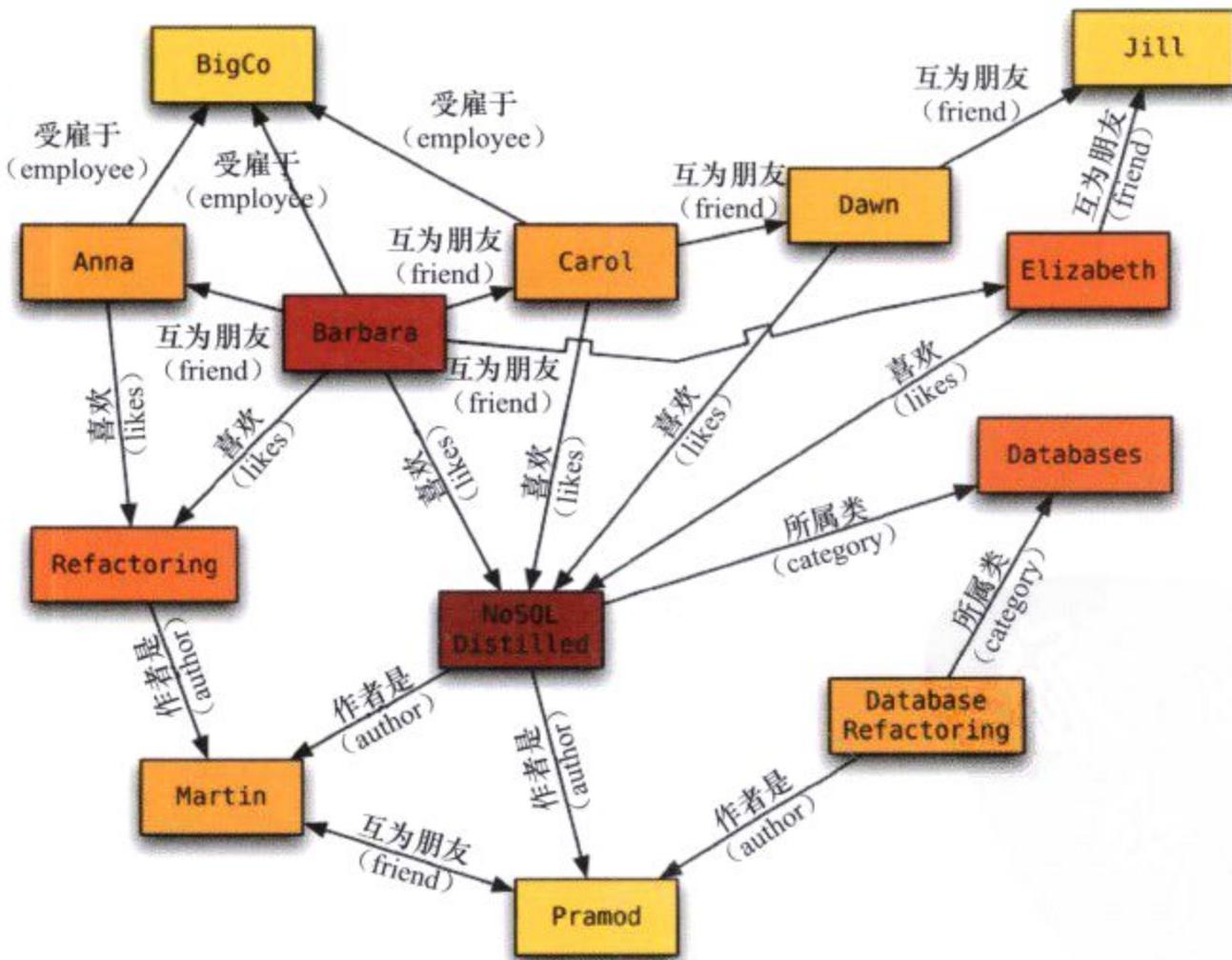


NoSQL Distilled  
A Brief Guide to the Emerging World of Polyglot Persistence

# NoSQL精粹

(美) Pramod J. Sadalage Martin Fowler 著  
爱飞翔 译



持续增长的海量数据，催生了一种名为NoSQL的非关系型数据库。其倡导者宣称，该技术可构建出更高效、更易扩展且更易编码的系统。

本书言简意赅地介绍了这项新技术。书中解释了NoSQL数据库的工作原理，以及NoSQL可能优于传统关系型数据库之处。作者讲解了有关概念，以指导读者评估NoSQL数据库是否有利于解决当前项目需求，并介绍了采用NoSQL数据库后还需深入研究的其他技术。

本书第一部分专注于讲解无模式数据模型、聚合、新的分布式模型、CAP定理、映射-化简等核心概念，第二部分研究实现NoSQL时的架构与设计问题。作者以实际用例演示了如何在工作中运用NoSQL数据库，并以Riak、MongoDB、Cassandra和Neo4j为例，着重讲解了每一种NoSQL数据库的典型用法。

此外，本书利用Pramod Sadalage先生的开拓性研究成果，展示了怎样在模式迁移问题上实现演进式设计：这是运用NoSQL数据库时必备的技巧。本书结尾描绘了NoSQL如何引领即将到来的混合持久化新时代，那将是多种数据库并存的世界，架构师可针对每种数据访问类型选择最优技术。

## 本书内容包括

- 评估企业级应用程序是否应使用NoSQL技术
- 理解部署NoSQL时的架构权衡
- 用NoSQL简化开发工作，避免因为在内存数据结构与关系型数据库数据结构之间映射而引发的问题
- 对比时下几项领先的NoSQL数据库产品
- 研究CQL与Cypher查询语言
- 管理数据库的性能、可靠程度、可用性及故障恢复能力。
- 在敏捷开发环境中使用NoSQL
- 在元数据搜寻/检索管理、文本分析、社交网络、商务智能、金融服务等领域运用NoSQL
- 用运行于集群中的NoSQL数据库降低“大数据”问题的解决成本
- 如何以CAP定理为思路，考量一致性、可用性与延迟问题
- 怎样用映射-化简模式在集群中并行计算
- NoSQL这一术语为何没有严格的定义



PEARSON

www.pearson.com

客服热线: (010) 88378991 88361066  
购书热线: (010) 68326294 88379649 68995259  
投稿热线: (010) 88379604

数字阅读: www.hzmedia.com.cn  
华章网站: www.hzbook.com  
网上购书: www.china-pub.com

上架指导: 计算机/数据库

ISBN 978-7-111-43303-3



9 787111 433033 >

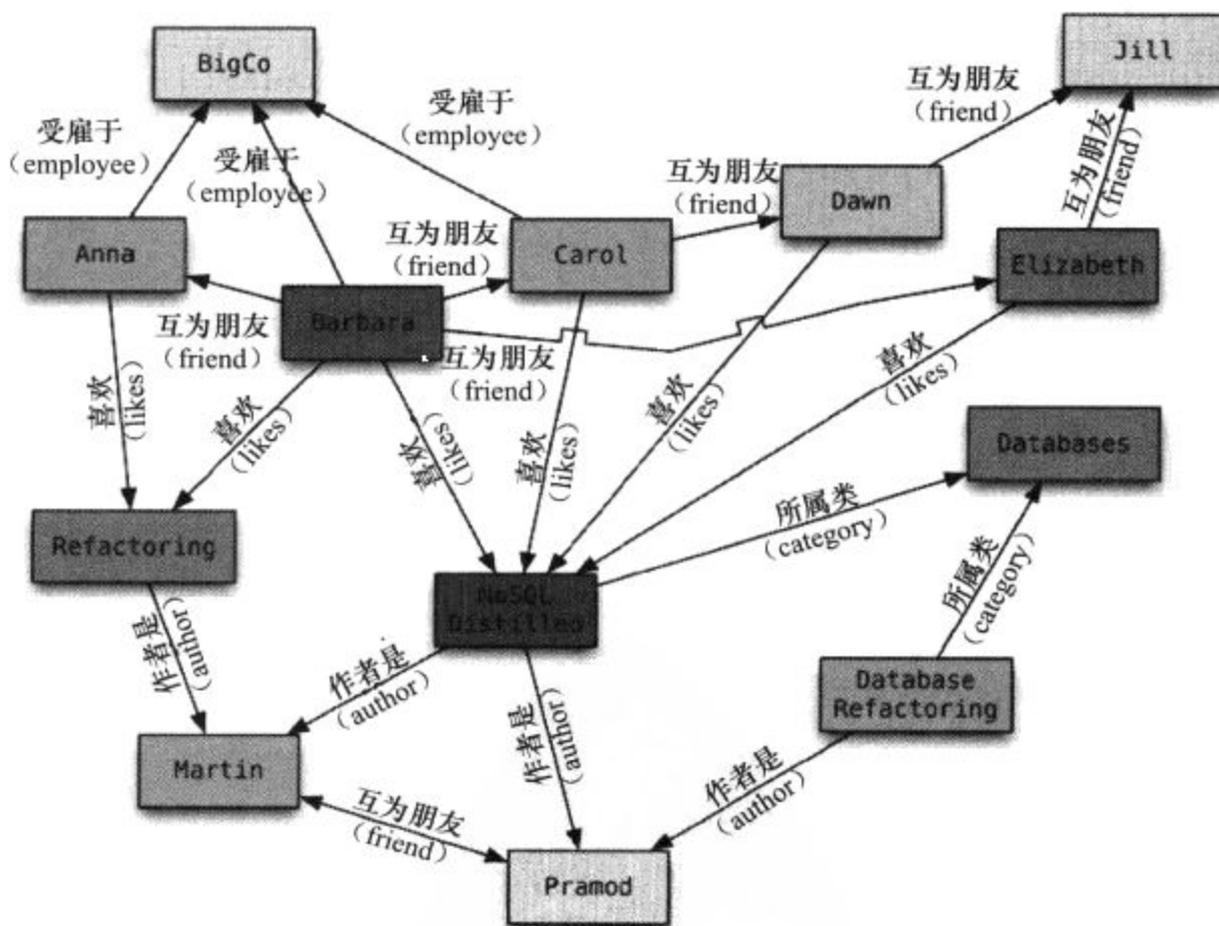
定价: 49.00元

# NoSQL Distilled

A Brief Guide to the Emerging World of Polyglot Persistence

# NoSQL精粹

(美) Pramod J. Sadalage Martin Fowler 著  
爱飞翔 译



## 图书在版编目 (CIP) 数据

NoSQL 精粹 / (美) 塞得拉吉 (Sadalage, P. J.), (美) 福勒 (Fowler, M.) 著; 爱飞翔译. —北京: 机械工业出版社, 2013.9

(华章程序员书库)

书名原文: NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence

ISBN 978-7-111-43303-3

I. N… II. ①塞… ②福… ③爱… III. 数据库—系统 IV. TP311.138

中国版本图书馆 CIP 数据核字 (2013) 第 161584 号

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问 北京市展达律师事务所

本书版权登记号: 图字: 01-2012-6632

Authorized translation from the English language edition, entitled *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*, 9780321826626 by Pramod J. Sadalage, Martin Fowler, published by Pearson Education, Inc., Copyright © 2013.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Chinese Simplified language edition published by Pearson Education Asia Ltd., and China Machine Press Copyright © 2013.

本书中文简体字版由 Pearson Education (培生教育出版集团) 授权机械工业出版社在中华人民共和国境内 (不包括中国台湾地区和澳门特别行政区) 独家出版发行。未经出版者书面许可, 不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封底贴有 Pearson Education (培生教育出版集团) 激光防伪标签, 无标签者不得销售。

本书为考虑是否可以使用和如何使用 NoSQL 数据库的企业提供了可靠的决策依据。它由世界级软件开发大师和软件开发“教父”Martin Fowler 与 Jolt 生产效率大奖图书作者 Pramod J. Sadalage 共同撰写。书中全方位比较了关系型数据库与 NoSQL 数据库的异同; 分别以 Riak、MongoDB、Cassandra 和 Neo4J 为代表, 详细讲解了键值数据库、文档数据库、列族数据库和图数据库这 4 大类 NoSQL 数据库的优劣势、用法和适用场合; 深入探讨了实现 NoSQL 数据库系统的各种细节, 以及与关系型数据库的混用。

全书分为两部分, 共 15 章: 第一部分 (第 1~7 章) 主要讲述 NoSQL 的核心概念。其中第 1 章解释了 NoSQL 发展迅速的原因; 第 2 章描述了在 NoSQL 领域的三种主要的数据模型中如何体现“聚合”这一概念; 第 3 章介绍了聚合的缺点; 第 4 章描述了数据库如何在集群中分布数据; 第 5 章论及了更新与读取操作对一致性的影响; 第 6 章讨论了版本戳; 第 7 章描述了适合用在 NoSQL 系统中的“映射-化简”操作。第二部分 (第 8~15 章) 讲述了如何实现 NoSQL 数据库系统。其中第 8 章~第 11 章每章各以一种 NoSQL 数据库为例, 演示了如何实现第一部分介绍的概念; 第 12 章解释了数据如何在强模式系统与无模式系统之间迁移; 第 13 章着眼于混合持久化领域的趋势; 第 14 章探讨了在混合持久化领域中会考虑到的其他一些技术; 第 15 章提供了选择数据库时可以参考的一些建议。

机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑: 关敏

三河市杨庄长鸣印刷装订厂印刷

2013 年 9 月第 1 版第 1 次印刷

186mm×240mm·11 印张

标准书号: ISBN 978-7-111-43303-3

定 价: 49.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzsj@hzbook.com

## 译者序

数据库技术是企业级应用程序经常会用到的，在习惯于传统的关系型数据库多年之后，一群先驱开始探索新的解决方案。随着待处理的数据量逐渐增多，大家越来越需要一种在集群环境中易于编程且执行效率高的大数据处理技术，在此情势下，NoSQL 数据库应运而生。

新技术诞生后，我们应该以既稳健又前瞻的心态看待它。一方面不宜在尚未充分理解时就盲目跟风，另一方面却也要紧密关注业界动向，顺应发展趋势。本书正是这样一本具有指导意义的手册，它虽篇幅短小，内涵却非常丰富。

书中首先分析了传统关系型数据库所要解决的问题，以及在解决手段方面存在的待改进之处。然后引入 NoSQL 这一新概念，并从数据模型、分布模型、一致性、版本戳、映射 - 化简操作等角度逐一详细比较了它与关系型数据库的异同。读者可以领略到 NoSQL 传承并发展了关系型数据库的哪些优秀特性，放弃了哪些不适合的特性，又新增了哪些内容，同时还了解到产生这些异同的原因。

以上就是本书的第一部分。读者在理解了上述概念后，会在第二部分看到同为 NoSQL 数据库的四种子类型之间的关系。本书分别以 Riak、MongoDB、Cassandra 和 Neo4J 为代表，举例讲解了键值数据库、文档数据库、列族数据库和图数据库这四大类 NoSQL 数据库的用法，分别说明了其优势与劣势。

其后，书中又系统地讲解了关系型数据库与 NoSQL 数据库的数据模式迁移问题，描述了混合持久化这一新兴领域的样貌，并简述了此领域还将用到的一系列新技术。

通过分析与对比，我们可以发现，关系型数据库与 NoSQL 数据库并不矛盾，它们是从两个不同的角度来解决数据存储问题。在混合持久化的新环境下，二者互为补充，相辅相成，若运用得当，其整体效果将好于单用一门技术。

本书最后得出结论：鉴于 NoSQL 技术尚未成熟，所以大部分企业级应用程序开发者目前还应以现有关系型数据库为主，但是在前瞻性项目中可以先试先行；与此同时，

不论是否打算开始运用 NoSQL 数据库，都应该将传统项目中与数据库相关的代码抽离，便于将来 NoSQL 技术成熟后迅速切换。

通过阅读本书，我们可以掌握数据库技术的新动向，将现有的数据库技术细节从应用程序业务中提取出来，把它按数据用法分别封装到各个模块里，然后，根据书中所讲的知识，通过实践找出项目中适合改用 NoSQL 数据库的地方，并在 NoSQL 日臻成熟的过程中逐渐迁移过去。

如果我们能在 NoSQL 技术发展之初就把握其脉动，积极尝试并及时总结经验，那么待其成熟时，就能在混合持久化领域占得先机了，这也正是本书的一大意义。

由于本书作者乃业界巨擘，其观点影响颇广，为求谨慎，译文写出了很多中文术语的英文原名，以便读者查对，个别容易引发误解的称谓，加引号以强调其特殊含义。

在翻译过程中，得到了机械工业出版社华章公司诸位编辑与工作人员的帮助，在此深表谢意。同时还要感谢乔晓萌女士对我翻译工作的支持和鼓励。

本书由爱飞翔翻译，舒亚林与张军也参与了部分翻译工作。由于时间仓促，译者水平有限，错误与疏漏之处敬请读者批评指正。若您对本书有意见和建议，请通过电子邮件 [eastarstormlee@gmail.com](mailto:eastarstormlee@gmail.com) 联系译者，或访问网址 [http://agilemobidev.com/eastarlee/book/nosql\\_distilled/](http://agilemobidev.com/eastarlee/book/nosql_distilled/) 留言。

爱飞翔

2013 年 6 月 15 日

# 前 言

我们已经在企业级计算领域研究了 20 余年，编程语言、架构、平台、软件开发流程等技术都在改变，然而这期间有一件事却一直没变，那就是：大家依然使用关系型数据库来存储数据。虽说也出现了一些挑战关系型数据库的产品，而且有的还在某些领域成功了，但是总体来说，留给架构师的数据存储问题仍然是选择使用哪款关系型数据库的问题。

稳定性在此领域颇受重视。企业的数据比程序存储的时间要长很多（至少大家都是这么说的。当然啦，我们也见过许多非常老的程序）。拥有一个既稳定，又容易理解，而且还能让许多应用程序编程平台访问的数据库，是非常有价值的。

不过，关系型数据库现在碰上新对手了，它的名字叫 NoSQL。由于我们需要处理的数据量越来越大，必须以商用服务器集群来构建大型硬件平台，因此 NoSQL 就应运而生了。这也使大家要再次考虑那个存在已久的难题，即代码如何才能同关系型数据库良好地结合起来。

“NoSQL”这个词的定义是非常不明确的。它泛指那些最近诞生的非关系型数据库，诸如 Cassandra、MongoDB、Neo4J 和 Riak 等。它们主张使用无模式（schemaless）<sup>⊖</sup>的数据，可以运行在集群环境中，并且能够牺牲传统数据库所具备的一致性，以换取另外一些有用的特性。NoSQL 的倡导者声称，使用它们可以构建出性能更高、扩展度更好且更易编程的系统。

这会不会敲响了关系型数据库即将灭亡的第一声警钟呢？还是说 NoSQL 要抢走数据库领域的头把交椅？我们的回答是：“这两种情况都不会出现。”关系型数据库是一个非常强大的工具，我们希望能长时间使用下去；然而大家也要看到一场深远的变革，那就是：关系型数据库不再是唯一的选择了。我们认为，数据库领域正进入混合持久

---

⊖ schema 也译作“纲要”、“大纲”、“概要”等，本书统称其为模式，与表示“处理技术问题的固定范式”之“模式”（Pattern）一词不同。——译者注

化 (Polyglot Persistence) 时代, 由企业乃至个人研发的应用程序, 可以使用多种技术来管理数据。因此架构师需要熟悉这些技术, 并且能根据不同的需求做出适当的选择。若非如此, 笔者怎会花那么多时间和精力来写这本书呢?

本书给诸位读者提供足够多的信息, 协助大家在以后的研发过程中思考: 项目是否真的值得使用 NoSQL 数据库。每个项目都是不同的, 我们不可能写出一个简单的决策树, 用它来选出合适的数据存储方式。与之相反, 本书力求讲解大量的背景知识, 以便大家了解 NoSQL 的工作原理, 这样的话, 你不用在互联网上四处寻找, 就能够做出适合自己项目的决定了。笔者刻意将本书写得很短, 以便读者能够快速阅览它。虽说本书不会回答各种具体问题, 但是, 它可以帮你缩小考虑的范围, 让你明白自己当前应该提出哪些问题。

## NoSQL 数据库为何引人关注

我们来看一下大家选用 NoSQL 数据库的两个主要原因。

- 应用程序的开发效率。在很多应用程序的开发过程中, 大量精力和时间都放在了内存 (in-memory) 数据结构和关系型数据库之间的映射上面。NoSQL 数据库可以提供一种更加符合应用程序需求的数据模型, 从而简化了数据交互, 减少了所需编写、调试并修改的代码量。
- 大规模的数据。企业所重视的是, 数据库要能够快速获取并处理数据。他们发现, 即便关系型数据库能达成这一目标, 其成本也很高。主要原因在于, 关系型数据库是为独立运行的计算机而设计的, 但是现在大家通常使用由更小、更廉价的计算机所组成的集群来计算数据, 这样更实惠些。许多 NoSQL 数据库正是为集群环境而设计, 因此它们更适合大数据量的应用场景。

## 本书内容

本书分为两个部分。第一部分主要讲述核心概念, 让读者能够判断出 NoSQL 数据库是否适合自己, 并且了解各种 NoSQL 数据库之间的差别。第二部分更加专注于实现 NoSQL 数据库系统。

第 1 章解释了 NoSQL 发展如此迅速的原因: 由于需要处理的数据量越来越多, 所以大型系统的扩展方式, 由原来在单一计算机上的纵向扩展, 转变为在计算机集群上

的横向扩展。这也印证了许多 NoSQL 数据库的数据模型所具备的一个重要特性，那就是：可以把内容密切相关的数据组织成一种丰富的结构，并将其显式存储起来，以便作为一个单元（unit）来访问。本书中，我们将这种类型的结构称为聚合（aggregate）。

第 2 章描述了在 NoSQL 领域的三种主要数据模型中，如何体现“聚合”这一概念。这三种数据库模型是：“键值模型”（key-value，参见 2.2 节），“文档模型”（document，参见 2.2 节）和“列族模型”（column family，参见 2.3 节）。聚合为许多种应用提供了一个自然的交互单元，既改善了集群的运行状况，又使编写程序来访问数据库变得更为容易。第 3 章转到聚合的缺点上面：难以处理位于不同聚合的实体之间的关系（参见 3.1 节）。这自然就引出了图数据库（参见 3.2 节），它是一个不属于面向聚合（aggregated-oriented）阵营的 NoSQL 数据模型。我们也会讲到 NoSQL 数据库的共同特性：它们都是以“无模式”的形式来操作的（参见 3.3 节）。模式的这种特性确实提供了更大的灵活性，但是它并不像大家想象的那么万能。

在讲完 NoSQL 数据模型方面的内容之后，我们接下来要讲分布模型。第 4 章描述了数据库如何在集群中分布数据。这个问题又细分为“分片”（sharding，参见 4.2 节）和“复制”（replication），复制方式可以是“主从复制”（master-slave replication，参见 4.3 节）或者“对等复制”（peer-to-peer replication，参见 4.4 节）。了解完分布模型的概念后，接下来要讲“一致性”（consistency）问题。与关系型数据库相比，NoSQL 数据库在一致性方面提供了更多选择，这么做是因为 NoSQL 要更好地支持集群。于是，第 5 章谈到了更新与读取操作对一致性的影响（分别参见 5.1 节和 5.2 节），如何在一致性与持久性之间进行仲裁（参见 5.5 节），以及如何放宽对持久性的约束以提升其他特性（参见 5.4 节）。如果之前听过 NoSQL，那么就应该听过“CAP 定理”（The CAP Theorem）。5.3 节中介绍了 CAP 定理的相关知识，告诉大家如何根据该理论来权衡一致性与其他特性。

前面这些章节主要侧重于如何分布数据并保持其一致性，接下来的两章讨论了要完成这项工作所需的一些重要工具。第 6 章讲述了版本戳（version stamp），它用来记录数据库的内容变更，并且可以检测数据是否一致。第 7 章概述了“映射 - 化简”（Map-Reduce）操作，这种计算方式很适合在集群中组织并行计算，因而也适用于 NoSQL 系统。

讲完这些概念后，我们针对以下 4 种数据库各举一些例子，来演示如何实现上述

概念。第 8 章使用 Riak 来演示“键值数据库”，第 9 章使用 MongoDB 作为“文档数据库”的示例，第 10 章选用 Cassandra 来探讨“列族数据库”，第 11 章选择了 Neo4J 作为“图数据库”的示例。此处必须强调：要想全面学习数据库，只依靠这些章节是不够的。因为除此之外还有很多内容，没办法写在这本书中，而且还有更多东西必须尝试之后才能学会。本书选择这些示例，并不是建议大家在工作中使用它们，其目的是让读者知道数据的各种存储方式，明白不同的数据库技术如何使用前面提到的概念。读者会看到这些数据库系统都需要何种程序代码，并且简单了解使用它们时所应遵循的开发思路。

有些人经常会觉得：因为 NoSQL 数据库没有模式，所以在应用程序的生命期中，可以毫无困难地改变其数据结构。本书不同意此观点，因为无模式的数据库其实隐含了一种模式，在实现数据结构变更时，也必须修改其规则。所以，第 12 章解释了数据如何在强模式与无模式系统之间迁移。

所有这一切都清楚地表明：NoSQL 不是独立存在的，也不会取代关系型数据库。第 13 章着眼于混合持久化领域的发展趋势：多种数据存储方式将共存，有时甚至会存在于同一个应用中。第 14 章将大家的视野扩展至本书之外，在混合持久化领域中，考虑一些前面没有涉及的技术。

掌握了前面所讲的全部内容之后，读者就应该明白如何选择合适的数据存储技术了。所以最后一章（第 15 章）提供了一些选择数据库时可以参考的建议。笔者认为，有两个关键因素：找到一种高效的编程模型。其数据存储模型要非常符合待开发的应用程序，并且确保其获取数据的效率与弹性均符合开发者的需求。从 NoSQL 诞生之初，我们就担心没有一套定义明确的流程可以遵循，现在，你仍然需要结合自己的需求，来验证自己所选择的数据库技术是否合适。

本书只是个简要的概述，所以笔者一直在尽力压缩篇幅。我们精选了自己认为最重要的信息，这部分内容读者就不必再去找了。如果打算认真研究这些技术，那就需要进一步研读本书以外的知识了，不过，我们还是希望本书能为你的探索之路开个好头。

还需要强调的是：计算机领域中的这些技术是日新月异的，存储技术的某些重要方面在不断变化，每年都会出现新的特性与新的数据库。笔者投入了巨大的精力来专门讲述概念，因为就算底层技术变了，对这些概念的理解也依然有价值。我们非常确信，

本书所讲的大部分概念都会历久绵长，但绝不能保证所有概念都会如此。

## 谁应该阅读本书

如果正在考虑选用某种形式的 NoSQL 数据库，那就应该阅读本书。选用 NoSQL 的原因可能是你打算做一个新的项目，也可能是既有项目遭遇瓶颈，所以要将其数据库迁移到 NoSQL 数据库上。

本书致力于给读者提供足够的信息，以判断自己所选的 NoSQL 技术是否符合需求，如果符合的话，应该深入研究哪些工具。我们设想本书的主要读者是架构师或技术主管，然而那些想大概了解这门新技术的软件管理人员也可以阅读本书。此外，对于想大概了解这项技术的开发人员来说，这也是本很好的入门读物。

本书不讲编程细节，也不去部署某个特定的数据库，那些内容留待更为专业的教材来写吧。我们还严格限制了本书的篇幅。笔者认为，这种书应该在坐飞机的时候读：它不会回答你提出的所有问题，但却会激发你提出一堆好问题来。

若是之前已经深入研究了 NoSQL 领域，那么本书可能不会增加你的知识储备。不过，它仍然有助于你将之前学到的东西解释给别人听。把围绕着 NoSQL 的争论理解清楚是很重要的，尤其当你要劝说别人在项目中也采用 NoSQL 技术时更是如此。

## 本书要讲的数据库类型

本书遵循常见的分类方式，也就是按照数据模型来划分各种 NoSQL 数据库。下表列出了 4 种数据模型，以及归属于每种数据模型的数据库。这份列表并不完整，其中只列出了较为常见的数据库。撰写本书时，在 <http://nosql-database.org> 与 <http://nosql.mypopescu.com/kb/nosql> 都可查阅到更为完整的列表。每个分类中，以斜体标出的数据库，都会在相关章节中作为范例来讲解。

数据模型	范例数据库	数据模型	范例数据库
键值（参见第 8 章）	BerkeleyDB LevelDB Memcached Project Voldemort Redis <i>Riak</i>	文档（参见第 9 章）	CouchDB <i>MongoDB</i> OrientDB RavenDB Terrastore

(续)

数据模型	范例数据库	数据模型	范例数据库
列族 (参见第 10 章)	Amazon SimpleDB <i>Cassandra</i> HBase Hypertable	图 (参见第 11 章)	FlockDB HyperGraphDB Infinite Graph <i>Neo4J</i> OrientDB

这样划分的目的是从每一类数据库中，选出一个最有代表性的工具来讲。尽管每个分类下列出的那些数据库各不相同，不可像这样一概而论，但是，书中提到的那些具体示例，其实大多数情况下也适用于此分类中的其他数据库。我们会从“键值数据库”、“文档数据库”、“列族数据库”和“图数据库”这 4 类中各选一个作为范例，此外，在必要时，还会提到可以满足某个特定功能的其他产品。

按数据模型来分类是可行的，但却失之武断。不同数据模型之间的界限往往是模糊的，比如键值和文档数据库（参见 2.2 节）之间的区别就不是很明显。许多数据库并不能明确地归入某一类。例如，OrientDB 称自己既是文档数据库又是图数据库。

## 致谢

首先感谢 ThoughtWorks<sup>⊖</sup> 的诸位同仁，在过去的几年中，很多同事在交付的项目中应用了 NoSQL。笔者写作本书的动机主要来源于他们的经验，而这些经验亦是能印证 NoSQL 技术价值的实用信息。目前为止通过使用 NoSQL 数据存储积累了一些有益的经验，基于这些经验，我们认为：NoSQL 是一项重要的数据存储技术，它正引发该领域内的一场重大变革。

我们也要感谢举办公开讲座、发表文章和博客来分享 NoSQL 使用心得的各种社群。若是大家都不愿意与同行分享研究成果的话，那么许多软件开发领域的发展就不为人知了。特别感谢谷歌及亚马逊的 BigTable 和 Dynamo 技术规范论文，它们对 NoSQL 的发展影响深远。也要感谢为开源 NoSQL 数据库的开发提供赞助及技术贡献的公司。这一次发生在数据存储领域的变革，与以往相比有一个较为有趣的差别：

⊖ ThoughtWorks 是一家在全球诸多国家都有分公司的软件设计与定制领袖企业。主要业务模式是通过咨询来改善企业 IT 组织及软件开发方法，以软件带动企业业务发展。详情参见：<http://www.thoughtworks.com/>。  
——译者注

NoSQL 的发展深度植根于开源工作。

特别感谢 ThoughtWorks 公司给予笔者时间来写作本书。我们两个大约同一时间加入 ThoughtWorks，并且在这里工作了 10 余年。ThoughtWorks 对我们来说一直是个非常友好的大家庭，同时也是知识和实践的来源。在这个良好的环境中，大家可以公开分享各自所学的知识，这与传统的系统交付公司（System Delivery Organization）非常不同。

Bethany Anders-Beck、Ilias Bartolini、Tim Berglund、Duncan Craig、Paul Duvall、Oren Eini、Perryn Fowler、Michael Hunger、Eric Kascic、Joshua Kerievsky、Anand Krishnaswamy、Bobby Norton、Ade Oshineye、Thiyagu Palanisamy、Prasanna Pendse、Dan Pritchett、David Rice、Mike Roberts、Marko Rodriguez、Andrew Slocum、Toby Tripp、Steve Vinoski、Dean Wampler、Jim Webber 和 Wee Witthawaskul 审阅了本书初稿，并提出了改进建议。

此外，Pramod 要感谢绍姆堡图书馆（Schaumburg Library）提供的一流服务和安静的写作空间；感谢爱女 Arhana 和 Arula，你们知道爸爸到图书馆是为了写书，而没有带你们同去；感谢爱妻 Rupali，你给了我巨大的支持和帮助，让我能够集中精力完成本书。

# 目 录

译者序

前言

## 第一部分 概 念

第 1 章 为什么使用 NoSQL .....	2
1.1 关系型数据库的价值 .....	3
1.1.1 获取持久化数据 .....	3
1.1.2 并发 .....	3
1.1.3 集成 .....	4
1.1.4 近乎标准的模型 .....	4
1.2 阻抗失谐 .....	4
1.3 “应用程序数据库”与“集成数据库” .....	6
1.4 蜂拥而来的集群 .....	8
1.5 NoSQL 登场 .....	9
1.6 要点 .....	13
第 2 章 聚合数据模型 .....	15
2.1 聚合 .....	16
2.1.1 关系模型与聚合模型示例 .....	16
2.1.2 面向聚合的影响 .....	20
2.2 键值数据模型与文档数据模型 .....	22
2.3 列族存储 .....	23
2.4 面向聚合数据库总结 .....	25
2.5 延伸阅读 .....	26

2.6	要点	26
<b>第 3 章</b>	<b>数据模型详解</b>	<b>27</b>
3.1	关系	28
3.2	图数据库	29
3.3	无模式数据库	31
3.4	物化视图	33
3.5	构建数据存取模型	34
3.6	要点	39
<b>第 4 章</b>	<b>分布式模型</b>	<b>40</b>
4.1	单一服务器	41
4.2	分片	41
4.3	主从复制	43
4.4	对等复制	45
4.5	结合“分片”与“复制”技术	47
4.6	要点	48
<b>第 5 章</b>	<b>一致性</b>	<b>49</b>
5.1	更新一致性	50
5.2	读取一致性	51
5.3	放宽“一致性”约束	55
5.4	放宽“持久性”约束	60
5.5	仲裁	62
5.6	延伸阅读	63
5.7	要点	64
<b>第 6 章</b>	<b>版本戳</b>	<b>65</b>
6.1	“商业事务”与“系统事务”	66
6.2	在多节点环境中生成版本戳	68
6.3	要点	70
<b>第 7 章</b>	<b>映射 - 化简</b>	<b>71</b>
7.1	基本“映射 - 化简”	72

7.2	分区与归并	73
7.3	组合“映射-化简”计算	76
7.3.1	举例说明两阶段“映射-化简”	77
7.3.2	增量式“映射-化简”	80
7.4	延伸阅读	81
7.5	要点	81

## 第二部分 实 现

第 8 章	键值数据库	84
8.1	何谓“键值数据库”	85
8.2	键值数据库特性	86
8.2.1	一致性	86
8.2.2	事务	87
8.2.3	查询功能	87
8.2.4	数据结构	89
8.2.5	可扩展性	89
8.3	适用案例	90
8.3.1	存放会话信息	90
8.3.2	用户配置信息	90
8.3.3	购物车数据	90
8.4	不适用场合	90
8.4.1	数据间关系	90
8.4.2	含有多项操作的事务	91
8.4.3	查询数据	91
8.4.4	操作关键字集合	91
第 9 章	文档数据库	92
9.1	何谓文档数据库	93
9.2	特性	94
9.2.1	一致性	94

9.2.2	事务	95
9.2.3	可用性	96
9.2.4	查询功能	97
9.2.5	可扩展性	99
9.3	适用案例	100
9.3.1	事件记录	100
9.3.2	内容管理系统及博客平台	101
9.3.3	网站分析与实时分析	101
9.3.4	电子商务应用程序	101
9.4	不适用场合	101
9.4.1	包含多项操作的复杂事务	101
9.4.2	查询持续变化的聚合结构	101
<b>第 10 章</b>	<b>列族数据库</b>	<b>102</b>
10.1	何谓列族数据库	103
10.2	特性	103
10.2.1	一致性	105
10.2.2	事务	107
10.2.3	可用性	107
10.2.4	查询功能	108
10.2.5	可扩展性	110
10.3	适用案例	110
10.3.1	事件记录	110
10.3.2	内容管理系统与博客平台	111
10.3.3	计数器	111
10.3.4	限期使用	111
10.4	不适用场合	112
<b>第 11 章</b>	<b>图数据库</b>	<b>113</b>
11.1	何谓图数据库	114
11.2	特性	115

11.2.1	一致性	116
11.2.2	事务	117
11.2.3	可用性	117
11.2.4	查询功能	118
11.2.5	可扩展性	121
11.3	适用案例	122
11.3.1	互联数据	122
11.3.2	安排运输路线、分派货物和基于位置的服务	123
11.3.3	推荐引擎	123
11.4	不适用场合	123
<b>第 12 章</b>	<b>模式迁移</b>	<b>124</b>
12.1	模式变更	125
12.2	变更关系型数据库的模式	125
12.2.1	迁移全新项目	126
12.2.2	迁移既有项目	127
12.3	变更 NoSQL 数据库的模式	129
12.3.1	增量迁移	131
12.3.2	迁移图数据库的模式	132
12.3.3	改变聚合结构	132
12.4	延伸阅读	133
12.5	要点	133
<b>第 13 章</b>	<b>混合持久化</b>	<b>134</b>
13.1	各异的数据存储需求	135
13.2	混用各类数据库	135
13.3	将直接数据库操作封装为服务	137
13.4	扩展数据库以增强其功能	138
13.5	选用合适的数据库技术	139
13.6	企业使用混合持久化技术时的考量	139
13.7	部署复杂度	140

13.8 要点	140
<b>第 14 章 超越 NoSQL</b>	<b>141</b>
14.1 文件系统	142
14.2 事件溯源	142
14.3 内存映像	145
14.4 版本控制	146
14.5 XML 数据库	146
14.6 对象数据库	147
14.7 要点	147
<b>第 15 章 选择合适的数据库</b>	<b>148</b>
15.1 程序员的工作效率	149
15.2 数据访问性能	150
15.3 继续沿用默认的关系型数据库	151
15.4 抽离数据库策略以降低风险	152
15.5 要点	153
15.6 结语	153
<b>参考资料</b>	<b>154</b>

# 第一部分

# 概 念

- 第 1 章 为什么使用 NoSQL
- 第 2 章 聚合数据模型
- 第 3 章 数据模型详解
- 第 4 章 分布式模型
- 第 5 章 一致性
- 第 6 章 版本戳
- 第 7 章 映射 - 化简

# 第 1 章

## 为什么使用 NoSQL

几乎从我们踏入软件行业开始，关系型数据库就是存储正式数据的首选方案，企业级应用尤其如此。如果你是某个新项目的架构师，那么唯一能做的决定也许就是选用哪一款关系型数据库罢了。（要是公司的服务提供商在行业内占主导地位，那你经常连这种选择的余地都没有。）其他数据库技术也曾多次企图染指这一领域，比如 20 世纪 90 年代的对象数据库（object database），但是，这些替代品都没有产生任何实际的威胁。

在关系型数据库长久占领市场之后，NoSQL 的出现让我们眼前一亮，为之惊喜。本章将探讨关系型数据库成为主流的原因，同时也要解释笔者为什么觉得目前正在崛起的 NoSQL 数据库并不会只是昙花一现。

## 1.1 关系型数据库的价值

因为关系型数据库已经成为计算机文化的一部分，所以大家能够很自然地接受它。于是，我们在这里需要先回顾一下它的优点。

### 1.1.1 获取持久化数据

数据库的最大价值也许就是持久存储大量数据了。在大多数的计算机架构中，有两个存储区域：一个是速度快但是数据易丢失的“主存储器”（main memory），另一个则是存储量大但速度较慢的“后备存储器”（backing store）。主存储器的空间十分有限，一旦断电或操作系统出错，那么全部数据就会丢失，因此，为了保存数据，我们要将它写入后备存储器。最常见的后备存储器就是磁盘（虽说近来磁盘也可以作为持久内存使用）。

后备存储的形式多种多样。许多生产力应用程序（productivity application，比如文字处理软件）将其作为一个文件，保存在操作系统的文件系统之中。然而大多数企业级应用程序都以数据库做后备存储。在数据量较大时，数据库比文件系统更灵活，它能让应用程序快速而便捷地获取其中一小部分数据。

### 1.1.2 并发

在企业级应用程序中，多个用户会一起访问同一份数据体，并且可能要修改这份数据。大多数情况下，他们都在不同数据区域内各自操作，但是，偶尔也会同时操作一小块数据。如此一来，我们就要花心思协调这些交互操作了，以免出现诸如两人同时预订某家旅馆同一间客房的情况。

要想在并发操作的情况下获取正确的结果是极为困难的，即便是最谨慎的程序员，也会掉入各种错误陷阱中。由于同时访问某个企业应用的用户很多，而且应用程序还会与其他系统一起运行，所以出错的几率就更大了。关系型数据库通过“事务”<sup>⊖</sup>来控制对其数据的访问，以便处理此问题。尽管这并不是万全之策（当你要预订的房间刚被别人订走时，仍然需要处理“事务错误”），但事务机制还是可以在并发情况下良好运行的，并且能应付各种麻烦事情。

---

⊖ transaction，是数据库执行过程中的逻辑单位，由数据库操作序列构成，并不完全等同于日常用语中的“事务”、“交易”等词汇。详情参见：<http://zh.wikipedia.org/wiki/数据库事务>。——译者注

事务在处理错误时也有用。通过事务更改数据时，如果在处理变更的过程中出错了，那么就可以回滚（roll back）这一事务，以保证数据不受破坏。

### 1.1.3 集成

企业级应用程序居于一个丰富的生态系统中，它需要与其他应用程序协同工作，而那些程序是由不同的团队合作开发出来的。这种应用程序间的合作很棘手，因为它意味着各个开发者群体之间必须联动。不同的应用程序经常要使用同一份数据，而且某个应用程序更新完数据之后，必须让其他应用程序知道这份数据已经改变了。

常用的办法是使用**共享数据库集成**（shared database integration）[Hohpe and Woolf]，多个应用程序都将数据保存在同一个数据库中。这样一来，所有应用程序很容易就能使用彼此的数据了。而且，与多用户访问单一应用程序时一样，数据库的并发控制机制也可以应对多个应用程序。

### 1.1.4 近乎标准的模型

关系型数据库之所以成功，是因为它们以近乎标准的方式提供了上面简述的这些核心优势。如此一来，开发人员和数据库专家就可以学习基本的关系模型，并且将其运用到许多项目中了。尽管各种关系型数据库之间仍有差异，但其核心机制相同：不同厂商的 SQL 方言<sup>Ⓔ</sup>相似，“事务”的操作方式也几乎一样。

## 1.2 阻抗失谐

关系型数据库有许多优势，但绝非完美。甚至从诞生之初，就有很多令人不满意之处。

对于应用程序开发者来说，最令他们失望的就是，关系模型和内存中的数据结构之间存在差异。这种现象通常称为“阻抗失谐”<sup>Ⓕ</sup>。关系模型把数据组织成“表”（table）和“行”（row），更准确地说，应该是“关系”（relation）和“元组”（tuple）。在关系模型中，元组是由“键值对”（name-value pair）构成的集合，而关系则是元组

---

Ⓔ SQL dialect，此处的 dialect 是指一门编程语言的扩展或变种形式，与日常用语中的“方言”有相似之处，但并不完全等同。详情参见：[http://en.wikipedia.org/wiki/Dialect\\_\(computing\)](http://en.wikipedia.org/wiki/Dialect_(computing))。——译者注

Ⓕ 英文原文是 impedance mismatch，该词是数据库领域术语，反用了微波电子学术语“阻抗匹配”（impedance matching，<http://zh.wikipedia.org/wiki/阻抗匹配>），用来比喻数据模型与实际编程语言不搭调的窘境。详情参见：[http://en.wikipedia.org/wiki/Object-relational\\_impedance\\_mismatch](http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch)。——译者注

的集合。（“元组”一词在关系型数据库中的定义与数学和许多编程语言中的意思略有不同。很多语言中也有“元组”这一数据类型，不过它指的是值序列。）SQL 操作所使用及返回的数据都是“关系”，于是就形成了一种从数学角度来看十分优雅的关系代数（relational algebra）。

建立在“关系”基础上的数据库，的确有几分优雅与简洁，然而它也由此产生了一些局限，特别是“关系元组”（relational tuple）中的值必须很简单才行。它们不能包含“嵌套记录”（nested record）或“列表”（list）等任何结构。而内存中的数据结构则无此限制，它可以使用的数据组织形式比“关系”更丰富。这样一来，如果在内存中使用了较为丰富的数据结构，那么要把它保存到磁盘之前，必须先将其转换成“关系”形式。于是就发生了“阻抗失谐”：需要在两种不同的表示形式之间转译（见图 1.1）。

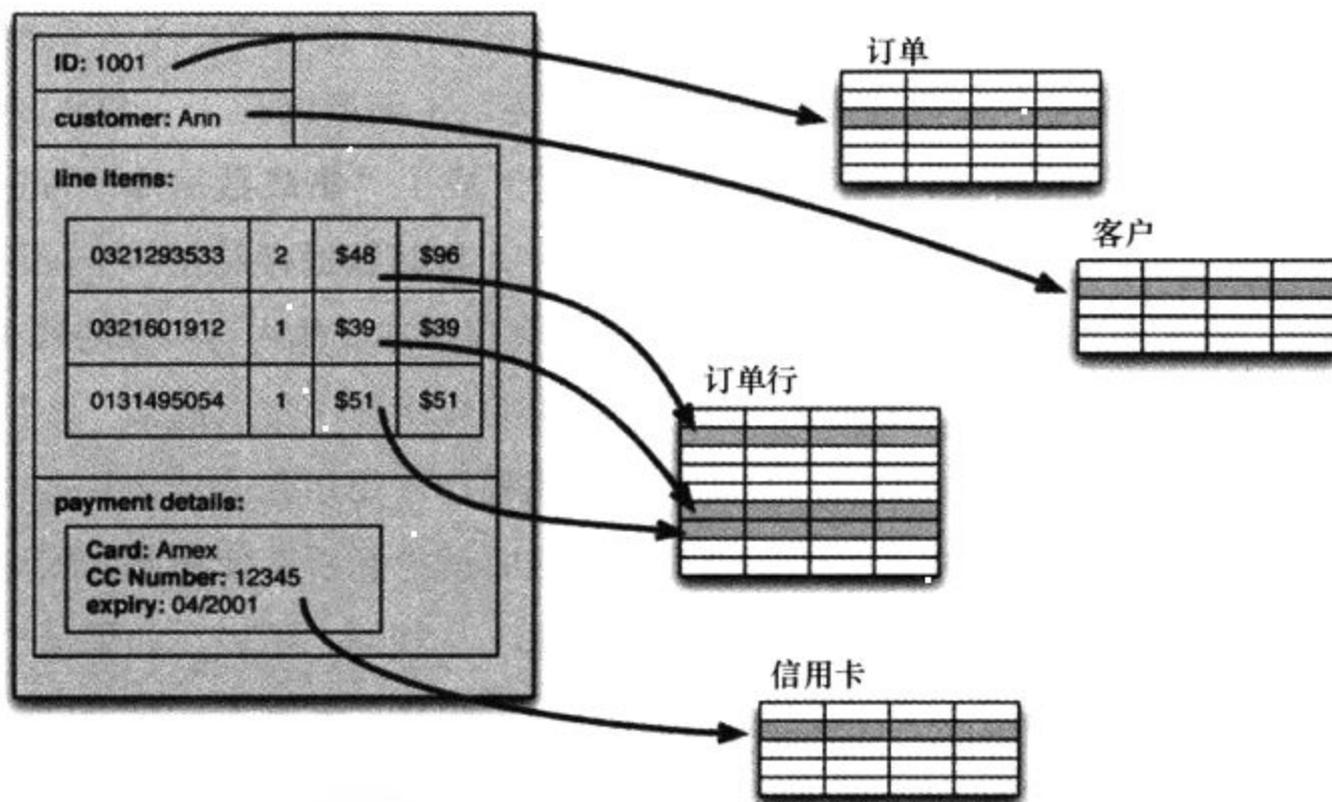


图 1.1 这是一张订单。在用户界面中看起来像一个聚合结构，然而其数据却存放在关系型数据库的多张表中。每张表内的行对应具体的数据

阻抗失谐是造成应用程序开发者不满的主要原因。在 20 世纪 90 年代，许多人认为关系型数据库将被那种能把内存数据结构复制到磁盘上的数据库所取代。那十年正是面向对象编程语言蓬勃发展的时候，而且面向对象数据库也随之出现。这两种技术都想成为 21 世纪的主流软件开发环境。

然而，在面向对象语言成为编程主力军时，面向对象数据库却销声匿迹了。关系型数据库经受住了挑战。它强化了自身在集成机制中的角色，为最标准的数据操作语

言（即 SQL）所支持，并且使应用程序开发者与数据库管理员之间的分工更为明晰。

随处可见的“对象 - 关系映射框架”（object-relational mapping framework）可以更为轻松地解决阻抗失谐问题。例如 Hibernate<sup>①</sup>和 iBATIS<sup>②</sup>，它们实现了著名的“映射模式”（mapping pattern）[Fowler, PoEAA]。然而，映射问题却依然存在。“对象 - 关系映射框架”简化了很多繁重的工作，但如果过分依赖它而刻意不使用数据库的话，那么这套框架本身就成了问题：因为查询性能会下降。

关系型数据库在 21 世纪初期一直占领着企业级计算领域，但是在过去的 10 年间，动摇它们统治地位的因素出现了。

### 1.3 “应用程序数据库”与“集成数据库”

工作了一定年头的程序员，在酒吧里聚会时，偶尔还会争辩一下关系型数据库到底为什么能战胜面向对象数据库。然而笔者认为，主要原因在于：SQL 充当了应用程序之间的一种集成机制。数据库在这种情况下成了“集成数据库”（integration database）：通常由不同团队所开发的多个应用程序，将其数据存储在一个公用的数据库中。这就提高了数据通信的效率，因为所有应用程序都在操作内容一致的持久数据。

共享同一份集成数据库也有缺点。为了能将很多应用程序集成起来，数据库的结构必须设计得复杂一些。而事实上，它比单个应用程序所要用到的结构复杂得多。此外，如果某个应用程序想要修改存储的数据，那么它就得和所有使用此数据库的其他应用程序相协调。各种应用程序的结构和性能要求不尽相同，所以，如果某个应用程序用其他程序获取到的索引来插入数据，那么就可能出错。实际上，每个应用程序基本都是由不同团队开发的，这也就是说，数据库通常不能任由应用程序更新其数据。为了保持数据库的完整性，我们需要将这一责任交由数据库自身负责。

另一种办法是，将数据库视为“应用程序数据库”（application database），其内容只能由一个应用程序的代码库直接访问，而这份代码库是由一个团队来维护的。对应用程序数据库来说，只有开发应用程序的那个团队才需要知道其结构，这样一来，模

---

① 是一种 Java 语言下的对象关系映射解决方案。详情参见：<http://www.hibernate.org/>。——译者注

② iBATIS 一词来源于“internet”和“abatis”（鹿寨）的组合，是由 Clinton Begin 在 2001 年发起的开源项目。最初侧重于密码软件的开发，现在是一个基于 Java 的持久层框架。详情参见：<http://zh.wikipedia.org/wiki/IBATIS>。——译者注

式的维护与更新就更简单了。由于应用程序开发团队同时管理数据库和应用程序代码，因此可以把维护数据库完整性的工作放在应用程序代码中。

如果采用上述办法，那么交互工作就可以转由应用程序接口来做了，这样一来，就会出现更好的交互协议，而且还可以修改应用程序的接口。在 20 世纪初期，Web 服务有了明显变化 [Daigneau]：应用程序间可以通过 HTTP 协议通信了。在各种广泛使用的通信机制中，Web 服务算是一种新的形式，它挑战了“以 SQL 来共享数据库”的使用方式。（其中大量内容都打着“面向服务架构”（Service-Oriented Architecture）的旗号，而这一术语最显著的特征，就是它缺乏固定含义。）

转用 Web 服务作为集成机制，会引发一个有趣的现象，那就是，所交换的数据可以拥有更为灵活的结构。如果用 SQL 交互，那么必须使用关系型的数据结构；然而用 Web 服务交互时，则可以使用嵌套记录及列表等更丰富的数据结构。这些数据通常存放在 XML 文档中，最近也流行以 JSON 格式来保存它们。通常我们都想减少远程通信中的往返次数，所以可将结构丰富的信息放到单一的“请求”（request）或“响应”（response）中传输。

如果打算使用 Web 服务做集成，那么大多数情况下可以使用传输文本信息的 HTTP 协议。然而，如果交互过程对性能要求很高，那么可以考虑使用二进制协议（binary protocol）。除非你能肯定自己真的需要用它，否则还是用文本协议（text protocol）容易些，看一下当前因特网上的各种协议就明白了。

一旦决定使用应用程序数据库，那么选择具体数据库技术的余地就更大了。由于内部数据库与外部通信服务之间已经解耦<sup>⊖</sup>，所以外界并不关心数据如何存储，这样就可以选用非关系型数据库了。此外，关系型数据库的许多特性，诸如安全性等，对应用程序用处不大，因为它们可以交给使用该数据库的外围应用程序（enclosing application）来做。

尽管应用程序数据库很自由，但它却不会对其他数据存储方式造成很大冲击。大多数愿意采用应用程序数据库的团队，还是无法摆脱关系型数据库。毕竟除了数据库的灵活性之外，使用应用程序数据库还会带来很多好处（这也是笔者通常推荐它的原

---

⊖ decouple，是“耦合”（couple）的反义词，为计算机专用术语。通俗地讲，两者“解耦”的意思就是两者之间已经各自独立，没有联动关系了。详情参见：[http://zh.wikipedia.org/wiki/耦合性\\_\(计算机科学\)](http://zh.wikipedia.org/wiki/耦合性_(计算机科学))。——译者注

因)。大家已经很熟悉关系型数据库了，而且一般情况下它们运行得都很好，或者说，至少还过得去。也许假以时日，越来越多的人会转而使用应用程序数据库，让它能在关系型数据库的独霸之下闯出一片天地。不过，到了那时，真正挑战关系型数据库的，恐怕另有其人。

## 1.4 蜂拥而来的集群

新千年伊始，IT业受20世纪90年代“互联网泡沫”（dot-com bubble）的影响，许多人质疑因特网的经济前景。然而，2000年至2009年这十年间，很多大型网络公司的规模都在急剧增加。

规模的增加体现在很多方面。网站开始用非常详细的方式来记录活动和结构，并且出现了链接、社交网络、活动日志、测绘数据等大型数据集。伴随着数据量的增长，用户也越来越多：很多大型网站每天都要服务大量访问者，随之也积累了巨额财富。

必须有更多的计算资源，才能应对数据和流量的增加。处理此类增长有两种方案：纵向扩展（scale up）及横向扩展（scale out）。如果要纵向扩展，那么就需要功能更强大的计算机，要购买更多的处理器、磁盘存储空间和内存。但是机器的功能越强，其成本也越高，更何况其扩展尺度也有限。另一种方案是：采用由多个小型计算机组成的集群。集群中的小型机可以使用性价比较高的硬件，这样就能降低扩展所需的成本。而且，这么做也更有弹性：我们可以构建一个高度稳定的集群，就算其中的某些电脑经常发生故障，也不会影响整个集群的运行。

在大型企业向集群迁移的过程中，产生了一个新问题：关系型数据库并不是设计给集群用的。Oracle RAC<sup>⊖</sup>或Microsoft SQL Server这种适用于集群的关系型数据库，要依靠一种名为“共享磁盘子系统”（shared disk subsystem）的概念才能运行。它们使用一种可以支持集群的文件系统，该文件系统可将数据写入随时可用的磁盘子系统中。但是这样一来，磁盘子系统就成了整个集群的软肋<sup>⊖</sup>。关系型数据库也可以把数据划分为几个集合，并将其分别放在各自独立的服务器上运行，于是就能有效地对数据库分

---

⊖ 是 Oracle Real Application Cluster 的简写，官方中文文档一般称其为“真正应用集群”，它通常由两台或者两台以上同构计算机及共享存储设备构成，可提供强大的数据库处理能力。详情参见：<http://zh.wikipedia.org/wiki/RAC>。——译者注

⊖ 原文为 a single point of failure，即“单一故障点”，意思是，只要这里发生故障，整个系统就崩溃了。——译者注

片（参见 4.2 节）了。这么做虽然能将负载分散到多个服务器之中，但是应用程序必须控制所有分片，它要知道数据库中的每份小数据都存放在哪个服务器里才行。而且，查询、参照完整性（referential integrity）、事务、一致性控制（consistency control）等操作也都无法以跨分片的方式执行了。经常能听见大家用“非常手法”（unnatural act）一词来称呼此现象。

