

大数据挑战与 NoSQL数据库技术



陆嘉恒 编著

BIG DATA

内 容 简 介

本书共分为三部分。理论篇重点介绍大数据时代下数据处理的基本理论及相关处理技术，并引入 NoSQL 数据库；系统篇主要介绍了各种类型 NoSQL 数据库的基本知识；应用篇对国内外几家知名公司在利用 NoSQL 数据库处理海量数据方面的实践做了阐述。

本书对大数据时代面临的挑战，以及 NoSQL 数据库的基本知识做了清晰的阐述，有助于读者整理思路，了解需求，并更有针对性、有选择地深入学习相关知识。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

大数据挑战与 NoSQL 数据库技术 / 陆嘉恒编著. —北京：电子工业出版社，2013.4
（大数据丛书）

ISBN 978-7-121-19660-7

I. ①大… II. ①陆… III. ①数据处理②数据库系统 IV. ①TP274 ②TP311.13

中国版本图书馆 CIP 数据核字(2013)第 035612 号

责任编辑：许 艳

印 刷：三河市鑫金马印装有限公司

装 订：三河市鑫金马印装有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×980 1/16 印张：27.5 字数：446.8 千字

印 次：2013 年 4 月第 1 次印刷

印 数：4000 册 定价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前言 |

为什么写本书

计算机技术已经深刻地影响了我们的工作、学习和生活。大数据及 NoSQL 技术是当下 IT 领域最炙手可热的话题，其发展非常迅速，潜力巨大，悄然改变着整个行业的面貌。随着 Web 2.0 技术的发展，微博、社交网络、电子商务、生物工程等领域的不断发展，各领域数据呈现爆炸式的增长，传统关系型数据库越来越显得力不从心。NoSQL 数据库技术的出现为眼下的问题提供了新的解决方案，它摒弃了传统关系型数据库 ACID 的特性，采用分布式多节点的方式，更加适合大数据的存储和管理。

政府和高校都十分重视对大数据及 NoSQL 技术的研究和投入；在产业界，各大 IT 公司也在投入大量的资源研究和开发相关的 NoSQL 产品，与之相应的新兴技术和产品正在不断涌现。这一切都极大地推动了 NoSQL 技术的发展。

大数据处理和 NoSQL 技术涉及的内容繁多，目前不同公司也有不同的 NoSQL 数据库产品，而且某一产品往往是为特定的应用而设计的，并不一定能够适用于所有的场景。很多人在学习的初始阶段需要进行大量的摸索和实践，然而目前这方面系统的参考资料却非常少。为了便于所有想了解 and 掌握 NoSQL 技术的朋友学习并在学习的过程中少走弯路，笔者将自己在该领域的经验和积累凝聚成了这本书，希望能够推动大数据处理及 NoSQL 相关技术在国内的发展。

本书面向的读者

在编写本书时，我们力图使不同背景和职业的读者都能从其中获益。

如果你是专业技术人员，本书将带领你快速进入大数据处理及 NoSQL 的世界，全面掌握 NoSQL 及其相关技术，能帮助你使用 NoSQL 技术解决当前面临的问题或提供必要的参考。

如果你是高等院校计算机及相关专业的学生，本书为你在课堂之外了解最新的 IT 打开一扇

窗户，能帮助你拓宽视野，完善知识结构，为迎接未来的挑战做好知识准备。

在学习本书之前，应具有如下的基础：

- 有一定的 Linux 操作系统的基础知识。
- 有较好的编程基础和阅读代码的能力。
- 对数据库知识有一定的了解。

如何阅读本书

本书一共包括 16 章，分为三个部分。其中第一部分为理论篇，包括：大数据产生的背景，数据一致性理论、数据存储模型、数据分区与防治策略、海量数据处理方法、数据复制与容错技术、数据压缩技术和数据缓存技术。此部分重点从理论上介绍、分析大数据管理过程中遇到的各方面问题。第二部分为系统篇，包括：键值数据库、列存数据库、文档数据库、图存数据库、基于 Hadoop 的数据库管理系统、NoSQL 数据库以及分布式缓存系统。该部分以理论篇为基础，根据数据存储模型对数据库类型进行划分，每一部分以具体开源数据库为实例进行介绍，涉及系统的架构、安装以及使用等方面，力图使读者对 NoSQL 数据库有具体的认识。第三部分为应用篇，包括企业应用以及总结和展望。该部分介绍企业如何使用 NoSQL 数据库解决自身遇到的问题。

在阅读本书时，读者可以先系统地学习理论篇的知识，目的是对海量数据处理方法有一个很好的理解，在此基础之上，读者可以对后面的章节进行选择性的学习。本书涉及内容较多，从开源数据库方面讲，包括了 Dynamo、Redis、Voldemort、Cassandra、Hypertable、CouchDB、MongoDB、Neo4j、GraphDB、OrientDB、HBase、Hive、Pig、MySQL Cluster、VolteDB、MS-Velocity、Memcached 等将近 20 个数据库。因此，建议读者可以重点学习感兴趣或有一定需求的数据库系统。当然，如果时间允许，还是建议读者系统地学习本书的内容。

另外，在系统篇的学习过程中，建议读者能够一边阅读，一边根据书中的指导动手实践，亲自实践本书中所给出的编程范例。

致谢

在本书的编写过程中，还有很多 NoSQL 领域的实践者和研究者为本书做了大量的工作，他们是张林林、许翔、程明、王海涌、顾向楠、吴少辉、杨宁、杨华、吴梦迪、任乔意、於洋、张轩等，在此特别感谢。

在线资源及勘误

本书官方网站为：<http://datasearch.ruc.edu.cn/NoSQL/>。本书的勘误、讨论以及相关资料等都会在该网站上发布和更新。

在本书的撰写和相关技术的研究中，尽管笔者投入了大量的精力，付出了艰辛的努力，然而受知识水平所限，错误和疏漏之处在所难免，恳请大家批评指正。如果有任何问题和建议，可发送邮件至 jiahenglu@gmail.com 或 jiahenglu@ruc.edu.cn。

陆嘉恒

目 录 |

第 1 章 概论	1
1.1 引子	2
1.2 大数据挑战	3
1.3 大数据的存储和管理	5
1.3.1 并行数据库	5
1.3.2 NoSQL数据管理系统	6
1.3.3 NewSQL数据管理系统	8
1.3.4 云数据管理	11
1.4 大数据的处理和分析	11
1.5 小结	13
参考文献	13

理 论 篇

第 2 章 数据一致性理论	16
2.1 CAP理论	17
2.2 数据一致性模型	21
2.3 ACID与BASE	22
2.4 数据一致性实现技术	23
2.4.1 Quorum系统NRW策略	23
2.4.2 两阶段提交协议	24
2.4.3 时间戳策略	27
2.4.4 Paxos	30

2.4.5 向量时钟	38
2.5 小结	43
参考文献	43
第3章 数据存储模型	45
3.1 总论	46
3.2 键值存储	48
3.2.1 Redis	49
3.2.2 Dynamo	49
3.3 列式存储	50
3.3.1 Bigtable	51
3.3.2 Cassandra与HBase	51
3.4 文档存储	52
3.4.1 MongoDB	53
3.4.2 CouchDB	53
3.5 图形存储	54
3.5.1 Neo4j	55
3.5.2 GraphDB	55
3.6 本章小结	56
参考文献	56
第4章 数据分区与放置策略	58
4.1 分区的意义	59
4.1.1 为什么要分区	59
4.1.2 分区的优点	60
4.2 范围分区	61
4.3 列表分区	62
4.4 哈希分区	63
4.5 三种分区的比较	64
4.6 放置策略	64
4.6.1 一致性哈希算法	65
4.6.2 容错性与可扩展性分析	66
4.6.3 虚拟节点	68

4.7 小结	69
参考文献	69
第 5 章 海量数据处理方法	70
5.1 MapReduce简介	71
5.2 MapReduce数据流	72
5.3 MapReduce数据处理	75
5.3.1 提交作业	76
5.3.2 初始化作业	78
5.3.3 分配任务	78
5.3.4 执行任务	79
5.3.5 更新任务执行进度和状态	80
5.3.6 完成作业	81
5.4 Dryad简介	81
5.4.1 DFS Cosmos介绍	82
5.4.2 Dryad执行引擎	84
5.4.3 DryadLINQ解释引擎	86
5.4.4 DryadLINQ编程	88
5.5 Dryad数据处理步骤	90
5.6 MapReduce vs Dryad	92
5.7 小结	94
参考文献	95
第 6 章 数据复制与容错技术	96
6.1 海量数据复制的作用和代价	97
6.2 海量数据复制的策略	97
6.2.1 Dynamo的数据库复制策略	97
6.2.2 CouchDB的复制策略	99
6.2.3 PNUTS的复制策略	99
6.3 海量数据的故障发现与处理	101
6.3.1 Dynamo的数据库的故障发现与处理	101
6.3.2 CouchDB的故障发现与处理	103
6.3.3 PNUTS的故障发现与处理	103

6.4 小结	104
参考文献	104
第 7 章 数据压缩技术	105
7.1 数据压缩原理	106
7.1.1 数据压缩的定义	106
7.1.2 数据为什么可以压缩	107
7.1.3 数据压缩分类	107
7.2 传统压缩技术 ^[1]	108
7.2.1 霍夫曼编码	108
7.2.2 LZ77 算法	109
7.3 海量数据带来的 3V 挑战	112
7.4 Oracle 混合列压缩	113
7.4.1 仓库压缩	114
7.4.2 存档压缩	114
7.5 Google 数据压缩技术	115
7.5.1 寻找长的重复串	115
7.5.2 压缩算法	116
7.6 Hadoop 压缩技术	118
7.6.1 LZ0 简介	118
7.6.2 LZ0 原理 ^[5]	119
7.7 小结	121
参考文献	121
第 8 章 缓存技术	122
8.1 分布式缓存简介	123
8.1.1 分布式缓存的产生	123
8.1.2 分布式缓存的应用	123
8.1.3 分布式缓存的性能	124
8.1.4 衡量可用性的标准	125
8.2 分布式缓存的内部机制	125
8.2.1 生命期机制	126
8.2.2 一致性机制	126

8.2.3	直读与直写机制	129
8.2.4	查询机制	130
8.2.5	事件触发机制	130
8.3	分布式缓存的拓扑结构	130
8.3.1	复制式拓扑	131
8.3.2	分割式拓扑	131
8.3.3	客户端缓存拓扑	131
8.4	小结	132
	参考文献	132

系 统 篇

第 9 章	key-value数据库	134
9.1	key-value模型综述	134
9.2	Redis	135
9.2.1	Redis概述	135
9.2.2	Redis下载与安装	135
9.2.3	Redis入门操作	136
9.2.4	Redis在业内的应用	143
9.3	Voldemort	143
9.3.1	Voldemort概述	143
9.3.2	Voldemort下载与安装	144
9.3.3	Voldemort配置	145
9.3.4	Voldemort开发介绍 ^[3]	147
9.4	小结	149
	参考文献	149
第 10 章	Column-Oriented数据库	150
10.1	Column-Oriented数据库简介	151
10.2	Bigtable数据库	151
10.2.1	Bigtable数据库简介	151
10.2.2	Bigtable数据模型	152
10.2.3	Bigtable基础架构	154

10.3	Hypertable数据库	157
10.3.1	Hypertable简介	157
10.3.2	Hypertable安装	157
10.3.3	Hypertable架构	163
10.3.4	基本概念和原理	164
10.3.5	Hypertable的查询	168
10.4	Cassandra数据库	175
10.4.1	Cassandra简介	175
10.4.2	Cassandra配置	175
10.4.3	Cassandra数据库的连接	177
10.4.4	Cassandra集群机制	180
10.4.5	Cassandra的读/写机制	182
10.5	小结	183
	参考文献	183
第 11 章	文档数据库	185
11.1	文档数据库简介	186
11.2	CouchDB数据库	186
11.2.1	CouchDB简介	186
11.2.2	CouchDB安装	188
11.2.3	CouchDB入门	189
11.2.4	CouchDB查询	200
11.2.5	CouchDB的存储结构	207
11.2.6	SQL和CouchDB	209
11.2.7	分布式环境中的CouchDB	210
11.3	MongoDB数据库	211
11.3.1	MongoDB简介	211
11.3.2	MongoDB的安装	212
11.3.3	MongoDB入门	215
11.3.4	MongoDB索引	224
11.3.5	SQL与MongoDB	226
11.3.6	MapReduce与MongoDB	229

11.3.7 MongoDB与CouchDB对比	234
11.4 小结	236
参考文献	237
第 12 章 图存数据库	238
12.1 图存数据库的由来及基本概念	239
12.1.1 图存数据库的由来	239
12.1.2 图存数据库的基本概念	239
12.2 Neo4j图存数据库	240
12.2.1 Neo4j简介	240
12.2.2 Neo4j使用教程	241
12.2.3 分布式Neo4j——Neo4j HA	251
12.2.4 Neo4j工作机制及优缺点浅析	256
12.3 GraphDB	258
12.3.1 GraphDB简介	258
12.3.2 GraphDB的整体架构	260
12.3.3 GraphDB的数据模型	264
12.3.4 GraphDB的安装	266
12.3.5 GraphDB的使用	268
12.4 OrientDB	276
12.4.1 背景	276
12.4.2 OrientDB是什么	276
12.4.3 OrientDB的原理及相关技术	277
12.4.4 Windows下OrientDB的安装与使用	282
12.4.5 相关Web应用	286
12.5 三种图存数据库的比较	288
12.5.1 特征矩阵	288
12.5.2 分布式模式及应用比较	289
12.6 小结	289
参考文献	290
第 13 章 基于Hadoop的数据管理系统	291
13.1 Hadoop简介	292

13.2	HBase	293
13.2.1	HBase体系结构	293
13.2.2	HBase数据模型	297
13.2.3	HBase的安装和使用	298
13.2.4	HBase与RDBMS	303
13.3	Pig	304
13.3.1	Pigr的安装和使用	304
13.3.2	Pig Latin语言	306
13.3.3	Pig实例	311
13.4	Hive	315
13.4.1	Hive的数据存储	316
13.4.2	Hive的元数据存储	316
13.4.3	安装Hive	317
13.4.4	HiveQL简介	318
13.4.5	Hive的网络接口 (WebUI)	328
13.4.6	Hive的JDBC接口	328
13.5	小结	330
	参考文献	331
第 14 章	NewSQL数据库	332
14.1	NewSQL数据库简介	333
14.2	MySQL Cluster	333
14.2.1	概述	334
14.2.2	MySQL Cluster的层次结构	336
14.2.3	MySQL Cluster的优势和应用	337
14.2.4	海量数据处理中的sharding技术	339
14.2.5	单机环境下MySQL Cluster的安装	343
14.2.6	MySQL Cluster的分布式安装与配置指导	348
14.3	VoltDB	350
14.3.1	传统关系数据库与VoltDB	351
14.3.2	VoltDB的安装与配置	351
14.3.3	VoltDB组件	354

14.3.4	Hello World	355
14.3.5	使用Generate脚本	361
14.3.6	Eclipse集成开发	362
14.4	小结	365
	参考文献	365
第 15 章	分布式缓存系统	366
15.1	Memcached缓存技术	367
15.1.1	背景介绍	367
15.1.2	Memcached缓存技术的特点	368
15.1.3	Memcached安装 ^[3]	374
15.1.4	Memcached中的数据操作	375
15.1.5	Memcached的使用	376
15.2	Microsoft Velocity分布式缓存系统	378
15.2.1	Microsoft Velocity简介	378
15.2.2	数据分类	379
15.2.3	Velocity核心概念	380
15.2.4	Velocity安装	382
15.2.5	一个简单的Velocity客户端应用	385
15.2.6	扩展型和可用性	387
15.3	小结	388
	参考文献	388

应 用 篇

第 16 章	企业应用	392
16.1	Instagram	393
16.1.1	Instagram如何应对数据的急剧增长	395
16.1.2	Instagram的数据分片策略	398
16.2	Facebook对Hadoop以及HBase的应用	400
16.2.1	工作负载类型	401
16.2.2	为什么采用Apache Hadoop和HBase	403
16.2.3	实时HDFS	405

16.2.4 Hadoop HBase的实现	409
16.3 淘宝大数据解决之道	411
16.3.1 淘宝数据分析	412
16.3.2 淘宝大数据挑战	413
16.3.3 淘宝OceanBase数据库	414
16.3.4 淘宝将来的工作	422
16.4 小结	423
参考文献	423

第 1 章

概 论

“这是最好的时代，这是最坏的时代；这是智慧的时代，这是愚蠢的时代；这是信仰的时期，这是怀疑的时期；这是光明的季节，这是黑暗的季节；这是希望之春，这是绝望之冬；人们面前什么都有，人们面前一无所有；人们正在直登天堂，人们正在直下地狱。”

——狄更斯《双城记》

对于数据管理界来说，这是一个充满挑战的时代。急速增长的数据让人们焦头烂额，传统关系型数据库在扩展性方面的瓶颈让人们无所适从：如何存储大数据，如何处理大数据，如何挖掘大数据，大数据已经成为数据管理界的新挑战。这又是一个充满机遇的时代，新的系统孕育而出，百花齐放，它们“标新立异”，它们“独树一帜”，它们在数据模型、事务处理等方面采取不同的策略解决海量数据带来的问题。这注定是一段不平凡的岁月。

1.1 引子

MySpace是全球知名的在线交友平台,自从 2004 年创建以来用户数迅速增长,直到Facebook崛起其用户数量才不断下降。虽然如今MySpace黯然衰落,但是其信息系统的发展值得大家借鉴,这里我们通过MySpace的例子^[3]来说明互联网公司是如何应对数据海量增长的,见表 1-1。

表 1-1 MySpace 信息系统的发展阶段

阶段（时间）	用户数	主要配置	主要瓶颈	应对策略
第一阶段	50 万	两台 Web 服务器 一台数据库服务器	用户访问量的增加	增加 Web 服务器
第二阶段	200 万	三台 SQL Server 数据库服务器	数据库服务器	增加数据库服务器 将用户数据垂直分割,并放置在不同数据服务器上
	300 万	三台 SQL Server 数据库服务器	数据库服务器	使用存储区域网络（SAN）将大量磁盘存储设备连接在一起
第三阶段 （分布式架构）	900 万	分布式计算架构 每台服务器约 200 万用户	数据库服务器	使用分布式计算架构对用户数据水平划分
第四阶段	1000 万	增加服务器	存储	使用微软.NET 框架
第五阶段	1700 万	增加服务器	存储	在 Web 服务器与数据库服务器之间增加缓存层
	2600 万	服务器内存增加到 32GB	存储	使用 64 位的 SQL Server 和 Windows Server

MySpace 最初的用户数量很小,两台 Web 服务器和一台数据库服务器就可以满足用户的访问。后来随着用户的增加,MySpace 公司通过增加 Web 服务器来应对访问量的增长。而当用户数量继续增长时,数据库服务器开始出现瓶颈,但是增加数据库服务器并不像增加 Web 服务器那样简单,分布式事务的代价也很高。MySpace 将用户数据垂直分割,如将用户的资料和博客存放于不同的数据库服务器上,随后又使用了存储区域网络（SAN）,这种高带宽的网络可以将大量磁盘存储设备连接在一起。

即使采用了以上措施,在用户数增加到三四百万的时候,数据库服务器还是出现了瓶颈。于是 MySpace 采用分布式架构。为了使服务器负载均衡,MySpace 对用户数据水平划分,每台

服务器存放二百万用户的数据。由于现有系统的水平扩展能力较差，MySpace 开始使用微软的解决方案，包括将系统采用.NET 框架、数据库采用 SQL Server。之后，MySpace 在 Web 服务层和数据库层增加了缓存层，并将其硬件和软件全面升级到 64 位。尽管求助于微软，但是 MySpace 的服务器经常超负荷运行，用户等待的时间较长。

整个过程发生在 2004—2006 年，仅仅两年时间 MySpace 的数据量增加了几十倍，这是互联网公司普遍遇到的问题，虽然后来 MySpace 受到 Facebook 崛起的影响其用户群迅速下降，但是海量数据的存储与管理仍然是互联网公司绕不开的问题。IDC¹ 数据显示，2006 年全世界的电子数据存储量为 18 万 PB，但是到了 2011 年这个数字已经到达 180 万 PB。

通过上面的案例我们可以看出，为了解决大数据及其高访问量带来的问题，MySpace 尝试了提高软硬件配置、增加服务器数目、采用分布式架构等方法，这些方法虽然提高了系统的性能，但并未完全解决大数据的问题，随着数据量的增加问题还会再次出现，大数据对现有 IT 架构的冲击是不可避免的。为了应对大数据的挑战，人们尝试转变思路，提出多种不同的解决方案，并构建各种各样的管理系统，这些系统可以水平扩展，可以很好地管理与分析大数据。

1.2 大数据挑战

什么是大数据？多大的数据量可以称为大数据？不同的年代有不同的答案^[2]。20 世纪 80 年代早期，大数据指的是数据量大到需要存储在数千万个磁带中的数据；20 世纪 90 年代，大数据指的是数据量超过单个台式机存储能力的数据库；如今，大数据指的是那些关系型数据库难以存储、单机数据分析统计工具无法处理的数据，这些数据需要存放在拥有数千万台机器的大规模并行系统上。大数据出现在日常生活和科学研究的各个领域，数据的持续增长使人们不得不重新考虑数据的存储和管理。

随着社会计算的兴起，人们习惯于在网上分享和交流信息。比如，社交网站 Facebook 拥有庞大的用户群，而且在不断增长。这些用户每天发出的日志以及分享的资料更是不计其数，其数据量已经达到 PB 级别，传统的解决方案已经不能很好地处理这些数据。Facebook 自己开发了 Cassandra 系统，现在又采用 HBase，这些针对海量数据的管理系统能够较好地为用户提供服务，而且具有可扩展性和容错性，这是解决大数据问题所需要的性能。微博服务商 Twitter 也面临大数据的挑战，消息的发送量达到每天数亿条，而查询量则达到每天数十亿次，这要求存储

1 IDC 是全球著名的信息技术、电信行业和消费科技市场咨询、顾问和活动服务专业提供商。

管理系统不仅能够存储大规模数据,而且能够提供高吞吐的读/写服务。Twitter 原先使用 MySQL 数据库,之后由于用户暴增便将数据迁移到 NoSQL 系统上,尽管 NoSQL 系统还未成熟,但却是解决海量数据的较为有效的方案。其他的互联网公司同样面临着大数据带来的问题,如 Google 搜索引擎需要处理大规模的网页信息,YouTube 则需要存储和提供用户分享的视频数据,维基百科提交用户分享的知识等,这些都涉及大规模数据信息存储与管理。

随着电子商务的发展,越来越多的人在网上选购商品,商务网站需要存储大量的商品信息和用户的交易信息,涉及大规模的数据。同时网站需要提供迅速的请求响应,以提高用户体验来吸引客户。而且网站还要对这些海量数据进行处理和分析,以便更有针对性地向用户推荐商品,海量数据成为系统构建和业务成败的关键因素。中国商业网站淘宝使用 HBase 来存储数据,同时不断探索自己的解决之路,开发了支持大数据的数据库系统 OceanBase 来实现部分在线应用。全球最大的线上拍卖和购物网站 eBay 也积极寻求海量数据的解决方案,其基于 Hadoop 建立了自己的集群系统 Athena 来处理大规模数据,同时开发了自己的开源云平台项目 Turmeric 来更好地开发和管理各种服务。同时,各大零售公司无论是线上销售还是实体销售,都会注意收集客户的消费信息以便有针对性地提供服务或推荐商品,这些都涉及大规模数据的应用。

各个领域的科学研究同样面临海量数据的挑战,从生物基因到天文气象,从物理实验到临床医学,得益于测量技术和设备的发展,这些领域在实验或实践中产生了大量的数据,而人们需要对这些数据进行处理分析从而挖掘出有价值的信息,但这不是容易的事情。随着下一代基因测序技术的发展,基因中所蕴含的信息逐渐被人们所发掘,人们获得更多更准确的基因数据,但是如何匹配基因数据,如何从这些数据中挖掘出所需要的信息,这是生物信息学遇到的新挑战。在环境气象研究中,科学家已经收集了数十年甚至上百年的气象环境数据,在这些数据中分析气候的变化需要海量数据处理技术的支持。在医学药物研究中搜集的大量的病人生理数据和药物测试数据,这些数据的规模很大,需要从中分析出有用的信息。在人文社会科学中,社会学家开始注意互联网社交网络上的人际交往和社会关系,其涉及的数据量也是非常巨大的,从海量数据中找出社会学家感兴趣的内容是富有挑战性的。人工智能研究方面,人们希望计算机拥有人类的学习能力和逻辑推理能力,这就需要机器存储大量的经验数据和知识数据,还需要从这些大量数据中迅速获得所需要的内容,并对其进行分析处理,从而做出正确有效的判断。

如今传感器的广泛使用,数据采集更加方便,这些传感器会连续地产生数据,如实时监控系统、网络流量监测等。除了传感器源源不断地产生数据外,许多领域都会涉及流数据,如经济金融领域中股票价格和交易数据、零售业中的交易数据、通信领域中的数据等都是流数据,

这些数据最大的特点就是海量，因为它们每时每刻连续不断地产生，但与其他的海量数据不同，流数据连续有序、变化迅速，而且对处理分析的响应度要求较高，因此对于流数据的处理和挖掘往往采用不同的方法。经济金融领域各个方面都产生海量数据，如证券价格变化和股票交易形成的流数据，企业或个人各种经济活动而产生的数据等。现代经济已经步入海量数据时代，在新时代下可以带来创新和生产率增长，并可能出现新的商业模式。利用好经济生活产生的海量数据，可以发挥重要的经济作用，不仅有利于企业的商业活动，也有利于国民经济，提高国家的竞争力。面对大规模的经济数据，人们除了需要提高获取、存储和分析数据的能力，同时需要保障数据的安全和隐私，但这仍然是巨大的挑战。

传统的关系型数据库并不能够很好地解决海量数据带来的问题，单机的统计和可视化工具也变得力不从心。一些新的数据管理系统如并行数据库、网格数据库、分布式数据库、云平台、可扩展数据库等孕育而生，它们为解决海量数据提供了多种选择。

1.3 大数据的存储和管理

任何机器都会有物理上的限制：内存容量、硬盘容量、处理器速度等等，我们需要在这些硬件的限制和性能之间做出取舍，比如内存的读取速度比硬盘快得多，因此内存数据库比硬盘数据库性能好，但是内存为 2GB 的机器不可能将大小为 100GB 的数据全部放入内存中，也许内存大小为 128GB 的机器能够做到，但是数据增加到 200GB 时就无能为力了。

数据不断增长造成单机系统性能不断下降，即使不断提升硬件配置也难以跟上数据的增长速度。然而，当今主流的计算机硬件比较便宜而且可以扩展，现在购置八台 8 内核、128GB 内存的机器比购置一台 64 内核、TB 级别内存的服务器划算得多，而且还可以增加或减少机器来应对将来的变化。这种分布式架构策略对于海量数据来说是比较适合的，因此，许多海量数据系统选择将数据放在多个机器中，但也带来了许多单机系统不曾有的问题。

下面我们介绍大数据存储和管理发展过程中出现的四类大数据存储和管理数据库系统。

1.3.1 并行数据库

并行数据库^[1]是指那些在无共享的体系结构中进行数据操作的数据库系统。这些系统大部分采用了关系数据模型并且支持SQL语句查询，但为了能够并行执行SQL的查询操作，系统中采用了两个关键技术：关系表的水平划分和SQL查询的分区执行。

水平划分的主要思想就是根据某种策略将关系表中的元组分布到集群中的不同节点上，这些节点上的表结构是一样的，这样就可以对元组并行处理。现有的分区策略有哈希分区、范围分区、循环分区等。例如，哈希分区策略是将表 T 中的元组分布到 n 个节点上，可以使用统一的哈希算法对元组中的某个或某几个属性进行哈希，如 $hash(T.attribute1) \bmod n$ ，然后根据哈希值将元组放置到不同的节点上。

在分区存储的表中处理SQL查询需要使用基于分区的执行策略，如获取表 T 中某一数值范围内的元组，系统首先为整个表 T 生成总的执行计划 P ，然后将 P 拆分成 n 个子计划 $\{P_1, \dots, P_n\}$ ，子计划 P_i 在节点 n_i 上独立执行，最后每个节点将生成的中间结果发送到某一选定的节点上，该节点对中间结果进行聚集产生最终的结果。

并行数据库系统的目标是高性能和高可用性，通过多个节点并行执行数据库任务，提高整个数据库系统的性能和可用性。最近一些年不断涌现一些提高系统性能的新技术，如索引、压缩、实体化视图、结果缓存、I/O 共享等，这些技术都比较成熟且经得起时间的考验。与一些早期的系统如 Teradata 必须部署在专有硬件上不同，最近开发的系统如 Aster、Vertica 等可以部署在普通的商业机器上，这些数据库系统可以称得上准云系统。

并行数据库系统的主要缺点就是没有较好的弹性，而这种特性对中小型企业 and 初创企业是有利的。人们在对并行数据库进行设计和优化的时候认为集群中节点的数量是固定的，若需要对集群进行扩展和收缩，则必须为数据转移过程制订周全的计划。这种数据转移的代价是昂贵的，并且会导致系统在某段时间内不可访问，而这种较差的灵活性直接影响到并行数据库的弹性以及现用现付商业模式的实用性。

并行数据库的另一个问题就是系统的容错性较差，过去人们认为节点故障是个特例，并不经常出现，因此系统只提供事务级别的容错功能，如果在查询过程中节点发生故障，那么整个查询都要从头开始重新执行。这种重启任务的策略使得并行数据库难以在拥有数以千个节点的集群上处理较长的查询，因为在这类集群中节点的故障经常发生。基于这种分析，并行数据库只适合于资源需求相对固定的应用程序。不管怎样，并行数据库的许多设计原则为其他海量数据系统的设计和优化提供了比较好的借鉴。

