科大恒星信息技术有限公司

MongoDB 学习手册

作者：李三来

邮箱：sanlai_lee@lisanlai.cn

2011

合肥市高新区软件园 2 号楼 4 层

说明

文档大部分内容来自 MongoDB 官网网站，仅供学习使用！

目录

一、 Mongodb 简介

1. MongoDB 的名称取自“humongous”(巨大的) 的中间部分，足见 mongodb 的宗旨在处理大量数据上面
2. MongoDB 是一个开源的、面向文档存储的数据库，属于 Nosql 数据库的一种
3. MongoDB 可运行在 unix、Windows 和 OSX 平台上，支持 32 位和 64 位应用，并且提供了 java、php、c、c++、c#、javaScript 多种语言的驱动程序
4. 目前正在使用 MongoDB 的网站和企业已经超过 100 多家

二、 MongoDB 特性

MongoDB 是一个可扩展、高性能的下一代数据库，由 C++语言编写，旨在为 web 应用提供可扩展的高性能数据存储解决方案。它的特点是高性能、易部署、易使用，存储数据非常方便，主要特性有：

- ✧ 模式自由，支持动态查询、完全索引，可轻易查询文档中内嵌的对象及数组
- ✧ 面向文档存储，易存储对象类型的数据, 包括文档内嵌对象及数组
- ✧ 高效的数据存储,支持二进制数据及大型对象(如照片和视频)
- ✧ 支持复制和故障恢复；提供了主-从、主-主模式的数据复制及服务器之间的数据复制
- ✧ 自动分片以支持云级别的伸缩性，支持水平的数据库集群，可动态添加额外的服务器

适用场景：

- ◆ 适合作为信息基础设施的持久化缓存层
- ◆ 适合实时的插入，更新与查询，并具备应用程序实时数据存储所需的复制及高度伸缩性
- ◆ Mongo 的 BSON 数据格式非常适合文档化格式的存储及查询
- ◆ 适合由数十或数百台服务器组成的数据库。因为 Mongo 已经包含了对 MapReduce 引擎的内置支持

不适用场景：

- 要求高度事务性的系统
- 传统的商业智能应用
- 复杂的跨文档(表)级联查询

三、 MongoDB 的工作方式

- 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并且最像关系型数据库。
- 传统的关系数据库一般由数据库(database)、表(table)、记录(record)三个层次概念组成，MongoDB 同样也是由数据库(database) 、集合(collection)、文档对象(document)三个层次组成。 MongoDB 里的集合对应于关系型数据库里的表，但是集合中没有列、行和关系的概念，这体现了模式自由的特点。
- 在 MongoDB 中数据以单文档为单位存储，这样就能在单个数据对象中表示复杂的关系。文档可以由独立的基本类型属性、内嵌文档或文档数组组成。
- MongoDB 存储的数据格式是 key-value 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,包括数组和文档对象。这种数据格式称作 BSON，即 “Binary SerializedDocument Notation”，是一种类似 JSON 的二进制序列化文档 。
- MongoDB 是一个免安装的数据库,将它解压后生成一个 bin 目录,其中包含 11 个工具命令,除此之外不再需要任何其它的二进制依

赖文件。

- 通常情况下启动数据库只需要关注其中的两个命令：mongod 和 mongo。前者是 MongoDB 数据库进程本身,是核心数据库服务器,后者是命令行 Shell 客户端,其使用方法通常类似于 MySQL 命令行 Shell 客户端,用于确保所有内容都已正常安装且能正常运行,并且可以对数据进行 CRUD 操作、执行管理任务等等。
- MongoDB 使用了内存映射文件进行数据管理，把所有空闲内存当缓存使用,且不能指定内存大小。这既是优点也是缺点：优点--可以最大限度提升性能；缺点--容易受其它程序干扰。
- 数据空间采用预分配,目的是为了避免形成过多的硬盘碎片。它为每个数据库分配一系列文件，每个数据文件都会被预分配一个大小，第一个文件名字为“.0 ”，大小为 64MB，第二个文件“.1”为 128MB ,依此类推,在 32 位模式运行时支持的最大文件为 2GB。随着数据量的增加，可以在其数据目录里看到这些不断递增的文件。
- MongoDB 没有自动递增或序列特性,当 BSON 对象插入到数据库中时,如果没有提供“_id”字段 ,数据库会自动生成一个 ObjectId 对象作为“_id”的值插入到集合中作为该文档的主键(这就避免了其它数据库意外地选择相同的惟一标识符的情况) ,“_id”的值由 4 字节的时间戳,3 字节的机器号,2 字节的进程 id 以及 3 字节的自增计数组成。当然字段“_id”的值可以手动生成(任意类型都可),只要能够保证惟一性。

- 每个插入的 BSON 对象大小不能超过 4MB，如果超过 4M 时需使用 GridFS 来储存数据。
- 为避免记录删除后的数据的大规模挪动，原记录空间不删除，只标记“已删除”即可，以后还可以重复利用，所以删除记录不释放空间。

四、 MongoDB 的下载

1. MongoDB 的官网：http://www.mongodb.org/
2. MongoDB 的下载地址：http://www.mongodb.org/downloads

	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
1.8.2-rc2 *5/20/2011* 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

如上图，红色标记的是我们用来学习的版本

3. 在联网的情况下，也可以通过如下命令来获取安装介质：

```
$curl http://downloads.mongodb.org/linux/mongodb-linux-x86_64-1.8.2-rc2.tgz > mongo.tgz
```

五、 MongoDB 的安装

1. 解压安装介质：

```
$ tar xzf mongo.tgz
```

2. 进入 mongo/bin 目录，可以看到如下的文件列表：

```
-rwxr-xr-x 1 tsaip tsaip 7719480 05-21 05:05 bsondump
-rwxr-xr-x 1 tsaip tsaip 3364032 05-21 05:05 mongo
-rwxr-xr-x 1 tsaip tsaip 7749408 05-21 05:05 mongod
-rwxr-xr-x 1 tsaip tsaip 7756376 05-21 05:05 mongodump
-rwxr-xr-x 1 tsaip tsaip 7727672 05-21 05:05 mongoexport
-rwxr-xr-x 1 tsaip tsaip 7731768 05-21 05:05 mongofiles
-rwxr-xr-x 1 tsaip tsaip 7735864 05-21 05:05 mongoimport
-rwxr-xr-x 1 tsaip tsaip 7735992 05-21 05:05 mongorestore
-rwxr-xr-x 1 tsaip tsaip 5035000 05-21 05:06 mongos
-rwxr-xr-x 1 tsaip tsaip 1176496 05-21 05:06 mongosniff
-rwxr-xr-x 1 tsaip tsaip 7797656 05-21 05:06 mongostat
```

3. 启动 mongodb 服务进程：

MongoDB 默认的数据库文件的位置是/data/db，但是它不会自动的生产目录，所以如果采用默认的文件位置的话，我们需要自己先建立该目录，如下：

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

接下来可以启动 mongodb 的服务了：

```
$ ./mongod
```

我们也可以用-dbpath 选项来指定自己的数据库位置，如下：

```
$ ./mongod -dbpath ../../data/db
```

启动成功以后，会有如下界面显示：

```
[tsaip@ossesbwas bin]$ Mon May 30 12:17:16 [initandlisten] MongoDB starting :
pid=3662 port=27017 dbpath=../../data/db 64-bit
Mon May 30 12:17:16 [initandlisten] db version v1.8.2-rc2, pdfile version 4.5
Mon May 30 12:17:16 [initandlisten] git version:
373038f53049071fddb5404698c8bebf99e3b51f
Mon May 30 12:17:16 [initandlisten] build sys info: Linux bs-linux64.10gen.cc
2.6.21.7-2.ec2.v1.2.fc8xen #1 SMP Fri Nov 20 17:48:28 EST 2009 x86_64
BOOST_LIB_VERSION=1_41
Mon May 30 12:17:16 [initandlisten] waiting for connections on port 27017
Mon May 30 12:17:16 [websvr] web admin interface listening on port 28017
```

4. 启动命令常用参数选项说明

mongod 启动数据库进程

--dbpath 指定数据库的目录

--port 指定数据库的端口,默认是 27017

--bind_ip 绑定 IP

--directoryperdb 为每个 db 创建一个独立子目录

--logpath 指定日志存放目录

--logappend 指定日志生成方式(追加/覆盖)

--pidfilepath 指定进程文件路径，如果不指定，那么将不产生进程文件

--keyFile 集群模式的关键标识

--cpu 周期性的显示 CPU 和 IO 的利用率

--journal 启用日志

--ipv6 启用 IPV6 支持

--nssize 指定.ns 文件的大小，单位 MB，默认是 16M，最大是 2GB

--maxConns 最大的并发连接数

--notablescan 不允许进行表扫描

--quota 限制每个数据库的文件个数，默认是 8 个

--quotaFiles 每个数据库的文件个数，配合—quota 参数

--noprealloc 关闭数据文件的预分配功能

……更多的参数选项利用 mongod –help 进行查看

5. 启动客户端

服务端进程启动成功以后，就可以启动客户端，跟服务端进行连接，如下：

```
[tsaip@ossesbwas bin]$ ./mongo

MongoDB shell version: 1.8.2-rc2

connecting to: test

Mon May 30 12:36:18 [initandlisten] connection accepted from

127.0.0.1:59154 #1

>
```

6. 关闭 mongodb 服务

```
> use admin;
switched to db admin
> db.shutdownServer();
Mon May 30 12:41:52 [conn2] terminating, shutdown command received
Mon May 30 12:41:52 dbexit: shutdown called
Mon May 30 12:41:52 [conn2] shutdown: going to close listening sockets...
Mon May 30 12:41:52 [conn2] closing listening socket: 5
Mon May 30 12:41:52 [conn2] closing listening socket: 6
Mon May 30 12:41:52 [conn2] closing listening socket: 7
Mon May 30 12:41:52 [conn2] closing listening socket: 8
Mon May 30 12:41:52 [conn2] removing socket file: /tmp/mongodb-27017.sock
Mon May 30 12:41:52 [conn2] removing socket file: /tmp/mongodb-28017.sock
Mon May 30 12:41:52 [conn2] shutdown: going to flush diaglog...
Mon May 30 12:41:52 [conn2] shutdown: going to close sockets...
```

```
Mon May 30 12:41:52 [conn2] shutdown: waiting for fs preallocator...
Mon May 30 12:41:52 [conn2] shutdown: closing all files...
Mon May 30 12:41:52 closeAllFiles() finished
Mon May 30 12:41:52 [conn2] shutdown: removing fs lock...
Mon May 30 12:41:52 dbexit: really exiting now
Mon May 30 12:41:52 DBClientCursor::init call() failed
Mon May 30 12:41:52 query failed : admin.$cmd { shutdown: 1.0 } to: 127.0.0.1
server should be down...
Mon May 30 12:41:52 trying reconnect to 127.0.0.1
Mon May 30 12:41:52 reconnect 127.0.0.1 failed couldn't connect to server
127.0.0.1
Mon May 30 12:41:52 Error: error doing query: unknown shell/collection.js:150
```

