



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最新力作，权威性毋庸置疑

全方位比较关系型数据库与NoSQL数据库的异同，详细讲解4大主流NoSQL数据库的优劣势、用法和适用场合，深入探讨实现NoSQL数据库系统的各种细节

华章程序员书库

PEARSON

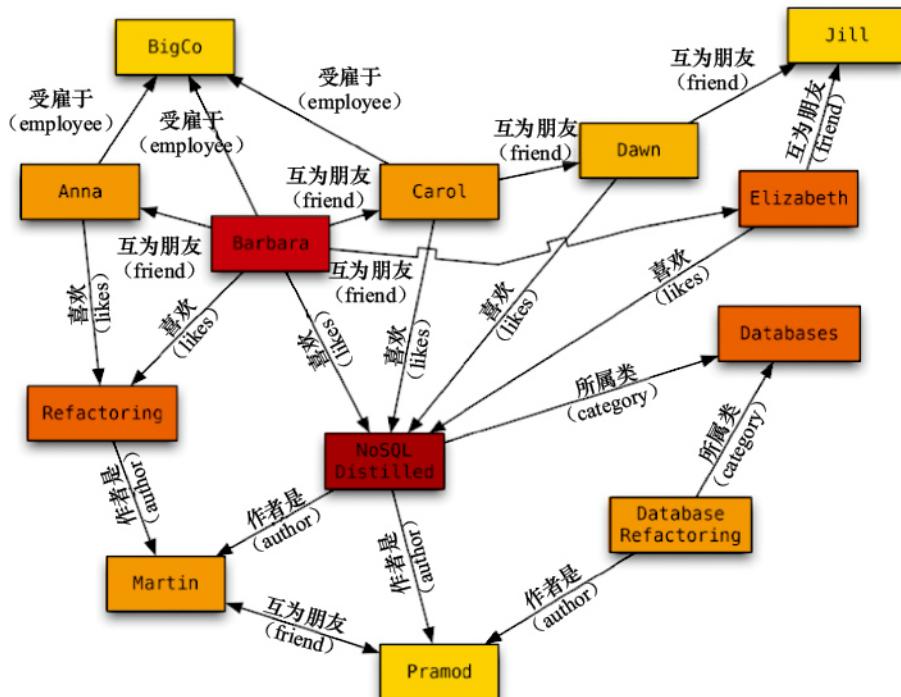
NoSQL Distilled

A Brief Guide to the Emerging World of Polyglot Persistence

NoSQL 精粹

(美) Pramod J. Sadalage Martin Fowler 著

爱飞翔 译



华章程序员书库

NoSQL 精粹

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

(美) Pramod J. Sadalage Martin Fowler 著

爱飞翔 译



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

NoSQL 精粹 / (美) 塞得拉吉 (Sadlage, 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. Sadlage, 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中国香港、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. Sadlage 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

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: hzjsj@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，同时还将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 联系译者，或访问网址 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）^①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

^① 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，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[⊖] 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 Rodriquez、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
第 3 章 数据模型详解	27
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
第 4 章 分布式模型	40
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
第 5 章 一致性	49
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
第 6 章 版本戳	65
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
第 7 章 映射 - 化简	71
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
第 10 章 列族数据库	102
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
第 11 章 图数据库	113
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
第 12 章 模式迁移	124
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
第 13 章 混合持久化	134
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
第 14 章 超越 NoSQL	141
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
第 15 章 选择合适的数据库	148
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
参考资料	154

华章图书

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 数据模型詳解
- 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。

华章图书

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”[⊖]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^②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”^③。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

② 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

③ 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。——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）。