1.3.2 NoSQL 数据管理系统

NoSQL^[5]一词最早出现于 1998 年，它是 Carlo Strozzi 开发的一个轻量、开源、不提供 SQL 功能的关系型数据库（他认为，由于 NoSQL 悖离传统关系数据库模型，因此，它应该有一个全新的名字，比如 “NoREL” 或与之类似的名字^[6]）。

2009 年, Last.fm 的 Johan Oskarsson 发起了一次关于分布式开源数据库的讨论^[7], 来自 Rackspace 的 Eric Evans 再次提出了 NoSQL 的概念, 这时的 NoSQL 主要指非关系型、分布式、不提供 ACID 的数据库设计模式。

2009 年在亚特兰大举行的 “no:sql(east)” 讨论会是一个里程碑, 其口号是 “select fun, profit from real_world where relational=false;”。因此, 对 NoSQL 最普遍的解释是 “非关系型的”, 强调键值存储和文档数据库的优点, 而不是单纯地反对关系型数据库。

传统关系型数据库在处理数据密集型应用方面显得力不从心, 主要表现在灵活性差、扩展性差、性能差等方面。最近出现的一些存储系统摒弃了传统关系型数据库管理系统的设计思想, 转而采用不同的解决方案来满足扩展性方面的需求。这些没有固定数据模式并且可以水平扩展的系统现在统称为 NoSQL (有些人认为称为 NoREL 更为合理), 这里的 NoSQL 指的是 “Not Only SQL”, 即对关系型 SQL 数据系统的补充。NoSQL 系统普遍采用的一些技术有:

- **简单数据模型。**不同于分布式数据库, 大多数 NoSQL 系统采用更加简单的数据模型, 这种数据模型中, 每个记录拥有唯一的键, 而且系统只需支持单记录级别的原子性, 不支持外键和跨记录的关系。这种一次操作获取单个记录的约束极大地增强了系统的可扩展性, 而且数据操作就可以在单台机器中执行, 没有分布式事务的开销。
- **元数据和应用数据的分离。**NoSQL 数据管理系统需要维护两种数据: 元数据和应用数据。元数据是用于系统管理的, 如数据分区到集群中节点和副本的映射数据。应用数据就是用户存储在系统中的商业数据。系统之所以将这两类数据分开是因为它们有着不同的一致性要求。若要系统正常运转, 元数据必须是一致且实时的, 而应用数据的一致性需求则因应用场合而异。因此, 为了达到可扩展性, NoSQL 系统在管理两类数据上采用不同的策略。还有一些 NoSQL 系统没有元数据, 它们通过其他方式解决数据和节点的映射问题。
- **弱一致性。**NoSQL 系统通过复制应用数据来达到一致性。这种设计使得更新数据时副本同步的开销很大, 为了减少这种同步开销, 弱一致性模型如最终一致性和时间轴一致性得到广泛应用。

通过这些技术, NoSQL 能够很好地应对海量数据的挑战。相对于关系型数据库, NoSQL 数据存储管理系统的主要优势有:

- **避免不必要的复杂性。**关系型数据库提供各种各样的特性和强一致性, 但是许多特性只能在某些特定的应用中使用, 大部分功能很少被使用。NoSQL 系统则提供较少的功能来提高性能。

- **高吞吐量。**一些 NoSQL 数据系统的吞吐量比传统关系数据管理系统要高很多，如 Google 使用 MapReduce 每天可处理 20PB 存储在 Bigtable 中的数据。
- **高水平扩展能力和低端硬件集群。**NoSQL 数据系统能够很好地进行水平扩展，与关系型数据库集群方法不同，这种扩展不需要很大的代价。而基于低端硬件的设计理念为采用 NoSQL 数据系统的用户节省了很多硬件上的开销。
- **避免了昂贵的对象-关系映射。**许多 NoSQL 系统能够存储数据对象，这就避免了数据库中关系模型和程序中对象模型相互转化的代价。

NoSQL 向人们提供了高效便宜的数据管理方案，许多公司不再使用 Oracle 甚至 MySQL，他们借鉴 Amazon 的 Dynamo 和 Google 的 Bigtable 的主要思想建立自己的海量数据存储管理系统，一些系统也开始开源，如 Facebook 将其开发的 Cassandra 捐给了 Apache 软件基金会。

虽然 NoSQL 数据库提供了高扩展性和灵活性，但是它也有自己的缺点，主要有：

- **数据模型和查询语言没有经过数学验证。**SQL 这种基于关系代数和关系演算的查询结构有着坚实的数学保证，即使一个结构化的查询本身很复杂，但是它能够获取满足条件的所有数据。由于 NoSQL 系统都没有使用 SQL，而使用的一些模型还未有完善的数学基础。这也是 NoSQL 系统较为混乱的主要原因之一。
- **不支持 ACID 特性。**这为 NoSQL 带来优势的同时也是其缺点，毕竟事务在很多场合下还是需要的，ACID 特性使系统在中断的情况下也能够保证在线事务能够准确执行。
- **功能简单。**大多数 NoSQL 系统提供的功能都比较简单，这就增加了应用层的负担。例如如果在应用层实现 ACID 特性，那么编写代码的程序员一定极其痛苦。
- **没有统一的查询模型。**NoSQL 系统一般提供不同查询模型，这一定程度上增加了开发者的负担。

1.3.3 NewSQL 数据管理系统

人们曾普遍认为传统数据库支持 ACID 和 SQL 等特性限制了数据库的扩展和处理海量数据的性能，因此尝试通过牺牲这些特性来提升对海量数据的存储管理能力，但是现在一些人则持有不同的观念，他们认为并不是 ACID 和支持 SQL 的特性，而是其他的一些机制如锁机制、日志机制、缓冲区管理等制约了系统的性能，只要优化这些技术，关系型数据库系统在处理海量数据时仍能获得很好的性能。

关系型数据库处理事务时对性能影响较大、需要优化的因素有：

- **通信**。应用程序通过 ODBC 或 JDBC 与 DBMS 进行通信是 OLTP 事务中的主要开销。
- **日志**。关系型数据库事务中对数据的修改需要记录到日志中，而日志则需要不断写到硬盘上来保证持久性，这种代价是昂贵的，而且降低了事务的性能。
- **锁**。事务中修改操作需要对数据进行加锁，这就需要在锁表中进行写操作，造成了一定的开销。
- **闕**。关系型数据库中一些数据结构，如 B 树、锁表、资源表等的共享影响了事务的性能。这些数据结构常常被多线程读取，所以需要短期锁即闕。
- **缓冲区管理**。关系型数据将数据组织成固定大小的页，内存中磁盘页的缓冲管理会造成一定的开销。

为了解决上面的问题，一些新的数据库采用部分不同的设计，它取消了耗费资源的缓冲池，在内存中运行整个数据库。它还摒弃了多线程服务的锁机制，也通过使用冗余机器来实现复制和故障恢复，取代原有的昂贵的恢复操作。这种可扩展、高性能的SQL数据库被称为NewSQL，其中“New”用来表明与传统关系型数据库系统的区别，但是NewSQL也是很宽泛的概念。它首先由 451 集团²在一份报告中提出，其主要包括两类系统：拥有关系型数据库产品和服务，并将关系模型的好处带到分布式架构上；或者提高关系数据库的性能，使之达到不用考虑水平扩展问题的程度。前一类NewSQL包括Clustrix、GenieDB、ScalArc、ScaleBase、NimbusDB，也包括带有NDB的MySQL集群、Drizzle等。后一类NewSQL包括Tokutek、JustOne DB。还有一些“NewSQL即服务”，包括Amazon的关系数据库服务、Microsoft的SQL Azure、FathomDB等。

当然，NewSQL 和 NoSQL 也有交叉的地方，例如，RethinkDB 可以看作 NoSQL 数据库中键/值存储的高速缓存系统，也可以当作 NewSQL 数据库中 MySQL 的存储引擎。现在许多 NewSQL 提供商使用自己的数据库为没有固定模式的数据提供存储服务，同时一些 NoSQL 数据库开始支持 SQL 查询和 ACID 事务特性。

NewSQL 能够提供 SQL 数据库的质量保证，也能提供 NoSQL 数据库的可扩展性。VoltDB 是 NewSQL 的实现之一，其开发公司的 CTO 宣称，它们的系统使用 NewSQL 的方法处理事务的速度比传统数据库系统快 45 倍。VoltDB 可以扩展到 39 个机器上，在 300 个 CPU 内核中每分钟处理 1600 万事务，其所需的机器数比 Hadoop 集群要少很多。

2 451 集团是涉及最新技术、商业模式的分析及咨询公司。

随着NoSQL、NewSQL数据库阵营的迅速崛起，当今数据库系统“百花齐放”，现有系统达数百种之多，图 1-1 将广义的数据库系统进行了分类³。

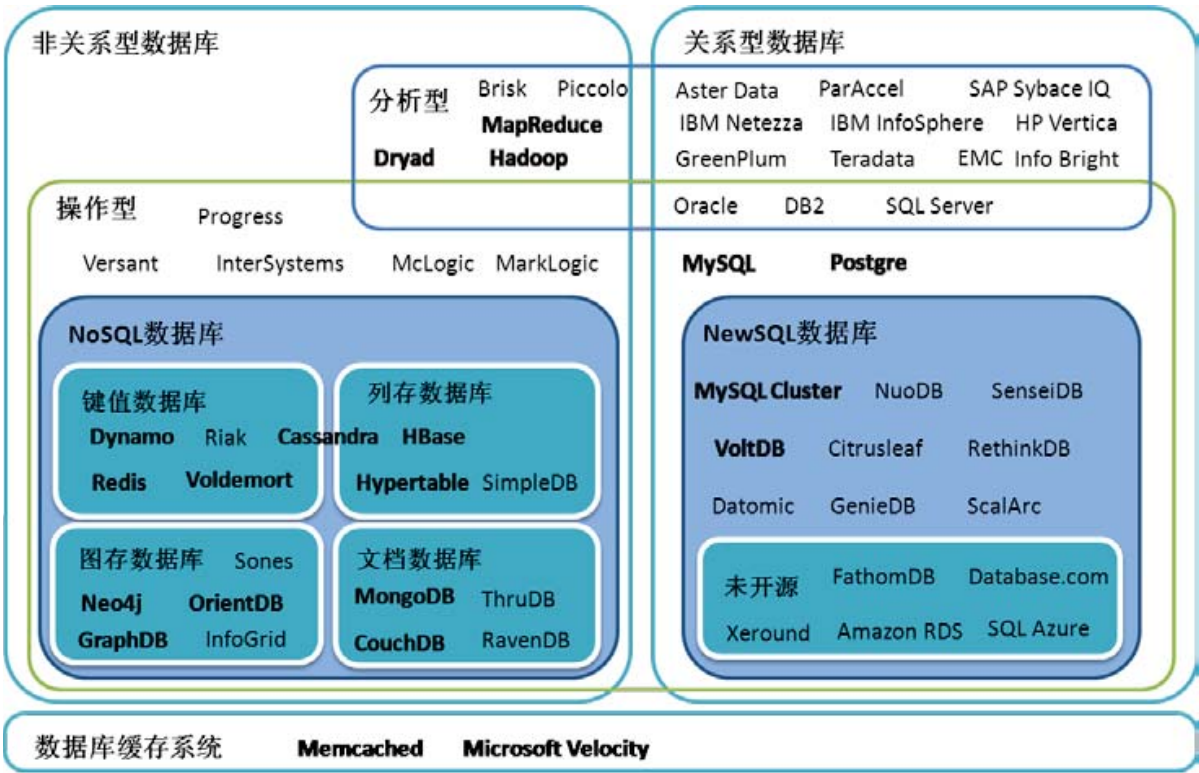


图 1-1 数据库系统的分类⁴

图 1-1 中将数据库分为关系型数据库、非关系型数据库以及数据库缓存系统。其中，非关系型数据库主要指的是 NoSQL 数据库，分为：键值数据库、列存数据库、图存数据库以及文档数据库四大类。关系型数据库包含了传统关系数据库系统以及 NewSQL 数据库。

高容量、高分布式、高复杂性应用程序的需求迫使传统数据库不断扩展自己的容量极限，这些驱动传统关系型数据库采用不同的数据管理技术的 6 个关键因素可以概括为“SPRAIN”，即：

- 可扩展性（Scalability）——硬件价格
- 高性能（Performance）——MySQL 的性能瓶颈
- 弱一致性（Relaxed consistency）——CAP 理论
- 敏捷性（Agility）——持久多样性

3 <http://nosql-database.org/>
4 粗体形式的数据库为本书所涉及的数据库。

- 复杂性（Intricacy）——海量数据
- 必然性（Necessity）——开源

1.3.4 云数据管理

云数据管理^[4]指的是“数据库即服务”，用户无须在本机安装数据库管理软件，也不需要搭建自己的数据管理集群，而只需要使用服务提供商提供的数据库服务。比较著名的服务有Amazon提供的关系型数据库服务RDS和非关系型数据库服务SimpleDB。

云数据管理系统的优势就是可以弹性地分配资源，用户只需为所使用的资源付费即可。这使得用户对资源的需求可以动态扩展或缩减。例如，需要对大小为 1TB 和 100 GB 的两个数据集分别进行分析，若在弹性伸缩的模式下，我们可以在云中分配 100 个节点处理 1TB 的数据集，然后将集群缩减到 10 个节点来处理 100 GB 的数据集。假设数据处理系统是线性扩展的，那么两个处理任务大约在相同的时间内完成。这样，弹性伸缩的能力加上现用现付的商业模式会提供较高的性价比。

云数据管理系统的主要优势有：

- **透明性**。用户无须考虑服务实现所使用的硬件和软件，利用其提供的接口使用其服务即可。
- **可伸缩性**。伸缩性是云系统提供的重要特性，用户根据自己的需求申请各种资源即可，而且需求还可以动态变化。
- **高性价比**。用户无须购买自己的基础设施和软件，节约了硬件费用及软件版权费用。

云数据管理系统也有不足的地方，如用户隐私和数据安全问题、服务可靠性问题、服务质量保证问题等等。

1.4 大数据的处理和分析

对于大数据，串行的处理方式难以满足人们的要求，现在主要采用并行计算方式。现有的并行计算可以分为两种：

- **细粒度的并行计算**。这里细粒度主要是指指令或进程级别，由于 GPU 比 CPU 拥有更强的并行处理能力，人们将一些任务交给 GPU 并行处理，一些 GPU 制造商也推出了方便程序员使用的编程模型，如 NVIDIA 推出的 CUDA 等。

- **粗粒度的并行计算。**这里粗粒度指的是任务级别，人们将工作分布到不同机器中执行，最近流行的网格计算、分布式计算都属于粗粒度级别。

由于现有 GPU 编程模型还未完善，开发人员需要考虑大量的并行细节且任务较重，因此未得到流行。而一些新推出的分布式编程模型以其简单、方便等特点受到开发人员的欢迎并变得炙手可热，这里我们主要讨论粗粒度的并行计算。

由于大数据都分布在集群中，因此对数据的处理和分析需要在集群中进行，但是在多台机器上对分布式数据进行分析会产生巨大的性能开销，即使采用千兆比特或万兆比特带宽的网络，随机读取速度和连续读取速度都会比内存慢几个数量级。但是，现在高速局域网技术使得网络读取速度比硬盘读取要快很多。因此，将数据存储在其他节点上比存储在硬盘上的性能要好，而且还可以在多个节点上并行处理数据集。

对大数据分布处理会带来一些问题，首先就是节点间通信对并行处理的代价，一些操作如搜索、计数、部分聚集、联合等可以在每个节点上独立执行。单个节点处理后的结果需要合并，因此节点间的通信是不可避免的，但是并不是所有的聚集操作都能分散成可以独立操作的子操作，如求得所有数据的中位数。不过，大部分重要的操作都有分布式算法来减少节点间的通信。

节点间负载不平衡也是出现的主要问题。理想情况下，每个节点的计算量是相同的，否则工作量最大的节点将决定整个任务的完成时间，这个时间往往比负载均衡情况下的时间要长。最坏的情况下，所有的工作都集中在某个机器上，无法体现出并行的优势。数据在节点间如何分布对负载平衡产生影响，例如，一个包含 1000 个传感器 10 年内的观测值的数据集，传感器每 15 秒收集一次数据，这样一个传感器 10 年内将产生两千多万个观测值。我们将数据根据传感器并按时间顺序分布到 10 个节点上，每个节点包含 100 个传感器的观测值，如果对某个传感器收集的数据进行操作，那么大部分节点将处于闲置状态。如果先按时间顺序对数据进行分布，那么根据时间的操作也会造成负载不平衡。

分布式系统的另一个问题就是可靠性。就像拥有四个引擎的飞机比拥有两个引擎的飞机更容易出现引擎故障一样，一个拥有 10 个节点的集群很容易出现节点故障。这可以通过在节点间复制数据来解决，对数据进行复制，既可以提高数据分析的效率，也可以通过冗余来应对节点故障。当然，数据集越大，对数据副本的管理和维护也越困难。

目前对大数据处理和分析的应用更多的是集中在数据仓库技术、预测分析、实时分析、商业智能、数据统计等方面。这些需求对企业有巨大的帮助。

将 PB 级的数据存储起来并不是一件困难的事情，但是如何进行高效的存储并不简单。首先要考虑的是，如何组织数据的结构使其能够更多地支持上层的软件，而不需要对数据进行转储和重新组织。当数据需要发生转换的时候避免因转储、抽取、整合等而带来的延迟。

有效的预测分析技术，尤其是实时分析对企业的决策有很大的帮助。例如，超市可以根据庞大的用户历史消费记录来预测某一用户下次购买商品的倾向，从而在结账的时候可以专门针对某一用户打印其关心的优惠券。足球队管理层可以根据用户的购票记录为其推荐更人性化的月票、季票等套票。

目前，像 SAS、SPSS 等传统数据分析软件因其数据处理能力受限于单机的计算能力，对大数据的处理显得力不从心。IBM Netezza 等新兴的数据分析软件往往需要支付昂贵的许可费用，因此 Hadoop，MapReduce，R 等开源的大数据分析工具受到越来越多的关注和青睐。

相比于商业软件，开源软件完全免费且不需要支付昂贵的许可费用，另外在其背后还拥有庞大的开源团队的支持。但是能否完全跟得上市场的需求和发展速度是关键性的问题，毕竟这些软件不像商业软件那样有巨大的利益驱动推动它们的发展。

1.5 小结

本章首先介绍了大数据产生的背景以及大数据所带来的挑战：正是由于数据规模的迅速持续增长动摇了传统关系型数据管理系统的霸主地位，使得人们寻求其他的解决方案，促使新的管理系统和处理框架的产生。然后介绍了新的数据管理系统，主要有 NoSQL 数据库和 NewSQL 数据库。最后简要介绍了大数据的处理方法和常见的应用需求。在下面的章节中，我们将对大数据管理方法和 NoSQL 数据库系统进行详细介绍。

参考文献

- [1] Erik Meijer, Gavin Bierman. A Co-Relational Model of Data for Large Shared Data Banks[J]. 2011, 54(4). Communications of the ACM, 2011.
- [2] Adam Jacobs. The Pathologies of Big Data[J]. 2009, 52(8). Communications of the ACM, 2009.
- [3] Wiseman. 通过了解 MySpace 的六次重构经历，来认识分布式系统到底该如何创建 [EB/OL]. 2007-07-17[2012-06-10]. <http://www.cnblogs.com/wuxilin/archive/2007/07/17/820482.html>.

- [4] 陆嘉恒. 分布式系统及云计算概论[M]. 北京: 清华大学出版社, 2010.
- [5] Wikipedia. NoSQL [EB/OL]. 2012-05[2012-06]. <http://en.wikipedia.org/wiki/NoSQL>.
- [6] Strozzi. NoSQL: a non-SQL RDBMS [EB/OL]. http://www.strozzi.it/cgi-bin/CSA/tw7/I/en_US/nosql/Home%20Page.
- [7] Eric Evan. NOSQL 2009 [EB/OL]. 2010- 03[2012-06].
http://blog.sym-link.com/2009/05/12/nosql_2009.htm.

理 论 篇

第 2 章

数据一致性理论

本章导读

本章主要介绍海量数据管理中的数据一致性理论，包括 CAP 理论、BASE 模型、数据一致性模型，以及现有的经典数据一致性技术。海量数据管理涉及多方面的内容，比如海量数据存储放置策略、海量数据一致性策略、并行计算方法、索引技术、海量数据库等。其中数据一致性理论为海量数据管理的理论基础，了解该方面的内容有助于读者对本书的阅读和理解。本章对于每一种数据一致性实现技术均结合生动的实例进行讲解。

本章首先介绍 CAP 理论所包含的内容及其主要特征。在了解具体的数据一致性实现技术之后，希望读者能够对所介绍的数据库一致性实现技术进行简单比较，发现它们的异同之处，以及它们与 CAP 理论的联系。

本章要点

- CAP 理论
- 数据一致性模型
- BASE 模型
- 数据一致性实现技术
 - ◆ Quorum 系统的 NWR 策略
 - ◆ 两阶段提交协议
 - ◆ 时间戳策略
 - ◆ PAXOS 理论
 - ◆ 向量时钟理论

2.1 CAP 理论

CAP理论^[6]由Eric Brewer在ACM PODC会议上的主题报告^[1]中提出，这个理论是NoSQL数据管理系统构建的基础，如图 2-1 所示。



图 2-1 CAP 理论

其中字母“C”、“A”、“P”分别代表以下三个特征。

- **强一致性（Consistency）**。系统在执行过某项操作后仍然处于一致的状态。在分布式系统中，更新操作执行成功后所有的用户都应该读取到最新的值，这样的系统被认为具有强一致性。
- **可用性（Availability）**。每一个操作总是能够在一定的时间内返回结果，这里需要注意的是“一定时间内”和“返回结果”。

“一定时间内”是指，系统的结果必须在给定时间内返回，如果超时则被认为不可用。这是至关重要的。比如通过网上银行的网络支付功能购买物品。当等待了很长时间，如 15 分钟，系统还是没有返回任务操作结果，购买者一直处于等待状态，那么购买者就不知道现在是否支付成功，还是需要进行其他操作。这样当下次购买者再次使用网络支付功能时必将心有余悸。

“返回结果”同样非常重要。还是拿这个例子来说，假如购买者点击支付之后很快出现了结果，但是结果却是“java.lang.error……”之类的错误信息。这对于普通购买者来说相

当于没有返回任何结果。因为他仍旧不知道系统处于什么状态，是支付成功还是失败，或者需要重新操作。

- **分区容错性 (Partition Tolerance)**。分区容错性可以理解为系统在存在网络分区的情况下仍然可以接受请求（满足一致性和可用性）。这里网络分区是指由于某种原因网络被分成若干个孤立的区域，而区域之间互不相通。还有一些人将分区容错性理解为系统对节点动态加入和离开的处理能力，因为节点的加入和离开可以认为是集群内部的网络分区。

CAP 是在分布式环境中设计和部署系统时所要考虑的三个重要的系统需求。根据 CAP 理论，数据共享系统只能满足这三个特性中的两个，而不能同时满足三个条件。因此系统设计者必须在这三个特性之间做出权衡。例如 Amazon 的 Dynamo 具有高可用性和分区容错性但不支持强一致性，也就是说用户不能立即看到其他用户更新的内容。

下面通过图示来解释数据共享系统为什么不能同时满足 CAP 理论三个条件。

如图 2-1 所示，在网络中有两个节点分别为 G_1 和 G_2 ，这两个节点上存储着同一数据的不同副本，现在数据是一致的，两个副本的值都为 V_0 ，A、B 分别是运行在 G_1 、 G_2 上与数据交互的应用程序。

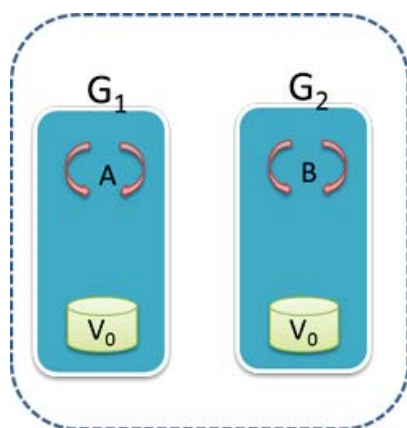


图 2-1 网络节点及数据分布图

在正常情况下，操作过程如下（如图 2-2 所示）：

- (1) A 将 V_0 更新，数据值为 V_1 ；
- (2) G_1 发送消息 m 给 G_2 ，数据 V_0 更新为 V_1 ；
- (3) B 读取到 G_2 中的数据 V_1 。

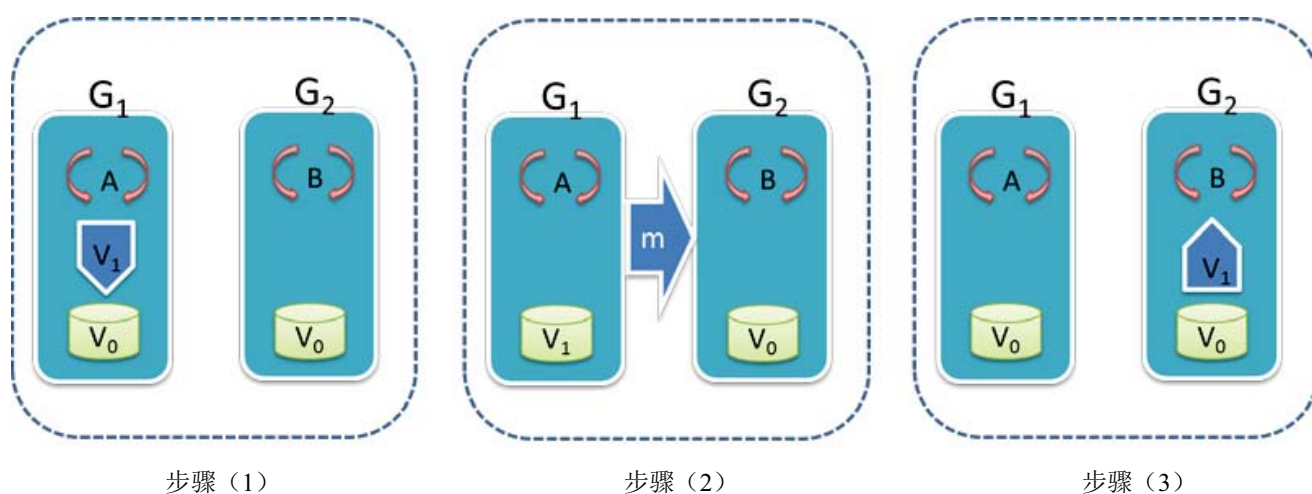


图 2-2 正常情况

如果发生网络分区故障，那么在操作的步骤（2）将发生错误： G_1 发送的消息不能传送到 G_2 上。这样数据就处于不一致的状态， B 读取到的就不是最新的数据，如图 2-3 所示。如果我们采用一些技术如阻塞、加锁、集中控制等来保证数据的一致，那么必然会影响到系统的可用性和分区容错性。

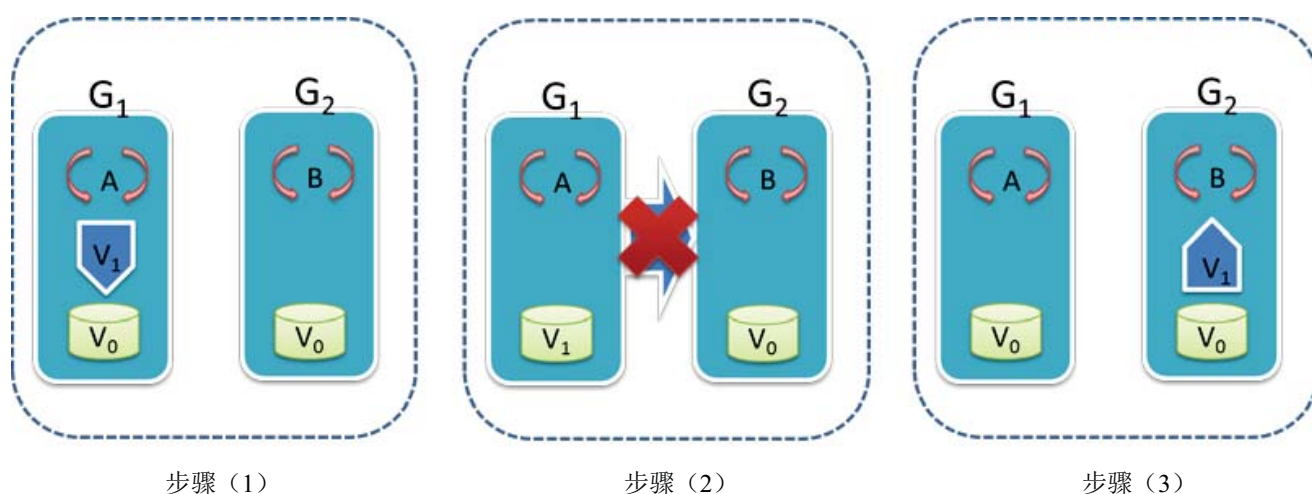


图 2-3 网络分区故障

CAP理论告诉我们如果系统具有较高的可用性和较小延迟，那么节点必须能够容忍网络分区，但这时候应用程序可能会得到不同的数据（ B 读取到的值为 V_1 ）。

有人可能会想，如果我们对步骤（2）的操作加一个同步消息，问题会不会解决呢（同时满足CAP理论三个特性）？答案是否定的。不加同步的情况下， G_1 到 G_2 的更新是不可知的，也就是说， B 读取到的数据可能是 V_1 也可能是 V_2 ，但是服务是可用的。即同时满足了“A”和“P”，但是不保证“C”一定满足。加了同步消息之后，将能够保证 B 读取到数据 V_1 。但是这个同步操

作必定要消耗一定的时间，尤其是在网络规模较大的情况下，当节点规模成百上千的时候，不一定能保证此时的服务是可用的。这种情况下只能满足“C”和“P”，而不能保证“A”一定满足。

