MongoDB

实战

作者： 王文龙
出版社：芒果出版社
ISBN：2345678910JQK
上架时间：2011 年 6 月 12 日星期日 21:01
出版日期：2011 年 6 月 12 日星期日 21:01
开本：16 开
页码：91
版次：1-1

目录

第一部分 基础篇

第一章 走进 MongoDB

MongoDB 是一个高性能，开源，无模式的文档型数据库，是当前 NoSQL 数据库产品中最热门的一种。它在许多场景下可用于替代传统的关系型数据库或键/值存储方式，MongoDB 使用 C++开发。MongoDB 的官方网站地址是：http://www.mongodb.org/，读者朋友们可以在此获得更详细的信息。

1.1 为什么要用 NoSQL

1.1.1 NoSQL 简介

NoSQL，全称是”Not Only Sql”,指的是非关系型的数据库。这类数据库主要有这些特点：非关系型的、分布式的、开源的、水平可扩展的。原始的目的是为了大规模 web 应用，这场全新的数据库革命运动早期就有人提出，发展至 2009 年趋势越发高涨。NoSQL 的拥护者们提倡运用非关系型的数据存储，通常的应用如：模式自由、支持简易复制、简单的 API、最终的一致性（非 ACID）、大容量数据等。NoSQL 被我们用得最多的当数 key-value 存储，当然还有其他的文档型的、列存储、图型数据库、xml 数据库等。相对于目前铺天盖地的关系型数据库运用，这一概念无疑是一种全新思维的注入。

1.1.2 发展现状

现今的计算机体系结构在数据存储方面要求应用架构具备庞大的水平扩展性，而 NoSQL 正在致力于改变这一现状。目前新浪微博的 Redis 和 Google 的 Bigtable 以及 Amazon 的 SimpleDB 使用的就是 NoSQL 型数据库。

NoSQL 项目的名字上看不出什么相同之处，但是，它们通常在某些方面相同：它们可以处理超大量的数据。

这场革命目前仍然需要等待。NoSQL 对大型企业来说还不是主流，但是，一两年之后很可能就会变个样子。在 NoSQL 运动的最新一次聚会中，来自世界各地的 150 人挤满了 CBS Interactive 的一间会议室。分享他们如何推翻缓慢而昂贵的关系数据库的暴政，怎样使用更有效和更便宜的方法来管理数据。

关系型数据库给你强加了太多东西。它们要你强行修改对象数据，以满足数据库系统的需要。在 NoSQL 拥护者们来看，基于 NoSQL 的数据库替代方案“只是给你所需要的”。

1.1.3 为什么是 NoSQL

随着互联网 web2.0 网站的兴起，非关系型的数据库现在成了一个极其热门的新领域，非关

系数据库产品的发展非常迅速，而传统的关系型数据库在应付 web2.0 网站，特别是超大规模和高并发的 SNS 类型的 web2.0 纯动态网站已经显得力不从心，暴露了很多难以克服的问题，例如：

1、 High performance - 对数据库高并发读写的需求

web2.0 网站要根据用户个性化信息来实时生成动态页面和提供动态信息，所以基本上无法使用动态页面静态化技术，因此数据库并发负载非常高，往往要达到每秒上万次读写请求。关系型数据库应付上万次 SQL 查询还勉强顶得住，但是应付上万次 SQL 写数据请求，硬盘 IO 就已经无法承受了，其实对于普通的 BBS 网站，往往也存在对高并发写请求的需求。

2、Huge Storage - 对海量数据的高效率存储和访问的需求

对于大型的 SNS 网站，每天用户产生海量的用户动态信息，以国外的 Friend feed 为例，一个月就达到了 2.5 亿条用户动态，对于关系数据库来说，在一张 2.5 亿条记录的表里面进行 SQL 查询，效率是极其低下乃至不可忍受的。再例如大型 web 网站的用户登录系统，例如腾讯，盛大，动辄数以亿计的帐号，关系数据库也很难应付。

3、High Scalability && High Availability - 对数据库的高可扩展性和高可用性的需求

在基于 web 的架构当中，数据库是最难进行横向扩展的，当一个应用系统的用户量和访问量与日俱增的时候，你的数据库却没有办法像 web server 和 app server 那样简单的通过添加更多的硬件和服务节点来扩展性能和负载能力。对于很多需要提供 24 小时不间断服务的网站来说，对数据库系统进行升级和扩展是非常痛苦的事情，往往需要停机维护和数据迁移，可是停机维护随之带来的就是公司收入的减少。

在上面提到的“三高”需求面前，关系数据库遇到了难以克服的障碍，而对于 web2.0 网站来说，关系数据库的很多主要特性却往往无用武之地，例如：

1、数据库事务一致性需求

很多 web 实时系统并不要求严格的数据库事务，对读一致性的要求很低，有些场合对写一致性要求也不高。因此数据库事务管理成了数据库高负载下一个沉重的负担。

2、数据库的写实时性和读实时性需求

对关系数据库来说，插入一条数据之后立刻查询，是肯定可以读出来这条数据的，但是对于很多 web 应用来说，并不要求这么高的实时性。

3、对复杂的 SQL 查询，特别是多表关联查询的需求

任何大数据量的 web 系统，都非常忌讳多个大表的关联查询，以及复杂的数据分析类型的复杂 SQL 报表查询，特别是 SNS 类型的网站，从需求以及产品设计角度，就避免了这种情况的产生。往往更多的只是单表的主键查询，以及单表的简单条件分页查询，SQL 的功能被极大的弱化了。

因此，关系数据库在这些越来越多的应用场景下显得不那么合适了，为了解决这类问题的 NoSQL 数据库应运而生。

NoSQL 是非关系型数据存储的广义定义。它打破了长久以来关系型数据库与 ACID 理论大一统的局面。NoSQL 数据存储不需要固定的表结构，通常也不存在连接操作。在大数据存取上具备关系型数据库无法比拟的性能优势，该概念在 2009 年初得到了广泛认同。

当今的应用体系结构需要数据存储在横向伸缩性上能够满足需求。而 NoSQL 存储就是为了实现这个需求。Google 的 BigTable 与 Amazon 的 Dynamo 是非常成功的商业 NoSQL 实现。一些开源的 NoSQL 体系，如 Facebook 的 Cassandra， Apache 的 HBase，也得到了广泛认同。从这些 NoSQL 项目的名字上看不出什么相同之处：Hadoop、Voldemort、Dynomite，还有其它很多，但它们都有一个共同的特点，就是要改变大家对数据库在传统意义上的理解。

1.1.4 NoSQL 特点

1、 它可以处理超大量的数据

2、 它运行在便宜的 PC 服务器集群上

PC 集群扩充起来非常方便并且成本很低，避免了传统商业数据库“sharding”操作的复杂性和成本。

3、 它击碎了性能瓶颈

NoSQL 的支持者称，通过 NoSQL 架构可以省去将 Web 或 Java 应用和数据转换成 SQL 格式的时间，执行速度变得更快。

“SQL 并非适用于所有的程序代码”，对于那些繁重的重复操作的数据，SQL 值得花钱。但是当数据库结构非常简单时，SQL 可能没有太大用处。

4、 它没有过多的操作

虽然 NoSQL 的支持者也承认关系型数据库提供了无可比拟的功能集合，而且在数据完整性上也发挥绝对稳定，他们同时也表示，企业的具体需求可能没有那么复杂。

5、 它的支持者源于社区

因为 NoSQL 项目都是开源的，因此它们缺乏供应商提供的正式支持。这一点它们与大多数开源项目一样，不得不从社区中寻求支持。

NoSQL 发展至今，出现了好几种非关系性数据库，本书就以 NoSQL 中目前表现最好的 MongoDB 为例来进行说明。

1.2 初识 MongoDB

MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。他支持的数据结构非常松散，是类似 json 的 bjson 格式，因此可以存储比较复杂的数据类型。MongoDB 最大的特点是他支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。它是一个面向集合的,模式自由的文档型数据库。

1、 面向集合（Collenction-Orented）

意思是数据被分组存储在数据集中， 被称为一个集合（Collenction)。每个集合在数据库中都有一个唯一的标识名，并且可以包含无限数目的文档。集合的概念类似关系型数据库

（RDBMS）里的表（table），不同的是它不需要定义任何模式（schema)。

2、 模式自由（schema-free)

意味着对于存储在 MongoDB 数据库中的文件，我们不需要知道它的任何结构定义。提了这么多次"无模式"或"模式自由"，它到是个什么概念呢？例如，下面两个记录可以存在于同一个集合里面：

{"welcome" : "Beijing"}

{"age" : 25}

3、 文档型

意思是我们存储的数据是键-值对的集合,键是字符串,值可以是数据类型集合里的任意类型,包括数组和文档. 我们把这个数据格式称作 “BSON” 即 “Binary Serialized dOcument Notation.”

下面将分别介绍 MongoDB 的特点、功能和适用场合。

1.2.1 特点

- 面向集合存储，易于存储对象类型的数据
- 模式自由
- 支持动态查询
- 支持完全索引，包含内部对象
- 支持查询
- 支持复制和故障恢复
- 使用高效的二进制数据存储，包括大型对象（如视频等）
- 自动处理碎片，以支持云计算层次的扩展性
- 支持 Python，PHP，Ruby，Java，C，C#，Javascript，Perl 及 C++语言的驱动程序，社区中也提供了对 Erlang 及.NET 等平台的驱动程序
- 文件存储格式为 BSON（一种 JSON 的扩展）
- 可通过网络访问

1.2.2 功能

- 面向集合的存储：适合存储对象及 JSON 形式的数据
- 动态查询：MongoDB 支持丰富的查询表达式。查询指令使用 JSON 形式的标记，可轻易查询文档中内嵌的对象及数组
- 完整的索引支持：包括文档内嵌对象及数组。MongoDB 的查询优化器会分析查询表达式，并生成一个高效的查询计划
- 查询监视：MongoDB 包含一系列监视工具用于分析数据库操作的性能
- 复制及自动故障转移：MongoDB 数据库支持服务器之间的数据复制，支持主-从模式及服务器之间的相互复制。复制的主要目标是提供冗余及自动故障转移
- 高效的传统存储方式：支持二进制数据及大型对象（如照片或图片）
- 自动分片以支持云级别的伸缩性：自动分片功能支持水平的数据库集群，可动态添加额

外的机器

1.2.3 适用场合

- 网站数据：MongoDB 非常适合实时的插入，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
- 缓存：由于性能很高，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 搭建的持久化缓存层可以避免下层的数据源过载
- 大尺寸，低价值的数据：使用传统的关系型数据库存储一些数据时可能会比较昂贵，在此之前，很多时候程序员往往会选择传统的文件进行存储
- 高伸缩性的场景：MongoDB 非常适合由数十或数百台服务器组成的数据库。MongoDB 的路线图中已经包含对 MapReduce 引擎的内置支持
- 用于对象及 JSON 数据的存储：MongoDB 的 BSON 数据格式非常适合文档化格式的存储及查询

第二章 安装和配置

MongoDB 的官方下载站是 http://www.mongodb.org/downloads，可以去上面下载最新的安装程序下来。在下载页面可以看到，它对操作系统支持很全面，如 OS X、Linux、Windows、Solaris 都支持，而且都有各自的 32 位和 64 位版本。目前的稳定版本是 1.8.1 版本。

MongoDB Downloads

This table lists MongoDB distributions by platform and version. There are also packages available for various package managers.

	OS X 32-bit note	OS X 64-bit	Linux 32-bit note	Linux 64-bit	Windows 32-bit note	Windows 64-bit	Solaris i86pc note	Solaris 64	Source
Production Release (Recommended)									
1.8.1 *4/6/2011* Changelog Release Notes	download	download	download *legacy-static	download *legacy-static	download	download	download	download	tgz zip
Nightly Changelog	download	download	download *legacy-static	download *legacy-static	download	download	download	download	tgz zip

注意:
1. MongoDB 1.8.1 Linux 版要求 glibc 必须是 2.5 以上，所以需要先确认操作系统的 glibc 的版本，笔者最初用 Linux AS 4 安装不上，最后用的是 RHEL5 来安装才成功的。
2. 在 32 位平台 MongoDB 不允许数据库文件（累计总和）超过 2G，而 64 位平台没有这个限制。

怎么安装 MongoDB 数据库呢？下面将分别介绍 Windows 和 Linux 版本的安装方法

2.1 Windows 平台的安装

步骤一： 下载 MongoDB

url 下载地址： http://downloads.mongodb.org/win32/mongodb-win32-i386-1.8.1.zip

步骤二： 设置 MongoDB 程序存放目录

将其解压到 c:\，再重命名为 mongo，路径为 c:\mongo

步骤三： 设置数据文件存放目录

在 c:盘建一个 db 文件夹，路径 c:\db

步骤四： 启动 MongoDB 服务

进入 cmd 提示符控制台，c:\mongo\bin\mongod.exe --dbpath=c:\db

```
C:\mongo\bin>C:\mongo\bin\mongod --dbpath=c:\db
Sun Apr 10 22:34:09 [initandlisten] MongoDB starting : pid=5192 port=27017 dbpat
h=d:\data\db 32-bit
** NOTE: when using MongoDB 32 bit, you are limited to about 2 gigabytes of data
**       see http://blog.mongodb.org/post/137788967/32-bit-limitations
**       with --dur, the limit is lower
……
Sun Apr 10 22:34:09 [initandlisten] waiting for connections on port 27017
Sun Apr 10 22:34:09 [websvr] web admin interface listening on port 28017
```

MongoDB 服务端的默认监听端口是 27017

步骤五：将 MongoDB 作为 Windows 服务随机启动

先创建 C:\mongo\logs\mongodb.log 文件，用于存储 MongoDB 的日志文件，再安装系统服务。

```
C:\mongo\bin>C:\mongo\bin\mongod --dbpath=c:\ db --logpath=c:\mongo\lo
gs\mongodb.log --install
all output going to: c:\mongo\logs\mongodb.log
Creating service MongoDB.
Service creation successful.
Service can be started from the command line via 'net start "MongoDB"'.
C:\mongo\bin>net start mongodb
Mongo DB 服务已经启动成功。
C:\mongo\bin>
```

步骤六：客户端连接验证

新打开一个 CMD 输入：c:\mongo\bin\mongo，如果出现下面提示，那么您就可以开始 MongoDB 之旅了

```
C:\mongo\bin>c:\mongo\bin\mongo
MongoDB shell version: 1.8.1
connecting to: test
>
```

步骤七：查看 MongoDB 日志

查看 C:\mongo\logs\mongodb.log 文件，即可对 MongoDB 的运行情况进行查看或排错了，这样就完成了 Windows 平台的 MongoDB 安装。

2.2 Linux 平台的安装

步骤一：下载 MongoDB

下载安装包：curl -O http://fastdl.mongodb.org/linux/mongodb-linux-i686-1.8.1.tgz

步骤二：设置 MongoDB 程序存放目录

将其解压到/Apps，再重命名为 mongo，路径为/Apps/mongo

步骤三：设置数据文件存放目录

建立/data/db 的目录, mkdir –p /data/db

步骤四：启动 MongoDB 服务

/Apps/mongo/bin/mongod --dbpath=/data/db

```
[root@localhost ~]# /Apps/mongo/bin/mongod --dbpath=/data/db
Sun Apr  8 22:41:06 [initandlisten] MongoDB starting : pid=13701 port=27017 dbpath=/data/db
32-bit
** NOTE: when using MongoDB 32 bit, you are limited to about 2 gigabytes of data
**       see http://blog.mongodb.org/post/137788967/32-bit-limitations
**       with --dur, the limit is lower
……
Sun Apr  8 22:41:06 [initandlisten] waiting for connections on port 27017
Sun Apr  8 22:41:06 [websvr] web admin interface listening on port 28017
```

MongoDB 服务端的默认连接端口是 27017

步骤五：将 MongoDB 作为 Linux 服务随机启动

先创建/Apps/mongo/logs/mongodb.log 文件，用于存储 MongoDB 的日志文件

vi /etc/rc.local，使用 vi 编辑器打开配置文件，并在其中加入下面一行代码

/Apps/mongo/bin/mongod --dbpath=/data/db --logpath=/Apps/mongo/logs/mongodb.log

步骤六：客户端连接验证

新打开一个 Session 输入：/Apps/mongo/bin/mongo，如果出现下面提示，那么您就可以开始 MongoDB 之旅了

```
[root@localhost ~]# /Apps/mongo/bin/mongo
MongoDB shell version: 1.8.1
connecting to: test
>
```

步骤七：查看 MongoDB 日志

查看/Apps/mongo/logs/mongodb.log 文件，即可对 MongoDB 的运行状况进行查看或分析了

```
 [root@localhost logs]# ll
总计 0
-rw-r--r-- 1 root root 0 04-08 20:15 mongodb.log
[root@localhost logs]#
```

以上的几个步骤就 OK 了!! 这样一个简单的 MongoDB 数据库就可以畅通无阻地运行起来了。

第三章 体系结构

MongoDB 是一个可移植的数据库，它在流行的每一个平台上都可以使用，即所谓的跨平台特性。在不同的操作系统上虽然略有差别，但是从整体构架上来看，MongoDB 在不同的平台上是一样的，如数据逻辑结构和数据的存储等等。

一个运行着的 MongoDB 数据库就可以看成是一个 MongoDB Server，该 Server 由实例和数据库组成，在一般的情况下一个 MongoDB Server 机器上包含一个实例和多个与之对应的数据库，但是在特殊情况下，如硬件投入成本有限或特殊的应用需求，也允许一个 Server 机器上可以有多个实例和多个数据库。

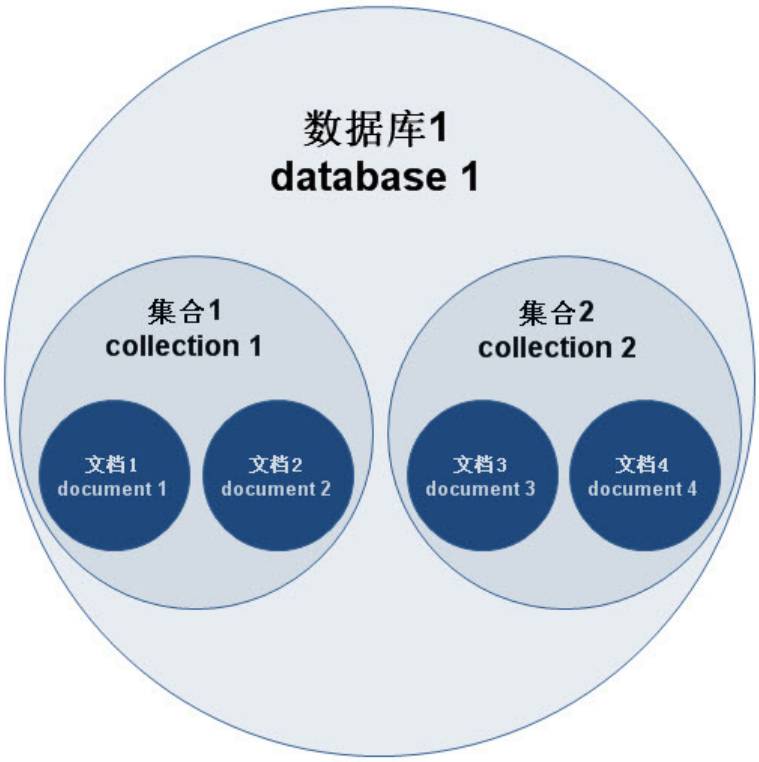
MongoDB 中一系列物理文件（数据文件，日志文件等）的集合或与之对应的逻辑结构（集合，文档等）被称为数据库，简单的说，就是数据库是由一系列与磁盘有关系的物理文件的组成。

3.1 数据逻辑结构

很多人在学习 MongoDB 体系结构的时候会遇到各种各样的问题，我在这里给大家简单的介绍一下 MongoDB 体系结构之一的逻辑结构。MongoDB 的逻辑结构是一种层次结构。主要由：文档(document)、集合(collection)、数据库(database)这三部分组成的。逻辑结构是面向用户的，用户使用 MongoDB 开发应用程序使用的就是逻辑结构。

- MongoDB 的文档（document），相当于关系数据库中的一行记录。
- 多个文档组成一个集合（collection），相当于关系数据库的表。
- 多个集合（collection），逻辑上组织在一起，就是数据库（database）。
- 一个 MongoDB 实例支持多个数据库（database）。

文档(document)、集合(collection)、数据库(database)的层次结构如下图:



对于习惯了关系型数据库的朋友们，我将 MongoDB 与关系型数据库的逻辑结构进行了对比，以便让大家更深刻的理解 MongoDB 的逻辑结构

逻辑结构对比	
MongoDB	**关系型数据库**
文档(document)	行(row)
集合(collection)	表(table)
数据库(database)	数据库(database)

