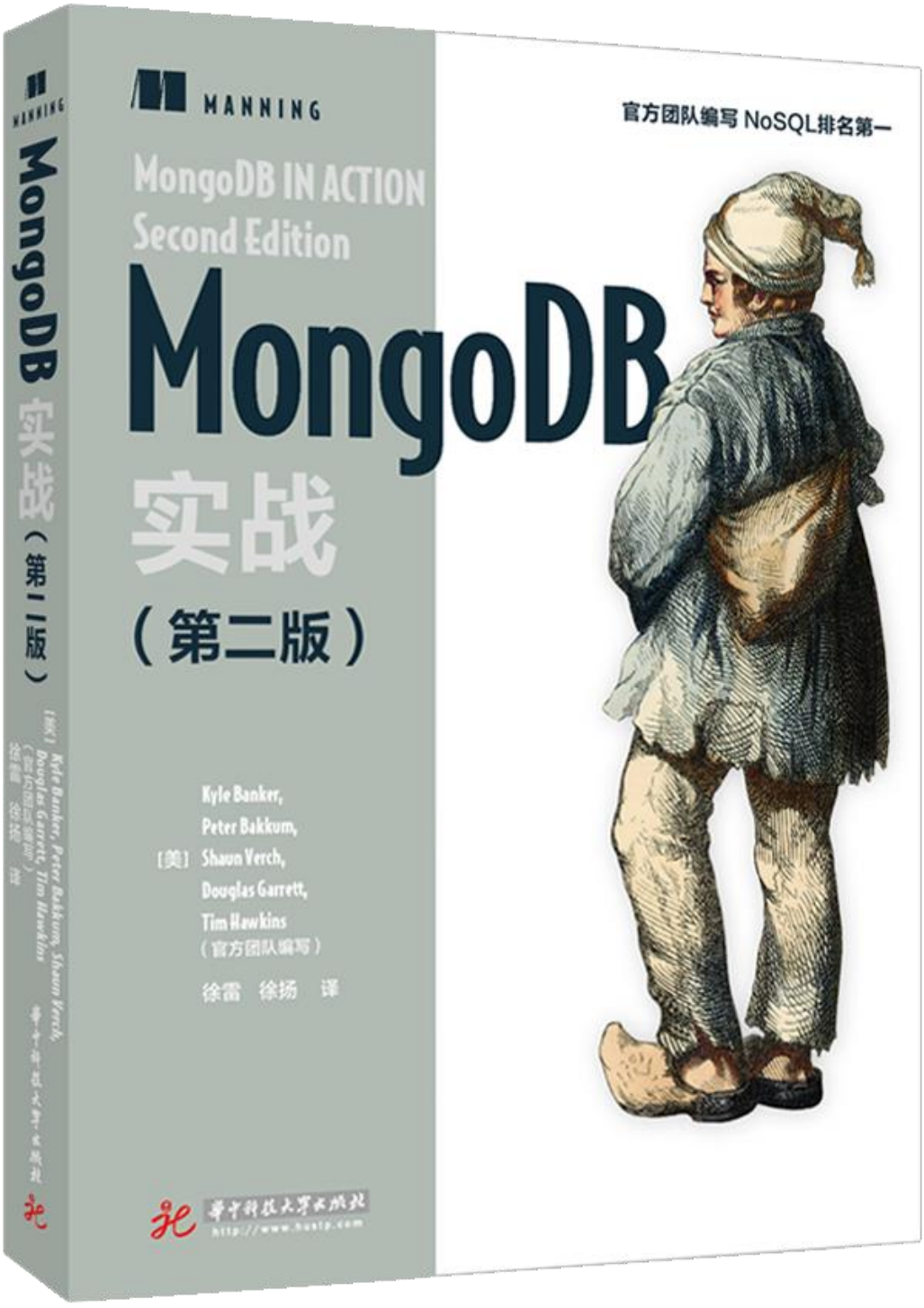
MANNING
MongoDB IN ACTION
Second Edition
MongoDB
实战
（第二版）
官方团队编写 NoSQL排名第一
Kyle Banker,
Peter Bakkum,
[美] Shaun Verch,
Douglas Garrett,
Tim Hawkins
（官方团队编写）
徐雷　徐扬　译
MongoDB 实战（第二版）
华中科技大学出版社
http://www.hustp.com

第一部分　Mongodb 实战入门

Getting started

1.**MongoDB优点**

感谢各位读者支持厚爱，本京东等多次断货。本次再版感谢热心读者的反馈建议。MongoDB是最流行的NoSQL数据库！在NoSQL中排名第一！高并发、高性能、灵活数据模型、易伸缩、易扩展、支持分布式查询。适用于互联网敏捷开发的需求。互联网架构师必备技术！MongoDB现在也成为Google、Facebook、阿里巴巴、腾讯、百度等一线互联网公司招聘的关键技能！

2.**NoSQL排名第一**

非常荣幸获得作者授权翻译MongoDB官方团队撰写的《MongoDB in Action》第2版。

非常荣幸受邀成为阿里巴巴Mongodb技术大会嘉宾讲师，并成为Mongodb中国专家组成员。MongoDB官方团队Kyle Banker和Shaun Verch等精心撰写的书籍。此书是MongoDB领域最权威的书籍，也是Manning出版社的“In Action实战”系列的经典书籍之一。全球读者五星好评。

3.MongoDB互联网和物联网公司必备技术

MongoDB的公司国外著名公司：Google、Facebook、Tweet、思科、Bosch、Adobe、SAP等，国内著名公司包括：阿里巴巴、腾讯、百度、360公司、新浪微博、京东、携程、滴滴打车、摩拜单车等名企。新版MongoDB 3.4更是对云计算和大数据分析Spark提供更强的支持，大数据工程师也必须掌握MongoDB，重要性不言而喻。

《Mongodb实战》第一部分对MongoDB做了整体介绍，并介绍了实际的开发例子。另外还介绍了JavaScript shell和Ruby驱动，Java和C#例子在读者群中下载。这两个概念会贯穿于本书的所有例子中。

本书主要是面对开发人员，但是如果你是临时使用MongoDB，本书也可以提供很多帮助。虽然我们主要关注MongoDB数据库，但是之前的编程经验会帮助我们理解例子代码。如果之前使用过关系型数据库，那就太棒了！我们会经常对比这两种数据库。

在编写本书的时候，MongoDB 3.x版本是最新的版本，但是本书大部分介绍会兼顾之前的MongoDB版本。我们也会强调哪些特性不适用于旧版本。

大部分例子我们会使用JavaScript编写，因为MongoDB JavaScript shell便于实验操作。Ruby是特殊的MongoDB语言之一，我们的例子代码会介绍如何在真实的项目中使用Ruby操作MongoDB数据库。放心，就算大家不是Ruby程序员，我们也可以使用其他语言来开发MongoDB，因为语法都是相似的。

第1章介绍MongoDB的历史、设计目标，以及典型的使用场景。大家也会看到为什么MongoDB可以在与其他NoSQL数据库的竞争中胜出，做到独一无二。

第2章介绍MongoDB shell的用法，让大家熟悉常见的命令工具。我们会介绍MongoDB基本的查询语言，并且通过创建、查询、更新和删除文档来练习实战。本章还会介绍一些高级的shell技巧和MongoDB命令。

第3章主要介绍MongoDB驱动以及BSON数据格式。

这里我们会介绍如何使用Ruby语言来操作数据库，而且会使用Ruby开发一个Demo程序来演示MongoDB的灵活性和强大的查询机制。

要充分掌握本书介绍的知识，请仔细阅读并且亲自尝试每一个例子。如果还没有安装MongoDB，就请参考本书附录A的内容，里面有详细的安装向导。[1]

2017年3月12日，阿里巴巴Mongodb技术大会新青年合影留念

[1]【译注】MongoDB 的安装可以参考官方文档 mongodb.org，如果你是其他语言比如 C#、Java 或者 Node 的开发者，也不要担心，官方文档提供了详细的介绍。大家也可以加群 203822816 索取例子代码。

关于译者：

1. Mongodb中国区专家、微软特邀讲师

2. 微软大企业客户技术顾问，曾任：购酒网、北极绒 复旦元方、广西省政府云平台等公司技术架构顾问

3. 获得吉林大学计算机科学与技术学士学位，上海交通大学硕士学位

4. 国外经典《WCF技术内幕》《WCF服务编程》第3，4版《ASP.NET MVC4 Web编程》、《JQuery实战》第3版、《MongoDB实战》第2版、《24种云计算架构模式》译者

5. 受邀为微软中国、盛大网络、玫琳凯中国研发中心、世界500强约翰迪尔、一嗨租车、沪江网、中国东方航空、美国IGT、世界500强花旗银行、世界500强达丰集团、世界500强台达集团、美国国家仪器、上海交通大学软件学院、奥伯特石油、中国体彩集团总部、世界500强南方电网集团、阿里系恒生证券、丹麦诺和诺德集团、阿里巴巴Mongodb大会等中外名企、名校授课。

6. 2017年3月12日，受邀作为嘉宾讲师在阿里巴巴Mongodb技术大会，发表Mongodb分布式高可用架构主题演讲。

7. 喜欢格言座右铭：Stay Hungry,Stay Foolish(谦卑若愚，好学若饥)。

8. 新青年软件训练营（54peixun.com） 联合创始人

（Mongodb中国学习交流群 二维码）

第 1 章

全新Web数据库

A database for the modern Web

本章内容

- MongoDB历史、设计目标以及关键特性
- 简要介绍shell和驱动
- 使用场景和限制
- MongoDB 最新更新

如果你最近几年在开发Web应用，则可能一直使用关系型数据库作为主要的存储方式。如果你熟悉SQL，可能非常喜欢标准化[1]的数据模型、事务的必要性，以及持久化引擎提供的保证。简单来说就是关系型数据库成熟并且大名鼎鼎。当开发者开始寻找其他替代数据库时，关于新技术的有效性和实用性的疑问不断产生：这些新的数据库要取代传统关系型数据库吗？谁在生产环境中使用它，为什么？迁移到非关系型数据库的动机是什么？其实所有问题的答案归结到一起就只有一个：为什么开发者喜欢MongoDB？

MongoDB是为快速开发互联网Web应用而设计的数据库系统。其数据模型和持久化策略就是为了构建高读/写吞吐量和高自动灾备伸缩性的系统。无论系统需要单个还是多个节点，MongoDB都可以提供高性能。如果你经历过关系型数据库的伸缩困境，那么使用MongoDB就可避免这种困境。但是并非每个人都需要伸缩性操作。如果你需要的就是单台数据库服务器，那么为什么还要使用MongoDB呢？

或许开发者使用MongoDB的最大理由并非是其伸缩策略特性，而是其直观的数据模型。MongoDB在文档里存储数据而不是在行里。什么是文档？下面就是一个例子：

```
{
```

[1]当我们提到标准化的数据模型时，通常是指数据库减少冗余的设计范式。例如，在 SQL 数据库中我们可以把数据分割为不同的部分，比如 Users 和 Orders 表，然后减少用户信息存储的冗余数据，我们称作 3NF。

```
  _id: 10,
  username: 'peter',
  email: 'pbbakkum@gmail.com'
}
```

这是个非常简单的文档；它存储了用户的一些简单信息字段（听起来很酷）。那这个数据模型的优势是什么？设想这种情况：需要为每个用户存储多个email。在关系型数据库里，需要创建单独的email表，然后通过外键关联起来；而MongoDB提供了非常简单的方法来解决这种问题：

```
{
  _id: 10,
  username: 'peter',
  email: [
    'pbbakkum@gmail.com',
    'pbb7c@virginia.edu'
  ]
}
```

如上所示，我们只是创建了一个email数组就解决了问题。作为开发者，我们会发现，这一优点非常有用，在数据经常变化的时候我们只需要存储一个结构化文档即可，无须担心数据模型的变化。

MongoDB的文档格式基于JSON，一种流行的数据存储格式。JSON是的JavaScript Object Notation的简称。正如我们看到的，JSON数据结构由键值对组成，也可以内置嵌套。这一点与其他语言中的哈希映射以及字典类型相似。

基于文档的数据模型可以表示丰富的、多层次的数据结构。它经常用来处理无须多表关联的工作。

例如，假设要为电商网站的数据库建模，若使用标准的范式设计数据库，信息可能分割到几十个表中存储，要获得完整的数据库里存储的商品信息，则可能需要关联SQL查询。

相比之下，使用文档模型时，绝大部分单个商品信息都可以存储在单个文档里。打开MongoDB JavaScript shell就可以轻易获取商品的类JSON数据信息。我们也可以查询或者操作它。MongoDB的查询功能是专门用于处理结构化文档操作的，所以用户从关系型数据库到非关系型数据库查询体验基本在同一层次。此外，大部分开发者使用的是面向对象编辑语言，他们希望找到更好的数据库用于映射对象。使用MongoDB，语言中定义的对象可以原样持久化保存，减少了对象映射的复杂性。如果你开发过关系型数据库，这些经验也有助于转换现有技能到新的数据库中。

如果你不熟悉表格数据和对象数据之间的区别，则可能会有很多疑问。不过请放心，学习完

第1章你就会对MongoDB的特性以及设计目标有个清晰的认识。我们会先介绍一下MongoDB的发展历史，以及其主要特性。然后会介绍其他NoSQL[1]解决方案，以及MongoDB是如何解决问题的。最后我们会介绍MongoDB最佳的使用场景，以及它的局限性。

MongoDB在多个方面备受批评，其中有些是公正的，但有些是歪曲的。我们的观点是，它就是开发人员武器库中的一件兵器，和其他数据库一样，你应该知道它的优点和缺点。某些工作需要多表关联和更多的内存管理，而有些工作更适合使用基于文档的数据模型。缺少数据架构定义意味着MongoDB更灵活，更适合快速开发模式。我们的目标是告诉大家自己决定MongoDB是否适合自己，以及如何高效地使用它。

1.1 为互联网而生

Built for the Internet

MongoDB的历史不长，但是它诞生于一个雄心勃勃的项目。在2007年，纽约一个叫10gen的创业团队开始工作在一个平台即服务(PaaS)上，由一个应用服务器和一个数据库组成，托管Web应用，根据需要伸缩。与Google App Engine类似，10gen平台的设计目标就是自动处理伸缩和管理硬件与软件基础架构，解放开发者，使得他们可以专注于应用开发。10gen最终发现，开发者并不喜欢放弃对于技术栈的控制，但是用户却喜欢上了10gen的数据库技术。这就导致了10gen团队专注于开发这个数据库产品，最后就形成了MongoDB项目。

10gen公司的名字也已经修改为MongoDB, Inc。该公司继续支持这个开源项目的开发工作。代码完全公开下载，并且可以免费修改、使用，只要遵守代码开源协议即可。而且鼓励社区提交Bug和补丁。另外，MongoDB核心开发者要么是公司的联合创始人，要么是公司的员工，项目的路线图专注于满足用户社区的需求以及创建兼具关系型数据库和分布式键值存储最佳特性的数据库。因此MongoDB公司的业务模型与其他著名的开源公司不同：支持开源产品的开发并且提供给终端用户订阅服务。

从MongoDB历史中我们了解的最重要的事情就是MongoDB的设计目标就是极简、灵活、作为Web应用栈的一部分。这些使用情况驱动了MongoDB开发过程，并且可以帮助解释它的特性。

[1] 2009 年出现的 NoSQL 一词主要用于描述当时日益流行的非关系型数据库，它们的共同点是使用了 SQL 之外的查询语言。

1.2 MongoDB 关键特性

MongoDB's key features

数据库很大程度上是由其数据模型定义的。本节里，我们会看到文档数据模型，然后会看到MongoDB对此数据模型的高效处理特性。本节里也会介绍其他操作，专注于MongoDB的主从复制以及水平扩展策略。

1.2.1 文档数据模型

MongoDB的数据模型是面向文档的。如果对于这里的文档一词不熟悉，则可以通过下面的例子来理解。

JSON文档中除了数值类型以外，其他都需要一对引号。

列表1.1展示了JavaScript版本的JSON文档。这里的引号不是必须的。

列表1.1 表示新闻网入口的文档

```
{
  _id: ObjectID('4bd9e8e17cefd644108961bb'),
  title: 'Adventures in Databases',
  url: 'http://example.com/databases.txt',
  author: 'msmith',
  vote_count: 20,
  tags: ['databases', 'mongodb', 'indexing'],
  image: {
    url: 'http://example.com/db.jpg',
    caption: 'A database.',
    type: 'jpg',
    size: 75381,
    data: 'Binary'
  },
  comments: [
    {
      user: 'bjones',
      text: 'Interesting article.'
    },
    {
      user: 'sverch',
      text: 'Color me skeptical!'
    }
  ]
}
```

id 字段
主键

❶ 作为字符串数组存储的标签

❷ 指向另外一个文档的特性

❸ 作为对象数组存储的评论

这个例子代码展示了新闻网站上一篇文章的JSON文档格式（想想Reddit或者Twitter，国内读者若无法访问，则可以类比网易或者今日头条）。正如大家看到的，该文档包含一系列名称

和值的集合。这些值可以是简单的数据类型，比如字符串、数字和日期等。当然，这些值也可以是数组，甚至是JSON文档❷。后面构造的文档表示了各种不同的数据结构。我们可以看到例子文档包含tags属性❶，它把文章的标签存储到数组里。但是最有意思的是comments属性❸，它是一个由评论组成的数组。

从内部来讲，MongoDB以二进制JOSN格式存储文档数据，或者叫做BSON。BSON有相似的数据结构，但是专门为文档存储设计。当查询MongoDB并返回结果时，这些数据就会转换为易于阅读的数据格式。MongoDB shell使用JavaScript获取JSON格式的文档数据，这也是我们绝大多数例子使用的格式。我们会在后面的章节里深入讨论BSON数据格式。

关系型数据库包含表，MongoDB 拥有集合。换句话说，MySQL在表的行里保存数据，而MongoDB在集合的文档里保存数据，你可以把集合当做一组文档数据。集合是MongoDB中非常重要的概念。集合中的数据存储在磁盘上，而且大部分查询需要指定查询的目标集合。