六、 MongoDB 数据类型

MongoDB 除了包含这些 string, integer, boolean, double, null, array, and object 基本的数据类型外，还包含：date, object id, binary data, regular expression, and code 这些附加的数据类型。

1. Timestamp 类型

Timestamp 类型从 1.8 版本开始支持，Timestamp 有一个特殊的用法:timestamp 类型的字段必须是位于文档的前两位.看下面例子:

```
//位于第三个字段
> db.coll.insert({_id:1,x:2,y:new Timestamp()});
> db.coll.findOne({_id:1});
{ "_id" : 1, "x" : 2, "y" : { "t" : 0, "i" : 0 } }
//位于第二个字段
> db.coll.insert({_id:2,y:new Timestamp(),x:2});
> db.coll.findOne({_id:2});
{ "_id" : 2, "y" : { "t" : 1306746538000, "i" : 1 }, "x" : 2 }
```

2. ObjectId 类型

在 mongodb 中,几乎每个文档（除了某些系统的 Collection 或者某些

Capped Collection）都要求有一个主键:_id,用来唯一标识他们，通常—它的值就是 ObjectId 类型。当用户往文档中插入一条新记录的时候，如果没有指定_id 属性，那么 MongoDB 会自动生成一个 ObjectId 类型的值，保存为_id 的值。

_id 的值可以为任何类型，除了数组，在实际应用中，鼓励用户自己定义_id 值，但是要保证它的唯一性。如下有两个方案：

✧ Sequence Numbers：序列号

传统的数据库中，通常用一个递增的序列来提供主键，在 MongoDB 中用 ObjectId 的来代替，我们可以通过如下的函数来获取主键：

```
function counter(name) {
    var ret = db.counters.findAndModify({query:{_id:name},
update:{$inc : {next:1}}, "new":true, upsert:true});
    return ret.next;
}
db.users.insert({_id:counter("users"), name:"Sarah C."}) // _id : 1
db.users.insert({_id:counter("users"), name:"Bob D."}) // _id :2
```

✧ 利用 UUID

如果用 UUID 来提供主键，我们的应用需要自己去生成 UUID，考虑到效率，建议把 UUID 保存为 BSON BinData 类型，如果用例中对效率要求不是很高，也可以保存为字符串类型。

3. 数据库关联

在 MongoDB 中，通常的关联习惯有两种，一种是简单的手动关联，一种是用 DBRef。

✧ 简单的手工关联

```
//查找
> db.post.save({title:'MongoDB Manual',author:'sam'});
> p = db.post.findOne();
{
    "_id" : ObjectId("4de36b33282677bdc555a83a"),
    "title" : "MongoDB Manual",
    "author" : "sam"
}

//关联
> db.authors.findOne({name:p.author});
{
    "_id" : ObjectId("4de36c14282677bdc555a83b"),
    "name" : "sam",
    "age" : 24,
    "email" : "sanlai_lee@lisanlai.cn"
}
```

✧ 利用 DBRef 关联

DBRef 关联语法：

```
{ $ref : <collname>, $id : <idvalue>[, $db : <dbname>] }
```

例子：

```
> x = { name : 'Biology' }
{ "name" : "Biology" }
> db.courses.save(x)
> x
{ "name" : "Biology", "_id" :
```

```
ObjectId("4b0552b0f0da7d1eb6f126a1") }
> stu = { name : 'Joe', classes : [ new DBRef('courses',
x._id) ] }
// or we could write:
// stu = { name : 'Joe', classes :
[ {$ref:'courses',$id:x._id} ] }
> db.students.save(stu)
> stu
{
        "name" : "Joe",
        "classes" : [
                {
                        "$ref" : "courses",
                        "$id" :
ObjectId("4b0552b0f0da7d1eb6f126a1")
                }
        ],
        "_id" : ObjectId("4b0552e4f0da7d1eb6f126a2")
}
> stu.classes[0]
{ "$ref" : "courses", "$id" :
ObjectId("4b0552b0f0da7d1eb6f126a1") }
> stu.classes[0].fetch()
{ "_id" : ObjectId("4b0552b0f0da7d1eb6f126a1"), "name" :
"Biology"
```

七、 GridFS 文件系统

由于在 MongoDB 中，1.7 版本之前，BSON 对象的大小只有 4MB 的限制，1.7-1.8 版本，大小限制是 16MB，将来的版本，这个数值还会提高，不适合存储一些大型文件。但是 MongoDB 提供了 GridFS 文件系统，为大型文件的存储提供了解决方案。

八、 索引

MongoDB 的索引跟传统数据库的索引相似，一般的如果在传统数据库中需要建立索引的字段，在 MongoDB 中也可以建立索引。

MongoDB 中_id 字段默认已经建立了索引，这个索引特殊，并且不可删除，不过 Capped Collections 例外。

1. 建立索引

建立索引的函数：ensureIndex()

例子：

```
>$ db.persons.ensureIndex({name:1});
```

2．使用索引

a) 普通索引

```
>$ db.persons.find({name :'sam'});  // fast - uses index
>$ db. persons.find({age: 3});  // slow - has to check all
because 'age' isn't indexed
```

b) 嵌入式索引

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

c) 文档式索引

```
>$db.factories.insert( { name: "xyz", metro: { city: "New
York", state: "NY" } } );
```

```
>$db.factories.ensureIndex( { metro : 1 } );
```

d) 组合索引

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

e) 唯一索引

```
>$db.things.ensureIndex({firstname: 1, lastname: 1},
{unique: true});
```

当一个记录被插入到唯一性索引文档时，缺失的字段会以 null 为默认值被插入文档

例子：

```
>$db.things.ensureIndex({firstname: 1}, {unique: true});

>$db.things.save({lastname: "Smith"});

 //下面这个操作将会失败，因为 firstname 上有唯一性索引，值为 null

>$db.things.save({lastname: "Jones"});
```

3. 查看索引

```
> db.persons.getIndexes();
[
    {
        "name" : "_id_",
        "ns" : "test.persons",
        "key" : {
            "_id" : 1
        },
        "v" : 0
    },
    {
        "_id" : ObjectId("4de39060282677bdc555a83d"),
```

```
            "ns" : "test.persons",
            "key" : {
                "name" : 1
            },
            "name" : "name_1",
            "v" : 0
        }
    ]
```

4. 删除索引

a) 删除所有索引

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

b) 删除单个索引

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

c) 用运行命令的方式删除索引

```
// note: command was "deleteIndexes", not "dropIndexes",
before MongoDB v1.3.2
// remove index with key pattern {y:1} from collection foo
db.runCommand({dropIndexes:'foo', index : {y:1}})
// remove all indexes:
db.runCommand({dropIndexes:'foo', index : '*'})
```

5. 重建索引

```
db.myCollection.reIndex()
// same as:
db.runCommand( { reIndex : 'myCollection' } )
```

这个操作是个加锁操作，并且如果集合很大，这个操作会很耗时。

注：用 repair 命令修复数据库的时候，会重建索引

九、 主（Master）/从（slave）数据库同步

需要启动的两个 MongoDB 文档数据库，一个是以主模式启动，另一个属于从模式启动。因此，主服务器进程将创建一个 local.oplog，将通过这个“交易记录”同步到 Slave 服务器中。

1. 建立主/从服务器

主服务器：132.129.31.213:10111（A）

从服务器：132.129.31.213:10112（B）

启动 Master 数据库服务器：

```
$./mongod  –master -port=10111 -dbpath=/home/tsaip/mongodb/data/10111
-nohttpinterface &
```

启动 Slave 数据库服务器：5s 同步一次

```
$./mongod  –slave –source=132.129.31.213:10111 -port=10112
-dbpath=/home/tsaip/mongodb/data/10112 –slavedelay 5  -nohttpinterface &
```

测试同步结果：

```
//登录 master 数据库服务器
$ ./mongo -host 132.129.31.213 -port 10111
MongoDB shell version: 1.8.2-rc2
connecting to: 132.129.31.213:10111/test
> use test;
switched to db test
> db.user.insert({_id:1,name:'samlee',age:80});
> db.user.find();
{ "_id" : 1, "name" : "samlee", "age" : 80 }
>
//登录 slave 数据库服务器
$ ./mongo -host 132.129.31.213 -port 10112
MongoDB shell version: 1.8.2-rc2
connecting to: 132.129.31.213:10112/test
> use test;
switched to db test
> db.user.find();
{ "_id" : 1, "name" : "samlee", "age" : 80 }
```

```
>
```

数据同步成功!!

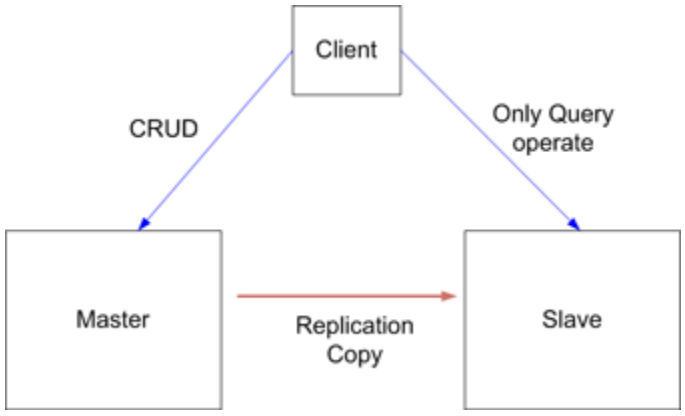
在 slave 数据库服务器上执行如下操作：

```
> db.user.insert({_id:2,name:'Jack',age:23});

not master

>
```

提示 not master ，所以 slave 服务器只可以执行读操作，不可以执行写操作，如下图所示：



2. 相关参数说明

Master

--master	master 模式
--oplogSize arg	size limit (in MB) for op log

Slave

--slave	slave 模式
--source arg	source 指定 master 位置
--only arg	单独指定备份某一 database
--slavedelay arg	指定与 Master 延迟时间(秒)
--autoresync	当 Slave 数据过时后自动重连

