

大数据应用与技术丛书

The Definitive Guide to MongoDB, Second Edition

MongoDB

大数据处理权威指南 (第2版)

David Hows

Eelco Plugge

[美]

Peter Membrey

著

Tim Hawkins

王肖峰

译

Apress®



清华大学出版社

大数据应用与技术丛书

MongoDB 大数据处理

权威指南(第2版)

David Hows
[美] Eelco Plugge 著
Peter Membrey
Tim Hawkins
王肖峰 译

清华大学出版社

北 京

David Hows, Eelco Plugge, Peter Membrey, Tim Hawkins
The Definitive Guide to MongoDB: A Complete Guide to Dealing with Big Data Using MongoDB, Second Edition
EISBN: 978-1-4302-5821-6

Original English language edition published by Apress Media. Copyright © 2013 by Apress Media.
Simplified Chinese-Language edition copyright © 2015 by Tsinghua University Press. All rights reserved.

本书中文简体字版由 Apress 出版公司授权清华大学出版社出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

北京市版权局著作权合同登记号 图字：01-2014-7360

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

MongoDB 大数据处理权威指南(第2版)/(美)豪斯(Hows, D.)等著;王肖峰译.

—北京:清华大学出版社,2015

(大数据应用与技术丛书)

书名原文: The Definitive Guide to MongoDB: A Complete Guide to Dealing with Big Data Using
MongoDB, Second Edition

ISBN 978-7-302-38366-6

I. M… II. ①豪… ②王… III. ①关系数据库系统—指南 IV. ①TP311.138-62

中国版本图书馆 CIP 数据核字(2014)第 247391 号

责任编辑:王军 李维杰

装帧设计:牛艳敏

责任校对:邱晓玉

责任印制:刘海龙

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦 A 座 邮 编:100084

社总机:010-62770175 邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质量反馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

印装者:北京密云胶印厂

经 销:全国新华书店

开 本:185mm×260mm 印 张:17.75 字 数:477千字

版 次:2015年1月第1版 印 次:2015年1月第1次印刷

印 数:1~3000

定 价:49.80元

产品编号:060831-01

译者序

随着大数据的不断发展，非关系型数据库已经变得越来越重要，相关的产品也都得到了飞速发展。而其中 MongoDB 更是佼佼者，作为高性能开源文档数据库，MongoDB 以敏捷、可扩展和对企业应用友好而著称。使用 MongoDB 的公司包括 Foursquare、Craiglist、迪士尼、SAP、Intuit、EA 等。

本人从 2013 年开始有幸能够接触到 MongoDB，并在实际工作中使用它。但是在学习使用过程中始终存在着一些困难，MongoDB 相关的书籍太少，尤其是中文方面的，并且对 MongoDB 的讲解也不够详尽。因此，如果想彻底了解和学习 MongoDB，就必须硬啃它的官方文档，这对于英文能力不佳的读者肯定也是难点。

所以这次清华大学出版社准备引进出版此书时，我感到非常高兴，也非常荣幸能够负责本书的翻译工作。能够为 MongoDB 的推广做出一点贡献，能帮助大家深入了解 MongoDB 的强大特性，我就感到非常满足了。

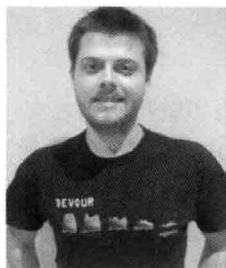
《MongoDB 大数据处理权威指南(第 2 版)》由浅入深地介绍 MongoDB 的基础知识、使用和高级功能，并且提供了丰富的样例，保证读者在读完本书之后能独立搭建起自己的开发环境。即便是对 NoSQL 毫不了解的读者，也可以顺利完成本书的阅读。对于有一定开发经验的读者，相信本书关于数据库优化、复制和分片方面的讲解也会让你感到耳目一新。总而言之，本书是学习 MongoDB 数据库的一本不可多得的精品之作。

另外，感谢清华大学出版社的编辑们为本书付出的心血。同样感谢妻子对我翻译工作的支持和鼓励。没有你们的支持和鼓励，本书就不可能顺利出版。

对于这本经典之作，译者对本书进行了严格审校，对其中一些具有争议的地方也进行了反复考证，但个人精力有限，难免有疏漏之处，敬请各位读者谅解。如有任何意见或建议，请不吝指正。本书全部章节由王肖峰翻译，参与翻译活动的还有杜欣、高国一、孙其淳、孙绍辰、徐保科、尤大鹏、袁强强、张宏、张立红、邓伟、王蕊。

最后，希望各位读者能够早日掌握强大的 MongoDB 特性，轻松地处理工作中遇到的问题，如原书作者所描述的：轻松工作，按时下班！

作者简介



David Hows 以优异的成绩毕业于澳大利亚新南威尔士州的卧龙岗大学。他第一次接触计算机，是在尝试不花钱的情况下改进家庭 PC 的性能。这促使他加入 IT 行业，David 曾经担任过系统管理员、性能工程师、软件开发者、解决方案架构师和数据库工程师等职务。David 也曾徒劳地尝试过多年足球，并且他的咖啡杯上写着“Grumble Bum”。



Peter Membrey 是一位特许 IT 研究员，他拥有 15 年使用 Linux 和开源解决方案解决现实中问题的经验。从 17 岁起他就是一位红帽认证工程师，并且他也有幸在 Red Hat 工作过，并编写了几本开源解决方案相关的书籍。他拥有利物浦大学的信息安全硕士学位，目前是香港理工大学的博士生，他的研究方向包括云计算、大数据、科学和安全。他与自己出色的妻子 Sara 和儿子 Kaydyn 一起居住在香港。尽管他的世界语一直不错，但广东话却在不断退步。



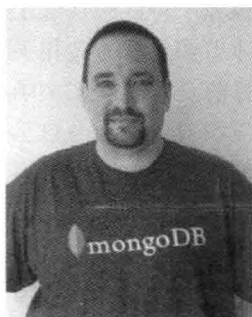
Eelco Plugge 是一位年轻的 IET/BSC 专家，他对 IT 安全领域非常感兴趣。在 21 岁时他就成为一位加密专家，目前从事移动设备管理行业，偶尔也会编写一些书籍。Eelco 最近拿到了利物浦大学的计算机安全硕士学位，并持有多项专业证书。他对 Linux、网络安全和加密技术都抱有极大的热情。Eelco 与他的年轻家庭一起居住在荷兰。他喜欢寿司，讨厌过于复杂的事情。

Tim Hawkins 曾在 1993 年创建了世界上第一个在线分类广告门户网站 loot.com，之后负责管理雅虎欧盟的许多非媒体属性产品，例如搜索、本地搜索、邮件、消息和社交网络。他目前正在管理美国主要电子零售商的一个大型离岸团队，负责开发和部署下一代电子商务应用。他喜欢帽子，讨厌复杂性。

技术编辑简介



Alexandre Beaulne 来自于加拿大的北部，他获得了蒙特利尔麦吉尔大学的神经系统学士学位，随后获得了蒙特利尔 HEC 的金融工程硕士学位。不过，他的研究在他参加了纽约的黑客学校之后达到了巅峰。他一直不断地在声明，表示他并未“明白”Haskell 的 monads¹。



Stephen Stenecker(又名 Stennie)是一位经验丰富的全能软件开发者、咨询师和讲师。Stephen 曾长期工作于澳大利亚的技术公司，是雅虎澳大利亚和新西兰、HomeScreen Entertainment 和 Grox 的技术负责人。他获得了英属哥伦比亚大学的学士学位(计算机科学)。

他目前的职位是 MongoDB Inc.的技术服务工程师，为 MongoDB 提供支持、咨询和培训。他经常在用户组和会议上发言，并且他也是悉尼 MongoDB 用户组的创始人和管理者(<http://www.meetup.com/SydneyMUG/>)。

在 Twitter、StackOverflow 或 Github 上可以找到他，他的 ID 为@stennie。

1. 译者注：Haskell 是一种编程语言，monads 则是该语言下的一种抽象数据类型。

致 谢

感谢 MongoDB 团队曾经和现有的所有成员。没有他们的帮助我们就无法完成本书的编写，人们对数据存储的观点也会出现极大的不同。我要特别感谢悉尼 MongoDB 团队的同事们，他们为本书的编写提供了极大的帮助。

——David Hows

编写书籍是一个团队性的工作。即使有时只有一个作者，在幕后也会有许多人帮助将所有东西整合在一起。因此我非常感谢 MongoDB 社区和 Apress 的所有人，感谢他们的辛勤工作、耐心和支持。特别要感谢 Dave 和 Eelco，他们帮助完成了本书的第 2 版。

我还想感谢 Chuck Smith，一位让我迷上世界语并让我保持有很高兴趣的朋友(维基百科世界语版本的创始人)。这正好与本书第 2 版的编写产生了交集，我对学习世界语的极大兴趣延伸到了本书的编写上。另外还必须感谢 L.L.Zamenhof 博士，他创建了世界语。一个远超前时代的人，我希望能带着这个梦想前进。

最后，感谢妻子 Sarah 的耐心和长久的支持。

——Peter Membrey

感谢 9gag 社区，没有他们本书就无法在数月前完成。

——Eelco Plugge

我想感谢 mongodb-user 和 mongodb-dev 邮件列表中的所有成员，感谢他们能够忍耐我提出的无休止的问题。

——Tim Hawkins

前言

我接触数据库的时间相对较晚，从 2006 年才开始使用 MySQL。在学完所有计算机本科都提供的逻辑课程之后，我开始使用 MySQL 构建出一个完整的 LAMP 架构，其中用到了一些基本表。此时，我并未对 SQL 表管理的复杂性进行深入思考。不过，随着时间的流逝，我看到了存储越来越多异构数据的需要，并且随着时间的推移，简单的模式可以如何在它的生命周期中成长和演变。

我第一次接触 MongoDB 是在 2011 年，当时 Peter Membrey 建议我不要使用包含 30 个键行以及 30 个值行的上下文表，而是应该使用 MongoDB 实例来存储数据。就像所有开发者在面对一项新技术时的感觉一样，我对之嗤之以鼻并且坚持我原来的计划。直到我使用糟糕的设计已经完成一半代码时，Peter 仍然坚持我应该尝试使用 MongoDB，此时我才接受意见。如同所有来自 SQL 阵营的开发者一样，MongoDB 能够接受任何类型数据并且可以根据任何搜索条件返回这些数据的能力让我感到震惊。直到现在我也仍然大呼过瘾。

——David Hows

本书的组织方式

在本书中，Peter、Eelco Plugge、Tim Hawkins 和我都希望能够完全地展示出我们在学习 MongoDB 时的经验：在保持设计简单和清晰的同时，教会你如何使用 MongoDB。每章内容都将展示一个单独的样例数据库，因此你可以按照模块或线性的方式阅读本书；这完全取决于你自己。这意味着如果愿意，你可以略过某些特定的章节，而不会影响你对其他内容的学习。

本书的样例命令将显示在它们的输出中。它们将以等宽的“代码”字体出现，并且会以加粗的方式显示，以便与其他结果输出加以区分。在大多数章节中，你都会遇到提示、警告和注意，它们包含有用的有时甚至极其重要的信息。

——David Hows

目 录

第 I 部分 MongoDB 基础

第 1 章 MongoDB 简介	3
1.1 了解 MongoDB 哲学	3
1.1.1 使用正确的工具处理正确的工作	3
1.1.2 天生缺少对事务的支持	5
1.1.3 JSON 和 MongoDB	5
1.1.4 采用非关系的方式	7
1.1.5 选择性能还是特性	8
1.1.6 在任何地方均可运行数据库	8
1.2 将所有组合在一起	9
1.2.1 生成或创建键	9
1.2.2 使用键和值	9
1.2.3 实现集合	10
1.2.4 了解数据库	10
1.3 了解特性列表	10
1.3.1 使用面向文档存储(BSON)	11
1.3.2 支持动态查询	11
1.3.3 为文档创建索引	12
1.3.4 使用地理空间索引	13
1.3.5 分析查询	13
1.3.6 就地更新信息	13
1.3.7 存储二进制数据	14
1.3.8 复制数据	14
1.3.9 实施分片	14
1.3.10 使用映射和归约函数	15
1.3.11 全新的聚合框架	15
1.4 获取帮助	15
1.4.1 访问网站	15
1.4.2 与 MongoDB 开发者沟通	16
1.4.3 剪切和粘贴 MongoDB 代码	16
1.4.4 在 Google 小组中寻找解决方案	16
1.4.5 利用 JIRA 跟踪系统	16
1.5 小结	16
第 2 章 安装 MongoDB	17
2.1 选择版本	17
2.2 在系统中安装 MongoDB	18

2.2.1 在 Linux 中安装 MongoDB	18
2.2.2 在 Windows 中安装 MongoDB	19
2.3 运行 MongoDB	20
2.3.1 先决条件	20
2.3.2 研究安装目录布局	20
2.3.3 使用 MongoDB shell	21
2.4 添加额外的驱动	22
2.4.1 安装 PHP 驱动	22
2.4.2 确认 PHP 安装正确	25
2.4.3 安装 Python 驱动	26
2.4.4 确认 PyMongo 安装正确	28
2.5 小结	29
第 3 章 数据模型	31
3.1 设计数据库	31
3.1.1 集合的更多细节	32
3.1.2 使用文档	33
3.1.3 在文档中内嵌或引用信息	34
3.1.4 创建_id 字段	35
3.2 构建索引	36
3.3 使用地理空间索引	37
3.4 在真实世界中使用 MongoDB	42
3.5 小结	42
第 4 章 使用数据	43
4.1 浏览数据库	43
4.2 在集合中插入数据	44
4.3 查询数据	45
4.3.1 使用点号	47
4.3.2 使用函数 sort、limit 和 skip	47
4.3.3 使用固定集合、自然顺序和 \$natural	48
4.3.4 获取单个文档	50
4.3.5 使用聚集命令	50
4.3.6 使用条件操作符	52
4.3.7 使用正则表达式	59
4.4 更新数据	59
4.4.1 使用 update()更新	60

4.4.2	使用 save()命令实现 upsert	60	6.3	连接和断开连接	91
4.4.3	自动更新信息	60	6.4	插入数据	92
4.4.4	指定匹配数组的位置	64	6.5	查询数据	94
4.4.5	原子操作	65	6.5.1	返回单个文档	94
4.4.6	以原子的方式修改和返回文档	66	6.5.2	列出所有文档	95
4.5	重命名集合	67	6.5.3	使用查询操作符	96
4.6	删除数据	67	6.5.4	查询特定信息	96
4.7	引用数据库	68	6.5.5	排序、限制和忽略数据项	97
4.7.1	手动引用数据	68	6.5.6	统计匹配结果的数目	98
4.7.2	使用 DBRef 引用数据	70	6.5.7	使用聚集框架对数组分组	98
4.8	使用索引相关的函数	71	6.5.8	使用 hint()函数指定索引	99
4.8.1	学习索引相关的命令	72	6.5.9	使用条件操作符重新定义查询	100
4.8.2	强制使用某个索引查询数据	72	6.5.10	正则表达式	106
4.8.3	限制查询匹配	73	6.6	使用 PHP 修改数据	106
4.9	小结	74	6.6.1	使用 update()函数更新数据	106
第 5 章	GridFS	75	6.6.2	节省更新操作的时间	108
5.1	背景	75	6.6.3	使用 \$since 增加特定键的值	108
5.2	使用 GridFS	76	6.6.4	使用 \$set 修改键值	109
5.3	开始使用命令行工具	76	6.6.5	使用 \$unset 删除字段	109
5.3.1	使用 _id 键	77	6.6.6	使用 \$rename 重命名字段	110
5.3.2	使用文件名	77	6.6.7	在更新/插入期间使用 \$setOnInsert 修改键值	110
5.3.3	决定文件长度	78	6.6.8	使用 \$push 向指定字段中添加值	111
5.3.4	使用块大小	78	6.6.9	使用 \$push 和 \$each 向某个键中 添加多个值	111
5.3.5	处理上传日期	79	6.6.10	使用 \$addToSet 将数据添加到 数组中	112
5.3.6	生成文件的哈希值	79	6.6.11	使用 \$pop 从数组中删除元素	113
5.4	查看 MongoDB 中的数据	79	6.6.12	使用 \$pull 删除所有指定值	113
5.4.1	使用搜索命令	81	6.6.13	同时删除多个元素	113
5.4.2	删除	81	6.6.14	使用 save()函数更新/插入数据	114
5.4.3	从 MongoDB 中获取文件	82	6.6.15	以原子的方式修改文档	115
5.4.4	mongofiles 命令小结	82	6.7	删除数据	116
5.5	使用 Python	82	6.8	DBRef	118
5.5.1	连接数据库	83	6.9	GridFS 和 PHP 驱动	120
5.5.2	访问单词	83	6.9.1	存储文件	120
5.6	在 MongoDB 中添加文件	84	6.9.2	在已存储的文件中添加元数据	121
5.7	从 GridFS 中读取文件	84	6.9.3	获取文件	121
5.8	删除文件	85	6.9.4	删除数据	122
5.9	小结	85	6.10	小结	122
第 II 部分	在开发中使用 MongoDB		第 7 章	Python 和 MongoDB	123
第 6 章	PHP 和 MongoDB	89	7.1	在 Python 中使用文档	123
6.1	比较 MongoDB 和 PHP 中的 文档	89	7.2	使用 PyMongo 模块	124
6.2	MongoDB 类	90			

7.3	连接和断开	124	8.3.5	调试 MapReduce	174
7.4	插入数据	125	8.4	小结	175
7.5	搜索数据	126	第III部分 使用MongoDB处理大数据		
7.5.1	搜索单个文档	126	第9章 数据库管理 179		
7.5.2	搜索多个文档	127	9.1	使用管理工具	179
7.5.3	使用点操作符	128	9.1.1	mongo——MongoDB 控制台	180
7.5.4	返回字段	128	9.1.2	使用第三方管理工具	180
7.5.5	使用 sort()、limit()和 skip()简化 查询	129	9.2	备份 MongoDB 服务器	180
7.5.6	聚集查询	130	9.2.1	创建第一个备份	180
7.5.7	使用 hint()指定索引	132	9.2.2	备份单个数据库	182
7.5.8	使用条件操作符重定义查询	133	9.2.3	备份单个集合	182
7.5.9	使用正则表达式执行搜索	137	9.3	深入学习备份	183
7.6	修改数据	138	9.4	恢复单个数据库或集合	184
7.6.1	更新数据	139	9.4.1	恢复单个数据库	185
7.6.2	修改操作符	140	9.4.2	恢复单个集合	185
7.6.3	使用 save()函数快速保存文档	144	9.5	自动化备份	185
7.6.4	以原子的方式修改文档	145	9.5.1	使用本地数据存储	185
7.6.5	使用参数	145	9.5.2	使用远端数据存储(基于云)	187
7.7	删除数据	146	9.6	备份大数据库	188
7.8	在两个文档之间创建链接	146	9.6.1	使用隐藏的辅助服务器备份 数据	189
7.9	小结	149	9.6.2	使用日志文件系统创建快照	189
第8章 高级查询 151			9.6.3	使用卷管理器时的磁盘布局	191
8.1	文本搜索	151	9.7	将数据导入 MongoDB	191
8.1.1	文本搜索的代价和限制	152	9.8	从 MongoDB 导出数据	192
8.1.2	启用文本索引	152	9.9	通过限制对 MongoDB 服务器的 访问保护数据安全	194
8.1.3	使用文本索引	153	9.10	使用认证保护服务器	194
8.1.4	其他语言中的文本索引	158	9.10.1	添加 admin 用户	195
8.1.5	文本索引的复合索引	159	9.10.2	启用认证	195
8.2	聚集框架	161	9.10.3	在 mongo 控制台中执行认证	195
8.2.1	\$group	162	9.10.4	MongoDB 用户角色	196
8.2.2	\$sum	162	9.10.5	代理凭证	197
8.2.3	\$limit	163	9.10.6	修改用户凭证	197
8.2.4	\$match	164	9.10.7	添加只读用户	198
8.2.5	\$sort	165	9.10.8	删除用户	198
8.2.6	\$unwind	166	9.10.9	在 PHP 应用中认证连接	198
8.2.7	\$project	167	9.11	管理服务器	199
8.2.8	\$skip	168	9.11.1	启动服务器	199
8.3	MapReduce	170	9.11.2	重新配置服务器	200
8.3.1	MapReduce 的工作方式	170	9.11.3	获得服务器版本	201
8.3.2	创建测试文档	170	9.11.4	获得服务器状态	201
8.3.3	使用 map 函数	170			
8.3.4	高级 MapReduce	173			

9.11.5	关闭服务器	203	10.7	优化小对象的存储	229
9.12	使用 MongoDB 日志文件	203	10.8	小结	230
9.13	验证和修复数据	204	第 11 章 复制		231
9.13.1	修复服务器	204	11.1	MongoDB 复制的目标	231
9.13.2	验证单个集合	206	11.1.1	改善可扩展性	231
9.13.3	修复集合验证错误	207	11.1.2	改善持久性/可靠性	232
9.14	升级 MongoDB	209	11.1.3	提供隔离性	232
9.15	MongoDB 的滚动升级	209	11.2	复制基础	232
9.16	监控 MongoDB	209	11.2.1	主服务器的定义	233
9.17	使用 MongoDB 管理服务	211	11.2.2	辅助服务器的定义	233
9.18	小结	211	11.2.3	仲裁服务器的定义	234
第 10 章 优化		213	11.3	深入学习 oplog	234
10.1	优化服务器硬件	213	11.4	实现复制集	235
10.1.1	了解 MongoDB 使用内存的方式	213	11.4.1	创建复制集	236
10.1.2	了解工作集大小	214	11.4.2	启动服务器成员	237
10.1.3	选择正确的数据库服务器硬件	214	11.4.3	向复制集中添加服务器	238
10.2	评估查询性能	214	11.4.4	添加仲裁服务器	241
10.2.1	MongoDB 分析器	215	11.4.5	复制集链	242
10.2.2	使用 explain() 分析特定的查询	217	11.4.6	管理复制集	242
10.2.3	使用分析器和 explain() 优化查询	218	11.4.7	为复制集成员设置选项	246
10.3	管理索引	222	11.4.8	从应用连接到复制集	248
10.3.1	显示索引	222	11.5	小结	251
10.3.2	创建简单的索引	223	第 12 章 分片		253
10.3.3	创建复合索引	223	12.1	了解分片的需求	253
10.4	指定索引选项	224	12.2	对数据进行水平和垂直分区	254
10.4.1	使用 {background: true} 在后台创建索引	224	12.2.1	对数据进行垂直分区	254
10.4.2	使用 {unique: true} 创建唯一键索引	225	12.2.2	对数据进行水平分区	254
10.4.3	使用 {dropdups: true} 自动去重	225	12.3	分析一个简单的分片场景	255
10.4.4	使用 {sparse: true} 创建稀疏索引	226	12.4	使用 MongoDB 实现分片	256
10.4.5	TTL 索引	226	12.4.1	创建分片设置	258
10.4.6	文本索引	227	12.4.2	在集群中添加新的分片	260
10.4.7	删除索引	227	12.4.3	从集群中移除分片服务器	262
10.4.8	重建集合索引	227	12.4.4	确定连接的方式	263
10.5	MongoDB 选择索引的方式	228	12.4.5	列出分片服务器的状态	263
10.6	使用 hint() 强制使用特定的索引	229	12.4.6	使用复制集实现分片	264
			12.5	均衡器	264
			12.6	哈希片键	266
			12.7	标签分片	267
			12.8	小结	269

第 I 部分



MongoDB 基础

第 1 章 MongoDB 简介

第 2 章 安装 MongoDB

第 3 章 数据模型

第 4 章 使用数据

第 5 章 GridFS

第 1 章

MongoDB 简介

想象一下这样的世界：数据库使用是如此的简单，以至于你忘记了正在使用它。再想象一下这样的世界：不需要任何复杂配置或设置，数据库仍然能够快速运行，并且具有良好的扩展性。想一下，如何可以只关注 gg 于手上的任务，完成它，并可以按时下班。这听起来有点神奇，但是 MongoDB 承诺帮助你完成所有这些事情(甚至更多)。

MongoDB(源自单词 humongous)是一种相对较新的数据库，它没有表、模式、SQL 或行的概念。它没有事务、ACID 兼容性、连接、外键或其他许多容易在凌晨引起问题的特性。简单地说，MongoDB 是一个非常特别的数据库，它不同于你之前所使用的数据库，特别是关系数据库管理系统(Relational DataBase Management System, RDBMS)。事实上，当你知道 MongoDB 缺少对这些所谓的“标准”特性的支持时，你甚至可能已经在摇头了。

不要担心！在接下来的内容中，你将学习 MongoDB 的背景和指导原则，以及 MongoDB 团队这样做的原因。我们也将浏览 MongoDB 的特性列表，并提供足够的细节，从而保证你会完全沉迷于本书其余部分的话题中。

首先我们将了解创建 MongoDB 背后的哲学和理念，以及一些有趣的和有争议的设计决策。我们将学习面向文档数据库的概念、文档的组织方式以及它们的优缺点。我们还将学习 JSON 以及如何在 MongoDB 中应用 JSON。最后，我们将学习一些 MongoDB 最引人注目的特性。

1.1 了解 MongoDB 哲学

如同所有的项目一样，MongoDB 有一套自己的设计哲学用于帮助指导开发。本节内容将介绍一些 MongoDB 数据库的基本原则。

1.1.1 使用正确的工具处理正确的工作

MongoDB 中最重要的哲学概念是：一鞋难合众人脚。在过去的许多年中，传统的关系(SQL)数据库(MongoDB 是面向文档的数据库)一直被用于存储所有类型的数据。无论该数据是否符合关系模型(被用在所有的 RDBMS 数据库中，例如 MySQL、PostgreSQL、SQLite、Oracle、SQL Server 等)都无所谓；无论如何，数据都将被填充到数据库中。一般来说，部分原因是因为读取和修改数据库相比操作文件系统更加简单(和安全)。如果选择一本 PHP 方面的书籍，例如 *PHP for Absolute Beginners*, by Jason Lengstorf (Apress, 2009)，它将会教你使用数据库存储信息，而不是文件系统。这样做只是因为它更简单。在使用数据库存储信息的时候，开发者必须一直遵守它的工作流程。很明显，我们并未按照数据库原有的意图使用它。任何尝试在数据库中存储复杂数据、创建 5 张表，然后将它们组合在一起的开发者，都会明白我们在讲什么。

PDF电子书说明:

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 **QQ: 461573687**, 或者 **QQ: 2404062482**。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢!

MongoDB 团队决定他们不会创建另一个试图为所有人做所有事情的数据库。相反，该团队希望创建一个只用于处理文档的数据库，而不是行，并且它的速度要快，还要具有强大的扩展性和易用性。为了实现这个目标，他们不得不忽略一些特性，这意味着 MongoDB 在某些特定的情况下并不是最理想的选择。例如，它缺少事务支持，意味着无法使用 MongoDB 编写财务应用。也就是说，MongoDB 可能对于之前提到的部分应用(例如存储复杂数据)是非常合适的。不过这不是个问题，因为你完全可以在财务模块中使用传统的 RDBMS，而使用 MongoDB 存储文档。这样的混合解决方案非常常见，并且一些产品级应用(例如 New York Times 网站)已经在这样做了。

一旦适应 MongoDB 可能无法解决所有问题的理念之后，你会发现对于某些问题，MongoDB 可以完美地解决它们，例如分析(例如网站中使用的实时 Google Analytics)和复杂数据结构(例如博客文章和评论)。如果你仍然无法接受 MongoDB 是一个正式的数据库工具这个观点，那么请提前跳到 1.3 节“了解特性列表”，该部分内容将展示 MongoDB 的一些强大特性。

注意：

缺少事务和其他传统数据库特性并不意味着 MongoDB 不稳定，或者不能用于管理重要数据。

MongoDB 设计背后的另一个关键概念是：数据库应该一直具有多个副本。如果单个数据库实例出现问题，那么它可以轻松地通过另一个服务器恢复到正常状态。因为 MongoDB 的目标是尽可能地快，所以它采取了一些捷径，导致它难以从系统崩溃中恢复。开发者认为最严重的系统崩溃可能就是从服务中移除一台计算机；这意味着即使数据库完全恢复了，也无法正常使用。记住：MongoDB 不会尝试为所有人完成所有事情。但对于许多目的(例如构建 Web 应用)，MongoDB 是一个能够实现解决方案的完美工具。

现在你应该已经明白了 MongoDB 的起源。它不会尝试在所有方面都表现完美，也乐于承认它不会适用于所有人。不过，对于选择使用它的开发者，MongoDB 提供了一个功能丰富的面向文档数据库，并且对运行速度和扩展性做了优化。它也几乎可以运行在任何目标上。MongoDB 的网站上包含了可运行在 Linux、Mac OS、Windows 和 Solaris 中的安装文件。

MongoDB 成功实现了这些目标，因此使用 MongoDB 有点像梦幻一样(至少对于我们来说)。不必担心如何将数据压缩到一张表中，只需要将数据组合在一起，然后将数据传递给 MongoDB。考虑一个真实的例子。本书的合作者 Peter Membrey 最近开发了一个应用，用于存储一组 eBay 搜索结果。搜索结果的数量是不固定的(最多 100 个)，因此他需要一种简单的方式在数据库中将用户和搜索结果关联起来。

Peter 曾尝试使用 MySQL，他不得不设计出一张表用于存储数据，并编写相应代码存储他的结果，然后再编写代码将结果组合在一起。这是一个相当常见的场景，大多数开发者在开发中经常会遇到。通常，我们不得不这样做；不过，对于该项目，他决定使用 MongoDB，因此事情就变得有点不同了。

具体地说，他添加了下面这样两行代码：

```
request['ebay_results'] = ebay_results_array
collection.save(request)
```

在本例中，request 是 Peter 的文档，ebay_result 是键，而 ebay_result_array 包含来自 eBay 的搜索结果。第二行保存了修改后的数据。将来当他访问该文档时，他将获得与之前格式完全相同的数据。他不需要任何 SQL；也不需要执行任何会话；更不需要创建任何新表或编写任何特殊代码——MongoDB 就可以完成工作。他最终轻松地完成了工作，并按时回家。

1.1.2 天生缺少对事务的支持

MongoDB 开发者做出的另一个重要设计决定是：该数据库将不会包含事务的语义(用于保证数据一致性和存储的元素)。这是根据 MongoDB 的目标——简单、快速和可扩展做出的权衡之计。一旦移除这些重量级特性，数据库就可以更轻松地实现水平扩展。

通常在使用传统的 RDBMS 时，如果需要提高性能，就要购买更大、更快的机器。这是一种垂直扩展的方式，但只能做到这种程度。如果使用了水平扩展，就不需要使用一台大型机器，相反可以使用许多稍弱一些小机器。从历史上讲，这样的服务器集群非常有利于负载均衡网站，但因为数据库的内部设计限制，这种方式一直存在着问题。

你可能会认为缺少事务的支持将构成致命缺陷；不过，许多人都忘了一种在 MySQL 中最流行的表类型(MYISAM——这也恰好是默认的表类型)也不支持事务。但这个事实并未阻止 MySQL 在之前的 10 年中变成并保持着主流开源数据库的地位。与开发解决方案时的大多数选择一样，是否使用 MongoDB 取决于个人的偏好，以及它是否符合你的项目。

注意：

MongoDB 在同时使用至少两台服务器时可以提供持久性，这也是为生产环境部署时推荐使用的基本配置。主服务器可以在复制服务器确认接收到数据后，再确认数据已被接受。

尽管目前单服务器的持久性是无法保证的，但在将来可能会改变，在目前这也是一个热门领域。

1.1.3 JSON 和 MongoDB

JSON(JavaScript Object Notation, JavaScript 对象记号)不止是一种交换数据的方式，它也是一种存储数据的良好方式。RDBMS 是高度结构化的，包含了多个文件(表)用于存储不同的数据。而 MongoDB 将在单个文档中存储所有的数据。MongoDB 如同 JSON 一样，该模型为数据存储提供了丰富和高效的方式。而且，JSON 可以高效地描述指定文档中的所有内容，所以不需要提前指定文档的结构。JSON 是没有模式的(不需要模式)，因为文档可以单独更新，或者独立于其他文档进行修改。另外，JSON 还通过将相关数据存储在同一位置的方式提供了出色的性能。

实际上，MongoDB 并未使用 JSON 存储数据，而使用由 MongoDB 团队开发的一种称为 BSON(二进制 JSON 的英文简称)的开放数据格式。大多数情况下，使用 BSON 取代 JSON 并不会改变处理数据的方式。BSON 通过使计算机更容易处理和搜索文档的方式，使 MongoDB 处理速度变得更快。BSON 还添加了一些标准 JSON 不支持的特性，包括为处理二进制数据添加类型的能力。在本章的 1.3.1 节“使用面向文档存储(BSON)”中将深入讲解 BSON。

RFC 4627 中描述了 JSON 的初始规范，它由 Douglas Crockford 制订。JSON 允许在简单的、可读文本格式中展现复杂的数据结构，使用它通常比阅读和理解 XML 要简单得多。如同 XML 一样，JSON 被设计为一种在 Web 客户端(例如浏览器)和 Web 应用之间交换数据的方式。由于它描述对象的丰富方式和简单性，大多数开发者都选择它作为交换数据的格式。

你可能会好奇复杂数据结构到底意味着什么。历史上，CSV(Comma-Separated Value, 逗号分隔值)曾用于交换数据(确实，这种方式在今天也仍然非常常见)。CSV 是一种简单的文本格式，使用换行符分隔不同行，使用逗号分隔不同字段。例如下面的 CSV 文件：

```
Membrey, Peter, +852 1234 5678
Thielen, Wouter, +81 1234 5678
```

人们可以看懂这些信息，并且迅速找到其中传递的信息。问题是，第 3 列的数字是电话号码还是传真号码？它们甚至可能是寻呼机的号码。为了避免这种不确定性，CSV 文件通常会提供头字段，添加在文件的第一行。下面的片段显示了之前样例的更多细节：

```
Lastname, Firstname, Phone Number
Membrey, Peter, +852 1234 5678
Thielen, Wouter, +81 1234 5678
```

这样就好多了。但现在假设 CSV 文件提到的某些人拥有多个电话号码。你可以添加另一个字段用作办公室电话号码，但当希望使用多个办公室电话号码时，你将遇到同样的问题。当希望处理多个邮件地址时也会遇到其他的问题。大多数人都会有多个邮件地址，这些地址无法被清晰地标注为家庭或工作地址。此时，CSV 就暴露出了它的限制。CSV 文件只适合用于存储扁平且没有重复值的数据。通常会出现这种情况，提供几个 CSV 文件，每个都包含不同的信息。然后将这些文件结合到一起(通常使用 RDBMS)组成完整的数据。例如，一家大型的零售公司在每天结束时，可能会从它的每个商店收到 CSV 文件格式的销售数据。这些文件将被结合到一起，这样公司才能看到某天的销售状况。这个处理并不简单，随着需要的文件数目增加，出错的概率也会增大。

XML 很大程度上解决了这个问题，但使用 XML 未免“杀鸡用牛刀”：可以工作，但更像是大题小做，因为 XML 是高度可扩展的。XML 并非定义特定的数据格式，而是定义如何定义数据的格式。如果需要交换复杂并高度结构化的数据，XML 是非常有用的；不过对于简单数据交换，它通常会导致过多的工作。事实上，该场景就是“XML 地狱”这个词的来源。

JSON 提供了一个折中方案。与 CSV 不同，它可以存储结构化的内容；但与 XML 也不同，JSON 使内容易于理解和使用。下面使用 JSON 显示之前样例中的内容：

```
{
  "firstname": "Peter",
  "lastname": "Membrey",
  "phone_numbers": [
    "+852 1234 5678",
    "+44 1234 565 555"
  ]
}
```

在这个版本的例子中，每个 JSON 对象(或文档)包含所有必需的信息。例如 `phone_numbers` 中包含一个含有不同数字的列表。该列表可以变成任意大小。还可以指定数字的具体类型，如下所示：

```
{
  "firstname": "Peter",
  "lastname": "Membrey",
  "numbers": [
    {
      "phone": "+852 1234 5678"
    },
    {
```

```
    "fax": "+44 1234 565 555"
  }
  ]
}
```

该版本的样例有了一些改进。现在可以清楚地看到每个数字代表的含义。JSON 具有丰富的表达力，尽管手动编写 JSON 也非常容易，但通常 JSON 都是由软件自动生成的。例如，Python 中包含一个称为 `json` 的模块，用于将已有的 Python 对象自动转换为 JSON 格式。因为许多平台都支持和使用 JSON，所以它是交换数据的理想选择。

在 JSON 中添加电话号码列表这样的元素时，事实上是在创建嵌入文档。在 JSON 中添加复杂的内容时其实就是在创建嵌入文档，例如添加列表(或数组，以使用 JSON 常用的术语)。例如，文档 `Person` 中可以内嵌几个 `Address` 文档。类似地，`Invoice` 文档中可以内嵌多个 `LineItem` 文档。当然，内嵌的 `Address` 文档也可以内嵌包含了电话号码的文档。

在决定如何存储信息时，决定是否使用嵌入文档。这通常被称为模式设计。称之为模式设计似乎有点奇怪，因为 MongoDB 被认为是没有模式的数据库。不过，虽然 MongoDB 不强迫创建模式或增强已有的模式，你仍然需要思考如何将数据融合在一起。第 3 章将会深入进行讲解。

1.1.4 采用非关系的方式

改进关系型数据库性能的方式是非常直接的：购买更大更快的服务器。直到无法购买更快的服务器之前，这种方式都可以正常工作。而达到瓶颈时，唯一的选择就是扩展至两台服务器。这听起来似乎很容易，但它是大多数数据库的绊脚石。例如，MySQL 和 PostgreSQL 都无法在两台服务器上同时运行单个数据库(两台服务器都能够读取或修改数据，通常被称为活跃/活跃集群)。尽管 Oracle 可以通过强大的实时应用集群(Real Application Clusters, RAC)架构实现，但如果希望使用该解决方案，那么你可能会希望使用抵押贷款——实施基于 RAC 的解决方案需要多台服务器、共享存储和多个软件许可。

你可能会好奇为什么在两个数据库上部署活跃/活跃集群这么难。因为在查询数据库时，数据库需要找到所有相关的数据，并将它们关联在一起。RDBMS 解决方案使用了许多巧妙的方式来提高性能，但它们都必须依赖于能否获得所有完整的数据。而这正是问题所在：当有一半数据在另一台服务器上时，这种方式是无法工作的。

当然，你可能只有一个小型的数据库，只是因为收到了大量的请求，所以希望通过共享负载的方式解决负载问题。遗憾的是，此时会出现另一个问题。需要保证写入第一台服务器的数据在第二台服务器上也是可用的。并且如果在两台服务器上同时更新数据，将会遇到另一个问题。例如，需要决定哪个更新才是正确的。可能遇到的另一个问题是：某人可能在第二台服务器上查询刚刚写入第一台服务器的信息，但是该信息尚未更新到第二台服务器。鉴于以上这些问题，你很容易就会明白为什么 Oracle 的解决方案这么贵——这些问题都很难解决。

MongoDB 通过一种非常聪明的方式解决了这些活跃/活跃集群问题，并完全避免这些问题的发生。回想一下，MongoDB 是以 BSON 文档格式存储数据的，所以数据是自包含的。尽管相似的数据文档被存储在一起，但各个文档之间并没有关系。这意味着，你需要的所有东西都在同一个地方。因为 MongoDB 查询将在文档中寻找特定的键和值，该信息可以轻松地被扩散到所有可用的服务器上。每台服务器都将检查该查询，并返回结果。这样，可扩展性和性能的提升几乎是线性的。作为额外好处，你不需要办理新的抵押贷款来实现该解决方案。

确实, MongoDB 并不提供主/主复制, 两台不同的服务器都可以接受这些请求。不过, 它支持分片, 允许将数据分散到多台机器中, 每台机器都将负责更新数据集的不同部分。这种设计的优势在于, 在某些解决方案支持两台主数据库时, MongoDB 已经可以扩展到数百台机器, 如同运行在两台机器上一样简单。

1.1.5 选择性能还是特性

性能是很重要的, 不过 MongoDB 同样提供了一个巨大的特性集。我们已经讨论了一些 MongoDB 未实现的特性, 你可能会怀疑 MongoDB 如何通过移除其他数据库中常见的特性来实现惊人的性能。不过, 现在还有一些类似的数据库系统可用, 它们也非常快, 但同样功能也很有限, 例如实现了键/值存储的数据库。

一个完美的例子就是 memcached。该应用用于提供高速数据缓存, 并且速度极快。使用它缓存网站内容时, 它可以帮助加速应用许多倍。它被应用于非常大的网站, 例如 Facebook 和 LiveJournal。

美中不足的是, 该应用有两个重大缺陷。首先, 它是基于内存的数据库。如果系统断电, 所有的数据都会丢失。其次, 并不能真正用于数据搜索; 只能请求特定的键。

这些听起来都是非常严重的限制; 然而, 要记住 memcached 被设计用来解决的问题。首先, memcached 是一个数据缓存。它并不负责持久数据存储, 只是为现有的数据库提供一个缓存层。在构建动态 Web 页面时, 通常会请求非常特殊的数据(例如当前最热门的 10 篇文章)。这意味着, 你可以向 memcached 请求该数据, 不需要执行查询。如果 cache 过期或者没有数据, 那么你将正常查询数据、构建数据, 然后将数据存储到 memcached 中以便将来使用。

一旦接受这些限制, 你会发现 memcached 通过实现一个非常有限的功能集, 提供了卓越的性能。这样的性能是传统数据库所无法比拟的。事实上, memcached 并不能替代 RDBMS。最重要的是它也不应该用于替代数据库。

与 memcached 相比, MongoDB 本身功能非常丰富。MongoDB 为了具有竞争力, 必须提供一组强大的功能, 例如搜索特定文档的能力。它还必须能够将文档存储在硬盘中, 这样在重启之后数据也不会丢失。幸运的是, 对于大多数 Web 应用和许多其他类型的应用, MongoDB 都提供了足够的功能, 具有强大的竞争力。

如同 memcached 一样, MongoDB 不是一个适合所有人的数据库。因此, 要实现应用预期的目标, 就必须作出一些权衡。

1.1.6 在任何地方均可运行数据库

MongoDB 以 C++ 编写, 因此迁移相对容易, 并且可以在任何位置运行该应用。目前, MongoDB 网站提供了针对 Linux、Mac OS、Windows 和 Solaris 的二进制包。在其他平台上, 还为 Fedora 和 CentOS 提供了不同的官方版本。尽管推荐使用官网提供的二进制包, 但你仍然可以下载源代码, 构建自己的 MongoDB。所有二进制包都提供了 32 位和 64 位版本。

警告:

32 位版本的 MongoDB 数据库大小被限制为小于等于 2GB, 因为 MongoDB 内部使用内存映射文件来实现高性能。在 32 位系统中任何大于 2GB 的文件都需要一些特殊的处理, 这样会降低处理速度, 也会使应用代码变得复杂。官方关于该限制的观点是: 64 位环境很容易获得; 因此, 增加代码的复杂性并不是很好的权衡之计。64 位版本的 MongoDB 可以实现所有的意图和目的, 并且不含任何限制。

MongoDB 适度的要求使它可以运行在高性能服务器或虚拟机上，甚至可以运行在基于云的应用中。通过保持事情简单，关注于速度和效率，无论将 MongoDB 部署在哪里，它都可以提供稳定的性能。

1.2 将所有组合在一起

在开始了解 MongoDB 的特性列表之前，首先需要了解一些基本的术语。开始使用 MongoDB 并不要求太多专业知识，许多特定于 MongoDB 的术语大致都可以被翻译成对应的 RDBMS 术语，这些术语你之前可能已经熟悉了。因此不要担心；我们将详细地解释每个术语。即使你不熟悉标准的数据库术语，也仍然可以轻松地明白它们。

1.2.1 生成或创建键

一个文档代表了 MongoDB 中的一个存储单元。在 RDBMS 中，存储单元将被称为行。不过，文档中包含的信息要远比行多，因为它们可以存储复杂的信息，例如列表、词典，甚至是词典的列表。在传统数据库中，它们的行是固定的，而 MongoDB 中的文档可以由任意数目的键和值组成(下一节中将进行详细讲解)。键只不过是一个标签，它大致相当于 RDBMS 中的列名。可以使用键引用文档中的数据。

在关系数据库中，必须能够通过某种方式唯一定位一条指定的记录；否则，将无法引用特定的行。为此，必须添加一个字段用于存储一个唯一值(称为主键)，或者一组能够唯一定位指定行的字段(称为复合主键)。

出于相同的原因，MongoDB 要求每个文档必须有唯一标识符；在 MongoDB 中，该标识符被称为 `_id`。除非为该字段指定某个值，否则 MongoDB 将为你创建唯一值。即使是在已经成熟的 RDBMS 数据库世界中，也存在着是应该自己提供唯一键还是由数据库提供的分歧。最近，由数据库创建唯一键的方式已经变得更加流行。

这是因为由人创建的唯一数字，例如汽车注册号码，有不断改变的坏习惯。例如，在 2001 年，英国实施了一套新的牌照系统，与之前的系统完全不同。碰巧，MongoDB 也可以完美地解决这个问题；不过，问题是如果使用车牌号码作为主键，就需要认真思考是否存在什么问题。当 ISBN(International Standard Book Number, 国际标准书号)从 10 位数字升级到 13 位时，也出现了类似的情况。

之前，使用 MongoDB 的大多数开发者似乎更喜欢创建自己的唯一键，由自己来维护键的唯一性。然而，现在人们更愿意使用 MongoDB 创建的默认 ID 值。不过，在使用 RDBMS 数据库时，选择哪种方式更多地取决于个人偏好。我们更愿意使用数据库提供的值，因为这意味着我们可以保证键是唯一的，并且是独立的。如前所述，其他人更愿意提供自己的键。

最终，你必须决定哪种方式更适合自己。如果有信心保证自己的键一定是唯一的(并且可能不会改变)，那么就可以使用。如果不确定键的唯一性或者不希望担心这件事情，那么最好还是使用 MongoDB 提供的默认键。

1.2.2 使用键和值

文档由键和值组成。下面是之前章节中讨论过的一个样例：

```
{
  "firstname": "Peter",
  "lastname": "Membrey",
  "phone_numbers": [
    "+852 1234 5678",
    "+44 1234 565 555"
  ]
}
```

键和值总是成对出现的。与 RDBMS 不同，RDMBS 中的所有字段必须有值，即使值是 NULL(有些矛盾的是，NULL 意味着未知)，MongoDB 不要求文档必须含有特定的值。例如，如果不知道某人的电话号码，那就不添加。对于此事，一个通俗的类比是名片。如果你有传真号码，那就填入自己的传真号码；不过如果没有，也不用写入“传真号码：没有”。相反，不填写即可。如果 MongoDB 中不含某个键/值对，那它就被认为是不存在的。

1.2.3 实现集合

集合有点类似于表，但它们不那么死板。集合非常像一个贴有标签的盒子。在家里，你可能会有一个贴着“DVD”的盒子，用于放置 DVD。这是合理的，但没有什么事情可以阻止你将 CD 或磁带放入该盒子中。在 RDBMS 中，表是严格定义的，只能够将指定的元素放入表中。在 MongoDB 中，集合就是一组类似元素的集合。其中的元素不必相似(MongoDB 本质上是非常灵活的)；不过，当我们开始学习索引和高级查询时，就会看到在集合中放置类似元素的优点。

你可以在一个集合中混合各种不同的元素，但几乎没有必要这么做。创建一个称为媒体的集合，那么家里所有的 DVD、CD 和磁带都应该放在其中。毕竟，这些元素有着共同的特性，例如艺术家名称、发布日期和内容。换句话说，是否应该将特定的文档存储在相同的集合中取决于应用的需求。至于性能方面，创建多个集合并不比只使用一个集合慢。记住：MongoDB 的目标是帮助你让生活变得轻松，所以你应该按照自己感觉正确的方式去实现。

最后但并非最不重要的是，集合可以按需求即时创建。尤其是，在第一次尝试保存文档时，MongoDB 将创建引用它的集合。这意味着可以按照需求即时创建集合(但并不是应该这么做)。因为 MongoDB 也允许动态地创建索引和执行其他数据库级别命令，所以可以利用该特性构建出一些非常动态的应用。

1.2.4 了解数据库

可能理解 MongoDB 中数据库的最简单方式就是将它看作一个集合的集合。如同集合一样，数据库可以按需创建。这意味着为每个客户创建一个数据库是很简单的，你的应用代码就可以做到。除了 MongoDB，也可以在数据库中这样做；不过，在 MongoDB 中以这种方式创建数据库是非常自然的。也就是说，只因为你可以通过这种方式创建数据库，并不意味着必须或应该这么做。同样，你拥有权力，如果希望执行它的话。

1.3 了解特性列表

在了解 MongoDB 的定义以及它的特点之后，现在开始了解它的特性列表。在数据库网站 www.mongodb.org/ 上有 MongoDB 的完整特性列表；一定要访问它们最新的列表。本章讲到的

特性列表涉及一些 MongoDB 幕后的内容，事实上在使用 MongoDB 时并不需要了解所有的特性。换句话说，你可以自由挑选希望阅读的特性！

1.3.1 使用面向文档存储(BSON)

我们已经讨论过面向文档的设计，还简单接触了 BSON。如之前学到的，JSON 使它更容易地以文档真正的形式存储和获取文档，有效地避免了任何映射或特有转换代码的需求。事实上，该特性也使 MongoDB 变得更容易扩展。

BSON 是一个开放的标准，在网址 <http://bsonspec.org/> 上可以找到它的规范。当人们听到 BSON 是 JSON 的二进制形式时，他们期望 BSON 占用的空间要比 JSON 少得多。不过，事实并不一定是这样的；在许多情况下，BSON 版本与相同的 JSON 相比要占用更多的空间。

你可能会想知道，到底为什么要使用 BSON。毕竟，CouchDB(另一个强大的面向文档数据库)使用的就是纯 JSON，因此值得怀疑到底需不需在 BSON 和 JSON 之间来回转换文档。

首先，要记住 MongoDB 的设计目标是快速，而不是节省空间。虽然这并不意味着 MongoDB 会浪费空间(它不会)；不过，如果处理数据的速度更快(它确实是这样的)，那么存储文档时的一点开销是完全可以接受的。简单地说，BSON 更易于遍历(即浏览)，遍历索引页非常快速。尽管比起 JSON，BSON 需要稍微多一些的硬盘空间，但这并不是问题，因为硬盘很便宜，而且 MongoDB 可跨机器扩展。这种情况下的折中方案还是很合理的：多占用一点硬盘空间，换来更好的查询和索引性能。

第二个关键的优点在于，BSON 的使用很简单，并且易于将 BSON 数据转换为编程语言的原生数据格式。如果以纯 JSON 方式存储数据，那就需要添加一个相对高级别的转换。对于大量的编程语言(例如 Python、Ruby、PHP、C、C++和 C#)，MongoDB 都有对应的驱动，它们的工作方式也稍有不同。使用简单的二进制格式，可以在各种语言中快速构建原生数据结构，而不需要首先对 JSON 进行处理。这将使代码更简单和快速，而它们都是 MongoDB 的目标。

BSON 也提供了对 JSON 的一些扩展。例如，通过 BSON 可以存储二进制数据，以及处理特定的数据类型。因此，BSON 可以存储任何 JSON 文档，但有效的 BSON 文档可能不是有效的 JSON。这并没有关系，因为每种语言都有自己的驱动，可完成数据和 BSON 之间的转换，而不需要使用 JSON 作为中间语言。

事实上，BSON 并不是影响你是否使用 MongoDB 的关键因素。如同所有伟大的工具一样，MongoDB 将静静地后台工作，完成它应做的事情。除了可以使用图形工具查看数据，你将一直使用自己的本地语言进行开发，让驱动来担心如何将数据存储到 MongoDB 中。

1.3.2 支持动态查询

MongoDB 支持动态查询，这意味着可以运行查询而不需要提前做出计划。这类似于在 RDBMS 中运行 SQL 查询。你可能会想知道为什么这也会作为特性列出来；因为这应该是所有数据库都支持的特性。

实际上不是。例如，CouchDB(通常被认为是 MongoDB 的最大“竞争对手”)并不支持动态查询，因为 CouchDB 提出了一种全新的思考数据的方式(诚然这令人兴奋)。传统的 RDBMS 中有静态数据和动态查询。这意味着数据的结构必须提前确定下来——必须先定义表，并且每一行都必须符合该结构。因为数据库提前知道数据的结构，所以它可以做出特定的假设和优化，从而使动态查询更快。

CouchDB 将这点发挥到了极致。作为一个面向文档的数据库，CouchDB 是无模式的，所以数据也是动态的。然而，这里的新理念是：查询是静态的。也就是说，在使用查询之前，要提前定义它们。

这听起来似乎并不坏，因为许多查询都可以轻松提前定义。例如，一个允许你搜索图书的系统，可能也会让你通过 ISBN 搜索图书。在 CouchDB 中，你将创建一个包含所有文档 ISBN 的列表的索引。当查询 ISBN 时，该查询是非常快速的，因为它并不需要搜索任何数据。无论何时在系统中添加新的数据，CouchDB 都将自动更新它的索引。

从技术上讲，可以在不创建索引的情况下查询 CouchDB；不过，在这种情况下，CouchDB 不得不在处理请求之前自己创建索引。如果系统中只有数百本书，那么不会有任何问题；不过，如果系统中包含成千上万本书，它将导致性能变差，因为每个查询都将创建一个索引(又一次)。为此，CouchDB 不推荐在生产环境中使用动态查询，也就是未被预定义的查询。

CouchDB 还允许为查询编写映射和规约函数。如果这听起来似乎有些麻烦，那么你一定是在一家好公司工作；CouchDB 的学习过程稍微难一些。公平地讲，对于 CouchDB，有经验的程序员可能可以快速地掌握它；不过，对于大多数人来说，学习过程都是很难的，他们可能不会使用该工具。

对于我们这些凡人来说，幸运的是 MongoDB 非常易于使用。本书将详细讲解如何使用 MongoDB，不过这里有一个简单的版本：在 MongoDB 中，只需要提供希望匹配的部分文档即可，MongoDB 将完成其他的部分。不过 MongoDB 可以做更多。例如，如果希望使用映射或规约函数，那么你会发现 MongoDB 已经支持了该功能。同时，你可以轻松地使用 MongoDB，不必知道之前介绍的所有工具。

1.3.3 为文档创建索引

MongoDB 对文档索引提供广泛的支持，这是一个在处理成千上万文档时，使用起来非常方便的特性。没有索引，MongoDB 就必须按顺序查看所有的文档，检查它们是否符合查询。这就像是在向图书管理员索要特定的图书，他将在图书馆查看每一本书来寻找。在使用索引系统之后(图书馆倾向于使用 Dewey Decimal 系统)，他可以轻松地找到该书所属的区域，然后快速地确定它是否在那片区域。

与图书馆图书不同，MongoDB 中的所有文档都将会在 `_id` 键上自动创建索引。该键被认为是一个特殊的例子，因为你不能删除它；该索引用于保证每个值都是唯一的。使用该键的优点之一是：可以确定每个文档都是唯一可识别的，RDBMS 并不会保证这一点。

当你创建自己的索引时，可以决定是否希望它们具有强制唯一性。如果决定要创建唯一索引，可以告诉 MongoDB 删除所有重复的数据。这可能是也可能不是你所希望的，因此在使用该选项之前要进行认真思考，因为你可能一不小心就删除了一半的数据。默认情况下，如果在一个具有重复值的键上创建唯一索引，将会返回一个错误。

在许多情况下，你会希望创建一个允许重复的索引。例如，如果应用是按照姓氏搜索的，那么在姓氏键上创建索引是合理的。当然，这里无法保证每个姓氏都是唯一的；并且在任何具有合理大小的数据库中，含有重复数据实际上是正常的。

不过，MongoDB 的索引能力并不止于此。MongoDB 还可以在嵌套文档上创建索引。例如，如果在地址键中存储了大量的地址，那么可以在 ZIP 或邮政编码上创建索引。这意味着，可以轻松地将基于任意邮政编码的文档提取出来，并且非常快速。

MongoDB 另外还支持复合索引。在复合索引中，可以使用两个或多个键用于构建索引。例如，你可能需要创建一个包含了姓氏和名字标签的索引。这时搜索全名将是非常快速的，因为 MongoDB 可以快速地定位到姓氏，然后同样快速地定位到名字。

第 10 章将对索引进行深入讲解，MongoDB 提供了所有索引相关的功能。

1.3.4 使用地理空间索引

索引中特别值得一提的一种就是地理空间索引。MongoDB 1.4 中引入了这个全新的特殊索引技术。通过使用该特性可以索引基于位置的数据，从而处理从给定坐标开始的特定距离内有多少个元素这样的查询。

随着使用基于位置数据的 Web 应用的增加，该特性在日常开发中的作用越来越重要。尽管现在，地理空间索引的应用仍然不够广泛；不过，如果你需要它，那么你会很高兴 MongoDB 已经支持它。

1.3.5 分析查询

通过一个内置的分析工具可以显示出 MongoDB 如何找到返回的文档。这非常有用，因为在许多情况下，通过添加一个索引就可以提高查询的性能。如果现在有一个复杂的查询，并且不确定到底为什么运行速度这么慢，那么查询分析器可以为你提供极具价值的信息。第 10 章将对 MongoDB 分析器进行详细讲解。

1.3.6 就地更新信息

当数据库更新一行数据时(对于 MongoDB 来说更新的是文档)，它有多种更新方式可供选择。许多数据库选择多版本并发控制(Multi-Version Concurrency Control, MVCC)方式，这种方式允许多用户看到不同版本的数据。因为在给定的事务过程中，这种方式可以保证数据不会被另一个程序改变，所以它非常有用。

这种方式不利的一面在于数据库需要追踪数据的多个副本。例如，CouchDB 提供了非常强大的版本控制，但这是以记录所有数据为代价的。这种方式虽然能够保证数据以一种健壮的方式存储，但也增加了复杂性并降低了性能。

而 MongoDB 将就地更新信息。这意味着(与 CouchDB 相比)，MongoDB 可以在数据所在的位置更新它们。这通常意味着不需要更新额外的空间，索引可以保持不变。

这种方法的另一个优点是，MongoDB 将执行懒写入。写入内存或从内存读取是非常快速的，但写入磁盘就会慢上几千倍。这意味着要尽可能地限制从磁盘读写数据。在 CouchDB 中这是不可能的，因为程序需要保证每个文档都被快速地写入磁盘。这种方式保证数据将被安全地写入磁盘，但也会对性能产生巨大影响。

MongoDB 只在必须的时候才向磁盘写入，频率大概是每秒一次左右。这意味着如果一秒内某个值被更新了许多次——如果将某个值用于页面计数或存活统计，这种情况并不少见——那么该值只会被写入一次，而不是 CouchDB 要求的几千次。

这种方式使 MongoDB 更快速，当然它也是权衡的结果。CouchDB 可能更慢一些，但它保证了数据将被安全地存储在硬盘中。MongoDB 就无法保证这一点，这就是为什么传统的 RDBMS 可能更适用于管理一些关键数据，例如计费或应收账款。

1.3.7 存储二进制数据

GridFS 是 MongoDB 在数据库中存储二进制数据的解决方案。BSON 支持在一个文档中存储最多 4MB 的二进制数据，这可能已经可以满足你的需求。例如，如果希望存储个人资料，比如图片或声音，那么 4MB 将超出你的需要。另一方面，如果希望存储电影剪辑、高品质音频剪辑或几百兆大小的文件，MongoDB 仍然可以做到。

GridFS 通过在 files 集合中存储文件的信息(称为元数据)来实现。数据本身被分成多块(称为信息块)存储在 chunks 集合中。这种方式使数据存储既简单又有扩展性；还使范围操作(例如获取文件的特定部分)变得简单。

通常来说，我们将通过编程语言对应的 MongoDB 驱动来使用 GridFS，所以并不需要清楚 GridFS 的细节。与其他所有 MongoDB 中的特性一样，GridFS 的目标也是快速和可扩展性。这意味着如果需要处理大数据文件，MongoDB 同样能够胜任。

1.3.8 复制数据

在讨论 MongoDB 背后的指导原则时，我们提到过 RDBMS 数据能够为数据存储的安全提供某种保证，而这是 MongoDB 所不具备的。由于一些原因，MongoDB 并未实现这些保证。第一，这些特性会降低数据库的速度。第二，它们将大大地增加程序的复杂性。第三，最常见的故障应该是硬件故障，这种情况下即使数据被安全地保存到硬盘中，数据也无法使用。

当然，这些原因都不意味着数据安全不重要。如果无法在需要的时候访问到数据，那就不会有人愿意使用 MongoDB。刚开始，MongoDB 提供了包含主从复制特性的安全网络，这种情况下只有一个数据库可以处于活跃状态，并且可以在任何时间写入，这种方式在 RDBMS 世界中相当常见。该特性已经被复制集替代，基本的主从复制已经被弃用，也不应该再使用。

复制集有一台主服务器(类似于主从复制中的主服务器)，它将处理所有来自于客户端的请求。因为在指定的复制集中只有一台主服务器，它可以保证所有的写入都会被正确处理。当写入操作发生时，该操作也会被写入主服务器的“oplog”集合中。

oplog 集合将被复制到辅助服务器(可能有许多)，并帮助它们将数据更新到与主服务器一致的状态。一旦主服务器出现故障，辅助服务器中的某一台将会成为主服务器，并负责处理所有来自客户端的写入请求。

1.3.9 实施分片

对于涉及大规模部署的应用，自动分片可能是 MongoDB 最重要和最常用的特性。

在自动分片场景中，MongoDB 将处理所有数据的分割和重组。它将保证数据进入正确的服务器，并以最高效的方式运行查询和重组结果。事实上，从开发者的角度来看，使用含有数百个分片的 MongoDB 数据库和使用单个 MongoDB 数据库并没有区别。该特性尚未正式应用在生产环境中；不过一旦就绪，它将从根本上提升 MongoDB 的可扩展性。

与此同时，如果刚刚开始构建第一个基于 MongoDB 的网站，那么可能单实例 MongoDB 就足够满足需求。如果希望构建下一个 Facebook 或 Amazon，那么你会很高兴使用这么一门规模可以无限扩展的技术。本书的第 12 章将对分片进行详细讲解。

1.3.10 使用映射和归约函数

许多人在听到术语 **MapReduce** 时会感到脊背发凉。而另一个极端，许多 **RDBMS** 对映射和归约函数的复杂性嗤之以鼻。它们对于某些人来说是可怕的，因为这些函数要求使用一种完全不同的方式来思考如何查询和排序数据，许多专业的程序员对映射和归约函数背后的概念也感到头疼。这些函数提供了一种极其强大的查询数据的方式。事实上，**CouchDB** 只支持这种方式，这也是为什么它有着陡峭的学习曲线。

MongoDB 并不要求使用映射和归约函数。事实上，**MongoDB** 只依赖于简单的查询语法，这种语法与 **MySQL** 中使用的类似。不过，对于希望使用该功能的人，**MongoDB** 也提供了对这些函数的支持。映射和归约函数以 **JavaScript** 编写并运行在服务器中。映射函数的工作是查找所有符合条件的文档。然后这些结果会被传递到归约函数，归约函数将处理这些数据。不过归约函数并不会返回一个文档的集合；而是返回一个新的文档，其中包含衍生出的信息。作为一般规则，如果你会在 **SQL** 中使用 **GROUP BY**，那么在 **MongoDB** 中就应该会使用映射和归约函数。

注意：

不应该将 **MongoDB** 的映射和归约函数看做 **CouchDB** 所采用方式的粗糙模仿。如果愿意的话，可以使用 **MongoDB** 的映射和归约函数取代 **MongoDB** 自有的所有查询支持。

1.3.11 全新的聚合框架

MapReduce 是一个非常强大的工具，但它有一个主要的缺点：不易于使用。许多数据库系统都被用于生成报表，在特定的 **SQL** 数据库中这种方式非常容易。如果希望对结果分组或者查找最大值和平均值，那么它可以轻松地表达出你的想法并获得查找结果。遗憾的是，在 **MapReduce** 中这并不容易，必须自己完成所有的事情。这通常意味着，对于简单的任务来说使用 **MapReduce** 存在着不必要的困难。

出于这个原因，**MongoDB Inc.**(之前的 **10gen**) 添加了聚合框架。它基于管道，并允许依次处理查询的各个部分，然后将它们按顺序串联起来，获得要查询的结果。这在保持了 **MongoDB** 面向文档设计的优点之外，还提供了高性能。

所以如果需要使用强大的 **MapReduce**，那么仍然可以使用它。如果只是希望完成一些简单的统计和数字运算，你一定会喜欢新的聚合框架。在第 4 和第 6 章将会详细讲解聚合框架和它的命令。

1.4 获取帮助

MongoDB 有一个伟大的社区，而且核心开发者都非常活跃和平易近人，他们通常都非常乐于帮助社区的其他成员。**MongoDB** 易于使用并拥有很棒的文档；另外，你不孤单，如果需要任何帮助，都可以轻松获得！

1.4.1 访问网站

寻找更新信息或帮助的第一个地方就是 **MongoDB** 网站(<http://www.mongodb.org>)。该网站将定时更新，包含 **MongoDB** 最新的优点。在该网站中，可以找到驱动、教程、样例、常见问题等。

1.4.2 与 MongoDB 开发者沟通

MongoDB 开发者通常会使用 Freenode 网站(www.freenode.net)上的 Internet Relay Chat (IRC), 频道为#MongoDB。MongoDB 的开发者位于纽约, 但他们通常会在这个频道中一直聊到深夜。当然, 开发者也需要一些休息时间(咖啡的作用只能坚持这么久!); 幸运的是, 还有来自于世界各地的知识渊博的 MongoDB 用户愿意伸出帮助之手。许多访问#MongoDB 频道的人并不是专家; 不过, 其中的气氛非常友好, 所以他们愿意待在这个频道中。随时可以加入#MongoDB 频道与其他人交流, 你可能会发现一些很棒的提示和技巧。即便你真的卡在某个地方, 也能够很快回到正轨。

1.4.3 剪切和粘贴 MongoDB 代码

Pastie(<http://pastie.org>)并不是一个严格意义上的 MongoDB 站点; 不过, 在#MongoDB 频道中待一会儿总会遇到它。在 Pastie 站点中可以剪切和粘贴(以此得名)一些输出或程序代码, 然后放到网上供其他人查看。在 IRC 中, 粘贴的多行文本可能是混乱的或难于阅读的。如果需要贴出一些代码(例如三行以上), 那么应该访问 <http://pastie.org>, 粘出你的内容, 然后将页面的链接粘贴到频道中。

1.4.4 在 Google 小组中寻找解决方案

MongoDB 有一个称为 mongodb-user 的 Google 小组(<http://groups.google.com/group/mongodb-user>)。这个小组是咨询问题或寻找答案的好地方。也可以通过电子邮件与小组交互。与短暂的 IRC 交流不同, Google 小组是一个非常长期的资源。如果真的希望参与到 MongoDB 社区中, 加入小组将是一个很好的开始。

1.4.5 利用 JIRA 跟踪系统

MongoDB 使用 JIRA 问题跟踪系统。可以访问跟踪网站 <http://jira.mongodb.org/>, 并且可以在网站提交遇到的问题或 bug。将这样的问题报告到社区是一件真正的好事。当然也可以搜索之前已存在的问题, 并且可以查看 MongoDB 的发展蓝图和下一版本的更新计划。

在尚未提交问题之前, 可以先访问 IRC 聊天室。你很快会发现自己的问题是否是个新的问题, 如果是, 你将会明白如何报告它。

1.5 小结

本章对 MongoDB 的优点进行了全面介绍。我们了解了 MongoDB 创建和开发背后的哲学和指导原则, 以及在实现这些理念时 MongoDB 开发者作出的权衡。另外, 还了解了 MongoDB 中使用的一些关键术语, 如何组合它们以及它们在 SQL 中对应的术语。

接着, 我们了解了 MongoDB 提供的一些特性, 包括如何使用和应该在哪里使用它们。最后, 我们对社区和可以寻求帮助的地方做了快速浏览, 你一定会需要它们的!