我们来花点时间来比较MongoDB集合与标准的关系型数据库表示相同数据的差别。图1.1所示为可能的关系结构。因为表本质上是平滑的，表示文档中的文章一对多关系需要多个表。我们可以创建一个posts表存储文章的核心信息。然后可以再创建三个其他的表，每个表包含一个外键字段post_id，关联最初的文章。符合范式设计的数据集保证数据集只在一个地方存储一次。

但是严格的范式设计是需要成本的。值得注意的是，有时候需要一些程序集。要显示刚才关联的文章post，需要执行post和comments表的链接查询。是否需要严格范式设计则最终取决于要建模的数据类型。第4章会深入讨论这个问题。面向文档的数据模型天生就适合表示集中形式的数据，允许我们处理整个数据，从评论到标签，都可以包含在单个数据库对象中。

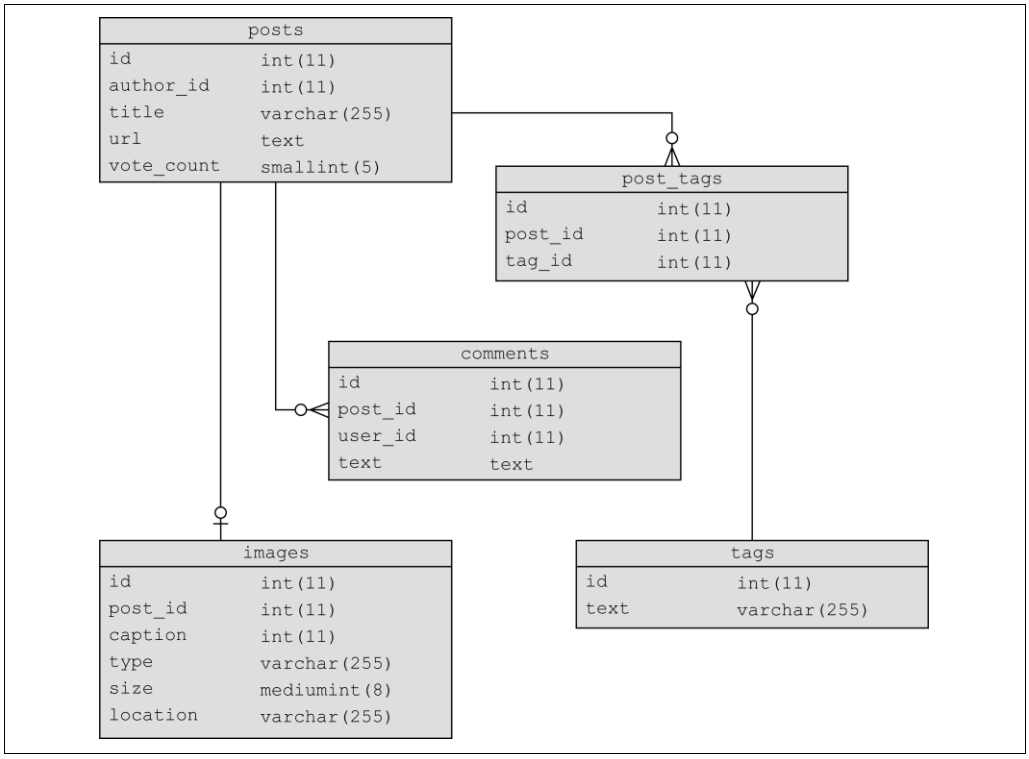

图 1.1　社交新闻网站的实体关联数据模型

十字架线条代表一对一关系，所以一个posts表记录对应一个images记录。三叉线表示一对多关系，所以comments表可以多个评论关联posts表的一条记录。

你可能已经注意到了，除了支持丰富的数据结构外，文档不需要遵守严格的数据定义schema。我们在表里存储行，每个表有严格的schema，指定每个列的数据类型。如果某行需要扩展字段，就必须修改表结构。MongoDB把文档归集到集合中，集合不需要定义任何schema。理论上，每个集合中的文档都可以拥有不同的数据结构；实际上，集合中的文档都是相对一致的。例如，每个posts文章集合中的文档都有标题、标签和评论字段等。

无 schema 模型的优点

不强制定义schema带来了一些好处。首先，应用程序的代码强制数据结构而不是数据库。在频繁修改数据定义的时候这可以加速应用程序开发。

其次，无schema模型允许用户使用真正的变量属性来表示数据。例如，假设你在构建一个电商的商品目录表。由于没有办法知道一个商品包含什么属性，因此应用程序需要处理这些变

化。传统的处理方法是在固定schema数据库里使用实体属性值模式[1]，如图1.2所示。

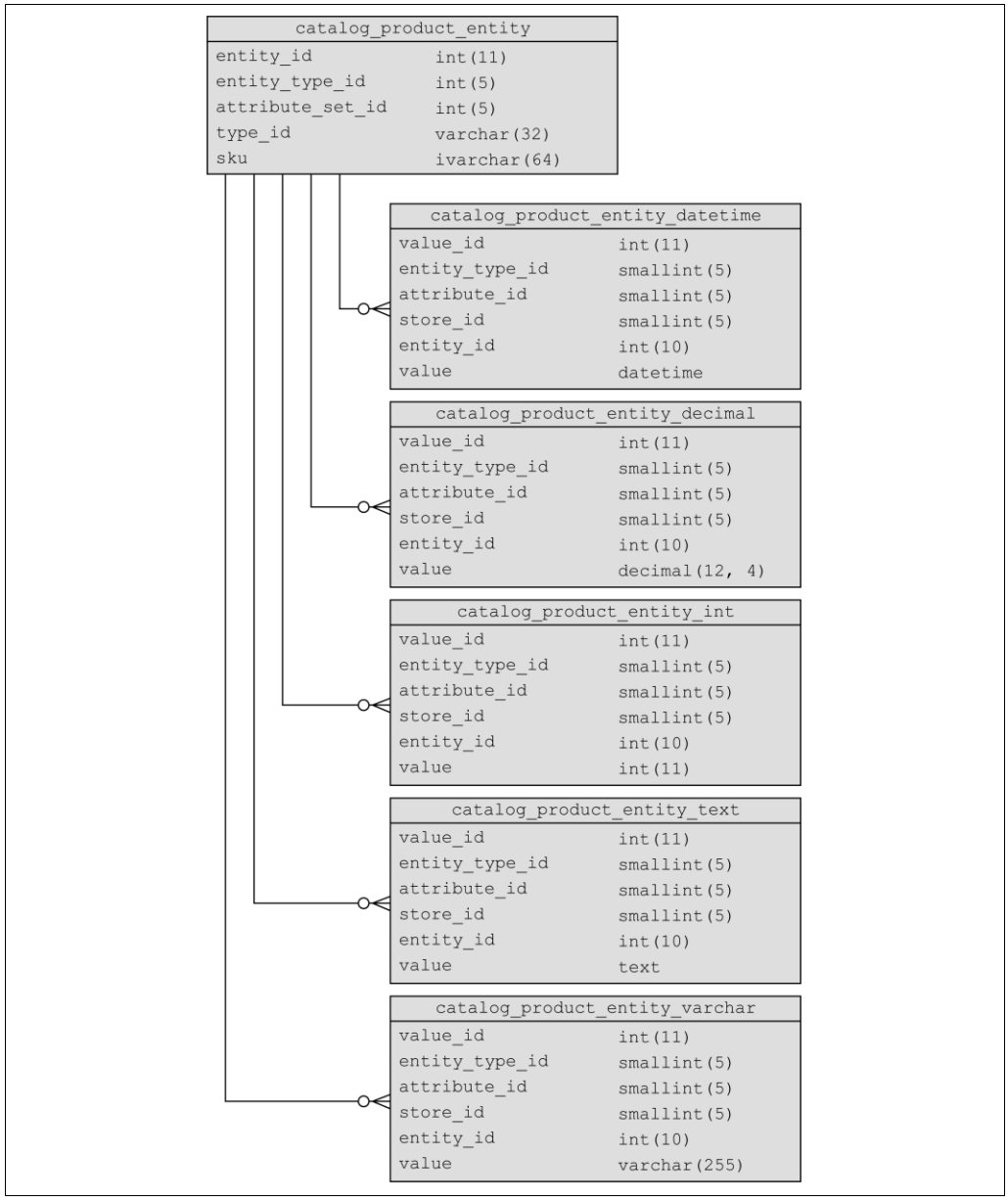

图 1.2　电商应用的部分 schema，这些商品表使用了动态属性创建模式

你所看到的只是电商网站的部分数据模型。注意：这些基本都是相似的，除了一个属性值是

[1]更多信息请参考 http://en.wikipedia.org/wiki/Entity-attribute-value_model.

通过数据类型datatype变换的以外。这个数据结构允许管理员来定义额外的产品类型以及属性，但是导致的问题相当复杂。思考一下，如果使用MySQL shell来检查或者更新某个产品模型的数据，则商品SQL关联查询就会相当复杂。而文档建模就不需要关联，而且可以动态添加新属性。虽然不是所有的模型都这么复杂，但是使用MongoDB开发应用就不需要担心未来可能的数据字段的变化。

1.2.2 ad hoc 查询

常说的系统支持主动查询模式（ad hoc queries）是指不需要事先定义系统接收何种查询。关系型数据库有这个属性，它们忠实地执行格式正确的包含各种条件的SQL查询。如果使用过关系型数据库，就知道ad hoc查询很容易。但是不是所有的数据库都支持动态查询，例如，键值存储的查询只支持一个领域的查询：键key。与其他系统类似，键-值存储数据库牺牲丰富的查询功能来换取更简单的伸缩模型。MongoDB的设计目标之一是保留大部分关系型数据库的功能。

要看下MongoDB查询语言如何工作，可先来看个简单的例子：包含文章和评论。假设我们要查询所有包含标签politics并且有10个以上投票的文章。若使用SQL语句，代码如下：

```
SELECT * FROM posts
  INNER JOIN posts_tags ON posts.id = posts_tags.post_id
  INNER JOIN tags ON posts_tags.tag_id == tags.id
  WHERE tags.text = 'politics' AND posts.vote_count > 10;
```

等价的MongoDB查询，使用了document文档作为匹配器，则代码如下。特别的`$gt`键表示大于条件。

```
db.posts.find({'tags': 'politics', 'vote_count': {'$gt': 10}});
```

注意：两个查询假设了不同的数据模型。SQL查询依赖于严格的范式模型，posts和tags存储在不同的表里，而MongoDB假设tags保存在每个post文档对象里。但是两个查询演示了任意组合属性的功能，这也是ad hoc查询的功能。

1.2.3 索引

ad hoc查询的一个关键元素就是查找在创建数据库时还不知道的值。随着数据库中添加的文档数据越来越多，查询值的成本变得越来越高。这有时无异于大海捞针。因此，需要一种高效的方式来搜索数据。

理解数据库索引最好的方法就是类比：许多书籍都有索引，包含关键字和页码。假设数据库索引也是提供相似服务的数据结构。假设你有一本秘笈食谱，而你想找出所有和梨子有关的烹饪方法（假设你有许多梨子，又不希望它们坏掉）。耗时的方法就是翻遍图书的所有页面查找每个方法，逐页查找梨子的做法。而大部分人会选择查看书籍的索引，找出所有包含梨子关键字的食谱列表。

MongoDB中的索引就是使用了B-树（平衡树）数据结构。B-树索引也大量使用于许多关系型数据库中，对于不同的查询做了优化，包括范围扫描和条件子句查询。但是新的引擎已经支持日志结构合并-树(LSM)，可以在MongoDB 3.2版本中使用。

大部分数据库会给每个文档对象一个主键（primary key），一个唯一的数据标识。每个主键会自动索引，这样就可以使用唯一的键来高效地访问每个数据，MongoDB也不例外。但是不是所有数据库都允许我们来为单个的行或者文档建立索引。这些叫做辅助索引（secondary indexes）。许多NoSQL数据库，比如HBase，被当做keyvalue数据库，这是因为它们不允许辅助索引（secondary indexes）。通过允许多个辅助索引，MongoDB可以允许用户优化不同的查询。这是MongoDB重要的功能特性。

使用MongoDB，每个集合我们可以创建64个索引。这些索引也可以在其他关系型数据库中找到；升序、降序、复合键、哈希、文本以及地理空间索引[1]。因为MongoDB和绝大多数关系型数据库RDBMSs使用了相同的索引数据结构，所以管理这些系统的建议都是类似的。你会在下一章里阅读索引的内容，而且必须明白索引对于高效操作数据库至关重要，第8章里会深入介绍这个主题。

1.2.4 复制

MongoDB提供了数据库复制特性，叫做可复制集合（replica set）。可复制集合在多个机器上分布式存储数据，在服务器或者网络出错时，实现数据冗余存储和自动灾备。此外，复制还用于伸缩数据库读操作。如果你在开发读取密集型应用，比如常见的一些网站项目，就可以通过分散读取压力到可复制集群中的服务器来实现。

可复制集合由多台服务器组成集群。通常，每个服务器有独立的物理机。我们调用这些节点，任意时候，一个节点作为主节点，则其他的作为次节点。与其他数据库中的主从复制类似，可复制集合的主节点可以同时接受读/写操作，但是从服务器只能进行读操作。

[1]地理位置索引允许高效率查询经纬度点信息；会在本书后面进行讨论。

真正让可复制集独一无二的是它可以支持自动化灾备：如果主节点失败，则集群中会选择一个从节点，并自动提升为主节点。当之前的主节点回归时，它会继续作为从节点。整个过程如图1.3所述。

复制是MongoDB最有用的特性，我们会在后面的章节里深入讲解。

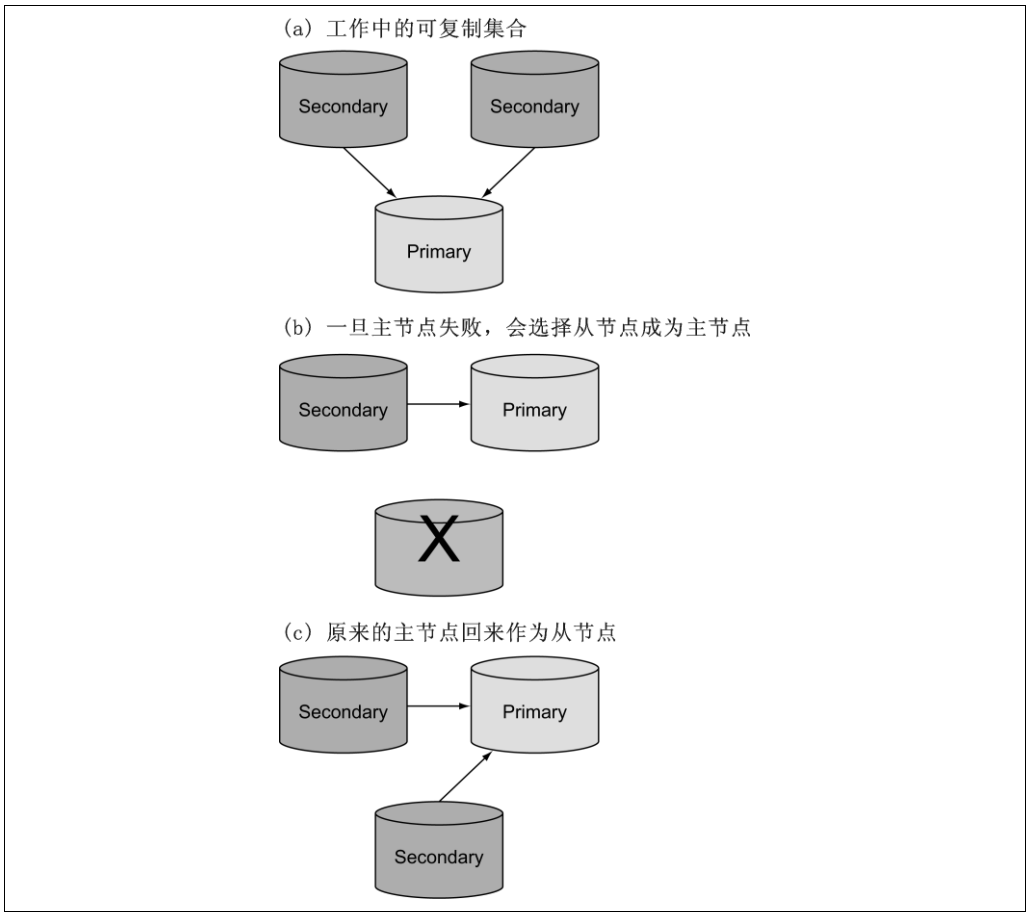

图 1.3　可复制集合的自动化灾备

1.2.5　加速与持久化

要理解MongoDB的持久化方法，首先要思考一些新概念。在数据库系统领域，写入速度和持久性之间存在矛盾的关系。

写入速度（write speed）可以理解为数据库在给定的时间内插入、更新和删除的容量。持久

性（durability）指的是这些写操作被永久保存的保证级别。

例如，假如写入100条每条50 KB的记录到数据库中，然后立即切断电源。当重新开机的时候，这些数据还能恢复吗？答案是：取决于数据库系统的配置以及托管硬件。绝大部分数据库默认启动了很好的持久性，所以如果发生这种事情也是安全的。对于一些应用，比如存储日志线，即使可能导致数据丢失，快速写入也很有意义。写入磁盘的速度远远慢于写入内存RAM的速度。特定的数据库，比如Memcached，专门写入RAM，所以它的速度非常快，但是数据容易丢失。换句话说，很少有数据库只写磁盘，因为这种操作性能太低，无法接受。因此，数据库设计者经常需要在速度和持久性之间妥协，做出最佳平衡。

事 务 日 志

可以在MySQL的InnoDB里看到写入速度和持久性之间的妥协。InnoDB是一个事务性存储引擎，它可以确保持久性。它通过两个地方确保实现这一目标：一是在事务日志，二是在内存缓存池里。事务日志会立即同步到磁盘，而缓存池会通过后台线程同步到磁盘上。双写的原因是随机I/O比顺序I/O慢得多。因为写入主数据文件是随机I/O，而把这些修改写入RAM就快得多，而后再允许同步到磁盘上。