除了技术问题外，还有一个更麻烦的地方就是许可费。商用的关系型数据库通常按单台服务器计费，所以在集群中使用会非常贵，这也增加了与采购部门沟通的难度。

由于关系型数据库与集群不协调，所以某些公司开始考虑另外一类存储数据的办法。谷歌和亚马逊这两家影响力巨大的公司更是如此，二者都是这种庞大集群的典型用户。此外，它们还要收集巨量数据。上述因素都促使其创新。这两个蒸蒸日上的公司，都有雄厚的技术实力，这也提供了实现想法的手段和机会，于是它们自然就萌生了自主研发关系型数据库的念头。在 2000 至 2009 年间，谷歌和亚马逊这两家公司都将各自的成果分别发表在一篇简短却极具影响力的论文上，它们就是 BigTable<sup>Ⓔ</sup>（谷歌）与 Dynamo<sup>Ⓕ</sup>（亚马逊）。

经常有人提出：因为亚马逊和谷歌所操作的数据规模远远超过大多数企业的需要，所以它们追求的解决方案可能并不适用于一般企业。诚然，大多数软件项目所需的数据规模都没有那么大，但是，越来越多的企业也开始采集并处理更多数据，以探究这些数据的用途。在此过程中，它们也同样会遇到关系型数据库与集群之间不协调的问题，这也是不争的事实。所以，随着谷歌和亚马逊所做的事情被逐步披露出来，大家也开始沿着与之相似的思路来研发数据库了，我们尤其想要研发那种适用于集群环境的数据库。尽管早些时候试图动摇关系型数据库垄断地位的那些技术都消失了，但是现在它却正面临来自集群领域的严峻挑战。

## 1.5 NoSQL 登场

说来也很有趣，“NoSQL”这个词首次出现是在 20 世纪 90 年代末，它是一个开源关系型数据库的名字 [Strozzi NoSQL]。该项目由 Carlo Strozzi 先生主导，这款数据库以 ASCII 文件存储数据表，每一个元组都占一行，其中的字段以制表符分隔。因为该

Ⓔ 该文档请参见：<http://research.google.com/archive/bigtable-osdi06.pdf>。——译者注

Ⓕ 该文档请参见：<http://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>。——译者注

数据库不以 SQL 为其查询语言，所以起名叫 NoSQL。可以通过 Shell 脚本<sup>①</sup>来操作这种数据库，并且能够使用常见的 UNIX 管道<sup>②</sup>将脚本与其他命令组合起来。除了术语上的巧合外，Strozzi 的 NoSQL 与我们在本书中所描述的数据库没有任何关系。

我们现在所说的“NoSQL”，源于 2009 年 6 月 11 日在旧金山举行的一场技术聚会（meetup）。这次聚会由伦敦的软件开发者 Johan Oskarsson 先生组织。BigTable 和 Dynamo 这两个例子催生了一大批项目，它们都在寻找一种数据存储方案。当时大家觉得，一场比较好的软件会议，应该要能讨论这些内容才对。当时 Johan 正在旧金山参加 Hadoop 峰会（Hadoop Summit），他很想再寻找一些这种类型的新数据库。由于时间紧迫，不可能逐个拜访其开发者，所以决定举办一个聚会，这样他们就可以聚在一起，并将作品展示给对其感兴趣的人了。

Johan 想给聚会起个名字，这个名字要适合做 Twitter 话题<sup>③</sup>才行，因而它必须简短、容易记忆，此外，要保证没有太多人在谷歌中搜索过它，这样根据此名称就能快速找到这个聚会了。他在“#cassandra IRC channel”<sup>④</sup>发问并得到了一些回答，最终选定了由 Eric Evans 先生所提出的名称“NoSQL”（他是 Rackspace 公司的一名开发人员，与提出“领域驱动设计”这一术语的 Eric Evans 先生<sup>⑤</sup>不是同一人）。虽说此名称有一些缺点，它比较消极，不能准确描述这些系统，不过，从 Twitter 话题的角度来看，它确实是个好名字。当时只是考虑为聚会起一个名字罢了，并没有要为整个技术发展趋势起名 [Oskarsson]。

“NoSQL”一词以野火燎原之势迅速流行起来，然而它始终没有一个严谨的定义。最初的 NoSQL 聚会 [NoSQL Meetup] 用它表示“开源分布式的非关系型数据库”，在聚会中，来自 Voldemort、Cassandra、Dynomite、HBase、Hypertable、CouchDB 和 MongoDB

① shell script，可以由操作系统的命令提示符来解析的脚本，通常可用于操作文件、执行程序、打印文本等。详情参见：[http://en.wikipedia.org/wiki/Shell\\_script](http://en.wikipedia.org/wiki/Shell_script)。——译者注

② pipeline，是由“标准输入”、“标准输出”、“标准错误”等标准流链接起来的一系列命令，每一个命令的输出都视为下一个命令的输入。详情参见：[http://zh.wikipedia.org/wiki/管道\\_\(Unix\)](http://zh.wikipedia.org/wiki/管道_(Unix))。——译者注

③ 原文是 hashtag，社交软件中为快速寻找某一相关主题所设的标签，一般格式是“#话题”或“#话题#”。——译者注

④ IRC 是一种即时聊天服务，其中的聊天室叫做“频道”（channel），频道名称以 # 开头。详情参见：<http://zh.wikipedia.org/wiki/IRC>。——译者注

⑤ 领域驱动设计（Domain-Driven Design）简称 DDD，是一种软件开发方法学。Eric Evans 著有同名书籍。详情参见：[http://en.wikipedia.org/wiki/Domain-driven\\_design](http://en.wikipedia.org/wiki/Domain-driven_design)。——译者注

的开发人员都发了言 [NoSQL Debrief], 然而, 该词的含义决不局限于上述 7 个项目的开发者所说的内容。这个词既没有公认的定义, 也没有权威人士为其下定义, 所以, 本书所能做的就只是讨论那些应该叫做 “NoSQL” 的数据库所共同具备的特征。

第一个较为明显的特征就是, NoSQL 数据库不使用 SQL。有一些 NoSQL 数据库确实带有查询语言, 而且与 SQL 类似。这么做是明智的, 因为大家学起来更容易。Cassandra 的 CQL 看上去和 SQL 非常像 (除了那些两者有差别的地方之外) [CQL]。但是到目前为止, 就算从广义的角度看, 也没有哪个 NoSQL 数据库真正实现了标准的 SQL 语言。如果某个已经发行的 NoSQL 数据库决定实现一套比较标准的 SQL, 那就有意思了。若是真出现了这种事情, 那么唯一可以肯定的就是, 它将会引发大量的争论。

这些数据库的另一个重要特征就是, 它们通常都是开源项目。虽然 NoSQL 这个术语也经常用在闭源系统中, 但是大家总认为 NoSQL 数据库就应该是开源的。

大多数 NoSQL 数据库的研发动机, 都是为了要在集群环境中运行, 而且最初那次聚会所讨论的那些数据库, 也是为了这一目的而开发的。这既影响了它们的数据模型, 也影响了这些数据库为了保持数据一致性所使用的方式。关系型数据库使用 ACID 事务 (参见 2.1.2 节) 来保持整个数据库的一致性, 而这种方式本身与集群环境相冲突, 所以, NoSQL 数据库为处理并发及分布问题提供了众多选项。

然而, 并非所有 NoSQL 数据库都是为了运行在集群上而设计的。图数据库就属于这种风格的 NoSQL 数据库, 它的分布模型与关系型数据库相似, 但其数据模型却与之不同。它的数据模型能更好地处理复杂的数据关系。

NoSQL 数据库通常是 21 世纪初的互联网企业而设, 所以一般来说只有这段时间内开发的数据库系统才有可能称为 NoSQL。这就排除一大批 2000 年之前业已存在的数据库, 更不用说前 Codd 时代<sup>⊖</sup>的那些数据库了。

操作 NoSQL 数据库不需要使用 “模式”, 这样不用事先修改结构定义, 即可自由添加字段了。这在处理不规则数据和自定义字段时非常有用。而在关系型数据库中,

---

⊖ Before Codd, 其中 Codd 是指英国计算机科学家埃德加·弗兰克·科德 (Edgar Frank “Ted” Codd, 1923—2003), 他为关系型数据库理论做出了奠基性的贡献, 提出了 “科德十二定律”。他于 1970 年左右在 IBM 工作期间, 首创了关系模型理论。详情参见: <http://zh.wikipedia.org/wiki/埃德加·科德> 及 <http://zh.wikipedia.org/wiki/科德十二定律>。本书作者将 Before Codd 一词缩写为 BC, 与 “公元前” (Before Christ) 一词的缩写恰好双关。——译者注

我们必须使用类似 `customField6` 这样的字段名，或是使用自定义的字段表才行。那样做既难于处理，也不易理解。

上面这些就是 NoSQL 数据库所具备的共同特征了。这些特点都不能用作“NoSQL”一词的明确定义，而且令人遗憾的是，这个词实际上也不可能出现统一的定义。然而，笔者就是按这些粗略的特性来写作本书的。我们之所以热衷于这个话题，是因为 NoSQL 的崛起扩大了数据存储方式的选择范围。由此导致的结果是，在通常所说的 NoSQL 数据库之外，还出现了很多种数据存储方式，我们希望有更多人能够接受这些新的存储方式，这其中也包括许多诞生于 NoSQL 之前的产品。但是，本书所能讨论的内容有限，所以笔者决定专注于传统意义上<sup>⊖</sup>的 NoSQL 数据库。

第一次听到“NoSQL”这个词时，大家肯定立马就要问它究竟代表什么：“是想对 SQL 说‘不’吗？”（a “no” to SQL?）许多人实际上都把 NoSQL 理解成了“不只是 SQL”（“Not Only SQL”），但这么解释有几个问题。首先，如果意思是“Not Only SQL”，那么缩写应为“NOSQL”，而大家写的却是“NoSQL”。另外，以“‘不只是’ SQL”一语来定义 NoSQL 数据库没多大意义，要是真能这么定义的话，那 Oracle 和 Postgres<sup>⊖</sup>也都符合此定义了，这就等于模糊了两类数据库之间的界限，并把所有的数据库都归到 NoSQL 名下了。

为了解决这个问题，笔者建议大家不必深究这个术语到底是哪些词的缩略语了，你只需知道它想表达的意思就行了（我们建议大家在解读大多数首字母缩略词时也采取此态度）。因此，当“NoSQL”一词修饰数据库时，它就泛指那些大多开发于 21 世纪初，基本上不使用 SQL，而且差不多都是开源的数据库。

将“NoSQL”解读为“不只是 SQL”也有合理之处，因为这样可以描述出很多人对数据库生态系统的展望。实际上，这种思维方式最有价值之处还在于：它不单单把 NoSQL 当成一项技术，更把它视为一场变革。笔者不认为关系型数据库会就此消失，它们依然是最常用的数据库形式，即便写了这本书，我们仍然推荐使用关系型数据库。关系型数据库为人熟知，运行得比较稳定，具备很多功能，而且广受支持——这些令人信服的理由促使大多数项目在选择数据库时都会考虑它。

---

⊖ 原文为 `noDefinition`，也就是“诞生于 21 世纪初的、开源的且不使用 SQL 的”数据库。——译者注

⊖ 一种开源的对象-关系型数据库，详情参见：<http://zh.wikipedia.org/wiki/PostgreSQL> 及 <http://www.postgresql.org/>。——译者注

现在的关系型数据库与原来不同了，它只是众多选项中的一个而已。这种观点通常称为“混合持久化”（polyglot persistence），也就是在不同的场景下使用不同的数据存储方式。我们不再因为别人都使用关系型数据库，而自己也跟风用它，相反，我们要从本质上理解待存储的数据，以及操作这些数据的方式。如此一来，许多公司就会根据不同的场景选用不同的数据存储技术。

为了使混合持久化技术能够顺利运作，笔者认为，企业还需要从集成数据库迁移到应用程序数据库。事实上，本书假定 NoSQL 数据库将作为应用程序数据库来用，我们觉得集成数据库一般不宜采用 NoSQL 技术。这不应视为 NoSQL 的缺陷，相反，即便你不使用 NoSQL，也应该考虑把原来存放在集成数据库中的数据转到应用程序数据库中，并将其封装起来，改用 Web 服务来在应用程序之间通信，这是个良好的迁移方向。

基于对 NoSQL 开发历史的理解，笔者专注于那些在集群上运行的“大数据”（big data）。我们认为它是推动数据库领域发展的一个关键因素，但这并不是许多项目团队选用 NoSQL 数据库的唯一原因。阻抗失谐这个老问题与上述因素同样重要。大数据给了我们一个机会，让大家重新思考自己需要何种数据存储技术。有些开发团队发现，就算他们不需要由单一服务器扩展到集群上，使用 NoSQL 数据库也能简化数据库访问，从而提高团队工作效率。

所以，在阅读本书后续章节时，要记住选用 NoSQL 的两个主要原因。一是待处理的数据量很大，或对数据访问的效率要求很高，从而必须将数据放在集群上；二是想采用一种更为方便的数据交互方式来提高应用程序开发效率。

## 1.6 要点

- 关系型数据库二十多年来一直很成功，它能够持久保存数据，控制并发访问，同时也提供了一套集成机制。
- 由于关系模型与内存中的数据结构不匹配，所以应用程序开发人员一直为这种阻抗失谐问题所困扰。
- 数据库领域的迁移趋势是：原来，各个应用程序都把同一份数据库当成共用的集成点（integration point），而现在，各个应用程序都会封装自己的数据库，并

通过服务彼此集成。

- 促使数据存储方式发生变化的重要原因是：需要在集群上运行大量数据，而关系型数据库不能在集群中高效运行。
- NoSQL 是偶然出现的新名词。它没有规范的定义，我们只能描述此类数据库所共有的特征。
- 各种 NoSQL 数据库的共同特性是
  - 不使用关系模型
  - 在集群中运行良好
  - 开源
  - 适用于 21 世纪的互联网公司
  - 无模式
- NoSQL 崛起所产生的重要影响就是混合持久化。

## 第 2 章

# 聚合数据模型

数据模型是认知和操作数据时所用的模型。对于使用数据库的人来说，数据模型描述了我们如何同数据库中的数据打交道。它与存储模型不同，后者描述了数据库内部存储及操作数据的机制。在理想情况下，用户应该感觉不到存储模型才对，然而实际应用中，我们还是得对其略知一二，这主要是为了实现良好的性能。

大家日常所说的“数据模型”一词，一般指应用程序的特定数据所具备的模型。开发者可能会指着一张数据库的“实体 - 关系图”（entity-relationship diagram），把这个包含客户、订单、产品等信息的东西叫做他们的数据模型。然而本书的“数据模型”通常表示数据库组织数据的方式，它的正式名称是“元模型”（metamodel）。

在过去的几十年中，关系型数据模型是占主导地位的数据模型，它看上去就是一组非常直观的表格，或者说，更像电子数据表<sup>⊖</sup>。每张表（table）有若干行（row），每行包含相关实体。这些实体通过列来描述，行列交汇处都有单一值（single value）。列可以引用同一张表内或不同表格中的其他列，从而把这些实体关联起来。（上文所说的“表”和“行”都不是正规术语，只是大家都这么称呼罢了。更正式的说法应该是“关系”及“元组”。）

NoSQL 技术与传统的关系型数据库相比，一个最明显的转变就是抛弃了关系模型。每种 NoSQL 解决方案的模型都不同，本书把 NoSQL 生态系统中广泛使用的模型分为四类：“键值”、“文档”、“列族”和“图”。前三类数据模型有一个共同特征，我们称其为“面向聚合”（aggregate orientation）。本章将解释面向聚合的含义及其对数据模型的意义。

---

<sup>⊖</sup> 原文是 spreadsheet，由 Calc、Excel 等办公处理软件所创建的电子表，也叫“试算表”。——译者注

## 2.1 聚合

关系模型把待存储的信息分隔成元组（行）。元组是种受限的数据结构：它只能包含一系列的值，因此不能在元组中嵌套另一个元组，也不能包含由值或元组所组成的列表。这种简单的数据结构支撑着关系模型：所有操作都必须以元组为目标，而且其返回值也必须是元组。

面向聚合所用的方式与之不同，我们通常操作数据时所用的单元，其结构都比元组集合复杂得多。如果能够以这种复杂的结构来存放列表或嵌套其他记录结构就好了。大家在后面的章节中将会看到，“键值数据库”、“文档数据库”、“列族数据库”都使用这种更为复杂的记录。然而，没有公认的术语来称呼这种复杂的记录，在本书中，把它叫做“聚合”（aggregate）。

聚合是“领域驱动设计”[Evans]中的术语。在领域驱动设计中，我们想把一组相互关联的对象视为一个整体单元来操作，而这个单元就叫聚合。在涉及数据操作与一致性管理时，更是如此。一般情况下，我们通过原子操作（atomic operation）更新聚合的值，并且在与数据存储通信时，也以聚合为单位。这个定义也非常符合“键值数据库”、“文档数据库”和“列族数据库”的工作方式。因为用聚合为单位来复制和分片显得比较自然，所以在集群中操作数据库时，还是使用聚合比较简单一些。此外，由于程序员经常通过聚合结构来操作数据，故而采用聚合也能让其工作更为轻松。

### 2.1.1 关系模型与聚合模型示例

现在可以用示例来帮大家理解我们所讲的内容。假设我们要建立一个电子商务网站，把商品通过网站直接卖给消费者，那么必须存储用户信息、商品目录（product catalog）、订单、收货地址（shipping address）、账单地址和付款方式等信息。这个应用场景，既可以用关系型数据模型建模，也可以用 NoSQL 数据模型建模，我们要比较两者的优劣。如果采用关系型数据库，那么就可以从图 2.1 所示的数据模型开始。

图 2.2 展示了该模型所用的一些范例数据。

既然大家都是使用关系模型的高手了，那一切就按老规矩办，这样各张表格间就不会出现重复数据了。我们也维护了表格间的“参照完整性”<sup>⊖</sup>。实际工作中的订单系统肯定更为复杂，不过对于本书来说，它倒是挺合适的。

⊖ 参照完整性（referential integrity）也叫引用完整性，是指某一系列里的每一个值，都能在本表或其他表格的另一列中找到，以便相互参引核对。详情参见：[http://en.wikipedia.org/wiki/Referential\\_integrity](http://en.wikipedia.org/wiki/Referential_integrity)。——译者注

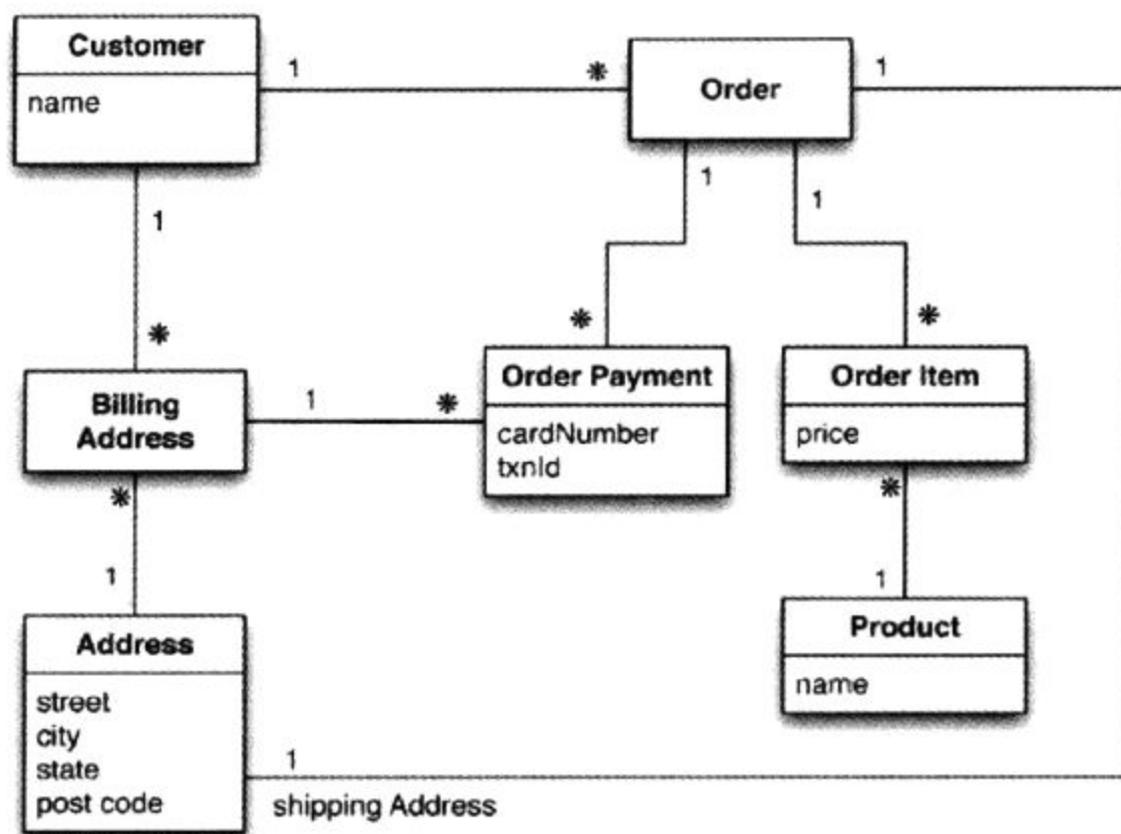


图 2.1 面向关系型数据库的数据模型（使用 [Fowler UML] 一书中的 UML 记法）

Customer	
Id	Name
1	Martin

Orders		
Id	CustomerId	ShippingAddressId
99	1	77

Product	
Id	Name
27	NoSQL Distilled

BillingAddress		
Id	CustomerId	AddressId
55	1	77

OrderItem			
Id	OrderId	ProductId	Price
100	99	27	32.45

Address	
Id	City
77	Chicago

OrderPayment				
Id	OrderId	CardNumber	BillingAddressId	txnId
33	99	1000-1000	55	abelif879rft

图 2.2 使用 RDBMS<sup>⊖</sup>数据模型的范例数据

现在我们来再看看，如果用面向聚合的思路来做，那么数据模型会是什么样子（如图 2.3 所示）。

⊖ RDBMS 是 Relational Database Management System 的简称，即“关系型数据库管理系统”，是管理“关系型数据库”的系统，然而该词有时也泛指“关系型数据库”本身。——译者注

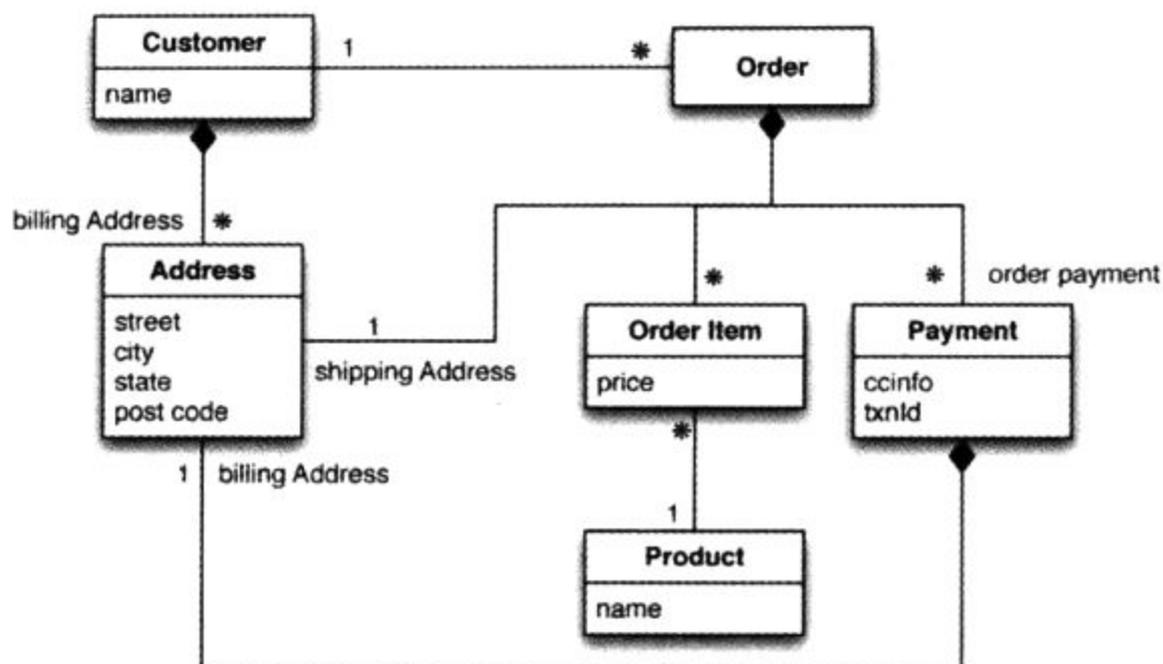


图 2.3 聚合数据模型

这次也要用一些范例数据，我们使用 JSON 格式来表示，因为它是 NoSQL 领域中常用的数据格式。

```

// in customers
{
  "id":1,
  "name":"Martin",
  "billingAddress":[{"city":"Chicago"}]
}

// in orders
{
  "id":99,
  "customerId":1,
  "orderItems":[
    {
      "productId":27,
      "price": 32.45,
      "productName": "NoSQL Distilled"
    }
  ],
  "shippingAddress":[{"city":"Chicago"}]
  "orderPayment":[
    {
      "ccinfo":"1000-1000-1000-1000",
      "txnId":"abelif879rft",
      "billingAddress": {"city": "Chicago"}
    }
  ],
}

```

这个模型有两个主要的聚合：客户（Customer）和订单（Order）。我们使用 UML 图里表示“组合”的黑色菱形标记，来表示各数据在聚合结构中的关系。客户数据包含一个账单地址（billing address）列表，订单数据包含订单项（order item）列表、收货地址

(shipping address) 及付款信息 (payment)，而付款信息本身又包含它所对应的账单地址。

同一个逻辑地址<sup>⊖</sup>在示例数据中出现了三次，但是此处我们不用 ID 来指代它，而是直接复制这个地址字符串。如果收货地址和账单地址都不会改变，那么这种做法就比较合适。在关系型数据库中，这种情况意味着 Address 表中 ID 为 77 的那行数据保持不变。若要改变某个地址，则需要在该表中创建新行。如果使用聚合模型，我们就可以把整个地址结构复制到所需的聚合模型中。

客户与订单之间的关联并不在某个聚合结构内部，它们算是两个聚合之间的关系。同理，订单项也可以同包含产品 (product) 信息的另一个聚合结构之间产生关联，只是此处我们没有深入描述后者罢了。与使用关系型数据库时要做的权衡一样，我们在这里也采用了一种非常规的办法，将产品名称直接写入订单项。这种做法在聚合模型中更常见，因为我们在数据交互时尽量减少所需访问的聚合个数。

此处最需要注意的事情，其实并不是划分聚合边界，而是规划数据访问方式。在制定应用程序的数据模型时，一定要考虑这个问题。实际上我们也可以另外一种方式画出聚合的边界，那就是把客户下的全部订单都放到客户聚合中（如图 2.4 所示）。

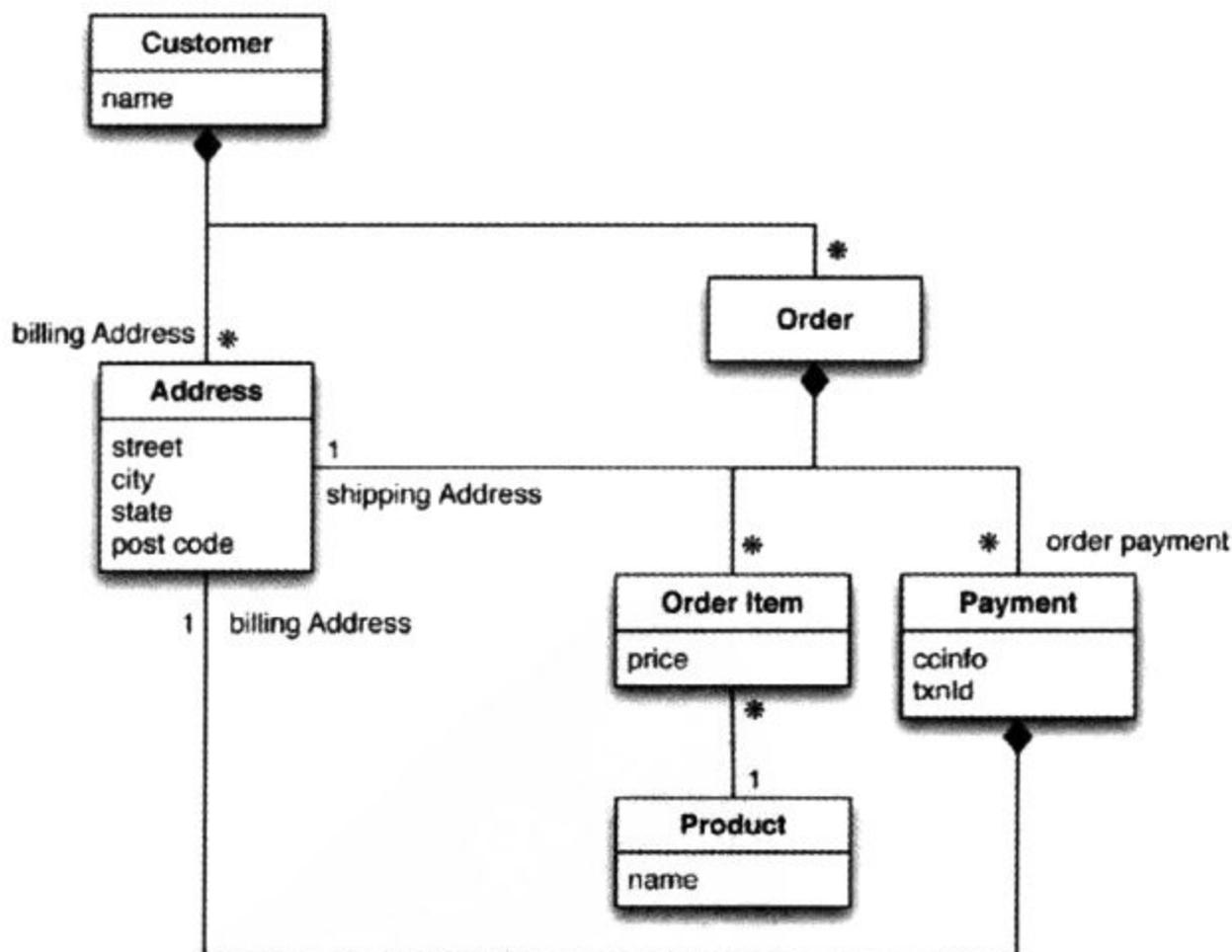


图 2.4 将涉及客户与客户订单的所有对象全部嵌入一个聚合结构中

⊖ 逻辑地址 (logical address)，这里指账单地址和收货地址所共用的地址“Chicago”。——译者注

如果使用上面讲的那种数据模型，那么 Customer 与 Order 的数据就该这么写：

```
// in customers
{
  "customer": {
    "id": 1,
    "name": "Martin",
    "billingAddress": [{"city": "Chicago"}],
    "orders": [
      {
        "id": 99,
        "customerId": 1,
        "orderItems": [
          {
            "productId": 27,
            "price": 32.45,
            "productName": "NoSQL Distilled"
          }
        ],
        "shippingAddress": [{"city": "Chicago"}]
      },
      {
        "orderPayment": [
          {
            "ccinfo": "1000-1000-1000-1000",
            "txnId": "abelif879rft",
            "billingAddress": {"city": "Chicago"}
          }
        ]
      }
    ]
  }
}
```

与建模过程中的大多数问题一样，对于如何划分聚合边界并没有标准答案，这完全取决于你打算怎么来操作数据。如果想一次性访问客户的全部订单，那就应该把它们放在一个大的聚合里面，反之，若是每次只想专门处理一笔订单，则应将“客户”和“订单”划为两个彼此分离的聚合。当然，这得根据具体情况来判断。有些应用程序会有自己的偏好，甚至同一个系统中也会存在不同的建模方式。正因为此，所以在各种不同的聚合划分方式中，很多人都不会有特定的倾向。

### 2.1.2 面向聚合的影响

虽然关系映射能够很好地捕捉各种数据元素及其关系，但是它却没有“聚合实体”（aggregate entity）这一概念。在我们的领域语言（domain language）中，可以说订单由订单项、收货地址及付款信息组成。在关系模型中，可以用“外键”（foreign key）来表达这种关系，但是那样做无法区分某个关系是否表示聚合。因此，数据库无法使用聚合结构来帮助其存储与分布数据。

多种数据建模都提供了其标记聚合（aggregate）结构或组合（composite）结构的方式。然而问题是，建模者很少会提供一种语义（semantic）来描述各类聚合关系之

间的区别。就算提供了，这些建模技术所用的语义也各不相同。如果使用面向聚合的数据库，那么通过考虑与数据存储交互时所用的单位，我们就能得出一种更为清晰的语义了。然而，它并不是数据的逻辑属性，它只描述了应用程序使用数据的方式而已，而这一事项通常不属于数据建模的范畴。

关系型数据库的数据模型中，没有“聚合”这一概念，因此我们称之为“聚合无知”（aggregate-ignorant）。NoSQL 领域中的“图数据库”也是聚合无知的。这一特征并不是坏事。聚合的边界一般都很难正确划分出来，当不同场景要使用同一份数据时，更是如此。在客户下单并核查订单，以及零售商处理订单时，将订单视为一个聚合结构就比较合适。然而，如果零售商要分析过去几个月的产品销售情况，那么把订单做成一个聚合结构反而麻烦了。要取得商品销售记录，就必须深挖数据库中的每一个聚合。因此，对某些数据交互有用的聚合结构，可能会阻碍另一些数据交互。若是采用“聚合无知模型”，那么很容易就能以不同方式来查看数据，因此，在操作数据时，如果没有一种占主导地位的结构，那么选用此模型效果会更好。

选用面向聚合模型的决定性因素，就在于它非常适合在集群上运行。大家应该还记得，这正是 NoSQL 崛起的杀手锏。在集群上运行时，我们需要把采集数据时所需的节点数降至最小。如果在数据库中明确包含聚合结构，那么它就可以根据这一重要信息，知道哪些数据需要一起操作了，而且这些数据应该放在同一个节点中。

聚合对于事务处理有一个重要影响。关系型数据库允许把任意表格中的任意行组合起来，放在一个事务中操作。这种事务就叫“**ACID 事务**”：它具有原子性（Atomic）、一致性（Consistent）、隔离性（Isolated）和持久性（Durable）。ACID 这个首字母缩略词有点做作，其实真正的重点是原子性：在单一操作中更新跨越多张表的数个行。该操作要么完全成功，要么彻底失败，而且并发执行的多个操作之间是彼此隔离的，因此它们不可能看到某个尚未全部完成的更新操作。

经常会听人说：NoSQL 数据库不支持 ACID 事务，因而其一致性会受损。这一论述简化得有些过头了。通常情况下，面向聚合的数据库确实不支持跨越多个聚合的 ACID 事务。取而代之的是，它每次只能在一个聚合结构上执行原子操作。也就是说，如果我们想以原子方式操作多个聚合，那么就必须自己组织应用程序的代码。在实际应用中，大多数原子操作都可以局限于某个聚合结构内部，而且，在将数据划分为聚合时，这也是要考虑的因素之一。我们还应该记住，图数据库和其他一些“聚合无知

式数据库”都支持与关系型数据库类似的 ACID 操作。最为重要的是，“一致性”这个话题相当复杂，数据库支持不支持 ACID，只是该问题的一个方面而已。本书第 5 章将详细研究它。

## 2.2 键值数据模型与文档数据模型

早前提到过，键值数据库和文档数据库都特别面向聚合。这么说的意思是，这些数据库主要是通过聚合来构建的。这两类数据库都包含大量聚合，每个聚合中都有一个获取数据所用的键或 ID。

两种模型的区别是：键值数据库的聚合不透明<sup>Ⓐ</sup>，只包含一些没有太多意义的大块信息；与此相反，在文档数据库的聚合中，可以看到其结构。不透明的优势在于，聚合中可以存储任意数据。数据库可能会限制聚合的总大小，但除此之外，其他方面都很随意。文档数据库则要限制其中存放的内容，它定义了其允许的结构与数据类型，而这样做的好处是，能够更加灵活地访问数据。

在键值数据库中，要访问聚合内容，只能通过键来查找。而使用文档数据库时，则可以用聚合中的字段查询。我们可以只获取一部分聚合，而不用获取全部内容，此外，数据库还可以按照聚合内容创建索引。

实际上，键值数据库和文档数据库之间的界限有点模糊。经常有人将 ID 字段放在文档数据库中，用它做“键值式查询”，而归入键值型的那些数据库，也可能允许在聚合中使用略带含义的数据结构。例如，Riak 就可以在聚合中添加元数据（metadata），以便实现索引与聚合间关联（interaggregate link），而 Redis<sup>Ⓑ</sup>可以将聚合分解成列表或集合。若想支持查询功能，可以把 Solr 这样的搜索工具集成进去。比方说，Riak 就包含一种搜索机制，它能够使用类似 Solr 的工具，在 JSON 或 XML 格式的聚合结构中搜索。

尽管界限模糊，但两者大体上还是有区别的。在键值数据库中，基本上都是通过键来搜索聚合内容，而在文档数据库中，我们提交的查询关键词往往基于文档的内部结构。它也许会键，但是更有可能是其他东西。

---

Ⓐ 这里原文是 opaque，计算术语中，“不透明”一词形容某个数据结构不需知道其内部实现细节即可为外部程序使用。——译者注

Ⓑ 使用 ANSI C 编写的开源键值型数据库，提供多种语言 API。详情参见：<http://redis.io/>。——译者注

## 2.3 列族存储

早期有影响力的一种 NoSQL 数据库是谷歌的 BigTable [Chang etc.]。这个名字让人以为它是那种带有稀疏列的无模式表结构。正如稍后将会讲到的那样，将此结构视为表格是没有多大意义的，不如把它看成“两级映射”（two-level map）更好。不过，无论你怎样看待这种结构，此模型都影响了后来的 HBase 和 Cassandra 等数据库。

这些采用“大表格式数据模型”（bigtable-style data model）的数据库通常称为“列存储（column store）数据库”，但是这个词以前指的却是另外一个意思。前 NoSQL 时代的“列存储数据库”是指 C-Store [C-Store] 等产品，它们与 SQL 及关系模型结合得很好。新旧两种含义的区别在于，数据的物理存储方式不同。大部分数据库都以行为单元存储数据，尤其是在需要提高写入性能の場合更是如此。然而，有些情况下写入操作执行得很少，但是经常需要一次读取若干行中的很多列。在这种情况下，将所有行的某一组列作为基本数据存储单元，效果会更好。“列存储数据库”一词正是由此得名。

Bigtable 和它的后继者都遵循了“以一组列（也就是列族）来存储”的概念，不过有一部分数据库与 C-Store 等一并放弃了关系模型和 SQL。本书将此类数据库称为“列族数据库”（column-family database）。

理解列族模型的最好方式也许就是将其视为两级聚合结构（two-level aggregate structure）。与“键值存储”<sup>⊖</sup>相同，第一个键通常代表行标识符，可以用它来获取想要的聚合。列族结构与“键值存储”的区别在于，其“行聚合”（row aggregate）本身又是一个映射，其中包含一些更为详细的值。这些“二级值”（second-level value）就叫做“列”。与整体访问某行数据一样，我们也可以操作特定的列。因此，可以用 `get('1234', 'name')` 来获取图 2.5 中那个客户的名字。

列族数据库将列组织为列族。每一列都必须是某个列族的一部分，而且访问数据的单元也得是列。这样设计的前提是，某个列族中的数据经常需要一起访问。

---

<sup>⊖</sup> 在本书语境中，“键值存储”（key-value store）通常与“键值数据库”（key-value database）一词互用，“文档存储”（document store）也与“文档数据库”（document database）一词互用，“列族数据库”与“图数据库”亦然。这是同一概念的两种不同表述形式，前者侧重存储方式，后者强调数据库类型。在不致混淆时，译文也将其视为同一概念。——译者注

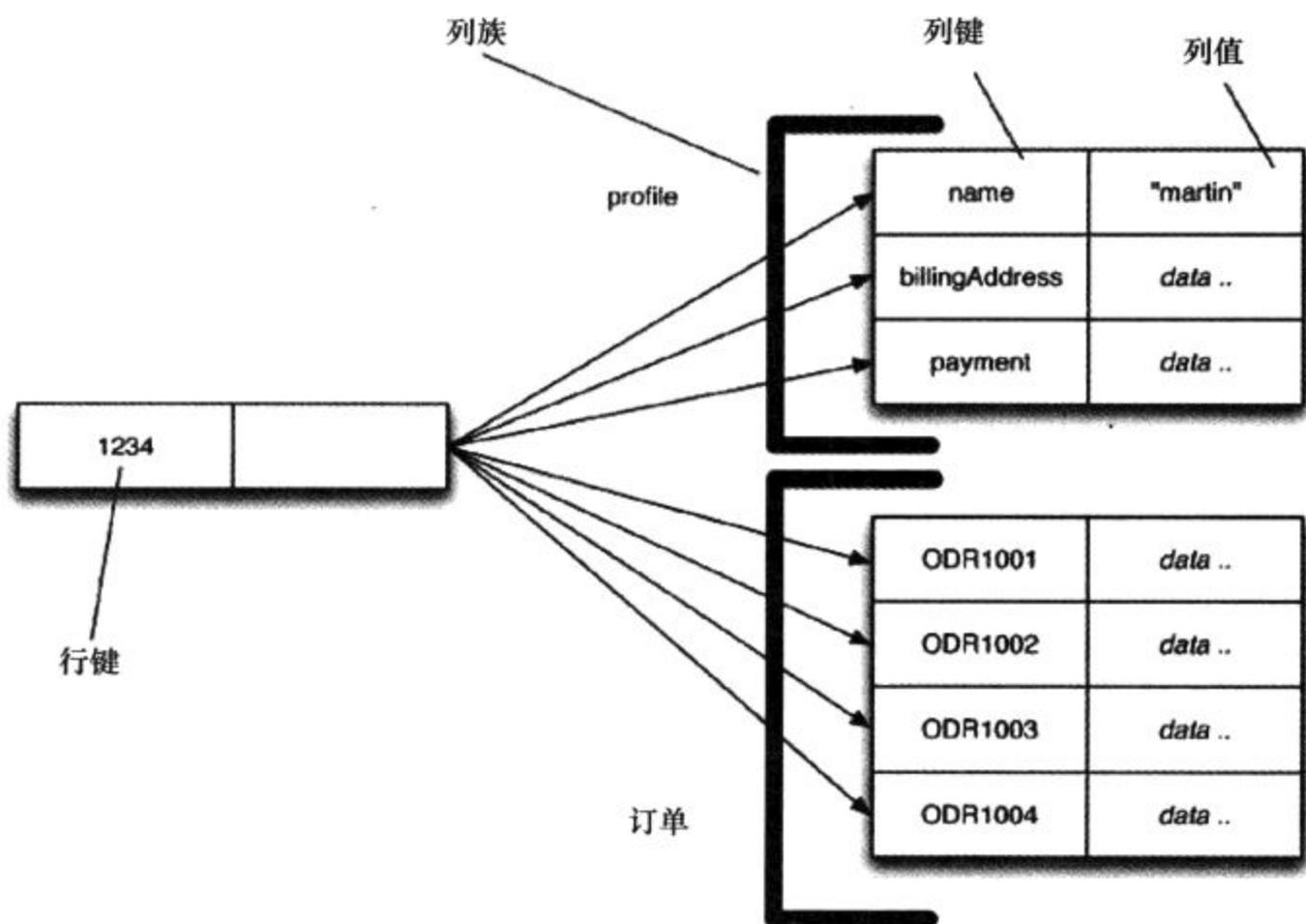


图 2.5 以列族结构表示客户信息

于是，我们也得出了两种数据组织方式。

- 面向行 (row-oriented): 每一行都是一个聚合 (例如 ID 为 1234 的顾客就是一个聚合), 该聚合内部存有一些包含有用数据块 (客户信息、订单记录) 的列族。
- 面向列 (column-oriented): 每个列族都定义了一种记录类型 (例如客户信息), 其中每行都表示一条记录<sup>⊖</sup>。你可以将数据库中的大“行”理解为列族中每一个短行记录的串接。

后者反映了“列”在列族数据库中的重要性。由于数据库了解这些数据通常的分组方式, 所以它可在存储及访问时利用此信息。即使文档数据库声明了某种结构, 每个文档也依然被视作独立单元。“列族”体现了“列族数据库”二维映射这一特点。

这套术语是由谷歌 BigTable 和 HBase 创立的, 但是 Cassandra 处理数据的方式与之略有不同。Cassandra 中的“行”只能出现在一个列族中, 然而列族可以包含“超列” (supercolumn), 也就是列里可以再嵌套列。Cassandra 中的超级列恰好对应于经典 Bigtable 里面的列族。

将列簇视为表格, 仍然让人困惑。可以在任意行中添加任何列, 并且行中可以有

<sup>⊖</sup> 在图 2.5 右上方和右下方的表格中, 水平方向表示“列” (column)。——译者注

差异很大的“列键”(column key)。在常规数据库操作中，一般都是向行中添加新的列，很少会定义新的列族，因为那样做可能需要先停止数据库服务才行。

图 2.5 所举范例，演示了列族数据库的另一个方面，习惯使用关系型表格的人可能对“orders”这种列族形式不太熟悉。由于列族中可以随意添加列，所以在建模项目清单时，可以把其中每个项目都表示为单独的列。如果把列族看作一张表，那这种行为就显得非常怪异，反之，若是把列族中的行视为聚合，那这么做就顺理成章了。Cassandra 有“宽”(wide)、“瘦”(skinny)两个术语。“瘦行”(skinny row)的列不多，但是很多行都会出现相同的列。在这种情况下，列族定义了一种记录类型，每一行都是一条记录，每一列都是一个字段。而“宽行”(wide row)则有许多列(可能有几千个)，然而各个行中的列差别很大。“宽列族”(wide column family)模型反映了一个列表，其中每列都是列表中的一个元素。

“宽列族”可以定义各列之间的排列顺序，这样就可以根据订单的键来访问订单了，而且还可以用键来获取某个范围内的多个订单。若使用 ID 作为订单的键，这么做也许意义不大，然而订单的键如果同时包含日期和 ID (例如 20111027-1001)，那么排序功能就很有用了。

虽然将列族根据其性质区分为“宽”、“瘦”两类比较有用，然而从技术角度来讲，列族也不是不能同时包含“字段式的列”(field-like column)与“列表式的列”(list-like column)，只是这样做会给排序带来麻烦。

## 2.4 面向聚合数据库总结

通过上述知识，读者应该能大概了解三种不同风格的面向聚合数据模型及其差别。

三者的共同点是，它们都使用聚合这一概念，而且聚合中都有一个可以查找其内容的索引键。在集群上运行时，聚合是中心环节，因为数据库必须保证将聚合内的数据存放在同一个节点上。聚合还是“更新”操作的最小数据单位(atomic unit)，对事务控制来说，以聚合为操作单元，其大小正合适。

在聚合的大概念下，三者有一些差别。键值数据模型将聚合看作不透明的整体，这意味着只能根据键来查出整个聚合，而不能仅仅查询或获取其中的一部分。

文档模型的聚合对数据库透明，于是就可以只查询并获取其中一部分数据了，不过，由于文档没有模式，因此在想优化存储并获取聚合中的部分内容时，数据库不太

好调整文档结构。

列族模型把聚合分为列族，让数据库将其视为行聚合内的一个数据单元。此类聚合的结构有某种限制，但是数据库可利用此种结构的优点来提高其易访问性。

## 2.5 延伸阅读

聚合的通用概念大多也适用于关系型数据库，更多内容请参考 [Evans]。在领域驱动设计社区中，可以获得很多关于聚合的详细信息，最新消息一般发布在网站 <http://domaindrivendesign.org> 上。

## 2.6 要点

- 聚合是作为交互单元的数据集合。数据库中的 ACID 操作以聚合为界。
- 键值数据库、文档数据库、列族数据库都属于面向聚合的数据库。
- 聚合使数据库在集群上管理数据存储更为方便。
- 如果数据交互大多在同一聚合内执行，则应使用面向聚合的数据库；若交互操作需要使用多种不同格式的数据，那么最好选用“聚合无知式数据库”。

## 第 3 章

# 数据模型详解

目前为止，我们已经知道了大多数 NoSQL 数据库的关键特征在于它们都使用聚合，而各种面向聚合的数据库对聚合的建模方式不同。聚合是 NoSQL 的核心内容，不过关于数据模型，还有很多内容要讲。本章就来深入研究这些概念。

## 3.1 关系

聚合的有用之处在于它可以把经常访问的数据存放在一起。但在有些情况下，我们需要以不同方式来访问相互关联的数据。考虑一下客户和其全部订单之间的关系。有些应用程序在访问客户数据时想要随时查询其订单历史记录，若是把客户和其订单记录放入一个聚合中，那对这种程序来说就很方便了。然而，另外一些程序想要分别处理订单，所以它们在建模时，要把订单存放到单独的聚合里面。

这种情况下，我们会把订单和客户放在两个聚合中，但是想在它们之间设定某种关系，以便能根据订单查出客户数据。要提供这种关联，最简单的办法就是把客户 ID 嵌入订单的聚合数据中。这样的话，如果需要获取客户记录中的数据，你可以先读取订单，并从中查出客户 ID，然后让数据库根据此 ID 查出客户数据。这种方式可行，而且在许多场合下还不错。只是数据库并不了解数据之间的关系。这种关系很重要，因为有时如果数据库能知道数据间的关联，那就会非常有用。

因此，许多数据库都提供了描述这种关系的手段，就连某些“键值数据库”也不例外。“文档数据库”可以访问聚合的内容，并据此编订索引，用户也可以查询这些内容。Riak 是“键值数据库”，我们可以把链接信息放在其元数据中，这样它就支持“局部检索”（partial retrieval）和“链接遍历”（link-walking）了。

如果两个聚合之间有了关系，那么一个重要问题就是如何更新其数据。面向聚合数据库获取数据时以聚合为单元，因此，它只能保证单一聚合内部内容的原子性。如果一次更新多个聚合，那就必须设法应对中途发生的错误。若要在关系型数据库中同时修改多条记录，你可以把它们放在一个事务中执行，这样在改动多行内容时，还能保证本次操作的 ACID 属性<sup>⊖</sup>。

所有这些都意味着，面向聚合数据库在操作多个聚合时显得相当笨拙。本章稍后就会讲到，有很多种办法能够应对此情况，不过从根本上来说，它处理这一问题还是不够灵巧。

如果待处理的数据中存在大量关系，那么这意味着你更应该选用关系型数据库，而不是 NoSQL 型数据库。虽说面向聚合的数据库处理复杂的关系时效果不好，但是大家也要明白，关系型数据库应对这一问题时表现也不尽如人意。查询时可将多个表用

---

⊖ 意思就是：能保证这次操作是原子的、一致的、隔离的、持久的。——译者注

SQL 的 JOIN 谓词连接起来<sup>⊖</sup>，然而那么做很快就会让代码变糟：随着 JOIN 子句数量越来越多，SQL 语句更加难写，而且查询效率也更低了。

此时正好可以介绍另一类数据库了，大家通常将其归入 NoSQL 阵营。

## 3.2 图数据库

图数据库（Graph Database）是 NoSQL 世界中的怪客。因为想要在集群环境上运行，所以很多 NoSQL 数据库都因之而生，它们使用面向聚合的模型来描述一些具备简单关联的大型记录组。图数据库的催生动机与之不同，它是为解决关系型数据库的另外一项缺点而设计的，因此其数据模型也与其他 NoSQL 数据库相反。这种数据模型适合处理像图 3.1 这样相互关系比较复杂的一小组记录。

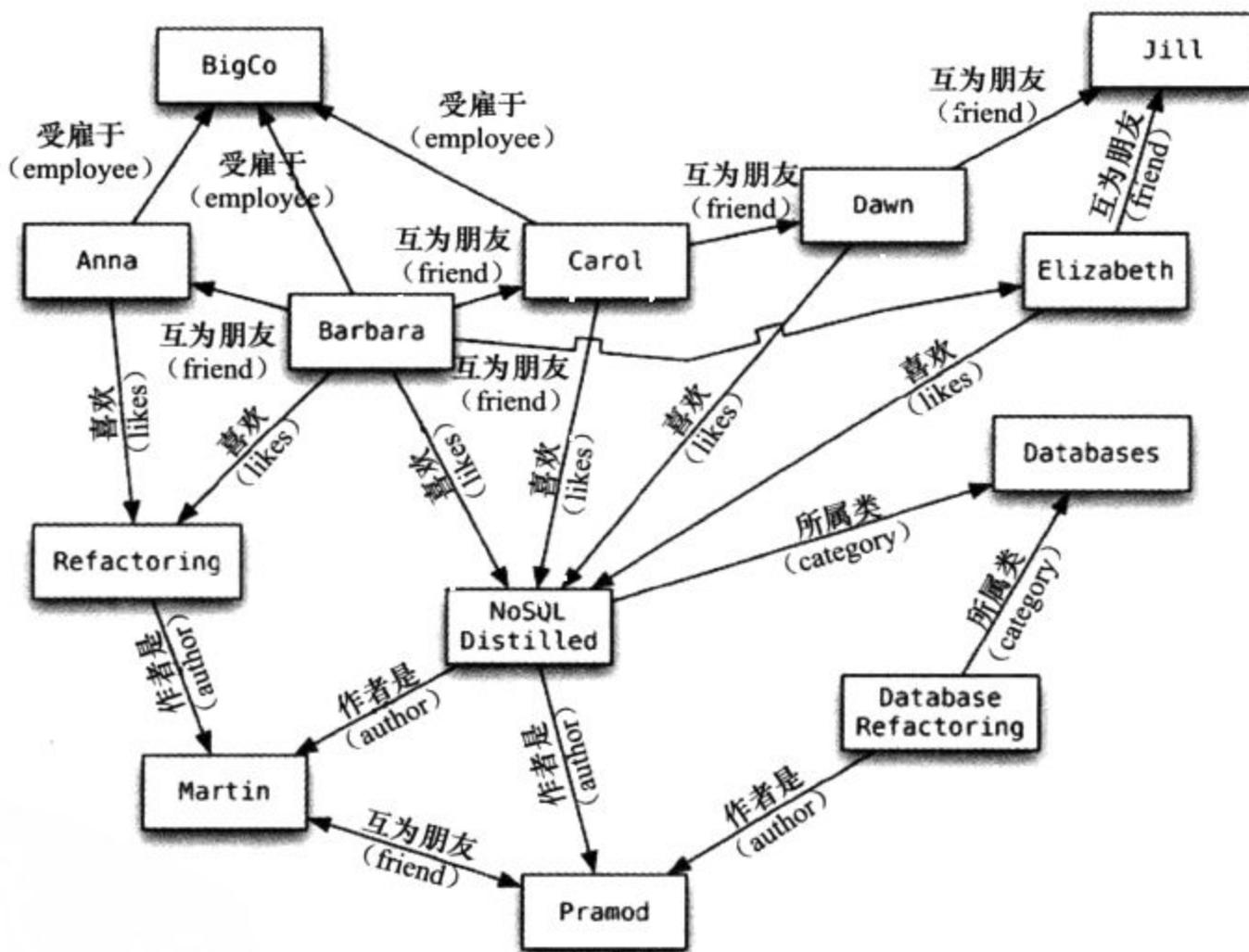


图 3.1 “图结构”示例

本语境下的“图”（Graph），既不是条形图（bar chart），也不是直方图（histogram），而是一种图形数据结构（graph data structure），其中含有连接节点（node）的边（edge）。

<sup>⊖</sup> JOIN 的详细用法参见：[http://zh.wikipedia.org/wiki/连接\\_\(SQL\)](http://zh.wikipedia.org/wiki/连接_(SQL))。——译者注

在图 3.1 所示的信息网中，节点都很简单（只包含名称），然而节点间的互连结构却很丰富。有了此结构，我们就可以像这样来查询：“找出数据库方面的书，其作者必须是我的某位朋友所喜欢的人。”

图数据库专门擅长捕捉此类信息，不过它处理的数据规模，比这种我们能一眼望尽的图要大很多。在捕获社交网络、产品偏好（product preference）、资格认定规则（eligibility rule）等包含复杂关系的数据时，使用图数据库较为理想。

图数据库的基本数据模型很简单：由边（或称“弧”，arc）连接而成的若干节点。除了这个共同的基本特征外，图数据库所用的数据模型有很多种变化，尤其是在节点和边的数据存储机制上。我们举几个例子，快速了解一下现有各种数据库在此问题上的方案。FlockDB<sup>①</sup>只存储节点与边，没有用于存储附加属性的机制；Neo4J 可以用无模式的方式将 Java 对象作为属性，附加到节点与边之中（参见 11.2 节）；Infinite Graph<sup>②</sup>可以把 Java 对象作为其内建类型的子类对象，存储成节点与边。