第 2 章

安装 MongoDB

第 1 章已经讲解了 MongoDB 提供的功能。本章将会讲解如何安装和扩展 MongoDB，以及如何使用个人最喜爱的编程语言与 MongoDB 进行交互。

MongoDB 是一个跨平台的数据库，在 MongoDB 网站(www.mongodb.org)上可以找到一个包含了大量可用安装包的列表。众多的可用版本会让你难于决定使用哪个版本。正确的选择可能依赖于服务器所使用的操作系统、服务器的处理器类型，以及你是愿意使用稳定版本，还是愿意使用尚在开发中但提供了一些让人兴奋的特性的新版本。你可能希望同时安装数据库的稳定版本和前瞻性版本。也可能还无法完全确定应该使用哪个版本。无论是哪种情况，请继续向下阅读！

2.1 选择版本

在查看 MongoDB 网站的下载区域时，你会看到对可供下载的安装包的简单概述。首先需要注意的是将要运行 MongoDB 的服务器的操作系统类型。目前，该网站为 Windows、各种类型的 Linux 操作系统、Mac OS 和 Solaris 都提供了预编译版本。

注意：

要记住很重要的一点，那就是 32 位产品与 64 位产品之间的区别。32 位和 64 位版本的数据库目前有着相同的功能，唯一的区别是：32 位版本将每个服务器的数据集总大小限制在 2GB 左右；64 位版本没有任何限制，所以在生产环境中应该优先使用 64 位版本。另外，不同的版本之间也有可能发生变化。

另外还需要关注 MongoDB 软件自己的版本：正式版、旧版和开发版。正式版表示它是最近可用的稳定版本。当新的通常也是改进或增强的版本发布之后，之前最稳定的版本就成了旧版。这种发布方式是稳定和可靠的，但通常包含的可用特性要稍微少一些。最后是开发版。开发版通常被认为是不稳定版本。该版本仍然在开发中，其中包含许多修改，包括重大的新特性。尽管还未得到充分的开发和测试，但 MongoDB 的开发者已经将其发布给公众用于测试或用作其他尝试。

了解版本号

MongoDB 使用的版本号方式为：奇数版本号代表开发版。换句话说，可以通过查看版本号的第二个号码判断该版本是开发版还是稳定版。如果第二个号码是偶数，那它就是稳定版。如果第二个号码是奇数，那它就是不稳定的版本或开发版。

下面详细讲解一下版本号包含的三部分数字：A、B 和 C。

- A，第一个数字(或者最左面的数字)：代表主版本，只有在完整的版本升级时才会改变。
- B，第二个数字(或者中间的数字)：代表发布版本，表示该版本是开发版还是稳定版。如果数字是偶数，代表是稳定版；如果数字是奇数，代表是不稳定的开发版。
- C，第三个数字(或者最右面的数字)：代表修订号，用于解决缺陷和安全问题。

例如，在编写本书时，MongoDB 网站上包含以下可用版本：

- 2.4.3(正式版)
- 2.2.4(旧版)
- 2.5.0(开发版)

2.2 在系统中安装 MongoDB

到目前为止，你已经了解了所有可用的 MongoDB 版本。下面开始学习如何在特定的系统中安装 MongoDB。目前两个应用于服务器的主要操作系统是 Linux 和 Microsoft Windows，所以本章将讲解如何在这两种操作系统中安装 MongoDB，首先从 Linux 开始。

2.2.1 在 Linux 中安装 MongoDB

基于 UNIX 的操作系统是用于托管服务的最流行选择，包括 Web 服务、邮件服务和数据库服务。在本章，我们将学习如何在流行的 Linux 发布版本 Ubuntu 中运行 MongoDB。

取决于个人需要，在 Ubuntu 中可以通过两种方式安装 MongoDB：通过仓库自动安装，或者手动安装。下面将详细讲解这两种方式。

1. 通过仓库安装 MongoDB

仓库是一个在线目录，其中包含许多软件。所有的安装包都包含必要的信息：版本号、先决条件和可能存在的兼容性。该信息非常有用，因为有时在安装一个软件包时要求必须首先安装另一个软件，那么此时就可以同时安装它们。

Ubuntu(另一个基于 Debian 的发行版本)中默认的可用仓库已经包含 MongoDB，但它们可能并不是最新的版本。因此，首先使用 `apt-get`(用于从仓库中安装软件的命令)查找自定义仓库。在仓库列表(/etc/apt/sources.list)中添加下面的内容：

```
deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen
```

接下来，需要导入 10gen 的公钥，用于对安装包进行签名，保证它们的一致性；可以通过使用 `apt-key` 命令完成：

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10
```

完成之后，需要告诉 `apt-get` 它已经包含新的仓库；可使用 `apt-get` 的 `update` 命令来完成：

```
$ sudo apt-get update
```

此行代码将使 `aptitude` 注意到之前手动添加的仓库。这意味着，现在可以使用 `apt-get` 安装 MongoDB 了。在 shell 中输入下面的命令：

```
$ sudo apt-get install mongodb-10gen
```

该命令将安装 MongoDB 的当前稳定版本。相反，如果希望安装 MongoDB 的任何其他版本，那就需要指定版本号。例如，为了安装 MongoDB 的不稳定版本(开发版)，请输入下面的命令：

```
$ sudo apt-get install mongodb-10gen=2.5.0
```

这就是所有的步骤。现在，MongoDB 已经安装完毕并且可供使用。

注意：

在已经运行旧版 MongoDB 的系统中执行 `apt-get update` 将会把 MongoDB 升级到可用的稳定版本。通过运行下面的命令可以阻止该行为：

```
echo"mongodb-10gen hold" | sudo dpkg --set-selections
```

2. 手动安装 MongoDB

接下来将讲解如何手动安装 MongoDB。鉴于使用 `aptitude` 自动安装 MongoDB 是如此简单，为什么还需要手动安装软件呢？对于初学者，你们要明白并非所有的 Linux 的发行版本都使用 `apt-get`。当然，它们中的许多都使用(主要包括那些基于 Debian Linux 的发行版本)，但仍然有一些不使用。另外，打包工作可能正在进行中，所以仓库中可能没有可用的版本。也可能仓库中并未包含希望使用的 MongoDB 版本。手动安装软件还提供了同时运行多个 MongoDB 版本的能力。

如果已经决定要使用哪个版本的 MongoDB，从它们的网站 <http://mongodb.org/downloads> 下载到个人主目录中。然后使用下面的命令解压安装包：

```
$ tar xzf mongodb-linux-x86_64-latest.tgz
```

该命令将把安装包中的所有内容解压到新目录 `mongodb-linux-x86_64-xxxx-yy-zz` 中；该目录位于你的当前目录中。该目录将包含许多子目录和文件。包含可执行文件的目录被称为 `bin` 目录。我们很快将介绍其中的应用都用于完成哪些任务。

到此为止，安装应用不再需要任何额外的操作。确实，手动安装 MongoDB 并不会花费更多的时间——取决于需要安装的内容，可能会更快。不过，手动安装 MongoDB 也确实有自身的缺点。例如，解压到 `bin` 目录中的可执行文件默认只能从 `bin` 目录执行。因此，如果希望运行 `mongod` 服务，就需要直接从之前提到的 `bin` 目录中运行。这个缺点也突出了通过仓库安装 MongoDB 的优点。

2.2.2 在 Windows 中安装 MongoDB

作为服务器软件(包括基于网络的服务)，Microsoft Windows 也是流行的选择。

Windows 中并不存在类似于 `apt-get` 的仓库应用，所以需要从 MongoDB 网站下载和解压软件包来运行。不需要经历任何设置过程；要安装软件，只需要下载安装包、解压后运行即可。

例如，假设已经决定要为 64 位的 Windows 2008 服务器下载最近的稳定版。首先将安装包 (`mongodb-win32-x86_64-x.y.x.zip`)解压到 C 盘根目录下。此时，所有需要做的就是打开一个命令提示符(Start | Run |cmd| OK)，然后进入到解压后的目录中：

```
>cd C:\mongodb-win32-x86_64-x.y.z\  
>cd bin\  
>
```


该命令将带你进入包含了 MongoDB 可执行文件的目录。一切就这么简单：如前所述，并不需要进行安装。

2.3 运行 MongoDB

终于要真正开始接触 MongoDB 了。之前已经学过如何获得适合个人需求和硬件的 MongoDB 版本，还学习了如何安装软件。现在终于要开始运行和使用 MongoDB 了。

2.3.1 先决条件

在启动 MongoDB 服务之前，需要为 MongoDB 创建一个数据目录用于存储文件。默认情况下，对于基于 UNIX 的系统(例如 Linux 和 OS X)，MongoDB 将数据存储在 /data/db 目录中；对于 Windows 系统，MongoDB 将数据存储在 C:\data\db 目录中。

注意：

MongoDB 不会自动创建这些目录，所以需要手动创建它们；否则，MongoDB 将运行失败并报出错误消息。另外，要确定设置了正确的权限：为了正确运行，MongoDB 必须有读写和目录创建权限。

如果希望使用 /data/db 或 C:\data\db 之外的目录，那么可以在运行服务时使用 --dbpath 标志告诉 MongoDB 目标目录。

一旦创建必需的目录并赋予正确的权限，就可以通过执行 mongod 命令启动 MongoDB 核心数据库服务。在 Windows 的命令提示符或 Linux 的 shell 中运行该命令。

2.3.2 研究安装目录布局

在成功安装或解压 MongoDB 之后，bin 目录(Linux 和 Windows)中将出现如表 2-1 所示的应用。

表 2-1 MongoDB 中包含的应用

应用	功能
-- bsondump	读取 BSON 格式的回滚文件的内容
-- mongo	数据库 shell
-- mongod	核心数据库服务
-- mongodump	数据库备份工具
-- mongoexport	导出工具(JSON、CSV、TSV)，不可靠的备份
-- mongofiles	操作 GridFS 对象中的文件
-- mongoimport	导入工具(JSON、CSV、TSV)，不可靠的恢复
-- mongooplog	从另一个 mongod 实例中更新 oplog 条目
-- mongoperf	检查磁盘 I/O 性能
-- mongorestore	数据库备份恢复工具
-- mongos	数据库分片进程

(续表)

应 用	功 能
--mongosniff	实时嗅探/跟踪 MongoDB 数据库的活动，只在类 UNIX 系统中可用
--mongostat	返回数据库操作的内容
--mongotop	跟踪/报告 MongoDB 的读/写活动
-- mongorestore	恢复/导入工具

注意：所有的应用都在 bin 目录中。

已安装的软件中包含 15 个应用(Microsoft Windows 中为 14 个)，可以与 MongoDB 数据库一起使用。其中两个“最重要的”应用是 mongo 和 mongod。通过 mongo 应用可以使用数据库 shell，该 shell 可以帮助完成任何 MongoDB 相关的任务。

mongod 应用将启动 MongoDB 服务或守护进程。在启动 MongoDB 应用时，还有许多可用的标志。例如，可以指定数据的存储位置(--dbpath)、显示版本信息(--version)，甚至打印一些系统诊断信息(--sysinfo)！在运行服务时可以使用--help 标志查看完整的选项列表。现在使用默认配置即可，在 shell 或命令提示符中输入 mongod 以启动服务。

2.3.3 使用 MongoDB shell

一旦创建数据库目录并成功启动 mongod 数据库应用，就可以启动 shell，体验 MongoDB 的强大之处。

启动 shell(UNIX)或命令提示符(Windows)，确保处于能够找到 mongo 可执行文件的正确位置。在命令提示符中输入 mongo 并按下回车键，启动 MongoDB shell。屏幕上将立即显示出一个空白的窗口和一个闪烁的光标(如图 2-1 所示)。女士们、先生们，欢迎使用 MongoDB！



图 2-1 MongoDB shell

如果使用默认的参数启动 MongoDB 服务，并使用默认的设置启动 shell，那么该 shell 将连接到本地机器上运行的默认数据库 test。该数据库将在连接到它时自动创建。这是 MongoDB 最强大的特性：如果尝试连接到一个不存在的数据库，MongoDB 将会自动创建它。这既可能是好处也可能是坏处，取决于你能否正确使用键盘。

小窍门：

在 MongoDB 网站上有一个在线演示 shell，可以用它尝试列表中的所有命令。

在进行下一步之前，例如实现适合特定语言的额外驱动，看一眼 MongoDB shell 中一些有用的命令会有很大的帮助(如表 2-2 所示)。

表 2-2 MongoDB shell 中的基本命令

命 令	功 能
show dbs	显示出可用数据库的名称
show collections	显示出当前数据库中的集合
show users	显示出当前数据库中的用户
use <db name>	将当前数据库设置为<db name>

小窍门:

可以在 MongoDB shell 中输入 help 命令来获得完整的命令列表。

2.4 添加额外的驱动

在成功地启动 MongoDB 并学习了如何使用它的 shell 之后,你可能会觉得是时候正式接触 MongoDB 的世界了。这句话可能只有部分是正确的;你可能希望使用自己喜欢的编程语言来查询或操作 MongoDB 数据库,而不是 shell。10gen 提供了多个官方驱动,社区也提供了许多可用的驱动。例如,在 MongoDB 网站上可以找到下列语言的驱动:

- C
- C++
- C#
- Erlang
- Java
- JavaScript
- Node.js
- Perl
- PHP
- Python
- Ruby
- Scala

本节将讲解如何为 MongoDB 实现对两种流行编程语言 PHP 和 Python 的支持。

小窍门:

现在有许多可用的 MongoDB 驱动是由社区创建的。在 MongoDB 网站上可以找到一个驱动动的列表: www.mongodb.org。

2.4.1 安装 PHP 驱动

PHP 是今天现有的最流行编程语言之一。该语言的主要目标是 Web 开发,并且它能够轻松地处理 HTML。因此使用 PHP 设计 Web 应用(例如博客、留言板或名片资料卡)是一个很好的选择。下面几节内容将讲解在安装和使用 MongoDB PHP 驱动时可用的选项。

1. 获得 PHP 版 MongoDB 驱动

与 MongoDB 一样, PHP 也是一个跨平台开发工具, 取决于目标平台的不同, 以 PHP 方式设置 MongoDB 所要求的步骤也不同。之前, 本章之前展示过如何在 Ubuntu 和 Windows 中安装 MongoDB; 在这里将采用相同的方式, 演示如何在 Ubuntu 和 Windows 中安装 PHP 驱动。

首先下载与个人操作系统对应的 PHP 驱动版本。启动浏览器, 访问网址 www.mongodb.org。在编写本书时, 网站上包含一个单独的菜单选项, 称为驱动。单击该选项将显示出一个现有可用语言驱动的列表, 如图 2-2 所示。

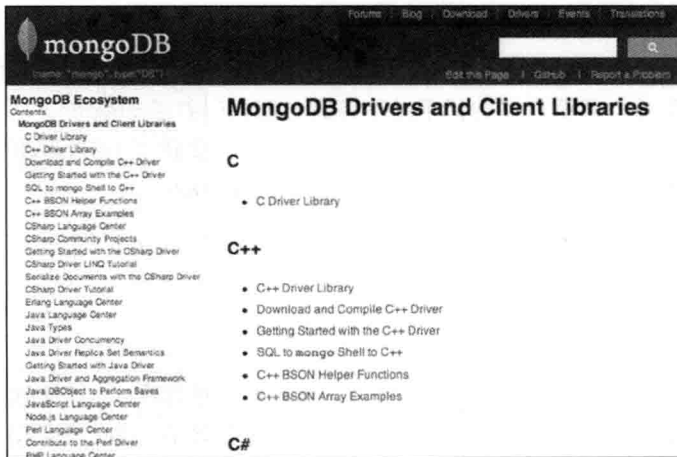


图 2-2 MongoDB 当前可用语言驱动的简短列表

接下来, 在列表语言中搜索 PHP, 然后单击链接进入下载页面, 下载稳定版本的驱动。不同的操作系统在自动安装 MongoDB 扩展时的方式是不同的。正如可以在 Ubuntu 中自动安装 MongoDB 一样, 也可以自动安装 PHP 驱动。当然, 同样也可以选择手动安装 PHP 语言驱动。下面将学习这两种可用的方式。

2. 在基于 UNIX 的平台上自动安装 PHP 驱动

PHP 的开发者提出了一个很棒的解决方案 PECL, 它允许使用其他流行的扩展来扩充 PHP。PECL 是专属于 PHP 的仓库; 它提供了一个目录用于存储所有已知的扩展, 可以用于下载、安装甚至开发 PHP 扩展。如果已经熟悉了称为 aptitude 的包安装系统(之前曾用它安装 MongoDB), 那你就发现 PECL 的接口与 aptitude 非常相似。

假设系统中已经安装了 PECL, 那么打开控制台, 输入下面的命令以安装 MongoDB 扩展:

```
$ sudo pecl install mongo
```

输入该命令之后, PECL 将自动为 PHP 下载和安装 MongoDB 扩展。换句话说, PECL 将下载 PHP 版本的扩展, 并将它放入 PHP 扩展目录中。这里只有一个问题: PECL 不能自动将扩展添加到已下载扩展列表中; 需要手动完成该步骤。打开文本编辑器(vim、nano 或任何你喜欢的文本编辑器), 修改文件 `php.ini`, 其中包含了 PHP 用于控制自己行为的主要配置, 包括它应该加载的扩展。

接着, 打开 `php.ini` 文件, 滚动到扩展区域, 添加下面的代码以告诉 PHP 加载 MongoDB 驱动:

```
extension=mongo.so
```

注意:

之前的步骤是必需的;如果不这样做, MongoDB 命令将无法在 PHP 中正常工作。可以在 shell 中使用 `grep` 命令 “`php -i | grep Configuration` 来查找系统中的 `php.ini` 文件”。

本章稍后的 2.4.2 节 “确认 PHP 安装正确” 将讲解如何确认扩展已被成功加载。

这就是全部步骤! 你已经成功地安装了 PHP 版本的 MongoDB 扩展, 可以开始使用它了。接下来学习如何手动安装驱动。

3. 在基于 UNIX 的平台上手动安装 PHP 驱动

如果由于某种原因不能使用 PECL 应用自动安装驱动(例如, 托管服务商可能不支持该选项)或者希望自己编译驱动, 那么可以选择手动下载驱动源代码并手动编译。

首先从 `github` 网站(<http://github.com>)下载驱动。该网站提供了 PHP 驱动最新的源代码包。在下载之后, 需要解压源代码包, 然后运行以下命令编译驱动:

```
$ tar zxvf mongodb-mongodb-php-driver-<commit_id>.tar.gz
$ cd mongodb-mongodb-php-driver-<commit_id>
$ phpize
$ ./configure
$ sudo make install
```

这个过程可能需要一点儿时间, 取决于个人计算机系统的速度。一旦该处理完成, MongoDB PHP 驱动就已经安装成功, 并可以开始使用! 在执行完命令之后, shell 中将显示出驱动的位置; 通常, 输出的内容应该如下所示:

```
Installing '/usr/lib/php/extensions/no-debug-zts-20060613/mongo.so'
```

需要确认该目录是否与 PHP 默认存储扩展的位置相同。可以使用下面的命令确认 PHP 存储扩展的位置:

```
$ php -i | grep extension_dir
```

该命令将输出所有 PHP 扩展所在的位置。如果该目录与 `mongo.so` 驱动所在的位置不符, 就必须将 `mongo.so` 驱动移到正确的目录中, 所以 PHP 需要知道在哪里能够找到它。

与之前一样, 需要告诉 PHP 新创建的扩展已经添加到它的扩展目录中, 并且它应该加载该扩展。可以通过修改 `php.ini` 文件的扩展区域来完成; 在该区域添加以下内容:

```
extension=mongo.so
```

最后, 必须重启 Web 服务。如果使用的是 Apache HTTPd 服务, 可以通过以下服务命令完成:

```
sudo /etc/init.d/apache2 restart
```

这就是全部步骤! 该过程比 PECL 自动安装方式稍微要长一点; 不过, 如果不能使用 PECL 或者你是一名驱动开发者并且对修复问题感兴趣, 那么你会希望使用手动安装方式。

4. 在 Windows 平台上安装 PHP 驱动

之前已经学习了如何在 Windows 操作系统中安装 MongoDB。现在学习如何在 Windows 系统中安装 MongoDB PHP 驱动。

对于 Windows 系统, MongoDB PHP 驱动的每个版本都有可用的预编译二进制包。可以从

之前提到的 `github` 网站(<http://github.com>)获得这些二进制包。这种情况下最大的挑战是为自己的 PHP 版本(有众多可用的包)选择正确的安装包。如果不确定需要使用哪个版本的包,可以在 PHP 页面中使用 `<? phpinfo(); ?>` 查看具体环境所需要的版本。下一节将详细讲解 `phpinfo()` 命令。

在下载正确的版本并解压出它的内容之后,将驱动文件 `php_mongo.dll` 复制到 PHP 扩展目录中即可;PHP 将可以正确地识别它。

取决于 PHP 的版本,扩展目录可能被称为 `Ext` 或 `Extensions`。如果不确定它使用的是哪个目录,可以查看系统中 PHP 包含的文档。

在将驱动 DLL 复制到 PHP 扩展目录之后,仍然需要告诉 PHP 加载该驱动。修改 `php.ini` 文件,在它的扩展区域添加下面的内容:

```
extension=php_mongo.dll
```

完成之后重启系统中的 HTTP 服务,接着就可以使用 MongoDB 的 PHP 驱动了。在开始体验 MongoDB PHP 驱动的神奇之前,需要确定 PHP 已经正确地加载了该扩展。

2.4.2 确认 PHP 安装正确

到目前为止,我们已经安装了 MongoDB 和 MongoDB 的 PHP 驱动。现在开始检查 PHP 是否已经成功地加载了该驱动。PHP 提供了一种简单且直接的方式来完成该操作:`phpinfo()`命令。该命令将显示出所有已加载模块的概述,包括版本号、编译选项、服务器信息、操作系统信息等。

打开文本编辑器或 HTML 编辑器,并输入下面的内容,用于执行 `phpinfo()`命令:

```
<? phpinfo(); ?>
```

接着,将文档保存在 Web 服务器的 `www` 目录中,任意名字均可。例如,可以称它为 `test.php` 或 `phpinfo.php`。现在打开浏览器并访问本机或外部服务器(访问正在使用的服务器),然后查看刚才创建的页面。页面上将显示所有 PHP 模块和所有其他各种相关信息的概述。这里需要关注的是显示出 MongoDB 信息的区域。该区域列出了它的版本号、端口号、机器名等,如图 2-3 所示。

mongo

MongoDB Support		enabled
Version		1.3.2

Directive	Local Value	Master Value
<code>mongo.allow_empty_keys</code>	0	0
<code>mongo.chunk_size</code>	262144	262144
<code>mongo.cmd</code>	\$	\$
<code>mongo.default_host</code>	localhost	localhost
<code>mongo.default_port</code>	27017	27017
<code>mongo.is_master_interval</code>	<i>no value</i>	<i>no value</i>
<code>mongo.long_as_object</code>	0	0
<code>mongo.native_long</code>	0	0
<code>mongo.ping_interval</code>	<i>no value</i>	<i>no value</i>
<code>mongo.utf8</code>	1	1

图 2-3 在 PHP 中显示出 MongoDB 信息

确认驱动已经加载成功之后,下面可以编写一些 PHP 代码,并利用 PHP 完成一些使用 MongoDB 的示例。

1. 通过 PHP 驱动连接 MongoDB 和断开连接

在确认 MongoDB PHP 驱动加载成功之后，现在开始编写一些 PHP 代码！首先从两个简单的基础选项开始：初始化 MongoDB 与 PHP 之间的连接，然后断开连接。

使用 `Mongo` 类初始化 MongoDB 和 PHP 之间的连接。同样，通过这个类可以使用数据库服务命令。一个简单而典型的连接命令如下所示：

```
$connection = new Mongo();
```

如果使用该命令而不使用任何参数，它将会连接本机 MongoDB 服务的默认端口(27017)。如果 MongoDB 运行在其他服务器上，那就需要指定希望连接的远程服务器：

```
$connection = new Mongo("example.com");
```

上述代码将初始化 MongoDB 服务的一个全新连接，该 MongoDB 服务运行在服务器上并一直在监听 `example.com` 域名(注意它将仍然连接到默认的端口 27017)。如果希望连接一个不同的端口(例如，不希望使用默认端口，或者已经在该端口上运行了 MongoDB 服务的一个会话)，可以通过指定端口和机器名的方式实现：

```
$connection = new Mongo("example.com:12345");
```

上述代码创建了 PHP 与数据库服务的一个连接。接下来，你将学习如何断开与服务之间的连接。假设使用刚才描述的方法创建了连接，那么可以再次调用 `$connection`，并使用 `close()` 命令来终止连接，如下所示：

```
$connection->close();
```

除了一些不正常的情况外，实际上并不需要调用该关闭方法。原因是一旦 `Mongo` 对象无效，PHP 驱动就会关闭与数据库的连接。然后，推荐在 PHP 代码的末尾调用 `close()` 方法；这将避免旧连接一直到超时才关闭。这也可以帮助确定所有现有连接都已被关闭，从而可以创建新的连接，如下所示：

```
$connection = new Mongo();
$connection->close();
$connection->connect();
```

下面的代码片段显示了如何在 PHP 中使用这些命令：

```
<?php
// Establish the database connection
$connection = new Mongo()

// Close the database connection
$connection->close();

?>
```

2.4.3 安装 Python 驱动

Python 是一门通用的并且易于阅读的编程语言。

这些特质使 Python 成为对编程和脚本初学者十分友好的语言。如果已经可以熟练完成编程工作，并且在寻找一种多范式编程语言——允许使用几种不同编程风格(面向对象编程、结构化

编程等), 那么 Python 对你来说也是一门很棒的语言。在接下来的内容中, 你将会学习如何安装 Python 和 MongoDB 的 Python 驱动。

1. 在 Linux 中安装 PyMongo

Python 添加了一个特定的包用于支持 MongoDB, 它被称为 PyMongo。通过该包可以与 MongoDB 数据库进行交互, 但在使用它之前, 需要安装并运行它。与安装 PHP 驱动时一样, 安装 PyMongo 也有两种方式: 依赖于 `setuptools` 的自动方式以及下载项目源代码进行安装的手动方式。下面将展示如何使用这两种方式安装 PyMongo。

自动安装 PyMongo

通过 `python-pip` 包中的 `pip` 应用可以自动下载、构建、安装和管理 Python 包。这非常方便, 将帮助完成扩展 Python 模块安装包的所有工作。

注意:

在使用 `pip` 应用之前必须首先安装 `setuptools`。在安装 `python-pip` 包时, 该操作会自动完成。

告诉 `apt-get` 下载和安装 `pip`, 如下所示:

```
$ sudo apt-get install python-pip
```

当该命令执行时, `pip` 将首先检测当前运行的 Python 版本, 然后将自己安装到系统中。这就是所有的步骤。现在就可以使用 `pip` 命令下载、编译和安装 MongoDB 模块, 如下所示:

```
$ sudo pip install pymongo
```

PyMongo 现在已经安装成功并且可供使用。

小窍门:

还可以使用 `pip` 命令安装 PyMongo 的旧版本: `pip install pymongo=x.y.z`。这里的 `x.y.z` 表示模块的版本。

手动安装 PyMongo

还可以选择手动安装 PyMongo。访问含有 PyMongo 插件的网站(<http://pypi.python.org/pypi/pymongo>)的下载区域。接下来, 下载 `tarball` 并解压。典型的下载和解压过程如下所示:

```
$ wget http://pypi.python.org/packages/source/p/pymongo/pymongo-2.5.1.tar.gz
$ tar xzf pymongo-2.5.1.tar.gz
```

在成功下载和解压文件后, 立即访问解压内容所在的目录, 然后使用 Python 运行 `install.py` 命令以安装 PyMongo:

```
$ cd pymongo-2.5.1
$ sudo python setup.py install
```

之前的代码片段输出 PyMongo 模块创建和安装的完整过程。最终, 该过程将返回到提示符, 此时就可以开始使用 PyMongo 了。

2. 在 Windows 中安装 PyMongo

在 Windows 中安装 PyMongo 是一个非常直接的过程。与在 Linux 中安装 PyMongo 一样，Easy Install 也可以简化 Windows 下 PyMongo 的安装过程。如果尚未安装 `setuptools`(其中包含 `easy_install` 命令)，那就访问 Python 的安装包索引网站(<http://pypi.python.org>)以寻找 `setuptools` 安装文件。

警告：

下载的 `setuptools` 版本必须与系统中已安装的 Python 版本匹配。

例如系统中安装的 Python 版本为 2.7.5，那么需要下载的 `setuptools` 包应为 `v2.7.x`。好消息是不需要编译它们；只需要下载正确的安装包，然后双击可执行文件在系统中安装 `setuptools` 即可！就是这么简单。

警告：

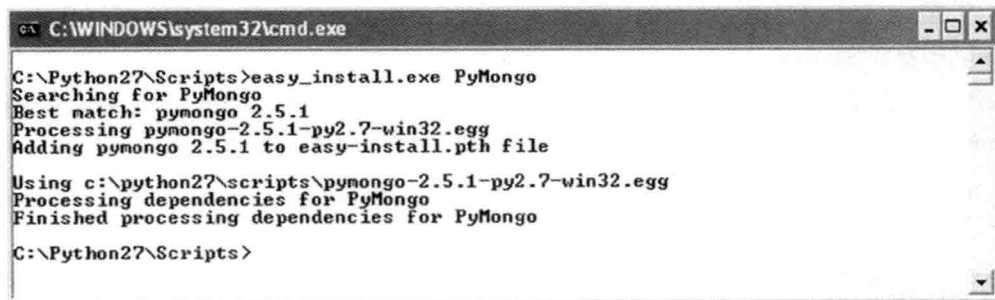
如果之前安装过旧版的 `setuptools`，那么在安装新版之前，需要使用系统的添加/删除程序功能卸载该版本。

安装完成之后，Python 的 `Scripts` 子目录中将出现 `easy_install.exe` 文件。此时就可以开始在 Windows 中安装 PyMongo 了。

在完成 `setuptools` 的安装之后，打开命令提示符，并切换到 Python 的 `Scripts` 目录。默认情况下，它的目录为 `C:\Pythonxy\Scripts\`，`xy` 代表 Python 的版本号。接着使用与之前安装 UNIX 版本时相同的命令，如下所示：

```
C:\Python27\Scripts>easy_installPyMongo
```

与在 Linux 中安装程序时得到的输出不同，这里的输出相当简洁。该输出只表示该扩展已经下载和安装完成(如图 2-4 所示)。在此情况下，表示该信息已经足够满足需要。



```
C:\WINDOWS\system32\cmd.exe
C:\Python27\Scripts>easy_install.exe PyMongo
Searching for PyMongo
Best match: pymongo 2.5.1
Processing pymongo-2.5.1-py2.7-win32.egg
Adding pymongo 2.5.1 to easy-install.pth file

Using c:\python27\scripts\pymongo-2.5.1-py2.7-win32.egg
Processing dependencies for PyMongo
Finished processing dependencies for PyMongo
C:\Python27\Scripts>
```

图 2-4 在 Windows 中安装 PyMongo

2.4.4 确认 PyMongo 安装正确

为了确认 PyMongo 已经安装成功，可以打开 Python shell。在 Linux 中，可通过打开控制台并输入 `python` 来实现。在 Windows 中，单击 `Start | Programs | Python xy | Python(命令行)`，打开 Python shell。此时将进入 Python 的世界，如图 2-5 所示。

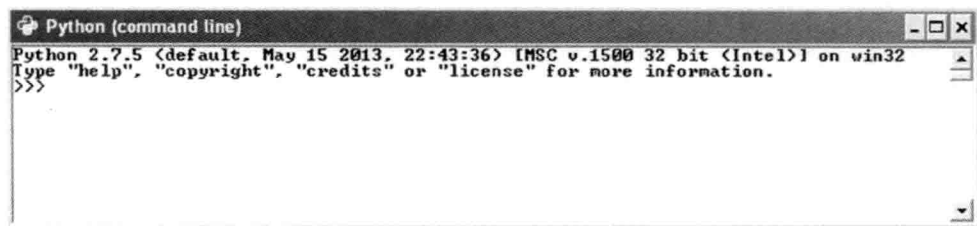


图 2-5 Python shell

可以使用 `import` 命令告诉 Python 开始使用新安装的扩展:

```
>>> import pymongo
>>>
```

注意:

每次希望使用 PyMongo 时都必须使用 `import pymongo` 命令。

如果一切进行顺利, 控制台不会再出现任何输出, 然后就可以开始使用 MongoDB 的命令了。不过, 如果收到错误消息, 表示什么地方出错了, 那就需要检查之前的步骤, 发现到底是什么地方出现了错误。

2.5 小结

本章首先讲解了如何获得 MongoDB 软件, 包括如何为自己的环境选择正确的版本。还学习了版本号的概念、如何安装和运行 MongoDB 以及如何安装和运行它所依赖的软件。接着, 学习了如何通过集合 PHP 和 Python shell 建立与数据库的连接。

另外, 你还学习了如何扩展 MongoDB, 使它能够与个人最喜欢的编程语言一起工作, 以及如何确定特定语言的驱动是否已经安装成功。

在下一章中, 你将学习如何正确地设计与构建 MongoDB 数据库和数据, 接着将学习如何对信息进行索引从而加快查询速度, 如何引用数据以及如何使用新特性地理空间索引。

第 3 章

数据模型

在之前的章节中,你已经学习了如何在两个常用的平台(Windows 和 Linux)上安装 MongoDB,以及如何使用某些额外的驱动扩展数据库。在本章,我们将把注意力从操作系统转移到 MongoDB 数据库的通用设计上。我们将学习集合的定义、文档的使用、索引的工作方式和通过它可以实现的目标,最后学习应该在什么时候和什么地方引用数据,而不是内嵌数据。在第 1 章中你已经简单地接触了其中的一些概念,本章将对这些概念进行深入讲解。本章将通过一些代码样例,展示这些正在讨论的概念。不要太担心其中的命令,第 4 章将会对它们进行详细讲解。

3.1 设计数据库

如同在前两章学到的一样, MongoDB 数据库是非关系数据库并且是无模式的。这意味着,不同于关系数据库(例如 MySQL), MongoDB 数据库并未绑定到任何预定义的列或数据类型。这种实现方式最大的优势在于,处理数据非常灵活,因为该文档不需要遵守任何预定义的结构。

简单地说:可以在一个集合中包含数百个甚至数千个结构不同的文档,而不会破坏 MongoDB 数据库的任何规则。

这种灵活的无模式设计的优势之一是,在使用动态类型语言(例如 Python 或 PHP)编程时不会受到限制。确实,如果由于数据库天生的限制,使一门极其灵活和动态的编程语言无法发挥出潜力,将是一个十分严重的问题。

下面让我们看一下 MongoDB 中文档的数据设计,尤其是要关注与关系数据库相比时, MongoDB 数据所体现出的灵活性。在 MongoDB 中,文档是一个包含了真正数据的元素,对应着 SQL 中的行。下面的例子将会展示出同一集合 **Media** 中的两个完全不同类型的文档(注意集合大致等同于 SQL 世界中的表):

```
{
  "Type": "CD",
  "Artist": "Nirvana",
  "Title": "Nevermind",
  "Genre": "Grunge",
  "Releasedate": "1991.09.24",
  "Tracklist": [
    {
      "Track": "1",
      "Title": "Smells Like Teen Spirit",
      "Length": "5:02"
    },
  ],
}
```

```
{
  "Track": "2",
  "Title": "In Bloom",
  "Length": "4:15"
}
]
}
{
  "Type": "Book",
  "Title": "Definitive Guide to MongoDB: A complete guide to dealing with Big Data using
MongoDB 2nd, The",
  "ISBN": "987-1-4302-5821-6",
  "Publisher": "Apress",
  "Author": [
    "Hows, David",
    "Plugge, Eelco",
    "Membrey, Peter",
    "Hawkins, Tim ]
}
```

如你所见，这两个文档的大多数字段相互之间都有联系。它们都拥有字段 `Title` 和 `Type`；但除了相似性，这两个文档是完全不同的。然后，它们被添加到了同一个集合 `Media` 中。

MongoDB 被称为无模式数据库，但并不意味着 MongoDB 的数据结构是完全没有模式的。例如，在 MongoDB 中也需要定义集合和索引(本章稍后将进行详细讲解)。然而，不需要为新增加的文档预定义任何结构，这与使用 MySQL 时是不同的。

简单地讲，MongoDB 是一个极其动态的数据库；之前的例子在关系数据库中完全无法正常工作，除非在表中添加了所有可能的字段。这样做会浪费空间和性能，更会引起高度混乱。

3.1.1 集合的更多细节

如前所述，集合是 MongoDB 中的一个常用术语。可以将集合看作存储文档的容器，如图 3-1 所示。

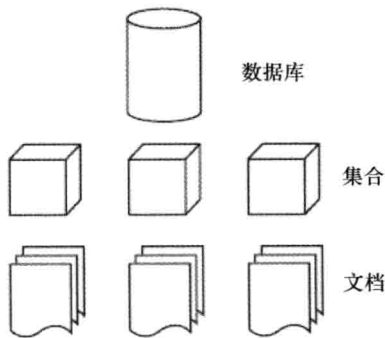


图 3-1 MongoDB 数据库模型

现在将 MongoDB 的数据库模型与关系数据库的典型模型做比较，如图 3-2 所示。

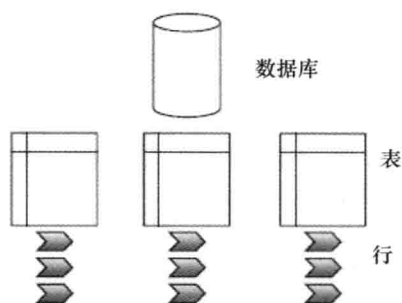


图 3-2 典型的关系数据库模型

如你所见，两种类型数据库的通用结果是相同的；不过，无法按照类似的方式使用它们。MongoDB 中有几种不同类型的集合。默认的集合按照大小进行扩展：添加的数据越多，集合就变得越大。还可以定义固定大小(capped)的集合。这些固定大小的集合只可以包含特定数量的数据，最老的文档将被新增加的文档替代(第 4 章将对这些集合进行深入讲解)。

MongoDB 中的所有集合都有唯一的名字。在使用 `createCollection` 函数创建集合时，其中的名字应该以字母或下划线(_)开头。名字中可以包含数字和字母；不过\$符号是 MongoDB 的保留关键字。类似地，其中不允许使用空白字符串(“ ”)；也不可以使用 `null` 字符，并且不可以以“system.”开头。通常，建议使用简短的集合名称(大约不超过 9 个字符)；不过 MongoDB 支持的集合名称的最大长度为 128。很明显，并没有任何实际的理由需要创建如此长的名称。

单个数据库默认最多可以创建 24000 个名称空间。每个集合至少包含两个名称空间：一个用于集合自身，另一个用于集合中创建的第一个索引。如果为每个集合添加更多的索引，将会使用更多的名称空间。这意味着从理论上讲，如果每个集合都只含有一个索引，那么每个数据库默认最多可以拥有 12000 个集合。不过在执行 MongoDB 服务应用(mongod)时，可以通过提供 `nssize` 参数增加名称空间的数目限制。

3.1.2 使用文档

文档由键值对组成。例如，“Type”: “Book”由键 Type 和值 Book 组成。键的类型为字符串，但可以使用许多不同类型的数据作为值。值可以是任何类型的数据，例如数组甚至是二进制数据。记住：MongoDB 使用 BSON 格式存储数据(第 1 章对该话题进行了详细讲解)。

接下来，学习所有可以添加到文档中的数据类型：

- **String**: 常用数据类型，包含一串文本(或任何其他种类的字符)。该数据类型常用于存储文本值(例如，“Country”: “Japan”)。
- **Integer(32 位和 64 位)**: 该数据类型常用于存储数值(例如，{"Rank":1})。注意在整数的前后没有引号。
- **Boolean**: 该数据类型的值要么为真，要么为假。
- **Double**: 该数据类型用于存储浮点数。
- **Min/Max keys**: 该数据类型分别用于与 BSON 中的最低和最高值作比较。
- **Arrays**: 该数据类型用于存储数组(例如，["Membrey, Peter", "Plugge, Eelco", "Hows, David"])。
- **Timestamp**: 该数据类型用于存储时间戳。可以方便记录文档修改或添加的时间。

- *Object*: 该数据类型用于存储嵌入文档。
- *Null*: 该数据类型用于存储 Null 值。
- *Symbol*: 该数据类型的用法与字符串一致；不过，通常该数据类型将被语言保留用于特定的符号类型。
- *Date **: 该数据类型用于存储 UNIX 时间格式的当前日期或时间(POSIX 时间)。
- *Object ID **: 该数据类型用于存储文档的 ID。
- *Binary data **: 该数据类型用于存储二进制数据。
- *Regular expression **: 该数据类型用于正则表达式。所有选项都通过按字母顺序提供的特殊字符表示。第 4 章将对正则表达式进行详细讲解。
- *JavaScript Code **: 该数据类型用于 JavaScript 代码。

最后 5 种带有星号的数据类型(日期、对象 ID、二进制数据、正则表达式和 JavaScript 代码)都不是 JSON 类型；它们是 BSON 中使用的特殊数据类型。在第 4 章中将学习如何使用 \$type 操作符识别数据类型。

理论上，这些听起来都非常直观。不过，你可能会好奇实际上究竟应该如何设计文档，以及在其中使用什么类型的数据。因为文档可以包含任何类型的数据，你可能会认为不需要引用另一个文档中的信息。在下一节中，将比较在文档中内嵌信息和引用另一个文档中的信息这两种方式的优劣。

3.1.3 在文档中内嵌或引用信息

可以选择在文档中内嵌信息，或者引用另一个文档中的信息。内嵌信息意味着在文档自身中添加某种类型的数据(例如包含了更多数据的数组)。引用信息意味着创建对另一个包含了特定数据的文档的应用。通常，使用关系数据库时会采用引用信息的方式。例如，假设希望使用关系数据库记录 CD、DVD 和图书的信息。在该数据库中，可能需要一张表用于存储 CD 信息，另一张表用于存储 CD 的曲目列表。因此，可能就需要查询多个表来获取某个 CD 中包含的曲目列表。

不过，在 MongoDB(和其他非关系数据库)中，内嵌此类型信息会更加简单。毕竟，文档天生能够实现这样的操作。采用这种方式将保持数据库简洁，保证所有相关的信息都存储在单个文档中，甚至因为数据在磁盘中存储位置相近，处理速度会更快。

现在通过一个真实的场景——在数据库中存储 CD 数据，演示内嵌信息和引用信息这两种方式的差别。

在关系数据库中，数据结构将如下所示：

```
|_media
  |_cds
    |_id, artist, title, genre, releasedate
  |_ cd_tracklists
    |_cd_id, songtitle, length
```

在非关系数据库中，数据结构将如下所示：

```
|_media
  |_items
    |_<document>
```

在非关系数据库中，文档结构将如下所示：

```
{
  "Type": "CD",
  "Artist": "Nirvana",
  "Title": "Nevermind",
  "Genre": "Grunge",
  "Releasedate": "1991.09.24",
  "Tracklist": [
    {
      "Track" : "1",
      "Title" : "Smells Like Teen Spirit",
      "Length" : "5:02"
    },
    {
      "Track" : "2",
      "Title" : "In Bloom",
      "Length" : "4:15"
    }
  ]
}
```

在本例中，曲目列表信息将内嵌在文档中。这种方式不仅高效，而且结构清晰。关于该 CD 希望存储的所有信息都将被添加到单个文档中。在该数据的关系数据库版本中，至少需要两个表；在非关系数据库中，只要求一个集合和一个文档。

当获取指定 CD 的信息时，只需要将单个文档的信息加载到内存中即可，而不是多个文档。记住，所有的引用都将在数据库中产生另一个查询。

小窍门：

使用 MongoDB 的经验法则是：尽可能地使用内嵌数据。这种方式要高效得多，并且总是可行的。

此时，你可能会好奇拥有多用户的应用是如何工作的。通常来说，之前提到的 CD 应用的关系数据库版本将要求至少使用一个表存储所有的用户以及另外两个表用于存储 CD 信息。对于非关系数据库，可以分别为用户和 CD 信息各使用一个集合。对于此类问题，MongoDB 允许通过两种方式创建引用：手动方式和自动方式。对于后一种情况，将需要使用 DBRef 规范，它提供了更多的灵活性，因为引用文档所在的集合可能会发生变化。第 4 章将对这两种方式进行详细讲解。

3.1.4 创建_id 字段

MongoDB 数据库中的所有对象都包含唯一标识符，用于区分不同的对象。该标识符被称为_id 键，它将在创建集合时自动添加到所有的文档中。

_id 键是添加到所有新创建文档中的第 1 个属性。即使不告诉 MongoDB 创建该键，它也会这么做。例如，之前样例中的代码都不曾使用_id 键。然而，MongoDB 自动在所有文档中都创建_id 键。它这么做是因为_id 键是集合中所有文档的必需元素。

如果不曾手动指定_id 值，那么它的类型将被设置为由 12 字节二进制数据组成的 BSON 数据类型。多亏有该设计，它具有唯一值的概率非常高。这个 12 字节的值包含 4 字节的时间戳(从 1970 年 1 月 1 日以来的秒数)、3 字节的机器 ID、2 字节的进程 ID 和 3 字节的计数器。计数器

和时间戳字段都以大端格式存储。这是因为 MongoDB 希望保证这些值能够按升序存储，而大端格式最符合此需求。

注意：

术语大端和小端指的是在内存中存储一个字的每个字节/位的方式。大端通常意味着最大的数字存储在开头。类似地，小端意味着最小的值存储在开头。

图 3-3 显示了 `_id` 键值的组成和每个部分的来源。

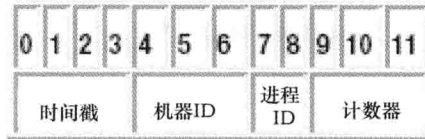


图 3-3 在 MongoDB 中创建 `_id` 键

访问 MongoDB 时使用的所有额外支持的驱动(例如 PHP 驱动或 Python 驱动)，都将支持这种特殊的 BSON 数据类型，并在创建新数据时使用它。还可以在 MongoDB shell 中调用 `ObjectId()` 以创建 `_id` 键值。另外，可以使用 `ObjectId(string)` 指定自己的 `_id` 键值，这里的 `string` 代表指定的十六进制字符串。

3.2 构建索引

如第 1 章提到的，索引不过是一种数据结构，用于收集集中文档特定字段的值的信息。MongoDB 的查询优化器使用该数据结构对集合中的文档进行快速排序。

记住，索引保证了在文档中查询数据的速度。基本上，可以将索引看作已经执行并存储了结果的预定义查询。在 MongoDB 中通用的经验规则是：对于需要在 MySQL 中创建索引的场景，在 MongoDB 中也应该创建索引。

创建索引的最大优点在于在查询常用信息时会非常快速，因为这些查询不需要遍历整个数据库以收集该信息。

一旦掌握索引，创建(或删除)索引相对就会很容易。第 4 章将学习如何完成这些操作。在第 10 章还将学习一些使用索引的高级技术，包括如何最大化性能。

使用索引对性能的影响

你可能会好奇为什么需要删除索引、重建索引甚至删除集合内的所有索引。最简单的答案是清除一些违规数据。例如，有时数据库的大小会毫无原因地剧烈增长。另外，索引也可能使用过多的空间。

另一件要记住的事：每个集合最多可以拥有 40 个索引。通常来说，这远超个人的需要，不过某天你有可能不小心就达到这个限制。

注意：

添加索引将提高查询速度，但也会降低插入或删除的速度。最好在读操作多于写操作的集合中添加索引。但当写操作多于读操作时，索引甚至可能降低性能。

最后，所有索引信息都存储在数据库的 `system.indexes` 集合中。例如，可以运行 `db.indexes.find()` 命令来查看目前已经存储的索引。为了查看某个集合中创建的索引，可以使用 `getIndexes` 命令：

```
db.collection.getIndexes()
```

3.3 使用地理空间索引

如第1章中简单介绍过的，MongoDB 从版本 1.4 开始就已经实现了对地理空间索引的支持。这意味着，除了普通的索引，MongoDB 还支持地理空间索引，可用于处理基于位置的查询。例如，可以使用该特性查找距离用户当前位置最近的已知目标。或者重新定义搜索，查询距离当前位置最近的餐馆。如果希望创建一个应用，希望根据指定的客户 ZIP 代码找到最近的分公司，那么这种类型的查询将特别有用。

要添加地理空间信息的文档必须含有一个子对象或数组(第一个元素指定了对象类型，紧接着是该元素的经度和纬度)，如下所示：

```
> db.restaurants.insert({name: "Kimono", loc: { type: "Point", coordinates: [ 52.370451, 5.217497]}})
```

注意，参数 `type` 可用于指定文档的对象类型，可以是 `Point`、`LineString` 或 `Polygon`。如同预料的一样，`Point` 类型用于指定某个条目(本例中为餐馆)所在的准确位置，因此需要两个值：经度和纬度。类型 `LineString` 可用于指定某个沿着特定路线扩展的条目(例如街道)，因此需要起点和终点，如下所示：

```
> db.streets.insert( {name: "Westblaak", loc: { type: "LineString", coordinates: [ [52.36881, 4.890286], [52.368762, 4.890021] ] } } )
```

`Polygon` 类型可用于指定(非默认的)图形(购物区域)。使用该类型时，需要保证起点和终点是一致的，从而可以闭合这个环。另外，该点的坐标将通过在数组中内嵌数组的方式提供，如下所示：

```
> db.stores.insert( {name: "SuperMall", loc: { type: "Polygon", coordinates: [ [ [52.146917, 5.374337], [52.146966, 5.375471], [52.146722, 5.375085], [52.146744, 5.37437], [52.146917, 5.374337] ] ] } } )
```

在大多数情况下，`Point` 类型都是适用的。

一旦在文档中添加地理空间信息，就可以创建此种类型的索引(当然也可以提前创建索引)，为 `ensureIndex()` 函数提供 `2dsphere` 参数：

```
> db.restaurants.ensureIndex( { loc: "2dsphere" } )
```

注意：

`ensureIndex()` 函数用于添加自定义函数。目前不用担心该函数的语法，在下一章中将详细讲解 `ensureIndex()` 函数。

参数 `2dsphere` 将告诉 `ensureIndex()`，它在索引坐标或类地球球体上的其他形式的二维信息。默认情况下，`ensureIndex()` 将假设提供的是经度和纬度，并认为它们的范围是 -180 到 180。不过，可以使用 `min` 和 `max` 参数覆写这些值：

```
> db.restaurants.ensureIndex( { loc: "2dsphere" }, { min : -500 , max : 500 } )
```

还可以通过使用辅助键值(也称为复合索引)扩展地理空间索引。如果希望查询多个值,例如位置(地理空间信息)或分类(升序),那么该结构是非常有用的:

```
> db.restaurants.ensureIndex( { loc: "2dsphere", category: 1 } )
```

注意:

此时的地理空间基于如下概念实现:地球是一个完美的球体。因此,每个经度和纬度的范围都是精确的 111 公里(69 英里)长。不过,只有在赤道上这才是正确的;越是远离赤道,每个经度的范围就变得越短,直至在两极变成 0。

查询地理空间信息

本章将主要关注两件事情:如何制定数据模型以及数据在应用的背后如何工作。操作地理空间信息在各种各样的应用中变得越来越重要,因此下面将讲解如何在 MongoDB 数据库中使用地理空间信息。

在开始之前,先提出一个温和的警告。如果你是一个 MongoDB 新手,之前也未曾使用过地理空间索引数据,本节内容的难度对于你来说会比较难。不过不要担心;现在完全可以忽略本节内容,在之后随时可以再回来学习。下面的样例将展示如何(以及为什么)在实际应用中使用地理空间索引,使本节内容变得易懂一些。通过这种方式,如果你觉得能够接受,那么请继续阅读。

在集合中添加数据并创建索引之后,可以执行地理空间查询。例如,下面的代码将显示如何使用地理空间索引。

首先打开 MongoDB shell,并使用 use 函数选择数据库。在本例中,数据库的名称为 restaurants:

```
> use restaurants
```

选择数据库之后,可以定义一些包含了地理空间信息的文档,并将它们插入到 places 集合中(记住:不需要提前创建集合):

```
> db.restaurants.insert( { name: "Kimono", loc: { type: "Point", coordinates: [ 52.370451, 5.217497 ] } } )
```

```
> db.restaurants.insert( { name: "ShabuShabu", loc: { type: "Point", coordinates: [51.915288, 4.472786] } } )
```

```
> db.restaurants.insert( { name: "Tokyo Cafe", loc: { type: "Point", coordinates: [52.368736, 4.890530] } } )
```

在添加数据之后,需要告诉 MongoDB shell 基于 loc 键指定的位置信息来创建索引,如下所示:

```
> db.restaurants.ensureIndex( {loc: "2dsphere" } )
```

创建索引之后,就可以开始搜索文档。首先搜索一个精确值(到目前为止仍是普通的查询;此时尚未涉及任何地理空间信息):

```
> db.restaurants.find( { loc : [52,5] } )
```

之前的搜索并未返回结果。这是因为该查询太具体了。本例中更好的方式应该是搜索某个包含接近指定值的信息的文档。可以使用 `$near` 操作符实现该操作。注意,这种方式要求指定 `type` 操作符,如下所示:

```
> db.restaurants.find( { loc : { $geoNear : { $geometry : { type : "Point", coordinates:
[52.338433,5.513629] } } } } )
```

上述代码产生的输出如下所示:

```
{
  "_id" : ObjectId("51ace0f380523d89efd199ac"),
  "name" : "Kimono",
  "loc" : {
    "type" : "Point",
    "coordinates" : [ 52.370451, 5.217497 ]
  }
}
{
  "_id" : ObjectId("51ace13380523d89efd199ae"),
  "name" : "Tokyo Cafe",
  "loc" : {
    "type" : "Point",
    "coordinates" : [ 52.368736, 4.89053 ]
  }
}
{
  "_id" : ObjectId("51ace11b80523d89efd199ad"),
  "name" : "ShabuShabu",
  "loc" : {
    "type" : "Point",
    "coordinates" : [ 51.915288, 4.472786 ]
  }
}
```

尽管该结果集看着好多了,但仍然存在着一个问题:所有的文档都被返回了!在不使用任何其他操作符的情况下, `$near` 将返回头 100 条记录,并按照它们与指定坐标的距离进行排序。现在,我们可以限制返回的结果。例如,使用 `limit` 函数返回开始的两条记录,或者只返回指定范围内的结果(这种方式更好)。

这可以通过添加 `$maxDistance` 操作符来实现。使用该操作符将告诉 MongoDB 只返回在从指定点开始的最大距离(按米计算)之内的结果,如下所示:

```
> db.restaurants.find( { loc : { $geoNear : { $geometry : { type : "Point", coordinates:
[52.338433,5.513629] }, $maxDistance : 40000 } } } )
{
  "_id" : ObjectId("51ace0f380523d89efd199ac"),
  "name" : "Kimono",
  "loc" : {
    "type" : "Point",
    "coordinates" : [ 52.370451, 5.217497 ]
  }
}
```

如你所见,该查询只返回了一个结果:从起点开始 40 公里(或者大约 25 英里)范围之内的一家餐馆。

注意:

返回结果的数目与执行查询所需的时间存在着直接的关系。

除了\$geoNear 操作符, MongoDB 还有\$geoWithin 操作符。可以使用该操作符寻找特定图形中的所有记录。这时候就可以查找位于\$box、\$polygon、\$center 和\$centerSphere 图形中的记录, \$box 代表矩形, \$polygon 代表选择的特殊图形、\$center 代表圆形、\$centerSphere 定义球体上的圆形。下面通过一些例子来演示它们的用法。

注意:

在 MongoDB 2.4 中, \$within 操作符被弃用, 取而代之的是\$geoWithin。该操作符并不严格要求使用地理空间索引。另外, 与\$near 操作符不同, \$geoWithin 将返回未排序的结果, 这提高了查询的性能。

为了使用\$box 图形, 首先需要指定矩形的左下角和右上角坐标, 如下所示:

```
> db.restaurants.find( { loc: { $geoWithin : { $box : [ [52.368549,4.890238], [52.368849,4.89094] ] } } } )
```

类似地, 为了查找特定图形中的记录, 需要指定一组包含了点坐标信息的嵌套数组。另外, 注意第一个和最后一个坐标必须是一致的, 用于正确地闭合图形所形成的环, 如下所示:

```
> db.restaurants.find( { loc :
  { $geoWithin :
    { $geometry :
      { type : "Polygon" ,
        coordinates : [ [
          [52.368739,4.890203], [52.368872,4.890477], [52.368726,4.890793],
          [52.368608,4.89049], [52.368739,4.890203]
        ] ]
      }
    }
  }
} )
```

查找\$circle 图形中记录的代码非常简单。在这种情况下, 只需要在执行 find()函数之前指定圆的中心和弧度(使用坐标系统使用的单位)即可:

```
> db.restaurants.find( { loc: { $geoWithin : { $center : [ [52.370524, 5.217682], 10 ] } } } )
```

注意, 自从 MongoDB 2.2.3 开始, \$center 操作符可以与非地理空间索引一起使用。不过, 还是推荐创建地理空间索引, 这样可以提高性能。

最后, 为了查找球体(例如地球)上某个圆形之内的记录, 可以使用\$centerSphere 操作符, 该操作符类似于\$center, 如下所示:

```
> db.restaurants.find( { loc: { $geoWithin : { $centerSphere : [ [52.370524, 5.217682], 10 ] } } } )
```

默认情况下, 使用 find()函数运行查询是非常理想的。不过, MongoDB 还提供了 geoNear()函数, 它的工作方式与 find()一样, 不过它还在结果中提供了从指定点到每个记录的距离。函数 geoNear()中还包含一些额外的诊断信息。下面的样例使用 geoNear()函数查找距离指定位置最近的两个结果:

```
> db.runCommand( { geoNear : "restaurants", near : { type : "Point", coordinates:
```

```
[52.338433,5.513629] }, spherical : true}}
```

它将返回以下结果:

```
{
  "ns" : "stores.restaurants",
  "results" : [
    {
      "dis" : 33155.517810497055,
      "obj" : {
        "_id" : ObjectId("51ace0f380523d89efd199ac"),
        "name" : "Kimono",
        "loc" : {
          "type" : "Point",
          "coordinates" : [
            52.370451,
            5.217497
          ]
        }
      }
    },
    {
      "dis" : 69443.96264213261,
      "obj" : {
        "_id" : ObjectId("51ace13380523d89efd199ae"),
        "name" : "Tokyo Cafe",
        "loc" : {
          "type" : "Point",
          "coordinates" : [
            52.368736,
            4.89053
          ]
        }
      }
    },
    {
      "dis" : 125006.87383713324,
      "obj" : {
        "_id" : ObjectId("51ace11b80523d89efd199ad"),
        "name" : "ShabuShabu",
        "loc" : {
          "type" : "Point",
          "coordinates" : [
            51.915288,
            4.472786
          ]
        }
      }
    }
  ],
  "stats" : {
    "time" : 6,
    "nscanned" : 3,
    "avgDistance" : 75868.7847632543,
    "maxDistance" : 125006.87383713324
  },
  "ok" : 1
}
```

以上是关于地理空间信息的所有介绍；在本书接下来的章节中还将有一些这样的例子，为大家展示如何使用地理空间函数。

3.4 在真实世界中使用 MongoDB

现在我们已经安装了 MongoDB 及其相关的插件，并且了解了它的数据模型。现在是时候开始真正使用它了。在接下来的 5 章内容中，你将会学习如何构建、查询和操作一些不同的 MongoDB 样例数据库(表 3-1 列出了将要学习的话题)。每章都将主要使用一个数据库，通过这种方式可以方便读者以模块化的方式阅读本书。

表 3-1 本书涉及的 MongoDB 样例数据库

所在章	数据库名称	话题
4	library	使用数据和索引
5	test	GridFS
6	contacts	PHP 和 MongoDB
7	inventory	Python 和 MongoDB
8	test	高级查询

3.5 小结

在本章内容中，我们学习了数据库在后台完成的工作。还深入学习了集合与文档的概念；并且学习了 MongoDB 中支持的数据类型，以及如何嵌入和引用数据。

接下来，我们学习了索引，包括何时以及为什么使用(或者不使用)它们。

我们还接触了地理空间索引的概念。例如，如何存储地理空间数据；如何使用普通的 find() 函数或基于 geoNear 数据库命令以更接近地理空间的方式搜索这样的结果。

在下一章中，我们将学习如何使用 MongoDB shell，包括使用哪些函数可以插入、查找、更新或删除数据。另外，还将学习如何结合条件操作符使用这些函数。

第 4 章

使用数据

在前一章，你已经学习了数据库在幕后完成的工作、索引的定义、使用数据库快速查询数据的方式以及文档的结构，你还看到了一些使用 MongoDB shell 完成数据添加和查询的样例。本章内容将关注于如何通过 shell 使用数据。

本章将只会用到一个名为 `library` 的数据库，我们将在其中执行一些操作，例如添加数据、搜索数据、修改数据、删除数据和创建索引。你还将学习如何使用各种命令浏览数据库，以及 `DBRef` 的定义和它所完成的工作。如果已经按照之前章节中的指令安装了 MongoDB 软件，那么接下来通过本章的样例你将会慢慢熟悉它。与此同时，你还将学会应该使用哪种命令处理对应的操作。

4.1 浏览数据库

首先需要知道的是如何浏览数据库和集合。在传统的 SQL 数据库中，需要做的第一件事是创建一个真正的数据库；不过，在 MongoDB 中并不需要这么做，因为它将在第一次存储数据时自动创建数据库和集合。

在 shell 中可以使用 `use` 函数切换到已有的数据库或者创建新的数据库，在命令后加上希望使用的数据库名称即可，无论它存不存在。如下脚本展示了如何使用 `library` 数据库：

```
> use library
Switched to db library
```

该例调用了 `use` 函数，然后紧跟着数据库名称，通过这种方式将全局变量 `db` 设置为 `library`。这意味着，接下来所有输入到 shell 中的命令都将在数据库 `library` 中执行，除非将该变量重置为另一个数据库。

4.1.1 查看可用的数据库和集合

MongoDB 将在保存数据的时候自动创建数据库，并且还区分大小写。出于这个原因，很难确定目前使用的是否是正确的数据库。因此，在切换到某个数据库之前，最好先查看 MongoDB 中目前所有可用的数据库，避免出现忘记数据库名称或者拼写不正确的情况。可以通过 `show dbs` 函数实现：

```
> show dbs
admin
local
```

注意该函数只会显示出已经存在的数据库。此时，数据库尚未包含任何数据，所以只显示

了 admin 和 local 这两个数据库。如果希望查看当前数据库中的所有集合,可以使用 show collections 函数:

```
> show collections
system.indexes
```

注意, system.indexes 集合将在保存数据的时候创建。该集合包含了一个基于已插入文档_id 键值的索引;它还包含了任何自定义创建的索引。

小窍门:

为了查看当前正在使用的数据库,在 MongoDB shell 中输入 db 即可。

4.2 在集合中插入数据

通常,最常用的操作就是在集合中插入数据。所有数据都以 BSON 格式存储(不仅紧凑,并且扫描速度快),所以也需要以 BSON 格式插入数据。有几种方式可以完成数据的插入。例如,可以先定义数据,然后使用 insert 函数将它们保存在集合中,或者在使用 insert 函数时,临时输入文档内容:

```
> document = ( { "Type" : "Book", "Title" : "Definitive Guide to MongoDB 2nd ed.,
The", "ISBN" : "978-1-4302-5821-6", "Publisher" : "Apress", "Author": [
"Hows, David", "Plugge, Eelco", "Membrey, Peter", "Hawkins, Tim" ] } )
> db.media.insert(document)
```

注意:

在 shell 中定义变量时(例如, document = ({ ... })), 变量的内容将被立即输出。

在 shell 中输入时也可以使用换行符。当编写一个相当长的文档时,这是非常方便的,如下所示:

```
> document = ( { "Type" : "Book",
... "Title" : "Definitive Guide to MongoDB 2nd ed., The",
... "ISBN" : "978-1-4302-5821-6",
... "Publisher" : "Apress",
... "Author" : ["Hows, David", "Plugge, Eelco", "Membrey, Peter", "Hawkins, Tim"]
... } )
> db.media.insert(document)
```

如前所述,另一种方式是直接通过 shell 插入数据,不需要提前定义文档。可以调用 insert 函数,然后紧跟着文档的内容:

```
> db.media.insert( { "Type" : "CD", "Artist" : "Nirvana", "Title" : "Nevermind" } )
```

还可以插入含有换行符的文档。例如,可以添加一组曲目,对之前的例子进行扩充。注意下面的样例中是如何使用逗号和括号的:

```
> db.media.insert( { "Type" : "CD",
... "Artist" : "Nirvana",
... "Title" : "Nevermind",
... "Tracklist" : [
```

```

... {
... "Track" : "1",
... "Title" : "Smells Like Teen Spirit",
... "Length" : "5:02"
... },
... {
... "Track" : "2",
... "Title" : "In Bloom",
... "Length" : "4:15"
... }
... ]
... }
... )

```

如你所见，使用 Mongo shell 插入数据是非常直接的。

插入数据的过程极其灵活，但必须遵守一些规则。例如，在插入文档时，键的名字必须遵守如下规则：

- \$字符不能是键名的第一个字符。例子：\$tags
- 圆点[.]不能出现在键名中。例子：ta.gs
- 名称_id 被保留用作主键 ID；尽管不推荐，但可以保存任何唯一值，例如字符串或整数。类似地，创建集合时也存在一些限制。例如，集合的名称必须遵守以下规则：
- 集合的名称不能超过 128 个字符。
- 空字符串(“”)不能被用作集合名称。
- 集合名必须以字母或下划线开头。
- 集合名 system 被 MongoDB 保留，不能使用。
- 集合名不能包含 null 字符“\0”。

4.3 查询数据

之前你已经学习了如何切换数据库和插入数据；接下来，你将学习如何查询集合中的数据。现在基于之前的样例构建新的例子，学习如何查询指定集合中的数据。

注意：

查询数据时，有大量可用的选项、操作符、表达式、过滤器等。在接下来的几节内容中你将学习这些选项。

函数 `find()` 提供了从同一集合的多个文档中获取数据的最简单方式。它将是最常用的函数之一。

假设已经在 `library` 数据库的 `media` 集合中插入了之前的两个样例。如果在该集合上使用 `find()`，结果将是其中的所有文档：

```

> db.media.find()
{ "_id" : "ObjectId("4c1a8a56c603000000007ecb)", "Type" : "Book", "Title" :
"Definitive Guide to MongoDB 2nd ed., The", "ISBN" : "978-1-4302-5821-6", "Publisher" :
"Apress", "Author" : ["Hows, David ", "Plugge, Eelco", "Membrey, Peter", "Hawkins, Tim"]}

{ "_id" : "ObjectId("4c1a86bb2955000000004076)", "Type" : "CD", "Artist" :
"Nirvana", "Title" : "Nevermind", "Tracklist" : [

```

```

    {
      "Track" : "1",
      "Title" : "Smells Like Teen Spirit",
      "Length" : "5:02"
    },
    {
      "Track" : "2",
      "Title" : "In Bloom",
      "Length" : "4:15"
    }
  ] }

```

该例非常简单，但通常你并不希望获取集中所有的文档。相反，你可能只希望获取特定类型的文档。例如，返回 Nirvana 的所有 CD，如下所示：

```

> db.media.find ( { Artist : "Nirvana" } )
{ "_id" : "ObjectId("4c1a86bb2955000000004076)", "Type" : "CD", "Artist" :
"Nirvana", "Title" : "Nevermind", "Tracklist" : [
  {
    "Track" : "1",
    "Title" : "Smells Like Teen Spirit",
    "Length" : "5:02"
  },
  {
    "Track" : "2",
    "Title" : "In Bloom",
    "Length" : "4:15"
  }
] }

```

该例看起来好多了！不需要看到集中的所有文档信息，只得到了感兴趣的信息。不过，如果仍然对返回的结果不满意，该怎么办呢？例如，希望得到一个列表，只包含 Nirvana 相关 CD 的标题，忽略任何其他信息，比如曲目列表。那么可以在查询中插入一个额外的参数，指定希望返回的键的名称，并在其后紧跟一个 1：

```

> db.media.find ( {Artist : "Nirvana"}, {Title: 1} )
{ "_id" : ObjectId("4c1a86bb2955000000004076)", "Title" : "Nevermind" }

```

插入的参数 {Title: 1} 表示只返回标题字段中的信息。结果将按升序显示。

注意：

升序排序将基于文档的插入顺序。

还可以通过插入 {Type: 0} 实现相反的操作，获取所有与 Nirvana 相关的文档，但只显示除 Type 字段之外的信息。

注意：

字段 `_id` 默认将一直显示，除非特意将它隐藏。

尝试运行修改后的查询(插入了 {Title: 1})，它只返回必要的信息。这将节省许多时间，因为你只看到了需要的信息。还可以节省数据查询的时间，因为它不需要返回不必要的信息。

4.3.1 使用点号

在使用复杂文档结构(例如包含数组或内嵌对象的文档)时,也可以使用其他方法查询这些对象中的信息。例如,假设希望找到所有包含了特定歌曲的 CD。下面的代码将执行一个更具体的查询:

```
> db.media.find( { "Tracklist.Title" : "In Bloom" } )
{ "_id" : "ObjectId("4c1a86bb2955000000004076)", "Type" : "CD", "Artist" :
"Nirvana", "Title" : "Nevermind", "Tracklist" : [
  {
    "Track" : "1",
    "Title" : "Smells Like Teen Spirit",
    "Length" : "5:02"
  },
  {
    "Track" : "2",
    "Title" : "In Bloom",
    "Length" : "4:15"
  }
]
}
```

在键的名称之后使用点号[.], 将告诉 find 函数查找文档中内嵌的信息。处理数组时代码会更简单一些。例如,可以执行下面的查询,返回由 Peter Membrey 编写的图书列表:

```
> db.media.find( { "Author" : "Membrey, Peter" } )
{ "_id" : "ObjectId("4c1a8a56c603000000007ecb)", "Type" : "Book", "Title" :
"Definitive Guide to MongoDB 2nd ed., The", "ISBN" : "978-1-4302-5821-6", "Publisher" :
"Apress", "Author" : ["Hows, David", "Plugge, Eelco", "Membrey, Peter", "Hawkins, Tim"] }
```

不过,下面的命令不会匹配任何文档,尽管它似乎与之前的曲目列表查询一致:

```
> db.media.find ( { "Tracklist" : { "Track" : "1" } } )
```

子对象必须精确匹配;因此,之前的查询无法匹配到任何还包含其他信息的文档,例如 Track.Title:

```
{ "Type" : "CD",
  "Artist" : "Nirvana"
  "Title" : "Nevermind",
  "Tracklist" : [
    {
      "Track" : "1",
    },
    {
      "Track" : "2",
      "Title" : "In Bloom",
      "Length" : "4:15"
    }
  ]
}
```

4.3.2 使用函数 sort、limit 和 skip

MongoDB 包含了几个函数,它们可用于更精确地控制查询。本节将讲解如何使用 sort、limit 和 skip 函数。

通过使用 `sort` 函数可以对查询返回的结果进行排序。可以分别使用 `1` 或 `-1` 将结果按照升序或降序排序。该函数与 SQL 中的 `ORDER BY` 语句类似，并且它将使用键的名称和排序方法作为条件，如下所示：

```
> db.media.find().sort( { Title: 1 } )
```

该例基于 `Title` 键的值对结果进行升序排序。在不指定参数的情况下，这是默认的排序顺序。还可以使用 `-1` 来进行降序排序。

注意：

如果指定一个不存在的键用于排序，结果将按照文档插入的顺序返回。

通过使用 `limit()` 函数可以限制返回结果的最大数目。该函数只需要一个参数：希望返回的结果的数目。如果使用的参数为 `0`，那将返回所有结果。下面的样例只返回 `media` 集合中的前 10 个文档：

```
> db.media.find().limit( 10 )
```

还可以使用 `skip()` 函数忽略掉集合中的前 `n` 个文档。下面的样例略过了 `media` 集合中的前 20 个文档：

```
> db.media.find().skip( 20 )
```

该例将返回集合中的所有文档，除了前 20 个文档。记住，它将按照文档插入的顺序找到这些文档。

如果不能结合使用这些命令，MongoDB 也就算不上强大。实际上，任何函数都可以与其他函数结合使用。下面的样例首先忽略了一些文档，然后对返回结果的数目加以限制，最后将结果按降序排序：

```
> db.media.find().sort ( { Title : -1 } ).limit ( 10 ).skip ( 20 )
```

如果希望在应用中实现分页功能，那么可能会用到该例。如你猜到的，该命令不会返回任何结果，因为到目前为止，`media` 集合中插入的文档数目比样例中忽略的数目还小。

注意：

可以在 `find()` 函数中使用快捷方式忽略并限制结果的数目：`find({}, {}, 10, 20)`。这里，将首先忽略开头的 20 个文档，然后将结果数目限制为 10 个。

4.3.3 使用固定集合、自然顺序和 `$natural`

在 MongoDB 中使用查询时，还需要注意一些额外的概念和特性，包括固定集合、自然顺序和 `$natural`。本节将讲解这些术语的定义以及如何在排序中使用它们。

自然顺序是数据库中集合的原生排序方法。所以，在查询集合中的文档时，结果将默认按照前向自然顺序返回。这通常与文档插入的顺序一致；不过，不能完全保证这样，因为数据在修改之后，可能无法存储在它之前的位置，因此会被移到其他位置。

固定集合是数据库的一种集合，它的自然顺序保证与文档插入的顺序一致。保证自然顺序一直与文档插入顺序一致，这一点在需要将查询的结果严格按照文档插入顺序排序时非常有用。

固定集合还有另一个优点：大小固定。一旦固定集合达到设置的大小，最老的数据将被删除，最新的数据将被添加到末端，保证自然顺序与文档插入的顺序一致。该类型的集合可用于日志或自动归档数据。

与标准集合不同，固定集合必须使用 `createCollection` 函数，以显式的方式创建。必须使用参数指定集合的大小。例如，下面的代码将创建一个名为 `audit` 的固定集合，它的大小最大不能超过 20480 字节：

```
> db.createCollection("audit", {capped:true, size:20480})
{ "ok" : 1 }
```

鉴于固定集合保证了自然顺序与插入顺序一致，那么在查询数据时，就不需要再使用任何特殊的参数、任何其他特殊的命令或函数，除非希望逆转默认结果的顺序。这时将会用到 `$natural` 参数。例如，假设希望找到固定集合中最近的 10 条记录，该集合用于保存登录失败的记录。可以使用 `$natural` 参数来实现：

```
> db.audit.find().sort( { $natural: -1 } ).limit ( 10 )
```

注意：

已经添加到固定集合中的文档可以被更新，但文档大小不能改变。如果出现这种情况，更新将会失败。也不可以从固定集合中删除文档。相反，如果希望删除文档，就必须删除整个集合并重建。在本章将会详细讲解删除集合的方法。

还可以使用 `max` 参数限制添加到固定集合中的文档数量，创建集合时使用该参数。不过，必须注意保证集合有足够的空间容纳这些文档。如果在文档数目达到上限之前，数据大小先达到上限，那么最老的文档将被删除。`MongoDB shell` 包含一个工具，可用于检查现有集合已经使用的空间大小，无论是固定集合还是普通集合都可以查看。使用 `validate()` 函数调用该工具即可。如果希望评估一个集合可能会变得多大，那么该方法非常有用。

正如之前描述的，可以使用 `max` 参数限制插入到固定集合中的文档数目，如下所示：

```
> db.createCollection("audit100", { capped:true, size:20480, max: 100})
{ "ok" : 1 }
```

接下来，使用 `validate()` 函数检查集合的大小：

```
> db.audit100.validate()
{
  "ns" : "media.audit100",
  "result" : "
    validate
    capped:1 max:100
    firstExtent:0:54000 ns:media.audit100
    lastExtent:0:54000 ns:media.audit100
    # extents:1
    datasize?:0 nrecords?:0 lastExtentSize:20736
    padding:1
    first extent:
    loc:0:54000 xnext:null xprev:null
    nsdiag:media.audit100
    size:20736 firstRecord:null lastRecord:null
    capped outOfOrder:0 (OK)
    0 objects found, nobj:0"
```

```
0 bytes data w/headers
0 bytes data wout/headers
deletedList: 11000000000000000000
deleted: n: 2 size: 20560
nIndexes:0
",
"ok" : 1,
"valid" : true,
"lastExtentSize" : 20736
}
```

输出结果显示表 `audit100` 是一个最多容纳 100 个文档的固定集合，它目前不包含文档。

4.3.4 获取单个文档

到目前为止，我们已经学习了如何获取多个文档。不过，如果只希望返回一个结果，那么查询所有的文档——执行 `find()` 函数时所做的事情——将会浪费 CPU 和内存。对于这种情况，可以使用 `findOne()` 函数来获取集合中的单个文档。该结果将与使用 `limit(1)` 函数得到的结果一致，但这种方式明显更简单。

函数 `findOne()` 的语法与 `find()` 函数一致：

```
> db.media.findOne()
```

如果希望返回一个结果，那么建议使用 `findOne()` 函数。

4.3.5 使用聚集命令

MongoDB 提供了一组很棒的聚集命令。开始可能看不出它们巨大的用途，但一旦开始使用它们，你就会明白聚集命令组成了一个极其强大的工具集。例如，可以使用它们获得数据库的一些基本统计的概述。本节将学习如何使用所有可用聚集命令中的 3 个函数：`count`、`distinct` 和 `group`。

除了这 3 个基本的聚集函数，MongoDB 还包含一个聚集框架。这个强大的特性可以帮助在不使用映射/归约框架(通常非常复杂)的情况下，计算聚集结果。第 5 章将对聚集框架进行详细讲解。

1. 使用 `count()` 函数返回文档的数目

函数 `count()` 将返回指定集合中的文档数目。到目前为止，我们已经在 `media` 中添加了许多文档。下面的例子将显示到底添加了多少个文档：

```
> db.media.count()
2
```

如下所示，还可以执行额外的过滤，结合条件操作符使用 `count()`：

```
> db.media.find( { Publisher : "Apress", Type: "Book" } ).count()
1
```

该例只返回集合中包含 Apress 出版图书信息的文档的数目。注意：`count()` 函数默认将忽略 `skip()` 或 `limit()` 参数。为了确保查询不会忽略这些参数，也为了保证进行计数的结果都符合 `limit()` 和/或 `skip()` 参数，请使用 `count(true)`：

```
> db.media.find( { Publisher: "Apress", Type: "Book" }).skip ( 2 ) .count (true)
0
```

2. 使用 distinct()函数获取唯一值

之前的样例显示了如何获取含有特定出版商图书信息的文档的数目。不过，这种方式并不准确。毕竟，如果集中有多本书(例如，实体书和电子书)含有相同的标题，那么从技术上讲，其实集中只含有一本书。这时就将用到 `distinct()` 函数：它将只返回一个唯一值。

出于完整性考虑，可以在集中添加一个额外的文档。该文档含有相同的标题，但使用另一个不同的 ISBN 号码：

```
> document = ( { "Type" : "Book", "Title" : "Definitive Guide to MongoDB 2nd ed., The", ISBN:
"978-1-4302-5821-6", "Publisher" : "Apress", "Author" :
["Hows, David", "Membrey, Peter", "Plugge, Eelco", "Hawkins, Tim"] } )
> db.media.insert (document)
```

此时，数据库中将含有两本具有相同标题的图书。在集合的标题上使用 `distinct()` 函数，结果将只会显示出两个唯一值。两本具有相同标题的图书将被合并成一条记录。另一个结果将显示专辑“Nevermind”的标题：

```
> db.media.distinct( "Title" )
[ "Definitive Guide to MongoDB, The", "Nevermind" ]
```

类似地，如果查询 ISBN 号码的唯一值列表，同样会显示出两个结果：

```
> db.media.distinct ( "ISBN" )
[ "1-4302-3051-7", "987-4302-3051-9" ]
```

在查询时，`distinct()` 函数还可以接受嵌套键。例如，下面的命令将返回 CD 标题的唯一值列表：

```
> db.media.distinct ( "Tracklist.Title" )
[ "In Bloom", "Smells Like Teen Spirit" ]
```

3. 将结果分组

最后但并不是最不重要的是：将结果分组。MongoDB 的 `group()` 函数类似于 SQL 的 GROUP BY 子句，尽管语法稍有不同。该命令的目的是返回一个已分组元素的数组。函数 `group()` 将接受 3 个参数：`key`、`initial` 和 `reduce`。

参数 `key` 指定希望使用哪个键对结果进行分组。例如，假设希望通过 `Title` 对结果进行分组。参数 `initial` 允许为每个已分组的结果提供基数(元素开始统计的起始基数)。`reduce` 将接受两个参数：正在遍历的当前文档和聚集计数对象。在接下来的样例中，这些参数被称为 `items` 和 `prev`。实质上，`reduce` 参数在每遇到一个匹配标题的文档时都将把总数加 1。

如果在寻找 `tagcloud` 之类的函数，那么 `group()` 函数就是你的目标。例如，假设希望获得集中任何类型元素的所有唯一标题的列表。另外，只要发现任何相同的元素，就基于标题将它们添加到同一分组中：

```
> db.media.group (
{
  key: {Title : true},
  initial: {Total : 0},
  reduce : function (items,prev)
{
```



```
    prev.Total += 13;
  }
}
[
  {
    "Title" : "Nevermind",
    "Total" : 1
  },
  {
    "Title" : "Definitive Guide to MongoDB, The",
    "Total" : 2
  }
]
```

除了 `key`、`initial` 和 `reduce` 参数，还可以指定另外 3 个可选参数：

- **keyf**: 如果不希望使用文档中已有的键对结果进行分组，可以使用该参数替代 `key` 参数。此时，可以使用另一个指定了如何分组的函数对结果进行分组。
- **cond**: 可以使用该参数指定一个额外的语句条件，文档必须满足该条件才能参加分组。可以像使用 `find()` 查询一样使用该参数搜索集合中的文档。如果未设置该参数(默认)，那么集合中所有的文档都将被检查。
- **finalize**: 可以使用该参数指定一个函数，用于在最终结果返回之前执行。例如，可以计算平均数或执行计数，并在结果中包含该信息。

注意：

函数 `group()` 目前在分片环境中无法正常工作。因此，在这种环境中应该使用 `mapreduce()`。另外，在 `group()` 函数的输出结果中包含的键不能超过 10000 个，否则将会抛出异常。这样的情况也可以通过 `mapreduce()` 来处理。

4.3.6 使用条件操作符

MongoDB 支持大量的条件操作符用于过滤结果。接下来将提供这些操作符的概述，包括一些演示如何使用它们的基本例子。在学习这些样例之前，首先需要在数据库中再添加一些文档，以便于能看出这些操作符的效果。

```
dvd = ( { "Type" : "DVD", "Title" : "Matrix, The", "Released" : 1999,
  "Cast" : ["Keanu Reeves", "Carrie-Anne Moss", "Laurence Fishburne", "Hugo Weaving", "Gloria Foster", "Joe Pantoliano"] } )
{
  "Type" : "DVD",
  "Title" : "Matrix, The",
  "Released" : 1999,
  "Cast" : [
    "Keanu Reeves",
    "Carrie-Anne Moss",
    "Laurence Fishburne",
    "Hugo Weaving",
    "Gloria Foster",
    "Joe Pantoliano"
  ]
}
```

```

> db.media.insert(dvd)

> dvd = ( { "Type" : "DVD", Title : "Blade Runner", Released : 1982 } )
{ "Type" : "DVD", "Title" : "Blade Runner", "Released" : 1982 }
> db.media.insert(dvd)

> dvd = ( { "Type" : "DVD", Title : "Toy Story 3", Released : 2010 } )
{ "Type" : "DVD", "Title" : "Toy Story 3", "Released" : 2010 }
> db.media.insert(dvd)

```

1. 执行大于和小于比较

以下特殊参数可用于在查询中执行大于和小于比较: `$gt`、`$lt`、`$gte` 和 `$lte`。本小节将学习如何使用这些参数。

第一个要学习的是 `$gt`(大于)参数。使用该参数指定文档中的某个整数必须大于一个指定值时,才能在结果中返回:

```

> db.media.find ( { Released : { $gt : 2000 } }, { "Cast" : 0 } )
{ "_id" : ObjectId("4c4369a3c60300000007ed3"), "Type" : "DVD", "Title" :
"Toy Story 3", "Released" : 2010 }

```

注意在上面的例子中,2000 不会出现在结果中。为此,可以使用 `$gte`(大于或等于)参数:

```

> db.media.find ( { Released : { $gte : 1999 } }, { "Cast" : 0 } )
{ "_id" : ObjectId("4c43694bc60300000007ed1"), "Type" : "DVD", "Title" :
"Matrix, The", "Released" : 1999 }
{ "_id" : ObjectId("4c4369a3c60300000007ed3"), "Type" : "DVD", "Title" :
"Toy Story 3", "Released" : 2010 }

```

同样,可以使用 `$lt`(小于)参数查找集合中年份在 1999 年之前发布的文档:

```

> db.media.find ( { Released : { $lt : 1999 } }, { "Cast" : 0 } )
{ "_id" : ObjectId("4c436969c60300000007ed2"), "Type" : "DVD", "Title" : "Blade Runner",
"Released" : 1982 }

```

还可以使用 `$lte`(小于或等于)参数获取年份在 1999 年之前且含 1999 年发布的文档列表:

```

> db.media.find( {Released : { $lte : 1999 }}, { "Cast" : 0 })
{ "_id" : ObjectId("4c43694bc60300000007ed1"), "Type" : "DVD", "Title" :
"Matrix, The", "Released" : 1999 }
{ "_id" : ObjectId("4c436969c60300000007ed2"), "Type" : "DVD", "Title" :
"Blade Runner", "Released" : 1982 }

```

可以结合这些参数指定一个范围:

```

> db.media.find( {Released : { $gte : 1990, $lt : 2010 }}, { "Cast" : 0 })
{ "_id" : ObjectId("4c43694bc60300000007ed1"), "Type" : "DVD", "Title" :
"Matrix, The", "Released" : 1999 }

```

这些参数的使用相对简单,在查询特定范围数据的查询中将会大量地用到它们。

2. 获取除特定文档之外的所有文档

使用参数 `$ne` 可以获取集合中除某些符合特定条件的文档之外的所有文档。例如,下面的代码将返回作者不是 Eelco Plugge 的所有图书的列表:

```

> db.media.find( { Type : "Book", Author : { $ne : "Plugge, Eelco" } })

```

3. 指定一个匹配的数组

可以使用 \$in 操作符指定一组可能的匹配值。SQL 中对应的操作符为 IN。

下面的代码将使用 \$in 操作符从 media 集合中获取数据：

```
> db.media.find( {Released : { $in : [1999,2008,2009] } }, { "Cast" : 0 } )
{ "_id" : ObjectId("4c43694bc603000000007ed1"), "Type" : "DVD", "Title" : "Matrix, The",
  "Released" : 1999 }
```

该例只返回一个文档，因为只有它满足了条件：发布年份为 1999 年，其他文档的发布年份都不符合条件 2008 年和 2009 年。

4. 查找某个不在数组中的值

操作符 \$nin 与 \$in 类似，不过它将搜索特定字段的值不在数组列表中的文档：

```
> db.media.find( {Released : { $nin : [1999,2008,2009] }, Type : "DVD" },
  { "Cast" : 0 } )
{ "_id" : ObjectId("4c436969c603000000007ed2"), "Type" : "DVD", "Title" :
  "Blade Runner", "Released" : 1982 }
{ "_id" : ObjectId("4c4369a3c603000000007ed3"), "Type" : "DVD", "Title" :
  "Toy Story 3", "Released" : 2010 }
```

5. 匹配文档中的所有属性

操作符 \$all 的工作方式也与 \$in 类似。不过，\$all 要求文档的所有属性都匹配，而 \$in 操作符只要求文档中的一个属性匹配即可。下面将通过例子演示它们之间的区别。首先是 \$in 的用法：

```
> db.media.find ( { Released : { $in : ["2010","2009"] } }, { "Cast" : 0 } )
{ "_id" : ObjectId("4c4369a3c603000000007ed3"), "Type" : "DVD", "Title" :
  "Toy Story 3", "Released" : 2010 }
```

该例返回一个结果，因为只有一个文档的发布年份为 2010 年，没有符合 2009 年发布的文档。不过，\$all 参数不会返回任何结果，因为没有文档满足 2009 年发布这个条件：

```
> db.media.find ( { Released : { $all : ["2010","2009"] } }, { "Cast" : 0 } )
```

6. 在文档中搜索多个表达式

在单个查询中可以使用 \$or 操作符搜索多个表达式，它将返回满足其中任何一个条件的文档。与 \$in 操作符不同，\$or 允许同时指定键和值，而不是只指定值：