3. Slave 顶替 Master

如果上面的主服务器 A 宕了，此时需要用 B 机器来顶替 master 服务，步骤如下：

- 停止 B 进程(mongod)
- 删除 B 数据目录中的 local.*
- 以--master 模式启动 B

4. 切换 Master/Slave 角色

a) 假设已经具备主机 A 和从机 B，此时想切换它们的角色，步骤如下：（假设 A 是健康的）
b) 用 fsync 命令暂停 A 上的写操作，
c) 确定 B 是从机，关闭 B 上的服务
d) 清空 B 上的 local.*文件
e) 用-master 选项重启 B 服务
f) 在 B 上执行一次写操作，初始化 oplog，获得一个同步起始点
g) 关闭 B 服务，此时 B 已经有了信的 local.*文件
h) 关闭 A 服务，并且用 B 上新的 local.*文件来代替 A 上的 local.*文件（拷贝之前，记得先压缩，因为文件可能很大）
i) 用-master 选项重启 B 服务
j) 在平时的 slave 选项上加一个-fastsync 选项来重启 A 服务

如果 A 不是健康的，但是硬件是健康的，那么跳过上面的前两步，并且用 B 上所有文件去替换 A 上的文件，重启服务。

5. 更新主服务器位置

假设现有从机启动方式如下：

```
$ mongod --slave --source 132.129.31.213: 10000
```

此时如果想更换主机的位置，可以通过以下的步骤来完成：
重启 mongod 服务，不要加-slave 和 –source 选项：

```
$ mongod
```

启动 shell，执行如下操作：

```
> use local
switched to db local

>  db.sources.update({host : "132.129.31.213: 10000"},

{$set : {host : "132.129.31.213: 10001"}})
```

接着重启从机上的服务：

```
$ ./mongod --slave --source 132.129.31.213:10001
$ # or
$ ./mongod --slave
```

十、 MongoDB 分片和集群

1. 简单分片实例

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: mongod 实例，用于存储实际的数据块。

•Config Server: mongod 实例，存储了整个 Cluster Metadata，其中包括 chunk 信息。

•Route Server: mongos 实例，前端路由，客户端由此接入，且让整个集群看上去像单一进程数据库。

Route 转发请求到实际的目标服务进程，并将多个结果合并回传给客户端。Route 本身并不存储任何数据和状态，仅在启动时从 Config Server 获取信息。Config Server 上的任何变动都会传递给所有的 Route Process。

在实际使用中，为了获取高可用、高性能的集群方案，我们会将 Shard Server 部署成 Replica Sets，然后用 LVS 部署多个 Route。

我们先构建一个简单的 Sharding Cluster，以熟悉相关的配置。

1) 启动 Shard Server

```
//server_1
$ ./mongod -shardsvr -port 10000 -dbpath /home/tsaip/mongodb/data/10000 -fork -logpath /home/tsaip/mongodb/data/logs/null
all output going to: /home/tsaip/mongodb/data/logs/null
forked process: 9347
//server_2
$ ./mongod -shardsvr -port 10011 -dbpath /home/tsaip/mongodb/data/10011 -fork -logpath /home/tsaip/mongodb/data/logs/null
all output going to: /home/tsaip/mongodb/data/logs/null
forked process: 9383
```

2) 启动 Config Server

```
$ ./mongod -configsvr -port 20000 -dbpath /home/tsaip/mongodb/data/config -fork -logpath /home/tsaip/mongodb/data/logs/null
all output going to: /home/tsaip/mongodb/data/logs/null
forked process: 9399
```

3) 启动 Route Process

```
./mongos -configdb 132.129.31.213:20000 -fork -logpath /home/tsaip/mongodb/data/logs/null
all output going to: /home/tsaip/mongodb/data/logs/null
forked process: 9431
```

注：可以用 -chunkSize 参数指定分块大小

4) 开始配置

相关命令：

- addshard：添加 Shard Server，相关的命令还有 listshards 和 removeshard。
- enablesharding：用于设置可以被分布存储的数据库。
- shardcollection：用于设置具体被切块的集合名称，且必须指定 Shard Key，系统会自动创建索引。

注：Sharded Collection 只能有一个 unique index，且必须是 shard key

```
$ ./mongo
MongoDB shell version: 1.8.2-rc2
connecting to: test
> use admin;
switched to db admin
> db.runCommand({addshard:'132.129.31.213:10000'});
{ "shardAdded" : "shard0000", "ok" : 1 }
> db.runCommand({addshard:'132.129.31.213:10011'});
{ "shardAdded" : "shard0001", "ok" : 1 }
> db.runCommand({enablesharding:'test'});
{ "ok" : 1 }
> db.runCommand({shardcollection:'test.user',key:{_id:1}});
```

```
{ "collectionsharded" : "test.user", "ok" : 1 }
>
```

5) 管理命令

✧ **listshards:列出所有的 Shard Server**

```
> db.runCommand({listshards:1});
{
    "shards" : [
        {
            "_id" : "shard0000",
            "host" : "132.129.31.213:10000"
        },
        {
            "_id" : "shard0001",
            "host" : "132.129.31.213:10011"
        }
    ],
    "ok" : 1
}
```

✧ **移除 shard**

```
> db.runCommand( { removeshard : "localhost:10000" } );
{ msg : "draining started successfully" , state: "started" ,
shard :"localhost:10000" , ok : 1
```

✧ **printShardingStatus:查看 Sharding 信息**

```
> printShardingStatus();
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
        { "_id" : "shard0000", "host" : "132.129.31.213:10000" }
        { "_id" : "shard0001", "host" : "132.129.31.213:10011" }
  databases:
    { "_id" : "admin", "partitioned" : false, "primary" : "config" }
    { "_id" : "test", "partitioned" : true, "primary" : "shard0000" }
          test.user chunks:
                  shard0000    1
              { "_id" : { $minKey : 1 } } -->> { "_id" : { $maxKey : 1 } } on : shard0000
{ "t" : 1000, "i" : 0 }
>
```

✧ **db.<collection_name>.stats()：查看具体的 Shard 存储信息**

```
> use test;
switched to db test
> db.user.stats();
{
    "sharded" : true,
    "ns" : "test.user",
    "count" : 0,
    "size" : 0,
    "avgObjSize" : NaN,
    "storageSize" : 8192,
    "nindexes" : 1,
    "nchunks" : 1,
    "shards" : {
        "shard0000" : {
            "ns" : "test.user",
            "count" : 0,
            "size" : 0,
            "storageSize" : 8192,
            "numExtents" : 1,
            "nindexes" : 1,
            "lastExtentSize" : 8192,
            "paddingFactor" : 1,
            "flags" : 1,
            "totalIndexSize" : 8192,
            "indexSizes" : {
                "_id_" : 8192
            },
            "ok" : 1
        }
    },
    "ok" : 1
}
>
```

✧ **isdbgrid：用来确认当前是否是 Sharding Cluster**

```
> db.runCommand({isdbgrid:1});
{ "isdbgrid" : 1, "hostname" : "ossesbwas", "ok" : 1 }
> db.runCommand({ismaster:1});
{
    "ismaster" : true,
    "msg" : "isdbgrid",
    "maxBsonObjectSize" : 16777216,
    "ok" : 1
}
>
```

2. 高级分片实例

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(high reliability)、高可用(high availability)还有些距离。
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(two-phase commit)。
- Route: 配合 LVS，实现负载平衡，提高接入性能(high performance)。

■ 配置一个 Replica Sets + Sharding 环境

1. 创建所有需要的数据库目录

```
$ mkdir -p /home/tsaip/mongodb/data/10001
$ mkdir -p /home/tsaip/mongodb/data/10002
$ mkdir -p /home/tsaip/mongodb/data/10003

$ mkdir -p /home/tsaip/mongodb/data/10011
$ mkdir -p /home/tsaip/mongodb/data/10012
$ mkdir -p /home/tsaip/mongodb/data/10013

$ mkdir -p /home/tsaip/mongodb/data/config1
$ mkdir -p /home/tsaip/mongodb/data/config2
$ mkdir -p /home/tsaip/mongodb/data/config3

$ mkdir -p /home/tsaip/mongodb/data/logs
```

2. 配置 Shard Replica Sets

//第一组 sets

```
//10001
$./mongod --shardsvr --fork --logpath /home/tsaip/mongodb/data/logs/null --dbpath
/home/tsaip/mongodb/data/10001 --port 10001 --nohttpinterface --replSet set1
forked process: 9704
all output going to: /home/tsaip/mongodb/data/logs/null
//10002
$./mongod --shardsvr --fork --logpath /home/tsaip/mongodb/data/logs/null --dbpath
/home/tsaip/mongodb/data/10002 --port 10002 --nohttpinterface --replSet set1
all output going to: /home/tsaip/mongodb/data/logs/null
forked process: 9718
//10003
$./mongod --shardsvr --fork --logpath /home/tsaip/mongodb/data/logs/null --dbpath
/home/tsaip/mongodb/data/10003 --port 10003 --nohttpinterface --replSet set1
forked process: 9732
all output going to: /home/tsaip/mongodb/data/logs/null
```

```
$ ./mongo -port 10001
MongoDB shell version: 1.8.2-rc2
connecting to: 127.0.0.1:10001/test
>cfg= {_id:'set1',
members:[
{_id:1,host:'132.129.31.213:10001'},
{_id:2,host:'132.129.31.213:10002'},
{_id:3,host:'132.129.31.213:10003'}]
};
{
      "_id" : "set1",
      "members" : [
            {
                  "_id" : 1,
                  "host" : "132.129.31.213:10001"
            },
            {
                  "_id" : 2,
                  "host" : "132.129.31.213:10002"
            },
            {
                  "_id" : 3,
                  "host" : "132.129.31.213:10003"
            }
```

```
        ]
}
> rs.initiate(cfg);
{
        "info" : "Config now saved locally.    Should come online in about a minute.",
        "ok" : 1
}
> rs.status();
{
        "set" : "set1",
        "date" : ISODate("2011-05-31T04:28:50Z"),
        "myState" : 1,
        "members" : [
                {
                        "_id" : 1,
                        "name" : "132.129.31.213:10001",
                        "health" : 1,
                        "state" : 1,
                        "stateStr" : "PRIMARY",
                        "optime" : {
                                "t" : 1306816113000,
                                "i" : 1
                        },
                        "optimeDate" : ISODate("2011-05-31T04:28:33Z"),
                        "self" : true
                },
                {
                        "_id" : 2,
                        "name" : "132.129.31.213:10002",
                        "health" : 1,
                        "state" : 3,
                        "stateStr" : "RECOVERING",
                        "uptime" : 9,
                        "optime" : {
                                "t" : 0,
                                "i" : 0
                        },
                        "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
                        "lastHeartbeat" : ISODate("2011-05-31T04:28:49Z")
                },
                {
                        "_id" : 3,
                        "name" : "132.129.31.213:10003",
                        "health" : 0,
```

```
                "state" : 6,
                "stateStr" : "(not reachable/healthy)",
                "uptime" : 0,
                "optime" : {
                        "t" : 0,
                        "i" : 0
                },
                "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
                "lastHeartbeat" : ISODate("2011-05-31T04:28:49Z"),
                "errmsg" : "still initializing"
        }
    ],
    "ok" : 1
}
set1:PRIMARY>
```