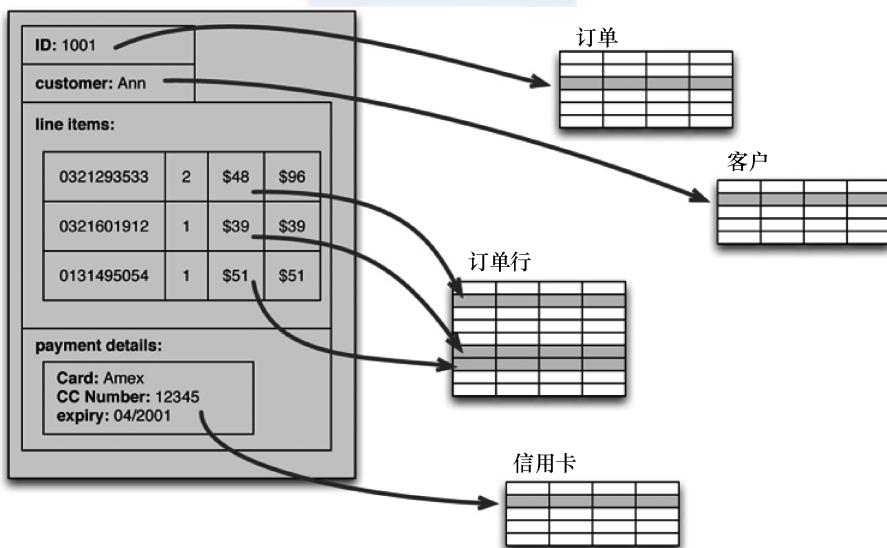


图1.1 这是一张订单。在用户界面中看起来像一个聚合结构，然而其数据却存放在关系型数据库的多张表中。每张表内的行对应具体的数据

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期，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[⊖] 和 iBATIS[⊖]，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

⊖ 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

⊖ 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容易了。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[⊖]，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/%E6%9D%A2%E5%8A%A0%E5%8C%85_\(%E7%AC%94%E8%AF%89\)](http://zh.wikipedia.org/wiki/%E6%9D%A2%E5%8A%A0%E5%8C%85_(%E7%AC%94%E8%AF%89))。——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^①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^②。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

① 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

② 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[⊖]（谷歌）与Dynamo[⊖]（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^①来操作这种数据库，并且能够使用常见的 UNIX 管道^②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^③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”^④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^⑤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。——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。——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 数据库通常是为 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^②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，

^② 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^①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^②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。

① 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

② 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 to 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，我们认为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[⊖]。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。

⊖ 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”^①。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。

^① 参照完整性（referential integrity）也叫引用完整性，是指某一列里的每一个值，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。——译者注