由于某些类型的写磁盘会确保持久性，而且重要的是顺序写磁盘的，因此很快；这也是事物日志提供的特性。

对于意外关机的情况，InnoDB可以替换事物日志，并且更新主数据文件。这在确保高级别持久性的同时提供了一个可以接受的性能级别。

对于MongoDB，用户通过选择写入语义来维持速度与持久性之间的平衡，决定是否启用日志。从MongoDB 2.0以后就默认启用了日志功能。在2012年11月份发布的驱动里，MongoDB确保写入操作在返回用户之前已经写进了RAM，但此特性可以配置。我们也可以配置MongoDB为fire-and-forget，发送给服务一个写命令而不需要等待确认结果。也可以配置MongoDB来确保已经写入到各个可复制集群的节点中，再返回确认结果。对于高容量、低价值的数据（比如点击流和日志），写入后就不用管的模式比较理想。对于重要的数据，安全模式的设置是必须的。

要知道在MongoDB 2.0之前的版本里，默认是使用不安全的写入后不用管的模式fire-and-forget，因为10gen开始开发MongoDB时，只专注于数据层，相信应用层会处理这些错误。但是随着MongoDB使用越来越流行，不仅仅在Web层，对于不想丢失数据的应用来说

这样太不安全了。

从MongoDB 2.0开始，日志功能默认是启用的。启用日志功能后，默认100毫秒就会写一次日志文件。如果服务器意外关机，日志会通过重启服务器来确保MongoDB数据文件恢复为一致状态。这是运行MongoDB最安全的方式。

对于写入压力，可以通过关闭日志功能以提高性能。坏处是意外关机之后可能导致数据文件冲突。因此，关闭日志功能，就应该使用主从复制模式，推荐一个从服务器，即使一台机器关机，还有一台机器保证数据的完整性。

设计MongoDB的目标就是让大家可以保持速度与持久性平衡，但是对于重要的数据，我们强烈推荐安全设置。复制和持久化的主题范围很大，我们会在第11章里详细介绍。

1.2.6 伸缩

伸缩数据库的最简单方式就升级服务器硬件。如果你的应用是运行在单个节点上，则通常可行的方案就是通过组合添加更快的磁盘、更多内存以及更强的CPU来解除数据库性能瓶颈。提升单节点参数的做法通常也称垂直扩展（vertical scaling或scaling up）。垂直扩展非常简单、可靠，但是达到某个点后成本很高，但是最终我们会达到一个无法低成本垂直扩展的临界点。

然后可以考虑水平扩展（horizontally或scaling out），如图1.4所示。水平扩展指的是在多台机器上分布式存储数据库，而不是提升单个节点的配置。水平扩展架构可以运行在许多台很小的、廉价的机器上，通常可以减少硬件的成本。而且，跨机器分布式存储数据可以降低宕机带来的丢失数据的后果。服务器无法避免关机。如果你已经做过垂直伸缩，经历了宕机，那就应该处理一下系统最依赖的服务器失败问题了。相比水平扩展架构的失败，作为一个整体它降低了单个节点宕机带来的风险概率。

设计MongoDB的目标就是利用其水平伸缩。它通过基于范围的分区机制来实现水平扩展，称为分片机制，它可以自动化管理每个分布式节点存储的数据。另外，还有基于哈希和基于tag的分片机制，这也是另外一种形式的基于范围的分片机制。

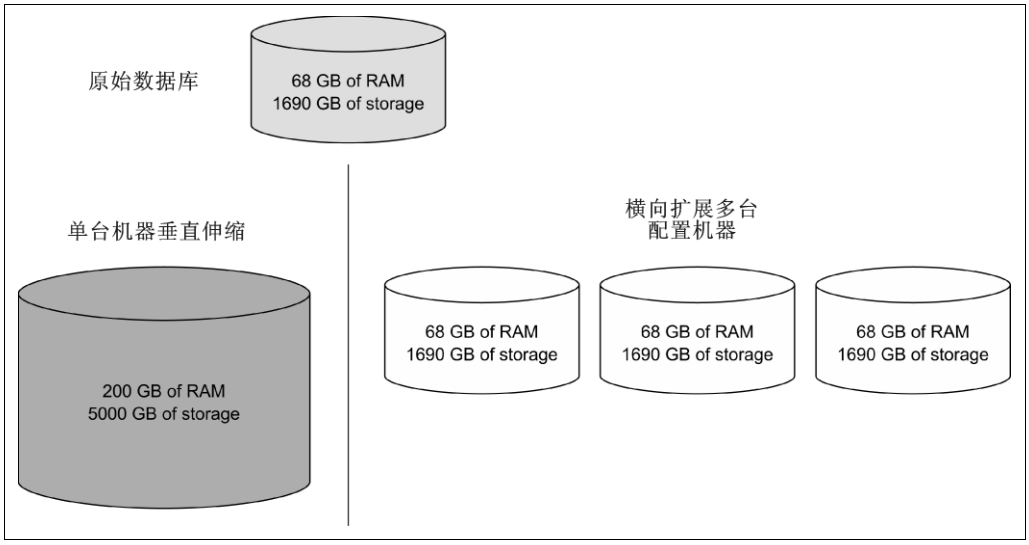

图 1.4　水平与垂直扩展

分片系统处理额外的分片节点，而且它还会处理自动化灾备。每个独立的节点是一个可复制集合，至少由2台机器组成，确保节点失败的时候可以自动恢复。所有这些都意味着没有应用节点必须处理这些逻辑；我们的应用与MongoDB集群通信就好像与单个节点通信一样。第12章会深入讲解分片集群的知识。

我们已经了解了MongoDB最重要的功能特性；第2章里我们会开始学习如何实际开发MongoDB。现在我们近距离来看看这个数据库。下一节，我们会在它的环境里学习MongoDB，与核心服务器一起发布的工具，以及一些存取数据的方法。

1.3　核心服务和工具

MongoDB's core server and tools

MongoDB使用C++编写，由MongoDB，Inc公司负责开发。这个项目支持主流的操作系统，包括Mac OS X、Windows、Solaris 以及最流行的Linux版本。这些平台的预编译版本可以在http://mongodb.org下载。MongoDB是开源的，基于GNU-Affero General Public License (AGPL)开源协议。源代码可以在GitHub下载，社区也可以贡献力量。但是项目由MongoDB, Inc核心服务器团队主导，而且他们开发了绝大多数的代码[1]。

[1]【译者注】翻译此书时最新的版本是 3.2.6。我们也使用了最新的 3.4 进行集群和安全的实战配置。

> **关于 GNU-AGPL 开源协议**
>
> GNU-AGPL开源协议存在一些争议。实际上，这个许可协议意味着源代码可以下载，鼓励社区参与。但是GNU-AGPL要求，任何对于源码的修改必须公开发布造福社区。这可能对于那些想修改MongoDB源码，但是不想公开的公司是个困扰。对于这些想要保护自己公司核心服务器代码的公司，MongoDB，Inc提供了特别的商业许可证。

MongoDB 1.0于2009年11月发布。主要的发布大约3个月一次，偶数版本作为稳定版，奇数版本作为开发版。编写此书时最新的版本是v3.0[1]。

下面介绍的是与MongoDB一起发布的开发应用需要的工具和各种语言的驱动组件。

1.3.1 核心服务器

核心服务器通过名为`mongod`的可执行文件运行（`mongodb.exe`在Windows系统中运行）。`Mongod`服务器进程使用自定义的二进制协议从网络上接受命令。所有`mongod`进程的数据库在类Unix系统中默认存储在/data/db路径下，在Windows中存储在c:\data\db下。本书中的某些例子更适用于Linux系统。绝大部分MongoDB生产服务器都运行在Linux上，因为它们更加可靠、应用更广泛和更具优越性。

Mongod可以在几种模式下运行，比如独立模式，或者可复制集群模式。在生产环境中，我们推荐使用可复制群模式MongoDB。通常我们会看到，可复制集群由两台服务器加上一个mongod作为裁判组成。最后，还有一个独立的mongos路由服务器，它用来在分片集群中转发不同的请求到后台服务器。不用担心这些选项，我们会在第11章和第12章中详细介绍这些知识。

配置mongod进程相对简单，它可以接受命令行参数，也可以接受配置文件文本控制。一些常见的配置可以通过修改mongod侦听的端口和存储数据的目录来实现。要查看这些配置参数，可以运行mongod 帮助文件。

[1]你应该使用最新的稳定版本，例如 v3.6（2017 年 9 月发布）。在附录 A 里有完整的安装指南。

1.3.2 JavaScript shell

MongoDB命令行工具是一个基于JavaScript[1]的数据库操作和管理工具。`Mongo`加载命令`shell`后连接特定的`mongod`进程，或者默认本地运行。MongoDB shell和MySQL shell类似，最大的不同是它基于JavaScript和SQL脚本。例如，可以选择一个数据库，然后插入一个简单的文档对象到`users`集合中：

```
> use my_database
> db.users.insert({name: "Kyle"})
```

第一个命令用于选择你要使用的数据库，MySQL用户很熟悉。第二个命令是Javascript表达式，用于插入一个简单的文档。要查看插入的结果，可以使用如下简单的查询：

```
> db.users.find()
{ _id: ObjectId("4ba667b0a90578631c9caea0"), name: "Kyle" }
```

`Find`方法返回插入的文档数据，带有一个对象`ID`。所有的文档都需要一个`_id`字段作为主键。如果可以确保唯一，也可以自己设置`_id`的值。如果忽略这个`_id`值，MongoDB会自动生成一个唯一的ID插入数据库。

除了允许插入和查询数据，shell还允许我们运行管理员命令。某些例子还保护查看当前数据库操作，检查复制到从节点的状态，以及配置分片集群的某个集合。正如你看到的，MongoDB shell是个非常强大的工具，值得我们学习。

所有这些知识点，都会通过使用某个语言结合MongoDB开发一个应用来完成。为此，我们必须了解一些MongoDB的语言驱动程序。

1.3.3 数据库驱动

也许数据库驱动的概念让人想起底层设备驱动的黑客噩梦，其实不要怕，因为MongoDB使用起来非常简单。驱动是应用程序用来与MongoDB服务器通信的代码。所有的驱动都保护查询功能、搜索结果、写入数据和运行数据库命令。针对MongoDB驱动开发提供的API都尽量保证所有的语言使用统一的接口。例如，所有的驱动都实现了相似的方法来保存数据到集合中，但是文档的表示类型都是基于自己语言最原始的类型定义。

[1]如果要学习 Javascript，可以阅读 http://eloquentjavascript.net 网站。Javascript 的语法与 C 语言的很像，C++、Java 和 C#都属于 C 语言语系。如果你熟悉其中一种语言，应该很容易理解 Javascript 的例子代码。

在Ruby语言中使用Ruby哈希；在Python中，字典是合适的数据结构类型；在Java中，缺少类似的语言基元类型，通常要使用Map对象表示，或者其他类似的类型。有些开发者喜欢使用ORM框架来管理这些数据的表示工作，但是实际上，MongoDB驱动已经非常强大，这些工作都是多余的[1]。

语 言 驱 动

在撰写本书时，MongoDB公司官方支持C、C++、C#、Erlang、Java、Node.js、JavaScript、Perl、PHP、Python、Scala、Ruby语言的驱动——这个列表还在增加。如果需要其他语言开发新的驱动，就很有可能已经有社区开发了，虽然并非MongoDB公司官方管理的驱动项目，但是应该也不错。

如果没有你的语言对应的社区支持的驱动，则也可以找到构建新驱动的官方文档规范http://mongodb.org。所有官方的驱动都大量使用在生产环境中，而且遵守Apache许可证，为驱动开发者提供了许多好的参考例子。

从第3章开始，我们将会深入介绍驱动如何工作，以及如何使用它们编写程序。

1.3.4 命令行工具

MongoDB包含了几个命令行工具。

- mongodump 和 mongorestore——备份和恢复数据库的工具。mongodump把数据库数据保存为原生的BSON格式，因此最适用于备份。这个工具的一大优点是适合热备份，而且非常容易使用mongorestore命令恢复。
- mongoexport 和 mongoimport——导入或者导出JSON、CSV、TSV[2]格式的数据。这在大家需要多种格式的数据时非常有用。Mongoimport还可以用来导入大数据集合，只是经常需要在导入之前调整数据模型以便于发挥MongoDB的最大优势。此时最简单的导入数据的方式就是使用自定义脚本。
- mongosniff——一个用于查看发送给数据库命令的嗅探工具。通常会把BSON转换为人

[1]【译者注】在 Java 和 C#等语言中，MongoDB 不同的客户端驱动也提供了自定义封装的文档类型。

[2] CSV 表示逗号分隔的值(comma-separated value)，意味着使用逗号把数据分割为几块。这是表示表格数据的流行方式，因为列名和行值可以列举在可读的文件中。TSV 表示制表符分割的值（tab-separated values），用制表符 Tab 来分割数据。中国 MongoDB 学习交流群 511943641

类可读的shell语句。

- mongostat——与iostat类似，这个工具用来轮训MongoDB，提供有帮助的状态信息，包括每秒的操作数(增、删、改、查等)，分配虚拟内存的数量，以及服务器的连接数量。
- mongotop——与top类似，这个工具用来轮训MongoDB，并且显示它在每个集合里花费的读取和写入数据的时间总数。
- mongoperf——帮助我们了解MongoDB实例磁盘操作的情况。
- mongooplog——展示MongoDB操作日志里的信息。
- Bsondump——把BSON文件转换为人类可读的格式，包括JSON。

我们会在第2章里详细介绍这些知识点。

1.4 为什么是 MongoDB?

Why MongoDB?

我们已经了解了为什么MongoDB是项目不错的选择的一些原因。这里，我们要更加详细地澄清一下，首先，要考虑一下MongoDB的设计目标。根据MongoDB之父的解释，它被用来设计组合键值对存储和关系数据库的最佳特性。因为简单，所以键值对存储非常快，而且容易伸缩。关系型数据库难以伸缩，至少在水平方向是这样的，但是关系型数据库拥有更加丰富的数据模型和查询语言。MongoDB在两者之间做了妥协，具备了二者的某些有用的功能。它最终的目标就是易于伸缩，存储丰富的数据结构，并且提供复杂的查询语言。

关于使用场景：MongoDB是做Web应用、分析应用的首要数据库。此外，它还比较容易存储无schema数据，也就是弱数据结构的数据。MongoDB也适用于存储无法事先知道数据结构的数据。

这些说法看起来比较虚幻。为了验证这些说法，我们来对比MongoDB和当前使用的不同数据库。接下来你会看到一些MongoDB的专门使用场景，还包括一些生产环境使用的例子。然后，我们会讨论一些使用MongoDB进行实际开发的重要考虑。

1.4.1 MongoDB 与其他数据库对比

数据库的数量非常多，而且对比所有的数据库是不现实的。幸运的是，绝大部分数据库属于

某个类别。在表1.1以及接下来的章节里，我们描述了简单而且详细的键值对存储、关系型数据库以及文档数据库，把它们与MongoDB做了对比。

表 1.1　数据库家族

	例子	数据模型	伸缩性模型	使用场景
简单的键值存储	Memcached	键值，值是二进制对象	变化的 Memcached 可以跨节点伸缩，把所有可用的 RAM 变为一个存储库	缓存、Web 网站等

续表

	例子	数据模型	伸缩性模型	使用场景
复查键值存储	HBase, Cassandra, Riak KV,Redis, CouchDB	变化的 Cassandra 使用的键值结构是列；HBase 和 Redis 存储二进制对象，CouchDB 存储 JSON 文档	最终一致性、多节点、分布式高可用和容易灾备	高吞吐量（活动源、消息队列）、缓存、Web 网站等
关系型数据库	Oracle 数据库、IBM DB2、SQL Server、MySQL、PostgreSQL	表	垂直伸缩。限制支持集群和手动分区	需要事务的系统(银行和金融)或SQL、规范化数据模型

简单的键值存储

简单的键值存储功能如其名字所示：索引值是基于提供的key键。常见的使用场景就是缓存。例如，假设要缓存App渲染的HTML页面。此时的key可能就是页面的URL，值就是HTML页面本身的数据。注意，对于键值对存储而言，数据值是字节数组。没有强制的schema数据定义，这一点和关系型数据库不同，也没有数据类型的概念。这自然也限制了键值存储的操作：可以插入新的值，然后使用key来查询或者删除值。如此简单的系统自然也就非常快速和容易伸缩。

最有名的键值存储就是Memcached，它只在内存里存储数据，是以牺牲持久性来换取速度。它也是分布式的。Memcached节点运行在多个服务器上，也作为单个存储库，去除了跨机器节点保持高速缓存状态的复杂性。

与MongoDB相比，像Memcached这样简单的键值存储允许更快的读/写速度。与MongoDB不同，这些系统不能作为主要的存储数据库。简单的键值存储最好作为辅助手段，或者用作关系型数据库的缓存层，或者作为简单持久性的临时服务，比如工作队列。

复杂的键值存储

可以通过完善简单的键值存储模型来处理复杂的读/写模式或者提供更丰富的数据类型。此时，需要使用复杂的键值存储。其中一个例子就是亚马逊的Dynamo，在一篇非常流行的论文里有介绍，标题是“Dynamo: Amazon’s Highly Available Key-Value Store”（亚马逊高可用键值存储）。Dynamo的设计目标是在网络失败、数据中心故障时可以正常提供强壮的数据库功能，

这需要数据可以一直读/写，本质上需要数据自动化复制到各个节点上。如果一个节点失败，系统的用户——此时使用亚马逊购物车——不会经历任何的服务中断。Dynamo提供了一种系统向多个节点写入相同数据时解决不可避免冲突的方法，而且Dynamo易于伸缩。因为它是无主的——所有的节点都一样——这样比较容易理解系统是一个整体，而且可以方便地加入新节点。虽然Dynamo是个亚马逊的数据库系统，但是它的设计思想启发和影响了许多NoSQL数据库的设计，包括Cassandra、HBase、Riak KV。

通过了解谁开发了这些复杂的键值数据库，以及如何在实际项目中使用，我们可以知道这些系统如何发挥作用。我们来看下Cassandra，它实现了Dynamo的许多伸缩属性，而且受到Google的BigTable启发，提供了面向列的数据模型。Cassandra是个开源数据库，由Facebook构建，其目标是支持站内搜索功能。这个系统水平伸缩，支持索引超过50TB的数据，允许基于用户关键字搜索，属于基于用户ID索引，每条记录都由一组搜索关键字数组和用户ID数组组成，就是为了基于用户进行搜索[1]。