以节点与边把图结构搭建好之后，就可以用专门为“图”而设计的查询操作来搜寻图数据库的网络了。这就是图数据库和关系型数据库的重要差别。尽管关系型数据库也可以通过外键来实现这种关系，但是在各种关系中导览所需的 JOIN 语句非常耗时。这也就是说，用它来操作内部相互关系比较紧密的数据模型，其效率通常较差。而在图数据库中遍历关系，则非常迅速。很大一部分原因是由于图数据库会多花一些时间用于插入关系数据，以此来缩短遍历关系时所需的时间。在那种查询效率比插入数据的速度更为重要的场合，这种权衡就非常划算了。

大部分情况下，我们都是沿网络的边来浏览数据库，以查找所需数据。待查询的通常是“告诉我 Anna 和 Barbara 都喜欢的东西”这种问题。然而查询需要有起点，所以数据库通常会用某些节点的 ID 等属性来编制索引。这样一来，你可能先得根据 ID 查到节点（例如，先找到名为“Anna”和“Barbara”的人），然后再沿着边继续往下搜寻。使用图数据库时，大部分操作都应该是浏览各种关系才对。

图数据库与面向聚合数据库的明显差异，就在于其重视数据间的“关系”。这种数据模型上的差异也导致了其他方面的一些区别。这种图形数据库通常运行在单一的服务器上，而不是分布于集群中。为了使数据保持一致，ACID 事务需要涵盖多个节点与

① 一种开源分布式图数据库，其官方网站是：<https://github.com/twitter/flockdb>。——译者注

② 其官方网站是：<http://www.objectivity.com/infinitegraph>。——译者注

边。它和面向聚合数据库仅有的相似之处在于，二者都不使用关系模型，而且它们受人关注的时间点也和 NoSQL 领域中的其他新技术大致一样。

### 3.3 无模式数据库

各种形式的 NoSQL 数据库有个共同点，那就是它们都没有模式。若要在关系型数据库中存储数据，首先必须定义“模式”，也就是用一种预定义结构向数据库说明：要有哪些表格，表中有哪些列，每一列都存放何种类型的数据。必须先定义好模式，然后才能存放数据。

相比之下，NoSQL 数据库的数据存储就比较随意了。“键值数据库”可以把任何数据存放在一个“键”的名下。“文档数据库”实际上也如此，因为它对所存储的文档结构没有限制。在列族数据库中，任意列里面都可以随意存放数据。你可以在图数据库中新增边，也可以随意向节点和边中添加属性。

无模式数据库的倡导者们很享受它所带来的自由与灵活。如果用了模式，那么就必须提前指明你要存储的数据，然而这一点却比较难办。摆脱模式的制约后，就可以轻易存储所需数据了，于是我们很容易就能根据项目的进展情况来修改原有的数据存储方式，一旦发现了新东西，只要把它们加入数据库中就好。此外，若是发现某些东西已经没用了，那么不再存储它们就行了。在使用模式的关系型数据库中，如果删除了某列，那么你恐怕得担心此操作会不会导致旧数据丢失。

除了更改数据方面的差别外，无模式数据库也更容易处理“格式不一致的数据”（nonuniform data），也就是那种每条记录都拥有不同字段集（set of field）的数据。“模式”会将表内每一行的数据类型强行统一，若不同行所存放的数据类型不同，那这么做就很别扭。要么得分别用很多列来存放这些数据，而且把用不到的字段值填成 null（这就成了“稀疏表”，sparse table），要么就要使用类似 custom column 4 这样没有意义的列类型。而无模式表则没有这么麻烦，每条记录只要包含其需要的数据即可，不用再担心上面的问题了。

无模式数据库很吸引人，而且确实能避免使用“固定模式数据库”（fixed-schema database）时所面临的诸多麻烦，不过，它自身也存在一些问题。若存储数据就是为了将其显示成一系列“字段名：字段值”（fieldName：value）格式的简单报表，那“模式”的确是个障碍。可是，通常我们在处理数据时所要完成的任务并不止于此，而且处理

数据的程序需要知道存放账单地址的字段叫做 `billingAddress` 而非 `addressForBilling`, `quality` 字段应该包含整数“5”而非单词“five”。

在编写数据访问程序时，必须面对一个关键问题：尽管有时不甚方便，但程序通常要依赖于某种形式的“隐含模式”（implicit schema）。除非只需要执行下面这种极为简单的逻辑

```
//pseudo code
foreach (Record r in records) {
  foreach (Field f in r.fields) {
    print (f.name, f.value)
  }
}
```

否则，程序必须假定表中存在某些特定的字段名，这些字段中包含带有一定意义的数据，而且还要假定该字段存有某种类型的数据。程序与人类不同，它们不能在看到“qty”后立刻推断出它与“quality”的意义一样，至少在我们没有专门为其编写特定处理代码时，它不行。所以说，不管数据库“无模式”到何种地步，总会存在“隐含模式”。它是指在编写数据操作代码时，对数据结构所做的一系列假设。

应用程序代码中的隐含模式会带来一些问题。它意味着，要想理解数据库中存放的数据，必须深入研究应用程序的代码才行。若代码结构非常好，那么你就可以根据它明确推断出数据的模式了，然而这一点却无法保证，因为它完全取决于应用程序的代码是否清晰。此外，无模式数据库感知不到模式，所以它无法用模式来提升存储与获取数据的效率，它也无法自行验证数据，以防止多个应用程序以不一致的方式操作其数据。

上述问题就是关系型数据库采用固定模式的原因，而且过去的数据库基本都使用固定数据模式，也正是基于此种考量。“模式”有其价值，而 NoSQL 数据库弃用模式，真是个相当令人吃惊的决定。

从本质上说，无模式数据库是把模式交由访问其数据的应用程序代码来处理。如果由多位开发者制作的不同程序要访问相同的数据库，那就麻烦了。有几个办法能缓解此问题。一个办法就是，将所有数据库互动操作封装成独立的应用程序，并通过 Web 服务将它与其他应用程序集成。当前很多开发者都通过 Web 服务集成应用程序，该方法非常适合此类开发场景。还有一种办法是，在聚合中为不同应用程序明确划分出不同区域。在文档数据库中，可以把文档分成不同的区段（section）；在列族数据库

中，可以把不同的列族分给不同的应用程序。

尽管 NoSQL 支持者经常批评关系型数据库，说它必须预定义模式，而且其模式也很不灵活，但事实并非如此。关系型数据库的模式随时可以通过标准 SQL 命令修改。在必要时，可以立即添加新列，以存储“类型不一致的数据”。我们只是很少碰到这种情况罢了，但如果真的遇上了，那此方法完全能应付。然而，在大多数情况下，如果你发现待存储的数据类型不统一，那么应该优选无模式数据库。

无模式数据库一直深远地影响着数据库的结构变更，尤其是存储格式不一致的数据时更是如此。关系型数据库的模式也可以用可控的方式改变，只是其运用范围没有理想中那么广罢了。同理，也可以控制无模式数据库存储数据的方式，让访问新、旧数据都比较容易。此外，“无模式”的灵活性仅限于聚合内部，如果改动了聚合边界，那么其数据迁移工作与关系型数据库一样，都非常复杂。本书稍后将谈到数据库的迁移问题（参见第 12 章）。

### 3.4 物化视图

在谈到面向聚合的数据模型时，我们强调了它的优点。如果想访问订单数据，那么把一份订单内的所有数据都放在一个聚合中比较合适，这样它就能作为一个数据单元来存储与访问了。但是与之相应，面向聚合也有一个缺点：如果产品经理要知道过去几周内某个产品的销售量该怎么办？在此情况下，面向聚合反而帮了倒忙。你可能得从数据库中读出每份订单，才能得出这个数据。针对产品编订索引，可以缓解此问题，不过这种做法仍然与聚合结构相悖。

关系型数据库在这个问题上就体现出它的优势了。因为不存在聚合结构，所以它们可以用不同方式访问数据。此外，它们还提供了“视图”这一便利机制，其展示数据的角度与数据库存储数据的方式有所不同。视图就好比一张关系表（relational table，也就是一个“关系”），只不过它是通过基表（base table）算出来的。在访问视图时，数据库会算出视图中的数据，这种封装形式很方便。

有了视图这种机制，客户端就不用再操心它访问的数据到底是基本数据（base data）还是派生数据（derived data）了，不过，生成某些视图需要大量的计算工作，这一事实不容回避。于是，“物化视图”（materialized view）就应运而生了，这是一种预先算好并缓存在磁盘中的视图。如果数据读取非常频繁，而访问者又不介意略显陈旧

的数据，那么使用物化视图效率比较高。

虽说 NoSQL 数据库没有视图，但是它们可以预先计算查询操作的结果，并将其缓存起来。NoSQL 借用“物化视图”这个术语来表述此操作。与关系型数据库相比，面向聚合的数据库更强调这个问题，因为大多数应用程序都要处理某种与聚合结构不甚相符的查询操作。（NoSQL 数据库通常以“映射化简”来计算物化视图，第 7 章会讲解此内容。）

粗略地讲，构建物化视图有两种办法。第一种是比较积极的办法，也就是一旦基础数据（base data）有变动，那么立即更新物化视图。在这种情况下，只要向数据库中加入一份订单，那么保存每件产品销售记录的那个聚合就会即刻更新。如果读取物化视图的次数远比写入次数多，而且想获得更为及时的数据，那么这种方法比较合适。此时，使用应用程序数据库（见 1.3 节）更容易些，因为它能够保证物化视图会随着基础数据而更新。

若是不想在每次改变基础数据时都去更新物化视图，那么可以定期通过批处理操作来更新它。你需要理解业务需求，据此判断物化视图可以使用多久以前的数据。

可以在数据库之外构建物化视图：先读出数据，计算好视图内容，然后将其存回数据库。一般来说，数据库都可以自己构建物化视图。在这种情况下，你告诉数据库需要做何种计算，然后数据库会在需要时根据配置好的参数自行运算。如果想以“增量式映射化简”（incremental map-reduce，参见 7.3.2 节）来积极更新视图内容，那么这样做非常方便。

物化视图也可以在同一个聚合内使用。订单文档中，也许会含有描述其汇总信息的“订单汇总”元素，这样的话，若要查询某订单的汇总信息，就不需要传输整份文档了。根据不同列族来创建物化视图，是列族数据库的常见功能。这么做的一个好处是，可以在同一个原子操作内更新物化视图。

### 3.5 构建数据存取模型

早前说过，构建数据聚合模型时，既要考虑数据的读取方式，也要考虑模型对这些同聚合相关联的数据会产生何种副作用（side effect）。

首先来看这种模型：将所有客户数据都嵌入一个“键值存储”中（如图 3.2 所示）。

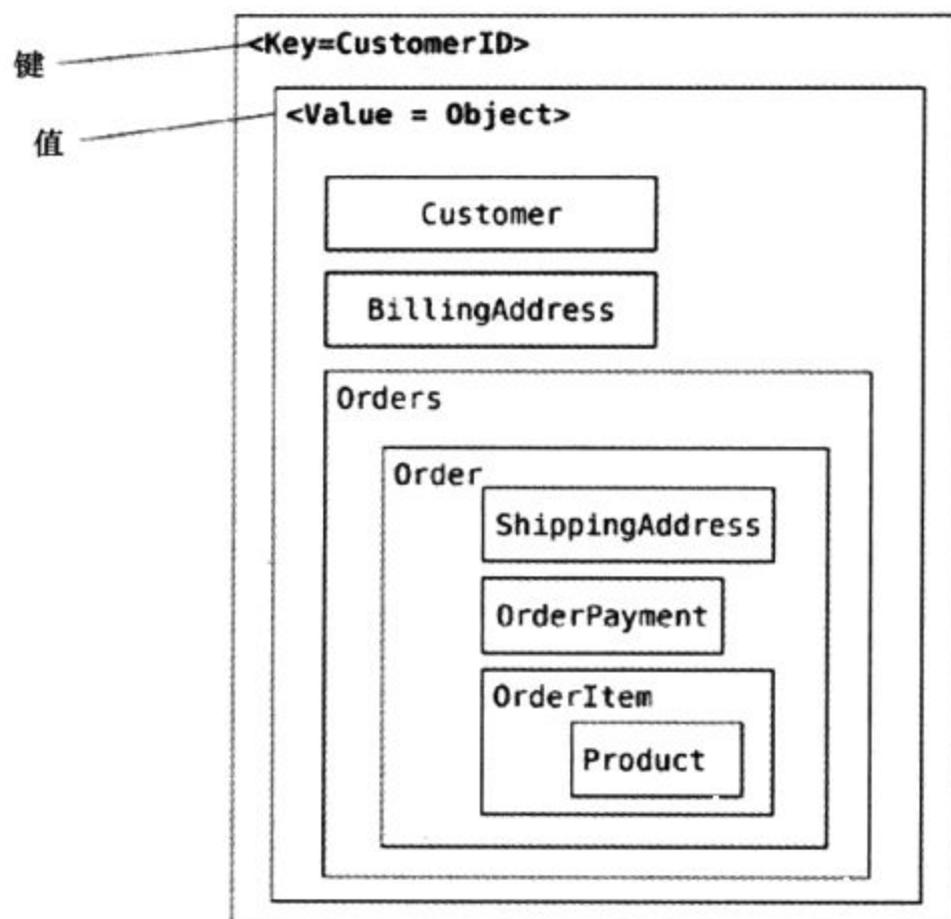


图 3.2 将表示客户及其订单的所有对象都嵌入同一聚合

在这种场景下，应用程序可以通过“键”读出客户信息及其全部数据。若是需要读取订单或每份订单内的产品，那么必须读取整个对象，让客户端解析并生成结果。如果需要引用，那么可改用文档数据库，并在文档内部查询。也可以把“键值存储”中的数据切分为 Customer 和 Order 两个对象，让它们各自维护一份指向对方的引用。

如果在数据中加入引用信息（如图 3.3 所示），那么就可以单独根据 Customer 对象查询其订单，通过 Customer 对象的 orderId 引用，就能查到该客户（Customer）所下的全部订单（Order）。以这种方式使用聚合，可以优化读取速度，但是每次向 Customer 对象中新增 Order 对象时，也要将指向它的 orderId 引用一并加进去。

```
# Customer object
{
  "customerId": 1,
  "customer": {
    "name": "Martin",
    "billingAddress": [{"city": "Chicago"}],
    "payment": [{"type": "debit", "ccinfo": "1000-1000-1000-1000"}],
    "orders": [{"orderId": 99}]
  }
}

# Order object
```

```

{
  "customerId": 1,
  "orderId": 99,
  "order": {
    "orderDate": "Nov-20-2011",
    "orderItems": [{"productId": 27, "price": 32.45}],
    "orderPayment": [{"ccinfo": "1000-1000-1000-1000",
      "txnId": "abelif879rft"}],
    "shippingAddress": {"city": "Chicago"}
  }
}

```

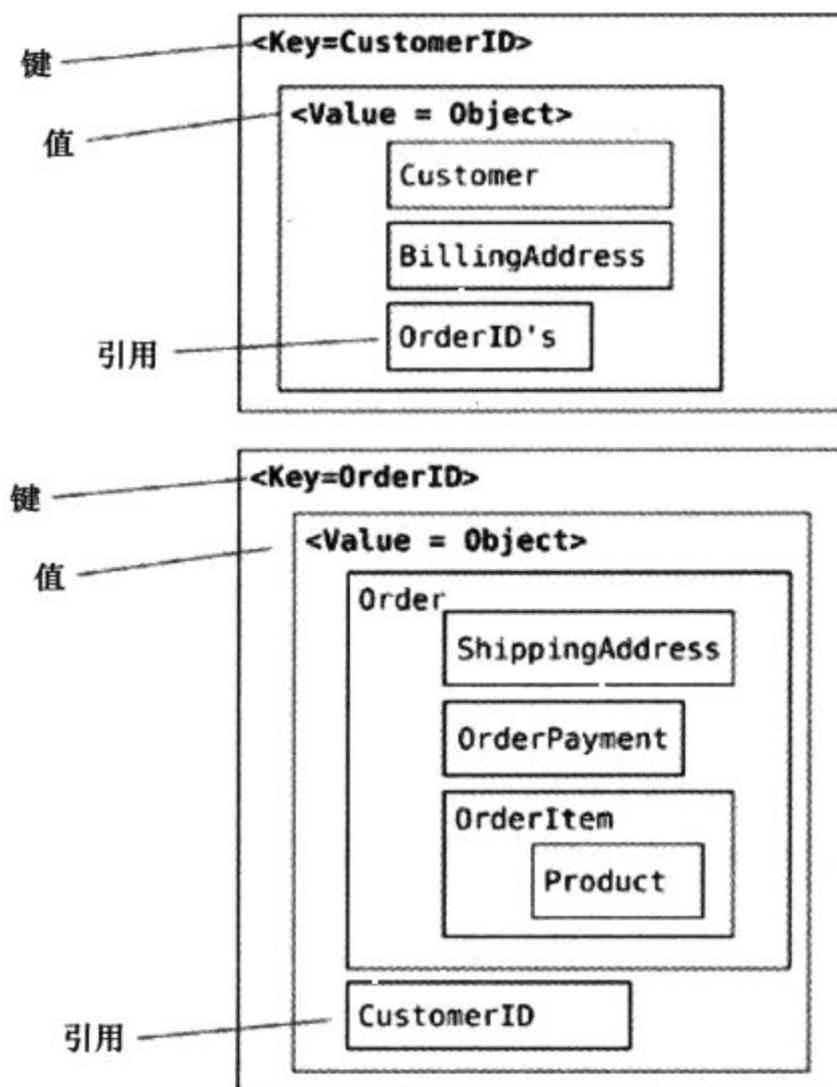


图 3.3 将 Customer 与 Order 对象分开存放

也可以用聚合来分析数据，比方说，在更新聚合时，可以将包含特定产品（Product）的订单（Order）汇总信息填入其中。通过这种不规范的数据形式，可以迅速找到想要的信息，这就是“实时 BI”<sup>⊖</sup>或“实时分析”（Real Time Analytics）的基础。企业不用像原来那样在每天工作结束后批量统计“数据仓库表”（data warehouse table）并生成分析结果了，现在只要客户下完订单，它们就可以把此类数据填入数据库中，

⊖ Real Time BI，是 Real Time Business Intelligence 的缩写，该词也写为“RTBI”，中文叫做“实时商务智能”。详情参见：[http://en.wikipedia.org/wiki/Business\\_Intelligence](http://en.wikipedia.org/wiki/Business_Intelligence)。——译者注

以满足各种不同类型的需求。

```
{
  "itemid":27,
  "orders":{99,545,897,678}
}
{
  "itemid":29,
  "orders":{199,545,704,819}
}
```

在文档数据库中，因为能够快速在文档内搜寻，所以可移除 Customer 对象里指向 Order 对象的引用。修改完后，如果客户（Customer）新下了订单（Order），就不用再更新 Customer 对象了。

```
# Customer object
{
  "customerId": 1,
  "name": "Martin",
  "billingAddress": [{"city": "Chicago"}],
  "payment": [
    {"type": "debit",
     "ccinfo": "1000-1000-1000-1000"}
  ]
}

# Order object
{
  "orderId": 99,
  "customerId": 1,
  "orderDate": "Nov-20-2011",
  "orderItems": [{"productId":27, "price": 32.45}],
  "orderPayment": [{"ccinfo": "1000-1000-1000-1000",
                    "txnId": "abelif879rft"}],
  "shippingAddress": {"city": "Chicago"}
}
```

由于在文档数据库中可按属性查询文档内容，所以，就能够执行诸如“找到所有包含《Refactoring Databases》一书的订单”之类的搜索了。但是，决定将订单商品与订单放在同一个聚合内，所依据的并不是数据库的查询能力，而是应用程序要如何优化数据读取。

如果以“列族存储”形式建模，那么就可以调整各列的次序了。我们可以给频繁用到的那些列起一个能够排在前面的名字，这样就能优先读取它们了。使用列族建模时，应该按照查询需求而非写入需求来做。建模的通则是要便于查询，而且在写入数据时要对其执行“反规范化”（denormalize）操作。

正如大家所想，有很多种数据建模方式。其中一种是把 Customer 与 Order 存放在不同的列族小组中（column-family family，如图 3.4 所示）。这里要注意一点，指向客户所下全部订单的引用，都放在 Customer 列族中。其他与之相似的“反规范化”操作

也都是为了改善查询（读取）性能。

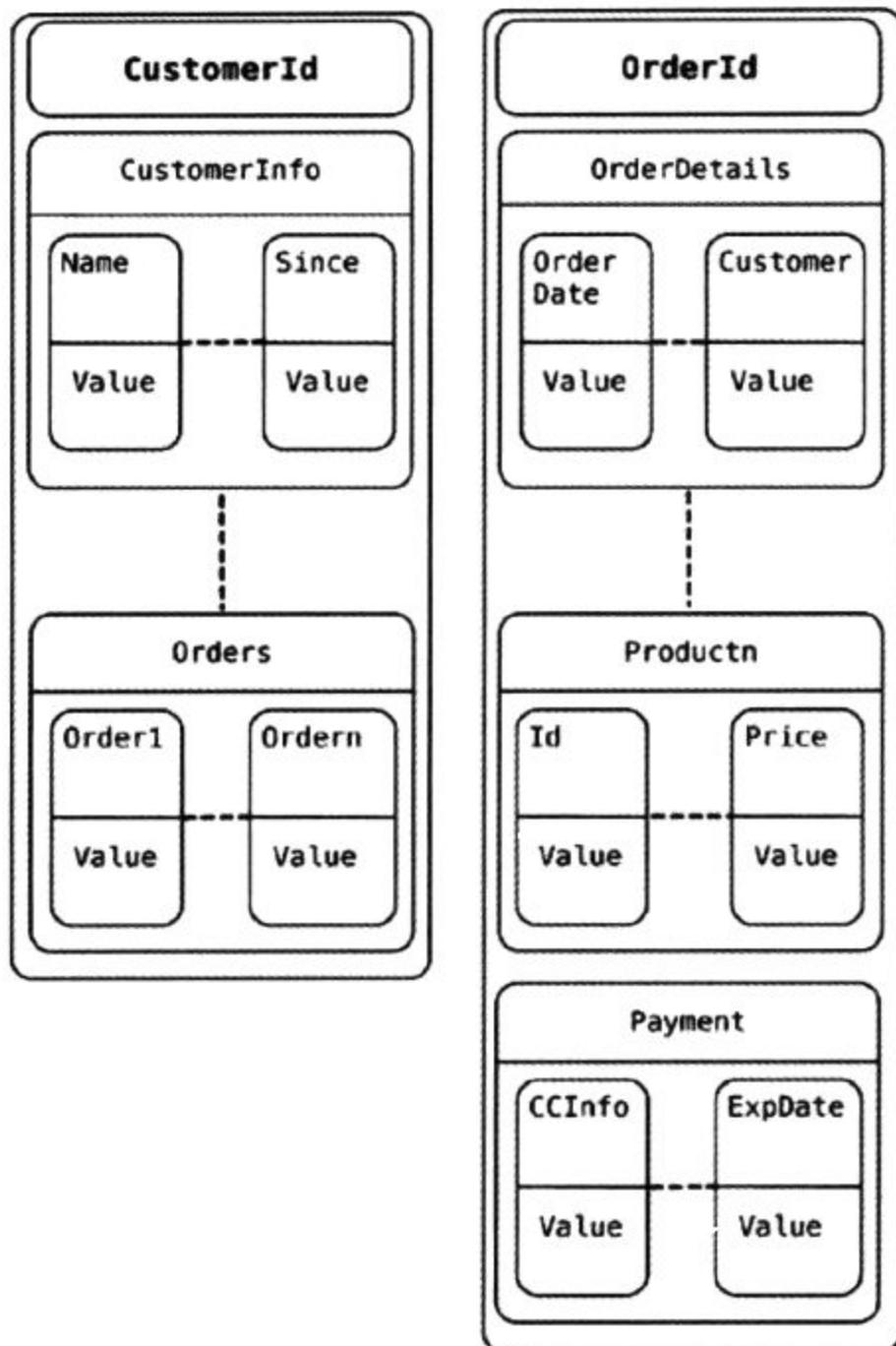


图 3.4 列式数据存储的概念图

同样一份数据，若用图数据库来建模，则要将其中的对象做成节点，将对象之间的关系变成节点之间的关系。这些关系的类型与方向很重要。

每个节点与其他节点的关系都各自独立。这些关系标有 *PURCHASED*（为某位客户所购买）、*PAID\_WITH*（以某种方式支付）或 *BELONGS\_TO*（是某种支付方式的使用者）等名字（如图 3.5 所示），我们可以通过这些名称来遍历图。比方说，想要找出购买（*PURCHASED*）了《*Refactoring Databases*》一书的所有客户（*Customer*），那么只需查出名为 *Refactoring Databases* 的产品节点，然后由该节点出发，找出所有和它具有 *PURCHASED* 关系的客户节点即可。

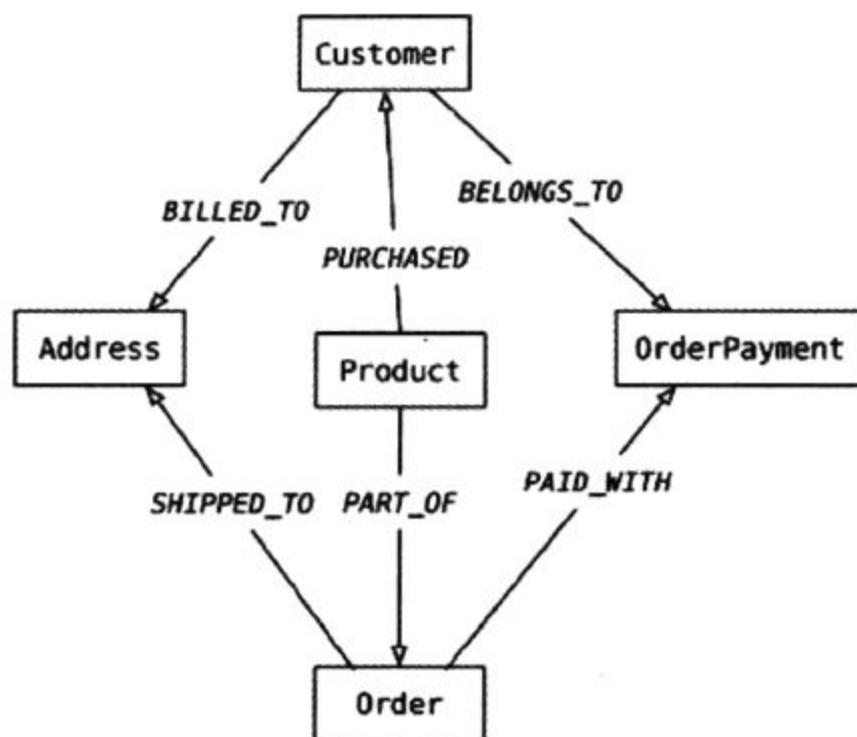


图 3.5 用“图模型”表示电子商务数据

此类关系用图数据库遍历起来非常容易，尤其是想使用此数据为用户推荐产品或发掘用户的操作模式时，更应该使用图数据库。

### 3.6 要点

- 使用面向聚合的数据库处理不同聚合之间的关系，要比处理同一聚合内部的关系更难。
- 图数据库将数据组织成一张由节点和边所组成的图，它们适合处理关系复杂的数据结构。
- 在无模式数据库中，可以给记录随意新增字段，然而用户在使用数据时，通常还是要遵循一套隐式模式。
- 面向聚合的数据库通常能够用不同方式重组主聚合（primary aggregate）的数据，以计算出各种“物化视图”。计算过程一般通过“映射化简”来实现。

## 第 4 章

# 分布式模型

催生 NoSQL 的主要原因是：我们需要一种能够运行在大集群上的数据库。随着数据量越来越多，以购买更大服务器来运行数据库的纵向扩展（**scale up**）方式，会变得愈发困难和昂贵。与之相比，将数据库运行在服务器集群上的横向扩展（**scale out**）方式，却颇受青睐。面向聚合数据库非常适用于横向扩展方式，因为聚合此时就自然成了数据分布单元。

各种分布式模型所带来的好处也不同。有些模型的数据存储能够处理大量数据，有些能够处理大量的网络读取或写入请求，还有些能够更好地应对网络速度慢或网络故障等状况。这些优势都很重要，不过它们都有其成本。在集群上运行数据库比较复杂，所以除非刚才说的那些优点确有必要，否则不应随意选用。

宽泛地说，数据分布有两条路径：复制（**replication**）与分片（**sharding**）。“复制”就是将同一份数据拷贝至多个节点；而“分片”则是将不同数据存放在不同节点中。复制与分片是两项“正交的”<sup>⊖</sup>（**orthogonal**）技术：既可以在两者中选一个来用，也可以同时使用它们。“复制”有两种形式：“主从式”（**master-slave**）和“对等式”（**peer-to-peer**）。现在按照由简至繁的顺序来讨论这些技术：先讲单一服务器，然后讲分片，再讲主从复制，最后讲对等复制。

---

⊖ 本是表示“垂直”含义的数学用语，在软件工程中，喻指多个因素互不干扰，可分别对待。——译者注

## 4.1 单一服务器

首先，在大多数情况下，都推荐使用最简单的分布形式：也就是根本不分布。将数据库放在一台电脑中，让它处理对数据存储的读取与写入操作。这种方式好就好在它不用考虑使用其他方案时所需应对的复杂事务，这对数据操作管理者与应用程序开发者来说，都比较简单。

尽管许多 NoSQL 数据库都是为集群运行环境而设计的，但是只要某个 NoSQL 数据库的数据模型符合应用程序需求，那就完全可以按照单一服务器的分布模型来用它。图数据库显然属于此类：它们最好配置在一台服务器上。若使用数据库基本上是为了处理聚合，那么可以考虑在单一服务器上部署“文档数据库”或“键值数据库”，这样也能简化应用程序开发者的工作。

本章接下来就要逐步讲解那些更为复杂的分布方案所带来的好处和麻烦。不过，别因为那些方案所占的篇幅较大，就误认为笔者更倾向于使用它们。在不需分布数据就能应对时，总应选用“单一服务器”方案。

## 4.2 分片

一般来说，数据库的繁忙体现在：不同用户需要访问数据集中的不同部分。在这种情况下，我们把数据的各个部分存放于不同的服务器中，以此实现横向扩展。该技术就叫“分片”（sharding，如图 4.1 所示）。

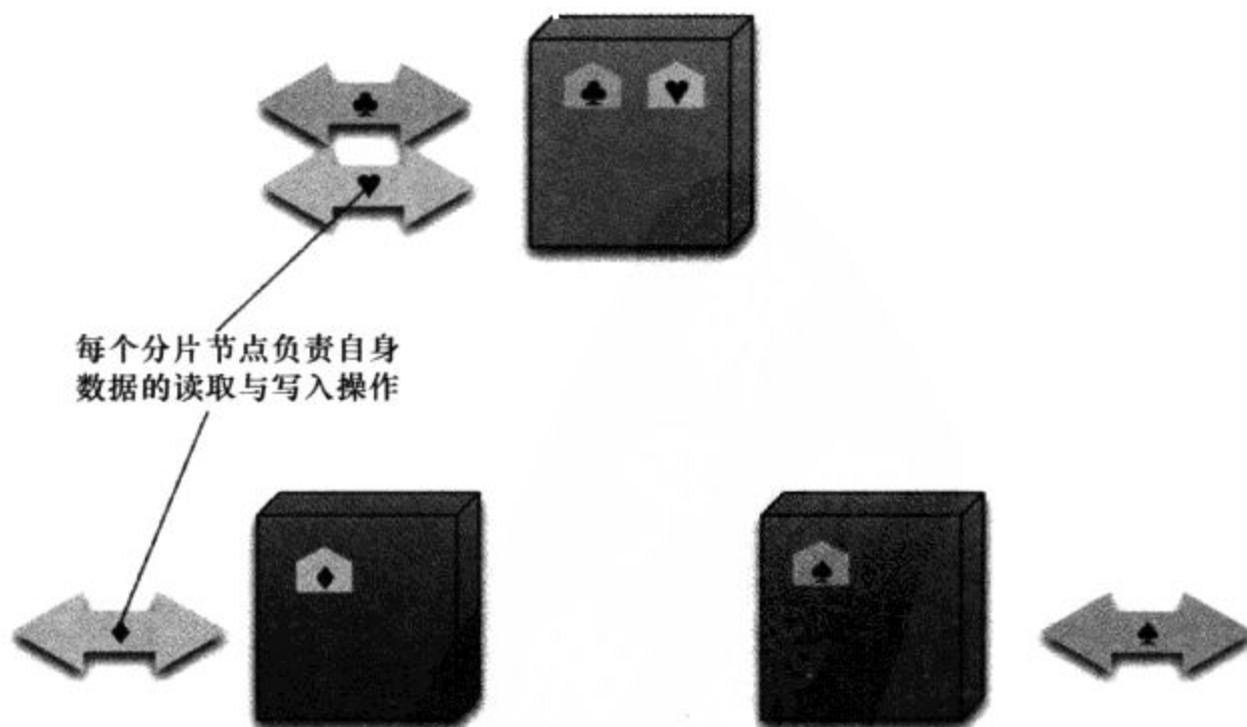


图 4.1 将不同部分的数据分片存放于独立的节点上，每个节点负责自身数据的读取与写入操作

在理想情况下，不同的服务器节点会服务于不同的用户。每位用户只需与一台服务器通信，并且很快就能获得服务器的响应。网络负载相当均衡地分布于各台服务器上。比方说，如果有 10 台服务器，那么每台服务器只需分担 10% 的负载。

理想状况当然比较罕见。为了获得近乎理想的效果，必须保证需要同时访问的那些数据都存放在同一节点上，而且节点必须排布好这些数据块，使访问速度最优。

要解决这个问题，首先得考虑：怎样存放数据，才能保证用户基本上只需要从一台服务器中获取它。若使用面向聚合的数据库，那就非常方便了。之所以设计“聚合”这一结构，就是为了把那些经常需要同时访问的数据放在一起。因此，显然可以把聚合作为分布数据的单元。

在节点的数据排布问题上，有若干个与性能改善相关的因素。若是能够确定聚合的访问者基本上都处在某个地理范围内，那么就可以把数据放得离访问者近一些。如果查询订单数据的用户住在波士顿，那么可以把该用户所需的数据放在美国东部的数据中心。

另外一个因素就是要保持负载均衡。意思就是，要把聚合数据均匀地分布在各个节点中，让它们需要处理的负载量相等。负载分布情况可能会随着时间变化，例如某些数据的访问量可能集中在一周的某几天内。因此，可能需要使用一些“领域特定规则”（domain-specific rule）。

在某些情况下，可以把有可能需要依次读取的聚合放在一起。Bigtable 论文 [Chang, etc.] 中说，可以按照“字典顺序”<sup>⊖</sup>排列表中的“行”，以“逆域名”（reversed domain name，例如 com.martinfowler）为序来排列网址。如果要同时访问跨越多个页面的数据，那么此方式有助于改善处理效率。

以前很多人都把分片工作放在应用程序的逻辑里实现。可以把姓氏首字母为 A ~ D 的客户放在一个分片，而把姓氏首字母为 E ~ G 的客户放在另一个分片。这就使编程模型变复杂了，因为应用程序的代码必须负责把查询操作分布到多个分片上。此外，若想重新调整分片，那么既要修改程序代码，又要迁移数据。很多 NoSQL 数据库都提供了“自动分片”（auto-sharding）功能，可以让数据库自己负责把数据分布到各分片，并且将数据访问请求引导至适当的分片上。这样一来，应用程

---

⊖ 这里原文是 lexicographic order，也就是字母表顺序。——译者注

序使用分片就能方便多了。

分片对提升性能尤其有用，因为它可以同时提升读取与写入效率。使用“复制”技术，尤其是带缓存的复制，可以极大地改善读取性能，但对那种需要频繁执行写入操作的应用程序，却帮助不大。而分片则提供了一种可以横向扩展写入能力的方式。

如果单就分片技术本身来说，它对改善数据库的“故障恢复能力”<sup>⊖</sup>帮助并不大。尽管数据分布在不同的节点上，但是和“单一服务器”方案一样，只要某节点出错，那么该分片上的数据就无法访问了。它所提供的弹性仅仅在于：只有访问此数据的那些用户才会受影响，而其余用户则能正常访问。然而，数据库缺失了一部分数据，毕竟不好。如果数据库只放在一台服务器上，那么花些精力和成本保证服务器能持续正常运转就好，而运行在集群上的电脑，其稳定性会稍差些，所以难免遇到节点出错的情况。因此，从实际角度看，分片技术可能会降低数据库的错误恢复能力。

有了“聚合”结构之后，使用起分片来确实更容易了，但尽管如此，也不应草率决定。某些数据库一开始就决定要使用分片技术，在这种情况下，最好是开发伊始就将其部署在集群上，在进入产品上线阶段之后，当然更要如此。而在另外一些情况下，如果想对原来运行在单一服务器上的数据库应用分片技术，则要三思而后行。此时最好的办法是先使用一台服务器，等到现有服务能力已经明显无法应对负载量时，再改用分片。

无论如何，从单一节点向分片迁移，都比较难办。笔者听说有的团队到项目马上开发完毕时才开始往分片迁移，这样的话，一旦在产品上线阶段启用分片技术，那么数据库一下子就无法访问了，因为在把数据向新分片迁移的过程中，所有数据库资源都用来支持分片，就没法处理数据访问请求了。此处的经验是，在真正要使用分片技术之前，应该尽早准备好，也就是说，在尚有余地之时尽快将数据迁移至分片。

### 4.3 主从复制

在“主从式分布”（master-slave distribution）中，我们要把数据复制到多个节点

---

⊖ 这里原文是 resilience，直译“弹性”，在数据库语境中，是指它从错误中恢复的能力。——译者注

上。其中有一个节点叫做“主（master）节点”，或“主要（primary）节点”。主节点存放权威数据，而且通常负责处理数据更新操作。其余节点都叫“从（slave）节点”，或“次要（secondary）节点”。复制操作就要让从节点与主节点同步（如图 4.2 所示）。

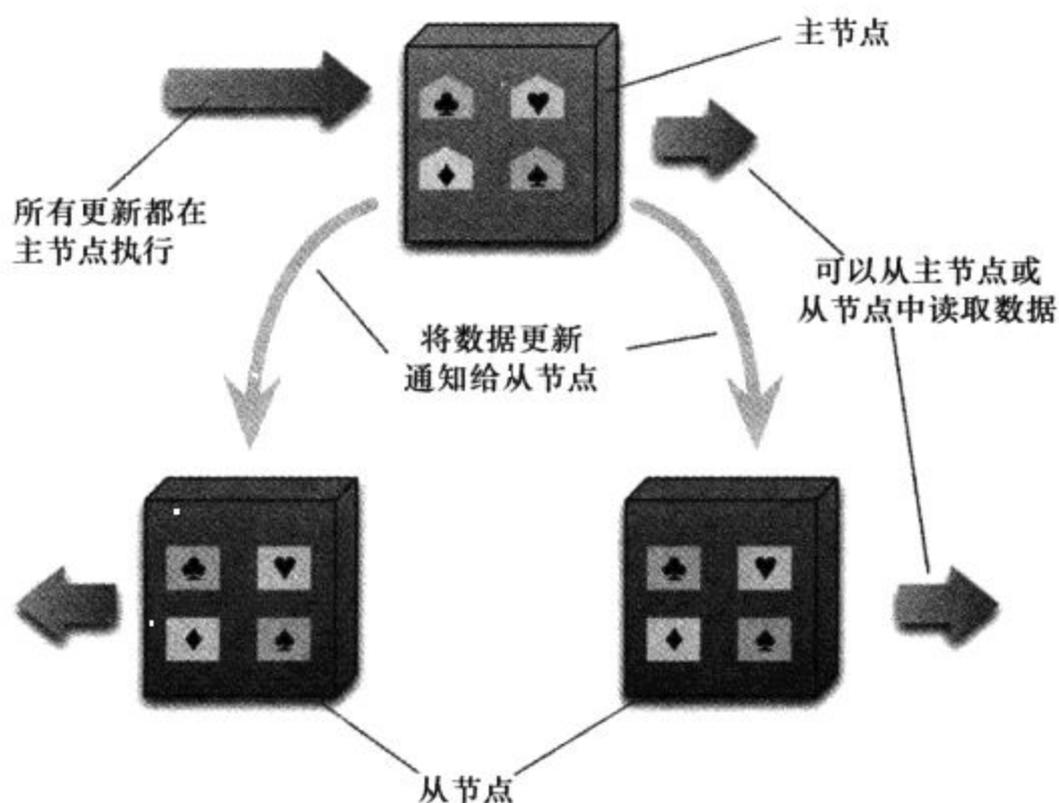


图 4.2 数据由主节点复制到从节点。所有写入请求都由主节点处理，读取请求可以由主节点处理，也可以交给从节点

在需要频繁读取数据集的情况下，“主从复制”（master-slave replication）最有助于提升数据访问性能了。以新增更多从节点的方式来进行水平扩展，就可以同时处理更多数据读取请求，并且能保证将所有请求都引导至从节点。然而，数据库仍受制于主节点处理更新，以及向从节点发布更新的能力。所以在写入操作特别频繁的场所，虽说将读取操作分流，可以稍微缓解写入操作的处理压力，但是这样安排数据集的效果并不好。

“主从复制”的第二个好处是，它可以增强“读取操作的故障恢复能力”（read resilience）：万一主节点出错了，那么从节点依然可以处理读取请求。这个优势也是要在数据读取操作占大头时才能体现出来。主节点出错之后，除非将其恢复，或另行指派新的主节点，否则数据库就无法处理写入操作了。然而，拥有内容与主节点相同的从节点，可以提高数据库的恢复速度，因为主节点出错之后，很快就能指派一个从节点作为新的主节点。

能够用新指派的从节点来代替出错的主节点，这意味着就算不需要横向扩展，“主从复制”也有其用处。在主节点处理所有读写操作的同时，从节点可以充当“即时备份”（hot backup）。在此情况下，可将整个系统视为带有“即时备份”功能的“单服务器存储（single-server store）方案”，这样理解起来最容易了。这种方案和单服务器方案一样方便，然而它更具故障恢复能力，在需要优雅地处理服务器故障时，这么做尤为便利。

主节点可以手工指派，也可自动选择。若想手工指派，那么一般需要自己来配置集群，将其中一个节点设为主节点。如果自动指派，那么就创建好节点集群，让它们自行选举出主节点。采用自动指派方案，不仅配置起来较为简单，而且当主节点出错时，集群可以自动指派新的主节点，以减少停机时间（downtime）。

为了使读取操作具备故障恢复能力，必须确保应用程序分别沿着不同的路径发出读取请求与写入请求，这样才能在处理写入路径的故障时保证读取操作不受影响。这就需要那种分别使用不同的数据库连接来处理读取与写入请求的机制，而提供数据库交互操作的程序库一般都不支持此功能。与研发其他功能一样，我们必须用可靠的测试来验证：禁用写入功能后，读取操作依然可以照常执行。要是不这么做，那就没办法保证读取操作的故障恢复能力。

“复制”技术除了带给我们这些诱人的好处之外，也伴有一个不可避免的缺陷，那就是数据的不一致性。如果数据更新还没有全部通知给从节点，那么不同的客户端就可能于不同的从节点中读出内容各异的值。在最糟糕的情况下，客户端甚至无法读出它刚刚写入的那个值。就算使用“主从复制”只是为了做“即时备份”，也得考虑这个问题，因为一旦主节点出错，那么尚未更新到从节点的数据就会丢失。稍后将讲解如何应对此类问题（参见第5章）。

## 4.4 对等复制

“主从复制”有助于增强读取操作的故障恢复能力，然而对写入操作却帮助不大。它所提供的故障恢复能力，只有在从节点出错时才能体现出来，而不针对主节点。实际上，主节点仍然是系统的瓶颈与弱点。“对等复制”（peer-to-peer replication，如图4.3所示）能解决此问题，它没有“主节点”这一概念。所有“副

本” (replica) 地位相同, 都可以接受写入请求, 而且丢失其中一个副本, 并不影响整个数据库的访问。

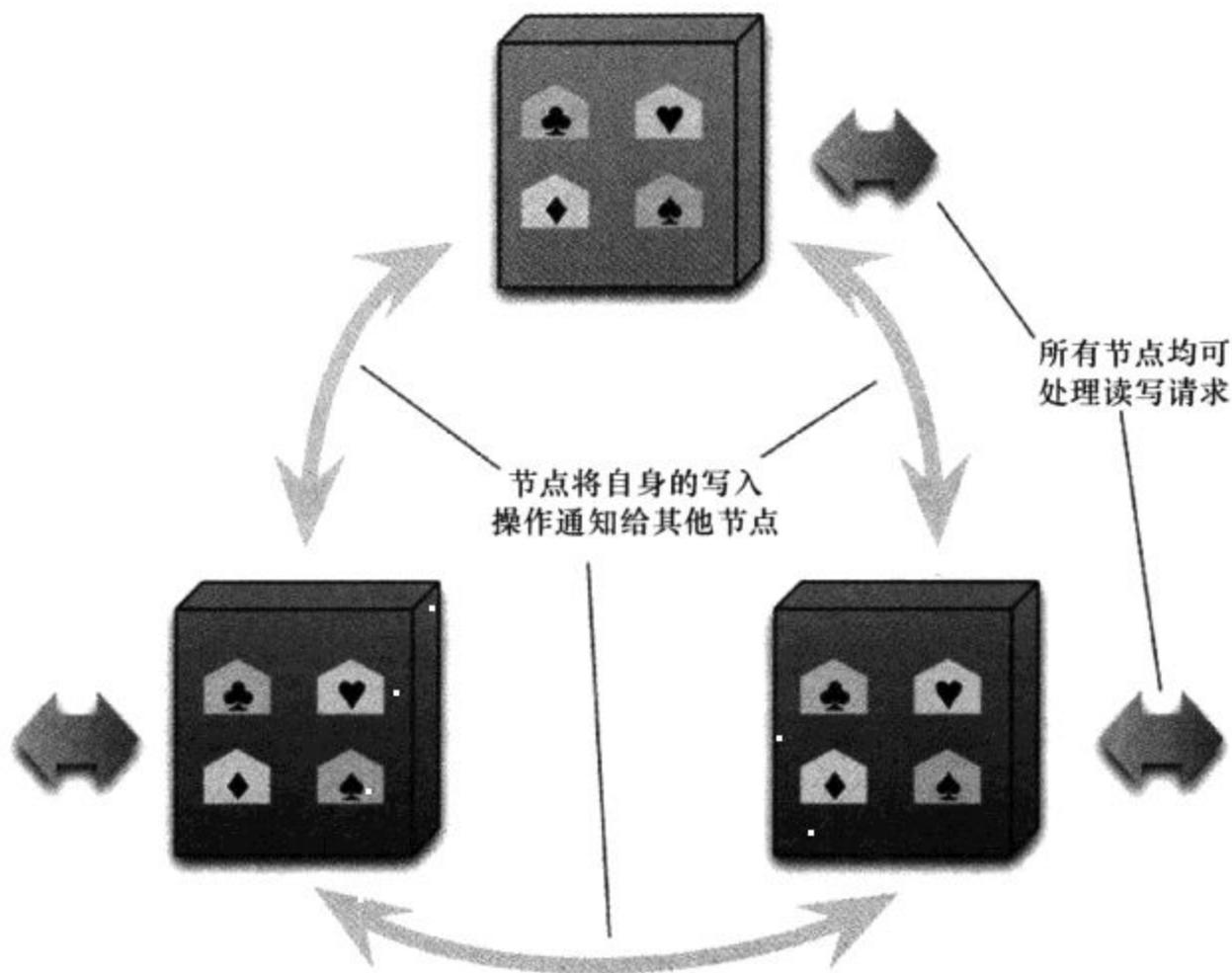


图 4.3 “对等复制”方案的所有节点都可以执行数据读写操作

这个方案看上去真的不错。在集群上使用“对等复制”方案时, 可以从容处理那些出错的节点, 而不必担心数据请求会丢失。此外, 只需增加节点, 就能轻易提升性能。它优点很多, 不过也有其复杂之处。

最大的麻烦还是数据的一致性。由于两个不同的节点可以同时处理写入操作, 所以有可能出现两位用户在同一时间试图更新同一条记录的情况, 这就导致“写入冲突”(write-write conflict)。数据读取的不一致也是个问题, 不过它们至少持续时间相对较短, 而写入操作导致的不一致数据却总是存在。

本书后续内容将会谈到如何解决写入操作的不一致问题。不过现在读者只需知道几条大致思路就好。有一种比较极至的思路是, 不管何时写入数据, 各副本之间总能相互协调, 确保不发生冲突。尽管要花费一些网络流量来协调写入操作, 不过它和“主从复制”方案中的“主节点”一样, 切实地保证了数据的一致性。写入操作不需要在所有副本节点中都保持一致, 只需大部分一致即可, 这样就算丢失了那一小部分副本, 数据库还是能用。

另一种与之相对的极端思路是，设法处理这些不一致的写入操作。在某些情况下，可以想出一些办法，把这些相互冲突的写入操作合并起来。这样的话，任意副本节点都能写入数据，而此系统的性能优势也就完全发挥出来了。

在一致性与可用性之间权衡时，可以考虑很多办法，而上述两种思路代表了两个极端。

## 4.5 结合“分片”与“复制”技术

复制与分片两种策略可以结合起来用。如果同时使用“主从复制”与“分片”（如图4.4所示），那么就意味着整个系统有多个主节点，然而对每项数据来说，负责它的主节点只有一个。根据配置需要，同一个节点既可以做某些数据的主节点，也可以充当其他数据的从节点，此外，也可以指派全职的主节点或从节点。

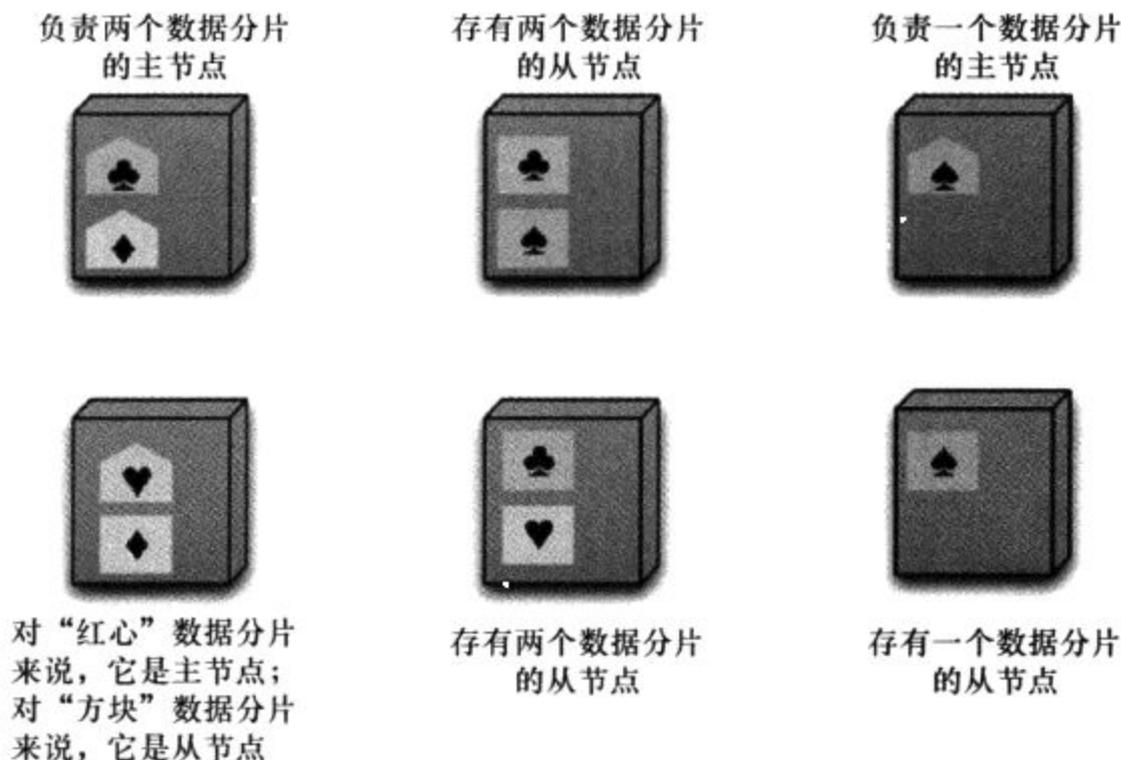


图 4.4 结合使用“主从复制”与“分片”技术<sup>⊖</sup>

使用列族数据库时，经常会将“对等复制”与“分片”结合起来。在这种情况下，数据可能分布于集群中的数十个或数百个节点上。在采用“对等复制”方案时，一开始可以用“3”作为复制因子（replication factor），也就是把每个分片数据放在3个节点中。一旦某个节点出错，那么它上面保存的那些分片数据会由其他节点重建（如图4.5所示）。

<sup>⊖</sup> 带有尖顶五边形背景的数据分片存放在主节点中；带有矩形背景的数据分片存放在从节点中。——译者注

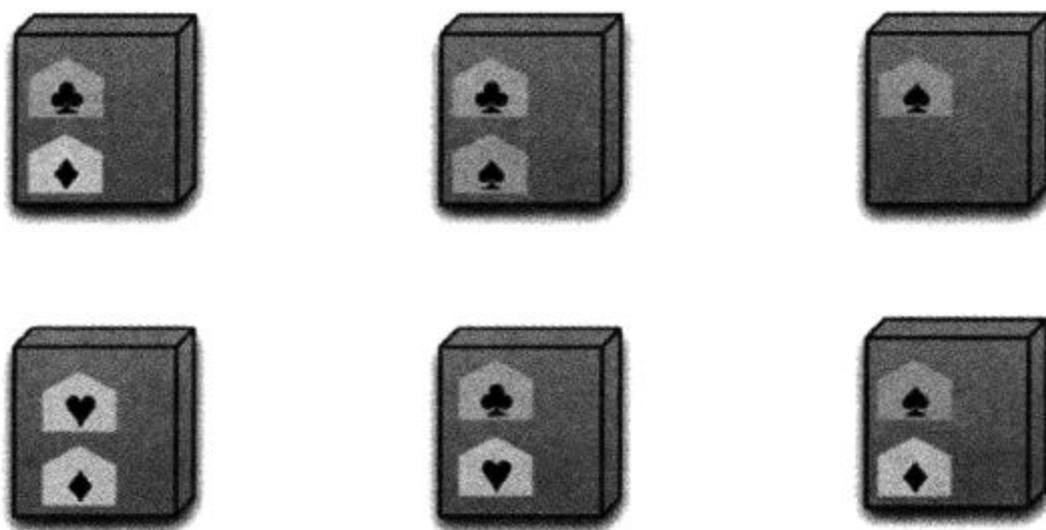


图 4.5 结合使用“对等复制”与“分片”技术

## 4.6 要点

- 数据分布有两种方式：

- 将不同的数据分片存放在多个服务器中，每一个数据子集（subset of data）都专门由一台服务器负责。
- 将数据复制到多个服务器上，每份数据都能在多个节点中找到。

数据库系统可以只选用其中一种技术，也可以两种都用。

- “复制”技术又有两种形式：

- “主从复制”：将其中一个节点当做权威数据源，并负责写入操作；其他从节点都要和主节点保持同步，它们可以负责读取操作。
- “对等复制”：任何节点均可写入，节点间相互协调以同步其数据。

“主从复制”减少了更新数据时的冲突几率，但它却会让主节点成为写入操作的瓶颈，而“对等复制”则避免了这一点。

# 第 5 章

## 一 致 性

由集中式关系型数据库迁移到面向集群的 NoSQL 数据库时，改变较大的一个地方，就是对一致性的思考方式。关系型数据库试图通过“强一致性”（strong consistency）来避免各种不一致问题，本章很快就要谈到它；而在 NoSQL 领域中，则会出现诸如“CAP 定理”（CAP theorem）与“最终一致性”（eventual consistency）这种术语，一旦开始构建，就必须思考当前系统需要何种一致性。

一致性有多种表现方式，而且它下面也潜藏着众多可能会出错的地方。所以，本章先谈谈一致性所具有的各种形式，然后再来讨论有哪些理由可以让开发者放宽对一致性的约束（并放宽另一个与之相伴的因素：持久性）。

