案，因為價格不是其中的一個維度。因此，大多數資料倉庫試圖保留儘可能多的原始資料，並將聚合資料（如資料立方體）僅用作某些查詢的效能提升手段。

本章小結

在本章中，我們試圖深入瞭解資料庫是如何處理儲存和檢索的。將資料儲存在資料庫中會發生什麼？稍後再次查詢資料時資料庫會做什麼？

在高層次上，我們看到儲存引擎分為兩大類：針對 **事務處理 (OLTP)** 最佳化的儲存引擎和針對 **線上分析 (OLAP)** 最佳化的儲存引擎。這兩類使用場景的訪問模式之間有很大的區別：

- OLTP 系統通常面向終端使用者，這意味著系統可能會收到大量的請求。為了處理負載，應用程式在每個查詢中通常只訪問少量的記錄。應用程式使用某種鍵來請求記錄，儲存引擎使用索引來查詢所請求的鍵的資料。硬碟查詢時間往往是這裡的瓶頸。
- 資料倉庫和類似的分析系統會少見一些，因為它們主要由業務分析人員使用，而不是終端使用者。它們的查詢量要比 OLTP 系統少得多，但通常每個查詢開銷高昂，需要在短時間內掃描數百萬條記錄。硬碟頻寬（而不是查詢時間）往往是瓶頸，列式儲存是針對這種工作負載的日益流行的解決方案。

在 OLTP 這一邊，我們能看到兩派主流的儲存引擎：

- 日誌結構學派：只允許追加到檔案和刪除過時的檔案，但不會更新已經寫入的檔案。Bitcask、SSTables、LSM 樹、LevelDB、Cassandra、HBase、Lucene 等都屬於這個類別。
- 就地更新學派：將硬碟視為一組可以覆寫的固定大小的頁面。B 樹是這種理念的典範，用在所有主要的關係資料庫和許多非關係型資料庫中。

日誌結構的儲存引擎是相對較新的技術。他們的主要想法是，透過系統性地將隨機訪問寫入轉換為硬碟上的順序寫入，由於硬碟驅動器和固態硬碟的效能特點，可以實現更高的寫入吞吐量。

關於 OLTP，我們最後還介紹了一些更複雜的索引結構，以及針對所有資料都放在記憶體裡而最佳化的資料庫。

然後，我們暫時放下了儲存引擎的內部細節，查看了典型資料倉庫的高階架構，並說明了為什麼分析工作負載與 OLTP 差別很大：當你的查詢需要在大量行中順序掃描時，索引的重要性就會降低很多。相反，非常緊湊地編碼資料變得非常重要，以最大限度地減少查詢需要從硬碟讀取的資料量。我們討論了列式儲存如何幫助實現這一目標。

作為一名應用程式開發人員，如果你掌握了有關儲存引擎內部的知識，那麼你就能更好地瞭解哪種工具最適合你的特定應用程式。當你調整資料庫的最佳化引數時，這種理解讓你能夠設想增減某個值會產生怎樣的效果。

儘管本章不能讓你成為一個特定儲存引擎的調參專家，但它至少大機率使你有了足夠的概念與詞彙儲備去讀懂你所選擇的資料庫的文件。

參考文獻

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman: *Data Structures and Algorithms*. Addison-Wesley, 1983. ISBN: 978-0-201-00023-8
2. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein: *Introduction to Algorithms*, 3rd edition. MIT Press, 2009. ISBN: 978-0-262-53305-8
3. Justin Sheehy and David Smith: “[Bitcask: A Log-Structured Hash Table for Fast Key/Value Data](#),” Basho

- Technologies, April 2010.
4. Yinan Li, Bingsheng He, Robin Jun Yang, et al.: “[Tree Indexing on Solid State Drives](#),” *Proceedings of the VLDB Endowment*, volume 3, number 1, pages 1195–1206, September 2010.
 5. Goetz Graefe: “[Modern B-Tree Techniques](#),” *Foundations and Trends in Databases*, volume 3, number 4, pages 203–402, August 2011. doi:[10.1561/1900000028](https://doi.org/10.1561/1900000028)
 6. Jeffrey Dean and Sanjay Ghemawat: “[LevelDB Implementation Notes](#),” leveldb.googlecode.com.
 7. Dhruba Borthakur: “[The History of RocksDB](#),” rocksdb.blogspot.com, November 24, 2013.
 8. Matteo Bertozi: “[Apache HBase I/O – HFile](#),” blog.cloudera.com, June, 29 2012.
 9. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al.: “[Bigtable: A Distributed Storage System for Structured Data](#),” at *7th USENIX Symposium on Operating System Design and Implementation (OSDI)*, November 2006.
 10. Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil: “[The Log-Structured Merge-Tree \(LSM Tree\)](#),” *Acta Informatica*, volume 33, number 4, pages 351–385, June 1996. doi:[10.1007/s002360050048](https://doi.org/10.1007/s002360050048)
 11. Mendel Rosenblum and John K. Ousterhout: “[The Design and Implementation of a Log-Structured File System](#),” *ACM Transactions on Computer Systems*, volume 10, number 1, pages 26–52, February 1992. doi:[10.1145/146941.146943](https://doi.org/10.1145/146941.146943)
 12. Adrien Grand: “[What Is in a Lucene Index?](#),” at *Lucene/Solr Revolution*, November 14, 2013.
 13. Deepak Kandepet: “[Hacking Lucene—The Index Format](#),” hackerlabs.org, October 1, 2011.
 14. Michael McCandless: “[Visualizing Lucene’s Segment Merges](#),” blog.mikemccandless.com, February 11, 2011.
 15. Burton H. Bloom: “[Space/Time Trade-offs in Hash Coding with Allowable Errors](#),” *Communications of the ACM*, volume 13, number 7, pages 422–426, July 1970. doi:[10.1145/362686.362692](https://doi.org/10.1145/362686.362692)
 16. “[Operating Cassandra: Compaction](#),” Apache Cassandra Documentation v4.0, 2016.
 17. Rudolf Bayer and Edward M. McCreight: “[Organization and Maintenance of Large Ordered Indices](#),” Boeing Scientific Research Laboratories, Mathematical and Information Sciences Laboratory, report no. 20, July 1970.
 18. Douglas Comer: “[The Ubiquitous B-Tree](#),” *ACM Computing Surveys*, volume 11, number 2, pages 121–137, June 1979. doi:[10.1145/356770.356776](https://doi.org/10.1145/356770.356776)
 19. Emmanuel Goossaert: “[Coding for SSDs](#),” codecapsule.com, February 12, 2014.
 20. C. Mohan and Frank Levine: “[ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging](#),” at *ACM International Conference on Management of Data (SIGMOD)*, June 1992. doi:[10.1145/130283.130338](https://doi.org/10.1145/130283.130338)
 21. Howard Chu: “[LDAP at Lightning Speed](#),” at *Build Stuff ’14*, November 2014.
 22. Bradley C. Kuszmaul: “[A Comparison of Fractal Trees to Log-Structured Merge \(LSM\) Trees](#),” tokutek.com, April 22, 2014.
 23. Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, et al.: “[Designing Access Methods: The RUM Conjecture](#),” at *19th International Conference on Extending Database Technology (EDBT)*, March 2016. doi:[10.5441/002/edbt.2016.42](https://doi.org/10.5441/002/edbt.2016.42)
 24. Peter Zaitsev: “[Innodb Double Write](#),” percona.com, August 4, 2006.
 25. Tomas Vondra: “[On the Impact of Full-Page Writes](#),” blog.2ndquadrant.com, November 23, 2016.
 26. Mark Callaghan: “[The Advantages of an LSM vs a B-Tree](#),” smalldatum.blogspot.co.uk, January 19, 2016.
 27. Mark Callaghan: “[Choosing Between Efficiency and Performance with RocksDB](#),” at *Code Mesh*, November 4, 2016.
 28. Michi Mutsuzaki: “[MySQL vs. LevelDB](#),” github.com, August 2011.
 29. Benjamin Coverston, Jonathan Ellis, et al.: “[CASSANDRA-1608: Redesigned Compaction](#),” issues.apache.org, July 2011.
 30. Igor Canadi, Siying Dong, and Mark Callaghan: “[RocksDB Tuning Guide](#),” github.com, 2016.
 31. *MySQL 5.7 Reference Manual*. Oracle, 2014.
 32. *Books Online for SQL Server 2012*. Microsoft, 2012.
 33. Joe Webb: “[Using Covering Indexes to Improve Query Performance](#),” simple-talk.com, 29 September 2008.
 34. Frank Ramsak, Volker Markl, Robert Fenk, et al.: “[Integrating the UB-Tree into a Database System Kernel](#),” at *26th International Conference on Very Large Data Bases (VLDB)*, September 2000.
 35. The PostGIS Development Group: “[PostGIS 2.1.2dev Manual](#),” postgis.net, 2014.
 36. Robert Escriva, Bernard Wong, and Emin Gün Sirer: “[HyperDex: A Distributed, Searchable Key-Value Store](#),” at

- ACM SIGCOMM Conference, August 2012. doi:10.1145/2377677.2377681
37. Michael McCandless: “Lucene’s FuzzyQuery Is 100 Times Faster in 4.0,” blog.mikemccandless.com, March 24, 2011.
 38. Steffen Heinz, Justin Zobel, and Hugh E. Williams: “Burst Tries: A Fast, Efficient Data Structure for String Keys,” *ACM Transactions on Information Systems*, volume 20, number 2, pages 192–223, April 2002. doi:10.1145/506309.506312
 39. Klaus U. Schulz and Stoyan Mihov: “Fast String Correction with Levenshtein Automata,” *International Journal on Document Analysis and Recognition*, volume 5, number 1, pages 67–85, November 2002. doi:10.1007/s10032-002-0082-8
 40. Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze: *Introduction to Information Retrieval*. Cambridge University Press, 2008. ISBN: 978-0-521-86571-5, available online at nlp.stanford.edu/IR-book
 41. Michael Stonebraker, Samuel Madden, Daniel J. Abadi, et al.: “The End of an Architectural Era (It’s Time for a Complete Rewrite),” at *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.
 42. “VoltDB Technical Overview White Paper,” VoltDB, 2014.
 43. Stephen M. Rumble, Ankita Kejriwal, and John K. Ousterhout: “Log-Structured Memory for DRAM-Based Storage,” at *12th USENIX Conference on File and Storage Technologies* (FAST), February 2014.
 44. Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker: “OLTP Through the Looking Glass, and What We Found There,” at *ACM International Conference on Management of Data* (SIGMOD), June 2008. doi:10.1145/1376616.1376713
 45. Justin DeBrabant, Andrew Pavlo, Stephen Tu, et al.: “Anti-Caching: A New Approach to Database Management System Architecture,” *Proceedings of the VLDB Endowment*, volume 6, number 14, pages 1942–1953, September 2013.
 46. Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor: “Let’s Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems,” at *ACM International Conference on Management of Data* (SIGMOD), June 2015. doi:10.1145/2723372.2749441
 47. Edgar F. Codd, S. B. Codd, and C. T. Salley: “Providing OLAP to User-Analysts: An IT Mandate,” E. F. Codd Associates, 1993.
 48. Surajit Chaudhuri and Umeshwar Dayal: “An Overview of Data Warehousing and OLAP Technology,” *ACM SIGMOD Record*, volume 26, number 1, pages 65–74, March 1997. doi:10.1145/248603.248616
 49. Per-Åke Larson, Cipri Clinciu, Campbell Fraser, et al.: “Enhancements to SQL Server Column Stores,” at *ACM International Conference on Management of Data* (SIGMOD), June 2013.
 50. Franz Färber, Norman May, Wolfgang Lehner, et al.: “The SAP HANA Database – An Architecture Overview,” *IEEE Data Engineering Bulletin*, volume 35, number 1, pages 28–33, March 2012.
 51. Michael Stonebraker: “The Traditional RDBMS Wisdom Is (Almost Certainly) All Wrong,” presentation at EPFL, May 2013.
 52. Daniel J. Abadi: “Classifying the SQL-on-Hadoop Solutions,” hadapt.com, October 2, 2013.
 53. Marcel Kornacker, Alexander Behm, Victor Bittorf, et al.: “Impala: A Modern, Open-Source SQL Engine for Hadoop,” at *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.
 54. Sergey Melnik, Andrey Gubarev, Jing Jing Long, et al.: “Dremel: Interactive Analysis of Web-Scale Datasets,” at *36th International Conference on Very Large Data Bases* (VLDB), pages 330–339, September 2010.
 55. Ralph Kimball and Margy Ross: *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd edition. John Wiley & Sons, July 2013. ISBN: 978-1-118-53080-1
 56. Derrick Harris: “Why Apple, eBay, and Walmart Have Some of the Biggest Data Warehouses You’ve Ever Seen,” gigaom.com, March 27, 2013.
 57. Julien Le Dem: “Dremel Made Simple with Parquet,” blog.twitter.com, September 11, 2013.
 58. Daniel J. Abadi, Peter Boncz, Stavros Harizopoulos, et al.: “The Design and Implementation of Modern Column-Oriented Database Systems,” *Foundations and Trends in Databases*, volume 5, number 3, pages 197–280, December 2013. doi:10.1561/1900000024
 59. Peter Boncz, Marcin Zukowski, and Niels Nes: “MonetDB/X100: Hyper-Pipelining Query Execution,” at *2nd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2005.
 60. Jingren Zhou and Kenneth A. Ross: “Implementing Database Operations Using SIMD Instructions,” at *ACM*

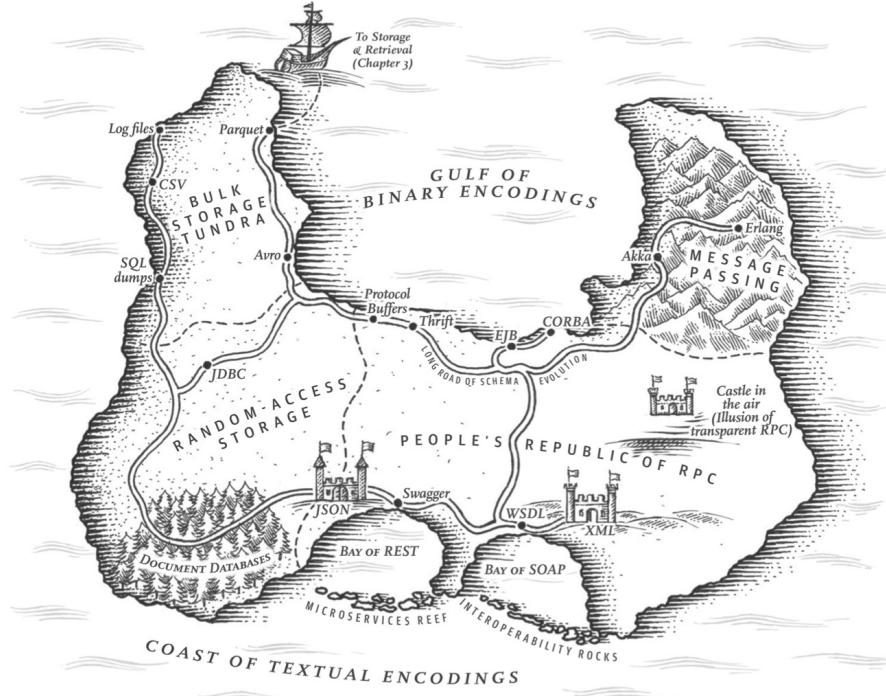
International Conference on Management of Data (SIGMOD), pages 145–156, June 2002.

[doi:10.1145/564691.564709](https://doi.org/10.1145/564691.564709)

61. Michael Stonebraker, Daniel J. Abadi, Adam Batkin, et al.: “[C-Store: A Column-oriented DBMS](#),” at *31st International Conference on Very Large Data Bases (VLDB)*, pages 553–564, September 2005.
 62. Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, et al.: “[The Vertica Analytic Database: C-Store 7 Years Later](#),” *Proceedings of the VLDB Endowment*, volume 5, number 12, pages 1790–1801, August 2012.
 63. Julien Le Dem and Nong Li: “[Efficient Data Storage for Analytics with Apache Parquet 2.0](#),” at *Hadoop Summit*, San Jose, June 2014.
 64. Jim Gray, Surajit Chaudhuri, Adam Bosworth, et al.: “[Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals](#),” *Data Mining and Knowledge Discovery*, volume 1, number 1, pages 29–53, March 2007. [doi:10.1023/A:1009726021843](https://doi.org/10.1023/A:1009726021843)
-

上一章	目錄	下一章
第二章：資料模型與查詢語言	設計資料密集型應用	第四章：編碼與演化

第四章：編碼與演化



唯變所適

——以弗所的赫拉克利特，為柏拉圖所引（公元前 360 年）

[TOC]

應用程式不可避免地隨時間而變化。新產品的推出，對需求的深入理解，或者商業環境的變化，總會伴隨著 **功能 (feature)** 的增增改改。[第一章](#) 介紹了 **可演化性 (evolvability)** 的概念：應該盡力構建能靈活適應變化的系統（請參閱 “[可演化性：擁抱變化](#)”）。

在大多數情況下，修改應用程式的功能也意味著需要更改其儲存的資料：可能需要使用新的欄位或記錄型別，或者以新方式展示現有資料。

我們在 [第二章](#) 討論的資料模型有不同的方法來應對這種變化。關係資料庫通常假定資料庫中的所有資料都遵循一個模式：儘管可以更改該模式（透過模式遷移，即 `ALTER` 語句），但是在任何時間點都有且僅有一個正確的模式。相比之下，**讀時模式 (schema-on-read)**，或 **無模式 (schemaless)** 資料庫不會強制一個模式，因此資料庫可以包含在不同時間寫入的新老資料格式的混合（請參閱 [“文件模型中的模式靈活性”](#)）。

當資料 **格式 (format)** 或 **模式 (schema)** 發生變化時，通常需要對應用程式程式碼進行相應的更改（例如，為記錄新增新欄位，然後修改程式開始讀寫該欄位）。但在大型應用程式中，程式碼變更通常不會立即完成：

- 對於 **服務端 (server-side)** 應用程式，可能需要執行 **滾動升級 (rolling upgrade)**（也稱為 **階段釋出 (staged rollout)**），一次將新版本部署到少數幾個節點，檢查新版本是否執行正常，然後逐漸部完所有的節點。這樣無需中斷服務即可部署新版本，為頻繁釋出提供了可行性，從而帶來更好的可演化性。
- 對於 **客戶端 (client-side)** 應用程式，升不升級就要看使用者的心情了。使用者可能相當長一段時間裡都不會去

升級軟體。

這意味著，新舊版本的程式碼，以及新舊資料格式可能會在系統中同時共處。系統想要繼續順利執行，就需要保持 **雙向相容性**：

- 向後相容 (backward compatibility)

新的程式碼可以讀取由舊的程式碼寫入的資料。

- 向前相容 (forward compatibility)

舊的程式碼可以讀取由新的程式碼寫入的資料。

向後相容性通常並不難實現：新程式碼的作者當然知道由舊程式碼使用的資料格式，因此可以顯示地處理它（最簡單的辦法是，保留舊程式碼即可讀取舊資料）。

向前相容性可能會更棘手，因為舊版的程式需要忽略新版資料格式中新增的部分。

本章中將介紹幾種編碼資料的格式，包括 JSON、XML、Protocol Buffers、Thrift 和 Avro。尤其將關注這些格式如何應對模式變化，以及它們如何對新舊程式碼資料需要共存的系統提供支援。然後將討論如何使用這些格式進行資料儲存和通訊：在 Web 服務中，**表述性狀態傳遞 (REST)** 和 **遠端過程呼叫 (RPC)**，以及 **訊息傳遞系統**（如 Actor 和訊息佇列）。

編碼資料的格式

程式通常（至少）使用兩種形式的資料：

1. 在記憶體中，資料儲存在物件、結構體、列表、陣列、散列表、樹等中。這些資料結構針對 CPU 的高效訪問和操作進行了最佳化（通常使用指標）。
2. 如果要將資料寫入檔案，或透過網路傳送，則必須將其 **編碼 (encode)** 為某種自包含的位元組序列（例如，JSON 文件）。由於每個程序都有自己獨立的地址空間，一個程序中的指標對任何其他程序都沒有意義，所以這個位元組序列表示會與通常在記憶體中使用的資料結構完全不同ⁱ。

ⁱ. 除一些特殊情況外，例如某些記憶體對映檔案或直接在壓縮資料上操作（如“**列壓縮**”中所述）。 ↵

所以，需要在兩種表示之間進行某種型別的翻譯。從記憶體中表示到位元組序列的轉換稱為 **編碼 (Encoding)**（也稱為 **序列化 (serialization)** 或 **編組 (marshalling)**），反過來稱為 **解碼 (Decoding)**ⁱⁱ（**解析 (Parsing)**，反序序列化 (deserialization)，反編組 (unmarshalling)）ⁱ。

ⁱⁱ. 請注意，**編碼 (encode)** 與 **加密 (encryption)** 無關。本書不討論加密。 ↵

ⁱ. Marshal 與 Serialization 的區別：Marshal 不僅傳輸物件的狀態，而且會一起傳輸物件的方法（相關程式碼）。 ↵

術語衝突

不幸的是，在 **第七章：事務 (Transaction)** 的上下文裡，**序列化 (Serialization)** 這個術語也出現了，而且具有完全不同的含義。儘管序列化可能是更常見的術語，為了避免術語過載，本書中堅持使用 **編碼 (Encoding)** 表達此含義。

這是一個常見的問題，因而有許多庫和編碼格式可供選擇。首先讓我們概覽一下。

語言特定的格式

許多程式語言都內建了將記憶體物件編碼為位元組序列的支援。例如，Java 有 `java.io.Serializable` [\[1\]](#)，Ruby 有 `Marshal` [\[2\]](#)，Python 有 `pickle` [\[3\]](#)，等等。許多第三方庫也存在，例如 `Kryo for Java` [\[4\]](#)。

這些編碼庫非常方便，可以用很少的額外程式碼實現記憶體物件的儲存與恢復。但是它們也有一些深層次的問題：

- 這類編碼通常與特定的程式語言深度繫結，其他語言很難讀取這種資料。如果以這類編碼儲存或傳輸資料，那你就和這門語言綁死在一起了。並且很難將系統與其他組織的系統（可能用的是不同的語言）進行整合。
- 為了恢復相同物件型別的資料，解碼過程需要 **例項化任意類** 的能力，這通常是安全問題的一個來源【5】：如果攻擊者可以讓應用程式解碼任意的位元組序列，他們就能例項化任意的類，這會允許他們做可怕的事情，如遠端執行任意程式碼【6,7】。
- 在這些庫中，資料版本控制通常是事後才考慮的。因為它們旨在快速簡便地對資料進行編碼，所以往往忽略了前向後向相容性帶來的麻煩問題。
- 效率（編碼或解碼所花費的 CPU 時間，以及編碼結構的大小）往往也是事後才考慮的。例如，Java 的內建序列化由於其糟糕的效能和臃腫的編碼而臭名昭著【8】。

因此，除非臨時使用，採用語言內建編碼通常是一個壞主意。

JSON、XML 和二進位制變體

當我們談到可以被多種程式語言讀寫的標準編碼時，JSON 和 XML 是最顯眼的角逐者。它們廣為人知，廣受支援，也“廣受憎惡”。XML 經常收到批評：過於冗長與且過份複雜【9】。JSON 的流行則主要源於（透過成為 JavaScript 的一個子集）Web 瀏覽器的內建支援，以及相對於 XML 的簡單性。CSV 是另一種流行的與語言無關的格式，儘管其功能相對較弱。

JSON，XML 和 CSV 屬於文字格式，因此具有人類可讀性（儘管它們的語法是一個熱門爭議話題）。除了表面的語法問題之外，它們也存在一些微妙的問題：

- **數字 (numbers)** 編碼有很多模糊之處。在 XML 和 CSV 中，無法區分數字和碰巧由數字組成的字串（除了引用外部模式）。JSON 雖然區分子串與數字，但並不區分整數和浮點數，並且不能指定精度。這在處理大數字時是個問題。例如大於 $\$2^{53}$ 的整數無法使用 IEEE 754 雙精度浮點數精確表示，因此在使用浮點數（例如 JavaScript）的語言進行分析時，這些數字會變得不準確。Twitter 有一個關於大於 $\$2^{53}$ 的數字的例子，它使用 64 位整數來標識每條推文。Twitter API 返回的 JSON 包含了兩個推特 ID，一個是 JSON 數字，另一個是十進位制字串，以解決 JavaScript 程式中無法正確解析數字的問題【10】。
- JSON 和 XML 對 Unicode 字串（即人類可讀的文字）有很好的支援，但是它們不支援二進位制資料（即不帶 **字元編碼 (character encoding)** 的位元組序列）。二進位制串是很有用的功能，人們透過使用 Base64 將二進位制資料編碼為文字來繞過此限制。其特有的模式標識著這個值應當被解釋為 Base64 編碼的二進位制資料。這種方案雖然管用，但比較 Hacky，並且會增加三分之一的資料大小。
- XML 【11】和 JSON 【12】都有可選的模式支援。這些模式語言相當強大，所以學習和實現起來都相當複雜。XML 模式的使用相當普遍，但許多基於 JSON 的工具才不會去折騰模式。對資料的正確解讀（例如區分數值與二進位制串）取決於模式中的資訊，因此不使用 XML/JSON 模式的應用程式可能需要對相應的編碼 / 解碼邏輯進行硬編碼。
- CSV 沒有任何模式，因此每行和每列的含義完全由應用程式自行定義。如果應用程式變更添加了新的行或列，那麼這種變更必須透過手工處理。CSV 也是一個相當模糊的格式（如果一個值包含逗號或換行符，會發生什麼？）。儘管其轉義規則已經被正式指定【13】，但並不是所有的解析器都正確的實現了標準。

儘管存在這些缺陷，但 JSON、XML 和 CSV 對很多需求來說已經足夠好了。它們很可能會繼續流行下去，特別是作為資料交換格式來說（即將資料從一個組織傳送到另一個組織）。在這種情況下，只要人們對格式是什麼意見一致，格式有多美觀或者效率有多高效就無所謂了。讓不同的組織就這些東西達成一致的難度超過了絕大多數問題。

二進位制編碼

對於僅在組織內部使用的資料，使用最小公約數式的編碼格式壓力較小。例如，可以選擇更繁湊或更快的解析格式。雖然對小資料集來說，收益可以忽略不計；但一旦達到 TB 級別，資料格式的選型就會產生巨大的影響。

JSON 比 XML 簡潔，但與二進位制格式相比還是太佔空間。這一事實導致大量二進位制編碼版本 JSON (MessagePack、BSON、BJSON、UBJSON、BISON 和 Smile 等) 和 XML (例如 WBXML 和 Fast Infoset) 的出現。這些格式已經在各種各樣的領域中採用，但是沒有一個能像文字版 JSON 和 XML 那樣被廣泛採用。

這些格式中的一些擴充套件了一組資料型別（例如，區分整數和浮點數，或者增加對二進位制字串的支援），另一方面，它們沒有改變 JSON / XML 的資料模型。特別是由於它們沒有規定模式，所以它們需要在編碼資料中包含所有的物件欄位名稱。也就是說，在 [例 4-1](#) 中的 JSON 文件的二進位制編碼中，需要在某處包含字串

`userName`，`favoriteNumber` 和 `interests`。

例 4-1 本章中用於展示二進位制編碼的示例記錄

```
{
  "userName": "Martin",
  "favoriteNumber": 1337,
  "interests": ["daydreaming", "hacking"]
}
```

我們來看一個 MessagePack 的例子，它是一個 JSON 的二進位制編碼。圖 4-1 顯示瞭如果使用 MessagePack [【14】](#) 對 [例 4-1](#) 中的 JSON 文件進行編碼，則得到的位元組序列。前幾個位元組如下：

- 第一個位元組 `0x83` 表示接下來是 3 個欄位（低四位 = `0x03`）的 物件 **object**（高四位 = `0x80`）。（如果想知道如果一個物件有 15 個以上的欄位會發生什麼情況，欄位的數量塞不進 4 個 bit 裡，那麼它會用另一個不同的型別識別符號，欄位的數量被編碼兩個或四個位元組）。
- 第二個位元組 `0xa8` 表示接下來是 8 位元組長（低四位 = `0x08`）的字串（高四位 = `0x0a`）。
- 接下來八個位元組是 ASCII 字串形式的欄位名稱 `userName`。由於之前已經指明長度，不需要任何標記來標識字串的結束位置（或者任何轉義）。
- 接下來的七個位元組對字首為 `0xa6` 的六個字母的字串值 `Martin` 進行編碼，依此類推。

二進位制編碼長度為 66 個位元組，僅略小於文字 JSON 編碼所取的 81 個位元組（刪除了空白）。所有的 JSON 的二進位制編碼在這方面是相似的。空間節省了一丁點（以及解析加速）是否能彌補可讀性的損失，誰也說不準。

在下面的章節中，能達到比這好得多的結果，只用 32 個位元組對相同的記錄進行編碼。

MessagePack

Byte sequence (66 bytes):

83	a8	75	73	65	72	4e	61	6d	65	a6	4d	61	72	74	69	6e	ae	66	61
76	6f	72	69	74	65	4e	75	6d	62	65	72	cd	05	39	a9	69	6e	74	65
72	65	73	74	73	92	ab	64	61	79	64	72	65	61	6d	69	6e	67	a7	68
61	63	6b	69	6e	67														

Breakdown:

object (3 entries)	string (length 8)	u s e r N a m e	string (length 6)	M a r t i n
83	a8	75 73 65 72 4e 61 6d 65	a6	4d 61 72 74 69 6e
	string (length 14)	f a v o r i t e N u m b e r		
	ae	66 61 76 6f 72 69 74 65 4e 75 6d 62 65 72		
	uint16	1337	string (length 9)	i n t e r e s t s
	cd	05 39	a9	69 6e 74 65 72 65 73 74 73
array (2 entries)	string (length 11)	d a y d r e a m i n g		
92	ab	64 61 79 64 72 65 61 6d 69 6e 67		
	string (length 7)	h a c k i n g		
	a7	68 61 63 6b 69 6e 67		

圖 4-1 使用 MessagePack 編碼的記錄（例 4-1）

Thrift 與 Protocol Buffers

Apache Thrift 【15】和 Protocol Buffers (protobuf) 【16】是基於相同原理的二進位制編碼庫。Protocol Buffers 最初是在 Google 開發的，Thrift 最初是在 Facebook 開發的，並且都是在 2007~2008 開源的【17】。Thrift 和 Protocol Buffers 都需要一個模式來編碼任何資料。要在 Thrift 的 [例 4-1](#) 中對資料進行編碼，可以使用 Thrift 介面定義語言 (IDL) 來描述模式，如下所示：

```
struct Person {
    1: required string      userName,
    2: optional int64       favoriteNumber,
    3: optional list<string> interests
}
```

Protocol Buffers 的等效模式定義看起來非常相似：

```
message Person {
    required string user_name      = 1;
    optional int64 favorite_number = 2;
    repeated string interests     = 3;
}
```

Thrift 和 Protocol Buffers 每一個都帶有一個程式碼生成工具，它採用了類似於這裡所示的模式定義，並且生成了以各種程式語言實現模式的類【18】。你的應用程式程式碼可以呼叫此生成的程式碼來對模式的記錄進行編碼或解碼。用這個模式編碼的資料是什麼樣的？令人困惑的是，Thrift 有兩種不同的二進位制編碼格式ⁱⁱⁱ，分別稱為 BinaryProtocol 和 CompactProtocol。先來看看 BinaryProtocol。使用這種格式的編碼來編碼 例 4-1 中的訊息只需要 59 個位元組，如 圖 4-2 所示【19】。

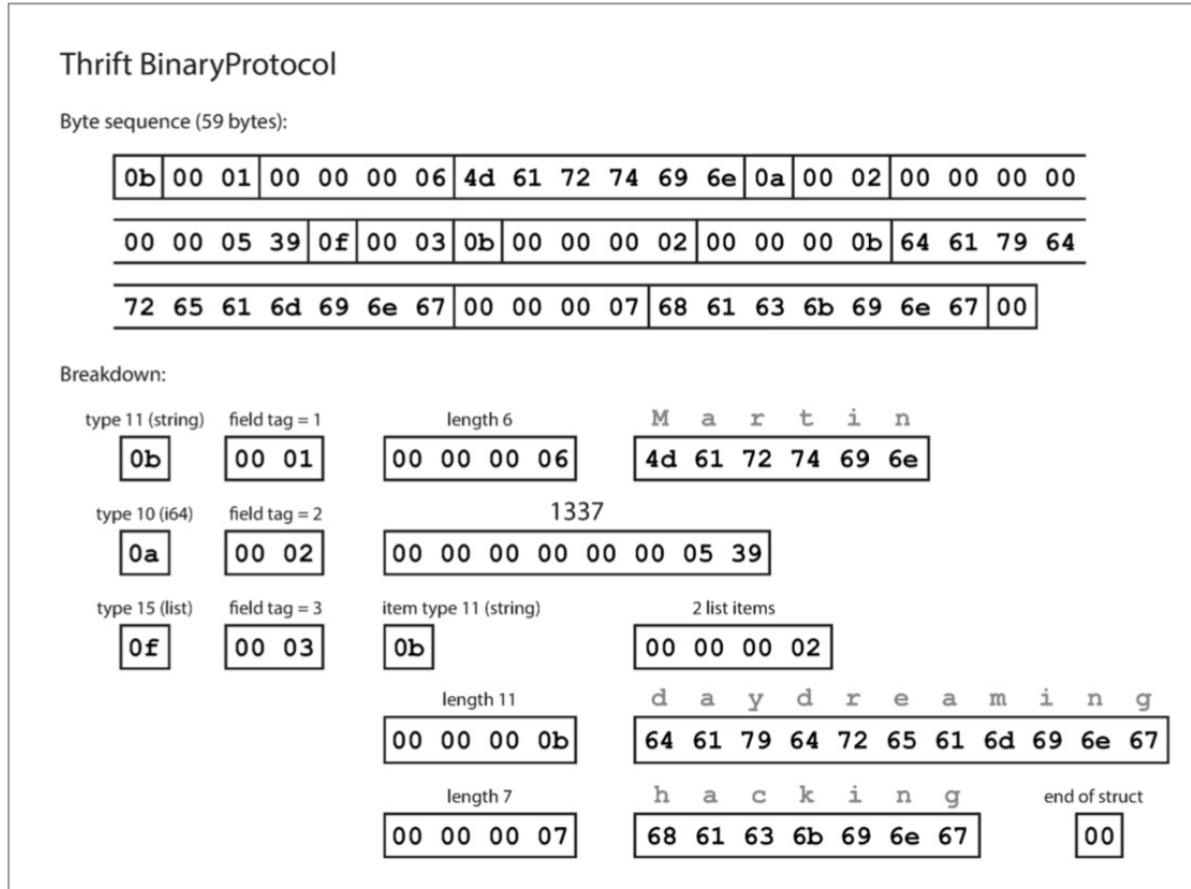


圖 4-2 使用 Thrift 二進位制協議編碼的記錄

ⁱⁱⁱ. 實際上，Thrift 有三種二進位制協議：BinaryProtocol、CompactProtocol 和 DenseProtocol，儘管 DenseProtocol 只支援 C++ 實現，所以不算作跨語言【18】。除此之外，它還有兩種不同的基於 JSON 的編碼格式【19】。真逗！ ↵

與 圖 4-1 類似，每個欄位都有一個型別註釋（用於指示它是一個字串、整數、列表等），還可以根據需要指定長度（字串的長度，列表中的專案數）。出現在資料中的字串（“Martin”，“daydreaming”，“hacking”）也被編碼為 ASCII（或者說，UTF-8），與之前類似。

與 圖 4-1 相比，最大的區別是沒有欄位名（`userName`, `favoriteNumber`, `interests`）。相反，編碼資料包含欄位標籤，它們是數字（1, 2 和 3）。這些是模式定義中出現的數字。欄位標記就像欄位的別名 - 它們是說我們正在談論的欄位的一種緊湊的方式，而不必拼出欄位名稱。

Thrift CompactProtocol 編碼在語義上等同於 BinaryProtocol，但是如 圖 4-3 所示，它只將相同的資訊打包成只有 34 個位元組。它透過將欄位型別和標籤號打包到單個位元組中，並使用可變長度整數來實現。數字 1337 不是使用全部八個位元組，而是用兩個位元組編碼，每個位元組的最高位用來指示是否還有更多的位元組。這意味著 -64 到 63 之間的數字被編碼為一個位元組，-8192 和 8191 之間的數字以兩個位元組編碼，等等。較大的數字使用更多的位元組。

Thrift CompactProtocol

Byte sequence (34 bytes):

18	06	4d	61	72	74	69	6e	16	f2	14	19	28	0b	64	61	79	64	72	65
61	6d	69	6e	67	07	68	61	63	6b	69	6e	67	00						

Breakdown:

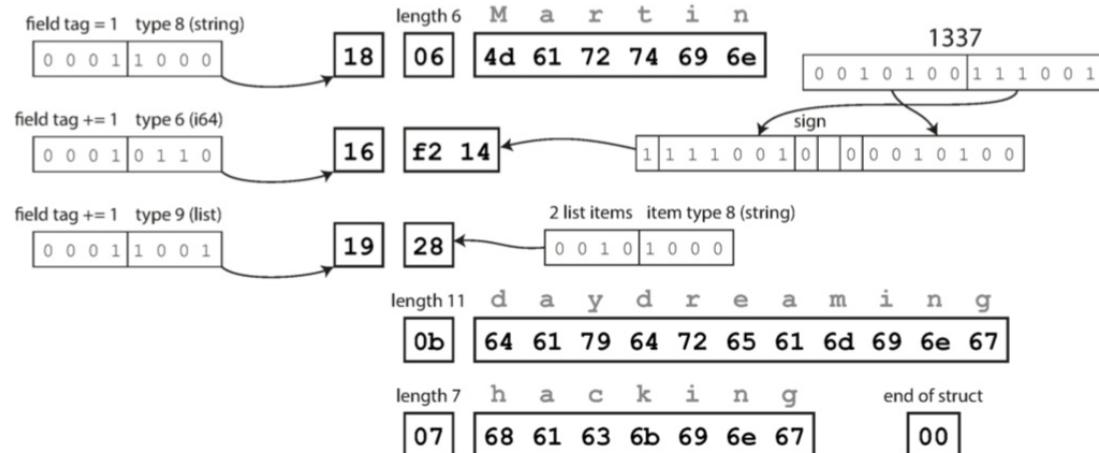


圖 4-3 使用 Thrift 壓縮協議編碼的記錄

最後，Protocol Buffers（只有一種二進位制編碼格式）對相同的資料進行編碼，如 圖 4-4 所示。它的打包方式稍有不同，但與 Thrift 的 CompactProtocol 非常相似。Protobuf 將同樣的記錄塞進了 33 個位元組中。

Protocol Buffers

Byte sequence (33 bytes):

0a	06	4d	61	72	74	69	6e	10	b9	0a	1a	0b	64	61	79	64	72	65	61
6d	69	6e	67	1a	07	68	61	63	6b	69	6e	67	00						

Breakdown:

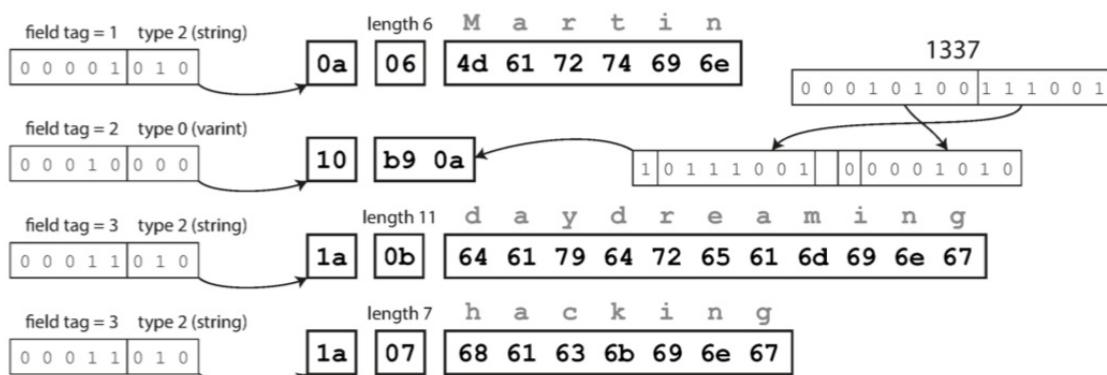


圖 4-4 使用 Protobuf 編碼的記錄

需要注意的一個細節：在前面所示的模式中，每個欄位被標記為必需或可選，但是這對欄位如何編碼沒有任何影響（二進位制資料中沒有任何欄位指示某欄位是否必須）。區別在於，如果欄位設定為 `required`，但未設定該欄位，則所需的執行時檢查將失敗，這對於捕獲錯誤非常有用。

欄位標籤和模式演變

我們之前說過，模式不可避免地需要隨著時間而改變。我們稱之為模式演變。Thrift 和 Protocol Buffers 如何處理模式更改，同時保持向後相容性？

從示例中可以看出，編碼的記錄就是其編碼欄位的拼接。每個欄位由其標籤號碼（樣本模式中的數字 1,2,3）標識，並用資料型別（例如字串或整數）註釋。如果沒有設定欄位值，則簡單地從編碼記錄中省略。從中可以看到，欄位標記對編碼資料的含義至關重要。你可以更改架構中欄位的名稱，因為編碼的資料永遠不會引用欄位名稱，但不能更改欄位的標記，因為這會使所有現有的編碼資料無效。

你可以新增新的欄位到架構，只要你給每個欄位一個新的標籤號碼。如果舊的程式碼（不知道你新增的新的標籤號碼）試圖讀取新程式碼寫入的資料，包括一個新的欄位，其標籤號碼不能識別，它可以簡單地忽略該欄位。資料型別註釋允許解析器確定需要跳過的位元組數。這保持了向前相容性：舊程式碼可以讀取由新程式碼編寫的記錄。

向後相容性呢？只要每個欄位都有一個唯一的標籤號碼，新的程式碼總是可以讀取舊的資料，因為標籤號碼仍然具有相同的含義。唯一的細節是，如果你新增一個新的欄位，你不能設定為必需。如果你要新增一個欄位並將其設定為必需，那麼如果新程式碼讀取舊程式碼寫入的資料，則該檢查將失敗，因為舊程式碼不會寫入你新增的新欄位。因此，為了保持向後相容性，在模式的初始部署之後 **新增的每個欄位必須是可選的或具有預設值**。

刪除一個欄位就像新增一個欄位，只是這回要考慮的是向前相容性。這意味著你只能刪除可選的欄位（必需欄位永遠不能刪除），而且你不能再次使用相同的標籤號碼（因為你可能仍然有資料寫在包含舊標籤號碼的地方，而該欄位必須被新程式碼忽略）。

資料型別和模式演變

如何改變欄位的資料型別？這也許是可能的——詳細資訊請查閱相關的文件——但是有一個風險，值將失去精度或被截斷。例如，假設你將一個 32 位的整數變成一個 64 位的整數。新程式碼可以輕鬆讀取舊程式碼寫入的資料，因為解析器可以用零填充任何缺失的位。但是，如果舊程式碼讀取由新程式碼寫入的資料，則舊程式碼仍使用 32 位變數來儲存該值。如果解碼的 64 位值不適合 32 位，則它將被截斷。

Protobuf 的一個奇怪的細節是，它沒有列表或陣列資料型別，而是有一個欄位的重複標記（`repeated`，這是除必需和可選之外的第三個選項）。如 [圖 4-4](#) 所示，重複欄位的編碼正如它所說的那樣：同一個欄位標記只是簡單地出現在記錄中。這具有很好的效果，可以將可選（單值）欄位更改為重複（多值）欄位。讀取舊資料的新程式碼會看到一個包含零個或一個元素的列表（取決於該欄位是否存在）。讀取新資料的舊程式碼只能看到列表的最後一個元素。

Thrift 有一個專用的列表資料型別，它使用列表元素的資料型別進行引數化。這不允許 Protocol Buffers 所做的從單值到多值的演變，但是它具有支援巢狀列表的優點。

Avro

Apache Avro [【20】](#) 是另一種二進位制編碼格式，與 Protocol Buffers 和 Thrift 有著有趣的不同。它是作為 Hadoop 的一個子專案在 2009 年開始的，因為 Thrift 不適合 Hadoop 的用例 [【21】](#)。

Avro 也使用模式來指定正在編碼的資料的結構。它有兩種模式語言：一種（Avro IDL）用於人工編輯，一種（基於 JSON）更易於機器讀取。

我們用 Avro IDL 編寫的示意模式可能如下所示：

```
record Person {
    string          userName;
    union { null, long } favoriteNumber = null;
    array<string>   interests;
```

```
}
```

等價的 JSON 表示：

```
{
  "type": "record",
  "name": "Person",
  "fields": [
    {"name": "userName", "type": "string"},
    {"name": "favoriteNumber", "type": ["null", "long"], "default": null},
    {"name": "interests", "type": {"type": "array", "items": "string"}}
  ]
}
```

首先，請注意模式中沒有標籤號碼。如果我們使用這個模式編碼我們的例子記錄（例 4-1），Avro 二進位制編碼只有 32 個位元組長，這是我們所見過的所有編碼中最緊湊的。編碼位元組序列的分解如 圖 4-5 所示。

如果你檢查位元組序列，你可以看到沒有什麼可以識別字段或其資料型別。編碼只是由連在一起的值組成。一個字串只是一個長度字首，後跟 UTF-8 位元組，但是在被包含的資料中沒有任何內容告訴你它是一個字串。它可以是一個整數，也可以是其他的整數。整數使用可變長度編碼（與 Thrift 的 CompactProtocol 相同）進行編碼。

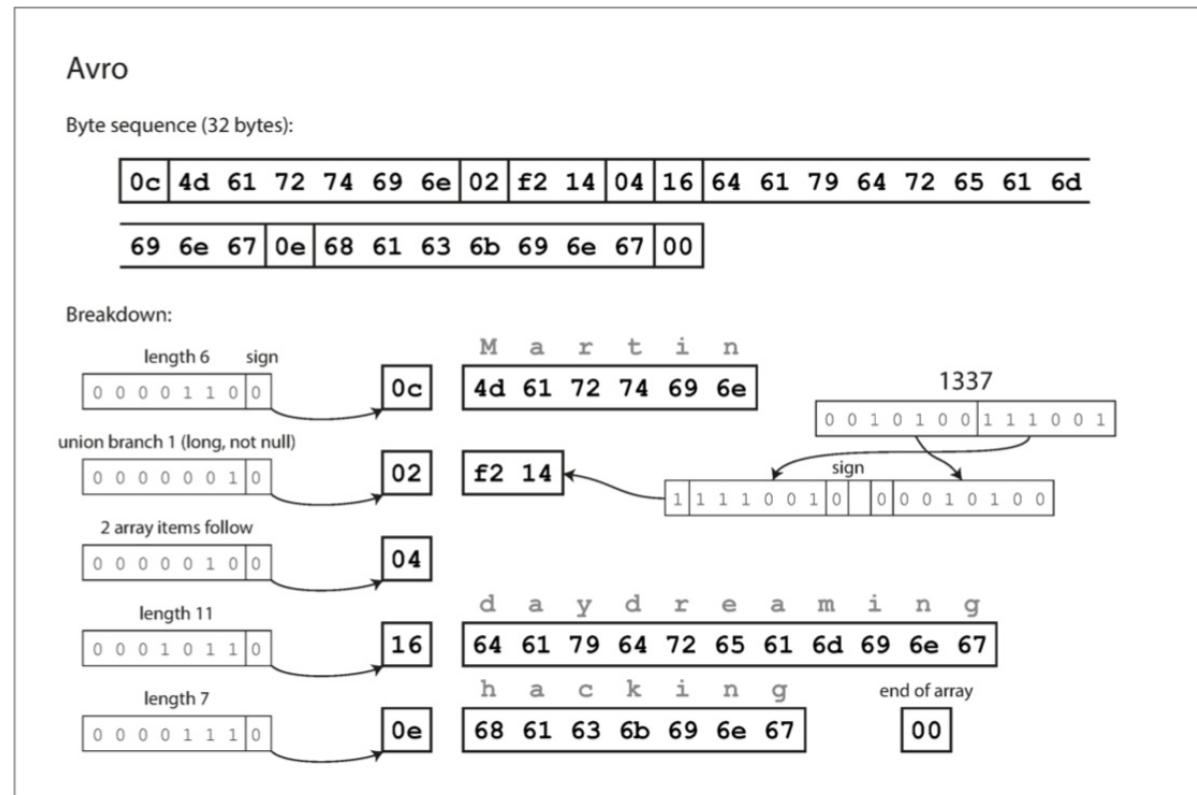


圖 4-5 使用 Avro 編碼的記錄

為了解析二進位制資料，你按照它們出現在模式中的順序遍歷這些欄位，並使用模式來告訴你每個欄位的資料型別。這意味著如果讀取資料的程式碼使用與寫入資料的程式碼完全相同的模式，才能正確解碼二進位制資料。Reader 和 Writer 之間的模式不匹配意味著錯誤地解碼資料。

那麼，Avro 如何支援模式演變呢？

Writer模式與Reader模式

有了 Avro，當應用程式想要編碼一些資料（將其寫入檔案或資料庫、透過網路傳送等）時，它使用它知道的任何版本的模式編碼資料，例如，模式可能被編譯到應用程式中。這被稱為 Writer 模式。

當一個應用程式想要解碼一些資料（從一個檔案或資料庫讀取資料、從網路接收資料等）時，它希望資料在某個模式中，這就是 Reader 模式。這是應用程式程式碼所依賴的模式，在應用程式的構建過程中，程式碼可能已經從該模式生成。

Avro 的關鍵思想是 Writer 模式和 Reader 模式不必是相同的 - 他們只需要相容。當資料解碼（讀取）時，Avro 庫透過並排檢視 Writer 模式和 Reader 模式並將資料從 Writer 模式轉換到 Reader 模式來解決差異。Avro 規範【20】確切地定義了這種解析的工作原理，如 圖 4-6 所示。

例如，如果 Writer 模式和 Reader 模式的欄位順序不同，這是沒有問題的，因為模式解析透過欄位名匹配欄位。如果讀取資料的程式碼遇到出現在 Writer 模式中但不在 Reader 模式中的欄位，則忽略它。如果讀取資料的程式碼需要某個欄位，但是 Writer 模式不包含該名稱的欄位，則使用在 Reader 模式中宣告的預設值填充。

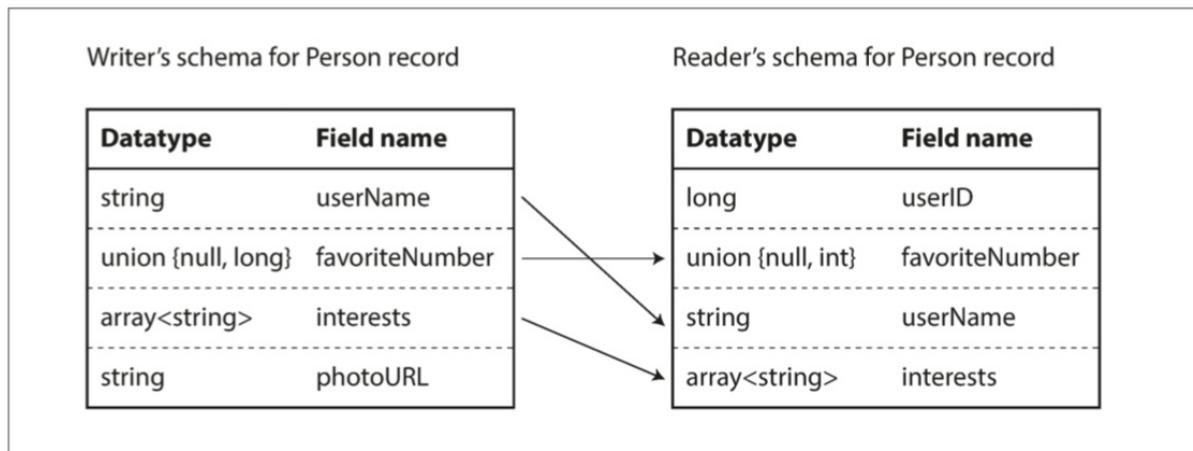


圖 4-6 一個 Avro Reader 解決讀寫模式的差異

模式演變規則

使用 Avro，向前相容性意味著你可以將新版本的模式作為 Writer，並將舊版本的模式作為 Reader。相反，向後相容意味著你可以有一個作為 Reader 的新版本模式和作為 Writer 的舊版本模式。

為了保持相容性，你只能新增或刪除具有預設值的欄位（我們的 Avro 模式中的欄位 `favoriteNumber` 的預設值為 `null`）。例如，假設你添加了一個有預設值的欄位，這個新的欄位將存在於新模式而不是舊模式中。當使用新模式的 Reader 讀取使用舊模式寫入的記錄時，將為缺少的欄位填充預設值。

如果你要新增一個沒有預設值的欄位，新的 Reader 將無法讀取舊 Writer 寫的資料，所以你會破壞向後相容性。如果你要刪除沒有預設值的欄位，舊的 Reader 將無法讀取新 Writer 寫入的資料，因此你會打破向前相容性。在一些程式語言中，`null` 是任何變數可以接受的預設值，但在 Avro 中並不是這樣：如果要允許一個欄位為 `null`，則必須使用聯合型別。例如，`union {null, long, string} field;` 表示 field 可以是數字或字串，也可以是 `null`。如果要將 `null` 作為預設值，則它必須是 union 的分支之一^{iv}。這樣的寫法比預設情況下就允許任何變數是 `null` 顯得更加冗長，但是透過明確什麼可以和什麼不可以是 `null`，有助於防止出錯【22】。

^{iv}. 確切地說，預設值必須是聯合的第一個分支的型別，儘管這是 Avro 的特定限制，而不是聯合型別的一般特徵。 ↪

因此，Avro 沒有像 Protocol Buffers 和 Thrift 那樣的 `optional` 和 `required` 標記（但它有聯合型別和預設值）。

只要 Avro 可以支援相應的型別轉換，就可以改變欄位的資料型別。更改欄位的名稱也是可能的，但有點棘手：Reader 模式可以包含欄位名稱的別名，所以它可以匹配舊 Writer 的模式欄位名稱與別名。這意味著更改欄位名稱是向後相容的，但不能向前相容。同樣，向聯合型別新增分支也是向後相容的，但不能向前相容。

但Writer模式到底是什麼？

到目前為止，我們一直跳過了一個重要的問題：對於一段特定的編碼資料，Reader 如何知道其 Writer 模式？我們不能只將整個模式包括在每個記錄中，因為模式可能比編碼的資料大得多，從而使二進位制編碼節省的所有空間都是徒勞的。

答案取決於 Avro 使用的上下文。舉幾個例子：

- 有很多記錄的大檔案

Avro 的一個常見用途 - 尤其是在 Hadoop 環境中 - 用於儲存包含數百萬條記錄的大檔案，所有記錄都使用相同的模式進行編碼（我們將在 [第十章](#) 討論這種情況）。在這種情況下，該檔案的作者可以在檔案的開頭只包含一次 Writer 模式。Avro 指定了一個檔案格式（物件容器檔案）來做到這一點。

- 支援獨立寫入的記錄的資料庫

在一個數據庫中，不同的記錄可能會在不同的時間點使用不同的 Writer 模式來寫入 - 你不能假定所有的記錄都有相同的模式。最簡單的解決方案是在每個編碼記錄的開始處包含一個版本號，並在資料庫中保留一個模式版本列表。Reader 可以獲取記錄，提取版本號，然後從資料庫中獲取該版本號的 Writer 模式。使用該 Writer 模式，它可以解碼記錄的其餘部分（例如 Espresso [【23】](#) 就是這樣工作的）。

- 透過網路連線傳送記錄

當兩個程序透過雙向網路連線進行通訊時，他們可以在連線設定上協商模式版本，然後在連線的生命週期中使用該模式。Avro RPC 協議（請參閱 [“服務中的資料流：REST 與 RPC”](#)）就是這樣工作的。

具有模式版本的資料庫在任何情況下都是非常有用的，因為它充當文件併為你提供了檢查模式相容性的機會 [【24】](#)。作為版本號，你可以使用一個簡單的遞增整數，或者你可以使用模式的雜湊。

動態生成的模式

與 Protocol Buffers 和 Thrift 相比，Avro 方法的一個優點是架構不包含任何標籤號碼。但為什麼這很重要？在模式中保留一些數字有什麼問題？

不同之處在於 Avro 對動態生成的模式更友善。例如，假如你有一個關係資料庫，你想要把它的內容轉儲到一個檔案中，並且你想使用二進位制格式來避免前面提到的文字格式（JSON，CSV，SQL）的問題。如果你使用 Avro，你可以很容易地從關係模式生成一個 Avro 模式（在我們之前看到的 JSON 表示中），並使用該模式對資料庫內容進行編碼，並將其全部轉儲到 Avro 物件容器檔案 [【25】](#) 中。你為每個資料庫表生成一個記錄模式，每個列成為該記錄中的一個欄位。資料庫中的列名稱對映到 Avro 中的欄位名稱。

現在，如果資料庫模式發生變化（例如，一個表中添加了一列，刪除了一列），則可以從更新的資料庫模式生成新的 Avro 模式，並在新的 Avro 模式中匯出資料。資料匯出過程不需要注意模式的改變 - 每次執行時都可以簡單地進行模式轉換。任何讀取新資料檔案的人都會看到記錄的欄位已經改變，但是由於欄位是透過名字來標識的，所以更新的 Writer 模式仍然可以與舊的 Reader 模式匹配。

相比之下，如果你為此使用 Thrift 或 Protocol Buffers，則欄位標籤可能必須手動分配：每次資料庫模式更改時，管理員都必須手動更新從資料庫列名到欄位標籤的對映（這可能會自動化，但模式生成器必須非常小心，不要分配以前使用的欄位標籤）。這種動態生成的模式根本不是 Thrift 或 Protocol Buffers 的設計目標，而是 Avro 的。

程式碼生成和動態型別的語言

Thrift 和 Protobuf 依賴於程式碼生成：在定義了模式之後，可以使用你選擇的程式語言生成實現此模式的程式碼。這在 Java、C++ 或 C# 等靜態型別語言中很有用，因為它允許將高效的記憶體中的資料結構用於解碼的資料，並且在編寫訪問資料結構的程式時允許在 IDE 中進行型別檢查和自動補全。

在動態型別程式語言（如 JavaScript、Ruby 或 Python）中，生成程式碼沒有太多意義，因為沒有編譯時型別檢查器來滿足。程式碼生成在這些語言中經常被忽視，因為它們避免了顯式的編譯步驟。而且，對於動態生成的模式（例如從資料庫表生成的 Avro 模式），程式碼生成對獲取資料是一個不必要的障礙。

Avro 為靜態型別程式語言提供了可選的程式碼生成功能，但是它也可以在不生成任何程式碼的情況下使用。如果你有一個物件容器檔案（它嵌入了 Writer 模式），你可以簡單地使用 Avro 庫開啟它，並以與檢視 JSON 檔案相同的方式檢視資料。該檔案是自描述的，因為它包含所有必要的元資料。

這個屬性特別適用於動態型別的資料處理語言如 Apache Pig 【26】。在 Pig 中，你可以開啟一些 Avro 檔案，開始分析它們，並編寫派生資料集以 Avro 格式輸出檔案，而無需考慮模式。

模式的優點

正如我們所看到的，Protocol Buffers、Thrift 和 Avro 都使用模式來描述二進位制編碼格式。他們的模式語言比 XML 模式或者 JSON 模式簡單得多，而後者支援更詳細的驗證規則（例如，“該欄位的字串值必須與該正則表示式匹配”或“該欄位的整數值必須在 0 和 100 之間”）。由於 Protocol Buffers，Thrift 和 Avro 實現起來更簡單，使用起來也更簡單，所以它們已經發展到支援相當廣泛的程式語言。

這些編碼所基於的想法絕不是新的。例如，它們與 ASN.1 有很多相似之處，它是 1984 年首次被標準化的模式定義語言【27】。它被用來定義各種網路協議，例如其二進位制編碼（DER）仍然被用於編碼 SSL 證書（X.509）【28】。ASN.1 支援使用標籤號碼的模式演進，類似於 Protocol Buffers 和 Thrift 【29】。然而，它也非常複雜，而且沒有好的配套文件，所以 ASN.1 可能不是新應用程式的好選擇。

許多資料系統也為其資料實現了某種專有的二進位制編碼。例如，大多數關係資料庫都有一個網路協議，你可以透過該協議向資料庫傳送查詢並獲取響應。這些協議通常特定於特定的資料庫，並且資料庫供應商提供將來自資料庫的網路協議的響應解碼為記憶體資料結構的驅動程式（例如使用 ODBC 或 JDBC API）。

所以，我們可以看到，儘管 JSON、XML 和 CSV 等文字資料格式非常普遍，但基於模式的二進位制編碼也是一個可行的選擇。他們有一些很好的屬性：

- 它們可以比各種“二進位制 JSON”變體更緊湊，因為它們可以省略編碼資料中的欄位名稱。
- 模式是一種有價值的文件形式，因為模式是解碼所必需的，所以可以確定它是最新的（而手動維護的文件可能很容易偏離現實）。
- 維護一個模式的資料庫允許你在部署任何內容之前檢查模式更改的向前和向後相容性。
- 對於靜態型別程式語言的使用者來說，從模式生成程式碼的能力是有用的，因為它可以在編譯時進行型別檢查。

總而言之，模式演化保持了與 JSON 資料庫提供的無模式 / 讀時模式相同的靈活性（請參閱“[文件模型中的模式靈活性](#)”），同時還可以更好地保證你的資料並提供更好的工具。

資料流的型別

在本章的開始部分，我們曾經說過，無論何時你想要將某些資料傳送到不共享記憶體的另一個程序，例如，只要你想透過網路傳送資料或將其寫入檔案，就需要將它編碼為一個位元組序列。然後我們討論了做這個的各種不同的編碼。

我們討論了向前和向後的相容性，這對於可演化性來說非常重要（透過允許你獨立升級系統的不同部分，而不必一次改變所有內容，可以輕鬆地進行更改）。相容性是編碼資料的一個程序和解碼它的另一個程序之間的一種關係。

這是一個相當抽象的概念 - 資料可以透過多種方式從一個流程流向另一個流程。誰編碼資料，誰解碼？在本章的其餘部分中，我們將探討資料如何在流程之間流動的一些最常見的方式：

- 透過資料庫（請參閱“[資料庫中的資料流](#)”）
- 透過服務呼叫（請參閱“[服務中的資料流：REST 與 RPC](#)”）
- 透過非同步訊息傳遞（請參閱“[訊息傳遞中的資料流](#)”）

資料庫中的資料流

在資料庫中，寫入資料庫的過程對資料進行編碼，從資料庫讀取的過程對資料進行解碼。可能只有一個程序訪問資料庫，在這種情況下，讀者只是相同程序的後續版本 - 在這種情況下，你可以考慮將資料庫中的內容儲存為向未來的自我傳送訊息。

向後相容性顯然是必要的。否則你未來的自己將無法解碼你以前寫的東西。

一般來說，幾個不同的程序同時訪問資料庫是很常見的。這些程序可能是幾個不同的應用程式或服務，或者它們可能只是幾個相同服務的例項（為了可伸縮性或容錯性而並行執行）。無論哪種方式，在應用程式發生變化的環境中，訪問資料庫的某些程序可能會執行較新的程式碼，有些程序可能會執行較舊的程式碼，例如，因為新版本當前正在部署滾動升級，所以有些例項已經更新，而其他例項尚未更新。

這意味著資料庫中的一個值可能會被更新版本的程式碼寫入，然後被仍舊執行的舊版本的程式碼讀取。因此，資料庫也經常需要向前相容。

但是，還有一個額外的障礙。假設你將一個欄位新增到記錄模式，並且較新的程式碼將該新欄位的值寫入資料庫。隨後，舊版本的程式碼（尚不知道新欄位）將讀取記錄，更新記錄並將其寫回。在這種情況下，理想的行為通常是舊程式碼保持新的欄位不變，即使它不能被解釋。

前面討論的編碼格式支援未知欄位的儲存，但是有時候需要在應用程式層面保持謹慎，如圖 4-7 所示。例如，如果將資料庫值解碼為應用程式中的模型物件，稍後重新編碼這些模型物件，那麼未知欄位可能會在該翻譯過程中丟失。解決這個問題不是一個難題，你只需要意識到它。

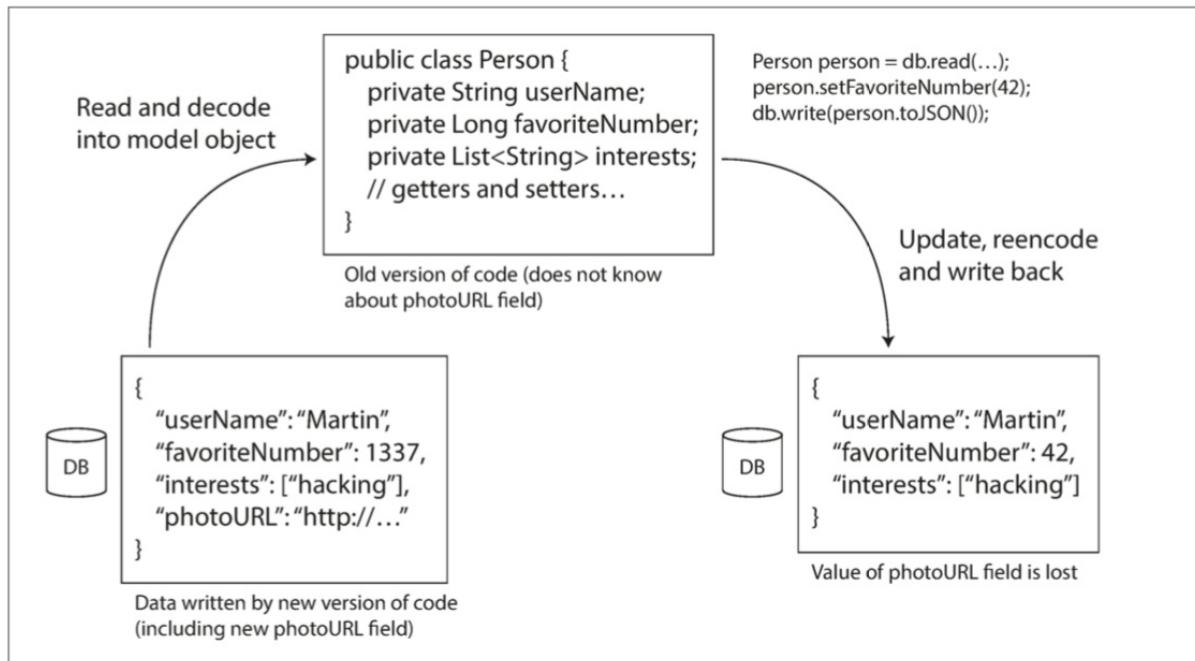


圖 4-7 當較舊版本的應用程式更新以前由較新版本的應用程式編寫的資料時，如果不小心，資料可能會丟失。

在不同的時間寫入不同的值

資料庫通常允許任何時候更新任何值。這意味著在一個單一的資料庫中，可能有一些值是五毫秒前寫的，而一些值是五年前寫的。

在部署應用程式的新版本時，也許用不了幾分鐘就可以將所有的舊版本替換為新版本（至少伺服器端應用程式是這樣的）。但資料庫內容並非如此：對於五年前的資料來說，除非對其進行顯式重寫，否則它仍然會以原始編碼形式存在。這種現象有時被概括為：資料的生命週期超出程式碼的生命週期。

將資料重寫（遷移）到一個新的模式當然是可能的，但是在一個大資料集上執行是一個昂貴的事情，所以大多數資料庫如果可能的話就避免它。大多數關係資料庫都允許簡單的模式更改，例如新增一個預設值為空的新列，而不重寫現有資料^v。讀取舊行時，對於磁碟上的編碼資料缺少的任何列，資料庫將填充空值。LinkedIn 的文件資料庫 Espresso 使用

Avro 儲存，允許它使用 Avro 的模式演變規則【23】。

因此，模式演變允許整個資料庫看起來好像是用單個模式編碼的，即使底層儲存可能包含用各種歷史版本的模式編碼的記錄。

除了 MySQL，即使並非真的必要，它也經常會重寫整個表，正如“[文件模型中的模式靈活性](#)”中所提到的。 ↵

歸檔儲存

也許你不時為資料庫建立一個快照，例如備份或載入到資料倉庫（請參閱[“資料倉庫”](#)）。在這種情況下，即使源資料庫中的原始編碼包含來自不同時代的模式版本的混合，資料轉儲通常也將使用最新模式進行編碼。既然你不管怎樣都要複製資料，那麼你可以對這個資料複製進行一致的編碼。

由於資料轉儲是一次寫入的，而且以後是不可變的，所以 Avro 物件容器檔案等格式非常適合。這也是一個很好的機會，可以將資料編碼為面向分析的列式格式，例如 Parquet（請參閱[“列壓縮”](#)）。

在[第十章](#)中，我們將詳細討論使用檔案儲存中的資料。

服務中的資料流：REST與RPC

當你需要透過網路進行程序間的通訊時，安排該通訊的方式有幾種。最常見的安排是有兩個角色：客戶端和伺服器。伺服器透過網路公開 API，並且客戶端可以連線到伺服器以向該 API 發出請求。伺服器公開的 API 被稱為服務。

Web 以這種方式工作：客戶（Web 瀏覽器）向 Web 伺服器發出請求，透過 GET 請求下載 HTML、CSS、JavaScript、影象等，並透過 POST 請求提交資料到伺服器。API 包含一組標準的協議和資料格式（HTTP、URL、SSL/TLS、HTML 等）。由於網路瀏覽器、網路伺服器和網站作者大多同意這些標準，你可以使用任何網路瀏覽器訪問任何網站（至少在理論上！）。

Web 瀏覽器不是唯一的客戶端型別。例如，在移動裝置或桌面計算機上執行的本地應用程式也可以向伺服器發出網路請求，並且在 Web 瀏覽器內執行的客戶端 JavaScript 應用程式可以使用 XMLHttpRequest 成為 HTTP 客戶端（該技術被稱為 Ajax 【30】）。在這種情況下，伺服器的響應通常不是用於顯示給人的 HTML，而是便於客戶端應用程式程式碼進一步處理的編碼資料（如 JSON）。儘管 HTTP 可能被用作傳輸協議，但頂層實現的 API 是特定於應用程式的，客戶端和伺服器需要就該 API 的細節達成一致。

此外，伺服器本身可以是另一個服務的客戶端（例如，典型的 Web 應用伺服器充當資料庫的客戶端）。這種方法通常用於將大型應用程式按照功能區域分解為較小的服務，這樣當一個服務需要來自另一個服務的某些功能或資料時，就會向另一個服務發出請求。這種構建應用程式的方式傳統上被稱為 **面向服務的體系結構（service-oriented architecture, SOA）**，最近被改進和更名為 **微服務架構【31,32】**。

在某些方面，服務類似於資料庫：它們通常允許客戶端提交和查詢資料。但是，雖然資料庫允許使用我們在[第二章](#)中討論的查詢語言進行任意查詢，但是服務公開了一個特定於應用程式的 API，它只允許由服務的業務邏輯（應用程式程式碼）預定的輸入和輸出【33】。這種限制提供了一定程度的封裝：服務能夠對客戶可以做什麼和不可以做什麼施加細粒度的限制。

面向服務 / 微服務架構的一個關鍵設計目標是透過使服務獨立部署和演化來使應用程式更易於更改和維護。例如，每個服務應該由一個團隊擁有，並且該團隊應該能夠經常釋出新版本的服務，而不必與其他團隊協調。換句話說，我們應該期望伺服器和客戶端的舊版本和新版本同時執行，因此伺服器和客戶端使用的資料編碼必須在不同版本的服務 API 之間相容——這正是我們在本章所一直在談論的。

Web服務

當服務使用 HTTP 作為底層通訊協議時，可稱之為 **Web 服務**。這可能是一個小錯誤，因為 Web 服務不僅在 Web 上使用，而且在幾個不同的環境中使用。例如：

1. 執行在使用者裝置上的客戶端應用程式（例如，移動裝置上的本地應用程式，或使用 Ajax 的 JavaScript web 應用程式）透過 HTTP 向服務發出請求。這些請求通常透過公共網際網路進行。

2. 一種服務向同一組織擁有的另一項服務提出請求，這些服務通常位於同一資料中心內，作為面向服務 / 微服務架構的一部分。（支援這種用例的軟體有時被稱為 **中介軟體（middleware）**）
3. 一種服務透過網際網路向不同組織所擁有的服務提出請求。這用於不同組織後端系統之間的資料交換。此類別包括由線上服務（如信用卡處理系統）提供的公共 API，或用於共享訪問使用者資料的 OAuth。

有兩種流行的 Web 服務方法：REST 和 SOAP。他們在哲學方面幾乎是截然相反的，往往也是各自支持者之間的激烈辯論的主題^{vi}。

^{vi}. 即使在每個陣營內也有很多爭論。例如，HATEOAS（超媒體作為應用程式狀態的引擎）就經常引發討論【35】。 ↵

REST 不是一個協議，而是一個基於 HTTP 原則的設計哲學【34,35】。它強調簡單的資料格式，使用 URL 來標識資源，並使用 HTTP 功能進行快取控制，身份驗證和內容型別協商。與 SOAP 相比，REST 已經越來越受歡迎，至少在跨組織服務整合的背景下【36】，並經常與微服務相關【31】。根據 REST 原則設計的 API 稱為 RESTful。

相比之下，SOAP 是用於製作網路 API 請求的基於 XML 的協議^{vii}。雖然它最常用於 HTTP，但其目的是獨立於 HTTP，並避免使用大多數 HTTP 功能。相反，它帶有龐大而複雜的多種相關標準（Web 服務框架，稱為 ws-*），它們增加了各種功能【37】。

^{vii}. 儘管首字母縮寫詞相似，SOAP 並不是 SOA 的要求。SOAP 是一種特殊的技術，而 SOA 是構建系統的一般方法。 ↵

SOAP Web 服務的 API 使用稱為 Web 服務描述語言（WSDL）的基於 XML 的語言來描述。WSDL 支援程式碼生成，客戶端可以使用本地類和方法呼叫（編碼為 XML 訊息並由框架再次解碼）訪問遠端服務。這在靜態型別程式語言中非常有用，但在動態型別程式語言中很少（請參閱 “[程式碼生成和動態型別的語言](#)”）。

由於 WSDL 的設計不是人類可讀的，而且由於 SOAP 訊息通常因為過於複雜而無法手動構建，所以 SOAP 的使用者在很大程度上依賴於工具支援，程式碼生成和 IDE【38】。對於 SOAP 供應商不支援的程式語言的使用者來說，與 SOAP 服務的整合是困難的。

儘管 SOAP 及其各種擴充套件表面上是標準化的，但是不同廠商的實現之間的互操作性往往會造成問題【39】。由於所有這些原因，儘管許多大型企業仍然使用 SOAP，但在大多數小公司中已經不再受到青睞。

REST 風格的 API 傾向於更簡單的方法，通常涉及較少的程式碼生成和自動化工具。定義格式（如 OpenAPI，也稱為 Swagger【40】）可用於描述 RESTful API 並生成文件。

遠端過程呼叫（RPC）的問題

Web 服務僅僅是透過網路進行 API 請求的一系列技術的最新版本，其中許多技術受到了大量的炒作，但是存在嚴重的問題。Enterprise JavaBeans（EJB）和 Java 的 **遠端方法呼叫（RMI）** 僅限於 Java。**分散式元件物件模型（DCOM）** 僅限於 Microsoft 平臺。**公共物件請求代理體系結構（CORBA）** 過於複雜，不提供前向或後向相容性【41】。

所有這些都是基於 **遠端過程呼叫（RPC）** 的思想，該過程呼叫自 20 世紀 70 年代以來一直存在【42】。RPC 模型試圖向遠端網路服務發出請求，看起來與在同一程序中呼叫程式語言中的函式或方法相同（這種抽象稱為位置透明）。儘管 RPC 起初看起來很方便，但這種方法根本上是有缺陷的【43,44】。網路請求與本地函式呼叫非常不同：

- 本地函式呼叫是可預測的，並且成功或失敗僅取決於受你控制的引數。網路請求是不可預測的：請求或響應可能由於網路問題會丟失，或者遠端計算機可能很慢或不可用，這些問題完全不在你的控制範圍之內。網路問題很常見，因此必須有所準備，例如重試失敗的請求。
- 本地函式呼叫要麼返回結果，要麼丟擲異常，或者永遠不返回（因為進入無限迴圈或程序崩潰）。網路請求有另一個可能的結果：由於超時，它返回時可能沒有結果。在這種情況下，你根本不知道發生了什麼：如果你沒有得到來自遠端服務的響應，你無法知道請求是否透過（我們將在 [第八章](#) 更詳細地討論這個問題）。
- 如果你重試失敗的網路請求，可能會發生請求實際上已經完成，只是響應丟失的情況。在這種情況下，重試將導致該操作被執行多次，除非你在協議中建立資料去重機制（**冪等性**，即 idempotence）。本地函式呼叫時沒有這樣的問題。（在 [第十一章](#) 更詳細地討論冪等性）

- 每次呼叫本地函式時，通常需要大致相同的時間來執行。網路請求比函式呼叫要慢得多，而且其延遲也是非常可變的：好的時候它可能會在不到一毫秒的時間內完成，但是當網路擁塞或者遠端服務超載時，可能需要幾秒鐘的時間才能完成相同的操作。
- 呼叫本地函式時，可以高效地將引用（指標）傳遞給本地記憶體中的物件。當你發出一個網路請求時，所有這些引數都需要被編碼成可以透過網路傳送的一系列位元組。如果引數是像數字或字串這樣的基本型別倒是沒關係，但是對於較大的物件很快就會出現問題。
- 客戶端和服務可以用不同的程式語言實現，所以 RPC 框架必須將資料型別從一種語言翻譯成另一種語言。這可能會變得很醜陋，因為不是所有的語言都具有相同的型別——例如回想一下 JavaScript 的數字大於 2^{53} 的問題（請參閱 “[JSON、XML 和二進位制變體](#)”）。用單一語言編寫的單個程序中不存在此問題。

所有這些因素意味著嘗試使遠端服務看起來像程式語言中的本地物件一樣毫無意義，因為這是一個根本不同的事情。REST 的部分吸引力在於，它並不試圖隱藏它是一個網路協議的事實（儘管這似乎並沒有阻止人們在 REST 之上構建 RPC 庫）。

RPC的當前方向

儘管有這樣那樣的問題，RPC 不會消失。在本章提到的所有編碼的基礎上構建了各種 RPC 框架：例如，Thrift 和 Avro 帶有 RPC 支援，gRPC 是使用 Protocol Buffers 的 RPC 實現，Finagle 也使用 Thrift，Rest.li 使用 JSON over HTTP。

這種新一代的 RPC 框架更加明確的是，遠端請求與本地函式呼叫不同。例如，Finagle 和 Rest.li 使用 futures (promises) 來封裝可能失敗的非同步操作。futures 還可以簡化需要並行發出多項服務並將其結果合併的情況【45】。gRPC 支援流，其中一個呼叫不僅包括一個請求和一個響應，還可以是隨時間的一系列請求和響應【46】。

其中一些框架還提供服務發現，即允許客戶端找出在哪個 IP 地址和埠號上可以找到特定的服務。我們將在 “[請求路由](#)” 中回到這個主題。

使用二進位制編碼格式的自定義 RPC 協議可以實現比通用的 JSON over REST 更好的效能。但是，RESTful API 還有其他一些顯著的優點：方便實驗和除錯（只需使用 Web 瀏覽器或命令列工具 curl，無需任何程式碼生成或軟體安裝即可向其請求），能被所有主流的程式語言和平臺所支援，還有大量可用的工具（伺服器、快取、負載平衡器、代理、防火牆、監控、除錯工具、測試工具等）的生態系統。

由於這些原因，REST 似乎是公共 API 的主要風格。RPC 框架的主要重點在於同一組織擁有的服務之間的請求，通常在同一資料中心內。

資料編碼與RPC的演化

對於可演化性，重要的是可以獨立更改和部署 RPC 客戶端和伺服器。與透過資料庫流動的資料相比（如上一節所述），我們可以在透過服務進行資料流的情況下做一個簡化的假設：假定所有的伺服器都會先更新，其次是所有的客戶端。因此，你只需要在請求上具有向後相容性，並且對響應具有前向相容性。

RPC 方案的前後向相容性屬性從它使用的編碼方式中繼承：

- Thrift、gRPC (Protobuf) 和 Avro RPC 可以根據相應編碼格式的相容性規則進行演變。
- 在 SOAP 中，請求和響應是使用 XML 模式指定的。這些可以演變，但有一些微妙的陷阱【47】。
- RESTful API 通常使用 JSON (沒有正式指定的模式) 用於響應，以及用於請求的 JSON 或 URI 編碼 / 表單編碼的請求引數。新增可選的請求引數並向響應物件新增新的欄位通常被認為是保持相容性的改變。

由於 RPC 經常被用於跨越組織邊界的通訊，所以服務的相容性變得更加困難，因此服務的提供者經常無法控制其客戶，也不能強迫他們升級。因此，需要長期保持相容性，也許是無限期的。如果需要進行相容性更改，則服務提供商通常會並排維護多個版本的服務 API。

關於 API 版本化應該如何工作（即，客戶端如何指示它想要使用哪個版本的 API）沒有一致意見【48】。對於 RESTful API，常用的方法是在 URL 或 HTTP Accept 頭中使用版本號。對於使用 API 金鑰來標識特定客戶端的服務，另一種選擇是將客戶端請求的 API 版本儲存在伺服器上，並允許透過單獨的管理介面更新該版本選項【49】。

訊息傳遞中的資料流

我們一直在研究從一個過程到另一個過程的編碼資料流的不同方式。到目前為止，我們已經討論了 REST 和 RPC（其中一個程序透過網路向另一個程序傳送請求並期望儘可能快的響應）以及資料庫（一個程序寫入編碼資料，另一個程序在將來再次讀取）。

在最後一節中，我們將簡要介紹一下 RPC 和資料庫之間的非同步訊息傳遞系統。它們與 RPC 類似，因為客戶端的請求（通常稱為訊息）以低延遲傳送到另一個程序。它們與資料庫類似，不是透過直接的網路連線傳送訊息，而是透過稱為訊息代理（也稱為訊息佇列或面向訊息的中介軟體）的中介來臨時儲存訊息。

與直接 RPC 相比，使用訊息代理有幾個優點：

- 如果收件人不可用或過載，可以充當緩衝區，從而提高系統的可靠性。
- 它可以自動將訊息重新發送到已經崩潰的程序，從而防止訊息丟失。
- 避免發件人需要知道收件人的 IP 地址和埠號（這在虛擬機器經常出入的雲部署中特別有用）。
- 它允許將一條訊息傳送給多個收件人。
- 將發件人與收件人邏輯分離（發件人只是釋出郵件，不關心使用者）。

然而，與 RPC 相比，差異在於訊息傳遞通訊通常是單向的：傳送者通常不期望收到其訊息的回覆。一個程序可能傳送一個響應，但這通常是在一個單獨的通道上完成的。這種通訊模式是非同步的：傳送者不會等待訊息被傳遞，而只是傳送它，然後忘記它。

訊息代理

過去，訊息代理（**Message Broker**）主要是 TIBCO、IBM WebSphere 和 webMethods 等公司的商業軟體的秀場。最近像 RabbitMQ、ActiveMQ、HornetQ、NATS 和 Apache Kafka 這樣的開源實現已經流行起來。我們將在 [第十一章](#) 中對它們進行更詳細的比較。

詳細的交付語義因實現和配置而異，但通常情況下，訊息代理的使用方式如下：一個程序將訊息傳送到指定的佇列或主題，代理確保將訊息傳遞給那個佇列或主題的一個或多個消費者或訂閱者。在同一主題上可以有許多生產者和許多消費者。

一個主題只提供單向資料流。但是，消費者本身可能會將訊息釋出到另一個主題上（因此，可以將它們連結在一起，就像我們將在 [第十一章](#) 中看到的那樣），或者傳送給原始訊息的傳送者使用的回覆佇列（允許請求 / 嘴應資料流，類似於 RPC）。

訊息代理通常不會執行任何特定的資料模型——訊息只是包含一些元資料的位元組序列，因此你可以使用任何編碼格式。如果編碼是向後和向前相容的，你可以靈活地對釋出者和消費者的編碼進行獨立的修改，並以任意順序進行部署。

如果消費者重新發布訊息到另一個主題，則可能需要小心保留未知欄位，以防止前面在資料庫環境中描述的問題（[圖 4-7](#)）。

分散式的Actor框架

Actor 模型是單個程序中併發的程式設計模型。邏輯被封裝在 actor 中，而不是直接處理執行緒（以及競爭條件、鎖定和死鎖的相關問題）。每個 actor 通常代表一個客戶或實體，它可能有一些本地狀態（不與其他任何角色共享），它透過傳送和接收非同步訊息與其他角色通訊。不保證訊息傳送：在某些錯誤情況下，訊息將丟失。由於每個角色一次只能處理一條訊息，因此不需要擔心執行緒，每個角色可以由框架獨立排程。

在分散式 Actor 框架中，此程式設計模型用於跨多個節點伸縮應用程式。不管傳送方和接收方是在同一個節點上還是在不同的節點上，都使用相同的訊息傳遞機制。如果它們在不同的節點上，則該訊息被透明地編碼成位元組序列，透過網路傳送，並在另一側解碼。

位置透明在 actor 模型中比在 RPC 中效果更好，因為 actor 模型已經假定訊息可能會丟失，即使在單個程序中也是如此。儘管網路上的延遲可能比同一個程序中的延遲更高，但是在使用 actor 模型時，本地和遠端通訊之間的基本不匹配是較少的。

分散式的 Actor 框架實質上是將訊息代理和 actor 程式設計模型整合到一個框架中。但是，如果要執行基於 actor 的應用程式的滾動升級，則仍然需要擔心向前和向後相容性問題，因為訊息可能會從執行新版本的節點發送到執行舊版本的節點，反之亦然。

三個流行的分散式 actor 框架處理訊息編碼如下：

- 預設情況下，Akka 使用 Java 的內建序列化，不提供前向或後向相容性。但是，你可以用類似 Prototol Buffers 的東西替代它，從而獲得滾動升級的能力【50】。
- Orleans 預設使用不支援滾動升級的自定義資料編碼格式；要部署新版本的應用程式，你需要設定一個新的叢集，將流量從舊叢集遷移到新叢集，然後關閉舊叢集【51,52】。像 Akka 一樣，可以使用自定義序列化外掛。
- 在 Erlang OTP 中，對記錄模式進行更改是非常困難的（儘管系統具有許多為高可用性設計的功能）。滾動升級是可能的，但需要仔細計劃【53】。一個新的實驗性的 `maps` 資料型別（2014 年在 Erlang R17 中引入的類似於 JSON 的結構）可能使得這個資料型別在未來更容易【54】。

本章小結

在本章中，我們研究了將資料結構轉換為網路中的位元組或磁碟上的位元組的幾種方法。我們看到了這些編碼的細節不僅影響其效率，更重要的是也影響了應用程式的體系結構和部署它們的選項。

特別是，許多服務需要支援滾動升級，其中新版本的服務逐步部署到少數節點，而不是同時部署到所有節點。滾動升級允許在不停機的情況下發布新版本的服務（從而鼓勵在罕見的大型版本上頻繁釋出小型版本），並使部署風險降低（允許在影響大量使用者之前檢測並回滾有故障的版本）。這些屬性對於可演化性，以及對應用程式進行更改的容易性都是非常有利的。

在滾動升級期間，或出於各種其他原因，我們必須假設不同的節點正在執行我們的應用程式碼的不同版本。因此，在系統周圍流動的所有資料都是以提供向後相容性（新程式碼可以讀取舊資料）和向前相容性（舊程式碼可以讀取新資料）的方式進行編碼是重要的。

我們討論了幾種資料編碼格式及其相容性屬性：

- 程式語言特定的編碼僅限於單一程式語言，並且往往無法提供前向和後向相容性。
- JSON、XML 和 CSV 等文字格式非常普遍，其相容性取決於你如何使用它們。他們有可選的模式語言，這有時是有用的，有時是一個障礙。這些格式對於資料型別有些模糊，所以你必須小心數字和二進位制字串。
- 像 Thrift、Protocol Buffers 和 Avro 這樣的二進位制模式驅動格式允許使用清晰定義的前向和後向相容性語義進行緊湊，高效的編碼。這些模式可以用於靜態型別語言的文件和程式碼生成。但是，他們有一個缺點，就是在資料可讀之前需要對資料進行解碼。

我們還討論了資料流的幾種模式，說明了資料編碼重要性的不同場景：

- 資料庫，寫入資料庫的程序對資料進行編碼，並從資料庫讀取程序對其進行解碼
- RPC 和 REST API，客戶端對請求進行編碼，伺服器對請求進行解碼並對響應進行編碼，客戶端最終對響應進行解碼
- 非同步訊息傳遞（使用訊息代理或參與者），其中節點之間透過傳送訊息進行通訊，訊息由傳送者編碼並由接收者解碼

我們可以小心地得出這樣的結論：前向相容性和滾動升級在某種程度上是可以實現的。願你的應用程式的演變迅速、敏捷部署。

參考文獻

1. “Java Object Serialization Specification,” docs.oracle.com, 2010.
2. “Ruby 2.2.0 API Documentation,” ruby-doc.org, Dec 2014.
3. “The Python 3.4.3 Standard Library Reference Manual,” docs.python.org, February 2015.
4. “EsotericSoftware/kryo,” github.com, October 2014.

5. “[CWE-502: Deserialization of Untrusted Data](#),” Common Weakness Enumeration, cwe.mitre.org, July 30, 2014.
6. Steve Breen: “[What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability](#),” foxglovesecurity.com, November 6, 2015.
7. Patrick McKenzie: “[What the Rails Security Issue Means for Your Startup](#),” kalzumeus.com, January 31, 2013.
8. Eishay Smith: “[jvm-serializers wiki](#),” github.com, November 2014.
9. “[XML Is a Poor Copy of S-Expressions](#),” c2.com wiki.
10. Matt Harris: “[Snowflake: An Update and Some Very Important Information](#),” email to *Twitter Development Talk* mailing list, October 19, 2010.
11. Shudi (Sandy) Gao, C. M. Sperberg-McQueen, and Henry S. Thompson: “[XML Schema 1.1](#),” W3C Recommendation, May 2001.
12. Francis Galiegue, Kris Zyp, and Gary Court: “[JSON Schema](#),” IETF Internet-Draft, February 2013.
13. Yakov Shafranovich: “[RFC 4180: Common Format and MIME Type for Comma-Separated Values \(CSV\) Files](#),” October 2005.
14. “[MessagePack Specification](#),” msgpack.org.
15. Mark Slee, Aditya Agarwal, and Marc Kwiatkowski: “[Thrift: Scalable Cross-Language Services Implementation](#),” Facebook technical report, April 2007.
16. “[Protocol Buffers Developer Guide](#),” Google, Inc., developers.google.com.
17. Igor Anishchenko: “[Thrift vs Protocol Buffers vs Avro - Biased Comparison](#),” slideshare.net, September 17, 2012.
18. “[A Matrix of the Features Each Individual Language Library Supports](#),” wiki.apache.org.
19. Martin Kleppmann: “[Schema Evolution in Avro, Protocol Buffers and Thrift](#),” martin.kleppmann.com, December 5, 2012.
20. “[Apache Avro 1.7.7 Documentation](#),” avro.apache.org, July 2014.
21. Doug Cutting, Chad Walters, Jim Kellerman, et al.: “[PROPOSAL] New Subproject: Avro,” email thread on *hadoop-general* mailing list, mail-archives.apache.org, April 2009.
22. Tony Hoare: “[Null References: The Billion Dollar Mistake](#),” at *QCon London*, March 2009.
23. Aditya Auradkar and Tom Quiggle: “[Introducing Espresso—LinkedIn’s Hot New Distributed Document Store](#),” engineering.linkedin.com, January 21, 2015.
24. Jay Kreps: “[Putting Apache Kafka to Use: A Practical Guide to Building a Stream Data Platform \(Part 2\)](#),” blog.confluent.io, February 25, 2015.
25. Gwen Shapira: “[The Problem of Managing Schemas](#),” radar.oreilly.com, November 4, 2014.
26. “[Apache Pig 0.14.0 Documentation](#),” pig.apache.org, November 2014.
27. John Larmouth: *ASN.1Complete*. Morgan Kaufmann, 1999. ISBN: 978-0-122-33435-1
28. Russell Housley, Warwick Ford, Tim Polk, and David Solo: “[RFC 2459: Internet X.509 Public Key Infrastructure: Certificate and CRL Profile](#),” IETF Network Working Group, Standards Track, January 1999.
29. Lev Walkin: “[Question: Extensibility and Dropping Fields](#),” lionet.info, September 21, 2010.
30. Jesse James Garrett: “[Ajax: A New Approach to Web Applications](#),” adaptivepath.com, February 18, 2005.
31. Sam Newman: *Building Microservices*. O’Reilly Media, 2015. ISBN: 978-1-491-95035-7
32. Chris Richardson: “[Microservices: Decomposing Applications for Deployability and Scalability](#),” infoq.com, May 25, 2014.
33. Pat Helland: “[Data on the Outside Versus Data on the Inside](#),” at *2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2005.
34. Roy Thomas Fielding: “[Architectural Styles and the Design of Network-Based Software Architectures](#),” PhD Thesis, University of California, Irvine, 2000.
35. Roy Thomas Fielding: “[REST APIs Must Be Hypertext-Driven](#),” roy.gbiv.com, October 20 2008.
36. “[REST in Peace, SOAP](#),” royal.pingdom.com, October 15, 2010.
37. “[Web Services Standards as of Q1 2007](#),” infoq.com, February 2007.
38. Pete Lacey: “[The S Stands for Simple](#),” harmful.cat-v.org, November 15, 2006.
39. Stefan Tilkov: “[Interview: Pete Lacey Criticizes Web Services](#),” infoq.com, December 12, 2006.
40. “[OpenAPI Specification \(fka Swagger RESTful API Documentation Specification\) Version 2.0](#),” swagger.io, September 8, 2014.
41. Michi Henning: “[The Rise and Fall of CORBA](#),” *ACM Queue*, volume 4, number 5, pages 28–34, June 2006.

[doi:10.1145/1142031.1142044](https://doi.org/10.1145/1142031.1142044)

42. Andrew D. Birrell and Bruce Jay Nelson: “[Implementing Remote Procedure Calls](#),” *ACM Transactions on Computer Systems* (TOCS), volume 2, number 1, pages 39–59, February 1984. [doi:10.1145/2080.357392](https://doi.org/10.1145/2080.357392)
43. Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall: “[A Note on Distributed Computing](#),” Sun Microsystems Laboratories, Inc., Technical Report TR-94-29, November 1994.
44. Steve Vinoski: “[Convenience over Correctness](#),” *IEEE Internet Computing*, volume 12, number 4, pages 89–92, July 2008. [doi:10.1109/MIC.2008.75](https://doi.org/10.1109/MIC.2008.75)
45. Marius Eriksen: “[Your Server as a Function](#),” at *7th Workshop on Programming Languages and Operating Systems* (PLOS), November 2013. [doi:10.1145/2525528.2525538](https://doi.org/10.1145/2525528.2525538)
46. “[grpc-common Documentation](#),” Google, Inc., *github.com*, February 2015.
47. Aditya Narayan and Irina Singh: “[Designing and Versioning Compatible Web Services](#),” *ibm.com*, March 28, 2007.
48. Troy Hunt: “[Your API Versioning Is Wrong, Which Is Why I Decided to Do It 3 Different Wrong Ways](#),” *troyhunt.com*, February 10, 2014.
49. “[API Upgrades](#),” Stripe, Inc., April 2015.
50. Jonas Bonér: “[Upgrade in an Akka Cluster](#),” email to *akka-user* mailing list, *grokbase.com*, August 28, 2013.
51. Philip A. Bernstein, Sergey Bykov, Alan Geller, et al.: “[Orleans: Distributed Virtual Actors for Programmability and Scalability](#),” Microsoft Research Technical Report MSR-TR-2014-41, March 2014.
52. “[Microsoft Project Orleans Documentation](#),” Microsoft Research, *dotnet.github.io*, 2015.
53. David Mercer, Sean Hinde, Yinsuo Chen, and Richard A O’Keefe: “[beginner: Updating Data Structures](#),” email thread on *erlang-questions* mailing list, *erlang.com*, October 29, 2007.
54. Fred Hebert: “[Postscript: Maps](#),” *learnyousomeerlang.com*, April 9, 2014.

上一章	目錄	下一章
第三章：儲存與檢索	設計資料密集型應用	第二部分：分散式資料

第二部分：分散式資料

一個成功的技術，現實的優先順序必須高於公關，你可以糊弄別人，但糊弄不了自然規律。

—— 羅傑斯委員會報告（1986）

在本書的 [第一部分](#) 中，我們討論了資料系統的各個方面，但僅限於資料儲存在單臺機器上的情況。現在我們到了 [第二部分](#)，進入更高的層次，並提出一個問題：如果 **多臺機器** 參與資料的儲存和檢索，會發生什麼？

你可能會出於各種各樣的原因，希望將資料庫分佈到多臺機器上：

- 可伸縮性

如果你的資料量、讀取負載、寫入負載超出單臺機器的處理能力，可以將負載分散到多臺計算機上。

- 容錯 / 高可用性

如果你的應用需要在單臺機器（或多臺機器，網路或整個資料中心）出現故障的情況下仍然能繼續工作，則可使用多臺機器，以提供冗餘。一臺故障時，另一臺可以接管。

- 延遲

如果在世界各地都有使用者，你也許會考慮在全球範圍部署多個伺服器，從而每個使用者可以從地理上最近的資料中心獲取服務，避免了等待網路資料包穿越半個世界。

伸縮至更高的載荷

如果你需要的只是伸縮至更高的 **載荷**（load），最簡單的方法就是購買更強大的機器（有時稱為 **垂直伸縮**，即 vertical scaling，或 **向上伸縮**，即 scale up）。許多處理器，記憶體和磁碟可以在同一個作業系統下相互連線，快速的相互連線允許任意處理器訪問記憶體或磁碟的任意部分。在這種 **共享記憶體架構**（shared-memory architecture）中，所有的元件都可以看作一臺單獨的機器ⁱ。

ⁱ. 在大型機中，儘管任意處理器都可以訪問記憶體的任意部分，但總有一些記憶體區域與一些處理器更接近（稱為 **非均勻記憶體訪問**（nonuniform memory access, NUMA）【1】）。為了有效利用這種架構特性，需要對處理進行細分，以便每個處理器主要訪問臨近的記憶體，這意味著即使表面上看起來只有一臺機器在執行，分割槽（partitioning）仍然是必要的。 ↪

共享記憶體方法的問題在於，成本增長速度快於線性增長：一臺有著雙倍處理器數量，雙倍記憶體大小，雙倍磁碟容量的機器，通常成本會遠遠超過原來的兩倍。而且可能因為存在瓶頸，並不足以處理雙倍的載荷。

共享記憶體架構可以提供有限的容錯能力，高階機器可以使用熱插拔的元件（不關機更換磁碟，記憶體模組，甚至處理器）——但它必然囿於單個地理位置的桎梏。

另一種方法是 **共享磁碟架構**（shared-disk architecture），它使用多臺具有獨立處理器和記憶體的機器，但將資料儲存在機器之間共享的磁碟陣列上，這些磁碟透過快速網路連線ⁱⁱ。這種架構用於某些資料倉庫，但競爭和鎖定的開銷限制了共享磁碟方法的可伸縮性【2】。

ⁱⁱ. 瀑路附屬儲存（Network Attached Storage, NAS），或 儲存區網路（Storage Area Network, SAN） ↪

無共享架構

相比之下，無共享架構【3】（shared-nothing architecture，有時被稱為 **水平伸縮**，即 horizontal scaling，或 **向外伸縮**，即 scaling out）已經相當普及。在這種架構中，執行資料庫軟體的每臺機器 / 虛擬機器都稱為 **節點（node）**。每個節點只使用各自的處理器，記憶體和磁碟。節點之間的任何協調，都是在軟體層面使用傳統網路實現的。

無共享系統不需要使用特殊的硬體，所以你可以用任意機器——比如價效比最好的機器。你也許可以跨多個地理區域分佈資料從而減少使用者延遲，或者在損失一整個資料中心的情況下倖免於難。隨著雲端虛擬機器部署的出現，即使是小公司，現在無需 Google 級別的運維，也可以實現異地分散式架構。

在這一部分裡，我們將重點放在無共享架構上。它不見得是所有場景的最佳選擇，但它是最需要你謹慎從事的架構。如果你的資料分佈在多個節點上，你需要意識到這樣一個分散式系統中約束和權衡——資料庫並不能魔術般地把這些東西隱藏起來。

雖然分散式無共享架構有許多優點，但它通常也會給應用帶來額外的複雜度，有時也會限制你可用資料模型的表達力。在某些情況下，一個簡單的單執行緒程式可以比一個擁有超過 100 個 CPU 核的叢集表現得更好【4】。另一方面，無共享系統可以非常強大。接下來的幾章，將詳細討論分散式資料會帶來的問題。

複製 vs 分割槽

資料分佈在多個節點上有兩種常見的方式：

- **複製（Replication）**

在幾個不同的節點上儲存資料的相同副本，可能放在不同的位置。複製提供了冗餘：如果一些節點不可用，剩餘的節點仍然可以提供資料服務。複製也有助於改善效能。[第五章](#) 將討論複製。

- **分割槽（Partitioning）**

將一個大型資料庫拆分成較小的子集（稱為 **分割槽**，即 partitions），從而不同的分割槽可以指派給不同的 **節點（nodes）**，亦稱 **分片**，即 sharding）。[第六章](#) 將討論分割槽。

複製和分割槽是不同的機制，但它們經常同時使用。如 [圖 II-1](#) 所示。

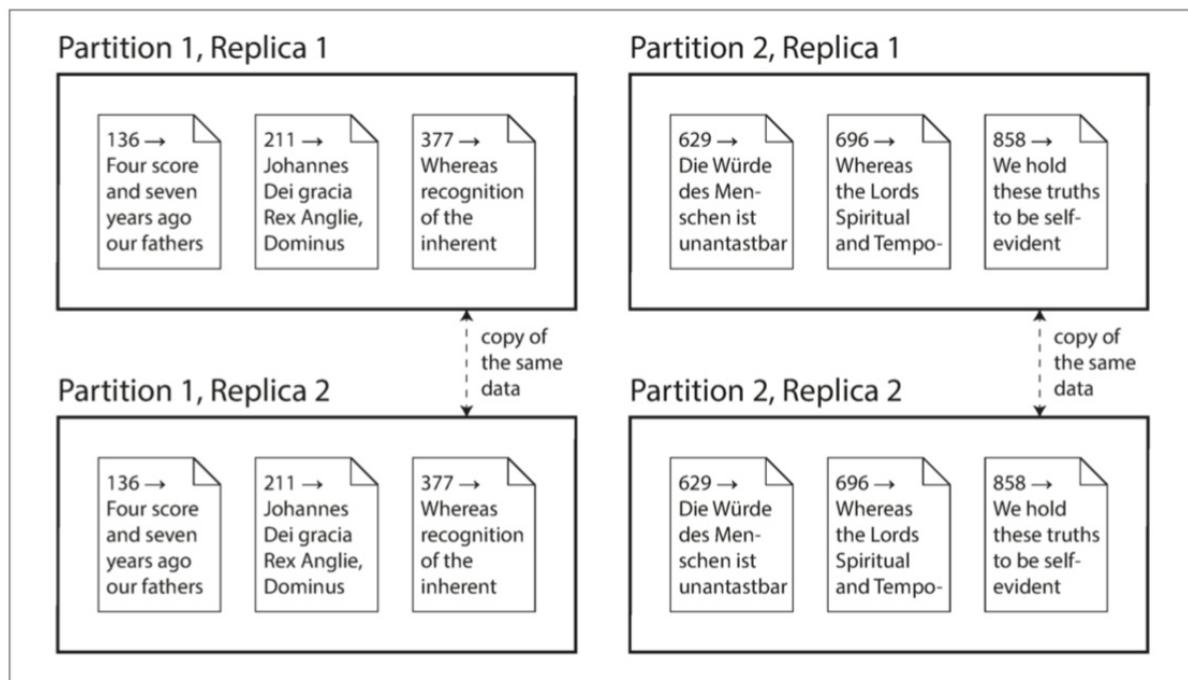


圖 II-1 一個數據庫切分為兩個分割槽，每個分割槽都有兩個副本

理解了這些概念，就可以開始討論在分散式系統中需要做出的困難抉擇。[第七章](#) 將討論 **事務（Transaction）**，這對於瞭解資料系統中可能出現的各種問題，以及我們可以做些什麼很有幫助。[第八章](#) 和 [第九章](#) 將討論分散式系統的根本侷限性。

在本書的 [第三部分](#) 中，將討論如何將多個（可能是分散式的）資料儲存整合為一個更大的系統，以滿足複雜的應用需求。但首先，我們來聊聊分散式的資料。

索引

1. 複製
2. 分割槽
3. 事務
4. 分散式系統的麻煩
5. 一致性與共識

參考文獻

1. Ulrich Drepper: “[What Every Programmer Should Know About Memory](#),” akka-dia.org, November 21, 2007.
2. Ben Stopford: “[Shared Nothing vs. Shared Disk Architectures: An Independent View](#),” benstopford.com, November 24, 2009.
3. Michael Stonebraker: “[The Case for Shared Nothing](#),” IEEE Database Engineering Bulletin, volume 9, number 1, pages 4–9, March 1986.
4. Frank McSherry, Michael Isard, and Derek G. Murray: “[Scalability! But at What COST?](#),” at 15th USENIX Workshop on Hot Topics in Operating Systems (HotOS), May 2015.

上一章	目錄	下一章
第四章：編碼與演化	設計資料密集型應用	第五章：複製

第五章：複製



與可能出錯的東西比，“不可能”出錯的東西最顯著的特點就是：一旦真的出錯，通常就徹底玩完了。

——道格拉斯·亞當斯（1992）

[TOC]

複製意味著在透過網路連線的多臺機器上保留相同資料的副本。正如在 [第二部分](#) 的介紹中所討論的那樣，我們希望能複製資料，可能出於各種各樣的原因：

- 使得資料與使用者在地理上接近（從而減少延遲）
- 即使系統的一部分出現故障，系統也能繼續工作（從而提高可用性）
- 伸縮可以接受讀請求的機器數量（從而提高讀取吞吐量）

本章將假設你的資料集非常小，每臺機器都可以儲存整個資料集的副本。在 [第六章](#) 中將放寬這個假設，討論對單個機器來說太大的資料集的分割（分片）。在後面的章節中，我們將討論複製資料系統中可能發生的各種故障，以及如何處理這些故障。

如果複製中的資料不會隨時間而改變，那複製就很簡單：將資料複製到每個節點一次就萬事大吉。複製的困難之處在於處理複製資料的 **變更**（change），這就是本章所要講的。我們將討論三種流行的變更復制演算法：**單領導者**（single leader，單主），**多領導者**（multi leader，多主）和**無領導者**（leaderless，無主）。幾乎所有分散式資料庫都使用這三種方法之一。

在複製時需要進行許多權衡：例如，使用同步複製還是非同步複製？如何處理失敗的副本？這些通常是資料庫中的配置選項，細節因資料庫而異，但原理在許多不同的實現中都類似。本章會討論這些決策的後果。

資料庫的複製算得上是老生常談了——70 年代研究得出的基本原則至今沒有太大變化【1】，因為網路的基本約束仍保持不變。然而在研究之外，許多開發人員仍然假設一個數據庫只有一個節點。分散式資料庫變為主流只是最近發生的事。許多程式設計師都是這一領域的新手，因此對於諸如 **最終一致性**（eventual consistency）等問題存在許多誤解。在“[複製延遲問題](#)”一節，我們將更加精確地瞭解最終一致性，並討論諸如 **讀已之寫**（read-your-writes）和 **單調讀**（monotonic read）等內容。

領導者與追隨者

儲存了資料庫複製的每個節點被稱為 **副本**（replica）。當存在多個副本時，會不可避免的出現一個問題：如何確保所有資料都落在了所有的副本上？

每一次向資料庫的寫入操作都需要傳播到所有副本上，否則副本就會包含不一樣的資料。最常見的解決方案被稱為 **基於領導者的複製**（leader-based replication）（也稱 **主動/被動**（active/passive）複製或 **主/從**（master/slave）複製），如 [圖 5-1](#) 所示。它的工作原理如下：

1. 其中一個副本被指定為 **領導者**（leader），也稱為 **主庫**（master|primary）。當客戶端要向資料庫寫入時，它必須將請求傳送給該 **領導者**，其會將新資料寫入其本地儲存。
2. 其他副本被稱為 **追隨者**（followers），亦稱為 **只讀副本**（read replicas）、**從庫**（slaves）、**備庫**（secondaries）或 **熱備**（hot standby）ⁱ。每當領導者將新資料寫入本地儲存時，它也會將資料變更傳送給所有的追隨者，稱之為 **複製日誌**（replication log）或 **變更流**（change stream）。每個跟隨者從領導者拉取日誌，並相應更新其本地資料庫副本，方法是按照與領導者相同的處理順序來進行所有寫入。
3. 當客戶想要從資料庫中讀取資料時，它可以向領導者或任一追隨者進行查詢。但只有領導者才能接受寫入操作（從客戶端的角度來看從庫都是隻讀的）。

ⁱ 不同的人對 **熱**（hot）、**溫**（warm）和 **冷**（cold）備份伺服器有不同的定義。例如在 PostgreSQL 中，**熱備**（hot standby）指的是能接受客戶端讀請求的副本。而 **溫備**（warm standby）只是追隨領導者，但不處理客戶端的任何查詢。就本書而言，這些差異並不重要。 ↪

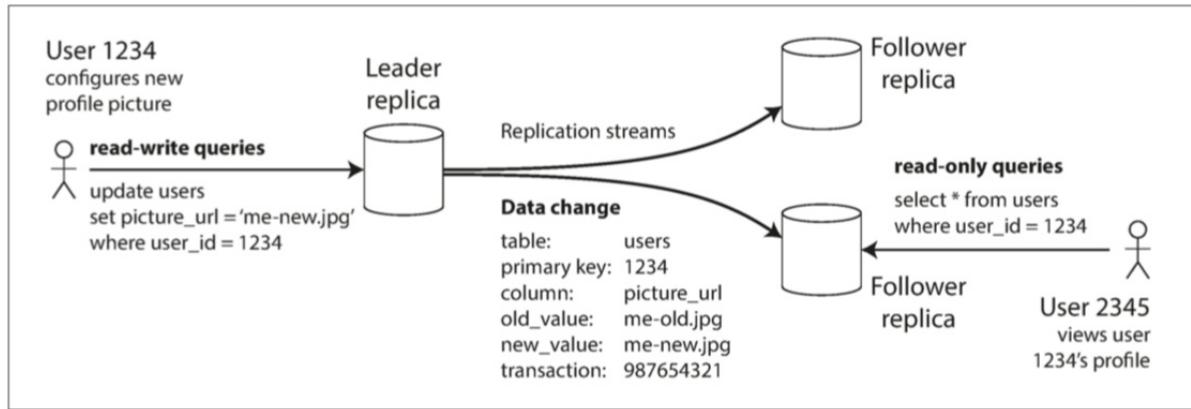


圖 5-1 基於領導者的（主/從）複製

這種複製模式是許多關係資料庫的內建功能，如 PostgreSQL（從 9.0 版本開始）、MySQL、Oracle Data Guard [2] 和 SQL Server 的 AlwaysOn 可用性組 [3]。它也被用於一些非關係資料庫，包括 MongoDB、RethinkDB 和 Espresso [4]。最後，基於領導者的複製並不僅限於資料庫：像 Kafka [5] 和 RabbitMQ 高可用佇列 [6] 這樣的分散式訊息代理也使用它。某些網路檔案系統，例如 DRBD 這樣的塊複製裝置也與之類似。

同步複製與非同步複製

複製系統的一個重要細節是：複製是 **同步 (synchronously)** 發生的還是 **非同步 (asynchronously)** 發生的。（在關係型資料庫中這通常是一個配置項，其他系統則通常硬編碼為其中一個）。

想象一下 圖 5-1 中發生的場景，即網站的使用者更新他們的個人頭像。在某個時間點，客戶向主庫傳送更新請求；不久之後主庫就收到了請求。在某個時間點，主庫又會將資料變更轉發給自己的從庫。最終，主庫通知客戶更新成功。

圖 5-2 顯示了系統各個元件之間的通訊：使用者客戶端、主庫和兩個從庫。時間從左向右流動。請求或響應訊息用粗箭頭表示。

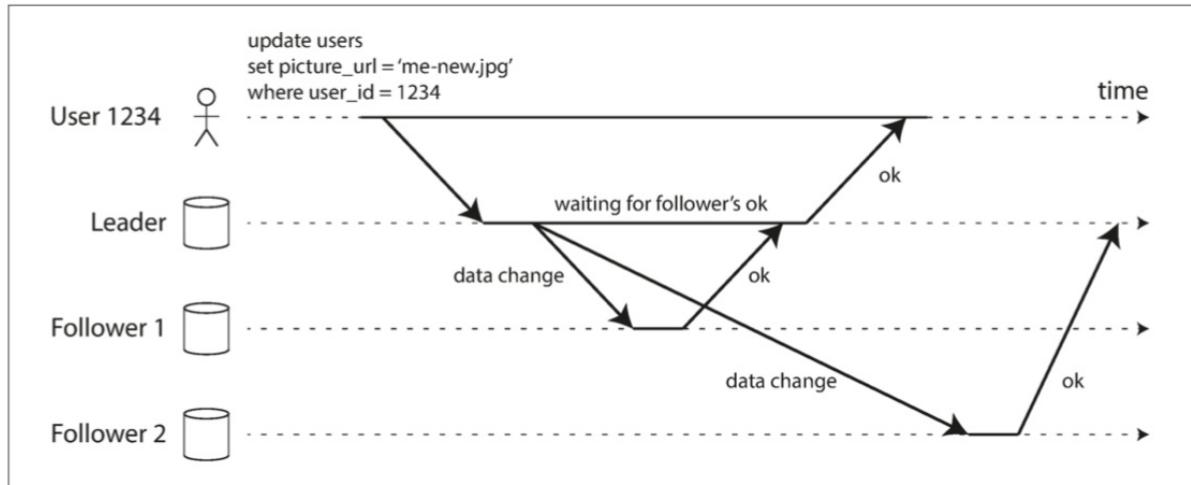


圖 5-2 基於領導者的複製：一個同步從庫和一個非同步從庫

在 圖 5-2 的示例中，從庫 1 的複製是同步的：在向用戶報告寫入成功並使結果對其他使用者可見之前，主庫需要等待從庫 1 的確認，確保從庫 1 已經收到寫入操作。而從庫 2 的複製是非同步的：主庫傳送訊息，但不等待該從庫的響應。

在這幅圖中，從庫 2 處理訊息前存在一個顯著的延遲。通常情況下，複製的速度相當快：大多數資料庫系統能在不到一秒內完成從庫的同步，但它們不能提供複製用時的保證。有些情況下，從庫可能落後主庫幾分鐘或更久，例如：從庫正在從故障中恢復，系統正在最大容量附近執行，或者當節點間存在網路問題時。

同步複製的優點是，從庫能保證有與主庫一致的最新資料副本。如果主庫突然失效，我們可以確信這些資料仍然能在從庫上找到。缺點是，如果同步從庫沒有響應（比如它已經崩潰，或者出現網路故障，或其它任何原因），主庫就無法處理寫入操作。主庫必須阻止所有寫入，並等待同步副本再次可用。

因此，將所有從庫都設定為同步的是不切實際的：任何一個節點的中斷都會導致整個系統停滯不前。實際上，如果在資料庫上啟用同步複製，通常意味著其中一個從庫是同步的，而其他的從庫則是非同步的。如果該同步從庫變得不可用或緩慢，則將一個非同步從庫改為同步執行。這保證你至少在兩個節點上擁有最新的資料副本：主庫和同步從庫。這種配置有時也被稱為 **半同步** (**semi-synchronous**) 【7】。

通常情況下，基於領導者的複製都配置為完全非同步。在這種情況下，如果主庫失效且不可恢復，則任何尚未複製給從庫的寫入都會丟失。這意味著即使已經向客戶端確認成功，寫入也不能保證是 **持久** (**Durable**) 的。然而，一個完全非同步的配置也有優點：即使所有的從庫都落後了，主庫也可以繼續處理寫入。

弱化的永續性可能聽起來像是一個壞的折衷，但非同步複製其實已經被廣泛使用了，特別是在有很多從庫的場景下，或者當從庫在地理上分佈很廣的時候。我們將在討論“[複製延遲問題](#)”時回到這個問題。

關於複製的研究

對於非同步複製系統而言，主庫故障時會丟失資料可能是一個嚴重的問題，因此研究人員仍在研究不丟資料但仍能提供良好效能和可用性的複製方法。例如，鏈式複製 (**chain replication**) 【8,9】是同步複製的一種變體，已經在一些系統（如 Microsoft Azure Storage 【10,11】）中成功實現。

複製的一致性與 **共識** (**consensus**，使幾個節點就某個值達成一致) 之間有著密切的聯絡，[第九章](#) 將詳細地探討這一領域的理論。本章主要討論實踐中的資料庫常用的簡單複製形式。

設定新從庫

有時候需要設定一個新的從庫：也許是為了增加副本的數量，或替換失敗的節點。如何確保新的從庫擁有主庫資料的精確副本？

簡單地將資料檔案從一個節點複製到另一個節點通常是不夠的：客戶端不斷向資料庫寫入資料，資料總是在不斷地變化，標準的檔案複製會看到資料庫的不同部分在不同的時間點的內容，其結果可能沒有任何意義。

可以透過鎖定資料庫（使其不可用於寫入）來使磁碟上的檔案保持一致，但是這會違背高可用的目標。幸運的是，設定新從庫通常並不需要停機。從概念上講，其過程如下所示：

1. 在某個時刻獲取主庫的一致性快照（如果可能，不必鎖定整個資料庫）。大多數資料庫都具有這個功能，因為它是備份必需的。對於某些場景，可能需要第三方工具，例如用於 MySQL 的 `innobackupex` 【12】。
2. 將快照複製到新的從庫節點。
3. 從庫連線到主庫，並拉取快照之後發生的所有資料變更。這要求快照與主庫複製日誌中的位置精確關聯。該位置有不同的名稱，例如 PostgreSQL 將其稱為 **日誌序列號** (**log sequence number**, LSN)，MySQL 將其稱為 **二進位制日誌座標** (**binlog coordinates**)。
4. 當從庫處理完快照之後積累的資料變更，我們就說它 **趕上** (**caught up**) 了主庫，現在它可以繼續及時處理主庫產生的資料變化了。

建立從庫的實際步驟因資料庫而異。在某些系統中，這個過程是完全自動化的，而在另外一些系統中，它可能是一個需要由管理員手動執行的、有點神秘的多步驟工作流。

處理節點宕機

系統中的任何節點都可能宕機，可能因為意外的故障，也可能由於計劃內的維護（例如，重啟機器以安裝核心安全補丁）。對運維而言，能在系統不中斷服務的情況下重啟單個節點好處多多。我們的目標是，即使個別節點失效，也能保持整個系統執行，並儘可能控制節點停機帶來的影響。

如何透過基於領導者的複製實現高可用？

從庫失效：追趕恢復

在其本地磁碟上，每個從庫記錄從主庫收到的資料變更。如果從庫崩潰並重新啟動，或者，如果主庫和從庫之間的網路暫時中斷，則比較容易恢復：從庫可以從日誌中知道，在發生故障之前處理的最後一個事務。因此，從庫可以連線到主庫，並請求在從庫斷開期間發生的所有資料變更。當應用完所有這些變更後，它就趕上了主庫，並可以像以前一樣繼續接收資料變更流。

主庫失效：故障切換

主庫失效處理起來相當棘手：其中一個從庫需要被提升為新的主庫，需要重新配置客戶端，以將它們的寫操作傳送給新的主庫，其他從庫需要開始拉取來自新主庫的資料變更。這個過程被稱為 **故障切換（failover）**。

故障切換可以手動進行（通知管理員主庫掛了，並採取必要的步驟來建立新的主庫）或自動進行。自動的故障切換過程通常由以下步驟組成：

1. 確認主庫失效。有很多事情可能會出錯：崩潰、停電、網路問題等等。沒有萬無一失的方法來檢測出現了什麼問題，所以大多數系統只是簡單使用 **超時（Timeout）**：節點頻繁地相互來回傳遞訊息，如果一個節點在一段時間內（例如 30 秒）沒有響應，就認為它掛了（因為計劃內維護而故意關閉主庫不算）。
2. 選擇一個新的主庫。這可以透過選舉過程（主庫由剩餘副本以多數選舉產生）來完成，或者可以由之前選定的 **控制器節點（controller node）** 來指定新的主庫。主庫的最佳人選通常是擁有舊主庫最新資料副本的從庫（以最小化資料損失）。讓所有的節點同意一個新的領導者，是一個 **共識** 問題，將在 [第九章](#) 詳細討論。
3. 重新配置系統以啟用新的主庫。客戶端現在需要將它們的寫請求傳送給新主庫（將在“[請求路由](#)”中討論這個問題）。如果舊主庫恢復，可能仍然認為自己是主庫，而沒有意識到其他副本已經讓它失去領導權了。系統需要確保舊主庫意識到新主庫的存在，併成為一個從庫。

故障切換的過程中有很多地方可能出錯：

- 如果使用非同步複製，則新主庫可能沒有收到老主庫宕機前最後的寫入操作。在選出新主庫後，如果老主庫重新加入叢集，新主庫在此期間可能會收到衝突的寫入，那這些寫入該如何處理？最常見的解決方案是簡單丟棄老主庫未複製的寫入，這很可能打破客戶對於資料永續性的期望。
- 如果資料庫需要和其他外部儲存相協調，那麼丟棄寫入內容是極其危險的操作。例如在 GitHub [【13】](#) 的一場事故中，一個過時的 MySQL 從庫被提升為主庫。資料庫使用自增 ID 作為主鍵，因為新主庫的計數器落後於老主庫的計數器，所以新主庫重新分配了一些已經被老主庫分配掉的 ID 作為主鍵。這些主鍵也在 Redis 中使用，主鍵重用使得 MySQL 和 Redis 中的資料產生不一致，最後導致一些私有資料洩漏到錯誤的使用者手中。
- 發生某些故障時（見 [第八章](#)）可能會出現兩個節點都以為自己是主庫的情況。這種情況稱為 **腦裂（split brain）**，非常危險：如果兩個主庫都可以接受寫操作，卻沒有衝突解決機制（請參閱“[多主複製](#)”），那麼資料就可能丟失或損壞。一些系統採取了安全防範措施：當檢測到兩個主庫節點同時存在時會關閉其中一個節點ⁱⁱ，但設計粗糙的機制可能最後會導致兩個節點都被關閉 [【14】](#)。

ⁱⁱ. 這種機制稱為 **屏障（fencing）**，或者更充滿感情的術語是：爆彼之頭（**Shoot The Other Node In The Head, STONITH**）。我們將在“[領導者和鎖](#)”中對屏障進行詳細討論。 ↪

- 主庫被宣告死亡之前的正確超時應該怎麼配置？在主庫失效的情況下，超時時間越長意味著恢復時間也越長。但是如果超時設定太短，又可能會出現不必要的故障切換。例如，臨時的負載峰值可能導致節點的響應時間增加到超出超時時間，或者網路故障也可能導致資料包延遲。如果系統已經處於高負載或網路問題的困擾之中，那麼不必要的故障切換可能會讓情況變得更糟糕。

這些問題沒有簡單的解決方案。因此，即使軟體支援自動故障切換，不少運維團隊還是更願意手動執行故障切換。

節點故障、不可靠的網路、對副本一致性、永續性、可用性和延遲的權衡，這些問題實際上是分散式系統中的基本問題。[第八章](#) 和 [第九章](#) 將更深入地討論它們。

複製日誌的實現

基於領導者的複製在底層是如何工作的？實踐中有好幾種不同的複製方式，所以先簡要地看一下。

基於語句的複製

在最簡單的情況下，主庫記錄下它執行的每個寫入請求（語句，即 statement）並將該語句日誌傳送給從庫。對於關係資料庫來說，這意味著每個 `INSERT`、`UPDATE` 或 `DELETE` 語句都被轉發給每個從庫，每個從庫解析並執行該 SQL 語句，就像直接從客戶端收到一樣。

雖然聽上去很合理，但有很多問題會搞砸這種複製方式：

- 任何呼叫 **非確定性函式 (nondeterministic)** 的語句，可能會在每個副本上生成不同的值。例如，使用 `NOW()` 獲取當前日期時間，或使用 `RAND()` 獲取一個隨機數。
- 如果語句使用了 **自增列 (auto increment)**，或者依賴於資料庫中的現有資料（例如，`UPDATE ... WHERE <某些條件>`），則必須在每個副本上按照完全相同的順序執行它們，否則可能會產生不同的效果。當有多個併發執行的事務時，這可能成為一個限制。
- 有副作用的語句（例如：觸發器、儲存過程、使用者定義的函式）可能會在每個副本上產生不同的副作用，除非副作用是絕對確定性的。

的確有辦法繞開這些問題——例如，當語句被記錄時，主庫可以用固定的返回值替換掉任何不確定的函式呼叫，以便所有從庫都能獲得相同的值。但是由於邊緣情況實在太多了，現在通常會選擇其他的複製方法。

基於語句的複製在 5.1 版本前的 MySQL 中被使用到。因為它相當繁瑣，現在有時候也還在用。但現在在預設情況下，如果語句中存在任何不確定性，MySQL 會切換到基於行的複製（稍後討論）。VoltDB 使用了基於語句的複製，但要求事務必須是確定性的，以此來保證安全【15】。

傳輸預寫式日誌 (WAL)

在 [第三章](#) 中，我們討論了儲存引擎如何在磁碟上表示資料，我們也發現了通常會將寫操作追加到日誌中：

- 對於日誌結構儲存引擎（請參閱“[SSTables 和 LSM 樹](#)”），日誌是主要的儲存位置。日誌段在後臺壓縮，並進行垃圾回收。
- 對於覆寫單個磁碟塊的 **B 樹**，每次修改都會先寫入 **預寫式日誌 (Write Ahead Log, WAL)**，以便崩潰後索引可以恢復到一個一致的狀態。

在任何一種情況下，該日誌都是包含了所有資料庫寫入的僅追加位元組序列。可以使用完全相同的日誌在另一個節點上構建副本：除了將日誌寫入磁碟之外，主庫還可以透過網路將其傳送給從庫。

透過使用這個日誌，從庫可以構建一個與主庫一模一樣的資料結構複製。

這種複製方法在 PostgreSQL 和 Oracle 等一些產品中被使用到【16】。其主要缺點是日誌記錄的資料非常底層：WAL 包含哪些磁碟塊中的哪些位元組發生了更改。這使複製與儲存引擎緊密耦合。如果資料庫將其儲存格式從一個版本更改为另一個版本，通常不可能在主庫和從庫上執行不同版本的資料庫軟體。

看上去這可能只是一個小的實現細節，但卻可能對運維產生巨大的影響。如果複製協議允許從庫使用比主庫更新的軟體版本，則可以先升級從庫，然後執行故障切換，使升級後的節點之一成為新的主庫，從而允許資料庫軟體的零停機升級。如果複製協議不允許版本不匹配（傳輸 WAL 經常出現這種情況），則此類升級需要停機。

邏輯日誌複製（基於行）

另一種方法是對複製和儲存引擎使用不同的日誌格式，這樣可以將複製日誌從儲存引擎的內部實現中解耦出來。這種複製日誌被稱為邏輯日誌（logical log），以將其與儲存引擎的（物理）資料表示區分開來。

關係資料庫的邏輯日誌通常是以行的粒度來描述對資料庫表的寫入記錄的序列：

- 對於插入的行，日誌包含所有列的新值。
- 對於刪除的行，日誌包含足夠的資訊來唯一標識被刪除的行，這通常是主鍵，但如果表上沒有主鍵，則需要記錄所有列的舊值。
- 對於更新的行，日誌包含足夠的資訊來唯一標識被更新的行，以及所有列的新值（或至少所有已更改的列的新值）。

修改多行的事務會生成多條這樣的日誌記錄，後面跟著一條指明事務已經提交的記錄。MySQL 的二進位制日誌（當配置為使用基於行的複製時）使用了這種方法【17】。

由於邏輯日誌與儲存引擎的內部實現是解耦的，系統可以更容易地做到向後相容，從而使主庫和從庫能夠執行不同版本的資料庫軟體，或者甚至不同的儲存引擎。

對於外部應用程式來說，邏輯日誌格式也更容易解析。如果要將資料庫的內容傳送到外部系統，例如複製到資料倉庫進行離線分析，或建立自定義索引和快取【18】，這一點會很有用。這種技術被稱為 **資料變更捕獲 (change data capture)**，[第十一章](#) 將重新講到它。

基於觸發器的複製

到目前為止描述的複製方法是由資料庫系統實現的，不涉及任何應用程式程式碼。在很多情況下，這就是你想要的。但在某些情況下需要更多的靈活性。例如，如果你只想複製資料的一個子集，或者想從一種資料庫複製到另一種資料庫，或者如果你需要衝突解決邏輯（請參閱 “處理寫入衝突”），則可能需要將複製操作上移到應用程式層。

一些工具，如 Oracle Golden Gate 【19】，可以透過讀取資料庫日誌，使得其他應用程式可以使用資料。另一種方法是使用許多關係資料庫自帶的功能：觸發器和儲存過程。

觸發器允許你將資料更改（寫入事務）發生時自動執行的自定義應用程式程式碼註冊在資料庫系統中。觸發器有機會將更改記錄到一個單獨的表中，使用外部程式讀取這個表，再加上一些必要的業務邏輯，就可以將資料變更復制到另一個系統去。例如，Databus for Oracle 【20】和 Bucardo for Postgres 【21】就是這樣工作的。

基於觸發器的複製通常比其他複製方法具有更高的開銷，並且比資料庫內建的複製更容易出錯，也有很多限制。然而由於其靈活性，它仍然是很有用的。

複製延遲問題

容忍節點故障只是需要複製的一個原因。正如在 [第二部分](#) 的介紹中提到的，其它原因還包括可伸縮性（處理比單個機器更多的請求）和延遲（讓副本在地理位置上更接近使用者）。

基於領導者的複製要求所有寫入都由單個節點處理，但只讀查詢可以由任何一個副本來處理。所以對於讀多寫少的場景（Web 上的常見模式），一個有吸引力的選擇是建立很多從庫，並將讀請求分散到所有的從庫上去。這樣能減小主庫的負載，並允許由附近的副本來處理讀請求。

在這種讀伸縮（read-scaling）的體系結構中，只需新增更多的從庫，就可以提高只讀請求的服務容量。但是，這種方法實際上只適用於非同步複製——如果嘗試同步複製到所有從庫，則單個節點故障或網路中斷將導致整個系統都無法寫入。而且節點越多越有可能出現個別節點宕機的情況，所以完全同步的配置將是非常不可靠的。

不幸的是，當應用程式從非同步從庫讀取時，如果從庫落後，它可能會看到過時的資訊。這會導致資料庫中出現明顯的不一致：同時對主庫和從庫執行相同的查詢，可能得到不同的結果，因為並非所有的寫入都反映在從庫中。這種不一致只是一個暫時的狀態——如果停止寫入資料庫並等待一段時間，從庫最終會趕上並與主庫保持一致。出於這個原因，這種效應被稱為 **最終一致性 (eventual consistency)** 【22,23】。ⁱⁱⁱ

ⁱⁱⁱ. 道格拉斯·特里 (Douglas Terry) 等人【24】創造了最終一致性這個術語，並經由 Werner Vogels 【22】的推廣，成為了許多 NoSQL 專案的口號。然而，最終一致性並不只屬於 NoSQL 資料庫：關係型資料庫中的非同步複製從庫也有相同的特性。 ↪

最終一致性中的“最終”一詞有意進行了模糊化：總的來說，副本落後的程度是沒有限制的。在正常的操作中，**複製延遲 (replication lag)**，即寫入主庫到反映至從庫之間的延遲，可能僅僅是幾分之一秒，在實踐中並不顯眼。但如果系統在接近極限的情況下執行，或網路中存在問題時，延遲可以輕而易舉地超過幾秒，甚至達到幾分鐘。

因為滯後時間太長引入的不一致性，不僅僅是一個理論問題，更是應用設計中會遇到的真實問題。本節將重點介紹三個在複製延遲時可能發生的問題例項，並簡述解決這些問題的一些方法。

讀己之寫

許多應用讓使用者提交一些資料，然後檢視他們提交的內容。可能是使用者資料庫中的記錄，也可能是對討論主題的評論，或其他類似的內容。提交新資料時，必須將其傳送給主庫，但是當用戶檢視資料時，可以透過從庫進行讀取。如果資料經常被檢視，但只是偶爾寫入，這是非常合適的。

但對於非同步複製，問題就來了。如 圖 5-3 所示：如果使用者在寫入後馬上就檢視資料，則新資料可能尚未到達副本。對使用者而言，看起來好像是剛提交的資料丟失了，所以他們不高興是可以理解的。

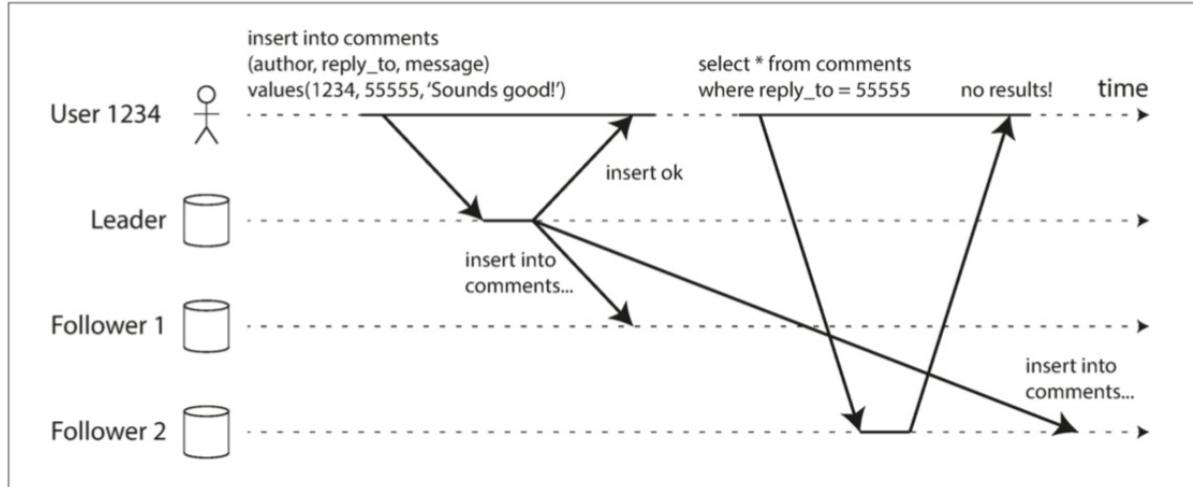


圖 5-3 使用者寫入後從舊副本中讀取資料。需要寫後讀 (read-after-write) 的一致性來防止這種異常

在這種情況下，我們需要 寫後讀一致性 (read-after-write consistency)，也稱為 讀己之寫一致性 (read-your-writes consistency) 【24】。這是一個保證，如果使用者重新載入頁面，他們總會看到他們自己提交的任何更新。它不會對其他使用者的寫入做出承諾：其他使用者的更新可能稍等才會看到。它保證使用者自己的輸入已被正確儲存。

如何在基於領導者的複製系統中實現寫後讀一致性？有各種可能的技術，這裡說一些：

- 對於使用者 可能修改過 的內容，總是從主庫讀取；這就要求得有辦法不透過實際的查詢就可以知道使用者是否修改了某些東西。舉個例子，社交網路上的使用者個人資料資訊通常只能由使用者本人編輯，而不能由其他人編輯。因此一個簡單的規則就是：總是從主庫讀取使用者自己的檔案，如果要讀取其他使用者的檔案就去從庫。
- 如果應用中的大部分內容都可能被使用者編輯，那這種方法就沒用了，因為大部分內容都必須從主庫讀取（讀伸縮就沒效果了）。在這種情況下可以使用其他標準來決定是否從主庫讀取。例如可以跟蹤上次更新的時間，在上次更新後的一分鐘內，從主庫讀。還可以監控從庫的複製延遲，防止向任何滯後主庫超過一分鐘的從庫發出查詢。
- 客戶端可以記住最近一次寫入的時間戳，系統需要確保從庫在處理該使用者的讀取請求時，該時間戳前的變更都已經傳播到了本從庫中。如果當前從庫不夠新，則可以從另一個從庫讀取，或者等待從庫追趕上來。這裡的時間戳可以是邏輯時間戳（表示寫入順序的東西，例如日誌序列號）或實際的系統時鐘（在這種情況下，時鐘同步變得至關重要，請參閱 “[不可靠的時鐘](#)”）。
- 如果你的副本分佈在多個數據中心（為了在地理上接近使用者或者出於可用性目的），還會有額外的複雜性。任何需要由主庫提供服務的請求都必須路由到包含該主庫的資料中心。

另一種複雜的情況發生在同一位使用者從多個裝置（例如桌面瀏覽器和移動 APP）請求服務的時候。這種情況下可能就需要提供跨裝置的寫後讀一致性：如果使用者在一個裝置上輸入了一些資訊，然後在另一個裝置上檢視，則應該看到他們剛輸入的資訊。

在這種情況下，還有一些需要考慮的問題：

- 記住使用者上次更新時間戳的方法變得更加困難，因為一個裝置上執行的程式不知道另一個裝置上發生了什麼。需要對這些元資料進行中心化的儲存。
- 如果副本分佈在不同的資料中心，很難保證來自不同裝置的連線會路由到同一資料中心。（例如，使用者的臺式計

算機使用家庭寬頻連線，而移動裝置使用蜂窩資料網路，則裝置的網路路由可能完全不同）。如果你的方法需要讀主庫，可能首先需要把來自該使用者所有裝置的請求都路由到同一個資料中心。

單調讀

在從非同步從庫讀取時可能發生的異常的第二個例子是使用者可能會遇到 時光倒流（moving backward in time）。

如果使用者從不同從庫進行多次讀取，就可能發生這種情況。例如，[圖 5-4](#) 顯示了使用者 2345 兩次進行相同的查詢，首先查詢了一個延遲很小的從庫，然後是一個延遲較大的從庫（如果使用者重新整理網頁時每個請求都被路由到一個隨機的伺服器，這種情況就很有可能發生）。第一個查詢返回了最近由使用者 1234 新增的評論，但是第二個查詢不返回任何東西，因為滯後的從庫還沒有拉取到該寫入內容。實際上可以認為第二個查詢是在比第一個查詢更早的時間點上觀察系統。如果第一個查詢沒有返回任何內容，那問題並不大，因為使用者 2345 可能不知道使用者 1234 最近添加了評論。但如果使用者 2345 先看見使用者 1234 的評論，然後又看到它消失，這就會讓人覺得非常困惑了。

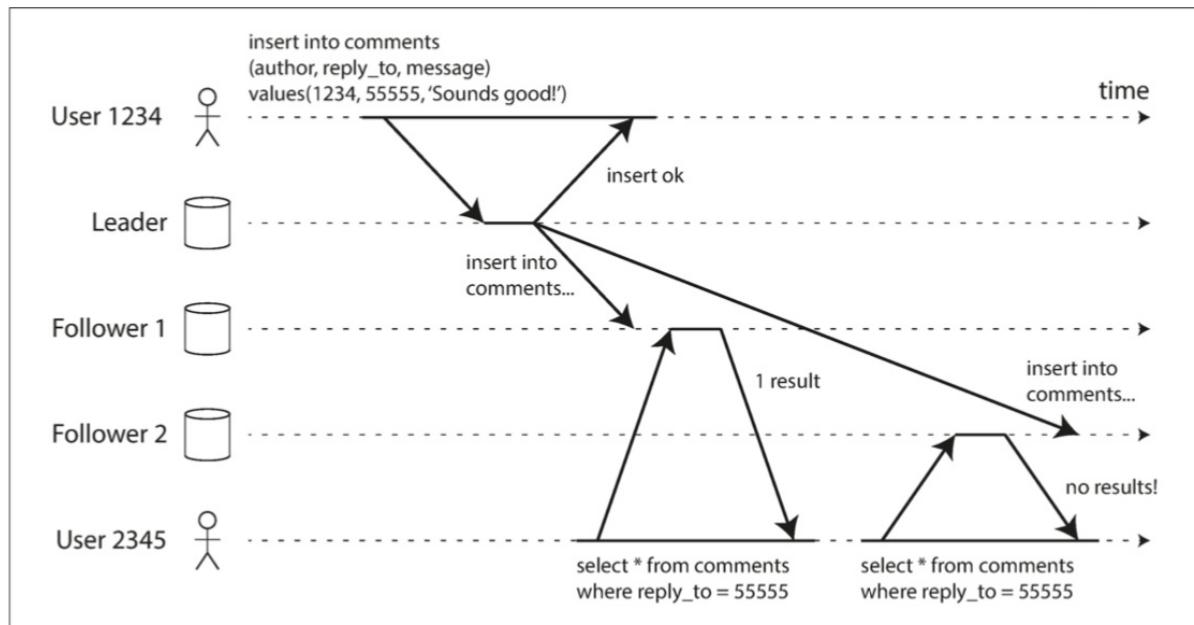


圖 5-4 使用者首先從新副本讀取，然後從舊副本讀取。時間看上去回退了。為了防止這種異常，我們需要單調的讀取。

單調讀 (monotonic reads) [【23】](#) 可以保證這種異常不會發生。這是一個比 **強一致性 (strong consistency)** 更弱，但比 **最終一致性 (eventual consistency)** 更強的保證。當讀取資料時，你可能會看到一箇舊值；單調讀僅意味著如果一個使用者順序地進行多次讀取，則他們不會看到時間回退，也就是說，如果已經讀取到較新的資料，後續的讀取不會得到更舊的資料。

實現單調讀的一種方式是確保每個使用者總是從同一個副本進行讀取（不同的使用者可以從不同的副本讀取）。例如，可以基於使用者 ID 的雜湊來選擇副本，而不是隨機選擇副本。但是，如果該副本出現故障，使用者的查詢將需要重新路由到另一個副本。

一致字首讀

第三個複製延遲異常的例子違反了因果律。想像一下 Poons 先生和 Cake 夫人之間的以下簡短對話：

Mr. Poons

Mrs. Cake，你能看到多遠的未來？

Mrs. Cake

通常約十秒鐘，Mr. Poons.

