10.12 聚合管道

MongoDB 2.2 版本后开始支持 Aggregation Pipeline（聚合管道），Java 驱动包从 2.9.0 版本开始支持 MongoDB 2.2 的特性。

MongoDB 中聚合（aggregate）主要用于处理数据，诸如统计平均值、求和等，并返回计算后的数据结果。

聚合管道是一个强大的工具，能够在分片集群中很好地运行。

聚合管道使用不同的管道阶段操作器可以做不同的统计，管道阶段操作器的值叫管道表达式，表达式几乎兼容了 find 中大量的查询用法，还增加了一些统计表达式。管道的概念是将当前命令的输出结果作为下一个命令的参数，MongoDB 的聚合管道将 MongoDB 文档在一个管道处理完毕后将结果传递给下一个管道处理。管道阶段操作器是可以重复使用的，比如同一批数据进行管道处理时，可以多次使用管道阶段操作器$group。

10.12.1 aggregate 用法

aggregate 命令的语法：

```
db.collection.aggregate(
[{},{}...]
)
```

aggregate 方法中的参数数组表示管道操作，数组中的每个文档表示一种管道操作。aggregate 执行流程如图 10-5 所示。

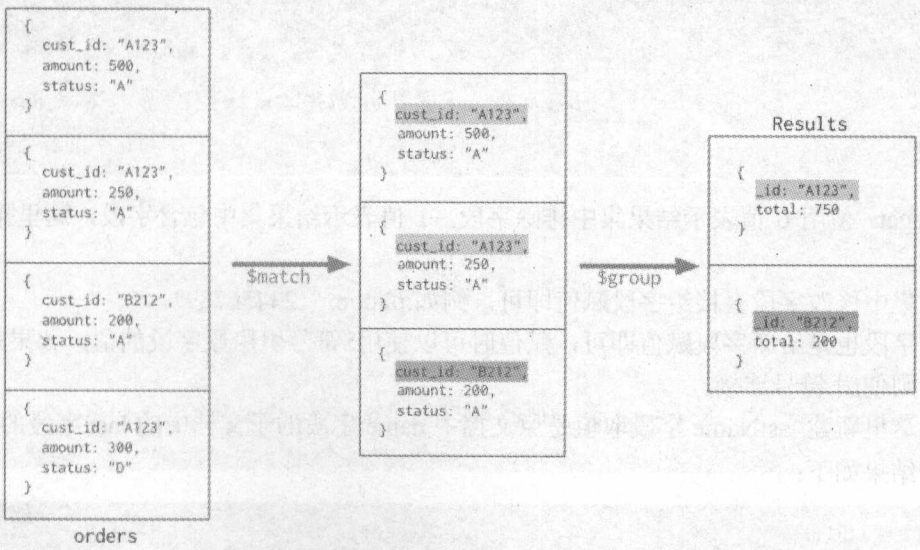

图 10-5　aggregate 执行流程

10.12.2 管道操作器

官方用语 Pipeline Stages，也叫管道阶段。管道阶段需要使用 Stage Operators 识别，我把它称为管道操作器，也叫管道操作符。管道操作器类型很多，这里给出几种常用的管道操作器的用法，更多相关管道操作器请查看官网链接：

```
https://docs.mongodb.com/manual/reference/operator/aggregation-pipeline/
```

1. $project

$project 用于修改输入文档的结构。例如引入文档已存在字段，排除 aggregat 结果集中的 _id 字段（aggregat 结果集默认包含_id 字段），以及在结果集中增加新字段，或者在结果集中修改原字段（不影响原数据）。

我们有数据如下：

```
> db.user.find()
{ "_id" : ObjectId("58d9ba2d6097167df6313438"), "name" : { "first" : "adrian",
"last" : "joe" }, "age" : 14, "phone" : 111128912345 }
```

使用代码：

```
db.user.aggregate(
   [
     { $project: { _id:0,
           age: 1,
           phone: "123456789",
           lastName: "$name.last"
       }
     }
   ]
)
```

aggregate 使用 0 值表示结果集中排除字段，1 值表示结果集中包含字段，这里排除_id，引入 age。

结果集中修改字段直接给字段赋值即可，例如:phone: "123456789"。

新建字段也是给新字段赋值即可，赋值时可以使用$符号引用原字段的值，如果是子文档中的值，则使用.符号连接。

我们这里新建 lastName 字段取值是原文档中 name 字段的子文档中的 last 字段的值。

输出结果如下：

```
> db.user.find()
{ "_id" : ObjectId("58d9ba2d6097167df6313438"), "name" : { "first" : "adrian",
"last" : "joe" }, "age" : 14, "phone" : 111128912345 }
> db.user.aggregate(
```

```
...     [
...         {
...             $project: {
...                 _id:0,
...                 age: 1,
...                 phone: "123456789",
...                 lastName: "$name.last"
...             }
...         }
...     ]
... )
{ "age" : 14, "phone" : "123456789", "lastName" : "joe" }
> db.user.find()
{ "_id" : ObjectId("58d9ba2d6097167df6313438"), "name" : { "first" : "adrian",
"last" : "joe" }, "age" : 14, "phone" : 111128912345 }
```

2. $match

$match 用于过滤数据，只输出符合条件的文档。$match 使用 MongoDB 的标准查询操作，也就是兼容大多数 find 中的查询表达式。

```
db.user.aggregate([
                {
                    $match:{"name.last":"joe"}
                }
])
```

输出 name 字段子文档中 last 字段的值为 joe 的文档。

输出结果如下：

```
> db.user.aggregate([
...                 {
...                     $match:{"name.last":"joe"}
...                 }
... ])
{ "_id" : ObjectId("58d9ba2d6097167df6313438"), "name" : { "first" : "adrian",
"last" : "joe" }, "age" : 14, "phone" : 111128912345 }
```

3. $limit

$limit 用来限制 MongoDB 聚合管道返回的文档数。

```
db.user.aggregate(
```

```
    { $limit : 1 }
);
```

输出 1 条数据。

4. $skip

$skip 在聚合管道中跳过指定数量的文档，并返回余下的文档。

```
db.user.aggregate(
    { $skip : 2 });
```

跳过 2 条记录返回其他文档。

5. $unwind

$unwind 将文档中的某一个数组类型字段拆分成多条，每条包含数组中的一个值。

我们有数据：

```
> db.product.find()
{ "_id" : 1, "item" : "ABC1", "sizes" : [ "S", "M", "L" ] }
```

使用命令：

```
db.product.aggregate( [ { $unwind : "$sizes" } ] )
```

输出结果如下：

```
> db.product.aggregate( [ { $unwind : "$sizes" } ] )
{ "_id" : 1, "item" : "ABC1", "sizes" : "S" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "M" }
{ "_id" : 1, "item" : "ABC1", "sizes" : "L" }
```

6. $group

$group 将集合中的文档分组，可用于统计结果，一般与管道表达式$sum 等组合使用。

我们有数据：

```
> db.user.find()
{ "_id" : ObjectId("58d9ba2d6097167df6313438"), "myName" : "joe", "age" : 14 }
{ "_id" : ObjectId("58d9ba2d6097167df6313439"), "myName" : "ad", "age" : 14 }
{ "_id" : ObjectId("58d9ba2d6097167df631343a"), "myName" : "ad", "age" : 38 }
```

把数据根据 age 字段进行分组，使用命令：

```
db.user.aggregate([{$group : {_id : "$age"}}])
```

输出结果如下：

```
> db.user.aggregate([{$group : {_id : "$age"}}])
```

```
{ "_id" : 38 }
{ "_id" : 14 }
```

7. $sort

$sort 将输入文档排序后输出，1 为按字段升序，-1 为降序。

我们有数据：

```
> db.user.find()
{ "_id" : ObjectId("58d9ba2d6097167df6313438"), "myName" : "joe", "age" : 14 }
{ "_id" : ObjectId("58d9ba2d6097167df6313439"), "myName" : "ad", "age" : 14 }
{ "_id" : ObjectId("58d9ba2d6097167df631343a"), "myName" : "ad", "age" : 38 }
```

使用命令：

```
db.user.aggregate([{$sort : {age :-1}}])
```

输出结果为：

```
> db.user.aggregate([{$sort : {age :-1}}])
{ "_id" : ObjectId("58d9ba2d6097167df631343a"), "myName" : "ad", "age" : 38 }
{ "_id" : ObjectId("58d9ba2d6097167df6313438"), "myName" : "joe", "age" : 14 }
{ "_id" : ObjectId("58d9ba2d6097167df6313439"), "myName" : "ad", "age" : 14 }
```

8. $lookup

$lookup 是 MongoDB 3.2 版本增加的新属性，用于多表连接返回关联数据。

$lookup 执行左连接到一个集合（非分片集合），两个集合必须在同一数据库中。

左连接是数据库操作的专有名词，数据库对多集合操作有左连接、内连接和右连接，目前 MongoDB 3.4 版本只支持左连接。数据集合连接如图 10-6 所示。

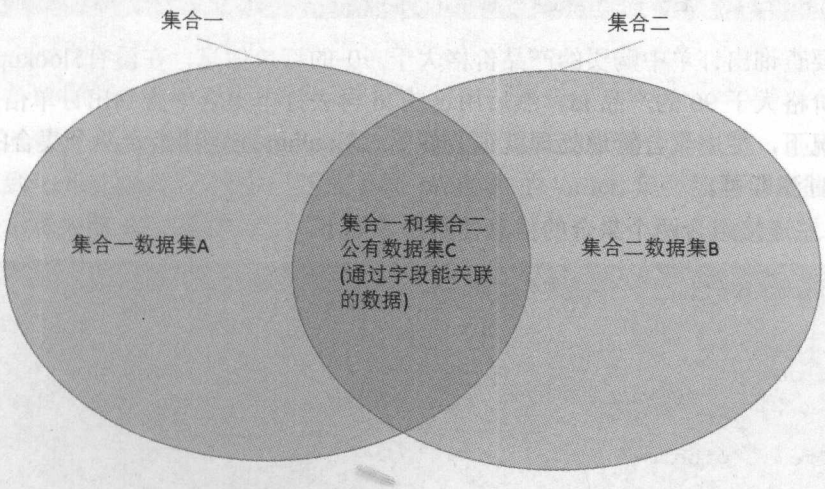

图 10-6　数据集合连接

如上图，获取 C 部分的公有数据叫做数据库的内连接操作，获取 A+C 部分是数据库的左连接操作，获取 B+C 部分是数据库的右连接操作。

集合一和集合二进行内连接时，结果集中只包含集合一和集合二中能相互关联匹配的数据。

集合一和集合二进行左连接时，无论集合一在集合二中是否得到关联匹配，集合一的所有文档都会出现在结果集中。

集合一和集合二进行右连接时，无论集合二在集合一中是否得到关联匹配，集合二的所有文档都会出现在结果集中。

我们之前很多查询统计操作都是针对单个集合的，如果我们在数据结构设计时使用关联存储，在旧版本的 MongoDB 中就需要查询多次才能得到想要的结果。而有了$lookup 之后就能很方便地查询关联的数据集合了。

