- 全面讲解MongoDB的相关知识，使读者对MongoDB有全面的认识
- 以最直接、最细致的方式指导读者轻松掌握MongoDB的安装、部署与使用
- 以实际工作框架为例子进行讲解，使读者真正能胜任MongoDB的开发管理工作
- 由浅入深，层层递进，路线清晰

数据库技术丛书

基于分布式文件存储的数据库

MongoDB游记

之轻松入门到进阶

张泽泉 著

清華大學出版社

MongoDB游记

之轻松入门到进阶

张泽泉 著

清华大学出版社
北京

内 容 简 介

MongoDB 作为最受欢迎的文档存储类型的 NoSQL 数据库，越来越多的公司在使用它。本书以符合初学者的思维方式，系统全面、层层递进地介绍了 MongoDB 数据库，通过本书的学习，读者能够胜任实际工作环境中 MongoDB 的相关开发管理工作。

本书共分四个部分 23 章，第一部分讲解了 MongoDB 的相关概念和原理以及其内部工作机制，可以让读者对 MongoDB 有一个全面的认识。第二部分和第三部分从应用角度，结合实例讲解了 MongoDB 的安装、配置、部署、开发、集群部署和管理等在实际工作中会用到的技能。第四部分是经验部分，这部分是作者多年使用 MongoDB 后总结的技巧，对读者在工作中使用 MongoDB 有极大的参考价值。

本书适合 MongoDB 的初学者，希望深入了解 MongoDB 安装部署、开发优化的软件工程师，希望深入了解 MongoDB 管理、集群扩展的数据运维管理员，以及任何对 MongoDB 相关技术感兴趣的读者。

图书在版编目（CIP）数据

MongoDB 游记之轻松入门到进阶 / 张泽泉著. — 北京：清华大学出版社，2017
（数据库技术丛书）

ISBN 978-7-302-47860-7

I. ①M… II. ①张… III. ①关系数据库系统 IV.①TP311.138

中国版本图书馆 CIP 数据核字（2017）第 181049 号

责任编辑：夏毓彦
封面设计：王　翔
责任校对：闫秀华
责任印制：沈　露

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：北京泽宇印刷有限公司
经　　销：全国新华书店
开　　本：190mm×260mm　　**印　　张**：19　　**字　　数**：486 千字
版　　次：2017 年 9 月第 1 版　　**印　　次**：2017 年 9 月第1 次印刷
印　　数：1～3500
定　　价：59.00 元

产品编号：070704-01

前 言

我思考了很长时间，到底要写一本什么样的书，才能让读者轻松、全面地认识 MongoDB。

从 2012 年有幸开始接触 MongoDB 并在实际工作环境中使用它，不知不觉已经过了 5 年多的时间。在这 5 年中，大数据兴起，NoSQL 来势汹汹。

“有 MongoDB 使用经验优先”“精通 MongoDB 等 NoSQL 数据库”这样的要求也渐渐出现在招聘要求中。MongoDB 作为 NoSQL 数据库的典型代表，越来越多的公司在使用它。

在开始学习使用 MongoDB 的过程中，因为相关书籍资料太少，一路走来确实算是翻山越岭，跋山涉水。这也是本书名的由来。

本书定位

关于本书的定位，在我的想象中应该有如下几点。

1. 这不是一本严肃的教科书

在轻松的氛围中快速学习知识才能达到比较好的效果，所以我会在书中尽可能多地加入图画以帮助读者加深理解。

2. 它能让读者从零开始学习数据库

笔者阅读了很多有关 MongoDB 的书籍，我发现大多数 MongoDB 的书籍在讲解时都喜欢拿关系型数据库进行类比学习，这样会对一个数据库的初学者（不了解关系型数据库的读者）造成困扰，所以我会尽量少用关系型数据库的用法来解释 MongoDB，让没有关系型数据库基础的读者也能独立地理解学习 MongoDB，而对有关系型数据库基础的读者能抛开关系型数据的用法学习 MongoDB。

3. 这是一本入门的书，但不代表它不全面

请原谅我没有在本书中进行深入的原理讲解，因为我一直认为对于学习一个技术来说踏出

第一步尤其重要。一旦你开始使用它，随着解决以后遇到的问题，你自然而然地就会更深入地理解它。

当然，为了让初学者对 MongoDB 有个全面的了解，我会尽可能地把它写得全面。

4. 它能够帮助读者建立 NoSQL 的思维

随着本书从理论到实战的讲解，读者能初步了解并且适应 NoSQL 的思维，或者说脱离关系型数据库的束缚。在之后的实际开发应用中读者会感受到 NoSQL 数据库的魅力——自由（对之前经常使用关系型数据库的读者而言这一点感受会尤其深刻）。

5. 阅读完本书后读者能够胜任实际工作环境中 MongoDB 的相关开发管理工作

这是本书唯一且最终的目的。只要跟着书中的思路一步一步走，读者就能达到相关工作的要求。

6. 一本好书应该包含作者的思想，一些不会过时的东西

当我开始写这本书的时候，MongoDB 的版本是 3.4，当这本书写完发行的时候，我不能确保软件版本还是 3.4，这也是计算机方面的书籍总是容易过时的原因。考虑到这一点，我在写本书时会讲解一些我获取更新知识的途径、我的学习思路以及分享一些有趣的网站，以确保当本书中的例子因为版本原因有些不再适用时，读者能很快地获取到最新的软件用法在哪里。希望这些方法不仅仅为读者在学习 MongoDB 上，也能在学习其他计算机技术时有所帮助。

本书特点

- 内容全面：对 MongoDB 相关知识进行全面讲解，让读者对 MongoDB 有全面的认识。
- 轻松入门：以最直接、最细致的方式指导读者轻松掌握 MongoDB 安装、部署与使用。
- 层次清晰：理论与实践部分分离，4 个部分由浅入深，层层递进，学习路线清晰。
- 实战引导：以实际工作框架为例子，讲解 MongoDB 的运用，让读者真正能胜任 MongoDB 的开发管理工作。

本书适合读者

本书适合 MongoDB 的初学者，希望深入了解 MongoDB 安装部署、开发优化的软件工程师，希望深入了解 MongoDB 管理、集群扩展的数据运维管理员，以及任何对 MongoDB 相关技术感兴趣的读者。

注意事项

本书中所用示例以及操作步骤全部经过作者实际操作，检验有效，请读者在学习时按照本

书选用的软件版本进行操作，已经在使用 MongoDB 软件的开发者在查阅本书示例时，请注意 MongoDB 版本和驱动版本以及环境配置。

本书使用方式

初学者

希望初学者能够通读本书，理论部分快速阅读，实践部分动手操作。

初学者如果时间充裕，建议按本书章节先后顺序进行学习，先通读理论，如有不懂的地方可以不必深究，然后实战，接着进阶架构，最后学习经验篇。

初学者如果时间紧迫，可以从实战部分开始学习，直接上手使用 MongoDB 参与到工作中，闲暇时再学习理论会有更深刻的理解，接着进阶架构，最后再学习经验篇。

有相关经验的技术人员

有相关经验的技术人员可以根据自身需求和薄弱环节选择性阅读。闲暇时可阅读理论部分和经验部分，本书理论部分和经验部分是作者结合自身工作经验对 MongoDB 的理解，希望能与有经验读者的看法相互印证碰撞，让读者形成自己的理解。

工作时也可参考本书相关章节，基本命令章节、Java 驱动操作章节、Spring Data MongoDB 驱动操作章节，管理维护命令、集群构建章节等，可以作为读者身边的速查手册。

祝大家开卷有益！

本书示例代码

本书示例代码以及软件开发包下载地址（注意数字与字母大小写）如下：

http://pan.baidu.com/s/1boKG28R

如果下载有问题，请联系 booksaga@163.com，邮件主题为“MongoDB 代码”。

致谢

感谢 23 魔方公司和大厂开发组，让我在 MongoDB 实战中积累了宝贵的经验。

玩网络游戏，最害怕的是游戏公司不再对游戏进行维护，玩家对游戏失去信心，游戏区变成鬼区。10gen 公司在 2013 年正式更名为 MongoDB Inc，感谢 MongoDB Inc 公司技术人员不断地完善 MongoDB，让我们能够用上更快、更方便、更安全的 MongoDB。没有程序是没有 bug 的，MongoDB 在发布初期确实存在很多问题，感谢那些能够正视 MongoDB 并伴随它成长的人，让一款经典的数据库和一种新的数据思维没有被埋没。好在 MongoDB 已经长成了一颗参天大树，在数据库领域占据了一席之地。

本书在写作期间，参考了大量的 MongoDB 相关书籍，以及 MongoDB 官网、MongoDB 中文社区、CSDN 博客等网站的大量文章，感谢这些使用 MongoDB 并乐于分享的大神们。作

者已经尽力对书中的知识点用词用语做了各种查证，但由于时间仓促，难免有错误和遗漏之处，恳请广大读者提出宝贵意见。

奋斗在一线的程序员，加班加点是常有的事情。编写本书使用了作者一年的周末和假期时间以及很多个 10 小时以后的美好时光。没有家人的鼓励和支持是很难坚持下来的。在此特别感谢我的家人刘悦梦露的陪伴和谅解，以及好朋友杨娅苏、古曙强、姚思伶对我的鼓励。

最后特别感谢清华大学出版社的夏毓彦老师和他的同事们，正因为有了他们的辛勤工作，这本书才能顺利出版。

张泽泉

2017 年 7 月于成都

目 录

第一部分 基础与架构理论篇

第一部分

基础与架构理论篇

第 1 章

◂ 初识MongoDB ▸

1.1 MongoDB 简介

1.1.1 MongoDB 是什么

MongoDB（源自单词 humongous 巨大的）是由 C++语言编写的数据库，目的是为 Web 应用程序[1]提供高性能、高可用性且易扩展的数据存储解决方案，如图 1-1 所示。

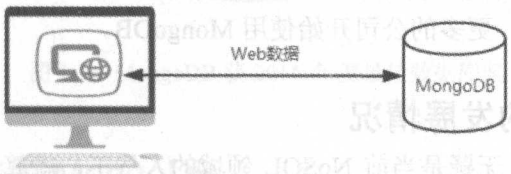

图 1-1　MongoDB 为 Web 提供数据服务

MongoDB 是当前 NoSQL 数据库产品中最热门的一种，是一种开源[2]（目前免费）、容易扩展、表结构自由（模式自由）、高性能且面向文档的数据库。

MongoDB 的官方网址是 https://www.mongodb.com/ 。读者可以在官方网址中获取更多 MongoDB 的相关介绍和更新信息。

1.1.2 MongoDB 的历史

关系型的数据库已经出现了近 40 年，并且在很长一段时间里一直是数据库领域当之无愧的王者，例如 SQL Server、MySQL 和 Oracle 等，目前在数据库领域中仍处于主导地位。但随着信息时代数据量的增大以及 Web2.0 的数据结构复杂化，关系型数据库的一些缺陷也逐渐显现出来，主要包括以下几项。

（1）大数据处理能力差。关系型数据库被设计为单机运行，在处理海量数据方面代价高

---

[1] Web 应用程序是一种可以通过 Web 访问的应用程序。Web 应用程序的一个最大好处是用户很容易访问应用程序。用户只需要有浏览器即可，不需要再安装其他软件。我们平时所说的网站 Web 站点主要提供信息，而 Web 应用程序与用户互动性更强，但二者的界限已越来越模糊。

[2] 开源就是源代码公开，有利于改进代码，开源的不一定免费，需要看采用哪种开源协议才能确定是否免费。

昂，甚至无法承担重任。

（2）程序产出效率低。我们在开发程序使用关系型数据库过程中会发现，更多的精力被用在了建立关系型数据库表的数据结构与开发语言数据结构的映射上。使用关系型数据库时，为了实现系统中某个实体的存储查询操作，我们首先需要设计表的结构和字段以及数据类型。于是无论是创建、删除还是更新，我们要涉及的操作增加了许多。

（3）数据结构变动困难。互联网项目时刻都在发展和变动，改变一个存储单元的结构是常事，但是生产环境中关系型数据库要增加或减少一个字段无疑是非常严肃、重要并且是容易产生意外的事情。

为了解决这些存在的问题，一个更好的数据存储方案，NoSQL 数据库应运而生。所谓 NoSQL，并不是指没有 SQL，而是指“Not Only SQL”，即非传统关系型数据库。这类数据库的主要特点包括非关系型、水平可扩展、分布式与开源；另外，它还具有模式自由、最终一致性（不同于 ACID[3]）等特点。正是由于这些有别于关系型数据库的特点，它更能适用于当前海量数据的环境。NoSQL 常用的存储模式有 key-value 存储、文档存储、列存储、图形存储、XML 存储等，MongoDB 正是文档数据库的典型代表。

带着建立一种灵活、高效、易于扩展、功能完备的数据库的愿景，10gen 团队于 2007 年 10 月开发了 MongoDB 数据库，并于 2009 年 2 月首度推出。经过这几年的发展，MongoDB 数据库已经逐渐趋于稳定，更多的公司开始使用 MongoDB。

1.1.3 MongoDB 的发展情况

MongoDB 发展迅速，无疑是当前 NoSQL 领域的人气王，就算与传统的关系数据库比较也不甘落后，数据库知识网站 DB-Engines 根据搜索结果对 308 个数据库系统进行了流行度排名，2016 年 7 月的数据库流行度排行榜前 10 名如图 1-2 所示。

308 systems in ranking, July 2016

Rank			DBMS	Database Model	Score		
Jul 2016	Jun 2016	Jul 2015			Jul 2016	Jun 2016	Jul 2015
1.	1.	1.	Oracle	Relational DBMS	1441.53	-7.72	-15.20
2.	2.	2.	MySQL	Relational DBMS	1363.29	-6.85	+79.95
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1192.89	+27.08	+89.83
4.	4.	4.	MongoDB	Document store	315.00	+0.38	+27.61
5.	5.	5.	PostgreSQL	Relational DBMS	311.15	+4.55	+38.33
6.	6.	6.	DB2	Relational DBMS	185.08	-3.49	-13.04
7.	7.	↑ 8.	Cassandra	Wide column store	130.70	-0.42	+17.99
8.	8.	↓ 7.	Microsoft Access	Relational DBMS	124.90	-1.32	-19.40
9.	9.	9.	SQLite	Relational DBMS	108.53	+1.75	+2.66
10.	10.	10.	Redis	Key-value store	108.03	+3.54	+12.96

图 1-2　2016 年 7 月 DB-Engines 上的数据库排行榜

[3] ACID 是指数据库事务正确执行的 4 个基本要素的缩写，包含原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）、持久性（Durability）。事务是指访问并更新数据库中各种数据项的一个程序执行单元（unit）。一个支持事务（Transaction）的数据库，必须要具有这 4 种特性，否则在事务过程（Transaction processing）当中无法保证数据的正确性，交易过程极可能达不到交易方的要求，也就是说 MongoDB 目前不支持事务。

我们可以看到前 3 名依然是 Oracle 、MySQL 和微软的 SQL Server，值得关注的是，第 4 名 MongoDB 已经超越了很多传统关系型数据库。前 3 名都是关系数据库，由于历史原因，许多大型的垄断行业仍然在使用这些关系数据库。关系型数据库已经发展了 40 多年，而 MongoDB 距离 2009 年首度推出至今只用了 8 年时间，就达到了如此高度，可见其发展之迅猛。

MongoDB 作为 NoSQL 数据库，产品性能、稳定性和生态系统逐渐走向成熟。在 2014 年流行度榜单中，它是流行度最高的非关系型数据库。2015 年 1 月 DB-engines 网站首次推出了年底评奖，2014 年最佳数据库的荣誉归于 MongoDB，如图 1-3 所示。

图 1-3　MongoDB 获 2014 年度最佳数据库奖

1.1.4　哪些公司在用 MongoDB

目前正在使用 MongoDB 的网站或者企业已经非常多了，如阿里巴巴、腾讯、百度、京东、58 同城、360、视觉中国、大众点评、盛大、Google、Facebook、Ebay、FourSquare、Wordnik、OpenShift、SourceForge、Github 等，很多创业型公司也把 MongoDB 当作首选。更多使用 MongoDB 的网站及企业可查看官网 https://www.mongodb.com/who-uses-mongodb。

1.2　MongoDB 的特点

（1）数据文件存储格式为 BSON（一种 JSON 的扩展）

{"name":"joe"} 这是一个 BSON 的例子，其中"name"是键，"joe"是值。键值对组成了 BSON 格式。

（2）面向集合存储，易于存储对象类型和 JSON 形式的数据

所谓集合（collection）有点类似一张表格，区别在于集合没有固定的表头。

（3）模式自由

一个集合中可以存储一个键值对的文档，也可以存储多个键值对的文档，还可以存储键

不一样的文档，而且在生产环境下可以轻松增减字段而不影响现有程序的运行。

（4）支持动态查询

MongoDB 支持丰富的查询表达式，查询语句使用 JSON 形式作为参数，可以很方便地查询内嵌文档和对象数组。

（5）完整的索引支持

文档内嵌对象和数组都可以创建索引。

（6）支持复制和故障恢复

MongoDB 数据库从节点可以复制主节点的数据，主节点所有对数据的操作都会同步到从节点，从节点的数据和主节点的数据是完全一样的，以作备份。当主节点发生故障之后，从节点可以升级为主节点，也可以通过从节点对故障的主节点进行数据恢复。