CAP 理论不但对此网络和通信模型有效，对其他模型同样有效，有兴趣的读者可以换其他的网络和通信模型来验证 CAP 理论。

根据 CAP 理论，系统满足三个条件中不同的两个条件会具有不同的特点，见表 2-1。

表 2-1 处理 CAP 问题的选择

序 号	选 择	特 点	例 子
1	C、A	两阶段提交、缓存验证协议	传统数据库、集群数据库、LDAP、GFS 文件系统
2	C、P	悲观加锁	分布式数据库、分布式加锁
3	A、P	冲突处理、乐观	DNS、Coda

可以看出，三种不同的组合对应着放弃了 CAP 三个特性其中的一个。

- **放弃 P**：如果想避免分区容错性问题的发生，一种做法是将所有的数据（与事务相关的）都放到一台机器上。虽然无法 100%地保证系统不会出错，但不会碰到由分区带来的负面效果。当然，这个选择会严重影响系统的扩展性。
- **放弃 A**：相对于放弃“分区容错性”来说，其反面就是放弃可用性。一旦遇到分区容错故障，那么受到影响的服务需要等待数据一致，因此在等待期间系统就无法对外提供服务。
- **放弃 C**：这里所说的放弃一致性，并不是完全放弃数据的一致性，而是放弃数据的强一致性，而保留数据的最终一致性。以网络购物为例，对只剩最后一件库存的商品，如果同时接收到了两份订单，那么较晚的订单将被告知商品售罄。
- **其他选择**：引入 BASE（Basically Available, Soft-state, Eventually consistent），该方法支持最终一致性，其实是放弃 C 的一个特例，我们将在后文进行介绍。

传统关系型数据管理系统注重数据的强一致性，但是对于海量数据的分布式存储和处理其性能不能满足人们的需求，因此现在许多 NoSQL 数据库牺牲了强一致性来提高性能，CAP 理论对于非关系性数据库的设计有很大的影响并被 NoSQL 阵营所认可。

CAP理论已经得到人们的普遍认同，但是并不是所有人都这样认为，这种不同的声音主要是因为对CAP的概念有不同的理解所造成的^[7]。此外，我们不能说一种解决方案在任何时间任何环境下永远都是正确的，因为随着时间和环境的变迁情况将会有所不同，正如随着海量数据

的出现，传统关系型数据库已经不能完全满足人们的需求一样。因此，我们应该肯定CAP理论对海量数据管理的研究与发展所作出的贡献。

2.2 数据一致性模型

一些分布式系统通过复制数据来提高系统的可靠性和容错性，并且将数据的不同的副本存放在不同的机器上，由于维护数据副本的一致性代价很高，因此许多系统采用弱一致性来提高性能，一些不同的一致性模型^[4]也相继被提出，主要有以下几种。

- **强一致性**：要求无论更新操作是在哪个数据副本上执行，之后所有的读操作都要能获得最新的数据。对于单副本数据来说，读写操作是在同一数据上执行的，容易保证强一致性。对多副本数据来说，则需要使用分布式事务协议（如两阶段提交或Paxos^[3]）。
- **弱一致性**：在这种一致性下，用户读到某一操作对系统特定数据的更新需要一段时间，我们将这段时间称为“不一致性窗口”。
- **最终一致性**：是弱一致性的一种特例，在这种一致性下系统保证用户最终能够读取到某操作对系统特定数据的更新（读取操作之前没有该数据的其他更新操作）。此种情况下，如果没有发生失败，“不一致性窗口”的大小依赖于交互延迟、系统的负载，以及复制技术中 replica 的个数（这个可以理解为 master/slave 模式中，slave 的个数）。DNS 系统可以说是在最终一致性方面最出名的系统，当更新一个域名的 IP 以后，根据配置策略以及缓存控制策略的不同，最终所有的客户都会看到最新的值。

最终一致性模型^[12]根据其提供的不同保证可以划分为更多的模型。

- **因果一致性**（Causal Consistency）：假如有相互独立的 A、B、C 三个进程对数据进行操作。进程 A 对某数据进行更新后并将该操作通知给 B，那么 B 接下来的读操作能够读取到 A 更新的数据值。但是由于 A 没有将该操作通知给 C，那么系统将不保证 C 一定能够读取到 A 更新的数据值。
- **读自写一致性**（Read Your Own Writes Consistency）：这个一致性是指用户更新某个数据后读取该数据时能够获取其更新后的值，而其他的用户读取该数据时则不能保证读取到最新值。
- **会话一致性**（Session Consistency）：是指读取自写更新一致性被限制在一个会话的范围内，也就是说提交更新操作的用户在同一个会话里读取该数据时能够保证数据是最新

的。

- **单调读一致性 (Monotonic Read Consistency)**: 是指用户读取某个数据值, 后续操作不会读取到该数据更早版本的值。
- **时间轴一致性 (Timeline Consistency)**: 要求数据的所有副本以相同的顺序执行所有的更新操作, 另一种说法叫单调写一致性 (Monotonic Write Consistency)。

系统选择哪种一致性模型取决于应用对一致性的需求, 所选取的一致性模型还会影响到系统如何处理用户的请求以及对副本维护技术的选择等。

2.3 ACID 与 BASE

正如CAP理论所指出的, 一致性、可用性和分区容错性不能同时满足。对于数据不断增长的系统 (如社会计算、网络服务的系统), 它们对可用性 & 分区容错性的要求高于强一致性, 并且很难满足事务所要求的ACID特性, 因此BASE理论⁵被提出。

BASE 方法通过牺牲一致性和孤立性来提高可用性和系统性能, 其中 BASE 分别代表:

- **基本可用 (Basically Available)**: 系统能够基本运行、一直提供服务。
- **软状态 (Soft-state)**: 系统不要求一直保持强一致状态。
- **最终一致性 (Eventual consistency)**: 系统需要在某一时刻后达到一致性要求。

事务是用户定义的一个数据库操作序列, 这些操作要么全不做, 要么全做, 是一个不可分割的工作单位, ACID^[5]是事务所具有的特性。

- **原子性 (Atomicity)**: 事务中的操作要么都做, 要么都不做。
- **一致性 (Consistency)**: 系统必须始终处在强一致状态下。
- **隔离性 (Isolation)**: 一个事务的执行不能被其他事务所干扰。
- **持续性 (Durability)**: 一个已提交的事务对数据库中数据的改变是永久性的。

保证 ACID 特性是传统关系型数据库中事务管理的重要任务, 也是恢复和并发控制的基本单位。

对于 ACID 和 BASE 的一些比较如表 2-2 所示。

5 在英文单词中, acid 指的是酸, 而 base 则有碱的意思, 因此取名 BASE 与 ACID 相对。

表 2-2 ACID 和 BASE 的比较

ACID	BASE
强一致性	弱一致性
隔离性	可用性优先
采用悲观、保守方法	采用乐观方法
难以变化	适应变化、更简单、更快

2.4 数据一致性实现技术

分布式存储在不同的节点的数据采取什么技术保证一致性，取决于应用对于系统一致性的需求，在关系型数据管理系统中一般会采用悲观的方法（如加锁），这些方法代价比较高，对系统性能也有较大影响，而在一些强调性能的系统中则会采用乐观的方法。

2.4.1 Quorum 系统 NRW 策略

对于数据不同副本中的一致性，采用类似于 Quorum 系统的一致性协议实现。这个协议有三个关键值 N 、 R 和 W 。

- N 表示数据所具有的副本数。
- R 表示完成读操作所需要读取的最小副本数，即一次读操作所需参与的最小节点数目。
- W 表示完成写操作所需要写入的最小副本数，即一次写操作所需要参与的最小节点数目。

该策略中，只需要保证 $R + W > N$ ，就可以保证强一致性。

例如： $N=3$ ， $W=2$ ， $R=2$ ，那么表示系统中数据有 3 个不同的副本，当进行写操作时，需要等待至少有 2 个副本完成了该写操作系统才会返回执行成功状态，对于读操作，系统有同样的特性。由于 $R + W > N$ ，因此该系统是可以保证强一致性的。

$R + W > N$ 会产生类似 Quorum 的效果。该模型中的读（写）延迟由最慢的 $R(W)$ 副本决定，有时为了获得较高的性能和较小的延迟， R 和 W 的和可能小于 N ，这时系统不能保证读操作能获取最新的数据。

如果 $R + W > N$ ，那么分布式系统就会提供强一致性的保证，因为读取数据的节点和被同步

写入的节点是有重叠的。在关系型数据管理系统中，如果 $N=2$ ，可以设置为 $W=2$ ， $R=1$ ，这是比较强的一致性约束，写操作的性能比较低，因为系统需要 2 个节点上的数据都完成更新后才将确认结果返回给用户。

如果 $R + W \leq N$ ，这时读取和写入操作是不重叠的，系统只能保证最终一致性，而副本达到一致的时间则依赖于系统异步更新的实现方式，不一致性的时间段也就等于从更新开始到所有的节点都异步完成更新之间的时间。

R 和 W 的设置直接影响系统的性能、扩展性与一致性。如果 W 设置为 1，则一个副本完成更改就可以返回给用户，然后通过异步的机制更新剩余的 $N-W$ 的副本；如果 R 设置为 1，只要有一个副本被读取就可以完成读操作， R 和 W 的值如较小会影响一致性，较大则会影响性能，因此对这两个值的设置需要权衡。

下面为不同设置的几种特殊情况。

- 当 $W = 1$ ， $R = N$ 时，系统对写操作有较高的要求，但读操作会比较慢，若 N 个节点中有节点发生故障，那么读操作将不能完成。
- 当 $R = 1$ ， $W = N$ 时，系统要求读操作高性能、高可用，但写操作性能较低，用于需要大量读操作的系统，若 N 个节点中有节点发生故障，那么写操作将无法完成。
- 当 $R = Q$ ， $W = Q$ ($Q = N/2 + 1$) 时，系统在读写性能之间取得了平衡，兼顾了性能和可用性，Dynamo 系统的默认设置就是这种，即 $N=3$ ， $W=2$ ， $R=2$ 。

2.4.2 两阶段提交协议

两阶段提交协议^[10]（Two Phase Commit Protocol，2PC 协议）可以保证数据的强一致性，许多分布式关系型数据管理系统采用此协议来完成分布式事务。它是协调所有分布式原子事务参与者，并决定提交或取消（回滚）的分布式算法，同时也是解决一致性问题的一致性算法。该算法能够解决很多的临时性系统故障（包括进程、网络节点、通信等故障），被广泛地使用。但是，它并不能通过配置来解决所有的故障。为了能够从故障中恢复，两阶段提交协议使用日志来记录参与者（节点）的状态，虽然使用日志降低了性能，但是参与者（节点）能够从故障中恢复。

在两阶段提交协议中，系统一般包含两类机器（或节点）：一类为协调者（Coordinator），通常一个系统中只有一个；另一类为事务参与者（Participants，Cohorts 或 Workers），一般包含多个，在数据存储系统中可以理解为数据副本的个数。协议中假设每个节点都会记录写前日志

(Write-ahead Log) 并持久性存储，即使节点发生故障日志也不会丢失。协议中还假设节点不会发生永久性故障，而且任意两个节点都可以互相通信。

当事务的最后一步完成之后，协调者执行协议，参与者根据本地事务是否成功完成来回复同意提交事务或者回滚事务。

顾名思义，两阶段提交协议由两个阶段组成。在正常的执行下，这两个阶段的执行过程如下所述。

阶段 1：请求阶段（commit-request phase，或称表决阶段，voting phase）

在请求阶段，协调者将通知事务参与者准备提交或取消事务，然后进入表决过程。在表决过程中，参与者将告知协调者自己的决策：同意（事务参与者本地作业执行成功）或取消（本地作业执行发生故障）。

阶段 2：提交阶段（commit phase）

在该阶段，协调者将基于第一个阶段的投票结果进行决策：提交或取消。当且仅当所有的参与者同意提交，事务协调者才通知所有的参与者提交事务，否则协调者将通知所有的参与者取消事务。参与者在接收到协调者发来的消息后将执行相应的操作。

注意 两阶段提交协议与两阶段锁协议不同，两阶段锁协议为一致性控制协议。

该协议的执行过程可以通过下图 2-2 来描述。

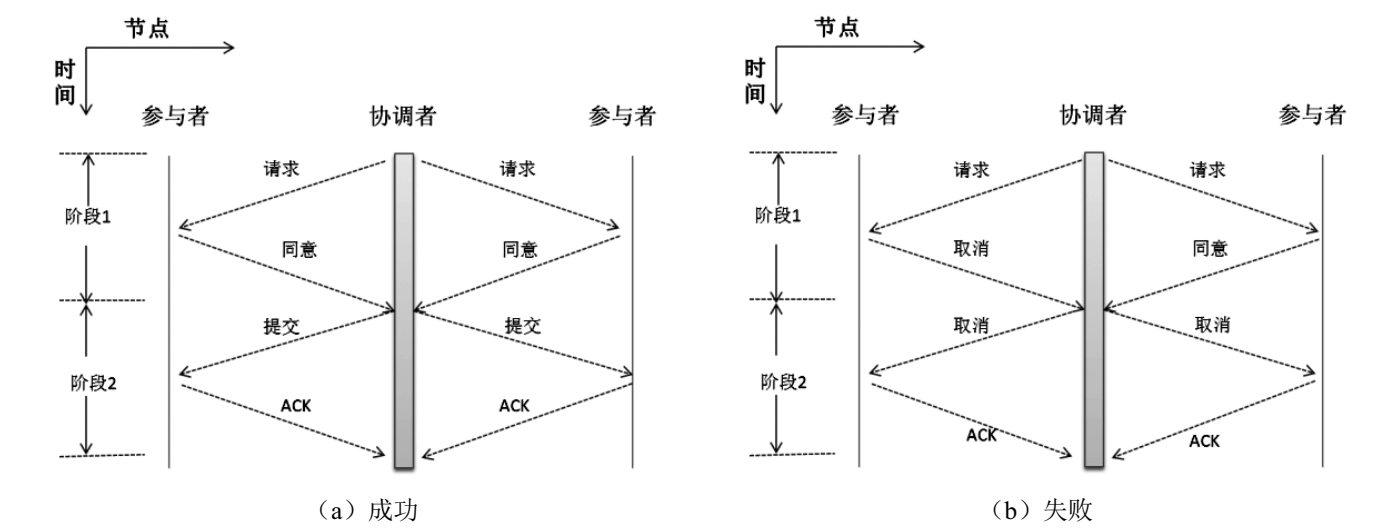


图 2-2 两阶段提交协议

两阶段提交协议最大的缺点在于它是通过阻塞完成的协议，节点在等待消息的时候处于阻塞状态，节点中其他进程则需要等待阻塞进程释放资源。如果协调者发生了故障，那么参与者

将无法完成事务而一直等待下去。以下情况可能会导致节点发生永久阻塞。

如果参与者发送同意提交消息给协调者，进程将阻塞直至收到协调者的提交或回滚的消息。如果协调者发生永久故障，参与者将一直等待，这里可以采用备份的协调者，所有参与者将回复发给备份协调者，由它承担原协调者的功能。

如果协调者发送“请求提交”消息给参与者，它将被阻塞直到所有参与者都回复完，如果某个参与者发生永久故障，那么协调者也不会一直阻塞，因为协调者在某一时间内还未收到某参与者的消息，那么它将通知其他参与者回滚事务。

同时两阶段提交协议没有容错机制，一个节点发生故障整个事务都要回滚，代价比较大。

下面我们通过一个例子来说明两阶段提交协议的工作过程。

A 组织 B、C 和 D 三个人去爬长城：如果所有人都同意去爬长城，那么活动将举行；如果有一人不同意去爬长城，那么活动将取消。用两阶段提交协议解决该问题的过程如下。

首先 A 将成为该活动的协调者，B、C 和 D 将成为该活动的参与者。

阶段 1

A 发邮件给 B、C 和 D，提出下周三去爬山，问是否同意，那么此时 A 需要等待 B、C 和 D 的邮件。

B、C 和 D 分别查看自己的日程安排表。B、C 发现自己在当日没有活动安排，则发邮件告诉 A 他们同意下周三去爬长城。由于某种原因，D 白天没有查看邮件。那么此时 A、B 和 C 均需要等待。到晚上的时候，D 发现了 A 的邮件，然后查看日程安排，发现周三当天已经有别的安排，因此 D 回复 A “活动取消”。

阶段 2

此时 A 收到了所有活动参与者的邮件，并且 A 发现 D 下周三不能去爬山，于是 A 发邮件通知 B、C 和 D，下周三爬长城活动取消。

此时 B、C 回复 A “太可惜了”，D 回复 A “不好意思”。至此该事务终止。

通过该例子可以发现，两阶段提交协议存在明显的问题。假如 D 一直不能回复邮件，那么 A、B 和 C 将不得不处于一直等待的状态。并且 B 和 C 所持有的资源一直不能释放，即下周三不能安排其他活动。其他等待该资源释放的活动也将不得不处于等待状态。

基于此，后来有人提出了三阶段提交协议，在其中引入超时的机制，将阶段 1 分解为两个

阶段：在超时发生以前，系统处于不确定阶段；在超时发生以后，系统则转入确定阶段。

两阶段提交协议包含协调者和参与者，并且二者都有出现问题的可能性。假如协调者出现问题，我们可以选出另一个协调者来提交事务。例如，班长组织活动，如果班长生病了，我们可以请副班长来组织。如果参与者出问题，那么事务将不会取消。例如，班级活动希望每个人都能参加，假如有一位同学不能参加了，那么直接取消活动即可。或者，如果大多数人参加，那么活动如期举行（两阶段提交协议变种）。为了能够更好地解决实际的问题，两阶段提交协议存在很多的变种，例如：树形两阶段提交协议（或称递归两阶段提交协议）、动态两阶段提交协议（D2PC）等。

2.4.3 时间戳策略

时间戳策略在关系数据库中有广泛的应用，该策略主要用于关系数据库日志系统中记录事务操作，以及数据恢复时的 Undo/Redo 等操作。在并行系统中，时间戳策略有更加广泛的应用。从较高的层次来说，时间戳策略可用于 SNA 架构或并行架构系统中时间及数据的同步。

在并行数据存储系统或并行数据库中，由于数据是分散存储在不同的节点上的，那么对于不同节点上的数据，如何区分它们的版本信息将成为比较烦琐的事情，该问题涉及不同节点之间的同步问题。若使用时间戳策略将能够很好地缓解这一境况，例如，我们可以为每一份或一组数据附加一个时间戳标记，在进行数据版本比较或数据同步的时候只需要比较其时间戳就可以区分它们的版本。但是分布式系统中不同节点之间的物理时钟可能会有偏差，这样就可能导致较早更新的数据其时间戳却比较晚。因此，我们设置一个全局时钟来进行时间同步。当一份数据更新之后，该数据所在节点向全局时钟请求一个时间戳。不过此时将出现新的问题：该全局时钟同步开销过大，影响系统效率；若全局时钟出现宕机，整个系统将无法工作。因此，该系统时钟将成为系统效率和可用性的瓶颈。

使用时间戳的策略将能够很好地解决这一问题。对时间戳策略进行改进，使其不依赖于任何单个的机器，也不依赖于物理时钟同步。该时间戳为逻辑上的时钟，并且通过时间戳版本的更新可以在系统中生成一个全局有序的逻辑关系。下面我们将简单介绍该策略的核心思想。

2.4.3.1 时间戳

时间戳最早用于分布式系统中进程之间的控制，用来确定分布式系统中事件的先后关系，可协调分布式系统中的资源控制。

假设发送或接受消息是进程中的一个事件，下面我们来定义分布式系统事件集中的先后关

系，用“ \rightarrow ”符号来表示，例如：若事件 a 发生在事件 b 之前，那么 $a \rightarrow b$ 。

该关系需要满足下列三个条件：

- 如果事件 a 和事件 b 是同一个进程中的事件，并且 a 在 b 之前发生，那么 $a \rightarrow b$ 。
- 如果事件 a 是某消息发送方进程中的事件，事件 b 是该消息接收方进程中接收该消息的事件，那么 $a \rightarrow b$ 。
- 对于事件 a 、事件 b 和事件 c ，如果有 $a \rightarrow b$ 和 $b \rightarrow c$ ，那么 $a \rightarrow c$ 。

如果两个不同的事件 a 和事件 b ，既不能得出 $a \rightarrow b$ 也不能得出 $b \rightarrow a$ ，那么事件 a 和事件 b 同时发生。

下面我们通过图 2-3 说明系统中可能存在的事件先后关系。在图 2-3 中，纵向代表事件轴，虚线表示进程之间的消息通信。在该模型中，如果存在着一个从 a 到 b 的时间或消息的先后关系，那么 $a \rightarrow b$ 。

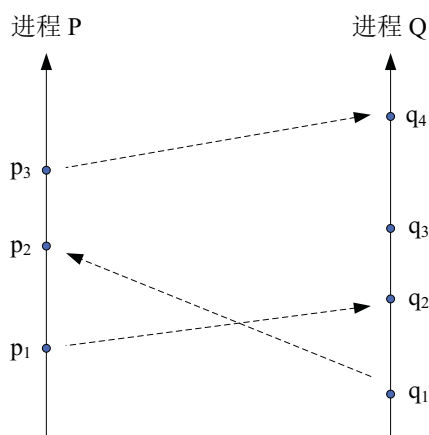


图 2-3 分布式系统多进程通信

例 1：在同一进程 P 中，事件 p_2 发生在事件 p_1 之后，因此 $p_1 \rightarrow p_2$ 。

例 2：对于事件 q_1 和事件 p_3 ，由于存在从 q_1 到 p_2 的消息传递，因此 $q_1 \rightarrow p_2$ ，同时，在同一进程 P 中，我们知道 $p_2 \rightarrow p_3$ ，因此根据该模型， $q_1 \rightarrow p_3$ 。

例 3：对于事件 p_3 和事件 q_3 ，在逻辑上，我们不能确定 p_3 是否在 q_3 之前发生（只能得出 p_3 在 q_1 之后发生），也不能确定 q_3 是否在 p_3 之前发生（只能得出 q_3 在 p_1 之后发生）。尽管在物理时间上， q_3 要先于 p_3 发生，但是我们不能确定这两个事件在该模型下的逻辑关系，因此我们说 p_3 和 q_3 同时发生。

2.4.3.2 逻辑时钟

现在将时钟引入到系统中。这里我们并不关心时钟值是如何产生的，它可以通过本地时钟产生，也可以为有序的数字。这里为每一个进程 P_i 定义一个时钟 C_i ，该时钟能够为任意一个事件 a 分配一个时钟值： $C_i(a)$ 。在全局上，同样存在一个时钟 C ，对于事件 b ，该时钟能够分配一个时钟值 $C(b)$ ，并且如果事件 b 发生在进程 P_i 上，那么 $C(b)=C_i(b)$ 。

我们的系统并不依赖于物理时钟，因为物理时钟可能会出现错误，因此我们要求：

时钟条件：如果对于事件 a 和事件 b ， $a \rightarrow b$ ，那么 $C(a) < C(b)$ 。

但是，事件 a 的逻辑时钟值小于事件 b 的逻辑时钟值并不意味着有 $a \rightarrow b$ ，因为可能事件 a 与事件 b 同时发生（见例 3）。

另外，在图 2-3 中可以得到 p_2 与 q_3 同时发生， p_3 与 q_3 也同时发生，那么这意味着 p_2 与 p_3 同时发生，但是这与实际情况相左，因为 $p_2 \rightarrow p_3$ 。因此，我们给出下面两个限制条件。

C1：如果事件 a 和事件 b 是同一个进程 P_i 中的事件，并且 a 在 b 之前发生，那么：

$$C_i(a) < C_i(b)$$

C2：如果 a 为进程 P_i 上某消息发送事件， b 为进程 P_j 上该消息接收事件，那么：

$$C_i(a) < C_j(b)$$

现在可以进一步考虑下“时钟走表”的概念，若事件 a 发生在事件 b 之前， $C(a) < C(b)$ ，例如 $C(a)=4$ ， $C(b)=7$ ，那么在事件 a 和事件 b 之间存在 4 到 5，5 到 6 和 6 到 7 三个时间间隔。也就是说存在先后顺序的事件之间一定至少存在一个时间间隔。那么 C1 意味着，同一个进程中的两个事件之间一定存在至少一个时间间隔；C2 意味着，每一条消息一定跨越了至少一个时间间隔。

根据以上规则，我们为存在事件间隔的事件或消息之间添加一条灰色的事件线来表示时间间隔的存在，那么图 2-3 可以转换为图 2-4，如下所示。

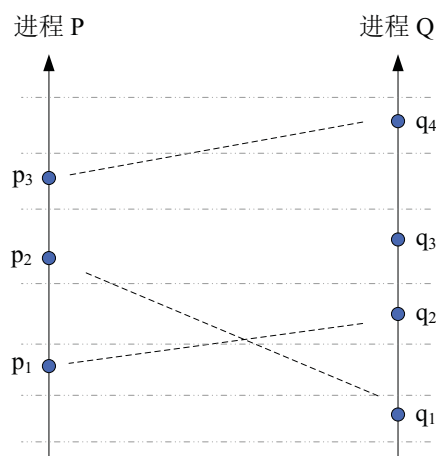


图 2-4 时间线

2.4.3.3 应用

假定进程为分布式存储系统或并行数据库系统中的不同节点，下面我们将时钟的概念引入到并行系统。在系统中，每一个节点 i 均包含一个时钟 C_i ，系统中包含两类事件，一种为节点上的数据更新；另一类为节点之间的消息通信（或数据同步）。

下面我们来说明该系统是如何满足 C1 和 C2 条件的。

对于 C1 条件来说，系统需要满足下面的实现规则。

IR1：对于同一节点上任意的连续事件来说，该节点上的时钟只需要保证较晚发生事件的时钟值大于较早发生事件的时钟值即可。

对于 C2 条件来说，当节点发送消息 m 时，该消息需要同时携带发送时刻在该节点产生的时间戳。在接收方收到消息 m 之后，接收方节点所产生的时间戳要大于 m 所携带的时间戳。但是仅仅如此还是不够的。

假设某节点 A 向节点 B 发送消息 m ，在发送消息时刻节点 A 的本地时间为 15:33:30，那么 m 所携带的时间戳可能为 $T_{m_a}=153330$ 。节点 B 在接收到 m 后，可能设置该事件的时间戳 T_{m_b} 为 153400，假如机器之间存在时间误差。比如，在 B 节点接收 m 时，B 节点设置 T_{m_b} 为 153400，但此时 B 节点系统时间为 15:50:05，这将会引起逻辑上的错误，因为在 153400 到 155005 之间可能有其他事件发生，这些事件本来早于接收消息 m 事件，却被错误地分配了更高的时间戳。

那么，为了满足 C2 约束，需要满足下面的规则。

IR2：(a) 如果事件 a 代表节点 N_i 发送消息 m ，那么消息 m 将携带时间戳 T_m ，且 $T_m=C_i(a)$ ；(b)

当节点 N_j 接收到消息 m 后，节点将设置该事件的时钟 C_j 大于或等于该节点上一事件的时钟并且大于或等于 T_m 。

该理论为时间戳策略的基本理论，具体的系统和实现要根据当前环境来决定。其中向量时钟技术为时间戳策略的演变，该技术能够更好地解决实际中的问题，详见 2.4.5 节。

2.4.4 Paxos

Paxos算法^[2]常用于具有较高容错性的分布式系统中，其核心就是一致性算法，该算法解决的问题是一个分布式系统如何就某个值（决议）达成一致。一个典型的场景是，在一个分布式数据库系统中，如果各节点的初始状态一致，每个节点都执行相同的操作序列，那么它们最后能得到一个一致的状态。为保证每个节点执行相同的命令序列，需要在每一条指令上执行一个“一致性算法”以保证每个节点看到的指令一致。一个通用的一致性算法应该可以应用多个场景中，这是分布式计算中要考虑的重要问题。目前，开源分布式系统Hadoop中的Zookeeper为Paxos算法的开源实现。

2.4.4.1 问题

假设许多节点都可以提交提议（value），一致性算法需要保证在所有被提出的提议中只有一个能够被接受，如果没有提议被提出，那么也不会有提议被选择，而如果某个提议一旦被选择，其他的节点能够知道所选择的提议。总之，一致性的保证需要满足以下条件：

- 提议只能被提出后才能被选择。
- 算法的一次执行实例中只能选择一个提议。
- 提议只有被选中后才能让其他节点（learner）所知道。

该算法的目的是保证某个提出的提议最终能够被选择，而且一旦被选中后，其他节点最终能够知道这个值。

在一致性算法中有三种角色：提议者（proposer）、批准者（acceptor）、学习者（learner）。提议者提出提议，批准者负责批准提议，而学习者主要学习提议。在具体实现中，一个节点可以担当多个角色。

假设节点之间通过发送消息进行通信，这里使用常用的异步、非拜占庭模型。在该模型中：

- 节点以任意的速度进行操作，可能因为故障而停止，也可以重新启动。并且节点所选择的提议不会因为重启等其他故障而消失。

- 消息可以延迟发送、多次发送或丢失，但不会被篡改（即不存在拜占庭问题）。

2.4.4.2 选择提议

选择提议的最简单的方法就是采用唯一的批准者，当提议者发送提议给批准者，批准者只需要选择他所接受到的第一个提议即可。这种方法虽然简单，但是批准者一旦发生故障，将无法进行之后的过程。因此，我们采用多个批准者的方法，此时选择提议的规则如下：

R：当提议被大多数批准者所批准，则表明该协议被批准。

提议者将提议发送给批准者，每个批准者都可以批准提议。如果一个批准者最多只能批准一个协议，那么将“大多数”设置为超过半数即可。这样，一定能够保证系统只有一个协议被批准，否则将与超过半数的约束相违背。