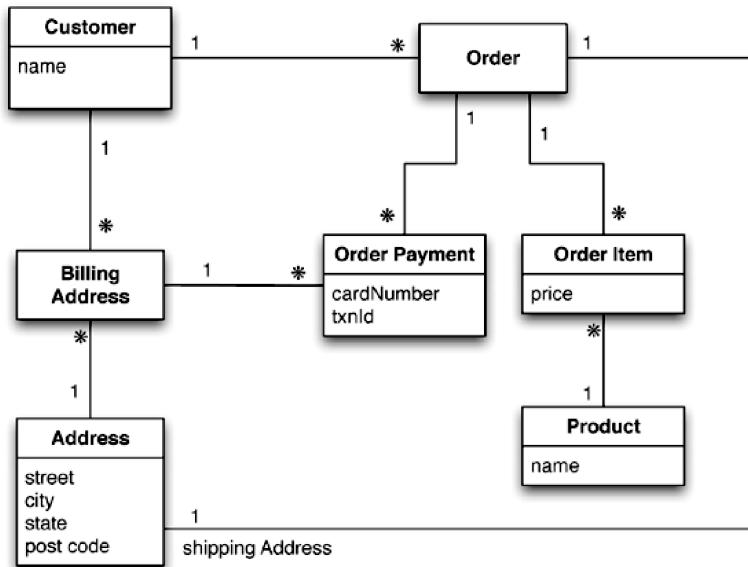


图 2.1 面向关系型数据库的数据模型（使用〔Fowler UML〕一书中的 UML 记法）

Customer	Orders
Id	Name
1	Martin

Product	BillingAddress
Id	Name
27	NoSQL Distilled

OrderItem	Address
Id	CustomerId
100	1
99	ShippingAddressId
27	77
32.45	

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

图 2.2 使用 RDBMS^② 数据模型的范例数据

现在我们再来看看，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）。

② 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

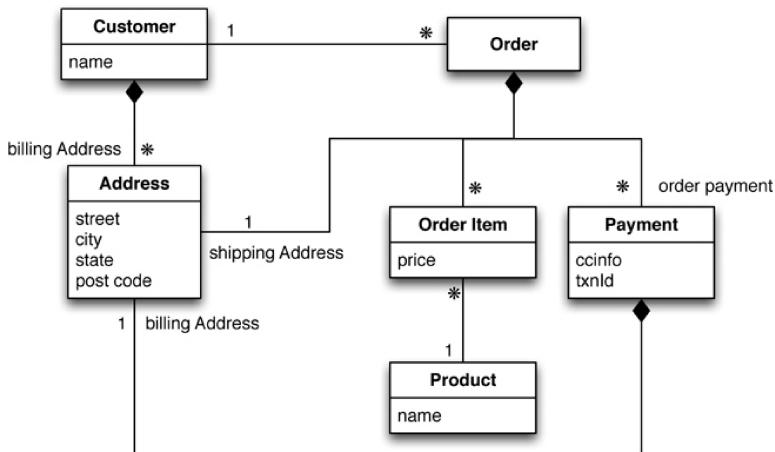


图 2.3 聚合数据模型

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^②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，因为我们希望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）。

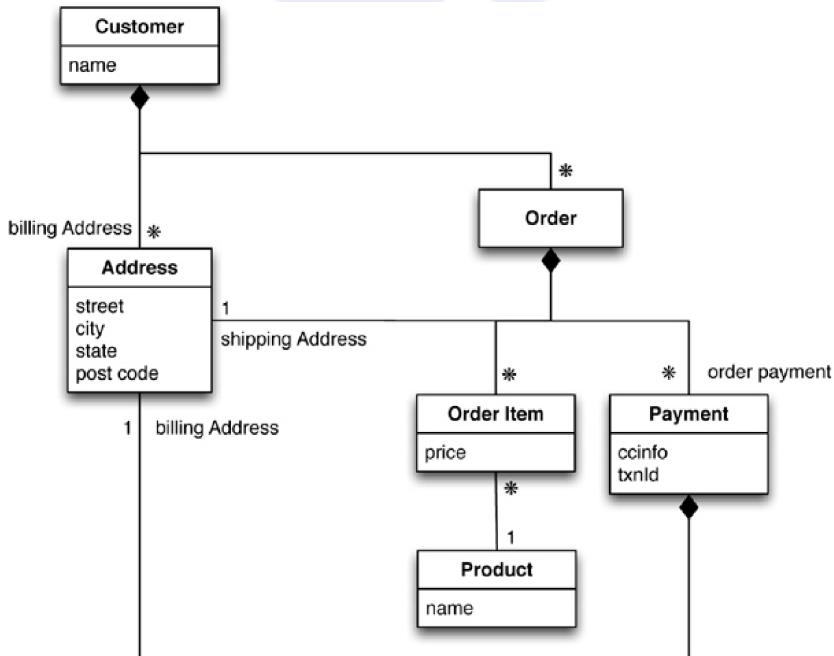


图 2.4 将涉及客户与客户订单的所有对象全部嵌入一个聚合结构中

② 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[⊖]，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[⊖]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。它也许会是键，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的场合更是如此。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”[⊖]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。

[⊖] 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

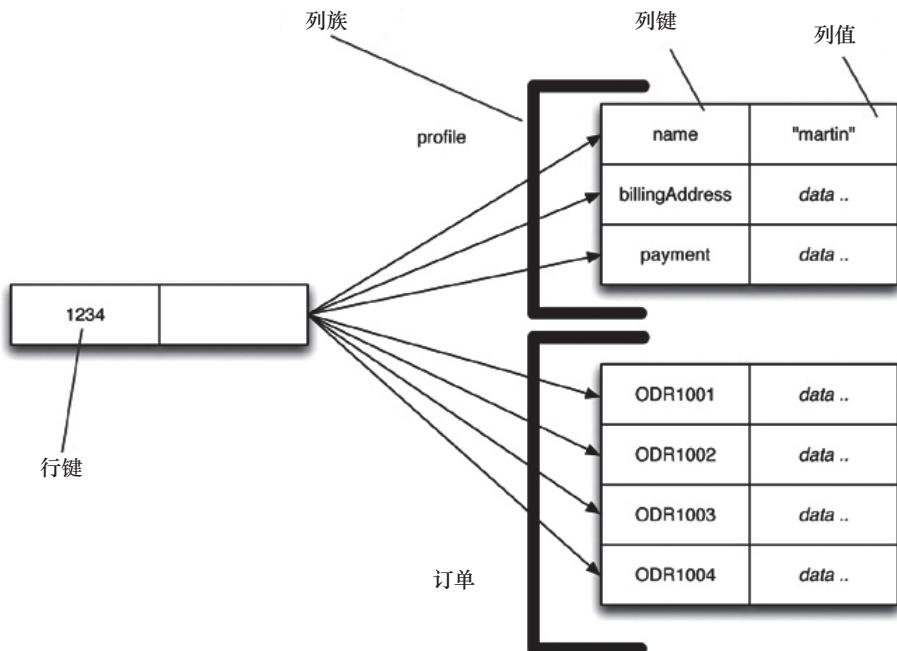


图 2.5 以列族结构表示客户信息

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[⊖]。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