（7）二进制数据存储

MongoDB 使用传统高效的二进制数据存储方式，可以将图片文件甚至视频转换成二进制的数据存储到数据库中。

（8）自动分片

自动分片功能支持水平的数据库集群，可动态添加机器。分片的功能实现海量数据的分布式存储，分片通常与复制集配合起来使用，实现读写分离、负载均衡，当然如何选择片键是实现分片功能的关键。如何实现读写分离请参考第 19 章“分片+副本集部署”的介绍。

（9）支持多种语言

MongoDB 支持 C、C++、C#、Erlang、Haskell、JavaScript、Java、Perl、PHP、Python、Ruby、Scala 等开发语言。

（10）MongoDB 使用的是内存映射存储引擎。

MongoDB 会把磁盘 IO 操作转换成内存操作，如果是读操作，内存中的数据起到缓存的作用；如果是写操作，内存还可以把随机的写操作转换成顺序的写操作，总之可以大幅度提升性能。但坏处是没有办法很方便地控制 MongoDB 占多大内存，事实上 MongoDB 会占用所有能用的内存，所以最好不要把别的服务和 MongoDB 放一起。

1.3 MongoDB 应用场景

1.3.1 MongoDB 适用于以下场景

（1）网站数据

MongoDB 非常适合实时地插入、更新与查询，并具备网站实时数据存储所需的复制及高度伸缩性。如果你正考虑搭建一个网站，可以考虑使用 MongoDB，你会发现它非常适用于迭代更新快、需求变更多、以对象数据为主的网站应用。

（2）缓存

由于 MongoDB 是内存型数据库，性能很高，MongoDB 也适合作为信息基础设施的缓存层。在系统重启之后，由 MongoDB 搭建的持久化缓存可以避免下层的数据源过载。以前如果网站应用要做缓存大家都会想到 Memcached 等高性能的分布式内存缓存服务器，但现在内存型数据库 MongoDB 也可以作为选择方案之一，而且缓存数据更可靠。

（3）大尺寸、低价值的数据

使用传统的关系数据库存储一些数据会超级麻烦，首先得创建表格，再设计数据表结构，进行数据清理，得到有用的数据，按格式存入表格中；而 MongoDB 可以随意构建一个 JSON 格式的文档就能把它先保存起来，留着以后处理。

（4）高伸缩性的场景

如果网站数据量非常大，很快就会超过一台服务器能够承受的范围，那么 MongoDB 可以胜任网站对数据库的需求，MongoDB 可以轻松地自动分片到数十甚至数百台服务器。

（5）用于对象及 JSON 数据的存储

MongoDB 的 BSON 数据格式非常适合文档格式化的存储及查询。

1.3.2 MongoDB 不适合的场景

（1）高度事务性的系统

传统的关系型数据库目前还是更适用于需要大量原子性复杂事务的应用程序，例如银行或会计系统。支持事务的传统关系型数据库，当原子性操作失败时数据能够回滚，以保证数据在操作过程中的正确性，而目前 MongoDB 暂时不支持此事务。

（2）传统的商业智能应用

针对特定问题的 BI 数据库需要高度优化的查询方式。对于此类应用，数据仓库可能是更合适的选择。

（3）使用 SQL 方便时

MongoDB 的查询方式是 JSON 类型的查询方式，虽然查询也比较灵活，但如果使用 SQL 进行统计会比较方便时，这种情况就不适合使用 MongoDB。

第 2 章

◂MongoDB的结构▸

要很好地使用 MongoDB，需要对它的组成结构进行了解，本章我们就来学习 MongoDB 的结构。

MongoDB 的组成结构如下：数据库包含集合，集合包含文档，文档包含一个或多个键值对，如图 2-1 所示。

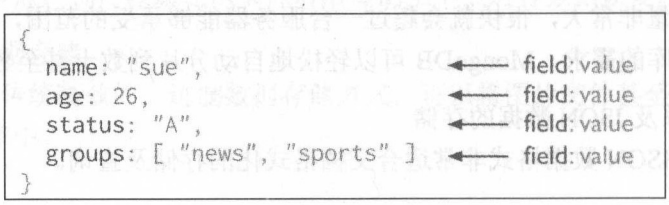

图 2-1　文档包含键值对 key:value

2.1 数据库

2.1.1 数据库的层次

MongoDB 中数据库包含集合，集合包含文档。一个 MongoDB 服务器实例可以承载多个数据库，数据库之间是完全独立的。每个数据库有独立的权限控制，在磁盘上不同的数据库放置在不同的文件中。一个应用的所有数据建议存储在同一个数据库中。当同一个 MongoDB 服务器上存放多个应用数据时，建议使用多个数据库，每个应用对应一个数据库。

2.1.2 数据的命名

数据库通过名字来标识。数据库名可以使用满足以下条件的任意 UTF-8 字符串来命名：

- 不能是空字符串（""）。
- 不能含有' '（空格）、.（点）、$、/、\和\0（空字符）。
- 应全部小写。
- 最多 64 字节。

数据库名有这么多的限制是因为数据库名最终会变成系统中的文件。

2.1.3　自带数据库

MongoDB 有一些一安装就存在的数据库，这些数据库介绍如下：

（1）admin

从权限角度来看，这是超级管理员（"root"）数据库。在 admin 数据库中添加的用户会具有管理数据库的权限。一些特定的服务器端命令也只能从这个数据库运行，如列出所有的数据库或者关闭服务器。

（2）local

这个数据库永远不会被复制，可以用来存储限于本地单台服务器的任意集合。

（3）config

当 Mongo 用于分片设置时，config 数据库在内部使用，用于保存分片的相关信息。

2.2　普通集合

2.2.1　集合是什么

集合就是一组文档。同一个应用的数据我们建议存放在同一个数据库中，但是一个应用可能有很多个对象，比如一个网站可能需要记录用户信息，也需要记录商品信息。集合解决了上述问题，我们可以在同一个数据库中存储一个用户集合和商品集合。集合类似于关系型数据库中的表。

2.2.2　集合的特点——无模式

集合是无模式的，也就是说一个集合里的文档可以是各式各样的，非常自由。集合跟表最大的差异在于表是有表头的，每一列存的什么信息需要对应，表在存储信息之前需要先设计表，每一列是什么数据类型，字符串类型的数据是不能存储进数值类型的列中的。而集合则不需要设计结构，只要满足文档的格式就可以存储，即使他们的键名不同，非常灵活。MongoDB 会自动识别每个字段的类型。

2.2.3　集合命名

集合通过名字来标识区分。集合名可以是满足下列条件的任意 UTF-8 字符串。

- 不能是空字符串（""）。
- 不能含有\0（空字符），这个字符表示集合名的结尾。
- 不能以“system.”开头，这是为系统集合保留的前缀。
- 不能含有保留字符$。这是因为某些系统生成的集合中包含该字符。除非你要访问这种系统创建的集合，否则千万不要在名字里出现$。有些驱动程序的确支持在集合名里面包含$，但是我们不建议使用。

2.2.4 子集合

子集合是集合下的另一个集合，可以让我们更好地组织存放数据。惯例是使用“.”字符分开命名来表示子集合。

在 MongoDB 中使用子集合，可以让数据的组织更清晰。例如我做一个论坛模块，按照面向对象的编程我们应该有一个论坛的集合 forum，但是论坛功能里应该还有很多对象，比如用户、帖子。我们就可以把论坛用户集合命名为 forum.user，把论坛帖子集合命名为 forum.post。

也就是我们把数据存储在子集合 forum.user 和 forum.post 里，数据 forum 集合是不存储数据的，甚至可以删除掉。也就是说 forum 这个集合跟它的子集合没有数据上的关系。子集合只是为了让数据组织结构更清晰。

2.3 固定集合（Capped）

2.3.1 Capped 简介

MongoDB 固定集合（Capped Collections）是性能出色且有着固定大小的集合，对于大小固定，我们可以想象它就像一个环形队列，如果空间不足，最早的文档就会被删除，为新的文档腾出空间。这意味着固定集合在新文档插入的时候自动淘汰最早的文档。

2.3.2 Capped 属性特点

（1）对固定集合插入速度极快。

（2）按照插入顺序的查询输出速度极快。

（3）能够在插入最新数据时，淘汰最早的数据。

（4）固定集合文档按照插入顺序储存，默认情况下查询全部就是按照插入顺序返回的，也可以使用$natural 属性反序返回。

（5）可以插入及更新，但更新不能超出 collection 的大小，否则更新失败。

（6）不允许删除，但是可以调用 drop()删除集合中的所有行，drop 后需要显式地重建集合。

（7）在 32 位机器上一个 cappped collection 的最大值约为 482.5MB，64 位机器上只受系统文件大小的限制。

2.3.3 Capped 应用场景

（1）储存日志信息。

（2）缓存一些少量的文档。

一般来说，固定集合适用于任何想要自动淘汰过期文档的场景，没有太多的操作限制。

2.4 文档

2.4.1 文档简介

文档是 MongoDB 中数据的基本单元。我们前面已经讲过 MongoDB 数据存储格式为 BSON。键值对按照 BSON 格式组合起来存入 MongoDB 就是一个文档。

2.4.2 文档的特点

（1）每一个文档都有一个特殊的键“_id”，它在文档所处的集合中是唯一的。
（2）文档中的键值对是有序的，前后顺序不同就是不同的文档。
（3）文档中的键值对，值不仅可以是字符串，还可以是数值，日期等数据类型。
（4）文档的键值对区分大小写。
（5）文档的键值对不能用重复的键。

2.4.3 文档的键名命名规则

文档的键是字符串。除了少数例外情况，键可以使用任意 UTF-8 字符。

（1）键名不能含有\0（空字符）。
（2）键名最好不含有.和$，它们存在特别含义。
（3）键名最好不使用下划线“_”开头。

2.5 数据类型

数据类型在数据结构中的定义是一个值的集合以及定义在这个值集合上的一组操作。通俗地说，数据类型的意义就是告诉计算机这个变量是用来干什么的。

比如我们有一个值是“2016-08-15”和一个值是“2016-08-16”，当它们是字符串类型时，它就是一个文本，它们的比较是没有多大意义的；而当它们都是日期类型时，它们就有了先后之分。我们在数据库的使用中，可以使用日期字段作为排序。由此可见了解数据类型可以帮助我们更好地使用 MongoDB。

更多数据类型的信息可查看官网：https://docs.mongodb.com/manual/reference/bson-types/。

2.5.1 基本数据类型

MongoDB 支持比较丰富的数据类型。如果你学过一些编程语言你会发现很多相似的类型，因为它们确实是共通的。比如我在 Java 语言中用 MongoDB 的 Java 驱动存入一个 Java

的整数，那么 MongoDB 中保存的数据类型也是一个整数。整数根据存储时分配的内存位数又分为 32 位整数和 64 位整数，它们在 MongoDB 中的表达有些特殊，会在 2.5.2 小节中详细说明。

MongoDB 支持的基本数据类型如表 2-1 所示。

表 2-1 MongoDB 的基本数据类型

数据类型	文档表示方式	说明
null	{"key":null}	Null 表示空值或者不存在该字段
布尔	{"key":true} {"key":false}	布尔类型表示真和假，有两个值分别是 true 和 false
32 位整数	{"key":8}	MongoDB 数据库可以存 32 位的整数。 但通过 shell 界面来显示的话，会自动转成 64 位的浮点数
64 位整数	{"key":{"floatApprox":8}}	floatApprox 的意思是用 64 位浮点数近似地表示了一个 64 位的整数
64 位浮点数 Double	{"key":8.21} {"key":8}	MongoDB 数据库可以存 64 位的浮点数。shell 客户端 Mongo 里数字都是这种类型
字符串	{"key":"value"} {"key":"8"}	字符串类型起到记录展示文本的作用，只要是 UTF-8 的字符串都能当作字符串类型。注意这里的字符串与浮点数类型的区别——字符串有双引号，浮点数没有双引号
对象 id	{"key":ObjectId（）}	对象 id 类型是 12 字节的唯一 ID
日期	{"key":new Date()}	日期类型用来表示时间，一般日期格式的数据存储到 MongoDB 会自动识别成日期类型。日期类型存储的是从标准纪元开始的毫秒数，不存储时区
正则表达式	{"name": /张/}	正则表达式可以作为值选出 key 对应的值符合正则规则的数据。例子中就表示 name 字段中含有“张”这个字的数据
代码	{"key":function(){}}	我们也可以用 JavaScript 代码段作为 value，表示符合代码段的数据
二进制数据		二进制主要表示文件的存储，不过在 shell 中是无法使用和表示的
未定义	{"key":undefined}	表示该值没有定义，JavaScript 中 null 和 undefined 是不同的类型
数组	{"age":[16,18,20]}	表示值的集合或者列表
内嵌文档	{"user":{"name":" 张小凡 "}}	文档里可以包含文档
Decimal128	{ "price": NumberDecimal("2.099") }	MongoDB 3.4 版本新增对 decimal128 数据类型的支持，最多支持 34 位小数位。 跟 Double 类型不同，decimal 数据存储的是实际的数据，无精度问题，以 9.99 为例，decimal NumberDecimal("9.99") 的值就是 9.99；而 Double 类型的 9.99 则是一个大概值 9.99000000000000002131628.... 金额的操作一般都使用 Decimal 类型才不会造成精度丢失

2.5.2 数字类型说明

关于 MongoDB 数字的数据类型，理解起来可能有点绕，不过没关系。

我们只要清楚通过 JavaScript shell（MongoDB 客户端 Mongo 命令行交互界面，详见 9.4 节启动 MongoDB 客户端）存入数值在 MongoDB 中都是 64 位浮点数。

通过 Java 等语言存入 MongoDB 时会根据 Java 等语言的数据类型保存成 32 位或者 64 位整数，或者 64 位浮点数。如图 2-2 所示。

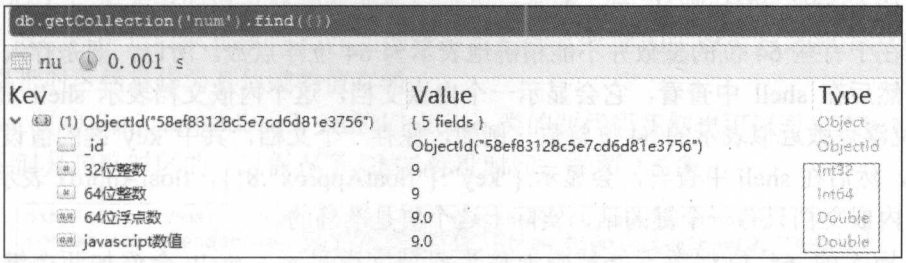

图 2-2　JavaScript shell 存入的数值和 Java 存入的数值

之所以会有这样的区别原因如下：

- 计算机在存储时使用的二进制位数有 32 位（4 字节）和 64 位（8 字节）之分。位数越多表示存储的数值范围越广，计算能力越强。
- 数据类型中所说的 32 位整数，64 位整数的区别也就是使用的二进制位数不同，64 位的位数多一些，可以表示的值域也就大一些。

32 位整数，带正负符号的话可以表示的数值范围是-2^31～2^31-1，也就是-2147483648～2147483647；不带符号可以表示的数值范围是无符号整数的范围 0~2^32-1，也就是 0～4294967295。如表 2-2 所示。

表 2-2　32 位二进制数的值

二进制数	表示的值（十进制）
0000 0000 0000 0000 0000 0000 0000 0000	0
0000 0000 0000 0000 0000 0000 0000 0001	1
0000 0000 0000 0000 0000 0000 0000 0010	2
0000 0000 0000 0000 0000 0000 0000 0011	3
0000 0000 0000 0000 0000 0000 0000 0100	4
0000 0000 0000 0000 0000 0000 0000 0101	5
1111 1111 1111 1111 1111 1111 1111 1111	4294967295

64 位整数，带正负符号的话可以表示的数值范围是-2^63~2^63-1，也就是-9223372036854775808～9223372036854775807；不带符号可以表示的数值范围是 0~2^64-1，也就是 0~18446744073709551615。

C#语言中用 int32 和 int64，分别表示 32 位整数和 64 位整数。

Java 语言用 int 和 long 分别表示 32 位整数和 64 位整数。

因为 MongoDB 支持 32 位整数和 64 位整数，所以如果用其他语言的驱动（比如 Java 的 int）保存整数进去，MongoDB 就保存的是整数。

但是我们对 MongoDB 的查看和操作主要是通过 shell 界面和 JavaScript 命令。JavaScript 只支持 64 位浮点数。

也就是说我们在其他语言驱动中存入 MongoDB 的整数经过 shell 界面的操作后再存入数据库时，整数会被自动转换成 64 位浮点数（即使我们没有特意去修改操作这个整数，保持它原封不动地存入）。