在不发生故障或消息丢失的情况下，假如只有某个提议者提出了一个提议，如果我们希望该提议能够被接受，这就要求：

P1：一个批准者必须批准它所收到的第一个提议。

这个要求会产生一个问题：在该约束下，同一时刻不同的提议者可能会发出不同的提议，那么可能会发生每一个批准者都批准了一份提议，但是却没有哪一个提议被大多数人同时批准。即使只有两个提议，如果每个提议恰好被半数的批准者所批准，这时就无法判定到底哪个提议才能被选择。

在一个系统中，一个批准者应该需要被设置为可以批准多个提议。以数据存储为例，一个分布式存储系统中不可能只存在一份数据。为了区分批准者所批准的提议，每个提议都分配有版本号，这样一个提议就包含了版本号（proposal number）和提议的值（value）。为了避免混淆，不同的提议拥有不同的版本号。因此，我们说某个提议被选定是指拥有该值的提议被大部分批准者所批准。

我们允许有多个提议被选中，但是必须保证这些提议拥有相同的内容，因此需要满足 P2 约束：

P2：如果某个提议被选定，那么之后选定的提议必须拥有相同的值。

由于版本号是有序且单调递增的，那么 P2 约束能够保证只有一个“值”被选定。由于提议被选定至少需要一个批准者批准，因此为了安全起见，我们可以将 P2 修改为：

P2^a：如果某个提议被选定，那么之后批准者只能批准拥有相同值的提议。

由于通信是异步的，因此某个特别的批准者可能会批准具有不同值的提议。例如一个新的提议者刚刚加入，并且提出了一个拥有高版本号但是不同值的提议。那么该提议可能会被一个从来没有接受过任何提议的批准者所批准（根据P1约束），而这将和P2^a约束相违背。

如果想要让系统同时满足P1和P2^a约束，那么P2^a需要修改为：

P2^b：如果某个提议被选定，那么之后任何提议者只能提出具有相同值的提议。

因为只有某个提议被批准之后新的提议才能被提出，约束P2^b隐含了约束P2^a，也隐含了P2。

假设一个版本号为 m 、值为 v 的提议已经被选中，我们必须保证任何版本号为 n （ $n > m$ ）的提议都拥有相同的值 v 。既然 m 已经被选中，显然最终会存在一个批准者的多数派 C ，他们都批准了 v 。考虑到任何多数派都和 C 具有至少一个公共成员，可以找到一个蕴含P2^b的约束：

P2^c：对于任意的 v 和 n ，如果带有值 v 、版本号为 n 的提议被某个提议者提出，那么在一个批准者的大多数集 S 中，要么没有批准者接收到版本号小于 n 的提议；要么 v 是 S 中版本号低于 n 的提议中具有最高版本号提议的值。

我们可以通过保持P2^c的不变性来满足P2^b。为了满足P2^c的不变性，若一个提议者想要发起版本号为 n 提议，那么他必须知道系统中已经被或将要被大多数集所批准的版本号小于 n 的提议。想要知道系统中已经被批准的提议是非常容易的事情，但是预测将来可能被批准的提议是非常困难的。因此在Paxos中，提议者要求批准者不会批准任何版本号低于 n 的提议。该解决方案可以用如下算法来描述，表示如何发起提议。

（1）提议者选择一个新的版本号 n ，并发送请求给一个批准者大多数集 S 中的每个成员。该请求要求：

- ① 每一个批准者承诺不会接受版本号低于 n 的提议。
- ② 如果批准者曾经批准过版本号低于 n 的提议，那么批准者需要将该信息发送给新的提议者。

我们将该请求称为版本号 n 的“准备请求”。

（2）如果提议者接收到来自大多数集 S 中所有批准者的请求响应，那么该提议者可以发起一个版本号为 n ，值为 v 的提议。这里 v 的值为所有响应中版本号最高的提议的值，或者如果新提议者没有接收到任何回应，那么它可以自行指定提议的值。

一个提议者通过向大多数集（该大多数集不一定要和“准备请求”所涉及的大多数集相同）

发送关于新提议的请求，我们将该请求称为“批准请求”。

上面是关于提议者的算法描述。那么批准者方面如何呢？批准者方面将接收到来自提议者的两类请求：准备请求和批准请求。在系统中，批准者可以忽略任何一种请求而不会给系统带来影响。如果批准者可以对请求做出响应，那么批准者可以对任意的“准备请求”做出响应；如果批准者没有许诺过不再批准提议，那么它将可以对任意的“批准请求”做出响应，并批准该提议。也就是说：

P1^a：批准者可以接受版本号为 n 的提议，当且仅当它没有响应过版本号大于 n 的“准备请求”时。

可以看到P1^a蕴含了P1 的内容。

现在已经有有了一个满足 Paxos 安全特性的完整算法，我们将对其进行优化而得出最终版本的算法。假定一个批准者接收到了版本号为 n 的“准备请求”，但是它已经响应了另一个版本号大于 n 的“准备请求”，那么它将不会批准该版本号为 n 的提议，因此它也不会对任何版本号为 n 的新提议做出响应，即它将忽略该“准备请求”。此外，它也不会对已经批准的提议的“准备请求”做出响应。

经优化后，一个批准者仅需要记录被它批准过的提议的最高版本号，以及曾响应过的“准备请求”所对应的最高版本号。因为P2^c必须保持不变性而不用考虑故障等问题，所以一个批准者即使发生故障重启也必须记录这些信息。注意：提议者可以随时终止自己所发起的提议并且不记录该信息，但是它不能再次发起具有与终止提议相同版本号的提议。

下面，我们将上述提议者与批准者两方面的操作合并起来讲述，该算法将包括如下两个阶段^[11]。

阶段 1：准备阶段

(1) 提议者选择一个恰当的版本号 n ，并发送一个版本号为 n 的“准备请求”到一个批准者的多数集。

(2) 如果某个批准者接收到该“准备请求”，并且该请求的版本号 n 大于该批准者之前所响应过的任意“准备请求”的版本号，那么此批准者将向该提议者承诺不会再响应任何版本号小于 n 的提议，并告知批准者它曾经批准过的提议的最高版本号。

阶段 2：批准阶段

(1) 如果提议者接收到来自大多数集的对于其关于版本号为 n 的“准备请求”的响应信息，

那么它向该（或其他）大多数集发送关于“版本号为 n 、值为 v 的提议”的“批准请求”，如果该响应信息不为空，其中 v 的值为该响应信息中最高版本号提议的值；如果该响应信息为空，那么提议者可以自主指定新的提议的值。

（2）如果某批准者接收到了关于版本号为 n 的提议的“批准请求”，那么除非它曾响应过版本号大于 n 的提议的“准备请求”，否则它将批准该提议。

一个提议者可以发起多个提议，只要它的每一步操作遵循该算法。一个提议者也可以在发起提议的中间任何时刻终止该操作（即便是关于某提议的请求或响应可能会在终止操作之后很长时间才到达，这也不会影响系统的正确性）。因此，如果某批准者接收到了更高版本的提议，那么它应该终止关于“旧”版本提议的请求，并通知该提议的提议者以便该提议者可以终止关于该提议的操作。

2.4.4.3 学习选择的提议

为了能够学习已被批准的提议，学习者需要找到被大多数集所批准的提议。比较直观的算法是：每当批准者批准了一个提议，该批准者就将此提议信息发送给每一个学习者。但是此种方法产生的通信量过大，会加重系统的负担。

在非拜占庭模型下，该问题比较容易解决。我们可以让批准者把批准的提议发送给不同于自己的学习者，然后该学习者又通知其他学习者这个信息。虽然问题能够解决，但是如果中间某个学习者出现故障，则之后的学习者将不能够收到该信息。

那么介于前面所提到的两个解决方案，我们可以令批准者将关于提议的信息发送给某几个不同的学习者，然后这些学习者再通知其他学习者该信息。

但是在某些特殊的情况下，使用上述方法学习者也可能收不到任何提议被批准的信息。是否可以令学习者向批准者询问是否有提议被批准呢？答案是否定的，因为即使不考虑故障的因素，并不是所有的批准者拥有最新的提议信息。那么如何让学习者能够主动获取到最新提议信息呢？我们可以让学习者充当一个提议者来发起一个新的提议。

2.4.4.4 其他问题

系统可能会出现这样的情况：有两个不同的提议者不断尝试提出更高版本的提议，并且任何一个都不会被选择为提议。例如，提议者 p 在第一阶段提出版本号为 n_1 的提议，另一个提议者 q 在第一阶段提出版本号为 n_2 的提议，并且 $n_2 > n_1$ 。在提议者 p 的第二个阶段，版本号为 n_1 的提议将被忽略，因为版本号更高的提议 n_2 被提出，提议者 p 可能尝试提出版本号为 n_3 的新提议，且 $n_3 > n_2$ 。那么这将反过来导致提议者 q 在第二阶段的失败，然后 q 又尝试提出更高版本的提议，如此周而

复始将一直不会有提议被批准。

为了保证程序的正常运行，系统需要选出某个提议者作为唯一的提议者来发出提议。如果该提议者能够和一个大多数集进行有效的通信，并且它所提出的提议版本号大于其他的提议，那么显然，这个提议将被批准。

2.4.4.5 实例

A、B、C、D、E 五个人计划下周去爬长城，但是还没有协商好具体时间。下面我们使用 Paxos 算法来选择出合适的时间。下面就可能发生的情况进行分析。

情况 1

A 提议爬长城时间定为下周三，并且无人与其竞争。

阶段 1：A 向所有人发出一个提议，该提议包含提议版本号和提议内容，我们将其格式定义为：[提议内容]：[提议版本号]。那么，该提议为“下周三：2012 年 3 月 2 日，12:00”。A 向所有人员发出该“准备请求”。B、C、D 和 E 在收到该“准备请求”之后向 A 回复同意该提议。

阶段 2：A 在收到 B、C、D 的回复之后，发现自己的提议已经被大多数人所接受。那么 A 向所有人广播该提议。其他人在收到该广播消息之后，该提议被正式批准。

结果：A 的提议被批准，爬山时间定为下周三。

情况 2

A 提议爬长城时间定为下周三，同时 E 提议爬长城时间定为下周五，并且 A、E 在同一时刻发起提议（即提议的版本号相同）。

由于某种原因，A 只能与 B、C 进行通信，而不能与 D、E 进行正常通信。同时，E 只能与 C、D 进行通信，而不能与 A、B 进行通信。

阶段 1：A 向所有人发出提议“下周三：2012 年 3 月 2 日，12:00”，同时 E 向所有人发出提议“下周五：2012 年 3 月 2 日，12:00”。B 在收到 A 发来的提议之后，由于还未接收到过任何提议，因此向 A 回复同意该提议；同样，D 也同意 E 的提议。那么最终提议版本将决定于 C 的选择。假如 A 的提议首先被 C 所接收，此时，C 还未收到任何提议，那么 A 的提议将被批准。当 E 的提议到达的时候，由于 C 已经批准了相同版本号的 A 提议，那么 C 将拒绝 E 的提议，并向 E 回复：“已经有相同版本号的提议被提出”。E 在接收到该回复之后将放弃自己的提议。

阶段 2: A在收到来自大多数集⁶ (B、C) 的批准回复之后, 将再次向某大多数集发送“批准请求”, 此时没有人与其竞争, 那么最终A的提议将被批准。

结果: A 的提议被批准, 爬山时间定为下周三。

情况 3

A 提议爬长城时间定为下周三, 同时 E 提议爬长城事件定为下周五, A、E 在不同时刻发起提议 (即提议的版本号不同)。

由于某种原因, A 只能与 B、C 进行通信, 而不能与 D、E 进行正常通信。同时, E 只能与 C、D 进行通信, 而不能与 A、B 进行通信。

阶段 1: A 向所有人发出提议“下周三: 2012 年 3 月 2 日, 12:00”, 半小时后 E 向所有人发出提议“下周五: 2012 年 3 月 2 日, 12:30”。与情况 2 类似, B、D 均能批准 A、E 的提议。那么关键在于 C 批准了谁的提议。假如 A 的提议优先到达 C 处, 那么 C 在看到 A 的提议之后将首先批准 A 的提议。之后, 在 C 看到 E 发来的提议之后, 发现 E 提议的版本要高于 A 发来的提议版本, 因此 C 向 E 回复同意该提议内容。

阶段 2: 此时 A 和 E 的提议同时得到了一个系统中大多数集的同意。下面 A 和 E 将同时能够进入批准阶段, 并向系统的某个大多数集发送“批准请求”。假定 A 和 E 所对应的大多数集与之前相同, 那么 C 在接收到发自 A 的批准请求时, 将会告诉 A 已经有更高版本的提议被 E 提出。那么 A 将取消自己的提议。

结果: E 的提议被批准, 爬山时间定为下周五。

分析: 假如在第一阶段 C 首先接收到了来自 E 的请求, 那么当 A 的请求到来时, C 将直接忽略 A 的请求, 并告知 A 已经有更高版本的提议被提出。此时, A 将不能进入批准阶段。

情况 4

A 提议爬长城时间定为下周三, 同时 E 提议爬长城事件定为下周五, 并且 A、E 在同一时刻发起提议 (即提议的版本号相同)。

由于某种原因 D 失去了联系, 并且 A、E 均能与 B、C 进行正常通信, 但 A、E 不能进行通信。

阶段 1: A 向所有人发出提议“下周三: 2012 年 3 月 2 日, 12:00”, 同时 E 向所有人发出

6 全集为 B、C 和 D。

提议“下周五：2012年3月2日，12:00”。假如B、C接收提议的顺序按如下序列进行，我们来分析最终的结果。

(1) B接收到来自A的提议，那么B批准A的提议，并回复A。

(2) C接收到来自E的提议，那么C批准E的提议，并回复E。

(3) B接收到来自E的提议，B发现该提议与之前批准的A提议版本号相同，将拒绝E的提议，并回复E已经有相同版本号的提议被提出。

(4) C接收到来自A的提议，C发现该提议与之前批准的E提议版本号相同，将拒绝A的提议，并回复A已经有相同版本号的提议被提出。

阶段2：由于A和E均没有收到一个大多数集的回复，那么A、E将均不能进入阶段2，此时没有任何版本的提议被批准。

结果：没有任何提议被批准。

情况5

A提议爬长城时间定为下周三，同时E提议爬长城活动定为下周五，A、E在不同时刻发起提议（即提议的版本号不同）。

由于某种原因D失去了联系，并且A、E均能与B、C进行正常通信，但A、E不能进行通信。

阶段1：A向所有人发出提议“下周三：2012年3月2日，12:00”，半小时后E向所有人发出提议“下周五：2012年3月2日，12:30”。假如B、C按照情况4所述的顺序处理来自A、E的提议，那么我们分析一下最终结果。

(1) B接收到来自A的提议，那么B批准A的提议，并回复A。

(2) C接收到来自E的提议，那么C批准E的提议，并回复E。

(3) B接收到来自E的提议，B发现该提议版本号比之前批准的A提议高，那么B将批准该版本的提议，并回复B。

(4) C接收到来自A的提议，C发现该提议比之前批准的E提议版本低，将拒绝A的提议，并回复A已经有更高版本的提议被提出。

阶段2：最终A将只收到来自B的回复，不能构成大多数集（A、B）；而E最终收到来自B、C的回复，从而构成大多数集（B、C、E），这样E将最终进入到“批准请求”阶段。最终

E 的提议将被批准。

结果：E 的提议被批准，爬山时间定为下周五。

从上述 5 种情况可以看出，该算法还存在一定的缺陷。因此，正如 2.4.4.4 节所述，为了保证程序的正常运行，系统需要能够选出某个提议者作为唯一的提议者来发出提议。这样能够有效避免情况 4 的出现。在 ZooKeeper 中使用的正是该种策略，ZooKeeper 将这种策略称为“领导选取（Leader Election）”。

2.4.5 向量时钟

向量时钟（Vector Clock）^[8,9]是一种在分布式环境中为各种操作或事件产生偏序值的技术，它可以检测操作或事件的并行冲突，用来保持系统的一致性。

向量时钟方法在分布式系统中用于保证操作的有序性和数据的一致性。向量时钟通常可以被认为是一组来自不同节点的时钟值 $V_i[1]$ 、 $V_i[2]$ 、 \dots 、 $V_i[n]$ 。在分布式环境中，第 i 个节点维护某一数据的时钟时，根据这些值可以知道其他节点或副本的状态，例如 $V_i[0]$ 是第 i 个节点所了解的第 0 个节点上的时钟值，而 $V_i[n]$ 是第 i 个节点所了解的第 n 个节点上的时钟值。时钟值代表了节点上数据的版本信息，该值可以是来自节点本地时间的时间戳或者是根据某一规则生成有序数字。

以 3 副本系统为例，该系统包含节点 0、节点 1 和节点 2。某一时刻的状态可由表 2-3 来表示。

表 2-3 向量时钟实例

	V_0	V_1	V_2
V_0	4	2	0
V_1	1	4	0
V_2	0	0	1

该表表示当前时刻各节点的向量时钟为：

节点 0： $V_0(4,2,0)$

节点 1： $V_1(1,4,0)$

节点 2： $V_2(0,0,1)$

在表 2-3 中， V_i 代表第 i 个节点上的时钟信息， $V_i[j]$ 表示第 i 个节点所了解的第 j 个节点的时钟

信息。以第 2 行为例，该行为 V_1 节点的向量时钟 (1,4,0)，其中“1”表示 V_1 节点所了解的 V_0 节点上的时钟值；“0”表示 V_1 节点所了解的 V_2 节点上的时钟值；“4”表示 V_1 节点自身所维护的时钟值。

下面具体描述向量时钟在分布式系统中的运维规则。

规则 1：

初始时，我们将每个节点的值设置为 0。每当有数据更新发生，该节点所维护的时钟值将增长一定的步数 d ， d 的值通常由系统提前设置好。

该规则表明，如果操作 a 在操作 b 之前完成，那么 a 的向量时钟值大于 b 向量时钟值。

向量时钟根据以下两个规则进行更新。

规则 2：

在节点 i 的数据更新之前，我们对节点 i 所维护的向量 V_i 进行更新：

$$V_i[i] = V_i[i] + d \quad (d > 0)$$

该规则表明，当 $V_i[i]$ 处理事件时，其所维护的向量时钟对应的自身数据版本的时钟值将进行更新。

规则 3：

当节点 i 向节点 j 发送更新消息时，将携带自身所了解的其他节点的向量时钟信息。节点 j 将根据接收到的向量与自身所了解的向量时钟信息进行比对，然后进行更新：

$$V_j[k] = \max\{V_i[k], V_j[k]\}$$

在合并时，节点 j 的向量时钟每一维的值取节点 i 与节点 j 向量时钟该维度值的较大者。

两个向量时钟是否存在偏序关系，通过以下规则进行比较：

对于 n 维向量来说， $V_i > V_j$ ，如果任意 k ($0 \leq k \leq n-1$) 均有 $V_i[k] > V_j[k]$ 。

如果 V_i 既不大于 V_j 且 V_j 也不大于 V_i ，这说明在并行操作中发生了冲突，这时需要采用冲突解决方法进行处理，比如合并。

如上所示，向量时钟主要用来解决不同副本更新操作所产生的数据一致性问题，副本并不保留客户的向量时钟，但客户有时需要保存所交互数据的向量时钟。如在单调读一致性模型中，用户需要保存上次读取到的数据的向量时钟，下次读取到的数据所维护的向量时钟则要求比上一个向量时钟大（即比较新的数据）。

相对于其他方法，向量时钟的主要优势在于：

- 节点之间不需要同步时钟，即不需要全局时钟。
- 不需要在所有节点上存储、维护一段数据的版本数。

下面我们通过一个例子来体会向量时钟如何维护数据版本的一致性。

A、B、C、D 四个人计划下周去爬长城。A 首先提议周三去，此时 B 给 D 发邮件建议周四去，他俩通过邮件联系后决定周四去比较好。之后 C 与 D 通电话后决定周二去。然后，A 询问 B、C、D 三人是否同意周三去，C 回复说已经商量好了周二去，而 B 则回复已经决定周四去，D 又联系不上，这时 A 得到不同的回复。如果他们决定以最新的决定为准，而 A、B、C 没有记录商量的时间，因此无法确定什么时候去爬长城。

下面我们使用向量时钟来“保证数据的一致性”：为每个决定附带一个向量时钟值，并通过时钟值的更新来维护数据的版本。在本例中我们设置步长 d 的值为 1，初始值为 0。

(1) 在初始状态下，将四个人（四个节点）根据规则 1 将自身所维护的向量时钟清零，如下所示：

A(0,0,0,0)

B(0,0,0,0)

C(0,0,0,0)

D(0,0,0,0)

(2) A 提议周三出去

A 首先根据规则 2 对自身所维护的时钟值进行更新，同时将该向量时钟发往其他节点。B、C、D 节点在接收到 A 所发来的时钟向量后发现它们所知晓的 A 节点向量时钟版本已经过时，因此同样进行更新。更新后的向量时钟状态如下所示：

A(1,0,0,0)

B(1,0,0,0)

C(1,0,0,0)

D(1,0,0,0)

(3) B 和 D 通过邮件进行协商

B 觉得周四去比较好，那么此时 B 首先根据规则 2 更新向量时钟版本 (B(1,1,0,0))，然后将向量时钟信息发送给 D (D(1,0,0,0))。D 通过与 B 进行版本比对，发现 B 的数据较新，那么

D 根据规则 3 对向量时钟更新，如下所示。

A(1,0,0,0)

B(1,1,0,0)

C(1,0,0,0)

D(1,1,0,0)

(4) C 和 D 进行电话协商

C 觉得周二去比较好，那么此时 C 首先更新自身向量时钟版本 (C(1, 0, 1, 0))，然后打电话通知 D (D(1, 1, 0, 0))。D 根据规则 3 对向量时钟进行更新。

此时系统的向量时钟如下所示：

A(1,0,0,0)

B(1,1,0,0)

C(1,0,1,0)

D(1,1,1,0)

最终，通过对各个节点的向量时钟进行比对，发现 D 的向量时钟与其他节点相比具有偏序关系。因此该系统将决定“周二”一起去爬山。

下面我们用图示来描述上述过程，如图 2-5 所示。

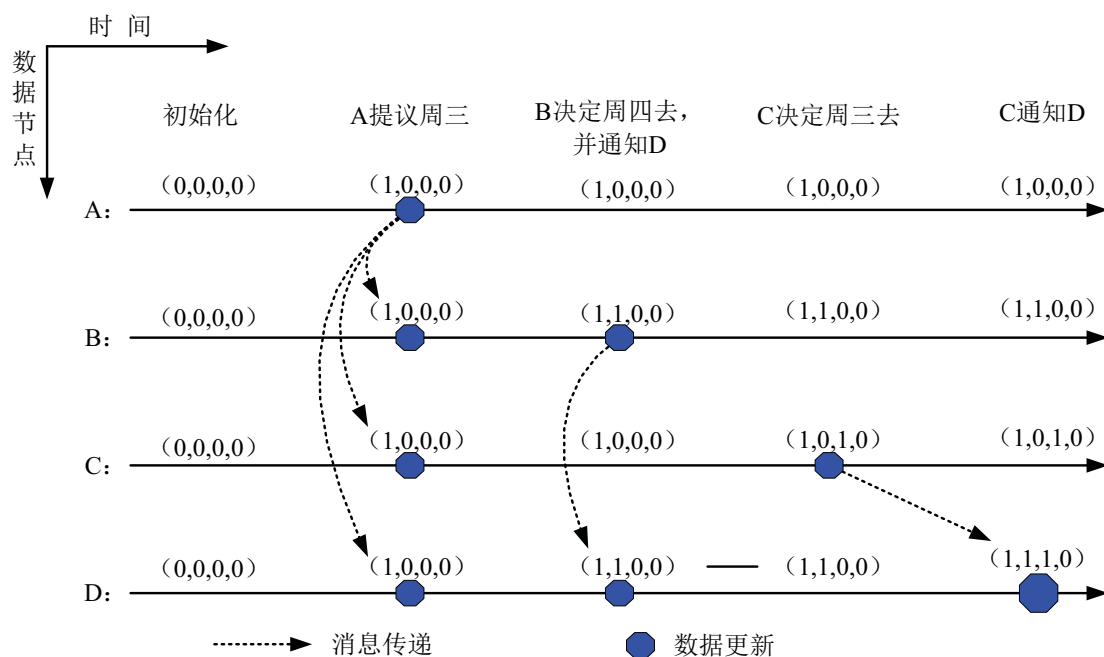


图 2-5 向量时钟

该方法中数据版本可能出现冲突，即不能确定向量时钟的偏序关系。如图 2-5 所示，假如 C 在决定周三爬山后并没有将该决定告诉其他人，那么系统在此刻将不能确定某一数据向量时钟的绝对偏序关系。比较简单的冲突解决方案是随机选择一个数据的版本返回给用户。而在 Dynamo 中系统将数据的不一致性冲突交给客户端来解决。当用户查询某一数据的最新版本时，若发生数据冲突，系统将把所有版本的数据返回给客户端，交由客户端进行处理。

该方法的主要缺点就是向量时钟值的大小与参与的用户有关，在分布式系统中参与的用户很多，随着时间的推移，向量时钟值会增长到很大。一些系统中为向量时钟记录时间戳，某一时间根据记录的时间对向量时钟值进行裁剪，删除最早记录的字段。

向量时钟在实现中有两个主要问题：如何确定持有向量时钟值的用户，如何防止向量时钟值随着时间不断增长。

2.5 小结

本章介绍了关于海量数据存储以及 NoSQL 数据库中的数据一致性理论。CAP 理论为 NoSQL 数据管理系统的基石，该理论告诉我们强一致性、可用性和分区容错性不能同时满足。在进行系统设计的时候必须在这三者之间做出权衡，需根据不同的应用和环境进行系统设计。

为了提高系统的效率，在大多数的系统中采用的是“最终一致性策略”，而放弃了 CAP 理论中的“强一致性”要求。BASE 模型正是这一方面应用的典型理论代表，该方法通过牺牲一致性和孤立性来提高可用性和系统性能，其中 BASE 分别代表：基本可用、软状态和最终一致性。

在数据一致性的最终实现上，不同的系统采用不同的策略，包括：Quorum 的 NWR 策略、两阶段提交协议、Paxos、时间戳、向量时钟等，本章只列举了其中的一部分，现实中还有更多的实现。但是这些系统或者模型均以 CAP 理论为基石，并依据不同的情况作出权衡，例如 Paxos 具有较强的一致性，但是系统延迟较大。此外，很多系统中采用多种策略的结合，例如，NWR 策略经常与向量时钟一同使用，用以解决数据的一致性问题的。

参考文献

- [1] Eric Brewer. Towards Robust Distributed Systems [R]. Keynote at the ACM Symposium on Principles of Distributed Computing (PODC) on 2000.
- [2] Leslie Lamport: Paxos Made Simple, Fast, and Byzantine [C]. OPODIS 2002:7-9.
- [3] Leslie Lamport: The Part-Time Parliament [J]. 1998, 16(2):133-169. ACM Trans. Comput. Syst. (TOCS), 1998.
- [4] Todd Lipcon. Design Patterns for Distributed Non-relational Databases [R]. 2009.
- [5] 王珊，萨师煊. 数据库系统概论 [M]. 北京：高等教育出版社，2007.
- [6] Julian Browne. Brewer's CAP Theorem [EB/OL]. 2009-01-11[2012-06-20].
<http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>.
- [7] Guy Pardon. A CAP Solution (Proving Brewer Wrong) [EB/OL]. 2008-09-07[2012-06-20].
<http://guysblogspot.blogspot.com/2008/09/cap-solution-proving-brewer-wrong.html>.
- [8] Basho. Why Vector Clock are Easy [EB/OL]. 2010-01-29[2012-06-20].

<http://basho.com/blog/technical/2010/01/29/why-vector-clocks-are-easy/>.

[9] Basho. Why Vector Clock are Hard [EB/OL]. 2010-04-05[2012-06-20].

<http://basho.com/blog/technical/2010/04/05/why-vector-clocks-are-hard/>.

[10] http://en.wikipedia.org/wiki/Two-phase_commit_protocol

[11] http://en.wikipedia.org/wiki/Paxos_algorithm

[12] Werner Vogels. Eventually Consistent - Revisited [EB/OL]. 2008-12-22[2012-06-20].

http://www.allthingsdistributed.com/2008/12/eventually_consistent.html.