复杂键值存储由主流的互联网公司开发，比如Amazon、Google、Facebook，用来管理分布式系统中的超大量数据。换句话说，复杂键值存储管理着一个要求大存储与高可用的相对独立的域。因为它们采用了无主架构，系统容易使用额外的节点进行伸缩。它们选择了最终的一致性，意味着读不一定必须反映最新的写。但是为了换取弱一致性，牺牲的是写入时可能出现的节点失败。

对比MongoDB，它提供了更强的一致性、更丰富的数据模型以及辅助索引。后两者特性简单容易；键值存储可以在值里存储任意结构的数据，但是数据库不能查询这些值，除非他们被索引过。也可以使用主键进行查询，或者扫描索引的键，但是如果不使用索引，数据库对于这种查询毫无用处。

关系型数据库

我们已经介绍了很多关系型数据库知识，为了简洁起见，我们只需要讨论RDBMS（关系型数据库管理系统）与MongoDB的异同。流行的关系型数据库包括MySQL、PostgreSQL、Microsoft SQL Server、Oracle Database、IBM DB2等，有些开源，有些不开源。MongoDB和关系型数据库都可以表示丰富的数据模型。而关系型数据库使用固定格式的schema表，MongoDB属于无schema文档。绝大部分关系型数据库支持辅助索引和聚合查询。

从用户角度来看，关系型数据库最大的特性就是支持SQL查询语言。SQL是处理数据强大的

[1]参见 《Cassandra: A Decentralized Structured Storage System》，载于 http://mng.bz/5321。

工具，但是并非能完美解决所有的工作问题。某些情况下，相比MongoDB，它处理数据时更易于表示和更简单。此外，SQL在各个数据库之间的移植性很强，即使各个数据库支持有点差别。一种思考方式就是，SQL对于数据科学家和分析师来说更容易编写查询语句。MongoDB的查询语言更偏向于开发者，它们在程序里嵌入自己的查询代码。两个模型各有自己的优点和缺点，有时候还取决于个人喜好。

许多关系型数据库支持数据分析（或者数据仓库），而不仅仅是作为数据库。通常数据可以大量导入数据库，然后通过分析来支持商务智能决策问题。这个领域被HP Vertica 或Teradata Database大公司垄断了，它们都可以提供水平伸缩的SQL数据库。

现在通过在Hadoop存储的数据上运行SQL查询来分析数据的情况越来越多。Apache Hive就是一个广泛使用的工具，可以把SQL查询语句转换为Map-Reduce的工作，这提供了一种伸缩性的方式来查询大的数据集。这些查询使用了关系型模型，但是只针对慢速的分析查询，而不是应用程序内置的查询。

文档数据库

很少有数据库标示自己是文档数据库。在编写本书时，对比MongoDB最近的开源数据库是Apache的CouchDB。CouchDB的文档模型与MongoDB的有些相似，虽然数据使用了原始的JSON格式，而MongoDB使用了BSON格式。与MongoDB类似，CouchDB支持辅助索引；不同点在于，CouchDB的索引通过编写mapreduce函数代码实现，相比MySQL 和 MongoDB，其进程不仅仅使用的是声明式语法。在伸缩性方面也不一样，CouchDB不支持跨机器分区；相反，CouchDB节点只是其他节点的完整的复制。

1.4.2 使用场景和部署

坦率地说，你不会只根据数据库特性来选择数据库。我们需要知道真实行业里的成功使用案例。我们来看下MongoDB广泛定义的使用案例，以及一些生产环境下使用的例子[1]。

Web 应用

MongoDB非常适合做Web应用的主存储数据库。即使是简单的Web应用，也需要使用许多数据模型，比如管理用户、会话、App专有数据、上传以及权限等。这也可以与关系型数据库提供的表格方法很好地兼容，它也可以从MongoDB的集合和文档模型里获取好处。因为文档模

[1]最新的 MongoDB 企业名单，参见 http://mng.bz/z2CH。Google,facebook，BAT，360,，新浪微博、摩拜单车等

型可以表示更丰富的数据结构，需要的集合数量肯定比关系型数据库表的数量要少很多，因为关系型数据库的表需要规范化设计范式。此外，动态查询和索引让我们可以轻易实现绝大多数SQL支持的查询功能。最后，随着Web应用的增长，MongoDB提供了更加清晰的伸缩路径。

MongoDB可以很好地解决高吞吐量的Web网站需求，例如The Business Insider (TBI)案例。它们从2008年1月就开始用MongoDB作为主要的存储库。TBI是一个新闻网站，它每天的吞吐量超过100万独立的页面浏览量。有意思的是，除处理网站主要的内容（文章、评论、用户等）以外，MongoDB还要处理和存储实时的分析数据。这些分析数据被TBI用来生成动态心跳地图，以显示不同新闻的点击量。

敏捷开发

无论你怎么看待敏捷开发，都无法否认快速构建应用系统的热情。许多开发团队，包括Shutterfly和纽约时报，都部分选择了MongoDB，因为它们可以比关系型数据库更快速地开发应用。另外一个显著的原因就是MongoDB没有固定的数据架构定义schema，所以大量节约了花费在提交、沟通和应用schema修改上的时间。

此外，花费在将数据关系表示推进到面向对象数据模型中和优化ORM框架生成的SQL语句工作上的时间大大减少了。因此，MongoDB通常适用于实现较短的开发周期的项目，以及敏捷、中型规模的团队。

分析和日志

我们之前说过MongoDB也适用于分析和日志，而且使用MongoDB分析的应用还在增长。通常情况下，一个完善的公司会开始考虑使用MongoDB来做特殊的App分析工作。这些公司包括GitHub、Disqus、Justin.tv、Gilt Groupe等。

MongoDB的相关性分析得益于它的速度和两个特性：针对性原子更新和盖子集合。原子更新允许客户端高效地增加计数器，而且把值推进数组里。盖子集合对于日志非常有用，因为它只存储最近的文档数据。存储日志数据在数据库里与之对应的是文件系统，提供了更简单的组织和更强的查询功能。现在，用户可以使用MongoDB查询来获取日志信息，而不是`grep`或者自定义搜索工具。

缓存

许多Web应用使用缓存层来帮助快速返回内容数据。允许支持更多对象结构的数据模型（可

以把任意文档存储到MongoDB而不需要担心数据结构），结合更快的查询速度，经常允许MongoDB可以作为支持更多查询功能的缓存使用，或者直接与缓存层集成到一起。以The Business Insider网站为例，可以抛弃Memcached，直接从MongoDB处理页面请求。

可变的 Schema

你可以从https://dev.twitter.com/rest/tools/console获取一些JSON数据的例子，只要知道如何使用它即可。在获取数据后，保存为sample. Json文件，可以使用如下方式把它导入MongoDB数据库里：

```
$ cat sample.json | mongoimport -c tweets
2015-08-28T11:48:27.584+0300     connected to: localhost
2015-08-28T11:48:27.660+0300     imported 1 document
```

也可以下载一些Twitter流的例子数据，然后直接导入MongoDB集合里。因为流生成了JSON文档，在发给数据库之前不需要修改数据。Mongoimport工具可以直接把数据翻译给BSON。这意味着每个推文微博都会按照自己的格式存储，以一个独立文档存储在集合中。索引和查询内容时，不需要提前声明数据的结构。

你的应用需要调用JSON API时，有这样一个可以自动转换JSON的系统是非常棒的。在存储数据之前不用知道数据结构，而且MongoDB缺少schema约束，因此可以简化我们的数据模型。

1.5 提示和限制

Tips and limitations

对于所有这些好的功能特性，值得记住的是系统权衡和限制。在大家使用MongoDB开发项目之前，我们要先强调一些限制。这些知识都是关于MongoDB如何管理数据，以及如何使用memorymapped文件在磁盘和内存之间移动数据的。

首先，MongoDB通常用于64位系统。32位系统只能寻址4GB内存。这意味着只要你的数据集，包括元数据和存储库达到4GB，MongoDB就无法存储额外的数据。绝大部分生产环境需要的内存比这大，所以64位系统是必须的[1]。

使用虚拟内存映射的第二个结果是数据内存会根据需要自动分配。这使得在共享环境中运行数据库变得复杂。通常对于数据库服务器，MongoDB最好运行在专门的服务器上。

[1] 16EB 的内存，几乎可以满足所有的需求和目标。中国 MongoDB 学习交流群 511943641

或许关于MongoDB使用内存映射文件最重要的知识就是它在底层如何处理超过内存大小的数据集。当查询如此大的数据集时，它通常需要通过访问磁盘来获取额外的数据。结果是许多用户报告卓越的MongoDB性能，直到处理的数据超出了内存，而且查询开始变慢。这个问题不仅仅存在于MongoDB中，它也是一个常见的陷阱，值得去留意。

一个关联的问题是MongoDB用来存储集合和文档的数据结构，从数据大小的角度来看并非是最有效的。例如，MongoDB在每个文档里存储key，这意味着对于每个包含“username”字段的文档，都必须使用8个字节来存储该字段的名称。

对于SQL开发者，使用MongoDB常见的痛苦就是，它的查询语言与SQL差别很大，而且有时候确实是事实。MongoDB比绝大多数数据库更针对开发人员——不是分析员。它的哲学是查询一次编写，然后嵌入应用中。正如你将要看到的，MongoDB查询通常由JSON对象组成而不是SQL文本。这使得查询更容易创建和解析，这是个重要的考虑，但是很难为ad-hoc查询去改变。如果你是个分析员，每天要编写查询语句，你会愿意选择使用SQL语句。

最后值得一提的是，虽然MongoDB是最简单的数据库之一，可以作为单个节点运行，但是运行大规模集群还是有维护的成本的。大部分分布式数据库都有类似的问题，而MongoDB更是如此，因为它的集群需要三个配置节点来单独处理分片集群的复制问题。

在一些数据库，例如HBase中，数据存储到每个片中，每个分片数据可以在集群的任意集群上复制。MongoDB不会在所有分片节点上复制数据，而是在每个可复制集群里复制数据。分片和可复制集群是单独的概念，这样有特殊的优势，但是也意味着在配置MongoDB集群时需要单独配置和管理。

我们来快速看下MongoDB里的其他改变。

1.6 MongoDB 历史

当第一版《MongoDB in Action》出版时，MongoDB 1.8.x是最稳定的版本，2.0.0版本刚刚开始启动。对于本书的第二版，3.4.x是最稳定的版本[1]。

官方每次重大修改的版本列表如下所示。你最好使用最新的稳定版本，如果是这样，你可以忽略本列表。否则，这个列表可以帮助你决定自己的版本和本书内容的不同。这绝不是一个详尽的列表，因为篇幅限制，我们只列举了每个发布的4～5个项目。

[1] MongoDB 实际上从 2.6 直接跳到 3.0，忽略了 2.8。参考 http://www.mongodb.com/blog/post/announcing-mongodb-30，获取更多关于 3.4 的信息。

版本 1.8.X

(官方不再支持)

- 分片——分片集群由实验状态修改为产品环境准备状态。
- 可复制集——可复制集状态为产品环境准备。
- 可复制对弃用——可复制集对不再被MongoDB公司支持。
- GEO搜索——引入二维GEO索引（坐标系、2D索引）。

版本 2.0.X

(官方不再支持)

- 默认弃用日志——新版本默认弃用日志功能，日志是阻止数据冲突的重要功能。
- 查询——此版本增加了`$and`查询操作符来完善`$or`操作。
- 稀疏索引——之前的MongoDB保护每个文档的索引节点，即使文档部包括索引跟踪的字段。稀疏索引只添加包含相关字段的文档节点。这个功能显著降低了索引的大小。某些情况下还可以改善索引的性能，因为小索引可以更有效地使用内存。
- 可复制集优先级——这个版本允许指定可复制集中服务器的优先级，以便于选择新的主服务器。
- 集合级别的压缩和修复——之前的版本只能执行在单个数据库上压缩和修复；这次已扩展到单个的集合中。

版本 2.2.X

(官方不再支持)

- 聚合框架——这个改变使得数据分析和转换更加简单、高效。从某些方面而言，这个工具代替了map/reduce的部分工作；它是基于管道构建，而不是map/reduce模型（难以理解掌握）。
- TTL集合——引入了带有生命周期的集合，允许我们创建与Memcached类似的缓存模型。
- DB级别锁——此版本添加了数据库级别的锁来代替全局锁,它通过允许多个操作同时在不同的数据库发生来改善写并发。
- 标签识别分片——此版本允许节点可以使用ID来标识数据存储的物理位置。这样的应用可

以控制数据存储在集群中的位置，因此提升效率（只读节点部署在同一个数据中心），减少协作管理的问题（只能在某个国家的服务器上存储该国家需要的数据）。

版本 2.4.X

(最老的稳定版本)

- 企业版——MongoDB的第一个订阅者版本，包括额外的验证模块，可以使用Kerberos验证系统来管理登录数据。免费版包含企业版其他的所有功能。
- 聚合框架性能——改进聚合框架的性能来支持实时分析。第6章会详细介绍聚合框架。
- 文本搜索——企业级的搜索方案作为MongoDB的实验特性集成进来。第9章会介绍这个新的搜索功能。
- 增强GEO地理位置索引——此版本包括支持多边形交叉查询和GeoJSON，以及球星模型的改进，支持椭球模型。
- V8 JavaScript引擎——MongoDB以及从Spider Monkey JavaScript引擎切换到Google V8引擎，这改进了多线程操作，并且提升了基于JavaScript的MongoDB map/reduce系统性能。

版本 2.6.X

(稳定发布)

- `$text`查询——此版本添加了`$text`操作符来支持正常查询中的文本搜索。
- 聚合改进——此版本中聚合有很大的改进。可以在游标上流处理数据，也可以输出数据到集合中。除了其他特性和性能改进，还有许多新增的操作符和管道阶段。
- 为写入改进wire协议——现在大量写入将会受到更细粒度的应答。批量写入中幸亏有了每次写入的成功或者失败状态，使得写入错误可以通过网络返回给客户端。
- 新更新操作符——已经为更新操作符添加了`$mul`，它可以乘以要更新的值。
- Sharding改进——为了更好地处理特定的情况，已经改进了分片集群特性。连续块可以合并，而且重复数据留下来等到数据块迁移完成后自动清理干净。
- 安全改进——此版本支持集合级别的访问控制，还有用户角色定义。另外还改进了SSL和X509证书支持。
- 查询系统改进——查询系统的许多部分都被重构过了。这改进了性能和查询的可预测性。

- 企业模块——MongoDB企业模块改进并扩展了已有的功能，还有审计支持。

版本 3.4.6

(最新的稳定发布版本，3.6版本9月发布)

- MMAPv1存储引擎选择支持集合级别的锁。
- 可复制集选择可以有50个成员。
- 支持WiredTiger存储引擎；WiredTiger只有在MongoDB 3.0以后的64位版本可用。
- WiredTiger 3.0存储引擎提供了文档级别的锁和压缩功能。
- 可拔插存储引擎API允许第三方开发MongoDB存储引擎。
- 改进了解释功能。
- SCRAM-SHA-1验证机制。
- `ensureIndex()`函数被`createIndex()`取代，不应该再使用。

1.7 其他资源

Additional resources

本书的目标是作为教程和权威参考，所以许多语言都是为了介绍主题，并且详细地介绍这些概念。如果需要纯参考，最好的资源是MongoDB的用户手册http://docs.mongodb.org/manual。这是一个深入的数据库指南，在需要复习这些概念的时候非常有用，而且我们强烈推荐官方文档。

如果你遇到特别的MongoDB问题，可能其他人已经知道了。简单的搜索可能都会返回结果，像博客或者网站Stack Overflow (http://stackoverflow.com) ——全球最大的面向技术问答的网站。在遇到问题的时候这些都是很大的帮助，但是在用于自己的MongoDB之前要详细检查下答案。

你也可以从MongoDB IRC聊天组和用户论坛获取帮助。MongoDB公司也提供了咨询服务来帮助企业使用MongoDB数据库。许多城市有自己的MongoDB用户组，可以通过http://meetup.com查询。还有一些方式比如接触熟悉MongoDB的人来学习如何使用数据库。最后，你也可以在曼宁出版社论坛直接联系我们，《MongoDB in Action》专门的论坛为http://manning-sandbox.com/forum.jspa?forumID=677。在这里可以咨询本书中可能没有介

绍的难题，也可以指出漏洞和勘误表。发表问题吧，请不要犹豫！

1.8 总结

Summary

我们已经介绍了许多内容。总结一下，MongoDB是一个开源的、面向文档的数据库管理系统，为全新的互联网应用的数据模型和伸缩性而设计，具有动态查询和辅助索引、快速原子更新以及复杂聚合，支持自动化灾备的复制，还有水平伸缩的分片集群等特性。

虽然知识点很多，但是如果你仔细阅读，你可能现在已经急不可耐要为这些功能写代码了。介绍数据库新特性只是其中一个任务，还要在实际中使用它。幸运的是，这就是你在下两章里要学习的知识。首先，大家要熟悉MongoDB JavaScript shell，它可以很方便地与数据库服务引擎交互。其次，在第2章，会开始试验驱动，并尝试构建简单的基于MongoDB的Ruby程序。

第 2 章

通过JavaScript shell操作MongoDB

MongoDB through the JavaScript shell

本章内容

- MongoDB shell里使用CRUD操作
- 构建索引和使用explain()
- 掌握基本管理
- 获取帮助

前一章提到了MongoDB运行的经验。如果想学习更多关于实战操作的介绍，本章内容就是。使用MongoDB shell，本章通过一系列实际操作来教会大家基本的数据库知识。你将会学习如何新增、读取、更新和删除(CRUD)文档，在这个过程中熟悉MongoDB查询语言。此外，我们还会初步学习数据库索引，以及如何使用它们来优化查询。然后我们将会学习一些基本的管理命令，建议一些使用MongoDB shell获取帮助的方式。可以把本章作为已经介绍的概念的详细阐述和MongoDB shell操作实战。