32 位的整数都能用 64 位的浮点数精确表示，所以从文档格式上看 32 位整数跟 64 位浮点数没有什么区别。例如都是: {"key":8}。

问题在于有些 64 位的整数并不能精确地表示为 64 位浮点数。所以，要是存入了一个 64 位整数，然后在 shell 中查看，它会显示一个内嵌文档，这个内嵌文档表示 shell 显示的是一个用 64 位浮点数近似表示的 64 位整数。例如，保存一个文档，其中"key"键的值设为一个 64 位整数 8，然后在 shell 中查看，会显示:{"key":{"floatApprox":8}}，floatApprox 表示可能不准确。若是内嵌文档只有一个键的话，实际上这个值是准确的。

要是插入的 64 位整数不能精确地作为双精度数显示，shell 会添加两个键，"top"和'bottom"，分别表示高 32 位和低 32 位。例如，如果插入 9 223 372 036 854 775 807，shell 会这样显示（如图 2-3 所示）：

```
{"key":
{
"floatApprox":9 223 372 036 854 776000,
"top":2147483647,
"bottom":4294967295
}
}
```

```
> db.num.save({"javascript写入64位整数的值":{"floatApprox":9}})
WriteResult({ "nInserted" : 1 })
> db.num.find().pretty()
{
        "_id" : ObjectId("58e78c7cbdbacc799deda87b"),
        "javascript写入64位整数的值" : {
                "floatApprox" : 9
        }
}
>
```

图 2-3　MongoDB 客户端 JavaScript Shell 存入 64 位整数

总结：在 MongoDB 中使用数字类型需要注意精度和极限值的问题，特别是金额等敏感数字需要使用 128 位的 Decimal 类型才不会导致精度丢失而造成数值变化。Decimal 类型在 MongoDB3.4 版本才支持，带有 Decimal 类型数值的文档在有些第三方图形操作客户端工具中无法显示。

2.5.3　日期类型说明

在 shell 中 JavaScript 语言的 Date 对象对应 MongoDB 的日期类型 ISODate。其他语言比如 Java 中的 Date 对象通过驱动存入 MongoDB 都会自动保存成 MongoDB 日期类型 ISODate，如图 2-4 所示。

```
> db.date.find()
{ "_id" : ObjectId("58ef95978c5e7ce098db6401"), "时间类型" : ISODate("2017-04-13T15:13:27.956Z") }
>
```

图 2-4　MongoDB 客户端 JavaScript Shell 查看时间类型

MongoDB 日期类型的值可以判断先后，也可以加减。但有一点需要注意的是，日期在 MongoDB 数据库中是以从标准纪元开始的毫秒数的形式存储的，没有与之相关的时区信息（当然可以把时区信息作为其他键的值存储）。

从 Java 驱动 com.mongodb.util. JSONCallback 类的源代码我们也可以看出来，MongoDB 在存储日期时是忽略时区的，只保存了 GMT 标准时间，如图 2-5 所示。

```
SimpleDateFormat format = new SimpleDateFormat(_msDateFormat);
format.setCalendar(new GregorianCalendar(new SimpleTimeZone(0, "GMT")));
o = format.parse(b.get("$date").toString(), new ParsePosition(0));

if (o == null) {
    // try older format with no ms
    format = new SimpleDateFormat(_secDateFormat);
    format.setCalendar(new GregorianCalendar(new SimpleTimeZone(0, "GM
    o = format.parse(b.get("$date").toString(), new ParsePosition(0));
}
```

图 2-5　Java 驱动保存 Date 数据时时区为 0

默认情况下 MongoDB 中存储的是标准的时间，中国时间是东八区，我们把当前的时间存入 MongoDB 就会发现少 8 个小时。

比如中国在 GMT+8 时区，Java 使用 Date 方法获取当前时间是带有时区的。Java 驱动把当前时间 2017-04-13 23:13:27 保存到 MongoDB 数据库中，查询后结果竟然是 2017-04-13 15:13:27，缺少了 8 个小时。因为只存了 GMT 部分，没有存时区部分。

所以使用 MongoDB 时记得时区对日期类型造成的影响：存入 MongoDB 后会缺少时区部分的，例如在中国取当前时间存入就会减少 8 小时，使用 MongoDB 中的日期可以加上 8 小时来使用。当然有些计算机语言的驱动已经帮我们处理好了，比如通过 Java 驱动读取时会自动加上时区 8 小时。所以，如果你是使用 Java 驱动存储，Java 驱动来读取的话可以忽略时区问题，如图 2-6 所示。

```
Problems  Javadoc  Declaration  Progress  Debug  Servers  Console
<terminated> TestNum [Java Application] C:\Program Files\Java\jre1.8.0_121\bin\javaw.exe (2017年4月13日 下午11:19:45)
17 11:19:47 下午 com.mongodb.diagnostics.logging.JULLogger log
ter created with settings {hosts=[192.168.199.8:27017], mode=SINGLE,

17 11:19:47 下午 com.mongodb.diagnostics.logging.JULLogger log
ed connection [connectionId{localValue:1, serverValue:90}] to 192.16
17 11:19:47 下午 com.mongodb.diagnostics.logging.JULLogger log
tor thread successfully connected to server with description ServerD
17 11:19:47 下午 com.mongodb.diagnostics.logging.JULLogger log
ed connection [connectionId{localValue:2, serverValue:91}] to 192.16
{_id=58ef95978c5e7ce098db6401, 时间类型=Thu Apr 13 23:13:27 CST 2017}}
```

图 2-6　Java 驱动读取 MongoDB 时间自动加上 8 小时

2.5.4 数组类型说明

MongoDB 的数组由一组值组成，数组中可以包含不同数据类型的元素，甚至是内嵌数组也能作为其中的一个元素。

例如：

```
{"list":["d",8888,[ "a",123, "b",12.23] ,65, "c"]}
```

list 的值是一个数组，里面包含了 5 个元素分别是字符串"d"和"c"，数值 8888 和 65 以及内嵌数组["a",123, "b",12.23]，内嵌数组中又包含了字符串 a 和 b 以及数值 123 和 12.23。

2.5.5 内嵌文档类型说明

1. 普通内嵌文档

把整个 MongoDB 文档作为另一个文档中键的值称为内嵌文档。使用内嵌文档可以更好地组织数据，而不需要几个集合相互关联才能得到完整的一个对象数据。

例如，存储员工的信息，员工作为一个对象，我们需要保存他的基本信息，还要保存他的部门信息，这时就可以将部门信息作为内嵌文档。

```
{
"name":"张小凡",
"age":26,
"sex":1,
"department ": {"id":"1","name":"开发部","number ":10}
}
```

{"id":"1","name":"开发部","number ":10}作为 department 的值是一个内嵌文档。

使用内嵌文档的好处在于页面展示时方便，只需要操作一个集合即可展示一个完整的对象。

当然也会导致一些坏处，因为内嵌文档会导致储存更多的重复数据，这样是反规范化的。

如果 department 是一个单独的集合，员工信息文档中只是通过 id 关联它，那么我们要修改部门信息时，只需要修改部门这一条数据即可。其他关联了这一条部门信息的员工信息都会得到更新。但内嵌文档如果要修改部门信息，则需要修改所有内嵌了这个部门信息的员工数据。

总结来说：普通内嵌文档读取方便，修改涉及数据量多。集合关联的方式修改方便，读取麻烦。

2. 自动关联内嵌文档 DBRef

普通内嵌文档读取方便，修改复杂。而如果在文档中通过 id 来手动关联另一个集合则是

修改方便，读取麻烦，需要手动写两次查询。那有没有两全其美的办法？DBRef 自动关联内嵌文档解决了这个问题。DBRef 的意思是按照规范格式来存储关联 id，MongoDB 就会自动实现关联。格式是：

```
{ $ref : <value>, $id : <value>, $db : <value> }
```

$ref：集合名称；$id：引用的 id；$db:数据库名称，可选参数。

按照这个格式存储的 id 会自动关联引用指定集合中对应文档。也就是说我们可以把这个对应文档当成普通内嵌文档那样读取，很方便。修改时也只要修改指定集合中的对应文档，则所有引用了对应文档的文档也会得到更新。

举例来说：

我们有一个部门信息 department 集合，里面保存了一条部门数据：

```
{"id":"1","name":"开发部","number ":10}
```

我们在员工信息 person 集合中有多条数据需要用到部门信息，我们就可以按照 DBRef 的格式把部门信息关联进去如下：

```
{
"name":"张小凡",
"age":26,
"sex":1,
"department ": {"$id":"1","$ref ":" department "}
}
```

这样就实现了自动关联，当我们要展示这个员工的数据时只需要读取一次 person 集合就行了，DBRef 会自动的帮我们关联查询出 department 数据，我们读取得到的文档如下：

```
{
"name":"张小凡",
"age":26,
"sex":1,
"department ": {"id":"1","name":"开发部","number ":10}
}
```

而当我们要修改部门信息，比如把人数 number 增加 2，也只需要把 department 集合中数据修改成{"id":"1","name":"开发部","number ":12}即可。所有关联了这条数据的文档再次读取时，department 字段的值都会自动更新成{"id":"1","name":"开发部","number ":12}。

2.5.6　_id 键和 ObjectId 对象说明

_id 是 MongoDB 中的文档的唯一标识，也就是说通过_id 你就能对应找到集合中的某个文档。

ObjectId 是 MongoDB 特有的数据类型，它是一串满足一些特定规律的字符，可以作为自

动生成的_id 并且保证即使 MongoDB 是分布式的部署也不会重复，如图 2-7 所示。

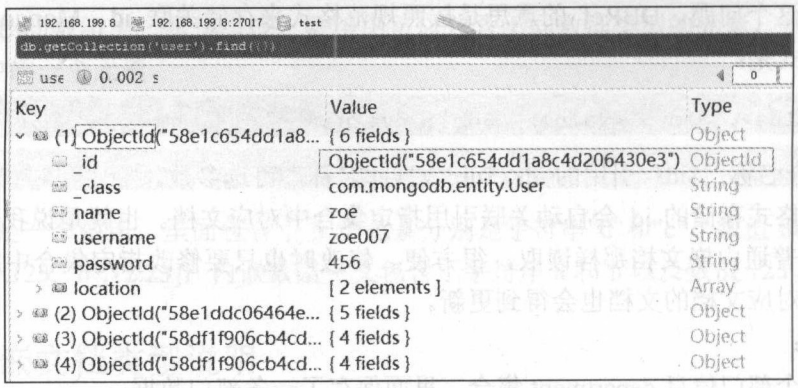

图 2-7　_id 和 ObjectId

1. _id

MongoDB 中存储的文档必须有一个“_id”键。这个键的值可以是任何类型的，默认是个 ObjectId 对象，也可以我们自定义，但是自定义的_id 想要它不重复维护起来就比较麻烦了，特别是 MongoDB 部署在多台服务器中时维护自定义的 id 更麻烦，所以一般是直接使用 ObjectId，这也是 ObjectId 存在的意义。在一个集合里面，每个文档的“_id”值不能有重复来确保集合里面每个文档都能被唯一标识。比如同一个集合中只能有一个“_id”是 123 的文档。不同集合则不需要唯一，如果是两个集合的话，两个集合都可以有一个“_id”是 123 的文档。

2. ObjectId

ObjectId 是“_id”的默认类型。它的存在是为了满足 MongoDB 分布式部署在多台机器上时_id 也能不重复地自动增长。

我们现在就来看看 ObjectId 是如何实现不同的机器都能用全局唯一的同种方法，方便地生成自动增长 id 并且保证生成的 id 是不重复的，如表 2-3 所示。

表 2-3　ObjectId 创建方式

0	1	2	3	4	5	6	7	8	9	10	11
时间戳				机器			PID		计数器		

ObjectId 使用 12 字节的存储空间，每个字节两位十六进制数字，显示出来是一个 24 位的字符串，例如：ObjectId("583aaf224395860d483d4af8")，ObjectId("583aaf234395860d483d4af9")。如果快速连续创建多个 ObjectId，会发现每次只有最后几位数字有变化。另外，中间的几位数字也会变化（如果在创建的过程中停顿几秒钟），这是 ObjectId 的创建方式导致的。

前 4 个字节是从标准纪元开始的时间戳，单位为秒。这会带来如下有用的属性。

- 时间戳，与随后的 5 个字节组合起来，提供了秒级别的唯一性。
- 由于时间戳在前，这意味着 ObjectId 大致会按照插入的顺序排列。这对于某些方面

很有用，如将其作为索引提高效率，但是这个是没有保证的，仅仅是“大致”。

- 这 4 个字节也隐含了文档创建的时间。绝大多数驱动都会公开一个方法从 ObjectId 获取这个信息。

因为使用的是当前时间，很多用户担心要对服务器进行时间同步。其实没有这个必要，因为时间戳的实际值并不重要，只要其总是不停增加就好了（每秒一次）。

接下来的 3 字节是所在主机的唯一标识符。通常是机器主机名的散列值。这样就可以确保不同主机生成不同的 ObjectId，不产生重复冲突。

为了确保在同一台机器上并发的多个进程产生的 ObjectId 是唯一的，接下来的两字节来自产生 ObjectId 的进程标识符（PID ）。

前 9 字节保证了同一秒钟不同机器不同进程产生的 ObjectId 是唯一的。后 3 字节就是一个自动增加的计数器，确保相同进程同一秒产生，ObjectId 也是不一样的。

同一秒钟最多允许每个进程拥有 256^3（16 777 216）个不同的 ObjectId。

2.5.7　二进制类型说明——小文件存储

MongoDB 的存储基本单元是 BSON 文档对象，字段值可以是二进制类型。也就是说 MongoDB 可以储存图片、视频、文件资料。但是有一个限制，因为 MongoDB 中的单个 BSON 对象目前为止最大不能超过 16MB，所以这种方式只能存储小文件。

应用场景：在网站中用户可以上传自己的照片、常用的文件（格式如 doc、pdf、excel、ppt 等不限），其中单个照片、文件基本上小于 16MB，这种情况直接使用 MongoDB 的二进制存储功能就能存储。

只要在 MongoDB 的驱动语言（例如 Java）中构造一个 BSON 对象，把图片、文件转换为二进制值作为 BSON 对象的值存入 MongoDB 即可。

2.6　索引简介

2.6.1　什么是索引

索引就是给数据库做一个目录，类似于字典的目录。字典的目录在我们查找信息时给我们提供了很多方便，同样地，有了索引，我们在数据库中查询数据就不需要扫描整个库了，而是先在索引中查找，使得查询的速度能提高几个数量级（当然是在我们正确的建立索引的前提下，建立索引有很多的技巧，读者可参考 22.5 节索引设置的技巧），在索引中找到条目之后，就可以直接跳转到目标文档的位置。索引的例子如图 2-8 所示。

汉语拼音音节索引

1. 每一音节后举一字做例，可按例字读音去查同音的字。
2. 数字指本字典正文页码。

图 2-8　新华字典索引

2.6.2　索引的作用

了解了索引之后，我们知道索引是用来加速查询的。除此之外，索引还能帮助排序。如果用没做索引的键来排序，MongoDB 需要把所有数据放到内存中进行排序，如果集合太大了 MongoDB 就会报错。

这种情况我们可以对需要排序的键设置索引，MongoDB 就能按索引顺序提取数据，这样就能排序大规模的数据，而不必担心内存用光。

2.6.3　普通索引

我们可以给 MongoDB 文档中任何一个键建立索引，无论这个键的数据类型是什么，甚至可以是文档。也可以同时给两个键建立索引，组合索引。这样的索引我们都把它归类为普通索引（区别于唯一索引）。

2.6.4　唯一索引

唯一索引是用 unique 属性给索引声明，表示这个索引是唯一的，不允许这个键有重复的值出现。如果对有重复数据的键建立唯一索引会建立失败，对已经建立了唯一索引的集合插入重复数据也会看到存在重复键的提示（安全插入的模式下才有提示）。

唯一索引也可以复合，创建复合唯一索引的时候，单个键的值可以重复，只要所有键的值组合起来不同就好。

2.6.5　地理空间索引

现在大多数的软件开发都与地图地址有关，比如大众点评、美团的定位、附近有哪些店、滴滴打车搜索附近的车等 LBS（基于位置的服务）相关项目，一般存储每个地点的经纬度的坐标，如果要查询附近的场所，则需要建立索引来提升查询效率。

MongoDB 1.4 中引入了地理空间索引。这是比较新的特殊索引技术，与 SQL Server 等关系型数据库中的空间索引类似，通过使用该特性可以索引基于位置的数据，从而处理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。如果你需要使用地址坐标，那么 MongoDB 的地理空间索引会给你提供方便和高效的查询。

第 3 章
MongoDB的大文件存储规范 GridFs

3.1 GridFS 简介

我们在 2.5.7 小节中已经说了 MongoDB 是支持二进制数据类型的，也就是能存储文件。但是这里有个限制，因为 MongoDB 中的单个 BSON 对象目前为止最大不能超过 16MB，这个限制是为了避免单个文档过大，完整读取时对内存或者网络带宽占用过高。根据目前 MongoDB 主开发人员的意思，他们不打算放开这个限制，但会随着计算资源相对成本的降低（内存更便宜，网络更快）而适度调高[4]，这样的限制其实是有助于我们更改不良的数据库结构设计，所以短期内应该不会取消这样的限制，所以为了应对存储更大的文件，MongoDB 提供了 GridFS，如图 3-1 所示。