## 5.1 更新一致性

首先思考一个更新电话号码的例子。Martin 和 Pramod 在浏览公司网站时发现，上面留的电话号码是旧的。可不巧的是，两人都有更新权限，于是，他们同时开始修改此号码。为了让该例更有趣些，我们假定两者选用的电话号码格式稍有差别，这样的话，他们所提交的新号码就会略有不同了。此时将产生“写冲突”（write-write conflict）问题：两人在同一时刻更新同一条数据。

服务器收到写入请求之后，就会将其“序列化”（serialize），也就是要决定这两个请求的处理顺序。此处假定服务器按照字母顺序来排列，也就是先受理 Martin 的更新请求，然后再受理 Pramod 的更新请求。若没有并发控制机制，则 Martin 提交的更新数据会立刻被 Pramod 的更新所覆盖。在这种情况下，Martin 提交的数据就发生了“更新丢失”（lost update）问题。此处的更新丢失不算大问题，然而一般来说，这种错误还是挺严重的。Pramod 提交更新时所依据的数据，是未经 Martin 修改的那一份，而当服务器真正处理其更新请求时，那份数据却已经被 Martin 修改了。

在并发环境下维护数据一致性所用的方式，通常分为“悲观方式”与“乐观方式”。“悲观”（pessimistic）方式就是避免发生冲突；而“乐观”（optimistic）方式则是先让冲突发生，然后检测冲突并对发生冲突的操作排序。在处理更新冲突时，最常见的“悲观方式”就是采用“写入锁”（write lock），这样的话，在修改某个值之前，必须先获取“写入锁”，系统确保某一时刻只有一个客户能够获得这把锁。如果 Martin 与 Pramod 同时想要获取“写入锁”，那么只有 Martin（也就是服务器先受理的那位客户）能获得该锁。Pramod 看了 Martin 所写入的数据之后，再来决定是否要更新它。

“乐观方式”通常采用“条件更新”（conditional update），也就是任意客户在执行更新操作之前，都要先测试数据的当前值和其上一次读入的值是否相同。在这种情况下，Martin 的更新操作能够成功执行，而 Pramod 的更新操作则会失败。Pramod 得知这个更新错误后，他可以再次查询该数据，以决定是否需要继续修改。

上面所讲的“悲观”与“乐观”这两种方式，都有个先决条件，那就是更新操作的顺序必须一致。在单服务器环境中，这显然成立：因为它必须先处理完一个操作，才能处理下一个。然而，如果服务器的数量不只一个（例如在“对等复制”环境中），那么两个节点就可能会按照不同的次序执行更新操作，这样造成的结果是，每个节点最

终保存的电话号码不一致。在谈到分布式系统的并发问题时，大家通常说的是“顺序一致性”（sequential consistency），也就是所有节点都要保证以相同次序执行操作。

还有一种处理“写冲突”的“乐观方式”，那就是将两份更新数据都保存起来，并标注出它们存在冲突。很多使用版本控制系统（version control system）的程序员都熟悉这种方法，尤其是使用“分布式版本控制系统”（distributed version control system）<sup>⊖</sup>的人，因为此类系统经常会碰见相互冲突的提交操作。在处理冲突时，该方式遵循与版本控制系统相同的步骤：必须以某种方式将两个互相冲突的更新操作“合并”（merge）起来。系统可以将有冲突的两个值都呈现给用户，让其自行处理。如果通过手机端和电脑端修改了通讯录中同一个人的联系信息，那么就可能出现这种情况。另一种情况则是，计算机可以自己完成“合并”操作。假如它发现造成冲突的原因仅仅是电话号码格式问题，那么采用标准格式将新号码写入即可。以“自动合并”（automated merge）方式处理“写冲突”，是个极具“领域特定”（domain-specific）性质的问题，需要根据具体情况来编程。

头一次遇到此类问题时，由于决意要避免冲突，所以大家通常都会选用“悲观方式”来处理并发。某些时候这么做确实合适，然而一般情况下总是需要取舍。并发编程涉及一个根本问题，那就是在安全性（避免“更新冲突”之类的错误）与响应能力（liveness，快速响应客户操作）之间权衡。“悲观方式”通常会大幅降低系统响应能力，以致无法满足需求。而且它还有出错的危险：采用“悲观方式”处理并发问题，通常会导致“死锁”（deadlock），这一情况既难于防范，也不易调试。

采用“复制”模型来分布数据时，更容易遇到“写冲突”。如果不同节点含有同一份数据拷贝，那么它们可能会以各自独立的顺序来更新此数据。这时，除非采用某种具体的预防措施，否则就要发生冲突。把针对某份数据的所有写入操作都交由一个节点来完成，就更容易保持更新操作的一致性了。在早前讲到的各种分布模型中，除了“对等复制”模型外，其余方案都采用上述办法。

## 5.2 读取一致性

有一个能够保持更新一致性的数据库，只是解决了一方面的问题，它未必能保

<sup>⊖</sup> 有关版本控制系统的分类及术语，请参考：<http://zh.wikipedia.org/wiki/版本控制>。——译者注

证用户所提交的访问请求总是能得到内容一致的响应。假设有一个包含商品项（line item）与运费（shipping charge）的订单，其中运费要根据订单里的商品项来计算。若向订单中新增一项商品，则需重新计算并更新运费。在关系型数据库中，运费与商品项分别存放在不同的表中。如果 Martin 向其订单中新增了一项商品，Pramod 继而读出商品项及运费，最后 Martin 才更新运费，那么就会有数据不一致的风险。如图 5.1 所示，Pramod 在 Martin 的两个写入操作步骤之间读出数据，这就导致了“读取不一致”（inconsistent read）或“读写冲突”（read-write conflict）现象。

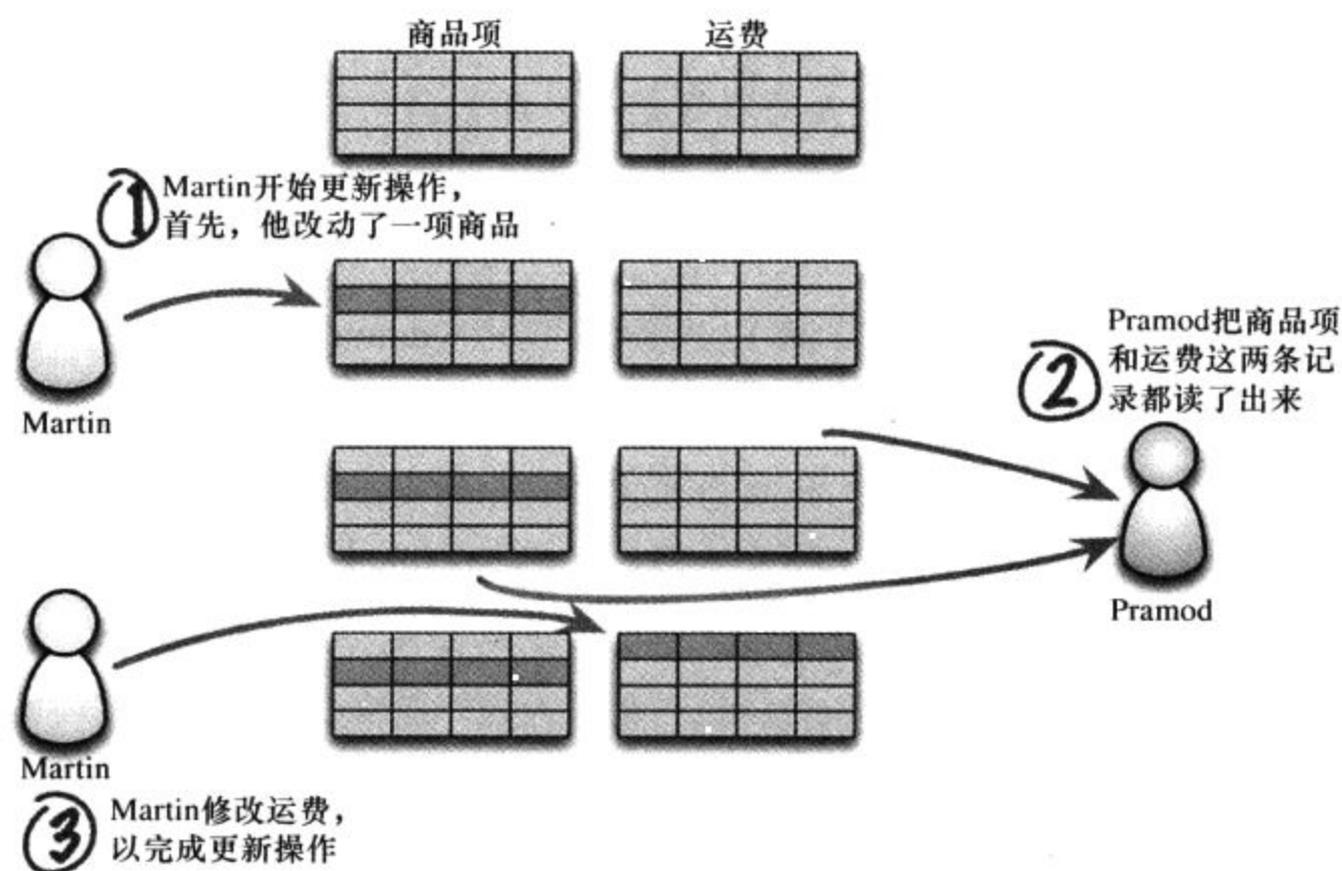


图 5.1 违反“逻辑一致性”的“读写冲突”

上述例子中的一致性就叫“逻辑一致性”（logical consistency），也就是要确保不同的数据项放在一起，其含义符合逻辑。为了避免“读写冲突”造成的“逻辑不一致”，关系型数据库支持“事务”这一概念。若是 Martin 将两个写入步骤封装到一个事务之中，则系统能够确保 Pramod 所读出的那两项数据，要么都是更新之前的值，要么都是更新之后的值。

有种常见的说法是：NoSQL 数据库不支持事务，所以也就无法保持数据的一致性。这种说法大多是错误的，因为它掩盖了诸多重要细节。首先要澄清一点：“不支持事务”这一说法，仅适用于部分 NoSQL 数据库，尤其是面向聚合的数据库。而与之相反，“图数据库”常常和关系型数据库一样，也支持 ACID 事务。

其次，面向聚合的数据库通常支持“原子更新”（atomic update），但仅限于单一聚合内部。这就是说，“逻辑一致性”可以在某个聚合内部保持，但在各聚合之间则不行。所以，在上例中，若是把订单、运费、商品项全都放在一个订单聚合里，则可避免“逻辑不一致”问题。

我们显然不能把所有数据都放在一个聚合里面，所以，在执行影响多个聚合的更新操作时，会留下一段时间空档，让客户端有可能在此刻读出逻辑不一致的数据。存在不一致风险的时间长度就叫“不一致窗口”（inconsistency window）。NoSQL 系统的“不一致窗口”很短暂。有份数据显示：亚马逊公司在文档中声称其 SimpleDB 服务的“不一致窗口”通常少于 1 秒。

上面这个例子说明了读取操作可能发生的“逻辑不一致”问题，在任何一本谈到数据库编程的教材中，都能看到这种经典范例。然而，一旦引入“复制”机制，就会遭遇一种全新的不一致问题。假设为了参加某活动，大家都想预订客房，而酒店只剩最后一间房了。Martin 与 Cindy 夫妇正考虑要不要预订，由于 Martin 在伦敦而 Cindy 在波士顿，所以他们打电话讨论此事。同时，在孟买的 Pramod 把这最后一间房预订了。这时数据库就要把房间剩余情况更新到其他副本中，然而新数据到达波士顿的时间比伦敦早。所以当 Martin 和 Cindy 打开浏览器想看看还有没有房间时，Cindy 看到的结果是房间已订走，而 Martin 看到的却是该房间尚未预订。这又是一个“读取不一致”问题，它违反了另一种形式的一致性，也就是“复制一致性”（replication consistency）。它要求从不同副本中读取同一个数据项时，所得到的值相同（如图 5.2 所示）。

当然，最终更新操作还是会传播到全部节点中，这样 Martin 就能看到所有房间都预订完了。因此，这种情况通常叫做“最终一致性”（eventually consistent），也就是说，在任意时刻，节点中都可能存在“复制不一致”（replication inconsistency）问题，然而只要不再继续执行其他更新操作，那么上一次更新操作的结果最终将会反映到全部节点中去。过期的数据通常称为“陈旧”（stale）数据，这提醒我们：缓存也算一种“复制”形式，尤其在“主从式分布模型”中，更是如此。

虽说“复制一致性”与“逻辑一致性”是两个相互独立的问题，不过如果“复制”过程中的“不一致窗口”太长，那么也会加剧“逻辑不一致”问题。两个间隔时间很短且内容不同的更新操作，在主节点中留下的“不一致窗口”也就是几毫秒而已，但

是由于网络延迟，这个“不一致窗口”在从节点上的持续时间比主节点长很多。

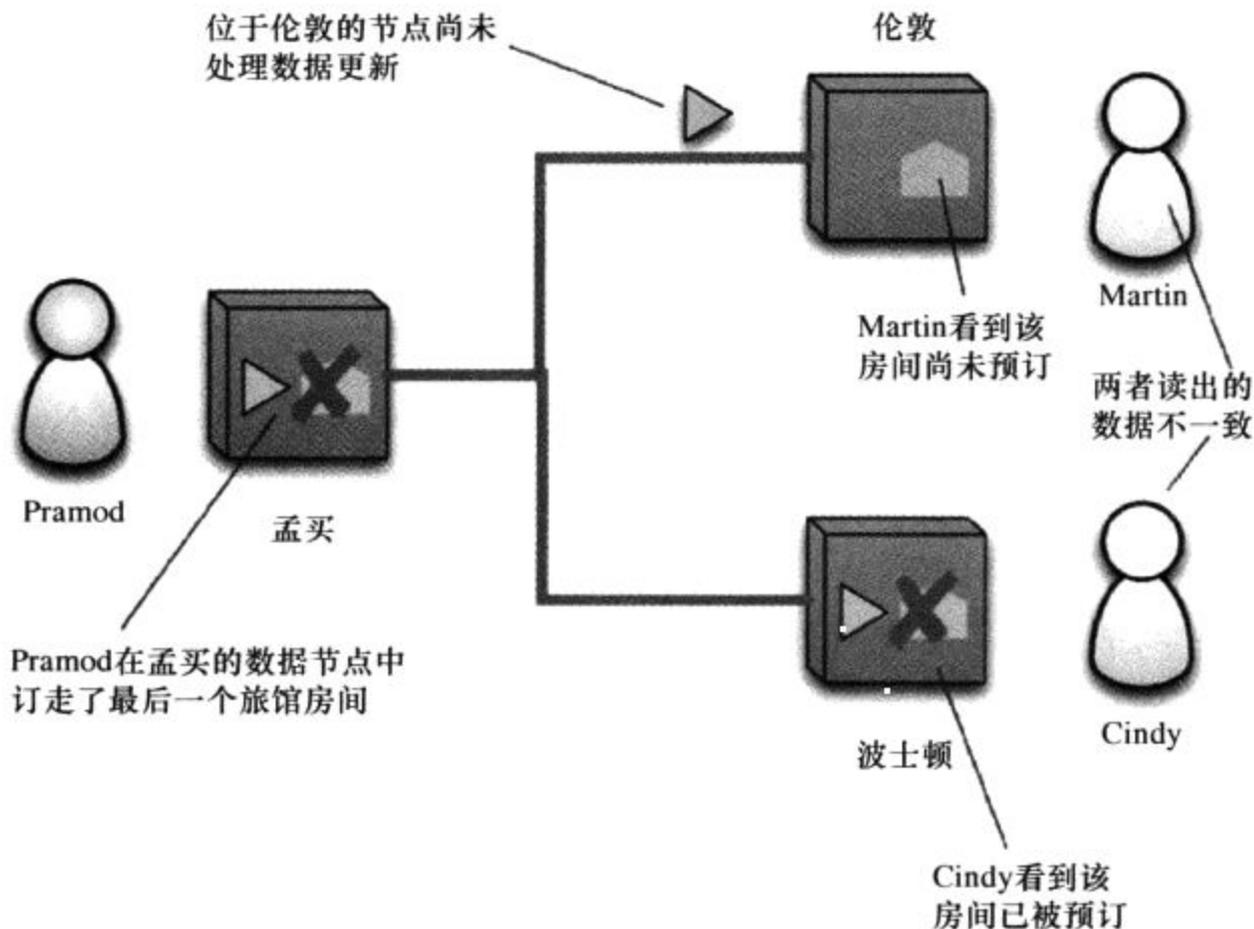


图 5.2 举例说明“复制不一致”问题

“一致性”不是某种针对整个应用程序的保证。通常可以指定单个请求所需达到的一致性级别。这样的话，就可以使用一致性比较弱的操作了，这在大多数情况下都不会出问题，而当有必要时，则可以使用一致性比较强的请求。

由于存在“不一致窗口”，所以不同的人在同一时刻可能会看到不同的数据。如果 Martin 和 Cindy 正在越洋电话中讨论预订房间的事情，那么这种情况可能会让他们困惑；而一般情况下，用户都是各自操作，所以就不会引发问题。但是，有时在用户独自一人操作时，“不一致窗口”却会带来麻烦。举例来说，假设用户想对一篇博文发表评论。在把刚想好的观点打成文字时，就算“不一致窗口”持续好几分钟，你也不用担心。像这种运行于集群中的网站，在处理负载时，其系统一般会把传入的请求均衡地分配到不同节点之中。这存在风险：如果你刚通过某个节点发布了一条评论，然后刷新浏览器，而负责处理刷新请求的另一个节点却尚未收到刚才的评论，那么你就会误认为刚贴上去的消息丢了。

在这种情况下，你可以容忍相当长的“不一致窗口”，然而需要确保“照原样读出所写内容的一致性”（read-your-writes consistency），也就是说，在执行完更新操作之

后，紧接着必须能看到更新之后的值才行。在具备“最终一致性”的系统中，有一种确保此性质的办法，那就是提供“会话一致性”（session consistency）：在用户会话内部保持“照原样读出所写内容的一致性”，这意味着，假如用户会话因为某种原因而终止，或是用户同时使用多台计算机访问同一个系统，那么就有可能失去一致性。但是，这些情况相对来说比较少见。

有几个技巧可以确保“会话一致性”，一种常见且最为简单的方式就是：使用“黏性会话”（sticky session），也就是绑定到某个节点的会话（这种性质也叫做“会话亲和力”，session affinity）。“黏性会话”可以保证，只要某节点具备“照原样读出所写内容的一致性”，那么与之绑定的会话就都具备这种特性了。“黏性会话”的缺点是，它会降低“负载均衡器”（load balancer）的效能。

另一种实现“会话一致性”的方式，就是使用“版本戳”（version stamp，参见第6章），并确保同数据库的每次交互操作中，都包含会话所见的最新版本戳。服务器节点在响应请求之前必须先保证，它所含有的更新数据包含此版本戳。

使用“黏性会话”和“主从复制”来保证“会话一致性”时，如果想把读取操作指派给从节点以改善读取性能，而同时仍然想将写入操作指派给主节点的话，就比较难办了。解决这个问题的一种办法是，将写入请求先发给从节点，由它负责将其转发至主节点，并同时保持客户端的“会话一致性”。另一种办法是，在执行写入操作时临时切换到主节点，并且在从节点尚未收到更新数据的这一段时间内，把读取操作都交由主节点来处理。

本书是在数据库语境下讨论“复制一致性”这个问题的，然而在为应用程序做整体设计时，它也是个重要因素。哪怕在一个简单的数据库系统中，也会经常出现用户查看数据，思考其内容，并更新数据的情况。在用户和数据库的交互过程中，通常不要把事务一直开着，因为在实际使用中，当用户更新数据时，可能真的会发生冲突，这种情况下就要使用“离线锁”（offline lock）一类的办法了 [Fowler PoEAA]。

### 5.3 放宽“一致性”约束

一致性是个好东西，不过有时必须舍弃它。在设计系统时，我们总是尽可能避免“不一致”现象，然而，若要真正做到这一点，通常需要放弃系统中的其他一些特性，而那些特性却是必不可少的。如此说来，我们经常需要牺牲一致性以换取其他特性。

某些架构师把这当成一场灾难，而笔者却将其视为系统设计中必须面对的权衡。此外，各个领域对“不一致”问题的容忍程度也不同，所以在决策时要考虑容忍度这一因素。

使用单服务器关系型数据库的人，对如何权衡一致性应该比较熟悉。在这种环境中，加强一致性的主要工具就是“事务”，它们能够提供较强的一致性保证。然而，事务系统通常具备放松“隔离级别”<sup>⊖</sup>（isolation level）的功能，以允许查询操作读取尚未提交的数据。在实际应用中，大多数应用程序都将一致性从最高的隔离级别（可序列化，serialized）往下调，以便提升性能。最常使用的隔离级别是“只能读取已提交数据”（read-committed transaction level），这样可以避免某些“读写冲突”，然而会导致另外一些冲突。

很多系统已经彻底弃用“事务”了，因为它们对性能的影响实在太大了。有两种不采用事务的使用方式。在数据规模较小的情况下，MySQL 比较流行，那时它还不支持事务处理。许多网站喜欢 MySQL 所带来的高速访问能力，并且不准备再使用事务了。在数据较多的情况下，像 eBay [Pritchett] 这种非常大的网站，为了让网站性能符合用户要求，它们必须弃用“事务”，尤其在需要引入分片机制时，更是如此。就算没有这些限制，很多应用程序构建者也需要同远程系统交互，而那些系统无法合理地容纳于事务范围之内，所以说，在企业级应用程序中，经常需要更新事务范围之外的数据。

## CAP 定理

在 NoSQL 领域中，我们通常认为“CAP 定理”是需要放宽一致性约束的原因。它最初由 Eric Brewer 在 2000 年提出 [Brewer]，并于数年之后为 Seth Gilbert 与 Nancy Lynch 所证明 [Lynch and Gilbert]。（这个定理也叫做“Brewer 猜想”，Brewer Conjecture。）

CAP 定理的基本表述是：给定“一致性”（Consistency）、“可用性”（Availability）、“分区耐受性”（Partition tolerance）<sup>⊖</sup>这三个属性，我们只能同时满足其中两个属性。显然，这在很大程度上取决于属性的定义，而且不同的看法会导致对“CAP 定理”实际效果的争论。

“一致性”一词的意思与前面讲过的几乎一样。而“CAP 定理”中的“可用性”则有其特殊含义，它的意思是，如果客户可以同集群中的某个节点通信，那么该节点就

---

⊖ 关于事务隔离及其级别的详情，请参阅：<http://zh.wikipedia.org/wiki/事务隔离>。——译者注

⊖ 此处的“分区”是数据库专用语，不是俗称的磁盘分区（disk partition）。——译者注

必然能够处理读取及写入操作。这与通常含义有微妙的差别，稍后将会详谈此问题。“分区耐受性”意思是，如果发生通信故障，导致整个集群被分割成多个无法互相通信的分区时（这种情况也叫“脑裂”，split brain，参见图 5.3），集群仍然可用。

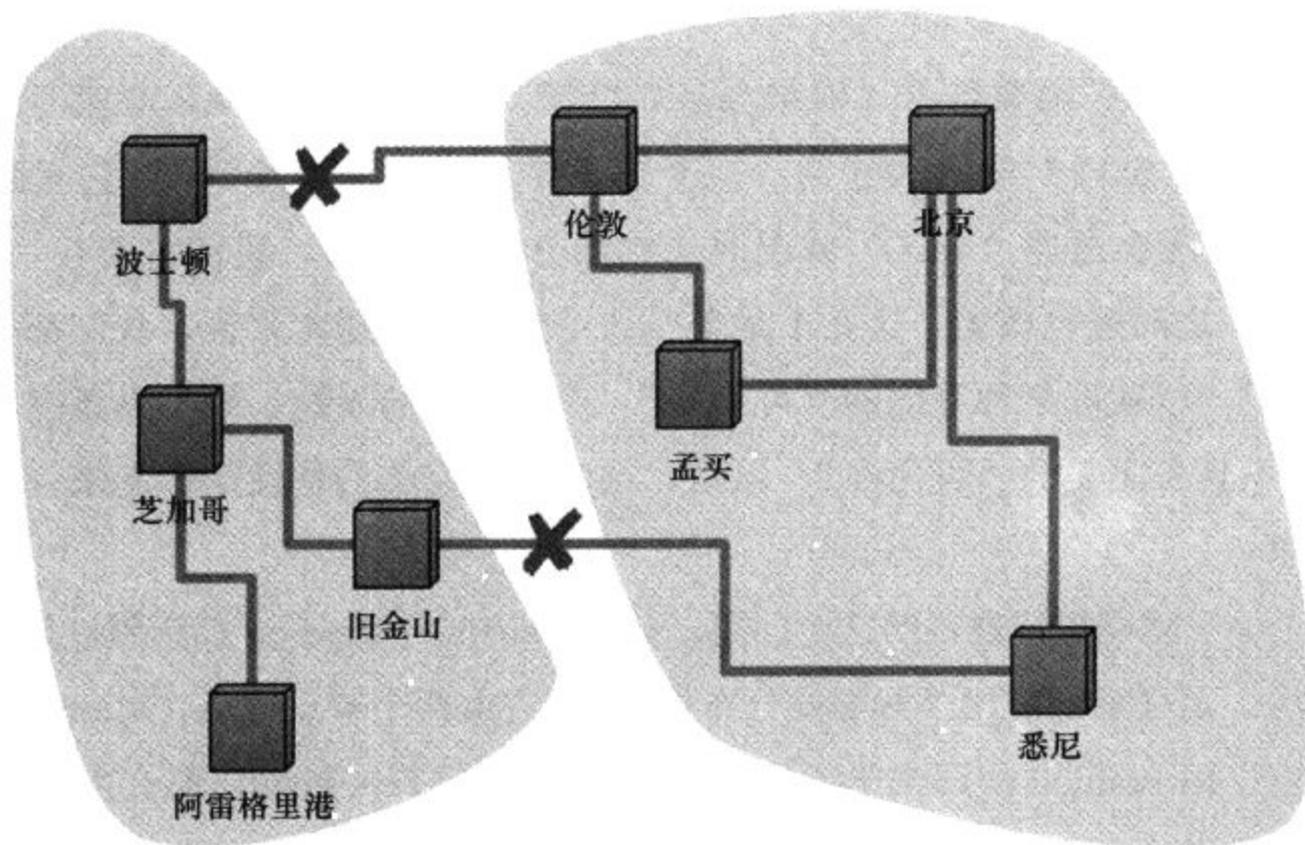


图 5.3 通信线路中有两处发生故障，将整个网络分为两组

单服务器系统显然是一种“CA 系统”，也就是具备“一致性”（Consistency）与“可用性”（Availability）， just 不具备“分区耐受性”的系统。一台电脑无法继续分割，所以不需要担心“分区耐受性”。由于只有一个节点，所以只要它正常运转，那么系统就可以使用。能正常运转且保持“一致性”的系统，是合理的。现实中的大多数关系型数据库都属于此种情况。

从理论上说，也存在“CA 集群”。然而，这意味着，一旦集群中出现“分区”，所有节点都将无法运转，如此一来，客户端就无法与任意一个节点通信了。按照“可用性”一词的常规定义来看，该系统此时缺乏“可用性”，然而如果按照“CAP 定理”中“可用性”一词的特殊含义来解释，则会令人困惑。“CAP 定理”将“可用性”一词定义为“系统中某个无故障节点所接收的每一条请求，无论成功或失败，都必将得到响应。” [Lynch and Gilbert]。所以按照这个定义来看，发生故障且无法响应客户请求的节点，并不会导致系统失去“CAP 定理”所定义的那种“可用性”。

这确实意味着你可以构建一个“CA 集群”，然而要保证它很少出现“分区”现象，

而且一旦出现此现象，所有节点必须全部停止工作。这是能够实现的，至少在一个数据中心内部是如此，然而实现它的代价通常高得令人无法承受。需要明白一点：为了使集群上某个“分区”内的全部节点都停止运转，需要实时检测是否发生“分区”状况，而这个操作可不是轻而易举的。

所以说，集群必须要容忍“网络分区”状况，而这正是“CAP定理”的意义所在。尽管“CAP定理”经常表述为“三个属性中只能保有两个”，但它实际上是在讲：当系统可能会遭遇“分区”状况时（比如分布式系统），我们需要在“一致性”与“可用性”之间进行权衡。这并不是个二选一的决定，通常来说，我们都会略微舍弃“一致性”，以获取某种程度的“可用性”。这样产生的系统，既不具备完美的“一致性”，也不具备完美的“可用性”，但是，这两种不完美的特性结合起来，却能够满足特定需求。

举一个例子就能更好地说明这个问题。假设 Martin 与 Pramod 都想预订某旅馆的最后一间客房，预订系统使用“对等式分布模型”，它由两个节点组成（Martin 使用位于伦敦的节点，而 Pramod 使用位于孟买的节点）。若要确保一致性，那么当 Martin 要通过位于伦敦的节点预订房间时，该节点在确认预订操作之前，必须先告知位于孟买的节点。实际上，两个节点必须按照相互一致的顺序来处理它们所收到的操作请求。此方案保证了“一致性”，但是假如网络连接发生故障，那么由故障导致的两个“分区”系统，就都无法预订旅馆房间了，于是系统失去了“可用性”。

改善“可用性”的一种办法是，指派其中一个节点作为某家旅馆的“主节点”，确保所有预订操作都由“主节点”来处理。假设位于孟买的节点是“主节点”，那么在两个节点之间的网络连接发生故障之后，它仍然可以处理该旅馆的房间预订工作，这样 Pramod 将会订到最后一间客房。在使用“主从复制”模型时，位于伦敦的用户看到的房间剩余情况会与孟买不一致，但是他们无法预订客房，于是就出现了“更新不一致”现象。然而在这种情况下，它就是应该如此。所以说，这种在“一致性”与“可用性”之间所做的权衡，也能正确处理上述特殊状况。

上述方案确实改善了状况，然而在发生网络故障时，如果负责旅馆客房预订的主节点位于孟买，那么处在伦敦的节点仍然无法预订客房。用“CAP定理”的术语来说，这就是“可用性”故障（availability failure），因为 Martin 可以和位于伦敦的节点通信，但是该节点却无法更新数据。为了继续提高“可用性”，我们也可以让两个“分区”系

统都接受客房预订请求，即使在发生网络故障时也如此。这么做带来的风险是，Martin 和 Pramod 有可能都订到了最后一间客房。然而，根据这家旅馆的具体运营情况，这也许不会出问题。通常来说，旅行公司都允许一定数量的超额预订，这样的话，如果有某些客人预订了房间而最终没有入住，那么就可以把这部分空余房间分给那些超额预订的人了。与之相对，某些旅馆总是会在全部订满的名额之外多留出几间客房，这样万一哪间客房出了问题，或者在房间订满之后又来了一位贵宾，那么旅馆可以把客人安排到预留出来的空房中。还有些旅馆甚至选择在发现预订冲突之后向客户致歉并取消此预订。这么做也说得通，因为该方案所付出的代价，要比因为网络故障而彻底无法预订的代价小。

允许“写入不一致”现象的一个经典示例，就是购物车，Dynamo 论文曾讨论过此事 [Amazon's Dynamo]。在此情况下，即使网络有故障，你也总是能够修改购物车中的商品。这么做有可能导致多个购物车出现。而结账过程则会将两个购物车合并，具体做法是，将两个购物车中的每件商品都拿出来，放到另外一个购物车中，并按照新的购物车结账。这个办法基本上不会出错，万一有问题，客户也有机会在下单之前先检视一下购物车中的东西。

我们在这个问题上的心得是：尽管大多数软件开发都将“更新一致性”看成一件天经地义的事情 (The Way Things Must Be)，但是，在某些情况下，就算不同的客户请求所得到的响应内容不一致，我们依然可以优雅地完成任务。这些情况与领域密切相关，只有懂得该领域内的知识，才能解决它们。因此，通常不能单纯依靠开发团队来解决此类问题，而是必须求助于领域专家 (domain expert)。如果能找到应对“更新不一致”问题的办法，那么就有更多的机会来提高“可用性”及性能了。对于前面提到的购物车来说，购物者必须要能够持续而便捷地购物才行。要想成为“购物达人”，这可是必备的神器哟！<sup>①</sup>

在处理“读取一致性”时，类似的逻辑也适用。如果正通过电脑交易软件来买卖“金融商品”<sup>②</sup>，那么也许不能接受任何未即时更新的数据。然而，若是正在一个媒体网

① 原文 “And as Patriotic Americans, we know how vital it is to support Our Retail Destiny”。大意是：“咱们都是赤诚的美国人，自然明白好的购物车对于‘促进零售业繁荣’这一伟大使命来说，是何等重要啊！”作者采用反讽而自嘲的口吻来调侃乐于购物的美国人。——译者注

② financial instrument，也称金融工具，是对债券、股票等金融合约的统称。详情参见：<http://zh.wikipedia.org/wiki/金融商品>。——译者注

站中发帖子，那么旧页面持续几分钟也没多大关系。

在这些情况下，需要知道用户对陈旧数据的容忍程度，以及“不一致窗口”的时长。一般会用平均长度、最差情况下的长度等指标来衡量“不一致窗口”，而且还要考虑不同时长的分布情况。对于不同的数据项来说，用户对陈旧数据的容忍程度也不同，因此，在采用“复制”技术配置数据库时，可能需要做出不同的设置。

NoSQL 的倡导者经常说，与关系型数据库所支持的 ACID 事务不同，NoSQL 系统具备“BASE 属性”（基本可用、柔性状态、最终一致性，英文是 Basically Available, Soft state, Eventual consistency）[Brewer]。尽管笔者觉得此处有必要提一下“BASE”这个首字母缩略词，不过它并不是十分有用。这个缩略词甚至比“ACID”还要做作，其中所说的“基本可用”及“柔性状态”，也没有明确定义。另外要说的一点是，Brewer 先生在引入“BASE”这一概念时，认为“ACID”与“BASE”不是非此即彼的关系，两者之间存在着多个逐渐过渡的权衡方案可选。

本书之所以要讨论“CAP 定理”，是因为在权衡分布式数据库的“一致性”时，经常会提到（并滥用）它。然而，与其考虑如何权衡“一致性”与“可用性”，不如思考怎样在“一致性”与“延迟”（latency）之间取舍。在讨论分布式系统的“一致性”问题时，通常可以概括地说：参与交互操作的节点越多，“一致性”就越好。然而问题是，每新增一个节点，都会使交互操作的响应时间变长。“可用性”可以视为能够忍受的最大延迟时间，一旦延迟过高，我们就放弃操作，并认为数据不可用，这样一来，就和“CAP 定理”对“可用性”所下的定义相当吻合了。

## 5.4 放宽“持久性”约束

上面已经讲了“一致性”，在谈到数据库事务的“ACID”属性时，大家最常说起的就是它。“一致性”的关键在于，将请求序列化，使之成为原子的（Atomic）、相互隔离的（Isolated）“工作单元”（work unit）。然而，大部分人对放宽“持久性”不屑一顾，他们在想：失去了更新能力的数据库究竟有什么用呢？

事实上，在某些场合，我们也可以牺牲一些“持久性”以换取更好的性能。如果某个数据库大部分时间都在内存中运行，更新操作也直接写入内存，并且定期将数据变更写回磁盘，那么，它就可以大大提高响应请求的速度了。这种做法的代价在于，

一旦服务器发生故障，任何尚未写回磁盘的更新数据都将丢失。

比方说，上述权衡方案在保存用户会话状态时就值得考虑。大型网站可能会有很多用户来访问，而网站要将每个用户正在做的事情作为临时信息，以某种“会话状态”的形式保存起来。这种状态下会有大量活动，要产生大量请求，这些都将影响网站的响应能力。此处的关键问题在于：会话数据就算丢了也无所谓，因为这只不过是个小麻烦而已，与整个网站访问速度都变慢相比，它造成的损失要小很多。这时可以考虑“非持久性写入操作”（nondurable write）。通常来说，我们可以在每次发出请求时，指定该请求所需的持久性。这样的话，就可以把某些极为重要的更新操作立刻写回磁盘了。

另外一个可以放宽持久性约束的例子，就是捕获物理设备的遥测数据（telemetric data）。在这种情况下，就算最近的更新数据可能会因为服务器发生故障而丢失，我们也还是选择把快速捕获数据放在首位。

还有一类对“持久性”的权衡，是由“复制数据”（replicated data）所引发的。如果一个节点处理完更新操作之后，在更新数据尚未复制到其他节点之前，就出错了，那么则会发生“复制持久性”（replication durability）故障。有一种比较简单的情况可以说明此事：假设有一个采用“主从式分布模型”的数据库，在其主节点出错时，它会自动指派一个从节点作为新的主节点。若主节点真的发生故障，则所有还未复制到其他副本的写入操作就都将丢失，然而一旦主节点从故障中恢复过来，那么，该节点上的更新数据就会和发生故障这段时间内新产生的那些更新数据相冲突。我们把这视为一个“持久化”问题，因为主节点既然已经接纳了这个更新操作，那么用户自然就会认为该操作已经顺利执行完，但实际上，这份更新数据却因为主节点出错而丢失了。

如果你有把握能在主节点出错之后迅速将其恢复，那么可以考虑不将某个从节点指派为新的主节点。另一种办法是，确保主节点在收到某些副本对更新数据的确认之后，再告知用户它已接纳此更新。然而，这么做无疑会拖慢更新速度，而且，一旦从节点也发生故障，那么整个集群就无法使用了。所以说，在这种情况下，还是得根据“持久性”的重要程度来权衡。与处理“持久性”的基本手段类似，我们也可以针对单个请求来指定其所需的持久性。

## 5.5 仲裁

“一致性”与“持久性”之间的取舍，并不是一个非此即彼的议题。处理请求所用的节点越多，避免“不一致”问题的能力就越强。这自然就引出一个问题：要想保证“强一致性”（strong consistency），需要使用多少个节点才行？

假设某份数据需要复制到三个节点中。为了保证“强一致性”，不需要所有节点都确认写入操作，只需要其中两个节点（也就是超过半数的节点）确认就可以了。在这种情况下，如果发生两个相互冲突的写入操作，那么只有其中一个操作能为超过半数的节点所认可。这就是“写入仲裁”（write quorum），如果用稍微正规一些的方式说，那就是  $W > N/2$ 。这个不等式的意思是，参与写入操作的节点数（ $W$ ），必须超过副本节点数（ $N$ ）的一半。副本个数又称为“复制因子”（replication factor）。

与“写入仲裁”相似，还有一个概念叫做“读取仲裁”（read quorum），也就是说，想要保证能够读到最新数据，必须与多少个节点联系才行？“读取仲裁”稍微有点复杂，因为它取决于确认写入操作所需的节点数。

现在考虑“复制因子”为3的情况。假设写入操作需要两个节点来确认（ $W=2$ ），那么我们至少得联系两个节点，才能保证获取到最新数据。然而，假如某些写入操作只被一个节点所确认（ $W=1$ ），那么我们就必须和3个节点都通信一遍，才能确保获取到的数据是最新的。在这种情况下，由于写入操作没有获得足够的节点支持率，所以可能会产生更新冲突。但是，只要从足够数量的节点中读出数据，就一定能侦测出此类冲突。因此，即便在写入操作不具备“强一致性”的情况下，也可以实现出具有“强一致性”的读取操作来。

执行读取操作时所需联系的节点数（ $R$ ），确认写入操作时所需征询的节点数（ $W$ ），以及复制因子（ $N$ ）这三者之间的关系，可以用一个不等式来表述。那就是，只有当  $R+W > N$  时，才能保证读取操作的“强一致性”。

上述两个不等式都适用于“对等式分布模型”。如果使用“主从式分布模型”，那么只需要向主节点中写入数据，就可以避免“写入冲突”了。与之类似，若想避免“写读冲突”，只需从主节点中读取数据就好。在这种环境下，集群中的节点数经常容易与“复制因子”相混淆，实际上，这两个概念不同。比方说，一个集群有100个节点，然而其“复制因子”可能仅仅是3，因为大部分数据都分布在各个“分片”之中。

实际上，很多权威说法都提议：将“复制因子”设为3，就可以获得足够好的“故障恢复能力”了。这时，如果只有一个节点出错，那么仍然能够满足读取与写入所需的最小法定节点数。若是有自动均衡（automatic rebalancing）机制，那么用不了多久，集群中就会建立起第三个副本，由此看来，在替代副本建立好之前，再次发生副本故障的概率很小。

参与某个操作的节点数，可能会随着该操作的具体情况而改变。在写入数据时，根据“一致性”与“可用性”这两个因素的重要程度，有一些更新操作可能需要获得足够的节点支持率才能执行，而另外一些则不需要。与之相似，某些读取操作可能更看中执行速度，而可以容忍过时数据，此时，它就可以少联系几个节点。

通常要将两方面因素都考虑进来。假设需要快速且具备“强一致性”的读取操作，那么写入操作就要得到全部节点的确认才行，这样的话，只需联系一个节点，就能完成读取操作了（ $N=3$ ， $W=3$ ， $R=1$ ）。这个方案意味着，写入操作会比较慢，因为它们必须得到全部三个节点确认之后，才能执行，而且此时连一个节点都不能出错。实际上，在某些情况下，还是需要在“一致性”与“可用性”之间权衡的。

之所以要讲这些内容，是为了让读者明白，在“一致性”与“可用性”的权衡问题上，存在很多选项，需要根据每一种方案的优缺点来选出一种适合自己的办法。某些 NoSQL 技术文章的写作者认为，在“一致性”与“可用性”之间存在较为简单的权衡方案，而笔者的意思则是，大家应该了解，实际情况要比上述说法更为灵活，也更为复杂。

## 5.6 延伸阅读

因特网上可以找到许多有趣的博客与论文，它们都在谈论分布式系统中的“一致性”。不过笔者认为，其中最有帮助的资料还是要数 [Tanenbaum and Van Steen] 一书。该教材精彩地讲解了分布式系统的许多基础知识，如果在学完本章之后还想深入研究，那么最好去看看那本书。

本书刚脱稿时，《IEEE Computer》杂志发行了一期特刊 [IEEE Computer Feb 2012]，其中谈到了对业界影响越来越大的“CAP 定理”。这份资料有助于进一步阐明此话题。

## 5.7 要点

- 当两个客户端试图同时修改一份数据时，会发生“写入冲突”。而当某客户端在另一个客户端执行写入操作的过程中读取数据时，则会发生“读写冲突”。
- 悲观方式以锁定数据记录来避免冲突，而乐观方式则在事后检测冲突并将其修复。
- 在分布式系统中，如果某些节点收到了更新数据，而另外一些节点却尚未收到，那么这种情况就视为“读写冲突”。若写入操作已经传播至所有节点，则此刻的数据库就具备“最终一致性”。
- 客户端通常希望具备“照原样读出所写内容”的一致性，也就是说，客户在执行完写入操作之后，要能够立刻看到新值。如果读取与写入操作分别发生在不同节点，那么想保证这一点会比较困难。
- 想取得较好的“一致性”，就要用许多节点来执行数据操作，而这又会增大延迟。所以说，经常需要在“一致性”与延迟之间权衡。
- “CAP定理”宣称，当有可能发生“网络分区”现象时，必须在数据的“可用性”与“一致性”之间权衡。
- 可以舍弃一部分“持久性”以减小延迟，如果想让数据库在复制数据出错的情况下依然可用，那么更应该考虑这种权衡方式了。
- 在采用“复制”技术的分布式模型中执行数据操作时，无需联系所有副本，只要为足够多的副本所认可，就能保持“强一致性”了。

## 第 6 章

# 版本戳

许多人批评 NoSQL 数据库不支持“事务”。事务是个有用的工具，可以帮助程序员维护数据的“一致性”。对不支持“事务”这一点，好些 NoSQL 的支持者并不担心，因为面向聚合的 NoSQL 数据库确实支持聚合内部的原子操作，以聚合为单元来更新数据，是比较自然的。即便如此，在选择数据库时，依然应该把是否支持“事务”这一因素考虑在内。

说到这个问题时，一定要明白，“事务”也有其局限。在“事务系统”中，依然会有那种需要人工干预的更新操作，而且有些更新操作通常无法封装在一个“事务”之内，因为那会导致“事务”的打开时间过长。这些问题都可以用“版本戳”（version stamp）来应对，而且此技术在其他情况下也很好用，尤其是从“单服务器分布模型”（single-server distribution model）迁移到多服务器时，更是如此。

## 6.1 “商业事务”与“系统事务”

即便是那种构建于“事务型数据库”（transactional database）之上的系统，也经常需要在不使用“事务”的前提下确保“更新一致性”。用户在说“事务”一词时，通常指的是“商业事务”（business transaction）。比如说，用户浏览产品目录，选中了一瓶价格很实惠的 Talisker 威士忌，填入信用卡信息，然后确认订单，这些都是“商业事务”。然而，上述操作通常不会发生在数据库的“系统事务”（system transaction）中，因为那样做必须锁住数据库中各个元素才行，而在那段时间里，用户可能会去找信用卡，也可能被同事叫去吃午饭。

应用程序通常只在处理完用户交互操作之后才开始“系统事务”，这样的话，锁定时间就比较短了。然而问题在于，当需要计算和决策的时候，数据有可能已经改动了。价格表上 Talisker 威士忌的售价也许已经变了，或是某人可能会修改客户的地址，从而导致运费改变。

宽泛地说，处理这种问题可以采用“离线并发”（offline concurrency）技术 [Fowler PoEAA]，这适用于 NoSQL 环境下。“乐观离线锁”（Optimistic Offline Lock）[Fowler PoEAA] 是一种特别有用的方式，它是“条件更新”（conditional update）的一种形式，客户端执行操作时，将重新读取“商业事务”所依赖的信息，并检测该信息在首次读取之后是否一直没有变动，若一直未变，则将其展示给用户。实现此技术有个好办法，那就是保证数据库中的记录都有某种形式的“版本戳”（version stamp）。版本戳是一个字段，每当记录中的底层数据改变时，其值也随之改变。读取数据时可以记下版本戳，这样的话，在写入数据之前，就可以先检查一下数据版本是否已经变了。

在通过 HTTP 协议更新资源时 [HTTP]，可能也会用到这种技术。其中一种实现方式就是“etag”。不论何时获取资源，服务器总会在响应信息的头部（header）放置一个“etag”，它是个“无明显意义的字符串”<sup>⊖</sup>（opaque string），只是用来标识资源的版本而已。稍后如果想更新此资源，那么可以采用“条件更新”形式，把上次通过 GET 请求获取的 etag 提交给服务器。若服务器上的相应资源已经改变，那么提交的

---

⊖ 是一种没有明确定义在编程接口中的字符串，其具体格式、具体实现形式不为一般用户所知。详情参见：[http://en.wikipedia.org/wiki/Opaque\\_data\\_type](http://en.wikipedia.org/wiki/Opaque_data_type)。——译者注

“etag”就与服务器的“etag”不匹配了，这样的话，会得到一个状态码为412（先决条件未满足，Precondition Failed）的响应。

某些数据库也提供了一种类似的“条件更新”机制，保证不会在陈旧数据上执行更新操作。开发者也可以自己来执行这项检测，不过要确保在读取及更新资源的过程中，不要有其他线程修改此资源。（这种操作有时也称“compare-and-set操作”，“compare-and-set”一词缩写为“CAS”，中文意思为“比较并设置”），它得名于处理器中的“CAS操作”。两者的区别在于，处理器CAS在设置之前，比较的是值本身，而数据库条件更新比较的却是值所对应的版本戳。）

构建版本戳有很多办法。可以使用计数器，每当资源更新时，就把它的值加1。计数器很有用，因为我们根据它的值很容易就能看出哪个版本比较新。而另一方面，这需要服务器来生成该值，并且要有一个主节点来保证不同版本的计数器值不会重复。

还有一种办法是创建GUID<sup>①</sup>，也就是一个值很大且保证唯一的随机数。可以将日期、硬件信息，以及其他一些随机出现的资源组合起来以构建此值。GUID的好处是任何人都可以生成，不用担心重复，而其缺点则是数值比较大，而且无法通过直接比较来判断版本的新旧。

第三个办法就是根据资源内容生成哈希码（hash）。只要哈希键足够大，那么“内容哈希码”（content hash）就可以像GUID那样全局唯一，而且任何人都可以来生成它。此方法的好处在于，哈希码的内容是确定的：只要资源数据相同，那么任何节点生成的“内容哈希码”都是一样的。但是，它们与GUID一样，都无法通过直接比较来看出版本新旧，而且比较冗长。

第四种办法是使用上一次更新时的时间戳（timestamp）。与计数器一样，它们也相当短小，而且可以直接通过比较其数值来确定版本先后，然而，这种办法有一个地方比计数器好，那就是，它们不需要由主节点来生成。时间戳可以由多台计算机生成，不过，它们的时钟必须同步。如果某个节点的时钟出错了，那么可能会导致各种数据损毁（data corruption）现象。使用时间戳还有一个风险，那就是，若精确度过低，则可能重复：假如每毫秒都要更新很多次的话，那么将时间戳的精确度设为毫秒是不够的。

---

① 该词是 Globally Unique Identifier 的缩写，中文称“全局唯一标识符”，其意义及生成方式可参见：[https://en.wikipedia.org/wiki/Globally\\_unique\\_identifier](https://en.wikipedia.org/wiki/Globally_unique_identifier)。——译者注

可以把几种时间戳生成方案的优点融合起来，同时使用多种手法创建一个“复合版本戳”（composite stamp）。举例来说，CouchDB<sup>Ⓐ</sup>创建版本戳时，使用了计数器与“内容哈希码”。大部分情况下，只要比较版本戳就可以看出两个版本的新旧了，万一碰到两个节点同时更新数据的情况，那么立刻就能发现冲突，因为两个版本戳的计数器相同，而“内容哈希码”却不同。

除了可以避免“更新冲突”之外，版本戳也有助于维护“会话一致性”（参见 5.2 节）。

## 6.2 在多节点环境中生成版本戳

如果只有一个权威数据源（authoritative source for data），比如采用单服务器或“主从式复制模型”，那么使用基本的版本戳生成方案就好。在这种情况下，由主节点负责生成版本戳，而从节点必须使用主节点的版本戳。但是，在“对等式分布模型”中，这套版本戳生成机制就必须改进，因为没有统一设置版本戳的地方了。

如果向两个节点索要同一份数据，那么有可能获得不同的答案。一旦发生此现象，我们可以根据导致此差异的原因来采取相应措施。有可能是更新操作已经通知给其中一个节点了，而另外一个节点尚未收到通知，这种情况下你可以选用最新的数据（假设有办法分辨两份数据的新旧）。还有一种可能，那就是发生了“更新不一致”现象，此时你需要决定如何处理此问题。在这种情况下，仅凭简单的 GUID 或 etag 是不够的，因为只依靠它们无法判断出数据之间的关系。

最简单的版本戳就是计数器。节点每次更新数据时，都将它加 1，并把其值放入版本戳中。假设某主节点有两个副本，我们用“蓝色”和“绿色”来区分这两个从节点。如果在蓝色节点所给出的应答数据中，版本戳为 4，而绿色节点的版本戳是 6，那么绿色节点上的数据就比较新。

如果有多个主节点的话，那么就动一些脑筋了。一种办法是像“分布式版本控制系统”那样，确保所有节点都有一份“版本戳记录”（version stamp history）。这样的话，就可以判断出蓝色节点给出的应答数据是不是绿色节点所给数据的“祖先”<sup>Ⓑ</sup>。如果

---

Ⓐ 创建于 2005 年的开源 NoSQL 数据库，详情参见：<https://en.wikipedia.org/wiki/CouchDB>。官方网站是：<http://couchdb.apache.org/>。——译者注

Ⓑ 这里“祖先”的意思是，蓝色节点中的旧数据，是绿色节点中新数据所依据的范本，或者说绿色节点中的新数据，继承并修订了蓝色节点中的旧数据，而不是日常用语所说的“祖先”。——译者注

想实现这一点，要么让客户端保存“版本戳记录”，要么由服务器节点来维护此记录，并且把它放在应答数据中，传给客户端。用“版本戳记录”可以检测出数据“不一致”现象：如果两份应答数据中的版本戳都无法在对方的“版本戳记录”中找到，那么就可以判定发生了“不一致”问题。虽说“版本控制系统”都会存留此类记录，但是 NoSQL 数据库中却没有这种东西。

有一种简单但是可能会出问题的办法，那就是使用“时间戳”。主要问题在于，通常很难确保所有节点的时间都一致，尤其是在更新比较频繁时。万一其中某个节点的时钟没有同步，那么就会引发各种麻烦。此外，“时间戳”也无法检测“写入冲突”，所以说，它只能在“单一主节点”（single-master）的环境中正常运作，而那种情况下，采用计数器通常更好。

“对等式 NoSQL 数据库系统”最常使用的一种版本戳形式，叫做“数组式版本戳”<sup>⊖</sup>（vector stamp）。实际上，“数组式版本戳”就是由一系列计数器组成的，每个计数器都代表一个节点。假设某“数组式版本戳”中有三个节点（分别记为“蓝色”（blue）、“绿色”（green）、“黑色”（black）），那么它的写法就类似 [blue : 43, green : 54, black : 12] 这样。每当节点执行“内部更新”（internal update）操作时，就将其计数器加 1，所以，假设绿色节点执行了一次更新操作，那么现在这个“数组式版本戳”就成了 [blue : 43, green : 55, black : 12]。只要两个节点通信，它们就同步其“数组式版本戳”。具体的同步方式有很多种。笔者造了“数组式版本戳”这个术语，并在整本书中使用它。读者也许还会听到“数组式时钟”（vector clock）或“版本号数组”（version vector）等叫法，它们都是“数组式版本戳”的特定形式，只不过其同步方式不同。

使用此方案，我们就能辨别某个“数组式版本戳”是否比另外一个新，因为新版本戳中的计数器总是大于或等于旧版本戳。比如，[blue : 1, green : 2, black : 5] 就比 [blue : 1, green : 1, black : 5] 新，因为前者之中有一个计数器比后者大。若两个版本戳中都有一个计数器比对方大，那么就发生了“写入冲突”，比如 [blue : 1, green : 2, black : 5] 与 [blue : 2, green : 1, black : 5]。

数组中可能缺失某些值，我们将其视为 0。这样一来，就可以用 [blue : 6, black : 2]

⊖ 此处 vector 一词不采用常见译法“向量”或“矢量”，以免与该词的其他含义混淆。——译者注

来表示 [blue: 6, green: 0, black: 2] 了。于是，不需要弃用现有的“数组式版本戳”，就可以向其中轻易新增节点了。

“数组式版本戳”是一种能够侦测出“不一致”现象的有用工具，然而它们无法解决此问题。要想解决冲突，就得依赖领域知识。在“一致性”与延迟之间权衡时，也要考虑到这一点。如果偏向“一致性”，那么系统在出现“网络分区”现象时就无法使用；反之，若要减少延迟，则必须自己检测并处理“不一致”问题。

### 6.3 要点

- 版本戳可用来检测并发冲突。读取并更新某份数据之后，可检测其版本戳，以确保在读取和写入操作之间，没有其他人更新过此数据。
- 版本戳可以用计数器、GUID、“内容哈希码”、时间戳等方式来实现，也可以将上述几种方式组合起来。
- 在分布式系统中，可以采用“由版本戳构成的数组”（a vector of version stamps）来检测不同节点之间是否发生了“相互冲突的更新操作”（conflicting update）。

## 第 7 章

# 映射 - 化简

“面向聚合的数据库”能够兴起，很大程度上是由于集群的增长。运行在集群环境中的数据库对数据存储问题的权衡方式与单机环境有所不同。集群不仅改变了数据存储的规则，而且改变了数据计算的规则。如果要把大量数据放在集群中，那么为了有效处理数据，你必须以另外一种思路来考虑如何安排数据处理流程。

如果使用“集中式数据库”（**centralized database**），那么通常有两种方式来处理计算逻辑：一种是在数据库服务器上，另一种是在客户端计算机上。如果将计算放在客户端计算机中执行，那么选择编程环境时就更加灵活了，这可以让开发者更为容易地创建或扩展程序。这么做的缺点是，要把大量数据从数据库服务器搬到客户端。如果数据很多，那么可以考虑放在数据库服务器端来处理，这么做的代价是：编程环境不如客户端方便，而且加重了数据库服务器的负载。

把数据库放到集群之后，立刻就带来一个好处，那就是可以把运算工作分布到多台计算机中去。然而，此时仍要试着减少通过网络传输的数据量，把某个节点所需的数据尽可能多地放在该节点上执行。