//第二组 sets

```
//10011
[tsaip@ossesbwas bin]$ ./mongod --shardsvr --fork --logpath /home/tsaip/mongodb/data/logs/null --dbpath /home/tsaip/mongodb/data/10011 --port 10011 --nohttpinterface --replSet set2
all output going to: /home/tsaip/mongodb/data/logs/null
forked process: 10326
//10012
[tsaip@ossesbwas bin]$ ./mongod --shardsvr --fork --logpath /home/tsaip/mongodb/data/logs/null --dbpath /home/tsaip/mongodb/data/10012 --port 10012 --nohttpinterface --replSet set2
all output going to: /home/tsaip/mongodb/data/logs/null
forked process: 10335
//10013
[tsaip@ossesbwas bin]$ ./mongod --shardsvr --fork --logpath /home/tsaip/mongodb/data/logs/null --dbpath /home/tsaip/mongodb/data/10013 --port 10013 --nohttpinterface --replSet set2
forked process: 10347
```

```
[tsaip@ossesbwas bin]$ ./mongo -port 10011
MongoDB shell version: 1.8.2-rc2
connecting to: 127.0.0.1:10011/test
> cfg = {_id:'set2',members:[{_id:0,host:'132.129.31.213:10011'},{_id:1,host:'132.129.31.213:10012'},{_id:2,host:'132.129.31.213:10013'}]};
{
    "_id" : "set2",
    "members" : [
        {
```

```
                "_id" : 0,
                "host" : "132.129.31.213:10011"
            },
            {
                "_id" : 1,
                "host" : "132.129.31.213:10012"
            },
            {
                "_id" : 2,
                "host" : "132.129.31.213:10013"
            }
        ]
}
> rs.initiate(cfg);
{
        "info" : "Config now saved locally.    Should come online in about a minute.",
        "ok" : 1
}
set2:STARTUP2>
set2:PRIMARY>
set2:PRIMARY> rs.status();
{
        "set" : "set2",
        "date" : ISODate("2011-05-31T06:32:07Z"),
        "myState" : 1,
        "members" : [
            {
                "_id" : 0,
                "name" : "132.129.31.213:10011",
                "health" : 1,
                "state" : 1,
                "stateStr" : "PRIMARY",
                "optime" : {
                    "t" : 1306823510000,
                    "i" : 1
                },
                "optimeDate" : ISODate("2011-05-31T06:31:50Z"),
                "self" : true
            },
            {
                "_id" : 1,
                "name" : "132.129.31.213:10012",
                "health" : 1,
                "state" : 3,
```

```
                "stateStr" : "RECOVERING",
                "uptime" : 11,
                "optime" : {
                        "t" : 0,
                        "i" : 0
                },
                "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
                "lastHeartbeat" : ISODate("2011-05-31T06:32:07Z"),
                "errmsg" : "."
        },
        {
                "_id" : 2,
                "name" : "132.129.31.213:10013",
                "health" : 1,
                "state" : 3,
                "stateStr" : "RECOVERING",
                "uptime" : 3,
                "optime" : {
                        "t" : 0,
                        "i" : 0
                },
                "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
                "lastHeartbeat" : ISODate("2011-05-31T06:32:06Z")
        }
    ],
    "ok" : 1
}
```

3. 启动 Config Server

可以只使用 1 个 Config Server，但 3 个理论上更有保障性。

```
[tsaip@ossesbwas bin]$ ./mongod --configsvr --fork --logpath /home/tsaip/mongodb/data/logs/null --dbpath /home/tsaip/mongodb/data/config1 --port 20000 --nohttpinterface --fork --logpath /home/tsaip/mongodb/data/logs/null --dbpath /home/tsaip/mongodb/data/config1 --port 20000
all output going to: /home/tsaip/mongodb/data/logs/null
forked process: 10460
[tsaip@ossesbwas bin]$ ./mongod --configsvr --fork --logpath /home/tsaip/mongodb/data/logs/null --dbpath /home/tsaip/mongodb/data/config2 --port 20001 --nohttpinterface
all output going to: /home/tsaip/mongodb/data/logs/null
forked process: 10467
```

```
[tsaip@ossesbwas bin]$ ./mongod --configsvr --fork --logpath /home/tsaip/mongodb/data/logs/null --dbpath /home/tsaip/mongodb/data/config3 --port 20002 --nohttpinterface
all output going to: /home/tsaip/mongodb/data/logs/null
forked process: 10476
```

注：这个不是 Replica Sets，不需要 --replSet 参数。

4. 启动 Route Server

```
[tsaip@ossesbwas bin]$ ./mongos --fork --logpath /home/tsaip/mongodb/data/logs/null --configdb "132.129.31.213:20000,132.129.31.213:20001,132.129.31.213:20002"
all output going to: /home/tsaip/mongodb/data/logs/null
forked process: 10512

[tsaip@ossesbwas bin]$ ps aux | grep mongos | grep -v grep
tsaip     10497  0.0  0.0 155988  2092 ?        Sl   14:40   0:00 ./mongos --fork --logpath /home/tsaip/mongodb/data/logs/null --configdb 132.129.31.213:20000,132.129.31.213:20001,132.129.31.213:20002
```

注：注意 --configdb 参数

5. 开始配置 Sharding

```
[tsaip@ossesbwas bin]$ ./mongo
MongoDB shell version: 1.8.2-rc2
connecting to: test
> use admin;
switched to db admin
>
db.runCommand({addshard:'set1/132.129.31.213:10001,132.129.31.213:10002,132.129.31.213:10003'});
{ "shardAdded" : "set1", "ok" : 1 }
>
db.runCommand({ addshard:'set2/132.129.31.213:10011,132.129.31.213:10012,132.129.31.213:10013' })
{ "shardAdded" : "set2", "ok" : 1 }
> db.runCommand({ enablesharding:'test' });
{ "ok" : 1 }
> db.runCommand({ shardcollection:'test.user', key:{_id:1} });
{ "collectionsharded" : "test.user", "ok" : 1 }
> db.runCommand({ listshards:1 });
{
        "shards" : [
```

```
        {
                "_id" : "set1",
                "host" :
"set1/132.129.31.213:10001,132.129.31.213:10002,132.129.31.213:10003"
        },
        {
                "_id" : "set2",
                "host" :
"set2/132.129.31.213:10011,132.129.31.213:10012,132.129.31.213:10013"
        }
    ],
    "ok" : 1
}
> printShardingStatus();
--- Sharding Status ---
  sharding version: { "_id" : 1, "version" : 3 }
  shards:
      {
    "_id" : "set1",
    "host" : "set1/132.129.31.213:10001,132.129.31.213:10002,132.129.31.213:10003"
}
      {
    "_id" : "set2",
    "host" : "set2/132.129.31.213:10011,132.129.31.213:10012,132.129.31.213:10013"
}
  databases:
    { "_id" : "admin", "partitioned" : false, "primary" : "config" }
    { "_id" : "test", "partitioned" : true, "primary" : "set1" }
        test.user chunks:
                set1 1
            { "_id" : { $minKey : 1 } } -->> { "_id" : { $maxKey : 1 } } on : set1 { "t" : 1000, "i" : 0 }

>
```

注：addshard 添加 Replica Sets 的格式。

至此配置结束，测试下：

```
> use test;
switched to db test
> db.user.insert({_id:10,name:'tom',age:20});
> db.user.insert({_id:11,name:'jim',age:20});
> db.user.insert({_id:12,name:'smith',age:20});
> db.user.find();
{ "_id" : 10, "name" : "tom", "age" : 20 }
{ "_id" : 11, "name" : "jim", "age" : 20 }
{ "_id" : 12, "name" : "smith", "age" : 20 }
```

十一、 数据库基本操作：增查删改

1. Insert

Mongodb 是面向文档存储的数据库，文档结构形式叫 BSON（类似 JSON）。
实例：

```
//定义文档
>doc = {
    "_id" : 1,
    "author" : "sam",
    "title" : "i love you",
    "text" : "this is a test",
    "tags" : [
        "love",
        "test"
    ],
    "comments" : [
        {
            "author" : "jim",
            "comment" : "yes"
        },
        {
            "author" : "tom",
            "comment" : "no"
        }
    ]
}
//插入文档
> db.posts.insert(doc);
//查找文档
> db.posts.find({'comments.author':'jim'});
{ "_id" : 1, "author" : "sam", "title" : "i love you", "text" : "this is a test", "tags" :
[ "love", "test" ], "comments" : [
    {
        "author" : "jim",
        "comment" : "yes"
    },
    {
        "author" : "tom",
        "comment" : "no"
    }
] }
```

2. Query

Mongodb 最大的功能之一就是它支持动态查询，就跟传统的关系型数据库查询一样，但是它的查询来的更灵活。

1) Query Expression Objects：查询表达式对象

查询表达式文档也是一个 BSON 结构的文档，例如，我们可以用下面的查询语句来查询集合中的所有记录：

```
db.users.find({})
```

这里，表达式对象是一个空文档，在查询的时候去去匹配所有的记录。
再看：

```
db.users.find({'last_name': 'Smith'})
```

这里，我们将会查询出所有“last_name”属性值为“Smith”的文档记录。

2) 查询选项

除了查询表达式意外，Mongodb 还支持一些额外的参数选项。例如，我们可能仅仅只想返回某些特定的字段值：

```
//返回除了 age 字段外的所有字段
> db.user.find({},{age:0});
//返回 tags=tennis 除了 comments 的所有列
db.posts.find( { tags : 'tennis' }, { comments : 0 } );
//返回 userid=16 的 name 字段
> db.user.find({userid:16},{name:1});
{ "_id" : 16, "name" : "user16" }
//返回 x=john 的所有 z 字段
db.things.find( { x : "john" }, { z : 1 } );
```

注： _id 字段始终都会被返回，哪怕没有明确指定

3) 条件表达式

1) <, <=, >, >=

语法：

```
// 大于: field > value
db.collection.find({ "field" : { $gt: value } } );
```