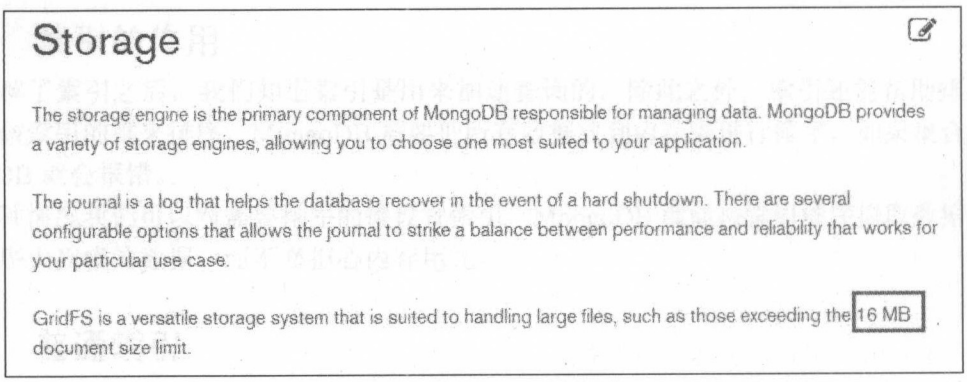

图 3-1　单个文档大小限制在官网的相关信息

GridFS 是一种将大型文件存储在 MongoDB 数据库中的文件规范。所有 MongoDB 官方支持的语言（Java、C#、PHP、Perl 等）驱动均实现了 GridFS 规范，都可以实现将大型文件保存到 MongoDB 数据库中。

---

[4] 1.7.2 版本之前是 4MB，目前是 16MB，什么时候会调高可以在官网中关注，例如访问 https://docs.mongodb.com/manual/storage/

3.2 GridFS 原理

GridFS 本质上还是建立在 MongoDB 的基本功能上的，那么它是如何实现大文件存储的呢？其实我们可以把大文件分成很多份满足 BSON 单文档限制条件的小文件来保存。是的，GridFS 的原理就是规定了一套规范，告诉 MongoDB 怎样自动分割大文件，形成许多小块，然后将这些小块封装成 BSON 对象，插入到特意为 GridFS 准备的集合中。GridFS 规范指定了一个将文件分块的标准。大文件分成小块后，每块作为一个单独的文档存储。然后用一个特别的文档记录来存储分块的信息和文件的元数据[5]，也就是记录这些小块装的是哪一段信息，先后顺序是怎样的，等到用的时候就能按顺序拼接起来返回一个完整的大文件。

默认情况下为 GridFS 准备的集合是 fs.files 和 fs.chunks，如图 3-2 所示。当然在其他驱动语言中可以自己命名这 2 个集合。

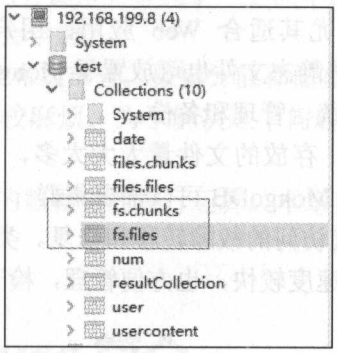

图 3-2　fs.chunks 和 fs.files

- fs.files：用来存储元数据对象。
- fs.chunks：用来存储二进制数据块。

fs.files 中的每个文档代表 GridFS 中的一个文件，与文件相关的自定义元数据也可以存在其中。GridFS 规范还定义了一些 fs.files 文档必需的键：

- _id：文件唯一的 id，在块中作为" files_id"键的值存储。
- Length：文件内容总的字节数。
- chunkSize：每块的大小，以字节为单位。默认是 256KB，必要时可以调整。
- uploadDate：文件存入 GridFS 的时间戳。
- md5：文件内容的 md5 校验和，在服务器端由 filemd5 生成，用于计算上传块的 md5 校验和，用户可以校验 md5 的值，确保文件正确上传了。

fs.chunks 用来存储块，结构很简单，GridFS 规范定义了一些 fs. chunks 文档必需的键：

[5] 元数据是关于数据的组织、数据域及其关系的信息，简言之，元数据就是关于数据的数据，主要是描述数据的信息，算是一种电子目录，用来记录存储的位置等。

- _id：和别的 MongoDB 文档一样，块也有自己唯一的标记。
- files_ id：是包含这个块元数据的文档的_id，对应 fs.files 集合中文档的_id。
- n：表示块编号，也就是这个块在原文件中的顺序编号。
- data：包含组成文件块的二进制数据。

看到这里我想读者已经对 GridFS 的工作机制和原理有了一些理解，如果对 GridFS 的使用有优化需求的话，可以多了解一下它的细节，但是一般使用的话，不必深究 GridFS 规范的工作细节，了解它的大体工作原理以及各个语言版本的驱动中有关 GridFS API 的部分或是如何使用 mongofiles 工具即可。

3.3 GridFS 应用场景

（1）有大量的上传图片（尤其适合 Web 应用，用户上传或者系统本身的文件发布等），类似于 CDN 的功能，一些静态文件也可放置于 MongoDB 中，而不用像以前一样放于其他文件管理系统中，这样方便统一管理和备份。

（2）很多大文件需要存放，存放的文件量太大太多，单台文件服务器已经放不下的情况，可以考虑使用 GridFS，毕竟 MongoDB 可以部署集群。

（3）文件的备份，文件系统访问的故障转移和修复。类似于一些比较小型的存储系统，比如说小型网盘，可以做到存取速度较快，也方便管理，检查重复文件等也比较方便。

3.4 GridFS 的局限性

GridFS 也并非十全十美的，它也有一些局限性：

（1）工作集

随着数据库中 GridFS 文件会越来越多地显著搅动 MongoDB 的内存工作集，如果你不想让 GridFS 的文件影响到你的内存工作集，那么可以把 GridFS 的文件存储到不同的 MongoDB 服务器上。

（2）性能

文件服务性能会慢于从 Web 服务器或文件系统中提供本地文件服务的性能，但是这个性能的损失换来的是管理上的优势。

（3）原子更新

GridFS 没有提供对文件的原子更新方式。如果你需要满足这种需求，那么你需要维护文件的多个版本，并选择正确的版本。

第 4 章

MongoDB的分布式运算模型 MapReduce

我们在前面三章已经了解了 MongoDB 作为一个数据库在数据存储方面的属性和功能。一个数据库，不仅仅是要存储数据，有时候也需要提供一些简单的运算，包括对数据进行比较、排序等。MongoDB 提供了聚合框架，实现了一些简单的常用功能，比如 count、distinct 和 group （后面的命令章节我们会学到）。

MongoDB 的特点在于可以分布式部署，数据分散存储在不同的计算机中。这就导致了要对数据做比较、排序等运算会比较麻烦。为了解决这个问题，比较复杂的运算操作则采用了分布式的运算模型 MapReduce。

这个模型实现了分布式保存的数据也能进行运算。本章我们就来学习这个模型。

4.1 MapReduce 简介

MapReduce 是一种编程模型，一种分布式编程思想，尤其适合处理大数据。那 MapReduce 到底是干什么的呢？

我们可以结合工作中遇到的问题情景来理解。

笔者之前有一个运算任务，是对比几个网站之间的数据。

假设每个网站有 5 万条数据，2 个网站之间需要比较 25 万次。随着网站的增加，比较次数增长很快。如果用 1 台机子来进行运算，即使用上多线程，因为单机的性能瓶颈，可能需要 5 天。但是我们如果用 2 台机子来运算，可能需要 2.5 天（理想状态），但是需要手动分割任务。如果用 5 台，10 台甚至更多计算机就可能把时间缩短到 1 天、甚至几个小时即可运算完成。这就是分布式运算。

但是传统的分布式运算，需要我们人工地去切分任务。

MapReduce 则具有一定的策略，只要我们设置相关配置，只需要一次输入这几个网站的所有数据，就可以帮助我们很方便地进行自动分类、任务分配，并运算。

这下我们就清楚了，MapReduce 是一个根据我们给的规则能够自动分割任务，在多台计算机中运算并返回结果给我们的编程模型。

简单地说就是将大批量的工作（数据）分解，将每个部分发送到不同的计算机中执行，让每台计算机都完成一部分，然后再将结果合并成最终结果。这样做的好处是在任务被分解

后，可以通过大量机器进行并行计算，减少整个操作的时间。

4.2 MapReduce 原理

MapReduce 是怎样实现分割任务和运算返回的呢？主要通过 map、shuffle、reduce 三个过程。Map（映射）是让对象表明身份，shuffle（洗牌）是把对象按表明的身份进行分割集中排列，reduce（化简）是把多个集中排列好的结果集简化成最终结果。

我们通过身边的一个例子来理解，就是我们军训时最常见的报数。

假设我们自己是教官，面前凌乱地站着很多参与军训的学生。现在需要知道参与军训的人数总和。

一种方法是派一个同学去数人数，一个一个地清点，这种方法类似于单机单线程运行。

第二种方法是派多个同学去数人数，然后分别上报，这种方法类似于单机多线程运行。

第三种方法则是教官制定规则，比如按班级集中在一起，报数。然后每班把报数情况集中到一个同学手中，这个同学再对每个班级的报数进行汇总并将结果反馈给教官。这种方法就类似于 MapReduce。

从三种方法对比中我们也可以看出，当人数很多时，MapReduce 使用起来是比其他两种方法高效和有序的，它的工作流程对于用户来说是封装好的。我们只需要按照格式规则对每个过程操作，它就能自动完成分布式的运算了。

按班级集中在一起，报数，就是我们的 map 过程。map 一般需要提交 2 个参数：key 和 value。在 map 部分需要输入<key，value>。key 作为映射规则，我们这里按照班级分，所以 key 就对应班级字段。如果按照男女分，key 就对应性别字段。value 作为参与运算的值（每个对象提供的属性值），我们这里只是要计数，所以 value 是计数器数字 1，也就是说每个对象都提供一个 1，如图 4-1 所示。

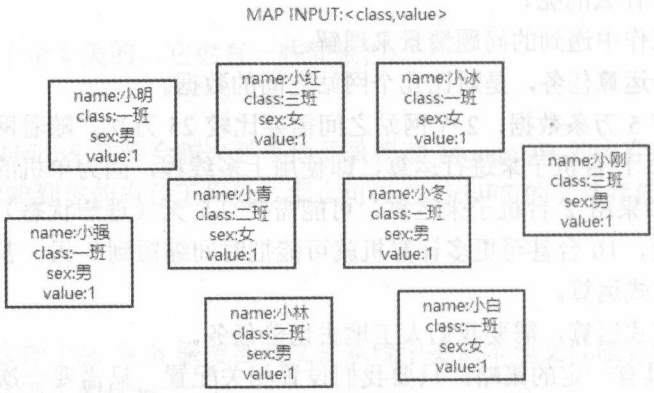

图 4-1　每个同学报出自己的班级和数量之前的情况

我们从图 4-1 中可以看出没经过 map 流程之前，我们的对象是乱序摆放的，而且每个对象都有 4 个属性值。但是经过 map 之后它们就会根据 map<key,value>的规则表明自己的身份，如图 4-2 所示。

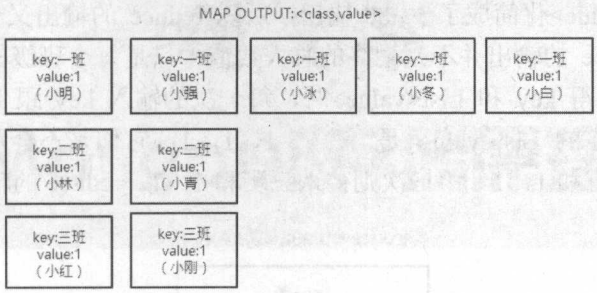

图 4-2　每个同学报出自己的班级和数量之后的情况

map 的输出只包含了 key 和 value 两个字段的值，这里也就是我们的 class 班级字段和 value 计算字段的值，name 名字字段是不被包含在 map 输出中的，图 4-2 只是为了让大家更好地理解每一个报数是从哪来的。

把每个班级的报数收集起来就是 shuffle 过程。map 的输出之后就到了 shuffle 流程，注意 shuffle 部分由 MongoDB 自行完成，我们在写 MapReduce 的时候只需要实现 map 和 reduce 即可。shuffle 会对 map 的输出部分进行洗牌，通过 key 进行分组会获得一个 List<value>，如图 4-3 所示。

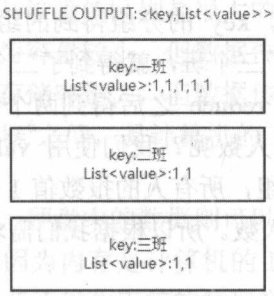

图 4-3　shuffle 洗牌后的输出

把集中上来的每个班级的数量进行相加求和化简就是 reduce 过程。shuffle 洗牌后的输出会作为 reduce 的输入，reduce 从 shuffle 的输出中获得 key 和 List<value>，如图 4-4 所示。

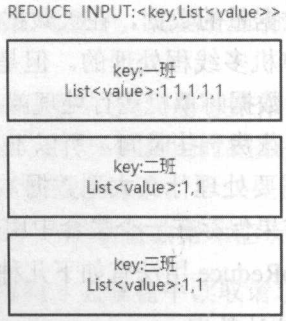

图 4-4　reduce 从 shuffle 的输出中获得 key 和 List<value>

获得输入之后，在 reduce 过程中进行逻辑业务操作。我们这里的业务就是进行计数求

和。求和之后 List<value>化简成了 value 输出，所以 reduce 的输出又变成了<key,value>。这里需要注意的是 reduce 的输出并不是最终的总人数值，只是每个班级的人数值。因为 reduce 从 shuffle 的输出中获得 key 和 List<value>时，每一次的输入 key 跟 List<value>是对应起来的，也就是说一班对应的 List<value>是 1，1，1，1，1。它们是不会跟二班的数值混合起来的，第二次才传入二班和 1，1。第三次则传入三班和 1，1。reduce 的输出如图 4-5 所示。

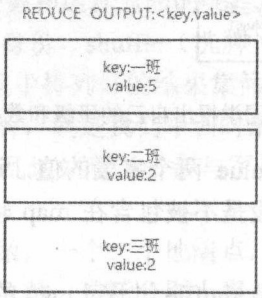

图 4-5　reduce 的输出

这三个键值就是我们这次 MapReduce 得到的值，最终的总人数值则需要我们再汇总一次，即 5+2+2=9。

从这个例子中我们也可以看出，key 的分组得到的组数就是 MapReduce 返回的键值个数。我们使用 class 作为 key 分为了三个班，就得到了三个班的人数键值。如果我们使用 sex 作为 key 则会分为男女两个分组，reduce 之后得到两个键值（男女人数）。那有没有办法 MapReduce 只返回一个键值就是总人数呢？我们使用 value 字段作为 key 就可以了。因为 value 的值都是 1，所以只有一个分组，所有人的报数值 1 都会在 reduce 时作为 List<value>输入，相加化简后就是我们需要的总人数。所以根据我们需求，MapReduce 的 key 和 value 的选择是很讲究技巧的。

4.3 MapReduce 应用场景

MapReduce 擅长的是处理大数据量的数据，在大数据的处理上，使用 MapReduce 分布式（多台计算机）处理肯定是快于单机多线程处理的。但是分布式运算交互时是需要花费一些时间的，也就是说处理小数据量的数据时单机运行处理就有可能快于 MapReduce。

因为分布式运算交互时是需要花费一些时间，所以千万不要把 MapReduce 使用在需要马上返回结果的环境中。如果我们需要处理什么大型数据，而且不需要马上返回，就可以使用 MapReduce 来运行，得到的最终结果保存在一个集合中用来实时查询。

笔者在工作中，常使用的 MapReduce 用法有如下几种：

（1）计数以及实现聚合函数统计数据。

（2）对数据进行分组简化或者构造自己想要的格式。

（3）根据条件进行数据筛选。

第 5 章

◂MongoDB存储原理▸

MongoDB 存取读写速度快，甚至可以用来当作缓存数据库。但是在使用过程中会发现 MongoDB 服务非常占内存，几乎是服务器有多少内存就会占用多少内存。为什么会出现这种情况呢？我们要从 MongoDB 的读写工作流程和对内存的使用方式说起。

5.1 存取工作流程

我们都知道一台计算机的存储分为内存存储和硬盘存储。

内存与硬盘都是存储器，内存与硬盘的区别是很大的。

内存是半导体材料制作，特点为容量较小，但数据传送速度较快。

硬盘是磁性材料制作，特点是存储容量大，但数据传送速度慢。

内存被架设在硬盘和高速缓存器[6]之间，是计算机的工作场所，硬盘用来存放暂时不用的数据。

内存中的数据会随断电而丢失，硬盘中的数据则可以长久保存。

内存与硬盘的联系非常密切，因为内存是计算机的工作场所，所以硬盘上的数据只有在装入内存后才能被处理。CPU 与硬盘不发生直接的数据交换，CPU 通过控制信号指挥硬盘工作。硬盘上的数据如果要使用，就得先通过 IO 操作，例如调用 read/write 函数装入内存。