“映射 - 化简模式”（**map-reduce pattern**，“分散 - 聚集模式”（**Scatter-Gather**）的一种形式 [Hohpe and Woolf]）是一种安排数据处理流程的手段，可以利用集群中的多台计算机，同时又能将某台计算机所需的数据及处理工作尽量放在本机执行。它起初因谷歌的 **MapReduce** 框架 [Dean and Ghemawat] 而成名。虽说很多数据库都有各自的实现方式，不过使用最广的还是 **Hadoop** 项目中所包含的开源实现。与大多数模式一样，这些实现之间存在细节差异，所以本书着重讲述其一般概念。顾名思义，“映射 - 化简”（**map-reduce**）一词来源于“函数式编程语言”（**functional programming language**）对集合（**collection**）的“映射”（**map**）与“化简”（**reduce**）操作。

## 7.1 基本“映射-化简”

为了解释“映射-化简”的基本思路，我们还是不厌其烦地举出那个老例子，也就是“客户与订单”。假设数据库选用订单作为聚合单元，每个订单中包含商品项（line item），而每个商品项中又有产品 ID、数量及价格。以订单为聚合单元是合理的，因为大家通常需要一次访问整个订单。由于订单数量大，所以要将其数据集“分片”存放于多台电脑中。

然而，销售分析人员需要查看某个商品 7 天内的总销售额（total revenue）。此类数据报表与现有聚合结构不符——我们由此看到了使用聚合的缺点。为了获取产品销售额报表（product revenue report），必须访问集群中的每台电脑，并查看其上的多条记录才行。

这种情况正好可以用“映射-化简”来解决。“映射-化简”操作的第一步就是“映射”，它是一个函数。其输入值是某个聚合，而输出值则是一大把键值对。在本例中，输入值可以是订单，而输出值则是与订单中产品项相对应的键值对。每一个键值对中，都有一个用作其键的产品 ID，还嵌有一张包含产品数量及总价的数值表（如图 7.1 所示）。

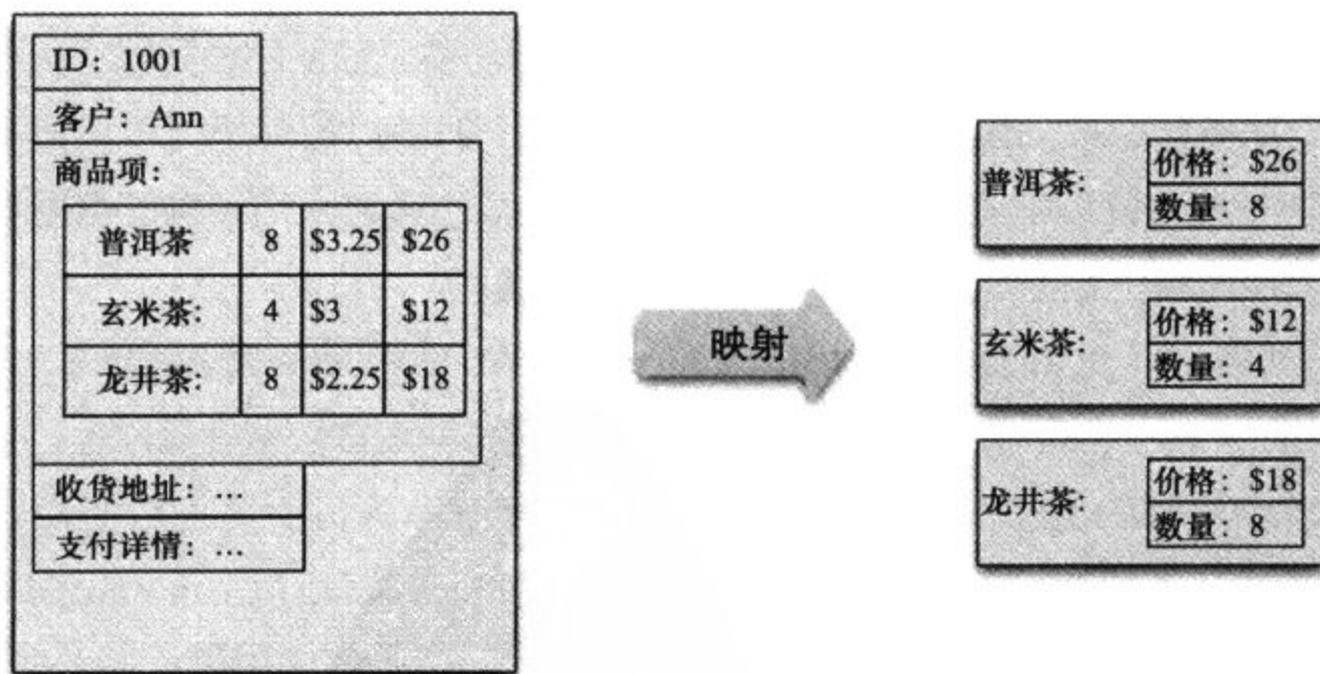


图 7.1 从数据库中读取记录并将其打散为键值对的“映射函数”

每个应用程序的映射函数（map function）都各自独立，以便能够安全地执行并发运算。这样一来，“映射-化简框架”（map-reduce framework）就可以在每个节点上高效地创建“映射任务”（map task），并随意指派一份订单让其运算了。

此类框架可以利用并发能力访问大量数据，并将数据运算局限在各自节点内执行。比方说，本例中只是选取了记录里的某个值，然而，我们也完全可以随意将某个复杂的函数用作映射操作的一部分，只要此函数所依赖的数据都位于某个聚合内部就好。

映射操作只限于一条记录；而“化简函数”（reduce function）则可以接受多个关键字相同的映射操作输出值作为其输入参数，然后将之合并。因此，某个“映射函数”可能会从所有与“Databases Refactoring”相关的订单中提取出1000个商品项；然而“化简函数”却可以把每个商品项的销售量与销售额汇总，将所有商品项合并为1条数据。“映射函数”仅能操作于某个聚合内部的数据，而“化简函数”则可以操作所有具备同一关键字的数据（如图7.2所示）。



图 7.2 使用“化简函数”，将具备相同关键字的多个键值对汇聚成一个

“映射 - 化简”框架会将所有文档数据的映射任务安排在合适的节点上执行，并把映射后的数据移交给“化简函数”。为了让“化简函数”编写起来更加容易，“映射 - 化简”框架会将所有键值对收集起来，把相同关键字下的数值汇聚成集合，并以此关键字与数值集合为参数，调用“化简函数”。因此，要执行“映射 - 化简”任务，只需编写这两个函数即可。

## 7.2 分区与归并

在最简单的情况下，可以把“映射 - 化简”任务看成只有一个“映射函数”的操作：将不同节点中所有“映射任务”的输出值都连接起来，并传递给“化简函数”即可。这样做确实可行，然而我们可以设法提升该操作的并发能力并减少数据传输量（如图7.3所示）。

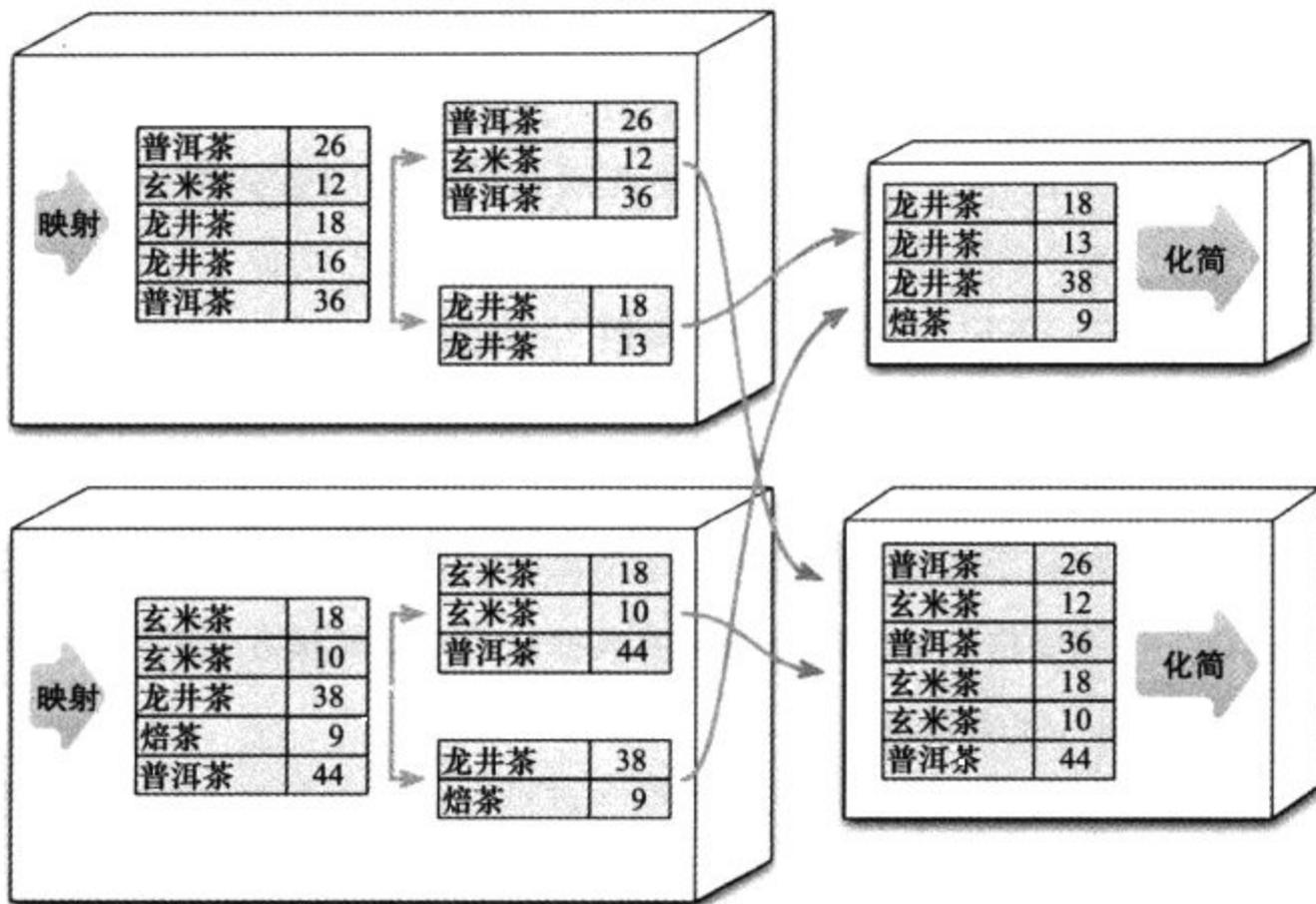


图 7.3 “分区”之后，多个“化简函数”就可以并发化简不同关键字所对应的数据了

首先要将“映射函数”的输出数据分区，以提升并发处理能力。每个“化简函数”都仅能操作具备相同关键字的一组结果。这是一个限制，它意味着化简函数的参数不能有多关键字；然而它也有好的一方面，那就是可以并发执行多个化简函数。为了发挥并发执行的优势，我们需要根据每个节点所处理的关键字，把映射结果分割开。一般情况下，我们把若干个关键字划分到同一“分区”中。框架将所有节点中应该归入某个“分区”的数据拿出来，把它们合并为一组，放入那个“分区”里，并将其发送给“化简函数”。这样的话，多个“化简函数”就可以在各个“分区”中并发执行了，最后将其结果合并起来就好。（这一步也叫做“混排”（shuffling），“分区”一词有时也称“存储区”（bucket）或“区域”（region）。）

接下来的问题是：在由映射阶段进入化简阶段时，如何减少传输的数据量。大部分数据都是重复的，因为它们都是关键字相同的多个键值对。“归并函数”（combiner function）可以把具备同一关键字的所有数据合并为一个值（如图 7.4 所示）。从本质上说，“归并函数”就是一种“化简函数”，而且实际上，“归并函数”在很多情况下确实能够充当最后的“化简函数”。“化简函数”若想用作“归并函数”，还需具备一个特性：输出值必须与输入值的形式相匹配。满足此特性的“化简函数”称为“可用作归

并函数的化简函数” (combinable reducer)。

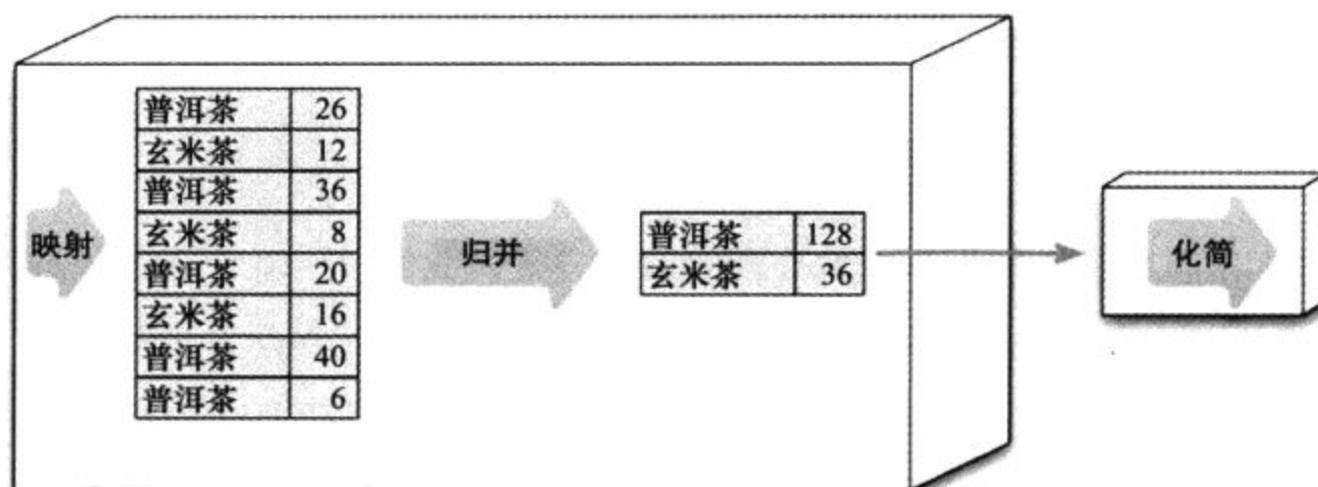


图 7.4 在通过网络传送数据之前，先将其归并，以减少传输量

并非所有“化简函数”都“可用作归并函数” (combinable)。比如一个函数需要统计购买某产品的客户数量（同一客户多次购买某产品，不重复计算）。该操作的“映射函数”需要生成包含产品与客户的数据。而这个“化简函数”则要将刚才的数据合并起来，计算某个产品有多少位顾客购买，并生成包含产品与购买次数的数据（如图 7.5 所示）。此“化简函数”的输出值与其输入值不同，因而不能用作“归并函数”。此时仍然可以使用“归并函数”，不过它只是消除重复的“产品 - 客户”键值对而已，与最终的“化简函数”并不相同。

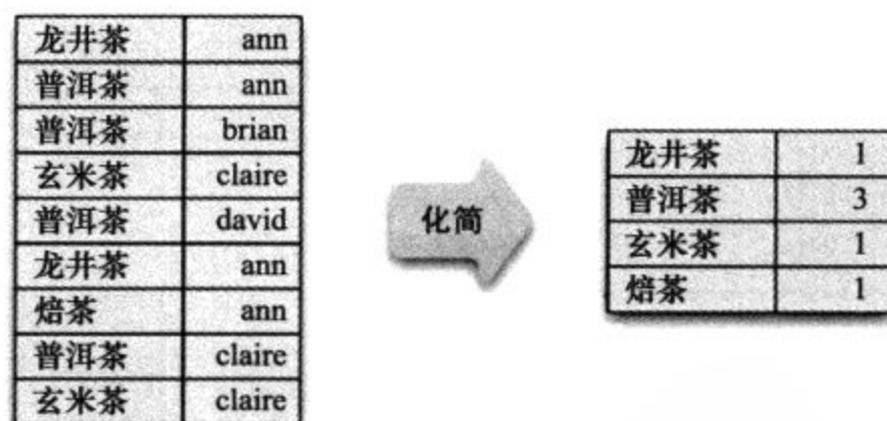


图 7.5 这个“化简函数”用于统计购买某种茶叶的客户数量，它不能用作“归并函数”

如果有了“可用作归并函数的化简函数”，那么“映射 - 化简框架”不仅可以安全地执行并发操作（同时化简多个分区），而且可以在不同的时间和地点相继化简同一分区内的数据。除了可以在传输数据之前先于某个节点内归并数据之外，我们也可以在“映射函数”还未执行完毕时就开始归并。这就让“映射 - 化简”过程变得更加灵活。某些“映射 - 化简”框架要求所有“化简函数”都必须能当“归并函数”来用，这么

做最为灵活。如果想在这些框架中使用不能完成归并功能的“化简函数”，那么需要把整个“映射 - 化简”过程分为几个步骤来做。

### 7.3 组合“映射 - 化简”计算

“映射 - 化简”是一种思考并发处理问题的方式，我们可以将计算过程组织为一个相对直观模型，然而为了使计算工作可以在集群中的各个节点上并发执行，又需要降低模型的灵活程度。由于这个问题需要权衡，所以计算过程要受限制。在“映射任务”中，只能操作同一聚合内的数据；而在“化简任务”中，只能操作具有同一关键字的数据。这意味着在规划程序结构时，必须用两种不同的思路来处理这两个问题，以便运算过程能够遵从这些限制。

有一个简单的限制，那就是计算操作的结构必须与化简操作的思路搭配得当。计算平均值操作能很好地说明此问题。假设我们使用本书范例中一直在用的那种订单格式。现在，要计算出每个产品在包含该产品的订单中，其平均购买量是多少。平均值操作的一个重要属性在于，它们不能简单地组合起来（not composable），也就是说，如果有两组订单，那么仅仅将其各自的平均购买量相加，无法正确得出该产品总的平均购买量。必须把两组订单数据中，该产品的总数及订单总数归并，用合并后的总销售数除以订单数，才能算出正确的平均值（如图 7.6 所示）。

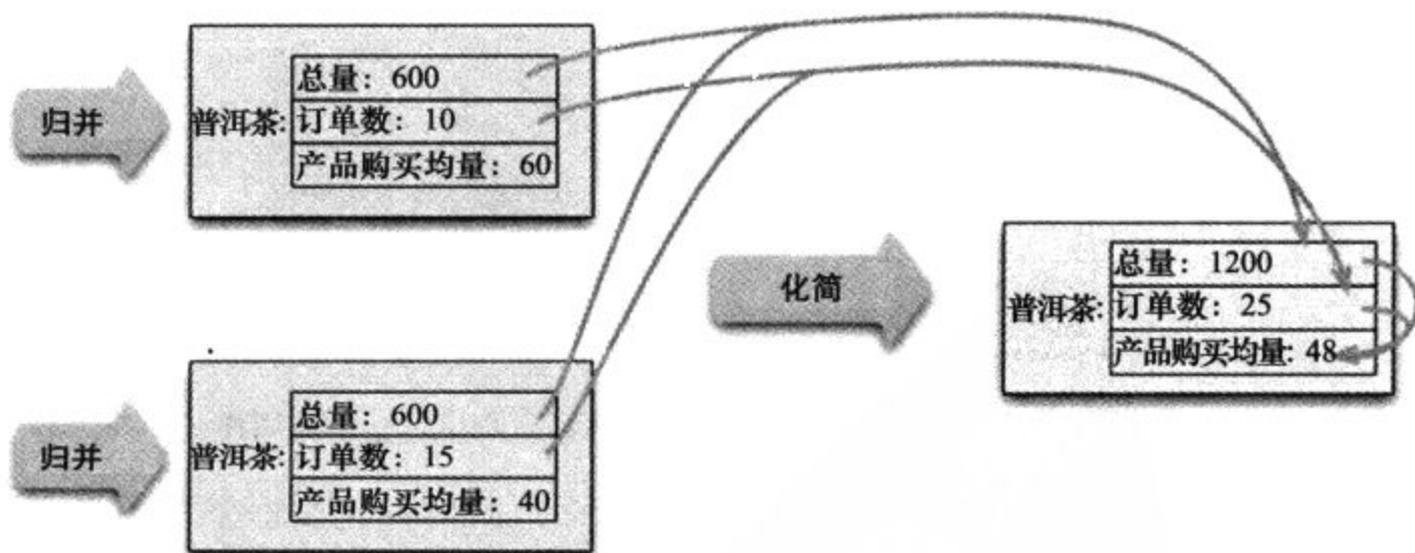


图 7.6 在计算平均值时，购买总量与订单数可以在化简计算过程中合并，然而平均值必须用归并之后的总量和订单数才能算出

寻找优雅的化简算法时所用的思路，也可以套用在计数工作上。在统计时，“映射函数”可以将“订单数”字段设为 1，这样只需要把各自的产品销量加起来，就可以得

到总销量了（如图 7.7 所示）。

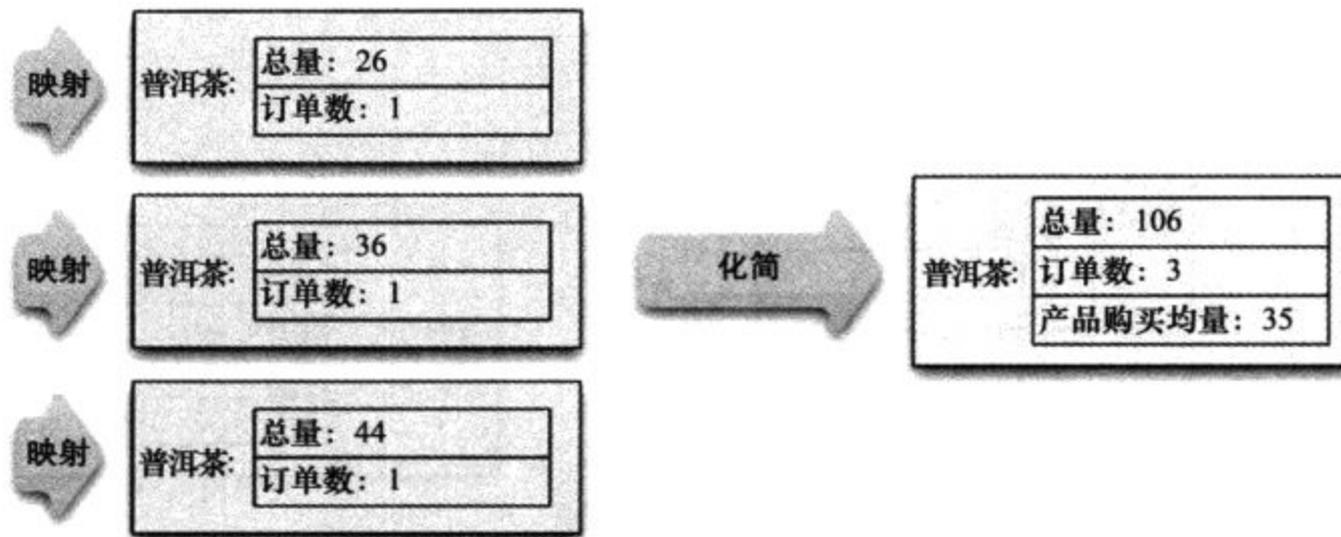


图 7.7 在统计时，每个“映射函数”都将订单数设为 1，这样只需将各自商品数相加即可得出总销量

### 7.3.1 举例说明两阶段“映射 - 化简”

如果“映射 - 化简”计算比较复杂，那么可以使用“管道及过滤器”（pipes-and-filters），也就是将某一阶段的输出数据作为下一阶段的输入数据。这很像 UNIX 系统的“管道”（pipeline）。

举个例子，现在要将某产品 2011 年每个月的销量与上一年同月份进行比较。为了完成此任务，需要将计算过程分解成两个阶段。第一阶段产生的聚合记录，表示某商品在一年中某个月的销量，而第二阶段则用此数据作为输入，将该商品某个月的销量与上一年同月销量相比，以求出运算结果（如图 7.8 所示）。

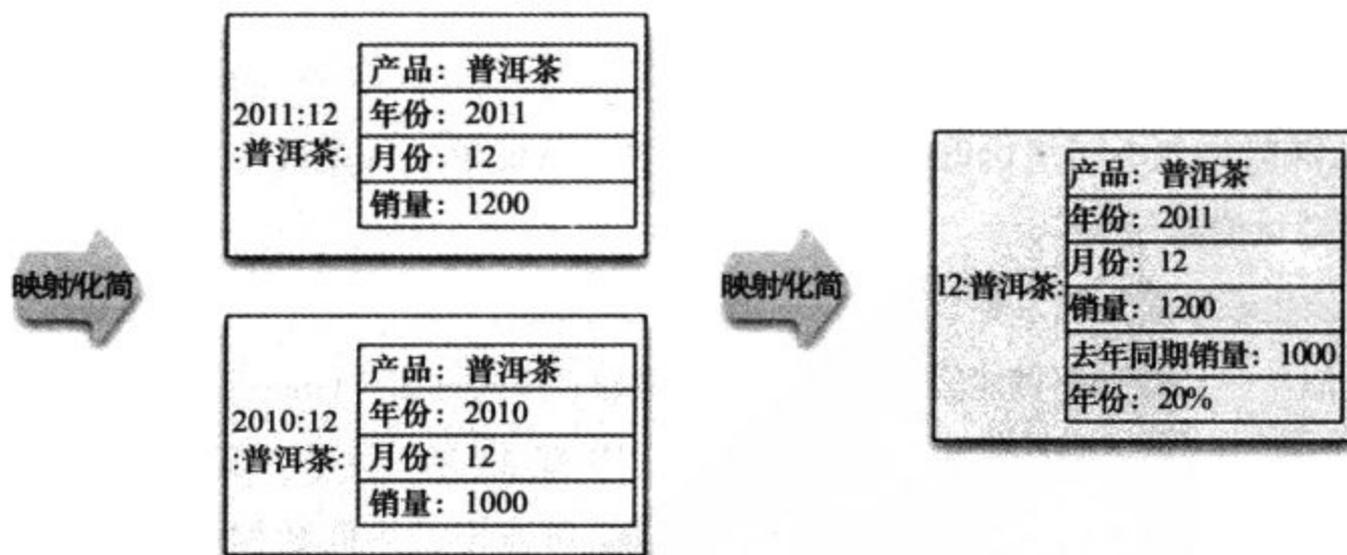


图 7.8 将计算过程分解为两个“映射 - 化简”步骤，后续三张图将详细解释其中细节

第一阶段（如图 7.9 所示）读出原始订单记录，将每件商品各个月的销量输出为一组键值对。

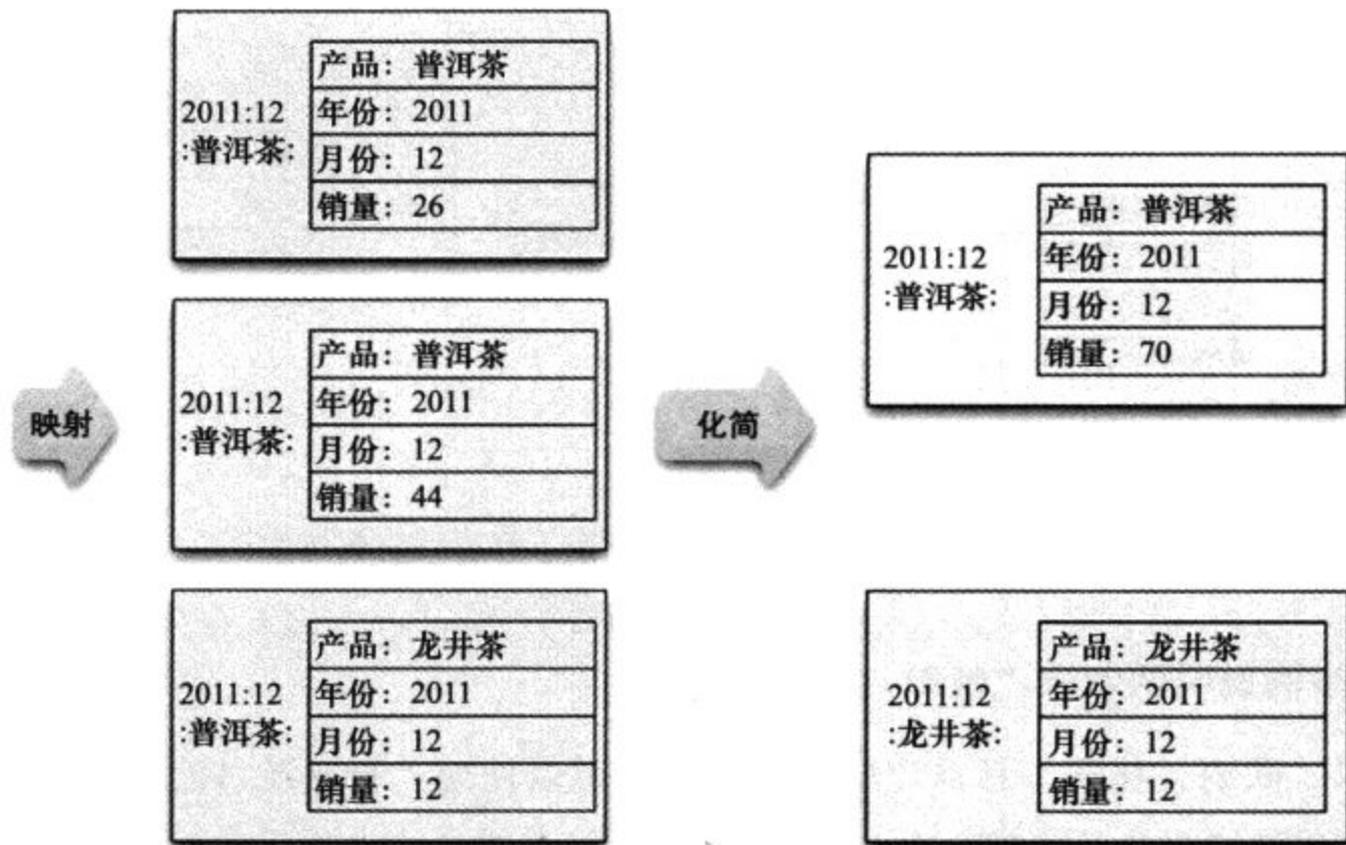


图 7.9 创建产品月销量记录

该阶段与早前所举“映射 - 化简”范例相似，唯一有变化的地方就是采用了“复合键”（composite key），以便基于多个字段值来化简记录。

第二阶段的“映射函数”（如图 7.10 所示）会按年度处理刚才输出的数据。它用 2011 年的记录生成当年某个月的销量，用 2010 年的记录生成去年同月销量。“映射函数”在计算输出结果时不使用早前年份（例如 2009 年）的记录。

此时的“化简”操作就是合并记录（如图 7.11 所示）：将两份记录中的数值汇总，这样就可以把两个不同年份的输出记录化简为一条记录了（此外，还根据化简之后的记录算出销量增幅）。

将报表生成过程分解为多个“映射 - 化简”步骤之后，编程就更简单了。与很多数据转换案例相似，一旦找到那种能轻易组合多个步骤的“转换框架”（transformation framework），那么就可以把这些小步骤拼起来了，这比把所有逻辑都堆砌在一个大步骤中更方便。

两阶段“映射 - 化简”的另一个好处是，它所输出的中间数据可以用来计算其他输出数据，也就是可以对其复用。复用是个很关键的优势，因为它既可以节省编码时间，也可以加快执行速度。这些中间记录（intermediate record）可以放在数据库里，

以构成“物化视图”（参见 3.4 节）。“映射 - 化简”操作的早期阶段所生成的数据尤其有用，因为通常情况下，很多数据访问操作都要用到这些数据，所以可以将它们一次算好，提供给下游服务使用，这样就能节省大量精力。然而，与所有“复用”行为一样，最好是有实际查询需要了再复用，凭空想出的复用很难派上用场。所以说，在构建待复用的数据时，要分析各种查询请求，将其共有的计算操作提取到物化视图中。

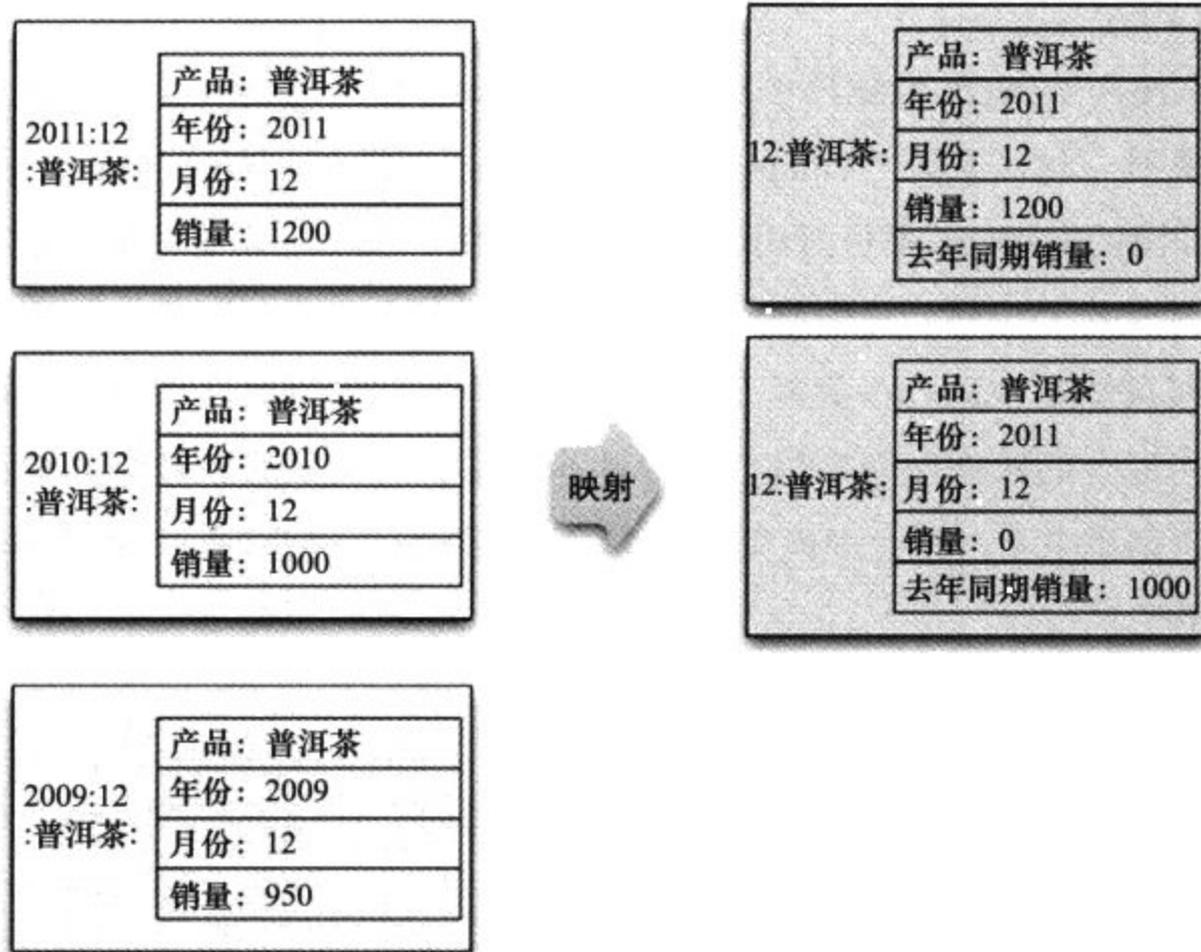


图 7.10 第二阶段的“映射函数”创建出计算同比增长所用的基础记录

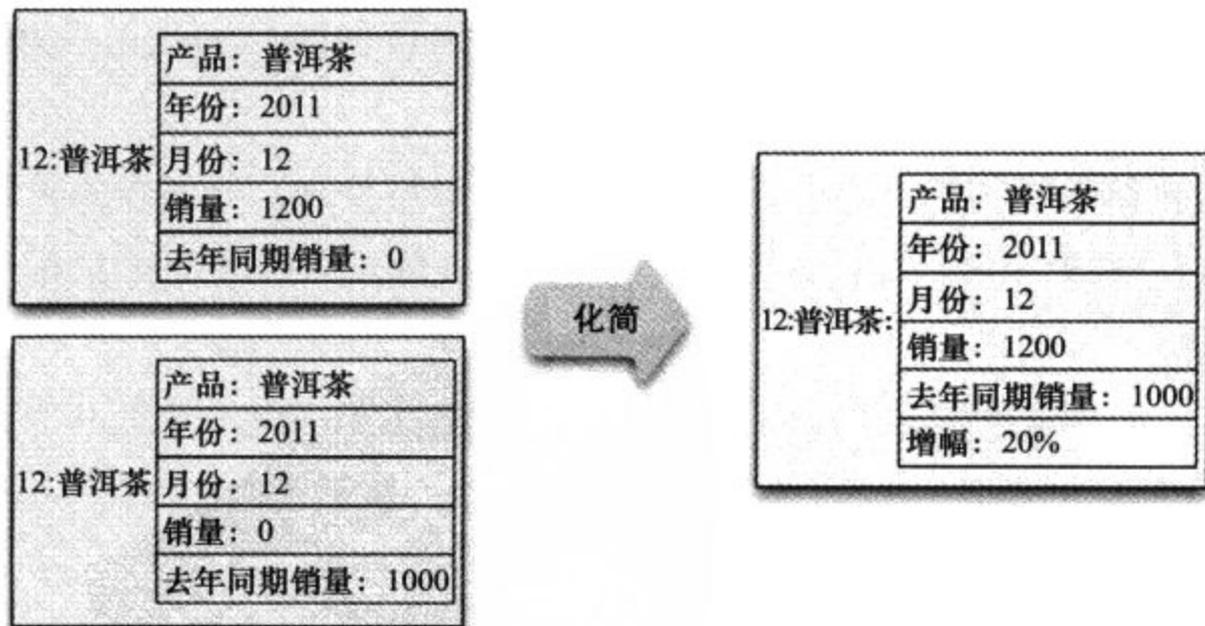


图 7.11 第二阶段的“化简”步骤将两条不完整的记录合并起来

“映射 - 化简”模式可用任意编程语言实现。然而，受其风格所限，最好能使用一门专为“映射 - 化简”计算而设计的语言。Hadoop 项目 [Hadoop] 的分支 Apache Pig [Pig] 就是这样一门专用语言，用它编写“映射 - 化简”程序很方便。利用 Hadoop 框架来编程，当然要比使用底层 Java 库简单多了。与之类似，若是想用“类 SQL 语法” (SQL-like syntax) 来编写“映射 - 化简”程序，那么可以使用另一个 Hadoop 项目的分支：hive [Hive]。

就算在不使用 NoSQL 数据库的环境中，也应该了解“映射 - 化简”模式。谷歌公司起初用“映射 - 化简系统” (map-reduce system) 来操作分布式文件系统中的文件，而这也正是开源的 Hadoop 项目所用的办法。要使用“映射 - 化简”模式，就得将数据计算操作组织成若干步骤，我们确实需要费些心思来适应这种限制，然而这样设计出来的运算过程自然非常适合在集群上运行。在处理海量数据时，需要使用面向集群的方式，而面向聚合的数据库非常适合此种计算风格。笔者认为在数年之内，很多公司所需处理的数据量之多，只有面向集群的方案才能应付，到那时，“映射 - 化简”模式就将大显身手了。

### 7.3.2 增量式“映射 - 化简”

刚才讨论的那些示例都使用完整的“映射 - 化简”计算流程，也就是从原始输入数据开始，直至算出最终输出结果。许多“映射 - 化简”计算，即便将其分布到集群中的多台机器上执行，也还是较为耗时，而且在计算过程中，新数据会不断涌入，为了保证输出的数据不过时，必须重新执行计算流程。每次都从头开始计算会很耗时，所以通常都会将“映射 - 化简”计算组织成易于“增量更新” (incremental update) 的形式，这样就能把计算量减至最低了。

“映射 - 化简”操作的“映射”阶段比较容易增量处理，只是在输入数据改变时必须重新执行“映射函数”而已。因为映射操作彼此隔离，所以对其应用“增量更新”技术非常容易。

“化简”这一步就复杂多了，因为它要把很多映射操作的输出数据汇聚起来，如果某个映射操作的输出数据改变了，那么必须再次化简。根据化简操作的“平行程度”<sup>⊖</sup>，

---

⊖ 原文为“how parallel the reduce step is”，这里的“平行程度”指操作之间的依赖性。联动关系少的操作，其“平行程度”高。也可以将之理解为并发执行能力。——译者注

我们可以减少重新化简时的计算量。假如把待化简的数据“分区”，那么其数据未改变的“分区”就无需再次化简了。与此相似，如果有“归并”步骤，那么在归并操作的源数据未改变时，就不需要重新归并了。

如果“化简函数”也可以充当“归并函数”，那么还可以免去更多计算。假如数据变更是可迭加的（additive），那么只需加入新记录，并对现有结果和新加入的数据执行化简操作即可，不需修改或删除现有记录。若数据变更是“破坏性的”（destructive），也就是包含更新及删除操作，那么还是可以将化简操作分解为多个步骤，只重新计算那些输入数据改变的步骤就好。实际上，此时可用“依赖网络”（Dependency Network）[Fowler DSL] 来组织计算流程。

上述很多因素都可以用“映射 - 化简框架”调控，所以需要明白你所使用的那个“映射 - 化简”框架是如何支持增量操作（incremental operation）的。

## 7.4 延伸阅读

如果打算使用“映射 - 化简”计算，那么首先应该查阅当前使用的数据库文档。每个数据库都各有其处理方式、术语表及微妙之处，使用者必须熟悉这些。除此之外，还需要获知一些更为通用的信息，以了解如何组织“映射 - 化简”操作才能尽力提高其维护性与效率。这个问题尚未有可供参考的专著，不过笔者觉得讲 Hadoop 的那些书对此很有帮助——我们容易忽视这一点。虽说 Hadoop 不是数据库，但它却频繁使用“映射 - 化简”操作，所以说，用 Hadoop 写成的高效“映射 - 化简”任务，或许也适用于其他环境（适用程度要看 Hadoop 与当前使用的系统之间存在多少细节差异）。

## 7.5 要点

- “映射 - 化简”是一种在集群上执行并发计算所用的模式。
- “映射”任务从聚合中读出数据，将之缩减为相关键值对。“映射”操作每次只能读取一条记录，所以可在存放记录的节点上并发执行。
- “映射任务”会生成许多具备同一关键字的值，而“化简任务”则将它们简化为单一的输出值。每个“化简函数”只操作与单个键相关的映射结果，所以多个

“化简函数”可以依据关键字执行并发化简。

- 输入数据与输出数据形式相同的多个“化简函数”可归并为“管道”，以提高并发执行能力，并减少所需传输的数据量。
- 若某个“化简操作”的输出是下一个“映射操作”的输入，那么就可以用“管道”组合“映射 - 化简”操作。
- 如果需要广泛使用“映射 - 化简”计算的结果，那么可将其存储为“物化视图”。
- 可用增量式“映射 - 化简”操作更新“物化视图”，这样只需计算视图中发生改变的那部分数据即可，不需要把全部数据都从头算一遍。

## 第二部分

# 实 现

- 第 8 章 键值数据库
- 第 9 章 文档数据库
- 第 10 章 列族数据库
- 第 11 章 图数据库
- 第 12 章 模式迁移
- 第 13 章 混合持久化
- 第 14 章 超越 NoSQL
- 第 15 章 选择合适的数据库

## 第 8 章

# 键值数据库

键值数据库 (key-value store) 是一张简单的哈希表 (hash table), 主要用在所有数据库访问均通过主键 (primary key) 来操作的情况下。可把此表想象成传统的“关系型数据库管理系统” (Relational Database Management System, 缩写为 RDBMS, 简称关系型数据库, 下同), 它有两列, 可叫做 ID 与 NAME。ID 列代表关键字, NAME 列存放值。在关系型数据库中, NAME 列仅能存放 String 型的数据。应用程序可提供 ID 及 VALUE 值, 并将这一键值对持久化。假如 ID 已存在, 就用新值覆盖当前值, 否则就新建一条数据。下表对比了 Oracle 与 Riak 的术语。

Oracle	Riak
数据库实例 (database instance)	Riak 集群 (Riak cluster)
表 (table)	存储区 (bucket)
行 (row)	键值对 (key-value)
rowid <sup>⊖</sup>	键 (key)

⊖ 该词中文称“伪列”, 但这与将“row”称为“行”的习惯不符, 故此处保留英文。——译者注

## 8.1 何谓“键值数据库”

从 API 的角度来看，键值数据库是最简单的 NoSQL 数据库。客户端可以根据键查询值，设置键所对应的值，或从数据库中删除键。“值”只是数据库存储的一块数据而已，它并不关心也无需知道其中的内容；应用程序负责理解所存数据的含义。由于键值数据库总是通过主键访问，所以它们一般性能较高，且易于扩展。

流行的键值数据库有：Riak[Riak]、Redis（通常称为“数据结构服务器”（Data Structure server））[Redis]、Memcached DB 及其变种 [Memcached]、Berkeley DB[Berkeley DB]、HamsterDB（尤其适合嵌入式开发）[HamsterDB]、Amazon DynamoDB[Amazon's Dynamo]（不开源）和 Project Voldemort[Project Voldemort]（Amazon DynamoDB 的开源实现）。

在 Redis 等键值数据库中，所存储的聚合不一定非要是领域对象（domain object），任何数据结构都可以。Redis 能够存储 list、set、hash 等数据结构，而且支持“获取某个范围内的数值”（range）、“求差集”（diff）、“求并集”（union）、“求交集”（intersection）等操作。这些功能使 Redis 数据库的用途变得比标准键值数据库更多。

键值数据库还有很多种，而且新品层出不穷。为了让讨论变得简单些，本书主要关注 Riak。它可以把关键字放在“存储区”（bucket）中。而“存储区”只是区隔关键字的一种手段，可以将其视为存放关键字所用的“平坦命名空间”（flat namespace）<sup>①</sup>。

如果要在 Riak 中存储用户会话数据、购物车信息及用户配置信息，那么只需要把上述所有对象都放在一个值中，并将其关联至存储区中的一个关键字即可。在此情况下，我们把所有数据都放在一个对象里，并将其存入单一的存储区中（如图 8.1 所示<sup>②</sup>）。

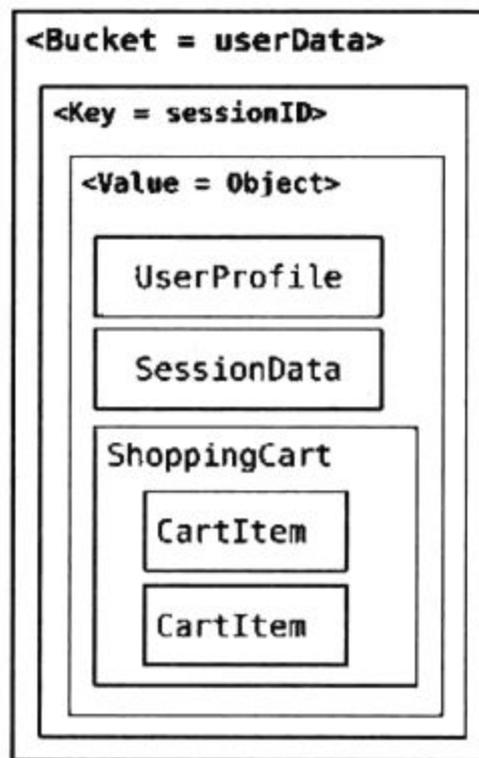
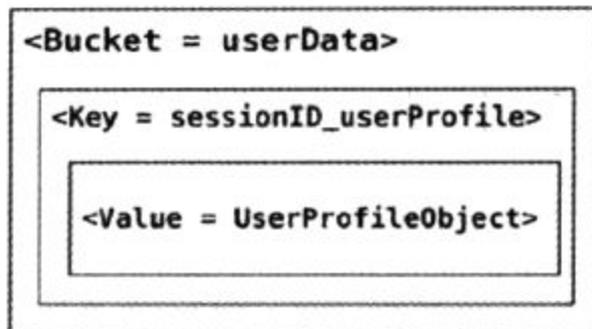


图 8.1 将所有数据放在一个“存储区”中

① 也就是没有层级结构的命名空间，与之相对的是“hierarchical namespace”，可参考：[http://www.uotechnology.edu.iq/sweit/Lectures/SaifGh-Advanced-IT/AdvanceIT4th\\_Ch1.pdf](http://www.uotechnology.edu.iq/sweit/Lectures/SaifGh-Advanced-IT/AdvanceIT4th_Ch1.pdf) 的 1.3 节与 1.4 节。——译者注

② 其中 UserProfile、SessionData、ShoppingCart、CartItem 分别表示用户配置信息、会话数据、购物车和购物车中的商品。——译者注

将各类对象（也就是聚合）全部存放在一个“存储区”中，其缺点是：“存储区”中可能要存放类型不同的多个聚合，这增加了关键字冲突的几率。还有一种办法是把对象名放在键名后面，例如 288790b8a421\_userProfile，这样就可用它查出所需的单个对象了（如图 8.2 所示）。



还可以创建“存储区”来存放特定数据。在 Riak 中，这些叫做“领域存储区”（domain bucket），客户端驱动程序可以对其执行“序列化”（serialization）与“反序列化”（deserialization）操作。

图 8.2 用另外一种方式命名键，以区隔“存储区”中的数据

```
Bucket bucket = client.fetchBucket(bucketName).execute();
DomainBucket<UserProfile> profileBucket =
DomainBucket.builder(bucket, UserProfile.class).build();
```

将跨越多个“存储区”的数据分割成对象，将之存放在“领域存储区”或不同的“存储区”中，这样一来，无需改变关键字的命名方式，即可读出所需对象。

像 Redis 这样的键值数据库也能存放集合、哈希表、字符串等随机数据结构。此特性可用于存放数值列表，例如 states、addressTypes，或是包含用户访问记录的数组。

## 8.2 键值数据库特性

不论使用哪种 NoSQL 数据库，都必须理解其特性与熟知的标准关系型数据库之间有何异同。这么说的主要原因在于：我们需要了解当前数据库缺失哪些特性，以及应用程序需要如何改变其架构，才能更好地利用键值数据库的特性。讨论每一种 NoSQL 数据库的特性时，都需要了解其“一致性”、“事务”、查询特性、数据结构及可扩展性。

### 8.2.1 一致性

只有针对单个键的操作才具备“一致性”，因为这种操作只可能是“获取”、“设置”或“删除”。“乐观写入”（optimistic write）功能其实也可以做出来，然而由于数据库无法侦测数值改动，所以其实现成本太高。

Riak 这种分布式键值数据库，用“最终一致性模型”（参见 5.2 节）实现“一致性”。因为数值可能已经复制到其他节点，所以 Riak 有两种解决“更新冲突”的办法：一种是采纳新写入的数据而拒绝旧数据，另一种是将两者（或存在冲突的所有数据）

返回给客户端，令其解决冲突。

在 Riak 中，可以在创建“存储区”时设置上述选项。“存储区”只是用以减少键名冲突的一种命名空间，比方说，我们可以把与客户有关的全部键都放在 customer 这个“存储区”中。创建存储区时可以提供一些默认值，以确保其“一致性”。例如可以规定：执行完写入操作后，只有当存放此数据的全部节点一致将其更新，我们才认定该操作生效。

```
Bucket bucket = connection
    .createBucket(bucketName)
    .withRetrier(attempts(3))
    .allowSiblings(siblingsAllowed)
    .nVal(numberOfReplicasOfTheData)
    .w(numberOfNodesToRespondToWrite)
    .r(numberOfNodesToRespondToRead)
    .execute();
```

假如我们想让每个节点中的数据都一致，那么可以把 w 方法的 numberOfNodesToRespondToWrite 参数值设置成与 nVal 方法的参数相同。这么做无疑会降低集群的写入效率。若想提高写入冲突或读取冲突的解决速度，可在创建“存储区”时改变 allowSiblings 标志：如果将其设为 false，那么数据库就接纳最新的写入操作，而不再创立“旁系记录”（sibling）<sup>①</sup>了。

### 8.2.2 事务

不同类型的键值数据库产品，其“事务”规范也不同。一般说来，无法保证写入操作的“一致性”。各种数据库实现“事务”的方式各异。Riak 采用“仲裁”这一概念（参见 5.5 节），在调用写入数据的 API 时，它使用 W 值与复制因子来实现“仲裁”。

假设某个 Riak 集群的复制因子是 5，而 W 值为 3。在写入数据时，必须有至少 3 个节点汇报其写入操作已顺利完成，数据库才会认为此操作执行完毕。这样的话，Riak 就具备了“写入操作容错性”（write tolerate）。在本例中，由于 N 等于 5 而 W 是 3，所以集群在两个节点（N-W=2）故障时仍可执行写入操作，不过，此时我们无法从那些发生故障的节点中读取某些数据。

### 8.2.3 查询功能

所有键值数据库都可以按关键字查询。它们的查询功能基本上仅限于此。如果想

<sup>①</sup> sibling record 的详情可参考：<http://docs.basho.com/java/latest/cookbooks/storingdata/>。——译者注

根据“值列”(value column)的某些属性来查询,那么无法用数据库完成此操作:应用程序需要自己读出值,并判断其属性是否符合查询条件。

按关键字查询还会带来一种值得注意的副作用:如果不知道关键字该怎么办?调试程序时所需的“即时查询”(ad-hoc querying)操作尤其如此。大部分数据库都不提供全部主键列表,即便提供了,获取关键字列表并查询其值的操作也很烦琐。某些键值数据库支持数值搜寻,以解决此问题。例如 Riak Search 技术<sup>⊖</sup>支持以“Lucene 全文检索系统”的方式查询数据。

使用键值数据库时,大部分精力都要花在设计键名上:可以用某种算法生成键吗?可以拿用户信息(例如 ID、电子邮件地址等)当键吗?或者可以用时间戳等数据库以外的值来生成吗?