第 11 章

文档数据库

本章导读

本章分别从安装、基本概念、查询方法、分布式环境下的使用等几个方面介绍 CouchDB 和 MongoDB 两款典型的开源文档数据库。这两款数据库有许多相同的地方，比如，所存储的数据格式都是 JSON 数据类型，都使用 JavaScript 进行操作，都支持 Map/Reduce 等等。但仔细比较它们也有很多不同的地方，比如：CouchDB 使用 MVCC 机制，而 MongoDB 使用 Update-in-one-place 机制；CouchDB 仅支持静态查询，而 MongoDB 除支持静态查询外还支持动态查询等。希望大家在实践中仔细体会两者的异同。

本章要点

- CouchDB 简介
- CouchDB 安装
- Map/Reduce 查询
- 分布式环境下的 CouchDB
- MongoDB 简介
- MongoDB 安装
- MongoDB 索引
- SQL 与 MongoDB
- MapReduce 与 MongoDB

11.1 文档数据库简介

1989 年 IBM 通过其 Lotus 群件产品 Notes 提出了数据库技术的全新概念——“文档数据库”，文档数据库有别于传统数据库，它是一种用来管理文档的数据库。在传统的数据库中，信息被分割成离散的数据段，而在文档数据库中，文档是信息处理的基本单位。一个文档可以很长、很复杂，也可以很短，甚至可以无结构。

文档数据库与 20 世纪五六十年代的文件系统不同，它仍属于数据库范畴。首先，文件系统文件基本上对应于某个应用程序。即使不同的应用程序所需要的数据部分相同，也必须建立各自的文件，而不能共享数据，而文档数据库可以共享相同的数据。因此，文件系统比文档数据库的数据冗余度更大，更浪费存储空间，且更难于管理维护。其次，文件系统中的文件是为某一特定应用服务的，所以，要想对现有的数据再增加一些新的应用是很困难的，系统不容易扩充，数据和程序缺乏独立性。而文档数据库具有数据的物理独立性和逻辑独立性，数据和程序分离。

文档数据库也不同于关系数据库，关系数据库是高度结构化的，而文档数据库允许创建许多不同类型的非结构化的或任意格式的字段。它与关系数据库的主要不同在于，它不提供对数据完整性的支持，但它和关系数据库也不是相互排斥的，它们之间可以相互交换数据，从而相互补充、扩展。

下面，我们选取文档数据库的两个典型代表 CouchDB 和 MongoDB 进行详细介绍。这两个数据库都是开源数据库，感兴趣的读者可以下载使用，加深对它们的了解。

11.2 CouchDB 数据库

11.2.1 CouchDB 简介

CouchDB 是用 Erlang 开发的文档数据库系统，是 NoSQL 数据库中的重要代表。CouchDB 的全称是 Cluster Of Unreliable Commodity Hardware Database，从其中可看出 CouchDB 的目标是具有高度可伸缩性，提供高可用性和高可靠性，即使运行在容易出现故障的硬件上也是如此。CouchDB 还是一个比较年轻的数据库，发布的 0.10.0 版本至今仅 3 年多的时间。

在 2005 年 4 月，IBM 的工程师 Damien Katz 在自己的博客上发表了一个自己正在开发的数

数据库引擎,称为 CouchDB。当时,他的目标是开发出一款 Internet 数据库(database of the Internet),即专门面向 Web 应用程序设计的数据库。最初 Katz 使用 C++作为他的平台语言。从一开始 Katz 就把 CouchDB 设计为一个无模式的具有索引的数据库引擎,同时受长期编写 Lotus Notes 程序的影响,他选择使用只增长模式(Append-only)来设计数据库,这就意味着 CouchDB 中的数据将永远不会被重写,它们只是在数据库中变成旧的版本而已,用户所看到的只是最新的数据。

在 2006 年 2 月,该项目有了里程碑式的发展,CouchDB 的底层开发语言从 C++变成了 Erlang。Erlang 是一种通用的面向并发的编程语言,它由瑞典电信设备制造商爱立信所辖的 CS-Lab 开发,目的是创造一种可以应对大规模并发活动的编程语言和运行环境。该语言在电信领域应用得比较广泛,具有支持超大量级的进程并发、高容错性等优点,很适合用于分布式程序的开发。所以,Katz 认为这一定是最适合 CouchDB 的语言。

另一个较大突破是 2006 年的 4 月,Katz 宣布 CouchDB 可以通过基于 HTTP 的 RESTful 的 API 来进行访问。也就是说,用户可以通过任何一个可以访问 HTTP 服务器的软件来对数据库进行访问。

2006 年 8 月,CouchDB 0.2 发布。2007 年 8 月 Damien Katz 宣布他决定使用 JSON 替代 XML 作为数据的存储格式,并选择 JavaScript 作为数据查询引擎,这一改变是 CouchDB 发展史上的重要的一步,正是这一改变使得 CouchDB 引来了更多人的关注。

2007 年 11 月,0.7.0 版本发布,较之于以往版本,它具有很多新的特性。现在的 CouchDB 使用基于 Mozilla Spidermonkey 的 JavaScript 视图引擎和一个嵌入浏览器的管理界面。

2008 年 2 月,CouchDB 开始支持 Map/Reduce 查询操作,同时,这个项目成为 Apache 基金会的培育项目。2008 年 11 月该项目成为和 Apache HTTP Server、Tomcat 和 Ant 一样的顶级项目。2009 年 4 月,CouchDB 0.9.0 发布,2009 年 10 月,第一个 Beta 版本 CouchDB 0.10.0 发布。当前最新的版本为 1.1.1。

CouchDB 提出的口号是:下一代的 Web 应用存储系统。CouchDB 的设计在很大程度上借鉴了 Web 的架构、资源、方法、表示形式等许多概念,在此基础上它还提供了强大的数据查询、映射、过滤功能,以及良好的容错性、高可扩展性和数据库备份功能,可以说 CouchDB 是一个文档数据库的优秀范例。Django 的开发者 Jacob Kaplan-Moss 在谈到 CouchDB 时这样说道:“Django 也许是为 Web 的设计建立的,而 CouchDB 则应该说是 Web 的一部分。因为我从来没有见过任何一个软件像 CouchDB 一样几乎完全采用了 HTTP 的设计理念。CouchDB 的出现使得 Django 像是旧式学校所教授的东西,这就像当年 Django 的出现使得 ASP 很快就过时了一样。”^[1]不仅如此,由于 CouchDB 数据库支持数据本地存储的特性和对分布式环境的良好适应能力,使其在手机等数

据终端的应用和云计算环境的应用中都有着良好的应用前景。

11.2.2 CouchDB 安装

CouchDB 属于开源软件，可以安装在 Windows 系统和其他系统中。下面分别以 Windows 和 Ubuntu 环境为例，来讲解 CouchDB 的安装方法。CouchDB 当前的最新版本为 1.1.1，我们在 Ubuntu 中安装 0.10.0 和 1.0.0 版，在 Windows 中安装 1.1.1 版。

1. 在 Windows 中安装 CouchDB

Windows 环境下的软件安装比较简单，从网站 http://wiki.apache.org/couchdb/Installing_on_Windows 下载 `setup-couchdb-1.1.1_js185_otp_R14B03+fix-win32-crypto.exe` 安装文件，直接双击安装即可。

安装好后，在 Firefox 浏览器中输入：

```
http://127.0.0.1:5984/_utils
```

通过 Futon 运行 CouchDB 和相应的测试用例（test suite），检查安装是否正确。注意，只能在 Firefox 中进行验证。

2. 在 Linux 中安装 CouchDB

从 Ubuntu 9.10 开始 Ubuntu 系统中就集成了 CouchDB，在 Ubuntu 中安装已经集成的 CouchDB 比较方便，要安装非集成版本则相对来讲比较麻烦。下面介绍在 Ubuntu 10.04 中安装自带版本的 CouchDB 和 CouchDB 1.0.0。

（1）安装自带版本

在终端中输入以下命令：

```
sudo aptitude install couchdb
```

则会自动安装版本为 0.10.0 的 CouchDB。对于其他版本的 Ubuntu 系统，使用相同的命令可以安装对应版本的 CouchDB。

（2）安装 CouchDB 1.0.0

从网站 <http://couchdb.apache.org/downloads.html> 下载得到相应版本的 CouchDB。

输入如下命令解压：

```
tar -zxvf apache-couchdb-1.0.0.tar.gz
```

进入目录：

```
cd apache-couchdb-1.0.0
```

查看当前的 XULRunner 的版本：

```
XULRunner -V
```

接下来配置安装文件（示例系统中的 XULRunner 版本为 1.9.2.26）：

```
./configure --with-js-include=/usr/lib/xulrunner-devel-1.9.2.26/include --with-js-lib=/usr/lib/xulrunner-devel-1.9.2.26/lib
```

若没有配置 XULRunner，安装过程中会出错，这时应该按照以下步骤操作：

```
sudo vi /etc/ld.so.conf.d/xulrunner.conf
/usr/lib/xulrunner-1.9.2.26
/usr/lib/xulrunner-devel-1.9.2.26
sudo /sbin/ldconfig
```

执行此步骤后执行下面的语句则可安装成功。

```
make && sudo make install
```

11.2.3 CouchDB 入门

11.2.3.1 数据格式

CouchDB 的数据是自包含的。先看一个实例：生活中一张发票包含了所有与消费相关的信息，包括交易双方的名称、交易日期、交易金额、结算方式等等，即交易过程中的数据由一张发票来进行记录和处理，那么对于这个发票来说，其数据即自包含的^[1]。同样的道理，CouchDB 将功能实现程序和与功能相关的数据记录在一个数据库中进行管理。

在 CouchDB 中数据的存储格式为 JSON（JavaScript Object Notation）格式。JSON 是一种轻量级数据交换格式。它基于 JavaScript 的一个子集，并采用完全独立于语言的文本格式。JSON 可以将 JavaScript 对象表示的数据转换为字符串的形式，所以 JSON 常常作为参数在函数之间传递，此外，在 Ajax 程序中数据也会以 JSON 格式从客户端传递给服务器端。由于这种格式很灵活而且容易在 JavaScript 中使用，所以很适合用来做 Web 应用。

JSON 的数据类型和结构主要有数值型、字符串型、布尔型、数组和对象等，其一般的表示方式如表 11-1 所示。

表 11-1 JSON 各类型值示例

类 型	示 例	示 例	示 例
数值	"count"3	"score":-15	"height":13.21
字符串	"database":"CouchDB"	"context": "can I help you? "	
布尔	"flag": "false"	"flag":"true"	
数组	"运动":["篮球","足球","游泳"]	[{"运动":["篮球", "足球", "游泳"]},{ "height":1.80}, {"context": "can I help you? "}]	
对象	{ "姓名": "李刚", "性别": "男", "生日": "1990-01-12" }		
空值	"Email":null		

CouchDB 中的每一个文档都以 JSON 格式存放在数据库中，下面是一个简单文档的示例：

```
{
  "_id": "7dcb49716a7200925ac170e35c0074d3",
  "_rev": "1-4ec0d58e0ec8ee45ece0cecb01c069ee",
  "title": "本章讲的是文档数据库",
  "article": "CouchDB 是一个文档数据库",
  "time": "Tue Mar 13 2012 11:00:30"
}
```

11.2.3.2 RESTful API

CouchDB 是一个 RESTful 的数据库，其操作完全通过 HTTP 协议完成，我们可以通过 CouchDB 提供的简单的 API 访问它。首先使用 curl 命令和 CouchDB 提供的 API 来与 CouchDB 进行交互。curl 命令可以控制原始的 HTTP 请求，而且我们能很清楚地看到数据库到底做了什么。

为确保 CouchDB 仍然在运行，可以输入如下命令：

```
curl http://127.0.0.1:5984/
```

这个语句发送了一个 GET 命令到新安装的 CouchDB 中，如果 CouchDB 在运行，你会看到这样的回复：

```
{"couchdb":"Welcome","version":"1.0.0"}
```

接着，通过下面的这个命令，可以得到当前 CouchDB 中所有数据库的列表：

```
curl -X GET http://127.0.0.1:5984/_all_dbs
```

这时命令行的回复为:

```
[ ]
```

因为我们还没有在 CouchDB 中创建任何数据库, 所以得到的是一个空列表。

那么, 我们来创建一个数据库, 命名为 `nosqldatabase`, 注意 CouchDB 的数据库命名要求全部小写:

```
curl -X PUT http://127.0.0.1:5984/nosqldatabase
```

创建成功! 可以看到 CouchDB 的回应为:

```
{"ok":true}
```

此时, 查看数据库列表, 可以得到如下结果:

```
curl -X GET http://127.0.0.1:5984/_all_dbs  
["nosqldatabase"]
```

这说明我们已经成功创建了 `nosqldatabase` 数据库。接着, 创建另一个数据库:

```
curl -X PUT http://127.0.0.1:5984/nosqldatabase
```

由于相同名字的数据库已经存在, 所以会得到 CouchDB 返回的错误信息:

```
{"error":"file_exists","reason":"The database could not be created, the file  
already exists."}
```

我们重新创建一个名为 `docdatabase` 数据库:

```
curl -X PUT http://127.0.0.1:5984/docdatabase  
{"ok":true}
```

再次查看数据库列表, 可以得到如下的信息:

```
["nosqldatabase","docdatabase"]
```

现在我们可以使用 `DELETE` 来删除数据库:

```
curl -X DELETE http://127.0.0.1:5984/docdatabase
```

成功后得到如下信息:

```
{"ok":true}
```

这时再查看数据库列表, 则会发现只剩下一个数据库了。

```
curl -X GET http://127.0.0.1:5984/_all_dbs
```

```
["nosqldatabase"]
```

本节介绍了利用 `curl` 命令和原始的 HTTP API 来访问数据库，但是在实际过程中，更多的情况是使用 `Futon` 这个工具来访问，因此下面将着重介绍 `Futon`。

11.2.3.3 Futon

`Futon` 是一个可以通过浏览器来访问 `CouchDB` 的管理接口。它提供了丰富的方法来访问 `CouchDB` 的各种相关属性，使得我们更方便地处理一些复杂的工作。通过 `Futon`，可以创建和删除数据库；浏览和编辑数据库中的文档；编写和运行 `Map/Reduce` 方法；触发复制器，在两个数据库之间复制文件等。

在 `Firefox` 浏览器中输入 `http://127.0.0.1:5984/_utils/`，可以进入 `Futon` 的界面，如图 11-1 所示。可以看到我们新建的数据库 `nosqldatabase` 已经存在于数据库列表中。



图 11-1 进入 Futon 后的初始界面

一般情况下，安装 `CouchDB` 后，需要做的第一件事情就是在 `CouchDB` 上运行测试样例来验证 `CouchDB` 所有的组件是否都能正常工作。单击图 11-1 所示界面右下方的“`Test Suite`”选项，单击“`Run All`”按钮，图 11-2 显示了 `Futon` 正在运行一些测试样例。由于这些测试样例是通过浏览器来运行的，所以在测试样例运行的过程中，不仅可以检测到 `CouchDB` 的运行是否正确，还可以检测浏览器和数据库之间的连接是否正确地配置了。

在 `Futon` 中创建数据库和文档都是十分简单的。在 `Futon` 的 `Overview` 页面中，单击“`Create Database`”选项，并在弹出的框中输入数据库的名称，就可以创建一个数据库了。以创建“`firstcouchhdb`”数据库为例，过程如图 11-3 所示。

创建数据库后，`Futon` 将呈现出该数据库中的所有文档。由于文档还没有创建，所以目前文档列表为空，单击“`Create Document`”可以创建一个文档。创建文档时务必使 `document ID` 的

那一栏为空，因为当保存文档的时候，CouchDB 会为该文档自动生成一个 UUID（Universal Unique Identifier）。该 UUID 可以唯一地标识该文档。

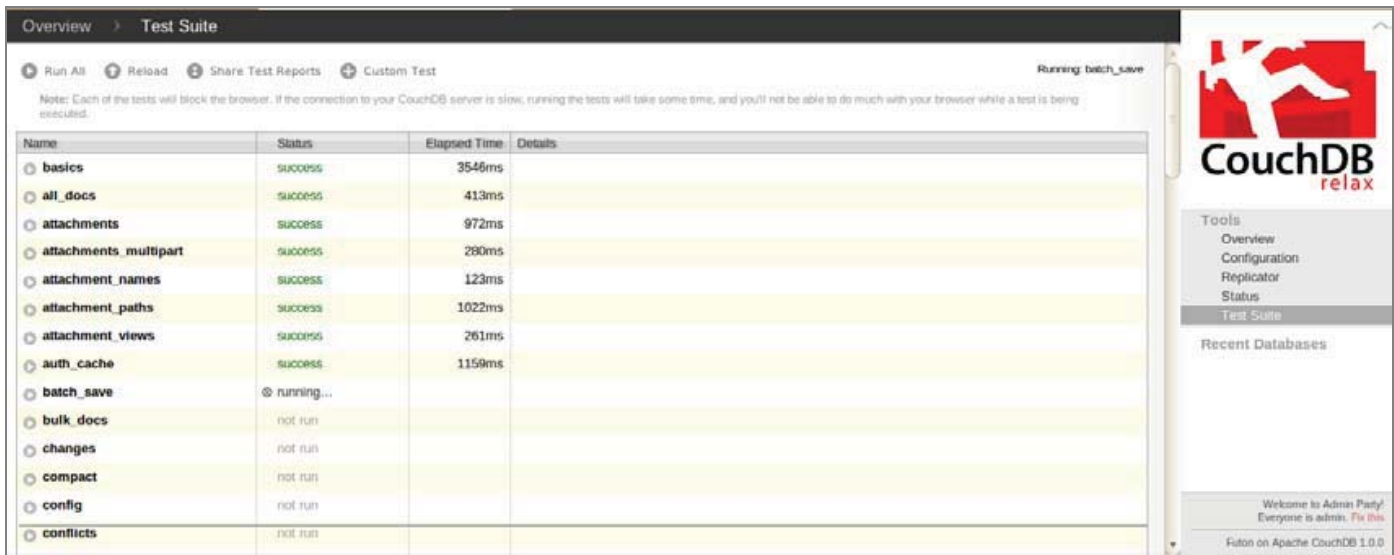


图 11-2 测试过程的检测

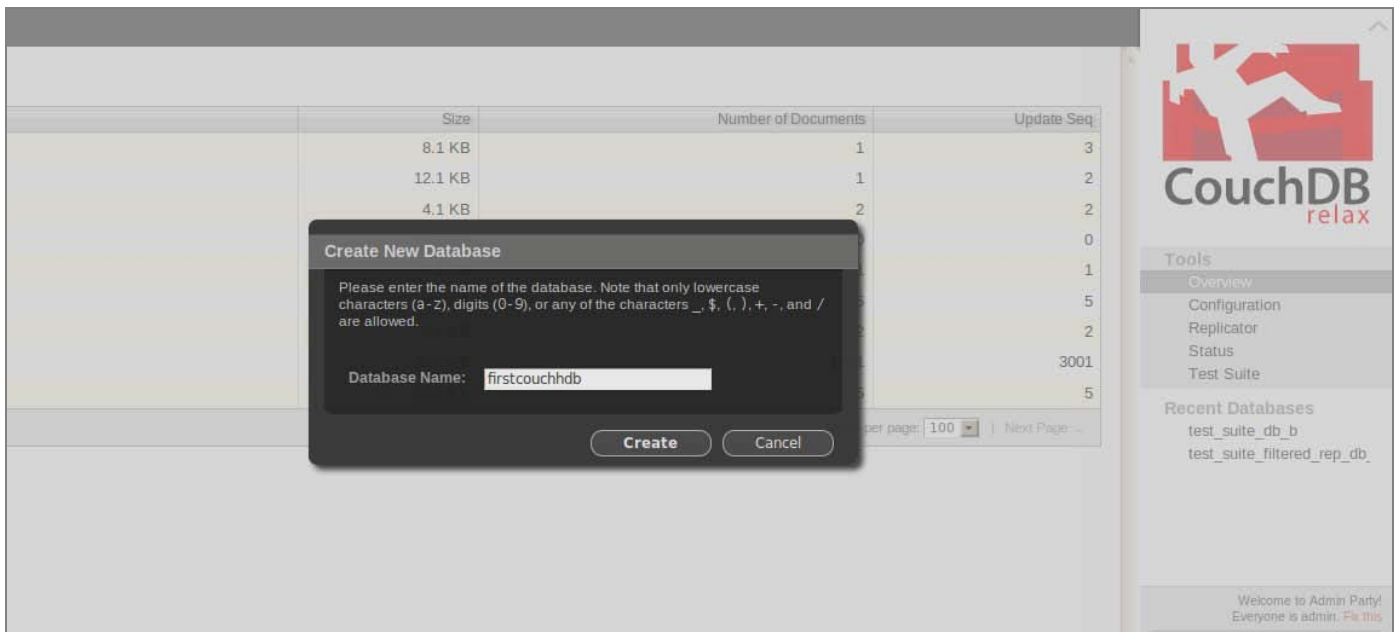


图 11-3 创建第一个数据库

建立一个新的文档后，可以看到文档中包含两个主要的部分，分别为域（field）和值（value）。其中仅有两个 field，分别为_id（标识）和_rev（版本），单击“Add Field”可以添加新的域，如图 11-4 所示。当为一个域添加值的时候，需要注意所添加的值必须是一种合法的 JSON 数据的格式，例如"world", [1,2, "c"], {"foo": "bar"}等。添加了合适的值之后，单击“Save Document”来保存当前的文档。留意一下你一定会发现，保存前后文档的_rev 发生了变化，即文档的版本

进行了更新，关于一个文档的 `_rev` 值和相关的属性我们将在后续的章节中详细介绍。

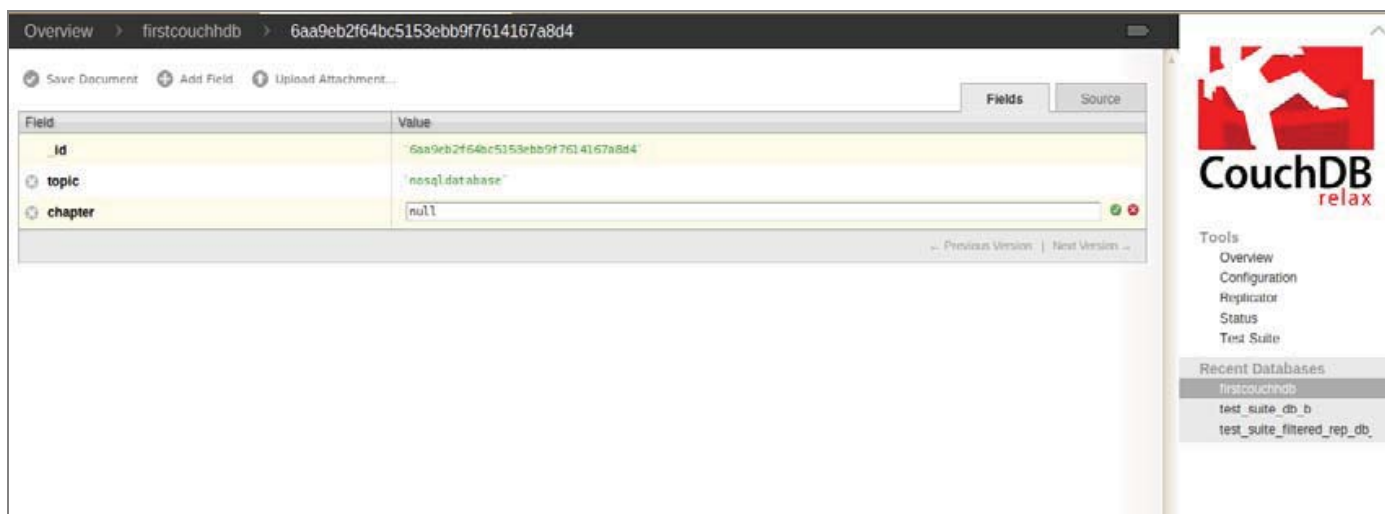


图 11-4 添加新的域

此时已经成功为文档添加一个新的域，将 Fields 切换至 Source 标签可以看到“firstcouchdb”文档的 JSON 格式。

文档是 CouchDB 的核心数据结构。其设计原型来自于日常生活中的实际文档，比如一张发票、一个处方单据等。前面提到了 CouchDB 使用 JSON 格式的数据来存储文档，下面我们来看一下文档的工作方式。

11.2.3.4 文档

为了帮助读者更清楚地理解文档的基本概念，本节的部分内容采用了 curl 命令，希望读者可以一边阅读，一边实践。在 CouchDB 中，每一个文档都有一个 ID，这个 ID 对于每一个数据库来说都是唯一的，CouchDB 对于文档的 ID 没有特别地强调，也就是说任何一个字符串都可以用作文档的 ID，但是我们还是推荐使用系统自动生成的全局唯一的 ID，也就是 UUID (Universally Unique Identifier)。UUID 是系统产生的随机字符串，使用 UUID 作为文档的 ID 不可能重复。

下面我们通过建立一个文档的实例来了解文档的 ID。在命令行终端中输入以下命令：

```
curl -X GET http://127.0.0.1:5984/_uuids
```

该命令会产生一个 UUID：

```
{"uuids":["6aa9eb2f64bc5153ebb9f7614167af6a"]}
```

当然，如果需要的 UUID 的数量较多，可以使用以下命令产生 N 个 UUID：

```
curl -X GET http://127.0.0.1:5984/_uuids?count=N
```

接下来我们可以利用得到的 UUID 和下面的命令来创建一个文档：

```
curl -X PUT http://127.0.0.1:5984/firstcouchhdb/ 6aa9eb2f64bc5153ebb9f7614167af6a -d '{"title":"A docdatabase","name":"CouchDB"}
```

该命令中-X PUT 告诉 curl 产生一个 PUT 请求命令，后面的 127.0.0.1:5984 是请求的地址；firstcouchhdb/6aa9eb2f64bc5153ebb9f7614167af6a 则是请求的具体数据库的具体文档，其中的那一串随机的数字是请求文档的 UUID；-d 告诉 curl 使用后面的数据作为 PUT 请求的文档的内容，可以看到文档的内容为 JSON 数据格式。

CouchDB 会做如下响应：

```
{
  "ok":true,
  "id":"6aa9eb2f64bc5153ebb9f7614167af6a",
  "rev":"1-1ce8db7a9462cc4c349c76080f9f9e73"
}
```

我们可以利用 GET 命令来查看刚刚创建的文档：

```
curl -X GET http://127.0.0.1:5984/firstcouchhdb/ 6aa9eb2f64bc5153ebb9f7614167af6a
```

可以看到 CouchDB 做出的回应为：

```
{
  "_id":"6aa9eb2f64bc5153ebb9f7614167af6a",
  "_rev":"1-1ce8db7a9462cc4c349c76080f9f9e73",
  "title":"A docdatabase",
  "name":"CouchDB"
}
```

从这个命令可以看出，CouchDB 中的所有元素都有一个地址、一个 URI (Uniform Resource Identifier)，并且所有的元素都可以通过 HTTP 的方法访问。

同时，从上面得到的文档的内容中可以看到，除了我们自己添加的两个部分之外，文档中还增加了_id 和_rev 两个部分，其中_id 的内容就是我们添加的 UUID，而_rev 部分则是我们马上要介绍的内容——文档的版本。

文档的版本信息记录在文档的_rev 中，当用户试图更新或者删除一个文档的时候，CouchDB 要求用户必须在_rev 区域中包含想要更新或删除的文档的版本。当 CouchDB 接受了文本的改变之后会为该文档产生一个新的版本号。这种机制保证了如果在用户修改该文档之前已经有其他

用户在该用户不知道的情况下对该文档进行了修改，CouchDB 将不会保存这种修改，因为这样会导致用户重写自己不知道的数据。下面来看一下在修改文档的时候没有提供文档的版本号会出现什么现象：

```
curl -X PUT http://127.0.0.1:5984/firstcouchhdb/ 6aa9eb2f64bc5153ebb9f7614167af6a -d '{"title":"A docdatabase","name":"CouchDB","type":"nosqldatabase"}'
```

这时 CouchDB 会出现这样的提示信息：

```
{"error":"conflict","reason":"Document update conflict."}
```

如果看到这样的回应，则应该在更新语句中添加当前最新的文档的版本号：

```
curl -X PUT http://127.0.0.1:5984/firstcouchhdb/ 6aa9eb2f64bc5153ebb9f7614167af6a -d '{"_rev":"1-1ce8db7a9462cc4c349c76080f9f9e73","title":"A docdatabase","name":"CouchDB","type":"nosqldatabase"}'
```

此时，你将会看到文档的修改成功，并可以看到相应的版本号已经做了修改和更新：

```
{"ok":true,"id":"6aa9eb2f64bc5153ebb9f7614167af6a","rev":"2-c0a490f60f95ced99ef7a5b0b826e285"}
```

文档的版本号是一个前缀为“N-”的经过 MD5 加密的序列，其中 N 表示该文档被更新的次数。CouchDB 的这种版本控制机制被称为 MVCC（Multi-Version Concurrency Control，多版本并发控制机制）。

采用这种机制有两个原因。一方面 CouchDB 所使用的 HTTP 协议是无状态的，也就是说当需要和 CouchDB 进行会话时，整个过程包括三个步骤：打开一个到 CouchDB 的网络连接，交换数据，关闭网络连接。这一点与其他协议不同，有的协议允许用户在打开一个连接交换一部分数据之后保持连接，以便继续交换更多的数据，最后再将数据连接关闭。通常来说，为后续的数据交换长时间的保持连接会增加服务器的负担，所以一个比较普遍的解决方法就是，在连接打开期间，使用户得到一个和服务器上的数据一致的静态视图，通过视图对数据进行访问和操作。因为处理大规模的并行连接工作量很大，而且 HTTP 连接保持的时间通常很短，所以 CouchDB 采用了这种模式来处理更多的并发连接。

另一个原因是 MVCC 模型从概念上讲要相对简单一些，而且比较容易在程序中实现，所以 CouchDB 使用了较少的代码来做这份工作。

下面来介绍一下文档的附件（Attachment）。CouchDB 的文档中的附件与 E-mail 中的附件类似。CouchDB 中的附件由它的名称、其所包含数据的 MIME 类型（或者 Content-Type）以及表示附件大小的字节数等几个部分组成。附件内容可以是任何形式的数据，包括文本、图片、