MongoDB shell是实验数据库的帮助工具，可以运行ad-hoc查询、管理MongoDB实例。当使用MongoDB开发应用时，我们会使用语言驱动而不是shell，但是shell可以用来做测试和重构这些查询。任意的MongoDB查询都可以在shell里运行。

如果你是MongoDB shell新手，也不要担心，记住它提供了你期望的所有功能；它允许你检查和操作数据，以及管理数据库服务器。MongoDB shell在查询语言方面不同于其他数据库，它没有采用标准的SQL查询语言，我们可以使用Javascript语言和简单的API操作数据库。这意味着你可以在shell里编写脚本与MongoDB数据库交互。如果不熟悉Javascript，也没有关系，只需要简单的语法就可以使用shell的功能，本章里所有的例子都会详细解释。MongoDB shell里提供的API与各个语言驱动里提供的接口一样，所以shell里编写的查询代码很容易移植到自己的应用程序代码里。

如果跟着例子进行操作，就会收获很大、受益匪浅，但是必须先安装MongoDB。安装步骤可以在附录A里找到。

2.1 深入 MongoDB shell

Diving into the MongoDB shell

用MongoDB JavaScript shell操作数据库非常简单，而且能够对于文档、集合和数据库查询语言有个实际的感知。下面对MongoDB进行详细介绍。

我们将会从获取和运行shell开始，然后学习用JavaScript如何表示文档，学习如何插入这些文档到MongoDB集合中。要检验这些插入命令，就需要实际查询集合数据、更新操作。最后，通过学习删除数据和集合来完成CRUD操作的练习。

2.1.1 启动 shell

参考附录A，可以很快在本机安装一个可以工作的MongoDB服务和一个运行的`mongod`实例。通过运行`mongo`执行文件启动MongoDB shell：

```
mongo
```

如果shell程序启动成功，你的屏幕看起来就会如图2.1所示。shell顶部显示的是运行的MongoDB服务器的版本信息，后面是选择连接的当前数据库信息。

图 2.1 启动 MongoDB JavaScript shell

如果你熟悉JavaScript，就可以开始输入代码，然后探索学习shell。或者，看看如何运行MongoDB数据库的第一个操作命令。

2.1.2 数据库、集合和文档

正如我们知道的，MongoDB把数据存储在文档中，数据可以以JSON (JavaScript Object Notation)格式输出。你可能喜欢在不同的地方存储不同类型的文档，比如users 和 orders。

这意味着MongoDB需要一种方式来分类文档，这与关系型数据库RDBMS中的表类似。在MongoDB数据库中，我们称之为集合（collection）。

MongoDB把集合分别存储在不同的数据库中。与传统的SQL数据库不同，MongoDB的数据库只区分集合的命名空间。要查询MongoDB数据库，需要知道存储文档数据的数据库和集合的名字。如果开始没有指定数据库，shell会选择默认的test数据库。为了与后面的练习例子统一，我们切换到tutorial数据库：

```
> use tutorial
switched to db tutorial
```

你会看到一个切换数据库成功的提示信息。

为什么MongoDB有数据库和集合？答案取决于MongoDB如何在磁盘上写数据。数据库中索引的集合都分组在相同的文件中，所以从内存的角度说，这是合理的，保证相关的结合在同一个库里。你可能还希望不同的应用访问同一个集合（多租户），而且保持数据组织性很有帮助，以备未来需求。

创建数据库和集合

你可能好奇，怎样才能切换到tutorial数据库而不需要显示创建它。其实创建数据库不是必须的。只有在第一次插入数据库和集合时才会创建。这个行为符合MongoDB动态操作数据的模式。正如文档的数据结构不需要提前定义一样，单个的数据库和集合也可以在运行时创建。

这样可以简化并加速开发过程。也就是说，你不用担心意外创建数据库或者集合，绝大部分驱动都会阻止这种事情发生。

现在是时候创建第一个文档了。因为使用JavaScript shell，文档使用JSON格式指定，所以简单的文档定义如下所示：

```
{username: "smith"}
```

这个文档包含了一个简单的key和value，用来存储smith的用户名。

2.1.3 插入和查询

要插入文档，就需要选择一个目标集合。我们恰如其分地选择`users`集合。下面是插入代码：

```
> db.users.insert({username: "smith"})
WriteResult({ "nInserted" : 1 })
```

注意：例子中，我们选择MongoDB shell命令开头带个>符号，这样可以区分输出结果。

你会注意到，在输入这个命令后会出现一些延迟。此刻，既没有在磁盘上创建tutorial数据库，也没有创建users集合。延迟的原因是要为二者分配初始化文件。

如果插入成功，就已经成功保存了第一个文档。在默认的MongoDB数据库配置里，确保已插入这个数据，即使你关闭shell或者重启机器。你可以使用查询来查看新文档数据：

```
> db.users.find()
```

因为数据是users集合的一部分，所以重新打开shell，运行查询，就会显示如下的结果。应答消息如下所示：

```
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

MONGODB 的_ID 字段

注意到_id字段已经默认添加到文档里。我们可以把_id字段的值作为文档的主键。每个MongoDB文档都需要一个_id，而且如果创建文档时没有_id，就会专门创建一个MongoDB ObjectID添加到文档里。出现在控制台中的ObjectID与代码里列举的不同，但是在集合中_id值的作用是唯一的，这是基本要求。可以在文档里插入自己的_id，ObjectID是MongoDB默认的。

下一章会介绍更多的关于ObjectID的知识。我们选择继续添加第二个用户user到集合中：

```
> db.users.insert({username: "jones"})
WriteResult({ "nInserted" : 1 })
```

现在集合中有2个文档了。继续通过运行count来验证下结果：

```
> db.users.count()
```

传递查询条件

现在集合里既然有1个以上文档了，我们来看下更复杂的查询。和前面一样，我们先来查询集合里所有的文档：

```
> db.users.find()
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

我们也可以给find方法传递简单的查询选择权。查询选择器是用来匹配集合中文档的。要查

询集合中username 为jones的数据，可以传递简单的条件，语句如下：

```
> db.users.find({username: "jones"})
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

这个查询条件{username: "jones"}会返回索引名字为jones的文档数据，它会迭代所有的文档。

注意，调用find方法若不传递参数，就等价于传递空条件。也就是说，`db.users.find()`和`db.users.find({})`的效果一样。

当然也可以在查询语句中指定多个字段，这样隐式创建AND语句。例如，可以使用下面的选择器查询：

```
> db.users.find({
... _id: ObjectId("552e458158cd52bcb257c324"),
... username: "smith"
... })
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

程序中的三个点是MongoDB shell自动添加的，表示这是单行命令。

查询条件会返回等价的文档。条件会使用AND，也就是按并且关系进行查询，索引必须匹配`_id`和`username` 字段。

也可以使用MongoDB的`$and`操作符。之前的查询语句等价修改为

```
> db.users.find({ $and: [
... { _id: ObjectId("552e458158cd52bcb257c324") },
... { username: "smith" }
... ] })
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

使用OR查询文档的语法类似：只需要把`$or`操作符替换一下就可以了。思考下面的语句：

```
> db.users.find({ $or: [
... { username: "smith" },
... { username: "jones" }
... ]})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
{ "_id" : ObjectId("552e542a58cd52bcb257c325"), "username" : "jones" }
```

这个查询条件会返回smith和jones的文档，因为我们要找的是名字等于smith或jones的数据。

这个例子和之前的不同，因为它不是简单地插入或查找一个数据文档，而是查询文档本身。在MongoDB里，使用文档来表示命令的做法很普遍，如果你习惯关系型数据库，则可能会感到吃惊。这种做法的一个好处就是，它非常容易在应用中构建自己的查询，因为它们是文档而不是SQL字符串。

我们已经看了基本的创建和读取数据的操作。现在是时候来看一下如何更新数据了。

2.1.4 更新文档

所有的更新至少需要2个参数。第一个指定要更新的文档，第二个定义要如何修改此文档。第一个例子演示了如何修改单个文档，第二个例子演示了如何修改多个文档，甚至如本节的末尾部分一样是集合中的所有文档。但是要记住，默认`update()`只更新一个文档。

通常有两种类型的更新操作，用于不同的属性和使用场景。其中一个类型的更新是在一个文档或者多个文档上修改，另外一个是使用新文档取代旧的文档。

从下面的例子我们来看一下简单的文档：

```
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

更新操作符

第一种类型的更新需要传递一个文档参数，还有一些操作符作为第二个参数。本节里，我们先来看下如何使用`$set`操作符，它可以为单个字段设置特定的值。

假设用户smith要增加自己的居住地国家，我们可以使用如下命令更新：

```
> db.users.update({username: "smith"}, {$set: {country: "Canada"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

这个命令会告诉MongoDB，找到一个用户名为`smith`的文档，然后把`country`属性设置为`Canada`。我们可以在服务器返回的消息里看到更新结果。如果查询被修改的数据，就可以看到该文档已经被更新：

```
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith",
     "country" : "Canada" }
```

替换更新

另外一个更新文档的方式就是替换文档，而不是更新某个字段。当使用`$set`操作符的时候这一点容易混淆。思考下面不同的更新命令：

```
> db.users.update({username: "smith"}, {country: "Canada"})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

这个例子里，文档被替换为只包含`country`字段的文档。`username`字段被删除，因为它只

是用来匹配文档，第二个参数用来更新替换。当我们在使用这种更新时应该多加注意。新文档的查询如下所示：

```
> db.users.find({country: "Canada"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "country" : "Canada" }
```

_id相同，但是数据已经被替换为新的文档了。当确定是新增或者修改数据而不是替换整个文档时，就使用$set操作符。

把用户名username重新添加到记录里：

```
> db.users.update({country: "Canada"}, {$set: {username: "smith"}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.users.find({country: "Canada"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "country" : "Canada",
     "username" : "smith" }
```

如果以后不想要country字段了，使用$unset操作符删除即可：

```
> db.users.update({username: "smith"}, {$unset: {country: 1}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.users.find({username: "smith"})
{ "_id" : ObjectId("552e458158cd52bcb257c324"), "username" : "smith" }
```

更新复杂数据

我们来丰富一下以上例子。正如在第1章里看到的，我们使用文档来标识数据。文档可以包含复杂的数据。假设除了保存配置信息，用户还可以保存自己喜欢的东西。符合要求的文档格式可能看起来如下：

```
{
  username: "smith",
  favorites: {
    cities: ["Chicago", "Cheyenne"],
    movies: ["Casablanca", "For a Few Dollars More", "The Sting"]
  }
}
```

favorites键指向了一个包含2个键的新对象，它包含喜欢的城市和电影。假设你已经弄懂了，就可以把之前smith的文档修改为这个格式了。此时应该想起了$set操作符了：

```
> db.users.update( {username: "smith"},
...   {
...     $set: {
...       favorites: {
...         cities: ["Chicago", "Cheyenne"],
...         movies: ["Casablanca", "For a Few Dollars More", "The Sting"]
...     }
...   }
... })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

请注意，间距缩进不是必须的，但是这样可以避免错误，文档的可读性更强。

下面我们来用相似的方式修改jones数据。此时，我们只能添加一些喜欢的电影：

```
> db.users.update( {username: "jones"},
...   {
...     $set: {
...       favorites: {
...         movies: ["Casablanca", "Rocky"]
...       }
...     }
...   })
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
```

如果输入错误，则可以使用向上的方向键回到前一条语句。

选择查询users集合来确保两个更新成功：

```
> > db.users.find().pretty()
{
    "_id" : ObjectId("552e458158cd52bcb257c324"),
    "username" : "smith",
    "favorites" : {
        "cities" : [
            "Chicago",
            "Cheyenne"
        ],
        "movies" : [
            "Casablanca",
            "For a Few Dollars More",
            "The Sting"
        ]
```

```
    }
}
{
    "_id" : ObjectId("552e542a58cd52bcb257c325"),
    "username" : "jones",
    "favorites" : {
        "movies" : [
            "Casablanca",
            "Rocky"
        ]
    }
}
```

严格来说，`find()`命令文档返回一个cursor游标。因此，要访问文档，就需要迭代游标。`find()`命令自动返回`20`个文档——如果可用——就会迭代游标20次。

有了这些例子文档，我们可以开始体验MongoDB查询语言的强大之处了。特别是，查询引擎可以深入内嵌对象的内部，以及根据数据元素进行匹配，这些都是非常有用的功能。注意，我们是怎样通过给`find`操作附加`pretty`操作来获取从服务器端返回的良好格式的结构。严格来说，`pretty()`实际上就是`cursor.pretty()`，它可以配置游标以容易阅读的方式来显示结果。

我们可以在本次查询里看到这两个概念的演示例子，找出所有喜欢电影Casablanca（卡萨布兰卡）的用户：

```
> db.users.find({"favorites.movies": "Casablanca"})
```

favorites和movies之间的原点会告诉查询引擎来搜索一个键名为`favorites`的对象，内部键名为`movies`作为新的匹配条件。因此，这个查询返回2个用户文档，如果数组中的元素匹配了最初查询，数组上的查询将会匹配。

假设你知道任意喜欢Casablanca的用户也会喜欢电影The Maltese Falcon，而且想要用更细数据库来反映这个事实，那么就要看更复杂的查询。如何在MongoDB更新命令里实现呢？

高级更新

你可能想到再次使用`$set`操作符，但是这样做需要重新编写并发送整个电影数组。因为我们想要做的就是给列表添加元素，最好还是使用`$push`或者`$addToSet`。这2个命令都是往数组中添加数据，但是第二个是唯一的，阻止了重复的数据。

这就是你要的更新语句:

```
> db.users.update( {"favorites.movies": "Casablanca"},
...     {$addToSet: {"favorites.movies": "The Maltese Falcon"} },
```

```
...             false,
...             true )
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 2 })
```

这个代码容易理解。第一个参数是查询条件，匹配电影列表中包含Casablanca的用户。第二个参数使用`$addToSet`添加The Maltese Falcon到列表中。

第三个参数`false`，控制是否允许upsert。这个命令告诉更新操作，当一个文档不存在的时候是否插入它，这取决于更新操作是操作符更新还是替换更新。

第四个参数`true`，表示是否是多个更新。默认情况下，MongoDB更新只针对第一个匹配文档。如果想更新所有匹配的文档，就必须显示指定这个参数。如果你想对`smith`和`jones`都更新数据，那么这个参数是必须的。

我们将会详细介绍更新，但是在继续学习之前先练习一下这些例子。

2.1.5 删除数据

现在我们已经学习了通过MongoDB Shell 创建、读取和更新数据的基本操作。最后我们来学习下最简单的操作：删除数据。

如果没有参数，删除操作将会清空集合里的所有文档。如果要清空`foo`集合里的所有文档内存，可以输入以下命令：

```
> db.foo.remove()
```

通常我们只需要删除集合中某个文档，因此，我们要传递查询选择器给`remove()`方法。如果要删除所有喜欢Cheyenne城市的用户，则可以这样编写简单的表达式：

```
> db.users.remove({"favorites.cities": "Cheyenne"})
WriteResult({ "nRemoved" : 1 })
```

注意：`remove()`操作不会删除集合，它只会删除集合中的某个文档。我们可以把它和SQL中的DELETE命令进行类比。

如果要删除集合及其附带的索引数据，可以使用`drop()`方法：

```
> db.users.drop()
```

创建、读取、更新和删除都是任意数据库的基本操作。如果你已经用过，那么可以直接在MongoDB里进行CRUD的操作练习。下一节里，我们将会学习如何通过辅助索引来增强查询、修改和删除。

2.1.6 shell 的其他特性

你可能已经注意到了，shell可以很方便地做很多与MongoDB有关的工作。我们可以使用上下方向键快速切换之前的命令，使特定的输入自动完成，比如集合名字。自动完成特性使用tab键来自动完成或者列举完成的可能情况[1]。

[1]完整的快捷键列表，请参考 http://docs.mongodb.org/v3.0/reference/program/mongo/#mongo-keyboard- shortcuts。

我们也可以通过输入help来获取更多关于shell的帮助信息：

```
> help
```

许多函数也可以打印帮助信息，用来提示如何使用函数。试一下：

```
> db.help()
DB methods:
    db.adminCommand(nameOrDocument) - switches to 'admin' db, and runs
    command [ just calls db.runCommand(...) ]
    db.auth(username, password)
    db.cloneDatabase(fromhost)
    db.commandHelp(name) returns the help for the command
    db.copyDatabase(fromdb, todb, fromhost)
⋮
```

查询的帮助通过一个不同的函数`explain`来提供。我们会在后面详细讲解。在启动MongoDB shell时还有许多参数选项可用。要显示这个列表，可以在启动MongoDB shell时指定help标签：

```
$ mongo --help
```

你不需要担心怎样使用这些功能特性，我们还没有详细介绍shell呢！但是它确实可以提供我们需要的帮助信息。

2.2 使用索引创建和查询

Creating and querying with indexes

通过创建索引来改善查询性能是常见的做法。幸运的是，MongoDB的索引可以方便地从shell创建。如果你还不了解数据库索引，则本节可以帮你弄清楚这些概念；如果你应经使用过索引，就会发现创建索引非常简单。使用`explain()`方法监控查询。

2.2.1 创建大集合

索引只有在集合包含许多文档的时候才有意义。我们先往`numbers`集合里添加`20000`个文档。因为MongoDB shell也是个Javascript解析器，代码实现起来非常简单：

```
> for(i = 0; i < 20000; i++) {
    db.numbers.save({num: i});
  }
WriteResult({ "nInserted" : 1 })
```

因为文档很多，所以如果花费了几秒钟也不要惊讶。一旦返回，我们可以运行一些查询语句来检验所有显示的文档：

```
> db.numbers.count()
20000
```

```
> db.numbers.find()
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830a"), "num": 0 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830b"), "num": 1 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830c"), "num": 2 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830d"), "num": 3 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830e"), "num": 4 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac830f"), "num": 5 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8310"), "num": 6 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8311"), "num": 7 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8312"), "num": 8 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8313"), "num": 9 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8314"), "num": 10 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8315"), "num": 11 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8316"), "num": 12 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8317"), "num": 13 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8318"), "num": 14 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8319"), "num": 15 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831a"), "num": 16 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831b"), "num": 17 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831c"), "num": 18 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831d"), "num": 19 }
Type "it" for more
```

`count()`命令显示已经插入了20000个文档。下一个查询显示了前20条结果（shell里的结果可能不太一样）。

我们可以使用`it`命令来显示其余的结果：