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

实例：

```
//大于
> db.user.find({_id:{$gt:20}}).limit(8);
{ "_id" : 21, "name" : "user21", "userid" : 21, "age" : 30 }
{ "_id" : 22, "name" : "user22", "userid" : 22, "age" : 30 }
{ "_id" : 23, "name" : "user23", "userid" : 23, "age" : 30 }
{ "_id" : 24, "name" : "user24", "userid" : 24, "age" : 30 }
{ "_id" : 25, "name" : "user25", "userid" : 25, "age" : 30 }
{ "_id" : 26, "name" : "user26", "userid" : 26, "age" : 30 }
{ "_id" : 27, "name" : "user27", "userid" : 27, "age" : 30 }
{ "_id" : 28, "name" : "user28", "userid" : 28, "age" : 30 }
//大于等于
> db.user.find({_id:{$gte:20}}).limit(8);
{ "_id" : 20, "name" : "user20", "userid" : 20, "age" : 30 }
{ "_id" : 21, "name" : "user21", "userid" : 21, "age" : 30 }
{ "_id" : 22, "name" : "user22", "userid" : 22, "age" : 30 }
{ "_id" : 23, "name" : "user23", "userid" : 23, "age" : 30 }
{ "_id" : 24, "name" : "user24", "userid" : 24, "age" : 30 }
{ "_id" : 25, "name" : "user25", "userid" : 25, "age" : 30 }
{ "_id" : 26, "name" : "user26", "userid" : 26, "age" : 30 }
{ "_id" : 27, "name" : "user27", "userid" : 27, "age" : 30 }
//小于
> db.user.find({_id:{$lt:8}}).limit(8);
{ "_id" : 0, "name" : "user0", "userid" : 0, "age" : 30 }
{ "_id" : 1, "name" : "user1", "userid" : 1, "age" : 30 }
{ "_id" : 2, "name" : "user2", "userid" : 2, "age" : 30 }
{ "_id" : 3, "name" : "user3", "userid" : 3, "age" : 30 }
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 30 }
{ "_id" : 5, "name" : "user5", "userid" : 5, "age" : 30 }
{ "_id" : 6, "name" : "user6", "userid" : 6, "age" : 30 }
{ "_id" : 7, "name" : "user7", "userid" : 7, "age" : 30 }
//小于等于
> db.user.find({_id:{$lte:8}}).limit(8);
{ "_id" : 0, "name" : "user0", "userid" : 0, "age" : 30 }
{ "_id" : 1, "name" : "user1", "userid" : 1, "age" : 30 }
```

```
{ "_id" : 2, "name" : "user2", "userid" : 2, "age" : 30 }
{ "_id" : 3, "name" : "user3", "userid" : 3, "age" : 30 }
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 30 }
{ "_id" : 5, "name" : "user5", "userid" : 5, "age" : 30 }
{ "_id" : 6, "name" : "user6", "userid" : 6, "age" : 30 }
{ "_id" : 7, "name" : "user7", "userid" : 7, "age" : 30 }
//区间查询
> db.user.find({_id:{$gt:5,$lte:10}});
{ "_id" : 6, "name" : "user6", "userid" : 6, "age" : 30 }
{ "_id" : 7, "name" : "user7", "userid" : 7, "age" : 30 }
{ "_id" : 8, "name" : "user8", "userid" : 8, "age" : 30 }
{ "_id" : 9, "name" : "user9", "userid" : 9, "age" : 30 }
{ "_id" : 10, "name" : "user10", "userid" : 10, "age" : 30 }
```

2) $all

$all 操作类似$in 操作，但是不同的是，$all 操作要求数组里面的值全部被包含在返回的记录里面，如：

```
> use test;
switched to db test
> db.things.insert({a:[1,2,3]});
> db.things.find();
{ "_id" : ObjectId("4de73360059e7f4bdf907cfe"), "a" : [ 1, 2, 3 ] }
> db.things.find({a:{$all:[2,3]}});
{ "_id" : ObjectId("4de73360059e7f4bdf907cfe"), "a" : [ 1, 2, 3 ] }
> db.things.find({a:{$all:[1,2,3]}});
{ "_id" : ObjectId("4de73360059e7f4bdf907cfe"), "a" : [ 1, 2, 3 ] }
> db.things.find({a:{$all:[1]}});
{ "_id" : ObjectId("4de73360059e7f4bdf907cfe"), "a" : [ 1, 2, 3 ] }
> db.things.find({a:{$all:[1,2,3,4]}});
>
```

3) $exists

$exists 操作检查一个字段是否存在，如：

```
> for(var i=0;i<1000;i++) db.user.save({_id:i,name:'user'+i,userid:i,age:20});
//包含_id，索引
> db.user.find({_id:{$exists:true}}).limit(5);
{ "_id" : 0, "name" : "user0", "userid" : 0, "age" : 20 }
{ "_id" : 1, "name" : "user1", "userid" : 1, "age" : 20 }
{ "_id" : 2, "name" : "user2", "userid" : 2, "age" : 20 }
{ "_id" : 3, "name" : "user3", "userid" : 3, "age" : 20 }
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 20 }
```

```
//包含 userid
> db.user.find({userid:{$exists:true}}).limit(5);
{ "_id" : 0, "name" : "user0", "userid" : 0, "age" : 20 }
{ "_id" : 1, "name" : "user1", "userid" : 1, "age" : 20 }
{ "_id" : 2, "name" : "user2", "userid" : 2, "age" : 20 }
{ "_id" : 3, "name" : "user3", "userid" : 3, "age" : 20 }
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 20 }
//不包含 sex 字段
> db.user.find({sex:{$exists:true}}).limit(5);
>
```

注：1.8 版本之前，$exists 操作不可用于索引上面

4) $mod

$mod 操作可以让我们简单的进行取模操作，而不需要用到 where 子句，如：

```
//where 子句
> db.user.find("this._id%10==1").limit(5);
{ "_id" : 1, "name" : "user1", "userid" : 1, "age" : 20 }
{ "_id" : 11, "name" : "user11", "userid" : 11, "age" : 20 }
{ "_id" : 21, "name" : "user21", "userid" : 21, "age" : 20 }
{ "_id" : 31, "name" : "user31", "userid" : 31, "age" : 20 }
{ "_id" : 41, "name" : "user41", "userid" : 41, "age" : 20 }
//$mod 操作
> db.user.find({_id:{$mod:[10,1]}}).limit(5);
{ "_id" : 1, "name" : "user1", "userid" : 1, "age" : 20 }
{ "_id" : 11, "name" : "user11", "userid" : 11, "age" : 20 }
{ "_id" : 21, "name" : "user21", "userid" : 21, "age" : 20 }
{ "_id" : 31, "name" : "user31", "userid" : 31, "age" : 20 }
{ "_id" : 41, "name" : "user41", "userid" : 41, "age" : 20 }
>
```

5) $ne

$ne 意思是 not equal，不等于，不用多说，看例子：

```
> db.user.find().limit(5);
{ "_id" : 0, "name" : "user0", "userid" : 0, "age" : 20 }
{ "_id" : 1, "name" : "user1", "userid" : 1, "age" : 20 }
{ "_id" : 2, "name" : "user2", "userid" : 2, "age" : 20 }
{ "_id" : 3, "name" : "user3", "userid" : 3, "age" : 20 }
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 20 }
> db.user.find({_id:{$ne:0}}).limit(5);
{ "_id" : 1, "name" : "user1", "userid" : 1, "age" : 20 }
```

```
{ "_id" : 2, "name" : "user2", "userid" : 2, "age" : 20 }
{ "_id" : 3, "name" : "user3", "userid" : 3, "age" : 20 }
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 20 }
{ "_id" : 5, "name" : "user5", "userid" : 5, "age" : 20 }
>
```

6) $in

$in 操作类似于传统关系数据库中的 IN，看例子：

```
//数据库中有所有数组对应的记录
> db.user.find({_id:{$in:[2,3,4,5,6]}}).limit(5);
{ "_id" : 2, "name" : "user2", "userid" : 2, "age" : 20 }
{ "_id" : 3, "name" : "user3", "userid" : 3, "age" : 20 }
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 20 }
{ "_id" : 5, "name" : "user5", "userid" : 5, "age" : 20 }
{ "_id" : 6, "name" : "user6", "userid" : 6, "age" : 20 }
//因为数据库中没有_id=1111 的记录
> db.user.find({_id:{$in:[2,3,4,5,1111]}}).limit(5);
{ "_id" : 2, "name" : "user2", "userid" : 2, "age" : 20 }
{ "_id" : 3, "name" : "user3", "userid" : 3, "age" : 20 }
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 20 }
{ "_id" : 5, "name" : "user5", "userid" : 5, "age" : 20 }
>
```

7) $nin

$nin 跟$in 操作相反，看例子：

```
//扣掉_id=1/2/3/4 的记录
> db.user.find({_id:{$nin:[1,2,3,4]}}).limit(5);
{ "_id" : 0, "name" : "user0", "userid" : 0, "age" : 20 }
{ "_id" : 5, "name" : "user5", "userid" : 5, "age" : 20 }
{ "_id" : 6, "name" : "user6", "userid" : 6, "age" : 20 }
{ "_id" : 7, "name" : "user7", "userid" : 7, "age" : 20 }
{ "_id" : 8, "name" : "user8", "userid" : 8, "age" : 20 }
>
```

8) $nor

$nor 跟$or 相反，不好解释，看例子：

```
> db.user.find({$nor:[{_id:2},{name:'user3'},{userid:4}]}).limit(5);
{ "_id" : 0, "name" : "user0", "userid" : 0, "age" : 20 }
{ "_id" : 1, "name" : "user1", "userid" : 1, "age" : 20 }
```

```
{ "_id" : 5, "name" : "user5", "userid" : 5, "age" : 20 }
{ "_id" : 6, "name" : "user6", "userid" : 6, "age" : 20 }
{ "_id" : 7, "name" : "user7", "userid" : 7, "age" : 20 }
>
```

可以看到，_id=2,name=user3 和 userid=4 的记录都被过滤了

9) $or

```
> db.user.find({$or:[{_id:2},{name:'user3'},{userid:4}]}).limit(5);
{ "_id" : 2, "name" : "user2", "userid" : 2, "age" : 20 }
{ "_id" : 3, "name" : "user3", "userid" : 3, "age" : 20 }
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 20 }
>
```

可以看到，只有_id=2,name=user3 和 userid=4 三条记录都选择了

10) $size

$size 操作将会查询数组长度等于输入参数的数组，例子：

```
> db.things.find();
{ "_id" : ObjectId("4de73360059e7f4bdf907cfe"), "a" : [ 1, 2, 3 ] }
> db.things.find({a:{$size:3}});
{ "_id" : ObjectId("4de73360059e7f4bdf907cfe"), "a" : [ 1, 2, 3 ] }
> db.things.find({a:{$size:2}});
> db.things.find({a:{$size:1}});
```

11) $where

例子：

```
db.mycollection.find( { $where : function() { return this.a
== 3 || this.b == 4; } } );
//同上效果
db.mycollection.find( function() { return this.a == 3 ||
this.b == 4; } );
```