例子如下：

创建产品信息：

```
db.product.insert({"_id":1,"name":"产品1","price":99})
db.product.insert({"_id":2,"name":"产品2","price":88})
```

用户购买产品时会创建订单信息，如果我们使用内嵌文档的方式存储订单，让订单包含产品信息：产品名称和价格，那么当产品名称和价格有修改时，就需要大量地更新文档。

所以一般来说需要用产品_id 来做关联，订单中 pid 关联到 product 的_id，这样修改产品名称和价格时不需要更新 order 集合。

```
db.order.insert({"_id":1,"pid":1,"name":"订单1"})
db.order.insert({"_id":2,"pid":2,"name":"订单2"})
db.order.insert({"_id":3,"pid":2,"name":"订单3"})
db.order.insert({"_id":4,"pid":1,"name":"订单4"})
db.order.insert({"_id":5,"name":"订单5"})
db.order.insert({"_id":6,"name":"订单6"})
```

现在需要查询出订单中购买的产品价格大于 90 的订单信息，在没有$lookup 之前我们只能先查询出价格大于 90 的产品 id，然后用这些 id 再去订单集合中查询出订单信息。

这种情况下，使用聚合管道处理就很方便了，$lookup 左连接组合两个集合的信息，然后使用$match 过滤即可。

$lookup 左连接组合两个集合的信息使用命令如下：

```
db.order.aggregate([
    {
        $lookup:
        {
         from: "product",
         localField: "pid",
         foreignField: "_id",
```

```
        as: "orderDetail"
      }
   }
])
```

命令使用 order 集合与 product 集合做左连接，localField 表示 order 集合用来关联的字段，foreignField 表示 product 集合用来关联的字段，as 设置关联数据的字段名。

输出结果如下：

```
> db.order.aggregate([
...     {
...         $lookup:
...         {
...          from: "product",
...          localField: "pid",
...          foreignField: "_id",
...          as: "orderDetail"
...         }
...     }
... ])
{ "_id" : 1, "pid" : 1, "name" : "订单1", "orderDetail" : [ { "_id" : 1,
"name" : "产品1", "price" : 99 } ] }
{ "_id" : 2, "pid" : 2, "name" : "订单2", "orderDetail" : [ { "_id" : 2,
"name" : "产品2", "price" : 88 } ] }
{ "_id" : 3, "pid" : 2, "name" : "订单3", "orderDetail" : [ { "_id" : 2,
"name" : "产品2", "price" : 88 } ] }
{ "_id" : 4, "pid" : 1, "name" : "订单4", "orderDetail" : [ { "_id" : 1,
"name" : "产品1", "price" : 99 } ] }
{ "_id" : 5, "name" : "订单5", "orderDetail" : [ ] }
{ "_id" : 6, "name" : "订单6", "orderDetail" : [ ] }
```

我们可以看到 order 集合与 product 集合左连接，以 order 集合为基础，匹配了 product 集合的数据放到 orderDetail 字段中，匹配不到 product 的 order 集合数据仍然被包含在结果集中。左连接示意如图 10-7 所示。

```
> db.order.aggregate([
...     {
...         $lookup:
...         {
...             from: "product",
...             localField: "pid",
...             foreignField: "_id",
...             as: "orderDetail"
...         }
...     }
... ])
{ "_id" : 1, "pid" : 1, "name" : "订单1", "orderDetail" : [ { "_id" : 1, "name" : "产品1", "price" : 99 } ] }
{ "_id" : 2, "pid" : 2, "name" : "订单2", "orderDetail" : [ { "_id" : 2, "name" : "产品2", "price" : 88 } ] }
{ "_id" : 3, "pid" : 2, "name" : "订单3", "orderDetail" : [ { "_id" : 2, "name" : "产品2", "price" : 88 } ] }
{ "_id" : 4, "pid" : 1, "name" : "订单4", "orderDetail" : [ { "_id" : 1, "name" : "产品1", "price" : 99 } ] }
{ "_id" : 5, "name" : "订单5", "orderDetail" : [ ] }
{ "_id" : 6, "name" : "订单6", "orderDetail" : [ ] }
```

图 10-7　左连接

9. $geoNear

$geoNear 用于输出接近某一地理位置的有序文档。

$geoNear 是 MongoDB 2.4 版本增加的新属性，用于地理位置的查询。

find()语法中也有地理位置查询的相关用法，但在 aggregate 中使用$geoNear 有个好处是会返回距离信息。

存储地理数据和编写查询条件前，首先，你必须选择表面类型，这将被用在计算中。你所选择的类型将会影响你的数据如何被存储、建立的索引的类型，以及你的查询的语法形式。

MongoDB 提供了两种表面类型：平面和球面。

（1）平面

如果需要计算距离，就像在一个欧几里得平面上，您可以按照正常坐标对的形式存储位置数据并使用 2d 索引。

平面类型的地理位置信息能解决短距离的距离搜索场景。如果涉及大范围的距离搜索，可能会有偏差，因为地球是球型的。涉及大范围距离搜索请使用球面类型。

平面类型保存位置数据时使用普通坐标对，坐标对可以是数组或者内嵌文档，但是前两个 elements 必须存储固定的一对空间位置数值。例如以下坐标对都可用：

```
{ loc : [ 60 , 30 ] }
{ loc : { x : 90 , y : 32 } }
{ loc : { foo : 70 , y : 80 } }
{ loc : {  lng: 40.739037, lat: 73.992964 } }
```

我们使用 lng 和 lat 的命名，新建数据如下：

```
db.places.save({name:"肯德基",loc : { lng: 40.739037, lat:
73.992964 },category:"餐饮"})
db.places.save({name:"麦当劳",loc : { lng : 42.739037, lat:
73.992964 },category:"餐饮"})
db.places.save({name:"农行",loc : { lng: 41.739037, lat: 73.992964 },category:"
```

```
银行"})
db.places.save({name:"地铁站",loc : { lng: 40.639037, lat:
73.992964 },category:"交通"})
```

需要给坐标对字段创建空间索引，才可以使用地理位置查询。

2d 索引默认取值范围[-179,-179]到[180,180]，包含这两个点，超出范围创建 2d 索引时将报错。

2d 索引创建方式如下：

```
db.places.createIndex( { "loc": "2d" } )
```

我们现在需要查到地铁站附近的文档信息，可以使用：

```
db.places.find({loc : {$near : { lng: 40.639037, lat:73.992964  }}})
```

但是 find 中查询出的文档只能自己去计算范围进行筛选。

aggregate 和$geoNear 能指定范围，比如我们要查范围在坐标值相差 2 度（平面单位）以内的文档：

```
db.places.aggregate([
  {
    $geoNear: {
      spherical:false,
      distanceMultiplier:1,
      near: { lng: 40.639037, lat:73.992964 },
      distanceField: "dist.distacnce",
      maxDistance: 2,
      query: { category: "餐饮" },
      includeLocs: "dist.location",
      num: 1
    }
  }
])
```

spherical 决定是否启用单位弧度，默认为 false，使用的是平面单位度。使用 2dsphere 索引 spherical 必须设置为 true。

near 设置中心坐标点。

distanceField 设置计算所得距离存放的字段名，子文档使用.符号。

distanceMultiplier 可选，设置返回距离数值乘以的倍数。

地理位置数据是普通坐标对的情况下：spherical 为 flase 时$geoNear 返回的计算距离值的单位默认是度（平面单位），如有需要可以使用 distanceMultiplier 调整距离值的单位。

因为平面单位 1 度约 111 千米，乘以 111（推荐值）得到千米数，平面单位 1 度约 69 英里（1 英里＝1.609344 千米），乘以 69（推荐值）得到英里数。

所以在 spherical 为 flase 时，如果需要返回距离单位是千米，distanceMultiplier 设置为 distanceMultiplier:111；如果需要返回的距离单位是英里，distanceMultiplier 设置为 69。

spherical 为 true 时，$geoNear 返回的计算距离值的单位默认是弧度（球面单位），如有需要可以使用 distanceMultiplier 调整距离值的单位。

因为球面单位 1 弧度约 6371 千米（地球半径），1 弧度约 6378137 米（地球半径），所以返回距离值乘以 6371 得到千米数，乘以 6378137 得到米数。

所以在 spherical 为 true 时，如果需要返回距离单位是千米，distanceMultiplier 设置为 6371；如果需要返回的距离单位是米，distanceMultiplier 设置为 6378137。