[⊖] 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^②。

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

^② 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^②，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。图数据库的催生产机与之不同，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。

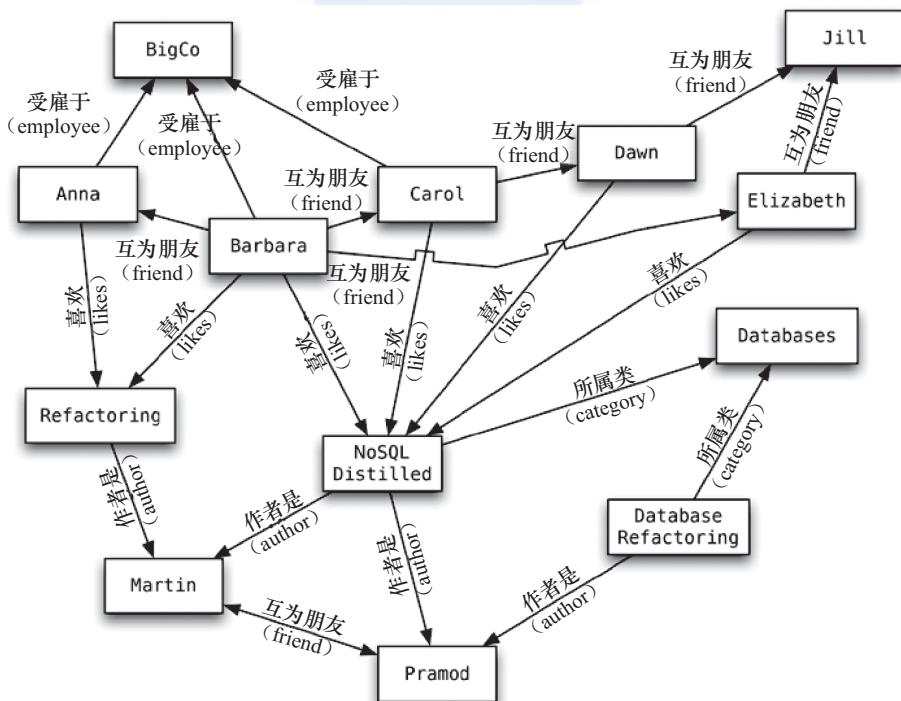


图 3.1 “图结构”示例

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)。

⊕ 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[⊖]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[⊖]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

[⊖] 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

[⊖] 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实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）。