Word 文档、音乐、电影等等。

附件在文档上传的时候会获得自己的 URL，文档的上传一般通过 Futon 进行，整个过程就像上传邮件附件一样，如图 11-5 所示。

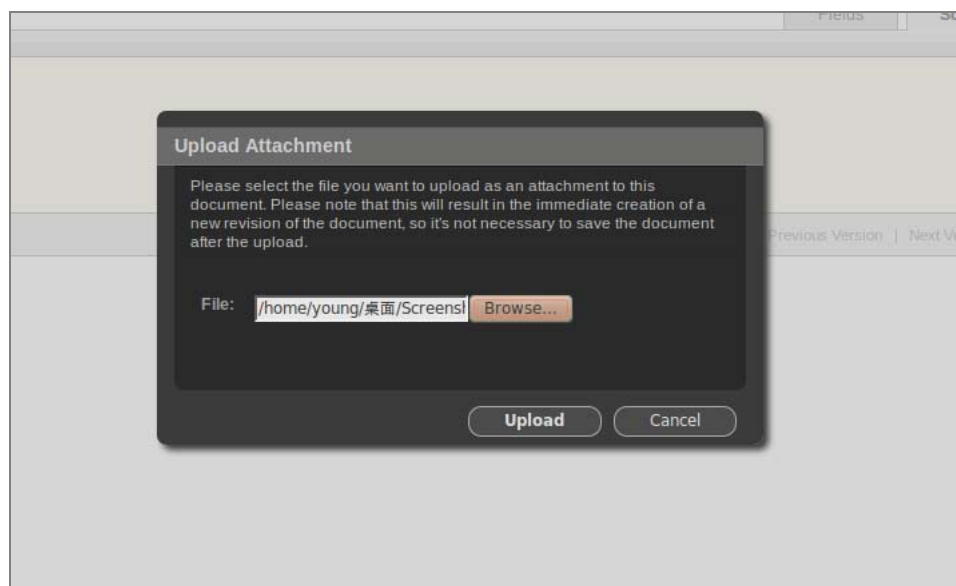


图 11-5 利用 Futon 上传文档

11.2.3.5 Design 文档

Design 文档是 id 以 “_design/” 开头的文档，它和 CouchDB 中的其他文档一样可以随着数据库的复制而复制，可以随着文档的修改而更新版本，但是它还有着与一般文档所不同的特点。一个 Design 文档一般情况下由 _id、_rev、views、shows、list、_attachments、validate_doc_update 等几个区域组成，下面来具体介绍一下各主要部分。

1. views

对于一个 CouchDB 的应用程序来说，CouchDB 会在 Design 文档中寻找 views 和其他的应用功能，程序所用到的静态 HTML 文件都会以附件的形式存放在 Design 文档中，而 views 和 validate_doc_update 并不是以附件的形式存放的，它们直接被包含在 Design 文档的 JSON 结构中。

CouchDB 的 MapReduce 查询存储在 Design 文档的 views 区域中，这也就是为什么我们可以通过 Futon 来直接编辑 MapReduce 查询。CouchDB 会根据每一个函数的内容信息将 views 的索引存储在每一个 Design 文档的底部，当用户在编辑文档的 _attachments、validate_doc_update 等区域的时候，views 将不会更新。然而，如果用户改变了一个 Map 或者 Reduce 函数，views

索引将会更新，旧的 views 索引将被删除，而新的 views 索引将会按照新的函数重新产生。

2. show 和 list

CouchDB 能使用原始的 JSON 数据响应请求。在 Design 文档中 show 和 list 这两个区域包含那些可以将原始的 JSON 数据转换为 HTML 和 XML 或者其他的数据类型的函数。这就使得 CouchDB 可以直接为应用程序提供原子数据而不需要其他任何的中间件。show 和 list 中的函数有点像传统 Web 框架中的“action”，这些“action”基于一个请求和响应代码来运行。然而，与“action”不同的是，这些函数不能为系统带来副作用，这就是说它们在很大程度上被限制在操作 GET 请求上，但这也意味着它们可以被类似于 Varnish 之类的 HTTP 代理所存储。

由于应用逻辑存储在一个单独的文档中，文档的更新和升级可以利用 CouchDB 的同步机制来完成。但是，我们也可以不使用 show 和 list 中的函数来进行应用系统的开发。在每一个 CouchDB 中都使用了 JavaScript 对 CouchDB 的 API 进行了外层的包装，可以通过 JavaScript 编写 Ajax 程序来访问数据库开发应用程序，并且使用 Ajax 查询来访问数据库可以更清楚地看到 CouchDB 的 JSON/HTTP API 是怎样工作的，所以在 JavaScript 中进行 CouchDB 应用程序的开发，会使得程序中的对象和数据库中的关系模式之间的转换变得更容易，而且浏览器的 XMLHttpRequest 对象会为开发人员处理 HTTP 请求中间过程的相关细节，使开发人员将更多的精力放在应用程序的设计和开发上。

3. validate_doc_update

CouchDB 使用 validate_doc_update 函数来阻止系统中进行的无效的或者没有经过认证的文档的更新，其在应用程序中也可以用来阻止非登录用户对程序的修改。CouchDB 的 validate_doc_update 函数不可以有任何的副作用，因为它们的功能很强大，既可以阻止终端用户对文档的保存，又可以在分布式的环境中阻止文档从一个节点复制到另外一个节点。

4. _attachments

与普通的文档一样，Design 文档同样可以有自己的附件，在附件中可以存储程序中用到的 JavaScript、CSS、HTML 文件等，这样对程序资源的管理更方便。

11.2.3.6 Replication

CouchDB 的 Replication 不仅可以用来复制数据库做数据库备份，还可以像 rsync 一样使两个在本地或者远程的数据库保持数据同步。通过一个简单的 POST 请求告诉 CouchDB 所执行复制命令的源数据库和目标数据库，CouchDB 将会根据源和目标数据库确定哪些文档存在于源数

数据库中而不在目标数据库中，此时 **Replication** 会把这些文档从源数据库复制到目标数据库，从而保证两个数据库中的数据一致。

由于 **CouchDB** 不会为用户自动创建一个目标数据库，所以在复制数据之前，一定要新建一个目标数据库。

```
curl -X PUT http://127.0.0.1:5984/firstcouchhdb-replica
```

建立数据库后，就可以复制数据库了：

```
curl -vX POST http://127.0.0.1:5984/_replicate -d '{"source":"firstcouchhdb",
"target":"firstcouchhdb-replica"}'
```

复制后可以得到这样的响应：

```
{
  "ok": true,
  "source_last_seq": 10,
  "session_id": "470d23651d0b64da8c8071c10db72b37",
  "history": [
    {
      {"session_id":"470d23651d0b64da8c8071c10db72b37",
      "start_time":"Wed, 21 Mar 2012 02:11:18 GMT",
      "end_time":"Wed, 21 Mar 2012 02:11:18 GMT",
      "start_last_seq":0,"end_last_seq":4,
      "recorded_seq":4,
      "missing_checked":0,
      "missing_found":2,
      "docs_read":2,
      "docs_written":2,
      "doc_write_failures":0}
    }
  ]
}
```

此时可看到目标数据库中的数据已经成功更新。上面的这段 JSON 格式的语句是 **CouchDB** 留存的同步会话历史记录。其中主要的内容为：

- "ok": true 表示整个同步的过程进行得很好，没有出现错误。
- 每一个复制过程都由一个 `session_id` 唯一标识，这个 `session_id` 也是一个 UUID。
- `start_time` 和 `end_time` 记录了整个同步进程的开始时间和结束时间。
- `missing_checked` 表示在同步的过程中目标数据库已经存在的不用再复制的文档的个数。
- `missing_found` 表示源数据库不存在的文件的个数。

- docs_read、docs_written、doc_write_failures 分别表示从源数据库读出的文档数目、写入目标数据库的文档数目、同步失败的文档数目，当 docs_read 和 docs_written 数目相等而且 doc_write_failures 的数目为 0 的时候表示同步成功。

在 Futon 中该操作如图 11-6 所示。



图 11-6 同步数据库的操作

上面是对本地数据库进行的同步操作，这种同步操作对数据库的备份、保存数据库快照十分有用。但是 CouchDB 的 Replication 还支持远程的数据同步，这时目标数据库和源数据库的名称都会以 HTML 的连接的形式出现：

```
curl -vX POST http://127.0.0.1:5984/_replicate -d '{"source":"http://127.0.0.1:5984/firstcouchhdb","target":"http://127.0.0.1:5984/firstcouchhdb-replica"}'
```

通过该语句可以将本地的数据库备份到远程的数据库。

11.2.4 CouchDB 查询

一般的关系数据库只要数据的结构是正确的，就允许运行任何的查询操作，同样的，CouchDB 使用事先定义好的 Map 和 Reduce 函数以 Map/Reduce 的方式来进行查询。这些函数为用户提供了极大的灵活性，由于每一个文档都可以单独、并行地进行计算，所以它们可以适应不同的文档结构和索引^[2]。CouchDB 将一个 Map 函数和一个 Reduce 函数结合在一起称之为视图。

CouchDB 中的视图是用来获得开发者或当前程序想要的数据集的方式，而且是一种静态的数据查询方式^[3]。视图很有用，在筛选数据库中的文档，为某个特殊的程序找到相关的文档；按照一定的顺序从数据库的文档中提取数据；按照某种需要的值为文档建立相应的索引等情况下都可以使用。总之，CouchDB 中的视图所做的事情与关系数据库中的 SQL 查询所做的事情有许多类似之处，是开发应用程序，浏览数据库中的文档必不可少的工具。

在 CouchDB 中一般将视图称为 MapReduce 视图，一个 MapReduce 视图由两个 JavaScript 函数

组成，一个是Map函数，这个函数必须定义；另一个是Reduce函数，这个是可选的，根据程序的需要可以进行选择性定义^[3]。

详细介绍 CouchDB 的 Map 和 Reduce 执行过程之前，先了解一下 Futon 为我们提供的临时视图（Temporary View）。临时视图是 Futon 中的一个可以对视图进行调试的界面（见图 11-7），没有问题的函数可以保存在 Design 文档中形成固定的视图。为了了解 Map 和 Reduce 的过程，我们以建立一个数据库 groups 为例来进行讲解，如图 11-8 所示。

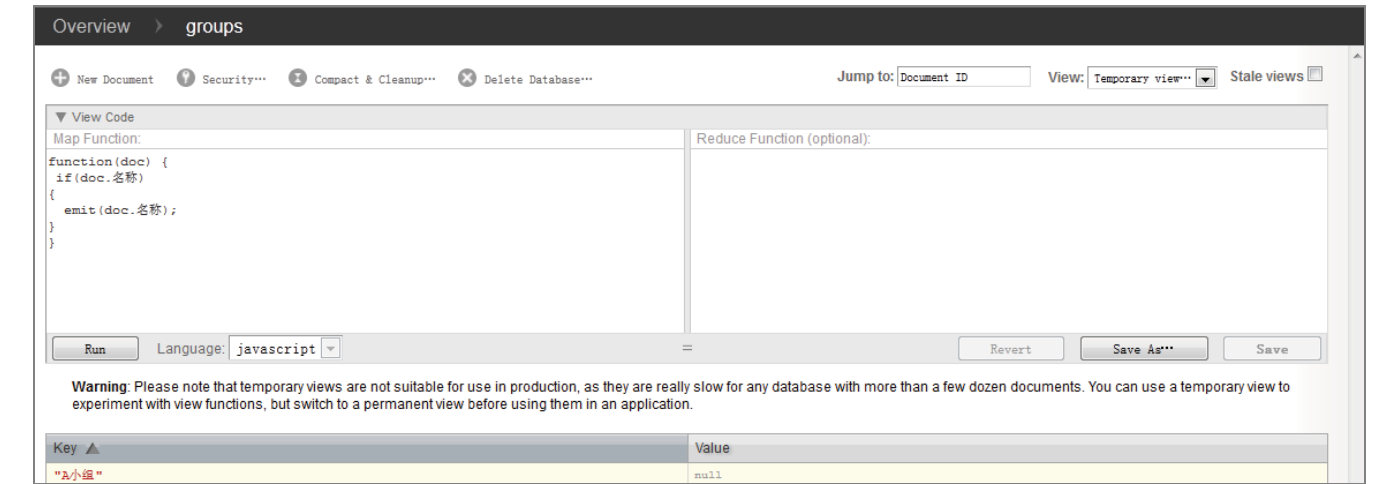


图 11-7 临时视图运行界面

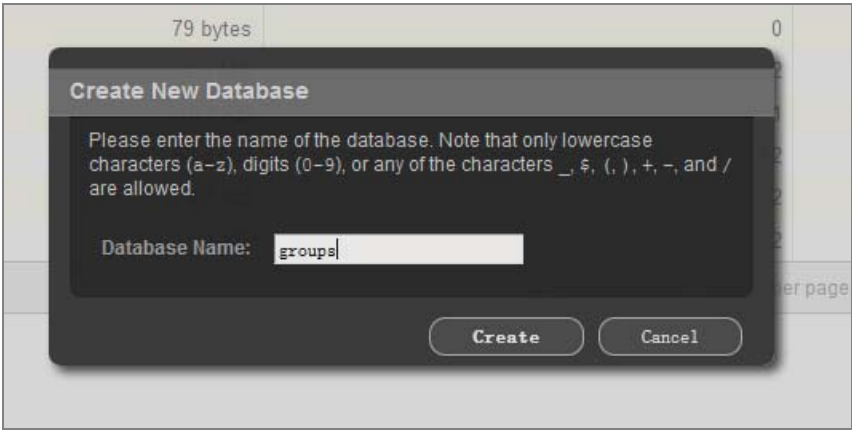


图 11-8 新建一个数据库 groups

在 groups 中我们添加三个文档作为示例，文档的内容分别如下：

```
{  "_id": "4c1168fca1d0ad9f69a1267b86000ed5",
  "_rev": "1-283727dba699785809c18abc7eaedfcc",
  "名称": "A 小组",
  "成员": [
    "李刚",
```



```

        "刘伟",
        "赵小云"
    ],
    "平均年龄": 23,
    "所获荣誉": [
        "卫生先进小组",
        "文体先进小组",
        "学习先进小组"
    ]
}

```

```

{
    "_id": "4c1168fca1d0ad9f69a1267b86001c23",
    "_rev": "2-42b7064413f6667f3c3cfc77dc0dd0de",
    "名称": "B 小组",
    "成员": [
        "张力",
        "刘明"
    ],
    "平均年龄": 22.5,
    "所获荣誉": [
        "学习先进小组",
        "文体先进小组",
        "卫生先进小组"
    ]
}

```

```

{
    "_id": "4c1168fca1d0ad9f69a1267b86002695",
    "_rev": "1-2d7bd5cff3806e3227fc9b48fa3b8cc8",
    "名称": "C 小组",
    "成员": [
        "刘明",
        "杨丽",
        "马晓雯",
        "朱慧"
    ],
    "平均年龄": 21,
    "所获荣誉": [

```

```
"文体先进小组",
"学习先进小组"
]
}
```

在 Map 的步骤中，输入的文档会从原始的结构转换或者映射为新的 key/value 对。现在我们通过一个 Map 函数来完成对小组名称的查询。在临时视图中输入以下函数：

```
function(doc) {
  if (doc.名称)
  {
    emit(doc.名称);
  }
}
```

运行后得到如图 11-9 所示的结果。

Key ▲	Value
"A小组" ID: 4c1168fca1d0ad9f69a1267b86000ed5	null
"B小组" ID: 4c1168fca1d0ad9f69a1267b86001c23	null
"C小组" ID: 4c1168fca1d0ad9f69a1267b86002695	null

Showing 1-3 of 3 rows

← Previous Page | Rows per page: 10 | Next Page →

图 11-9 查看小组名称的结果

可以看到我们查询到了所有小组的名称，同时也可以看到小组的名称的下面还有该文档的 ID。还可以查询出所有小组中的所有人的姓名。

Map 函数如下：

```
function(doc) {
  if (doc.成员)
  {
    for(var i in doc.成员)
    {
      emit(doc.成员[i]);
    }
  }
}
```

该函数的运行结果如图 11-10 所示。

Key ▲	Value
"刘伟" ID: 4c1168fca1d0ad9f69a1267b86000ed5	null
"刘明" ID: 4c1168fca1d0ad9f69a1267b86001c23	null
"刘明" ID: 4c1168fca1d0ad9f69a1267b86002695	null
"张力" ID: 4c1168fca1d0ad9f69a1267b86001c23	null
"朱慧" ID: 4c1168fca1d0ad9f69a1267b86002695	null
"李刚" ID: 4c1168fca1d0ad9f69a1267b86000ed5	null
"杨丽" ID: 4c1168fca1d0ad9f69a1267b86002695	null
"赵小云" ID: 4c1168fca1d0ad9f69a1267b86000ed5	null
"马晓雯" ID: 4c1168fca1d0ad9f69a1267b86002695	null
Showing 1-9 of 9 rows	
← Previous Page Rows per page: 10 Next Page →	

图 11-10 查看所有的成员的姓名

单击图 11-10 所示界面左上角的“key”按钮可以执行改变成员的姓名称的排序等操作。

现在来看看 Reduce 函数的执行过程。Reduce 就是将执行 Map 函数后得到的 key/value 对削减为一个单个的值或一个数据集合，这个过程是可选择的，而 Map 函数的执行过程是一个视图所必需的。前面提到一个 Map 函数执行的过程会产生包含文档 ID 和 key 以及 value 的行，Reduce 的输入则是这些由 Map 得到的 key/value 的值，而不是文档 ID。执行 Reduce 函数过后，将会产生一个单独的对所有 value 的化简结果或一个基于 key 的所有分组的计算。分组（Grouping）的操作无法在 Reduce 函数中控制，它只能通过传递到视图中的参数来控制。

CouchDB 本身包含了三个内嵌的 Reduce 函数，分别是：_count，_sum 和 _stats，它们的功能分别如表 11-2 所示。在大多数情况下，对于我们开发 CouchDB 的应用程序来说，它们基本上够用了。在这三个函数中，_sum 和 _stats 仅仅会对数据的值进行化简，而 _count 函数可以对任何类型的值计数，当然也包括 null 类型的值。

表 11-2 内嵌 Reduce 函数

函 数	输 出
_count	返回 Map 结果集中值的个数
_sum	返回 Map 结果集中数值的求和结果
_stats	返回 Map 结果集中数值的统计结果

内嵌的 _count 函数是开发程序的时候最常用的 Reduce 函数，由于该函数可以用来统计任何值的个数，包括 null 值，所以当 Map 函数中 emit 的 value 值略去不写的时候，也可以使用该函数。下面是一个示例。

Map 函数：

```
function(doc) {
    if(doc.成员)
    {
        for(var i in doc.成员)
        {
            emit(doc.成员[i]);
        }
    }
}
```

Reduce 函数:

```
_count
```

程序运行的结果如图 11-11 所示。

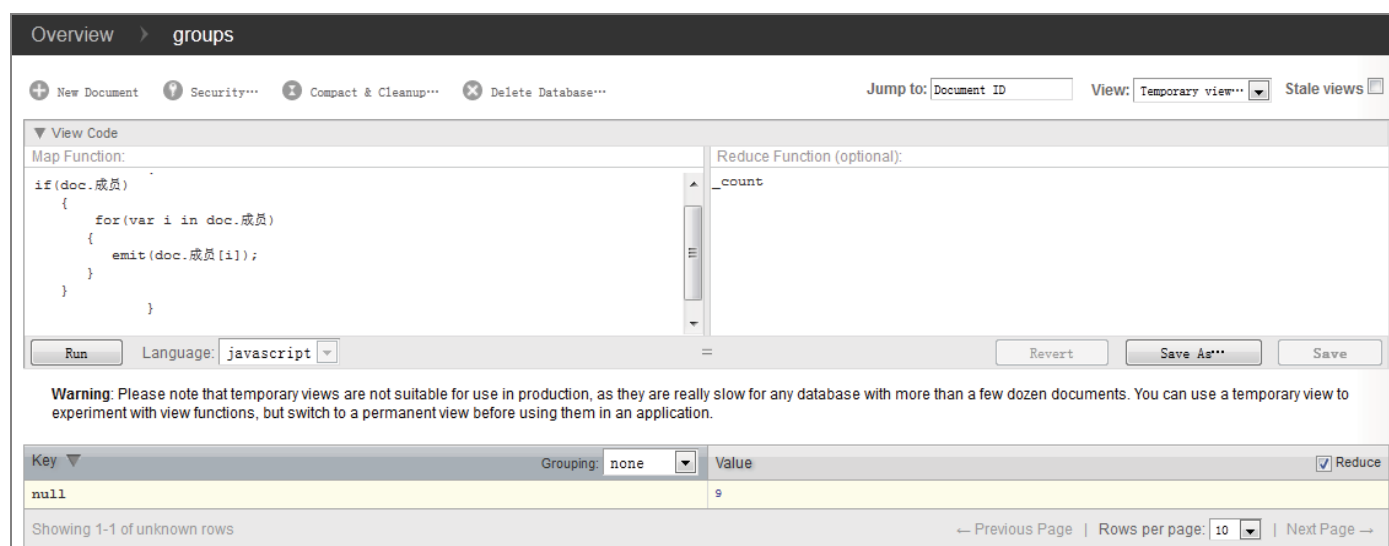


图 11-11 _count 函数的运行结果

内嵌函数_sum 会返回一个 Map 函数输出的值的和，所以该函数要求所有求和的值均为数值类型。

Map 函数:

```
function(doc) {
    if(doc.成员)
    {
        for(var i in doc.成员)
        {
            emit(doc.成员[i], doc.平均年龄);
        }
    }
}
```

```

    }
}

```

Reduce 函数:

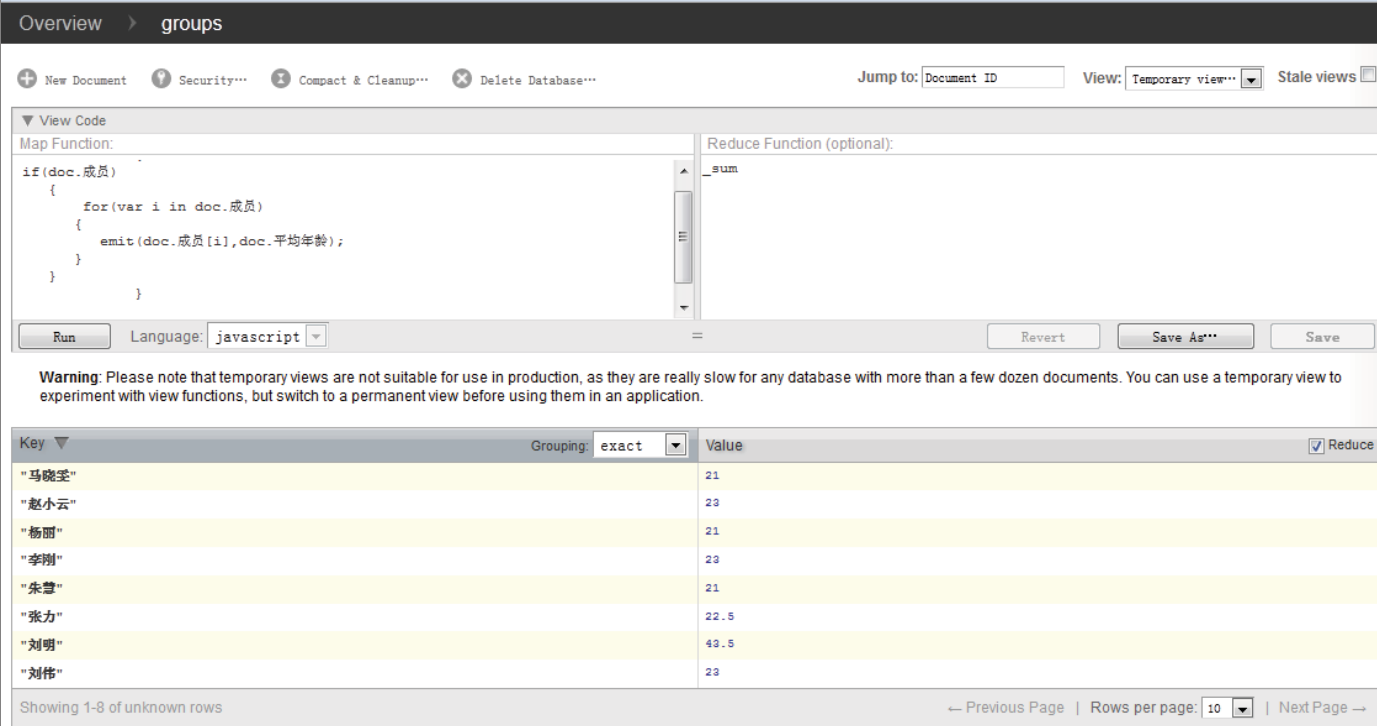
```

_sum

```

在“Grouping”下拉框中选择“exact”，则返回每个人对应的平均年龄。运行的结果如图 11-12 所示。

内嵌函数_stats 返回一个 JSON 数据对象，该对象中包含 sum、count、min、max、sumsq 的值，分别表示求和、计数、求最小值、求最大值、求和的平方。和_sum 一样，该函数同样要求所有的参与运算的值均为数值类型。



Warning: Please note that temporary views are not suitable for use in production, as they are really slow for any database with more than a few dozen documents. You can use a temporary view to experiment with view functions, but switch to a permanent view before using them in an application.

Key	Grouping: exact	Value
"马晓雯"		21
"赵小云"		23
"杨丽"		21
"李刚"		23
"朱慧"		21
"张力"		22.5
"刘明"		43.5
"刘伟"		23

Showing 1-8 of unknown rows

图 11-12 _sum 函数的运行结果

Map 函数:

```

function(doc) {
    if(doc.成员)
    {
        for(var i in doc.成员)
        {
            emit(doc.成员[i], doc.平均年龄);
        }
    }
}

```

```
}  
}
```

Reduce 函数:

```
_stats
```

执行后的结果如图 11-13 所示。

在执行 Map/Reduce 的过程中，数据库中的每一个文档为 Map 函数的参数，该函数可以将所有的文档都忽略，或者通过 emit() 返回一行或者多行 key/value 对。除了文档之外 Map 函数几乎不依靠任何信息，正是这种执行的独立性，使得 CouchDB 视图可以大量、并行地产生。

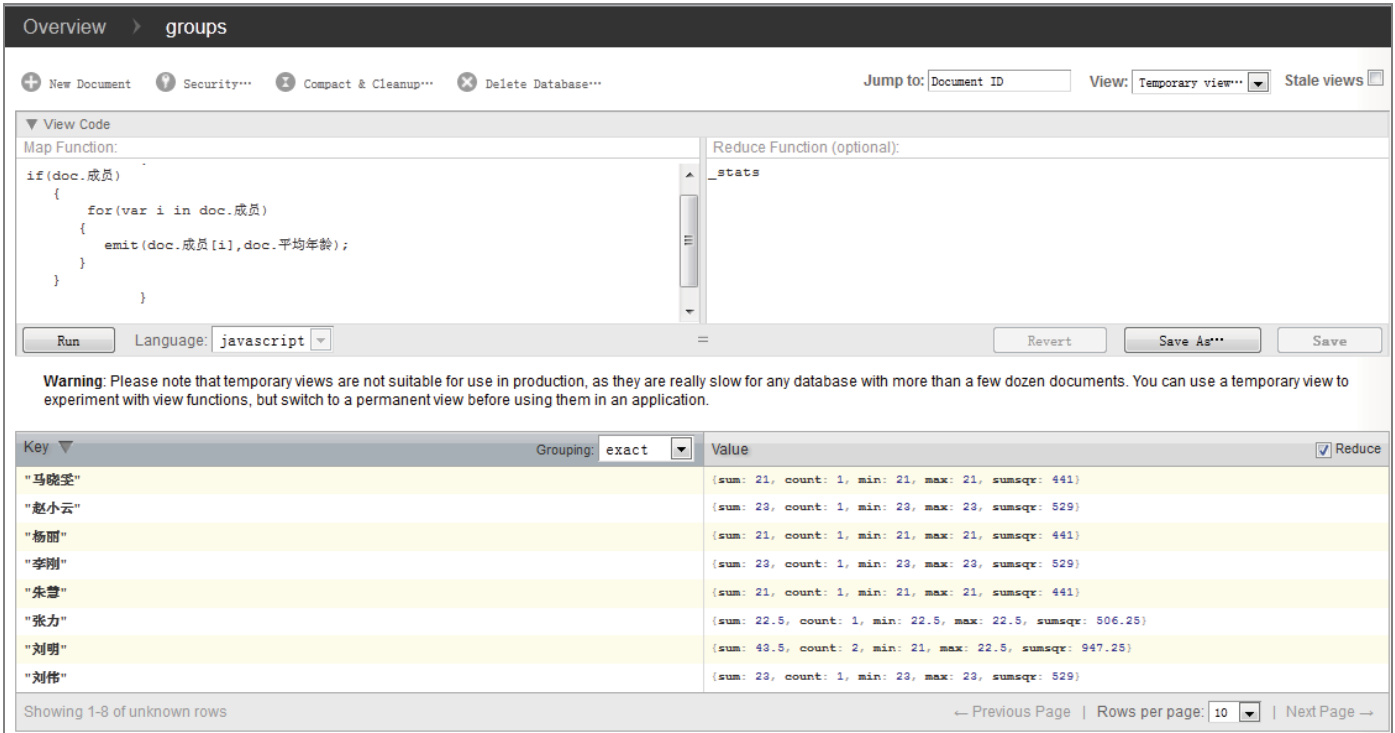


图 11-13 _stats 函数的运行结果

CouchDB 的视图按照键（key）的顺序进行排序并以行组（rows）的形式进行存储，因此即使按照多个键值在成百万、上千万的数据中进行检索也会具有很高的执行效率。所以，当编写 Map 函数的时候，首要的目标就是建立一个按照相似的键值来存储相关数据的索引。

虽然 MapReduce 的功能很强大，可以完成很多的工作，但它也有自己的局限性。在 Map 阶段产生的索引是一维的，这就意味着在 Reduce 的阶段不能产生大量的数据，否则会大大降低程序的性能。比如，CouchDB 的 MapReduce 并不适于全文本检索和自组织搜索等应用，这类问

题更适合用 Lucene 这样的工具来解决。此外，在 CouchDB 中处理地理信息数据也并不是很方便，如果需要处理地理信息数据，则最好使用 CouchDB 的一个分支系统 GeoCouch。

11.2.5 CouchDB 的存储结构

简而言之，CouchDB 落实到底层的数据结构就是两类 B+树。第一类 B+树使用文档的 ID 作为 key，通过文档 ID 来查找文档的位置，由于一个 ID 会对应多个版本的文档，所以，它指向的是一个 reversion 列表的集合。第二类 B+树使用序列号来作为 key，记录最新的 reversion，当文档进行更新的时候，就会产生一个新的版本号。同步复制的追踪，数据的显示及更新都与它密不可分。可以说，某种程度上，CouchDB 就是一个带有 HTTP 接口的 B+树的管理结构。