12) $type

$type 将会根据字段的 BSON 类型来检索数据，例如：

```
//返回 a 是字符串的记录
db.things.find( { a : { $type : 2 } } ); // matches if a is
a string
```

```
//返回 a 是 int 类型的记录
db.things.find( { a : { $type : 16 } } ); // matches if a is an int
```

下表是 BSON 数据类型表格映射：

Type Name	Type Number
Double	1
String	2
Object	3
Array	4
Binary data	5
Object id	7
Boolean	8
Date	9
Null	10
Regular expression	11
JavaScript code	13
Symbol	14
JavaScript code with scope	15
32-bit integer	16
Timestamp	17
64-bit integer	18
Min key	255
Max key	127

4) 正则表达式

Mongodb 同样支持正则表达式进行检索，如：

```
//检索 name 属性是以 u 开头，4 结尾的所有用户
> db.user.find({name:/u.*4$/i}).limit(5);
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 20 }
{ "_id" : 14, "name" : "user14", "userid" : 14, "age" : 20 }
```

```
{ "_id" : 24, "name" : "user24", "userid" : 24, "age" : 20 }
{ "_id" : 34, "name" : "user34", "userid" : 34, "age" : 20 }
{ "_id" : 44, "name" : "user44", "userid" : 44, "age" : 20 }
>
//同样效果的查询语句
> db.user.find({name:{$regex:'u.*4$',$options:'i'}}).limit(5);
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 20 }
{ "_id" : 14, "name" : "user14", "userid" : 14, "age" : 20 }
{ "_id" : 24, "name" : "user24", "userid" : 24, "age" : 20 }
{ "_id" : 34, "name" : "user34", "userid" : 34, "age" : 20 }
{ "_id" : 44, "name" : "user44", "userid" : 44, "age" : 20 }
>
```

配合其他操作一起使用：

```
> db.user.find({name:{$regex:'u.*4$',$options:'i',$nin:['user4']}}).limit(5);
{ "_id" : 14, "name" : "user14", "userid" : 14, "age" : 20 }
{ "_id" : 24, "name" : "user24", "userid" : 24, "age" : 20 }
{ "_id" : 34, "name" : "user34", "userid" : 34, "age" : 20 }
{ "_id" : 44, "name" : "user44", "userid" : 44, "age" : 20 }
{ "_id" : 54, "name" : "user54", "userid" : 54, "age" : 20 }
> db.user.find({name:{$regex:'u.*4$',$options:'i',$in:['user4']}}).limit(5);
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 20 }
>
```

注意：/^a/；/^a.*/； /^a.*$/这三个表达式最后的效果一样，但是后两种查询效率比第一种要低很多，因为后两种表达式会执行扫描整个字符串，然而第一种扫描到第一个字符就停止了

选项：

-i:忽略大小写

-m:起始符^,结束符$对于每一个新行都起作用

-x:忽略空白字符

-s:这个选项从 1.9 版本后才支持，加上它就可以让“.”表示所有字符了，包括换行符，例如 /a.*b/不匹配"apple\nbanana"，但是 /a.*b/s 可以

5) 排序

Mongodb 支持排序，例如，按照 last_name 属性进行升序排序返回所有文档：

```
//1 表示升序，-1 表示降序

db.users.find({}).sort({last_name: 1});
```

6) Group

语法：

```
db.coll.group(
    { cond: {filed：conditions}
    , key: {filed: true}
    , initial: {count: 0, total_time:0}
    , reduce: function(doc, out){ }
    , finalize: function(out){}
    } );
```

参数说明：

Key：对那个字段进行 Group

Cond：查询条件

Initial：初始化 group 计数器

Reduce：通常做统计操作

Finalize：通常都统计结果进行进一步操作，例如求平均值

Keyf：用一个函数来返回一个替代 KEY 的值

例子：

```
db.test.group(
   { cond: {"invoked_at.d": {$gte: "2009-11", $lt:
"2009-12"}}
   , key: {http_action: true}
   , initial: {count: 0, total_time:0}
   , reduce: function(doc, out){ out.count++;
out.total_time+=doc.response_time }
   , finalize: function(out){ out.avg_time = out.total_time
/ out.count }
   } );

[
  {
    "http_action" : "GET /display/DOCS/Aggregation",
    "count" : 1,
    "total_time" : 0.05,
    "avg_time" : 0.05
  }
]
```

7) Distinct

类似于关系数据库中的 Distinct，如：

```
//
> db.addresses.insert({"zip-code": 10010})
> db.addresses.insert({"zip-code": 10010})
> db.addresses.insert({"zip-code": 99701})

> // shell helper:
> db.addresses.distinct("zip-code");
[ 10010, 99701 ]

> // running as a command manually:
> db.runCommand( { distinct: 'addresses', key: 'zip-code' } )
{ "values" : [ 10010, 99701 ], "ok"
//
> db.comments.save({"user": {"points": 25}})
> db.comments.save({"user": {"points": 31}})
> db.comments.save({"user": {"points": 25}})

> db.comments.distinct("user.points");
[ 25, 31 ]
```

8) 分页查询

Mongodb 支持 skip 和 limit 命令来进行分页查询，例如：

```
//跳过前 10 条记录
> db.user.find().skip(10);
//每页返回 8 条记录
> db.user.find().limit(8);
//跳过前 20 条记录，并且每页返回 10 条记录
>db.user.find().skip(20).limit(8);
//下面这个语句跟上一条一样，只是表达不够清晰
> db.user.find({},{},8,20);
```

9) $elemMatch

```
> t.find( { x : { $elemMatch : { a : 1, b : { $gt : 1 } } } } )

{ "_id" : ObjectId("4b5783300334000000000aa9"),
  "x" : [ { "a" : 1, "b" : 3 }, 7, { "b" : 99 }, { "a" : 11 } ]
}
//同样效果
> t.find( { "x.a" : 1, "x.b" : { $gt : 1 } } )
```

10) slaveOk

当我们在 replica set 进行检索操作时，默认的路由会选择 master 机器，当我们需要针对任意的从机进行查询的时候，我们需要开启 slaveOk 选项。当我们在没有开启 salveOk 选项的时候，如果进行此操作会报如下错：

```
*** not master and slaveok=false
```

所以我们要进行如下操作：

```
rs.slaveOk(); // enable querying a secondary
db.users.find(...)
```

11) Cursors 游标及 Cursor Methods

a) 遍历游标

类似传统的关系型数据库，Mongodb 也有游标的概念。可以利用游标对查询结果进行遍历，如：

```
//利用游标遍历检索结果
> var cur = db.user.find().skip(10).limit(8);
> cur.forEach(function(x){print(tojson(x))});
{ "_id" : 10, "name" : "user10", "userid" : 10, "age" : 30 }
{ "_id" : 11, "name" : "user11", "userid" : 11, "age" : 30 }
{ "_id" : 12, "name" : "user12", "userid" : 12, "age" : 30 }
{ "_id" : 13, "name" : "user13", "userid" : 13, "age" : 30 }
{ "_id" : 14, "name" : "user14", "userid" : 14, "age" : 30 }
{ "_id" : 15, "name" : "user15", "userid" : 15, "age" : 30 }
{ "_id" : 16, "name" : "user16", "userid" : 16, "age" : 30 }
{ "_id" : 17, "name" : "user17", "userid" : 17, "age" : 30 }
>
```

b) skip()

skip()方法指定查询记录从第几条开始返回，如，我要从第 10 个用户开始返回：

```
> db.user.find().skip(10);
{ "_id" : 10, "name" : "user10", "userid" : 10, "age" : 30 }
{ "_id" : 11, "name" : "user11", "userid" : 11, "age" : 30 }
{ "_id" : 12, "name" : "user12", "userid" : 12, "age" : 30 }
{ "_id" : 13, "name" : "user13", "userid" : 13, "age" : 30 }
{ "_id" : 14, "name" : "user14", "userid" : 14, "age" : 30 }
{ "_id" : 15, "name" : "user15", "userid" : 15, "age" : 30 }
{ "_id" : 16, "name" : "user16", "userid" : 16, "age" : 30 }
{ "_id" : 17, "name" : "user17", "userid" : 17, "age" : 30 }
{ "_id" : 18, "name" : "user18", "userid" : 18, "age" : 30 }
{ "_id" : 19, "name" : "user19", "userid" : 19, "age" : 30 }
{ "_id" : 20, "name" : "user20", "userid" : 20, "age" : 30 }
{ "_id" : 21, "name" : "user21", "userid" : 21, "age" : 30 }
{ "_id" : 22, "name" : "user22", "userid" : 22, "age" : 30 }
{ "_id" : 23, "name" : "user23", "userid" : 23, "age" : 30 }
{ "_id" : 24, "name" : "user24", "userid" : 24, "age" : 30 }
{ "_id" : 25, "name" : "user25", "userid" : 25, "age" : 30 }
{ "_id" : 26, "name" : "user26", "userid" : 26, "age" : 30 }
{ "_id" : 27, "name" : "user27", "userid" : 27, "age" : 30 }
{ "_id" : 28, "name" : "user28", "userid" : 28, "age" : 30 }
{ "_id" : 29, "name" : "user29", "userid" : 29, "age" : 30 }
has more
>
```

c) limit()

limit()方法指定查询的时候每页返回的记录数，如，我要查询所有用户，没有返回 5 条记录：

```
> db.user.find().limit(5);
{ "_id" : 0, "name" : "user0", "userid" : 0, "age" : 30 }
{ "_id" : 1, "name" : "user1", "userid" : 1, "age" : 30 }
{ "_id" : 2, "name" : "user2", "userid" : 2, "age" : 30 }
{ "_id" : 3, "name" : "user3", "userid" : 3, "age" : 30 }
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 30 }
>
```

注：在 shell 和大多数驱动中，limit(0)相当于没设置 limit

d) snapshot()

snapshot()能够保证已经存在于数据中的数据在返回的时候不会缺少对象、不会有重复记录，但是不能保证在查询过程中新增和记录和被删除的记录也被返回。在返回记录小于 1MB 的时候，snapshot()效果是最好的。

e) sort()

sort()方法对返回记录集按照指定字段进行排序返回，1 表示升序，-1 表示降序，如：

```
> db.user.find().limit(10).sort({userid:1});
{ "_id" : 0, "name" : "user0", "userid" : 0, "age" : 30 }
{ "_id" : 1, "name" : "user1", "userid" : 1, "age" : 30 }
{ "_id" : 2, "name" : "user2", "userid" : 2, "age" : 30 }
{ "_id" : 3, "name" : "user3", "userid" : 3, "age" : 30 }
{ "_id" : 4, "name" : "user4", "userid" : 4, "age" : 30 }
{ "_id" : 5, "name" : "user5", "userid" : 5, "age" : 30 }
{ "_id" : 6, "name" : "user6", "userid" : 6, "age" : 30 }
{ "_id" : 7, "name" : "user7", "userid" : 7, "age" : 30 }
{ "_id" : 8, "name" : "user8", "userid" : 8, "age" : 30 }
{ "_id" : 9, "name" : "user9", "userid" : 9, "age" : 30 }
> db.user.find().limit(10).sort({userid:-1});
{ "_id" : 999999, "name" : "user999999", "userid" : 999999, "age" : 30 }
{ "_id" : 999998, "name" : "user999998", "userid" : 999998, "age" : 30 }
{ "_id" : 999997, "name" : "user999997", "userid" : 999997, "age" : 30 }
{ "_id" : 999996, "name" : "user999996", "userid" : 999996, "age" : 30 }
{ "_id" : 999995, "name" : "user999995", "userid" : 999995, "age" : 30 }
{ "_id" : 999994, "name" : "user999994", "userid" : 999994, "age" : 30 }
{ "_id" : 999993, "name" : "user999993", "userid" : 999993, "age" : 30 }
{ "_id" : 999992, "name" : "user999992", "userid" : 999992, "age" : 30 }
{ "_id" : 999991, "name" : "user999991", "userid" : 999991, "age" : 30 }
{ "_id" : 999990, "name" : "user999990", "userid" : 999990, "age" : 30 }
>
```

f) count()

count()方法返回查询记录的总数目，如：

```
> db.user.find().count();
1000000
> db.user.find({_id:{$lt:20}}).count();
20
```

下面这条语句效果一样，但是效率低，而且耗内存大，不建议采用：

```
> db.user.find({_id:{$lt:20}}).toArray().length;
20
```

当查询语句用到了 skip()和 limit()方法的时候，默认情况下 count()会忽略这些方法，如果想要计算这些方法，需要给 count()方法传一个 true 作为参数，如：

```
> db.user.find({_id:{$lt:20}}).skip(3).limit(9).count();
20
> db.user.find({_id:{$lt:20}}).skip(3).limit(9).count(true);
9
>
```

12) $query/$orderby/$explain

```
// select * from users where x=3 and y='abc' order by x asc;
> db.users.find( { x : 3, y : "abc" } ).sort({x:1});
//同上
> db.users.find( { $query : { x : 3, y : "abc" }, $orderby : { x : 1 } } );
//返回执行计划
> db.user.find({$query:{},$explain:1});
```

13) NULL 查询

```
//包含 NULL 情况的查询
> db.foo.insert( { x : 1, y : 1 } )
> db.foo.insert( { x : 2, y : "string" } )
> db.foo.insert( { x : 3, y : null } )
> db.foo.insert( { x : 4 } )

// Query #1
> db.foo.find( { "y" : null } )
{ "_id" : ObjectId("4dc1975312c677fc83b5629f"), "x" : 3, "y" : null }
{ "_id" : ObjectId("4dc1975a12c677fc83b562a0"), "x" : 4 }

// Query #2
> db.foo.find( { "y" : { $type : 10 } } )
{ "_id" : ObjectId("4dc1975312c677fc83b5629f"), "x" : 3, "y" : null }

// Query #3
> db.foo.find( { "y" : { $exists : false } } )
{ "_id" : ObjectId("4dc1975a12c677fc83b562a0"), "x" : 4 }
```

14) Covered Indexes

```
//Covered Indexes
> db.user.ensureIndex({name:1,userid:1,age:1});
> db.user.find({name:'user11',userid:11},{age:30,_id:0});
{ "age" : 30 }
> db.user.find({name:'user11',userid:11},{age:30,_id:1});
{ "_id" : 11, "age" : 30 }
```

15) Dot Notation

```
//Dot Notation
> t.find({})
{ "_id" : ObjectId("4c23f0486dad1c3a68457d20"), "x" : { "y" : 1, "z" : [ 1, 2, 3 ] } }
> t.find({}, {'x.y':1})
{ "_id" : ObjectId("4c23f0486dad1c3a68457d20"), "x" : { "y" : 1 } }
```

16) $slice

```
db.posts.find({}, {comments:{$slice: 5}}) //  前 5 条评论
db.posts.find({}, {comments:{$slice: -5}}) //后 5 条评论
db.posts.find({}, {comments:{$slice: [20, 10]}}) // skip 20, limit 10
db.posts.find({}, {comments:{$slice: [-20, 10]}}) // 20 from end, limit 10
```

3. Remove

Remove 操作用于从集合中删除记录，例如：

```
> db.stu.find();
{ "_id" : 10, "name" : "stu10", "score" : 80 }
{ "_id" : 11, "name" : "stu11", "score" : 81 }
{ "_id" : 12, "name" : "stu12", "score" : 82 }
{ "_id" : 13, "name" : "stu13", "score" : 83 }
{ "_id" : 14, "name" : "stu14", "score" : 84 }
{ "_id" : 15, "name" : "stu15", "score" : 85 }
{ "_id" : 16, "name" : "stu16", "score" : 86 }
{ "_id" : 17, "name" : "stu17", "score" : 87 }
{ "_id" : 18, "name" : "stu18", "score" : 88 }
{ "_id" : 19, "name" : "stu19", "score" : 89 }
//删除一条记录
> db.stu.remove({_id:17});
> db.stu.find();
```

```
{ "_id" : 10, "name" : "stu10", "score" : 80 }
{ "_id" : 11, "name" : "stu11", "score" : 81 }
{ "_id" : 12, "name" : "stu12", "score" : 82 }
{ "_id" : 13, "name" : "stu13", "score" : 83 }
{ "_id" : 14, "name" : "stu14", "score" : 84 }
{ "_id" : 15, "name" : "stu15", "score" : 85 }
{ "_id" : 16, "name" : "stu16", "score" : 86 }
{ "_id" : 18, "name" : "stu18", "score" : 88 }
{ "_id" : 19, "name" : "stu19", "score" : 89 }
>
//删除所有记录
> db.stu.remove();
> db.stu.find();
>
```

建议：删除操作的时候，尽量用_id 作为条件

注意：

某些情况下，当你在对一个记录执行 remove 操作的时候，可能会有 update 操作在这个记录上，这样就可能删除不掉这个记录，如果你觉得这不尽人意，那么你可以在 remove 操作的时候加上$atomic：

```
db.videos.remove( { rating : { $lt : 3.0 }, $atomic : true } )
```

4. Update

1) 语法

```
db.collection.update( criteria, objNew, upsert, multi )
```

参数说明：

Criteria：用于设置查询条件的对象

Objnew：用于设置更新内容的对象

Upsert：如果记录已经存在，更新它，否则新增一个记录

Multi：如果有多个符合条件的记录，全部更新

注意：默认情况下，只会更新第一个符合条件的记录

2) save()

```
//如果存在更新它，如果不存在，新增记录

db.mycollection.save(x);
```

3) 常用操作

$inc

语法：

```
{ $inc : { field : value } }
```

功能：
把 field 的值加一个 value
例子：我要对一个_id=0 的 user 的年龄进行加 1，普通的做法如下：

```
> var u = db.user.findOne({_id:0});
> u
{ "_id" : 0, "name" : "user0", "userid" : 0, "age" : 21 }
> u.age++
21
> db.user.save(u);
> var u = db.user.findOne({_id:0});
> u
{ "_id" : 0, "name" : "user0", "userid" : 0, "age" : 22 }
>
```

但是有了$inc 操作符呢，就不需要这么麻烦，看例子：

```
> db.user.update({_id:0},{$inc:{age:1}});
> var u = db.user.findOne({_id:0});
> u
{ "_id" : 0, "name" : "user0", "userid" : 0, "age" : 23 }
>
```

$set

语法：

```
{ $set : { field : value } }
```

功能：
把 field 的值设置成 value，当 field 不存在时，增加一个字段，类似 SQL 的 set 操作，value 支持所有类型
例子：把上面的 age 改回到 20

```
> db.user.update({_id:0},{$set:{age:20}});
> var u = db.user.findOne({_id:0});
> u
{ "_id" : 0, "name" : "user0", "userid" : 0, "age" : 20 }
//当 field 不存在时，增加一个字段
```

```
> db.user.update({_id:0},{$set:{sex:'boy'}});
> var u = db.user.findOne({_id:0});
> u
{ "_id" : 0, "age" : 20, "name" : "user0", "sex" : "boy", "userid" : 0 }
>
```

$unset

语法：

```
{ $unset : { field : 1} }
```

功能：
删除给定的字段 field，从 1.3 版本以后支持
例子：
删除上一步增加的 sex 字段

```
> db.user.findOne({_id:0});
{ "_id" : 0, "age" : 20, "name" : "user0", "sex" : "boy", "userid" : 0 }
> db.user.update({_id:0},{$unset:{sex:1}});
> db.user.findOne({_id:0});
{ "_id" : 0, "age" : 20, "name" : "user0", "userid" : 0 }
>
```

$push

语法：

```
{ $push : { field : value } }
```

功能：
如果 filed 是一个已经存在的数组，那么把 value 追加给 field；
如果 field 原来不存在，那么新增 field 字段，把 value 的值赋给 field；
如果 field 存在，但是不是一个数组，将会出错；
例子：

```
//第一种情况
> db.user.findOne({_id:0});
{
    "_id" : 0,
    "age" : 20,
    "aihao" : [
        "swim",
        "basketball"
    ],
    "name" : "user0",
```

```
        "userid" : 0
}
> db.user.update({_id:0},{$push:{aihao:'football'}});
> db.user.findOne({_id:0});
{
        "_id" : 0,
        "age" : 20,
        "aihao" : [
                "swim",
                "basketball",
                "football"
        ],
        "name" : "user0",
        "userid" : 0
}
//第二种情况
> db.user.findOne({_id:0});
{ "_id" : 0, "age" : 20, "name" : "user0", "userid" : 0 }
> db.user.update({_id:0},{$push:{aihao:'football'}});
> db.user.findOne({_id:0});
{
        "_id" : 0,
        "age" : 20,
        "aihao" : [
                "football"
        ],
        "name" : "user0",
        "userid" : 0
}
>
//第三种情况
> db.user.find({_id:0});
{ "_id" : 0, "age" : 20, "aihao" : [ "football" ], "name" : "user0", "userid" : 0 }
> db.user.update({_id:0},{$push:{age:6}});
Cannot apply $push/$pushAll modifier to non-array
>
```

$pushAll

语法：

```
{ $pushAll : { field : value_array } }
```

功能：

功能同$push，只是这里的 value 是数组，相当于对数组里的每一个值进行$push 操作

$addToSet

语法：

```
{ $addToSet : { field : value } }
```

功能：
如果 filed 是一个已经存在的数组，并且 value 不在其中，那么把 value 加入到数组；
如果 filed 不存在，那么把 value 当成一个数组形式赋给 field；
如果 field 是一个已经存在的非数组类型，那么将会报错；
例子：

```
> db.user.findOne({_id:0});
{
    "_id" : 0,
    "age" : 20,
    "aihao" : [
        "football"
    ],
    "name" : "user0",
    "userid" : 0
}
> db.user.update({_id:0},{$addToSet:{aihao:'swim'}});
> db.user.update({_id:0},{$addToSet:{aihao:'basketball'}});
> db.user.findOne({_id:0});
{
    "_id" : 0,
    "age" : 20,
    "aihao" : [
        "football",
        "swim",
        "basketball"
    ],
    "name" : "user0",
    "userid" : 0
}
>
```

扩展用法：

```
{ $addToSet : { a : { $each : [ 3 , 5 , 6 ] } } }
```

$pop

语法：

```
{ $pop : { field : 1  } }
```

功能：
删除数组中最后一个元素
语法：

```
{ $pop : { field : -1  } }
```

功能：
删除数组中第一个元素
例子：

```
> db.user.findOne({_id:0});
{
	"_id" : 0,
	"age" : 20,
	"aihao" : [
		"football",
		"swim",
		"basketball"
	],
	"name" : "user0",
	"userid" : 0
}
> db.user.update({_id:0},{$pop:{aihao:1}});
> db.user.findOne({_id:0});
{
	"_id" : 0,
	"age" : 20,
	"aihao" : [
		"football",
		"swim"
	],
	"name" : "user0",
	"userid" : 0
}
> db.user.update({_id:0},{$pop:{aihao:-1}});
> db.user.findOne({_id:0});
{
	"_id" : 0,
	"age" : 20,
	"aihao" : [
```

```
            "swim"
        ],
        "name" : "user0",
        "userid" : 0
}
>
```

$pull

语法：

```
{ $pull : { field : _value } }
```

功能：
如果 field 是一个数组，那么删除符合_value 检索条件的记录；
如果 field 是一个已经存在的非数组，那么会报错；
例子：

```
> db.user.findOne({_id:0});
{
        "_id" : 0,
        "age" : 20,
        "aihao" : [
                "basketball",
                "swim",
                "football",
                "bike"
        ],
        "name" : "user0",
        "userid" : 0
}
> db.user.update({_id:0},{$pull:{aihao:'bike'}});
> db.user.findOne({_id:0});
{
        "_id" : 0,
        "age" : 20,
        "aihao" : [
                "basketball",
                "swim",
                "football"
        ],
        "name" : "user0",
        "userid" : 0
}
//配合其他操作符使用
```

```
> db.user.find({_id:0});
{ "_id" : 0, "age" : 20, "aihao" : [ "swim", "football" ], "name" : "user0", "userid" : 0 }
> db.user.update({_id:0},{$pull:{aihao:{$nin:['swim']}}});
> db.user.find({_id:0});
{ "_id" : 0, "age" : 20, "aihao" : [ "swim" ], "name" : "user0", "userid" : 0 }
>
```

$pullAll

语法：

```
{ $pullAll : { field : value_array } }
```

功能：
同$push 类似，只是 value 的数据类型是一个数组

$rename

语法：

```
{ $rename : { old_field_name : new_field_name }
```

功能:
重命名指定的字段名称，从 1.7.2 版本后开始支持
例子：给 aihao 字段重命名为 ah

```
> db.user.update({_id:0},{$rename:{'aihao':'ah'}});
> db.user.find({_id:0});
{ "_id" : 0, "age" : 20, "ah" : [ "swim" ], "name" : "user0", "userid" : 0 }
>
```

$bit

语法：

```
{$bit : { field : {and : int_value}}}

{$bit : {field : {or : int_value }}}

{$bit : {field : {and : int_value, or : int_value }}}
```

功能：
进行位运算，value 的必须整数型。从 1.7.5 版本后开始支持

4) 特殊操作符：$

$操作符代表查询记录中第一个匹配条件的记录项。
应用：

```
> t.find()

{ "_id" : ObjectId("4b97e62bf1d8c7152c9ccb74"), "title" :
"ABC",
  "comments" : [ { "by" : "joe", "votes" : 3 }, { "by" : "jane",
"votes" : 7 } ] }

> t.update( {'comments.by':'joe'},
{$inc:{'comments.$.votes':1}}, false, true )

> t.find()
{ "_id" : ObjectId("4b97e62bf1d8c7152c9ccb74"), "title" :
"ABC",
  "comments" : [ { "by" : "joe", "votes" : 4 }, { "by" : "jane",
"votes" : 7 } ] }
//$只代表第一个匹配到的元素，如：

> t.find();

{ "_id" : ObjectId("4b9e4a1fc583fa1c76198319"), "x" : [ 1,
2, 3, 2 ] }
> t.update({x: 2}, {$inc: {"x.$": 1}}, false, true);
> t.find();
{ "_id" : ObjectId("4b9e4a1fc583fa1c76198319"), "x" : [ 1,
3, 3, 2 ] }
```

注意：在数组中用$配合$unset 操作符的时候，效果不是删除匹配的元素，而是把匹配的元素变成了 null，如：

```
> t.insert({x: [1,2,3,4,3,2,3,4]})

> t.find()

{ "_id" : ObjectId("4bde2ad3755d00000000710e"), "x" : [ 1,
2, 3, 4, 3, 2, 3, 4 ] }
> t.update({x:3}, {$unset:{"x.$":1}})
> t.find()
{ "_id" : ObjectId("4bde2ad3755d00000000710e"), "x" : [ 1,
2, null, 4, 3, 2, 3, 4 ] }
```

十二、 Shell 控制台

1. 执行.js 文件

MongoDB 的 shell 控制台不仅仅是一个可以互动的控制台，它还可以执行.js 文件，基本语法如下：

```
./mongo server:27017/dbname--quiet my_commands.js
```

说明：

- ./mongo: command to start the interactive shell, may vary on your shell of choice
- server:27017/dbname: basic connection information
- --quiet: this is a flag for the mongo command. This switch removes some header information that is not typically necessary when building unattended scripts.
- my_commands.js: a file containing a series of shell commands to execute

2. –eval

-eval 可以让我们执行一个 js 片段，如：

```
bash-3.2$ ./mongo test --eval
"printjson(db.getCollectionNames())"
MongoDB shell version: 1.8.0
connecting to: test
[
        "system.indexes",
        "t1",
        "t2",
        "test.fam",
        "test1",
        "test11",
        "testBinary",
        "testarray"
]
```

3. 脚本和互动的区别

互动和脚本存在同，下面是四个主要的不同：

a) Printing

在互动控制台上，控制台会自动把返回的数据进行格式化，然而在脚本中，我们明确的去格式化返回结果，下面是两个常用来进行格式化的方法：

print():

普通的打印操作

printjson():

那对象打印成 json 格式

b) use dbname

在互动模式下，该命令可以使用，在脚本模式下，该命令不起作用，我们可以通过如下方式替代：

```
db2 = connect("server:27017/otherdbname")
```

c) it

该命令旨在互动模式下起作用

d) Get Last Error

在互动模式下执行 update/insert 操作时，shell 会自动的等到一个响应，但是在脚本中，这个不会发生。

十三、 安全与认证

Mongodb 目前的安全机制不是很强大，只有基本的对用户名和密码进行认证的功能，同时可以控制用户的读写权限。然而目前的安全机制在分片的时候不起作用，所以在分片的时候官方建议采用信任环境操作。

1) 开启安全认证

用-auth 选项启动 mongod 进程

2) 添加用户

```
$ ./mongo
> use admin
> db.addUser("theadmin", "anadminpassword")
```

3) 认证

```
> db.auth("theadmin", "anadminpassword")
```

4) 查看用户

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

5) 添加普通用户

```
> use projectx
> db.addUser("joe", "passwordForJoe")
```

6) 添加只读用户

```
> use projectx
> db.addUser("guest", "passwordForGuest", true)
```

7) 修改密码

addUser 命令也可以用来修改密码，当用户名已经存在的情况下执行该命令就会更新密码

8) 删除用户

```
db.removeUser( username )
//或者
```

```
db.system.users.remove( { user: username } )
```

十四、 常用 DBA 操作

```
help                        show help

show dbs                    show database names

show collections            show collections in current
database

show users                  show users in current database

show profile                show most recent system.profile
entries with time >= 1ms

use <db name>               set curent database to <db name>


db.addUser (username, password)

db.removeUser(username)


db.cloneDatabase(fromhost)

db.copyDatabase(fromdb, todb, fromhost)

db.createCollection(name, { size : ..., capped : ...,
max : ... } )


db.getName()

db.dropDatabase()

db.printCollectionStats()


db.currentOp() displays the current operation in the db

db.killOp() kills the current operation in the db


db.getProfilingLevel()

db.setProfilingLevel(level) 0=off 1=slow 2=all
```

```
db.getReplicationInfo()

db.printReplicationInfo()

db.printSlaveReplicationInfo()

db.repairDatabase()

db.version() current version of the server

db.shutdownServer()


db.foo.drop() drop the collection

db.foo.dropIndex(name)

db.foo.dropIndexes()

db.foo.getIndexes()

db.foo.ensureIndex(keypattern,options) - options object has
these possible
                                    fields: name, unique,
dropDups


db.foo.find( [query] , [fields])      - first parameter is
an optional
                                    query filter. second
parameter
                                    is optional

                                    set of fields to return.
                                     e.g. db.foo.find(
                                           { x : 77 } ,
                                           { name : 1 , x :
1 } )
db.foo.find(...).count()
db.foo.find(...).limit(n)
db.foo.find(...).skip(n)
db.foo.find(...).sort(...)
db.foo.findOne([query])

db.foo.getDB() get DB object associated with collection
```

```
db.foo.count()
db.foo.group( { key : ..., initial: ..., reduce : ...[,
cond: ...] } )

db.foo.renameCollection( newName ) renames the collection

db.foo.stats()
db.foo.dataSize()
db.foo.storageSize() - includes free space allocated to this
collection
db.foo.totalIndexSize() - size in bytes of all the indexes
db.foo.totalSize() - storage allocated for all data and
indexes
db.foo.validate() (slow)

db.foo.insert(obj)
db.foo.update(query, object[, upsert_bool])
db.foo.save(obj)
db.foo.remove(query)   - remove objects matching query
remove({}) will remove all
```

十五、 图形化管理工具

MongoDB 有几款图形化的管理工具，参考：

http://www.mongodb.org/display/DOCS/Admin+UIs