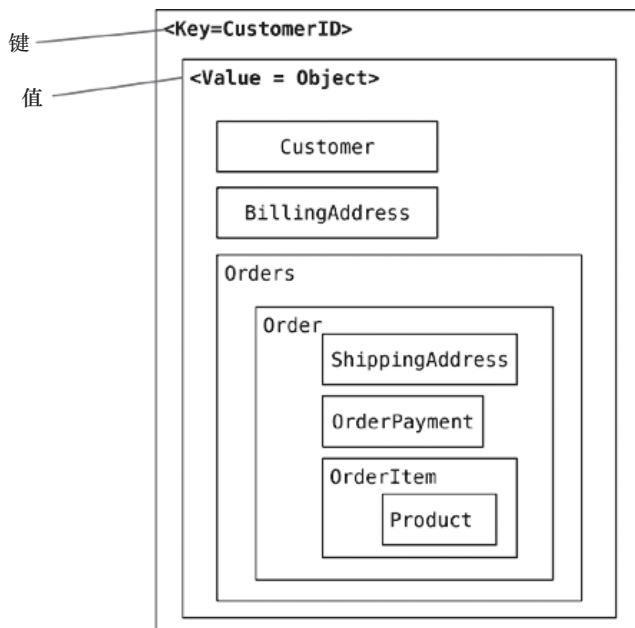


图 3.2 将表示客户及其订单的所有对象都嵌入同一聚合

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"}
  }
}
```

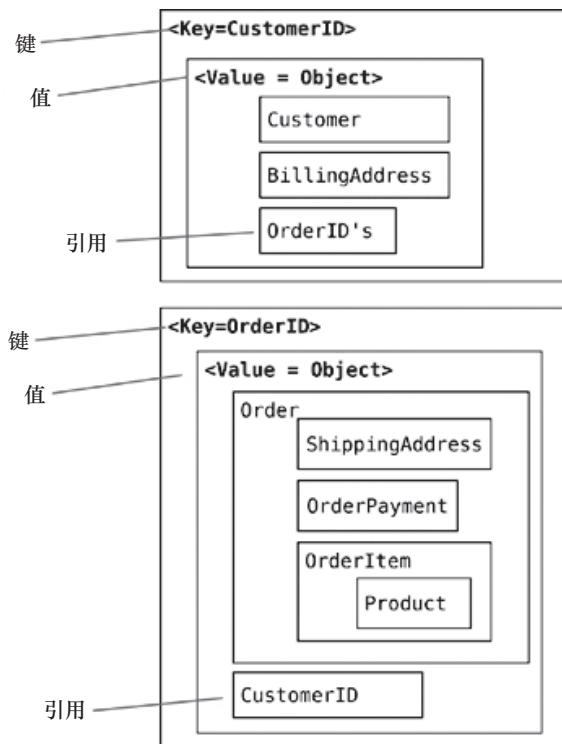


图 3.3 将 Customer 与 Order 对象分开存放

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”^②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，

^② 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。——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进行“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。

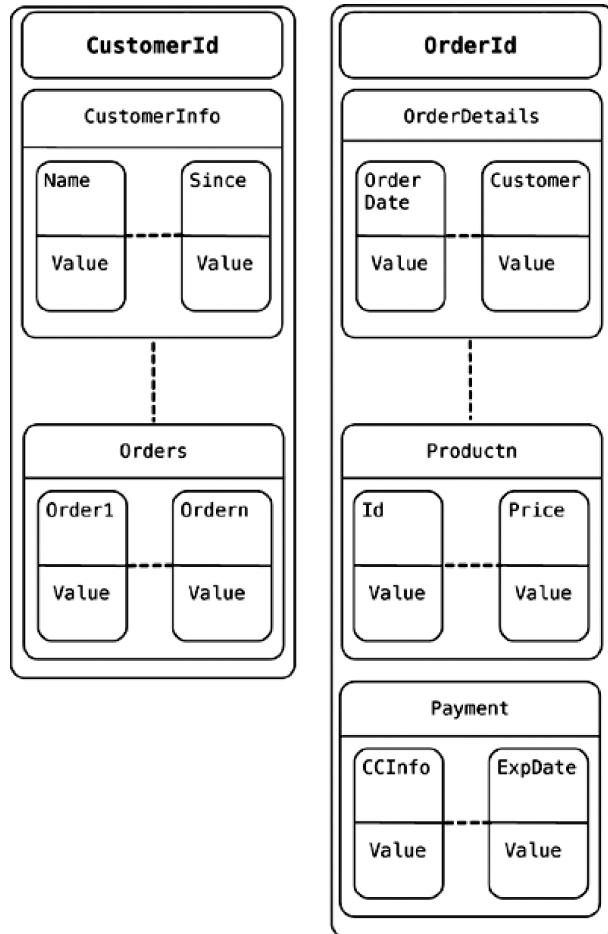


图 3.4 列式数据存储的概念图

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。

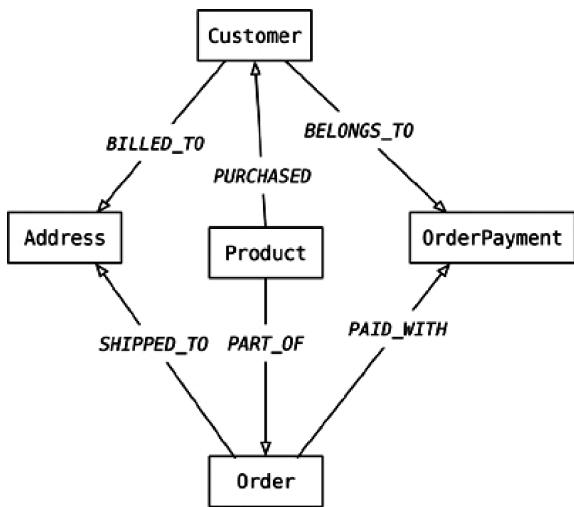


图 3.5 用“图模型”表示电子商务数据

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。