```
> db.media.find({ $or : [ { "Title" : "Toy Story 3" }, { "ISBN" :
  "987-1-4302-3051-9" } ] } )
{ "_id" : ObjectId("4c5fc7d8db290000000067c5"), "Type" : "Book", "Title" :
  "Definitive Guide to MongoDB, The", "ISBN" : "987-1-4302-3051-9",
  "Publisher" : "Apress", "Author" : ["Hows, David", "Membrey, Peter", "Plugge, Eelco",
  "Hawkins, Tim" ] }
{ "_id" : ObjectId("4c5fc943db290000000067ca"), "Type" : "DVD", "Title" :
  "Toy Story 3", "Released" : 2010 }
```

还可以将 \$or 与另一个查询参数结合使用。这将会要求返回的文档必须先满足第一个条件，然后再满足 \$or 操作符中的两个键值对之一，如下所示：

```
> db.media.find({ "Type" : "DVD", $or : [ { "Title" : "Toy Story 3" }, {
```

```
"ISBN" : "987-1-4302-3051-9" } ] })
{ "_id" : ObjectId("4c5fcd3edb290000000067ca"), "Type" : "DVD", "Title" :
  "Toy Story 3", "Released" : 2010 }
```

通过\$or操作符可以同时执行两个查询，将两个无关的查询结果结合在一起。

7. 使用\$slice 获取文档

通过\$slice获取的文档将只包含数组中特定范围的值。如果希望通过限制结果元素的数目来节省带宽，那么该操作符非常有用。通过该操作符还可以获取结果中每页的n项数据，该特性通常被称为分页。

理论上，操作符\$slice结合了limit()和skip()函数的功能；不过，limit()和skip()无法作用于数组，而\$slice可以。该操作符接受两个参数：第一个参数表示将要返回数据项的总数；第二个参数是可选的，如果使用了该参数，那么第一个参数将用于定义偏移，第二个参数用于定义限制。参数limit也可以使用负值。

下面的样例将只返回Cast列表中的前3项：

```
> db.media.find({"Title" : "Matrix, The"}, {"Cast" : {$slice: 3}})
{ "_id" : ObjectId("4c5fcd3edb290000000067cb"), "Type" : "DVD", "Title" :
  "Matrix, The", "Released" : 1999, "Cast" : [ "Keanu Reeves", "Carrie-Anne
  Moss", "Laurence Fishburne" ] }
```

还可以通过使用负整数返回它的最后3项：

```
> db.media.find({"Title" : "Matrix, The"}, {"Cast" : {$slice: -3}})
{ "_id" : ObjectId("4c5fcd3edb290000000067cb"), "Type" : "DVD", "Title" :
  "Matrix, The", "Released" : 1999, "Cast" : [ "Hugo Weaving", "Gloria Foster",
  "Joe Pantoliano" ] }
```

或者可以忽略前两项，并限制返回从某个点开始的3个数据项(注意其中的括号)：

```
> db.media.find({"Title" : "Matrix, The"}, {"Cast" : {$slice: [2,3] }})
{ "_id" : ObjectId("4c5fcd3edb290000000067cb"), "Type" : "DVD", "Title" :
  "Matrix, The", "Released" : 1999, "Cast" : [ "Laurence Fishburne", "Hugo
  Weaving", "Gloria Foster" ] }
```

最后，指定一个负整数，用于略过最后5项数据，并将结果输出限制为4条，如下所示：

```
> db.media.find({"Title" : "Matrix, The"}, {"Cast" : {$slice: [-5,4] }})
{ "_id" : ObjectId("4c5fcd3edb290000000067cb"), "Type" : "DVD", "Title" :
  "Matrix, The", "Released" : 1999, "Cast" : [ "Carrie-Anne Moss", "Laurence
  Fishburne", "Hugo Weaving", "Gloria Foster" ] }
```

注意：

MongoDB 2.4 还为\$push操作符引入了\$slice操作符，在向数组添加数据时可以同时限制数组元素的数目。本章稍后将讲解该操作符。不过，不要混淆这两个操作符。

8. 搜索奇数/偶数

通过使用\$mod操作符可以搜索由奇数或偶数组成的特定数据。该操作符将把目标值除以2，并检查该运算的余数是否为0，通过这种方式只提供偶数结果。

例如，下面的代码将返回集合中任何 Released 字段为偶数的文档：

```
> db.media.find ( { Released : { $mod: [2,0] } }, { "Cast" : 0 } )
{ "_id" : ObjectId("4c45b5c18e0f0000000062aa"), "Type" : "DVD", "Title" :
"Blade Runner", "Released" : 1982 }
{ "_id" : ObjectId("4c45b5df8e0f0000000062ab"), "Type" : "DVD", "Title" :
"Toy Story 3", "Released" : 2010 }
```

同样，可以修改参数 \$mod 来搜索所有 Released 字段为奇数的文档，如下所示：

```
> db.media.find ( { Released : { $mod: [2,1] } }, { "Cast" : 0 } )
{ "_id" : ObjectId("4c45b5b38e0f0000000062a9"), "Type" : "DVD", "Title" :
"Matrix, The", "Released" : 1999 }
```

注意：

操作符 \$mod 只能作用于整数，不能作用于包含了数值的字符串。例如，不能将该操作符用于 {Released : "2010"}，因为 2010 在引号中只是个字符串。

9. 使用 \$size 过滤结果

通过操作符 \$size 可以过滤出文档中数组大小符合条件的结果。例如，可以使用该操作符搜索只含有两首歌曲的 CD：

```
> db.media.find ( { Tracklist : { $size : 2 } } )
{ "_id" : ObjectId("4c1a86bb2955000000004076"), "Type" : "CD", "Artist" :
"Nirvana", "Title" : "Nevermind", "Tracklist" : [
  {
    "Track" : "1",
    "Title" : "Smells Like Teen Spirit",
    "Lenght" : "5:02"
  },
  {
    "Track" : "2",
    "Title" : "In Bloom",
    "Length" : "4:15"
  }
] }
```

注意：

不能使用 \$size 操作符搜索大小范围。例如，不能使用它搜索含有超过一个元素的数组。

10. 返回含有特定字段的对象

使用 \$exists 操作符将在特定字段存在或不存在的情况下，返回该对象。下面的样例将返回集合中含有键 Author 的所有文档：

```
> db.media.find ( { Author : { $exists : true } } )
```

类似地，如果将该操作符的值修改为 false，那么所有不含键 Author 的文档将会返回：

```
> db.media.find ( { Author : { $exists : false } } )
```

警告：

目前，\$exists 不能使用索引；因此，使用 \$exists 将会进行全表扫描。

11. 基于 BSON 类型匹配结果

使用操作符\$type 可以基于数据的 BSON 类型匹配结果。例如，下面的代码将搜索所有含有曲目列表的文档，并且它们的曲目列表是嵌入对象类型(也就是说，它包含了一个信息的列表)：

```
> db.media.find ( { Tracklist: { $type : 3 } } )
{ "_id" : ObjectId("4c1a86bb2955000000004076"), "Type" : "CD", "Artist" :
"Nirvana", "Title" : "Nevermind", "Tracklist" : [
  {
    "Track" : "1",
    "Title" : "Smells Like Teen Spirit",
    "Lenght" : "5:02"
  },
  {
    "Track" : "2",
    "Title" : "In Bloom",
    "Length" : "4:15"
  }
]
}
```

表 4-1 列出了已定义的数据类型。

表 4-1 已知的 BSON 类型和代码

代 码	数据类型
-1	MiniKey
1	Double
2	Character 字符串(UTF8)
3	嵌入式对象
4	嵌入式数组
5	二进制数据
7	对象 ID
8	Boolean 型
9	Date 型
10	Null
11	正则表达式
13	JavaScript 代码
14	Symbol
15	带作用域的 JavaScript 代码
16	32 位整型
17	时间戳
18	64 位整型
127	MaxKey
255	MinKey

12. 匹配完整的数组

如果希望匹配文档中的完整数组，可以使用\$elemMatch 操作符。如果集合中有多个文档，

其中一些文档含有一些相同的信息，那么此时该操作符就非常有用了。此时默认的查询将无法找到准确的文档，因为标准查询语法不匹配数组中的单个完整文档。

下面的样例演示了这一点。首先需要在集合中添加另一个文档，该文档与其他文档有一个一致的数据项，但其他部分不同。在此，添加 Nirvana 的一个 CD，它碰巧与之前提到的 CD(“Smells Like Teen Spirit”)含有相同的曲目。不过，在该版本的 CD 中，该歌曲是 CD 的第 5 首歌曲，而不是第 1 首：

```
{
  "Type" : "CD",
  "Artist" : "Nirvana",
  "Title" : "Nirvana",
  "Tracklist" : [
    {
      "Track" : "1",
      "Title" : "You know you're right",
      "Length" : "3:38"
    },
    {
      "Track" : "5",
      "Title" : "Smells like teen spirit",
      "Length" : "5:02"
    }
  ]
}

> nirvana = ( { "Type" : "CD", "Artist" : "Nirvana", "Title" : "Nirvana",
"Tracklist" : [ { "Track" : "1", "Title" : "You Know You're Right", "Length" :
: "3:38"}, {"Track" : "5", "Title" : "Smells Like Teen Spirit", "Length" :
: "5:02" } ] } )

> db.media.insert(nirvana)
```

如果希望搜索由 Nirvana 创作的含有歌曲“Smells Like Teen Spirit”的专辑，该歌曲应该是 CD 的第 1 首歌曲，那么可以使用下面的查询：

```
> db.media.find ( { "Tracklist.Title" : "Smells Like Teen Spirit",
"Tracklist.Track" : "1" } )
```

遗憾的是，之前的查询将同时返回两个文档。原因是这两个文档都含有标题为“Smells Like Teen Spirit”的歌曲，并且都含有歌曲 1。如果希望匹配数组中的完整文档，可以使用 \$elemMatch，如下所示：

```
> db.media.find ( { Tracklist: { "$elemMatch" : { Title:
"Smells like teen spirit", Track : "1" } } } )

{ "_id" : ObjectId("4c1a86bb2955000000004076"), "Type" : "CD", "Artist" :
"Nirvana", "Title" : "Nevermind", "Tracklist" : [
  {
    "Track" : "1",
    "Title" : "Smells Like Teen Spirit",
    "Lenght" : "5:02"
  },
  {
    "Track" : "2",
    "Title" : "In Bloom",
```

```

    "Length" : "4:15"
  }
}

```

该查询将返回目标结果，并且只返回第一个文档。

13. \$not(元操作符)

可以使用\$not 元操作符否定任何标准操作符执行的检查。下面的例子将返回集合中的所有文档，除了由\$elemMatch 样例匹配的文档：

```

> db.media.find ( { Tracklist : { $not : { "$elemMatch" : { Title:
"Smells Like Teen Spirit", "Track" : "1" } } } } )

```

14. 指定额外的查询表达式

除了目前学过的结构化查询语法，还可以使用 JavaScript 指定额外的查询表达式。这种方式最大的优势在于非常灵活，并且允许执行大量的额外查询。使用 JavaScript 的缺点在于，它比 MongoDB 支持的原生操作符稍微慢一点。

例如，现在希望搜索集合中发布年份早于 1995 年的 DVD。下面的所有代码样例都将返回该信息：

```

db.media.find ( { "Type" : "DVD", "Released" : { $lt : 1995 } } )

db.media.find ( { "Type" : "DVD", $where: "this.Released < 1995" } )

db.media.find ("this.Released < 1995")

f = function() { return this.Released < 1995 }
db.media.find(f)

```

MongoDB 就是这么灵活！通过使用这些操作符可以在集合中搜索任何信息。

4.3.7 使用正则表达式

正则表达式是另一个可用于查询信息的强大工具。正则表达式是一种特殊的文本，用于描述搜索模式。它们的工作方式与通配符相似，但更加强大和灵活。

MongoDB 允许在集合中搜索数据时使用正则表达式；不过，如果符合简单的前缀查询，那么它将尝试使用索引。

下面的样例在查询中使用了正则表达式，用于在 media 集合中搜索标题以“Matrix:”开头的文档：

```

> db.media.find ( { Title : /Matrix*/i } )

```

在 MongoDB 中使用正则表达式，可以使工作轻松许多。所以如果时间允许或者该技术对你有帮助的话，推荐你对该特性进行深入学习。

4.4 更新数据

到目前为止，你已经学习了如何在数据库中插入和查询数据。接下来，你将学习如何更新数据。MongoDB 支持许多更新操作符，接下来将对它们进行详细讲解。

4.4.1 使用 update()更新

在 MongoDB 中可以使用 update()函数执行数据更新操作。该函数将接受 3 个主要参数：criteria、objNew 和 options。

参数 criteria 可用于指定一个查询，该查询选择将要更新的目标记录。使用 objNew 参数指定更新信息，也可以使用操作符来完成。参数 options 用于指定更新文档时的选项，它的可选值有：upsert 和 multi。通过选项 upsert 可以指定该更新是否是 upsert 操作——它将告诉 MongoDB，如果数据存在就更新，否则创建数据。最后，通过选项 multi 可以指定是否应该更新所有匹配的文档，或者只更新第一个文档(默认行为)。

下面的样例将使用 update()函数更新数据，不含任何操作符：

```
> db.media.update( { "Title" : "Matrix, The"}, {"Type" : "DVD", "Title" :  
"Matrix, The", "Released" : 1999, "Genre" : "Action"}, { upsert: true } )
```

该例覆写了集合中的文档，并为之保存更新后的值。注意，任何忽略的字段都将被移除(该文档基本上整个被重写)。因为参数 upsert 被指定为真，任何不存在的字段都将被添加到文档中。

如果碰巧有多个文档符合该条件，并且希望对它们全部执行 upsert 操作，那么可以在使用 \$set 修改操作符的同时添加 upsert 和 multi 选项，如下所示：

```
> db.media.update( { "Title" : "Matrix, The"}, {$set: {"Type" : "DVD", "Title" :  
"Matrix, The", "Released" : 1999, "Genre" : "Action"} }, {upsert: true, multi: true} )
```

注意：

参数 upsert 将告诉数据库：“如果记录存在就更新，否则插入新的记录”。

4.4.2 使用 save()命令实现 upsert

还可以使用 save()命令执行 upsert。为了实现该操作，可以指定_id 值；可以手动或自动添加该值。如果不指定_id 值，save()命令将认为它是一个插入操作，并将文档添加到集合中。

使用 save()命令的主要优势是：不需要指定使用 update()命令时所需要使用的 upsert 参数。因此，save()命令提供了一种更快的方式来实现 upsert 数据操作。实际上，save()和 update()命令看起来相似：

```
> db.media.update( { "Title" : "Matrix, The"}, {"Type" : "DVD", "Title" :  
"Matrix, The", "Released" : "1999", "Genre" : "Action"}, { upsert: true } )  
  
> db.media.save( { "Title" : "Matrix, The"}, {"Type" : "DVD", "Title" :  
"Matrix, The", "Released" : "1999", "Genre" : "Action"})
```

很明显，该例将 Title 值用作 id 字段。

4.4.3 自动更新信息

可以使用修改操作符以简单的方式对文档进行快速更新，不需要手动输入任何信息。例如，可以使用这些操作符增加数值，或从数组中删除元素。

接下来将详细讲解这些操作符，并提供实际的样例展示如何使用它们。

1. 使用\$inc 增加值

操作符\$inc 可以为指定的键执行(原子)更新操作, 如果字段存在, 就将该值增加给定的增量。如果字段不存在, 就创建该字段。现在首先在集合中添加另一个文档:

```
> manga = ( { "Type" : "Manga", "Title" : "One Piece", "Volumes" : 612,
"Read" : 520 } )
{
  "Type" : "Manga",
  "Title" : "One Piece",
  "Volumes" : "612",
}
> db.media.insert(manga)
```

现在可以更新文档了。例如, 假设你已经阅读了 One Piece 漫画的 4 卷内容, 那么你会希望更新文档中已读册数的数目, 如下所示:

```
> db.media.update ( { "Title" : "One Piece"}, {$inc: {"Read" : 4} } )
> db.media.find ( { "Title" : "One Piece" } )
{
  "Type" : "Manga",
  "Title" : "One Piece ",
  "Volumes" : "612",
  "Read" : "524"
}
```

2. 设置字段值

可以使用\$set 操作符将某个字段设置为指定值。如下所示, 该操作符可以更新任何数据类型:

```
> db.media.update ( { "Title" : "Matrix, The" }, {$set : { Genre : "Sci-Fi" } } )
```

上述代码将更新之前文档中的 Genre 字段, 将它设置为 Sci-Fi。

3. 删除指定字段

通过\$unset 操作符可以删除指定的字段, 如下所示:

```
> db.media.update ( {"Title": "Matrix, The"}, {$unset : { "Genre" : 1 } } )
```

上述代码将从文档中删除 Genre 键和它的值。

4. 在指定字段中添加某个值

通过\$push 操作符可以在指定字段中添加某个值。如果该字段是个数组, 那么该值将被添加到数组中。如果该字段尚不存在, 那么该字段的值将被设置为数组。如果该字段存在, 但不是数组, 那么将会抛出错误。

首先在集合的文档中添加另一个作者:

```
> db.media.update ( {"ISBN" : "978-1-4302-5821-6"}, {$push: { Author : "Griffin,
Stewie"} } )
```

下面的代码将抛出一条错误消息, 因为 Title 字段的值不是数组:

```
> db.media.update ( {"ISBN" : "978-1-4302-5821-6"}, {$push: { Title :
```

```
"This isn't an array"} } )
Cannot apply $push/$pushAll modifier to non-array
```

下面展示了文档更新后的内容:

```
> db.media.find ( { "ISBN" : "978-1-4302-5821-6" } )
{
  "Author" :
  [
    "Hows, David",
    "Membrey, Peter",
    "Plugge, Eelco",
    "Griffin, Stewie",
  ],
  "ISBN" : "978-1-4302-5821-6",
  "Publisher" : "Apress",
  "Title" : "Definitive Guide to MongoDB 2nd ed., The",
  "Type" : "Book",
  "_id" : ObjectId("4c436231c60300000007ed0")
}
```

5. 指定数组中的多个值

使用\$push操作符可以将值添加到指定数组中,扩展指定元素中存储的数据。如果希望在给定的数组中添加几个不同的值,可以使用可选的\$each修改操作符,如下所示:

```
> db.media.update( { "ISBN" : "978-1-4302-5821-6" }, { $push: { Author : { $each:
["Griffin, Peter", "Griffin, Brian"] } } } )
{
  "Author" :
  [
    "Hows, David",
    "Membrey, Peter",
    "Plugge, Eelco",
    "Hawkins, Tim",
    "Griffin, Stewie",
    "Griffin, Peter",
    "Griffin, Brian"
  ],
  "ISBN" : "978-1-4302-5821-6",
  "Publisher" : "Apress",
  "Title" : "Definitive Guide to MongoDB 2nd ed., The",
  "Type" : "Book",
  "_id" : ObjectId("4c436231c60300000007ed0")
}
```

另外,在使用\$each时还可以使用\$slice操作符。通过这种方式可以限制\$push操作符中数组内元素的数量。\$slice接受负数或0。使用负数将保证数组中的最后n个元素会被保留,而使用0则表示清空数组。注意,操作符\$slice必须是\$push操作中的第一个修改操作符,如下所示:

```
> db.media.update( { "ISBN" : "978-1-4302-5821-6" }, { $push: { Author : { $each:
["Griffin, Meg", "Griffin, Louis"], $slice: -2 } } } )
{
  "Author" :
  [
    "Griffin, Meg",
    "Griffin, Louis"
  ]
}
```

```

    ],
    "ISBN" : "978-1-4302-5821-6",
    "Publisher" : "Apress",
    "Title" : "Definitive Guide to MongoDB 2nd ed., The",
    "Type" : "Book",
    "_id" : ObjectId("4c436231c603000000007ed0")
  }
}

```

如你所见，操作符\$slice 保证不仅两个新值会被添加到数组中，还保证将把数组的大小限制为指定值(2)。操作符\$slice 在处理固定数组时是一个非常有用的工具。

6. 使用\$addToSet 向数组中添加数据

操作符\$addToSet 是另一个可用于向数组中添加数据的命令。不过，只有数据不存在的时候，该操作符才能将数据添加到数组中。它的工作方式与\$push 不同。操作符\$addToSet 默认将接受一个参数。不过，在使用\$addToSet 时，可以使用\$each 操作符指定额外的参数。下面的代码将把作者 Griffin Brian 添加到作者数组中，因为它尚未存在于数组之中：

```
> db.media.update( { "ISBN" : "1-4302-3051-7" }, { $addToSet : { Author : "Griffin, Brian" } } )
```

再次执行该代码不会改变任何数据，因为该作者已经在数组中了。为了再添加一个值，可以采取另一种方式并使用\$each 操作符：

```
> db.media.update( { "ISBN" : "1-4302-3051-7" }, { $addToSet : { Author : { $each : ["Griffin, Brian","Griffin, Meg"] } } } )
```

此时，我们的文档将如下所示：

```

{
  "Author" :
  [
    "Hows, David",
    "Membrey, Peter",
    "Plugge, Eelco",
    "Hawkins, Tim",
    "Griffin, Stewie",
    "Griffin, Peter",
    "Griffin, Brian",
    "Griffin, Louis",
    "Griffin, Meg"
  ],
  "ISBN" : "1-4302-3051-7",
  "Publisher" : "Apress",
  "Title" : "Definitive Guide to MongoDB, The",
  "Type" : "Book",
  "_id" : ObjectId("4c436231c603000000007ed0")
}

```

7. 从数组中删除元素

MongoDB 中还包含了几种从数组中删除元素的方法，包括\$pop、\$pull 和\$pullAll。本小节将讲解如何使用这些方法从数组中删除元素。

操作符\$pop 可以从数组中删除单个元素。该操作符允许删除数组中的第一个或最后一个元

素，具体取决于传入的参数。例如，下面的代码将删除数组中的最后一个元素：

```
> db.media.update( { "ISBN" : "1-4302-3051-7" }, { $pop : { Author : 1 } } )
```

在本例中，\$pop 操作符将从作者列表中移除 Meg Griffin。如果传入的是负数，数组的第一个元素将被移除。下面的例子将从作者列表中移除 Peter Membrey：

```
> db.media.update( { "ISBN" : "1-4302-3051-7" }, { $pop : { Author : -1 } } )
```

注意：

将传入的数值修改为-2 或 1000 不会改变被移除的元素。只要使用的是负数，就会移除第一个元素，而使用正数将会移除最后一个元素。使用数字 0 可以移除数组的最后一个元素。

8. 删除所有指定值

通过使用 \$pull 操作符可以从数组中删除所有指定值。如果数组中有多个元素的值相同，那么该操作符是非常有用的。下面的例子将通过使用 \$pull 参数将 Stewie Griffin 添加回作者列表中：

```
> db.media.update ( { "ISBN" : "1-4302-3051-7" }, { $push : { Author : "Griffin, Stewie" } } )
```

在本例中，Stewie Griffin 将被移进移出数据库多次。下面将使用 \$pull 从文档的作者列表中删除该作者(出现的所有地方)：

```
> db.media.update ( { "ISBN" : "1-4302-3051-7" }, { $pull : { Author : "Griffin, Stewie" } } )
```

9. 删除数组中的多个元素

还可以从数组中删除多个含有不同值的元素。操作符 \$pullAll 可以完成该操作。该操作符将接受一个希望移除元素的数组，如下所示：

```
> db.media.update( { "ISBN" : "1-4302-3051-7" }, { $pullAll : { Author : ["Griffin, Louis", "Griffin, Peter", "Griffin, Brian"] } } )
```

希望删除的所有元素(之前样例中的 Author)必须放在数组中传给 \$pullAll。如果不这样做，将会抛出错误消息。

4.4.4 指定匹配数组的位置

可以在查询中使用 \$ 操作符指定查询中匹配数组元素的位置。该操作符可用于在搜索到一个数组元素之后，对它进行数据操作。例如，在歌曲列表中添加了另一首歌曲，但输入的时候不小心将歌曲序号输错了：

```
> db.media.update( { "Title" : "Nirvana" }, { $addToSet : { Tracklist : { "Track" : 2, "Title": "Been a Son", "Length": "2:23" } } } )

{
  "Artist" : "Nirvana",
  "Title" : "Nirvana",
  "Tracklist" : [
    {
      "Track" : "1",
```

```

        "Title" : "You Know You're Right",
        "Length" : "3:38"
    },
    {
        "Track" : "5",
        "Title" : "Smells Like Teen Spirit",
        "Length" : "5:02"
    },
    {
        "Track" : 2,
        "Title" : "Been a Son",
        "Length" : "2:23"
    }
],
"Type" : "CD",
"_id" : ObjectId("4c443ad6c603000000007ed5")
}

```

恰巧你知道，最近添加的歌曲序号应该是 3 而不是 2。可以使用 \$inc 操作符与 \$操作符一起将 2 增加到 3，如下所示：

```

> db.media.update( { "Tracklist.Title" : "Been a son"},
  {$inc:{"Tracklist.$Track" : 1} } )

```

注意，只有匹配的第一条记录会被更新。因此，如果歌曲列表中有两个一致的元素，那么只有第一个元素会被修改。

4.4.5 原子操作

MongoDB 支持针对单个文档的原子操作。原子操作是一组可以结合在一起使用的操作，它们对于系统的其他部分来说就像单个操作一样。这组操作将产生正面的或负面的结果作为最终结果。

如果一组操作满足下面的条件，那就可以称它们为原子操作：

- 其他进程无法获得修改的结果，除非整组操作都已经完成。
- 如果其中一个操作失败，整组操作(整个原子操作)都将失败，并全部回滚，数据将被恢复至运行原子操作之前的状态。

执行原子操作时的标准行为是锁定数据，不允许其他查询访问。不过，出于多种原因，MongoDB 并不支持锁或复杂的事务：

- 在分片环境中(第 12 章将对该环境进行详细讲解)，分布式锁是昂贵并且缓慢的。MongoDB 的目标是轻量并且快速，所以昂贵和缓慢的操作违反了它的原则。
- MongoDB 开发者不喜欢死锁。在他们看来，系统最好是简单并且可预测的。
- MongoDB 被设计用于处理实时问题。当执行一个操作需要锁定大量数组时，它将会停止一些轻量级查询的执行。该操作同样违反了 MongoDB 要求快速的目标。

MongoDB 包含几种更新操作(如之前所述)，它们都可以原子操作的方式更新元素：

- \$set: 设置特定值。
- \$unset: 删除特定值。
- \$inc: 将某个值增大特定的量
- \$push: 向数组中添加值。

- `$pull`: 从现有数组中删除一个或多个值。
- `$pullAll`: 从现有数组中删除一些值。

使用 Update if Current 方法

另一个更新元素的策略是使用 Update if Current(如果数据目前仍未改变就更新)方法。该方法有 3 个步骤:

- (1) 从文档中取得对象。
- (2) 在本地修改对象(使用之前提到的操作)。
- (3) 发送更新请求来更新对象值, 假定当前值仍然匹配之前取得的旧值。

可以使用 `getlasterror` 方法检查是否出现问题。注意, 所有的步骤都以原子的方式完成。下面是一个之前使用过的例子:

```
> db.media.update( { "Tracklist.Title" : "Been a son" },
  { $inc: { "Tracklist.$Track" : 1 } } )
```

现在使用 `getlasterror` 命令检查更新是否成功:

```
> db.$cmd.findOne({getlasterror:1})
```

如果原子更新成功, 将显示出下面的结果:

```
{ "err" : null, "updatedExisting" : true, "n" : 1, "ok" : 1 }
```

在本例中, 使用曲目列表标题作为标识符, 修改 `Tracklist.Track` 的值。当 MongoDB 执行该更新时, 如果同时有另一个用户使用相同的方法在修改曲目列表数据, 会发生什么事情呢? 因为 `Tracklist.Title` 并未改变, 所以可以认为该操作在更新原始数据, 但事实上它覆盖了我们修改后的数据。

这被称为 ABA 问题。在目前的场景中, 似乎不可能出现这个问题, 但在多用户环境中, 许多应用都在同时处理数据, 这就可能是一个明显的问题。

为了避免这个问题, 可以使用下面的方法:

- 在更新的查询表达式中使用完整的对象, 而不是只使用 `_id` 和 `comments.by` 字段。
- 使用 `$set` 更新重要的字段。即使其他字段已经改变, 也不会受该字段的影响。
- 在对象中添加一个版本变量, 并在每次更新时增加它的值。
- 如果可能, 请使用 `$` 操作符, 而不是 Update-if-Current 序列操作。

注意:

MongoDB 不支持在单个操作中以原子的方式更新多个文档。相反, 可以使用嵌套对象来实现单个文档中的原子操作。

4.4.6 以原子的方式修改和返回文档

还可以通过执行 `findAndModify` 命令来实现对文档的原子操作。该命令将修改并返回文档。它将接受 3 个主要的操作符: `<query>` 用于指定目标文档, `<sort>` 用于对多个匹配文档进行排序, `<operations>` 用于指定希望执行的操作。

下面将通过一些例子演示如何使用该命令。第一个样例找到了搜索的目标文档并在找到之后删除它:

```
> db.media.findAndModify( { "Title" : "One Piece",sort:{"Title": -1}, remove:
true} )
{
  "_id" : ObjectId("4c445218c60300000007ede"),
  "Type" : "Manga",
  "Title" : "One Piece",
  "Volumes" : 612,
  "Read" : 524
}
```

上述代码返回匹配搜索条件的文档。在本例中，找到并删除了匹配标题“**One Piece**”的第一个文档。如果现在执行 `find()` 函数，被删除的文档将不再出现在集合中。

下一个样例将修改文档而不是删除：

```
> db.media.findAndModify( { query: { "ISBN" : "987-1-4302-3051-9" }, sort:
{"Title":-1}, update: {$set: {"Title" : " Different Title"} } )
```

之前的样例将文档的标题“**Definitive Guide to MongoDB**”更新成了“**Different Title**”并返回旧文档(更新之前的文档)。如果希望返回更新后的文档，那么可以在查询后添加 `new` 操作符：

```
> db.media.findAndModify( { query: { "ISBN" : "987-1-4302-3051-9" }, sort:
{"Title":-1}, update: {$set: {"Title" : " Different Title"} }, new:true } )
```

注意在该命令中可以使用任何修改操作符，而不只是 `$set`。

4.5 重命名集合

有时你可能会发现使用了一个错误的集合名字，但已经向其中插入了一些数据。从头删除并重新插入这些数据可能很麻烦。

那么可以使用 `renameCollections()` 函数重命名现有的集合。下面演示了如何使用该函数：

```
> db.media.renameCollection("newname")
{ "ok" : 1 }
```

如果执行成功，将会返回 `OK`。如果执行失败(例如集合不存在)，那么将返回下面的消息：

```
{ "errmsg" : "assertion: source namespace does not exist", "ok" : 0 }
```

该函数接受的参数不多(与之前看到的一些命令不同)，不过如果用在正确的环境中，它是相当有用的。

4.6 删除数据

到目前为止，你已经学习了如何添加、搜索和修改数据。接下来学习如何删除文档、整个集合和数据库。

你之前已经学习了如何从特定的文档中删除数据(例如使用 `$pop` 命令)。在本节将学习如何删除完整的文档和集合。正如 `insert()` 函数用于插入，`update()` 函数用于修改数据，`remove()` 函数则用于删除文档。

为了从集合中删除单个文档，需要指定搜索条件以找到该文档。一种良好的方式是首先执行 `find()`，这将保证使用该条件可以搜索到目标文档。在确保搜索条件正确之后，再使用该条件作为参数调用 `remove()` 函数：

```
> db.newname.remove( { "Title" : "Different Title" } )
```

该语句将删除集合中所有标题为“Different Title”的文档。事实上，正因为该语句将删除所有的文档，所以最好使用 `_id` 值，因为它总是唯一的。

也可以使用下面的代码从 `newname` 库中删除所有的文档(记住，之前曾将 `media` 集合重命名为 `newname`)：

```
> db.newname.remove({})
```

警告：

在删除文档时，需要记住：对该文档的任何引用都将保留在数据库中。出于该原因，一定要保证同时删除或更新这些引用；否则，这些引用在执行时将返回 `null`。在下一节将讨论引用相关的内容。

如果希望删除整个集合，可以使用 `drop()` 函数。下面的代码将删除整个 `newname` 集合，包括它所有的文档：

```
> db.newname.drop()
true
```

函数 `drop()` 将返回真或假，取决于该操作是否成功执行。同样，如果希望从 MongoDB 中删除整个数据库，可以使用 `dropDatabase()` 函数，如下所示：

```
> db.dropDatabase()
{ "dropped" : "library", "ok" : 1 }
```

注意该代码将删除目前正在使用的数据库(一定要检查目前使用的是哪个数据库)。

4.7 引用数据库

此时我们已经清空了数据库，并且也已经学习了如何在集合中插入各种不同的数据。现在将开始学习数据库引用。如你所见，众多场景中，在文档中嵌入数据已经可以满足应用的需求(例如歌曲列表或书目中的作者列表)。不过，有时确实可能需要引用另一个文档中的信息。下面的将讲解如何实现引用。

如同 SQL 一样，MongoDB 文档之间的引用也是通过在服务器上执行额外的查询来解析完成的。MongoDB 提供了两种实现方式：手动引用它们，或者使用 `DBRef` 标准(多种驱动都支持它们)。

4.7.1 手动引用数据

引用数据的最简单和最直接方式就是手动完成。在手动引用数据时，可以将另一个文档中的 `_id` 存储在该文档中(通过完整的 ID 或更简单的常用条目)。在开始演示样例之前，首先添加一个新的文档，并在其中指定发布者的信息(注意 `_id` 字段)：

```

> apress = ( { "_id" : "Apress", "Type" : "Technical Publisher", "Category" :
["IT", "Software", "Programming"] } )
{
  "_id" : "Apress",
  "Type" : "Technical Publisher",
  "Category" : [
    "IT",
    "Software",
    "Programming"
  ]
}
> db.publisherscollection.insert(apress)

```

在添加发布者信息之后，现在开始在 `media` 集合中添加一个真正的文档(例如图书信息)。下面的样例添加了一个文档，它将指定 `Apress` 作为发布者的名字：

```

> book = ( { "Type" : "Book", "Title" : "Definitive Guide to MongoDB 2nd ed., The",
"ISBN" : "987-1-4302-5821-6", "Publisher" : "Apress", "Author" : ["Hows, David", "Plugge,
Eelco", "Membrey, Peter", "Hawkins, Tim"] } )
{
  "Type" : "Book",
  "Title" : "Definitive Guide to MongoDB 2nd ed., The",
  "ISBN" : "987-1-4302-5821-6",
  "Publisher" : "Apress",
  "Author" : [
    "Hows, David",
    "Membrey, Peter",
    "Plugge, Eelco",
    "Hawkins, Tim"
  ]
}
> db.media.insert(book)

```

所有必需的信息都已经被分别插入到 `publisherscollection` 和 `media` 集合中。现在可以开始使用数据库引用了。首先，将含有发布者信息的文档赋给一个变量：

```

> book = db.media.findOne()
{
  "_id" : ObjectId("4c458e848e0f00000000628e"),
  "Type" : "Book",
  "Title" : "Definitive Guide to MongoDB, The",
  "ISBN" : "987-1-4302-3051-9",
  "Publisher" : "Apress",
  "Author" : [
    "Hows, David",
    "Membrey, Peter",
    "Plugge, Eelco",
    "Hawkins, Tim"
  ]
}

```

为了获得其中的信息，可以结合使用 `findOne` 函数和点操作符：

```

> db.publisherscollection.findOne( { _id : book.Publisher } )
{
  "_id" : "Apress",
  "Type" : "Technical Publisher",
  "Category" : [
    "IT",

```

```
        "Software",  
        "Programming"  
    ]  
}
```

如样例所示，手动引用数据很直观并且不需要太多的脑力工作。在集合 `users` 中，文档的 `_id` 已经通过手动方式设置，不需要由 MongoDB 生成(否则，`_id` 的类型将为对象 ID)。

4.7.2 使用 DBRef 引用数据

DBRef 标准提供了在文档之间引用数据的更正式规范。使用 DBRef 而不是手动引用的主要原因是，引用中文档所在的集合可能发生变化。所以，如果引用的一直都是相同的集合，那么手动引用数据也可以。

使用 DBRef 可以将数据库引用存储为标准的嵌入对象(JSON/BSON)。使用一种标准方式代表引用，意味着驱动和数据框架可以添加辅助方法，以标准的方法操作引用。

添加 DBRef 引用值的语法如下所示：

```
{ $ref : <collectionname>, $id : <id value>[, $db : <database name>] }
```

这里，`<collectionname>` 代表着被引用集合的名字(例如 `publisherscollection`)；`<id value>` 代表着被引用对象的 `_id` 字段；通过使用可选的 `$db` 可以引用其他数据库中的文档。

下面是一个使用了 DBRef 的样例。首先清空两个集合并添加一个新的文档：

```
> db.publisherscollection.drop()  
true  
> db.media.drop()  
true  
> apress = ( { "Type" : "Technical Publisher", "Category" :  
["IT", "Software", "Programming"] } )  
{  
  "Type" : "Technical Publisher",  
  "Category" : [  
    "IT",  
    "Software",  
    "Programming"  
  ]  
}  
> db.publisherscollection.save(apress)
```

到目前为止，我们已经定义了变量 `apress`，并使用 `save()` 函数保存它。接下来，通过输入变量的名字显示出它的内容：

```
> apress  
{  
  "Type" : "Technical Publisher",  
  "Category" : [  
    "IT",  
    "Software",  
    "Programming"  
  ],  
  "_id" : ObjectId("4c4597e98e0f000000006290")  
}
```

另外之前还定义了发布者，并将它保存在 `publisherscollection` 集合中。现在，在 `media` 集合中添加一个引用该数据的文档：

```

> book = { "Type" : "Book", "Title" : "Definitive Guide to MongoDB 2nd ed., The",
"ISBN" : "978-1-4302-5821-6", "Author": ["Hows, David", "Membrey, Peter", "Plugge,
Eelco", "Hawkins, Tim"], "Publisher" : [ new DBRef ('publisherscollection',
apress._id) ] }
{
  "Type" : "Book",
  "Title" : "Definitive Guide to MongoDB 2nd ed., The",
  "ISBN" : "987-1-4302-5821-6",
  "Author" : [
    "Hows, David",
    "Membrey, Peter",
    "Plugge, Eelco",
    "Hawkins, Tim"
  ],
  "Publisher" : [
    DBRef("publishercollection", "Apress")
  ]
}
> db.media.save(book)

```

这就是所有的代码！该例看起来比手动引用数据的方式简单一些；不过，如果需要修改文档所在的集合，那么它是一个很好的选择。

4.8 使用索引相关的函数

在之前的章节中，你已经学习了索引在数据库中的用途。现在开始学习如何创建和使用索引。第10章将对索引进行详细讲解，现在只学习一些基础知识。MongoDB 中包含了许多可用于维护索引的函数；接下来首先将使用 `ensureIndex()` 函数创建索引。

函数 `ensureIndex()` 可以接受至少一个参数，该参数将指定文档中某个键的名字，用于构建索引。在之前的样例中，曾向 `media` 集合中添加了一个使用 `Title` 键的文档。基于该键的索引将使 `media` 集合的使用更加方便。

小窍门：

在 MongoDB 中创建索引的经验法则是，对于相同的场景，如果在 MySQL 中需要创建索引，那么同样也需要在 MongoDB 中创建。

通过调用下面的命令可以为 `media` 集合创建索引：

```
> db.media.ensureIndex( { Title : 1 } )
```

该命令将确保基于 `media` 集合的所有文档的 `Title` 值创建一个索引。行尾的 `:1` 指定了索引的方向，`:1` 表示升序，`:-1` 表示降序。

```
// 创建升序索引
db.media.ensureIndex( { Title :1 } )

// 创建降序索引
db.media.ensureIndex( { Title :-1 } )
```

小窍门：

搜索索引信息是非常快速的。搜索非索引信息就很慢，因为需要检查每个文档并判断是否匹配。

BSON 允许在文档中存储完整的数组；如果能够在内嵌的键上创建索引，那将会非常方便。幸运的是，MongoDB 的开发者也想到了这一点，并提供了对该特性的支持。现在我们将基于本章之前的样例构建出新的样例，在数据库中添加另一个含有内嵌信息的文档：

```
> db.media.insert( { "Type" : "CD", "Artist" : "Nirvana", "Title" :
"Nevermind", "Tracklist" : [ { "Track" : "1", "Title" : "Smells Like Teen
Spirit", "Length" : "5:02" }, { "Track" : "2", "Title" : "In Bloom", "Length" :
"4:15" } ] } )

{ "_id" : ObjectId("4c45aa2f8e0f000000006293"), "Type" : "CD", "Artist" :
"Nirvana", "Title" : "Nevermind", "Tracklist" : [
  {
    "Track" : "1",
    "Title" : "Smells Like Teen Spirit",
    "Length" : "5:02"
  },
  {
    "Track" : "2",
    "Title" : "In Bloom",
    "Length" : "4:15"
  }
] }
}
```

接下来，为曲目列表的所有目录中的 Title 键创建一个索引：

```
> db.media.ensureIndex( { "Tracklist.Title" : 1 } )
```

下次在集合中搜索任何一个标题时——假设它们都在 Tracklist 嵌套数组中——标题将会迅速显示出来。接下来，再次采用该概念，使用整个(子)文档为键，如下所示：

```
> db.media.ensureIndex( { "Tracklist" : 1 } )
```

该语句将为数组中的所有元素构建索引，这意味着可以搜索数组中的任何对象。这些键的类型也被称为多键。还可以基于一组文档的多键创建索引。该过程也被称为复合索引。创建复合索引的方法基本上是一样的；不同之处在于，创建复合索引需要指定多个键，而不是一个键，如下所示：

```
> db.media.ensureIndex({"Tracklist.Title": 1, "Tracklist.Length": -1})
```

这种方式的优点在于，可以基于多个键创建索引(如之前为整个子文档创建索引的样例所示)。与索引子文档的方式不同，通过复合索引可以指定是否以降序的方式为其中一个字段创建索引。如果使用子文档的方式执行索引，就只能使用升序或降序。在第 10 章将对复合索引进行深入讲解。

4.8.1 学习索引相关的命令

到目前为止，你已经学习了索引相关的命令 `ensureIndex()`。毫无疑问，这是用于创建索引的主要方法。另外，还有一对与索引相关的函数：`hint()`和`min()/max()`。这些函数将用于查询数据。之前并未讲解它们的相关知识，因为没有自定义索引它们是无法工作的。现在我们将学习如何使用它们。

4.8.2 强制使用某个索引查询数据

通过 `hint()`函数可以强制使用某个指定的索引查询数据。使用该命令的目的是提高查询性

能。为了演示该函数的使用，下面的样例在不定义索引的情况下，结合使用 `find()`和 `hint()`函数来查询数据：

```
> db.media.find( { ISBN: " 978-1-4302-5821-6" } ) . hint ( { ISBN: -1 } )
error: { "$err" : "bad hint", "code" : 10113 }
```

如果基于 ISBN 书号创建了索引，该例的执行将更加成功。注意，下面第一个命令中的 `background` 参数将保证索引在后台完成：

```
> db.media.ensureIndex({ISBN: 1}, {background: true});
> db.media.find( { ISBN: " 978-1-4302-5821-6" } ) . hint ( { ISBN: 1 } )

{ "_id" : ObjectId("4c45a5418e0f000000006291"), "Type" : "Book", "Title" : "Definitive Guide to
MongoDB, The", "ISBN" : " 978-1-4302-5821-6", "Author" : ["Hows, David","Membrey, Peter",
"Plugge, Eelco","Hawkins,Tim"], "Publisher" : [
  {
    "$ref" : "publisherscollection",
    "$id" : ObjectId("4c4597e98e0f000000006290")
  }
] }
```

为了确定使用的是指定的索引，可以选择使用 `explain()`函数，返回所选择的查询计划的相关信息。这里，`indexBounds` 值将显示出所使用的索引：

```
> db.media.find( { ISBN: " 978-1-4302-5821-6" } ) . hint ( { ISBN: 1 } ) . explain()
{
  "cursor" : "BtreeCursor ISBN_1",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 1,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 1,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {
    "ISBN" : [
      [
        {
          "$minElement" : 1
        },
        {
          "$maxElement" : 1
        }
      ]
    ]
  },
  "server" : "localhost:27017"
}
```

4.8.3 限制查询匹配

函数 `min()`和 `max()`用于限制查询匹配，只有在指定的 `min` 和 `max` 键之间的索引键才会返回。因此，需要为指定的键创建索引。另外，可以结合使用这两个函数，或者分别使用它们。下面

先添加一些文档，用于演示如何使用这些函数。首先在 Released 字段上创建一个索引：

```
> db.media.insert( { "Type" : "DVD", "Title" : "Matrix, The", "Released" : 1999 } )
> db.media.insert( { "Type" : "DVD", "Title" : "Blade Runner", "Released" : 1982 } )
> db.media.insert( { "Type" : "DVD", "Title" : "Toy Story 3", "Released" : 2010 } )
> db.media.ensureIndex( { "Released": 1 } )
```

如下所示，现在可以使用 max()和 min()命令：

```
> db.media.find() . min ( { Released: 1995 } ) . max ( { Released : 2005 } )
{ "_id" : ObjectId("4c45b5b38e0f0000000062a9"), "Type" : "DVD", "Title" :
"Matrix, The", "Released" : 1999 }
```

如果未创建索引，该命令将返回一条错误消息，表示无法为指定的键模式找到索引。很明显，需要使用 hint()函数来定义必须采用哪个索引：

```
> db.media.find() . min ( { Released: 1995 } ) .
max ( { Released : 2005 } ) . hint ( { Released : 1 } )
{ "_id" : ObjectId("4c45b5b38e0f0000000062a9"), "Type" : "DVD", "Title" :
"Matrix, The", "Released" : 1999 }
```

注意：

结果中将包含 min()值，但不包含 max()值。

一般来说，建议使用 \$gt 和 \$lt(分别是大于和小于)而不是 min()和 max()，因为 \$gt 和 \$lt 不要使用索引。函数 min()和 max()主要用于复合键。

4.9 小结

在本章，我们学习了最常用的命令和选项，在 MongoDB shell 中可通过它们操作数据。还学习了如何搜索、添加、修改和删除数据，以及如何修改集合和数据库。接下来，又学习了原子操作、聚集，以及何时使用 \$elemMatch 这样的操作符。最后，学习了如何创建索引和使用它们。还学习了应该使用什么索引、如何删除它们、如何使用已创建的索引搜索数据，以及如何检查运行的索引操作。

下一章将学习 GridFS 的基础知识，包括它的定义、功能及用法。

第 5 章

GridFS

在这个高清视频、1200 万像素摄像头以及可在磁盘上存储 50GB 数据的存储媒介(大小与 CD-ROM 相同)普遍存在的世界上, MongoDB 将文档大小的上限设置为 16MB 似乎并不够用。确实,你可能会好奇为什么像 MongoDB 这样一个在高科技时代设计出的数据库,却有着这样看似奇怪的限制。最简单的答案是:性能。

如果将数组存储在文档中,文档明显会变得非常大,从而导致数据难于处理。例如,读取整个文档也会要求加载文档中所有的文件。尽管可以解决这个问题,但是无论何时访问,都仍然需要读取整个文件,即使实际所需要的仅仅是其中一小部分。不能只读取文档中间的一块数据——要么全部读取,要么全部不读取。幸运的是, MongoDB 为该问题提供了一个独特的并且还有点优雅的解决方案。通过 MongoDB 可以轻松存储大文件,并且可以只访问部分文件,而不是读取完整的文件,从而保持高性能。MongoDB 通过 GridFS 规范实现该功能。

注意:

关于 GridFS 有一件有意思的事情,它并不是一个真正的软件特性。例如, MongoDB 中并没有任何专门用于管理 GridFS 的服务器端代码。相反, GridFS 是由 MongoDB 的所有驱动使用的一个简单规范。这样的规范最大的优势在于,由一个驱动存储的文件可以被另一个使用了相同规范的驱动使用。

这种方式与 MongoDB 保持简单的原则一脉相承。因为 GridFS 使用标准的 MongoDB 特性,因此从驱动的角度来看,它易于实现和使用。它还意味着在需要的时候可以手动操作,因为 GridFS 规范中的 MongoDB 文件只是一些包含文档的普通集合。

5.1 背景

第 1 章提到了一个事实,在过去的许多年中,即使是简单的存储我们也会使用数据库来完成。例如,我们曾购买过一本 PHP 方面的书籍,在第 3 章就介绍了如何使用 MySQL 存储数据,而这已经是 15 年前的事了。考虑到事实上(不是理论)SQL 和数据库的复杂度,你可能会好奇为什么作为一本面向初学者的书籍却会从学习使用 SQL 开始。毕竟,它是一本 PHP 书籍而不是 MySQL 书籍。

大多数人并未意识到的是,直接在磁盘上读取和输入数据是非常困难的。某些人并不认同这一点,毕竟,在 Python 中打开和读取文件并不复杂。确实,在简单的场景中,通过 PHP 操作文件并不麻烦。如果只是希望从文件中读取一些数据并处理它们,那么你可能不会有任何麻烦。

另一方面,如果希望搜索文件或者存储复杂的/结构化的数据,事情就会变得困难得多。即

使可以通过某种方式解决该问题，但与使用数据库比起来，你的解决方法可能就不够快速或高效。在今天，应用的运行依赖于快速地搜索和存储数据，对于我们这些无法自己编写出这样一个系统的人来说，数据库可以帮助实现这些操作。

许多书籍都讲解了文件的存储方式。其中教你使用数据库存储数据的大部分书籍，也都会教你通过读写文件系统的方式存储文件。在某些情况下，这并不是问题，因为读写简单文件比处理它们要简单。不过，这样也有一些问题。首先，开发者必须有权限读写这些文件，并且要求为 Web 服务器赋予本地文件系统的写权限。这看起来似乎不会引起问题，但这将给系统管理员带来噩梦——将文件读取到服务器只是处理它的第一阶段。

数据库可以存储二进制文件；通常，这么做并不优雅。MySQL 还有一种特殊的列类型，称为 BLOB。PostgreSQL 要求使用特殊的存储过程存储这样的文件——数据并不存储在表中。换句话说，这种方式是混乱的。这些解决方案明显是后补充上的。因此，人们选择将数据写入磁盘并不奇怪。但这种方式也有问题。除了安全问题，它还添加了另一个需要备份的目录，必须保证该信息被复制到所有正确的服务器。现在有些系统可以提供写入磁盘并完全复制这些内容的能力(包括 GFS)，但这些解决方案非常复杂并且增加了开销；而且，这些特性通常会使你的解决方案难于维护。

另一方面，MongoDB 强制要求文件的大小不得超过 16MB。这个大小对于存储富文档是足够的，对于许多其他类型的文件，在过去的许多年中也可以满足。不过，该限制对于今天的环境就显得不足了。

5.2 使用 GridFS

接下来，你将学习 GridFS 的实现。MongoDB 网站指出，在使用 GridFS 时并不需要理解或意识到它内在的实现方式。事实上，由驱动完成这些复杂的工作即可。大部分支持 GridFS 的驱动都将以特定语言的方式实现文件处理。例如，MongoDB Python 驱动的处理方式与 Python 的工作方式一致。如果你对 GridFS 的细节不感兴趣，可以直接跳到下一节。我们保证你不会错过任何东西，并且你仍然可以高效地使用 MongoDB！

GridFS 由两部分组成。更具体地说，它由两个集合组成。一个集合用于存储文件名和诸如大小这样的相关信息(称为元数据)，而另一个集合保存文件数据自身，以 256KB 为一块。该规范将这两个集合分别称为 `files` 和 `chunks`。默认情况下，`files` 和 `chunks` 集合在 `fs` 名称空间中创建，但这可以修改。如果希望存储不同类型的文件，那么修改默认名称空间的能力就非常有用。例如，可以将图像和电影文件分开存储。

5.3 开始使用命令行工具

之前已经了解了 GridFS 的一些背景，现在开始学习如何使用它的命令行工具。首先，需要创建一个文件。为了保持简单，可以直接使用词典文件。在 Ubuntu 中，可以在 `/usr/share/dict/words` 目录中找到该文件。不过，其中还包含了各种不同级别的符号链接，所以可以先运行以下命令：

```
root@core2:/usr/share/dict# cat words > /tmp/dictionary
```

注意:

在 Ubuntu 中，可以使用 `apt-get install wbritish` 安装该词典文件。

该命令将复制文件的所有内容到一个可以轻松使用的简单文件中。当然，对于本例可以使用任何希望使用的文件；不需要使用特定大小或类型的文件。

这里先不描述所有可用于 `mongofiles` 的选项，让我们首先从使用该工具的一些特性开始。本书假设你已经在同一机器上同时运行了 MongoDB 和 `mongofiles`。如果没有，那么需要使用 `-h` 选项指定运行 MongoDB 的主机。在学会使用 `mongofiles` 之后，再开始学习它的其他可用选项。

首先，在数据库中列出所有文件。数据库中不应该有任何文件，但让我们再确认一下。命令 `list` 将列出数据库中的所有文件：

```
$ mongofiles list
connected to: 127.0.0.1
$
```

这看起来并不新奇。记住，`mongofiles` 是一个概念验证工具；它可能并不是一个应该在自己应用中使用的工具。不过，`mongofiles` 是一个很棒的学习和测试工具。一旦创建文件，就可以使用该工具遍历创建出的文件和块。

下面开始使用它，并通过 `put` 命令添加之前创建的词典文件(记住：本例可以使用任何你喜欢的文件)：

```
$ mongofiles put /tmp/dictionary
connected to: 127.0.0.1
added file: { _id: ObjectId('51cb61b26487b3d8ce7af440'), filename: "/tmp/dictionary", chunkSize:
262144, uploadDate: new Date(1372283314621), md5: "40c0825855792bd20e8a2d515fe9c3e3",
length: 4953699 }}
done!
$
```

该例将返回一些有用的信息；不过，让我们再次检查数据库以确认文件得到存储。通过运行 `list` 命令可完成：

```
$ mongofiles list
connected to: 127.0.0.1
/tmp/dictionary 4953699
$
```

该例显示了词典文件及其大小。上述信息明显来自于 `files` 集合，但我们提前了一步。现在先回退一步，检查一下本例中 `put` 命令返回的输出。

5.3.1 使用 `_id` 键

如你所知，MongoDB 中的每个文档都在 `_id` 键中存储了一个唯一标识符。如同 MySQL 的自增字段一样，`_id` 键并不需要直接使用，除非需要使用它获取特定的文件。

5.3.2 使用文件名

命令 `put` 的输出中还显示出了键 `Filename`。在这里讲解一下该键。通常，你会希望保持该字段唯一，从而防止混淆；不过，并不需要这样做。事实上，如果再次运行 `put` 命令，将会出

现两个看起来一致的文档。此时，这些文件和元数据都是一致的，除了 `_id` 键。你可能会感到好奇，奇怪 MongoDB 为什么没有选择更新已有文档，而是创建一个新的文档。原因是：在许多情况下都可能会出现同名的文件。例如，如果构建了一个系统用于存储学生作业，那么很有可能至少其中一些文件会是重名的。MongoDB 不能假设一致的文件名(甚至大小也相同)代表着相同的文件。因此，如果 MongoDB 更新已有的文件，那么极有可能会引起错误。当然，你可以使用 `_id` 键更新某个特定的文件；在接下来基于 Python 的实验中讲解更多关于该话题的内容。

5.3.3 决定文件长度

命令 `put` 还返回了文件的长度，它既是有用的信息，也是 GridFS 如何工作的关键。

在编写自己的应用时，文件的大小不仅仅是参考，而且还起到了非常大的作用。例如，在 Web 中发送文件时(例如，通过 HTTP)，就需要指定文件的大小。也不是所有的服务器都会这么做；例如，从特定网站下载文件时，你可能已经注意到，浏览器可以显示出下载文件的速度，但无法显示完成下载所需的时间。这是因为服务器未提供文件大小信息。

知道文件的大小还有另外一个重要的作用。之前，我们曾提到文件将被拆分为块，也就是说将文件分成更小的碎片。默认情况下，块的大小为 256KB，但如果愿意，可以将其修改为另一个值。为了判断出一个文件占用了多少个块，就需要知道两个信息。第一必须知道每个块的大小；第二必须知道文件大小，这样才可以知道该文件占用了多少个块。

你可能认为这并不重要。毕竟，如果现在有一个文件大小为 1MB，块大小为 256KB，那么如果希望从 800KB 的位置开始访问数据，那就需要从第 4 块开始读取数据。然而你仍然需要知道该文件的总大小，原因是：如果不知道文件的大小，就不能判断出到底有多少有效的块。在刚才的例子中，你完全可能从 1.26MB(第 6 块)开始访问数据，并且没有什么机制可以阻止你。在这种情况下，第 6 块是不存在的，但如果没有文件大小信息，就无法知道这一点。当然，驱动会帮你完成所有这些事情，所以你并不需要过于担心；不过，知道 GridFS 在幕后是如何工作的，有助于帮助调试个人应用。

5.3.4 使用块大小

命令 `put` 还将返回块的大小，尽管已经有了一个默认值，但随着文件的不同，块的大小也可能被修改为不同的值。这样，块的大小就可以非常灵活。如果网站使用的是流式视频，那么可以将视频分成许多块，这样就可以轻松跳到指定视频的任意一部分。如果现在有一个大文件，并且需要返回整个文件，那就需要找到文件指定区域的起始点。有了 GridFS，就可以在块级别取出所有数据。如果使用的是默认大小，那么可以从任何一个 256KB 块开始读取数据。当然，也可以指定实际希望读取的数据位(例如，只希望查看一个 6 分钟电影中的第 5 分钟)。这是一个非常高效的系统，256KB 在大多数情况下也是一个非常好的块大小。如果决定要修改它，那么你应该有一个好的理由。并且不要忘记建立基准并测试自定义块大小的性能；理论上更好的系统却无法达到预期，这种情况并不少见。

注意：

MongoDB 文档有 16MB 的大小限制。因为 GridFS 只是标准 MongoDB 框架下存储文件的一种不同方式，所以它也受此限制。这就意味着不能创建大于 16MB 的块，这不会构成问题，因为 GridFS 的目的就在于缓解对大文件的需求。如果担心当前存储的大文件会生成太多的块文档，那么不用担心。目前，在生产环境中已经有 MongoDB 系统在使用超过 10 亿个文档。

5.3.5 处理上传日期

顾名思义，键 `uploadDate` 的作用是：存储文件在 MongoDB 中创建的日期。这也表明 `files` 集合只是一个普通的 MongoDB 集合，它包含一些普通的文档。这意味着可以在其中添加任何额外的键值对(如果需要的话)，与使用任何其他集合的方式相同。

例如，考虑一个真实世界的应用，该应用需要存储从各种不同文件中提取出的文本内容。通过这种方式，可以执行一些额外的索引和搜索。为了实现该任务，可以添加一个键 `file_text` 用于存储文本。GridFS 系统的优雅性意味着，可以用这个系统完成任何其他 MongoDB 文档可以完成的事情。优雅和强大是使用 MongoDB 的两个突出优势。

5.3.6 生成文件的哈希值

MongoDB 提供了 MD5 哈希算法。从网络下载软件时，你可能已经遇到过该算法。MD5 背后的理论是每个文件都有唯一的签名。修改文件的任何一点内容都将引起签名的巨大变化。使用该签名有两个原因：安全和完整性。出于安全的目的，如果已经知道原始的 MD5 哈希值且相信文件的来源(可能是朋友发送给你的)，并且如果对文件重新计算后的哈希值(通常称为校验和)并未改变，那就表示文件未被修改。这也保证了文件的完整性，因为没有数据丢失或损坏。特定文件的 MD5 哈希值就像文件的指纹。哈希值还可以用于区分不同名但内容相同的文件。

警告：

MD5 算法已不再被认为是安全的，并且已经证实可以创建出两个具有相同 MD5 校验和的不同文件，它们的内容可以完全不同。在加密数据中，这被称为冲突。这样的冲突是糟糕的，因为这意味着攻击者可以通过这种方式修改文件，而无法被检测到。该警告稍微有点过去理论性，因为为了故意制造出这样的冲突，需要花费大量的精力和时间；并且修改后的文件可能会与原始文件完全不同，因此可以轻松识别出来。出于该原因，MD5 仍然是判断文件完整性的首选方法(MD5 已经得到广泛支持)。不过，如果希望哈希用于安全目的，最好还是使用 SHA 系列规范——更理想的是 SHA-256 和 SHA-512。尽管这些哈希规范有一些理论上的缺陷；不过，尚未有人能够构建出 SHA 系列哈希冲突的实际例子。MongoDB 使用 MD5 用于确保文件的完整性，在大多数情况下都是没问题的。不过，如果希望对一些重要数据(用户密码)进行哈希计算，应该考虑使用 SHA 系列哈希算法。

5.4 查看 MongoDB 中的数据

此时，MongoDB 数据库中已经存储了一些数据。现在我们将查看 MongoDB 数据库中存储的数据。再次使用命令行工具连接到数据库并查询它。例如，针对之前创建的文件运行 `find()` 命令：

```
$ mongo test
MongoDB shell version: 2.5.1-pre
connecting to: test
db.fs.files.find()
{ "_id" : ObjectId("51cb61b26487b3d8ce7af440"), "filename" : "/tmp/dictionary",
"chunkSize" : 262144, "uploadDate" : ISODate("2013-06-26T21:48:34.621Z"), "md5" :
"40c0825855792bd20e8a2d515fe9c3e3", "length" : 4953699 }
>
```

输出看起来很熟悉，因为数据与本章之前看到的一样。现在你会发现，`mongofiles` 输出的信息来自于 `fs.files` 集合中该文件所在的条目。

接下来，查看 `chunks` 集合(必须添加一个过滤器；否则，也将显示出大量原生二进制数据)：

```
$ mongo test
MongoDB shell version: 2.5.1-pre
connecting to: test
> db.fs.chunks.find({}, {"data":0});
{ "_id" : ObjectId("51cb61b29b2daad9857ca205"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"),
"n" : 4 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca206"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 5 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca207"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 6 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca208"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 7 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca209"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 8 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca20a"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 9 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca20b"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 10 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca20c"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 11 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca20d"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 12 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca20e"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 13 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca20f"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 14 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca210"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 15 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca211"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 16 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca212"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 17 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca201"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 0 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca202"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 1 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca203"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 2 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca204"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 3 }
{ "_id" : ObjectId("51cb61b29b2daad9857ca213"), "files_id" : ObjectId("51cb61b26487b3d8ce7af440"), "n" : 18 }>
```

你会好奇为什么输出这么多的条目。如前所述，`GridFS` 只是一个规范。这意味着它将使用 `MongoDB` 提供的数据库。在我们测试本书的命令时，词典文件被添加了数次。之后，在我们清空 `fs.files` 集合时，该文件被删除。然后你就看到了现在的输出！事实上，从一个集合中删除某些文档与另一个集合没有任何关系。记住：这些文档和集合并未受到 `MongoDB` 的特殊处理。如果要通过驱动或 `mongofiles` 工具正确地删除文件，该工具也会清除 `chunks` 集合。

警告：

直接访问文档和集合是一个强大的特性，但需要小心。因为该特性也容易让你自讨苦吃。如果决定手动编辑文档和集合，一定要确保知道自己在做什么并且进行大量测试。另外要记住，支持 `GridFS` 的 `MongoDB` 驱动并不知道已经完成的这些自定义操作。

5.4.1 使用搜索命令

接下来学习 MongoDB 的搜索命令。到目前为止，数据库中只有一个文件，这将限制我们可执行的搜索类型！因此再添加一些新的文件。下面的代码将把词典文件复制成另一个文件，然后导入该文件：

```
$ cp /tmp/dictionary /tmp/hello_world
$ mongofiles put /tmp/hello_world
connected to: 127.0.0.1
added file: { _id: ObjectId('51cb63d167961ebc919edbd5'), filename: "/tmp/hello_world",
chunkSize:262144, uploadDate: new Date(1372283858021), md5: "40c0825855792bd20e8a2d515fe9c3e3",
length:4953699 }done!
root@core2:~# mongofiles list
connected to: 127.0.0.1
/tmp/dictionary 4953699
/tmp/hello_world 4953699
$
```

第一行命令复制生成了一个文件，第二行命令将该文件导入 MongoDB 中。和之前的样例一样，put 命令将为你输出 MongoDB 已经创建的新文档。接下来，可以运行 mongofiles 的 List 命令，检测文件是否已经被正确存储到数据库中。结果将显示出集合中有两个文件；毫不奇怪的是这两个文件的大小相同。

搜索命令的工作方式将如预期一样。告诉 mongofiles 希望搜索的文件，然后它将会尝试找到该文件，如下所示：

```
$ mongofiles search hello
connected to: 127.0.0.1
/tmp/hello_world 4953699
$ mongofiles search dict
connected to: 127.0.0.1
/tmp/dictionary 4953699
$
```

这里并未发生什么令人激动的事情。不过，这里还有一点非常重要。MongoDB 可以如你所需的一样简单或复杂。工具 mongofiles 仅供参考使用，它包含非常基础的调试信息。好消息是：在 MongoDB 中通过它可以轻松地执行一些针对文件的搜索任务。更好的消息是：MongoDB 还可以执行一些极其复杂的搜索任务。

5.4.2 删除

mongofiles 的 delete 命令就不需要再额外解释了，但有必要给一个大的警告。该命令将基于文件名删除文件。因此，如果同时有多个同名文件，那么该命令将删除所有文件。下面的代码演示了如何使用 delete 命令：

```
$ mongofiles delete /tmp/hello_world
connected to: 127.0.0.1
$ mongofiles list
connected to: 127.0.0.1
/tmp/dictionary 4953699
$
```

注意：

许多人都表示这个删除多个同名文件的问题不是真正的问题，因为没有应用会含有重复的名字。这并不是真的；在许多情况下，强制唯一的名字是不合理的。例如，如果应用允许用户上传相片到他们的个人资料中，那么应用收到的文件极有可能是 `poto.jpg` 或 `me.png`。

当然，如果不喜欢使用 `mongofiles` 管理在线数据——确实没有人希望使用这种方式——那么只需要在删除数据时小心一点即可。

5.4.3 从 MongoDB 中获取文件

到目前为止，我们还未从 MongoDB 中真正读取过数据。任何数据库最重要的特性就是：一旦将数据存储到数据库中，就可以通过它搜索和读取数据。下面的代码将使用 `mongofiles` 的 `get` 命令从 MongoDB 中读取文件：

```
$ mongofiles get /tmp/dictionary
connected to: 127.0.0.1
done write to: /tmp/dictionary
$
```

该例故意包含了一个错误。因为指定了希望读取文件的全名和文件路径(按照需要)，`mongofiles` 将把数据写入到含有相同名字和路径的文件中。事实上，该命令将覆写原始的词典文件！这不会造成太大的损失，因为只是在覆写相同的文件，并且该文件只是原始文件的一个临时副本。不过，如果你不小心覆写了两个星期的工作，此时你一定非常震惊。相信我们，直到发生某个事件，你才会注意到你所有的工作成果都已经丢了！如同使用 `delete` 命令一样，在使用 `get` 命令时一定要小心。

5.4.4 mongofiles 命令小结

`mongofiles` 是一个可用于快速查看数据库的有用工具。如果你编写了一些软件，但怀疑其中可能存在问题，那么可以使用 `mongofiles` 对数据进行复查。

它是非常简单的实现，所以不需要任何花哨的逻辑，从而避免使手头的工作变得复杂。是否在生产环境中使用 `mongofiles` 取决于个人偏好。它并不完全是一把瑞士军刀；不过，它确实提供了一些有用的命令集，可用于帮助分析应用中出现的问题。简单地说，你应该熟悉这个工具，因为某天它没准就可以帮你解决一个非常棘手的问题。

5.5 使用 Python

此时你已经清楚地知道 GridFS 是如何工作的。接下来，我们将学习如何通过 Python 访问 GridFS。第 2 章已经讲解了如何安装 PyMongo；如果在接下来的例子中遇到任何问题，可以参阅第 2 章，检查是否所有软件都已经正确安装。

如果已经执行了本章之前的样例，那么现在 GridFS 中应该已经有一个文件了。该文件是一个词典文件，它包含一个单词的列表。在本节，将学习如何编写一个简单的 Python 脚本，输出词典文件中的所有单词。当然，使用 `cat` 命令查看原始文件更简单和高效，但这样就没有乐趣可言了。

首先启动 Python:

```
Python 2.6.6 (r266:84292, Oct 12 2012, 14:23:48)
[GCC 4.4.6 20120305 (Red Hat 4.4.6-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.>>>
```

Python 版本的标准驱动称为 PyMongo, 由 Mike Dirolf 编写。因为 PyMongo 驱动由发布 MongoDB 的 MongoDB Inc. 公司支持, 所以可以保证它会得到定时升级和维护。现在继续下一步, 导入必要的 gridfs 库。应该显示出以下信息:

```
>>> from pymongo import Connection
>>> import gridfs
>>>
```

如果未正确安装 PyMongo, 那么将会显示出以下类似错误:

```
>>> import gridfs
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ImportError: No module named gridfs
>>>
```

如果看到了后面的消息, 说明在安装过程中丢失了一些信息。在这种情况下, 请返回第 2 章, 并按照命令重新安装 PyMongo。

5.5.1 连接数据库

在从数据库读取数据之前, 首先必须建立到数据库的连接。在本章之前使用 mongofiles 工具时, 你会注意到它将连接到 127.0.0.1。该值也被称为 localhost, 代表本机回环地址。这是一条高速计算机连接到自身的捷径。mongofiles 提到该 IP 地址的原因是它将通过网络连接该地址中存在的 MongoDB。默认是连接本地机器的默认端口。因为并未修改默认的设置, mongofiles 将搜索并连接到本地数据。

不过, 在通过 Python 使用 MongoDB 时, 需要连接到数据库, 然后设置 GridFS。幸运的是, 这很简单:

```
>>> db = Connection().test
>>> fs = gridfs.GridFS(db)
>>>
```

第一行打开了连接并选择数据库。默认情况下, mongofiles 将使用 test 数据库; 因此, 在 test 数据库中将可以找到词典文件。第二行对 GridFS 进行设置并准备使用它。

5.5.2 访问单词

在 PyMongo 驱动力的原始实现中, 它通过与文件操作类似的接口使用 GridFS。这有点不同于之前 mongofiles 的方式(有点接近于 FTP 的方式)。在 PyMongo 的原始实现中, 可以如同操作普通文件一样读写数据。

这使 PyMongo 与 Python 的使用方式非常相似, 可以轻松与现有的脚本进行集成。不过, 该行为在驱动的 1.6 版本中有了改变, PyMongo 不再支持该功能。因为这种与 Python 类似的行为导致了一些问题, 使该工具变得不够高效。

一般来说, PyMongo 驱动尝试使 GridFS 文件看起来并且感觉与文件系统的普通文件一样。一方面这是有利的, 因为意味着没有学习曲线, 并且该驱动可完成任何文件操作。另一方面, 这种方式也有些限制, 无法体现出 GridFS 的强大。在 PyMongo 的 1.6 版本中, 它的工作方式发生了重大改变, 尤其是 `get` 和 `put` 命令。

注意:

该版本的 PyMongo 与该工具的之前版本并未有太多的不同, 许多使用之前版本 API 的人发现很容易适应修订后的版本。尽管如此, 并不是所有人都能适应 Mike 所做的修改。例如, 一些人发现旧 API 中基于文件的机制极其易用。PyMongo 修订后的版本支持创建文件名, 所以缺失的行为可以被复制到修改版中; 不过, 这样做要求使用一些额外的代码。

5.6 在 MongoDB 中添加文件

通过 PyMongo 向 GridFS 中添加文件是非常直接的, 并且类似于命令行的使用方式。MongoDB 的主要目标是吞吐量, 所以 PyMongo 修订版中的 API 也体现了这一点。通过这些修改, PyMongo 不仅获得了更好的性能, 还使 Python 驱动与其他 GridFS 实现保持一致。

现在将词典文件再次添加到 GridFS 中:

```
>>> with open("/tmp/dictionary") as dictionary:
...     uid = fs.put(dictionary)
...
>>> uid
ObjectId('51cb65be2f50332093f67b98') >>>
```

本例使用 `put` 方法添加该文件。最重要的是要从该方法中获得结果, 因为它包含文件的 `_id`。mongofiles 假设文件名就是所添加文件的键(哪怕可能会出现重复), 而 PyMongo 使用的方式与它不同。它将基于文件的 `_id` 引用它们。如果未获得该信息, 那就无法以可靠的方式再次找到该文件。事实上, 这也不完全是真的——可以相当轻松地搜索到一个文件——但如果希望将该文件链接到一个特定的用户账户, 那就需要使用 `_id`。

有两个有用的参数可以与 `put` 命令结合使用, 分别是 `filename` 和 `content_type`。如你所想, 通过这两个参数可以分别设置文件名和文件的内容类型。从磁盘中直接加载文件时这是非常有用的。不过, 在处理通过网络接收到或者在内容中生成的文件时, 这种方式更有用, 因为在这些情况下, 通过这种方式可以使用与文件类似的语义, 但不需要真正在磁盘上创建文件。

5.7 从 GridFS 中读取文件

现在终于可以从 GridFS 中读取数据集了! 此时, 你已经得到了该文件的 `_id`, 因此找到它是非常简单的。使用 `get` 方法从 GridFS 中读取文件:

```
>>> new_dictionary = fs.get(uid)
```

之前的代码将返回一个类似于文件的对象; 因此, 可以使用下面的代码打印出词典中的所有单词:

```
>>> for word in new_dictionary:
```

```
... print word
```

现在，你将看到单词列表快速地在屏幕上向上滚动！好的，这并不是真正的高科技。不过，正是由于 GridFS 的存在，才使这件事情变得简单——它确实与宣传的一样，以一种直观和简单易于理解的方式进行工作！

5.8 删除文件

删除文件也很简单。所有需要做的就是调用 `fs.delete()` 并传入该文件的 `_id`，如下所示：

```
>>> fs.delete(uid)
>>> new_dictionary = fs.get(uid)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python2.6/site-packages/pymongo-2.5.2-py2.6-linux-x86_64.egg/gridfs/
__init__.py", line 140, in get return GridOut(self.__collection, file_id)
  File "/usr/lib/python2.6/site-packages/pymongo-2.5.2-py2.6-linux-x86_64.egg/gridfs/
grid_file.py", line 392, in __init__(files, file_id)
  gridfs.errors.NoFile: no file in gridfs collection Collection(Database(Connection
('localhost',27017), u'test'), u'fs.files') with _id ObjectId('51cb65be2f50332093f67b98') >>>
```

这些结果看起来有点吓人，不过它们只是 MongoDB 无法找到该文件时显示出的信息。这并不奇怪，因为你刚刚删除了它！

5.9 小结

本章对 GridFS 进行了快速讲解。我们学习了 GridFS 的定义，它与 MongoDB 结合使用的方法以及它的基本语法。本章并未深入学习 GridFS，但在下一章，你将学习如何使用 PHP 在真实应用中集成 GridFS。对于现在，了解 GridFS 如何处理文件排序或其他大块数据，可以帮助你节省时间和减少麻烦，这已足够。

在下一章中，我们开始真正使用学到的这些知识，尤其是，你将学习创建一个功能齐全的通讯录！

第 II 部分



在开发中使用 MongoDB

第 6 章 PHP 和 MongoDB

第 7 章 Python 和 MongoDB

第 8 章 高级查询

第 6 章

PHP 和 MongoDB

在之前的 5 章内容中，已经讲解了如何在 MongoDB shell 中执行各种操作。例如，如何添加、修改和删除文档。还讲解了 DBRef 和 GridFS 的工作方式，包括如何使用它们。

不过到目前为止，之前学习的大多数操作都是在 MongoDB shell 中完成的。它是一个很强大的应用，但 MongoDB 软件还提供了大量其他驱动(更多详细信息请参阅第 2 章)，通过这些驱动可以在 shell 之外以编程的方式完成许多任务。

其中的一个工具就是 PHP 驱动，如果你希望使用 PHP 而不是 shell，那么可以使用它扩展 PHP 安装包，从而连接、修改和管理 MongoDB 数据库。当你需要设计一个 Web 应用，或者无法访问 MongoDB shell 时，这是非常有用的。正如本章所演示的，大多数可以在 PHP 驱动中执行的操作与在 MongoDB shell 中执行的操作相比，功能非常类似；不过，PHP 要求通过数组的方式指定选项，而不是在两个花括号之间。尽管两者相似，但需要注意使用 PHP 驱动时的一些不同之处。本章将浏览使用 PHP 操作 MongoDB 的一些优点，以及如何适应之前所提到的不同之处。

本章将通过多种方式从头开始演示 MongoDB 的使用。我们首先将学习如何通过 PHP 浏览数据库和集合的使用。接下来学习如何使用 PHP 插入、修改和删除文档。然后学习如何使用 GridFS 和 DBRef；不过这次的焦点在于如何在 PHP 中使用它们，而不是这些技术背后的理论。

6.1 比较 MongoDB 和 PHP 中的文档

如前所述，MongoDB 集合中的文档使用类似于 JSON 的格式(由键和值组成)存储数据。这种方式类似于 PHP 定义关联数组的方式，所以适应这种格式并不难。

例如，假设 MongoDB shell 中的一个文档如下所示：

```
contact = ( {
  "First Name" : "Philip",
  "Last Name" : "Moran",
  "Address" : [
    {
      "Street" : "681 Hinkle Lake Road",
      "Place" : "Newton",
      "Postal Code" : "MA 02160",
      "Country" : "USA"
    }
  ],
  "E-Mail" : [
    "pm@example.com",
    "pm@office.com",
    "philip@example.com",
```

```
        "philip@office.com",
        "moran@example.com",
        "moran@office.com",
        "pmoran@example.com",
        "pmoran@office.com"
    ],
    "Phone" : "617-546-8428",
    "Age" : 60
})
```

在 PHP 中，上面同样的文档将如下所示(键值包含在数组中):