3.2 数据存储结构

MongoDB 对国内用户来说比较新，它就像是一个黑盒子，但是如果对于它内部的数据存储了解多一些的话，那么将会很快的理解和驾驭 MongoDB，让它发挥它更大的作用。

MongoDB 的默认数据目录是/data/db，它负责存储所有的 MongoDB 的数据文件。在 MongoDB 内部，每个数据库都包含一个.ns 文件和一些数据文件，而且这些数据文件会随着数据量的增加而变得越来越多。所以如果系统中有一个叫做 foo 的数据库，那么构成 foo 这个数据库的文件就会由 foo.ns，foo.0，foo.1，foo.2 等等组成，具体如下:

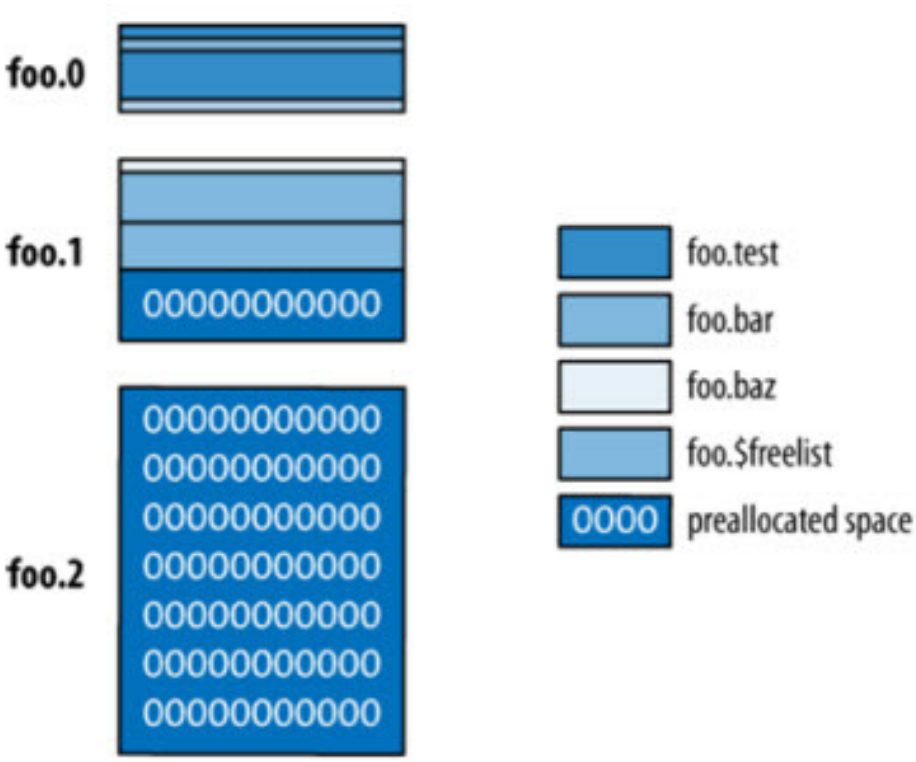
```
[root@localhost db]# ll /data/db/
总计 196844
-rw------- 1 root root 16777216 04-15 16:33 admin.0
-rw------- 1 root root 33554432 04-15 16:33 admin.1
-rw------- 1 root root 16777216 04-15 16:33 admin.ns
-rw------- 1 root root 16777216 04-21 17:30 foo.0
-rw------- 1 root root 33554432 04-21 17:30 foo.1
-rw------- 1 root root 67108864 04-21 17:30 foo.2
-rw------- 1 root root 16777216 04-21 17:30 foo.ns
-rwxr-xr-x 1 root root        6 04-21 17:16 mongod.lock
-rw------- 1 root root 16777216 04-15 16:30 test.0
-rw------- 1 root root 33554432 04-15 16:30 test.1
-rw------- 1 root root 16777216 04-15 16:30 test.ns
drwxr-xr-x 2 root root     4096 04-21 17:30 _tmp
```

MongoDB 内部有预分配空间的机制，每个预分配的文件都用 0 进行填充，由于有了这个机制, MongoDB 始终保持额外的空间和空余的数据文件，从而有效避免了由于数据暴增而带来的磁盘压力过大的问题。

由于表中数据量的增加，数据文件每新分配一次，它的大小都会是上一个数据文件大小的 2 倍，每个数据文件最大 2G。这样的机制有利于防止较小的数据库浪费过多的磁盘空间，同时又能保证较大的数据库有相应的预留空间使用。

数据库的每张表都对应一个命名空间，每个索引也有对应的命名空间。这些命名空间的元数据都集中在*.ns 文件中。

在下图中，foo 这个数据库包含 3 个文件用于存储表和索引数据，foo.2 文件属于预分配的空文件。foo.0 和 foo.1 这两个数据文件被分为了相应的盘区对应不同的名字空间。



上图显示了命名空间和盘区的关系。每个命名空间可以包含多个不同的盘区，这些盘区并不是连续的。与数据文件的增长相同，每一个命名空间对应的盘区大小的也是随着分配的次数不断增长的。这样做的目的是为了平衡命名空间浪费的空间与保持某一个命名空间中数据的连续性。上图中还有一个需要注意的命名空间：$freelist，这个命名空间用于记录不再使用的盘区（被删除的 Collection 或索引）。每当命名空间需要分配新的盘区的时候，都会先查看$freelist 是否有大小合适的盘区可以使用，这样就回收空闲的磁盘空间。

第四章 快速入门

MongoDB Shell 是 MongoDB 自带的交互式 Javascript shell，用来对 MongoDB 进行操作和管理的交互式环境。

使用 "./mongo --help" 可查看相关连接参数，下面将从常见的操作，如插入，查询，修改，删除等几个方面阐述 MongoDB shell 的用法。

4.1 启动数据库

MongoDB 安装、配置完后，必须先启动它，然后才能使用它。怎么启动它呢？下面分别展示了 3 种方式来启动实例。

4.1.1 命令行方式启动

MongoDB 默认存储数据目录为/data/db/ (或者 c:\data\db)，默认端口 27017，默认 HTTP 端

口 28017。当然你也可以修改成不同目录,只需要指定 dbpath 参数: /Apps/mongo/bin/mongod --dbpath=/data/db

```
[root@localhost ~]# /Apps/mongo/bin/mongod --dbpath=/data/db
Sun Apr  8 22:41:06 [initandlisten] MongoDB starting : pid=13701 port=27017 dbpath=/data/db 32-bit
……
Sun Apr  8 22:41:06 [initandlisten] waiting for connections on port 27017
Sun Apr  8 22:41:06 [websvr] web admin interface listening on port 28017
```

4.1.2 配置文件方式启动

如果是一个专业的 DBA，那么实例启动时会加很多的参数以便使系统运行的非常稳定，这样就可能会在启动时在 mongod 后面加一长串的参数，看起来非常混乱而且不好管理和维护，那么有什么办法让这些参数有条理呢？MongoDB 也支持同 mysql 一样的读取启动配置文件的方式来启动数据库，配置文件的内容如下:

```
[root@localhost bin]# cat /etc/mongodb.cnf
dbpath=/data/db/
```

启动时加上”-f”参数，并指向配置文件即可

```
[root@localhost bin]# ./mongod -f /etc/mongodb.cnf
Mon May 28 18:27:18 [initandlisten] MongoDB starting : pid=18481 port=27017 dbpath=/data/db/ 32-bit
……
Mon May 28 18:27:18 [initandlisten] waiting for connections on port 27017
Mon May 28 18:27:18 [websvr] web admin interface listening on port 28017
```

4.1.3 Daemon 方式启动

大家可以注意到上面的两种方式都慢在前台启动 MongoDB 进程，但当启动 MongoDB 进程的 session 窗口不小心关闭时，MongoDB 进程也将随之停止，这无疑是非常不安全的，幸好 MongoDB 提供了一种后台 Daemon 方式启动的选择，只需加上一个”--fork”参数即可，这就使我们可以更方便的操作数据库的启动，但如果用到了”--fork”参数就必须也启用”--logpath”参数，这是强制的

```
[root@localhost ~]# /Apps/mongo/bin/mongod --dbpath=/data/db --fork
--fork has to be used with --logpath
[root@localhost ~]# /Apps/mongo/bin/mongod --dbpath=/data/db --logpath=/data/log/r3.log --fork
all output going to: /data/log/r3.log
forked process: 19528
[root@localhost ~]#
```

4.1.4 mongod 参数说明

最简单的，通过执行 mongod 即可以启动 MongoDB 数据库服务，mongod 支持很多的参数，但都有默认值，其中最重要的是需要指定数据文件路径，或者确保默认的/data/db 存在并且有访问权限，否则启动后会自动关闭服务。Ok，那也就是说，只要确保 dbpath 就可以启动 MongoDB 服务了

mongod 的主要参数有：

- **dbpath:**
  数据文件存放路径，每个数据库会在其中创建一个子目录，用于防止同一个实例多次运行的 mongod.lock 也保存在此目录中。
- **logpath**
  错误日志文件
- **logappend**
  错误日志采用追加模式（默认是覆写模式）
- **bind_ip**
  对外服务的绑定 ip，一般设置为空，及绑定在本机所有可用 ip 上，如有需要可以单独指定
- **port**
  对外服务端口。Web 管理端口在这个 port 的基础上+1000
- **fork**
  以后台 Daemon 形式运行服务
- **journal**
  开启日志功能，通过保存操作日志来降低单机故障的恢复时间，在 1.8 版本后正式加入，取代在 1.7.5 版本中的 dur 参数。
- **syncdelay**
  系统同步刷新磁盘的时间，单位为秒，默认是 60 秒。
- **directoryperdb**
  每个 db 存放在单独的目录中，建议设置该参数。与 MySQL 的独立表空间类似
- **maxConns**
  最大连接数
- **repairpath**
  执行 repair 时的临时目录。在如果没有开启 journal，异常 down 机后重启，必须执行 repair 操作。

在源代码中，mongod 的参数分为一般参数，windows 参数，replication 参数，replica set 参数，以及隐含参数。上面列举的都是一般参数。如果要配置 replication，replica set 等，还需要设置对应的参数，这里先不展开，后续会有专门的章节来讲述。执行 mongod --help 可以看到对大多数参数的解释，但有一些隐含参数，则只能通过看代码来获得(见 db.cpp po::options_description hidden_options(“Hidden options”);)，隐含参数一般要么是还在开发中，要么是准备废弃，因此在生产环境中不建议使用。

可能你已经注意到，mongod 的参数中，没有设置内存大小相关的参数，是的，MongoDB 使

用 os mmap 机制来缓存数据文件数据，自身目前不提供缓存机制。这样好处是代码简单，mmap 在数据量不超过内存时效率很高。但是数据量超过系统可用内存后，则写入的性能可能不太稳定，容易出现大起大落，不过在最新的 1.8 版本中，这个情况相对以前的版本已经有了一定程度的改善。

这么多参数，全面写在命令行中则容易杂乱而不好管理。因此，mongod 支持将参数写入到一个配置文本文件中，然后通过 config 参数来引用此配置文件：
./mongod --config /etc/mongo.cnf

4.2 停止数据库

MongoDB 提供的停止数据库命令也非常丰富，如果 Control-C、发送 shutdownServer()指令及发送 Unix 系统中断信号等

4.2.1 Control-C

如果处理连接状态，那么直接可以通过 Control-C 的方式去停止 MongoDB 实例，具体如下:

```
[root@localhost ~]#    /Apps/mongo/bin/mongo --port 28013
MongoDB shell version: 1.8.1
connecting to: 127.0.0.1:28013/test
> use test
switched to db test
> ^C[root@localhost ~]#
```

4.2.2 shutdownServer()指令

如果处理连接状态，那么直接可以通过在 admin 库中发送 db.shutdownServer()指令去停止 MongoDB 实例，具体如下:

```
[root@localhost ~]#    /Apps/mongo/bin/mongo --port 28013
MongoDB shell version: 1.8.1
connecting to: 127.0.0.1:28013/test
> use admin
switched to db admin
> db.shutdownServer()
Thu May 31 23:22:00 DBClientCursor::init call() failed
Thu May 31 23:22:00 query failed : admin.$cmd { shutdown: 1.0 } to: 127.0.0.1:28013
server should be down...
Thu May 31 23:22:00 trying reconnect to 127.0.0.1:28013
Thu May 31 23:22:00 reconnect 127.0.0.1:28013 failed couldn't connect to server 127.0.0.1:28013
Thu May 31 23:22:00 Error: error doing query: unknown shell/collection.js:150
>
```

4.2.3 Unix 系统指令

在找到实例的进程后，可能通过发送 kill -2 PID 或 kill -15 PID 来停止进程

```
[root@localhost ~]# ps aux|grep mongod
root 19269 0.3 1.3 76008 3108 Sl 23:24 0:00 /Apps/mongo/bin/mongod --fork --port 28013
[root@localhost ~]# kill -2 19269
```

注意:
不要用 **kill -9 PID** 来杀死 MongoDB 进程，这样可以会导致 MongoDB 的数据损坏

4.3 连接数据库

现在我们就可以使用自带的 MongoDB shell 工具来操作数据库了. (我们也可以使用各种编程语言的驱动来使用 MongoDB, 但自带的 MongoDB shell 工具可以方便我们管理数据库)。
新打开一个 Session 输入：/Apps/mongo/bin/mongo，如果出现下面提示，那么您就说明连接上数据库了，可以进行操作了

```
[root@localhost ~]# /Apps/mongo/bin/mongo
MongoDB shell version: 1.8.1
connecting to: test
>
```

默认 shell 连接的是本机 localhost 上面的 test 库，"connecting to:" 这个会显示你正在使用的数据库的名称. 想换数据库的话可以用”use mydb”来实现。

4.4 插入记录

下面我们来建立一个 test 的集合并写入一些数据. 建立两个对象 j 和 t ，并保存到集合中去. 在例子里 “>” 来表示是 shell 输入提示符

```
> j = { name : "mongo" };
{"name" : "mongo"}
> t = { x : 3 };
{ "x" : 3 }
> db.things.save(j);
> db.things.save(t);
> db.things.find();
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
>
```

有几点需要注意一下:

- 不需要预先创建一个集合. 在第一次插入数据时候会自动创建.

- 在文档中其实可以存储任何结构的数据，当然在实际应用我们存储的还是相同类型文档的集合. 这个特性其实可以在应用里很灵活，你不需要类似 alter table 语句来修改你的数据结构
- 每次插入数据时候集合中都会有一个 ID，名字叫 _id.

下面再加点数据:

```
> for( var i = 1; i < 10; i++ ) db.things.save( { x:4, j:i } ); > db.things.find();
{"name" : "mongo" , "_id" : ObjectId("497cf60751712cf7758fbdbb")}
{"x" : 3 , "_id" : ObjectId("497cf61651712cf7758fbdbc")}
{"x" : 4 , "j" : 1 , "_id" : ObjectId("497cf87151712cf7758fbdbd")}
{"x" : 4 , "j" : 2 , "_id" : ObjectId("497cf87151712cf7758fbdbe")}
{"x" : 4 , "j" : 3 , "_id" : ObjectId("497cf87151712cf7758fbdbf")}
{"x" : 4 , "j" : 4 , "_id" : ObjectId("497cf87151712cf7758fbdc0")}
{"x" : 4 , "j" : 5 , "_id" : ObjectId("497cf87151712cf7758fbdc1")}
{"x" : 4 , "j" : 6 , "_id" : ObjectId("497cf87151712cf7758fbdc2")}
{"x" : 4 , "j" : 7 , "_id" : ObjectId("497cf87151712cf7758fbdc3")}
{"x" : 4 , "j" : 8 , "_id" : ObjectId("497cf87151712cf7758fbdc4")}
```

请注意一下，这里循环次数是 10，但是只显示到第 8 条，还有 2 条数据没有显示. 如果想继续查询下面的数据只需要使用”it”命令，就会继续显示下面的数据:

```
{ "_id" : ObjectId("4c220a42f3924d31102bd866"), "x" : 4, "j" : 17 }
{ "_id" : ObjectId("4c220a42f3924d31102bd867"), "x" : 4, "j" : 18 }
has more
> it
{ "_id" : ObjectId("4c220a42f3924d31102bd868"), "x" : 4, "j" : 19 }
{ "_id" : ObjectId("4c220a42f3924d31102bd869"), "x" : 4, "j" : 20 }
```

从技术上讲 find() 返回一个游标对象. 但在上面的例子里，并没有拿到一个游标的变量. 所以 shell 自动遍历游标，返回一个初始化的 set，并允许我们继续用 it 迭代输出.
当然我们也可以直接用游标来输出，不过这个是”游标”部分的内容了.

4.5 _id key

MongoDB 支持的数据类型中，_id 是其自有产物，下面对其做些简单的介绍。

存储在 MongoDB 集合中的每个文档（document）都有一个默认的主键_id，这个主键名称是固定的，它可以是 MongoDB 支持的任何数据类型，默认是 ObjectId。在关系数据库 schema 设计中，主键大多是数值型的，比如常用的 int 和 long，并且更通常的是主键的取值由数据库自增获得，这种主键数值的有序性有时也表明了某种逻辑。反观 MongoDB，它在设计之初就定位于分布式存储系统，所以它原生的不支持自增主键。

_id key 举例说明 ：
当我们在往一个集合中写入一条文档时，系统会自动生成一个名为_id 的 key.如：

```
> db.c1.find()
{ "_id" : ObjectId("4fb5faaf6d0f9d8ea3fc91a8"), "name" : "Tony", "age" : 20 }
{ "_id" : ObjectId("4fb5fab96d0f9d8ea3fc91a9"), "name" : "Joe", "age" : 10 }
```

这里多出了一个类型为 ObjectId 的 key ,在插入时并没有指定，这有点类似 Oracle 的 rowid 的信息，属于自动生成的。

在 MongoDB 中，每一个集合都必须有一个叫做_id 的字段,字段类型默认是 ObjectId ,换句话说，字段类型可以不是 ObjectId,例如:

```
> db.c1.find()
{ "_id" : ObjectId("4fb5faaf6d0f9d8ea3fc91a8"), "name" : "Tony", "age" : 20 }
{ "_id" : ObjectId("4fb5fab96d0f9d8ea3fc91a9"), "name" : "Joe", "age" : 10 }
{ "_id" : 3, "name" : "Bill", "age" : 55 }
```

虽然_id 的类型可以自由指定,但是在同一个集合中必须唯一，如果插入重复的值的话，系统将会抛出异常，具体如下:

```
> db.c1.insert({_id:3, name:"Bill_new", age:55})
E11000 duplicate key error index: test.c1.$_id_  dup key: { : 3.0 }
>
```

因为前面已经插入了一条_id=3 的记录，所以再插入相同的文档就不允许了。

4.6 查询记录

4.6.1 普通查询

在没有深入查询之前，我们先看看怎么从一个查询中返回一个游标对象. 可以简单的通过 find() 来查询，他返回一个任意结构的集合. 如果实现特定的查询稍后讲解.
实现上面同样的查询，然后通过 while 来输出:

```
> var cursor = db.things.find();
> while (cursor.hasNext()) printjson(cursor.next());
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
```

上面的例子显示了游标风格的迭代输出. hasNext() 函数告诉我们是否还有数据，如果有则可以调用 next() 函数.

当我们使用的是 JavaScript shell，可以用到 JS 的特性, forEach 就可以输出游标了. 下面的例子就是使用 forEach() 来循环输出: forEach() 必须定义一个函数供每个游标元素调用.

```
> db.things.find().forEach(printjson);
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
```

```
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
```

在 MongoDB shell 里，我们也可以把游标当作数组来用:

```
> var cursor = db.things.find();
> printjson(cursor[4]);
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
```

使用游标时候请注意占用内存的问题，特别是很大的游标对象，有可能会内存溢出．所以应该用迭代的方式来输出．下面的示例则是把游标转换成真实的数组类型:

```
> var arr = db.things.find().toArray();
> arr[5];
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
```

请注意这些特性只是在 MongoDB shell 里使用，而不是所有的其他应用程序驱动都支持．MongoDB 游标对象不是没有快照，如果有其他用户在集合里第一次或者最后一次调用 next()，你可能得不到游标里的数据．所以要明确的锁定你要查询的游标．

4.6.2 条件查询

到这里我们已经知道怎么从游标里实现一个查询并返回数据对象，下面就来看看怎么根据指定的条件来查询．
下面的示例就是说明如何执行一个类似 SQL 的查询，并演示了怎么在 MongoDB 里实现．这是在 MongoDB shell 里查询，当然你也可以用其他的应用程序驱动或者语言来实现:

SELECT * FROM things WHERE name="mongo"

```
> db.things.find({name:"mongo"}).forEach(printjson);
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
```

SELECT * FROM things WHERE x=4

```
> db.things.find({x:4}).forEach(printjson);
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "x" : 4, "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "x" : 4, "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "x" : 4, "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "x" : 4, "j" : 5 }
```

查询条件是 { a:A, b:B, … } 类似 “where a==A and b==B and …”．

上面显示的是所有的元素，当然我们也可以返回特定的元素，类似于返回表里某字段的值，只需要在 find({x:4}) 里指定元素的名字

SELECT j FROM things WHERE x=4

```
> db.things.find({x:4}, {j:true}).forEach(printjson);
```

```
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "j" : 1 }
{ "_id" : ObjectId("4c220a42f3924d31102bd857"), "j" : 2 }
{ "_id" : ObjectId("4c220a42f3924d31102bd858"), "j" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd859"), "j" : 4 }
{ "_id" : ObjectId("4c220a42f3924d31102bd85a"), "j" : 5 }
```

4.6.3 findOne()语法

为了方便考虑，MongoDB shell 避免游标可能带来的开销，提供一个 findOne() 函数. 这个函数和 find() 函数一样，不过它返回的是游标里第一条数据，或者返回 null，即空数据.