当地理位置数据是 GeoJSON 格式时：

spherical 无论设置为 true 还是 false，返回距离的单位都是米。

maxDistance，可选参数，最大范围。

地理位置数据是普通坐标对的情况下：

spherical 为 flase 时$geoNear 的 maxDistance 单位默认是度（平面单位）。

如果要限制千米的范围，需要把公里值转换为平面单位度。

例如 2 千米的范围：

```
maxDistance: 2/111
```

spherical 为 true 时，$geoNear 的 maxDistance 单位默认是弧度（球面单位）。

如果要限制公里的范围，需要把公里值转换为弧度。

例如 2 千米的范围：

```
maxDistance: 2/6371
```

例如 500 米的范围：

```
maxDistance: 500/6378137
```

当地理位置数据是 GeoJSON 格式时：spherical 无论设置为 true 还是 false，范围的单位都是米。

query，可选参数，查询条件。

includeLocs，可选参数，设置查询到文档的坐标点的信息字段名。

num，限制返回条数，也可以使用 limit。

输出结果如下：

```
> db.places.aggregate([
...    {
...      $geoNear: {
...         spherical:false,
...         distanceMultiplier:1,
...         near: { lng: 40.639037, lat:73.992964 },
...         distanceField: "dist.distacnce",
...         maxDistance: 2,
```

```
...         query: { category: "餐饮" },
...         includeLocs: "dist.location",
...         num: 1
...     }
...   }
... ])
{ "_id" : ObjectId("58dc801cb9651c39afdafc3e"), "name" : "肯德基", "loc" :
{ "lng" : 40.739037, "lat" : 73.992964 }, "category" : "餐饮", "dist" :
{ "distacnce" : 0.10000000000000142, "location" : { "lng" : 40.739037, "lat" :
73.992964 } } }
```

（2）球面

如果需要计算地理数据就像在一个类似于地球的球形表面上，您可以选择球形表面来存储数据，这样就可以使用 2dsphere 索引。

您可以按照坐标轴:经纬度的方式把位置数据存储为 GeoJSON 对象。GeoJSON 的坐标参考系使用的是 WGS84 数据。

球面类型保存位置数据时可以使用普通坐标对，也可以使用 GeoJSON 对象。

GeoJSON 对象可以有单点、线段、多边形、多点、多线段、多个多边形、几何体集合，例如：

单点：

```
{ type: "Point", coordinates: [ 40, 5 ] }
```

线段：

```
{ type: "LineString", coordinates: [ [ 40, 5 ], [ 41, 6 ] ] }
```

多边形：

```
{
  type: "Polygon",
  coordinates: [ [ [ 0 , 0 ] , [ 3 , 6 ] , [ 6 , 1 ] , [ 0 , 0 ] ] ]
}
```

type 表示类型，coordinates 表示几何点。注意几何点的顺序是 lng, lat。

更多 GeoJSON 信息可以查看官网链接：

```
https://docs.mongodb.com/manual/reference/geojson/ 。
```

我们这里以最简单的单点来学习。

新建数据如下：

```
db.places.save({name:"肯德基",loc : { type: "Point", coordinates: [ 40.639037,
73.992964 ] },category:"餐饮"})
```

```
db.places.save({name:"麦当劳",loc : { type: "Point", coordinates: [ 42.739037,
73.992964 ] },category:"餐饮"})
db.places.save({name:"农行",loc : { type: "Point", coordinates: [ 41.739037,
73.992964 ] },category:"银行"})
db.places.save({name:"地铁站",loc : { type: "Point", coordinates: [ 40.639037,
73.992964 ] },category:"交通"})
```

创建 2dsphere 索引使用代码：

```
db.places.createIndex( { loc : "2dsphere" } )
```

$geoNear 用于输出某一地理位置 2 千米内的文档代码如下：

```
db.places.aggregate([
  {
    $geoNear: {
      spherical: true,
      near: { type: "Point", coordinates: [ 40.639037, 73.992964 ] },
      distanceField: "dist.distacnce",
      maxDistance: 2000,
      query: { category:"餐饮" },
      includeLocs: "dist.location",
      num: 5
    }
  }
])
```

2dsphere 索引 spherical 必须设置为 true，因为 GeoJSON 格式的数据，maxDistance 和返回距离的单位都是米，所以直接使用参数 maxDistance: 2000 就是 2 千米。

输出结果如下：

```
> db.places.aggregate([
...   {
...     $geoNear: {
...       spherical: true,
...       near: { type: "Point", coordinates: [ 40.639037, 73.992964 ] },
...       distanceField: "dist.distacnce",
...       maxDistance: 2000,
...       query: { category:"餐饮" },
...       includeLocs: "dist.location",
...       num: 5
...
```

```
...     }
...   }
... ])
{ "_id" : ObjectId("58dc7f31ad307d26ec284271"), "name" : "肯德基", "loc" :
{ "type" : "Point", "coordinates" : [ 40.639037, 73.992964 ] }, "category" : "
餐饮", "dist" : { "distacnce" : 0, "location" : { "type" : "Point",
"coordinates" : [ 40.639037, 73.992964 ] } } }
```

更多 MongoDB 地理信息距离单位说明查看附录 A。

更多 2d 索引的信息可查看官网链接：

```
https://docs.mongodb.com/manual/core/2d/
```

更多 2dsphere 索引的信息可查看官网链接：

```
https://docs.mongodb.com/manual/core/2dsphere/
```

更多地理位置查询操作可查看官网链接：

```
https://docs.mongodb.com/manual/reference/operator/query-geospatial/
```

10.12.3　管道表达式

管道操作器的值就叫做管道表达式，并且每个管道表达式是一个文档结构，由字段名、字段值和一些表达式操作符组成。管道表达式很多跟 find 中使用的表达式类似，例如：$or、$not、$gt 等。因为种类和数量太多，这里也只给出常用的几种用法，更多管道表达式请查看官网链接：

```
https://docs.mongodb.com/manual/reference/operator/aggregation/#expression-
operators
```

测试数据准备：

```
db.product.insert({"_id":1,"name":"产品1","price":99,"type":"服装"})
db.product.insert({"_id":2,"name":"产品2","price":88,"type":"服装"})
db.product.insert({"_id":3,"name":"产品3","price":29,"type":"饰品"})
db.product.insert({"_id":4,"name":"产品4","price":78,"type":"服装"})
db.product.insert({"_id":5,"name":"产品5","price":9,"type":"饰品"})
db.product.insert({"_id":6,"name":"产品6","price":18,"type":"饰品"})
```

数据如下：

```
> db.product.find()
{ "_id" : 1, "name" : "产品1", "price" : 99, "type" : "服装" }
{ "_id" : 2, "name" : "产品2", "price" : 88, "type" : "服装" }
{ "_id" : 3, "name" : "产品3", "price" : 29, "type" : "饰品" }
```

```
{ "_id" : 4, "name" : "产品4", "price" : 78, "type" : "服装" }
{ "_id" : 5, "name" : "产品5", "price" : 9, "type" : "饰品" }
{ "_id" : 6, "name" : "产品6", "price" : 18, "type" : "饰品" }
```

（1）求和$sum

```
db.product.aggregate([{$group : {_id : "$type", price : {$sum : "$price"}}}])
```

输出结果：

```
> db.product.aggregate([{$group : {_id : "$type", price : {$sum : "$price"}}}])
{ "_id" : "饰品", "price" : 56 }
{ "_id" : "服装", "price" : 265 }
```

（2）平均值$avg

```
db.product.aggregate([{$group : {_id : "$type", price : {$avg : "$price"}}}])
```

输出结果：

```
> db.product.aggregate([{$group : {_id : "$type", price : {$avg : "$price"}}}])
{ "_id" : "饰品", "price" : 18.666666666666668 }
{ "_id" : "服装", "price" : 88.33333333333333 }
```

（3）最小值$min

```
db.product.aggregate([{$group : {_id : "$type", price : {$min : "$price"}}}])
```

输出结果：

```
> db.product.aggregate([{$group : {_id : "$type", price : {$min : "$price"}}}])
{ "_id" : "饰品", "price" : 9 }
{ "_id" : "服装", "price" : 78 }
```

（4）最大值$max

```
db.product.aggregate([{$group : {_id : "$type", price : {$max : "$price"}}}])
```

输出结果：

```
> db.product.aggregate([{$group : {_id : "$type", price : {$max : "$price"}}}])
{ "_id" : "饰品", "price" : 29 }
{ "_id" : "服装", "price" : 99 }
```

（5）数组添加$push

```
db.product.aggregate([{$group : {_id : "$type", tags : {$push : "$name"}}}])
```

输出结果：

```
> db.product.aggregate([{$group : {_id : "$type", tags : {$push : "$name"}}}])
```

```
{ "_id" : "饰品", "tags" : [ "产品3", "产品5", "产品6" ] }
{ "_id" : "服装", "tags" : [ "产品1", "产品2", "产品4" ] }
```

（6）数组添加$addToSet

```
db.product.aggregate([{$group : {_id : "$type", tags : {$addToSet :
"$name"}}}])
```

输出结果：

```
> db.product.aggregate([{$group : {_id : "$type", tags : {$addToSet :
"$name"}}}])
{ "_id" : "饰品", "tags" : [ "产品6", "产品5", "产品3" ] }
{ "_id" : "服装", "tags" : [ "产品4", "产品2", "产品1" ] }
```

$addToSet 与$push 的区别在于重复的值不会进入数组中。

（7）首元素$first

$first 获取分组文档中第一个文档的数据。

```
db.product.aggregate([{$group : {_id : "$type", product : {$first :
"$name"}}}])
```

输出结果：

```
> db.product.aggregate([{$group : {_id : "$type", product : {$first :
"$name"}}}])
{ "_id" : "饰品", "product" : "产品3" }
{ "_id" : "服装", "product" : "产品1" }
```

（8）尾元素$last

$last 获取分组文档中最后一个文档的数据。

```
db.product.aggregate([{$group : {_id : "$type", product : {$last : "$name"}}}])
```

输出结果：

```
> db.product.aggregate([{$group : {_id : "$type", product : {$last :
"$name"}}}])
{ "_id" : "饰品", "product" : "产品6" }
{ "_id" : "服装", "product" : "产品4" }
```