```
$contact = array(
    "First Name" => "Philip",
    "Last Name" => "Moran",
    "Address" =>array(
        "Street" => "681 Hinkle Lake Road",
        "Place" => "Newton",
        "Postal Code" => "MA 02160",
        "Country" => "USA"
    )
    ,
    "E-Mail" =>array(
        "pm@example.com",
        "pm@office.com",
        "philip@example.com",
        "philip@office.com",
        "moran@example.com",
        "moran@office.com",
        "pmoran@example.com",
        "pmoran@office.com"
    )
    ,
    "Phone" => "617-546-8428",
    "Age" => 60
);
```

这两种版本的文档看起来非常相似。明显的区别是，分隔键值的冒号(:)在 PHP 中被替换成了箭头状符号(=>)。语法上的区别并不难适应。

6.2 MongoDB 类

MongDB 的 PHP 驱动中包含 4 个核心类、一些操作 GridFS 的类以及几个代表 MongoDB 数据类型的类。核心类组成了驱动的最重要部分。通过结合使用这些类，可以执行一组非常丰富的命令。这 4 个核心类如下：

- **MongoClient**: 初始化数据库连接并提供数据库服务器命令，例如 `connection()`、`close()`、`listDBs()`、`selectDBs()`和 `selectCollection()`。
- **MongoDB**: 与数据库交互并提供一些类似于 `createCollection()`、`selectCollection()`、`createDBRef()`、`getDBRef()`、`drop()`和 `getGridFS()`的命令。
- **MongoCollection**: 与集合交互，其中包含一些类似于 `count()`、`find()`、`findOne()`、`insert()`、`remove()`、`save()`和 `update()`的命令。

- **MongoCursor**: 与 `find()` 命令返回的结果交互, 并包含类似于 `getNext()`、`count()`、`hint()`、`skip()` 和 `sort()` 的命令。

本章将学习之前提到的所有命令; 毫无疑问, 这些命令的使用频率也是最高的。

注意:

本章不会按类分组讨论这些命令; 相反, 我们将按照逻辑顺序对这些命令进行排序。

6.3 连接和断开连接

首先开始学习如何使用 **MongoDB** 驱动连接和选择数据库及集合。使用 **Mongo** 类建立连接, 然后使用它执行数据库服务器命令。下面展示了如何在 **PHP** 中快速连接到数据库:

```
// 连接到数据库
$c = new MongoClient();
//选择希望连接的数据库, 例如 contacts
$c->contacts;
```

Mongo 类还包含了 `selectDB()` 函数, 用于选择数据库:

```
//连接到数据库
$c = new MongoClient();
//选择希望连接的数据库, 例如 contacts
$c->selectDB("contacts");
```

接下来的样例显示了如何选择希望使用的集合。规则与使用 **shell** 时相同: 如果选择了一个不存在的集合, 它将在保存数据时创建。选择目标集合的过程与连接到数据库相同; 换句话说, 使用 (`->`) 语法指向希望访问的集合, 如下所示:

```
//连接到数据库
$c = new Mongo();
// 选择希望连接到的数据库 ('contacts') 和集合 ('people')
$c->contacts->people;
```

通过 `selectCollection()` 函数也可以选择或切换集合, 如下所示:

```
//连接到数据库
$c = new Mongo();
// 选择希望连接到的数据库 ('contacts') 和集合 ('people')
$c->selectDB("contacts")->selectCollection("people");
```

在选择数据库或集合之前, 有时需要首先找到目标数据库或集合。**Mongo** 类包含了两个额外的命令, 分别用于列出可用的数据库和集合。可以通过调用 `listDBs()` 函数获取可用数据库的列表, 并打印到输出中(显示在数组中):

```
//连接到数据库
$c = new Mongo();
// 列出可用的数据库
print_r($c->listDBs());
```

同样, 还可以使用 `listCollections()` 函数获得数据库中可用集合的列表:

```
//连接到数据库
```



```
$c = new Mongo();  
// 列出'contacts'数据库中可用的集合  
print_r($c->contacts->listCollections());
```

注意:

本例中使用的 `print_r` 命令是一个可用于打印数组内容的 PHP 命令。函数 `listDBs()` 将直接返回一个数组，所以该命令可用作 `print_r` 函数的参数。

`MongoClient` 类还包含了 `close()` 函数，用于断开与数据库服务器的 PHP 会话。不过，除非在一些特殊的环境中，一般不需要使用它，因为驱动会在 `Mongo` 对象超出有效范围时，自动关闭数据库连接。

有时你可能不想强制关闭连接。例如，你可能不确定连接的真实状态，或者你希望保证可以创建新的连接。在这种情况下，可以使用 `close()` 函数，如下所示：

```
// Connecting to the database  
$c = new Mongo();  
// Closing the connection  
$c->close();
```

6.4 插入数据

到目前为止，你已经学习了如何建立与数据库的连接。现在开始学习如何在集合中插入数据。在 PHP 中完成该过程与在 MongoDB 中一样。该过程分为两步。首先在变量中定义文档，然后使用 `insert()` 函数插入它。

定义文档并未具体涉及 MongoDB，相反，可以在其中存储含有键值对的数组，如下所示：

```
$contact = array(  
    "First Name" => "Philip",  
    "Last Name" => "Moran",  
    "Address" => array(  
        "Street" => "681 Hinkle Lake Road",  
        "Place" => "Newton",  
        "Postal Code" => "MA 02160",  
        "Country" => "USA"  
    )  
,  
    "E-Mail" => array(  
        "pm@example.com",  
        "pm@office.com",  
        "philip@example.com",  
        "philip@office.com",  
        "moran@example.com",  
        "moran@office.com",  
        "pmoran@example.com",  
        "pmoran@office.com"  
    )  
,  
    "Phone" => "617-546-8428",  
    "Age" => 60  
);
```

警告:

发送到数据库的字符串必须是 UTF-8 格式的字符串，这样可以防止出现异常。

一旦将数据正确地赋给变量——本例中称为 `$contact`——就可以使用 `MongoCollections` 类的 `insert()` 函数将它们插入到集合中:

```
// 连接到数据库
$c = new MongoClient();
// 选择集合'people'
$collection = $c->contacts->people;
//将文档'$contact'插入到代表 people 集合的'$collection'中
$collection->insert($contact);
```

函数 `insert()` 可以接受 5 个选项, 通过数组的方式指定它们: `fsync`、`j`、`w`、`wtimeout` 和 `timeout`。选项 `fsync` 可以是真或假; 假是该选项的默认值。如果设置为真, MongoDB 在确认数据插入成功之前, 将会强制把数据写入硬盘。该选项将把选项 `w` 的任何设置覆写为 0。选项 `j` 可以为真或假, 而假是默认值。如果设置为真, MongoDB 在确认数据插入成功之前, 将强制把数据写入日志。如果不熟悉日志, 可以认为它是一个在数据被写入磁盘之前, 记录数据更改的日志文件。它将保证 `mongod` 在突然停止工作之后, 仍然可以恢复写入到日志中的更改, 从而阻止出现数据不一致的状态。

选项 `w` 可用于确认或不确认写操作(该选项也用于 `remove()` 和 `update()` 函数)。如果设置为 0, 写操作将不会得到确认; 设置为 1, 写操作将会被(主)服务器确认。使用复制集时, `w` 可以被设置为 `n`, 用于确保主服务器在将数据修改成功地复制到 `n` 个节点后确认该写操作。还可以将它设置为 `'majority'`——保留字符串——用于保证复制集的大部分节点或特定标签中的节点都确认该写操作。该选项的默认值为 1。选项 `wtimeout` 可用于指定服务器等待接收确认的时间(以毫秒为单位)。该选项默认被设置为 10000。最后, `timeout` 选项可以指定客户端需要等待服务器响应的超时时间(以毫秒为单位)。

下面的样例将演示如何使用 `w` 和 `wtimeout` 选项插入数据:

```
// 定义另一个联系人
$contact = array(
    "Fir't Name" => "Victoria",
    "Last Name" => "Wood",
    "Address" =>array(
        "Street" => "50 Ash lane",
        "Place" => "Ystradgynlais",
        "Postal Code" => "SA9 6XS",
        "Country" => "UK"
    )
    ,
    "E-Mail" =>array(
        "vw@example.com",
        "vw@office.com"
    ),
    "Phone" => "078-8727-8049",
    "Age" => 28
);
//连接到数据库
$c = new MongoClient();
// 选择 people 集合
```

```
$collection = $c->contacts->people;
//指定 w 和 wtimeout 选项
$options = array("w" => 1, "wtimeout" => 5000);
//将文档'$contact'插入到代表 people 集合的'$collection'中
$collection->insert($contact,$options);
```

这就是使用 PHP 驱动将数据插入数据库中的所有代码。大多数情况下，都需要定义一个包含了数据的数组，而不是将数据插入到数组中。

6.5 查询数据

我们通常使用 `find()` 函数来查询数据。该函数将接受一个参数用于指定搜索条件；一旦指定搜索条件，就可以执行 `find()` 以获得结果。但是，`find()` 函数只是简单地返回集合中的所有文档。这类似于第 4 章中谈到的 shell 样例。不过，大多数情况下我们都不会这么做。相反，一般我们都希望返回特定的数据。下一节将讲解 `find()` 函数中常用的选项和参数，通过它们对结果进行过滤。

6.5.1 返回单个文档

返回单个文档是很简单的：执行 `findOne()` 函数，并且不需要指定任何参数，它将返回在集合中找到的第一个文档。`findOne()` 函数将在数组中存储返回信息，如下所示：

```
//连接到数据库
$c = new MongoClient();
//选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
//找到集合中的第一个文档，并使用 print_r() 命令输出
print_r($collection->findOne());
```

正如之前提到的，列出集合中的单个文档很简单：所有需要做的就是定义 `findOne()` 函数。当然也可以在 `findOne()` 函数中使用额外的过滤器。例如，如果知道目标人物的名字，可以将它用作 `findOne()` 函数的选项：

```
//连接到数据库
$c = new MongoClient();
//选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 在变量$lastname 中定义某个人的姓
$lastname = array("Last Name" => "Moran");
// 找到集合中姓为"Moran"的第一个人
print_r($collection->findOne($lastname));
```

当然，有更多的选项可用于过滤数据；本章稍后将学习这些额外的选项。下面通过使用 `print_r()` 命令输出一些样例(该例为了使代码易于阅读，添加了一些换行符)：

```
Array (
    [_id] => MongoClient Object ( )
    [First Name] => Philip
    [Last Name] => Moran
    [Address] => Array (
        [Street] => 681 Hinkle Lake Road
        [Place] => Newton
        [Postal Code] => MA 02160
```

```

    [Country] => USA
)
[E-Mail] => Array (
    [0] => pm@example.com
    [1] => pm@office.com
    [2] => philip@example.com
    [3] => philip@office.com
    [4] => moran@example.com
    [5] => moran@office.com
    [6] => pmoran@example.com
    [7] => pmoran@office.com
)
[Phone] => 617-546-8428
[Age] => 60

```

6.5.2 列出所有文档

如同使用 `findOne()` 函数列出单个文档一样，可以使用 `find()` 函数列出所有文档。不过不要误解，也可以通过在 `find()` 函数中限制结果的数量来返回单个文档；但如果不确定所返回的文档数目，或者如果希望返回多个文档，那么最好使用 `find()` 函数。

如同之前章节讲解的一样，`find()` 函数有许多可用于过滤结果的选项，它们可适用于任何环境中。下面首先从一些简单的样例开始。

首先，学习如何使用 PHP 和 `find()` 函数显示特定集中的所有文档。唯一需要注意的是，在打印多个文档时，每个文档都将作为一个数组返回，因此要将每个数组单独输出。通过使用 PHP 的 `while()` 函数可以实现该操作。如前所述，需要在处理下一个文档之前输出每个文档。命令 `getNext()` 将从 MongoDB 中获取游标所指向的下一个文档；该命令将以高效的方式返回游标的下一个对象，并向前移动游标。下面的代码列出了集合中所有已找到的文档：

```

//连接到数据库
$c = new MongoClient();
//选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
//执行查询并将结果保存在变量$cursor中
$cursor = $collection->find();
//遍历它在集合中找到的文档，并输出内容
while ($document = $cursor->getNext())
{
    print_r($document);
}

```

注意：

可以通过几种不同的方式实现该例中的语法。例如，执行之前命令的更快捷方式：`$cursor = $c->contacts->people->find()`。不过，为了保持代码清晰，本章的代码样例将一行代码分成了两行，以留出更多的空间用于注释。

此时，如果你已经按照本章之前描述的样例添加了这些文档的话，那么结果输出中将只显示出两个数组。如果添加了更多的文档，那么每个文档都将输出在它自己的数组中。当然，这看起来并不漂亮；不过，只需要一点额外的代码就可以解决这个问题。

6.5.3 使用查询操作符

所有在 MongoDB shell 中可完成的事，都可以通过 PHP 驱动完成。正如之前章节所述，shell 中包含了许多选项可用于过滤结果。例如，可以使用点操作符；对结果排序或限制结果数量；忽略、计数或对一组文档分组；甚至在许多其他选项中应用正则表达式。下面将学习如何在 PHP 驱动中使用其中的大多数选项。

6.5.4 查询特定信息

第 4 章曾讲过，可以使用点操作符查询文档的内嵌对象中的特定信息。例如，如果希望查找某个联系人，但只知道他的地址细节，那么可以使用点操作符找到他，如下所示：

```
//连接到数据库
$c = new MongoClient();
//选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
//使用点操作符搜索地址为“Newton”的文档
$address = array("Address.Place" => "Newton");
//执行查询并将结果保存在变量$cursor中
$cursor = $collection->find($address);
//遍历它在集合中找的文档，并输出它们的 ID 和内容
while($document = $cursor->getNext())
{
    print_r($document);
}
```

可以通过在数组中指定其中一个数据项，以相似的方式搜索文档数组中的信息，例如电子邮件地址。因为电子邮件地址通常是唯一的，所以本例中使用 findOne() 函数即可满足需求：

```
//连接到数据库
$c = new MongoClient();
//选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
//在变量$email中定义希望搜索的电子邮件地址
$email = array("E-Mail" => "vw@example.com");
//查找集合中匹配电子邮件地址的第一个人
print_r($collection->findOne($email));
```

和预期效果一样，该例将返回匹配电子邮件地址 vw@example.com 的第一个文档，在本例中 Victoria Wood 的电子邮件地址就符合要求。该文档将以数组的形式返回：

```
Array (
    [_id] => MongoClient Object ( )
    [First Name] => Victoria
    [Last Name] => Wood
    [Address] => Array (
        [Street] => 50 Ash lane
        [Place] => Ystradgynlais
        [Postal Code] => SA9 6XS
        [Country] => UK
    )
    [E-Mail] => Array (
        [0] => vw@example.com
        [1] => vw@office.com
    )
)
```

```

    )
    [Phone] => 078-8727-8049
    [Age] => 28
)

```

6.5.5 排序、限制和忽略数据项

MongoCursor 类提供了 `sort()`、`limit()` 和 `skip()` 函数，分别用于对结果进行排序、限制返回结果的总数和忽略特定数目的结果。下面我们将开始在 PHP 中练习这些函数并学习它们的用法。

PHP 的 `sort()` 函数接受一个数组作为参数。在该数组中，可以指定用于排序文档的字段。与在 shell 中排序一样，使用 1 按升序对结果排序，-1 按降序对结果排序。注意，需要在一个已存在的游标(也就是之前执行 `find()` 命令时返回的结果)上执行这些函数。

下面的样例将基于联系人的年龄对联系人进行升序排列：

```

//连接到数据库
$c = new MongoClient();
//选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
//执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find();
//基于年龄使用排序命令对$cursor 中的所有结果进行排序
$cursor->sort(array('Age' => 1));
//输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}

```

下面我们将在真实的游标上执行 `limit()` 函数；它将接受一个参数，用于指定希望返回结果的数目。命令 `limit()` 将返回集合中匹配搜索条件的前 `n` 个数据项。下面的样例将只返回一个文档(当然，也可以使用 `findOne()` 函数，但通过 `limit()` 函数来实现更合理)：

```

//连接到数据库
$c = new MongoClient();
//选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
//执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find();
//使用限制函数将结果数目限制为 1
$cursor->limit(1);
//输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}

```

最后，使用 `skip()` 函数忽略前 `n` 个匹配搜索条件的结果。该函数也基于游标工作：

```

// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find();

```

```
// 使用忽略函数忽略第一个结果
$cursor->skip(1);
// 输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}
```

6.5.6 统计匹配结果的数目

使用 PHP 的 `count()` 函数可以统计匹配搜索条件的文档数目，并返回数组中元素的数目。该函数是 `MongoCursor` 类的一部分，因此它也基于游标工作。下面的样例显示了如何得到集合中所有居住在美国的联系人总数：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定搜索参数
$country = array("Address.Country" => "USA");
// 执行查询并将结果保存在变量$cursor 中，以便执行进一步处理
$cursor = $collection->find($country);
// 统计结果并返回
print_r($cursor->count());
```

该查询将只返回一个结果。该技术对于所有操作都是有用的，无论是统计评论、注册用户总数还是其他信息。

6.5.7 使用聚集框架对数组分组

聚集框架是 MongoDB 内嵌的强大特性之一，通过它可以计算聚集数值，而不需要使用——通常过于复杂——映射/规约功能。该框架中包含的最有用的一个管道操作符是 `$group` 操作符，它大致等同于 SQL 中的 `GROUP BY` 子句。通过该操作符可以基于文档的集合计算聚集值。例如，聚集函数 `$max` 可用于搜索和返回组中的最大值，`$min` 函数可用于搜索和返回最小值，`$sum` 函数可用于计算指定值出现的次数。

如果希望得到集合中所有联系人的列表，并按照居住的国家分组，使用聚集框架可以轻松实现，如下所示：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 执行查询并将结果保存在$result 中
$result = $collection->aggregate(array(
    '$group' => array(
        '_id' => '$Address.Country',
        'total' => array('$sum' => 1)
    )
));
// 统计结果并返回
print_r($result);
```

如你所见，聚集函数接受一个(或多个)含有管道操作符(在本例中为 `$group`)的数组。在这里，

可以指定如何返回结果输出和希望执行的可选聚集函数：本例中为\$sum 函数。本例将为每个已找到的国家(唯一的)返回唯一的文档，由文档的_id 字段表示。接下来，国家的总数将使用\$sum 函数进行统计，并使用 total 字段返回。注意，\$sum 函数由数组表示，指定的数值 1 表示为每个匹配的文档将总数加 1。

你可能会好奇结果输出的样子。下面是一个样例，显示出目前有两人居住在英国，有一人居住在美国：

```
Array (
  [result] => Array (
    [0] => Array (
      [_id] => UK [total] => 2
    )
    [1] => Array (
      [_id] => USA [total] => 1
    )
  )
  [ok] => 1
)
```

该例比较简单，但聚集框架非常强大，第 8 章将会详细讲解它的强大之处。

6.5.8 使用 hint()函数指定索引

使用 PHP 的 hint()函数可以指定查询数据时使用的索引，这样做可以帮助提高查询性能。假设集合中有数千个联系人，并且通常需要基于名字搜索联系人。对于本例，推荐基于集合中的 Last Name 键创建索引。

注意：

对于接下来展示的 hint()样例，如果未创建索引，将什么也不会返回。

只能在游标上使用 hint()函数，如下所示：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find(array("Last Name" => "Moran"));
// 使用 hint()函数指定目标索引
$cursor->hint(array("Last Name" => -1));
// 输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}
```

注意：

关于如何创建索引的更多细节请参阅第 4 章。正如之前讨论的，也可以使用 PHP 驱动的 ensureIndex()函数创建索引。

6.5.9 使用条件操作符重新定义查询

可以使用条件操作符重新定义查询。PHP 使用一组默认的条件操作符，例如 <(小于)、>(大于)、<=(小于或等于)和 >=(大于或等于)。坏消息是，在 PHP 驱动中不能使用这些条件操作符。相反，需要使用这些操作符的 MongoDB 版本。幸运的是，MongoDB 自己提供了大量的条件操作符(在第 4 章可以找到更多信息)。在通过 PHP 查询数据时可以使用所有这些操作符，将它们传入到 find()函数中即可。

在 PHP 驱动中使用所有这些参数时必须遵循特定的语法：将它们添加到一个数组中，并将该数组传入 find()函数。下面将学习如何使用几种常用的操作符。

1. 使用 \$lt、\$gt、\$lte 和 \$gte 操作符

MongoDB 的 \$lt、\$gt、\$lte 和 \$gte 操作符分别等同于 <、>、<=和 >=。这些操作符在搜索存储整数的文档时非常有用。

可以使用 \$lt(小于)操作符找到任何整数值小于 n 的数据，如下所示：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定条件操作符
$cond = array('Age' => array('$lt' => 30));
// 执行查询并将结果保存在变量 $cursor 中
$cursor = $collection->find($cond);
// 输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}
```

结果输出中只显示了一条信息：Victoria Wood 的联系信息，碰巧他的年龄小于 30 岁。

```
Array (
    [_id] => MongoId Object ( )
    [First Name] => Victoria
    [Last Name] => Wood
    [Address] => Array (
        [Street] => 50 Ash lane
        [Place] => Ystradgynlais
        [Postal Code] => SA9 6XS
        [Country] => UK
    )
    [E-Mail] => Array (
        [0] => vw@example.com
        [1] => vw@office.com
    )
    [Phone] => 078-8727-8049
    [Age] => 28
)
```

类似地，可以使用 \$gt 操作符找到所有年龄大于 30 岁的联系人。下面的样例将 \$lt 操作符更换成了 \$gt(大于)：

```

// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定条件操作符
$cond = array('Age' => array('$gt' => 30));
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find($cond);
// 输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}

```

该代码将返回 Philip Moran 的联系信息，因为他的年龄大于 30 岁：

```

Array (
    [_id] => MongoId Object ( )
    [First Name] => Philip
    [Last Name] => Moran
    [Address] => Array (
        [Street] => 681 Hinkle Lake Road
        [Place] => Newton
        [Postal Code] => MA 02160
        [Country] => USA
    )
    [E-Mail] => Array (
        [0] => pm@example.com
        [1] => pm@office.com
        [2] => philip@example.com
        [3] => philip@office.com
        [4] => moran@example.com
        [5] => moran@office.com
        [6] => pmoran@example.com
        [7] => pmoran@office.com
    )
    [Phone] => 617-546-8428
    [Age] => 60
)

```

可以使用 `$lte` 操作符指定小于等于的目标数值。记住：`$lt` 将找到所有小于 30 岁的联系人，但不可以是 30 岁。同样，`$gte` 操作符可以找到所有大于或等于指定年龄的联系人。下面将演示两个样例。

第一个样例将在界面上显示出集合中的所有文档：

```

// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定条件操作符
$cond = array('Age' => array('$lte' => 60));
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find($cond);
// 输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}

```

第二个样例只会显示出一条信息，因为该集合只有一名年龄大于等于 60 岁的联系人：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定条件操作符
$cond = array('Age' => array('$gte' => 60));
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find($cond);
// 输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}
```

2. 搜索不匹配的文档

可以使用\$ne(不等于)操作符搜索不等于指定值的所有文档。该操作符的语法非常直观。下面的样例显示了任何年龄等于 28 岁的联系人：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定条件操作符
$cond = array('Age' => array('$ne' => 28));
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find($cond);
// 输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}
```

3. 使用\$in 匹配多个值中的任意一个

通过操作符\$in 可以搜索匹配数组中任何一个值的文档，如下所示：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定条件操作符
$cond = array('Address.Country' => array('$in' => array("USA","UK")));
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find($cond);
// 输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}
```

结果输出中将显示出任何已添加联系人的信息，无论该联系人是居住在美国还是英国。注意，所有的目标值都被添加到了一个数组中；不能输入“就是这样”的信息。

4. 在查询中使用\$all 匹配所有条件

同\$in 操作符一样，通过\$all 可以在一个额外的数组中匹配多个值。它们的不同点在于，\$all 操作符要求在返回结果之前，其中的文档必须能够匹配数组中的所有元素。下面的样例展示了如何执行这样的查询：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定条件操作符
$con = array('E-Mail' => array('$all' => array("vw@example.com", "vw@office.com")));
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find($con);
//输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}
```

5. 使用\$or 搜索多个表达式

可以使用\$or 操作符指定文档可以包含多个表达式。\$in 和\$or 操作符之间的区别在于，\$in 操作符不允许同时指定键和值，而\$or 可以。可以将\$or 操作符与任何键/值对结合使用。下面是关于它的两个例子。

第一个样例将搜索并返回任何包含了 Age 键为 28 或 Address.Country 键为 USA 的文档：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定条件操作符
$con = array('$or' => array(
    array("Age" => 28),
    array("Address.Country" => "USA")
));
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find($con);
// 输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}
```

第二个样例将搜索并返回任何 Address.Country 键为 USA(必须满足)，并且包含键值对"Last Name":"Moran"或"E-Mail": "vw@example.com"的文档：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定条件操作符
$con = array(
    "Address.Country" => "USA",
    '$or' =>array(
```

```
array("Last Name" => "Moran"),
array("E-Mail" => "vw@example.com")
);
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find($cond);
// 输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}
```

通过\$or 操作符可以同时执行两个搜索，并将它们的结果输出相结合，哪怕这两个搜索并没有相同的地方。

6. 使用\$slice 获取指定数目的文档

可以使用\$slice 投影操作符从文档数组中获取指定数目的文档。该函数与本章之前详细讲解的 skip()和 limit()函数相似。它们的不同是，skip()和 limit()函数作用于整个文档，而\$slice 投影操作符将基于数组工作，而不是单个文档。

该操作符是按页显示数据的好方法(通常被称为分页)。下一个样例将展示如何限制之前指定的联系人(Philip Moran)的电子邮件地址数目；在本例中，将只返回他的前三个电子邮件地址。

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
//指定搜索操作符
$query = array("Last Name" => "Moran");
// 使用$slice 操作符从数组中创建新的对象
$cond = (object)array('E-Mail' => array('$slice' => 3));
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find($query, $cond);
// 遍历集合中找到的每个文档，并输出它们的内容
while ($document = $cursor->getNext())
{
    print_r($document);
}
```

类似地，通过将操作符的值设置为负数可以只返回最后三个电子邮件地址，如下所示：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定搜索操作符
$query = array("Last Name" => "Moran");
// 指定条件操作符
$cond = (object)array('E-Mail' => array('$slice' => -3));
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find($query, $cond);
// 遍历集合中找到的每个文档，并输出它们的内容
while($document = $cursor->getNext())
{
    print_r($document);
}
```

也可以忽略前两个元素，并将结果数目限制为 3：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定搜索操作符
$query = array("Last Name" => "Moran");
// 指定条件操作符
$cond = (object)array('E-Mail' => array('$slice' => array(2, 3)));
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find($query, $cond);
// 遍历集合中找到的每个文档，并输出它们的内容
while($document = $cursor->getNext())
{
    print_r($document);
}
```

操作符\$slice 是限制数组中元素数目的好方法；在使用 MongoDB 驱动和 PHP 进行编程时，一定要记住该操作符。

7. 判断某个字段是否有值

可以使用\$exists 操作符根据一个字段是否有值来返回结果(无论该字段的值是什么)。这听起来似乎不合逻辑，但它实际上非常方便。例如，可以搜索尚未设置 Age 字段的联系人；或者可以搜索已经设置了街道名称的联系人。

下面的样例将返回任何未设置 Age 字段的联系人：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定条件操作符
$cond = array('Age' => array('$exists' => false));
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find($cond);
// 输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}
```

类似地，下一个样例将返回任何已经设置了 Street 字段的联系人：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定条件操作符
$cond = array("Address.Street" => array('$exists' => true));
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find($cond);
// 输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}
```

6.5.10 正则表达式

正则表达式非常简洁。可以使用它们完成任何事情(可能除了做咖啡);在搜索数据时,它们可以极大地简化这个过程。PHP 驱动为正则表达式提供了自己的类: `MongoRegex` 类。通过使用该类可以创建正则表达式,然后使用它们搜索数据。

`MongoRegex` 类支持正则表达式的如下 6 个标志,它们可用于查询数据:

- `i`: 表示忽略大小写。
- `m`: 搜索将跨越多行(换行)内容。
- `x`: 允许搜索包含#注释的内容。
- `l`: 指定区域设置。
- `s`: 也称为 `dotall`, “.” 可用于匹配一切,包含换行。
- `u`: 匹配 Unicode 字符。

现在我们将详细讲解如何在 PHP 中使用正则表达式搜索集合中的数据。很明显,最好的方式就是通过简单的样例展示它们的用法。

例如,现在希望搜索一个只知道一点信息的联系人。你可能只记得这个联系人居住的地方,并且地址中包含了一些类似于 `stradgynl` 的字符串。正则表达式提供了实现该搜索的一种简单优雅的方式:

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
//指定正则表达式
$regex = new MongoRegex("/stradgynl/i");
// 执行查询并将结果保存在变量$cursor 中
$cursor = $collection->find(array("Address.Place" => $regex));
// 输出结果
while($document = $cursor->getNext())
{
    print_r($document);
}
```

在创建 PHP 应用时,通常希望搜索特定的数据。在之前的样例中,可以使用 `$_POST` 变量替换目标文本(在本例中为“`stradgynl`”)。

6.6 使用 PHP 修改数据

如果我们处在一个所有数据都保持不变,并且人们永远也不犯错的世界,那么永远也不需要更新文档。但真实的世界要复杂一些,有时我们会犯错,并且也希望能够更正错误。

对于这样的情况,可以使用 MongoDB 中的一组修改函数更新已有数据。有几种方式可完成更新。例如,可以使用 `update()` 函数更新已有信息,然后使用 `save()` 函数保存修改。接下来将讲解其中的一些函数以及修改操作符,并举例演示如何高效地使用它们。

6.6.1 使用 `update()` 函数更新数据

如第 4 章讲解的,可以使用 `update()` 函数执行大多数的文档更新操作。与 MongoDB shell

的 `update()` 版本一样，PHP 驱动中的 `update()` 函数也支持使用修改操作符帮助快速、轻松地更新文档。PHP 版本的 `update()` 函数操作几乎与 MongoDB shell 版本一致；不过，PHP 版本的使用方式大大不同。接下来将讲解如何在 PHP 中使用该函数。

PHP 版本的 `update()` 函数最少接受两个参数：第一个参数用于指定更新的目标对象；第二个参数指定用于更新匹配记录的对象。另外，可以指定第三个参数作为扩展选项组。

`options` 参数提供了 7 个可用于 `update()` 函数的额外标志；下面的列表展示了它们的定义和用法：

- **upsert**: 如果该布尔选项被设置为真，并且搜索条未被匹配，那就创建一个新的文档。
- **multiple**: 如果该布尔选项被设置为真，那么匹配该搜索条件的所有文档都将被更新。
- **fsync**: 如果该布尔选项被设置为真，那么数据将在更新结果返回前同步到磁盘。如果该选项被设置为真，那么即使选项 `w` 已经设置了其他值，也会把选项 `w` 设置为 0。
- **w**: 如果设置为 0，更新操作将不会得到确认。在使用复制集时，可以将它设置为 `n`，保证主服务器在将修改复制到 `n` 个节点后才确认该更新操作。还可以设置为 'majority'——一个保留字符串——用于保证大多数复制节点或特定标签中的节点都确认该更新操作。该选项默认值为 1，表示确认更新操作。
- **j**: 如果该布尔选项被设置为真，那么数据将在更新结果返回之前写入到日志中。默认值为假。
- **wtimeout**: 用于指定服务器等待接收确认的超时时间(以毫秒为单位)。默认为 10 000。
- **timeout**: 用于指定客户端需要等待数据库返回的时间(以毫秒为单位)。

下面的例子将在不使用任何修改操作符的情况下，将 Victoria Wood 的姓修改为 “Vicky”：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库 'contacts' 的 'people' 集合
$collection = $c->contacts->people;
// 指定搜索条件
$criteria = array("Last Name" => "Wood");
// 指定将被修改的信息
$update = array(
    "First Name" => "Vicky",
    "Last Name" => "Wood",
    "Address" => array(
        "Street" => "50 Ash lane",
        "Place" => "Ystradgynlais",
        "Postal Code" => "SA9 6XS",
        "Country" => "UK"
    )
),
    "E-Mail" => array(
        "vw@example.com",
        "vw@office.com"
    ),
    "Phone" => "078-8727-8049",
    "Age" => 28
);
// 选项
$options = array("upsert" => true);
// 执行更新
$collection->update($criteria,$update,$options);
// 显示结果
print_r($collection->findOne($criteria));
```


结果输出将如下所示:

```
Array (
  [_id] => MongoId Object ()
  [First Name] => Vicky
  [Last Name] => Wood
  [Address] => Array (
    [Street] => 50 Ash lane
    [Place] => Ystradgynlais
    [Postal Code] => SA9 6XS
    [Country] => UK
  )
  [E-Mail] => Array (
    [0] => vw@example.com
    [1] => vw@office.com
  )
  [Phone] => 078-8727-8049
  [Age] => 28
)
```

本例中只修改了一个字段,但却需要用到许多代码,这并不是我们所希望的。不过,这是在不使用 PHP 修改操作符的情况下所必须做的。现在开始学习在 PHP 中如何使用操作符来帮助快速和轻松地更新文档。

警告:

在修改文档时如果不指定任何额外的操作符,匹配文档中的数据将被数组中的信息替代。通常,如果只希望更新一个字段,最好使用 \$set。

6.6.2 节省更新操作的时间

更新操作可以节省输入信息所需的工作量。你可能会同意之前的样例并不可行。幸运的是,PHP 驱动包含了大约 6 个更新操作符可用于快速修改数据,通过它们可以不用完全输入所有数据。在这里,我们将再次总结每个操作符的目的,尽管此时你可能已经熟悉其中的大部分操作符(更多关于更新操作的细节信息请查看第 4 章的内容)。不过,在 PHP 中使用它们的方式将大大不同,与更新相关选项的使用方式也变得不同。下面将通过其中一些操作符的样例来帮助你熟悉它们在 PHP 中的语法。

注意:

接下来的更新操作符样例中都不包含用于检阅修改的 PHP 代码;相反,这些例子都只是应用了对应的修改。因此建议在使用 PHP 代码的同时启动 MongoDB shell,这样就可以执行搜索并确认 MongoDB 已经应用了预期的修改。另外,也可以编写额外的 PHP 代码来执行这些检查。

6.6.3 使用 \$inc 增加特定键的值

在键已存在的情况下,通过 \$inc 操作符可以将该键的值增加 n。如果键不存在,它将会被创建。下面的样例将把所有小于 40 岁的人的年龄增加 3 岁:

```
// 连接到数据库
$c = new MongoClient();
```

```

// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 搜索所有年龄小于 40 岁的人
$criteria = array("Age" => array('$lt' => 40));
// 使用$inc 将他们的年龄增加 3 岁
$update = array('$inc' => array('Age' => 3));
// 选项
$options = array("upsert" => true);
// 执行更新
$collection->update($criteria,$update,$options);

```

6.6.4 使用\$set 修改键值

通过\$set 可以修改某个键的值，而忽略其他任何字段。和之前描述的一样，这种方式将比之前更新 Victoria 姓名的那个样例简单得多。下面的样例将展示如何使用\$set 操作将联系人的名字修改为“Vicky”：

```

// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定搜索条件
$criteria = array("Last Name" => "Wood");
// 指定将被修改的信息
$update = array('$set' => array("First Name" => "Vicky"));
// 选项
$options = array("upsert" => true);
// 执行更新
$collection->update($criteria,$update,$options);

```

还可以使用\$set 为所有匹配查询的文档添加一个字段：

```

// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 使用正则表达式指定搜索条件
$criteria = array("E-Mail" => new MongoRegex("/@office.com/i"));
// 在所有找到的文档中添加“Category->Work”
$update = array('$set' => array('Category' => 'Work'));
// 选项
$options = array('upsert' => true, 'multi' => true);
// 通过 save() 执行更新/插入操作
$collection->update($criteria,$update,$options);

```

6.6.5 使用\$unset 删除字段

\$unset 的工作方式与\$set 类似。不同之处在于，\$unset 可以从文档中删除指定的字段。例如，下面的样例将从 Victoria Wood 的联系人信息中删除 Phone 字段以及与之相关联的数据：

```

// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定搜索条件

```

```
$criteria = array("Last Name" => "Wood");  
// 指定将被移除的信息  
$update = array('$unset' => array("Phone" => 1));  
// 执行更新  
$collection->update($criteria,$update);
```

6.6.6 使用\$rename 重命名字段

\$rename 操作符可用于重命名字段。如果不小心输入错误或者只是希望使用一个更准确的名字，该操作符非常有用。该操作符将在每个文档及其数组和子文档中搜索指定的字段名。

警告：

使用该操作符时要小心。如果文档中已经包含使用指定名称的字段，那么该字段将会被删除，然后将旧的字段名修改为指定的字段名。

下面的样例将分别把 Vicky Wood 联系信息中的 First Name 和 Last Name 字段重命名为 Given Name 和 Family Name:

```
// 连接到数据库  
$c = new MongoClient();  
// 选择数据库'contacts'的'people'集合  
$collection = $c->contacts->people;  
// 指定搜索条件  
$criteria = array("Last Name" => "Wood");  
// 指定将被修改的信息  
$update = array('$rename' => array("First Name" => "Given Name", "Last Name" => "Family Name"));  
// 执行更新  
$collection->update($criteria,$update);
```

6.6.7 在更新/插入期间使用\$setOnInsert 修改键值

当使用\$upsert 操作符执行插入操作时，可使用 PHP 的\$setOnInsert 操作符为某个键赋予特定的值。这听起来可能有点混乱，但你可以将这个操作符看作条件操作符，只在\$upsert 执行插入时才设置特定的值，而不是更新某个值。下面的样例将演示如何使用它。首先，在一个现有文档的基础上执行\$upsert 操作，忽略指定的\$setOnInsert 操作符：

```
// 连接到数据库  
$c = new MongoClient();  
// 选择数据库'contacts'的'people'集合  
$collection = $c->contacts->people;  
// 指定搜索条件  
$criteria = array("Family Name" => "Wood");  
// 指定在插入操作执行时设置的信息  
$update = array('$setOnInsert' => array("Country" => "Unknown"));  
// 执行更新/插入选项  
$options = array("upsert" => true);  
// 执行更新  
$collection->update($criteria,$update,$options);
```

接下来的例子将使用\$upsert 插入一个尚不存在的文档。此时\$setOnInsert 操作符将会成功作用到结果中：

```

// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定搜索条件
$criteria = array("Family Name" => "Wallace");
// 指定在插入操作执行时设置的信息
$update = array('$setOnInsert' => array("Country" => "Unknown"));
// 执行更新/插入选项
$options = array("upsert" => true);
// 执行更新
$collection->update($criteria,$update,$options);

```

此段代码将搜索任何 Family Name 字段被设置为“Wallace”的文档。如果未找到，就执行一个插入操作，被插入文档中的 Country 字段也将被设置为“Unknown”，创建出的空文档如下所示：

```

{
  "_id" : ObjectId("1"),
  "Country" : "Unknown",
  "Last Name" : "Wallace"
}

```

6.6.8 使用\$push 向指定字段中添加值

通过 PHP 的\$push 操作符可以向指定的字段中添加值。如果该字段是一个已有的数组，那么数据将被添加到其中；如果该字段不存在，那么该操作符将会创建该字段。如果该字段存在，但不是数组，那将会抛出错误消息。下面的样例演示了如何使用\$push 向已有数组中添加一些数据：

```

// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定搜索条件
$criteria = array("Family Name" => "Wood");
// 指定将要添加的信息
$update = array('$push' => array("E-Mail" => "vw@mongo.db"));
// 执行更新
$collection->update($criteria,$update);

```

6.6.9 使用\$push 和\$each 向某个键中添加多个值

通过\$push 操作符可以向某个键中添加多个值。为此，同时也需要使用\$each 修改操作符。如果数组中的值在指定字段中尚不存在，它们将被添加到该字段中。如同单独使用\$push 一样，规则是相同的：如果字段存在并且是数组，那么数据将被添加到数组中；如果不存在，该字段将被创建；如果字段存在但不是数组，那将抛出错误消息。下面的样例演示了如何使用\$each 修改操作符：

```

// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;

```

```
// 指定搜索条件
$criteria = array("Family Name" => "Wood");
// 指定将要添加的信息
$update = array(
    '$push' => array(
        "E-Mail" => array(
            '$each' => array(
                "vicwo@mongo.db",
                "vicwo@example.com"
            )
        )
    )
);
// 执行更新
collection->update($criteria,$update);
```

6.6.10 使用\$addToSet 将数据添加到数组中

操作符\$addToSet 类似于\$push 操作符，一个重要的区别在于：**\$addToSet** 只在目标数组不包含该数据时，才将数据添加到数组中。**\$addToSet** 操作符接受一个数组作为参数：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定搜索条件
$criteria = array("Family Name" => "Wood");
// 指定将要添加的信息(执行成功，因为该数据尚不存在)
$update = array('$addToSet' => array("E-Mail" => "vic@example.com"));
// 执行更新
collection->update($criteria,$update);
```

类似地，结合使用\$addToSet 和\$each 操作符可以将多个尚不存在的数据添加到数组中：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定搜索条件
$criteria = array("Family Name" => "Wood");
// 指定将要添加的信息(只有部分数据添加成功，因为其中一些数据已经存在)
$update = array(
    '$addToSet' => array
    (
        "E-Mail" => array
        (
            '$each' => array
            (
                "vw@mongo.db",
                "vicky@mongo.db",
                "vicky@example.com"
            )
        )
    )
);
// 执行更新
collection->update($criteria,$update);
```

6.6.11 使用\$pop 从数组中删除元素

通过 PHP 的\$pop 操作符可以从数组中删除元素。记住，只可以删除数组中的第一个或最后一个元素——中间部分不变。通过将值指定为-1 可以删除第一个元素；类似地，将值指定为 1 可以删除最后一个元素：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定搜索条件
$criteria = array("Family Name" => "Wood");
// 删除列表中的第一个电子邮件地址
$update = array('$pop' => array("E-Mail" => -1));
// 执行更新
$collection->update($criteria,$update);
```

注意：

将值指定为-2 或 1000 并不会改变被删除的元素。使用任何负值都将删除第一个元素，而使用任何正值都将删除最后一个元素。使用 0 也将从数组中删除最后一个元素。

6.6.12 使用\$pull 删除所有指定值

通过 PHP 的\$pull 操作符可以从数组中删除所有指定值。例如，如果在使用\$push 或\$pushAll 时不小心在数组中添加了重复值，那么使用该操作符就可以非常方便地删除这些重复值。下面的样例将删除一个电子邮件地址的所有重复值：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定搜索条件
$criteria = array("Family Name" => "Wood");
// 删除重复的所有电子邮件地址“Vicky@example.com”
$update = array('$pull' => array("E-Mail" => "vicky@example.com"));
// 执行更新
$collection->update($criteria,$update);
```

6.6.13 同时删除多个元素

类似地，可以使用\$pullAll 操作符从文档删除多个元素的所有值，如下所示：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定搜索条件
$criteria = array("Family Name" => "Wood");
// 删除下面所有的电子邮件地址
$update = array(
    '$pullAll' =>array(
        "E-Mail" =>array("vw@mongo.db","vw@office.com")
    )
);
```

```
);  
// 执行更新  
$collection->update($criteria,$update);
```

6.6.14 使用 save()函数更新/插入数据

同 insert()函数一样, save()函数也可以在集合中插入数据。唯一的区别是,使用 save()也可以更新已有的字段。该操作被称作更新/插入。此时执行 save()函数的方式就不显得奇怪了。同 MongoDB shell 中的 save()函数一样, PHP 的 save()函数将接受两个参数:包含了希望保存的信息的数组,以及作用于保存的选项。以下是可用选项:

- **fsync**: 如果该布尔选项被设置为真,那么数据将在更新结果返回前同步到磁盘。如果该选项被设置为真,那么即使选项 w 已经设置了其他值,也会把选项 w 设置为 0。
- **w**: 如果设置为 0,更新操作将不会得到确认。在使用复制集时,可以将它设置为 n,保证主服务器在将修改复制到 n 个节点后才确认该更新操作。还可以设置为'majority'——一个字符串——用于保证大多数复制节点或者特定标签中的节点都确认该更新操作。该选项默认值为 1,表示确认更新操作。
- **j**: 如果该布尔选项被设置为真,那么数据将在更新结果返回之前写入到日志中。默认值为假。
- **wtimeout**: 用于指定服务器等待接收确认的超时时间(以毫秒为单位)。默认为 10 000。
- **timeout**: 用于指定客户端需要等待数据库返回的时间(以毫秒为单位)。

PHP 版本的 save()函数的语法也类似于 MongoDB shell 版本,如下所示:

```
//指定将要保存的文档  
$contact = array(  
    "Given Name" => "Kenji",  
    "Family Name" => "Kitahara",  
    "Address" =>array(  
        "Street" => "149 Bartlett Avenue",  
        "Place" => "Southfield",  
        "Postal Code" => "MI 48075",  
        "Country" => "USA"  
    )  
    ,  
    "E-Mail" =>array(  
        "kk@example.com",  
        "kk@office.com"  
    )  
    ,  
    "Phone" => "248-510-1562",  
    "Age" => 34  
);  
  
// 连接到数据库  
$c = new MongoClient();  
// 选择'people'集合  
$collection = $c->contacts->people;  
// 通过 save() 函数保存数据  
$options = array("fsync" => true);  
// 指定 save()选项  
$collection->save($contact,$options);  
  
// 意识到丢了什么东西,因此使用更新/插入操作更新联系人  
$contact['Category'] = 'Work';
```

```
// 执行更新插入操作
$collection->save($contact);
```

6.6.15 以原子的方式修改文档

同 `save()` 和 `update()` 函数一样，在 PHP 驱动中也可以调用 `findAndModify()` 函数。记住，通过 `findAndModify()` 函数能够以原子的方式修改文档，并在更新成功之后返回结果。它将更新单个文档，并且默认返回的是修改之前的文档。只有在指定额外的参数 `new` 时，它才会返回修改后的文档。

函数 `findAndModify()` 将接受 4 个参数：`query`、`update`、`fields` 和 `options`。其中一些是可选的，取决于目标操作。例如，在指定更新条件时，`fields` 和 `options` 参数都是可选的。不过，如果希望使用删除选项，那就需要指定 `update` 和 `fields` 参数(例如使用 `null`)。下面的列表将详细讲解这些可用的参数：

- **query**: 指定查询使用的过滤器。如果未指定该参数，那么集合中所有的文档都将被认为是目标，并且搜索到的第一个文档将被更新或删除。
- **update**: 指定用于更新文档的信息。注意，之前指定的任何修改操作符都可作用于该参数。
- **fields**: 指定希望返回的字段，而不是整个文档。该参数的行为与 `find()` 函数中的 `fields` 参数一致。注意：`_id` 字段总是会被返回，即使该字段并未在字段列表中指定。
- **options**: 指定选项。以下是可用选项：
 - **sort**: 以特定的顺序对匹配文档进行排序。
 - **remove**: 如果设置为真，那么第一个匹配文档将被删除。
 - **update**: 如果设置为真，那么将在被选择的文档上执行更新操作。
 - **new**: 如果设置为真，更新后的文档将被返回，而不是返回被选择的文档。注意，该参数默认未被设置，在某些环境中可能会引起混淆。
 - **upsert**: 如果设置为真，那么函数 `findAndModify()` 将执行更新/插入操作。

下面的样例将展示如何使用这些参数。第一个样例将搜索所有姓为“Kitahara”的联系人，并结合使用 `update()` 和 `$push` 操作符将电子邮件地址添加到他的联系信息中。下面的样例未设置 `new` 参数，所以结果输出将仍然显示原始文档：

```
// 连接到数据库
$c = new MongoClient();
// 指定目标数据库和集合
$collection = $c->contacts->people;
// 指定搜索条件
$criteria = array("Family Name" => "Kitahara");
// 指定更新条件
$update = array('$push' => array("E-Mail" => "kitahara@mongo.db"));
// 执行 findAndModify() 操作
$collection->findAndModify($criteria,$update);
```

返回的文档将如下所示：

```
Array (
    [value] => Array (
        [Given Name] => Kenji
        [Family Name] => Kitahara
        [Address] => Array (
```



```
[Street] => 149 Bartlett Avenue
[Place] => Southfield
[Postal Code] => MI 48075
[Country] => USA
)
[E-Mail] => Array (
    [0] => kk@example.com
    [1] => kk@office.com
)
[Phone] => 248-510-1562
[Age] => 34
[_id] => MongoId Object ( )
[Category] => Work
)
[ok] => 1
)
```

下面的样例演示了如何使用 `remove` 和 `sort` 参数:

```
// 连接到数据库
$c = new MongoClient();
// 指定目标数据库和集合
$collection = $c->contacts->people;
// 指定搜索条件
$criteria = array("Category" => "Work");
// 指定选项
$options = array("sort" => array("Age" => -1), "remove" => true);
// 执行 findAndModify() 操作
$collection->findAndModify($criteria,null,null,$options);
```

6.7 删除数据

之前的样例曾演示在 MongoDB shell 中如何使用 `remove()` 函数删除文档, 这里同样也可以做到。PHP 驱动中也包含了用于删除数据的 `remove()` 函数。该函数的 PHP 版本将接受两个参数: 一个包含了希望删除的记录描述; 另一个包含了作用于删除过程的额外选项。

以下是 7 个可用选项:

- **justOne**: 如果设置为真, 最多只有一个匹配条件的文档将被删除。
- **fsync**: 如果该布尔选项被设置为真, 那么数据将在更新结果返回前同步到磁盘。如果该选项被设置为真, 那么即使选项 `w` 已经设置了其他值, 也会把选项 `w` 设置为 0。
- **w**: 如果设置为 0, 更新操作将不会得到确认。在使用复制集时, 可以将它设置为 `n`, 保证主服务器在将修改复制到 `n` 个节点后才确认该更新操作。还可以设置为 `'majority'`——一个保留字符串——用于保证大多数复制节点或特定标签中的节点都确认该更新操作。该选项默认值为 1, 表示确认更新操作。
- **j**: 如果该布尔选项被设置为真, 那么数据将在更新结果返回之前写入到日志中。默认值为假。
- **wtimeout**: 用于指定服务器等待接收确认的超时时间(以毫秒为单位)。默认为 10 000。
- **timeout**: 用于指定客户端需要等待数据库返回的时间(以毫秒为单位)。

下面是一些演示如何删除文档的样例代码：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 指定搜索条件
$criteria = array("Family Name" => "Wallace");
// 指定选项
$options = array('justOne' => true, 'w' => 0);
// 执行删除操作
$collection->remove($criteria,$options);
```

类似地，下面的样例将同时删除多个文档：

```
// 连接到数据库
$c = new Mongo();
// 选择数据库'contacts'的'people'集合
$collection = $c->contacts->people;
// 使用正则表达式指定搜索条件
$criteria = array("E-Mail" => new MongoRegex("/@office.com/i"));
// 指定选项
$options = array("justOne" => false);
// 执行删除操作
$collection->remove($criteria,$options);
```

警告：

在删除文档时，要记住文档的引用仍然会保留在数据库中。确保手动删除或更新了这些被删除文档的引用；否则，这些引用在执行时将返回 `null`。

类似地，可以使用 `drop()` 函数删除整个集合。下面的样例将返回含有删除结果信息的数组：

```
// 连接到数据库
$c = new Mongo();
// 选择将被删除的集合
$collection = $c->contacts->people;
// 删除集合并返回结果
print_r($collection->drop());
```

执行结果将如下所示：

```
Array (
    [nIndexesWas] => 1
    [msg] => indexes dropped for collection
    [ns] => contacts.people
    [ok] => 1
)
```

最后但不是最不重要的是，可以使用 PHP 删除整个数据库。通过在 `MongoDB` 类中调用 `drop()` 函数可完成该操作，如下所示：

```
// 连接到数据库
$c = new Mongo();
// 选择将要删除的数据库
$db = $c->contacts;
// 删除数据库并返回结果
print_r($db->drop());
```

结果将显示出被删除数据库的名字:

```
Array (  
  [dropped] =>contacts  
  [ok] => 1  
)
```

6.8 DBRef

通过 DBRef 可以创建出两个文档(存储在不同位置)之间的链接, 通过该功能可以实现与关系数据库类似的行为。如果希望将人们的地址保存在 `addresses` 集合中, 而不是将信息包含在 `people` 集合中, 该功能会非常方便。

实现该需求有两种方式。首先, 可以使用一个简单的链接(被称为手动引用); 在这种情况下, 需要在一个文档中包含另一个文档的 `_id` 字段。其次, 可以使用 DBRef 自动创建这样的链接。

首先我们将演示如何实现手动引用。在下面的样例中添加一个联系人, 并在他的地址信息中指定另一个文档的 `_id`:

```
// 连接到数据库  
$c = new MongoClient();  
$db = $c->contacts;  
// 选择用于存储联系人和地址的集合  
$people = $db->people;  
$addresses = $db->addresses;  
// 指定地址  
$address = array(  
  "Street" => "St. Annastraat 44",  
  "Place" => "Monster",  
  "Postal Code" => "2681 SR",  
  "Country" => "Netherlands"  
);  
// 保存地址  
$addresses->insert($address);  
// 添加一个居住在该地址的联系人  
$contact = array(  
  "First Name" => "Melvyn",  
  "Last Name" => "Babel",  
  "Age" => 35,  
  "Address" => $address['_id']  
);  
$people->insert($contact);
```

假设现在希望找到之前联系人的地址信息。那么为了实现该操作, 需要查询 `address` 字段中的 Object ID; 可以在 `addresses` 集合中找到该信息(假设你知道该集合的名称)。

这种方式是可行的, 但我们推荐使用 DBRef 引用另一个文档。这是因为 DBRef 将使用一种通用格式, 数据库和所有的驱动都能够识别。接下来让我们学习一下之前样例的 DBRef 版本。不过, 在开始之前, 首先需要学习 DBRef 类的 `create()` 函数; 以后我们将使用该类创建引用。

函数 `create()` 将接受 3 个参数:

- `collection`: 指定保存信息的集合名称(不需要数据库名)。
- `id`: 指定要链接到的文档的 ID。
- `database`: 指定文档所在的数据库名。

下面的样例将使用 `create()` 函数在一个文档中创建对地址的引用：

```
// 连接到数据库
$c = new Mongo();
$db = $c->contacts;

// 选择用于存储联系人和地址的集合
$people = $db->people;
$addresses = $db->addresses;

// 指定地址
$address = array(
    "Street" => "WA Visser het Hoofthlaan 2621",
    "Place" => "Driebergen",
    "Postal Code" => "3972 SR",
    "Country" => "Netherlands"
);
// 保存地址
$addresses->insert($address);

// 创建对地址的引用
$addressRef = MongoDBRef::create($addresses->getName(), $address['_id']);

// 添加一个居住在该地址的联系人
$contact = array(
    "First Name" => "Ivo",
    "Last Name" => "Lauw",
    "Age" => 24,
    "Address" => $addressRef
);
$people->insert($contact);
```

注意：

本例中的 `getName()` 函数用于获取集合的名称。

获取信息

到目前为止，我们已经使用 `DBRef` 创建了一个引用。现在学习如何获得所引用的信息，并正确地显示出其中的内容。通过 `MongoDBRef` 的 `get()` 函数可以实现。

`MongoDBRef` 的 `get()` 函数接受两个参数。第一个参数指定将要使用的数据库；第二个参数提供要读取的引用：

```
// 连接到数据库
$c = new Mongo();
// 选择数据库'contacts'的'people'集合
$people = $c->contacts->people;
// 定义搜索参数
$lastname = array("Last Name" => "Lauw");
// 查找联系人，并存储在变量$person中
$person = $people->findOne(array("Last Name" => "Lauw"));
// 解析地址的引用
$address = MongoDBRef::get($people->db, $person['Address']);
// 输出所匹配联系人的地址
print_r($address);
```

结果将显示出被引用的文档：

```
Array (
  [_id] => MongoId Object ( )
  [Street] => WA Visser het Hooftlaan 2621
  [Place] => Driebergen
  [Postal Code] => 3972 SR
  [Country] => Netherlands
)
```

DBRef 提供了一种存储数据引用的好方法，并且还允许灵活地使用集合和数据库名。

6.9 GridFS 和 PHP 驱动

之前的章节已经详细地讲解了 GridFS 及其优点。例如，如何使用该技术存储和读取数据，以及 GridFS 相关的技术。本节将学习如何在 PHP 中使用 GridFS 存储和获取文档。

PHP 驱动中包含了它自己的类，用于处理 GridFS；以下是它的 3 个最重要的类以及它们的功能：

- **MongoGridFS**：从数据库中存储和读取文件。该类包含了几个方法，包括 `delete()`、`find()`、`storeUpload()` 等 6 个其他方法。
- **MongoGridFSFile**：基于数据库中的特定文件进行工作。它包含了一些函数，例如 `__construct()`、`getFilename()`、`getSize()` 和 `write()`。
- **MongoGridFSCursor**：基于游标进行工作。它包含了一些有用的函数，例如 `__construct()`、`current()`、`getNext()` 和 `key()`。

下面学习如何使用 PHP 将文件上传到数据库中。

注意：

下面样例中的代码，在没有提供具有上传文件功能的 HTML 表单的情况下是无法工作的。这样的代码超出了本章的范围，所以这里并未显示它们。

6.9.1 存储文件

通过使用 GridFS 的 `storeUpload()` 函数可以将文件存储到数据库中。该函数将接受两个参数：一个参数表示存储文件的字段名；另一个可选参数用于指定文件的元数据。一旦使用该选项，`storeUpload()` 函数将返回所存储文件的 `_id`。

下面的样例代码将展示如何使用 `storeUpload()` 函数：

```
// 连接到数据库
$c = new MongoClient();
// 选择数据库
$db = $c->contacts;
// 定义 GridFS 类用于保证我们可以处理文件
$gridFS = $db->getGridFS();
// 指定 HTML 字段的 name 特性
$file = 'fileField';
// 指定文件的元数据 (可选的)
$metadata = array('uploadDate' => date());
// 将文件上传至数据库
$id = $gridFS->storeUpload($file, $metadata);
```

这就是所有的代码，如你所见，`$id` 被用作存储文件(在数据库中)时的参数。还可以使用该参数引用 `DBRef` 指向的数据。

6.9.2 在已存储的文件中添加元数据

有时你可能希望在已存储的文件中添加更多的元数据。但是，`_id` 字段是唯一的额外元数据，当在联系卡中存储图像时可用该字段引用数据。不过当试图通过这些标签搜索数据时，你会发现这种方式弊大于利，

下面的样例演示了如何在已上传的数据中存储元数据。本例基于之前的代码构建，并将用到 `$id` 参数。很明显，可以使用任何其他希望使用的搜索条件自定义该代码：

```
// 指定将要添加的元数据
$metadata = array('$set' => array("Tag" => "Avatar"));
// 通过搜索条件匹配将要应用元数据的文件
$criteria = array('_id' => $id);
// 插入元数据
$db->grid->update($criteria, $metadata);
```

6.9.3 获取文件

当然，如果将来不能获取这些文件的话，那么在数据库中存储文件也就没什么作用了。获取文件与存储文件一样简单。接下来的两个例子将分别演示如何获得已存储文件的名称和内容。

下面的样例演示了如何获取已存储文件的名称，通过使用 `getFilename()` 函数实现：

```
// 连接到数据库
$c = new MongoClient();
$db = $c->contacts;
// 初始化 GridFS
$gridFS = $db->getGridFS();
// 找到 GridFS 存储中的所有文件，并存储在参数 $cursor 中
$cursor = $gridFS->find();
// 返回所有文件的名称
foreach($cursor as $object) {
    echo "Filename:". $object->getFilename();
}
```

这非常简单！当然，该例假设数据库中之前已经存储了一些数据。在添加更多数据之后，你可能希望在 `find()` 函数中添加更多的搜索参数，或者希望搜索更具体的数据。注意：`find()` 函数将搜索每个已上传文件的元数据(如本章之前所讲解的)。

你可能会好奇如何才能获取这些文件。毕竟，获取数据可能是最后一个步骤。通过使用 `getBytes()` 函数可以将数据发送到浏览器。接下来的样例将使用 `getBytes()` 函数获取之前已存储的图片。注意，可以通过查询数据库获得 `_id` 值(接下来的样例只是编造了一些参数)。另外，必须指定内容类型，因为从逻辑上讲，当再次构建数据时，浏览器无法识别数据的类型：

```
// 连接到数据库
$c = new MongoClient();
// 指定数据库名称
$db = $c->contacts;
// 初始化 GridFS 文件集合
$gridFS = $db->getGridFS();
// 指定搜索参数
```

```
$id = new MongoId('4c555c70be90968001080000');  
// 获取文件  
$file = $gridFS->findOne(array('_id' => $id));  
// 指定文件头并将数据写入文件  
header('Content-Type: image/jpeg');  
echo $file->getBytes();  
exit;
```

6.9.4 删除数据

通过使用 `delete()` 函数可以删除任何之前已存储的数据。该函数将接受一个参数：文件的 `_id`。下面的样例演示了如何使用 `delete()` 函数删除匹配对象 ID `4c555c70be90968001080000` 的文件：

```
// 连接到数据库  
$c = new MongoClient();  
// 指定文件名称  
$db = $c->contacts;  
// 初始化 GridFS  
$gridFS = $db->getGridFS();  
// 通过 ID 指定文件  
$id = new MongoId('4c555c70be90968001080000');  
$file = $gridFS->findOne(array('_id' => $id));  
// 使用 remove() 函数删除文件  
$gridFS->delete($id);
```

6.10 小结

本章深入讲解了如何使用 MongoDB 的 PHP 驱动。例如，如何使用 PHP 驱动中最常用的函数，包括 `insert()`、`update()` 和 `modify()` 函数。还学习了如何使用 PHP 驱动中的 `find()` 函数搜索文档。最后，学习了如何使用 `DBRef` 功能，以及如何使用 `GridFS` 存储和获取文件。

因为只有一章内容，所以不可能面面俱到，无法对 MongoDB PHP 驱动的所有相关知识进行详细讲解；不过尽管如此，本章还是提供了执行大多数操作的必要基础知识，可帮助你实现希望完成的操作。另外，在事情变得更复杂时，我们还学习了如何使用服务器端命令进行处理。

下一章我们将学习相同的概念，但这次将应用在 Python 驱动中。

第 7 章

Python 和 MongoDB

Python 是较易于学习和掌握的编程语言之一。对于编程新手来说，它是一门很棒的语言。如果你已经非常熟悉编程的话，那么可以很快掌握它。

Python 可以在保证代码具有良好可读性的情况下，快速开发应用。记住，本章将演示如何使用 MongoDB 的 Python 驱动编写出简单、优雅、清晰和强大的代码(也称为 PyMongo 驱动，本章将交替使用这两个术语)。

首先我们将学习 Connection() 函数，通过它可以建立与数据库的连接。然后我们将学习如何编写文档或词典，以及如何插入它们。接着我们将学习如何使用 Python 驱动中的 find() 或 find_one() 命令获取文档。这两个命令都可以接受一组丰富的查询修改操作符，用于减小搜索目标并使查询实现更简单。接下来我们学习执行更新时可使用的许多操作符。最后学习如何使用 PyMongo 在文档甚至数据库级别删除数据。另外，作为附加的部分，本章还将学习如何使用 DBRef 模块引用存储在其他部分的数据。

注意：

通过本章内容，你将看到许多实际的代码样例，它们将演示文中提到的概念。代码之前的 3 个大于号(>>>)表示该命令是在 Python shell 中编写的。查询代码将以粗体显示，而结果输出将以普通文本显示。

7.1 在 Python 中使用文档

正如之前章节中所提到的，MongoDB 使用 BSON 样式的文档，PHP 使用关联数组。在 Python 中使用的是词典。如果之前曾接触过 MongoDB 控制台，相信你一定会喜欢 Python。毕竟，它们的语法如此相似，学习曲线几乎可以忽略不计。

在之前的章节中已经讲解过 MongoDB 文档的结构，这里就不再重复解释。下面通过样例演示文档在 Python shell 中的样子：

```
item = {
    "Type" : "Laptop",
    "ItemNumber" : "1234EXD",
    "Status" : "In use",
    "Location" : {
        "Department" : "Development",
    "Building" : "2B",
    "Floor" : 12,
    "Desk" : 120101,
    "Owner" : "Anderson, Thomas"
```



```
    },  
    "Tags" : ["Laptop", "Development", "In Use"]  
  }  
}
```

将 Python 术语“词典”记在脑中，在大多数情况下，本章都会使用它在 MongoDB 中对等的术语：文档。毕竟，大多数时间我们都在使用 MongoDB 文档。

7.2 使用 PyMongo 模块

Python 驱动以模块的方式工作。可以将它们看做 PHP 驱动中的类。PyMongo 驱动中的每个模块都将负责一组操作。以下每个任务(还有更多)都将由一个单独的模块来完成：建立连接、使用数据库、使用集合、操作游标、使用 DBRef 模块、转换 ObjectId 和运行服务器端 JavaScript 代码。

本章将首先讲解 PyMongo 中一些最基础但也是最有用的一组操作。接下来本章将通过简单和易于理解的代码，一步一步演示如何使用这些命令，你还可以将这些代码直接粘贴复制到 Python shell(或脚本)中。学完这些样例之后，你很快就可以管理 MongoDB 数据了。

7.3 连接和断开

要建立与数据库的连接，首先需要导入 PyMongo 驱动的 MongoClient 模块，这样才可以使用它建立连接。在 shell 中输入以下语句以加载 MongoClient 模块：

```
>>> from pymongo import MongoClient
```

一旦 MongoDB 服务启动并运行(如果希望连接到它，这是必需的操作)，就可以调用 MongoClient() 函数来连接到该服务。

如果未指定额外的参数，该函数将连接到本机的 MongoDB 服务(本机的默认端口号是 27017)。下面的代码将建立起数据库连接：

```
>>> c = MongoClient()
```

通过 MongoDB 服务 shell 可以看到有连接建立成功。建立连接之后，就可以使用词典 c 引用该连接，和 shell 使用的 db 以及 PHP 中的 \$c 是一样的。接下来，选择希望使用的数据库，将该数据库赋予本地的词典 db。下面的例子将演示如何使用 inventory 数据库：

```
>>> db = c.inventory  
>>> db  
Database(Connection('localhost', 27017), u'inventory')
```

本例的输出显示我们已经连接到本机 MongoDB 服务，并且正在使用 inventory 数据库。

在选择数据库之后，接着以相同的方式选择 MongoDB 集合。因为数据库名已经存储在词典 db 中，所以可以使用 db 选择集合名；此时 db 中的集合被称为 items：

```
>>> collection = db.items
```

7.4 插入数据

剩下的工作就是定义文档，并将它存储到词典中。下面的样例演示了如何在 Python 中定义文档：

```
>>> item = {
...     "Type" : "Laptop",
...     "ItemNumber" : "1234EXD",
...     "Status" : "In use",
...     "Location" : {
...         "Department" : "Development",
...         "Building" : "2B",
...         "Floor" : 12,
...         "Desk" : 120101,
...         "Owner" : "Anderson, Thomas"
...     },
...     "Tags" : ["Laptop", "Development", "In Use"]
... }
```

完成文档的定义之后，同样可以使用 insert() 函数(与 MongoDB shell 中的 insert() 函数相同)向数据库中插入该文档：

```
>>> collection.insert(item)
ObjectId('4c57207b4abffe0e0c000000')
```

这就是所有要做的工作：定义文档并使用 insert() 函数插入它。

在插入文档时，还有一个有趣的技巧值得学习：同时插入多个文档。通过在单个词典中指定多个文档，然后插入该词典数据的方式可以实现同时插入多个文档。结果将返回两个 ObjectId 值；注意下面的样例是如何使用括号的：

```
>>> two = [{
...     "Type" : "Laptop",
...     "ItemNumber" : "2345FDX",
...     "Status" : "In use",
...     "Location" : {
...         "Department" : "Development",
...         "Building" : "2B",
...         "Floor" : 12,
...         "Desk" : 120102,
...         "Owner" : "Smith, Simon"
...     },
...     "Tags" : ["Laptop", "Development", "In Use"]
... },
... {
...     "Type" : "Laptop",
...     "ItemNumber" : "3456TFS",
...     "Status" : "In use",
...     "Location" : {
...         "Department" : "Development",
...         "Building" : "2B",
...         "Floor" : 12,
...         "Desk" : 120103,
...         "Owner" : "Walker, Jan"
...     },
...     "Tags" : ["Laptop", "Development", "In Use"]
... }
... }
```

```

...     "Tags" : ["Laptop","Development","In Use"]
...   }]
>>> collection.insert(two)
[ObjectId('4c57234c4abffe0e0c000001'), ObjectId('4c57234c4abffe0e0c000002')]

```

7.5 搜索数据

PyMongo 提供了两个函数用于搜索数据: `find_one()` 用于搜索集合中匹配搜索条件的单个文档; `find()` 可基于所提供的参数搜索多个文档(如果不指定任何参数, 那么 `find()` 将返回集合中的所有文档)。下面通过一些样例演示如何搜索数据。

7.5.1 搜索单个文档

正如之前提到的, `find_one()` 函数可用于搜索单个文档。该函数类似于 MongoDB shell 中的 `findOne()` 函数, 所以掌握该函数并不困难。如果未使用任何参数, 那么该函数默认将返回集合中的第一个文档, 如下所示:

```

>>> collection.find_one()
{
  u'Status': u'In use',
  u'Tags': [u'Laptop', u'Development', u'In Use'],
  u'ItemNumber': u'1234EXD',
  u'Location':{
    u'Department': u'Development',
    u'Building': u'2B',
    u'Floor': 12,
    u'Owner': u'Anderson, Thomas',
    u'Desk': 120101
  },
  u'_id': ObjectId('4c57207b4abffe0e0c000000'),
  u'Type': u'Laptop'
}

```

可以指定额外的参数用于保证返回的第一个文档匹配查询条件。函数 `find()` 中的所有参数也都可以用在 `find_one()` 中, 不过 `limit` 参数将被忽略。如果是在 shell 中定义查询参数, 只需要按照它们应有的格式编写即可; 需要指定键及对应的值。例如, 假设我们希望搜索一个 `ItemNumber` 值为 3456TFS 的文档, 并且不希望返回文档的 `_id`。下面的样例将演示如何完成该搜索, 并显示出结果:

```

>>> collection.find_one({"ItemNumber" : "3456TFS"} ,fields={'_id' : False})
{
  u'Status': u'In use',
  u'Tags': [u'Laptop', u'Development', u'In Use'],
  u'ItemNumber': u'3456TFS',
  u'Location': {
    u'Department': u'Development',
    u'Building': u'2B',
    u'Floor': 12,
    u'Owner': u'Walker, Jan',
    u'Desk': 120103
  },
  u'Type': u'Laptop'
}

```

注意:

Python 是大小写敏感的。因此, `true` 和 `false` 不同于 `True` 和 `False`。

如果搜索条件在多个文档中出现, 那么还可以指定额外的查询操作符。例如, 如果查询 `{"Department": "Development"}` 的话, 那么结果将返回多个文档。之后我们将会简单地演示这种情况; 不过, 目前还是先学习如何返回多个文档(而不是一个)。这可能与你的猜想稍有不同。

7.5.2 搜索多个文档

通过使用 `find()` 函数可以返回多个文档。至此, 本书已经多次在 MongoDB 中使用该命令, 所以你可能已经对它相当熟悉了。这个概念在 Python 中是相同的: 在括号之间指定查询参数以搜索特定的信息。

不过, 结果返回到屏幕的方式稍有不同。如同在 PHP 和 shell 中一样, 查询一组文档最终将返回一个游标。不过, 与在 shell 中不同, 不能使用 `db.items.find()` 显示出所有结果。相反, 需要使用游标获取所有文件。下面的样例演示了如何显示出 `items` 集中的所有文档(注意之前已经定义了匹配集合名的 `collection`; 为了保持代码清晰, 这里不再显示执行结果):

```
>>> for doc in collection.find():
...     doc
... 
```

注意单词 `doc` 之前的缩进。如果未使用缩进, 那么结果将会显示错误消息, 表示未找到预期的缩进块。这是 Python 的优势之一, 它通过将这种缩进作为块分隔符的方式, 保证代码良好有序。放心, 你很快会习惯此编码约定。如果确实忘记了添加缩进, 那么结果将显示下面的错误消息:

```
File "<stdin>", line 2
    doc
    ^
IndentationError: expected an indented block
```

接下来我们将学习如何在 `find()` 函数中指定查询操作符。具体用法与之前本书中的其他样例是一致的:

```
>>> for doc in collection.find({"Location.Owner" : "Walker, Jan"}):
...     doc
... 
```

```
{
  u'Status': u'In use',
  u'Tags': [u'Laptop', u'Development', u'In Use'],
  u'ItemNumber': u'3456TFS',
  u'Location': {
    u'Department': u'Development',
    u'Building': u'2B',
    u'Floor': 12,
    u'Owner': u'Walker, Jan',
    u'Desk': 120103
  },
  u'__id': ObjectId('4c57234c4abffe0e0c000002'),
  u'Type': u'Laptop'
}
```

7.5.3 使用点操作符

点操作符可用于搜索嵌入对象中的匹配元素。之前的代码显示了一个这样的样例。使用该技术时，只需要指定内嵌对象中一个数据项的键名，即可搜索它，如下所示：

```
>>> for doc in collection.find({"Location.Department" : "Development"}):  
...     doc  
...
```

该例将返回所有部门设置为 Development 的文档。在搜索简单数组中的信息时(例如标签)，需要在参数中填入所有希望匹配的标签：

```
>>> for doc in collection.find({"Tags" : "Laptop"}):  
...     doc  
...
```

7.5.4 返回字段

如果文档相对较大，并且不希望返回文档存储的所有键值对，那么可以在 find()函数中添加额外的参数，用于返回某个特定的字段集，使用真表示返回，使用假表示隐藏。这可通过提供 fields 参数，并在搜索条件之后紧接着一个字段列表的方式来实现。注意，除了 _id 字段，在查询中不能混合使用真和假。

下面的样例将只返回当前所有者的名字、数据编号和对象 ID(除非显式地告诉 MongoDB 不要返回它，否则 ID 将一直被返回)：

```
>>> for doc in collection.find({'Status' : 'In use'}, fields={'ItemNumber' : True,  
'Location.Owner': True}):  
...     doc  
...  
{  
    'ItemNumber': u'1234EXD',  
    u'_id': ObjectId('4c57207b4abffe0e0c000000'),  
    u'Location': {  
        u'Owner': u'Anderson, Thomas'  
    }  
}  
{  
    'ItemNumber': u'2345FDX',  
    u'_id': ObjectId('4c57234c4abffe0e0c000001'),  
    u'Location': {  
        u'Owner': u'Smith, Simon'  
    }  
}  
{  
    'ItemNumber': u'3456TFS',  
    u'_id': ObjectId('4c57234c4abffe0e0c000002'),  
    u'Location': {  
        u'Owner': u'Walker, Jan'  
    }  
}
```

我想你会同意，这种指定搜索条件的方式非常方便。

7.5.5 使用 sort()、limit()和 skip()简化查询

函数 `sort()`、`limit()` 和 `skip()` 将使查询的实现变得更容易。其中的每一个函数都有自己的用处，当你将它们结合起来使用时，它们会变得更好更强大。可以使用 `sort()` 函数按特定的键对结果进行排序；使用 `limit()` 函数限制返回结果的总数；使用 `skip()` 函数在返回所有匹配查询的文档之前，忽略前 `n` 个文档。

下面通过一些样例演示这些函数的用法，首先从 `sort()` 函数开始。为了节省空间，下面的样例将使用另一个参数用于保证只返回一些特定的字段：

```
>>> for doc in collection.find({'Status' : 'In use'},
... fields={'ItemNumber' : True, 'Location.Owner' : True}).sort('ItemNumber'):
...     doc
...
{
    'ItemNumber': u'1234EXD',
    '_id': ObjectId('4c57207b4abffe0e0c000000'),
    'Location': {
        'Owner': u'Anderson, Thomas'
    }
}
{
    'ItemNumber': u'2345FDX',
    '_id': ObjectId('4c57234c4abffe0e0c000001'),
    'Location': {
        'Owner': u'Smith, Simon'
    }
}
{
    'ItemNumber': u'3456TFS',
    '_id': ObjectId('4c57234c4abffe0e0c000002'),
    'Location': {
        'Owner': u'Walker, Jan'
    }
}
```

接下来学习 `limit()` 函数。本例将使用该函数只返回在集合中搜索到的头两个文档，并且只返回它们的 `ItemNumber` (注意本例中未指定搜索条件)：

```
>>> for doc in collection.find({}, {"ItemNumber" : "true"}).limit(2):
...     doc
...
{'ItemNumber': u'1234EXD', 'ItemNumber': u'2345FDX', 'ItemNumber': u'3456TFS'}
```

在返回文档集之前，可以使用 `skip()` 函数忽略一些文档，如下所示：

```
>>> for doc in collection.find({}, {"ItemNumber" : "true"}).skip(2):
...     doc
...
{'ItemNumber': u'3456TFS', 'ItemNumber': u'4567GHI', 'ItemNumber': u'5678JKL'}
```

还可以结合使用这 3 个函数，只选择匹配文档中的一部分，同时忽略其中特定数目的文档，最终再对结果进行排序：

```
>>> for doc in collection.find({'Status' : 'In use'},
... fields={'ItemNumber' : True, 'Location.Owner' : True}).limit(2).skip(1).sort
...     'ItemNumber'): doc
...
{'ItemNumber': u'3456TFS', 'ItemNumber': u'4567GHI'}
```

```

...
{
    u'ItemNumber': u'2345FDX',
    u'_id': ObjectId('4c57234c4abffe0e0c000001'),
    u'Location': {
        u'Owner': u'Smith, Simon'
    }
}
{
    u'ItemNumber': u'3456TFS',
    u'_id': ObjectId('4c57234c4abffe0e0c000002'),
    u'Location': {
        u'Owner': u'Walker, Jan'
    }
}
}

```

本例所完成的操作——限制返回结果的数目，并忽略特定数目的结果——通常被称为分页。通过 \$slice 操作符可以更轻松地实现该功能，本章稍后将讲解如何实现。

7.5.6 聚集查询

如之前提到的，MongoDB 提供了一组强大的聚集工具(关于这些工具的详细信息请参阅第 4 章)。可以通过 Python 驱动使用所有这些工具。在这些工具中，可以使用 count() 函数执行对数据的统计；使用 distinct() 函数获取一个不含重复值的唯一值列表；最后但不是最不重要的，是使用 map_reduce() 函数对数据分组，并以批量处理的方式操作结果或执行统计。

这组命令可以单独使用，也可以一起使用，通过这种方式可以高效地查询出所需的信息。

除了基本的聚集命令，PyMongo 驱动还包含聚集框架。这个强大的特性可以帮助计算聚集值，而不需要使用通常过于复杂的映射/规约(或 MapReduce)框架。

1. 使用 count() 统计数目

如果希望统计匹配搜索条件的文档总数，那么可以使用 count() 函数。该函数不会像 find() 函数那样返回所有文档；相反，它只返回一个整数值，表示所有找到的文档的总数。

接下来学习一些简单的样例。首先返回整个集中文档的总数，无须指定任何条件：

```
>>> collection.count()
3
```

还可以指定更准确的统计查询：

```
>>> collection.find({"Status": "In use", "Location.Owner": "Walker, Jan"}).count()
1
```