由于具备这些查询特征,所以键值数据库非常适合保存会话(用会话 ID 作为键)、购物车数据、用户配置等信息。可以用 expiry\_secs 属性指定关键字的过期时间,这对会话或购物车对象特别有用。

```
Bucket bucket = getBucket(bucketName);
IRiakObject riakObject = bucket.store(key, value).execute();
```

使用名为 store 的 API,可将对象按指定的关键字存储到 Raik 存储区中。同理,通过名叫 fetch 的 API,可以获取某关键字所对应的值。

```
Bucket bucket = getBucket(bucketName);
IRiakObject riakObject = bucket.fetch(key).execute();
byte[] bytes = riakObject.getValue();
String value = new String(bytes);
```

Riak 提供了基于 HTTP 协议的接口,所有操作都可以用 Web 浏览器或 curl 命令行执行。假设要将下列数据存入 Riak:

```
{
  "lastVisit":1324669989288,
  "user":{
    "customerId":"91cfd5bcb7c",
    "name":"buyer",
    "countryCode":"US",
    "tzOffset":0
  }
}
```

使用 curl 命令投递 (POST) 数据,将之存于名叫 session 的“存储区”中,并与

⊖ 详情参见: <http://docs.basho.com/riak/latest/cookbooks/Riak-Search-Querying/>。——译者注

名为 a7e618d9db25 的键（网址中必须包含键名）相关联：

```
curl -v -X POST -d '{
  "lastVisit":1324669989288,
  "user":{"customerId":"91cfd5bcb7c",
  "name":"buyer",
  "countryCode":"US",
  "tzOffset":0}
}'
-H "Content-Type: application/json"
http://localhost:8098/buckets/session/keys/a7e618d9db25
```

键名 a7e618d9db25 所对应的数据可用如下 curl 命令获取：

```
curl -i http://localhost:8098/buckets/session/keys/a7e618d9db25
```

## 8.2.4 数据结构

键值数据库并不关心键值对里的值。它可以是二进制块、文本、JSON、XML 等。使用 Riak 时，可在 POST 请求中用 Content-Type 指定数据类型。

## 8.2.5 可扩展性

很多键值数据库都可用“分片”技术（参见 4.2 节）扩展。采用此技术后，键的名字就决定了负责存储该键的节点。假设按照键名的首字母“分片”。如果键名是 f4b19d79587d，那么由于其首字母为 f，所以存放它的节点就与存放 ad9c7a396542 这个键的节点不同。当集群中的节点数变多时，这种“分片”设定可提高效率。

“分片”也会引发某些问题。假如存放首字母为 f 的键所用的那个节点坏了，那么其上的数据将无法访问，而且也不能再写入其他键名首字母为 f 的新数据了。

像 Riak 这样的数据库，可以控制“CAP 定理”（参见 5.3.1 节）中的参数：N（存放键值对的副本节点数）、R（顺利完成读取操作所需的最小节点数）和 W（顺利完成写入操作所需的最小节点数）。

假设 Riak 集群有 5 个节点。将 N 设为 3，意思就是所有数据都至少要复制到 3 个节点中，将 R 设为 2，意思是 GET 请求要有两个节点应答，才能成功。将 W 设为 2，意思是 PUT 请求必须写入两个节点，才算执行完毕。

可以利用这些设置来微调读取及写入操作所能容忍的故障节点数。我们应该按照自己的需要来改变这些值，以提升数据库的“可读能力”（read availability）及“可写

能力”(write availability)。通常应该根据“一致性”需求来确定 W 值。创建“存储区”时可设定上述各参数的默认值。

## 8.3 适用案例

现在讲几个适合使用键值数据库的情况。

### 8.3.1 存放会话信息

通常来说，每一次网络会话都是唯一的，所以分配给它们的 sessionid 值也各不相同。如果应用程序原来要把 sessionid 存在磁盘上或关系型数据库中，那么将其迁移到键值数据库之后，会获益良多，因为全部会话内容都可以用一条 PUT 请求来存放，而且只需一条 GET 请求就能取得。由于会话中的所有信息都放在一个对象中，所以这种“单请求操作”(single-request operation)很迅速。许多网络应用程序都使用像 Memcached 这样的解决方案。如果“可用性”较为重要，可使用 Riak。

### 8.3.2 用户配置信息

几乎每位用户都有 userId、username 或其他独特的属性，而且其配置信息也各自独立，诸如语言、颜色、时区、访问过的产品等。这些内容可全部放在一个对象里，以便只用一次 GET 操作即获取某位用户的全部配置信息。同理，产品信息也可如此存放。

### 8.3.3 购物车数据

电子商务网站的用户都与其购物车相绑定。由于购物车的内容要在不同时间、不同浏览器、不同电脑、不同会话中保持一致，所以可把购物信息放在 value 属性中，并将其绑定到 userid 这个键名上。此类应用程序最宜使用 Riak 集群了。

## 8.4 不适用场合

键值数据库在某些场合下并不是最佳方案。

### 8.4.1 数据间关系

如果要在不同数据集之间建立关系，或是将不同的关键字集合联系起来，那么即

便某些键值数据库提供了“链接遍历”等功能，它们也不是最佳选择了。

#### 8.4.2 含有多项操作的事务

如果在保存多个键值对时，其中有一个关键字出错，而你又需要复原或回滚其余操作，那么键值数据库就不是最好的解决方案。

#### 8.4.3 查询数据

如果要根据键值对的某部分值来搜寻关键字，那么键值数据库就不是很理想了。我们无法直接检视键值数据库中的值，除非使用某些类似 Riak Search 的产品或是像 Lucene[Lucene]、Solr[Solr] 这样的“检索引擎”（indexing engine）。

#### 8.4.4 操作关键字集合

由于键值数据库一次只能操作一个键，所以它无法同时操作多个关键字。假如需要操作多个关键字，那么最好在客户端处理此问题。

## 第 9 章

# 文档数据库

“文档”（document）是文档数据库中的主要概念。此类数据库可存放并获取文档，其格式可以是 XML、JSON、BSON<sup>⊖</sup> 等。这些文档具备自述性（self-describing），呈现分层的树状数据结构（hierarchical tree data structure），可以包含映射表、集合和纯量值。数据库中的文档彼此相似，但不必完全相同。文档数据库所存放的文档，就相当于键值数据库所存放的“值”。文档数据库可视为其值可查的键值数据库。下表对比了 Oracle 与 MongoDB 的术语。

Oracle	MongoDB
数据库实例（database instance）	MongoDB 实例（MongoDB instance）
模式（schema）	数据库（database）
表（table）	集合（collection）
行（row）	文档（document）
rowid	_id
join	DBRef

与 Oracle 中的 ROWID 相似，MongoDB 数据库的所有文档都包含名叫 `_id` 的特殊字段。在 MongoDB 中，`_id` 字段可由用户设置，只要其值唯一即可。

---

⊖ “Binary JSON” 的缩写，是 MongoDB 所使用的二进制数据文件，可存放简单的数据结构及映射表。详情参见：<https://en.wikipedia.org/wiki/BSON>。——译者注

## 9.1 何谓文档数据库

```
{ "firstname": "Martin",
  "likes": [ "Biking",
            "Photography" ],
  "lastcity": "Boston",
  "lastvisited":
}
```

上述文档相当于传统关系型数据库中的一行记录。下面再看另外一份文档：

```
{
  "firstname": "Pramod",
  "citiesvisited": [ "Chicago", "London", "Pune", "Bangalore" ],
  "addresses": [
    { "state": "AK",
      "city": "DILLINGHAM",
      "type": "R"
    },
    { "state": "MH",
      "city": "PUNE",
      "type": "R" }
  ],
  "lastcity": "Chicago"
}
```

这两份文档看上去相似，然而其中各属性的名字不同。文档数据库中可以这么做。各文档的“数据模式”（the schema of the data）也许不同，但是它们仍然能放在同一“集合”内，而不是像关系型数据库那样，表格中每行数据的模式都要相同。citiesvisited 这个列表可视为数组，而 addresses 列表则可看成嵌入主文档的一系列小文档。将“子文档”（child document）以“子对象”（subobject）的形式嵌入主文档，可方便访问并提升效率。

大家在这两份文档中会发现，它们有某些相似的属性，比如 `firstname` 或 `lastcity`。同时，第二份文档中的 `addresses` 等属性第一份里没有，而第一份文档的 `likes` 属性第二份里也没有。

这种数据表示方法与关系型数据库不同，后者需要定义表中的每一列，而且若某条记录中的某列没有数据，则要将其留空（empty）或设为 `null`。文档数据库的文档则没有空属性，若其中不存在某属性，我们就假定该属性值未设定或与此文档无关。向文档中新增属性时，既无需预先定义，也不用修改已有文档内容。

流行的文档数据库有：MongoDB[MongoDB]、CouchDB[CouchDB]、Terrastore[Terrastore]、OrientDB[OrientDB] 和 RavenDB[RavenDB]。当然了，饱受责难的 Lotus Notes[Notes Storage Facility] 也使用文档数据库。

## 9.2 特性

专属的文档数据库有很多，而本书以 MongoDB 为例，讲解其各项特性。请记住，每个文档数据库都具备同类数据库所没有的某些特性。

现在要花些时间理解 MongoDB 的工作原理。每个“MongoDB 实例”都包含许多“数据库”<sup>⊖</sup>，而每个“数据库”又有许多“集合”（collection）。把这套机制与关系型数据库比较一下：“关系型数据库实例”与“MongoDB 实例”相仿，其“模式”和 MongoDB 的“数据库”类似，而关系型数据库的“表”相当于 MongoDB 的“集合”。存储文档时，必须像 `database.collection.insert (document)` 这样，选择其所属的“数据库”与“集合”。前述代码通常写为 `db.coll.insert (document)`。

### 9.2.1 一致性

为了在 MongoDB 数据库中确保“一致性”，可以配置“副本集”（replica set），也可以规定写入操作必须等待所写数据复制到全部或是给定数量的从节点之后，才能返回。每次写入数据时，都可以指定写入操作返回之前，必须将所写数据传播到多少个服务器节点上。

可以用类似 `db.runCommand ({ getlasterror: 1, w: "majority" })` 这样的命令指定数据库的“一致性”强度。例如，在只有一台服务器时如果指定 `w` 为 `majority`<sup>⊖</sup>，那么写入操作立刻就会返回，因为总共只有一个节点。假设“副本集”中有三个节点，而 `w` 设为 `majority`，则写入操作必须在至少两个节点上执行完毕，才会视为成功。提升 `w` 值可以增强“一致性”，但是会降低写入效率，因为写入操作必须在更多的节点上完成才行。也可以增加“副本集”的读取效率：设置 `slaveOk` 选项之后，就可以于从节点中读取数据了。该参数既可设置到整个“连接”、“数据库”、“集合”之上，也可针对每项操作独立设置。

```
Mongo mongo = new Mongo("localhost:27017");
mongo.slaveOk();
```

下列代码对单个操作设置 `slaveOk` 选项，这样就可以控制哪些操作只需要于从节点中读取数据就好。

---

⊖ 此语境下的“数据库”（database）是 MongoDB 的术语，与广义的数据库不完全相同，故译文将其打上引号。——译者注

⊖ 中文意为“大多数”。——译者注

```

DBCollection collection = getOrderCollection();
BasicDBObject query = new BasicDBObject();
query.put("name", "Martin");
DBCursor cursor = collection.find(query).slaveOk();

```

读取操作可设置各种选项，与此类似，如有需要，也能用很多设置来增强写入操作的“一致性”。在默认情况下，只要数据库收到了写入数据，就认定该操作成功。我们可以对此修改，让写入操作必须等待所写数据同步至磁盘或传播到至少两个从节点后，才能返回。这叫做 WriteConcern。将 WriteConcern 设为 REPLICAS\_SAFE，即可确保数据能写入主节点及某些从节点中。下列代码所设置的 WriteConcern 会影响集合的每一次写入操作。

```

DBCollection shopping = database.getCollection("shopping");
shopping.setWriteConcern(REPLICAS_SAFE);

```

也可以针对每次操作来设定 WriteConcern。在调用命令时指定该选项即可：

```

WriteResult result = shopping.insert(order, REPLICAS_SAFE);

```

开发者要仔细思量应用程序的需要及业务需求，据此权衡，以决定读取操作应该使用何种 slaveOk 设置，并通过 WriteConcern 设置写入操作的安全级别。

## 9.2.2 事务

从传统的关系型数据库角度讲，“事务”一词意味着我们可以先用 insert、update 或 delete 等命令操作不同的表，然后用 commit 提交修改或以 rollback 命令回滚。NoSQL 数据库通常没有这些机制：其写入操作要么成功，要么失败。“单文档级别”（single-document level）的“事务”叫做“原子事务”（atomic transaction）。虽说 RavenDB 这样的产品确实支持跨越多个操作的“事务”，但此类数据库的“事务”一般无法执行多个操作。

在默认情况下，所有写入操作都将顺利执行。使用 WriteConcern 参数可对此微调。以 WriteConcern.REPLICAS\_SAFE 为参数写入 order，即可确保该操作至少要写入两个节点才算成功。可以用不同级别的 WriteConcern 参数来确保各种安全级别的写入操作。比方说，在写日志条目（log entry）时，就可使用最低的安全级别，也就是 WriteConcern.NONE。

```

final Mongo mongo = new Mongo(mongoURI);
mongo.setWriteConcern(REPLICAS_SAFE);
DBCollection shopping = mongo.getDB(orderDatabase)
    .getCollection(shoppingCollection);

```

```

try {
    WriteResult result = shopping.insert(order, REPLICAS_SAFE);
    //Writes made it to primary and at least one secondary
} catch (MongoException writeException) {
    //Writes did not make it to minimum of two nodes including primary
    dealWithWriteFailure(order, writeException);
}

```

### 9.2.3 可用性

“CAP 定理”（参见 5.3.1 节）断言：“一致性”（Consistency）、“可用性”（Availability）和“分区耐受性”（Partition Tolerance）三者只可有其二。文档数据库试图用主从式数据复制技术来增强“可用性”。多个节点都保有同一份数据，即便主节点故障，客户端也依然能获取数据。应用程序代码一般不需检测主节点是否可用。MongoDB 通过“副本集”实现“复制”，以提供较高的“可用性”。

副本集中至少有两个节点参与“异步主从式复制”（asynchronous master-slave replication）。副本集在其内部选举“主”（master）节点，或曰“主要”（primary）节点。假定所有节点投票权相同，其中某些节点可能会因为距离其他服务器较近，或具有更多运行内存（RAM）等因素而获得更多选票。用户也可以为节点指定一个值在 0 ~ 1000 之间的优先级（priority）来影响选举过程。

所有请求都由主节点处理，而其数据会复制到从节点。若主节点故障，则“副本集”中剩下的节点就会在其自身范围内选出新的主节点，所有后续请求就交由新的主节点处理，从节点也开始从新的主节点处获取数据。当原来的主节点从故障中恢复时，它会作为从节点重新加入，并获取全部最新数据，以求与其他节点一致。

图 9.1 演示了一个“副本集”配置。其中，主数据中心有两个运行 MongoDB 数据库的节点，分别叫做 mongo A 与 mongo B，而辅助数据中心里的节点是 mongo C。假如想把主数据中心里的节点选为主节点，那么可以给它们分配高于其他节点的优先级。在向“副本集”中加入新节点时，无需将现有节点离线。

应用程序的写入或读取操作都针对主节点（primary node，或称 master node）。建立连接后，应用程序只需要同“副本集”中的一个节点相连即可（是不是主节点无所谓），数据库会自动找到其余节点。若主节点故障，则数据库驱动会同“副本集”中新选出的主节点联系。应用程序不用处理通信错误，也无需干预主节点的选拔准则。“副本集”技术打造了高度可用的文档数据库。

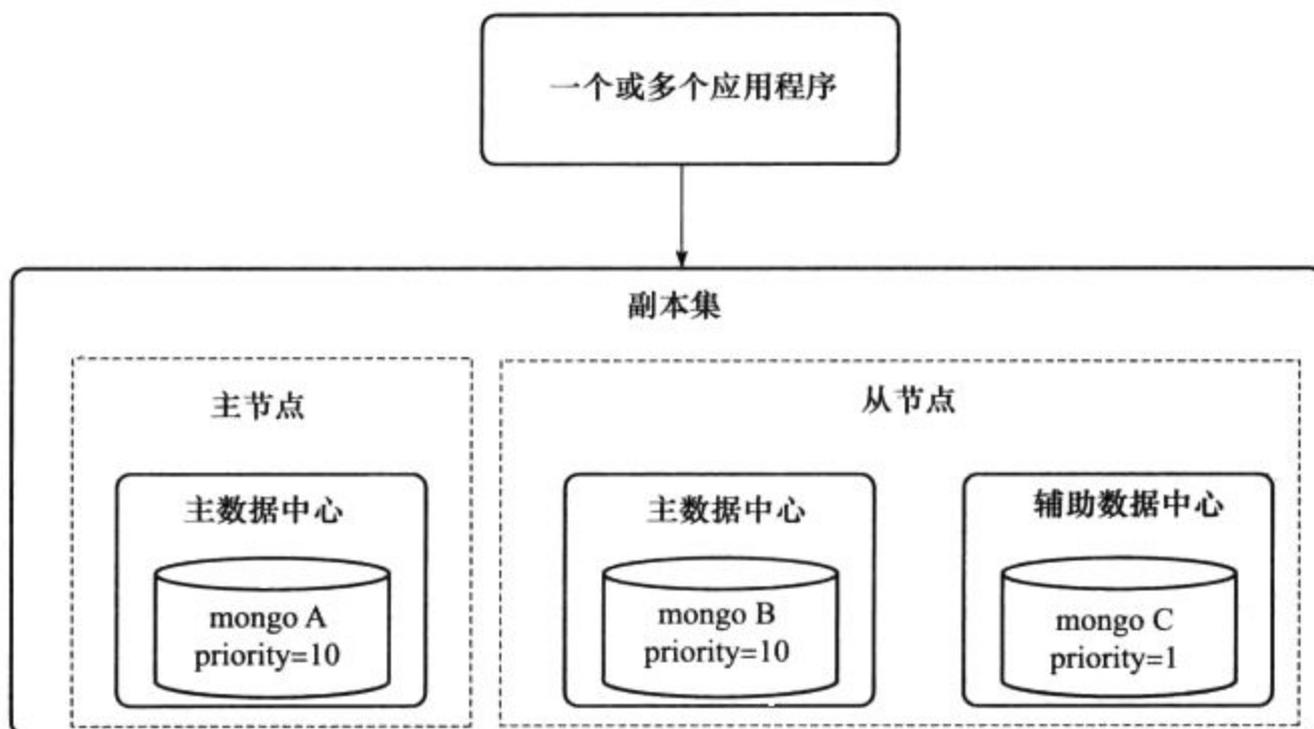


图 9.1 配置“副本集”时为同一数据中心里的节点赋予较高优先级

“副本集”通常用于处理“数据冗余”（data redundancy）、“自动故障切换”（automated failover）、“读取能力扩展”（read scaling）、“无需停机的服务器维护”（server maintenance without downtime）和“灾难恢复”（disaster recovery）等事项。CouchDB、RavenDB 和 Terrastore 等其他产品也有类似的“可用性”提升机制。

#### 9.2.4 查询功能

各种文档数据库提供了不同的查询功能。CouchDB 可通过视图查询：可用“物化视图”（materialized view，参见 3.4 节）或“动态视图”（dynamic view，相当于关系型数据库中的视图，不在乎是否物化）实现复杂的文档查询。在 CouchDB 中，假如需要把某产品的评价数（number of review）与平均评级（average rating）聚合起来，那么可以增加一个通过“映射 - 化简”（参见 7.1 节）实现出来的视图，让该视图返回评价数与平均评级。

如果查询请求比较多，那就不要在每次处理查询请求时重新计算评价数与平均评级了，而应该新增一个物化视图，预先算好这些值，并将结果存至数据库。若查询时发现物化视图中的数据与上次更新时不同，则更新此视图。

文档数据库有个好处：它可以查询文档中的数据，而不用像键值数据库那样，必须根据关键字获取整个文档，然后再检视其内容。这个特性使得此类数据库的查询功能更接近关系型数据库的查询模型了。

MongoDB 支持一种 JSON 格式的查询语言，它的功能包括：用 `MYMquery` 实现 `where` 子句，用 `MYMorderby` 对数据排序，用 `MYMexplain` 列出查询操作的执行计划等。还有很多诸如此类的用法可以合起来创建一条 MongoDB 查询命令。

下面讲讲如何在 MongoDB 中查询。假设需要返回某个订单集合内的全部文档（订单表的所有行），如果用 SQL 来做，那就是：

```
SELECT * FROM order
```

与之等效的 Mongo shell 命令是：

```
db.order.find()
```

若想选出 `customerId` 为 `883c2c5b4e5b` 的用户所下订单，可使用下列命令：

```
SELECT * FROM order WHERE customerId = "883c2c5b4e5b"
```

在 Mongo 中，可以用下列等效命令查出 `customerId` 为 `883c2c5b4e5b` 的用户所下的全部订单：

```
db.order.find({"customerId":"883c2c5b4e5b"})
```

同理，如果要用 SQL 找出此用户所下订单的 `orderId` 与 `orderDate`，那就是：

```
SELECT orderId,orderDate FROM order WHERE customerId = "883c2c5b4e5b"
```

与之等效的 Mongo 命令是：

```
db.order.find({customerId:"883c2c5b4e5b"},{orderId:1,orderDate:1})
```

数量、总价等属性都可以用类似办法查出。由于文档是“聚合对象”（`aggregated object`），所以用带子对象的字段查询待匹配的文档非常方便。假设要搜寻其商品项名称与 `Refactoring` 相似的全部订单，如果用 SQL 实现此需求，那就是：

```
SELECT * FROM customerOrder, orderItem, product
WHERE
customerOrder.orderId = orderItem.customerOrderId
AND orderItem.productId = product.productId
AND product.name LIKE '%Refactoring%'
```

与之等效的 Mongo 命令是：

```
db.orders.find({"items.product.name":/Refactoring/})
```

上述查询用 MongoDB 实现起来较为简单。由于那些对象都嵌在一份文档里，所以可基于“内嵌子文档”（`embed child document`）来查询。

### 9.2.5 可扩展性

这里“扩展”一词是指在不将数据库迁移到更大电脑的前提下，向其中新增节点或修改其内容。此处不谈如何让应用程序处理更多负载，我们关心的是数据库本身有哪些功能可提升负载处理能力。

增加更多的“读取从节点”（read slave），将全部读取操作都导引至从节点上，这样就可以扩展数据库应对频繁读取的能力了。假设某个应用程序的读取操作很频繁，而现有集群中的“副本集”里有3个节点，那么当读取负载增加时，可向“副本集”中加入更多从节点，并在执行读取操作时设定 `slaveOk` 标志，以提升集群的读取能力（如图 9.2 所示）。这就是读取操作的横向扩展。

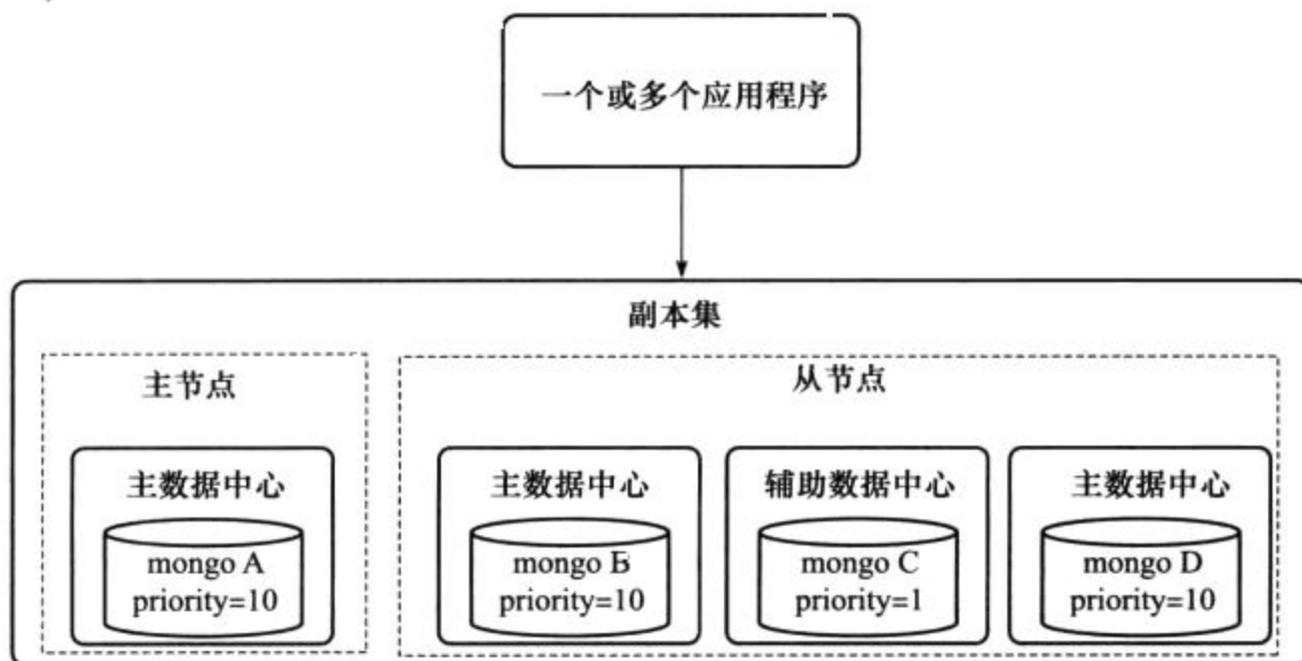


图 9.2 向集群中已有的“副本集”里新增 mongo D 节点

启动新节点 mongo D 之后，需要将其加入“副本集”。

```
rs.add("mongod:27017");
```

新节点加入后，会和已有节点同步，它以“辅助节点”（secondary node）的身份加入“副本集”，并开始处理读取请求。这样配置有个好处：不用重启其余节点，而且应用程序也无需暂停。

如果想扩展写入能力，可以把数据“分片”（参见 4.2 节）。“分片”与关系型数据库的“分区”类似，后者是根据某列的值，例如状态或年份，将数据分割开。关系型数据库的“分区”通常位于同一节点，所以客户端应用程序只查询“基表”（base table）就好，不需查询某个特定分区，关系型数据库会根据查询内容搜索适当的分区并返回数据。

“分片”操作也根据特定字段来划分数据，然而那些数据要移动到不同的 Mongo 节

点中。为了让各“分片”的负载保持均衡，需要在节点之间动态转移数据。向集群中新增更多节点，并提高可写入的节点数，就能横向扩展其写入能力。

```
db.runCommand( { shardcollection : "ecommerce.customer",
                key : {firstname : 1} } )
```

按照客户名字（first name）来分隔，可确保将数据平衡地散布在各个“分片”上，以获得较好的写入效率。此外，还可以把每个“分片”都做成“副本集”，以提高其读取效率（如图 9.3 所示）。如果向已有的“分片集群”（sharded cluster）中再加一个新“分片”，就可以把原来分布在 3 个“分片”中的数据打散到 4 个“分片”中。在转移数据与底层设施重构的全过程中，虽说集群为了重新平衡“分片”负载而传输大量数据时性能也许会下降，但是应用程序却无需停止工作。

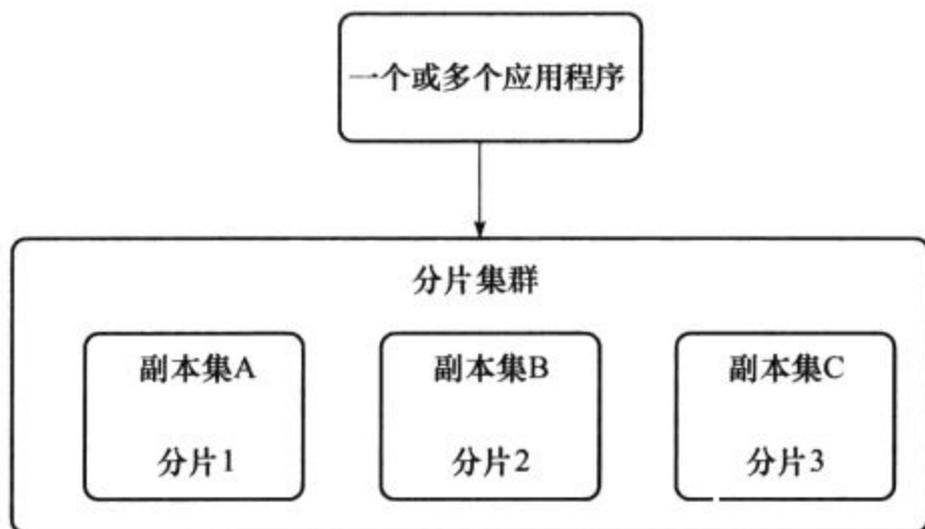


图 9.3 配置 MongoDB “分片”：将每个“分片”都做成“副本集”

“分片”的关键字很重要。如果想把 MongoDB 数据库“分片”放在距离用户近的地方，那么以用户位置来分片也许效果不错。按客户位置分片时，美国东海岸的全部用户数据都会放在居于东海岸的“分片”中，而所有西海岸的用户数据则将放在位于西海岸的“分片”中。

## 9.3 适用案例

### 9.3.1 事件记录

应用程序对事件记录各有需求。在企业级解决方案中，许多不同的应用程序都需要记录事件。文档数据库可以把所有这些不同类型的事件都存起来，并作为事件存储的“中心数据库”（central data store）使用。如果事件捕获的数据类型一直在变，那么就更应该用文档数据库了。可以按照触发事件的应用程序名“分片”，也可以按照

order\_processed 或 customer\_logged<sup>⊖</sup> 等事件类型“分片”。

### 9.3.2 内容管理系统及博客平台

由于文档数据库没有“预设模式”(predefined schema),而且通常支持JSON文档,所以它们很适合用在“内容管理系统”(content management system)及网站发布程序上,也可以用来管理用户评论、用户注册、用户配置和面向Web文档(web-facing document)。

### 9.3.3 网站分析与实时分析

文档数据库可存储实时分析数据。由于可以只更新部分文档内容,所以用它来存储“页面浏览量”(page view)或“独立访客数”(unique visitor)会非常方便,而且无需改变模式即可新增度量标准。

### 9.3.4 电子商务应用程序

电子商务类应用程序通常需要较为灵活的模式,以存储产品和订单。同时,它们也需要在不做高成本数据库重构及数据迁移(参见12.3节)的前提下进化其数据模型。

## 9.4 不适用场合

某些场合文档数据库并非最佳方案。

### 9.4.1 包含多项操作的复杂事务

文档数据库也许不适合执行“跨文档的原子操作”(atomic cross-document operation),然而像RavenDB等文档数据库其实也支持此类操作。

### 9.4.2 查询持续变化的聚合结构

灵活的模式意味着数据库对模式不施加任何限制。数据以“应用程序实体”(application entity)的形式存储。如果要即时查询这些持续改变的实体,那么所用的查询命令也得不停变化(用关系型数据库的术语讲,就是:用JOIN语句将数据表按查询标准连接起来时,待连接的表一直在变)。由于数据保存在聚合中,所以假如聚合的设计持续变动,那么就需要以“最低级别的粒度”(lowest level of granularity)来保存聚合了,这实际上就等于要统一数据格式了。在这种情况下,文档数据库也许不合适。

<sup>⊖</sup> 这两个事件类型的中文含义分别为“订单已处理”、“客户已记录”。——译者注

# 第 10 章

## 列族数据库

Cassandra[Cassandra]、HBase[HBase]、Hypertable[Hypertable]、Amazon SimpleDB[Amazon SimpleDB] 等列族数据库，可以存储关键字及其映射值，并且可以把值分成多个列族，让每个列族代表一张数据映射表（map of data）。

关系型数据库	Cassandra
数据库实例（database instance）	集群（cluster）
数据库（database）	键空间（keyspace）
表（table）	列族（column family）
行（row）	行（row）
列（column，每行所对应的各列均相同）	列（column，不同的行所对应的列可以有差别）

## 10.1 何谓列族数据库

列族数据库有很多种。本章讲的是 Cassandra，不过在谈及特定场景下用到的某些功能时，也会提到其他列族数据库。

列族数据库将数据存储于列族中，而列族里的行则把许多列数据与本行的“行键”（row key）关联起来（如图 10.1 所示）。列族用来把通常需要一并访问的相关数据分成组。可能要同时访问多个 Customer（客户）的 Profile（个人配置）信息，然而很少需要同时访问他们的 Orders（订单）。

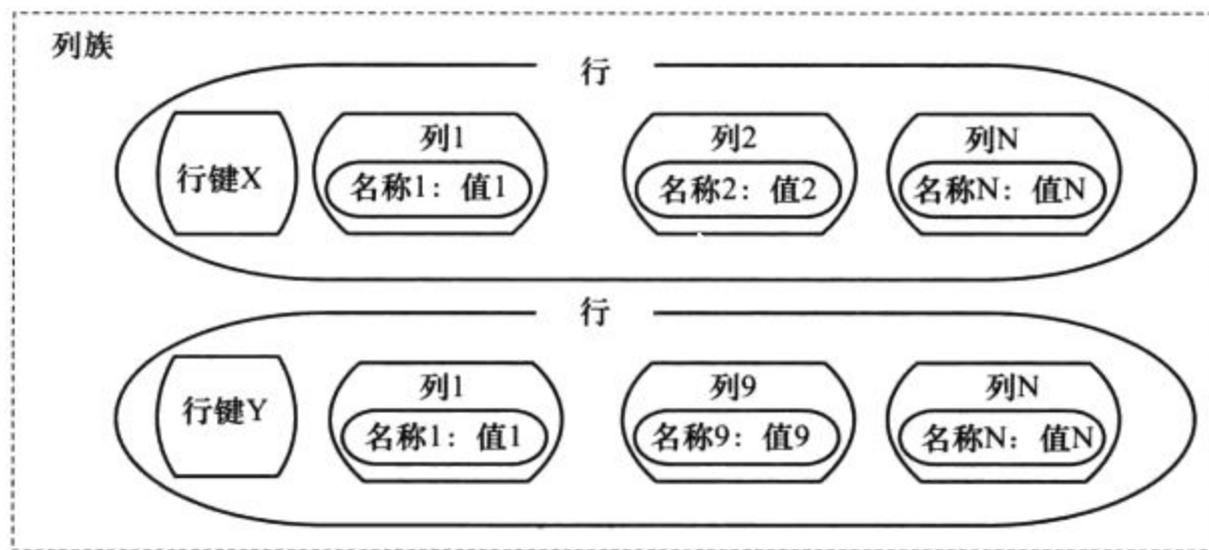


图 10.1 Cassandra 数据库所使用的列族数据模型

Cassandra 是一款流行的列族数据库，此外还有 HBase、Hypertable 和 Amazon DynamoDB [Amazon DynamoDB] 等其他产品。Cassandra 可以说是一种能快速执行跨集群写入操作并易于对此扩展的数据库。集群中没有主节点，其中每个节点均可处理读取与写入请求。

## 10.2 特性

现在看看 Cassandra 如何安排数据的结构。它的基本存储单元叫做“列”。Cassandra 的列由一个“名值对”（name-value pair）组成，其中的名字也充当关键字。每个键值对都占据一列，并且都存有一个“时间戳”值。令数据过期、解决写入冲突、处理陈旧数据等操作都会用到时间戳。若某列数据不再使用，则数据库可于稍后的“压缩阶段”（compaction phase）回收其所占空间。

```
{
  name: "firstName",
  value: "Martin",
  timestamp: 12345667890
}
```

上面这个列的关键字是“firstName”，值是“Martin”，还附有一枚时间戳。行是列的集合，这些列都附在某个关键字名下，或与之相连。由相似行所构成的集合就是列族。如果列族中的列都是“简单列”（simple column），那么我们就叫它“标准列族”（standard column family）。

```
//column family
{
//row
  "pramod-sadalage" : {
    firstName: "Pramod",
    lastName: "Sadalage",
    lastVisit: "2012/12/12"
  }
//row
  "martin-fowler" : {
    firstName: "Martin",
    lastName: "Fowler",
    location: "Boston"
  }
}
```

每个列族都可以与关系型数据库的“行容器”（container of rows）相对照：两者都用关键字标识行，并且每一行都由多个列组成。其差别在于，列族数据库的各行不一定要具备完全相同的列，并且可以随意向其中某行加入一列，而不用把它添加到其他行中。“pramod-sadalage”和“martin-fowler”这两行所具有的列不同，而它们都是列族的一部分。

如果某列中包含一个由小列组成的映射表，那么它就是“超列”（super column）。它有名称和值，而值是一个由小列组成的映射表。可将超列视为“列容器”（container of columns）。

```
{
  name: "book:978-0767905923",
  value: {
    author: "Mitch Albon",
    title: "Tuesdays with Morrie",
    isbn: "978-0767905923"
  }
}
```

用超列构建的列族叫做“超列族”（super column family）。

```
//super column family
{
//row
name: "billing:martin-fowler",
value: {
  address: {
```

```

    name: "address:default",
    value: {
      fullName: "Martin Fowler",
      street:"100 N. Main Street",
      zip: "20145"
    }
  },
  billing: {
    name: "billing:default",
    value: {
      creditcard: "8888-8888-8888-8888",
      expDate: "12/2016"
    }
  }
}
//row
name: "billing:pramod-sadalage",
value: {
  address: {
    name: "address:default",
    value: {
      fullName: "Pramod Sadalage",
      street:"100 E. State Parkway",
      zip: "54130"
    }
  },
  billing: {
    name: "billing:default",
    value: {
      creditcard: "9999-8888-7777-4444",
      expDate: "01/2016"
    }
  }
}
}
}

```

超列族适合将相关数据存在一起。但是，对于那些大部分时间都用不到的列，Cassandra 仍然要将其反序列化，在这种情况下，它就不是最优方案了。

Cassandra 将标准列族和超列族都放入“键空间”（keyspace）里。“键空间”与关系型数据库中的“数据库”类似，与应用程序有关的全部列族都存放于此。必须先创建列空间，才能为其增添列族：

```
create keyspace ecommerce
```

### 10.2.1 一致性

Cassandra 收到写入请求后，会先将待写数据记录到“提交日志”（commit log）中，然后将其写入内存里一个名为“内存表”（memtable）的结构中。写入操作在写入“提交日志”及“内存表”后，就算成功了。写入请求成批堆积在内存中，并定期

写入一种叫做“SSTable”的结构中。该结构中的缓存一旦写入数据库，就不会再向其继续写入了。若其数据变动，则需新写一张 SSTable。无用的 SSTable 可由“压缩”（compaction）操作回收。

先讲讲“一致性”设定如何影响读取操作。若将“一致性”设为 ONE，并以此作为所有读取操作的默认值，那么当 Cassandra 收到读取请求后，会返回第一个副本中的数据——即便其是陈旧数据，也照样返回。若此数据陈旧，则启动“读取修复”（read repair）过程，使后续的读取操作皆能获得最近（也就是最新）数据。如果不关心数据是否陈旧，或是需要高效执行读取操作，那么采用低级别的“一致性”就好。

同理，如果以最低的“一致性”执行写入操作，那么 Cassandra 只将其写入一个节点的“提交日志”中，然后就向客户端返回响应。若需要极为高效的写入操作，并且不介意丢失某些写入的数据，那么将“一致性”设为 ONE 就可以了。此时，如果某节点在尚未将写入的数据复制到其他节点前出了故障，那么这些数据就会丢失。

```
quorum = new ConfigurableConsistencyLevel();
quorum.setDefaultReadConsistencyLevel(HConsistencyLevel.QUORUM);
quorum.setDefaultWriteConsistencyLevel(HConsistencyLevel.QUORUM);
```

若将读取与写入操作的“一致性”都设为 QUORUM，那么读取操作将在过半数的节点响应之后，根据时间戳返回最新的列数据给客户端，并通过“读取修复”操作把最新数据复制到那些陈旧的副本中。而“一致性”为 QUORUM 的写入操作则必须等所写数据传播至过半数的节点后，才能顺利结束其工作并通知客户端。

如果将“一致性”级别设为 ALL，那么全部节点就必须响应读取或写入操作，这将使集群失去容错能力：一旦某个节点故障，全部读取操作或写入操作都将阻塞并失败。因此，系统设计师应根据应用程序需求调整“一致性”级别。同一应用程序内部也会有不同的“一致性”需求，所以也可以针对每次操作来设定其“一致性”。例如，显示产品评论所需的“一致性”，就与读取客户所下最新订单状态不同。

在创建“键空间”时，可以配置存储数据用的副本数，它决定了数据的“复制因子”。若复制因子为 3，则数据将复制至 3 个节点上。使用 Cassandra 写入及读取数据时，若将“一致性”设为 2，则 R+W 的值就会大于复制因子（2+2>3），这使得读取操作与写入操作的“一致性”都比较好。

可以在“键空间”上执行“节点修复”（node repair）命令，这会迫使 Cassandra 将其负责的每一个关键字与其余副本相比对。由于此操作开销较大，所以有时可以只修

复一个或一组列族。

```
repair ecommerce
repair ecommerce customerInfo
```

若某节点故障，则其存储数据会移交给其他节点。而当它重新上线时，数据库会把变更后的数据交还此节点。这种技术叫做“提示移交”（hinted handoff），它可以帮助故障节点更快地恢复。

### 10.2.2 事务

Cassandra 没有传统意义上的“事务”，也就是那种可以封装多个写入操作并决定是否提交其数据变更的东西。Cassandra 的写入操作在“行”级别是“原子的”，也就是说，根据某个给定的行键向行中插入或更新多个列，将算作一个写入操作，它要么成功，要么失败。写入操作首先会写在“提交日志”及“内存表”中，只有它向这两者写入数据后，才算顺利执行完。假如某节点故障，稍后可根据“提交日志”将数据变更恢复至该节点中，这与 Oracle 数据库中的“重做日志”（redo log）<sup>①</sup>类似。

可用 ZooKeeper[ZooKeeper] 等外部的“事务”程序库同步读写操作。还有 Cages[Cages] 等程序库可把 ZooKeeper 形式的“事务”封装起来。

### 10.2.3 可用性

Cassandra 的设计极具“可用性”，因为集群里没有主节点，其中每个节点地位等同。减少操作请求的“一致性”级别，即可提升集群“可用性”。“可用性”受制于  $(R+W) > N$  这一公式（参见 5.5 节）。W 是成功执行写入操作所需的最小节点数，R 是顺利执行读取操作所需获取的最小应答节点数，而 N 是参与数据复制的节点数。对于某定值 N，可改变 R 与 W 的值，以调整“可用性”。

假设在 10 节点的 Cassandra 集群中，有一个复制因子为 3 的“键空间”（N=3）。如果 R=2 且 W=2，那么  $(2+2) > 3$ 。在此情况下，若有一个节点故障，则不影响“可用性”，因为数据还可以从其他两个节点中获得。若 W=2 而 R=1，则集群在两个节点故障时将无法写入，但仍可读取。同理，若 R=2 而 W=1，则集群在两个节点故障时仍可写入，但无法读取。我们可以使用  $R+W > N$  这一式子，在“一致性”与“可用性”

<sup>①</sup> 详情参见：[https://en.wikipedia.org/wiki/Redo\\_log](https://en.wikipedia.org/wiki/Redo_log)。——译者注

之间做出明智的权衡。

“键空间”与“读/写操作”应该按照需求来设置：要么提高写入操作的“可用性”，要么提高读取操作的“可用性”。

## 10.2.4 查询功能

由于 Cassandra 没有功能丰富的查询语言，所以在设计其数据模型时，应该优化列与列族，以提升数据读取速度。在列族中插入数据后，每行中的数据都会按列名排序。假如某一列的获取次数比其他列更频繁，那么为了性能起见，应该将其值用作行键。

### 10.2.4.1 基本查询

可以使用 Cassandra 客户端运行的基本查询包括：GET、SET 和 DEL。查询数据前，必须先执行“键空间”命令：`use ecommerce;`。这样的话，查询范围就局限于我们存放数据的这个“键空间”了。在“键空间”中使用列族前，必须先定义它。

```
CREATE COLUMN FAMILY Customer
WITH comparator = UTF8Type
AND key_validation_class=UTF8Type
AND column_metadata = [
{column_name: city, validation_class: UTF8Type}
{column_name: name, validation_class: UTF8Type}
{column_name: web, validation_class: UTF8Type}
];
```

已经建立好名为 Customer 的列族了，它里面有 name、city 和 web 三列。现在用 Cassandra 客户端向列族中插入数据：

```
SET Customer['mfowler']['city']='Boston';
SET Customer['mfowler']['name']='Martin Fowler';
SET Customer['mfowler']['web']='www.martinfowler.com';
```

也可以通过 Hector 客户端 [Hector]，用 Java 代码向列族中插入同样的数据：

```
ColumnFamilyTemplate<String, String> template =
    cassandra.getColumnFamilyTemplate();
ColumnFamilyUpdater<String, String> updater =
    template.createUpdater(key);
for (String name : values.keySet()) {
    updater.setString(name, values.get(name));
}
try {
    template.update(updater);
} catch (HectorException e) {
    handleException(e);
}
```

GET 命令可以读回刚才写入的数据。获取数据有多种方式，下面这个命令可获取整个列族：

```
GET Customer['mfowler'];
```

还可以只从列族中读出所需的一列：

```
GET Customer['mfowler']['web'];
```

获取特定列，要比获取整个列族更高效，因为只返回所需数据即可。这样做能节省很多数据传输，尤其是列族中的列数较多时。更新数据与读取数据一样，用 SET 命令为待修改的列设置新值就好。DEL 命令可以删去一列或整个列族。

```
DEL Customer['mfowler']['city'];
```

```
DEL Customer['mfowler'];
```

#### 10.2.4.2 高级查询与索引编订

Cassandra 的列族可以用关键字之外的其他列当索引。下列代码将 city 列定义为索引：

```
UPDATE COLUMN FAMILY Customer
WITH comparator = UTF8Type
AND column_metadata = [{column_name: city,
                        validation_class: UTF8Type,
                        index_type: KEYS}];
```

现在可以直接查询索引后的列了：

```
GET Customer WHERE city = 'Boston';
```

这些索引以“位映射图”（bit-mapped）<sup>⊖</sup>的形式实现，在列中频繁出现重复数值<sup>⊖</sup>的情况下，性能较好。

#### 10.2.4.3 Cassandra 查询语言（CQL）

Cassandra 支持一种类似 SQL 命令的查询语言，叫做“Cassandra 查询语言”（Cassandra Query Language，简称 CQL）。CQL 命令可以创建列族。

⊖ 一种使用“位映射图”（bitmap）的数据库索引技术，详情参见：[https://en.wikipedia.org/wiki/Bitmap\\_index](https://en.wikipedia.org/wiki/Bitmap_index)。——译者注

⊖ 这里原文是“for low-cardinality column values”，其中 low-cardinality 意为“低基数的”。此类数据其取值范围或取值种数较少，例如布尔数据就是“低基数的”数据，因为其取值只有“真”、“假”两种可能。详情参见：[https://en.wikipedia.org/wiki/Cardinality\\_\(SQL\\_statements\)](https://en.wikipedia.org/wiki/Cardinality_(SQL_statements))。——译者注

```
CREATE COLUMNFAMILY Customer (
  KEY varchar PRIMARY KEY,
  name varchar,
  city varchar,
  web varchar);
```

也可以用 CQL 插入与前面相同的数据：

```
INSERT INTO Customer (KEY,name,city,web)
VALUES ('mfowler',
       'Martin Fowler',
       'Boston',
       'www.martinfowler.com');
```

SELECT 命令可读取数据。下述命令读出全部列：

```
SELECT * FROM Customer
```

SELECT 命令也可以只读取需要的列：

```
SELECT name,web FROM Customer
```

可以用 CREATE INDEX 命令为列创立索引，然后就能据此查询数据了：

```
SELECT name,web FROM Customer WHERE city='Boston'
```

CQL 中还有很多查询数据的功能，不过它并未包含 SQL 的全部功能。CQL 不支持“连接”（JOIN）及“子查询”（subquery），而且其 where 子句通常也比较简单。

### 10.2.5 可扩展性

在已有的 Cassandra 集群中扩展，也就意味着增加更多节点。由于不存在主节点，所以向集群中新增节点后，即可改善其服务能力，令其可以处理更多的写入及读取操作。这种横向扩展可以尽力提高其正常运行时间，因为集群在新增节点时，仍能处理客户端请求。

## 10.3 适用案例

现在讨论几个适合用列族数据库解决的问题。

### 10.3.1 事件记录

由于列族数据库可存放任意数据结构，所以它很适合用来保存应用程序状态或运行中遇到的错误等事件信息。在企业级环境下，所有应用程序都可以把事件写入 Cassandra 数据库。它们可以用 appname: timestamp（应用程序名：时间戳）作为行键，

并使用自己需要的列。由于 Cassandra 的写入能力可扩展，所以在事件记录系统中使用它效果会很好（参见图 10.2）。

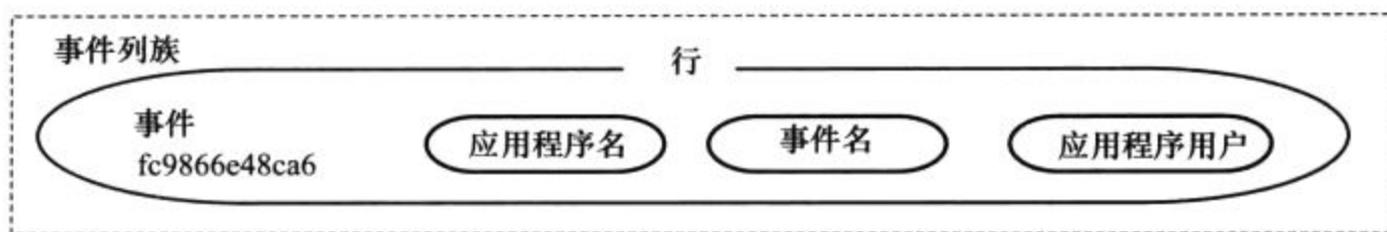


图 10.2 用 Cassandra 记录事件

### 10.3.2 内容管理系统与博客平台

使用列族，可以把博文的“标签”（tag）、“类别”（category）、“链接”（link）和“trackback”<sup>⊖</sup>等属性放在不同的列中。评论信息既可以与上述内容放在同一行，也可以移到另一个“键空间”。同理，博客用户与实际博文亦可存于不同列族中。

### 10.3.3 计数器

在网络应用程序中，通常要统计某页面的访问人数并对其分类，以算出分析数据。此时可使用 CounterColumnType 来创建列族。

```
CREATE COLUMN FAMILY visit_counter
WITH default_validation_class=CounterColumnType
AND key_validation_class=UTF8Type AND comparator=UTF8Type;
```

创建好列族后，可以使用任意列记录网络应用程序中每个用户访问每一页面的次数。

```
INCR visit_counter['mfowler'][home] BY 1;
INCR visit_counter['mfowler'][products] BY 1;
INCR visit_counter['mfowler'][contactus] BY 1;
```

也可以用 CQL 增加计数器的值：

```
UPDATE visit_counter SET home = home + 1 WHERE KEY='mfowler'
```

### 10.3.4 限期使用

我们可能需要向用户提供试用版，或是在网站上将某个广告条显示一定时间。这些功能可以通过“带过期时限的列”（expiring column）来完成。这种列过了给定时限后，就会由 Cassandra 自动删除。这个时限叫做 TTL（Time To Live，生存时间），以秒

<sup>⊖</sup> 俗称“引用告知”，是一种网志工具，它可以让文章作者知道该文读者中有哪些人撰写了哪些与之有关的文章。详情参见：<https://zh.wikipedia.org/wiki/TrackBack>。——译者注

为单位。经过 TTL 指定的时长后，这种列就被删掉了。程序若检测到此列不存在，则可收回用户访问权限或移除广告条。

```
SET Customer['mfowler']['demo_access'] = 'allowed' WITH ttl=2592000;
```

## 10.4 不适用场合

有些问题用列族数据库来解决并不是最佳选择，例如需要以“ACID 事务”执行写入及读取操作的系统。如果想让数据库根据查询结果来聚合数据（例如 SUM（求和）或 AVG（求平均值）），那么得把每一行数据都读到客户端，并在此执行操作。

在开发早期原型或刚开始试探某个技术方案时，不太适合用 Cassandra。开发初期无法确定查询模式的变化情况，而查询模式一旦改变，列族的设计也要随之修改。这将阻碍产品创新团队的工作并降低开发者的生产能力。在关系型数据库中，数据模式的修改成本很高，而这却降低了查询模式的修改成本；Cassandra 则与之相反，改变其查询模式要比改变其数据模式代价更高。

# 第 11 章

## 图数据库

图数据库可存放实体及实体间关系。实体也叫“节点”（**node**），它们具有属性（**property**）。可将节点视为应用程序中某对象的实例。关系又叫“边”（**edge**），它们也有属性。边具备方向性（**directional significance**），而节点则按关系组织起来，以便在其中查找所需模式。用图将数据一次性组织好，稍后便可根据“关系”以不同方式解读它。

## 11.1 何谓图数据库

在图 11.1 所举例子中，有大量互相关联的节点。节点就是带有 name 等属性的实体。Martin 就是个节点，它有一个值为 Martin 的 name 属性。

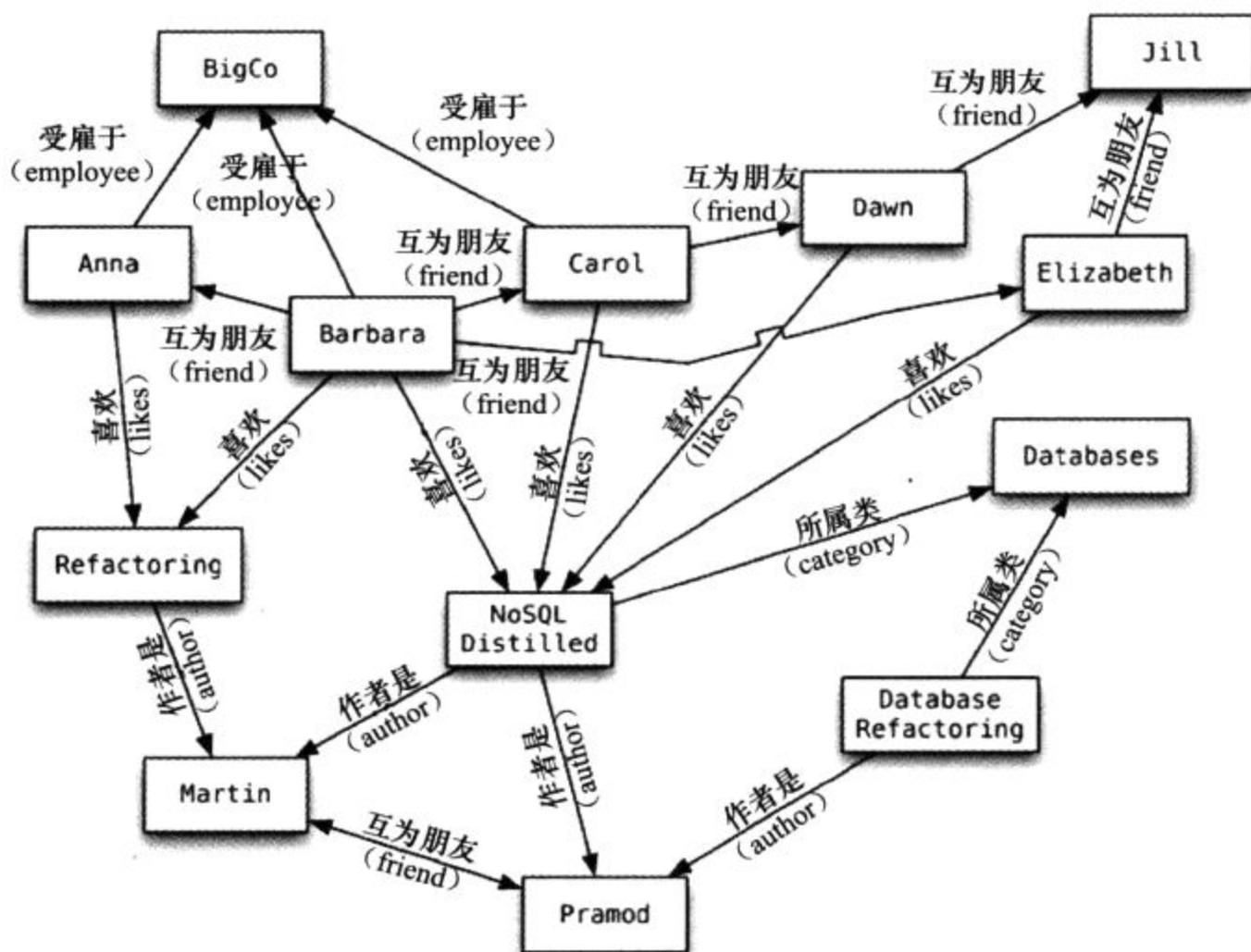


图 11.1 图结构示例

边也有类型，例如 likes（喜欢）、author（作者是）等。可依这些属性来组织节点。例如，有一条边连接了 Martin 节点与 Pramod 节点，其关系类型（relationship type）是 friend（互为朋友）。边可具备多个属性。可以把 since（从何时起）属性赋给 Martin 与 Pramod 之间那条关系类型为 friend 的边。关系类型有方向性。friend 关系类型是双向的，而 likes 则不是。Dawn 喜欢（likes）NoSQL Distilled 并不意味着 NoSQL Distilled 就一定喜欢 Dawn。

用节点和边建立好图之后，即可用多种方式查询它。例如，可“获取所有受雇于 Big Co 且喜欢 NoSQL Distilled 的节点”。查询图也称“遍历”（traverse）图。图数据库的一个好处就是，无需改变节点或边，即可应对新的遍历需求。假设现在想“获取所有喜欢 NoSQL Distilled 的节点”，那么不用改变已有数据或数据库模型，因为可以按照

需要随意遍历图。

在关系型数据库中存储的图状结构 (graph-like structure)，通常是单一关系类型 (常见的例子：“我的经理是谁”)。向已有数据中增加另一条关系，一般要修改许多模式并转移大批数据，而图数据库则不用这样。同理，关系型数据库必须根据需要遍历的内容 (Traversal) 提前建好图模型，若待遍历的内容改变，则数据也要随之变动。

图数据库遍历“连接”及“关系”非常快。节点间的关系不在查询时计算，而是在创建时已经持久化了。遍历持久化之后的关系，要比每次查询时都计算关系更快。

节点间可有多种不同的关系类型，这样既能表现领域实体 (domain entity) 之间的关系，也可以表示辅助关系 (secondary relationship)，例如：“类别” (category)、“路径” (path)、“时间树” (time-tree)、编订“空间索引” (spatial index) 用的“四叉树” (quad-tree)<sup>⊖</sup>或是“有序存取”所用的“链表” (linked list)。由于节点关系的数量及类型不限，所以这些关系可存放在同一图数据库中。

## 11.2 特性

图数据库有很多种，如 Neo4J[Neo4J]、Infinite Graph[Infinite Graph]、OrientDB[OrientDB] 和 FlockDB[FlockDB] (FlockDB 是个特例，它仅支持单深度的 (single-depth) 关系及邻接表 (adjacency list)，所以无法遍历深度超过 1 的关系) 等。本书以 Neo4J 为代表，讨论图数据库的工作原理，研究如何用它们来解决应用程序问题。

在 Neo4J 中创建图很简单，只需建立两个节点及其关系即可。现在创建两个名为 Martin 和 Pramod 的节点：

```
Node martin = graphDb.createNode();
martin.setProperty("name", "Martin");

Node pramod = graphDb.createNode();
pramod.setProperty("name", "Pramod");
```

我们将这两个节点的 name (名称) 属性分别赋值为 Martin 及 Pramod。只要节点数大于 1，即可创建关系：

<sup>⊖</sup> “空间索引”是“空间数据库” (spatial database) 所用的一种索引技术，而“四叉树”则是编订“空间索引”的一种方式。两者详情可分别参考：[https://en.wikipedia.org/wiki/Spatial\\_index](https://en.wikipedia.org/wiki/Spatial_index) 及 <https://zh.wikipedia.org/wiki/四叉树>。——译者注

```

martin.createRelationshipTo(pramod, FRIEND);
pramod.createRelationshipTo(martin, FRIEND);

```

必须从两个方向来创建节点间关系，因为关系的方向很重要。例如，product（产品）节点可以受 user（用户）欢迎（be liked by），但 product 不可能“喜欢”（like）user。方向性（directionality）有助于设计出丰富的领域模型（如图 11.2 所示）。可以根据传入关系（INCOMING relationship）与传出关系（OUTGOING relationship）双向遍历节点。

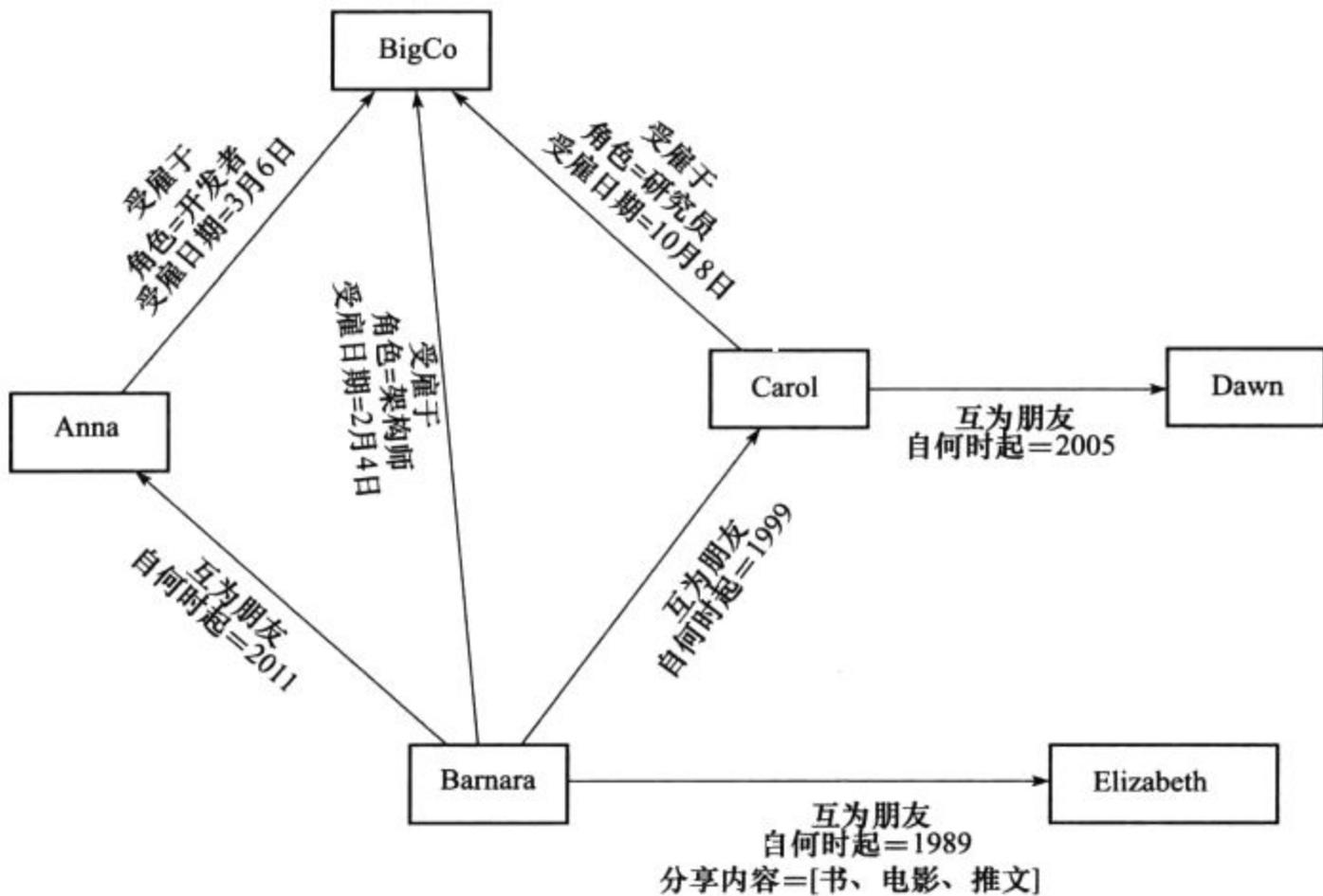


图 11.2 带属性的关系

关系是图数据库中的“一等公民”（first-class citizen），图数据库中的大多数值都源自关系。关系不只含有类型、起始节点和终止节点，而且还有自己的属性。使用这些属性，可令关系更智能。例如，可以指定两人之间何时成为朋友，两节点间距多远，两节点共享何种内容等。这些关系上的属性可用于查询图。

由于图数据库基本上得力于关系及其属性，因此在建模所要面对领域关系时，需做大量思考与设计。新增关系类型比较容易，而要改变已有节点及其关系，则相当于数据迁移（参见 12.3.2 节）了，因为现存数据的每个节点及关系都要变动。

### 11.2.1 一致性

由于图数据库操作互相连接的节点，所以大部分图数据库通常不支持把节点分布

在不同服务器上。然而，Infinite Graph 等某些解决方案，可以把节点分布在集群中的服务器上。在单服务器环境下，数据总是一致的，尤其是 Neo4J 这种完全兼容 ACID 事务的（fully ACID-compliant）数据库。如果在集群上运行 Neo4J，那么写入主节点的数据会逐渐同步至从节点，而读取操作则总是可在从节点执行。也可以向从节点写入数据，所写数据将立刻同步至主节点，但是其他从节点并不会立刻同步，它们必须等待由主节点传播过来的数据。

图数据库通过事务来保证“一致性”。它们不允许出现“悬挂关系”（dangling relationship）：所有关系必须具备起始节点与终止节点，而且在删除节点前，必须先移除其上的关系。

### 11.2.2 事务

Neo4J 是兼容 ACID 事务的数据库。修改节点或向现有节点新增关系前，必须先启动事务。若未将操作包装在事务中，则可能会抛出 `NotInTransactionException`。读取操作可不通过事务执行。

```
Transaction transaction = database.beginTx();
try {
    Node node = database.createNode();
    node.setProperty("name", "NoSQL Distilled");
    node.setProperty("published", "2012");
    transaction.success();
} finally {
    transaction.finish();
}
```

上述代码先在数据库上发起事务，然后创建节点并设置其属性。接下来，将事务标注为 `success`（成功），最后调用 `finish` 方法完成此事务。事务必须标注为 `success`，否则 Neo4J 就假定它失败了，并会在执行 `finish` 时回滚。仅设定 `success` 而不执行 `finish`，也会导致数据提交不到数据库。开发时要记住这种事务管理方式，因为它与关系型数据库的标准事务执行方式不同。

### 11.2.3 可用性

Neo4J 自 1.8 版本起，支持“副本从节点”（replicated slave），并借此获得较高的可用性。这些从节点可处理写入请求：向其写入后，它会先将所写数据同步至当前主节点。写入操作会先提交至主节点，然后再提交至从节点。其他从节点将逐渐获得更新数据。Infinite Graph 与 FlockDB 等图数据库则支持分布式节点存储。

Neo4J 使用 Apache ZooKeeper[ZooKeeper] 来记录每个从节点及当前主节点中最新的事务 ID。服务器启动后，将与 ZooKeeper 通信，以找出主服务器。若该服务器第一个加入集群，则它就成了主节点。当主节点故障时，集群将在可用节点中新选主节点，因此极具可用性。

#### 11.2.4 查询功能

图数据库可以使用 Gremlin[Gremlin] 等查询语言。Gremlin 是一门可以遍历图的领域特定语言（domain-specific language），它可以遍历所有实现了 Blueprints[Blueprints] 属性图（property graph）的图数据库。Neo4J 也可用 Cypher[Cypher] 查询语言来查询图。除了这些查询语言之外，Neo4J 还可以按节点属性查询图、遍历图，或通过“语言绑定”（language binding）<sup>⊖</sup> 浏览节点关系。

可以用“索引服务”（indexing service）来编订节点属性索引。同理，也可以索引关系或边的属性，以便根据属性值来查询。开始遍历图之前，应先查询索引以找出起始节点。现在讲讲如何以“编订节点索引”（node indexing）来搜寻节点。

有了图 11.1 中的图结构，在向数据库中添加节点时，就可以为其编订索引了，也可以稍后遍历全部节点时再对其索引。首先需要用 IndexManager 为节点创建索引。

```
Index<Node> nodeIndex = graphDb.index().forNodes("nodes");
```

我们使用 name 属性来索引节点。Neo4J 以 Lucene[Lucene] 为其索引服务。稍后将会用到 Lucene 的全文搜索（full-text search）功能。创建新节点后，可将其加入索引。

```
Transaction transaction = graphDb.beginTx();
try {
    Index<Node> nodeIndex = graphDb.index().forNodes("nodes");
    nodeIndex.add(martin, "name", martin.getProperty("name"));
    nodeIndex.add(pramod, "name", pramod.getProperty("name"));
    transaction.success();
} finally {
    transaction.finish();
}
```

将节点加入索引这一操作，必须在事务中执行。节点编入索引后，即可用索引属性搜寻之。若想查找名叫 Barbara 的节点，可在索引中以 Barbara 为 name 属性值来搜寻。

```
Node node = nodeIndex.get("name", "Barbara").getSingle();
```

⊖ 是一种通过“黏合代码”（glue code）以某门编程语言访问程序库或服务的机制，详情参见：[https://en.wikipedia.org/wiki/Language\\_binding](https://en.wikipedia.org/wiki/Language_binding)。——译者注

下列代码可以获得 name 属性为 Martin 的节点，找到了节点，就可以得知其全部关系了。

```
Node martin = nodeIndex.get("name", "Martin").getSingle();
allRelationships = martin.getRelationships();
```

也可以分别获取 INCOMING（传入）与 OUTGOING（传出）关系。

```
incomingRelations = martin.getRelationships(Direction.INCOMING);
```

查询关系时也可以按其方向过滤搜索结果。假设在图 11.1 中，要找出所有“喜欢”（like）NoSQL Distilled 的人，那么可以先找到 NoSQL Distilled 节点，然后搜寻 Direction.INCOMING 关系。由于我们只想寻找与 NoSQL Distilled 有 LIKE（喜欢）关系的节点，所以在过滤搜索结果时，还要加上关系类型这一项标准。

```
Node nosqlDistilled = nodeIndex.get("name",
                                     "NoSQL Distilled").getSingle();
relationships = nosqlDistilled.getRelationships(INCOMING, LIKES);
for (Relationship relationship : relationships) {
likesNoSQLDistilled.add(relationship.getStartNode());
}
```

寻找节点及其直接关系较为容易，不过这也能用关系型数据库来做。图数据库真正强大之处在于，它可以从指定的起始节点开始，以任意深度遍历图。在待查节点同起始节点之间的关系深度大于 1 时，图数据库尤为有用。如果图的深度较大，那就更应该使用 Traverser 来遍历其关系了，遍历时可指明关系类型是 INCOMING（传入）、OUTGOING（传出），还是 BOTH（两者皆可）。还可以用 BREADTH\_FIRST（广度优先）或 DEPTH\_FIRST（深度优先）为 Order 值，告诉遍历器（traverser）先向两旁搜索还是先自顶向下搜索。遍历器必须有起始节点。在本例中，要不限深度地查找所有与 Barbara 有 FRIEND（朋友）关系的节点。

```
Node barbara = nodeIndex.get("name", "Barbara").getSingle();

Traverser friendsTraverser = barbara.traverse(Order.BREADTH_FIRST,
                                             StopEvaluator.END_OF_GRAPH,
                                             ReturnableEvaluator.ALL_BUT_START_NODE,
                                             EdgeType.FRIEND,
                                             Direction.OUTGOING);
```

friendsTraverser 可以找出所有与 Barbara 的关系类型为 FRIEND 的节点。待查节点深度不限，即使是“朋友的朋友”也行，不管中间经过多少层间接关系。这个遍历器会探寻整个树状结构（tree structure）。

图数据库的一个优点就是能找到两节点间的路径，也就是判断它们之间是否有多条路径，如果有，就找出全部路径或最短路径。在图 11.1 所示图结构中，Barbara 与 Jill 间有两条不同路径。下列代码可找到 Barbara 与 Jill 之间的这些路径及其距离。

```
Node barbara = nodeIndex.get("name", "Barbara").getSingle();
Node jill = nodeIndex.get("name", "Jill").getSingle();
PathFinder<Path> finder = GraphAlgoFactory.allPaths(
    Traversal.expanderForTypes(FRIEND, Direction.OUTGOING)
    , MAX_DEPTH);
Iterable<Path> paths = finder.findAllPaths(barbara, jill);
```

此功能可用于显示社交网络中任意两节点的关系。要寻找两节点间所有路径及每条路径的距离，首先要取得两者之间的路径列表。每条路径的长度等于图中从起始节点到目标节点所经“跳数”（number of hops）<sup>⊖</sup>。我们通常需要找出两节点间的最短路径。下列代码可在 Barbara 与 Jill 之间的两条路径中，找出距离较短者。

```
PathFinder<Path> finder = GraphAlgoFactory.shortestPath(
    Traversal.expanderForTypes(FRIEND, Direction.OUTGOING)
    , MAX_DEPTH);
Iterable<Path> paths = finder.findAllPaths(barbara, jill);
```

也可以将其他图算法运用到待搜寻的图中。例如可用 Dijkstra 算法 [Dijkstra's] 求出节点间最短路径或“开销最小的路径”（cheapest path）。

```
START beginingNode = (beginning node specification)
MATCH (relationship, pattern matches)
WHERE (filtering condition: on data in nodes and relationships)
RETURN (What to return: nodes, relationships, properties)
ORDER BY (properties to order by)
SKIP (nodes to skip frcm top)
LIMIT (limit results)
```

Neo4J 还提供了 Cypher 查询语言来查询图。Cypher 需要以一个节点“启动”（START）查询。起始节点可通过节点 ID 或节点 ID 列表指明，也可用索引查出。Cypher 使用 MATCH 关键字匹配关系中的模式，以 WHERE 关键字过滤节点或关系的属性。RETURN 关键字则指定了查询所返回的数据是节点，是关系，还是节点或关系中的字段。

Cypher 也提供了排序（ORDER）、聚合（AGGREGATE）、略过（SKIP）和限定（LIMIT）数据所用的方法。使用 -- 符号，即可找出图 11.2 中所有与 Barbara 相连的节点。

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)--(connected_node)
RETURN connected_node
```

⊖ Neo4J 的跳数等于路径中的节点数减 1，详情参见：[http://api.neo4j.org/current/org/neo4j/graphdb/Path.html#length\(\)](http://api.neo4j.org/current/org/neo4j/graphdb/Path.html#length())。——译者注

如果只想找出某个方向的节点，那么可用

```
MATCH (barbara)<--(connected_node)
```

找寻传入起始节点的待查节点，而用

```
MATCH (barbara)-->(connected_node)
```

找寻从起始节点传出的待查节点。也可以用：`RELATIONSHIP_TYPE (:关系类型)` 格式来匹配特定的关系，而返回值既可以是节点，也可以是其中字段。

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)-[:FRIEND]->(friend_node)
RETURN friend_node.name, friend_node.location
```

上述代码从 `Barbara` 节点开始，寻找所有关系类型为 `FRIEND`（朋友）的传出节点，并返回这些友人的名字。上面这段代码，其关系类型查询<sup>⊖</sup>的深度为 1，接下来改写它，令其搜索得更深一些<sup>⊖</sup>，并且在结果中返回每个朋友节点的深度。

```
START barbara=node:nodeIndex(name = "Barbara")
MATCH path = barbara-[:FRIEND*1..3]->end_node
RETURN barbara.name, end_node.name, length(path)
```

与之类似，也可以只搜寻存在某个特定属性的关系。可以用是否存在某关系属性为标准，过滤查询结果。

```
START barbara = node:nodeIndex(name = "Barbara")
MATCH (barbara)-[relation]->(related_node)
WHERE type(relation) = 'FRIEND' AND relation.share
RETURN related_node.name, relation.since
```

Cypher 语言中还有许多能查询图数据库的功能。

### 11.2.5 可扩展性

NoSQL 数据库最常用的扩展技术就是“分片”，也就是把数据分割并存放在不同服务器上。对图数据库分片比较难，因为它们不是面向聚合的（`aggregate-oriented`），而是面向关系的（`relationship-oriented`）。由于任何节点都可能关联其他节点，所以把相关节点放在同一台服务器中，遍历图时会更方便些。若图中的节点放在不同电脑上，则遍历性能不佳。尽管图数据库有此局限，我们仍可用 Jim Webber[Webber Neo4J Scaling]所说的一些常用技术扩展它们。

⊖ 指第二行的“[: FRIEND]”。——译者注

⊖ 指第二行的“[: FRIEND\*1..3]”。——译者注

图数据库一般有三种扩展方式。由于时下电脑内存<sup>①</sup>都较大，所以可给服务器配备足量内存，使之可完全容纳“工作集”（working set）<sup>②</sup>中的全部节点与关系。只有当存放工作数据集所需的内存量比较合理时，这项技术才能派上用场。

增加仅能读取数据的从节点，即可改善数据库读取能力，所有写入操作仍由主节点负责。使用此模式，只需向主节点写入一次数据，就能从许多服务器中读取了。这是一种成熟的 MySQL 集群技术，当数据集大到无法容纳于单机内存，而又足以在多台服务器之间复制时，这项技术尤为有用。我们可以配置从节点，让其只能用于读取数据而不会成为主节点，这样就有助于扩展数据库的“可用性”及读取能力了。

若数据集太大，导致多节点复制不太现实，那么可用“领域特定知识”（domain-specific knowledge）在应用程序端对其分片（参见 4.2 节）。例如，可将与北美有关的节点放在一台服务器上，而把和亚洲有关的节点放在另一台上。使用“应用程序级分片”（application-level sharding）时，必须明白，节点存放在地理位置不同的数据库中（见图 11.3）。

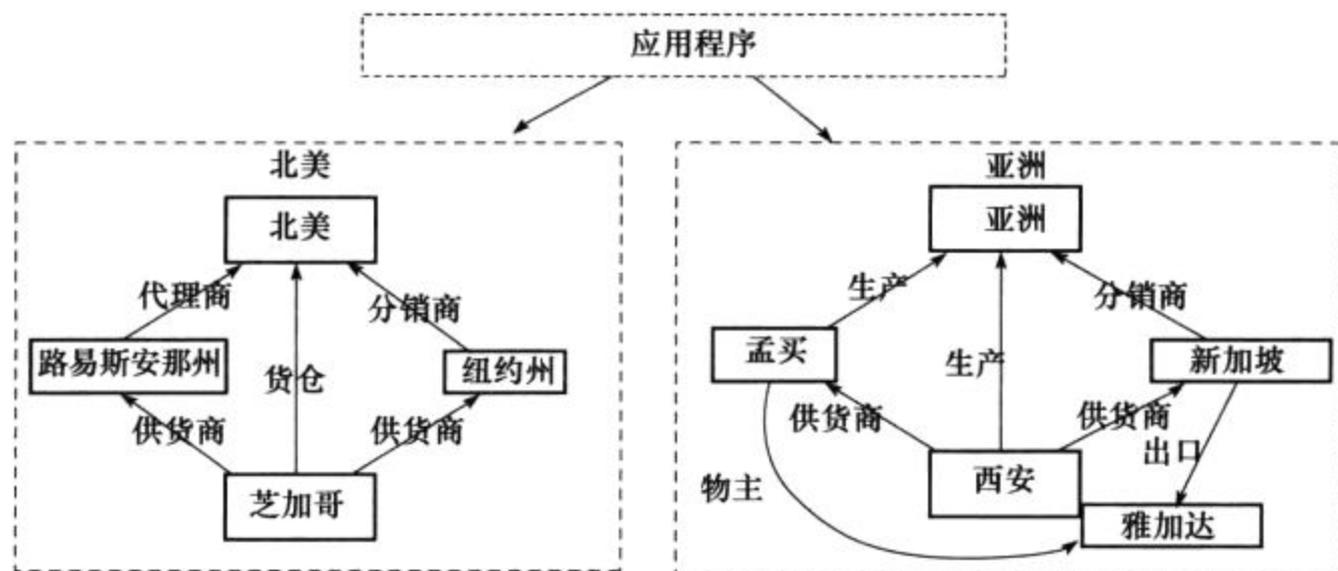


图 11.3 应用程序级节点分片

## 11.3 适用案例

接下来讲一些适合使用图数据库的用例。

### 11.3.1 互联数据

部署并使用图数据库来处理社交网络非常高效。社交图里并不是只能有“朋友”这种关系，例如也可以用它们表示雇员、雇员的学识，以及这些雇员与其他雇员在不

① 原文为 RAM，是 Random Access Memory 的简称，中译“随机存取存储器”，是与 CPU 直接交换数据的存储器。本书在不致混淆时，采用“内存”这一俗称代之。——译者注

② 进程在某时间段内所获取的内存分页集合，详情参见：[https://en.wikipedia.org/wiki/Working\\_set](https://en.wikipedia.org/wiki/Working_set)。——译者注

同项目中的工作位置。任何富含链接关系的领域都很适合用图数据库表示。

假如同一个数据库含有不同领域（像社交领域、空间领域、商务领域等）的领域实体，而这些实体之间又有关系，那么图数据库提供的跨领域遍历功能，可以让这些关系变得更有价值。

### 11.3.2 安排运输路线、分派货物和基于位置的服务

投递过程中的每个地点或地址都是一个节点，可以把送货员投递货物时所经全部节点建模为一张节点图。节点间关系可带有距离属性，以便高效投递货物。距离与位置属性也可用在名胜图（graph of places of interest）中，这样应用程序就可向用户推荐其附近的好餐馆及娱乐场所了。还可将书店、餐馆等销售点（point of sales）做成节点，当用户靠近时通知他们，以提供基于位置的服务。

### 11.3.3 推荐引擎

在系统中创建节点与关系时，可以用它们为客户推荐信息，例如“您的朋友也买了这件产品”或“给这些货品开发票时，通常也要为那些货品一并开票”。还可以用它们向旅行者提议：来巴塞罗那旅游的人一般都会去看看安东尼·高迪<sup>⊖</sup>所设计的建筑。

用图数据库推荐信息时，有个副作用值得注意：随着数据量变多，推荐信息所用的节点及关系数也激增。同一份数据可以挖掘出不同信息。例如，既可以从中看出客户总是将其与哪些产品一并购买，也可以查出与此产品一并开发票的其余产品。若两者不匹配，则可发出警示。图数据库与其他“推荐引擎”（recommendation engine）一样，也可以根据关系间的模式侦测交易欺诈（fraud in transaction）。

## 11.4 不适用场合

图数据库在某些情形下也许不适用。在更新全部或某子集内的实体时就是这样。比如，在某个“数据分析解决方案”（analytics solution）中，只要一个属性变了，全部实体就都得更新。此时图数据库的效果就不理想了，因为没有哪个简单的操作能一次性改变所有节点中的某个属性。即便数据模型适合问题领域，某些图数据库可能也无法处理那么大的数据量，尤其在执行“全局图操作”（global graph operation，涉及整张图的操作）时更是如此。

⊖ Antoni Gaudí i Cornet (1852-1926)，西班牙“加泰罗尼亚现代主义”建筑家，新艺术运动的代表人物。其设计的圣家大教堂（Sagrada Família）是巴塞罗那旅游胜地。详情参见：<https://zh.wikipedia.org/wiki/安东尼·高迪>。——译者注

# 第 12 章

## 模式迁移

## 12.1 模式变更

时下讨论 NoSQL 数据库时，大家乐于强调其“无模式（schemaless）性”。这一特性颇受欢迎，开发者可专注于领域设计而无需担心模式变化。注重应对需求变化的“敏捷方法”<sup>①</sup> [Agile Method] 兴起后，尤其如此。

为了正确理解数据，在讨论、迭代、反馈回路（feedback loop）等过程中，领域专家（domain expert）及产品拥有者（product owner）要参与其中。数据库模式的复杂度不应妨碍讨论过程。改变 NoSQL 数据库的模式不会遇到多少麻烦，这也增进了开发者的生产效率（参见 1.5 节）。我们发现，在无模式数据库的“美丽新世界”（brave new world）中开发与维护应用程序时，要谨慎应对模式迁移。

## 12.2 变更关系型数据库的模式

使用标准关系型数据库技术开发时，要开发的是对象，它们所对应的表，以及对象间的关系。思考一套简单的对象模型与数据模型，它包含 Customer（客户）、Order（订单）与 OrderItems（订单项）。图 12.1 画出了“实体关系模型”（ER model, Entity-Relationship model 的缩写）。

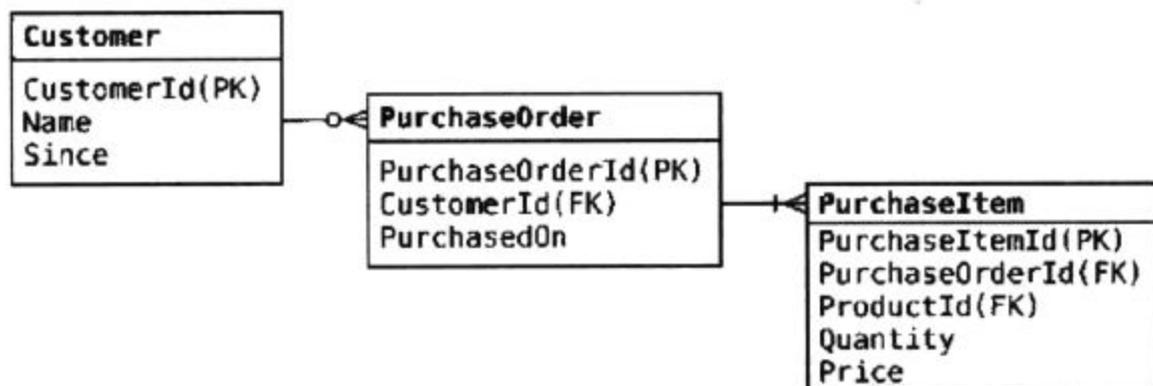


图 12.1 电子商务系统的数据模型

如果数据模型支持现有的对象模型，那么一切还好。然而一旦想改变对象模型，比如给 Customer 对象加入 preferredShippingType（偏好的送货类型）属性，那么就必须改变对象及数据库表了。要是不改动表，那么应用程序就和数据库不同步了。当发生“ORA-00942: table or view does not exist”或“ORA-00904: “PREFERRED\_SHIPPING\_TYPE”: invalid identifier”<sup>②</sup>等错误时，就表明出现此类问题了。

<sup>①</sup> Agile Method 是 Agile Software Development Methodology 的简称，全译“敏捷软件开发方法学”。——译者注

<sup>②</sup> 以“ORA-”开头的是 Oracle 数据库错误代码，详情可查询：<http://www.ora-error.com>。这两个错误的大意是“表或视图不存在”、“无效标识符：‘PREFERRED\_SHIPPING\_TYPE’”。——译者注

数据库模式迁移通常本身就是一个项目。为了部署模式变更，要编写“数据库变更脚本”（database change script），使用“diff技术”（diff technique）比对“开发数据库”（development database）在模式迁移前后所发生的改变。在“部署/发行”（deployment/release）阶段采用此方式创建“迁移脚本”（migration script）容易出错，而且无法运用各种敏捷开发方式。

### 12.2.1 迁移全新项目

数据库模式变更脚本最好能在开发阶段编写，因为这样做可以把模式变更信息与数据迁移脚本存在同一份脚本文件中。脚本文件可用递增数字命名，以体现数据库版本。例如，第一次修改数据库的脚本可叫做 001\_Description\_Of\_Change.sql。以此方式修改脚本，可在数据库迁移过程中维护变更顺序。图 12.2 中的文件夹列出了当前数据库的历次修改过程。

现在假定要修改 OrderItem（订单项）表，以保存每项商品的 DiscountedPrice（折扣价）与 FullPrice（全价）。修改 OrderItem 表的操作脚本将冠以数字 007，并保存至变更序列中，如图 12.3 所示。

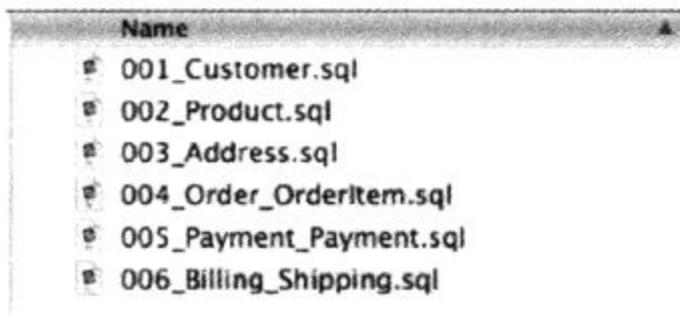


图 12.2 数据库所经历的多次迁移

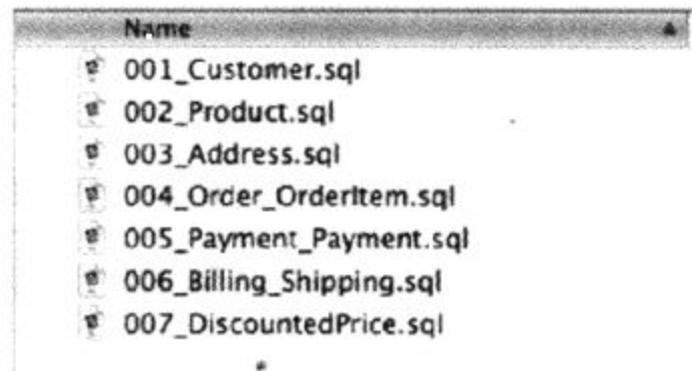


图 12.3 对数据库运用了新的变更脚本 007\_DiscountedPrice.sql

我们再次修改数据库，用变更脚本中的代码新增了一列数据，给已有的列改名，并为实现新功能而迁移所需数据。下面列出变更脚本 007\_DiscountedPrice.sql 中的相关代码：

```
ALTER TABLE orderitem ADD discountedprice NUMBER(18,2) NULL;
UPDATE orderitem SET discountedprice = price;
ALTER TABLE orderitem MODIFY discountedprice NOT NULL;
ALTER TABLE orderitem RENAME COLUMN price TO fullprice;
--//@UNDO
ALTER TABLE orderitem RENAME fullprice TO price;
ALTER TABLE orderitem DROP COLUMN discountedprice;
```

“变更脚本”（change script）既包含数据库的模式修改，又包括所需的数据迁移操作。本例使用 DBDeploy[DBDeploy] 作为管理数据库变更的框架。DBDeploy 在数据库中维护一张名为 ChangeLog 的表，所有数据库变更都记录于此。表中的 Change\_Number 表示数据库经历过的变更次数。这个 Change\_Number 也就是数据库版本，它可用于在文件夹中寻找以数字开头的相关脚本，并将其中尚未执行的脚本运用于数据库。假设写好了一个变更序号为 007 的脚本，并通过 DBDeploy 将其运用至数据库，此时 DBDeploy 会检测 ChangeLog 表，据此执行文件夹中所有尚未应用的脚本。图 12.4 是 DBDeploy 向数据库运用变更脚本时的运行截图。

```

project $>ant dbupgrade
Buildfile: /project/build.xml

init:

dbupgrade:
[dbdeploy] dbdeploy 3.0M3
[dbdeploy] Reading change scripts from directory /project/db/migrations...
[dbdeploy] Changes currently applied to database:
[dbdeploy] 1..6
[dbdeploy] Scripts available:
[dbdeploy] 1..7
[dbdeploy] To be applied:
[dbdeploy] 7
[dbdeploy] Applying #7: 007_DiscountedPrice.sql...
[dbdeploy] -> statement 1 of 4...
[dbdeploy] -> statement 2 of 4...
[dbdeploy] -> statement 3 of 4...
[dbdeploy] -> statement 4 of 4...

BUILD SUCCESSFUL
Total time: 0 seconds
project $>

```

图 12.4 DBDeploy 以 007 号变更脚本升级数据库

与其他开发者合作的最佳集成方式是采用项目的版本控制库（version control repository）来保存所有变更脚本，这样就可以同时记录软件版本与数据库版本，以避免数据库与应用程序之间版本不合。还有其他一些工具能执行这种升级操作，例如 Liquibase[Liquibase]、MyBatis Migrator[MyBatis Migrator] 和 DBMaintain[DBMaintain] 等。

### 12.2.2 迁移既有项目

并非每个项目都是全新的<sup>⊖</sup>。如何在一个已经投入生产状态的既有应用程序中实现

⊖ 原文为“Not every project is green field.”。12.2.1 节标题中有“Green Field Project”一词，指不受早前工作之约束的全新项目，中译“绿地项目”或“绿野项目”，详情参见：[https://en.wikipedia.org/wiki/Greenfield\\_project](https://en.wikipedia.org/wiki/Greenfield_project)。本节标题中“Legacy Project”一词与之相对，该词中译“遗留项目”。——译者注

数据迁移呢？我们发现，可以把已有数据库的结构提取到脚本里，并把所有数据库代码及全部“引用数据”（reference data）也都放在其中，然后用该脚本作为项目基线（baseline for the project）。“事务数据”（transactional data）不含于基线内。准备好基准版本后，即可采用前述各项迁移技术继续修改数据库了（如图 12.5 所示）。

迁移过程中的一个重要问题就是应该维护数据库模式的“向后兼容性”（backward compatibility）。很多企业级数据库都为多个应用程序所用。其中某一程序改变数据库之后，不应

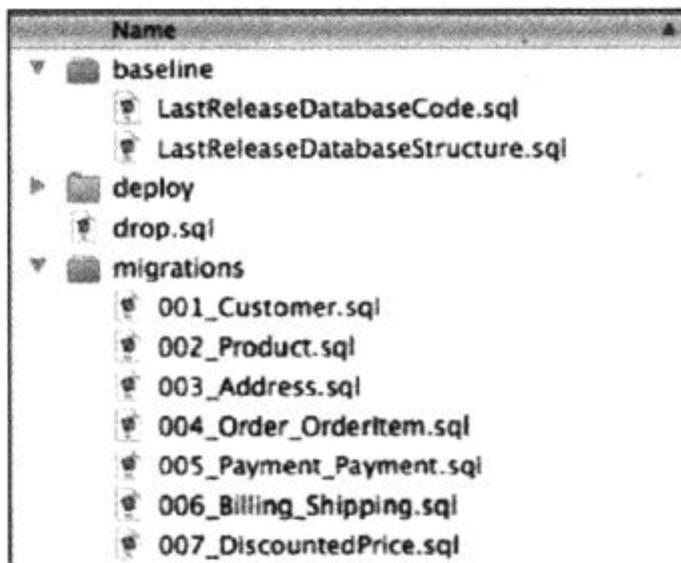


图 12.5 在既有数据库中使用基准脚本

该影响其余程序正常运作。变更数据时可引入一个“过渡阶段”（transition phase）以解决此问题，详情参见《Refactoring Databases》一书 [Ambler and Sadalage]。

在“过渡阶段”（transition phase），旧模式与新模式并行，所有使用数据库的应用程序都可以采用它们。为此必须引入“触发器”（trigger）、“视图”（view）和“虚拟列”（virtual column）等“脚手架代码”（scaffolding code）<sup>⊖</sup>，以确保其他应用程序无需修改代码，即可访问数据库模式及其数据。

```
ALTER TABLE customer ADD fullname VARCHAR2(60);
UPDATE customer SET fullname = fname;

CREATE OR REPLACE TRIGGER SyncCustomerFullName
BEFORE INSERT OR UPDATE
ON customer
REFERENCING OLD AS OLD NEW AS NEW
FOR EACH ROW
BEGIN
    IF :NEW.fname IS NULL THEN
        :NEW.fname := :NEW.fullname;
    END IF;
    IF :NEW.fullname IS NULL THEN
        :NEW.fullname := :NEW.fname;
    END IF;
END;
/

--Drop Trigger and fname
--when all applications start using customer.fullname
```

⊖ 描述应用程序如何使用其数据的模板代码，详情参见：[https://en.wikipedia.org/wiki/Scaffold\\_programming](https://en.wikipedia.org/wiki/Scaffold_programming)。——译者注

本例中，由于 `fname` 既可理解成 `fullname`（全名），又可理解成 `firstname`（首名），所以为了避免歧义，我们想将 `customer.fname` 列改名为 `customer.fullname`。当然可以直接修改 `fname` 列的名字以及我们所负责的应用程序代码，不过这样做只方便了自己的应用程序，而对使用同一企业级数据库的其他应用程序则无法奏效。

使用“过渡阶段”这一技巧时，我们引入名为 `fullname` 的新列，将数据复制到这个 `fullname` 列中，然而保留原有的 `fname` 列。同时还引入“BEFORE UPDATE”触发器，在将这两列数据提交至数据库之前，先对其进行同步。

现在，应用程序读取数据表时，不论使用 `fname` 列还是 `fullname` 列，都能获取正确的数据。等所有应用程序都转而使用新的 `fullname` 列之后，就可以删除触发器及 `fname` 列了。

关系型数据库中的大数据集很难迁移其模式。如果想在迁移时令应用程序仍然可以访问数据库，那就更难了，因为移动大量数据并改变其结构，通常需要锁定数据库表。

### 12.3 变更 NoSQL 数据库的模式

在改变应用程序之前，必须先变更其关系型数据库。而“无模式”（`schema-free` 或 `schemaless`）数据库恰恰试图避免这一问题，它们致力于按实体来自由变更其模式。为了因应频繁的市场变化及产品创新，必须经常改换数据库模式。

用 NoSQL 数据库开发时，某些情况下并不需要提前考虑模式。我们仍然需要设计并思考数据库的其他方面：比方说，使用图数据库时，要思考关系的类型；使用列族数据库时，得考虑列族、行、列的名字以及列的顺序；使用键值数据库时，则要想想如何设置关键字以及值对象具备何种数据结构。即便不预先考虑上述问题，或是需要改变其决定，使用 NoSQL 数据库也比较容易。

NoSQL 完全没有模式这种说法会误导人。虽说无论数据遵从何种模式，NoSQL 数据库都能存储，但是应用程序却必须定义模式，因为从数据库中读取数据时，应用程序必须解析数据流。而且，要保存至数据库中的数据必须由应用程序建立。要是应用程序无法从数据库中解析数据，那么还是会出现“模式不匹配”（`schema mismatch`）现象，虽然不像关系型数据库那样由数据库本身抛出错误，但应用程序仍会出错。因此，即便采用无模式数据库，重构应用程序时也依然要考虑数据模式。

若应用程序已部署，而且已存有产品数据，那么就更要注意模式的变化。为简洁

起见，假定使用 MongoDB[MongoDB] 这种文档数据库，而且数据模型也和原来一样，由 customer、order 与 orderItems 组成。

```
{
  "_id": "4BD8AE97C47016442AF4A580",
  "customerid": 99999,
  "name": "Foo Sushi Inc",
  "since": "12/12/2012",
  "order": {
    "orderid": "4821-UXWE-122012", "orderdate": "12/12/2001",
    "orderItems": [{"product": "Fortune Cookies",
                    "price": 19.99}]
  }
}
```

将此文档结构写入 MongoDB 所用的应用程序代码如下：

```
BasicDBObject orderItem = new BasicDBObject();
orderItem.put("product", productName);
orderItem.put("price", price);
orderItems.add(orderItem);
```

从数据库中读回此文档，可用下列代码：

```
BasicDBObject item = (BasicDBObject) orderItem;
String productName = item.getString("product");
Double price = item.getDouble("price");
```

如果想修改对象，为其新增 preferredShippingType 属性，那么无需修改数据库，因为数据库不在乎不同文档是否遵循同一模式。这使得开发应用程序更加迅速而部署起来也较为方便。要部署的只是应用程序而已，数据库一端无需修改。代码只要确保不含 preferredShippingType 属性的文档仍然能照常解析就好了。

当然了，我们这里要讲的模式修改较为简单。就以上一节所举的模式变更为例：引入 discountedPrice 属性，并将价格（price）属性改名为 fullPrice。要修改数据库模式，只需将 price 属性改为 fullPrice 并新增 discountedPrice 属性即可。修改后的文档是：

```
{
  "_id": "5BD8AE97C47016442AF4A580",
  "customerid": 66778,
  "name": "India House",
  "since": "12/12/2012",
  "order": {
    "orderid": "4821-UXWE-222012",
    "orderdate": "12/12/2001",
    "orderItems": [{"product": "Chair Covers",
                    "fullPrice": 29.99,
                    "discountedPrice": 26.99}]
  }
}
```

将修改过的数据部署好之后，就可以正常保存并读取新顾客及其订单了，然而从

已有的订单中却无法读出产品价格，因为应用程序代码现在要找的是 fullPrice 属性，而旧文档里却只有 price 属性。

### 12.3.1 增量迁移

很多 NoSQL 新手都会犯模式不匹配的错误。在应用程序改变模式后，必须把既有数据全都转化成新模式才行（若数据较多，则该操作可能开销较大）。另一个办法是，确保改换模式之前的那些旧数据仍然可以为新代码所解析，而在保存时，将其以新模式写回数据库。这种渐进迁移数据的技术，就叫“增量迁移”（incremental migration）。某些数据也许从来不会迁移，因为没人访问它们。下列代码可以读取文档中的 price 与 fullPrice 属性：

```
BasicDBObject item = (BasicDBObject) orderItem;
String productName = item.getString("product");
Double fullPrice = item.getDouble("price");
if (fullPrice == null) {
    fullPrice = item.getDouble("fullPrice");
}
Double discountedPrice = item.getDouble("discountedPrice");
```

在把数据写回文档时，就不保存旧的 price 属性了：

```
BasicDBObject orderItem = new BasicDBObject();
orderItem.put("product", productName);
orderItem.put("fullPrice", price);
orderItem.put("discountedPrice", discountedPrice);
orderItems.add(orderItem);
```

执行增量迁移时，应用程序端可能会存在多个对象版本，它们都可将旧模式转译为新模式。在把对象存回数据库时，则使用新的对象格式。这种渐进式数据迁移能帮应用程序更快地演化。

增量迁移技术会令对象设计变复杂，尤其是新的修改已经引入，而旧的修改尚未执行完时。自开始部署修改，到数据库中最后一个旧对象迁移至新模式为止，中间的这段时间叫做“过渡期”（transition period，如图 12.6 所示）。过渡期的时间越短越好，而且其范围也要尽量缩小，这样才能让保持对象整洁。

在数据中使用 schema\_version 字段，也可

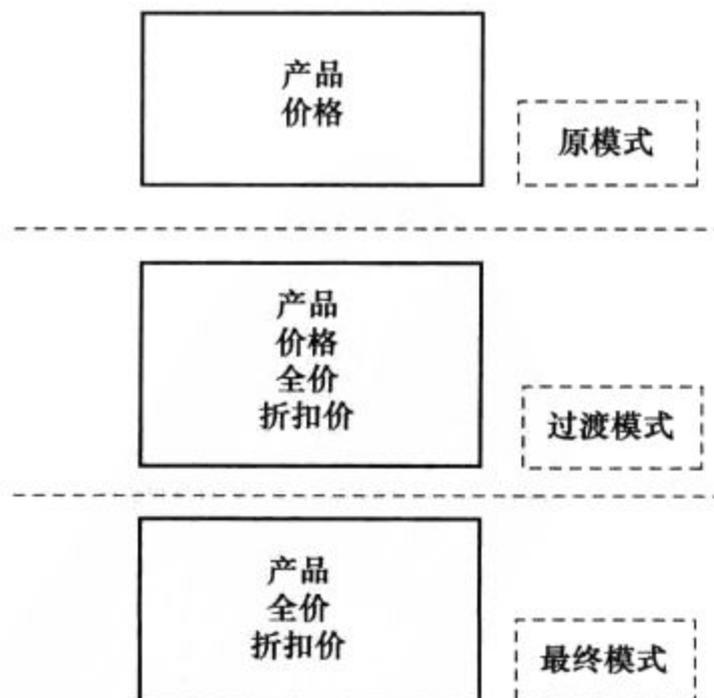


图 12.6 模式变更的过渡期

以实现增量迁移技术。应用程序将据此选择适当代码来把数据解析为对象。保存数据时，程序会将其迁移到最新版，并把 `schema_version` 字段更新为与之相符的值。

在领域和数据库之间一定要有适当的“转译层”（translation layer）。这样的话，在模式变更时，管理多个模式版本的代码就会局限于转译层而不会弥漫到整个应用程序之中了。

移动应用程序需求特殊。由于不便强迫所有用户都升级至最新版程序，所以应用程序应该差不多要能处理所有版本的模式才行。

### 12.3.2 迁移图数据库的模式

图数据库的边带有类型及属性。若改变了代码库里这些边的类型，则无法遍历数据库，于是也就无法使用其数据了。为了解决此问题，可以遍历每一条边并更改其类型。此操作也许开销较高，而且必须写代码来迁移数据库中的所有边才行。

如果想向后兼容，或不愿一次性改动整张图，那么可以在节点间创建新的边。稍后若对此修改满意，则可删掉旧的边。可依多种边类型标准来遍历图，既能按照旧的边类型遍历，也能以新类型遍历。这种技术处理大数据库很有用，尤其是想在数据迁移过程中维持“高可用性”时。

若要修改所有节点或边的属性，则必须获取全部节点并修改每一个待更动的属性。比方说，为了记录每个节点的修改操作，需要给全部既有节点新增 `NodeCreatedBy`（节点创建者）及 `NodeCreatedOn`（节点创建时间）属性。

```
for (Node node : database.getAllNodes()) {
    node.setProperty("NodeCreatedBy", getSystemUser());
    node.setProperty("NodeCreatedOn", getSystemTimeStamp());
}
```

节点中的数据也许会改变，新数据可能衍生自现存数据，也可能从其他某个数据源引入。迁移数据时，可以根据数据源提供的索引获取所有节点，并将相关数据写入每个节点中。

### 12.3.3 改变聚合结构

有时需要修改模式设计，比如将大对象分割为独立存储的小对象。假定原来的客户聚合里包含全部客户订单，现在需要把每位客户和其所下订单分至不同的聚合单元中。

此时就要保证代码能够处理两种版本的聚合。如果找不到旧版对象，那么就从新版聚合中读取数据。

迁移数据时，运行在后台的代码一次可以只读取一个聚合，修订好待改之处后，再将其存回不同的聚合里面。一次只操作一个聚合的好处是，它不影响应用程序的“数据可用性”（data availability）。

## 12.4 延伸阅读

[Ambler and Sadalage] 一书详述了如何迁移关系型数据库。尽管其中许多内容都针对关系型数据库，但是迁移操作的一般原理亦适用于其他数据库。

## 12.5 要点

- 若要迁移关系型数据库等“强模式”（strong schema）数据库，可将历次模式变更及其数据迁移操作保存于“版本控制序列”（version-controlled sequence）中。
- 因程序代码要依照“隐含模式”（implicit schema）访问无模式数据库（schemaless database）的数据，故其数据迁移仍需谨慎处理。
- 无模式数据库亦可借用强模式数据库的迁移技术。
- 无模式数据库可以用“增量迁移”技术更新数据，以便在不影响应用程序读取其数据的前提下，修改数据的隐含模式。

## 第 13 章

# 混合持久化

不同的数据库用来解决不同的问题。只用一种“数据库引擎”（database engine）应对所有需求，这种解决办法较为低效。保存“事务数据”（transactional）、缓存“会话信息”（session information）、遍历客户图并找出其友人购买的产品，这三者根本就是各不相同的问题。就算在关系型数据库领域内部，OLAP<sup>⊖</sup>与 OLTP<sup>⊖</sup>系统其实差别也很大，只是通常被迫采用同一种数据模式而已。

现在思考数据关系：关系型数据库善于确保数据之间确实存在关系，然而如果要探寻关系，或是从不同表中找出属于同一对象的数据，那么使用关系型数据库就有些困难了。

各种数据库引擎都有其适合操作的数据结构及数据量。比方说，有些数据库擅长操作数据集（set of data），有些则适合存储并迅速获取键值对；有的数据库擅长存放“富文档”（rich document），而另一些则适合保存复杂的信息图（graph of information）。

---

⊖ 联机分析处理（On-Line Analytical Processing），是一套以多维度方式分析数据并呈现集成性决策信息的方法，多用于决策支持系统、商务智能或数据仓库。主要功能是方便大规模数据分析及统计计算，对决策提供参考和支持。详情参见：<https://zh.wikipedia.org/wiki/联机分析处理>。——译者注

⊖ 联机交易处理（Online transaction processing）是指通过信息系统、电脑网络及数据库，以联机交易方式处理实时作业数据。通常用于自动化数据处理，如订单输入、金融业务等反复的日常交易活动。详情参见：<https://zh.wikipedia.org/wiki/联机交易处理>。——译者注

## 13.1 各异的数据存储需求

许多企业不仅打算用同一个数据库引擎来保存“商业事务”（business transaction）和会话管理数据（session management data），而且还要用它存储报表（reporting）、商务智能（Business Intelligence，简称 BI）、数据仓库（data warehousing）和日志信息（logging information）等内容（如图 13.1 所示）。

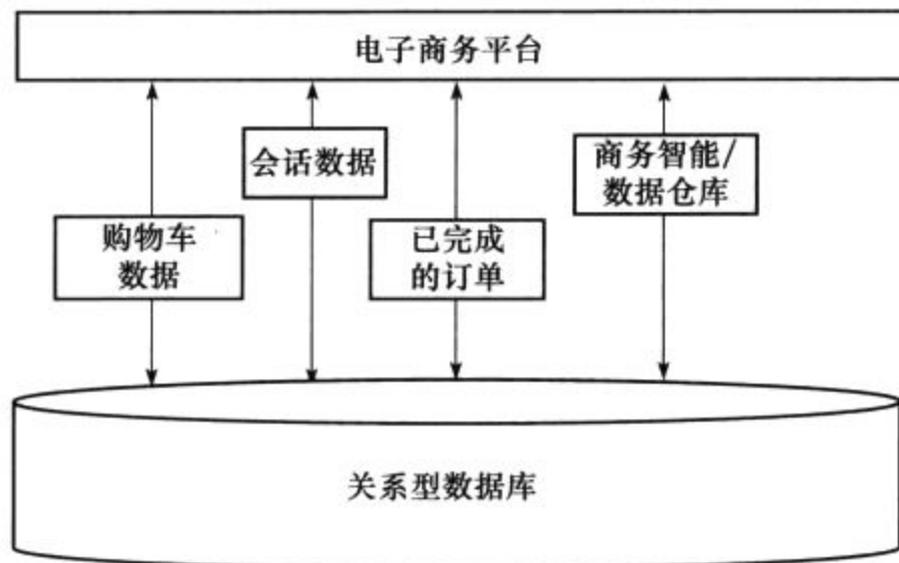


图 13.1 用关系型数据库存储应用程序中的各种数据

会话、购物车、订单数据，其所需的“可用性”、“持久性”、备份策略不一定相同。存放会话管理数据所用的备份/恢复策略（backup/recovery strategy），需要和电子商务订单数据所用的一样健壮吗？会话管理数据库在读写会话数据时，是否需要让数据库引擎提供一个“可用性”更高的实例呢？

Neal Ford 先生在 2006 年造了“多语言编程”（polyglot programming）一词。它要表达的意思是：应该以多种语言混合编写同一应用程序，以这些语言各自的优势来解决其中不同的问题。复杂的应用程序通常要面对各类问题，针对每个问题，选用适当的语言来解决，要比用一门编程语言包揽全部问题更高效。

同理，解决电子商务问题时，一定要以“高可用性”且能扩展的数据库保存购物车数据，但是若想找出顾客的朋友所购买的产品，这种数据库却不擅长，因为那是个完全不同的问题。我们把这种解决持久化问题的混合方式（hybrid approach）定义为“混合持久化”（polyglot persistence）。

## 13.2 混用各类数据库

还是以电子商务为例，我们来演示怎样以“混合持久化”的方式运用各类数据库

(如图 13.2 所示)。在客户确认订单前, 可用键值数据库保存购物车信息, 还可以用它存储会话数据, 这样就无需以关系型数据库来保存这种“瞬态数据”(transient data)了。此处使用键值数据库较合理, 因为通常要以用户 ID 查询购物车, 而且一旦客户确认订单并支付, 就可以将之保存到关系型数据库了。同理, 由于会话数据也要按会话 ID 来查询, 所以也适合用键值数据库保存。

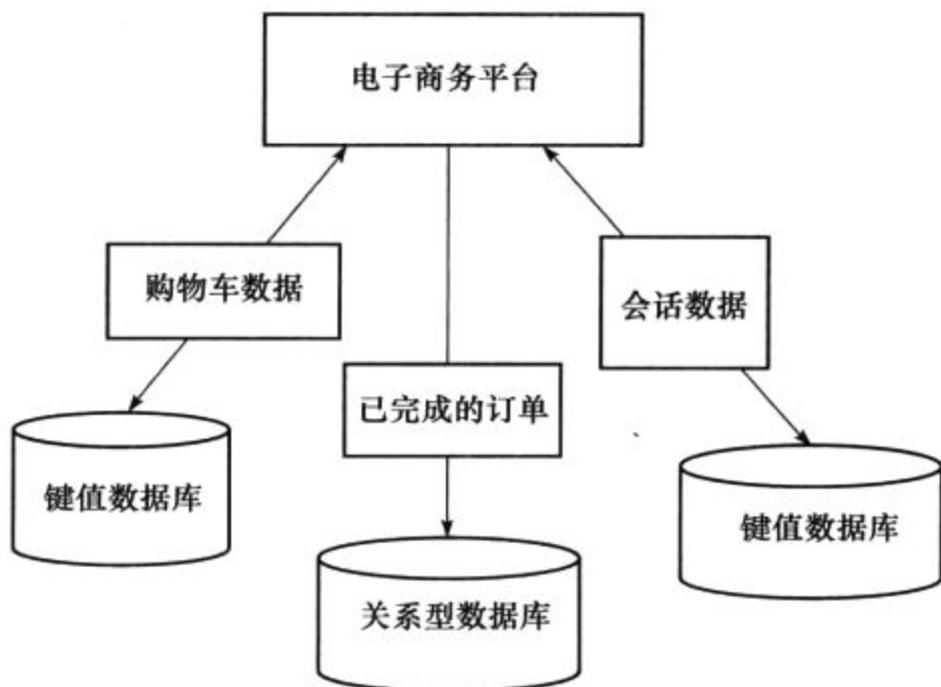


图 13.2 用键值数据库分流会话数据及购物车数据

如果需要在客户向购物车中放入商品时推荐其他产品, 比方说“您的朋友还购买了这些产品”或“您的朋友还购买了此产品的这些配件”, 那么图数据库就能派上用场了(如图 13.3 所示)。

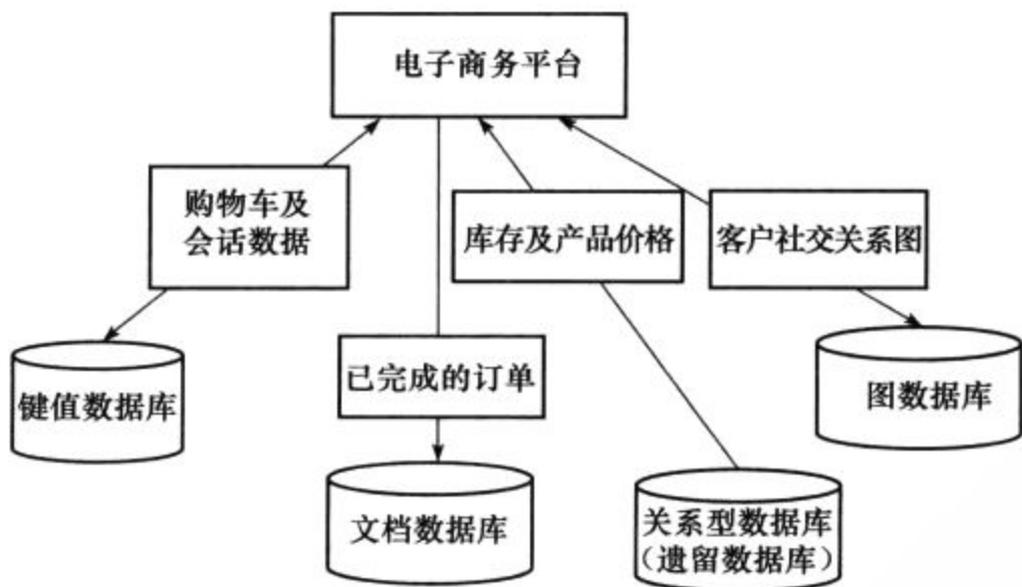


图 13.3 混合持久化实现范例

应用程序没有必要非得用一种数据库应对所有需求, 因为不同数据库的设计目标不同, 并非所有问题都能用一种数据库优雅地解决。

即使用专属关系型数据库来解决不同问题，比如同一应用程序分别使用两套特殊配置的关系型数据库来处理数据仓库和数据分析，也还是可以视为混合持久化的一种形式。

### 13.3 将直接数据库操作封装为服务

应用程序使用多种数据库之后，企业级环境下的其他应用程序就可能得益于这些数据库或其数据了。在电子商务这一示例中，图数据库可以服务于其他应用程序，比方说，那些程序想知道客户群中的某部分人购买了何种产品。