作为一个例子，name=”mongo” 可以用很多方法来实现，可以用 next() 来循环游标或者当做数组返回第一个元素.

但是用 findOne() 方法则更简单和高效:

```
> printjson(db.things.findOne({name:"mongo"}));
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
```

4.6.4 通过 limit 限制结果集数量

如果需要限制结果集的长度，那么可以调用 limit 方法.
这是强烈推荐解决性能问题的方法，就是通过限制条数来减少网络传输，例如:

```
> db.things.find().limit(3);
{ "_id" : ObjectId("4c2209f9f3924d31102bd84a"), "name" : "mongo" }
{ "_id" : ObjectId("4c2209fef3924d31102bd84b"), "x" : 3 }
{ "_id" : ObjectId("4c220a42f3924d31102bd856"), "x" : 4, "j" : 1 }
```

4.7 修改记录

将 name 是 mongo 的记录的 name 修改为 mongo_new

```
> db.things.update({name:"mongo"},{$set:{name:"mongo_new"}});
```

我们来查询一下是否改过来了

```
> db.things.find();
{ "_id" : ObjectId("4faa9e7dedd27e6d86d86371"), "x" : 3 }
{ "_id" : ObjectId("4faa9e7bedd27e6d86d86370"), "name" : "mongo_new" }
```

4.8 删除记录

将用户 name 是 mongo_new 的记录从集合 things 中删除

```
> db.things.remove({name:"mongo_new"});
> db.things.find();
{ "_id" : ObjectId("4faa9e7dedd27e6d86d86371"), "x" : 3 }
```

经验证，该记录确实被删除了

4.9 常用工具集

MongoDB 在 bin 目录下提供了一系列有用的工具，这些工具提供了 MongoDB 在运维管理上的方便。

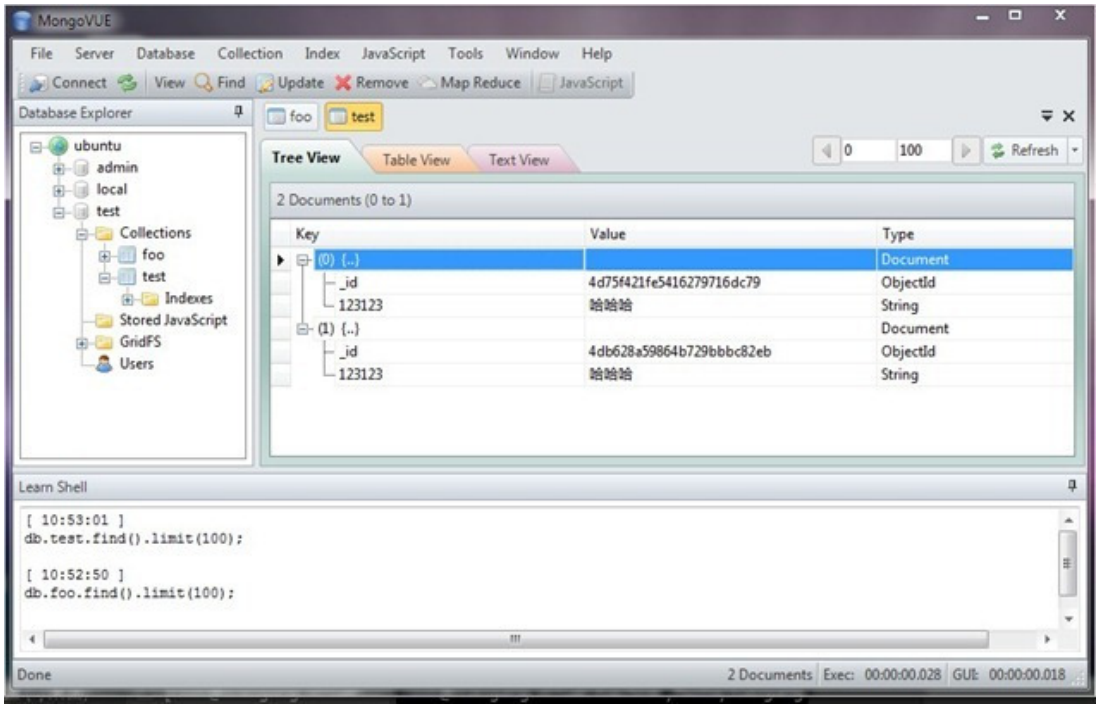
- bsondump: 将 bson 格式的文件转储为 json 格式的数据
- mongo: 客户端命令行工具，其实也是一个 js 解释器，支持 js 语法
- mongod: 数据库服务端，每个实例启动一个进程，可以 fork 为后台运行
- mongodump/ mongorestore: 数据库备份和恢复工具
- mongoexport/ mongoimport: 数据导出和导入工具
- mongofiles: GridFS 管理工具，可实现二制文件的存取
- mongos: 分片路由，如果使用了 sharding 功能，则应用程序连接的是 mongos 而不是 mongod
- mongosniff: 这一工具的作用类似于 tcpdump，不同的是他只监控 MongoDB 相关的包请求，并且是以指定的可读性的形式输出
- mongostat: 实时性能监控工具

4.10 客户端 GUI 工具

看一个产品是否得到认可,可以从一个侧面看其第三方工具的数量和成熟程度,下面我们就来细数一下 MongoDB 常用的 GUI 管理工具。

4.10.1 MongoVUE

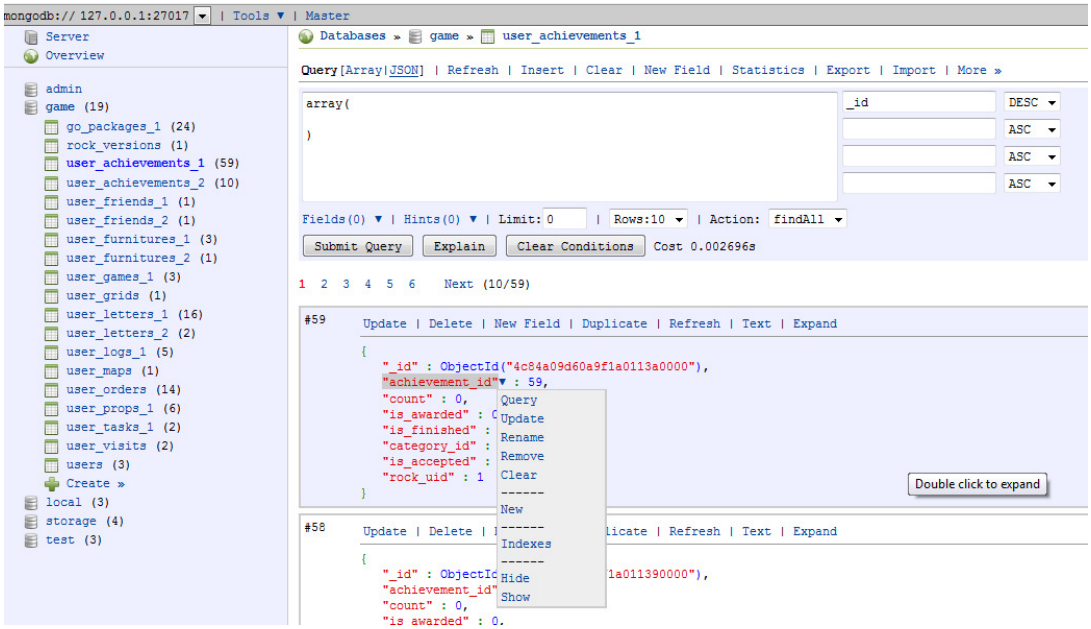
主页：http://www.mongovue.com/
一个桌面程序，提供了对 MongoDB 数据库的基本操作，如查看、查询、更新、删除等，简单易用，但是功能还比较弱，以后发展应该不错。



4.10.2 RockMongo

主页：http://code.google.com/p/rock-php/

RockMongo 是一个 PHP5 写的 MongoDB 管理工具。



主要特征:

使用宽松的 New BSD License 协议

速度快，安装简单

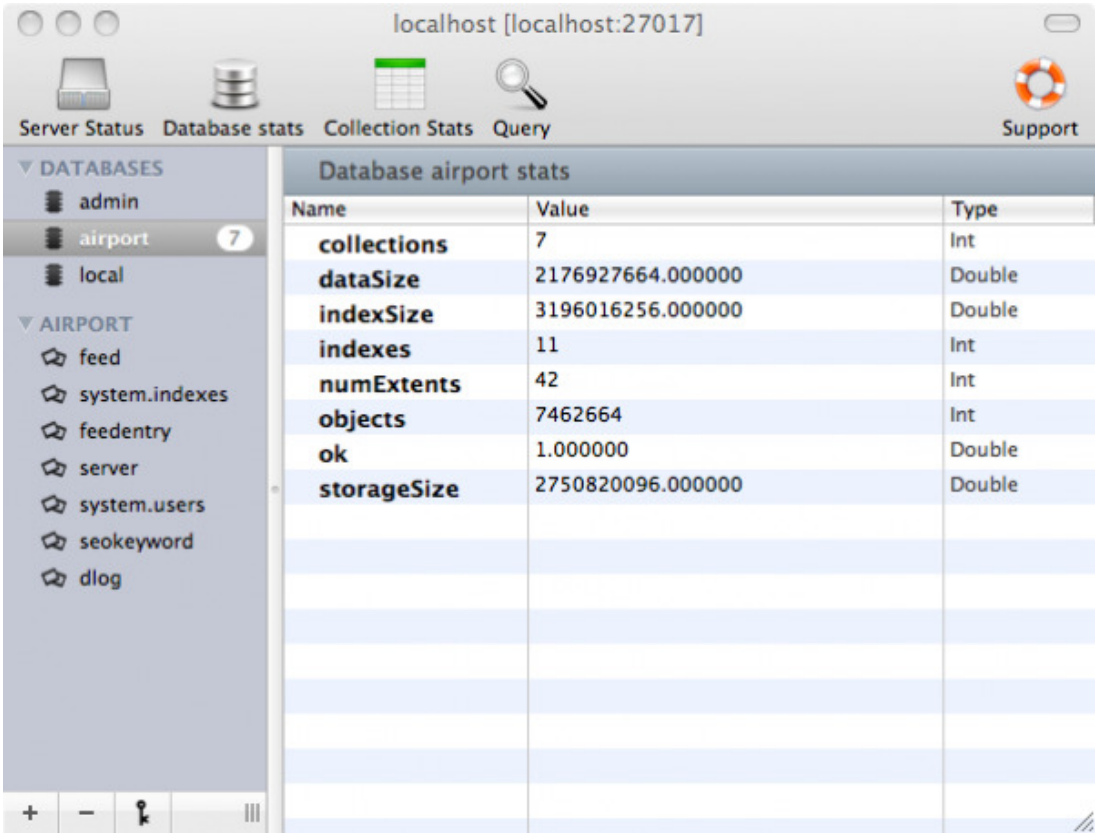
支持多语言（目前提供中文、英文、日文、巴西葡萄牙语、法语、德语）

系统可以配置多个主机，每个主机可以有多个管理员，需要管理员密码才能登入操作，确保数据库的安全性

服务器信息 (WEB 服务器, PHP, PHP.ini 相关指令 ...)状态
数据查询，创建和删除
执行命令和 Javascript 代码

4.10.3 MongoHub

主页：https://github.com/bububa/MongoHub
MongoHub 是一个针对 Mac 平台的 MongoDB 图形管理客户端，可用来管理 MongoDB 数据的应用程序。



第二部分 应用篇

本章将结合实际应用，重点阐述一些实际工作中最常用的方法。

第五章 高级查询

面向文档的 NoSQL 数据库主要解决的问题不是高性能的并发读写，而是保证海量数据存储的同时，具有良好的查询性能。

MongoDB 最大的特点是他支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

最后由于 MongoDB 可以支持复杂的数据结构，而且带有强大的数据查询功能，因此非常受到欢迎，很多项目都考虑用 MongoDB 来替代 MySQL 等传统数据库来实现不是特别复杂的 Web 应用。由于数据量实在太大，所以迁移到了 MongoDB 上面，数据查询的速度得到了非常显著的提升。

下面将介绍一些高级查询语法:

5.1 条件操作符

5.1 条件操作符

<, <=, >, >= 这个操作符就不用多解释了，最常用也是最简单的

```
db.collection.find({ "field" : { $gt: value } } );      // 大于:       field > value
db.collection.find({ "field" : { $lt: value } } );      // 小于:        field < value
db.collection.find({ "field" : { $gte: value } } );    // 大于等于: field >= value
db.collection.find({ "field" : { $lte: value } } );    // 小于等于: field <= value
```

如果要同时满足多个条件，可以这样做

```
db.collection.find({ "field" : { $gt: value1, $lt: value2 } } );       // value1 < field < value
```

5.2 $all 匹配所有

这个操作符跟 SQL 语法的 in 类似，但不同的是，in 只需满足( )内的某一个值即可，而$all 必须满足[ ]内的所有值，例如:

```
db.users.find({age : {$all : [6, 8]}});
```

可以查询出 {name: 'David', age: 26, age: [6, 8, 9] }

但查询不出 {name: 'David', age: 26, age: [6, 7, 9] }

5.3 $exists 判断字段是否存在

查询所有存在 age 字段的记录

```
db.users.find({age: {$exists: true}});
```

查询所有不存在 name 字段的记录

```
db.users.find({name: {$exists: false}});
```

举例如下:

C1 表的数据如下:

```
> db.c1.find();
{ "_id" : ObjectId("4fb4a773afa87dc1bed9432d"), "age" : 20, "length" : 30 }
{ "_id" : ObjectId("4fb4a7e1afa87dc1bed9432e"), "age_1" : 20, "length_1" : 30 }
```

查询存在字段 age 的数据

```
> db.c1.find({age:{$exists:true}});
{ "_id" : ObjectId("4fb4a773afa87dc1bed9432d"), "age" : 20, "length" : 30 }
```

可以看出只显示出了有 age 字段的数据，age_1 的数据并没有显示出来

5.4 Null 值处理

Null 值的处理稍微有一点奇怪，具体看下面的样例数据：

```
> db.c2.find()
{ "_id" : ObjectId("4fc34bb81d8a39f01cc17ef4"), "name" : "Lily", "age" : null }
{ "_id" : ObjectId("4fc34be01d8a39f01cc17ef5"), "name" : "Jacky", "age" : 23 }
{ "_id" : ObjectId("4fc34c1e1d8a39f01cc17ef6"), "name" : "Tom", "addr" : 23 }
```

其中”Lily”的 age 字段为空，Tom 没有 age 字段，我们想找到 age 为空的行，具体如下：

```
> db.c2.find({age:null})
{ "_id" : ObjectId("4fc34bb81d8a39f01cc17ef4"), "name" : "Lily", "age" : null }
{ "_id" : ObjectId("4fc34c1e1d8a39f01cc17ef6"), "name" : "Tom", "addr" : 23 }
```

奇怪的是我们以为只能找到”Lily”，但”Tom”也被找出来了，所以”null”不仅能找到它自身，连不存在 age 字段的记录也找出来了。那么怎么样才能只找到”Lily”呢?我们用 exists 来限制一下即可:

```
> db.c2.find({age:{"$in":[null], "$exists":true}})

{ "_id" : ObjectId("4fc34bb81d8a39f01cc17ef4"), "name" : "Lily", "age" : null }
```

这样如我们期望一样，只有”Lily”被找出来了。

5.5 $mod 取模运算

查询 age 取模 10 等于 0 的数据

db.student.find({ age: { $mod : [10 , 1] } })

举例如下:

C1 表的数据如下:

```
> db.c1.find()
{ "_id" : ObjectId("4fb4af85afa87dc1bed94330"), "age" : 7, "length_1" : 30 }
{ "_id" : ObjectId("4fb4af89afa87dc1bed94331"), "age" : 8, "length_1" : 30 }
{ "_id" : ObjectId("4fb4af8cafa87dc1bed94332"), "age" : 6, "length_1" : 30 }
```

查询 age 取模 6 等于 1 的数据

```
>   db.c1.find({age: {$mod : [ 6 , 1 ] } })
{ "_id" : ObjectId("4fb4af85afa87dc1bed94330"), "age" : 7, "length_1" : 30 }
```

可以看出只显示出了 age 取模 6 等于 1 的数据，其它不符合规则的数据并没有显示出来

5.6 $ne 不等于

查询 x 的值不等于 3 的数据
db.things.find({ x : { $ne : 3 } });
举例如下:
C1 表的数据如下:

```
> db.c1.find()
{ "_id" : ObjectId("4fb4af85afa87dc1bed94330"), "age" : 7, "length_1" : 30 }
{ "_id" : ObjectId("4fb4af89afa87dc1bed94331"), "age" : 8, "length_1" : 30 }
{ "_id" : ObjectId("4fb4af8cafa87dc1bed94332"), "age" : 6, "length_1" : 30 }
```

查询 age 的值不等于 7 的数据

```
> db.c1.find( { age : { $ne : 7 } } );
{ "_id" : ObjectId("4fb4af89afa87dc1bed94331"), "age" : 8, "length_1" : 30 }
{ "_id" : ObjectId("4fb4af8cafa87dc1bed94332"), "age" : 6, "length_1" : 30 }
```

可以看出只显示出了 age 等于 7 的数据，其它不符合规则的数据并没有显示出来

5.7 $in 包含

与 sql 标准语法的用途是一样的，即要查询的是一系列枚举值的范围内
查询 x 的值在 2,4,6 范围内的数据
db.things.find({x:{$in: [2,4,6]}});
举例如下:
C1 表的数据如下:

```
> db.c1.find()
{ "_id" : ObjectId("4fb4af85afa87dc1bed94330"), "age" : 7, "length_1" : 30 }
{ "_id" : ObjectId("4fb4af89afa87dc1bed94331"), "age" : 8, "length_1" : 30 }
{ "_id" : ObjectId("4fb4af8cafa87dc1bed94332"), "age" : 6, "length_1" : 30 }
```

查询 age 的值在 7,8 范围内的数据

```
> db.c1.find({age:{$in: [7,8]}});
{ "_id" : ObjectId("4fb4af85afa87dc1bed94330"), "age" : 7, "length_1" : 30 }
{ "_id" : ObjectId("4fb4af89afa87dc1bed94331"), "age" : 8, "length_1" : 30 }
```

可以看出只显示出了 age 等于 7 或 8 的数据，其它不符合规则的数据并没有显示出来

5.8 $nin 不包含

与 sql 标准语法的用途是一样的，即要查询的数据在一系列枚举值的范围外
查询 x 的值在 2,4,6 范围外的数据
db.things.find({x:{$nin: [2,4,6]}});
举例如下:

C1 表的数据如下:

```
> db.c1.find()
{ "_id" : ObjectId("4fb4af85afa87dc1bed94330"), "age" : 7, "length_1" : 30 }
{ "_id" : ObjectId("4fb4af89afa87dc1bed94331"), "age" : 8, "length_1" : 30 }
{ "_id" : ObjectId("4fb4af8cafa87dc1bed94332"), "age" : 6, "length_1" : 30 }
```

查询 age 的值在 7,8 范围外的数据

```
> db.c1.find({age:{$nin: [7,8]}});
{ "_id" : ObjectId("4fb4af8cafa87dc1bed94332"), "age" : 6, "length_1" : 30 }
```

可以看出只显示出了 age 不等于 7 或 8 的数据，其它不符合规则的数据并没有显示出来

5.9 $size 数组元素个数

对于{name: 'David', age: 26, favorite_number: [6, 7, 9] }记录
匹配 db.users.find({favorite_number: {$size: 3}});
不匹配 db.users.find({favorite_number: {$size: 2}});
举例如下:
C1 表的数据如下:

```
> db.c1.find()
{ "_id" : ObjectId("4fb4af85afa87dc1bed94330"), "age" : 7, "length_1" : 30 }
{ "_id" : ObjectId("4fb4af89afa87dc1bed94331"), "age" : 8, "length_1" : 30 }
{ "_id" : ObjectId("4fb4af8cafa87dc1bed94332"), "age" : 6, "length_1" : 30 }
```