如果需要对符合条件的文档进行快速的数量统计，使用 count() 函数是很好的选择。

2. 使用 distinct() 统计唯一数据的数目

函数 count() 是获得返回文档总数的一个好方法。不过，有时你可能不小心会在集合中添加一些重复的数据(例如只是忘记删除或修改一个旧文档)，这时如果希望获得不包含重复数据的准确统计，那么可以使用 distinct() 函数。该函数将确保只有唯一的数据被返回。下面首先在集合中添加另一个文档，并使用一个之前已经添加过的 ItemNumber：

```
>>> dup = ( {
```

```

    "ItemNumber" : "2345FDX",
    "Status" : "Not used",
    "Type" : "Laptop",
    "Location" : {
      "Department" : "Storage",
      "Building" : "1A"
    },
    "Tags" : ["Not used","Laptop","Storage"]
  } )
>>> collection.insert(dup)
ObjectId('4c592eb84abffe0e0c000004')

```

这时如果使用 `count()` 函数，统计出来的唯一数据的总数将是错误的：

```

>>> collection.find({}).count()
4

```

相反，此时可以使用 `distinct()` 函数确保所有重复数据被过滤掉：

```

>>>collection.distinct("ItemNumber")
[u'1234EXD', u'2345FDX', u'3456TFS']

```

3. 使用聚集框架对数据分组

聚集框架是在不使用 MapReduce 的情况下，用于统计聚集值的好工具。尽管 MapReduce 非常强大，并可在 PyMongo 驱动中使用，但聚集框架也可以完成大多数工作，并且性能更好。为了演示这一点，现在使用 `aggregate()` 函数最强大的管道操作符 `$group` 将之前添加的文档按照标签进行分组，然后使用 `$sum` 聚集表达式基于它进行统计，如下所示：

```

>>> collection.aggregate([
... {'$unwind' : '$Tags'},
... {'$group' : {'_id' : '$Tags', 'Totals' : {'$sum' : 1}}}
... ])

```

首先，`aggregate()` 函数使用 `$unwind` 管道操作符，根据文档的 `'$Tags'` 数组(注意必须在名字中使用 `$`)创建了一个文档流。接下来，调用 `$group` 管道操作符，将所有唯一标签的值用作它们的 `'_id'`，创建一个单行以及它们的总数——使用 `$group` 的 `$sum` 表达式计算 `'Totals'` 值。结果输出将如下所示：

```

{
  u'ok': 1.0,
  u'result': [
    {u'_id': u'Laptop', u'Totals': 4},
    {u'_id': u'In Use', u'Totals': 3},
    {u'_id': u'Development', u'Totals': 3},
    {u'_id': u'Storage', u'Totals': 1},
    {u'_id': u'Not used', u'Totals': 1}
  ]
}

```

该输出将正确地返回所请求的数据。不过，如果我们希望按照每行数据的 `'Totals'` 将结果排序，该如何做到呢？可以通过添加另一个管道操作符 `$sort` 来实现。不过，在此之前我们需要导入 SON 模块：

```

>>> from bson.son import SON

```


下面的样例将基于'Totals'值对结果进行降序排序:

```
>>> collection.aggregate([
...     {'$unwind' : '$Tags'},
...     {'$group' : {'_id' : '$Tags', 'Totals' : {'$sum' : 1}}},
...     {'$sort' : SON(['Totals', -1])}]
... ])
```

结果将按照降序输出, 如下所示:

```
{
  u'ok': 1.0,
  u'result': [
    {u'_id': u'Laptop', u'Totals': 4},
    {u'_id': u'In Use', u'Totals': 3},
    {u'_id': u'Development', u'Totals': 3},
    {u'_id': u'Storage', u'Totals': 1},
    {u'_id': u'Not used', u'Totals': 1}
  ]
}
```

除了\$sum 管道表达式, \$group 管道操作符还支持各种其他不同的操作符, 这里列出了其中的一部分:

- \$push: 创建并返回一个数组, 它包含了在分组中搜索到的所有数据。
- \$addToSet: 在分组中创建并返回一个数组, 它包含了在分组中搜索到的所有唯一数据。
- \$first: 返回分组中找到的第一个数据。
- \$last: 返回分组中找到的最后一个数据。
- \$max: 返回分组中找到的最大值。
- \$min: 返回分组中找到的最小值。
- \$avg: 返回分组中找到的平均值。

在本例中, 我们已经学习了\$group、\$unwind 和\$sort 管道操作符的用法, 但还有许多强大的管道操作符尚未学习, 例如\$geoNear 操作符。第 4、第 6、第 8 章将详细讲解聚集框架及其操作符。

7.5.7 使用 hint()指定索引

可以使用 hint()函数指定查询数据时所应使用的索引。使用该函数可以帮助提高查询性能。在 Python 中, hint()也基于游标执行。不过要记住, 在 Python 中指定的提示索引名称需要与传入 create_index()函数中的名称一致。

下面的例子将先创建一个索引, 然后指定使用该索引搜索数据。不过在使用升序对数据排序之前, 首先需要使用 import()函数导入 ASCENDING 方法。最后, 执行 create_index()函数即可:

```
>>> from pymongo import ASCENDING
>>> collection.create_index(["ItemNumber", ASCENDING])
u'ItemNumber_1'

>>> for doc in collection.find({"Location.Owner" : "Walker,
Jan"}) .hint(["ItemNumber",ASCENDING]):
...     doc
...
  u'Status': u'In use',
  u'Tags': [u'Laptop', u'Development', u'In Use'],
```

```

u'ItemNumber': u'3456TFS',
u'Location': {
    u'Department': u'Development',
    u'Building': u'2B',
    u'Floor': 12,
    u'Owner': u'Walker, Jan',
    u'Desk': 120103
},
u'_id': ObjectId('4c57234c4abffe0e0c000002'),
u'Type': u'Laptop'
}

```

当集合的大小不断增长时，使用索引可以帮助极大地提高性能(关于性能调优的更多细节请参阅第 10 章)。

7.5.8 使用条件操作符重定义查询

通过使用条件操作符可以重定义查询。MongoDB 包含了超过半打的条件操作符，可通过 PyMongo 访问它们；它们与之前章节中学习的条件操作符是一样的。下面将学习 Python 中可用的条件操作符，以及如何使用它们重定义 Python 中的 MongoDB 查询。

1. 使用 \$lt、\$gt、\$lte 和 \$gte 操作符

首先要学习的是 \$lt、\$gt、\$lte 和 \$gte 条件操作符。可以使用 \$lt 操作符搜索任何小于 n 的数值信息。该操作符接受一个参数：数字 n，它将指定对数值的限制。下面的样例将查找所有桌子号码小于 120102 的文档。注意比较值自身不会被包含在结果中：

```

>>> for doc in collection.find({"Location.Desk" : {"$lt" : 120102} }):
...     doc
...
{
  u'Status': u'In use',
  u'Tags': [u'Laptop', u'Development', u'In Use'],
  u'ItemNumber': u'1234EXD',
  u'Location': {
    u'Department': u'Development',
    u'Building': u'2B',
    u'Floor': 12,
    u'Owner': u'Anderson, Thomas',
    u'Desk': 120101
  },
  u'_id': ObjectId('4c57207b4abffe0e0c000000'),
  u'Type': u'Laptop'
}

```

与此类似，可以使用 \$gt 操作符搜索所有大于比较值的数据。注意，比较值自身同样也不会被包含在结果中：

```

>>> for doc in collection.find({"Location.Desk" : {"$gt" : 120102} }):
...     doc
...
{
  u'Status': u'In use',
  u'Tags': [u'Laptop', u'Development', u'In Use'],

```

第 II 部分 在开发中使用 MongoDB

```
u'ItemNumber': u'3456TFS',
u'Location': {
  u'Department': u'Development',
  u'Building': u'2B',
  u'Floor': 12,
  u'Owner': u'Walker, Jan',
  u'Desk': 120103
},
u'_id': ObjectId('4c57234c4abffe0e0c000002'),
u'Type': u'Laptop'
}
```

如果希望在结果中包含比较值，那么可以使用 `$lte` 或 `$gte` 操作符分别搜索小于或等于 `n` 的值以及大于或等于 `n` 的值。下面的样例演示了如何使用这两个操作符：

```
>>> for doc in collection.find({"Location.Desk" : {"$lte" : 120102}}):
...     doc
...
{
  u'Status': u'In use',
  u'Tags': [u'Laptop', u'Development', u'In Use'],
  u'ItemNumber': u'1234EXD',
  u'Location': {
    u'Department': u'Development',
    u'Building': u'2B',
    u'Floor': 12,
    u'Owner': u'Anderson, Thomas',
    u'Desk': 120101
  },
  u'_id': ObjectId('4c57207b4abffe0e0c000000'),
  u'Type': u'Laptop'
}
{
  u'Status': u'In use',
  u'Tags': [u'Laptop', u'Development', u'In Use'],
  u'ItemNumber': u'2345FDX',
  u'Location': {
    u'Department': u'Development',
    u'Building': u'2B',
    u'Floor': 12,
    u'Owner': u'Smith, Simon',
    u'Desk': 120102
  },
  u'_id': ObjectId('4c57234c4abffe0e0c000001'),
  u'Type': u'Laptop'
}

>>> for doc in collection.find({"Location.Desk" : {"$gte" : 120102}}):
...     doc
...
{
  u'Status': u'In use',
  u'Tags': [u'Laptop', u'Development', u'In Use'],
  u'ItemNumber': u'2345FDX',
  u'Location': {
    u'Department': u'Development',
    u'Building': u'2B',
    u'Floor': 12,
```

```

    u'Owner': u'Smith, Simon',
    u'Desk': 120102
  },
  u'_id': ObjectId('4c57234c4abffe0e0c000001'),
  u'Type': u'Laptop'
}
{
  u'Status': u'In use',
  u'Tags': [u'Laptop', u'Development', u'In Use'],
  u'ItemNumber': u'3456TFS',
  u'Location': {
    u'Department': u'Development',
    u'Building': u'2B',
    u'Floor': 12,
    u'Owner': u'Walker, Jan',
    u'Desk': 120103
  },
  u'_id': ObjectId('4c57234c4abffe0e0c000002'),
  u'Type': u'Laptop'
}

```

2. 使用\$ne 搜索不匹配的数据

可以使用\$ne(不等于)搜索集合中所有不匹配指定条件的文档。该操作符只接受一个参数——键值信息，所有含有该信息的文档都不应该出现在结果中：

```
>>> collection.find({"Status" : {"$ne" : "In use"}}).count()
1
```

3. 使用\$in 指定匹配的数组

通过使用\$in 操作符可以指定一个包含所有可能匹配的值的数组。

例如，目前在搜索两种类型的开发计算机：未使用的和开发中的。另外，假设你希望将返回结果限制为两个，并且只返回 ItemNumber：

```
>>> for doc in collection.find({"Tags" : {"$in" : ["Not used","Development"]}} ,
fields={"ItemNumber":"true"}).limit(2):
...     doc
...
{'ItemNumber': u'1234EXD', u'_id': ObjectId('4c57207b4abffe0e0c000000')}
{'ItemNumber': u'2345FDX', u'_id': ObjectId('4c57234c4abffe0e0c000001')}
```

4. 使用\$nin 指定不匹配的数组

\$nin 的使用方式与\$in 操作符一致；区别在于该操作符将排除所有匹配指定数组中某个值的文档。例如，下面的查询将搜索开发部门中未使用的计算机的信息：

```
>>> for doc in collection.find({"Tags" : {"$nin" : ["Development"]}} , fields={"ItemNumber":
True}):
...     doc
...
{'ItemNumber': u'2345FDX', u'_id': ObjectId('4c592eb84abffe0e0c000004')}
```

5. 搜索匹配数组值的文档

\$in 操作符可用于搜索匹配指定数组中任何一个值的文档，而通过\$all 操作符可以搜索任何

匹配指定数组中所有值的文档。\$all 的语法与\$in 一致:

```
>>> for doc in collection.find({"Tags" : {"$all" : ["Storage", "Not used"]}},
fields ={"ItemNumber": "true"}):
...     doc
...
{'ItemNumber': 'u'2345FDX', 'u'_id': ObjectId('4c592eb84abffe0e0c000004')}
```

6. 使用\$or 指定多个匹配表达式

可以使用\$or 操作符指定文档中可以包含的多个值，只有当文档包含至少其中一个值时，才会显示在结果中。它类似于\$in 操作符；区别在于使用\$or 可以同时指定键和值。还可以将\$or 操作符与其他键值对结合使用。下面我们将通过例子演示它的用法。

接下来的例子将返回所有位置为 Storage 或所有者为 AndersonThomas 的文档:

```
>>> for doc in collection.find({"$or" : [ { "Location.Department" : "Storage" },
... { "Location.Owner" : "Anderson, Thomas" } ] }):
...     doc
...
...
```

还可以将之前的代码与其他键值对结合使用，如下所示:

```
>>> for doc in collection.find({ "Location.Building" : "2B", "$or" :
[ { "Location.Department" : "Storage" },
... { "Location.Owner" : "Anderson, Thomas" } ] }):
...     doc
...
...
```

基本上，通过\$or 操作符可以同时执行两个搜索，并结合它们的结果输出，即使各个搜索的结果并无交集。另外，\$or 子句将并行执行，每个子句都将使用不同的索引。

7. 使用\$slice 从数组中获取元素

可以使用\$slice 操作符从文档的指定数组中获得特定数目的元素。该操作符提供了类似于skip()和 limit()函数的功能；区别在于这两个函数都作用于整个文档，而\$slice 操作符只作用于单个文档的数组。

在学习样例之前，首先添加一个新的文档，用于帮助我们理解这个操作符。假设公司正致力于追踪所有的椅子库存以及它们的位置信息。自然，每个椅子都有它自己曾经所属于的桌子的信息。\$slice 操作符非常有利于完成此类库存追踪工作。

首先添加下面的文档:

```
>>> chair = ({
...     "Status" : "Not used",
...     "Tags" : ["Chair", "Not used", "Storage"],
...     "ItemNumber" : "6789SID",
...     "Location" : {
...     "Department" : "Storage",
...     "Building" : "2B"
...     },
...     "PreviousLocation" :
...     [ "120100", "120101", "120102", "120103", "120104", "120105",
...     "120106", "120107", "120108", "120109", "120110" ]
... })
```

```
>>> collection.insert(chair)
ObjectId('4c5973554abffe0e0c000005')
```

现在假设你已经看到了之前样例中返回的所有可用椅子的信息，不过要注意的是：你不希望看到椅子之前的所有位置信息，而是只希望看到椅子所属的前三张桌子。

```
>>> collection.find_one({'ItemNumber': '6789SID'}, {'PreviousLocation': {'$slice': 3}})
{
  u'Status': u'Not used',
  u'PreviousLocation': [u'120100', u'120101', u'120102'],
  u'Tags': [u'Chair', u'Not used', u'Storage'],
  u'ItemNumber': u'6789SID',
  u'Location': {
    u'Department': u'Storage',
    u'Building': u'2B'
  },
  u'_id': ObjectId('4c5973554abffe0e0c000005')
}
```

类似地，通过将 \$slice 的值取反，结果将显示出椅子最后所属的三张桌子：

```
>>> collection.find_one({'ItemNumber': '6789SID'}, {'PreviousLocation': {'$slice': -3}})
{
  u'Status': u'Not used',
  u'PreviousLocation': [u'120108', u'120109', u'120110'],
  u'Tags': [u'Chair', u'Not used', u'Storage'],
  u'ItemNumber': u'6789SID',
  u'Location': {
    u'Department': u'Storage',
    u'Building': u'2B'
  },
  u'_id': ObjectId('4c5973554abffe0e0c000005')
}
```

还可以忽略椅子所属的前 5 个位置，并将返回的结果数目限制为 3 个(注意这里的括号)：

```
>>> collection.find_one({'ItemNumber': '6789SID'}, {'PreviousLocation': {'$slice': [5, 3]}})
{
  u'Status': u'Not used',
  u'PreviousLocation': [u'120105', u'120106', u'120107'],
  u'Tags': [u'Chair', u'Not used', u'Storage'],
  u'ItemNumber': u'6789SID',
  u'Location': {
    u'Department': u'Storage',
    u'Building': u'2B'
  },
  u'_id': ObjectId('4c5973554abffe0e0c000005')
}
```

你应该已经明白了 \$slice 的用法。该例似乎并不常见，但库存控制系统经常会进入不正确的状态；\$slice 操作符可以帮助在不寻常或复杂的环境中执行统计工作。例如，\$slice 操作符是用于网站评论区分页实现的一个非常高效的工具。

7.5.9 使用正则表达式执行搜索

执行搜索的一个非常有用的工具是正则表达式。Python 默认的正则表达式模块被称为 re。

在使用它执行搜索之前必须先加载它，如下所示：

```
>>> import re
```

在加载该模块之后，可以在搜索条件的 `value` 字段中指定正则表达式查询。下面的样例展示了如何搜索 `ItemNumber` 值含有 4 的文档(为了保持代码简单,该例只返回了 `ItemNumber` 中的值):

```
>>> for doc in collection.find({'ItemNumber' : re.compile('4')}, {'ItemNumber' : True}):
...     doc
...
{'ItemNumber': u'1234EXD', u'_id': ObjectId('4c57207b4abffe0e0c000000')}
{'ItemNumber': u'2345FDX', u'_id': ObjectId('4c57234c4abffe0e0c000001')}
{'ItemNumber': u'2345FDX', u'_id': ObjectId('4c592eb84abffe0e0c000004')}
{'ItemNumber': u'3456TFS', u'_id': ObjectId('4c57234c4abffe0e0c000002')}
```

更进一步，还可以定义正则表达式。此时的查询是大小写敏感的，并且它将匹配任何 `ItemNumber` 值中含有 4 的文档,无论 4 出现在什么位置。不过,假设你希望搜索一个 `ItemNumber` 值以 FS 结尾的文档,FS 之前可以是任意值,FS 之后不能含有任何数据,那么可以使用下面的代码:

```
>>> for doc in collection.find({'ItemNumber' : re.compile('.FS$')},fields={'ItemNumber' :
True}):
...     doc
...
{'ItemNumber': u'3456TFS', u'_id': ObjectId('4c57234c4abffe0e0c000002')}
```

还可以使用 `find()`以大小写不敏感的方式搜索信息,但首先必须添加另一个函数,如下所示:

```
>>> for doc in collection.find({'Location.Owner' : re.compile('^anderson. ',
re.IGNORECASE)},
... fields={'ItemNumber' : True, 'Location.Owner' : True}):
...     doc
...
{
  u'ItemNumber': u'1234EXD',
  u'_id': ObjectId('4c57207b4abffe0e0c000000'),
  u'Location': {
    u'Owner': u'Anderson, Thomas'
  }
}
```

只要使用方式正确,正则表达式将会是非常强大的工具。关于 `re` 模块如何工作的更多细节及其包含的函数,请参阅该模块的官方文档: <http://docs.python.org/library/re.html>。

7.6 修改数据

到目前为止,我们已经学习了在 Python 中如何使用条件操作符和正则表达式查询数据库中的信息。在下一节中,我们将学习如何使用 Python 修改集合中已有的数据。在 Python 中可以通过几种不同的方式完成该操作。接下来将使用之前学习的操作符匹配即将被修改的文档。在某些情况下,你可能需要返回到本章之前的内容中,再次学习如何使用某些查询操作符,这是正常学习过程中的一部分,通过这种方式你将再次加强之前已经学过的知识。

7.6.1 更新数据

使用 Python `update()` 函数的方式基本上与 MongoDB shell 和 PHP 驱动中该函数的使用方式相同。在 Python 中，为了更新数据，需要为该函数提供两个参数：`arg` 和 `doc`。参数 `arg` 指定了用于匹配文档的键值信息，参数 `doc` 则包含了更新后的信息。还可以提供几种不同的可选参数，用于指定选项。下面的列表显示了可用于更新信息的选项以及它们的用途：

- `upsert`(可选的)：如果该布尔选项被设置为真，那么如果搜索条件未被匹配，它就会创建一个新的文档。
- `manipulate`(可选的)：如果设置为真，表示在对该文档执行更新之前，所有的 `SONManipulator` 实例都将对它进行操作。关于它的更多信息，请参阅 `son_manipulator` 的文档(http://api.mongodb.org/python/current/api/pymongo/son_manipulator.html)。
- `check_keys`(可选的)：如果设置为真，`update()` 将在替换 `arg` 时检查文档中的键是否以受限的字符 '\$' 或 '.' 开头。
- `multi`(可选的)：如果该布尔选项被设置为真，那么匹配该搜索条件的所有文档都将被更新，而不只是第一个文档被更新(默认行为)。因此推荐总是将该值设置为真或假，而不是依赖于它的默认行为(这在将来可能会改变)。
- `w`(可选的)：如果设置为 0，更新操作将不会得到确认。在使用复制集时，可以将它设置为 `n`，保证主服务器在将修改复制到 `n` 个节点后才确认该更新操作。还可以设置为 'majority'——一个保留字符串——用于保证大多数复制节点或特定标签中的节点都确认该更新操作。该选项默认值为 1，表示确认更新操作。
- `j`(可选的)：如果该布尔选项被设置为真，那么数据将在更新结果返回之前写入到日志中。默认值为假。
- `fsync`(可选的)：如果该布尔选项被设置为真，那么数据将在更新结果返回前同步到磁盘。如果该选项被设置为真，那么即使选项 `w` 已经设置了其他值，也会把选项 `w` 设置为 0。

在更新文档时，如何不指定任何修改操作符，那么默认文档中的所有信息都将被 `doc` 参数中的数据所替换。因此我们应该避免依赖于默认行为；应该使用之前提到的操作符，以显式地指定希望执行的更新(稍后会讲解如何做到这一点)。

下面是一个未使用任何条件操作符的样例，通过它你可以看到为什么最好在 `update()` 命令中使用条件操作符：

```
//定义更新数据
>>> update = ( {
    "Type" : "Chair",
    "Status" : "In use",
    "Tags" : ["Chair", "In use", "Marketing"],
    "ItemNumber" : "6789SID",
    "Location" : {
        "Department" : "Marketing",
        "Building" : "2B",
        "DeskNumber" : 131131,
        "Owner" : "Martin, Lisa"
    }
} )
// 现在执行更新
>>> collection.update({"ItemNumber" : "6789SID"}, update)
```



```
// 检查更新结果
>>> collection.find_one({"Type" : "Chair"})
{
  u'Status': u'In use',
  u'Tags': [u'Chair', u'In use', u'Marketing'],
  u'ItemNumber': u'6789SID',
  u'Location': {
    u'Department': u'Marketing',
    u'Building': u'2B',
    u'DeskNumber': 131131,
    u'Owner': u'Martin, Lisa'
  },
  u'_id': ObjectId('4c5973554abffe0e0c000005'),
  u'Type': u'Chair'
}
```

该例最大的缺点是：代码有点长并且只更新了几个字段。接下来，我们将学习如何使用修改操作符完成该操作。

7.6.2 修改操作符

第 4 章详细讲解了如何在 MongoDB shell 中使用它所提供的大量修改操作符，通过它们可以轻松的操作数据，而不需要通过重写整个文档来修改某个字段的值(如同之前样例所做的一样)。

通过修改操作符可以完成所有操作，包括修改文档的某个值、插入整个数组或者根据数组中指定的多个元素删除所有匹配的文档。通过使用它们可以轻松地完成数据修改。现在我们开始学习这些操作符及其用法。

1. 使用\$inc 增加整数值

通过\$inc 可以将文档中的某个整数值增加 n。下面的样例演示了如何将 Location.Desknumber 增加 20:

```
>>> collection.update({"ItemNumber" : "6789SID"}, {"$inc" : {"Location.DeskNumber" : 20}})
```

接下来，检查该更新操作的执行结果:

```
>>> collection.find_one({"Type" : "Chair"}, fields={"Location" : "True"})
{
  u'_id': ObjectId('4c5973554abffe0e0c000005'),
  u'Location': {
    u'Department': u'Marketing',
    u'Building': u'2B',
    u'Owner': u'Martin, Lisa',
    u'DeskNumber': 131151
  }
}
```

注意，\$inc 操作符只能作用于整数值，而不能作用于任何字符串或者作为字符串添加的数值(例如"123"和 123)。

2. 使用\$set 修改现有值

通过\$set 操作符可以修改所有匹配文档中的现有值。这是一个常用操作符。下一个样例将

修改任何匹配键/值"Location.Department/Development"的文档中的“Building”字段值。

首先使用\$set 执行更新，保证所有文档都被更新：

```
>>> collection.update({"Location.Department" : "Development"},
... {"$set" : {"Location.Building" : "3B"} },
... multi = True )
```

接下来，使用 find_one()命令确认结果正确：

```
>>> collection.find_one({"Location.Department" : "Development"}, fields={"Location.
Building" :True})
{
  u'_id': ObjectId('4c57207b4abffe0e0c000000'),
  u'Location': {u'Building': u'3B'}
}
```

3. 使用\$unset 移除键/值字段

同样，通过使用\$unset 操作符可以移除文档中的键/值字段，如下所示：

```
>>> collection.update({"Status" : "Not used", "ItemNumber" : "2345FDX"},
... {"$unset" : {"Location.Building" : 1 } } )
```

接下来，使用 find_one()命令确认结果正确：

```
>>> collection.find_one({"Status" : "Not used", "ItemNumber" : "2345FDX"},
fields={"Location" :True})
{
  u'_id': ObjectId('4c592eb84abffe0e0c000004'),
  u'Location': {u'Department': u'Storage'}
}
```

4. 使用\$push 向数组中添加值

在数组存在的情况下，通过\$push 可以在数组中添加值。如果数组不存在，那么它将使用指定的值创建该数组。

警告：

如果使用\$push 更新非数组字段，那么将会出现错误消息。

现在开始向已经存在的数组中添加值，并确认结果正确。首先，执行更新：

```
>>> collection.update({"Location.Owner" : "Anderson, Thomas"},
... {"$push" : {"Tags" : "Anderson"} }, multi = True )
```

现在执行 find_one()以确认更新正确执行：

```
>>> collection.find_one({"Location.Owner" : "Anderson, Thomas"}, fields={"Tags" : "True"})
{
  u'_id': ObjectId('4c57207b4abffe0e0c000000'),
  u'Tags': [u'Laptop', u'Development', u'In Use', u'Anderson']
}
```

5. 使用\$push 和\$each 向数组中添加多个值

操作符\$push 还可用于一次向现有数组中添加多个值，这可以通过添加\$each 修改符来实现。

这里的规则是相同的：数组必须已经存在，否则将会出现错误。下面的样例将结合使用\$each修改符和\$正则表达式执行搜索；通过它们可以将修改应用到所有匹配的查询结果中：

```
>>> collection.update({ "Location.Owner" : re.compile("^Walker,") },
... { '$push' : { 'Tags' : { '$each' : ['Walker','Warranty'] } } })
```

接下来，执行 find_one() 以确认执行结果是否正确：

```
>>> collection.find_one({"Location.Owner" : re.compile("^Walker,")}, fields=
{"Tags" : True})
{
  u'_id': ObjectId('4c57234c4abffe0e0c000002'),
  u'Tags': [u'Laptop', u'Development', u'In Use', u'Walker', u'Warranty']
}
```

6. 使用\$addToSet 向现有数组中添加值

通过\$addToSet也可以在现有数组中添加值。区别在于该方法在尝试更新之前，将检查该数组是否已经存在(\$push操作符不会进行检查)。

该操作符只接受一个额外值；不过，\$addToSet也可以与\$each操作符结合使用。下面通过两个例子演示它的用法。首先，对所有匹配"Type": "Chair"的对象使用\$addToSet操作符进行更新，然后使用 find_one()函数检查结果是否正确：

```
>>> collection.update({"Type" : "Chair"}, {"$addToSet" : {"Tags" : "Warranty"}}, multi = True)
>>> collection.find_one({"Type" : "Chair"}, {"Tags" : "True"})
{
  u'_id': ObjectId('4c5973554abffe0e0c000005'),
  u'Tags': [u'Chair', u'In use', u'Marketing', u'Warranty']
}
```

还可以使用\$each语句添加多个标签。注意，该搜索将通过使用正则表达式完成。另外，列表中的其中一个标签之前已经添加过；幸运的是，它不会再次被添加，因为\$addToSet将避免这种情况的发生：

```
// Use the $each operator to add multiple tags, including one that was already added
>>> collection.update({"Type" : "Chair", "Location.Owner" : re.compile("^Martin,")},
... {"$addToSet" : { "Tags" : { "$each" : ["Martin","Warranty", "Chair", "In use"] } } })
```

现在检查执行结果是否正确；尤其是，检查重复的Warranty标签是否被再次添加：

```
>>> collection.find_one({"Type" : "Chair", "Location.Owner" : re.compile("^Martin,")},
fields={"Tags" : True})
{
  u'_id': ObjectId('4c5973554abffe0e0c000005'),
  u'Tags': [u'Chair', u'In use', u'Marketing', u'Warranty', u'Martin']
}
```

7. 使用\$pop 从数组中删除元素

到目前为止，你已经学习如何使用update()函数向已有的文档中添加值。现在学习如何删除数据。首先从\$pop操作符开始。

通过该操作符可以从数组中删除第一个或最后一个元素，而不是其中的任何其他元素。下

面将从匹配"Type": "Chair"的第一个文档的 Tags 数组中删除第一个元素；然后使用 `find_one()` 命令确认该更新执行正确：

```
>>> collection.update({"Type" : "Chair"}, {"$pop" : {"Tags" : -1}})
>>> collection.find_one({"Type" : "Chair"}, fields={"Tags" : True})
{
  u'_id': ObjectId('4c5973554abffe0e0c000005'),
  u'Tags': [u'In use', u'Marketing', u'Warranty', u'Martin']
}
```

下面的样例将使用正值，用于删除数组中的最后一个值：

```
>>> collection.update({"Type" : "Chair"}, {"$pop" : {"Tags" : 1}})
```

接下来，再次执行 `find_one()` 以确认更新结果：

```
>>> collection.find_one({"Type" : "Chair"}, fields={"Tags" : True})
{
  u'_id': ObjectId('4c5973554abffe0e0c000005'),
  u'Tags': [u'In use', u'Marketing', u'Warranty']
}
```

8. 使用 \$pull 删除特定的值

通过 \$pull 操作符可以从数组中删除特定值的所有实例，不管该值出现多少次；只要数组中的值与指定值相同就会被删除。

下面的样例将使用 \$push 操作符在 Tags 数组中添加相同的值 Double：

```
>>> collection.update({"Type" : "Chair"}, {"$push" : {"Tags" : "Double"}}, multi = False)
>>> collection.update({"Type" : "Chair"}, {"$push" : {"Tags" : "Double"}}, multi = False)
```

接下来，通过执行 `find_one()` 命令确保该标签被添加了两次。一旦确认该标签被添加两次，就使用 \$pull 操作符删除该标签的所有实例：

```
>>> collection.find_one({"Type" : "Chair"}, fields={"Tags" : True})
{
  u'_id': ObjectId('4c5973554abffe0e0c000005'),
  u'Tags': [u'In use', u'Marketing', u'Warranty', u'Double', u'Double']
}
```

```
>>> collection.update({"Type" : "Chair"}, {"$pull" : {"Tags" : "Double"}}, multi = False)
```

再次执行 `find_one()` 命令，确认更新结果执行成功，这次结果中一定不会再出现 Double 标签：

```
>>> collection.find_one({"Type" : "Chair"}, fields={"Tags" : True})
{
  u'_id': ObjectId('4c5973554abffe0e0c000005'),
  u'Tags': [u'In use', u'Marketing', u'Warranty']
}
```

为了实现该操作，还可以使用 \$pullAll 操作符；它们的区别在于通过 \$pullAll 可以同时删除多个标签。接下来通过样例演示它们的区别。首先，在 Tags 数组中添加多个元素，然后确认它们已经添加成功：

```
>>> collection.update({"Type" : "Chair"}, {"$addToSet" : { "Tags" : {"$each" : ["Bacon",
"Spam"]} } }
})
>>> collection.find_one({"Type" : "Chair"}, fields={"Tags" : True})
{
  u'_id': ObjectId('4c5973554abffe0e0c000005'),
  u'Tags': [u'In use', u'Marketing', u'Warranty', u'Bacon', u'Spam']
}
```

现在开始使用 \$pullAll 操作符删除多个标签。下面的样例将演示如何使用该操作符；之后该样例还执行了 find_one() 命令，以确认 Bacon 和 Spam 标签已经被移除：

```
>>> collection.update({"Type" : "Chair"}, {"$pullAll" : {"Tags" : ["Bacon", "Spam"]} },
multi = False)
>>> collection.find_one({"Type" : "Chair"}, fields={"Tags" : "True"})
{
  u'_id': ObjectId('4c5973554abffe0e0c000005'),
  u'Tags': [u'In use', u'Marketing', u'Warranty']
}
```

7.6.3 使用 save() 函数快速保存文档

可以使用 save() 函数，再通过 upsert 方法快速地添加文档。为了实现该操作，首先必须定义 _id 字段的值。如果希望保存的文档已经存在，那么该文档将被更新；如果不存在，该文档将被创建。

下面的样例将保存一个名为 Desktop 的文档。首先在 shell 中输入一个含有标识符的文档，然后使用 save() 函数保存它。一旦 save() 函数执行成功，它将返回一个 ObjectId：

```
>>> Desktop = ( {
  "Status" : "In use",
  "Tags" : ["Desktop", "In use", "Marketing", "Warranty"],
  "ItemNumber" : "4532FOO",
  "Location" : {
    "Department" : "Marketing",
    "Building" : "2B",
    "Desknumber" : 131131,
    "Owner" : "Martin, Lisa",
  }
} )
>>> collection.save(Desktop)
ObjectId('4c5ddb24abffe0f34000001')
```

现在，假设你已经意识到自己忘记在文档中指定某个键/值对。为了将该信息添加到文档中，首先需要定义文档名称，然后在括号中添加它的键，接着添加希望保存的内容。完成之后，可以通过保存整个文档执行 upsert 操作；该操作完成之后将再次返回该文档的 ObjectId：

```
>>> Desktop[ "Type" ] = "Desktop"
>>> collection.save(Desktop)
ObjectId('4c5ddb24abffe0f34000001')
```

如你所见，返回的 ObjectId 值并未改变。

7.6.4 以原子的方式修改文档

通过 `find_and_modify()` 函数能够以原子的方式修改文档并返回结果。该函数只可用于更新单个文档。记住，该函数默认返回的文档中不会包含被修改的内容；如果需要获取修改的内容，那么需要指定一个额外的参数 `new:True`。

函数 `find_and_modify()` 中可使用多个参数，并且必须包含更新参数或删除参数。下面的列表中列出了所有可用的参数，以及它们的定义和用法：

- `query`: 指定查询使用的过滤器。如果未指定该参数，那么集合中所有的文档都将被认为是目标，并且搜索到的第一个文档将被更新或删除。
- `update`: 指定用于更新文档的信息。注意，之前指定的任何修改操作符都可用在该参数中。
- `upsert`: 如果设置为真，执行更新插入操作。
- `sort`: 以特定的顺序对匹配文档进行排序。
- `full_response`: 如果设置为真，那么结果将返回完整的响应对象。
- `remove`: 如果设置为真，那么所匹配的文档将被删除。
- `new`: 如果设置为真，更新后的文档将被返回，而不是返回选择的文档。注意，该参数默认未被设置，在某些环境中这可能会引起混淆。
- `fields`: 指定希望返回的字段，而不是整个文档。该参数的行为与 `find()` 函数中的 `fields` 参数一致。注意：`_id` 字段总是会被返回，即使该字段并未在字段列表中指定。

7.6.5 使用参数

在学习了这些参数的用法之后；现在，在真实世界的样例中使用 `find_and_modify()` 函数。首先使用 `find_and_modify()` 搜索任何键值对等于 `"Type": "Desktop"` 的文档，然后通过设置额外的键值对 `"Status": "In repair"` 对匹配查询的所有文档进行更新。最后，确保 MongoDB 返回更新后的文档，而不是返回匹配查询的旧文档：

```
>>> collection.find_and_modify(query={"Type" : "Desktop"},
... update={'$set' : {'Status' : 'In repair'} }, new=True)
{
  u'ok': 1.0,
  u'value': {
    u'Status': u'In repair',
    u'Tags': [u'Desktop', u'In use', u'Marketing', u'Warranty'],
    u'ItemNumber': u'4532FOO',
    u'Location': {
      u'Department': u'Marketing',
      u'Building': u'2B',
      u'Owner': u'Martin, Lisa',
      u'Desknumber': 131131
    },
    u'_id': ObjectId('4c5dda114abffe0f34000000'),
    u'Type': u'Desktop'
  }
}
```

下面的例子将使用 `find_and_modify()` 删除文档；在该例中，结果将返回被删除的文档：

```
>>> collection.find_and_modify(query={'Type' : 'Desktop'},
... sort={'ItemNumber' : -1}, remove=True){
```

```

u'ok': 1.0,
u'value': {
  u'Status': u'In use',
  u'Tags': [u'Desktop', u'In use', u'Marketing', u'Warranty'],
  u'ItemNumber': u'4532FOO',
  u'Location': {
    u'Department': u'Marketing',
    u'Building': u'2B',
    u'Owner': u'Martin, Lisa',
    u'Desknumber': 131131
  },
  u'_id': ObjectId('4c5ddb24abffe0f34000001'),
  u'Type': u'Desktop'
}
}

```

7.7 删除数据

在多数情况下，你都只是通过 PHP 驱动添加或修改数据。不过，了解如何删除数据也很重要。Python 驱动提供了几种删除数据的方法。首先，可以使用 `remove()` 函数从集合中删除单个文档。其次，可以使用 `drop()` 或 `drop_collection()` 函数删除整个集合。最后，可以使用 `drop_database()` 函数删除整个数据库(你似乎不可能经常使用该函数！)。

下面将通过样例演示每个函数的用法。

首先学习 `remove()` 函数。通过该函数可以从当前集合中删除所有匹配参数条件的文档。下面的样例将使用 `remove()` 函数删除所有含有键/值对 "Status": "In use" 的文档；最后，使用 `find_one()` 函数确认结果：

```

>>> collection.remove({"Status" : "In use"})
>>> collection.find_one({"Status" : "In use"})
>>>

```

要小心该函数中使用的搜索条件类型。通常，应该先执行 `find()` 以检查将要被删除的文档。也可以通过使用 `ObjectId` 删除数据。

如果希望删除整个集合，可以使用 `drop()` 或 `drop_collection()` 函数。这两个函数的工作方式相同(其中一个只是另一个的别名)；尤其是，它们都只接受一个参数：集合名称。

```

>>> db.items.drop()

```

最后(也是最重要的，因为可能会毁坏整个数据库)，通过 `drop_database()` 函数可以删除整个数据库。通过使用 `Connection` 模块可以调用该函数，如下所示：

```

>>> c.drop_database("inventory")

```

7.8 在两个文档之间创建链接

在 PyMongo 中，数据库引用(DBRef)是通过 DBRef 模块中的 `DBRef()` 函数实现的，它可在处于不同位置的两个文档之间创建链接。这是一种存储字段的惯例，可用于应用逻辑中。例如，可以为所有员工创建一个集合，然后将其他相关信息添加到另一个集合中，再使用 `DBRef()` 函数在

员工和位置所在的文档之间创建引用(而不是为每个文档手动输入这些信息)。

注意:

MongoDB 不保证 DBRef 是有效的。因此对于可能被 DBRef 引用的文档,在删除时一定要小心。

正如之前章节中所讲解的,可以通过两种方式引用数据。第一种方式是使用简单引用,在一个文档中使用 `_id` 字段引用另一个文档。第二种方式是使用 DBRef 模块,与手动引用相比,这种方式提供了更多的选项。

接下来先演示如何创建手动索引。首先保存一个文档。例如,假设希望将人员的信息保存在特定的集合中。下面的样例定义了一个字典 `jan`,并将它保存在 `people` 集合中,然后返回它的 `ObjectId`:

```
>>> jan = {
...     "First Name" : "Jan",
...     "Last Name" : "Walker",
...     "Display Name" : "Walker, Jan",
...     "Department" : "Development",
...     "Building" : "2B",
...     "Floor" : 12,
...     "Desk" : 120103,
...     "E-Mail" : "jw@example.com"
... }

>>> people = db.people
>>> people.insert(jan)
ObjectId('4c5e5f104abffe0f34000002')
```

在添加文档并获得它的 ID 之后,可以使用该信息在另一个集合的文档中链接到这个文档:

```
>>> laptop = {
...     "Type" : "Laptop",
...     "Status" : "In use",
...     "ItemNumber" : "12345ABC",
...     "Tags" : ["Warranty","In use","Laptop"],
...     "Owner" : jan[ "_id" ]
... }

>>> items = db.items
>>> items.insert(laptop)
ObjectId('4c5e6f6b4abffe0f34000003')
```

现在假设希望搜索所有者信息。在这种情况下,所有需要做的就是查询 `Owner` 字段中提供的 `ObjectId`;当然,只有在知道数据存储存储在哪个集合中时,才能这样做。

现在假设你不知道信息存储的位置。`DBRef()`函数正是为了解决这样的问题才出现的。你可以在不知道哪个集合存储原始数据的情况下使用该函数,并且在搜索信息的时候也不需要担心集合名称。

`DBRef()`函数接受 3 个参数;还可以接受用于指定额外关键字参数的第 4 个参数。下面列出了这 3 个主要的参数以及它们的用途:

- `collection`(必选的): 指定原始数据所在的集合(例如 `people`)。
- `id`(必选的): 指定所引用文档的 `_id` 值。

- `database`(可选的): 指定所引用数据库的名字。
在使用 `DBRef` 函数之前, 必须先加载 `DBRef` 模块:

```
>>> from bson.dbref import DBRef
```

现在开始学习 `DBRef()` 函数的一个实际例子。下面的样例将在 `people` 集合中插入一个人员的信息, 并在 `items` 集合中添加一个元素, 再使用 `DBRef` 引用所有者:

```
>>> mike = {
...     "First Name" : "Mike",
...     "Last Name" : "Wazowski",
...     "Display Name" : "Wazowski, Mike",
...     "Department" : "Entertainment",
...     "Building" : "2B",
...     "Floor" : 10,
...     "Desk" : 120789,
...     "E-Mail" : "mw@monsters.inc"
... }
```

```
>>> people.save(mike)
ObjectId('4c5e73714abffe0f34000004')
```

此时, 一切看起来都很平常。样例中添加了一个文档, 但并未添加任何引用。不过, 此时已经得到了文档的 `ObjectId`, 所以现在可以在集合中添加另一个文档, 然后使用 `DBRef()` 指向之前插入文档的 `owner` 字段。注意 `DBRef()` 函数的语法; 尤其要注意的是, 这里给出的第一个参数是之前指定文档所在的集合名, 而第二个参数不过是 `mike` 词典中的 `_id` 键的引用:

```
>>> laptop = {
...     "Type" : "Laptop",
...     "Status" : "In use",
...     "ItemNumber" : "2345DEF",
...     "Tags" : ["Warranty", "In use", "Laptop"],
...     "Owner" : DBRef('people', mike[ "_id" ])
... }
```

```
>>> items.save(laptop)
ObjectId('4c5e740a4abffe0f34000005')
```

你可能会注意到, 上述代码与之前创建手动引用的代码并没有太大区别。不过, 我们推荐在需要引用特定信息的时候使用 `DBRef`, 而不是内嵌信息。采用这种方式也意味着无论何时查询引用信息, 都不需要再查询集合的名称。

获取信息

在学习如何使用 `DBRef()` 引用信息之后; 现在假设希望获取之前引用的信息。可以使用 Python 驱动的 `dereference()` 函数来完成。所有需要做的是, 将之前定义的包含了引用信息的字段作为参数, 然后按下回车键。

下面通过一个样例, 演示如何使用该函数从一个文档引用并获取另一个文档的信息。首先搜索含有引用数据的文档, 然后获取被引用的文档。第一步创建一个查询, 用于搜索含有引用信息的随机文档:

```
>>> items.find_one({"ItemNumber" : "2345DEF"})
```

```
{
  u'Status': u'In use',
  u'Tags': [u'Warranty', u'In use', u'Laptop'],
  u'ItemNumber': u'2345DEF',
  u'Owner': DBRef(u'people', ObjectId('4c5e73714abffe0f34000004')),
  u'_id': ObjectId('4c5e740a4abffe0f34000005'),
  u'Type': u'Laptop'
}
```

接下来，将数据存储在 `person` 词典中：

```
>>> person = items.find_one({"ItemNumber" : "2345DEF"})
```

此时，可以使用 `dereference()` 函数解析参数 `person["Owner"]` 所引用的 `Owner` 字段 (`Owner` 字段被链接到希望获取的数据)：

```
>>> db.dereference(person["Owner"])
```

```
{
  u'Building': u'2B',
  u'Floor': 10,
  u'Last Name': u'Wazowski',
  u'Desk': 120789,
  u'E-Mail': u'mw@monsters.inc',
  u'First Name': u'Mike',
  u'Display Name': u'Wazowski, Mike',
  u'Department': u'Entertainment',
  u'_id': ObjectId('4c5e73714abffe0f34000004')
}
```

这看起来并不糟！该例表达的观点是：`DBRef` 技术提供了一种存储引用数据的好方法，另外，在指定集合名称和数据库名称时，也提供了一定的灵活性。如果希望保持数据库整洁，那你将会经常使用该特性，尤其是在不能使用嵌入数据的情况下。

7.9 小结

本章讲解了 MongoDB Python 驱动 (PyMongo) 的常用操作的基础知识。另外，还讲解了如何搜索、存储、更新和删除数据。

我们还学习了引用另一个集合中文档的两种方法：手动引用和 `DBRef`。它们的使用方式非常相似，但 `DBRef` 方式在功能方面更加强大，所以在大多数情况下都推荐使用 `DBRef`。

下一章将详细讲解 MongoDB 的更多高级查询方法。

高级查询

在之前的章节中，我们已经学习了大多数的基础查询机制：通过指定的条件搜索一个或一系列文档。有许多机制可用于搜索指定的文档，并将结果返回到应用中进行处理。但如果你希望在集合中的大多数或所有文档中执行复杂操作，那么普通的查询机制可能会显得不足。当需要使用此类查询或操作时，许多开发者会考虑遍历集合中所有的文档或者编写一系列查询，并按顺序执行的方式执行必要的计算。尽管这是一种有效的方式，但可能难于编写和维护，并且效率不高。出于这些原因，MongoDB 提供了一些高级查询机制，用于解决这种处理大多数数据的问题。本章将要讲解的高级 MongoDB 特性是全文搜索、聚集框架和 MapReduce 框架。

全文索引是希望被添加到 MongoDB 中的呼声最高的一个特性。它代表了在 MongoDB 中创建专有文本索引的能力，通过这些索引可以执行文本搜索，从而定位到包含了匹配文本元素的文档。MongoDB 全文搜索特性比简单的字符串匹配强大得多，它将基于为文档选择的语言以全词干(full-stemmed)的方式创建出索引，它是基于文档执行语言查询的一个极其强大的工具。这个最近引入的特性被标志为 MongoDB 2.4 版本的“实验”特性，因为开发团队仍然在努力改进它，这意味着为了在 MongoDB 环境中使用全文索引，必须先激活它。

本章将要讲解的第二个特性是 MongoDB 聚集框架。第 4 和第 6 章中曾介绍过该特性，它提供了许多查询特性，可用于遍历所选择的文档或所有文档，并收集或修改信息。这些查询函数将按照管道操作的方式依次在集合中执行，并从查询中收集信息。

本章将要讲解的第三个，也是最后一个特性被称作 MapReduce，它听起来类似于 Hadoop 所采用的方式。MapReduce 是一个强大的机制，可利用 MongoDB 内置的 JavaScript 引擎，以实时的方式执行抽象代码。它是一个极其强大的工具，通过使用两个 JavaScript 函数实现映射规约操作，一个用于映射数据，另一个用于转换并从映射数据中取出信息。

学习本章内容最需要记住的就是：这些是真正的高级特性，如果误用它们，那么可能会在 MongoDB 节点中引发严重的性能问题。所以，在将这些特性部署到重要的系统之前，一定要在测试环境中进行测试。

8.1 文本搜索

MongoDB 文本搜索的工作方式为：首先创建全文索引，然后指定希望索引的字段(用于执行全文搜索)。文本索引将遍历集合中的所有文档，并分析出每个文本字符串中的标记和词干。这种标记和词干解析的过程将把文本分割成记号，在概念上它们接近于单词。接着 MongoDB 将取得每个记号的词干，然后找到该记号的根词汇。例如，假设将字符串分割后得到了记号 fishing。然后解析出该记号的词干，得到根词汇 fish，MongoDB 将为该文档创建一个索引条目

fish。对于用户输入的用于执行搜索的文本，同样会进行这样的标记和词干解析。然后 MongoDB 会将该参数与每个文档进行比较，并计算出相对得分。最后基于它们的得分将文档返回给用户。

你可能会好奇像 the 或 it 这样的单词如何抽取词干，对于非英语的文档又应该如何处理。答案是与它们类似的单词都不会被抽取词干，另外 MongoDB 的文本搜索引擎将支持多种语言。

MongoDB 的文本搜索引擎是由 MongoDB Inc. 为文本数据处理而编写的自有引擎。MongoDB 文本索引还利用了 Snowball 的字符串处理语言，该语言提供了对单词词干抽取和终止词(这些单词都不会被抽取词干，因为它们并不代表索引或搜索中的任何有价值的概念)的支持。

需要注意的是 MongoDB 的文本搜索极其复杂，但它已经被设计得尽可能灵活和准确。

8.1.1 文本搜索的代价和限制

根据之前学习的文本搜索的工作方式，你可以想象使用 MongoDB 文本索引时所需付出的一些代价。首先是它将使用 usePowerOf2Sizes 选项改变未来文档的存储分配，该选项将指示 MongoDB 为了更高效地重用空闲空间而分配存储。其次，文本索引非常大，并且可能增长非常迅速，这取决于所存储的文档数目以及每个被索引字段中的记号数目。第 3 个限制是基于现有文档构建文本索引非常耗时，在一个已经具有文本索引的字段中添加新的条目，代价就更加昂贵。第 4，如同 MongoDB 中的所有其他特性一样，文本索引在 RAM 中工作更好。最后，由于文本索引的复杂性和大小，最多只能为每个集合创建一个文本索引。

8.1.2 启用文本索引

正如之前提到的，文本索引在 MongoDB 2.4 中作为实验或测试特性引入。因此，需要在所有集群中，在将要使用该特性的 MongoDB 实例中显式启用文本搜索功能(如果正在使用分片的话，也需要在 MongoS 中启用)。有三种方式可以启用文本索引；第一种是在启动或停止 MongoDB 进程的命令中添加下面的选项：

```
--setParameter textSearchEnabled=true
```

第二种方式是在 MongoDB 实例的配置文件中添加下面的选项：

```
setParameter = textSearchEnabled=true
```

第三种也是最后一种方式，是在 Mongo shell 中运行下面的命令：

```
db.adminCommand({ setParameter: 1, textSearchEnabled : true })
```

运行该命令之后，就可以在该节点上使用 MongoDB 的全文搜索特性了。

注意：

事实上，该特性是一个测试特性并不意味着它无法正常工作。MongoDB Inc. 团队已经在该特性上花费了大量的精力来保证它能正确运行。通过使用该特性并向 MongoDB Inc. 的 JIRA 中报告发现的问题，你可以帮助他们完善该特性并最终正式发布该特性。

到目前为止，我们已经在 MongoDB 实例中启用了全文搜索特性，并做好了使用它的准备！下面让我们创建一个文本搜索索引并执行文本搜索。

8.1.3 使用文本索引

尽管文本搜索非常复杂，但 MongoDB 文本搜索也非常易用；创建文本索引的方式与创建其他索引的方式相同。例如，如果需要在博客集合的“content”元素上创建文本索引，那么可执行下面的命令：

```
db.blog.ensureIndex( { content : "text" } );
```

这就是全部命令。MongoDB 将处理剩下的工作，并在数据库中插入文本索引。将来所有含有内容字段的文档都将被处理，并且在文本索引中添加用于搜索的条目。但事实上，刚才创建的索引并没有足够的数据可供使用。我们需要一组合适的文本数据用于处理和查询。

1. 加载文本数据

最初，我们计划使用来自 twitter 的在线流数据，但这些文本过于难用。所以我们决定创建一小批数据(8 个文档)来模拟 twitter 的数据，用于练习文本搜索。

接下来使用 mongoimport 将 twitter.tgz 中的数据导入到 MongoDB 数据库中：

```
$ mongoimporttest.json -d test -c texttest
connected to: 127.0.0.1
Sat Jul 6 17:52:19 imported 8 objects
```

现在数据已经准备就绪，接下来启用文本索引：

```
db.adminCommand( { setParameter: 1, textSearchEnabled : true } );
{ "was" : false, "ok" : 1 }
```

现在文本索引已经启用，接下来基于 twitter 的数据创建文本索引。

2. 创建文本索引

对于 twitter 数据，我们关心的是它的 text 字段，该字段是 tweet 的正文。运行下面的命令以创建文本索引：

```
use test;
db.texttest.ensureIndex( { body : "text" } );
```

如果遇到错误消息“text search not enabled”，那么需要通过运行刚才的命令，确认文本索引已经在运行。现在，如果在日志中看到下面的内容，那就说明 MongoDB 正在构建文本索引：

```
Sat Jul 6 17:54:16.078 [conn41] build index test.texttest{ _fts: "text", _ftsx: 1 }
Sat Jul 6 17:54:16.089 [conn41] build index done. scanned 8 total records. 0.01 secs
```

还可以检查集合的索引：

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

```
    },  
    {  
      "v" : 1,  
      "key" : {  
        "_fts" : "text",  
        "_ftsx" : 1  
      },  
      "ns" : "test.textttest",  
      "name" : "body_text",  
      "weights" : {  
        "body" : 1  
      },  
      "default_language" : "english",  
      "language_override" : "language",  
      "textIndexVersion" : 1  
    }  
  ]  
}
```

现在文本搜索已经启用，并且创建了索引，还确认了索引已经创建成功；接下运行文本搜索命令。

3. 运行文本搜索命令

我们目前使用的 MongoDB 版本中的 shell 并未提供专门的文本命令，所以需要通过执行 `runCommand` 来实现，如下所示：

```
> db.textttest.runCommand( "text", { search : "fish" } )
```

该命令将返回所有匹配查询字符串“fish”的文档。在本例中，它将返回两个文档。输出中显示了许多调试信息以及最终的“results”数组；并且还包含了文档的数目。在这些信息中还包含匹配文档和返回文档的得分，匹配的文档被显示为 `obj`。你会看到包含单词 `fish` 或 `fishing` 的文档都将匹配我们的查询！另外 MongoDB 文本索引忽略大小写，在执行文本查询时一定要注意这一点。

注意：

记住文本搜索中的所有条目都将被分词和抽取词干。这意味着对于 `fishy` 或 `fishing` 这样的单词，最终得到的都是词干 `fish`。

另外，从结果中可以看到得分为 0.75 或 0.666，这表示查询结果的相关性——值越大，相关性就越大。结果中还显示了查询的统计信息，包括返回对象的数目(2)和花费的时间(112 毫秒)。

```
{  
  "queryDebugString" : "fish|||||",  
  "language" : "english",  
  "results" : [  
    {  
      "score" : 0.75,  
      "obj" : {  
        "_id" : ObjectId("51d7ccb36bc6f959debe5514"),  
        "number" : 1,  
        "body" : "i like fish",  
        "about" : "food"  
      }  
    }  
  ]  
}
```

```

    },
    {
      "score" : 0.6666666666666666,
      "obj" : {
        "_id" : ObjectId("51d7ccb36bc6f959debe5516"),
        "number" : 3,
        "body" : "i like to go fishing",
        "about" : "recreation"
      }
    }
  ],
  "stats" : {
    "nscanned" : 2,
    "nscannedObjects" : 0,
    "n" : 2,
    "nfound" : 2,
    "timeMicros" : 112
  },
  "ok" : 1
}

```

接下来学习一些其他的可用于增强查询的文本搜索选项。

4. 过滤文本查询结果

首先可以对文本查询结果进行过滤。重新定义查询，查找条件是将鱼(fish)当作食物，而不是匹配钓鱼(fishing)活动。添加一个额外的 filter 选项，并提供一个含有普通查询的文档。具体命令如下所示：

```

> db.texttest.runCommand( "text", { search : "fish", filter : { about : "food" } })
{
  "queryDebugString" : "fish|||||",
  "language" : "english",
  "results" : [
    {
      "score" : 0.75,
      "obj" : {
        "_id" : ObjectId("51d7ccb36bc6f959debe5514"),
        "number" : 1,
        "body" : "i like fish",
        "about" : "food"
      }
    }
  ],
  "stats" : {
    "nscanned" : 2,
    "nscannedObjects" : 2,
    "n" : 1,
    "nfound" : 1,
    "timeMicros" : 101
  },
  "ok" : 1
}

```

结果十分正确；返回了一个我们希望得到的文档，并没有返回所有与钓鱼(fishing)相关的文档。注意其中的 nscanned 和 nscannedObjects 值为 2，这意味着该查询扫描了索引(nscanned)中

的两个文档，然后获取这两个文档以检查它们的内容(`nscannedObjects`)，最终返回一个匹配文档(`n`)。接下来我们将学习另一个样例。

5. 更复杂的文本搜索

首先运行下面的查询，它将返回两个文档。为了保持代码简洁，这里只显示出它们的文本字段：

```
db.texttest.runCommand( "text", { search : "cook" })
"body" : "i want to cook dinner",
"body" : "i am to cooking lunch",
```

如你所见，其中有两个文档是关于做饭的。如果希望从搜索中排除午餐(`lunch`)，只返回晚餐(`dinner`)，那么可以在搜索中添加`-lunch`以排除午餐：

```
> db.texttest.runCommand( "text", { search : "cook -lunch" })
{
  "queryDebugString" : "cook||lunch||||",
  "language" : "english",
  "results" : [
    {
      "score" : 0.6666666666666666,
      "obj" : {
        "_id" : ObjectId("51d7ccb36bc6f959debe5518"),
        "number" : 5,
        "body" : "i want to cook dinner",
        "about" : "activities"
      }
    }
  ],
  "stats" : {
    "nscanned" : 2,
    "nscannedObjects" : 0,
    "n" : 1,
    "nfound" : 1,
    "timeMicros" : 150
  },
  "ok" : 1
}
```

首先要注意的是，`queryDebugString` 中同时包含了 `cook` 和 `lunch`，因为它们是我们正在使用的搜索关键词。另外还要注意该查询扫描了两个文档，但只返回了一个。该搜索首先将找到所有匹配的文档，然后再删除不匹配的文档。

最后一个人觉得非常有价值的搜索功能是字符串字面量搜索，它可用于在不抽取词干的情况下匹配特定的单词或词组。正如之前讲过的，单个搜索中的所有元素都将被分词并抽取词干，然后对每个关键字进行评估，如下所示：

```
> db.texttest.runCommand( "text", { search : "mongodb text search" })
{
  "queryDebugString" : "mongodb|search|text|||||",
  "language" : "english",
  "results" : [
    {
      "score" : 3.875,
      "obj" : {
        "_id" : ObjectId("51d7ccb36bc6f959debe551a"),
```

```

        "number" : 7,
        "body" : "i like mongodb text search",
        "about" : "food"
    }
},
{
    "score" : 3.8000000000000003,
    "obj" : {
        "_id" : ObjectId("51d7ccb36bc6f959debe551b"),
        "number" : 8,
        "body" : "mongodb has a new text search feature",
        "about" : "food"
    }
}
],
"stats" : {
    "nscanned" : 6,
    "nscannedObjects" : 0,
    "n" : 2,
    "nfound" : 2,
    "timeMicros" : 537
},
"ok" : 1
}

```

在 `queryDebugString` 中，每个元素都将被单独评估和查询。另外，最终该查询返回了两个文档。现在，在相同查询的两端添加转义后的引号，将它变成一个字符串字面量，注意运行结果的区别：

```

> db.texttest.runCommand( "text", { search : "\"mongodb text search\"" })
{
  "queryDebugString" : "mongodb|search|text|||mongodb text search|",
  "language" : "english",
  "results" : [
    {
      "score" : 3.875,
      "obj" : {
        "_id" : ObjectId("51d7ccb36bc6f959debe551a"),
        "number" : 7,
        "body" : "i like mongodb text search",
        "about" : "food"
      }
    }
  ],
  "stats" : {
    "nscanned" : 6,
    "nscannedObjects" : 0,
    "n" : 1,
    "nfound" : 1,
    "timeMicros" : 134
  },
  "ok" : 1
}

```

现在，该查询将只返回一个文档，该文档包含查询中指定的文本。另外，`queryDebugString` 中的最后一个元素是查询字符串自身，而不是 3 个被分词和抽取词干后的元素。

6. 额外的选项

除了之前已经讨论过的选项，还有 3 个选项可用在文本搜索功能中。第一个是 `limit`，它将限制返回的文档的数目，如下所示：

```
> db.texttest.runCommand( "text", { search : "fish", limit : 1 } )
```

第二个选项是 `project`，通过它可以控制显示在查询结果中的字段。该选项将接受一个用于描述希望显示的字段的文档，0 代表不显示，1 代表显示。如果使用该选项，默认除了 `_id`，所有的元素都不显示。

```
> db.texttest.runCommand( "text", { search : "fish", project : { _id : 0, body : 1 } } )
```

第三个也是最后一个选项是 `language`，通过它可以指定文本搜索所使用的语言。如果不指定语言，那么查询将使用索引的默认语言。该语言必须以小写的方式指定，如下所示：

```
> db.texttest.runCommand( "text", { search : "fish", language : "french" } )
```

目前文本搜索支持下列语言：

- Danish
- Dutch
- English
- Finnish
- French
- German
- Hungarian
- Italian
- Norwegian
- Portuguese
- Romanian
- Russian
- Spanish
- Swedish
- Turkish

关于 MongoDB 文本搜索的更多详细信息，请参考网页 <http://docs.mongodb.org/manual/reference/command/text/>。

8.1.4 其他语言中的文本索引

我们最初创建的简单文本索引只是为了能够演示文本索引的用法，但还有许多额外的技术可用于帮助改进文本索引。你应该还记得之前 MongoDB 是如何基于语言对单词进行词干抽取的。默认情况下，所有的索引都将按照英语进行创建，但这并不适合所有人，因为他们的数据可能不是用英语表达的，并且该语言的规则也不同于英语。因此可以在每个查询中都指定语言，但这并不是完全友好的，因为必须知道目前使用的语言。在索引创建时，可以添加选项来指定该索引的默认语言：

```
db.texttest.ensureIndex( { content : "text" }, { default_language : "french" } );
```

该命令将创建一个将法语作为默认语言的索引。记住，在每个集合上只能创建一个文本索引，所以在创建这个索引之前需要删除之前的文本索引。

但如果是在同一个集合中存在着多种语言，该如何处理呢？文本索引为此提供了一个解决方案，但它要求为所有的文档都标记上正确的语言。你可能会认为最好由 MongoDB 来决定某个文档的语言，但实际上无法通过编程的方式判断出语言类型。相反，MongoDB 允许在文档中指定自己的语言，例如下面的 4 个文档：

```
{ _id : 1, content : "cheese", lingvo : "english" }
{ _id : 2, content : "fromage", lingvo: "french" }
{ _id : 3, content : "queso", lingvo: "spanish" }
{ _id : 4, content : "ost", lingvo: "swedish" }
```

它们包含了 4 种语言(在 lingvo 字段中)，如果将某种语言用作默认语言，那么在搜索时就需要指定特定的语言。因为我们已经为内容指定了语言，所以可以将该字段用作 language_override，使用文档中给定的语言而不是默认语言。通过下面的命令将创建一个这样的索引：

```
db.texttest.ensureIndex( { content : "text" }, { language_override : "lingvo" } );
```

因此这些文档的默认语言将是文档中 lingvo 字段提供的语言，对于缺少 lingvo 字段的文档，它们将使用默认语言，在本例中也就是英语。

8.1.5 文本索引的复合索引

尽管在一个集合中确实只能创建一个文本索引，但文本索引可以覆盖文档中的多个字段，甚至是所有字段。可以如同普通索引一样，在文本索引中指定额外的字段。例如希望同时索引内容和评论，可以通过下面的命令实现：

```
db.texttest.ensureIndex( { content : "text", comments : "text" } );
```

如果希望基于文档的所有字段创建文本索引，那么 MongoDB 提供了通配符可用于引用所有文档的所有文本元素；该记号是 "\$*"。如果希望在文本索引中使用这种形式，那就需要添加一个索引选项以提供文档的名字。因为在这种情况下，自动生成出来的名字不能使用，并且会因为索引字段过长而引起问题。索引的最大长度是 121 个字符，包括集合、数据库和被索引字段的名称。

注意：

强烈推荐在使用混合索引时指定名称，从而避免因名称长度引起的问题。

下面的样例将创建一个名为 alltextindex 的文本索引，它基于 texttest 集合中所有文档的所有字符串元素创建：

```
db.texttest.ensureIndex( { "$*": "text" }, { name: "alltextindex" } )
```

创建复合索引时下一件需要完成的事情是，为索引中不同的文本字段指定权重。基于每个被索引字段的默认权重，为它们增加权重值即可。该值将会提高字段在索引结果中的重要性，比率为 N:1，如下所示：

```
db.texttest.ensureIndex( { content : "text", comments : "text"}, { weights : { content: 10, comments: 5, } } );
```

该索引意味着文档的内容部分将比评论值的优先级高(10:5)。相较于评论的权重 5 和内容的权重 10, 所有其他字段的默认权重都将是 1。还可以结合权重和通配文本搜索参数对特定的字段进行权重设置。

注意:

小心, 如果创建了太多文本索引, 将会出现 “too many text indexes” 错误。如果发生了该错误, 那么应该删除现有的文本索引, 再创建新的索引。

除了可以使用其他文本字段创建复合索引外, 还可以使用其他非文本字段创建复合索引。下面的命令将创建一个这样的索引:

```
db.texttest.ensureIndex( { content : "text", username : 1 } );
```

该命令将基于文档的内容部分创建一个文本索引和一个基于用户名的普通索引。在使用 filter 参数时, 这是非常有用的, 因为过滤器实际上是一个作用于所有子文档的查询。这些操作也需要从索引或文档中读取数据, 如下所示:

```
db.texttest.runCommand( "text", { search : "fish", filter : { about : "food" } } )
```

鉴于上面给出的查询, 我们需要索引文档的 about 部分; 否则, 所有理论上匹配的文档都将被全部读取和验证, 这是一个代价很高的处理过程。不过, 如果在索引中添加 about 元素, 那么将可以避免出现读取所有文档的情况:

```
db.texttest.ensureIndex( { about : 1, content : "text" } );
```

现在再次运行搜索命令:

```
> db.texttest.runCommand( "text", { search : "fish", filter : { about : "food" } } )
{
  "queryDebugString" : "fish|||||",
  "language" : "english",
  "results" : [
    {
      "score" : 0.75,
      "obj" : {
        "_id" : ObjectId("51d7ccb36bc6f959debe5514"),
        "number" : 1,
        "body" : "i like fish",
        "about" : "food"
      }
    }
  ],
  "stats" : {
    "nscanned" : 1,
    "nscannedObjects" : 0,
    "n" : 1,
    "nfound" : 1,
    "timeMicros" : 95
  },
  "ok" : 1
}
```

这次结果中将不再出现被扫描的对象，这将极大地提高查询的总体性能。通过这些选项，你应该能够感受到文本搜索的灵活和强大。

现在我们已经学习了 MongoDB 最新的搜索特性的众多强大之处，你也应该已经掌握了文本搜索的相关知识。

8.2 聚集框架

MongoDB 中的聚集框架代表着在集合的所有数据上执行选择操作的能力。它将通过创建聚集操作的管道流来实现，这些聚集操作将在数据上依次执行，后面的操作都将基于之前的结果执行。你所熟悉的 Linux 或 UNIX shell 都将这种行为看作 shell 的管道操作。

在聚集框架中有太多的操作符可用作聚集的一部分，进行数据收集。这里将讲解一些高级的管道操作符，然后通过一些样例演示如何使用它们。这意味着我们将讲解以下操作符：

- \$group
- \$limit
- \$match
- \$sort
- \$unwind
- \$project
- \$skip

关于所有操作符的更多详细信息，请查看聚集框架的文档 <http://docs.mongodb.org/manual/aggregation/>。我们已经创建了一些样例集合，它们可用于测试聚集命令。使用下面的命令解压文件：

```
$ tar -xvf test.tgz
x test/
x test/aggregation.bson
x test/aggregation.metadata.json
x test/mapreduce.bson
x test/mapreduce.metadata.json
```

接下来执行 mongorestore 命令，将数据恢复到 test 数据库中：

```
$ mongorestore test
connected to: 127.0.0.1
Sun Jul 21 19:26:21.342 test/aggregation.bson
Sun Jul 21 19:26:21.342 going into namespace [test.aggregation]
1000 objects found
Sun Jul 21 19:26:21.350 Creating index: { key: { _id: 1 }, ns: "test.aggregation ",
name: "_id_" }
Sun Jul 21 19:26:21.688 test/mapreduce.bson
Sun Jul 21 19:26:21.689 going into namespace [test.mapreduce]
1000 objects found
Sun Jul 21 19:26:21.695 Creating index: { key: { _id: 1 }, ns: "test.mapreduce",
name: "_id_" }
```

现在我们已经有了一个可用的数据集合，接下来学习如何执行聚集命令以及如何构建聚集管道。为了运行聚集查询，我们将使用 aggregate 命令，并提供一个包含了管道的文档。为了测

试，我们将在下面的聚集命令中使用各种不同的管道文档：

```
> db.aggregation.aggregate({pipeline document})
```

不再多言，现在就开始学习聚集样例。

8.2.1 \$group

顾名思义，命令\$group 要完成的任务就是对文档进行分组，只有通过它才可以根据分组数据创建出聚集结果。首先我们将创建一个简单的分组命令，它将列出 aggregation 集合中所有不同的颜色。在开始之前，先创建一个_id 文档，它将用于列出集合中我们希望进行分组的所有元素。所以，先使用\$group 创建管道文档，并将它添加到我们的_id 文档中：

```
{ $group : { _id : "$color" } }
```

现在可以看到_id 的值为“\$color”。注意在名字 color 之前有一个\$符号，这表示该元素是对文档中字段的引用。这将为我们创建出基本的文档结构，接着执行下面的聚集命令即可：

```
> db.aggregation.aggregate( { $group : { _id : "$color" } } )
{
  "result" : [
    {
      "_id" : "red"
    },
    {
      "_id" : "maroon"
    },
    ...
    {
      "_id" : "grey"
    },
    {
      "_id" : "blue"
    }
  ],
  "ok" : 1
}
```

8.2.2 \$sum

从\$group 操作符的执行结果可以看到集合中有许多不同的颜色。结果是一个元素的数组，其中包含了许多文档，每个_id 值都是文档中“color”字段中的一种颜色。这并未显示出太多信息，所以让我们对\$group 所实现的任务进行扩展。在分组中使用\$sum 操作符添加一个计数，每遇到一个找到的文档就将该数字加 1。为此，需要在\$group 命令中添加一个新的字段，并指定该字段应该显示的值。在本例中，我们称该字段为“count”，因为它代表了每种颜色出现的次数；它的值为{\$sum:1}，这意味着我们希望为每个文档创建一个计数，每次遇到相同的文档时就将该数字加 1。修改后的文档如下所示：

```
{ $group : { _id : "$color", count : { $sum : 1 } }
```

现在使用新的文档再次运行聚集命令：

```
> db.aggregation.aggregate( { $group : { _id : "$color", count : { $sum : 1 } } }
```

```

{
  "result" : [
    {
      "_id" : "red",
      "count" : 90
    },
    {
      "_id" : "maroon",
      "count" : 91
    },
    ...
    {
      "_id" : "grey",
      "count" : 91
    },
    {
      "_id" : "blue",
      "count" : 91
    }
  ],
  "ok" : 1
}

```

现在可以查看每种颜色出现的次数了。我们可以通过在 `_id` 文档中添加额外的元素来对分组数据进行扩展。例如我们希望搜索“color”和“transport”的分组。为此，可以将 `_id` 文档修改为一个包含了以下子文档的文档：

```
{ $group : { _id : { color: "$color", transport: "$transport" }, count : { $sum : 1 } } }
```

如果使用该文档再次运行聚集命令，结果中大概将返回 50 个元素，因为太长了，所以无法显示在这里。为了解决这个问题，可以使用 `$limit` 操作符。

8.2.3 \$limit

操作符 `$limit` 是我们将使用的下一个管道操作符。顾名思义，`$limit` 可用于限制返回的结果的数目。在这里，为了使已有管道的结果更加易于管理，让我们将结果的数量限制为 5。为了添加这个限制，需要将我们的一个文档转换为一组管道文档。

```

[
  { $group : { _id : { color: "$color", transport: "$transport" }, count : { $sum : 1 } } },
  { $limit : 5 }
]

```

运行该文档将显示出下面的结果：

```

> db.aggregation.aggregate( [ { $group : { _id : { color: "$color", transport:
"$transport" }, count: { $sum : 1 } } }, { $limit : 5 } ] )
{
  "result" : [
    {
      "_id" : {
        "color" : "maroon",
        "transport" : "motorbike"
      },
      "count" : 18
    }
  ]
}

```



```

    },
    {
      "_id" : {
        "color" : "orange",
        "transport" : "automobile"
      },
      "count" : 18
    },
    {
      "_id" : {
        "color" : "green",
        "transport" : "train"
      },
      "count" : 18
    },
    {
      "_id" : {
        "color" : "purple",
        "transport" : "train"
      },
      "count" : 18
    },
    {
      "_id" : {
        "color" : "grey",
        "transport" : "plane"
      },
      "count" : 18
    }
  ],
  "ok" : 1
}

```

从结果可以看到 `_id` 中额外添加了 `"transport"` 元素，并且由于限制的存在，只返回了 5 个元素。现在你应该已经明白了如何使用多个操作符构建出管道，从而根据集合中的数据计算出聚集信息。

8.2.4 \$match

下一个学习的操作符是 `$match`，通过它可在聚集管道中高效地返回一个普通 MongoDB 查询的结果。操作符 `$match` 最好用在管道的开始，用于限制一开始被输入到管道中的文档的数量；通过限制被处理文档的数量，可以大大地减小性能开销。例如，假设我们只希望基于 `num` 值大于 500 的文档执行管道操作，可以使用查询 `{num: {$gt: 500}}` 返回所有匹配该条件的文档。如果将该查询作为 `$match` 添加到已有的聚集中，将得到下面的文档：

```

[
  { $match : { num : { $gt : 500 } } },
  { $group : { _id : { color: "$color", transport: "$transport" }, count : { $sum :
1 } } },
  { $limit : 5 }
]

```

使用聚集命令执行该文档，将得到下面的结果：

```

{
  "result" : [

```

```

{
  "_id" : {
    "color" : "white",
    "transport" : "boat"
  },
  "count" : 9
},
{
  "_id" : {
    "color" : "black",
    "transport" : "motorbike"
  },
  "count" : 9
},
{
  "_id" : {
    "color" : "maroon",
    "transport" : "train"
  },
  "count" : 9
},
{
  "_id" : {
    "color" : "blue",
    "transport" : "automobile"
  },
  "count" : 9
},
{
  "_id" : {
    "color" : "green",
    "transport" : "automobile"
  },
  "count" : 9
}
],
"ok" : 1
}

```

注意，这里返回的结果与之前的样例几乎完全不同，这是因为文档的创建顺序已经变了。因此，当我们运行该查询并限制结果输出时，之前输出中显示的文档将被移除，并且结果中的计数值只有之前结果的一半，这是因为我们将聚集的目标数据减少到了之前的一半。如果希望每次返回的结果顺序保持一致，那就需要使用另一个管道操作符\$sort。

8.2.5 \$sort

如你所见，\$limit 将改变结果中返回的文档，因为它反映了文档在聚集执行输出中的原始顺序。通过使用\$sort 命令可以解决这个问题。为了返回相同的限制结果，只需要在执行限制之前在特定的字段上排序即可。\$sort 语法与它在普通查询中的使用方式相同；可以对文档进行排序，正数代表升序，负数代表降序。为了演示它的用法，让我们分别运行两个查询，一个使用匹配并且将结果限制为 1，另一个则什么也不使用。此时由于在\$limit 之前使用了\$sort，所以结果将总是以相同的顺序显示文档。

下面是第一个查询:

```
[
  { $group : { _id : { color: "$color", transport: "$transport" } , count : { $sum :
1 } } },
  { $sort : { _id :1 } },
  { $limit : 5 }
]
```

该查询的结果是:

```
{
  "result" : [
    {
      "_id" : {
        "color" : "black",
        "transport" : "autombile"
      },
      "count" : 18
    }
  ],
  "ok" : 1
}
```

下面是第二个查询:

```
[
  { $match : { num : { $gt : 500 } } },
  { $group : { _id : { color: "$color", transport: "$transport" } , count : { $sum :
1 } } },
  { $sort : { _id :1 } },
  { $limit : 1 }
]
```

该查询的结果为:

```
{
  "result" : [
    {
      "_id" : {
        "color" : "black",
        "transport" : "autombile"
      },
      "count" : 9
    }
  ],
  "ok" : 1
}
```

你会注意到这两个查询的结果中将包含相同的文档, 尽管它们的总数不同。这意味着在对结果进行限制之前已经执行了排序, 所以可以帮助我们得到一致的结果。这些操作符将帮助你构建出操作符管道来操作查询的结果, 直到得到期望的结果。

8.2.6 \$unwind

下一个将要学习的操作符是\$unwind。它将接受一个数组并将每个元素分割到一个新的文档中(在内存中而不是添加到集合中)。如同学习 shell 管道一样, 理解\$unwind 操作符结果的最好

方式就是执行它并查看它的输出结果。下面是\$unwind的结果输出：

```

db.aggregation.aggregate({ $unwind : "$vegetables" });
{
  "result" : [
    {
      "_id" : ObjectId("51de841747f3a410e3000001"),
      "num" : 1,
      "color" : "blue",
      "transport" : "train",
      "fruits" : [
        "orange",
        "banana",
        "kiwi"
      ],
      "vegetables" : "corn"
    },
    {
      "_id" : ObjectId("51de841747f3a410e3000001"),
      "num" : 1,
      "color" : "blue",
      "transport" : "train",
      "fruits" : [
        "orange",
        "banana",
        "kiwi"
      ],
      "vegetables" : "brocoli"
    },
    {
      "_id" : ObjectId("51de841747f3a410e3000001"),
      "num" : 1,
      "color" : "blue",
      "transport" : "train",
      "fruits" : [
        "orange",
        "banana",
        "kiwi"
      ],
      "vegetables" : "potato"
    },
    ...
  ],
  "ok" : 1
}