与其让每个应用程序各自同图数据库通信，不如把图数据库封装为服务，这样就可以把节点间的全部关系都存于一处，并供所有应用程序查询（如图 13.4 所示）。以服务形式提供数据所有权管理及 API，要比让应用程序直接和多个数据库通信更有用。

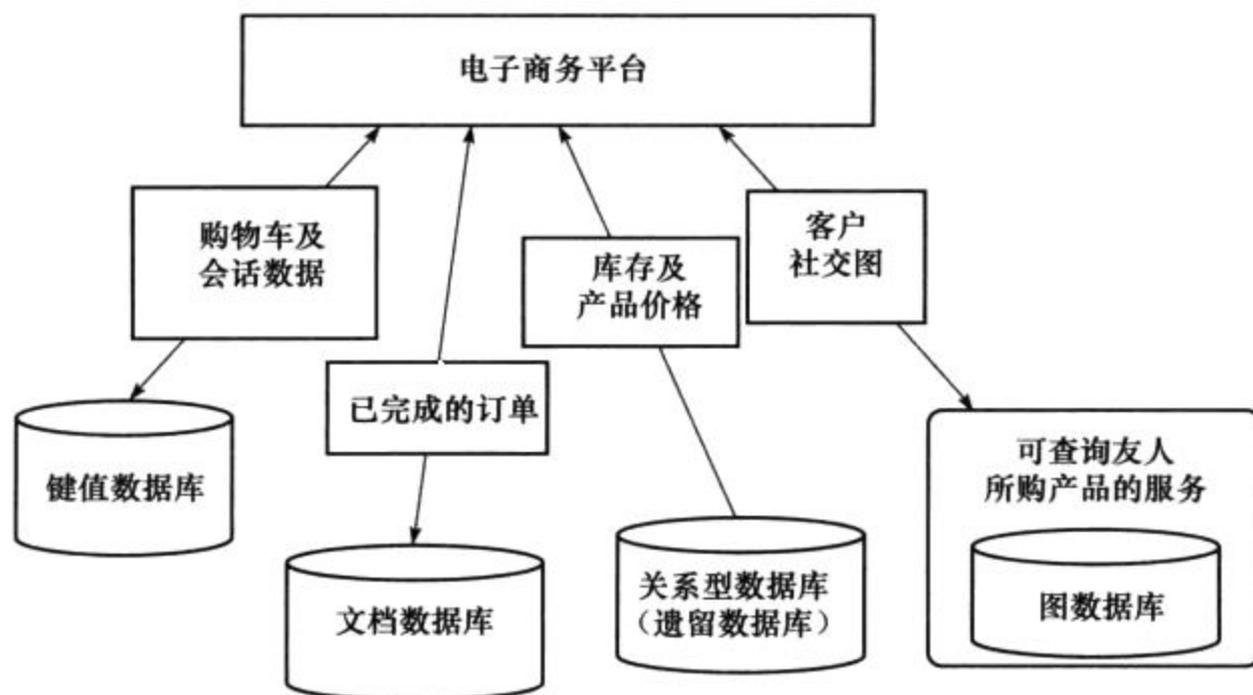


图 13.4 将数据库封装为服务的实现范例

可以进一步运用此思路，把全部数据库都封装成服务，让应用程序只和一系列服务通信（如图 13.5 所示）。这样的话，无需修改依赖其数据的应用程序，即可在服务内部完善数据库。

实际上，Riak[Riak] 和 Neo4J[Neo4J] 等诸多 NoSQL 数据库都提供了现成的 REST API<sup>⊖</sup>。

⊖ REST 为“表征状态转移”（Representational State Transfer）的简称，是一种网络开发所使用的软件架构风格。详情参见：<https://zh.wikipedia.org/wiki/REST>。——译者注

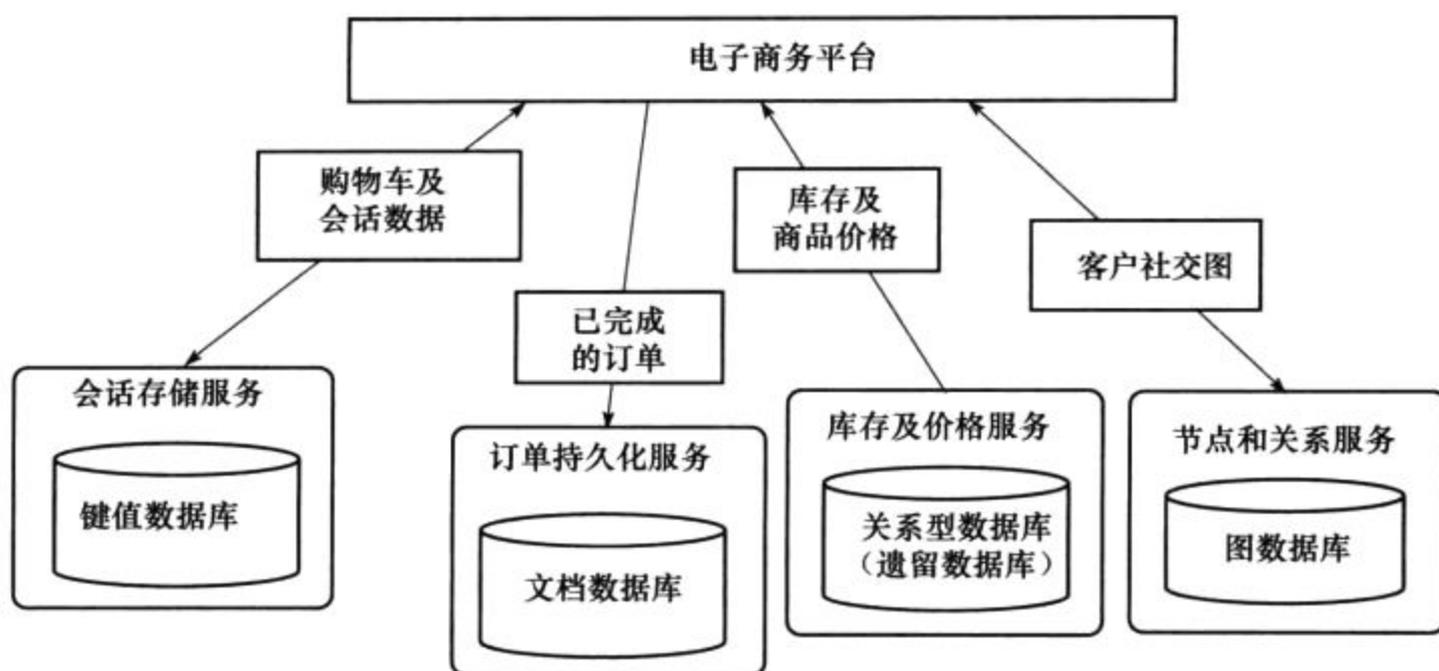


图 13.5 经由服务调用数据库，而非直接与之通信

## 13.4 扩展数据库以增强其功能

我们一般无法改变某个专属数据库的用途，因为有些遗留应用程序仍要使用它们。但是，我们可以增加功能，比如用缓存改善性能，用 Solr[Solr] 等检索引擎（indexing engine）提升搜索效率（如图 13.6 所示）。引入此类技术时，要确保应用程序数据库与缓存或索引引擎之间的数据同步。

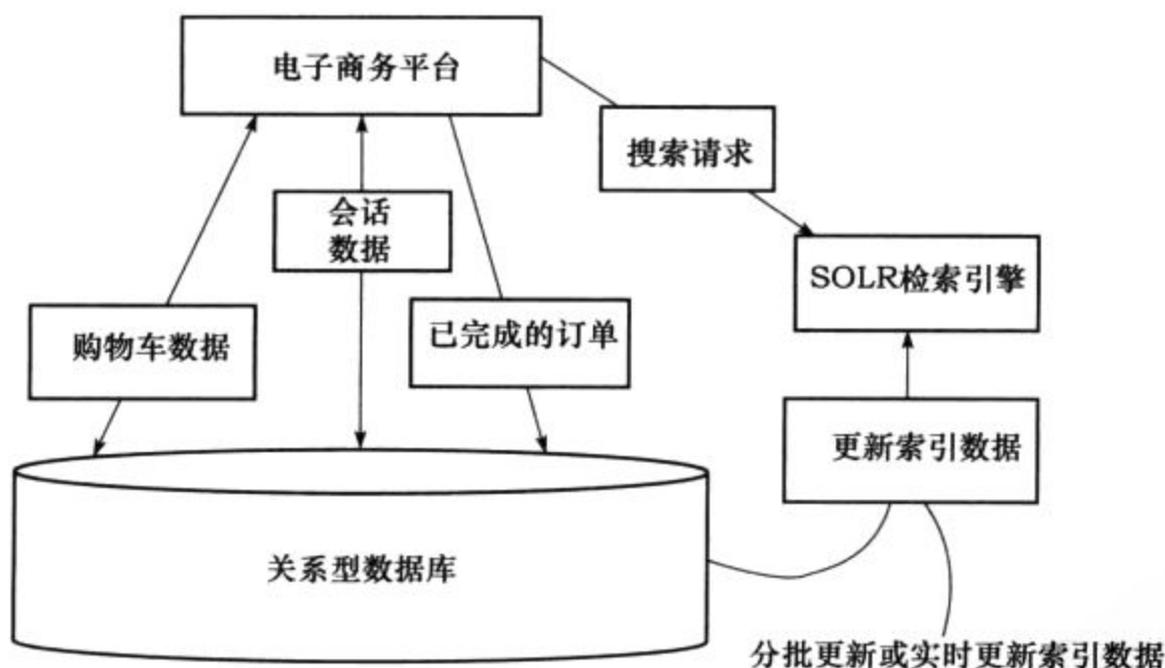


图 13.6 用辅助数据库增强既有数据库的功能

若采用上述做法，则当应用程序数据库中的数据变更时，也要一并更新索引数据。可以实时更新数据，也可以分批更新，只要保证应用程序能处理检索引擎或搜索引擎里的陈旧数据就行。可采用“事件溯源”（event sourcing，参见 14.2 节）模式更新索引。

## 13.5 选用合适的数据库技术

可选的数据存储方案很多。一开始，业界从专属数据库（speciality database）迁移到单一关系型数据库，后者尽管有些抽象，但是能存储各类数据模型。现在的趋势却又重新回到能够实现原生解决方案的那种数据库了。

如果想根据顾客购物车中的产品，向其推荐别人在购买它们时还购买了哪些产品，那么只要能把属性正确的数据持久化，任何一种数据库都能满足需要。选好数据库技术的窍门在于，所选数据库在需求改变之后仍可继续使用，既不损失现有数据，又无需将之换成新格式。

继续来讲实现新功能的问题。采用关系型数据库解决上述需求，需要使用分层查询（hierarchal query）并据此建模数据表。如需修改数据遍历方式，则要重构数据库，迁移数据并持久化新数据。反之，若使用一种能记录节点间关系的数据库，则只需编码新关系即可。此时仅做少许修改，就能继续使用同一份数据库。

## 13.6 企业使用混合持久化技术时的考量

引入 NoSQL 数据库技术，企业的数据库管理员（DBA，Database Administrator 的缩写）就必须思考如何使用这些新数据库了。企业习惯于统一的关系型数据库环境。某企业一开始采用的那种数据库，也许多年后它的全部应用程序依然在用。在混合持久化的新环境下，数据库管理员团队要掌握更多技能（poly-skilled），他们要学会某些 NoSQL 数据库技术的原理，要会检测这些数据库系统，要知道如何备份数据，如何向系统输入数据并从中输出数据。

一旦企业决定采用 NoSQL 技术，那么就要面对使用许可、支持、工具、升级、驱动、审计、安全等问题。许多 NoSQL 技术都是开源的，而且有活跃的支持社区，此外也有公司提供商业支持。这个领域的工具现在还不丰富，然而工具厂商和开源社区正在加速制作。他们发行的工具有 MongoDB Monitoring Service[Monitoring]、Datastax Ops Center[OpsCenter] 和 Riak 专用的 Rekon 浏览器 [Rekon] 等等。

企业还要关心一个问题，那就是数据安全，也就是创建用户并赋予其权限的能力。权限决定了用户是否能在数据库级别访问数据。大多数 NoSQL 数据库的安全性都不甚健壮，但这是由于其操作方式不同于关系型数据库而致。在传统的关系型数据库中，数据库负责提供数据，而使用任何查询工具都能搜索数据库，所以必须加强安全。

NoSQL 数据库也有查询工具，不过其理念是：这些数据为应用程序所有，数据库只是以“服务”的形式来提供它们。在这种使用方式下，由应用程序负责数据安全。虽说如此，但有些 NoSQL 数据库仍提供了安保功能。

企业的数据仓库系统、商务智能、分析系统所用数据，通常来自多种数据源。ETL<sup>Ⓔ</sup> 工具或其他从源系统向数据仓库搬移数据的机制，必须要能读取 NoSQL 数据库中的数据才行。ETL 工具商也在发布能与 NoSQL 数据库通信的工具，例如 Pentaho[Pentaho] 就支持 MongoDB 及 Cassandra。

每个企业都要执行某种分析，它所捕捉的数据量很大，而且越来越多，所以它们极力扩展其关系型数据库系统，以便将所有数据写入其中。如果要频繁执行写入操作并想扩展数据库的写入能力，那么正应该使用能够写入大量数据的 NoSQL 数据库。

## 13.7 部署复杂度

一旦决意在应用程序中使用混合持久化技术，那么就要谨慎考虑部署复杂度（deployment complexity）了。投入生产状态后的应用程序要能同时访问所有数据库。在用户验收测试（User Acceptance Testing，简称 UAT）、质量保证（Quality Assurance，简称 QA）及开发（Development，简称 Dev）环境中都要用到这些数据库。大部分 NoSQL 产品都开源，所以在选择时不太需要考虑授权费问题。它们也支持自动安装与配置。比如，想安装某个数据库，只需下载其压缩包并将之解压即可，而此过程可由 curl 及 unzip 命令自动完成。这些产品的默认参数都较为合理，稍加配置即可使用。

## 13.8 要点

- 混合持久化旨在使用不同数据库技术处理多种数据存储需求。
- 混合持久化既可为企业中多个程序所用，也可运用在单个应用程序中。
- 将数据访问封装为服务，可减少数据库变动对系统其他部分的影响。
- 新增数据库技术会让编程及操作更复杂，所以要权衡候选数据库带来的好处是否抵得过其引入的复杂度。

---

<sup>Ⓔ</sup> Extract-Transform-Load 的缩写，在数据仓库领域，是指将数据从来源传输至目标的过程中所经历的提取（extract）、转换（transform）和加载（load）操作。详情参见：<https://zh.wikipedia.org/wiki/ETL>。——译者注

## 第 14 章

# 超越 NoSQL

新登场的 NoSQL 数据库轰动了数据库领域，也拓展了其空间。然而笔者认为，本书讨论的这类 NoSQL 数据库只是混合持久化的一部分，我们值得花些时间来讲讲那些不能简单归结为 NoSQL 技术的解决方案。

## 14.1 文件系统

数据库很常见，而文件系统则无处不在。过去几十年里，它广泛运用于“个人生产力文档”<sup>⊖</sup>（personal productivity document）领域而非企业级应用程序中。文件系统并不明确定义其内部结构，它们更像一个采用“分层键”（hierarchic key）的键值数据库。除了简单的文件锁定之外，它也很少提供其他并发控制手段，这种方式和只能锁定单一聚合的 NoSQL 数据库相似。

文件系统的好处是简单、适用面广。它们擅长保存视频及音频等庞大实体。通常数据库为以文件形式存储的媒体资源编订索引。“流”（streaming）等按序存取（sequential access）操作也适合以文件为载体，因为那种只需追加的（append-only）数据用它来表示比较方便。

分布式文件系统最近在集群环境领域受到关注。Google File System<sup>⊗</sup>与 Hadoop[Hadoop] 等技术提供了文件副本（replication of file）制作功能。很多“映射-化简”问题涉及在集群系统中操作大文件，并用工具将其自动切分为数段，放在多个节点中处理。事实上经常能见到一些企业因为用了 Hadoop 技术而步入 NoSQL 领域。

最好是用文件系统存放数量相对较少且需要按大块来处理（process in big chunk）的大文件，以流形式访问则更佳。用它存放大量的小文件则性能不好，那是数据库适合做的事。文件系统也未对 Solr[Solr] 这样的附加检索工具提供技术支援。

## 14.2 事件溯源

事件溯源是一种持久化手段，它专注于将某个持续状态（persistent state）所发生的全部变更都持久化，而非仅仅持久化当前应用程序自身的状态。这是一种架构模式（architectural pattern），它与包括关系型数据库在内的大部分持久化技术协同运作得很好。某些不太寻常的持久化思路也以之为基础，所以本书要讲一下它。

现以记录轮船位置日志的系统为例（如图 14.1 所示）。用来保存轮船状态的记录很简单，只包含轮船名称和其当前位置。依惯常思路，当得知 King Roy 号轮船抵达旧金山时，就应把 King Roy 记录的 location（地点）字段改为 San Francisco（旧金山）。稍

⊖ 此词指个人日常工作中用到的文字文件、电子表、幻灯片等工作文档。——译者注

⊗ 缩写为 GFS 或 GoogleFS，是 Google 公司为了满足其需求而开发的基于 Linux 的专有分布式文件系统。详情参见：<https://zh.wikipedia.org/wiki/Google文件系统>。——译者注

后若其已离岸，则将该字段改为 at sea（海上），如果它到香港了，那么还要再次修改这一字段。

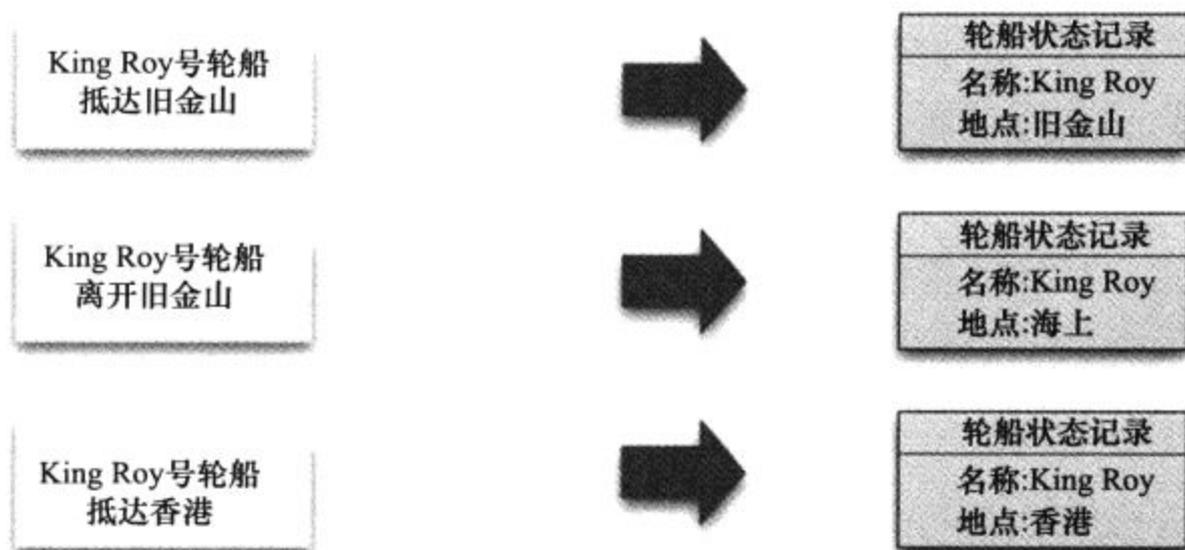


图 14.1 常见系统中，应用程序一接到变更通知，就更新其状态

要使用事件溯源系统（event-sourced system），首先需构建捕获变更信息的事件对象（如图 14.2 所示），然后把该事件对象存储在可持久化的事件日志中。最后，依次处理事件并更新应用程序状态。

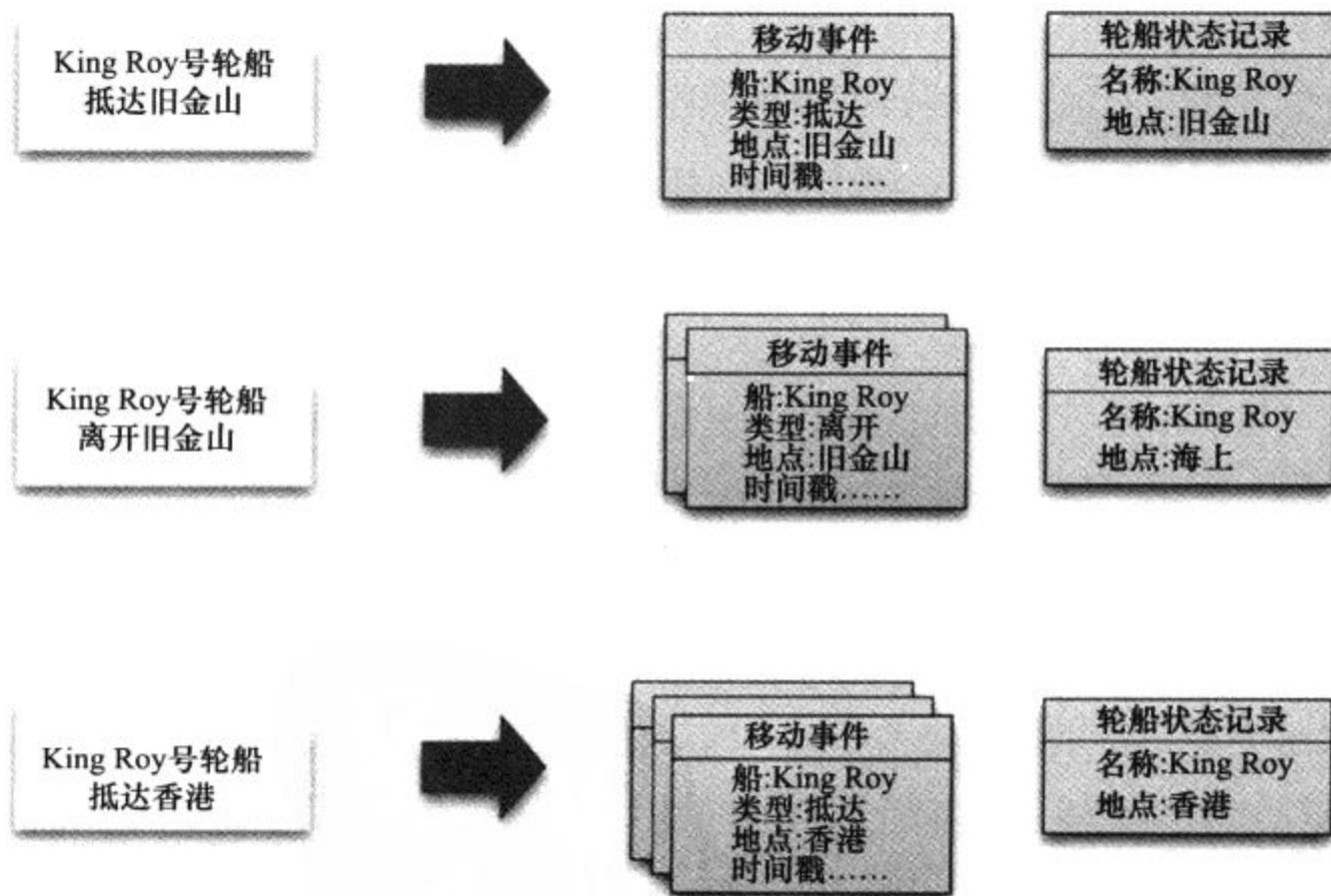


图 14.2 事件溯源系统保存应用程序的每个事件及由此衍生的状态

如此一来，事件系统就可将每个引发系统状态变更的事件保存在事件日志中，而应用程序状态则完全可由该事件日志推算出。可随时丢弃应用程序状态，无需担心，

稍后即能由事件日志重建。

理论上说，仅用事件日志即可重新创建应用程序状态，因为不管何时需要重建，只要回放事件日志就好。然而实际上这样做可能太慢了。所以说，一般情况下，最好能提供快照功能，以便存储并重建应用程序状态。快照（snapshot）用来持久化内存映像（memory image），这种映像是为了迅速恢复应用程序状态而优化过的。它只是一种辅助优化手段，我们决不应把其中的数据看得比事件日志本身更权威。

抓取快照的频率取决于所需的正常运行时间<sup>Ⓔ</sup>。快照中的数据并不一定要最新，因为可以载入最近一份快照并回放抓取快照之后所发生的所有事件。举例来说，我们可以每天晚上抓取一份快照。若某天系统故障了，则可重新加载昨晚的快照并回放从那时起到今日此刻所发生的全部事件。如果这一操作执行起来很快，那就没问题。

要获取一份应用程序历次状态变更的完整记录，需要在事件日志中保存从程序起初到现在的每一个事件。然而很多时候并不需要如此长久的记录，因为只要把旧的事件保存到快照里，并在事件日志中记录发生于快照抓取日之后那些事件即可。

事件溯源模式有诸多好处。可将事件广播至多个系统，而每个系统都根据各自的目标构建不同的应用程序状态（如图 14.3 所示）。对于读取操作很频繁的系统来说，可以提供多个数据模式不同的节点以供读取，同时用另外一个处理系统专门应对写入请求（这就是广为人知的 CQRS 方式 [CQRS]）。

由于可以将事件日志中的既往状态复制到别处，所以事件溯源也是分析历史信息的有效平台。此外，要想研究各种备选方案的效果也很容易，在分析处理器（analysis processor）中引入假想事件（hypothetical event）就好。

事件溯源模式确实让事情变得有些繁琐，尤其是必须保证能以事件形式捕捉并存储所有状态变更。使用某些架构和工具时可能不便这样做。只要同外部系统协作，就得考量事件溯源的影响：在回放事件以重建应用程序状态时，要留意它对外部系统产生的副作用。

---

Ⓔ 由于修复系统时需要恢复快照并回放从快照点到当前的全部事件，所以若系统对正常运行时间要求较高，或者说想尽量缩短下线维护时间，则应频繁抓取快照，反之则可降低抓取频率。——译者注

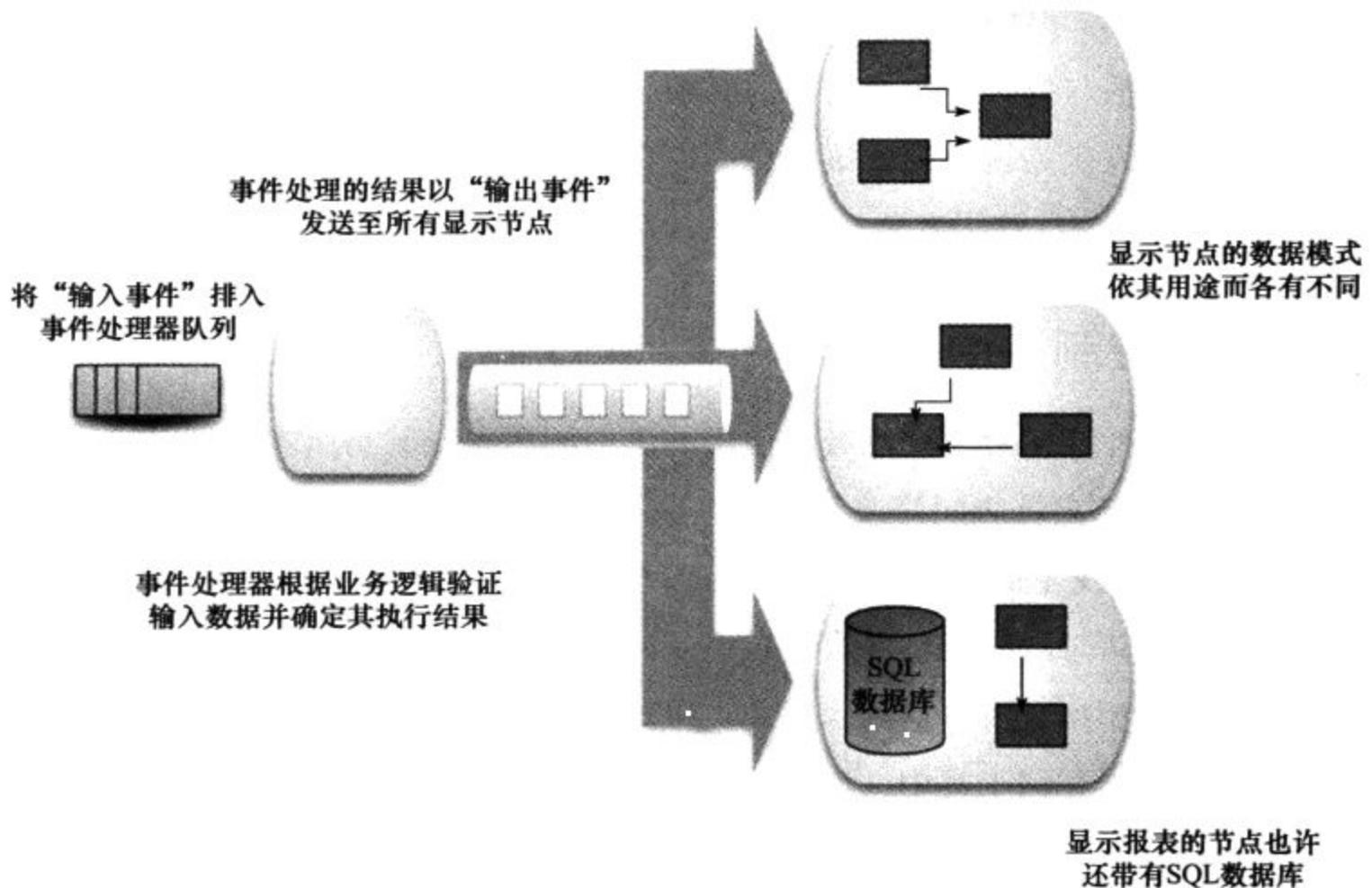


图 14.3 事件可广播至多个显示系统

### 14.3 内存映像

使用事件溯源模式会导致事件日志最后变成持久化记录，不过应用程序状态未必非得持久化才行。我们也可以把应用程序状态放在内存中，并且只使用内存中的数据结构。因为处理事件时不再有磁盘输入/输出操作（disk I/O），所以将全部工作数据都放在内存里可以提高性能。这也简化了编码工作，因为不再需要映射磁盘与内存中的数据结构了。

内存映像的局限很明显，那就是内存必须放得下所需访问的全部数据才行。不过这越来越不成问题了，因为现在的内存容量比原来的硬盘都大。还要注意一点：在系统崩溃后，不管是从事件日志中重新载入事件，还是切换到另一个据此复制的系统上，都要保证能迅速恢复才行。

可能要使用某种显性的并发处理机制。一种办法是采用诸如 Clojure 语言<sup>①</sup>所提供的那种“事务内存系统”（transactional memory system），还有一种办法是用单线程处理全部

① 一套动态 Lisp 方言，也是一种函数式多用途语言。详情参见：<https://zh.wikipedia.org/wiki/Clojure>。——译者注

输入。若精心设计，单线程事件处理器完全能以较低延迟处理大量数据 [Fowler lmax]。

打破内存数据与持久化数据之间的界限，也会影响错误处理。常用的方法是先更新模型，若其间发生错误，则回滚。然而内存映像通常缺乏自动回滚机制，此时要么自己写一套回滚机制（这很复杂），要么在改动数据之前彻底验证该操作的合法性。

## 14.4 版本控制

对大多数软件开发者来说，最常使用的事件溯源系统就是版本控制系统了。版本控制功能可让团队成员协同修改复杂的互联系统，他们可以浏览该系统的既往状态，也可以看到各个分支上的替代实现（alternative reality）<sup>Ⓐ</sup>。

思考数据存储时，我们总喜欢站在单一时间点上看问题，这与能够管理复杂数据的版本控制系统相比，极其局限。因此，很难想象大家为何不把版本控制系统中的某些想法借用到数据存储工具中。毕竟，很多情况下需要查询数据库中的既往信息并从多种视角解读它。

版本控制系统基于文件系统，因此，用文件系统保存数据时所受的限制，它也一样有。它并不是为保存应用程序数据而设计的，所以在那种情境下其效果很糟。然而，在需要使用时间线（timeline）的场合，还是值得一试的。

## 14.5 XML 数据库

在新千年来临之际，大家似乎想用 XML 保存一切内容，所以专门查询并保存 XML 文档的数据库颇受青睐。虽说它们没有如其宣称的那样极度动摇关系型数据库的统治地位，但是 XML 数据库依然存在。

笔者认为 XML 数据库也算文档数据库，它以一种相容于 XML 的数据模型存储其数据，并且可以用各种 XML 技术来操作其文档。DTD、XML Schema 和 RelaxNG<sup>Ⓑ</sup>等多种形式的 XML 模式定义（schema definition）都可用于检查文档格式，XPath 与

---

Ⓐ 此词源自科幻术语“平行宇宙”（Parallel universe），中译“替代现实”，译文从技术角度，将其改称“替代实现”。——译者注

Ⓑ 这 3 种模式定义的详情，可分别参见：<https://zh.wikipedia.org/wiki/文件类型描述>、[https://zh.wikipedia.org/wiki/XML\\_Schema](https://zh.wikipedia.org/wiki/XML_Schema)、[https://en.wikipedia.org/wiki/RELAX\\_NG](https://en.wikipedia.org/wiki/RELAX_NG)。其中“模式定义”一词也称“纲要定义”。——译者注

XQuery<sup>①</sup> 等技术可用于查询其内容，而 XSLT<sup>②</sup> 则可将之转化成其他文档。

关系型数据库也可以存放 XML 数据，它在处理关系型数据时，也能一并处理这些 XML 数据，其通常做法是，将 XML 文档也算成一种列类型，并允许以某种方式混用 SQL 与 XML 查询语言。

当然了，在键值数据库中把 XML 用作数据组织机制（structuring mechanism）也并无不可。XML 在眼下已不如 JSON 时髦了，然而它也同样能存放复杂的聚合，而且其模式与查询能力一般都比 JSON 强。若选用 XML 数据库，则其本身即可发挥 XML 的结构优势，它不会把值仅仅视为一堆二进制数据，但是，这一优势要和数据库其他特性综合起来权衡。

## 14.6 对象数据库

面向对象编程流行起来之后，大家对面向对象的数据库也极感兴趣。这里要说的意思是，它想解决将内存中的数据结构映射为关系表这一复杂问题。面向对象数据库的思路是不让开发者执行这一繁杂任务，而是由数据库自行将内存中的数据结构保存到磁盘上。可将其视为持久化的虚拟内存系统（virtual memory system），开发者根本不用考虑数据库，只需针对持久化编程即可。

对象数据库还未成气候。一个原因是，同应用程序紧密结合，也就意味着除此以外的其他应用程序无法轻松获取其数据。由于很多人正在从集成数据库迁移至应用程序数据库，所以对象数据库的前景很可能比现在更为乐观。

使用对象数据库时要考虑的一个重要问题，是当数据结构改变后如何迁移。持久化存储与内存中数据结构之间的紧密联系，此处成了障碍。某些对象数据库可在对象中定义迁移函数（migration function）。

## 14.7 要点

- NoSQL 只是数据存储技术的一部分。一方面，它们在混合持久化这个领域中渐入佳境；另一方面，我们仍要考虑其他数据存储技术，不论其是否冠以 NoSQL 之名。

① 两者详情可参阅：<https://zh.wikipedia.org/wiki/XPath>、<https://en.wikipedia.org/wiki/XQuery>。——译者注

② 可扩展样式表转换语言（Extensible Stylesheet Language Transformation），是一种转化 XML 文档的语言，详情参见：<https://zh.wikipedia.org/wiki/XSLT>。——译者注

## 第 15 章

# 选择合适的数据库

到此为止，我们已经学习了很多在混合持久化这一新兴领域内做决策时的注意事项。现在该谈谈怎样为以后的开发工作选取合适的数据库了。笔者当然无法确知各位读者的具体情况，所以此处无法给出明确答案，而且，也不能将之归结为几条简单的守则。此外，NoSQL 系统才刚刚投入实际工作中，笔者对其所知也未必成熟，可能再过几年我们的观点就会与现在大不相同。

宽泛地说，选用 NoSQL 数据库有两个原因：提高程序员的工作效率，改善数据访问性能。不同场景下，这些因素可能互补，也可能冲突。两者都很难在项目早期评估，这就比较棘手了：因为很难将数据存储模型的决策过程抽象出来，所以稍后再要修改，也颇为不易。

## 15.1 程序员的工作效率

随便问一个企业级应用程序开发者，你就会发现他对关系型数据库很头疼。收集和显示信息时，通常以聚合为单位，但为了把它持久化，却必须将之变换为关系形式。这件事现在做起来比以前简单多了。20 世纪 90 年代，许多项目还在极力构建“对象 - 关系映射层”（object-relational mapping layer，简称 ORM 层），然而到了 21 世纪，已经出现了 Hibernate、iBATIS 和 Rails Active Record<sup>Ⓐ</sup> 等流行的 ORM 框架，它们大大简化了数据映射工作。不过这并未彻底解决问题。ORM 是一种不严谨的抽象（leaky abstraction）<sup>Ⓑ</sup> 总有一些需要更加留意的情况，尤其是想获得较为理想的性能时。

在这种情况下，面向聚合的数据库就很诱人了。我们可以移除 ORM，以聚合这种相当自然的方式来持久化数据。笔者听说，很多项目迁移到面向聚合的解决方案后，都颇为受用。

图数据库提供了另一种简化方案。关系型数据库不善于处理关系较多的数据。而图数据库既提供了更自然的 API 以存储这种数据，又能够查询此类数据结构。

每种 NoSQL 系统都适于存放“格式不一致的数据”（nonuniform data）。如果使用某种强模式数据库时总要添加临时字段（ad-hoc field）以应付数据模式问题，那么切换到无模式的 NoSQL 数据库会好很多。

NoSQL 数据库的编程模型之所以能提高开发团队的工作效率，有几个重要原因。评估是否使用 NoSQL 的第一步，就是要看软件当前的需求。逐一审视现有功能，观察其是否与所用数据模型相符。在此过程中，可能会发现某个更合适的数据模型。选用匹配较佳的模型，可简化编程工作。

考量数据模型时要记住，混合持久化是一种多数据库解决方案（multiple data storage solution）。各部分数据也许要使用不同的数据存储模型。这将导致不同数据由

---

Ⓐ Active Record，一般译为“活动记录”，是一种领域模型模式，特点是模型类对应关系型数据库中的表，而模型类的实例对应表中的记录。详情参见：[https://zh.wikipedia.org/wiki/Active\\_Record](https://zh.wikipedia.org/wiki/Active_Record)。本书作者之一 Martin Fowler 对其定义如下：<http://martinfowler.com/eaCatalog/activeRecord.html>。Rails Active Record 是指用 Ruby on Rails 框架所做的 Ruby 语言实现，详情参见：<https://github.com/rails/rails/tree/master/activerecord>。——译者注

Ⓑ 此词一般译为“脆弱的抽象”或“有漏洞的抽象”，源自 Joel Spolsky 提出的“抽象漏洞定律”：“All non-trivial abstractions, to some degree, are leaky.”（若一抽象概念无法不证自明，则其必有某种程度之疏漏），用来描述本打算减少或隐藏复杂问题，结果却未能完全掩盖底层细节的抽象。详情参见：[https://zh.wikipedia.org/wiki/Leaky\\_abstraction](https://zh.wikipedia.org/wiki/Leaky_abstraction)。——译者注

多个数据库进行管理。多数据库固然比单数据库复杂，然而若能使用合适的数据库分别处理每种数据用例，其总体效果仍好于单数据库。

在审视数据模型匹配度时，尤其要注意会出问题的地方。聚合模型也许能很好地应对大部分功能，但却不适用于另外一小部分功能。不能仅因某模型不适用于少数几个功能就弃用它，由于能够很好地处理大多数功能，所以它体现出来的优势会盖过处理个别问题时的麻烦。然而，还是应该指出并强调这些匹配不佳的情况。

逐个检视软件功能并评判其所需的数据模型，应该能找到好几种备选的数据数据库解决方案。这只是个起点，接下来就要实际构建软件以尝试其效果了。选一些初始功能并开始构建，其间密切留意所选技术是否能较为容易地实现这些功能。此时宜采用好几种不同数据库构建同一功能，以择其佳者。通常我们并不愿这么做，因为没人爱写可能会遭弃用的软件。然而，这确实是判断某框架是否有效的基本途径。

遗憾的是，并没有一种判断各设计方案优劣的适当标准。没办法合理衡量其产出效率。即便构建同一功能，也无法精确比较各种方案的工作效率，因为你原来可能做过一次这项功能，再做第二次时就容易了，而且也无法找到几个完全一致的团队，令其分别以各自方案同时构建这一功能。我们能做的就是倾听开发者的意见。大多数开发者都能感知自己在不同环境下的工作效率孰高孰低。虽说这是主观判断，且团队成员之间或有分歧，但它却是最佳的评判方式了。最后还是应尊重工作团队所做的决定。

在评判使用某数据库的工作效率时，也要尝试一下刚才提到的某些不适用场景。这样团队即可分别感知此技术在其强项与弱项上的表现，以把握总体印象。

这种方法有缺陷。通常不花数月时间使用某技术，就无法全面了解它，而用如此长的时间去评估，多半显得不太划算。但是，与生活中的许多事情一样，必须选出最佳方案，尽管它并不完美，我们还是得接受。关键问题是，必须要尽量根据编程实践来决策。某个技术哪怕只试用一星期，也能学到一些从大量产品介绍中绝对领会不到的知识。

## 15.2 数据访问性能

促使 NoSQL 数据库发展的原因是，需要迅速访问大量数据。大型网站出现后，它们要横向扩张，并运行于集群之中。这些网站研发了早期的 NoSQL 数据库，以助其能在此架构下高效运行。其他数据用户也追随这一潮流，他们同样关注如何迅速访问数

据，而且通常要访问大量数据。

好些因素都决定了 NoSQL 数据库在多种场合下都比默认配置的关系型数据库性能更优。面向聚合的数据库读出或取得聚合的速度要比关系型数据库快很多，因为后者的数据散布在许多表中。因为 NoSQL 数据库便于在集群中“分片”与“复制”，所以可对其进行水平扩展。联系紧密的数据，用图数据库来获取，要快于使用 join 命令的关系型数据库。

如果想从性能角度探究 NoSQL 数据库，那么首要任务就是在你所关心的场景下测试它们。推测数据库的性能，也许能帮你缩小选择范围，然而只有真正构建、测试、估量它，才能正确判断其性能。

构建性能评估模型时，最难的通常是设定一套合乎实际的性能测试。因为是预先评估，所以不能构建实际系统，只能构建一套有代表性的子集。而这个子集，一定要尽量忠实地体现待评估的系统。如果某数据库打算同时服务上百位用户，那么在单用户环境下评估其性能就没有意义了。构建的测试要能体现出真实系统的负载与数据量才行。

构建公共网站时，尤难找到一套高负载测试平台（high-load testbed）。有个好主意，就是利用云计算资源（cloud computing resource）来生成负载并构建测试集群（test cluster）。云配给（cloud provisioning）的弹性非常利于执行短期的性能评估。

不可能把应用程序的每种用法都测一遍，只能构建一个有代表性的功能子集。要选择最常见、最依赖于性能、与当前数据库模型匹配不佳的用例来测试。最后一种用例可能会揭示出数据库在主要用例之外所存在的风险。

测试所需的数据量有些不太好想，尤其是项目早期，我们并不清楚实际应用中的数据量会是什么情况。然而必须提出一个值，作为后续思考的出发点，这一假定必须明示，而且要与参与项目的所有人沟通。明确提出此假设，即可令大家对于“沉重的读取负载”（heavy read load）这一表述的含义不再众说纷纭。而且稍后项目状况一旦偏离了起初假定，也很容易就能发现问题。若不明示此假设，则当稍后发现的新信息揭示出实情已偏离原初预想时，我们很可能就更意识不到自己应该重建测试平台了。

### 15.3 继续沿用默认的关系型数据库

笔者自然认为 NoSQL 数据库在很多情境中都可行，否则也就不会花费数月时间写这本书了。但是，我们也意识到，在许多情况下，实际上在大多数情况下，最好的办法还是继续使用默认选项，也就是关系型数据库。

关系型数据库广为人知，很容易找到有经验的使用者。这些数据库其技术业已成熟，开发者不容易成为新技术的实验品，而且还有大批建立在关系型数据库技术上的工具可供利用。此外，也可远离因采用非常规技术而带来的政策纠葛（political issue），因为选用新技术要冒险，它很可能使项目陷入困境。

所以，总体上讲，笔者的观点倾向于：在用 NoSQL 数据库解决当前问题确实优于关系型数据库时，才选用它。如果从编程效率及性能两方面评估出其没有明显优势，那么继续使用关系型数据库也不成问题。在许多情况下使用 NoSQL 数据库的确较佳，然而“许多”并不意味着“全部”，也不等于“大多数”。

## 15.4 抽离数据库策略以降低风险

就数据库选择问题给出建议时，难点之一就是现有决策信息不足。写作本书时，只有一些先行者在讨论使用 NoSQL 技术的经验，所以笔者并未清楚掌握其实际利弊。

在情况不明朗时，不妨多讨论如何封装数据库决策：将全部数据库代码都放到代码库中的某处，假如稍后要选用其他数据库，那么替换起来相对容易些。实现此封装的典型方式就是在应用程序中以“数据映射器”（Data Mapper）及“储存库”（Repository）[Fowler PoEAA] 等模式明确引入数据储存层（data store layer）。实现此封装层的确要花工夫，在未确定是否使用类似键值数据库与图数据库这种迥然不同的数据模型时，尤其如此。更糟糕的是，我们也许还没有经验将差别如此巨大的几种数据库封装到数据层中。

总的来说，笔者建议将封装视为预设的策略，同时留意隔离层（insulating layer）的实现成本。如果其成为负担，比如说它妨碍到数据库发挥某些实用功能，那么就不如直接使用带有此功能的数据库。这一信息也许能让我们判定：无需再封装了，直接使用某一数据库就好。

还有一种办法，就是把数据库层分解为封装数据库操作的服务（参见 13.3 节）。只要能降低各服务间的耦合，此方案就还能体现出另外一个优势：将来万一出了问题，可以较为容易地替换数据库。即便最后决定只在工作中使用同一种数据库，此方案看上去也仍有其价值，因为假如项目运行得不顺利，我们可以逐个替换每个服务中的数据库，首先解决那些问题最大的服务。

在你决定使用关系型数据库时，上述设计建议仍然适用。可以先将数据库各个部分封

装为服务，等到 NoSQL 技术成熟，其优势也更加明朗之后，再把其中部分数据库替换掉。

## 15.5 要点

- 使用 NoSQL 技术的两个主要原因是：
- 通过使用更符合应用程序需求的数据库来改善程序员的工作效率。
- 以能处理大量数据、降低延迟且增进数据吞吐量的某种技术组合来改善数据访问性能。
- 在决定使用某个 NoSQL 技术前，一定要测试其是否如预期般改进了程序员工作效率及数据访问性能。
- 用服务来封装数据库，即能在需求变更或技术成熟后改换其所封装的数据库技术。可将应用程序各部分划归到不同服务中，以便为既有程序引入 NoSQL 数据库。
- 大部分应用程序，尤其是“非战略性的”（nonstrategic）应用程序，应该继续使用关系型数据库技术，至少在 NoSQL 技术环境尚未更加成熟之前是如此。

## 15.6 结语

笔者希望本书能启发大家的思路。刚开始下笔时，我们也苦于找不到有关 NoSQL 领域的广泛调查资料。为写作此书，笔者亲自调查这一领域，并自得其乐。我们也愿读者能相当迅速而又愉快地领略其中的内容。

现在你也许开始考虑使用 NoSQL 技术了。如果真是这样，本书也只能作为理解 NoSQL 的出发点。赶紧下载几个 NoSQL 数据库然后试用一下吧，笔者深信，只有亲自用过一项技术，才能正确理解它，才能体会其优势，并发现一些文档里绝对读不到但却必须面对的麻烦事。

笔者希望大多数人，当然也包括本书的大部分读者，暂时不要用 NoSQL。这是一项新技术，我们还尚未充分理解何时使用并怎样用好它。不过正如软件业中的其他新事物一样，其变化之快将超出预期，所以要留意 NoSQL 领域的发展。

我们还希望读者能找到更多有益的教材及文章。优秀的 NoSQL 书籍估计要等本书出版之后才会问世，所以此时笔者还没发现其他参考资料。我们二位作者也会在网上积极讨论此技术，欲知我们对 NoSQL 的新想法，请访问 [www.sadalage.com](http://www.sadalage.com) 与 <http://martinfowler.com/nosql.html>。

# 参考资料

- [Agile Methods] [www.agilealliance.org](http://www.agilealliance.org).
- [Amazon's Dynamo] [www.allthingsdistributed.com/2007/10/amazons\\_dynamo.html](http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html).
- [Amazon DynamoDB] <http://aws.amazon.com/dynamodb>.
- [Amazon SimpleDB] <http://aws.amazon.com/simpledb>.
- [Ambler and Sadalage] Ambler, Scott and Pramodkumar Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley. 2006. ISBN 978-0321293534.
- [Berkeley DB] [www.oracle.com/us/products/database/berkeley-db](http://www.oracle.com/us/products/database/berkeley-db).
- [Blueprints] <https://github.com/tinkerpop/blueprints/wiki>.
- [Brewer] Brewer, Eric. *Towards Robust Distributed Systems*. [www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf](http://www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf).
- [Cages] <http://code.google.com/p/cages>.
- [Cassandra] <http://cassandra.apache.org>.
- [Chang etc.] Chang, Fay, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. *Bigtable: A Distributed Storage System for Structured Data*. <http://research.google.com/archive/bigtable-osdi06.pdf>.
- [CouchDB] <http://couchdb.apache.org>.
- [CQL] [www.slideshare.net/jericevans/cql-sql-in-cassandra](http://www.slideshare.net/jericevans/cql-sql-in-cassandra).
- [CQRS] <http://martinfowler.com/bliki/CQRS.html>.
- [C-Store] Stonebraker, Mike, Daniel Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. *C-Store: A Column-oriented DBMS*. <http://db.csail.mit.edu/projects/cstore/vldb.pdf>.
- [Cypher] <http://docs.neo4j.org/chunked/1.6.1/cypher-query-lang.html>.
- [Daigneau] Daigneau, Robert. *Service Design Patterns*. Addison-Wesley. 2012. ISBN 032154420X.
- [DBDeploy] <http://dbdeploy.com>.
- [DBMaintain] [www.dbmaintain.org](http://www.dbmaintain.org).
- [Dean and Ghemawat] Dean, Jeffrey and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. [http://static.usenix.org/event/osdi04/tech/full\\_papers/dean/dean.pdf](http://static.usenix.org/event/osdi04/tech/full_papers/dean/dean.pdf).
- [Dijkstra's] [http://en.wikipedia.org/wiki/Dijkstra%27s\\_algorithm](http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm).
- [Evans] Evans, Eric. *Domain-Driven Design*. Addison-Wesley. 2004. ISBN 0321125215.
- [FlockDB] <https://github.com/twitter/flockdb>.
- [Fowler DSL] Fowler, Martin. *Domain-Specific Languages*. Addison-Wesley. 2010. ISBN 0321712943.
- [Fowler lmax] Fowler, Martin. *The LMAX Architecture*. <http://martinfowler.com/articles/lmax.html>.
- [Fowler PoEAA] Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-

- Wesley. 2003. ISBN 0321127420.
- [Fowler UML] Fowler, Martin. *UML Distilled*. Addison-Wesley. 2003. ISBN 0321193687.
- [Gremlin] <https://github.com/tinkerpop/gremlin/wiki>.
- [Hadoop] <http://wiki.apache.org/hadoop/MapReduce>.<sup>⊖</sup>
- [HamsterDB] <http://hamsterdb.com>.
- [Hbase] <http://hbase.apache.org>.
- [Hector] <https://github.com/rantav/hector>.
- [Hive] <http://hive.apache.org>.
- [Hohpe and Woolf] Hohpe, Gregor and Bobby Woolf. *Enterprise Integration Patterns*. Addison-Wesley. 2003. ISBN 0321200683.
- [HTTP] Fielding, R., J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol—HTTP/1.1*. [www.w3.org/Protocols/rfc2616/rfc2616.html](http://www.w3.org/Protocols/rfc2616/rfc2616.html).
- [Hypertable] <http://hypertable.org>.
- [Infinite Graph] [www.infinitegraph.com](http://www.infinitegraph.com).
- [JSON] <http://json.org>.
- [LevelDB] <http://code.google.com/p/leveldb>.
- [Liquibase] [www.liquibase.org](http://www.liquibase.org).
- [Lucene] <http://lucene.apache.org>.
- [Lynch and Gilbert] Lynch, Nancy and Seth Gilbert. *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. <http://pd.epfl.ch/sgilbert/pubs/BrewersConjecture-SigAct.pdf>.
- [Memcached] <http://memcached.org>.
- [MongoDB] [www.mongodb.org](http://www.mongodb.org).
- [Monitoring] [www.mongodb.org/display/DOCS/MongoDB+Monitoring+Service](http://www.mongodb.org/display/DOCS/MongoDB+Monitoring+Service).
- [MyBatis Migrator] <http://mybatis.org>.
- [Neo4J] <http://neo4j.org>.
- [NoSQL Debrief] <http://blog.oskarsson.nu/post/22996140866/nosql-debrief>.
- [NoSQL Meetup] <http://nosql.eventbrite.com>.
- [Notes Storage Facility] [http://en.wikipedia.org/wiki/IBM\\_Lotus\\_Domino](http://en.wikipedia.org/wiki/IBM_Lotus_Domino).
- [OpsCenter] [www.datastax.com/products/opscenter](http://www.datastax.com/products/opscenter).
- [OrientDB] [www.orientdb.org](http://www.orientdb.org).
- [Oskarsson] *Private Correspondence*.
- [Pentaho] [www.pentaho.com](http://www.pentaho.com).
- [Pig] <http://pig.apache.org>.
- [Pritchett] [www.infoq.com/interviews/dan-pritchett-ebay-architecture](http://www.infoq.com/interviews/dan-pritchett-ebay-architecture).
- [Project Voldemort] <http://project-voldemort.com>.
- [RavenDB] <http://ravendb.net>.
- [Redis] <http://redis.io>.
- [Rekon] <https://github.com/basho/rekon>.
- [Riak] <http://wiki.basho.com/Riak.html>.
- [Solr] <http://lucene.apache.org/solr>.
- [Strozzi NoSQL] [www.strozzi.it/cgi-bin/CSA/tw7/1/en\\_US/NoSQL](http://www.strozzi.it/cgi-bin/CSA/tw7/1/en_US/NoSQL).
- [Tanenbaum and Van Steen] Tanenbaum, Andrew and Maarten Van Steen. *Distributed*

⊖ 原书给出的链接“<http://hadoop.apache.org/mapreduce>”已无法打开，这里将其更正为新链接。——译者注

*Systems*. Prentice-Hall. 2007. ISBN 0132392275.

[Terrastore] <http://code.google.com/p/terrastore>.

[Vogels] Vogels, Werner. *Eventually Consistent—Revisited*. [www.allthingsdistributed.com/2008/12/eventually\\_consistent.html](http://www.allthingsdistributed.com/2008/12/eventually_consistent.html).

[Webber Neo4J Scaling] <http://jim.webber.name/2011/03/22/ef4748c3-6459-40b6-bcfa-818960150e0f.aspx>.

[ZooKeeper] <http://zookeeper.apache.org>.