查询 age 的值在 7,8 范围外的数据

```
> db.c1.find({age:{$nin: [7,8]}});
{ "_id" : ObjectId("4fb4af8cafa87dc1bed94332"), "age" : 6, "length_1" : 30 }
```

可以看出只显示出了 age 不等于 7 或 8 的数据，其它不符合规则的数据并没有显示出来

5.10 正则表达式匹配

查询不匹配 name=B*带头的记录
db.users.find({name: {$not: /^B.*/}});
举例如下:
C1 表的数据如下:

```
> db.c1.find();
{ "_id" : ObjectId("4fb5faaf6d0f9d8ea3fc91a8"), "name" : "Tony", "age" : 20 }
{ "_id" : ObjectId("4fb5fab96d0f9d8ea3fc91a9"), "name" : "Joe", "age" : 10 }
```

查询 name 不以 T 开头的数据

```
> db.c1.find({name: {$not: /^T.*/}});
{ "_id" : ObjectId("4fb5fab96d0f9d8ea3fc91a9"), "name" : "Joe", "age" : 10 }
```

可以看出只显示出了 name=Tony 的数据，其它不符合规则的数据并没有显示出来

5.11 Javascript 查询和$where 查询

查询 a 大于 3 的数据，下面的查询方法殊途同归

- db.c1.find({ a : { $gt: 3 } });
- db.c1.find({ $where: "this.a > 3" });
- db.c1.find("this.a > 3");
- f = function() { return this.a > 3; } db.c1.find(f);

5.12 count 查询记录条数

count 查询记录条数
db.users.find().count();
以下返回的不是 5，而是 user 表中所有的记录数量
db.users.find().skip(10).limit(5).count();
如果要返回限制之后的记录数量，要使用 count(true)或者 count(非 0)
db.users.find().skip(10).limit(5).count(true);
举例如下:
C1 表的数据如下:

```
> db.c1.find()
{ "_id" : ObjectId("4fb5faaf6d0f9d8ea3fc91a8"), "name" : "Tony", "age" : 20 }
{ "_id" : ObjectId("4fb5fab96d0f9d8ea3fc91a9"), "name" : "Joe", "age" : 10 }
```

查询 c1 表的数据量

```
> db.c1.count()
2
```

可以看出表中共有 2 条数据

5.13 skip 限制返回记录的起点

从第 3 条记录开始，返回 5 条记录(limit 3, 5)
db.users.find().skip(3).limit(5);
举例如下:
C1 表的数据如下:

```
> db.c1.find()
{ "_id" : ObjectId("4fb5faaf6d0f9d8ea3fc91a8"), "name" : "Tony", "age" : 20 }
{ "_id" : ObjectId("4fb5fab96d0f9d8ea3fc91a9"), "name" : "Joe", "age" : 10 }
```

查询 c1 表的第 2 条数据

```
> db.c1.find().skip(1).limit(1)
{ "_id" : ObjectId("4fb5fab96d0f9d8ea3fc91a9"), "name" : "Joe", "age" : 10 }
```

可以看出表中第 2 条数据被显示了出来

5.14 sort 排序

以年龄升序 asc
db.users.find().sort({age: 1});
以年龄降序 desc
db.users.find().sort({age: -1});
C1 表的数据如下:

```
> db.c1.find()
{ "_id" : ObjectId("4fb5faaf6d0f9d8ea3fc91a8"), "name" : "Tony", "age" : 20 }
{ "_id" : ObjectId("4fb5fab96d0f9d8ea3fc91a9"), "name" : "Joe", "age" : 10 }
```

查询 c1 表按 age 升序排列

```
> db.c1.find().sort({age: 1});
{ "_id" : ObjectId("4fb5fab96d0f9d8ea3fc91a9"), "name" : "Joe", "age" : 10 }
{ "_id" : ObjectId("4fb5faaf6d0f9d8ea3fc91a8"), "name" : "Tony", "age" : 20 }
```

第 1 条是 age=10 的，而后升序排列结果集

查询 c1 表按 age 降序排列

```
> db.c1.find().sort({age: -1});
{ "_id" : ObjectId("4fb5faaf6d0f9d8ea3fc91a8"), "name" : "Tony", "age" : 20 }
{ "_id" : ObjectId("4fb5fab96d0f9d8ea3fc91a9"), "name" : "Joe", "age" : 10 }
```

第 1 条是 age=20 的，而后降序排列结果集

5.2 游标

象大多数数据库产品一样，MongoDB 也是用游标来循环处理每一条结果数据，具体语法如下：

```
> for( var c = db.t3.find(); c.hasNext(); ) {
...       printjson( c.next());
... }
{ "_id" : ObjectId("4fb8e4838b2cb86417c9423a"), "age" : 1 }
{ "_id" : ObjectId("4fb8e4878b2cb86417c9423b"), "age" : 2 }
{ "_id" : ObjectId("4fb8e4898b2cb86417c9423c"), "age" : 3 }
{ "_id" : ObjectId("4fb8e48c8b2cb86417c9423d"), "age" : 4 }
{ "_id" : ObjectId("4fb8e48e8b2cb86417c9423e"), "age" : 5 }
```

MongoDB 还有另一种方式来处理游标

```
> db.t3.find().forEach( function(u) { printjson(u); } );
{ "_id" : ObjectId("4fb8e4838b2cb86417c9423a"), "age" : 1 }
{ "_id" : ObjectId("4fb8e4878b2cb86417c9423b"), "age" : 2 }
{ "_id" : ObjectId("4fb8e4898b2cb86417c9423c"), "age" : 3 }
{ "_id" : ObjectId("4fb8e48c8b2cb86417c9423d"), "age" : 4 }
```

```
{ "_id" : ObjectId("4fb8e48e8b2cb86417c9423e"), "age" : 5 }
>
```

5.3 存储过程

MongoDB 为很多问题提供了一系列的解决方案，针对于其它数据库的特性，它仍然毫不示弱，表现的非比寻常。
MongoDB 同样支持存储过程。关于存储过程你需要知道的第一件事就是它是用 javascript 来写的。也许这会让你很奇怪，为什么它用 javascript 来写，但实际上它会让你非常满意，MongoDB 存储过程是存储在 db.system.js 表中的，我们想象一个简单的 sql 自定义函数如下：

```
function addNumbers( x , y ) {
    return x + y;
}
```

下面我们将这个 sql 自定义函数转换为 MongoDB 的存储过程:

```
> db.system.js.save({_id:"addNumbers", value:function(x, y){ return x + y; }});
```

存储过程可以被查看，修改和删除，所以我们用 find 来查看一下是否这个存储过程已经被创建上了。

```
> db.system.js.find()
{ "_id" : "addNumbers", "value" : function cf__1__f_(x, y) {
    return x + y;
} }
>
```

这样看起来还不错，下面我看来实际调用一下这个存储过程:

```
> db.eval('addNumbers(3, 4.2)');
7.2
>
```

这样的操作方法简直太简单了，也许这就是 MongoDB 的魅力所在。

db.eval()是一个比较奇怪的东西，我们可以将存储过程的逻辑直接在里面并同时调用，而无需事先声明存储过程的逻辑。

```
> db.eval( function() { return 3+3; } );
6
>
```

从上面可以看出，MongoDB 的存储过程可以方便的完成算术运算，但其它数据库产品在存储过程中可以处理数据库内部的一些事情，例如取出某张表的数据量等等操作，这些 MongoDB 能做到吗？答案是肯定的，MongoDB 可以轻而易举的做到，看下面的实例吧:

```
> db.system.js.save({_id:"get_count", value:function(){ return db.c1.count(); }});
> db.eval('get_count()')
2
```

可以看到存储过程可以很轻松的在存储过程中操作表

第六章 Capped Collection

6.1 简单介绍

capped collections 是性能出色的有着固定大小的集合，以 LRU(Least Recently Used 最近最少使用)规则和插入顺序进行 age-out(老化移出)处理，自动维护集合中对象的插入顺序，在创建时要预先指定大小。如果空间用完，新添加的对象将会取代集合中最旧的对象。

6.2 功能特点

可以插入及更新，但更新不能超出 collection 的大小，否则更新失败。不允许删除，但是可以调用 drop() 删除集合中的所有行，但是 drop 后需要显式地重建集合。在 32 位机上，一个 capped collection 的最大值约为 482.5M，64 位上只受系统文件大小的限制。

6.3 常见用处

1、 logging
MongoDB 中日志机制的首选，MongoDB 没有使用日志文件，而是把日志事件存储在数据库中。在一个没有索引的 capped collection 中插入对象的速度与在文件系统中记录日志的速度相当。
2、 cache
缓存一些对象在数据库中，比如计算出来的统计信息。这样的需要在 collection 上建立一个索引，因为使用缓存往往是读比写多。
3、 auto archiving
可以利用 capped collection 的 age-out 特性，省去了写 cron 脚本进行人工归档的工作。

6.4 推荐用法

1、 为了发挥 capped collection 的最大性能，如果写比读多，最好不要在上面建索引，否则插入速度从"log speed"降为"database speed"。

2、使用"nature ordering"可以有效地检索最近插入的元素，因为 capped collection 能够保证自然排序就是插入时的顺序，类似于 log 文件上的 tail 操作。

6.5 注意事项

1、可以在创建 capped collection 时指定 collection 中能够存放的最大文档数。但这时也要指定 size，因为总是先检查 size 后检查 maxRowNumber。可以使用 validate()查看一个 collection

已经使用了多少空间，从而决定 size 设为多大。如：
db.createCollection("mycoll", {capped:true, size:100000, max:100});
db.mycoll.validate();
max=1 时会往 collection 中存放尽量多的 documents。

2、上述的 createCollection 函数也可以用来创建一般的 collection，还有一个参数"autoIndexID"，值可以为"true"和"false"来决定是否需要在"_id"字段上自动创建索引，如：db.createCollection("mycoll", {size:10000000, autoIndexId:false})。
默认情况下对一般的 collection 是创建索引的，但不会对 capped collection 创建。

第七章 GridFS

GridFS 是一种将大型文件存储在 MongoDB 数据库中的文件规范。所有官方支持的驱动均实现了 GridFS 规范。

7.1 为什么要用 GridFS

由于 MongoDB 中 BSON 对象大小是有限制的，所以 GridFS 规范提供了一种透明的机制，可以将一个大文件分割成为多个较小的文档，这样的机制允许我们有效的保存大文件对象，特别对于那些巨大的文件，比如视频、高清图片等。

7.2 如何实现海量存储

为实现这点，该规范指定了一个将文件分块的标准。每个文件都将在文件集合对象中保存一个元数据对象，一个或多个 chunk 块对象可被组合保存在一个 chunk 块集合中。大多数情况下，你无需了解此规范中细节，而可将注意力放在各个语言版本的驱动中有关 GridFS API 的部分或是如何使用 mongofiles 工具上。

7.3 语言支持

GridFS 对 Java, Perl, PHP, Python, Ruby 等程序语言均支持，且提供了良好的 API 接口。

7.4 简单介绍

GridFS 使用两个表来存储数据：

- files 包含元数据对象
- chunks 包含其他一些相关信息的二进制块

为了使多个 GridFS 命名为一个单一的数据库，文件和块都有一个前缀，默认情况下，前缀是 fs，所以任何默认的 GridFS 存储将包括命名空间 fs.files 和 fs.chunks。各种第三方语言的驱动有权限改变这个前缀，所以你可以尝试设置另一个 GridFS 命名空间用于存储照片，它的具体位置为:photos.files 和 photos.chunks。下面我们看一下实际的例子吧。

7.5 命令行工具

mongofiles 是从命令行操作 GridFS 的一种工具，例如我们将”mongosniff”这个文件存到库里面，具体用法如下：

```
[root@localhost bin]# ./mongofiles put testfile
connected to: 127.0.0.1
added file: { _id: ObjectId('4fc60175c714c5d960fff76a'), filename: "testfile", chunkSize: 262144, uploadDate: new Date(1338376565745), md5: "8addbeb77789ae6b2cb75deee30faf1a", length: 16 }
done!
```

下面我们查一下看库里有哪些 GridFS 文件，在”mongofiles”后加一个参数”list”即可

```
[root@localhost bin]# ./mongofiles list
connected to: 127.0.0.1
testfile        16
```

接下来我们进库里看一下是否有新的东西

```
[root@localhost bin]# ./mongo
MongoDB shell version: 1.8.1
connecting to: test
> show collections
fs.chunks                --上文提到的 fs.chunks
fs.files                 --上文提到的 fs.files
system.indexes
system.js
>
```

我们继续查看 fs.files 中的内容

```
> db.fs.files.find()
{ "_id" : ObjectId("4fc60175c714c5d960fff76a"), "filename" : "testfile", "chunkSize" : 262144, "uploadDate" : ISODate("2012-05-30T11:16:05.745Z"), "md5" : "8addbeb77789ae6b2cb75deee30faf1a", "length" : 16 }
```

字段说明:

- Filename: 存储的文件名
- chunkSize: chunks 分块的大小
- uploadDate: 入库时间
- md5: 此文件的 md5 码
- length: 文件大小，单位”字节”

看来 fs.files 中存储的是一些基础的元数据信息

我们继续查看 fs.chunks 中的内容

```
> db.fs.chunks.find()
{ "_id" : ObjectId("4fc60175cf1154905d949336"), "files_id" :
```

```
ObjectId("4fc60175c714c5d960fff76a"), "n" : 0, "data" : BinData(0,"SGVyZSBpcyBCZWlqaW5nCg==") }
```

其中比较重要的字段是”n”，它代表的是 chunks 的序号，此序号从 0 开始，看来 fs.chunks 中存储的是一些实际的内容数据信息

我们即然能将此文件存进去，我们就应该有办法将其取出来，下面看一下实例：

```
[root@localhost bin]# rm testfile
rm：是否删除 一般文件 “testfile”? y                --先删文件
[root@localhost bin]# ./mongofiles get testfile      --将其从库里取出来
connected to: 127.0.0.1
done write to: testfile
[root@localhost bin]# md5sum   testfile                  --校验 md5，结果跟库里相同
8addbeb77789ae6b2cb75deee30faf1a   testfile
[root@localhost bin]#
```

7.6 索引

db.fs.chunks.ensureIndex({files_id:1, n:1}, {unique: true});

这样，一个块就可以利用它的 files_id 和 n 的值进行检索。注意，GridFS 仍然可以用 findOne 得到第一个块，如下：
db.fs.chunks.findOne({files_id: myFileID, n: 0});

第八章 MapReduce

MongoDB 的 MapReduce 相当于 Mysql 中的"group by"，所以在 MongoDB 上使用 Map/Reduce 进行并行"统计"很容易。

使用 MapReduce 要实现两个函数 Map 函数和 Reduce 函数，Map 函数调用 emit(key, value)，遍历 collection 中所有的记录，将 key 与 value 传递给 Reduce 函数进行处理。Map 函数和 Reduce 函数可以使用 JavaScript 来实现，可以通过 db.runCommand 或 mapReduce 命令来执行一个 MapReduce 的操作：

```
db.runCommand(
 { mapreduce : <collection>,
   map : <mapfunction>,
   reduce : <reducefunction>
   [, query : <query filter object>]
   [, sort : <sorts the input objects using this key. Useful for optimization, like sorting by the emit key for fewer reduces>]
   [, limit : <number of objects to return from collection>]
   [, out : <see output options below>]
```

```
    [, keeptemp: <true|false>]
    [, finalize : <finalizefunction>]
    [, scope : <object where fields go into javascript global scope >]
    [, verbose : true]
 }
);
```

参数说明:

- mapreduce: 要操作的目标集合。
- map: 映射函数 (生成键值对序列，作为 reduce 函数参数)。
- reduce: 统计函数。
- query: 目标记录过滤。
- sort: 目标记录排序。
- limit: 限制目标记录数量。
- out: 统计结果存放集合 (不指定则使用临时集合，在客户端断开后自动删除)。
- keeptemp: 是否保留临时集合。
- finalize: 最终处理函数 (对 reduce 返回结果进行最终整理后存入结果集合)。
- scope: 向 map、reduce、finalize 导入外部变量。
- verbose: 显示详细的时间统计信息。

下面我们先准备一些数据:

```
> db.students.insert({classid:1, age:14, name:'Tom'})
> db.students.insert({classid:1, age:12, name:'Jacky'})
> db.students.insert({classid:2, age:16, name:'Lily'})
> db.students.insert({classid:2, age:9, name:'Tony'})
> db.students.insert({classid:2, age:19, name:'Harry'})
> db.students.insert({classid:2, age:13, name:'Vincent'})
> db.students.insert({classid:1, age:14, name:'Bill'})
> db.students.insert({classid:2, age:17, name:'Bruce'})
>
```

接下来，我们将演示如何统计 1 班和 2 班的学生数量

8.1 Map

Map 函数必须调用 emit(key, value) 返回键值对，使用 this 访问当前待处理的 Document。

```
> m = function() { emit(this.classid, 1) }
function () {
    emit(this.classid, 1);
}
>
```

value 可以使用 JSON Object 传递 (支持多个属性值)。例如:

emit(this.classid, {count:1})

8.2 Reduce

Reduce 函数接收的参数类似 Group 效果，将 Map 返回的键值序列组合成 { key, [value1, value2, value3, value...] } 传递给 reduce。

```
> r = function(key, values) {
... var x = 0;
... values.forEach(function(v) { x += v });
... return x;
... }
function (key, values) {
        var x = 0;
        values.forEach(function (v) {x += v;});
        return x;
}
>
```

Reduce 函数对这些 values 进行 "统计" 操作，返回结果可以使用 JSON Object。

8.3 Result

```
> res = db.runCommand({
... mapreduce:"students",
... map:m,
... reduce:r,
... out:"students_res"
... });
{
            "result" : "students_res",
            "timeMillis" : 1587,
            "counts" : {
                        "input" : 8,
                        "emit" : 8,
                        "output" : 2
            },
            "ok" : 1
}
> db.students_res.find()
{ "_id" : 1, "value" : 3 }
{ "_id" : 2, "value" : 5 }
>
```

mapReduce() 将结果存储在 "students_res" 表中。

8.4 Finalize

利用 finalize() 我们可以对 reduce() 的结果做进一步处理。

```
> f = function(key, value) { return {classid:key, count:value}; }
function (key, value) {
	return {classid:key, count:value};
}
>
```

我们再重新计算一次，看看返回的结果:

```
> res = db.runCommand({
... mapreduce:"students",
... map:m,
... reduce:r,
... out:"students_res",
... finalize:f
... });
{
		"result" : "students_res",
		"timeMillis" : 804,
		"counts" : {
			"input" : 8,
			"emit" : 8,
			"output" : 2
		},
		"ok" : 1
}
> db.students_res.find()
{ "_id" : 1, "value" : { "classid" : 1, "count" : 3 } }
{ "_id" : 2, "value" : { "classid" : 2, "count" : 5 } }
>
```

列名变与“classid”和”count”了，这样的列表更容易理解。

8.5 Options

我们还可以添加更多的控制细节。

```
> res = db.runCommand({
... mapreduce:"students",
... map:m,
... reduce:r,
... out:"students_res",
... finalize:f,
... query:{age:{$lt:10}}
... });
```

```
{
        "result" : "students_res",
        "timeMillis" : 358,
        "counts" : {
                "input" : 1,
                "emit" : 1,
                "output" : 1
        },
        "ok" : 1
}
> db.students_res.find();
{ "_id" : 2, "value" : { "classid" : 2, "count" : 1 } }
>
```

可以看到先进行了过滤，只取 age<10 的数据，然后再进行统计，所以就没有 1 班的统计数据了。

第三部分 管理篇

第九章 数据导出 mongoexport

作为 DBA，经常会碰到导入导出数据的需求，下面就介绍实用工具 mongoexport 和 mongoimport 的使用方法，望你会有所收获。

假设库里有一张 user 表，里面有 2 条记录，我们要将它导出

```
> use my_mongodb
switched to db my_mongodb
> db.user.find();
{ "_id" : ObjectId("4f81a4a1779282ca68fd8a5a"), "uid" : 2, "username" : "Jerry", "age" : 100 }
{ "_id" : ObjectId("4f844d1847d25a9ce5f120c4"), "uid" : 1, "username" : "Tom", "age" : 25 }
>
```

9.1 常用导出方法

```
[root@localhost bin]# ./mongoexport -d my_mongodb -c user -o user.dat
connected to: 127.0.0.1
exported 2 records
[root@localhost bin]# cat user.dat
{ "_id" : { "$oid" : "4f81a4a1779282ca68fd8a5a" }, "uid" : 2, "username" : "Jerry", "age" : 100 }
{ "_id" : { "$oid" : "4f844d1847d25a9ce5f120c4" }, "uid" : 1, "username" : "Tom", "age" : 25 }
[root@localhost bin]#
```

参数说明:

- -d 指明使用的库，本例中为” my_mongodb”

- -c 指明要导出的表，本例中为”user”
- -o 指明要导出的文件名，本例中为”user.dat”

从上面可以看到导出的方式使用的是 JSON 的样式

9.2 导出 CSV 格式的文件

```
[root@localhost bin]# ./mongoexport -d my_mongodb -c user --csv -f uid,username,age -o user_csv.dat
connected to: 127.0.0.1
exported 2 records
[root@localhost bin]# cat user_csv.dat
uid,username,age
2,"Jerry",100
1,"Tom",25
[root@localhost bin]#
```

参数说明:

- -csv 指要要导出为 csv 格式
- -f 指明需要导出哪些例

更详细的用法可以 mongoexport –help 来查看

第十章 数据导入 mongoimport

在上例中我们讨论的是导出工具的使用，那么本节将讨论如何向表中导入数据

10.1 导入 JSON 数据

我们先将表 user 删除掉，以便演示效果

```
> db.user.drop();
true
> show collections;
system.indexes
>
```

然后导入数据

```
 [root@localhost bin]# ./mongoimport -d my_mongodb -c user user.dat
connected to: 127.0.0.1
imported 2 objects
[root@localhost bin]#
```

可以看到导入数据的时候会隐式创建表结构

10.2 导入 CSV 数据

我们先将表 user 删除掉，以便演示效果

```
> db.user.drop();
true
> show collections;
system.indexes
>
```

然后导入数据

```
[root@localhost bin]# ./mongoimport -d my_mongodb -c user --type csv --headerline --file user_csv.dat
connected to: 127.0.0.1
imported 3 objects
[root@localhost bin]#
```

参数说明:

- -type 指明要导入的文件格式
- -headerline 批明不导入第一行，因为第一行是列名
- -file 指明要导入的文件路径

注意:
CSV 格式良好，主流数据库都支持导出为 CSV 的格式，所以这种格式非常利于异构数据迁移

第十一章 数据备份 mongodump

可以用 mongodump 来做 MongoDB 的库或表级别的备份，下面举例说明:
备份 my_mongodb 数据库

```
[root@localhost bin]# ./mongodump -d my_mongodb
connected to: 127.0.0.1
DATABASE: my_mongodb        to        dump/my_mongodb
        my_mongodb.system.indexes to dump/my_mongodb/system.indexes.bson
                1 objects
        my_mongodb.user to dump/my_mongodb/user.bson
                2 objects
[root@localhost bin]# ll
总计 67648
-rwxr-xr-x 1 root root 7508756 2011-04-06 bsondump
drwxr-xr-x 3 root root       4096 04-10 23:54 dump
-rwxr-xr-x 1 root root 2978016 2011-04-06 mongo
```

此时会在当前目录下创建一个 dump 目录，用于存放备份出来的文件
也可以指定备份存放的目录，

```
[root@localhost bin]# ./mongodump -d my_mongodb -o my_mongodb_dump
connected to: 127.0.0.1
DATABASE: my_mongodb        to        my_mongodb_dump/my_mongodb
            my_mongodb.system.indexes                                                    to
my_mongodb_dump/my_mongodb/system.indexes.bson
                        1 objects
            my_mongodb.user to my_mongodb_dump/my_mongodb/user.bson
                        2 objects
[root@localhost bin]#
```

这个例子中将备份的文件存在了当前目录下的 my_mongodb_dump 目录下

第十二章 数据恢复 mongorestore

由于刚刚已经做了备份，所以我们先将库 my_mongodb 删除掉

```
> use my_mongodb
switched to db my_mongodb
> db.dropDatabase()
{ "dropped" : "my_mongodb", "ok" : 1 }
> show dbs
admin      (empty)
local      (empty)
test       (empty)
>
```

接下来我们进行数据库恢复

```
[root@localhost bin]# ./mongorestore -d my_mongodb my_mongodb_dump/*
connected to: 127.0.0.1
Wed Apr 11 00:03:03 my_mongodb_dump/my_mongodb/user.bson
Wed Apr 11 00:03:03          going into namespace [my_mongodb.user]
Wed Apr 11 00:03:03          2 objects found
Wed Apr 11 00:03:03 my_mongodb_dump/my_mongodb/system.indexes.bson
Wed Apr 11 00:03:03          going into namespace [my_mongodb.system.indexes]
Wed Apr 11 00:03:03 { name: "_id_", ns: "my_mongodb.user", key: { _id: 1 }, v: 0 }
Wed Apr 11 00:03:03          1 objects found
[root@localhost bin]#
```

经验证数据库又回来了，其实要是想恢复库，也大可不必先删除 my_mongodb 库，只要指明 –drop 参数，就可以在恢复的时候先删除表然后再向表中插入数据的

第十三章 访问控制

官方手册中启动 MongoDB 服务时没有任何参数，一旦客户端连接后可以对数据库任意操

作，而且可以远程访问数据库，所以推荐开发阶段可以不设置任何参数，但对于生产环境还是要仔细考虑一下安全方面的因素，而提高 MongoDB 数据库安全有几个方面：

- 绑定 IP 内网地址访问 MongoDB 服务
- 设置监听端口
- 使用用户名和口令登录

13.1 绑定 IP 内网地址访问 MongoDB 服务

MongoDB 可以限制只允许某一特定 IP 来访问，只要在启动时加一个参数 bind_ip 即可，如下:

服务端限制只有 192.168.1.103 这个 IP 可以访问 MongoDB 服务

```
[root@localhost bin]# ./mongod --bind_ip 192.168.1.103
```

客户端访问时需要明确指定服务端的 IP，否则会报错:

```
[root@localhost bin]# ./mongo 192.168.1.102
MongoDB shell version: 1.8.1
connecting to: 192.168.1.103/test
>
```

13.2 设置监听端口

官方默认的监听端口是 27017，为了安全起见，一般都会修改这个监听端口，避免恶意的连接尝试，具体如下：

将服务端监听端口修改为 28018

```
[root@localhost bin]# ./mongod --bind_ip 192.168.1.103 --port 28018
```

端户访问时不指定端口，会连接到默认端口 27017，对于本例会报错

```
[root@localhost bin]# ./mongo 192.168.1.102
MongoDB shell version: 1.8.1
connecting to: 192.168.1.102/test
Sun Apr 15 15:55:51 Error: couldn't connect to server 192.168.1.102 shell/mongo.js:81
exception: connect failed
```

所以当服务端指定了端口后，客户端必须要明确指定端口才可以正常访问

```
[root@localhost bin]# ./mongo 192.168.1.102:28018
MongoDB shell version: 1.8.1
connecting to: 192.168.1.102:28018/test
>
```

13.3 使用用户名和口令登录

MongoDB 默认的启动是不验证用户名和密码的,启动 MongoDB 后,可以直接用 MongoDB 连接

上来,对所有的库具有 root 权限。所以启动的时候指定参数,可以阻止客户端的访问和连接。

先启用系统的登录验证模块，只需在启动时指定 auth 参数即可，如:

```
[root@localhost bin]# ./mongod --auth
```

本地客户端连接一下看看效果;

```
[root@localhost bin]# ./mongo
MongoDB shell version: 1.8.1
connecting to: test
> show collections;
>
```

很奇怪，为什么我们启用了登录验证模块，但我们登录时没有指定用户，为什么还可以登录呢？在最初始的时候 MongoDB 都默认有一个 admin 数据库（默认是空的），
而 admin.system.users 中将会保存比在其它数据库中设置的用户权限更大的用户信息。
注意：当 admin.system.users 中没有添加任何用户时，即使 MongoDB 启动时添加了 --auth 参数，如果在除 admin 数据库中添加了用户，此时不进行任何认证依然可以使用任何操作，直到知道你在 admin.system.users 中添加了一个用户。

1、建立系统 root 用户

在 admin 库中新添一个用户 root:

```
[root@localhost bin]# ./mongo
MongoDB shell version: 1.8.1
connecting to: test
> db.addUser("root","111")
{
        "user" : "root",
        "readOnly" : false,
        "pwd" : "e54950178e2fa777b1d174e9b106b6ab"
}
> db.auth("root","111")
1
>
```

本地客户端连接，但不指定用户，结果如下:

```
[root@localhost bin]# ./mongo
MongoDB shell version: 1.8.1
connecting to: test
> show collections;
Sun Apr 15 16:36:52 uncaught exception: error: {
        "$err" : "unauthorized db:test lock type:-1 client:127.0.0.1",
        "code" : 10057
}
>
```

连上 test 库了，但进一步操作时有异常，看来 MongoDB 允许未授权连接，不能进行任何

本地客户端连接，指定用户，结果如下:

```
[root@localhost bin]# ./mongo -u root -p
MongoDB shell version: 1.8.1
Enter password:
connecting to: test
> show collections;
system.indexes
system.users
>
```

看来指定了用户名之后，访问数据库才是正常

2、建立指定权限用户

MongoDB 也支持为某个特定的数据库来设置用户，如我们为 test 库设一个只读的用户 user_reader:

```
[root@localhost bin]# ./mongo -u root -p
MongoDB shell version: 1.8.1
Enter password:
connecting to: test
> show collections;
system.indexes
system.users
> use test
switched to db test
> db.addUser("user_reader", "user_pwd", true)
{
        "user" : "user_reader",
        "readOnly" : true,
        "pwd" : "0809760bb61ee027199e513c5ecdedc6"
}
>
```

客户端用此用户来访问:

```
[root@localhost bin]# ./mongo -u user_reader -p
MongoDB shell version: 1.8.1
Enter password:
connecting to: test
> show collections;
system.indexes
system.users
>
```

第十四章 命令行操作

MongoDB shell 不仅仅是一个交互式的 shell，它也支持执行指定 javascript 文件，也支持执行指定的命令片断。

有了这个特性，就可以将 MongoDB 与 linux shell 完美结合，完成大部分的日常管理和维护工作。

14.1 通过 eval 参数执行指定语句

例如，需要查询 test 库的 t1 表中的记录数有多少，常用方法如下:

```
[root@localhost bin]# ./mongo test
MongoDB shell version: 1.8.1
connecting to: test
> db.t1.count()
7
>
```

通过命令行 eval 参数直接执行语句:

```
[root@localhost bin]# ./mongo test --eval "printjson(db.t1.count())"
MongoDB shell version: 1.8.1
connecting to: test
7
```

14.2 执行指定文件中的内容

如果涉及到很多的操作后，才能得到结果，那么用 eval 的方式来做的话是不可能完成的，那么更灵活的执行指定文件的方式就派上用场了。例如我们仍然要查看 test 库 t1 表中的记录数：

t1_count.js 就是我们要执行的文件，里面的内容如下

```
[root@localhost bin]# cat t1_count.js
var totalcount = db.t1.count();
printjson('Total count of t1 is :    ' + totalcount);
printjson('----------------------');
```

下面我们将执行这个文件

```
[root@localhost bin]# ./mongo t1_count.js
MongoDB shell version: 1.8.1
connecting to: test
"Total count of t1 is :    7"
"----------------------"
```

```
[root@localhost bin]#
```

大家可以看到最终得到 t1 表的记录数 7，那么一些不必要的说明性文字我们要是不希望出现该怎么办呢?

```
[root@localhost bin]# ./mongo --quiet    t1_count.js
"Total count of t1 is :    7"
"----------------------"
[root@localhost bin]#
```

通过指定 quiet 参数，即可以将一些登录信息屏蔽掉，这样可以让结果更清晰。

第十五章 进程控制

DBA 经常要解决系统的一些查询性能问题，此时一般的操作习惯是先查看有哪些进程，然后将异常的进程杀掉，那么 MongoDB 是怎么样处理的呢?

15.1 查看活动进程

查看活动进程，便于了解系统正在做什么，以便做下一步判断

```
> db.currentOp();
> // 等同于: db.$cmd.sys.inprog.findOne()
{ inprog: [ { "opid" : 18 , "op" : "query" , "ns" : "mydb.votes" ,
                    "query" : "{ score : 1.0 }" , "inLock" : 1 }
            ]
}
```

字段说明:

- Opid: 操作进程号
- Op: 操作类型(查询，更新等)
- Ns: 命名空间, 指操作的是哪个对象
- Query: 如果操作类型是查询的话，这里将显示具体的查询内容
- lockType: 锁的类型，指明是读锁还是写锁

15.2 结束进程

如果某个异常是由于某个进程产生的，那么一般 DBA 都会毫不留情的杀掉这个罪魁祸首的进程，下面将是这操作

```
> db.killOp(1234/*opid*/)
> // 等同于: db.$cmd.sys.killop.findOne({op:1234})
```

```
注意:
不要 kill 内部发起的操作，比如说 replica set 发起的 sync 操作等
```

第四部分 性能篇

第十六章 索引

MongoDB 提供了多样性的索引支持，索引信息被保存在 system.indexes 中，且默认总是为_id 创建索引，它的索引使用基本和 MySQL 等关系型数据库一样。其实可以这样说说，索引是凌驾于数据存储系统之上的另一层系统，所以各种结构迥异的存储都有相同或相似的索引实现及使用接口并不足为奇。

16.1 基础索引

在字段 age 上创建索引，1(升序);-1(降序)

```
> db.t3.ensureIndex({age:1})
> db.t3.getIndexes();
[
        {
                "name" : "_id_",
                "ns" : "test.t3",
                "key" : {
                        "_id" : 1
                },
                "v" : 0
        },
        {
                "_id" : ObjectId("4fb906da0be632163d0839fe"),
                "ns" : "test.t3",
                "key" : {
                        "age" : 1
                },
                "name" : "age_1",
                "v" : 0
        }
]
>
```

上例显示出来的一共有 2 个索引，其中_id 是创建表的时候自动创建的索引，此索引是不能够删除的。

当系统已有大量数据时，创建索引就是个非常耗时的活，我们可以在后台执行，只需指定“backgroud:true”即可。

```
> db.t3.ensureIndex({age:1}, {backgroud:true})
```

16.2 文档索引

索引可以任何类型的字段，甚至文档

```
db.factories.insert( { name: "wwl", addr: { city: "Beijing", state: "BJ" } } );

//在 addr 列上创建索引
db.factories.ensureIndex( { addr : 1 } );

//下面这个查询将会用到我们刚刚建立的索引
db.factories.find( { addr: { city: "Beijing", state: "BJ" } } );

//但是下面这个查询将不会用到索引，因为查询的顺序跟索引建立的顺序不一样
db.factories.find( { addr: { state: "BJ" , city: "Beijing"} } );
```

16.3 组合索引

跟其它数据库产品一样，MongoDB 也是有组合索引的，下面我们将在 addr.city 和 addr.state 上建立组合索引。当创建组合索引时，字段后面的 1 表示升序，-1 表示降序，是用 1 还是用-1 主要是跟排序的时候或指定范围内查询 的时候有关的。

```
db.factories.ensureIndex( { "addr.city" : 1, "addr.state" : 1 } );

// 下面的查询都用到了这个索引
db.factories.find( { "addr.city" : "Beijing", "addr.state" : "BJ" } );
db.factories.find( { "addr.city" : "Beijing" } );
db.factories.find().sort( { "addr.city" : 1, "addr.state" : 1 } );
db.factories.find().sort( { "addr.city" : 1 } )
```

16.4 唯一索引

只需在 ensureIndex 命令中指定”unique:true”即可创建唯一索引。例如，往表 t4 中插入 2 条记录

```
db.t4.insert({firstname: "wang", lastname: "wenlong"});
db.t4.insert({firstname: "wang", lastname: "wenlong"});
```

在 t4 表中建立唯一索引

```
> db.t4.ensureIndex({firstname: 1, lastname: 1}, {unique: true});
E11000 duplicate key error index: test.t4.$firstname_1_lastname_1  dup key: { : "wang", : "wenlong" }
```

可以看到，当建唯一索引时，系统报了“表里有重复值”的错，具体原因就是因为表中有 2 条一模一模的数据，所以建立不了唯一索引。

16.5 强制使用索引

hint 命令可以强制使用某个索引。

```
> db.t5.insert({name: "wangwenlong",age: 20})
> db.t5.ensureIndex({name:1, age:1})
> db.t5.find({age:{$lt:30}}).explain()
{
        "cursor" : "BasicCursor",
        "nscanned" : 1,
        "nscannedObjects" : 1,
        "n" : 1,
        "millis" : 0,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "isMultiKey" : false,
        "indexOnly" : false,
        "indexBounds" : {                    --并没有用到索引

        }
}
> db.t5.find({age:{$lt:30}}).hint({name:1, age:1}).explain()          --强制使用索引
{
        "cursor" : "BtreeCursor name_1_age_1",
        "nscanned" : 1,
        "nscannedObjects" : 1,
        "n" : 1,
        "millis" : 1,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "isMultiKey" : false,
        "indexOnly" : false,
        "indexBounds" : {                    --被强制使用索引了
                "name" : [
                        [
                                {
                                        "$minElement" : 1
                                },
                                {
                                        "$maxElement" : 1
                                }
                        ]
                ],
                "age" : [
```

```
                        [
                                -1.7976931348623157e+308,
                                30
                        ]
                ]
        }
}
>
```

16.6 删除索引

删除索引分为删除某张表的所有索引和删除某张表的某个索引，具体如下：

```
//删除 t3 表中的所有索引
db.t3.dropIndexes()

//删除 t4 表中的 firstname 索引
db.t4.dropIndex({firstname: 1})
```

第十七章 explain 执行计划

MongoDB 提供了一个 explain 命令让我们获知系统如何处理查询请求。利用 explain 命令，我们可以很好地观察系统如何使用索引来加快检索，同时可以针对性优化索引。

```
> db.t5.ensureIndex({name:1})
> db.t5.ensureIndex({age:1})
> db.t5.find({age:{$gt:45}}, {name:1}).explain()
{
        "cursor" : "BtreeCursor age_1",
        "nscanned" : 0,
        "nscannedObjects" : 0,
        "n" : 0,
        "millis" : 0,
        "nYields" : 0,
        "nChunkSkips" : 0,
        "isMultiKey" : false,
        "indexOnly" : false,
        "indexBounds" : {
                "age" : [
                        [
                                45,
                                1.7976931348623157e+308
                        ]
                ]
```

```
        }
}
```

字段说明:

- cursor: 返回游标类型(BasicCursor 或 BtreeCursor)
- nscanned: 被扫描的文档数量
- n: 返回的文档数量
- millis: 耗时(毫秒)
- indexBounds: 所使用的索引

第十八章 优化器 profile

在 MySQL 中，慢查询日志是经常作为我们优化数据库的依据，那在 MongoDB 中是否有类似的功能呢?答案是肯定的，那就是 MongoDB Database Profiler。所以 MongoDB 不仅有，而且还有一些比 MySQL 的 Slow Query Log 更详细的信息。

18.1 开启 Profiling 功能

有两种方式可以控制 Profiling 的开关和级别，第一种是直接在启动参数里直接进行设置。启动 MongoDB 时加上–profile=级别 即可。
也可以在客户端调用 db.setProfilingLevel(级别) 命令来实时配置，Profiler 信息保存在 system.profile 中。我们可以通过 db.getProfilingLevel()命令来获取当前的 Profile 级别，类似如下操作

```
> db.setProfilingLevel(2);
{ "was" : 0, "slowms" : 100, "ok" : 1 }
```

上面 profile 的级别可以取 0，1，2 三个值，他们表示的意义如下：

- 0 – 不开启
- 1 – 记录慢命令 (默认为>100ms)
- 2 – 记录所有命令

Profile 记录在级别 1 时会记录慢命令，那么这个慢的定义是什么?上面我们说到其默认为 100ms，当然有默认就有设置，其设置方法和级别一样有两种，一种是通过添加–slowms 启动参数配置。第二种是调用 db.setProfilingLevel 时加上第二个参数：

```
db.setProfilingLevel( level , slowms )
db.setProfilingLevel( 1 , 10 );
```

18.2 查询 Profiling 记录

与 MySQL 的慢查询日志不同，MongoDB Profile 记录是直接存在系统 db 里的，记录位置 system.profile ，所以，我们只要查询这个 Collection 的记录就可以获取到我们的 Profile 记录了。列出执行时间长于某一限度(5ms)的 Profile 记录：

```
db.system.profile.find( { millis : { $gt : 5 } } )
```

查看最新的 Profile 记录：

db.system.profile.find().sort({$natural:-1}).limit(1)

```
> db.system.profile.find().sort({$natural:-1}).limit(1)
{ "ts" : ISODate("2012-05-20T16:50:36.321Z"), "info" : "query test.system.profile reslen:1219 nscanned:8 \nquery: { query: {}, orderby: { $natural: -1.0 } } nreturned:8 bytes:1203", "millis" : 0 }
>
```

字段说明:

- ts： 该命令在何时执行
- info: 本命令的详细信息
- reslen: 返回结果集的大小
- nscanned: 本次查询扫描的记录数
- nreturned: 本次查询实际返回的结果集
- millis: 该命令执行耗时，以毫秒记

MongoDB Shell 还提供了一个比较简洁的命令 show profile，可列出最近 5 条执行时间超过 1ms 的 Profile 记录。

第十九章 性能优化

如果 nscanned(扫描的记录数)远大于 nreturned(返回结果的记录数)的话，那么我们就要考虑通过加索引来优化记录定位了。

reslen 如果过大，那么说明我们返回的结果集太大了，这时请查看 find 函数的第二个参数是否只写上了你需要的属性名。

对于创建索引的建议是：如果很少读，那么尽量不要添加索引，因为索引越多，写操作会越慢。如果读量很大，那么创建索引还是比较划算的。

假设我们按照时间戳查询最近发表的 10 篇博客文章：

```
articles = db.posts.find().sort({ts:-1});

for (var i=0; i< 10; i++) {
    print(articles[i].getSummary());
}
```

19.1 优化方案 1: 创建索引

在查询条件的字段上，或者排序条件的字段上创建索引，可以显著提高执行效率：

```
db.posts.ensureIndex({ts:1});
```

19.2 优化方案 2: 限定返回结果条数

使用 limit()限定返回结果集的大小，可以减少 database server 的资源消耗，可以减少网络传

输数据量。

```
articles = db.posts.find().sort({ts:-1}).limit(10);
```

19.3 优化方案 3: 只查询使用到的字段，而不查询所有字段

在本例中，博客日志记录内容可能非常大，而且还包括了评论内容（作为 embeded 文档）。所以只查询使用的字段，比查询所有字段效率更高：

```
articles = db.posts.find({}, {ts:1,title:1,author:1,abstract:1}).sort({ts:-1}).limit(10);
```

注意：如果只查询部分字段的话，不能用返回的对象直接更新数据库。下面的代码是错误的：

```
a_post = db.posts.findOne({}, Post.summaryFields);
a_post.x = 3;
db.posts.save(a_post);
```

19.4 优化方案 4: 采用 capped collection

capped Collections 比普通 Collections 的读写效率高。Capped Collections 是高效率的 Collection 类型，它有如下特点：

1、 固定大小；Capped Collections 必须事先创建，并设置大小：

```
db.createCollection("mycoll", {capped:true, size:100000})
```

2、 Capped Collections 可以 insert 和 update 操作；不能 delete 操作。只能用 drop（）方法删除整个 Collection。

3、 默认基于 Insert 的次序排序的。如果查询时没有排序，则总是按照 insert 的顺序返回。

4、 FIFO。如果超过了 Collection 的限定大小，则用 FIFO 算法，新记录将替代最先 insert 的记录。

19.5 优化方案 5: 采用 Server Side Code Execution

Server-Side Processing 类似于 SQL 数据库的存储过程，使用 Server-Side Processing 可以减小网络通讯的开销。

19.6 优化方案 6: Hint

一般情况下 MongoDB query optimizer 都工作良好，但有些情况下使用 hint()可以提高操作效率。Hint 可以强制要求查询操作使用某个索引。例如，如果要查询多个字段的值，如果在其中一个字段上有索引，可以使用 hint：

```
db.collection.find({user:u, foo:d}).hint({user:1});
```

19.7 优化方案 7: 采用 Profiling

Profiling 功能肯定是会影响效率的，但是不太严重，原因是他使用的是 system.profile 来记录，而 system.profile 是一个 capped collection 这种 collection 在操作上有一些限制和特点，

但是效率更高。

第二十章 性能监控

20.1 mongosniff

此工具可以从底层监控到底有哪些命令发送给了 MongoDB 去执行，从中就可以进行分析：
以 root 身份执行：

```
./mongosniff --source NET lo
```

然后其会监控位到本地以 localhost 监听默认 27017 端口的 MongoDB 的所有包请求，如执行”show dbs” 操作

```
[root@localhost bin]# ./mongo
MongoDB shell version: 1.8.1
connecting to: test
> show dbs
admin      0.0625GB
foo        0.0625GB
local      (empty)
test       0.0625GB
>
```

那么你可以看到如下输出。

```
[root@localhost bin]# ./mongosniff --source NET lo
sniffing... 27017
127.0.0.1:38500   -->> 127.0.0.1:27017 admin.$cmd   60 bytes   id:537ebe0f 1400815119
            query: { whatsmyuri: 1 }   ntoreturn: 1 ntoskip: 0
127.0.0.1:27017   <<--   127.0.0.1:38500   78 bytes   id:531c3855   1394358357 - 1400815119
            reply n:1 cursorId: 0
            { you: "127.0.0.1:38500", ok: 1.0 }
127.0.0.1:38500   -->> 127.0.0.1:27017 admin.$cmd   80 bytes   id:537ebe10 1400815120
            query: { replSetGetStatus: 1, forShell: 1 }   ntoreturn: 1 ntoskip: 0
127.0.0.1:27017   <<--   127.0.0.1:38500   92 bytes   id:531c3856   1394358358 - 1400815120
            reply n:1 cursorId: 0
            { errmsg: "not running with --replSet", ok: 0.0 }
127.0.0.1:38500   -->> 127.0.0.1:27017 admin.$cmd   67 bytes   id:537ebe11 1400815121
            query: { listDatabases: 1.0 }   ntoreturn: -1 ntoskip: 0
127.0.0.1:27017   <<--   127.0.0.1:38500   293 bytes   id:531c3857 1394358359 - 1400815121
            reply n:1 cursorId: 0
            { databases: [ { name: "foo", sizeOnDisk: 67108864.0, empty: false }, { name: "test",
sizeOnDisk: 67108864.0, empty: false }, { name: "admin", sizeOnDisk: 67108864.0, empty: false },
{ name: "local", sizeOnDisk: 1.0, empty: true } ], totalSize: 201326592.0, ok: 1.0 }
127.0.0.1:38500   -->> 127.0.0.1:27017 admin.$cmd   80 bytes   id:537ebe12 1400815122
```

```
        query: { replSetGetStatus: 1, forShell: 1 }    ntoreturn: 1 ntoskip: 0
127.0.0.1:27017   <<--   127.0.0.1:38500     92 bytes    id:531c3858    1394358360 - 1400815122
        reply n:1 cursorId: 0
        { errmsg: "not running with --replSet", ok: 0.0 }
```

如果将这些输出到一个日志文件中，那么就可以保留下所有数据库操作的历史记录，对于后期的性能分析和安全审计等工作将是一个巨大的贡献。

20.2 Mongostat

此工具可以快速的查看某组运行中的 MongoDB 实例的统计信息，用法如下：

[root@localhost bin]# ./mongostat

下面是执行结果(部分):

```
[root@localhost bin]# ./mongostat
insert   query update delete ..... locked % idx miss % qr|qw     ar|aw    conn        time
    *0      *0      *0      *0 .....       0                0     0|0       1|0       4 01:19:15
    *0      *0      *0      *0 .....       0                0     0|0       1|0       4 01:19:16
    *0      *0      *0      *0 .....       0                0     0|0       1|0       4 01:19:17
```

字段说明:

- insert: 每秒插入量
- query: 每秒查询量
- update: 每秒更新量
- delete: 每秒删除量
- locked: 锁定量
- qr | qw: 客户端查询排队长度(读|写)
- ar | aw: 活跃客户端量(读|写)
- conn: 连接数
- time: 当前时间

它每秒钟刷新一次状态值，提供良好的可读性，通过这些参数可以观察到一个整体的性能情况。

20.3 db.serverStatus

这个命令是最常用也是最基础的查看实例运行状态的命令之一，下面我们看一下它的输出:

```
> db.serverStatus()
{
        "host" : "localhost.localdomain",
        "version" : "1.8.1",                        --服务器版本
        "process" : "mongod",
        "uptime" : 3184,                            --启动时间(秒)
        "uptimeEstimate" : 3174,
        "localTime" : ISODate("2012-05-28T11:20:22.819Z"),
```

```
        "globalLock" : {
                "totalTime" : 3183918151,
                "lockTime" : 10979,
                "ratio" : 0.00000344826703429914 9,
                "currentQueue" : {
                        "total" : 0,                    --当前全部队列量
                        "readers" : 0,                  --读请求队列量
                        "writers" : 0                   --写请求队列量
                },
                "activeClients" : {
                        "total" : 0,                     --当前全部客户端连接量
                        "readers" : 0,                  --客户端读请求量
                        "writers" : 0                   --客户端写请求量
                }
        },
        "mem" : {
                "bits" : 32,                    --32 位系统
                "resident" : 20,                --占用物量内存量
                "virtual" : 126,                 --虚拟内存量
                "supported" : true,             --是否支持扩展内存
                "mapped" : 32
        },
        "connections" : {
                "current" : 1,                  --当前活动连接量
                "available" : 818               --剩余空闲连接量
        },
……
        "indexCounters" : {
                "btree" : {
                        "accesses" : 0,                 --索引被访问量
                        "hits" : 0,                     --索引命中量
                        "misses" : 0,                    --索引偏差量
                        "resets" : 0,
                        "missRatio" : 0                 --索引偏差率(未命中率)
                }
        },
……
        "network" : {
                "bytesIn" : 1953,                   --发给此服务器的数据量(单位:byte)
                "bytesOut" : 25744,                  --此服务器发出的数据量(单位:byte)
                "numRequests" : 30                  --发给此服务器的请求量
        },
        "opcounters" : {
                "insert" : 0,                     --插入操作的量
```

```
                "query" : 1,                    --查询操作的量
                "update" : 0,                   --更新操作的量
                "delete" : 0,                   --删除操作的量
                "getmore" : 0,
                "command" : 31                  --其它操作的量
        },
……
        "ok" : 1
}
>
```

此工具提了比较详细的信息，以上已经对主要的一些参数做了说明，请大家参考。、

20.4 db.stats

db.stats 查看数据库状态信息。使用样例如下:

```
> db.stats()
{
        "db" : "test",
        "collections" : 7,                      --collection 数量
        "objects" : 28,                         --对象数量
        "avgObjSize" : 50.57142857142857,       --对象平均大小
        "dataSize" : 1416,                      --数据大小
        "storageSize" : 31744,                  --数据大小(含预分配空间)
        "numExtents" : 7,                       --事件数量
        "indexes" : 7,                          --索引数量
        "indexSize" : 57344,                    --索引大小
        "fileSize" : 50331648,                  --文件大小
        "ok" : 1                                --本次取 stats 是否正常
}
>
```

通过这个工具，可以查看所在数据库的基本信息

20.5 第三方工具

MongoDB 从一面世就得到众多开源爱好者和团队的重视，在常用的监控框架如 cacti、Nagios、Zabbix 等基础上进行扩展，进行 MongoDB 的监控都是非常方便的，有兴趣的朋友可以自已去多试试。

第五部分 架构篇

第二十一章 Replica Sets 复制集

MongoDB 支持在多个机器中通过异步复制达到故障转移和实现冗余。多机器中同一时刻只有一台是用于写操作。正是由于这个情况，为 MongoDB 提供了数据一致性的保障。担当 Primary 角色的机器能把读操作分发给 slave。

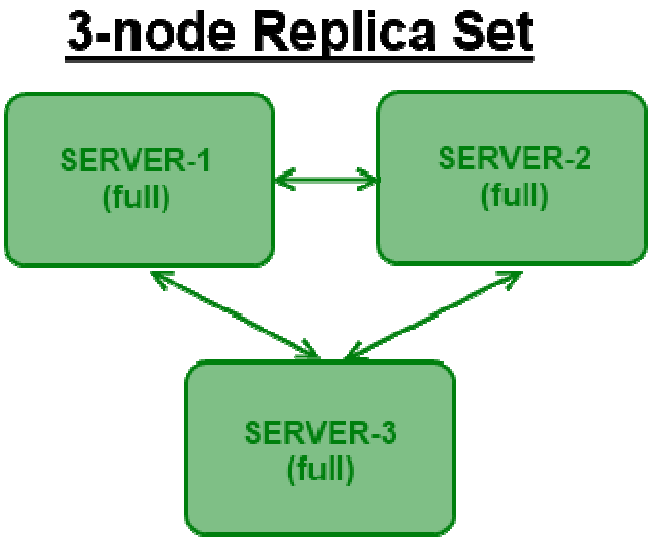
MongoDB 高可用可用分两种:

- **Master-Slave 主从复制：**

  只需要在某一个服务启动时加上–master 参数，而另一个服务加上–slave 与–source 参数，即可实现同步。MongoDB 的最新版本已不再推荐此方案。

- **Replica Sets 复制集：**

  MongoDB 在 1.6 版本对开发了新功能 replica set，这比之前的 replication 功能要强大一些，增加了故障自动切换和自动修复成员节点，各个 DB 之间数据完全一致，大大降低了维护成功。auto shard 已经明确说明不支持 replication paris，建议使用 replica set，replica set 故障切换完全自动。



如果上图所示，Replica Sets 的结构非常类似一个集群。是的，你完全可以把它当成集群，因为它确实跟集群实现的作用是一样的，其中一个节点如果出现故障，其它节点马上会将业务接过来而无须停机操作。

21.1 部署 Replica Sets

接下来将一步一步的给大家分享一下实施步骤：

1、 创建数据文件存储路径

```
[root@localhost ~]# mkdir -p /data/data/r0
[root@localhost ~]# mkdir -p /data/data/r1
[root@localhost ~]# mkdir -p /data/data/r2
```

2、 创建日志文件路径

```
[root@localhost ~]# mkdir -p /data/log
```

3、创建主从 key 文件，用于标识集群的私钥的完整路径，如果各个实例的 key file 内容不一致，程序将不能正常用。

```
[root@localhost ~]# mkdir -p /data/key
[root@localhost ~]# echo "this is rs1 super secret key" > /data/key/r0
[root@localhost ~]# echo "this is rs1 super secret key" > /data/key/r1
[root@localhost ~]# echo "this is rs1 super secret key" > /data/key/r2
[root@localhost ~]# chmod 600 /data/key/r*
```

4、启动 3 个实例

```
[root@localhost ~]# /Apps/mongo/bin/mongod --replSet rs1 --keyFile /data/key/r0 --fork --port 28010 --dbpath /data/data/r0 --logpath=/data/log/r0.log --logappend
all output going to: /data/log/r0.log
forked process: 6573
[root@localhost ~]# /Apps/mongo/bin/mongod --replSet rs1 --keyFile /data/key/r1 --fork --port 28011 --dbpath /data/data/r1 --logpath=/data/log/r1.log --logappend
all output going to: /data/log/r1.log
forked process: 6580
[root@localhost ~]# /Apps/mongo/bin/mongod --replSet rs1 --keyFile /data/key/r2 --fork --port 28012 --dbpath /data/data/r2 --logpath=/data/log/r2.log --logappend
all output going to: /data/log/r2.log
forked process: 6585
[root@localhost ~]#
```

5、配置及初始化 Replica Sets

```
[root@localhost bin]# /Apps/mongo/bin/mongo -port 28010
MongoDB shell version: 1.8.1
connecting to: 127.0.0.1:28010/test
> config_rs1 = {_id: 'rs1', members: [
...     {_id: 0, host: 'localhost:28010', priority:1},    --成员 IP 及端口， priority=1 指 PRIMARY
...     {_id: 1, host: 'localhost:28011'},
...     {_id: 2, host: 'localhost:28012'}]
...                         }
{
        "_id" : "rs1",
        "members" : [
                {
```

```
                        "_id" : 0,
                        "host" : "localhost:28010"
                },
                {
                        "_id" : 1,
                        "host" : "localhost:28011"
                },
                {
                        "_id" : 2,
                        "host" : "localhost:28012"
                }
        ]
}
> rs.initiate(config_rs1);      --初始化配置
{
        "info" : "Config now saved locally.    Should come online in about a minute.",
        "ok" : 1
}
```

6、查看复制集状态

```
> rs.status()
{
        "set" : "rs1",
        "date" : ISODate("2012-05-31T09:49:57Z"),
        "myState" : 1,
        "members" : [
                {
                        "_id" : 0,
                        "name" : "localhost:28010",
                        "health" : 1,          --1 表明正常; 0 表明异常
                        "state" : 1,           -- 1 表明是 Primary; 2 表明是 Secondary;
                        "stateStr" : "PRIMARY",    --表明此机器是主库
                        "optime" : {
                                "t" : 1338457763000,
                                "i" : 1
                        },
                        "optimeDate" : ISODate("2012-05-31T09:49:23Z"),
                        "self" : true
                },
                {
                        "_id" : 1,
                        "name" : "localhost:28011",
                        "health" : 1,
```

```
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "uptime" : 23,
                        "optime" : {
                                "t" : 1338457763000,
                                "i" : 1
                        },
                        "optimeDate" : ISODate("2012-05-31T09:49:23Z"),
                        "lastHeartbeat" : ISODate("2012-05-31T09:49:56Z")
                },
                {
                        "_id" : 2,
                        "name" : "localhost:28012",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "uptime" : 23,
                        "optime" : {
                                "t" : 1338457763000,
                                "i" : 1
                        },
                        "optimeDate" : ISODate("2012-05-31T09:49:23Z"),
                        "lastHeartbeat" : ISODate("2012-05-31T09:49:56Z")
                }
        ],
        "ok" : 1
}
rs1:PRIMARY>
```

还可以用 isMaster 查看 Replica Sets 状态。

```
rs1:PRIMARY> rs.isMaster()
{
        "setName" : "rs1",
        "ismaster" : true,
        "secondary" : false,
        "hosts" : [
                "localhost:28010",
                "localhost:28012",
                "localhost:28011"
        ],
        "maxBsonObjectSize" : 16777216,
        "ok" : 1
}
rs1:PRIMARY>
```

21.2 主从操作日志 oplog

MongoDB 的 Replica Set 架构是通过一个日志来存储写操作的，这个日志就叫做”oplog”。oplog.rs 是一个固定长度的 capped collection，它存在于”local”数据库中，用于记录 Replica Sets 操作日志。在默认情况下,对于 64 位的 MongoDB,oplog 是比较大的，可以达到 5%的磁盘空间。oplog 的大小是可以通过 mongod 的参数”—oplogSize”来改变 oplog 的日志大小。

Oplog 内容样例:

```
rs1:PRIMARY> use local
switched to db local
rs1:PRIMARY> show collections
oplog.rs
system.replset
rs1:PRIMARY> db.oplog.rs.find()
{ "ts" : { "t" : 1338457763000, "i" : 1 }, "h" : NumberLong(0), "op" : "n", "ns" : "", "o" : { "msg" : "initiating set" } }
{ "ts" : { "t" : 1338459114000, "i" : 1 }, "h" : NumberLong("5493127699725549585"), "op" : "i", "ns" : "test.c1", "o" : { "_id" : ObjectId("4fc743e9aea289af709ac6b5"), "age" : 29, "name" : "Tony" } }
rs1:PRIMARY>
```

字段说明:

- ts: 某个操作的时间戳
- op: 操作类型，如下：
  - i: insert
  - d: delete
  - u: update
- ns: 命名空间，也就是操作的 collection name
- o: document 的内容

查看 master 的 oplog 元数据信息：

```
rs1:PRIMARY> db.printReplicationInfo()
configured oplog size:   47.6837158203125MB
log length start to end: 1351secs (0.38hrs)
oplog first event time:  Thu May 31 2012 17:49:23 GMT+0800 (CST)
oplog last event time:   Thu May 31 2012 18:11:54 GMT+0800 (CST)
now:                     Thu May 31 2012 18:21:58 GMT+0800 (CST)
rs1:PRIMARY>
```

字段说明:

- configured oplog size: 配置的 oplog 文件大小
- log length start to end: oplog 日志的启用时间段
- oplog first event time: 第一个事务日志的产生时间
- oplog last event time: 最后一个事务日志的产生时间

- now: 现在的时间

查看 slave 的同步状态：

```
rs1:PRIMARY> db.printSlaveReplicationInfo()
source:    localhost:28011
           syncedTo: Thu May 31 2012 18:11:54 GMT+0800 (CST)
                     = 884secs ago (0.25hrs)
source:    localhost:28012
           syncedTo: Thu May 31 2012 18:11:54 GMT+0800 (CST)
                     = 884secs ago (0.25hrs)
rs1:PRIMARY>
```

字段说明：

- source: 从库的 IP 及端口
- syncedTo: 目前的同步情况，延迟了多久等信息

21.3 主从配置信息

在 local 库中不仅有主从日志 oplog 集合，还有一个集合用于记录主从配置信息 – system.replset

```
rs1:PRIMARY> use local
switched to db local
rs1:PRIMARY> show collections
oplog.rs
system.replset
rs1:PRIMARY> db.system.replset.find()
{ "_id" : "rs1", "version" : 1, "members" : [
        {
                "_id" : 0,
                "host" : "localhost:28010"
        },
        {
                "_id" : 1,
                "host" : "localhost:28011"
        },
        {
                "_id" : 2,
                "host" : "localhost:28012"
        }
] }
rs1:PRIMARY>
```

从这个集合中可以看出，Replica Sets 的配置信息，也可以在任何一个成员实例上执行 rs.conf() 来查看配置信息

21.4 管理维护 Replica Sets

21.4.1 读写分离

有一些第三方的工具，提供了一些可以让数据库进行读写分离的工具。我们现在是否有一个疑问，从库要是能进行查询就更好了，这样可以分担主库的大量的查询请求。

1、 先向主库中插入一条测试数据

```
[root@localhost bin]# ./mongo --port 28010
MongoDB shell version: 1.8.1
connecting to: 127.0.0.1:28010/test
rs1:PRIMARY> db.c1.insert({age:30})
db.c2rs1:PRIMARY> db.c1.find()
{ "_id" : ObjectId("4fc77f421137ea4fdb653b4a"), "age" : 30 }
```

2、 在从库进行查询等操作

```
[root@localhost bin]# ./mongo --port 28011
MongoDB shell version: 1.8.1
connecting to: 127.0.0.1:28011/test
rs1:SECONDARY> show collections
Thu May 31 22:27:17 uncaught exception: error: { "$err" : "not master and slaveok=false", "code" : 13435 }
rs1:SECONDARY>
```

当查询时报错了，说明是个从库且不能执行查询的操作

3、 让从库可以读，分担主库的压力

```
rs1:SECONDARY> db.getMongo().setSlaveOk()
not master and slaveok=false
rs1:SECONDARY> show collections
c1
system.indexes
rs1:SECONDARY> db.c1.find()
{ "_id" : ObjectId("4fc77f421137ea4fdb653b4a"), "age" : 30 }
rs1:SECONDARY>
```

看来我们要是执行 db.getMongo().setSlaveOk()， 我们就可查询从库了。

21.4.2 故障转移

复制集比传统的 Master-Slave 有改进的地方就是他可以进行故障的自动转移，如果我们停掉复制集中的一个成员，那么剩余成员会再自动选举出一个成员，做为主库，例如:
我们将 28010 这个主库停掉，然后再看一下复制集的状态

1、杀掉 28010 端口的 MongoDB

```
[root@localhost bin]# ps aux|grep mongod
root              6706     1.6      6.9   463304     6168                    Sl       21:49        0:26
/Apps/mongo/bin/mongod --replSet rs1 --keyFile /data/key/r0 --fork --port 28010
root              6733     0.4      6.7   430528     6044   ?                Sl       21:50        0:06
/Apps/mongo/bin/mongod --replSet rs1 --keyFile /data/key/r1 --fork --port 28011
root              6747     0.4      4.7   431548     4260   ?                Sl       21:50        0:06
/Apps/mongo/bin/mongod --replSet rs1 --keyFile /data/key/r2 --fork --port 28012
root         7019  0.0  0.7     5064      684 pts/2       S+      22:16      0:00 grep mongod
[root@localhost bin]# kill -9 6706
```

2、 查看复制集状态

```
[root@localhost bin]# ./mongo --port 28011
MongoDB shell version: 1.8.1
connecting to: 127.0.0.1:28011/test
rs1:SECONDARY> rs.status()
{
        "set" : "rs1",
        "date" : ISODate("2012-05-31T14:17:03Z"),
        "myState" : 2,
        "members" : [
                {
                        "_id" : 0,
                        "name" : "localhost:28010",
                        "health" : 0,
                        "state" : 1,
                        "stateStr" : "(not reachable/healthy)",
                        "uptime" : 0,
                        "optime" : {
                                "t" : 1338472279000,
                                "i" : 1
                        },
                        "optimeDate" : ISODate("2012-05-31T13:51:19Z"),
                        "lastHeartbeat" : ISODate("2012-05-31T14:16:42Z"),
                        "errmsg" : "socket exception"
                },
                {
                        "_id" : 1,
                        "name" : "localhost:28011",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "optime" : {
```

```
                                        "t" : 1338472279000,
                                        "i" : 1
                                },
                                "optimeDate" : ISODate("2012-05-31T13:51:19Z"),
                                "self" : true
                        },
                        {
                                "_id" : 2,
                                "name" : "localhost:28012",
                                "health" : 1,
                                "state" : 1,
                                "stateStr" : "PRIMARY",
                                "uptime" : 1528,
                                "optime" : {
                                        "t" : 1338472279000,
                                        "i" : 1
                                },
                                "optimeDate" : ISODate("2012-05-31T13:51:19Z"),
                                "lastHeartbeat" : ISODate("2012-05-31T14:17:02Z")
                        }
        ],
        "ok" : 1
}
rs1:SECONDARY>
```

可以看到 28010 这个端口的 MongoDB 出现了异常，而系统自动选举了 28012 这个端口为主，所以这样的故障处理机制，能将系统的稳定性大大提高。

21.4.3 增减节点

MongoDB Replica Sets 不仅提供高可用性的解决方案，它也同时提供负载均衡的解决方案，增减 Replica Sets 节点在实际应用中非常普遍，例如当应用的读压力暴增时，3 台节点的环境已不能满足需求，那么就需要增加一些节点将压力平均分配一下；当应用的压力小时，可以减少一些节点来减少硬件资源的成本；总之这是一个长期且持续的工作。

21.4.3.1 增加节点

官方给我们提了 2 个方案用于增加节点，一种是通过 oplog 来增加节点，一种是通过数据库快照(--fastsync)和 oplog 来增加节点，下面将分别介绍。

21.4.3.1.1 通过 oplog 增加节点

①、配置并启动新节点，启用 28013 这个端口给新的节点

```
[root@localhost ~]# mkdir -p /data/data/r3
```

```
[root@localhost ~]# echo "this is rs1 super secret key" > /data/key/r3
[root@localhost ~]# chmod 600 /data/key/r3
[root@localhost ~]# /Apps/mongo/bin/mongod --replSet rs1 --keyFile /data/key/r3 --fork --port 28013 --dbpath /data/data/r3 --logpath=/data/log/r3.log --logappend
all output going to: /data/log/r3.log
forked process: 10553
[root@localhost ~]#
```

②、添加此新节点到现有的 Replica Sets

```
rs1:PRIMARY> rs.add("localhost:28013")
{ "ok" : 1 }
```

③、查看 Replica Sets 我们可以清晰的看到内部是如何添加 28013 这个新节点的.

步骤一: 进行初始化

```
rs1: PRIMARY > rs.status()
{
        "set" : "rs1",
        "date" : ISODate("2012-05-31T12:17:44Z"),
        "myState" : 1,
        "members" : [
……
                {
                        "_id" : 3,
                        "name" : "localhost:28013",
                        "health" : 0,
                        "state" : 6,
                        "stateStr" : "(not reachable/healthy)",
                        "uptime" : 0,
                        "optime" : {
                                "t" : 0,
                                "i" : 0
                        },
                        "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
                        "lastHeartbeat" : ISODate("2012-05-31T12:17:43Z"),
                        "errmsg" : "still initializing"
                }
        ],
        "ok" : 1
}
```

步骤二: 进行数据同步

```
rs1:PRIMARY> rs.status()
{
        "set" : "rs1",
```

```
        "date" : ISODate("2012-05-31T12:18:07Z"),
        "myState" : 1,
        "members" : [
……
                {
                        "_id" : 3,
                        "name" : "localhost:28013",
                        "health" : 1,
                        "state" : 3,
                        "stateStr" : "RECOVERING",
                        "uptime" : 16,
                        "optime" : {
                                "t" : 0,
                                "i" : 0
                        },
                        "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
                        "lastHeartbeat" : ISODate("2012-05-31T12:18:05Z"),
                        "errmsg" : "initial sync need a member to be primary or secondary
to do our initial sync"
                }
        ],
        "ok" : 1
}
```

步骤三：初始化同步完成

```
rs1:PRIMARY> rs.status()
{
        "set" : "rs1",
        "date" : ISODate("2012-05-31T12:18:08Z"),
        "myState" : 1,
        "members" : [
……
                {
                        "_id" : 3,
                        "name" : "localhost:28013",
                        "health" : 1,
                        "state" : 3,
                        "stateStr" : "RECOVERING",
                        "uptime" : 17,
                        "optime" : {
                                "t" : 1338466661000,
                                "i" : 1
                        },
                        "optimeDate" : ISODate("2012-05-31T12:17:41Z"),
```

```
                        "lastHeartbeat" : ISODate("2012-05-31T12:18:07Z"),
                        "errmsg" : "initial sync done"
                }
        ],
        "ok" : 1
}
```

步骤四: 节点添加完成，状态正常

```
rs1:PRIMARY> rs.status()
{
        "set" : "rs1",
        "date" : ISODate("2012-05-31T12:18:10Z"),
        "myState" : 1,
        "members" : [
......
                {
                        "_id" : 3,
                        "name" : "localhost:28013",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "uptime" : 19,
                        "optime" : {
                                "t" : 1338466661000,
                                "i" : 1
                        },
                        "optimeDate" : ISODate("2012-05-31T12:17:41Z"),
                        "lastHeartbeat" : ISODate("2012-05-31T12:18:09Z")
                }
        ],
        "ok" : 1
}
```

④、验证数据已经同步过来了

```
[root@localhost data]# /Apps/mongo/bin/mongo -port 28013
MongoDB shell version: 1.8.1
connecting to: 127.0.0.1:28013/test
rs1:SECONDARY> rs.slaveOk()
rs1:SECONDARY> db.c1.find()
{ "_id" : ObjectId("4fc760d2383ede1dce14ef86"), "age" : 10 }
rs1:SECONDARY>
```

21.4.3.1.2 通过数据库快照(--fastsync)和 oplog 增加节点

通过 oplog 直接进行增加节点操作简单且无需人工干预过多，但 oplog 是 capped collection，

采用循环的方式进行日志处理，所以采用 oplog 的方式进行增加节点，有可能导致数据的不一致，因为日志中存储的信息有可能已经刷新过了。不过没关系，我们可以通过数据库快照(--fastsync)和 oplog 结合的方式来增加节点，这种方式的操作流程是，先取某一个复制集成员的物理文件来做为初始化数据，然后剩余的部分用 oplog 日志来追，最终达到数据一致性

①、取某一个复制集成员的物理文件来做为初始化数据

```
[root@localhost ~]# scp -r /data/data/r3 /data/data/r4
[root@localhost ~]# echo "this is rs1 super secret key" > /data/key/r4
[root@localhost ~]# chmod 600 /data/key/r4
```

②、在取完物理文件后，在 c1 集中插入一条新文档，用于最后验证此更新也同步了

```
rs1:PRIMARY> db.c1.find()
{ "_id" : ObjectId("4fc760d2383ede1dce14ef86"), "age" : 10 }
rs1:PRIMARY> db.c1.insert({age:20})
rs1:PRIMARY> db.c1.find()
{ "_id" : ObjectId("4fc760d2383ede1dce14ef86"), "age" : 10 }
{ "_id" : ObjectId("4fc7748f479e007bde6644ef"), "age" : 20 }
rs1:PRIMARY>
```

③、启用 28014 这个端口给新的节点

```
/Apps/mongo/bin/mongod --replSet rs1 --keyFile /data/key/r4 --fork --port 28014 --dbpath /data/data/r4 --logpath=/data/log/r4.log --logappend --fastsync
```

④、添加 28014 节点

```
rs1:PRIMARY> rs.add("localhost:28014")
{ "ok" : 1 }
```

⑤、验证数据已经同步过来了

```
[root@localhost data]# /Apps/mongo/bin/mongo -port 28014
MongoDB shell version: 1.8.1
connecting to: 127.0.0.1:28014/test
rs1:SECONDARY> rs.slaveOk()
rs1:SECONDARY> db.c1.find()
{ "_id" : ObjectId("4fc760d2383ede1dce14ef86"), "age" : 10 }
{ "_id" : ObjectId("4fc7748f479e007bde6644ef"), "age" : 20 }
rs1:SECONDARY>
```

21.4.3.2 减少节点

下面将刚刚添加的两个新节点 28013 和 28014 从复制集中去除掉，只需执行 rs.remove 指令就可以了，具体如下:

```
rs1:PRIMARY> rs.remove("localhost:28014")
```

```
{ "ok" : 1 }
rs1:PRIMARY> rs.remove("localhost:28013")
{ "ok" : 1 }
```

查看复制集状态，可以看到现在只有 28010、28011、28012 这三个成员，原来的 28013 和 28014 都成功去除了

```
rs1:PRIMARY> rs.status()
{
        "set" : "rs1",
        "date" : ISODate("2012-05-31T14:08:29Z"),
        "myState" : 1,
        "members" : [
                {
                        "_id" : 0,
                        "name" : "localhost:28010",
                        "health" : 1,
                        "state" : 1,
                        "stateStr" : "PRIMARY",
                        "optime" : {
                                "t" : 1338473273000,
                                "i" : 1
                        },
                        "optimeDate" : ISODate("2012-05-31T14:07:53Z"),
                        "self" : true
                },
                {
                        "_id" : 1,
                        "name" : "localhost:28011",
                        "health" : 1,
                        "state" : 2,
                        "stateStr" : "SECONDARY",
                        "uptime" : 34,
                        "optime" : {
                                "t" : 1338473273000,
                                "i" : 1
                        },
                        "optimeDate" : ISODate("2012-05-31T14:07:53Z"),
                        "lastHeartbeat" : ISODate("2012-05-31T14:08:29Z")
                },
                {
                        "_id" : 2,
                        "name" : "localhost:28012",
                        "health" : 1,
                        "state" : 2,
```

```
                                "stateStr" : "SECONDARY",
                                "uptime" : 34,
                                "optime" : {
                                        "t" : 1338473273000,
                                        "i" : 1
                                },
                                "optimeDate" : ISODate("2012-05-31T14:07:53Z"),
                                "lastHeartbeat" : ISODate("2012-05-31T14:08:29Z")
                        }
                ],
                "ok" : 1
}
rs1:PRIMARY>
```

第二十二章 Sharding 分片

这是一种将海量的数据水平扩展的数据库集群系统，数据分表存储在 sharding 的各个节点上，使用者通过简单的配置就可以很方便地构建一个分布式 MongoDB 集群。

MongoDB 的数据分块称为 chunk。每个 chunk 都是 Collection 中一段连续的数据记录，通常最大尺寸是 200MB，超出则生成新的数据块。

要构建一个 MongoDB Sharding Cluster，需要三种角色：

- **Shard Server**

即存储实际数据的分片，每个 Shard 可以是一个 mongod 实例，也可以是一组 mongod 实例构成的 Replica Set。为了实现每个 Shard 内部的 auto-failover，MongoDB 官方建议每个 Shard 为一组 Replica Set。

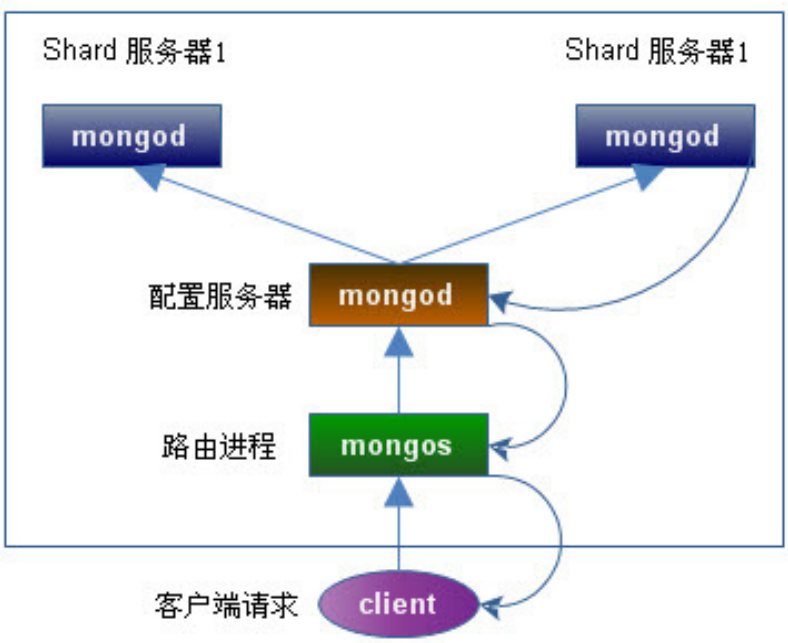
- **Config Server**

为了将一个特定的 collection 存储在多个 shard 中，需要为该 collection 指定一个 shard key，例如{age: 1} ，shard key 可以决定该条记录属于哪个 chunk。Config Servers 就是用来存储：所有 shard 节点的配置信息、每个 chunk 的 shard key 范围、chunk 在各 shard 的分布情况、该集群中所有 DB 和 collection 的 sharding 配置信息。

- **Route Process**

这是一个前端路由，客户端由此接入，然后询问 Config Servers 需要到哪个 Shard 上查询或保存记录，再连接相应的 Shard 进行操作，最后将结果返回给客户端。客户端只需要将原本发给 mongod 的查询或更新请求原封不动地发给 Routing Process，而不必关心所操作的记录存储在哪个 Shard 上。

下面我们在同一台物理机器上构建一个简单的 Sharding Cluster：
架构图如下：



- Shard Server 1：20000
- Shard Server 2：20001
- Config Server ：30000
- Route Process：40000

22.1 启动 Shard Server

```
mkdir -p /data/shard/s0          --创建数据目录
mkdir -p /data/shard/s1
mkdir -p /data/shard/log         --创建日志目录
/Apps/mongo/bin/mongod --shardsvr --port 20000 --dbpath /data/shard/s0 --fork --logpath
/data/shard/log/s0.log  --directoryperdb      --启动 Shard Server 实例 1
/Apps/mongo/bin/mongod --shardsvr --port 20001 --dbpath /data/shard/s1 --fork --logpath
/data/shard/log/s1.log  --directoryperdb      --启动 Shard Server 实例 2
```

22.2 启动 Config Server

```
mkdir -p /data/shard/config      --创建数据目录
/Apps/mongo/bin/mongod --configsvr --port 30000 --dbpath /data/shard/config --fork --logpath
/data/shard/log/config.log  --directoryperdb     --启动 Config Server 实例
```

22.3 启动 Route Process

```
/Apps/mongo/bin/mongos  --port  40000  --configdb  localhost:30000  --fork  --logpath
/data/shard/log/route.log  --chunkSize 1  --启动 Route Server 实例
```

mongos 启动参数中，chunkSize 这一项是用来指定 chunk 的大小的，单位是 MB，默认大小为 200MB，为了方便测试 Sharding 效果，我们把 chunkSize 指定为 1MB。

22.4 配置 Sharding

接下来，我们使用 MongoDB Shell 登录到 mongos，添加 Shard 节点

```
[root@localhost ~]# /Apps/mongo/bin/mongo admin --port 40000    --此操作需要连接 admin 库
MongoDB shell version: 1.8.1
connecting to: 127.0.0.1:40000/admin
> db.runCommand({ addshard:"localhost:20000" })    --添加 Shard Server
{ "shardAdded" : "shard0000", "ok" : 1 }
> db.runCommand({ addshard:"localhost:20001" })
{ "shardAdded" : "shard0001", "ok" : 1 }
> db.runCommand({ enablesharding:"test" })          --设置分片存储的数据库
{ "ok" : 1 }
> db.runCommand({ shardcollection: "test.users", key: { _id:1 }})    --设置分片的集合名称，且必须指定 Shard Key，系统会自动创建索引
{ "collectionsharded" : "test.users", "ok" : 1 }
>
```

22.5 验证 Sharding 正常工作

我们已经对 test.users 表进行了分片的设置，下面我们们插入一些数据看一下结果

```
> use test
switched to db test
> for (var i = 1; i <= 500000; i++) db.users.insert({age:i, name:"wangwenlong", addr:"Beijing", country:"China"})
> db.users.stats()
{
        "sharded" : true,               --说明此表已被 shard
        "ns" : "test.users",
        "count" : 500000,
        "size" : 48000000,
        "avgObjSize" : 96,
        "storageSize" : 66655232,
        "nindexes" : 1,
        "nchunks" : 43,
        "shards" : {
                "shard0000" : {             --在此分片实例上约有 24.5M 数据
                        "ns" : "test.users",
                        "count" : 254889,
                        "size" : 24469344,
                        "avgObjSize" : 96,
```

```
                        "storageSize" : 33327616,
                        "numExtents" : 8,
                        "nindexes" : 1,
                        "lastExtentSize" : 12079360,
                        "paddingFactor" : 1,
                        "flags" : 1,
                        "totalIndexSize" : 11468800,
                        "indexSizes" : {
                                "_id_" : 11468800
                        },
                        "ok" : 1
                },
                "shard0001" : {         --在此分片实例上约有 23.5M 数据
                        "ns" : "test.users",
                        "count" : 245111,
                        "size" : 23530656,
                        "avgObjSize" : 96,
                        "storageSize" : 33327616,
                        "numExtents" : 8,
                        "nindexes" : 1,
                        "lastExtentSize" : 12079360,
                        "paddingFactor" : 1,
                        "flags" : 1,
                        "totalIndexSize" : 10649600,
                        "indexSizes" : {
                                "_id_" : 10649600
                        },
                        "ok" : 1
                }
        },
        "ok" : 1
}
>
```

我们看一下磁盘上的物理文件情况

```
[root@localhost bin]# ll /data/shard/s0/test                    --此分片实例上有数据产生
总计 262420
-rw------- 1 root root    16777216 06-03 15:21 test.0
-rw------- 1 root root    33554432 06-03 15:21 test.1
-rw------- 1 root root    67108864 06-03 15:22 test.2
-rw------- 1 root root 134217728 06-03 15:24 test.3
-rw------- 1 root root    16777216 06-03 15:21 test.ns
[root@localhost bin]# ll /data/shard/s1/test                    --此分片实例上有数据产生
总计 262420
```

```
-rw------- 1 root root    16777216 06-03 15:21 test.0
-rw------- 1 root root    33554432 06-03 15:21 test.1
-rw------- 1 root root    67108864 06-03 15:22 test.2
-rw------- 1 root root 134217728 06-03 15:23 test.3
-rw------- 1 root root    16777216 06-03 15:21 test.ns
[root@localhost bin]#
```

看上述结果，表明 test.users 集合已经被分片处理了，但是通过 mongos 路由，我们并感觉不到是数据存放在哪个 shard 的 chunk 上的，这就是 MongoDB 用户体验上的一个优势，即对用户是透明的。

22.6 管理维护 Sharding

22.6.1 列出所有的 Shard Server

```
> db.runCommand({ listshards: 1 })          --列出所有的 Shard Server
{
        "shards" : [
                {
                        "_id" : "shard0000",
                        "host" : "localhost:20000"
                },
                {
                        "_id" : "shard0001",
                        "host" : "localhost:20001"
                }
        ],
        "ok" : 1
}
```

22.6.2 查看 Sharding 信息

```
> printShardingStatus()                     --查看 Sharding 信息
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
      { "_id" : "shard0000", "host" : "localhost:20000" }
      { "_id" : "shard0001", "host" : "localhost:20001" }
  databases:
        { "_id" : "admin", "partitioned" : false, "primary" : "config" }
        { "_id" : "test", "partitioned" : true, "primary" : "shard0000" }
                test.users chunks:
```

```
                                        shard0000           1
                                { "_id" : { $minKey : 1 } } -->> { "_id" : { $maxKey : 1 } } on :
shard0000 { "t" : 1000, "i" : 0 }

>
```

22.6.3 判断是否是 Sharding

```
> db.runCommand({ isdbgrid:1 })
{ "isdbgrid" : 1, "hostname" : "localhost", "ok" : 1 }
>
```

22.6.4 对现有的表进行 Sharding

刚才我们是对表 test.users 进行分片了，下面我们将对库中现有的未分片的表 test.users_2 进行分片处理

表最初状态如下，可以看出他没有被分片过：

```
> db.users_2.stats()
{
        "ns" : "test.users_2",
        "sharded" : false,
        "primary" : "shard0000",
        "ns" : "test.users_2",
        "count" : 500000,
        "size" : 48000016,
        "avgObjSize" : 96.000032,
        "storageSize" : 61875968,
        "numExtents" : 11,
        "nindexes" : 1,
        "lastExtentSize" : 15001856,
        "paddingFactor" : 1,
        "flags" : 1,
        "totalIndexSize" : 20807680,
        "indexSizes" : {
                "_id_" : 20807680
        },
        "ok" : 1
}
```

对其进行分片处理:

```
> use admin
```

```
switched to db admin
> db.runCommand({ shardcollection: "test.users_2", key: { _id:1 }})
{ "collectionsharded" : "test.users_2", "ok" : 1 }
```

再次查看分片后的表的状态，可以看到它已经被我们分片了

```
> use test
switched to db test
> db.users_2.stats()
{
        "sharded" : true,
        "ns" : "test.users_2",
        "count" : 505462,
……
        "shards" : {
                "shard0000" : {
                        "ns" : "test.users_2",
……
                        "ok" : 1
                },
                "shard0001" : {
                        "ns" : "test.users_2",
……
                        "ok" : 1
                }
        },
        "ok" : 1
}
>
```

22.6.5 新增 Shard Server

刚才我们演示的是新增分片表，接下来我们演示如何新增 Shard Server

启动一个新 Shard Server 进程

```
[root@localhost ~]# mkdir /data/shard/s2
[root@localhost ~]# /Apps/mongo/bin/mongod --shardsvr --port 20002 --dbpath /data/shard/s2
--fork --logpath /data/shard/log/s2.log    --directoryperdb
all output going to: /data/shard/log/s2.log
forked process: 6772
```

配置新 Shard Server

```
[root@localhost ~]# /Apps/mongo/bin/mongo admin --port 40000
MongoDB shell version: 1.8.1
```

```
connecting to: 127.0.0.1:40000/admin
> db.runCommand({ addshard:"localhost:20002" })
{ "shardAdded" : "shard0002", "ok" : 1 }
> printShardingStatus()
--- Sharding Status ---
   sharding version: { "_id" : 1, "version" : 3 }
   shards:
        { "_id" : "shard0000", "host" : "localhost:20000" }
        { "_id" : "shard0001", "host" : "localhost:20001" }
        { "_id" : "shard0002", "host" : "localhost:20002" }     --新增 Shard Server
   databases:
           { "_id" : "admin", "partitioned" : false, "primary" : "config" }
           { "_id" : "test", "partitioned" : true, "primary" : "shard0000" }
                   test.users chunks:
                                   shard0002       2
                                   shard0000       21
                                   shard0001       21
                           too many chunksn to print, use verbose if you want to force print
                   test.users_2 chunks:
                                   shard0001       46
                                   shard0002       1
                                   shard0000       45
                           too many chunksn to print, use verbose if you want to force print
```

查看分片表状态，以验证新 Shard Server

```
> use test
switched to db test
> db.users_2.stats()
{
           "sharded" : true,
           "ns" : "test.users_2",
……
                   "shard0002" : {        --新的 Shard Server 已有数据
                           "ns" : "test.users_2",
                           "count" : 21848,
                           "size" : 2097408,
                           "avgObjSize" : 96,
                           "storageSize" : 2793472,
                           "numExtents" : 5,
                           "nindexes" : 1,
                           "lastExtentSize" : 2097152,
                           "paddingFactor" : 1,
                           "flags" : 1,
                           "totalIndexSize" : 1277952,
```

```
                                "indexSizes" : {
                                        "_id_" : 1277952
                                },
                                "ok" : 1
                        }
                },
                "ok" : 1
}
>
```

我们可以发现，当我们新增 Shard Server 后数据自动分布到了新 Shard 上，这是由 MongoDB 内部自己实现的。

22.6.6 移除 Shard Server

有些时候有于硬件资源有限，所以我们不得不进行一些回收工作，下面我们就要将刚刚启用的 Shard Server 回收，系统首先会将在这个即将被移除的 Shard Server 上的数据先平均分配到其它的 Shard Server 上，然后最终在将这个 Shard Server 踢下线，我们需要不停的调用 db.runCommand({"removeshard" : "localhost:20002"});来观察这个移除操作进行到哪里了：

```
> use admin
switched to db admin
> db.runCommand({"removeshard" : "localhost:20002"});
{
        "msg" : "draining started successfully",
        "state" : "started",
        "shard" : "shard0002",
        "ok" : 1
}
> db.runCommand({"removeshard" : "localhost:20002"});
{
        "msg" : "draining ongoing",
        "state" : "ongoing",
        "remaining" : {
                "chunks" : NumberLong(44),
                "dbs" : NumberLong(0)
        },
        "ok" : 1
}
……
> db.runCommand({"removeshard" : "localhost:20002"});
{
        "msg" : "draining ongoing",
        "state" : "ongoing",
        "remaining" : {
```

```
                        "chunks" : NumberLong(1),
                        "dbs" : NumberLong(0)
                },
                "ok" : 1
}
> db.runCommand({"removeshard" : "localhost:20002"});
{
                "msg" : "removeshard completed successfully",
                "state" : "completed",
                "shard" : "shard0002",
                "ok" : 1
}
> db.runCommand({"removeshard" : "localhost:20002"});
{
                "assertion" : "can't find shard for: localhost:20002",
                "assertionCode" : 13129,
                "errmsg" : "db assertion failure",
                "ok" : 0
}
```

最终移除后，当我们再次调用 db.runCommand({"removeshard" : "localhost:20002"});的时候系统会报错，已便通知我们不存在 20002 这个端口的 Shard Server 了，因为它已经被移除掉了。

接下来我们看一下表中的数据分布：

```
> use test
switched to db test
> db.users_2.stats()
{
                "sharded" : true,
                "ns" : "test.users_2",
                "count" : 500000,
                "size" : 48000000,
                "avgObjSize" : 96,
                "storageSize" : 95203584,
                "nindexes" : 1,
                "nchunks" : 92,
                "shards" : {
                        "shard0000" : {
                                "ns" : "test.users_2",
                                "count" : 248749,
                                "size" : 23879904,
                                "avgObjSize" : 96,
                                "storageSize" : 61875968,
                                "numExtents" : 11,
                                "nindexes" : 1,
```

```
                        "lastExtentSize" : 15001856,
                        "paddingFactor" : 1,
                        "flags" : 1,
                        "totalIndexSize" : 13033472,
                        "indexSizes" : {
                                "_id_" : 13033472
                        },
                        "ok" : 1
                },
                "shard0001" : {
                        "ns" : "test.users_2",
                        "count" : 251251,
                        "size" : 24120096,
                        "avgObjSize" : 96,
                        "storageSize" : 33327616,
                        "numExtents" : 8,
                        "nindexes" : 1,
                        "lastExtentSize" : 12079360,
                        "paddingFactor" : 1,
                        "flags" : 1,
                        "totalIndexSize" : 10469376,
                        "indexSizes" : {
                                "_id_" : 10469376
                        },
                        "ok" : 1
                }
        },
        "ok" : 1
}
```

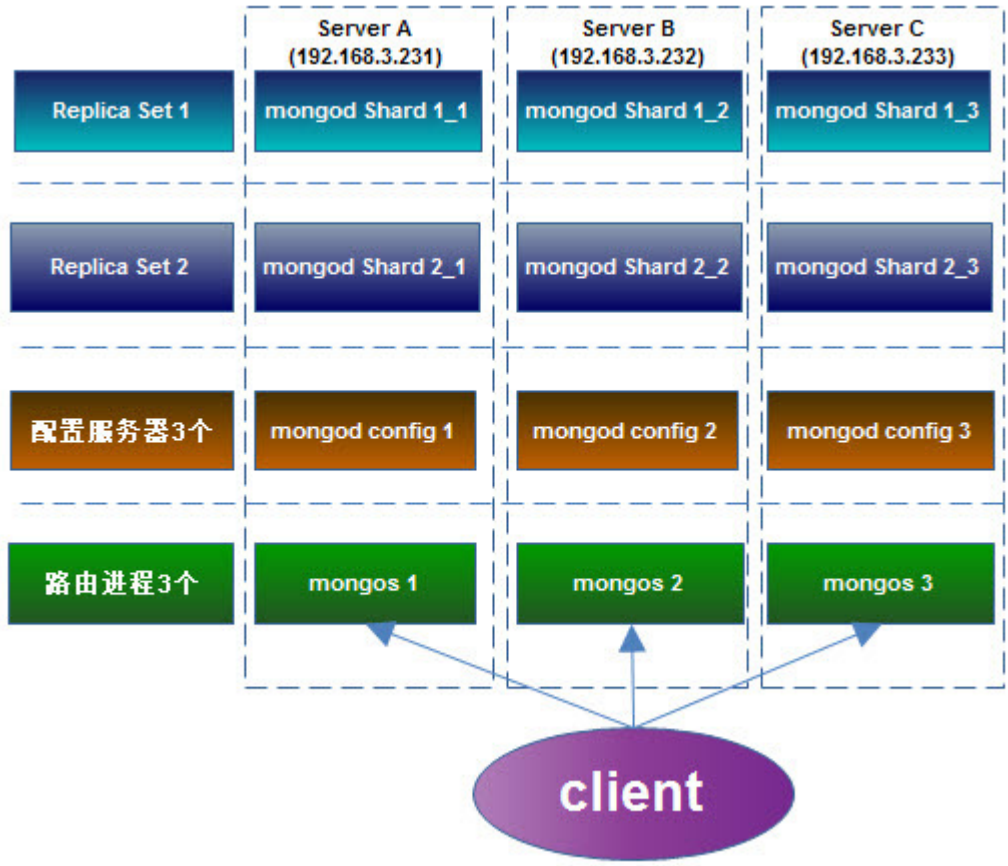

可以看出数据又被平均分配到了另外 2 台 Shard Server 上了，对业务没什么特别大的影响。

第二十三章 Replica Sets + Sharding

MongoDB Auto-Sharding 解决了海量存储和动态扩容的问题，但离实际生产环境所需的高可靠、高可用还有些距离，所以有了” Replica Sets + Sharding”的解决方案:

- Shard:
  使用 Replica Sets，确保每个数据节点都具有备份、自动容错转移、自动恢复能力。
- Config:
  使用 3 个配置服务器，确保元数据完整性
- Route:
  使用 3 个路由进程，实现负载平衡，提高客户端接入性能

以下我们配置一个 Replica Sets + Sharding 的环境，架构图如下:



开放的端口如下：

主机	IP	服务及端口
Server A	192.168.3.231	mongod shard1_1:27017 mongod shard2_1:27018 mongod config1:20000 mongs1:30000
Server B	192.168.3.232	mongod shard1_2:27017 mongod shard2_2:27018 mongod config2:20000 mongs2:30000
Server C	192.168.3.233	mongod shard1_3:27017 mongod shard2_3:27018 mongod config3:20000 mongs3:30000

23.1 创建数据目录

在 Server A 上:

```
[root@localhost bin]# mkdir -p /data/shard1_1
[root@localhost bin]# mkdir -p /data/shard2_1
```

```
[root@localhost bin]# mkdir -p /data/config
```

在 **Server B** 上:

```
[root@localhost bin]# mkdir -p /data/shard1_2
[root@localhost bin]# mkdir -p /data/shard2_2
[root@localhost bin]# mkdir -p /data/config
```

在 **Server C** 上:

```
[root@localhost bin]# mkdir -p /data/shard1_3
[root@localhost bin]# mkdir -p /data/shard2_3
[root@localhost bin]# mkdir -p /data/config
```

23.2 配置 Replica Sets

23.2.1 配置 shard1 所用到的 Replica Sets

在 **Server A** 上:

```
[root@localhost bin]# /Apps/mongo/bin/mongod --shardsvr --replSet shard1 --port 27017
--dbpath /data/shard1_1 --logpath /data/shard1_1/shard1_1.log --logappend --fork
[root@localhost bin]# all output going to: /data/shard1_1/shard1_1.log
forked process: 18923
```

在 **Server B** 上:

```
[root@localhost bin]# /Apps/mongo/bin/mongod --shardsvr --replSet shard1 --port 27017
--dbpath /data/shard1_2 --logpath /data/shard1_2/shard1_2.log --logappend --fork
forked process: 18859
[root@localhost bin]# all output going to: /data/shard1_2/shard1_2.log

[root@localhost bin]#
```

在 **Server C** 上:

```
[root@localhost bin]# /Apps/mongo/bin/mongod --shardsvr --replSet shard1 --port 27017
--dbpath /data/shard1_3 --logpath /data/shard1_3/shard1_3.log --logappend --fork
all output going to: /data/shard1_3/shard1_3.log
forked process: 18768
[root@localhost bin]#
```

用 mongo 连接其中一台机器的 27017 端口的 mongod，初始化 Replica Sets“shard1”，执行:

```
[root@localhost bin]# ./mongo --port 27017
MongoDB shell version: 1.8.1
connecting to: 127.0.0.1:27017/test
> config = {_id: 'shard1', members: [
```

```
...                                     {_id: 0, host: '192.168.3.231:27017'},
...                                     {_id: 1, host: '192.168.3.232:27017'},
...                                     {_id: 2, host: '192.168.3.233:27017'}]
...                 }
......
> rs.initiate(config)
{
            "info" : "Config now saved locally.    Should come online in about a minute.",
            "ok" : 1
}
```

23.2.2 配置 shard2 所用到的 Replica Sets

在 Server A 上:

```
[root@localhost bin]# /Apps/mongo/bin/mongod --shardsvr --replSet shard2 --port 27018
--dbpath /data/shard2_1 --logpath /data/shard2_1/shard2_1.log --logappend --fork
all output going to: /data/shard2_1/shard2_1.log
[root@localhost bin]# forked process: 18993

[root@localhost bin]#
```

在 Server B 上:

```
[root@localhost bin]# /Apps/mongo/bin/mongod --shardsvr --replSet shard2 --port 27018
--dbpath /data/shard2_2 --logpath /data/shard2_2/shard2_2.log --logappend --fork
all output going to: /data/shard2_2/shard2_2.log
forked process: 18923
[root@localhost bin]#
```

在 Server C 上:

```
[root@localhost bin]# /Apps/mongo/bin/mongod --shardsvr --replSet shard2 --port 27018
--dbpath /data/shard2_3 --logpath /data/shard2_3/shard2_3.log --logappend --fork
[root@localhost bin]# all output going to: /data/shard2_3/shard2_3.log
forked process: 18824

[root@localhost bin]#
```

用 mongo 连接其中一台机器的 27018 端口的 mongod，初始化 Replica Sets “shard2”，执行:

```
[root@localhost bin]# ./mongo --port 27018
MongoDB shell version: 1.8.1
connecting to: 127.0.0.1:27018/test
> config = {_id: 'shard2', members: [
...                                     {_id: 0, host: '192.168.3.231:27018'},
```

```
...                                     {_id: 1, host: '192.168.3.232:27018'},
...                                     {_id: 2, host: '192.168.3.233:27018'}]
...                 }
......
> rs.initiate(config)
{
        "info" : "Config now saved locally.  Should come online in about a minute.",
        "ok" : 1
}
```

23.3 配置 3 台 Config Server

在 Server A、B、C 上执行:

```
/Apps/mongo/bin/mongod --configsvr --dbpath /data/config --port 20000 --logpath /data/config/config.log --logappend --fork
```

23.4 配置 3 台 Route Process

在 Server A、B、C 上执行:

```
/Apps/mongo/bin/mongos                                                  --configdb 192.168.3.231:20000,192.168.3.232:20000,192.168.3.233:20000 --port 30000 --chunkSize 1 --logpath /data/mongos.log --logappend --fork
```

23.5 配置 Shard Cluster

连接到其中一台机器的端口 30000 的 mongos 进程，并切换到 admin 数据库做以下配置

```
[root@localhost bin]# ./mongo --port 30000
MongoDB shell version: 1.8.1
connecting to: 127.0.0.1:30000/test
> use admin
switched to db admin
>db.runCommand({addshard:"shard1/192.168.3.231:27017,192.168.3.232:27017,192.168.3.233:27017"});
{ "shardAdded" : "shard1", "ok" : 1 }
>db.runCommand({addshard:"shard2/192.168.3.231:27018,192.168.3.232:27018,192.168.3.233:27018"});
{ "shardAdded" : "shard2", "ok" : 1 }
>
```

激活数据库及集合的分片

```
db.runCommand({ enablesharding:"test" })
db.runCommand({ shardcollection: "test.users", key: { _id:1 }})
```

23.6 验证 Sharding 正常工作

连接到其中一台机器的端口 30000 的 mongos 进程，并切换到 test 数据库，以便添加测试数据

```
use test
for(var i=1;i<=200000;i++) db.users.insert({id:i,addr_1:"Beijing",addr_2:"Shanghai"});
db.users.stats()
{
        "sharded" : true,
        "ns" : "test.users",
        "count" : 200000,
        "size" : 25600384,
        "avgObjSize" : 128,
        "storageSize" : 44509696,
        "nindexes" : 2,
        "nchunks" : 15,
        "shards" : {
                "shard0000" : {
……
                },
                "shard0001" : {
……
                }
        },
        "ok" : 1
}
```

可以看到Sharding搭建成功了，跟我们期望的结果一致，至此我们就将Replica Sets与Sharding结合的架构也学习完毕了。