```

现在结果数组中有 3000 个文档，每个版本的文档都有自己的蔬菜(vegetables)和原始文档的剩余部分！通过该结果可以看到\$unwind的强大之处，不过与此同时，一个巨大的文档集合也可能给自己带来麻烦。一定要记住先运行匹配，这样就可以在执行其他操作符之前减少所处理的对象的数量，得到更密集的聚集操作。

8.2.7 \$project

下一个将要讲解的操作符是\$project，通过它可以限制返回的文档中的字段或者重命名其中的字段。这仅仅是一个可用在 find 命令中的字段限制参数。这是减少聚集中所返回的多余字段

的最好方式。例如，如果只希望看到每个文档中的水果和蔬菜；那么可以提供一個文档以列出希望显示(或不显示)的元素，正如之前添加到 `find` 命令中的文档一样，如下所示：

```
[
  { $unwind : "$vegetables" },
  { $project : { _id: 0, fruits:1, vegetables:1 } }
]
```

该命令将返回下面的结果：

```
{
  "result" : [
    {
      "fruits" : [
        "orange",
        "banana",
        "kiwi"
      ],
      "vegetables" : "corn"
    },
    {
      "fruits" : [
        "orange",
        "banana",
        "kiwi"
      ],
      "vegetables" : "broccoli"
    },
    {
      "fruits" : [
        "orange",
        "banana",
        "kiwi"
      ],
      "vegetables" : "potato"
    },
    ...
  ],
  "ok" : 1
}
```

这个结果比之前的要好，因为现在文档的大小也变小了。但如果能够减少返回的文档的数量，那就更好了。下一个将要学习的操作符可以帮助解决这个问题。

8.2.8 \$skip

`$skip` 管道操作符是对 `$limit` 操作符的补充，但它不是将结果限制为前 X 个文档，而是忽略它们，并返回所有剩余的文档。我们可以用它减少返回文档的数量。如果在之前的查询中添加它，并将值设置为 2995，那么最后将会得到 5 个结果，如下所示：

```
[
  { $unwind : "$vegetables" },
  { $project : { _id: 0, fruits:1, vegetables:1 } },
  { $skip : 2995 }
]
```

执行结果为:

```
{
  "result" : [
    {
      "fruits" : [
        "kiwi",
        "pear",
        "lemon"
      ],
      "vegetables" : "pumpkin"
    },
    {
      "fruits" : [
        "kiwi",
        "pear",
        "lemon"
      ],
      "vegetables" : "mushroom"
    },
    {
      "fruits" : [
        "pear",
        "lemon",
        "cherry"
      ],
      "vegetables" : "pumpkin"
    },
    {
      "fruits" : [
        "pear",
        "lemon",
        "cherry"
      ],
      "vegetables" : "mushroom"
    },
    {
      "fruits" : [
        "pear",
        "lemon",
        "cherry"
      ],
      "vegetables" : "capsicum"
    }
  ],
  "ok" : 1
}
```

这就是通过使用 `$skip` 操作符减少返回文档数量的方式。还可以使用 `$limit` 操作符以相同的方式限制结果的数量，并且可以通过结合使用它们获得集合中间的一组数据。假设现在希望得到 3000 条数据中的第 1500-1510 条数据，那么可以将 `$skip` 设置为 1500，并将 `$limit` 设置为 10，这样就可以得到我们所需的 10 条数据。

我们已经学习了 MongoDB 聚集框架中的一些可用的高级管道操作符，还有许多更小的操作符可作为管道表达式，作用于高级管道操作符中。它们包括一些地理空间函数、数学计算函数，用于执行诸如求平均值、第一和最后一个，以及日期/时间和其他操作。如同之前讲解的高

级操作符一样，所有这些操作都可单独或结合使用于聚集操作中。不过要记住，管道中的每个操作符都将基于之前操作的结果执行，并且可以输出和逐个执行它们以创建出目标结果。

8.3 MapReduce

MapReduce 是 MongoDB 中最复杂的查询机制之一。它通过两个 JavaScript 函数实现查询：map 和 reduce。这两个函数完全由用户自定义，并且能够提供极其强大的灵活性！下面将通过一些简短的例子演示如何使用 MapReduce。

8.3.1 MapReduce 的工作方式

在深入学习样例之前，最好先学习映射/规约的定义和工作方式。在 MongoDB 的 MapReduce 实现中，它将对指定的集合执行一个专门的查询，所有匹配该查询的文档都将被输入到 map 函数中。map 函数被设计用于生成键/值对。任何含有多个值的键都将被输入到 reduce 函数中，reduce 函数将返回输入数据的聚集结果。在此之后，还有一个可选的步骤，通过 finalize 函数对数据的显示进行完善。

8.3.2 创建测试文档

在开始之前，首先需要创建测试文档。在之前恢复的 test 数据库中，我们已经创建了一个 mapreduce 集合。如果尚未恢复该数据库，那么可以使用下面的命令解压该压缩文件：

```
$ tar -xvf test.tgz
x test/
x test/aggregation.bson
x test/aggregation.metadata.json
x test/mapreduce.bson
x test/mapreduce.metadata.json
```

然后运行 mongorestore 命令以恢复 test 数据库：

```
$ mongorestore test
connected to: 127.0.0.1
Sun Jul 21 19:26:21.342 test/aggregation.bson
Sun Jul 21 19:26:21.342 going into namespace [test.aggregation]
1000 objects found
Sun Jul 21 19:26:21.350 Creating index: { key: { _id: 1 }, ns: "test.aggregation",
name: "_id_" }
Sun Jul 21 19:26:21.688 test/mapreduce.bson
Sun Jul 21 19:26:21.689 going into namespace [test.mapreduce]
1000 objects found
Sun Jul 21 19:26:21.695 Creating index: { key: { _id: 1 }, ns: "test.mapreduce", name:
"_id_" }
```

现在，MongoDB 集合中已经添加了可用于执行 MapReduce 的文档。接下来，我们先学习一下简单的 map 函数。

8.3.3 使用 map 函数

map 函数将从 mapreduce 集合的每个文档中“发射”颜色和 num 值。这两个字段都将以键/值

对的形式输出，第一个参数(颜色)为键，第二个参数(数字)为值。这里有许多内容需要了解，所以先演示一下简单的 map 函数：

```
var map = function() {
  emit(this.color, this.num);
};
```

为了执行映射/规约，还需要 reduce 函数，但在此之前先查看空 reduce 函数所能提供的结果：

```
var reduce = function(color, numbers) { };
```

在 shell 中输入这些命令，现在可以开始运行 MapReduce 命令了。

现在为 MapReduce 提供一个输出字符串。该字符串定义了 MapReduce 命令输出的位置。

下面是两个常用选项：

- 集合
- 控制台(内嵌)

为了演示 reduce 函数的用法，现在将结果输出到控制台。将文档中 out 选项的值设置为 {inline:1}，如下所示：

```
{ out : { inline : 1 } }
```

最终得到的命令将如下所示：

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

执行结果为：

```
{
  "results" : [
    {
      "_id" : "black",
      "value" : null
    },
    {
      "_id" : "blue",
      "value" : null
    },
    {
      "_id" : "brown",
      "value" : null
    },
    {
      "_id" : "green",
      "value" : null
    },
    {
      "_id" : "grey",
      "value" : null
    },
    {
      "_id" : "maroon",
      "value" : null
    },
    {
      "_id" : "orange",
```



```

        "value" : null
      },
      {
        "_id" : "purple",
        "value" : null
      },
      {
        "_id" : "red",
        "value" : null
      },
      {
        "_id" : "white",
        "value" : null
      },
      {
        "_id" : "yellow",
        "value" : null
      }
    ],
    "timeMillis" : 95,
    "counts" : {
      "input" : 1000,
      "emit" : 1000,
      "reduce" : 55,
      "output" : 11
    },
    "ok" : 1,
  }
}

```

结果显示，为每种颜色创建了一个单独的文档，并且使用颜色作为文档的唯一 `_id` 值。因为每个文档的值部分什么也没有指定，所以它们被设置为 `null`。为了得到目标 `MapReduce` 结果，可以通过添加输出块代码来修改 `MapReduce` 的行为。在本例中，我们希望得到每个函数要接受的汇总数据。为此，我们可以使用函数将返回的数据修改为对象，取代 `null`。现在让我们返回每种颜色的所有值的总数。为此创建一个函数，返回每种颜色的所有数组中数字的总和，并将该值传递给 `reduce` 函数。幸亏我们可以通过一个方便的函数 `Array.sum` 来计算数组中所有值的总和。下面就是我们的 `reduce` 函数：

```

var reduce = function(color, numbers) {
  return Array.sum(numbers);
};

```

另外，除了使用内嵌输出，还可以让 `MapReduce` 将结果写入到集合中；为了做到这一点，可以使用希望输出的集合名称替换 `{inline: 1}`。将结果输出到集合 `mrresult` 中，如下所示：

```

db.mapreduce.mapReduce(map, reduce, { out: "mrresult" });

```

执行新的 `reduce` 函数，结果将如下所示：

```

{
  "result" : "mrresult",
  "timeMillis" : 111,
  "counts" : {
    "input" : 1000,
    "emit" : 1000,
    "reduce" : 55,

```

```

    "output" : 11
  },
  "ok" : 1,
}

```

如果希望看到文档结果，可以查询 `mrresult` 集合：

```

> db.mrresult.findOne();
{ "_id" : "black", "value" : 45318 }

```

现在，这个基础系统已经可以正常运行了，接下来我们将对它进行改进！

8.3.4 高级 MapReduce

现在我们希望得到所有值的平均值而不是总和！这将变得困难得多，因为我们需要添加另一个变量——已有对象的数量！但如何从 `map` 函数中得到两个变量呢？毕竟 `emit` 函数只接受两个参数。我们执行一个“欺骗”操作：返回一个包含了所有必需字段的 JSON 文档！让我们扩展原来的 `map` 函数，让它返回一个包含了颜色值和计数器值的文档。首先，将文档定义为一个新变量，使用 JSON 文档设置该变量的值，然后发射该文档：

```

var map = function() {
  var value = {
    num : this.num,
    count : 1
  };
  emit(this.color, value);
};

```

注意为了只统计每个文档一次，这里我们将计数器的值设置为 1！现在需要修改 `reduce` 函数，让它处理之前创建的文档数组。最后需要注意的是，我们在 `reduce` 函数的 `return` 函数中返回的值，需要与之前在 `map` 函数中创建并发送到 `emit` 函数的值相同。

注意：

还可以通过使用包含了所有数字的数组的长度来替代我们之前所完成的代码，但使用这种方式需要对 MapReduce 有更多的了解。

为处理该数组，我们已经创建了一个简单的 `for` 循环和数组长度，并且我们将遍历该数组的每个成员，将它们的 `num` 和 `count` 值添加到新返回的变量 `reduceValue` 中。现在只需要返回该值即可得到所需的结果。

```

var reduce = function(color, val ) {
  reduceValue = { num : 0, count : 0};
  for (var i = 0; i <val.length; i++) {
    reduceValue.num += val[i].num;
    reduceValue.count += val[i].count;
  }
  return reduceValue;
};

```

此时，你应该想知道如何才能得到平均值。我们已经得到了统计值和数字，但不是真正的平均值！如果再次运行 MapReduce，你就可以看到结果。现在，要注意 MapReduce 每次将结果输出到集合时，都将在写入之前删除该集合！这对于我们来说是好事，因为我们只需要当前运

行的结果，不过在将来你可能需要多次运行的结果。如果希望将两次运行的结果合并在一起，那么可以使用 `{out: {merge: "mrresult"}}` 这样的文档。

```
db.mapreduce.mapReduce(map,reduce,{ out: "mrresult" });
```

现在快速检查该命令的结果：

```
> db.mrresult.findOne();
{
  "_id" : "black",
  "value" : {
    "num" : 18381,
    "count" : 27028,
  }
}
```

这并不是平均值。这意味着我们还有更多的事情要做，但如何才能在一个匹配 `emit` 函数输入的文档中包含平均值呢？我们需要第 3 个函数！`MapReduce` 提供了 `finalize` 函数。通过该函数可以在返回 `MapReduce` 结果之前执行任何最后的结束工作。下面编写一个函数，从 `reduce` 函数接收结果，并计算出平均值：

```
var finalize = function(key, value) {
  value.avg = value.num/value.count;
  return value;
};
```

是的，就是这么简单。在编写 `map`、`reduce` 和 `finalize` 函数之后，就可以调用它们了。终止选项被设置在最后一个文档中；与 `out` 参数一起，如下所示：

```
db.mapreduce.mapReduce(map,reduce,{ out: "mrresult", finalize : finalize });
```

现在快速地确认执行结果：

```
> db.mrresult.findOne();
{
  "_id" : "black",
  "value" : {
    "num" : 45318,
    "count" : 91,
    "avg" : 498
  }
}
```

现在好多了！我们得到了数目、统计值和平均值！

8.3.5 调试 MapReduce

调试映射/规约是非常耗时的任务，但有一些技巧可以简化这个任务。首先我们将学习如何调试 `map` 函数。通过重载 `map` 函数的 `emit` 函数调试它，如下所示：

```
var emit = function(key, value) {
  print("emit results - key: " + key + " value: " + tojson(value));
}
```

`emit` 函数将返回与 `map` 函数返回结果相同的键和值。通过使用 `map.apply()` 和集合中的一个

样例文档对它进行测试:

```
> map.apply(db.mapreduce.findOne());  
emit results - key: blue value: { "num" : 1, "count" : 1 }
```

在了解了如何调试 map 函数之后,接下来学习调试 reduce 函数。首先需要确认 map 和 reduce 函数返回结果的格式相同,这非常严格。下一件需要完成的事情是创建一个简短的数组,如同传入到 reduce 函数中的数组一样,如下所示:

```
a = [{ "num" : 1, "count" : 1 },{ "num" : 2, "count" : 1 },{ "num" : 3, "count" : 1 }]
```

现在调用 reduce 函数。将显示出 emit 函数返回的结果:

```
> reduce("blue",a);  
{ "num" : 6, "count" : 3 }
```

如果出现某些问题,并且你对函数中的内容感到混淆,那么可以使用 printjson() 函数将 JSON 值输出到 mongod 日志文件中。在调试软件时,这总是一个非常有价值的工具。

8.4 小结

到现在为止,你应该已经了解了 MongoDB 中最强大且灵活的查询系统的独特之处。通过阅读本章内容,你应该已经明白如何使用文本索引在大量的语言中执行高效的文本搜索。你应该具有使用 MongoDB 聚集框架创建高度复杂和灵活的聚集的能力。最后,你现在应该具有使用强大的基于 JavaScript 的 MapReduce 的能力,通过它可以针对数据执行强大的分组和转换操作。

第III部分



使用 MongoDB 处理大数据

第 9 章 数据库管理

第 10 章 优化

第 11 章 复制

第 12 章 分片

数据库管理

在本章，我们将学习 MongoDB 服务器的一些基本管理操作。我们还将学习如何以自动化的方式完成某些操作，例如备份服务器。

因为 MongoDB 是一个非关系数据库系统，所以许多数据库管理员需要执行的传统功能在这里并不需要执行。例如，不需要在服务器上创建数据库、集合或字段，因为 MongoDB 将在访问它们时动态创建这些元素。因此，在大多数情况下都不需要管理数据库和模式。

不过，这种不需要预定义所有元素的自由特性也可能导致创建出预料之外的元素，例如无关的集合或文档中无关的字段。管理员和开发者只需要偶尔从数据库中清除未使用的数据元素即可，尤其是在项目的开发阶段，此时变化通常是非常快的。开发者在最后确定解决方案之前必须尝试多种方式并清理数据库。MongoDB 的易用性也鼓励了这种探索性的开发模式；不过，也可能导致数据库中产生混乱，因为创建数据结构的工作量几乎为零。

产生混乱的另一个因素，也是 MongoDB 和 SQL 数据库之间最大的区别之一：在所有平台上，MongoDB 中的所有对象和元素名称都是大小写敏感的。因此，foo 和 Foo 集合名称引用的是两个完全不同的集合。因此，需要小心数据库和集合的名称，避免不小心创建出多个只是名字大小写不同的数据库（不过，在 MongoDB 2.4 中有一个例外：不能创建只是名称大小写不同的数据库，这将会产生错误）。

这些数据库的不同版本将填满你的硬盘，它们可能会使系统的开发者和终端用户产生混淆，因为开发者和用户可能会连接到不完整或意料之外的数据。

在本章，我们将学习如何执行下面的任务：

- 备份和还原 MongoDB 系统。
- 使用 MongoDB shell(通过 mongo 命令调用)执行常见的任务。
- 使用认证控制服务器的访问权限。
- 监控数据库实例。

不过，在深入学习这些任务之前，首先学习一些用于执行其中大多数任务的工具。

9.1 使用管理工具

管理者需要一些执行日常任务的工具，通过它们可以保持服务器的平稳运行。在 MongoDB 包中有一些非常好的工具可以使用，另外它还包含了一些有用的第三方工具。下面将讲解一些最重要的可用工具以及如何使用它们。

9.1.1 mongo——MongoDB 控制台

作为管理员，你将要使用的主要工具是 mongo——MongoDB 控制台工具。mongo 是一个基于 JavaScript 的命令行控制台工具，它类似于主流关系数据库所提供的许多查询工具。不过，mongo 有自己的独特之处：可以直接运行以 JavaScript 编写的与 MongoDB 数据库交互的程序。

通过该控制台可以使用 JavaScript 与 MongoDB 进行交互，并且可以将这些脚本保存为 .js 文件，以便在需要的时候运行。事实上，mongo 控制台中的许多内置命令都是以 JavaScript 编写的。

可以将命令 shell 中输入的许多命令添加到一个扩展名为 .js 的文件中，并且可以通过在启动 shell 时，以添加该文件名到命令行或者从 shell 中使用 load() 函数的方式运行它们。shell 将会执行文件的内容，然后退出。这在运行一组重复命令时非常有用。

在本章，我们将使用 mongo 控制台演示许多可在 MongoDB 服务器上执行的管理任务，并且结果将被传入 MongoDB 服务器，所以我们可以保证这些任务的执行结果。

9.1.2 使用第三方管理工具

MongoDB 现在有几个可用的第三方管理工具。MongoDB Inc. 在 MongoDB 网站上维护了一个页面，列出了所有可用的第三方工具。具体信息请访问网址 <http://docs.mongodb.org/ecosystem/tools/administration-interfaces/>。

其中许多工具都是基于 Web 的，并且类似于 MySQL 的 phpMyAdmin，但也有一些成熟的桌面 UI。

9.2 备份 MongoDB 服务器

MongoDB 管理员需要学习的第一个技巧是备份和恢复 MongoDB 服务器。学习了该技巧之后，就可以放心学习更高级的管理功能了，因为数据已经被安全地保存在某个地方。

9.2.1 创建第一个备份

首先让我们执行一个简单的备份，然后恢复它。与此同时还要保证备份数据是完整的，然后再通过一些实际的样例演示备份和恢复特性是如何工作的。一旦对这些特性有了深入了解之后，就可以继续学习 MongoDB 更高级的管理特性了。

这个简单的备份样例将基于以下假设构建：

- MongoDB 服务器运行在你目前登录的机器上。
- 硬盘空间足以容纳转储文件，该文件最大可能与数据库大小相同。
- 备份文件将保存在个人主目录中。这就避免了处理权限相关的问题。

MongoDB 的备份工具被称为 mongodump，该工具作为标准发行版本的一部分发布。下面的样例将在运行的服务器上执行一个简单的备份，并将文件保存在指定的磁盘目录中：

```
$> cd ~
$> mkdir testmongobackup
$> cd testmongobackup
$> mongodump
```

当 `mongodump` 运行时，它的输出将如下所示：

```
$ mongodump
connected to: 127.0.0.1
Tue May 21 20:52:58.639 all dbs
Tue May 21 20:52:58.640 DATABASE: blog to dump/blog
Tue May 21 20:52:58.640   blog.system.indexes to dump/blog/system.indexes.bson
Tue May 21 20:52:58.641     4 objects
Tue May 21 20:52:58.641   blog.system.profile to dump/blog/system.profile.bson
Tue May 21 20:52:58.645     3688 objects
Tue May 21 20:52:58.645   Metadata for blog.system.profile to
dump/blog/system.profile.metadata.json
Tue May 21 20:52:58.645   blog.authors to dump/blog/authors.bson
Tue May 21 20:52:58.646     1 objects
Tue May 21 20:52:58.646   Metadata for blog.authors to dump/blog/authors.
metadata.json
Tue May 21 20:52:58.646   blog.posts to dump/blog/posts.bson
Tue May 21 20:52:58.686     29997 objects
Tue May 21 20:52:58.709   Metadata for blog.posts to
dump/blog/posts.metadata.json
Tue May 21 20:52:58.710   blog.tagcloud to dump/blog/tagcloud.bson
Tue May 21 20:52:58.710     1 objects
Tue May 21 20:52:58.710   Metadata for blog.tagcloud to dump/blog/tagcloud.
metadata.json
```

如果看到的输出与上面的不同，那就需要再检查一下个人的环境是否符合该样例的假设条件。

如果看到正确的输出，那就表明你的数据库已经被备份到了 `testmongobackup/dump` 目录中。

下面的脚本将把数据库恢复到之前备份时的状态：

```
$> cd ~/testmongobackup
$> mongorestore --drop
connected to: 127.0.0.1
Tue May 21 20:53:46.337 dump/blog/authors.bson
Tue May 21 20:53:46.337   going into namespace [blog.authors]
Tue May 21 20:53:46.337   dropping
1 objects found
Tue May 21 20:53:46.338   Creating index: { key: { _id: 1 }, ns: "blog.authors",
name: "_id_" }
Tue May 21 20:53:46.339 dump/blog/posts.bson
Tue May 21 20:53:46.339   going into namespace [blog.posts]
Tue May 21 20:53:46.339   dropping
29997 objects found
Tue May 21 20:53:47.284   Creating index: { key: { _id: 1 }, ns: "blog.posts",
name: "_id_" }
Tue May 21 20:53:47.375   Creating index: { key: { Tags: 1 }, ns: "blog.posts",
name: "Tags_1" }
Tue May 21 20:53:47.804 dump/blog/system.profile.bson
Tue May 21 20:53:47.804   skipping
Tue May 21 20:53:47.804 dump/blog/tagcloud.bson
Tue May 21 20:53:47.804   going into namespace [blog.tagcloud]
Tue May 21 20:53:47.804   dropping
1 objects found
Tue May 21 20:53:47.821   Creating index: { key: { _id: 1 }, ns: "blog.tagcloud",
name: "_id_" }
```

选项 `--drop` 告诉 `mongorestore` 工具在恢复集合之前先丢弃现有的数据。最终，备份数据将替换数据库现有的数据。如果不使用 `--drop` 选项，被恢复的数据将被追加到每个集合的尾部，这将导致出现重复的数据。

让我们认真检查一下该例到底完成了什么工作。

默认情况下，`mongodump` 工具将使用默认端口连接到本地数据库，并将所有数据库和集合相关的数据抓取出来，然后存储在预定义的文件夹结构中。

`mongodump` 默认创建的文件夹结构形式为：

```
./dump/[databasename]/[collectionname].bson
```

样例中的数据库系统由单个数据库 `blog` 组成。数据库 `blog` 中包含了 3 个集合：`authors`、`posts` 和 `tagcloud`。

`mongodump` 将它从数据库服务器取出的数据存储在 `.bson` 文件中，该文件只是 MongoDB 内部存储的 BSON 格式文件的一个副本。在之前的样例中，还可以看到每个集合恢复后构建出的索引。MongoDB 服务器将每个集合的索引及其定义都保存在 `metadata.json` 文件中。正是这些元数据文件保证了从备份数据恢复时可以正确地构建出索引。

将数据库转储至文件之后，就可以将文件夹归档或保存在任何在线或离线媒介中，例如 CD、USB 驱动、磁带或 S3 格式。

注意：

`mongodump` 工具在写入备份文件之前，不会清空输出目录中的内容。如果输出目录中已经具有某些数据，那么它们不会被删除，除非它们的名字正好与 `mongodump` 将要备份的文件名 (`collectionname.bson`) 相同。如果希望将多个集合的转储文件保存在同一个目录中，那么这是一件好事；不过，如果每次备份时都使用相同的目录但不清空，那么可能会引起问题。例如，假设现在有一个数据库需要定时进行备份，并且在某个时间你决定要从数据库中删除一个集合。除非清空正在执行备份的目录，或者手动删除与被删除集合相关的集合，否则下次恢复数据时，被删除的集合将再次出现。除非你希望覆盖备份中的数据，否则应该在使用 `mongodump` 之前保证清空目标目录。

9.2.2 备份单个数据库

在相同的服务器上运行多个应用时，通常你会发现希望自己分别备份每个数据库，而不是和之前的样例一样一次备份所有数据库。

`mongodump` 工具提供了 `-d database_name` 选项用于实现单个数据库的备份。`mongodump` 工具将为它创建 `./dump` 文件夹；不过，该文件夹中将只包含单个数据库的所有备份文件。

9.2.3 备份单个集合

想象一下，现在有一个博客网站，其中的作者集合基本不变。相反，保存在 `posts` 和 `tagcloud` 集合中的博客网站的内容将一直保持快速变化。此时可以考虑每天只备份一次所有的数据，但每小时备份一次这两个集合。幸运的是，通过在 `mongodump` 命令中使用 `-c` 选项指定希望备份的集合可以轻松实现该功能。

`mongodump` 工具不会清空目标目录。这意味着，对于每个希望备份的集合，都可以成功调用 `mongodump` 将指定的集合添加到备份中，如下所示：

```
$mkdir ~/backuptemp
$cd ~/backuptemp
$mongodump -d blog -c posts
$mongodump -d blog -c tagcloud
```

...

将转储目录~/backuptemp 归档为 tar 文件

...

```
$ cd ~
```

```
$ rm -rf backuptemp
```

9.3 深入学习备份

此时，你已经知道如何执行备份和恢复数据这样的基本任务了。接下来你将要学习一些强大的选项，对 MongoDB 的备份和恢复功能进行调整，以便符合个人的特殊需求。

mongodump 工具中包含的选项如图 9-1 所示(可通过在 MongoDB 2.5.3 中运行 help 选项得到)。

```

$ mongodump --help
Export MongoDB data to BSON files.

Options:
  --help                produce help message
  -v [ --verbose ]      be more verbose (include multiple times
                        for more verbosity e.g. -vvvv)
  --quiet               silence all non error diagnostic
                        messages
  --version             print the program's version and exit
  -h [ --host ] arg     mongo host to connect to (<set
                        name>/s1,s2 for sets)
  --port arg            server port. Can also use --host
                        hostname:port
  --ipv6               enable IPV6 support (disabled by
                        default)
  -u [ --username ] arg username
  -p [ --password ] arg password
  --authenticationDatabase arg user source (defaults to dbname)
  --authenticationMechanism arg (=MONGODB-CR) authentication mechanism
  --dbpath arg         directly access mongod database files
                        in the given path, instead of
                        connecting to a mongod server - needs
                        to lock the data directory, so cannot
                        be used if a mongod is currently
                        accessing the same path
  --directoryperdb     each db is in a separate directory
                        (relevant only if dbpath specified)
  --journal            enable journaling (relevant only if
                        dbpath specified)
  -d [ --db ] arg      database to use
  -c [ --collection ] arg collection to use (some commands)
  -o [ --out ] arg (=dump) output directory or "-" for stdout
  -q [ --query ] arg   json query
  --oplog             Use oplog for point-in-time
                        snapshotting
  --repair            try to recover a crashed database
  --forceTableScan    force a table scan (do not use
                        $snapshot)

Auditing Options:
  --auditlog arg       turn on auditing and specify output for log: textfile,
                        bsonfile, syslog, console
  --auditPath arg     full filespec for audit log file
  --auditFilter arg   filter spec to screen audit records

```

图 9-1 mongodump 工具中包含的选项

这里列出的大多数选项都是自说明的，下面这几个例外：

- **--dbpath arg**: 如果有大量需要备份的数据和快速驱动器，并且不考虑索引的备份，那么推荐通过直接将 MongoDB 服务器上的数据文件复制到备份媒介中的方式备份数据库。通过该选项可以直接备份服务器上的数据文件，但只有在服务器离线或写冻结时才可使用(详细信息请参阅本章稍后的 9.6 节“备份大数据库”)。
- **--directoryperdb**: 结合--dbpath 命令行选项一起使用，可以指定正在备份的 MongoDB 服务器将每个数据库中的数据分别放到不同的目录中。默认情况下，MongoDB 将把它

的数据文件放在一个目录中。如果已经将服务器配置为这种模式，那就应该使用该选项。

- `-o [--out] arg`: 使用该选项指定数据库转储文件的存放目录。默认情况下，`mongodump` 工具将在当前目录中创建一个名为 `/dump` 的文件夹，并将转储文件保存在其中。通过使用 `-o/--out` 选项可以选择其他路径用于放置转储文件。
- `--authenticationDatabase arg`: 指定保存用户凭证的数据库。在未使用该选项的情况下，`mongodump` 默认将使用 `-db` 选项指定的数据库。
- `--authenticationMechanism arg`: 默认使用 MongoDB 的挑战/响应机制(用户名/密码)。该命令用于将验证模式切换为 MongoDB 企业版的 Kerberos 验证。

9.4 恢复单个数据库或集合

你之前已经学习了如何使用 `mongodump` 工具备份单个数据库或集合，`mongorestore` 工具同样也可以实现相同的操作。如果备份目录中含有目标集合或数据库的备份文件，那就可以使用 `mongorestore` 恢复该数据库或集合；并不需要恢复备份文件中的所有内容。如果愿意的话，可以单独恢复每个数据库或集合。

`mongorestore` 中的可用选项如图 9-2 所示。

```

$ mongorestore --help
Import BSON files into MongoDB.

usage: mongorestore [options] [directory or filename to restore from]

Options:
  --help                produce help message
  -v [ --verbose ]     be more verbose (include multiple times
                        for more verbosity e.g. -vvvv)
  --quiet              silence all non error diagnostic
                        messages
  --version            print the program's version and exit
  -h [ --host ] arg   mongo host to connect to (<set
                        name>/s1,s2 for sets)
  --port arg          server port. Can also use --host
                        hostname:port
  --ipv6              enable IPv6 support (disabled by
                        default)
  -u [ --username ] arg username
  -p [ --password ] arg password
  --authenticationDatabase arg user source (defaults to dbName)
  --authenticationMechanism arg (=MONGODB-CR) authentication mechanism
  --dbpath arg        directly access mongod database files
                        in the given path, instead of
                        connecting to a mongod server - needs
                        to lock the data directory, so cannot
                        be used if a mongod is currently
                        accessing the same path
  --directoryperdb   each db is in a separate directory
                        (relevant only if dbpath specified)
  --journal           enable journaling (relevant only if
                        dbpath specified)
  -d [ --db ] arg    database to use
  -c [ --collection ] arg collection to use (some commands)
  --objcheck         validate object before inserting
                        (default)
  --noobjcheck       don't validate object before inserting
  --filter arg       filter to apply before inserting
  --drop            drop each collection before import
  --oplogReplay     replay oplog for point-in-time restore
  --oplogLimit arg  include oplog entries before the
                        provided Timestamp (seconds[.ordinal])
                        during the oplog replay; the ordinal
                        value is optional
  --keepIndexVersion don't upgrade indexes to newest version
  --noOptionsRestore don't restore collection options
  --noIndexRestore  don't restore indexes
  -w arg (=0)       minimum number of replicas per write

Auditing Options:
  --auditLog arg    turn on auditing and specify output for log: textfile,
                        bsonfile, syslog, console
  --auditPath arg  full filespec for audit log file
  --auditFilter arg filter spec to screen audit records
    
```

图 9-2 `mongorestore` 工具中的可用选项

通过之前对 `mongodump` 工具的学习，你应该可以识别其中的大多数选项；不过下面的两

个选项值得单独一提：

- `--drop`：该选项将告诉 `mongorestore` 在恢复集合之前删除现有的集合。这将保证不会出现重复的数据。如果未使用该选项，被恢复的数据将被添加(插入)到目标集合中。
- `--noobjcheck`：该选项将告诉 `mongorestore` 忽略将对象插入目标集合之前的验证步骤。

9.4.1 恢复单个数据库

通过使用 `mongorestore` 工具的 `-d` 选项可以恢复单个数据库。正如之前所提到的，如果数据库在 MongoDB 服务器上已经存在，不要忘记使用 `--drop` 选项：

```
$cd ~/testmongobackup
$mongorestore -d blog --drop
```

9.4.2 恢复单个集合

通过使用类似的语法也可以恢复数据库中的单个集合；区别在于还需要使用 `-c` 选项指定集合的名称，如下所示：

```
$cd ~/testmongobackup
$mongorestore -d blog -c posts --drop
```

9.5 自动化备份

对于小型的 MongoDB 服务器或开发服务器，简单运行 `mongodump` 工具并保存结果就足以满足即时备份的需求。例如，针对 Mac OS X 工作站的通用实践是让 Time Machine(Mac 备份工具)存储备份。

对于生产环境中的服务器，你希望自动备份它们的数据；定时备份可以帮助防止出现问题，或者在出现问题的时候恢复到正常状态。这是真的，因为不仅服务器安装设置可能出现问题(例如，数据库损坏)，用户也可能不小心损坏或破坏数据。

下面学习一些可用于自动化备份的脚本。

9.5.1 使用本地数据存储

如果系统中具有大的备份驱动器或者可以通过 NFS 或 SMB 挂载一个外部文件系统，那么可以使用一个简单的脚本在指定的目录中创建归档文件。下面的脚本很容易设置，只需要将脚本顶部的变量改为本地系统的路径即可：

```
#!/bin/bash
#####
# Edit these to define source and destinations

MONGO_DBS=""
BACKUP_TMP=~/.tmp
BACKUP_DEST=~/.backups
MONGODUMP_BIN=/usr/bin/mongodump
TAR_BIN=/usr/bin/tar

#####
```

```

BACKUPFILE_DATE=`date +%Y%m%d-%H%M`

# _do_store_archive <Database><Dump_dir><Dest_Dir><Dest_file>

function _do_store_archive {
    mkdir -p $3
    cd $2
    tar -cvzf $3/$4 dump
}

# _do_backup <Database name>

function _do_backup {
    UNIQ_DIR="$BACKUP_TMP/$1"`date +%s``
    mkdir -p $UNIQ_DIR/dump
    echo "dumping Mongo Database $1"
    if [ "all" = "$1" ]; then

        $MONGODUMP_BIN -o $UNIQ_DIR/dump

    else

        $MONGODUMP_BIN -d $1 -o $UNIQ_DIR/dump

    fi
    KEY="database-$BACKUPFILE_DATE.tgz"
    echo "Archiving Mongo database to $BACKUP_DEST/$1/$KEY"
    DEST_DIR=$BACKUP_DEST/$1

    _do_store_archive $1 $UNIQ_DIR $DEST_DIR $KEY

    rm -rf $UNIQ_DIR
}

# check to see if individual databases have been specified, otherwise backup the whole server
# to "all"

if [ "" = "$MONGO_DBS" ]; then
    MONGO_DB="all"
    _do_backup $MONGO_DB
else
    for MONGO_DB in $MONGO_DBS; do
        _do_backup $MONGO_DB
    done
fi

```

表 9-1 列出了使用该备份脚本所需修改的变量。

表 9-1 本地数据存储备份脚本中使用的变量

变 量	描 述
MONGO_DBS	保持该变量为空，本地服务器上的所有数据库都将被备份。也可以在该变量中设置一个数据库的列表，用于备份指定的数据库(“db1 db2 db3”)
BACKUP_TMP	将该变量设置为保存备份文件的临时目录。在归档完成之后，删除该目录中的临时数据。要选择一个与脚本相关的合适目录。例如，如果你在自己的本地账户中使用脚本创建备份，那就使用目录~/tmp；如果在系统的 cronjob 中以系统账户运行该脚本，那么使用/tmp。在 Amazon EC2 实例上，你可能就应该使用/mnt/tmp，这样可以保证该文件夹不会被创建在系统的根分区上(它的空间非常小)

(续表)

变 量	描 述
BACKUP_DEST	该变量保存了备份文件的目标目录，所有其他文件夹都将在该文件夹下创建。同样，要选择一个与脚本相关的合适目录
MONGODUMP_BIN	由于运行备份脚本的账户可能并不拥有完整的路径设置，最好使用该变量指定二进制文件的全路径。通过在终端窗口中输入 <code>which mongodump</code> 可以获取系统中该工具的正确路径
TAR_BIN	使用该变量设置 <code>tar</code> 二进制文件的全路径；在终端窗口中输入 <code>which tar</code> 可获取此路径

现在可以使用该脚本备份数据库了；脚本运行之后将在指定的 `BACKUP_DEST` 目录中创建一组归档备份文件。这些文件的命名格式为：

Database Name/database-YYYYMMDD-HHMM.tgz

例如，下面的脚本将显示本章 `test` 数据库的备份文件名：

```
Backups:$ tree
.
|-- blog
|   |-- database-20100611-1144.tgz
|   `-- database-20100611-1145.tgz
`-- all
    |-- database-20100611-1210.tgz
    |-- database-20100611-1221.tgz
    |-- database-20100611-1222.tgz
    |-- database-20100611-1224.tgz
    `-- database-20100611-1233.tgz
```

当然，你也需要安装该脚本。如果希望每天运行该脚本，那就将它添加到 `/etc/cron.daily` 中，然后重启 `cron` 服务让它生效。这种方式在大多数的 Linux 发行版本中都可以正常运行，例如 Ubuntu、Fedora、CentOS 和 Red Hat。如果希望备份的频率低一些，就将脚本移到 `/etc/cron.weekly` 或 `/etc/cron.monthly`。对于更频繁的备份，还可以使用 `/etc/cron.hourly`。

9.5.2 使用远端数据存储(基于云)

之前讲述的脚本有一个独立的函数用于创建和存储归档文件。这样在修改该脚本，让它将备份归档文件保存到外部数据存储时就相对容易一些。表 9-2 仅提供了一些样例，事实上还有许多其他的机制可供选择。

表 9-2 远端(基于云)备份存储选项

方 法	描 述
使用 <code>rsync/ftp/tftp</code> 或 <code>scp</code> 将文件传输到另一台服务器	可以使用 <code>rsync</code> 将归档文件移到一台备份存储服务器上
S3 存储	如果你的系统运行在 EC2 实例上，那么 S3 存储是存储备份文件的好地方，因为花费较低并且 Amazon 将会为它们创建冗余备份

下面我们将讲解如何使用 S3 的方式存储备份文件；不过，同样的规则也适用于任何其他机制。

以下样例使用的 s3cmd 工具(用 Python 编写)可从 <http://s3tools.org> 获得。在 Ubuntu 中，可以通过 `sudo apt-get install s3cmd` 命令安装该脚本；对于 Mac OS X，该脚本位于 MacPorts 集合中。对于 Fedora、CentOS 和 Red Hat，可从 <http://s3tools.org> 获得 yum 包，然后使用 yum 安装它。

在安装该包之后，运行 `s3cmd -configure` 以设置你的 Amazon S3 凭证。注意只需要提供两个密钥：AWS_ACCESS_KEY 和 AWS_SECRET_ACCESS_KEY。s3cmd 工具将创建一个包含了必需信息的配置文件：`~/.s3cfg`。

下面是为了使用 S3 而做出的修改：

```
# _do_store_archive <Database><Dump_dir><Dest_Dir><Dest_file>

BACKUP_S3_CONFIG=~/.s3cfg
BACKUP_S3_BUCKET=somecompany.somebucket
S3CMD_BIN=/usr/bin/s3cmd

function _do_store_archive {
    UNIQ_FILE="aws"`date "+%s"`
    cd $2
    tar -cvzf $BACKUP_TMP/$UNIQ_FILE dump
    $S3CMD_BIN --config $BACKUP_S3_CONFIG put $BACKUP_TMP/$UNIQ_FILE \
    s3://$BACKUP_S3_BUCKET/$1/$4
    rm -f $BACKUP_TMP/$UNIQ_FILE
}
```

表 9-3 列出了为运行修改后的脚本所需要配置的变量：

表 9-3 为修改后的备份脚本配置变量

变 量	描 述
BACKUP_S3_CONFIG	运行 <code>s3cmd -configure</code> 时创建的配置文件路径，包含个人 S3 账户的详细信息
BACKUP_S3_BUCKET	设置希望用于存储备份文件的 bucket
S3CMD_BIN	s3cmd 可执行程序的路径，同样，可以使用 <code>which s3cmd</code> 在系统中找到它

9.6 备份大数据库

在使用大数据库系统时，如何创建出高效的备份解决方案可能是一个问题。通常备份大数据库的时间是非常长的，甚至可能需要数个小时才能完成。在此期间，还需要将数据库维护在一致的状态，因此备份中不能出现在不同时间点复制的文件。数据库备份系统有一个必杀技，

就是时间点快照，它的速度非常快。快照完成的速度越快，数据库服务器需要被冻结的时间就越短。

9.6.1 使用隐藏的辅助服务器备份数据

执行大数据备份的一种技术是：从隐藏的辅助服务器备份数据，在备份期间可以冻结该服务器。在备份完成之后，重启该辅助服务器，让它从应用的其他服务器获得最新的数据。

在 MongoDB 中创建隐藏辅助服务器是非常简单的，并且可以使用 MongoDB 的复制机制来保证它与主服务器一致。它的配置也相对比较容易(关于如何设置隐藏的辅助服务器请参阅第 11 章)。

9.6.2 使用日志文件系统创建快照

许多现代的卷管理系统都有在任意时间点创建驱动状态快照的能力。使用文件系统快照是为 MongoDB 实例创建备份中的速度最快也是最高效的方式。如何设置这些系统超出了本书的范围，不过我们可以展示如何将 MongoDB 服务器保持一致的状态下，此时磁盘中的所有数据也应该处于一致的状态。我们还将展示如何阻塞写入操作，这样接下来的数据修改就不会被写入到磁盘中，而是缓存在内存里。

通过快照可以读取创建快照时驱动体的准确状态。系统卷或文件系统管理器将保证磁盘上的所有数据块，在创建快照之后发生的修改都不会被写入原来的位置；这样可以保留所有用于读取的磁盘数据。通常，使用快照的过程如下：

- (1) 创建快照。
- (2) 从快照复制数据或者将快照恢复到另一个卷中，取决于个人的卷管理器。
- (3) 释放快照；这样将会释放所有预留的磁盘块，它们不需要再返回驱动器的空闲空间链中。
- (4) 在服务器仍然运行时，备份所有的副本数据。

这种方式最棒的地方在于创建快照时，数据仍然可以继续被读取。

支持该功能的卷管理器包括：

- Linux 和 LVM 卷管理系统
- Sun ZFS
- Amazon EBS 卷
- 使用卷影副本的 Windows Server

大多数卷管理都具有快速创建快照的能力，通常只需要数秒时间，即使是在数据量非常巨大的情况下。此时卷管理器并未真正地复制数据；相反，它们将在驱动中插入一个标签，从而使你可以读取快照发生时驱动体的状态。

一旦备份系统完成对驱动快照的读取，接下来就可以将那些发生过修改的旧数据块放入到驱动的空闲空间链中(或文件系统所使用的任何一种标识空闲空间的方式)。

为了使这种创建备份的方式高效，我们必须将 MongoDB 日志文件保存在相同的设备上，或者使 MongoDB 将所有未完成的磁盘写入输出到磁盘，这样才可以创建快照。强制 MongoDB

完成刷新的特性被称为 `fsync`；阻塞写入的函数被称为 `lock`。MongoDB 具有同时执行这两个操作的能力，这样在 `fsync` 执行之后，就不会再有磁盘写入发生，直至锁被释放。通过将日志保存在同一设备上或者同时执行 `fsync` 和 `lock`，我们可以保证磁盘上数据库的镜像处于一致状态，并保证在完成快照之前它们仍然保持一致的状态。

使用下面的命令可以使 MongoDB 进入 `fsync` 和 `lock` 状态：

```
$mongo
> use admin
> db.fsyncLock()
{
  "info" : "now locked against writes",
  "ok" : 1
}
```

使用下面的命令可以检查锁的当前状态：

```
$mongo
> use admin
> db.currentOp()
{
  "inprog" : [
  ],
  "fsyncLock" : 1
}
```

状态`"fsyncLock": 1`表示 MongoDB 的 `fsync` 进程(负责将修改写入磁盘)正在阻塞写入操作。此时，就可以使用必要的命令通知卷管理器创建 MongoDB 用于存储文件的文件夹的快照。一旦快照完成，就可以使用下面的命令释放锁：

```
$mongo
> db.fsyncUnlock();
{ "ok" : 1, "info" : "unlock requested" }
```

注意在锁释放之前可能有小的延迟；不过，可以使用 `db.currentOp()` 函数检查执行结果。当锁最终被释放时，`db.currentOp()` 将返回下面的结果：

```
$mongo
> use admin
> db.currentOp()
{ "inprog" : [] }
```

`{"inprog":[]}` 意味着锁已经被释放，MongoDB 可以开始向磁盘写入数据。

现在插入快照标签，然后就可以使用卷管理器提供的工具将快照的内容复制到合适的地方，用于备份。不要忘记在备份完成之后释放快照。

关于快照的更多详细信息请访问以下链接：

- <http://docs.mongodb.org/manual/tutorial/backup-databases-with-filesystem-snapshots/>
- http://tldp.org/HOWTO/LVM-HOWTO/snapshots_backup.html
- <http://docs.huihoo.com/opensolaris/solaris-zfs-administration-guide/html/ch06.html>
- http://support.rightscale.com/09-Clouds/AWS/02-Amazon_EC2/EBS/Create_an_EBS_Snapshot

9.6.3 使用卷管理器时的磁盘布局

某些卷管理器可以创建某个分区上子目录的快照，但大多数都无法做到，所以最好将计划用于存储 MongoDB 数据的卷挂载到文件系统上一个合适的位置(例如/mnt/mongodb)，并使用服务器配置选项将数据目录、配置文件和任何其他 MongoDB 相关的文件(例如日志)存储在该目录中。

这意味着创建卷的快照时，可以捕捉到服务器的完整状态，包括它的配置。甚至将 MongoDB 服务器的二进制文件也存储在该卷中也是个好主意，因为这样备份中将包含一套完全协调的组件。

9.7 将数据导入 MongoDB

有时可能将大量数据导入 MongoDB 中用作参考数据。这些数据可能包括邮编表、IP 地理定位表、零件目录等。

MongoDB 中包含了批量加载器 `mongoimport`，它被设计用于将数据直接导入服务器的特定集合中；它不同于 `mongorestore`，因为 `mongorestore` 的目的在于从备份文件恢复 MongoDB 数据。

`mongoimport` 工具可以加载以下三种文件格式中任何一种文件格式的数据：

- **CSV**：在此种文件格式中，每行代表着一个文档，字段之间由逗号分隔。
- **TSV**：该文件格式类似于 CSV；不过，它使用 Tab 字符作为分隔符。该格式非常常见，因为它不要求对任何文本字符进行转义(除了换行符)。
- **JSON**：该文件格式每行都包含一块 JSON，代表一个文档。与其他格式不同，JSON 可以支持可变模式的文档。

该工具的使用相当直观。它将接受一个使用了以上三种格式中任何一种格式的文件、一个字符串或含有一组列头的文件(这些组成了 MongoDB 文档的元素名)，以及几个用于控制如何解释数据的服务器选项。图 9-3 展示了如何使用 `mongoimport` 工具。

下列选项值得进行详细讲解：

- **--headerline**：使用文件的第一行作为字段名称列表。注意该选项只可作用于 CSV 和 TSV 格式。
- **-ignoreblanks**：不导入任何空字段。如果字段为空，文档中将不会创建该元素；如果不使用该选项，那么文档中将创建一个使用了该列名的空元素。
- **--drop**：删除集合，并使用导入的数据重新创建该集合；否则，导入的数据将被追加到集合中。

使用 `mongoimport` 导入数据时还可以使用 `-d` 和 `-c` 选项指定数据库名和集合名，如下所示：

```
$mongoimport -d blog -c tagcloud --type csv --headerline < csvimportfile.csv
```

```

$mongoimport --help
Import CSV, TSV or JSON data into MongoDB.

When importing JSON documents, each document must be a separate line of the input file.

Example:
mongoimport --host myhost --db my_cms --collection docs < mydocfile.json

Options:
--help                produce help message
-v [ --verbose ]      be more verbose (include multiple times
                      for more verbosity e.g. -vvvv)
--quiet              silence all non error diagnostic
                      messages
--version            print the program's version and exit
-h [ --host ] arg    mongo host to connect to ( <set
                      name>/s1,s2 for sets)
--port arg           server port. Can also use --host
                      hostname:port
--ipv6              enable IPv6 support (disabled by
                      default)
-u [ --username ] arg username
-p [ --password ] arg password
--authenticationDatabase arg user source (defaults to dbname)
--authenticationMechanism arg (=MONGODB-CR)
                      authentication mechanism
--dbpath arg        directly access mongod database files
                      in the given path, instead of
                      connecting to a mongod server - needs
                      to lock the data directory, so cannot
                      be used if a mongod is currently
                      accessing the same path
--directoryperdb    each db is in a separate directory
                      (relevant only if dbpath specified)
--journal           enable journaling (relevant only if
                      dbpath specified)
-d [ --db ] arg     database to use
-c [ --collection ] arg collection to use (some commands)
-f [ --fields ] arg comma separated list of field names
                      e.g. -f name,age
--fieldFile arg     file with field names - 1 per line
--ignoreBlanks      if given, empty fields in csv and tsv
                      will be ignored
--type arg          type of file to import. default: json
                      (json,csv,tsv)
--file arg          file to import from; if not specified
                      stdin is used
--drop             drop collection first
--headerline       first line in input file is a header
                      (CSV and TSV only)
--upsert           insert or update objects that already
                      exist
--upsertFields arg  comma-separated fields for the query
                      part of the upsert. You should make
                      sure this is indexed
--stopOnError      stop importing at first error rather
                      than continuing
--jsonArray        load a json array, not one item per
                      line. Currently limited to 16MB.

Auditing Options:
--auditLog arg      turn on auditing and specify output for log: textfile,
                      bsonfile, syslog, console
--auditPath arg    full filespec for audit log file
--auditFilter arg  filter spec to screen audit records
    
```

图 9-3 mongoimport 工具中的可用选项

9.8 从 MongoDB 导出数据

工具 mongoexport 类似于 mongoimport，顾名思义，mongoexport 用于从现有的 MongoDB 集合中导出数据到文件中。这是将 MongoDB 实例中的数据导出成一种可由其他数据库或电子

表格应用读取的文件格式的最好方式之一。图 9-4 展示了 mongoexport 工具的使用方法。

```

$mongoexport --help
Export MongoDB data to CSV, TSV or JSON files.

Options:
  --help                produce help message
  -v [ --verbose ]     be more verbose (include multiple times
                        for more verbosity e.g. -vvvv)
  --quiet               silence all non error diagnostic
                        messages
  --version             print the program's version and exit
  -h [ --host ] arg    mongo host to connect to ( <set
                        name>/s1,s2 for sets)
  --port arg           server port. Can also use --host
                        hostname:port
  --ipv6               enable IPv6 support (disabled by
                        default)
  -u [ --username ] arg username
  -p [ --password ] arg password
  --authenticationDatabase arg user source (defaults to dbname)
  --authenticationMechanism arg (=MONGODB-CR) authentication mechanism
  --dbpath arg         directly access mongod database files
                        in the given path, instead of
                        connecting to a mongod server - needs
                        to lock the data directory, so cannot
                        be used if a mongod is currently
                        accessing the same path
  --directoryperdb    each db is in a separate directory
                        (relevant only if dbpath specified)
  --journal            enable journaling (relevant only if
                        dbpath specified)
  -d [ --db ] arg     database to use
  -c [ --collection ] arg collection to use (some commands)
  -f [ --fields ] arg comma separated list of field names
                        e.g. -f name,age
  --fieldFile arg     file with field names - 1 per line
  -q [ --query ] arg  query filter, as a JSON string
  --csv               export to csv instead of json
  -o [ --out ] arg    output file; if not specified, stdout
                        is used
  --jsonArray         output to a json array rather than one
                        object per line
  -k [ --slaveOk ] arg (=1) use secondaries for export if
                        available, default true
  --forceTableScan    force a table scan (do not use
                        $snapshot)
  --skip arg (=0)     documents to skip, default 0
  --limit arg (=0)    limit the numbers of documents
                        returned, default all

Auditing Options:
  --auditlog arg      turn on auditing and specify output for log: textfile,
                        bsonfile, syslog, console
  --auditPath arg    full filespec for audit log file
  --auditFilter arg  filter spec to screen audit records

```

图 9-4 mongoexport 工具的可用选项

需要注意的选项有：

- **-q**：指定用于选择输出记录的查询。该查询可以是任何 JSON 查询字符串(但不是 JavaScript 查询字符串，因为它通常无法如预期一样工作)，这些查询字符串应该可以通过使用 `db.collection.find()` 函数选择数据库中记录的一个子集。不指定该选项或者将它的值设置为 `{}`，`mongoexport` 工具将输出所有的记录。
- **-f**：列出需要被导出的数据库元素名。

下面的样例演示了如何使用 `mongoexport` 工具的选项：

```
$mongoexport -d blog -c posts -q {} -f _id,Title,Message,Author --csv >blogposts.csv
```

```
connected to: 127.0.0.1
exported 1 records
```

9.9 通过限制对 MongoDB 服务器的访问保护数据安全

在某些情况下，应用中可能会处理一些敏感数据，例如社交网络中的用户记录或电子商务应用中的支付细节。在许多情况下，都有规则强制要求保证限制对数据库系统中敏感数据的访问。

MongoDB 支持简单的基于角色的认证系统，通过该系统可以控制用户对数据库的访问以及他们被授予的访问级别。

大多数修改配置数据或者对结果进行大幅度调整的 MongoDB 服务器命令都被限制在专门的 `admin` 数据库中执行，该数据库将在每个新的 MongoDB 安装时自动创建。

在可以使用这些命令之前，必须使用 `use admin` 命令切换到 `admin` 数据库。在接下来的内容中会标识只能在 `admin` 数据库中运行的命令，这样你在使用该命令之前就知道需要切换到 `admin` 数据库。本章假设在必要的时候，可以选择数据库和完成认证。

默认情况下，MongoDB 不使用任何认证方式。任何可以访问网络的人都可以连接到服务器并执行命令。不过，可以在任何数据库中添加用户，这样就可以对 MongoDB 进行配置，使得在访问数据库时既要求有连接，还需要进行控制台验证。这是限制对管理函数访问的推荐方式。

9.10 使用认证保护服务器

MongoDB 支持一个简单的认证模型，管理员通过它可以根据用户的不同，设置不同的数据库访问权限。

MongoDB 支持对每个数据库的访问进行单独控制，访问控制信息被存储在特有的 `system.users` 集合中。对于希望访问两个数据库(例如，`db1` 和 `db2`)的普通用户，他们的凭证和权限必须被同时添加到两个数据库中。

如果在不同的数据库中为同一用户分别创建了登录和访问权限，这些记录不会相互同步。换句话说，修改一个数据库中的用户密码不会影响另一个数据库中的用户密码。不过，MongoDB 在 2.4 版本中引入了一个新的机制，通过它可以支持代理凭证。以这种方式使用用户凭证时，你可以创建一个具有密码的主用户。然后在另一个数据库中创建用户，并表明该用户已经存在于主数据库中，并且该用户的凭证应该被用于当前数据库的认证。

该规则还有最后一个例外：任何被添加到 `admin` 数据库中的用户在所有数据库中都拥有相同的访问权限；不需要为这样的用户单独赋予权限。

注意：

即便在添加 `admin` 用户之前已经启用了认证，也仍然可以通过本机访问数据库，即从 MongoDB 实例所在的服务器连接该服务器。通过该安全特性，管理员可以在启用认证之后创建用户。

9.10.1 添加 admin 用户

添加 admin 用户非常简单，只需要切换到 admin 数据库，然后使用 addUser() 函数即可：

```
$mongo
> use admin
> db.addUser({user : "admin", pwd: "pass", roles: [ "readWrite", "dbAdmin" ] })
{
  "user" : "admin",
  "pwd" : "e4e538f5dcb52537cad02bbf8491693c",
  "roles" : [
    "readWrite",
    "dbAdmin"
  ],
  "_id" : ObjectId("5239915b1ce3dc1efebb3c84")
}
```

此时，只需要添加单个 admin 用户即可；一旦该用户定义成功，就可以使用该账户在 admin 数据库中添加其他 admin 用户，或在任何其他数据库中添加普通用户。

9.10.2 启用认证

现在修改服务器配置以启用认证。停止服务器并在启动参数中添加--auth。

如果是使用包安装器(例如 yum 或 aptitude)安装的 MongoDB，那么通常可以编辑/etc/mongodb.conf 以启用 auth=true。接下来，使用下面的命令重启服务器并启用认证：

```
$sudo service mongodb restart
```

另外，除了认证，还可以使用 keyfile，它是一个包含了预分享密钥(一些描述信息)的文件，通过它可以确认 MongoDB 节点之前的通信。只需要创建一个简单的文件，并在其中添加一个词组或字符串即可。然后如同启用认证的方式一样，添加选项 keyfile=/path/to/keyfile。甚至可以移除 auth=true 选项，只使用 keyfile 完成认证。

9.10.3 在 mongo 控制台中执行认证

在 admin 数据库中运行受限的命令之前，首先需要将自己认证为 admin 用户，如下所示：

```
$mongo
> use admin
switched to db admin
> show collections
Sun May 26 17:22:26.132 JavaScript execution failed: error: {
  "$err" : "not authorized for query on admin.system.namespaces",
  "code" : 16550
} at src/mongo/shell/query.js:L131 }
> db.auth("admin", "pass");
1
```

此时，mongo 控制台将会输出 1(认证成功)或 0(认证失败)：

```
1
> show collections
system.indexes
system.users
```


如果认证已经成功，那么可以执行用户权限范围内的所有操作。

如果认证失败，那么需要检查用户名/密码是否正确，以及 `admin` 用户是否已经被正确地添加到 `admin` 数据库中。重置服务器以停用认证，然后使用下面的命令列出 `admin` 数据库的 `system.users` 集合中的所有内容：

```
$mongo
> use admin
> db.system.users.find()
{ "_id" : ObjectId("5239915b1ce3dc1efebb3c84"), "user" : "admin", "pwd" :
"e4e538f5dcb52537cad02bbf8491693c", "roles" : [ "readWrite", "dbAdmin" ] }
```

注意：

如果使用 `admin` 凭证访问 `admin` 之外的数据库，那么必须首先针对 `admin` 数据库进行认证。否则，你将无法访问系统中的任何其他数据库。

`mongo` 控制台显示了 `user` 集合中的内容，通过这种方式可以找到准确的 `userid`，而密码将显示为原始密码的 MD5 哈希值：

```
$ mongo
> use blog
switched to db blog
> show collections
Wed Sep 18 21:42:51.855 JavaScript execution failed: error: {
  "$err" : "not authorized for query on blog.system.namespaces",
  "code" : 13
} at src/mongo/shell/query.js:L128
> db.auth("admin", "pass")
Error: 18 { code: 18, ok: 0.0, errmsg: "auth fails" }
0
> use admin
switched to db admin
> db.auth("admin", "pass")
1
> use blog
switched to db blog
> show collections
system.indexes
system.users
authors
posts
tagcloud
```

9.10.4 MongoDB 用户角色

目前在 MongoDB 的权限框架中，用户可使用的角色有：

- **Read:** 允许用户读取指定的数据库。
- **readWrite:** 授予用户指定数据库的读写权限。
- **dbAdmin:** 允许用户在指定的数据库中执行管理函数，例如创建或删除索引、查看统计或访问 `system.profile` 集合。
- **userAdmin:** 允许用户向 `system.users` 集合中写入。具有该权限的用户可以在该数据库中创建、删除和管理用户。

- **clusterAdmin**: 只在 **admin** 数据库中可用。为用户赋予所有分片和复制集相关函数的完全管理权限。
- **readAnyDatabase**: 只在 **admin** 数据库中可用。为用户赋予所有数据库的读权限。
- **readWriteAnyDatabase**: 只在 **admin** 数据库中可用。为用户赋予所有数据库的读写权限。
- **userAdminAnyDatabase**: 只在 **admin** 数据库中可用。为用户赋予所有数据库的 **userAdmin** 权限。
- **dbAdminAnyDatabase**: 只在 **admin** 数据库中可用。为用户赋予所有数据库的 **dbAdmin** 权限。

9.10.5 代理凭证

正如之前提到的，从 MongoDB 的 2.4 版本开始，可以使用一个主用户，接下来创建的用户可以使用该主用户的凭证用于认证，该特性被称为代理凭证。例如在 **foo** 数据库中创建用户 **test**:

```
> use foo
> db.addUser( user : "test", pwd: "password", roles: ["readWrite" ])
```

现在，接着在 **bar** 数据库中创建相同的 **test** 用户。运行下面的命令将会创建一个用户(当然，该用户拥有 **bar** 数据库的 **userAdmin** 权限)，该用户会使用 **foo** 数据库中用户定义的密码作为自己的密码:

```
> use bar
> db.system.users.insert{ user: "test", roles: ["read"], userSource: "foo"}
```

注意新创建的用户被赋予只读权限“**read**”。这是因为赋予 **bar** 数据库中 **test** 用户的权限仍然基于 **bar** 数据库用户的凭证。我们只是从 **foo** 数据库获得了其他所需的信息(即密码)。通过使用代理凭证，可以在一个位置对所有用户的用户名和密码进行更新。

9.10.6 修改用户凭证

修改用户访问权限或密码非常简单。再次执行 **addUser()** 函数即可，这将在 MongoDB 中更新已有的用户记录。理论上讲，可以使用任何数据操作命令修改用户记录；不过，只有 **addUser()** 函数可以创建密码字段。

下面通过显示 **addUser()** 函数的内容来学习它的工作方式:

```
$mongo
> use admin
> db.addUser
function () {
  if (arguments.length == 0) {
    throw Error("No arguments provided to addUser");
  }
  if (typeof arguments[0] == "object") {
    this._addUser.apply(this, arguments);
  } else {
    this._addUserV22.apply(this, arguments);
  }
}
```

`addUser()`只是一个使用 JavaScript 定义的函数。如果希望创建一个 Web 表单用于在数据库中添加用户,或者希望从另一个凭证源将用户导入系统,那么此时了解密码的构造就非常有用。

大多数 mongo 控制台函数都可以通过这种方式显示出来,这样我们就可以了解它们的工作细节。

9.10.7 添加只读用户

函数 `addUser()`包含一个额外的参数,可用于创建只读用户。如果一个进程使用该用户的凭证认证之后,并尝试执行一些会修改数据库内容的命令,那么 MongoDB 客户端将会抛出异常。下面的例子演示了如何添加一个用户来访问数据库(用于状态监测或报告目的):

```
$mongo
> use admin
switched to db admin
> db.addUser(user : "admin", pwd: "pass", roles: [ "read" ])
1
> use blog
switched to db blog
> db.addUser("shadycharacter","shadypassword", true)
```

9.10.8 删除用户

为了从数据库中删除用户,可以使用作用于集合的普通 `remove()`函数。下面的样例将删除刚刚添加的用户;注意在删除用户之前,必须针对 `admin` 数据库进行认证:

```
$mongo
> use admin
switched to db admin
> db.auth("admin","pass")
1
> use blog
switched to db blog
> db.removeUser("shadycharacter")
```

9.10.9 在 PHP 应用中认证连接

在第 4 章,你已经学习了如何使用 PHP 创建到 MongoDB 数据库的连接。一旦在服务器上启用认证,PHP 在执行服务器命令之前也必须提供凭证。下面的样例将演示如何打开到数据库的认证连接:

```
<?php

//建立数据库连接
$connection = new Mongo();
$db = $connection->selectDB("admin");
$result = $db->authenticate("admin", "pass");
if(!$result['ok']){
    //错误处理
    die("Authentication Error: {$result['errmsg']}");
}
```

```
//关闭数据库连接

$conn->close();

?>
```

9.11 管理服务器

作为管理员，必须保证 MongoDB 服务器的平稳和可靠运行。

必须周期性地对服务器进行优化，从而使服务器达到最优性能，或对它们进行重新配置，以更好地匹配你所使用的环境。为此，必须熟悉大量用户管理和服务器控制方面的函数。

9.11.1 启动服务器

大多数现代 Linux 发行版本都包含一组/etc/init.d 脚本，用于管理服务。如果使用的是 MongoDB 网站上的发行包之一来安装 MongoDB 服务器(关于这些包的更多细节信息请参阅第2章)，那么管理服务器的 init.d 脚本就已经被安装到系统中。

在 Ubuntu、Fedora、CentOS 和 RedHat 中可以使用 service 命令启动、停止和重启服务器，如下所示：

```
$sudo service mongodb start
mongodb start/running, process 3474
$sudo service mongodb stop
mongodb stop/waiting

$sudo service mongodb restart
mongodb start/running, process 3474
```

如果没有可用的初始化脚本，那么可以打开一个终端窗口并输入下面的命令，以手动的方式启动 MongoDB 服务器：

```
$ mongod
Fri May 24 15:06:20.475 [initandlisten] MongoDB starting : pid=97163 port=27017
dbpath=/var/lib/mongodb 64-bit host=Pixl.local
Fri May 24 15:06:20.475 [initandlisten] db version v2.5.1-pre
Fri May 24 15:06:20.475 [initandlisten] git version:
704dc4fdf5248077c53271f249260478d6c56cd3
Fri May 24 15:06:20.475 [initandlisten] build info: Darwin bs-osx-106-x86-64-1.local
10.8.0
Darwin Kernel Version 10.8.0: Tue Jun 7 16:33:36 PDT 2011;
root:xnu-1504.15.3~1/RELEASE_I386
i386 BOOST_LIB_VERSION=1_49
Fri May 24 15:06:20.475 [initandlisten] allocator: system
Fri May 24 15:06:20.475 [initandlisten] options: {}
Fri May 24 15:06:20.479 [initandlisten] journal dir=/data/db/journal
Fri May 24 15:06:20.479 [initandlisten] recover : no journal files present, no recovery
needed
Fri May 24 15:06:20.547 [websvr] admin web console waiting for connections on port
28017
```

服务器将显示出所有正在创建的连接，以及可用于监控服务器状态的信息。

为了终止以手动模式启动的服务器，可以输入`^C`；这将会导致服务器正常关闭。

如果未提供任何配置文件，那么 MongoDB 将会使用默认的数据库路径`/data/db`启动，并使用默认端口 27017(`mongod`)和 28017(管理界面)连接到所有网络 IP，如下所示：

```
$ mkdir -p /data/db
$ mongod
mongod --help for help and startup options
...
Sun Jun 13 13:38:00 waiting for connections on port 27017
Sun Jun 13 13:38:00 web admin interface listening on port 28017
^C
Sun Jun 13 13:40:26 got kill or ctrl c signal 2 (Interrupt), will terminate after
current cmd ends
...
Sun Jun 13 13:40:26 dbexit: really exiting now
```

9.11.2 重新配置服务器

MongoDB 提供了 3 种主要的方式用于配置服务器。首先，可以结合 `mongod` 服务器守护进程使用命令行选项。其次，可以加载一个配置文件。最后，可以使用 `setParameter` 密码修改这些设置中的大多数。例如，通过下面的命令可以将 `logLevel` 修改为默认的 0：

```
> db.adminCommand( {setParameter:1, logLevel:0 } )
```

大多数预打包的 MongoDB 安装包都选择第二种方式，使用(通常)`/etc/mongod.conf` 配置文件，它位于 UNIX/Linux 服务器上。

通过修改该文件并重启服务器可以修改服务器的配置。该文件的内容如下所示：

```
# mongod.conf
dbpath=/var/lib/mongod
logpath=/var/log/mongod/mongod.log
logappend=true
auth = false
#enable the rest interface
rest =true
```

移除选项之前的`#`代码并设置为目标值，即可启用该行对应的选项，因为每行以`#`开头的代码将被认为是注释并被忽略掉。

在配置文件中替换下面的任意一个选项值，如同在启动 MongoDB 时的命令行中指定 `--<optionname> <optionvalue>` 一样：

- `dbpath`：表示 MongoDB 存储数据的位置；需要保证数据被存储在一个快速存储卷上，并且它的大小要足以容纳数据库中的数据。
- `logpath`：表示 MongoDB 存储日志的文件；标准路径是 `/var/logs/mongod/mongod.log`；需要使用 `logrotate` 对日志进行轮转，防止出现日志填满服务器硬盘的情况。
- `logappend`：将该选项设置为假，MongoDB 在每次启动时都将清空日志文件。将该选项设置为真，所有的日志将被追加到现有日志文件的尾部。
- `auth`：启用或禁用 MongoDB 服务器的认证模式；关于认证的更多信息，请参阅本章之前的讨论。

- **rest**: 启用或禁用 MongoDB 的 rest 接口。如果希望使用基于 Web 的状态显示链接显示一些额外的信息，那就必须启用该接口，但在生产环境中不建议这么做，因为所有的信息通过 Mongo shell 都可以获得。

9.11.3 获得服务器版本

使用 `db.version()` 函数可以获得服务器的构建和版本信息。决定是否需要升级或向支持论坛报告问题时，该信息非常有用。下面的脚本将展示如何使用该命令：

```
$mongo
> use admin
switched to db admin
> db.version()
version: 2.5.1-pre-
```

9.11.4 获得服务器状态

MongoDB 提供了一个简单的方法用于判断服务器的状态。

注意：

如果启用了认证，那么使用的用户必须获得运行这些命令的权限。

下面的样例显示了返回的信息，保存一些诸如服务器运行时间、最大连接数等信息：

```
$mongo
> db.serverStatus()
{
  "host" : "Pixl.local",
  "version" : "2.5.1-pre-",
  "process" : "mongod",
  "pid" : 3737,
  "uptime" : 44,
  "uptimeMillis" : NumberLong(43035),
  "uptimeEstimate" : 39,
  "localTime" : ISODate("2013-05-25T12:38:34.015Z"),
  "asserts" : {
    "regular" : 0,
    "warning" : 0,
    "msg" : 0,
    "user" : 1,
    "rollovers" : 0
  },
  "connections" : {
    "current" : 1,
    "available" : 2047,
    "totalCreated" : NumberLong(1)
  },
  "cursors" : {
    "totalOpen" : 0,
    "clientCursors_size" : 0,
    "timedOut" : 0
  },
  "globalLock" : {
    "totalTime" : NumberLong(43035000),
    "lockTime" : NumberLong(48184),
```

第III部分 使用 MongoDB 处理大数据

```
    "currentQueue" : {
      "total" : 0,
      "readers" : 0,
      "writers" : 0
    },
  },
  "locks" : {
    "admin" : {
      "timeLockedMicros" : {
        "r" : NumberLong(54),
        "w" : NumberLong(0)
      },
      "timeAcquiringMicros" : {
        "r" : NumberLong(2190),
        "w" : NumberLong(0)
      }
    },
    "local" : {
      "timeLockedMicros" : {
        "r" : NumberLong(45),
        "w" : NumberLong(6)
      },
      "timeAcquiringMicros" : {
        "r" : NumberLong(7),
        "w" : NumberLong(1)
      }
    }
  },
  ...
},
"network" : {
  "bytesIn" : 437,
  "bytesOut" : 6850,
  "numRequests" : 7
},
"opcounters" : {
  "insert" : 1,
  "query" : 6,
  "update" : 0,
  "delete" : 0,
  "getmore" : 0,
  "command" : 7
},
...
"mem" : {
  "bits" : 64,
  "resident" : 37,
  "virtual" : 3109,
  "supported" : true,
  "mapped" : 320,
  "mappedWithJournal" : 640
},
  "ttl" : {
    "deletedDocuments" : NumberLong(0),
    "passes" : NumberLong(0)
  }
},
"ok" : 1
}
```

如你所见，`serverStatus` 输出了相当多的细节信息，上面的信息甚至被截断了！在该函数输

出的信息中，可以找到两个最重要的部分：`opcounters` 和 `asserts`。

`opcounters` 部分显示了服务器上已经执行的每种类型操作的数目。对于特定的服务器，你应该已经知道这些计数器的正常情况。如果这些计数器开始偏离正常的比例，那么可能就是早期的警告：应用中出现了问题。

例如，某个输出中有非常高的插入/读取比例。这对于日志应用来说可能是正常的；不过，对于博客应用，可能表示垃圾邮件程序正在使用“评论”功能，或者搜索引擎正在反复读取一个会导致数据库写入的 URL 模式。此时，要么在评论表单中添加验证码，要么在 `robots.tx` 文件中阻止特定的 URL 模式。

`asserts` 部分展示了服务器和客户端抛出异常或警告的数目。如果异常或警告的数目迅速增加，那么最好查看服务器的日志文件以检查是否系统出现了问题。大量的断言也可能表示数据库中的数据出现了问题，你应该检查 MongoDB 实例的日志文件以确认这些断言是否属于普通的“用户断言”，这种断言代表重复键值冲突这样的问题或者其他更紧迫的问题。

9.11.5 关闭服务器

如果使用安装包安装 MongoDB 服务器，那么可以使用操作系统服务管理脚本关闭服务器。例如，在 Ubuntu、Fedora、CentOS 和 RedHat 中可以通过下面的命令关闭服务器：

```
$sudo service mongod stop
```

还可以从 mongo 控制台关闭服务器：

```
$mongo
> use admin
> db.shutdownServer()
```

也可以使用 Posix 进程管理命令终止服务器，或者使用 `SIG_TERM(-15)` 或 `SIG_INT(-2)` 信号关闭服务器。

当且仅当服务器不响应这两种方法时，才可以使用下面的命令：

```
$sudo killall -15 mongod
```

警告：

禁止使用 `SIG_KILL(-9)` 信号终止服务器，因为这将导致数据库损坏，可能需要修复数据库。

某个活跃服务器可能有大量的写入操作，并且已经重新配置该服务器使它具有较大的同步延迟。此时，该服务器可能无法快速响应终止请求，因为它正将所有内存中的修改写入到磁盘。此时就需要稍微等待一会儿。

9.12 使用 MongoDB 日志文件

默认情况下，MongoDB 将把所有的日志都输出标准输出流中；不过，使用之前描述的 `logpath` 选项可以将日志输出重定向至文件。

通过分析日志文件的内容，可以发现诸如某台机器创建过多连接这样的问题，日志中的错误消息也可能表示应用逻辑或数据出现了问题。

9.13 验证和修复数据

如果服务器意外重启或 MongoDB 服务器由于某种原因崩溃，那么数据可能就处于损坏或不完整的状态。

下面是一些数据已损坏的迹象：

- 数据库服务器拒绝启动，表示数据文件已损坏。
- 在服务器日志文件中发现断言或使用 `db.serverStatus()` 命令时发现断言数目很大。
- 查询结果很奇怪或出乎意料。
- 集合中的记录数目与预期不匹配。

任何一种迹象都可能表示应用出现了问题，或更麻烦的是，数据损坏或处于不一致状态。

幸运的是，MongoDB 含有修复或恢复数据库服务器的工具。不过，你可能仍然会丢失许多数据，因此谨记如下黄金法则：保证对数据进行备份或使用复制主从服务器。

9.13.1 修复服务器

在启动服务器修复进程之前，必须注意使用 `repair` 命令是一个代价很高的操作，它将会花费很长的时间，并要求使用两倍于 MongoDB 数据文件大小的空间，因为所有的数据都将被克隆到新的文件并重建，这本质上是对所有数据文件的重建。这是使用复制集的最好理由之一：如果需要将某个机器离线并修复，此时不需要完全停止复制集，也仍然可以处理客户端的请求。

为了启动修复进程，手动启动服务器进程即可(如本章之前所描述的)。不过，此次需要在命令的尾部添加 `--repair` 选项，如下所示：

```
$ mongod --dbpath /data/db --repair
Wed Sep 18 21:21:21.364 [initandlisten] MongoDB starting : pid=5973 port=27017
dbpath=/data/db
64-bit host=Pixl.local
Wed Sep 18 21:21:21.364 [initandlisten]
Wed Sep 18 21:21:21.364 [initandlisten] ** WARNING: soft rlimits too low. Number of
files is 256,
should be at least 1000
Wed Sep 18 21:21:21.364 [initandlisten] db version 2.5.1-pre-
Wed Sep 18 21:21:21.364 [initandlisten] git version:
704dc4fdf5248077c53271f249260478d6c56cd3
Wed Sep 18 21:21:21.364 [initandlisten] build info: Darwin
bs-osx-106-x86-64-2.10gen.cc 10.8.0
Darwin Kernel Version 10.8.0: Tue Jun 7 16:32:41 PDT 2011;
root:xnu-1504.15.3~1/RELEASE_X86_64
x86_64 BOOST_LIB_VERSION=1_49
Wed Sep 18 21:21:21.364 [initandlisten] allocator: system
Wed Sep 18 21:21:21.364 [initandlisten] options: { dbpath: "/data/db", repair: true }
Wed Sep 18 21:21:21.367 [initandlisten] build index test.system.users { user: 1,
userSource: 1 }
Wed Sep 18 21:21:21.368 [initandlisten] build index done. scanned 1 total records.
0.001 secs
Wed Sep 18 21:21:21.368 [initandlisten] ****
Wed Sep 18 21:21:21.368 [initandlisten] ****
Wed Sep 18 21:21:21.368 [initandlisten] need to upgrade database test with pdf file
version 4.5,
new version: 4.5
```

```

Wed Sep 18 21:21:21.368 [initandlisten] starting upgrade
Wed Sep 18 21:21:21.368 [initandlisten] test repairDatabase test
Wed Sep 18 21:21:21.369 [FileAllocator] allocating new datafile
/data/db/_tmp_repairDatabase_0/test.
ns, filling with zeroes...
Wed Sep 18 21:21:21.369 [FileAllocator] creating directory
/data/db/_tmp_repairDatabase_0/_tmp
Wed Sep 18 21:21:21.389 [FileAllocator] done allocating datafile
/data/db/_tmp_repairDatabase_0/
test.ns, size: 16MB, took 0.02 secs
Wed Sep 18 21:21:21.389 [FileAllocator] allocating new datafile
/data/db/_tmp_repairDatabase_0/
test.0, filling with zeroes...
Wed Sep 18 21:21:21.583 [FileAllocator] done allocating datafile
/data/db/_tmp_repairDatabase_0/
test.0, size: 64MB, took 0.193 secs
Wed Sep 18 21:21:21.583 [FileAllocator] allocating new datafile
/data/db/_tmp_repairDatabase_0/
test.1, filling with zeroes...
Wed Sep 18 21:21:21.586 [initandlisten] build index test.foo { _id: 1 }
Wed Sep 18 21:21:21.661 [initandlisten] fastBuildIndex dupsToDrop:0
Wed Sep 18 21:21:21.661 [initandlisten] build index done. scanned 1 total records.
0.074 secs
Wed Sep 18 21:21:21.661 [initandlisten] build index test.system.users { user: 1,
userSource: 1 }
Wed Sep 18 21:21:21.662 [initandlisten] fastBuildIndex dupsToDrop:0
Wed Sep 18 21:21:21.662 [initandlisten] build index done. scanned 0 total records.
0 secs
Wed Sep 18 21:21:21.662 [initandlisten] build index test.system.users { _id: 1 }
Wed Sep 18 21:21:21.663 [initandlisten] fastBuildIndex dupsToDrop:0
Wed Sep 18 21:21:21.663 [initandlisten] build index done. scanned 1 total records.
0 secs
Wed Sep 18 21:21:22.002 [FileAllocator] done allocating datafile
/data/db/_tmp_repairDatabase_0/
test.1, size: 128MB, took 0.418 secs
Wed Sep 18 21:21:22.018 [initandlisten] finished checking dbs
Wed Sep 18 21:21:22.018 dbexit:
Wed Sep 18 21:21:22.018 [initandlisten] shutdown: going to close listening sockets...
Wed Sep 18 21:21:22.018 [initandlisten] shutdown: going to flush diaglog...
Wed Sep 18 21:21:22.018 [initandlisten] shutdown: going to close sockets...
Wed Sep 18 21:21:22.018 [initandlisten] shutdown: waiting for fs preallocator...
Wed Sep 18 21:21:22.018 [initandlisten] shutdown: closing all files...
Wed Sep 18 21:21:22.018 [initandlisten] closeAllFiles() finished
Wed Sep 18 21:21:22.018 [initandlisten] shutdown: removing fs lock...
Wed Sep 18 21:21:22.018 dbexit: really exiting now

```

在本例中，修复命令检测到 **admin** 数据库可能是由旧版本的 MongoDB 创建的，因此要求对存储格式进行升级以匹配目前运行的服务器版本。

注意：

使用 `--repair` 选项启动的服务器最终会退出，这是正常的；为了正式启动服务器，只需要移除 `--repair` 选项并再次启动即可。

一旦修复进程结束，就可以正常启动服务器，然后从备份中恢复任何丢失的数据。

如果尝试修复一个大型数据库，那么驱动器上的磁盘空间可能会不足，因为 MongoDB 需

要在同一驱动器中创建数据库的副本作为数据源(参见之前样例中的目录.../\$tmp_repariDatabase_0/...).