由此可见操作，存取内存中的数据是比存取硬盘的数据更快。在很多情况下，磁盘 IO（特别是随机 IO）是系统的瓶颈。

基于这样的一种情况，MongoDB 在存取工作流程上有一个非常酷的设计决策，MongoDB 的所有数据实际上是存放在硬盘的，然后把部分或者全部要操作的数据通过内存映射存储引擎映射到内存中。

如果是读操作，直接从内存中取数据，如果是写操作，就会修改内存中对应的数据，然后就不需要管了。操作系统的虚拟内存管理器会定时把数据刷新保存到硬盘中。内存中的数据什么时候写到硬盘中，则是操作系统的事情了。

MongoDB 的存取工作流程区别于一般硬盘数据库在于两点：

读：一般硬盘数据库在需要数据时才去硬盘中读取请求数据，MongoDB 则是尽可能地放入内存中。

---

[6] 高速缓存器：容量比内存更小同时速度比内存更快的存储器，架设在内存和 CPU 之间。

写：一般硬盘数据库在有数据需要修改时会马上写入刷新到硬盘，MongoDB 只是修改内存中的数据就不管了，写入的数据会排队等待操作系统的定时刷新保存到硬盘。

MongoDB 的设计思路有两个好处：

（1）将什么时候调用 IO 操作写入硬盘这样的内存管理工作交给操作系统的虚拟内存管理器来完成，大大简化了 MongoDB 的工作。

（2）把随机的写操作转换成顺序的写操作，顺其自然地写入，而不是一有数据修改就调用 IO 操作去写入，这样减少了 IO 操作，避免了零碎的硬盘操作，大幅度提升性能。

但是这样的设计思路也有坏处：

如果 MongoDB 在内存中修改了数据，在数据刷新到硬盘之前，停电了或者系统宕机了，就会丢失数据了。针对这样的问题，MongoDB 设计了 Journal 模式，Journal 是服务器意外宕机的情况下，将数据库操作进行重演的日志。如果打开 Journal，默认情况下 MongoDB 每 100 毫秒（这是在数据文件和 Journal 文件处于同一磁盘卷上的情况，而如果数据文件和 Journal 文件不在同一磁盘卷上时，默认刷新输出时间是 30 毫秒）往 Journal 文件中 flush 一次数据，那么即使断电也只会丢失 100ms 的数据，这对大多数应用来说都可以容忍了。从版本 1.9.2+，MongoDB 默认打开 Journal 功能，以确保数据安全。而且 Journal 的刷新时间是可以改变的，使用--journalCommitInterval 命令修改，范围是 2 ~ 300ms。值越低，刷新输出频率越高，数据安全度也就越高，但磁盘性能上的开销也更高。

MongoDB 存取工作流程的实现关键在于通过内存映射存储引擎把数据映射到内存中。我们下一节就来学习 MongoDB 的内存映射存储引擎。

5.2 存储引擎

存储引擎是 MongoDB 数据库的一个重要组成部分。它的主要职责就是负责管理数据如何存储在硬盘和内存中，以及使用内存的方式。不同的存储引擎对不同的应用需求有特别的优化。如某个存储引擎可以是专为高并发写设计的，而另一个则是为高压缩率设计从而达到节省磁盘空间的目标。

MongoDB 从最初版本一直到 2.6 版本都只支持一种基于内存映射技术的存储引擎，叫做 MMAP 存储引擎。

2.6 版本之后 MMAP 存储引擎优化更名为 MMAP V1 引擎。

到了 MongoDB 3.0 版本，为了通过不同的数据引擎来满足不同的数据需求以及考虑到未来更多的场景扩展，MongoDB 引入了可插拔的存储引擎 API，并在此基础上增加了 WiredTiger 引擎（3.0 版本 WiredTiger 引擎只限于 MongoDB 3.0 的 64 位版本），但 MMAP V1 引擎仍是默认的存储引擎。

从 MongoDB 3.2 版本开始，WiredTiger 成为 MongDB 默认的存储引擎，并且 MongoDB 新增了 In-Memory 存储引擎。MongoDB 3.2 版本支持的存储引擎有：WiredTiger、MMAPv1 和 In-Memory。

我们先来了解 MongoDB 目前支持的 MMAP、MMAPV1、WiredTiger 以及 In-Memory 存储引擎。

5.2.1　MMAP 引擎

MongoDB 最初使用的存储引擎是内存映射存储引擎，即 Memory Mapped Storage Engine，简称 MMAP。

MMAP 使用了操作系统底层提供的内存映射机制，把磁盘文件的一部分或全部数据直接映射到内存，这样磁盘文件中的数据位置就会在内存中有对应的地址空间，把磁盘 IO 操作转换成内存操作。这时对文件的读写可以直接用指针来做，而不需要 read/write 函数这些 IO 操作了。MongoDB 并没有将数据直接放入到物理内存，只有访问到这块数据时才会被操作系统以 Page 的方式交换到物理内存。MongoDB 将内存管理工作交给操作系统的虚拟内存管理器来完成，MongoDB 只负责做映射，什么时候把数据存入磁盘文件，什么时候从磁盘文件取出数据，都由操作系统来完成，这样就大大简化了 MongoDB 的工作。图 5-1 所示的就是 MMAP 的核心工作原理。

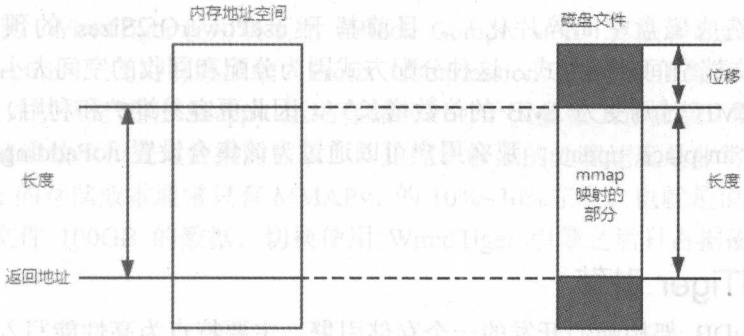

图 5-1　MMAP 把磁盘文件直接映射到内存

将内存管理工作交给操作系统的虚拟内存管理器也有坏处，就是你没有方法很方便地控制 MongoDB 占多大内存，事实上，MongoDB 会占用所有能用的内存，所以最好不要把别的服务和 MongoDB 放一起。

5.2.2　MMAPv1 引擎

MongoDB2.6 及以下版本用的是 MMAP 引擎，2.6 之后 MMAP 存储引擎优化更名为 MMAPv1 引擎。它在原理上是跟 MMAP 一样的，属于内存映射存储引擎，MMAP 版本的数据可以在线无缝迁移至 MMAPv1 版本。

MMAPv1 相对于 MMAP 有 2 个方面的改变：

（1）锁粒度由库级别锁提升为集合级别锁

到了 2.8 版本 MMAP v1 引擎增加了 collection 锁（collection level locking），在 MMAP 版本中，只提供了 DataBase 的锁（即当一个用户对一个 collection 进行操作时，其他的 collection 也被挂起），增加了 collection 锁之后进一步提高了 MongoDB 的并发性能。

（2）文档空间分配方式改变

在 MMAP 存储引擎中，文档按照写入顺序排列存储。如果文档更新后长度变长且原有存储位置后面没有足够的空间放下增长部分的数据，那么文档就要移动到文件中的其他位置。这种因更新导致的文档位置移动会严重降低写性能，因为一旦文档发生移动，集合中的所有索引都要同步修改文档新的存储位置。

MMAP 存储引擎为了减少这种情况的发生提供了两种文档空间分配方式：基于 paddingFactor（填充因子）的自适应分配方式和基于 usePowerOf2Sizes 的预分配方式，其中前者为默认方式。第一种方式会基于每个集合中文档更新历史计算文档更新的平均增长长度，然后在新文档插入或旧文档移动时填充一部分空间，如当前集合 paddingFactor 的值为 1.5，那么一个大小为 200 字节的文档插入时就会自动在文档后填充 100 个字节的空间。第二种方式则不考虑更新历史，直接为文档分配 2 的 N 次方大小的存储空间，如一个大小同样为 200 字节的文档插入时直接分配 256 个字节的空间。

MongoDB 3.0 版本中的 MMAPv1 抛弃了基于 paddingFactor 的自适应分配方式，因为这种方式看起来很智能，但是因为一个集合中的文档的大小不一，所以经过填充后的空间大小也不一样。如果集合上的更新操作很多，那么因为记录移动后导致的空闲空间会因为大小不一而难以重用（造成磁盘空间碎片化）。目前基于 usePowerOf2Sizes 的预分配方式成为 MMAPv1 默认的文档空间分配方式，这种分配方式因为分配和回收的空间大小都是 2 的 N 次方（当大小超过 2MB 时则变为 2MB 的倍数增长），因此更容易维护和利用。如果某个集合上只有 insert 或者 in-place update，那么用户可以通过为该集合设置 noPadding 标志位，关闭空间预分配。

5.2.3 WiredTiger 引擎

这是 BerkerlyDB 架构师们开发的一个存储引擎，主要特点为高性能写入、支持压缩和文档级锁。MongoDB3.2 已经将 WiredTiger 设置为默认的存储引擎。WiredTiger 和目前的 MMAP v1 存储引擎是 100%兼容的，用户不需要对程序做任何的修改便可切换直接使用。

WiredTiger 与 MMAP V1 的主要区别在于如下几点：

（1）支持多核 CPU、充分利用内存/芯片级别缓存。

（2）基于 B-TREE 及 LSM 算法。

（3）提供文档级锁（document-level concurrency control），大幅提升了大并发下的写负载。也就是说在同一时间，多个写操作能够修改同一个集合中的不同文档，这样写入的效率是很高的。如果是多个写操作修改同一个文档时，就需要按序列化方式执行写操作，比如一个文档正在被修改，其他写操作必须等待，直到在该文档上的写操作完成之后，其他写操作相互竞争，获胜的写操作在该文档上执行修改操作。

对于大多数读写操作，WiredTiger 使用乐观并发控制（optimistic concurrency control），只在 Global、Database 和 Collection 级别上使用意向锁（Intent Lock），如果 WiredTiger 检测到两个操作发生冲突时，导致 MongoDB 将其中一个操作重新执行，这个过程是系统自动完成的。

（4）支持文件压缩，包括三种压缩类型：

- 不压缩。
- 2.Snappy 压缩。默认的压缩方式， Snappy 是在谷歌内部生产环境中被许多项目使用的压缩库，包括 BigTable，MapReduce 和 RPC 等，压缩速度比 Zlib 快，但是压缩处理文件的大小会比 Zlib 大 20%~100%，Snappy 对于纯文本的压缩率为 1.5~1.7，对于 HTML 是 2~4，对于 JPEG、PNG 和其他已经压缩过的数据压缩率为 1.0。在 I7 i7 5500u 单核 CPU 测试中，压缩性能可在 200M/s~500M/s。
- 3.Zlib 压缩。Z1ib 是一个免费、通用、跨平台、不受任何法律阻碍的、无损的数据压缩开发库，相对于 Snappy 压缩，消耗 CPU 性能高、压缩速度慢，但是压缩效果好。

WiredTiger 引擎会压缩存储集合（Collection）和索引（Index），压缩减少 Disk 空间消耗，但是消耗额外的 CPU 执行数据压缩和解压缩的操作。

用户可以自己选择储存数据的压缩比例，MongoDB 3.0 提供最高达 80%的压缩率，不过压缩率越高数据处理的时间成本也越多，用户可以自行权衡应用。

默认情况下，WiredTiger 使用块压缩（Block Compression）算法和 Snappy 压缩库来压缩集合数据 Collections，使用前缀压缩（Prefix Compression）算法来压缩索引数据 Indexes，Journal 日志文件默认是使用 Snappy 压缩存储的。对于大多数工作负载，默认的压缩设置能够均衡数据存储的效率和处理数据的需求，即压缩和解压的处理速度是非常高的。

WiredTiger 的存储成本通常只有 MMAPv1 的 10%~30%左右。也就是说使用 MMAPv1 引擎保存在磁盘文件 100GB 的数据，切换使用 WiredTiger 引擎之后只占据磁盘 30GB 的空间了。

5.2.4　In-Memory

WiredTiger 和 MMAPv1 都用于持久化存储数据，也就是说数据最终会刷新保存到磁盘文件中，断电后数据也不会丢失。

In-Memory 存储引擎则将数据几乎全部存储在内存中，除了少量的元数据和诊断（Diagnostic）日志，In-Memory 存储引擎不会维护任何存储在磁盘上的数据（On-Disk Data），避免 Disk 的 IO 操作，减少数据查询的延迟。也就是说当你启用了 In-Memory 存储引擎，MongoDB 就变成了一个内存数据库。

内存数据库是指一种将全部内容存放在内存中，而非传统数据库那样存放在外部磁盘中的数据库。内存数据库指的是所有的数据访问控制都在内存中进行，这是与磁盘数据库相对而言的。磁盘数据库虽然也有一定的缓存机制，但都不能避免从外设到内存的交换，而这种交换过程对性能的损耗是致命的。由于内存的读写速度极快（双通道 DDR3-1333 可以达到 9300 MB/s，一般磁盘约 150 MB/s），随机访问时间更是以纳秒计（一般磁盘约 10 ms，双通道 DDR3-1333 可以达到 0.05 ms），所以这种数据库的读写性能很高，主要用在对性能要求极高的环境中，但是在服务器关闭后会立刻丢失全部储存的数据。

5.2.5 引擎的选择

不同的数据引擎满足不同的数据需求和应用场景。我们在上面的学习也了解到了MongoDB 现有版本的三种引擎 In-Memory、WiredTiger 和 MMAPv1。WiredTiger 作为MongoDB3.2 版本默认的存储引擎，一般来说，WiredTiger 会对大部分应用场景提供更好的性能表现，功能也比 MMAPv1 强大。所以对于正常使用场景 WiredTiger 是比较好的选择。In-Memory 因为它的数据是保存在内存中，断电后会消失，所以适用于 MongoDB 做缓存服务的场景。

5.2.6 未来的引擎

其他一些正在考虑支持的存储引擎（预计 3.2 版本之后会增加）：

（1）RocksDB：Facebook 开发的优化写操作的一个存储引擎。

（2）TokuFT：Tokutek 开发。这个引擎也在 MySQL (TokuDB) 和 TokuMX 下使用。

（3）FusionIO：跳过文件系统和 OS 直接访问 FushionIO 存储介质，提供高效读写。

第 6 章

◂ 了解MongoDB复制集 ▸

6.1 复制集简介

可以集群部署多个 MongoDB 服务器是 MongoDB 数据库的特点之一。集群部署 MongoDB 有什么好处？可以进行复制是集群部署带来的好处之一。MongoDB 复制是 MongoDB 自动将数据同步到多个服务器的过程，设置好策略之后免去了人工操作。

复制提供了数据的冗余备份，并在多个服务器上存储数据副本，提高了数据的可用性，并保证数据的安全性。有了复制，我们就可以从硬件故障和服务中断中恢复数据。

MongoDB 的复制也就是为数据实现了狡兔三窟。做过数据库管理员的都知道数据的重要性，数据错误和数据丢失都容易导致更严重的问题，尤其是在金融行业和电商领域。MongoDB 经过复制之后在多个服务器都会有数据的冗余，防止数据的丢失。所以强烈建议在生产环境中使用 MongoDB 的复制功能。

复制功能不仅可以用来应对故障（故障时切换数据库或者故障恢复），还可以用来做读扩展、热备份或者作为离线批处理的数据源。

我们下面就来了解复制功能的特点以及实现原理。

6.1.1 主从复制和副本集

MongoDB 提供了两种复制部署方案：主从复制（Master-Slave）和副本集（Replica Sets）（有些资料中也翻译成复制集）。两种方式的共同点在于都是只在一个主节点上进行写操作，然后写入的数据会异步[7]地同步到所有的从节点上。主从复制和副本集都使用了相同的复制机制。

那么它们的差别在哪里呢？副本集其实是 MongoDB1.6 版本才推出的功能，它是早期 MongoDB 版本中主从复制的优化方案。

主从复制只有一个主节点，至少有一个从节点，可以有多个从节点。它们的身份是在启动 MongoDB 数据库服务时就需要指定的。所有的从节点都会自动地去主节点获取最新数据，做到主从节点数据保持一致。注意主节点是不会去从节点上拿数据的，只会输出数据到从节点。理论上一个集群中可以有无数个从节点，但是这么多的从节点对主节点进行访问，

---

[7] 不影响 MongoDB 读写功能的情况下进行数据的同步，主从节点无须阻塞等待同步结束也能照常使用。

主节点会受不了。《MongoDB 权威指南》中有说不超过 12 个从节点的集群就可以运作良好，大家可以根据自己实际情况进行测试部署，主节点的机子性能应该会对支持的从节点个数有一定的影响。Master-Slave 主从复制集群如图 6-1 所示。

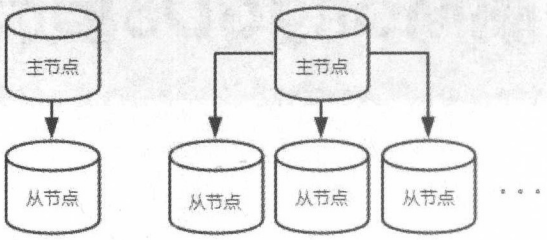

图 6-1　Master-Slave 主从复制集群

我们在生产环境下使用主从复制集群的过程中会发现一个比较明显的缺陷：当主节点出现故障，比如停电或者死机等情况发生时，整个 MongoDB 服务集群就不能正常运作了。需要人工地去处理这种情况，修复主节点之后再重启所有服务，当主节点一时难以修复时，我们也可以把其中一个从节点启动为主节点。在这个过程中就需要人工操作处理，而且需要停机操作，我们对外的服务会有一段空白时间，给网站和其他应用的用户造成影响，所以说主从复制集群的容灾性并不算太好。

为了解决主从复制集群的容灾性的问题，副本集应运而生。副本集是具有自动故障恢复功能的主从集群。副本集是对主从复制的一种完善，它跟主从集群最明显的区别就是副本集没有固定的主节点，也就是主节点的身份不需要我们去指明，而是整个集群自己会选举出一个主节点，当这个主节点不能正常工作时，又会另外选举出其他的节点作为主节点。副本集中总会有一个活跃节点（Primary）和一个或者多个备份节点（Secondary）。这样就大大提升了 MongoDB 服务集群的容灾性。在足够多的节点情况下，即使一两个节点不工作了，MongoDB 服务集群仍能正常提供数据库服务。

而且副本集的整个流程都是自动化的，我们只需要为副本集指定有哪些服务器作为节点，驱动程序就会自动去连接服务器，在当前活跃节点出故障后，自动提升备份节点为活跃节点。如果停电死机或者故障的节点来电或者启动之后，只要服务器地址没改变，副本集会自动连接它作为备份节点。副本集的自动化工作流程如图 6-2 所示。

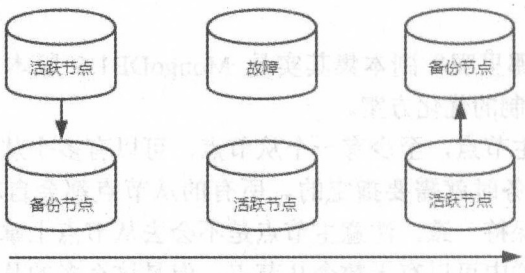

图 6-2　副本集的自动化工作流程

关于使用主从复制集群还是副本集，在新版本的 MongoDB 中都推荐使用副本集。只有一种情况需要选择主从复制，使用早期版本的 MongoDB 建立复制集群需要超过 11 个从节点时，因为早期版本的 MongoDB 副本集不能包含 12 个以上的成员。MongoDB 3.0 版本之后，副本集成员限额提升到了 50 个，而且在 MongoDB 版本升级过程中也对选举流程等做了优化，提出了仲裁节点的概念。仲裁节点（Arbiter）是副本集中的一个 MongoDB 实例，它并不保存数据，只负责选举时的投票，投票的原理我们下面会说。仲裁节点使用最小的资源，并且不要求硬件设备，建议不把 Arbiter 部署在同一个数据集节点中，否则如果数据集节点服务器挂了，Arbiter 也失效，就没起到它的作用。

Mongodb 的投票需要超过半数的节点投票给同一节点才能生效，把该节点提升为主节点，所以在某些情况下（尤其是偶数节点投票时）会导致节点们票数一致，会导致无法决定出主节点，卡在投票环节。

仲裁节点的意义就在于，当出现票数一致的情况，仲裁节点就被邀请跳出来判决，能让节点们投出结果。

仲裁节点不复制数据，仅参与投票。由于它没有访问的压力，比较空闲，因此不容易出故障。由于副本集出现故障的时候，存活的节点必须大于副本集节点总数的一半，否则无法选举主节点，整个副本集变为只读。因此，增加一个不容易出故障的仲裁节点，可以增加有效选票，降低整个副本集不可用的风险。仲裁节点可多于一个。副本集的结构如图 6-3 所示。

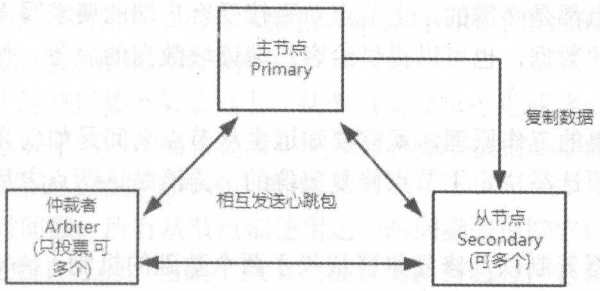

图 6-3　副本集的结构

MongoDB 官方推荐 MongoDB 集群的节点数量为奇数，主要在于副本集常常为分布式，当集群远程分布时可能位于不同的 IDC[8]。如果为偶数，可能出现每个 IDC 的节点数一样，这种情况下如果网络故障，那么每个 IDC 里的节点都无法选出主节点，导致全部不可用的情况发生。比如，节点数为 4，分处于 2 个 IDC，现在 IDC 之间的网络出现故障，每个 IDC 里的节点都没有大于 2，所以副本集没有主节点，变成只读。所以我们有两种方案，一是设置 MongoDB 集群的节点数量为奇数，二是当 MongoDB 集群的节点数量为偶数时，适当增加仲裁节点，增加集群的稳定性。

Mongodb 3.0 版本之后一个副本集集群中可设置 50 个成员，但只有 7 个投票成员（包括 primary），其余为非投票成员（Non-Voting Members）。非投票成员是复制集中数据的备份

[8] IDC 是对入驻企业、商户或网站服务器群托管的场所，也就是寄存服务器的地方。

副本，不参与投票，但可以被投票或成为主节点。

6.1.2 副本集的特点

根据上面对主从复制以及副本集的介绍，我们已经对副本集有了一定的了解，副本集的特点可以总结为以下几点：

（1）是多个节点组成的集群，保障数据的安全性。
（2）数据高可用性[9]。
（3）故障灾难重启服务器后自动恢复。
（4）任何从节点都可作为主节点，无须停机维护（如备份、重建索引、压缩等）。
（5）所有的写入操作都在主节点上进行，可分布式读取数据，实现读写隔离。

6.2 副本集工作原理

副本集有那么多的特点，它是如何实现的呢？本节我们来了解副本集的工作原理。了解副本集的工作原理有助于我们更好地应用它，并方便后期的优化和故障问题排查。

副本集中主要有三个角色：主节点、从节点、仲裁者。要组建副本集集群至少需要两个节点，主节点和从节点都是必需的，主节点负责接受客户端的请求写入数据等操作，从节点则负责复制主节点上的数据，也可以提供给客户端读取数据的服务。仲裁者则是辅助投票修复集群。

我们要了解副本集的工作原理，就需要知道主从节点之间是如何完成数据的复制的，以及集群是如何通过投票选举决定主节点修复集群的。弄清楚这两点之后我们就算了解副本集的工作原理了。

副本集要完成数据复制以及修复集群依赖于两个基础的机制：oplog（operation log，操作日志）和心跳（heartbeat）。oplog 让数据的复制成为可能，而“心跳”则监控节点的健康情况并触发故障转移。下面我们就看这些机制是如何工作的，看完之后你就能初步理解并预测副本集的行为了，能够预测副本集的行为对我们在故障诊断的时候尤其有帮助。

6.2.1 oplog（操作日志）

副本集中只有主节点会接受客户端的写入操作，也就是说从节点只要监控住主节点的写入操作，并且能够模仿主节点的写入操作就能完成一样的数据新增和更新，这样就实现了主节点到从节点的数据的复制，而不需要经常去完整地遍历对比主节点的数据。那从节点是如何能够获知主节点做了哪些操作呢？就是通过主节点的 oplog。oplog 是 MongoDB 复制的关键。oplog 是一个固定集合，位于每个复制节点的 local 数据库里，记录了所有对数据的变更操作。oplog 只记录改变了数据的操作，例如更新数据或者插入数据，读取查询这些操作是不

[9] 数据高可用性指副本集的数据很少存在关键数据不完整、缺失错误和误差。

会存储在 oplog 中的。新操作会自动替换旧的操作，以保证 oplog 不会超过预设的大小，oplog 中的每个文档都代表主节点上执行的一个操作。默认的 oplog 大小会随着安装 MongoDB 服务的环境变化。在 32 位系统上，oplog 默认是 50MB，在 64 位系统上，oplog 的默认大小是空余磁盘空间的 5%。oplog 的大小可以通过启动 MongoDB 服务时的参数来设置，这个我们在后面搭建副本集时会详细讲到，oplog 作为从节点与主节点保持数据同步的机制，数据库中的 oplog 如图 6-4 所示。

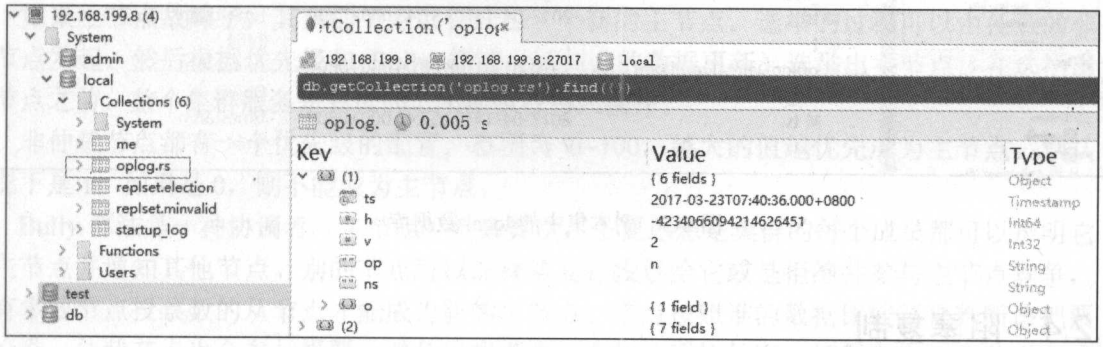

图 6-4　数据库中的 oplog

6.2.2　数据同步

在副本集中，每次客户端向主节点写入数据，就会自动向主节点的 oplog 里添加一个文档，其中包含了足够的信息来重现这次写操作。

一旦写操作文档被复制到某个从节点上，从节点就会执行重现这个写操作，然后从节点的 oplog 也会保存一条关于写入的操作记录。主节点有哪些数据变动的操作，从节点也同步做出这样的操作，从而保证了数据同步的一致性。

每个 oplog 都有时间戳，所有从节点都使用这个时间戳来追踪它们最后执行的写入操作记录。从节点是定时更新自己的，当某个从节点准备更新自己时，它会做三件事：首先，查看自己 oplog 里最后一条的时间戳；其次，查询主节点 oplog 里所有大于此时间戳的文档；最后，把那些文档应用到自己库里，并添加写操作文档到自己的 oplog 里。

从节点第一次启动时，会对主节点的数据进行一次完整的同步。同步时从节点会复制主节点上的每个库和文档（除了 local 数据库）。

同步完成之后，从节点就会开始查询主节点的 oplog 并执行里面记录的操作。这就是副本集数据复制的原理。

6.2.3　复制状态和本地数据库

除了 oplog 操作记录之外，主从节点还会存放复制状态。记录下主从节点交互连接的状态，记录同步参数时间戳和选举情况等。主从节点都会检查这些复制状态，以确保从节点能跟上主节点的数据更新。

复制状态的文档记录在本地数据库 local 中。主节点的 local 数据库的内容是不会被从节点复制的。如果有不想被从节点复制的文档，可以将它放在本地数据库 local 中。副本集中的

local 数据库如图 6-5 所示。

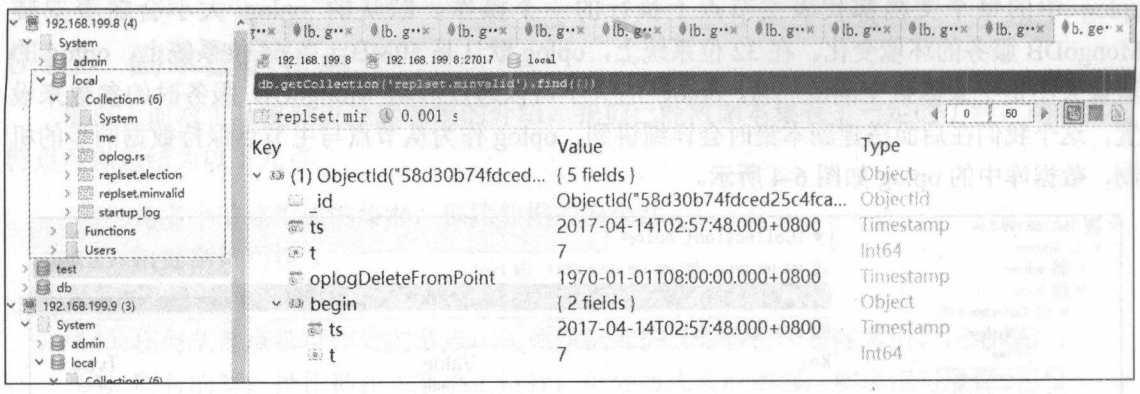

图 6-5　副本集中的 local 数据库

6.2.4　阻塞复制

从节点复制主节点的 oplog 并且执行它，对主节点来说是异步的，也就是主节点不需要等到从节点执行完同样的操作就可以继续下一个写入操作了。而当写入操作太快时，从节点的更新状态就有可能跟不上。如果从节点的操作已经被主节点落下很远，oplog 日志在从节点还没执行完，oplog 可能已经轮滚一圈了，从节点跟不上同步，复制就会停下，从节点需要重新做完整的同步。为了避免此种情况，尽量保证主节点的 oplog 足够大，能够存放相当长时间的操作记录。

还有一种方法就是暂时阻塞主节点的操作，以确保从节点能够跟上主节点的数据更新，这种方式就叫阻塞复制。阻塞复制是在主节点使用 getLastError 命令加参数“w”来确保数据的同步性。比如我们把“w”参数设置为 N[10]，运行 getLastError 命令后，主节点会进入阻塞状态，直到 N-1 个从节点复制了最新的写入操作为止。

阻塞复制会导致写操作明显变慢，尤其是“w”的值比较大时。实际上，对于重要操作，将其值为 2 或者 3 就能效率和安全兼备了。

6.2.5　心跳机制

副本集的心跳检测有助于发现故障进行自动选举和故障转移。默认情况下，每个副本集成员每两秒钟 ping 一次其他所有成员。这样一来，系统可以弄清自己的健康状况。

只要每个节点都保持健康且有应答，说明副本集就很正常，没有故障发生。如果哪个节点失去了响应，副本集就会采取相应的措施了。这时候副本集会去判断失去响应的是主节点还是从节点，如果是多个从节点中的某一个从节点，则副本集不做任何处理，只是等待从节点重新上线。如果是主节点挂掉了，则副本集就会开始进行选举了，选出新的主节点。

还有一种场景是副本集中主节点突然失去了其他大多数节点的心跳，主节点会把自己降

[10] N 为任意数字，但 N 小于 2 时，不会阻塞，当 N 等于 2 时，主节点会等到至少一个从节点复制了上个操作才会解除阻塞。

级为从节点。这是为了防止网络原因让主节点和其他从节点断开时，其他的从节点中推举出了一个新的主节点，而原来的主节点又没降级的话，当网络恢复之后，副本集就出来了两个主节点。如果客户端继续运行，就会对两个主节点都进行读写操作，肯定副本集就混乱了。所以，当主节点失去多数节点的心跳时（不够半数），必须降级为从节点。

6.2.6　选举机制

如果主节点故障了，其余的节点就会选出一个新的主节点。选举的过程可以由任意的非主节点发起，然后根据优先级和 Bully 算法（评判谁的数据更新）选举出主节点。在选举出主节点之前，整个集群服务是只读的，不能执行写入操作。

非仲裁节点都有一个优先级的配置，范围为 0~100，越大的值越优先成为主节点。默认情况下是 1；如果是 0，则不能成为主节点。

Bully 算法是一种协调者（主节点）竞选算法，主要思想是集群的每个成员都可以声明它是主节点并通知其他节点。别的节点可以选择接受并投票给它或是拒绝并参与主节点竞争，拥有多数节点投票数的从节点才能成为新的主节点。节点按照谁的数据比较新来判断该把票投给谁。仲裁节点也会参与投票，避免出来僵局。比如，网络故障，把偶数个集群节点分成两半时，详见 6.1.1 小节。

举例来说：当主节点故障之后，有资格成为主节点的从节点就会向其他节点发起一个选举提议，基本的意思就是“我觉得我能成为主节点，你觉得呢？”，而其他节点在收到选举提议后会判断下面三个条件：

（1）副本集中是否有其他节点已经是主节点了？
（2）自己的数据是否比请求成为主节点的从节点上的数据更新？
（3）副本集中其他节点的数据是否比请求成为主节点的从节点的数据更新？

如果上面三个条件中只要有一个条件成立，那么都会认为对方的提议不可行，于是返回一个消息给请求节点说“我觉得你成为主节点不合适，你退出选举吧!”，请求节点只要收到其他任何一个节点返回不合适，都会立刻退出选举，并将自己保持在从节点的角色；但是如果上面三个条件都是否定的，那么就会在返回包中回复说“我觉得你可以”，也就是把票投给这个请求节点，投票环节结束后就会进入选举的第二阶段。获得认可的请求节点会向其他节点发送一个确认的请求包，基本意思就是“我要宣布自己是主节点了，有人反对吗？”，如果在这次确认过程中其他节点都没人反对，那么请求节点就将自己升级为主节点，所有节点在 30 秒内不再进行其他选举投票决定。如果有节点在确认环节反对请求节点做主节点，那么请求节点在收到反对回复后，会保持自己的节点角色依然是从节点，然后等待下一次选举。

那优先级是如何影响到选举的呢？选举机制会尽最大的努力让优先级最高的节点成为主节点，即使副本集中已经选举出了比较稳定的、但优先级比较低的主节点。优先级比较低的节点会短暂地作为主节点运行一段时间，但不能一直作为主节点。也就是说，如果优先级比较高的节点在 Bully 算法投票中没有胜出，副本集运行一段时间后会继续发起选举，直到优先级最高的节点成为主节点为止。

由此可见，优先级的配置参数在选举机制中是很重要的，我们要么不设置，保持大家都是优先级 1 的公平状态，要么可以把性能比较好的几台服务器设置得优先级高一些。这个可以看大家的业务场景需求。详细的设置方法将在本书第二部分实战中讲解。

6.2.7 数据回滚

不论哪一个从节点升级为主节点，新的主节点的数据都被认定为 MongoDB 服务副本集的最新数据，对其他节点（特别是原来的主节点）的操作都会回滚，即使之前故障的主节点恢复工作作为从节点加入集群之后。为了完成回滚，所有节点连接新的主节点后要重新同步。这些节点会查看自己的 oplog，找出其中新主节点中没有执行过的操作，然后向新主节点请求这些操作影响的文档的数据样本，替换掉自己的异常样本。正在执行重新同步的、之前故障的主节点被视为恢复中，在完成这个过程之前不能成为主节点的候选者。

第 7 章

◂ 了解MongoDB分片 ▸

我们在上一章学习了 MongoDB 的副本集集群，已经初步体会到了集群的优势。分片则是 MongoDB 支持的另一种集群功能。MongoDB 能够实现分布式数据库服务，很大程度上得益于分片机制。由此可见它是多么重要，它是区别于其他传统数据库的一种重要的、具有代表性的功能。本章我们就来学习了解分片，以及它的工作原理，讨论它适合应用于哪些场景。

7.1 分片的简介

分片（sharding）是将数据进行拆分，将它们分散地保存在不同的机器上的过程。MongoDB 实现了自动分片功能，能够自动地切换数据和做负载均衡。

为什么会诞生分片这种功能的需求呢？我们可以结合工作中的生产环境来思考。比如我们启动了一个 MongoDB 服务，放置在一台服务器中，作为对 Web 网站的数据库服务。随着 Web 网站的用户增加、数据量的增长，以及 Web 网站对读写吞吐量的要求越来越高，普通的服务器性能就不够用了，普通服务器可能无法分配足够的内存或者没有足够的 CPU 核数来有效处理工作负荷。这个时候一般情况下我们需要提高服务器的配置，例如使用更强大的 CPU、增加 RAM 或增加存储空间的量。如果是使用了副本集的 MongoDB 服务集群，每个从节点都完整地克隆了主节点的数据，我们在提高主节点配置的同时，还需要提高从节点的配置。我们都知道配置越往上提高，价格是成几何倍数地增长的，而且服务器受到当前的科技限制，是无法有效地满足性能需求的。也就是说当数据量达到一定程度时，目前市场上能买到的最好性能的单个服务器也会不堪重负的，有钱也买不到更好的服务器了。这就是垂直的方式解决系统数据增长的困境。

聪明的 IT 从业者想到了应对的措施：既然垂直方向不能解决，我们能不能水平扩展。也就是我们使用集群去分担数据的压力，每一台服务器只复制一部分数据的管理，当数据量再增加时，我们就再追加普通的服务器即可。这样无论数据量多大，我们都能水平地扩张，每台普通服务器负责的数据量都是稳定和可控的，甚至能照常使用副本集功能。虽然每台普通服务器的速度和容量可能不高，但每台服务器处理总工作量的一部分，比一个单一的高速高容量的服务器处理总工作量的效率更高（是不是有点类似于多线程的处理原理），而且多个普通的服务器花费比单个高速高容量的服务器成本要低得多。

这种将数据库服务分布在多台服务器上的机制就叫分片。

分片的缺点是会增加基础设施的复杂性和部署维护的难度。这就需要结合我们工作的业

务场景来考虑到底使不使用分片了。

7.2 分片的工作原理

分片并不是 MongoDB 独有的机制，MongoDB 没有诞生之前，人们还在使用 SQL 数据库的时候，就已经在手动实现分片了。当时的手动分片如何实现的？主要需要人工地去设计一套分片的逻辑，比如说一个用户表 Users，很多个 user，我们在应用程序把数据保存到 Users 表时先进行 id 的判断，如果是奇数就存在这一台服务器，如果是偶数就存在另一台服务器，取的时候也根据同样的规则去取出数据；或者根据取余数的方式来划分。总之，划分的方法多种多样，需要人工去设置，有时候甚至需要一台专门的数据库服务器来记录这些存储的规则，查询数据时先查一遍存储规则，再根据规则去取出我们所需要的数据。人工的分片大多依赖于应用程序的代码来实现。

MongoDB 则实现了自动分片。分片是 MongoDB 数据库的核心内容，它内置了几种分片逻辑，用户不再需要自己去设计外置分片方案和框架，也不需要我们在应用程序上做处理，也就是说，在数据库需要启用分片框架时，或者增加新的分片节点时，我们的应用程序代码几乎不需要改动。MongoDB 是如何实现这一点的呢？我们来看看它的分片的详细工作原理：分片机制的重点是数据的分流、块的拆分和块的迁移。我们通过三个小节来了解它们是如何实现的。

7.2.1 数据分流

数据分流是实现分片的很重要的一部分。我们之前已经讲过 MongoDB 区别于手动分片的核心是内置了几种数据分流存放的策略。

目前 MongoDB3.2 版本有哈希分片、区间分片和标签分片三种策略。

MongoDB 在进行分片之前都需要设置一个片键（shard key）。这个片键可以是集合文档中的某个字段或者几个字段组成一个复合片键，MongoDB 分片集群数据库服务是不允许插入没有片键的文档的。然后，MongoDB 根据我们启用的策略来使用片键的值进行对数据进行分配，这样就完成了我们对数据的分流。接着，我们来看看现有的三种分片策略。

1. 区间分片

区间分片是根据片键的值的区域来把数据划分成数据块的，这也是早期的 MongoDB 分片的策略。因为在 MongoDB 中数据类型之间是有严格的次序的，所以 MongoDB 能够对片键的值进行一个排序。类型的先后次序如下：

null<数字<字符串<对象<数组<二进制数据<objectId<布尔值<日期<正则表达式

同类型也可以进行排序，而且同类型的排序与我们熟悉和期望的排序可能相同：比如数字类型 5<6，或者字符串类型“a”<“b”。

在排序的基础上，MongoDB 就能获取到片键的最大值和最小值了，然后 MongoDB 会根据目前已有的片键和目前有多少台服务器参与分片来进行数据分配。

例子如下：

我们设置集合中文档的 x 字段作为片键。MongoDB 会对 x 片键进行统计，得到最小值和最大值，如果目前我们有三个分片服务器，MongoDB 可能会把区间分成三块：最小值到 10、10 到 20 以及 20 到最大值（这里只是举例，MongoDB 不一定会这样分配，但是原理是一样的）。然后，当有新的文档要写入 MongoDB 数据库时，MongoDB 就会根据这个区间把属于不同区域的文档分配到不同的分片服务器上。在使用区间分片时，拥有相近片键的文档很可能存储在同一个数据块中，因此也会存储在同一个分片中。这就是区间分片的原理。区间分片如图 7-1 所示。

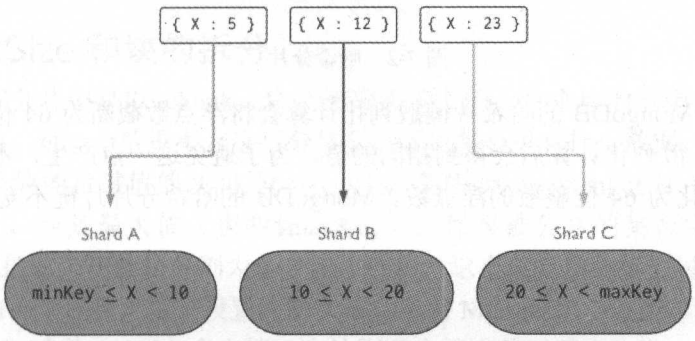

图 7-1　区间分片

2. 哈希分片

MongoDB 2.4 版本及以上才支持哈希分片，哈希分片仍然是基于区间分片，只是将提供的片键散列成一个非常大的长整型作为最终的片键。当我们选定片键之后，MongoDB 会自动将片键进行散列计算之后再进行区间的划分。普通的区间分片可以支持复合片键，但是哈希分片只支持单个字段作为片键。

哈希片键最大的好处就是保证数据在各个节点分布基本均匀。

我们来对比一下基于区间分片和哈希分片有什么不同。区间分片方式提供了更高效的范围查询，给定一个片键的范围，分发路由可以很简单地确定哪个数据块存储了请求需要的数据，并将请求转发到相应的分片中，不需要请求所有的分片服务器。不过，区间分片会导致数据在不同分片上的不均衡，有时候，带来的消极作用会大于查询性能的积极作用。比如，如果片键所在的字段是线性增长的，例如数字型自增 id 或者时间戳，一定时间内的所有请求都会落到某个固定的数据块中，最终导致分布在同一个分片中。在这种情况下，一小部分分片服务器承载了集群大部分的数据，系统并不能很好地进行扩展。

哈希分片方式则是以查询性能的损失为代价，保证了集群中数据的均衡。哈希值的随机性使数据随机分布在每个数据块中，因此也随机分布在不同分片中。但是也正由于随机性，一个范围查询很难确定应该请求哪些分片，通常为了返回需要的结果，需要请求所有分片服务器。哈希分片如图 7-2 所示。

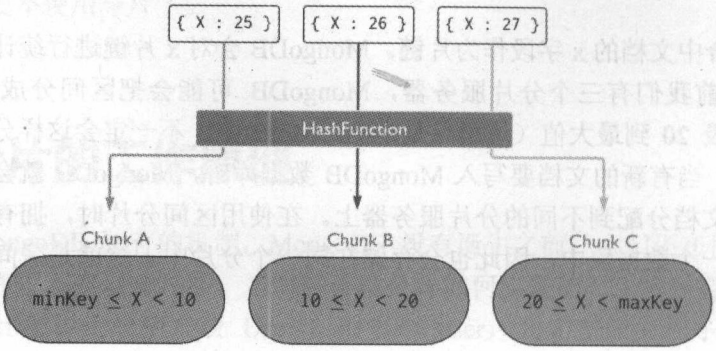

图 7-2　哈希分片

需要注意的是 MongoDB 的哈希片键散列化计算会将浮点数截断为 64 位整数，比如，对于 2.3、2.2 和 2.9，散列化计算后会得到相同的值，为了避免这一点产生，不要在哈希索引中使用不能可靠地转化为 64 位整数的浮点数。MongoDB 的哈希分片片键不支持大于 253 的浮点数。

3. 标签分片

MongoDB 可以手工设置数据保存在哪个分片节点服务器上，这非常有用，主要就是通过标签分片策略来实现的。标签分片是 MongoDB 2.2 版本中引入的新特性，此特性支持人为控制数据的分片方式，从而使数据存储到合适的分片节点上。具体的做法是通过对分片节点打 tag 标识，再将片键按范围对应到这些标识上。

例子如下：我们有三个分片节点，定义了两个 tag 标识，A 的 tag 标识表示片键 x 的区间是 1~10，B 的 tag 标识表示片键 x 的区间是 10~20。然后把节点 Alpha 和 Beta 都打上 A 标签，节点 Beta 打上 B 标签。最后有数据要写入 MongoDB 数据库时，我们可以看到片键 x 在 1~10 范围内的文档数据只会分配到带有 A 标签的节点，也就是 Alpha 或者 Beta；片键 x 在 10~20 范围内的文档数据只会分配到带有 B 标签的节点上，也就是 Beta；而如果 x 片键的值没有被包含在已有的 tag 的范围内，那么它就可能任意分配到任意分片节点中。标签分片如图 7-3 所示。

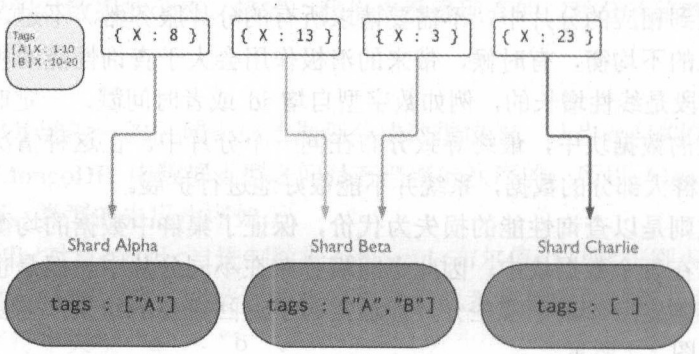

图 7-3　标签分片

生产环境中有哪些场景比较常用标签分片呢？比如说对于一些日志集合，我们只希望运行在配置比较低的节点服务器上，我们就可以指定日志集合文档片键所在区间为一个标签 tag，然后把这个 tag 打在某个配置比较低的节点服务器上即可。

还有一个例子是用户集合。一般来说用户集合是比较小的，如果我们对它进行分片就会把用户数据分散在各个节点中，造成查询上的性能消耗。对于用户这种比较小的集合，我们也可以对它进行标签分片，这样用户集合就可以存储到一台指定的分片节点服务器上，接下来就可以在内存中直接存取所有的用户数据，这样速度就会快很多了。这就是标签分片的原理以及应用。

7.2.2　chunkSize 和块的拆分

数据分流后在分片中是以 chunk（块）的形式存在的。每个块的范围都是由片键起始值和终止值来标识的。对一个数据集合进行分片后，无论集合里有什么数据，MongoDB 都只会创建一个块。这个块的片键值的区间是(-∞，∞)，其中-∞是 MongoDB 可以表示的最小值（也叫$minKey），∞是最大值（也叫$maxKey）。如果被分片的集合中包含大量数据，MongoDB 会立刻把这个初始块分割为多个较小的块。这个就是块的拆分过程。chunkSize 是分片集群启动时的一个参数，用于设置块的大小。目前 MongoDB 3.4 版本默认块的最大尺寸 chunkSize 是 64MB 或者 100000 个文档，先达到哪个标准就以哪个为准。在向新的分片集群添加数据时，原始的块最终会达到某个阈值，触发块的拆分。这是一个简单的操作，基本是把原来的范围一分为二，这样就有了两个块，每个块都有相同数量的文档。

注意，块的拆分是逻辑操作，也就是只是修改 Config 配置数据库中记录的块的元数据，让一个块变成两个，而不会影响分片中文档的物理位置。详情可参考第 18 章“分片部署”有关测试块的拆分的功能。

7.2.3　平衡器和块的迁移

分片机制中三个重点：数据分流、块的拆分和块的迁移，我们已经了解了数据分流与块的拆分，本小节就来了解块的迁移。

我们通过上面章节的学习，也知道了不管是手动设计分片系统还是选择片键和策略后 MongoDB 自动去分片都避免不了一个难题，就是保证数据始终均匀分布。只有分片集群中的数据均匀分布，才能发挥出集群最大的性能。MongoDB 为了解决这个问题，设计了平衡器，用来完成数据的迁移尽量保证数据均匀分布。

数据的迁移是以块（chunk）为单位的，它是位于一个分片中一段连续的片键的范围。每个块的范围都是由起始值和终止值来标识的。

例如，用户集合被分片保存在两个分片节点中，它们的片键根据块的范围可以把用户数据分成很多个块。

它们只是逻辑上的分块，并不是物理上的。也就是说，块不表示磁盘上连续的文档，虽然每个单独的块都表示一段连续范围的数据；但这些块能出现在任意分片节点服务器中，这些块的数据也任意出现在分片节点的服务器中。

MongoDB 的分片集群是通过在分片中移动块来实现均衡的，我们称之为迁移，这是一个

真实的物理操作（也就是说磁盘上的数据文件也会被移动）。

迁移是由平衡器（balancer）的软件进程进行管理的。它的任务就是确保数据在各个分片节点服务器中保持均匀分布。平衡器通过跟踪各分片上块的数量，就能实现这个功能。什么时候平衡器会做一次数据迁移呢？通常来说当集群中拥有块最多的分片节点服务器与拥有块最少的分片的节点服务器相差的块数太大时，平衡器就会触发数据的迁移，做一次均衡处理。至于相差多少算是块数相差大，这个会根据总数据量的不同而变化，一般是相差 8 个块的时候就会触发。在数据迁移过程中，块会从块较多的分片节点服务器迁移到块较少的分片上，直到分片节点服务器上的块数大致相等为止。

这就是分片机制的第 3 个要点：数据迁移和平衡器的工作原理。到这里我们已经对分片机制的工作原理做了一个大概的了解了。主要还是两点：数据的拆分以及数据迁移。

7.3 分片的应用场景

我们在 7.1 节中已经讲述了当应用程序数据量和请求量太大时垂直扩展的困境，也在 7.2 小节“分片工作原理”中了解了分片的优势，那么在实际工作中我们什么时候考虑使用分片呢？

一般一开始时我们使用 MongoDB 单个实例服务器即可，到后面遇到性能瓶颈之后再部署分片。

建议在以下场景出现时再考虑应用分片：

（1）当请求量巨大，出现单个 MongoDB 实例服务器不能满足读写数据的性能需求时。

（2）当数据量太大出现本地磁盘不足时。

（3）想要将大量数据放在内存中提高性能，而单个 MongoDB 实例服务器内存不足时。

第二部分

管理与开发入门篇

我们在第一部分理论部分已经对 MongoDB 进行了全面的了解，完成了对 MongoDB 远观的过程。从第二部分开始我们就要开始进入实践部分了，希望读者也能跟着步骤根据自己的计算机的情况一起来进行操作，近距离地接触 MongoDB。当第二部分结束之后，你会发现你已经学会使用 MongoDB 了。

第 8 章

◂ 安装MongoDB ▸

8.1 版本和平台的选择

8.1.1 版本的选择

MongoDB 从最初的版本发展到现在的 MongoDB3.4 版本，对于初学者来说，众多的可用版本会让你难于决定使用哪个版本。

在给出安装步骤之前，有必要说明下 MongoDB 的版本号。如果版本选得不对，在使用过程中就会遇到各种各样的问题。

MongoDB 的版本分为稳定版和开发版。开发版表示仍在开发中的版本，其中包含许多修改，包括一些新的功能特性，尽管还未得到充分的测试，但仍发布出来给开发者用于测试或者做尝试。稳定版本则是经过充分测试的版本，是稳定和可靠的，但通常包含的功能特性会少一些。从这里我们就知道了生产环境中是不建议使用开发版本的，最好使用稳定版，否则就会遇到一些无法预估的情况。

那么稳定版本和开发版本怎么区分呢？我们可以从版本号来区分。版本号一般有 3 位数，第一位数字是主版本号，代表着重大版本的更新时主版本号才会变动。第二位数字则用来代表是开发版还是稳定版的更新。当第二位数字是偶数时，说明它是稳定版本；当第二位数字是奇数时，它就是开发版。第三位数字表示修订号，用于解决缺陷和修复 bug 等。举例来说：3.4.2 是稳定版本，3.3.2 是开发版本。

通过对版本的了解，我们在生产环境中，应该尽量选择最新的稳定版本。最新的稳定版本才是可靠的，同时也是 bug 相对来说较少的。

在编写本书的时候最新的版本是 3.4.2，所以我们在之后的操作中会以 3.4.2 为例子。即使版本更新后，本书中大部分的操作以及功能都还是适用的（一般新版本的发布会尽量兼容老版本），只需要额外关注新版本更新了哪些功能、修复了哪些 bug 以及丢弃了哪些用法即可。

对于每个版本更新了哪些功能以及修复了哪些 bug，可以在官网中查看发布日志 release-notes。例如 MongoDB3.4 版本的发布日志路径是 https://docs.mongodb.com/manual/release-notes/3.4/。

8.1.2 平台的选择

MongoDB 是一个跨平台的数据库，也就是说它可以运行在不同的操作系统上为我们提供数据库服务。所以对于平台的选择，只需要根据我们打算安装 MongoDB 数据库服务的计算机的操作系统来选择平台安装包即可。

目前 MongoDB 官网提供了 4 个操作平台的安装包，分别是 Windows、Linux、Mac OSX[11]和 Solaris。

很多读者会好奇不同操作系统之间的 MongoDB 性能会不会有大的区别，不同的操作系统中 MongoDB 之间的性能对比是一个比较复杂的过程，需要考虑硬件性能、网络情况、操作系统的版本和位数，以及 MongoDB 的版本，所以不能轻易地断言 MongoDB 在哪个操作系统中性能更好，只能具体情况具体分析，有兴趣的读者可以根据自己的服务器情况测试一下。

那是不是意味着 MongoDB 安装在哪一种操作系统上没有最优选择了呢？ 笔者建议生产环境使用 Linux 系统的计算机作为 MongoDB 数据库服务器。因为作为数据库来说，当不同的操作系统提供的性能相差不是很明显的情况下，我们更多地需要考虑其他方面的因素。

首先是大环境。Linux 是开源的免费的操作系统，作为服务器是一种趋势，因为大家都在用 Linux 作为服务器，很多坑就会有人先踩过了，所以当我们遇到问题时，更容易得到文档和论坛的支持。

其次是稳定性。由于文件系统的区别以及内存管理方式的差异，Linux 系统可以长时间地运作，不需要关机重启也不会出现卡顿的情况，同时在高负载的情况下表现较为稳定。

所以，笔者建议生产环境使用 Linux 系统的计算机作为 MongoDB 数据库服务器。测试开发环境下，可以使用其他操作系统的计算机作为 MongoDB 数据库服务器。

Windows 和 Mac OSX 都是比较日常生活中常见的操作系统，Solaris 在 Solaris10 之前一直是私有系统，Solaris10 之后才开始开源面向公众，使用的场景比较少，Solaris 被认为是 UNIX 操作系统的衍生版本之一，所以安装的方式与 Linux 有点类似。 所以我们在接下来会详细讲解 MongoDB 在 Windows、Linux 以及 Mac OSX 操作系统上的安装，有使用 Solaris 系统的开发者，可以参考 MongoDB 在 Linux 系统上的安装或者查阅官网。

8.1.3 32 位和 64 位

MongoDB 的安装包分别是针对不同的操作系统的位数编译的，32 位和 64 位版本的数据库功能是相同的，唯一的区别是针对 32 位系统的版本将单个实例的数据集总大小限制在 2GB 左右。32 位的计算机系统受地址空间的限制，所以单个实例最大数据空间仅为 2GB，64 位基本无限制（128T），故建议使用 64 位计算机部署 MongoDB。MongoDB 的 32 位版本只用于在 32 位的系统上部署测试和开发，不能在正式生产环境中使用。

---

[11] 苹果公司在旧金山举办了 2016 年的 WWDC 开发者大会，在大会上宣布将 OS X 更名为 macOS，最新的版本名称为 macOS Sierra。

8.2 Windows 系统安装 MongoDB

本节记录 Windows 环境的安装的详细步骤。

8.2.1 查看安装环境

在 Windows 系统中，右击“我的电脑”图标，单击“属性”，能够看到系统的环境。我的计算机系统是： Windows 10，64 位，如图 8-1 所示。

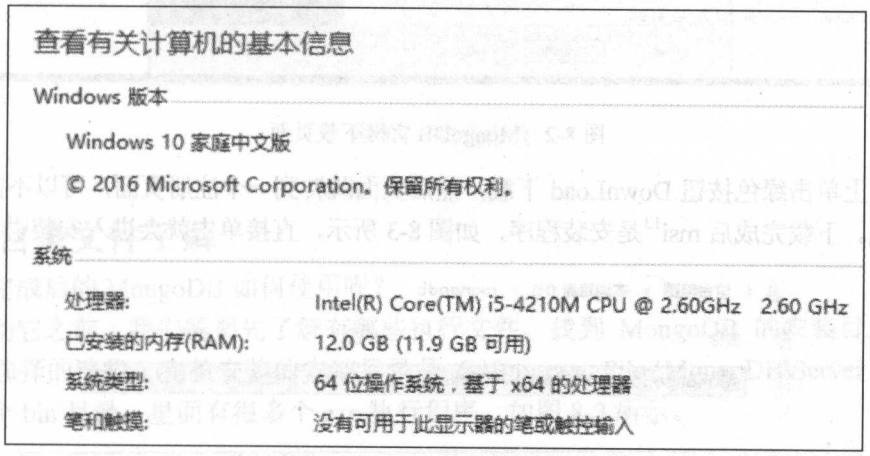

图 8-1　查看属性

读者可以根据自己的计算机情况来进行相应的 MongoDB 安装包选择，一般来说 MongoDB 的 Windows 安装包在 Windows 系统中是通用的，需要注意的是 32 位还是 64 位的系统。在 32 位的计算机上运行针对 64 位系统的安装包会报错的。老版本的 MongoDB 有支持 32 位系统的 x86 安装包，但是在最新的 MongoDB 3.4 版本中，Windows 的安装包只有支持 64 位系统的 x64 安装包了。

8.2.2 安装步骤

MongoDB 官方下载地址：http://www.mongodb.org/downloads。

MongoDB 官网下载页面如图 8-2 所示，下载地址提供了各种平台的版本。我这里选择的是社区版本[12]：Windows 平台下的 Windows Server 2008 R2 64-bit and later, with SSL support x64。

[12] 在下载界面中我们可以看到有很多版本可供选择，包括一些商业版本和管理工具，初学者和初步使用可以先用社区版本即可。商业版本目前不收费，与社区版本的主要区别在于安全认证等方面会多一些支持。

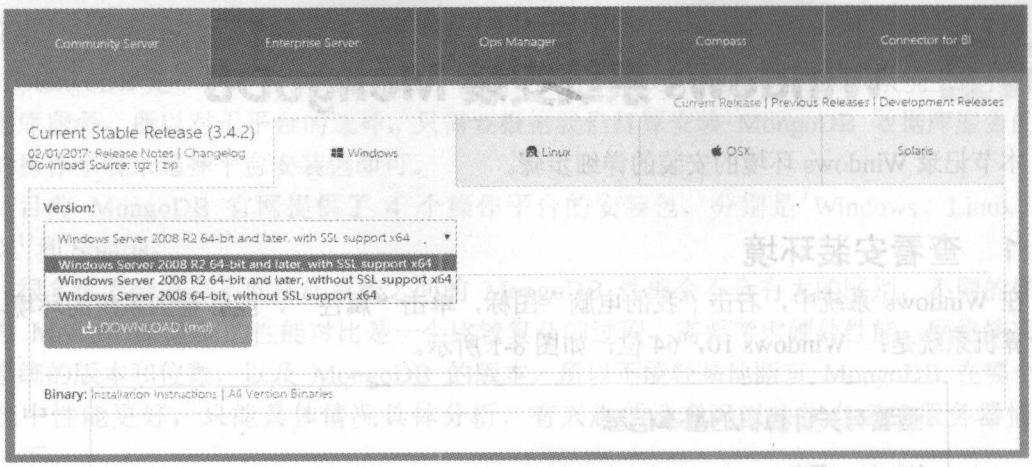

图 8-2　MongoDB 官网下载页面

在页面上单击绿色按钮 DownLoad 下载，然后页面跳转到一个注册页面，可以不需要注册也会开始下载。下载完成后 msi[13]是安装程序，如图 8-3 所示，直接单击就会进入安装引导。

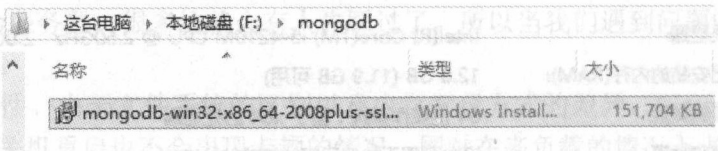

图 8-3　下载完成的 MongoDB 安装程序

跟着安装引导安装即可。需要注意的是，在过程中会让选择完整安装还是定制安装。定制安装可以取消部分不需要的功能和自定义选择安装路径，完整版会安装在默认路径 C:\Program Files\MongoDB\Server 目录下。

安装过程的主要步骤如图 8-4、图 8-5、图 8-6 所示。

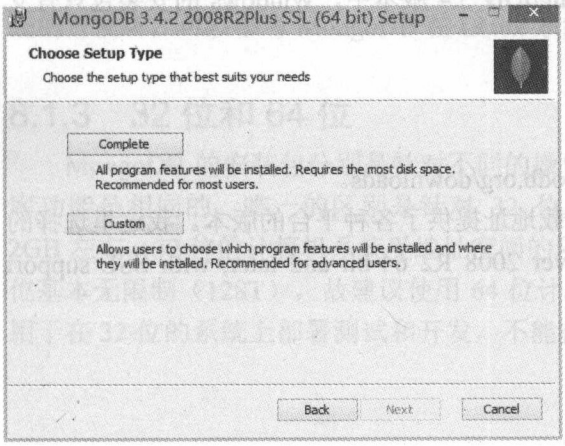

图 8-4　完整安装和定制安装的选择

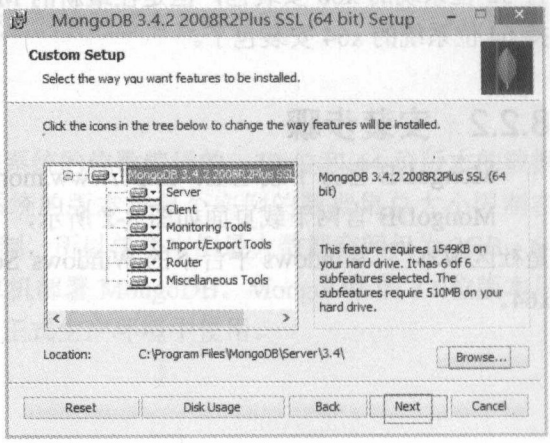

图 8-5　定制安装

[13] 现在的 Windows 版本的安装包是 msi 安装程序，老版本的 MongoDB 会是压缩包，直接解压就可以用。

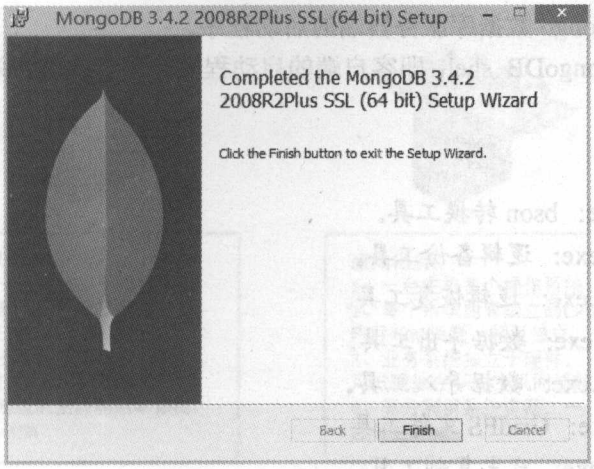

图 8-6　安装完成

8.2.3　目录文件了解

安装完成后的 MongoDB 如何使用呢？

在启动它之前，我们需要先了解有哪些执行文件。找到 MongoDB 的安装目录（定制安装是自己选择的路径，完整安装的安装目录是 C:\Program Files\MongoDB\Server），安装目录中有一个 bin 目录，里面有很多个 exe 执行程序，如图 8-7 所示。

图 8-7　安装目录中的文件

其中有两个重要的启动程序：mongod.exe 和 mongo.exe。

mongod.exe 是 mongo 数据库服务器端的启动程序。

mongo.exe 是 MongoDB shell 即客户端的启动程序，可以在客户端里对数据库做增删改查等命令操作。

其他文件包括：

- bsondump.exe：bson 转换工具。
- mongodump.exe：逻辑备份工具。
- mongorestore.exe：逻辑恢复工具。
- mongoexport.exe：数据导出工具。
- mongoimport.exe：数据导入工具。
- mongofiles.exe：GridFS 文件工具。
- mongooplog.exe：日志复制工具。
- mongoperf.exe：性能检查工具。
- mongos.exe：分片路由工具。
- mongostat.exe：状态监控工具。
- mongotop.exe：读写监控工具。

对于这些工具我们暂时只需要大概知道是什么就可以了，在后面的章节我们学习到相关命令的时候就会完全弄懂了。

8.3 Linux 系统安装 MongoDB

本小节介绍 Linux 环境下安装的详细步骤。

8.3.1 虚拟机简介

我们在 8.1.2 小节“平台的选择”中已经讲述了生产环境 MongoDB 最好运行在 Linux 系统中，但是日常生活中直接使用 Linux 系统的人还是比较少的，日常生活中计算机的操作系统还是 Windows 系统或者 OS 系统占据相当大的比例，那我们怎么才能在不重新买一台计算机的情况下学习 Linux 系统安装 MongoDB 呢？答案是使用虚拟机。

虚拟机（Virtual Machine）指通过软件模拟的具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统。

虚拟机通过生成操作系统的全新虚拟镜像来实现，它具有和真实系统完全一样的功能，进入虚拟机后，所有操作都是在这个全新的独立的虚拟系统里面进行，可以独立安装运行软件、保存数据、拥有自己的独立桌面，不会对真实系统产生任何影响，而且我们能够在真实系统与虚拟镜像之间灵活切换。虚拟机实现一台计算机多系统如图 8-8 所示。