10.12.4　复合使用示例

我们有数据如下：

```
> db.user.find()
{ "_id" : ObjectId("58d9b6afa159504ca6c572e0"), "myName" : "joe", "age" : 28,
```

```
"company" : "google", "name" : "a2" }
{ "_id" : ObjectId("58d9ba2d6097167df6313438"), "myName" : "joe", "age" : 14 }
{ "_id" : ObjectId("58d9ba2d6097167df6313439"), "myName" : "ad", "age" : 14 }
{ "_id" : ObjectId("58d9ba2d6097167df631343a"), "myName" : "ad", "age" : 38 }
{ "_id" : ObjectId("58d9ba2d6097167df631343b"), "myName" : "ad", "age" : 24 }
{ "_id" : ObjectId("58d9ba2e6097167df631343c"), "myName" : "ab", "age" : 14 }
```

我们使用 aggregate 做一个统计，age 大于 13 的文档按 age 分组后统计每组数量。

使用代码如下：

```
db.user.aggregate([
     { $match: { age:{"$gt":13} } },
 { $sort: { age: 1 } },
     { $limit: 2 },
     { $group: { _id: "$age", "人数": { $sum: 1 } } },
   ]
)
```

返回结果如下：

```
> db.user.aggregate([
...       { $match: { age:{"$gt":13} } },
...       { $group: { _id: "$age", "人数": { $sum: 1 } } }
...        ]
... )
{ "_id" : 24, "人数" : 1 }
{ "_id" : 38, "人数" : 1 }
{ "_id" : 14, "人数" : 3 }
{ "_id" : 28, "人数" : 1 }
```

aggregate 默认返回分组信息_id，如果要去掉，可使用$project。

因为 aggregate 是按顺序处理的管道阶段操作器，所以管道的排序也很重要，下面两端代码使用相同的管道阶段操作器，但是不同的顺序，实现的效果也是不同的：

代码一：

```
db.user.aggregate([
     { $match: { age:{"$gt":13} } },
    { $sort: { age: 1 } },
     { $limit: 2 },
     { $group: { _id: "$age", "人数": { $sum: 1 } } }
   ]
)
```

代码段一表示选出 age 大于 13 的文档按 age 的升序排序，取前 2 个文档参与 group，用 age 分组并统计人数。

返回结果：

```
> db.user.aggregate([
...       { $match: { age:{"$gt":13} } },
...      { $sort: { age: 1 } },
...       { $limit: 2 },
...       { $group: { _id: "$age", "人数": { $sum: 1 } } }
...    ]
... )
{ "_id" : 14, "人数" : 2 }
```

代码二：

```
db.user.aggregate([
     { $match: { age:{"$gt":13} } },
     { $group: { _id: "$age", "人数": { $sum: 1 } } },
      { $sort: { _id: 1 } },
     { $limit: 2 }
  ]
)
```

代码二表示选出 age 大于 13 的文档用 age 分组并统计人数，所得结果按_id 的升序排序，取前 2 个文档作为结果返回。

输出结果：

```
> db.user.aggregate([
...       { $match: { age:{"$gt":13} } },
...       { $group: { _id: "$age", "人数": { $sum: 1 } } },
...        { $sort: { _id: 1 } },
...       { $limit: 2 }
...    ]
... )
{ "_id" : 14, "人数" : 3 }
{ "_id" : 24, "人数" : 1 }
```

第 11 章
GUI工具：数据库外部管理工具

第 10 章我们在 MongoDB 提供的原生 Shell 客户端中对 MongoDB 数据库进行了操作。Shell 客户端只能使用命令行进行操作，为了更方便、更直观地可视化操作 MongoDB，很多第三方的 GUI 工具已经被开发出来，本章我们就来学习 GUI 工具。

11.1 MongoDB 的 GUI 工具简介

图形用户界面（Graphical User Interface，简称 GUI，又称图形用户接口）是指采用图形方式显示操作界面，让用户进行可视化操作。MongoDB 官方本身提供了一些可视化工具，例如 MongoDB Cloud Manager、MongoDB Compass。MongoDB Cloud Manager 偏向于部署、运维、监控，而 MongoDB Compass 则偏向于数据管理、查询优化等。MongoDB Atlas 是 MongoDB 官方提供的 DBaaS 服务（DataBase as a Service），目前支持在 Amazon AWS 上构建 MongoDB 的云服务，未来有可能会支持更多的云厂商（ 例如 Azure、Alibaba Cloud 等），并通过 Cloud Manager +Compass 来提供可视化的数据管理。但是 MongoDB 官方提供的 GUI 工具是企业版才支持的功能，社区用户只可以下载试用。

其他第三方的工具有 MongoClient、Mongo-express、AdminMongo、HumongouS.io、NoSQL Manager for MongoDB 、Robomongo 、MongoChef 、Mongobooster 、Mongo Management Studio、MongoMonito、MongoCMS、MongoApp、Mongobird、PHPmongoDB、MongoVision、MongoVUE、Edda 等。

读者可以选择自己喜欢的 GUI 工具，需要注意的是第三方工具有些工具维护跟不上，不支持 MongoDB 3.0 版本以上的数据显示。

更多 MongoDB 的 GUI 工具信息可参考官网链接：

```
https://docs.mongodb.com/ecosystem/tools/administration-interfaces/
```

11.2 Robomongo 基本操作

我们选用 Robomongo 来进行操作示范，在 Robomongo 官网下载安装程序后跟着引导安装即可，Robomongo 官网地址：https://robomongo.org/。

11.2.1　连接 MongoDB

打开工具后在头部选项卡中选择 File→Connect，在打开的窗口中选择 Create，输入要连接的数据库 ip 和端口即可创建新连接，单击连接（如果启用了用户认证 tab 到 Authentication 针对数据库输入用户名密码和加密方式）。Robomongo 连接数据库如图 11-1 所示。

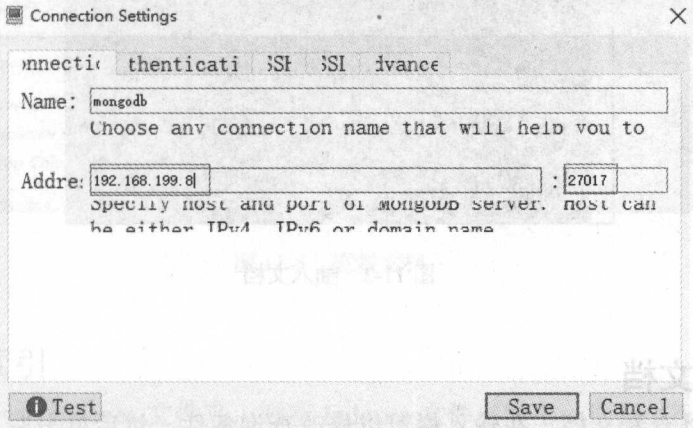

图 11-1　Robomongo 连接数据库

11.2.2　创建删除数据库

连接好数据库之后对着数据库右击，选择 Create Database，如图 11-2 所示，输入数据库名 test 即可创建数据库。对着数据库名右击，选择 Drop Database，即可删除数据库。

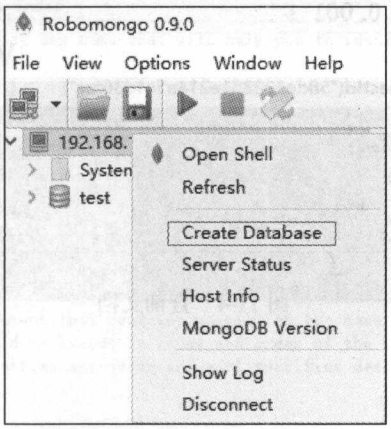

图 11-2　创建数据库

11.2.3　插入文档

数据库展开后有一个 Collections 文件夹，右击可以创建 Collection，对着创建好的 Collection 右击，选择 Insert Document 可以进行文档输入，文档需要符合 BSON 文档格式，

输入完毕后单击右下角的 Save。插入文档如图 11-3 所示。

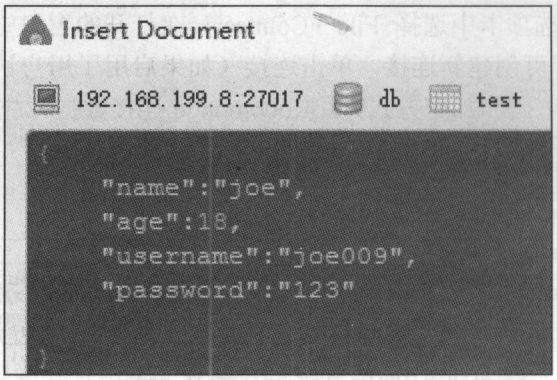

图 11-3　插入文档

11.2.4　查询文档

双击集合可以查看文档，在输入栏可以修改查询条件，然后单击左上角的绿色按钮运行。例如使用 db.getCollection('test').find({"name":"joe"})查询语句，查询文档如图 11-4 所示。