```
> it
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831e"), "num": 20 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac831f"), "num": 21 }
{ "_id": ObjectId("4bfbf132dba1aa7c30ac8320"), "num": 22 }
⋮
```

`it`命令告诉shell返回下一个结果集[1]。

既然已经有了大量的测试数据，我们来尝试一些新的查询。假设你已经熟悉了MongoDB的查询引擎，则匹配文档属性`num`的查询语句如下：

```
> db.numbers.find({num: 500})
{ "_id" : ObjectId("4bfbf132dba1aa7c30ac84fe"), "num" : 500 }
```

范围查询

更有意思的是，你可以使用`$gt`和`$lt`运算符来进行范围查询。这两个符号表示大于和小于。假如你要查询所有`num`值大于19995的所有文档，则语句如下：

```
> db.numbers.find( {num: {"$gt": 19995 }} )
```

[1]你可能好奇背后发生的事情。所有的查询都创建一个游标，以便于在结果集上迭代。这是使用 shell 时候看不到的工作，所以此时没有必要讨论太详细。如果你等不及学习游标及其特性，那可以直接阅读第 3 章和第 4 章的内容。

```
{ "_id" : ObjectId("552e660b58cd52bcb2581142"), "num" : 19996 }
{ "_id" : ObjectId("552e660b58cd52bcb2581143"), "num" : 19997 }
{ "_id" : ObjectId("552e660b58cd52bcb2581144"), "num" : 19998 }
{ "_id" : ObjectId("552e660b58cd52bcb2581145"), "num" : 19999 }
```

可以结合使用两个运算符，设置上下限：

```
> db.numbers.find( {num: {"$gt": 20, "$lt": 25 }} )
{ "_id" : ObjectId("552e660558cd52bcb257c33b"), "num" : 21 }
{ "_id" : ObjectId("552e660558cd52bcb257c33c"), "num" : 22 }
{ "_id" : ObjectId("552e660558cd52bcb257c33d"), "num" : 23 }
{ "_id" : ObjectId("552e660558cd52bcb257c33e"), "num" : 24 }
```

可以使用简单的JSON文档查看，你也可以用与SQL一样的方式来指定范围。`$gt`和`$lt`是MongoDB查询语言里两个主要的运算符。`$gte`表示大于等于，`$lte`表示小于等于，而`$ne`表示不等于。我们会在后续的章节里看到更多的运算符。

当然，除非这种查询非常高效，否则也没有价值。下一节，我们会思考如何通过MongoDB的索引功能来提升查询效率。

2.2.2 索引和 explain( )

如果你使用过关系型数据库，则可能对于SQL的`EXPLAIN`非常熟悉了，它是一个调试和优化查询的好工具。当所有数据库接收到查询后，它必须弄清楚如何执行查询；这就称为查询计划。`EXPLAIN`描述了查询路径并且允许开发者通过确定查询使用的索引来诊断慢的查询语句。查询通常可以有多种方式执行，但有时候并非我们期望的方式[1]。MongoDB 有自己的`EXPLAIN`版本，它可以提供相同的功能。

要了解其工作原理，我们来分析一下刚才已经使用的查询语句。在系统里运行下面的命令：

```
> db.numbers.find({num: {"$gt": 19995}}).explain("executionStats")
```

结果应该与列表2.1里显示的类似。`"execution-Stats"`关键字是MongoDB 3.0新增的，请求不同的模式并输出更详细的信息。

列表2.1 未使用索引的典型的explain("executionStats")输出结果

```
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "tutorial.numbers",
        "indexFilterSet" : false,
        "parsedQuery" : {
            "num" : {
```

[1]【译者注】SQL Server 有执行计划，可视化工具，可以优化分析 SQL 语句的性能。

```
                "$gt" : 19995
            }
        },
        "winningPlan" : {
            "stage" : "COLLSCAN",
            "filter" : {
                "num" : {
                    "$gt" : 19995
                }
            },
            "direction" : "forward"
        },
        "rejectedPlans" : [ ]
    },
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 4,
        "executionTimeMillis" : 8,
        "totalKeysExamined" : 0,
        "totalDocsExamined" : 20000,
        "executionStages" : {
            "stage" : "COLLSCAN",
            "filter" : {
                "num" : {
                    "$gt" : 19995
                }
            },
            "nReturned" : 4,
            "executionTimeMillisEstimate" : 0,
            "works" : 20002,
            "advanced" : 4,
            "needTime" : 19997,
            "needFetch" : 0,
            "saveState" : 156,
            "restoreState" : 156,
            "isEOF" : 1,
            "invalidates" : 0,
            "direction" : "forward",
            "docsExamined" : 20000
        }
    },
    "serverInfo" : {
        "host" : "rMacBook.local",
        "port" : 27017,
        "version" : "3.0.6",
        "gitVersion" : "nogitversion"
    },
    "ok" : 1
}
```

看到explain()的输出结果[1]，你可能感到惊讶，居然查询引擎会扫描整个集合（即20000个文档）(docsExamined)，而只返回了4个结果。totalKeysExamined字段显示了整个扫描的索引

[1]这些例子里我们插入“hostname”作为机器主机名。在你的机器上可能是 localhost，或者机器名或者名字加上.local。不要担心与我们的结果不同，这和平台的版本还有 MongoDB 版本有关系。

数量，它的值是0。

扫描文档的数量和低效查询的数量居然差距这么大。现实的情况是集合和文档可能更大，实际查询消耗的时间可能会更多，超过8 ms（不同的机器可能不太一样）。

集合需要的就是一个索引。我们可以使用createIndex()方法为num 键创建索引。可以输入以下命令：

```
> db.numbers.createIndex({num: 1})
{
    "createdCollectionAutomatically" : false,
    "numIndexesBefore" : 1,
    "numIndexesAfter" : 2,
    "ok" : 1
}
```

在MongoDB 3.0以后的版本，`createIndex()`方法取代了`ensureIndex()`方法。如果你使用的是旧版本的MongoDB，则可以继续使用`ensureIndex()`而不用`createIndex()`。在MongoDB 3中，`ensureIndex()`依然可用，它是`createIndex()`的别称。

对于其他的MongoDB操作，比如查询和更新，要传递文档给createIndex()方法来确定索引的键。此时{num: 1}文档表示应该为numbers集合中的所有文档的num键建立升序索引。

我们可以通过`getIndexes()`方法来检验索引是否创建成功：

```
> db.numbers.getIndexes()
[
    {
        "v" : 1,
        "key" : {
            "_id" : 1
        },
            "name" : "_id_",
            "ns" : "tutorial.numbers"
        },
        {
            "v" : 1,
            "key" : {
            "num" : 1
        },
        "name" : "num_1",
        "ns" : "tutorial.numbers"
    }
]
```

集合现在有2个索引。第一个是标准的`_id`索引，自动为每个集合构建的；第二个是我们自己创建的`num`索引。这些索引的名字分别叫`_id_` 和 `num_1`。如果没有设置名字，MongoDB会自动创建一个有意义的名字给它们。

如果使用`explain()`进行查询，可以看到应答时间有很大改变。具体如列表2.2所示。

列表2.2　explain()输出的索引查询

```
> db.numbers.find({num: {"$gt": 19995 }}).explain("executionStats")
{
    "queryPlanner" : {
        "plannerVersion" : 1,
        "namespace" : "tutorial.numbers",
        "indexFilterSet" : false,
        "parsedQuery" : {
            "num" : {
                "$gt" : 19995
            }
        },
        "winningPlan" : {
            "stage" : "FETCH",
            "inputStage" : {
                "stage" : "IXSCAN",
                "keyPattern" : {
                    "num" : 1
                },
                "indexName" : "num_1",
                "isMultiKey" : false,
                "direction" : "forward",
                "indexBounds" : {
                    "num" : [
                    "(19995.0, inf.0]"
                ]
            }
        }
    },
    "rejectedPlans" : [ ]
},
    "executionStats" : {
        "executionSuccess" : true,
        "nReturned" : 4,
        "executionTimeMillis" : 0,
        "totalKeysExamined" : 4,
        "totalDocsExamined" : 4,
        "executionStages" : {
            "stage" : "FETCH",
            "nReturned" : 4,
            "executionTimeMillisEstimate" : 0,
            "works" : 5,
            "advanced" : 4,
            "needTime" : 0,
            "needFetch" : 0,
            "saveState" : 0,
            "restoreState" : 0,
            "isEOF" : 1,
            "invalidates" : 0,
            "docsExamined" : 4,
            "alreadyHasObj" : 0,
            "inputStage" : {
                "stage" : "IXSCAN",
                "nReturned" : 4,
```

使用 num_1 索引

返回 4 个文档

只扫描 4 个文档

更快

```
"executionTimeMillisEstimate" : 0,
"works" : 4,
"advanced" : 4,
"needTime" : 0,
"needFetch" : 0,
"saveState" : 0,
"restoreState" : 0,
"isEOF" : 1,
```

```
                "invalidates" : 0,
                "keyPattern" : {
                    "num" : 1
                },
                "indexName" : "num_1",
                "isMultiKey" : false,
                "direction" : "forward",
                "indexBounds" : {
                    "num" : [
                        "(19995.0, inf.0]"
                    ]
                },
                "keysExamined" : 4,
                "dupsTested" : 0,
                "dupsDropped" : 0,
                "seenInvalidated" : 0,
                "matchTested" : 0
            }
        }
    },
    "serverInfo" : {
        "host" : "rMacBook.local",
        "port" : 27017,
        "version" : "3.0.6",
        "gitVersion" : "nogitversion"
    },
    "ok" : 1
}
```

使用 num_1 索引

现在查询使用了num 键的num_1索引，它只扫描了与查询有关的4个文档。整个查询消耗的总时间从8 ms 降低到0 ms!

索引并非没有成本，它们会占用空间，而且会让插入成本稍微提升，对于查询优化来说这是必备的工具。如果对这个例子理解有些疑问，可以阅读本书第8章，第8章专门深入讲解了索引和查询优化机制。接下来我们要学习关于MongoDB实例的管理命令。我们也会学习如何从shell里获取帮助，这有助于我们掌握各种不同的shell命令。

2.3 基本管理

Basic administration

本章承诺要介绍通过JavaScript shell 管理MongoDB。我们已经学习了基本的数据操作和索引。这里，我们将会介绍获取mongod进程信息的方法。例如，你可能想知道不同的集合占用的存储空间，或者集合中定义了多少索引。命令返回的信息可以帮助我们来诊断性能问题并且跟踪分析数据。

我们将了解一下MongoDB的命令接口。最特殊的就是 ,在MongoDB上执行的是非CRUD操作，

从服务器状态检查到数据文件完整性验证，都使用数据库命令来实现。我们将会解释MongoDB上下文里的命令，以及展示它使用起来有多么简单。最后，最好要知道去哪里获取帮助。我们后面将会告诉大家在shell里如何获取深入学习MongoDB的更多帮助。

2.3.1 获取数据库信息

通常，我们想知道安装MongoDB的服务器上存在什么集合和数据库。

幸运的是，MongoDB shell提供了许多命令，包括一些语法糖，用来获取系统的信息。

`show dbs`打印系统中所有的数据库列表信息：

```
> show dbs
admin (empty)
local 0.078GB
tutorial 0.078GB
```

`show collections`展示了当前数据库里所有的集合[1]。如果选择的还是`tutorial`数据库，就会看到这个集合。后面我们还会继续使用这个集合。

```
> show collections
numbers
system.indexes
users
```

有个可能你不认识的集合`system.indexes`。这个是每个数据库都存在的特殊集合。`system.indexes`中的每个入口都定义了一个数据库索引，我们可以使用`getIndexes()`方法来查看，正如之前我们看到的一样。但是MongoDB 3.0放弃了对于`system.indexes`集合的访问，我们可以使用`createIndexes`或者`listIndexes`替换。`getIndexes()` JavaScript方法可以被`db.runCommand( {"listIndexes": "numbers"} )` shell命令取代。

对于低级别数据库和集合分析，`stats()`方法非常有用。当我们在数据库对象上运行此方法时，就会获取下面的结果：

```
> db.stats()
{
    "db" : "tutorial",
    "collections" : 4,
    "objects" : 20010,
    "avgObjSize" : 48.0223888055972,
    "dataSize" : 960928,
```

[1]这个结果里显示的有些信息只有在复杂的调试或者优化时才会有用。但是我们至少也可以了解某个集合以及索引占用的空间。

```
    "storageSize" : 2818048,
    "numExtents" : 8,
    "indexes" : 3,
    "indexSize" : 1177344,
    "fileSize" : 67108864,
    "nsSizeMB" : 16,
    "extentFreeList" : {
        "num" : 0,
        "totalSize" : 0
    },
    "dataFileVersion" : {
        "major" : 4,
        "minor" : 5
    },
    "ok" : 1
}
```

我们也可以在单个集合上执行stats()：

```
> db.numbers.stats()
{
    "ns" : "tutorial.numbers",
    "count" : 20000,
    "size" : 960064,
    "avgObjSize" : 48,
    "storageSize" : 2793472,
    "numExtents" : 5,
    "nindexes" : 2,
    "lastExtentSize" : 2097152,
    "paddingFactor" : 1,
    "paddingFactorNote" : "paddingFactor is unused and unmaintained in 3.0.
    It remains hard coded to 1.0 for compatibility only.",
    "systemFlags" : 1,
    "userFlags" : 1,
    "totalIndexSize" : 1169168,
    "indexSizes" : {
        "_id_" : 654080,
        "num_1" : 515088
    },
    "ok" : 1
}
```

2.3.2 命令如何执行

MongoDB特定的操作集合——与本章目前为止介绍的insert、update、remove、query命令不同——称为数据库命令。数据库命令通常是管理性的，正如刚才介绍的stats()方法，但是也可以控制MongoDB的核心功能，比如更新数据。

无论提供什么功能，所有的数据库命令实现都有个共同点，就是它们在一个叫做$cmd的虚拟集合上实现查询。

要理解这句话，我们来快速看个例子，回忆下我们如何调用stats()命令：

```
> db.stats()
```

stats()方法包装了shell命令的方法调用。

可以尝试输入下面等价的操作：

```
> db.runCommand( {dbstats: 1} )
```

执行后的结果与使用stats()方法显示的一样。注意，这个命令也是使用{dbstats: 1}定义的。通常，我们可以通过给runCommand()方法传递参数来调用任意命令。

以下是说明如何运行集合stats命令：

```
> db.runCommand( {collstats: "numbers"} )
```

输出的结果看起来很熟悉。

要掌握数据库命令的内部原理，我们还要看看runCommand()方法是怎么工作的。这个也不难，因为MongoDB shell将会在无括号调用的时候打印任意方法的实现。

如不使用这种运行命令的方式：

```
> db.runCommand()
```

我们可以执行无括号的版本，来看看内部机制：

```
> db.runCommand
  function ( obj, extra ){
    if ( typeof( obj ) == "string" ){
        var n = {};
        n[obj] = 1;
        obj = n;
        if ( extra && typeof( extra ) == "object" ) {
            for ( var x in extra ) {
                n[x] = extra[x];
            }
        }
    }
    return this.getCollection( "$cmd" ).findOne( obj );
  }
```

最后一行就是在$cmd集合上的查询。恰当的定义为数据库命令就是特殊集合上的查询，$cmd，查询选择器定义了命令本身。这就是背后原理。你能设想一个手动运行统计命令的方法吗？其实非常简单。

```
> db.$cmd.findOne( {collstats: "numbers"} );
```

使用runCommand帮助方法可以更简单，但是通常最好要理解底层的机制。

2.4 获取帮助

Getting help

目前为止，使用MongoDB shell实验数据操作和管理数据库的价值是显而易见的。但是因为可能大家会在shell花费很多时间，所以要知道如何获取帮助。

内置的帮助命令是首选方式。用db.help()也可以打印通常数据库使用的操作方法。通过运行db.numbers.help()，我们可以找到一个相似的方法列表。

也有tab键只能完成功能。开始时输入任意方法的首字母，然后按tab键两次，就会看到所有匹配的方法。下面是所有以get开头的集合操作方法：

```
> db.numbers.get
db.numbers.getCollection( db.numbers.getIndexes(
db.numbers.getShardDistribution(
db.numbers.getDB( db.numbers.getIndices(
db.numbers.getShardVersion(
db.numbers.getDiskStorageStats( db.numbers.getMongo(
db.numbers.getSlaveOk(
db.numbers.getFullName( db.numbers.getName(
db.numbers.getSplitKeysForChunks(
db.numbers.getIndexKeys( db.numbers.getPagesInRAM(
db.numbers.getWriteConcern(
db.numbers.getIndexSpecs( db.numbers.getPlanCache(
db.numbers.getIndexStats( db.numbers.getQueryOptions(
```

官方的MongoDB手册是最好的资源，我们可以在http://docs.mongodb.org找到。它包括教程和参考资料，而且更新到最新的MongoDB版本。文档还包括每个语言的MongoDB驱动实现，比如Ruby驱动，这些驱动在我们使用应用程序访问MongoDB的时候是必须用到的。

如果你还想学习更多内容，而且喜欢使用JavaScript，shell可以让你查看任意的方法实现。例如，假设你想知道save()到底是如何工作的。虽然我们可以通过查看MongoDB源码来达到目的，但是还有更简单的方法，就输入方法名，且不需要括号。下面是大家正常执行save()方法的命令：