为了处理这个潜在的问题, MongoDB 修复工具提供了一个额外的命令行参数--repairPath。可以使用该参数指定一个具有足够空间的驱动器用于保存修复过程中创建的临时文件, 如下所示:

```
$ mongod -f /etc/mongodb.conf --repair --repairpath /mnt/bigdrive/tempdir
```

9.13.2 验证单个集合

有时, 你可能会怀疑运行服务器上的数据是否存在问题。此时, 可以使用一些 MongoDB 提供的工具判断目标服务器是否损坏。

通过 validate 选项可以验证数据库中集合的内容。下一个样例将显示如何在一个含有 1 百万条数据记录的集合中运行 validate 选项:

```
$mongo
> use blog
switched to db blog
> db.posts.ensureIndex({Author:1})
> db.posts.validate()
{
  "ns" : "blog.posts",
  "firstExtent" : "0:f000 ns:blog.posts",
  "lastExtent" : "0:2b9000 ns:blog.posts",
  "extentCount" : 6,
  "datasize" : 6717520,
  "nrecords" : 29997,
  "lastExtentSize" : 8388608,
  "padding" : 1,
  "firstExtentDetails" : {
    "loc" : "0:f000",
    "xnext" : "0:11000",
    "xprev" : "null",
    "nsdiag" : "blog.posts",
    "size" : 8192,
    "firstRecord" : "0:f0b0",
    "lastRecord" : "0:10e50"
  },
  "lastExtentDetails" : {
    "loc" : "0:2b9000",
    "xnext" : "null",
    "xprev" : "0:b9000",
    "nsdiag" : "blog.posts",
    "size" : 8388608,
    "firstRecord" : "0:2b90b0",
    "lastRecord" : "0:6ec830"
  },
  "deletedCount" : 4,
  "deletedSize" : 3983552,
```

```

    "nIndexes" : 2,
    "keysPerIndex" : {
      "blog.posts.$_id_" : 29997,
      "blog.posts.$Author_1" : 29997
    },
    "valid" : true,
    "errors" : [ ],
    "warning" : "Some checks omitted for speed. use {full:true} option to do more
thorough scan.",
    "ok" : 1
  }

```

之前的样例大概需要 30 秒钟才能完成。默认情况下，`validate` 选项将同时检查数据文件和索引，并在操作完成时提供集合的一些统计信息。该选项将显示数据文件或索引中是否存在问题，但不会检查所有文档的正确性。如果希望检查所有的文档，那么可以使用 `{full: true}` 选项运行 `validate()` 函数(如同输出中建议的一样)，在函数调用中添加 `true` 参数即可，如下所示：

```
db.posts.validate(true)
```

如果现在有一个非常大的数据库并且只希望验证索引，也可以使用 `validate` 选项。当前版本(2.6.0)并未提供相关的 `shell` 帮助命令。但这并不是问题，可以通过使用 `runCommand` 选项完成索引验证：

```

$mongo
> use blog
> db.runCommand({validate:"posts", scandata:false})

```

在该例中，服务器不会扫描数据文件；相反，它只会报告一些集合相关的信息。

9.13.3 修复集合验证错误

如果在验证集合的过程中出现错误(显示在验证文档的 `errors` 部分)，有几种方式可以修复数据(再次提醒你，一定要记住对数据进行备份)。在恢复备份之前，应该先查看 MongoDB 实例的日志，检查是否存在任何关于该错误的相关信息；如果有，那么该信息将提示你下一个需要完成的步骤。

1. 修复集合的索引

如果验证结果显示索引是损坏的，那么可以使用 `reIndex()` 函数重建受影响集合的索引。在下面的例子中，我们将使用 `reIndex()` 函数修复博客的 `posts` 集合中的 `author` 索引：

```

$mongo
> use blog
> db.posts.reIndex()
{
  "nIndexesWas" : 2,
  "msg" : "indexes dropped for collection",
  "nIndexes" : 2,
  "indexes" : [
    {

```

```

    "key" : {
      "_id" : 1
    },
    "ns" : "blog.posts",
    "name" : "_id_"
  },
  {
    "key" : {
      "Author" : 1
    },
    "ns" : "blog.posts",
    "name" : "Author_1"
  }
],
"ok" : 1
}

```

MongoDB 将会删除集合中目前的所有索引，然后重建它们。不过，如果使用了数据库的 `repair` 选项，也将在数据库的所有集合中运行 `reIndex()` 函数。

2. 修复集合的数据文件

修复所有数据文件的最好也是最危险的方式是：使用服务器的 `--repair` 选项或 `shell` 中的 `db.repairDatabase()` 命令。一种方式将修复单个数据库中的所有集合文件，然后重建所有已定义的索引。不过，`repairDatabase()` 不适合在在线服务器上运行，因为它在重建数据文件时会阻塞对数据的所有请求。这将导致数据库修复过程中的所有读取和写入操作都被阻塞。下面的脚本展示了使用 `repairDatabase()` 函数的语法：

```

$mongo
> use blog
> db.repairDatabase()
{ "ok" : 1 }

```

警告：

MongoDB 的修复功能是一个强力选项。它将尝试修复并重建数据结构和索引。它通过尝试从磁盘读取并重建整个数据结构的方式来完成。如果可能，你应该尝试从备份文件恢复；`repairDatabase()` 只应被用作最后的解决方案。

3. 压缩集合的数据文件

由于 MongoDB 内部分配数据文件的方式，你可能会碰到“瑞士奶酪”这种情况，它意味着一小块空的存储空间被保留在磁盘数据结构中。这可能是一个问题，因为数据文件中可能有大量的空间未被使用。使用修复功能重建整个数据结构可能能够解决这个问题，但也可能会带来其他不可预料的结果。而命令 `compact` 将会整理并重组数据文件中指定集合的数据结构，但不会恢复磁盘空间。

```

$mongo
> use blog
> db.runCommand({compact:"posts"})
{ "ok" : 1 }

```

9.14 升级 MongoDB

偶尔,新版 MongoDB 会要求升级数据库文件的格式。MongoDB Inc.的团队已经注意到了在正在运行的生产环境中执行升级所产生的影响(包括服务器宕机时间);不过,有时为了支持某些所需的新特性必须进行升级。

警告:

在尝试执行任何升级之前,进行数据的完整备份是必不可少的。另外,还应该检查对应版本的发行声明,地址为 <http://docs.mongodb.org/manual/release-notes/>。

MongoDB 的开发者尝试解决所有在升级过程中出现的问题;不过,你也必须采取措施来保护自己的数据。升级过程中通常会将系统的所有数据重写为新的格式,这意味着哪怕是进程中一个微小的问题也可能会引起严重的后果。

下面的列表显示了升级数据库服务器的正确步骤(也是必需的):

- (1) 备份数据并保证备份可用。如果可能,将备份数据恢复到另一个服务器,确认备份是正确的。
- (2) 停止应用,或者将它转移到另一台服务器。
- (3) 停止 MongoDB 服务器。
- (4) 升级 MongoDB 服务器的代码至目标版本。
- (5) 使用 shell 对数据集进行初始的完整性检测。
- (6) 只要有任何地方看起来可能有问题,就使用验证工具检查数据。
- (7) 完成所有的检查之后,重新启动应用。
- (8) 在重新开启服务或者将流量转移回当前服务器时,对应用认真进行测试。

9.15 MongoDB 的滚动升级

使用复制集的最大特点之一就是:可用于执行滚动升级。该方法被设计用于减小潜在的宕机时间和一些 MongoDB 大改动(例如升级)所造成的影响。除了下面列出的流程,你还应该创建备份并在非生产环境中进行测试。一旦完成详细的调查并保证系统是可恢复的,就可以按照下面的流程进行滚动升级:

- (1) 一次停止一台辅助服务器并进行升级。
- (2) 在主服务器上执行 `rs.stepDown()` 命令。已经升级成功的某台辅助服务器将变成主服务器。
- (3) 升级主服务器。

9.16 监控 MongoDB

MongoDB 发行包中包含了一个简单的状态监控工具,被称为 `mongostat`。该工具主要被设计用于提供系统状态的简单概览(见图 9-5)。

```

smongostat
connected to: 127.0.0.1
insert query update delete getmore command flushes mapped vsize res faults locked db idx miss % qriqw arlaw netIn netOut conn time
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:34
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:35
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:36
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:37
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:38
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:39
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:40
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:41
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:42
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:43
insert query update delete getmore command flushes mapped vsize res faults locked db idx miss % qriqw arlaw netIn netOut conn time
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:44
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:45
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:46
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:47
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:48
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:49
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:50
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:51
*0 *0 *0 *0 0 110 0 320m 3.04g 37m 0 local:0.0% 0 0 010 010 62b 3k 1 17:34:52
    
```

图 9-5 使用 mongostat 工具监控 MongoDB 的状态

该工具生成的统计信息并不全面，但它们提供了 MongoDB 服务器当前状态的概览。例如，将显示出数据库操作的执行频率、索引的命中率以及应用等待数据库释放锁所花费的时间。

最有力的主要数据列是前 6 列，它们将显示出 mongod 服务器处理特定操作(例如插入或查询)的速率。在分析问题的时候，值得关注的其他列还有：

- **pagefaults:** 代表 MongoDB 实例为了完成查询需要从磁盘读取数据。这通常标志着当前服务器未达到最优性能，日常操作所需的所有数据并未完全保存在 MongoDB 可用的 RAM 中。应该检查是否有某些查询导致对所有的文档进行扫描并且未使用索引，或者需要将 MongoDB 实例移到一台有更多可用 RAM 的服务器。
- **queues:** 代表排队等待执行的操作的数目。因为 MongoDB 支持一个写者(插入、更新和删除)和多个读者(查找)，这可能导致出现读取查询被表现不佳的写操作阻塞的情况。更糟的是，可能出现许多读/写操作同时被一个性能不佳的写操作阻塞的情况。检查是否存在某个查询阻塞了其他查询的执行。
- **% locked:** 显示指定集合中写锁所占用的时间百分比。一个非常高的数字表明你正在执行一个或多个写操作，并且几乎占用了所有的时间窗口。一个高%locked 值可能会影响所有查询的性能。检查是否存在性能不佳的写操作，或者系统是否正处于缺页错误状态，这可能表示需要更多的 RAM。这也可能是与模式相关的问题，例如文档中有一个非常大的数组，记住在 MongoDB 2.2 版本之前锁是在实例级别创建的，所以升级 MongoDB 可能会降低该值，因为新版 MongoDB 将会增加并发度。

创建自己的状态监控工具

mongostat 提供的许多信息与 db.serverStatus()调用获得的信息一致。因此，创建一个服务，每隔数秒调用一次该 API 以抓取服务器状态，然后存储到 MongoDB 集合中并不是一项很大的任务。

在应用中包含一些指标、一些精心设计的查询和一个图形包，这样就可以创建一个简单的实时监控工具，并可以产生历史日志。

MongoDB 也有许多可用的第三方工具，既有公共开源的系统，也有商业监控系统，包括

Nagios、Ganglia 和 Cacti 这样的工具。如之前提到的，MongoDB 在网站的手册中包含了一个页面用于分享 MongoDB 可用的监控工具的最新信息(关于该话题的更多信息，请参见 <http://docs.mongodb.org/manual/administration/monitoring/>)。

9.17 使用 MongoDB 管理服务

到目前为止所讨论的统计信息都可以通过 MongoDB 管理服务(MongoDB Management Service, MMS)获得，MMS 是由 MongoDB Inc.提供的监控服务。它提供了一个可安装在本地机器中的代理。一旦安装成功，就可以通过 MMS Web 页面添加正确的服务器，从而指示代理对它们进行监控。从单个 MongoDB 实例到复制集，以及完整的分片集群，包括配置服务器和 MongoS，可以监控 MongoDB 的所有信息。MMS 也提供了一种机制，用于查看这些组中的所有单个成员，获得它们的聚集统计信息。接下来还可以配置一些发送给自己的警告，这些警告可以基于特定的性能需求或 MongoDB 实例中发生的事件。可以在 mms.mongodb.com 上注册 MMS 账户，我们也推荐你这么。没有什么能比直接看到 MongoDB 节点正在执行的操作更强大，如图 9-6 所示。

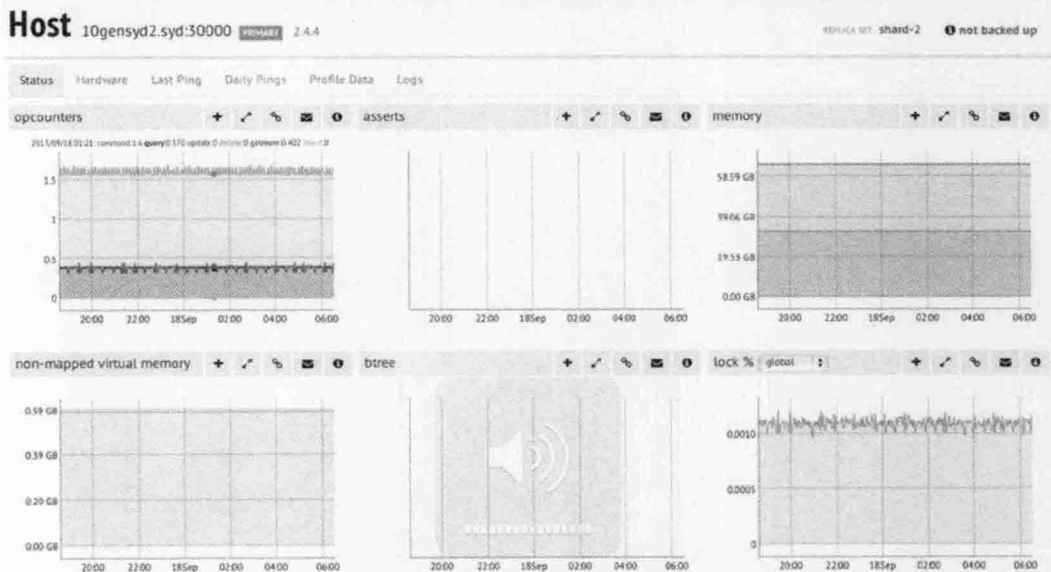


图 9-6 使用 MMS 查看统计信息

9.18 小结

为了保证 MongoDB 数据库运行平稳，通常只需要花费一点功夫即可。本章讲解了如何使用 MongoDB 发行包所提供的工具管理和维护系统，同时也讲解了所有你可能会遇到的问题。

通过学习本章内容，你应该能够备份、恢复、升级和监控 MongoDB 实例，熟悉 `mongodump` 和 `mongorestore` 此类工具，知道如何导入和导出特定集合中的数据，了解许多可用于 MongoDB 实例性能改善和统计信息收集的管理命令。

最后，必须强调(你可能也已经在学习本章的过程中感受到)本章最重要的一个观点：作为数据库管理员，首先必须确保数据有可靠的备份以及可执行的恢复方案。

第 10 章

优 化

某个 Twitter 用户曾发出这样一个带有调侃的声明：“如果 MongoDB 查询运行超过 0 毫秒，那它一定是出什么问题了。”自从 MongoDB 在 2009 年出现以来，这种对 MongoDB 的调侃并不少见。

事实上 MongoDB 是非常快的。但与任何数据存储系统一样，如果你使用了错误的数据结构，或者并未在集合中创建正确的索引，MongoDB 的速度可能会急剧地下降。

MongoDB 也包含了一些高级特性，为了使它们以最佳性能运行，也需要对服务器进行一些调整。

数据模式的设计也会对性能产生巨大的影响；本章首先将讲解一些技术，它们将把数据塑造成某种格式，从而能够最大地利用 MongoDB 的优势，并最小化由于它的缺点所造成的影响。

在开始学习如何改善服务器上运行的查询的性能或者优化数据结构之前，我们将首先学习 MongoDB 如何与它所运行的硬件进行交互，以及影响性能的因素。然后我们将学习索引以及如何使用它们改善查询的性能，同时我们还将学习如何分析 MongoDB 实例以确定其中是否存在性能不佳的查询。

10.1 优化服务器硬件

通常对数据库进行的最快速也是最便宜的优化就是为之选择正确大小的硬件。如果数据库服务器的内存太少或者驱动器速度太低，就可能会对数据库性能产生巨大影响。其中的一些限制对于开发环境来说可能是可以接受的，此时服务器将运行在开发者的本地工作站中，但对于生产环境来说这是不可接受的，为了达到最佳性能，必须小心地对硬件配置进行计算。

10.1.1 了解 MongoDB 使用内存的方式

MongoDB 使用内存映射文件 I/O 来访问数据存储。这种文件 I/O 有一些特点需要注意，因为它们会同时受操作系统(OS)类型和内存大小的限制。

内存映射文件的第一个需要注意的特点是，在现代 64 位操作系统中，Linux 中的最大文件可以达到 128TB 左右(Linux 虚拟内存地址限制)，Windows 对内存映射文件的限制使得最大文件最多可以达到 8TB(启用日志之后则只有 4TB 可用)。在 32 位操作系统中，最大文件被限制为 2GB，所以不推荐使用 32 位操作系统运行，除非只是用作小的开发环境。

第二个需要注意的特点是，内存映射文件将使用操作系统的虚拟内存，系统将按需把数据文件的某个部分映射到 RAM 中。这会产生一种稍微有点让人吃惊的印象：MongoDB 将会用尽系统的所有 RAM。其实并不是这样，因为 MongoDB 将与其他应用一起共享虚拟地址空间。并

且 OS 也会按需将内存释放给其他进程。使用空闲内存的总数作为过度消耗内存的标志并不是一个好的实践，因为好的 OS 将会保证有很少的或者压根儿没有“空闲”内存。所有昂贵的内存都将被用于缓存或缓冲硬盘 I/O。空闲的内存是对内存的浪费。

通过提供合适大小的内存，MongoDB 可以在内存中保持更多它所需的数据，这将减少对昂贵的磁盘 I/O 的访问。

通常，为 MongoDB 分配的内存越多，它的运行速度就越快。不过，如果数据库大小只有 2GB，那么添加超过 3GB 的内存并不会加快它的速度，因为整个数据库都已经在内存中了。

10.1.2 了解工作集大小

现在我们需要讨论涉及 MongoDB 实例性能调优的一个更复杂的概念：工作集大小。这个大小代表着 MongoDB 实例中存储的数据量（“在正常使用的过程中”将会访问到的数据）。前面这个独立的短语表示工作集大小的计算是一种主观的方式，并且难于得到准确值。

尽管难于量化，了解工作集大小的影响可以帮助你更好地优化 MongoDB 实例。主要的规则是：对于大多数 MongoDB 实例，常规操作通常只会访问到其中的一部分数据。了解正常工作中使用的是哪部分数据，可以帮助你正确地计算出硬件的大小，从而提高 MongoDB 的性能。

10.1.3 选择正确的数据库服务器硬件

现在有一种普遍的压力，要将服务移到低功耗的系统中。不过为了实现更低的能耗，许多低功耗服务器都使用笔记本电脑组件。遗憾的是，低质量的服务器硬件可能会使用较为廉价的磁盘驱动器。这样的驱动器并不适合高负载的服务器应用，因为它们的磁盘转速低，这将降低从驱动器读取数据和写入数据的速度。另外，要确保使用的是著名供应商的产品，使用它们的产品可以构建出一个已经为服务器操作做过优化的系统。另外值得一提的是，更快以及更现代的驱动器(例如 SSD)可以提供显著的性能。如果可以使用，那么 MongoDB Inc. 推荐使用 RAID10，既为了性能也为了冗余。如果使用的是云，那么使用一些诸如 Amazon 提供的 Provisioned IOPS 之类的技术，这是提升系统性能和磁盘可靠性的好方式。

如果计划使用复制或者任何一种频繁备份系统(它们将需要从网络连接中读取数据)，你应该考虑添加一个额外的网卡并组成单独的网络，使这些服务器可以彼此交流。这将减少在连接应用和服务器的网络上传输和接受的数据量，而这也将会影响应用的性能。

购买硬件时可能最需要注意的硬件就是 RAM。因为 MongoDB 使用内存映射文件，所以拥有足够的空间用于保存必要的数据是确保高性能的一种好方式。此时可以与之前讨论的工作集概念链接起来。知道需要分配多少数据是购买什么硬件的关键。最后要记住，不需要在服务器上安装 512GB 的 RAM；可以使用分片将数据分散到多台服务器(第 12 章将进行讲解)。

10.2 评估查询性能

MongoDB 有两个用于优化查询性能的工具：`explain()`和 MongoDB 分析器。MongoDB 分析器是查找性能不佳的查询以及选择查询用于进一步检查的好工具，而 `explain()`是研究单个查询的好工具，通过它可以判断查询的执行性能。

对于熟悉 MySQL 的读者来说，可能也熟悉慢速查询日志，该日志将帮助查找耗费时间过长的查询；MongoDB 使用分析器提供这种功能。

10.2.1 MongoDB 分析器

MongoDB 分析器将记录统计信息，以及所有符合触发条件的查询的执行计划细节。通过在启动 MongoDB 进程时添加 `--profile` 和 `--slowms` 选项(很快会讲解这些选项的含义)可以单独在每个数据库上启用或同时在所有数据库上启用。这些选项也可以被添加到 `mongodb.conf` 文件中，如果你选择这种方式启动 MongoDB 进程的话。

一旦启用分析器，MongoDB 将为应用提交的每个查询在一个特殊的固定大小集合 `system.profile` 中，插入一个含有性能和执行细节的文档。通过使用标准的集合查询命令，可以访问该集合，获得被记录的每个查询的细节信息。

`system.profile` 集合的大小被限制为 1024KB。所以磁盘空间不会被分析器的日志所填满。该限制足以捕捉数千条分析信息，哪怕是最复杂的查询。

警告：

启动分析器后，它将会影响服务器的性能，所以一直在生产环境中运行它并不是一个好主意，除非正在分析一些已经发现的问题。不要尝试为了提供最近执行的查询而保持分析器一直运行。

1. 启用和禁用 MongoDB 分析器

启用 MongoDB 分析器非常简单：

```
$mongo
> use blog
> db.setProfilingLevel(1)
```

禁用分析器同样简单：

```
$mongo
> use blog
> db.setProfilingLevel(0)
```

还可以只针对超过了特定执行时间的查询启用分析器。下面的样例将只记录执行超过半秒钟的查询：

```
$mongo
> use blog
> db.setProfilingLevel(1,500)
```

如该例所示，对于分析级别 1，可以提供以毫秒(ms)为单位的最大查询执行时间。如果查询的执行时间超过该设置，它将被分析，然后被记录下来；否则，它将被忽略。这种方式提供了与 MySQL 慢速查询日志相同的功能。

最后，通过将分析级别设置为 2，可以为所有的查询启用分析器：

```
$mongo
> use blog
> db.setProfilingLevel(2)
```

2. 查找慢速查询

system.profile 集合中的典型记录如下所示:

```
> db.system.profile.find()
{
  "op" : "query",
  "ns" : "blog.system.profile",
  "query" : {
  },
  "ntoreturn" : 0,
  "ntoskip" : 0,
  "nscanned" : 1,
  "keyUpdates" : 0,
  "numYield" : 0,
  "lockStats" : {
    "timeLockedMicros" : {
      "r" : NumberLong(60),
      "w" : NumberLong(0)
    },
    "timeAcquiringMicros" : {
      "r" : NumberLong(4),
      "w" : NumberLong(3)
    }
  },
  "nreturned" : 1,
  "responseLength" : 370,
  "millis" : 12,
  "ts" : ISODate("2013-05-18T05:40:27.106Z"),
  "client" : "127.0.0.1",
  "user" : ""
}
```

其中的每条记录都包含了一些字段, 下面的列表显示了它们的定义和含义:

- op: 显示操作的类型; 可以是查询、插入、更新、命令或删除操作。
- query: 正在运行的查询。
- ns: 查询所在的完整名称空间。
- ntoreturn: 返回文档的数目。
- nscanned: 为返回该文档而扫描的索引条目数目。
- ntoskip: 被忽略的文档数目。
- keyUpdates: 该查询更新的索引键数目。
- numYields: 该查询为其他查询让出锁的次数。
- lockStats: 数据库花费在获取读写锁上的毫秒数。
- nreturned: 返回文档的数目。
- responseLength: 响应的字节长度。
- millis: 执行查询所花费的毫秒数。
- ts: 以 UTC 格式显示出查询执行时的时间戳。
- client: 运行该查询的客户端的连接信息。
- user: 运行该操作的用户。

因为 `system.profile` 只是一个普通的集合，所以可以使用 MongoDB 的查询工具快速找到有问题的查询。

接下来的样例将查找出所有执行时间长于 10ms 的查询。在这种情况下，你可以先在 `system.profile` 集合中查询 `millis>10` 的记录，然后按照执行时间对结果进行降序排序：

```
> db.system.profile.find({millis:{$gt:10}}).sort({millis:-1})
{ "op" : "query", "ns" : "blog.system.profile", "query" : { }, "ntoreturn" : 0,
"ntoskip" : 0,"nscanned" : 1, "keyUpdates" : 0, "numYield" : 0, "lockStats" :
{ "timeLockedMicros" : { "r" : NumberLong(60), "w" : NumberLong(0) }, "timeAcquiringMicros" :
{ "r" : NumberLong(4), "w" :NumberLong(3) } }, "nreturned" : 1, "responseLength" : 370,
"millis" : 12, "ts" : ISODate("2013-05-18T05:40:27.106Z"), "client" : "127.0.0.1",
"allUsers" : [ ], "user" : " " }
```

如果还知道问题可能发生在某个特定的时间范围内，那么可以使用 `ts` 字段添加查询条件，用于限制时间范围。

3. 增大分析器集合的大小

如果出于某种原因，发现分析器集合太小，那么可以增加它的大小。

首先禁用目标数据库(希望增加分析器集合大小的数据库)上的分析器，保证执行下列操作时不会有写入操作发生：

```
$mongo
> use blog
> db.setProfilingLevel(0)
```

接下来需要删除现有的 `system.profile` 集合：

```
> db.system.profile.drop()
```

删除该集合之后，可以使用 `createCollection` 命令创建新的分析器集合，并使用目标字节大小作为参数。下面的样例将创建一个大小为 50MB 的固定大小集合。它通过乘以 1024 的方式将 50 字节转换成 KB，然后再转换成 MB：

```
> db.createCollection( "system.profile", { capped: true, size: 50 * 1024 * 1024 } )
{ "ok" : 1 }
```

成功创建更大的固定大小集合之后，可以重新启用分析器：

```
> db.setProfilingLevel(2)
```

10.2.2 使用 `explain()` 分析特定的查询

如果怀疑某个查询的性能不佳，可以使用 `explain()` 检查 MongoDB 是如何执行该查询的。

在查询中添加 `explain()` 之后，MongoDB 将在执行时返回一个文档来描述它是如何处理该查询的，而不是返回一个结果的游标。下面的查询将运行在 `blog` 数据库的 `posts` 集合上，结果显示：为了返回所有的帖子，该查询扫描 13325 条记录并创建一个游标：

```
$mongo
> use blog
> db.posts.find().explain()
{
  "cursor" : "BasicCursor",
```

```

    "isMultiKey" : false,
    "n" : 13235,
    "nscannedObjects" : 13235,
    "nscanned" : 13235,
    "nscannedObjectsAllPlans" : 13235,
    "nscannedAllPlans" : 13235,
    "scanAndOrder" : false,
    "indexOnly" : false,
    "nYields" : 0,
    "nChunkSkips" : 0,
    "millis" : 0,
    "indexBounds" : {
    },
    "server" : "Pixl.local:27017"
}

```

表 10-1 列出了由 explain()返回的字段。

表 10-1 由 explain()返回的元素

元 素	描 述
cursor	表示用于遍历结果的游标类型。通常是下列 3 种类型中的一种：BasicCursor (按自然排序的只读游标)、BtreeCursor(索引游标)、GeoSearchCursor(地理空间索引游标)
indexBounds	表示在索引查询中使用的最小/最大值
nScanned	表示为了搜索查询结果而扫描的索引条目
nScannedObjects	代表被扫描的真正对象，而不是它们的索引条目
n	代表游标中数据的数目(即返回的结果数目)
millis	代表执行查询所花费的时间
nscannedAllPlans	代表所有尝试执行的计划所扫描的 nscanned 值
nscannedObjectsAllPlans	代表所有尝试执行的计划所扫描的 nScannedObjects 值
scanAndOrder	真或假，表示是否需要扫描文档进行排序，而不是使用索引
indexOnly	为了对查询的结果进行排序不需要搜索任何文档；所有查询和返回的字段都使用同一索引
nYields	该查询为了让写操作执行而让出读锁的次数
nChunkSkips	由于活跃的块迁移被忽略的文档数目
server	执行该查询的服务器

10.2.3 使用分析器和 explain()优化查询

现在我们将讲解一个真实世界的优化场景，并学习如何使用 MongoDB 的分析器和 explain() 工具解决真正应用中的问题。

本章讨论的样例基于一个小的博客应用。该数据库提供了一个功能，可以将博客文章与特定标签关联起来；此时使用的是偶数标签。假设你已经注意到该函数运行速度非常慢，所以希望判断它是否存在问题。

首先编写一个小程序，使用数据填充之前提到的数据库，用作查询的目标并演示优化的过程。

```

<?php

//获得数据库连接

$mongo = new MongoClient();
$db=$mongo->blog;

//首先获得第一个 AuthorsID
//用它作为假的作者

$author = $db->authors->findOne();

if(!$author){
    die("There are no authors in the database");
}

for( $i = 1; $i < 10000; $i++){
    $blogpost=array();
    $blogpost['author'] = $author['_id'];
    $blogpost['Title'] = "Completely fake blogpost number {$i}";
    $blogpost['Message'] = "Some fake text to create a database of blog posts";
    $blogpost['Tags'] = array();
    if($i%2){
        //奇数博客
        $blogpost['Tags'] = array("blog", "post", "odd", "tag{$i}");
    } else {
        //偶数博客
        $blogpost['Tags'] = array("blog", "post", "even", "tag{$i}");
    }
    $db->posts->insert($blogpost);
}
?>

```

该程序将找到 `blog` 数据库的 `authors` 集合中的第一个作者，然后假设该作者是一个高产作家。以该作者的名字创建 10 000 篇假的博客文章，所有这些操作都将在一瞬间完成。这些文章读起来并不有趣；不过，它们将分别被赋予奇数和偶数标签。这些标签将用于演示如何优化一个简单的查询。

下一步将程序保存为 `fastblogger.php`，然后使用 PHP 的命令行工具运行它：

```
$phpfastblogger.php
```

接下来在数据库中启用分析器，用于判断是否需要改进样例查询的性能：

```

$ mongo
> use blog
switched to db blog
> show collections
authors
posts
...
system.profile
tagcloud
...
users
> db.setProfilingLevel(2)
{ "was" : 0, "slows" : 100, "ok" : 1 }

```


现在等待该命令执行并生效，然后打开必需的集合，再执行其他任务。接下来，模拟网站对所有含有偶数标签的博客文章的访问。执行网站用于实现该功能的查询即可：

```
$Mongo
use blog
$db.posts.find({Tags:"even"})
...
```

如果在分析器集合中查询执行时间超过 5ms，你将会看到类似于下面的结果：

```
> db.system.profile.find({millis:{$gt:5}}).sort({millis:-1})
{ "op" : "query", "ns" : "blog.posts", "query" : { "tags" : "even" }, "ntoreturn" :
0, "ntoskip" : 0, "nscanned" : 19998, "keyUpdates" : 0, "numYield" : 0, "lockStats" :
{ "timeLockedMicros" : { "r": NumberLong(12869), "w" : NumberLong(0) },
"timeAcquiringMicros" : { "r" : NumberLong(5), "w" : NumberLong(3) } }, "nreturned" : 0,
"responseLength" : 20, "millis" : 12, "ts" : ISODate("2013-05-18T09:04:32.974Z"), "client" :
"127.0.0.1", "allUsers" : [ ], "user" : "" }...
```

这里返回的结果展示了某些执行时间超过 5ms 的查询。

接下来，重新构造第一个查询(性能最差)，这样就可以看到当前数据库返回的数据。之前的输出表示有性能不佳的查询在查询 `blog.posts` 集合，并且该查询的条件为 `{Tags: "even"}`。最后，你将看到该查询的执行时间超过 15ms。

重新构建的查询如下所示：

```
>db.posts.find({Tags:"even"})
{ "_id" : ObjectId("4c727cbd91a01b2a14010000"), "author" : ObjectId("4c637ec8b8642
fea02000000"), "Title" : "Completly fake blogpost number 2", "Message" : "Some fake text
to create a database of blog posts", "Tags" : [ "blog", "post", "even", "tag2" ] }
{ "_id" : ObjectId("4c727cbd91a01b2a14030000"), "author" : ObjectId ("4c637ec8b8642
fea02000000"), "Title" : "Completly fake blogpost number 4", "Message" : "Some fake text
to create a database of blog posts", "Tags" : [ "blog", "post", "even", "tag4" ] }
{ "_id" : ObjectId("4c727cbd 91a01b2a14050000"), "author" : ObjectId ("4c637ec8b86
42fea02000000"), "Title" : "Completly fake blogpost number 6", "Message" : "Some fake text
to create a database of blog posts", "Tags" : [ "blog", "post", "even", "tag6" ] }
...
```

输出并无特别之处；创建该查询只是为了演示如何查找和改进慢速查询。

我们的目标是分析出如何使查询运行速度更快，所以使用 `explain()` 函数分析 MongoDB 是如何执行该查询的：

```
> db.posts.find({Tags:"even"}).explain()
{
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 14997,
  "nscannedObjects" : 29997,
  "nscanned" : 29997,
  "nscannedObjectsAllPlans" : 29997,
  "nscannedAllPlans" : 29997,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 27,
  "indexBounds" : {
```

```

    },
    "server" : "Pixl.local:27017"
  }
}

```

从这里的输出可以看到该查询并未使用任何索引。函数 `explain()` 显示该查询使用的是 `BasicCursor`，这意味着该查询只是在执行对集合记录的简单扫描。尤其是为了查找所有的标签，将逐个扫描数据库中的所有记录(9999 个)；该过程将花费 27ms。这个时间听起来并不长，但如果在网站的热门页面上使用该查询，将引起磁盘 I/O 的额外负载，同时也会让 Web 服务器差生巨大压力。最终，当页面正在创建时，该查询将导致 Web 浏览器的连接时间变得更长。

注意：

在详细的查询解释中，如果显示有大量的扫描记录，并扫描的数目远大于返回的结果，那么查询可能需要添加索引。

下一步，判断是否需要在 `Tags` 字段上添加索引以改善查询性能：

```
> db.posts.ensureIndex({Tags:1})
```

再次运行 `explain()` 函数，查看添加索引的效果：

```

> db.posts.find({Tags:"even"}).explain()
{
  "cursor" : "BtreeCursor Tags_1",
  "isMultiKey" : true,
  "n" : 14997,
  "nscannedObjects" : 14997,
  "nscanned" : 14997,
  "nscannedObjectsAllPlans" : 14997,
  "nscannedAllPlans" : 14997,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 4,
  "indexBounds" : {
    "Tags" : [
      [
        "even",
        "even"
      ]
    ]
  },
  "server" : "Pixl.local:27017"
}

```

查询的性能得到了极大改善。可以看到，现在该查询使用的是 `BtreeCursor`(基于 `Tags_1` 索引)。扫描记录的数目从 29994 减少到 14997，该数目与返回的结果数目相同，执行时间也减少到了 4ms。

注意：

最常用的，也是 MongoDB 所使用的唯一索引类型是 `btree`(二叉树)。`BtreeCursor` 是 MongoDB 的一种数据游标，使用二叉树索引遍历文档。`btree` 索引在数据库系统中非常常见，因为它们提供了更快的插入和删除速度，而且在遍历数据或对数据排序时，也提供了合理的性能。

10.3 管理索引

之前已经演示了使用一个认真选择的索引所带来的影响。

如第 3 章所讲的，MongoDB 的索引被用于查询(`find`、`findOne`)和排序。如果倾向于在集合中大量使用排序，那么应该根据排序的需求添加索引。如果在一个没有索引的集合中对目标字段使用 `sort()`，并且数据量超过内部排序缓冲的最大大小，那你将会看到错误消息。在接下来的内容中，我们将再次对基础知识进行介绍，但同时也会增加一些与管理及操作系统中索引相关的内容。我们还将讲解这些索引与某些样例的关系。

当在集合中添加索引时，MongoDB 必须维护它们，并在每次执行写操作(例如更新、插入或删除)时对它们进行更新。如果在集合中有过多的索引，它们有可能会对写操作的性能造成负面的影响。

索引最好用在主要访问为读访问的集合中。对于写入频繁的集合(例如日志系统)，引入索引可能会减少每秒写入的文档数量峰值(这些文档可能会大量涌入集合中)。

警告:

目前，每个集合中最多可以拥有 64 个索引。

10.3.1 显示索引

MongoDB 提供了一个简单的辅助函数 `getIndexes()`，可以显示指定集合中的索引。执行时，它将打印出包含指定集合中所有索引细节的 JSON 数组，包括它们引用的字段或元素，以及在该索引上设置的任何选项。

```
$mongo
> use blog
> db.posts.getIndexes()
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "ns" : "blog.posts",
    "name" : "_id_"
  }
]
```

MongoDB 在每个数据库中都维护了一个特有的集合，名为 `system.indexes`。该集合将记录在数据库的所有集合上创建的全部索引。

集合 `system.indexes` 如同普通的集合一样。可以查看它的内容、执行查询，并且可以执行一些可在普通集合中完成的任务。

下面的样例列出了样例数据库中的索引：

```
$mongo
> use blog
> db.system.indexes.find()
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "blog.posts", "name" : "_id_" }
{ "v" : 1, "key" : { "_id" : 1 }, "ns" : "blog.authors", "name" : "_id_" }
```

blog 数据库中并未含有任何用户自定义的索引，但它自动为两个集合 `posts` 和 `authors` 的 `_id` 字段创建了索引。不需要做任何操作进行创建或删除 ID 索引；MongoDB 将在集合创建时创建该索引，在集合删除时删除该索引。

当在某个元素上定义索引时，MongoDB 将构造一个内部的二叉树索引，并用它高效地定位文档。如果未找到合适的索引，MongoDB 将扫描集合中的所有文档，搜索满足查询的记录。

10.3.2 创建简单的索引

MongoDB 提供了 `ensureIndex()` 函数，用于向集合中添加新的索引。该函数首先将检查在目前的集合中是否已经创建了该索引。如果是，那么 `ensureIndex()` 只是返回该索引。这意味着可以调用 `ensureIndex()` 任意次，它不会在集合中创建大量额外的索引。

下面的样例定义了一个简单的索引：

```
$mongo
> use blog
> db.posts.ensureIndex({Tags:1})
```

该例在 `Tags` 字段上创建了一个简单的升序二叉树索引。如果希望创建降序的索引，那么需要做出一点改动：

```
> db.posts.ensureIndex({Tags:-1})
```

为了索引嵌入式文档中的字段，可以使用普通的点标记寻址模式；例如，如果子文档 `comments` 中有一个 `count` 字段，那么可以使用下面的字段索引它：

```
> db.posts.ensureIndex({"comments.count":1})
```

如果指定的文档字段是数组类型，那么该索引将把数组中的所有元素作为不同的索引条目添加到其中。这被称为多键索引，每个文档都将被链接到索引中的多个值。返回到之前的内容，检查 `explain()` 的输出，你会看到其中提到了多值索引。

MongoDB 提供了一个特殊的操作符 `all`，用于选择包含了参数所提供的所有数据项的文档。在 `blog` 数据库中，我们使用 `posts` 集合中的一个称为 `Tags` 的元素。该元素中包含文档所有的标签。下面的查询将找出所有同时包含 `sailor` 和 `moon` 标签的文章：

```
> db.posts.find({Tags:{$all: ['sailor', 'moon']}})
```

如果不在 `Tags` 字段上使用多值索引，那么查询引擎将不得不扫描集合中的所有文档，检查这些数据项是否存在于文档中，如果是，就检查这两个数据项是否都存在。

10.3.3 创建复合索引

你可能会尝试为查询中用到的所有字段都分别创建索引。如果不考虑太多的话，这种方式可以加快查询的速度，但遗憾的是，这对数据库数据的添加和删除造成了巨大的影响，因为这些索引在每次修改操作执行时都将被更新。另外还要注意，在 MongoDB 2.4 及其之前的版本中，查询过程只会使用一个索引，所以添加许多小的索引通常并不会改善查询性能。

复合索引提供了一种减少集合中索引数目的好方法，它允许将多个字段结合在一起创建一个索引，所以应该尽量使用复合索引。

复合索引主要有两种类型：子文档索引和手动定义的复合索引。

MongoDB 制定了一些规则，通过这些规则可以在查询未使用所有复合索引键的情况下使用复合索引。了解了这些规则之后，就可以构造出一组复合索引，覆盖到所有希望在集合中执行的查询，而不是需要单独索引每个元素(从而避免对插入/更新操作的性能造成负面影响)。

不过复合索引在使用索引进行排序时就不那么有用了。排序并不善于利用复合索引，除非数据项的列表和排序与索引结构匹配。

使用子文档作为索引键时，构建多键索引的元素顺序，将与它们出现在子文档内部 BSON 数据存储中的顺序相同。在许多情况下，这种方式都无法对创建索引的过程进行充分控制。

为了解决这个限制，并保证查询使用以目标方式构建的索引，需要保证构建的查询的文档结构与创建索引所使用的子文档结构相同，如下所示：

```
> db.articles.find({author:{name: 'joe', email: 'joe@blogger.com'}})
```

还可以为所有希望用于创建复合索引的字段显式命名，然后指定这些字段的结合顺序。下面的样例演示了如何手动构建复合索引：

```
> db.posts.ensureIndex({"author.name":1, "author.email":1})
```

10.4 指定索引选项

创建索引时可以指定几个有趣的选项，例如创建唯一索引或启用后台索引处理；接下来将学习这些选项。在 `ensureIndex()` 函数中可以将这些选项指定为额外的参数，如下所示：

```
> db.posts.ensureIndex({author:1}, {option1:true, option2:true, ..... })
```

10.4.1 使用{background: true}在后台创建索引

第一次使用 `ensureIndex()` 函数创建索引时，服务器必须读取集合中的所有数据并创建指定的索引。默认情况下，索引构建将在前台完成，集合所在数据库上的所有操作都将被阻塞，直至索引操作完成。

MongoDB 因此引入了一个特性，它允许索引构建在后台执行。在索引构建的过程中，其他连接在数据库上的操作并不会被阻塞。直到索引构建完成，查询才会使用该索引，但服务器允许读和写操作继续执行。一旦索引操作完成之后，所有需要使用索引的查询都将立即开始使用它。值得注意的是，索引在后台构建时将会花费更长的时间。

因此，你可能希望寻找构建索引的其他策略。最好的策略可能就是在复制集中轮流执行索引构建。实施步骤是：一次停止一台辅助服务器，并移除 `--replSet` 参数，然后使用另一个端口启动它，将它临时作为一个独立运行的节点。然后就可以在该辅助服务器节点上随意执行索引构建。一旦索引构建完成，就以正常的方式重新启动该节点，它将重新加入复制集并更新数据。为所有的辅助服务器节点重复该操作过程。最后，停止主服务器并同样移除 `--replSet` 参数并执行索引过程，然后将其按正常步骤加入复制集中。通过以这种方式在复制集中轮流执行索引构建，将不会产生任何宕机或中断时间！

注意：

索引将在后台构建。不过，初始化该请求的连接将被阻塞(如果是在 MongoDB shell 中执行命令的话)。在该连接上执行的命令，直到索引操作完成后才会返回。乍一看，这似乎是相互矛盾的，因为它是一个后台索引。不过，如果你同时打开另一个 MongoDB shell，你会发现该集合上的查询和更新在索引构建过程中都可以正常执行。只有执行它的连接才会被阻塞。这不同于只使用 `ensureIndex()` 命令时的行为，此时索引将在前台运行，第二个 MongoDB shell 中的操作也会被阻塞。

终止索引进程

如果认为当前的索引进程已经挂起或者花费了太长时间，那么可以终止该进程。通过调用 `killOp()` 函数可以实现：

```
> db.killOp(<operation id>)
```

为了执行 `killOp()` 命令，首先需要知道操作 ID。通过执行 `db.currentOp()` 命令可以得到目前 MongoDB 实例中运行的所有操作的列表。

注意在调用 `killOp()` 命令时，已完成的部分索引也将被删除。这将阻止数据库中出现损坏或无关的数据。

10.4.2 使用 `{unique: true}` 创建唯一键索引

指定 `unique` 选项之后，MongoDB 将创建一个所有键都必须不同的索引。这意味着如果尝试插入一个文档，并且该文档中的键与现有文档中的键相同，那么 MongoDB 将会返回一个错误。这对于希望保持唯一的字段是非常有用的，例如保证任何两个人都不能有相同的 ID。

不过，如果希望在已经填充了数据的集合中创建唯一索引，那么必须保证已经去除了重复的键值。在此情况下，如果任何两个键不是唯一的，那么索引创建操作将会失败。

选项 `unique` 适用于简单索引和复合索引，但不适用于多键值索引，因为这样并不合理。

如果某个文档在插入时缺少作为唯一键的字段，那么 MongoDB 将会自动插入该字段并将值设置为 `null`。这意味着，集合中只能插入一个缺少该键的文档；出现任何其他 `null` 值都意味着该键不是唯一的。

10.4.3 使用 `{dropdups: true}` 自动去重

如果希望在存在重复键值的字段上创建唯一索引，那么可以指定 `dropdups` 选项。该选项将告诉 MongoDB 删除会引起索引创建失败的文档。在此情况下，MongoDB 将保留它以集合自然顺序找到的第一个文档，然后去除任何其他会违反索引限制的文档。

警告：

在使用 `dropdups` 选项时要非常小心，因为它将从集合中删除文档。

在使用该选项之前一定要非常小心地检查集合中的数据；否则，你可能会得到不可预料的(更不要提有可能是不希望出现的)结果。在希望使用唯一键的集合中执行分组查询是一个非常好的注意；这将可以帮助在执行该选项之前，确定会被认为是重复的那些文档。

10.4.4 使用{sparse: true}创建稀疏索引

有时可能需要只为具有某个字段的文档创建索引。例如，你希望对邮件进行索引，但并不是所有邮件都有 CC 或 BCC 字段。如果在 CC 或 BCC 字段上创建索引，那么所有文档都会被添加一个 null 值，因此可以考虑使用稀疏索引。这是一种节省空间的机制，因为只对有效的文档进行索引，而不是所有文档。当然，它只对所有使用稀疏索引的查询有影响；因为对于某些文档，该查询可能并未进行检查。

10.4.5 TTL 索引

在计算机术语中，TTL(Time To Live, 生存时间)是一种为特定数据块或请求赋予时间范围的方式，它将指定一个时间点，一旦超过该时间点，数据将变为无效。这对于存储在 MongoDB 实例中的数据也是非常有用的，因为通常自动删除旧数据是非常好的一种方式。为了创建 TTL 索引，必须在单索引(非复合索引)中添加 expireAfterSeconds 标志和对应的秒数。这将表示任何索引字段值大于指定的 TTL 值的文档，都将在 TTL 删除任务下一次执行时被删除。因为删除任务每 60 秒运行一次，因此旧文档在被删除之前可能有一些延迟。

警告：

被索引的字段必须是 BSON 日期类型，否则将不会被删除。在下一个样例中，从 shell 查询时 BSON 日期将被显示为 ISODate。

假设我们希望自动删除 blog 数据库的 comments 集合中时间戳超过特定时间的评论。从 comments 集合中找到下面的样例文档：

```
> db.comments.find();
{
  "_id" : ObjectId("519859b32fee8059a95eeace"),
  "author" : "david",
  "body" : "foo",
  "ts" : ISODate("2013-05-19T04:48:51.540Z"),
  "tags" : [ ]
}
```

假设我们希望将创建时间大于 28 天的评论全部删除。计算出 28 天为 2419200 秒。然后创建下面这样的索引：

```
> db.comments.ensureIndex({ts:1},{ expireAfterSeconds: 2419200})
```

当该文档存在的时间超过 2419200 秒后，它将被删除。可以使用下面的语法创建一个时间超过 2419200 秒的旧文档：

```
date = new Date(new Date().getTime()-2419200000);
db.comments.insert({ "author" : test, "body" : "foo", "ts" : date, "tags" : [ ] });
```

现在等待 1 分钟，然后该文档将会被删除。

注意：

使用 TTL 索引时，MongoDB 将在指定的集合中设置 usePowerOfSizes 标志。这将标准化每个文档的存储空间。这种方式允许在文档被删除以后，空出的空间可以被将来的文档高效地利用。

10.4.6 文本索引

MongoDB 2.4 引入了一种新的索引类型——文本索引，通过它可以执行全文搜索！第 8 章已经对文本索引特性进行了详细讲解，但值得在这里讲解的是：如何对它进行优化。文本搜索是大家对 MongoDB 期待已久的一个特性，因为通过它可以在一个大的文本块中搜索特定的单词或文本。使用文本搜索的最佳例子就是博客文章正文的搜索功能。这类搜索允许你在文档的一个字段(例如正文)或多个字段(例如正文和评论)中搜索单词或短语。运行下面的命令以创建文本索引：

```
> db.posts.ensureIndex( { body: "text" } )
```

注意：

文本索引是大小写不敏感的，这意味着它将忽略大小写：MongoDB 和 mongodb 被认为是相同的文本。

现在通过 text 命令使用文本索引进行搜索。通过使用 runCommand 语法，使用 text 命令并提供搜索的目标值即可：

```
> db.posts.runCommand( "text", { search: "MongoDB" } )
```

文本搜索的结果将按照相关性返回。

警告：

MongoDB 文本搜索相对来说仍是一个新功能，并且正在发生快速变化。

10.4.7 删除索引

可以选择删除集合中的所有索引或某个特定的索引。使用下面的函数删除集合中的所有索引：

```
> db.posts.dropIndexes()
```

为了从集合中删除单个索引，可以使用下面的语法(对应于使用 ensureIndex() 函数创建索引的语法)完成：

```
> db.posts.dropIndex({"author.name":1, "author.email":1});
```

10.4.8 重建集合索引

如果怀疑集合中的索引已经损坏，例如，查询后得到不一致的结果，那么可以强制重建受影响的索引。

这将强制 MongoDB 删除并重建特定集合中的所有索引(如何检测和解决索引相关的问题请参见第 9 章)，如下所示：

```
> db.posts.reIndex()
{
  "nIndexesWas" : 2,
  "msg" : "indexes dropped for collection",
  "nIndexes" : 2,
  "indexes" : [
    {
```



```

    "key" : {
      "_id" : 1
    },
    "ns" : "blog.posts",
    "name" : "_id_"
  },
  {
    "key" : {
      "Tags" : 1
    },
    "ns" : "blog.posts",
    "name" : "Tags_1"
  }
],
"ok" : 1
}

```

输出中列出了所有使用该命令重建的索引，包括键名。另外，`nIndexWas` 字段显示了在运行该命令之前，有多少索引存在，而 `nIndex` 字段将提供命令完成之后集合中索引的数目。如果两个值不同，那么表示在重建集合中某些索引的过程中出现了问题。

10.5 MongoDB 选择索引的方式

当数据库系统需要运行查询时，它必须制定出一个查询计划，该计划代表了执行查询所需的步骤。每个查询都可能多个产生相同结果的查询计划。不过，每个计划都可能含有比其他查询计划执行代价更高的元素。例如，扫描集合中的所有记录是一个高代价的操作，任何采用这种方式的计划都可能花费很长的时间。这些计划也可以在查询或排序操作中使用备选的索引。

道路方向是该概念一个很好的例证。如果希望从一个角落到达它的斜对面，那么可以采用“先左后右”或“先右后左”的方式，这两个计划是等同的。不过，如果其中有一条路径有两个停止标志，而另一个没有，那么前一种方式就是代价更高的计划，而后一种是可用的最好计划。在执行查询时，集合扫描就相当于潜在的终止标志。

MongoDB 含有一个称为查询分析器的模块。该模块将接受查询和查询的目标集合，然后产生一组 MongoDB 将要执行的计划。本章之前描述的 `explain()` 函数列出了指定查询使用的计划及其产生的所有备用计划。

MongoDB 还含有一个查询优化器模块。该模块的工作是为特定的查询选择最优的执行计划。在大多数关系数据库系统中，查询优化器将使用表中键的分布统计信息、记录数目、可用索引、之前选择的有效性和权重因子的分类计算每种方式的代价。然后选择使用代价最小的计划。

MongoDB 中的查询优化器比典型的 RDBMS 查询分析器更愚蠢也更简单。例如，它不使用基于代价的方法选择执行计划；相反，它将并行运行所有的计划，并使用最快返回结果的计划，在成功者返回之后终止所有其他计划。因此，MongoDB 的查询分析器使用简单的机制来保证总是得到最快的结果(也最聪明)，然后缓存这些计划的执行结果，并在接下来的 1000 次写入操作过程中重用它们，直到 `explain()` 函数被执行或者在那个集合上添加、删除或重建索引。

10.6 使用 hint()强制使用特定的索引

MongoDB 中的查询优化器将从备用索引中选择一个最佳索引用于查询。它将使用之前讲述的方法尝试为指定的查询匹配最佳索引或一组索引。不过，有时查询优化器无法做出正确的选择，此时就需要为该组件提供帮助。

可以为查询优化器提供提示，避免组件做出不同的选择。例如，如果已经使用 `explain()` 显示出查询正在使用的索引，并且希望使用一个不同的索引，那么可以强制优化器这么做。

下面通过一个样例演示如何实现。假设包含了 `name` 和 `email` 字段的子文档 `author` 上有一个索引。再假设你已经创建了下面的索引：

```
> db.posts.ensureIndex({author.name:1, author.email:1})
```

可以使用下面的提示强制查询优化器使用已定义的索引：

```
> db.posts.find({author:{name:'joe', email:'joe@mongodb.com'}}).hint({author.name:1, author.email:1})
```

如果出于某种原因不希望使用任何索引，即希望使用集合文档扫描作为选择记录的方式，那么可以使用下面的提示：

```
> db.posts.find({author:{name:'joe', email:'joe@mongodb.com'}}).hint({$natural: 1})
```

10.7 优化小对象的存储

索引是加快数据查询的关键，但另一个可能影响应用性能的因素是查询所访问的数据的大小。与使用固定模式的数据库系统不同，MongoDB 将把每条记录的所有模式数据都存储在记录中。因此，对于每个字段数据内容都比较大的大记录来说，模式数据与记录数据的比例是较小的；不过，对于含有小数据的小记录，该比例就会变得非常大。

选择一种非常适用于 MongoDB 的应用——日志记录，考虑它的一个常见问题。MongoDB 非比寻常的写入速率可以将流事件作为小文档快速地写入集合中。不过，如果希望进一步优化该操作速度，可以使用下面的方法：

首先，可以考虑执行批量插入。MongoDB 提供了多文档 `insert()` 调用。通过该调用可以将多个事件同时添加到集合中。它将减少数据库接口 API 的调用次数。

其次(也更重要)，可以减小字段名的大小。如果字段名更小，MongoDB 在将记录属性写入到磁盘之前，可以在内存中保存更多的记录。这将使整个系统更高效。

例如，假设有一个集合用于记录三个字段：时间戳、计数器和用于表示数据来源的 4 字符串。数据的总存储大小如表 10-2 所示。

表 10-2 日志样例集合存储大小

字 段	大 小
时间戳	8 字节
整数	4 字节
字符串	4 字节
总共	16 字节

如果使用 `ts`、`n` 和 `src` 作为它们的字段名，那么字段名的总大小为 6 字节。该大小相对于数据大小是一个相对较小的值。但假设你决定使用的字段名为 `WhenTheEventHappened`、`NumberOfEvents` 和 `SourceOfEvents`。此时，字段名的总大小为 48 字节或三倍于数据自身的大小。如果在集合中写入 1TB 的数据，那么将存储 750GB 的字段名，但只有 250GB 的真正数据。

这种方式将浪费更多的磁盘空间，还将影响系统性能的所有其他方面，包括索引大小、数据传输时间和(可能更重要)缓存数据文件的系统宝贵 RAM 的使用。

在日志应用中，还需要避免在写入记录时向集合中添加索引；如之前解释的，索引需要花费时间和资源进行维护。相反，你应该在开始分析数据之前立即添加索引。

最后，应该考虑使用某种模式将事件流分到多个集合中。例如，可以将每天的事件写入一个不同的集合中。更小的集合将花费更少的时间进行索引和分析。

10.8 小结

在本章，我们学习了一些用于追踪 MongoDB 中性能不佳查询的工具，以及优化这些慢速查询的潜在解决方案。还学习了一些优化数据存储的方式，例如，确认我们完全利用了 MongoDB 服务器可用的资源。

本章描述的特定技术将使你可以优化查询和调节 MongoDB 系统的数据存储。对于不同的应用，优化的最佳方式也不同，具体取决于许多因素，包括应用访问类型、数据访问模式、读/写比等。

复 制

如同许多关系数据库一样，MongoDB 支持以实时或接近实时的方式，将数据内容复制到另一台服务器中。MongoDB 的复制特性非常易于设置和使用。在 MongoDB 的关键特性中，除了复制还有分片，这更强调了该数据库既是一个 Web 2.0 数据存储系统，也是一个基于云的数据存储系统。

有许多场景可能需要使用复制，所以 MongoDB 对复制的支持必须足够灵活，能够处理所有的场景。MongoDB Inc.对 MongoDB 的架构已经进行了增强，保证它的复制特性能够满足今天的需求。

在本章，我们将讲解 MongoDB 中复制的基础知识，包括以下主题：

- MongoDB 复制的定义
- 主服务器的定义
- 辅助服务器的定义
- oplog 的定义

注意：

复制是 MongoDB 中不断发展的一个特性，随着产品的发展，复制的工作方式也可能会做出一些改变。对于数据库服务器集群来说更是如此。MongoDB Inc.正在投入大量的精力，保证 MongoDB 满足和超过所有人对可扩展性和可用性的期待；对复制的支持是 MongoDB Inc.用来满足这些期待的关键特性。

在学习复制设置细节之前，让我们先学习各种设置希望实现的目标。我们还将列出复制在 MongoDB 中如何运行的一些基础知识，并学习 oplog 及其在复制集成员之间进行数据复制时的作用。这些主题组成了理解复制的基础。

11.1 MongoDB 复制的目标

在所有特性中，复制被用于实现可扩展性、可持久性/可靠性和隔离性。在接下来的内容中，我们将学习如何使用复制实现这 3 个目标，同时指出其中潜在的陷阱和需要避免的错误。

11.1.1 改善可扩展性

尤其是对于 Web 应用来说，可扩展性是一个关键的设计需求，尤其是那些严重依赖于后台数据库的应用。复制可以通过两种方式帮助创建更具扩展性的应用：

- **提高冗余度：**复制可以通过运行多个数据中心的方式提高冗余度。通过这种方式，保证每个数据中心都有一份数据的本地副本，这样应用就可以快速地访问它们。用户可以连接到离它最近的数据中心，减小延迟。
- **改善性能：**复制在某些特定的情况下，可以帮助改善应用的性能。对于一个主要以数据库读取为主的大 Web 应用，如果希望在多个数据库服务器之间派发查询以提高并发度的话，更是如此。对于广泛使用不同工作集的查询负载来说也是这样，例如报表或聚集。

注意：

MongoDB 还支持分片特性，它也是一个用于帮助创建更具扩展性的应用，无论是否使用复制都能提供高扩展性。关于在 MongoDB 中如何结合使用分片和复制的更多信息请参考第 12 章。

11.1.2 改善持久性/可靠性

复制通常用于防止硬件故障或数据库损坏，同时为备份和其他具有重要影响的维护活动提供灵活性，只对系统造成一点儿影响或没有影响，因为这些任务可以分别在复制集的成员中单独执行，而不会影响整个复制集。人们以这种方式使用复制的一些样例包括：

- 希望拥有数据库的一个副本，该副本将延迟运行。你可能希望保护自己避免应用中的缺陷，或者提供一种简单的机制，通过标志出所有数据集的查询结果的不同，提供趋势信息。还可以为人为错误提供一个安全的环境，避免完全从备份完全恢复的需求。
- 希望拥有一个备份系统，以避免系统出现失败的情况。为了避免由于系统故障所造成的影响，避免使用普通的备份模式进行长时间恢复，可以使用复制作为备份。
- 出于管理的目的，希望拥有一个冗余系统。使用复制之后，就可以在各个节点之间轮流执行备份或升级这样的管理任务。

11.1.3 提供隔离性

对于某些进程来说，如果在生成数据库中运行它们，那么将对数据库的性能或可用性造成巨大的影响。可以使用复制创建一个同步的副本，将进程从生成数据库隔离出来，例如：

- 希望运行报表或备份，而不希望影响生产系统的性能：维护一台隐藏的辅助复制服务器将可以帮助从报告系统中隔离出查询，并保证月末的报告不会延迟或影响正常操作。

11.2 复制基础

如你所见，复制集(或 replSet)是一种创建多个 MongoDB 实例的方式，这些实例将拥有相同的数据(冗余)和其他相关设置。除了知道这一点，还需要了解 MongoDB 如何完成复制，这样你就会明白应该如何管理复制集。

你已经注意到了 MongoDB 中复制的目标，并且已经早就开始使用 MongoDB，那么你会知道有许多不同种方式可以完成复制，包括：

- 主/从复制
- 主/主复制
- 复制对

复制的这些方法都被复制集的概念所取代。在 MongoDB 中，一个复制集由一个主节点和

许多辅助或仲裁节点组成。复制集成员的个数应该为奇数，这意味着最少应该有 3 个节点。之所以有该要求是因为，MongoDB 中有一条规则：主服务器必须能够看到其他的大多数节点，才能继续成为主服务器。这条规则主要是为了避免“脑裂”(Split Brain)问题，也就是当网络出现问题时，有两台服务器成为主服务器的情况，如图 11-1 所示。

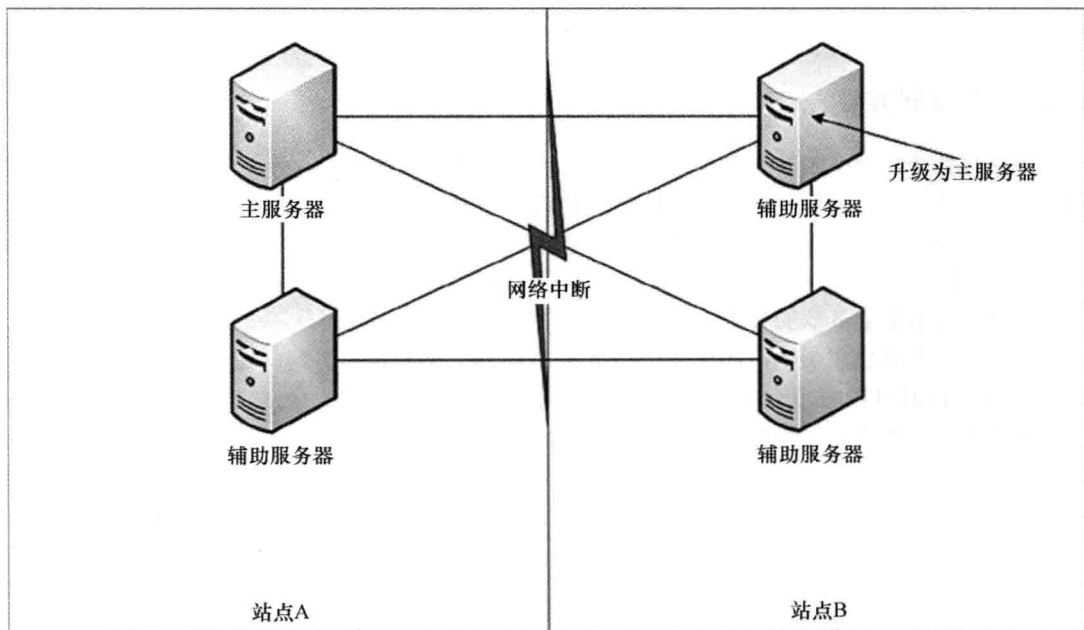


图 11-1 脑裂问题

11.2.1 主服务器的定义

在复制集的术语中，主服务器是在特定的时间内复制集的数据来源。它是复制集中唯一可以写入的节点，所有其他的节点都将从这里复制出它们的数据。主服务器由所有投票成员中的大多数选择产生，这被称为法定人数(quorum)。

一旦选择主服务器，所有辅助服务器将使用它作为复制数据的来源，并且所有的写入都必须被重定向至该成员。

11.2.2 辅助服务器的定义

辅助服务器成员是一个具有数据的非主服务器成员，理论上它可以成为主服务器(除了一些例外)。它是一个可读取的节点，同时它将以接近于实时的方式从主服务器复制数据。默认情况下，如果连接到主服务器但不使用任何读取偏好，那就不能执行读操作。这是因为如果复制过程中有延迟存在，那么你可能读取的是旧数据。可以使用 `rs.slaveOK()` 将当前连接设置为可从辅助服务器读取数据。或者如果使用的是某种驱动，那么也可以设置读取偏好，本章稍后将进行讲解。

注意：

主服务器的概念是并且应该是短暂的。从理想的角度来说，不应该固定地认为哪个节点是主服务器。在复制集中，所有的辅助服务器为了保证复制成功，都将写入与主服务器相同的数据。不过，如果辅助服务器能力不足，那么它们即使变成主服务器，可能也无法胜任这样的工作。

11.2.3 仲裁服务器的定义

仲裁服务器是一个不含有数据的节点，它用于提供额外的投票，用于帮助维持复制集选举的大多数原则。它不会投出决定性的投票或者直接决定哪个节点是主服务器，但会参与并作为投票成员中的一员，决定哪个节点成为主服务器。如之前所描述的，仲裁服务器用于帮助避免“脑裂”问题。考虑图 11-2 所示的图表。有了站点 A 中的仲裁服务器，我们总是可以在某一边成功完成大多数服务器的投票选举。这意味着当网络出现问题时，不会出现两个主服务器。我们可以进一步增加冗余性，在第 3 个站点 C 中添加一个仲裁服务器。这样如果站点 A 宕机，站点 B 和 C 仍然可以成功完成投票选举。通过以这种方式使用第 3 个站点，哪怕失去任何一个站点的连接，服务器都可以继续正常运行。

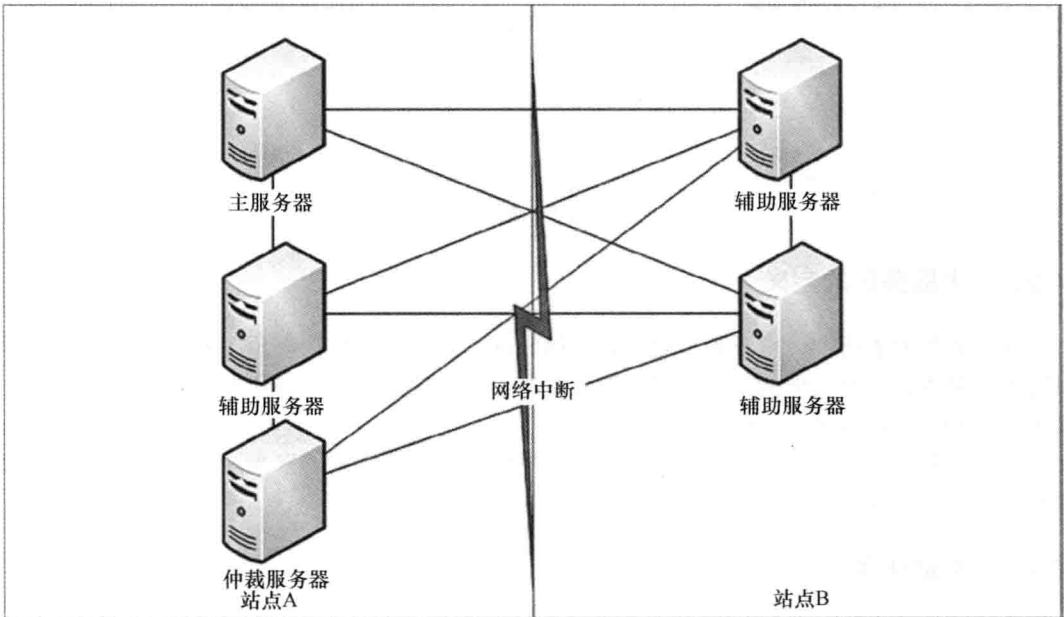


图 11-2 解决脑裂问题

11.3 深入学习 oplog

简单地说，oplog(操作日志)就是一个固定大小的集合，保存了主服务器实例对数据库做出修改的记录，目的是在辅助服务器上重做这些操作，保证数据库处于一致状态。服务器的每个成员都将维护自己的 oplog，并且辅助服务器将查询主服务器(或者其他数据更新的辅助服务器)的 oplog，从而获得新的条目并应用到自己的数据库副本中。

oplog 将为每个条目创建一个时间戳。通过这种方式，辅助服务器可以记录从上一次读取开始过去了多久，以及有多少 oplog 需要读取。如果终止辅助服务器并在较短的时间内重启它，它将从主服务器的 oplog 中获取在它离线的时间内所发生的所有修改。

因为具有一个无限大的 oplog 是不现实的，所以 oplog 通常具有固定的大小。

可以将 oplog 看作主服务器实例最近活动的记录；如果记录太小，那么记录中的某些操作在被应用到辅助服务器之前将会丢失。如果当前实例的 oplog 尚未创建，那么使用--oplogSize 启动选项可以设置 oplog 的大小，以 MB 为单位。默认在 Linux 和 Windows 64 位系统中，oplogSize 将被设置为数据存储可用的空闲磁盘空间的 5%。如果系统是写入/更新密集型的，那么可能需要增加该大小，以保证辅助服务器在合理的时间范围内处于离线状态而不会丢失数据。

例如，如果需要从辅助服务器执行日常备份，这个任务需要花费一小时，那么 oplog 的大小将不得被重新设置，该大小应该允许辅助服务器离线一小时再加上一定的安全边际时间。

计算 oplog 的合适大小时，考虑主服务器上所有数据库的更新频率是非常关键的。

通过执行 db.printReplicationInfo() 命令可以得到一些 oplog 大小方面的参考，该命令将运行在主服务器上：

```
$mongo
> db.printReplicationInfo()
configuredoplog size: 15000MB
log length start to end: 6456672secs (1793.52hrs)
oplog first event time: Wed Mar 20 2013 17:00:43 GMT+1100 (EST)
oplog last event time: Mon Jun 03 2013 09:31:55 GMT+1000 (EST)
now: Mon Jun 03 2013 20:22:20 GMT+1000 (EST)
```

该命令将显示出 oplog 当前的大小，以及以当前的更新速率 oplog 被填满所需的时间。从该信息中可以估计出是需要增加还是减小 oplog 的大小。还可以通过查看 MongoDB 监控服务中的 repl lag 区域，检查指定的辅助服务器成员落后主服务器多少。如果尚未安装 MMS，那么建议现在就安装，随着 MongoDB 集群变得越来越大，MMS 这种工具提供的状态信息就越重要。关于更多背景知识，请参考第 9 章的 9.17 节。

11.4 实现复制集

本节将讲解如何创建一个简单的复制集配置。另外还将讲解如何从集群中添加和删除成员。正如之前讨论过的，复制集由一个主服务器、多个辅助服务器(从主服务器复制写操作)或仲裁服务器组成(见图 11-3)。

复制集还提供了主动成员和被动成员。当前的主服务器不可用时，被动辅助服务器不会参与新的主服务器的选举；相反，它们的作用与隐藏成员一致，可被用于报表或备份数据集。

复制集的成员不需要在启动时被设置为复制集成员。相反，配置将通过服务器端命令(通过普通的服务器接口发送)完成。这将使配置管理工具的创建变得容易，通过它们可以动态地配置和管理集群。

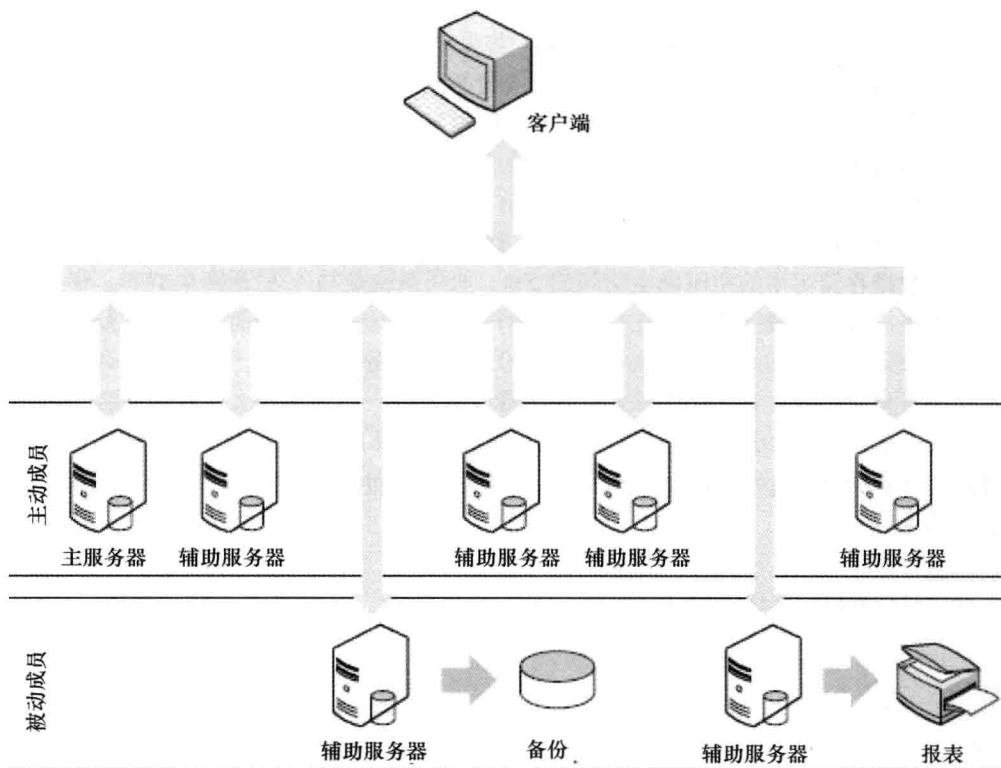


图 11-3 使用了复制集的集群

接下来，我们将学习如何完成以下步骤：

- (1) 创建复制集
- (2) 向复制集添加服务器
- (3) 向复制集添加仲裁服务器
- (4) 在服务器上检查和执行操作
- (5) 配置复制集中的单个成员
- (6) 从应用中连接复制集
- (7) 在应用中设置读偏好
- (8) 在应用中设置写顾虑(Write Concern)
- (9) 结合读偏好和写顾虑使用复制集标签
- (10) 使用 Web 界面检查复制集的状态

11.4.1 创建复制集

学习如何创建复制集的最佳方式就是学习样例。在下面的样例中，我们将创建一个名为 testset 的复制集。该复制集将拥有 3 个成员(两个主动成员和一个被动成员)。表 11-1 列出了该复制集的成员。

表 11-1 配置复制集

服 务	守护进程	地 址	数据库路径
主动成员 1	mongod	[hostname]:27021	/db/active1/data
主动成员 2	mongod	[hostname]:27022	/db/active2/data
被动成员 1	mongod	[hostname]:27023	/db/passive1/data

在复制集中可以使用 `localhost` 作为标识符，但只有所有数据库实例都设在同一个服务器上时才可以。因为复制集中的每个成员都必须能够通过主机名联系其他 MongoDB 实例，这样复制集才能正常工作。

通常使用复制集时采用的是主机名；通过 `hostname` 命令可以找到当前主机名，如下所示：

```
$hostname
Pixl.local
```

在接下来的样例中，将使用 `hostname` 命令返回的结果替换[hostname]。

11.4.2 启动服务器成员

第一步是启动第一个主动成员。打开一个终端窗口并输入下面的命令：

```
$ mkdir -p /db/active1/data
$ mongod --dbpath /db/active1/data --port 27021 --replSettestset
```

选项 `--replSet` 将告诉实例它所加入的复制集的名称。这是复制集的第一个成员，所以可以给予它其他成员的地址，哪怕这些成员都尚未启动。只有一个成员地址是必需的，不过也可以通过以逗号分隔的方式提供多个成员的名称，如下所示：

```
$ mongod --dbpath /db/active1/data --port 27021 --replSettestset
```