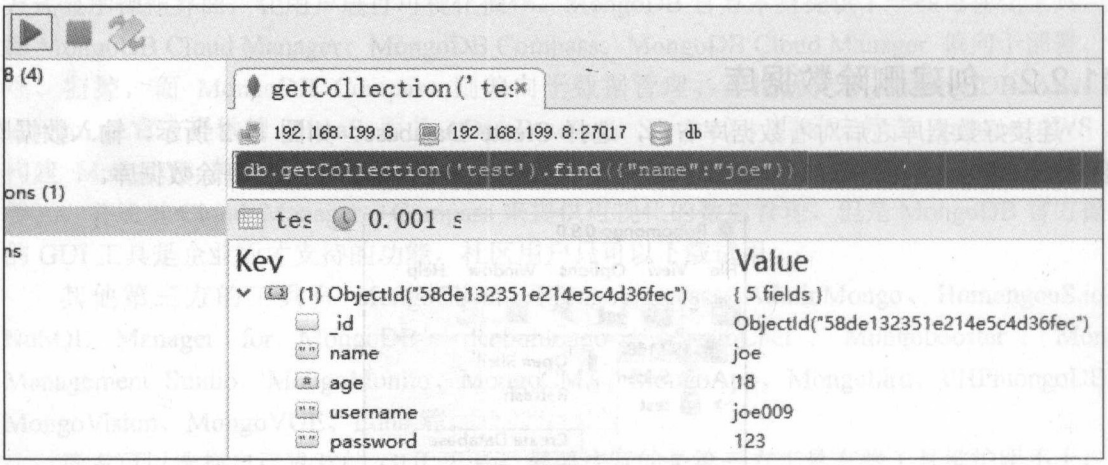

图 11-4　查询文档

11.2.5　更新文档

对着集合名右击，选择 Update Documents，然后输入查询修改参数，更新文档如图 11-5 所示。

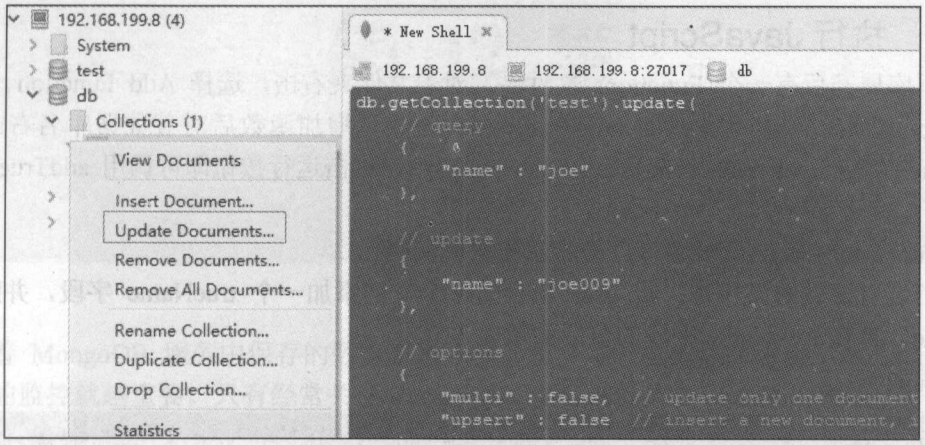

图 11-5　更新文档

11.2.6　创建索引

集合展开后有一个 Indexes 文件夹，对着 Indexes 文件夹右击，选择 Add Index 可以增加索引，输入索引名称，输入索引文档（字段和排序），索引文档满足 Bson 格式：1 表示升序，-1 表示降序。添加索引如图 11-6 所示。

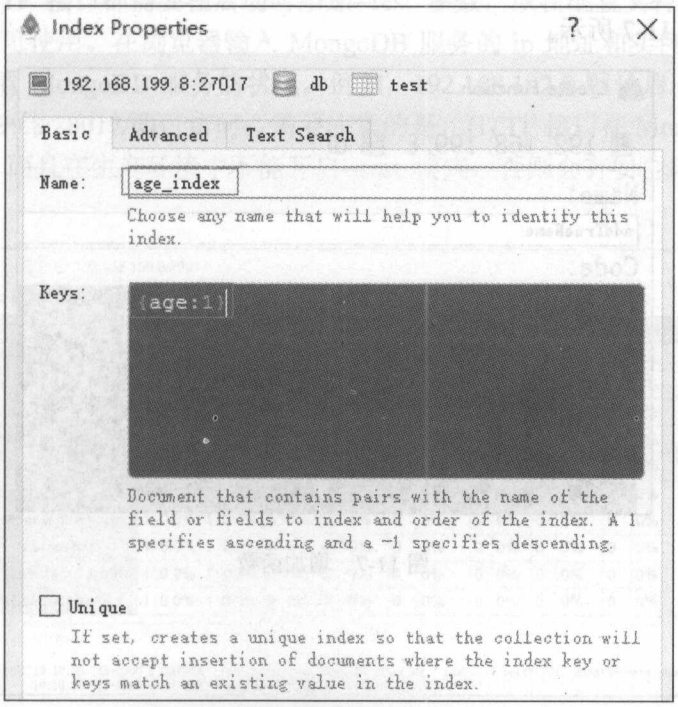

图 11-6　添加索引

11.2.7 执行 JavaScript

数据库展开后有一个 Functions 文件夹，对着文件夹右击，选择 Add Function 可以增加 JavaScript 函数，我们这里增加 addTrueName 函数。增加函数后对着数据库名右击，选择 Open Shell，输入 db.eval("return addTrueName();")，单击运行按钮即可调用 addTrueName 函数。

在 function 中我们可以编写自己的业务实现代码。

我们这里要做的操作是：给 user 集合的每个文档添加一个 trueName 字段，并赋值等于 userName，再在 userName 字段的值后面加上 110。

代码如下：

```
function () {
  db.user.find().forEach(function(item){
      item.trueName=item.userName;
        item.userName=item.userName+"110";
        db.user.save(item);
  }
 )
}
```

增加函数如图 11-7 所示。

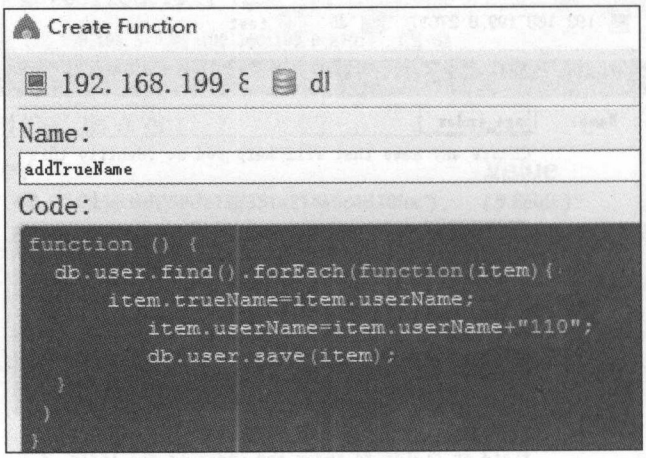

图 11-7 增加函数

第 12 章

◂ 监控 ▸

随着 MongoDB 服务中保存的数据越来越多，承受越来越密集的访问，对 MongoDB 服务状态的监控就越重要。只有经常关注 MongoDB 服务的状态是否健康，才能更好地防止故障以及对 MongoDB 服务的部署做出相应优化。

12.1 原生管理接口监控

MongoDB 提供了原生的管理接口：REST 接口和 HTTP 接口。REST 接口可用于配置监控、告警脚本和其他一些管理任务，HTTP 接口在 Web 界面上显示 MongoDB 服务的情况。REST 接口和 HTTP 接口都需要在启动时添加--rest 参数，或在配置文件加上 rest=true 开启 REST 接口支持才可使用。在浏览器输入 MongoDB 服务的 ip 地址和实例端口加上 1000，即可在 Web 界面查看 MongoDB 服务的状况。例如，192.168.192.8 默认启动端口是 27017，使用 http://192.168.199.8:28017/即可访问。需要注意的是，HTTP 接口在 MongoDB 3.2 版本之后已经不赞成使用，而且在生产环境中不能开启--rest 模式，否则会有安全风险。HTTP 接口监控如图 12-1 所示。

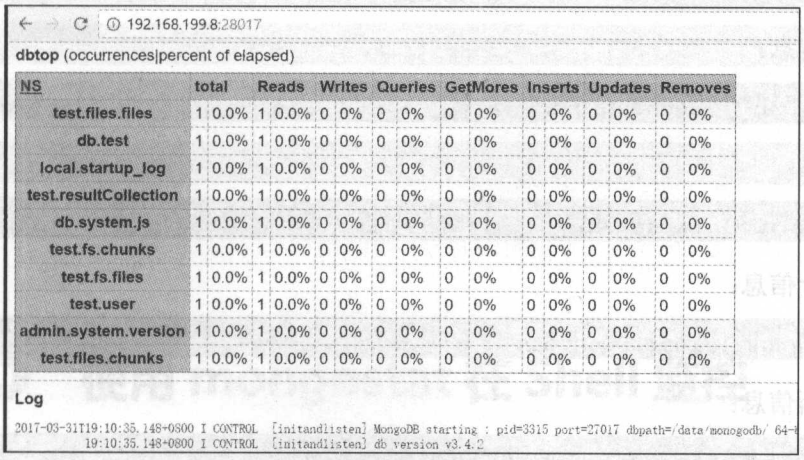

图 12-1 HTTP 接口监控

12.2 使用 serverStatus 在 Shell 监控

serverStatus 命令可以查看 MongoDB 服务的状态，有助于了解诊断和性能分析。使用 mongo 命令进入 Shell 客户端后输入命令：

```
db.serverStatus();
```

serverStatus 输出的信息非常多，书中就不全部给出了，需要详细了解的可参考官网说明。serverStatus 命令相关官网地址：

```
https://docs.mongodb.com/manual/reference/command/serverStatus/
```

在实际使用中 db.serverStatus()命令输出的信息太多，看不过来，一般是根据要监控的情况使用细分的函数来查看。分为以下几种情况。

主机名：

```
db.serverStatus().host
```

锁信息：

```
db.serverStatus().locks
```

全局锁信息：

```
db.serverStatus().globalLock
```

内存信息：

```
db.serverStatus().mem
```

连接数信息：

```
db.serverStatus().connections
```

额外信息：

```
db.serverStatus().extra_info
```

索引统计信息：

```
db.serverStatus().indexCounters
```

后台刷新信息：

```
db.serverStatus().backgroundFlushing
```

游标信息：

```
db.serverStatus().cursors
```

网络信息：

```
db.serverStatus().network
```

副本集信息：

```
db.serverStatus().repl
```

副本集的操作计数器：

```
db.serverStatus().opcountersRepl
```

操作计数器：

```
db.serverStatus().opcounters
```

断言信息 asserts：

```
db.serverStatus().asserts
```

writeBacksQueued：

```
db.serverStatus().writeBacksQueued
```

持久化 dur：

```
db.serverStatus().dur
```

记录状态信息：

```
db.serverStatus().recordStats
```

工作集配置：

```
db.serverStatus( { workingSet: 1 } ).workingSet
```

指标信息 metrics：

```
db.serverStatus().metrics
```

想监控具体某个参数，把参数名放在 db.serverStatus().之后即可。

12.3 使用 mongostat 在 Shell 监控

serverStatus 命令是静态的监控，MongoDB 提供了动态的监控执行工具 mongostat。mongostat 会动态输出一些 serverStatus 提供的重要信息，每秒输出一次。mongostat 的使用方式跟 mongo 客户端一样，需要在 mongostat 可执行文件下使用命令：

```
./ mongostat
```