```
> db.numbers.save({num: 123123123});
```

以下就是如何来查看实现代码：

```
> db.numbers.save
function ( obj , opts ){
    if ( obj == null )
```

```
        throw "can't save a null";

    if ( typeof( obj ) == "number" || typeof( obj) == "string" )
        throw "can't save a number or string"

    if ( typeof( obj._id ) == "undefined" ){
    obj._id = new ObjectId();
        return this.insert( obj , opts );
    }
    else {
        return this.update( { _id : obj._id } , obj , Object.merge({
    upsert:true }, opts));
    }
}
```

仔细阅读代码，你就会看到`save()`方法只是对`insert()`和`update()`方法的包装。在检查`obj`参数的类型后，如果要保存的对象没有`_id`字段，就会添加字段，然后调用`insert()`，否则就执行更新操作。

检查shell方法的实现非常方便。记住这个技巧，后面扩展学习MongoDB shell的时候非常有用。

2.5 总结

Summary

现在我们已经实际看到了文档的数据模型，而且我们也演示了数据模型上的不同MongoDB操作。我们已经学习了如何创建索引，以及通过`explain()`基于索引的性能优化。此外，我们已经了解如何查询系统上集合和数据库的信息，也知道了聪明的`$cmd`集合，需要帮助时也知道找到合适的方式。

我们学习了许多MongoDB shell的命令，但是这替代不了我们构建真实应用的经验。这也是我们为什么要在下一章里从实验操作转换到真实的数据库开发。我们会学习驱动是如何工作的，然后使用Ruby驱动来构建一个简单的应用，使用真实的数据来操作MongoDB数据库[1]。

[1]【译者注】所有语言的官方驱动都可以在官方网站下载，也可以通过包管理工具搜索 MongoDB。无论是 Node.js 还是 Java、C#都可以找到。对于别的语言的例子，大家也可以自己动手实践，参考官方文档即可：https://docs.mongodb.com/ecosystem/drivers/。我写了 Java 和 C#的例子程序，包括 Helper 工具类。中国 MongoDB 学习交流群 511943641

第 3 章

编写代码操作MongoDB

Writing programs using MongoDB

本章内容

- 通过Ruby介绍MongoDB API
- 理解驱动的工作原理
- 使用BSON和MongoDB网络协议
- 构建完整的例子应用

现在是实战的时候了！虽然实验MongoDB shell还有很多要学习的，但是只有在实际开发过程中使用这个数据库我们才能体会出它的真实价值。这意味着我们要进行编程并且了解MongoDB驱动。正如之前提到的，MongoDB公司提供了针对几乎所有语言的官方支持，Apache许可所有的MongoDB驱动。本书中的例子代码使用了Ruby语言，但是我们演示时使用的方法同样适用于其他语言。本书里我们会使用JavaScript shell演示绝大部分命令，但是通过应用程序访问MongoDB的演示我们将使用Ruby语言。

分三个阶段来学习MongoDB开发编程。第一阶段，学习如何安装MongoDB Ruby驱动，学习基本的CRUD操作。这个过程很快，大家应该也感觉很熟悉，因为API与shell类似。第二阶段会深入学习驱动，了解驱动如何与MongoDB交互。在这个阶段我们将会介绍通常驱动背后的机制。第三阶段，我们通过开发一个简单的Ruby应用来监控Twitter，使用真实的数据集，我们会看到MongoDB如何工作。这个例子也会为第二部分的深入内容奠定基础。

Ruby 新手？

Ruby是一种非常流行、易于阅读的脚本语言。例子代码设计得非常简单，即使不懂Ruby语言的程序员也会从中获益。任何难以理解的Ruby术语我们都在书里做了介绍。如果你想花点时间学习Ruby，可以阅读官方的20分钟教程http://mng.bz/THR3。

Java和C#等主流编程语言也提供了全面的支持，大家可以加QQ群下载例子代码。

3.1 通过 Ruby lens 连接 MongoDB

MongoDB through the Ruby lens

通常，当我们思考驱动时，首先想到的就是低级别的二进制数据以及僵硬的接口。幸运的是，MongoDB驱动不是这样设计的。相反，它设计成非常直观、语言敏感的API，许多专业应用都可以使用MongoDB驱动作为连接数据库的唯一接口了。

驱动API跨语言时依然保持一致，这意味着开发者可以方便地选择合适的语言，任意在JavaScript API可以做的事情在Ruby API也都可以做。如果你是应用开发者，就肯定可以找到适合自己的语言的MongoDB驱动，而不需要担心底层的实现细节。

第一节里我们会先安装MongoDB Ruby驱动，连接数据库，并且介绍如何执行基本的CRUD操作。这将为本章结束时开发的应用打下基础。

3.1.1 安装与连接

可以使用RubyGems来安装MongoDB Ruby驱动，这是Ruby的包管理工具[1]。

许多新操作系统提前预装了Ruby。你也可以通过shell里运行ruby–version来检查安装的Ruby。如果没有安装，可以在www.ruby-lang.org/en/downloads下载，其中有详细的安装指南。

我们还需要RubyGems包管理工具。可能也已经安装了。可通过运行gem–version来检查。RubyGems的安装指南可以在http://docs.rubygems.org/read/chapter/3找到。一旦安装完成，就可以运行命令：

```
gem install mongo
```

这里应该会安装mongo和bson[2]的gems包，会看到下面的输出信息（新的版本号可能不太一样）：

```
Fetching: bson-3.2.1.gem (100%)
Building native extensions. This could take a while...
Successfully installed bson-3.2.1
```

[1] 【译者注】.NET 程序使用 NuGet, Node.js 可以使用 NPM，Java 可以使用 Maven。中国 MongoDB 学习交流群 511943641

[2] BSON，在下一节里介绍，是二进制 JSON 格式，MongoDB 用它来表示文档数据。bson Ruby gem 包会从 BSON 序列化 Ruby 对象。

```
Fetching: mongo-2.0.6.gem (100%)
Successfully installed mongo-2.0.6
2 gems installed
```

虽然是可选的，但还是推荐大家安装bson_ext gem包。bson_ext是官方的保护C语言实现的BSON包，在MongoDB驱动里使用了更高效的BSON处理方式。默认没有安装这个gem包，因为需要编译器。放心，就算我们没有安装bson_ext，我们的程序也可以正常运行。

我们会从连接MongoDB数据库开始。首先，通过运行`mongo` shell来确保已经连接成功运行的`mongod`进程。接下来，创建一个connect.rb文件，然后输入以下代码：

```
require 'rubygems'
require 'mongo'

$client = Mongo::Client.new([ '127.0.0.1:27017' ], :database => 'tutorial')
Mongo::Logger.logger.level = ::Logger::ERROR
$users = $client[:users]
puts 'connected!'
```

前两个`require`语句确保我们可以下载驱动。下面三行初始化客户端连接localhost和tutorial数据库，在`$users`变量里存储`users`集合引用，然后打印`connected`!字符串！我们在每个变量前面加了`$`符号，确保每个变量是全局的，这样可以在connect.rb脚本之外访问、保存文件，然后运行：

```
$ ruby connect.rb
D, [2015-06-05T12:32:38.843933 #33946] DEBUG -- : MONGODB | Adding
    127.0.0.1:27017 to the cluster. | runtime: 0.0031ms
D, [2015-06-05T12:32:38.847534 #33946] DEBUG -- : MONGODB | COMMAND |
    namespace=admin.$cmd selector={:ismaster=>1} flags=[] limit=-1 skip=0
    project=nil | runtime: 3.4170ms
connected!
```

不出意外的话，可以正常通过Ruby连接MongoDB数据库。应该可以在shell里看到connected！字符串。这看起来非常简单，但是确实是连接MongoDB数据库的第一步。接下来，我们会使用连接再插入一些文档数据。

3.1.2 Ruby里插入文档数据

要运行有趣的MongoDB查询测试，首先要有一些数据，因此我们先来创建一些。设计MongoDB驱动时使用了自己的语言文档表示形式。在Javascript里，JSON对象是个理所当然的选择，因为JSON是文档数据结构；在Ruby里，哈希结构是最有意义的。原生的Ruby哈希与JSON对象只有一些差别，最显著的就是JSON使用冒号来分割键值，而Ruby使用的

是哈希符号(=>)[1]。

如果跟着流程，可以继续往connect.rb文件里添加数据。另外一个方法是直接使用Ruby的shell，`Irb`。`Irb`是个REPL(read, evaluate, print loop)控制台，可以输入动态执行的Ruby代码，用于实验操作是非常理想的。`Irb`编写的任意代码都可以保存到脚本里，所以我们推荐使用它来学习新知识。你也可以启动`irb`，并且只需要使用connect.rb就可以立即访问链接、数据库和集合对象了，然后就可以运行Ruby代码并接受反馈结果。

以下是个例子：

```
$ irb -r ./connect.rb
irb(main):017:0> id = $users.insert_one({"last_name" => "mtsouk"})
=> #<Mongo::Operation::Result:70275279152800 documents=[{"ok"=>1, "n"=>1}]>
irb(main):014:0> $users.find().each do |user|
irb(main):015:1* puts user
irb(main):016:1> end
{"_id"=>BSON::ObjectId('55e3ee1c5ae119511d000000'), "last_name"=>"knuth"}
{"_id"=>BSON::ObjectId('55e3f13d5ae119516a000000'), "last_name"=>"mtsouk"}
=> #<Enumerator: #<Mongo::Cursor:0x70275279317980
@view=#<Mongo::Collection::View:0x70275279322740 namespace='tutorial.users
@selector={} @options={}>>:each>
```

irb的命令行shell带有>（不同的机器看起来可能不一样）。它允许我们输入不同的命令，以及之前代码里我们高亮的命令。当我们在irb里运行命令时，如果有数据，它就会返回命令的输出结果。这就是=>符号后显示的东西。

我们来为users集合添加一些文档数据。我们可以创建2个文档表示2个用户，smith 和 jones。每个文档使用Ruby hash哈希表示，赋值给变量：

```
smith = {"last_name" => "smith", "age" => 30}
jones = {"last_name" => "jones", "age" => 40}
```

要保存文档，需要把对象传递给集合的`insert`方法。每次调用`insert`都会返回我们可以保存到变量里便于后期查询使用的唯一ID：

```
smith_id = $users.insert_one(smith)
jones_id = $users.insert_one(jones)
```

我们可以使用简单的查询来检验保存的文档数据，所以可以使用user集合的`find()`方法进行查询，如下所示：

```
irb(main):013:0> $users.find("age" => {"$gt" => 20}).each.to_a do |row|
irb(main):014:1* puts row
```

[1]Ruby 1.9 里，也可以使用冒号分割键值，像哈希`hash={foo: 'bar'}`一样，但是为了兼容性我们还是坚持使用哈希符号。

```
irb(main):015:1> end
=> [{"_id"=>BSON::ObjectId('55e3f7dd5ae119516a000002'),
"last_name"=>"smith",
"age"=>30}, {"_id"=>BSON::ObjectId('55e3f7e25ae119516a000003'),
"last_name"=>"jones", "age"=>40}]
```

如果运行irb，命令窗口里会显示查询结果。如果从Ruby文件里运行代码，则前置Ruby的p方法用来打印输出信息：

```
p $users.find( :age => {"$gt" => 20}).to_a
```

我们已经成功地从Ruby插入了2条文档数据。现在来详细看一下查询功能。

3.1.3 查询与游标

既然我们已经创建了文档，现在应该来学习MongoDB提供的查询操作了（CRUD的R）。Ruby驱动定义了丰富的接口来访问数据，而且处理了许多底层的细节。本节里展示的查询是非常简单的查询，其实MongoDB允许我们执行更加复杂的查询，比如文本搜索和聚合，这在后面的章节里会介绍。

我们会从标准的find方法来看这个如何实现。这里是数据集上两个可能的查找操作：

```
$users.find({"last_name" => "smith"}).to_a
$users.find({"age" => {"$gt" => 30}}).to_a
```

第一个查询last_name是smith的用户文档，第二个查询所有年龄age大于30的用户文档。尝试在irb输入第三个查询：

```
2.1.4 :020 > $users.find({"age" => {"$gt" => 30}})
=> #<Mongo::Collection::View:0x70210212601420 namespace='tutorial.users
@selector={"age"=>{"$gt"=>30}} @options={}>
```

结果在Mongo::Collection::View对象里返回，它扩展自Iterable接口，方便迭代结果集合。我们会在3.2.3一节里详细讨论这个问题。同时，我们也可以获取$gt查询的结果：

```
cursor = $users.find({"age" => {"$gt" => 30}})
cursor.each do |doc|
puts doc["last_name"]
end
```

这是Ruby的 each迭代器，传递每个结果给代码块。last_name被打印在控制台上。查询中使用的$gt是个MongoDB操作符，$字符与Ruby里定义全局变量的Ruby没有关系，比如$users。如果集合里文档没有last_name字段，你可能发现Ruby会打印nil（Ruby的null值），这表示缺少值，而且很常见。

从上一章shell的例子里，你可能要考虑的是游标。shell使用游标的方式与其他驱动基本一样，其不同点在于shell会自动迭代`find()`结果里的20个游标。要获取其他的结果，可以继续使用`it`命令手动迭代。

3.1.4 更新和删除

回忆一下第2章里的updates，至少需要2个参数：查询选择权和更新文档。下面是使用Ruby驱动的简单例子：

```
$users.find({"last_name" => "smith"}).update_one({"$set" => {"city" =>
"Chicago"}})
```

这个更新会首先找到`last_name`是`smith`的文档，如果找到就会把`city`设置为Chicago。这个更新使用了`$set`操作符。你可以运行查询来查看修改：

```
$users.find({"last_name" => "smith"}).to_a
```

该视图允许我们决定只更新一个文档还是更新匹配查询的所有文档。在之前的例子里，就算你有几个名字为smith的文档，也只有一个会被更新。要为特别的smith更新数据，就需要添加更多的查询选择器。如果你想更新所有的smith文档，就必须使用`update_many`替换`update_one`方法：

```
$users.find({"last_name" => "smith"}).update_many({"$set" => {"city" =>
"Chicago"}})
```

删除数据更加简单。我们已经讨论了MongoDB shell是如何工作的，与Ruby驱动也没有什么不同。复习下：只需要使用`remove`方法。这个方法接受一个选择器参数，只删除匹配的文档数据。如果不提供参数，就会删除集合里的所有数据。以下例子假设要删除所有年龄大于40岁的用户文档：

```
$users.find({"age" => {"$gte" => 40}}).delete_one
```

这里只会删除匹配查询的第一个数据。如果要删除所有匹配的文档，就需要运行：

```
$users.find({"age" => {"$gte" => 40}}).delete_many
```

无参数时，`drop`方法会删除所有剩余的文档：

```
$users.drop
```

3.1.5 数据库命令

在前面一章，我们已经学习了数据库命令。我们看到了2个stats命令。这里，我们将会学习如何在驱动里使用listDatabases运行命令。这是管理数据库必须运行的命令，它在启动验证的时候会被特殊对待。关于验证和管理数据库的详细内容，请参考第10章。

首先，我们创建一个Ruby数据库对象来引用admin数据库。然后传递查询命令给command方法：

```
$admin_db = $client.use('admin')
$admin_db.command({"listDatabases" => 1})
```

注意：这些代码仍然会用到我们在connect.rb脚本里保存的代码，因为它需要使用$client里保存的连接。应答结果使用Ruby哈希，列出了所有的数据库以及磁盘大小：

```
#<Mongo::Operation::Result:70112905054200 documents=[{"databases"=>[
{
    "name"=>"local",
    "sizeOnDisk"=>83886080.0,
    "empty"=>false
},
{
    "name"=>"tutorial",
"sizeOnDisk"=>83886080.0,
"empty"=>false
},
{
    "name"=>"admin",
```

```
    "sizeOnDisk"=>1.0, "empty"=>true
}], "totalSize"=>167772160.0, "ok"=>1.0}]>
=> nil
```

这可能与irb和MongoDB驱动不同，但是应该很容易访问。一旦适应了Ruby哈希表示的文档，shell API的转换应该是无缝的。

绝大部分驱动都提供了和数据库命令一致的功能接口。你可能还记得前一章里介绍的，用`remove`命令不会真正删除集合。要删除集合和索引，就必须使用`drop_collection`方法：

```
db = $client.use('tutorial')
db['users'].drop
```

如果对于使用MongoDB的Ruby驱动还不够熟练，那也没有关系，我们会在3.3节里获得更多练习机会。但是现在我们将会花点时间来学习MongoDB驱动的工作原理。这将会揭示更多的MongoDB设计原理，方便大家更高效地使用驱动。

3.2 驱动工作原理

How the drivers work

现在是时候明白在驱动或者MongoDB shell里输入命令后发生的故事了。本节里，我们将会看到驱动序列化数据并与数据库通信。

所有的MongoDB驱动都执行三个主要的功能：首先，生成MongoDB对象ID。默认都存储在所有的文档的`_id`字段里。其次，驱动会把任意语言表示的文档对象转换为BSON或者从BSON转换回来，BSON是MongoDB使用的二进制JSON格式。之前的例子里，驱动序列化所有的Ruby哈希为BSON，或者把BSON转换为Ruby哈希。

最后，使用TCP socket与数据库连接通信，此时使用的是MongoDB自定义协议。协议的详细内容不在本节的讨论范围里。socket通信的风格，特别是写入数据等待应答是非常重要的，我们将会在本节里详细介绍。

每个MongoDB文档都需要一个主键。这个键对于每个集合里的文档来说必须是唯一的，存储在文档的`_id`字段里。开发者也可以使用自己的值作为`_id`的值，但是没有提供值时，MongoDB会使用自己的默认值。在发送文档数据给服务器之前，驱动会检查是否有`_id`字段。如果没有，就会生产一个对象`_id`。

MongoDB对象ID被设计成全局唯一，这意味着它可以在特定的上下文里确保唯一。如何确保

呢？我们来详细看一下算法。

如果你仔细研究过MongoDB，就应该见过对象ID。首先，它看起来很像一串随机文本，比如4c291856238d3b19b2000001。你可能没有注意到，这串字符是12个字节的十六进制形式。它们确实保存着一些有用的信息。这些字节有特殊的结构，如图3.1所示。

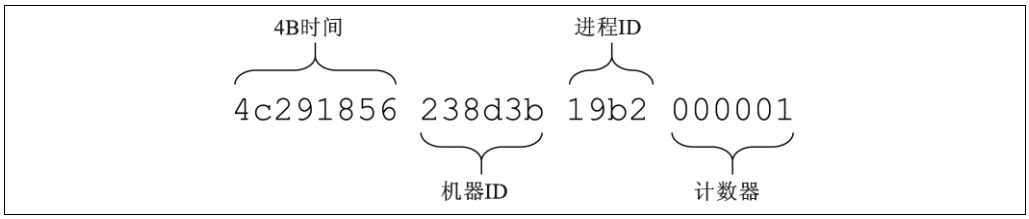

图 3.1　MongoDB 对象 ID 格式

最重要的4个字节包含着标准的Unix时间戳[1]。后面3个字节是机器ID，紧跟着是2个字节的进程ID。最后3个字节存储的是进程本地计数器，每次生成新ID就会自动增长。计数器可以保证同一个进程和同一时刻内不会重复。

为什么对象ID有这种格式？很重要的原因是ID是在驱动里生成的，而不是在服务器上生成的。这与许多RDBMS系统不同。关系型数据在服务器端自增主键，因此导致服务器成为生成键的瓶颈。如果多个驱动生成ID并插入文档，它们就需要创建唯一标识符而不会互相影响的方法。因此，时间戳、机器ID和进程ID包含在标识符中，让ID无法重复。

你可能已经考虑了这种问题发生的概率。实际上，在遇到计数器（每秒2^{24}百万次）瓶颈之前已经遇到了插入文档的瓶颈了。可以稍微想象一下（不太可能），如果你跨不同的机器分布式部署驱动，就不可能有相同的机器ID。例如，Ruby驱动使用下面的机器ID计算公式：