B+树是一个非常适于存储和检索大数据的数据结构，具有检索信息迅速的优点，当需要存储数百万甚至数以亿计的数据的时候，很适合用B+树的结构^[1]。B+树所构建的存储结构的宽度远大于其深度，即便存储很大的数据量，其构建的深度也仅仅有个位数。CouchDB构建的存储结构的叶子节点存储在硬盘等访问速度较慢的介质中，而树顶端的数据将被存放在缓存中，所以仅需要在访问叶子节点的时候访问磁盘存储就可以了，这样保证了数据的存取速度。

CouchDB中B+树机制的实现与原始的B+树有所不同。它保持了原有的B+树的一切特性，并添加了MVCC机制和只添加（append-only）属性设计。B+树用于存储主要的数据库文件和视图索引，也就是说一个数据库是一个B+树，一个视图索引也是一个B+树^[2]。

MVCC 使得在没有锁机制的情况下进行并发的读和写操作成为可能。写操作的序列化使得在每一个时间点操作某个数据库的写操作仅有一个，同时写操作不能阻塞读的操作，而且在任何时间可以有任意个读操作。每一个读操作都要保证数据库视图的一致性，这些都在 CouchDB 数据库的核心存储模型中完成。

简单来说，因为 CouchDB 使用了只添加的文件，B+树的根节点在每一次文件更新的时候都会重写。然而，文件中旧的部分将不会改变，所以每一个旧的 B+树的根都会指向一个始终如一的数据库快照。

在前面我们说过，MVCC系统使用文档的_rev属性的值来保证每一个用户当前只能修改一个版本的文档。B+树会利用这个值进行对比，当一次写操作完成之后，通过这次对比就可以知道该更新的版本是否是一个授权的版本^[4]。

由于旧版本的文档在新版本的文档产生时并没有被覆盖或者删除，所以当用户读取某一个旧版本的文档时并不影响新版本的修改。对于一个修改操作频繁的文档来说，可能出现用户同

时读取文档的三个版本的现象，这三个版本的文档对每一个用户来说都是最新的，这时候如果有别的用户对新版本的文档进行了更新并提交，此时新进入系统读取该文档的用户读取的是新版本的文档，而老用户读取的仍然是旧版本的文档。

对数据库来说，提交就是反映数据库文件变动与更新的过程，这个过程在文件页脚中完成^[1]。文件页脚是距数据库文件结尾处 4 KB 大小的内容，包含两个部分，每个大小为 2KB，每次连续写入两次。首先，CouchDB 将用户对文件的修改写入文件，并在文件页脚的第一部分记录文件的新长度，此时强制刷新磁盘存储的数据，然后将文件页脚第一部分的内容复制一份到第二部分的位置处，并再次进行一次强制数据刷新。

在这个过程中，如果出现了什么问题，比如主机断电，CouchDB 需要重启等，数据库文件都处在一致的状态而不用再进行其他检查操作。维持数据一致性的具体过程为：CouchDB 从文件的尾部反向读取文件，当找到文件页脚对的时候就检查文件页脚两个部分的内容，如果第一个包含校验和的 2KB 区域的数据已经被污染，CouchDB 将会用第二个替换它。如果第二个被污染了，那么 CouchDB 将会用第一个的内容替换。只有当文件页脚两个部分的内容都在磁盘上更新成功后，CouchDB 才会响应一次写操作成功的消息。数据没有丢失，在磁盘上的数据也不会被污染。这种设计使得在对 CouchDB 进行写操作结束后没有类似关系数据库中的关闭操作，可以直接退出。

11.2.6 SQL 和 CouchDB

SQL 和 CouchDB 有很大的不同，下面分几个方面进行对比分析。

- 传统的 SQL 数据库的结构是预定义的，在使用数据库之前需要对数据库各个表的结构进行设计；CouchDB 的结构并不需要预先的定义，每个文档都是无模式的，不需要遵循某种结构，其中可以存放任何 JSON 格式的数据。
- 传统的 SQL 数据库是一组表结构的集合；CouchDB 是一组结构不同的命名的文档的集合。
- 传统的 SQL 数据库是满足范式的，具有原子属性，数据通过关系连接在表之间传递，数据一般没有冗余；CouchDB 的文档不满足范式，各个文档的数据是自包含的，数据常常存在冗余。
- 传统的 SQL 数据库在更新数据时需要知道数据的表的结构，否则不能对表进行更新；CouchDB 仅需要知道所操作的数据的文档的名称即可。
- 传统的 SQL 数据库的查询是在静态表模式下进行的动态查询；CouchDB 是在动态文档模式下进行静态的查询，即 Map/Reduce 的查询是静态的，必须事先知道自己需要的数据

是什么才能进行查询。

总的来说，传统的 SQL 数据库和 CouchDB 之间的对比如表 11-3 所示。

表 11-3 传统 SQL 数据库与 CouchDB 的对比

传统 SQL 数据库	CouchDB
预定义、需要遵循一定的模式	不用预定义，无模式
结构统一的表结构集合	结构任意的文档集合
满足一定范式、无数据冗余	不用满足范式、存在数据冗余
操作者需清楚表结构	只需知道文档名，不用了解文档结构
静态模式下的动态查询	动态模式下的静态查询

11.2.7 分布式环境中的 CouchDB

CouchDB 是一个分布式的数据库，它可以把存储系统分布到 n 台物理节点，并且能很好地协调与同步节点之间的数据读写一致性。由于 Erlang 无与伦比的并发特性能，使得 CouchDB 在分布式环境中具有很大的优势。对于基于 Web 的大规模文档应用(如博客、Wiki 等), CouchDB 无须像传统的关系数据库那样分库拆表，就可以实现。

CouchDB 的在分布式环境下的工作主要是靠 CouchDB Lounge 来实现的^[1]。CouchDB Lounge 是一个 CouchDB 的基于代理的分区和集群框架。Lounge 有两个主要的组件 Dumbproxy 和 Smartproxy，Dumbproxy 负责用户对文档的 GET 和 PUT 请求，Smartproxy 负责 view 的处理。

Lounge 为每一个节点主机分配一个哈希值的一部分 (keyspace)，这样使得用户可以向集群中尽可能多地添加主机^[5]。由于按照哈希结果的前几个字符来分配对用户请求，所以可以保证所有的节点主机分配到的数据量都近乎相等。因为 Lounge 是通过 HTTP 请求中的 URI 来获得文档的 ID 同时映射到每一个节点主机上的，所以这种方法保证了分配的一致性。

因为 CouchDB 使用 HTTP 协议，代理可以按照请求的 URI 来对文档进行分配，而不用对文档的内容进行检测，这在一定程度上建立了与 Web 架构相似的概念，这是 REST 的一个核心的准则，也是使用 HTTP 带来的益处。实际上，这是通过哈希函数处理过的 URI 实现的。

CouchDB 实现负载均衡的方法是冗余存储，它通过自己的复制器来实现，冗余存储使得主机的负载平衡和附属主机的故障恢复能力相对来讲都得到了一定的改善和提高。对于数据的安全性来说，一般都需要对数据进行两到三个备份，将数据的这种冗余性进行封装，使得在集群的上层可以将备份的数据作为一个整体的单元来对待，而让逻辑分区自己管理数据冗余和数据

恢复。实现数据冗余存储需要在 Lounge 中配置 shard map，来指定分配到每个节点的 keypace，其中 keypace 是一个通过对文档的 UUID 进行哈希处理后得到的字符串的一段。

通过以上操作，把一个逻辑上单一的数据库分成了多个部分，这些部分可以分布到不同的机器上。从较高的层次来讲，每个部分都可以看成一个单独的单元，而每个单元自身需要通过 Replication 来实现数据冗余和故障转移。

CouchDB 保证数据安全性的方法是冗余代理。为了防止硬件的失败带来的数据丢失，多开启几个代理使得负载均衡，并增加集群的工作能力和可靠性。

分布式环境下 Map/Reduce 的目的是利用函数得到需要的数据，所以我们需要通过 HTTP 代理来对各个节点处的数据进行查询，并利用 Smartproxy 将查询的结果合并在一起。具体的执行方式为：把 Map/Reduce 函数送到有数据的节点（这里和 Hadoop 的方式一样），把结果通过 Smartproxy 合并再给客户端。要指出的是，合并操作需要的内存保持为一个常数，而不会随着需要合并的结果集而变大，这一点对于集群的扩展来说很关键。

集群实现中，非常重要的一个方面就是数据的分片，Oversharding 是一种将集群分片的技术，之所以要将集群分片是因为若要扩展集群，就会涉及数据的移动，而数据的移动可以变为分片的移动。将一个大分片切成小分片，要比把小分片数据从一个机器移到另一个机器复杂很多。所以人为地把分片的数量做多，数据量做小，把多个逻辑分片放在同一个物理机器上，当分片扩大到物理机器不能承受的时候，只需要把分片移到另一台物理机器上。

限制一个代理负责的分区数量也是集群扩展的一个重要手段，为了增加集群的大小，可以使用多层代理，由根代理负责多个中间代理，这个中间代理再跟数据库通信。正由于合并时内存不会随数据集扩大，才能实现这样的多层结构。

实现的方法有两种。一种是移动分片（Moving Partitions），这是一种简单的办法，在新机器上创建一个空的数据库，然后用复制器把需要移走的数据库复制过去^[5]。修改代理的分区配置，最后回收掉老的数据库。或者用 rsync（数据镜像备份工具）将老的数据库复制到新机器，然后用复制器使新数据库和老数据库同步。另一种是分片分区（Splitting Partitions），该方法利用复制器中 _change 接口的 filter，只把一部分符合条件的数据复制到另一个数据库。通过这个办法把过大的分区中的一部分数据同步到新数据库。

11.3 MongoDB 数据库

11.3.1 MongoDB 简介

在当前数据库领域，关系数据库虽仍然占据统治地位，但是NoSQL数据库带的冲击正在逐渐被大家重视。所谓NoSQL，并不是指没有SQL，而是指“Not Only SQL”，即非传统关系型数据库。这类数据库的主要特点包括：非关系型、水平可扩展、分布式与开源。另外，它还具有模式自由、最终一致性（不同于ACID）等特点。正是由于这些有别于关系型数据库的特点，它更能适用于当前海量数据的环境。NoSQL常用的存储模式有key-value存储、文档存储、列存储、图形存储、XML存储等等，MongoDB正是文档数据库的典型代表^[6]。

MongoDB 具有以下特点。

（1）面向集合存储。所谓集合（collection），有点类似关系数据库中的表，区别在于它不要求定义模式（schema）。每个集合在数据库中有唯一的标识名，并且可以包含无限数目的文档。因此，MongoDB可以存储对象和JSON形式的数据^[7]。

（2）模式自由。不必为存储到 MongoDB 中数据定义任何结构，不同结构的数据也可以放到同一个集合中，可以根据需要灵活地存储数据。

（3）支持动态查询。MongoDB 支持丰富的查询表达式，查询指令为 JSON 形式的标记，便于查询文档中内嵌的对象和数组。

（4）完整的索引支持，包含文档内嵌对象和数组。

（5）支持复制和故障恢复。MongoDB 数据库支持主从模式和服务器之间的数据复制。复制的目的是提供冗余和自动故障恢复。

（6）二进制数据存储。MongoDB 使用传统高效的二进制数据存储方式，支持二进制数据和大型对象（包括图片或者视频等）。

（7）自动分片以支持云级别的伸缩性。自动分片功能支持水平的数据库集群，可动态添加机器。

（8）支持多种语言。MongoDB 支持 C、C++、C#、Erlang、Haskell、JavaScript、Java、Perl、PHP、Python、Ruby 及 Scala (via Casbah)的驱动语言。

11.3.2 MongoDB 的安装

从 MongoDB 官网下载最新的安装程序，目前最新的版本为 2.0.2，地址为 <http://www.mongodb.org/downloads>。它提供了多个版本，适用多种操作系统，包括 OS X、Linux、Windows、Solaris 的 32 位及 64 位版本。

注意：

- 在 32 位平台上 MongoDB 不允许数据库文件（累计总和）超过 2 GB。
- MongoDB 服务器（mongod）必须运行在小字节序 CPU 中，所以如果使用 PPC OS X 机器，mongod 将无法工作。
- 如果你运行的是比较老版本的 Linux，以致数据库不能够启动，或者出现浮点数异常，可以试试 legacy-static 版本。

1. 在 Windows 下安装 MongoDB

（1）下载 MongoDB。

注意，选择适用于 Windows 32 位或者 64 位平台的 MongoDB。

（2）设置 MongoDB 安装目录。

为了方便操作，将压缩文件解压到 C 盘根目录，将解压文件夹重命名为 mongodb，完整的安装路径为 C:\mongodb。

（3）设置数据文件存放目录。

MongoDB 默认的数据文件存放目录是\data\db，但是它并不会自动创建这个目录，需要手动创建。从控制台创建目录，命令如下：

```
C:\>mkdir \data
C:\>mkdir \data\db
```

当然，你也可以直接在“我的电脑”界面中手动创建文件夹。

如果想将数据目录放到其他地方，比如放到 C:\db 中，可以用控制台命令：

```
C:\mongodb\bin\>mongod--dbpath c:\db
```

（4）设置日志目录。

为了便于对 MongoDB 进行维护，需要设置日志目录。首先手动创建日志文件，完整路径为 C:\mongodb\logs\test.log。然后，安装 MongoDB 系统服务记录日志，命令如下：


```
C:\mongodb\bin\>mongod--logpath c:\mongodb\logs\test.log
```

(5) 启动 MongoDB 服务。

使用控制台命令：

```
C:\mongodb\bin>mongod-dbpauthdb
```

或者直接运行安装目录下的 `mongod.exe` 文件，显示如下：

```
c:\Mongo>mongod -dbpathdb
...
Sun Apr 08 15:41:26 [initandlisten] waiting for connections on port 27017
Sun Apr 08 15:41:26 [websvr] admin web console waiting for connections on port 28017
```

(6) 连接客户端。

在新打开的控制台中输入 `mongo`，显示如下信息，表明安装成功：

```
c:\Mongo>mongo
MongoDB shell version: 2.0.4
```

```
connecting to: test
>
```

2. 在 Linux 下安装 MongoDB

(1) 下载 MongoDB。

对于 32 位 Linux，命令如下：

```
$ # replace "1.6.4" in the url below with the version you want
$ curl http://downloads.mongodb.org/linux/mongodb-linux-i686-1.6.4.tgz >
mongo.tgz
$ tar xzf mongo.tgz
```

对于 64 位 Linux，命令如下：

```
$ # replace "1.6.4" in the url below with the version you want
$ curl http://downloads.mongodb.org/linux/mongodb-linux-x86_64-1.6.4.tgz >
mongo.tgz
$ tar xzf mongo.tgz
```

(2) 创建数据目录。

默认情况下 MongoDB 将数据存储到 `/data/db` 中，但是它不会自动创建这个目录。创建目录的命令如下：

```
$ sudo mkdir -p /data/db/
$ sudo chown `id -u` /data/db
```

你也可以通过 “`--dbpath`” 选项选择其他的目录作为数据目录。

(3) 运行并且连接服务器。

首先在一个终端启动 MongoDB 服务器：

```
$ ./mongodb-xxxxxxx/bin/mongod
```

再从另外一个终端启动一个 shell，默认情况下它会自动连接本地服务器：

```
$ ./mongodb-xxxxxxx/bin/mongo
>db.foo.save( { a : 1 } )
>db.foo.find()
```

现在 MongoDB 就安装成功了。

11.3.3 MongoDB 入门

11.3.3.1 连接数据库

在这里我们使用 MongoDB 提供的 JavaScript shell 进行数据库操作,当然也可以通过不同的驱动利用其他编程语言实现同样的功能,不过 shell 在管理数据库的方面还是很方便的。

启动 JavaScript shell 的方法很简单, 命令如下:

```
C:\mongodb\bin\mongo
```

在默认情况下, shell 连接到本地 test 数据库, 可以看到如下信息:

```
C:\mongodb\bin>mongo
MongoDB shell version: 2.0.2
connecting to: test
>
```

“connecting to”的后面是要连接的数据库的名字, 如果想换成其他的数据库, 可以用如下命令:

```
> use mydb
switched to dbmydb
```

注意: 切换数据库后, 如果切换的数据库不存在, 系统并不会马上创建这个数据库, 而是在对它进行第一次插入操作时才创建。这意味着当你使用“show dbs”命令查看现有数据库时, 并不能看到切换的数据库。

11.3.3.2 动态模式 (无模式)

MongoDB 包含数据库 (database)、集合 (collection), 以及和传统关系数据库很相似的索引结构 (index)。对于数据库和集合这些对象 (object), 系统会隐式地进行创建, 然而一旦创建它们就被记录到了系统目录中 (db.systems.collections, db.system.indexes)。

集合由文档 (document) 组成, 文档中包含域 (field), 也就是传统关系数据库中的字段。但与关系数据库不同, MongoDB 不会对域进行预定义, 也没有给文档定义模式, 这就意味着文档中不同域和它们的值是可以变化的。因此, MongoDB 并没有 “alter table” 操作来增加或者减少域的个数。在实际应用中, 一个文档中通常包含相同类型的结构, 但这并不是必须的。这种弹性使得模式变动或者增加变得非常容易, 几乎不用写任何脚本程序就可实现 “alter table” 操作。另外, 动态模式机制便于对基于 MongoDB 数据库的软件进行重复性开发, 大大减少了由

于模式变化所带来的工作量。

11.3.3.3 向集合中插入数据

首先创建一个名为 `test` 的集合，再向 `test` 中插入数据。我们创建两个对象 `j` 和 `t`，然后将它们保存到集合 `things` 中。

在下面的例子中，“>”表示 `shell` 命令提示符：

```
>j={name:"mongo"}
{ "name" : "mongo" }
> t={x:3}
{ "x" : 3 }
>db.things.save(j)
>db.things.save(t)
>db.things.find()
{ "_id" : ObjectId("4f361b1f64480e0bcb6d6021"), "name" : "mongo" }
{ "_id" : ObjectId("4f361c6364480e0bcb6d6022"), "x" : 3 }
>
```

注意：

- 我们并没有预定义集合，数据库是在进行第一次插入操作时自动创建集合的。
- 我们存储的文档包含不同的域，在这个例子中，文档中没有相同的数据元素。但在实际应用中，往往把相同结构的数据存储在一个集合中。
- 一旦被存储到数据库中，如果没有事先定义，对象就会被自动分配一个 `ObjectId`，并且存储到 `field_id` 域中。当你运行上面的例子时，会有不同的 `ObjectId`。

向集合中增加更多的记录，下面的代码利用了循环结构：

```
> for (var i=1; i<=20; i++) db.things.save({x:4,j:i})
>db.things.find()
{ "_id" : ObjectId("4f361b1f64480e0bcb6d6021"), "name" : "mongo" }
{ "_id" : ObjectId("4f361c6364480e0bcb6d6022"), "x" : 3 }
{ "_id" : ObjectId("4f36234964480e0bcb6d6023"), "x" : 4, "j" : 1 }
.....
{ "_id" : ObjectId("4f36234964480e0bcb6d6034"), "x" : 4, "j" : 18 }
has more
>
```

值得注意的是，上面例子中没有显示出全部的文档，`shell` 默认显示的数目是 20。如果想查看其余的文档，可以使用 `it` 命令：

```

{ "_id" : ObjectId("4f36234964480e0bcb6d6034"), "x" : 4, "j" : 18 }
has more
>it
{ "_id" : ObjectId("4f36234964480e0bcb6d6035"), "x" : 4, "j" : 19 }
{ "_id" : ObjectId("4f36234964480e0bcb6d6036"), "x" : 4, "j" : 20 }
>

```

严格来讲，`find()`返回的是指针对象。但是在上面的例子中，我们并没有把指针对象赋予任何变量。所以，`shell` 自动对指针进行迭代，直到给出初始的结果集，我们可以通过 `it` 命令显示出其余的信息，但是也可以直接对 `find()`返回的指针进行操作，11.3.3.4 节中将其进行介绍。

11.3.3.4 查询数据

在讨论数据查询之前，先了解一下如何操作查询结果，即指针对象。我们将使用简单的 `find()` 方法，返回集合中全部的文档，之后再讨论如何写出特定的查询语句。

为了了解使用 `mongo shell` 时集合中的全部元素，我们需要明确地使用 `find()`方法返回的指针。利用 `while` 循环对 `find()`返回的指针进行迭代，实现和前面例子相同的查询结果：

```

>var cursor=db.things.find()
>while (cursor.hasNext()) printjson(cursor.next())
{ "_id" : ObjectId("4f361b1f64480e0bcb6d6021"), "name" : "mongo" }
{ "_id" : ObjectId("4f361c6364480e0bcb6d6022"), "x" : 3 }
{ "_id" : ObjectId("4f36234964480e0bcb6d6023"), "x" : 4, "j" : 1 }
.....
{ "_id" : ObjectId("4f36234964480e0bcb6d6036"), "x" : 4, "j" : 20 }
>

```

上面的例子显示了指针迭代器，`hasNext()`方法判断是否还能返回文档，`next()`方法返回下一个文档。我们也使用了内置的 `printjson()`方法使文档以 JSON 形式展现。

当在 `JavaScript shell` 下工作时，我们也可以利用 `JavaScript` 语言的特征，比如使用 `forEach` 输出指针对象。下面的代码输出与上面相同的查询结果，但是在代码中使用 `forEach` 而不是 `while` 循环：

```

>db.things.find().forEach(printjson)
{ "_id" : ObjectId("4f361b1f64480e0bcb6d6021"), "name" : "mongo" }
{ "_id" : ObjectId("4f361c6364480e0bcb6d6022"), "x" : 3 }
{ "_id" : ObjectId("4f36234964480e0bcb6d6023"), "x" : 4, "j" : 1 }
.....
{ "_id" : ObjectId("4f36234964480e0bcb6d6036"), "x" : 4, "j" : 20 }
>

```

在使用 `forEach()` 方法的时候，必须为指针指向的每一个文档定义函数（这里用了内置方法 `printjson()`）。

在 `mongo shell` 中，可以像对数组一样操作指针：

```
>var cursor=db.things.find()
>printjson(cursor[4])
{ "_id" : ObjectId("4f36234964480e0bcb6d6025"), "x" : 4, "j" : 3 }
>
```

当用这种方式使用指针时，指针指示的值都能同时加载到内存中，这一点不利于返回较大的查询结果，因为有可能发生内存溢出。对于结果较大的查询，应该用迭代方式输出指针值。

另外，你可以将指针转变为真正的数组进行处理：

```
>arr[5]
{ "_id" : ObjectId("4f36234964480e0bcb6d6026"), "x" : 4, "j" : 4 }
>
```

注意，这些数组特性都仅适用于 `shell` 模式，但对于其他语言环境并不适合。`MongoDB` 指针并不是快照，当在集合上进行操作时，如果有其他人在集合里第一次或者最后一次调用 `next()`，那么你的指针可能不能成功返回结果，所以要明确锁定你要查询的指针。

11.3.3.5 条件查询

我们已经知道如何操作查询返回的指针，现在我们要针对特定条件实现对查询结果的筛选。一般来说，实现条件查询就需要建立“查询文档”，即指明键需要匹配的模式和值的文档。对于这一点用例子证明要比用文字解释清楚得多。在下面的例子中，我们将给出 `SQL` 查询，并且说明如何借助 `mongo shell` 使得 `MongoDB` 能实现相同的查询（参见表 11-4 与表 11-5）。这种条件查询是 `MongoDB` 的基本功能，所以你也可以用其他程序驱动或者语言实现条件查询。

表 11-4 MongoDB 条件查询 (name="mongo")

SELECT * FROM things WHERE name="mongo"
<pre>>db.things.find({name:"mongo"}).forEach(printjson) { "_id" : ObjectId("4f361b1f64480e0bcb6d6021"), "name" : "mongo" } ></pre>

表 11-5 MongoDB 条件查询 (x=4)

SELECT * FROM things WHERE x=4

```
>db.things.find({x:4}).forEach(printjson)
{ "_id" : ObjectId("4f36234964480e0bcb6d6023"), "x" : 4, "j" : 1 }
{ "_id" : ObjectId("4f36234964480e0bcb6d6024"), "x" : 4, "j" : 2 }
```

续表

```
.....
{ "_id" : ObjectId("4f36234964480e0bcb6d6036"), "x" : 4, "j" : 20 }
>
```

查询表达式本身是一个文档，一个查询文档{a:A, b:B, ...}意思是“where a==A and b==B and ...”。如果想了解更多条件查询有关的信息，可以到 MongoDB 官网上查看 MongoDB 开发者手册，网址如下：

<http://www.mongodb.org/display/DOCS/Manual>

MongoDB 也允许返回“部分文档”，也就是结果中只包含数据库文档的一些子元素，类似于关系数据库中针对某些列的查询。为了实现这个查询，可以在 find()方法中增加第二个参数，表示返回某些特定元素。为了便于说明，下面我们还是实现 find({x:4})的查询，只不过增加了额外的参数使得结果中只包含 j 元素：

表 11-6 MongoDB 条件查询（返回特定元素 j）

SELECT j FROM things WHERE x=4
<pre>>db.things.find({x:4},{j:true}).forEach(printjson) { "_id" : ObjectId("4f36234964480e0bcb6d6023"), "j" : 1 } { "_id" : ObjectId("4f36234964480e0bcb6d6036"), "j" : 20 } ></pre>

注意：“_id”字段总是会返回在结果中的。

11.3.3.6 便捷查询 findOne()

为了方便用户，mongo shell（以及一些其他的程序驱动）不必编写程序来处理查询指针，就能通过 findOne()方法实现返回一个文档的功能。findOne()方法和 find()方法的参数是一样的，但它不是返回一个指针，而是返回数据库中满足条件的第一个文档，或者在没有满足条件文档的情况下返回 null。

下面以查找满足 “name=='mongo'” 的第一个文档为例。有很多种方法可以实现，包括调用 `next()` 方法（在判断非空之后），或者将指针当成数组返回第一个位置上（下标为 0）的元素。然而，相比之下 `findOne()` 是最简便的：

```
>printjson(db.things.findOne({name:"mongo"}))
{ "_id" : ObjectId("4f361b1f64480e0bcb6d6021"), "name" : "mongo" }
```

这是一种更有效的方法，因为用户只会收到从数据库返回的唯一一个对象，所以能大大减少数据库和网络的负荷。它和查询 `find({name:"mongo"}).limit(1)` 是等效的。

下面是一个针对 `_id` 查询的例子：

```
>var doc=db.things.findOne({_id:ObjectId("4f36234964480e0bcb6d6036")})
>doc
{ "_id" : ObjectId("4f36234964480e0bcb6d6036"), "x" : 4, "j" : 20 }
>
```

从这个例子也可以看出，`findOne()` 方法返回的是文档对象，而不是指针。

11.3.3.7 通过 `limit()` 限制结果个数

你可以通过 `limit()` 方法控制查询结果中返回的最大数目。这对于提高数据库性能是很重要的，因为它可以限制数据库的工作负荷以及网络的传输负荷。下面是一个例子：

```
>db.things.find().limit(3)
{ "_id" : ObjectId("4f361b1f64480e0bcb6d6021"), "name" : "mongo" }
{ "_id" : ObjectId("4f361c6364480e0bcb6d6022"), "x" : 3 }
{ "_id" : ObjectId("4f36234964480e0bcb6d6023"), "x" : 4, "j" : 1 }
>
```

11.3.3.8 修改数据

1. 在 mongo shell 中利用 `save()` 方法修改文档

正如前面例子所示，可以利用 `save()` 方法存储新的文档。同样我们可以利用 `save()` 方法更新集合中现有的文档。还是利用前面的例子，现在要向 `name` 为 `mongo` 的文档中增加新的信息：

```
>var mongo=db.things.findOne({name:"mongo"})
>print(tojson(mongo))
{ "_id" : ObjectId("4f361b1f64480e0bcb6d6021"), "name" : "mongo" }
>mongo.type="database"
database
>db.things.save(mongo)
>db.things.findOne({name:"mongo"})
```

```
{
  "_id" : ObjectId("4f361b1f64480e0bcb6d6021"),
  "name" : "mongo",
  "type" : "database"
}
>
```

这是一个简单的例子，向现有文档中增加了字符串类型的元素。当我们调用 `save()` 方法的时候，它会发现这个文档已经有了一个 “_id” 字段，所以它只是对文档进行了更新工作。

2. 在文档中直接嵌入文档

下面我们将学习如何往现有文档中嵌入其他文档，同时还会学习如何基于内嵌文档的值来查询文档。作为对现有文档进行更新的另一个例子，我们将向集合中的现有文档嵌入新的文档。为了简单起见，还是对文档 `{name: "mongo"}` 进行操作：

```
>var mongo=db.things.findOne({name:"mongo"})
>print(tojson(mongo))
{
  "_id" : ObjectId("4f361b1f64480e0bcb6d6021"),
  "name" : "mongo",
  "type" : "database"
}
>mongo.data={a:1,b:2}
{ "a" : 1, "b" : 2 }
>db.things.save(mongo)
>db.things.findOne({name:"mongo"})
{
  "_id" : ObjectId("4f361b1f64480e0bcb6d6021"),
  "name" : "mongo",
  "type" : "database",
  "data" : {
    "a" : 1,
    "b" : 2
  }
}
>
```

正如你看到的，我们向文档中增加了新的数据，即键值为 “data” 的新文档 `{a:1, b:2}`。注意，`data` 字段本身也是一个文档，它被内嵌到父文档中。借助 BSON 这种数据格式，你可以把文档嵌套或者内嵌到任何层次。也可以通过对内嵌文档进行查询，如下所示：

```
>db.things.findOne({"data.a":1})
```

```
{  
  "_id" : ObjectId("4f361b1f64480e0bcb6d6021"),  
  "name" : "mongo",  
  "type" : "database",
```

```

        "data" : {
            "a" : 1,
            "b" : 2
        }
    }
}
>db.things.findOne({"data.a":2})
null
>