如果 MongoDB 可执行文件 Bin 目录已经加入环境变量，则直接使用：

```
mongostat
```

mongostat 监控界面如图 12-2 所示。

```
[root@mongodb0 ~]# mongostat
insert query update delete getmore command dirty used flushes vsize  res qrw arw net_in net_out conn
    time
    *0    *0     *0    *0       0     2|0  0.0% 0.0%       0 343M 41.0M 0|0 0|0   158b   44.9k    8
6:18.833
    *0    *0     *0    *0       0     1|0  0.0% 0.0%       0 343M 41.0M 0|0 0|0   157b   44.8k    8
6:19.834
    *0    *0     *0    *0       0     2|0  0.0% 0.0%       0 343M 41.0M 0|0 0|0   215b   45.0k    8
6:20.834
    *0    *0     *0    *0       0     1|0  0.0% 0.0%       0 343M 41.0M 0|0 0|0   157b   44.7k    8
6:21.835
    *0    *0     *0    *0       0     1|0  0.0% 0.0%       0 343M 41.0M 0|0 0|0   156b   44.5k    8
```

图 12-2　mongostat 监控

12.4 使用第三方插件监控

目前已经有很多第三方 MongoDB 的监控插件，比如 Hyperic、Nagios、Ganglia、Cacti、Zabbix、Munin、Openfalcon 等。感兴趣的读者可以进行了解学习，本书不再讲解。

第 13 章

◂ 安全和访问控制 ▸

MongoDB 的安全模式默认是关闭的，也就是不需要账号密码就能够访问数据库，这给我们的开发和使用带来了很多便利，但是 MongoDB 需要在一个可信任的运行环境中。很多 MongoDB 的使用者并没有意识到这点，没有经过任何设置就把 MongoDB 暴露在外网环境中，无异于让数据裸奔。2017 年年初发生了比较轰动的黑客赎金事件，很多 MongoDB 数据库中的数据被黑客删除索要赎金，MongoDB 官方做出了回应，这些攻击完全可以通过 MongoDB 中内置的完善的安全机制来预防，只要按照我们的安全文档正确使用这些功能，就可以避免攻击事件的发生。由此可见，正确的 MongoDB 打开姿势很重要。

我们可以通过以下几个方面来提高 MongoDB 数据库的安全性。

13.1 绑定监听 ip

MongoDB 可以通过设置--bind_ip 参数来设置 MongoDB 服务监听哪些 ip，设置了监听之后，只有使用这些 ip 才能够访问这个 MongoDB 服务。只需要在启动时或者配置文件中加上--bind_ip 即可，多个 ip 用逗号隔开。例如，192.168.199.8 上的 MongoDB 实例可使用命令如下：

```
mongod  --bind_ip 127.0.0.1, 192.168.199.8
```

表示只监听 127.0.0.1 和 192.168.199.8，只有使用这两个 ip 才能连接该 MongoDB。

绑定监听 ip 后的 MongoDB 服务在 mongo 客户端连接时需要加上设置的 ip 参数，例如：

```
mongo 192.168.199.8
```

如果 192.168.199.8 上的 MongoDB 服务使用 192.168.199.9 作为--bind_ip 参数会报错 bind() failed Cannot assign requested address for socket: 192.168.199.9:27017，说明这个 ip 与本机不对应。

注意，网上很多资料说--bind_ip 是用来限制哪些 ip 能够访问 MongoDB 的，这种说法是错误的。限制 ip 访问是限制监听后的效果，--bind_ip 并不能直接限制哪个 ip 不能访问 MongoDB 服务（使用 Linux 等防火墙能实现该功能）。

通俗点说，--bind_ip 就是告诉 MongoDB 实例它自己叫什么名字，比如 ip 是 192.168.199.8 的服务器启动了 MongoDB 实例，它的外网 ip 是 122.130.22.14。如果我们不设置--bind_ip，在连接这个 MongoDB 实例时，可以在 192.168.199.8 上使用 mongo 127.0.0.1 来连接，可以在局域网的机子中使用 mongo 192.168.199.8 来连接，可以在公网环境中使用 mongo 122.130.22.14 来连接。因为这三个身份都是它。但是如果我们使用了 mongod --bind_ip 192.168.199.8，告诉 MongoDB 你的名字是 192.168.199.8，你只监听这个 ip，其他机子用这个称呼连接你时你才答应。设置好之后在 192.168.199.8 上使用 mongo 127.0.0.1 来连接会失败，在公网环境中使用 mongo 122.130.22.14 来连接也会失败，因为 MongoDB 现在只认 192.168.199.8 这个名字。也就是只有局域网中的机子（包括本机）才能使用 mongo 192.168.199.8 连接访问 MongoDB，间接实现了限制 ip 访问的功能。

若绑定为 127.0.0.1，则只能本机访问 MongoDB 服务，不指定--bind_ip 默认所有 IP 都能访问 MongoDB 服务。

13.2 设置监听端口

MongoDB 默认的监听端口是 27017，为了安全起见，可以修改这个监听端口，避免恶意的连接尝试。在启动时或者配置文件中加上--port 即可，使用命令：

```
mongod  --port 36000
```

把 MongoDB 服务的端口设置为 36000，则 mongo 客户端连接时也需要带端口，使用命令如下：

```
mongo 127.0.0.1:36000
```

13.3 用户认证

MongoDB 在默认的情况下启动时是没有开启用户认证的，如果需要使用账号密码验证功能，需要先打开用户认证的开关。MongoDB 3.0 版本之后用户创建和权限方面变化了很多，早期版本在网上能找到很多资料，本书以 MongoDB 3.4 版本为例，更多信息可查看官网链接：

```
https://docs.mongodb.com/manual/tutorial/enable-authentication/
```

13.3.1 启用认证

启动 MongoDB 时加上--auth 即可开启认证模式，使用命令：

```
mongod --auth
```

在开启了访问权限控制的 MongoDB 实例上，用户能进行的操作取决于登录账号的角色（roles）。

13.3.2　添加用户

在开启访问权限控制时，需要确保 admin 库中有一个被分配了 userAdmin 或者 userAdminAnyDatabase 角色的用户账号。这个账号可以管理用户和角色，比如：创建用户、获取角色权限、创建或修改自定义角色等。

在访问权限控制开启之前或之后，都可以执行创建用户的操作。如果你在开启访问权限控制之前没有创建任何用户，MongoDB 提供一个特有机制，让你能够在 admin 库中创建管理员账号。一旦管理员账号创建完毕，其他账号则必须使用该管理员账号进行创建和控制权限。

（1）创建管理员账号

```
use admin
db.createUser(
  {
    user: "myUserAdmin",
    pwd: "abc123",
    roles: [ { role: "userAdminAnyDatabase", db: "admin" } ]
  }
)
```

使用命令 use admin 创建的用户账号信息保存在 admin 数据库下。

（2）创建普通账号

```
use test
db.createUser(
  {
    user: "myTester",
    pwd: "xyz123",
    roles: [ { role: "readWrite", db: "test" },
             { role: "read", db: "reporting" } ]
  }
)
```

使用命令 use test 创建的用户账号信息保存在 test 数据库下。

13.3.3　用户权限控制

我们已经看到创建用户时是需要带 roles 参数的，这就是用户的权限控制。role 表示可以执行的操作，db 表示可以操作的数据库。

权限除了在创建用户时赋值，也可以之后修改。

（1）查看用户权限

```
use test
db.getUser("myTester")
```

会输出：

```
...
roles: [ { role: "readWrite", db: "test" },
   { role: "read", db: "reporting" } ]
```

（2）查看权限能执行哪些操作

例如我们要看 test 数据库中 read 权限能执行哪些操作：

```
use test
db.getRole( "read", { showPrivileges: true } )
```

（3）授权

```
use test
db.grantRolesToUser(
   "myTester",
   [
     { role: "readWrite", db: "reporting" }
   ]
)
```

myTester 用户增加 reporting 数据库的读写权限。

（4）取消权限

```
use test
db.revokeRolesFromUser(
   "myTester",
   [
     { role: "readWrite", db: "reporting" }
   ]
)
```

myTester 用户取消 reporting 数据库的读写权限。

更多权限参数相关信息可以查看官网链接：

```
https://docs.mongodb.com/manual/reference/built-in-roles/
```

和

```
https://docs.mongodb.com/manual/tutorial/manage-users-and-roles/#view-a-role-s-privileges
```

13.3.4　用户登录

（1）启动 mongo 客户端时登录：

```
mongo --port 27017 -u "myUserAdmin" -p "abc123" --authenticationDatabase "admin"
```

参数--authenticationDatabase "admin"表示 myUserAdmin 用户在 admin 数据库下。

（2）进入 mongo 客户端后再登录：

```
mongo --port 27017
use admin
db.auth("myUserAdmin", "abc123" )
```

输出 1 则表示登录成功。

13.3.5　修改密码

```
db.changeUserPassword("myTester", "456789")
```

将 myTester 的密码修改为 456789，需要 admin 管理员权限。

13.3.6　删除用户

```
db.dropUser("myTester")
```

需要 admin 管理员权限。

第 14 章

◂ 数据管理 ▸

数据的备份和恢复是工作中常用的操作，本章学习数据的管理。

14.1 数据备份 mongodump

MongoDB 提供了可执行文件 mongodump 用于数据备份，mongodump 的原理是对 MongoDB 进行普通查询，然后写入文件中。在 mongodump 可执行文件的 bin 目录使用命令：

```
./mongodump  -d test  -o /home/joe/
```

有配置环境的 Linux 任意路径使用命令（mongodump 备份的数据文件如图 14-1 所示）：

```
mongodump  -d test  -o /home/joe/
```

mongodump 也可以使用-q 参数增加查询条件，只导出满足条件的文档，使用命令：

```
mongodump  -d test  -c user  -q "{name: 'joe'}"  -o /home/joe/
```

注意-q 参数值的标点符号，否则会报错 positional arguments not allowed。

更多 mongodump 的参数使用命令./mongodump --help 查看，如图 14-1 所示。

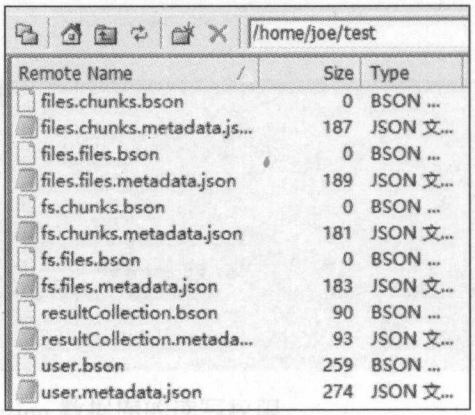

图 14-1　mongodump 备份的数据文件

14.2 数据恢复 mongorestore

mongorestore 可执行文件与 mongodump 搭配使用，用于恢复数据库。

mongorestore 使用的数据文件就是 mongodump 备份的数据文件，使用命令如下：

```
mongorestore  -d  test  /home/joe/test  --drop
```

使用/home/joe/test 路径下的 BSON 和 JSON 文件恢复数据库 test，--drop 参数表示如果已经存在 test 数据库则删除原数据库，去掉--drop 则恢复数据库时与原数据库合并。

14.3 数据导出 mongoexport

mongodump 主要是针对库的备份，MongoDB 还提供了一种针对集合的备份工具：可执行文件 mongoexport。mongoexport 可以指定导出的格式，还可以指定导出的字段，比较灵活。

14.3.1 导出 JSON 格式

导出 JSON 格式的备份文件使用命令：

```
mongoexport -d test -c user -o /home/joe/user.dat
```

导出 test 数据库中 user 集合到目录/home/joe 下的 user.dat 文件中，查看 user.dat 文件发现里面的数据是 JSON 格式的。

mongoexport 也可以使用-q 参数增加查询条件，只导出满足条件的文档，使用命令：

```
mongoexport -d test -c user  -q  "{name: 'joe'}"  -o /home/joe/user.dat
```

注意-q 参数值的标点符号，否则会报错 too many positional arguments。

14.3.2 导出 CSV 格式

导出 CSV 格式的备份文件使用命令：

```
mongoexport -d test -c user --csv -f id,name,age -o /home/joe/user.csv
```

导出 test 数据库中 user 集合到目录/home/joe 下的 user.csv 文件中。-f 参数用于指定只导出 id、name 以及 age 字段。因为 CSV 是表格类型的，对于内嵌文档太深的数据导出效果不是很好，所以一般来说会指定某些字段导出。

14.4 数据导入 mongoimport

数据导入工具 mongoimport 与 mongoexport 配合使用，使用 mongoexport 导出的备份文件进行数据恢复。

14.4.1 JSON 格式导入

```
mongoimport -d test -c user  /home/joe/user.dat  --upsert
```

使用备份文件/home/joe/user.dat 导入数据到 test 数据库的 user 集合中，--upsert 表示更新现有数据，如果不使用--upsert，则导入时已经存在的文档会报_id 重复，数据不再插入。也可以使用--drop 删除原数据。

14.4.2 CSV 格式导入

```
mongoimport -d test -c user --type csv --headerline --file /home/joe/user.csv
```

导入/home/joe 目录下的 user.csv 文件中的数据到 test 的 user 集合。

--headerline 指明不导入第一行，CSV 格式的文件第一行为列名。

第 15 章

◂MongoDB驱动▸

前面几个章节我们都是使用 MongoDB 原生客户端或者第三方客户端在对 MongoDB 进行操作，这样的使用场景并不能满足我们在工作环境中使用 MongoDB。比如我们的 Web 应用可能是使用 Java 语言编写的，或者使用 PHP 语言编写。

这时候怎么能够让我们的程序使用 MongoDB 数据库做存储和读取呢？答案是使用 MongoDB 驱动。MongoDB 驱动让我们可以使用自己熟悉和喜欢的计算机语言对 MongoDB 数据库进行操作，前提是 MongoDB 提供了这种语言的驱动支持。

15.1 MongoDB 驱动支持的开发语言

计算机语言的种类非常多，我们可以在 TIOBE 官网上进行了解。TIOBE 编程语言排行榜是编程语言流行趋势的一个指标，每月更新，这份排行榜排名基于互联网上有经验的程序员、课程和第三方厂商的数量。排名使用著名的搜索引擎（诸如 Google、MSN、Yahoo!、Wikipedia、YouTube 以及 Baidu 等）进行计算。请注意这个排行榜只是反映某个编程语言的热门程度，并不能说明一门编程语言好不好，或者一门语言所编写的代码数量多少。

这个排行榜可以用来考查你的编程技能是否与时俱进，也可以在开发新系统时作为一个语言选择依据。 TIOBE 官网地址：

```
https://www.tiobe.com/tiobe-index/
```

比如，2017 年 3 月份 TIOBE 编程语言排行榜前 20 列表如图 15-1 所示。

Mar 2017	Mar 2016	Change	Programming Language	Ratings	Change
1	1		Java	16.384%	-4.14%
2	2		C	7.742%	-6.86%
3	3		C++	5.184%	-1.54%
4	4		C#	4.409%	+0.14%
5	5		Python	3.919%	-0.34%
6	7	^	Visual Basic .NET	3.174%	+0.61%
7	6	v	PHP	3.009%	+0.24%
8	8		JavaScript	2.667%	+0.33%
9	11	^	Delphi/Object Pascal	2.544%	+0.54%
10	14	^^	Swift	2.268%	+0.68%
11	9	v	Perl	2.261%	+0.01%
12	10	v	Ruby	2.254%	+0.02%
13	12	v	Assembly language	2.232%	+0.39%
14	16	^	R	2.016%	+0.73%
15	13	v	Visual Basic	2.008%	+0.33%
16	15	v	Objective-C	1.997%	+0.54%
17	48	^^	Go	1.982%	+1.78%
18	18		MATLAB	1.854%	+0.66%
19	19		PL/SQL	1.672%	+0.48%
20	26	^^	Scratch	1.472%	+0.70%

图 15-1　2017 年 3 月份 TIOBE 编程语言排行榜前 20

MongoDB 还没来得及对所有的计算机语言提供驱动支持，但是可以放心的是目前常用的比较热门的计算机语言都已经得到了 MongoDB 驱动支持。

目前 MongoDB 驱动支持的开发语言有：C、C++、C#、Java、Node.js、Perl、PHP、Python、Motor、Ruby、Scala、Go、Erlang。

相关驱动的下载和使用 API 等可以查看官网 MongoDB Drivers 的信息，官网地址：

```
https://docs.mongodb.com/ecosystem/drivers/
```

MongoDB Drivers 列表如图 15-2 所示。

Documentation	Releases	Source	API	JIRA	Online Course
C	Releases	Source	API	JIRA	
C++11	Releases	Source	API	JIRA	
C#	Releases	Source	API	JIRA	Course
Java	Releases	Source	API	JIRA	Course
Node.js	Releases	Source	API	JIRA	Course
Perl	Releases	Source	API	JIRA	
PHP	Releases	Source	API	JIRA	
Python	Releases	Source	API	JIRA	Course
Motor	Releases	Source	API	JIRA	
Ruby	Releases	Source	API	JIRA	
Scala	Releases	Source	API	JIRA	

图 15-2　MongoDB Drivers

15.2　驱动使用流程

不同开发语言的驱动使用流程是类似的，我们这里可以先了解大概的流程。按照流程走一遍，我们就可以在这种开发语言中使用 MongoDB 了。

（1）开发语言开发环境配置

首先是开发语言开发环境的配置，不同的开发语言需要不同的环境和编辑器，比如 Java 语言需要 JDK 的环境，以及有 Eclipse 等各种编辑器可以选择；C#需要.NET 框架环境，有 VS 等编辑器可以选择。

（2）加载驱动

首先在官网 MongoDB Drivers 的地址 https://docs.mongodb.com/ecosystem/drivers/里面选择 MongoDB 驱动包并下载。

环境配置完后新建项目，在该项目中加载 MongoDB 驱动即可。加载驱动的方法每个开发语言有所差异，原理上都是把 MongoDB 驱动包引入项目中，让项目可以使用即可。

（3）查阅操作语法

如何查阅开发语言使用 MongoDB 的操作语法因人而异，有经验的开发者可以在项目中调用方法时查看方法和参数或者查看 MongoDB 驱动包的源码，新手可以在搜索引擎中搜索

相关资料，或者在官网中查看，例如 github[16]中就有相关操作的例子。

官网 MongoDB Drivers 的地址 https://docs.mongodb.com/ecosystem/drivers/ ，里面有 API 信息的链接，以及 github 的地址。github 中还提供不同语言的操作示例，如图 15-3 所示。

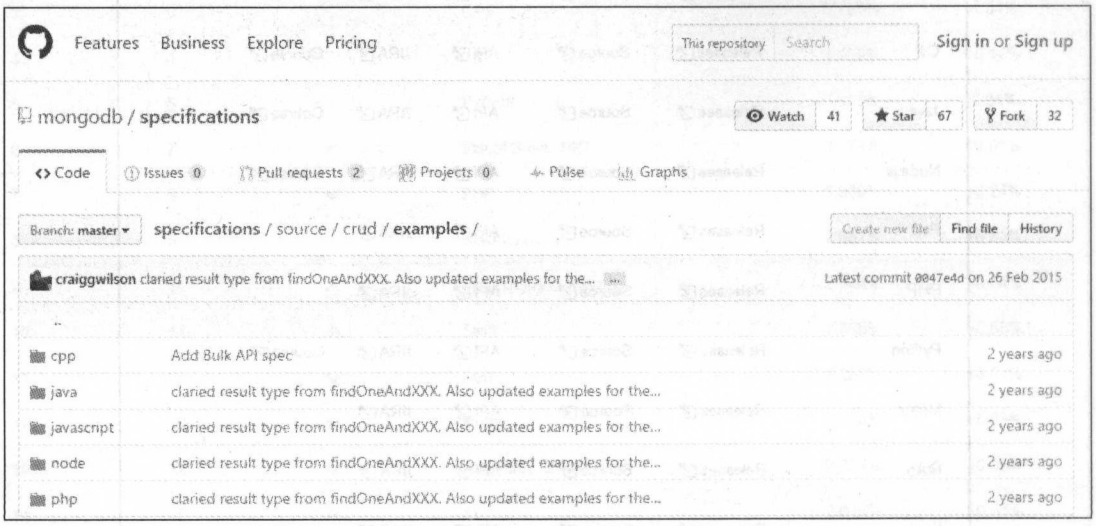

图 15-3　github 中不同语言的操作示例

（4）测试操作

加载好驱动之后根据查到的操作语法，尝试对 MongoDB 数据库做操作。主要完成连接数据库，增删改查，聚合操作 count、distinct、mapreduce 和 aggregate，以及对 GridFS 文件的操作，之后基本上就能使用这门语言进行开发工作了。