這兩句話之間有因果關係：Cake 夫人聽到了 Poons 先生的問題並回答了這個問題。

現在，想像第三個人正在透過從庫來聽這個對話。Cake 夫人說的內容是從一個延遲很低的從庫讀取的，但 Poons 先生所說的內容，從庫的延遲要大的多（見 圖 5-5）。於是，這個觀察者會聽到以下內容：

Mrs. Cake

通常約十秒鐘，Mr. Poons.

Mr. Poons

Mrs. Cake，你能看到多遠的未來？

對於觀察者來說，看起來好像 Cake 夫人在 Poons 先生提問前就回答了這個問題。這種超能力讓人印象深刻，但也會把人搞糊塗。【25】。

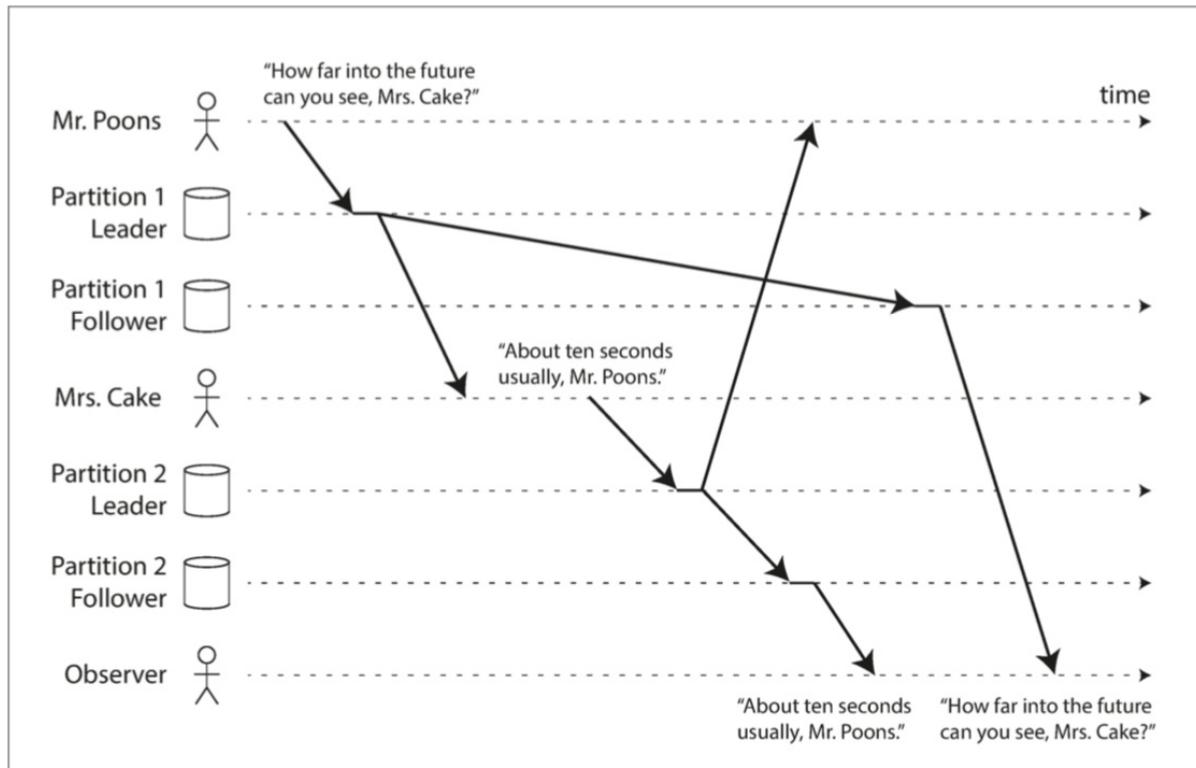


圖 5-5 如果某些分割槽的複製速度慢於其他分割槽，那麼觀察者可能會在看到問題之前先看到答案。

要防止這種異常，需要另一種型別的保證：一致字首讀（consistent prefix reads）【23】。這個保證的意思是說：如果一系列寫入按某個順序發生，那麼任何人讀取這些寫入時，也會看見它們以同樣的順序出現。

這是 分割槽（partitioned） 或 分片（sharded） 資料庫中的一個特殊問題，我們將在 第六章 中討論分割槽資料庫。如果資料庫總是以相同的順序應用寫入，而讀取總是看到一致的字首，那麼這種異常不會發生。但是在許多分散式資料庫中，不同的分割槽獨立執行，因此不存在 全域性的寫入順序：當用戶從資料庫中讀取資料時，可能會看到資料庫的某些部分處於較舊的狀態，而某些則處於較新的狀態。

一種解決方案是，確保任何因果相關的寫入都寫入相同的分割槽，但在一些應用中可能無法高效地完成這種操作。還有一些顯式跟蹤因果依賴關係的演算法，我們將在 “[此前發生”的關係和併發](#)” 一節中回到這個話題。

複製延遲的解決方案

在使用最終一致的系統時，如果複製延遲增加到幾分鐘甚至幾小時，則應該考慮應用程式的行為。如果答案是“沒問題”，那很好。但如果結果對於使用者來說是不好的體驗，那麼設計系統來提供更強的保證（例如 寫後讀）是很重要的。明明是非同步複製卻假設複製是同步的，這是很多麻煩的根源。

如前所述，應用程式可以提供比底層資料庫更強有力的保證，例如透過主庫進行某種讀取。但在應用程式程式碼中處理這些問題是複雜的，容易出錯。

如果應用程式開發人員不必擔心微妙的複製問題，並可以信賴他們的資料庫“做了正確的事情”，那該多好呀。這就是 **事務 (transaction)** 存在的原因：資料庫透過事務提供強大的保證，所以應用程式可以更加簡單。

單節點事務已經存在了很長時間。然而在走向分散式（複製和分割槽）資料庫時，許多系統放棄了事務，聲稱事務在效能和可用性上的代價太高，並斷言在可伸縮系統中最終一致性是不可避免的。這個敘述有一些道理，但過於簡單了，本書其餘部分將提出更為細緻的觀點。我們將在 [第七章](#) 和 [第九章](#) 回到事務的話題，並將在 [第三部分](#) 討論一些替代機制。

多主複製

本章到目前為止，我們只考慮了使用單個主庫的複製架構。雖然這是一種常見的方法，但還有其它一些有趣的選擇。

基於領導者的複製有一個主要的缺點：只有一個主庫，而且所有的寫入都必須透過它^{iv}。如果出於任何原因（例如和主庫之間的網路連線中斷）無法連線到主庫，就無法向資料庫寫入。

^{iv}. 如果資料庫被分割槽（見 [第六章](#)），每個分割槽都有一個主庫。不同的分割槽的主庫可能在不同的節點上，但是每個分割槽都必須有一個主庫。 ↪

基於領導者的複製模型的自然延伸是允許多個節點接受寫入。複製仍然以同樣的方式發生：處理寫入的每個節點都必須將該資料變更轉發給所有其他節點。我們將其稱之為 **多領導者配置**（multi-leader configuration，也稱多主、多活複製，即 master-master replication 或 active/active replication）。在這種情況下，每個主庫同時是其他主庫的從庫。

多主複製的應用場景

在單個數據中心內部使用多個主庫的配置沒有太大意義，因為其導致的複雜性已經超過了能帶來的好處。但在一些情況下，這種配置也是合理的。

運維多個數據中心

假如你有一個數據庫，副本分散在好幾個不同的資料中心（可能會用來容忍單個數據中心的故障，或者為了在地理上更接近使用者）。如果使用常規的基於領導者的複製設定，主庫必須位於其中一個數據中心，且所有寫入都必須經過該資料中心。

多主配置中可以在每個資料中心都有主庫。[圖 5-6](#) 展示了這個架構。在每個資料中心內使用常規的主從複製；在資料中心之間，每個資料中心的主庫都會將其更改複製到其他資料中心的主庫中。

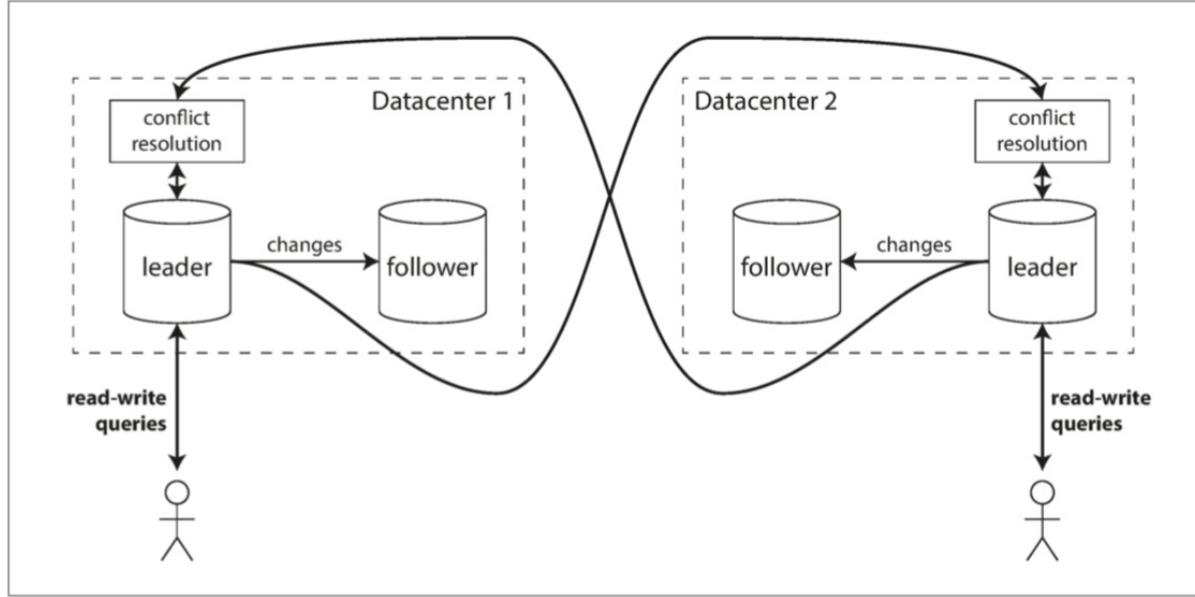


圖 5-6 跨多個數據中心的多主複製

我們來比較一下在運維多個數據中心時，單主和多主的適應情況：

- 效能

在單主配置中，每個寫入都必須穿過網際網路，進入主庫所在的資料中心。這可能會增加寫入時間，並可能違背了設定多個數據中心的初心。在多主配置中，每個寫操作都可以在本地資料中心進行處理，並與其他資料中心非同步複製。因此，資料中心之間的網路延遲對使用者來說是透明的，這意味著感覺到的效能可能會更好。

- 容忍資料中心停機

在單主配置中，如果主庫所在的資料中心發生故障，故障切換必須使另一個數據中心裡的從庫成為主庫。在多主配置中，每個資料中心可以獨立於其他資料中心繼續執行，並且當發生故障的資料中心歸隊時，複製會自動趕上。

- 容忍網路問題

資料中心之間的通訊通常穿過公共網際網路，這可能不如資料中心內的本地網路可靠。單主配置對資料中心之間的連線問題非常敏感，因為透過這個連線進行的寫操作是同步的。採用非同步複製功能的多主配置通常能更好地承受網路問題：臨時的網路中斷並不會妨礙正在處理的寫入。

有些資料庫預設情況下支援多主配置，但使用外部工具實現也很常見，例如用於 MySQL 的 Tungsten Replicator [26]，用於 PostgreSQL 的 BDR [27] 以及用於 Oracle 的 GoldenGate [19]。

儘管多主複製有這些優勢，但也有一個很大的缺點：兩個不同的資料中心可能會同時修改相同的資料，寫衝突是必須解決的（如 圖 5-6 中的“衝突解決（conflict resolution）”）。本書將在“處理寫入衝突”中詳細討論這個問題。

由於多主複製在許多資料庫中都屬於改裝的功能，所以常常存在微妙的配置缺陷，且經常與其他資料庫功能之間出現意外的反應。比如自增主鍵、觸發器、完整性約束等都可能會有麻煩。因此，多主複製往往被認為是危險的領域，應儘可能避免 [28]。

需要離線操作的客戶端

多主複製的另一種適用場景是：應用程式在斷網之後仍然需要繼續工作。

例如，考慮手機、膝上型電腦和其他裝置上的日曆應用。無論裝置目前是否有網際網路連線，你需要能隨時檢視你的會議（發出讀取請求），輸入新的會議（發出寫入請求）。如果在離線狀態下進行任何更改，則裝置下次上線時，需要與伺服器和其他裝置同步。

在這種情況下，每個裝置都有一個充當主庫的本地資料庫（它接受寫請求），並且在所有裝置上的日曆副本之間同步時，存在非同步的多主複製過程。複製延遲可能是幾小時甚至幾天，具體取決於何時可以訪問網際網路。

從架構的角度來看，這種設定實際上與資料中心之間的多主複製類似，每個裝置都是一個“資料中心”，而它們之間的網路連線是極度不可靠的。從歷史上各類日曆同步功能的破爛實現可以看出，想把多主複製用好是多麼困難的一件事。

有一些工具旨在使這種多主配置更容易。例如，CouchDB 就是為這種操作模式而設計的【29】。

協同編輯

實時協作編輯應用程式允許多個人同時編輯文件。例如，Etherpad 【30】和 Google Docs 【31】允許多人同時編輯文字文件或電子表格（該演算法在“[自動衝突解決](#)”中簡要討論）。我們通常不會將協作式編輯視為資料庫複製問題，但它與前面提到的離線編輯用例有許多相似之處。當一個使用者編輯文件時，所做的更改將立即應用到其本地副本（Web 瀏覽器或客戶端應用程式中的文件狀態），並非同步複製到伺服器和編輯同一文件的任何其他使用者。

如果要保證不會發生編輯衝突，則應用程式必須先取得文件的鎖定，然後使用者才能對其進行編輯。如果另一個使用者想要編輯同一個文件，他們首先必須等到第一個使用者提交修改並釋放鎖定。這種協作模式相當於主從複製模型下在主節點上執行事務操作。

但是，為了加速協作，你可能希望將更改的單位設定得非常小（例如單次按鍵），並避免鎖定。這種方法允許多個使用者同時進行編輯，但同時也帶來了多主複製的所有挑戰，包括需要解決衝突【32】。

處理寫入衝突

多主複製的最大問題是可能發生寫衝突，這意味著需要解決衝突。

例如，考慮一個由兩個使用者同時編輯的維基頁面，如 圖 5-7 所示。使用者 1 將頁面的標題從 A 更改為 B，並且使用者 2 同時將標題從 A 更改為 C。每個使用者的更改已成功應用到其本地主庫。但當非同步複製時，會發現衝突【33】。單主資料庫中不會出現此問題。

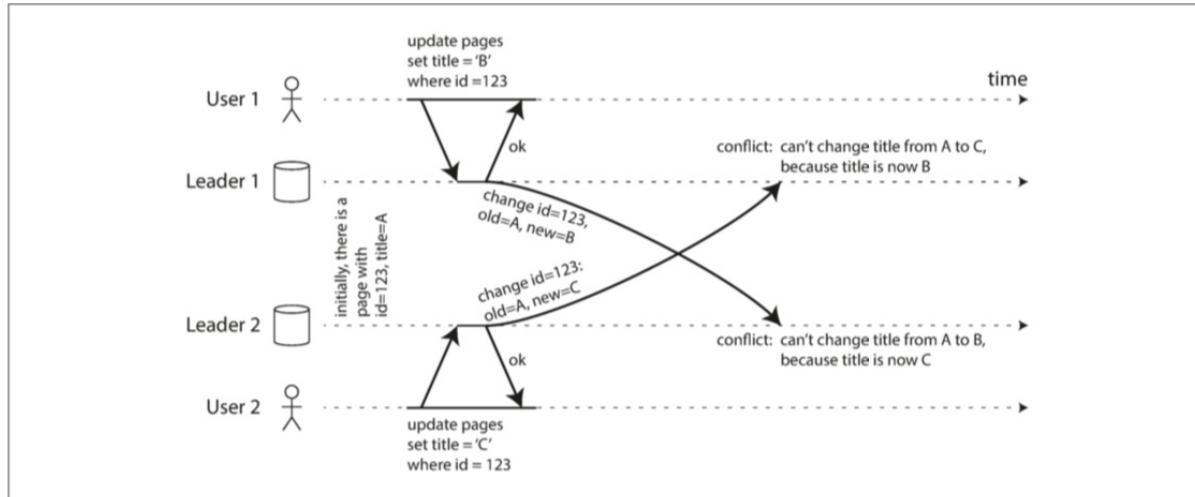


圖 5-7 兩個主庫同時更新同一記錄引起的寫入衝突

同步與非同步衝突檢測

在單主資料庫中，第二個寫入將被阻塞並等待第一個寫入完成，或者中止第二個寫入事務並強制使用者重試。另一方面，在多主配置中，兩個寫入都是成功的，在稍後的某個時間點才能非同步地檢測到衝突。那時再來要求使用者解決衝突可能為時已晚。

原則上，可以使衝突檢測同步 - 即等待寫入被複制到所有副本，然後再告訴使用者寫入成功。但是，透過這樣做，你將失去多主複製的主要優點：允許每個副本獨立地接受寫入。如果你想要同步衝突檢測，那麼你可能不如直接使用單主複製。

避免衝突

處理衝突的最簡單的策略就是避免它們：如果應用程式可以確保特定記錄的所有寫入都透過同一個主庫，那麼衝突就不會發生。由於許多的多主複製實現在處理衝突時處理得相當不好，避免衝突是一個經常被推薦的方法【34】。

例如，在一個使用者可以編輯自己資料的應用程式中，可以確保來自特定使用者的請求始終路由到同一資料中心，並使用該資料中心的主庫進行讀寫。不同的使用者可能有不同的“主”資料中心（可能根據使用者的地理位置選擇），但從任何一位使用者的角度來看，本質上就是單主配置了。

但是，有時你可能需要更改被指定的主庫——可能是因為某個資料中心出現故障，你需要將流量重新路由到另一個數據中心，或者可能是因為使用者已經遷移到另一個位置，現在更接近其它的資料中心。在這種情況下，衝突避免將失效，你必須處理不同主庫同時寫入的可能性。

收斂至一致的狀態

單主資料庫按順序進行寫操作：如果同一個欄位有多個更新，則最後一個寫操作將決定該欄位的最終值。

在多主配置中，沒有明確的寫入順序，所以最終值應該是什麼並不清楚。在 [圖 5-7](#) 中，在主庫 1 中標題首先更新為 B 而後更新為 C；在主庫 2 中，首先更新為 C，然後更新為 B。兩種順序都不比另一種“更正確”。

如果每個副本只是按照它看到寫入的順序寫入，那麼資料庫最終將處於不一致的狀態：最終值將是在主庫 1 的 C 和主庫 2 的 B。這是不可接受的，每個複製方案都必須確保資料最終在所有副本中都是相同的。因此，資料庫必須以一種收斂 (**convergent**) 的方式解決衝突，這意味著所有副本必須在所有變更復制完成時收斂至一個相同的最終值。

實現衝突合併解決有多種途徑：

- 紿每個寫入一個唯一的 ID（例如時間戳、長隨機數、UUID 或者鍵和值的雜湊），挑選最高 ID 的寫入作為勝利者，並丟棄其他寫入。如果使用時間戳，這種技術被稱為 **最後寫入勝利 (LWW, last write wins)**。雖然這種方法很流行，但是很容易造成資料丟失【35】。我們將在本章末尾的 [檢測併發寫入](#) 一節更詳細地討論 LWW。
- 為每個副本分配一個唯一的 ID，ID 編號更高的寫入具有更高的優先順序。這種方法也意味著資料丟失。
- 以某種方式將這些值合併在一起 - 例如，按字母順序排序，然後連線它們（在 [圖 5-7](#) 中，合併的標題可能類似於“B/C”）。
- 用一種可保留所有資訊的顯式資料結構來記錄衝突，並編寫解決衝突的應用程式程式碼（也許透過提示使用者的方式）。

自定義衝突解決邏輯

解決衝突的最合適的方法可能取決於應用程式，大多數多主複製工具允許使用應用程式程式碼編寫衝突解決邏輯。該程式碼可以在寫入或讀取時執行：

- 寫時執行

只要資料庫系統檢測到複製更改日誌中存在衝突，就會呼叫衝突處理程式。例如，Bucardo 允許你為此編寫一段 Perl 程式碼。這個處理程式通常不能提示使用者——它在後臺程序中執行，並且必須快速執行。

- 讀時執行

當檢測到衝突時，所有衝突寫入被儲存。下一次讀取資料時，會將這些多個版本的資料返回給應用程式。應用程式可以提示使用者或自動解決衝突，並將結果寫回資料庫。例如 CouchDB 就以這種方式工作。

請注意，衝突解決通常適用於單行記錄或單個文件的層面，而不是整個事務【36】。因此，如果你有一個事務會原子性地進行幾次不同的寫入（請參閱 [第七章](#)），對於衝突解決而言，每個寫入仍需分開單獨考慮。

自動衝突解決

衝突解決規則可能很容易變得越來越複雜，自定義程式碼可能也很容易出錯。亞馬遜是一個經常被引用的例子，由於衝突解決處理程式而產生了令人意外的效果：一段時間以來，購物車上的衝突解決邏輯將保留新增到購物車的物品，但不包括從購物車中移除的物品。因此，顧客有時會看到物品重新出現在他們的購物車中，即使他們之前已經被移走【37】。

已經有一些有趣的研究來自動解決由於資料修改引起的衝突。有幾項研究值得一提：

- 無衝突複製資料型別（Conflict-free replicated datatypes，CRDT）【32,38】是可以由多個使用者同時編輯的集合、對映、有序列表、計數器等一系列資料結構，它們以合理的方式自動解決衝突。一些 CRDT 已經在 Riak 2.0 中實現【39,40】。
- 可合併的持久資料結構（Mergeable persistent data structures）【41】顯式跟蹤歷史記錄，類似於 Git 版本控制系統，並使用三向合併功能（而 CRDT 使用雙向合併）。
- 操作轉換（operational transformation）【42】是 Etherpad【30】和 Google Docs【31】等協同編輯應用背後的衝突解決演算法。它是專為有序列表的併發編輯而設計的，例如構成文字文件的字元列表。

這些演算法在資料庫中的實現還很年輕，但很可能將來它們會被整合到更多的複製資料系統中。自動衝突解決方案可以使應用程式處理多主資料同步更為簡單。

什麼是衝突？

有些衝突是顯而易見的。在 [圖 5-7](#) 的例子中，兩個寫操作併發地修改了同一條記錄中的同一個欄位，並將其設定為兩個不同的值。毫無疑問這是一個衝突。

其他型別的衝突可能更為微妙而難以發現。例如，考慮一個會議室預訂系統：它記錄誰訂了哪個時間段的哪個房間。應用程式需要確保每個房間在任意時刻都只能被一組人進行預訂（即不得有相同房間的重疊預訂）。在這種情況下，如果為同一個房間同時建立兩個不同的預訂，則可能會發生衝突。即使應用程式在允許使用者進行預訂之前先檢查會議室的可用性，如果兩次預訂是由兩個不同的主庫進行的，則仍然可能會有衝突。

雖然現在還沒有一個現成的答案，但在接下來的章節中，我們將更好地瞭解這個問題。我們將在 [第七章](#) 中看到更多的衝突示例，在 [第十二章](#) 中我們將討論用於檢測和解決複製系統中衝突的可伸縮方法。

多主複製拓撲

複製拓撲（replication topology）用來描述寫入操作從一個節點傳播到另一個節點的通訊路徑。如果你有兩個主庫，如 [圖 5-7](#) 所示，只有一個合理的拓撲結構：主庫 1 必須把它所有的寫入都發送到主庫 2，反之亦然。當有兩個以上的主庫，多種不同的拓撲都是可能的。[圖 5-8](#) 舉例說明了一些例子。

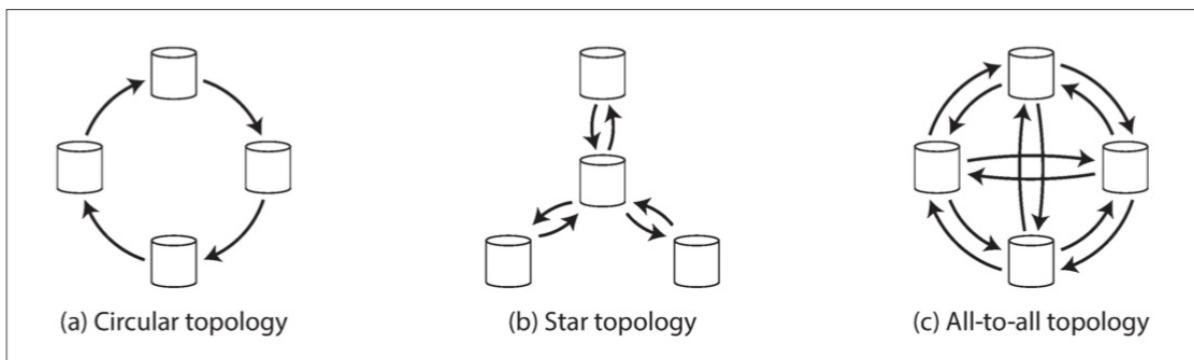


圖 5-8 三種可以在多主複製中使用的拓撲示例。

最常見的拓撲是全部到全部（all-to-all，如 [圖 5-8 \(c\)](#)），其中每個主庫都將其寫入傳送給其他所有的主庫。然而，一些更受限的拓撲也會被使用到：例如，預設情況下 MySQL 僅支援 **環形拓撲**（circular topology）【34】，其中每個節點都從一個節點接收寫入，並將這些寫入（加上自己的寫入）轉發給另一個節點。另一種流行的拓撲結構具有星形的形

狀^V：一個指定的根節點將寫入轉發給所有其他節點。星形拓撲可以推廣到樹。

▼ 不要與星型模式混淆（請參閱“[星型和雪花型：分析的模式](#)”），其中描述了資料模型的結構，而不是節點之間的通訊拓撲。 ↪

在環形和星形拓撲中，寫入可能需要在到達所有副本之前透過多個節點。因此，節點需要轉發從其他節點收到的資料更改。為了防止無限複製迴圈，每個節點被賦予一個唯一的識別符號，並且在複製日誌中，每次寫入都會使用其經過的所有節點的識別符號進行標記【43】。當一個節點收到用自己的識別符號標記的資料更改時，該資料更改將被忽略，因為節點知道它已經被處理過。

環形和星形拓撲的問題是，如果只有一個節點發生故障，則可能會中斷其他節點之間的複製訊息流，導致它們無法通訊，除非節點被修復。拓撲結構可以重新配置為跳過發生故障的節點，但在大多數部署中，這種重新配置必須手動完成。更密集連線的拓撲結構（例如全部到全部）的容錯性更好，因為它允許訊息沿著不同的路徑傳播，可以避免單點故障。

另一方面，全部到全部的拓撲也可能有問題。特別是，一些網路連結可能比其他網路連結更快（例如由於網路擁塞），結果是一些複製訊息可能“超越”其他複製訊息，如 [圖 5-9](#) 所示。

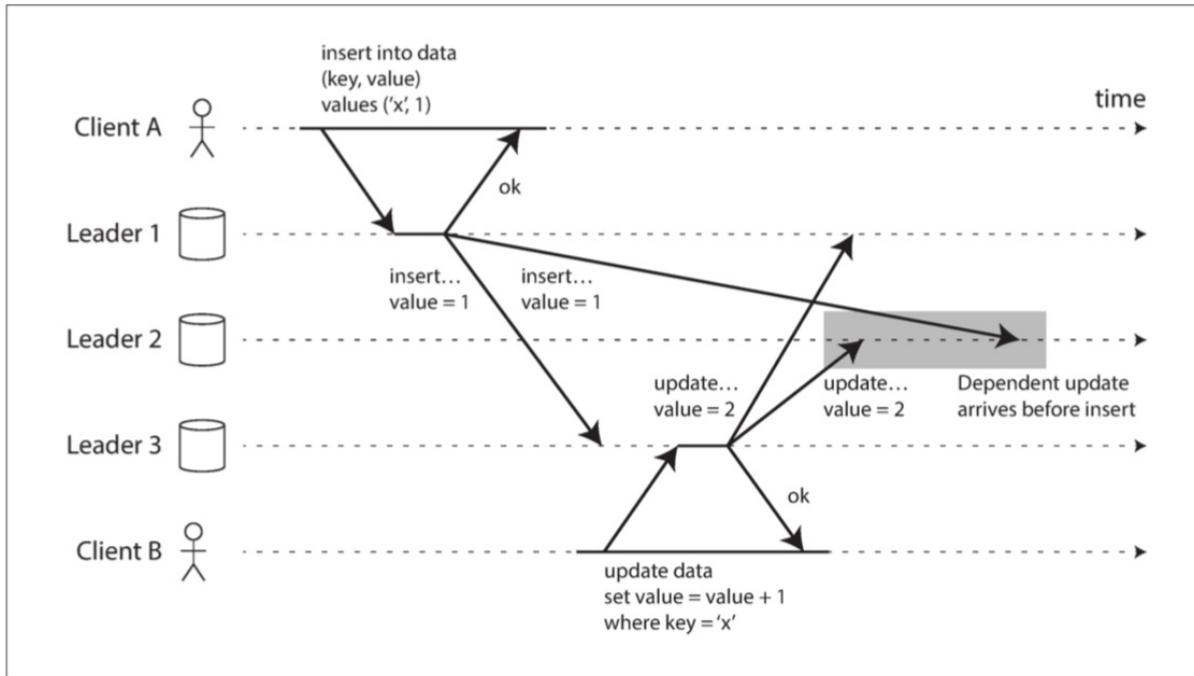


圖 5-9 使用多主複製時，寫入可能會以錯誤的順序到達某些副本。

在 [圖 5-9](#) 中，客戶端 A 向主庫 1 的表中插入一行，客戶端 B 在主庫 3 上更新該行。然而，主庫 2 可以以不同的順序接收寫入：它可能先接收到更新（從它的角度來看，是對資料庫中不存在的行的更新），稍後才接收到相應的插入（其應該在更新之前）。

這是一個因果關係的問題，類似於我們在“[一致字首讀](#)”中看到的：更新取決於先前的插入，所以我們需要確保所有節點先處理插入，然後再處理更新。僅僅在每一次寫入時新增一個時間戳是不夠的，因為時鐘不可能被充分地同步，所以主庫 2 就無法正確地對這些事件進行排序（見 [第八章](#)）。

要正確排序這些事件，可以使用一種稱為 **版本向量** (**version vectors**) 的技術，本章稍後將討論這種技術（請參閱“[檢測併發寫入](#)”）。然而，許多主複製系統中的衝突檢測技術實現得並不好。例如，在撰寫本文時，PostgreSQL BDR 不提供寫入的因果排序【27】，而 Tungsten Replicator for MySQL 甚至都不做檢測衝突【34】。

如果你正在使用基於多主複製的系統，那麼你應該多瞭解這些問題，仔細閱讀文件，並徹底測試你的資料庫，以確保它確實提供了你想要的保證。

無主複製

我們在本章到目前為止所討論的複製方法——單主複製、多主複製——都是這樣的想法：客戶端向一個主庫傳送寫請求，而資料庫系統負責將寫入複製到其他副本。主庫決定寫入的順序，而從庫按相同順序應用主庫的寫入。

一些資料儲存系統採用不同的方法，放棄主庫的概念，並允許任何副本直接接受來自客戶端的寫入。最早的一些的複製資料系統是 **無主的（leaderless）** 【1,44】，但是在關係資料庫主導的時代，這個想法幾乎已被忘卻。在亞馬遜將其用於其內部的 Dynamo 系統^{vi} 之後，它再一次成為資料庫的一種時尚架構【37】。Riak, Cassandra 和 Voldemort 是受 Dynamo 啟發的無主複製模型的開源資料儲存，所以這類資料庫也被稱為 *Dynamo 風格*。

^{vi} Dynamo 不適用於 Amazon 以外的使用者。令人困惑的是，AWS 提供了一個名為 DynamoDB 的託管資料庫產品，它使用了完全不同的體系結構：它基於單主複製。 ↪

在一些無主複製的實現中，客戶端直接將寫入傳送到幾個副本中，而另一些情況下，由一個 **協調者（coordinator）** 節點代表客戶端進行寫入。但與主庫資料庫不同，協調者不執行特定的寫入順序。我們將會看到，這種設計上的差異對資料庫的使用方式有著深遠的影響。

當節點故障時寫入資料庫

假設你有一個帶有三個副本的資料庫，而其中一個副本目前不可用，或許正在重新啟動以安裝系統更新。在基於領導者的配置中，如果要繼續處理寫入，則可能需要執行故障切換（請參閱「[處理節點宕機](#)」）。

另一方面，在無主配置中，不存在故障轉移。圖 5-10 演示了會發生了什麼事情：客戶端（使用者 1234）並行傳送寫入到所有三個副本，並且兩個可用副本接受寫入，但是不可用副本錯過了它。假設三個副本中的兩個承認寫入是足夠的：在使用者 1234 已經收到兩個確定的響應之後，我們認為寫入成功。客戶簡單地忽略了其中一個副本錯過了寫入的事實。

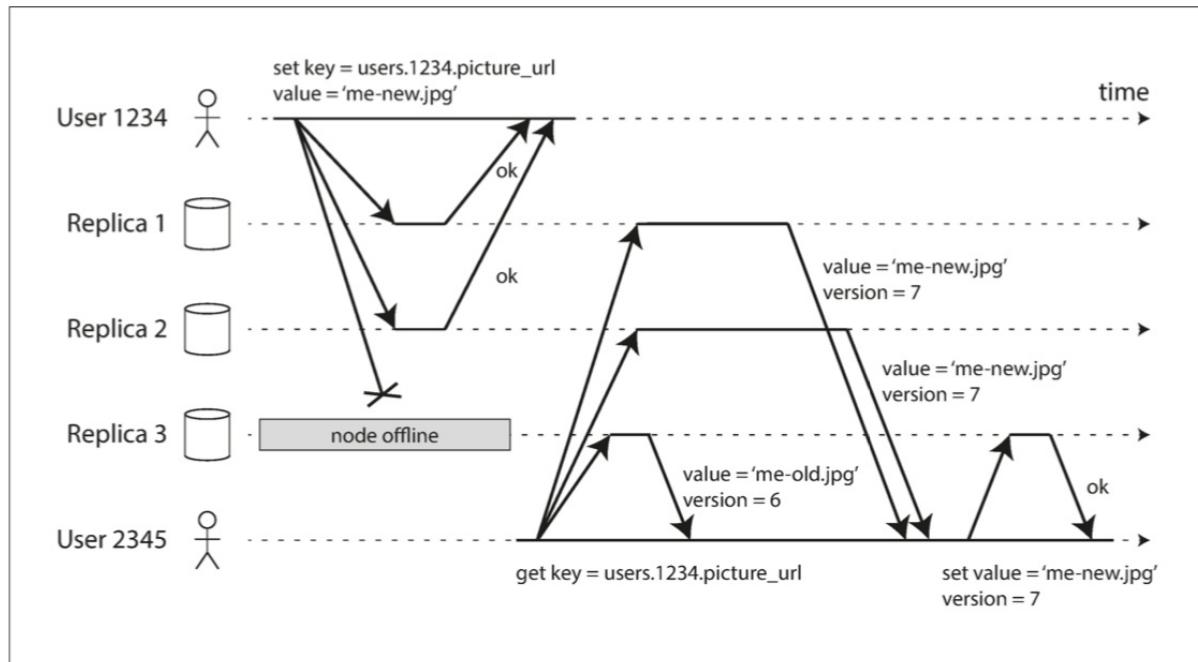


圖 5-10 法定寫入，法定讀取，並在節點中斷後讀修復。

現在想象一下，不可用的節點重新聯機，客戶端開始讀取它。節點關閉期間發生的任何寫入都不在該節點上。因此，如果你從該節點讀取資料，則可能會從響應中拿到陳舊的（過時的）值。

為了解決這個問題，當一個客戶端從資料庫中讀取資料時，它不僅僅把它的請求傳送到一個副本：讀請求將被並行地傳送到多個節點。客戶可能會從不同的節點獲得不同的響應，即來自一個節點的最新值和來自另一個節點的陳舊值。版本號將被用於確定哪個值是更新的（請參閱“[檢測併發寫入](#)”）。

讀修復和反熵

複製方案應確保最終將所有資料複製到每個副本。在一個不可用的節點重新聯機之後，它如何趕上它錯過的寫入？

在 Dynamo 風格的資料儲存中經常使用兩種機制：

- 讀修復 (Read repair)

當客戶端並行讀取多個節點時，它可以檢測到任何陳舊的響應。例如，在 [圖 5-10](#) 中，使用者 2345 獲得了來自副本 3 的版本 6 值和來自副本 1 和 2 的版本 7 值。客戶端發現副本 3 具有陳舊值，並將新值寫回到該副本。這種方法適用於讀頻繁的值。

- 反熵過程 (Anti-entropy process)

此外，一些資料儲存具有後臺程序，該程序不斷查詢副本之間的資料差異，並將任何缺少的資料從一個副本複製到另一個副本。與基於領導者的複製中的複製日誌不同，此反熵過程不會以任何特定的順序複製寫入，並且在複製資料之前可能會有顯著的延遲。

並不是所有的系統都實現了這兩種機制，例如，Voldemort 目前沒有反熵過程。請注意，如果沒有反熵過程，很少被讀取的值可能會從某些副本中丟失，從而降低了永續性，因為只有在應用程式讀取值時才執行讀修復。

讀寫的法定人數

在 [圖 5-10](#) 的示例中，我們認為即使僅在三個副本中的兩個上進行處理，寫入仍然是成功的。如果三個副本中只有一個接受了寫入，會怎樣？以此類推，究竟多少個副本完成才可以認為寫入成功？

如果我們知道，每個成功的寫操作意味著在三個副本中至少有兩個出現，這意味著至多有一個副本可能是陳舊的。因此，如果我們從至少兩個副本讀取，我們可以確定至少有一個是最新的。如果第三個副本停機或響應速度緩慢，則讀取仍可以繼續返回最新值。

更一般地說，如果有 n 個副本，每個寫入必須由 w 個節點確認才能被認為是成功的，並且我們必須至少為每個讀取查詢 r 個節點。（在我們的例子中， $n = 3$ ， $w = 2$ ， $r = 2$$ ）。只要 $w + r > n$$ ，我們可以預期在讀取時能獲得最新的值，因為 r 個讀取中至少有一個節點是最新的。遵循這些 r 值和 w 值的讀寫稱為 **法定人數 (quorum)**^{vii} 的讀和寫

[\[44\]](#)。你可以認為， r 和 w 是有效讀寫所需的最低票數。

^{vii}. 有時候這種法定人數被稱為嚴格的法定人數，其相對“寬鬆的法定人數”而言（見“[寬鬆的法定人數與提示移交](#)”）[←](#)

在 Dynamo 風格的資料庫中，引數 n 、 w 和 r 通常是可配置的。一個常見的選擇是使 n 為奇數（通常為 3 或 5）並設定 $w = r = (n + 1) / 2$$ （向上取整）。但是你可以根據需要更改數字。例如，寫入次數較少且讀取次數較多的工作負載可以從設定 $w = n$$ 和 $r = 1$$ 中受益。這會使得讀取速度更快，但缺點是隻要有一個不可用的節點就會導致所有的資料庫寫入都失敗。

叢集中可能有多於 n 個的節點（叢集的機器數可能多於副本數目）。但是任何給定的值只能儲存在 n 個節點上。這允許對資料集進行分割槽，從而可以支援比單個節點的儲存能力更大的資料集。我們將在 [第六章](#) 繼續討論分割槽。

法定人數條件 $w + r > n$$ 允許系統容忍不可用的節點，如下所示：

- 如果 $w < n$$ ，當節點不可用時，我們仍然可以處理寫入。
- 如果 $r < n$$ ，當節點不可用時，我們仍然可以處理讀取。
- 對於 $n = 3$ ， $w = 2$ ， $r = 2$$ ，我們可以容忍一個不可用的節點。
- 對於 $n = 5$ ， $w = 3$ ， $r = 3$$ ，我們可以容忍兩個不可用的節點。這個案例如 [圖 5-11](#) 所示。
- 通常，讀取和寫入操作始終並行傳送到所有 n 個副本。引數 w 和 r 決定我們等待多少個節點，即在我們認為讀或寫成功之前，有多少個節點需要報告成功。

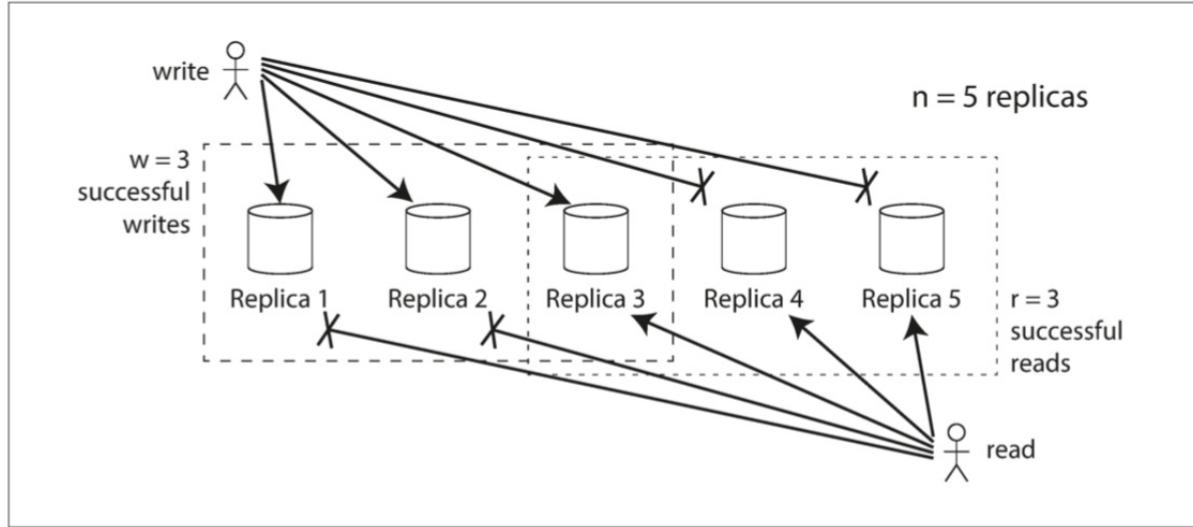


圖 5-11 如果 $w + r > n$ ，讀取 r 個副本，至少有一個副本必然包含了最近的成功寫入。

如果可用的節點少於所需的 w 或 r ，則寫入或讀取將返回錯誤。節點可能由於多種原因而不可用，比如：節點關閉（異常崩潰，電源關閉）、操作執行過程中的錯誤（由於磁碟已滿而無法寫入）、客戶端和伺服器節點之間的網路中斷或任何其他原因。我們只需要關心節點是否返回了成功的響應，而不需要區分不同型別的錯誤。

法定人數一致性的侷限性

如果你有 n 個副本，並且你選擇了滿足 $w + r > n$ 的 w 和 r ，你通常可以期望每次讀取都能返回最近寫入的值。情況就是這樣，因為你寫入的節點集合和你讀取的節點集合必然有重疊。也就是說，你讀取的節點中必然至少有一個節點具有最新值（如 圖 5-11 所示）。

通常， r 和 w 被選為多數（超過 $n/2$ ）節點，因為這確保了 $w + r > n$ ，同時仍然容忍多達 $n/2$ 個節點故障。但是，法定人數不一定必須是大多數，重要的是讀寫使用的節點至少有一個節點的交集。其他法定人數的配置是可能的，這使得分散式演算法的設計有一定的靈活性【45】。

你也可以將 w 和 r 設定為較小的數字，以使 $w + r \leq n$ （即法定條件不滿足）。在這種情況下，讀取和寫入操作仍將被傳送到 n 個節點，但操作成功只需要少量的成功響應。

較小的 w 和 r 更有可能會讀取到陳舊的資料，因為你的讀取更有可能未包含具有最新值的節點。另一方面，這種配置允許更低的延遲和更高的可用性：如果存在網路中斷，並且許多副本變得無法訪問，則有更大的機會可以繼續處理讀取和寫入。只有當可達副本的數量低於 w 或 r 時，資料庫才變得不可寫入或讀取。

但是，即使在 $w + r > n$ 的情況下，也可能存在返回陳舊值的邊緣情況。這取決於實現，但可能的情況包括：

- 如果使用寬鬆的法定人數（見“寬鬆的法定人數與提示移交”）， w 個寫入和 r 個讀取有可能落在完全不同的節點上，因此 r 節點和 w 之間不再保證有重疊節點【46】。
- 如果兩個寫入同時發生，不清楚哪一個先發生。在這種情況下，唯一安全的解決方案是合併併發寫入（請參閱“處理寫入衝突”）。如果根據時間戳（最後寫入勝利）挑選出一個勝者，則寫入可能由於時鐘偏差【35】而丟失。我們將在“檢測併發寫入”繼續討論此話題。
- 如果寫操作與讀操作同時發生，寫操作可能僅反映在某些副本上。在這種情況下，不確定讀取返回的是舊值還是新值。
- 如果寫操作在某些副本上成功，而在其他節點上失敗（例如，因為某些節點上的磁碟已滿），在小於 w 個副本上寫入成功。所以整體判定寫入失敗，但整體寫入失敗並沒有在寫入成功的副本上回滾。這意味著一個寫入雖然報告失敗，後續的讀取仍然可能會讀取這次失敗寫入的值【47】。
- 如果攜帶新值的節點發生故障，需要從其他帶有舊值的副本進行恢復，則儲存新值的副本數可能會低於 w ，從而打破法定人數條件。
- 即使一切工作正常，有時也會不幸地出現關於 時序 (timing) 的邊緣情況，我們將在“線性一致性和法定人數”中看到這點。

因此，儘管法定人數似乎保證讀取返回最新的寫入值，但在實踐中並不那麼簡單。Dynamo 風格的資料庫通常針對可以忍受最終一致性的用例進行最佳化。你可以透過引數 w 和 r 來調整讀取到陳舊值的機率，但把它們當成絕對的保證是不明智的。

尤其是，因為通常得不到“[複製延遲問題](#)”中討論的那些保證（讀己之寫，單調讀，一致字首讀），前面提到的異常可能會發生在應用程式中。更強有力的保證通常需要 [事務](#) 或 [共識](#)。我們將在 [第七章](#) 和 [第九章](#) 回到這些話題。

監控陳舊度

從運維的角度來看，監視你的資料庫是否返回最新的結果是很重要的。即使應用可以容忍陳舊的讀取，你也需要了解複製的健康狀況。如果顯著落後，它應該提醒你以便你可以調查原因（例如網路中的問題或過載的節點）。

對於基於領導者的複製，資料庫通常會提供複製延遲的測量值，你可以將其提供給監視系統。這之所以能做到，是因為寫入是按照相同的順序應用於主庫和從庫，並且每個節點對應了複製日誌中的一個位置（已經在本地應用的寫入數量）。透過從主庫的當前位置中減去從庫的當前位置，你可以測量複製延遲的程度。

然而，在無主複製的系統中，沒有固定的寫入順序，這使得監控變得更加困難。而且，如果資料庫只使用讀修復（沒有反熵過程），那麼對於一個值可能會有多陳舊其實是沒有限制的 - 如果一個值很少被讀取，那麼由一個陳舊副本返回的值可能是古老的。

已經有一些關於衡量無主複製資料庫中的複製陳舊度的研究，並根據引數 n 、 w 和 r 來預測陳舊讀取的預期百分比【48】。不幸的是，這還不是很常見的做法，但是將陳舊測量值包含在資料庫的標準度量集中是一件好事。雖然最終一致性是一種有意模糊的保證，但是從可操作性角度來說，能夠量化“最終”也是很重要的。

寬鬆的法定人數與提示移交

合理配置的法定人數可以使資料庫無需故障切換即可容忍個別節點的故障。它也可以容忍個別節點變慢，因為請求不必等待所有 n 個節點響應——當 w 或 r 個節點響應時它們就可以返回。對於需要高可用、低延時、且能夠容忍偶爾讀到陳舊值的應用場景來說，這些特性使無主複製的資料庫很有吸引力。

然而，法定人數（如迄今為止所描述的）並不像它們可能的那樣具有容錯性。網路中斷可以很容易地將客戶端從大量的資料庫節點上切斷。雖然這些節點是活著的，而其他客戶端可能也能夠連線到它們，但是從資料庫節點切斷的客戶端來看，它們也可能已經死亡。在這種情況下，剩餘的可用節點可能會少於 w 或 r ，因此客戶端不再能達到法定人數。

在一個大型的叢集中（節點數量明顯多於 n 個），網路中斷期間客戶端可能仍能連線到一些資料庫節點，但又不足以組成一個特定的法定人數。在這種情況下，資料庫設計人員需要權衡一下：

- 對於所有無法達到 w 或 r 個節點法定人數的請求，是否返回錯誤是更好的？
- 或者我們是否應該接受寫入，然後將它們寫入一些可達的節點，但不在這些值通常所存在的 n 個節點上？

後者被認為是一個 **寬鬆的法定人數 (sloppy quorum)** 【37】：寫和讀仍然需要 w 和 r 個成功的響應，但這些響應可能來自不在指定的 n 個“主”節點中的其它節點。就好比說，如果你把自己鎖在房子外面了，你可能會去敲開鄰居的門，問是否可以暫時呆在他們的沙發上。

一旦網路中斷得到解決，一個節點代表另一個節點臨時接受的任何寫入都將被傳送到適當的“主”節點。這就是所謂的 **提示移交 (hinted handoff)**（一旦你再次找到你的房子的鑰匙，你的鄰居可以禮貌地要求你離開沙發回家）。

寬鬆的法定人數對寫入可用性的提高特別有用：只要有任何 w 個節點可用，資料庫就可以接受寫入。然而，這意味著即使當 $w + r > n$ 時，也不能確保讀取到某個鍵的最新值，因為最新的值可能已經臨時寫入了 n 之外的某些節點【47】。

因此，在傳統意義上，寬鬆的法定人數實際上並不是法定人數。它只是一個永續性的保證，即資料已儲存在某處的 w 個節點。但不能保證 r 個節點的讀取能看到它，除非提示移交已經完成。

在所有常見的 Dynamo 實現中，寬鬆的法定人數是可選的。在 Riak 中，它們預設是啟用的，而在 Cassandra 和 Voldemort 中它們預設是禁用的【46,49,50】。

運維多個數據中心

我們先前討論了跨資料中心複製，作為多主複製的用例（請參閱“[多主複製](#)”）。其實無主複製也適用於多資料中心操作，既然它旨在容忍衝突的併發寫入、網路中斷和延遲尖峰。

Cassandra 和 Voldemort 在正常的無主模型中實現了他們的多資料中心支援：副本的數量 n 包括所有資料中心的節點，你可以在配置中指定每個資料中心所擁有的副本的數量。無論資料中心如何，每個來自客戶端的寫入都會發送到所有副本，但客戶端通常只等待來自其本地資料中心內的法定節點的確認，從而不會受到跨資料中心鏈路延遲和中斷的影響。對其他資料中心的高延遲寫入通常被配置為非同步執行，儘管該配置仍有一定的靈活性【50,51】。

Riak 將客戶端和資料庫節點之間的所有通訊保持在一個本地的資料中心，因此 n 描述了一個數據中心內的副本數量。資料庫叢集之間的跨資料中心複製在後臺非同步發生，其風格類似於多主複製【52】。

檢測併發寫入

Dynamo 風格的資料庫允許多個客戶端同時寫入相同的鍵（Key），這意味著即使使用嚴格的法定人數也會發生衝突。這種情況與多主複製相似（請參閱“[處理寫入衝突](#)”），但在 Dynamo 風格的資料庫中，在 [讀修復](#) 或 [提示移交](#) 期間也可能會產生衝突。

其問題在於，由於可變的網路延遲和部分節點的故障，事件可能以不同的順序到達不同的節點。例如，[圖 5-12](#) 顯示了兩個客戶機 A 和 B 同時寫入三節點資料儲存中的鍵 X：

- 節點 1 接收來自 A 的寫入，但由於暫時中斷，未接收到來自 B 的寫入。
- 節點 2 首先接收來自 A 的寫入，然後接收來自 B 的寫入。
- 節點 3 首先接收來自 B 的寫入，然後從 A 寫入。

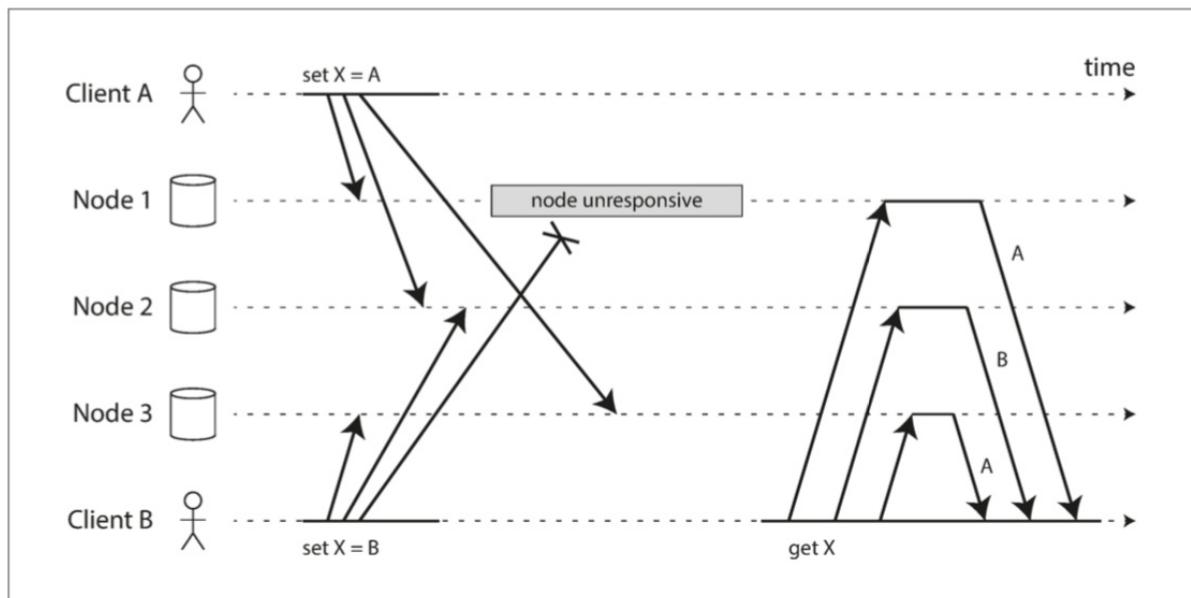


圖 5-12 併發寫入 Dynamo 風格的資料儲存：沒有明確定義的順序。

如果每個節點只要接收到來自客戶端的寫入請求就簡單地覆寫某個鍵值，那麼節點就會永久地不一致，如 [圖 5-12](#) 中的最終獲取請求所示：節點 2 認為 X 的最終值是 B，而其他節點認為值是 A。

為了最終達成一致，副本應該趨於相同的值。如何做到這一點？有人可能希望複製的資料庫能夠自動處理，但不幸的是，大多數的實現都很糟糕：如果你想避免丟失資料，你（應用程式開發人員）需要知道很多有關資料庫衝突處理的內部資訊。

在“[處理寫入衝突](#)”一節中已經簡要介紹了一些解決衝突的技術。在總結本章之前，讓我們來更詳細地探討這個問題。

最後寫入勝利（丟棄併發寫入）

實現最終收斂的一種方法是宣告每個副本只需要儲存“最近”的值，並允許“更舊”的值被覆蓋和拋棄。然後，只要我們有一種明確的方式來確定哪個寫是“最近的”，並且每個寫入最終都被複制到每個副本，那麼複製最終會收斂到相同的值。

正如“最近”的引號所表明的，這個想法其實頗具誤導性。在 [圖 5-12](#) 的例子中，當客戶端向資料庫節點發送寫入請求時，兩個客戶端都不知道另一個客戶端，因此不清楚哪一個先發送請求。事實上，說這兩種情況誰先發送請求是沒有意義的：既然我們說寫入是 **併發 (concurrent)** 的，那麼它們的順序就是不確定的。

即使寫入沒有自然的排序，我們也可以強制進行排序。例如，可以為每個寫入附加一個時間戳，然後挑選最大的時間戳作為“最近的”，並丟棄具有較早時間戳的任何寫入。這種衝突解決演算法被稱為 **最後寫入勝利 (LWW, last write wins)**，是 Cassandra 唯一支援的衝突解決方法 [\[53\]](#)，也是 Riak 中的一個可選特徵 [\[35\]](#)。

LWW 實現了最終收斂的目標，但以 **永續性** 為代價：如果同一個鍵有多個併發寫入，即使它們反饋給客戶端的結果都是成功的（因為它們被寫入 w 個副本），也只有一個寫入將被保留，而其他寫入將被默默地丟棄。此外，LWW 甚至可能會丟棄不是併發的寫入，我們將在“[有序事件的時間戳](#)”中進行討論。

在類似快取的一些情況下，寫入丟失可能是可以接受的。但如果資料丟失不可接受，LWW 是解決衝突的一個很爛的選擇。

在資料庫中使用 LWW 的唯一安全方法是確保一個鍵只寫入一次，然後視為不可變，從而避免對同一個鍵進行併發更新。例如，Cassandra 推薦使用的方法是使用 UUID 作為鍵，從而為每個寫操作提供一個唯一的鍵 [\[53\]](#)。

“此前發生”的關係和併發

我們如何判斷兩個操作是否是併發的？為了建立一個直覺，讓我們看看一些例子：

- 在 [圖 5-9](#) 中，兩個寫入不是併發的：A 的插入發生在 B 的遞增之前，因為 B 遞增的值是 A 插入的值。換句話說，B 的操作建立在 A 的操作上，所以 B 的操作必須後發生。我們也可以說 B 因果依賴 (**causally dependent**) 於 A。
- 另一方面，[圖 5-12](#) 中的兩個寫入是併發的：當每個客戶端啟動操作時，它不知道另一個客戶端也正在對同樣的鍵執行操作。因此，操作之間不存在因果關係。

如果操作 B 瞭解操作 A，或者依賴於 A，或者以某種方式構建於操作 A 之上，則操作 A 在操作 B 之前發生 (**happens before**)。一個操作是否在另一個操作之前發生是定義併發含義的關鍵。事實上，我們可以簡單地說，如果兩個操作中的任何一個都不在另一個之前發生（即，兩個操作都不知道對方），那麼這兩個操作是併發的 [\[54\]](#)。

因此，只要有兩個操作 A 和 B，就有三種可能性：A 在 B 之前發生，或者 B 在 A 之前發生，或者 A 和 B 併發。我們需要的是一個演算法來告訴我們兩個操作是否是併發的。如果一個操作發生在另一個操作之前，則後面的操作應該覆蓋前面的操作，但是如果這些操作是併發的，則存在需要解決的衝突。

併發性、時間和相對性

如果兩個操作“同時”發生，似乎應該稱為併發——但事實上，它們在字面時間上重疊與否並不重要。由於分散式系統中的時鐘問題，現實中是很難判斷兩個事件是否是 同時 發生的，這個問題我們將在 [第八章](#) 中詳細討論。

為了定義併發性，確切的時間並不重要：如果兩個操作都意識不到對方的存在，就稱這兩個操作 **併發**，而不管它們實際發生的物理時間。人們有時把這個原理和物理學中的狹義相對論聯絡起來 [\[54\]](#)，該理論引入了資訊不能比光速更快的思想。因此，如果兩個事件發生的時間差小於光透過它們之間的距離所需要的時間，那麼這兩個事件不可能相互影響。

在計算機系統中，即使光速原則上允許一個操作影響另一個操作，但兩個操作也可能是 併發的。例如，如果網路緩慢或中斷，兩個操作間可能會出現一段時間間隔，但仍然是併發的，因為網路問題阻止一個操作意識到另一個操作的存在。

捕獲“此前發生”關係

我們來看一個演算法，它可以確定兩個操作是否為併發的，還是一個在另一個之前。簡單起見，我們從一個只有一個副本的資料庫開始。一旦我們知道了如何在單個副本上完成這項工作，我們可以將該方法推廣到具有多個副本的無主資料庫。

圖 5-13 顯示了兩個客戶端同時向同一購物車新增專案。（如果這樣的例子讓你覺得無趣，那麼可以想象一下兩個空中交通管制員同時把飛機新增到他們正在跟蹤的區域。）最初，購物車是空的。然後客戶端向資料庫發出五次寫入：

1. 客戶端 1 將牛奶加入購物車。這是該鍵的第一次寫入，伺服器成功儲存了它併為其分配版本號 1，最後將值與版本號一起回送給客戶端。
2. 客戶端 2 將雞蛋加入購物車，不知道客戶端 1 同時添加了牛奶（客戶端 2 認為它的雞蛋是購物車中的唯一物品）。伺服器為此寫入分配版本號 2，並將雞蛋和牛奶儲存為兩個單獨的值。然後它將這兩個值都返回給客戶端 2，並附上版本號 2。
3. 客戶端 1 不知道客戶端 2 的寫入，想要將麵粉加入購物車，因此認為當前的購物車內容應該是 [牛奶, 麵粉]。它將此值與伺服器先前向客戶端 1 提供的版本號 1 一起傳送到伺服器。伺服器可以從版本號中知道 [牛奶, 麵粉] 的寫入取代了 [牛奶] 的先前值，但與 [雞蛋] 的值是 併發 的。因此，伺服器將版本號 3 分配給 [牛奶, 麵粉]，覆蓋版本 1 的值 [牛奶]，但保留版本 2 的值 [蛋]，並將所有的值返回給客戶端 1。
4. 同時，客戶端 2 想要加入火腿，不知道客戶端 1 剛剛加了麵粉。客戶端 2 在最近一次響應中從伺服器收到了兩個值 [牛奶] 和 [蛋]，所以客戶端 2 現在合併這些值，並新增火腿形成一個新的值 [雞蛋, 牛奶, 火腿]。它將這個值傳送到伺服器，帶著之前的版本號 2。伺服器檢測到新值會覆蓋版本 2 的值 [雞蛋]，但新值也會與版本 3 的值 [牛奶, 麵粉] 併發，所以剩下的兩個值是版本 3 的 [牛奶, 麵粉]，和版本 4 的 [雞蛋, 牛奶, 火腿]。
5. 最後，客戶端 1 想要加培根。它之前從伺服器接收到了版本 3 的 [牛奶, 麵粉] 和 [雞蛋]，所以它合併這些，新增培根，並將最終值 [牛奶, 麵粉, 雞蛋, 培根] 連同版本號 3 發往伺服器。這會覆蓋版本 3 的值 [牛奶, 麵粉]（請注意 [雞蛋] 已經在上一步被覆蓋），但與版本 4 的值 [雞蛋, 牛奶, 火腿] 併發，所以伺服器將保留這兩個併發值。

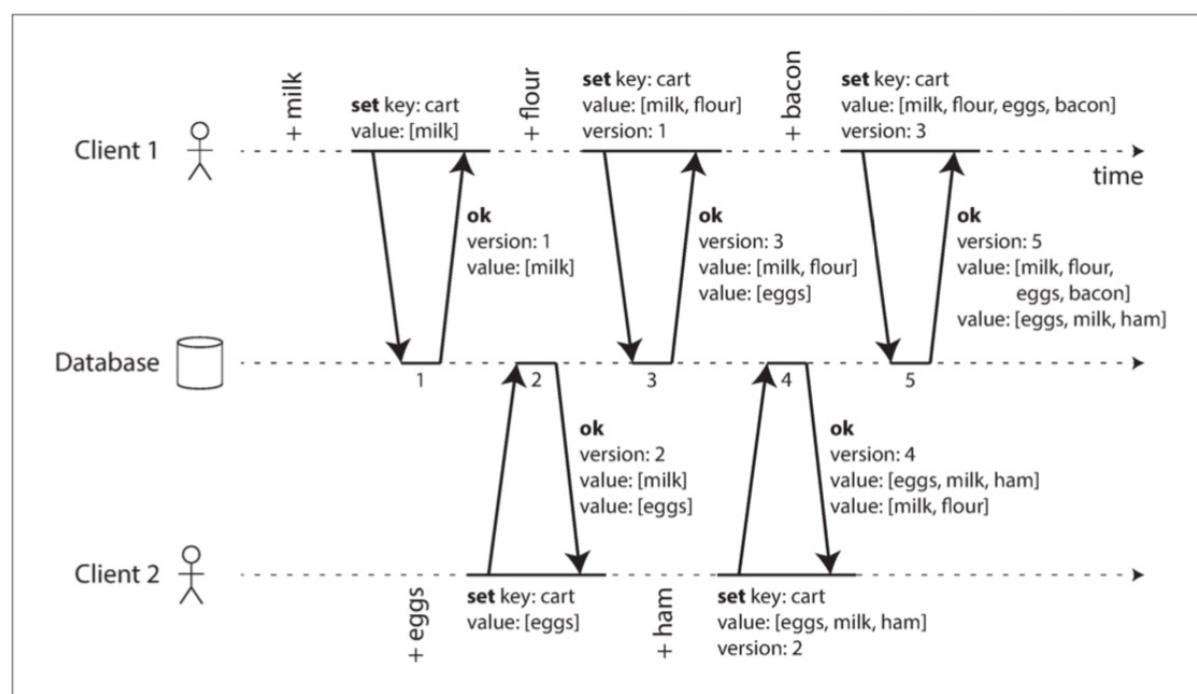


圖 5-13 在同時編輯購物車時捕獲兩個客戶端之間的因果關係。

圖 5-13 中的操作之間的資料流如 圖 5-14 所示。箭頭表示哪個操作發生在其他操作之前，意味著後面的操作知道或依賴於較早的操作。在這個例子中，客戶端永遠不會完全拿到伺服器上的最新資料，因為總是有另一個操作同時進行。但是舊版本的值最終會被覆蓋，並且不會丟失任何寫入。

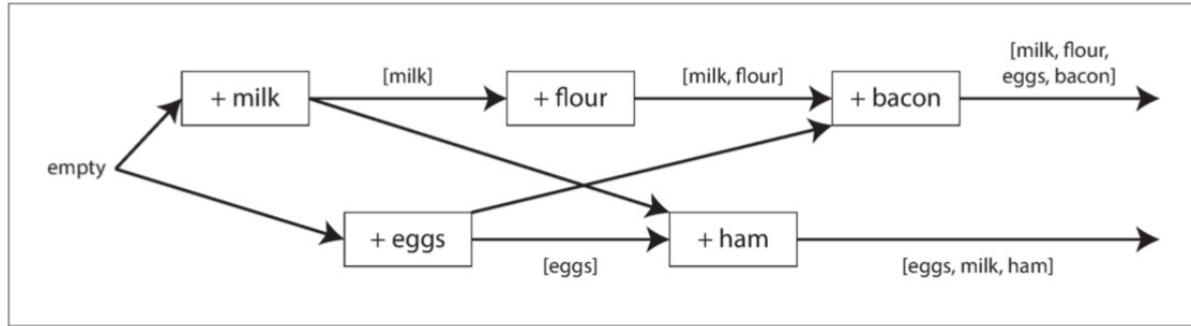


圖 5-14 圖 5-13 中的因果依賴關係圖。

請注意，伺服器可以只通過檢視版本號來確定兩個操作是否是併發的——它不需要對值本身進行解釋（因此該值可以是任何資料結構）。該演算法的工作原理如下：

- 伺服器為每個鍵維護一個版本號，每次寫入該鍵時都遞增版本號，並將新版本號與寫入的值一起儲存。
- 當客戶端讀取鍵時，伺服器將返回所有未覆蓋的值以及最新的版本號。客戶端在寫入前必須先讀取。
- 當客戶端寫入鍵時，必須包含之前讀取的版本號，並且必須將之前讀取的所有值合併在一起（針對寫入請求的響應可以像讀取請求一樣，返回所有當前值，這使得我們可以像購物車示例那樣將多個寫入串聯起來）。
- 當伺服器接收到具有特定版本號的寫入時，它可以覆蓋該版本號或更低版本的所有值（因為它知道它們已經被合併到新的值中），但是它必須用更高的版本號來儲存所有值（因為這些值與正在進行的其它寫入是併發的）。

當一個寫入包含前一次讀取的版本號時，它會告訴我們的寫入是基於之前的哪一種狀態。如果在不包含版本號的情況下進行寫操作，則與所有其他寫操作併發，因此它不會覆蓋任何內容——只會在隨後的讀取中作為其中一個值返回。

合併併發寫入的值

這種演算法可以確保沒有資料被無聲地丟棄，但不幸的是，客戶端需要做一些額外的工作：客戶端隨後必須合併併發寫入的值。Riak 稱這些併發值為 **兄弟 (siblings)**。

合併併發值，本質上是與多主複製中的衝突解決問題相同，我們先前討論過（請參閱“[處理寫入衝突](#)”）。一個簡單的方法是根據版本號或時間戳（最後寫入勝利）來選擇一個值，但這意味著丟失資料。所以，你可能需要在應用程式程式碼中額外做些更聰明的事情。

以購物車為例，一種合理的合併值的方法就是做並集。在 [圖 5-14](#) 中，最後的兩個兄弟是 [牛奶，麵粉，雞蛋，燻肉] 和 [雞蛋，牛奶，火腿]。注意牛奶和雞蛋雖然同時出現在兩個併發值裡，但他們每個只被寫過一次。合併的值可以是 [牛奶，麵粉，雞蛋，培根，火腿]，不再有重複了。

然而，如果你想讓人們也可以從他們的購物車中 **移除** 東西，而不是僅僅新增東西，那麼把併發值做並集可能不會產生正確的結果：如果你合併了兩個客戶端的購物車，並且只在其中一個客戶端裡面移除了一個專案，那麼被移除的專案將會重新出現在這兩個客戶端的交集結果中【37】。為了防止這個問題，要移除一個專案時不能簡單地直接從資料庫中刪除；相反，系統必須留下一個具有適當版本號的標記，以在兄弟合併時表明該專案已被移除。這種刪除標記被稱為 **墓碑 (tombstone)**（我們上一次看到墓碑是在“[雜湊索引](#)”章節的日誌壓縮部分）。

因為在應用程式程式碼中做兄弟合併是複雜且容易出錯的，所以有一些資料結構被設計出來用於自動執行這種合併，比如在“[自動衝突解決](#)”中討論過的那些。舉例來說，Riak 的資料型別就支援使用稱為 CRDT 【38,39,55】的能以合理方式自動進行兄弟合併的資料結構家族，包括對保留刪除的支援。

版本向量

[圖 5-13](#) 中的示例只使用了一個副本。當有多個副本但又沒有主庫時，演算法該如何修改？

[圖 5-13](#) 使用單個版本號來捕獲操作之間的依賴關係，但是當多個副本併發接受寫入時，這是不夠的。相反，除了對每個鍵，我們還需要對 **每個副本** 使用版本號。每個副本在處理寫入時增加自己的版本號，並且跟蹤從其他副本中看到的版本號。這個資訊指出了要覆蓋哪些併發值，以及要保留哪些併發值或兄弟值。

所有副本的版本號集合稱為 **版本向量** (**version vector**) 【56】。這個想法的一些變體正在被使用，但最有趣的可能是在 Riak 2.0 【58,59】中使用的 **虛線版本向量** (**dotted version vector**) 【57】。我們不會深入細節，但是它的工作方式與我們在購物車示例中看到的非常相似。

與 圖 5-13 中的版本號一樣，當讀取值時，版本向量會從資料庫副本傳送到客戶端，並且隨後寫入值時需要將其傳送回資料庫。（Riak 將版本向量編碼為一個字串，並稱其為 **因果上下文**，即 causal context）。版本向量允許資料庫區分覆蓋寫入和併發寫入。

另外，就像在單個副本中的情況一樣，應用程式可能需要合併併發值。版本向量結構能夠確保從一個副本讀取並隨後寫回到另一個副本是安全的。這樣做雖然可能會在其他副本上面建立資料，但只要能正確合併就不會丟失資料。

版本向量和向量時鐘

版本向量有時也被稱為向量時鐘，即使它們不完全相同。其中的差別很微妙——細節請參閱參考資料【57,60,61】。簡而言之，在比較副本的狀態時，版本向量才是正確的資料結構。

本章小結

在本章中，我們考察了複製的問題。複製可以用於幾個目的：

- **高可用性**

即使在一臺機器（或多臺機器，或整個資料中心）停機的情況下也能保持系統正常執行

- **斷開連線的操作**

允許應用程式在網絡中斷時繼續工作

- **延遲**

將資料放置在地理上距離使用者較近的地方，以便使用者能夠更快地與其互動

- **可伸縮性**

透過在副本上讀，能夠處理比單機更大的讀取量

儘管是一個簡單的目標 - 在幾臺機器上保留相同資料的副本，但複製卻是一個非常棘手的問題。它需要仔細考慮併發和所有可能出錯的事情，並處理這些故障的後果。至少，我們需要處理不可用的節點和網路中斷（這還不包括更隱蔽的故障，例如由於軟體錯誤導致的靜默資料損壞）。

我們討論了複製的三種主要方法：

- **單主複製**

客戶端將所有寫入操作傳送到單個節點（主庫），該節點將資料更改事件流傳送到其他副本（從庫）。讀取可以在任何副本上執行，但從庫的讀取結果可能是陳舊的。

- **多主複製**

客戶端將每個寫入傳送到幾個主庫節點之一，其中任何一個主庫都可以接受寫入。主庫將資料更改事件流傳送給彼此以及任何從庫節點。

- **無主複製**

客戶端將每個寫入傳送到幾個節點，並從多個節點並行讀取，以檢測和糾正具有陳舊資料的節點。

每種方法都有優點和缺點。單主複製是非常流行的，因為它很容易理解，不需要擔心衝突解決。在出現故障節點、網路中斷和延遲峰值的情況下，多主複製和無主複製可以更加健壯，其代價是難以推理並且僅提供非常弱的一致性保證。

複製可以是同步的，也可以是非同步的，這在發生故障時對系統行為有深遠的影響。儘管在系統執行平穩時非同步複製速度很快，但是要弄清楚在複製延遲增加和伺服器故障時會發生什麼，這一點很重要。如果主庫失敗後你將一個非同步更新的從庫提升為新的主庫，那麼最近提交的資料可能會丟失。

我們研究了一些可能由複製延遲引起的奇怪效應，我們也討論了一些有助於決定應用程式在複製延遲時的行為的一致性模型：

- 寫後讀一致性

使用者應該總是能看到自己提交的資料。

- 單調讀

使用者在看到某個時間點的資料後，他們不應該再看到該資料在更早時間點的情況。

- 一致字首讀

使用者應該看到資料處於一種具有因果意義的狀態：例如，按正確的順序看到一個問題和對應的回答。

最後，我們討論了多主複製和無主複製方法所固有的併發問題：因為他們允許多個寫入併發發生，這可能會導致衝突。我們研究了一個數據庫可以使用的演算法來確定一個操作是否發生在另一個操作之前，或者它們是否併發發生。我們還談到了透過合併併發更新來解決衝突的方法。

在下一章中，我們將繼續考察資料分佈在多臺機器間的另一種不同於 複製 的形式：將大資料集分割成 分割槽 。

參考文獻

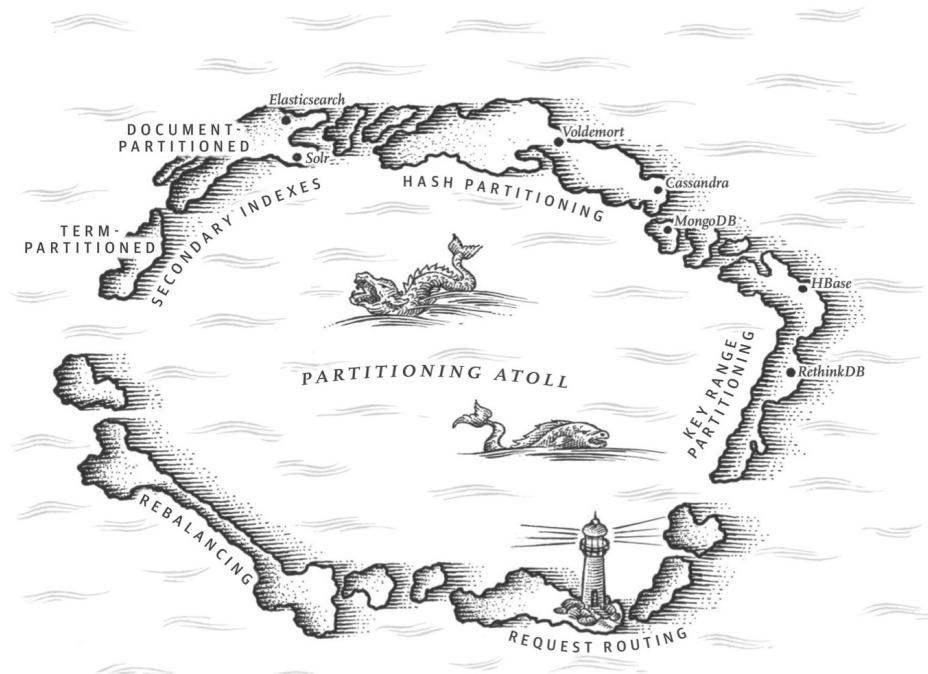
1. Bruce G. Lindsay, Patricia Griffiths Selinger, C. Galtieri, et al.: “[Notes on Distributed Databases](#),” IBM Research, Research Report RJ2571(33471), July 1979.
2. “[Oracle Active Data Guard Real-Time Data Protection and Availability](#),” Oracle White Paper, June 2013.
3. “[AlwaysOn Availability Groups](#),” in *SQL Server Books Online*, Microsoft, 2012.
4. Lin Qiao, Kapil Surlaker, Shirshanka Das, et al.: “[On Brewing Fresh Espresso: LinkedIn’s Distributed Data Serving Platform](#),” at *ACM International Conference on Management of Data* (SIGMOD), June 2013.
5. Jun Rao: “[Intra-Cluster Replication for Apache Kafka](#),” at *ApacheCon North America*, February 2013.
6. “[Highly Available Queues](#),” in *RabbitMQ Server Documentation*, Pivotal Software, Inc., 2014.
7. Yoshinori Matsunobu: “[Semi-Synchronous Replication at Facebook](#),” *yoshinorimatsunobu.blogspot.co.uk*, April 1, 2014.
8. Robbert van Renesse and Fred B. Schneider: “[Chain Replication for Supporting High Throughput and Availability](#),” at *6th USENIX Symposium on Operating System Design and Implementation* (OSDI), December 2004.
9. Jeff Terrace and Michael J. Freedman: “[Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads](#),” at *USENIX Annual Technical Conference* (ATC), June 2009.
10. Brad Calder, Ju Wang, Aaron Ogus, et al.: “[Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency](#),” at *23rd ACM Symposium on Operating Systems Principles* (SOSP), October 2011.
11. Andrew Wang: “[Windows Azure Storage](#),” *umbrant.com*, February 4, 2016.
12. “[Percona Xtrabackup - Documentation](#),” Percona LLC, 2014.
13. Jesse Newland: “[GitHub Availability This Week](#),” *github.com*, September 14, 2012.
14. Mark Imbriaco: “[Downtime Last Saturday](#),” *github.com*, December 26, 2012.
15. John Hugg: “[All in’ with Determinism for Performance and Testing in Distributed Systems](#),” at *Strange Loop*, September 2015.
16. Amit Kapila: “[WAL Internals of PostgreSQL](#),” at *PostgreSQL Conference* (PGCon), May 2012.
17. *[MySQL Internals Manual](#)*. Oracle, 2014.
18. Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, et al.: “[Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services](#),” at *12th USENIX Symposium on Networked Systems Design and Implementation*

- (NSDI), May 2015.
19. "Oracle GoldenGate 12c: Real-Time Access to Real-Time Information," Oracle White Paper, October 2013.
 20. Shirshanka Das, Chavdar Botev, Kapil Surlaker, et al.: "All Aboard the Databus!," at ACM Symposium on Cloud Computing (SoCC), October 2012.
 21. Greg Sabino Mullane: "Version 5 of Bucardo Database Replication System," blog.endpoint.com, June 23, 2014.
 22. Werner Vogels: "Eventually Consistent," ACM Queue, volume 6, number 6, pages 14–19, October 2008.
[doi:10.1145/1466443.1466448](https://doi.org/10.1145/1466443.1466448)
 23. Douglas B. Terry: "Replicated Data Consistency Explained Through Baseball," Microsoft Research, Technical Report MSR-TR-2011-137, October 2011.
 24. Douglas B. Terry, Alan J. Demers, Karin Petersen, et al.: "Session Guarantees for Weakly Consistent Replicated Data," at 3rd International Conference on Parallel and Distributed Information Systems (PDIS), September 1994.
[doi:10.1109/PDIS.1994.331722](https://doi.org/10.1109/PDIS.1994.331722)
 25. Terry Pratchett: *Reaper Man: A Discworld Novel*. Victor Gollancz, 1991. ISBN: 978-0-575-04979-6
 26. "Tungsten Replicator," Continuent, Inc., 2014.
 27. "BDR 0.10.0 Documentation," The PostgreSQL Global Development Group, bdr-project.org, 2015.
 28. Robert Hodges: "If You Must Deploy Multi-Master Replication, Read This First," scale-out-blog.blogspot.co.uk, March 30, 2012.
 29. J. Chris Anderson, Jan Lehnardt, and Noah Slater: *CouchDB: The Definitive Guide*. O'Reilly Media, 2010. ISBN: 978-0-596-15589-6
 30. AppJet, Inc.: "Etherpad and EasySync Technical Manual," github.com, March 26, 2011.
 31. John Day-Richter: "What's Different About the New Google Docs: Making Collaboration Fast," googledrive.blogspot.com, 23 September 2010.
 32. Martin Kleppmann and Alastair R. Beresford: "A Conflict-Free Replicated JSON Datatype," arXiv:1608.03960, August 13, 2016.
 33. Frazer Clement: "Eventual Consistency – Detecting Conflicts," messagepassing.blogspot.co.uk, October 20, 2011.
 34. Robert Hodges: "State of the Art for MySQL Multi-Master Replication," at Percona Live: MySQL Conference & Expo, April 2013.
 35. John Daily: "Clocks Are Bad, or, Welcome to the Wonderful World of Distributed Systems," basho.com, November 12, 2013.
 36. Riley Berton: "Is Bi-Directional Replication (BDR) in Postgres Transactional?," sdf.org, January 4, 2016.
 37. Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, et al.: "Dynamo: Amazon's Highly Available Key-Value Store," at 21st ACM Symposium on Operating Systems Principles (SOSP), October 2007.
 38. Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski: "A Comprehensive Study of Convergent and Commutative Replicated Data Types," INRIA Research Report no. 7506, January 2011.
 39. Sam Elliott: "CRDTs: An UPDATE (or Maybe Just a PUT)," at RICON West, October 2013.
 40. Russell Brown: "A Bluffers Guide to CRDTs in Riak," gist.github.com, October 28, 2013.
 41. Benjamin Farinier, Thomas Gazagnaire, and Anil Madhavapeddy: "Mergeable Persistent Data Structures," at 26es Journées Francophones des Langages Applicatifs (JFLA), January 2015.
 42. Chengzheng Sun and Clarence Ellis: "Operational Transformation in Real-Time Group Editors: Issues, Algorithms, and Achievements," at ACM Conference on Computer Supported Cooperative Work (CSCW), November 1998.
 43. Lars Hofhansl: "HBASE-7709: Infinite Loop Possible in Master/Master Replication," issues.apache.org, January 29, 2013.
 44. David K. Gifford: "Weighted Voting for Replicated Data," at 7th ACM Symposium on Operating Systems Principles (SOSP), December 1979. [doi:10.1145/800215.806583](https://doi.org/10.1145/800215.806583)
 45. Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman: "Flexible Paxos: Quorum Intersection Revisited," arXiv:1608.06696, August 24, 2016.
 46. Joseph Blomstedt: "Re: Absolute Consistency," email to *riak-users* mailing list, lists.basho.com, January 11, 2012.
 47. Joseph Blomstedt: "Bringing Consistency to Riak," at RICON West, October 2012.

48. Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, et al.: “Quantifying Eventual Consistency with PBS,” *Communications of the ACM*, volume 57, number 8, pages 93–102, August 2014. doi:[10.1145/2632792](https://doi.org/10.1145/2632792)
49. Jonathan Ellis: “Modern Hinted Handoff,” datastax.com, December 11, 2012.
50. “Project Voldemort Wiki,” github.com, 2013.
51. “Apache Cassandra 2.0 Documentation,” DataStax, Inc., 2014.
52. “Riak Enterprise: Multi-Datacenter Replication.” Technical whitepaper, Basho Technologies, Inc., September 2014.
53. Jonathan Ellis: “Why Cassandra Doesn’t Need Vector Clocks,” datastax.com, September 2, 2013.
54. Leslie Lamport: “Time, Clocks, and the Ordering of Events in a Distributed System,” *Communications of the ACM*, volume 21, number 7, pages 558–565, July 1978. doi:[10.1145/359545.359563](https://doi.org/10.1145/359545.359563)
55. Joel Jacobson: “Riak 2.0: Data Types,” blog.joeljacobson.com, March 23, 2014.
56. D. Stott Parker Jr., Gerald J. Popek, Gerard Rudisin, et al.: “Detection of Mutual Inconsistency in Distributed Systems,” *IEEE Transactions on Software Engineering*, volume 9, number 3, pages 240–247, May 1983. doi:[10.1109/TSE.1983.236733](https://doi.org/10.1109/TSE.1983.236733)
57. Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, et al.: “Dotted Version Vectors: Logical Clocks for Optimistic Replication,” arXiv:1011.5808, November 26, 2010.
58. Sean Cribbs: “A Brief History of Time in Riak,” at RICON, October 2014.
59. Russell Brown: “Vector Clocks Revisited Part 2: Dotted Version Vectors,” basho.com, November 10, 2015.
60. Carlos Baquero: “Version Vectors Are Not Vector Clocks,” haslab.wordpress.com, July 8, 2011.
61. Reinhard Schwarz and Friedemann Mattern: “Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail,” *Distributed Computing*, volume 7, number 3, pages 149–174, March 1994. doi:[10.1007/BF02277859](https://doi.org/10.1007/BF02277859)

上一章	目錄	下一章
第二部分：分散式資料	設計資料密集型應用	第六章：分割槽

第六章：分割槽



我們必須跳出電腦指令序列的窠臼。敘述定義、描述元資料、梳理關係，而不是編寫過程。

—— Grace Murray Hopper，未來的計算機及其管理（1962）

[TOC]

在 [第五章](#) 中，我們討論了複製——即資料在不同節點上的副本，對於非常大的資料集，或非常高的吞吐量，僅僅進行複制是不夠的：我們需要將資料進行 **分割槽 (partitions)**，也稱為 **分片 (sharding)**ⁱ。

ⁱ. 正如本章所討論的，分割槽是一種有意將大型資料庫分解成小型資料庫的方式。它與 網路分割槽 (network partitions, netsplits) 無關，這是節點之間網路故障的一種。我們將在 [第八章](#) 討論這些錯誤。 ↵

術語澄清

上文中的 **分割槽 (partition)**，在 MongoDB，Elasticsearch 和 Solr Cloud 中被稱為 **分片 (shard)**，在 HBase 中稱之為 **區域 (Region)**，Bigtable 中則是 **表塊 (tablet)**，Cassandra 和 Riak 中是 **虛節點 (vnode)**，Couchbase 中叫做 **虛桶 (vBucket)**。但是 **分割槽 (partitioning)** 是最約定俗成的叫法。

通常情況下，每條資料（每條記錄，每行或每個文件）屬於且僅屬於一個分割槽。有很多方法可以實現這一點，本章將進行深入討論。實際上，每個分割槽都是自己的小型資料庫，儘管資料庫可能支援同時進行多個分割槽的操作。

分割槽主要是為了 **可伸縮性**。不同的分割槽可以放在不共享叢集中的不同節點上（請參閱 [第二部分](#) 關於 **無共享架構** 的定義）。因此，大資料集可以分佈在多個磁碟上，並且查詢負載可以分佈在多個處理器上。

對於在單個分割槽上執行的查詢，每個節點可以獨立執行對自己的查詢，因此可以透過新增更多的節點來擴大查詢吞吐量。大型，複雜的查詢可能會跨多個節點並行處理，儘管這也帶來了新的困難。

分割槽資料庫在 20 世紀 80 年代由 Teradata 和 NonStop SQL 【1】等產品率先推出，最近因為 NoSQL 資料庫和基於 Hadoop 的資料倉庫重新被關注。有些系統是為事務性工作設計的，有些系統則用於分析（請參閱 “[事務處理還是分析](#)”）：這種差異會影響系統的運作方式，但是分割槽的基本原理均適用於這兩種工作方式。

在本章中，我們將首先介紹分割大型資料集的不同方法，並觀察索引如何與分割槽配合。然後我們將討論 [分割槽再平衡 \(rebalancing\)](#)，如果想要新增或刪除叢集中的節點，則必須進行再平衡。最後，我們將概述資料庫如何將請求路由到正確的分割槽並執行查詢。

分割槽與複製

分割槽通常與複製結合使用，使得每個分割槽的副本儲存在多個節點上。這意味著，即使每條記錄屬於一個分割槽，它仍然可以儲存在多個不同的節點上以獲得容錯能力。

一個節點可能儲存多個分割槽。如果使用主從複製模型，則分割槽和複製的組合如 圖 6-1 所示。每個分割槽領導者（主庫）被分配給一個節點，追隨者（從庫）被分配給其他節點。每個節點可能是某些分割槽的主庫，同時是其他分割槽的從庫。

我們在 [第五章](#) 討論的關於資料庫複製的所有內容同樣適用於分割槽的複製。大多數情況下，分割槽方案的選擇與複製方案的選擇是獨立的，為簡單起見，本章中將忽略複製。

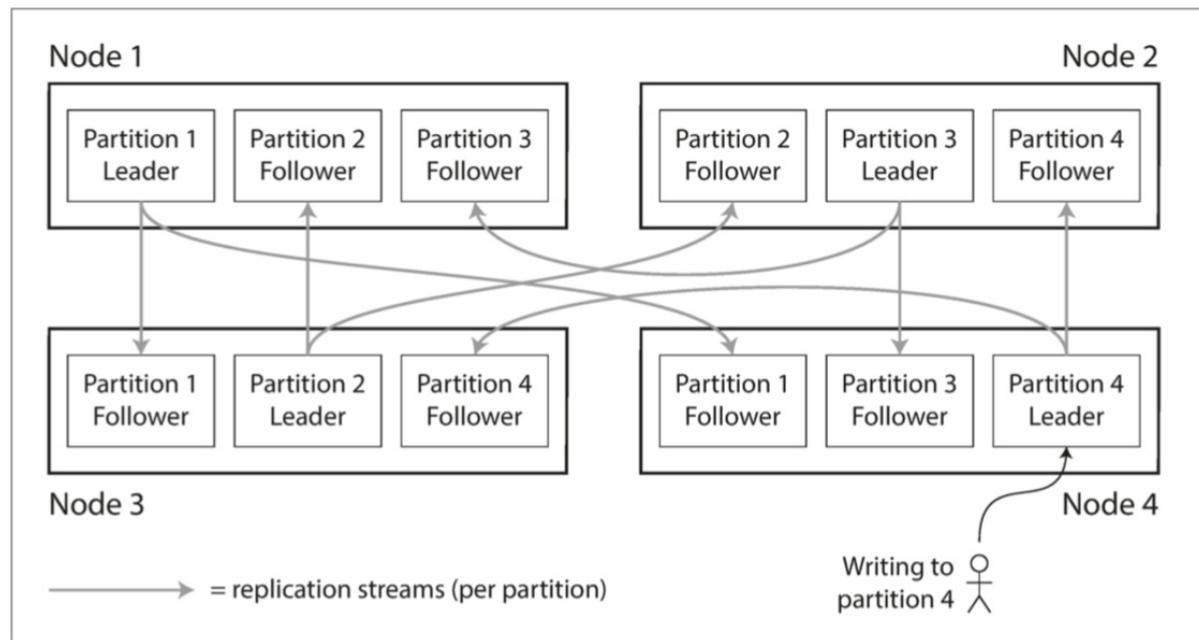


圖 6-1 組合使用複製和分割槽：每個節點充當某些分割槽的主庫，其他分割槽充當從庫。

鍵值資料的分割槽

假設你有大量資料並且想要分割槽，如何決定在哪些節點上儲存哪些記錄呢？

分割槽目標是將資料和查詢負載均勻分佈在各個節點上。如果每個節點公平分享資料和負載，那麼理論上 10 個節點應該能夠處理 10 倍的資料量和 10 倍的單個節點的讀寫吞吐量（暫時忽略複製）。

如果分割槽是不公平的，一些分割槽比其他分割槽有更多的資料或查詢，我們稱之為 **偏斜 (skew)**。資料偏斜的存在使分割槽效率下降很多。在極端的情況下，所有的負載可能壓在一個分割槽上，其餘 9 個節點空閒的，瓶頸落在這一個繁忙的節點上。不均衡導致的高負載的分割槽被稱為 **熱點 (hot spot)**。

避免熱點最簡單的方法是將記錄隨機分配給節點。這將在所有節點上平均分配資料，但是它有一個很大的缺點：當你試圖讀取一個特定的值時，你無法知道它在哪個節點上，所以你必須並行地查詢所有的節點。

我們可以做得更好。現在假設你有一個簡單的鍵值資料模型，其中你總是透過其主鍵訪問記錄。例如，在一本老式的紙質百科全書中，你可以透過標題來查詢一個條目；由於所有條目按字母順序排序，因此你可以快速找到你要查詢的條目。

根據鍵的範圍分割槽

一種分割槽的方法是為每個分割槽指定一塊連續的鍵範圍（從最小值到最大值），如紙質百科全書的卷（[圖 6-2](#)）。如果知道範圍之間的邊界，則可以輕鬆確定哪個分割槽包含某個值。如果你還知道分割槽所在的節點，那麼可以直接向相應的節點發出請求（對於百科全書而言，就像從書架上選取正確的書籍）。

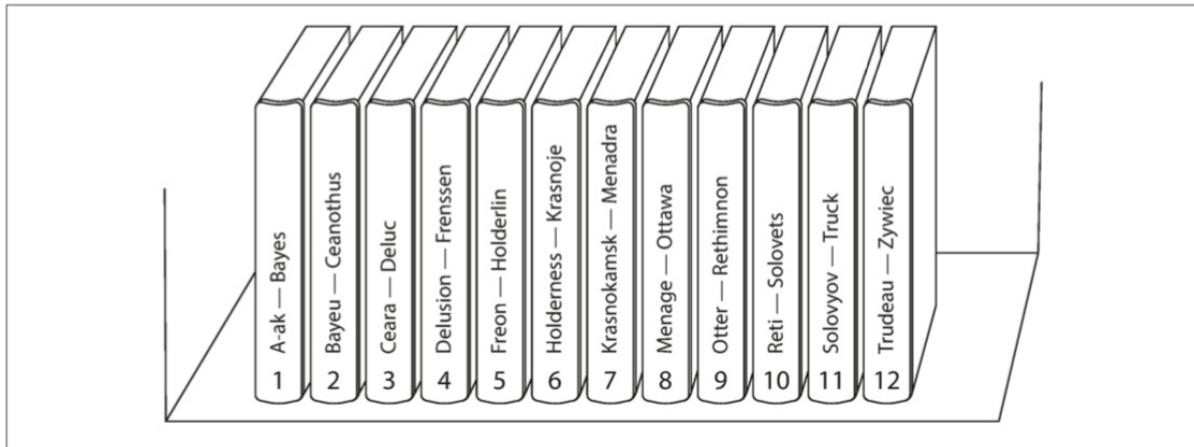


圖 6-2 印刷版百科全書按照關鍵字範圍進行分割槽

鍵的範圍不一定均勻分佈，因為資料也很可能不均勻分佈。例如在 [圖 6-2](#) 中，第 1 卷包含以 A 和 B 開頭的單詞，但第 12 卷則包含以 T、U、V、X、Y 和 Z 開頭的單詞。只是簡單的規定每個捲包含兩個字母會導致一些捲比其他捲大。為了均勻分配資料，分割槽邊界需要依據資料調整。

分割槽邊界可以由管理員手動選擇，也可以由資料庫自動選擇（我們會在“[分割槽再平衡](#)”中更詳細地討論分割槽邊界的選擇）。Bigtable 使用了這種分割槽策略，以及其開源等價物 HBase 【2, 3】，RethinkDB 和 2.4 版本之前的 MongoDB 【4】。

在每個分割槽中，我們可以按照一定的順序儲存鍵（請參閱“[SSTables 和 LSM 樹](#)”）。好處是進行範圍掃描非常簡單，你可以將鍵作為聯合索引來處理，以便在一次查詢中獲取多個相關記錄（請參閱“[多列索引](#)”）。例如，假設我們有一個程式來儲存感測器網路的資料，其中主鍵是測量的時間戳（年月日時分秒）。範圍掃描在這種情況下非常有用，因為我們可以輕鬆獲取某個月份的所有資料。

然而，Key Range 分割槽的缺點是某些特定的訪問模式會導致熱點。如果主鍵是時間戳，則分割槽對應於時間範圍，例如，給每天分配一個分割槽。不幸的是，由於我們在測量發生時將資料從感測器寫入資料庫，因此所有寫入操作都會轉到同一個分割槽（即今天的分割槽），這樣分割槽可能會因寫入而過載，而其他分割槽則處於空閒狀態【5】。

為了避免感測器資料庫中的這個問題，需要使用除了時間戳以外的其他東西作為主鍵的第一個部分。例如，可以在每個時間戳前新增感測器名稱，這樣會首先按感測器名稱，然後按時間進行分割槽。假設有多個感測器同時執行，寫入負載將最終均勻分佈在不同分割槽上。現在，當想要在一個時間範圍內獲取多個感測器的值時，你需要為每個感測器名稱執行一個單獨的範圍查詢。

根據鍵的雜湊分割槽

由於偏斜和熱點的風險，許多分散式資料儲存使用雜湊函式來確定給定鍵的分割槽。

一個好的雜湊函式可以將偏斜的資料均勻分佈。假設你有一個 32 位雜湊函式，無論何時給定一個新的字串輸入，它將返回一個 0 到 $2^{32} - 1$ 之間的“隨機”數。即使輸入的字串非常相似，它們的雜湊也會均勻分佈在這個數字範圍內。

出於分割槽的目的，雜湊函式不需要多麼強壯的加密演算法：例如，Cassandra 和 MongoDB 使用 MD5，Voldemort 使用 Fowler-Noll-Vo 函式。許多程式語言都有內建的簡單雜湊函式（它們用於散列表），但是它們可能不適合分割槽：例如，在 Java 的 `Object.hashCode()` 和 Ruby 的 `Object#hash`，同一個鍵可能在不同的程序中有不同的雜湊值【6】。

一旦你有一個合適的鍵雜湊函式，你可以為每個分割槽分配一個雜湊範圍（而不是鍵的範圍），每個透過雜湊雜湊落在分割槽範圍內的鍵將被儲存在該分割槽中。如 圖 6-3 所示。

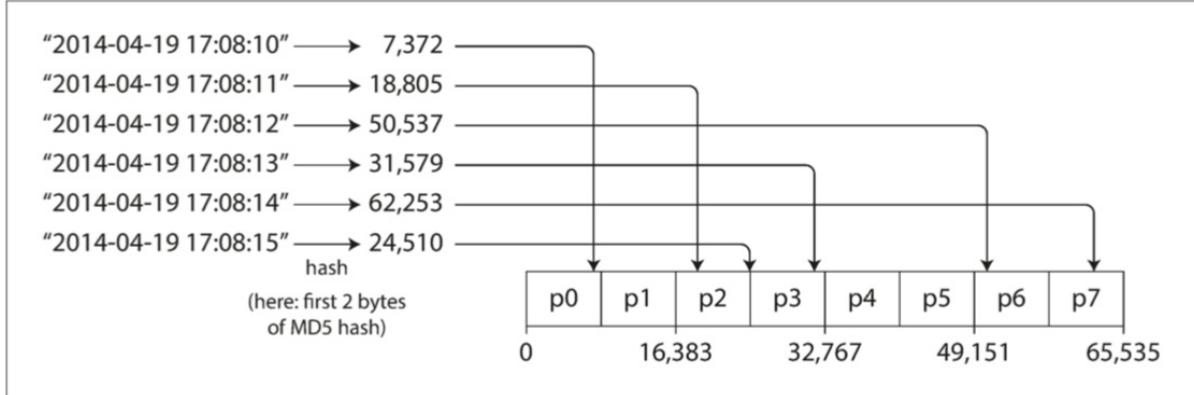


圖 6-3 按雜湊鍵分割槽

這種技術擅長在分割槽之間公平地分配鍵。分割槽邊界可以是均勻間隔的，也可以是偽隨機選擇的（在這種情況下，該技術有時也被稱為 **一致性雜湊**，即 consistent hashing）。

一致性雜湊

一致性雜湊由 Karger 等人定義。【7】用於跨網際網路級別的快取系統，例如 CDN 中，是一種能均勻分配負載的方法。它使用隨機選擇的 **分割槽邊界**（partition boundaries）來避免中央控制或分散式共識的需要。請注意，這裡的一致性與複製一致性（請參閱 第五章）或 ACID 一致性（請參閱 第七章）無關，而只是描述了一種再平衡（rebalancing）的特定方法。

正如我們將在“**分割槽再平衡**”中所看到的，這種特殊的方法對於資料庫實際上並不是很好，所以在實際中很少使用（某些資料庫的文件仍然會使用一致性雜湊的說法，但是它往往是不準確的）。因為有可能產生混淆，所以最好避免使用一致性雜湊這個術語，而只是把它稱為 **雜湊分割槽**（hash partitioning）。

不幸的是，透過使用鍵雜湊進行分割槽，我們失去了鍵範圍分割槽的一個很好的屬性：高效執行範圍查詢的能力。曾經相鄰的鍵現在分散在所有分割槽中，所以它們之間的順序就丟失了。在 MongoDB 中，如果你使用了基於雜湊的分割槽模式，則任何範圍查詢都必須傳送到所有分割槽【4】。Riak【9】、Couchbase【10】或 Voldemort 不支援主鍵上的範圍查詢。

Cassandra 採取了折衷的策略【11, 12, 13】。Cassandra 中的表可以使用由多個列組成的複合主鍵來宣告。鍵中只有第一列會作為雜湊的依據，而其他列則被用作 Cassandra 的 SSTables 中排序資料的連線索引。儘管查詢無法在複合主鍵的第一列中按範圍掃表，但如果第一列已經指定了固定值，則可以對該鍵的其他列執行有效的範圍掃描。

組合索引方法為一對多關係提供了一個優雅的資料模型。例如，在社交媒體網站上，一個使用者可能會發布很多更新。如果更新的主鍵被選擇為 `(user_id, update_timestamp)`，那麼你可以有效地檢索特定使用者在某個時間間隔內按時間戳排序的所有更新。不同的使用者可以儲存在不同的分割槽上，對於每個使用者，更新按時間戳順序儲存在單個分割槽上。

負載偏斜與熱點消除

如前所述，雜湊分割槽可以幫助減少熱點。但是，它不能完全避免它們：在極端情況下，所有的讀寫操作都是針對同一個鍵的，所有的請求都會被路由到同一個分割槽。

這種場景也許並不常見，但並非聞所未聞：例如，在社交媒體網站上，一個擁有數百萬追隨者的名人使用者在做某事時可能會引發一場風暴【14】。這個事件可能導致同一個鍵的大量寫入（鍵可能是名人的使用者 ID，或者人們正在評論的動作的 ID）。雜湊策略不起作用，因為兩個相同 ID 的雜湊值仍然是相同的。

如今，大多數資料系統無法自動補償這種高度偏斜的負載，因此應用程式有責任減少偏斜。例如，如果一個主鍵被認為是非常火爆的，一個簡單的方法是在主鍵的開始或結尾新增一個隨機數。只要一個兩位數的十進位制隨機數就可以將主鍵分散為 100 種不同的主鍵，從而儲存在不同的分割槽中。

然而，將主鍵進行分割之後，任何讀取都必須要做額外的工作，因為他們必須從所有 100 個主鍵分佈中讀取資料並將其合併。此技術還需要額外的記錄：只需要對少量熱點附加隨機數；對於寫入吞吐量低的絕大多數主鍵來說是不必要的開銷。因此，你還需要一些方法來跟蹤哪些鍵需要被分割。

也許在將來，資料系統將能夠自動檢測和補償偏斜的工作負載；但現在，你需要自己來權衡。

分割槽與次級索引

到目前為止，我們討論的分割槽方案依賴於鍵值資料模型。如果只通過主鍵訪問記錄，我們可以從該鍵確定分割槽，並使用它來將讀寫請求路由到負責該鍵的分割槽。

如果涉及次級索引，情況會變得更加複雜（參考“[其他索引結構](#)”）。次級索引通常並不能唯一地標識記錄，而是一種搜尋記錄中出現特定值的方式：查詢使用者 123 的所有操作、查詢包含詞語 `hogwash` 的所有文章、查詢所有顏色為紅色的車輛等等。

次級索引是關係型資料庫的基礎，並且在文件資料庫中也很普遍。許多鍵值儲存（如 HBase 和 Voldemort）為了減少實現的複雜度而放棄了次級索引，但是一些（如 Riak）已經開始新增它們，因為它們對於資料模型實在是太有用了。並且次級索引也是 Solr 和 Elasticsearch 等搜尋伺服器的基石。

次級索引的問題是它們不能整齊地對映到分割槽。有兩種用次級索引對資料庫進行分割槽的方法：[基於文件的分割槽 \(document-based\)](#) 和 [基於關鍵詞 \(term-based\)](#) 的分割槽。

基於文件的次級索引進行分割槽

假設你正在經營一個銷售二手車的網站（如 [圖 6-4](#) 所示）。每個列表都有一個唯一的 ID——稱之為文件 ID——並且用文件 ID 對資料庫進行分割槽（例如，分割槽 0 中的 ID 0 到 499，分割槽 1 中的 ID 500 到 999 等）。

你想讓使用者搜尋汽車，允許他們透過顏色和廠商過濾，所以需要一個在顏色和廠商上的次級索引（文件資料庫中這些是 **欄位 (field)**，關係資料庫中這些是 **列 (column)**）。如果你聲明瞭索引，則資料庫可以自動執行索引ⁱⁱ。例如，無論何時將紅色汽車新增到資料庫，資料庫分割槽都會自動將其新增到索引條目 `color:red` 的文件 ID 列表中。

ⁱⁱ. 如果資料庫僅支援鍵值模型，則你可能會嘗試在應用程式程式碼中建立從值到文件 ID 的對映來實現次級索引。如果沿著這條路線走下去，請萬分小心，確保你的索引與底層資料保持一致。競爭條件和間歇性寫入失敗（其中一些更改已儲存，但其他更改未儲存）很容易導致資料不同步 - 請參閱“[多物件事務的需求](#)”。 ↪

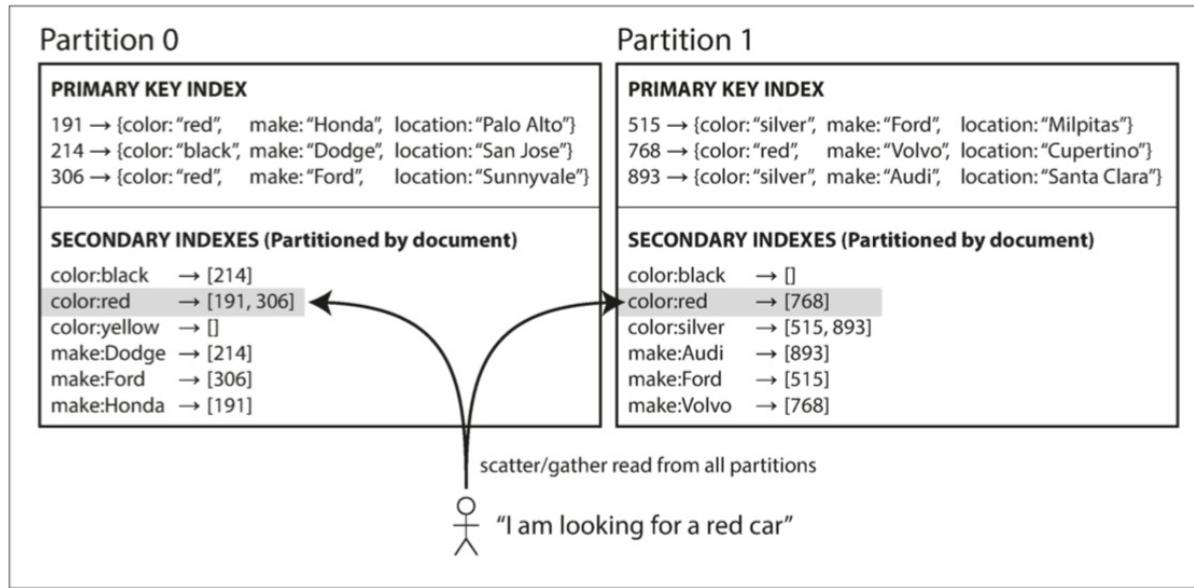


圖 6-4 基於文件的次級索引進行分割槽

在這種索引方法中，每個分割槽是完全獨立的：每個分割槽維護自己的次級索引，僅覆蓋該分割槽中的文件。它不關心儲存在其他分割槽的資料。無論何時你需要寫入資料庫（新增，刪除或更新文件），只需處理包含你正在編寫的文件ID的分割槽即可。出於這個原因，**文件分割槽索引** 也被稱為 **本地索引**（而不是將在下一節中描述的 **全域性索引**）。

但是，從文件分割槽索引中讀取需要注意：除非你對文件ID做了特別的處理，否則沒有理由將所有具有特定顏色或特定品牌的汽車放在同一個分割槽中。在 圖 6-4 中，紅色汽車出現在分割槽 0 和分割槽 1 中。因此，如果要搜尋紅色汽車，則需要將查詢傳送到所有分割槽，併合並所有返回的結果。

這種查詢分割槽資料庫的方法有時被稱為 **分散 / 聚集 (scatter/gather)**，並且可能會使次級索引上的讀取查詢相當昂貴。即使並行查詢分割槽，分散 / 聚集也容易導致尾部延遲放大（請參閱 “實踐中的百分位點”）。然而，它被廣泛使用：MongoDB，Riak【15】，Cassandra【16】，Elasticsearch【17】，SolrCloud【18】和 VoltDB【19】都使用文件分割槽次級索引。大多數資料庫供應商建議你構建一個能從單個分割槽提供次級索引查詢的分割槽方案，但這並不總是可行，尤其是當在單個查詢中使用多個次級索引時（例如同時需要按顏色和製造商查詢）。

基於關鍵詞(Term)的次級索引進行分割槽

我們可以構建一個覆蓋所有分割槽資料的 **全域性索引**，而不是給每個分割槽建立自己的次級索引（本地索引）。但是，我們不能只把這個索引儲存在一個節點上，因為它可能會成為瓶頸，違背了分割槽的目的。全域性索引也必須進行分割槽，但可以採用與主鍵不同的分割槽方式。

圖 6-5 描述了這可能是什麼樣子：來自所有分割槽的紅色汽車在紅色索引中，並且索引是分割槽的，首字母從 `a` 到 `r` 的顏色在分割槽 0 中，`s` 到 `z` 的在分割槽 1。汽車製造商的索引也與之類似（分割槽邊界在 `f` 和 `h` 之間）。

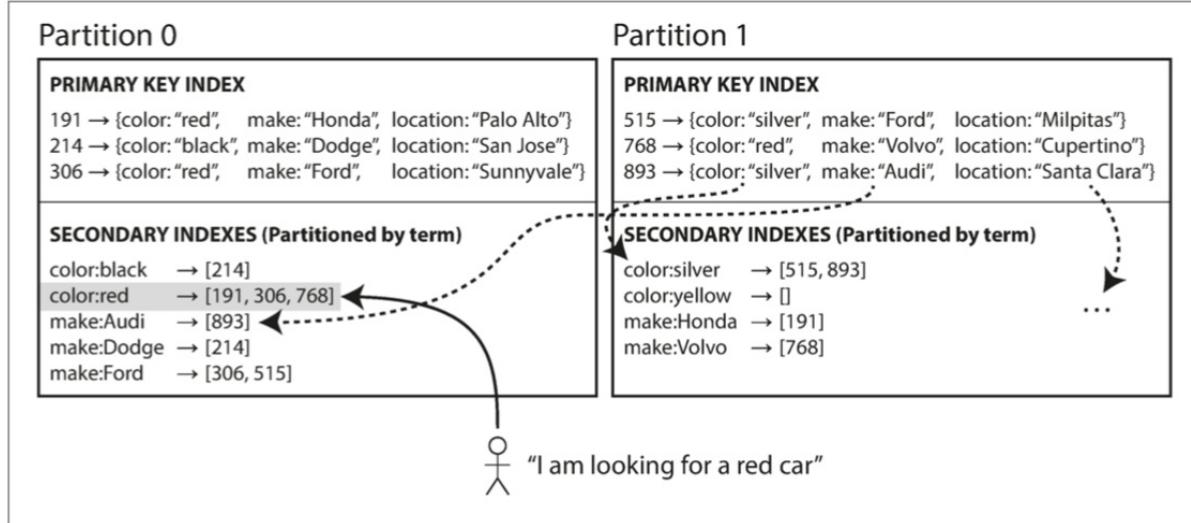


圖 6-5 基於關鍵詞對次級索引進行分割槽

我們將這種索引稱為 **關鍵詞分割槽 (term-partitioned)**，因為我們尋找的關鍵詞決定了索引的分割槽方式。例如，一個關鍵詞可能是：`color:red`。**關鍵詞 (Term)** 這個名稱來源於全文搜尋索引（一種特殊的次級索引），指文件中出現的所有單詞。

和之前一樣，我們可以透過 **關鍵詞** 本身或者它的雜湊進行索引分割槽。根據關鍵詞本身來分割槽對於範圍掃描非常有用（例如對於數值類的屬性，像汽車的報價），而對關鍵詞的雜湊分割槽提供了負載均衡的能力。

關鍵詞分割槽的全域性索引優於文件分割槽索引的地方點是它可以使讀取更有效率：不需要 **分散 / 收集** 所有分割槽，客戶端只需要向包含關鍵詞的分割槽發出請求。全域性索引的缺點在於寫入速度較慢且較為複雜，因為寫入單個文件現在可能會影響索引的多個分割槽（文件中的每個關鍵詞可能位於不同的分割槽或者不同的節點上）。

理想情況下，索引總是最新的，寫入資料庫的每個文件都會立即反映在索引中。但是，在關鍵詞分割槽索引中，這需要跨分割槽的分散式事務，並不是所有資料庫都支援（請參閱 [第七章](#) 和 [第九章](#)）。

在實踐中，對全域性次級索引的更新通常是 **非同步** 的（也就是說，如果在寫入之後不久讀取索引，剛才所做的更改可能尚未反映在索引中）。例如，Amazon DynamoDB 聲稱在正常情況下，其全域性次級索引會在不到一秒的時間內更新，但在基礎架構出現故障的情況下可能會有延遲【20】。

全域性關鍵詞分割槽索引的其他用途包括 Riak 的搜尋功能【21】和 Oracle 資料倉庫，它允許你在本地和全域性索引之間進行選擇【22】。我們將在 [第十二章](#) 中繼續關鍵詞分割槽次級索引實現的話題。

分割槽再平衡

隨著時間的推移，資料庫會有各種變化：

- 查詢吞吐量增加，所以你想要新增更多的 CPU 來處理負載。
- 資料集大小增加，所以你想新增更多的磁碟和 RAM 來儲存它。
- 機器出現故障，其他機器需要接管故障機器的責任。

所有這些更改都需要資料和請求從一個節點移動到另一個節點。將負載從叢集中的一個節點向另一個節點移動的過程稱為 **再平衡 (rebalancing)**。

無論使用哪種分割槽方案，再平衡通常都要滿足一些最低要求：

- 再平衡之後，負載（資料儲存，讀取和寫入請求）應該在叢集中的節點之間公平地共享。
- 再平衡發生時，資料庫應該繼續接受讀取和寫入。
- 節點之間只移動必須的資料，以便快速再平衡，並減少網路和磁碟 I/O 負載。

再平衡策略

有幾種不同的分割槽分配方法【23】，讓我們依次簡要討論一下。

反面教材：hash mod N

我們在前面說過（圖 6-3），最好將可能的雜湊分成不同的範圍，並將每個範圍分配給一個分割槽（例如，如果 $0 \leq \text{hash}(\text{key}) < b_0$ ，則將鍵分配給分割槽 0，如果 $b_0 \leq \text{hash}(\text{key}) < b_1$ ，則分配給分割槽 1）

也許你想知道為什麼我們不使用 **取模 (mod)**（許多程式語言中的 % 運算子）。例如， $\text{hash}(\text{key}) \bmod 10$ 會返回一個介於 0 和 9 之間的數字（如果我們將雜湊寫為十進位制數，雜湊模 10 將是最後一個數字）。如果我們有 10 個節點，編號為 0 到 9，這似乎是將每個鍵分配給一個節點的簡單方法。

模 N ($\bmod N$) 方法的問題是，如果節點數量 N 發生變化，大多數鍵將需要從一個節點移動到另一個節點。例如，假設 $\text{hash}(\text{key})=123456$ 。如果最初有 10 個節點，那麼這個鍵一開始放在節點 6 上（因為 $123456 \bmod 10 = 6$ ）。當你增長到 11 個節點時，鍵需要移動到節點 3 ($123456 \bmod 11 = 3$)，當你增長到 12 個節點時，需要移動到節點 0 ($123456 \bmod 12 = 0$)。這種頻繁的舉動使得再平衡的成本過高。

我們需要一種只移動必需資料的方法。

固定數量的分割槽

幸運的是，有一個相當簡單的解決方案：建立比節點更多的分割槽，併為每個節點分配多個分割槽。例如，執行在 10 個節點的叢集上的資料庫可能會從一開始就被拆分為 1,000 個分割槽，因此大約有 100 個分割槽被分配給每個節點。

現在，如果一個節點被新增到叢集中，新節點可以從當前每個節點中 竊取 一些分割槽，直到分割槽再次公平分配。這個過程如 圖 6-6 所示。如果從叢集中刪除一個節點，則會發生相反的情況。

只有分割槽在節點之間的移動。分割槽的數量不會改變，鍵所指定的分割槽也不會改變。唯一改變的是分割槽所在的節點。這種變更並不是即時的 — 在網路上傳輸大量的資料需要一些時間 — 所以在傳輸過程中，原有分割槽仍然會接受讀寫操作。

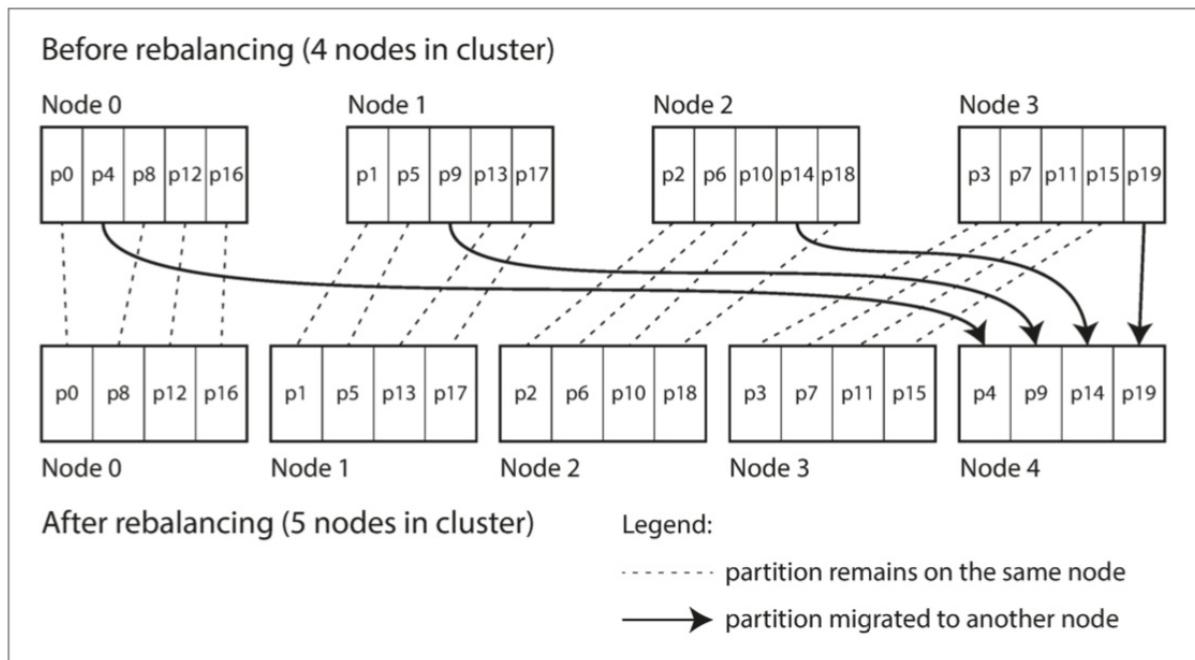


圖 6-6 將新節點新增到每個節點具有多個分割槽的資料庫叢集。

原則上，你甚至可以解決叢集中的硬體不匹配問題：透過為更強大的節點分配更多的分割槽，可以強制這些節點承載更多的負載。在 Riak 【15】、Elasticsearch 【24】、Couchbase 【10】和 Voldemort 【25】中使用了這種再平衡的方法。

在這種配置中，分割槽的數量通常在資料庫第一次建立時確定，之後不會改變。雖然原則上可以分割和合並分割槽（請參閱下一節），但固定數量的分割槽在操作上更簡單，因此許多固定分割槽資料庫選擇不實施分割槽分割。因此，一開始配置的分割槽數就是你可以擁有的最大節點數量，所以你需要選擇足夠多的分割槽以適應未來的增長。但是，每個分割槽也有管理開銷，所以選擇太大的數字會適得其反。

如果資料集的總大小難以預估（例如，可能它開始很小，但隨著時間的推移會變得更大），選擇正確的分割槽數是困難的。由於每個分割槽包含了總資料量固定比率的資料，因此每個分割槽的大小與叢集中的資料總量成比例增長。如果分割槽非常大，再平衡和從節點故障恢復變得昂貴。但是，如果分割槽太小，則會產生太多的開銷。當分割槽大小“恰到好處”的時候才能獲得很好的效能，如果分割槽數量固定，但資料量變動很大，則難以達到最佳效能。

動態分割槽

對於使用鍵範圍分割槽的資料庫（請參閱“[根據鍵的範圍分割槽](#)”），具有固定邊界的固定數量的分割槽將非常不便：如果出現邊界錯誤，則可能會導致一個分割槽中的所有資料或者其他分割槽中的所有資料為空。手動重新配置分割槽邊界將非常繁瑣。

出於這個原因，按鍵的範圍進行分割槽的資料庫（如 HBase 和 RethinkDB）會動態建立分割槽。當分割槽增長到超過配置的大小時（在 HBase 上，預設值是 10GB），會被分成兩個分割槽，每個分割槽約佔一半的資料【26】。與之相反，如果大量資料被刪除並且分割槽縮小到某個閾值以下，則可以將其與相鄰分割槽合併。此過程與 B 樹頂層發生的過程類似（請參閱“[B 樹](#)”）。

每個分割槽分配給一個節點，每個節點可以處理多個分割槽，就像固定數量的分割槽一樣。大型分割槽拆分後，可以將其中的一半轉移到另一個節點，以平衡負載。在 HBase 中，分割槽檔案的傳輸透過 HDFS（底層使用的分散式檔案系統）來實現【3】。

動態分割槽的一個優點是分割槽數量適應總資料量。如果只有少量的資料，少量的分割槽就足夠了，所以開銷很小；如果有大量的資料，每個分割槽的大小被限制在一個可配置的最大值【23】。

需要注意的是，一個空的資料庫從一個分割槽開始，因為沒有關於在哪裡繪製分割槽邊界的先驗資訊。資料集開始時很小，直到達到第一個分割槽的分割點，所有寫入操作都必須由單個節點處理，而其他節點則處於空閒狀態。為了解決這個問題，HBase 和 MongoDB 允許在一個空的資料庫上配置一組初始分割槽（這被稱為 **預分割**，即 pre-splitting）。在鍵範圍分割槽的情況中，預分割需要提前知道鍵是如何進行分配的【4,26】。

動態分割槽不僅適用於資料的範圍分割槽，而且也適用於雜湊分割槽。從版本 2.4 開始，MongoDB 同時支援範圍和雜湊分割槽，並且都支援動態分割分割槽。

按節點比例分割槽

透過動態分割槽，分割槽的數量與資料集的大小成正比，因為拆分和合並過程將每個分割槽的大小保持在固定的最小值和最大值之間。另一方面，對於固定數量的分割槽，每個分割槽的大小與資料集的大小成正比。在這兩種情況下，分割槽的數量都與節點的數量無關。

Cassandra 和 Ketama 使用的第三種方法是使分割槽數與節點數成正比——換句話說，每個節點具有固定數量的分割槽【23,27,28】。在這種情況下，每個分割槽的大小與資料集大小成比例地增長，而節點數量保持不變，但是當增加節點數時，分割槽將再次變小。由於較大的資料量通常需要較大數量的節點進行儲存，因此這種方法也使每個分割槽的小較為穩定。

當一個新節點加入叢集時，它隨機選擇固定數量的現有分割槽進行拆分，然後佔有這些拆分分割槽中每個分割槽的一半，同時將每個分割槽的另一半留在原地。隨機化可能會產生不公平的分割，但是平均在更大數量的分割槽上時（在 Cassandra 中，預設情況下，每個節點有 256 個分割槽），新節點最終從現有節點獲得公平的負載份額。Cassandra 3.0 引入了另一種再平衡的演算法來避免不公平的分割【29】。

隨機選擇分割槽邊界要求使用基於雜湊的分割槽（可以從雜湊函式產生的數字範圍中挑選邊界）。實際上，這種方法最符合一致性雜湊的原始定義【7】（請參閱“[一致性雜湊](#)”）。最新的雜湊函式可以在較低元資料開銷的情況下達到類似的效果【8】。

運維：手動還是自動再平衡

關於再平衡有一個重要問題：自動還是手動進行？

在全自動再平衡（系統自動決定何時將分割槽從一個節點移動到另一個節點，無須人工干預）和完全手動（分割槽指派給節點由管理員明確配置，僅在管理員明確重新配置時才會更改）之間有一個權衡。例如，Couchbase、Riak 和 Voldemort 會自動生成建議的分割槽分配，但需要管理員提交才能生效。

全自動再平衡可以很方便，因為正常維護的操作工作較少。然而，它可能是不可預測的。再平衡是一個昂貴的操作，因為它需要重新路由請求並將大量資料從一個節點移動到另一個節點。如果沒有做好，這個過程可能會使網路或節點負載過重，降低其他請求的效能。

這種自動化與自動故障檢測相結合可能十分危險。例如，假設一個節點過載，並且對請求的響應暫時很慢。其他節點得出結論：過載的節點已經死亡，並自動重新平衡叢集，使負載離開它。這會對已經超負荷的節點，其他節點和網路造成額外的負載，從而使情況變得更糟，並可能導致級聯失敗。

出於這個原因，再平衡的過程中有人參與是一件好事。這比全自動的過程慢，但可以幫助防止運維意外。

請求路由

現在我們已經將資料集分割到多個機器上執行的多個節點上。但是仍然存在一個懸而未決的問題：當客戶想要發出請求時，如何知道要連線哪個節點？隨著分割槽的重新平衡，分割槽對節點的分配也發生變化。為了回答這個問題，需要有人知曉這些變化：如果我想讀或寫鍵“foo”，需要連線哪個 IP 地址和埠號？

這個問題可以概括為 **服務發現（service discovery）**，它不僅限於資料庫。任何可透過網路訪問的軟體都有這個問題，特別是如果它的目標是高可用性（在多臺機器上執行冗餘配置）。許多公司已經編寫了自己的內部服務發現工具，其中許多已經作為開源釋出【30】。

概括來說，這個問題有幾種不同的方案（如圖 6-7 所示）：

1. 允許客戶聯絡任何節點（例如，透過 **迴圈策略的負載均衡**，即 Round-Robin Load Balancer）。如果該節點恰巧擁有所請求的分割槽，則它可以直接處理該請求；否則，它將請求轉發到適當的節點，接收回復並傳遞給客戶端。
2. 首先將所有來自客戶端的請求傳送到路由層，它決定了應該處理請求的節點，並相應地轉發。此路由層本身不處理任何請求；它僅負責分割槽的負載均衡。
3. 要求客戶端知道分割槽和節點的分配。在這種情況下，客戶端可以直接連線到適當的節點，而不需要任何中介。

以上所有情況中的關鍵問題是：作出路由決策的元件（可能是節點之一，還是路由層或客戶端）如何瞭解分割槽 - 節點之間的分配關係變化？

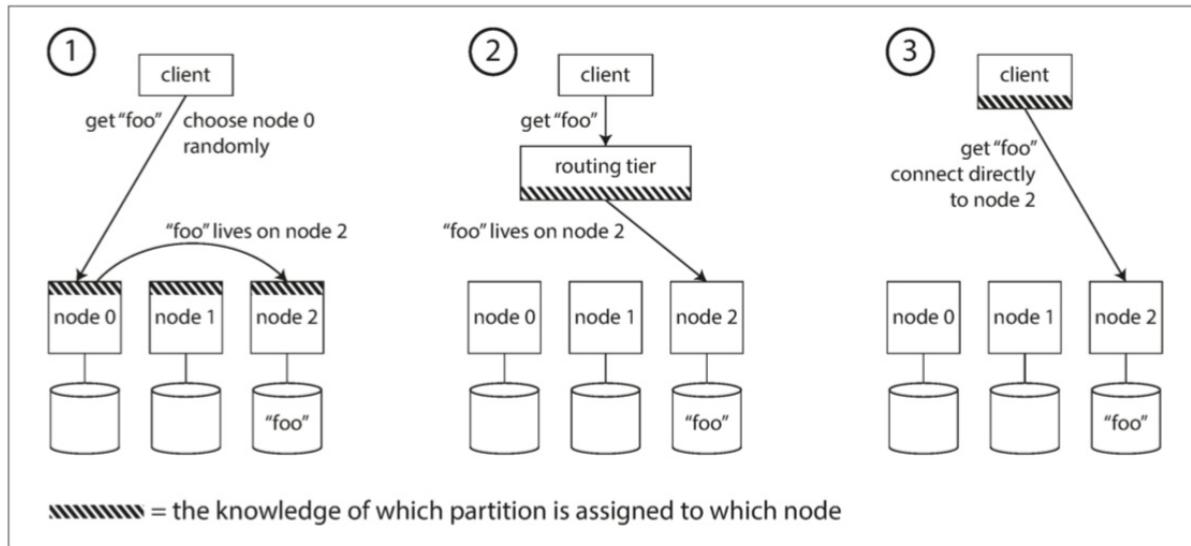


圖 6-7 將請求路由到正確節點的三種不同方式。

這是一個具有挑戰性的問題，因為重要的是所有參與者都達成共識 - 否則請求將被傳送到錯誤的節點，得不到正確的處理。在分散式系統中有達成共識的協議，但很難正確地實現（見 [第九章](#)）。

許多分散式資料系統都依賴於一個獨立的協調服務，比如 ZooKeeper 來跟蹤叢集元資料，如 [圖 6-8](#) 所示。每個節點在 ZooKeeper 中註冊自己，ZooKeeper 維護分割槽到節點的可靠對映。其他參與者（如路由層或分割槽感知客戶端）可以在 ZooKeeper 中訂閱此資訊。只要分割槽分配發生了改變，或者叢集中新增或刪除了一個節點，ZooKeeper 就會通知路由層使路資訊保持最新狀態。

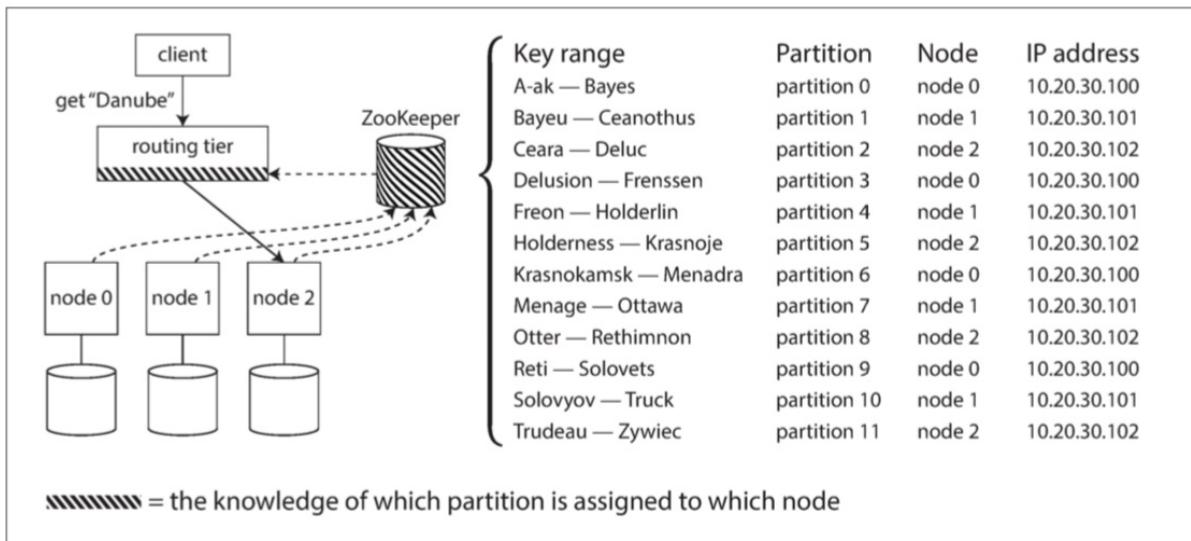


圖 6-8 使用 ZooKeeper 跟踪分割槽分配給節點。

例如，LinkedIn的Espresso使用Helix [【31】](#) 進行叢集管理（依靠ZooKeeper），實現瞭如 [圖6-8](#)所示的路由層。HBase、SolrCloud和Kafka也使用ZooKeeper來跟蹤分割槽分配。MongoDB具有類似的體系結構，但它依賴於自己的配置伺服器（**config server**）實現和mongos守護程序作為路由層。

Cassandra 和 Riak 採取不同的方法：他們在節點之間使用 **流言協議（gossip protocol）** 來傳播叢集狀態的變化。請求可以傳送到任意節點，該節點會轉發到包含所請求的分割槽的適當節點（[圖 6-7](#) 中的方法 1）。這個模型在資料庫節點中增加了更多的複雜性，但是避免了對像 ZooKeeper 這樣的外部協調服務的依賴。

Couchbase 不會自動進行再平衡，這簡化了設計。通常情況下，它配置了一個名為 moxi 的路由層，它會從叢集節點了解路由變化 [【32】](#)。

當使用路由層或向隨機節點發送請求時，客戶端仍然需要找到要連線的 IP 地址。這些地址並不像分割槽的節點分佈變化的那麼快，所以使用 DNS 通常就足夠了。

執行並行查詢

到目前為止，我們只關注讀取或寫入單個鍵的非常簡單的查詢（加上基於文件分割槽的次級索引場景下的分散 / 聚集查詢）。這也是大多數 NoSQL 分散式資料儲存所支援的訪問層級。

然而，通常用於分析的 **大規模並行處理 (MPP, Massively parallel processing)** 關係型資料庫產品在其支援的查詢型別方面要複雜得多。一個典型的資料倉庫查詢包含多個連線，過濾，分組和聚合操作。MPP 查詢最佳化器將這個複雜的查詢分解成許多執行階段和分割槽，其中許多可以在資料庫叢集的不同節點上並行執行。涉及掃描大規模資料集的查詢特別受益於這種並行執行。

資料倉庫查詢的快速並行執行是一個專門的話題，由於分析有很重要的商業意義，可以帶來很多利益。我們將在 [第十章](#) 討論並行查詢執行的一些技巧。有關並行資料庫中使用的技術的更詳細的概述，請參閱參考文獻【1,33】。

本章小結

在本章中，我們探討了將大資料集劃分成更小的子集的不同方法。資料量非常大的時候，在單臺機器上儲存和處理不再可行，而分割槽則十分必要。分割槽的目標是在多臺機器上均勻分佈資料和查詢負載，避免出現熱點（負載不成比例的節點）。這需要選擇適合於你的資料的分割槽方案，並在將節點新增到叢集或從叢集刪除時重新平衡分割槽。

我們討論了兩種主要的分割槽方法：

- **鍵範圍分割槽**

其中鍵是有序的，並且分割槽擁有從某個最小值到某個最大值的所有鍵。排序的優勢在於可以進行有效的範圍查詢，但是如果應用程式經常訪問相鄰的鍵，則存在熱點的風險。

在這種方法中，當分割槽變得太大的時，通常將分割槽分成兩個子分割槽來動態地重新平衡分割槽。

- **雜湊分割槽**

雜湊函式應用於每個鍵，分割槽擁有一定範圍的雜湊。這種方法破壞了鍵的排序，使得範圍查詢效率低下，但可以更均勻地分配負載。

透過雜湊進行分割槽時，通常先提前建立固定數量的分割槽，為每個節點分配多個分割槽，並在新增或刪除節點時將整個分割槽從一個節點移動到另一個節點。也可以使用動態分割槽。

兩種方法搭配使用也是可行的，例如使用複合主鍵：使用鍵的一部分來標識分割槽，而使用另一部分作為排序順序。

我們還討論了分割槽和次級索引之間的相互作用。次級索引也需要分割槽，有兩種方法：

- 基於文件分割槽（本地索引），其中次級索引儲存在與主鍵和值相同的分割槽中。這意味著只有一個分割槽需要在寫入時更新，但是讀取次級索引需要在所有分割槽之間進行分散 / 收集。
- 基於關鍵詞分割槽（全域性索引），其中次級索引存在不同的分割槽中。次級索引中的條目可以包括來自主鍵的所有分割槽的記錄。當文件寫入時，需要更新多個分割槽中的次級索引；但是可以從單個分割槽中進行讀取。

最後，我們討論了將查詢路由到適當的分割槽的技術，從簡單的分割槽負載平衡到複雜的並行查詢執行引擎。

按照設計，多數情況下每個分割槽是獨立執行的 — 這就是分割槽資料庫可以伸縮到多臺機器的原因。但是，需要寫入多個分割槽的操作結果可能難以預料：例如，如果寫入一個分割槽成功，但另一個分割槽失敗，會發生什麼情況？我們將在下面的章節中討論這個問題。

參考文獻

1. David J. DeWitt and Jim N. Gray: “Parallel Database Systems: The Future of High Performance Database Systems,” *Communications of the ACM*, volume 35, number 6, pages 85–98, June 1992.
doi:10.1145/129888.129894
2. Lars George: “HBase vs. BigTable Comparison,” *larsgeorge.com*, November 2009.
3. “The Apache HBase Reference Guide,” Apache Software Foundation, *hbase.apache.org*, 2014.
4. MongoDB, Inc.: “New Hash-Based Sharding Feature in MongoDB 2.4,” *blog.mongodb.org*, April 10, 2013.
5. Ikai Lan: “App Engine Datastore Tip: Monotonically Increasing Values Are Bad,” *ikaisays.com*, January 25, 2011.
6. Martin Kleppmann: “Java’s hashCode Is Not Safe for Distributed Systems,” *martin.kleppmann.com*, June 18, 2012.
7. David Karger, Eric Lehman, Tom Leighton, et al.: “Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web,” at *29th Annual ACM Symposium on Theory of Computing* (STOC), pages 654–663, 1997. doi:10.1145/258533.258660
8. John Lamping and Eric Veach: “A Fast, Minimal Memory, Consistent Hash Algorithm,” *arxiv.org*, June 2014.
9. Eric Redmond: “A Little Riak Book,” Version 1.4.0, Basho Technologies, September 2013.
10. “Couchbase 2.5 Administrator Guide,” Couchbase, Inc., 2014.
11. Avinash Lakshman and Prashant Malik: “Cassandra – A Decentralized Structured Storage System,” at *3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware* (LADIS), October 2009.
12. Jonathan Ellis: “Facebook’s Cassandra Paper, Annotated and Compared to Apache Cassandra 2.0,” *datastax.com*, September 12, 2013.
13. “Introduction to Cassandra Query Language,” DataStax, Inc., 2014.
14. Samuel Axon: “3% of Twitter’s Servers Dedicated to Justin Bieber,” *mashable.com*, September 7, 2010.
15. “Riak 1.4.8 Docs,” Basho Technologies, Inc., 2014.
16. Richard Low: “The Sweet Spot for Cassandra Secondary Indexing,” *wentnet.com*, October 21, 2013.
17. Zachary Tong: “Customizing Your Document Routing,” *elasticsearch.org*, June 3, 2013.
18. “Apache Solr Reference Guide,” Apache Software Foundation, 2014.
19. Andrew Pavlo: “H-Store Frequently Asked Questions,” *hstore.cs.brown.edu*, October 2013.
20. “Amazon DynamoDB Developer Guide,” Amazon Web Services, Inc., 2014.
21. Rusty Klophaus: “Difference Between 2I and Search,” email to *riak-users* mailing list, *lists.basho.com*, October 25, 2011.
22. Donald K. Burleson: “Object Partitioning in Oracle,” *dba-oracle.com*, November 8, 2000.
23. Eric Evans: “Rethinking Topology in Cassandra,” at *ApacheCon Europe*, November 2012.
24. Rafal Kuć: “Reroute API Explained,” *elasticsearchserverbook.com*, September 30, 2013.
25. “Project Voldemort Documentation,” *project-voldemort.com*.
26. Enis Soztutar: “Apache HBase Region Splitting and Merging,” *hortonworks.com*, February 1, 2013.
27. Brandon Williams: “Virtual Nodes in Cassandra 1.2,” *datastax.com*, December 4, 2012.
28. Richard Jones: “libketama: Consistent Hashing Library for Memcached Clients,” *metabrew.com*, April 10, 2007.
29. Branimir Lambov: “New Token Allocation Algorithm in Cassandra 3.0,” *datastax.com*, January 28, 2016.
30. Jason Wilder: “Open-Source Service Discovery,” *jasonwilder.com*, February 2014.
31. Kishore Gopalakrishna, Shi Lu, Zhen Zhang, et al.: “Untangling Cluster Management with Helix,” at *ACM Symposium on Cloud Computing* (SoCC), October 2012. doi:10.1145/2391229.2391248
32. “Moxi 1.8 Manual,” Couchbase, Inc., 2014.
33. Shivnath Babu and Herodotos Herodotou: “Massively Parallel Databases and MapReduce Systems,” *Foundations and Trends in Databases*, volume 5, number 1, pages 1–104, November 2013. doi:10.1561/1900000036

上一章	目錄	下一章
第五章：複製	設計資料密集型應用	第七章：事務

第七章：事務



一些作者聲稱，支援通用的兩階段提交代價太大，會帶來效能與可用性的問題。讓程式設計師來處理過度使用事務導致的效能問題，總比缺少事務程式設計好得多。

—— James Corbett 等人，Spanner：Google 的全球分散式資料庫（2012）

[TOC]

在資料系統的殘酷現實中，很多事情都可能出錯：

- 資料庫軟體、硬體可能在任意時刻發生故障（包括寫操作進行到一半時）。
- 應用程式可能在任意時刻崩潰（包括一系列操作的中間）。
- 網路中斷可能會意外切斷資料庫與應用的連線，或資料庫之間的連線。
- 多個客戶端可能會同時寫入資料庫，覆蓋彼此的更改。
- 客戶端可能讀取到無意義的資料，因為資料只更新了一部分。
- 客戶端之間的競爭條件可能導致令人驚訝的錯誤。

為了實現可靠性，系統必須處理這些故障，確保它們不會導致整個系統的災難性故障。但是實現容錯機制工作量巨大。需要仔細考慮所有可能出錯的事情，並進行大量的測試，以確保解決方案真正管用。

數十年來，**事務 (transaction)** 一直是簡化這些問題的首選機制。事務是應用程式將多個讀寫操作組合成一個邏輯單元的一種方式。從概念上講，事務中的所有讀寫操作被視作單個操作來執行：整個事務要麼成功 提交 (commit)，要麼失敗 中止 (abort) 或 回滾 (rollback)。如果失敗，應用程式可以安全地重試。對於事務來說，應用程式的錯誤處理變得簡單多了，因為它不用再擔心部分失敗的情況了，即某些操作成功，某些失敗（無論出於何種原因）。

和事務打交道時間長了，你可能會覺得它顯而易見。但我們不應將其視為理所當然。事務不是天然存在的；它們是為了**簡化應用程式設計模型**而建立的。透過使用事務，應用程式可以自由地忽略某些潛在的錯誤情況和併發問題，因為資料庫會替應用處理好這些。（我們稱之為**安全保證**，即 safety guarantees）。

並不是所有的應用都需要事務，有時候弱化事務保證、或完全放棄事務也是有好處的（例如，為了獲得更高效能或更高可用性）。一些安全屬性也可以在沒有事務的情況下實現。

怎樣知道你是否需要事務？為了回答這個問題，首先需要確切理解事務可以提供的安全保障，以及它們的代價。儘管乍看事務似乎很簡單，但實際上有許多微妙但重要的細節在起作用。

本章將研究許多出錯案例，並探索資料庫用於防範這些問題的演算法。尤其會深入**併發控制**的領域，討論各種可能發生的競爭條件，以及資料庫如何實現**讀已提交 (read committed)**，**快照隔離 (snapshot isolation)** 和 **可序列化 (serializability)** 等隔離級別。

本章同時適用於單機資料庫與分散式資料庫；在 [第八章](#) 中將重點討論僅出現在分散式系統中的特殊挑戰。

事務的棘手概念

現今，幾乎所有的關係型資料庫和一些非關係資料庫都支援**事務**。其中大多數遵循 IBM System R（第一個 SQL 資料庫）在 1975 年引入的風格【1,2,3】。40 年裡，儘管一些實現細節發生了變化，但總體思路大同小異：MySQL、PostgreSQL、Oracle 和 SQL Server 等資料庫中的事務支援與 System R 異乎尋常地相似。

2000 年以後，非關係 (NoSQL) 資料庫開始普及。它們的目標是在關係資料庫的現狀基礎上，透過提供新的資料模型選擇（請參閱 [第二章](#)）並預設包含複製（第五章）和分割槽（第六章）來進一步提升。事務是這次運動的主要犧牲品：這些新一代資料庫中的許多資料庫完全放棄了事務，或者重新定義了這個詞，描述比以前所理解的更弱得多的一套保證【4】。

隨著這種新型分散式資料庫的炒作，人們普遍認為事務是可伸縮性的對立面，任何大型系統都必須放棄事務以保持良好的效能和高可用性【5,6】。另一方面，資料庫廠商有時將事務保證作為“重要應用”和“有價值資料”的基本要求。這兩種觀點都是**純粹的誇張**。

事實並非如此簡單：與其他技術設計選擇一樣，事務有其優勢和侷限性。為了理解這些權衡，讓我們瞭解事務所提供的保證的細節——無論是在正常執行中還是在各種極端（但是現實存在）的情況下。

ACID的含義

事務所提供的安全保證，通常由眾所周知的首字母縮略詞 ACID 來描述，ACID 代表 原子性（Atomicity），一致性（Consistency），隔離性（Isolation）和 永續性（Durability）。它由 Theo Härder 和 Andreas Reuter 於 1983 年提出，旨在為資料庫中的容錯機制建立精確的術語。

但實際上，不同資料庫的 ACID 實現並不相同。例如，我們將會看到，關於 隔離性 的含義就有許多含糊不清【8】。高層次上的想法很美好，但魔鬼隱藏在細節裡。今天，當一個系統聲稱自己“符合 ACID”時，實際上能期待的是什麼保證並不清楚。不幸的是，ACID 現在幾乎已經變成了一個營銷術語。

（不符合 ACID 標準的系統有時被稱為 BASE，它代表 基本可用性（Basically Available），軟狀態（Soft State）和 最終一致性（Eventual consistency）【9】，這比 ACID 的定義更加模糊，似乎 BASE 的唯一合理的定義是“不是 ACID”，即它幾乎可以代表任何你想要的東西。）

讓我們深入瞭解原子性，一致性，隔離性和永續性的定義，這可以讓我們提煉出事務的思想。

原子性

一般來說，原子是指不能分解成小部分的東西。這個詞在計算機的不同領域中意味著相似但又微妙不同的東西。例如，在多執行緒程式設計中，如果一個執行緒執行一個原子操作，這意味著另一個執行緒無法看到該操作的一半結果。系統只能處於操作之前或操作之後的狀態，而不是介於兩者之間的狀態。

相比之下，ACID 的原子性並不是關於 併發（concurrent）的。它並不是在描述如果幾個程序試圖同時訪問相同的資料會發生什麼情況，這種情況包含在 隔離性 中。

ACID 的原子性描述了當客戶想進行多次寫入，但在一些寫操作處理完之後出現故障的情況。例如程序崩潰，網路連線中斷，磁碟變滿或者某種完整性約束被違反。如果這些寫操作被分組到一個原子事務中，並且該事務由於錯誤而不能完成（提交），則該事務將被中止，並且資料庫必須丟棄或撤消該事務中迄今為止所做的任何寫入。

如果沒有原子性，在多處更改進行到一半時發生錯誤，很難知道哪些更改已經生效，哪些沒有生效。該應用程式可以再試一次，但冒著進行兩次相同變更的風險，可能會導致資料重複或錯誤的資料。原子性簡化了這個問題：如果事務被中止（abort），應用程式可以確定它沒有改變任何東西，所以可以安全地重試。

ACID 原子性的定義特徵是：能夠在錯誤時中止事務，丟棄該事務進行的所有寫入變更的能力。或許 可中止性（abortability）是更好的術語，但本書將繼續使用原子性，因為這是慣用詞。

一致性

一致性這個詞被賦予太多含義：

- 在 第五章 中，我們討論了副本一致性，以及非同步複製系統中的最終一致性問題（請參閱 “[複製延遲問題](#)”）。
- [一致性雜湊](#) 是某些系統用於重新分割槽的一種分割槽方法。
- 在 [CAP 定理](#) 中，一致性一詞用於表示 [線性一致性](#)。
- 在 ACID 的上下文中，一致性是指資料庫在應用程式的特定概念中處於“良好狀態”。

很不幸，這一個詞就至少有四種不同的含義。

ACID 一致性的概念是，對資料的一組特定約束必須始終成立，即 不變式（invariants）。例如，在會計系統中，所有賬戶整體上必須借貸相抵。如果一個事務開始於一個滿足這些不變式的有效資料庫，且在事務處理期間的任何寫入操作都保持這種有效性，那麼可以確定，不變式總是滿足的。

但是，一致性的這種概念取決於應用程式對不變式的理解，應用程式負責正確定義它的事務，並保持一致性。這並不是資料庫可以保證的事情：如果你寫入違反不變式的髒資料，資料庫也無法阻止你（一些特定型別的不變式可以由資料庫檢查，例如外來鍵約束或唯一約束，但是一般來說，是應用程式來定義什麼樣的資料是有效的，什麼樣是無效的。——

資料庫只管儲存）。

原子性、隔離性和永續性是資料庫的屬性，而一致性（在 ACID 意義上）是應用程式的屬性。應用可能依賴資料庫的原子性和隔離性來實現一致性，但這並不僅取決於資料庫。因此，字母 C 不屬於 ACIDⁱ。

ⁱ 喬·海勒斯坦 (Joe Hellerstein) 指出，在 Härdler 與 Reuter 的論文中，“ACID 中的 C”是被“扔進去湊縮寫單詞的”【7】，而且那時候大家都不怎麼在乎一致性。 ↪

隔離性

大多數資料庫都會同時被多個客戶端訪問。如果它們各自讀寫資料庫的不同部分，這是沒有問題的，但是如果它們訪問相同的資料庫記錄，則可能會遇到併發問題（競爭條件，即 race conditions）。

圖 7-1 是這類問題的一個簡單例子。假設你有兩個客戶端同時在資料庫中增長一個計數器。（假設資料庫沒有內建的自增操作）每個客戶端需要讀取計數器的當前值，加 1，再回寫新值。圖 7-1 中，因為發生了兩次增長，計數器應該從 42 增至 44；但由於競態條件，實際上只增至 43。

ACID 意義上的隔離性意味著，同時執行的事務是相互隔離的：它們不能相互冒犯。傳統的資料庫教科書將隔離性形式化為可序列化 (Serializability)，這意味著每個事務可以假裝它是唯一在整個資料庫上執行的事務。資料庫確保當多個事務被提交時，結果與它們序列執行（一個接一個）是一樣的，儘管實際上它們可能是併發執行的【10】。

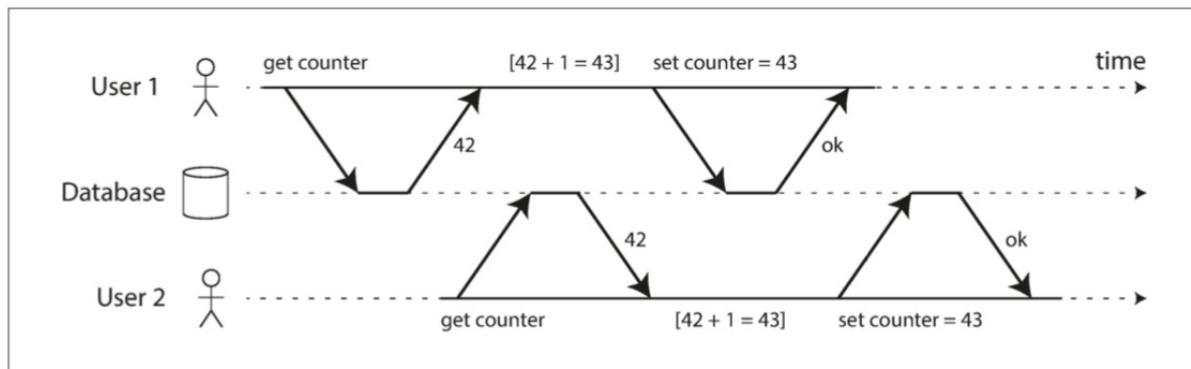


圖 7-1 兩個客戶之間的競爭狀態同時遞增計數器

然而實踐中很少會使用可序列的隔離，因為它有效能損失。一些流行的資料庫如 Oracle 11g，甚至沒有實現它。在 Oracle 中有一個名為“可序列的”隔離級別，但實際上它實現了一種叫做快照隔離 (snapshot isolation) 的功能，這是一種比可序列化更弱的保證【8,11】。我們將在“弱隔離級別”中研究快照隔離和其他形式的隔離。

永續性

資料庫系統的目的是，提供一個安全的地方儲存資料，而不用擔心丟失。永續性是一個承諾，即一旦事務成功完成，即使發生硬體故障或資料庫崩潰，寫入的任何資料也不會丟失。

在單節點資料庫中，永續性通常意味著資料已被寫入非易失性儲存裝置，如硬碟或 SSD。它通常還包括預寫日誌或類似的檔案（請參閱“讓 B 樹更可靠”），以便在磁碟上的資料結構損壞時進行恢復。在帶複製的資料庫中，永續性可能意味著資料已成功複製到一些節點。為了提供永續性保證，資料庫必須等到這些寫入或複製完成後，才能報告事務成功提交。

如“可靠性”一節所述，完美的永續性是不存在的：如果所有硬碟和所有備份同時被銷燬，那顯然沒有任何資料庫能救得了你。

複製與永續性

在歷史上，永續性意味著寫入歸檔磁帶。後來它被理解為寫入磁碟或 SSD。再後來它又有了新的內涵即“複製 (replication)”哪種實現更好一些？

真相是，沒有什麼是完美的：

- 如果你寫入磁碟然後機器宕機，即使資料沒有丟失，在修復機器或將磁碟轉移到其他機器之前，也是無法訪問的。這種情況下，複製系統可以保持可用性。
- 一個相關性故障（停電，或一個特定輸入導致所有節點崩潰的 Bug）可能會一次性摧毀所有副本（請參閱「[可靠性](#)」），任何僅儲存在記憶體中的資料都會丟失，故記憶體資料庫仍然要和磁碟寫入打交道。
- 在非同步複製系統中，當主庫不可用時，最近的寫入操作可能會丟失（請參閱「[處理節點宕機](#)」）。
- 當電源突然斷電時，特別是固態硬碟，有證據顯示有時會違反應有的保證：甚至 fsync 也不能保證正常工作【12】。硬碟韌體可能有錯誤，就像任何其他型別的軟體一樣【13,14】。
- 儲存引擎和檔案系統之間的微妙互動可能會導致難以追蹤的錯誤，並可能導致磁碟上的檔案在崩潰後被損壞【15,16】。
- 磁碟上的資料可能會在沒有檢測到的情況下逐漸損壞【17】。如果資料已損壞一段時間，副本和最近的備份也可能損壞。這種情況下，需要嘗試從歷史備份中恢復資料。
- 一項關於固態硬碟的研究發現，在執行的前四年中，30% 到 80% 的硬碟會產生至少一個壞塊【18】。相比固態硬碟，磁碟的壞道率較低，但完全失效的機率更高。
- 如果 SSD 斷電，可能會在幾周內開始丟失資料，具體取決於溫度【19】。

在實踐中，沒有一種技術可以提供絕對保證。只有各種降低風險的技術，包括寫入磁碟，複製到遠端機器和備份——它們可以且應該一起使用。與往常一樣，最好抱著懷疑的態度接受任何理論上的“保證”。

單物件和多物件操作

回顧一下，在 ACID 中，原子性和隔離性描述了客戶端在同一事務中執行多次寫入時，資料庫應該做的事情：

- **原子性**

如果在一系列寫操作的中途發生錯誤，則應中止事務處理，並丟棄當前事務的所有寫入。換句話說，資料庫免去了使用者對部分失敗的擔憂——透過提供“寧為玉碎，不為瓦全 (**all-or-nothing**)”的保證。

- **隔離性**

同時執行的事務不應該互相干擾。例如，如果一個事務進行多次寫入，則另一個事務要麼看到全部寫入結果，要麼什麼都看不到，但不應該是一些子集。

這些定義假設你想同時修改多個物件（行，文件，記錄）。通常需要 **多物件事務 (multi-object transaction)** 來保持多塊資料同步。圖 7-2 展示了一個來自電郵應用的例子。執行以下查詢來顯示使用者未讀郵件數量：

```
SELECT COUNT (*) FROM emails WHERE recipient_id = 2 AND unread_flag = true
```

但如果郵件太多，你可能會覺得這個查詢太慢，並決定用單獨的欄位儲存未讀郵件的數量（一種反規範化）。現在每當一個新訊息寫入時，必須也增長未讀計數器，每當一個訊息被標記為已讀時，也必須減少未讀計數器。

在 圖 7-2 中，使用者 2 遇到異常情況：郵件列表裡顯示有未讀訊息，但計數器顯示為零未讀訊息，因為計數器增長還沒有發生 ⁱⁱ。隔離性可以避免這個問題：透過確保使用者 2 要麼同時看到新郵件和增長後的計數器，要麼都看不到，而不是一個前後矛盾的中間結果。

ⁱⁱ. 可以說郵件應用中的錯誤計數器並不是什麼特別重要的問題。但換種方式來看，你可以把未讀計數器換成客戶帳戶餘額，把郵件收發看成支付交易。 ↪

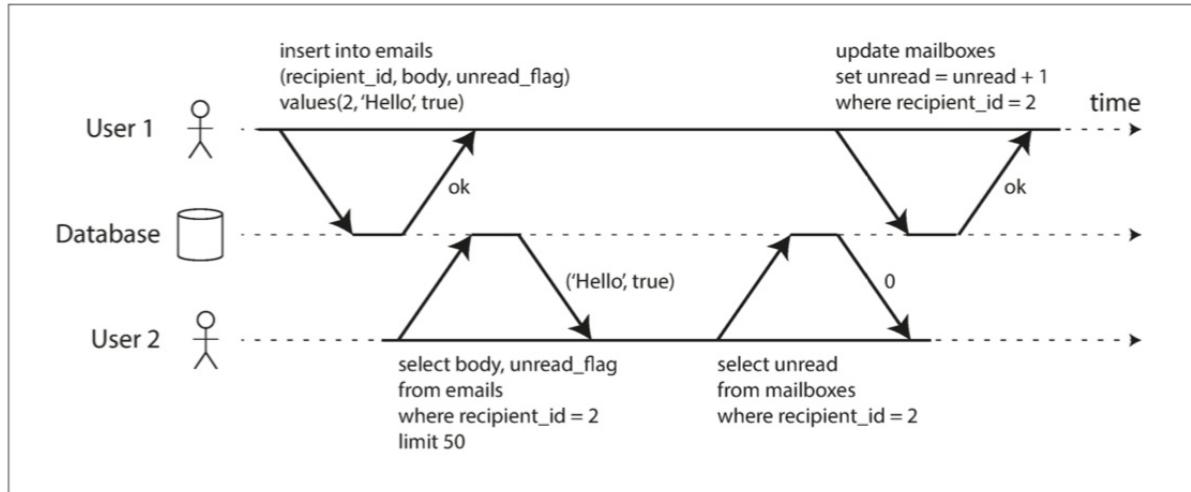


圖 7-2 違反隔離性：一個事務讀取另一個事務的未被執行的寫入（“髒讀”）。

圖 7-3 說明了對原子性的需求：如果在事務過程中發生錯誤，郵箱和未讀計數器的內容可能會失去同步。在原子事務中，如果對計數器的更新失敗，事務將被中止，並且插入的電子郵件將被回滾。

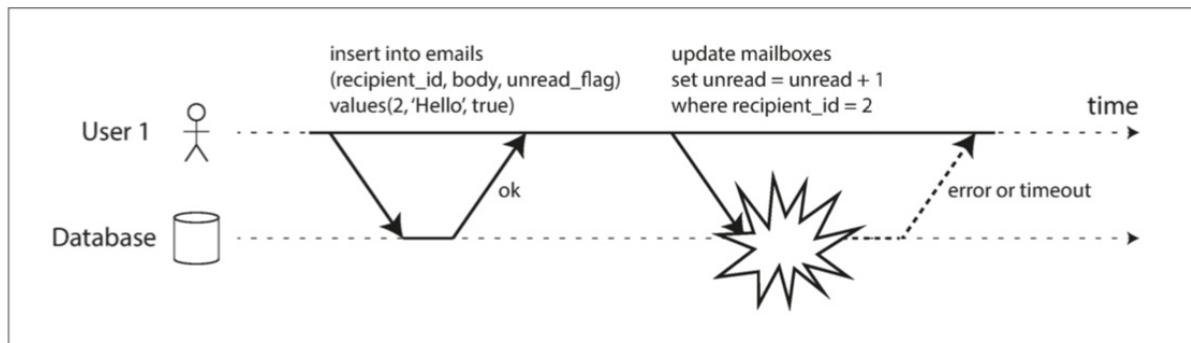


圖 7-3 原子性確保發生錯誤時，事務先前的任何寫入都會被撤消，以避免狀態不一致

多物件事務需要某種方式來確定哪些讀寫操作屬於同一個事務。在關係型資料庫中，通常基於客戶端與資料庫伺服器的 TCP 連線：在任何特定連線上，`BEGIN TRANSACTION` 和 `COMMIT` 語句之間的所有內容，被認為是同一事務的一部分。ⁱⁱⁱ

ⁱⁱⁱ 這並不完美。如果 TCP 連線中斷，則事務必須中止。如果中斷發生在客戶端請求提交之後，但在伺服器確認提交發生之前，客戶端並不知道事務是否已提交。為了解決這個問題，事務管理器可以透過一個唯一事務識別符號來對操作進行分組，這個識別符號並未繫結到特定 TCP 連線。後續再 “[資料庫的端到端原則](#)” 一節將回到這個主題。 ↵

另一方面，許多非關係資料庫並沒有將這些操作組合在一起的方法。即使存在多物件 API（例如，某鍵值儲存可能具有在一個操作中更新幾個鍵的 multi-put 操作），但這並不意味著它具有事務語義：該命令可能在一些鍵上成功，在其他的鍵上失敗，使資料庫處於部分更新的狀態。

單物件寫入

當單個物件發生改變時，原子性和隔離性也是適用的。例如，假設你正在向資料庫寫入一個 20 KB 的 JSON 文件：

- 如果在傳送第一個 10 KB 之後網路連線中斷，資料庫是否儲存了不可解析的 10KB JSON 片段？
- 如果在資料庫正在覆蓋磁碟上的前一個值的過程中電源發生故障，是否最終將新舊值拼接在一起？
- 如果另一個客戶端在寫入過程中讀取該文件，是否會看到部分更新的值？

這些問題非常讓人頭大，故儲存引擎一個幾乎普遍的目標是：對單節點上的單個物件（例如鍵值對）上提供原子性和隔離性。原子性可以透過使用日誌來實現崩潰恢復（請參閱 “[讓 B 樹更可靠](#)”），並且可以使用每個物件上的鎖來實現隔離（每次只允許一個執行緒訪問物件）。

iv

一些資料庫也提供更複雜的原子操作^{iv}，例如自增操作，這樣就不再需要像 圖 7-1 那樣的讀取 - 修改 - 寫入序列了。同樣流行的是 **比較和設定（CAS, compare-and-set）** 操作，僅當值沒有被其他併發修改過時，才允許執行寫操作。

^{iv}. 嚴格地說，原子自增（atomic increment）這個術語在多執行緒程式設計的意義上使用了原子這個詞。在 ACID 的情況下，它實際上應該被稱為隔離的（isolated）或可序列的（serializable）的增量。但這就太吹毛求疵了。 ↪

這些單物件操作很有用，因為它們可以防止在多個客戶端嘗試同時寫入同一個物件時丟失更新（請參閱“[防止丟失更新](#)”）。但它們不是通常意義上的事務。CAS 以及其他單一物件操作被稱為“輕量級事務”，甚至出於營銷目的被稱為“ACID”【20,21,22】，但是這個術語是誤導性的。事務通常被理解為，將多個物件上的多個操作合併為一個執行單元的機制。

多物件事務的需求

許多分散式資料儲存已經放棄了多物件事務，因為多物件事務很難跨分割槽實現，而且在需要高可用性或高效能的情況下，它們可能會礙事。但說到底，在分散式資料庫中實現事務，並沒有什麼根本性的障礙。[第九章](#) 將討論分散式事務的實現。

但是我們是否需要多物件事務？是否有可能只用鍵值資料模型和單物件操作來實現任何應用程式？

有一些場景中，單物件插入、更新和刪除是足夠的。但是許多其他場景需要協調寫入幾個不同的物件：

- 在關係資料模型中，一個表中的行通常具有對另一個表中的行的外來鍵引用。（類似的是，在一個圖資料模型中，一個頂點有著到其他頂點的邊）。多物件事務使你確保這些引用始終有效：當插入幾個相互引用的記錄時，外來鍵必須是正確的和最新的，不然資料就沒有意義。
- 在文件資料模型中，需要一起更新的欄位通常在同一個文件中，這被視為單個物件——更新單個文件時不需要多物件事務。但是，缺乏連線功能的文件資料庫會鼓勵非規範化（請參閱“[關係型資料庫與文件資料庫在今日的對比](#)”）。當需要更新非規範化的資訊時，如 圖 7-2 所示，需要一次更新多個文件。事務在這種情況下非常有用，可以防止非規範化的資料不同步。
- 在具有次級索引的資料庫中（除了純粹的鍵值儲存以外幾乎都有），每次更改值時都需要更新索引。從事務角度來看，這些索引是不同的資料庫物件：例如，如果沒有事務隔離性，記錄可能出現在一個索引中，但沒有出現在另一個索引中，因為第二個索引的更新還沒有發生。

這些應用仍然可以在沒有事務的情況下實現。然而，沒有原子性，錯誤處理就要複雜得多，缺乏隔離性，就會導致併發問題。我們將在“[弱隔離級別](#)”中討論這些問題，並在 [第十二章](#) 中探討其他方法。

處理錯誤和中止

事務的一個關鍵特性是，如果發生錯誤，它可以中止並安全地重試。ACID 資料庫基於這樣的哲學：如果資料庫有違反其原子性、隔離性或永續性的危險，則寧願完全放棄事務，而不是留下半成品。

然而並不是所有的系統都遵循這個哲學。特別是具有 [無主複製](#) 的資料儲存，主要是在“盡力而為”的基礎上進行工作。可以概括為“資料庫將做盡可能多的事，執行遇到錯誤時，它不會撤消它已經完成的事情”——所以，從錯誤中恢復是應用程式的責任。

錯誤發生不可避免，但許多軟體開發人員傾向於只考慮樂觀情況，而不是錯誤處理的複雜性。例如，像 Rails 的 ActiveRecord 和 Django 這樣的 [物件關係對映（ORM, object-relation Mapping）](#) 框架不會重試中斷的事務——這個錯誤通常會導致一個從堆疊向上傳播的異常，所以任何使用者輸入都會被丟棄，使用者拿到一個錯誤資訊。這實在是太恥辱了，因為中止的重點就是允許安全的重試。

儘管重試一箇中止的事務是一個簡單而有效的錯誤處理機制，但它並不完美：

- 如果事務實際上成功了，但是在伺服器試圖向客戶端確認提交成功時網路發生故障（所以客戶端認為提交失敗了），那麼重試事務會導致事務被執行兩次——除非你有一個額外的應用級去重機制。
- 如果錯誤是由於負載過大造成的，則重試事務將使問題變得更糟，而不是更好。為了避免這種正反饋迴圈，可以限制重試次數，使用指數退避演算法，並單獨處理與過載相關的錯誤（如果允許）。

- 僅在臨時性錯誤（例如，由於死鎖，異常情況，臨時性網路中斷和故障切換）後才值得重試。在發生永久性錯誤（例如，違反約束）之後重試是毫無意義的。
- 如果事務在資料庫之外也有副作用，即使事務被中止，也可能發生這些副作用。例如，如果你正在傳送電子郵件，那你肯定不希望每次重試事務時都重新發送電子郵件。如果你想確保幾個不同的系統一起提交或放棄，**兩階段提交（2PC, two-phase commit）** 可以提供幫助（“[原子提交與兩階段提交](#)”中將討論這個問題）。
- 如果客戶端程序在重試中失效，任何試圖寫入資料庫的資料都將丟失。

弱隔離級別

如果兩個事務不觸及相同的資料，它們可以安全地 **並行（parallel）** 執行，因為兩者都不依賴於另一個。當一個事務讀取由另一個事務同時修改的資料時，或者當兩個事務試圖同時修改相同的資料時，併發問題（競爭條件）才會出現。

併發 BUG 很難透過測試找到，因為這樣的錯誤只有在特殊時序下才會觸發。這樣的時序問題可能非常少發生，通常很難重現 [譯註1](#)。併發性也很難推理，特別是在大型應用中，你不一定知道哪些其他程式碼正在訪問資料庫。在一次只有一個使用者時，應用開發已經很麻煩了，有許多併發使用者使得它更加困難，因為任何一個數據都可能隨時改變。

[譯註1](#). 軼事：偶然出現的瞬時錯誤有時稱為 *Heisenbug*，而確定性的問題對應地稱為 *Bohrbugs* ↪

出於這個原因，資料庫一直試圖透過提供 **事務隔離（transaction isolation）** 來隱藏應用程式開發者的併發問題。從理論上講，隔離可以透過假裝沒有併發發生，讓你的生活更加輕鬆：**可序列的（serializable）** 隔離等級意味著資料庫保證事務的效果如同序列執行（即一次一個，沒有任何併發）。

實際上不幸的是：隔離並沒有那麼簡單。**可序列的隔離** 會有效能損失，許多資料庫不願意支付這個代價 [【8】](#)。因此，系統通常使用較弱的隔離級別來防止一部分，而不是全部的併發問題。這些隔離級別難以理解，並且會導致微妙的錯誤，但是它們仍然在實踐中被使用 [【23】](#)。

弱事務隔離級別導致的併發性錯誤不僅僅是一個理論問題。它們造成了很多的資金損失 [【24,25】](#)，耗費了財務審計人員的調查 [【26】](#)，並導致客戶資料被破壞 [【27】](#)。關於這類問題的一個流行的評論是“如果你正在處理財務資料，請使用 ACID 資料庫！”——但是這一點沒有提到。即使是很多流行的關係型資料庫系統（通常被認為是“ACID”）也使用弱隔離級別，所以它們也不一定能防止這些錯誤的發生。

比起盲目地依賴工具，我們需要對存在的各種併發問題，以及如何防止這些問題有深入的理解。然後就可以使用我們所掌握的工具來構建可靠和正確的應用程式。

在本節中，我們將看幾個在實踐中使用的弱（非序列的，即 nonserializable）隔離級別，並詳細討論哪種競爭條件可能發生也可能不發生，以便你可以決定什麼級別適合你的應用程式。一旦我們完成了這個工作，我們將詳細討論可序列化（請參閱“[可序列化](#)”）。我們討論的隔離級別將是非正式的，透過示例來進行。如果你需要嚴格的定義和分析它們的屬性，你可以在學術文獻中找到它們 [【28,29,30】](#)。

讀已提交

最基本的事務隔離級別是 **讀已提交（Read Committed）** [▼](#)，它提供了兩個保證：

1. 從資料庫讀時，只能看到已提交的資料（沒有 髒讀，即 dirty reads）。
2. 寫入資料庫時，只會覆蓋已提交的資料（沒有 髒寫，即 dirty writes）。

我們來更詳細地討論這兩個保證。

[▼](#). 某些資料庫支援甚至更弱的隔離級別，稱為 **讀未提交（Read uncommitted）**。它可以防止髒寫，但不防止髒讀。 ↪

沒有髒讀

設想一個事務已經將一些資料寫入資料庫，但事務還沒有提交或中止。另一個事務可以看到未提交的資料嗎？如果是的話，那就叫做 **髒讀（dirty reads）** [【2】](#)。

在 讀已提交 隔離級別執行的事務必須防止髒讀。這意味著事務的任何寫入操作只有在該事務提交時才能被其他人看到（然後所有的寫入操作都會立即變得可見）。如 圖 7-4 所示，使用者 1 設定了 `x = 3`，但使用者 2 的 `get x` 仍舊返回舊值 2（當用戶 1 尚未提交時）。

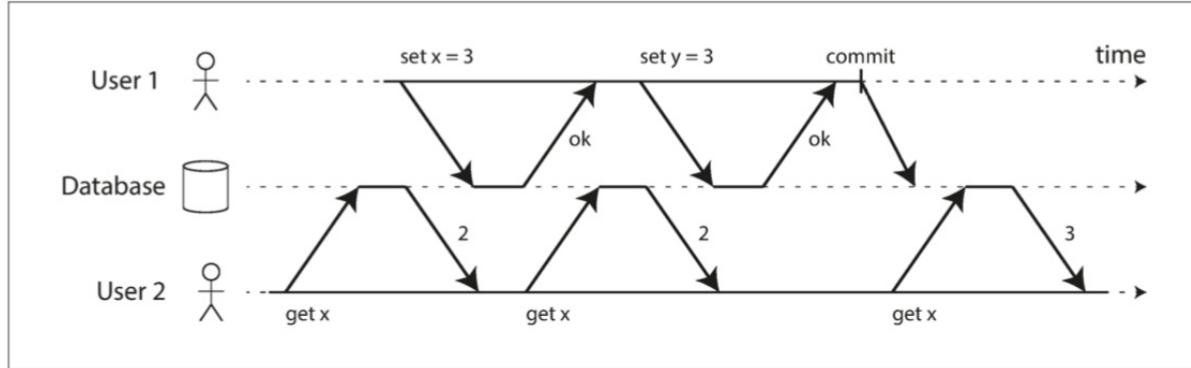


圖 7-4 沒有髒讀：使用者 2 只有在使用者 1 的事務已經提交後才能看到 x 的新值。

為什麼要防止髒讀，有幾個原因：

- 如果事務需要更新多個物件，髒讀取意味著另一個事務可能會只看到一部分更新。例如，在 圖 7-2 中，使用者看到新的未讀電子郵件，但看不到更新的計數器。這就是電子郵件的髒讀。看到處於部分更新狀態的資料庫會讓使用者感到困惑，並可能導致其他事務做出錯誤的決定。
- 如果事務中止，則所有寫入操作都需要回滾（如 圖 7-3 所示）。如果資料庫允許髒讀，那就意味著一個事務可能會看到稍後需要回滾的資料，即從未實際提交給資料庫的資料。想想後果就讓人頭大。

沒有髒寫

如果兩個事務同時嘗試更新資料庫中的相同物件，會發生什麼情況？我們不知道寫入的順序是怎樣的，但是我們通常認為後面的寫入會覆蓋前面的寫入。

但是，如果先前的寫入是尚未提交事務的一部分，又會發生什麼情況，後面的寫入會覆蓋一個尚未提交的值？這被稱作髒寫（dirty write）【28】。在 讀已提交 的隔離級別上執行的事務必須防止髒寫，通常是延遲第二次寫入，直到第一次寫入事務提交或中止為止。

透過防止髒寫，這個隔離級別避免了一些併發問題：

- 如果事務更新多個物件，髒寫會導致不好的結果。例如，考慮 圖 7-5，以一個二手車銷售網站為例，Alice 和 Bob 兩個人同時試圖購買同一輛車。購買汽車需要兩次資料庫寫入：網站上的商品列表需要更新，以反映買家的購買，銷售發票需要傳送給買家。在 圖 7-5 的情況下，銷售是屬於 Bob 的（因為他成功更新了商品列表），但發票卻寄送給了 Alice（因為她成功更新了發票表）。讀已提交會防止這樣的事故。
- 但是，讀已提交併不能防止 圖 7-1 中兩個計數器增量之間的競爭狀態。在這種情況下，第二次寫入發生在第一個事務提交後，所以它不是一個髒寫。這仍然是不正確的，但是出於不同的原因，在“[防止丢失更新](#)”中將討論如何使這種計數器增量安全。

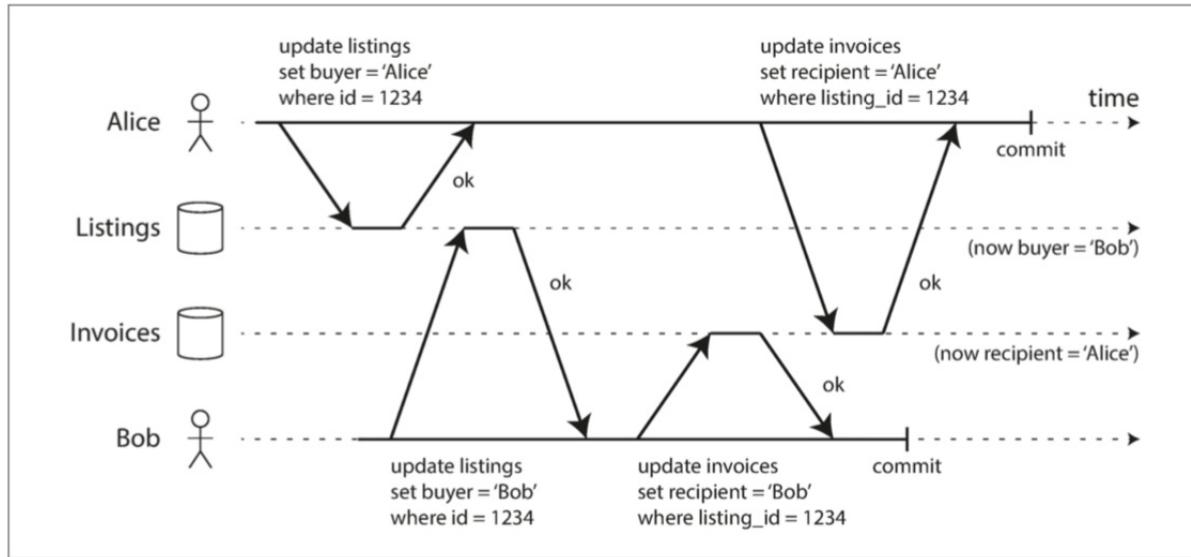


圖 7-5 如果存在髒寫，來自不同事務的衝突寫入可能會混淆在一起

實現讀已提交

讀已提交 是一個非常流行的隔離級別。這是 Oracle 11g、PostgreSQL、SQL Server 2012、MemSQL 和其他許多資料庫的預設設定【8】。

最常見的情況是，資料庫透過使用 行鎖（row-level lock）來防止髒寫：當事務想要修改特定物件（行或文件）時，它必須首先獲得該物件的鎖。然後必須持有該鎖直到事務被提交或中止。一次只有一個事務可持有任何給定物件的鎖；如果另一個事務要寫入同一個物件，則必須等到第一個事務提交或中止後，才能獲取該鎖並繼續。這種鎖定是讀已提交模式（或更強的隔離級別）的資料庫自動完成的。

如何防止髒讀？一種選擇是使用相同的鎖，並要求任何想要讀取物件的事務來簡單地獲取該鎖，然後在讀取之後立即再次釋放該鎖。這將確保在物件具有髒的、未提交的值時不會發生讀取（因為在此期間，鎖將由進行寫入的事務持有）。

但是要求讀鎖的辦法在實踐中效果並不好。因為一個長時間執行的寫入事務會迫使許多隻讀事務等到這個慢寫入事務完成。這會影響只讀事務的響應時間，並且不利於可操作性：因為等待鎖，應用某個部分的遲緩可能由於連鎖效應，導致其他部分出現問題。

出於這個原因，大多數資料庫^{vi} 使用 圖 7-4 的方式防止髒讀：對於寫入的每個物件，資料庫都會記住舊的已提交值，和由當前持有寫入鎖的事務設定的新值。當事務正在進行時，任何其他讀取物件的事務都會拿到舊值。只有當新值提交後，事務才會切換到讀取新值。

^{vi} 在撰寫本文時，唯一在讀已提交隔離級別使用讀鎖的主流資料庫是 IBM DB2 和使用 `read_committed_snapshot = off` 配置的 Microsoft SQL Server 【23,36】。 ↪

快照隔離和可重複讀

如果只從表面上看讀已提交隔離級別，你可能就認為它完成了事務所需的一切，這是情有可原的。它允許 中止（原子性的要求）；它防止讀取不完整的事務結果，並且防止併發寫入造成的混亂。事實上這些功能非常有用，比起沒有事務的系統來，可以提供更多的保證。

但是在使用此隔離級別時，仍然有很多地方可能會產生併發錯誤。例如 圖 7-6 說明了讀已提交時可能發生的問題。

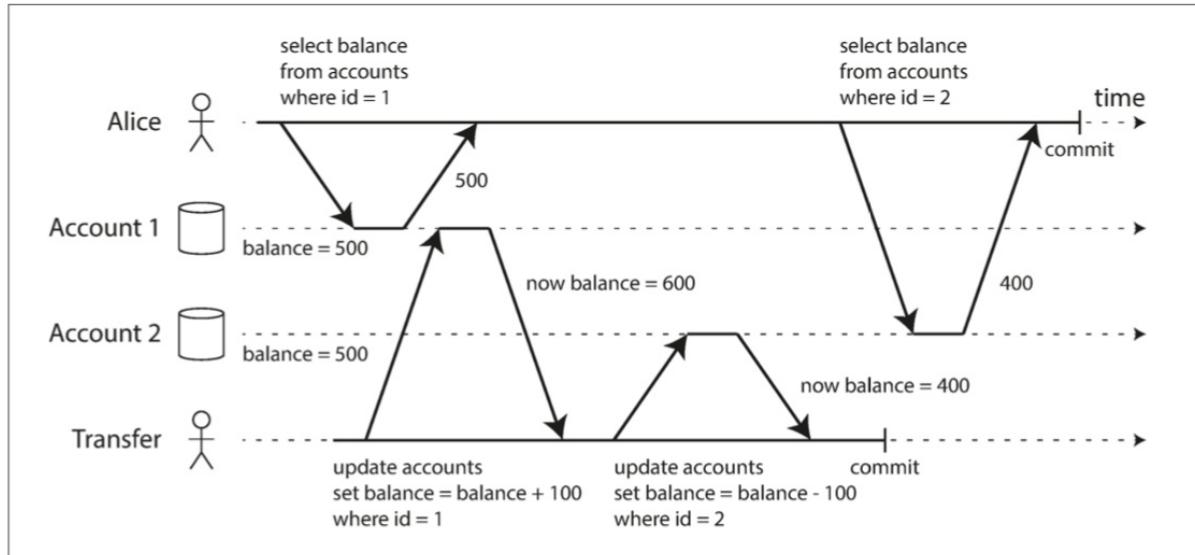


圖 7-6 讀取偏差：Alice 觀察資料庫處於不一致的狀態

Alice 在銀行有 1000 美元的儲蓄，分為兩個賬戶，每個 500 美元。現在有一筆事務從她的一個賬戶轉移了 100 美元到另一個賬戶。如果她非常不幸地在事務處理的過程中檢視其賬戶餘額列表，她可能會在收到付款之前先看到一個賬戶的餘額（收款賬戶，餘額仍為 500 美元），在發出轉賬之後再看到另一個賬戶的餘額（付款賬戶，新餘額為 400 美元）。對 Alice 來說，現在她的賬戶似乎總共只有 900 美元——看起來有 100 美元已經憑空消失了。

這種異常被稱為 **不可重複讀 (nonrepeatable read)** 或 **讀取偏差 (read skew)**：如果 Alice 在事務結束時再次讀取賬戶 1 的餘額，她將看到與她之前的查詢中看到的不同的值（600 美元）。在讀已提交的隔離條件下，**不可重複讀** 被認為是可接受的：Alice 看到的帳戶餘額確實在閱讀時已經提交了。

不幸的是，術語 **偏差 (skew)** 這個詞是過載的：以前使用它是因為熱點的不平衡工作量（請參閱“[負載偏斜與熱點消除](#)”），而這裡偏差意味著異常的時序。

對於 Alice 的情況，這不是一個長期持續的問題。因為如果她幾秒鐘後重新整理銀行網站的頁面，她很可能會看到一致的帳戶餘額。但是有些情況下，不能容忍這種暫時的不一致：

- 備份

進行備份需要複製整個資料庫，對大型資料庫而言可能需要花費數小時才能完成。備份程序執行時，資料庫仍然會接受寫入操作。因此備份可能會包含一些舊的部分和一些新的部分。如果從這樣的備份中恢復，那麼不一致（如消失的錢）就會變成永久的。

- 分析查詢和完整性檢查

有時，你可能需要執行一個查詢，掃描大部分的資料庫。這樣的查詢在分析中很常見（請參閱“[事務處理還是分析？](#)”），也可能是定期完整性檢查（即監視資料損壞）的一部分。如果這些查詢在不同時間點觀察資料庫的不同部分，則可能會返回毫無意義的結果。

快照隔離 (snapshot isolation) 【28】是這個問題最常見的解決方案。想法是，每個事務都從資料庫的 **一致快照 (consistent snapshot)** 中讀取——也就是說，事務可以看到事務開始時在資料庫中提交的所有資料。即使這些資料隨後被另一個事務更改，每個事務也只能看到該特定時間點的舊資料。

快照隔離對長時間執行的只讀查詢（如備份和分析）非常有用。如果查詢的資料在查詢執行的同時發生變化，則很難理解查詢的含義。當一個事務可以看到資料庫在某個特定時間點凍結時的一致快照，理解起來就很容易了。

快照隔離是一個流行的功能：PostgreSQL、使用 InnoDB 引擎的 MySQL、Oracle、SQL Server 等都支援【23,31,32】。

實現快照隔離

與讀取提交的隔離類似，快照隔離的實現通常使用寫鎖來防止髒寫（請參閱“[讀已提交](#)”），這意味著進行寫入的事務會阻止另一個事務修改同一個物件。但是讀取則不需要加鎖。從效能的角度來看，快照隔離的一個關鍵原則是：**讀不阻塞寫，寫不阻塞讀**。這允許資料庫在處理一致性快照上的長時間查詢時，可以正常地同時處理寫入操作，且兩者間沒有任何鎖爭用。

為了實現快照隔離，資料庫使用了我們看到的用於防止 [圖 7-4](#) 中的髒讀的機制的一般化。資料庫必須可能保留一個物件的幾個不同的提交版本，因為各種正在進行的事務可能需要看到資料庫在不同的時間點的狀態。因為它同時維護著單個物件的多個版本，所以這種技術被稱為 **多版本併發控制 (MVCC, multi-version concurrency control)**。

如果一個數據庫只需要提供 [讀已提交](#) 的隔離級別，而不提供 [快照隔離](#)，那麼保留一個物件的兩個版本就足夠了：已提交的版本和被覆蓋但尚未提交的版本。不過支援快照隔離的儲存引擎通常也使用 MVCC 來實現 [讀已提交](#) 隔離級別。一種典型的方法是 [讀已提交](#) 為每個查詢使用單獨的快照，而 [快照隔離](#) 對整個事務使用相同的快照。

[圖 7-7](#) 說明了 PostgreSQL 如何實現基於 MVCC 的快照隔離 [31]（其他實現類似）。當一個事務開始時，它被賦予一個唯一的，永遠增長 ^{vii} 的事務 ID (`txid`)。每當事務向資料庫寫入任何內容時，它所寫入的資料都會被標記上寫入者的事務 ID。

^{vii} 事實上，事務 ID 是 32 位整數，所以大約會在 40 億次事務之後溢位。PostgreSQL 的 Vacuum 過程會清理老舊的事務 ID，確保事務 ID 溢位（回捲）不會影響到資料。 ↩

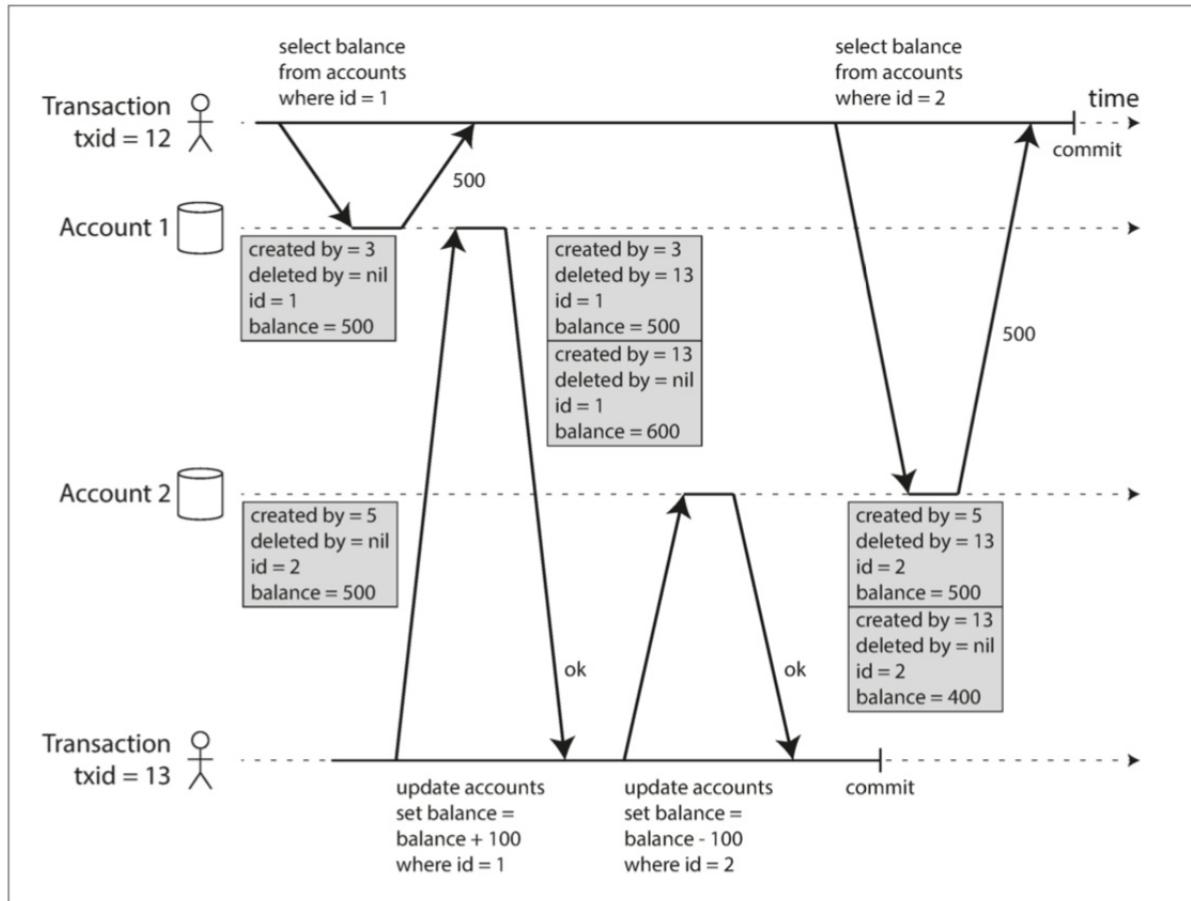


圖 7-7 使用多版本物件實現快照隔離

表中的每一行都有一個 `created_by` 欄位，其中包含將該行插入到表中的的事務 ID。此外，每行都有一個 `deleted_by` 欄位，最初是空的。如果某個事務刪除了一行，那麼該行實際上並未從資料庫中刪除，而是透過將 `deleted_by` 欄位設定為請求刪除的事務的 ID 來標記為刪除。在稍後的時間，當確定沒有事務可以再訪問已刪除的資料時，資料庫中的垃圾收集過程會將所有帶有刪除標記的行移除，並釋放其空間。[譯註ii](#)

[譯註ii](#) 在 PostgreSQL 中，`created_by` 的實際名稱為 `xmin`，`deleted_by` 的實際名稱為 `xmax` ↩

`UPDATE` 操作在內部翻譯為 `DELETE` 和 `INSERT`。例如，在圖 7-7 中，事務 13 從賬戶 2 中扣除 100 美元，將餘額從 500 美元改為 400 美元。實際上包含兩條賬戶 2 的記錄：餘額為 \$500 的行被標記為 被事務 13 刪除，餘額為 \$400 的行由事務 13 建立。

觀察一致性快照的可見性規則

當一個事務從資料庫中讀取時，事務 ID 用於決定它可以看見哪些物件，看不見哪些物件。透過仔細定義可見性規則，資料庫可以嚮應用程式呈現一致的資料庫快照。工作如下：

1. 在每次事務開始時，資料庫列出當時所有其他（尚未提交或尚未中止）的事務清單，即使之後提交了，這些事務已執行的任何寫入也都會被忽略。
2. 被中止事務所執行的任何寫入都將被忽略。
3. 由具有較晚事務 ID（即，在當前事務開始之後開始的）的事務所做的任何寫入都被忽略，而不管這些事務是否已經提交。
4. 所有其他寫入，對應用都是可見的。

這些規則適用於建立和刪除物件。在圖 7-7 中，當事務 12 從賬戶 2 讀取時，它會看到 \$500 的餘額，因為 \$500 餘額的刪除是由事務 13 完成的（根據規則 3，事務 12 看不到事務 13 執行的刪除），且 400 美元記錄的建立也是不可見的（按照相同的規則）。

換句話說，如果以下兩個條件都成立，則可見一個物件：

- 讀事務開始時，建立該物件的事務已經提交。
- 牆件未被標記為刪除，或如果被標記為刪除，請求刪除的事務在讀事務開始時尚未提交。

長時間執行的事務可能會長時間使用快照，並繼續讀取（從其他事務的角度來看）早已被覆蓋或刪除的值。由於從來不原地更新值，而是每次值改變時建立一個新的版本，資料庫可以在提供一致快照的同時只產生很小的額外開銷。

索引和快照隔離

索引如何在多版本資料庫中工作？一種選擇是使索引簡單地指向物件的所有版本，並且需要索引查詢來過濾掉當前事務不可見的任何物件版本。當垃圾收集刪除任何事務不再可見的舊物件版本時，相應的索引條目也可以被刪除。

在實踐中，許多實現細節決定了多版本併發控制的效能。例如，如果同一物件的不同版本可以放入同一個頁面中，PostgreSQL 的最佳化可以避免更新索引【31】。

在 CouchDB、Datomic 和 LMDB 中使用另一種方法。雖然它們也使用 B 樹，但它們使用的是一種 **僅追加 / 寫時複製 (append-only/copy-on-write)** 的變體，它們在更新時不覆蓋樹的頁面，而為每個修改頁面建立一份副本。從父頁面直到樹根都會級聯更新，以指向它們子頁面的新版本。任何不受寫入影響的頁面都不需要被複制，並且保持不變【33,34,35】。

使用僅追加的 B 樹，每個寫入事務（或一批事務）都會建立一棵新的 B 樹，當建立時，從該特定樹根生長的樹就是資料庫的一個一致性快照。沒必要根據事務 ID 過濾掉物件，因為後續寫入不能修改現有的 B 樹；它們只能建立新的樹根。但這種方法也需要一個負責壓縮和垃圾收集的後臺程序。

可重複讀與命名混淆

快照隔離是一個有用的隔離級別，特別對於只讀事務而言。但是，許多資料庫實現了它，卻用不同的名字來稱呼。在 Oracle 中稱為 **可序列化 (Serializable)** 的，在 PostgreSQL 和 MySQL 中稱為 **可重複讀 (repeatable read)**【23】。

這種命名混淆的原因是 SQL 標準沒有 **快照隔離** 的概念，因為標準是基於 System R 1975 年定義的隔離級別【2】，那時候 **快照隔離** 尚未發明。相反，它定義了 **可重複讀**，表面上看起來與快照隔離很相似。PostgreSQL 和 MySQL 稱其 **快照隔離** 級別為 **可重複讀 (repeatable read)**，因為這樣符合標準要求，所以它們可以聲稱自己“標準相容”。

不幸的是，SQL 標準對隔離級別的定義是有缺陷的——模糊，不精確，並不像標準應有的樣子獨立於實現【28】。有幾個資料庫實現了可重複讀，但它們實際提供的保證存在很大的差異，儘管表面上是標準化的【23】。在研究文獻【29,30】中已經有了可重複讀的正式定義，但大多數的實現並不能滿足這個正式定義。最後，IBM DB2 使用“可重複讀”來引用可序列化【8】。

結果，沒有人真正知道 可重複讀 的意思。

防止丟失更新

到目前為止已經討論的 讀已提交 和 快照隔離 級別，主要保證了 只讀事務在併發寫入時 可以看到什麼。卻忽略了兩個事務併發寫入的問題——我們只討論了髒寫（請參閱 “[沒有髒寫](#)”），一種特定型別的寫 - 寫衝突是可能出現的。

併發的寫入事務之間還有其他幾種有趣的衝突。其中最著名的是 丟失更新（lost update） 問題，如 [圖 7-1](#) 所示，以兩個併發計數器增量為例。

如果應用從資料庫中讀取一些值，修改它並寫回修改的值（讀取 - 修改 - 寫入序列），則可能會發生丟失更新的問題。如果兩個事務同時執行，則其中一個的修改可能會丟失，因為第二個寫入的內容並沒有包括第一個事務的修改（有時會說後面寫入 狠摟（clobber） 了前面的寫入）這種模式發生在各種不同的情況下：

- 增加計數器或更新賬戶餘額（需要讀取當前值，計算新值並寫回更新後的值）
- 將本地修改寫入一個複雜值中：例如，將元素新增到 JSON 文件中的一個列表（需要解析文件，進行更改並寫回修改的文件）
- 兩個使用者同時編輯 wiki 頁面，每個使用者透過將整個頁面內容傳送到伺服器來儲存其更改，覆寫資料庫中當前的任何內容。

這是一個普遍的問題，所以已經開發了各種解決方案。

原子寫

許多資料庫提供了原子更新操作，從而消除了在應用程式程式碼中執行讀取 - 修改 - 寫入序列的需要。如果你的程式碼可以用這些操作來表達，那這通常是最好的解決方案。例如，下面的指令在大多數關係資料庫中是併發安全的：

```
UPDATE counters SET value = value + 1 WHERE key = 'foo';
```

類似地，像 MongoDB 這樣的文件資料庫提供了對 JSON 文件的一部分進行本地修改的原子操作，Redis 提供了修改資料結構（如優先順序佇列）的原子操作。並不是所有的寫操作都可以用原子操作的方式來表達，例如 wiki 頁面的更新涉及到任意文字編輯 [viii](#)，但是在可以使用原子操作的情況下，它們通常是最好的選擇。

[viii](#). 將文字文件的編輯表示為原子的變化流是可能的，儘管相當複雜。請參閱 “[自動衝突解決](#)”。 ↩

原子操作通常透過在讀取物件時，獲取其上的排它鎖來實現。以便更新完成之前沒有其他事務可以讀取它。這種技術有時被稱為 遊標穩定性（cursor stability）【36,37】。另一個選擇是簡單地強制所有的原子操作在單一執行緒上執行。

不幸的是，ORM 框架很容易意外地執行不安全的讀取 - 修改 - 寫入序列，而不是使用資料庫提供的原子操作【38】。如果你知道自己在做什麼那當然不是問題，但它經常產生那種很難測出來的微妙 Bug。

顯式鎖定

如果資料庫的內建原子操作沒有提供必要的功能，防止丟失更新的另一個選擇是讓應用程式顯式地鎖定將要更新的物件。然後應用程式可以執行讀取 - 修改 - 寫入序列，如果任何其他事務嘗試同時讀取同一個物件，則強制等待，直到第一個 讀取 - 修改 - 寫入序列 完成。

例如，考慮一個多人遊戲，其中幾個玩家可以同時移動相同的棋子。在這種情況下，一個原子操作可能是不夠的，因為應用程式還需要確保玩家的移動符合遊戲規則，這可能涉及到一些不能合理地用資料庫查詢實現的邏輯。但你可以使用鎖來防止兩名玩家同時移動相同的棋子，如例 7-1 所示。

例 7-1 顯式鎖定行以防止丢失更新

```
BEGIN TRANSACTION;
SELECT * FROM figures
WHERE name = 'robot' AND game_id = 222
FOR UPDATE;

-- 檢查玩家的操作是否有效，然後更新先前 SELECT 返回棋子的位置。
UPDATE figures SET position = 'c4' WHERE id = 1234;
COMMIT;
```

- FOR UPDATE 子句告訴資料庫應該對該查詢返回的所有行加鎖。

這是有效的，但要做對，你需要仔細考慮應用邏輯。忘記在程式碼某處加鎖很容易引入競爭條件。

自動檢測丟失的更新

原子操作和鎖是透過強制 讀取 - 修改 - 寫入序列 按順序發生，來防止丢失更新的方法。另一種方法是允許它們並行執行，如果事務管理器檢測到丢失更新，則中止事務並強制它們重試其 讀取 - 修改 - 寫入序列。

這種方法的一個優點是，資料庫可以結合快照隔離高效地執行此檢查。事實上，PostgreSQL 的可重複讀，Oracle 的可序列化和 SQL Server 的快照隔離級別，都會自動檢測到丢失更新，並中止惹麻煩的事務。但是，MySQL/InnoDB 的可重複讀並不會檢測 丢失更新【23】。一些作者【28,30】認為，資料庫必須能防止丢失更新才稱得上是提供了 快照隔離，所以在這個定義下，MySQL 下不提供快照隔離。

丢失更新檢測是一個很好的功能，因為它不需要應用程式碼使用任何特殊的資料庫功能，你可能會忘記使用鎖或原子操作，從而引入錯誤；但丢失更新的檢測是自動發生的，因此不太容易出錯。

比較並設定（CAS）

在不提供事務的資料庫中，有時會發現一種原子操作：比較並設定（CAS，即 Compare And Set，先前在“[單物件寫入](#)”中提到）。此操作的目的是為了避免丢失更新：只有當前值從上次讀取時一直未改變，才允許更新發生。如果當前值與先前讀取的值不匹配，則更新不起作用，且必須重試讀取 - 修改 - 寫入序列。

例如，為了防止兩個使用者同時更新同一個 wiki 頁面，可以嘗試類似這樣的方式，只有當用戶開始編輯後頁面內容未發生改變時，才會更新成功：

```
-- 根據資料庫的實現情況，這可能安全也可能不安全
UPDATE wiki_pages SET content = '新內容'
WHERE id = 1234 AND content = '舊內容';
```

如果內容已經更改並且不再與“舊內容”相匹配，則此更新將不起作用，因此你需要檢查更新是否生效，必要時重試。但是，如果資料庫允許 WHERE 子句從舊快照中讀取，則此語句可能無法防止丢失更新，因為即使發生了另一個併發寫入， WHERE 條件也可能為真。在依賴資料庫的 CAS 操作前要檢查其是否安全。

衝突解決和複製

在複製資料庫中（請參閱 [第五章](#)），防止丢失的更新需要考慮另一個維度：由於在多個節點上存在資料副本，並且在不同節點上的資料可能被併發地修改，因此需要採取一些額外的步驟來防止丢失更新。

鎖和 CAS 操作假定只有一個最新的資料副本。但是多主或無主複製的資料庫通常允許多個寫入併發執行，並非同步複製到副本上，因此無法保證只有一個最新資料的副本。所以基於鎖或 CAS 操作的技術不適用於這種情況（我們將在“[線性一致性](#)”中更詳細地討論這個問題）。

相反，如“[檢測併發寫入](#)”一節所述，這種複製資料庫中的一種常見方法是允許併發寫入建立多個衝突版本的值（也稱為兄弟），並使用應用程式碼或特殊資料結構在事實發生之後解決和合並這些版本。

原子操作可以在複製的上下文中很好地工作，尤其當它們具有可交換性時（即，可以在不同的副本上以不同的順序應用它們，且仍然可以得到相同的結果）。例如，遞增計數器或向集合新增元素是可交換的操作。這是 Riak 2.0 資料型別背後的思想，它可以防止複製副本丟失更新。當不同的客戶端同時更新一個值時，Riak 自動將更新合併在一起，以免丟失更新【39】。

另一方面，最後寫入勝利（LWW）的衝突解決方法很容易丟失更新，如“[最後寫入勝利（丟棄併發寫入）](#)”中所述。不幸的是，LWW 是許多複製資料庫中的預設方案。

寫入偏差與幻讀

前面的章節中，我們看到了 [髒寫](#) 和 [丟失更新](#)，當不同的事務併發地嘗試寫入相同的物件時，會出現這兩種競爭條件。為了避免資料損壞，這些競爭條件需要被阻止——既可以由資料庫自動執行，也可以透過鎖和原子寫操作這類手動安全措施來防止。

但是，併發寫入間可能發生的競爭條件還沒有完。在本節中，我們將看到一些更微妙的衝突例子。

首先，想像一下這個例子：你正在為醫院寫一個醫生輪班管理程式。醫院通常會同時要求幾位醫生待命，但底線是至少有一位醫生在待命。醫生可以放棄他們的班次（例如，如果他們自己生病了），只要至少有一個同事在這一班中繼續工作【40,41】。

現在想像一下，Alice 和 Bob 是兩位值班醫生。兩人都感到不適，所以他們都決定請假。不幸的是，他們恰好在同一時間點選按鈕下班。圖 7-8 說明了接下來的事情。

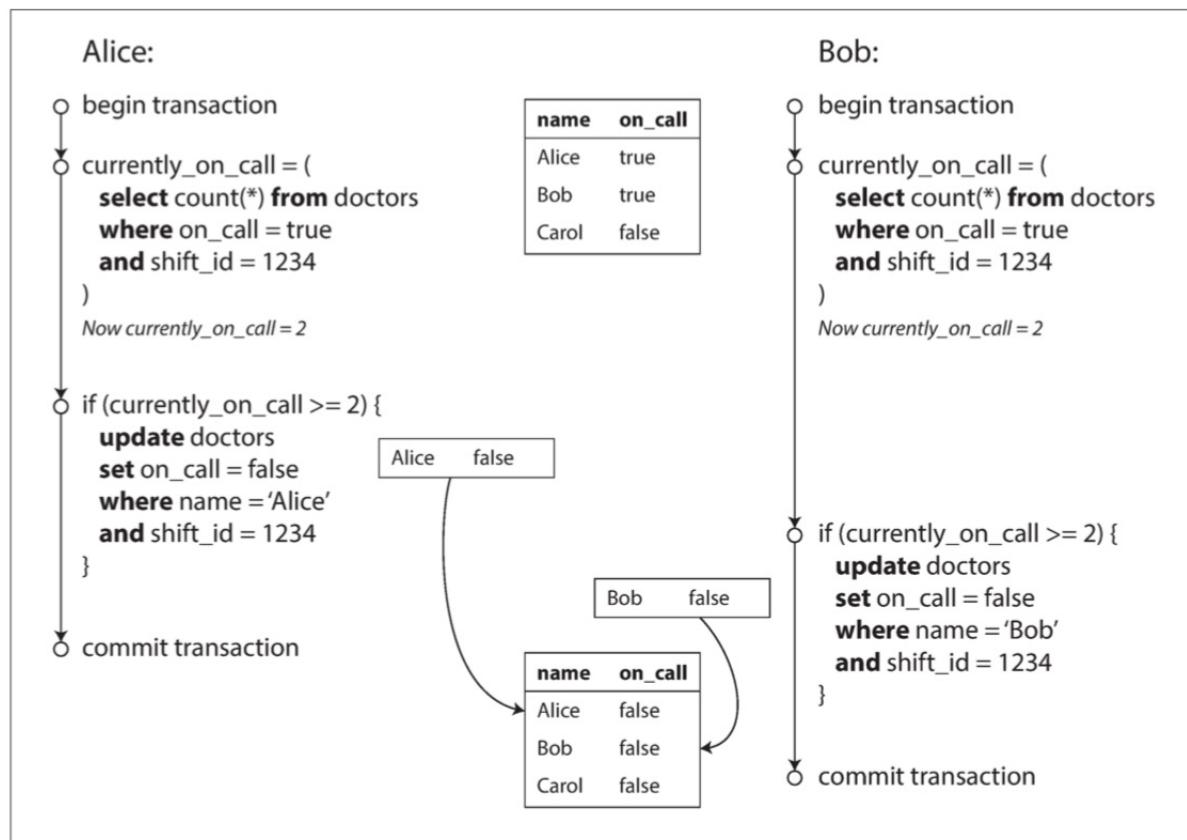


圖 7-8 寫入偏差導致應用程式錯誤的示例

在兩個事務中，應用首先檢查是否有兩個或以上的醫生正在值班；如果是的話，它就假定一名醫生可以安全地休班。由於資料庫使用快照隔離，兩次檢查都返回 2，所以兩個事務都進入下一個階段。Alice 更新自己的記錄休班了，而 Bob 也做了一樣的事情。兩個事務都成功提交了，現在沒有醫生值班了。違反了至少有一名醫生在值班的要求。

寫入偏差的特徵

這種異常稱為 **寫入偏差**【28】。它既不是 **髒寫**，也不是 **丟失更新**，因為這兩個事務正在更新兩個不同的物件（Alice 和 Bob 各自的待命記錄）。在這裡發生的衝突並不是那麼明顯，但是這顯然是一個競爭條件：如果兩個事務一個接一個地執行，那麼第二個醫生就不能歇班了。異常行為只有在事務併發進行時才有可能發生。

可以將寫入偏差視為丟失更新問題的一般化。如果兩個事務讀取相同的物件，然後更新其中一些物件（不同的事務可能更新不同的物件），則可能發生寫入偏差。在多個事務更新同一個物件的特殊情況下，就會發生髒寫或丟失更新（取決於時序）。

我們已經看到，有各種不同的方法來防止丟失的更新。但對於寫入偏差，我們的選擇更受限制：

- 由於涉及多個物件，單物件的原子操作不起作用。
- 不幸的是，在一些快照隔離的實現中，自動檢測丟失更新對此並沒有幫助。在 PostgreSQL 的可重複讀，MySQL/InnoDB 的可重複讀，Oracle 可序列化或 SQL Server 的快照隔離級別中，都不會自動檢測寫入偏差【23】。自動防止寫入偏差需要真正的可序列化隔離（請參閱“[可序列化](#)”）。
- 某些資料庫允許配置約束，然後由資料庫強制執行（例如，唯一性，外來鍵約束或特定值限制）。但是為了指定至少有一名醫生必須線上，需要一個涉及多個物件的約束。大多數資料庫沒有內建對這種約束的支援，但是你可以使用觸發器，或者物化檢視來實現它們，這取決於不同的資料庫【42】。
- 如果無法使用可序列化的隔離級別，則此情況下的次優選項可能是顯式鎖定事務所依賴的行。在例子中，你可以寫下如下的程式碼：

```
BEGIN TRANSACTION;
SELECT * FROM doctors
WHERE on_call = TRUE
AND shift_id = 1234 FOR UPDATE;

UPDATE doctors
SET on_call = FALSE
WHERE name = 'Alice'
AND shift_id = 1234;

COMMIT;
```

- 和以前一樣，`FOR UPDATE` 告訴資料庫鎖定返回的所有行以用於更新。

寫入偏差的更多例子

寫入偏差乍看像是一個深奧的問題，但一旦意識到這一點，很容易會注意到它可能發生在更多場景下。以下是一些例子：

- 會議室預訂系統

比如你想要規定不能在同一時間對同一個會議室進行多次的預訂【43】。當有人想要預訂時，首先檢查是否存在相互衝突的預訂（即預訂時間範圍重疊的同一房間），如果沒有找到，則建立會議（請參閱示例 7-2）[ix](#)。

[ix](#). 在 PostgreSQL 中，你可以使用範圍型別優雅地執行此操作，但在其他資料庫中並未得到廣泛支援。 ↩

例 7-2 會議室預訂系統試圖避免重複預訂（在快照隔離下不安全）

```
BEGIN TRANSACTION;

-- 檢查所有現存的與 12:00~13:00 重疊的預定
SELECT COUNT(*) FROM bookings
WHERE room_id = 123 AND
    end_time > '2015-01-01 12:00' AND start_time < '2015-01-01 13:00';

-- 如果之前的查詢返回 0
INSERT INTO bookings(room_id, start_time, end_time, user_id)
VALUES (123, '2015-01-01 12:00', '2015-01-01 13:00', 666);

COMMIT;
```

不幸的是，快照隔離並不能防止另一個使用者同時插入衝突的會議。為了確保不會遇到排程衝突，你又需要可序列化的隔離級別了。

- 多人遊戲

在 [例 7-1](#) 中，我們使用一個鎖來防止丟失更新（也就是確保兩個玩家不能同時移動同一個棋子）。但是鎖定並不妨礙玩家將兩個不同的棋子移動到棋盤上的相同位置，或者採取其他違反遊戲規則的行為。取決於你正在執行的規則型別，也許可以使用唯一約束（unique constraint），否則你很容易發生寫入偏差。

- 搶注使用者名稱

在每個使用者擁有唯一使用者名稱的網站上，兩個使用者可能會嘗試同時建立具有相同使用者名稱的帳戶。可以在事務檢查名稱是否被搶佔，如果沒有則使用該名稱建立帳戶。但是像在前面的例子中那樣，在快照隔離下這是不安全的。幸運的是，唯一約束是一個簡單的解決辦法（第二個事務在提交時會因為違反使用者名稱唯一約束而被中止）。

- 防止雙重開支

允許使用者花錢或使用積分的服務，需要檢查使用者的支付數額不超過其餘額。可以透過在使用者的帳戶中插入一個試探性的消費專案來實現這一點，列出帳戶中的所有專案，並檢查總和是否為正值【44】。在寫入偏差場景下，可能會發生兩個支出專案同時插入，一起導致餘額變為負值，但這兩個事務都不會注意到另一個。

導致寫入偏差的幻讀

所有這些例子都遵循類似的模式：

1. 一個 `SELECT` 查詢找出符合條件的行，並檢查是否符合一些要求。（例如：至少有兩名醫生在值班；不存在對該會議室同一時段的預定；棋盤上的位置沒有被其他棋子佔據；使用者名稱還沒有被搶注；帳戶裡還有足夠餘額）
2. 按照第一個查詢的結果，應用程式碼決定是否繼續。（可能會繼續操作，也可能中止並報錯）
3. 如果應用決定繼續操作，就執行寫入（插入、更新或刪除），並提交事務。

這個寫入的效果改變了步驟 2 中的先決條件。換句話說，如果在提交寫入後，重複執行一次步驟 1 的 `SELECT` 查詢，將會得到不同的結果。因為寫入改變了符合搜尋條件的行集（現在少了一個醫生值班，那時候的會議室現在已經被預訂了，棋盤上的這個位置已經被佔據了，使用者名稱已經被搶注，帳戶餘額不夠了）。

這些步驟可能以不同的順序發生。例如可以首先進行寫入，然後進行 `SELECT` 查詢，最後根據查詢結果決定是放棄還是提交。

在醫生值班的例子中，在步驟 3 中修改的行，是步驟 1 中返回的行之一，所以我們可以透過鎖定步驟 1 中的行（`SELECT FOR UPDATE`）來使事務安全並避免寫入偏差。但是其他四個例子是不同的：它們檢查是否 **不存在** 某些滿足條件的行，寫入會 **新增** 一個匹配相同條件的行。如果步驟 1 中的查詢沒有返回任何行，則 `SELECT FOR UPDATE` 鎖不了任何東西。

這種效應：一個事務中的寫入改變另一個事務的搜尋查詢的結果，被稱為 **幻讀**【3】。快照隔離避免了只讀查詢中幻讀，但是在像我們討論的例子那樣的讀寫事務中，幻讀會導致特別棘手的寫入偏差情況。

物化衝突

如果幻讀的問題是沒有物件可以加鎖，也許可以人為地在資料庫中引入一個鎖物件？

例如，在會議室預訂的場景中，可以想象建立一個關於時間槽和房間的表。此表中的每一行對應於特定時間段（例如 15 分鐘）的特定房間。可以提前插入房間和時間的所有可能組合行（例如接下來的六個月）。

現在，要建立預訂的事務可以鎖定（`SELECT FOR UPDATE`）表中與所需房間和時間段對應的行。在獲得鎖定之後，它可以檢查重疊的預訂並像以前一樣插入新的預訂。請注意，這個表並不是用來儲存預訂相關的資訊——它完全就是一組鎖，用於防止同時修改同一房間和時間範圍內的預訂。

這種方法被稱為 **物化衝突** (**materializing conflicts**)，因為它將幻讀變為資料庫中一組具體行上的鎖衝突【11】。不幸的是，弄清楚如何物化衝突可能很難，也很容易出錯，並且讓併發控制機制洩漏到應用資料模型是很醜陋的做法。出於這些原因，如果沒有其他辦法可以實現，物化衝突應被視為最後的手段。在大多數情況下。**可序列化** (**Serializable**) 的隔離級別是更可取的。

可序列化

在本章中，已經看到了幾個易於出現競爭條件的事務例子。讀已提交 和 快照隔離 級別會阻止某些競爭條件，但不會阻止另一些。我們遇到了一些特別棘手的例子，寫入偏差 和 幻讀。這是一個可悲的情況：

- 隔離級別難以理解，並且在不同的資料庫中實現的不一致（例如，“可重複讀”的含義天差地別）。
- 光檢查應用程式碼很難判斷在特定的隔離級別執行是否安全。特別是在大型應用程式中，你可能並不知道併發發生的所有事情。
- 沒有檢測競爭條件的好工具。原則上來說，靜態分析可能會有幫助【26】，但研究中的技術還沒法實際應用。併發問題的測試是很難的，因為它們通常是非確定性的——只有在倒楣的時序下才會出現問題。

這不是一個新問題，從 20 世紀 70 年代以來就一直是這樣了，當時首先引入了較弱的隔離級別【2】。一直以來，研究人員的答案都很簡單：使用 **可序列化** (**serializable**) 的隔離級別！

可序列化 (**Serializability**) 隔離通常被認為是最強的隔離級別。它保證即使事務可以並行執行，最終的結果也是一樣的，就好像它們沒有任何併發性，連續挨個執行一樣。因此資料庫保證，如果事務在單獨執行時正常執行，則它們在併發執行時繼續保持正確——換句話說，資料庫可以防止 **所有** 可能的競爭條件。

但如果可序列化隔離級別比弱隔離級別的爛攤子要好得多，那為什麼沒有人見人愛？為了回答這個問題，我們需要看看實現可序列化的選項，以及它們如何執行。目前大多數提供可序列化的資料庫都使用了三種技術之一，本章的剩餘部分將會介紹這些技術：

- 字面意義上地序列順序執行事務（請參閱“[真的序列執行](#)”）
- **兩階段鎖定** (**2PL, two-phase locking**)，幾十年來唯一可行的選擇（請參閱“[兩階段鎖定](#)”）
- 樂觀併發控制技術，例如 **可序列化快照隔離** (**serializable snapshot isolation**，請參閱“[可序列化快照隔離](#)”)

現在將主要在單節點資料庫的背景下討論這些技術；在 [第九章](#) 中，我們將研究如何將它們推廣到涉及分散式系統中多個節點的事務。

真的序列執行

避免併發問題的最簡單方法就是完全不要併發：在單個執行緒上按順序一次只執行一個事務。這樣做就完全繞開了檢測 / 防止事務間衝突的問題，由此產生的隔離，正是可序列化的定義。

儘管這似乎是一個明顯的主意，但資料庫設計人員只是在 2007 年左右才決定，單執行緒迴圈執行事務是可行的【45】。如果多執行緒併發在過去的 30 年中被認為是獲得良好效能的關鍵所在，那麼究竟是什麼改變致使單執行緒執行變為可能呢？

兩個進展引發了這個反思：

- RAM 足夠便宜了，許多場景現在都可以將完整的活躍資料集儲存在記憶體中（請參閱“[在記憶體中儲存一切](#)”）。當事務需要訪問的所有資料都在記憶體中時，事務處理的執行速度要比等待資料從磁碟載入時快得多。
- 資料庫設計人員意識到 OLTP 事務通常很短，而且只進行少量的讀寫操作（請參閱“[事務處理還是分析？](#)”）。相比之下，長時間執行的分析查詢通常是隻讀的，因此它們可以在序列執行迴圈之外的一致快照（使用快照隔離）上執行。

序列執行事務的方法在 VoltDB/H-Store、Redis 和 Datomic 中實現【46,47,48】。設計用於單執行緒執行的系統有時可以比支援併發的系統性能更好，因為它可以避免鎖的協調開銷。但是其吞吐量僅限於單個 CPU 核的吞吐量。為了充分利用單一執行緒，需要有與傳統形式的事務不同的結構。

在儲存過程中封裝事務

在資料庫的早期階段，意圖是資料庫事務可以包含整個使用者活動流程。例如，預訂機票是一個多階段的過程（搜尋路線，票價和可用座位，決定行程，在每段行程的航班上訂座，輸入乘客資訊，付款）。資料庫設計者認為，如果整個過程是一個事務，那麼它就可以被原子化地執行。

不幸的是，人類做出決定和回應的速度非常緩慢。如果資料庫事務需要等待來自使用者的輸入，則資料庫需要支援潛在的大量併發事務，其中大部分是空閒的。大多數資料庫不能高效完成這項工作，因此幾乎所有的 OLTP 應用程式都避免在事務中等待互動式的使用者輸入，以此來保持事務的簡短。在 Web 上，這意味著事務在同一個 HTTP 請求中被提交——一個事務不會跨越多個請求。一個新的 HTTP 請求開始一個新的事務。

即使已經將人類從關鍵路徑中排除，事務仍然以互動式的客戶端 / 伺服器風格執行，一次一個語句。應用程式進行查詢，讀取結果，可能根據第一個查詢的結果進行另一個查詢，依此類推。查詢和結果在應用程式程式碼（在一臺機器上執行）和資料庫伺服器（在另一臺機器上）之間來回傳送。

在這種互動式的事務方式中，應用程式和資料庫之間的網路通訊耗費了大量的時間。如果不允許在資料庫中進行併發處理，且一次只處理一個事務，則吞吐量將會非常糟糕，因為資料庫大部分的時間都花費在等待應用程式發出當前事務的下一個查詢。在這種資料庫中，為了獲得合理的效能，需要同時處理多個事務。

出於這個原因，具有單執行緒序列事務處理的系統不允許互動式的多語句事務。取而代之，應用程式必須提前將整個事務程式碼作為儲存過程提交給資料庫。這些方法之間的差異如 圖 7-9 所示。如果事務所需的所有資料都在記憶體中，則儲存過程可以非常快地執行，而不用等待任何網路或磁碟 I/O。

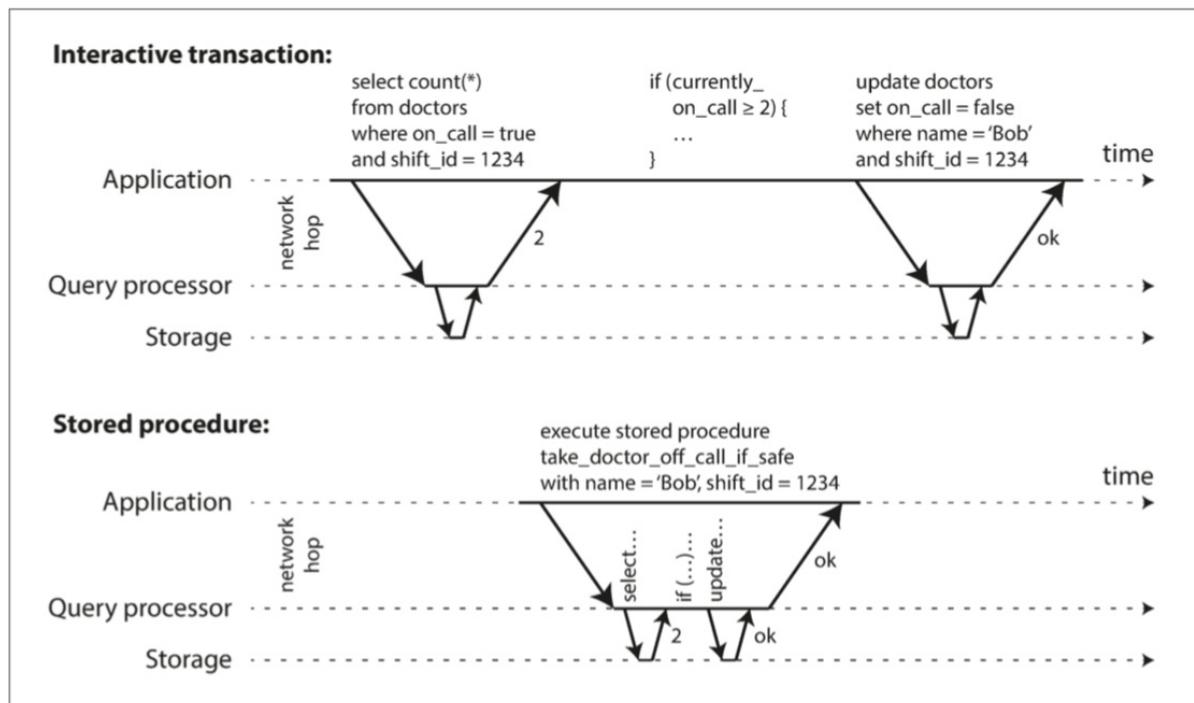


圖 7-9 互動式事務和儲存過程之間的區別（使用圖 7-8 的示意事務）

儲存過程的優點和缺點

儲存過程在關係型資料庫中已經存在了一段時間了，自 1999 年以來它們一直是 SQL 標準（SQL/PSM）的一部分。出於各種原因，它們的名聲有點不太好：

- 每個資料庫廠商都有自己的儲存過程語言（Oracle 有 PL/SQL，SQL Server 有 T-SQL，PostgreSQL 有 PL/pgSQL，等等）。這些語言並沒有跟上通用程式語言的發展，所以從今天的角度來看，它們看起來相當醜陋和陳舊，而且缺乏大多數程式語言中能找到的庫的生態系統。
- 在資料庫中執行的程式碼難以管理：與應用伺服器相比，它更難除錯，更難以保持版本控制和部署，更難測試，並

且難以整合到指標收集系統來進行監控。

- 資料庫通常比應用伺服器對效能敏感的多，因為單個數據庫例項通常由許多應用伺服器共享。資料庫中一個寫得不好的儲存過程（例如，佔用大量記憶體或 CPU 時間）會比在應用伺服器中相同的程式碼造成更多的麻煩。

但是這些問題都是可以克服的。現代的儲存過程實現放棄了 PL/SQL，而是使用現有的通用程式語言：VoltDB 使用 Java 或 Groovy，Datomic 使用 Java 或 Clojure，而 Redis 使用 Lua。

儲存過程與記憶體儲存，使得在單個執行緒上執行所有事務變得可行。由於不需要等待 I/O，且避免了併發控制機制的開銷，它們可以在單個執行緒上實現相當好的吞吐量。

VoltDB 還使用儲存過程進行複制：但不是將事務的寫入結果從一個節點複製到另一個節點，而是在每個節點上執行相同的儲存過程。因此 VoltDB 要求儲存過程是 **確定性的**（在不同的節點上執行時，它們必須產生相同的結果）。舉個例子，如果事務需要使用當前的日期和時間，則必須透過特殊的確定性 API 來實現。

分割槽

順序執行所有事務使併發控制簡單多了，但資料庫的事務吞吐量被限制為單機單核的速度。只讀事務可以使用快照隔離在其它地方執行，但對於寫入吞吐量較高的應用，單執行緒事務處理器可能成為一個嚴重的瓶頸。

為了伸縮至多個 CPU 核心和多個節點，可以對資料進行分割槽（請參閱 [第六章](#)），在 VoltDB 中支援這樣做。如果你可以找到一種對資料集進行分割槽的方法，以便每個事務只需要在單個分割槽中讀寫資料，那麼每個分割槽就可以擁有自己獨立執行的事務處理執行緒。在這種情況下可以為每個分割槽指派一個獨立的 CPU 核，事務吞吐量就可以與 CPU 核數保持線性伸縮 [\[47\]](#)。

但是，對於需要訪問多個分割槽的任何事務，資料庫必須在觸及的所有分割槽之間協調事務。儲存過程需要跨越所有分割槽鎖定執行，以確保整個系統的可序列性。

由於跨分割槽事務具有額外的協調開銷，所以它們比單分割槽事務慢得多。VoltDB 報告的吞吐量大約是每秒 1000 個跨分割槽寫入，比單分割槽吞吐量低幾個數量級，並且不能透過增加更多的機器來增加吞吐量 [\[49\]](#)。

事務是否可以是劃分至單個分割槽很大程度上取決於應用資料的結構。簡單的鍵值資料通常可以非常容易地進行分割槽，但是具有多個次級索引的資料可能需要大量的跨分割槽協調（請參閱 “[分割槽與次級索引](#)”）。

序列執行小結

在特定約束條件下，真的序列執行事務，已經成為一種實現可序列化隔離等級的可行辦法。

- 每個事務都必須小而快，只要有一個緩慢的事務，就會拖慢所有事務處理。
- 僅限於活躍資料集可以放入記憶體的情況。很少訪問的資料可能會被移動到磁碟，但如果需要在單執行緒執行的事務中訪問這些磁碟中的資料，系統就會變得非常慢 [x](#)。
- 寫入吞吐量必須低到能在單個 CPU 核上處理，如若不然，事務需要能劃分至單個分割槽，且不需要跨分割槽協調。
- 跨分割槽事務是可能的，但是它們能被使用的程度有很大的限制。

[x](#). 如果事務需要訪問不在記憶體中的資料，最好的解決方案可能是中止事務，非同步地將資料提取到記憶體中，同時繼續處理其他事務，然後在資料載入完畢時重新啟動事務。這種方法被稱為 **反快取 (anti-caching)**，正如前面在 “[在記憶體中儲存一切](#)” 中所述。 ↪

兩階段鎖定

大約 30 年來，在資料庫中只有一種廣泛使用的序列化演算法：**兩階段鎖定 (2PL, two-phase locking)** [xi](#)

[xi](#). 有時也稱為 **嚴格兩階段鎖定 (SS2PL, strong strict two-phase locking)**，以便和其他 2PL 變體區分。 ↪

2PL不是2PC

請注意，雖然兩階段鎖定（2PL）聽起來非常類似於兩階段提交（2PC），但它們是完全不同的東西。我們將在 [第九章](#) 討論 2PC。

之前我們看到鎖通常用於防止髒寫（請參閱 “[沒有髒寫](#)” 一節）：如果兩個事務同時嘗試寫入同一個物件，則鎖可確保第二個寫入必須等到第一個寫入完成事務（中止或提交），然後才能繼續。

兩階段鎖定類似，但是鎖的要求更強得多。只要沒有寫入，就允許多個事務同時讀取同一個物件。但物件只要有寫入（修改或刪除），就需要 **獨佔訪問（exclusive access）** 許可權：

- 如果事務 A 讀取了一個物件，並且事務 B 想要寫入該物件，那麼 B 必須等到 A 提交或中止才能繼續（這確保 B 不能在 A 底下意外地改變物件）。
- 如果事務 A 寫入了一個物件，並且事務 B 想要讀取該物件，則 B 必須等到 A 提交或中止才能繼續（像 [圖 7-1](#) 那樣讀取舊版本的物件在 2PL 下是不可接受的）。

在 2PL 中，寫入不僅會阻塞其他寫入，也會阻塞讀，反之亦然。快照隔離使得 讀不阻塞寫，寫也不阻塞讀（請參閱 “[實現快照隔離](#)”），這是 2PL 和快照隔離之間的關鍵區別。另一方面，因為 2PL 提供了可序列化的性質，它可以防止早先討論的所有競爭條件，包括丟失更新和寫入偏差。

實現兩階段鎖

2PL 用於 MySQL（InnoDB）和 SQL Server 中的可序列化隔離級別，以及 DB2 中的可重複讀隔離級別 [【23,36】](#)。

讀與寫的阻塞是透過為資料庫中每個物件新增鎖來實現的。鎖可以處於 **共享模式（shared mode）** 或 **獨佔模式（exclusive mode）**。鎖使用如下：

- 若事務要讀取物件，則須先以共享模式獲取鎖。允許多個事務同時持有共享鎖。但如果另一個事務已經在物件上持有排它鎖，則這些事務必須等待。
- 若事務要寫入一個物件，它必須首先以獨佔模式獲取該鎖。沒有其他事務可以同時持有鎖（無論是共享模式還是獨佔模式），所以如果物件上存在任何鎖，該事務必須等待。
- 如果事務先讀取再寫入物件，則它可能會將其共享鎖升級為獨佔鎖。升級鎖的工作與直接獲得獨佔鎖相同。
- 事務獲得鎖之後，必須繼續持有鎖直到事務結束（提交或中止）。這就是“兩階段”這個名字的來源：第一階段（當事務正在執行時）獲取鎖，第二階段（在事務結束時）釋放所有的鎖。

由於使用了這麼多的鎖，因此很可能會發生：事務 A 等待事務 B 釋放它的鎖，反之亦然。這種情況叫做 **死鎖（Deadlock）**。資料庫會自動檢測事務之間的死鎖，並中止其中一個，以便另一個繼續執行。被中止的事務需要由應用程式重試。

兩階段鎖定的效能

兩階段鎖定的巨大缺點，以及 70 年代以來沒有被所有人使用的原因，是其效能問題。兩階段鎖定下的事務吞吐量與查詢響應時間要比弱隔離級別下要差得多。

這一部分是由於獲取和釋放所有這些鎖的開銷，但更重要的是由於併發性的降低。按照設計，如果兩個併發事務試圖做任何可能導致競爭條件的事情，那麼必須等待另一個完成。

傳統的關係資料庫不限制事務的持續時間，因為它們是為等待人類輸入的互動式應用而設計的。因此，當一個事務需要等待另一個事務時，等待的時長並沒有限制。即使你保證所有的事務都很短，如果有多個事務想要訪問同一個物件，那麼可能會形成一個佇列，所以事務可能需要等待幾個其他事務才能完成。

因此，執行 2PL 的資料庫可能具有相當不穩定的延遲，如果在工作負載中存在爭用，那麼可能高百分位點處的響應會非常的慢（請參閱 “[描述效能](#)”）。可能只需要一個緩慢的事務，或者一個訪問大量資料並獲取許多鎖的事務，就能把系統的其他部分拖慢，甚至迫使系統停機。當需要穩健的操作時，這種不穩定性是有問題的。

基於鎖實現的讀已提交隔離級別可能發生死鎖，但在基於 2PL 實現的可序列化隔離級別中，它們會出現的頻繁的多（取決於事務的訪問模式）。這可能是一個額外的效能問題：當事務由於死鎖而被中止並被重試時，它需要從頭重做它的工作。如果死鎖很頻繁，這可能意味著巨大的浪費。

謂詞鎖

在前面關於鎖的描述中，我們掩蓋了一個微妙而重要的細節。在“導致寫入偏差的幻讀”中，我們討論了 幻讀 (phantoms) 的問題。即一個事務改變另一個事務的搜尋查詢的結果。具有可序列化隔離級別的資料庫必須防止 幻讀。

在會議室預訂的例子中，這意味著如果一個事務在某個時間視窗內搜尋了一個房間的現有預訂（見 例 7-2），則另一個事務不能同時插入或更新同一時間視窗與同一房間的另一個預訂（可以同時插入其他房間的預訂，或在不影響另一個預定的條件下預定同一房間的其他時間段）。

如何實現這一點？從概念上講，我們需要一個 謂詞鎖 (predicate lock) 【3】。它類似於前面描述的共享 / 排它鎖，但不屬於特定的物件（例如，表中的一行），它屬於所有符合某些搜尋條件的物件，如：

```
SELECT * FROM bookings
WHERE room_id = 123 AND
    end_time > '2018-01-01 12:00' AND
    start_time < '2018-01-01 13:00';
```

謂詞鎖限制訪問，如下所示：

- 如果事務 A 想要讀取匹配某些條件的物件，就像在這個 SELECT 查詢中那樣，它必須獲取查詢條件上的 共享謂詞鎖 (shared-mode predicate lock)。如果另一個事務 B 持有任何滿足這一查詢條件物件的排它鎖，那麼 A 必須等到 B 釋放它的鎖之後才允許進行查詢。
- 如果事務 A 想要插入，更新或刪除任何物件，則必須首先檢查舊值或新值是否與任何現有的謂詞鎖匹配。如果事務 B 持有匹配的謂詞鎖，那麼 A 必須等到 B 已經提交或中止後才能繼續。

這裡的關鍵思想是，謂詞鎖甚至適用於資料庫中尚不存在，但將來可能會新增的物件（幻象）。如果兩階段鎖定包含謂詞鎖，則資料庫將阻止所有形式的寫入偏差和其他競爭條件，因此其隔離實現了可序列化。

索引範圍鎖

不幸的是謂詞鎖效能不佳：如果活躍事務持有很多鎖，檢查匹配的鎖會非常耗時。因此，大多數使用 2PL 的資料庫實際上實現了索引範圍鎖 (index-range locking，也稱為 next-key locking)，這是一個簡化的近似版謂詞鎖【41,50】。

透過使謂詞匹配到一個更大的集合來簡化謂詞鎖是安全的。例如，如果你有在中午和下午 1 點之間預訂 123 號房間的謂詞鎖，則鎖定 123 號房間的所有時間段，或者鎖定 12:00~13:00 時間段的所有房間（不只是 123 號房間）是一個安全的近似，因為任何滿足原始謂詞的寫入也一定會滿足這種更鬆散的近似。

在房間預訂資料庫中，你可能會在 room_id 列上有一個索引，並且 / 或者在 start_time 和 end_time 上有索引（否則前面的查詢在大型資料庫上的速度會非常慢）：

- 假設你的索引位於 room_id 上，並且資料庫使用此索引查詢 123 號房間的現有預訂。現在資料庫可以簡單地將共享鎖附加到這個索引項上，指示事務已搜尋 123 號房間用於預訂。
- 或者，如果資料庫使用基於時間的索引來查詢現有預訂，那麼它可以將共享鎖附加到該索引中的一系列值，指示事務已經將 12:00~13:00 時間段標記為用於預定。

無論哪種方式，搜尋條件的近似值都附加到其中一個索引上。現在，如果另一個事務想要插入、更新或刪除同一個房間和 / 或重疊時間段的預訂，則它將不得不更新索引的相同部分。在這樣做的過程中，它會遇到共享鎖，它將被迫等到鎖被釋放。

這種方法能夠有效防止幻讀和寫入偏差。索引範圍鎖並不像謂詞鎖那樣精確（它們可能會鎖定更大範圍的物件，而不是維持可序列化所必需的範圍），但是由於它們的開銷較低，所以是一個很好的折衷。

如果沒有可以掛載範圍鎖的索引，資料庫可以退化到使用整個表上的共享鎖。這對效能不利，因為它會阻止所有其他事務寫入表格，但這是一個安全的回退位置。

可序列化快照隔離

本章描繪了資料庫中併發控制的黯淡畫面。一方面，我們實現了效能不好（2PL）或者伸縮性不好（序列執行）的可序列化隔離級別。另一方面，我們有效能良好的弱隔離級別，但容易出現各種競爭條件（丟失更新、寫入偏差、幻讀等）。序列化的隔離級別和高效能是從根本上相互矛盾的嗎？

也許不是：一個稱為 **可序列化快照隔離（SSI, serializable snapshot isolation）** 的演算法是非常有前途的。它提供了完整的可序列化隔離級別，但與快照隔離相比只有很小的效能損失。SSI 是相當新的：它在 2008 年首次被描述【40】，並且是 Michael Cahill 的博士論文【51】的主題。

今天，SSI 既用於單節點資料庫（PostgreSQL9.1 以後的可序列化隔離級別），也用於分散式資料庫（FoundationDB 使用類似的演算法）。由於 SSI 與其他併發控制機制相比還很年輕，還處於在實踐中證明自己表現的階段。但它有可能因為足夠快而在未來成為新的預設選項。

悲觀與樂觀的併發控制

兩階段鎖是一種所謂的 **悲觀併發控制機制（pessimistic）**：它是基於這樣的原則：如果有事情可能出錯（如另一個事務所持有的鎖所表示的），最好等到情況安全後再做任何事情。這就像互斥，用於保護多執行緒程式設計中的資料結構。

從某種意義上說，序列執行可以稱為悲觀到了極致：在事務持續期間，每個事務對整個資料庫（或資料庫的一個分割槽）具有排它鎖，作為對悲觀的補償，我們讓每筆事務執行得非常快，所以只需要短時間持有“鎖”。

相比之下，**序列化快照隔離** 是一種 **樂觀（optimistic）** 的併發控制技術。在這種情況下，樂觀意味著，如果存在潛在的危險也不阻止事務，而是繼續執行事務，希望一切都會好起來。當一個事務想要提交時，資料庫檢查是否有什麼不好的事情發生（即隔離是否被違反）；如果是的話，事務將被中止，並且必須重試。只有可序列化的事務才被允許提交。

樂觀併發控制是一個古老的想法【52】，其優點和缺點已經爭論了很長時間【53】。如果存在很多 **爭用**（contention，即很多事務試圖訪問相同的物件），則表現不佳，因為這會導致很大一部分事務需要中止。如果系統已經接近最大吞吐量，來自重試事務的額外負載可能會使效能變差。

但是，如果有足夠的空間容量，並且事務之間的爭用不是太高，樂觀的併發控制技術往往比悲觀的效能要好。可交換的原子操作可以減少爭用：例如，如果多個事務同時要增加一個計數器，那麼應用增量的順序（只要計數器不在同一個事務中讀取）就無關緊要了，所以併發增量可以全部應用且不會有衝突。

顧名思義，SSI 基於快照隔離——也就是說，事務中的所有讀取都是來自資料庫的一致性快照（請參閱“[快照隔離和可重複讀取](#)”）。與早期的樂觀併發控制技術相比這是主要的區別。在快照隔離的基礎上，SSI 添加了一種演算法來檢測寫入之間的序列化衝突，並確定要中止哪些事務。

基於過時前提的決策

先前討論了快照隔離中的寫入偏差（請參閱“[寫入偏差與幻讀](#)”）時，我們觀察到一個迴圈模式：事務從資料庫讀取一些資料，檢查查詢的結果，並根據它看到的結果決定採取一些操作（寫入資料庫）。但是，在快照隔離的情況下，原始查詢的結果在事務提交時可能不再是最新的，因為資料可能在同一時間被修改。

換句話說，事務基於一個 **前提（premise）** 採取行動（事務開始時候的事實，例如：“目前有兩名醫生正在值班”）。之後當事務要提交時，原始資料可能已經改變——前提可能不再成立。

當應用程式進行查詢時（例如，“當前有多少醫生正在值班？”），資料庫不知道應用邏輯如何使用該查詢結果。在這種情況下為了安全，資料庫需要假設任何對該結果集的變更都可能會使該事務中的寫入變得無效。換而言之，事務中的查詢與寫入可能存在因果依賴。為了提供可序列化的隔離級別，如果事務在過時的前提下執行操作，資料庫必須能檢測到這種情況，並中止事務。

資料庫如何知道查詢結果是否可能已經改變？有兩種情況需要考慮：

- 檢測對舊 MVCC 物件版本的讀取（讀之前存在未提交的寫入）
- 檢測影響先前讀取的寫入（讀之後發生寫入）

檢測舊MVCC讀取

回想一下，快照隔離通常是透過多版本併發控制（MVCC；見 [圖 7-10](#)）來實現的。當一個事務從 MVCC 資料庫中的一致快照讀時，它將忽略取快照時尚未提交的任何其他事務所做的寫入。在 [圖 7-10](#) 中，事務 43 認為 Alice 的 `on_call = true`，因為事務 42（修改 Alice 的待命狀態）未被提交。然而，在事務 43 想要提交時，事務 42 已經提交。這意味著在讀一致性快照時被忽略的寫入已經生效，事務 43 的前提不再為真。

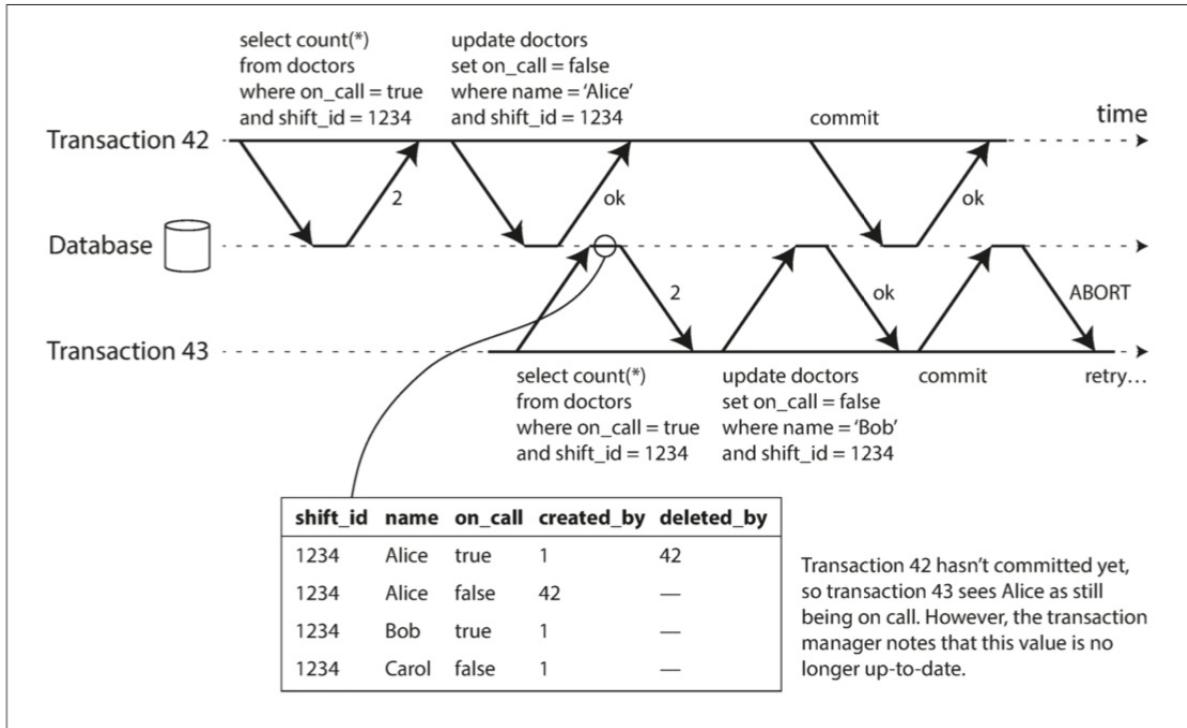


圖 7-10 檢測事務何時從 MVCC 快照讀取過時的值

為了防止這種異常，資料庫需要跟蹤一個事務由於 MVCC 可見性規則而忽略另一個事務的寫入。當事務想要提交時，資料庫檢查是否有任何被忽略的寫入現在已經被提交。如果是這樣，事務必須中止。

為什麼要等到提交？當檢測到陳舊的讀取時，為什麼不立即中止事務 43？因為如果事務 43 是隻讀事務，則不需要中止，因為沒有寫入偏差的風險。當事務 43 進行讀取時，資料庫還不知道事務是否要稍後執行寫操作。此外，事務 42 可能在事務 43 被提交的時候中止或者可能仍然未被提交，因此讀取可能終究不是陳舊的。透過避免不必要的中止，SSI 保留了快照隔離從一致快照中長時間讀取的能力。

檢測影響之前讀取的寫入

第二種情況要考慮的是另一個事務在讀取資料之後修改資料。這種情況如 [圖 7-11](#) 所示。

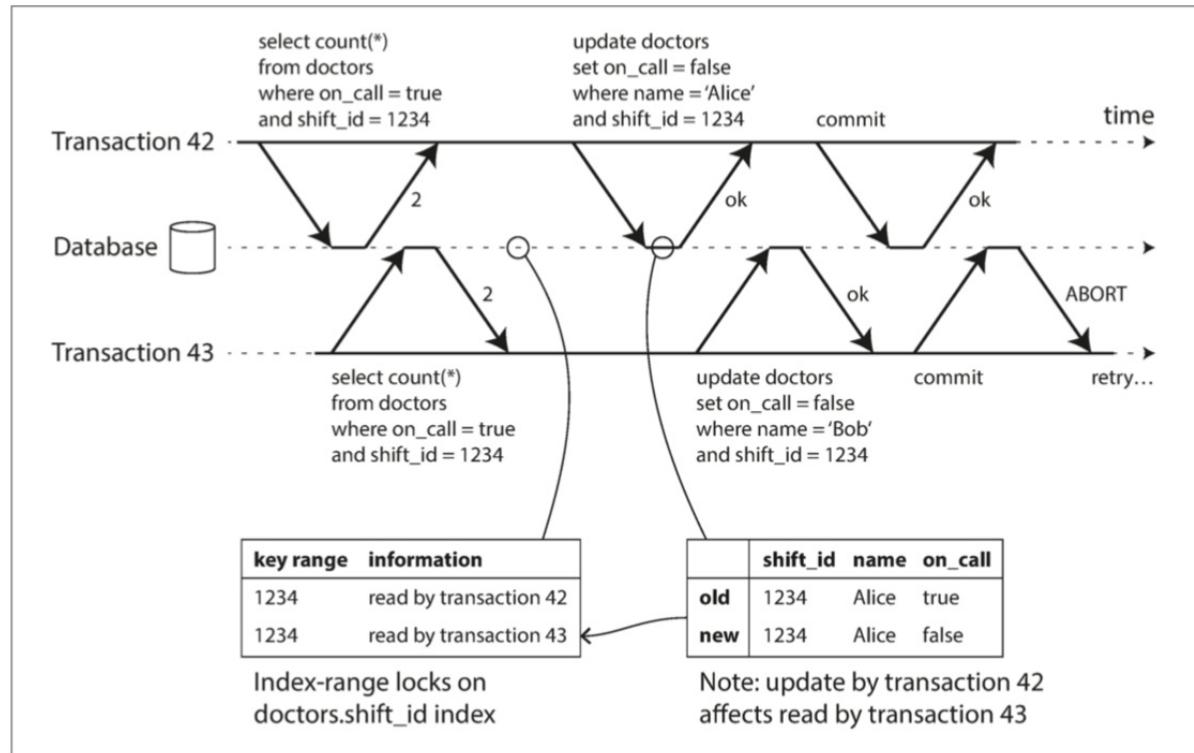


圖 7-11 在可序列化快照隔離中，檢測一個事務何時修改另一個事務的讀取。

在兩階段鎖定的上下文中，我們討論了索引範圍鎖（請參閱“[索引範圍鎖](#)”），它允許資料庫鎖定與某個搜尋查詢匹配的所有行的訪問權，例如 `WHERE shift_id = 1234`。可以在這裡使用類似的技術，除了 SSI 鎖不會阻塞其他事務。

在 圖 7-11 中，事務 42 和 43 都在班次 1234 查詢值班醫生。如果在 `shift_id` 上有索引，則資料庫可以使用索引項 1234 來記錄事務 42 和 43 讀取這個資料的事實。（如果沒有索引，這個資訊可以在表級別進行跟蹤）。這個資訊只需要保留一段時間：在一個事務完成（提交或中止），並且所有的併發事務完成之後，資料庫就可以忘記它讀取的資料了。

當事務寫入資料庫時，它必須在索引中查詢最近曾讀取受影響資料的其他事務。這個過程類似於在受影響的鍵範圍上獲取寫鎖，但鎖並不會阻塞事務直到其他讀事務完成，而是像警戒線一樣只是簡單通知其他事務：你們讀過的資料可能不是最新的啦。

在 圖 7-11 中，事務 43 通知事務 42 其先前讀已過時，反之亦然。事務 42 首先提交併成功，儘管事務 43 的寫影響了 42，但因為事務 43 尚未提交，所以寫入尚未生效。然而當事務 43 想要提交時，來自事務 42 的衝突寫入已經被提交，所以事務 43 必須中止。

可序列化快照隔離的效能

與往常一樣，許多工程細節會影響演算法的實際表現。例如一個權衡是跟蹤事務的讀取和寫入的 **粒度** (**granularity**)。如果資料庫詳細地跟蹤每個事務的活動（細粒度），那麼可以準確地確定哪些事務需要中止，但是簿記開銷可能變得很顯著。簡略的跟蹤速度更快（粗粒度），但可能會導致更多不必要的事務中止。

在某些情況下，事務可以讀取被另一個事務覆蓋的資訊：這取決於發生了什麼，有時可以證明執行結果無論如何都是可序列化的。PostgreSQL 使用這個理論來減少不必要的中止次數【11,41】。

與兩階段鎖定相比，可序列化快照隔離的最大優點是一個事務不需要阻塞等待另一個事務所持有的鎖。就像在快照隔離下一樣，寫不會阻塞讀，反之亦然。這種設計原則使得查詢延遲更可預測，波動更少。特別是，只讀查詢可以執行在一致快照上，而不需要任何鎖定，這對於讀取繁重的工作負載非常有吸引力。

與序列執行相比，可序列化快照隔離並不侷限於單個 CPU 核的吞吐量：FoundationDB 將序列化衝突的檢測分佈在多臺機器上，允許擴充套件到很高的吞吐量。即使資料可能跨多臺機器進行分割槽，事務也可以在保證可序列化隔離等級的同時讀寫多個分割槽中的資料【54】。

中止率顯著影響 SSI 的整體表現。例如，長時間讀取和寫入資料的事務很可能會發生衝突並中止，因此 SSI 要求同時讀寫的事務儘量短（只讀的長事務可能沒問題）。對於慢事務，SSI 可能比兩階段鎖定或序列執行更不敏感。

本章小結

事務是一個抽象層，允許應用程式假裝某些併發問題和某些型別的硬體和軟體故障不存在。各式各樣的錯誤被簡化為一種簡單情況：**事務中止 (transaction abort)**，而應用需要的僅僅是重試。

在本章中介紹了很多問題，事務有助於防止這些問題發生。並非所有應用都易受此類問題影響：具有非常簡單訪問模式的應用（例如每次讀寫單條記錄）可能無需事務管理。但是對於更複雜的訪問模式，事務可以大大減少需要考慮的潛在錯誤情景數量。

如果沒有事務處理，各種錯誤情況（程序崩潰、網路中斷、停電、磁碟已滿、意外併發等）意味著資料可能以各種方式變得不一致。例如，非規範化的資料可能很容易與源資料不同步。如果沒有事務處理，就很難推斷複雜的互動訪問可能對資料庫造成的影響。

本章深入討論了**併發控制**的話題。我們討論了幾個廣泛使用的隔離級別，特別是**讀已提交**、**快照隔離**（有時稱為可重複讀）和**可序列化**。並透過研究競爭條件的各種例子，來描述這些隔離等級：

- 髒讀

一個客戶端讀取到另一個客戶端尚未提交的寫入。**讀已提交** 或更強的隔離級別可以防止髒讀。

- 髒寫

一個客戶端覆蓋寫入了另一個客戶端尚未提交的寫入。幾乎所有的事務實現都可以防止髒寫。

- 讀取偏差（不可重複讀）

在同一個事務中，客戶端在不同的時間點會看見資料庫的不同狀態。**快照隔離** 經常用於解決這個問題，它允許事務從一個特定時間點的一致性快照中讀取資料。快照隔離通常使用**多版本併發控制 (MVCC)** 來實現。

- 丟失更新

兩個客戶端同時執行**讀取 - 修改 - 寫入序列**。其中一個寫操作，在沒有合併另一個寫入變更情況下，直接覆蓋了另一個寫操作的結果。所以導致資料丟失。快照隔離的一些實現可以自動防止這種異常，而另一些實現則需要手動鎖定（`SELECT FOR UPDATE`）。

- 寫入偏差

一個事務讀取一些東西，根據它所看到的值作出決定，並將該決定寫入資料庫。但是，寫入時，該決定的前提不再是真實的。只有可序列化的隔離才能防止這種異常。

- 幻讀

事務讀取符合某些搜尋條件的物件。另一個客戶端進行寫入，影響搜尋結果。快照隔離可以防止直接的幻像讀取，但是寫入偏差上下文中的幻讀需要特殊處理，例如索引範圍鎖定。

弱隔離級別可以防止其中一些異常情況，但要求你，也就是應用程式開發人員手動處理剩餘那些（例如，使用顯式鎖定）。只有可序列化的隔離才能防範所有這些問題。我們討論了實現可序列化事務的三種不同方法：

- 字面意義上的序列執行

如果每個事務的執行速度非常快，並且事務吞吐量足夠低，足以在單個 CPU 核上處理，這是一個簡單而有效的選擇。

- 兩階段鎖定

數十年來，兩階段鎖定一直是實現可序列化的標準方式，但是許多應用出於效能問題的考慮避免使用它。

- 可序列化快照隔離（SSI）

一個相當新的演算法，避免了先前方法的大部分缺點。它使用樂觀的方法，允許事務執行而無需阻塞。當一個事務想要提交時，它會進行檢查，如果執行不可序列化，事務就會被中止。

本章中的示例主要是在關係資料模型的上下文中。但是，正如在“[多物件事務的需求](#)”中所討論的，無論使用哪種資料模型，事務都是有價值的資料庫功能。

本章主要是在單機資料庫的上下文中，探討了各種想法和演算法。分散式資料庫中的事務，則引入了一系列新的困難挑戰，我們將在接下來的兩章中討論。

參考文獻

- Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, et al.: “[A History and Evaluation of System R](#),” *Communications of the ACM*, volume 24, number 10, pages 632–646, October 1981.
[doi:10.1145/358769.358784](https://doi.org/10.1145/358769.358784)
- Jim N. Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger: “[Granularity of Locks and Degrees of Consistency in a Shared Data Base](#),” in *Modelling in Data Base Management Systems: Proceedings of the IFIP Working Conference on Modelling in Data Base Management Systems*, edited by G. M. Nijssen, pages 364–394, Elsevier/North Holland Publishing, 1976. Also in *Readings in Database Systems*, 4th edition, edited by Joseph M. Hellerstein and Michael Stonebraker, MIT Press, 2005. ISBN: 978-0-262-69314-1
- Kapali P. Eswaran, Jim N. Gray, Raymond A. Lorie, and Irving L. Traiger: “[The Notions of Consistency and Predicate Locks in a Database System](#),” *Communications of the ACM*, volume 19, number 11, pages 624–633, November 1976.
- “[ACID Transactions Are Incredibly Helpful](#),” FoundationDB, LLC, 2013.
- John D. Cook: “[ACID Versus BASE for Database Transactions](#),” johndcook.com, July 6, 2009.
- Gavin Clarke: “[NoSQL's CAP Theorem Busters: We Don't Drop ACID](#),” theresister.co.uk, November 22, 2012.
- Theo Härdter and Andreas Reuter: “[Principles of Transaction-Oriented Database Recovery](#),” *ACM Computing Surveys*, volume 15, number 4, pages 287–317, December 1983. [doi:10.1145/289.291](https://doi.org/10.1145/289.291)
- Peter Bailis, Alan Fekete, Ali Ghodsi, et al.: “[HAT, not CAP: Towards Highly Available Transactions](#),” at *14th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2013.
- Armando Fox, Steven D. Gribble, Yatin Chawathe, et al.: “[Cluster-Based Scalable Network Services](#),” at *16th ACM Symposium on Operating Systems Principles* (SOSP), October 1997.
- Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 978-0-201-10715-9, available online at research.microsoft.com.
- Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, et al.: “[Making Snapshot Isolation Serializable](#),” *ACM Transactions on Database Systems*, volume 30, number 2, pages 492–528, June 2005.
[doi:10.1145/1071610.1071615](https://doi.org/10.1145/1071610.1071615)
- Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge: “[Understanding the Robustness of SSDs Under Power Fault](#),” at *11th USENIX Conference on File and Storage Technologies* (FAST), February 2013.
- Laurie Denness: “[SSDs: A Gift and a Curse](#),” laur.ie, June 2, 2015.
- Adam Surak: “[When Solid State Drives Are Not That Solid](#),” blog.algolia.com, June 15, 2015.
- Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, et al.: “[All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications](#),” at *11th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2014.
- Chris Siebenmann: “[Unix's File Durability Problem](#),” utcc.utoronto.ca, April 14, 2016.
- Lakshmi N. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, et al.: “[An Analysis of Data Corruption in the Storage Stack](#),” at *6th USENIX Conference on File and Storage Technologies* (FAST), February 2008.

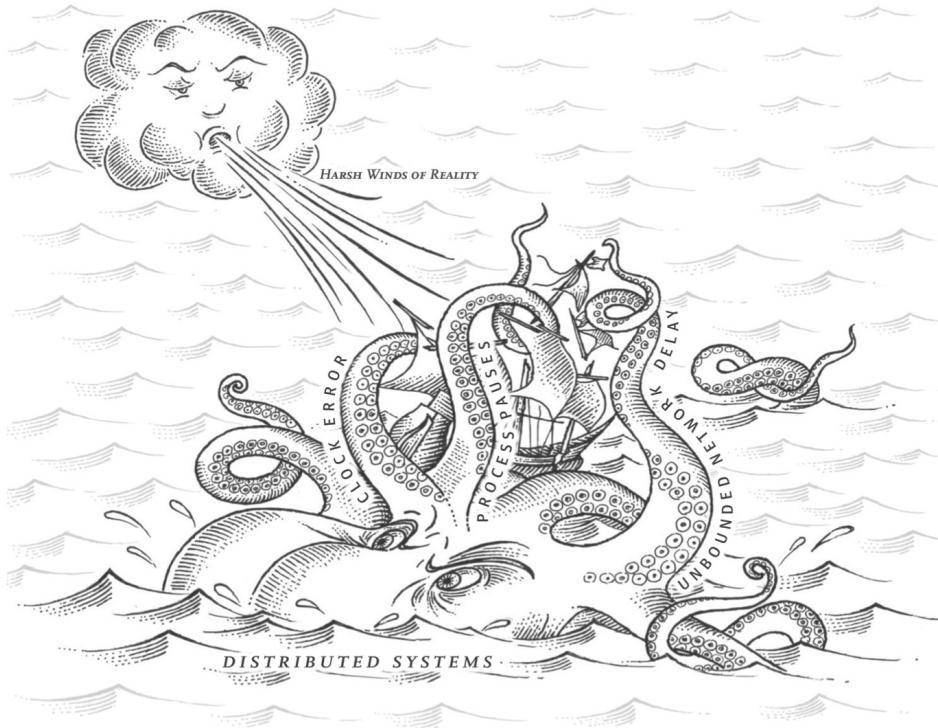
18. Bianca Schroeder, Raghav Lagisetty, and Arif Merchant: “[Flash Reliability in Production: The Expected and the Unexpected](#),” at *14th USENIX Conference on File and Storage Technologies* (FAST), February 2016.
19. Don Allison: “[SSD Storage – Ignorance of Technology Is No Excuse](#),” blog.korelogic.com, March 24, 2015.
20. Dave Scherer: “[Those Are Not Transactions \(Cassandra 2.0\)](#),” blog.foundationdb.com, September 6, 2013.
21. Kyle Kingsbury: “[Call Me Maybe: Cassandra](#),” aphyr.com, September 24, 2013.
22. “[ACID Support in Aerospike](#),” Aerospike, Inc., June 2014.
23. Martin Kleppmann: “[Hermitage: Testing the 'I' in ACID](#),” martin.kleppmann.com, November 25, 2014.
24. Tristan D’Agosta: “[BTC Stolen from Poloniex](#),” bitcointalk.org, March 4, 2014.
25. bitcoincntrf2: “[How I Stole Roughly 100 BTC from an Exchange and How I Could Have Stolen More!](#),” reddit.com, February 2, 2014.
26. Sudhir Jorwekar, Alan Fekete, Krithi Ramamirtham, and S. Sudarshan: “[Automating the Detection of Snapshot Isolation Anomalies](#),” at *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.
27. Michael Melanson: “[Transactions: The Limits of Isolation](#),” michaelmelanson.net, March 20, 2014.
28. Hal Berenson, Philip A. Bernstein, Jim N. Gray, et al.: “[A Critique of ANSI SQL Isolation Levels](#),” at *ACM International Conference on Management of Data* (SIGMOD), May 1995.
29. Atul Adya: “[Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions](#),” PhD Thesis, Massachusetts Institute of Technology, March 1999.
30. Peter Bailis, Aaron Davidson, Alan Fekete, et al.: “[Highly Available Transactions: Virtues and Limitations \(Extended Version\)](#),” at *40th International Conference on Very Large Data Bases* (VLDB), September 2014.
31. Bruce Momjian: “[MVCC Unmasked](#),” momjian.us, July 2014.
32. Annamalai Gurusami: “[Repeatable Read Isolation Level in InnoDB – How Consistent Read View Works](#),” blogs.oracle.com, January 15, 2013.
33. Nikita Prokopov: “[Unofficial Guide to Datomic Internals](#),” tonsky.me, May 6, 2014.
34. Baron Schwartz: “[Immutability, MVCC, and Garbage Collection](#),” xaprb.com, December 28, 2013.
35. J. Chris Anderson, Jan Lehnardt, and Noah Slater: *CouchDB: The Definitive Guide*. O'Reilly Media, 2010. ISBN: 978-0-596-15589-6
36. Rikdeb Mukherjee: “[Isolation in DB2 \(Repeatable Read, Read Stability, Cursor Stability, Uncommitted Read\) with Examples](#),” mframes.blogspot.co.uk, July 4, 2013.
37. Steve Hilker: “[Cursor Stability \(CS\) – IBM DB2 Community](#),” toadworld.com, March 14, 2013.
38. Nate Wiger: “[An Atomic Rant](#),” nateware.com, February 18, 2010.
39. Joel Jacobson: “[Riak 2.0: Data Types](#),” blog.joeljacobson.com, March 23, 2014.
40. Michael J. Cahill, Uwe Röhm, and Alan Fekete: “[Serializable Isolation for Snapshot Databases](#),” at *ACM International Conference on Management of Data* (SIGMOD), June 2008. doi:10.1145/1376616.1376690
41. Dan R. K. Ports and Kevin Grittner: “[Serializable Snapshot Isolation in PostgreSQL](#),” at *38th International Conference on Very Large Databases* (VLDB), August 2012.
42. Tony Andrews: “[Enforcing Complex Constraints in Oracle](#),” tonyandrews.blogspot.co.uk, October 15, 2004.
43. Douglas B. Terry, Marvin M. Theimer, Karin Petersen, et al.: “[Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System](#),” at *15th ACM Symposium on Operating Systems Principles* (SOSP), December 1995. doi:10.1145/224056.224070
44. Gary Fredericks: “[Postgres Serializability Bug](#),” github.com, September 2015.
45. Michael Stonebraker, Samuel Madden, Daniel J. Abadi, et al.: “[The End of an Architectural Era \(It's Time for a Complete Rewrite\)](#),” at *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.
46. John Hugg: “[H-Store/VoltDB Architecture vs. CEP Systems and Newer Streaming Architectures](#),” at *Data @Scale Boston*, November 2014.
47. Robert Kallman, Hideaki Kimura, Jonathan Natkins, et al.: “[H-Store: A High-Performance, Distributed Main Memory Transaction Processing System](#),” *Proceedings of the VLDB Endowment*, volume 1, number 2, pages 1496–1499, August 2008.
48. Rich Hickey: “[The Architecture of Datomic](#),” infoq.com, November 2, 2012.
49. John Hugg: “[Debunking Myths About the VoltDB In-Memory Database](#),” voltedb.com, May 12, 2014.
50. Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton: “[Architecture of a Database System](#),” *Foundations and Trends in Databases*, volume 1, number 2, pages 141–259, November 2007.

[doi:10.1561/1900000002](https://doi.org/10.1561/1900000002)

51. Michael J. Cahill: “[Serializable Isolation for Snapshot Databases](#),” PhD Thesis, University of Sydney, July 2009.
 52. D. Z. Badal: “[Correctness of Concurrency Control and Implications in Distributed Databases](#),” at *3rd International IEEE Computer Software and Applications Conference* (COMPSAC), November 1979.
 53. Rakesh Agrawal, Michael J. Carey, and Miron Livny: “[Concurrency Control Performance Modeling: Alternatives and Implications](#),” *ACM Transactions on Database Systems* (TODS), volume 12, number 4, pages 609–654, December 1987. [doi:10.1145/32204.32220](https://doi.org/10.1145/32204.32220)
 54. Dave Rosenthal: “[Databases at 14.4MHz](#),” *blog.foundationdb.com*, December 10, 2014.
-

上一章	目錄	下一章
第六章：分割槽	設計資料密集型應用	第八章：分散式系統的麻煩

第八章：分散式系統的麻煩



邂逅相遇

網路延遲

存之為吾

無食我數

—— Kyle Kingsbury, Carly Rae Jepsen 《網路分割槽的危害》 (2013 年) 譯著¹

[TOC]

最近幾章中反覆出現的主題是，系統如何處理錯誤的事情。例如，我們討論了 副本故障切換（“處理節點中斷”），複製延遲（“複製延遲問題”）和事務控制（“弱隔離級別”）。當我們瞭解可能在實際系統中出現的各種邊緣情況時，我們會更好地處理它們。

但是，儘管我們已經談了很多錯誤，但之前幾章仍然過於樂觀。現實更加黑暗。我們現在將悲觀主義最大化，假設任何可能出錯的東西 ⁱ 都會出錯。 (經驗豐富的系統運維會告訴你，這是一個合理的假設。如果你問得好，他們可能會一邊治療心理創傷一邊告訴你一些可怕的故事)

ⁱ 除了一個例外：我們將假定故障是非拜占庭式的（請參閱“拜占庭故障”）。 ↵

使用分散式系統與在一臺計算機上編寫軟體有著根本的區別，主要的區別在於，有許多新穎和刺激的方法可以使事情出錯【1,2】。在這一章中，我們將瞭解實踐中出現的問題，理解我們能夠依賴，和不可以依賴的東西。

最後，作為工程師，我們的任務是構建能夠完成工作的系統（即滿足使用者期望的保證），儘管一切都出錯了。在 [第九章](#) 中，我們將看看一些可以在分散式系統中提供這種保證的演算法的例子。但首先，在本章中，我們必須瞭解我們面臨的挑戰。

本章對分散式系統中可能出現的問題進行徹底的悲觀和沮喪的總結。我們將研究網路的問題（“[不可靠的網路](#)”）；時鐘和時序問題（“[不可靠的時鐘](#)”）；我們將討論他們可以避免的程度。所有這些問題的後果都是困惑的，所以我們將探索如何思考一個分散式系統的狀態，以及如何推理發生的事情（“[知識、真相與謊言](#)”）。

故障與部分失效

當你在一臺計算機上編寫一個程式時，它通常會以一種相當可預測的方式執行：無論是工作還是不工作。充滿錯誤的軟體可能會讓人覺得電腦有時候也會有“糟糕的一天”（這種問題通常是重新啟動就恢復了），但這主要是軟體寫得不好的結果。

單個計算機上的軟體沒有根本性的不可靠原因：當硬體正常工作時，相同的操作總是產生相同的結果（這是確定性的）。如果存在硬體問題（例如，記憶體損壞或聯結器鬆動），其後果通常是整個系統故障（例如，核心恐慌，“藍色畫面宕機”，啟動失敗）。裝有良好軟體的個人計算機通常要麼功能完好，要麼完全失效，而不是介於兩者之間。

這是計算機設計中的一個有意的選擇：如果發生內部錯誤，我們寧願電腦完全崩潰，而不是返回錯誤的結果，因為錯誤的結果很難處理。因為計算機隱藏了模糊不清的物理實現，並呈現出一個理想化的系統模型，並以數學一樣的完美的方式運作。CPU 指令總是做同樣的事情；如果你將一些資料寫入記憶體或磁碟，那麼這些資料將保持不變，並且不會被隨機破壞。從第一臺數字計算機開始，始終正確地計算這個設計目標貫穿始終 [\[3\]](#)。

當你編寫執行在多臺計算機上的軟體時，情況有本質上的區別。在分散式系統中，我們不再處於理想化的系統模型中，我們別無選擇，只能面對現實世界的混亂現實。而在現實世界中，各種各樣的事情都可能會出現問題 [\[4\]](#)，如下面的軼事所述：

在我有限的經驗中，我已經和很多東西打過交道：單個 資料中心（DC） 中長期存在的網路分割槽，配電單元 PDU 故障，交換機故障，整個機架的意外重啟，整個資料中心主幹網路故障，整個資料中心的電源故障，以及一個低血糖的司機把他的福特皮卡撞在資料中心的 HVAC（加熱，通風和空調）系統上。而且我甚至不是一個運維。

—— 柯達黑爾

在分散式系統中，儘管系統的其他部分工作正常，但系統的某些部分可能會以某種不可預知的方式被破壞。這被稱為部分失效（**partial failure**）。難點在於部分失效是 不確定性的（**nondeterministic**）：如果你試圖做任何涉及多個節點和網路的事情，它有時可能會工作，有時會出現不可預知的失敗。正如我們將要看到的，你甚至不知道是否成功了，因為訊息透過網路傳播的時間也是不確定的！

這種不確定性和部分失效的可能性，使得分散式系統難以工作 [\[5\]](#)。

雲計算與超級計算機

關於如何構建大型計算系統有一系列的哲學：

- 一個極端是高效能計算（HPC）領域。具有數千個 CPU 的超級計算機通常用於計算密集型科學計算任務，如天氣預報或分子動力學（模擬原子和分子的運動）。
- 另一個極端是 雲計算（**cloud computing**），雲計算並不是一個良好定義的概念 [\[6\]](#)，但通常與多租戶資料中心，連線 IP 網路（通常是乙太網）的商用計算機，彈性 / 按需資源分配以及計量計費等相關聯。
- 傳統企業資料中心位於這兩個極端之間。

不同的哲學會導致不同的故障處理方式。在超級計算機中，作業通常會不時地將計算的狀態存檔到持久儲存中。如果一個節點出現故障，通常的解決方案是簡單地停止整個叢集的工作負載。故障節點修復後，計算從上一個檢查點重新開始 [\[7,8\]](#)。因此，超級計算機更像是一個單節點計算機而不是分散式系統：透過讓部分失敗升級為完全失敗來處理部分失敗——如果系統的任何部分發生故障，只是讓所有的東西都崩潰（就像單臺機器上的核心恐慌一樣）。

在本書中，我們將重點放在實現網際網路服務的系統上，這些系統通常與超級計算機看起來有很大不同：

- 許多與網際網路有關的應用程式都是 **線上 (online)** 的，因為它們需要能夠隨時以低延遲服務使用者。使服務不可用（例如，停止叢集以進行修復）是不可接受的。相比之下，像天氣模擬這樣的離線（批處理）工作可以停止並重新啟動，影響相當小。
- 超級計算機通常由專用硬體構建而成，每個節點相當可靠，節點透過共享記憶體和 **遠端直接記憶體訪問 (RDMA)** 進行通訊。另一方面，雲服務中的節點是由商用機器構建而成的，由於規模經濟，可以以較低的成本提供相同的效能，而且具有較高的故障率。
- 大型資料中心網路通常基於 IP 和乙太網，以 CLOS 拓撲排列，以提供更高的對分（bisection）頻寬【9】。超級計算機通常使用專門的網路拓撲結構，例如多維網格和 Torus 網路【10】，這為具有已知通訊模式的 HPC 工作負載提供了更好的效能。
- 系統越大，其元件之一就越有可能壞掉。隨著時間的推移，壞掉的東西得到修復，新的東西又壞掉，但是在一個有成千上萬個節點的系統中，有理由認為總是有一些東西是壞掉的【7】。當錯誤處理的策略只由簡單放棄組成時，一個大的系統最終會花費大量時間從錯誤中恢復，而不是做有用的工作【8】。
- 如果系統可以容忍發生故障的節點，並繼續保持整體工作狀態，那麼這對於運營和維護非常有用：例如，可以執行滾動升級（請參閱 [第四章](#)），一次重新啟動一個節點，同時繼續給使用者提供不中斷的服務。在雲環境中，如果一臺虛擬機器執行不佳，可以殺死它並請求一臺新的虛擬機器（希望新的虛擬機器速度更快）。
- 在地理位置分散的部署中（保持資料在地理位置上接近使用者以減少訪問延遲），通訊很可能透過網際網路進行，與本地網路相比，通訊速度緩慢且不可靠。超級計算機通常假設它們的所有節點都靠近在一起。

如果要使分散式系統工作，就必須接受部分故障的可能性，並在軟體中建立容錯機制。換句話說，我們需要從不可靠的元件構建一個可靠的系統（正如“[可靠性](#)”中所討論的那樣，沒有完美的可靠性，所以我們需要理解我們可以實際承諾的極限）。

即使在只有少數節點的小型系統中，考慮部分故障也是很重要的。在一個小系統中，很可能大部分元件在大部分時間都正常工作。然而，遲早會有一部分系統出現故障，軟體必須以某種方式處理。故障處理必須是軟體設計的一部分，並且作為軟體的運維，你需要知道在發生故障的情況下，軟體可能會表現出怎樣的行為。

簡單地假設缺陷很罕見並希望始終保持最好的狀況是不明智的。考慮一系列可能的錯誤（甚至是不太可能的錯誤），並在測試環境中人為地建立這些情況來檢視會發生什麼是非常重要的。在分散式系統中，懷疑，悲觀和偏執狂是值得的。

從不可靠的元件構建可靠的系統

你可能想知道這是否有意義——直觀地看來，系統只能像其最不可靠的元件（最薄弱的環節）一樣可靠。事實並非如此：事實上，從不太可靠的潛在基礎構建更可靠的系統是計算機領域的一個古老思想【11】。例如：

- 糾錯碼允許數字資料在通訊通道上準確傳輸，偶爾會出現一些錯誤，例如由於無線網路上的無線電干擾【12】。
- 網際網路協議 (Internet Protocol, IP)** 不可靠：可能丟棄、延遲、重複或重排資料包。傳輸控制協議（Transmission Control Protocol, TCP）在網際網路協議（IP）之上提供了更可靠的傳輸層：它確保丟失的資料包被重新傳輸，消除重複，並且資料包被重新組裝成它們被傳送的順序。

雖然這個系統可以比它的底層部分更可靠，但它的可靠性總是有限的。例如，糾錯碼可以處理少量的單位元錯誤，但是如果你的訊號被幹擾所淹沒，那麼透過通道可以得到多少資料，是有根本性的限制的【13】。TCP 可以隱藏資料包的丟失，重複和重新排序，但是它不能神奇地消除網路中的延遲。

雖然更可靠的高階系統並不完美，但它仍然有用，因為它處理了一些棘手的低階錯誤，所以其餘的錯誤通常更容易推理想和處理。我們將在“[資料庫的端到端原則](#)”中進一步探討這個問題。

不可靠的網路

正如在 [第二部分](#) 的介紹中所討論的那樣，我們在本書中關注的分散式系統是無共享的系統，即透過網路連線的一堆機器。網路是這些機器可以通訊的唯一途徑——我們假設每臺機器都有自己的記憶體和磁碟，一臺機器不能訪問另一臺機器的記憶體或磁碟（除了透過網路向伺服器發出請求）。

無共享 並不是構建系統的唯一方式，但它已經成為構建網際網路服務的主要方式，其原因如下：相對便宜，因為它不需要特殊的硬體，可以利用商品化的雲計算服務，透過跨多個地理分佈的資料中心進行冗餘可以實現高可靠性。

網際網路和資料中心（通常是乙太網）中的大多數內部網路都是 **非同步分組網路 (asynchronous packet networks)**。在這種網路中，一個節點可以向另一個節點發送一個訊息（一個數據包），但是網路不能保證它什麼時候到達，或者是否到達。如果你傳送請求並期待響應，則很多事情可能會出錯（其中一些如 [圖 8-1](#) 所示）：

1. 請求可能已經丟失（可能有人拔掉了網線）。
2. 請求可能正在排隊，稍後將交付（也許網路或接收方過載）。
3. 遠端節點可能已經失效（可能是崩潰或關機）。
4. 遠端節點可能暫時停止了響應（可能會遇到長時間的垃圾回收暫停；請參閱“[程序暫停](#)”），但稍後會再次響應。
5. 遠端節點可能已經處理了請求，但是網路上的響應已經丟失（可能是網路交換機配置錯誤）。
6. 遠端節點可能已經處理了請求，但是響應已經被延遲，並且稍後將被傳遞（可能是網路或者你自己的機器過載）。

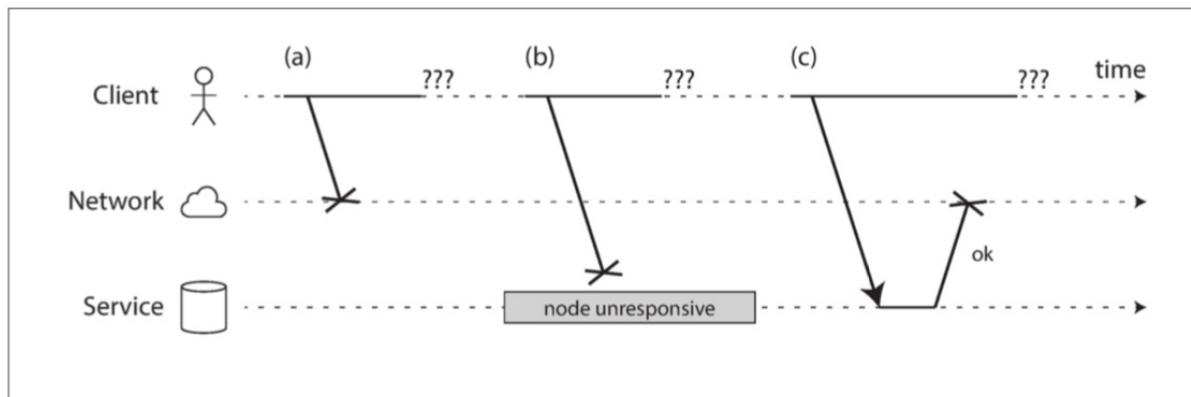


圖 8-1 如果傳送請求並沒有得到響應，則無法區分 (a) 請求是否丟失，(b) 遠端節點是否關閉，或 (c) 響應是否丟失。

傳送者甚至不能分辨資料包是否被傳送：唯一的選擇是讓接收者傳送響應訊息，這可能會丟失或延遲。這些問題在非同步網路中難以區分：你所擁有的唯一資訊是，你尚未收到響應。如果你向另一個節點發送請求並且沒有收到響應，則不可能判斷是什麼原因。

處理這個問題的通常方法是 **超時 (Timeout)**：在一段時間之後放棄等待，並且認為響應不會到達。但是，當發生超時時，你仍然不知道遠端節點是否收到了請求（如果請求仍然在某個地方排隊，那麼即使傳送者已經放棄了該請求，仍然可能會將其傳送給接收者）。

真實世界的網路故障

我們幾十年來一直在建設計算機網路——有人可能希望現在我們已經找出了使網路變得可靠的方法。但是現在似乎還沒有成功。

有一些系統的研究和大量的軼事證據表明，即使在像一家公司運營的資料中心那樣的受控環境中，網路問題也可能出乎意料地普遍。在一家中型資料中心進行的一項研究發現，每個月大約有 12 個網路故障，其中一半斷開一臺機器，一半斷開整個機架【15】。另一項研究測量了架頂式交換機、匯聚交換機和負載平衡器等元件的故障率【16】。它發現新增冗餘網路裝置不會像你所希望的那樣減少故障，因為它不能防範人為錯誤（例如，錯誤配置的交換機），這是造成中斷的主要原因。

諸如 EC2 之類的公有云服務因頻繁的暫態網路故障而臭名昭著【14】，管理良好的私有資料中心網路可能是更穩定的環境。儘管如此，沒有人不受網路問題的困擾：例如，交換機軟體升級過程中的一個問題可能會引發網路拓撲重構，在此期間網路資料包可能會延遲超過一分鐘【17】。鯊魚可能咬住海底電纜並損壞它們【18】。其他令人驚訝的故障包

括網路介面有時會丟棄所有入站資料包，但是成功傳送出站資料包【19】：僅僅因為網路連結在一個方向上工作，並不能保證它也在相反的方向工作。

網路分割槽

當網路的一部分由於網路故障而被切斷時，有時稱為 網路分割槽（network partition）或 網路斷裂（netsplit）。在本書中，我們通常會堅持使用更一般的術語 網路故障（network fault），以避免與 [第六章](#) 討論的儲存系統的分割槽（分片）相混淆。

即使網路故障在你的環境中非常罕見，故障可能發生的事實，意味著你的軟體需要能夠處理它們。無論何時透過網路進行通訊，都可能會失敗，這是無法避免的。

如果網路故障的錯誤處理沒有定義與測試，武斷地講，各種錯誤可能都會發生：例如，即使網路恢復【20】，叢集可能會發生 死鎖，永久無法為請求提供服務，甚至可能會刪除所有的資料【21】。如果軟體被置於意料之外的情況下，它可能會做出出乎意料的事情。

處理網路故障並不意味著容忍它們：如果你的網路通常是相當可靠的，一個有效的方法可能是當你的網路遇到問題時，簡單地向用戶顯示一條錯誤資訊。但是，你確實需要知道你的軟體如何應對網路問題，並確保系統能夠從中恢復。有意識地觸發網路問題並測試系統響應（這是 Chaos Monkey 背後的想法；請參閱“[可靠性](#)”）。

檢測故障

許多系統需要自動檢測故障節點。例如：

- 負載平衡器需要停止向已死亡的節點轉發請求（從輪詢列表移出，即 out of rotation）。
- 在單主複製功能的分散式資料庫中，如果主庫失效，則需要將從庫之一升級為新主庫（請參閱“[處理節點宕機](#)”）。

不幸的是，網路的不確定性使得很難判斷一個節點是否工作。在某些特定的情況下，你可能會收到一些反饋資訊，明確告訴你某些事情沒有成功：

- 如果你可以連線到執行節點的機器，但沒有程序正在偵聽目標埠（例如，因為程序崩潰），作業系統將透過傳送 FIN 或 RST 來關閉並重用 TCP 連線。但是，如果節點在處理請求時發生崩潰，則無法知道遠端節點實際處理了多少資料【22】。
- 如果節點程序崩潰（或被管理員殺死），但節點的作業系統仍在執行，則指令碼可以通知其他節點有關該崩潰的資訊，以便另一個節點可以快速接管，而無需等待超時到期。例如，HBase 就是這麼做的【23】。
- 如果你有權訪問資料中心網路交換機的管理介面，則可以透過它們檢測硬體級別的鏈路故障（例如，遠端機器是否關閉電源）。如果你透過網際網路連線，或者如果你處於共享資料中心而無法訪問交換機，或者由於網路問題而無法訪問管理介面，則排除此選項。
- 如果路由器確認你嘗試連線的 IP 地址不可用，則可能會使用 ICMP 目標不可達資料包回覆你。但是，路由器不具備神奇的故障檢測能力——它受到與網路其他參與者相同的限制。

關於遠端節點關閉的快速反饋很有用，但是你不能指望它。即使 TCP 確認已經傳送了一個數據包，應用程式在處理之前可能已經崩潰。如果你想確保一個請求是成功的，你需要應用程式本身的正確響應【24】。

相反，如果出了什麼問題，你可能會在堆疊的某個層次上得到一個錯誤響應，但總的來說，你必須假設你可能根本就得不到任何回應。你可以重試幾次（TCP 重試是透明的，但是你也可以在應用程式級別重試），等待超時過期，並且如果在超時時間內沒有收到響應，則最終宣告節點已經死亡。

超時與無窮的延遲

如果超時是檢測故障的唯一可靠方法，那麼超時應該等待多久？不幸的是沒有簡單的答案。

長時間的超時意味著長時間等待，直到一個節點被宣告死亡（在這段時間內，使用者可能不得不等待，或者看到錯誤資訊）。短的超時可以更快地檢測到故障，但有更高地風險誤將一個節點宣佈為失效，而該節點實際上只是暫時地變慢了（例如由於節點或網路上的負載峰值）。

過早地宣告一個節點已經死了是有問題的：如果這個節點實際上是活著的，並且正在執行一些動作（例如，傳送一封電子郵件），而另一個節點接管，那麼這個動作可能會最終執行兩次。我們將在“[知識、真相與謊言](#)”以及[第九章](#)和[第十一章](#)中更詳細地討論這個問題。

當一個節點被宣告死亡時，它的職責需要轉移到其他節點，這會給其他節點和網路帶來額外的負擔。如果系統已經處於高負荷狀態，則過早宣告節點死亡會使問題更嚴重。特別是如果節點實際上沒有死亡，只是由於過載導致其響應緩慢；這時將其負載轉移到其他節點可能會導致 **級聯失效**（即 cascading failure，表示在極端情況下，所有節點都宣告對方死亡，所有節點都將停止工作）。

設想一個虛構的系統，其網路可以保證資料包的最大延遲——每個資料包要麼在一段時間內傳送，要麼丟失，但是傳遞永遠不會比 $d + r$ 更長。此外，假設你可以保證一個非故障節點總是在一段時間 r 內處理一個請求。在這種情況下，你可以保證每個成功的請求在 $2d + r$ 時間內都能收到響應，如果你在此時間內沒有收到響應，則知道網路或遠端節點不工作。如果這是成立的， $2d + r$ 會是一個合理的超時設定。

不幸的是，我們所使用的大多數系統都沒有這些保證：非同步網路具有無限的延遲（即儘可能快地傳送資料包，但資料包到達可能需要的時間沒有上限），並且大多數伺服器實現並不能保證它們可以在一定的最大時間內處理請求（請參閱“[響應時間保證](#)”）。對於故障檢測，即使系統大部分時間快速執行也是不夠的：如果你的超時時間很短，往返時間只需要一個瞬時尖峰就可以使系統失衡。

網路擁塞和排隊

在駕駛汽車時，由於交通擁堵，道路交通網路的通行時間往往不盡相同。同樣，計算機網路上資料包延遲的可變性通常是由於排隊【25】：

- 如果多個不同的節點同時嘗試將資料包傳送到同一目的地，則網路交換機必須將它們排隊並將它們逐個送入目標網路鏈路（如 圖 8-2 所示）。在繁忙的網路鏈路上，資料包可能需要等待一段時間才能獲得一個插槽（這稱為網路擁塞）。如果傳入的資料太多，交換機佇列填滿，資料包將被丟棄，因此需要重新發送資料包 - 即使網路執行良好。
- 當資料包到達目標機器時，如果所有 CPU 核心當前都處於繁忙狀態，則來自網路的傳入請求將被作業系統排隊，直到應用程式準備好處理它為止。根據機器上的負載，這可能需要一段任意的時間。
- 在虛擬化環境中，正在執行的作業系統經常暫停幾十毫秒，因為另一個虛擬機器正在使用 CPU 核心。在這段時間內，虛擬機器不能從網路中消耗任何資料，所以傳入的資料被虛擬機器監視器【26】排隊（緩衝），進一步增加了網路延遲的可變性。
- TCP 執行 **流量控制**（flow control，也稱為 **擁塞避免**，即 congestion avoidance，或 **背壓**，即 backpressure），其中節點會限制自己的傳送速率以避免網路鏈路或接收節點過載【27】。這意味著甚至在資料進入網路之前，在傳送者處就需要進行額外的排隊。

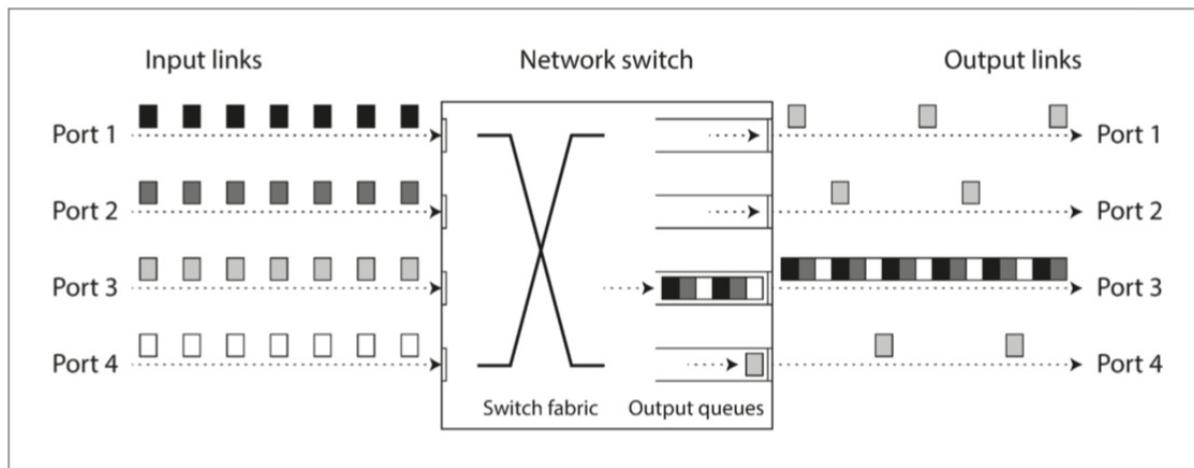


圖 8-2 如果有多臺機器將網路流量傳送到同一目的地，則其交換機佇列可能會被填滿。在這裡，埠 1,2 和 4 都試圖傳送資料包到埠 3

而且，如果 TCP 在某個超時時間內沒有被確認（這是根據觀察的往返時間計算的），則認為資料包丟失，丟失的資料包將自動重新發送。儘管應用程式沒有看到資料包丟失和重新傳輸，但它看到了延遲（等待超時到期，然後等待重新傳輸的資料包得到確認）。

TCP與UDP

一些對延遲敏感的應用程式，比如視訊會議和 IP 語音（VoIP），使用了 UDP 而不是 TCP。這是在可靠性和和延遲變化之間的折衷：由於 UDP 不執行流量控制並且不重傳丟失的分組，所以避免了網路延遲變化的一些原因（儘管它仍然易受切換佇列和排程延遲的影響）。

在延遲資料毫無價值的情況下，UDP 是一個不錯的選擇。例如，在 VoIP 電話呼叫中，可能沒有足夠的時間重新發送丟失的資料包，並在揚聲器上播放資料。在這種情況下，重發資料包沒有意義——應用程式必須使用靜音填充丟失資料包的時隙（導致聲音短暫中斷），然後在資料流中繼續。重試發生在人類層（“你能再說一遍嗎？聲音剛剛斷了一會兒。”）。

所有這些因素都會造成網路延遲的變化。當系統接近其最大容量時，排隊延遲的變化範圍特別大：擁有足夠備用容量的系統可以輕鬆排空佇列，而在高利用率的系統中，很快就能積累很長的佇列。

在公共雲和多租戶資料中心中，資源被許多客戶共享：網路連結和交換機，甚至每個機器的網絡卡和 CPU（在虛擬機器上執行時）。批處理工作負載（如 MapReduce，請參閱 [第十章](#)）能夠很容易使網路連結飽和。由於無法控制或瞭解其他客戶對共享資源的使用情況，如果附近的某個人（嘈雜的鄰居）正在使用大量資源，則網路延遲可能會發生劇烈變化【28,29】。

在這種環境下，你只能透過實驗方式選擇超時：在一段較長的時期內、在多臺機器上測量網路往返時間的分佈，以確定延遲的預期變化。然後，考慮到應用程式的特性，可以確定 [故障檢測延遲與過早超時風險](#) 之間的適當折衷。

更好的一種做法是，系統不是使用配置的常量超時時間，而是連續測量響應時間及其變化（抖動），並根據觀察到的響應時間分佈自動調整超時時間。這可以透過 Phi Accrual 故障檢測器【30】來完成，該檢測器在例如 Akka 和 Cassandra【31】中使用。TCP 的超時重傳機制也是以類似的方式工作【27】。

同步網路與非同步網路

如果我們可以依靠網路來傳遞一些 [最大延遲固定](#) 的資料包，而不是丟棄資料包，那麼分散式系統就會簡單得多。為什麼我們不能在硬體層面上解決這個問題，使網路可靠，使軟體不必擔心呢？

為了回答這個問題，將資料中心網路與非常可靠的傳統固定電話網路（非蜂窩，非 VoIP）進行比較是很有趣的：延遲音訊幀和掉話是非常罕見的。一個電話需要一個很低的端到端延遲，以及足夠的頻寬來傳輸你聲音的音訊取樣資料。在計算機網路中有類似的可靠性和可預測性不是很好嗎？

當你透過電話網路撥打電話時，它會建立一個電路：在兩個呼叫者之間的整個路線上為呼叫分配一個固定的，有保證的頻寬量。這個電路會保持至通話結束【32】。例如，ISDN 網路以每秒 4000 幀的固定速率執行。呼叫建立時，每個幀內（每個方向）分配 16 位空間。因此，在通話期間，每一方都保證能夠每 250 微秒傳送一個精確的 16 位音訊資料【33,34】。

這種網路是同步的：即使資料經過多個路由器，也不會受到排隊的影響，因為呼叫的 16 位空間已經在網路的下一跳中保留了下來。而且由於沒有排隊，網路的最大端到端延遲是固定的。我們稱之為 [有限延遲（bounded delay）](#)。

我們不能簡單地使網路延遲可預測嗎？

請注意，電話網路中的電路與 TCP 連線有很大不同：電路是固定數量的預留頻寬，在電路建立時沒有其他人可以使用，而 TCP 連線的資料包 [機會性地](#) 使用任何可用的網路頻寬。你可以給 TCP 一個可變大小的資料塊（例如，一個電子郵件或一個網頁），它會盡可能在最短的時間內傳輸它。TCP 連線空閒時，不使用任何頻寬ⁱⁱ。

ⁱⁱ. 除了偶爾的 keepalive 資料包，如果 TCP keepalive 被啟用。 ↵

如果資料中心網路和網際網路是電路交換網路，那麼在建立電路時就可以建立一個受保證的最大往返時間。但是，它們並不能這樣：乙太網和 IP 是 **分組交換協議**，不得不忍受排隊的折磨和因此導致的網路無限延遲，這些協議沒有電路的概念。

為什麼資料中心網路和網際網路使用分組交換？答案是，它們針對 **突發流量（bursty traffic）** 進行了最佳化。一個電路適用於音訊或視訊通話，在通話期間需要每秒傳送相當數量的位元。另一方面，請求網頁，傳送電子郵件或傳輸檔案沒有任何特定的頻寬要求——我們只是希望它儘快完成。

如果想透過電路傳輸檔案，你得預測一個頻寬分配。如果你猜的太低，傳輸速度會不必要的太慢，導致網路容量閒置。如果你猜的太高，電路就無法建立（因為如果無法保證其頻寬分配，網路不能建立電路）。因此，將電路用於突發資料傳輸會浪費網路容量，並且使傳輸不必要的緩慢。相比之下，TCP 動態調整資料傳輸速率以適應可用的網路容量。

已經有一些嘗試去建立同時支援電路交換和分組交換的混合網路，比如 ATMⁱⁱⁱ。InfiniBand 有一些相似之處【35】：它在鏈路層實現了端到端的流量控制，從而減少了在網路中排隊的需要，儘管它仍然可能因鏈路擁塞而受到延遲【36】。透過仔細使用 **服務質量**（quality of service，即 QoS，資料包的優先順序和排程）和 **准入控制**（admission control，限速傳送器），可以在分組網路上類比電路交換，或提供統計上的 **有限延遲**【25,32】。

ⁱⁱⁱ. **非同步傳輸模式（Asynchronous Transfer Mode, ATM）** 在 20 世紀 80 年代是乙太網的競爭對手【32】，但在電話網核心交換機之外並沒有得到太多的採用。它與自動櫃員機（也稱為自動取款機）無關，儘管共用一個縮寫詞。或許，在一些平行的世界裡，網際網路是基於像 ATM 這樣的東西，因此它們的網際網路視訊通話可能比我們的更可靠，因為它們不會遭受包的丟失和延遲。 ↪

但是，目前在多租戶資料中心和公共雲或透過網際網路^{iv} 進行通訊時，此類服務質量尚未啟用。當前部署的技術不允許我們對網路的延遲或可靠性作出任何保證：我們必須假設網路擁塞，排隊和無限的延遲總是會發生。因此，超時時間沒有“正確”的值——它需要透過實驗來確定。

^{iv}. 網際網路服務提供商之間的對等協議和透過 **BGP 閘道器協議（BGP）** 建立的路由，與 IP 協議相比，更接近於電路交換。在這個級別上，可以購買專用頻寬。但是，網際網路路由在網路級別執行，而不是主機之間的單獨連線，而且執行時間要長得多。 ↪

延遲和資源利用

更一般地說，可以將 **延遲變化** 視為 **動態資源分割槽** 的結果。

假設兩臺電話交換機之間有一條線路，可以同時進行 10,000 個呼叫。透過此線路切換的每個電路都佔用其中一個呼叫插槽。因此，你可以將線路視為可由多達 10,000 個併發使用者共享的資源。資源以靜態方式分配：即使你現在是線路上唯一的呼叫，並且所有其他 9,999 個插槽都未使用，你的電路仍將分配與線路充分利用時相同的固定數量的頻寬。

相比之下，網際網路動態分享網路頻寬。傳送者互相推擠和爭奪，以讓他們的資料包儘可能快地透過網路，並且網路交換機決定從一個時刻到另一個時刻傳送哪個分組（即，頻寬分配）。這種方法有排隊的缺點，但其優點是它最大限度地利用了線路。線路固定成本，所以如果你更好地利用它，你透過線路傳送的每個位元組都會更便宜。

CPU 也會出現類似的情況：如果你在多個執行緒間動態共享每個 CPU 核心，則一個執行緒有時必須在作業系統的執行佇列裡等待，而另一個執行緒正在執行，這樣每個執行緒都有可能被暫停一個不定的時間長度。但是，與為每個執行緒分配靜態數量的 CPU 週期相比，這會更好地利用硬體（請參閱“[響應時間保證](#)”）。更好的硬體利用率也是使用虛擬機器的重要動機。

如果資源是靜態分割槽的（例如，專用硬體和專用頻寬分配），則在某些環境中可以實現 **延遲保證**。但是，這是降低利用率為代價的——換句話說，它是更昂貴的。另一方面，動態資源分配的多租戶提供了更好的利用率，所以它更便宜，但它具有可變延遲的缺點。

網路中的可變延遲不是一種自然規律，而只是成本 / 收益權衡的結果。

不可靠的時鐘

時鐘和時間很重要。應用程式以各種方式依賴於時鐘來回答以下問題：

1. 這個請求是否超時了？
2. 這項服務的第 99 百分位響應時間是多少？
3. 在過去五分鐘內，該服務平均每秒處理多少個查詢？
4. 使用者在我們的網站上花了多長時間？
5. 這篇文章在何時釋出？
6. 在什麼時間傳送提醒郵件？
7. 這個快取條目何時到期？
8. 日誌檔案中此錯誤訊息的時間戳是什麼？

例 1-4 測量了 持續時間 (durations，例如，請求傳送與響應接收之間的時間間隔) ，而 **例 5-8** 描述了 時間點 (point in time，在特定日期和和特定時間發生的事件) 。

在分散式系統中，時間是一件棘手的事情，因為通訊不是即時的：訊息透過網路從一臺機器傳送到另一臺機器需要時間。收到訊息的時間總是晚於傳送的時間，但是由於網路中的可變延遲，我們不知道晚了多少時間。這個事實導致有時很難確定在涉及多臺機器時發生事情的順序。

而且，網路上的每臺機器都有自己的時鐘，這是一個實際的硬體裝置：通常是石英晶體振盪器。這些裝置不是完全準確的，所以每臺機器都有自己的時間概念，可能比其他機器稍快或更慢。可以在一定程度上同步時鐘：最常用的機制是 網路時間協議 (NTP) ，它允許根據一組伺服器報告的時間來調整計算機時鐘【37】。伺服器則從更精確的時間源（如 GPS 接收機）獲取時間。

單調鍾與日曆時鐘

現代計算機至少有兩種不同的時鐘：日曆時鐘 (time-of-day clock) 和單調鍾 (monotonic clock) 。儘管它們都衡量時間，但區分這兩者很重要，因為它們有不同的目的。

日曆時鐘

日曆時鐘是你直觀地瞭解時鐘的依據：它根據某個日曆（也稱為 掛鐘時間，即 wall-clock time）返回當前日期和時間。例如，Linux 上的 `clock_gettime(CLOCK_REALTIME)`^v 和 Java 中的 `System.currentTimeMillis()` 返回自 epoch (UTC 時間 1970 年 1 月 1 日午夜) 以來的秒數（或毫秒），根據公曆 (Gregorian) 日曆，不包括閏秒。有些系統使用其他日期作為參考點。

^v. 雖然該時鐘被稱為實時時鐘，但它與實時作業系統無關，如“響應時間保證”中所述。 ↵

日曆時鐘通常與 NTP 同步，這意味著來自一臺機器的時間戳（理想情況下）與另一臺機器上的時間戳相同。但是如下節所述，日曆時鐘也具有各種各樣的奇特之處。特別是，如果本地時鐘在 NTP 伺服器之前太遠，則它可能會被強制重置，看上去好像跳回了先前的時間點。這些跳躍以及他們經常忽略閏秒的事實，使日曆時鐘不能用於測量經過時間 (elapsed time) 【38】。

歷史上的日曆時鐘還具有相當粗略的解析度，例如，在較早的 Windows 系統上以 10 毫秒為單位前進【39】。在最近的系統中這已經不是一個問題了。

單調鍾

單調鍾適用於測量持續時間（時間間隔），例如超時或服務的響應時間：Linux 上的 `clock_gettime(CLOCK_MONOTONIC)`，和 Java 中的 `System.nanoTime()` 都是單調時鐘。這個名字來源於他們保證總是往前走的事實（而日曆時鐘可以往回跳）。

你可以在某個時間點檢查單調鍒的值，做一些事情，且稍後再次檢查它。這兩個值之間的差異告訴你兩次檢查之間經過了多長時間。但單調鍒的絕對值是毫無意義的：它可能是計算機啟動以來的納秒數，或類似的任意值。特別是比較來自兩臺不同計算機的單調鍒的值是沒有意義的，因為它們並不是一回事。

在具有多個 CPU 插槽的伺服器上，每個 CPU 可能有一個單獨的計時器，但不一定與其他 CPU 同步。作業系統會補償所有的差異，並嘗試嚮應用執行緒表現出單調鐘的樣子，即使這些執行緒被排程到不同的 CPU 上。當然，明智的做法是不要太把這種單調性保證當回事【40】。

如果 NTP 協議檢測到計算機的本地石英鐘比 NTP 伺服器要更快或更慢，則可以調整單調鐘向前走的頻率（這稱為 **偏移 (skewing) 時鐘**）。預設情況下，NTP 允許時鐘速率增加或減慢最高至 0.05%，但 NTP 不能使單調時鐘向前或向後跳轉。單調時鐘的解析度通常相當好：在大多數系統中，它們能在幾微秒或更短的時間內測量時間間隔。

在分散式系統中，使用單調鐘測量 經過時間 (elapsed time，比如超時) 通常很好，因為它不假定不同節點的時鐘之間存在任何同步，並且對測量的輕微不準確性不敏感。

時鐘同步與準確性

單調鐘不需要同步，但是日曆時鐘需要根據 NTP 伺服器或其他外部時間源來設定才能有用。不幸的是，我們獲取時鐘的方法並不像你所希望的那樣可靠或準確——硬體時鐘和 NTP 可能會變幻莫測。舉幾個例子：

- 計算機中的石英鐘不夠精確：它會 **漂移 (drifts)**，即執行速度快於或慢於預期）。時鐘漂移取決於機器的溫度。Google 假設其伺服器時鐘漂移為 200 ppm (百萬分之一) 【41】，相當於每 30 秒與伺服器重新同步一次的時鐘漂移為 6 毫秒，或者每天重新同步的時鐘漂移為 17 秒。即使一切工作正常，此漂移也會限制可以達到的最佳準確度。
- 如果計算機的時鐘與 NTP 伺服器的時鐘差別太大，可能會拒絕同步，或者本地時鐘將被強制重置【37】。任何觀察重置前後時間的應用程式都可能會看到時間倒退或突然跳躍。
- 如果某個節點被 NTP 伺服器的防火牆意外阻塞，有可能會持續一段時間都沒有人會注意到。有證據表明，這在實踐中確實發生過。
- NTP 同步只能和網路延遲一樣好，所以當你在擁有可變資料包延遲的擁塞網路上時，NTP 同步的準確性會受到限制。一個實驗表明，當透過網際網路同步時，35 毫秒的最小誤差是可以實現的，儘管偶爾的網路延遲峰值會導致大約一秒的誤差。根據配置，較大的網路延遲會導致 NTP 客戶端完全放棄。
- 一些 NTP 伺服器是錯誤的或者配置錯誤的，報告的時間可能相差幾個小時【43,44】。還好 NTP 客戶端非常健壯，因為他們會查詢多個伺服器並忽略異常值。無論如何，依賴於網際網路上的陌生人所告訴你的時間來保證你的系統的正確性，這還挺讓人擔憂的。
- 閏秒導致一分鐘可能有 59 秒或 61 秒，這會打破一些在設計之時未考慮閏秒的系統的時序假設【45】。閏秒已經使許多大型系統崩潰的事實【38,46】說明了，關於時鐘的錯誤假設是多麼容易偷偷溜入系統中。處理閏秒的最佳方法可能是讓 NTP 伺服器“撒謊”，並在一天中逐漸執行閏秒調整（這被稱為 **拖尾**，即 smearing）【47,48】，雖然實際的 NTP 伺服器表現各異【49】。
- 在虛擬機器中，硬體時鐘被虛擬化，這對於需要精確計時的應用程式提出了額外的挑戰【50】。當一個 CPU 核心在虛擬機器之間共享時，每個虛擬機器都會暫停幾十毫秒，與此同時另一個虛擬機器正在執行。從應用程式的角度來看，這種停頓表現為時鐘突然向前跳躍【26】。
- 如果你在沒有完整控制權的裝置（例如，移動裝置或嵌入式裝置）上執行軟體，則可能完全不能信任該裝置的硬體時鐘。一些使用者故意將其硬體時鐘設定為不正確的日期和時間，例如，為了規避遊戲中的時間限制，時鐘可能會被設定到很遠的過去或將來。

如果你足夠在乎這件事並投入大量資源，就可以達到非常好的時鐘精度。例如，針對金融機構的歐洲法規草案 MiFID II 要求所有高頻率交易基金在 UTC 時間 100 微秒內同步時鐘，以便除錯“閃崩”等市場異常現象，並幫助檢測市場操縱【51】。

透過 GPS 接收機，精確時間協議 (PTP) 【52】以及仔細的部署和監測可以實現這種精確度。然而，這需要很多努力和專業知識，而且有很多東西都會導致時鐘同步錯誤。如果你的 NTP 守護程序配置錯誤，或者防火牆阻止了 NTP 通訊，由漂移引起的時鐘誤差可能很快就會變大。

依賴同步時鐘

時鐘的問題在於，雖然它們看起來簡單易用，但卻具有令人驚訝的缺陷：一天可能不會有精確的 86,400 秒，日曆時鐘可能會前後跳躍，而一個節點上的時間可能與另一個節點上的時間完全不同。

本章早些時候，我們討論了網路丟包和任意延遲包的問題。儘管網路在大多數情況下表現良好，但軟體的設計必須假定網路偶爾會出現故障，而軟體必須正常處理這些故障。時鐘也是如此：儘管大多數時間都工作得很好，但需要準備健壯的軟體來處理不正確的時鐘。

有一部分問題是，不正確的時鐘很容易被視而不見。如果一臺機器的 CPU 出現故障或者網路配置錯誤，很可能根本無法工作，所以很快就會被注意和修復。另一方面，如果它的石英時鐘有缺陷，或者它的 NTP 客戶端配置錯誤，大部分事情似乎仍然可以正常工作，即使它的時鐘逐漸偏離現實。如果某個軟體依賴於精確同步的時鐘，那麼結果更可能是悄無聲息的，僅有微量的資料丟失，而不是一次驚天動地的崩潰 [53,54]。

因此，如果你使用需要同步時鐘的軟體，必須仔細監控所有機器之間的時鐘偏移。時鐘偏離其他時鐘太遠的節點應當被宣告死亡，並從叢集中移除。這樣的監控可以確保你在損失發生之前注意到破損的時鐘。

有序事件的時間戳

讓我們考慮一個特別的情況，一件很有誘惑但也很危險的事情：依賴時鐘，在多個節點上對事件進行排序。例如，如果兩個客戶端寫入分散式資料庫，誰先到達？哪一個更近？

圖 8-3 顯示了在具有多主複製的資料庫中對時鐘的危險使用（該例子類似於 圖 5-9）。客戶端 A 在節點 1 上寫入 $x = 1$ ；寫入被複制到節點 3；客戶端 B 在節點 3 上增加 x （我們現在有 $x = 2$ ）；最後這兩個寫入都被複制到節點 2。

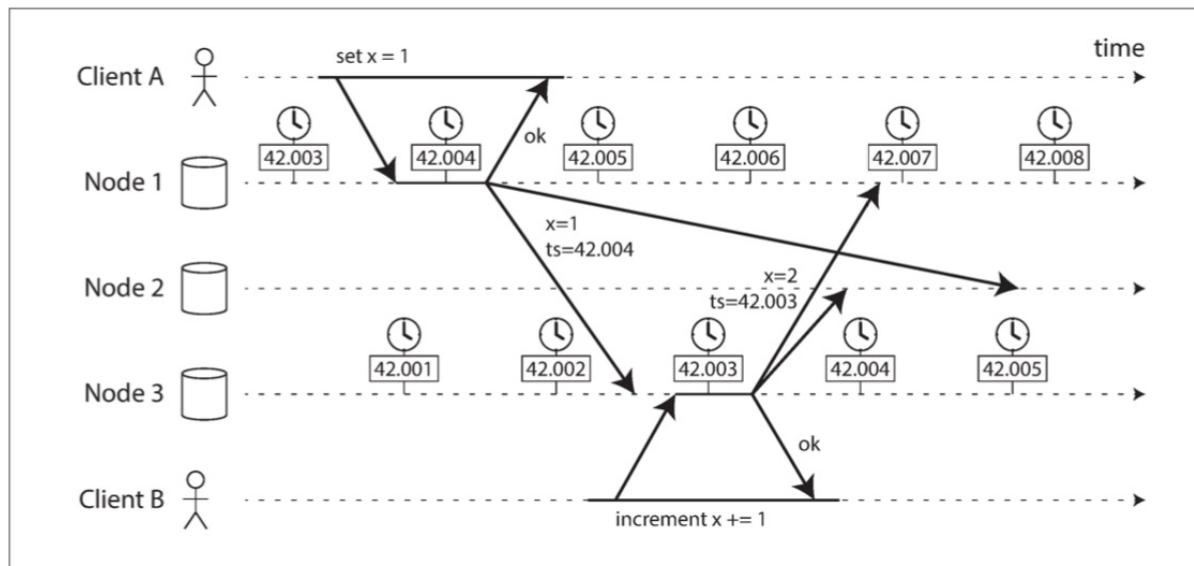


圖 8-3 客戶端 B 的寫入比客戶端 A 的寫入要晚，但是 B 的寫入具有較早的時間戳。

在 圖 8-3 中，當一個寫入被複制到其他節點時，它會根據發生寫入的節點上的日曆時鐘標記一個時間戳。在這個例子中，時鐘同步是非常好的：節點 1 和節點 3 之間的偏差小於 3ms，這可能比你在實踐中能預期的更好。

儘管如此，圖 8-3 中的時間戳卻無法正確排列事件：寫入 $x = 1$ 的時間戳為 42.004 秒，但寫入 $x = 2$ 的時間戳為 42.003 秒，即使 $x = 2$ 在稍後出現。當節點 2 接收到這兩個事件時，會錯誤地推斷出 $x = 1$ 是最近的值，而丟棄寫入 $x = 2$ 。效果上表現為，客戶端 B 的增量操作會丟失。

這種衝突解決策略被稱為 最後寫入勝利 (LWW)，它在多主複製和無主資料庫（如 Cassandra [53] 和 Riak [54]）中被廣泛使用（請參閱 “最後寫入勝利（丟棄併發寫入）” 一節）。有些實現會在客戶端而不是伺服器上生成時間戳，但這並不能改變 LWW 的基本問題：

- 資料庫寫入可能會神秘地消失：具有滯後時鐘的節點無法覆蓋之前具有快速時鐘的節點寫入的值，直到節點之間的時鐘偏差消逝 [54,55]。此方案可能導致一定數量的資料被悄悄丟棄，而未嚮應用報告任何錯誤。
- LWW 無法區分 高頻順序寫入（在 圖 8-3 中，客戶端 B 的增量操作 一定 發生在客戶端 A 的寫入之後）和 真正併發寫入（寫入者意識不到其他寫入者）。需要額外的因果關係跟蹤機制（例如版本向量），以防止違背因果關係（請參閱 “檢測併發寫入”）。
- 兩個節點很可能獨立地生成具有相同時間戳的寫入，特別是在時鐘僅具有毫秒解析度的情況下。為了解決這樣的衝

突，還需要一個額外的 決勝值 (tiebreaker，可以簡單地是一個大隨機數)，但這種方法也可能會導致違背因果關係【53】。

因此，儘管透過保留“最近”的值並放棄其他值來解決衝突是很誘惑人的，但是要注意，“最近”的定義取決於本地的日曆時鐘，這很可能是不正確的。即使使用嚴格同步的 NTP 時鐘，一個數據包也可能在時間戳 100 毫秒（根據傳送者的時鐘）時傳送，並在時間戳 99 毫秒（根據接收者的時鐘）處到達——看起來好像資料包在傳送之前已經到達，這是不可能的。

NTP 同步是否能足夠準確，以至於這種不正確的排序不會發生？也許不能，因為 NTP 的同步精度本身，除了石英鐘漂移這類誤差源之外，還受到網路往返時間的限制。為了進行正確的排序，你需要一個比測量物件（即網路延遲）要精確得多的時鐘。

所謂的 **邏輯時鐘 (logic clock)** 【56,57】是基於遞增計數器而不是振盪石英晶體，對於排序事件來說是更安全的選擇（請參閱“[檢測併發寫入](#)”）。邏輯時鐘不測量一天中的時間或經過的秒數，而僅測量事件的相對順序（無論一個事件發生在另一個事件之前還是之後）。相反，用來測量實際經過時間的 **日曆時鐘** 和 **單調鍾** 也被稱為 **物理時鐘 (physical clock)**。我們將在“[順序保證](#)”中來看順序問題。

時鐘讀數存在置信區間

你可能能夠以微秒或甚至納秒的精度讀取機器的時鐘。但即使可以得到如此細緻的測量結果，這並不意味著這個值對於這樣的精度實際上是準確的。實際上，大機率是不準確的——如前所述，即使你每分鐘與本地網路上的 NTP 伺服器進行同步，幾毫秒的時間漂移也很容易在不精確的石英時鐘上發生。使用公共網際網路上的 NTP 伺服器，最好的準確度可能達到幾十毫秒，而且當網路擁塞時，誤差可能會超過 100 毫秒【57】。

因此，將時鐘讀數視為一個時間點是沒有意義的——它更像是一段時間範圍：例如，一個系統可能以 95% 的置信度認為當前時間處於本分鐘內的第 10.3 秒和 10.5 秒之間，它可能沒法比這更精確了【58】。如果我們只知道 ±100 毫秒的時間，那麼時間戳中的微秒數字部分基本上是沒有意義的。

不確定性界限可以根據你的時間源來計算。如果你的 GPS 接收器或原子（銦）時鐘直接連線到你的計算機上，預期的錯誤範圍由製造商告知。如果從伺服器獲得時間，則不確定性取決於自上次與伺服器同步以來的石英鐘漂移的期望值，加上 NTP 伺服器的不確定性，再加上到伺服器的網路往返時間（只是獲取粗略近似值，並假設伺服器是可信的）。

不幸的是，大多數系統不公開這種不確定性：例如，當呼叫 `clock_gettime()` 時，返回值不會告訴你時間戳的預期錯誤，所以你不知道其置信區間是 5 毫秒還是 5 年。

一個有趣的例外是 Spanner 中的 Google TrueTime API 【41】，它明確地報告了本地時鐘的置信區間。當你詢問當前時間時，你會得到兩個值：[最早，最晚]，這是最早可能的時間戳和最晚可能的時間戳。在不確定性估計的基礎上，時鐘知道當前的實際時間落在該區間內。區間的寬度取決於自從本地石英鐘最後與更精確的時鐘源同步以來已經過了多長時間。

全域性快照的同步時鐘

在“[快照隔離和可重複讀](#)”中，我們討論了快照隔離，這是資料庫中非常有用的功能，需要支援小型快速讀寫事務和大型長時間執行的只讀事務（用於備份或分析）。它允許只讀事務看到特定時間點的處於一致狀態的資料庫，且不會鎖定和干擾讀寫事務。

快照隔離最常見的實現需要單調遞增的事務 ID。如果寫入比快照晚（即，寫入具有比快照更大的事務 ID），則該寫入對於快照事務是不可見的。在單節點資料庫上，一個簡單的計數器就足以生成事務 ID。

但是當資料庫分佈在許多機器上，也許可能在多個數據中心中時，由於需要協調，（跨所有分割槽）全域性單調遞增的事務 ID 會很難生成。事務 ID 必須反映因果關係：如果事務 B 讀取由事務 A 寫入的值，則 B 必須具有比 A 更大的事務 ID，否則快照就無法保持一致。在有大量的小規模、高頻率的事務情景下，在分散式系統中建立事務 ID 成為一個難以處理的瓶頸 ^{vi}。

^{vi}. 存在分散式序列號生成器，例如 Twitter 的雪花 (Snowflake)，其以可伸縮的方式（例如，透過將 ID 空間的塊分配給不同節點）近似單調地增加唯一 ID。但是，它們通常無法保證與因果關係一致的排序，因為分配的 ID

塊的時間範圍比資料庫讀取和寫入的時間範圍要長。另請參閱“順序保證”。 ↵

我們可以使用同步時鐘的時間戳作為事務 ID 嗎？如果我們能夠獲得足夠好的同步性，那麼這種方法將具有很合適的屬性：更晚的事務會有更大的時間戳。當然，問題在於時鐘精度的不確定性。

Spanner 以這種方式實現跨資料中心的快照隔離【59, 60】。它使用 TrueTime API 報告的時鐘置信區間，並基於以下觀察結果：如果你有兩個置信區間，每個置信區間包含最早和最晚可能的時間戳 ($A = [A\{earliest\}, A\{latest\}]$, $B = [B\{earliest\}, B\{latest\}]$)，這兩個區間不重疊（即： $A\{earliest\} < A\{latest\} < B\{earliest\} < B\{latest\}$ ）的話，那麼 B 肯定發生在 A 之後——這是毫無疑問的。只有當區間重疊時，我們才不確定 A 和 B 發生的順序。

為了確保事務時間戳反映因果關係，在提交讀寫事務之前，Spanner 在提交讀寫事務時，會故意等待置信區間長度的時間。透過這樣，它可以確保任何可能讀取資料的事務處於足夠晚的時間，因此它們的置信區間不會重疊。為了保持儘可能短的等待時間，Spanner 需要保持儘可能小的時鐘不確定性，為此，Google 在每個資料中心都部署了一個 GPS 接收器或原子鐘，這允許時鐘同步到大約 7 毫秒以內【41】。

對分散式事務語義使用時鐘同步是一個活躍的研究領域【57, 61, 62】。這些想法很有趣，但是它們還沒有在谷歌之外的主流資料庫中實現。

程序暫停

讓我們考慮在分散式系統中使用危險時鐘的另一個例子。假設你有一個數據庫，每個分割槽只有一個領導者。只有領導被允許接受寫入。一個節點如何知道它仍然是領導者（它並沒有被別人宣告為死亡），並且它可以安全地接受寫入？

一種選擇是領導者從其他節點獲得一個 **租約 (lease)**，類似一個帶超時的鎖【63】。任一時刻只有一個節點可以持有租約——因此，當一個節點獲得一個租約時，它知道它在某段時間內自己是領導者，直到租約到期。為了保持領導地位，節點必須週期性地在租約過期前續期。

如果節點發生故障，就會停止續期，所以當租約過期時，另一個節點可以接管。

可以想象，請求處理迴圈看起來像這樣：

```
while (true) {
    request = getIncomingRequest();
    // 確保租約還剩下至少 10 秒
    if (lease.expiryTimeMillis - System.currentTimeMillis() < 10000){
        lease = lease.renew();
    }

    if (lease.isValid()) {
        process(request);
    }
}
```

這個程式碼有什麼問題？首先，它依賴於同步時鐘：租約到期時間由另一臺機器設定（例如，當前時間加上 30 秒，計算到期時間），並將其與本地系統時鐘進行比較。如果時鐘不同步超過幾秒，這段程式碼將開始做奇怪的事情。

其次，即使我們將協議更改為僅使用本地單調時鐘，也存在另一個問題：程式碼假定在執行剩餘時間檢查 `System.currentTimeMillis()` 和實際執行請求 `process(request)` 中間的時間間隔非常短。通常情況下，這段程式碼執行得非常快，所以 10 秒的緩衝區已經足夠確保 **租約** 在請求處理到一半時不會過期。

但是，如果程式執行中出現了意外的停頓呢？例如，想像一下，執行緒在 `lease.isValid()` 行周圍停止 15 秒，然後才繼續。在這種情況下，在請求被處理的時候，租約可能已經過期，而另一個節點已經接管了領導。然而，沒有什麼可以告訴這個執行緒已經暫停了這麼長時間了，所以這段程式碼不會注意到租約已經到期了，直到迴圈的下一個迭代——到那個時候它可能已經做了一些不安全的處理請求。

假設一個執行緒可能會暫停很長時間，這是瘋了嗎？不幸的是，這種情況發生的原因有很多種：

- 許多程式語言執行時（如 Java 虛擬機器）都有一個垃圾收集器（GC），偶爾需要停止所有正在執行的執行緒。這些“**停止所有處理 (stop-the-world)**”GC 暫停有時會持續幾分鐘【64】！甚至像 HotSpot JVM 的 CMS 這樣的所

謂的“並行”垃圾收集器也不能完全與應用程式程式碼並行執行，它需要不時地停止所有處理【65】。儘管通常可以透過改變分配模式或調整 GC 設定來減少暫停【66】，但是如果我們想要提供健壯的保證，就必須假設最壞的情況發生。

- 在虛擬化環境中，可以 **掛起 (suspend)** 虛擬機器（暫停執行所有程序並將記憶體內容儲存到磁碟）並恢復（恢復記憶體內容並繼續執行）。這個暫停可以在程序執行的任何時候發生，並且可以持續任意長的時間。這個功能有時用於虛擬機器從一個主機到另一個主機的實時遷移，而不需要重新啟動，在這種情況下，暫停的長度取決於程序寫入記憶體的速率【67】。
- 在終端使用者的裝置（如膝上型電腦）上，執行也可能被暫停並隨意恢復，例如當用戶關閉膝上型電腦的蓋子時。
- 當作業系統上下文切換到另一個執行緒時，或者當管理程式切換到另一個虛擬機器時（在虛擬機器中執行時），當前正在執行的執行緒可能在程式碼中的任意點處暫停。在虛擬機器的情況下，在其他虛擬機器中花費的 CPU 時間被稱為 **竊取時間 (steal time)**。如果機器處於沉重的負載下（即，如果等待執行的執行緒佇列很長），暫停的執行緒再次執行可能需要一些時間。
- 如果應用程式執行同步磁碟訪問，則執行緒可能暫停，等待緩慢的磁碟 I/O 操作完成【68】。在許多語言中，即使程式碼沒有包含檔案訪問，磁碟訪問也可能出乎意料地發生——例如，Java 類載入器在第一次使用時惰性載入檔案，這可能在程式執行過程中隨時發生。I/O 暫停和 GC 暫停甚至可能合謀組合它們的延遲【69】。如果磁碟實際上是一個網路檔案系統或網路塊裝置（如亞馬遜的 EBS），I/O 延遲進一步受到網路延遲變化的影響【29】。
- 如果作業系統配置為允許交換到磁碟（頁面交換），則簡單的記憶體訪問可能導致 **頁面錯誤 (page fault)**，要求將磁碟中的頁面裝入記憶體。當這個緩慢的 I/O 操作發生時，執行緒暫停。如果記憶體壓力很高，則可能需要將另一個頁面換出到磁碟。在極端情況下，作業系統可能花費大部分時間將頁面交換到記憶體中，而實際上完成的工作很少（這被稱為 **抖動**，即 *thrashing*）。為了避免這個問題，通常在伺服器機器上禁用頁面排程（如果你寧願幹掉一個程序來釋放記憶體，也不願意冒抖動風險）。
- 可以透過傳送 SIGSTOP 訊號來暫停 Unix 程序，例如透過在 shell 中按下 Ctrl-Z。這個訊號立即阻止程序繼續執行更多的 CPU 週期，直到 SIGCONT 恢復為止，此時它將繼續執行。即使你的環境通常不使用 SIGSTOP，也可能由運維工程師意外發送。

所有這些事件都可以隨時 **搶佔 (preempt)** 正在執行的執行緒，並在稍後的時間恢復執行，而執行緒甚至不會注意到這一點。這個問題類似於在單個機器上使多執行緒程式碼執行緒安全：你不能對時序做任何假設，因為隨時可能發生上下文切換，或者出現並行執行。

當在一臺機器上編寫多執行緒程式碼時，我們有相當好的工具來實現執行緒安全：互斥量、訊號量、原子計數器、無鎖資料結構、阻塞佇列等等。不幸的是，這些工具並不能直接轉化為分散式系統操作，因為分散式系統沒有共享記憶體，只有透過不可靠網路傳送的訊息。

分散式系統中的節點，必須假定其執行可能在任意時刻暫停相當長的時間，即使是在一個函式的中間。在暫停期間，世界的其它部分在繼續運轉，甚至可能因為該節點沒有響應，而宣告暫停節點的死亡。最終暫停的節點可能會繼續執行，在再次檢查自己的時鐘之前，甚至可能不會意識到自己進入了睡眠。

響應時間保證

在許多程式語言和作業系統中，執行緒和程序可能暫停一段無限制的時間，正如討論的那樣。如果你足夠努力，導致暫停的原因是 **可以消除的**。

某些軟體的執行環境要求很高，不能在特定時間內響應可能會導致嚴重的損失：控制飛機、火箭、機器人、汽車和其他物體的計算機必須對其感測器輸入做出快速而可預測的響應。在這些系統中，軟體必須有一個特定的 **截止時間 (deadline)**，如果截止時間不滿足，可能會導致整個系統的故障。這就是所謂的 **硬實時 (hard real-time)** 系統。

實時是真的嗎？

在嵌入式系統中，實時是指系統經過精心設計和測試，以滿足所有情況下的特定時間保證。這個含義與 Web 上對實時術語的模糊使用相反，後者描述了伺服器將資料推送到客戶端以及沒有嚴格的響應時間限制的流處理（見第十一章）。

例如，如果車載感測器檢測到當前正在經歷碰撞，你肯定不希望安全氣囊釋放系統因為 GC 暫停而延遲彈出。

在系統中提供 實時保證 需要各級軟體棧的支援：一個實時作業系統（RTOS），允許在指定的時間間隔內保證 CPU 時間的分配。庫函式必須申明最壞情況下的執行時間；動態記憶體分配可能受到限制或完全不允許（實時垃圾收集器存在，但是應用程式仍然必須確保它不會給 GC 太多的負擔）；必須進行大量的測試和測量，以確保達到保證。

所有這些都需要大量額外的工作，嚴重限制了可以使用的程式語言、庫和工具的範圍（因為大多數語言和工具不提供實時保證）。由於這些原因，開發實時系統非常昂貴，並且它們通常用於安全關鍵的嵌入式裝置。而且，“實時”與“高效能”不一樣——事實上，實時系統可能具有較低的吞吐量，因為他們必須讓及時響應的優先順序高於一切（另請參閱“延遲和資源利用”）。

對於大多數伺服器端資料處理系統來說，實時保證是不經濟或不合適的。因此，這些系統必須承受在非實時環境中執行的暫停和時鐘不穩定性。

限制垃圾收集的影響

程序暫停的負面影響可以在不訴諸昂貴的實時排程保證的情況下得到緩解。語言執行時在計劃垃圾回收時具有一定的靈活性，因為它們可以跟蹤物件分配的速度和隨著時間的推移剩餘的空間記憶體。

一個新興的想法是將 GC 暫停視為一個節點的短暫計劃中斷，並在這個節點收集其垃圾的同時，讓其他節點處理來自客戶端的請求。如果執行時可以警告應用程式一個節點很快需要 GC 暫停，那麼應用程式可以停止向該節點發送新的請求，等待它完成處理未完成的請求，然後在沒有請求正在進行時執行 GC。這個技巧向客戶端隱藏了 GC 暫停，並降低了響應時間的高百分比【70,71】。一些對延遲敏感的金融交易系統【72】使用這種方法。

這個想法的一個變種是隻用垃圾收集器來處理短命物件（這些物件可以快速收集），並定期在積累大量長壽物件（因此需要完整 GC）之前重新啟動程序【65,73】。一次可以重新啟動一個節點，在計劃重新啟動之前，流量可以從該節點移開，就像 [第四章](#) 裡描述的滾動升級一樣。

這些措施不能完全阻止垃圾回收暫停，但可以有效地減少它們對應用的影響。

知識、真相與謊言

本章到目前為止，我們已經探索了分散式系統與執行在單臺計算機上的程式的不同之處：沒有共享記憶體，只有透過可變延遲的不可靠網路傳遞的訊息，系統可能遭受部分失效，不可靠的時鐘和處理暫停。

如果你不習慣於分散式系統，那麼這些問題的後果就會讓人迷惑不解。網路中的一個節點無法確切地知道任何事情——它只能根據它透過網路接收到（或沒有接收到）的訊息進行猜測。節點只能透過交換訊息來找出另一個節點所處的狀態（儲存了哪些資料，是否正確執行等等）。如果遠端節點沒有響應，則無法知道它處於什麼狀態，因為網路中的問題不能可靠地與節點上的問題區分開來。

這些系統的討論與哲學有關：在系統中什麼是真什麼是假？如果感知和測量的機制都是不可靠的，那麼關於這些知識我們又能多麼確定呢？軟體系統應該遵循我們對物理世界所期望的法則，如因果關係嗎？

幸運的是，我們不需要去搞清楚生命的意義。在分散式系統中，我們可以陳述關於行為（系統模型）的假設，並以滿足這些假設的方式設計實際系統。演算法可以被證明在某個系統模型中正確執行。這意味著即使底層系統模型提供了很少的保證，也可以實現可靠的行為。

但是，儘管可以使軟體在不可靠的系統模型中表現良好，但這並不是可以直截了當實現的。在本章的其餘部分中，我們將進一步探討分散式系統中的知識和真相的概念，這將有助於我們思考我們可以做出的各種假設以及我們可能希望提供的保證。在 [第九章](#) 中，我們將著眼於分散式系統的一些例子，這些演算法在特定的假設條件下提供了特定的保證。

真相由多數所定義

設想一個具有不對稱故障的網路：一個節點能夠接收發送給它的所有訊息，但是來自該節點的任何傳出訊息被丟棄或延遲【19】。即使該節點執行良好，並且正在接收來自其他節點的請求，其他節點也無法聽到其響應。經過一段時間後，其他節點宣佈它已經死亡，因為他們沒有聽到節點的訊息。這種情況就像夢魘一樣：**半斷開（semi-disconnected）**

的節點被拖向墓地，敲打尖叫道“我沒死！”——但是由於沒有人能聽到它的尖叫，葬禮隊伍繼續以堅忍的決心繼續行進。

在一個稍微不那麼夢魘的場景中，半斷開的節點可能會注意到它傳送的訊息沒有被其他節點確認，因此意識到網路中必定存在故障。儘管如此，節點被其他節點錯誤地宣告為死亡，而半連線的節點對此無能為力。

第三種情況，想像一個正在經歷長時間 **垃圾收集暫停 (stop-the-world GC Pause)** 的節點，節點的所有執行緒被 GC 搶佔並暫停一分鐘，因此沒有請求被處理，也沒有響應被傳送。其他節點等待，重試，不耐煩，並最終宣佈節點死亡，並將其丟到靈車上。最後，GC 完成，節點的執行緒繼續，好像什麼也沒有發生。其他節點感到驚訝，因為所謂的死亡節點突然從棺材中抬起頭來，身體健康，開始和旁觀者高興地聊天。GC 後的節點最初甚至沒有意識到已經經過了整整一分鐘，而且自己已被宣告死亡。從它自己的角度來看，從最後一次與其他節點交談以來，幾乎沒有經過任何時間。

這些故事的寓意是，節點不一定能相信自己對於情況的判斷。分散式系統不能完全依賴單個節點，因為節點可能隨時失效，可能會使系統卡死，無法恢復。相反，許多分散式演算法都依賴於法定人數，即在節點之間進行投票（請參閱“[讀寫的法定人數](#)”）：決策需要來自多個節點的最小投票數，以減少對於某個特定節點的依賴。

這也包括關於宣告節點死亡的決定。如果法定數量的節點宣告另一個節點已經死亡，那麼即使該節點仍感覺自己活著，它也必須被認為是死的。個體節點必須遵守法定決定並下臺。

最常見的法定人數是超過一半的絕對多數（儘管其他型別的法定人數也是可能的）。多數法定人數允許系統繼續工作，如果單個節點發生故障（三個節點可以容忍單節點故障；五個節點可以容忍雙節點故障）。系統仍然是安全的，因為在這個制度中只能有一個多數——不能同時存在兩個相互衝突的多數決定。當我們在 [第九章](#) 中討論 **共識演算法 (consensus algorithms)** 時，我們將更詳細地討論法定人數的應用。

領導者和鎖

通常情況下，一些東西在一個系統中只能有一個。例如：

- 資料庫分割槽的領導者只能有一個節點，以避免 **腦裂**（即 split brain，請參閱“[處理節點宕機](#)”）。
- 特定資源的鎖或物件只允許一個事務 / 客戶端持有，以防同時寫入和損壞。
- 一個特定的使用者名稱只能被一個使用者所註冊，因為使用者名稱必須唯一標識一個使用者。

在分散式系統中實現這一點需要注意：即使一個節點認為它是“**天選者 (the chosen one)**”（分割槽的負責人，鎖的持有者，成功獲取使用者名稱的使用者的請求處理程式），但這並不意味著有法定人數的節點同意！一個節點可能以前是領導者，但是如果其他節點在此期間宣佈它死亡（例如，由於網路中斷或 GC 暫停），則它可能已被降級，且另一個領導者可能已經當選。

如果一個節點繼續表現為 **天選者**，即使大多數節點已經宣告它已經死了，則在考慮不周的系統中可能會導致問題。這樣的節點能以自己賦予的權能向其他節點發送訊息，如果其他節點相信，整個系統可能會做一些不正確的事情。

例如，[圖 8-4](#) 顯示了由於不正確的鎖實現導致的資料損壞錯誤。（這個錯誤不僅僅是理論上的：HBase 曾經有這個問題 [【74,75】](#)）假設你要確保一個儲存服務中的檔案一次只能被一個客戶訪問，因為如果多個客戶試圖對此寫入，該檔案將被損壞。你嘗試透過在訪問檔案之前要求客戶端從鎖定服務獲取租約來實現此目的。

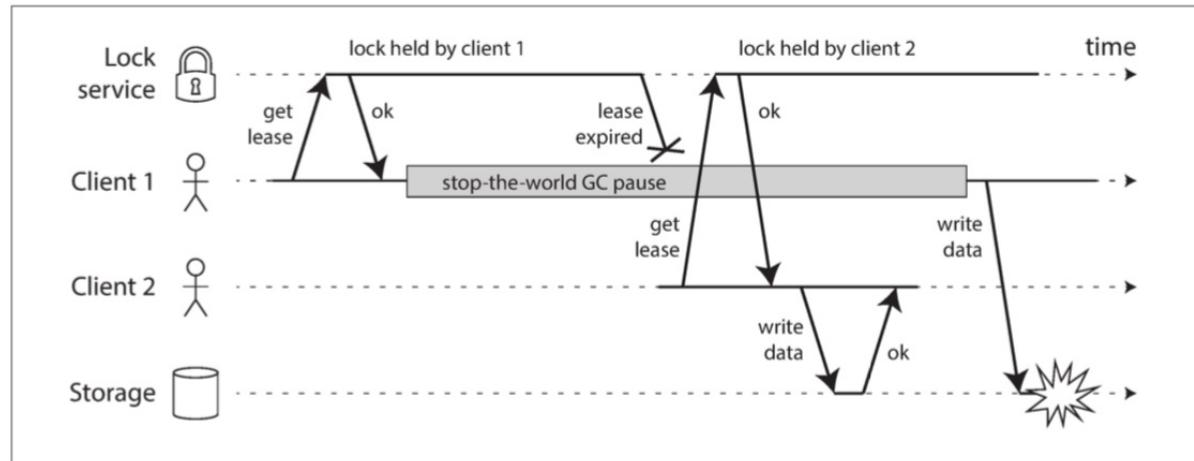


圖 8-4 分散式鎖的實現不正確：客戶端 1 認為它仍然具有有效的租約，即使它已經過期，從而破壞了儲存中的檔案

這個問題就是我們先前在“[程序暫停](#)”中討論過的一個例子：如果持有租約的客戶端暫停太久，它的租約將到期。另一個客戶端可以獲得同一檔案的租約，並開始寫入檔案。當暫停的客戶端回來時，它認為（不正確）它仍然有一個有效的租約，並繼續寫入檔案。結果，客戶的寫入將產生衝突並損壞檔案。

防護令牌

當使用鎖或租約來保護對某些資源（如 圖 8-4 中的檔案儲存）的訪問時，需要確保一個被誤認為自己是“天選者”的節點不能擾亂系統的其它部分。實現這一目標的一個相當簡單的技術就是 **防護**（fencing），如 圖 8-5 所示

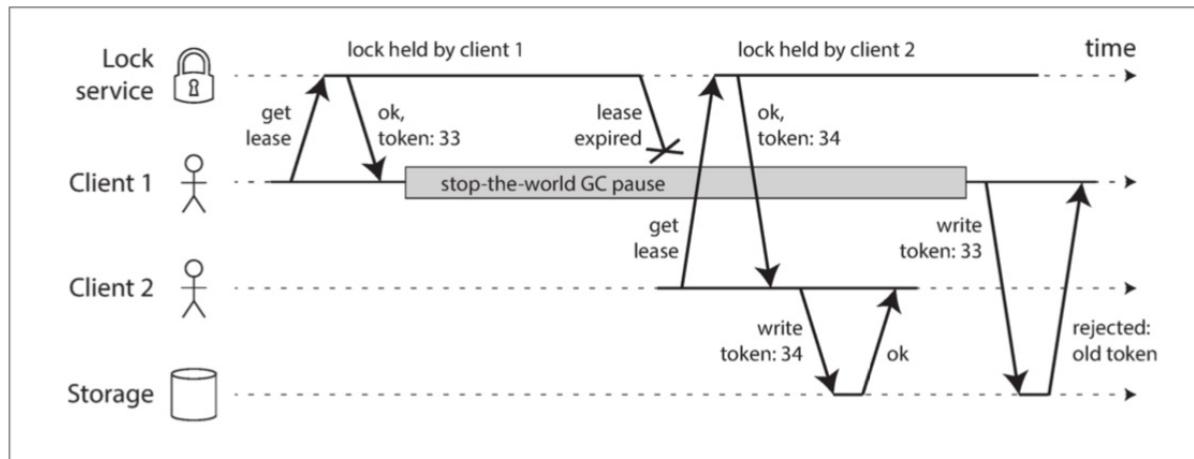


圖 8-5 只允許以增加防護令牌的順序進行寫操作，從而保證儲存安全

我們假設每次鎖定伺服器授予鎖或租約時，它還會返回一個 **防護令牌**（fencing token），這個數字在每次授予鎖定時都會增加（例如，由鎖定服務增加）。然後，我們可以要求客戶端每次向儲存服務傳送寫入請求時，都必須包含當前的防護令牌。

在 圖 8-5 中，客戶端 1 以 33 的令牌獲得租約，但隨後進入一個長時間的停頓並且租約到期。客戶端 2 以 34 的令牌（該數字總是增加）獲取租約，然後將其寫入請求傳送到儲存服務，包括 34 的令牌。稍後，客戶端 1 恢復生機並將其寫入儲存服務，包括其令牌值 33。但是，儲存伺服器會記住它已經處理了一個具有更高令牌編號（34）的寫入，因此它會拒絕帶有令牌 33 的請求。

如果將 ZooKeeper 用作鎖定服務，則可將事務標識 `zxid` 或節點版本 `cversion` 用作防護令牌。由於它們保證單調遞增，因此它們具有所需的屬性【74】。

請注意，這種機制要求資源本身在檢查令牌方面發揮積極作用，透過拒絕使用舊的令牌，而不是已經被處理的令牌來進行寫操作——僅僅依靠客戶端檢查自己的鎖狀態是不夠的。對於不明確支援防護令牌的資源，可能仍然可以解決此限制（例如，在檔案儲存服務的情況下，可以將防護令牌包含在檔名中）。但是，為了避免在鎖的保護之外處理請求，需要進行某種檢查。

在伺服器端檢查一個令牌可能看起來像是一個缺點，但這可以說是一件好事：一個服務假定它的客戶總是守規矩並不明智，因為使用客戶端的人與執行服務的人優先順序非常不一樣【76】。因此，任何服務保護自己免受意外客戶的濫用是一個好主意。

拜占庭故障

防護令牌可以檢測和阻止無意中發生錯誤的節點（例如，因為它尚未發現其租約已過期）。但是，如果節點有意破壞系統的保證，則可以透過使用假防護令牌傳送訊息來輕鬆完成此操作。

在本書中，我們假設節點是不可靠但誠實的：它們可能很慢或者從不響應（由於故障），並且它們的狀態可能已經過時（由於 GC 暫停或網路延遲），但是我們假設如果節點它做出了回應，它正在說出“真相”：盡其所知，它正在按照協議的規則扮演其角色。

如果存在節點可能“撒謊”（傳送任意錯誤或損壞的響應）的風險，則分散式系統的問題變得更困難了——例如，如果節點可能聲稱其實際上沒有收到特定的訊息。這種行為被稱為**拜占庭故障**（Byzantine fault），在不信任的環境中達成共識的問題被稱為**拜占庭將軍問題**【77】。

拜占庭將軍問題

拜占庭將軍問題是對所謂“兩將軍問題”的泛化【78】，它想象兩個將軍需要就戰鬥計劃達成一致的情況。由於他們在兩個不同的地點建立了營地，他們只能透過信使進行溝通，信使有時會被延遲或丟失（就像網路中的資訊包一樣）。我們將在[第九章](#)討論這個共識問題。

在這個問題的拜占庭版本里，有 n 位將軍需要同意，他們的努力因為有一些叛徒在他們中間而受到阻礙。大多數的將軍都是忠誠的，因而發出了真實的資訊，但是叛徒可能會試圖透過傳送虛假或不真實的資訊來欺騙和混淆他人（在試圖保持未被發現的同時）。事先並不知道叛徒是誰。

拜占庭是後來成為君士坦丁堡的古希臘城市，現在在土耳其的伊斯坦布林。沒有任何歷史證據表明拜占庭將軍比其他地方更容易出現詭計和陰謀。相反，這個名字來源於拜占庭式的過度複雜，官僚，迂迴等意義，早在計算機之前就已經在政治中被使用了【79】。Lamport 想要選一個不會冒犯任何讀者的國家，他被告知將其稱為阿爾巴尼亞將軍問題並不是一個好主意【80】。

當一個系統在部分節點發生故障、不遵守協議、甚至惡意攻擊、擾亂網路時仍然能繼續正確工作，稱之為**拜占庭容錯**（Byzantine fault-tolerant）的，這種擔憂在某些特定情況下是有意義的：

- 在航空航天環境中，計算機記憶體或 CPU 暫存器中的資料可能被輻射破壞，導致其以任意不可預知的方式響應其他節點。由於系統故障非常昂貴（例如，飛機撞毀和炸死船上所有人員，或火箭與國際空間站相撞），飛行控制系統必須容忍拜占庭故障【81,82】。
- 在多個參與組織的系統中，一些參與者可能會試圖欺騙或詐騙他人。在這種情況下，節點僅僅信任另一個節點的訊息是不安全的，因為它們可能是出於惡意的目的而被傳送的。例如，像比特幣和其他區塊鏈一樣的對等網路可以被認為是讓互不信任的各方同意交易是否發生的一種方式，而不依賴於中心機構（central authority）【83】。

然而，在本書討論的那些系統中，我們通常可以安全地假設沒有拜占庭式的錯誤。在你的資料中心裡，所有的節點都是由你的組織控制的（所以他們可以信任），輻射水平足夠低，記憶體損壞不是一個大問題。製作拜占庭容錯系統的協議相當複雜【84】，而容錯嵌入式系統依賴於硬體層面的支援【81】。在大多數伺服器端資料系統中，部署拜占庭容錯解決方案的成本使其變得不切實際。

Web 應用程式確實需要預期受終端使用者控制的客戶端（如 Web 瀏覽器）的任意和惡意行為。這就是為什麼輸入驗證，資料清洗和輸出轉義如此重要：例如，防止 SQL 注入和跨站點指令碼。然而，我們通常不在這裡使用拜占庭容錯協議，而只是讓伺服器有權決定是否允許客戶端行為。但在沒有這種中心機構的對等網路中，拜占庭容錯更為重要。

軟體中的一個錯誤（bug）可能被認為是拜占庭式的錯誤，但是如果你將相同的軟體部署到所有節點上，那麼拜占庭式的容錯演算法幫不到你。大多數拜占庭式容錯演算法要求超過三分之二的節點能夠正常工作（即，如果有四個節點，最多隻能有一個故障）。要使用這種方法對付 bug，你必須有四個獨立的相同軟體的實現，並希望一個 bug 只出現在四個實現之一中。

同樣，如果一個協議可以保護我們免受漏洞，安全滲透和惡意攻擊，那麼這將是有吸引力的。不幸的是，這也是不現實的：在大多數系統中，如果攻擊者可以滲透一個節點，那他們可能會滲透所有這些節點，因為它們可能都執行著相同的軟體。因此，傳統機制（認證，訪問控制，加密，防火牆等）仍然是抵禦攻擊者的主要保護措施。

弱謊言形式

儘管我們假設節點通常是誠實的，但值得向軟體中新增防止“撒謊”弱形式的機制——例如，由硬體問題導致的無效訊息，軟體錯誤和錯誤配置。這種保護機制並不是完全的拜占庭容錯，因為它們不能抵擋決心堅定的對手，但它們仍然是簡單而實用的步驟，以提高可靠性。例如：

- 由於硬體問題或作業系統、驅動程式、路由器等中的錯誤，網路資料包有時會受到損壞。通常，損壞的資料包會被內建於 TCP 和 UDP 中的校驗和所俘獲，但有時它們也會逃脫檢測【85,86,87】。要對付這種破壞通常使用簡單的方法就可以做到，例如應用程式級協議中的校驗和。
- 可公開訪問的應用程式必須仔細清理來自使用者的任何輸入，例如檢查值是否在合理的範圍內，並限制字串的大小以防止透過大記憶體分配的拒絕服務。防火牆後面的內部服務對於輸入也許可以只採取一些不那麼嚴格的檢查，但是採取一些基本的合理性檢查（例如，在協議解析中）仍然是一個好主意。
- NTP 客戶端可以配置多個伺服器地址。同步時，客戶端聯絡所有的伺服器，估計它們的誤差，並檢查大多數伺服器是否對某個時間範圍達成一致。只要大多數的伺服器沒問題，一個配置錯誤的 NTP 伺服器報告的時間會被當成特異值從同步中排除【37】。使用多個伺服器使 NTP 更健壯（比起只用單個伺服器來）。

系統模型與現實

已經有很多演算法被設計以解決分散式系統問題——例如，我們將在 [第九章](#) 討論共識問題的解決方案。為了有用，這些演算法需要容忍我們在本章中討論的分散式系統的各種故障。

演算法的編寫方式不應該過分依賴於執行的硬體和軟體配置的細節。這就要求我們以某種方式將我們期望在系統中發生的錯誤形式化。我們透過定義一個系統模型來做到這一點，這個模型是一個抽象，描述一個演算法可以假設的事情。

關於時序假設，三種系統模型是常用的：

- 同步模型

同步模型（synchronous model） 假設網路延遲、程序暫停和時鐘誤差都是受限的。這並不意味著完全同步的時鐘或零網路延遲；這隻意味著你知道網路延遲、暫停和時鐘漂移將永遠不會超過某個固定的上限【88】。同步模型並不是大多數實際系統的現實模型，因為（如本章所討論的）無限延遲和暫停確實會發生。

- 部分同步模型

部分同步（partial synchronous） 意味著一個系統在大多數情況下像一個同步系統一樣執行，但有時候會超出網路延遲，程序暫停和時鐘漂移的界限【88】。這是很多系統的現實模型：大多數情況下，網路和程序表現良好，否則我們永遠無法完成任何事情，但是我們必須承認，在任何時刻都存在時序假設偶然被破壞的事實。發生這種情況時，網路延遲、暫停和時鐘錯誤可能會變得相當大。

- 非同步模型

在這個模型中，一個演算法不允許對時序做任何假設——事實上它甚至沒有時鐘（所以它不能使用超時）。一些演算法被設計為可用於非同步模型，但非常受限。

進一步來說，除了時序問題，我們還要考慮 **節點失效**。三種最常見的節點系統模型是：

- 崩潰 - 停止故障

在 **崩潰停止 (crash-stop)** 模型中，演算法可能會假設一個節點只能以一種方式失效，即透過崩潰。這意味著節點可能在任意時刻突然停止響應，此後該節點永遠消失——它永遠不會回來。

- **崩潰 - 恢復故障**

我們假設節點可能會在任何時候崩潰，但也許會在未知的時間之後再次開始響應。在 **崩潰 - 恢復 (crash-recovery)** 模型中，假設節點具有穩定的儲存（即，非易失性磁碟儲存）且會在崩潰中保留，而記憶體中的狀態會丟失。

- **拜占庭 (任意) 故障**

節點可以做（絕對意義上的）任何事情，包括試圖戲弄和欺騙其他節點，如上一節所述。

對於真實系統的建模，具有 **崩潰 - 恢復故障 (crash-recovery)** 的 **部分同步模型 (partial synchronous)** 通常是最有用的模型。分散式演算法如何應對這種模型？

演算法的正確性

為了定義演算法是正確的，我們可以描述它的屬性。例如，排序演算法的輸出具有如下特性：對於輸出列表中的任何兩個不同的元素，左邊的元素比右邊的元素小。這只是定義對列表進行排序含義的一種形式方式。

同樣，我們可以寫下我們想要的分散式演算法的屬性來定義它的正確含義。例如，如果我們正在為一個鎖生成防護令牌（請參閱“[防護令牌](#)”），我們可能要求演算法具有以下屬性：

- **唯一性 (uniqueness)**

沒有兩個防護令牌請求返回相同的值。

- **單調序列 (monotonic sequence)**

如果請求 \$x\$ 返回了令牌 \$t_x\$，並且請求 \$y\$ 返回了令牌 \$t_y\$，並且 \$x\$ 在 \$y\$ 開始之前已經完成，那麼 \$t_x < t_y\$。

- **可用性 (availability)**

請求防護令牌並且不會崩潰的節點，最終會收到響應。

如果一個系統模型中的演算法總是滿足它在所有我們假設可能發生的情況下的性質，那麼這個演算法是正確的。但這如何有意義？如果所有的節點崩潰，或者所有的網路延遲突然變得無限長，那麼沒有任何演算法能夠完成任何事情。

安全性和活性

為了澄清這種情況，有必要區分兩種不同的屬性：**安全 (safety)** 屬性和**活性 (liveness)** 屬性。在剛剛給出的例子中，**唯一性** 和 **單調序列** 是安全屬性，而 **可用性** 是活性屬性。

這兩種性質有什麼區別？一個試金石就是，活性屬性通常在定義中通常包括“最終”一詞（是的，你猜對了——最終一致性是一個活性屬性【89】）。

安全通常被非正式地定義為：**沒有壞事發生**，而活性通常就類似：**最終好事發生**。但是，最好不要過多地閱讀那些非正式的定義，因為好與壞的含義是主觀的。安全和活性的實際定義是精確的和數學的【90】：

- 如果安全屬性被違反，我們可以指向一個特定的安全屬性被破壞的時間點（例如，如果違反了唯一性屬性，我們可以確定重複的防護令牌被返回的特定操作）。違反安全屬性後，違規行為不能被撤銷——損失已經發生。
- 活性屬性反過來：在某個時間點（例如，一個節點可能傳送了一個請求，但還沒有收到響應），它可能不成立，但總是希望在未來能成立（即透過接受答覆）。

區分安全屬性和活性屬性的一個優點是可以幫助我們處理困難的系統模型。對於分散式演算法，在系統模型的所有可能情況下，要求**始終保持安全屬性**是常見的【88】。也就是說，即使所有節點崩潰，或者整個網路出現故障，演算法仍然必須確保它不會返回錯誤的結果（即保證安全屬性得到滿足）。

但是，對於活性屬性，我們可以提出一些注意事項：例如，只有在大多數節點沒有崩潰的情況下，只有當網路最終從中斷中恢復時，我們才可以說請求需要接收響應。部分同步模型的定義要求系統最終返回到同步狀態——即任何網路中斷的時間段只會持續一段有限的時間，然後進行修復。

將系統模型對映到現實世界

安全屬性和活性屬性以及系統模型對於推理分散式演算法的正確性非常有用。然而，在實踐中實施演算法時，現實的混亂事實再一次地讓你咬牙切齒，很明顯系統模型是對現實的簡化抽象。

例如，在崩潰 - 恢復（crash-recovery）模型中的演算法通常假設穩定儲存器中的資料在崩潰後可以倖存。但是，如果磁碟上的資料被破壞，或者由於硬體錯誤或錯誤配置導致資料被清除，會發生什麼情況【91】？如果伺服器存在韌體錯誤並且在重新啟動時無法識別其硬碟驅動器，即使驅動器已正確連線到伺服器，那又會發生什麼情況【92】？

法定人數演算法（請參閱“[讀寫的法定人數](#)”）依賴節點來記住它聲稱儲存的資料。如果一個節點可能患有健忘症，忘記了以前儲存的資料，這會打破法定條件，從而破壞演算法的正確性。也許需要一個新的系統模型，在這個模型中，我們假設穩定的儲存大多能在崩潰後倖存，但有時也可能會丟失。但是那個模型就變得更難以推論了。

演算法的理論描述可以簡單宣稱一些事是不會發生的——在非拜占庭式系統中，我們確實需要對可能發生和不可能發生的故障做出假設。然而，真實世界的實現，仍然會包括處理“假設上不可能”情況的程式碼，即使程式碼可能就是 `printf("Sucks to be you")` 和 `exit(666)`，實際上也就是留給運維來擦屁股【93】。（這可以說是計算機科學和軟體工程間的一個差異）。

這並不是說理論上抽象的系統模型是毫無價值的，恰恰相反。它們對於將實際系統的複雜性提取成一個個我們可以推理的可處理的錯誤型別是非常有幫助的，以便我們能夠理解這個問題，並試圖系統地解決這個問題。我們可以證明演算法是正確的，透過表明它們的屬性在某個系統模型中總是成立的。

證明演算法正確並不意味著它在真實系統上的實現必然總是正確的。但這邁出了很好的第一步，因為理論分析可以發現演算法中的問題，這種問題可能會在現實系統中長期潛伏，直到你的假設（例如，時序）因為不尋常的情況被打破。理論分析與經驗測試同樣重要。

本章小結

在本章中，我們討論了分散式系統中可能發生的各種問題，包括：

- 當你嘗試透過網路傳送資料包時，資料包可能會丟失或任意延遲。同樣，答覆可能會丟失或延遲，所以如果你沒有得到答覆，你不知道訊息是否傳送成功了。
- 節點的時鐘可能會與其他節點顯著不同步（儘管你盡最大努力設定 NTP），它可能會突然跳轉或跳回，依靠它是很危險的，因為你很可能沒有好的方法來測量你的時鐘的錯誤間隔。
- 一個程序可能會在其執行的任何時候暫停一段相當長的時間（可能是因為停止所有處理的垃圾收集器），被其他節點宣告死亡，然後再次復活，卻沒有意識到它被暫停了。

這類 **部分失效**（partial failure）可能發生的事實是分散式系統的決定性特徵。每當軟體試圖做任何涉及其他節點的事情時，偶爾就有可能會失敗，或者隨機變慢，或者根本沒有響應（最終超時）。在分散式系統中，我們試圖在軟體中建立**部分失效**的容錯機制，這樣整個系統在即使某些組成部分被破壞的情況下，也可以繼續執行。

為了容忍錯誤，第一步是**檢測**它們，但即使這樣也很難。大多數系統沒有檢測節點是否發生故障的準確機制，所以大多數分散式演算法依靠**超時**來確定遠端節點是否仍然可用。但是，超時無法區分網路失效和節點失效，並且可變的網路延遲有時會導致節點被錯誤地懷疑發生故障。此外，有時一個節點可能處於降級狀態：例如，由於驅動程式錯誤，千兆網絡卡可能突然下降到 1 Kb/s 的吞吐量【94】。這樣一個“跛行”而不是死掉的節點可能比一個乾淨的失效節點更難處理。

一旦檢測到故障，使系統容忍它也並不容易：沒有全域性變數，沒有共享記憶體，沒有共同的知識，或機器之間任何其他種類的共享狀態。節點甚至不能就現在是什麼時間達成一致，就不用說更深奧的了。資訊從一個節點流向另一個節點的唯一方法是透過不可靠的網路傳送資訊。重大決策不能由一個節點安全地完成，因此我們需要一個能從其他節點獲得幫助的協議，並爭取達到法定人數以達成一致。

如果你習慣於在理想化的數學完美的單機環境（同一個操作總能確定地返回相同的結果）中編寫軟體，那麼轉向分散式系統的凌亂的物理現實可能會有些令人震驚。相反，如果能夠在單臺計算機上解決一個問題，那麼分散式系統工程師通常會認為這個問題是平凡的【5】，現在單個計算機確實可以做很多事情【95】。如果你可以避免開啟潘多拉的盒子，把東西放在一臺機器上，那麼通常是值得的。

但是，正如在 [第二部分](#) 的介紹中所討論的那樣，可伸縮性並不是使用分散式系統的唯一原因。容錯和低延遲（透過將資料放置在距離使用者較近的地方）是同等重要的目標，而這些不能用單個節點實現。

在本章中，我們也轉換了幾次話題，探討了網路、時鐘和程序的不可靠性是否是不可避免的自然規律。我們看到這並不是：有可能給網路提供硬實時的響應保證和有限的延遲，但是這樣做非常昂貴，且導致硬體資源的利用率降低。大多數非安全關鍵系統會選擇 **便宜而不可靠**，而不是 **昂貴和可靠**。

我們還談到了超級計算機，它們採用可靠的元件，因此當元件發生故障時必須完全停止並重新啟動。相比之下，分散式系統可以永久執行而不會在服務層面中斷，因為所有的錯誤和維護都可以在節點級別進行處理——至少在理論上是如此。（實際上，如果一個錯誤的配置變更被應用到所有的節點，仍然會使分散式系統癱瘓）。

本章一直在講存在的問題，給我們展現了一幅黯淡的前景。在 [下一章](#) 中，我們將繼續討論解決方案，並討論一些旨在解決分散式系統中所有問題的演算法。

參考文獻

1. Mark Savage: Just No Getting Around It: You're Building a Distributed System] (<http://queue.acm.org/detail.cfm?id=2482856>), "ACM Queue, volume 11, number 4, pages 80-89, April 2013. doi:[10.1145/2466486.2482856](https://doi.org/10.1145/2466486.2482856)
2. Jay Kreps: "Getting Real About Distributed System Reliability," blog.empathybox.com, March 19, 2012.
3. Sydney Padua: *The Thrilling Adventures of Lovelace and Babbage: The (Mostly) True Story of the First Computer*. Particular Books, April ISBN: 978-0-141-98151-2
4. Coda Hale: "You Can't Sacrifice Partition Tolerance," codahale.com, October 7, 2010.
5. Jeff Hodges: "Notes on Distributed Systems for Young Bloods," somethingsimilar.com, January 14, 2013.
6. Antonio Regalado: "Who Coined 'Cloud Computing'?", technologyreview.com, October 31, 2011.
7. Luiz André Barroso, Jimmy Clidaras, and Urs Hözle: "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition," *Synthesis Lectures on Computer Architecture*, volume 8, number 3, Morgan & Claypool Publishers, July 2013. doi:[10.2200/S00516ED2V01Y201306CAC024](https://doi.org/10.2200/S00516ED2V01Y201306CAC024), ISBN: 978-1-627-05010-4
8. David Fiala, Frank Mueller, Christian Engelmann, et al.: "Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing," at *International Conference for High Performance Computing, Networking, Storage and Analysis* (SC12), November 2012.
9. Arjun Singh, Joon Ong, Amit Agarwal, et al.: "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," at *Annual Conference of the ACM Special Interest Group on Data Communication* (SIGCOMM), August 2015. doi:[10.1145/2785956.2787508](https://doi.org/10.1145/2785956.2787508)
10. Glenn K. Lockwood: "Hadoop's Uncomfortable Fit in HPC," glennlockwood.blogspot.co.uk, May 16, 2014.
11. John von Neumann: "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," in *Automata Studies (AM-34)*, edited by Claude E. Shannon and John McCarthy, Princeton University Press, 1956. ISBN: 978-0-691-07916-5
12. Richard W. Hamming: *The Art of Doing Science and Engineering*. Taylor & Francis, 1997. ISBN: 978-9-056-99500-3
13. Claude E. Shannon: "A Mathematical Theory of Communication," *The Bell System Technical Journal*, volume 27, number 3, pages 379–423 and 623–656, July 1948.
14. Peter Bailis and Kyle Kingsbury: "The Network Is Reliable," *ACM Queue*, volume 12, number 7, pages 48-55, July 2014. doi:[10.1145/2639988.2639988](https://doi.org/10.1145/2639988.2639988)
15. Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish: "Taming Uncertainty in Distributed Systems with Help from the Network," at *10th European Conference on Computer Systems* (EuroSys), April 2015. doi:[10.1145/2741948.2741976](https://doi.org/10.1145/2741948.2741976)

16. Phillipa Gill, Navendu Jain, and Nachiappan Nagappan: “[Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications](#),” at *ACM SIGCOMM Conference*, August 2011.
[doi:10.1145/2018436.2018477](https://doi.org/10.1145/2018436.2018477)
17. Mark Imbriaco: “[Downtime Last Saturday](#),” *github.com*, December 26, 2012.
18. Will Oremus: “[The Global Internet Is Being Attacked by Sharks, Google Confirms](#),” *slate.com*, August 15, 2014.
19. Marc A. Donges: “[Re: bnx2 cards Intermittantly Going Offline](#),” Message to Linux *netdev* mailing list, *spinics.net*, September 13, 2012.
20. Kyle Kingsbury: “[Call Me Maybe: Elasticsearch](#),” *aphyr.com*, June 15, 2014.
21. Salvatore Sanfilippo: “[A Few Arguments About Redis Sentinel Properties and Fail Scenarios](#),” *antirez.com*, October 21, 2014.
22. Bert Hubert: “[The Ultimate SO_LINGER Page, or: Why Is My TCP Not Reliable](#),” *blog.netherlabs.nl*, January 18, 2009.
23. Nicolas Liochon: “[CAP: If All You Have Is a Timeout, Everything Looks Like a Partition](#),” *blog.thislongrun.com*, May 25, 2015.
24. Jerome H. Saltzer, David P. Reed, and David D. Clark: “[End-To-End Arguments in System Design](#),” *ACM Transactions on Computer Systems*, volume 2, number 4, pages 277–288, November 1984.
[doi:10.1145/357401.357402](https://doi.org/10.1145/357401.357402)
25. Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, et al.: “[Queues Don't Matter When You Can JUMP Them!](#),” at *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.
26. Guohui Wang and T. S. Eugene Ng: “[The Impact of Virtualization on Network Performance of Amazon EC2 Data Center](#),” at *29th IEEE International Conference on Computer Communications* (INFOCOM), March 2010.
[doi:10.1109/INFCOM.2010.5461931](https://doi.org/10.1109/INFCOM.2010.5461931)
27. Van Jacobson: “[Congestion Avoidance and Control](#),” at *ACM Symposium on Communications Architectures and Protocols* (SIGCOMM), August 1988. [doi:10.1145/52324.52356](https://doi.org/10.1145/52324.52356)
28. Brandon Philips: “[etcd: Distributed Locking and Service Discovery](#),” at *Strange Loop*, September 2014.
29. Steve Newman: “[A Systematic Look at EC2 I/O](#),” *blog.scalyr.com*, October 16, 2012.
30. Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama: “[The φ Accrual Failure Detector](#),” Japan Advanced Institute of Science and Technology, School of Information Science, Technical Report IS-RR-2004-010, May 2004.
31. Jeffrey Wang: “[Phi Accrual Failure Detector](#),” *ternarysearch.blogspot.co.uk*, August 11, 2013.
32. Srinivasan Keshav: *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*. Addison-Wesley Professional, May 1997. ISBN: 978-0-201-63442-6
33. Cisco, “[Integrated Services Digital Network](#),” *docwiki.cisco.com*.
34. Othmar Kyas: *ATM Networks*. International Thomson Publishing, 1995. ISBN: 978-1-850-32128-6
35. “[InfiniBand FAQ](#),” Mellanox Technologies, December 22, 2014.
36. Jose Renato Santos, Yoshio Turner, and G. (John) Janakiraman: “[End-to-End Congestion Control for InfiniBand](#),” at *22nd Annual Joint Conference of the IEEE Computer and Communications Societies* (INFOCOM), April 2003. Also published by HP Laboratories Palo Alto, Tech Report HPL-2002-359. [doi:10.1109/INFCOM.2003.1208949](https://doi.org/10.1109/INFCOM.2003.1208949)
37. Ulrich Windl, David Dalton, Marc Martinec, and Dale R. Worley: “[The NTP FAQ and HOWTO](#),” *ntp.org*, November 2006.
38. John Graham-Cumming: “[How and why the leap second affected Cloudflare DNS](#),” *blog.cloudflare.com*, January 1, 2017.
39. David Holmes: “[Inside the Hotspot VM: Clocks, Timers and Scheduling Events – Part I – Windows](#),” *blogs.oracle.com*, October 2, 2006.
40. Steve Loughran: “[Time on Multi-Core, Multi-Socket Servers](#),” *steveloughran.blogspot.co.uk*, September 17, 2015.
41. James C. Corbett, Jeffrey Dean, Michael Epstein, et al.: “[Spanner: Google's Globally-Distributed Database](#),” at *10th USENIX Symposium on Operating System Design and Implementation* (OSDI), October 2012.
42. M. Caporaso and R. Ambrosini: “[How Closely Can a Personal Computer Clock Track the UTC Timescale Via the Internet?](#),” *European Journal of Physics*, volume 23, number 4, pages L17–L21, June 2012. [doi:10.1088/0143-0807/23/4/103](https://doi.org/10.1088/0143-0807/23/4/103)
43. Nelson Minar: “[A Survey of the NTP Network](#),” *alumni.media.mit.edu*, December 1999.

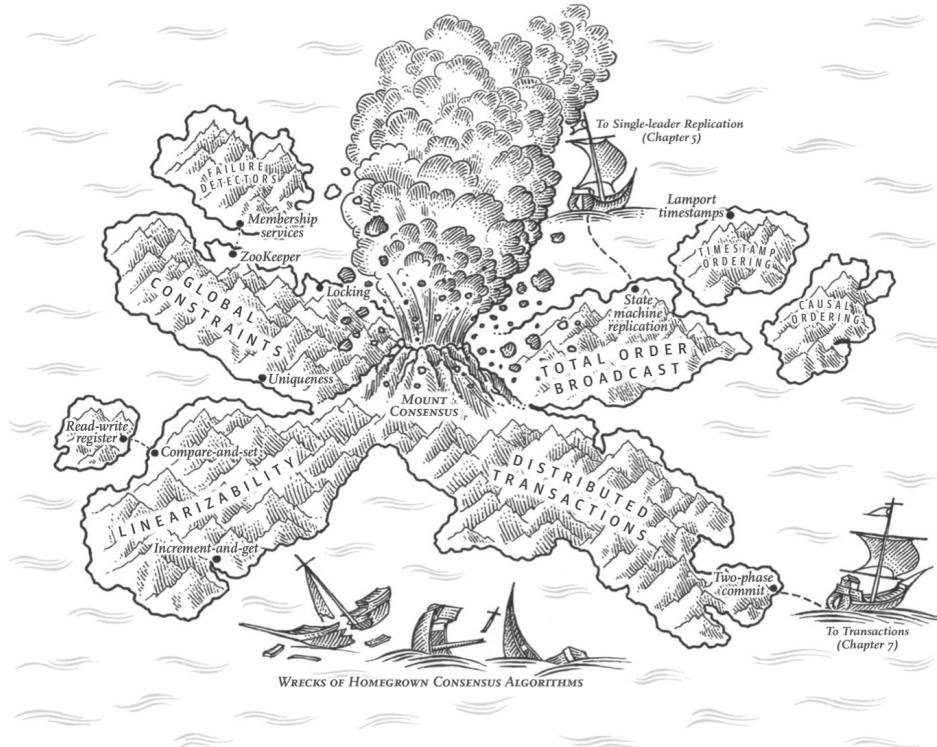
44. Viliam Holub: “[Synchronizing Clocks in a Cassandra Cluster Pt. 1 – The Problem](#),” blog.logentries.com, March 14, 2014.
45. Poul-Henning Kamp: “[The One-Second War \(What Time Will You Die?\)](#),” *ACM Queue*, volume 9, number 4, pages 44–48, April 2011. doi:[10.1145/1966989.1967009](https://doi.org/10.1145/1966989.1967009)
46. Nelson Minar: “[Leap Second Crashes Half the Internet](#),” somebits.com, July 3, 2012.
47. Christopher Pascoe: “[Time, Technology and Leaping Seconds](#),” googleblog.blogspot.co.uk, September 15, 2011.
48. Mingxue Zhao and Jeff Barr: “[Look Before You Leap – The Coming Leap Second and AWS](#),” aws.amazon.com, May 18, 2015.
49. Darryl Veitch and Kanthaiah Vijayalayan: “[Network Timing and the 2015 Leap Second](#),” at *17th International Conference on Passive and Active Measurement* (PAM), April 2016. doi:[10.1007/978-3-319-30505-9_29](https://doi.org/10.1007/978-3-319-30505-9_29)
50. “[Timekeeping in VMware Virtual Machines](#),” Information Guide, VMware, Inc., December 2011.
51. “[MiFID II / MiFIR: Regulatory Technical and Implementing Standards – Annex I \(Draft\)](#),” European Securities and Markets Authority, Report ESMA/2015/1464, September 2015.
52. Luke Bigum: “[Solving MiFID II Clock Synchronisation With Minimum Spend \(Part 1\)](#),” lmax.com, November 27, 2015.
53. Kyle Kingsbury: “[Call Me Maybe: Cassandra](#),” aphyr.com, September 24, 2013.
54. John Daily: “[Clocks Are Bad, or, Welcome to the Wonderful World of Distributed Systems](#),” basho.com, November 12, 2013.
55. Kyle Kingsbury: “[The Trouble with Timestamps](#),” aphyr.com, October 12, 2013.
56. Leslie Lamport: “[Time, Clocks, and the Ordering of Events in a Distributed System](#),” *Communications of the ACM*, volume 21, number 7, pages 558–565, July 1978. doi:[10.1145/359545.359563](https://doi.org/10.1145/359545.359563)
57. Sandeep Kulkarni, Murat Demirbas, Deepak Madepa, et al.: “[Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases](#),” State University of New York at Buffalo, Computer Science and Engineering Technical Report 2014-04, May 2014.
58. Justin Sheehy: “[There Is No Now: Problems With Simultaneity in Distributed Systems](#),” *ACM Queue*, volume 13, number 3, pages 36–41, March 2015. doi:[10.1145/2733108](https://doi.org/10.1145/2733108)
59. Murat Demirbas: “[Spanner: Google's Globally-Distributed Database](#),” muratbuffalo.blogspot.co.uk, July 4, 2013.
60. Dahlia Malkhi and Jean-Philippe Martin: “[Spanner's Concurrency Control](#),” *ACM SIGACT News*, volume 44, number 3, pages 73–77, September 2013. doi:[10.1145/2527748.2527767](https://doi.org/10.1145/2527748.2527767)
61. Manuel Bravo, Nuno Diegues, Jingna Zeng, et al.: “[On the Use of Clocks to Enforce Consistency in the Cloud](#),” *IEEE Data Engineering Bulletin*, volume 38, number 1, pages 18–31, March 2015.
62. Spencer Kimball: “[Living Without Atomic Clocks](#),” cockroachlabs.com, February 17, 2016.
63. Cary G. Gray and David R. Cheriton: “[Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency](#),” at *12th ACM Symposium on Operating Systems Principles* (SOSP), December 1989. doi:[10.1145/74850.74870](https://doi.org/10.1145/74850.74870)
64. Todd Lipcon: “[Avoiding Full GCs in Apache HBase with MemStore-Local Allocation Buffers: Part 1](#),” blog.cloudera.com, February 24, 2011.
65. Martin Thompson: “[Java Garbage Collection Distilled](#),” mechanical-sympathy.blogspot.co.uk, July 16, 2013.
66. Alexey Ragozin: “[How to Tame Java GC Pauses? Surviving 16GiB Heap and Greater](#),” java.dzone.com, June 28, 2011.
67. Christopher Clark, Keir Fraser, Steven Hand, et al.: “[Live Migration of Virtual Machines](#),” at *2nd USENIX Symposium on Symposium on Networked Systems Design & Implementation* (NSDI), May 2005.
68. Mike Shaver: “[fsyncers and Curveballs](#),” shaver.off.net, May 25, 2008.
69. Zhenyun Zhuang and Cuong Tran: “[Eliminating Large JVM GC Pauses Caused by Background IO Traffic](#),” engineering.linkedin.com, February 10, 2016.
70. David Terei and Amit Levy: “[Blade: A Data Center Garbage Collector](#),” arXiv:1504.02578, April 13, 2015.
71. Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz: “[Trash Day: Coordinating Garbage Collection in Distributed Systems](#),” at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.
72. “[Predictable Low Latency](#),” Cinnober Financial Technology AB, cinnober.com, November 24, 2013.
73. Martin Fowler: “[The LMAX Architecture](#),” martinfowler.com, July 12, 2011.
74. Flávio P. Junqueira and Benjamin Reed: *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, 2013.

- ISBN: 978-1-449-36130-3
75. Enis Söztutar: “[HBase and HDFS: Understanding Filesystem Usage in HBase](#),” at *HBaseCon*, June 2013.
 76. Caitie McCaffrey: “[Clients Are Jerks: AKA How Halo 4 DoSed the Services at Launch & How We Survived](#),” *caitiem.com*, June 23, 2015.
 77. Leslie Lamport, Robert Shostak, and Marshall Pease: “[The Byzantine Generals Problem](#),” *ACM Transactions on Programming Languages and Systems* (TOPLAS), volume 4, number 3, pages 382–401, July 1982.
[doi:10.1145/357172.357176](https://doi.org/10.1145/357172.357176)
 78. Jim N. Gray: “[Notes on Data Base Operating Systems](#),” in *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science, volume 60, edited by R. Bayer, R. M. Graham, and G. Seegmüller, pages 393–481, Springer-Verlag, 1978. ISBN: 978-3-540-08755-7
 79. Brian Palmer: “[How Complicated Was the Byzantine Empire?](#),” *slate.com*, October 20, 2011.
 80. Leslie Lamport: “[My Writings](#),” *research.microsoft.com*, December 16, 2014. This page can be found by searching the web for the 23-character string obtained by removing the hyphens from the string `a111a-mport-spubso-ntheweb`.
 81. John Rushby: “[Bus Architectures for Safety-Critical Embedded Systems](#),” at *1st International Workshop on Embedded Software* (EMSOFT), October 2001.
 82. Jake Edge: “[ELC: SpaceX Lessons Learned](#),” *lwn.net*, March 6, 2013.
 83. Andrew Miller and Joseph J. LaViola, Jr.: “[Anonymous Byzantine Consensus from Moderately-Hard Puzzles: A Model for Bitcoin](#),” University of Central Florida, Technical Report CS-TR-14-01, April 2014.
 84. James Mickens: “[The Saddest Moment](#),” *USENIX :login: logout*, May 2013.
 85. Evan Gilman: “[The Discovery of Apache ZooKeeper’s Poison Packet](#),” *pagerduty.com*, May 7, 2015.
 86. Jonathan Stone and Craig Partridge: “[When the CRC and TCP Checksum Disagree](#),” at *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (SIGCOMM), August 2000. [doi:10.1145/347059.347561](https://doi.org/10.1145/347059.347561)
 87. Evan Jones: “[How Both TCP and Ethernet Checksums Fail](#),” *evanjones.ca*, October 5, 2015.
 88. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer: “[Consensus in the Presence of Partial Synchrony](#),” *Journal of the ACM*, volume 35, number 2, pages 288–323, April 1988. [doi:10.1145/42282.42283](https://doi.org/10.1145/42282.42283)
 89. Peter Bailis and Ali Ghodsi: “[Eventual Consistency Today: Limitations, Extensions, and Beyond](#),” *ACM Queue*, volume 11, number 3, pages 55–63, March 2013. [doi:10.1145/2460276.2462076](https://doi.org/10.1145/2460276.2462076)
 90. Bowen Alpern and Fred B. Schneider: “[Defining Liveness](#),” *Information Processing Letters*, volume 21, number 4, pages 181–185, October 1985. [doi:10.1016/0020-0190\(85\)90056-0](https://doi.org/10.1016/0020-0190(85)90056-0))
 91. Flavio P. Junqueira: “[Dude, Where’s My Metadata?](#),” *fpj.me*, May 28, 2015.
 92. Scott Sanders: “[January 28th Incident Report](#),” *github.com*, February 3, 2016.
 93. Jay Kreps: “[A Few Notes on Kafka and Jepsen](#),” *blog.empathybox.com*, September 25, 2013.
 94. Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, et al.: “[Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems](#),” at *4th ACM Symposium on Cloud Computing* (SoCC), October 2013.
[doi:10.1145/2523616.2523627](https://doi.org/10.1145/2523616.2523627)
 95. Frank McSherry, Michael Isard, and Derek G. Murray: “[Scalability! But at What COST?](#),” at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

譯著¹。原詩為：Hey I just met you. The network’s laggy. But here’s my data. So store it maybe.Hey, 應改編自《Call Me Maybe》歌詞：I just met you, And this is crazy, But here’s my number, So call me, maybe? ↪

上一章	目錄	下一章
第七章：事務	設計資料密集型應用	第九章：一致性與共識

第九章：一致性與共識



好死還是賴活著？—— Jay Kreps, 關於 Kafka 與 Jepsen 的若干筆記 (2013)

[TOC]

正如 [第八章](#) 所討論的，分散式系統中的許多事情可能會出錯。處理這種故障的最簡單方法是簡單地讓整個服務失效，並向用戶顯示錯誤訊息。如果無法接受這個解決方案，我們就需要找到容錯的方法——即使某些內部元件出現故障，服務也能正常執行。

在本章中，我們將討論構建容錯分散式系統的演算法和協議的一些例子。我們將假設 [第八章](#) 的所有問題都可能發生：網路中的資料包可能會丟失、重新排序、重複推送或任意延遲；時鐘只是盡其所能地近似；且節點可以暫停（例如，由於垃圾收集）或隨時崩潰。

構建容錯系統的最好方法，是找到一些帶有實用保證的通用抽象，實現一次，然後讓應用依賴這些保證。這與 [第七章](#) 中的事務處理方法相同：透過使用事務，應用可以假裝沒有崩潰（原子性），沒有其他人同時訪問資料庫（隔離），儲存裝置是完全可靠的（永續性）。即使發生崩潰，競態條件和磁碟故障，事務抽象隱藏了這些問題，因此應用不必擔心它們。

現在我們將繼續沿著同樣的路線前進，尋求可以讓應用忽略分散式系統部分問題的抽象概念。例如，分散式系統最重要的抽象之一就是 **共識 (consensus)**：就是讓所有的節點對某件事達成一致。正如我們在本章中將會看到的那樣，要可靠地達成共識，且不被網路故障和程序故障所影響，是一個令人驚訝的棘手問題。

一旦達成共識，應用可以將其用於各種目的。例如，假設你有一個單主複製的資料庫。如果主庫掛掉，並且需要故障切換到另一個節點，剩餘的資料庫節點可以使用共識來選舉新的領導者。正如在“[處理節點宕機](#)”中所討論的那樣，重要的是隻有一個領導者，且所有的節點都認同其領導。如果兩個節點都認為自己是領導者，這種情況被稱為 **腦裂 (split)**

brain)，它經常會導致資料丟失。正確實現共識有助於避免這種問題。

在本章後面的“[分散式事務與共識](#)”中，我們將研究解決共識和相關問題的演算法。但首先，我們首先需要探索可以在分散式系統中提供的保證和抽象的範圍。

我們需要了解可以做什麼和不可以做什麼的範圍：在某些情況下，系統可以容忍故障並繼續工作；在其他情況下，這是不可能的。我們將深入研究什麼可能而什麼不可能的限制，既透過理論證明，也透過實際實現。我們將在本章中概述這些基本限制。

分散式系統領域的研究人員幾十年來一直在研究這些主題，所以有很多資料——我們只能介紹一些皮毛。在本書中，我們沒有空間去詳細介紹形式模型和證明的細節，所以我們會按照直覺來介紹。如果你有興趣，參考文獻可以提供更多深度。

一致性保證

在“[複製延遲問題](#)”中，我們看到了資料庫複製中發生的一些時序問題。如果你在同一時刻檢視兩個資料庫節點，則可能在兩個節點上看到不同的資料，因為寫請求在不同的時間到達不同的節點。無論資料庫使用何種複製方法（單主複製，多主複製或無主複製），都會出現這些不一致情況。

大多數複製的資料庫至少提供了**最終一致性**，這意味著如果你停止向資料庫寫入資料並等待一段不確定的時間，那麼最終所有的讀取請求都會返回相同的值【1】。換句話說，不一致性是暫時的，最終會自行解決（假設網路中的任何故障最終都會被修復）。最終一致性的一個更好的名字可能是**收斂**（convergence），因為我們預計所有的副本最終會收斂到相同的值【2】。

然而，這是一個非常弱的保證——它並沒有說什麼時候副本會收斂。在收斂之前，讀操作可能會返回任何東西或什麼都沒有【1】。例如，如果你寫入了一個值，然後立即再次讀取，這並不能保證你能看到剛才寫入的值，因為讀請求可能會被路由到另外的副本上。（請參閱“[讀已之寫](#)”）。

對於應用開發人員而言，最終一致性是很困難的，因為它與普通單執行緒程式中變數的行為有很大區別。對於後者，如果將一個值賦給一個變數，然後很快地再次讀取，不可能讀到舊的值，或者讀取失敗。資料庫表面上看起來像一個你可以讀寫的變數，但實際上它有更複雜的語義【3】。

在與只提供弱保證的資料庫打交道時，你需要始終意識到它的侷限性，而不是意外地作出太多假設。錯誤往往是微妙的，很難找到，也很難測試，因為應用可能在大多數情況下執行良好。當系統出現故障（例如網路中斷）或高併發時，最終一致性的邊緣情況才會顯現出來。

本章將探索資料系統可能選擇提供的更強一致性模型。它不是免費的：具有較強保證的系統可能會比保證較差的系統具有更差的效能或更少的容錯性。儘管如此，更強的保證能夠吸引人，因為它們更容易用對。只有見過不同的一致性模型後，才能更好地決定哪一個最適合自己的需求。

分散式一致性模型 和我們之前討論的事務隔離級別的層次結構有一些相似之處【4,5】（請參閱“[弱隔離級別](#)”）。儘管兩者有一部分內容重疊，但它們大多是無關的問題：事務隔離主要是為了避免由於同時執行事務而導致的競爭狀態，而分散式一致性主要關於在面對延遲和故障時如何協調副本間的狀態。

本章涵蓋了廣泛的話題，但我們將會看到這些領域實際上是緊密聯絡在一起的：

- 首先看一下常用的**最強一致性模型**之一，**線性一致性**（linearizability），並考察其優缺點。
- 然後我們將檢查分散式系統中**事件順序**的問題，特別是因果關係和全域性順序的問題。
- 在第三節的（“[分散式事務與共識](#)”）中將探討如何原子地提交分散式事務，這將最終引領我們走向共識問題的解決方案。

線性一致性

在最終一致的資料庫，如果你在同一時刻問兩個不同副本相同的問題，可能會得到兩個不同的答案。這很讓人困惑。如果資料庫可以提供只有一個副本的假象（即，只有一個數據副本），那麼事情就簡單太多了。那麼每個客戶端都會有相同的資料檢視，且不必擔心複製滯後了。

這就是線性一致性 (linearizability) 背後的想法【6】（也稱為原子一致性 (atomic consistency) 【7】，強一致性 (strong consistency)，立即一致性 (immediate consistency) 或外部一致性 (external consistency) 【8】）。線性一致性的精確定義相當微妙，我們將在本節的剩餘部分探討它。但是基本的想法是讓一個系統看起來好像只有一個數據副本，而且所有的操作都是原子性的。有了這個保證，即使實際中可能有多個副本，應用也不需要擔心它們。

在一個線性一致的系統中，只要一個客戶端成功完成寫操作，所有客戶端從資料庫中讀取資料必須能夠看到剛剛寫入的值。要維護資料的單個副本的假象，系統應保障讀到的值是最新的、最新的，而不是來自陳舊的快取或副本。換句話說，線性一致性是一個新鮮度保證 (recency guarantee)。為了闡明這個想法，我們來看看一個非線性一致系統的例子。

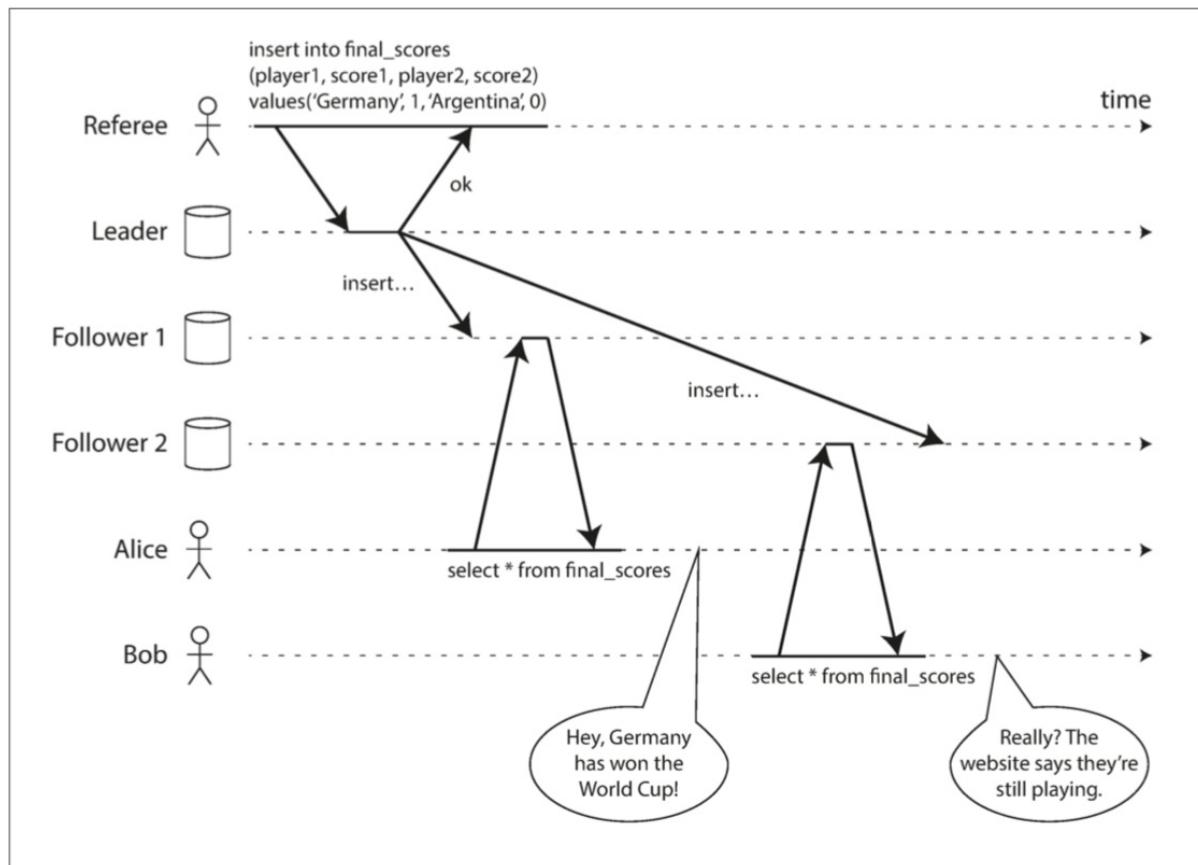


圖 9-1 這個系統是非線性一致的，導致了球迷的困惑

圖 9-1 展示了一個關於體育網站的非線性一致例子【9】。Alice 和 Bob 正坐在同一個房間裡，都盯著各自的手機，關注著 2014 年 FIFA 世界盃決賽的結果。在最後得分公佈後，Alice 重新整理頁面，看到宣佈了獲勝者，並興奮地告訴 Bob。Bob 難以置信地重新整理了自己的手機，但他的請求路由到了一個落後的資料庫副本上，手機顯示比賽仍在進行。

如果 Alice 和 Bob 在同一時間重新整理並獲得了兩個不同的查詢結果，也許就沒有那麼令人驚訝了。因為他們不知道伺服器處理他們請求的精確時刻。然而 Bob 是在聽到 Alice 驚呼最後得分之後，點選了重新整理按鈕（啟動了他的查詢），因此他希望查詢結果至少與愛麗絲一樣新鮮。但他的查詢返回了陳舊結果，這一事實違背了線性一致性的要求。

什麼使得系統線性一致？

線性一致性背後的基本思想很簡單：使系統看起來好像只有一個數據副本。然而確切來講，實際上有更多要操心的地方。為了更好地理解線性一致性，讓我們再看幾個例子。

圖 9-2 顯示了三個客戶端線性一致資料庫中同時讀寫相同的鍵 x 。在分散式系統文獻中， x 被稱為 **暫存器 (register)**，例如，它可以是鍵值儲存中的一個 **鍵**，關係資料庫中的一行，或文件資料庫中的一個 **文件**。

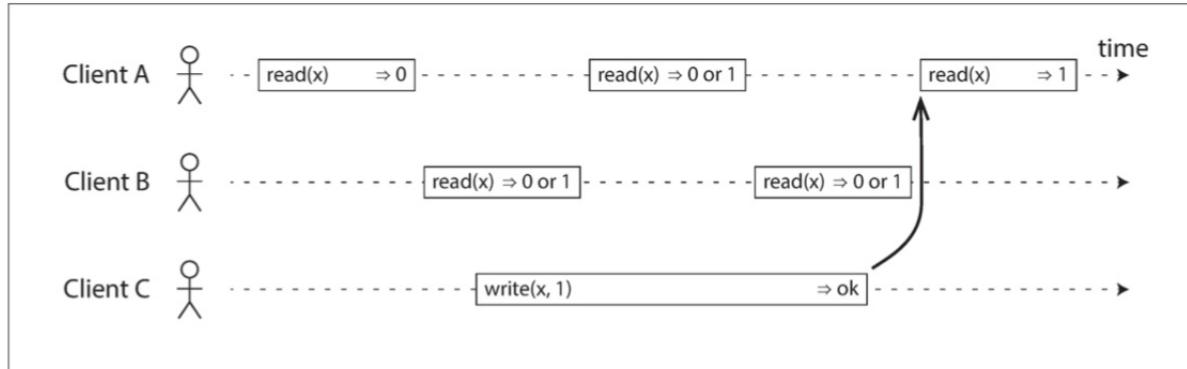


圖 9-2 如果讀取請求與寫入請求併發，則可能會返回舊值或新值

為了簡單起見，圖 9-2 採用了使用者請求的視角，而不是資料庫內部的視角。每個柱都是由客戶端發出的請求，其中柱頭是請求傳送的時刻，柱尾是客戶端收到響應的時刻。因為網路延遲變化無常，客戶端不知道資料庫處理其請求的精確時間——只知道它發生在傳送請求和接收響應之間的某個時刻。ⁱ

ⁱ. 這個圖的一個微妙的細節是它假定存在一個全域性時鐘，由水平軸表示。即使真實的系統通常沒有準確的時鐘（請參閱“[不可靠的時鐘](#)”），但這種假設是允許的：為了分析分散式演算法，我們可以假設一個精確的全域性時鐘存在，不過演算法無法訪問它【47】。演算法只能看到由石英振盪器和 NTP 產生的實時逼近。 ↩

在這個例子中，暫存器有兩種型別的操作：

- \$read(x) $\Rightarrow v$ \$ 表示客戶端請求讀取暫存器 x 的值，資料庫返回值 v 。
- \$write(x, v) $\Rightarrow r$ \$ 表示客戶端請求將暫存器 x 設定為值 v ，資料庫返回響應 r （可能正確，可能錯誤）。

在 圖 9-2 中， x 的值最初為 0 ，客戶端 C 執行寫請求將其設定為 1 。發生這種情況時，客戶端 A 和 B 反覆輪詢資料庫以讀取最新值。A 和 B 的請求可能會收到怎樣的響應？

- 客戶端 A 的第一個讀操作，完成於寫操作開始之前，因此必須返回舊值 0 。
- 客戶端 A 的最後一個讀操作，開始於寫操作完成之後。如果資料庫是線性一致性的，它必然返回新值 1 ：因為讀操作和寫操作一定是在其各自的起止區間內的某個時刻被處理。如果在寫入結束後開始讀取，則讀取處理一定發生在寫入完成之後，因此它必須看到寫入的新值。
- 與寫操作在時間上重疊的任何讀操作，可能會返回 0 或 1 ，因為我們不知道讀取時，寫操作是否已經生效。這些操作是 **併發 (concurrent)** 的。

但是，這還不足以完全描述線性一致性：如果與寫入同時發生的讀取可以返回舊值或新值，那麼讀者可能會在寫入期間看到數值在舊值和新值之間來回翻轉。這不是我們所期望的模擬“單一資料副本”的系統。ⁱⁱ

ⁱⁱ. 如果讀取（與寫入同時發生時）可能返回舊值或新值，則稱該暫存器為 **常規暫存器 (regular register)** 【7,25】 ↩

為了使系統線性一致，我們需要新增另一個約束，如 圖 9-3 所示

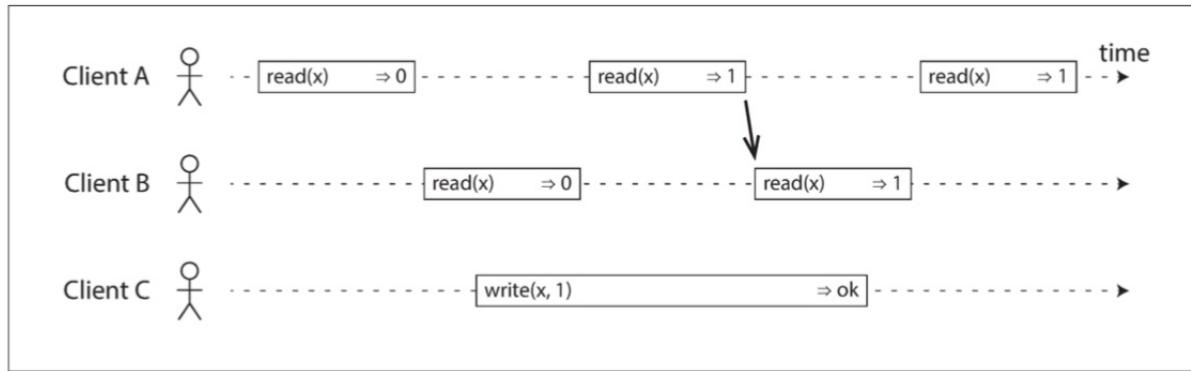


圖 9-3 任何一個讀取返回新值後，所有後續讀取（在相同或其他客戶端上）也必須返回新值。

在一個線性一致的系統中，我們可以想象，在 x 的值從 0 自動翻轉到 1 的時候（在寫操作的開始和結束之間）必定有一個時間點。因此，如果一個客戶端的讀取返回新的值 1 ，即使寫操作尚未完成，所有後續讀取也必須返回新值。

圖 9-3 中的箭頭說明了這個時序依賴關係。客戶端 A 是第一個讀取新的值 1 的位置。在 A 的讀取返回之後，B 開始新的讀取。由於 B 的讀取嚴格在發生於 A 的讀取之後，因此即使 C 的寫入仍在進行中，也必須返回 1 （與 圖 9-1 中的 Alice 和 Bob 的情況相同：在 Alice 讀取新值之後，Bob 也希望讀取新的值）。

我們可以進一步細化這個時序圖，展示每個操作是如何在特定時刻原子性生效的。圖 9-4 顯示了一個更複雜的例子【10】。

在 圖 9-4 中，除了讀寫之外，還增加了第三種類型的操作：

- $\$cas(x, v\{old\}, v\{new\}) \Rightarrow r\$$ 表示客戶端請求進行原子性的 比較與設定 操作。如果暫存器 $\$x\$$ 的當前值等於 $v\{old\}$ ，則應該原子地設定為 $v\{new\}$ 。如果 $\$x\$$ 不等於 $v\{old\}$ ，則操作應該保持暫存器不變並返回一個錯誤。 $r\$$ 是資料庫的響應（正確或錯誤）。

圖 9-4 中的每個操作都在我們認為執行操作的時候用豎線標出（在每個操作的條柱之內）。這些標記按順序連在一起，其結果必須是一個有效的暫存器讀寫序列（每次讀取都必須返回最近一次寫入設定的值）。

線性一致性的要求是，操作標記的連線總是按時間（從左到右）向前移動，而不是向後移動。這個要求確保了我們之前討論的新鮮度保證：一旦新的值被寫入或讀取，所有後續的讀都會看到寫入的值，直到它被再次覆蓋。

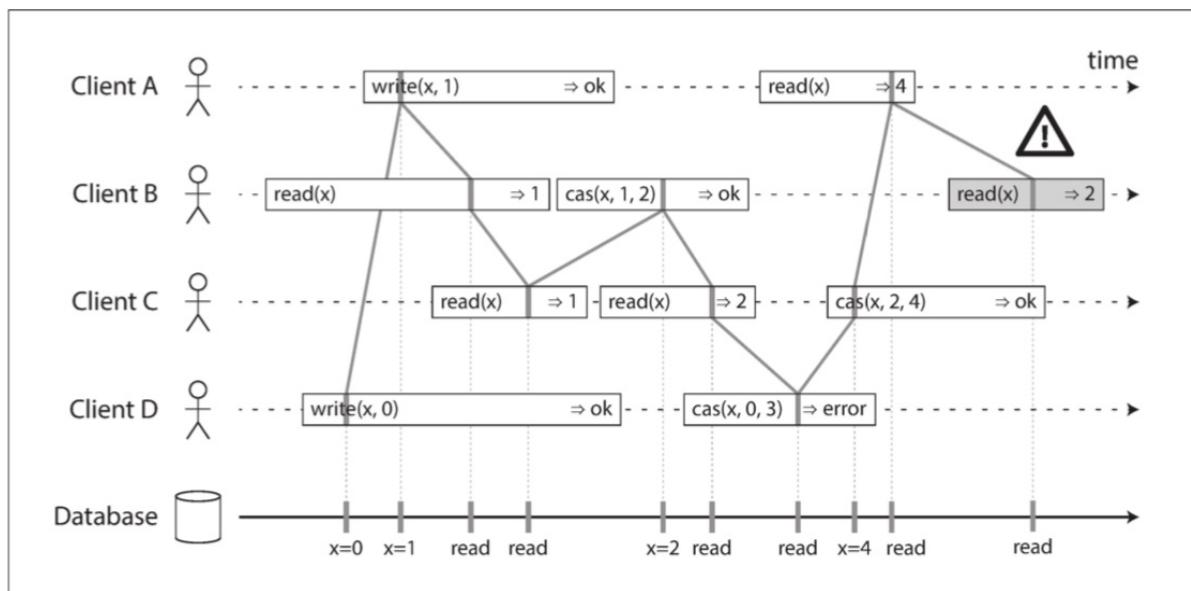


圖 9-4 視覺化讀取和寫入看起來已經生效的時間點。B 的最後讀取不是線性一致性的

圖 9-4 中有一些有趣的細節需要指出：

- 第一個客戶端 B 傳送一個讀取 x 的請求，然後客戶端 D 傳送一個請求將 x 設定為 0 ，然後客戶端 A 傳送請求將 x 設定為 1 。儘管如此，返回到 B 的讀取值為 1 （由 A 寫入的值）。這是可以的：這意味著資料庫首先處理 D 的寫入，然後是 A 的寫入，最後是 B 的讀取。雖然這不是請求傳送的順序，但這是一個可以接受的順序，因為這三個請求是併發的。也許 B 的讀請求在網路上略有延遲，所以它在兩次寫入之後才到達資料庫。
- 在客戶端 A 從資料庫收到響應之前，客戶端 B 的讀取返回 1 ，表示寫入值 1 已成功。這也是可以的：這並不意味著在寫之前讀到了值，這只是意味著從資料庫到客戶端 A 的正確響應在網路中略有延遲。
- 此模型不假設有任何事務隔離：另一個客戶端可能隨時更改值。例如，C 首先讀取 1 ，然後讀取 2 ，因為兩次讀取之間的值由 B 更改。可以使用原子 **比較並設定 (cas)** 操作來檢查該值是否未被另一客戶端同時更改：B 和 C 的 cas 請求成功，但是 D 的 cas 請求失敗（在資料庫處理它時， x 的值不再是 0 ）。
- 客戶 B 的最後一次讀取（陰影條柱中）不是線性一致性的。該操作與 C 的 cas 寫操作併發（它將 x 從 2 更新為 4 ）。在沒有其他請求的情況下，B 的讀取返回 2 是可以的。然而，在 B 的讀取開始之前，客戶端 A 已經讀取了新的值 4 ，因此不允許 B 讀取比 A 更舊的值。再次，與 [圖 9-1](#) 中的 Alice 和 Bob 的情況相同。

這就是線性一致性背後的直覺。正式的定義 [【6】](#) 更準確地描述了它。透過記錄所有請求和響應的時序，並檢查它們是否可以排列成有效的順序，以測試一個系統的行為是否線性一致性是可能的（儘管在計算上是昂貴的）[【11】](#)。

線性一致性與可序列化

線性一致性容易和 [可序列化](#) 相混淆，因為兩個詞似乎都是類似“可以按順序排列”的東西。但它們是兩種完全不同的保證，區分兩者非常重要：

可序列化

[可序列化 \(Serializability\)](#) 是事務的隔離屬性，每個事務可以讀寫多個物件（行，文件，記錄）——請參閱“[單物件和多物件操作](#)”。它確保事務的行為，與它們按照某種順序依次執行的結果相同（每個事務在下一個事務開始之前執行完成）。這種執行順序可以與事務實際執行的順序不同。[【12】](#)。

線性一致性

[線性一致性 \(Linearizability\)](#) 是讀取和寫入暫存器（單個物件）的新鮮度保證。它不會將操作組合為事務，因此它也不會阻止寫入偏差等問題（請參閱“[寫入偏差和幻讀](#)”），除非採取其他措施（例如 [物化衝突](#)）。

一個數據庫可以提供可序列化和線性一致性，這種組合被稱為嚴格的可序列化或 **強的單副本可序列化 (strong-1SR)** [【4,13】](#)。基於兩階段鎖定的可序列化實現（請參閱“[兩階段鎖定](#)”一節）或 **真的序列執行**（請參閱“[真的序列執行](#)”一節）通常是線性一致性的。

但是，可序列化的快照隔離（請參閱“[可序列化快照隔離](#)”）不是線性一致性的：按照設計，它從一致的快照中進行讀取，以避免讀者和寫者之間的鎖競爭。一致性快照的要點就在於它不會包括該快照之後的寫入，因此從快照讀取不是線性一致性的。

依賴線性一致性

線性一致性在什麼情況下有用？觀看體育比賽的最後得分可能是一個輕率的例子：滯後了幾秒鐘的結果不太可能在這種情況下造成任何真正的傷害。然而對於少數領域，線性一致性是系統正確工作的一個重要條件。

鎖定和領導選舉

一個使用單主複製的系統，需要確保領導者真的只有一個，而不是幾個（腦裂）。一種選擇領導者的方法是使用鎖：每個節點在啟動時嘗試獲取鎖，成功者成為領導者[【14】](#)。不管這個鎖是如何實現的，它必須是線性一致的：所有節點必須就哪個節點擁有鎖達成一致，否則就沒用了。

諸如 Apache ZooKeeper 【15】和 etcd 【16】之類的協調服務通常用於實現分散式鎖和領導者選舉。它們使用一致性演算法，以容錯的方式實現線性一致的操作（在本章後面的“[容錯共識](#)”中討論此類演算法）ⁱⁱⁱ。還有許多微妙的細節來正確地實現鎖和領導者選舉（例如，請參閱“[領導者和鎖](#)”中的防護問題），而像 Apache Curator 【17】這樣的庫則透過在 ZooKeeper 之上提供更高級別的配方來提供幫助。但是，線性一致性儲存服務是這些協調任務的基礎。

ⁱⁱⁱ. 嚴格地說，ZooKeeper 和 etcd 提供線性一致性的寫操作，但讀取可能是陳舊的，因為預設情況下，它們可以由任何一個副本提供服務。你可以選擇請求線性一致性讀取：etcd 稱之為 **法定人數讀取** (**quorum read**) 【16】，而在 ZooKeeper 中，你需要在讀取之前呼叫 `sync()` 【15】。請參閱[“使用全序廣播實現線性一致的儲存”](#)。 ↵

分散式鎖也在一些分散式資料庫（如 Oracle Real Application Clusters (RAC) 【18】）中有更細粒度級別的使用。RAC 對每個磁碟頁面使用一個鎖，多個節點共享對同一個磁碟儲存系統的訪問許可權。由於這些線性一致的鎖處於事務執行的關鍵路徑上，RAC 部署通常具有用於資料庫節點之間通訊的專用叢集互連網路。

約束和唯一性保證

唯一性約束在資料庫中很常見：例如，使用者名稱或電子郵件地址必須唯一標識一個使用者，而在檔案儲存服務中，不能有兩個具有相同路徑和檔名的檔案。如果要在寫入資料時強制執行此約束（例如，如果兩個人試圖同時建立一個具有相同名稱的使用者或檔案，其中一個將返回一個錯誤），則需要線性一致性。

這種情況實際上類似於一個鎖：當一個使用者註冊你的服務時，可以認為他們獲得了所選使用者名稱的“鎖”。該操作與原子性的比較與設定 (CAS) 非常相似：將使用者名稱賦予宣告它的使用者，前提是使用者名稱尚未被使用。

如果想要確保銀行賬戶餘額永遠不會為負數，或者不會出售比倉庫裡的庫存更多的物品，或者兩個人不會都預定了航班或劇院裡同一時間的同一個位置。這些約束條件都要求所有節點都同意一個最新的值（賬戶餘額，庫存水平，座位佔用率）。

在實際應用中，寬鬆地處理這些限制有時是可以接受的（例如，如果航班超額預訂，你可以將客戶轉移到不同的航班併為其提供補償）。在這種情況下，可能不需要線性一致性，我們將在“[及時性與完整性](#)”中討論這種寬鬆的約束。

然而，一個硬性的唯一性約束（關係型資料庫中常見的那種）需要線性一致性。其他型別的約束，如外來鍵或屬性約束，可以不需要線性一致性【19】。

跨通道的時序依賴

注意 [圖 9-1](#) 中的一個細節：如果 Alice 沒有驚呼得分，Bob 就不會知道他的查詢結果是陳舊的。他會在幾秒鐘之後再次重新整理頁面，並最終看到最後的分數。由於系統中存在額外的通道（Alice 的聲音傳到了 Bob 的耳朵中），線性一致性的違背才被注意到。

計算機系統也會出現類似的情況。例如，假設有一個網站，使用者可以上傳照片，一個後臺程序會調整照片大小，降低解析度以加快下載速度（縮圖）。該系統的架構和資料流如 [圖 9-5](#) 所示。

影象縮放器需要明確的指令來執行尺寸縮放作業，指令是 Web 伺服器透過訊息佇列傳送的（請參閱 [第十一章](#)）。Web 伺服器不會將整個照片放在佇列中，因為大多數訊息代理都是針對較短的訊息而設計的，而一張照片的空間佔用可能達到幾兆位元組。取而代之的是，首先將照片寫入檔案儲存服務，寫入完成後再將給縮放器的指令放入訊息佇列。

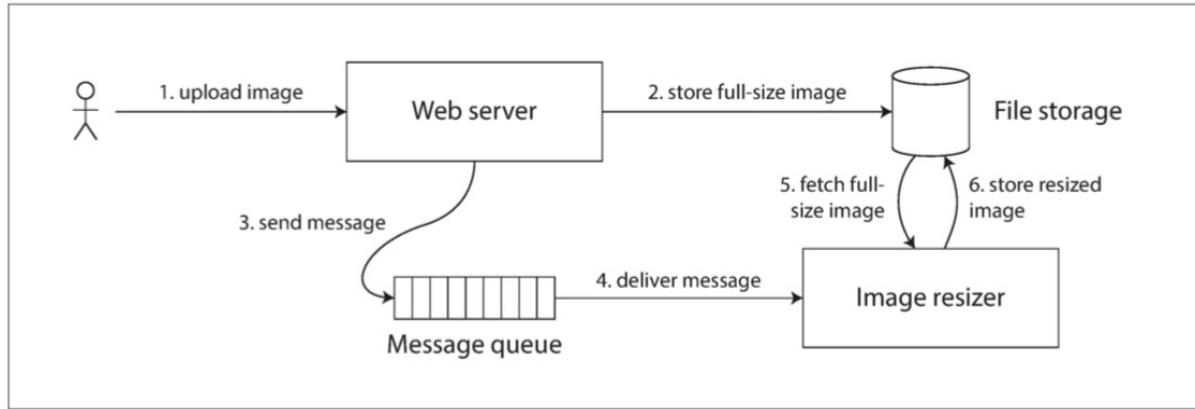


圖 9-5 Web 伺服器和影像縮放器透過檔案儲存和訊息佇列進行通訊，開啟競爭條件的可能性。

如果檔案儲存服務是線性一致的，那麼這個系統應該可以正常工作。如果它不是線性一致的，則存在競爭條件的風險：訊息佇列（圖 9-5 中的步驟 3 和 4）可能比儲存服務內部的複製（replication）更快。在這種情況下，當縮放器讀取影象（步驟 5）時，可能會看到影象的舊版本，或者什麼都沒有。如果它處理的是舊版本的影象，則檔案儲存中的全尺寸圖和縮圖就產生了永久性的不一致。

出現這個問題是因為 Web 伺服器和縮放器之間存在兩個不同的通道：檔案儲存與訊息佇列。沒有線性一致性的新鮮性保證，這兩個通道之間的競爭條件是可能的。這種情況類似於 圖 9-1，資料庫複製與 Alice 的嘴到 Bob 耳朵之間的真人音訊通道之間也存在競爭條件。

線性一致性並不是避免這種競爭條件的唯一方法，但它是最容易理解的。如果你可以控制額外通道（例如訊息佇列的例子，而不是在 Alice 和 Bob 的例子），則可以使用在“[讀已之寫](#)”討論過的類似方法，不過會有額外的複雜度代價。

實現線性一致的系統

我們已經見到了幾個線性一致性有用的例子，讓我們思考一下，如何實現一個提供線性一致語義的系統。

由於線性一致性本質上意味著“表現得好像只有一個數據副本，而且所有的操作都是原子的”，所以最簡單的答案就是，真的只用一個數據副本。但是這種方法無法容錯：如果持有該副本的節點失效，資料將會丟失，或者至少無法訪問，直到節點重新啟動。

使系統容錯最常用的方法是使用複製。我們再來回顧 第五章 中的複製方法，並比較它們是否可以滿足線性一致性：

- 單主複製（可能線性一致）

在具有單主複製功能的系統中（請參閱“[領導者與追隨者](#)”），主庫具有用於寫入的資料的主副本，而追隨者在其他節點上保留資料的備份副本。如果從主庫或同步更新的從庫讀取資料，它們可能（potential）是線性一致性的^{iv}。然而，實際上並不是每個單主資料庫都是線性一致性的，無論是因為設計的原因（例如，因為使用了快照隔離）還是因為在併發處理上存在錯誤【10】。

^{iv}. 對單主資料庫進行分割槽（分片），使得每個分割槽有一個單獨的領導者，不會影響線性一致性，因為線性一致性只是對單一物件的保證。交叉分割槽事務是一個不同的問題（請參閱“[分散式事務與共識](#)”）。 ↩

從主庫讀取依賴一個假設，你確切地知道領導者是誰。正如在“[真相由多數所定義](#)”中所討論的那樣，一個節點很可能會認為它是領導者，而事實上並非如此——如果具有錯覺的領導者繼續為請求提供服務，可能違反線性一致性【20】。使用非同步複製，故障切換時甚至可能會丟失已提交的寫入（請參閱“[處理節點宕機](#)”），這同時違反了永續性和線性一致性。

- 共識演算法（線性一致）

一些在本章後面討論的共識演算法，與單主複製類似。然而，共識協議包含防止腦裂和陳舊副本的措施。正是由於這些細節，共識演算法可以安全地實現線性一致性儲存。例如，Zookeeper【21】和 etcd【22】就是這樣工作的。

- 多主複製（非線性一致）

具有多主程式複製的系統通常不是線性一致的，因為它們同時在多個節點上處理寫入，並將其非同步複製到其他節點。因此，它們可能會產生需要被解決的寫入衝突（請參閱“[處理寫入衝突](#)”）。這種衝突是因為缺少單一資料副本所導致的。

- 無主複製（也許不是線性一致的）

對於無主複製的系統（Dynamo 風格；請參閱“[無主複製](#)”），有時候人們會聲稱透過要求法定人數讀寫（ $\$w + r > n\$$ ）可以獲得“強一致性”。這取決於法定人數的具體配置，以及強一致性如何定義（通常不完全正確）。

基於日曆時鐘（例如，在 Cassandra 中；請參閱“[依賴同步時鐘](#)”）的“最後寫入勝利”衝突解決方法幾乎可以確定是非線性一致的，由於時鐘偏差，不能保證時鐘的時間戳與實際事件順序一致。寬鬆的法定人數（請參閱“[寬鬆的法定人數與提示移交](#)”）也破壞了線性一致的可能性。即使使用嚴格的法定人數，非線性一致的行為也是可能的，如下節所示。

線性一致性和法定人數

直覺上在 Dynamo 風格的模型中，嚴格的法定人數讀寫應該是線性一致性的。但是當我們有可變的網路延遲時，就可能存在競爭條件，如 [圖 9-6](#) 所示。

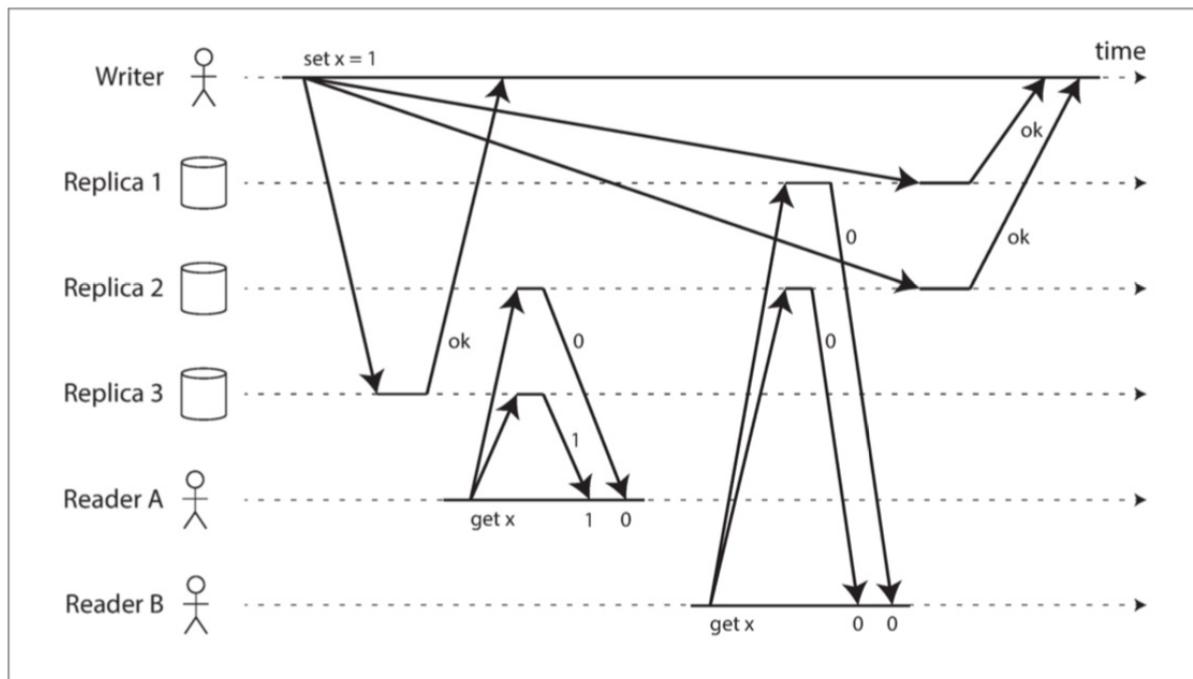


圖 9-6 非線性一致的執行，儘管使用了嚴格的法定人數

在 [圖 9-6](#) 中， $\$x\$$ 的初始值為 0，寫入客戶端透過向所有三個副本（ $\$n = 3, w = 3\$$ ）傳送寫入將 $\$x\$$ 更新為 1。客戶端 A 併發地從兩個節點組成的法定人群（ $\$r = 2\$$ ）中讀取資料，並在其中一個節點上看到新值 1。客戶端 B 也併發地從兩個不同的節點組成的法定人數中讀取，並從兩個節點中收回了舊值 0。

法定人數條件滿足（ $\$w + r > n\$$ ），但是這個執行是非線性一致的：B 的請求在 A 的請求完成後開始，但是 B 返回舊值，而 A 返回新值。（又一次，如同 Alice 和 Bob 的例子 [圖 9-1](#)）

有趣的是，透過犧牲效能，可以使 Dynamo 風格的法定人數線性化：讀取者必須在將結果返回給應用之前，同步執行讀修復（請參閱“[讀修復和反熵](#)”），並且寫入者必須在傳送寫入之前，讀取法定數量節點的最新狀態【24,25】。然而，由於效能損失，Riak 不執行同步讀修復【26】。Cassandra 在進行法定人數讀取時，確實在等待讀修復完成【27】；但是由於使用了最後寫入勝利的衝突解決方案，當同一個鍵有多個併發寫入時，將不能保證線性一致性。

而且，這種方式只能實現線性一致的讀寫；不能實現線性一致的比較和設定（CAS）操作，因為它需要一個共識演算法【28】。

總而言之，最安全的做法是：假設採用 Dynamo 風格無主複製的系統不能提供線性一致性。

線性一致性的代價

一些複製方法可以提供線性一致性，另一些複製方法則不能，因此深入地探討線性一致性的優缺點是很有趣的。

我們已經在 第五章 中討論了不同複製方法的一些用例。例如對多資料中心的複製而言，多主複製通常是理想的選擇（請參閱“[運維多個數據中心](#)”）。圖 9-7 說明了這種部署的一個例子。

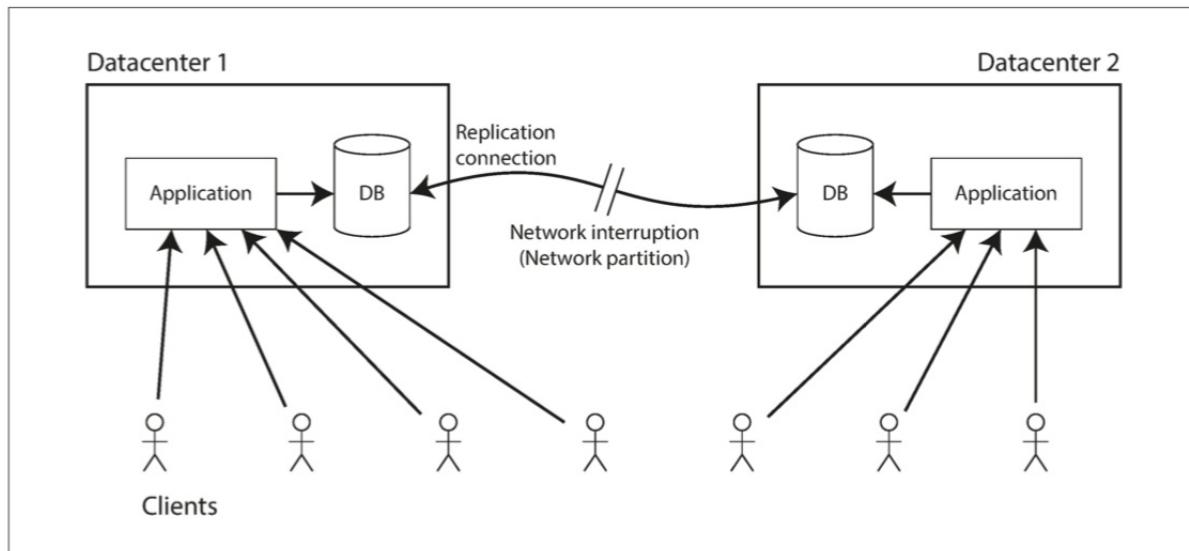


圖 9-7 網路中斷迫使線性和可用性之間做出選擇。

考慮這樣一種情況：如果兩個資料中心之間發生網路中斷會發生什麼？我們假設每個資料中心內的網路正在工作，客戶端可以訪問資料中心，但資料中心之間彼此無法互相連線。

使用多主資料庫，每個資料中心都可以繼續正常執行：由於在一個數據中心寫入的資料是非同步複製到另一個數據中心的，所以在恢復網路連線時，寫入操作只是簡單地排隊並交換。

另一方面，如果使用單主複製，則主庫必須位於其中一個數據中心。任何寫入和任何線性一致的讀取請求都必須傳送給該主庫，因此對於連線到從庫所在資料中心的客戶端，這些讀取和寫入請求必須透過網路同步傳送到主庫所在的資料中心。

在單主配置的條件下，如果資料中心之間的網路被中斷，則連線到從庫資料中心的客戶端無法聯絡到主庫，因此它們無法對資料庫執行任何寫入，也不能執行任何線性一致的讀取。它們仍能從從庫讀取，但結果可能是陳舊的（非線性一致）。如果應用需要線性一致的讀寫，卻又位於與主庫網路中斷的資料中心，則網路中斷將導致這些應用不可用。

如果客戶端可以直接連線到主庫所在的資料中心，這就不是問題了，那些應用可以繼續正常工作。但只能訪問從庫資料中心的客戶端會中斷執行，直到網路連線得到修復。

CAP定理

這個問題不僅僅是單主複製和多主複製的後果：任何線性一致的資料庫都有這個問題，不管它是如何實現的。這個問題也不僅僅侷限於多資料中心部署，而可能發生在任何不可靠的網路上，即使在同一個資料中心內也是如此。問題面臨的權衡如下：^V

- 如果應用需要線性一致性，且某些副本因為網路問題與其他副本斷開連線，那麼這些副本掉線時不能處理請求。請求必須等到網路問題解決，或直接返回錯誤。（無論哪種方式，服務都不可用）。
- 如果應用不需要線性一致性，那麼某個副本即使與其他副本斷開連線，也可以獨立處理請求（例如多主複製）。在

這種情況下，應用可以在網路問題解決前保持可用，但其行為不是線性一致的。

v. 這兩種選擇有時分別稱為 CP（在網路分割槽下一致但不可用）和 AP（在網路分割槽下可用但不一致）。但是，這種分類方案存在一些缺陷【9】，所以最好不要這樣用。 ↪

因此，不需要線性一致性的應用對網路問題有更強的容錯能力。這種見解通常被稱為 CAP 定理【29,30,31,32】，由 Eric Brewer 於 2000 年命名，儘管 70 年代的分散式資料庫設計者早就知道了這種權衡【33,34,35,36】。

CAP 最初是作為一個經驗法則提出的，沒有準確的定義，目的是開始討論資料庫的權衡。那時候許多分散式資料庫側重於在共享儲存的叢集上提供線性一致性的語義【18】，CAP 定理鼓勵資料庫工程師向分散式無共享系統的設計領域深入探索，這類架構更適合實現大規模的網路服務【37】。對於這種文化上的轉變，CAP 值得讚揚——它見證了自 00 年代中期以來新資料庫的技術爆炸（即 NoSQL）。

CAP定理沒有幫助

CAP 有時以這種面目出現：一致性，可用性和分割槽容錯性：三者只能擇其二。不幸的是這種說法很有誤導性【32】，因為網路分割槽是一種故障型別，所以它並不是一個選項：不管你喜不喜歡它都會發生【38】。

在網路正常工作的時候，系統可以提供一致性（線性一致性）和整體可用性。發生網路故障時，你必須線性一致性和整體可用性之間做出選擇。因此，CAP 更好的表述成：在分割槽時要麼選擇一致，要麼選擇可用【39】。一個更可靠的網路需要減少這個選擇，但是在某些時候選擇是不可避免的。

在 CAP 的討論中，術語可用性有幾個相互矛盾的定義，形式化作為一個定理【30】並不符合其通常的含義【40】。許多所謂的“高可用”（容錯）系統實際上不符合 CAP 對可用性的特殊定義。總而言之，圍繞著 CAP 有很多誤解和困惑，並不能幫助我們更好地理解系統，所以最好避免使用 CAP。

CAP 定理的正式定義僅限於很狹隘的範圍【30】，它只考慮了一個一致性模型（即線性一致性）和一種故障（網路分割槽^{vi}，或活躍但彼此斷開的節點）。它沒有討論任何關於網路延遲，死亡節點或其他權衡的事。因此，儘管 CAP 在歷史上有一些影響力，但對於設計系統而言並沒有實際價值【9,40】。

在分散式系統中有更多有趣的“不可能”的結果【41】，且 CAP 定理現在已經被更精確的結果取代【2,42】，所以它現在基本上成了歷史古蹟了。

vⁱ. 正如“真實世界的網路故障”中所討論的，本書使用分割槽（partition）指代將大資料集細分為小資料集的操作（分片；請參閱第六章）。與之對應的是，網路分割槽（network partition）是一種特定型別的網路故障，我們通常不會將其與其他型別的故障分開考慮。但是，由於它是 CAP 的 P，所以這種情況下我們無法避免混亂。 ↪

線性一致性和網路延遲

雖然線性一致是一個很有用的保證，但實際上，線性一致的系統驚人的少。例如，現代多核 CPU 上的記憶體甚至都不是線性一致的【43】：如果一個 CPU 核上執行的執行緒寫入某個記憶體地址，而另一個 CPU 核上執行的執行緒不久之後讀取相同的地址，並沒有保證一定能讀到第一個執行緒寫入的值（除非使用了記憶體屏障（memory barrier）或圍欄（fence）【44】）。

這種行為的原因是每個 CPU 核都有自己的記憶體快取和儲存緩衝區。預設情況下，記憶體訪問首先走快取，任何變更會非同步寫入主存。因為快取訪問比主存要快得多【45】，所以這個特性對於現代 CPU 的良好效能表現至關重要。但是現在就有幾個資料副本（一個在主存中，也許還有幾個在不同快取中的其他副本），而且這些副本是非同步更新的，所以就失去了線性一致性。

為什麼要做這個權衡？對多核記憶體一致性模型而言，CAP 定理是沒有意義的：在同一臺計算機中，我們通常假定通訊都是可靠的。並且我們並不指望一個 CPU 核能在脫離計算機其他部分的條件下繼續正常工作。犧牲線性一致性的原因是效能（performance），而不是容錯。

許多分散式資料庫也是如此：它們是為了提高效能而選擇了犧牲線性一致性，而不是為了容錯【46】。線性一致的速度很慢——這始終是事實，而不僅僅是網路故障期間。

能找到一個更高效的線性一致儲存實現嗎？看起來答案是否定的：Attiya 和 Welch 【47】證明，如果你想要線性一致性，讀寫請求的響應時間至少與網路延遲的不確定性成正比。在像大多數計算機網路一樣具有高度可變延遲的網路中（請參閱“超時與無窮的延遲”），線性讀寫的響應時間不可避免地會很高。更快地線性一致演算法不存在，但更弱的一致性模型可以快得多，所以對延遲敏感的系統而言，這類權衡非常重要。在 第十二章 中將討論一些在不犧牲正確性的前提下，繞開線性一致性的方法。

順序保證

之前說過，線性一致暫存器的行為就好像只有單個數據副本一樣，且每個操作似乎都是在某個時間點以原子性的方式生效的。這個定義意味著操作是按照某種良好定義的順序執行的。我們將操作以看上去被執行的順序連線起來，以此說明了 圖 9-4 中的順序。

順序 (ordering) 這一主題在本書中反覆出現，這表明它可能是一個重要的基礎性概念。讓我們簡要回顧一下其它曾經出現過 順序 的上下文：

- 在 第五章 中我們看到，領導者在單主複製中的主要目的就是，在複製日誌中確定 寫入順序 (order of write) —— 也就是從庫應用這些寫入的順序。如果不存在一個領導者，則併發操作可能導致衝突（請參閱“處理寫入衝突”）。
- 在 第七章 中討論的 可序列化，是關於事務表現的像按 某種先後順序 (some sequential order) 執行的保證。它可以字面意義上地以 序列順序 (serial order) 執行事務來實現，或者允許並行執行，但同時防止序列化衝突來實現（透過鎖或中止事務）。
- 在 第八章 討論過的在分散式系統中使用時間戳和時鐘（請參閱“依賴同步時鐘”）是另一種將順序引入無序世界的嘗試，例如，確定兩個寫入操作哪一個更晚發生。

事實證明，順序、線性一致性和共識之間有著深刻的聯絡。儘管這個概念比本書其他部分更加理論化和抽象，但對於明確系統的能力範圍（可以做什麼和不可以做什麼）而言是非常有幫助的。我們將在接下來的幾節中探討這個話題。

順序與因果關係

順序 反覆出現有幾個原因，其中一個原因是，它有助於保持 因果關係 (causality)。在本書中我們已經看到了幾個例子，其中因果關係是很重要的：

- 在“一致字首讀”（圖 5-5）中，我們看到一個例子：一個對話的觀察者首先看到問題的答案，然後才看到被回答的問題。這是令人困惑的，因為它違背了我們對 因 (cause) 與 果 (effect) 的直覺：如果一個問題被回答，顯然問題本身得先在那裡，因為給出答案的人必須先看到這個問題（假如他們並沒有預見未來的超能力）。我們認為在問題和答案之間存在 因果依賴 (causal dependency)。
- 圖 5-9 中出現了類似的模式，我們看到三位領導者之間的複製，並注意到由於網路延遲，一些寫入可能會“壓倒”其他寫入。從其中一個副本的角度來看，好像有一個對尚不存在的記錄的更新操作。這裡的因果意味著，一條記錄必須先被建立，然後才能被更新。
- 在“檢測併發寫入”中我們觀察到，如果有兩個操作 A 和 B，則存在三種可能性：A 發生在 B 之前，或 B 發生在 A 之前，或者 A 和 B 併發。這種 此前發生 (happened before) 關係是因果關係的另一種表述：如果 A 在 B 前發生，那麼意味著 B 可能已經知道了 A，或者建立在 A 的基礎上，或者依賴於 A。如果 A 和 B 是 併發 的，那麼它們之間並沒有因果聯絡；換句話說，我們確信 A 和 B 不知道彼此。
- 在事務快照隔離的上下文中（“快照隔離和可重複讀”），我們說事務是從一致性快照中讀取的。但此語境中“一致”到底又是什麼意思？這意味著 與因果關係保持一致 (consistent with causality)：如果快照包含答案，它也必須包含被回答的問題【48】。在某個時間點觀察整個資料庫，與因果關係保持一致意味著：因果上在該時間點之前發生的所有操作，其影響都是可見的，但因果上在該時間點之後發生的操作，其影響對觀察者不可見。讀偏差 (read skew) 意味著讀取的資料處於違反因果關係的狀態（不可重複讀，如 圖 7-6 所示）。
- 事務之間 寫偏差 (write skew) 的例子（請參閱“寫入偏差與幻讀”）也說明了因果依賴：在 圖 7-8 中，愛麗絲被允許離班，因為事務認為鮑勃仍在值班，反之亦然。在這種情況下，離班的動作因果依賴於對當前值班情況的觀察。可序列化快照隔離 透過跟蹤事務之間的因果依賴來檢測寫偏差。
- 在愛麗絲和鮑勃看球的例子中（圖 9-1），在聽到愛麗絲驚呼比賽結果後，鮑勃從伺服器得到陳舊結果的事實違背了因果關係：愛麗絲的驚呼因果依賴於得分宣告，所以鮑勃應該也能在聽到愛麗絲驚呼後查詢到比分。相同的模式

在“[跨通道的時序依賴](#)”一節中，以“影象大小調整服務”的偽裝再次出現。

因果關係對事件施加了一種 **順序**：因在果之前；訊息傳送在訊息收取之前。而且就像現實生活中一樣，一件事會導致另一件事：某個節點讀取了一些資料然後寫入一些結果，另一個節點讀取其寫入的內容，並依次寫入一些其他內容，等等。這些因果依賴的操作鏈定義了系統中的因果順序，即，什麼在什麼之前發生。

如果一個系統服從因果關係所規定的順序，我們說它是 **因果一致（causally consistent）** 的。例如，快照隔離提供了因果一致性：當你從資料庫中讀取到一些資料時，你一定還能夠看到其因果前驅（假設在此期間這些資料還沒有被刪除）。

因果順序不是全序的

全序（total order） 允許任意兩個元素進行比較，所以如果有兩個元素，你總是可以說出哪個更大，哪個更小。例如，自然數集是全序的：給定兩個自然數，比如說 5 和 13，那麼你可以告訴我，13 大於 5。

然而數學集合並不完全是全序的： $\{a, b\}$ 比 $\{b, c\}$ 更大嗎？好吧，你沒法真正比較它們，因為二者都不是對方的子集。我們說它們是 **無法比較（incomparable）** 的，因此數學集合是 **偏序（partially order）** 的：在某些情況下，可以說一個集合大於另一個（如果一個集合包含另一個集合的所有元素），但在其他情況下它們是無法比較的。[譯註i](#)

[譯註i](#). 設 R 為非空集合 A 上的關係，如果 R 是自反的、反對稱的和可傳遞的，則稱 R 為 A 上的偏序關係。簡稱偏序，通常記作 \leq 。一個集合 A 與 A 上的偏序關係 R 一起叫作偏序集，記作 (A, R) 或 (A, \leq) 。全序、偏序、關係、集合，這些概念的精確定義可以參考任意一本離散數學教材。 ↩

全序和偏序之間的差異反映在不同的資料庫一致性模型中：

- **線性一致性**

線性一致的系統中，操作是全序的：如果系統表現的就好像只有一個數據副本，並且所有操作都是原子性的，這意味著對任何兩個操作，我們總是能判定哪個操作先發生。這個全序在 [圖 9-4](#) 中以時間線表示。

- **因果性**

我們說過，如果兩個操作都沒有在彼此 **之前發生**，那麼這兩個操作是併發的（請參閱[“此前發生”的關係和併發](#)）。換句話說，如果兩個事件是因果相關的（一個發生在另一個事件之前），則它們之間是有序的，但如果它們是併發的，則它們之間的順序是無法比較的。這意味著因果關係定義了一個偏序，而不是一個全序：一些操作相互之間是有順序的，但有些則是無法比較的。

因此，根據這個定義，線性一致的資料儲存中是不存在併發操作的：必須有且僅有一條時間線，所有的操作都在這條時間線上，構成一個全序關係。可能有幾個請求在等待處理，但是資料儲存確保了每個請求都是在唯一時間線上的某個時間點自動處理的，不存在任何併發。

併發意味著時間線會分岔然後合併——在這種情況下，不同分支上的操作是無法比較的（即併發操作）。在 [第五章](#) 中我們看到了這種現象：例如，[圖 5-14](#) 並不是一條直線的全序關係，而是一堆不同的操作併發進行。圖中的箭頭指明瞭因果依賴——操作的偏序。

如果你熟悉像 Git 這樣的分散式版本控制系統，那麼其版本歷史與因果關係圖極其相似。通常，一個 **提交（Commit）** 發生在另一個提交之後，在一條直線上。但是有時你會遇到分支（當多個人同時在一個專案上工作時），**合併（Merge）** 會在這些併發建立的提交相融合時建立。

線性一致性強於因果一致性

那麼因果順序和線性一致性之間的關係是什麼？答案是線性一致性 **隱含著（implies）** 因果關係：任何線性一致的系統都能正確保持因果性【7】。特別是，如果系統中有多個通訊通道（如 [圖 9-5](#) 中的訊息併列和檔案儲存服務），線性一致性可以自動保證因果性，系統無需任何特殊操作（如在不同元件間傳遞時間戳）。

線性一致性確保因果性的事實使線性一致系統變得簡單易懂，更有吸引力。然而，正如“[線性一致性的代價](#)”中所討論的，使系統線性一致可能會損害其效能和可用性，尤其是在系統具有嚴重的網路延遲的情況下（例如，如果系統在地理上散佈）。出於這個原因，一些分散式資料系統已經放棄了線性一致性，從而獲得更好的效能，但它們用起來也更為困難。

好訊息是存在折衷的可能性。線性一致性並不是保持因果性的唯一途徑——還有其他方法。一個系統可以是因果一致的，而無需承擔線性一致帶來的效能折損（尤其對於 CAP 定理不適用的情況）。實際上在所有的不會被網路延遲拖慢的一致性模型中，因果一致性是可行的最強的一致性模型。而且在網路故障時仍能保持可用 [【2,42】](#)。

在許多情況下，看上去需要線性一致性的系統，實際上需要的只是因果一致性，因果一致性可以更高效地實現。基於這種觀察結果，研究人員正在探索新型的資料庫，既能保證因果一致性，且效能與可用性與最終一致的系統類似 [【49,50,51】](#)。

這方面的研究相當新鮮，其中很多尚未應用到生產系統，仍然有不少挑戰需要克服 [【52,53】](#)。但對於未來的系統而言，這是一個有前景的方向。

捕獲因果關係

我們不會在這裡討論非線性一致的系統如何保證因果性的細節，而只是簡要地探討一些關鍵的思想。

為了維持因果性，你需要知道哪個操作發生在哪個其他操作之前 (**happened before**)。這是一個偏序：併發操作可以以任意順序進行，但如果一個操作發生在另一個操作之前，那它們必須在所有副本上以那個順序被處理。因此，當一個副本處理一個操作時，它必須確保所有因果前驅的操作（之前發生的所有操作）已經被處理；如果前面的某個操作丟失了，後面的操作必須等待，直到前面的操作被處理完畢。

為了確定因果依賴，我們需要一些方法來描述系統中節點的“知識”。如果節點在發出寫入 Y 的請求時已經看到了 X 的值，則 X 和 Y 可能存在因果關係。這個分析使用了那些在欺詐指控刑事調查中常見的問題：CEO 在做出決定 Y 時是否知道 X？

用於確定哪些操作發生在其他操作之前的技術，與我們在“[檢測併發寫入](#)”中所討論的內容類似。那一節討論了無領導者資料儲存中的因果性：為了防止丟失更新，我們需要檢測到對同一個鍵的併發寫入。因果一致性則更進一步：它需要跟蹤整個資料庫中的因果依賴，而不僅僅是一個鍵。可以推廣版本向量以解決此類問題 [【54】](#)。

為了確定因果順序，資料庫需要知道應用讀取了哪個版本的資料。這就是為什麼在 [圖 5-13](#) 中，來自先前操作的版本號在寫入時被傳回到資料庫的原因。在 SSI 的衝突檢測中會出現類似的想法，如“[可序列化快照隔離](#)”中所述：當事務要提交時，資料庫將檢查它所讀取的資料版本是否仍然是最新的。為此，資料庫跟蹤哪些資料被哪些事務所讀取。

序列號順序

雖然因果是一個重要的理論概念，但實際上跟蹤所有的因果關係是不切實際的。在許多應用中，客戶端在寫入內容之前會先讀取大量資料，我們無法弄清寫入因果依賴於先前全部的讀取內容，還是僅包括其中一部分。顯式跟蹤所有已讀資料意味著巨大的額外開銷。

但還有一個更好的方法：我們可以使用 **序列號 (sequence number)** 或 **時間戳 (timestamp)** 來排序事件。時間戳不一定來自日曆時鐘（或物理時鐘，它們存在許多問題，如“[不可靠的時鐘](#)”中所述）。它可以來自一個 **邏輯時鐘 (logical clock)**，這是一個用來生成標識操作的數字序列的演算法，典型實現是使用一個每次操作自增的計數器。

這樣的序列號或時間戳是緊湊的（只有幾個位元組大小），它提供了一個全序關係：也就是說每個操作都有一個唯一的序列號，而且總是可以比較兩個序列號，確定哪一個更大（即哪些操作後發生）。

特別是，我們可以使用 **與因果一致 (consistent with causality)** 的全序來生成序列號 ^{vii}：我們保證，如果操作 A 因果地發生在操作 B 前，那麼在這個全序中 A 在 B 前（A 具有比 B 更小的序列號）。並行操作之間可以任意排序。這樣一個全序關係捕獲了所有關於因果的資訊，但也施加了一個比因果性要求更為嚴格的順序。

^{vii} 與因果關係不一致的全序很容易建立，但沒啥用。例如你可以為每個操作生成隨機的 UUID，並按照字典序比較 UUID，以定義操作的全序。這是一個有效的全序，但是隨機的 UUID 並不能告訴你哪個操作先發生，或者操

作是否為併發的。 ↪

在單主複製的資料庫中（請參閱“[領導者與追隨者](#)”），複製日誌定義了與因果一致的寫操作。主庫可以簡單地為每個操作自增一個計數器，從而為複製日誌中的每個操作分配一個單調遞增的序列號。如果一個從庫按照它們在複製日誌中出現的順序來應用寫操作，那麼從庫的狀態始終是因果一致的（即使它落後於領導者）。

非因果序列號生成器

如果主庫不存在（可能因為使用了多主資料庫或無主資料庫，或者因為使用了分割槽的資料庫），如何為操作生成序列號就沒有那麼明顯了。在實踐中有各種各樣的方法：

- 每個節點都可以生成自己獨立的一組序列號。例如有兩個節點，一個節點只能生成奇數，而另一個節點只能生成偶數。通常，可以在序列號的二進位制表示中預留一些位，用於唯一的節點識別符號，這樣可以確保兩個不同的節點永遠不會生成相同的序列號。可以將日曆時鐘（物理時鐘）的時間戳附加到每個操作上【55】。這種時間戳並不連續，但是如果它具有足夠高的解析度，那也許足以提供一個操作的全序關係。這一事實應用於最後寫入勝利*的衝突解決方法中（請參閱[有序事件的時間戳](#)）。
- 可以預先分配序列號區塊。例如，節點 A 可能要求從序列號 1 到 1,000 區塊的所有權，而節點 B 可能要求序列號 1,001 到 2,000 區塊的所有權。然後每個節點可以獨立分配所屬區塊中的序列號，並在序列號告急時請求分配一個新的區塊。

這三個選項都比單一主庫的自增計數器表現要好，並且更具可伸縮性。它們為每個操作生成一個唯一的，近似自增的序列號。然而它們都有同一個問題：生成的序列號與因果不一致。

因為這些序列號生成器不能正確地捕獲跨節點的操作順序，所以會出現因果關係的問題：

- 每個節點每秒可以處理不同數量的操作。因此，如果一個節點產生偶數序列號而另一個產生奇數序列號，則偶數計數器可能落後於奇數計數器，反之亦然。如果你有一個奇數編號的操作和一個偶數編號的操作，你無法準確地說出哪一個操作在因果上先發生。
- 來自物理時鐘的時間戳會受到時鐘偏移的影響，這可能會使其與因果不一致。例如 圖 8-3 展示了一個例子，其中因果上晚發生的操作，卻被分配了一個更早的時間戳。^{vii}

^{viii}. 可以使物理時鐘時間戳與因果關係保持一致：在“[全域性快照的同步時鐘](#)”中，我們討論了 Google 的 Spanner，它可以估計預期的時鐘偏差，並在提交寫入之前等待不確定性間隔。這種方法確保了實際上靠後的事務會有更大的時間戳。但是大多數時鐘不能提供這種所需的不確定性度量。 ↪

- 在分配區塊的情況下，某個操作可能會被賦予一個範圍在 1,001 到 2,000 內的序列號，然而一個因果上更晚的操作可能被賦予一個範圍在 1 到 1,000 之間的數字。這裡序列號與因果關係也是不一致的。

蘭伯特時間戳

儘管剛才描述的三個序列號生成器與因果不一致，但實際上有一個簡單的方法來產生與因果關係一致的序列號。它被稱為蘭伯特時間戳，萊斯利·蘭伯特（Leslie Lamport）於 1978 年提出【56】，現在是分散式系統領域中被引用最多的論文之一。

圖 9-8 說明了蘭伯特時間戳的應用。每個節點都有一個唯一識別符號，和一個儲存自己執行運算元量的計數器。蘭伯特時間戳就是兩者的簡單組合：（計數器，節點 ID）\$(counter, node ID)\$。兩個節點有時可能具有相同的計數器值，但透過在時間戳中包含節點 ID，每個時間戳都是唯一的。

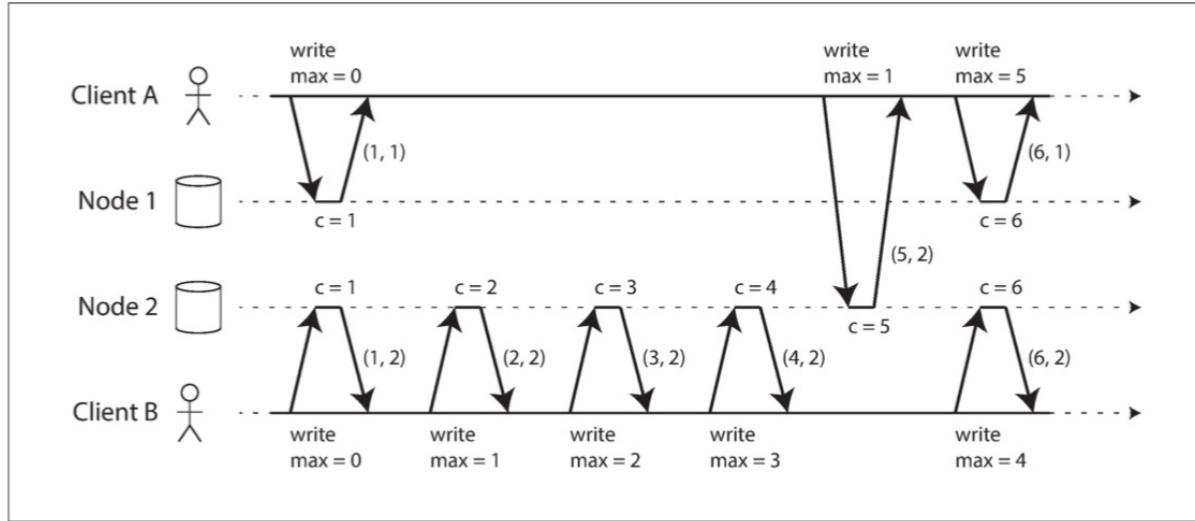


圖 9-8 Lamport 時間戳提供了與因果關係一致的全序。

蘭伯特時間戳與物理的日曆時鐘沒有任何關係，但是它提供了一個全序：如果你有兩個時間戳，則 計數器 值大者是更大的時間戳。如果計數器值相同，則節點 ID 越大的，時間戳越大。

迄今，這個描述與上節所述的奇偶計數器基本類似。使蘭伯特時間戳因果一致的關鍵思想如下所示：每個節點和每個客戶端跟蹤迄今為止所見到的最大 計數器 值，並在每個請求中包含這個最大計數器值。當一個節點收到最大計數器值大於自身計數器值的請求或響應時，它立即將自己的計數器設定為這個最大值。

這如 圖 9-8 所示，其中客戶端 A 從節點 2 接收計數器值 5，然後將最大值 5 傳送到節點 1。此時，節點 1 的計數器僅為 1，但是它立即前移至 5，所以下一個操作的計數器的值為 6。

只要每一個操作都攜帶著最大計數器值，這個方案確保蘭伯特時間戳的排序與因果一致，因為每個因果依賴都會導致時間戳增長。

蘭伯特時間戳有時會與我們在“[檢測併發寫入](#)”中看到的版本向量相混淆。雖然兩者有一些相似之處，但它們有著不同的目的：版本向量可以區分兩個操作是併發的，還是一個因果依賴另一個；而蘭伯特時間戳總是施行一個全序。從蘭伯特時間戳的全序中，你無法分辨兩個操作是併發的還是因果依賴的。蘭伯特時間戳優於版本向量的地方是，它更加緊湊。

光有時間戳排序還不夠

雖然蘭伯特時間戳定義了一個與因果一致的全序，但它還不足以解決分散式系統中的許多常見問題。

例如，考慮一個需要確保使用者名稱能唯一標識使用者帳戶的系統。如果兩個使用者同時嘗試使用相同的使用者名稱建立帳戶，則其中一個應該成功，另一個應該失敗（我們之前在“[領導者和鎖](#)”中提到過這個問題）。

乍看之下，似乎操作的全序關係足以解決這一問題（例如使用蘭伯特時間戳）：如果建立了兩個具有相同使用者名稱的帳戶，選擇時間戳較小的那個作為勝者（第一個抓到使用者名稱的人），並讓帶有更大時間戳者失敗。由於時間戳上有全序關係，所以這個比較總是可行的。

這種方法適用於事後確定勝利者：一旦你收集了系統中的所有使用者名稱建立操作，就可以比較它們的時間戳。然而當某個節點需要實時處理使用者建立使用者名稱的請求時，這樣的方法就無法滿足了。節點需要 馬上 (**right now**) 決定這個請求是成功還是失敗。在那個時刻，節點並不知道是否存在其他節點正在併發執行建立同樣使用者名稱的操作，罔論其它節點可能分配給那個操作的時間戳。

為了確保沒有其他節點正在使用相同的使用者名稱和較小的時間戳併發建立同名帳戶，你必須檢查其它每個節點，看看它在做什麼【56】。如果其中一個節點由於網路問題出現故障或不可達，則整個系統可能被拖至停機。這不是我們需要的那種容錯系統。

這裡的問題是，只有在所有的操作都被收集之後，操作的全序才會出現。如果另一個節點已經產生了一些操作，但你還不知道那些操作是什麼，那就無法構造所有操作最終的全序關係：來自另一個節點的未知操作可能需要被插入到全序中的不同位置。

總之：為了實現諸如使用者名稱上的唯一約束這種東西，僅有操作的全序是不夠的，你還需要知道這個全序何時會塵埃落定。如果你有一個建立使用者名稱的操作，並且確定在全序中沒有任何其他節點可以在你的操作之前插入對同一使用者名稱的聲稱，那麼你就可以安全地宣告操作執行成功。

如何確定全序關係已經塵埃落定，這將在 [全序廣播](#) 一節中詳細說明。

全序廣播

如果你的程式只執行在單個 CPU 核上，那麼定義一個操作全序是很容易的：可以簡單認為就是 CPU 執行這些操作的順序。但是在分散式系統中，讓所有節點對同一個全域性操作順序達成一致可能相當棘手。在上一節中，我們討論了按時間戳或序列號進行排序，但發現它還不如單主複製給力（如果你使用時間戳排序來實現唯一性約束，就不能容忍任何錯誤，因為你必須要從每個節點都獲取到最新的序列號）。

如前所述，單主複製透過選擇一個節點作為主庫來確定操作的全序，並在主庫的單個 CPU 核上對所有操作進行排序。接下來的挑戰是，如果吞吐量超出單個主庫的處理能力，這種情況下如何擴充套件系統；以及，如果主庫失效（“[處理節點宕機](#)”），如何處理故障切換。在分散式系統文獻中，這個問題被稱為 **全序廣播** (**total order broadcast**) 或 **原子廣播** (**atomic broadcast**)^{ix} [【25,57,58】](#)。

^{ix}. “原子廣播”是一個傳統的術語，非常混亂，而且與“原子”一詞的其他用法不一致：它與 ACID 事務中的原子性沒有任何關係，只是與原子操作（在多執行緒程式設計的意義上）或原子暫存器（線性一致儲存）有間接的聯絡。全序組播（total order multicast）是另一個同義詞。 ↪

順序保證的範圍

每個分割槽各有一個主庫的分割槽資料庫，通常只在每個分割槽內維持順序，這意味著它們不能提供跨分割槽的一致性保證（例如，一致性快照，外來鍵引用）。跨所有分割槽的全序是可能的，但需要額外的協調 [【59】](#)。

全序廣播通常被描述為在節點間交換訊息的協議。非正式地講，它要滿足兩個安全屬性：

- 可靠交付 (reliable delivery)

沒有訊息丟失：如果訊息被傳遞到一個節點，它將被傳遞到所有節點。

- 全序交付 (totally ordered delivery)

訊息以相同的順序傳遞給每個節點。

正確的全序廣播演算法必須始終保證可靠性和有序性，即使節點或網路出現故障。當然在網路中斷的時候，訊息是傳不出去的，但是演算法可以不斷重試，以便在網路最終修復時，訊息能及時透過並送達（當然它們必須仍然按照正確的順序傳遞）。

使用全序廣播

像 ZooKeeper 和 etcd 這樣的共識服務實際上實現了全序廣播。這一事實暗示了全序廣播與共識之間有著緊密聯絡，我們將在本章稍後進行探討。

全序廣播正是資料庫複製所需的：如果每個訊息都代表一次資料庫的寫入，且每個副本都按相同的順序處理相同的寫入，那麼副本間將相互保持一致（除了臨時的複製延遲）。這個原理被稱為 **狀態機複製** (**state machine replication**) [【60】](#)，我們將在 [第十一章](#) 中重新回到這個概念。

與之類似，可以使用全序廣播來實現可序列化的事務：如 “[真的序列執行](#)” 中所述，如果每個訊息都表示一個確定性事務，以儲存過程的形式來執行，且每個節點都以相同的順序處理這些訊息，那麼資料庫的分割槽和副本就可以相互保持一致 [【61】](#)。

全序廣播的一個重要表現是，順序在訊息送達時被固化：如果後續的訊息已經送達，節點就不允許追溯地將（先前）訊息插入順序中的較早位置。這個事實使得全序廣播比時間戳排序更強。

考量全序廣播的另一種方式是，這是一種建立日誌的方式（如在複製日誌、事務日誌或預寫式日誌中）：傳遞訊息就像追加寫入日誌。由於所有節點必須以相同的順序傳遞相同的訊息，因此所有節點都可以讀取日誌，並看到相同的訊息序列。

全序廣播對於實現提供防護令牌的鎖服務也很有用（請參閱“[防護令牌](#)”）。每個獲取鎖的請求都作為一條訊息追加到日誌末尾，並且所有的訊息都按它們在日誌中出現的順序依次編號。序列號可以當成防護令牌用，因為它是單調遞增的。在 ZooKeeper 中，這個序列號被稱為 `zxid` [【15】](#)。

使用全序廣播實現線性一致的儲存

如 [圖 9-4](#) 所示，線性一致的系統中，存在操作的全序。這是否意味著線性一致與全序廣播一樣？不盡然，但兩者之間有著密切的聯絡 [X](#)。

[X](#). 從形式上講，線性一致讀寫暫存器是一個“更容易”的問題。全序廣播等價於共識 [【67】](#)，而共識問題在非同步的崩潰 - 停止模型 [【68】](#) 中沒有確定性的解決方案，而線性一致的讀寫暫存器可以在這種模型中實現 [【23,24,25】](#)。然而，支援諸如 比較並設定（CAS, compare-and-set），或 自增並返回（increment-and-get）的原子操作使它等價於共識問題 [【28】](#)。因此，共識問題與線性一致暫存器問題密切相關。[←](#)

全序廣播是非同步的：訊息被保證以固定的順序可靠地傳送，但是不能保證訊息 **何時** 被送達（所以一個接收者可能落後於其他接收者）。相比之下，線性一致性是新鮮性的保證：讀取一定能看見最新的寫入值。

但如果有了全序廣播，你就可以在此基礎上構建線性一致的儲存。例如，你可以確保使用者名稱能唯一標識使用者帳戶。

設想對於每一個可能的使用者名稱，你都可以有一個帶有 CAS 原子操作的線性一致暫存器。每個暫存器最初的值為空值（表示未使用該使用者名稱）。當用戶想要建立一個使用者名稱時，對該使用者名稱的暫存器執行 CAS 操作，在先前暫存器值為空的條件，將其值設定為使用者的賬號 ID。如果多個使用者試圖同時獲取相同的使用者名稱，則只有一個 CAS 操作會成功，因為其他使用者會看到非空的值（由於線性一致性）。

你可以透過將全序廣播當成僅追加日誌 [【62,63】](#) 的方式來實現這種線性一致的 CAS 操作：

1. 在日誌中追加一條訊息，試探性地指明你要宣告的使用者名稱。
2. 讀日誌，並等待你剛才追加的訊息被讀回。[xi](#)
3. 檢查是否有任何訊息聲稱目標使用者名稱的所有權。如果這些訊息中的第一條就是你自己的訊息，那麼你就成功了：你可以提交聲稱的使用者名稱（也許是透過向日志追加另一條訊息）並向客戶端確認。如果所需使用者名稱的第一條訊息來自其他使用者，則中止操作。

[xi](#). 如果你不等待，而是在訊息入隊之後立即確認寫入，則會得到類似於多核 x86 處理器記憶體的一致性模型 [【43】](#)。該模型既不是線性一致的也不是順序一致的。[←](#)

由於日誌項是以相同順序送達至所有節點，因此如果有多個併發寫入，則所有節點會對最先到達者達成一致。選擇衝突寫入中的第一個作為勝利者，並中止後來者，以此確定所有節點對某個寫入是提交還是中止達成一致。類似的方法可以在一個日誌的基礎上實現可序列化的多物件事務 [【62】](#)。

儘管這一過程保證寫入是線性一致的，但它並不保證讀取也是線性一致的——如果你從與日誌非同步更新的儲存中讀取資料，結果可能是陳舊的。（精確地說，這裡描述的過程提供了**順序一致性**（sequential consistency）

[【47,64】](#)，有時也稱為**時間線一致性**（timeline consistency）[【65,66】](#)，比線性一致性稍微弱一些的保證）。為了使讀取也線性一致，有幾個選項：

- 你可以透過在日誌中追加一條訊息，然後讀取日誌，直到該訊息被讀回才執行實際的讀取操作。訊息在日誌中的位置因此定義了讀取發生的時間點（etcd 的法定人數讀取有些類似這種情況 [【16】](#)）。
- 如果日誌允許以線性一致的方式獲取最新日誌訊息的位置，則可以查詢該位置，等待該位置前的所有訊息都傳達到你，然後執行讀取。（這是 Zookeeper `sync()` 操作背後的思想 [【15】](#)）。
- 你可以從同步更新的副本中進行讀取，因此可以確保結果是最新的（這種技術用於鏈式複製（chain replication）

【63】；請參閱“關於複製的研究”）。

使用線性一致性儲存實現全序廣播

上一節介紹瞭如何從全序廣播構建一個線性一致的 CAS 操作。我們也可以把它反過來，假設我們有線性一致的儲存，接下來會展示如何在此基礎上構建全序廣播。

最簡單的方法是假設你有一個線性一致的暫存器來儲存一個整數，並且有一個原子 **自增並返回** 操作【28】。或者原子 CAS 操作也可以完成這項工作。

該演算法很簡單：每個要透過全序廣播發送的訊息首先對線性一致暫存器執行 **自增並返回** 操作。然後將從暫存器獲得的值作為序列號附加到訊息中。然後你可以將訊息傳送到所有節點（重新發送任何丟失的訊息），而收件人將按序列號依序傳遞（deliver）訊息。

請注意，與蘭伯特時間戳不同，透過自增線性一致性暫存器獲得的數字形式上是一個沒有間隙的序列。因此，如果一個節點已經發送了訊息 4 並且接收到序列號為 6 的傳入訊息，則它知道它在傳遞訊息 6 之前必須等待訊息 5。蘭伯特時間戳則與之不同——事實上，這是全序廣播和時間戳排序間的關鍵區別。

實現一個帶有原子性 **自增並返回** 操作的線性一致暫存器有多困難？像往常一樣，如果事情從來不出差錯，那很容易：你可以簡單地把它儲存在單個節點內的變數中。問題在於處理當該節點的網路連線中斷時的情況，並在該節點失效時能恢復這個值【59】。一般來說，如果你對線性一致性的序列號生成器進行過足夠深入的思考，你不可避免地會得出一個共識演算法。

這並非巧合：可以證明，線性一致的 CAS（或自增並返回）暫存器與全序廣播都等價於 **共識** 問題【28,67】。也就是說，如果你能解決其中的一個問題，你可以把它轉化成為其他問題的解決方案。這是相當深刻和令人驚訝的洞察！

現在是時候正面處理共識問題了，我們將在本章的其餘部分進行討論。

分散式事務與共識

共識 是分散式計算中最重要也是最基本的問題之一。從表面上看似乎很簡單：非正式地講，目標只是 **讓幾個節點達成一致** (*get several nodes to agree on something*)。你也許會認為這不會太難。不幸的是，許多出故障的系統都是因為錯誤地輕信這個問題很容易解決。

儘管共識非常重要，但關於它的內容出現在本書的後半部分，因為這個主題非常微妙，欣賞細微之處需要一些必要的知識。即使在學術界，對共識的理解也是在幾十年的過程中逐漸沉澱而來，一路上也有著許多誤解。現在我們已經討論了複製（第五章），事務（第七章），系統模型（第八章），線性一致以及全序廣播（本章），我們終於準備好解決共識問題了。

節點能達成一致，在很多場景下都非常重要，例如：

- 領導選舉

在單主複製的資料庫中，所有節點需要就哪個節點是領導者達成一致。如果一些節點由於網路故障而無法與其他節點通訊，則可能會對領導權的歸屬引起爭議。在這種情況下，共識對於避免錯誤的故障切換非常重要。錯誤的故障切換會導致兩個節點都認為自己是領導者（腦裂，請參閱“處理節點宕機”）。如果有兩個領導者，它們都會接受寫入，它們的資料會發生分歧，從而導致不一致和資料丟失。

- 原子提交

在支援跨多節點或跨多分割槽事務的資料庫中，一個事務可能在某些節點上失敗，但在其他節點上成功。如果我們想要維護事務的原子性（就 ACID 而言，請參閱“**原子性**”），我們必須讓所有節點對事務的結果達成一致：要麼全部中止 / 回滾（如果出現任何錯誤），要麼它們全部提交（如果沒有出錯）。這個共識的例子被稱為 **原子提交** (**atomic commit**) 問題^{xii}。

^{xii}. 原子提交的形式化與共識稍有不同：原子事務只有在所有參與者投票提交的情況下才能提交，如果有任何參與者需要中止，則必須中止。共識則允許就任意一個被參與者提出的候選值達成一致。然而，原子提

交和共識可以相互簡化為對方【70,71】。非阻塞原子提交則要比共識更為困難——請參閱“三階段提交”。[←](#)

共識的不可能性

你可能已經聽說過以作者 Fischer, Lynch 和 Paterson 命名的 FLP 結果【68】，它證明，如果存在節點可能崩潰的風險，則不存在總是能夠達成共識的演算法。在分散式系統中，我們必須假設節點可能會崩潰，所以可靠的共識是不可能的。然而這裡我們正在討論達成共識的演算法，到底是怎麼回事？

答案是 FLP 結果是在 **非同步系統模型** 中被證明的（請參閱“[系統模型與現實](#)”），而這是一種限制性很強的模型，它假定確定性演算法不能使用任何時鐘或超時。如果允許演算法使用 **超時** 或其他方法來識別可疑的崩潰節點（即使懷疑有時是錯誤的），則共識變為一個可解的問題【67】。即使僅僅允許演算法使用隨機數，也足以繞過這個不可能的結果【69】。

因此，雖然 FLP 是關於共識不可能性的重要理論結果，但現實中的分散式系統通常是可以達成共識的。

在本節中，我們將首先更詳細地研究 **原子提交** 問題。具體來說，我們將討論 **兩階段提交 (2PC, two-phase commit)** 演算法，這是解決原子提交問題最常見的辦法，並在各種資料庫、訊息佇列和應用伺服器中被實現。事實證明 2PC 是一種共識演算法，但不是一個非常好的共識演算法【70,71】。

透過對 2PC 的學習，我們將繼續努力實現更好的一致性演算法，比如 ZooKeeper (Zab) 和 etcd (Raft) 中使用的演算法。

原子提交與兩階段提交

在 [第七章](#) 中我們瞭解到，事務原子性的目的是在多次寫操作中途出錯的情況下，提供一種簡單的語義。事務的結果要麼是成功提交，在這種情況下，事務的所有寫入都是持久化的；要麼是中止，在這種情況下，事務的所有寫入都被回滾（即撤消或丟棄）。

原子性可以防止失敗的事務攪亂資料庫，避免資料庫陷入半成品結果和半更新狀態。這對於多物件事務（請參閱“[單物件和多物件操作](#)”）和維護次級索引的資料庫尤其重要。每個次級索引都是與主資料相分離的資料結構——因此，如果你修改了一些資料，則還需要在次級索引中進行相應的更改。原子性確保次級索引與主資料保持一致（如果索引與主資料不一致，就沒什麼用了）。

從單節點到分散式原子提交

對於在單個數據庫節點執行的事務，原子性通常由儲存引擎實現。當客戶端請求資料庫節點提交事務時，資料庫將使事務的寫入持久化（通常在預寫式日誌中，請參閱“[讓 B 樹更可靠](#)”），然後將提交記錄追加到磁碟中的日誌裡。如果資料庫在這個過程中間崩潰，當節點重啟時，事務會從日誌中恢復：如果提交記錄在崩潰之前成功地寫入磁碟，則認為事務被提交；否則來自該事務的任何寫入都被回滾。

因此，在單個節點上，事務的提交主要取決於資料持久化落盤的 **順序**：首先是資料，然後是提交記錄【72】。事務提交或終止的關鍵決定時刻是磁碟完成寫入提交記錄的時刻：在此之前，仍有可能中止（由於崩潰），但在此之後，事務已經提交（即使資料庫崩潰）。因此，是單一的裝置（連線到單個磁碟的控制器，且掛載在單臺機器上）使得提交具有原子性。

但是，如果一個事務中涉及多個節點呢？例如，你也許在分割槽資料庫中會有一個多物件事務，或者是一個按關鍵詞分分割槽的次級索引（其中索引條目可能位於與主資料不同的節點上；請參閱“[分割槽與次級索引](#)”）。大多數 “NoSQL” 分散式資料儲存不支援這種分散式事務，但是很多關係型資料庫叢集支援（請參閱“[實踐中的分散式事務](#)”）。

在這些情況下，僅向所有節點發送提交請求並獨立提交每個節點的事務是不夠的。這樣很容易發生違反原子性的情況：提交在某些節點上成功，而在其他節點上失敗：

- 某些節點可能會檢測到違反約束或衝突，因此需要中止，而其他節點則可以成功進行提交。
- 某些提交請求可能在網路中丟失，最終由於超時而中止，而其他提交請求則透過。
- 在提交記錄完全寫入之前，某些節點可能會崩潰，並在恢復時回滾，而其他節點則成功提交。

如果某些節點提交了事務，但其他節點卻放棄了這些事務，那麼這些節點就會彼此不一致（如 圖 7-3 所示）。而且一旦在某個節點上提交了一個事務，如果事後發現它在其它節點上被中止了，它是無法撤回的。出於這個原因，一旦確定事務中的所有其他節點也將提交，節點就必須進行提交。

事務提交必須是不可撤銷的——事務提交之後，你不能改變主意，並追溯性地中止事務。這個規則的原因是，一旦資料被提交，其結果就對其他事務可見，因此其他客戶端可能會開始依賴這些資料。這個原則構成了 **讀已提交** 隔離等級的基礎，在“**讀已提交**”一節中討論了這個問題。如果一個事務在提交後被允許中止，所有那些讀取了 **已提交卻又被追溯宣告不存在資料** 的事務也必須回滾。

（提交事務的結果有可能透過事後執行另一個補償事務（compensating transaction）來取消【73,74】，但從資料庫的角度來看，這是一個單獨的事務，因此任何關於跨事務正確性的保證都是應用自己的問題。）

兩階段提交簡介

兩階段提交 (two-phase commit) 是一種用於實現跨多個節點的原子事務提交的演算法，即確保所有節點提交或所有節點中止。它是分散式資料庫中的經典演算法【13,35,75】。2PC 在某些資料庫內部使用，也以 **XA 事務** 的形式對應用可用【76,77】（例如 Java Transaction API 支援）或以 SOAP Web 服務的 **WS-AtomicTransaction** 形式提供給應用【78,79】。

圖 9-9 說明了 2PC 的基本流程。2PC 中的提交 / 中止過程分為兩個階段（因此而得名），而不是單節點事務中的單個提交請求。

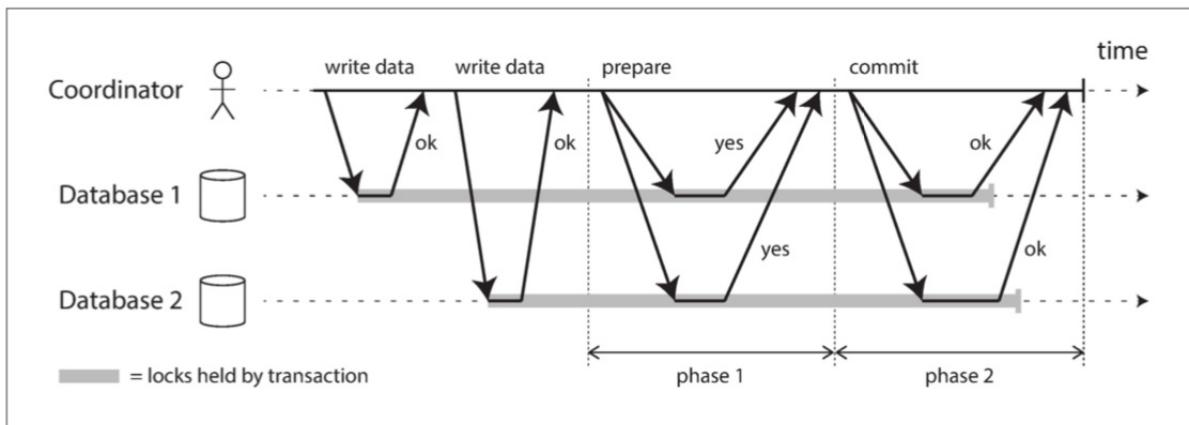


圖 9-9 兩階段提交 (2PC) 的成功執行

不要把2PC和2PL搞混了

兩階段提交 (2PC) 和兩階段鎖定（請參閱“**兩階段鎖定**”）是兩個完全不同的東西。2PC 在分散式資料庫中提供原子提交，而 2PL 提供可序列化的隔離等級。為了避免混淆，最好把它們看作完全獨立的概念，並忽略名稱中不幸的相似性。

2PC 使用一個通常不會出現在單節點事務中的新元件：協調者（coordinator，也稱為 **事務管理器**，即 **transaction manager**）。協調者通常在請求事務的相同應用程序中以庫的形式實現（例如，嵌入在 Java EE 容器中），但也可以是單獨的程序或服務。這種協調者的例子包括 Narayana、JOTM、BTM 或 MSDTC。

正常情況下，2PC 事務以應用在多個數據庫節點上讀寫資料開始。我們稱這些資料庫節點為 **參與者 (participants)**。當應用準備提交時，協調者開始階段 1：它傳送一個 **準備 (prepare)** 請求到每個節點，詢問它們是否能夠提交。然後協調者會跟蹤參與者的響應：

- 如果所有參與者都回答“是”，表示它們已經準備好提交，那麼協調者在階段 2 發出 **提交 (commit)** 請求，然後提交真正發生。
- 如果任意一個參與者回覆了“否”，則協調者在階段 2 中向所有節點發送 **中止 (abort)** 請求。

這個過程有點像西方傳統婚姻儀式：司儀分別詢問新娘和新郎是否要結婚，通常是從兩方都收到“我願意”的答覆。收到兩者的回覆後，司儀宣佈這對情侶成為夫妻：事務就提交了，這一幸福事實會廣播至所有的參與者中。如果新娘與新郎之一沒有回覆“我願意”，婚禮就會中止【73】。

系統承諾

這個簡短的描述可能並沒有說清楚為什麼兩階段提交保證了原子性，而跨多個節點的一階段提交卻沒有。在兩階段提交的情況下，準備請求和提交請求當然也可以輕易丟失。2PC 又有什麼不同呢？

為了理解它的工作原理，我們必須更詳細地分解這個過程：

1. 當應用想要啟動一個分散式事務時，它向協調者請求一個事務 ID。此事務 ID 是全域性唯一的。
2. 應用在每個參與者上啟動單節點事務，並在單節點事務上捎帶上這個全域性事務 ID。所有的讀寫都是在這些單節點事務中各自完成的。如果在這個階段出現任何問題（例如，節點崩潰或請求超時），則協調者或任何參與者都可以中止。
3. 當應用準備提交時，協調者向所有參與者傳送一個 **準備** 請求，並打上全域性事務 ID 的標記。如果任意一個請求失敗或超時，則協調者向所有參與者傳送針對該事務 ID 的中止請求。
4. 參與者收到準備請求時，需要確保在任意情況下都的確可以提交事務。這包括將所有事務資料寫入磁碟（出現崩潰、電源故障或硬碟空間不足都不能是稍後拒絕提交的理由）以及檢查是否存在任何衝突或違反約束。透過向協調者回答“是”，節點承諾，只要請求，這個事務一定可以不出差錯地提交。換句話說，參與者放棄了中止事務的權利，但沒有實際提交。
5. 當協調者收到所有準備請求的答覆時，會就提交或中止事務作出明確的決定（只有在所有參與者投贊成票的情況下才會提交）。協調者必須把這個決定寫到磁碟上的事務日誌中，如果它隨後就崩潰，恢復後也能知道自己所做的決定。這被稱為 **提交點** (**commit point**)。
6. 一旦協調者的決定落盤，提交或中止請求會發送給所有參與者。如果這個請求失敗或超時，協調者必須永遠保持重試，直到成功為止。沒有回頭路：如果已經做出決定，不管需要多少次重試它都必須被執行。如果參與者在此期間崩潰，事務將在其恢復後提交——由於參與者投了贊成，因此恢復後它不能拒絕提交。

因此，該協議包含兩個關鍵的“不歸路”點：當參與者投票“是”時，它承諾它稍後肯定能夠提交（儘管協調者可能仍然選擇放棄）；以及一旦協調者做出決定，這一決定是不可撤銷的。這些承諾保證了 2PC 的原子性（單節點原子提交將這兩個事件合為一體：將提交記錄寫入事務日誌）。

回到婚姻的比喻，在說“我願意”之前，你和你的新娘 / 新郎有中止這個事務的自由，只要回覆“沒門！”就行（或者有類似效果的話）。然而在說了“我願意”之後，你就不能撤回那個聲明瞭。如果你說“我願意”後暈倒了，沒有聽到司儀說“你們現在是夫妻了”，那也並不會改變事務已經提交的現實。當你稍後恢復意識時，可以透過查詢司儀的全域性事務 ID 狀態來確定你是否已經成婚，或者你可以等待司儀重試下一次提交請求（因為重試將在你無意識期間一直持續）。

協調者失效

我們已經討論了在 2PC 期間，如果參與者之一或網路發生故障時會發生什麼情況：如果任何一個 **準備** 請求失敗或者超時，協調者就會中止事務。如果任何提交或中止請求失敗，協調者將無條件重試。但是如果協調者崩潰，會發生什麼情況就不太清楚了。

如果協調者在傳送 **準備** 請求之前失敗，參與者可以安全地中止事務。但是，一旦參與者收到了準備請求並投了“是”，就不能再單方面放棄——必須等待協調者回答事務是否已經提交或中止。如果此時協調者崩潰或網路出現故障，參與者什麼也做不了只能等待。參與者的這種事務狀態稱為 **存疑** (**in doubt**) 的或 **不確定** (**uncertain**) 的。

情況如 [圖 9-10](#) 所示。在這個特定的例子中，協調者實際上決定提交，資料庫 2 收到提交請求。但是，協調者在將提交請求傳送到資料庫 1 之前發生崩潰，因此資料庫 1 不知道是否提交或中止。即使 **超時** 在這裡也沒有幫助：如果資料庫 1 在超時後單方面中止，它將最終與執行提交的資料庫 2 不一致。同樣，單方面提交也是不安全的，因為另一個參與者可能已經中止了。

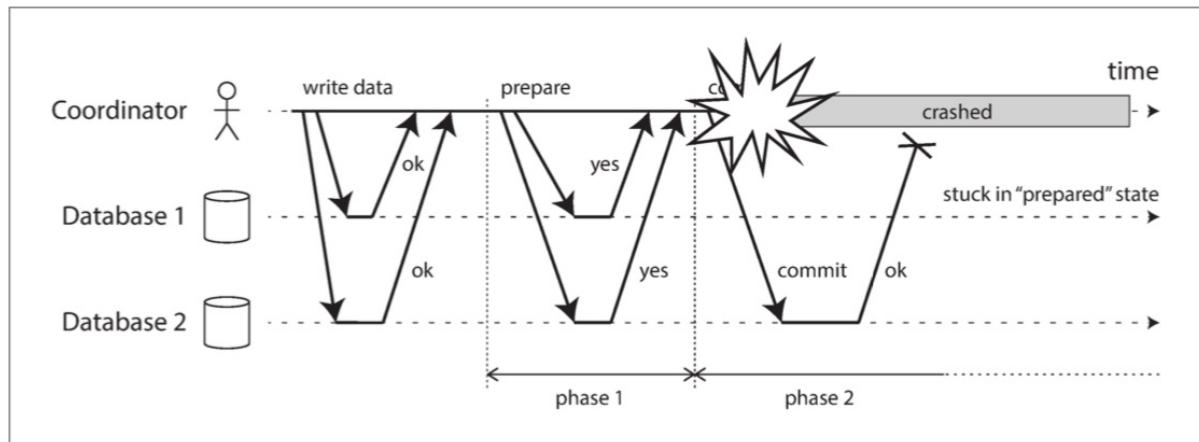


圖 9-10 參與者投贊成票後，協調者崩潰。資料庫 1 不知道是否提交或中止

沒有協調者的訊息，參與者無法知道是提交還是放棄。原則上參與者可以相互溝通，找出每個參與者是如何投票的，並達成一致，但這不是 2PC 協議的一部分。

可以完成 2PC 的唯一方法是等待協調者恢復。這就是為什麼協調者必須在向參與者傳送提交或中止請求之前，將其提交或中止決定寫入磁碟上的事務日誌：協調者恢復後，透過讀取其事務日誌來確定所有存疑事務的狀態。任何在協調者日誌中沒有提交記錄的事務都會中止。因此，2PC 的 **提交點** 歸結為協調者上的常規單節點原子提交。

三階段提交

兩階段提交被稱為 **阻塞 (blocking)** - 原子提交協議，因為存在 2PC 可能卡住並等待協調者恢復的情況。理論上，可以使一個原子提交協議變為 **非阻塞 (nonblocking)** 的，以便在節點失敗時不會卡住。但是讓這個協議能在實踐中工作並沒有那麼簡單。

作為 2PC 的替代方案，已經提出了一種稱為 **三階段提交 (3PC)** 的演算法【13,80】。然而，3PC 假定網路延遲有界，節點響應時間有限；在大多數具有無限網路延遲和程序暫停的實際系統中（見 [第八章](#)），它並不能保證原子性。

通常，非阻塞原子提交需要一個 **完美的故障檢測器 (perfect failure detector)** 【67,71】——即一個可靠的機制來判斷一個節點是否已經崩潰。在具有無限延遲的網路中，超時並不是一種可靠的故障檢測機制，因為即使沒有節點崩潰，請求也可能由於網路問題而超時。出於這個原因，2PC 仍然被使用，儘管大家都清楚可能存在協調者故障的問題。

實踐中的分散式事務

分散式事務的名聲譏譽參半，尤其是那些透過兩階段提交實現的。一方面，它被視作提供了一個難以實現的重要的安全性保證；另一方面，它們因為導致運維問題，造成效能下降，做出超過能力範圍的承諾而飽受批評【81,82,83,84】。許多雲服務由於其導致的運維問題，而選擇不實現分散式事務【85,86】。

分散式事務的某些實現會帶來嚴重的效能損失——例如據報告稱，MySQL 中的分散式事務比單節點事務慢 10 倍以上【87】，所以當人們建議不要使用它們時就不足為奇了。兩階段提交所固有的效能成本，大部分是由於崩潰恢復所需的額外強制刷盤（`fsync`）【88】以及額外的網路往返。

但我們不應該直接忽視分散式事務，而應當更加仔細地審視這些事務，因為從中可以汲取重要的經驗教訓。首先，我們應該精確地說明“**分散式事務**”的含義。兩種截然不同的分散式事務型別經常被混淆：

- 資料庫內部的分散式事務

一些分散式資料庫（即在其標準配置中使用複製和分割槽的資料庫）支援資料庫節點之間的內部事務。例如，VoltDB 和 MySQL Cluster 的 NDB 儲存引擎就有這樣的內部事務支援。在這種情況下，所有參與事務的節點都執行相同的資料庫軟體。

- 異構分散式事務

在異構 (**heterogeneous**) 事務中，參與者是由兩種或兩種以上不同技術組成的：例如來自不同供應商的兩個資料庫，甚至是非資料庫系統（如訊息代理）。跨系統的分散式事務必須確保原子提交，儘管系統可能完全不同。資料庫內部事務不必與任何其他系統相容，因此它們可以使用任何協議，並能針對特定技術進行特定的最佳化。因此資料庫內部的分散式事務通常工作地很好。另一方面，跨異構技術的事務則更有挑戰性。

恰好一次的訊息處理

異構的分散式事務處理能夠以強大的方式整合不同的系統。例如：訊息佇列中的一條訊息可以被確認為已處理，當且僅當用於處理訊息的資料庫事務成功提交。這是透過在同一個事務中原子提交 訊息確認 和 資料庫寫入 兩個操作來實現的。藉由分散式事務的支援，即使訊息代理和資料庫是在不同機器上執行的兩種不相關的技術，這種操作也是可能的。如果訊息傳遞或資料庫事務任意一者失敗，兩者都會中止，因此訊息代理可能會在稍後安全地重傳訊息。因此，透過原子提交 訊息處理及其副作用，即使在成功之前需要幾次重試，也可以確保訊息被 **有效地** (**effectively**) 恰好處理一次。中止會拋棄部分完成事務所導致的任何副作用。

然而，只有當所有受事務影響的系統都使用同樣的 **原子提交協議** (**atomic commit protocol**) 時，這樣的分散式事務才是可能的。例如，假設處理訊息的副作用是傳送一封郵件，而郵件伺服器並不支援兩階段提交：如果訊息處理失敗並重試，則可能會發送兩次或更多次的郵件。但如果處理訊息的所有副作用都可以在事務中止時回滾，那麼這樣的處理流程就可以安全地重試，就好像什麼都沒有發生過一樣。

在 [第十一章](#) 中將再次回到“恰好一次”訊息處理的主題。讓我們先來看看允許這種異構分散式事務的原子提交協議。

XA事務

X/Open XA (擴充套件架構 (**eXtended Architecture**) 的縮寫) 是跨異構技術實現兩階段提交的標準 [【76,77】](#)。它於 1991 年推出並得到了廣泛的實現：許多傳統關係資料庫（包括 PostgreSQL、MySQL、DB2、SQL Server 和 Oracle）和訊息代理（包括 ActiveMQ、HornetQ、MSMQ 和 IBM MQ）都支援 XA。

XA 不是一個網路協議——它只是一個用來與事務協調者連線的 C API。其他語言也有這種 API 的繫結；例如在 Java EE 應用的世界中，XA 事務是使用 **Java 事務 API (JTA, Java Transaction API)** 實現的，而許多使用 **Java 資料庫連線 (JDBC, Java Database Connectivity)** 的資料庫驅動，以及許多使用 **Java 訊息服務 (JMS)** API 的訊息代理都支援 **Java 事務 API (JTA)**。

XA 假定你的應用使用網路驅動或客戶端庫來與 **參與者**（資料庫或訊息服務）進行通訊。如果驅動支援 XA，則意味著它會呼叫 XA API 以查明操作是否為分散式事務的一部分——如果是，則將必要的資訊發往資料庫伺服器。驅動還會向協調者暴露回撥介面，協調者可以透過回撥來要求參與者準備、提交或中止。

事務協調者需要實現 XA API。標準沒有指明應該如何實現，但實際上協調者通常只是一個庫，被載入到發起事務的應用的同一個程序中（而不是單獨的服務）。它在事務中跟蹤所有的參與者，並在要求它們 **準備** 之後收集參與者的響應（透過驅動回撥），並使用本地磁碟上的日誌記錄每次事務的決定（提交 / 中止）。

如果應用程序崩潰，或者執行應用的機器報銷了，協調者也隨之往生極樂。然後任何帶有 **準備了** 但未提交事務的參與者都會在疑慮中卡死。由於協調程式的日誌位於應用伺服器的本地磁碟上，因此必須重啟該伺服器，且協調程式庫必須讀取日誌以恢復每個事務的提交 / 中止結果。只有這樣，協調者才能使用資料庫驅動的 XA 回撥來要求參與者提交或中止。資料庫伺服器不能直接聯絡協調者，因為所有通訊都必須透過客戶端庫。

懷疑時持有鎖

為什麼我們這麼關心存疑事務？系統的其他部分就不能繼續正常工作，無視那些終將被清理的存疑事務嗎？

問題在於 **鎖 (locking)**。正如在“[讀已提交](#)”中所討論的那樣，資料庫事務通常獲取待修改的行上的 **行級排他鎖**，以防止髒寫。此外，如果要使用可序列化的隔離等級，則使用兩階段鎖定的資料庫也必須為事務所讀取的行加上共享鎖（請參閱“[兩階段鎖定](#)”）。

在事務提交或中止之前，資料庫不能釋放這些鎖（如 圖 9-9 中的陰影區域所示）。因此，在使用兩階段提交時，事務必須在整個存疑期間持有這些鎖。如果協調者已經崩潰，需要 20 分鐘才能重啟，那麼這些鎖將會被持有 20 分鐘。如果協調者的日誌由於某種原因徹底丟失，這些鎖將被永久持有——或至少在管理員手動解決該情況之前。

當這些鎖被持有時，其他事務不能修改這些行。根據資料庫的不同，其他事務甚至可能因為讀取這些行而被阻塞。因此，其他事務沒法兒簡單地繼續它們的業務了——如果它們要訪問同樣的資料，就會被阻塞。這可能會導致應用大面積進入不可用狀態，直到存疑事務被解決。

從協調者故障中恢復

理論上，如果協調者崩潰並重新啟動，它應該乾淨地從日誌中恢復其狀態，並解決任何存疑事務。然而在實踐中，孤立（*orphaned*）的存疑事務確實會出現【89,90】，即無論出於何種理由，協調者無法確定事務的結果（例如事務日誌已經由於軟體錯誤丟失或損壞）。這些事務無法自動解決，所以它們永遠待在資料庫中，持有鎖並阻塞其他事務。

即使重啟資料庫伺服器也無法解決這個問題，因為在 2PC 的正確實現中，即使重啟也必須保留存疑事務的鎖（否則就會冒違反原子性保證的風險）。這是一種棘手的情況。

唯一的出路是讓管理員手動決定提交還是回滾事務。管理員必須檢查每個存疑事務的參與者，確定是否有任何參與者已經提交或中止，然後將相同的結果應用於其他參與者。解決這個問題潛在地需要大量的人力，並且可能發生在嚴重的生產中斷期間（不然為什麼協調者處於這種糟糕的狀態），並很可能要在巨大精神壓力和時間壓力下完成。

許多 XA 的實現都有一個叫做 **啟發式決策**（heuristic decisions）的緊急逃生艙口：允許參與者單方面決定放棄或提交一個存疑事務，而無需協調者做出最終決定【76,77,91】。要清楚的是，這裡 **啟發式** 是 **可能破壞原子性**（probably breaking atomicity）的委婉說法，因為它違背了兩階段提交的系統承諾。因此，啟發式決策只是為了逃出災難性的情況而準備的，而不是為了日常使用的。

分散式事務的限制

XA 事務解決了保持多個參與者（資料系統）相互一致的現實的和重要的問題，但正如我們所看到的那樣，它也引入了嚴重的運維問題。特別來講，這裡的核心認識是：事務協調者本身就是一種資料庫（儲存了事務的結果），因此需要像其他重要資料庫一樣小心地打交道：

- 如果協調者沒有複製，而是隻在單臺機器上執行，那麼它是整個系統的失效單點（因為它的失效會導致其他應用伺服器阻塞在存疑事務持有的鎖上）。令人驚訝的是，許多協調者實現預設情況下並不是高可用的，或者只有基本的複製支援。
- 許多伺服器端應用都是使用無狀態模式開發的（受 HTTP 的青睞），所有持久狀態都儲存在資料庫中，因此具有應用伺服器可隨意按需新增刪除的優點。但是，當協調者成為應用伺服器的一部分時，它會改變部署的性質。突然間，協調者的日誌成為持久系統狀態的關鍵部分——與資料庫本身一樣重要，因為協調者日誌是為了在崩潰後恢復存疑事務所必需的。這樣的應用伺服器不再是無狀態的了。
- 由於 XA 需要相容各種資料系統，因此它必須是所有系統的最小公分母。例如，它不能檢測不同系統間的死鎖（因為這將需要一個標準協議來讓系統交換每個事務正在等待的鎖的資訊），而且它無法與 SSI（請參閱 [可序列化快照隔離](#)）協同工作，因為這需要一個跨系統定位衝突的協議。
- 對於資料庫內部的分散式事務（不是 XA），限制沒有這麼大——例如，分散式版本的 SSI 是可能的。然而仍然存在問題：2PC 成功提交一個事務需要所有參與者的響應。因此，如果系統的 **任何** 部分損壞，事務也會失敗。因此，分散式事務又有 **擴大失效**（amplifying failures）的趨勢，這又與我們構建容錯系統的目標背道而馳。

這些事實是否意味著我們應該放棄保持幾個系統相互一致的所有希望？不完全是——還有其他的辦法，可以讓我們在沒有異構分散式事務的痛苦的情況下實現同樣的事情。我們將在 [第十一章](#) 和 [第十二章](#) 回到這些話題。但首先，我們應該概括一下關於 **共識** 的話題。

容錯共識

非正式地，共識意味著讓幾個節點就某事達成一致。例如，如果有幾個人 **同時 (concurrently)** 嘗試預訂飛機上的最後一個座位，或劇院中的同一個座位，或者嘗試使用相同的使用者名稱註冊一個帳戶。共識演算法可以用來確定這些**互不相容 (mutually incompatible)** 的操作中，哪一個才是贏家。

共識問題通常形式化如下：一個或多個節點可以**提議 (propose)** 某些值，而共識演算法**決定 (decides)** 採用其中的某個值。在座位預訂的例子中，當幾個顧客同時試圖訂購最後一個座位時，處理顧客請求的每個節點可以**提議** 將要服務的顧客的 ID，而**決定** 指明瞭哪個顧客獲得了座位。

在這種形式下，共識演算法必須滿足以下性質【25】：^{xiii}

^{xiii} 這種共識的特殊形式被稱為**統一共識 (uniform consensus)**，相當於在具有不可靠故障檢測器的非同步系統中的**常規共識 (regular consensus)**【71】。學術文獻通常指的是**程序 (process)** 而不是節點，但我們在這裡使用**節點 (node)** 來與本書的其餘部分保持一致。 ↫

- **一致同意 (Uniform agreement)**

沒有兩個節點的決定不同。

- **完整性 (Integrity)**

沒有節點決定兩次。

- **有效性 (Validity)**

如果一個節點決定了值 v ，則 v 由某個節點所提議。

- **終止 (Termination)**

由所有未崩潰的節點來最終決定值。

一致同意 和 **完整性** 屬性定義了共識的核心思想：所有人都決定了相同的結果，一旦決定了，你就不能改變主意。**有效性** 屬性主要是為了排除平凡的解決方案：例如，無論提議了什麼值，你都可以有一個始終決定值為 `null` 的演算法，該演算法滿足**一致同意** 和**完整性** 屬性，但不滿足**有效性** 屬性。

如果你不關心容錯，那麼滿足前三個屬性很容易：你可以將一個節點硬編碼為“獨裁者”，並讓該節點做出所有的決定。但如果該節點失效，那麼系統就無法再做出任何決定。事實上，這就是我們在兩階段提交的情況中所看到的：如果協調者失效，那麼存疑的參與者就無法決定提交還是中止。

終止 屬性形式化了容錯的思想。它實質上說的是，一個共識演算法不能簡單地永遠閒坐著等死——換句話說，它必須取得進展。即使部分節點出現故障，其他節點也必須達成一項決定（**終止** 是一種**活性屬性**，而另外三種是**安全屬性**——請參閱“**安全性和活性**”）。

共識的系統模型假設，當一個節點“崩潰”時，它會突然消失而且永遠不會回來。（不像軟體崩潰，想像一下地震，包含你的節點的資料中心被山體滑坡所摧毀，你必須假設節點被埋在 30 英尺以下的泥土中，並且永遠不會重新上線）在這個系統模型中，任何需要等待節點恢復的演算法都不能滿足**終止** 屬性。特別是，2PC 不符合終止屬性的要求。

當然如果**所有**的節點都崩潰了，沒有一個在執行，那麼所有演算法都不可能決定任何事情。演算法可以容忍的失效數量是有限的：事實上可以證明，任何共識演算法都需要至少佔總體**多數 (majority)** 的節點正確工作，以確保終止屬性【67】。多數可以安全地組成法定人數（請參閱“**讀寫的法定人數**”）。

因此**終止** 屬性取決於一個假設，**不超過一半的節點崩潰或不可達**。然而即使多數節點出現故障或存在嚴重的網路問題，絕大多數共識的實現都能始終確保安全屬性得到滿足——一致同意，完整性和有效性【92】。因此，大規模的中斷可能會阻止系統處理請求，但是它不能透過使系統做出無效的決定來破壞共識系統。

大多數共識演算法假設不存在**拜占庭式錯誤**，正如在“**拜占庭故障**”一節中所討論的那樣。也就是說，如果一個節點沒有正確地遵循協議（例如，如果它向不同節點發送矛盾的訊息），它就可能會破壞協議的安全屬性。克服拜占庭故障，穩健地達成共識是可能的，只要少於三分之一的節點存在拜占庭故障【25,93】。但我們沒有地方在本書中詳細討論這些演算法了。

共識演算法和全序廣播

最著名的容錯共識演算法是 檢視戳複製（VSR, Viewstamped Replication）【94,95】，Paxos 【96,97,98,99】，Raft 【22,100,101】以及 Zab 【15,21,102】。這些演算法之間有不少相似之處，但它們並不相同【103】。在本書中我們不會介紹各種演算法的詳細細節：瞭解一些它們共通的高階思想通常已經足夠了，除非你準備自己實現一個共識系統。（可能並不明智，相當難【98,104】）。

大多數這些演算法實際上並不直接使用這裡描述的形式化模型（提議與決定單個值，並滿足一致同意、完整性、有效性和終止屬性）。取而代之的是，它們決定了值的順序（sequence），這使它們成為全序廣播演算法，正如本章前面所討論的那樣（請參閱“[全序廣播](#)”）。

請記住，全序廣播要求將訊息按照相同的順序，恰好傳遞一次，準確傳送到所有節點。如果仔細思考，這相當於進行了幾輪共識：在每一輪中，節點提議下一條要傳送的訊息，然後決定在全序中下一條要傳送的訊息【67】。

所以，全序廣播相當於重複進行多輪共識（每次共識決定與一次訊息傳遞相對應）：

- 由於 **一致同意** 屬性，所有節點決定以相同的順序傳遞相同的訊息。
- 由於 **完整性** 屬性，訊息不會重複。
- 由於 **有效性** 屬性，訊息不會被損壞，也不能憑空編造。
- 由於 **終止** 屬性，訊息不會丟失。

檢視戳複製，Raft 和 Zab 直接實現了全序廣播，因為這樣做比重複一次一值（one value a time）的共識更高效。在 Paxos 的情況下，這種最佳化被稱為 Multi-Paxos。

單主複製與共識

在 [第五章](#) 中，我們討論了單主複製（請參閱“[領導者與追隨者](#)”），它將所有的寫入操作都交給主庫，並以相同的順序將它們應用到從庫，從而使副本保持在最新狀態。這實際上不就是一個全序廣播嗎？為什麼我們在 [第五章](#) 裡一點都沒擔心過共識問題呢？

答案取決於如何選擇領導者。如果主庫是由運維人員手動選擇和配置的，那麼你實際上擁有一種 **獨裁型別** 的“共識演算法”：只有一個節點被允許接受寫入（即決定寫入複製日誌的順序），如果該節點發生故障，則系統將無法寫入，直到運維手動配置其他節點作為主庫。這樣的系統在實踐中可以表現良好，但它無法滿足共識的 **終止** 屬性，因為它需要人為干預才能取得進展。

一些資料庫會自動執行領導者選舉和故障切換，如果舊主庫失效，會提拔一個從庫為新主庫（請參閱“[處理節點宕機](#)”）。這使我們向容錯的全序廣播更進一步，從而達成共識。

但是還有一個問題。我們之前曾經討論過腦裂的問題，並且說過所有的節點都需要同意是誰領導，否則兩個不同的節點都會認為自己是領導者，從而導致資料庫進入不一致的狀態。因此，選出一位領導者需要共識。但如果這裡描述的共識演算法實際上是全序廣播演算法，並且全序廣播就像單主複製，而單主複製需要一個領導者，那麼…

這樣看來，要選出一個領導者，我們首先需要一個領導者。要解決共識問題，我們首先需要解決共識問題。我們如何跳出這個先有雞還是先有蛋的問題？

紀元編號和法定人數

迄今為止所討論的所有共識協議，在內部都以某種形式使用一個領導者，但它們並不能保證領導者是獨一無二的。相反，它們可以做出更弱的保證：協議定義了一個 **紀元編號**（epoch number，在 Paxos 中被稱為 **投票編號**，即 ballot number，在檢視戳複製中被稱為 **檢視編號**，即 view number，以及在 Raft 中被稱為 **任期號碼**，即 term number），並確保在每個時代中，領導者都是唯一的。

每次當現任領導者被認為掛掉的時候，節點間就會開始一場投票，以選出一個新領導者。這次選舉被賦予一個遞增的紀元編號，因此紀元編號是全序且單調遞增的。如果兩個不同的時代的領導者之間出現衝突（也許是因為前任領導者實際上並未死亡），那麼帶有更高紀元編號的領導者勝出。

在任何領導者被允許決定任何事情之前，必須先檢查是否存在其他帶有更高紀元編號的領導者，它們可能會做出相互衝突的決定。領導者如何知道自己沒有被另一個節點趕下臺？回想一下在“[真相由多數所定義](#)”中提到的：一個節點不一定能相信自己的判斷——因為只有節點自己認為自己是領導者，並不一定意味著其他節點接受它作為它們的領導者。

相反，它必須從 **法定人數 (quorum)** 的節點中獲取選票（請參閱“[讀寫的法定人數](#)”）。對領導者想要做出的每一個決定，都必須將提議值傳送給其他節點，並等待法定人數的節點響應並贊成提案。法定人數通常（但不總是）由多數節點組成【105】。只有在沒有意識到任何帶有更高紀元編號的領導者的情況下，一個節點才會投票贊成提議。

因此，我們有兩輪投票：第一次是為了選出一位領導者，第二次是對領導者的提議進行表決。關鍵的洞察在於，這兩次投票的 **法定人群** 必須相互 **重疊 (overlap)**：如果一個提案的表決通過，則至少得有一個參與投票的節點也必須參加過最近的領導者選舉【105】。因此，如果在一個提案的表決過程中沒有出現更高的紀元編號。那麼現任領導者就可以得出這樣的結論：沒有發生過更高時代的領導選舉，因此可以確定自己仍然在領導。然後它就可以安全地對提議值做出決定。

這一投票過程表面上看起來很像兩階段提交。最大的區別在於，2PC 中協調者不是由選舉產生的，而且 2PC 則要求 **所有** 參與者都投贊成票，而容錯共識演算法只需要多數節點的投票。而且，共識演算法還定義了一個恢復過程，節點可以在選舉出新的領導者之後進入一個一致的狀態，確保始終能滿足安全屬性。這些區別正是共識演算法正確性和容錯性的關鍵。

共識的侷限性

共識演算法對於分散式系統來說是一個巨大的突破：它為其他充滿不確定性的系統帶來了基礎的安全屬性（一致同意，完整性和有效性），然而它們還能保持容錯（只要多數節點正常工作且可達，就能取得進展）。它們提供了全序廣播，因此它們也可以以一種容錯的方式實現線性一致的原子操作（請參閱“[使用全序廣播實現線性一致的儲存](#)”）。

儘管如此，它們並不是在所有地方都用上了，因為好處總是有代價的。

節點在做出決定之前對提議進行投票的過程是一種同步複製。如“[同步複製與非同步複製](#)”中所述，通常資料庫會配置為非同步複製模式。在這種配置中發生故障切換時，一些已經提交的資料可能會丟失——但是為了獲得更好的效能，許多人選擇接受這種風險。

共識系統總是需要嚴格多數來運轉。這意味著你至少需要三個節點才能容忍單節點故障（其餘兩個構成多數），或者至少有五個節點來容忍兩個節點發生故障（其餘三個構成多數）。如果網路故障切斷了某些節點同其他節點的連線，則只有多數節點所在的網路可以繼續工作，其餘部分將被阻塞（請參閱“[線性一致性的代價](#)”）。

大多數共識演算法假定參與投票的節點是固定的集合，這意味著你不能簡單的在叢集中新增或刪除節點。共識演算法的**動態成員擴充套件 (dynamic membership extension)** 允許叢集中的節點集隨時間推移而變化，但是它們比靜態成員演算法要難理解得多。

共識系統通常依靠超時來檢測失效的節點。在網路延遲高度變化的環境中，特別是在地理上散佈的系統中，經常發生一個節點由於暫時的網路問題，錯誤地認為領導者已經失效。雖然這種錯誤不會損害安全屬性，但頻繁的領導者選舉會導致糟糕的效能表現，因系統最後可能花在權力傾紝上的時間要比花在建設性工作的多得多。

有時共識演算法對網路問題特別敏感。例如 Raft 已被證明存在讓人不悅的極端情況【106】：如果整個網路工作正常，但只有一條特定的網路連線一直不可靠，Raft 可能會進入領導者在兩個節點間頻繁切換的局面，或者當前領導者不斷被迫辭職以致系統實質上毫無進展。其他一致性演算法也存在類似的問題，而設計能健壯應對不可靠網路的演算法仍然是一个開放的研究問題。

成員與協調服務

像 ZooKeeper 或 etcd 這樣的專案通常被描述為“分散式鍵值儲存”或“協調與配置服務”。這種服務的 API 看起來非常像資料庫：你可以讀寫給定鍵的值，並遍歷鍵。所以如果它們基本上算是資料庫的話，為什麼它們要把工夫全花在實現一個共識演算法上呢？是什麼使它們區別於其他任意型別的資料庫？

為了理解這一點，簡單瞭解如何使用 ZooKeeper 這類服務是很有幫助的。作為應用開發人員，你很少需要直接使用 ZooKeeper，因為它實際上不適合當成通用資料庫來用。更有可能的是，你會透過其他專案間接依賴它，例如 HBase、Hadoop YARN、OpenStack Nova 和 Kafka 都依賴 ZooKeeper 在後臺執行。這些專案從它那裡得到了什麼？

ZooKeeper 和 etcd 被設計為容納少量完全可以放在記憶體中的資料（雖然它們仍然會寫入磁碟以保證永續性），所以你不會想著把所有應用資料放到這裡。這些少量資料會透過容錯的全序廣播演算法複製到所有節點上。正如前面所討論的那樣，資料庫複製需要的就是全序廣播：如果每條訊息代表對資料庫的寫入，則以相同的順序應用相同的寫入操作可以使副本之間保持一致。

ZooKeeper 模仿了 Google 的 Chubby 鎖服務【14,98】，不僅實現了全序廣播（因此也實現了共識），而且還構建了一組有趣的其他特性，這些特性在構建分散式系統時變得特別有用：

- 線性一致性的原子操作

使用原子 CAS 操作可以實現鎖：如果多個節點同時嘗試執行相同的操作，只有一個節點會成功。共識協議保證了操作的原子性和線性一致性，即使節點發生故障或網路在任意時刻中斷。分散式鎖通常以 **租約 (lease)** 的形式實現，租約有一個到期時間，以便在客戶端失效的情況下最終能被釋放（請參閱“[程序暫停](#)”）。

- 操作的全序排序

如“[領導者和鎖](#)”中所述，當某個資源受到鎖或租約的保護時，你需要一個防護令牌來防止客戶端在程序暫停的情況下彼此衝突。防護令牌是每次鎖被獲取時單調增加的數字。ZooKeeper 透過全序化所有操作來提供這個功能，它為每個操作提供一個單調遞增的事務 ID (`zxid`) 和版本號 (`cversion`) 【15】。

- 失效檢測

客戶端在 ZooKeeper 伺服器上維護一個長期會話，客戶端和伺服器週期性地交換心跳包來檢查節點是否還活著。即使連線暫時中斷，或者 ZooKeeper 節點失效，會話仍保持在活躍狀態。但如果心跳停止的持續時間超出會話超時，ZooKeeper 會宣告該會話已死亡。當會話超時時（ZooKeeper 稱這些節點為 **臨時節點**，即 `ephemeral nodes`），會話持有的任何鎖都可以配置為自動釋放。

- 變更通知

客戶端不僅可以讀取其他客戶端建立的鎖和值，還可以監聽它們的變更。因此，客戶端可以知道另一個客戶端何時加入叢集（基於新客戶端寫入 ZooKeeper 的值），或發生故障（因其會話超時，而其臨時節點消失）。透過訂閱通知，客戶端不用再透過頻繁輪詢的方式來找出變更。

在這些功能中，只有線性一致的原子操作才真的需要共識。但正是這些功能的組合，使得像 ZooKeeper 這樣的系統在分散式協調中非常有用。

將工作分配給節點

ZooKeeper/Chubby 模型執行良好的一個例子是，如果你有幾個程序例項或服務，需要選擇其中一個例項作為主庫或首選服務。如果領導者失敗，其他節點之一應該接管。這對單主資料庫當然非常實用，但對作業排程程式和類似的有狀態系統也很好用。

另一個例子是，當你有一些分割槽資源（資料庫、訊息流、檔案儲存、分散式 Actor 系統等），並需要決定將哪個分割槽分配給哪個節點時。當新節點加入叢集時，需要將某些分割槽從現有節點移動到新節點，以便重新平衡負載（請參閱“[分割槽再平衡](#)”）。當節點被移除或失效時，其他節點需要接管失效節點的工作。

這類任務可以透過在 ZooKeeper 中明智地使用原子操作，臨時節點與通知來實現。如果設計得當，這種方法允許應用自動從故障中恢復而無需人工干預。不過這並不容易，儘管已經有不少在 ZooKeeper 客戶端 API 基礎之上提供更高層工具的庫，例如 Apache Curator 【17】。但它仍然要比嘗試從頭實現必要的共識演算法要好得多，這樣的嘗試鮮有成功記錄【107】。

應用最初只能在單個節點上執行，但最終可能會增長到數千個節點。試圖在如此之多的節點上進行多數投票將是非常低效的。相反，ZooKeeper 在固定數量的節點（通常是三到五個）上執行，並在這些節點之間執行其多數票，同時支援潛在的大量客戶端。因此，ZooKeeper 提供了一種將協調節點（共識，操作排序和故障檢測）的一些工作“外包”到外部

服務的方式。

通常，由 ZooKeeper 管理的資料型別的變化十分緩慢：代表“分割槽 7 中的節點執行在 10.1.1.23 上”的資訊可能會在幾分鐘或幾小時的時間內發生變化。它不是用來儲存應用的執行時狀態的，後者每秒可能會改變數千甚至數百萬次。如果應用狀態需要從一個節點複製到另一個節點，則可以使用其他工具（如 Apache BookKeeper【108】）。

服務發現

ZooKeeper、etcd 和 Consul 也經常用於服務發現——也就是找出你需要連線到哪個 IP 地址才能到達特定的服務。在雲資料中心環境中，虛擬機器來往往很常見，你通常不會事先知道服務的 IP 地址。相反，你可以配置你的服務，使其在啟動時註冊服務登錄檔中的網路端點，然後可以由其他服務找到它們。

但是，服務發現是否需要達成共識還不太清楚。DNS 是查詢服務名稱的 IP 地址的傳統方式，它使用多層快取來實現良好的效能和可用性。從 DNS 讀取是絕對不線性一致性的，如果 DNS 查詢的結果有點陳舊，通常不會有問題【109】。DNS 的可用性和對網路中斷的魯棒性更重要。

儘管服務發現並不需要共識，但領導者選舉卻是如此。因此，如果你的共識系統已經知道領導是誰，那麼也可以使用這些資訊來幫助其他服務發現領導是誰。為此，一些共識系統支援只讀快取副本。這些副本非同步接收共識演算法所有決策的日誌，但不主動參與投票。因此，它們能夠提供不需要線性一致性的讀取請求。

成員資格服務

ZooKeeper 和它的小夥伴們可以看作是成員資格服務（membership services）研究的悠久歷史的一部分，這個歷史可以追溯到 20 世紀 80 年代，並且對建立高度可靠的系統（例如空中交通管制）非常重要【110】。

成員資格服務確定哪些節點當前處於活動狀態並且是叢集的活動成員。正如我們在 第八章 中看到的那樣，由於無限的網路延遲，無法可靠地檢測到另一個節點是否發生故障。但是，如果你透過共識來進行故障檢測，那麼節點可以就哪些節點應該被認為是存在或不存在達成一致。

即使它確實存在，仍然可能發生一個節點被共識錯誤地宣告死亡。但是對於一個系統來說，知道哪些節點構成了當前的成員關係是非常有用的。例如，選擇領導者可能意味著簡單地選擇當前成員中編號最小的成員，但如果不同的節點對現有的成員都有誰有不同意見，則這種方法將不起作用。

本章小結

在本章中，我們從幾個不同的角度審視了關於一致性與共識的話題。我們深入研究了線性一致性（一種流行的一致性模型）：其目標是使多副本資料看起來好像只有一個副本一樣，並使其上所有操作都原子性地生效。雖然線性一致性因為簡單易懂而很吸引人——它使資料庫表現的好像單執行緒程式中的一個變數一樣，但它有著速度緩慢的缺點，特別是在網路延遲很大的環境中。

我們還探討了因果性，因果性對系統中的事件施加了順序（什麼發生在什麼之前，基於因與果）。與線性一致不同，線性一致性將所有操作放在單一的全序時間線中，因果一致性為我們提供了一個較弱的一致性模型：某些事件可以是併發的，所以版本歷史就像是一條不斷分叉與合併的時間線。因果一致性沒有線性一致性的協調開銷，而且對網路問題的敏感性要低得多。

但即使捕獲到因果順序（例如使用蘭伯特時間戳），我們發現有些事情也不能透過這種方式實現：在“光有時間戳排序還不夠”一節的例子中，我們需要確保使用者名稱是唯一的，並拒絕同一使用者名稱的其他併發註冊。如果一個節點要透過註冊，則需要知道其他的節點沒有在併發搶注同一使用者名稱的過程中。這個問題引領我們走向 共識。

我們看到，達成共識意味著以這樣一種方式決定某件事：所有節點一致同意所做決定，且這一決定不可撤銷。透過深入挖掘，結果我們發現很廣泛的一系列問題實際上都可以歸結為共識問題，並且彼此等價（從這個意義上來講，如果你有其中之一的解決方案，就可以輕易將它轉換為其他問題的解決方案）。這些等價的問題包括：

- 線性一致性的 CAS 暫存器

暫存器需要基於當前值是否等於操作給出的引數，原子地 決定 是否設定新值。

- 原子事務提交

資料庫必須 決定 是否提交或中止分散式事務。

- 全序廣播

訊息系統必須 決定 傳遞訊息的順序。

- 鎖和租約

當幾個客戶端爭搶鎖或租約時，由鎖來 決定 哪個客戶端成功獲得鎖。

- 成員 / 協調服務

給定某種故障檢測器（例如超時），系統必須 決定 哪些節點活著，哪些節點因為會話超時需要被宣告死亡。

- 唯一性約束

當多個事務同時嘗試使用相同的鍵建立衝突記錄時，約束必須 決定 哪一個被允許，哪些因為違反約束而失敗。

如果你只有一個節點，或者你願意將決策的權能分配給單個節點，所有這些事都很簡單。這就是在單領導者資料庫中發生的事情：所有決策權歸屬於領導者，這就是為什麼這樣的資料庫能夠提供線性一致的操作，唯一性約束，完全有序的複製日誌，以及更多。

但如果該領導者失效，或者如果網路中斷導致領導者不可達，這樣的系統就無法取得任何進展。應對這種情況可以有三種方法：

1. 等待領導者恢復，接受系統將在這段時間阻塞的事實。許多 XA/JTA 事務協調者選擇這個選項。這種方法並不能完全達成共識，因為它不能滿足 終止 屬性的要求：如果領導者續命失敗，系統可能會永久阻塞。
2. 人工故障切換，讓人類選擇一個新的領導者節點，並重新配置系統使之生效，許多關係型資料庫都採用這種方式。這是一種來自“天意”的共識——由計算機系統之外的運維人員做出決定。故障切換的速度受到人類行動速度的限制，通常要比計算機慢（得多）。
3. 使用演算法自動選擇一個新的領導者。這種方法需要一種共識演算法，使用成熟的演算法來正確處理惡劣的網路條件是明智之舉【107】。

儘管單領導者資料庫可以提供線性一致性，且無需對每個寫操作都執行共識演算法，但共識對於保持及變更領導權仍然是必須的。因此從某種意義上說，使用單個領導者不過是“緩兵之計”：共識仍然是需要的，只是在另一個地方，而且沒那麼頻繁。好訊息是，容錯的共識演算法與容錯的共識系統是存在的，我們在本章中簡要地討論了它們。

像 ZooKeeper 這樣的工具為應用提供了“外包”的共識、故障檢測和成員服務。它們扮演了重要的角色，雖說使用不易，但總比自己去開發一個能經受 第八章 中所有問題考驗的演算法要好得多。如果你發現自己想要解決的問題可以歸結為共識，並且希望它能容錯，使用一個類似 ZooKeeper 的東西是明智之舉。

儘管如此，並不是所有系統都需要共識：例如，無領導者複製和多領導者複製系統通常不會使用全域性的共識。這些系統中出現的衝突（請參閱“處理寫入衝突”）正是不同領導者之間沒有達成共識的結果，但這也許並沒有關係：也許我們只是需要接受沒有線性一致性的事實，並學會更好地與具有分支與合併版本歷史的資料打交道。

本章引用了大量關於分散式系統理論的研究。雖然理論論文和證明並不總是容易理解，有時也會做出不切實際的假設，但它們對於指導這一領域的實踐有著極其重要的價值：它們幫助我們推理什麼可以做，什麼不可以做，幫助我們找到反直覺的分散式系統缺陷。如果你有時間，這些參考資料值得探索。

這裡已經到了本書 第二部分 的末尾，第二部介紹了複製（第五章）、分割槽（第六章）、事務（第七章）、分散式系統的故障模型（第八章）以及最後的一致性與共識（第九章）。現在我們已經奠定了紮實的理論基礎，我們將在 第三部分 再次轉向更實際的系統，並討論如何使用異構的元件積木塊構建強大的應用。

參考文獻

1. Peter Bailis and Ali Ghodsi: “[Eventual Consistency Today: Limitations, Extensions, and Beyond](#),” *ACM Queue*, volume 11, number 3, pages 55–63, March 2013. doi:[10.1145/2460276.2462076](https://doi.org/10.1145/2460276.2462076)
2. Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin: “[Consistency, Availability, and Convergence](#),” University of Texas at Austin, Department of Computer Science, Tech Report UTCS TR-11-22, May 2011.
3. Alex Scotti: “[Adventures in Building Your Own Database](#),” at *All Your Base*, November 2015.
4. Peter Bailis, Aaron Davidson, Alan Fekete, et al.: “[Highly Available Transactions: Virtues and Limitations](#),” at *40th International Conference on Very Large Data Bases (VLDB)*, September 2014. Extended version published as pre-print arXiv:1302.0309 [cs.DB].
5. Paolo Viotti and Marko Vukolić: “[Consistency in Non-Transactional Distributed Storage Systems](#),” arXiv:1512.00168, 12 April 2016.
6. Maurice P. Herlihy and Jeannette M. Wing: “[Linearizability: A Correctness Condition for Concurrent Objects](#),” *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 12, number 3, pages 463–492, July 1990. doi:[10.1145/78969.78972](https://doi.org/10.1145/78969.78972)
7. Leslie Lamport: “[On interprocess communication](#),” *Distributed Computing*, volume 1, number 2, pages 77–101, June 1986. doi:[10.1007/BF01786228](https://doi.org/10.1007/BF01786228)
8. David K. Gifford: “[Information Storage in a Decentralized Computer System](#),” Xerox Palo Alto Research Centers, CSL-81-8, June 1981.
9. Martin Kleppmann: “[Please Stop Calling Databases CP or AP](#),” martin.kleppmann.com, May 11, 2015.
10. Kyle Kingsbury: “[Call Me Maybe: MongoDB Stale Reads](#),” aphyr.com, April 20, 2015.
11. Kyle Kingsbury: “[Computational Techniques in Knossos](#),” aphyr.com, May 17, 2014.
12. Peter Bailis: “[Linearizability Versus Serializability](#),” bailis.org, September 24, 2014.
13. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman: *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987. ISBN: 978-0-201-10715-9, available online at research.microsoft.com.
14. Mike Burrows: “[The Chubby Lock Service for Loosely-Coupled Distributed Systems](#),” at *7th USENIX Symposium on Operating System Design and Implementation (OSDI)*, November 2006.
15. Flavio P. Junqueira and Benjamin Reed: *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, 2013. ISBN: 978-1-449-36130-3
16. “[etcd 2.0.12 Documentation](#),” CoreOS, Inc., 2015.
17. “[Apache Curator](#),” Apache Software Foundation, curator.apache.org, 2015.
18. Morali Vallath: *Oracle 10g RAC Grid, Services & Clustering*. Elsevier Digital Press, 2006. ISBN: 978-1-555-58321-7
19. Peter Bailis, Alan Fekete, Michael J Franklin, et al.: “[Coordination-Avoiding Database Systems](#),” *Proceedings of the VLDB Endowment*, volume 8, number 3, pages 185–196, November 2014.
20. Kyle Kingsbury: “[Call Me Maybe: etcd and Consul](#),” aphyr.com, June 9, 2014.
21. Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini: “[Zab: High-Performance Broadcast for Primary-Backup Systems](#),” at *41st IEEE International Conference on Dependable Systems and Networks (DSN)*, June 2011. doi:[10.1109/DSN.2011.5958223](https://doi.org/10.1109/DSN.2011.5958223)
22. Diego Ongaro and John K. Ousterhout: “[In Search of an Understandable Consensus Algorithm \(Extended Version\)](#),” at *USENIX Annual Technical Conference (ATC)*, June 2014.
23. Hagit Attiya, Amotz Bar-Noy, and Danny Dolev: “[Sharing Memory Robustly in Message-Passing Systems](#),” *Journal of the ACM*, volume 42, number 1, pages 124–142, January 1995. doi:[10.1145/200836.200869](https://doi.org/10.1145/200836.200869)
24. Nancy Lynch and Alex Shvartsman: “[Robust Emulation of Shared Memory Using Dynamic Quorum-Acknowledged Broadcasts](#),” at *27th Annual International Symposium on Fault-Tolerant Computing (FTCS)*, June 1997. doi:[10.1109/FTCS.1997.614100](https://doi.org/10.1109/FTCS.1997.614100)
25. Christian Cachin, Rachid Guerraoui, and Luís Rodrigues: *Introduction to Reliable and Secure Distributed Programming*, 2nd edition. Springer, 2011. ISBN: 978-3-642-15259-7, doi:[10.1007/978-3-642-15260-3](https://doi.org/10.1007/978-3-642-15260-3)
26. Sam Elliott, Mark Allen, and Martin Kleppmann: [personal communication](#), thread on twitter.com, October 15, 2015.
27. Niklas Ekström, Mikhail Panchenko, and Jonathan Ellis: “[Possible Issue with Read Repair?](#),” email thread on [cassandra-dev mailing list](mailto:cassandra-dev), October 2012.
28. Maurice P. Herlihy: “[Wait-Free Synchronization](#),” *ACM Transactions on Programming Languages and Systems*

- (TOPLAS), volume 13, number 1, pages 124–149, January 1991. doi:10.1145/114005.102808
29. Armando Fox and Eric A. Brewer: “[Harvest, Yield, and Scalable Tolerant Systems](#),” at *7th Workshop on Hot Topics in Operating Systems* (HotOS), March 1999. doi:10.1109/HOTOS.1999.798396
 30. Seth Gilbert and Nancy Lynch: “[Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services](#),” *ACM SIGACT News*, volume 33, number 2, pages 51–59, June 2002. doi:10.1145/564585.564601
 31. Seth Gilbert and Nancy Lynch: “[Perspectives on the CAP Theorem](#),” *IEEE Computer Magazine*, volume 45, number 2, pages 30–36, February 2012. doi:10.1109/MC.2011.389
 32. Eric A. Brewer: “[CAP Twelve Years Later: How the 'Rules' Have Changed](#),” *IEEE Computer Magazine*, volume 45, number 2, pages 23–29, February 2012. doi:10.1109/MC.2012.37
 33. Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen: “[Consistency in Partitioned Networks](#),” *ACM Computing Surveys*, volume 17, number 3, pages 341–370, September 1985. doi:10.1145/5505.5508
 34. Paul R. Johnson and Robert H. Thomas: “[RFC 677: The Maintenance of Duplicate Databases](#),” Network Working Group, January 27, 1975.
 35. Bruce G. Lindsay, Patricia Griffiths Selinger, C. Galtieri, et al.: “[Notes on Distributed Databases](#),” IBM Research, Research Report RJ2571(33471), July 1979.
 36. Michael J. Fischer and Alan Michael: “[Sacrificing Serializability to Attain High Availability of Data in an Unreliable Network](#),” at *1st ACM Symposium on Principles of Database Systems* (PODS), March 1982. doi:10.1145/588111.588124
 37. Eric A. Brewer: “[NoSQL: Past, Present, Future](#),” at QCon San Francisco, November 2012.
 38. Henry Robinson: “[CAP Confusion: Problems with 'Partition Tolerance'](#),” blog.cloudera.com, April 26, 2010.
 39. Adrian Cockcroft: “[Migrating to Microservices](#),” at QCon London, March 2014.
 40. Martin Kleppmann: “[A Critique of the CAP Theorem](#),” arXiv:1509.05393, September 17, 2015.
 41. Nancy A. Lynch: “[A Hundred Impossibility Proofs for Distributed Computing](#),” at *8th ACM Symposium on Principles of Distributed Computing* (PODC), August 1989. doi:10.1145/72981.72982
 42. Hagit Attiya, Faith Ellen, and Adam Morrison: “[Limitations of Highly-Available Eventually-Consistent Data Stores](#) (<http://www.cs.technion.ac.il/people/mad/online-publications/podc2015-replds.pdf>),” at *ACM Symposium on Principles of Distributed Computing* (PODC), July 2015. doi:10.1145/2767386.2767419
 43. Peter Sewell, Susmit Sarkar, Scott Owens, et al.: “[x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors](#),” *Communications of the ACM*, volume 53, number 7, pages 89–97, July 2010. doi:10.1145/1785414.1785443
 44. Martin Thompson: “[Memory Barriers/Fences](#),” mechanical-sympathy.blogspot.co.uk, July 24, 2011.
 45. Ulrich Drepper: “[What Every Programmer Should Know About Memory](#),” akkadia.org, November 21, 2007.
 46. Daniel J. Abadi: “[Consistency Tradeoffs in Modern Distributed Database System Design](#),” *IEEE Computer Magazine*, volume 45, number 2, pages 37–42, February 2012. doi:10.1109/MC.2012.33
 47. Hagit Attiya and Jennifer L. Welch: “[Sequential Consistency Versus Linearizability](#),” *ACM Transactions on Computer Systems* (TOCS), volume 12, number 2, pages 91–122, May 1994. doi:10.1145/176575.176576
 48. Mustaque Ahamed, Gil Neiger, James E. Burns, et al.: “[Causal Memory: Definitions, Implementation, and Programming](#),” *Distributed Computing*, volume 9, number 1, pages 37–49, March 1995. doi:10.1007/BF01784241
 49. Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen: “[Stronger Semantics for Low-Latency Geo-Replicated Storage](#),” at *10th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2013.
 50. Marek Zawirski, Annette Bieniusa, Valter Balegas, et al.: “[SwiftCloud: Fault-Tolerant Geo-Replication Integrated All the Way to the Client Machine](#),” INRIA Research Report 8347, August 2013.
 51. Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica: “[Bolt-on Causal Consistency](#),” at *ACM International Conference on Management of Data* (SIGMOD), June 2013.
 52. Philippe Ajoux, Nathan Bronson, Sanjeev Kumar, et al.: “[Challenges to Adopting Stronger Consistency at Scale](#),” at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.
 53. Peter Bailis: “[Causality Is Expensive \(and What to Do About It\)](#),” bailis.org, February 5, 2014.
 54. Ricardo Gonçalves, Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte: “[Concise Server-Wide Causality Management for Eventually Consistent Data Stores](#),” at *15th IFIP International Conference on Distributed*

- Applications and Interoperable Systems* (DAIS), June 2015. doi:10.1007/978-3-319-19129-4_6
55. Rob Conery: “A Better ID Generator for PostgreSQL,” rob.conery.io, May 29, 2014.
 56. Leslie Lamport: “Time, Clocks, and the Ordering of Events in a Distributed System,” *Communications of the ACM*, volume 21, number 7, pages 558–565, July 1978. doi:10.1145/359545.359563
 57. Xavier Défago, André Schiper, and Péter Urbán: “Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey,” *ACM Computing Surveys*, volume 36, number 4, pages 372–421, December 2004. doi:10.1145/1041680.1041682
 58. Hagit Attiya and Jennifer Welch: *Distributed Computing: Fundamentals, Simulations and Advanced Topics*, 2nd edition. John Wiley & Sons, 2004. ISBN: 978-0-471-45324-6, doi:10.1002/0471478210
 59. Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, et al.: “CORFU: A Shared Log Design for Flash Clusters,” at *9th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2012.
 60. Fred B. Schneider: “Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial,” *ACM Computing Surveys*, volume 22, number 4, pages 299–319, December 1990.
 61. Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, et al.: “Calvin: Fast Distributed Transactions for Partitioned Database Systems,” at *ACM International Conference on Management of Data* (SIGMOD), May 2012.
 62. Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, et al.: “Tango: Distributed Data Structures over a Shared Log,” at *24th ACM Symposium on Operating Systems Principles* (SOSP), November 2013. doi:10.1145/2517349.2522732
 63. Robbert van Renesse and Fred B. Schneider: “Chain Replication for Supporting High Throughput and Availability,” at *6th USENIX Symposium on Operating System Design and Implementation* (OSDI), December 2004.
 64. Leslie Lamport: “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs,” *IEEE Transactions on Computers*, volume 28, number 9, pages 690–691, September 1979. doi:10.1109/TC.1979.1675439
 65. Enis Söztutar, Devaraj Das, and Carter Shanklin: “Apache HBase High Availability at the Next Level,” hortonworks.com, January 22, 2015.
 66. Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, et al.: “PNUTS: Yahoo!’s Hosted Data Serving Platform,” at *34th International Conference on Very Large Data Bases* (VLDB), August 2008. doi:10.14778/1454159.1454167
 67. Tushar Deepak Chandra and Sam Toueg: “Unreliable Failure Detectors for Reliable Distributed Systems,” *Journal of the ACM*, volume 43, number 2, pages 225–267, March 1996. doi:10.1145/226643.226647
 68. Michael J. Fischer, Nancy Lynch, and Michael S. Paterson: “Impossibility of Distributed Consensus with One Faulty Process,” *Journal of the ACM*, volume 32, number 2, pages 374–382, April 1985. doi:10.1145/3149.214121
 69. Michael Ben-Or: “Another Advantage of Free Choice: Completely Asynchronous Agreement Protocols,” at *2nd ACM Symposium on Principles of Distributed Computing* (PODC), August 1983. doi:10.1145/800221.806707
 70. Jim N. Gray and Leslie Lamport: “Consensus on Transaction Commit,” *ACM Transactions on Database Systems* (TODS), volume 31, number 1, pages 133–160, March 2006. doi:10.1145/1132863.1132867
 71. Rachid Guerraoui: “Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus,” at *9th International Workshop on Distributed Algorithms* (WDAG), September 1995. doi:10.1007/BFb0022140
 72. Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, et al.: “All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications,” at *11th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2014.
 73. Jim Gray: “The Transaction Concept: Virtues and Limitations,” at *7th International Conference on Very Large Data Bases* (VLDB), September 1981.
 74. Hector Garcia-Molina and Kenneth Salem: “Sagas,” at *ACM International Conference on Management of Data* (SIGMOD), May 1987. doi:10.1145/38713.38742
 75. C. Mohan, Bruce G. Lindsay, and Ron Obermarck: “Transaction Management in the R* Distributed Database Management System,” *ACM Transactions on Database Systems*, volume 11, number 4, pages 378–396, December 1986. doi:10.1145/7239.7266
 76. “Distributed Transaction Processing: The XA Specification,” X/Open Company Ltd., Technical Standard

- XO/CAE/91/300, December 1991. ISBN: 978-1-872-63024-3
77. Mike Spille: “[XA Exposed, Part II](#),” jroller.com, April 3, 2004.
 78. Ivan Silva Neto and Francisco Reverbel: “[Lessons Learned from Implementing WS-Coordination and WS-AtomicTransaction](#),” at *7th IEEE/ACIS International Conference on Computer and Information Science (ICIS)*, May 2008. doi:[10.1109/ICIS.2008.75](https://doi.org/10.1109/ICIS.2008.75)
 79. James E. Johnson, David E. Langworthy, Leslie Lamport, and Friedrich H. Vogt: “[Formal Specification of a Web Services Protocol](#),” at *1st International Workshop on Web Services and Formal Methods (WS-FM)*, February 2004. doi:[10.1016/j.entcs.2004.02.022](https://doi.org/10.1016/j.entcs.2004.02.022)
 80. Dale Skeen: “[Nonblocking Commit Protocols](#),” at *ACM International Conference on Management of Data (SIGMOD)*, April 1981. doi:[10.1145/582318.582339](https://doi.org/10.1145/582318.582339)
 81. Gregor Hohpe: “[Your Coffee Shop Doesn't Use Two-Phase Commit](#),” *IEEE Software*, volume 22, number 2, pages 64–66, March 2005. doi:[10.1109/MS.2005.52](https://doi.org/10.1109/MS.2005.52)
 82. Pat Helland: “[Life Beyond Distributed Transactions: An Apostate's Opinion](#),” at *3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2007.
 83. Jonathan Oliver: “[My Beef with MSDTC and Two-Phase Commits](#),” blog.jonathanoliver.com, April 4, 2011.
 84. Oren Eini (Ahende Rahien): “[The Fallacy of Distributed Transactions](#),” ayende.com, July 17, 2014.
 85. Clemens Vasters: “[Transactions in Windows Azure \(with Service Bus\) – An Email Discussion](#),” vasters.com, July 30, 2012.
 86. “[Understanding Transactionality in Azure](#),” NServiceBus Documentation, Particular Software, 2015.
 87. Randy Wigginton, Ryan Lowe, Marcos Albe, and Fernando Ipar: “[Distributed Transactions in MySQL](#),” at *MySQL Conference and Expo*, April 2013.
 88. Mike Spille: “[XA Exposed, Part I](#),” jroller.com, April 3, 2004.
 89. Ajmer Dharwal: “[Orphaned MSDTC Transactions \(-2 spids\)](#),” eraofdata.com, December 12, 2008.
 90. Paul Randal: “[Real World Story of DBCC PAGE Saving the Day](#),” sqlskills.com, June 19, 2013.
 91. “[in-doubt xact resolution Server Configuration Option](#),” SQL Server 2016 documentation, Microsoft, Inc., 2016.
 92. Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer: “[Consensus in the Presence of Partial Synchrony](#),” *Journal of the ACM*, volume 35, number 2, pages 288–323, April 1988. doi:[10.1145/42282.42283](https://doi.org/10.1145/42282.42283)
 93. Miguel Castro and Barbara H. Liskov: “[Practical Byzantine Fault Tolerance and Proactive Recovery](#),” *ACM Transactions on Computer Systems*, volume 20, number 4, pages 396–461, November 2002. doi:[10.1145/571637.571640](https://doi.org/10.1145/571637.571640)
 94. Brian M. Oki and Barbara H. Liskov: “[Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems](#),” at *7th ACM Symposium on Principles of Distributed Computing (PODC)*, August 1988. doi:[10.1145/62546.62549](https://doi.org/10.1145/62546.62549)
 95. Barbara H. Liskov and James Cowling: “[Viewstamped Replication Revisited](#),” Massachusetts Institute of Technology, Tech Report MIT-CSAIL-TR-2012-021, July 2012.
 96. Leslie Lamport: “[The Part-Time Parliament](#),” *ACM Transactions on Computer Systems*, volume 16, number 2, pages 133–169, May 1998. doi:[10.1145/279227.279229](https://doi.org/10.1145/279227.279229)
 97. Leslie Lamport: “[Paxos Made Simple](#),” *ACM SIGACT News*, volume 32, number 4, pages 51–58, December 2001.
 98. Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone: “[Paxos Made Live – An Engineering Perspective](#),” at *26th ACM Symposium on Principles of Distributed Computing (PODC)*, June 2007.
 99. Robbert van Renesse: “[Paxos Made Moderately Complex](#),” cs.cornell.edu, March 2011.
 00. Diego Ongaro: “[Consensus: Bridging Theory and Practice](#),” PhD Thesis, Stanford University, August 2014.
 01. Heidi Howard, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft: “[Raft Refloated: Do We Have Consensus?](#),” *ACM SIGOPS Operating Systems Review*, volume 49, number 1, pages 12–21, January 2015. doi:[10.1145/2723872.2723876](https://doi.org/10.1145/2723872.2723876)
 02. André Medeiros: “[ZooKeeper's Atomic Broadcast Protocol: Theory and Practice](#),” Aalto University School of Science, March 20, 2012.
 03. Robbert van Renesse, Nicolas Schiper, and Fred B. Schneider: “[Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab](#),” *IEEE Transactions on Dependable and Secure Computing*, volume 12, number 4, pages 472–484, September 2014. doi:[10.1109/TDSC.2014.2355848](https://doi.org/10.1109/TDSC.2014.2355848)

04. Will Portnoy: “[Lessons Learned from Implementing Paxos](#),” blog.willportnoy.com, June 14, 2012.
 05. Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman: “[Flexible Paxos: Quorum Intersection Revisited](#),” [arXiv:1608.06696](https://arxiv.org/abs/1608.06696), August 24, 2016.
 06. Heidi Howard and Jon Crowcroft: “[Coracle: Evaluating Consensus at the Internet Edge](#),” at *Annual Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, August 2015.
[doi:10.1145/2829988.2790010](https://doi.org/10.1145/2829988.2790010)
 07. Kyle Kingsbury: “[Call Me Maybe: Elasticsearch 1.5.0](#),” aphyr.com, April 27, 2015.
 08. Ivan Kelly: “[BookKeeper Tutorial](#),” github.com, October 2014.
 09. Camille Fournier: “[Consensus Systems for the Skeptical Architect](#),” at *Craft Conference*, Budapest, Hungary, April 2015.
 10. Kenneth P. Birman: “[A History of the Virtual Synchrony Replication Model](#),” in *Replication: Theory and Practice*, Springer LNCS volume 5959, chapter 6, pages 91–120, 2010. ISBN: 978-3-642-11293-5, [doi:10.1007/978-3-642-11294-2_6](https://doi.org/10.1007/978-3-642-11294-2_6)
-

上一章	目錄	下一章
第八章：分散式系統的麻煩	設計資料密集型應用	第三部分：衍生資料

第三部分：衍生資料

在本書的 [第一部分](#) 和 [第二部分](#) 中，我們自底向上地把所有關於分散式資料庫的主要考量都過了一遍。從資料在磁碟上的佈局，一直到出現故障時分散式系統一致性的侷限。但所有的討論都假定了應用中只用了一種資料庫。

現實世界中的資料系統往往更為複雜。大型應用程式經常需要以多種方式訪問和處理資料，沒有一個數據庫可以同時滿足所有這些不同的需求。因此應用程式通常組合使用多種元件：資料儲存、索引、快取、分析系統等等，並實現在這些元件中移動資料的機制。

本書的最後一部分，會研究將多個不同資料系統（可能有著不同資料模型，並針對不同的訪問模式進行最佳化）整合為一個協調一致的應用架構時，會遇到的問題。軟體供應商經常會忽略這一方面的生態建設，並聲稱他們的產品能夠滿足你的所有需求。在現實世界中，整合不同的系統是實際應用中最重要的事情之一。

記錄系統和衍生資料系統

從高層次上看，儲存和處理資料的系統可以分為兩大類：

- 記錄系統（System of record）

記錄系統，也被稱為 **真相源**（source of truth），持有資料的權威版本。當新的資料進入時（例如，使用者輸入）首先會記錄在這裡。每個事實正正好表示一次（表示通常是 **正規化的**，即 normalized）。如果其他系統和記錄系統之間存在任何差異，那麼記錄系統中的值是正確的（根據定義）。

- 衍生資料系統（Derived data systems）

衍生系統中的資料，通常是另一個系統中的現有資料以某種方式進行轉換或處理的結果。如果丟失衍生資料，可以從原始來源重新建立。典型的例子是 **快取**（cache）：如果資料在快取中，就可以由快取提供服務；如果快取不包含所需資料，則降級由底層資料庫提供。非規範化的值，索引和物化檢視亦屬此類。在推薦系統中，預測彙總資料通常衍生自使用者日誌。

從技術上講，衍生資料是 **冗餘的**（redundant），因為它重複了已有的資訊。但是衍生資料對於獲得良好的只讀查詢效能通常是至關重要的。它通常是非規範化的。可以從單個源頭衍生出多個不同的資料集，使你能從不同的“視角”洞察資料。

並不是所有的系統都在其架構中明確區分 **記錄系統** 和 **衍生資料系統**，但是這是一種有用的區分方式，因為它明確了系統中的資料流：系統的哪一部分具有哪些輸入和哪些輸出，以及它們如何相互依賴。

大多數資料庫，儲存引擎和查詢語言，本質上既不是記錄系統也不是衍生系統。資料庫只是一個工具：如何使用它取決於你自己。**記錄系統**和**衍生資料系統**之間的區別不在於工具，而在於應用程式中的使用方式。

透過梳理資料的衍生關係，可以清楚地理解一個令人困惑的系統架構。這將貫穿本書的這一部分。

章節概述

我們將從 [第十章](#) 開始，研究例如 MapReduce 這樣 **面向批處理**（batch-oriented）的資料流系統。對於建設大規模資料系統，我們將看到，它們提供了優秀的工具和思想。[第十一章](#) 將把這些思想應用到 **流式資料**（data streams）中，使我們能用更低的延遲完成同樣的任務。[第十二章](#) 將對本書進行總結，探討如何使用這些工具來構建可靠，可伸縮和可維護的應用。

索引

1. 批處理

- 2. 流處理
 - 3. 資料系統的未來
-

上一章	目錄	下一章
第九章：一致性與共識	設計資料密集型應用	第十章：批處理

第十章：批處理



帶有太強個人色彩的系統無法成功。當最初的設計完成並且相對穩定時，不同的人們以自己的方式進行測試，真正的考驗才開始。

—— 高德納

[TOC]

在本書的前兩部分中，我們討論了很多關於 **請求** 和 **查詢** 以及相應的 **響應** 或 **結果**。許多現有資料系統中都採用這種資料處理方式：你傳送請求指令，一段時間後（我們期望）系統會給出一個結果。資料庫、快取、搜尋索引、Web 伺服器以及其他一些系統都以這種方式工作。

像這樣的 **線上 (online)** 系統，無論是瀏覽器請求頁面還是呼叫遠端 API 的服務，我們通常認為請求是由人類使用者觸發的，並且正在等待響應。他們不應該等太久，所以我們非常關注系統的響應時間（請參閱 “[描述效能](#)”）。

Web 和越來越多的基於 HTTP/REST 的 API 使互動的請求 / 響應風格變得如此普遍，以至於很容易將其視為理所當然。但我們應該記住，這不是構建系統的唯一方式，其他方法也有其優點。我們來看看三種不同型別的系統：

- **服務（線上系統）**

服務等待客戶的請求或指令到達。每收到一個，服務會試圖儘快處理它，併發回一個響應。響應時間通常是服務效能的主要衡量指標，可用性通常非常重要（如果客戶端無法訪問服務，使用者可能會收到錯誤訊息）。

- **批處理系統（離線系統）**

一個批處理系統有大量的輸入資料，跑一個 **作業 (job)** 來處理它，並生成一些輸出資料，這往往需要一段時間（從幾分鐘到幾天），所以通常不會有使用者等待作業完成。相反，批次作業通常會定期執行（例如，每天一次）。批處理作業的主要效能衡量標準通常是吞吐量（處理特定大小的輸入所需的時間）。本章中討論的就是批處理。

- **流處理系統（準實時系統）**

流處理介於線上和離線（批處理）之間，所以有時候被稱為 **準實時 (near-real-time)** 或 **準線上 (nearline)** 處理。像批處理系統一樣，流處理消費輸入併產生輸出（並不需要響應請求）。但是，流式作業在事件發生後不久就會對事件進行操作，而批處理作業則需等待固定的一組輸入資料。這種差異使流處理系統比起批處理系統具有更低的延遲。由於流處理基於批處理，我們將在 [第十一章](#) 討論它。

正如我們將在本章中看到的那樣，批處理是構建可靠、可伸縮和可維護應用程式的重要組成部分。例如，2004 年釋出的批處理演算法 Map-Reduce（可能被過分熱情地）被稱為“造就 Google 大規模可伸縮性的演算法”【2】。隨後在各種開源資料系統中得到應用，包括 Hadoop、CouchDB 和 MongoDB。

與多年前為資料倉庫開發的並行處理系統【3,4】相比，MapReduce 是一個相當低級別的程式設計模型，但它使得在商用硬體上能進行的處理規模邁上一個新的臺階。雖然 MapReduce 的重要性正在下降【5】，但它仍然值得去理解，因為它描繪了一幅關於批處理為什麼有用，以及如何做到有用的清晰圖景。

實際上，批處理是一種非常古老的計算方式。早在可程式設計數字計算機誕生之前，打孔卡製表機（例如 1890 年美國人口普查【6】中使用的霍爾里斯機）實現了半機械化的批處理形式，從大量輸入中彙總計算。Map-Reduce 與 1940 年代和 1950 年代廣泛用於商業資料處理的機電 IBM 卡片分類機器有著驚人的相似之處【7】。正如我們所說，歷史總是在不斷重複自己。

在本章中，我們將瞭解 MapReduce 和其他一些批處理演算法和框架，並探索它們在現代資料系統中的作用。但首先我們將看看使用標準 Unix 工具的資料處理。即使你已經熟悉了它們，Unix 的哲學也值得一讀，Unix 的思想和經驗教訓可以遷移到大規模、異構的分散式資料系統中。

使用 Unix 工具的批處理

我們從一個簡單的例子開始。假設你有一臺 Web 伺服器，每次處理請求時都會在日誌檔案中附加一行。例如，使用 nginx 預設的訪問日誌格式，日誌的一行可能如下所示：

```
216.58.210.78 - - [27/Feb/2015:17:55:11 +0000] "GET /css/typography.css HTTP/1.1"
200 3377 "http://martin.kleppmann.com/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_5)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/40.0.2214.115 Safari/537.36"
```

(實際上這只是一行，分成多行只是為了便於閱讀。) 這一行中有很多資訊。為了解釋它，你需要了解日誌格式的定義，如下所示：

```
$remote_addr $remote_user [$time_local] "$request"
$status $body_bytes_sent "$http_referer" "$http_user_agent"
```

日誌的這一行表明在 UTC 時間的 2015 年 2 月 27 日 17 點 55 分 11 秒，伺服器從客戶端 IP 地址 216.58.210.78 接收到對檔案 /css/typography.css 的請求。使用者沒有認證，所以 \$remote_user 被設定為連字元 (-)。響應狀態是 200 (即請求成功)，響應的大小是 3377 位元組。網頁瀏覽器是 Chrome 40，它載入了這個檔案是因為該檔案在網址為 http://martin.kleppmann.com/ 的頁面中被引用到了。

簡單日誌分析

很多工具可以從這些日誌檔案生成關於網站流量的漂亮的報告，但為了練手，讓我們使用基本的 Unix 功能建立自己的工具。例如，假設你想在你的網站上找到五個最受歡迎的網頁。則可以在 Unix shell 中這樣做：ⁱ

ⁱ 有些人認為 cat 這裡並沒有必要，因為輸入檔案可以直接作為 awk 的引數。但這種寫法讓線性管道更為顯眼。 ↪

```
cat /var/log/nginx/access.log | #1
awk '{print $7}' | #2
sort | #3
uniq -c | #4
sort -r -n | #5
head -n 5 #6
```

1. 讀取日誌檔案
2. 將每一行按空格分割成不同的欄位，每行只輸出第七個欄位，恰好是請求的 URL。在我們的例子中是 /css/typography.css 。
3. 按字母順序排列請求的 URL 列表。如果某個 URL 被請求過 n 次，那麼排序後，檔案將包含連續重複出現 n 次的該 URL 。
4. uniq 命令透過檢查兩個相鄰的行是否相同來過濾掉輸入中的重複行。 -c 則表示還要輸出一個計數器：對於每個不同的 URL，它會報告輸入中出現該 URL 的次數。
5. 第二種排序按每行起始處的數字 (-n) 排序，這是 URL 的請求次數。然後逆序 (-r) 返回結果，大的數字在前。
6. 最後，只輸出前五行 (-n 5)，並丟棄其餘的。該系列命令的輸出如下所示：

```
4189 /favicon.ico
3631 /2013/05/24/improving-security-of-ssh-private-keys.html
2124 /2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html
1369 /
915 /css/typography.css
```

如果你不熟悉 Unix 工具，上面的命令列可能看起來有點吃力，但是它非常強大。它能在幾秒鐘內處理幾 GB 的日誌檔案，並且你可以根據需要輕鬆修改命令。例如，如果要從報告中省略 CSS 檔案，可以將 awk 引數更改為 '\$7 !~ /\.css\$/ {print \$7}'，如果想統計最多的客戶端 IP 地址，可以把 awk 引數改為 '{print \$1}'，等等。

我們不會在這裡詳細探索 Unix 工具，但是它非常值得學習。令人驚訝的是，使用 awk、sed、grep、sort、uniq 和 xargs 的組合，可以在幾分鐘內完成許多資料分析，並且它們的效能相當的好【8】。

命令鏈與自定義程式

除了 Unix 命令鏈，你還可以寫一個簡單的程式來做同樣的事情。例如在 Ruby 中，它可能看起來像這樣：

```
counts = Hash.new(0)          # 1
File.open('/var/log/nginx/access.log') do |file|
  file.each do |line|
    url = line.split[6]      # 2
    counts[url] += 1        # 3
  end
end

top5 = counts.map{|url, count| [count, url] }.sort.reverse[0...5] # 4
top5.each{|count, url| puts "#{count} #{url}" }                 # 5
```

1. `counts` 是一個儲存計數器的雜湊表，儲存了每個 URL 被瀏覽的次數，預設為 0。
2. 逐行讀取日誌，抽取每行第七個被空格分隔的欄位為 URL（這裡的陣列索引是 6，因為 Ruby 的陣列索引從 0 開始計數）
3. 將日誌當前行中 URL 對應的計數器值加一。
4. 按計數器值（降序）對雜湊表內容進行排序，並取前五位。
5. 打印出前五個條目。

這個程式並不像 Unix 管道那樣簡潔，但是它的可讀性很強，喜歡哪一種屬於口味的問題。但兩者除了表面上的差異之外，執行流程也有很大差異，如果你在大檔案上執行此分析，則會變得明顯。

排序 VS 記憶體中的聚合

Ruby 指令碼在記憶體中儲存了一個 URL 的雜湊表，將每個 URL 對映到它出現的次數。Unix 管道沒有這樣的雜湊表，而是依賴於對 URL 列表的排序，在這個 URL 列表中，同一個 URL 的只是簡單地重複出現。

哪種方法更好？這取決於你有多少個不同的 URL。對於大多數中小型網站，你可能可以為所有不同網址提供一個計數器（假設我們使用 1GB 記憶體）。在此例中，作業的工作集（working set，即作業需要隨機訪問的記憶體大小）僅取決於不同 URL 的數量：如果日誌中只有單個 URL，重複出現一百萬次，則散列表所需的空間表就只有一個 URL 加上一個計數器的大小。當工作集足夠小時，記憶體散列表表現良好，甚至在效能較差的膝上型電腦上也可以正常工作。

另一方面，如果作業的工作集大於可用記憶體，則排序方法的優點是可以高效地使用磁碟。這與我們在“[SSTables 和 LSM 樹](#)”中討論過的原理是一樣的：資料塊可以在記憶體中排序並作為段檔案寫入磁碟，然後多個排序好的段可以合併為一個更大的排序檔案。歸併排序具有在磁碟上執行良好的順序訪問模式。（請記住，針對順序 I/O 進行最佳化是[第三章](#)中反覆出現的主題，相同的模式在此重現）

GNU Coreutils (Linux) 中的 `sort` 程式透過溢位至磁碟的方式來自動應對大於記憶體的資料集，並能同時使用多個 CPU 核進行並行排序【9】。這意味著我們之前看到的簡單的 Unix 命令鏈很容易伸縮至大資料集，且不會耗盡記憶體。瓶頸可能是從磁碟讀取輸入檔案的速度。

Unix哲學

我們可以非常容易地使用前一個例子中的一系列命令來分析日誌檔案，這並非巧合：事實上，這實際上是 Unix 的關鍵設計思想之一，而且它直至今天也仍然令人訝異地重要。讓我們更深入地研究一下，以便從 Unix 中借鑑一些想法【10】。

Unix 管道的發明者道格 · 麥克羅伊 (Doug McIlroy) 在 1964 年首先描述了這種情況【11】：“我們需要一種類似園藝膠管的方式來拼接程式——當我們需要將訊息從一個程式傳遞另一個程式時，直接接上去就行。I/O 應該也按照這種方式進行”。水管的類比仍然在生效，透過管道連線程式的想法成為了現在被稱為 **Unix 哲學** 的一部分——這一組設計原

則在 Unix 使用者與開發者之間流行起來，該哲學在 1978 年表述如下【12,13】：

1. 讓每個程式都做好一件事。要做一件新的工作，寫一個新程式，而不是透過新增“功能”讓老程式複雜化。
2. 期待每個程式的輸出成為另一個程式的輸入。不要將無關資訊混入輸出。避免使用嚴格的列資料或二進位制輸入格式。不要堅持互動式輸入。
3. 設計和構建軟體時，即使是作業系統，也讓它們能夠儘早地被試用，最好在幾周內完成。不要猶豫，扔掉笨拙的部分，重建它們。
4. 優先使用工具來減輕程式設計任務，即使必須曲線救國編寫工具，且在用完後很可能要扔掉大部分。

這種方法——自動化，快速原型設計，增量式迭代，對實驗友好，將大型專案分解成可管理的塊——聽起來非常像今天的敏捷開發和 DevOps 運動。奇怪的是，四十年來變化不大。

`sort` 工具是一個很好的例子。可以說它比大多數程式語言標準庫中的實現（它們不會利用磁碟或使用多執行緒，即使這樣做有很大好處）要更好。然而，單獨使用 `sort` 幾乎沒什麼用。它只能與其他 Unix 工具（如 `uniq`）結合使用。

像 `bash` 這樣的 Unix shell 可以讓我們輕鬆地將這些小程式組合成為強大資料處理任務。儘管這些程式中有很多是由不同人群編寫的，但它們可以靈活地結合在一起。Unix 如何實現這種可組合性？

統一的介面

如果你希望一個程式的輸出成為另一個程式的輸入，那意味著這些程式必須使用相同的資料格式——換句話說，一個相容的介面。如果你希望能夠將任何程式的輸出連線到任何程式的輸入，那意味著所有程式必須使用相同的 I/O 介面。

在 Unix 中，這種介面是一個 檔案 (file，更準確地說，是一個檔案描述符)。一個檔案只是一串有序的位元組序列。因為這是一個非常簡單的介面，所以可以使用相同的介面來表示許多不同的東西：檔案系統上的真實檔案，到另一個程序 (Unix 套接字，`stdin`, `stdout`) 的通訊通道，裝置驅動程式（比如 `/dev/audio` 或 `/dev/lp0`），表示 TCP 連線的套接字，等等。很容易將這些設計視為理所當然的，但實際上能讓這些差異巨大的東西共享一個統一的介面是非常厲害的，這使得它們可以很容易地連線在一起ⁱⁱ。

ⁱⁱ. 統一介面的另一個例子是 URL 和 HTTP，這是 Web 的基石。一個 URL 標識一個網站上的一個特定的東西（資源），你可以連結到任何其他網站的任何網址。具有網路瀏覽器的使用者因此可以透過跟隨連結在網站之間無縫跳轉，即使伺服器可能由完全不相關的組織維護。這個原則現在似乎非常明顯，但它卻是網路取能取得今天成就的關鍵。之前的系統並不是那麼統一：例如，在公告板系統 (BBS) 時代，每個系統都有自己的電話號碼和波特率配置。從一個 BBS 到另一個 BBS 的引用必須以電話號碼和調變解調器設定的形式；使用者將不得不掛斷，撥打其他 BBS，然後手動找到他們正在尋找的資訊。直接連結到另一個 BBS 內的一些內容當時是不可能的。[←](#)

按照慣例，許多（但不是全部）Unix 程式將這個位元組序列視為 ASCII 文字。我們的日誌分析示例使用了這個事實：`awk`、`sort`、`uniq` 和 `head` 都將它們的輸入檔案視為由 `\n`（換行符，ASCII `0x0A`）字元分隔的記錄列表。`\n` 的選擇是任意的——可以說，ASCII 記錄分隔符 `0x1E` 本來就是一個更好的選擇，因為它是為了這個目的而設計的【14】，但是無論如何，所有這些程式都使用相同的記錄分隔符允許它們互操作。

每條記錄（即一行輸入）的解析則更加模糊。Unix 工具通常透過空白或製表符將行分割成欄位，但也使用 CSV（逗號分隔），管道分隔和其他編碼。即使像 `xargs` 這樣一個相當簡單的工具也有六個命令列選項，用於指定如何解析輸入。

ASCII 文字的統一介面大多數時候都能工作，但它不是很優雅：我們的日誌分析示例使用 `{print $7}` 來提取網址，這樣可讀性不是很好。在理想的世界中可能是 `{print $request_url}` 或類似的東西。我們稍後會回顧這個想法。

儘管幾十年後還不夠完美，但統一的 Unix 介面仍然是非常出色的設計。沒有多少軟體能像 Unix 工具一樣互動組合的這麼好：你不能透過自定義分析工具輕鬆地將電子郵件帳戶的內容和線上購物歷史記錄以管道傳送至電子表格中，並將結果釋出到社交網路或維基。今天，像 Unix 工具一樣流暢地執行程式是一種例外，而不是規範。

即使是具有 相同資料模型 的資料庫，將資料從一種資料庫匯出再匯入到另一種資料庫也並不容易。缺乏整合導致了資料的 [巴爾幹化](#)。

[譯註](#)ⁱ. [巴爾幹化](#) (Balkanization) 是一個常帶有貶義的地緣政治學術語，其定義為：一個國家或政區分裂成多

個互相敵對的國家或政區的過程。 ↵

邏輯與佈線相分離

Unix 工具的另一個特點是使用標準輸入（`stdin`）和標準輸出（`stdout`）。如果你執行一個程式，而不指定任何其他的東西，標準輸入來自鍵盤，標準輸出指向螢幕。但是，你也可以從檔案輸入和 / 或將輸出重定向到檔案。管道允許你將一個程序的標準輸出附加到另一個程序的標準輸入（有個小記憶體緩衝區，而不需要將整個中間資料流寫入磁碟）。

如果需要，程式仍然可以直接讀取和寫入檔案，但 Unix 方法在程式不關心特定的檔案路徑、只使用標準輸入和標準輸出時效果最好。這允許 shell 使用者以任何他們想要的方式連線輸入和輸出；該程式不知道或不關心輸入來自哪裡以及輸出到哪裡。（人們可以說這是一種 **松耦合** (*loose coupling*)，**晚期繫結** (*late binding*) 【15】或 **控制反轉** (*inversion of control*) 【16】）。將輸入 / 輸出佈線與程式邏輯分開，可以將小工具組合成更大的系統。

你甚至可以編寫自己的程式，並將它們與作業系統提供的工具組合在一起。你的程式只需要從標準輸入讀取輸入，並將輸出寫入標準輸出，它就可以加入資料處理的管道中。在日誌分析示例中，你可以編寫一個將 Usage-Agent 字串轉換為更靈敏的瀏覽器識別符號，或者將 IP 地址轉換為國家程式碼的工具，並將其插入管道。`sort` 程式並不關心它是否與作業系統的另一部分或者你寫的程式通訊。

但是，使用 `stdin` 和 `stdout` 能做的事情是有限的。需要多個輸入或輸出的程式雖然可能，卻非常棘手。你沒法將程式的輸出管道連線至網路連線中【17,18】ⁱⁱⁱ。如果程式直接開啟檔案進行讀取和寫入，或者將另一個程式作為子程序啟動，或者開啟網路連線，那麼 I/O 的佈線就取決於程式本身了。它仍然可以被配置（例如透過命令列選項），但在 Shell 中對輸入和輸出進行佈線的靈活性就少了。

ⁱⁱⁱ. 除了使用一個單獨的工具，如 `netcat` 或 `curl`。 Unix 起初試圖將所有東西都表示為檔案，但是 BSD 套接字 API 偏離了這個慣例【17】。研究用作業系統 Plan 9 和 Inferno 在使用檔案方面更加一致：它們將 TCP 連線表示為 `/net/tcp` 中的檔案【18】。 ↵

透明度和實驗

使 Unix 工具如此成功的部分原因是，它們使檢視正在發生的事情變得非常容易：

- Unix 命令的輸入檔案通常被視為不可變的。這意味著你可以隨意執行命令，嘗試各種命令列選項，而不會損壞輸入檔案。
- 你可以在任何時候結束管道，將管道輸出到 `less`，然後檢視它是否具有預期的形式。這種檢查能力對除錯非常有用。
- 你可以將一個流水線階段的輸出寫入檔案，並將該檔案用作下一階段的輸入。這使你可以重新啟動後面的階段，而無需重新執行整個管道。

因此，與關係資料庫的查詢最佳化器相比，即使 Unix 工具非常簡單，但仍然非常有用，特別是對於實驗而言。

然而，Unix 工具的最大侷限在於它們只能在一臺機器上執行——而 Hadoop 這樣的工具即應運而生。

MapReduce和分散式檔案系統

MapReduce 有點像 Unix 工具，但分佈在數千臺機器上。像 Unix 工具一樣，它相當簡單粗暴，但令人驚異地管用。一個 MapReduce 作業可以和一個 Unix 程序相類比：它接受一個或多個輸入，併產生一個或多個輸出。

和大多數 Unix 工具一樣，執行 MapReduce 作業通常不會修改輸入，除了生成輸出外沒有任何副作用。輸出檔案以連續的方式一次性寫入（一旦寫入檔案，不會修改任何現有的檔案部分）。

雖然 Unix 工具使用 `stdin` 和 `stdout` 作為輸入和輸出，但 MapReduce 作業在分散式檔案系統上讀寫檔案。在 Hadoop 的 MapReduce 實現中，該檔案系統被稱為 **HDFS** (*Hadoop 分散式檔案系統*)，一個 Google 檔案系統 (GFS) 的開源實現【19】。

除 HDFS 外，還有各種其他分散式檔案系統，如 GlusterFS 和 Quantcast File System (QFS) 【20】。諸如 Amazon S3、Azure Blob 儲存和 OpenStack Swift 【21】等物件儲存服務在很多方面都是相似的^{iv}。在本章中，我們將主要使用 HDFS 作為示例，但是這些原則適用於任何分散式檔案系統。

^{iv} 一個不同之處在於，對於 HDFS，可以將計算任務安排在儲存特定檔案副本的計算機上執行，而物件儲存通常將儲存和計算分開。如果網路頻寬是一個瓶頸，從本地磁碟讀取有效能優勢。但是請注意，如果使用糾刪碼 (Erasure Coding)，則會丟失區域性，因為來自多臺機器的資料必須進行合併以重建原始檔案【20】。[←](#)

與網路連線儲存 (NAS) 和儲存區域網路 (SAN) 架構的共享磁碟方法相比，HDFS 基於 無共享 原則（請參閱 [第二部分](#) 的介紹）。共享磁碟儲存由集中式儲存裝置實現，通常使用定製硬體和專用網路基礎設施（如光纖通道）。而另一方面，無共享方法不需要特殊的硬體，只需要透過傳統資料中心網路連線的計算機。

HDFS 在每臺機器上運行了一個守護程序，它對外暴露網路服務，允許其他節點訪問儲存在該機器上的檔案（假設資料中心中的每臺通用計算機都掛載著一些磁碟）。名為 **NameNode** 的中央伺服器會跟蹤哪個檔案塊儲存在哪臺機器上。因此，HDFS 在概念上建立了一個大型檔案系統，可以使用所有執行有守護程序的機器的磁碟。

為了容忍機器和磁碟故障，檔案塊被複制到多臺機器上。複製可能意味著多個機器上的相同資料的多個副本，如 [第五章](#) 中所述，或者諸如 Reed-Solomon 碼這樣的糾刪碼方案，它能以比完全複製更低的儲存開銷來支援恢復丟失的資料【20,22】。這些技術與 RAID 相似，後者可以在連線到同一臺機器的多個磁碟上提供冗餘；區別在於在分散式檔案系統中，檔案訪問和複製是在傳統的資料中心網路上完成的，沒有特殊的硬體。

HDFS 的可伸縮性已經很不錯了：在撰寫本書時，最大的 HDFS 部署執行在上萬臺機器上，總儲存容量達數百 PB【23】。如此大的規模已經變得可行，因為使用商品硬體和開源軟體的 HDFS 上的資料儲存和訪問成本遠低於在專用儲存裝置上支援同等容量的成本【24】。

MapReduce 作業執行

MapReduce 是一個程式設計框架，你可以使用它編寫程式碼來處理 HDFS 等分散式檔案系統中的大型資料集。理解它的最簡單方法是參考 “[簡單日誌分析](#)” 中的 Web 伺服器日誌分析示例。MapReduce 中的資料處理模式與此示例非常相似：

1. 讀取一組輸入檔案，並將其分解成 **記錄 (records)**。在 Web 伺服器日誌示例中，每條記錄都是日誌中的一行（即 `\n` 是記錄分隔符）。
2. 呼叫 Mapper 函式，從每條輸入記錄中提取一對鍵值。在前面的例子中，Mapper 函式是 `awk '{print $7}'`：它提取 URL (`$7`) 作為鍵，並將值留空。
3. 按鍵排序所有的鍵值對。在日誌的例子中，這由第一個 `sort` 命令完成。
4. 呼叫 Reducer 函式遍歷排序後的鍵值對。如果同一個鍵出現多次，排序使它們在列表中相鄰，所以很容易組合這些值而不必在記憶體中保留很多狀態。在前面的例子中，Reducer 是由 `uniq -c` 命令實現的，該命令使用相同的鍵來統計相鄰記錄的數量。

這四個步驟可以作為一個 MapReduce 作業執行。步驟 2 (Map) 和 4 (Reduce) 是你編寫自定義資料處理程式碼的地方。步驟 1 (將檔案分解成記錄) 由輸入格式解析器處理。步驟 3 中的排序步驟隱含在 MapReduce 中——你不必編寫它，因為 Mapper 的輸出始終在送往 Reducer 之前進行排序。

要建立 MapReduce 作業，你需要實現兩個回撥函式，Mapper 和 Reducer，其行為如下（請參閱 “[MapReduce 查詢](#)”）：

- **Mapper**

Mapper 會在每條輸入記錄上呼叫一次，其工作是從輸入記錄中提取鍵值。對於每個輸入，它可以生成任意數量的鍵值對（包括 None）。它不會保留從一個輸入記錄到下一個記錄的任何狀態，因此每個記錄都是獨立處理的。

- **Reducer**

MapReduce 框架拉取由 Mapper 生成的鍵值對，收集屬於同一個鍵的所有值，並在這組值上迭代呼叫 Reducer。Reducer 可以產生輸出記錄（例如相同 URL 的出現次數）。

在 Web 伺服器日誌的例子中，我們在第 5 步中有第二個 `sort` 命令，它按請求數對 URL 進行排序。在 MapReduce 中，如果你需要第二個排序階段，則可以透過編寫第二個 MapReduce 作業並將第一個作業的輸出用作第二個作業的輸入來實現它。這樣看來，Mapper 的作用是將資料放入一個適合排序的表單中，並且 Reducer 的作用是處理已排序的資料。

分散式執行MapReduce

MapReduce 與 Unix 命令管道的主要區別在於，MapReduce 可以在多臺機器上並行執行計算，而無需編寫程式碼來顯式處理並行問題。Mapper 和 Reducer 一次只能處理一條記錄；它們不需要知道它們的輸入來自哪裡，或者輸出去往什麼地方，所以框架可以處理在機器之間移動資料的複雜性。

在分散式計算中可以使用標準的 Unix 工具作為 Mapper 和 Reducer【25】，但更常見的是，它們被實現為傳統程式語言的函式。在 Hadoop MapReduce 中，Mapper 和 Reducer 都是實現特定介面的 Java 類。在 MongoDB 和 CouchDB 中，Mapper 和 Reducer 都是 JavaScript 函式（請參閱“[MapReduce 查詢](#)”）。

[圖 10-1](#) 顯示了 Hadoop MapReduce 作業中的資料流。其並行化基於分割槽（請參閱 [第六章](#)）：作業的輸入通常是 HDFS 中的一個目錄，輸入目錄中的每個檔案或檔案塊都被認為是一個單獨的分割槽，可以單獨處理 map 任務（[圖 10-1](#) 中的 m1, m2 和 m3 標記）。

每個輸入檔案的大小通常是數百兆位元組。MapReduce 排程器（圖中未顯示）試圖在其中一臺儲存輸入檔案副本的機器上執行每個 Mapper，只要該機器有足夠的備用 RAM 和 CPU 資源來執行 Mapper 任務【26】。這個原則被稱為 [將計算放在資料附近](#)【27】：它節省了透過網路複製輸入檔案的開銷，減少網路負載並增加區域性。

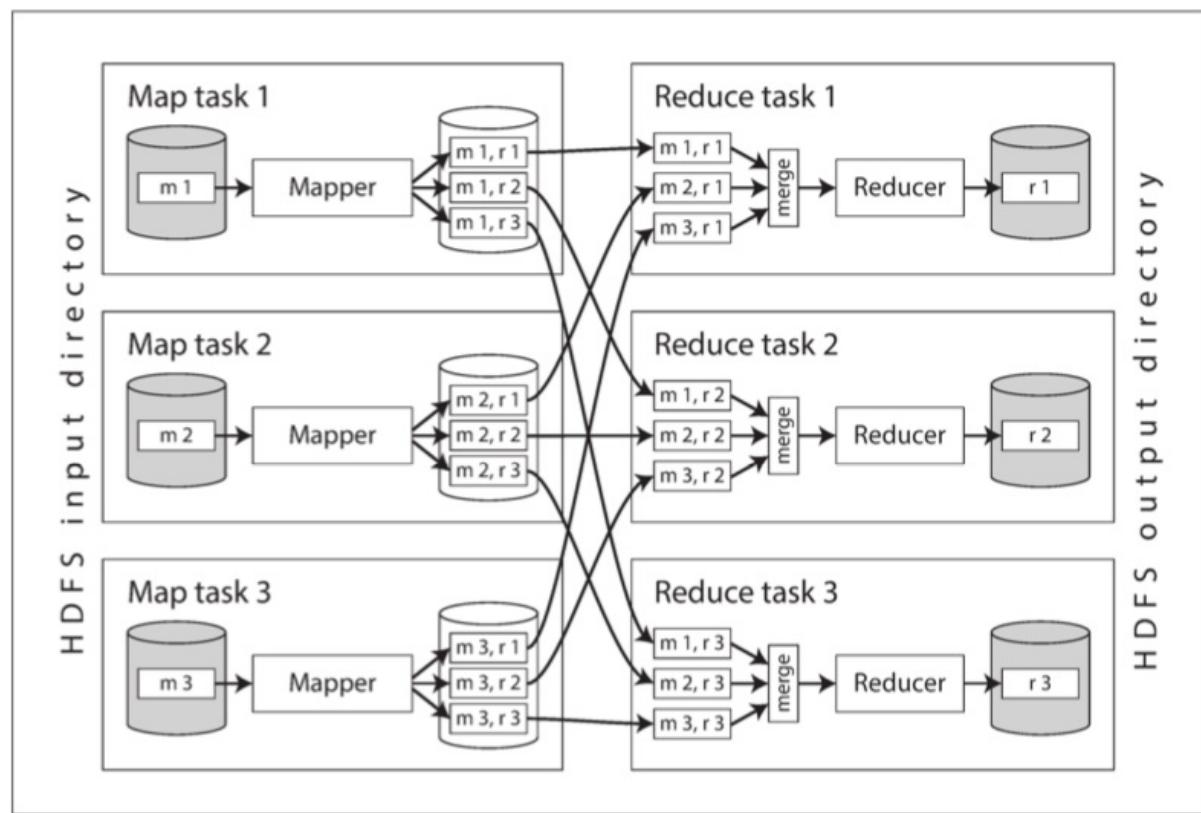


圖 10-1 具有三個 Mapper 和三個 Reducer 的 MapReduce 任務

在大多數情況下，應該在 Mapper 任務中執行的應用程式碼在將要執行它的機器上還不存在，所以 MapReduce 框架首先將程式碼（例如 Java 程式中的 JAR 檔案）複製到適當的機器。然後啟動 Map 任務並開始讀取輸入檔案，一次將一條記錄傳入 Mapper 回撥函式。Mapper 的輸出由鍵值對組成。

計算的 Reduce 端也被分割槽。雖然 Map 任務的數量由輸入檔案塊的數量決定，但 Reducer 的任務的數量是由作業作者配置的（它可以不同於 Map 任務的數量）。為了確保具有相同鍵的所有鍵值對最終落在相同的 Reducer 處，框架使用鍵的雜湊值來確定哪個 Reduce 任務應該接收到特定的鍵值對（請參閱“[根據鍵的雜湊分割槽](#)”）。

鍵值對必須進行排序，但資料集可能太大，無法在單臺機器上使用常規排序演算法進行排序。相反，分類是分階段進行的。首先每個 Map 任務都按照 Reducer 對輸出進行分割槽。每個分割槽都被寫入 Mapper 程式的本地磁碟，使用的技術與我們在“[SSTables 與 LSM 樹](#)”中討論的類似。

只要當 Mapper 讀取完輸入檔案，並寫完排序後的輸出檔案，MapReduce 排程器就會通知 Reducer 可以從該 Mapper 開始獲取輸出檔案。Reducer 連線到每個 Mapper，並下載自己相應分割槽的有序鍵值對檔案。按 Reducer 分割槽，排序，從 Mapper 向 Reducer 複製分割槽資料，這一整個過程被稱為 **混洗 (shuffle)** 【26】（一個容易混淆的術語——不像洗牌，在 MapReduce 中的混洗沒有隨機性）。

Reduce 任務從 Mapper 獲取檔案，並將它們合併在一起，並保留有序特性。因此，如果不同的 Mapper 生成了鍵相同的記錄，則在 Reducer 的輸入中，這些記錄將會相鄰。

Reducer 呼叫時會收到一個鍵，和一個迭代器作為引數，迭代器會順序地掃過所有具有該鍵的記錄（因為在某些情況可能無法完全放入記憶體中）。Reducer 可以使用任意邏輯來處理這些記錄，並且可以生成任意數量的輸出記錄。這些輸出記錄會寫入分散式檔案系統上的檔案中（通常是在跑 Reducer 的機器本地磁碟上留一份，並在其他機器上留幾份副本）。

MapReduce工作流

單個 MapReduce 作業可以解決的問題範圍很有限。以日誌分析為例，單個 MapReduce 作業可以確定每個 URL 的頁面瀏覽次數，但無法確定最常見的 URL，因為這需要第二輪排序。

因此將 MapReduce 作業連結成為 **工作流 (workflow)** 中是極為常見的，例如，一個作業的輸出成為下一個作業的輸入。Hadoop MapReduce 框架對工作流沒有特殊支援，所以這個鏈是透過目錄名隱式實現的：第一個作業必須將其輸出配置為 HDFS 中的指定目錄，第二個作業必須將其輸入配置為從同一個目錄。從 MapReduce 框架的角度來看，這是兩個獨立的作業。

因此，被連結的 MapReduce 作業並沒有那麼像 Unix 命令管道（它直接將一個程序的輸出作為另一個程序的輸入，僅用一個很小的記憶體緩衝區）。它更像是一系列命令，其中每個命令的輸出寫入臨時檔案，下一個命令從臨時檔案中讀取。這種設計有利也有弊，我們將在“[物化中間狀態](#)”中討論。

只有當作業成功完成後，批處理作業的輸出才會被視為有效的（MapReduce 會丟棄失敗作業的部分輸出）。因此，工作流中的一項作業只有在先前的作業——即生產其輸入的作業——成功完成後才能開始。為了處理這些作業之間的依賴，有很多針對 Hadoop 的工作流排程器被開發出來，包括 Oozie、Azkaban、Luigi、Airflow 和 Pinball 【28】。

這些排程程式還具有管理功能，在維護大量批處理作業時非常有用。在構建推薦系統時，由 50 到 100 個 MapReduce 作業組成的工作流是常見的【29】。而在大型組織中，許多不同的團隊可能執行不同的作業來讀取彼此的輸出。工具支援對於管理這樣複雜的資料流而言非常重要。

Hadoop 的各種高階工具（如 Pig 【30】、Hive 【31】、Cascading 【32】、Crunch 【33】和 FlumeJava 【34】）也能自動佈線組裝多個 MapReduce 階段，生成合適的工作流。

Reduce側連線與分組

我們在 [第二章](#) 中討論了資料模型和查詢語言的連線，但是我們還沒有深入探討連線是如何實現的。現在是我們再次撿起這條線索的時候了。

在許多資料集中，一條記錄與另一條記錄存在關聯是很常見的：關係模型中的 **外來鍵**，文件模型中的 **文件引用** 或圖模型中的 **邊**。當你需要同時訪問這一關聯的兩側（持有引用的記錄與被引用的記錄）時，連線就是必須的。正如 [第二章](#) 所討論的，非規範化可以減少對連線的需求，但通常無法將其完全移除^v。

^v. 我們在本書中討論的連線通常是等值連線，即最常見的連線型別，其中記錄透過與其他記錄在特定欄位（例如

ID) 中具有相同值相關聯。有些資料庫支援更通用的連線型別，例如使用小於運算子而不是等號運算子，但是我們沒有地方來講這些東西。 ↵

在資料庫中，如果執行只涉及少量記錄的查詢，資料庫通常會使用索引來快速定位感興趣的記錄（請參閱[第三章](#)）。如果查詢涉及到連線，則可能涉及到查詢多個索引。然而 MapReduce 沒有索引的概念——至少在通常意義上沒有。

當 MapReduce 作業被賦予一組檔案作為輸入時，它讀取所有這些檔案的全部內容；資料庫會將這種操作稱為全表掃描。如果你只想讀取少量的記錄，則全表掃描與索引查詢相比，代價非常高昂。但是在分析查詢中（請參閱“[事務處理還是分析？](#)”），通常需要計算大量記錄的聚合。在這種情況下，特別是如果能在多臺機器上並行處理時，掃描整個輸入可能是相當合理的事情。

當我們在批處理的語境中討論連線時，我們指的是在資料集中解析某種關聯的全量存在。例如我們假設一個作業是同時處理所有使用者的資料，而非僅僅是為某個特定使用者查詢資料（而這能透過索引更高效地完成）。

示例：使用者活動事件分析

[圖 10-2](#) 細出了一個批處理作業中連線的典型例子。左側是事件日誌，描述登入使用者在網站上做的事情（稱為活動事件，即 activity events，或點選流資料，即 clickstream data），右側是使用者資料庫。你可以將此示例看作是星型模式的一部分（請參閱“[星型和雪花型：分析的模式](#)”）：事件日誌是事實表，使用者資料庫是其中的一個維度。

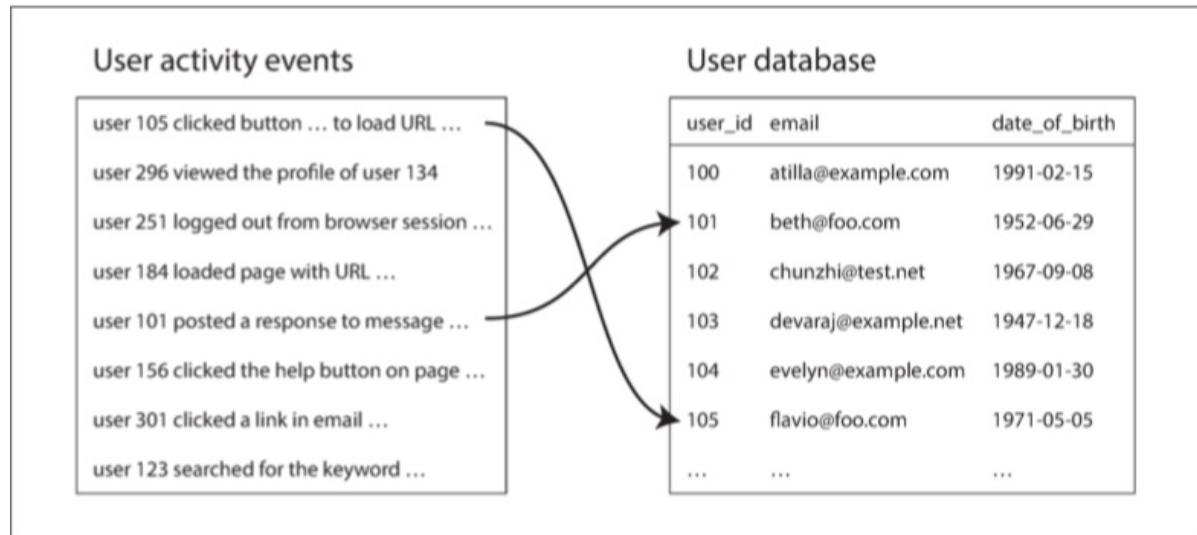


圖 10-2 使用者行為日誌與使用者檔案的連線

分析任務可能需要將使用者活動與使用者檔案資訊相關聯：例如，如果檔案包含使用者的年齡或出生日期，系統就可以確定哪些頁面更受哪些年齡段的使用者歡迎。然而活動事件僅包含使用者 ID，而沒有包含完整的使用者檔案資訊。在每個活動事件中嵌入這些檔案資訊很可能會非常浪費。因此，活動事件需要與使用者檔案資料庫相連線。

實現這一連線的最簡單方法是，逐個遍歷活動事件，併為每個遇到的使用者 ID 查詢使用者資料庫（在遠端伺服器上）。這是可能的，但是它的效能可能會非常差：處理吞吐量將受限於受資料庫伺服器的往返時間，本地快取的有效性很大程度上取決於資料的分佈，並行執行大量查詢可能會輕易壓垮資料庫【35】。

為了在批處理過程中實現良好的吞吐量，計算必須（儘可能）限於單臺機器上進行。為待處理的每條記錄發起隨機訪問的網路請求實在是太慢了。而且，查詢遠端資料庫意味著批處理作業變為非確定的（nondeterministic），因為遠端資料庫中的資料可能會改變。

因此，更好的方法是獲取使用者資料庫的副本（例如，使用 ETL 程序從資料庫備份中提取資料，請參閱“[資料倉庫](#)”），並將它和使用者行為日誌放入同一個分散式檔案系統中。然後你可以將使用者資料庫儲存在 HDFS 中的一組檔案中，而使用者活動記錄儲存在另一組檔案中，並能用 MapReduce 將所有相關記錄集中到同一個地方進行高效處理。

排序合併連線

回想一下，Mapper 的目的是從每個輸入記錄中提取一對鍵值。在 圖 10-2 的情況下，這個鍵就是使用者 ID：一組 Mapper 會掃過活動事件（提取使用者 ID 作為鍵，活動事件作為值），而另一組 Mapper 將會掃過使用者資料庫（提取使用者 ID 作為鍵，使用者的出生日期作為值）。這個過程如 圖 10-3 所示。

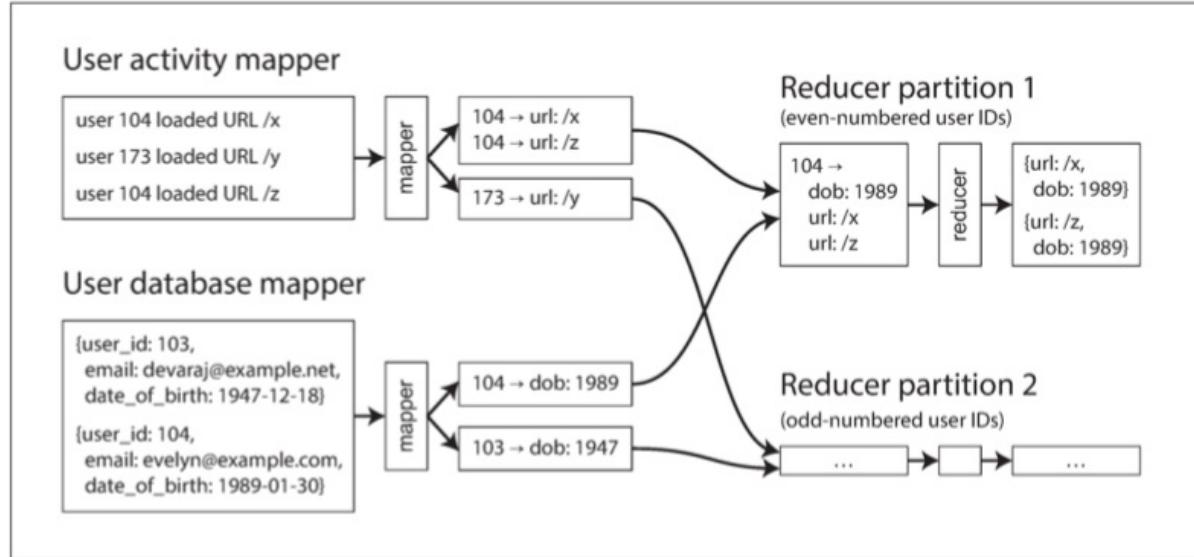


圖 10-3 在使用者 ID 上進行的 Reduce 端連線。如果輸入資料集分割槽為多個檔案，則每個分割槽都會被多個 Mapper 並行處理

當 MapReduce 框架透過鍵對 Mapper 輸出進行分割槽，然後對鍵值對進行排序時，效果是具有相同 ID 的所有活動事件和使用者記錄在 Reducer 輸入中彼此相鄰。Map-Reduce 作業甚至可以也讓這些記錄排序，使 Reducer 總能先看到來自使用者資料庫的記錄，緊接著是按時間戳順序排序的活動事件——這種技術被稱為 **二次排序 (secondary sort)** [26]。

然後 Reducer 可以容易地執行實際的連線邏輯：每個使用者 ID 都會被呼叫一次 Reducer 函式，且因為二次排序，第一個值應該是來自使用者資料庫的出生日期記錄。Reducer 將出生日期儲存在區域性變數中，然後使用相同的使用者 ID 遍歷活動事件，輸出 **已觀看網址** 和 **觀看者年齡** 的結果對。隨後的 Map-Reduce 作業可以計算每個 URL 的檢視者年齡分佈，並按年齡段進行聚集。

由於 Reducer 一次處理一個特定使用者 ID 的所有記錄，因此一次只需要將一條使用者記錄儲存在記憶體中，而不需要透過網路發出任何請求。這個演算法被稱為 **排序合併連線 (sort-merge join)**，因為 Mapper 的輸出是按鍵排序的，然後 Reducer 將來自連線兩側的有序記錄列表合併在一起。

把相關資料放在一起

在排序合併連線中，Mapper 和排序過程確保了所有對特定使用者 ID 執行連線操作的必須資料都被放在同一個地方：單次呼叫 Reducer 的地方。預先排好了所有需要的資料，Reducer 可以是相當簡單的單執行緒程式碼，能夠以高吞吐量和與低記憶體開銷掃過這些記錄。

這種架構可以看做，Mapper 將“訊息”傳送給 Reducer。當一個 Mapper 發出一個鍵值對時，這個鍵的作用就像值應該傳遞到的目標地址。即使鍵只是一個任意的字串（不是像 IP 地址和埠號那樣的實際的網路地址），它表現的就像一個地址：所有具有相同鍵的鍵值對將被傳遞到相同的目標（一次 Reducer 的呼叫）。

使用 MapReduce 程式設計模型，能將計算的物理網路通訊層面（從正確的機器獲取資料）從應用邏輯中剝離出來（獲取資料後執行處理）。這種分離與資料庫的典型用法形成了鮮明對比，從資料庫中獲取資料的請求經常出現在應用程式碼內部 [36]。由於 MapReduce 處理了所有的網路通訊，因此它也避免了讓應用程式碼去擔心部分故障，例如另一個節點的崩潰：MapReduce 在不影響應用邏輯的情況下能透明地重試失敗的任務。

分組

除了連線之外，“把相關資料放在一起”的另一種常見模式是，按某個鍵對記錄分組（如 SQL 中的 GROUP BY 子句）。所有帶有相同鍵的記錄構成一個組，而下一步往往是在每個組內進行某種聚合操作，例如：

- 統計每個組中記錄的數量（例如在統計 PV 的例子中，在 SQL 中表示為 `COUNT(*)` 聚合）
- 對某個特定欄位求和（SQL 中的 `SUM(fieldname)`）
- 按某種分級函式取出排名前 k 條記錄。

使用 MapReduce 實現這種分組操作的最簡單方法是設定 Mapper，以便它們生成的鍵值對使用所需的分組鍵。然後分割槽和排序過程將所有具有相同分割槽鍵的記錄導向同一個 Reducer。因此在 MapReduce 之上實現分組和連線看上去非常相似。

分組的另一個常見用途是整理特定使用者會話的所有活動事件，以找出使用者進行的一系列操作（稱為 **會話化 (sessionization)** 【37】）。例如，可以使用這種分析來確定顯示新版網站的使用者是否比那些顯示舊版本的使用者更有購買慾（A/B 測試），或者計算某個營銷活動是否值得。

如果你有多個 Web 伺服器處理使用者請求，則特定使用者的活動事件很可能分散在各個不同的伺服器的日誌檔案中。你可以透過使用會話 cookie，使用者 ID 或類似的識別符號作為分組鍵，以將特定使用者的所有活動事件放在一起來實現會話化，與此同時，不同使用者的事件仍然散佈在不同的分割槽中。

處理偏斜

如果存在與單個鍵關聯的大量資料，則“將具有相同鍵的所有記錄放到相同的位置”這種模式就被破壞了。例如在社交網路中，大多數使用者可能會與幾百人有連線，但少數名人可能有數百萬的追隨者。這種不成比例的活動資料庫記錄被稱為 **關鍵物件 (linchpin object)** 【38】或 **熱鍵 (hot key)**。

在單個 Reducer 中收集與某個名人相關的所有活動（例如他們釋出內容的回覆）可能導致嚴重的 **偏斜**（也稱為 **熱點**，即 **hot spot**）——也就是說，一個 Reducer 必須比其他 Reducer 處理更多的記錄（請參閱“[負載偏斜與熱點消除](#)”）。由於 MapReduce 作業只有在所有 Mapper 和 Reducer 都完成時才完成，所有後續作業必須等待最慢的 Reducer 才能啟動。

如果連線的輸入存在熱鍵，可以使用一些演算法進行補償。例如，Pig 中的 **偏斜連線 (skewed join)** 方法首先執行一個抽樣作業（Sampling Job）來確定哪些鍵是熱鍵【39】。連線實際執行時，Mapper 會將熱鍵的關聯記錄 **隨機**（相對於傳統 MapReduce 基於鍵雜湊的確定性方法）傳送到幾個 Reducer 之一。對於另外一側的連線輸入，與熱鍵相關的記錄需要被複制到 **所有** 處理該鍵的 Reducer 上【40】。

這種技術將處理熱鍵的工作分散到多個 Reducer 上，這樣可以使其更好地並行化，代價是需要將連線另一側的輸入記錄複製到多個 Reducer 上。Crunch 中的 **分片連線 (sharded join)** 方法與之類似，但需要顯式指定熱鍵而不是使用抽樣作業。這種技術也非常類似於我們在“[負載偏斜與熱點消除](#)”中討論的技術，使用隨機化來緩解分割槽資料庫中的熱點。

Hive 的偏斜連線最佳化採取了另一種方法。它需要在表格元資料中顯式指定熱鍵，並將與這些鍵相關的記錄單獨存放，與其它檔案分開。當在該表上執行連線時，對於熱鍵，它會使用 Map 端連線（請參閱下一節）。

當按照熱鍵進行分組並聚合時，可以將分組分兩個階段進行。第一個 MapReduce 階段將記錄傳送到隨機 Reducer，以便每個 Reducer 只對熱鍵的子集執行分組，為每個鍵輸出一個更緊湊的中間聚合結果。然後第二個 MapReduce 作業將所有來自第一階段 Reducer 的中間聚合結果合併為每個鍵一個值。

Map側連線

上一節描述的連線演算法在 Reducer 中執行實際的連線邏輯，因此被稱為 **Reduce 側連線**。Mapper 扮演著預處理輸入資料的角色：從每個輸入記錄中提取鍵值，將鍵值對分配給 Reducer 分割槽，並按鍵排序。

Reduce 側方法的優點是不需要對輸入資料做任何假設：無論其屬性和結構如何，Mapper 都可以對其預處理以備連線。然而不利的一面是，排序，複製至 Reducer，以及合併 Reducer 輸入，所有這些操作可能開銷巨大。當資料透過 MapReduce 階段時，資料可能需要落盤好幾次，取決於可用的記憶體緩衝區【37】。

另一方面，如果你能對輸入資料作出某些假設，則透過使用所謂的 Map 側連線來加快連線速度是可行的。這種方法使用了一個裁減掉 Reducer 與排序的 MapReduce 作業，每個 Mapper 只是簡單地從分散式檔案系統中讀取一個輸入檔案塊，然後將輸出檔案寫入檔案系統，僅此而已。

廣播雜湊連線

適用於執行 Map 端連線的最簡單場景是大資料集與小資料集連線的情況。要點在於小資料集需要足夠小，以便可以將其全部載入到每個 Mapper 的記憶體中。

例如，假設在 圖 10-2 的情況下，使用者資料庫小到足以放進記憶體中。在這種情況下，當 Mapper 啟動時，它可以首先將使用者資料庫從分散式檔案系統讀取到記憶體中的散列表中。完成此操作後，Mapper 可以掃描使用者活動事件，並簡單地在散列表中查詢每個事件的使用者 ID^{vi}。

^{vi}. 這個例子假定散列表中的每個鍵只有一個條目，這對使用者資料庫（使用者 ID 唯一標識一個使用者）可能是正確的。通常，雜湊表可能需要包含具有相同鍵的多個條目，而連線運算子將對每個鍵輸出所有的匹配。 ↵

參與連線的較大輸入的每個檔案塊各有一個 Mapper（在 圖 10-2 的例子中活動事件是較大的輸入）。每個 Mapper 都會將較小輸入整個載入到記憶體中。

這種簡單有效的演算法被稱為 **廣播雜湊連線 (broadcast hash join)**：廣播一詞反映了這樣一個事實，每個連線較大輸入端分割槽的 Mapper 都會將較小輸入端資料集整個讀入記憶體中（所以較小輸入實際上“廣播”到較大資料的所有分割槽上），雜湊一詞反映了它使用一個散列表。Pig（名為“複製連結（replicated join）”），Hive（“MapJoin”），Cascading 和 Crunch 支援這種連線。它也被諸如 Impala 的資料倉庫查詢引擎使用【41】。

除了將較小的連線輸入載入到記憶體散列表中，另一種方法是將較小輸入儲存在本地磁碟上的只讀索引中【42】。索引中經常使用的部分將保留在作業系統的頁面快取中，因而這種方法可以提供與記憶體散列表幾乎一樣快的隨機查詢效能，但實際上並不需要資料集能放入記憶體中。

分割槽雜湊連線

如果 Map 側連線的輸入以相同的方式進行分割槽，則雜湊連線方法可以獨立應用於每個分割槽。在 圖 10-2 的情況中，你可以根據使用者 ID 的最後一位十進位制數字來對活動事件和使用者資料庫進行分割槽（因此連線兩側各有 10 個分割槽）。例如，Mapper3 首先將所有具有以 3 結尾的 ID 的使用者載入到散列表中，然後掃描 ID 為 3 的每個使用者的所有活動事件。

如果分割槽正確無誤，可以確定的是，所有你可能需要連線的記錄都落在同一個編號的分割槽中。因此每個 Mapper 只需要從輸入兩端各讀取一個分割槽就足夠了。好處是每個 Mapper 都可以在記憶體散列表中少放點資料。

這種方法只有當連線兩端輸入有相同的分割槽數，且兩側的記錄都是使用相同的鍵與相同的雜湊函式做分割槽時才適用。如果輸入是由之前執行過這種分組的 MapReduce 作業生成的，那麼這可能是一個合理的假設。

分割槽雜湊連線在 Hive 中稱為 **Map 側桶連線 (bucketed map joins)**【37】。

Map側合併連線

如果輸入資料集不僅以相同的方式進行分割槽，而且還基於相同的鍵進行 排序，則可適用另一種 Map 側連線的變體。在這種情況下，輸入是否小到能放入記憶體並不重要，因為這時候 Mapper 同樣可以執行歸併操作（通常由 Reducer 執行）的歸併操作：按鍵遞增的順序依次讀取兩個輸入檔案，將具有相同鍵的記錄配對。

如果能進行 Map 側合併連線，這通常意味著前一個 MapReduce 作業可能一開始就已經把輸入資料做了分割槽並進行了排序。原則上這個連線就可以在前一個作業的 Reduce 階段進行。但使用獨立的僅 Map 作業有時也是合適的，例如，分好區且排好序的中間資料集可能還會用於其他目的。

MapReduce工作流與Map側連線

當下遊作業使用 MapReduce 連線的輸出時，選擇 Map 側連線或 Reduce 側連線會影響輸出的結構。Reduce 側連線的輸出是按照 [連線鍵](#) 進行分割槽和排序的，而 Map 端連線的輸出則按照與較大輸入相同的方式進行分割槽和排序（因為無論是使用分割槽連線還是廣播連線，連線較大輸入端的每個檔案塊都會啟動一個 Map 任務）。

如前所述，Map 側連線也對輸入資料集的大小，有序性和分割槽方式做出了更多假設。在最佳化連線策略時，瞭解分散式檔案系統中資料集的物理佈局變得非常重要：僅僅知道編碼格式和資料儲存目錄的名稱是不夠的；你還必須知道資料是按哪些鍵做的分割槽和排序，以及分割槽的數量。

在 Hadoop 生態系統中，這種關於資料集分割槽的元資料通常在 HCatalog 和 Hive Metastore 中維護 [【37】](#)。

批處理工作流的輸出

我們已經說了很多用於實現 MapReduce 工作流的演算法，但卻忽略了一個重要的問題：這些處理完成之後的最終結果是什麼？我們最開始為什麼要跑這些作業？

在資料庫查詢的場景中，我們將事務處理（OLTP）與分析兩種目的區分開來（請參閱[“事務處理還是分析？”](#)）。我們看到，OLTP 查詢通常根據鍵查詢少量記錄，使用索引，並將其呈現給使用者（比如在網頁上）。另一方面，分析查詢通常會掃描大量記錄，執行分組與聚合，輸出通常有著報告的形式：顯示某個指標隨時間變化的圖表，或按照某種排位取前 10 項，或將一些數字細化為子類。這種報告的消費者通常是需要做出商業決策的分析師或經理。

批處理放哪裡合適？它不屬於事務處理，也不是分析。它和分析比較接近，因為批處理通常會掃過輸入資料集的絕大部分。然而 MapReduce 作業工作流與用於分析目的的 SQL 查詢是不同的（請參閱[“Hadoop 與分散式資料庫的對比”](#)）。批處理過程的輸出通常不是報表，而是一些其他型別的結構。

建立搜尋索引

Google 最初使用 MapReduce 是為其搜尋引擎建立索引，其實現為由 5 到 10 個 MapReduce 作業組成的工作流 [【1】](#)。雖然 Google 後來也不僅僅是為這個目的而使用 MapReduce [【43】](#)，但如果從構建搜尋索引的角度來看，更能幫助理解 MapReduce。（直至今日，Hadoop MapReduce 仍然是為 Lucene/Solr 構建索引的好方法 [【44】](#)）

我們在[“全文搜尋和模糊索引”](#)中簡要地瞭解了 Lucene 這樣的全文搜尋索引是如何工作的：它是一個檔案（關鍵詞字典），你可以在其中高效地查詢特定關鍵字，並找到包含該關鍵字的所有文件 ID 列表（文章列表）。這是一種非常簡化的看法——實際上，搜尋索引需要各種額外資訊，以便根據相關性對搜尋結果進行排名、糾正拼寫錯誤、解析同義詞等等——但這個原則是成立的。

如果需要對一組固定文件執行全文搜尋，則批處理是一種構建索引的高效方法：Mapper 根據需要對文件集合進行分割槽，每個 Reducer 構建該分割槽的索引，並將索引檔案寫入分散式檔案系統。構建這樣的文件分割槽索引（請參閱[“分割槽與次級索引”](#)）並行處理效果拔群。

由於按關鍵字查詢搜尋索引是隻讀操作，因而這些索引檔案一旦建立就是不可變的。

如果索引的文件集合發生更改，一種選擇是定期重跑整個索引工作流，並在完成後用新的索引檔案批次替換以前的索引檔案。如果只有少量的文件發生了變化，這種方法的計算成本可能會很高。但它的優點是索引過程很容易理解：文件進，索引出。

另一個選擇是，可以增量建立索引。如[第三章](#)中討論的，如果要在索引中新增，刪除或更新文件，Lucene 會寫新的段檔案，並在後臺非同步合併壓縮段檔案。我們將在[第十一章](#)中看到更多這種增量處理。

鍵值儲存作為批處理輸出

搜尋索引只是批處理工作流可能輸出的一個例子。批處理的另一個常見用途是構建機器學習系統，例如分類器（比如垃圾郵件過濾器，異常檢測，影象識別）與推薦系統（例如，你可能認識的人，你可能感興趣的產品或相關的搜尋 [【29】](#)）。

這些批處理作業的輸出通常是某種資料庫：例如，可以透過給定使用者 ID 查詢該使用者推薦好友的資料庫，或者可以透過產品 ID 查詢相關產品的資料庫 [【45】](#)。

這些資料庫需要被處理使用者請求的 Web 應用所查詢，而它們通常是獨立於 Hadoop 基礎設施的。那麼批處理過程的輸出如何回到 Web 應用可以查詢的資料庫中呢？

最直接的選擇可能是，直接在 Mapper 或 Reducer 中使用你最愛的資料庫的客戶端庫，並從批處理作業直接寫入資料庫伺服器，一次寫入一條記錄。它能工作（假設你的防火牆規則允許從你的 Hadoop 環境直接訪問你的生產資料庫），但這並不是一個好主意，出於以下幾個原因：

- 正如前面在連線的上下文中討論的那樣，為每條記錄發起一個網路請求，要比批處理任務的正常吞吐量慢幾個數量級。即使客戶端庫支援批處理，效能也可能很差。
- MapReduce 作業經常並行執行許多工。如果所有 Mapper 或 Reducer 都同時寫入相同的輸出資料庫，並以批處理的預期速率工作，那麼該資料庫很可能被輕易壓垮，其查詢效能可能變差。這可能會導致系統其他部分的執行問題【35】。
- 通常情況下，MapReduce 為作業輸出提供了一個乾淨利落的“全有或全無”保證：如果作業成功，則結果就是每個任務恰好執行一次所產生的輸出，即使某些任務失敗且必須一路重試。如果整個作業失敗，則不會生成輸出。然而從作業內部寫入外部系統，會產生外部可見的副作用，這種副作用是不能以這種方式被隱藏的。因此，你不得不去操心對其他系統可見的部分完成的作業結果，並需要理解 Hadoop 任務嘗試與預測執行的複雜性。

更好的解決方案是在批處理作業 **內** 建立一個全新的資料庫，並將其作為檔案寫入分散式檔案系統中作業的輸出目錄，就像上節中的搜尋索引一樣。這些資料檔案一旦寫入就是不可變的，可以批次載入到處理只讀查詢的伺服器中。不少鍵值儲存都支援在 MapReduce 作業中構建資料庫檔案，包括 Voldemort 【46】、Terrapin 【47】、ElephantDB 【48】和 HBase 批次載入 【49】。

構建這些資料庫檔案是 MapReduce 的一種好用法：使用 Mapper 提取出鍵並按該鍵排序，已經完成了構建索引所必需的大量工作。由於這些鍵值儲存大多都是隻讀的（檔案只能由批處理作業一次性寫入，然後就不可變），所以資料結構非常簡單。比如它們就不需要預寫式日誌（WAL，請參閱“讓 B 樹更可靠”）。

將資料載入到 Voldemort 時，伺服器將繼續用舊資料檔案服務請求，同時將新資料檔案從分散式檔案系統複製到伺服器的本地磁碟。一旦複製完成，伺服器會自動將查詢切換到新檔案。如果在這個過程中出現任何問題，它可以輕易回滾至舊檔案，因為它們仍然存在而且不可變【46】。

批處理輸出的哲學

本章前面討論過的 Unix 哲學（“[Unix 哲學](#)”）鼓勵以顯式指明資料流的方式進行實驗：程式讀取輸入並寫入輸出。在這一過程中，輸入保持不變，任何先前的輸出都被新輸出完全替換，且沒有其他副作用。這意味著你可以隨心所欲地重新執行一個命令，略做改動或進行除錯，而不會攪亂系統的狀態。

MapReduce 作業的輸出處理遵循同樣的原理。透過將輸入視為不可變且避免副作用（如寫入外部資料庫），批處理作業不僅實現了良好的效能，而且更容易維護：

- 如果在程式碼中引入了一個錯誤，而輸出錯誤或損壞了，則可以簡單地回滾到程式碼的先前版本，然後重新執行該作業，輸出將重新被糾正。或者，甚至更簡單，你可以將舊的輸出儲存在不同的目錄中，然後切換回原來的目錄。具有讀寫事務的資料庫沒有這個屬性：如果你部署了錯誤的程式碼，將錯誤的資料寫入資料庫，那麼回滾程式碼將無法修復資料庫中的資料。（能夠從錯誤程式碼中恢復的概念被稱為 **人類容錯**（human fault tolerance）【50】）
- 由於回滾很容易，比起在錯誤意味著不可挽回的傷害的環境，功能開發進展能快很多。這種 **最小化不可逆性**（minimizing irreversibility）的原則有利於敏捷軟體開發【51】。
- 如果 Map 或 Reduce 任務失敗，MapReduce 框架將自動重新排程，並在同樣的輸入上再次執行它。如果失敗是由程式碼中的錯誤造成的，那麼它會不斷崩潰，並最終導致作業在幾次嘗試之後失敗。但是如果故障是由於臨時問題導致的，那麼故障就會被容忍。因為輸入不可變，這種自動重試是安全的，而失敗任務的輸出會被 MapReduce 框架丟棄。
- 同一組檔案可用作各種不同作業的輸入，包括計算指標的監控作業並且評估作業的輸出是否具有預期的性質（例如，將其與前一次執行的輸出進行比較並測量差異）。
- 與 Unix 工具類似，MapReduce 作業將邏輯與佈線（配置輸入和輸出目錄）分離，這使得關注點分離，可以重用程式碼：一個團隊可以專注實現一個做好一件事的作業；而其他團隊可以決定何時何地執行這項作業。

在這些領域，在 Unix 上表現良好的設計原則似乎也適用於 Hadoop，但 Unix 和 Hadoop 在某些方面也有所不同。例如，因為大多數 Unix 工具都假設輸入輸出是無型別文字檔案，所以它們必須做大量的輸入解析工作（本章開頭的日誌分析示例使用 `{print $7}` 來提取 URL）。在 Hadoop 上可以透過使用更結構化的檔案格式消除一些低價值的語法轉換：比如 Avro（請參閱“[Avro](#)”）和 Parquet（請參閱“[列式儲存](#)”）經常使用，因為它們提供了基於模式的高效編碼，並允許模式隨時推移而演進（見 [第四章](#)）。

Hadoop與分散式資料庫的對比

正如我們所看到的，Hadoop 有點像 Unix 的分散式版本，其中 HDFS 是檔案系統，而 MapReduce 是 Unix 程序的怪異實現（總是在 Map 階段和 Reduce 階段執行 `sort` 工具）。我們瞭解瞭如何在這些原語的基礎上實現各種連線和分組操作。

當 MapReduce 論文發表時【1】，它從某種意義上來說——並不新鮮。我們在前幾節中討論的所有處理和並行連線演算法已經在十多年前所謂的 **大規模並行處理（MPP，massively parallel processing）** 資料庫中實現了【3,40】。比如 Gamma database machine、Teradata 和 Tandem NonStop SQL 就是這方面的先驅【52】。

最大的區別是，MPP 資料庫專注於在一組機器上並行執行分析 SQL 查詢，而 MapReduce 和分散式檔案系統【19】的組合則更像是一個可以執行任意程式的通用作業系統。

儲存多樣性

資料庫要求你根據特定的模型（例如關係或文件）來構造資料，而分散式檔案系統中的檔案只是位元組序列，可以使用任何資料模型和編碼來編寫。它們可能是資料庫記錄的集合，但同樣可以是文字、影象、影片、感測器讀數、稀疏矩陣、特徵向量、基因組序列或任何其他型別的資料。

說白了，Hadoop 開放了將資料不加區分地轉儲到 HDFS 的可能性，允許後續再研究如何進一步處理【53】。相比之下，在將資料匯入資料庫專有儲存格式之前，MPP 資料庫通常需要對資料和查詢模式進行仔細的前期建模。

在純粹主義者看來，這種仔細的建模和匯入似乎是可取的，因為這意味著資料庫的使用者有更高質量的資料來處理。然而實踐經驗表明，簡單地使資料快速可用——即使它很古怪，難以使用，使用原始格式——也通常要比事先決定理想資料模型要更有價值【54】。

這個想法與資料倉庫類似（請參閱“[資料倉庫](#)”）：將大型組織的各個部分的資料集中在一起是很有價值的，因為它可以跨越以前相互分離的資料集進行連線。MPP 資料庫所要求的謹慎模式設計拖慢了集中式資料收集速度；以原始形式收集資料，稍後再操心模式的設計，能使資料收集速度加快（有時被稱為“[資料湖（data lake）](#)”或“[企業資料中心（enterprise data hub）](#)”【55】）。

不加區分的資料轉儲轉移瞭解釋資料的負擔：資料集的生產者不再需要強制將其轉化為標準格式，資料的解釋成為消費者的問題（[讀時模式](#) 方法【56】；請參閱“[文件模型中的模式靈活性](#)”）。如果生產者和消費者是不同優先順序的不同團隊，這可能是一種優勢。甚至可能存在一個理想的資料模型，對於不同目的有不同的合適視角。以原始形式簡單地轉儲資料，可以允許多種這樣的轉換。這種方法被稱為 **壽司原則（sushi principle）**：“原始資料更好”【57】。

因此，Hadoop 經常被用於實現 ETL 過程（請參閱“[資料倉庫](#)”）：事務處理系統中的資料以某種原始形式轉儲到分散式檔案系統中，然後編寫 MapReduce 作業來清理資料，將其轉換為關係形式，並將其匯入 MPP 資料倉庫以進行分析。資料建模仍然在進行，但它在一個單獨的步驟中進行，與資料收集相解耦。這種解耦是可行的，因為分散式檔案系統支援以任何格式編碼的資料。

處理模型的多樣性

MPP 資料庫是單體的，緊密整合的軟體，負責磁碟上的儲存佈局，查詢計劃，排程和執行。由於這些元件都可以針對資料庫的特定需求進行調整和最佳化，因此整個系統可以在其設計針對的查詢型別上取得非常好的效能。而且，SQL 查詢語言允許以優雅的語法表達查詢，而無需編寫程式碼，可以在業務分析師使用的視覺化工具（例如 Tableau）中訪問到。

另一方面，並非所有型別的處理都可以合理地表達為 SQL 查詢。例如，如果要構建機器學習和推薦系統，或者使用相關性排名模型的全文搜尋索引，或者執行影象分析，則很可能需要更一般的資料處理模型。這些型別的處理通常是特別針對特定應用的（例如機器學習的特徵工程，機器翻譯的自然語言模型，欺詐預測的風險評估函式），因此它們不可避免地需要編寫程式碼，而不僅僅是查詢。

MapReduce 使工程師能夠輕鬆地在大型資料集上執行自己的程式碼。如果你有 HDFS 和 MapReduce，那麼你可以在它之上建立一個 SQL 查詢執行引擎，事實上這正是 Hive 專案所做的【31】。但是，你也可以編寫許多其他形式的批處理，這些批處理不必非要用 SQL 查詢表示。

隨後，人們發現 MapReduce 對於某些型別的處理而言侷限性很大，表現很差，因此在 Hadoop 之上其他各種處理模型也被開發出來（我們將在“[MapReduce 之後](#)”中看到其中一些）。只有兩種處理模型，SQL 和 MapReduce，還不夠，需要更多不同的模型！而且由於 Hadoop 平臺的開放性，實施一整套方法是可行的，而這在單體 MPP 資料庫的範疇內是不可能的【58】。

至關重要的是，這些不同的處理模型都可以在共享的單個機器叢集上執行，所有這些機器都可以訪問分散式檔案系統上的相同檔案。在 Hadoop 方式中，不需要將資料匯入到幾個不同的專用系統中進行不同型別的處理：系統足夠靈活，可以支援同一個叢集內不同的工作負載。不需要移動資料，使得從資料中挖掘價值變得容易得多，也使採用新的處理模型容易的多。

Hadoop 生態系統包括隨機訪問的 OLTP 資料庫，如 HBase（請參閱“[SSTables 和 LSM 樹](#)”）和 MPP 風格的分析型資料庫，如 Impala 【41】。HBase 與 Impala 都不使用 MapReduce，但都使用 HDFS 進行儲存。它們是迥異的資料訪問與處理方法，但是它們可以共存，並被整合到同一個系統中。

針對頻繁故障設計

當比較 MapReduce 和 MPP 資料庫時，兩種不同的設計思路出現了：處理故障和使用記憶體與磁碟的方式。與線上系統相比，批處理對故障不太敏感，因為就算失敗也不會立即影響到使用者，而且它們總是能再次執行。

如果一個節點在執行查詢時崩潰，大多數 MPP 資料庫會中止整個查詢，並讓使用者重新提交查詢或自動重新執行它【3】。由於查詢通常最多執行幾秒鐘或幾分鐘，所以這種錯誤處理的方法是可以接受的，因為重試的代價不是太大。MPP 資料庫還傾向於在記憶體中保留儘可能多的資料（例如，使用雜湊連線）以避免從磁碟讀取的開銷。

另一方面，MapReduce 可以容忍單個 Map 或 Reduce 任務的失敗，而不會影響作業的整體，透過以單個任務的粒度重試工作。它也會非常急切地將資料寫入磁碟，一方面是為了容錯，另一部分是因為假設資料集太大而不能適應記憶體。

MapReduce 方式更適用於較大的作業：要處理如此之多的資料並執行很長時間的作業，以至於在此過程中很可能至少遇到一個任務故障。在這種情況下，由於單個任務失敗而重新執行整個作業將是非常浪費的。即使以單個任務的粒度進行恢復引入了使得無故障處理更慢的開銷，但如果任務失敗率足夠高，這仍然是一種合理的權衡。

但是這些假設有多麼現實呢？在大多數叢集中，機器故障確實會發生，但是它們不是很頻繁——可能少到絕大多數作業都不會經歷機器故障。為了容錯，真的值得帶來這麼大的額外開銷嗎？

要了解 MapReduce 節約使用記憶體和在任務的層次進行恢復的原因，瞭解最初設計 MapReduce 的環境是很有幫助的。Google 有著混用的資料中心，線上生產服務和離線批處理作業在同樣機器上執行。每個任務都有一個透過容器強制執行的資源配給（CPU 核心、RAM、磁碟空間等）。每個任務也具有優先順序，如果優先順序較高的任務需要更多的資源，則可以終止（搶佔）同一臺機器上較低優先順序的任務以釋放資源。優先順序還決定了計算資源的定價：團隊必須為他們使用的資源付費，而優先順序更高的程序花費更多【59】。

這種架構允許非生產（低優先順序）計算資源被過量使用（overcommitted），因為系統知道必要時它可以回收資源。與分離生產和非生產任務的系統相比，過量使用資源可以更好地利用機器並提高效率。但由於 MapReduce 作業以低優先順序執行，它們隨時都有被搶佔的風險，因為優先順序較高的程序可能需要其資源。在高優先順序程序拿走所需資源後，批次作業能有效地“撿麵包屑”，利用剩下的任何計算資源。

在谷歌，執行一個小時的 MapReduce 任務有大約有 5% 的風險被終止，為了給更高優先順序的程序挪地方。這一機率比硬體問題、機器重啟或其他原因的機率高了一個數量級【59】。按照這種搶佔率，如果一個作業有 100 個任務，每個任務執行 10 分鐘，那麼至少有一個任務在完成之前被終止的風險大於 50%。

這就是 MapReduce 被設計為容忍頻繁意外任務終止的原因：不是因為硬體很不可靠，而是因為任意終止程序的自由有利於提高計算叢集中的資源利用率。

在開源的叢集排程器中，搶佔的使用較少。YARN 的 CapacityScheduler 支援搶佔，以平衡不同佇列的資源分配【58】，但在編寫本文時，YARN，Mesos 或 Kubernetes 不支援通用的優先順序搶佔【60】。在任務不經常被終止的環境中，MapReduce 的這一設計決策就沒有多少意義了。在下一節中，我們將研究一些與 MapReduce 設計決策相異的替代方案。

MapReduce之後

雖然 MapReduce 在 2000 年代後期變得非常流行，並受到大量的炒作，但它只是分散式系統的許多可能的程式設計模型之一。對於不同的資料量，資料結構和處理型別，其他工具可能更適合表示計算。

不管如何，我們在這一章花了大把時間來討論 MapReduce，因為它是一種有用的學習工具，它是分散式檔案系統的一種相當簡單明晰的抽象。在這裡，簡單 意味著我們能理解它在做什麼，而不是意味著使用它很簡單。恰恰相反：使用原始的 MapReduce API 來實現複雜的處理工作實際上是非常困難和費力的——例如，任意一種連線演算法都需要你從頭開始實現【37】。

針對直接使用 MapReduce 的困難，在 MapReduce 上有很多高階程式設計模型（Pig、Hive、Cascading、Crunch）被創造出來，作為建立在 MapReduce 之上的抽象。如果你瞭解 MapReduce 的原理，那麼它們學起來相當簡單。而且它們的高階結構能顯著簡化許多常見批處理任務的實現。

但是，MapReduce 執行模型本身也存在一些問題，這些問題並沒有透過增加另一個抽象層次而解決，而對於某些型別的處理，它表現得非常差勁。一方面，MapReduce 非常穩健：你可以使用它在任務會頻繁終止的多租戶系統上處理幾乎任意大量級的資料，並且仍然可以完成工作（雖然速度很慢）。另一方面，對於某些型別的處理而言，其他工具有時會快上幾個數量級。

在本章的其餘部分中，我們將介紹一些批處理方法。在 [第十一章](#) 我們將轉向流處理，它可以看作是加速批處理的另一種方法。

物化中間狀態

如前所述，每個 MapReduce 作業都獨立於其他任何作業。作業與世界其他地方的主要連線點是分散式檔案系統上的輸入和輸出目錄。如果希望一個作業的輸出成為第二個作業的輸入，則需要將第二個作業的輸入目錄配置為第一個作業輸出目錄，且外部工作流排程程式必須在第一個作業完成後再啟動第二個。

如果第一個作業的輸出是要在組織內廣泛釋出的資料集，則這種配置是合理的。在這種情況下，你需要透過名稱引用它，並將其重用為多個不同作業的輸入（包括由其他團隊開發的作業）。將資料釋出到分散式檔案系統中眾所周知的位置能夠帶來 **松耦合**，這樣作業就不需要知道是誰在提供輸入或誰在消費輸出（請參閱 “[邏輯與佈線相分離](#)”）。

但在很多情況下，你知道一個作業的輸出只能用作另一個作業的輸入，這些作業由同一個團隊維護。在這種情況下，分散式檔案系統上的檔案只是簡單的 **中間狀態**（intermediate state）：一種將資料從一個作業傳遞到下一個作業的方式。在一個用於構建推薦系統的，由 50 或 100 個 MapReduce 作業組成的複雜工作流中，存在著很多這樣的中間狀態【29】。

將這個中間狀態寫入檔案的過程稱為 **物化**（materialization）。（在“[聚合：資料立方體和物化檢視](#)”中已經在物化檢視的背景中遇到過這個術語。它意味著對某個操作的結果立即求值並寫出來，而不是在請求時按需計算）

作為對照，本章開頭的日誌分析示例使用 Unix 管道將一個命令的輸出與另一個命令的輸入連線起來。管道並沒有完全物化中間狀態，而是隻使用一個小的記憶體緩衝區，將輸出增量地 **流**（stream）向輸入。

與 Unix 管道相比，MapReduce 完全物化中間狀態的方法存在不足之處：

- MapReduce 作業只有在前驅作業（生成其輸入）中的所有任務都完成時才能啟動，而由 Unix 管道連線的程序會同時啟動，輸出一旦生成就會被消費。不同機器上的資料偏斜或負載不均意味著一個作業往往會有一些掉隊的任務，比其他任務要慢得多才能完成。必須等待至前驅作業的所有任務完成，拖慢了整個工作流程的執行。

- Mapper 通常是多餘的：它們僅僅是讀取剛剛由 Reducer 寫入的同樣檔案，為下一個階段的分割槽和排序做準備。在許多情況下，Mapper 程式碼可能是前驅 Reducer 的一部分：如果 Reducer 和 Mapper 的輸出有著相同的分割槽與排序方式，那麼 Reducer 就可以直接串在一起，而不用與 Mapper 相互交織。
- 將中間狀態儲存在分散式檔案系統中意味著這些檔案被複制到多個節點，對這些臨時資料這麼搞就比較過分了。

資料流引擎

為了解決 MapReduce 的這些問題，幾種用於分散式批處理的新執行引擎被開發出來，其中最著名的是 Spark 【61,62】，Tez 【63,64】和 Flink 【65,66】。它們的設計方式有很多區別，但有一個共同點：把整個工作流作為單個作業來處理，而不是把它分解為獨立的子作業。

由於它們將工作流顯式建模為資料從幾個處理階段穿過，所以這些系統被稱為 **資料流引擎 (dataflow engines)**。像 MapReduce 一樣，它們在一條線上透過反覆呼叫使用者定義的函式來一次處理一條記錄，它們透過輸入分割槽來並行化載荷，它們透過網路將一個函式的輸出複製到另一個函式的輸入。

與 MapReduce 不同，這些函式不需要嚴格扮演交織的 Map 與 Reduce 的角色，而是可以以更靈活的方式進行組合。我們稱這些函式為 **運算元 (operators)**，資料流引擎提供了幾種不同的選項來將一個運算元的輸出連線到另一個運算元的輸入：

- 一種選項是對記錄按鍵重新分割槽並排序，就像在 MapReduce 的混洗階段一樣（請參閱 “[分散式執行 MapReduce](#)”）。這種功能可以用於實現排序合併連線和分組，就像在 MapReduce 中一樣。
- 另一種可能是接受多個輸入，並以相同的方式進行分割槽，但跳過排序。當記錄的分割槽重要但順序無關緊要時，這省去了分割槽雜湊連線的工作，因為構建散列表還是會把順序隨機打亂。
- 對於廣播雜湊連線，可以將一個運算元的輸出，傳送到連線運算元的所有分割槽。

這種型別的處理引擎是基於像 Dryad 【67】和 Nephele 【68】這樣的研究系統，與 MapReduce 模型相比，它有幾個優點：

- 排序等昂貴的工作只需要在實際需要的地方執行，而不是預設地在每個 Map 和 Reduce 階段之間出現。
- 沒有不必要的 Map 任務，因為 Mapper 所做的工作通常可以合併到前面的 Reduce 運算元中（因為 Mapper 不會更改資料集的分割槽）。
- 由於工作流中的所有連線和資料依賴都是顯式宣告的，因此排程程式能夠總覽全域性，知道哪裡需要哪些資料，因而能夠利用區域性進行最佳化。例如，它可以嘗試將消費某些資料的任務放在與生成這些資料的任務相同的機器上，從而資料可以透過共享記憶體緩衝區傳輸，而不必透過網路複製。
- 通常，運算元間的中間狀態足以儲存在記憶體中或寫入本地磁碟，這比寫入 HDFS 需要更少的 I/O（必須將其複製到多臺機器，並將每個副本寫入磁碟）。MapReduce 已經對 Mapper 的輸出做了這種最佳化，但資料流引擎將這種思想推廣至所有的中間狀態。
- 運算元可以在輸入就緒後立即開始執行；後續階段無需等待前驅階段整個完成後再開始。
- 與 MapReduce（為每個任務啟動一個新的 JVM）相比，現有 Java 虛擬機器（JVM）程序可以重用來執行新運算元，從而減少啟動開銷。

你可以使用資料流引擎執行與 MapReduce 工作流同樣的計算，而且由於此處所述的最佳化，通常執行速度要明顯快得多。既然運算元是 Map 和 Reduce 的泛化，那麼相同的處理程式碼就可以在任一執行引擎上執行：Pig，Hive 或 Cascading 中實現的工作流可以無需修改程式碼，可以透過修改配置，簡單地從 MapReduce 切換到 Tez 或 Spark 【64】。

Tez 是一個相當薄的庫，它依賴於 YARN shuffle 服務來實現節點間資料的實際複製 【58】，而 Spark 和 Flink 則是包含了獨立網路通訊層，排程器，及使用者向 API 的大型框架。我們將簡要討論這些高階 API。

容錯

完全物化中間狀態至分散式檔案系統的一個優點是，它具有永續性，這使得 MapReduce 中的容錯相當容易：如果一個任務失敗，它可以在另一臺機器上重新啟動，並從檔案系統重新讀取相同的輸入。

Spark、Flink 和 Tez 避免將中間狀態寫入 HDFS，因此它們採取了不同的方法來容錯：如果一臺機器發生故障，並且該機器上的中間狀態丟失，則它會從其他仍然可用的資料重新計算（在可行的情況下是先前的中間狀態，要麼就只能是原始輸入資料，通常在 HDFS 上）。

為了實現這種重新計算，框架必須跟蹤一個給定的資料是如何計算的——使用了哪些輸入分割槽？應用了哪些運算元？Spark 使用 **彈性分散式資料集（RDD，Resilient Distributed Dataset）** 的抽象來跟蹤資料的譜系【61】，而 Flink 對運算元狀態存檔，允許恢復執行在執行過程中遇到錯誤的運算元【66】。

在重新計算資料時，重要的是要知道計算是否是 **確定性的**：也就是說，給定相同的輸入資料，運算元是否始終產生相同的輸出？如果一些丟失的資料已經發送給下游運算元，這個問題就很 important。如果運算元重新啟動，重新計算的資料與原有的丟失資料不一致，下游運算元很難解決新舊資料之間的矛盾。對於不確定性運算元來說，解決方案通常是殺死下游運算元，然後再重跑新資料。

為了避免這種級聯故障，最好讓運算元具有確定性。但需要注意的是，非確定性行為很容易悄悄溜進來：例如，許多程式語言在迭代雜湊表的元素時不能對順序作出保證，許多機率和統計算法顯式依賴於使用隨機數，以及用到系統時鐘或外部資料來源，這些都是都不確定性的行為。為了能可靠地從故障中恢復，需要消除這種不確定性因素，例如使用固定的種子生成偽隨機數。

透過重算資料來從故障中恢復並不總是正確的答案：如果中間狀態資料要比源資料小得多，或者如果計算量非常大，那麼將中間資料物化為檔案可能要比重新計算廉價的多。

關於物化的討論

回到 Unix 的類比，我們看到，MapReduce 就像是將每個命令的輸出寫入臨時檔案，而資料流引擎看起來更像是 Unix 管道。尤其是 Flink 是基於管道執行的思想而建立的：也就是說，將運算元的輸出增量地傳遞給其他運算元，不待輸入完成便開始處理。

排序運算元不可避免地需要消費全部的輸入後才能生成任何輸出，因為輸入中最後一條輸入記錄可能具有最小的鍵，因此需要作為第一條記錄輸出。因此，任何需要排序的運算元都需要至少暫時地累積狀態。但是工作流的許多其他部分可以以流水線方式執行。

當作業完成時，它的輸出需要持續到某個地方，以便使用者可以找到並使用它——很可能它會再次寫入分散式檔案系統。因此，在使用資料流引擎時，HDFS 上的物化資料集通常仍是作業的輸入和最終輸出。和 MapReduce 一樣，輸入是不可變的，輸出被完全替換。比起 MapReduce 的改進是，你不用再自己去將中間狀態寫入檔案系統了。

圖與迭代處理

在“[圖資料模型](#)”中，我們討論了使用圖來建模資料，並使用圖查詢語言來遍歷圖中的邊與點。[第二章](#) 的討論集中在 OLTP 風格的應用場景：快速執行查詢來查詢少量符合特定條件的頂點。

批處理上下文中的圖也很有趣，其目標是在整個圖上執行某種離線處理或分析。這種需求經常出現在機器學習應用（如推薦引擎）或排序系統中。例如，最著名的圖形分析演算法之一是 PageRank【69】，它試圖根據連結到某個網頁的其他網頁來估計該網頁的流行度。它作為配方的一部分，用於確定網路搜尋引擎呈現結果的順序。

像 Spark、Flink 和 Tez 這樣的資料流引擎（請參閱“[物化中間狀態](#)”）通常將運算元作為 **有向無環圖（DAG）** 的一部分安排在作業中。這與圖處理不一樣：在資料流引擎中，從一個運算元到另一個運算元的資料流 被構造成一個圖，而資料本身通常由關係型元組構成。在圖處理中，資料本身具有圖的形式。又一個不幸的命名混亂！

許多圖演算法是透過一次遍歷一條邊來表示的，將一個頂點與近鄰的頂點連線起來，以傳播一些資訊，並不斷重複，直到滿足一些條件為止——例如，直到沒有更多的邊要跟進，或直到一些指標收斂。我們在 [圖 2-6](#) 中看到一個例子，它透過重複跟進標明地點歸屬關係的邊，生成了資料庫中北美包含的所有地點列表（這種演算法被稱為 **傳遞閉包**，即 transitive closure）。

可以在分散式檔案系統中儲存圖（包含頂點和邊的列表的檔案），但是這種“重複至完成”的想法不能用普通的 MapReduce 來表示，因為它只掃過一趟資料。這種演算法因此經常以 **迭代** 的風格實現：

1. 外部排程程式執行批處理來計算演算法的一個步驟。
2. 當批處理過程完成時，排程器檢查它是否完成（基於完成條件——例如，沒有更多的邊要跟進，或者與上次迭代相比的變化低於某個閾值）。
3. 如果尚未完成，則排程程式返回到步驟 1 並執行另一輪批處理。

這種方法是有效的，但是用 MapReduce 實現它往往非常低效，因為 MapReduce 沒有考慮演算法的迭代性質：它總是讀取整個輸入資料集併產生一個全新的輸出資料集，即使與上次迭代相比，改變的僅僅是圖中的一小部分。

Pregel處理模型

針對圖批處理的最佳化——**批次同步並行 (BSP, Bulk Synchronous Parallel)** 計算模型【70】已經開始流行起來。其中，Apache Giraph 【37】，Spark 的 GraphX API 和 Flink 的 Gelly API 【71】實現了它。它也被稱為 **Pregel** 模型，因為 Google 的 Pregel 論文推廣了這種處理圖的方法【72】。

回想一下在 MapReduce 中，Mapper 在概念上向 Reducer 的特定呼叫“傳送訊息”，因為框架將所有具有相同鍵的 Mapper 輸出集中在一起。Pregel 背後有一個類似的想法：一個頂點可以向另一個頂點“傳送訊息”，通常這些訊息是沿著圖的邊傳送的。

在每次迭代中，為每個頂點呼叫一個函式，將所有傳送給它的訊息傳遞給它——就像呼叫 Reducer 一樣。與 MapReduce 的不同之處在於，在 Pregel 模型中，頂點在一次迭代到下一次迭代的過程中會記住它的狀態，所以這個函式只需要處理新的傳入訊息。如果圖的某個部分沒有被傳送訊息，那裡就不需要做任何工作。

這與 Actor 模型有些相似（請參閱“[分散式的 Actor 框架](#)”），除了頂點狀態和頂點之間的訊息具有容錯性和永續性，且通訊以固定的回合進行：在每次迭代中，框架遞送上次迭代中傳送的所有訊息。Actor 通常沒有這樣的時序保證。

容錯

頂點只能透過訊息傳遞進行通訊（而不是直接相互查詢）的事實有助於提高 Pregel 作業的效能，因為訊息可以成批處理，且等待通訊的次數也減少了。唯一的等待是在迭代之間：由於 Pregel 模型保證所有在一輪迭代中傳送的訊息都在下輪迭代中送達，所以在下一輪迭代開始前，先前的迭代必須完全完成，而所有的訊息必須在網路上完成複製。

即使底層網路可能丟失、重複或任意延遲訊息（請參閱“[不可靠的網路](#)”），Pregel 的實現能保證在後續迭代中訊息在其目標頂點恰好處理一次。像 MapReduce 一樣，框架能從故障中透明地恢復，以簡化在 Pregel 上實現演算法的程式設計模型。

這種容錯是透過在迭代結束時，定期存檔所有頂點的狀態來實現的，即將其全部狀態寫入持久化儲存。如果某個節點發生故障並且其記憶體中的狀態丟失，則最簡單的解決方法是將整個圖計算回滾到上一個存檔點，然後重啟計算。如果演算法是確定性的，且訊息記錄在日誌中，那麼也可以選擇性地只恢復丟失的分割槽（就像之前討論過的資料流引擎）【72】。

並行執行

頂點不需要知道它在哪臺物理機器上執行；當它向其他頂點發送訊息時，它只是簡單地將訊息發往某個頂點 ID。圖的分割槽取決於框架——即，確定哪個頂點執行在哪臺機器上，以及如何透過網路路由訊息，以便它們到達正確的地方。

由於程式設計模型一次僅處理一個頂點（有時稱為“像頂點一樣思考”），所以框架可以以任意方式對圖分割槽。理想情況下如果頂點需要進行大量的通訊，那麼它們最好能被分割槽到同一臺機器上。然而找到這樣一種最佳化的分割槽方法是很困難的——在實踐中，圖經常按照任意分配的頂點 ID 分割槽，而不會嘗試將相關的頂點分組在一起。

因此，圖演算法通常會有很多跨機器通訊的額外開銷，而中間狀態（節點之間傳送的訊息）往往比原始圖大。透過網路傳送訊息的開銷會顯著拖慢分散式圖演算法的速度。

出於這個原因，如果你的圖可以放入一臺計算機的記憶體中，那麼單機（甚至可能是單執行緒）演算法很可能會超越分散式批處理【73,74】。圖比記憶體大也沒關係，只要能放入單臺計算機的磁碟，使用 GraphChi 等框架進行單機處理是就一個可行的選擇【75】。如果圖太大，不適合單機處理，那麼像 Pregel 這樣的分散式方法是不可避免的。高效的

並行圖演算法是一個進行中的研究領域【76】。

高階API和語言

自 MapReduce 開始流行的這幾年以來，分散式批處理的執行引擎已經很成熟了。到目前為止，基礎設施已經足夠強大，能夠儲存和處理超過 10,000 臺機器叢集上的數 PB 的資料。由於在這種規模下物理執行批處理的問題已經被認為或多或少解決了，所以關注點已經轉向其他領域：改進程式設計模型，提高處理效率，擴大這些技術可以解決的問題集。

如前所述，Hive、Pig、Cascading 和 Crunch 等高階語言和 API 變得越來越流行，因為手寫 MapReduce 作業實在是個苦力活。隨著 Tez 的出現，這些高階語言還有一個額外好處，可以遷移到新的資料流執行引擎，而無需重寫作業程式碼。Spark 和 Flink 也有它們自己的高階資料流 API，通常是從 FlumeJava 中獲取的靈感【34】。

這些資料流 API 通常使用關係型構建塊來表達一個計算：按某個欄位連線資料集；按鍵對元組做分組；按某些條件過濾；並透過計數求和或其他函式來聚合元組。在內部，這些操作是使用本章前面討論過的各種連線和分組演算法來實現的。

除了少寫程式碼的明顯優勢之外，這些高階介面還支援互動式用法，在這種互動式使用中，你可以在 Shell 中增量式編寫分析程式碼，頻繁執行來觀察它做了什麼。這種開發風格在探索資料集和試驗處理方法時非常有用。這也讓人聯想到 Unix 哲學，我們在“[Unix 哲學](#)”中討論過這個問題。

此外，這些高階介面不僅提高了人類的工作效率，也提高了機器層面的作業執行效率。

向宣告式查詢語言的轉變

與硬寫執行連線的程式碼相比，指定連線關係運算元的優點是，框架可以分析連線輸入的屬性，並自動決定哪種上述連線演算法最適合當前任務。Hive、Spark 和 Flink 都有基於代價的查詢最佳化器可以做到這一點，甚至可以改變連線順序，最小化中間狀態的數量【66,77,78,79】。

連線演算法的選擇可以對批處理作業的效能產生巨大影響，而無需理解和記住本章中討論的各種連線演算法。如果連線是以 **宣告式（declarative）** 的方式指定的，那這就這是可行的：應用只是簡單地說明哪些連線是必需的，查詢最佳化器決定如何最好地執行連線。我們以前在“[資料查詢語言](#)”中見過這個想法。

但 MapReduce 及其資料流後繼者在其他方面，與 SQL 的完全宣告式查詢模型有很大區別。MapReduce 是圍繞著回撥函式的概念建立的：對於每條記錄或者一組記錄，呼叫一個使用者定義的函式（Mapper 或 Reducer），並且該函式可以自由地呼叫任意程式碼來決定輸出什麼。這種方法的優點是可以基於大量已有庫的生態系統創作：解析、自然語言分析、影象分析以及執行數值或統計算法等。

自由執行任意程式碼，長期以來都是傳統 MapReduce 批處理系統與 MPP 資料庫的區別所在（請參閱“[Hadoop 與分散式資料庫的對比](#)”一節）。雖然資料庫具有編寫使用者定義函式的功能，但是它們通常使用起來很麻煩，而且與大多數程式語言中廣泛使用的程式包管理器和依賴管理系統相容不佳（例如 Java 的 Maven、Javascript 的 npm 以及 Ruby 的 gems）。

然而資料流引擎已經發現，支援除連線之外的更多 **宣告式特性** 還有其他的優勢。例如，如果一個回撥函式只包含一個簡單的過濾條件，或者只是從一條記錄中選擇了一些欄位，那麼在為每條記錄呼叫函式時會有相當大的額外 CPU 開銷。如果以宣告方式表示這些簡單的過濾和對映操作，那麼查詢最佳化器可以利用列式儲存佈局（請參閱“[列式儲存](#)”），只從磁碟讀取所需的列。Hive、Spark DataFrames 和 Impala 還使用了向量化執行（請參閱“[記憶體頻寬和向量化處理](#)”）：在對 CPU 快取友好的內部迴圈中迭代資料，避免函式呼叫。Spark 生成 JVM 位元組碼【79】，Impala 使用 LLVM 為這些內部迴圈生成本機程式碼【41】。

透過在高階 API 中引入宣告式部分，並使查詢最佳化器可以在執行期間利用這些來做最佳化，批處理框架看起來越來越像 MPP 資料庫了（並且能實現可與之媲美的效能）。同時，透過擁有執行任意程式碼和以任意格式讀取資料的可擴充套件性，它們保持了靈活性的優勢。

專業化的不同領域

儘管能夠執行任意程式碼的可擴充套件性是很有用的，但是也有很多常見的例子，不斷重複著標準的處理模式。因而這些模式值得擁有自己的可重用通用構建模組實現。傳統上，MPP 資料庫滿足了商業智慧分析和業務報表的需求，但這只是許多使用批處理的領域之一。

另一個越來越重要的領域是統計和數值演算法，它們是機器學習應用所需要的（例如分類器和推薦系統）。可重用的實現正在出現：例如，Mahout 在 MapReduce、Spark 和 Flink 之上實現了用於機器學習的各種演算法，而 MADlib 在關係型 MPP 資料庫（Apache HAWQ）中實現了類似功能【54】。

空間演算法也是有用的，例如 **k** 近鄰搜尋（**k-nearest neighbors, kNN**）【80】，它在一些多維空間中搜索與給定項最近的專案——這是一種相似性搜尋。近似搜尋對於基因組分析演算法也很重要，它們需要找到相似但不相同的字串【81】。

批處理引擎正被用於分散式執行日益廣泛的各領域演算法。隨著批處理系統獲得各種內建功能以及高階宣告式運算元，且隨著 MPP 資料庫變得更加靈活和易於程式設計，兩者開始看起來相似了：最終，它們都只是儲存和處理資料的系統。

本章小結

在本章中，我們探索了批處理的主題。我們首先看到了諸如 awk、grep 和 sort 之類的 Unix 工具，然後我們看到了這些工具的設計理念是如何應用到 MapReduce 和更近的資料流引擎中的。一些設計原則包括：輸入是不可變的，輸出是為了作為另一個（仍未知的）程式的輸入，而複雜的問題是透過編寫“做好一件事”的小工具來解決的。

在 Unix 世界中，允許程式與程式組合的統一介面是檔案與管道；在 MapReduce 中，該介面是一個分散式檔案系統。我們看到資料流引擎添加了自己的管道式資料傳輸機制，以避免將中間狀態物化至分散式檔案系統，但作業的初始輸入和最終輸出通常仍是 HDFS。

分散式批處理框架需要解決的兩個主要問題是：

- 分割槽

在 MapReduce 中，Mapper 根據輸入檔案塊進行分割槽。Mapper 的輸出被重新分割槽、排序併合併到可配置數量的 Reducer 分割槽中。這一過程的目的是把所有的 **相關** 資料（例如帶有相同鍵的所有記錄）都放在同一個地方。

後 MapReduce 時代的資料流引擎若非必要會盡量避免排序，但它們也採取了大致類似的分割槽方法。

- 容錯

MapReduce 經常寫入磁碟，這使得從單個失敗的任務恢復很輕鬆，無需重新啟動整個作業，但在無故障的情況下減慢了執行速度。資料流引擎更多地將中間狀態儲存在記憶體中，更少地物化中間狀態，這意味著如果節點發生故障，則需要重算更多的資料。確定性運算元減少了需要重算的資料量。

我們討論了幾種 MapReduce 的連線演算法，其中大多數也在 MPP 資料庫和資料流引擎內部使用。它們也很好地演示了分割槽演算法是如何工作的：

- 排序合併連線

每個參與連線的輸入都透過一個提取連線鍵的 Mapper。透過分割槽、排序和合並，具有相同鍵的所有記錄最終都會進入相同的 Reducer 呼叫。這個函式能輸出連線好的記錄。

- 廣播雜湊連線

兩個連線輸入之一很小，所以它並沒有分割槽，而且能被完全載入進一個雜湊表中。因此，你可以為連線輸入大端的每個分割槽啟動一個 Mapper，將輸入小端的散列表載入到每個 Mapper 中，然後掃描大端，一次一條記錄，併為每條記錄查詢散列表。

- 分割槽雜湊連線

如果兩個連線輸入以相同的方式分割槽（使用相同的鍵，相同的雜湊函式和相同數量的分割槽），則可以獨立地對每個分割槽應用散列表方法。

分散式批處理引擎有一個刻意限制的程式設計模型：回撥函式（比如 Mapper 和 Reducer）被假定是無狀態的，而且除了指定的輸出外，必須沒有任何外部可見的副作用。這一限制允許框架在其抽象下隱藏一些困難的分散式系統問題：當遇到崩潰和網路問題時，任務可以安全地重試，任何失敗任務的輸出都被丟棄。如果某個分割槽的多個任務成功，則其中只有一個能使其輸出實際可見。

得益於這個框架，你在批處理作業中的程式碼無需操心實現容錯機制：框架可以保證作業的最終輸出與沒有發生錯誤的情況相同，雖然實際上也許不得不重試各種任務。比起線上服務一邊處理使用者請求一邊將寫入資料庫作為處理請求的副作用，批處理提供的這種可靠性語義要強得多。

批處理作業的顯著特點是，它讀取一些輸入資料併產生一些輸出資料，但不修改輸入——換句話說，輸出是從輸入衍生出的。最關鍵的是，輸入資料是 **有界的 (bounded)**：它有一個已知的，固定的大小（例如，它包含一些時間點的日誌檔案或資料庫內容的快照）。因為它是有界的，一個作業知道自己什麼時候完成了整個輸入的讀取，所以一個工作在做完後，最終總是會完成的。

在下一章中，我們將轉向流處理，其中的輸入是 **無界的 (unbounded)**——也就是說，你還有活兒要幹，然而它的輸入是永無止境的資料流。在這種情況下，作業永無完成之日。因為在任何時候都可能有更多的工作湧入。我們將看到，在某些方面上，流處理和批處理是相似的。但是關於無盡資料流的假設也對我們構建系統的方式產生了很多改變。

參考文獻

1. Jeffrey Dean and Sanjay Ghemawat: “MapReduce: Simplified Data Processing on Large Clusters,” at *6th USENIX Symposium on Operating System Design and Implementation* (OSDI), December 2004.
2. Joel Spolsky: “The Perils of JavaSchools,” joelonsoftware.com, December 25, 2005.
3. Shivnath Babu and Herodotos Herodotou: “Massively Parallel Databases and MapReduce Systems,” *Foundations and Trends in Databases*, volume 5, number 1, pages 1–104, November 2013.
[doi:10.1561/1900000036](https://doi.org/10.1561/1900000036)
4. David J. DeWitt and Michael Stonebraker: “MapReduce: A Major Step Backwards,” originally published at databasecolumn.vertica.com, January 17, 2008.
5. Henry Robinson: “The Elephant Was a Trojan Horse: On the Death of Map-Reduce at Google,” the-paper-trail.org, June 25, 2014.
6. “The Hollerith Machine,” United States Census Bureau, census.gov.
7. “IBM 82, 83, and 84 Sorters Reference Manual,” Edition A24-1034-1, International Business Machines Corporation, July 1962.
8. Adam Drake: “Command-Line Tools Can Be 235x Faster than Your Hadoop Cluster,” aadrake.com, January 25, 2014.
9. “GNU Coreutils 8.23 Documentation,” Free Software Foundation, Inc., 2014.
10. Martin Kleppmann: “Kafka, Samza, and the Unix Philosophy of Distributed Data,” martin.kleppmann.com, August 5, 2015.
11. Doug McIlroy: Internal Bell Labs memo, October 1964. Cited in: Dennis M. Richie: “Advice from Doug McIlroy,” cm.bell-labs.com.
12. M. D. McIlroy, E. N. Pinson, and B. A. Tague: “UNIX Time-Sharing System: Foreword,” *The Bell System Technical Journal*, volume 57, number 6, pages 1899–1904, July 1978.
13. Eric S. Raymond: *The Art of UNIX Programming*. Addison-Wesley, 2003. ISBN: 978-0-13-142901-7
14. Ronald Duncan: “Text File Formats – ASCII Delimited Text – Not CSV or TAB Delimited Text,” ronaldduncan.wordpress.com, October 31, 2009.
15. Alan Kay: “Is ‘Software Engineering’ an Oxymoron?,” tinlizzie.org.
16. Martin Fowler: “InversionOfControl,” martinfowler.com, June 26, 2005.
17. Daniel J. Bernstein: “Two File Descriptors for Sockets,” cr.yp.to.
18. Rob Pike and Dennis M. Ritchie: “The Styx Architecture for Distributed Systems,” *Bell Labs Technical Journal*,

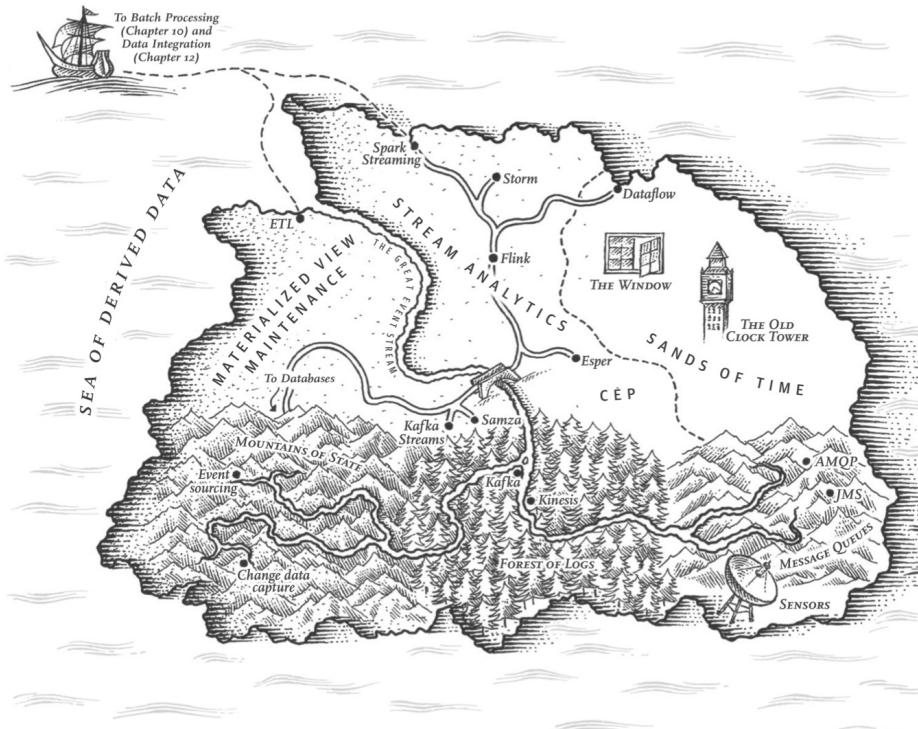
- volume 4, number 2, pages 146–152, April 1999.
19. Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: “[The Google File System](#),” at *19th ACM Symposium on Operating Systems Principles* (SOSP), October 2003. doi:[10.1145/945445.945450](https://doi.org/10.1145/945445.945450)
 20. Michael Ovsianikov, Silvius Rus, Damian Reeves, et al.: “[The Quantcast File System](#),” *Proceedings of the VLDB Endowment*, volume 6, number 11, pages 1092–1101, August 2013. doi:[10.14778/2536222.2536234](https://doi.org/10.14778/2536222.2536234)
 21. “[OpenStack Swift 2.6.1 Developer Documentation](#),” OpenStack Foundation, docs.openstack.org, March 2016.
 22. Zhe Zhang, Andrew Wang, Kai Zheng, et al.: “[Introduction to HDFS Erasure Coding in Apache Hadoop](#),” blog.cloudera.com, September 23, 2015.
 23. Peter Cnudde: “[Hadoop Turns 10](#),” yahoohadoop.tumblr.com, February 5, 2016.
 24. Eric Baldeschwieler: “[Thinking About the HDFS vs. Other Storage Technologies](#),” hortonworks.com, July 25, 2012.
 25. Brendan Gregg: “[Manta: Unix Meets Map Reduce](#),” dtrace.org, June 25, 2013.
 26. Tom White: *Hadoop: The Definitive Guide*, 4th edition. O'Reilly Media, 2015. ISBN: 978-1-491-90163-2
 27. Jim N. Gray: “[Distributed Computing Economics](#),” Microsoft Research Tech Report MSR-TR-2003-24, March 2003.
 28. Márton Trencséni: “[Luigi vs Airflow vs Pinball](#),” bytepawn.com, February 6, 2016.
 29. Roshan Sumbaly, Jay Kreps, and Sam Shah: “[The 'Big Data' Ecosystem at LinkedIn](#),” at *ACM International Conference on Management of Data (SIGMOD)*, July 2013. doi:[10.1145/2463676.2463707](https://doi.org/10.1145/2463676.2463707)
 30. Alan F. Gates, Olga Natkovich, Shubham Chopra, et al.: “[Building a High-Level Dataflow System on Top of MapReduce: The Pig Experience](#),” at *35th International Conference on Very Large Data Bases (VLDB)*, August 2009.
 31. Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, et al.: “[Hive – A Petabyte Scale Data Warehouse Using Hadoop](#),” at *26th IEEE International Conference on Data Engineering (ICDE)*, March 2010. doi:[10.1109/ICDE.2010.5447738](https://doi.org/10.1109/ICDE.2010.5447738)
 32. “[Cascading 3.0 User Guide](#),” Concurrent, Inc., docs.cascading.org, January 2016.
 33. “[Apache Crunch User Guide](#),” Apache Software Foundation, crunch.apache.org.
 34. Craig Chambers, Ashish Raniwala, Frances Perry, et al.: “[FlumeJava: Easy, Efficient Data-Parallel Pipelines](#),” at *31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2010. doi:[10.1145/1806596.1806638](https://doi.org/10.1145/1806596.1806638)
 35. Jay Kreps: “[Why Local State is a Fundamental Primitive in Stream Processing](#),” oreilly.com, July 31, 2014.
 36. Martin Kleppmann: “[Rethinking Caching in Web Apps](#),” martin.kleppmann.com, October 1, 2012.
 37. Mark Grover, Ted Malaska, Jonathan Seidman, and Gwen Shapira: *Hadoop Application Architectures*. O'Reilly Media, 2015. ISBN: 978-1-491-90004-8
 38. Philippe Ajoux, Nathan Bronson, Sanjeev Kumar, et al.: “[Challenges to Adopting Stronger Consistency at Scale](#),” at *15th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, May 2015.
 39. Sriranjan Manjunath: “[Skewed Join](#),” wiki.apache.org, 2009.
 40. David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri: “[Practical Skew Handling in Parallel Joins](#),” at *18th International Conference on Very Large Data Bases (VLDB)*, August 1992.
 41. Marcel Kornacker, Alexander Behm, Victor Bittorf, et al.: “[Impala: A Modern, Open-Source SQL Engine for Hadoop](#),” at *7th Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2015.
 42. Matthieu Monsch: “[Open-Sourcing PalDB, a Lightweight Companion for Storing Side Data](#),” engineering.linkedin.com, October 26, 2015.
 43. Daniel Peng and Frank Dabek: “[Large-Scale Incremental Processing Using Distributed Transactions and Notifications](#),” at *9th USENIX conference on Operating Systems Design and Implementation (OSDI)*, October 2010.
 44. “[Cloudera Search User Guide](#),” Cloudera, Inc., September 2015.
 45. Lili Wu, Sam Shah, Sean Choi, et al.: “[The Browsemaps: Collaborative Filtering at LinkedIn](#),” at *6th Workshop on Recommender Systems and the Social Web (RSWeb)*, October 2014.
 46. Roshan Sumbaly, Jay Kreps, Lei Gao, et al.: “[Serving Large-Scale Batch Computed Data with Project Voldemort](#),” at *10th USENIX Conference on File and Storage Technologies (FAST)*, February 2012.
 47. Varun Sharma: “[Open-Sourcing Terrapin: A Serving System for Batch Generated Data](#),” engineering.pinterest.com, September 14, 2015.

48. Nathan Marz: “[ElephantDB](#),” *slideshare.net*, May 30, 2011.
49. Jean-Daniel (JD) Cryans: “[How-to: Use HBase Bulk Loading, and Why](#),” *blog.cloudera.com*, September 27, 2013.
50. Nathan Marz: “[How to Beat the CAP Theorem](#),” *nathanmarz.com*, October 13, 2011.
51. Molly Bartlett Dishman and Martin Fowler: “[Agile Architecture](#),” at *O'Reilly Software Architecture Conference*, March 2015.
52. David J. DeWitt and Jim N. Gray: “[Parallel Database Systems: The Future of High Performance Database Systems](#),” *Communications of the ACM*, volume 35, number 6, pages 85–98, June 1992.
[doi:10.1145/129888.129894](https://doi.org/10.1145/129888.129894)
53. Jay Kreps: “[But the multi-tenancy thing is actually really really hard](#),” *tweetstorm, twitter.com*, October 31, 2014.
54. Jeffrey Cohen, Brian Dolan, Mark Dunlap, et al.: “[MAD Skills: New Analysis Practices for Big Data](#),” *Proceedings of the VLDB Endowment*, volume 2, number 2, pages 1481–1492, August 2009. [doi:10.14778/1687553.1687576](https://doi.org/10.14778/1687553.1687576)
55. Ignacio Terrizzano, Peter Schwarz, Mary Roth, and John E. Colino: “[Data Wrangling: The Challenging Journey from the Wild to the Lake](#),” at *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.
56. Paige Roberts: “[To Schema on Read or to Schema on Write, That Is the Hadoop Data Lake Question](#),” *adaptivesystemsinc.com*, July 2, 2015.
57. Bobby Johnson and Joseph Adler: “[The Sushi Principle: Raw Data Is Better](#),” at *Strata+Hadoop World*, February 2015.
58. Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, et al.: “[Apache Hadoop YARN: Yet Another Resource Negotiator](#),” at *4th ACM Symposium on Cloud Computing* (SoCC), October 2013. [doi:10.1145/2523616.2523633](https://doi.org/10.1145/2523616.2523633)
59. Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, et al.: “[Large-Scale Cluster Management at Google with Borg](#),” at *10th European Conference on Computer Systems* (EuroSys), April 2015. [doi:10.1145/2741948.2741964](https://doi.org/10.1145/2741948.2741964)
60. Malte Schwarzkopf: “[The Evolution of Cluster Scheduler Architectures](#),” *firmament.io*, March 9, 2016.
61. Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, et al.: “[Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing](#),” at *9th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), April 2012.
62. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia: *Learning Spark*. O'Reilly Media, 2015. ISBN: 978-1-449-35904-1
63. Bikas Saha and Hitesh Shah: “[Apache Tez: Accelerating Hadoop Query Processing](#),” at *Hadoop Summit*, June 2014.
64. Bikas Saha, Hitesh Shah, Siddharth Seth, et al.: “[Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications](#),” at *ACM International Conference on Management of Data* (SIGMOD), June 2015. [doi:10.1145/2723372.2742790](https://doi.org/10.1145/2723372.2742790)
65. Kostas Tzoumas: “[Apache Flink: API, Runtime, and Project Roadmap](#),” *slideshare.net*, January 14, 2015.
66. Alexander Alexandrov, Rico Bergmann, Stephan Ewen, et al.: “[The Stratosphere Platform for Big Data Analytics](#),” *The VLDB Journal*, volume 23, number 6, pages 939–964, May 2014. [doi:10.1007/s00778-014-0357-y](https://doi.org/10.1007/s00778-014-0357-y)
67. Michael Isard, Mihai Budiu, Yuan Yu, et al.: “[Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks](#),” at *European Conference on Computer Systems* (EuroSys), March 2007. [doi:10.1145/1272996.1273005](https://doi.org/10.1145/1272996.1273005)
68. Daniel Warneke and Odej Kao: “[Nephele: Efficient Parallel Data Processing in the Cloud](#),” at *2nd Workshop on Many-Task Computing on Grids and Supercomputers* (MTAGS), November 2009. [doi:10.1145/1646468.1646476](https://doi.org/10.1145/1646468.1646476)
69. Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd: “[The PageRank](#)”
70. Leslie G. Valiant: “[A Bridging Model for Parallel Computation](#),” *Communications of the ACM*, volume 33, number 8, pages 103–111, August 1990. [doi:10.1145/79173.79181](https://doi.org/10.1145/79173.79181)
71. Stephan Ewen, Kostas Tzoumas, Moritz Kauffmann, and Volker Markl: “[Spinning Fast Iterative Data Flows](#),” *Proceedings of the VLDB Endowment*, volume 5, number 11, pages 1268–1279, July 2012. [doi:10.14778/2350229.2350245](https://doi.org/10.14778/2350229.2350245)
72. Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, et al.: “[Pregel: A System for Large-Scale Graph Processing](#),” at *ACM International Conference on Management of Data* (SIGMOD), June 2010. [doi:10.1145/1807167.1807184](https://doi.org/10.1145/1807167.1807184)
73. Frank McSherry, Michael Isard, and Derek G. Murray: “[Scalability! But at What COST?](#),” at *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

74. Ionel Gog, Malte Schwarzkopf, Natacha Crooks, et al.: “[Musketeer: All for One, One for All in Data Processing Systems](#),” at *10th European Conference on Computer Systems* (EuroSys), April 2015. doi:[10.1145/2741948.2741968](https://doi.org/10.1145/2741948.2741968)
75. Aapo Kyrola, Guy Blelloch, and Carlos Guestrin: “[GraphChi: Large-Scale Graph Computation on Just a PC](#),” at *10th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), October 2012.
76. Andrew Lenharth, Donald Nguyen, and Keshav Pingali: “[Parallel Graph Analytics](#),” *Communications of the ACM*, volume 59, number 5, pages 78–87, May doi:[10.1145/2901919](https://doi.org/10.1145/2901919)
77. Fabian Hüske: “[Peeking into Apache Flink’s Engine Room](#),” flink.apache.org, March 13, 2015.
78. Mostafa Mokhtar: “[Hive 0.14 Cost Based Optimizer \(CBO\) Technical Overview](#),” hortonworks.com, March 2, 2015.
79. Michael Armbrust, Reynold S Xin, Cheng Lian, et al.: “[Spark SQL: Relational Data Processing in Spark](#),” at *ACM International Conference on Management of Data* (SIGMOD), June 2015. doi:[10.1145/2723372.2742797](https://doi.org/10.1145/2723372.2742797)
80. Daniel Blazevski: “[Planting Quadtrees for Apache Flink](#),” insightdataengineering.com, March 25, 2016.
81. Tom White: “[Genome Analysis Toolkit: Now Using Apache Spark for Data Processing](#),” blog.cloudera.com, April 6, 2016.

上一章	目錄	下一章
第三部分：衍生資料	設計資料密集型應用	第十一章：流處理

第十一章：流處理



有效的複雜系統總是從簡單的系統演化而來。反之亦然：從零設計的複雜系統沒一個能有效工作的。

—— 約翰·加爾，Systemantics (1975)

[TOC]

在 [第十章](#) 中，我們討論了批處理技術，它讀取一組檔案作為輸入，並生成一組新的檔案作為輸出。輸出是 **衍生資料 (derived data)** 的一種形式；也就是說，如果需要，可以透過再次執行批處理過程來重新建立資料集。我們看到了如何使用這個簡單而強大的想法來建立搜尋索引、推薦系統、做分析等等。

然而，在 [第十章](#) 中仍然有一個很大的假設：即輸入是有界的，即已知和有限的大小，所以批處理知道它何時完成輸入的讀取。例如，MapReduce 核心的排序操作必須讀取其全部輸入，然後才能開始生成輸出：可能發生這種情況：最後一條輸入記錄具有最小的鍵，因此需要第一個被輸出，所以提早開始輸出是不可行的。

實際上，很多資料是 **無界限** 的，因為它隨著時間的推移而逐漸到達：你的使用者在昨天和今天產生了資料，明天他們將繼續產生更多的資料。除非你停業，否則這個過程永遠都不會結束，所以資料集從來就不會以任何有意義的方式“完成”[【1】](#)。因此，批處理程式必須將資料人為地分成固定時間段的資料塊，例如，在每天結束時處理一天的資料，或者在每小時結束時處理一小時的資料。

日常批處理中的問題是，輸入的變更只會在一天之後的輸出中反映出來，這對於許多急躁的使用者來說太慢了。為了減少延遲，我們可以更頻繁地執行處理——比如說，在每秒鐘的末尾——或者甚至更連續一些，完全拋開固定的時間切片，當事件發生時就立即進行處理，這就是 **流處理 (stream processing)** 背後的想法。

一般來說，“流”是指隨著時間的推移逐漸可用的資料。這個概念出現在很多地方：Unix 的 `stdin` 和 `stdout`、程式語言（惰性列表）[【2】](#)、檔案系統 API（如 Java 的 `FileInputStream`）、TCP 連線、透過網際網路傳送音訊和影片等等。

在本章中，我們將把 **事件流 (event stream)** 視為一種資料管理機制：無界限，增量處理，與上一章中的批次資料相對應。我們將首先討論怎樣表示、儲存、透過網路傳輸流。在 “[資料庫與流](#)” 中，我們將研究流和資料庫之間的關係。最後在 “[流處理](#)” 中，我們將研究連續處理這些流的方法和工具，以及它們用於應用構建的方式。

傳遞事件流

在批處理領域，作業的輸入和輸出是檔案（也許在分散式檔案系統上）。流處理領域中的等價物看上去是什麼樣子的？

當輸入是一個檔案（一個位元組序列），第一個處理步驟通常是將其解析為一系列記錄。在流處理的上下文中，記錄通常被叫做 **事件 (event)**，但它本質上是一樣的：一個小的、自包含的、不可變的物件，包含某個時間點發生的某件事情的細節。一個事件通常包含一個來自日曆時鐘的時間戳，以指明事件發生的時間（請參閱 “[單調鍾與日曆時鐘](#)”）。

例如，發生的事件可能是使用者採取的行動，例如檢視頁面或進行購買。它也可能來源於機器，例如對溫度感測器或 CPU 利用率的週期性測量。在 “[使用 Unix 工具的批處理](#)” 的示例中，Web 伺服器日誌的每一行都是一個事件。

事件可能被編碼為文字字串或 JSON，或者某種二進位制編碼，如 [第四章](#) 所述。這種編碼允許你儲存一個事件，例如將其追加到一個檔案，將其插入關係表，或將其寫入文件資料庫。它還允許你透過網路將事件傳送到另一個節點以進行處理。

在批處理中，檔案被寫入一次，然後可能被多個作業讀取。類似地，在流處理術語中，一個事件由 **生產者 (producer)**（也稱為 **釋出者 (publisher)** 或 **傳送者 (sender)**）生成一次，然後可能由多個 **消費者 (consumer)**（**訂閱者 (subscribers)** 或 **接收者 (recipients)**）進行處理【3】。在檔案系統中，檔名標識一組相關記錄；在流式系統中，相關的事件通常被聚合為一個 **主題 (topic)** 或 **流 (stream)**。

原則上講，檔案或資料庫就足以連線生產者和消費者：生產者將其生成的每個事件寫入資料儲存，且每個消費者定期輪詢資料儲存，檢查自上次執行以來新出現的事件。這實際上正是批處理在每天結束時處理當天資料時所做的事情。

但當我們想要進行低延遲的連續處理時，如果資料儲存不是為這種用途專門設計的，那麼輪詢開銷就會很大。輪詢的越頻繁，能返回新事件的請求比例就越低，而額外開銷也就越高。相比之下，最好能在新事件出現時直接通知消費者。

資料庫在傳統上對這種通知機制支援的並不好，關係型資料庫通常有 **觸發器 (trigger)**，它們可以對變化（如，插入表中的一行）作出反應，但是它們的功能非常有限，並且在資料庫設計中有些後顧之憂【4,5】。相應的是，已經開發了專門的工具來提供事件通知。

訊息傳遞系統

向消費者通知新事件的常用方式是使用 **訊息傳遞系統 (messaging system)**：生產者傳送包含事件的訊息，然後將訊息推送给消費者。我們之前在 “[訊息傳遞中的資料流](#)” 中談到了這些系統，但現在我們將詳細介紹這些系統。

像生產者和消費者之間的 Unix 管道或 TCP 連線這樣的直接通道，是實現訊息傳遞系統的簡單方法。但是，大多數訊息傳遞系統都在這一基本模型上進行了擴充套件。特別的是，Unix 管道和 TCP 將恰好一個傳送者與恰好一個接收者連線，而一個訊息傳遞系統允許多個生產者節點將訊息傳送到同一個主題，並允許多個消費者節點接收主題中的訊息。

在這個 **釋出 / 訂閱** 模式中，不同的系統採取各種各樣的方法，並沒有針對所有目的的通用答案。為了區分這些系統，問一下這兩個問題會特別有幫助：

- 如果生產者傳送訊息的速度比消費者能夠處理的速度快會發生什麼？** 一般來說，有三種選擇：系統可以丟掉訊息，將訊息放入緩衝佇列，或使用 **背壓 (backpressure)**，也稱為 **流量控制**，即 **flow control**：阻塞生產者，以免其傳送更多的訊息）。例如 Unix 管道和 TCP 就使用了背壓：它們有一個固定大小的小緩衝區，如果填滿，傳送者會被阻塞，直到接收者從緩衝區中取出資料（請參閱 “[網路擁塞和排隊](#)”）。

如果訊息被快取在佇列中，那麼理解佇列增長會發生什麼是很重要的。當佇列裝不進記憶體時系統會崩潰嗎？還是將訊息寫入磁碟？如果是這樣，磁碟訪問又會如何影響訊息傳遞系統的效能【6】？

2. 如果節點崩潰或暫時離線，會發生什麼情況？——是否會有訊息丟失？與資料庫一樣，永續性可能需要寫入磁碟和 / 或複製的某種組合（請參閱“[複製與永續性](#)”），這是有代價的。如果你能接受有時訊息會丟失，則可能在同一硬體上獲得更高的吞吐量和更低的延遲。

是否可以接受訊息丟失取決於應用。例如，對於週期傳輸的感測器讀數和指標，偶爾丟失的資料點可能並不重要，因為更新的值會在短時間內發出。但要注意，如果大量的訊息被丟棄，可能無法立刻意識到指標已經不正確了【7】。如果你正在對事件計數，那麼它們能夠可靠送達是更重要的，因為每個丟失的訊息都意味著使計數器的錯誤擴大。

我們在 [第十章](#) 中探討的批處理系統的一個很好的特性是，它們提供了強大的可靠性保證：失敗的任務會自動重試，失敗任務的部分輸出會自動丟棄。這意味著輸出與沒有發生故障一樣，這有助於簡化程式設計模型。在本章的後面，我們將研究如何在流處理的上下文中提供類似的保證。

直接從生產者傳遞給消費者

許多訊息傳遞系統使用生產者和消費者之間的直接網路通訊，而不透過中間節點：

- UDP 組播廣泛應用於金融行業，例如股票市場，其中低時延非常重要【8】。雖然 UDP 本身是不可靠的，但應用層的協議可以恢復丟失的資料包（生產者必須記住它傳送的資料包，以便能按需重新發送資料包）。
- 無代理的訊息庫，如 ZeroMQ 【9】和 nanomsg 採取類似的方法，透過 TCP 或 IP 多播實現釋出 / 訂閱訊息傳遞。
- StatsD 【10】和 Brubeck 【7】使用不可靠的 UDP 訊息傳遞來收集網路中所有機器的指標並對其進行監控。（在 StatsD 協議中，只有接收到所有訊息，才認為計數器指標是正確的；使用 UDP 將使得指標處在一種最佳近似狀態【11】。另請參閱“[TCP 與 UDP](#)”
- 如果消費者在網路上公開了服務，生產者可以直接傳送 HTTP 或 RPC 請求（請參閱“[服務中的資料流：REST 與 RPC](#)”）將訊息推送给使用者。這就是 webhooks 背後的想法【12】，一種服務的回撥 URL 被註冊到另一個服務中，並且每當事件發生時都會向該 URL 發出請求。

儘管這些直接訊息傳遞系統在設計它們的環境中執行良好，但是它們通常要求應用程式碼意識到訊息丟失的可能性。它們的容錯程度極為有限：即使協議檢測到並重傳在網路中丟失的資料包，它們通常也只是假設生產者和消費者始終線上。

如果消費者處於離線狀態，則可能會丟失其不可達時傳送的訊息。一些協議允許生產者重試失敗的訊息傳遞，但當生產者崩潰時，它可能會丟失訊息緩衝區及其本應傳送的訊息，這種方法可能就沒用了。

訊息代理

一種廣泛使用的替代方法是透過 **訊息代理**（message broker，也稱為 **訊息佇列**，即 message queue）傳送訊息，訊息代理實質上是一種針對處理訊息流而最佳化的資料庫。它作為伺服器執行，生產者和消費者作為客戶端連線到伺服器。生產者將訊息寫入代理，消費者透過從代理那裡讀取來接收訊息。

透過將資料集中在代理上，這些系統可以更容易地容忍來來去去的客戶端（連線，斷開連線和崩潰），而永續性問題則轉移到代理的身上。一些訊息代理只將訊息儲存在記憶體中，而另一些訊息代理（取決於配置）將其寫入磁碟，以便在代理崩潰的情況下不會丟失。針對緩慢的消費者，它們通常會允許無上限的排隊（而不是丟棄訊息或背壓），儘管這種選擇也可能取決於配置。

排隊的結果是，消費者通常是 **非同步（asynchronous）** 的：當生產者傳送訊息時，通常只會等待代理確認訊息已經被快取，而不等待訊息被消費者處理。向消費者遞送訊息將發生在未來某個未定的時間點——通常在幾分之一秒之內，但有時當訊息堆積時會顯著延遲。

訊息代理與資料庫的對比

有些訊息代理甚至可以使用 XA 或 JTA 參與兩階段提交協議（請參閱“[實踐中的分散式事務](#)”）。這個功能與資料庫在本質上非常相似，儘管訊息代理和資料庫之間仍存在實踐上很重要的差異：

- 資料庫通常保留資料直至顯式刪除，而大多數訊息代理在訊息成功遞送給消費者時會自動刪除訊息。這樣的訊息代

理不適合長期的資料儲存。

- 由於它們很快就能刪除訊息，大多數訊息代理都認為它們的工作集相當小——即佇列很短。如果代理需要緩衝很多訊息，比如因為消費者速度較慢（如果記憶體裝不下訊息，可能會溢位到磁碟），每個訊息需要更長的處理時間，整體吞吐量可能會惡化【6】。
- 資料庫通常支援次級索引和各種搜尋資料的方式，而訊息代理通常支援按照某種模式匹配主題，訂閱其子集。雖然機制並不一樣，但對於客戶端選擇想要了解的資料的一部分，都是基本的方式。
- 查詢資料庫時，結果通常基於某個時間點的資料快照；如果另一個客戶端隨後向資料庫寫入一些改變了查詢結果的內容，則第一個客戶端不會發現其先前結果現已過期（除非它重複查詢或輪詢變更）。相比之下，訊息代理不支援任意查詢，但是當資料發生變化時（即新訊息可用時），它們會通知客戶端。

這是關於訊息代理的傳統觀點，它被封裝在諸如 JMS 【14】和 AMQP 【15】的標準中，並且被諸如 RabbitMQ、ActiveMQ、HornetQ、Qpid、TIBCO 企業訊息服務、IBM MQ、Azure Service Bus 和 Google Cloud Pub/Sub 所實現【16】。

多個消費者

當多個消費者從同一主題中讀取訊息時，有兩種主要的訊息傳遞模式，如 圖 11-1 所示：

- 負載均衡 (load balancing)

每條訊息都被傳遞給消費者之一，所以處理該主題下訊息的工作能被多個消費者共享。代理可以為消費者任意分配訊息。當處理訊息的代價高昂，希望能並行處理訊息時，此模式非常有用（在 AMQP 中，可以透過讓多個客戶端從同一個佇列中消費來實現負載均衡，而在 JMS 中則稱之為 共享訂閱，即 shared subscription）。

- 扇出 (fan-out)

每條訊息都被傳遞給 所有 消費者。扇出允許幾個獨立的消費者各自“收聽”相同的訊息廣播，而不會相互影響——這個流處理中的概念對應批處理中多個不同批處理作業讀取同一份輸入檔案（JMS 中的主題訂閱與 AMQP 中的交叉繫結提供了這一功能）。

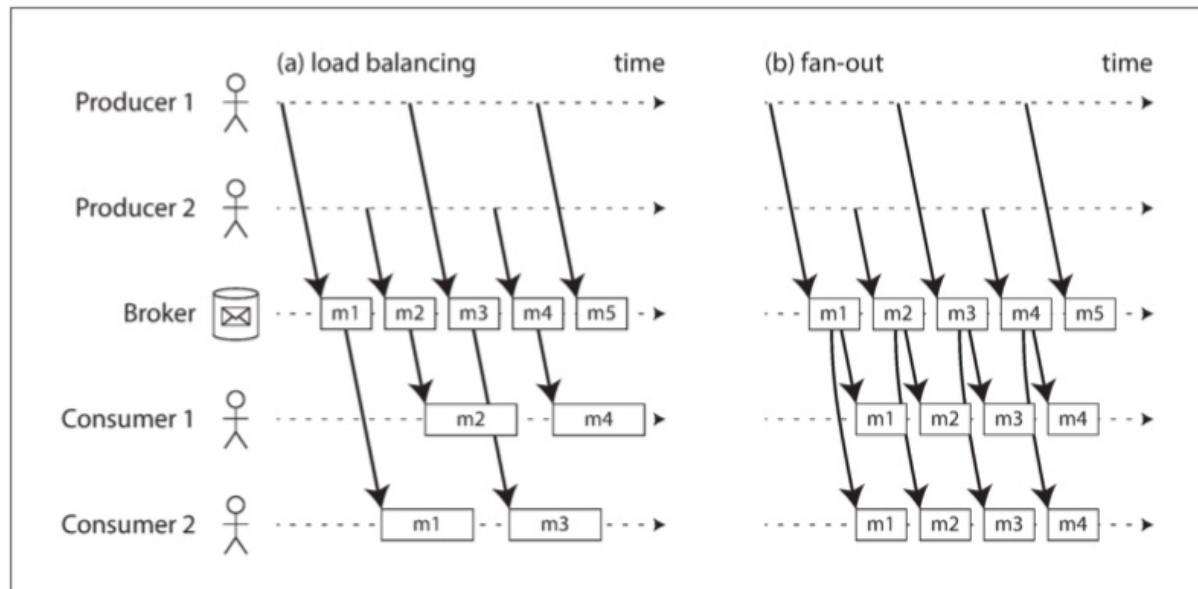


圖 11-1 (a) 負載平衡：在消費者間共享消費主題；(b) 扇出：將每條訊息傳遞給多個消費者。

兩種模式可以組合使用：例如，兩個獨立的消費者組可以每組各訂閱同一個主題，每一組都共同收到所有訊息，但在每一組內部，每條訊息僅由單個節點處理。

確認與重新傳遞

消費者隨時可能會崩潰，所以有一種可能的情況是：代理向消費者遞送訊息，但消費者沒有處理，或者在消費者崩潰之前只進行了部分處理。為了確保訊息不會丟失，訊息代理使用 **確認 (acknowledgments)**：客戶端必須顯式告知代理訊息處理完畢的時間，以便代理能將訊息從佇列中移除。

如果與客戶端的連線關閉，或者代理超出一段時間未收到確認，代理則認為訊息沒有被處理，因此它將訊息再遞送給另一個消費者。（請注意可能發生這樣的情況，訊息 實際上是 處理完畢的，但 確認 在網路中丟失了。需要一種原子提交協議才能處理這種情況，正如在“[實踐中的分散式事務](#)”中所討論的那樣）

當與負載均衡相結合時，這種重傳行為對訊息的順序有種有趣的影響。在 [圖 11-2](#) 中，消費者通常按照生產者傳送的順序處理訊息。然而消費者 2 在處理訊息 m3 時崩潰，與此同時消費者 1 正在處理訊息 m4。未確認的訊息 m3 隨後被重新發送給消費者 1，結果消費者 1 按照 m4, m3, m5 的順序處理訊息。因此 m3 和 m4 的交付順序與生產者 1 的傳送順序不同。

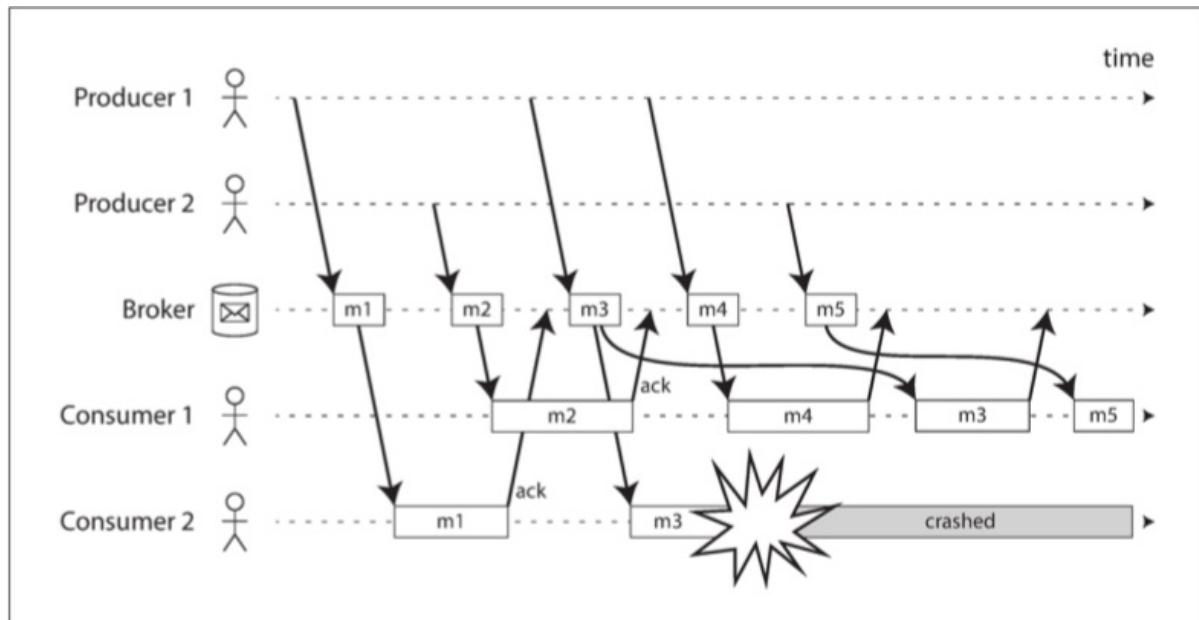


圖 11-2 在處理 m3 時消費者 2 崩潰，因此稍後重傳至消費者 1

即使訊息代理試圖保留訊息的順序（如 JMS 和 AMQP 標準所要求的），負載均衡與重傳的組合也不可避免地導致訊息被重新排序。為避免此問題，你可以讓每個消費者使用單獨的佇列（即不使用負載均衡功能）。如果訊息是完全獨立的，則訊息順序重排並不是一個問題。但正如我們將在本章後續部分所述，如果訊息之間存在因果依賴關係，這就是一個很重要的問題。

分割槽日誌

透過網路傳送資料包或向網路服務傳送請求通常是短暫的操作，不會留下永久的痕跡。儘管可以永久記錄（透過抓包與日誌），但我們通常不這麼做。即使是將訊息持久地寫入磁碟的訊息代理，在送達給消費者之後也會很快刪除訊息，因為它們建立在短暫訊息傳遞的思維方式上。

資料庫和檔案系統採用截然相反的方法論：至少在某人顯式刪除前，通常寫入資料庫或檔案的所有內容都要被永久記錄下來。

這種思維方式上的差異對建立衍生資料的方式有巨大影響。如 [第十章](#) 所述，批處理過程的一個關鍵特性是，你可以反覆執行它們，試驗處理步驟，不用擔心損壞輸入（因為輸入是隻讀的）。而 AMQP/JMS 風格的訊息傳遞並非如此：收到訊息是具有破壞性的，因為確認可能導致訊息從代理中被刪除，因此你不能期望再次運行同一個消費者能得到相同的結果。

如果你將新的消費者新增到訊息傳遞系統，通常只能接收到消費者註冊之後開始傳送的訊息。先前的任何訊息都隨風而逝，一去不復返。作為對比，你可以隨時為檔案和資料庫新增新的客戶端，且能讀取任意久遠的資料（只要應用沒有顯式覆蓋或刪除這些資料）。

為什麼我們不能把它倆雜交一下，既有資料庫的持久儲存方式，又有訊息傳遞的低延遲通知？這就是 基於日誌的訊息代理（log-based message brokers）背後的想法。

使用日誌進行訊息儲存

日誌只是磁碟上簡單的僅追加記錄序列。我們先前在 [第三章](#) 中日誌結構儲存引擎和預寫式日誌的上下文中討論了日誌，在 [第五章](#) 複製的上下文裡也討論了它。

同樣的結構可以用於實現訊息代理：生產者透過將訊息追加到日誌末尾來發送訊息，而消費者透過依次讀取日誌來接收訊息。如果消費者讀到日誌末尾，則會等待新訊息追加的通知。Unix 工具 `tail -f` 能監視檔案被追加寫入的資料，基本上就是這樣工作的。

為了伸縮超出單個磁碟所能提供的更高吞吐量，可以對日誌進行 **分割槽**（按 [第六章](#) 的定義）。不同的分割槽可以託管在不同的機器上，使得每個分割槽都有一份能獨立於其他分割槽進行讀寫的日誌。一個主題可以定義為一組攜帶相同型別訊息的分割槽。這種方法如 [圖 11-3](#) 所示。

在每個分割槽內，代理為每個訊息分配一個單調遞增的序列號或 **偏移量**（offset，在 [圖 11-3](#) 中，框中的數字是訊息偏移量）。這種序列號是有意義的，因為分割槽是僅追加寫入的，所以分割槽內的訊息是完全有序的。沒有跨不同分割槽的順序保證。

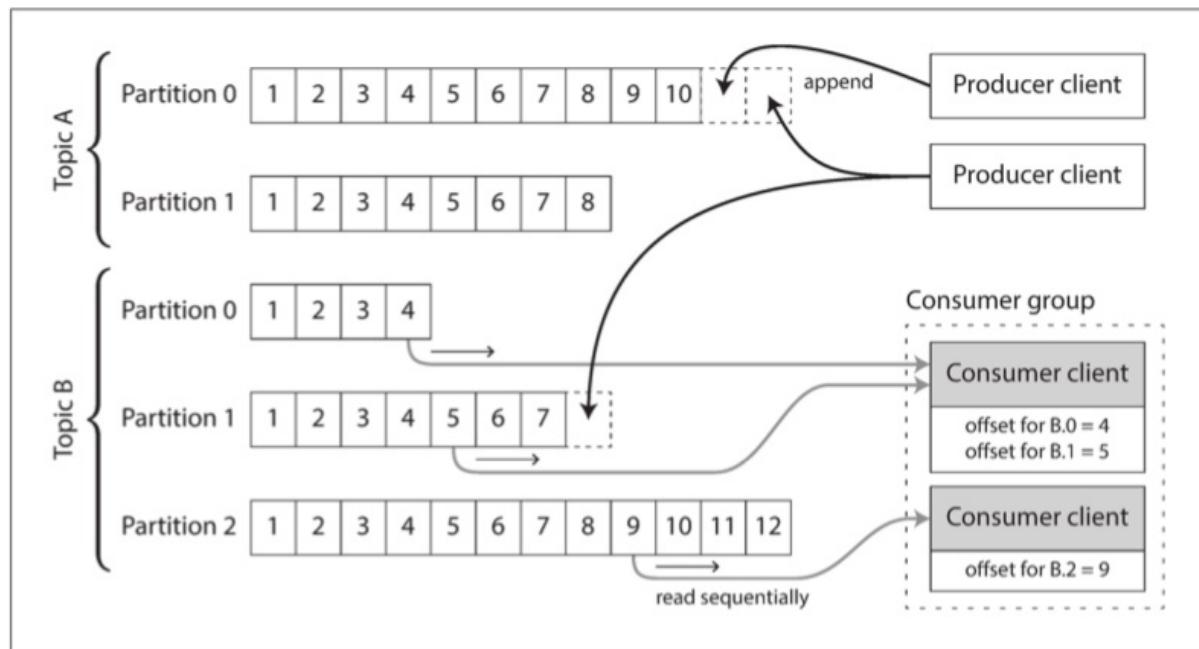


圖 11-3 生產者透過將訊息追加寫入主題分割槽檔案來發送訊息，消費者依次讀取這些檔案

Apache Kafka [【17,18】](#)、Amazon Kinesis Streams [【19】](#) 和 Twitter 的 DistributedLog [【20,21】](#) 都是基於日誌的訊息代理。Google Cloud Pub/Sub 在架構上類似，但對外暴露的是 JMS 風格的 API，而不是日誌抽象 [【16】](#)。儘管這些訊息代理將所有訊息寫入磁碟，但透過跨多臺機器分割槽，每秒能夠實現數百萬條訊息的吞吐量，並透過複製訊息來實現容錯性 [【22,23】](#)。

日誌與傳統的訊息傳遞相比

基於日誌的方法天然支援扇出式訊息傳遞，因為多個消費者可以獨立讀取日誌，而不會相互影響——讀取訊息不會將其從日誌中刪除。為了在一組消費者之間實現負載平衡，代理可以將整個分割槽分配給消費者組中的節點，而不是將單條訊息分配給消費者客戶端。

然後每個客戶端將消費被指派分割槽中的 **所有** 訊息。通常情況下，當一個使用者被指派了一個日誌分割槽時，它會以簡單的單執行緒方式順序地讀取分割槽中的訊息。這種粗粒度的負載均衡方法有一些缺點：

- 共享消費主題工作的節點數，最多為該主題中的日誌分割槽數，因為同一個分割槽內的所有訊息被遞送到同一個節點ⁱ。
- 如果某條訊息處理緩慢，則它會阻塞該分割槽中後續訊息的處理（一種行首阻塞的形式；請參閱“[描述效能](#)”）。

因此在訊息處理代價高昂，希望逐條並行處理，以及訊息的順序並沒有那麼重要的情況下，JMS/AMQP 風格的訊息代理是可取的。另一方面，在訊息吞吐量很高，處理迅速，順序很重要的情況下，基於日誌的方法表現得非常好。

ⁱ 要設計一種負載均衡方案也是有可能的，在這種方案中，兩個消費者透過讀取全部訊息來共享分割槽處理的工作，但是其中一個只考慮具有偶數偏移量的訊息，而另一個消費者只處理奇數編號的偏移量。或者你可以將訊息攤到一個執行緒池中來處理，但這種方法會使消費者偏移量管理變得複雜。一般來說，單執行緒處理單分割槽是合適的，可以透過增加更多分割槽來提高並行度。 ↵

消費者偏移量

順序消費一個分割槽使得判斷訊息是否已經被處理變得相當容易：所有偏移量小於消費者的當前偏移量的訊息已經被處理，而具有更大偏移量的訊息還沒有被看到。因此，代理不需要跟蹤確認每條訊息，只需要定期記錄消費者的偏移即可。這種方法減少了額外簿記開銷，而且在批處理和流處理中採用這種方法有助於提高基於日誌的系統的吞吐量。

實際上，這種偏移量與單領導者資料庫複製中常見的日誌序列號非常相似，我們在“[設定新從庫](#)”中討論了這種情況。在資料庫複製中，日誌序列號允許跟隨者斷開連線後，重新連線到領導者，並在不跳過任何寫入的情況下恢復複製。這裡原理完全相同：訊息代理表現得像一個主庫，而消費者就像一個從庫。

如果消費者節點失效，則失效消費者的分割槽將指派給其他節點，並從最後記錄的偏移量開始消費訊息。如果消費者已經處理了後續的訊息，但還沒有記錄它們的偏移量，那麼重啟後這些訊息將被處理兩次。我們將在本章後面討論這個問題的處理方法。

磁碟空間使用

如果只追加寫入日誌，則磁碟空間終究會耗盡。為了回收磁碟空間，日誌實際上被分割成段，並不時地將舊段刪除或移動到歸檔儲存。（我們將在後面討論一種更為複雜的磁碟空間釋放方式）

這就意味著如果一個慢消費者跟不上訊息產生的速率而落後得太多，它的消費偏移量指向了刪除的段，那麼它就會錯過一些訊息。實際上，日誌實現了一個有限大小的緩衝區，當緩衝區填滿時會丟棄舊訊息，它也被稱為 **迴圈緩衝區 (circular buffer)** 或 **環形緩衝區 (ring buffer)**。不過由於緩衝區在磁碟上，因此緩衝區可能相當的大。

讓我們做個簡單計算。在撰寫本文時，典型的大型硬碟容量為 6TB，順序寫入吞吐量為 150MB/s。如果以最快的速度寫訊息，則需要大約 11 個小時才能填滿磁碟。因而磁碟可以緩衝 11 個小時的訊息，之後它將開始覆蓋舊的訊息。即使使用多個磁碟和機器，這個比率也是一樣的。實踐中的部署很少能用滿磁碟的寫入頻寬，所以通常可以儲存一個幾天甚至幾周的日誌緩衝區。

不管保留多長時間的訊息，日誌的吞吐量或多或少保持不變，因為無論如何，每個訊息都會被寫入磁碟【18】。這種行為與預設將訊息儲存在記憶體中，僅當併列太長時才寫入磁碟的訊息傳遞系統形成鮮明對比。當併列很短時，這些系統非常快；而當這些系統開始寫入磁碟時，就要慢的多，所以吞吐量取決於保留的歷史數量。

當消費者跟不上生產者時

在“[訊息傳遞系統](#)”中，如果消費者無法跟上生產者傳送資訊的速度時，我們討論了三種選擇：丟棄資訊，進行緩衝或施加背壓。在這種分類法裡，基於日誌的方法是緩衝的一種形式，具有很大但大小固定的緩衝區（受可用磁碟空間的限制）。

如果消費者遠遠落後，而所要求的資訊比保留在磁碟上的資訊還要舊，那麼它將不能讀取這些資訊，所以代理實際上丟棄了比緩衝區容量更大的舊資訊。你可以監控消費者落後日誌頭部的距離，如果落後太多就發出報警。由於緩衝區很大，因而有足夠的時間讓運維人員來修復慢消費者，並在訊息開始丟失之前讓其趕上。

即使消費者真的落後太多開始丟失訊息，也只有那個消費者受到影響；它不會中斷其他消費者的服務。這是一個巨大的運維優勢：你可以實驗性地消費生產日誌，以進行開發，測試或除錯，而不必擔心會中斷生產服務。當消費者關閉或崩潰時，會停止消耗資源，唯一剩下的只有消費者偏移量。

這種行為也與傳統的訊息代理形成了鮮明對比，在那種情況下，你需要小心地刪除那些消費者已經關閉的佇列——否則那些佇列就會累積不必要的訊息，從其他仍活躍的消費者那裡佔走記憶體。

重播舊訊息

我們之前提到，使用 AMQP 和 JMS 風格的訊息代理，處理和確認訊息是一個破壞性的操作，因為它會導致訊息在代理上被刪除。另一方面，在基於日誌的訊息代理中，使用訊息更像是從檔案中讀取資料：這是隻讀操作，不會更改日誌。

除了消費者的任何輸出之外，處理的唯一副作用是消費者偏移量的前進。但偏移量是在消費者的控制之下的，所以如果需要的話可以很容易地操縱：例如你可以用昨天的偏移量跑一個消費者副本，並將輸出寫到不同的位置，以便重新處理最近一天的訊息。你可以使用各種不同的處理程式碼重複任意次。

這一方面使得基於日誌的訊息傳遞更像上一章的批處理，其中衍生資料透過可重複的轉換過程與輸入資料顯式分離。它允許進行更多的實驗，更容易從錯誤和漏洞中恢復，使其成為在組織內整合資料流的良好工具【24】。

資料庫與流

我們已經在訊息代理和資料庫之間進行了一些比較。儘管傳統上它們被視為單獨的工具類別，但是我們看到基於日誌的訊息代理已經成功地從資料庫中獲取靈感並將其應用於訊息傳遞。我們也可以反過來：從訊息傳遞和流中獲取靈感，並將它們應用於資料庫。

我們之前曾經說過，事件是某個時刻發生的事情的記錄。發生的事情可能是使用者操作（例如鍵入搜尋查詢）或讀取感測器，但也可能是寫入資料庫。某些東西被寫入資料庫的事實是可以被捕獲、儲存和處理的事件。這一觀察結果表明，資料庫和資料流之間的聯絡不僅僅是磁碟日誌的物理儲存——而是更深層的聯絡。

事實上，複製日誌（請參閱“[複製日誌的實現](#)”）是一個由資料庫寫入事件組成的流，由主庫在處理事務時生成。從庫將寫入流應用到它們自己的資料庫副本，從而最終得到相同資料的精確副本。複製日誌中的事件描述發生的資料更改。

我們還在“[全序廣播](#)”中遇到了狀態機複製原理，其中指出：如果每個事件代表對資料庫的寫入，並且每個副本按相同的順序處理相同的事件，則副本將達到相同的最終狀態（假設事件處理是一個確定性的操作）。這是事件流的又一種場景！

在本節中，我們將首先看看異構資料系統中出現的一個問題，然後探討如何透過將事件流的想法帶入資料庫來解決這個問題。

保持系統同步

正如我們在本書中所看到的，沒有一個系統能夠滿足所有的資料儲存、查詢和處理需求。在實踐中，大多數重要應用都需要組合使用幾種不同的技術來滿足所有的需求：例如，使用 OLTP 資料庫來為使用者請求提供服務，使用快取來加速常見請求，使用全文索引來處理搜尋查詢，使用資料倉庫用於分析。每一種技術都有自己的資料副本，並根據自己的目的進行儲存方式的最佳化。

由於相同或相關的資料出現在了不同的地方，因此相互間需要保持同步：如果某個專案在資料庫中被更新，它也應當在快取、搜尋索引和資料倉庫中被更新。對於資料倉庫，這種同步通常由 ETL 程序執行（請參閱“[資料倉庫](#)”），通常是先取得資料庫的完整副本，然後執行轉換，並批次載入到資料倉庫中——換句話說，批處理。我們在“[批處理工作流的輸出](#)”中同樣看到了如何使用批處理建立搜尋索引、推薦系統和其他衍生資料系統。

如果週期性的完整資料庫轉儲過於緩慢，有時會使用的替代方法是雙寫（dual write），其中應用程式碼在資料變更時明確寫入每個系統：例如，首先寫入資料庫，然後更新搜尋索引，然後使快取項失效（甚至同時執行這些寫入）。

但是，雙寫有一些嚴重的問題，其中一個是競爭條件，如 圖 11-4 所示。在這個例子中，兩個客戶端同時想要更新一個專案 X：客戶端 1 想要將值設定為 A，客戶端 2 想要將其設定為 B。兩個客戶端首先將新值寫入資料庫，然後將其寫入到搜尋索引。因為運氣不好，這些請求的時序是交錯的：資料庫首先看到來自客戶端 1 的寫入將值設定為 A，然後來自客戶端 2 的寫入將值設定為 B，因此資料庫中的最終值為 B。搜尋索引首先看到來自客戶端 2 的寫入，然後是客戶端 1 的寫入，所以搜尋索引中的最終值是 A。即使沒發生錯誤，這兩個系統現在也永久地不一致了。

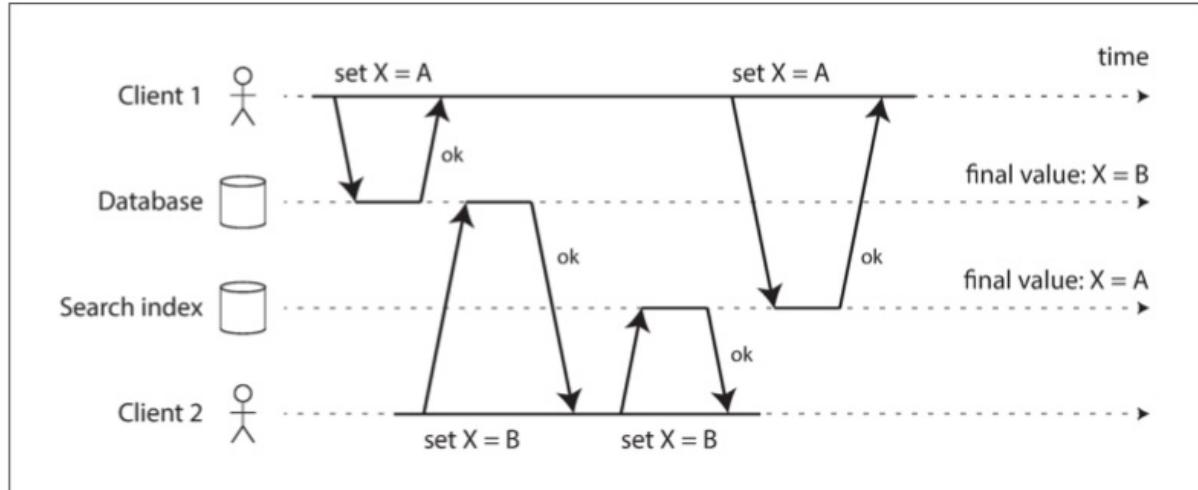


圖 11-4 在資料庫中 X 首先被設定為 A，然後被設定為 B，而在搜尋索引處，寫入以相反的順序到達

除非有一些額外的併發檢測機制，例如我們在“[檢測併發寫入](#)”中討論的版本向量，否則你甚至不會意識到發生了併發寫入——一個值將簡單地以無提示方式覆蓋另一個值。

雙重寫入的另一個問題是，其中一個寫入可能會失敗，而另一個成功。這是一個容錯問題，而不是一個併發問題，但也會造成兩個系統互相不一致的結果。確保它們要麼都成功要麼都失敗，是原子提交問題的一個例子，解決這個問題的代價是昂貴的（請參閱“[原子提交與兩階段提交](#)”）。

如果你只有一個單領導者複製的資料庫，那麼這個領導者決定了寫入順序，而狀態機複製方法可以在資料庫副本上工作。然而，在 圖 11-4 中，沒有單個主庫：資料庫可能有一個領導者，搜尋索引也可能有一個領導者，但是兩者都不追隨對方，所以可能會發生衝突（請參閱“[多主複製](#)”）。

如果實際上只有一個領導者——例如，資料庫——而且我們能讓搜尋索引成為資料庫的追隨者，情況要好得多。但這在實踐中可能嗎？

變更資料捕獲

大多數資料庫的複製日誌的問題在於，它們一直被當做資料庫的內部實現細節，而不是公開的 API。客戶端應該透過其資料模型和查詢語言來查詢資料庫，而不是解析複製日誌並嘗試從中提取資料。

數十年來，許多資料庫根本沒有記錄在檔的獲取變更日誌的方式。由於這個原因，捕獲資料庫中所有的變更，然後將其複製到其他儲存技術（搜尋索引、快取或資料倉庫）中是相當困難的。

最近，人們對 **變更資料捕獲 (change data capture, CDC)** 越來越感興趣，這是一種觀察寫入資料庫的所有資料變更，並將其提取並轉換為可以複製到其他系統中的形式的過程。CDC 是非常有意思的，尤其是當變更能被寫入後立刻用於流時。

例如，你可以捕獲資料庫中的變更，並不斷將相同的變更應用至搜尋索引。如果變更日誌以相同的順序應用，則可以預期搜尋索引中的資料與資料庫中的資料是匹配的。搜尋索引和任何其他衍生資料系統只是變更流的消費者，如 圖 11-5 所示。

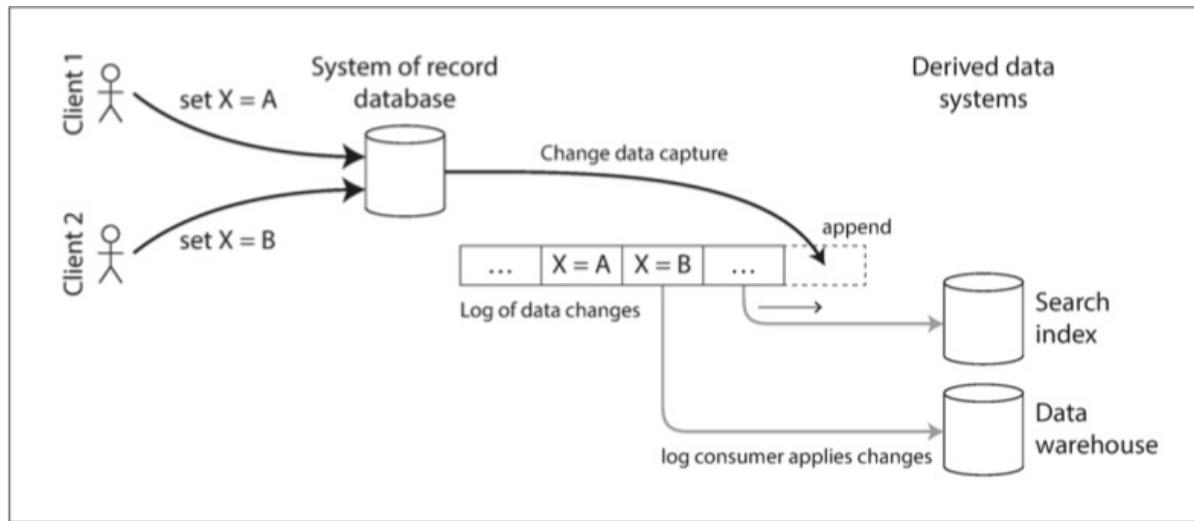


圖 11-5 將資料按順序寫入一個數據庫，然後按照相同的順序將這些更改應用到其他系統

變更資料捕獲的實現

我們可以將日誌消費者叫做 **衍生資料系統**，正如在 [第三部分](#) 的介紹中所討論的：儲存在搜尋索引和資料倉庫中的資料，只是 **記錄系統** 資料的額外檢視。變更資料捕獲是一種機制，可確保對記錄系統所做的所有更改都反映在衍生資料系統中，以便衍生系統具有資料的準確副本。

從本質上說，變更資料捕獲使得一個數據庫成為領導者（被捕獲變化的資料庫），並將其他元件變為追隨者。基於日誌的訊息代理非常適合從源資料庫傳輸變更事件，因為它保留了訊息的順序（避免了 [圖 11-2](#) 的重新排序問題）。

資料庫觸發器可用來實現變更資料捕獲（請參閱 “[基於觸發器的複製](#)”），透過註冊觀察所有變更的觸發器，並將相應的變更項寫入變更日誌表中。但是它們往往是脆弱的，而且有顯著的效能開銷。解析複製日誌可能是一種更穩健的方法，但它也很有挑戰，例如如何應對模式變更。

LinkedIn 的 Databus [\[25\]](#)，Facebook 的 Wormhole [\[26\]](#) 和 Yahoo! 的 Sherpa [\[27\]](#) 大規模地應用這個思路。Bottled Water 使用解碼 WAL 的 API 實現了 PostgreSQL 的 CDC [\[28\]](#)，Maxwell 和 Debezium 透過解析 binlog 對 MySQL 做了類似的事情 [\[29,30,31\]](#)，MongoRiver 讀取 MongoDB oplog [\[32,33\]](#)，而 GoldenGate 為 Oracle 提供類似的功能 [\[34,35\]](#)。

類似於訊息代理，變更資料捕獲通常是非同步的：記錄資料庫系統在提交變更之前不會等待消費者應用變更。這種設計具有的運維優勢是，新增緩慢的消費者不會過度影響記錄系統。不過，所有複製延遲可能有的問題在這裡都可能出現（請參閱 “[複製延遲問題](#)”）。

初始快照

如果你擁有 **所有** 對資料庫進行變更的日誌，則可以透過重播該日誌，來重建資料庫的完整狀態。但是在許多情況下，永遠保留所有更改會耗費太多磁碟空間，且重播過於費時，因此日誌需要被截斷。

例如，構建新的全文索引需要整個資料庫的完整副本——僅僅應用最近變更的日誌是不夠的，因為這樣會丟失最近未曾更新的專案。因此，如果你沒有完整的歷史日誌，則需要從一個一致的快照開始，如先前的 “[設定新從庫](#)” 中所述。

資料庫的快照必須與變更日誌中的已知位置或偏移量相對應，以便在處理完快照後知道從哪裡開始應用變更。一些 CDC 工具集成了這種快照功能，而其他工具則把它留給你手動執行。

日誌壓縮

如果你只能保留有限的歷史日誌，則每次要新增新的衍生資料系統時，都需要做一次快照。但 **日誌壓縮** (**log compaction**) 提供了一個很好的備選方案。

我們之前在“[雜湊索引](#)”中關於日誌結構儲存引擎的上下文中討論了日誌壓縮（請參閱 [圖 3-2](#) 的示例）。原理很簡單：儲存引擎定期在日誌中查詢具有相同鍵的記錄，丟掉所有重複的內容，並只保留每個鍵的最新更新。這個壓縮與合併過程在後臺執行。

在日誌結構儲存引擎中，具有特殊值 NULL（墓碑，即 tombstone）的更新表示該鍵被刪除，並會在日誌壓縮過程中被移除。但只要鍵不被覆蓋或刪除，它就會永遠留在日誌中。這種壓縮日誌所需的磁碟空間僅取決於資料庫的當前內容，而不取決於資料庫中曾經發生的寫入次數。如果相同的鍵經常被覆蓋寫入，則先前的值將最終將被垃圾回收，只有最新的值會保留下來。

在基於日誌的訊息代理與變更資料捕獲的上下文中也適用相同的想法。如果 CDC 系統被配置為，每個變更都包含一個主鍵，且每個鍵的更新都替換了該鍵以前的值，那麼只需要保留對鍵的最新寫入就足夠了。

現在，無論何時需要重建衍生資料系統（如搜尋索引），你可以從壓縮日誌主題的零偏移量處啟動新的消費者，然後依次掃描日誌中的所有訊息。日誌能保證包含資料庫中每個鍵的最新值（也可能是一些較舊的值）——換句話說，你可以使用它來獲取資料庫內容的完整副本，而無需從 CDC 源資料庫取一個快照。

Apache Kafka 支援這種日誌壓縮功能。正如我們將在本章後面看到的，它允許訊息代理被當成永續性儲存使用，而不僅僅是用於臨時訊息。

變更流的API支援

越來越多的資料庫開始將變更流作為第一等的介面，而不像傳統上要去做加裝改造，或者費工夫逆向工程一個 CDC。例如，RethinkDB 允許查詢訂閱通知，當查詢結果變更時獲得通知【36】，Firebase 【37】和 CouchDB 【38】基於變更流進行同步，該變更流同樣可用於應用。而 Meteor 使用 MongoDB oplog 訂閱資料變更，並改變了使用者介面【39】。

VoltDB 允許事務以流的形式連續地從資料庫中匯出資料【40】。資料庫將關係資料模型中的輸出流表示為一個表，事務可以向其中插入元組，但不能查詢。已提交事務按照提交順序寫入這個特殊表，而流則由該表中的元組日誌構成。外部消費者可以非同步消費該日誌，並使用它來更新衍生資料系統。

Kafka Connect 【41】致力於將廣泛的資料庫系統的變更資料捕獲工具與 Kafka 整合。一旦變更事件進入 Kafka 中，它就可以用於更新衍生資料系統，比如搜尋索引，也可以用於本章稍後討論的流處理系統。

事件溯源

我們在這裡討論的想法和 **事件溯源 (Event Sourcing)** 之間有一些相似之處，這是一個在 **領域驅動設計 (domain-driven design, DDD)** 社群中折騰出來的技術。我們將簡要討論事件溯源，因為它包含了一些關於流處理系統的有用想法。

與變更資料捕獲類似，事件溯源涉及到 將所有對應用狀態的變更 儲存為變更事件日誌。最大的區別是事件溯源將這一想法應用到了一個不同的抽象層次上：

- 在變更資料捕獲中，應用以 **可變方式 (mutable way)** 使用資料庫，可以任意更新和刪除記錄。變更日誌是從資料庫的底層提取的（例如，透過解析複製日誌），從而確保從資料庫中提取的寫入順序與實際寫入的順序相匹配，從而避免 [圖 11-4](#) 中的競態條件。寫入資料庫的應用不需要知道 CDC 的存在。
- 在事件溯源中，應用邏輯顯式構建在寫入事件日誌的不可變事件之上。在這種情況下，事件儲存是僅追加寫入的，更新與刪除是不鼓勵的或禁止的。事件被設計為旨在反映應用層面發生的事情，而不是底層的狀態變更。

事件溯源是一種強大的資料建模技術：從應用的角度來看，將使用者的行為記錄為不可變的事件更有意義，而不是在可變資料庫中記錄這些行為的影響。事件溯源使得應用隨時間演化更為容易，透過更容易理解事情發生的原因來幫助除錯的進行，並有利於防止應用 Bug（請參閱 “[不可變事件的優點](#)”）。

例如，儲存“學生取消選課”事件以中性的方式清楚地表達了單個行為的意圖，而其副作用“從登記表中刪除了一個條目，而一條取消原因的記錄被新增到學生反饋表”則嵌入了很多有關稍後對資料的使用方式的假設。如果引入一個新的應用功能，例如“將位置留給等待列表中的下一個人”——事件溯源方法允許將新的副作用輕鬆地從現有事件中脫開。

事件溯源類似於 **編年史 (chronicle)** 資料模型【45】，事件日誌與星型模式中的事實表之間也存在相似之處（請參閱“[星型和雪花型：分析的模式](#)”）。

諸如 Event Store 【46】這樣的專業資料庫已經被開發出來，供使用事件溯源的應用使用，但總的來說，這種方法獨立於任何特定的工具。傳統的資料庫或基於日誌的訊息代理也可以用來構建這種風格的應用。

從事件日誌中派生出當前狀態

事件日誌本身並不是很有用，因為使用者通常期望看到的是系統的當前狀態，而不是變更歷史。例如，在購物網站上，使用者期望能看到他們購物車裡的當前內容，而不是他們購物車所有變更的一個僅追加列表。

因此，使用事件溯源的應用需要拉取事件日誌（表示寫入系統的資料），並將其轉換為適合向用戶顯示的應用狀態（從系統讀取資料的方式【47】）。這種轉換可以使用任意邏輯，但它應當是確定性的，以便能再次執行，並從事件日誌中衍生出相同的應用狀態。

與變更資料捕獲一樣，重播事件日誌允許讓你重新構建系統的當前狀態。不過，日誌壓縮需要採用不同的方式處理：

- 用於記錄更新的 CDC 事件通常包含記錄的 **完整新版本**，因此主鍵的當前值完全由該主鍵的最近事件確定，而日誌壓縮可以丟棄相同主鍵的先前事件。
- 另一方面，事件溯源在更高層次進行建模：事件通常表示使用者操作的意圖，而不是因為操作而發生的狀態更新機制。在這種情況下，後面的事件通常不會覆蓋先前的事件，所以你需要完整的歷史事件來重新構建最終狀態。這裡進行同樣的日誌壓縮是不可能的。

使用事件溯源的應用通常有一些機制，用於儲存從事件日誌中匯出的當前狀態快照，因此它們不需要重複處理完整的日誌。然而這只是一種效能最佳化，用來加速讀取，提高從崩潰中恢復的速度；真正的目的是系統能夠永久儲存所有原始事件，並在需要時重新處理完整的事件日誌。我們將在“[不變性的侷限性](#)”中討論這個假設。

命令和事件

事件溯源的哲學是仔細區分 **事件 (event)** 和 **命令 (command)** 【48】。當來自使用者的請求剛到達時，它一開始是一個命令：在這個時間點上它仍然可能失敗，比如，因為違反了一些完整性條件。應用必須首先驗證它是否可以執行該命令。如果驗證成功並且命令被接受，則它變為一個持久化且不可變的事件。

例如，如果使用者試圖註冊特定使用者名稱，或預定飛機或劇院的座位，則應用需要檢查使用者名稱或座位是否已被佔用。（先前在“[容錯共識](#)”中討論過這個例子）當檢查成功時，應用可以生成一個事件，指示特定的使用者名稱是由特定的使用者 ID 註冊的，或者座位已經預留給特定的顧客。

在事件生成的時刻，它就成為了 **事實 (fact)**。即使客戶稍後決定更改或取消預訂，他們之前曾預定了某個特定座位的事實仍然成立，而更改或取消是之後新增的單獨的事件。

事件流的消費者不允許拒絕事件：當消費者看到事件時，它已經成為日誌中不可變的一部分，並且可能已經被其他消費者看到了。因此任何對命令的驗證，都需要在它成為事件之前同步完成。例如，透過使用一個可以原子性地自動驗證命令併發布事件的可序列事務。

或者，預訂座位的使用者請求可以拆分為兩個事件：第一個是暫時預約，第二個是驗證預約後的獨立的確認事件（如“[使用全序廣播實現線性一致的儲存](#)”中所述）。這種分割方式允許驗證發生在一個非同步的過程中。

狀態、流和不變性

我們在 [第十章](#) 中看到，批處理因其輸入檔案不變性而受益良多，你可以在現有輸入檔案上執行實驗性處理作業，而不用擔心損壞它們。這種不變性原則也是使得事件溯源與變更資料捕獲如此強大的原因。

我們通常將資料庫視為應用程式當前狀態的儲存——這種表示針對讀取進行了最佳化，而且通常對於服務查詢而言是最為方便的表示。狀態的本質是，它會變化，所以資料庫才會支援資料的增刪改。這又該如何匹配不變性呢？

只要你的狀態發生了變化，那麼這個狀態就是這段時間中事件修改的結果。例如，當前可用的座位列表是你已處理的預訂所產生的結果，當前帳戶餘額是帳戶中的借與貸的結果，而 Web 伺服器的響應時間圖，是所有已發生 Web 請求的獨立響應時間的聚合結果。

無論狀態如何變化，總是有一系列事件導致了這些變化。即使事情已經執行與回滾，這些事件出現是始終成立的。關鍵的想法是：可變的狀態與不可變事件的僅追加日誌相互之間並不矛盾：它們是一體兩面，互為陰陽的。所有變化的日誌——**變化日誌 (changelog)**，表示了隨時間演變的狀態。

如果你傾向於數學表示，那麼你可能會說，應用狀態是事件流對時間求積分得到的結果，而變更流是狀態對時間求微分的結果，如 圖 11-6 所示【49,50,51】。這個比喻有一些侷限性（例如，狀態的二階導似乎沒有意義），但這是考慮資料的一個實用出發點。

```
$$ state(now) = \int_{t=0}^{now} stream(t) \, dt \quad stream(t) = \frac{d state(t)}{dt}
```

\$\$

$$state(now) = \int_{t=0}^{now} stream(t) \, dt \quad stream(t) = \frac{d state(t)}{dt}$$

圖 11-6 應用當前狀態與事件流之間的關係

如果你持久儲存了變更日誌，那麼重現狀態就非常簡單。如果你認為事件日誌是你的記錄系統，而所有的衍生狀態都從它派生而來，那麼系統中的資料流動就容易理解的多。正如帕特·赫蘭（Pat Helland）所說的【52】：

事務日誌記錄了資料庫的所有變更。高速追加是更改日誌的唯一方法。從這個角度來看，資料庫的內容其實是日誌中記錄最新值的快取。日誌才是真相，資料庫是日誌子集的快取，這一快取子集恰好來自日誌中每條記錄與索引值的最新值。

日誌壓縮（如“[日誌壓縮](#)”中所述）是連線日誌與資料庫狀態之間的橋樑：它只保留每條記錄的最新版本，並丟棄被覆蓋的版本。

不可變事件的優點

資料庫中的不變性是一個古老的概念。例如，會計在幾個世紀以來一直在財務記賬中應用不變性。一筆交易發生時，它被記錄在一個僅追加寫入的分類帳中，實質上是描述貨幣、商品或服務轉手的事件日誌。賬目，比如利潤、虧損、資產負債表，是從分類帳中的交易求和衍生而來【53】。

如果發生錯誤，會計師不會刪除或更改分類帳中的錯誤交易——而是新增另一筆交易以補償錯誤，例如退還一筆不正確的費用。不正確的交易將永遠保留在分類帳中，對於審計而言可能非常重要。如果從不正確的分類賬衍生出的錯誤數字已經公佈，那麼下一個會計週期的數字就會包括一個更正。這個過程在會計事務中是很常見的【54】。

儘管這種可審計性只在金融系統中尤其重要，但對於不受這種嚴格監管的許多其他系統，也是很有幫助的。如“[批處理輸出的哲學](#)”中所討論的，如果你意外地部署了將錯誤資料寫入資料庫的錯誤程式碼，當代碼會破壞性地覆寫資料時，恢復要困難得多。使用不可變事件的僅追加日誌，診斷問題與故障恢復就要容易的多。

不可變的事件也包含了比當前狀態更多的資訊。例如在購物網站上，顧客可以將物品新增到他們的購物車，然後再將其移除。雖然從履行訂單的角度，第二個事件取消了第一個事件，但對分析目的而言，知道客戶考慮過某個特定項而之後又反悔，可能是很有用的。也許他們會選擇在未來購買，或者他們已經找到了替代品。這個資訊被記錄在事件日誌中，但對於移出購物車就刪除記錄的資料庫而言，這個資訊在移出購物車時可能就丟失了【42】。

從同一事件日誌中派生多個檢視

此外，透過從不變的事件日誌中分離出可變的狀態，你可以針對不同的讀取方式，從相同的事件日誌中衍生出幾種不同的表現形式。效果就像一個流的多個消費者一樣（圖 11-5）：例如，分析型資料庫 Druid 使用這種方式直接從 Kafka 攝取資料【55】，Pistachio 是一個分散式的鍵值儲存，使用 Kafka 作為提交日誌【56】，Kafka Connect 能將來自 Kafka 的資料匯出到各種不同的資料庫與索引【41】。這對於許多其他儲存和索引系統（如搜尋伺服器）來說是很有意義的，當系統要從分散式日誌中獲取輸入時亦然（請參閱“[保持系統同步](#)”）。

新增從事件日誌到資料庫的顯式轉換，能夠使應用更容易地隨時間演進：如果你想要引入一個新功能，以新的方式表示現有資料，則可以使用事件日誌來構建一個單獨的、針對新功能的讀取最佳化檢視，無需修改現有系統而與之共存。並行執行新舊系統通常比在現有系統中執行複雜的模式遷移更容易。一旦不再需要舊的系統，你可以簡單地關閉它並收回其資源【47,57】。

如果你不需要擔心如何查詢與訪問資料，那麼儲存資料通常是非常簡單的。模式設計、索引和儲存引擎的許多複雜性，都是希望支援某些特定查詢和訪問模式的結果（請參閱 [第三章](#)）。出於這個原因，透過將資料寫入的形式與讀取形式相分離，並允許幾個不同的讀取檢視，你能獲得很大的靈活性。這個想法有時被稱為 **命令查詢責任分離 (command query responsibility segregation, CQRS)** 【42,58,59】。

資料庫和模式設計的傳統方法是基於這樣一種謬論，資料必須以與查詢相同的形式寫入。如果可以將資料從針對寫入最佳化的事件日誌轉換為針對讀取最佳化的應用狀態，那麼有關規範化和非規範化的爭論就變得無關緊要了（請參閱“[多對一和多對多的關係](#)”）：在針對讀取最佳化的檢視中對資料進行非規範化是完全合理的，因為翻譯過程提供了使其與事件日誌保持一致的機制。

在“[描述負載](#)”中，我們討論了推特主頁時間線，它是特定使用者關注的人群所發推特的快取（類似郵箱）。這是 **針對讀取最佳化的狀態** 的又一個例子：主頁時間線是高度非規範化的，因為你的推文與你所有粉絲的時間線都構成了重複。然而，扇出服務保持了這種重複狀態與新推特以及新關注關係的同步，從而保證了重複的可管理性。

併發控制

事件溯源和變更資料捕獲的最大缺點是，事件日誌的消費者通常是非同步的，所以可能會出現這樣的情況：使用者會寫入日誌，然後從日誌衍生檢視中讀取，結果發現他的寫入還沒有反映在讀取檢視中。我們之前在“[讀已之寫](#)”中討論了這個問題以及可能的解決方案。

一種解決方案是將事件追加到日誌時同步執行讀取檢視的更新。而將這些寫入操作合併為一個原子單元需要 **事務**，所以要麼將事件日誌和讀取檢視儲存在同一個儲存系統中，要麼就需要跨不同系統進行分散式事務。或者，你也可以使用在“[使用全序廣播實現線性一致的儲存](#)”中討論的方法。

另一方面，從事件日誌匯出當前狀態也簡化了併發控制的某些部分。許多對於多物件事務的需求（請參閱“[單物件和多物件操作](#)”）源於單個使用者操作需要在多個不同的位置更改資料。透過事件溯源，你可以設計一個自包含的事件以表示一個使用者操作。然後使用者操作就只需要在一個地方進行單次寫入操作——即將事件附加到日誌中——這個還是很容易使原子化的。

如果事件日誌與應用狀態以相同的方式分割槽（例如，處理分割槽 3 中的客戶事件只需要更新分割槽 3 中的應用狀態），那麼直接使用單執行緒日誌消費者就不需要寫入併發控制了。它從設計上一次只處理一個事件（請參閱“[真的序列執行](#)”）。日誌透過在分割槽中定義事件的序列順序，消除了併發性的不確定性【24】。如果一個事件觸及多個狀態分割槽，那麼需要做更多的工作，我們將在 [第十二章](#) 討論。

不變性的侷限性

許多不使用事件溯源模型的系統也還是依賴不可變性：各種資料庫在內部使用不可變的資料結構或多版本資料來支援時間點快照（請參閱“[索引和快照隔離](#)”）。Git、Mercurial 和 Fossil 等版本控制系統也依靠不可變的資料來儲存檔案的版本歷史記錄。

永遠保持所有變更的不變歷史，在多大程度上是可行的？答案取決於資料集的流失率。一些工作負載主要是新增資料，很少更新或刪除；它們很容易保持不變。其他工作負載在相對較小的資料集上有較高的更新 / 刪除率；在這些情況下，不可變的歷史可能增至難以接受的巨大，碎片化可能成為一個問題，壓縮與垃圾收集的表現對於運維的穩健性變得至關重要【60,61】。

除了效能方面的原因外，也可能有出於管理方面的原因需要刪除資料的情況，儘管這些資料都是不可變的。例如，隱私條例可能要求在使用者關閉帳戶後刪除他們的個人資訊，資料保護立法可能要求刪除錯誤的資訊，或者可能需要阻止敏感資訊的意外洩露。

在這種情況下，僅僅在日誌中新增另一個事件來指明先前的資料應該被視為刪除是不夠的——你實際上是想改寫歷史，並假裝資料從一開始就沒有寫入。例如，Datomic 管這個特性叫 **切除**（excision）【62】，而 Fossil 版本控制系統有一個類似的概念叫 **避免**（shunning）【63】。

真正刪除資料是非常非常困難的【64】，因為副本可能存在於很多地方：例如，儲存引擎，檔案系統和 SSD 通常會向一個新位置寫入，而不是原地覆蓋舊資料【52】，而備份通常是特意做成不可變的，防止意外刪除或損壞。刪除操作更多的是指“使取回資料更困難”，而不是指“使取回資料不可能”。無論如何，有時你必須得嘗試，正如我們在“[立法與自律](#)”中所看到的。

流處理

到目前為止，本章中我們已經討論了流的來源（使用者活動事件，感測器和寫入資料庫），我們討論了流如何傳輸（直接透過訊息傳送，透過訊息代理，透過事件日誌）。

剩下的就是討論一下你可以用流做什麼——也就是說，你可以處理它。一般來說，有三種選項：

1. 你可以將事件中的資料寫入資料庫、快取、搜尋索引或類似的儲存系統，然後能被其他客戶端查詢。如 [圖 11-5](#) 所示，這是資料庫與系統其他部分所發生的變更保持同步的好方法——特別是當流消費者是寫入資料庫的唯一客戶端時。如“[批處理工作流的輸出](#)”中所討論的，它是寫入儲存系統的流等價物。
2. 你能以某種方式將事件推送给使用者，例如傳送報警郵件或推送通知，或將事件流式傳輸到可實時顯示的儀表板上。在這種情況下，人是流的最終消費者。
3. 你可以處理一個或多個輸入流，併產生一個或多個輸出流。流可能會經過由幾個這樣的處理階段組成的流水線，最後再輸出（選項 1 或 2）。

在本章的剩餘部分中，我們將討論選項 3：處理流以產生其他衍生流。處理這樣的流的程式碼片段，被稱為 **運算元**（operator）或 **作業**（job）。它與我們在 [第十章](#) 中討論過的 Unix 程序和 MapReduce 作業密切相關，資料流的模式是相似的：一個流處理器以只讀的方式使用輸入流，並將其輸出以僅追加的方式寫入一個不同的位置。

流處理中的分割槽和並行化模式也非常類似於 [第十章](#) 中介紹的 MapReduce 和資料流引擎，因此我們不再重複這些主題。基本的 Map 操作（如轉換和過濾記錄）也是一樣的。

與批次作業相比的一個關鍵區別是，流不會結束。這種差異會帶來很多隱含的結果。正如本章開始部分所討論的，排序對無界資料集沒有意義，因此無法使用 **排序合併連線**（請參閱 [Reduce 側連線與分組](#)）。容錯機制也必須改變：對於已經運行了幾分鐘的批處理作業，可以簡單地從頭開始重啟失敗任務，但是對於已經執行數年的流作業，重啟後從頭開始跑可能並不是一個可行的選項。

流處理的應用

長期以來，流處理一直用於監控目的，如果某個事件發生，組織希望能得到警報。例如：

- 欺詐檢測系統需要確定信用卡的使用模式是否有意外地變化，如果信用卡可能已被盜刷，則鎖卡。
- 交易系統需要檢查金融市場的價格變化，並根據指定的規則進行交易。
- 製造系統需要監控工廠中機器的狀態，如果出現故障，可以快速定位問題。
- 軍事和情報系統需要跟蹤潛在侵略者的活動，並在出現襲擊徵兆時發出警報。

這些型別的應用需要非常精密複雜的模式匹配與相關檢測。然而隨著時代的進步，流處理的其他用途也開始出現。在本節中，我們將簡要比較一下這些應用。

複合事件處理

複合事件處理（complex event processing, CEP）是 20 世紀 90 年代為分析事件流而開發出的一種方法，尤其適用於需要搜尋某些事件模式的應用【65,66】。與正則表示式允許你在字串中搜索特定字元模式的方式類似，CEP 允許你指定規則以在流中搜索某些事件模式。

CEP 系統通常使用高層次的宣告式查詢語言，比如 SQL，或者圖形使用者介面，來描述應該檢測到的事件模式。這些查詢被提交給處理引擎，該引擎消費輸入流，並在內部維護一個執行所需匹配的狀態機。當發現匹配時，引擎發出一個複合事件（即 complex event，CEP 因此得名），並附有檢測到的事件模式詳情【67】。

在這些系統中，查詢和資料之間的關係與普通資料庫相比是顛倒的。通常情況下，資料庫會持久儲存資料，並將查詢視為臨時的：當查詢進入時，資料庫搜尋與查詢匹配的資料，然後在查詢完成時丟掉查詢。CEP 引擎反轉了角色：查詢是長期儲存的，來自輸入流的事件不斷流過它們，搜尋匹配事件模式的查詢【68】。

CEP 的實現包括 Esper【69】、IBM InfoSphere Streams【70】、Apama、TIBCO StreamBase 和 SQLstream。像 Samza 這樣的分散式流處理元件，支援使用 SQL 在流上進行宣告式查詢【71】。

流分析

使用流處理的另一個領域是對流進行分析。CEP 與流分析之間的邊界是模糊的，但一般來說，分析往往對找出特定事件序列並不關心，而更關注大量事件上的聚合與統計指標——例如：

- 測量某種型別事件的速率（每個時間間隔內發生的頻率）
- 滾動計算一段時間視窗內某個值的平均值
- 將當前的統計值與先前的時間區間的值對比（例如，檢測趨勢，當指標與上週同比異常偏高或偏低時報警）

這些統計值通常是在固定時間區間內進行計算的，例如，你可能想知道在過去 5 分鐘內服務每秒查詢次數的均值，以及此時間段內響應時間的第 99 百分位點。在幾分鐘內取平均，能抹平秒和秒之間的無關波動，且仍然能向你展示流量模式的時間圖景。聚合的時間間隔稱為 視窗（window），我們將在“[時間推理](#)”中更詳細地討論視窗。

流分析系統有時會使用機率演算法，例如 Bloom filter（我們在“[效能最佳化](#)”中遇到過）來管理成員資格，HyperLogLog【72】用於基數估計以及各種百分比估計算法（請參閱“[實踐中的百分位點](#)”）。機率演算法產出近似的結果，但比起精確演算法的優點是記憶體使用要少得多。使用近似演算法有時讓人們覺得流處理系統總是有損的和不精確的，但這是錯誤看法：流處理並沒有任何內在的近似性，而機率演算法只是一種最佳化【73】。

許多開源分散式流處理框架的設計都是針對分析設計的：例如 Apache Storm、Spark Streaming、Flink、Concord、Samza 和 Kafka Streams【74】。託管服務包括 Google Cloud Dataflow 和 Azure Stream Analytics。

維護物化檢視

我們在“[資料庫與流](#)”中看到，資料庫的變更流可以用於維護衍生資料系統（如快取、搜尋索引和資料倉庫），並使其與源資料庫保持最新。我們可以將這些示例視作維護 物化檢視（materialized view）的一種具體場景（請參閱“[聚合：資料立方體和物化檢視](#)”）：在某個資料集上衍生出一個替代檢視以便高效查詢，並在底層資料變更時更新檢視【50】。

同樣，在事件溯源中，應用程式的狀態是透過應用事件日誌來維護的；這裡的應用程式狀態也是一種物化檢視。與流分析場景不同的是，僅考慮某個時間視窗內的事件通常是不夠的：構建物化檢視可能需要任意時間段內的 **所有** 事件，除了那些可能由日誌壓縮丟棄的過時事件（請參閱“[日誌壓縮](#)”）。實際上，你需要一個可以一直延伸到時間開端的視窗。

原則上講，任何流處理元件都可以用於維護物化檢視，儘管“永遠執行”與一些面向分析的框架假設的“主要在有限時間段視窗上執行”背道而馳，Samza 和 Kafka Streams 支援這種用法，建立在 Kafka 對日誌壓縮的支援上【75】。

在流上搜索

除了允許搜尋由多個事件構成模式的 CEP 外，有時也存在基於複雜標準（例如全文搜尋查詢）來搜尋單個事件的需求。

例如，媒體監測服務可以訂閱新聞文章 Feed 與來自媒體的播客，搜尋任何關於公司、產品或感興趣的話題的新聞。這是透過預先構建一個搜尋查詢來完成的，然後不斷地將新聞項的流與該查詢進行匹配。在一些網站上也有類似功能：例如，當市場上出現符合其搜尋條件的新房產時，房地產網站的使用者可以要求網站通知他們。Elasticsearch 的這種過濾器功能，是實現這種流搜尋的一種選擇【76】。

傳統的搜尋引擎首先索引檔案，然後在索引上跑查詢。相比之下，搜尋一個數據流則反了過來：查詢被儲存下來，文件從查詢中流過，就像在 CEP 中一樣。最簡單的情況就是，你可以為每個文件測試每個查詢。但是如果你有大量查詢，這可能會變慢。為了最佳化這個過程，可以像對文件一樣，為查詢建立索引。因而收窄可能匹配的查詢集合【77】。

訊息傳遞和RPC

在“[訊息傳遞中的資料流](#)”中我們討論過，訊息傳遞系統可以作為 RPC 的替代方案，即作為一種服務間通訊的機制，比如在 Actor 模型中所使用的那樣。儘管這些系統也是基於訊息和事件，但我們通常不會將其視作流處理元件：

- Actor 框架主要是管理模組通訊的併發和分散式執行的一種機制，而流處理主要是一種資料管理技術。
- Actor 之間的交流往往是短暫的、一對一的；而事件日誌則是持久的、多訂閱者的。
- Actor 可以以任意方式進行通訊（包括迴圈的請求 / 韻應模式），但流處理通常配置在無環流水線中，其中每個流都是一個特定作業的輸出，由良好定義的輸入流中派生而來。

也就是說，RPC 類系統與流處理之間有一些交叉領域。例如，Apache Storm 有一個稱為 **分散式 RPC** 的功能，它允許將使用者查詢分散到一系列也處理事件流的節點上；然後這些查詢與來自輸入流的事件交織，而結果可以被彙總併發回給使用者【78】（另請參閱“[多分割槽資料處理](#)”）。

也可以使用 Actor 框架來處理流。但是，很多這樣的框架在崩潰時不能保證訊息的傳遞，除非你實現了額外的重試邏輯，否則這種處理不是容錯的。

時間推理解

流處理通常需要與時間打交道，尤其是用於分析目的時候，會頻繁使用時間視窗，例如“過去五分鐘的平均值”。“過去五分鐘”的含義看上去似乎是清晰而無歧義的，但不幸的是，這個概念非常棘手。

在批處理中過程中，大量的歷史事件被快速地處理。如果需要按時間來分析，批處理器需要檢查每個事件中嵌入的時間戳。讀取執行批處理機器的系統時鐘沒有任何意義，因為處理執行的時間與事件實際發生的時間無關。

批處理可以在幾分鐘內讀取一年的歷史事件；在大多數情況下，感興趣的時間線是歷史中的一年，而不是處理中的幾分鐘。而且使用事件中的時間戳，使得處理是 **確定性** 的：在相同的輸入上再次執行相同的處理過程會得到相同的結果（請參閱“[容錯](#)”）。

另一方面，許多流處理框架使用處理機器上的本地系統時鐘（處理時間，即 processing time）來確定 **視窗 (windowing)**【79】。這種方法的優點是簡單，如果事件建立與事件處理之間的延遲可以忽略不計，那也是合理的。然而，如果存在任何顯著的處理延遲——即，事件處理顯著地晚於事件實際發生的時間，這種處理方式就失效了。

事件時間與處理時間

很多原因都可能導致處理延遲：排隊，網路故障（請參閱“[不可靠的網路](#)”），效能問題導致訊息代理 / 訊息處理器出現爭用，流消費者重啟，從故障中恢復時重新處理過去的事件（請參閱“[重播舊訊息](#)”），或者在修復程式碼 BUG 之後。

而且，訊息延遲還可能導致無法預測訊息順序。例如，假設使用者首先發出一個 Web 請求（由 Web 同伺服器 A 處理），然後發出第二個請求（由伺服器 B 處理）。A 和 B 發出描述它們所處理請求的事件，但是 B 的事件在 A 的事件發生之前到達訊息代理。現在，流處理器將首先看到 B 事件，然後看到 A 事件，即使它們實際上是以相反的順序發生的。

有一個類比也許能幫助理解，“星球大戰”電影：第四集於 1977 年發行，第五集於 1980 年，第六集於 1983 年，緊隨其後的是 1999 年的第一集，2002 年的第二集，和 2005 年的第三集，以及 2015 年的第七集【80】ⁱⁱ。如果你按照按照它們上映的順序觀看電影，你處理電影的順序與它們敘事的順序就是不一致的。（集數編號就像事件時間戳，而你

觀看電影的日期就是處理時間) 作為人類，我們能夠應對這種不連續性，但是流處理演算法需要專門編寫，以適應這種時序與順序的問題。

ii. 感謝 Flink 社群的 Kostas Kloudas 提出這個比喻。 ↵

將事件時間和處理時間搞混會導致錯誤的資料。例如，假設你有一個流處理器用於測量請求速率（計算每秒請求數）。如果你重新部署流處理器，它可能會停止一分鐘，並在恢復之後處理積壓的事件。如果你按處理時間來衡量速率，那麼在處理積壓日誌時，請求速率看上去就像有一個異常的突發尖峰，而實際上請求速率是穩定的（圖 11-7）。

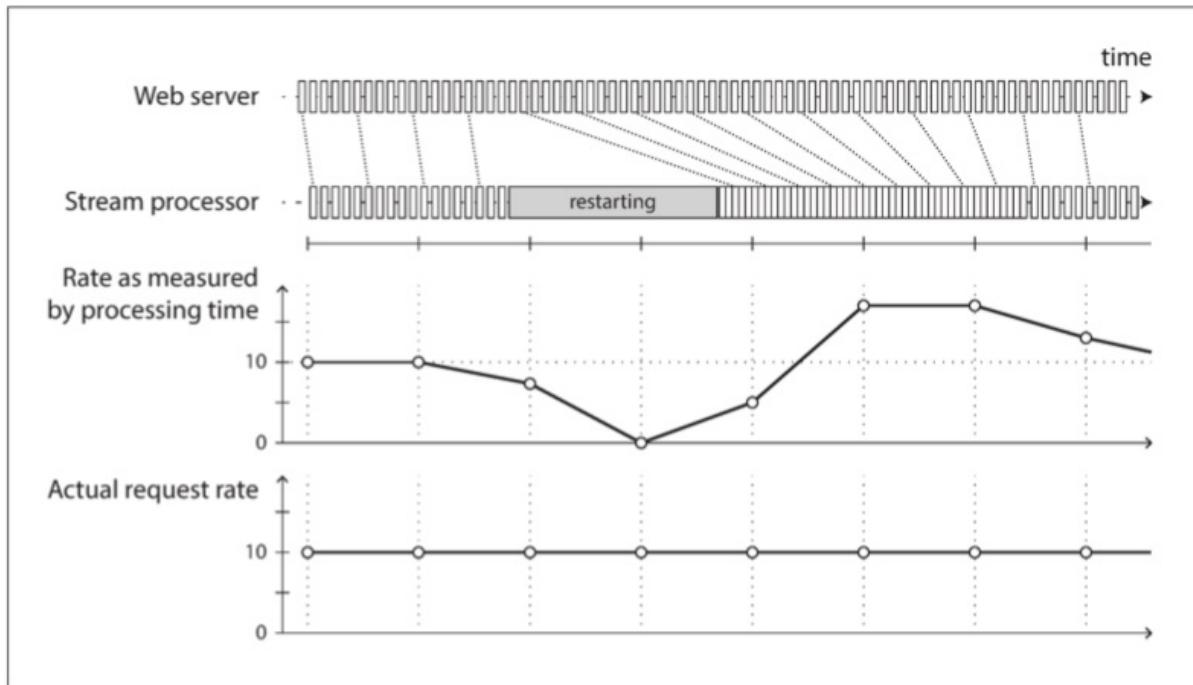


圖 11-7 按處理時間分窗，會因為處理速率的變動引入人為因素

知道什麼時候準備好了

用事件時間來定義視窗的一個棘手的問題是，你永遠也無法確定是不是已經收到了特定視窗的所有事件，還是說還有一些事件正在來的路上。

例如，假設你將事件分組為一分鐘的視窗，以便統計每分鐘的請求數。你已經計數了一些帶有本小時內第 37 分鐘時間戳的事件，時間流逝，現在進入的主要都是本小時內第 38 和第 39 分鐘的事件。什麼時候才能宣佈你已經完成了第 37 分鐘的視窗計數，並輸出其計數器值？

在一段時間沒有看到任何新的事件之後，你可以超時並宣佈一個視窗已經就緒，但仍然可能發生這種情況：某些事件被緩衝在另一臺機器上，由於網路中斷而延遲。你需要能夠處理這種在視窗宣告完成之後到達的滯留（straggler）事件。大體上，你有兩種選擇【1】：

1. 忽略這些滯留事件，因為在正常情況下它們可能只是事件中的一小部分。你可以將丟棄事件的數量作為一個監控指標，並在出現大量丟訊息的情況時報警。
2. 釋出一個更正（correction），一個包括滯留事件的更新視窗值。你可能還需要收回以前的輸出。

在某些情況下，可以使用特殊的訊息來指示“從現在開始，不會有比 t 更早時間戳的訊息了”，消費者可以使用它來觸發視窗【81】。但是，如果不同機器上的多個生產者都在生成事件，每個生產者都有自己的最小小時間戳閾值，則消費者需要分別跟蹤每個生產者。在這種情況下，新增和刪除生產者都是比較棘手的。

你用的是誰的時鐘？

當事件可能在系統內多個地方進行緩衝時，為事件分配時間戳更加困難了。例如，考慮一個移動應用向伺服器上報關於用量的事件。該應用可能會在裝置處於離線狀態時被使用，在這種情況下，它將在裝置本地緩衝事件，並在下一次網際網路連線可用時向伺服器上報這些事件（可能是幾小時甚至幾天）。對於這個流的任意消費者而言，它們就如延遲極大的滯留事件一樣。

在這種情況下，事件上的事件戳實際上應當是使用者交互發生的時間，取決於移動裝置的本地時鐘。然而使用者控制的裝置上的時鐘通常是不可信的，因為它可能會被無意或故意設定成錯誤的時間（請參閱“[時鐘同步與準確性](#)”）。伺服器收到事件的時間（取決於伺服器的時鐘）可能是更準確的，因為伺服器在你的控制之下，但在描述使用者互動方面意義不大。

要校正不正確的裝置時鐘，一種方法是記錄三個時間戳【82】：

- 事件發生的時間，取決於裝置時鐘
- 事件傳送往伺服器的時間，取決於裝置時鐘
- 事件被伺服器接收的時間，取決於伺服器時鐘

透過從第三個時間戳中減去第二個時間戳，可以估算裝置時鐘和伺服器時鐘之間的偏移（假設網路延遲與所需的時間精度相比可忽略不計）。然後可以將該偏移應用於事件時間戳，從而估計事件實際發生的真實時間（假設裝置時鐘偏移在事件發生時與送往伺服器之間沒有變化）。

這並不是流處理獨有的問題，批處理有著完全一樣的時間推理問題。只是在流處理的上下文中，我們更容易意識到時間的流逝。

視窗的型別

當你知道如何確定一個事件的時間戳後，下一步就是如何定義時間段的視窗。然後視窗就可以用於聚合，例如事件計數，或計算視窗內值的平均值。有幾種視窗很常用【79,83】：

- 滾動視窗（Tumbling Window）

滾動視窗有著固定的長度，每個事件都僅能屬於一個視窗。例如，假設你有一個 1 分鐘的滾動視窗，則所有時間戳在 10:03:00 和 10:03:59 之間的事件會被分組到一個視窗中，10:04:00 和 10:04:59 之間的事件被分組到下一個視窗，依此類推。透過將每個事件時間戳四捨五入至最近的分鐘來確定它所屬的視窗，可以實現 1 分鐘的滾動視窗。

- 跳動視窗（Hopping Window）

跳動視窗也有著固定的長度，但允許視窗重疊以提供一些平滑。例如，一個帶有 1 分鐘跳躍步長的 5 分鐘視窗將包含 10:03:00 至 10:07:59 之間的事件，而下一個視窗將覆蓋 10:04:00 至 10:08:59 之間的事件，等等。透過首先計算 1 分鐘的滾動視窗（tumbling window），然後在幾個相鄰視窗上進行聚合，可以實現這種跳動視窗。

- 滑動視窗（Sliding Window）

滑動視窗包含了彼此間距在特定時長內的所有事件。例如，一個 5 分鐘的滑動視窗應當覆蓋 10:03:39 和 10:08:12 的事件，因為它們相距不超過 5 分鐘（注意滾動視窗與步長 5 分鐘的跳動視窗可能不會把這兩個事件分組到同一個視窗中，因為它們使用固定的邊界）。透過維護一個按時間排序的事件緩衝區，並不斷從視窗中移除過期的舊事件，可以實現滑動視窗。

- 會話視窗（Session window）

與其他視窗型別不同，會話視窗沒有固定的持續時間，而定義為：將同一使用者出現時間相近的所有事件分組在一起，而當用戶一段時間沒有活動時（例如，如果 30 分鐘內沒有事件）視窗結束。會話切分是網站分析的常見需求（請參閱“[分組](#)”）。

流連線

在 [第十章](#) 中，我們討論了批處理作業如何透過鍵來連線資料集，以及這種連線是如何成為資料管道的重要組成部分的。由於流處理將資料管道泛化為對無限資料集進行增量處理，因此對流進行連線的需求也是完全相同的。

然而，新事件隨時可能出現在一個流中，這使得流連線要比批處理連線更具挑戰性。為了更好地理解情況，讓我們先來區分三種不同型別的連線：流 - 流 連線，流 - 表 連線，與 表 - 表 連線 [\[84\]](#)。我們將在下面的章節中透過例子來說明。

流流連線（視窗連線）

假設你的網站上有搜尋功能，而你想要找出搜尋 URL 的近期趨勢。每當有人鍵入搜尋查詢時，都會記錄下一個包含查詢與其返回結果的事件。每當有人點選其中一個搜尋結果時，就會記錄另一個記錄點選事件。為了計算搜尋結果中每個 URL 的點選率，你需要將搜尋動作與點選動作的事件連在一起，這些事件透過相同的會話 ID 進行連線。廣告系統中需要類似的分析 [\[85\]](#)。

如果使用者丟棄了搜尋結果，點選可能永遠不會發生，即使它出現了，搜尋與點選之間的時間可能是高度可變的：在很多情況下，它可能是幾秒鐘，但也可能長達幾天或幾周（如果使用者執行搜尋，忘掉了這個瀏覽器頁面，過了一段時間後重新回到這個瀏覽器頁面上，並點選了一個結果）。由於可變的網路延遲，點選事件甚至可能先於搜尋事件到達。你可以選擇合適的連線視窗——例如，如果點選與搜尋之間的時間間隔在一小時內，你可能會選擇連線兩者。

請注意，在點選事件中嵌入搜尋詳情與事件連線並不一樣：這樣做的話，只有當用戶點選了一個搜尋結果時你才能知道，而那些沒有點選的搜尋就無能為力了。為了衡量搜尋質量，你需要準確的點選率，為此搜尋事件和點選事件兩者都是必要的。

為了實現這種型別的連線，流處理器需要維護 狀態：例如，按會話 ID 索引最近一小時內發生的所有事件。無論何時發生搜尋事件或點選事件，都會被新增到合適的索引中，而流處理器也會檢查另一個索引是否有具有相同會話 ID 的事件到達。如果有匹配事件就會發出一個表示搜尋結果被點選的事件；如果搜尋事件直到過期都沒看見有匹配的點選事件，就會發出一個表示搜尋結果未被點選的事件。

流表連線（流擴充）

在“[示例：使用者活動事件分析](#)”（[圖 10-2](#)）中，我們看到了連線兩個資料集的批處理作業示例：一組使用者活動事件和一個使用者檔案資料庫。將使用者活動事件視為流，並在流處理器中連續執行相同的連線是很自然的想法：輸入是包含使用者 ID 的活動事件流，而輸出還是活動事件流，但其中使用者 ID 已經被擴充套件為使用者的檔案資訊。這個過程有時被稱為使用資料庫的資訊來 **擴充**（enriching）活動事件。

要執行此連線，流處理器需要一次處理一個活動事件，在資料庫中查詢事件的使用者 ID，並將檔案資訊新增到活動事件中。資料庫查詢可以透過查詢遠端資料庫來實現。但正如在“[示例：使用者活動事件分析](#)”一節中討論的，此類遠端查詢可能會很慢，並且有可能導致資料庫過載 [\[75\]](#)。

另一種方法是將資料庫副本載入到流處理器中，以便在本地進行查詢而無需網路往返。這種技術與我們在“[Map 側連線](#)”中討論的雜湊連線非常相似：如果資料庫的本地副本足夠小，則可以是記憶體中的散列表，比較大的話也可以是本地磁碟上的索引。

與批處理作業的區別在於，批處理作業使用資料庫的時間點快照作為輸入，而流處理器是長時間執行的，且資料庫的內容可能隨時間而改變，所以流處理器資料庫的本地副本需要保持更新。這個問題可以透過變更資料捕獲來解決：流處理器可以訂閱使用者檔案資料庫的更新日誌，如同活動事件流一樣。當增添或修改檔案時，流處理器會更新其本地副本。因此，我們有了兩個流之間的連線：活動事件和檔案更新。

流表連線實際上非常類似於流流連線；最大的區別在於對於表的變更日誌流，連線使用了一個可以回溯到“時間起點”的視窗（概念上是無限的視窗），新版本的記錄會覆蓋更早的版本。對於輸入的流，連線可能壓根兒就沒有維護任何視窗。

表表連線（維護物化檢視）

我們在“[描述負載](#)”中討論的推特時間線例子時說過，當用戶想要檢視他們的主頁時間線時，迭代使用者所關注人群的推文併合並它們是一個開銷巨大的操作。

相反，我們需要一個時間線快取：一種每個使用者的“收件箱”，在傳送推文的時候寫入這些資訊，因而讀取時間線時只需要簡單地查詢即可。物化與維護這個快取需要處理以下事件：

- 當用戶 u 傳送新的推文時，它將被新增到每個關注使用者 u 的時間線上。
- 使用者刪除推文時，推文將從所有使用者的時間表中刪除。
- 當用戶 $\$u_1\$$ 開始關注使用者 $\$u_2\$$ 時， $\$u_2\$$ 最近的推文將被新增到 $\$u_1\$$ 的時間線上。
- 當用戶 $\$u_1\$$ 取消關注使用者 $\$u_2\$$ 時， $\$u_2\$$ 的推文將從 $\$u_1\$$ 的時間線中移除。

要在流處理器中實現這種快取維護，你需要推文事件流（傳送與刪除）和關注關係事件流（關注與取消關注）。流處理需要維護一個數據庫，包含每個使用者的粉絲集合。以便知道當一條新推文到達時，需要更新哪些時間線【86】。

觀察這個流處理過程的另一種視角是：它維護了一個連線了兩個表（推文與關注）的物化檢視，如下所示：

```
SELECT follows.follower_id AS timeline_id,
       array_agg(tweets.* ORDER BY tweets.timestamp DESC)
  FROM tweets
 JOIN follows ON follows.followee_id = tweets.sender_id
 GROUP BY follows.follower_id
```

流連線直接對應於這個查詢中的表連線。時間線實際上是這個查詢結果的快取，每當底層的表發生變化時都會更新ⁱⁱⁱ。

ⁱⁱⁱ. 如果你將流視作表的衍生物，如 [圖 11-6](#) 所示，而把一個連線看作是兩個表的乘法 $u \cdot v$ ，那麼會發生一些有趣的事情：物化連線的變化流遵循乘積法則： $(u \cdot v)' = u'v + uv'$ 。換句話說，任何推文的變化量都與當前的關注聯絡在一起，任何關注的變化量都與當前的推文相連線【49,50】。[←](#)

連線的時間依賴性

這裡描述的三種連線（流流，流表，表表）有很多共通之處：它們都需要流處理器維護連線一側的一些狀態（搜尋與點選事件，使用者檔案，關注列表），然後當連線另一側的訊息到達時查詢該狀態。

用於維護狀態的事件順序是很重要的（先關注然後取消關注，或者其他類似操作）。在分割槽日誌中，單個分割槽內的事件順序是保留下來的。但典型情況下是沒有跨流或跨分割槽的順序保證的。

這就產生了一個問題：如果不同流中的事件發生在近似的時間範圍內，則應該按照什麼樣的順序進行處理？在流表連線的例子中，如果使用者更新了它們的檔案，哪些活動事件與舊檔案連線（在檔案更新前處理），哪些又與新檔案連線（在檔案更新之後處理）？換句話說：你需要對一些狀態做連線，如果狀態會隨著時間推移而變化，那應當使用什麼時間點來連線呢【45】？

這種時序依賴可能出現在很多地方。例如銷售東西需要對發票應用適當的稅率，這取決於所處的國家 / 州，產品型別，銷售日期（因為稅率時不時會變化）。當連線銷售額與稅率表時，你可能期望的是使用銷售時的稅率參與連線。如果你正在重新處理歷史資料，銷售時的稅率可能和現在的稅率有所不同。

如果跨越流的事件順序是未定的，則連線會變為不確定性的【87】，這意味著你在同樣輸入上重跑相同的作業未必會得到相同的結果：當你重跑任務時，輸入流上的事件可能會以不同的方式交織。

在資料倉庫中，這個問題被稱為 **緩慢變化的維度**（**slowly changing dimension, SCD**），通常透過對特定版本的記錄使用唯一的識別符號來解決：例如，每當稅率改變時都會獲得一個新的識別符號，而發票在銷售時會帶有稅率的識別符號【88,89】。這種變化使連線變為確定性的，但也會導致日誌壓縮無法進行：表中所有的記錄版本都需要保留。

容錯

在本章的最後一節中，讓我們看一看流處理是如何容錯的。我們在 [第十章](#) 中看到，批處理框架可以很容易地容錯：如果 MapReduce 作業中的任務失敗，可以簡單地在另一臺機器上再次啟動，並且丟棄失敗任務的輸出。這種透明的重試是可能的，因為輸入檔案是不可變的，每個任務都將其輸出寫入到 HDFS 上的獨立檔案中，而輸出僅當任務成功完成

後可見。

特別是，批處理容錯方法可確保批處理作業的輸出與沒有出錯的情況相同，即使實際上某些任務失敗了。看起來好像每條輸入記錄都被處理了恰好一次——沒有記錄被跳過，而且沒有記錄被處理兩次。儘管重啟任務意味著實際上可能會多次處理記錄，但輸出中的可見效果看上去就像只處理過一次。這個原則被稱為 **恰好一次語義 (exactly-once semantics)**，儘管 **等效一次 (effectively-once)** 可能會是一個更寫實的術語【90】。

在流處理中也出現了同樣的容錯問題，但是處理起來沒有那麼直觀：等待某個任務完成之後再使其輸出可見並不是一個可行選項，因為你永遠無法處理完一個無限的流。

微批次與存檔點

一個解決方案是將流分解成小塊，並像微型批處理一樣處理每個塊。這種方法被稱為 **微批次 (microbatching)**，它被用於 Spark Streaming 【91】。批次的大小通常約為 1 秒，這是對效能妥協的結果：較小的批次會導致更大的排程與協調開銷，而較大的批次意味著流處理器結果可見之前的延遲要更長。

微批次也隱式提供了一個與批次大小相等的滾動視窗（按處理時間而不是事件時間截分窗）。任何需要更大視窗的作業都需要顯式地將狀態從一個微批次轉移到下一個微批次。

Apache Flink 則使用不同的方法，它會定期生成狀態的滾動存檔點並將其寫入持久儲存【92,93】。如果流運算元崩潰，它可以從最近的存檔點重啟，並丟棄從最近檢查點到崩潰之間的所有輸出。存檔點會由訊息流中的 **壁障 (barrier)** 觸發，類似於微批次之間的邊界，但不會強制一個特定的視窗大小。

在流處理框架的範圍內，微批次與存檔點方法提供了與批處理一樣的 **恰好一次語義**。但是，只要輸出離開流處理器（例如，寫入資料庫，向外部訊息代理傳送訊息，或傳送電子郵件），框架就無法拋棄失敗批次的輸出了。在這種情況下，重啟失敗任務會導致外部副作用發生兩次，只有微批次或存檔點不足以阻止這一問題。

原子提交再現

為了在出現故障時表現出恰好處理一次的樣子，我們需要確保事件處理的所有輸出和副作用 **當且僅當** 處理成功時才會生效。這些影響包括傳送給下游運算元或外部訊息傳遞系統（包括電子郵件或推送通知）的任何訊息，任何資料庫寫入，對運算元狀態的任何變更，以及對輸入訊息的任何確認（包括在基於日誌的訊息代理中將消費者偏移量前移）。

這些事情要麼都原子地發生，要麼都不發生，但是它們不應當失去同步。如果這種方法聽起來很熟悉，那是因為我們在分散式事務和兩階段提交的上下文中討論過它（請參閱“[恰好一次的訊息處理](#)”）。

在 [第九章](#) 中，我們討論了分散式事務傳統實現中的問題（如 XA）。然而在限制更為嚴苛的環境中，也是有可能高效實現這種原子提交機制的。Google Cloud Dataflow 【81,92】和 VoltDB 【94】中使用了這種方法，Apache Kafka 有計劃加入類似功能【95,96】。與 XA 不同，這些實現不會嘗試跨異構技術提供事務，而是透過在流處理框架中同時管理狀態變更與訊息傳遞來內化事務。事務協議的開銷可以透過在單個事務中處理多個輸入訊息來分攤。

幂等性

我們的目標是丟棄任何失敗任務的部分輸出，以便能安全地重試，而不會生效兩次。分散式事務是實現這個目標的一種方式，而另一種方式是依賴 **幂等性 (idempotence)** 【97】。

幂等操作是多次重複執行與單次執行效果相同的操作。例如，將鍵值儲存中的某個鍵設定為某個特定值是幂等的（再次寫入該值，只是用同樣的值替代），而遞增一個計數器不是幂等的（再次執行遞增意味著該值遞增兩次）。

即使一個操作不是天生幂等的，往往可以透過一些額外的元資料做成幂等的。例如，在使用來自 Kafka 的訊息時，每條訊息都有一個持久的、單調遞增的偏移量。將值寫入外部資料庫時可以將這個偏移量帶上，這樣你就可以判斷一條更新是不是已經執行過了，因而避免重複執行。

Storm 的 Trident 基於類似的想法來處理狀態【78】。依賴幂等性意味著隱含了一些假設：重啟一個失敗的任務必須以相同的順序重播相同的訊息（基於日誌的訊息代理能做這些事），處理必須是確定性的，沒有其他節點能同時更新相同的值【98,99】。

當從一個處理節點故障切換到另一個節點時，可能需要進行 **防護** (fencing，請參閱“[領導者和鎖](#)”)，以防止被假死節點干擾。儘管有這麼多注意事項，幕等操作是一種實現 **恰好一次語義** 的有效方式，僅需很小的額外開銷。

失敗後重建狀態

任何需要狀態的流處理——例如，任何視窗聚合（例如計數器，平均值和直方圖）以及任何用於連線的表和索引，都必須確保在失敗之後能恢復其狀態。

一種選擇是將狀態儲存在遠端資料儲存中，並進行複制，然而正如在“[流表連線（流擴充）](#)”中所述，每個訊息都要查詢遠端資料庫可能會很慢。另一種方法是在流處理器本地儲存狀態，並定期複製。然後當流處理器從故障中恢復時，新任務可以讀取狀態副本，恢復處理而不丟失資料。

例如，Flink 定期捕獲運算元狀態的快照，並將它們寫入 HDFS 等持久儲存中【92,93】。Samza 和 Kafka Streams 透過將狀態變更傳送到具有日誌壓縮功能的專用 Kafka 主題來複制狀態變更，這與變更資料捕獲類似【84,100】。

VoltDB 透過在多個節點上對每個輸入訊息進行冗餘處理來複制狀態（請參閱“[真的序列執行](#)”）。

在某些情況下，甚至可能都不需要複製狀態，因為它可以從輸入流重建。例如，如果狀態是從相當短的視窗中聚合而成，則簡單地重播該視窗中的輸入事件可能是足夠快的。如果狀態是透過變更資料捕獲來維護的資料庫的本地副本，那麼也可以從日誌壓縮的變更流中重建資料庫（請參閱“[日誌壓縮](#)”）。

然而，所有這些權衡取決於底層基礎架構的效能特徵：在某些系統中，網路延遲可能低於磁碟訪問延遲，網路頻寬也可能與磁碟頻寬相當。沒有針對所有情況的普適理想權衡，隨著儲存和網路技術的發展，本地狀態與遠端狀態的優點也可能會互換。

本章小結

在本章中，我們討論了事件流，它們所服務的目的，以及如何處理它們。在某些方面，流處理非常類似於在 [第十章](#) 中討論的批處理，不過是在無限的（永無止境的）流而不是固定大小的輸入上持續進行。從這個角度來看，訊息代理和事件日誌可以視作檔案系統的流式等價物。

我們花了一些時間比較兩種訊息代理：

- AMQP/JMS 風格的訊息代理

代理將單條訊息分配給消費者，消費者在成功處理單條訊息後確認訊息。訊息被確認後從代理中刪除。這種方法適合作為一種非同步形式的 RPC（另請參閱“[訊息傳遞中的資料流](#)”），例如在任務佇列中，訊息處理的確切順序並不重要，而且訊息在處理完之後，不需要回頭重新讀取舊訊息。

- 基於日誌的訊息代理

代理將一個分割槽中的所有訊息分配給同一個消費者節點，並始終以相同的順序傳遞訊息。並行是透過分割槽實現的，消費者透過存檔最近處理訊息的偏移量來跟蹤工作進度。訊息代理將訊息保留在磁碟上，因此如有必要的話，可以回跳並重新讀取舊訊息。

基於日誌的方法與資料庫中的複製日誌（請參閱 [第五章](#)）和日誌結構儲存引擎（請參閱 [第三章](#)）有相似之處。我們看到，這種方法對於消費輸入流，併產生衍生狀態或衍生輸出資料流的系統而言特別適用。

就流的來源而言，我們討論了幾種可能性：使用者活動事件，定期讀數的感測器，和 Feed 資料（例如，金融中的市場資料）能夠自然地表示為流。我們發現將資料庫寫入視作流也是很有用的：我們可以捕獲變更日誌——即對資料庫所做的所有變更的歷史記錄——隱式地透過變更資料捕獲，或顯式地透過事件溯源。日誌壓縮允許流也能保有資料庫內容的完整副本。

將資料庫表示為流為系統整合帶來了很多強大機遇。透過消費變更日誌並將其應用至衍生系統，你能使諸如搜尋索引、快取以及分析系統這類衍生資料系統不斷保持更新。你甚至能從頭開始，透過讀取從創世至今的所有變更日誌，為現有資料建立全新的檢視。

像流一樣維護狀態以及訊息重播的基礎設施，是在各種流處理框架中實現流連線和容錯的基礎。我們討論了流處理的幾種目的，包括搜尋事件模式（複雜事件處理），計算分窗聚合（流分析），以及保證衍生資料系統處於最新狀態（物化檢視）。

然後我們討論了在流處理中對時間進行推理的困難，包括處理時間與事件時間戳之間的區別，以及當你認為視窗已經完事之後，如何處理到達的掉隊事件的問題。

我們區分了流處理中可能出現的三種連線型別：

- 流流連線

兩個輸入流都由活動事件組成，而連線運算元在某個時間視窗內搜尋相關的事件。例如，它可能會將同一個使用者 30 分鐘內進行的兩個活動聯絡在一起。如果你想要找出一個流內的相關事件，連線的兩側輸入可能實際上都是同一個流（自連線，即 self-join）。

- 流表連線

一個輸入流由活動事件組成，另一個輸入流是資料庫變更日誌。變更日誌保證了資料庫的本地副本是最新的。對於每個活動事件，連線運算元將查詢資料庫，並輸出一個擴充套件的活動事件。

- 表表連線

兩個輸入流都是資料庫變更日誌。在這種情況下，一側的每一個變化都與另一側的最新狀態相連線。結果是兩表連線所得物化檢視的變更流。

最後，我們討論了在流處理中實現容錯和恰好一次語義的技術。與批處理一樣，我們需要放棄任何失敗任務的部分輸出。然而由於流處理長時間執行並持續產生輸出，所以不能簡單地丟棄所有的輸出。相反，可以使用更細粒度的恢復機制，基於微批次、存檔點、事務或幕等寫入。

參考文獻

1. Tyler Akidau, Robert Bradshaw, Craig Chambers, et al.: “The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-Scale, Unbounded, Out-of-Order Data Processing,” *Proceedings of the VLDB Endowment*, volume 8, number 12, pages 1792–1803, August 2015. doi:10.14778/2824032.2824076
2. Harold Abelson, Gerald Jay Sussman, and Julie Sussman: *Structure and Interpretation of Computer Programs*, 2nd edition. MIT Press, 1996. ISBN: 978-0-262-51087-5, available online at mitpress.mit.edu
3. Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec: “The Many Faces of Publish/Subscribe,” *ACM Computing Surveys*, volume 35, number 2, pages 114–131, June 2003. doi:10.1145/857076.857078
4. Joseph M. Hellerstein and Michael Stonebraker: *Readings in Database Systems*, 4th edition. MIT Press, 2005. ISBN: 978-0-262-69314-1, available online at redbook.cs.berkeley.edu
5. Don Carney, Uğur Çetintemel, Mitch Cherniack, et al.: “Monitoring Streams – A New Class of Data Management Applications,” at *28th International Conference on Very Large Data Bases* (VLDB), August 2002.
6. Matthew Sackman: “Pushing Back,” *Ishift.net*, May 5, 2016.
7. Vicent Martí: “Brubeck, a statsd-Compatible Metrics Aggregator,” githubengineering.com, June 15, 2015.
8. Seth Lowenberger: “MoldUDP64 Protocol Specification V 1.00,” nasdaqtrader.com, July 2009.
9. Pieter Hintjens: *ZeroMQ – The Guide*. O'Reilly Media, 2013. ISBN: 978-1-449-33404-8
10. Ian Malpass: “Measure Anything, Measure Everything,” codeascraft.com, February 15, 2011.
11. Dieter Plaetinck: “25 Graphite, Grafana and statsd Gotchas,” blog.raintank.io, March 3, 2016.
12. Jeff Lindsay: “Web Hooks to Revolutionize the Web,” program.com, May 3, 2007.
13. Jim N. Gray: “Queues Are Databases,” Microsoft Research Technical Report MSR-TR-95-56, December 1995.
14. Mark Hapner, Rich Burridge, Rahul Sharma, et al.: “JSR-343 Java Message Service (JMS) 2.0 Specification,” jms-spec.java.net, March 2013.
15. Sanjay Aiyagari, Matthew Arrott, Mark Atwell, et al.: “AMQP: Advanced Message Queuing Protocol Specification,”

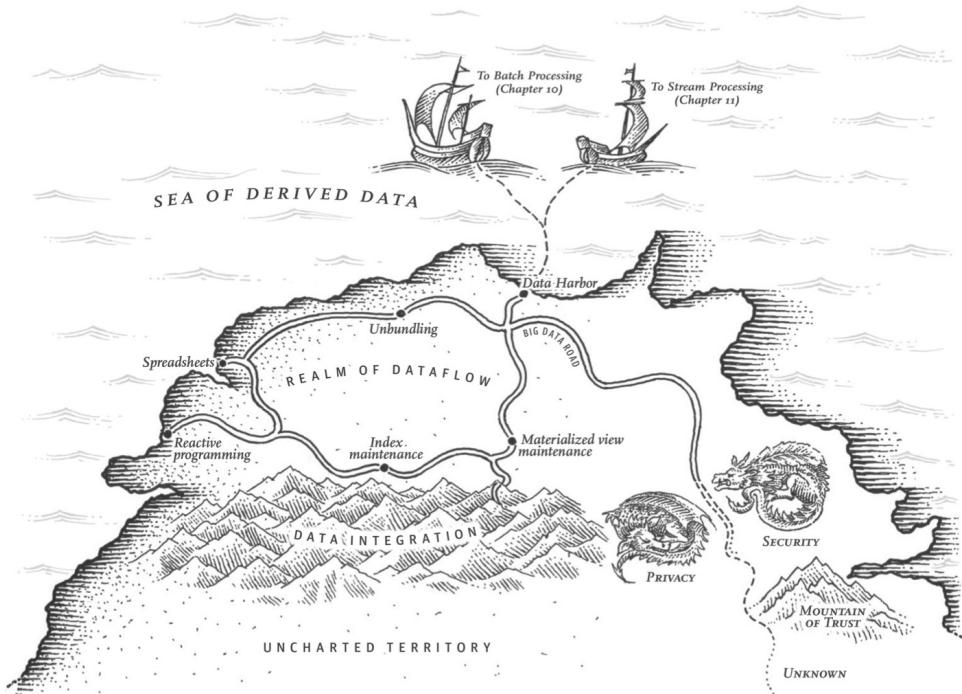
- Version 0-9-1, November 2008.
16. “Google Cloud Pub/Sub: A Google-Scale Messaging Service,” cloud.google.com, 2016.
 17. “Apache Kafka 0.9 Documentation,” kafka.apache.org, November 2015.
 18. Jay Kreps, Neha Narkhede, and Jun Rao: “Kafka: A Distributed Messaging System for Log Processing,” at *6th International Workshop on Networking Meets Databases* (NetDB), June 2011.
 19. “Amazon Kinesis Streams Developer Guide,” docs.aws.amazon.com, April 2016.
 20. Leigh Stewart and Sijie Guo: “Building DistributedLog: Twitter’s High-Performance Replicated Log Service,” blog.twitter.com, September 16, 2015.
 21. “DistributedLog Documentation,” Twitter, Inc., distributedlog.io, May 2016.
 22. Jay Kreps: “Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines),” engineering.linkedin.com, April 27, 2014.
 23. Kartik Paramasivam: “How We’re Improving and Advancing Kafka at LinkedIn,” engineering.linkedin.com, September 2, 2015.
 24. Jay Kreps: “The Log: What Every Software Engineer Should Know About Real-Time Data’s Unifying Abstraction,” engineering.linkedin.com, December 16, 2013.
 25. Shirshanka Das, Chavdar Botev, Kapil Surlaker, et al.: “All Aboard the Databus!,” at *3rd ACM Symposium on Cloud Computing* (SoCC), October 2012.
 26. Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, et al.: “Wormhole: Reliable Pub-Sub to Support Geo-Replicated Internet Services,” at *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.
 27. P. P. S. Narayan: “Sherpa Update,” developer.yahoo.com, June 8, .
 28. Martin Kleppmann: “Bottled Water: Real-Time Integration of PostgreSQL and Kafka,” martin.kleppmann.com, April 23, 2015.
 29. Ben Osherooff: “Introducing Maxwell, a mysql-to-kafka Binlog Processor,” developer.zendesk.com, August 20, 2015.
 30. Randall Hauch: “Debezium 0.2.1 Released,” debezium.io, June 10, 2016.
 31. Prem Santosh Udaya Shankar: “Streaming MySQL Tables in Real-Time to Kafka,” engineeringblog.yelp.com, August 1, 2016.
 32. “MongoRiver,” Stripe, Inc., github.com, September 2014.
 33. Dan Harvey: “Change Data Capture with Mongo + Kafka,” at *Hadoop Users Group UK*, August 2015.
 34. “Oracle GoldenGate 12c: Real-Time Access to Real-Time Information,” Oracle White Paper, March 2015.
 35. “Oracle GoldenGate Fundamentals: How Oracle GoldenGate Works,” Oracle Corporation, youtube.com, November 2012.
 36. Slava Akhmechet: “Advancing the Realtime Web,” rethinkdb.com, January 27, 2015.
 37. “Firebase Realtime Database Documentation,” Google, Inc., firebase.google.com, May 2016.
 38. “Apache CouchDB 1.6 Documentation,” docs.couchdb.org, 2014.
 39. Matt DeBergalis: “Meteor 0.7.0: Scalable Database Queries Using MongoDB Oplog Instead of Poll-and-Diff,” info.meteor.com, December 17, 2013.
 40. “Chapter 15. Importing and Exporting Live Data,” VoltDB 6.4 User Manual, docs.voltdb.com, June 2016.
 41. Neha Narkhede: “Announcing Kafka Connect: Building Large-Scale Low-Latency Data Pipelines,” confluent.io, February 18, 2016.
 42. Greg Young: “CQRS and Event Sourcing,” at *Code on the Beach*, August 2014.
 43. Martin Fowler: “Event Sourcing,” martinfowler.com, December 12, 2005.
 44. Vaughn Vernon: *Implementing Domain-Driven Design*. Addison-Wesley Professional, 2013. ISBN: 978-0-321-83457-7
 45. H. V. Jagadish, Inderpal Singh Mumick, and Abraham Silberschatz: “View Maintenance Issues for the Chronicle Data Model,” at *14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (PODS), May 1995. doi:[10.1145/212433.220201](https://doi.org/10.1145/212433.220201)
 46. “Event Store 3.5.0 Documentation,” Event Store LLP, docs.geteventstore.com, February 2016.
 47. Martin Kleppmann: *Making Sense of Stream Processing*. Report, O’Reilly Media, May 2016.
 48. Sander Mak: “Event-Sourced Architectures with Akka,” at *JavaOne*, September 2014.

49. Julian Hyde: [personal communication](#), June 2016.
50. Ashish Gupta and Inderpal Singh Mumick: *Materialized Views: Techniques, Implementations, and Applications*. MIT Press, 1999. ISBN: 978-0-262-57122-7
51. Timothy Griffin and Leonid Libkin: “[Incremental Maintenance of Views with Duplicates](#),” at *ACM International Conference on Management of Data* (SIGMOD), May 1995. doi:[10.1145/223784.223849](https://doi.org/10.1145/223784.223849)
52. Pat Helland: “[Immutability Changes Everything](#),” at *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.
53. Martin Kleppmann: “[Accounting for Computer Scientists](#),” martin.kleppmann.com, March 7, 2011.
54. Pat Helland: “[Accountants Don't Use Erasers](#),” blogs.msdn.com, June 14, 2007.
55. Fangjin Yang: “[Dogfooding with Druid, Samza, and Kafka: Metametrics at Metamarkets](#),” metamarkets.com, June 3, 2015.
56. Gavin Li, Jianqiu Lv, and Hang Qi: “[Pistachio: Co-Locate the Data and Compute for Fastest Cloud Compute](#),” yahoohadoop.tumblr.com, April 13, 2015.
57. Kartik Paramasivam: “[Stream Processing Hard Problems – Part 1: Killing Lambda](#),” engineering.linkedin.com, June 27, 2016.
58. Martin Fowler: “[CQRS](#),” martinfowler.com, July 14, 2011.
59. Greg Young: “[CQRS Documents](#),” cqrsrcodes.wordpress.com, November 2010.
60. Baron Schwartz: “[Immutability, MVCC, and Garbage Collection](#),” xaprb.com, December 28, 2013.
61. Daniel Eloff, Slava Akhmechet, Jay Kreps, et al.: “[Re: Turning the Database Inside-out with Apache Samza](#),” *Hacker News discussion*, news.ycombinator.com, March 4, 2015.
62. “[Datomic Development Resources: Excision](#),” Cognitect, Inc., docs.datomic.com.
63. “[Fossil Documentation: Deleting Content from Fossil](#),” fossil-scm.org, 2016.
64. Jay Kreps: “[The irony of distributed systems is that data loss is really easy but deleting data is surprisingly hard](#),” twitter.com, March 30, 2015.
65. David C. Luckham: “[What's the Difference Between ESP and CEP?](#),” complexevents.com, August 1, 2006.
66. Srinath Perera: “[How Is Stream Processing and Complex Event Processing \(CEP\) Different?](#),” quora.com, December 3, 2015.
67. Arvind Arasu, Shivnath Babu, and Jennifer Widom: “[The CQL Continuous Query Language: Semantic Foundations and Query Execution](#),” *The VLDB Journal*, volume 15, number 2, pages 121–142, June 2006. doi:[10.1007/s00778-004-0147-z](https://doi.org/10.1007/s00778-004-0147-z)
68. Julian Hyde: “[Data in Flight: How Streaming SQL Technology Can Help Solve the Web 2.0 Data Crunch](#),” *ACM Queue*, volume 7, number 11, December 2009. doi:[10.1145/1661785.1667562](https://doi.org/10.1145/1661785.1667562)
69. “[Esper Reference, Version 5.4.0](#),” EsperTech, Inc., espertech.com, April 2016.
70. Zubair Nabi, Eric Bouillet, Andrew Bainbridge, and Chris Thomas: “[Of Streams and Storms](#),” IBM technical report, developer.ibm.com, April 2014.
71. Milinda Pathirage, Julian Hyde, Yi Pan, and Beth Plale: “[SamzaSQL: Scalable Fast Data Management with Streaming SQL](#),” at *IEEE International Workshop on High-Performance Big Data Computing* (HPBDC), May 2016. doi:[10.1109/IPDPSW.2016.141](https://doi.org/10.1109/IPDPSW.2016.141)
72. Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier: “[HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm](#),” at *Conference on Analysis of Algorithms* (AofA), June 2007.
73. Jay Kreps: “[Questioning the Lambda Architecture](#),” oreilly.com, July 2, 2014.
74. Ian Hellström: “[An Overview of Apache Streaming Technologies](#),” databaseline.wordpress.com, March 12, 2016.
75. Jay Kreps: “[Why Local State Is a Fundamental Primitive in Stream Processing](#),” oreilly.com, July 31, 2014.
76. Shay Banon: “[Percolator](#),” elastic.co, February 8, 2011.
77. Alan Woodward and Martin Kleppmann: “[Real-Time Full-Text Search with Luwak and Samza](#),” martin.kleppmann.com, April 13, 2015.
78. “[Apache Storm 1.0.1 Documentation](#),” storm.apache.org, May 2016.
79. Tyler Akidau: “[The World Beyond Batch: Streaming 102](#),” oreilly.com, January 20, 2016.
80. Stephan Ewen: “[Streaming Analytics with Apache Flink](#),” at *Kafka Summit*, April 2016.
81. Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, et al.: “[MillWheel: Fault-Tolerant Stream Processing at Internet Scale](#),” at *39th International Conference on Very Large Data Bases* (VLDB), August 2013.

82. Alex Dean: “[Improving Snowplow's Understanding of Time](#),” snowplowanalytics.com, September 15, 2015.
83. “[Windowing \(Azure Stream Analytics\)](#),” Microsoft Azure Reference, msdn.microsoft.com, April 2016.
84. “[State Management](#),” Apache Samza 0.10 Documentation, samza.apache.org, December 2015.
85. Rajagopal Ananthanarayanan, Venkatesh Basker, Sumit Das, et al.: “[Photon: Fault-Tolerant and Scalable Joining of Continuous Data Streams](#),” at *ACM International Conference on Management of Data* (SIGMOD), June 2013. doi:[10.1145/2463676.2465272](https://doi.org/10.1145/2463676.2465272)
86. Martin Kleppmann: “[Samza Newsfeed Demo](#),” github.com, September 2014.
87. Ben Kirwin: “[Doing the Impossible: Exactly-Once Messaging Patterns in Kafka](#),” ben.kirw.in, November 28, 2014.
88. Pat Helland: “[Data on the Outside Versus Data on the Inside](#),” at *2nd Biennial Conference on Innovative Data Systems Research* (CIDR), January 2005.
89. Ralph Kimball and Margy Ross: *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling*, 3rd edition. John Wiley & Sons, 2013. ISBN: 978-1-118-53080-1
90. Viktor Klang: “[I'm coining the phrase 'effectively-once' for message processing with at-least-once + idempotent operations](#),” twitter.com, October 20, 2016.
91. Matei Zaharia, Tathagata Das, Haoyuan Li, et al.: “[Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters](#),” at *4th USENIX Conference in Hot Topics in Cloud Computing* (HotCloud), June 2012.
92. Kostas Tzoumas, Stephan Ewen, and Robert Metzger: “[High-Throughput, Low-Latency, and Exactly-Once Stream Processing with Apache Flink](#),” data-artisans.com, August 5, 2015.
93. Paris Carbone, Gyula Fóra, Stephan Ewen, et al.: “[Lightweight Asynchronous Snapshots for Distributed Dataflows](#),” arXiv:1506.08603 [cs.DC], June 29, 2015.
94. Ryan Betts and John Hugg: *Fast Data: Smart and at Scale*. Report, O'Reilly Media, October 2015.
95. Flavio Junqueira: “[Making Sense of Exactly-Once Semantics](#),” at *Strata+Hadoop World London*, June 2016.
96. Jason Gustafson, Flavio Junqueira, Apurva Mehta, Sriram Subramanian, and Guozhang Wang: “[KIP-98 – Exactly Once Delivery and Transactional Messaging](#),” cwiki.apache.org, November 2016.
97. Pat Helland: “[Idempotence Is Not a Medical Condition](#),” *Communications of the ACM*, volume 55, number 5, page 56, May 2012. doi:[10.1145/2160718.2160734](https://doi.org/10.1145/2160718.2160734)
98. Jay Kreps: “[Re: Trying to Achieve Deterministic Behavior on Recovery/Rewind](#),” email to samza-dev mailing list, September 9, 2014.
99. E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson: “[A Survey of Rollback-Recovery Protocols in Message-Passing Systems](#),” *ACM Computing Surveys*, volume 34, number 3, pages 375–408, September 2002. doi:[10.1145/568522.568525](https://doi.org/10.1145/568522.568525)
00. Adam Warski: “[Kafka Streams – How Does It Fit the Stream Processing Landscape?](#),” softwaremill.com, June 1, 2016.

上一章	目錄	下一章
第十章：批處理	設計資料密集型應用	第十二章：資料系統的未來

第十二章：資料系統的未來



如果船長的終極目標是保護船隻，他應該永遠待在港口。

—— 聖托馬斯·阿奎那《神學大全》 (1265-1274)

[TOC]

到目前為止，本書主要描述的是 **現狀**。在這最後一章中，我們將放眼 **未來**，討論應該是怎麼樣的：我將提出一些想法與方法，我相信它們能從根本上改進我們設計與構建應用的方式。

對未來的看法與推測當然具有很大的主觀性。所以在撰寫本章時，當提及我個人的觀點時會使用第一人稱。你完全可以不同意這些觀點並提出自己的看法，但我希望本章中的概念，至少能成為富有成效的討論出發點，並澄清一些經常被混淆的概念。

第一章 概述了本書的目標：探索如何建立 **可靠**、**可伸縮** 和 **可維護** 的應用與系統。這一主題貫穿了所有的章節：例如，我們討論了許多有助於提高可靠性的容錯演算法，有助於提高可伸縮性的分割槽，以及有助於提高可維護性的演化與抽象機制。在本章中，我們將把所有這些想法結合在一起，並在它們的基礎上展望未來。我們的目標是，發現如何設計出比現有應用更好的應用——健壯、正確、可演化、且最終對人類有益。

資料整合

本書中反覆出現的主題是，對於任何給定的問題都會有好幾種解決方案，所有這些解決方案都有不同的優缺點與利弊權衡。例如在 **第三章** 討論儲存引擎時，我們看到了日誌結構儲存、B樹以及列式儲存。在 **第五章** 討論複製時，我們看到了單領導者、多領導者和無領導者的方法。

如果你有一個類似於“我想儲存一些資料並稍後再查詢”的問題，那麼並沒有一種正確的解決方案。但對於不同的具體環境，總會有不同的合適方法。軟體實現通常必須選擇一種特定的方法。使單條程式碼路徑能做到穩定健壯且表現良好已經是一件非常困難的事情了——嘗試在單個軟體中完成所有事情，幾乎可以保證，實現效果會很差。

因此軟體工具的最佳選擇也取決於情況。每一種軟體，甚至所謂的“通用”資料庫，都是針對特定的使用模式設計的。

面對讓人眼花繚亂的諸多替代品，第一個挑戰就是弄清軟體與其適用環境的對映關係。供應商不願告訴你他們軟體不適用的工作負載，這是可以理解的。但是希望先前的章節能給你提供一些問題，讓你讀出字裡行間的言外之意，並更好地理解這些權衡。

但是，即使你已經完全理解各種工具與其適用環境間的關係，還有一個挑戰：在複雜的應用中，資料的用法通常花樣百出。不太可能存在適用於所有不同資料應用場景的軟體，因此你不可避免地需要拼湊幾個不同的軟體來以提供應用所需的功能。

組合使用衍生資料的工具

例如，為了處理任意關鍵詞的搜尋查詢，將 OLTP 資料庫與全文搜尋索引整合在一起是很常見的需求。儘管一些資料庫（例如 PostgreSQL）包含了全文索引功能，對於簡單的應用完全夠了【1】，但更複雜的搜尋能力就需要專業的資訊檢索工具了。相反的是，搜尋索引通常不適合作為持久的記錄系統，因此許多應用需要組合這兩種不同的工具以滿足所有需求。

我們在“[保持系統同步](#)”中接觸過整合資料系統的問題。隨著資料不同表示形式的增加，整合問題變得越來越困難。除了資料庫和搜尋索引之外，也許你需要在分析系統（資料倉庫，或批處理和流處理系統）中維護資料副本；維護從原始資料中衍生的快取，或反規範化的資料版本；將資料灌入機器學習、分類、排名或推薦系統中；或者基於資料變更傳送通知。

令人驚訝的是，我經常看到軟體工程師做出這樣的陳述：“根據我的經驗，99% 的人只需要 X”或者“.....不需要 X”（對於各種各樣的 X）。我認為這種陳述更像是發言人自己的經驗，而不是技術實際上的實用性。可能對資料執行的操作，其範圍極其寬廣。某人認為雞肋而毫無意義的功能可能是別人的核心需求。當你拉高視角，並考慮跨越整個組織範圍的資料流時，資料整合的需求往往就會變得明顯起來。

理解資料流

當需要在多個儲存系統中維護相同資料的副本以滿足不同的訪問模式時，你要對輸入和輸出瞭如指掌：哪些資料先寫入，哪些資料表示衍生自哪些來源？如何以正確的格式，將所有資料匯入正確的地方？

例如，你可能會首先將資料寫入 [記錄系統](#) 資料庫，捕獲對該資料庫所做的變更（請參閱“[變更資料捕獲](#)”），然後將變更以相同的順序應用於搜尋索引。如果變更資料捕獲（CDC）是更新索引的唯一方式，則可以確定該索引完全派生自記錄系統，因此與其保持一致（除軟體錯誤外）。寫入資料庫是向該系統提供新輸入的唯一方式。

允許應用程式直接寫入搜尋索引和資料庫引入瞭如 [圖 11-4](#) 所示的問題，其中兩個客戶端同時傳送衝突的寫入，且兩個儲存系統按不同順序處理它們。在這種情況下，既不是資料庫說了算，也不是搜尋索引說了算，所以它們做出了相反的決定，進入彼此間永續性的不一致狀態。

如果你可以透過單個系統來提供所有使用者輸入，從而決定所有寫入的排序，則透過按相同順序處理寫入，可以更容易地衍生出其他資料表示。這是狀態機複製方法的一個應用，我們在“[全序廣播](#)”中看到。無論你使用變更資料捕獲還是事件溯源日誌，都不如簡單的基於全序的決策原則更重要。

基於事件日誌來更新衍生資料的系統，通常可以做到 [確定性](#) 與 [冪等性](#)（請參閱“[冪等性](#)”），使得從故障中恢復相當容易。

衍生資料與分散式事務

保持不同資料系統彼此一致的經典方法涉及分散式事務，如“[原子提交與兩階段提交](#)”中所述。與分散式事務相比，使用衍生資料系統的方法如何？

在抽象層面，它們透過不同的方式達到類似的目標。分散式事務透過鎖進行互斥來決定寫入的順序（請參閱“[兩階段鎖定](#)”），而 CDC 和事件溯源使用日誌進行排序。分散式事務使用原子提交來確保變更只生效一次，而基於日誌的系統通常基於確定性重試和幂等性。

最大的不同之處在於事務系統通常提供線性一致性，這包含著有用的保證，例如[讀己之寫](#)。另一方面，衍生資料系統通常是非同步更新的，因此它們預設不會提供相同的時序保證。

在願意為分散式事務付出代價的有限場景中，它們已被成功應用。但是，我認為 XA 的容錯能力和效能很差勁（請參閱“[實踐中的分散式事務](#)”），這嚴重限制了它的實用性。我相信為分散式事務設計一種更好的協議是可行的。但使這樣一種協議被現有工具廣泛接受是很有挑戰的，且不是立竿見影的事。

在沒有廣泛支援的良好分散式事務協議的情況下，我認為基於日誌的衍生資料是整合不同資料系統的最有前途的方法。然而，諸如讀己之寫的保證是有用的，我認為告訴所有人“最終一致性是不可避免的——忍一忍並學會和它打交道”是沒有什麼建設性的（至少在缺乏如何應對的良好指導時）。

在“[將事情做正確](#)”中，我們將討論一些在非同步衍生系統之上實現更強保障的方法，並邁向分散式事務和基於日誌的非同步系統之間的中間地帶。

全序的限制

對於足夠小的系統，構建一個完全有序的事件日誌是完全可行的（正如單主複製資料庫的流行所證明的那樣，它正好建立了這樣一種日誌）。但是，隨著系統向更大更複雜的工作負載伸縮，限制開始出現：

- 在大多數情況下，構建完全有序的日誌，需要所有事件彙集於決定順序的單個領導者節點。如果事件吞吐量大於單臺計算機的處理能力，則需要將其分割槽到多臺計算機上（請參閱“[分割槽日誌](#)”）。然後兩個不同分割槽中的事件順序關係就不明確了。
- 如果伺服器分佈在多個地理位置分散的資料中心上，例如為了容忍整個資料中心掉線，你通常在每個資料中心都有單獨的主庫，因為網路延遲會導致同步的跨資料中心協調效率低下（請參閱“[多主複製](#)”）。這意味著源自兩個不同資料中心的事件順序未定義。
- 將應用程式部署為微服務時（請參閱“[服務中的資料流：REST 與 RPC](#)”），常見的設計選擇是將每個服務及其持久狀態作為獨立單元進行部署，服務之間不共享持久狀態。當兩個事件來自不同的服務時，這些事件間的順序未定義。
- 某些應用程式在客戶端儲存狀態，該狀態在使用者輸入時立即更新（無需等待伺服器確認），甚至可以繼續離線工作（請參閱“[需要離線操作的客戶端](#)”）。對於這樣的應用程式，客戶端和伺服器很可能以不同的順序看到事件。

在形式上，決定事件的全域性順序稱為全序廣播，相當於共識（請參閱“[共識演算法和全序廣播](#)”）。大多數共識演算法都是針對單個節點的吞吐量足以處理整個事件流的情況而設計的，並且這些演算法不提供多個節點共享事件排序工作的機制。設計可以伸縮至單個節點的吞吐量之上，且在地理位置分散的環境中仍然工作良好的共識演算法仍然是一個開放的研究問題。

排序事件以捕獲因果關係

在事件之間不存在因果關係的情況下，全序的缺乏並不是一個大問題，因為併發事件可以任意排序。其他一些情況很容易處理：例如，當同一物件有多個更新時，它們可以透過將特定物件 ID 的所有更新路由到相同的日誌分割槽來完全排序。然而，因果關係有時會以更微妙的方式出現（請參閱“[順序與因果關係](#)”）。

例如，考慮一個社交網路服務，以及一對曾處於戀愛關係但剛分手的使用者。其中一個使用者將另一個使用者從好友中移除，然後向剩餘的好友傳送訊息，抱怨他們的前任。使用者的心思是他們的前任不應該看到這些粗魯的訊息，因為訊息是在好友狀態解除後傳送的。

但是如果好友關係狀態與訊息儲存在不同的地方，在這樣一個系統中，可能會出現解除好友事件與傳送訊息事件之間的因果依賴丟失的情況。如果因果依賴關係沒有被捕捉到，則傳送有關新訊息的通知的服務可能會在解除好友事件之前處理傳送訊息事件，從而錯誤地向前任傳送通知。

在本例中，通知實際上是訊息和好友列表之間的連線，使得它與我們先前討論的連線的時序問題有關（請參閱“[連線的時間依賴性](#)”）。不幸的是，這個問題似乎並沒有一個簡單的答案【2,3】。起點包括：

- 邏輯時間戳可以提供無需協調的全域性順序（請參閱“[序列號順序](#)”），因此它們可能有助於全序廣播不可行的情況。但是，他們仍然要求收件人處理不按順序傳送的事件，並且需要傳遞其他元資料。
- 如果你可以記錄一個事件來記錄使用者在做出決定之前所看到的系統狀態，並給該事件一個唯一的識別符號，那麼後面的任何事件都可以引用該事件識別符號來記錄因果關係【4】。我們將在“[讀也是事件](#)”中回到這個想法。
- 衝突解決演算法（請參閱“[自動衝突解決](#)”）有助於處理以意外順序傳遞的事件。它們對於維護狀態很有用，但如果行為有外部副作用（例如，給使用者傳送通知），就沒什麼幫助了。

也許，隨著時間的推移，應用開發模式將出現，使得能夠有效地捕獲因果依賴關係，並且保持正確的衍生狀態，而不會迫使所有事件經歷全序廣播的瓶頸）。

批處理與流處理

我會說資料整合的目標是，確保資料最終能在所有正確的地方表現出正確的形式。這樣做需要消費輸入、轉換、連線、過濾、聚合、訓練模型、評估、以及最終寫出適當的輸出。批處理和流處理是實現這一目標的工具。

批處理和流處理的輸出是衍生資料集，例如搜尋索引、物化檢視、向用戶顯示的建議、聚合指標等（請參閱“[批處理工作流的輸出](#)”和“[流處理的應用](#)”）。

正如我們在[第十章](#)和[第十一章](#)中看到的，批處理和流處理有許多共同的原則，主要的根本區別在於流處理器在無限資料集上執行，而批處理輸入是已知的有限大小。處理引擎的實現方式也有很多細節上的差異，但是這些區別已經開始模糊。

Spark 在批處理引擎上執行流處理，將流分解為 **微批次 (microbatches)**，而 Apache Flink 則在流處理引擎上執行批處理【5】。原則上，一種型別的處理可以用另一種型別來模擬，但是效能特徵會有所不同：例如，在跳躍或滑動視窗上，微批次可能表現不佳【6】。

維護衍生狀態

批處理有著很強的函式式風格（即使其程式碼不是用函式式語言編寫的）：它鼓勵確定性的純函式，其輸出僅依賴於輸入，除了顯式輸出外沒有副作用，將輸入視為不可變的，且輸出是僅追加的。流處理與之類似，但它擴充套件了運算元以允許受管理的、容錯的狀態（請參閱“[失敗後重建狀態](#)”）。

具有良好定義的輸入和輸出的確定性函式的原理不僅有利於容錯（請參閱“[幕等性](#)”），也簡化了有關組織中資料流的推理【7】。無論衍生資料是搜尋索引、統計模型還是快取，採用這種觀點思考都是很有幫助的：將其視為從一個東西衍生出另一個的資料管道，透過函式式應用程式碼推送一個系統的狀態變更，並將其效果應用至衍生系統中。

原則上，衍生資料系統可以同步地維護，就像關係資料庫在與索引表寫入操作相同的事務中同步更新次級索引一樣。然而，非同步是使基於事件日誌的系統穩健的原因：它允許系統的一部分故障被抑制在本地。而如果任何一個參與者失敗，分散式事務將中止，因此它們傾向於透過將故障傳播到系統的其餘部分來放大故障（請參閱“[分散式事務的限制](#)”）。

我們在“[分割槽與次級索引](#)”中看到，次級索引經常跨越分割槽邊界。具有次級索引的分割槽系統需要將寫入傳送到多個分割槽（如果索引按關鍵詞分割槽的話）或將讀取傳送到所有分割槽（如果索引是按文件分割槽的話）。如果索引是非同步維護的，這種跨分割槽通訊也是最可靠和最可伸縮的【8】（另請參閱“[多分割槽資料處理](#)”）。

應用演化後重新處理資料

在維護衍生資料時，批處理和流處理都是有用的。流處理允許將輸入中的變化以低延遲反映在衍生檢視中，而批處理允許重新處理大量累積的歷史資料以便將新檢視匯出到現有資料集上。

特別是，重新處理現有資料為維護系統、演化並支援新功能和需求變更提供了一個良好的機制（請參閱[第四章](#)）。沒有重新進行處理，模式演化將僅限於簡單的變化，例如向記錄中新增新的可選欄位或新增新型別的記錄。無論是在寫時模式還是在讀時模式中都是如此（請參閱“[文件模型中的模式靈活性](#)”）。另一方面，透過重新處理，可以將資料集重組

為一個完全不同的模型，以便更好地滿足新的要求。

鐵路上的模式遷移

大規模的“模式遷移”也發生在非計算機系統中。例如，在 19 世紀英國鐵路建設初期，軌距（兩軌之間的距離）就有了各種各樣的競爭標準。為一種軌距而建的列車不能在另一種軌距的軌道上執行，這限制了火車網路中可能的相互連線【9】。

在 1846 年最終確定了一個標準軌距之後，其他軌距的軌道必須轉換——但是如何在不停運火車線路的情況下進行數月甚至數年的遷移？解決的辦法是首先透過新增第三條軌道將軌道轉換為 **雙軌距 (dual guage)** 或 **混合軌距**。這種轉換可以逐漸完成，當完成時，兩種軌距的列車都可以線上路上跑，使用三條軌道中的兩條。事實上，一旦所有的列車都轉換成標準軌距，那麼可以移除提供非標準軌距的軌道。

以這種方式“再加工”現有的軌道，讓新舊版本並存，可以在幾年的時間內逐漸改變軌距。然而，這是一項昂貴的事業，這就是今天非標準軌距仍然存在的原因。例如，舊金山灣區的 BART 系統使用了與美國大部分地區不同的軌距。

衍生檢視允許 **漸進演化 (gradual evolution)**。如果你想重新構建資料集，不需要執行突然切換式的遷移。取而代之的是，你可以將舊架構和新架構並排維護為相同基礎資料上的兩個獨立衍生檢視。然後可以開始將少量使用者轉移到新檢視，以測試其效能並發現任何錯誤，而大多數使用者仍然會被路由到舊檢視。你可以逐漸地增加訪問新檢視的使用者比例，最終可以刪除舊檢視【10】。

這種逐漸遷移的美妙之處在於，如果出現問題，每個階段的過程都很容易逆轉：你始終有一個可以回滾的可用系統。透過降低不可逆損害的風險，你能對繼續前進更有信心，從而更快地改善系統【11】。

Lambda 架構

如果批處理用於重新處理歷史資料，而流處理用於處理最近的更新，那麼如何將這兩者結合起來？Lambda 架構【12】是這方面的一個建議，引起了很多關注。

Lambda 架構的核心思想是透過將不可變事件附加到不斷增長的資料集來記錄傳入資料，這類似於事件溯源（請參閱“[事件溯源](#)”）。為了從這些事件中衍生出讀取最佳化的檢視，Lambda 架構建議並行執行兩個不同的系統：批處理系統（如 Hadoop MapReduce）和獨立的流處理系統（如 Storm）。

在 Lambda 方法中，流處理器消耗事件並快速生成對檢視的近似更新；批處理器稍後將使用同一組事件並生成衍生檢視的更正版本。這個設計背後的原因是批處理更簡單，因此不易出錯，而流處理器被認為是不太可靠和難以容錯的（請參閱“[容錯](#)”）。而且，流處理可以使用快速近似演算法，而批處理使用較慢的精確演算法。

Lambda 架構是一種有影響力的想法，它將資料系統的設計變得更好，尤其是透過推廣這樣的原則：在不可變事件流上建立衍生檢視，並在需要時重新處理事件。但是我也認為它有一些實際問題：

- 在批處理和流處理框架中維護相同的邏輯是很顯著的額外工作。雖然像 Summingbird【13】這樣的庫提供了一種可以在批處理和流處理的上下文中執行的計算抽象。除錯、調整和維護兩個不同系統的操作複雜性依然存在【14】。
- 由於流管道和批處理管道產生獨立的輸出，因此需要合併它們以響應使用者請求。如果計算是基於滾動視窗的簡單聚合，則合併相當容易，但如果檢視基於更複雜的操作（例如連線和會話化）而匯出，或者輸出不是時間序列，則會變得非常困難。
- 儘管有能力重新處理整個歷史資料集是很好的，但在大型資料集上這樣做經常會開銷巨大。因此，批處理流水線通常需要設定為處理增量批處理（例如，在每小時結束時處理一小時的資料），而不是重新處理所有內容。這引發了“[時間推理](#)”中討論的問題，例如處理滯留事件和處理跨批次邊界的視窗。增量化批處理計算會增加複雜性，使其更類似於流式傳輸層，這與保持批處理層儘可能簡單的目標背道而馳。

統一批處理和流處理

最近的工作使得 Lambda 架構的優點在沒有其缺點的情況下得以實現，允許批處理計算（重新處理歷史資料）和流計算（在事件到達時即處理）在同一個系統中實現【15】。

在一個系統中統一批處理和流處理需要以下功能，這些功能也正在越來越廣泛地被提供：

- 透過處理最近事件流的相同處理引擎來重播歷史事件的能力。例如，基於日誌的訊息代理可以重播訊息（請參閱“[重播舊訊息](#)”），某些流處理器可以從 HDFS 等分散式檔案系統讀取輸入。
- 對於流處理器來說，恰好一次語義——即確保輸出與未發生故障的輸出相同，即使事實上發生故障（請參閱“[容錯](#)”）。與批處理一樣，這需要丟棄任何失敗任務的部分輸出。
- 按事件時間進行視窗化的工具，而不是按處理時間進行視窗化，因為處理歷史事件時，處理時間毫無意義（請參閱“[時間推理](#)”）。例如，Apache Beam 提供了用於表達這種計算的 API，可以在 Apache Flink 或 Google Cloud Dataflow 使用。

分拆資料庫

在最抽象的層面上，資料庫，Hadoop 和作業系統都發揮相同的功能：它們儲存一些資料，並允許你處理和查詢這些資料【16】。資料庫將資料儲存為特定資料模型的記錄（表中的行、文件、圖中的頂點等），而作業系統的檔案系統則將資料儲存在檔案中——但其核心都是“資訊管理”系統【17】。正如我們在 [第十章](#) 中看到的，Hadoop 生態系統有點像 Unix 的分散式版本。

當然，有很多實際的差異。例如，許多檔案系統都不能很好地處理包含 1000 萬個小檔案的目錄，而包含 1000 萬個小記錄的資料庫完全是尋常而不起眼的。無論如何，作業系統和資料庫之間的相似之處和差異值得探討。

Unix 和關係資料庫以非常不同的哲學來處理資訊管理問題。Unix 認為它的目的是為程式設計師提供一種相當低層次的硬體的邏輯抽象，而關係資料庫則希望為應用程式設計師提供一種高層次的抽象，以隱藏磁碟上資料結構的複雜性、併發性、崩潰恢復等等。Unix 發展出的管道和檔案只是位元組序列，而資料庫則發展出了 SQL 和事務。

哪種方法更好？當然這取決於你想要的是什麼。Unix 是“簡單的”，因為它是對硬體資源相當薄的包裝；關係資料庫是“更簡單”的，因為一個簡短的宣告性查詢可以利用很多強大的基礎設施（查詢最佳化、索引、連線方法、併發控制、複製等），而不需要查詢的作者理解其實現細節。

這些哲學之間的矛盾已經持續了幾十年（Unix 和關係模型都出現在 70 年代初），仍然沒有解決。例如，我將 NoSQL 運動解釋為，希望將類 Unix 的低級別抽象方法應用於分散式 OLTP 資料儲存的領域。

在這一部分我將試圖調和這兩個哲學，希望我們能各取其美。

組合使用資料儲存技術

在本書的過程中，我們討論了資料庫提供的各種功能及其工作原理，其中包括：

- 次級索引，使你可以根據欄位的值有效地搜尋記錄（請參閱“[其他索引結構](#)”）
- 物化檢視，這是一種預計算的查詢結果快取（請參閱“[聚合：資料立方體和物化檢視](#)”）
- 複製日誌，保持其他節點上資料的副本最新（請參閱“[複製日誌的實現](#)”）
- 全文搜尋索引，允許在文字中進行關鍵字搜尋（請參閱“[全文搜尋和模糊索引](#)”），也內置於某些關係資料庫【1】

在 [第十章](#) 和 [第十一章](#) 中，出現了類似的主題。我們討論了如何構建全文搜尋索引（請參閱“[批處理工作流的輸出](#)”），瞭解瞭如何維護物化檢視（請參閱“[維護物化檢視](#)”）以及如何將變更從資料庫複製到衍生資料系統（請參閱“[變更資料捕獲](#)”）。

資料庫中內建的功能與人們用批處理和流處理器構建的衍生資料系統似乎有相似之處。

建立索引

想想當你執行 `CREATE INDEX` 在關係資料庫中建立一個新的索引時會發生什麼。資料庫必須掃描表的一致性快照，挑選出所有被索引的欄位值，對它們進行排序，然後寫出索引。然後它必須處理自一致快照以來所做的寫入操作（假設表在建立索引時未被鎖定，所以寫操作可能會繼續）。一旦完成，只要事務寫入表中，資料庫就必須繼續保持索引最新。

此過程非常類似於設定新的從庫副本（請參閱“[設定新從庫](#)”），也非常類似於流處理系統中的引導（bootstrap）變更資料捕獲（請參閱“[初始快照](#)”）。

無論何時執行 `CREATE INDEX`，資料庫都會重新處理現有資料集（如“[應用演化後重新處理資料](#)”中所述），並將該索引作為新檢視匯出到現有資料上。現有資料可能是狀態的快照，而不是所有發生變化的日誌，但兩者密切相關（請參閱“[狀態、流和不變性](#)”）。

一切的元資料庫

有鑑於此，我認為整個組織的資料流開始像一個巨大的資料庫【7】。每當批處理、流或 ETL 過程將資料從一個地方傳輸到另一個地方並組裝時，它表現地就像資料庫子系統一樣，使索引或物化檢視保持最新。

從這種角度來看，批處理和流處理器就像精心實現的觸發器、儲存過程和物化檢視維護例程。它們維護的衍生資料系統就像不同的索引型別。例如，關係資料庫可能支援 B 樹索引、雜湊索引、空間索引（請參閱“[多列索引](#)”）以及其他型別的索引。在新興的衍生資料系統架構中，不是將這些設施作為單個整合資料庫產品的功能實現，而是由各種不同的軟體提供，執行在不同的機器上，由不同的團隊管理。

這些發展在未來將會把我們帶到哪裡？如果我們從沒有適合所有訪問模式的單一資料模型或儲存格式的前提出發，我推測有兩種途徑可以將不同的儲存和處理工具組合成一個有凝聚力的系統：

聯合資料庫：統一讀取

可以為各種各樣的底層儲存引擎和處理方法提供一個統一的查詢介面——一種稱為 **聯合資料庫**（federated database）或 **多型儲存**（polystore）的方法【18,19】。例如，PostgreSQL 的 **外部資料包裝器**（foreign data wrapper）功能符合這種模式【20】。需要專用資料模型或查詢介面的應用程式仍然可以直接訪問底層儲存引擎，而想要組合來自不同位置的資料的使用者可以透過聯合介面輕鬆完成操作。

聯合查詢介面遵循著單一整合系統的關係型傳統，帶有高階查詢語言和優雅的語義，但實現起來非常複雜。

分拆資料庫：統一寫入

雖然聯合能解決跨多個不同系統的只讀查詢問題，但它並沒有很好的解決跨系統 同步 寫入的問題。我們說過，在單個數據庫中，建立一致的索引是一項內建功能。當我們構建多個儲存系統時，我們同樣需要確保所有資料變更都會在所有正確的位置結束，即使在出現故障時也是如此。想要更容易地將儲存系統可靠地插接在一起（例如，透過變更資料捕獲和事件日誌），就像將資料庫的索引維護功能以可以跨不同技術同步寫入的方式分開【7,21】。

分拆方法遵循 Unix 傳統的小型工具，它可以很好地完成一件事【22】，透過統一的低層級 API（管道）進行通訊，並且可以使用更高層級的語言進行組合（shell）【16】。

開展分拆工作

聯合和分拆是一個硬幣的兩面：用不同的元件構成可靠、可伸縮和可維護的系統。聯合只讀查詢需要將一個數據模型對映到另一個數據模型，這需要一些思考，但最終還是一個可解決的問題。而我認為同步寫入到幾個儲存系統是更困難的工程問題，所以我將重點關注它。

傳統的同步寫入方法需要跨異構儲存系統的分散式事務【18】，我認為這是錯誤的解決方案（請參閱“[衍生資料與分散式事務](#)”）。單個儲存或流處理系統內的事務是可行的，但是當資料跨越不同技術之間的邊界時，我認為具有幕等寫入的非同步事件日誌是一種更加健壯和實用的方法。

例如，分散式事務在某些流處理元件內部使用，以匹配 **恰好一次**（exactly-once）語義（請參閱“[原子提交再現](#)”），這可以很好地工作。然而，當事務需要涉及由不同人群編寫的系統時（例如，當資料從流處理元件寫入分散式鍵值儲存或搜尋索引時），缺乏標準化的事務協議會使整合更難。有幕等消費者的有序事件日誌（請參閱“[幕等性](#)”）是一種更簡單的抽象，因此在異構系統中實現更加可行【7】。

基於日誌的整合的一大優勢是各個元件之間的 **鬆散耦合 (loose coupling)**，這體現在兩個方面：

1. 在系統級別，非同步事件流使整個系統在個別元件的中斷或效能下降時更加穩健。如果消費者執行緩慢或失敗，那麼事件日誌可以緩衝訊息（請參閱“[磁碟空間使用](#)”），以便生產者和任何其他消費者可以繼續不受影響地執行。有問題的消費者可以在問題修復後趕上，因此不會錯過任何資料，並且包含故障。相比之下，分散式事務的同步互動往往會將本地故障升級為大規模故障（請參閱“[分散式事務的限制](#)”）。
2. 在人力方面，分拆資料系統允許不同的團隊獨立開發，改進和維護不同的軟體元件和服務。專業化使得每個團隊都可以專注於做好一件事，並與其他團隊的系統以明確的介面互動。事件日誌提供了一個足夠強大的介面，以捕獲相當強的一致性屬性（由於永續性和事件的順序），但也足夠普遍於幾乎任何型別的資料。

分拆系統vs整合系統

如果分拆確實成為未來的方式，它也不會取代目前形式的資料庫——它們仍然會像以往一樣被需要。為了維護流處理元件中的狀態，資料庫仍然是需要的，並且為批處理和流處理器的輸出提供查詢服務（請參閱“[批處理工作流的輸出](#)”與“[流處理](#)”）。專用查詢引擎對於特定的工作負載仍然非常重要：例如，MPP 資料倉庫中的查詢引擎針對探索性分析查詢進行了最佳化，並且能夠很好地處理這種型別的工作負載（請參閱“[Hadoop 與分散式資料庫的對比](#)”）。

執行幾種不同基礎設施的複雜性可能是一個問題：每種軟體都有一個學習曲線，配置問題和操作怪癖，因此部署儘可能少的移動部件是很有必要的。比起使用應用程式碼拼接多個工具而成的系統，單一整合軟體產品也可以在其設計應對的工作負載型別上實現更好、更可預測的效能【23】。正如在前言中所說的那樣，為了不需要的規模而構建系統是白費精力，而且可能會將你鎖死在一個不靈活的設計中。實際上，這是一種過早最佳化的形式。

分拆的目標不是要針對個別資料庫與特定工作負載的效能進行競爭；我們的目標是允許你結合多個不同的資料庫，以便在比單個軟體可能實現的更廣泛的工作負載範圍內實現更好的效能。這是關於廣度，而不是深度——與我們在“[Hadoop 與分散式資料庫的對比](#)”中討論的儲存和處理模型的多樣性一樣。

因此，如果有一項技術可以滿足你的所有需求，那麼最好使用該產品，而不是試圖用更低層級的元件重新實現它。只有當沒有單一軟體滿足你的所有需求時，才會出現拆分和聯合的優勢。

少了什麼？

用於組成資料系統的工具正在變得越來越好，但我認為還缺少一個主要的東西：我們還沒有與 Unix shell 類似的分拆資料庫等價物（即，一種宣告式的、簡單的、用於組裝儲存和處理系統的高階語言）。

例如，如果我們可以簡單地宣告 `mysql | elasticsearch`，類似於 Unix 管道【22】，成為 `CREATE INDEX` 的分拆等價物：它將讀取 MySQL 資料庫中的所有文件並將其索引到 Elasticsearch 叢集中。然後它會不斷捕獲對資料庫所做的所有變更，並自動將它們應用於搜尋索引，而無需編寫自定義應用程式碼。這種整合應當支援幾乎任何型別的儲存或索引系統。

同樣，能夠更容易地預先計算和更新快取將是一件好事。回想一下，物化檢視本質上是一個預先計算的快取，所以你可以透過為複雜查詢宣告指定物化檢視來建立快取，包括圖上的遞迴查詢（請參閱“[圖資料模型](#)”）和應用邏輯。在這方面有一些有趣的早期研究，如 [差分資料流 \(differential dataflow\)](#) 【24,25】，我希望這些想法能夠在生產系統中找到自己的方法。

圍繞資料流設計應用

使用應用程式碼組合專用儲存與處理系統來分拆資料庫的方法，也被稱為“**資料庫由內而外 (database inside-out)**”方法【26】，該名稱來源於我在 2014 年的一次會議演講標題【27】。然而稱它為“新架構”過於誇大，我僅將其看作是一種設計模式，一個討論的起點，我們只是簡單地給它起一個名字，以便我們能更好地討論它。

這些想法不是我的；它們是很多人的思想的融合，這些思想非常值得我們學習。尤其是，以 Oz【28】和 Juttle【29】為代表的資料流語言，以 Elm【30,31】為代表的函式式響應式程式設計（functional reactive programming, FRP），以 Bloom【32】為代表的邏輯程式語言。在這一語境中的術語 **分拆 (unbundling)** 是由 Jay Kreps 提出的【7】。

即使是 **電子表格** 也在資料流程式設計能力上甩開大多數主流程式語言幾條街【33】。在電子表格中，可以將公式放入一個單元格中（例如，對另一列中的單元格求和），並且只要公式的任何輸入發生變更，公式的結果都會自動重新計算。這正是我們在資料系統層次所需要的：當資料庫中的記錄發生變更時，我們希望自動更新該記錄的任何索引，並且自動重新整理依賴於記錄的任何快取檢視或聚合。你不必擔心這種重新整理如何發生的技術細節，但能夠簡單地相信它可以正常工作。

因此，我認為絕大多數資料系統仍然可以從 VisiCalc 在 1979 年已經具備的功能中學習【34】。與電子表格的不同之處在於，今天的資料系統需要具有容錯性，可伸縮性以及持久儲存資料。它們還需要能夠整合不同人群編寫的不同技術，並重用現有的庫和服務：期望使用某一種特定的語言、框架或工具來開發所有軟體是不切實際的。

在本節中，我將詳細介紹這些想法，並探討一些圍繞分拆資料庫和資料流的想法構建應用的方法。

應用程式碼作為衍生函式

當一個數據集衍生自另一個數據集時，它會經歷某種轉換函式。例如：

- 次級索引是由一種直白的轉換函式生成的衍生資料集：對於基礎表中的每行或每個文件，它挑選被索引的列或欄位中的值，並按這些值排序（假設使用 B 樹或 SSTable 索引，按鍵排序，如 [第三章](#) 所述）。
- 全文搜尋索引是透過應用各種自然語言處理函式而建立的，諸如語言檢測、分詞、詞幹或詞彙化、拼寫糾正和同義詞識別，然後構建用於高效查詢的資料結構（例如倒排索引）。
- 在機器學習系統中，我們可以將模型視作從訓練資料透過應用各種特徵提取、統計分析函式衍生的資料，當模型應用於新的輸入資料時，模型的輸出是從輸入和模型（因此間接地從訓練資料）中衍生的。
- 快取通常包含將以使用者介面（UI）顯示的形式的資料聚合。因此填充快取需要知道 UI 中引用的欄位；UI 中的變更可能需要更新快取填充方式的定義，並重建快取。

用於次級索引的衍生函式是如此常用的需求，以致於它作為核心功能被內建至許多資料庫中，你可以簡單地透過 `CREATE INDEX` 來呼叫它。對於全文索引，常見語言的基本語言特徵可能內建到資料庫中，但更複雜的特徵通常需要領域特定的調整。在機器學習中，特徵工程是眾所周知的特定於應用的特徵，通常需要包含很多關於使用者互動與應用部署的詳細知識【35】。

當建立衍生資料集的函式不是像建立次級索引那樣的標準搬磚函式時，需要自定義程式碼來處理特定於應用的東西。而這個自定義程式碼是讓許多資料庫掙扎的地方，雖然關係資料庫通常支援觸發器、儲存過程和使用者定義的函式，可以用它們來在資料庫中執行應用程式碼，但它們有點像資料庫設計裡的事後反思。（請參閱“[傳遞事件流](#)”）。

應用程式碼和狀態的分離

理論上，資料庫可以是任意應用程式碼的部署環境，就如同作業系統一樣。然而實踐中它們對這一目標適配的很差。它們不滿足現代應用開發的要求，例如依賴和軟體包管理、版本控制、滾動升級、可演化性、監控、指標、對網路服務的呼叫以及與外部系統的整合。

另一方面，Mesos、YARN、Docker、Kubernetes 等部署和叢集管理工具專為執行應用程式碼而設計。透過專注於做好一件事情，他們能夠做得比將資料庫作為其眾多功能之一執行使用者定義的功能要好得多。

我認為讓系統的某些部分專門用於持久資料儲存並讓其他部分專門執行應用程式程式碼是有意義的。這兩者可以在保持獨立的同時互動。

現在大多數 Web 應用程式都是作為無狀態服務部署的，其中任何使用者請求都可以路由到任何應用程式伺服器，並且伺服器在傳送響應後會忘記所有請求。這種部署方式很方便，因為可以隨意新增或刪除伺服器，但狀態必須到某個地方：通常是資料庫。趨勢是將無狀態應用程式邏輯與狀態管理（資料庫）分開：不將應用程式邏輯放入資料庫中，也不將持久狀態置於應用程式中【36】。正如函數語言程式設計社群喜歡開玩笑說的那樣，“我們相信 **教會（Church）** 與 **國家（state）** 的分離”【37】ⁱ

ⁱ 解釋笑話很少會讓人感覺更好，但我不想讓任何人感到被遺漏。在這裡，Church 指代的是數學家的阿隆佐·邱奇，他創立了 lambda 演算，這是計算的早期形式，是大多數函數語言程式設計語言的基礎。lambda 演算不具有可變狀態（即沒有變數可以被覆蓋），所以可以說可變狀態與 Church 的工作是分離的。 ↩

在這個典型的 Web 應用模型中，資料庫充當一種可以透過網路同步訪問的可變共享變數。應用程式可以讀取和更新變數，而資料庫負責維持它的永續性，提供一些諸如併發控制和容錯的功能。

但是，在大多數程式語言中，你無法訂閱可變變數中的變更——你只能定期讀取它。與電子表格不同，如果變數的值發生變化，變數的讀者不會收到通知（你可以在自己的程式碼中實現這樣的通知——這被稱為 **觀察者模式**——但大多數語言沒有將這種模式作為內建功能）。

資料庫繼承了這種可變資料的被動方法：如果你想知道資料庫的內容是否發生了變化，通常你唯一的選擇就是輪詢（即定期重複你的查詢）。訂閱變更只是剛剛開始出現的功能（請參閱“[變更流的 API 支援](#)”）。

資料流：應用程式碼與狀態變化的互動

從資料流的角度思考應用程式，意味著重新協調應用程式碼和狀態管理之間的關係。我們不再將資料庫視作被應用操縱的被動變數，取而代之的是更多地考慮狀態，狀態變更和處理它們的程式碼之間的相互作用與協同關係。應用程式碼透過在另一個地方觸發狀態變更來響應狀態變更。

我們在“[資料庫與流](#)”中看到了這一思路，我們討論了將資料庫的變更日誌視為一種我們可以訂閱的事件流。諸如 Actor 的訊息傳遞系統（請參閱“[訊息傳遞中的資料流](#)”）也具有響應事件的概念。早在 20 世紀 80 年代，元組空間（tuple space）模型就已經探索了表達分散式計算的方式：觀察狀態變更並作出反應的過程【38,39】。

如前所述，當觸發器由於資料變更而被觸發時，或次級索引更新以反映索引表中的變更時，資料庫內部也發生著類似的情況。分拆資料庫意味著將這個想法應用於在主資料庫之外，用於建立衍生資料集：快取、全文搜尋索引、機器學習或分析系統。我們可以為此使用流處理和訊息傳遞系統。

需要記住的重要一點是，維護衍生資料不同於執行非同步任務。傳統的訊息傳遞系統通常是為執行非同步任務設計的（請參閱“[日誌與傳統的訊息傳遞相比](#)”）：

- 在維護衍生資料時，狀態變更的順序通常很重要（如果多個檢視是從事件日誌衍生的，則需要按照相同的順序處理事件，以便它們之間保持一致）。如“[確認與重新傳遞](#)”中所述，許多訊息代理在重傳未確認訊息時沒有此屬性，雙寫也被排除在外（請參閱“[保持系統同步](#)”）。
- 容錯是衍生資料的關鍵：僅僅丟失單個訊息就會導致衍生資料集永遠與其資料來源失去同步。訊息傳遞和衍生狀態更新都必須可靠。例如，許多 Actor 系統預設在記憶體中維護 Actor 的狀態和訊息，所以如果執行 Actor 的機器崩潰，狀態和訊息就會丟失。

穩定的訊息排序和容錯訊息處理是相當嚴格的要求，但與分散式事務相比，它們開銷更小，執行更穩定。現代流處理元件可以提供這些排序和可靠性保證，並允許應用程式碼以流運算元的形式執行。

這些應用程式碼可以執行任意處理，包括資料庫內建衍生函式通常不提供的功能。就像透過管道連結的 Unix 工具一樣，流運算元可以圍繞著資料流構建大型系統。每個運算元接受狀態變更的流作為輸入，併產生其他狀態變化的流作為輸出。

流處理器和服務

當今流行的應用開發風格涉及將功能分解為一組透過同步網路請求（如 REST API）進行通訊的 **服務**（service，請參閱“[服務中的資料流：REST 與 RPC](#)”）。這種面向服務的架構優於單一龐大應用的優勢主要在於：通過鬆散耦合來提供組織上的可伸縮性：不同的團隊可以專職於不同的服務上，從而減少團隊之間的協調工作（因為服務可以獨立部署和更新）。

在資料流中組裝流運算元與微服務方法有很多相似之處【40】。但底層通訊機制有很大區別：資料流採用單向非同步訊息流，而不是同步的請求 / 響應式互動。

除了在“[訊息傳遞中的資料流](#)”中列出的優點（如更好的容錯性），資料流系統還能實現更好的效能。例如，假設客戶正在購買以一種貨幣定價，但以另一種貨幣支付的商品。為了執行貨幣換算，你需要知道當前的匯率。這個操作可以透過兩種方式實現【40,41】：

1. 在微服務方法中，處理購買的程式碼可能會查詢匯率服務或資料庫，以獲取特定貨幣的當前匯率。

2. 在資料流方法中，處理訂單的程式碼會提前訂閱匯率變更流，並在匯率發生變動時將當前匯率儲存在本地資料庫中。處理訂單時只需查詢本地資料庫即可。

第二種方法能將對另一服務的同步網路請求替換為對本地資料庫的查詢（可能在同一臺機器甚至同一個程序中）ⁱⁱ。資料流方法不僅更快，而且當其他服務失效時也更穩健。最快且最可靠的網路請求就是壓根沒有網路請求！我們現在不再使用 RPC，而是在購買事件和匯率更新事件之間建立流聯接（請參閱“[流表連線（流擴充）](#)”）。

ⁱⁱ. 在微服務方法中，你也可以透過在處理購買的服務中本地快取匯率來避免同步網路請求。但是為了保證快取的新鮮度，你需要定期輪詢匯率以獲取其更新，或訂閱變更流——這恰好是資料流方法中發生的事情。[←](#)

連線是時間相關的：如果購買事件在稍後的時間點被重新處理，匯率可能已經改變。如果要重建原始輸出，則需要獲取原始購買時的歷史匯率。無論是查詢服務還是訂閱匯率更新流，你都需要處理這種時間相關性（請參閱[“連線的時間依賴性”](#)）。

訂閱變更流，而不是在需要時查詢當前狀態，使我們更接近類似電子表格的計算模型：當某些資料發生變更時，依賴於此的所有衍生資料都可以快速更新。還有很多未解決的問題，例如關於時間相關連線等問題，但我認為圍繞資料流構建應用的想法是一個非常有希望的方向。

觀察衍生資料狀態

在抽象層面，上一節討論的資料流系統提供了建立衍生資料集（例如搜尋索引、物化檢視和預測模型）並使其保持更新的過程。我們將這個過程稱為寫路徑（**write path**）：只要某些資訊被寫入系統，它可能會經歷批處理與流處理的多個階段，而最終每個衍生資料集都會被更新，以適配寫入的資料。圖 12-1 顯示了一個更新搜尋索引的例子。

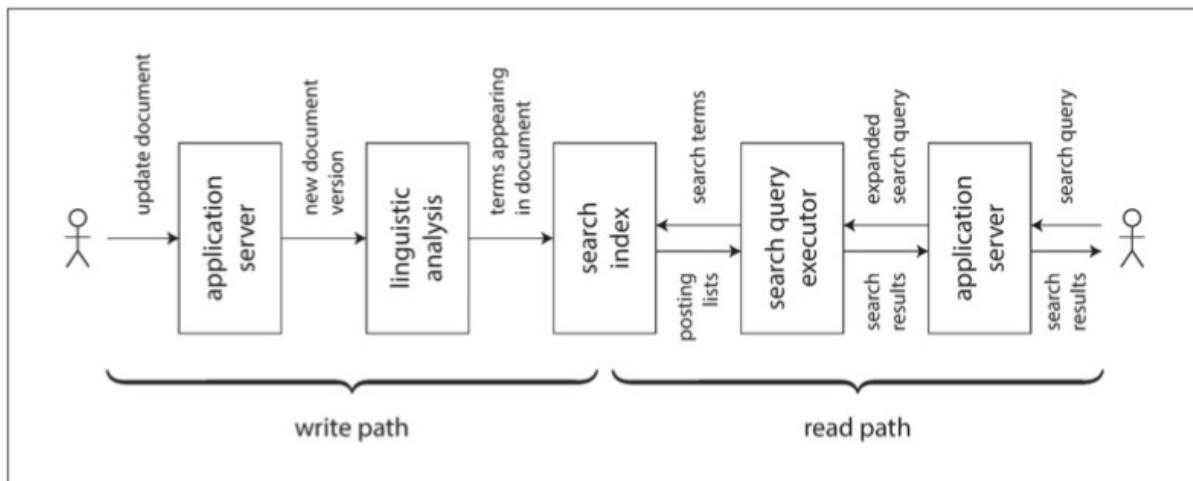


圖 12-1 在搜尋索引中，寫（文件更新）遇上讀（查詢）

但你為什麼一開始就要建立衍生資料集？很可能是因為你想在以後再次查詢它。這就是**讀路徑（read path）**：當服務使用者請求時，你需要從衍生資料集中讀取，也許還要對結果進行一些額外處理，然後構建給使用者的響應。

總而言之，寫路徑和讀路徑涵蓋了資料的整個旅程，從收集資料開始，到使用資料結束（可能是由另一個人）。寫路徑是預計算過程的一部分——即，一旦資料進入，即刻完成，無論是否有人需要看它。讀路徑是這個過程中只有當有人請求時才會發生的部分。如果你熟悉函數語言程式設計語言，則可能會注意到寫路徑類似於立即求值，讀路徑類似於惰性求值。

如圖 12-1 所示，衍生資料集是寫路徑和讀路徑相遇的地方。它代表了在寫入時需要完成的工作量與在讀取時需要完成的工作量之間的權衡。

物化檢視和快取

全文搜尋索引就是一個很好的例子：寫路徑更新索引，讀路徑在索引中搜索關鍵字。讀寫都需要做一些工作。寫入需要更新文件中出現的所有關鍵詞的索引條目。讀取需要搜尋查詢中的每個單詞，並應用布林邏輯來查詢包含查詢中所有單詞（AND 運算子）的文件，或者每個單詞（OR 運算子）的任何同義詞。

如果沒有索引，搜尋查詢將不得不掃描所有文件（如 grep），如果有著大量文件，這樣做的開銷巨大。沒有索引意味著寫入路徑上的工作量較少（沒有要更新的索引），但是在讀取路徑上需要更多工作。

另一方面，可以想象為所有可能的查詢預先計算搜尋結果。在這種情況下，讀路徑上的工作量會減少：不需要布林邏輯，只需查詢查詢結果並返回即可。但寫路徑會更加昂貴：可能的搜尋查詢集合是無限大的，因此預先計算所有可能的搜尋結果將需要無限的時間和儲存空間。那肯定沒戲ⁱⁱⁱ。

ⁱⁱⁱ. 假設一個有限的語料庫，那麼返回非空搜尋結果的搜尋查詢集合是有限的。然而，它是與語料庫中的術語數量呈指數關係，這仍是一個壞訊息。 ↪

另一種選擇是預先計算一組固定的最常見查詢的搜尋結果，以便可以快速提供它們而無需轉到索引。不常見的查詢仍然可以透過索引來提供服務。這通常被稱為常見查詢的 **快取（cache）**，儘管我們也可以稱之為 **物化檢視（materialized view）**，因為當新文件出現，且需要被包含在這些常見查詢的搜尋結果之中時，這些索引就需要更新。

從這個例子中我們可以看到，索引不是寫路徑和讀路徑之間唯一可能的邊界；快取常見搜尋結果也是可行的；而在少量文件上使用沒有索引的類 grep 掃描也是可行的。由此來看，快取，索引和物化檢視的作用很簡單：它們改變了讀路徑與寫路徑之間的邊界。透過預先計算結果，從而允許我們在寫路徑上做更多的工作，以節省讀路徑上的工作量。

在寫路徑上完成的工作和讀路徑之間的界限，實際上是本書開始處在“**描述負載**”中推特例子裡談到的主題。在該例中，我們還看到了與普通使用者相比，名人的寫路徑和讀路徑可能有所不同。在 500 頁之後，我們已經繞回了起點！

有狀態、可離線的客戶端

我發現寫路徑和讀路徑之間的邊界很有趣，因為我們可以試著改變這個邊界，並探討這種改變的實際意義。我們來看看不同上下文中的這一想法。

過去二十年來，Web 應用的火熱讓我們對應用開發作出了一些很容易視作理所當然的假設。具體來說就是，客戶端 / 同伺服器模型——客戶端大多是無狀態的，而伺服器擁有資料的權威——已經普遍到我們幾乎忘掉了還有其他任何模型的存在。但是技術在不斷地發展，我認為不時地質疑現狀非常重要。

傳統上，網路瀏覽器是無狀態的客戶端，只有當連線到網際網路時才能做一些有用的事情（能離線執行的唯一事情基本上就是上下滾動之前線上時載入好的頁面）。然而，最近的“單頁面”JavaScript Web 應用已經獲得了很多有狀態的功能，包括客戶端使用者介面互動，以及 Web 瀏覽器中的持久化本地儲存。移動應用可以類似地在裝置上儲存大量狀態，而且大多數使用者互動都不需要與伺服器往返互動。

這些不斷變化的功能重新引發了對 **離線優先（offline-first）** 應用的興趣，這些應用盡可能地在同一裝置上使用本地資料庫，無需連線網際網路，並在後臺網路連線可用時與遠端伺服器同步【42】。由於移動裝置通常具有緩慢且不可靠的蜂窩網路連線，因此，如果使用者的使用者介面不必等待同步網路請求，且應用主要是離線工作的，則這是一個巨大優勢（請參閱“**需要離線操作的客戶端**”）。

當我們擺脫無狀態客戶端與中央資料庫互動的假設，並轉向在終端使用者裝置上維護狀態時，這就開啟了新世界的大門。特別是，我們可以將裝置上的狀態視為 **伺服器狀態的快取**。螢幕上的畫素是客戶端應用中模型物件的物化檢視；模型物件是遠端資料中心的本地狀態副本【27】。

將狀態變更推送給客戶端

在典型的網頁中，如果你在 Web 瀏覽器中載入頁面，並且隨後伺服器上的資料發生變更，則瀏覽器在重新載入頁面之前對此一無所知。瀏覽器只能在一個時間點讀取資料，假設它是靜態的——它不會訂閱來自伺服器的更新。因此裝置上的狀態是陳舊的快取，除非你顯式輪詢變更否則不會更新。（像 RSS 這樣基於 HTTP 的 Feed 訂閱協議實際上只是一種基本的輪詢形式）

最近的協議已經超越了 HTTP 的基本請求 / 響應模式：服務端傳送的事件（EventSource API）和 WebSockets 提供了通訊通道，透過這些通道，Web 瀏覽器可以與伺服器保持開啟的 TCP 連線，只要瀏覽器仍然連線著，伺服器就能主動向瀏覽器推送資訊。這為伺服器提供了主動通知終端使用者客戶端的機會，伺服器能告知客戶端其本地儲存狀態的任何變化，從而減少客戶端狀態的陳舊程度。

用我們的寫路徑與讀路徑模型來講，主動將狀態變更推至到客戶端裝置，意味著將寫路徑一直延伸到終端使用者。當客戶端首次初始化時，它仍然需要使用讀路徑來獲取其初始狀態，但此後它就能夠依賴伺服器傳送的狀態變更流了。我們在流處理和訊息傳遞部分討論的想法並不侷限於資料中心中：我們可以進一步採納這些想法，並將它們一直延伸到終端使用者裝置【43】。

這些裝置有時會離線，並在此期間無法收到伺服器狀態變更的任何通知。但是我們已經解決了這個問題：在“[消費者偏移量](#)”中，我們討論了基於日誌的訊息代理的消費者能在失敗或斷開連線後重連，並確保它不會錯過掉線期間任何到達的訊息。同樣的技術適用於單個使用者，每個裝置都是一個小事件流的小小訂閱者。

端到端的事件流

最近用於開發有狀態的客戶端與使用者介面的工具，例如如 Elm 語言【30】和 Facebook 的 React、Flux 和 Redux 工具鏈，已經透過訂閱表示使用者輸入或伺服器響應的事件流來管理客戶端的內部狀態，其結構與事件溯源相似（請參閱“[事件溯源](#)”）。

將這種程式設計模型擴充套件為：允許伺服器將狀態變更事件推送到客戶端的事件管道中，是非常自然的。因此，狀態變化可以透過 **端到端（end-to-end）** 的寫路徑流動：從一個裝置上的互動觸發狀態變更開始，經由事件日誌，並穿過幾個衍生資料系統與流處理器，一直到另一臺裝置上的使用者介面，而有人正在觀察使用者介面上的狀態變化。這些狀態變化能以相當低的延遲傳播——比如說，在一秒內從一端到另一端。

一些應用（如即時訊息傳遞與線上遊戲）已經具有這種“實時”架構（在低延遲互動的意義上，不是在“[響應時間保證](#)”中的意義上）。但我們為什麼不用這種方式構建所有的應用？

挑戰在於，關於無狀態客戶端和請求 / 響應互動的假設已經根深蒂固地植入在我們的資料庫、庫、框架以及協議之中。許多資料儲存支援讀取與寫入操作，為請求返回一個響應，但只有極少數提供訂閱變更的能力——請求返回一個隨時間推移的響應流（請參閱“[變更流的 API 支援](#)”）。

為了將寫路徑延伸至終端使用者，我們需要從根本上重新思考我們構建這些系統的方式：從請求 / 響應互動轉向釋出 / 訂閱資料流【27】。更具響應性的使用者介面與更好的離線支援，我認為這些優勢值得我們付出努力。如果你正在設計資料系統，我希望你對訂閱變更的選項留有印象，而不只是查詢當前狀態。

讀也是事件

我們討論過，當流處理器將衍生資料寫入儲存（資料庫，快取或索引）時，以及當用戶請求查詢該儲存時，儲存將充當寫路徑和讀路徑之間的邊界。該儲存應當允許對資料進行隨機訪問的讀取查詢，否則這些查詢將需要掃描整個事件日誌。

在很多情況下，資料儲存與流處理系統是分開的。但回想一下，流處理器還是需要維護狀態以執行聚合和連線的（請參閱“[流連線](#)”）。這種狀態通常隱藏在流處理器內部，但一些框架也允許這些狀態被外部客戶端查詢【45】，將流處理器本身變成一種簡單的資料庫。

我願意進一步思考這個想法。正如到目前為止所討論的那樣，對儲存的寫入是透過事件日誌進行的，而讀取是臨時的網路請求，直接流向儲存著待查資料的節點。這是一個合理的設計，但不是唯一可行的設計。也可以將讀取請求表示為事件流，並同時將讀事件與寫事件送往流處理器；流處理器透過將讀取結果傳送到輸出流來響應讀取事件【46】。

當寫入和讀取都被表示為事件，並且被路由到同一個流運算元以便處理時，我們實際上是在讀取查詢流和資料庫之間執行流表連線。讀取事件需要被送往儲存資料的資料庫分割槽（請參閱“[請求路由](#)”），就像批處理和流處理器在連線時需要在同一個鍵上對輸入分割槽一樣（請參閱“[Reduce 側連線與分組](#)”）。

服務請求與執行連線之間的這種相似之處是非常關鍵的【47】。一次性讀取請求只是將請求傳遞到連線運算元，然後請求馬上就被忘掉了；而一個訂閱請求，則是與連線另一側過去與未來事件的持久化連線。

記錄讀取事件的日誌可能對於追蹤整個系統中的因果關係與資料來源也有好處：它可以讓你重現出當用戶做出特定決策之前看見了什麼。例如在網商中，向客戶顯示的預測送達日期與庫存狀態，可能會影響他們是否選擇購買一件商品【4】。要分析這種聯絡，則需要記錄使用者查詢運輸與庫存狀態的結果。

將讀取事件寫入持久儲存可以更好地跟蹤因果關係（請參閱“[排序事件以捕獲因果關係](#)”），但會產生額外的儲存與 I/O 成本。最佳化這些系統以減少開銷仍然是一個開放的研究問題【2】。但如果你已經出於運維目的留下了讀取請求日誌，將其作為請求處理的副作用，那麼將這份日誌作為請求事件源並不是什麼特別大的變更。

多分割槽資料處理

對於只涉及單個分割槽的查詢，透過流來發送查詢與收集響應可能是殺雞用牛刀了。然而，這個想法開啟了分散式執行複雜查詢的可能性，這需要合併來自多個分割槽的資料，利用了流處理器已經提供的訊息路由、分割槽和連線的基礎設施。

Storm 的分散式 RPC 功能支援這種使用模式（請參閱“[訊息傳遞和 RPC](#)”）。例如，它已經被用來計算瀏覽過某個推特 URL 的人數——即，發推包含該 URL 的所有人的粉絲集合的並集【48】。由於推特的使用者是分割槽的，因此這種計算需要合併來自多個分割槽的結果。

這種模式的另一個例子是欺詐預防：為了評估特定購買事件是否具有欺詐風險，你可以檢查該使用者 IP 地址，電子郵件地址，帳單地址，送貨地址的信用分。這些信用資料庫中的每一個都是有分割槽的，因此為特定購買事件採集分數需要連線一系列不同的分割槽資料集【49】。

MPP 資料庫的內部查詢執行圖有著類似的特徵（請參閱“[Hadoop 與分散式資料庫的對比](#)”）。如果需要執行這種多分割槽連線，則直接使用提供此功能的資料庫，可能要比使用流處理器實現它要更簡單。然而將查詢視為流提供了一種選項，可以用於實現超出傳統現成解決方案的大規模應用。

將事情做正確

對於只讀取資料的無狀態服務，出問題也沒什麼大不了的：你可以修復該錯誤並重啟服務，而一切都恢復正常。像資料庫這樣的有狀態系統就沒那麼簡單了：它們被設計為永遠記住事物（或多或少），所以如果出現問題，這種（錯誤的）效果也將潛在地永遠持續下去，這意味著它們需要更仔細的思考【50】。

我們希望構建可靠且 **正確** 的應用（即使面對各種故障，程式的語義也能被很好地定義與理解）。約四十年來，原子性、隔離性和永續性（第七章）等事務特性一直是構建正確應用的首選工具。然而這些地基沒有看上去那麼牢固：例如弱隔離級別帶來的困惑可以佐證（請參閱“[弱隔離級別](#)”）。

事務在某些領域被完全拋棄，並被提供更好效能與可伸縮性的模型取代，但後者有更複雜的語義（例如，請參閱“[無主複製](#)”）。**一致性 (Consistency)** 經常被談起，但其定義並不明確（請參閱“[一致性](#)”和 [第九章](#)）。有些人斷言我們應當為了高可用而“擁抱弱一致性”，但卻對這些概念實際上意味著什麼缺乏清晰的認識。

對於如此重要的話題，我們的理解，以及我們的工程方法卻是驚人地薄弱。例如，確定在特定事務隔離等級或複製配置下執行特定應用是否安全是非常困難的【51,52】。通常簡單的解決方案似乎在低併發性的情況下工作正常，並且沒有錯誤，但在要求更高的情況下卻會出現許多微妙的錯誤。

例如，Kyle Kingsbury 的 Jepsen 實驗【53】標出了一些產品聲稱的安全保證與其在網路問題與崩潰時的實際行為之間的明顯差異。即使像資料庫這樣的基礎設施產品沒有問題，應用程式碼仍然需要正確使用它們提供的功能才行，如果配置很難理解，這是很容易出錯的（在這種情況下指的是弱隔離級別，法定人數配置等）。

如果你的應用可以容忍偶爾的崩潰，以及以不可預料的方式損壞或丟失資料，那生活就要簡單得多，而你可能只要雙手合十念阿彌陀佛，期望佛祖能保佑最好的結果。另一方面，如果你需要更強的正確性保證，那麼可序列化與原子提交就是久經考驗的方法，但它們是有代價的：它們通常只在單個數據中心工作（這就排除了地理位置分散的架構），並限制了系統能夠實現的規模與容錯特性。

雖然傳統的事務方法並沒有走遠，但我也相信在使應用正確而靈活地處理錯誤方面上，事務也不是最後一個可以談的。在本節中，我將提出一些在資料流架構中考量正確性的方式。

資料庫的端到端原則

僅僅因為一個應用程式使用了具有相對較強安全屬性的資料系統（例如可序列化的事務），並不意味著就可以保證沒有資料丟失或損壞。例如，如果某個應用有個 Bug，導致它寫入不正確的資料，或者從資料庫中刪除資料，那麼可序列化的事務也救不了你。

這個例子可能看起來很無聊，但值得認真對待：應用會出 Bug，而人也會犯錯誤。我在“[狀態、流和不變性](#)”中使用了這個例子來支援不可變和僅追加的資料，閹割掉錯誤程式碼摧毀良好資料的能力，能讓從錯誤中恢復更為容易。

雖然不變性很有用，但它本身並非萬靈藥。讓我們來看一個可能發生的、非常微妙的資料損壞案例。

正好執行一次操作

在“[容錯](#)”中，我們見到了恰好一次（或等效一次）語義的概念。如果在處理訊息時出現問題，你可以選擇放棄（丟棄訊息——導致資料丟失）或重試。如果重試，就會有這種風險：第一次實際上成功了，只不過你沒有發現。結果這個訊息就被處理了兩次。

處理兩次是資料損壞的一種形式：為同樣的服務向客戶收費兩次（收費太多）或增長計數器兩次（誇大指標）都不是我們想要的。在這種情況下，恰好一次意味著安排計算，使得最終效果與沒有發生錯誤的情況一樣，即使操作實際上因為某種錯誤而重試。我們先前討論過實現這一目標的幾種方法。

最有效的方法之一是使操作幕等（idempotent，請參閱“[幕等性](#)”）：即確保它無論是執行一次還是執行多次都具有相同的效果。但是，將不是天生幕等的操作變為幕等的操作需要一些額外的努力與關注：你可能需要維護一些額外的元資料（例如更新了值的操作 ID 集合），並在從一個節點故障切換至另一個節點時做好防護（請參閱“[領導者和鎖](#)”）。

抑制重複

除了流處理之外，其他許多地方也需要抑制重複的模式。例如，TCP 使用了資料包上的序列號，以便接收方可以將它們正確排序，並確定網路上是否有資料包丟失或重複。在將資料交付應用前，TCP 協議棧會重新傳輸任何丟失的資料包，也會移除任何重複的資料包。

但是，這種重複抑制僅適用於單條 TCP 連線的場景中。假設 TCP 連線是一個客戶端與資料庫的連線，並且它正在執行例 12-1 中的事務。在許多資料庫中，事務是繫結在客戶端連線上的（如果客戶端傳送了多個查詢，資料庫就知道它們屬於同一個事務，因為它們是在同一個 TCP 連線上傳送的）。如果客戶端在傳送 COMMIT 之後並在從資料庫伺服器收到響應之前遇到網路中斷與連線超時，客戶端是不知道事務是否已經被提交的（圖 8-1）。

例 12-1 資金從一個賬戶到另一個賬戶的非幕等轉移

```
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance + 11.00 WHERE account_id = 1234;
UPDATE accounts SET balance = balance - 11.00 WHERE account_id = 4321;
COMMIT;
```

客戶端可以重連到資料庫並重試事務，但現在已經處於 TCP 重複抑制的範圍之外了。因為例 12-1 中的事務不是幕等的，可能會發生轉了 \$22 而不是期望的 \$11。因此，儘管例 12-1 是一個事務原子性的標準樣例，但它實際上並不正確，而真正的銀行並不會這樣辦事 [3]。

兩階段提交（請參閱“[原子提交與兩階段提交](#)”）協議會破壞 TCP 連線與事務之間的 1:1 對映，因為它們必須在故障後允許事務協調器重連到資料庫，告訴資料庫將存疑事務提交還是中止。這足以確保事務只被恰好執行一次嗎？不幸的是，並不能。

即使我們可以抑制資料庫客戶端與伺服器之間的重複事務，我們仍然需要擔心終端使用者裝置與應用伺服器之間的網路。例如，如果終端使用者的客戶端是 Web 瀏覽器，則它可能會使用 HTTP POST 請求向伺服器提交指令。也許使用者正處於一個訊號微弱的蜂窩資料網路連線中，它們成功地傳送了 POST，但卻在能夠從伺服器接收響應之前沒了訊號。

在這種情況下，可能會向用戶顯示錯誤訊息，而他們可能會手動重試。Web 瀏覽器警告說，“你確定要再次提交這個表單嗎？”——使用者選“是”，因為他們希望操作發生（Post/Redirect/Get 模式【54】可以避免在正常操作中出現此警告訊息，但 POST 請求超時就沒辦法了）。從 Web 伺服器的角度來看，重試是一個獨立的請求；從資料庫的角度來看，這是一個獨立的事務。通常的除重機制無濟於事。

操作識別符號

要在通過幾跳的網路通訊上使操作具有幂等性，僅僅依賴資料庫提供的事務機制是不夠的——你需要考慮 **端到端 (end-to-end)** 的請求流。例如，你可以為操作生成一個唯一的識別符號（例如 UUID），並將其作為隱藏表單欄位包含在客戶端應用中，或透過計算所有表單相關欄位的雜湊來生成操作 ID【3】。如果 Web 瀏覽器提交了兩次 POST 請求，這兩個請求將具有相同的操作 ID。然後，你可以將該操作 ID 一路傳遞到資料庫，並檢查你是否曾經使用給定的 ID 執行過一個操作，如 [例 12-2](#) 中所示。

例 12-2 使用唯一 ID 來抑制重複請求

```
ALTER TABLE requests ADD UNIQUE (request_id);

BEGIN TRANSACTION;
INSERT INTO requests
(request_id, from_account, to_account, amount)
VALUES('0286FDB8-D7E1-423F-B40B-792B3608036C', 4321, 1234, 11.00);
UPDATE accounts SET balance = balance + 11.00 WHERE account_id = 1234;
UPDATE accounts SET balance = balance - 11.00 WHERE account_id = 4321;
COMMIT;
```

[例 12-2](#) 依賴於 `request_id` 列上的唯一約束。如果一個事務嘗試插入一個已經存在的 ID，那麼 `INSERT` 失敗，事務被中止，使其無法生效兩次。即使在較弱的隔離級別下，關係資料庫也能正確地維護唯一性約束（而在“寫入偏差與幻讀”中討論過，應用級別的 **檢查 - 然後 - 插入** 可能會在不可序列化的隔離下失敗）。

除了抑制重複的請求之外，[例 12-2](#) 中的請求表表現得就像一種事件日誌，暗示著事件溯源的想法（請參閱“[事件溯源](#)”）。更新賬戶餘額事實上不必與插入事件發生在同一個事務中，因為它們是冗餘的，而能由下游消費者從請求事件中衍生出來——只要該事件被恰好處理一次，這又一次可以使用請求 ID 來強制執行。

端到端原則

抑制重複事務的這種情況只是一個更普遍的原則的一個例子，這個原則被稱為 **端到端原則 (end-to-end argument)**，它在 1984 年由 Saltzer、Reed 和 Clark 閣述【55】：

只有在通訊系統兩端應用的知識與幫助下，所討論的功能才能完全地正確地實現。因而將這種被質疑的功能作為通訊系統本身的功能是不可能的（有時，通訊系統可以提供這種功能的不完備版本，可能有助於提高效能）。

在我們的例子中 **所討論的功能** 是重複抑制。我們看到 TCP 在 TCP 連線層次抑制了重複的資料包，一些流處理器在訊息處理層次提供了所謂的恰好一次語義，但這些都無法阻止當一個請求超時時，使用者親自提交重複的請求。TCP，資料庫事務，以及流處理器本身並不能完全排除這些重複。解決這個問題需要一個端到端的解決方案：從終端使用者的客戶端一路傳遞到資料庫的事務識別符號。

端到端原則也適用於檢查資料的完整性：乙太網，TCP 和 TLS 中內建的校驗和可以檢測網路中資料包的損壞情況，但是它們無法檢測到由連線兩端傳送 / 接收軟體中 Bug 導致的損壞。或資料儲存所在磁碟上的損壞。如果你想捕獲資料所有可能的損壞來源，你也需要端到端的校驗和。

類似的原則也適用於加密【55】：家庭 WiFi 網路上的密碼可以防止人們竊聽你的 WiFi 流量，但無法阻止網際網路上其他地方攻擊者的窺探；客戶端與伺服器之間的 TLS/SSL 可以阻擋網路攻擊者，但無法阻止惡意伺服器。只有端到端的加密和認證可以防止所有這些事情。

儘管低層級的功能（TCP 重複抑制、乙太網校驗和、WiFi 加密）無法單獨提供所需的端到端功能，但它們仍然很有用，因為它們能降低較高層級出現問題的可能性。例如，如果我們沒有 TCP 來將資料包排成正確的順序，那麼 HTTP 請求通常就會被攬爛。我們只需要記住，低級別的可靠性功能本身並不足以確保端到端的正確性。

在資料系統中應用端到端思考

這將我帶回最初的論點：僅僅因為應用使用了提供相對較強安全屬性的資料系統，例如可序列化的事務，並不意味著應用的資料就不會丟失或損壞了。應用本身也需要採取端到端的措施，例如除重。

這實在是一個遺憾，因為容錯機制很難弄好。低層級的可靠機制（比如 TCP 中的那些）執行的相當好，因而剩下的高層級錯誤基本很少出現。如果能將這些剩下的高層級容錯機制打包成抽象，而應用不需要再去操心，那該多好呀——但恐怕我們還沒有找到這一正確的抽象。

長期以來，事務被認為是一個很好的抽象，我相信它們確實是很有用的。正如 [第七章](#) 導言中所討論的，它們將各種可能的問題（併發寫入、違背約束、崩潰、網路中斷、磁碟故障）合併為兩種可能結果：提交或中止。這是對程式設計模型而言是一種巨大的簡化，但恐怕這還不夠。

事務是代價高昂的，當涉及異構儲存技術時尤為甚（請參閱“[實踐中的分散式事務](#)”）。我們拒絕使用分散式事務是因為它開銷太大，結果我們最後不得不在應用程式碼中重新實現容錯機制。正如本書中大量的例子所示，對併發性與部分失敗的推理是困難且違反直覺的，所以我懷疑大多數應用級別的機制都不能正確工作，最終結果是資料丟失或損壞。

出於這些原因，我認為探索對容錯的抽象是很有價值的。它使提供應用特定的端到端的正確性屬性變得更簡單，而且還能在大規模分散式環境中提供良好的效能與運維特性。

強制約束

讓我們思考一下在 [分拆資料庫](#) 上下文中的 **正確性 (correctness)**。我們看到端到端的除重可以透過從客戶端一路透傳到資料庫的請求 ID 實現。那麼其他型別的約束呢？

我們先來特別關注一下 **唯一性約束**——例如我們在 [例 12-2](#) 中所依賴的約束。在“[約束和唯一性保證](#)”中，我們看到了幾個其他需要強制實施唯一性的應用功能例子：使用者名稱或電子郵件地址必須唯一標識使用者，檔案儲存服務不能包含多個重名檔案，兩個人不能在航班或劇院預訂同一個座位。

其他型別的約束也非常類似：例如，確保帳戶餘額永遠不會變為負數，確保不會超賣庫存，或者會議室沒有重複的預訂。執行唯一性約束的技術通常也可以用於這些約束。

唯一性約束需要達成共識

在 [第九章](#) 中我們看到，在分散式環境中，強制執行唯一性約束需要共識：如果存在多個具有相同值的併發請求，則系統需要決定衝突操作中的哪一個被接受，並拒絕其他違背約束的操作。

達成這共識的最常見方式是使單個節點作為領導，並使其負責所有決策。只要你不介意所有請求都擠過單個節點（即使客戶端位於世界的另一端），只要該節點沒有失效，系統就能正常工作。如果你需要容忍領導者失效，那麼就又回到了共識問題（請參閱“[單主複製與共識](#)”）。

唯一性檢查可以透過對唯一性欄位分割槽做橫向伸縮。例如，如果需要透過請求 ID 確保唯一性（如 [例 12-2](#) 所示），你可以確保所有具有相同請求 ID 的請求都被路由到同一分割槽（請參閱 [第六章](#)）。如果你需要讓使用者名稱是唯一的，則可以按使用者名稱的雜湊值做分割槽。

但非同步多主複製排除在外，因為可能會發生不同主庫同時接受衝突寫操作的情況，因而這些值不再是唯一的（請參閱“[實現線性一致的系統](#)”）。如果你想立刻拒絕任何違背約束的寫入，同步協調是無法避免的【56】。

基於日誌訊息傳遞中的唯一性

日誌確保所有消費者以相同的順序看見訊息——這種保證在形式上被稱為 **全序廣播 (total order broadcast)** 並且等價於共識（請參閱“[全序廣播](#)”）。在使用基於日誌的訊息傳遞的分拆資料庫方法中，我們可以使用非常類似的方法來執行唯一性約束。

流處理器在單個執行緒上依次消費單個日誌分割槽中的所有訊息（請參閱“[日誌與傳統的訊息傳遞相比](#)”）。因此，如果日誌是按需要確保唯一的值做的分割槽，則流處理器可以無歧義地、確定性地決定幾個衝突操作中的哪一個先到達。例如，在多個使用者嘗試宣告相同使用者名稱的情況下【57】：

1. 每個對使用者名稱的請求都被編碼為一條訊息，並追加到按使用者名稱雜湊值確定的分割槽。
2. 流處理器依序讀取日誌中的請求，並使用本地資料庫來追蹤哪些使用者名稱已經被佔用了。對於所有申請可用使用者名稱的請求，它都會記錄該使用者名稱，並向輸出流傳送一條成功訊息。對於所有申請已佔用使用者名稱的請求，它都會向輸出流傳送一條拒絕訊息。
3. 請求使用者名稱的客戶端監視輸出流，等待與其請求相對應的成功或拒絕訊息。

該演算法基本上與“[使用全序廣播實現線性一致的儲存](#)”中的演算法相同。它可以簡單地透過增加分割槽數伸縮至較大的請求吞吐量，因為每個分割槽都可以被獨立處理。

該方法不僅適用於唯一性約束，而且適用於許多其他型別的約束。其基本原理是，任何可能衝突的寫入都會路由到相同的分割槽並按順序處理。正如“[什麼是衝突？](#)”與“[寫入偏差與幻讀](#)”中所述，衝突的定義可能取決於應用，但流處理器可以使用任意邏輯來驗證請求。這個想法與 Bayou 在 90 年代開創的方法類似【58】。

多分割槽請求處理

當涉及多個分割槽時，確保操作以原子方式執行且同時滿足約束就變得很有趣了。在 [例 12-2](#) 中，可能有三個分割槽：一個包含請求 ID，一個包含收款人帳戶，另一個包含付款人帳戶。沒有理由把這三種東西放入同一個分割槽，因為它們都是相互獨立的。

在資料庫的傳統方法中，執行此事務需要跨全部三個分割槽進行原子提交，就這些分割槽上的所有其他事務而言，這實質上是將該事務嵌入一個全序。而這樣就要求跨分割槽協調，不同的分割槽無法再獨立地進行處理，因此吞吐量很可能會受到影響。

但事實證明，使用分割槽日誌可以達到等價的正確性而無需原子提交：

1. 從帳戶 A 向帳戶 B 轉賬的請求由客戶端提供一個唯一的請求 ID，並按請求 ID 追加寫入相應日誌分割槽。
2. 流處理器讀取請求日誌。對於每個請求訊息，它向輸出流發出兩條訊息：付款人帳戶 A 的借記指令（按 A 分割槽），收款人 B 的貸記指令（按 B 分割槽）。被發出的訊息中會帶有原始的請求 ID。
3. 後續處理器消費借記 / 貸記指令流，按照請求 ID 除重，並將變更應用至帳戶餘額。

步驟 1 和步驟 2 是必要的，因為如果客戶直接傳送貸記與借記指令，則需要在這兩個分割槽之間進行原子提交，以確保兩者要麼都發生或都不發生。為了避免對分散式事務的需要，我們首先將請求持久化記錄為單條訊息，然後從這第一條訊息中衍生出貸記指令與借記指令。幾乎在所有資料系統中，單物件寫入都是原子性的（請參閱“[單物件寫入](#)”），因此請求要麼出現在日誌中，要麼就不出現，無需多分割槽原子提交。

如果流處理器在步驟 2 中崩潰，則它會從上一個存檔點恢復處理。這樣做時，它不會跳過任何請求訊息，但可能會多次處理請求併產生重複的貸記與借記指令。但由於它是確定性的，因此它只是再次生成相同的指令，而步驟 3 中的處理器可以使用端到端請求 ID 輕鬆地對其除重。

如果你想確保付款人的帳戶不會因此次轉賬而透支，則可以使用一個額外的流處理器來維護帳戶餘額並校驗事務（按付款人帳戶分割槽），只有有效的事務會被記錄在步驟 1 中的請求日誌中。

透過將多分割槽事務分解為兩個不同分割槽方式的階段，並使用端到端的請求 ID，我們實現了同樣的正確性屬性（每個請求對付款人與收款人都恰好生效一次），即使在出現故障，且沒有使用原子提交協議的情況下依然如此。使用多個不同分割槽階段的想法與我們在“[多分割槽資料處理](#)”中討論的想法類似（也請參閱“[併發控制](#)”）。

及時性與完整性

事務的一個便利屬性是，它們通常是線性一致的（請參閱“線性一致性”），也就是說，寫入者會等到事務提交，而之後其寫入立刻對所有讀取者可見。

當我們把一個操作拆分為跨多個階段的流處理器時，卻並非如此：日誌的消費者在設計上就是非同步的，因此傳送者不會等其訊息被消費者處理完。但是，客戶端等待輸出流中的特定訊息是可能的。這正是我們在“[基於日誌訊息傳遞中的唯一性](#)”一節中檢查唯一性約束時所做的事情。

在這個例子中，唯一性檢查的正確性不取決於訊息傳送者是否等待結果。等待的目的僅僅是同步通知傳送者唯一性檢查是否成功。但該通知可以與訊息處理的結果相解耦。

更一般地來講，我認為術語 **一致性** (**consistency**) 這個術語混淆了兩個值得分別考慮的需求：

- 及時性 (Timeliness)

及時性意味著確保使用者觀察到系統的最新狀態。我們之前看到，如果使用者從陳舊的資料副本中讀取資料，它們可能會觀察到系統處於不一致的狀態（請參閱[“複製延遲問題”](#)）。但這種不一致是暫時的，而最終會透過等待與重試簡單地得到解決。

CAP 定理（請參閱[“線性一致性的代價”](#)）使用 **線性一致性** (**linearizability**) 意義上的一致性，這是實現及時性的強有力方法。像 [寫後讀](#) 這樣及時性更弱的一致性也很有用（請參閱[“讀已之寫”](#)）。

- 完整性 (Integrity)

完整性意味著沒有損壞；即沒有資料丟失，並且沒有矛盾或錯誤的資料。尤其是如果某些衍生資料集是作為底層資料之上的檢視而維護的（請參閱[“從事件日誌中派生出當前狀態”](#)），這種衍生必須是正確的。例如，資料庫索引必須正確地反映資料庫的內容——缺失某些記錄的索引並不是很有用。

如果完整性被違背，這種不一致是永久的：在大多數情況下，等待與重試並不能修復資料庫損壞。相反的是，需要顯式地檢查與修復。在 ACID 事務的上下文中（請參閱[“ACID 的含義”](#)），一致性通常被理解為某種特定於應用的完整性概念。原子性和永續性是保持完整性的重要工具。

口號形式：違反及時性，“最終一致性”；違反完整性，“永無一致性”。

我斷言在大多數應用中，完整性比及時性重要得多。違反及時性可能令人困惑與討厭，但違反完整性的結果可能是災難性的。

例如在你的信用卡對賬單上，如果某一筆過去 24 小時內完成的交易尚未出現並不令人奇怪——這些系統有一定的滯後是正常的。我們知道銀行是非同步核算與敲定交易的，這裡的及時性並不是非常重要【3】。但如果當期對賬單餘額與上期對賬單餘額加交易總額對不上（求和錯誤），或者出現一筆向你收費但未向商家付款的交易（消失的錢），那就實在是太糟糕了，這樣的問題就違背了系統的完整性。

資料流系統的正確性

ACID 事務通常既提供及時性（例如線性一致性）也提供完整性保證（例如原子提交）。因此如果你從 ACID 事務的角度來看待應用的正確性，那麼及時性與完整性的區別是無關緊要的。

另一方面，對於在本章中討論的基於事件的資料流系統而言，它們的一個有趣特性就是將及時性與完整性分開。在非同步處理事件流時不能保證及時性，除非你顯式構建一個在返回之前明確等待特定訊息到達的消費者。但完整性實際上才是流處理系統的核心。

恰好一次 或 **等效一次** 語義（請參閱[“容錯”](#)）是一種保持完整性的機制。如果事件丟失或者生效兩次，就有可能違背資料系統的完整性。因此在出現故障時，容錯訊息傳遞與重複抑制（例如，幕等操作）對於維護資料系統的完整性是很重要的。

正如我們在上一節看到的那樣，可靠的流處理系統可以在無需分散式事務與原子提交協議的情況下保持完整性，這意味著它們有潛力達到與後者相當的正確性，同時還具備好得多的效能與運維穩健性。為了達成這種正確性，我們組合使用了多種機制：

- 將寫入操作的內容表示為單條訊息，從而可以輕鬆地被原子寫入——與事件溯源搭配效果拔群（請參閱[“事件溯](#)

源”）。

- 使用與儲存過程類似的確定性衍生函式，從這一訊息中衍生出所有其他的狀態變更（請參閱“[真的序列執行](#)”和“[應用程式碼作為衍生函式](#)”）
- 將客戶端生成的請求 ID 傳遞透過所有的處理層次，從而允許端到端的除重，帶來幂等性。
- 使訊息不可變，並允許衍生資料能隨時被重新處理，這使從錯誤中恢復更加容易（請參閱“[不可變事件的優點](#)”）

這種機制組合在我看來，是未來構建容錯應用的一個非常有前景的方向。

寬鬆地解釋約束

如前所述，執行唯一性約束需要共識，通常透過在單個節點中彙集特定分割槽中的所有事件來實現。如果我們想要傳統的唯一性約束形式，這種限制是不可避免的，流處理也不例外。

然而另一個需要了解的事實是，許多真實世界的應用實際上可以擺脫這種形式，接受弱得多的唯一性：

- 如果兩個人同時註冊了相同的使用者名稱或預訂了相同的座位，你可以給其中一個人發訊息道歉，並要求他們換一個不同的使用者名稱或座位。這種糾正錯誤的變化被稱為 **補償性事務** (**compensating transaction**) 【59,60】。
- 如果客戶訂購的物品多於倉庫中的物品，你可以下單補倉，併為延誤向客戶道歉，向他們提供折扣。實際上，這麼說吧，如果叉車在倉庫中輒過了你的貨物，剩下的貨物比你想象的要少，那麼你也是得這麼做【61】。因此，既然道歉工作流無論如何已經成為你商業過程中的一部分了，那麼對庫存物品數目新增線性一致的約束可能就沒必要了。
- 與之類似，許多航空公司都會超賣機票，打著一些旅客可能會錯過航班的算盤；許多旅館也會超賣客房，抱著部分客人可能會取消預訂的期望。在這些情況下，出於商業原因而故意違反了“一人一座”的約束；當需求超過供給的情況出現時，就會進入補償流程（退款、升級艙位 / 房型、提供隔壁酒店的免費的房間）。即使沒有超賣，為了應對由惡劣天氣或員工罷工導致的航班取消，你還是需要道歉與補償流程——從這些問題中恢復僅僅是商業活動的正常組成部分。
- 如果有人從賬戶超額取款，銀行可以向他們收取透支費用，並要求他們償還欠款。透過限制每天的提款總額，銀行的風險是有限的。

在許多商業場景中，臨時違背約束並稍後透過道歉來修復，實際上是可以接受的。道歉的成本各不相同，但通常很低（以金錢或名聲來算）：你無法撤回已傳送的電子郵件，但可以傳送一封後續電子郵件進行更正。如果你不小心向信用卡收取了兩次費用，則可以將其中一項收費退款，而代價僅僅是手續費，也許還有客戶的投訴。儘管一旦 ATM 吐了錢，你無法直接收回，但原則上如果賬戶透支而客戶拒不支付，你可以派催收員收回欠款。

道歉的成本是否能接受是一個商業決策。如果可以接受的話，在寫入資料之前檢查所有約束的傳統模型反而會帶來不必要的限制，而線性一致性的約束也不是必須的。樂觀寫入，事後檢查可能是一種合理的選擇。你仍然可以在做一些挽回成本高昂的事情前確保有相關的驗證，但這並不意味著寫入資料之前必須先進行驗證。

這些應用確實需要完整性：你不會希望丟失預訂資訊，或者由於借方貸方不匹配導致資金消失。但是它們在執行約束時並不需要及時性：如果你銷售的貨物多於倉庫中的庫存，可以在事後道歉後並彌補問題。這種做法與我們在“[處理寫入衝突](#)”中討論的衝突解決方法類似。

無協調資料系統

我們現在已經做了兩個有趣的觀察：

1. 資料流系統可以維持衍生資料的完整性保證，而無需原子提交、線性一致性或者同步的跨分割槽協調。
2. 雖然嚴格的唯一性約束要求及時性和協調，但許多應用實際上可以接受寬鬆的約束：只要整個過程保持完整性，這些約束可能會被臨時違反並在稍後被修復。

總之這些觀察意味著，資料流系統可以為許多應用提供無需協調的資料管理服務，且仍能給出很強的完整性保證。這種**無協調** (**coordination-avoiding**) 的資料系統有著很大的吸引力：比起需要執行同步協調的系統，它們能達到更好的效能與更強的容錯能力【56】。

例如，這種系統可以使用多領導者配置運維，跨越多個數據中心，在區域間非同步複製。任何一個數據中心都可以持續獨立執行，因為不需要同步的跨區域協調。這樣的系統的及時性保證會很弱——如果不引入協調它是不可能是線性一致的——但它仍然可以提供有力的完整性保證。

在這種情況下，可序列化事務作為維護衍生狀態的一部分仍然是有用的，但它們只能在小範圍內執行，在那裡它們工作得很好【8】。異構分散式事務（如 XA 事務，請參閱“[實踐中的分散式事務](#)”）不是必需的。同步協調仍然可以在需要的地方引入（例如在無法恢復的操作之前強制執行嚴格的約束），但是如果只是應用的一小部分地方需要它，沒必要讓所有操作都付出協調的代價。【43】。

另一種審視協調與約束的角度是：它們減少了由於不一致而必須做出的道歉數量，但也可能會降低系統的效能和可用性，從而可能增加由於宕機中斷而需要做出的道歉數量。你不可能將道歉數量減少到零，但可以根據自己的需求尋找最佳平衡點——既不存在太多不一致性，又不存在太多可用性問題。

信任但驗證

我們所有關於正確性、完整性和容錯的討論都基於一些假設，假設某些事情可能會出錯，但其他事情不會。我們將這些假設稱為我們的 系統模型 (system model，請參閱[“將系統模型對映到現實世界”](#))：例如，我們應該假設程序可能會崩潰，機器可能突然斷電，網路可能會任意延遲或丟棄訊息。但是我們也可能假設寫入磁碟的資料在執行 `fsync` 後不會丟失，記憶體中的資料沒有損壞，而 CPU 的乘法指令總是能返回正確的結果。

這些假設是相當合理的，因為大多數時候它們都是成立的，如果我們不得不經常擔心計算機出錯，那麼基本上寸步難行。在傳統上，系統模型採用二元方法處理故障：我們假設有些事情可能會發生，而其他事情永遠不會發生。實際上，這更像是一個機率問題：有些事情更有可能，其他事情不太可能。問題在於違反我們假設的情況是否經常發生，以至於我們可能在實踐中遇到它們。

我們已經看到，資料可能會在尚未落盤時損壞（請參閱“[複製與永續性](#)”），而網路上的資料損壞有時可能規避了 TCP 校驗和（請參閱“[弱謊言形式](#)”）。也許我們應當更關注這些事情？

我過去所從事的一個應用收集了來自客戶端的崩潰報告，我們收到的一些報告，只有在這些裝置記憶體中出現了隨機位翻轉才解釋得通。這看起來不太可能，但是如果足夠多的裝置執行你的軟體，那麼即使再不可能發生的事也確實會發生。除了由於硬體故障或輻射導致的隨機儲存器損壞之外，一些病態的儲存器訪問模式甚至可以在沒有故障的儲存器中翻轉位【62】——一種可用於破壞作業系統安全機制的效應【63】（這種技術被稱為 **Rowhammer**）。一旦你仔細觀察，硬體並不是看上去那樣完美的抽象。

要澄清的是，隨機位翻轉在現代硬體上仍是非常罕見的【64】。我只想指出，它們並沒有超越可能性的範疇，所以值得一些關注。

維護完整性，儘管軟體有Bug

除了這些硬體問題之外，總是存在軟體 Bug 的風險，這些錯誤不會被較低層次的網路、記憶體或檔案系統校驗和所捕獲。即使廣泛使用的資料庫軟體也有 Bug：即使像 MySQL 與 PostgreSQL 這樣穩健、口碑良好、多年來被許多人充分測試過的軟體，就我個人所見也有 Bug，比如 MySQL 未能正確維護唯一約束【65】，以及 PostgreSQL 的可序列化隔離等級存在特定的寫入偏差異常【66】。對於不那麼成熟的軟體來說，情況可能要糟糕得多。

儘管在仔細設計，測試，以及審查上做出很多努力，但 Bug 仍然會在不知不覺中產生。儘管它們很少，而且最終會被發現並被修復，但總會有那麼一段時間，這些 Bug 可能會損壞資料。

而對於應用程式碼，我們不得不假設會有更多的錯誤，因為絕大多數應用的程式碼經受的評審與測試遠遠無法與資料庫的程式碼相比。許多應用甚至沒有正確使用資料庫提供的用於維持完整性的功能，例如外來鍵或唯一性約束【36】。

ACID 意義下的一致性（請參閱“[一致性](#)”）基於這樣一種想法：資料庫以一致的狀態啟動，而事務將其從一個一致狀態轉換至另一個一致的狀態。因此，我們期望資料庫始終處於一致狀態。然而，只有當你假設事務沒有 Bug 時，這種想法才有意義。如果應用以某種錯誤的方式使用資料庫，例如，不安全地使用弱隔離等級，資料庫的完整性就無法得到保證。

不要盲目信任承諾

由於硬體和軟體並不總是符合我們的理想，所以資料損壞似乎早晚不可避免。因此，我們至少應該有辦法查明資料是否已經損壞，以便我們能夠修復它，並嘗試追查錯誤的來源。檢查資料完整性稱為 **審計 (auditing)**。

如“[不可變事件的優點](#)”一節中所述，審計不僅僅適用於財務應用程式。不過，可審計性在財務中是非常非常重要的，因為每個人都知道錯誤總會發生，我們也都認為能夠檢測和解決問題是合理的需求。

成熟的系統同樣傾向於考慮不太可能的事情出錯的可能性，並管理這種風險。例如，HDFS 和 Amazon S3 等大規模儲存系統並不完全信任磁碟：它們執行後臺程序持續回讀檔案，並將其與其他副本進行比較，並將檔案從一個磁碟移動到另一個，以便降低靜默損壞的風險【67】。

如果你想確保你的資料仍然存在，你必須真正讀取它並進行檢查。大多數時候它們仍然會在那裡，但如果不是這樣，你一定想盡早知道答案，而不是更晚。按照同樣的原則，不時地嘗試從備份中恢復是非常重要的——否則當你發現備份損壞時，你可能已經遇到了資料丟失，那時候就真的太晚了。不要盲目地相信它們全都管用。

驗證的文化

像 HDFS 和 S3 這樣的系統仍然需要假設磁碟大部分時間都能正常工作——這是一個合理的假設，但與它們始終能正常工作的假設並不相同。然而目前還沒有多少系統採用這種“信任但是驗證”的方式來持續審計自己。許多人認為正確性保證是絕對的，並且沒有為罕見的資料損壞的可能性做過準備。我希望未來能看到更多的 **自我驗證 (self-validating)** 或 **自我審計 (self-auditing)** 系統，不斷檢查自己的完整性，而不是依賴盲目的信任【68】。

我擔心 ACID 資料庫的文化導致我們在盲目信任技術（如事務機制）的基礎上開發應用，而忽視了這種過程中的任何可審計性。由於我們所信任的技術在大多數情況下工作得很好，通常會認為審計機制並不值得投資。

但隨之而來的是，資料庫的格局發生了變化：在 NoSQL 的旗幟下，更弱的一致性保證成為常態，更不成熟的儲存技術越來越被廣泛使用。但是由於審計機制還沒有被開發出來，儘管這種方式越來越危險，我們仍不斷在盲目信任的基礎上構建應用。讓我們想一想如何針對可審計性而設計吧。

為可審計性而設計

如果一個事務在一個數據庫中改變了多個物件，在這一事實發生後，很難說清這個事務到底意味著什麼。即使你捕獲了事務日誌（請參閱[“變更資料捕獲”](#)），各種表中的插入、更新和刪除操作並不一定能清楚地表明為什麼要執行這些變更。決定這些變更的是應用邏輯中的呼叫，而這一應用邏輯稍縱即逝，無法重現。

相比之下，基於事件的系統可以提供更好的可審計性。在事件溯源方法中，系統的使用者輸入被表示為一個單一不可變事件，而任何其導致的狀態變更都衍生自該事件。衍生可以實現為具有確定性與可重複性，因而相同的事件日誌透過相同版本的衍生程式碼時，會導致相同的狀態變更。

顯式處理資料流（請參閱[“批處理輸出的哲學”](#)）可以使資料的來龍去脈（**provenance**）更加清晰，從而使完整性檢查更具可行性。對於事件日誌，我們可以使用雜湊來檢查事件儲存沒有被破壞。對於任何衍生狀態，我們可以重新執行從事件日誌中衍生它的批處理器與流處理器，以檢查是否獲得相同的結果，或者，甚至並行執行冗餘的衍生流程。

具有確定性且定義良好的資料流，也使除錯與跟蹤系統的執行變得容易，以便確定它為什麼做了某些事情【4,69】。如果出現意想之外的事情，那麼重現導致意外事件的確切事故現場的診斷能力——一種時間旅行除錯功能是非常有價值的。

端到端原則重現

如果我們不能完全相信系統的每個元件都不會損壞——每一個硬體都沒缺陷，每一個軟體都沒有 Bug——那我們至少必須定期檢查資料的完整性。如果我們不檢查，我們就不能發現損壞，直到無可挽回地導致對下游的破壞時，那時候再去追蹤問題就要難得多，且代價也要高的多。

檢查資料系統的完整性，最好是以端到端的方式進行（請參閱“[資料庫的端到端原則](#)”）：我們能在完整性檢查中涵蓋的系統越多，某些處理階層中出現不被察覺損壞的機率就越小。如果我們能檢查整個衍生資料管道端到端的正確性，那麼沿著這一路徑的任何磁碟、網路、服務以及演算法的正確性檢查都隱含在其中了。

持續的端到端完整性檢查可以不斷提高你對系統正確性的信心，從而使你能更快地進步【70】。與自動化測試一樣，審計提高了快速發現錯誤的可能性，從而降低了系統變更或新儲存技術可能導致損失的風險。如果你不害怕進行變更，就可以更好地充分演化一個應用，使其滿足不斷變化的需求。

用於可審計資料系統的工具

目前，將可審計性作為頂層關注點的資料系統並不多。一些應用實現了自己的審計機制，例如將所有變更記錄到單獨的審計表中，但是確保審計日誌與資料庫狀態的完整性仍然是很困難的。可以定期使用硬體安全模組對事務日誌進行簽名來防止篡改，但這無法保證正確的事務一開始就能進入到日誌中。

使用密碼學工具來證明系統的完整性是十分有趣的，這種方式對於寬泛的硬體與軟體問題，甚至是潛在的惡意行為都很穩健有效。加密貨幣、區塊鏈、以及諸如比特幣、以太坊、Ripple、Stellar 的分散式賬本技術已經迅速出現在這一領域【71,72,73】。

我沒有資格評論這些技術用於貨幣，或者合同商定機制的價值。但從資料系統的角度來看，它們包含了一些有趣的想法。實質上，它們是分散式資料庫，具有資料模型與事務機制，而不同副本可以由互不信任的組織託管。副本不斷檢查其他副本的完整性，並使用共識協議對應當執行的事務達成一致。

我對這些技術的拜占庭容錯方面有些懷疑（請參閱“[拜占庭故障](#)”），而且我發現 **工作證明 (proof of work)** 技術非常浪費（比如，比特幣挖礦）。比特幣的交易吞吐量相當低，儘管更多是出於政治與經濟原因而非技術上的原因。不過，完整性檢查的方面是很有意思的。

密碼學審計與完整性檢查通常依賴 **默克爾樹 (Merkle tree)** 【74】，這是一顆雜湊值的樹，能夠用於高效地證明一條記錄出現在一個數據集中（以及其他一些特性）。除了炒作的沸沸揚揚的加密貨幣之外，**證書透明性 (certificate transparency)** 也是一種依賴 Merkle 樹的安全技術，用來檢查 TLS/SSL 證書的有效性【75,76】。

我可以想象，那些在證書透明度與分散式賬本中使用的完整性檢查和審計方法，將會在通用資料系統中得到越來越廣泛的應用。要使得這些演算法對於沒有密碼學審計的系統同樣可伸縮，並儘可能降低效能損失還需要一些工作。但我認為這是一個值得關注的有趣領域。

做正確的事情

在本書的最後部分，我想退後一步。在本書中，我們考察了各種不同的資料系統架構，評價了它們的優點與缺點，並探討了構建可靠，可伸縮，可維護應用的技術。但是，我們忽略了討論中一個重要而基礎的部分，現在我想補充一下。

每個系統都服務於一個目的；我們採取的每個舉措都會同時產生期望的後果與意外的後果。這個目的可能只是簡單地賺錢，但其對世界的影響，可能會遠遠超出最初的目的。我們，建立這些系統的工程師，有責任去仔細考慮這些後果，並有意識地決定，我們希望生活在怎樣的世界中。

我們將資料當成一種抽象的東西來討論，但請記住，許多資料集都是關於人的：他們的行為，他們的興趣，他們的身份。對待這些資料，我們必須懷著人性與尊重。使用者也是人類，人類的尊嚴是至關重要的。

軟體開發越來越多地涉及重要的道德抉擇。有一些指導原則可以幫助軟體工程師解決這些問題，例如 ACM 的軟體工程道德規範與專業實踐【77】，但實踐中很少會討論這些，更不用說應用與強制執行了。因此，工程師和產品經理有時會對隱私與產品潛在的負面後果抱有非常傲慢的態度【78,79,80】。

技術本身並無好壞之分——關鍵在於它被如何使用，以及它如何影響人們。這對槍械這樣的武器是成立的，而搜尋引擎這樣的軟體系統與之類似。我認為，軟體工程師僅僅專注於技術而忽視其後果是不夠的：道德責任也是我們的責任。對道德推理很困難，但它太重要了，我們無法忽視。

預測性分析

舉個例子，預測性分析是“大資料”炒作的主要內容之一。使用資料分析預測天氣或疾病傳播是一碼事【81】；而預測一個罪犯是否可能再犯，一個貸款申請人是否有可能違約，或者一個保險客戶是否可能進行昂貴的索賠，則是另外一碼事。後者會直接影響到個人的生活。

當然，支付網路希望防止欺詐交易，銀行希望避免不良貸款，航空公司希望避免劫機，公司希望避免僱傭效率低下或不值得信任的人。從它們的角度來看，失去商機的成本很低，而不良貸款或問題員工的成本則要高得多，因而組織希望保持謹慎也是自然而然的事情。所以如果存疑，它們通常會 Say No。

然而，隨著演算法決策變得越來越普遍，被某種演算法（準確地或錯誤地）標記為有風險的某人可能會遭受大量這種“No”的決定。系統性地被排除在工作、航旅、保險、租賃、金融服務，以及其他社會關鍵領域之外。這是一種對個體自由的極大約束，因此被稱為“演算法監獄”【82】。在尊重人權的國家，刑事司法系統會做無罪推定（預設清白，直到被證明有罪）。另一方面，自動化系統可以系統地，任意地將一個人排除在社會參與之外，不需要任何有罪的證明，而且幾乎沒有申訴的機會。

偏見與歧視

演算法做出的決定不一定比人類更好或更差。每個人都可能有偏見，即使他們主動抗拒這一點；而歧視性做法也可能已經在文化上被制度化了。人們希望根據資料做出決定，而不是透過人的主觀評價與直覺，希望這樣能更加公平，並給予傳統體制中經常被忽視的人更好的機會【83】。

當我們開發預測性分析系統時，不是僅僅用軟體透過一系列 IF ELSE 規則將人類的決策過程自動化，那些規則本身甚至都是從資料中推斷出來的。但這些系統學到的模式是個黑盒：即使資料中存在一些相關性，我們可能也壓根不知道為什麼。如果演算法的輸入中存在系統性的偏見，則系統很有可能會在輸出中學習並放大這種偏見【84】。

在許多國家，反歧視法律禁止按種族、年齡、性別、性取向、殘疾或信仰等受保護的特徵區分對待不同的人。其他的個人特徵可能是允許用於分析的，但是如果這些特徵與受保護的特徵存在關聯，又會發生什麼？例如在種族隔離地區中，一個人的郵政編碼，甚至是他們的 IP 地址，都是很強的種族指示物。這樣的話，相信一種演算法可以以某種方式將有偏見的資料作為輸入，併產生公平和公正的輸出【85】似乎是很荒謬的。然而這種觀點似乎常常潛伏在資料驅動型決策的支持者中，這種態度被諷刺為“在處理偏差上，機器學習與洗錢類似”(machine learning is like money laundering for bias) 【86】。

預測性分析系統只是基於過去進行推斷；如果過去是歧視性的，它們就會將這種歧視歸納為規律。如果我們希望未來比過去更好，那麼就需要道德想象力，而這是隻有人類才能提供的東西【87】。資料與模型應該是我們的工具，而不是我們的主人。

責任與問責

自動決策引發了關於責任與問責的問題【87】。如果一個人犯了錯誤，他可以被追責，受決定影響的人可以申訴。演算法也會犯錯誤，但是如果它們出錯，誰來負責【88】？當一輛自動駕駛汽車引發事故時，誰來負責？如果自動信用評分算法系統性地歧視特定種族或宗教的人，這些人是否有任何追索權？如果機器學習系統的決定要受到司法審查，你能向法官解釋演算法是如何做出決定的嗎？

收集關於人的資料並進行決策，信用評級機構是一個很經典的例子。不良的信用評分會使生活變得更艱難，但至少信用分通常是基於個人 實際的 借款歷史記錄，而記錄中的任何錯誤都能被糾正（儘管機構通常會設定門檻）。然而，基於機器學習的評分演算法通常會使用更寬泛的輸入，並且更不透明；因而很難理解特定決策是怎樣作出的，以及是否有人被不公正地，歧視性地對待【89】。

信用分總結了“你過去的表現如何？”，而預測性分析通常是基於“誰與你類似，以及與你類似的人過去表現的如何？”。與他人的行為畫上等號意味著刻板印象，例如，根據他們居住的地方（與種族和階級關係密切的特徵）。那麼那些放錯位置的人怎麼辦？而且，如果是因為錯誤資料導致的錯誤決定，追索幾乎是不可能的【87】。

很多資料本質上是統計性的，這意味著即使機率分佈在總體上是正確的，對於個例也可能是錯誤的。例如，如果貴國的平均壽命是 80 歲，這並不意味著你在 80 歲生日時就會死掉。很難從平均值與機率分佈中對某個特定個體的壽命作出什麼判斷，同樣，預測系統的輸出是機率性的，對於個例可能是錯誤的。

盲目相信資料決策至高無上，這不僅僅是一種妄想，而是有切實危險的。隨著資料驅動的決策變得越來越普遍，我們需要弄清楚，如何使演算法更負責任且更加透明，如何避免加強現有的偏見，以及如何在它們不可避免地出錯時加以修復。

我們還需要想清楚，如何避免資料被用於害人，如何認識資料的積極潛力。例如，分析可以揭示人們生活的財務特點與社會特點。一方面，這種權力可以用來將援助與支援集中在幫助那些最需要援助的人身上。另一方面，它有時會被掠奪性企業用於識別弱勢群體，並向其兜售高風險產品，比如高利貸和沒有價值的大學文憑【87,90】。

反饋迴圈

即使是那些對人直接影響比較小的預測性應用，比如推薦系統，也有一些必須正視的難題。當服務變得善於預測使用者想要看到什麼內容時，它最終可能只會向人們展示他們已經同意的觀點，將人們帶入滋生刻板印象，誤導資訊，與極端思想的 **迴音室**。我們已經看到過社交媒體迴音室對競選的影響了【91】。

當預測性分析影響人們的生活時，自我強化的反饋迴圈會導致非常有害的問題。例如，考慮僱主使用信用分來評估候選人的例子。你可能是一個信用分不錯的好員工，但因不可抗力的意外而陷入財務困境。由於不能按期付帳單，你的信用分會受到影響，進而導致找到工作更為困難。失業使你陷入貧困，這進一步惡化了你的分數，使你更難找到工作【87】。在資料與數學嚴謹性的偽裝背後，隱藏的是由惡毒假設導致的惡性迴圈。

我們無法預測這種反饋迴圈何時發生。然而透過對整個系統（不僅僅是計算機化的部分，而且還有與之互動的人）進行整體思考，許多後果是可以夠預測的——一種稱為 **系統思維 (systems thinking)** 的方法【92】。我們可以嘗試理解資料分析系統如何響應不同的行為，結構或特性。該系統是否加強和增大了人們之間現有的差異（例如，損不足以奉有餘，富者愈富，貧者愈貧），還是試圖與不公作鬥爭？而且即使有著最好的動機，我們也必須當心意想不到的後果。

隱私和追蹤

除了預測性分析——使用資料來做出關於人的自動決策——資料收集本身也存在道德問題。收集資料的組織，與被收集資料的人之間，到底屬於什麼關係？

當系統只儲存使用者明確輸入的資料時，是因為使用者希望系統以特定方式儲存和處理這些資料，**系統是在為使用者提供服務**：使用者就是客戶。但是，當用戶的活動被跟蹤並記錄，作為他們正在做的其他事情的副作用時，這種關係就沒有那麼清晰了。該服務不再僅僅完成使用者想要它要做的事情，而是服務於它自己的利益，而這可能與使用者的利益相衝突。

追蹤使用者行為資料對於許多面向用戶的線上服務而言，變得越來越重要：追蹤使用者點選了哪些搜尋結果有助於改善搜尋結果的排名；推薦“喜歡 X 的人也喜歡 Y”，可以幫助使用者發現實用有趣的東西；A/B 測試和使用者流量分析有助於改善使用者介面。這些功能需要一定量的使用者行為跟蹤，而使用者也可以從中受益。

但不同公司有著不同的商業模式，追蹤並未止步於此。如果服務是透過廣告盈利的，那麼廣告主才是真正的客戶，而使用者的利益則屈居其次。跟蹤的資料會變得更詳細，分析變得更深入，資料會保留很長時間，以便為每個人建立詳細畫像，用於營銷。

現在，公司與被收集資料的使用者之間的關係，看上去就不太一樣了。公司會免費服務使用者，並引誘使用者儘可能多地使用服務。對使用者的追蹤，主要不是服務於該使用者個體，而是服務於掏錢資助該服務的廣告商。我認為這種關係可以用一個更具罪犯內涵的詞來恰當地描述：**監視 (surveillance)**。

監視

讓我們做一個思想實驗，嘗試用 **監視 (surveillance)** 一詞替換 **資料 (data)**，再看看常見的短語是不是聽起來還那麼漂亮【93】。比如：“在我們的監視驅動的組織中，我們收集實時監視流並將它們儲存在我們的監視倉庫中。我們的監視科學家使用高階分析和監視處理來獲得新的見解。”

對於本書《設計監控密集型應用》而言，這個思想實驗是罕見的爭議性內容，但我認為需要激烈的言辭來強調這一點。在我們嘗試製造軟體“吞噬世界”的過程中【94】，我們已經建立了世界上迄今為止所見過的最偉大的大規模監視基礎設施。我們正朝著萬物互聯邁進，我們正在迅速走近這樣一個世界：每個有人居住的空間至少包含一個帶網際網路連線

的麥克風，以智慧手機、智慧電視、語音控制助理裝置、嬰兒監視器甚至兒童玩具的形式存在，並使用基於雲的語音識別。這些裝置中的很多都有著可怕的安全記錄【95】。

即使是最為極權與專制的政權，可能也只會想著在每個房間裝一個麥克風，並強迫每個人始終攜帶能夠追蹤其位置與動向的裝置。然而，我們顯然是自願地，甚至熱情地投身於這個全域監視的世界。不同之處在於，資料是由公司，而不是由政府機構收集的【96】。

並不是所有的資料收集都稱得上監視，但檢視這一點有助於理解我們與資料收集者之間的關係。為什麼我們似乎很樂意接受企業的監視呢？也許你覺得自己沒有什麼好隱瞞的——換句話說，你與當權階級穿一條褲子，你不是被邊緣化的少數派，也不必害怕受到迫害【97】。不是每個人都如此幸運。或者，也許這是因為目的似乎是溫和的——這不是公然脅迫，也不是強制性的，而只是更好的推薦與更個性化的營銷。但是，結合上一節中對預測性分析的討論，這種區別似乎並不是很清晰。

我們已經看到與汽車追蹤裝置掛鈎的汽車保險費，以及取決於需要人佩戴健身追蹤裝置來確定的健康保險範圍。當監視被用於決定生活的重要方面時，例如保險或就業，它就開始變得不那麼溫和了。此外，資料分析可以揭示出令人驚訝的私密事物：例如，智慧手錶或健身追蹤器中的運動感測器能以相當好的精度計算出你正在輸入的內容（比如密碼）【98】。而分析演算法只會變得越來越精確。

同意與選擇的自由

我們可能會斷言使用者是自願選擇使用了會跟蹤其活動的服務，而且他們已經同意了服務條款與隱私政策，因此他們同意資料收集。我們甚至可以聲稱，使用者在用所提供的資料來換取有價值的服務，並且為了提供服務，追蹤是必要的。毫無疑問，社交網路、搜尋引擎以及各種其他免費的線上服務對於使用者來說都是有價值的，但是這個說法卻存在問題。

使用者幾乎不知道他們提供給我們的是什麼資料，哪些資料被放進了資料庫，資料又是怎樣被保留與處理的——大多數隱私政策都是模稜兩可的，忽悠使用者而不敢開啟天窗說亮話。如果使用者不瞭解他們的資料會發生什麼，就無法給出任何有意義的同意。有時來自一個使用者的資料還會提到一些關於其他人的事，而其他那些人既不是該服務的使用者，也沒有同意任何條款。我們在本書這一部分中討論的衍生資料集——來自整個使用者群的資料，加上行為追蹤與外部資料來源——就恰好是使用者無法（在真正意義上）理解的資料型別。

而且從使用者身上挖掘資料是一個單向過程，而不是真正的互惠關係，也不是公平的價值交換。使用者對能用多少資料換來什麼樣的服務，既沒有發言權也沒有選擇權：服務與使用者之間的關係是非常不對稱與單邊的。這些條款是由服務提出的，而不是由使用者提出的【99】。

對於不同意監視的使用者，唯一真正管用的備選項，就是簡單地不使用服務。但這個選擇也不是真正自由的：如果一項服務如此受歡迎，以至於“被大多數人認為是基本社會參與的必要條件”【99】，那麼指望人們選擇退出這項服務是不合理的——使用它事實上（*de facto*）是強制性的。例如，在大多數西方社會群體中，攜帶智慧手機，使用 Facebook 進行社交，以及使用 Google 查詢資訊已成為常態。特別是當一項服務具有網路效應時，人們選擇不使用會產生社會成本。

因為一個服務會跟蹤使用者而拒絕使用它，這只是少數人才擁有的權力，他們有足夠的時間與知識來了解隱私政策，並承受得起代價：錯過社會參與，以及使用服務可能帶來的專業機會。對於那些處境不太好的人而言，並沒有真正意義上的選擇：監控是不可避免的。

隱私與資料使用

有時候，人們聲稱“隱私已死”，理由是有些使用者願意把各種關於他們生活的事情釋出到社交媒體上，有時是平凡俗套，但有時是高度私密的。但這種說法是錯誤的，而且是對隱私（privacy）一詞的誤解。

擁有隱私並不意味著保密一切東西；它意味著擁有選擇向誰展示哪些東西的自由，要公開什麼，以及要保密什麼。隱私權是一項決定權：在從保密到透明的光譜上，隱私使得每個人都能決定自己想要在什麼地方位於光譜上的哪個位置【99】。這是一個人自由與自主的重要方面。

當透過監控基礎設施從人身上提取資料時，隱私權不一定受到損害，而是轉移到了資料收集者手中。獲取資料的公司實際上是說“相信我們會用你的資料做正確的事情”，這意味著，決定要透露什麼和保密什麼的權利從個體手中轉移到了公司手中。

這些公司反過來選擇保密這些監視結果，因為揭露這些會令人毛骨悚然，並損害它們的商業模式（比其他公司更瞭解人）。使用者的私密資訊只會間接地披露，例如針對特定人群定向投放廣告的工具（比如那些患有特定疾病的人群）。

即使特定使用者無法從特定廣告定向的人群中以個體的形式區分出來，但他們已經失去了披露一些私密資訊的能動性，例如他們是否患有某種疾病。決定向誰透露什麼並不是由個體按照自己的喜好決定的，而是由公司，以利潤最大化為目標來行使隱私權的。

許多公司都有一個目標，不要讓人感覺到毛骨悚然——先不說它們收集資料實際上是多麼具有侵犯性，讓我們先關注對使用者感受的管理。這些使用者感受經常被管理得很糟糕：例如，在事實上可能正確的一些東西，如果會觸發痛苦的回憶，使用者可能並不希望被提醒【100】。對於任何型別的資料，我們都應當考慮它出錯、不可取、不合時宜的可能性，並且需要建立處理這些失效的機制。無論是“不可取”還是“不合時宜”，當然都是由人的判斷決定的；除非我們明確地將演算法編碼設計為尊重人類的需求，否則演算法會無視這些概念。作為這些系統的工程師，我們必須保持謙卑，充分規劃，接受這些失效。

允許線上服務的使用者控制其隱私設定，例如控制其他使用者可以看到哪些東西，是將一些控制交還給使用者的第一步。但無論怎麼設定，服務本身仍然可以不受限制地訪問資料，並能以隱私策略允許的任何方式自由使用它。即使服務承諾不會將資料出售給第三方，它通常會授予自己不受限制的權利，以便在內部處理與分析資料，而且往往比使用者公開可見的部分要深入的多。

這種從個體到公司的大規模隱私權轉移在歷史上是史無前例的【99】。監控一直存在，但它過去是昂貴的、手動的，不是可伸縮的、自動化的。信任關係一直存在，例如患者與其醫生之間，或被告與其律師之間——但在這些情況下，資料的使用嚴格受到道德，法律和監管限制的約束。網際網路服務使得在未經有意義的同意下收集大量敏感資訊變得容易得多，而且無需使用者理解他們的個人資料到底發生了什麼。

資料資產與權力

由於行為資料是使用者與服務互動的副產品，因此有時被稱為“資料廢氣”——暗示資料是毫無價值的廢料。從這個角度來看，行為和預測性分析可以被看作是一種從資料中提取價值的回收形式，否則這些資料就會被浪費。

更準確的看法恰恰相反：從經濟的角度來看，如果定向廣告是服務的金主，那麼關於人的行為資料就是服務的核心資產。在這種情況下，使用者與之互動的應用僅僅是一種誘騙使用者將更多的個人資訊提供給監控基礎設施的手段【99】。線上服務中經常表現出的令人愉悅的人類創造力與社會關係，十分諷刺地被資料提取機器所濫用。

個人資料是珍貴資產的說法因為資料中介的存在得到支援，這是陰影中的秘密行業，購買、聚合、分析、推斷以及轉售私密個人資料，主要用於市場營銷【90】。初創公司按照它們的使用者數量，“眼球數”，——即它們的監視能力來估值。

因為資料很有價值，所以很多人都想要它。當然，公司也想要它——這就是為什麼它們一開始就收集資料的原因。但政府也想獲得它：透過秘密交易、脅迫、法律強制或者只是竊取【101】。當公司破產時，收集到的個人資料就是被出售的資產之一。而且資料安全很難保護，因此經常發生令人難堪的洩漏事件【102】。

這些觀察已經導致批評者聲稱，資料不僅僅是一種資產，而且是一種“有毒資產”【101】，或者至少是“有害物質”【103】。即使我們認為自己有能力阻止資料濫用，但每當我們收集資料時，我們都需要平衡收益以及這些資料落入惡人手中的風險：計算機系統可能會被犯罪分子或敵國特務滲透，資料可能會被內鬼洩露，公司可能會落入不擇手段的管理層手中，而這些管理者有著迥然不同的價值觀，或者國家可能被毫無愧色迫使我們交出資料的政權所接管。

俗話說，“知識就是力量”。更進一步，“在避免自己被審視的同時審視他人，是權力最重要的形式之一”【105】。這就是極權政府想要監控的原因：這讓它們有能力控制全體居民。儘管今天的科技公司並沒有公開地尋求政治權力，但是它們積累的資料與知識卻給它們帶來了很多權力，其中大部分是在公共監督之外偷偷進行的【106】。

回顧工業革命

資料是資訊時代的決定性特徵。網際網路，資料儲存，處理和軟體驅動的自動化正在對全球經濟和人類社會產生重大影響。我們的日常生活與社會組織在過去十年中發生了變化，而且在未來的十年中可能會繼續發生根本性的變化，所以我們會想到與工業革命對比【87,96】。

工業革命是透過重大的技術與農業進步實現的，它帶來了持續的經濟增長，長期的生活水平顯著提高。然而它也帶來了一些嚴重的問題：空氣汙染（由於煙霧和化學過程）和水汙染（工業垃圾和人類垃圾）是可怕的。工廠老闆生活在紛奢之中，而城市工人經常居住在非常糟糕的住房中，並且在惡劣的條件下長時間工作。童工很常見，甚至包括礦井中危險而低薪的工作。

制定保護措施花費了很長的時間，例如環境保護條例、工作場所安全條例、宣佈使用童工非法以及食品衛生檢查。毫無疑問，生產成本增加了，因為工廠再也不能把廢物倒入河流、銷售汙染的食物或者剝削工人。但是整個社會都從中受益良多，我們中很少會有人想回到這些管制條例之前的日子【87】。

就像工業革命有著黑暗面需要應對一樣，我們轉向資訊時代的過程中，也有需要應對與解決的重大問題。我相信資料的收集與使用就是其中一個問題。用 Bruce Schneier 的話來說【96】：

資料是資訊時代的汙染問題，保護隱私是環境挑戰。幾乎所有的電腦都能生產資訊。它堆積在周圍，開始潰爛。我們如何處理它——我們如何控制它，以及如何擺脫它——是資訊經濟健康發展的核心議題。正如我們今天回顧工業時代的早期年代，並想知道我們的祖先在忙於建設工業世界的過程時怎麼能忽略汙染問題；我們的孫輩在回望資訊時代的早期年代時，將會就我們如何應對資料收集和濫用的挑戰來評斷我們。

我們應該設法讓他們感到驕傲。

立法與自律

資料保護法可能有助於維護個人的權利。例如，1995 年的“歐洲資料保護指示”規定，個人資料必須“為特定的、明確的和合法的目的收集，而不是以與這些目的不相符的方式進一步處理”，並且資料必須“就收集的目的而言適當、相關、不過分。”【107】。

但是，這個立法在今天的網際網路環境下是否有效還是有疑問的【108】。這些規則直接否定了大資料的哲學，即最大限度地收集資料，將其與其他資料集結合起來進行試驗和探索，以便產生新的洞察。探索意味著將資料用於未曾預期的目的，這與使用者同意的“特定和明確”目的相反（如果我們可以有意義地表示同意的話）【109】。更新的規章正在制定中【89】。

那些收集了大量有關人的資料的公司反對監管，認為這是創新的負擔與阻礙。在某種程度上，這種反對是有道理的。例如，分享醫療資料時，存在明顯的隱私風險，但也有潛在的機遇：如果資料分析能夠幫助我們實現更好的診斷或找到更好的治療方法，能夠阻止多少人的死亡【110】？過度監管可能會阻止這種突破。在這種潛在機會與風險之間找出平衡是很困難的【105】。

從根本上說，我認為我們需要科技行業在個人資料方面的文化轉變。我們應該停止將使用者視作最佳化的指標資料，並記住他們是值得尊重、有尊嚴和能動性的人。我們應當在資料收集和實際處理中自我約束，以建立和維持依賴我們軟體的人們的信任【111】。我們應當將教育終端使用者視為己任，告訴他們我們是如何使用他們的資料的，而不是將他們矇在鼓裡。

我們應該允許每個人保留自己的隱私——即，對自己資料的控制——而不是透過監視來竊取這種控制權。我們控制自己資料的個體權利就像是國家公園的自然環境：如果我們不去明確地保護它、關心它，它就會被破壞。這將是公地的悲劇，我們都會因此而變得更糟。無所不在的監視並非不可避免的——我們現在仍然能阻止它。

我們究竟能做到哪一步，是一個開放的問題。首先，我們不應該永久保留資料，而是一旦不再需要就立即清除資料【111,112】。清除資料與不變性的想法背道而馳（請參閱“[不變性的侷限性](#)”），但這是可以解決的問題。我所看到的一種很有前景的方法是透過加密協議來實施訪問控制，而不僅僅是透過策略【113,114】。總的來說，文化與態度的改變是必要的。

本章小結

在本章中，我們討論了設計資料系統的新方式，而且也包括了我的個人觀點，以及對未來的猜測。我們從這樣一種觀察開始：沒有單種工具能高效服務所有可能的用例，因此應用必須組合使用幾種不同的軟體才能實現其目標。我們討論了如何使用批處理與事件流來解決這一 **資料整合 (data integration)** 問題，以便讓資料變更在不同系統之間流動。

在這種方法中，某些系統被指定為記錄系統，而其他資料則透過轉換衍生自記錄系統。透過這種方式，我們可以維護索引、物化檢視、機器學習模型、統計摘要等等。透過使這些衍生和轉換操作非同步且鬆散耦合，能夠防止一個區域中的問題擴散到系統中不相關部分，從而增加整個系統的穩健性與容錯性。

將資料流表示為從一個數據集到另一個數據集的轉換也有助於演化應用程式：如果你想變更其中一個處理步驟，例如變更索引或快取的結構，則可以在整個輸入資料集上重新執行新的轉換程式碼，以便重新衍生輸出。同樣，出現問題時，你也可以修復程式碼並重新處理資料以便恢復。

這些過程與資料庫內部已經完成的過程非常類似，因此我們將資料流應用的概念重新改寫為，**分拆 (unbundling)** 資料庫元件，並透過組合這些鬆散耦合的元件來構建應用程式。

衍生狀態可以透過觀察底層資料的變更來更新。此外，衍生狀態本身可以進一步被下游消費者觀察。我們甚至可以將這種資料流一路傳送至顯示資料的終端使用者裝置，從而構建可動態更新以反映資料變更，並在離線時能繼續工作的使用者介面。

接下來，我們討論了如何確保所有這些處理在出現故障時保持正確。我們看到可伸縮的強完整性保證可以透過非同步事件處理來實現，透過使用端到端操作識別符號使操作獨立等，以及透過非同步檢查約束。客戶端可以等到檢查通過，或者不等待繼續前進，但是可能會冒有違反約束需要道歉的風險。這種方法比使用分散式事務的傳統方法更具可伸縮性與可靠性，並且在實踐中適用於很多業務流程。

透過圍繞資料流構建應用，並非同步檢查約束，我們可以避免絕大多數的協調工作，建立保證完整性且效能仍然表現良好的系統，即使在地理散佈的情況下與出現故障時亦然。然後，我們對使用審計來驗證資料完整性，以及損壞檢測進行了一些討論。

最後，我們退後一步，審視了構建資料密集型應用的一些道德問題。我們看到，雖然資料可以用來做好事，但它也可能造成很大傷害：作出嚴重影響人們生活的決定卻難以申訴，導致歧視與剝削、監視常態化、曝光私密資訊。我們也冒著資料被洩露的風險，並且可能會發現，即使是善意地使用資料也可能會導致意想不到的後果。

由於軟體和資料對世界產生了如此巨大的影響，我們工程師們必須牢記，我們有責任為我們想要的那種世界而努力：一個尊重人們，尊重人性的世界。我希望我們能夠一起為實現這一目標而努力。

參考文獻

1. Rachid Belaid: “[Postgres Full-Text Search is Good Enough!](#),” *rachbelaid.com*, July 13, 2015.
2. Philippe Ajoux, Nathan Bronson, Sanjeev Kumar, et al.: “[Challenges to Adopting Stronger Consistency at Scale](#),” at *15th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, May 2015.
3. Pat Helland and Dave Campbell: “[Building on Quicksand](#),” at *4th Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2009.
4. Jessica Kerr: “[Provenance and Causality in Distributed Systems](#),” *blog.jessitron.com*, September 25, 2016.
5. Kostas Tzoumas: “[Batch Is a Special Case of Streaming](#),” *data-artisans.com*, September 15, 2015.
6. Shinji Kim and Robert Blafford: “[Stream Windowing Performance Analysis: Concord and Spark Streaming](#),” *concord.io*, July 6, 2016.
7. Jay Kreps: “[The Log: What Every Software Engineer Should Know About Real-Time Data's Unifying Abstraction](#),” *engineering.linkedin.com*, December 16, 2013.
8. Pat Helland: “[Life Beyond Distributed Transactions: An Apostate's Opinion](#),” at *3rd Biennial Conference on Innovative Data Systems Research (CIDR)*, January 2007.
9. “[Great Western Railway \(1835–1948\)](#),” Network Rail Virtual Archive, *networkrail.co.uk*.
10. Jacqueline Xu: “[Online Migrations at Scale](#),” *stripe.com*, February 2, 2017.
11. Molly Bartlett Dishman and Martin Fowler: “[Agile Architecture](#),” at *O'Reilly Software Architecture Conference*, March 2015.

12. Nathan Marz and James Warren: [Big Data: Principles and Best Practices of Scalable Real-Time Data Systems](#). Manning, 2015. ISBN: 978-1-617-29034-3
13. Oscar Boykin, Sam Ritchie, Ian O'Connell, and Jimmy Lin: “[Summingbird: A Framework for Integrating Batch and Online MapReduce Computations](#),” at *40th International Conference on Very Large Data Bases* (VLDB), September 2014.
14. Jay Kreps: “[Questioning the Lambda Architecture](#),” [oreilly.com](#), July 2, 2014.
15. Raul Castro Fernandez, Peter Pietzuch, Jay Kreps, et al.: “[Liquid: Unifying Nearline and Offline Big Data Integration](#),” at *7th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2015.
16. Dennis M. Ritchie and Ken Thompson: “[The UNIX Time-Sharing System](#),” *Communications of the ACM*, volume 17, number 7, pages 365–375, July 1974. doi:[10.1145/361011.361061](#)
17. Eric A. Brewer and Joseph M. Hellerstein: “[CS262a: Advanced Topics in Computer Systems](#),” lecture notes, University of California, Berkeley, [cs.berkeley.edu](#), August 2011.
18. Michael Stonebraker: “[The Case for Polystores](#),” [wp.sigmod.org](#), July 13, 2015.
19. Jennie Duggan, Aaron J. Elmore, Michael Stonebraker, et al.: “[The BigDAWG Polystore System](#),” *ACM SIGMOD Record*, volume 44, number 2, pages 11–16, June 2015. doi:[10.1145/2814710.2814713](#)
20. Patrycja Dybka: “[Foreign Data Wrappers for PostgreSQL](#),” [vertabelo.com](#), March 24, 2015.
21. David B. Lomet, Alan Fekete, Gerhard Weikum, and Mike Zwilling: “[Unbundling Transaction Services in the Cloud](#),” at *4th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2009.
22. Martin Kleppmann and Jay Kreps: “[Kafka, Samza and the Unix Philosophy of Distributed Data](#),” *IEEE Data Engineering Bulletin*, volume 38, number 4, pages 4–14, December 2015.
23. John Hugg: “[Winning Now and in the Future: Where VoltDB Shines](#),” [voltedb.com](#), March 23, 2016.
24. Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard: “[Differential Dataflow](#),” at *6th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2013.
25. Derek G Murray, Frank McSherry, Rebecca Isaacs, et al.: “[Naiad: A Timely Dataflow System](#),” at *24th ACM Symposium on Operating Systems Principles* (SOSP), pages 439–455, November 2013. doi:[10.1145/2517349.2522738](#)
26. Gwen Shapira: “[We have a bunch of customers who are implementing ‘database inside-out’ concept and they all ask ‘is anyone else doing it? are we crazy?’](#)” [twitter.com](#), July 28, 2016.
27. Martin Kleppmann: “[Turning the Database Inside-out with Apache Samza](#),” at *Strange Loop*, September 2014.
28. Peter Van Roy and Seif Haridi: *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004. ISBN: 978-0-262-22069-9
29. “[Juttle Documentation](#),” [juttle.github.io](#), 2016.
30. Evan Czaplicki and Stephen Chong: “[Asynchronous Functional Reactive Programming for GUIs](#),” at *34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), June 2013. doi:[10.1145/2491956.2462161](#)
31. Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter: “[A Survey on Reactive Programming](#),” *ACM Computing Surveys*, volume 45, number 4, pages 1–34, August 2013. doi:[10.1145/2501654.2501666](#)
32. Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak: “[Consistency Analysis in Bloom: A CALM and Collected Approach](#),” at *5th Biennial Conference on Innovative Data Systems Research* (CIDR), January 2011.
33. Felienne Hermans: “[Spreadsheets Are Code](#),” at *Code Mesh*, November 2015.
34. Dan Bricklin and Bob Frankston: “[VisiCalc: Information from Its Creators](#),” [danbricklin.com](#).
35. D. Sculley, Gary Holt, Daniel Golovin, et al.: “[Machine Learning: The High-Interest Credit Card of Technical Debt](#),” at *NIPS Workshop on Software Engineering for Machine Learning* (SE4ML), December 2014.
36. Peter Bailis, Alan Fekete, Michael J Franklin, et al.: “[Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity](#),” at *ACM International Conference on Management of Data* (SIGMOD), June 2015. doi:[10.1145/2723372.2737784](#)
37. Guy Steele: “[Re: Need for Macros \(Was Re: Icon\)](#),” email to *//1-discuss* mailing list, [people.csail.mit.edu](#), December 24, 2001.
38. David Gelernter: “[Generative Communication in Linda](#),” *ACM Transactions on Programming Languages and*

- Systems* (TOPLAS), volume 7, number 1, pages 80–112, January 1985. doi:10.1145/2363.2433
39. Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec: “[The Many Faces of Publish/Subscribe](#),” *ACM Computing Surveys*, volume 35, number 2, pages 114–131, June 2003. doi:10.1145/857076.857078
 40. Ben Stopford: “[Microservices in a Streaming World](#),” at *QCon London*, March 2016.
 41. Christian Posta: “[Why Microservices Should Be Event Driven: Autonomy vs Authority](#),” blog.christianposta.com, May 27, 2016.
 42. Alex Feyerke: “[Say Hello to Offline First](#),” hood.ie, November 5, 2013.
 43. Sebastian Burckhardt, Daan Leijen, Jonathan Protzenko, and Manuel Fähndrich: “[Global Sequence Protocol: A Robust Abstraction for Replicated Shared State](#),” at *29th European Conference on Object-Oriented Programming* (ECOOP), July 2015. doi:10.4230/LIPIcs.ECOOP.2015.568
 44. Mark Soper: “[Clearing Up React Data Management Confusion with Flux, Redux, and Relay](#),” medium.com, December 3, 2015.
 45. Eno Thereska, Damian Guy, Michael Noll, and Neha Narkhede: “[Unifying Stream Processing and Interactive Queries in Apache Kafka](#),” confluent.io, October 26, 2016.
 46. Frank McSherry: “[Dataflow as Database](#),” github.com, July 17, 2016.
 47. Peter Alvaro: “[I See What You Mean](#),” at *Strange Loop*, September 2015.
 48. Nathan Marz: “[Trident: A High-Level Abstraction for Realtime Computation](#),” blog.twitter.com, August 2, 2012.
 49. Edi Bice: “[Low Latency Web Scale Fraud Prevention with Apache Samza, Kafka and Friends](#),” at *Merchant Risk Council MRC Vegas Conference*, March 2016.
 50. Charity Majors: “[The Accidental DBA](#),” charity.wtf, October 2, 2016.
 51. Arthur J. Bernstein, Philip M. Lewis, and Shiyong Lu: “[Semantic Conditions for Correctness at Different Isolation Levels](#),” at *16th International Conference on Data Engineering* (ICDE), February 2000. doi:10.1109/ICDE.2000.839387
 52. Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan: “[Automating the Detection of Snapshot Isolation Anomalies](#),” at *33rd International Conference on Very Large Data Bases* (VLDB), September 2007.
 53. Kyle Kingsbury: [Jepsen blog post series](#), aphyr.com, 2013–2016.
 54. Michael Jouravlev: “[Redirect After Post](#),” theserverside.com, August 1, 2004.
 55. Jerome H. Saltzer, David P. Reed, and David D. Clark: “[End-to-End Arguments in System Design](#),” *ACM Transactions on Computer Systems*, volume 2, number 4, pages 277–288, November 1984. doi:10.1145/357401.357402
 56. Peter Bailis, Alan Fekete, Michael J. Franklin, et al.: “[Coordination-Avoiding Database Systems](#),” *Proceedings of the VLDB Endowment*, volume 8, number 3, pages 185–196, November 2014.
 57. Alex Yarmula: “[Strong Consistency in Manhattan](#),” blog.twitter.com, March 17, 2016.
 58. Douglas B Terry, Marvin M Theimer, Karin Petersen, et al.: “[Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System](#),” at *15th ACM Symposium on Operating Systems Principles* (SOSP), pages 172–182, December 1995. doi:10.1145/224056.224070
 59. Jim Gray: “[The Transaction Concept: Virtues and Limitations](#),” at *7th International Conference on Very Large Data Bases* (VLDB), September 1981.
 60. Hector Garcia-Molina and Kenneth Salem: “[Sagas](#),” at *ACM International Conference on Management of Data* (SIGMOD), May 1987. doi:10.1145/38713.38742
 61. Pat Helland: “[Memories, Guesses, and Apologies](#),” blogs.msdn.com, May 15, 2007.
 62. Yoongu Kim, Ross Daly, Jeremie Kim, et al.: “[Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors](#),” at *41st Annual International Symposium on Computer Architecture* (ISCA), June 2014. doi:10.1145/2678373.2665726
 63. Mark Seaborn and Thomas Dullien: “[Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges](#),” googleprojectzero.blogspot.co.uk, March 9, 2015.
 64. Jim N. Gray and Catharine van Ingen: “[Empirical Measurements of Disk Failure Rates and Error Rates](#),” Microsoft Research, MSR-TR-2005-166, December 2005.
 65. Annamalai Gurusami and Daniel Price: “[Bug #73170: Duplicates in Unique Secondary Index Because of Fix of Bug#68021](#),” bugs.mysql.com, July 2014.

66. Gary Fredericks: “Postgres Serializability Bug,” github.com, September 2015.
67. Xiao Chen: “HDFS DataNode Scanners and Disk Checker Explained,” blog.cloudera.com, December 20, 2016.
68. Jay Kreps: “Getting Real About Distributed System Reliability,” blog.empathybox.com, March 19, 2012.
69. Martin Fowler: “The LMAX Architecture,” martinfowler.com, July 12, 2011.
70. Sam Stokes: “Move Fast with Confidence,” blog.samstokes.co.uk, July 11, 2016.
71. “Sawtooth Lake Documentation,” Intel Corporation, intelledger.github.io, 2016.
72. Richard Gendal Brown: “Introducing R3 Corda™: A Distributed Ledger Designed for Financial Services,” genda.me, April 5, 2016.
73. Trent McConaghy, Rodolphe Marques, Andreas Müller, et al.: “BigchainDB: A Scalable Blockchain Database,” bigchaindb.com, June 8, 2016.
74. Ralph C. Merkle: “A Digital Signature Based on a Conventional Encryption Function,” at CRYPTO ’87, August 1987. doi:[10.1007/3-540-48184-2_32](https://doi.org/10.1007/3-540-48184-2_32)
75. Ben Laurie: “Certificate Transparency,” ACM Queue, volume 12, number 8, pages 10-19, August 2014. doi:[10.1145/2668152.2668154](https://doi.org/10.1145/2668152.2668154)
76. Mark D. Ryan: “Enhanced Certificate Transparency and End-to-End Encrypted Mail,” at Network and Distributed System Security Symposium (NDSS), February 2014. doi:[10.14722/ndss.2014.23379](https://doi.org/10.14722/ndss.2014.23379)
77. “Software Engineering Code of Ethics and Professional Practice,” Association for Computing Machinery, acm.org, 1999.
78. François Chollet: “Software development is starting to involve important ethical choices,” twitter.com, October 30, 2016.
79. Igor Perusic: “Making Hard Choices: The Quest for Ethics in Machine Learning,” engineering.linkedin.com, November 2016.
80. John Naughton: “Algorithm Writers Need a Code of Conduct,” theguardian.com, December 6, 2015.
81. Logan Kugler: “What Happens When Big Data Blunders?,” Communications of the ACM, volume 59, number 6, pages 15–16, June 2016. doi:[10.1145/2911975](https://doi.org/10.1145/2911975)
82. Bill Davidow: “Welcome to Algorithmic Prison,” theatlantic.com, February 20, 2014.
83. Don Peck: “They’re Watching You at Work,” theatlantic.com, December 2013.
84. Leigh Alexander: “Is an Algorithm Any Less Racist Than a Human?” theguardian.com, August 3, 2016.
85. Jesse Emusk: “How a Machine Learns Prejudice,” scientificamerican.com, December 29, 2016.
86. Maciej Ceglowski: “The Moral Economy of Tech,” idlewords.com, June 2016.
87. Cathy O’Neil: Weapons of Math Destruction: How Big Data Increases Inequality and Threatens Democracy. Crown Publishing, 2016. ISBN: 978-0-553-41881-1
88. Julia Angwin: “Make Algorithms Accountable,” nytimes.com, August 1, 2016.
89. Bryce Goodman and Seth Flaxman: “European Union Regulations on Algorithmic Decision-Making and a ‘Right to Explanation’,” [arXiv:1606.08813](https://arxiv.org/abs/1606.08813), August 31, 2016.
90. “A Review of the Data Broker Industry: Collection, Use, and Sale of Consumer Data for Marketing Purposes,” Staff Report, United States Senate Committee on Commerce, Science, and Transportation, commerce.senate.gov, December 2013.
91. Olivia Solon: “Facebook’s Failure: Did Fake News and Polarized Politics Get Trump Elected?” theguardian.com, November 10, 2016.
92. Donella H. Meadows and Diana Wright: Thinking in Systems: A Primer. Chelsea Green Publishing, 2008. ISBN: 978-1-603-58055-7
93. Daniel J. Bernstein: “Listening to a ‘big data’/‘data science’ talk,” twitter.com, May 12, 2015.
94. Marc Andreessen: “Why Software Is Eating the World,” The Wall Street Journal, 20 August 2011.
95. J. M. Porup: “‘Internet of Things’ Security Is Hilariously Broken and Getting Worse,” arstechnica.com, January 23, 2016.
96. Bruce Schneier: Data and Goliath: The Hidden Battles to Collect Your Data and Control Your World. W. W. Norton, 2015. ISBN: 978-0-393-35217-7
97. The Grugg: “Nothing to Hide,” grugg.tumblr.com, April 15, 2016.
98. Tony Beltramelli: “Deep-Spying: Spying Using Smartwatch and Deep Learning,” Masters Thesis, IT University of Copenhagen, December 2015. Available at arxiv.org/abs/1512.05616

99. Shoshana Zuboff: “Big Other: Surveillance Capitalism and the Prospects of an Information Civilization,” *Journal of Information Technology*, volume 30, number 1, pages 75–89, April 2015. doi:10.1057/jit.2015.5
00. Carina C. Zona: “Consequences of an Insightful Algorithm,” at *GOTO Berlin*, November 2016.
01. Bruce Schneier: “Data Is a Toxic Asset, So Why Not Throw It Out?,” *schneier.com*, March 1, 2016.
02. John E. Dunn: “The UK’s 15 Most Infamous Data Breaches,” *techworld.com*, November 18, 2016.
03. Cory Scott: “Data is not toxic - which implies no benefit - but rather hazardous material, where we must balance need vs. want,” *twitter.com*, March 6, 2016.
04. Bruce Schneier: “Mission Creep: When Everything Is Terrorism,” *schneier.com*, July 16, 2013.
05. Lena Ulbricht and Maximilian von Grafenstein: “Big Data: Big Power Shifts?,” *Internet Policy Review*, volume 5, number 1, March 2016. doi:10.14763/2016.1.406
06. Ellen P. Goodman and Julia Powles: “Facebook and Google: Most Powerful and Secretive Empires We’ve Ever Known,” *theguardian.com*, September 28, 2016.
07. Directive 95/46/EC on the protection of individuals with regard to the processing of personal data and on the free movement of such data, Official Journal of the European Communities No. L 281/31, *eur-lex.europa.eu*, November 1995.
08. Brendan Van Alsenoy: “Regulating Data Protection: The Allocation of Responsibility and Risk Among Actors Involved in Personal Data Processing,” Thesis, KU Leuven Centre for IT and IP Law, August 2016.
09. Michiel Rhoen: “Beyond Consent: Improving Data Protection Through Consumer Protection Law,” *Internet Policy Review*, volume 5, number 1, March 2016. doi:10.14763/2016.1.404
10. Jessica Leber: “Your Data Footprint Is Affecting Your Life in Ways You Can’t Even Imagine,” *fastcoexist.com*, March 15, 2016.
11. Maciej Ceglowski: “Haunted by Data,” *idlewords.com*, October 2015.
12. Sam Thielman: “You Are Not What You Read: Librarians Purge User Data to Protect Privacy,” *theguardian.com*, January 13, 2016.
13. Conor Friedersdorf: “Edward Snowden’s Other Motive for Leaking,” *theatlantic.com*, May 13, 2014.
14. Phillip Rogaway: “The Moral Character of Cryptographic Work,” Cryptology ePrint 2015/1162, December 2015.

上一章	目錄	下一章
第十一章：流處理	設計資料密集型應用	後記

術語表

請注意，本術語表中的定義簡短而簡單，旨在傳達核心思想，而不是術語的完整細微之處。有關更多詳細資訊，請參閱正文中的參考資料。

- **非同步 (asynchronous)**

不等待某些事情完成（例如，將資料傳送到網路中的另一個節點），並且不會假設要花多長時間。請參閱“[同步複製與非同步複製](#)”、“[同步網路與非同步網路](#)”以及“[系統模型與現實](#)”。

- **原子 (atomic)**

在併發操作的上下文中：描述一個在單個時間點看起來生效的操作，所以另一個併發程序永遠不會遇到處於“半完成”狀態的操作。另見隔離。

在事務的上下文中：將一些寫入操作分為一組，這組寫入要麼全部提交成功，要麼遇到錯誤時全部回滾。請參閱“[原子性](#)”和“[原子提交與兩階段提交](#)”。

- **背壓 (backpressure)**

接收方接收資料速度較慢時，強制降低傳送方的資料傳送速度。也稱為流量控制。請參閱“[訊息傳遞系統](#)”。

- **批處理 (batch process)**

一種計算，它將一些固定的（通常是大的）資料集作為輸入，並將其他一些資料作為輸出，而不修改輸入。見[第十一章](#)。

- **邊界 (bounded)**

有一些已知的上限或大小。例如，網路延遲情況（請參閱“[超時與無窮的延遲](#)”）和資料集（請參閱[第十一章](#)的介紹）。

- **拜占庭故障 (Byzantine fault)**

表現異常的節點，這種異常可能以任意方式出現，例如向其他節點發送矛盾或惡意訊息。請參閱“[拜占庭故障](#)”。

- **快取 (cache)**

一種元件，透過儲存最近使用過的資料，加快未來對相同資料的讀取速度。快取中通常存放部分資料：因此，如果快取中缺少某些資料，則必須從某些底層較慢的資料儲存系統中，獲取完整的資料副本。

- **CAP定理 (CAP theorem)**

一個被廣泛誤解的理論結果，在實踐中是沒有用的。請參閱“[CAP定理](#)”。

- **因果關係 (causality)**

事件之間的依賴關係，當一件事發生在另一件事情之前。例如，後面的事件是對早期事件的回應，或者依賴於更早的事件，或者應該根據先前的事件來理解。請參閱“[此前發生”的關係和併發](#)”和“[順序與因果關係](#)”。

- **共識 (consensus)**

分散式計算的一個基本問題，就是讓幾個節點同意某些事情（例如，哪個節點應該是資料庫叢集的領導者）。問題比乍看起來要困難得多。請參閱“[容錯共識](#)”。

- **資料倉庫 (data warehouse)**

一個數據庫，其中來自幾個不同的OLTP系統的資料已經被合併和準備用於分析目的。請參閱“[資料倉庫](#)”。

- **宣告式 (declarative)**

描述某些東西應有的屬性，但不知道如何實現它的確切步驟。在查詢的上下文中，查詢最佳化器採用宣告性查詢並決定如何最好地執行它。請參閱[“資料查詢語言”](#)。

- **非規範化 (denormalize)**

為了加速讀取，在標準資料集中引入一些冗餘或重複資料，通常採用快取或索引的形式。非規範化的值是一種預先計算的查詢結果，像物化檢視。請參閱[“單物件和多物件操作”](#)和[“從同一事件日誌中派生多個檢視”](#)。

- **衍生資料 (derived data)**

一種資料集，根據其他資料透過可重複執行的流程建立。必要時，你可以執行該流程再次建立衍生資料。衍生資料通常用於提高特定資料的讀取速度。常見的衍生資料有索引、快取和物化檢視。請參閱[第三部分](#)的介紹。

- **確定性 (deterministic)**

描述一個函式，如果給它相同的輸入，則總是產生相同的輸出。這意味著它不能依賴於隨機數字、時間、網路通訊或其他不可預測的事情。

- **分散式 (distributed)**

在由網路連線的多個節點上執行。對於部分節點故障，具有容錯性：系統的一部分發生故障時，其他部分仍可以正常工作，通常情況下，軟體無需瞭解故障相關的確切情況。請參閱[“故障與部分失效”](#)。

- **持久 (durable)**

以某種方式儲存資料，即使發生各種故障，也不會丟失資料。請參閱[“永續性”](#)。

- **ETL (Extract-Transform-Load)**

提取-轉換-載入 (Extract-Transform-Load)。從源資料庫中提取資料，將其轉換為更適合分析查詢的形式，並將其載入到資料倉庫或批處理系統中的過程。請參閱[“資料倉庫”](#)。

- **故障切換 (failover)**

在具有單一領導者的系統中，故障切換是將領導角色從一個節點轉移到另一個節點的過程。請參閱[“處理節點容機”](#)。

- **容錯 (fault-tolerant)**

如果出現問題（例如，機器崩潰或網路連線失敗），可以自動恢復。請參閱[“可靠性”](#)。

- **流量控制 (flow control)**

見背壓 (backpressure)。

- **追隨者 (follower)**

一種資料副本，僅處理領導者或主庫發出的資料變更，不直接接受來自客戶端的任何寫入。也稱為備庫、從庫、只讀副本或熱備份。請參閱[“領導者與追隨者”](#)。

- **全文檢索 (full-text search)**

透過任意關鍵字來搜尋文字，通常具有附加特徵，例如匹配類似的拼寫詞或同義詞。全文索引是一種支援這種查詢的次級索引。請參閱[“全文搜尋和模糊索引”](#)。

- **圖 (graph)**

一種資料結構，由頂點（可以指向的東西，也稱為節點或實體）和邊（從一個頂點到另一個頂點的連線，也稱為關係或弧）組成。請參閱[“圖資料模型”](#)。

- **雜湊 (hash)**

將輸入轉換為看起來像隨機數值的函式。相同的輸入會轉換為相同的數值，不同的輸入一般會轉換為不同的數值，也可能轉換為相同數值（也被稱為衝突）。請參閱“[根據鍵的雜湊分割槽](#)”。

- **幂等 (idempotent)**

用於描述一種操作可以安全地重試執行，即執行多次的效果和執行一次的效果相同。請參閱“[幂等性](#)”。

- **索引 (index)**

一種資料結構。透過索引，你可以根據特定欄位的值，在所有資料記錄中進行高效檢索。請參閱“[驅動資料庫的資料結構](#)”。

- **隔離性 (isolation)**

在事務上下文中，用於描述併發執行事務的互相干擾程度。序列執行具有最強的隔離性，不過其它程度的隔離也通常被使用。請參閱“[隔離性](#)”。

- **連線 (join)**

彙集有共同點的記錄。在一個記錄與另一個記錄有關（外來鍵，文件參考，圖中的邊）的情況下最常用，查詢需要獲取參考所指向的記錄。請參閱“[多對一和多對多的關係](#)”和“[Reduce側連線與分組](#)”。

- **領導者 (leader)**

當資料或服務被複制到多個節點時，領導者是被指定為可以接受資料變更的副本。領導者可以透過某些協議選舉產生，也可以由管理員手動選擇。領導者也被稱為主庫。請參閱“[領導者與追隨者](#)”。

- **線性化 (linearizable)**

表現為系統中只有一份透過原子操作更新的資料副本。請參閱“[線性一致性](#)”。

- **區域性 (locality)**

一種效能最佳化方式，如果經常在相同的時間請求一些離散資料，把這些資料放到一個位置。請參閱“[查詢的資料區域性](#)”。

- **鎖 (lock)**

一種保證只有一個執行緒、節點或事務可以訪問的機制，如果其它執行緒、節點或事務想訪問相同元素，則必須等待鎖被釋放。請參閱“[兩階段鎖定](#)”和“[領導者和鎖](#)”。

- **日誌 (log)**

日誌是一個只能以追加方式寫入的檔案，用於存放資料。預寫式日誌用於在儲存引擎崩潰時恢復資料（請參閱“[讓B樹更可靠](#)”）；結構化日誌儲存引擎使用日誌作為它的主要儲存格式（請參閱“[SSTables和LSM樹](#)”）；複製型日誌用於把寫入從領導者複製到追隨者（請參閱“[領導者與追隨者](#)”）；事件性日誌可以表現為資料流（請參閱“[分割槽日誌](#)”）。

- **物化 (materialize)**

急切地計算並寫出結果，而不是在請求時計算。請參閱“[聚合：資料立方體和物化檢視](#)”和“[物化中間狀態](#)”。

- **節點 (node)**

計算機上執行的一些軟體的例項，透過網路與其他節點通訊以完成某項任務。

- **規範化 (normalized)**

以沒有冗餘或重複的方式進行結構化。在規範化資料庫中，當某些資料發生變化時，你只需要在一個地方進行更改，而不是在許多不同的地方複製很多次。請參閱“[多對一和多對多的關係](#)”。

- **OLAP (Online Analytic Processing)**

線上分析處理。透過對大量記錄進行聚合（例如，計數，總和，平均）來表徵的訪問模式。請參閱“[事務處理還是分析？](#)”。

- **OLTP (Online Transaction Processing)**

線上事務處理。訪問模式的特點是快速查詢，讀取或寫入少量記錄，這些記錄通常透過鍵索引。請參閱“[事務處理還是分析？](#)”。

- **分割槽 (partitioning)**

將單機上的大型資料集或計算結果拆分為較小部分，並將其分佈到多臺機器上。也稱為分片。見[第六章](#)。

- **百分位點 (percentile)**

透過計算有多少值高於或低於某個閾值來衡量值分佈的方法。例如，某個時間段的第95個百分位響應時間是時間 t ，則該時間段中，95%的請求完成時間小於 t ，5%的請求完成時間要比 t 長。請參閱“[描述效能](#)”。

- **主鍵 (primary key)**

唯一標識記錄的值（通常是數字或字串）。在許多應用程式中，主鍵由系統在建立記錄時生成（例如，按順序或隨機）；它們通常不由使用者設定。另請參閱次級索引。

- **法定人數 (quorum)**

在操作完成之前，需要對操作進行投票的最少節點數量。請參閱“[讀寫的法定人數](#)”。

- **再平衡 (rebalance)**

將資料或服務從一個節點移動到另一個節點以實現負載均衡。請參閱“[分割槽再平衡](#)”。

- **複製 (replication)**

在幾個節點（副本）上保留相同資料的副本，以便在某些節點無法訪問時，資料仍可訪問。請參閱[第五章](#)。

- **模式 (schema)**

一些資料結構的描述，包括其欄位和資料型別。可以在資料生命週期的不同點檢查某些資料是否符合模式（請參閱“[文件模型中的模式靈活性](#)”），模式可以隨時變更（請參閱[第四章](#)）。

- **次級索引 (secondary index)**

與主要資料儲存器一起維護的附加資料結構，使你可以高效地搜尋與某種條件相匹配的記錄。請參閱“[其他索引結構](#)”和“[分割槽與次級索引](#)”。

- **可序列化 (Serializable)**

保證多個併發事務同時執行時，它們的行為與按順序逐個執行事務相同。請參閱第七章的“[可序列化](#)”。

- **無共享 (shared-nothing)**

與共享記憶體或共享磁碟架構相比，獨立節點（每個節點都有自己的CPU，記憶體和磁碟）透過傳統網路連線。見[第二部分](#)的介紹。

- **偏斜 (skew)**

各分割槽負載不平衡，例如某些分割槽有大量請求或資料，而其他分割槽則少得多。也被稱為熱點。請參閱“[負載偏斜與熱點消除](#)”和“[處理偏斜](#)”。

時間線異常導致事件以不期望的順序出現。請參閱“[快照隔離和可重複讀](#)”中的關於讀取偏差的討論，“[寫入偏差與幻讀](#)”中的寫入偏差以及“[有序事件的時間戳](#)”中的時鐘偏斜。

- **腦裂 (split brain)**

兩個節點同時認為自己是領導者的情況，這種情況可能違反系統擔保。請參閱“[處理節點宕機](#)”和“[真相由多數所定義](#)”。

- **儲存過程 (stored procedure)**

一種對事務邏輯進行編碼的方式，它可以完全在資料庫伺服器上執行，事務執行期間無需與客戶端通訊。請參閱“[真的序列執行](#)”。

- **流處理 (stream process)**

持續執行的計算。可以持續接收事件流作為輸入，並得出一些輸出。見[第十一章](#)。

- **同步 (synchronous)**

非同步的反義詞。

- **記錄系統 (system of record)**

一個儲存主要權威版本資料的系統，也被稱為真相的來源。首先在這裡寫入資料變更，其他資料集可以從記錄系統衍生。請參閱[第三部分](#)的介紹。

- **超時 (timeout)**

檢測故障的最簡單方法之一，即在一段時間內觀察是否缺乏響應。但是，不可能知道超時是由於遠端節點的問題還是網路中的問題造成的。請參閱“[超時與無窮的延遲](#)”。

- **全序 (total order)**

一種比較事物的方法（例如時間戳），可以讓你總是說出兩件事中哪一件更大，哪件更小。總的來說，有些東西是無法比擬的（不能說哪個更大或更小）的順序稱為偏序。請參閱“[因果順序不是全序的](#)”。

- **事務 (transaction)**

為了簡化錯誤處理和併發問題，將幾個讀寫操作分組到一個邏輯單元中。見[第七章](#)。

- **兩階段提交 (2PC, two-phase commit)**

一種確保多個數據庫節點全部提交或全部中止事務的演算法。請參閱“[原子提交與兩階段提交](#)”。

- **兩階段鎖定 (2PL, two-phase locking)**

一種用於實現可序列化隔離的演算法，該演算法透過事務獲取對其讀取或寫入的所有資料的鎖，直到事務結束。請參閱“[兩階段鎖定](#)”。

- **無邊界 (unbounded)**

沒有任何已知的上限或大小。反義詞是邊界 (bounded)。

後記

關於作者

Martin Kleppmann 是英國劍橋大學分散式系統的研究員。此前他曾在網際網路公司擔任過軟體工程師和企業家，其中包括 LinkedIn 和 Rapportive，負責大規模資料基礎架構。在這個過程中，他以艱難的方式學習了一些東西，他希望這本書能夠讓你避免重蹈覆轍。

Martin 是一位常規會議演講者，博主和開源貢獻者。他認為，每個人都應該有深刻的技術理念，深層次的理解能幫助我們開發出更好的軟體。



關於譯者

馮若航

PostgreSQL DBA @ TanTan

Alibaba+-Finplus 架構師/全棧工程師 (2015 ~ 2017)

後記

《設計資料密集型應用》封面上的動物是 **印度野豬** (**Sus scrofa cristatus**)，它是在印度、緬甸、尼泊爾、斯里蘭卡和泰國發現的一種野豬的亞種。與歐洲野豬不同，它們有更高的背部鬃毛，沒有體表絨毛，以及更大更直的頭骨。

印度野豬有一頭灰色或黑色的頭髮，脊背上有短而硬的毛。雄性有突出的犬齒（稱為 T），用來與對手戰鬥或抵禦掠食者。雄性比雌性大，這些物種平均肩高 33-35 英寸，體重 200-300 磅。他們的天敵包括熊、老虎和各種大型貓科動物。

這些動物夜行且雜食——它們吃各種各樣的東西，包括根、昆蟲、腐肉、堅果、漿果和小動物。野豬經常因為破壞農作物的根被人們所熟知，他們造成大量的破壞，並被農民所敵視。他們每天需要攝入 4,000 ~ 4,500 卡路里的能量。野豬有發達的嗅覺，這有助於尋找地下植物和挖掘動物。然而，它們的視力很差。

野豬在人類文化中一直具有重要意義。在印度教傳說中，野豬是毗溼奴神的化身。在古希臘的喪葬紀念碑中，它是一個勇敢失敗者的象徵（與勝利的獅子相反）。由於它的侵略，它被描繪在斯堪的納維亞、日耳曼和盎格魯撒克遜戰士的盔甲和武器上。在中國十二生肖中，它象徵著決心和急躁。

O'Reilly 封面上的許多動物都受到威脅，這些動物對世界都很重要。要了解有關如何提供幫助的更多資訊，請訪問 animals.oreilly.com。

封面圖片來自 Shaw's Zoology。封面字型是 URW Typewriter 和 Guardian Sans。文字字型是 Adobe Minion Pro；圖中的字型是 Adobe Myriad Pro；標題字型是 Adobe Myriad Condensed；程式碼字型是 Dalton Maag 的 Ubuntu Mono。