```

3. 数据库引用

如果不想通过内嵌方式来插入新的文档信息，那么可以通过数据库引用的方式来指向另一个数据库文档，这一点类似于关系数据库中的外键。一个数据库引用简称为“DBRef”，大部分程序驱动都会支持创建 DBRef。有一些还支持额外的功能，比如废弃引用或者自动引用。

让我们重复上面的例子，创建一个文档并且把它放置到另一个集合中，比如放到 otherthings 集合中，并且把它作为“mongo”文档的引用，放到“otherdata”字段中：

```

// 首先往 otherthings 集合中插入新的文档
>var other={s:"other thing",n:1}
>db.otherthings.save(other)
>db.otherthings.find()
{ "_id" : ObjectId("4f3e3d34489c888c166ac760"), "s" : "other thing", "n" : 1 }

//得到 mongo 对象，并且把 other 文档添加到 otherthings 集合
>var mongo=db.things.findOne({name:"mongo"})
>print(tojson(mongo))
{
    "_id" :ObjectId("4f361b1f64480e0bcb6d6021"),
    "name" : "mongo",
    "type" : "database",
    "data" : {
        "a" : 1,
        "b" : 2
    }
}
>mongo.otherthings=new DBRef('otherthings',other._id)
{ "$ref" : "otherthings", "$id" : ObjectId("4f3e3d34489c888c166ac760") }
>db.things.save(mongo)
>db.things.findOne({name:"mongo"}).otherthings.fetch()
{
    "_id" :ObjectId("4f3e3d34489c888c166ac760"),
    "s" : "other thing",

```

```

        "n" : 1
    }
    //修改 other 文档并保存，然后看看当 dbshell 得到 mongo 对象并且显示其信息的时候，值是否随着 DBRef 一起改变
    >other.n=2
    2
    >db.otherthings.save(other)
    >db.otherthings.find()
    { "_id" : ObjectId("4f3e3d34489c888c166ac760"), "s" : "other thing", "n" : 2 }
    >db.things.findOne({name:"mongo"}).otherthings.fetch()
    {
        "_id" :ObjectId("4f3e3d34489c888c166ac760"),
        "s" : "other thing",
        "n" : 2
    }
    >

```

11.3.3.9 删除数据

在 mongo shell 环境下从数据库集合中删除一个对象，可以用 `remove()` 方法。其他的程序驱动会提供类似的方法，但方法名或许不一样，可能叫作 “delete”。请查看你所使用的程序驱动说明。

`remove()` 方法像 `find()` 方法一样，使用 JSON 格式的数据作为参数来选择删除哪个文档。如果你调用 `remove()` 方法时没有带参数，或者用一个空的文档作参数，那么它将删除集合中的全部文档。下面是一些例子：

```

//删除 name 字段为 mongo 的文档
>db.things.remove({name:"mongo"})
>db.things.find().forEach(printjson)
{ "_id" : ObjectId("4f361c6364480e0bcb6d6022"), "x" : 3 }
{ "_id" : ObjectId("4f36234964480e0bcb6d6023"), "x" : 4, "j" : 1 }
{ "_id" : ObjectId("4f36234964480e0bcb6d6024"), "x" : 4, "j" : 2 }
{ "_id" : ObjectId("4f36234964480e0bcb6d6025"), "x" : 4, "j" : 3 }
{ "_id" : ObjectId("4f36234964480e0bcb6d6026"), "x" : 4, "j" : 4 }
{ "_id" : ObjectId("4f36234964480e0bcb6d6027"), "x" : 4, "j" : 5 }
.....
{ "_id" : ObjectId("4f36234964480e0bcb6d6036"), "x" : 4, "j" : 20 }

//删除集合中所有的文档
>db.things.remove()
>db.things.find().forEach(printjson)

```


删除某个特定文档最有效的方法是把文档的 `_id` 字段作为筛选参数。代码如下：

```
> // myobject 是在 db.things 集合中的某个文档
> db.things.remove({_id: myobject._id})
```

11.3.3.10 帮助

除了常用的“`help`”命令，如果你想了解某个方法的功能，还可以直接输入方法名（不需要加括号），就可以看到方法的源码，下面是一个例子：

```
> printjson
function (x) {
  print(tojson(x));
}
>
```

11.3.4 MongoDB 索引

索引能够提升查询的性能，可以针对你的应用环境建立相应的索引，在 MongoDB 中建立索引也是相当简单的。

MongoDB 中的索引在概念上和传统关系型数据库 MySQL 类似。索引是一种数据结构，它用来存储一个集合中某个文档的某些域值。MongoDB 查询优化器用这种数据结构来快速排序和查找文档。MongoDB 中的索引是 B-树结构的。

11.3.4.1 单列索引

在 MongoDB 提供的 shell 环境中，可以调用 `ensureIndex()` 方法来创建索引，并且可以为文档的一个或多个键值提供索引。我们还用 11.3.3 节的例子，在“`j`”列上建立索引：

```
db.things.ensureIndex({j:1});
```

命令中的“1”表示按照递增的顺序建立索引，如果是“-1”则表示递减的顺序。

可以利用下面的命令来查看已经建立的索引：

```
db.things.getIndexes()
```

或者：

```
db.system.indexes.find()
```

在没有人为创建索引之前，显示结果如下：

```
>db.things.getIndexes()
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "ns" : "test.things",
    "name" : "_id_"
  }
]
```

或者:

```
>db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "test.things", "name" : "_id_" }
```

可以看到，系统本身已经创建了一个 `_id` 索引，这个字段是全局唯一的，并且不能删除，MongoDB 利用这个字段来索引不同的文档。

在对 “j” 列创建索引以后，显示结果如下：

```
>db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "test.things", "name" : "_id_" }
{ "v" : 1, "key" : { "j" : 1 }, "ns" : "test.things", "name" : "j_1" }
```

11.3.4.2 内嵌对象索引

MongoDB 还支持对内嵌对象的索引，比如有一个对象内容如下：

```
{ name: "Joe", address: { city: "San Francisco", state: "CA" } ,
  likes: [ 'scuba', 'math', 'literature' ] }
```

现在想对 `address` 下的 `city` 创建索引，则命令如下：

```
db.things.ensureIndex({"address.city": 1})
```

11.3.4.3 多列索引

如果想对多个数据列建立索引，MongoDB 也有相应的命令。还拿前面的例子介绍，如果在 “j” 和 “name” 上面建立索引，命令如下：

```
db.things.ensureIndex({j:1, name:-1});
```

这里 `j` 是递增顺序排列，在 `j` 相同的情况下，按照 `name` 递减的顺序排列。

另外，如果文档中的 `value` 是数组结构时，MongoDB 也有相应的索引机制，这里不详细介绍，有兴趣的读者可以参考 MongoDB 的官方文档。

上面提到的建立索引的命令都可以进一步设置可选参数，如表 11-7 所示。

表 11-7 索引参数

参 数	含 义	合 法 值	默 认 值
<code>background</code>	后台创建索引	true/false	false
<code>dropDups</code>	删除索引重复的文档	true/false	false
<code>sparse</code>	只针对存在索引列的文档建立索引	true/false	false
<code>unique</code>	唯一索引，保证不会有重复的属性列	true/false	false
<code>v</code>	版本号	0/1 0 = pre-v2.0 1 = smaller/faster (current)	1 (MongoDB 2.0)

11.3.4.4 删除索引和重建索引

删除全部索引可以利用下面的命令：

```
db.collection.dropIndexes();
```

删除特定索引，则命令如下：

```
db.collection.dropIndex({x: 1, y: -1})
```

MongoDB 还支持 `runCommand` 模式：

```
// 从集合 foo 中以键模式 {y:1} 删除索引
db.runCommand({dropIndexes:'foo', index : {y:1}})
// 删除全部索引
db.runCommand({dropIndexes:'foo', index : '*'})
```

如果在删除索引后想重建索引，可以用下面的指令重建文档中的全部索引：

```
db.myCollection.reIndex()
```

11.3.5 SQL 与 MongoDB

为了更好地理解 MongoDB 与传统关系数据库的区别，我们将几款典型的关系数据库与 MongoDB 进行对比，如表 11-8 与表 11-9 所示。

表 11-8 可执行文件对比

MySQL	Oracle	MongoDB
mysqld	oracle	mongod
mysql	sqlplus	mongo

表 11-9 术语对比

MySQL	MongoDB
database	database
table	collection
index	index
row	BSON document
column	BSON field
join	embedding and linking
primary key	_id field
group by	aggregation

MongoDB 查询被表示成 JSON（或者 BSON）对象，表 11-10 对比了在 SQL 和 MongoDB 中的查询语句。

在 MongoDB 中查询语句（以及其他的東西，比如索引键模式）被表示成 JSON（BSON）。然而，实际的动作（比如“find”）是在某种常用的编程语言中实现的。因此这些动作的具体形式在不同语言中是不同的。

表 11-10 SQL 与 MongoDB 查询语句对比

SQL 语句	MongoDB 语句
CREATE TABLE USERS (a Number, b Number)	implicit; can also be done explicitly with db.createCollection("mycoll")
ALTER TABLE users ADD ...	implicit
INSERT INTO USERS VALUES (3,5)	db.users.insert({a:3,b:5})
SELECT a,b FROM users	db.users.find({}, {a:1,b:1})
SELECT * FROM users	db.users.find()
SELECT * FROM users WHERE age=33	db.users.find({age:33})
SELECT a,b FROM users WHERE age=33	db.users.find({age:33}, {a:1,b:1})

续表

SQL 语句	MongoDB 语句
SELECT * FROM users WHERE age=33 ORDER BY name	db.users.find({age:33}).sort({name:1})
SELECT * FROM users WHERE age>33	db.users.find({age:{\$gt:33}})
SELECT * FROM users WHERE age!=33	db.users.find({age:{\$ne:33}})
SELECT * FROM users WHERE name LIKE "%Joe%"	db.users.find({name:/Joe/})
SELECT * FROM users WHERE name LIKE "Joe%"	db.users.find({name:/^Joe/})
SELECT * FROM users WHERE age>33 AND age<=40	db.users.find({'age':{\$gt:33,\$lte:40}})
SELECT * FROM users ORDER BY name DESC	db.users.find().sort({name:-1})
SELECT * FROM users WHERE a=1 and b='q '	db.users.find({a:1,b: 'q '})
SELECT * FROM users LIMIT 10 SKIP 20	db.users.find().limit(10).skip(20)
SELECT * FROM users WHERE a=1 or b=2	db.users.find({\$or:[{a:1},{b:2}]})
SELECT * FROM users LIMIT 1	db.users.findOne()
SELECT order_id FROM orders o,order_line_items li WHERE li.order_id=o.order_id AND li.sku=12345	db.orders.find({"items.sku":12345},{_id:1})
SELECT customer.name FROM customers,orders WHERE orders.id="q179" AND orders.custid=customer.id	var o=db.orders.findOne({_id: "q179 "}); var name=db.customers.findOne({_id:o.custid})
SELECT DISTINCT last_name FROM users	db.users.distinct("last_name")
SELECT COUNT(*Y) FROM users	db.users.count()
SELECT COUNT(*Y) FROM users where AGE>30	db.users.find({age: {'\$gt':30}}).count()
SELECT COUNT(AGE) from users	db.users.find({age: {'\$exists':true}}).count()
CREATE INDEX myindexname ON users(name)	db.users.ensureIndex({name:1})
CREATE INDEX myindexname ON users(name, ts DESC)	db.users.ensureIndex({name:1,ts:-1})
EXPLAIN SELECT * FROM users WHERE z=3	db.users.find({z:3}).explain()
UPDATE users SET a=1 WHERE b='q '	db.users.update({b: 'q '},{ \$set:{a:1}})
UPDATE users SET a=a+2 WHERE b='q '	db.users.update({b: 'q '},{ \$inc:{a:2}}, false,true)
DELETE FROM users WHERE z="abc "	db.users.remove({z: 'abc '})

上面列出的一些操作可以实现与 11.3 节中相同的功能。下面的例子中表示的是 JavaScript 并且它们可以在 mongo shell 中执行，下面两条修改数据库文档的语句结果是相同的：

```
//查看原先的数据库内容
>db.things.find().forEach(printjson)
{ "_id" : ObjectId("4f3f49f05ef04a06cb24f920"), "a" : 1, "b" : 2 }
{ "_id" : ObjectId("4f3f4a1e5ef04a06cb24f921"), "x" : 3, "y" : 4 }

//将 b 的值改为 5
>var thing=db.things.findOne({a:1})
>thing
{ "_id" : ObjectId("4f3f49f05ef04a06cb24f920"), "a" : 1, "b" : 2 }
>thing.b=5
5
>db.things.save(thing)
>db.things.find().forEach(printjson)
{ "_id" : ObjectId("4f3f49f05ef04a06cb24f920"), "a" : 1, "b" : 5 }
{ "_id" : ObjectId("4f3f4a1e5ef04a06cb24f921"), "x" : 3, "y" : 4 }

//将 b 的值改回成 1
>db.things.update({a:1},{ $set:{b:1}},false,true)
>db.things.find().forEach(printjson)
{ "_id" : ObjectId("4f3f49f05ef04a06cb24f920"), "a" : 1, "b" : 1 }
{ "_id" : ObjectId("4f3f4a1e5ef04a06cb24f921"), "x" : 3, "y" : 4 }
```

由这个例子看出，MongoDB 为数据库操作提供了大量灵活的方式。当然，在增加灵活性的同时，也增加了程序员记忆的难度。

11.3.6 MapReduce 与 MongoDB

MongoDB 中的 Map/Reduce 对于批量处理数据以及聚合操作是非常有用的。在思想上它跟 Hadoop 一样，从一个集合中输入数据，然后将结果输出到一个集合中。通常在使用类似 SQL 中 GROUP BY 操作时，Map/Reduce 会是一个好用的工具。

Map/Reduce 是通过数据库指令调用的。一般数据库会创建一个集合来收集操作的输出结果，Map 和 Reduce 函数都是用 JavaScript 编写并且在服务器上执行的。调用 Map/Reduce 的指令语法如下：

```
db.runCommand(
{ mapreduce : <collection>,
  map : <mapfunction>,
```



```

reduce : <reducefunction>
  [, query : <query filter object>]
  [, sort : <sorts the input objects using this key. Useful for optimization,
like sorting by the emit key for fewer reduces>]
  [, limit : <number of objects to return from collection>]
  [, out : <see output options below>]
  [, keepTemp: <true|false>]
  [, finalize : <finalizefunction>]
  [, scope : <object where fields go into javascript global scope >]
  [, jsMode : true]
  [, verbose : true]
}
);

```

- **finalize:** 方法会在 MapReduce 结束时应用的所有结果中。
- **keepTemp:** 如果设置为 true，那么产生的集合就不会被当成临时数据处理。默认情况下这个值是 false。当指定了 out 时，这个集合自动被设置为永久的。
- **scope:** 指明通过 map/reduce/finalize 可以访问到的变量。
- **verbose:** 提供关于任务执行的统计数据。

11.3.6.1 增量 Map/Reduce

如果你想进行 Map/Reduce 聚合操作的数据集会不断增长，那么你可能会想要利用增量 Map-Reduce。这就避免了每次对整个数据重新进行聚合操作。

要想利用增量 Map/Reduce，有以下三步：

(1) 首先，在一个现有的集合上运行一个 Map/Reduce 工作，并且把数据输出到它自己的集合中。

(2) 当你有更多的数据需要处理时，再运行一个 Map/Reduce 工作，但是利用 QUERY 语句过滤文档，筛选出那些新的文档。

(3) 利用 Reduce 输出选项。MongoDB 将利用 Reduce 函数来合并新产生的数据，并将其输出到输出集合中。

11.3.6.2 输出选项 (out)

对于 MongoDB 1.8 之前的版本，如果你没有指定 out 的值，那么结果将会被存放到一个临时集合中，集合的名字在输出指令中指定。否则，你可以指定一个集合的名字作为 out 的选项，而结果将会被存储到你指定的集合中。

对于 MongoDB 1.8 及其以后的版本，输出选项变了。Map/Reduce 不再产生临时集合（因此，keeptemp 被去掉了）。现在，你必须为 out 指定一个值，设置 out 的指令如下：

- **collectionName**: 默认输出类型为 “replace”。
- **{replace: "collectionName"}**: 输出结果将被插入到一个集合中，并会自动替换掉现有的同名集合。
- **{merge: "collectionName"}**: 这个选项将会把新的数据连接到旧的输出集合中。换句话说，如果在结果集和旧集中存在相同键值，那么新的键将覆盖旧的。
- **{reduce: "collectionName"}**: 如果具有某个键值的文档同时存在于结果集和旧集合中，那么一个 Reduce 操作（利用特定的 Reduce 函数）将作用于这两个值，并且结果将被写到输出集合中。如果指定了 finalize 函数，那么当 Reduce 结束后它将被执行。
- **{inline: 1}**: 借助这个选项，将不会再创建集合，并且整个 Map/Reduce 操作将在内存中进行。同样，Map/Reduce 的结果将被返回到结果对象中。注意，这个选项只有在结果集的单个文档大小在 16 MB 限制范围内时才有效。

```
db.users.mapReduce(map, reduce, {out: { inline : 1}});
```

out 指令还有其他选项。

- **db**: 指明接收输出结果的数据库名称。

```
out : {replace : "collectionName", db : "otherDB"}
```

- **{sharded: true}**: 适用于 MongoDB 1.9 及以上版本。如果设置为 true，并且将输出模式设置为输出到集合中，那么输出的集合的每个文档就将用 _id 字符进行切分。

注意：out 参数中对象的顺序对结果是有影响的。

11.3.6.3 结果对象

根据你设定的输出类型，选择 result 或者 results。只有在输出选项中使用了 inline 时，results 元素才会被选择。results 的值由包含结果的内嵌文档数组构成。

```
{
  [results : <document_array>],
  [result : <collection_name> | {db: <db>, collection: <collection_name>},]
  timeMillis :<job_time>,
  counts : {
    input : <number of objects scanned>,
    emit :<number of times emit was called>,
```

```

    output : <number of items in output collection>
  } ,
  ok : <1_if_ok>
  [, err : <errmsg_if_error>]
}

```

11.3.6.4 Map 函数

Map 函数通过变量 `this` 来检验当前考察的对象。一个 Map 函数会通过任意多次调用 `emit(key,value)` 来将数据送入 `reducer` 中。在大多数情况下，对于每个文档都只会发送一次，但是在有些情况下需要发送多次，比如计数标签，一个给定的文档可能会有一个、多个或者零个标签。每一次发送的数据被限制到最大文档大小的 50%（比如 MongoDB 1.6.x 中是 4MB，MongoDB 1.8.x 中是 8MB）。

```
function map(void) -> void
```

11.3.6.5 Reduce 函数

当你运行 Map/Reduce 时，Reduce 函数将收到一个发送值构成的数组并且要把它们简化到单个值。因为针对一个键值 Reduce 函数可能会被调用好多次，Reduce 函数返回的对象结构与 Map 函数发送的值的结构必须是完全相同的。我们可以用一个简单的例子说明这一情况。

假设对一个代表用户评价的文档构成的集合进行迭代，典型的文档内容如下：

```

{ username: "jones",
  likes: 20,
  text: "Hello world!"
}

```

我们想利用 Map/Reduce 来统计每个用户的评论数，并且计算全部用户评价中“like”的总数。为了达到这个目的，首先写一个如下的 Map 函数：

```

function() {
  emit(this.username, {count: 1, likes: this.likes} );
}

```

这个函数实际上指明了要以 `username` 来分组，并且针对 `count` 和 `likes` 字段进行聚合运算。当 Map/Reduce 实际运行的时候，一个由用户名构成的数组将被发送给 Reduce 函数，这就是为什么 Reduce 函数总是用来处理数组。下面就是一个 Reduce 函数的例子：

```

function(key, values) {
  var result = {count: 0, likes: 0};

```

```

values.forEach(function(value) {
  result.count += value.count;
  result.likes += value.likes;
});

return result;
}

```

注意,结果文档和 Map 函数所传送的文档拥有相同的结构。这是非常重要的,因为当 Reduce 函数作用于某一个 key 值时,它并不保证会对这个 key 值(这里是 username)的每一个 value 进行操作。事实上,Reduce 函数不得不运行多次。比如,当处理评论集合时,Map 函数可能遇到来自“jones”的 10 条评论。它会把这些评论传送给 Reduce 函数,得到如下聚合结果:

```
{ count: 10, likes: 247 }
```

然后,Map 函数又遇到一个来自“jones”的评论,此时,这个值必须被重新考虑来修改聚合结果。如果遇到的新评论为:

```
{ count: 1, likes: 5 }
```

那么 Reduce 函数将被这样调用:

```
reduce("jones", [ {count: 10, likes: 247}, { count: 1, likes: 5} ] )
```

最后的结果将会是上面两个值的结合:

```
{ count: 11, likes: 252 }
```

只要你理解针对一个 key 值 Reduce 函数可能会被调用多次,就容易理解为什么这个函数必须返回一个和 Map 函数发送的值具有相同结构的结果了。

注意:

- Map 函数输出的 value 必须和 Reduce 输出的 value 具有相同的结构,这样才能保证多次调用 Reduce 函数成为可能。否则,会出现奇怪的很难调试的错误。
 - 目前,Reduce 返回的值不能是一个数组,通常是一个对象或者一个数值。
-

11.3.6.6 finalize 函数

finalize 函数可能会在 Reduce 函数结束后运行,这个函数是可选的,对于很多 Map/Reduce 任务来说不是必需的。finalize 函数接收一个 key 和一个 value,返回一个最终的 value。

```
function finalize(key, value) ->final_value
```

针对一个对象你的 **Reduce** 函数可能被调用了多次。当最后只需针对一个对象进行一次操作时可以使用 **finalize** 函数，比如计算平均值。

11.3.6.7 jsMode 标识

对于 MongoDB 2.0 及以上的版本，通常 Map/Reduce 的执行遵循下面两个步骤：

- (1) 从 BSON 转化为 JSON，执行 Map 过程，将 JSON 转化为 BSON。
- (2) 从 BSON 转化为 JSON，执行 Reduce 过程，将 JSON 转化为 BSON。

因此，需要多次转化格式，但是可以利用临时集合在 Map 阶段处理很大的数据集。为了节省时间，可以利用 `{jsMode:true}` 使 Map/Reduce 的执行保持在 JSON 状态。遵循如下两个步骤：

- (1) 从 BSON 转化为 JSON，执行 Map 过程。
- (2) 执行 Reduce 过程，从 JSON 转化为 BSON。

这样，执行时间可以大大减少，但需要注意，**jsMode** 受到 JSON 堆大小和独立主键最大 500 KB 的限制。因此，对于较大的任务 **jsMode** 并不适用，在这种情况下会转变为通常的模式。

11.3.7 MongoDB 与 CouchDB 对比

MongoDB 和 CouchDB 都是文档数据库，两者很相似，数据存储格式都是 JSON 型的，都使用 JavaScript 进行操作，都支持 Map/Reduce^[9]。但其实二者有着很多本质的区别，参考 MongoDB 官方网站^[8]给出的 MongoDB 与 CouchDB 的对比，我们对 MongoDB 和 CouchDB 的之间的相同点和不同点做了如下总结。

1. MVCC (Multiple Version Concurrency Control)

MongoDB 与 CouchDB 的一大区别就是 CouchDB 是一个 MVCC 的系统，而 MongoDB 是一个 **update-in-place** 的系统。MVCC 机制非常适用于对版本控制需求大的、涉及多个脱机的数据库同步或者涉及大量主机对主机数据同步问题的系统，但是 MVCC 需要对数据库进行周期性压缩，而且此类系统一般还需要有一定的算法解决多个写操作之间的冲突。而 **update-in-place** 的写操作都是即时完成的，如果写操作成功，那么数据更新就成功，其适用于对数据对象的更新速率要求高的领域。所以，CouchDB 在分布式环境下各个主机之间的数据同步机制更为简单，而 MongoDB 在写数据的性能方面更占优势。

2. 水平扩展性

在扩展性方面，CouchDB 使用 Replication 来实现水平扩展，MongoDB 在实现水平扩展性方面使用的是 Sharding，这一点类似于 Google 的 Bigtable，而其 Replication 仅仅用来增强数据的可靠性。

3. 数据查询操作

CouchDB 不支持动态查询，用户必须为每一个查询模式建立相应的 view，并在此 view 的基础上进行查询。而 MongoDB 与传统的数据库系统类似，支持动态查询，即使在没有建立索引的行上，也能进行任意的查询。

4. 原子性

CouchDB 和 MongoDB 都支持针对单个文档的并行修改（concurrent modifications of single documents），但不支持更多的复杂事务操作。

5. 数据可靠性

CouchDB 是一个“crash-only”的系统，在任何时候停掉 CouchDB 其存储结构中设置的文件页脚都能保证数据的一致性。而 MongoDB 在非正常停掉后需要运行 `repairDatabase()` 命令来修复数据文件，在 1.7.5 版本后支持单机可靠的 `--journal` 命令。

6. Map/Reduce

MongoDB 和 CouchDB 都支持 Map/Reduce，但两者有所不同，MongoDB 只有在数据统计操作中才会用到 Map/Reduce，而 CouchDB 在普通查询时也使用 Map/Reduce。

7. JavaScript 的使用

MongoDB 和 CouchDB 都支持 JavaScript，CouchDB 用 JSON 格式的数据来进行数据存储，并用 JavaScript 来创建 view。MongoDB 的 Map/Reduce 函数也是 JavaScript 格式的，并使用 JSON 作为普通数据库操作的表达式。此外，MongoDB 也可以在操作中包含 JavaScript 语句，MongoDB 还支持服务端的 JavaScript 脚本（running arbitrary Javascript functions server-side）。

8. REST

CouchDB 具有 RESTful 的 API，是一个 RESTful 的数据库，其操作完全按照 HTTP 协议来进行，而 MongoDB 使用的是自己的二进制协议。MongoDB Server 在启动时可以开放一个 HTTP 的接口供状态监控。

9. 性能

MongoDB 是面向性能设计的，而 CouchDB 采用面向功能的设计因而性能受到一定影响。由于 MongoDB 具有优秀的性能，因此它在以往的关系型数据库不能应用的领域发挥了独特的作用。下面列举 MongoDB 性能较高的几个因素：

- 采用自己的二进制协议，而非 CouchDB 的 HTTP 协议。
- 使用内存映射的方法进行数据存储。
- 面向集合（collection-oriented）的存储，同一个集合中的数据是连续存储的。
- 数据更新方式为直接修改（update-in-place），而非使用 MVCC 的机制。
- 使用 C++编写。

10. 适用场景

如果构建一个 Lotus Notes 型的应用，推荐使用采用了 MVCC 机制的 CouchDB。此外，如果需要用到 master-master 的架构，基于地理位置的数据分布或者用于手机等移动终端的嵌入式系统的数据存储，推荐使用 CouchDB。

如果你需要高性能的数据存储服务，比如存储大型网站的用户个人信息，构建在其他存储层之上的 Cache 层等，推荐使用 MongoDB。此外，由于 MongoDB 具有更新速度快的优点，所以如果需求中有大量 update 操作，那么建议使用 MongoDB。

11.4 小结

本章主要介绍了文档数据库的典型代表 CouchDB 和 MongoDB。

在 CouchDB 部分，介绍了 CouchDB 的功能和发展历史，以及 CouchDB 的数据格式、访问方式、文档、Design 文档等重要概念。为了适应分布式海量数据的管理，本部分着重阐述了 CouchDB 在 Map/Reduce 环境下对数据和文档的增加、删除和修改等操作，以及解决分布式环境中的数据一致性、分区、容错等方法。在存储方式上，对 CouchDB 的两类 B+树的存储结构进行了简要的介绍。

在 MongoDB 部分，首先针对 MongoDB 区别于传统关系数据库的特点进行了介绍，包括面向集合存储、模式自由、支持动态查询等。接着介绍了 MongoDB 在 Windows 和 Linux 环境下的安装步骤，它们的过程大致是一致的，关键步骤是设置数据存放目录。

本章还介绍了 MongoDB 中的常用操作，其中用了大量的例子便于读者更好地理解指令的作用。然后，我们对比了 SQL 和 MongoDB 中数据库操作语句的区别，便于初学者快速找到需要的命令。最后，将现在流行的 Map/Reduce 与 MongoDB 结合到一起，简单介绍了 MongoDB 如何运用 Map/Reduce 提高批量处理和聚合操作数据的能力。

通过对 CouchDB 和 MongoDB 的介绍，希望读者可以对文档数据库的特点和应用有进一步的了解，尤其是它们在分布式环境下处理海量数据时，都借助了 Map/Reduce 技术提高数据处理的效率。这对于解决海量非结构化和半结构化数据的处理及分析问题有重要意义。

参考文献

- [1] J. Chris Anderson, Jan Lehnardt, Noah Slater. CouchDB:The Definitive Guide [M]. O'Reilly Media, 2010.
- [2] Joe Lennon. Beginning CouchDB[M]. Springer-Verlag: New York, 2009.
- [3] Bradley Holt. Writing and Querying MapReduce Views in CouchDB[M]. O'Reilly Media, 2011.
- [4] Couchdb Wiki. Apache CouchDB [EB/OL]. <http://wiki.apache.org/couchdb/>.
- [5] Bradley Holt. Scaling CouchDB[M]. O'Reilly Media, 2011.
- [6] 10gen. MongoDB 官网 [EB/OL]. <http://www.mongodb.org>.
- [7] Banker, Kyle. MongoDB in Action [M]. Manning Publications, 2011.
- [8] 10gen. Comparing Mongo DB and Couch DB [EB/OL]. <http://www.mongodb.org/display/DOCS/Comparing+Mongo+DB+and+Couch+DB>.
- [9] Dwight Merriman. MongoDB 与 CouchDB 全方位对比 [EB/OL]. 2011-03-21.
<http://database.51cto.com/art/201103/249990.htm>,.