注意：

如果不希望这些 MongoDB 实例中的每个都占用一个 shell 实例，那么可以添加 `--fork` 和 `--logpath <file>` 选项，告诉该实例在后台运行并将日志输出到指定的文件中。

为了保持事情简单，该例只会依赖于一个地址。下一步是启动其他成员。另外打开两个终端窗口，并在第一个窗口中输入下面的命令以启动第二个成员：

```
$ mkdir -p /db/active2/data
$ mongod --dbpath /db/active2/data --port 27022 --replSettestset
```

接着在第二个窗口中输入下面的命令以启动最后一个(被动)成员：

```
$ mkdir -p /db/passive1/data
$ mongod --dbpath /db/passive1/data --port 27023 --replSettestset
```

此时，我们已经启动了 3 个服务器实例，并且它们之间可以进行通信；不过，复制集尚未正确运行，因为我们并未初始化服务器，也并未告诉它每个成员的角色和责任。

下面需要连接到其中一个服务器并初始化复制集。下面的代码选择连接到第一个服务器：

```
$mongo [hostname]:27021
```

接下来，需要初始化复制集中的第一个成员，创建它的 `oplog` 和默认的配置文档。在日志文件中可以看到 MongoDB 实例建议我们要做的事情：

```
Mon Jun 3 21:25:23.712 [rsStart] replSet can't get local.system.replsetconfig from self or any seed (EMPTYCONFIG)
Mon Jun 3 21:25:23.712 [rsStart] replSet info you may need to run replSetInitiate -- rs.initiate() in the shell -- if that is not already done
```

运行 `rs.initiate` 命令：

```
> rs.initiate()
{
  "info2" : "no configuration explicitly specified -- making one",
  "me" : "[hostname]:27021",
  "info" : "Config now saved locally. Should come online in about a minute.",
  "ok" : 1
}
```

最后，检查服务器的状态，判断它是否已经设置成功：

```
> rs.status()
{
  "set" : "testset",
  "date" : ISODate("2013-06-03T11:28:58Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "[hostname]:27021",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 264,
      "optime" : {
        "t" : 1370258919,
        "i" : 1
      },
      "optimeDate" : ISODate("2013-06-03T11:28:39Z"),
      "self" : true
    }
  ],
  "ok" : 1
}
```

该输出表示全部都没有问题：已经成功配置、创建和初始化一个新的复制集。记住应该使用自己的机器名替换 `[hostname]`，因为 `localhost` 和 `127.0.0.1` 在本例中都无法正确运行。

11.4.3 向复制集中添加服务器

在启动新的复制集之后，需要向其中添加成员。首先添加第一个辅助服务器。通过使用 `rs.add()` 命令并提供主机名和该实例的端口来完成。连接到主服务器并执行下面的命令：

```
$ mongo [hostname]:27021
> rs.add("[hostname]:27021")
{ "ok" : 1 }
```

因为该节点需要启动自身、创建 `oplog` 并让自身就绪，所以需要等待一到两分钟。在该节点作为辅助服务器成功启动之前，可以使用 `rs.status()` 监控进度：

```
> use admin
> rs.status() {
  "set" : "testset",
  "date" : ISODate("2013-06-03T11:36:37Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "[hostname]:27021",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 723,
      "optime" : {
        "t" : 1370259331,
        "i" : 1
      },
      "optimeDate" : ISODate("2013-06-03T11:35:31Z"),
      "self" : true
    },
    {
      "_id" : 1,
      "name" : "[hostname]:27022",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 66,
      "optime" : {
        "t" : 1370259331,
        "i" : 1
      },
      "optimeDate" : ISODate("2013-06-03T11:35:31Z"),
      "lastHeartbeat" : ISODate("2013-06-03T11:36:35Z"),
      "lastHeartbeatRecv" : ISODate("2013-06-03T11:36:36Z"),
      "pingMs" : 0,
      "syncingTo" : "[hostname]:27021"
    }
  ],
  "ok" : 1
}
```

现在添加第 3 个被动成员。首先使用 `rs.add()` 添加成员：

```
$ mongo [hostname]:27022
> rs.add("[hostname]:27022")
{ "ok" : 1 }
```

现在需要复制配置文档并修改它。运行下面的命令，创建一个名为 `conf` 的文档，它包含目前复制集的配置。

```
> conf = rs.conf()
{
  "_id" : "testset",
  "version" : 3,
```

```

"members" : [
  {
    "_id" : 0,
    "host" : "[hostname]:27021"
  },
  {
    "_id" : 1,
    "host" : "[hostname]:27022"
  },
  {
    "_id" : 2,
    "host" : "[hostname]:27023"
  }
]
}

```

加载配置文档之后，现在对它进行修改。我们希望将节点设置为隐藏的并且优先级为 0，这样它就不会被选举为主服务器。注意文档中有一个 **members** 数组，它包含服务器中每个成员的信息文档。使用数组操作符[]选择希望访问的成员。将数组中的第 3 个元素(数组索引为 2，因为数组从 0 开始计算的)的 **hidden** 值更新为 **true**。运行下面的命令：

```

> conf.members[2].hidden = true
true

```

使用相同的命令将它的 **priority** 设置为 0：

```

> conf.members[2].priority = 0
0

```

通过执行存储配置文档的命令可以显示出配置文档的内容：

```

> conf
{
  "_id" : "testset",
  "version" : 3,
  "members" : [
    {
      "_id" : 0,
      "host" : "[hostname]:27021"
    },
    {
      "_id" : 1,
      "host" : "[hostname]:27022"
    },
    {
      "_id" : 2,
      "host" : "[hostname]:27023",
      "hidden" : true,
      "priority" : 0
    }
  ]
}

```

如你所见，该成员已经设置了隐藏值并且将优先级设置为 0。现在需要做的就是使用该文档更新复制集的配置。将新的配置文档用作参数，执行 **rs.reconfig()** 命令。

```
> rs.reconfig(conf)
Tue Jun 4 20:01:45.234 DBClientCursor::initcall() failed
Tue Jun 4 20:01:45.235 trying reconnect to 127.0.0.1:27021
Tue Jun 4 20:01:45.235 reconnect 127.0.0.1:27021 ok
reconnected to server after rs command (which is normal)
```

整个复制集将先断开连接，然后再重新连接！这是由于重新配置引起的。

任何对复制集的修改都将引起复制集的重新配置，并重新选举主服务器。大多数情况下，之前的主服务器都将保持它的角色不变。现在如果再运行 `rs.conf()` 命令，你会看到新的复制集配置已经生效：

```
> rs.conf()
{
  "_id" : "testset",
  "version" : 4,
  "members" : [
    {
      "_id" : 0,
      "host" : "[hostname]:27021"
    },
    {
      "_id" : 1,
      "host" : "[hostname]:27022"
    },
    {
      "_id" : 2,
      "host" : "[hostname]:27023",
      "priority" : 0,
      "hidden" : true
    }
  ]
}
```

注意该复制集配置的版本号已经增加了。该操作在重配置的过程中自动完成，通过这种方式可以保证复制集成员不会使用错误的配置文档。

现在已成功创建一个拥有 3 个成员的复制集，其中有一个主动的主服务器和一个隐藏的被动服务器。

11.4.4 添加仲裁服务器

在复制集中添加一个仲裁服务器作为投票成员是非常容易的。首先创建一个新的成员：

```
$ mkdir -p /db/arbiter1/data
$ mongod --dbpath /db/ arbiter1/data --port 27024 --replSettestset --rest
```

创建新成员之后，使用 `rs.addArb()` 命令添加新的仲裁服务器：

```
> rs.addArb("[hostname]:27024")
{ "ok" : 1 }
```

如果现在运行 `rs.status()`，仲裁服务器将显示在输出中：

```
{
  "_id" : 3,
  "name" : "Pixl.local:27024",
  "health" : 1,
  "state" : 7,
  "stateStr" : "ARBITER",
  "uptime" : 721,
  "lastHeartbeat" : ISODate("2013-06-07T11:21:01Z"),
  "lastHeartbeatRecv" : ISODate("2013-06-07T11:21:00Z"),
  "pingMs" : 0
}
```

你可能已经意识到这里的问题：复制集中现在有 4 个节点。这是个偶数，而偶数是糟糕的事情！如果继续这样运行，MongoDB 节点将开始在日志中输出下面的内容：

```
[rsMgr] replSet total number of votes is even - add arbiter or give one member an extra vote
```

为了解决这个问题，需要添加奇数个成员；正如日志消息所建议的，其中一个解决方案是添加另一个仲裁服务器，但这并不是一个优雅的方法，因为它增加了不必要的复杂性。最佳的解决方案就是从投票成员中停止其中一个节点。将隐藏辅助服务器的 `votes` 值设置为 0 即可。设置 `votes` 值的方式与设置 `hidden` 和 `priority` 值的方式相同。

```
conf = rs.conf()
conf.members[2].votes = 0
rs.reconfig(conf)
```

就是这样。现在将被动节点完全变成了被动服务器：它永远不会变成主服务器；它被客户端看作复制集的一部分；并且不会参与选举，也不会被统计为选举的大多数。可以尝试关闭被动节点进行测试，仲裁服务器和其他两个节点会继续运行，主服务器也保持不变；而之前，主服务器将会让位，因为它无法看到大多数节点。

11.4.5 复制集链

通常，复制集成员会尝试从复制集的主服务器同步数据。但这不是复制集的辅助服务器同步数据的唯一服务器；它们也可以从其他辅助服务器同步数据。通过这种方式，所有的辅助服务器将组成一个“同步链”，每个节点都可以从复制集的其他辅助服务器同步数据。

11.4.6 管理复制集

MongoDB 提供了许多命令用于管理复制集的配置和状态。表 11-2 显示了可用于创建、操作和检测复制集中集群状态的可用命令。

表 11-2 操作和检测复制集的命令

命 令	描 述
<code>rs.help()</code>	返回该表中的命令列表
<code>rs.status()</code>	返回复制集当前的状态信息。该命令列出了每个成员服务器及其状态信息，包括最后联系时间。该调用可被用于提供整个集群的简单健康检查

(续表)

命 令	描 述
<code>rs.initiate()</code>	使用默认参数初始化复制集
<code>rs.initiate(replSetcfg)</code>	使用配置描述初始化复制集
<code>rs.add("host:port")</code>	使用含有主机名和特定端口(可选的)的简单字符串向复制集中添加成员服务器
<code>rs.add(membercfg)</code>	使用配置描述向复制集中添加成员服务器。如果希望指定特定的属性,那么必须使用这种方法(例如,新成员服务器的优先级)
<code>rs.addArb("host:port")</code>	添加新的成员服务器作为仲裁者。该成员不需要使用--replSet 选项;任何运行在可达机器上的 <code>mongod</code> 实例都可以执行该任务。注意该服务器必须对复制集中的所有成员都可达
<code>rs.stepDown()</code>	在复制集的主服务器成员中使用该命令时,将使主服务器放弃它的角色,并且在集群中重新选举新的主服务器。注意只有主动辅助服务器可用作主服务器的候选服务器,并且在 60 秒之内如果没有出现其他可用的成员,那么原有的主服务器将重新成为主服务器
<code>rs.syncFrom("host:port")</code>	使辅助服务器从指定的成员同步数据。可用于组成同步链
<code>rs.freeze(secs)</code>	冻结指定的成员,并使它在指定的秒数内无法成为主服务器
<code>rs.remove("host:port")</code>	从复制集中删除指定的成员
<code>rs.slaveOk()</code>	通过设置该选项,可以允许从辅助服务器读取数据
<code>rs.conf()</code>	重新显示当前复制集的配置结构。在获取复制集的配置结构时,该命令是非常有用的。该配置结构可以被修改,然后用作 <code>rs.initiate()</code> 的参数,从而修改结构的配置。该技术提供了从复制集中删除成员服务器的唯一方法;目前没有其他直接的方法可以完成该任务
<code>db.isMaster()</code>	该函数不只可作用于复制集;它是一个通用的复制支持函数,通过它,应用或驱动可以判断出被连接的特定实例在复制拓扑结构中是否是主服务器

下面将详细讲解表 11-2 中所列出的常用命令,包括它们可完成的任务以及如何使用它们。

1. 使用 `rs.status()` 检测实例的状态

从之前学习如何在复制集中添加成员的过程中,你应该已经注意到 `rs.status()` 可能是使用复制集时最常用的命令。通过它可以检测当前连接到的实例的状态,包括它在复制集中的角色:

```
> rs.status()
> rs.status();
{
  "set" : "testset",
  "date" : ISODate("2013-06-04T10:57:24Z"),
  "myState" : 1,
  "members" : [
    {
      "_id" : 0,
      "name" : "[hostname]:27021",
      "health" : 1,
      "state" : 1,
      "stateStr" : "PRIMARY",
      "uptime" : 4131,
      "optime" : Timestamp(1370340105, 1),
```



```

    "optimeDate" : ISODate("2013-06-04T10:01:45Z"),
    "self" : true
  },
  {
    "_id" : 1,
    "name" : "[hostname]:27022",
    "health" : 1,
    "state" : 2,
    "stateStr" : "SECONDARY",
    "uptime" : 3339,
    "optime" : Timestamp(1370340105, 1),
    "optimeDate" : ISODate("2013-06-04T10:01:45Z"),
    "lastHeartbeat" : ISODate("2013-06-04T10:57:23Z"),
    "lastHeartbeatRecv" : ISODate("2013-06-04T10:57:23Z"),
    "pingMs" : 0,
    "syncingTo" : "[hostname]:27021"
  },
  {
    "_id" : 2,
    "name" : "[hostname]:27023",
    "health" : 1,
    "state" : 2,
    "stateStr" : "SECONDARY",
    "uptime" : 3339,
    "optime" : Timestamp(1370340105, 1),
    "optimeDate" : ISODate("2013-06-04T10:01:45Z"),
    "lastHeartbeat" : ISODate("2013-06-04T10:57:22Z"),
    "lastHeartbeatRecv" : ISODate("2013-06-04T10:57:23Z"),
    "pingMs" : 0,
    "syncingTo" : "[hostname]:27021"
  }
],
"ok" : 1
}

```

如表 11-3 所示，样例中的每个字段都是有意义的。这些值可用于了解复制集中当前成员的状态。

表 11-3 rs.status 字段的值

值	描述
_id	复制集中该成员的 ID
Name	该成员的主机名
Health	replSet 的健康值
State	状态数值
StateStr	复制集状态的字符串表示
Uptime	该成员的运行时间
optime	应用在该成员上最后一个操作的时间，格式为一个时间戳和一个整数
optimeDate	最后一个被应用操作的日期
lastHeartbeat	最后一次发送心跳的日期
lastHeartbeatRecv	最后一次收到心跳的日期
pingMs	运行 rs.status() 命令的服务器和每个远程成员之间的响应(ping)时间
syncingTo	使用哪个复制集成员同步数据

之前的样例是在主服务器上运行 rs.status() 命令。该命令返回的信息显示，主服务器的 myState 值为 1；换句话说，“该成员是主服务器”。

2. 使用 rs.stepDown()强制进行新的选举

可以使用 `rs.stepDown()` 命令强制主服务器退出 60 秒；该命令将强制选出新的主服务器。该命令在下面的情况下非常有用：

- 需要使托管主服务器实例的服务器离线，无论是调查服务器，还是进行硬件升级或维护。
- 需要为数据结构运行诊断进程。
- 需要模拟主服务器崩溃产生的影响，并强制集群执行故障切换，从而测试应用如何对这样的事件做出响应。

如果在 `testset` 复制集中运行 `rs.stepDown()` 命令，那么将会输出下面的结果：

```
> rs.stepDown()
> rs.status()
{
  "set" : "testset",
  "date" : ISODate("2013-06-04T11:19:01Z"),
  "myState" : 2,
  "members" : [
    {
      "_id" : 0,
      "name" : "[hostname]:27021",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 5428,
      "optime" : Timestamp(1370340105, 1),
      "optimeDate" : ISODate("2013-06-04T10:01:45Z"),
      "self" : true
    },
    {
      "_id" : 1,
      "name" : "[hostname]:27022",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 4636,
      "optime" : Timestamp(1370340105, 1),
      "optimeDate" : ISODate("2013-06-04T10:01:45Z"),
      "lastHeartbeat" : ISODate("2013-06-04T11:19:00Z"),
      "lastHeartbeatRecv" : ISODate("2013-06-04T11:19:00Z"),
      "pingMs" : 0,
      "syncingTo" : "[hostname]:27021"
    },
    {
      "_id" : 2,
      "name" : "[hostname]:27023",
      "health" : 1,
      "state" : 2,
      "stateStr" : "SECONDARY",
      "uptime" : 4636,
      "optime" : Timestamp(1370340105, 1),
      "optimeDate" : ISODate("2013-06-04T10:01:45Z"),
      "lastHeartbeat" : ISODate("2013-06-04T11:19:01Z"),
      "lastHeartbeatRecv" : ISODate("2013-06-04T11:19:00Z"),
      "pingMs" : 0,
      "lastHeartbeatMessage" : "db exception in producer: 10278 dbclient
error communicating with server: [hostname]:27021",
      "syncingTo" : "[hostname]:27021"
    }
  ]
}
```

```

    }
  ],
  "ok" : 1
}

```

在本例中，我们在主服务器上运行了 `rs.stepDown()` 命令。`rs.status()` 命令的输出显示：复制集中的所有成员现在都是辅助服务器。如果接着运行 `rs.status()`，你应该看到另一个成员已经成为主服务器(假设有可用的服务器)。

3. 判断某个成员是否为主服务器

命令 `db.isMaster()` 并不只可以用在复制集中。而且该命令极其有用，因为它允许应用测试当前连接是否属于主服务器：

```

> db.isMaster()
{
  "setName" : "testset",
  "ismaster" : true,
  "secondary" : false,
  "hosts" : [
    "[hostname]:27022",
    "[hostname]:27021"
  ],
  "primary" : "[hostname]:27022",
  "me" : "[hostname]:27022",
  "maxBsonObjectSize" : 16777216,
  "maxMessageSizeBytes" : 48000000,
  "localTime" : ISODate("2013-06-04T11:22:28.771Z"),
  "ok" : 1
}

```

如果此时在 `testset` 复制集中运行 `isMaster()`，将显示出当前连接的服务器是否是主服务器 (`"ismaster"=false`)。如果运行命令所在的服务器是复制集的成员，该命令还会返回复制集中已知服务器的信息，包括每个服务器在复制集中的角色。

11.4.7 为复制集成员设置选项

复制集功能包含了许多可用于控制复制集成员的选项。当运行 `rs.initiate(replSetcfg)` 或 `rs.add(membercfg)` 选项时，需要提供描述复制集成员的特定配置结构：

```

{
  _id : <setname>,
  members: [
    {
      _id : <ordinal>,
      host : <hostname[:port]>,
      [ priority: <priority>, ]
      [ arbiterOnly : true, ]
      [ votes : <n>, ]
      [ hidden: true, ]
      [ tags: { document }, ]
      [ slaveDelay: <seconds>, ]
      [ buildIndexes: true, ]
    }
  ], ...
}

```

```

settings: {
  [ chainingAllowed : <boolean>, ]
  [ getLastErrorModes: <modes>, ]
  [ getLastErrorDefaults: <lasterrdefaults>, ]
}
}

```

如该结构所示，对于 `rs.initialize()`，我们应该提供完整的配置结构。最高级的配置结构包括 3 级：`_id`、`members` 和 `settings`。`_id` 是复制集的名称，与创建复制集成员时使用 `--replSet` 命令行选项时提供的名称一样。数组 `members` 由一个描述每个成员的集合组成；这是添加单个服务器到集合中时，应该在 `rs.add()` 命令中提供的成员结构。最后，数组 `settings` 包含应用到整个复制集的选项。

1. 成员结构的组织

成员结构中包含所有必需的用于配置复制集中每个成员实例的所有属性，表 11-4 列出了所有这些属性。

表 11-4 配置成员服务器属性

选 项	描 述
<code>members.\$._id</code> (必选)	整数：该元素指定成员结构在成员数组中的序数位置。该元素的值为大于等于 0 的整数。通过该值，可以帮助定位特定的成员结构，从而可以执行添加、删除和覆写操作
<code>members.\$.host</code> (必选)	字符串：该元素以 <code>host:port</code> 形式指定服务器的名称；注意主机部分不可以是 <code>localhost</code> 或 <code>127.0.0.1</code>
<code>members.\$.priority</code> (可选)	浮点数：该元素代表在选举新的主服务器时，分配给该服务器的权重。如果主服务器不可用，那么复制集将根据该值从辅助服务器中选举出新的主服务器。任何含有非 0 值的辅助服务器将被认为是活跃的，也是可用的服务器。因此，将该值设置为 0 将强制辅助服务器变成被动模式。如果多个辅助服务器的优先级相同，那就需要进行投票，还可以调用仲裁服务器(如果已配置的话)解决任何死锁。该元素的默认值是 1.0
<code>members.\$.arbiterOnly</code> (可选)	布尔值：该成员在选举新的主服务器时，角色为仲裁服务器。不涉及复制集的任何其他功能，并且也不需要 <code>--replSet</code> 命令行选项。系统中运行的任何 <code>mongod</code> 进程都可以执行该任务。该元素的默认值为假
<code>members.\$.votes</code> (可选)	整数：该元素指定当前实例在选举主服务器时可投的票数；该元素的默认值为 1
<code>members.\$.hidden</code> (可选)	布尔值：该元素将从 <code>db.isMaster()</code> 的输出中隐藏该节点，从而阻止在该节点上发生读取操作，即使设置了辅助服务器读取偏好也不可以
<code>members.\$.tags</code> (可选)	文档：通过该选项可以设置服务器标签读取偏好中的标签
<code>members.\$.slaveDelay</code> (可选)	整数：通过该选项可以设置辅助服务器落后于主服务器的延迟时间
<code>members.\$.buildIndexes</code> (可选)	布尔值：该选项用于禁止索引构建。永远也不应该在可能成为主服务器的节点上设置该选项。该功能对于备份节点，或者在索引不重要并且希望节省空间的情况下非常有用

2. 浏览 Settings 结构中可用的选项

表 11-5 列出了 `Settings` 结构中可用的复制集属性。这些设置将被应用于整个复制集，通过

这些属性可以设置复制集成员间如何通信。

表 11-5 Settings 结构中的服务器间通信属性

选 项	描 述
settings.chainingAllowed(可选)	布尔值：指定该成员是否允许从其他辅助服务器复制数据。默认为真
settings.getLastErrorModes(可选)	模式：用于自定义写顾虑设置(custom write concerns)，本章稍后将进行讲解
Settings.getLastErrorDefaults(可选)	默认值，用于自定义写顾虑设置

11.4.8 从应用连接到复制集

从 PHP 中连接到复制集类似于连接到单个 MongoDB 实例。唯一的区别在于可以提供单个复制集实例地址或复制集成员列表；连接库将会分析出哪个是主服务器，并将请求重定向至该服务器，即使主服务器不在所提供的成员列表中。因此，通常最好在连接字符串地址中指定多个成员；通过这种方式可避免在只连接到一个成员时，该成员离线的问题。下面的样例演示了如何从 PHP 应用连接到复制集：

```
<?php
$m = new MongoClient("mongodb://localhost:27021,
    localhost:27022", array("replicaSet" => "testSet"));
...
?>
```

1. 在应用中设置读取偏好

MongoDB 的读取偏好是它选择从哪个复制集成员读取数据的方式。通过为驱动指定一个读取偏好，它将知道应该在复制集的哪个成员上执行查询。目前，有 5 种模式可用于设置读取偏好，如表 11-6 所示。

表 11-6 读取偏好选项

选 项	描 述
primary	只从主服务器读取数据。如果用户显式指定使用标签读取偏好，那么该读取偏好将被阻塞。这也是默认的读取偏好
primaryPreferred	读取将被重定向至主服务器；如果没有可用的主服务器，那么读取将被重定向至某个辅助服务器
secondary	读取将被重定向至辅助服务器节点。如果没有可用的辅助服务器，该选项将会生成异常
secondaryPreferred	读取将被重定向至辅助服务器；如果没有可用的辅助服务器，那么读取将被重定向至主服务器。该选项对应旧的“slaveOK”方法
nearest	从最近的节点读取数据，不论它是主服务器还是辅助服务器。该选项通过网络延迟决定使用哪个节点

注意:

如果设置了读取偏好,就意味着你可能从辅助服务器读取数据。必须注意,你得到的数据可能不是最新的数据;特定的操作可能无法从主服务器复制到辅助服务器。

在 PHP 的连接对象中可以使用 `setReadPreference()`命令设置读取偏好,如下所示:

```
<?php
$m = new MongoClient("mongodb://localhost:27021,
                    localhost:27022", array("replicaSet" => "testSet"));
$m->setReadPreference(MongoClient::RP_SECONDARY_PREFERRED, array());
...
?>
```

从现在开始,该连接执行的任何查询都将在集群的辅助服务器节点上运行。还可以通过在 URI 中添加读取偏好标签的方式设置它。下面是一个使用 `nearest` 读取偏好设置的样例:

```
mongodb://localhost:27021,localhost:27022?readPreference=nearest
```

2. 在应用中设置写顾虑

写顾虑(Write Concern)的概念类似于读取偏好。通过该选项可以指定在写操作被确认完成之前,数据必须被安全提交到多少个节点。使用 MongoDB 的获取最后一个错误(Get Last Error, GLE)机制检查连接中发生的最后一个错误,即可对该选项进行测试。它有几种模式可供选择,通过它们可以设置在执行写操作时如何持久化数据。表 11-7 列出了所有选项。

表 11-7 MongoDB 写顾虑级别

选 项	描 述
w=0 或不确认	认单向写操作。写操作执行后,不需要确认提交状态
w=1 或确认	写入操作必须得到主服务器的确认。这是默认为
w=n 或复制集确认	主服务器必须确认该写操作,并且 n-1 个成员必须从主服务器复制该写入操作。该选项更强大,但会引起延迟(如果复制集的成员有复制延迟的话,或者由于发生系统中断等类似的情况,导致写操作提交时没有足够的成员处于运行状态)
w=majority	写操作必须被主服务器确认,同时也需要集合中的大多数成员都确认该操作。而 w=n 可能会因为系统中断或复制延迟引起问题
j=true 日志	可以与 w=写顾虑(Write Concern)一起共同指定写入操作必须被写入到日志中,只有这样才能算是确认完成

为了在插入时使用写顾虑,只需要在指定的 `insert()`函数中添加 w 选项即可,如下所示:

```
$col->insert($document, array("w" => 1));
```

该命令将尝试在集合中插入一个文档,并使用 w=1 对写入进行确认。

3. 在读取偏好和写顾虑中使用标签

除了刚才讨论的读取偏好和写顾虑,还有另一种可用的方式——标签。通过这种机制可以在复制集的成员中设置自定义标签,然后在读取偏好和写顾虑中使用这些标签,以一种更细粒度的方式控制操作的行为。下面开始在复制集中添加标签。可以将标签添加到复制集配置文件

的标签部分。首先在复制集配置文件中添加标签站点 a 和 b:

```
conf=rs.conf()
conf.members[0].tags = {site : "a"}
conf.members[1].tags = {site : "b"}
conf.members[2].tags = {site : "a"}
rs.reconfigure(conf)
```

现在检查新的配置,你会看到我们已经设置了两个站点;它们被定义在每个配置文档的 `tags` 部分。

```
rs.conf()
{
  "_id" : "testset",
  "version" : 8,
  "members" : [
    {
      "_id" : 0,
      "host" : "Pixl.local:27021",
      "tags" : {
        "site" : "a"
      }
    },
    {
      "_id" : 1,
      "host" : "Pixl.local:27022",
      "tags" : {
        "site" : "b"
      }
    },
    {
      "_id" : 2,
      "host" : "Pixl.local:27023",
      "priority" : 0,
      "hidden" : true,
      "tags" : {
        "site" : "a"
      }
    },
    {
      "_id" : 3,
      "host" : "Pixl.local:27024",
      "arbiterOnly" : true
    }
  ]
}
```

现在让我们开始使用新的标签!我们将读取偏好设置为复制集中站点 a 的最近成员。

```
$m->setReadPreference(MongoClient::RP_NEAREST, array( array('site' => 'a'),));
```

学习了如何设置读取偏好之后,下面开始学习设置写顾虑。写顾虑稍微有点复杂,因为我们首先需要修改复制集配置文件,添加额外的 `getLastErrorModes`。此时,我们需要创建一个写顾虑,表示指定的写入操作必须被提交到两个不同站点中足够多的节点。这意味着写入最起码必须被提交到站点 a 和 b。为此,我们需要将 `getLastErrorModes` 变量设置为一个文档,该文档中包含了我们新的写顾虑名称和规则(写入操作需要被提交到两个不同的站点标签),如下所示:

```
conf = rs.conf()
conf.settings.getLastErrorModes = { bothSites : { "site": 2 } }
rs.reconfig(conf)
```

现在需要查询我们的文档，并指定新的写顾虑：

```
$col->insert($document, array("w" => "bothSites"));
```

该命令同样简单。现在我们可以保证写入操作将被同时提交到两个站点！接下来，我们希望将该写顾虑设置为集群任何写入操作的默认写顾虑：

```
conf = rs.conf()
conf.settings.getLastErrorDefaults = { bothSites : 1 }
rs.reconfig(conf)
```

现在，任何写入操作都将使用默认的写顾虑 `bothSites`。所以只需要执行正常的插入操作即可！

11.5 小结

MongoDB 提供了丰富的工具，用于实现冗余和强大的复制拓扑技术。本章讲解了其中的许多工具，包括使用它们的一些原因和动机。还讲解了如何创建许多不同的复制集拓扑。另外，还讲解了如何使用命令行工具和内置的 Web 接口检测复制系统的状态。最后，讲解了如何创建和配置读取偏好和写顾虑(Write Concern)，用于保证从正确的地方读取数据，并将数据写入正确的地方。

请花些必要的时间对本章描述的每个选项和函数进行研究，在将复制集应用到生产环境之前，确保构建出的复制集符合特定的需求。使用 MongoDB 在单个机器中创建设置环境是非常容易的；正如本章所做的一样。因此，强烈鼓励你实验每种方法，确保自己完全明白每种方式的优点和限制，包括它将如何处理特定的数据和应用。

第 12 章

分 片

无论是在构建下一个 Facebook，还是在构建一个简单的数据库应用，一旦成功之后，你可能就需要扩大应用的规模。如果不希望继续更换硬件(或者已经开始接近所能使用硬件的极限)，那就需要使用一种能够为系统增加容量的技术。分片这种技术可以将数据分散到多台机器，但对于应用来说，仍然如同在使用单个数据库一样。

分片非常适用于基于云的计算平台，由 MongoDB 实现的分片非常擅长以动态和负载敏感的方式自动调整规模，可以在需要的时候增大容量，同时也可以减小容量。

本章将讲解 MongoDB 中分片的实现，并讲解其中提供的一些高级功能，例如标签分片和哈希分片键。

12.1 了解分片的需求

当万维网刚开始发展时，站点、用户和可用在线信息的数量是非常少的。Web 由数千个站点组成，并且只有数万或数十万用户，他们主要关注于学术和研究社区。在早期的日子里，数据通常非常简单：由超链接将手工维护的 HTML 文档连接在一起。该协议的初始设计目标是：使 Web 能够提供一种为网络中不同服务器上存储的文档创建可导航引用的方式。

即使是 Yahoo! 这样的公司，相对于今天它所能提供的信息来说，当时在 Web 中也只提供了很少信息。该公司最早创建的产品被称为 Yahoo 目录，只不过比手工编辑流行网站连接的网络稍微好一点。这些链接由一些热情的人们维护，他们被称为冲浪者。Yahoo 目录中的每个页面都是一个存储在文件系统目录树中的简单 HTML 文档，并且人们使用简单文本编辑器对它们进行维护。

但随着网络的爆炸性增长，网站和访问者的数量也随之呈指数级增长。数量庞大的可用资源，迫使早期的 Web 先驱从使用简单的文档开始转向更复杂的动态页面，并将信息保存在不同的数据存储结构中。

搜索引擎开始在 Web 中爬网，并将今天数以亿计的链接和网页保存在数据库中。

这些发展推动了由内容管理系统管理和维护的数据集的发展，为了便于访问它们，数据主要被存储在数据库中。

与此同时，新的服务也在不断发展，它们所存储的不仅是文档和链接。音频、视频、事件和所有其他数据种类都开始进入这些大的数据存储中。这个过程通常被称作“数据的工业化”，与 19 世纪主要发生于制造业的工业革命有着许多相似之处。

最终，所有在 Web 中获得成功的公司都面临着一个问题：如何访问这些存储在庞大数据库中的数据。它们发现单个数据库每秒只能处理这么多的查询，而网络和磁盘驱动器每秒也只能

从服务器获得或发送这么多的数据量。提供基于 Web 的服务的公司很快发现，它们的需求已经超出单个服务器、网络或驱动阵列的最大性能。在这种情况下，它们被迫将庞大的数据分割并分散开。常见的解决方案是将这些庞大的数据块分割成小块数据，以便于通过更可靠和快速的方式进行管理。与此同时，这些公司需要保持对整个数据集(保存在巨大的机器集群中)进行操作的能力。

之前在第 11 章详细讲解过的复制，可能是克服其中一些规模问题的有效工具，通过复制可以在多台服务器上创建多个一致的副本。通过这种方式可以将服务器负载分散到多个机器(在正确的环境中)。

不过，你很快还会遇到另一个问题，组成数据集的每个表或集合的大小可能会迅速超过单个数据库系统可以高效管理它们的最大容量。例如，Facebook 称每天需要处理超过 3 亿张的照片！而该网站已经运行了几乎 10 年。

一年时间过去就是 1095 亿张图片，对于单个表，存储这么多数据显然是不可行的。所以 Facebook 如同许多之前的公司一样，考虑将记录集分散到大量的数据库服务器中。Facebook 采用的解决方案是分片，这就是在现实世界中使用分片的最好证明。

12.2 对数据进行水平和垂直分区

数据分区是一种将数据分割到多个独立数据存储中的机制。这些数据存储既可以在本地(驻留在同一系统中)，也可以在远端(分散在不同的系统中)。使用本地分区的动机是减少每个索引的大小和更新记录所需的 I/O 访问量。使用远程分区的动机是增大访问数据的带宽，通过使用更多 RAM 存储数据、避免磁盘访问或提供更多可用网络接口和磁盘 I/O 通道的方式来实现。

12.2.1 对数据进行垂直分区

在数据库的传统视图中，数据以行和列的方式存储。垂直分区的实现方式为：拆分列边界上的记录，并将各个部分存储在不同的表或集合中。可以认为，关系数据库通过按照一比一的关系使用连接表构成的是本地垂直数据分区。

不过，MongoDB 并未使用这种形式的分区，因为它的记录结构(文档)并不适合整洁的行列模型。因此，几乎很难根据列边界将行清楚地分割开。MongoDB 还采用了嵌入文档，它并未直接提供在服务器上连接关联集合的能力(这些可以在应用中完成)。

12.2.2 对数据进行水平分区

使用 MongoDB 时，水平分区是唯一可采用的方式，而分片就是各种流行水平分区的通用术语。通过分片，可以将集合分割到多个服务器，从而改善包含大量文档的集合的性能。

当需要将用户记录的集合分割到一组服务器时，使用分片的一个简单样例是：将所有姓以 A-G 开头的人的所有记录保存在一台服务器上，将以 H-M 开头的人的所有记录保存在另一台服务器上，依此类推。分割数据的规则被称分片键。

简单来说，通过分片可以将分片云当成单个完整的集合使用，应用不需要关注数据是否被分散到多个机器。传统的分片实现要求应用积极参与决定特定的文档存储在哪台服务器上，这样才可以正确地发出请求。通常，需要将一个库绑定到应用，该库将负责存储和查询分片数据

集中的数据。

MongoDB 使用一种唯一的方法用于分片，由 MongoS 路径进程管理数据的分割，并将请求路由到必需的分片服务器。如果查询需要访问多个分片中的数据，那么 MongoS 将会管理从多个分片获取数据并将数据合并成单个游标的过程。

相较于其他特性，正是这个特性帮助 MongoDB 获得了云数据库或基于 Web 的数据库的名号。

12.3 分析一个简单的分片场景

假设现在希望为一个虚拟的 Gaelic 社交网络提供简单的分片解决方案。图 12-1 显示了对该应用进行分片的简化表示。

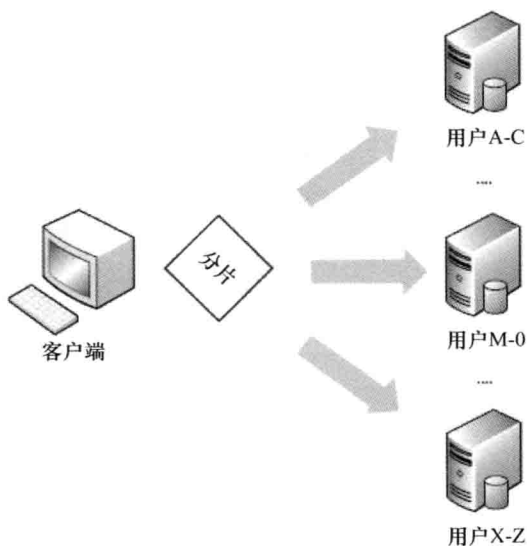


图 12-1 用户集合的简单分片

在应用中使用这种简化的分片会产生许多问题。我们先讨论一些明显的问题。

首先，如果 Gaelic 网络的目标是成为世界著名的 Irish 和 Scottish 社区，那么数据库中将会为苏格兰人存储大量以 Mac 和 Mc(MacDonald、McDougal 等)开头的名字，为爱尔兰人存储大量以 O'(O'Reilly、O'Conner 等)开头的名字。因此，通过把姓的第一个字母用作简单的分片键，会将过量的用户记录保存在支持字母范围“M-O”的分片中。类似地，支持字母范围“X-Z”的分片基本不需要存储太多数据。

分片系统的一个重要特性是，它必须保证数据被平均分散到可用的分片服务器的集合中。这将阻止对集群整体性能产生影响的热点的产生。将它称为需求 1：具有平均将数据分散到所有分片中的能力。

另一件要记住的事情是：将数据集分散到多个服务器中时，将会增大由硬件故障导致的数据集的脆弱性。也就是说，随着服务器的增加，所有可用数据受单个服务器故障影响的可能性也更大。可靠分片系统的一个重要特性是，如同磁盘驱动器中常用的 RAID 系统一样，也将每片数据存储多个系统中，并且可以容忍单个分片系统不可用的情况。让我们称它为需求 2：以

容错的方式存储分片数据的能力。

最后，希望可以保证从分片集中添加或删除服务器，而不影响数据的备份和恢复，以便于将数据重新分散到更小或更大的分片集中。更进一步，需要能够在不引起集群宕机的情况下完成服务器的添加和删除。让我们称它为需求 3：在系统运行时添加或删除分片的能力。

接下来将详细讲解如何满足这些需求。

12.4 使用 MongoDB 实现分片

MongoDB 使用代理机制实现分片(如图 12-2 所示)；其中的 mongos 守护进程将作为多个基于 mongod 的分片服务器的控制器。当应用连接到 mongos 时，将把这些分片服务器当作单个 MongoDB 数据服务器；此后，应用将把它的所有命令(例如更新、查询和删除)都发送到 mongos 进程。

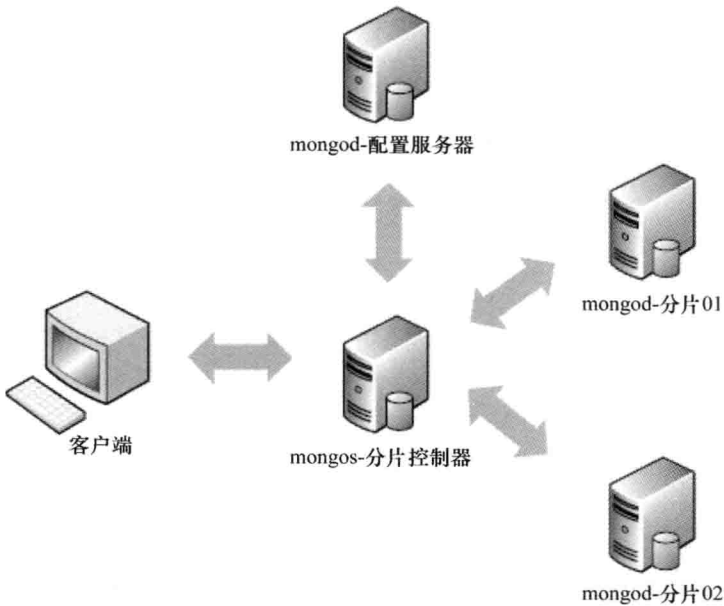


图 12-2 不使用冗余时的简单分片设置

进程 mongos 负责管理应用发送到 MongoDB 服务器的所有命令，并且该守护进程将重新发送跨多个分片的查询到多个服务器，再将结果聚集在一起。

MongoDB 在集合级别实现分片，而不是数据库级别。在许多系统中，只有一个或两个集合可以增长到需要使用分片的地步。因此应该理智地使用分片；如果不需要的话，就不要为较小的集合增加管理分布数据的开销。

现在返回到虚拟的 Gaelic 社交网络样例。在该应用中，用户集合包含用户和他们资料的细节信息。该集合可能会增长到需要分片的地步。不过，其他集合，例如事件、国家和州，就不可能变得这么大，因此也就不需要使用分片。

分片系统将使用分片键将数据映射到块，块是文档键的逻辑连续范围(关于块的更多信息请参见第 5 章)。每个块标志着分片键值特定连续范围内的许多文档；这些值使 mongos 控制器可

以快速找到包含它所需文档的块。然后 MongoDB 分片系统将把块存储在可用的分片系统中；配置服务器将记录每个块存储的分片服务器位置。这是分片实现的一个重要特性，因为通过它可以从中添加和删除分片，而不需要备份和恢复数据。

当在集群中添加新的分片时，该系统将会把许多块迁移到新的服务器集合中，从而平均地分散数据。类似地，从集群中删除分片时，分片控制器将会从即将离线的分片中抽取所有的块，并重新将它们分散到剩下的分片服务器中。

MongoDB 的分片设置还需要存储分片服务器的配置，以及集群中每个分片服务器的信息。为了支持该功能，需要使用一台称为配置服务器的 MongoDB 服务器；该服务器实例是一个以特殊角色运行的 mongod 服务器。正如之前解释的，配置服务器还可以用作目录，通过它可以找到每个块的位置。在集群中可以具有 1 台(开发)或 3 台(生产)配置服务器。推荐在生产环境中使用 3 台配置服务器，因为配置服务器的崩溃将意味着无法决定分片数据存储在哪一个分片中。

乍一看，似乎实现分片解决方案需要使用大量的服务器！不过，可以将多个不同服务的实例添加到同一服务器中，可以在一台相对较小的物理服务器中创建分片(类似于第 11 章讲解的复制)，但需要实行严格的资源管理，以避免 MongoDB 进程之间相互竞争类似 RAM 这样的资源。图 12-3 显示了一个完全冗余的分片系统，它将为分片存储和配置服务器使用复制集，并且使用一组 mongos 管理集群。它还显示了如何将这组服务以密集的方式运行在 3 台物理服务器中。

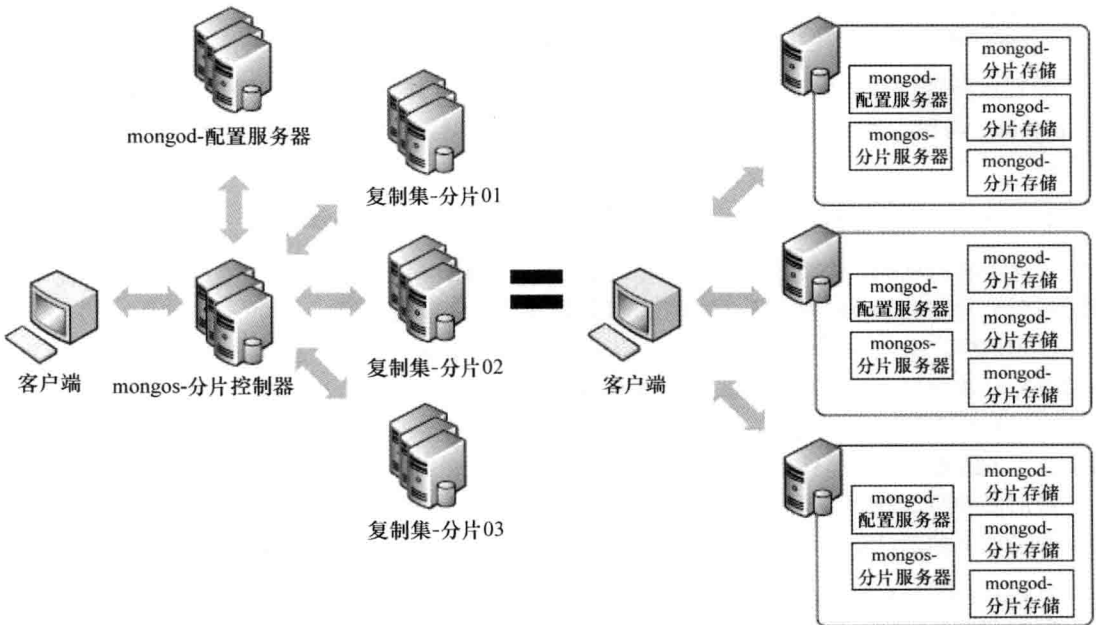


图 12-3 冗余分片配置

设置分片存储实例的时候一定要仔细，使它们正确地分布在物理服务器之间，只有这样系统才能容忍集群中的一台或多台服务器出现故障。这种方式与 RAID 磁盘控制器采取的方式一致，该控制器将把数据分散到条带中的多台服务器，从而使 RAID 配置可以恢复出现故障的驱动器中的数据。

12.4.1 创建分片设置

为了有效地使用分片，了解它的工作方式很重要。下面的样例将演示如何在单个机器中创建测试配置。该例的配置将与图 12-2 所示的简单分片系统一样，除了两点区别：为了简单，该例只使用了两台分片服务器，并且这些分片服务器仅仅只是 `mongod` 服务器而不是完整的复制集。最后，本节将讲解如何创建一个简单的分片集合，并使用一个简单的 PHP 测试程序演示如何使用该集合。

该例将会用到表 12-1 中列出的服务。

表 12-1 测试配置中使用的服务器实例

服 务	守护进程	端 口	数据库路径
分片控制器	mongos	27021	N/A
配置服务器	mongod	27022	/db/config/data
分片 0	mongod	27023	/db/shard1/data
分片 1	mongod	27024	/db/shard2/data

首先创建配置服务器。在新的终端窗口中输入下面的代码：

```
$ mkdir -p /db/config/data
$ mongod --port 27022 --dbpath /db/config/data --configsvr
```

配置服务器运行之后，一定要保证该终端窗口处于打开状态，也可以在命令中添加 `--fork` 和 `--logpath` 选项。接下来，创建分片控制器(mongos)。打开一个新的终端窗口并输入下面的命令：

```
$ mongos --configdb localhost:27022 --port 27021 --chunkSize 1
```

该命令将启动分片控制器，并监听端口 27021。如果此时查看配置服务器所在的终端窗口，你会看到分片服务器已经连接到它的配置服务器并在其中注册自己。

在本例中，块大小被设置为它的最小值：1MB。在实际的系统中这并不是一个合理值，因为这意味着块存储小于文档的最大大小(16MB)。不过，这只是一个演示样例，使用一个小的块大小可以让你创建出大量的块，从而在不加载许多数据的情况下练习分片的创建。默认块大小被设置为 64MB，除非另外指定。

最后启动两台分片服务器。打开两个全新的终端窗口，分别用于启动两台服务器。在一个窗口中输入下面的命令以启动第一台服务器：

```
$ mkdir -p /db/shard0/data
$ mongod --port 27023 --dbpath /db/shard0/data
```

在第二个窗口中输入下面的命令以启动第二台服务器：

```
$ mkdir -p /db/shard1/data
$ mongod --port 27024 --dbpath /db/shard1/data
```

现在服务器已经成功运行。接下来，需要告诉分片系统分片服务器的位置。使用服务器的主机名连接到分片控制器(mongos)。可以使用 `localhost`，但这将会把集群限制在该机器中。应该使用运行该例的主机的名称替换 `<hostname>`。有一点非常重要，尽管 `mongos` 不是一个完整的 MongoDB

实例，但它对应用来说却会显示为一个完整的实例。因此，可以使用 `mongo` 命令 `shell` 连接到分片控制器并添加两台分片服务器，如下所示：

```
$ mongo <hostname>:27021
> sh.addShard("<hostname>:27023")
{ "shardAdded" : "shard0000", "ok" : 1 }
> sh.addShard( "<hostname>:27024")
{ "shardAdded" : "shard0001", "ok" : 1 }
```

两台分片服务器现在已经被激活；接下来需要使用 `listshards` 命令检查分片服务器的状态：

```
> db.printShardingStatus();
--- Sharding Status ---
sharding version: {
  "_id" : 1,
  "version" : 3,
  "minCompatibleVersion" : 3,
  "currentVersion" : 4,
  "clusterId" : ObjectId("5240282df4ee9323185c70b2")
}
shards:
  { "_id" : "shard0000", "host" : "<hostname>:27023" }
  { "_id" : "shard0001", "host" : "<hostname>:27024" }
databases:
  { "_id" : "admin", "partitioned" : false, "primary" : "config" }
  { "_id" : "test", "partitioned" : false, "primary" : "shard0000" }
```

现在分片环境已经可以正常运行，但没有分片数据；接下来创建一个名为 `testdb` 的数据库，然后在该数据库中激活一个名为 `testcollection` 的集合。对该集合进行分片，赋予它一个名为 `testkey` 的参数，用作分片函数：

```
> sh.enableSharding("testdb")
{ "ok" : 1 }
> sh.shardCollection("testdb.testcollection", {testkey : 1})
{ "collectionsharded" : "testdb.testcollection", "ok" : 1 }
```

到目前为止，我们已经创建了一个拥有两台分片存储服务器的分片集群，还创建了一个含有分片集合的数据库。不含数据的服务器是没有用的，所以接下来需要在集合中添加一些数据，演示分片数据是如何分布的。

使用一个小的 PHP 程序在分片集合中加载一些数据。这些数据将包含一个名为 `testkey` 的字段(该字段包含一些随机数字)和一个包含一块固定文本的字段(使用该字段用于保证创建的分片块数合理)；该集合将作为虚拟网站 `TextAndARandomNumber.com` 的主数据表。下面的代码将创建一个在分片服务器中插入数据的 PHP 程序：

```
<?php
//创建连接到 mongos 守护进程的数据库连接
$mongo = new MongoClient("localhost:27021");
//选择 test 数据库
$db = $mongo->selectDB('testdb');
//选择 TestIndex 集合
$collection = $db->testcollection;

for($i=0; $i < 100000 ; $i++){
    $data=array();
```



```
$data['testkey'] = rand(1,100000);
$data['testtext'] = "Because of the nature of MongoDB, many of the more "
    . "traditional functions that a DB Administrator "
    . "would perform are not required. Creating new databases, "
    . "collections and new fields on the server are no longer
necessary, "
    . "asMongoDB will create these elements on-the-fly as
you access them."
    . "Therefore, for the vast majority of cases managing
databases and "
    . "schemas is not required.";
$collection->insert($data);
}
```

该程序将会连接到分片控制器(mongos)并插入 10 万条包含了随机 testkey 和一些 testtext 的记录。正如之前提到的, 该样例文本将使文档占据一定量的块, 从而保证分片机制合理运行。使用下面的命令运行测试程序:

\$phptestshard.php

一旦程序运行结束, 就可以在命令行 shell 中连接 mongos 实例, 并验证已经存储的数据:

```
$mongo localhost:27021
> use testdb
> db.testcollection.count()
100000
```

此时, 服务器中已经存储了 10 万条记录。现在需要连接到每个分片服务器并检查每个分片服务器的 testdb.testcollection 中存储的记录数目。下面的代码将连接到第一台分片服务器, 检查 testcollection 集合中存储的记录数目:

```
$mongo localhost:27023
> use testdb
> db.testcollection.count()
48875
```

接下来的代码将连接到第二台分片服务器, 检查 testcollection 集合中存储的记录数目:

```
$mongo localhost:27024
> use testdb
> db.testcollection.count()
51125
```

注意:

在每台分片服务器中你可能会看到不同的文档数目, 这取决于检查每台分片服务器的时间。mongos 实例开始会在一个分片中初始化所有的块, 但随着时间的推移, 将对数据集中的数据进行调整, 通过移动块的方式将数据平均地分布到所有分片服务器中。因此, 指定分片服务器中的记录数目可能会不断改变。这满足了“需求 1: 具有将数据分散到所有分片服务器的能力”。

12.4.2 在集群中添加新的分片

假设 TextAndARandomNumber.com 公司的业务开始了飞速发展。为了满足这个需求, 需要在集群中添加新的分片服务器, 从而降低每台服务器的负载。

添加新的分片非常简单；所有要做的就是重复之前描述的步骤。首先创建新的分片存储服务器，并使它监听端口 27025(避免与现有的服务器发生冲突)：

```
$ sudo mkdir -p /db/shard2/data
$ sudo mongod --port 27025 --dbpath /db/shard2/data
```

接下来，需要在集群中添加新的分片服务器。连接到分片控制器，然后使用 `admin` 命令 `addshard`：

```
$ mongo localhost:27021
> sh.addShard("localhost:27025")
{ "shardAdded" : "shard0002", "ok" : 1 }
```

此时，可以运行 `listshards` 命令以验证分片是否已经被添加到集群中。下面的输出显示新的分片服务器(shard2)已经被添加到 `shards` 数组中：

```
> db.printShardingStatus();
--- Sharding Status ---
  sharding version: {
    "_id" : 1,
    "version" : 3,
    "minCompatibleVersion" : 3,
    "currentVersion" : 4,
    "clusterId" : ObjectId("5240282df4ee9323185c70b2")
  }
  shards:
    { "_id" : "shard0000", "host" : "<hostname>:27023" }
    { "_id" : "shard0001", "host" : "<hostname>:27024" }
    { "_id" : "shard0002", "host" : "<hostname>:27025" }
  databases:
    { "_id" : "admin", "partitioned" : false, "primary" : "config" }
    { "_id" : "test", "partitioned" : false, "primary" : "shard0000" }
```

如果连接到新的存储服务器(监听端口 27025)并查看 `testcollection` 集合，将会显示出一些有意思的信息：

```
$ mongo localhost:27025
> use testdb
switched to dbtestdb
> show collections
system.indexes
testcollection
> db.testcollection.count()
4657
> db.testcollection.count()
4758
> db.testcollection.count()
6268
```

上面的输出显示：分片服务器 `shard2` 中 `testcollection` 集合的文档数目正在缓慢增长。该输出同时也证明分片系统正在将数据重新平均分布到扩展后的集群中。随着时间的发展，分片系统将从 `shard0` 和 `shard1` 存储服务器中迁移出一些块，从而将数据平均分布在组成集群内的三台服务器中。该过程将自动发生，即使 `testcollection` 集合中没有新的数据插入，也会执行。在此情况下，`mongos` 分片控制器将移动一些块到新的服务器，然后将它们注册到配置服务器中。

这是选择块大小时需要考虑的因素之一。如果块大小非常大，那么分片集群中的数据分布可能就不太均匀；相反，块大小越小，数据的分布就越均匀。

12.4.3 从集群中移除分片服务器

如果 TextAndARandomNumber.com 公司的业务可以持续，那么这是非常好的事情，但可惜只是昙花一现。在几个星期的疯狂发展之后，网站的访问量开始回落，所以必须开始寻找减少开销的方式，换句话说，必须移除新的分片服务器！

在接下来的样例中，我们将会移除之前添加的分片服务器。连接到分片控制器，然后执行 `removeShard` 命令：

```
$ mongo localhost:27021
> use admin
switched to db admin
> db.runCommand({removeShard : "localhost:27025"})
{
  "msg" : "draining started successfully",
  "state" : "started",
  "shard" : "shard0002",
  "ok" : 1
}
```

命令 `removeShard` 返回了一条信息，表示移除已经启动，另外还表示 `mongos` 已经开始重新在目标分片服务器中移出块到集群中的其他分片服务器。该过程被称为清空分片服务器。

可以再次执行 `removeShard` 命令以检查清空进程的进度。响应的内容将告诉我们仍然有多少块和数据库需要从该分片服务器中清空：

```
> db.runCommand({removeShard : "localhost:27025"})
{
  "msg" : "draining ongoing",
  "state" : "ongoing",
  "remaining" : {
    "chunks" : NumberLong( 12 ),
    "dbs" : NumberLong( 0 )
  },
  "ok" : 1
}
```

最后，`removeShard` 进程将会终止，显示移除进程已经完成：

```
> db.runCommand({removeShard : "localhost:27025"})
{
  "msg" : "removeshard completed successfully",
  "state" : "completed",
  "shard" : "shard0002",
  "ok" : 1
}
```

为了验证 `removeShard` 命令是否成功，可以使用 `listshards` 确认目标分片服务器是否已经从集群中移除。例如，下面的输出将显示之前创建的服务器 `shard2` 已经不在 `shards` 数组中：

```
> db.runCommand({listshards:1})
{
```

```

"shards" : [
  {
    "_id" : "shard0000",
    "host" : "localhost:27023"
  },
  {
    "_id" : "shard0001",
    "host" : "localhost:27024"
  }
],
"ok" : 1
}

```

此时，可以终止 Shard2 mongod 进程并删除它的存储文件，因为它的的数据已经被迁移到其他服务器中。

注意：

从集群中添加和删除分片，而不使集群离线，是 MongoDB 支持高扩展性、高可用性和大容量存储的能力的重要组成部分。这将满足最后一个需求：“需求 3：在系统运行时添加和删除分片服务器的能力”。

12.4.4 确定连接的方式

个人应用可以连接到标准的非分片数据库(mongod)或分片控制器(mongos)。MongoDB 同时提供了这两种进程；在所有情况而不是部分情况下，数据和分片控制器的表现及行为都完全一致。不过，有时确定连接的目标系统可能是非常重要的。

MongoDB 提供了 isdbgrid 命令，用于询问数据系统，判断它是否是分片系统。下面的脚本显示了如何使用该命令，以及它的输出：

```

$mongo
> use testdb
> db.runCommand({ isdbgrid : 1});
{ "isdbgrid" : 1, "hostname" : "Pixl.local", "ok" : 1 }

```

响应中包含了 isdbgrid: 1 字段，它意味着目前连接到的系统中已经启用分片。如果响应中包含了 isdbgrid: 0 字段，那么将表示连接到的是一个非分片数据库。

12.4.5 列出分片服务器的状态

MongoDB 还包含了一个用于显示分片集群状态的简单命令：printShardingStatus()。

该命令将提供分片系统的许多内部信息。下面的脚本演示了如何使用 printShardingStatus() 命令，但只显示了一些输出以保证易于阅读：

```

$mongo localhost:27021
> sh.status();
--- Sharding Status ---
sharding version: {
  "_id" : 1,
  "version" : 3,
  "minCompatibleVersion" : 3,
  "currentVersion" : 4,

```

```

    "clusterId" : ObjectId("51c699a7dd9fc53b6cdc4718")
  }
  shards:
    { "_id" : "shard0000", "host" : "localhost:27023" }
    { "_id" : "shard0001", "host" : "localhost:27024" }
  databases:
    { "_id" : "admin", "partitioned" : false, "primary" : "config" }
    { "_id" : "test", "partitioned" : false, "primary" : "shard0000" }
    { "_id" : "testdb", "partitioned" : true, "primary" : "shard0000" }
      testdb.testcollection
        shard key: { "testkey" : 1 }
        chunks:
          shard0000 2
          shard0001 3
        { "testkey" : { "$minKey" : 1 } } --> { "testkey" : 0 } on : shard0000
Timestamp(4, 0)
        { "testkey" : 0 } --> { "testkey" : 14860 } on : shard0000 Timestamp(3, 1)
        { "testkey" : 14860 } -->{ "testkey" : 45477 } on : shard0001 Timestamp(4, 1)
        { "testkey" : 45477 } -->{ "testkey" : 76041 } on : shard0001 Timestamp(3, 4)
        { "testkey" : 76041 } -->{ "testkey" : { "$maxKey" : 1 } } on : shard0001
Timestamp(3, 5)

```

该输出列出了分片服务器、每个分片数据库/集合的配置和分片数据集中的块信息。因为这里为了模拟大的分片系统，所以使用的 `chunkSize` 值较小，报告中也列出了许多的块。从该列表中获得的一条重要信息是：每个块关联的分片键的范围。该输出了显示了特定的块被存储在哪台分片服务器中。可以使用该命令返回的输出作为分析分片服务器键和块工具的基本信息。例如，可以使用该数据判断数据集中是否存在某个数据块。

12.4.6 使用复制集实现分片

到目前为止演示的样例都依赖于实现分片的每个 `mongod` 实例。在第 11 章，我们已经学习了如何创建复制集，通过该 `mongod` 实例集群可以提供数据冗余和故障安全存储。

在分片集群中添加分片时，可以提供复制集的名称和复制集成员的地址，此时该分片的数据将在所有的复制集成员中实例化。`mongos` 将跟踪哪个实例是复制集的主服务器；还将保证所有的分片写入都被重定向至主服务器。

结合使用分片和复制集可以创建出高性能、高可靠性集群(通过容忍多台机器出现故障的方式)。通过它们还可以最大程度利用廉价的普通硬件的性能和可用性。

注意：

在分片中使用复制集作为存储机制，可以满足“需求 2：具有以容错的方式恢复分片数据的能力”。

12.5 均衡器

之前我们曾讨论过 MongoDB 可以自动将负载分散到集群的所有分片服务器中。你可能认为这是由某种 MongoDB 魔法完成的，事实上并不是这样。MongoS 中包含一个均衡器元素，它将会移动集群中数据的逻辑块，从而保证它们均匀分布在所有分片服务器中。均衡器将与分片进行通信，并告诉它们从一台分片服务器将数据迁移到另一台服务器。在下面的样例中，我们

将使用 `sh.status()` 显示出块的分布状态。测试数据的逻辑块有两块存储在 `shard0000` 上，另外三块存储在 `shard0001` 上。

```
{ "_id" : "testdb", "partitioned" : true, "primary" : "shard0000" }
  testdb.testcollection
    shard key: { "testkey" : 1 }
    chunks:
      shard0000 2
      shard0001 3
```

均衡器会自动执行这些工作，当然也可以通过命令控制它的行为。可以按需要停止和启动均衡器，以及设置它可以操作的时间窗口。为了停止均衡器，可以连接到 `MongoS` 并执行 `sh.stopBalancer()` 命令：

```
> sh.stopBalancer();
Waiting for active hosts...
Waiting for the balancer lock...
Waiting again for active hosts after balancer is off...
```

如你所见，均衡器现在已经关闭了；该命令已经将均衡器的状态设置为关闭，并等待和确认均衡器已经完成当前正在进行的迁移工作。启动均衡器的过程相同；运行 `sh.startBalancer()` 命令即可：

```
> sh.startBalancer();
```

这两个命令都需要一点时间才能完成和返回，因为它们都需要等待确认均衡器的状态，对于启动命令，需要确认均衡器已经启动并正在运行。如果发现问题或者希望手动确认均衡器的状态，可以执行下面的检查步骤。首先检查均衡器的标志，它是均衡器的启动/关闭标志，并且它被保存在 `config` 数据库中。

```
> use config
switched to dbconfig
db.settings.find({_id:"balancer"})
{ "_id" : "balancer", "stopped" : true }
```

如上所示，这里的 `_id` 值为 `balancer` 的文档被设置为 `stopped: true`，这意味着均衡器目前处于停止状态。不过，这并不意味着目前没有任何正在执行的迁移工作；为了确认是否存在迁移，需要检查“均衡器锁”。

均衡器锁的存在是为了保证在指定的时间内，只有一个均衡器可以执行均衡操作。通过下面的命令可以找到均衡器锁：

```
> use config
switched to dbconfig
> db.locks.find({_id:"balancer"});
{ "_id" : "balancer", "process" : "Pixl.local:40000:1372325678:16807", "state" : 0,
"ts" : ObjectId("51cc11c57ce3f0ee9684caff"), "when" : SODate("2013-06-27T10:19:49.820Z"),
"who" : "Pixl.local:40000:1372325678:16807:Balancer:282475249", "why" : "doing balance
round" }
```

该文档相比设置文档复杂得多。不过，最重要的是 `state` 条目，它表示是否已经上锁，0 意味着“空闲”或“未上锁”，而任何其他值都表示“正在使用”。另外还需要关注时间戳，它表示上锁的时间。将“空闲”锁与接下来的“占用”锁相比较，下面的输出表示均衡器正处于活跃状态：

```
>db.locks.find({_id:"balancer"});
{ "_id" : "balancer", "process" : "Pixl.local:40000:1372325678:16807", "state" : 1, "ts" :
ObjectId("51cc11cc7ce3f0ee9684cb00"), "when" : ISODate("2013-06-27T10:19:56.307Z"),
"who" : "Pixl.local:40000:1372325678:16807:Balancer:282475249", "why" : "doing balance
round" }
```

现在我们已经学习了如何启动和停止均衡器，以及如何检查在指定时间点均衡器正在执行的操作。接下来学习如何设置均衡器的活跃时间窗口。假设我们需要让均衡器在 8PM 到 6AM 之间运行，即让均衡器在集群不太活跃(假设)的夜间运行。那么需要更新之前获得的均衡器设置文档，因为它控制了均衡器的运行时间。执行以下命令即可：

```
> use config
switched to dbconfig
> db.settings.update({_id:"balancer"}, { $set : { activeWindow : { start : "20:00",
stop : "6:00" } }
}
```

这就完成了对均衡器的设置；均衡器文档现在包含一个 activeWindow 设置，它的开始时间为 8PM，终止时间为 6AM。现在你应该能够启动和终止均衡器，确认它最后的运行状态，以及为均衡器设置活跃时间窗口。

12.6 哈希片键

之前已经讨论过选取正确片键的重要性。如果选择了错误的片键，那么可能引起各种性能问题。例如，在 `_id` 这个一直增长的值上进行分片。每次插入的记录都将被发送到目前保存了最大 `_id` 值的分片服务器。每次插入的记录都包含“最大的” `_id` 值，所以数据将总是被插入到同一位置。这意味着集群中出现了一个“热点”分片，该服务器将接受所有的插入操作，并将文档从自身迁移到其他服务器中，这种方式并不高效。

为了解决这个问题，MongoDB 2.4 引入了一个新的特性——哈希片键！哈希片键将为目标字段上的每个值创建哈希值，并使用该哈希值执行分块和分片操作。通过这种方式，可以采用某个一直增长的值(例如 `_id` 字段)，因为 MongoDB 将为它生成哈希值，而哈希值是一个随机值。这个级别的随机性，通常可以保证将写操作均匀地分布到所有分片服务器中。不过，代价是读取也是随机的，如果希望在某个范围的文档上执行操作，这也会对性能造成影响。出于该原因，在特定的工作负载下，将这种方式与用户选择的片键相比，它就显得不那么高效。

注意：

由于哈希的实现方式，使用浮点数(decimal)进行分片时会有一些限制，这意味着 2.3、2.4、2.9 这样的值都会被生成相同的哈希值。

为了创建哈希分片，只需要运行 `shardCollection` 并创建“哈希”索引即可：

```
sh.shardCollection( " testdb.testhashed", { _id: "hashed" } )
```

这就是所有的命令！现在我们已经创建了一个哈希片键，为了以更加“随机”的方式分布数据，将为 `_id` 值计算出哈希值。现在，学习了哈希片键之后，你可能会问：“为什么不一直使用哈希片键呢？”

答案是：哈希片键的分片方式只是“这些”黑暗艺术中的一种。最优的片键应该能将写入

操作分布到许多服务器中，从而保证可以有效地执行写入操作。使用分组也是一个关键，通过它，写入操作将只在一台或有限数目的分片服务器中执行，并且还可以更高效地使用每台分片服务器中保存的索引。所有这些因素最终都将由你的用例决定：存储的是什么数据以及如何获取数据。

12.7 标签分片

有时在特定的环境下，可能会出现“我希望将所有数据都存储在指定的分片服务器上”的情况。这时就需要用到 MongoDB 的标签分片。通过创建分片，可以将片键的指定值重定向至集群中的特定分片！该过程的第一步是分析出希望通过分片设置实现的目标。在接下来的样例中，我们将根据地理位置对数据进行分布，一个分片位于美国，另一个位于欧洲。

第一步是在现有的分片中添加一些标签。调用 `sh.addShardTag` 函数，并在其中添加分片的名字和希望使用的标签。在该例中，`shard0000` 将作为 US 分片服务器，而 `shard0001` 将作为 EU 分片服务器：

```
> sh.addShardTag("shard0000","US");
> sh.addShardTag("shard0001","EU");
```

现在运行 `sh.status()` 命令并查看它的输出：

```
> sh.status();
--- Sharding Status ---
  sharding version: {
    "_id" : 1,
    "version" : 3,
    "minCompatibleVersion" : 3,
    "currentVersion" : 4,
    "clusterId" : ObjectId("51c699a7dd9fc53b6cdc4718")
  }
  shards:
    { "_id" : "shard0000", "host" : "localhost:27023", "tags" : [ "US" ] }
    { "_id" : "shard0001", "host" : "localhost:27024", "tags" : [ "EU" ] }
  ...
```

如你所见，分片服务器已经分片并添加了 US 和 EU 标签，但只有这些什么也无法实现；接下来，我们需要告诉 MongoS 根据某些规则将数据路由到这些分片服务器的指定集合中。这是最有技巧的部分；我们需要配置分片服务器，保证正在分片的数据中包含一些可用于执行规则计算的信息，只有这样它们才能被路由到正确的服务器。与此同时，我们仍然希望保持之前的分片逻辑不变。回想一下之前关于如何划分块的讨论，你会发现大多数情况下，只需要按照区域对块进行划分即可。

这里的解决方案是：在片键中添加一个额外的键，用于表示数据所属的区域，并将该键作为片键的第一个元素。所以，为了添加标签，我们需要对一个新的集合进行分片：

```
> sh.shardCollection("testdb.testtagsharding", {region:1, testkey:1})
{ "collectionsharded" : "testdb.testtagsharding", "ok" : 1 }
```

注意：

片键的标签部分不需要是第一个元素，但通常最好将它用作第一个元素；通过这种方式，分片块将首先按照标签进行划分。

此时，我们已经创建了标签和片键，它们将把块拆分到合理的区域中，接下来我们需要添加规则！使用 `sh.addTagRange` 命令添加规则。该命令将接受集合的名称空间、最小和最大范围值，以及存储该数据的目标标签。MongoDB 的标签范围包含最小值，但排除最大值。因此，如果希望将含有 EU 值的数据发送到标签 EU 中，那么需要将范围设置为 EU 到 EV，对于 US 则需要设置为 US 到 UT，如下所示：

```
> sh.addTagRange("testdb.testtagsharding", {region:"EU"}, {region:"EV"}, "EU")
> sh.addTagRange("testdb.testtagsharding", {region:"US"}, {region:"UT"}, "US")
```

从现在开始，任何符合条件的文档都将被发送到这些分片服务器。下面让我们插入一些文档进行测试。使用循环在集群中添加 10000 个匹配片键的文档。

```
for(i=0;i<10000;i++){db.getSiblingDB("testdb").testtagsharding.insert({region:
"EU",testkey:i})}
```

现在运行 `sh.status()` 并查看分片服务器中块的划分情况：

```
testdb.testtagsharding
  shard key: { "region" : 1, "testkey" : 1 }
  chunks:
    shard0000 3
      { "region" : { "$minKey" : 1 }, "testkey" : { "$minKey" : 1 } -->> { "region" :
"EU", "testkey" : { "$minKey" : 1 } } on : shard0000 Timestamp(1, 3)
      { "region" : "EU", "testkey" : { "$minKey" : 1 } } -->> { "region" : "EU",
"testkey": 0 } on : shard0000 Timestamp(1, 4)
      { "region" : "EU", "testkey" : 0 } -->> { "region" : { "$maxKey" : 1 },
"testkey" : { "$maxKey" : 1 } } on : shard0000 Timestamp(1, 2)
      tag: EU { "region" : "EU" } -->> { "region" : "EV" }
      tag: US { "region" : "US" } -->> { "region" : "UT" }
```

从该输出中可以看到块的划分情况；有三个块存储在 EU 分片服务器中，US 分片服务器中未包含块。从范围上看，其中两个块应该是空的。如果检查每台服务器，你会发现 10000 个文档全部都存储在一个分片中。你可能已经注意到日志中的这些信息：

```
Sun Jun 30 12:11:16.549 [Balancer] chunk { _id:"testdb.testtagsharding-region_"EU"
testkey_MinKey",lastmod: Timestamp 1000|2, lastmodEpoch: ObjectId('51cf7c240a2cd2040f
766e38'), ns: "testdb.testtagsharding", min: { region: "EU", testkey: MinKey }, max:
{ region: MaxKey, testkey: MaxKey },shard: "shard0000" } is not on a shard with the right
tag:
```

```
EU Sun Jun 30 12:11:16.549 [Balancer] going to move to: shard0001
```

因为我们将标签范围设置为只处理 EU 和 US 值，所以日志中出现了该消息。根据我们已经获得的信息，接下来就可以对该范围进行调整，让它们覆盖所有的标签范围。删除这些标签范围并添加新的范围；可以使用下面的命令删除旧文档：

```
> use config
> db.tags.remove({ns:"testdb.testtagsharding"});
```

现在将标签添加回来，但这次可以将范围设置为 `minKey` 到 US 以及 US 到 `maxKey`，正如之前样例的块范围一样！可以使用特有的 `MinKey` 和 `MaxKey` 操作符，它们分表代表分片键范围中的最小值和最大值。

```
> sh.addTagRange("testdb.testtagsharding", {region:MinKey}, {region:"US"}, "EU")
> sh.addTagRange("testdb.testtagsharding", {region:"US"}, {region:MaxKey}, "US")
```

再次运行 `sh.status()` 并检查它的输出；这次分片的情况看起来就好多了：

```
testdb.testtagsharding
  shard key: { "region" : 1, "testkey" : 1 }
  chunks:
    shard0001 3
    shard0000 1
    { "region" : { "$minKey" : 1 }, "testkey" : { "$minKey" : 1 } } -->>
{ "region" : "EU", "testkey" : { "$minKey" : 1 } } on : shard0001 Timestamp(4, 0)
  { "region" : "EU", "testkey" : { "$minKey" : 1 } } -->> { "region" : "EU",
"testkey": 0 } on : shard0001 Timestamp(2, 0)
  { "region" : "EU", "testkey" : 0 } -->> { "region" : "US", "testkey" :
{ "$minKey" : 1 } } on : shard0001 Timestamp(3, 0)
  { "region" : "US", "testkey" : { "$minKey" : 1 } } -->> { "region" :
{ "$maxKey" : 1 }, "testkey" : { "$maxKey" : 1 } } on : shard0000 Timestamp(4, 1)
  tag: EU { "region" : { "$minKey" : 1 } } -->> { "region" : "US" }
  tag: US { "region" : "US" } -->> { "region" : { "$maxKey" : 1 } }
```

数据得到更好的分布，并且范围设置也覆盖到了片键的全部范围，从最小值到最大值。如果再插入一些文档到集合中，它们的数据将被正确地路由到目标分片服务器。一切都正确运行。

12.8 小结

通过分片可以对数据存储进行扩展，以处理极其庞大的数据集。通过分片还可以扩大集群的规模，以匹配系统的增长。MongoDB 提供了一种简单的自动化分片配置，可以满足大多数需求。尽管该过程是自动化的，你仍然可以对它的行为进行调整，从而符合个人特定的需求。分片是 MongoDB 的关键特性之一，该特性将 MongoDB 与其他数据存储技术区分开来。本章讲解了如何将数据通过分片的方式分布到许多 MongoDB 实例中，如何管理和维护分片集群以及如何利用标签分片和哈希片键。通过本书我们希望你能明白，相较于更传统的数据库工具，MongoDB 的目标是为了更好地满足现代网络应用的严峻需求。

本书讲解的主题包括：

- 如何在各种不同的平台上安装和配置 MongoDB。
- 如何使用各种不同的开发语言访问 MongoDB。
- 如何访问产品相关的社区，包括如何获得帮助和建议。
- 如何设计和构建利用 MongoDB 独特优势的应用。
- 如何优化、管理基于 MongoDB 的数据存储，以及故障检测。
- 如何创建跨多个服务器的可扩展服务器配置。

强烈推荐你认真学习本书提供的众多样例。在 PHP MongoDB 驱动的文档中可以找到其他 PHP 样例，网址为 www.php.net/manual/en/book.mongo.php。MongoDB 是一种非常易用的工具，它的易用性和易操作性也鼓励开发者进行实验。所以不要再犹豫，开始使用它！你很快会感激它为应用所提供的各种特性。

本书根据MongoDB的最新版本做了相应更新，其中包含MongoDB最新的所有特性，包括版本2.2中引入的聚集框架和版本2.4中引入的哈希索引。MongoDB是最流行的“大数据”NoSQL数据库技术，并且正处于发展中。来自10gen的David Hows以及具有丰富MongoDB开发经验的Peter Membrey和Eelco Plugge，组成了本书的专家团队，他们在本书中分享了自己的专业知识和经验，帮助你了解成为MongoDB专家所需的所有知识。

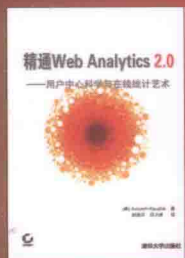
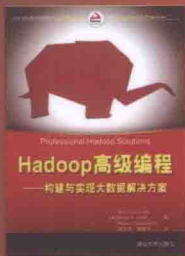
主要内容

- ◆ 在所有主流平台上搭建MongoDB，包括Windows、Linux、OS X和云平台(例如Rackspace、Azure和Amazon EC2)
- ◆ 用GridFS和新的聚集框架
- ◆ 使用非SQL命令处理数据
- ◆ 使用PHP或Python编写应用
- ◆ 优化MongoDB
- ◆ 掌握MongoDB管理方面的知识，包括复制、复制标签和标签分片

源代码下载及技术支持

<http://www.apress.com>

<http://www.tupwk.com.cn/downpage>



清华大学出版社数字出版网站

WQBook

www.wqbook.com

Apress®

www.apress.com

ISBN 978-7-302-38366-6



9 787302 383666 >

定价：49.80元

PDF电子书说明:

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 **QQ: 461573687**, 或者 **QQ: 2404062482**。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢!