# single启动

## 启动命令:

mongod --dbpath <path for store data>

# 搭建副本集

## 准备三台机器，分别安装好mongodb。

## 启动每台mongodb

命令：mongod  --dbpath /data/mongodbtest/replset/data   --replSet testrepl

参数分别是

数据存储位置（自己任意指定路径）：/data/mongodbtest/replset/data

副本集参数名称（可任意命名）：testrepl

## 配置副本集

任意登录一台mongodb数据库

#使用admin数据库

use admin

定义副本集配置变量，这里的 \_id:” testrepl” 和上面命令参数“ –replSet testrepl” 要保持一样。

config = { \_id:" testrepl ", members:[

{\_id:0,host:" 192.168.1.136:27017 ",priority:2},

{\_id:1,host:" 192.168.1.137:27017 ",priority:1},

{\_id:2,host:" 192.168.1.138:27017 ",arbiterOnly:true}]

}

使用副本集配置初始化副本集

rs.initiate(config);

接下来需等待一段时间。

完成后可查看节点状态：

rs.state();

输出：

{

         "set" : "tesrepl",

          "date" : ISODate("2013-12-29T12:54:25Z"),

          "myState" : 1,

         "members" : [

                 {

                         "\_id" : 0,

                         "name" : "192.168.1.136:27017",

                         "health" : 1,

                         "state" : 2,

                         "stateStr" : "SECONDARY",

                         "uptime" : 1682,

                         "optime" : Timestamp(1388319973, 1),

                         "optimeDate" : ISODate("2013-12-29T12:26:13Z"),

                         "lastHeartbeat" : ISODate("2013-12-29T12:54:25Z"),

                         "lastHeartbeatRecv" : ISODate("2013-12-29T12:54:24Z"),

                         "pingMs" : 1,

                         "syncingTo" : "192.168.1.138:27017"

                 },

                 {

                         "\_id" : 1,

                         "name" : "192.168.1.137:27017",

                         "health" : 1,

                         "state" : 2,

                         "stateStr" : "SECONDARY",

                         "uptime" : 1682,

                         "optime" : Timestamp(1388319973, 1),

                         "optimeDate" : ISODate("2013-12-29T12:26:13Z"),

                         "lastHeartbeat" : ISODate("2013-12-29T12:54:25Z"),

                         "lastHeartbeatRecv" : ISODate("2013-12-29T12:54:24Z"),

                         "pingMs" : 1,

                         "syncingTo" : "192.168.1.138:27017"

                 },

                 {

                         "\_id" : 2,

                         "name" : "192.168.1.138:27017",

                         "health" : 1,

                         "state" : 1,

                         "stateStr" : "PRIMARY",

                         "uptime" : 2543,

                         "optime" : Timestamp(1388319973, 1),

                         "optimeDate" : ISODate("2013-12-29T12:26:13Z"),

                         "self" : true

                 }

         ],

         "ok" : 1

## 测试副本集数据复制功能

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | #在主节点192.168.1.138 上连接到终端：  mongo 127.0.0.1    #建立test 数据库。  use test;    往testdb表插入数据。  > db.testdb.insert({"test1":"testval1"})    #在副本节点 192.168.1.136、192.168.1.137 上连接到mongodb查看数据是否复制过来。  /data/mongodbtest/mongodb-linux-x86\_64-2.4.8/bin/mongo 192.168.1.136:27017    #使用test 数据库。  repset:SECONDARY> use test;    repset:SECONDARY> show tables; |

#输出

|  |  |  |
| --- | --- | --- |
| 1 | Sun Dec 29 21:50:48.590 error: { "$err" : "not master and slaveOk=false", "code" : 13435 } at src/mongo/shell/query.js:128 | |
| 1  2  3  4  5 | #mongodb默认是从主节点读写数据的，副本节点上不允许读，需要设置副本节点可以读。  repset:SECONDARY> db.getMongo().setSlaveOk();    #可以看到数据已经复制到了副本集。  repset:SECONDARY> db.testdb.find(); |

|  |  |
| --- | --- |
| 1  2 | #输出  { "\_id" : ObjectId("52c028460c7505626a93944f"), "test1" : "testval1" } |

## 测试副本集故障转移功能

先停掉主节点mongodb 138，查看136、137的日志可以看到经过一系列的投票选择操作，查看整个集群的状态，可以看到138为状态不可达。

|  |  |
| --- | --- |
| 1  2  3 | /data/mongodbtest/mongodb-linux-x86\_64-2.4.8/bin/mongo 192.168.1.136:27017    repset:SECONDARY> rs.status(); |

#输出

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48 | {          "set" : "testrepl",          "date" : ISODate("2013-12-29T14:28:35Z"),          "myState" : 2,          "syncingTo" : "192.168.1.137:27017",          "members" : [                  {                          "\_id" : 0,                          "name" : "192.168.1.136:27017",                          "health" : 1,                          "state" : 2,                          "stateStr" : "SECONDARY",                          "uptime" : 9072,                          "optime" : Timestamp(1388324934, 1),                          "optimeDate" : ISODate("2013-12-29T13:48:54Z"),                          "self" : true                  },                  {                          "\_id" : 1,                          "name" : "192.168.1.137:27017",                          "health" : 1,                          "state" : 1,                          "stateStr" : "PRIMARY",                          "uptime" : 7329,                          "optime" : Timestamp(1388324934, 1),                          "optimeDate" : ISODate("2013-12-29T13:48:54Z"),                          "lastHeartbeat" : ISODate("2013-12-29T14:28:34Z"),                          "lastHeartbeatRecv" : ISODate("2013-12-29T14:28:34Z"),                          "pingMs" : 1,                          "syncingTo" : "192.168.1.138:27017"                  },                  {                          "\_id" : 2,                          "name" : "192.168.1.138:27017",                          "health" : 0,                          "state" : 8,                          "stateStr" : "(not reachable/healthy)",                          "uptime" : 0,                          "optime" : Timestamp(1388324934, 1),                          "optimeDate" : ISODate("2013-12-29T13:48:54Z"),                          "lastHeartbeat" : ISODate("2013-12-29T14:28:35Z"),                          "lastHeartbeatRecv" : ISODate("2013-12-29T14:28:23Z"),                          "pingMs" : 0,                          "syncingTo" : "192.168.1.137:27017"                  }          ],          "ok" : 1  } |

再启动原来的主节点 138，发现138 变为 SECONDARY，还是137 为主节点 PRIMARY。

## 添加用户

现在需要创建一个帐号，该账号需要有grant权限，即：账号管理的授权权限。注意一点，帐号是跟着库走的，所以在指定库里授权，必须也在指定库里验证(auth)。

> use admin

> db.createUser({user: "dba",pwd: "dba", roles: [ { role: "userAdminAnyDatabase", db: "admin" } ] })

返回： Successfully added user: { "user" : "dba", "roles" : [ { "role" : "userAdminAnyDatabase", "db" : "admin" } ]}

刚建立了 userAdminAnyDatabase 角色，用来管理用户，可以通过这个角色来创建、删除用户。

验证：需要开启auth参数。

mongod--dbpath"D:\develop\mongodb\data"--logpath"D:\develop\mongodb\data\log\MongoDB.log" --auth

链接到mongodb

>show dbs

没有权限：返回

Error: listDatabases failed:{ "ok" : 0, "errmsg" : "not authorized on admin to execute command { listDatabases: 1.0 }", "code" : 13} at Error (<anonymous>) at Mongo.getDBs (src/mongo/shell/mongo.js:47:15) at shellHelper.show (src/mongo/shell/utils.js:630:33) at shellHelper (src/mongo/shell/utils.js:524:36) at (shellhelp2):1:1 at src/mongo/shell/mongo.js:47> use admin #验证，因为在admin下面添加的帐号，所以要到admin下面验证。

>use admin

>db.auth('dba','dba');

1

> show dbs;

admin 0.078GB

local 0.078GB

>use test #在test库里创建帐号

>db.createUser( { user: "zjy", pwd: "zjy", roles: [ { role: "readWrite", db: "test" }]}) ;

Successfully added user: { "user" : "zjy", "roles" : [ { "role" : "readWrite", #读写账号 "db" : "test" } ]}

## 查看所有用户

在admin库中使用>db.system.users.find();

## 验证权限

上面创建了2个帐号，现在验证下：验证前提需要一个集合

> db.abc.insert({"a":1,"b":2}) #插入失败，没有权限，userAdminAnyDatabase 权限只是针对用户管理的，对其他是没有权限的。

WriteResult({

"writeError" : {

"code" : 13,

"errmsg" : "not authorized on test to execute command { insert: \"abc\", documents: [ { \_id: ObjectId('55915185d629831d887ce2cb'), a: 1.0, b: 2.0 } ], ordered: true }"

}

})

root@zhoujinyi:/usr/local/mongo4# mongo --port=27020

MongoDB shell version: 3.0.4

connecting to: 127.0.0.1:27020/test

> use test

switched to db test

> db.auth('zjy','zjy') #用创建的readWrite帐号进行写入

1> db.abc.insert({"a":1,"b":2})

WriteResult({ "nInserted" : 1 })

> db.abc.insert({"a":11,"b":22})

WriteResult({ "nInserted" : 1 })

> db.abc.insert({"a":111,"b":222})

WriteResult({ "nInserted" : 1 })

> db.abc.find()

{ "\_id" : ObjectId("559151a1b78649ebd8316853"), "a" : 1, "b" : 2 }

{ "\_id" : ObjectId("559151cab78649ebd8316854"), "a" : 11, "b" : 22 }

{ "\_id" : ObjectId("559151ceb78649ebd8316855"), "a" : 111, "b" : 222 }

## 加验证启动副本集

>mongod --dbpath "d:\develop\mongodb\data" --logpath "d:\develop\mongodb\log" --replSet testrepl --keyFile "d:\develop\mongodb\relset.key" --auth

注释:

dbpath：数据存储路径

logpath:日志存储路径

replSet:副本集名称

keyFile:key文件路径，该文件自己定义，如：abc.key，里面内容随便6-1024个长度的英文或数字

auth:表示加权限验证启动。

启动之后链接到主节点，验证之后就可以正常使用。

# 常用命令

## 登录命令:

mongo <IP:port>

## 查看所有的库

show dbs;

## 查看所有的collection

show collections;

## 查询记录

db.collectionName.find({<查询条件json>});

### 基本查询：

    构造查询数据。

    > db.test.findOne()

    {

         "\_id" : ObjectId("4fd58ecbb9ac507e96276f1a"),

         "name" : "stephen",

         "age" : 35,

         "genda" : "male",

         "email" : "[stephen@hotmail.com](mailto:stephen@hotmail.com)"

    }

    --多条件查询。下面的示例等同于SQL语句的where name = "stephen" and age = 35

    > db.test.find({"name":"stephen","age":35})

    { "\_id" : ObjectId("4fd58ecbb9ac507e96276f1a"), "name" : "stephen", "age" : 35, "genda" : "male", "email" : "[stephen@hotmail.com](mailto:stephen@hotmail.com)" }

    --返回指定的文档键值对。下面的示例将只是返回name和age键值对。

    > db.test.find({}, {"name":1,"age":1})

   { "\_id" : ObjectId("4fd58ecbb9ac507e96276f1a"), "name" : "stephen", "age" : 35 }

    --指定不返回的文档键值对。下面的示例将返回除name之外的所有键值对。

    > db.test.find({}, {"name":0})  
    { "\_id" : ObjectId("4fd58ecbb9ac507e96276f1a"), "age" : 35, "genda" : "male", "email" : "[stephen@hotmail.com](mailto:stephen@hotmail.com)" }

### 条件查询：

    MongoDB提供了一组比较操作符：$lt/$lte/$gt/$gte/$ne，依次等价于</<=/>/>=/!=。  
    --下面的示例返回符合条件age >= 18 && age <= 40的文档。  
    > db.test.find({"age":{"$gte":18, "$lte":40}})  
    { "\_id" : ObjectId("4fd58ecbb9ac507e96276f1a"), "name" : "stephen", "age" : 35,"genda" : "male", "email" : "[stephen@hotmail.com](mailto:stephen@hotmail.com)" }  
   
    --下面的示例返回条件符合name != "stephen1"  
    > db.test.find({"name":{"$ne":"stephen1"}})  
    { "\_id" : ObjectId("4fd58ecbb9ac507e96276f1a"), "name" : "stephen", "age" : 35,"genda" : "male", "email" : "[stephen@hotmail.com](mailto:stephen@hotmail.com)" }  
   
    --$in等同于SQL中的in，下面的示例等同于SQL中的in ("stephen","stephen1")  
    > db.test.find({"name":{"$in":["stephen","stephen1"]}})  
    { "\_id" : ObjectId("4fd58ecbb9ac507e96276f1a"), "name" : "stephen", "age" : 35,"genda" : "male", "email" : "[stephen@hotmail.com](mailto:stephen@hotmail.com)" }    
   
    --和SQL不同的是，MongoDB的in list中的数据可以是不同类型。这种情况可用于不同类型的别名场景。  
    > db.test.find({"name":{"$in":["stephen",123]}})  
    { "\_id" : ObjectId("4fd58ecbb9ac507e96276f1a"), "name" : "stephen", "age" : 35,"genda" : "male", "email" : "[stephen@hotmail.com](mailto:stephen@hotmail.com)" }   
   
    --$nin等同于SQL中的not in，同时也是$in的取反。如：  
    > db.test.find({"name":{"$nin":["stephen2","stephen1"]}})  
    { "\_id" : ObjectId("4fd58ecbb9ac507e96276f1a"), "name" : "stephen", "age" : 35,"genda" : "male", "email" : "[stephen@hotmail.com](mailto:stephen@hotmail.com)" }  
   
    --$or等同于SQL中的or，$or所针对的条件被放到一个数组中，每个数组元素表示or的一个条件。

 --下面的示例等同于name = "stephen1" or age = 35  
    > db.test.find({"$or": [{"name":"stephen1"}, {"age":35}]})  
    { "\_id" : ObjectId("4fd58ecbb9ac507e96276f1a"), "name" : "stephen", "age" : 35,"genda" : "male", "email" : "[stephen@hotmail.com](mailto:stephen@hotmail.com)" }   
   
    --下面的示例演示了如何混合使用$or和$in。  
    > db.test.find({"$or": [{"name":{"$in":["stephen","stephen1"]}}, {"age":36}]})  
    { "\_id" : ObjectId("4fd58ecbb9ac507e96276f1a"), "name" : "stephen", "age" : 35,"genda" : "male", "email" : "[stephen@hotmail.com](mailto:stephen@hotmail.com)" }   
   
    --$not表示取反，等同于SQL中的not。  
    > db.test.find({"name": {"$not": {"$in":["stephen2","stephen1"]}}})  
    { "\_id" : ObjectId("4fd58ecbb9ac507e96276f1a"), "name" : "stephen", "age" : 35,"genda" : "male", "email" : "[stephen@hotmail.com](mailto:stephen@hotmail.com)" }

### null数据类型的查询：

    --在进行值为null数据的查询时，所有值为null，以及不包含指定键的文档均会被检索出来。  
    > db.test.find({"x":null})  
    { "\_id" : ObjectId("4fd59d30b9ac507e96276f1b"), "x" : null }  
    { "\_id" : ObjectId("4fd59d49b9ac507e96276f1c"), "y" : 1 }  
   
    --需要将null作为数组中的一个元素进行相等性判断，即便这个数组中只有一个元素。  
 --再有就是通过$exists判断指定键是否存在。  
    > db.test.find({"x": {"$in": [null], "$exists":true}})  
    { "\_id" : ObjectId("4fd59d30b9ac507e96276f1b"), "x" : null }

### 正则查询：

    --MongoDB中使用了Perl规则的正则语法。如：  
    > db.test.find()  
    { "\_id" : ObjectId("4fd59ed7b9ac507e96276f1d"), "name" : "stephen" }  
    { "\_id" : ObjectId("4fd59edbb9ac507e96276f1e"), "name" : "stephen1" }  
    --i表示忽略大小写  
    > db.test.find({"name":/stephen?/i})  
    { "\_id" : ObjectId("4fd59ed7b9ac507e96276f1d"), "name" : "stephen" }  
    { "\_id" : ObjectId("4fd59edbb9ac507e96276f1e"), "name" : "stephen1" } 

### 数组数据查询：

    --基于数组的查找。  
    > db.test.find()  
    { "\_id" : ObjectId("4fd5a177b9ac507e96276f1f"), "fruit" : [ "apple", "banana", "peach" ] }  
    { "\_id" : ObjectId("4fd5a18cb9ac507e96276f20"), "fruit" : [ "apple", "kumquat","orange" ] }  
    { "\_id" : ObjectId("4fd5a1f0b9ac507e96276f21"), "fruit" : [ "cherry", "banana","apple" ] }  
    --数组中所有包含banana的文档都会被检索出来。  
    > db.test.find({"fruit":"banana"})  
    { "\_id" : ObjectId("4fd5a177b9ac507e96276f1f"), "fruit" : [ "apple", "banana", "peach" ] }  
    { "\_id" : ObjectId("4fd5a1f0b9ac507e96276f21"), "fruit" : [ "cherry", "banana","apple" ] }  
    --检索数组中需要包含多个元素的情况，这里使用$all。下面的示例中，数组中必须同时包含apple和banana，但是他们的顺序无关紧要。  
    > db.test.find({"fruit": {"$all": ["banana","apple"]}})  
    { "\_id" : ObjectId("4fd5a177b9ac507e96276f1f"), "fruit" : [ "apple", "banana", "peach" ] }  
    { "\_id" : ObjectId("4fd5a1f0b9ac507e96276f21"), "fruit" : [ "cherry", "banana", "apple" ] }   
    --下面的示例表示精确匹配，即被检索出来的文档，fruit值中的数组数据必须和查询条件完全匹配，即不能多，也不能少，顺序也必须保持一致。  
    > db.test.find({"fruit":["apple","banana","peach"]})  
    { "\_id" : ObjectId("4fd5a177b9ac507e96276f1f"), "fruit" : [ "apple", "banana", peach" ] }   
    --下面的示例将匹配数组中指定下标元素的值。数组的起始下标是0。  
    > db.test.find({"fruit.2":"peach"})  
    { "\_id" : ObjectId("4fd5a177b9ac507e96276f1f"), "fruit" : [ "apple", "banana", peach" ] }   
    --可以通过$size获取数组的长度，但是$size不能和比较操作符联合使用。  
    > db.test.find({"fruit": {$size : 3}})  
    { "\_id" : ObjectId("4fd5a177b9ac507e96276f1f"), "fruit" : [ "apple", "banana", "peach" ] }  
    { "\_id" : ObjectId("4fd5a18cb9ac507e96276f20"), "fruit" : [ "apple", "kumquat","orange" ] }  
    { "\_id" : ObjectId("4fd5a1f0b9ac507e96276f21"), "fruit" : [ "cherry", "banana","apple" ] }   
    --如果需要检索size > n的结果，不能直接使用$size，只能是添加一个额外的键表示数据中的元素数据，在操作数据中的元素时，需要同时更新size键的值。  
    --为后面的实验构造数据。  
    > db.test.update({}, {"$set": {"size":3}},false,true)  
    > db.test.find()  
    { "\_id" : ObjectId("4fd5a18cb9ac507e96276f20"), "fruit" : [ "apple", "kumquat", "orange" ], "size" : 3 }  
    { "\_id" : ObjectId("4fd5a1f0b9ac507e96276f21"), "fruit" : [ "cherry", "banana", "apple" ], "size" : 3 }   
    --每次添加一个新元素，都要原子性的自增size一次。  
    > test.update({},{"$push": {"fruit":"strawberry"},"$inc":{"size":1}},false,true)  
    > db.test.find()  
    { "\_id" : ObjectId("4fd5a18cb9ac507e96276f20"), "fruit" : [ "apple", "kumquat", "orange", "strawberry" ], "size" : 4 }  
    { "\_id" : ObjectId("4fd5a1f0b9ac507e96276f21"), "fruit" : [ "cherry", "banana", "apple", "strawberry" ], "size" : 4 }  
    --通过$slice返回数组中的部分数据。"$slice":2表示数组中的前两个元素。  
    > db.test.find({},{"fruit": {"$slice":2}, "size":0})  
    { "\_id" : ObjectId("4fd5a18cb9ac507e96276f20"), "fruit" : [ "apple", "kumquat" ]}  
    { "\_id" : ObjectId("4fd5a1f0b9ac507e96276f21"), "fruit" : [ "cherry", "banana" ]}   
    --通过$slice返回数组中的部分数据。"$slice":-2表示数组中的后两个元素。  
    > db.test.find({},{"fruit": {"$slice":-2}, "size":0})  
    { "\_id" : ObjectId("4fd5a18cb9ac507e96276f20"), "fruit" : [ "orange", "strawberry" ] }  
    { "\_id" : ObjectId("4fd5a1f0b9ac507e96276f21"), "fruit" : [ "apple", "strawberry" ] }  
    --$slice : [2,1]，表示从第二个2元素开始取1个，如果获取数量大于2后面的元素数量，则取后面的全部数据。  
    > db.test.find({},{"fruit": {"$slice":[2,1]}, "size":0})  
    { "\_id" : ObjectId("4fd5a18cb9ac507e96276f20"), "fruit" : [ "orange" ] }  
    { "\_id" : ObjectId("4fd5a1f0b9ac507e96276f21"), "fruit" : [ "apple" ] }

### 内嵌文档查询：

    --为后面的示例构造测试数据。

    > db.test.find()

    { "\_id" : ObjectId("4fd5ada3b9ac507e96276f22"), "name" : { "first" : "Joe", "last" : "He" }, "age" : 45 }

    --当嵌入式文档为数组时，需要**$elemMatch**操作符来帮助定位某一个元素匹配的情况，否则嵌入式文件将进行全部的匹配。

    --即检索时需要将所有元素都列出来作为查询条件方可。

    > db.test.findOne()

    {

         "\_id" : ObjectId("4fd5af76b9ac507e96276f23"),

         "comments" : [

                 {

                         "author" : "joe",

                         "score" : 3

                 },

                 {

                         "author" : "mary",

                         "score" : 6

                 }

         ]

    }

    > db.test.find({"comments": {"**$elemMatch**": {"author":"joe","score":{"$gte":3}}}}  
    { "\_id" : ObjectId("4fd5af76b9ac507e96276f23"), "comments" : [ { "author" : "joe", "score" : 3 }, { "author" : "mary", "score" : 6 } ] }

### 游标：

    数据库使用游标来返回find()的执行结果，客户端对游标可以进行有效的控制，如：限定结果集的数量、跳过部分结果、基于任意键的任意方向的排序等。  
    下面的例子将用于准备测试数据。  
    > db.testtable.remove()  
    > for (i = 0; i < 10; ++i) {  
    ... db.testtable.insert({x:i})  
    ... }  
    我们可以通过cursor提供的hasNext()方法判断是否还有未读取的数据，再通过next()方法读取结果集中的下一个文档。如：

    > var c = db.testtable.find()  
    > while (c.hasNext()) {  
    ... print(c.next().x)  
    ... }  
    0  
    1  
    2  
    3  
    4  
    5  
    6  
    7  
    8  
    9

    当调用find()的时候，shell并不立即查询数据库，而是等待真正开始要求获得结果的时候才发送查询，这样在执行之前可以给查询附加额外的选项。几乎所有的游标方法都返回本身，因此可以像下面这样将游标的方法链式组合起来。如：

    > var c1 = db.testtable.find().sort({"x":1}).limit(1).skip(4);  
    > var c2 = db.testtable.find().limit(1).sort({"x":1}).skip(4);  
    > var c3 = db.testtable.find().skip(4).limit(1).sort({"x":1});

    此时，查询并未执行，所有这些函数都是在构造查询，当执行下面的语句时，查询将被真正执行，

    > c.hasNext()

查询被发送到服务器，MongoDB服务器每次将返回一批数据，当本批被全部迭代后再从服务器读取下一批数据，直至查询结果需要的数据被全部迭代。

### 分页和排序

     对于上面的示例，limit(1)表示输出结果仅为一个，如果小于1，则不输出，即limit(n)函数限定的是最多输出结果。skip(4)表示跳过查询结果中的前4个文档，如果结果小于4，则不会返回任何文档。sort({"x":1})用于设定排序条件，即按照x键以升序(1)的方式排序，如果需要降序排序可以改为：sort({"x":-1})。sort也可以支持多键排序，如：sort({username:1, age:-1})即先按照username进行升序排序，如果username的值相同，再以age键进行降序排序。这里需要指出的是，如果skip过多的文档，将会导致性能问题。

## 过滤字段

db.collectionName.find({<查询条件json>},{<要显示或排除的字段>});

示例:

db.User.find({\_id:ObjectId(268746238412985)},{\_id:1,name:1,age:0});

查询id为268746238412985的记录,并且只显示\_id和name字段,不显示age字段,1代表显示,0代表不显示.

# 使用mongodb3遇到的问题

## 1、单机版认证管理

### 添加用户的命令的变化:

3.0之前:db.addUser("用户名","密码");

3.0之后:db.createUser({

user:'',

pwd:'',

roles:{

role:'',

db:''

}

});

### 集成spring时开启验证的变化

3.0之前:

|  |
| --- |
| <mongo:mongo-client host="${mongo.host}" port="${mongo.port}" credentials="${mongo.credential}" id="mongo">  </mongo:mongo-client> |

3.0之后:

|  |
| --- |
| <mongo:mongo host="${mongo.host}" port="${mongo.port}" username="" password=""/> |

## 2、集群版认证管理

### 单机版和集群版开启认证的变化

单机版开启认证只需要在启动mongodb时添加--auth即可

集群版开启认证需要添加--auth并且还要给每一个副本指定keyFile,并且各个副本需要使用同一个keyFile.如何添加keyFile参考搭建副本集中的第八步.