```
@@machine_id = Digest::MD5.digest(Socket.gethostname)[0, 3]
```

对于可能的问题，它们可能使用相同的进程ID启动MongoDB驱动，而且某个时刻有相同的计数值。

实际上，不要担心重复，因为几乎是不可能的。

使用MongoDB对象ID一个附加的好处就是它带有时间戳。绝大部分驱动都允许我们获取时间戳，因此提供了文档的创建时间，可以解析出最近的秒数。

[1]许多 UNIX 机器（也包括 Linux）存储时间的格式为 UNIX Time 或者 POSIX time 格式，它们只从 1970 年 1 月 1 号 00:00 开始计数。这意味着时间戳可以是整数。例如，2010-06-28 21:47:02 可以表示为 1277761622 (或者十六进制的 0x4c291856)，从开始 epoch 到现在的总秒数。

使用Ruby驱动，也可以通过调用对象ID的generation_time方法来获取ID的创建时间，作为Ruby Time对象。

```
irb> require 'mongo'
irb> id = BSON::ObjectId.from_string('4c291856238d3b19b2000001')
=> BSON::ObjectId('4c291856238d3b19b2000001')
irb> id.generation_time
=> 2010-06-28 21:47:02 UTC
```

自然而然地，你也可以使用对象ID来确定对象创建时间的查询范围。

例如，如果想要查询创建时间在2013年6月的文档，就可以创建2个对象ID，它的时间戳编码了日期，然后在_id限制范围。因为提供了方法从任意的Time对象来生成对象ID，操作代码很简单[1]，如下所示：

```
jun_id = BSON::ObjectId.from_time(Time.utc(2013, 6, 1))
jul_id = BSON::ObjectId.from_time(Time.utc(2013, 7, 1))
@users.find({'_id' => {'$gte' => jun_id, '$lt' => jul_id}})
```

正如前面提到的，你也可以设置_id的值。当文档的某个字段十分重要，而且一直唯一时，这个可以起作用。例如，users集合要存储username在_id字段里，而不是在对象ID中。这两种方法各有好处，可以依开发者的个人喜好来选择。

3.3 开发简单的应用程序

Building a simple application

接下来我们会构建一个简单的应用来保存和显示Tweets内容。可以把这个当做一个大型网站的组件，允许用户监控和保留与他们业务相关的搜索词语。这个例子会告诉大家，调用Twitter APi后返回的是JSON数据并转换为MongoDB文档是多么简单、方便。如果使用关系型数据库，必须提前设计数据schema，可能由许多表组成，而且要提前定义这些表。这里我们什么都不需要做，只需要保存有丰富数据结构的Tweet文档，而且要高效地查询它们。

我们把这个项目叫做TweetArchiver。TweetArchiver有2个组件：Archiver和viewer。Archiver会调用Twitter search API并保存相关的推文，viewer会在Web浏览器里显示结果。

[1]例子实际上无法工作，只是个详细的练习。现在你应该可以有足够的知识来创建有意义的数据进行查询练习了。为什么不开始动手呢？

3.3.1 设置

这个应用需要4个Ruby库。本章的代码存储库包含名为Gemfile的文件，它列出了这些gems包。

把工作目录修改为chapter3，然后确保`ls`命令显示Gemfile。然后使用如下命令进行安装：

```
gem install bundler
bundle install
```

这会确保安装`bundler` gem。接下来，使用Bundler包管理工具安装其他的gem。在Ruby里经常这么做，可以自动匹配需要的版本：我们例子代码需要的版本库。

我们的Gemfile 列举了 `mongo`、`twitter`、`bson`、`sinatra`等4个`gem`，所以都会安装。我们已经使用了`mongo` gem，这里包含它是为了确保正确的版本。`twitter` gem用来和Twitter API进行通信。`sinatra` gem是运行Ruby网站的简单Web服务器，我们将会在3.3.3里讨论更多详细内容。

我们单独提供了这个例子代码，但是逐步介绍可以方便大家理解代码。我们推荐大家动手实战，尝试新的东西。最好能适应配置文件以便在archiver和viewer脚本之间共享信息。创建一个config.rb的代码如下：

```
DATABASE_HOST = 'localhost'
DATABASE_PORT = 27017
DATABASE_NAME = "twitter-archive"
COLLECTION_NAME = "tweets"
TAGS = ["#MongoDB", "#Mongo"]

CONSUMER_KEY = "replace me"
CONSUMER_SECRET = "replace me"
TOKEN = "replace me"
TOKEN_SECRET = "replace me"
```

首先我们指定了网站要使用的数据库和集合的名字。然后我们定义了搜索词汇的可能要发送给Twitter API的数组。

Twitter需要用户注册一个免费账号才能访问API。大家可以在http://apps.twitter.com注册。一旦注册了应用，就可以看到它的验证信息页面，也可能在API keys选项卡里。我们会点击按钮，创建访问令牌(token)。我们会使用这些参数来调用后台的API。

3.3.2 搜集数据

接下来是要编写archiver脚本。我们可以从TweetArchiver开始。我们使用搜索关键字来初始化这个类。然后调用TweetArchiver实例的`update`方法，它会发起API调用。最后把结果保存到MongoDB集合里。

我们来看看此类构造函数：

```
def initialize(tag)
  connection = Mongo::Connection.new(DATABASE_HOST, DATABASE_PORT)
  db = connection[DATABASE_NAME]
  @tweets = db[COLLECTION_NAME]
  @tweets.ensure_index([['tags', 1], ['id', -1]])
  @tag = tag
  @tweets_found = 0

  @client = Twitter::REST::Client.new do |config|
  config.consumer_key = API_KEY
    config.consumer_secret = API_SECRET
    config.access_token = ACCESS_TOKEN
    config.access_token_secret = ACCESS_TOKEN_SECRET
  end
end
```

Initialize方法创建了一个连接、数据库和集合对象，我们用来存储Tweets数据。

可以在tags升序和id降序上创建一个复合索引。因为我们想根据tag进行搜索，然后从新到旧来排序显示，tags升序和id降序的索引会让查询结果基于索引进行过滤。正如你看到的，1表示升序，-1表示降序。不要担心，我们会在第8章里深入讨论这些知识。

我们还需使用config.rb.的验证信息来配置Twitter客户端。这一步会传递这些值给Twitter gem，它会使用这些参数来调用Twitter API。Ruby对于这种配置有特殊的语法；config变量传递给Ruby模块，我们可以在这里设置配置信息。

MongoDB允许我们在不知道数据结构的情况下插入数据。而关系型数据库需要一个提前定义好的schema，要提前知道要存储的数据。将来Twitter可能通过修改API来返回不同的数据，可能需要修改schema才能存储不同的数据。使用MongoDB，就无须担心，MongoDB使用无schema模式来存储Twitter API返回的数据。

Ruby Twitter库返回一个Ruby哈希对象，所以我们可以直接给MongoDB集合对象传递这个对象。在TweetArchiver代码里，我们可以直接添加如下实例方法：

```
def save_tweets_for(term)
  @client.search(term).each do |tweet|
    @tweets_found += 1
    tweet_doc = tweet.to_h
    tweet_doc[:tags] = term
    tweet_doc[:_id] = tweet_doc[:id]
    @tweets.insert_one(tweet_doc)
  end
end
```

在保存每个Tweet文档之前，要做两个小修改。为了简化查询，给每个文档对象添加一个tags属性。也可以为每个对象设置_id字段值作为文档的ID，替换集合的主键，确保每个Tweet推文对象都是唯一的。然后把修改过的文档对象传递给save方法。

要在类里使用这些代码，还需要做一些工作。首先，必须配置MongoDB驱动，这样才可以连接正确的mongod服务，使用正确的数据库和集合。当我们使用MongoDB的时候，这个代码非常简单，我们可以直接复制。其次，必须为Twitter gem配置开发者账号。

这一步是必须的，因为Twitter限制只有注册的开发者才可以调用API。列表3.1里也提供了update方法，它提供用户反馈和调用。

列表3.1　获取Tweet推文并保存到MongoDB数据库的类中

```
$LOAD_PATH << File.dirname(__FILE__)
require 'rubygems'
```

```
require 'mongo'
require 'twitter'
require 'config'

class TweetArchiver

def initialize(tag)
    client =
     Mongo::Client.new(["#{DATABASE_HOST}:#{DATABASE_PORT}"],:database =>
     "#{DATABASE_NAME}")
    @tweets = client["#{COLLECTION_NAME}"]
    @tweets.indexes.drop_all
    @tweets.indexes.create_many([
      { :key => { tags: 1 }},
      { :key => { id: -1 }}
      ])
    @tag = tag
    @tweets_found = 0

    @client = Twitter::REST::Client.new do |config|
      config.consumer_key = "#{API_KEY}"
      config.consumer_secret = "#{API_SECRET}"
      config.access_token = "#{ACCESS_TOKEN}"
      config.access_token_secret = "#{ACCESS_TOKEN_SECRET}"
    end
  end

  def update
    puts "Starting Twitter search for '#{@tag}'..."
    save_tweets_for(@tag)
    print "#{@tweets_found} Tweets saved.\n\n"
  end

  private
  def save_tweets_for(term)
    @client.search(term).each do |tweet|
      @tweets_found += 1
      tweet_doc = tweet.to_h
      tweet_doc[:tags] = term
      tweet_doc[:_id] = tweet_doc[:id]
      @tweets.insert_one(tweet_doc)
    end
  end
end
```

创建 Tweet-Archive 的实例

使用 config.rb 中的值配置

保存 save_tweets_for 的方法

使用 Twitter 搜索并保存结果到 Mongo

剩下的代码就是编写脚本来运行`TweetArchiver`代码，根据每次输入的搜索关键字来搜索结果。我们可以创建一个名为update.rb的文件（或者从提供的代码里复制），它包含的代码如下：

```
$LOAD_PATH << File.dirname(__FILE__)
require 'config'
require 'archiver'
TAGS.each do |tag|
  archive = TweetArchiver.new(tag)
  archive.update
```

```
end
```

接下来，运行更新的脚本：

```
ruby update.rb
```

我们会看到一些提示Tweet推文是否找到和保存的信息。我们也可以通过MongoDB shell直接查询集合来检查脚本是否可以工作：

```
> use twitter-archive
switched to db twitter-archive
> db.tweets.count()
30
```

这里最重要的是我们使用几行代码来管理Twitter搜索结果的推文保存过程[1]。接下来就是现实结果的代码了。

3.3.3 查看存档

我们可以使用Ruby的Sinatra Web框架来构建一个简单的App来显示结果。

Sinatra允许直接定义Web应用的终结点(endpoint)，以及指定应答消息。它的强大之处依赖于其简易性。例如，index页面的内容可以设置如下：

```
get '/' do
  "response"
End
```

这个代码指定了/终结点的`get`请求返回`response`的值给客户端。使用这个格式，你可以编写许多终结点给`Web`程序，每个都可以在返回结果之前执行特定的`Ruby`代码。我们也可以发现更多的信息，包括Sinatra完整文档，参见http://sinatrarb.com。

我们现在引入一个新的文件viewer.rb，与其他脚本放在同一个目录下。接下来，创建个新的文件夹叫views，然后创建个文件叫tweets.erb。完成这些工作以后，项目的目录结构应该如下：

```
- config.rb
- archiver.rb
- update.rb
- viewer.rb
- /views
  - tweets.erb
```

[1]也可以使用更少的代码实现。这个就作为练习留给读者了。

再说一次，大家可以自己创建文件或者直接从例子代码文件里复制文件。

- 现在编写viewer.rb文件，使用如表3.2的代码。

列表3.2 显示存档推文的Sinatra应用

```
$LOAD_PATH << File.dirname(__FILE__)
require 'rubygems'
require 'mongo'
require 'sinatra'
require 'config'
require 'open-uri'

configure do
  client = Mongo::Client.new(["#{DATABASE_HOST}:#{DATABASE_PORT}"], :database
     => "#{DATABASE_NAME}")
   TWEETS = client["#{COLLECTION_NAME}"]
end

get '/' do
  if params['tag']
    selector = {:tags => params['tag']}
  else
    selector = {}
  end

  @tweets = TWEETS.find(selector).sort(["id", -1])
  erb :tweets
end
```

❶ 必备库

❷ 实例化集合

❸ 动态构建 jQuery 选择器

❹ 或者使用空选择器

❺ 查询

❻ 渲染视图

第一行需要必备的库，与配置文件一致❶。接下来，有个配置块创建MongoDB连接并存储tweets集合的引用到TWEETS里❷。

真正核心的代码就是get '/' do开始的部分。这个部分的代码处理应用根目录URL的请求。首先，我们创建查询选择器。如果提供了URL Tag标签参数，就要创建一个查询选择器来限制结果集❸。否则就创建空选择器，它会返回集合里的所有文档❹。然后执行查询❺。目前为止，我们应该知道@tweets变量里赋值的是个游标而不是结果集。我们还需要在视图里迭代循环处理游标。

倒数第二行❻渲染视图文件tweets.erb（参见列表3.3）。

列表3.3 嵌入Ruby脚本的HTML代码来显示推文信息

```
<!DOCTYPE html>
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
  <style>
    body {
      width: 1000px;
      margin: 50px auto;
```

```
      font-family: Palatino, serif;
      background-color: #dbd4c2;
      color: #555050;
    }
    h2 {
      margin-top: 2em;
      font-family: Arial, sans-serif;
      font-weight: 100;
    }
  </style>
</head>
<body>
<h1>Tweet Archive</h1>
<% TAGS.each do |tag| %>
  <a href="/?tag=<%= URI::encode(tag) %>"><%= tag %></a>
<% end %>
<% @tweets.each do |tweet| %>
  <h2><%= tweet['text'] %></h2>
  <p>
    <a href="http://twitter.com/<%= tweet['user']['screen_name'] %>">
      <%= tweet['user']['screen_name'] %>
    </a>
    on <%= tweet['created_at'] %>
  </p>
  <img src="<%= tweet['user']['profile_image_url'] %>" width="48" />
<% end %>
</body>
</html>
```

绝大部分是HTML代码，混合了部分ERB（嵌入的Ruby代码）。Sinatra app 通过ERB处理器，运行`tweets.erb`文件，并评估`<%` 和 `%>`符号之间的`Ruby`代码。

最重要的部分就出现了，使用2个迭代器。第一个循环通过tags列表来展示某个tag的超链接。

第二个循环，使用`@tweets.each`代码，循环显示每个Tweet的文本，创建日期和用户头衔。我们可以通过运行程序来查看结果：

```
$ ruby viewer.rb
```

如果启动程序无错，我们就会看到标准的Sinatra启动消息，结果如下：

```
$ ruby viewer.rb
[2013-07-05 18:30:19] INFO WEBrick 1.3.1
[2013-07-05 18:30:19] INFO ruby 1.9.3 (2012-04-20) [x86_64-darwin10.8.0]
== Sinatra/1.4.3 has taken the stage on 4567 for development with backup from
   WEBrick
[2013-07-05 18:30:19] INFO WEBrick::HTTPServer#start: pid=18465 port=4567
```

我们可以把浏览器地址指向http://localhost:4567。页面截图应该如图3.2所示。尝试点击屏幕顶部的链接来限制搜索特定的tag标签。

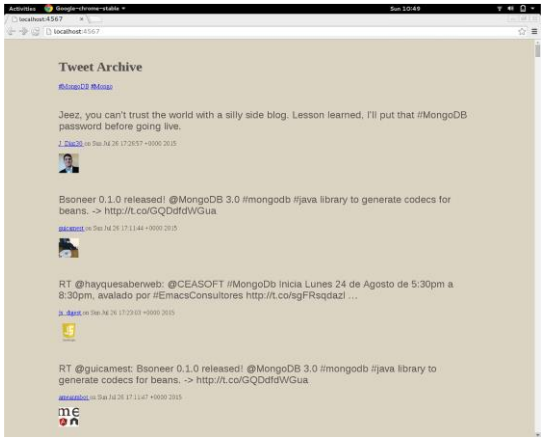

图 3.2　浏览器里渲染的 Tweet Archiver 存档数据

这是应用的扩展部分。确实很简单，但是它只演示了使用MongoDB多么简单。我们无须提前定义数据结构schema，而且可以使用辅助索引加速查询，阻止重复的插入，而且使用编程语言调用MongoDB也非常简单。

3.4　总结

Summary

我们刚刚已经学习了如何使用Ruby语言来操作MongoDB数据库。我们也看了Ruby表示文档数据多么简单，而且CRUD操作的API与shell的几乎一样。我们也深入学习了驱动内部关于对象ID、BSON以及MongoDB网络协议的知识。最后，我们也构建了一个简单的MongoDB应用程序来编写真实的代码。虽然真实的项目使用MongoDB时要复杂得多，但是开发应用程序操作数据库的方式应该是相似的。

在第4章里，我们将会用到目前为止学习的一切知识。特别是，我们将会实战开发使用MongoDB构建一个电子商务网站。这是个非常庞大的项目，所以我们将会关注以后开发的某些功能模块。我们将会介绍该领域的某些数据模型，而且会介绍如何插入和查询该类型的数据。