[16] GitHub 是一个面向开源及私有软件项目源码的托管平台，因为只支持 Git 作为唯一的版本库格式进行托管，故名 GitHub。

第 16 章

◂ Java操作MongoDB ▸

Java 是一门面向对象的编程语言，可以编写桌面应用程序、Web 应用程序、分布式系统和嵌入式系统应用程序等。要进行 Java 开发首先需要安装 Java 编辑器。目前有很多 Java IDE 编辑器，比较常用的有 Eclipse、NetBeans、IntelliJ IDEA Community Edition、Myeclipse 等。Eclipse 和 IntelliJ IDEA Community Edition 是免费的，NetBeans 是开源的，Myeclipse 则需要付费注册。我们这里选用 Eclipse。

16.1 安装 JDK

JDK 是整个 Java 开发的核心，它包含了 Java 的运行环境、Java 工具和 JAVA 基础的类库，所以在安装编辑器之前需要安装 JDK。

首先需要到官网下载 JDK，官网下载地址为：

```
http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-
2133151.html
```

根据自己计算机的环境选择版本即可。注意，最新版的 Eclipse Neon 至少需要 JDK 1.8 才能进行安装。我这里已经下载好了 jdk-8u121-windows-x64.exe，安装过程跟随向导单击 Next 按钮往下即可。

安装完成之后需要配置环境变量。

（1）新建变量名：JAVA_HOME（这是 JDK 安装路径）

变量值：C:\Program Files\Java\jdk1.8.0_121

（2）编辑变量名：Path

单击右下的编辑文本，在最后增加变量：;%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin

（3）新建变量名：CLASSPATH

变量值：.;%JAVA_HOME%\lib;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar

（注意：CLASSPATH 变量值前面有个"."）

设置完成之后，测试是否安装成功。重新打开 cmd 控制台，输入命令：java –version 或者 javac，打印出版本号或者命令提示则说明配置成功，如图 16-1 所示。

如果确认配置正确，但是不生效，可重启重试。

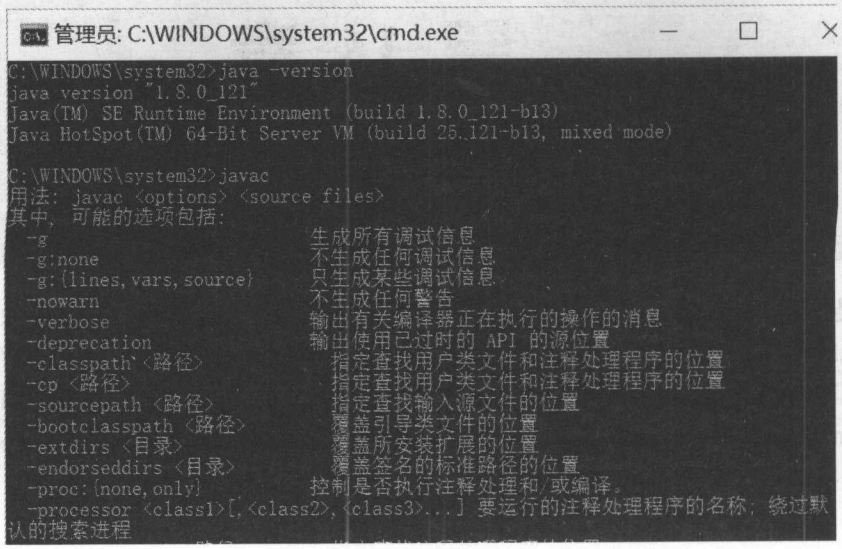

图 16-1　JDK 安装成功信息

16.2 Eclipse 安装

首先需要在 Eclipse 官网下载安装程序。Eclipse 官网下载地址：http://www.eclipse.org/downloads/。根据自己的计算机环境选择版本即可，这里选择 Win64 版本，如图 16-2 所示。

图 16-2　下载 Eclipse

下载完成后得到安装文件 eclipse-inst-win64.exe，双击运行，然后选择 Eclipse IDE for Java EE Developers，跟随向导完成安装即可。安装完成后启动，设置工作空间目录，也就是项目源代码存放的地方。

16.3 加载驱动

官方指向的 github 源码网站下载的 MongoDB 驱动包是驱动包的源码，有兴趣的读者可以自行研究，我们这里为了方便读者可以直接在其他第三方网站中下载 jar 包。

我们在第三方 jar 包管理网站 http://search.maven.org 中搜索 mongo-java-driver 下载 jar 包。如图 16-3 所示。

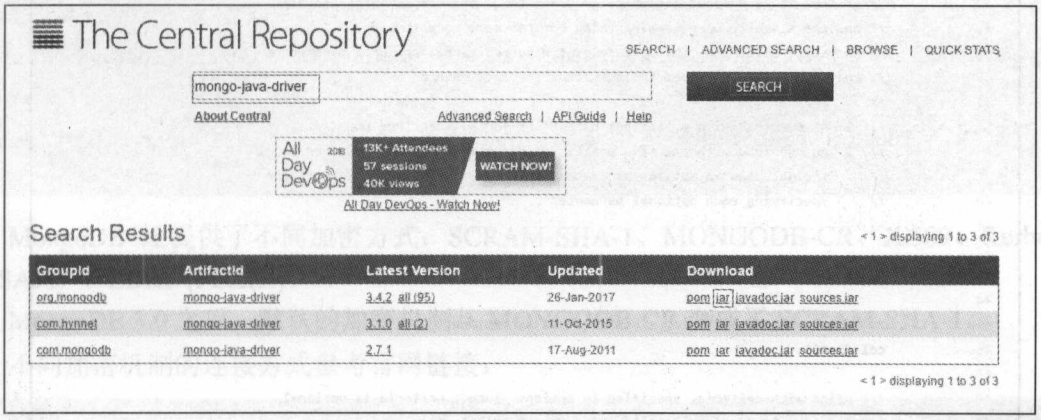

图 16-3　http://search.maven.org 驱动 jar 包下载

执行 File→New→Project→Java Project→填写项目名 test→Finish，新建一个 Java 项目后，把 jar 包复制粘贴放进项目中，对着 jar 包右击，单击 Build Path→Add to Build Path 即可，如图 16-4 所示。

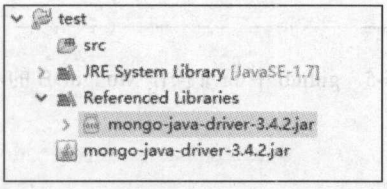

图 16-4　jar 包 Build Path 引入项目

16.4 查阅 Java 操作语法

github 中操作语法的文档链接是 http://mongodb.github.io/mongo-java-driver/3.4/driver/getting-started/quick-start/。

github 中 MongoDB 驱动在 Java 中的操作示例链接是 https://github.com/mongodb/specifications/blob/master/source/crud/examples/java/src/main/java/examples/MongoCollectionUsageExample.java 。github 中 Java 操作 MongoDB 的示例如图 16-5 所示。

```
import com.mongodb.client.model.UpdateOptions;
import org.mongodb.Document;

import static java.util.Arrays.asList;
import static java.util.concurrent.TimeUnit.SECONDS;

public class MongoCollectionUsageExample {

    public static void usageExample(MongoCollection<Document> col) {

        // Java's argument passing is very rudimentary compared to most other
        // languages.  No named parameters, no optional parameters, etc.
        // To make it as easy as possible for users to do the simple things simply
        // and the slightly more complex thing not too much more complex,
        // and still abiding by the spec rules for no positional optional parameters
        // the Java driver provides two overloads for each method:
        //
        //  1. An overload with positional parameters for all required model fields
        //  2. An overload with the same positional parameters for all required model
        //     fields, plus an options parameter whose type defines a fluent API for
        //     specifying each optional parameter


        // find

        //   find with no criteria, since criteria is optional
        col.find();

        //   find with criteria, specified in options since...criteria is optional
        col.find(new FindOptions().criteria(new Document("x", 1)));

        //   adding limit is easy once you have options
        col.find(new FindOptions().criteria(new Document("x", 1))
                                  .limit(10));

        //   as are query flags
        col.find(new FindOptions().tailable(true)
                                  .awaitData(true));

```

图 16-5　github 中 Java 操作 MongoDB 的示例

16.5 测试操作

我们先了解每种操作的用法，最后学习如何运行。使用代码时需要在 Java 文件中使用 import 引入相关的类，Eclipse 自动引入相关的快捷键是 Ctrl+Shift+O。

16.5.1 连接数据库

```
MongoClient mongoClient = new MongoClient("192.168.199.8" , 27017 );//创建数据库实体
MongoDatabase database = mongoClient.getDatabase("test"); //连接 test 数据库
MongoCollection<Document> collection = database.getCollection("user"); //获取 user 集合
```

或者使用 MongoClientURI：

```
MongoClient mongoClient = new MongoClient(new
MongoClientURI("mongodb://192.168.199.8:27017"));
```

开启数据库认证的情况下连接数据库：

```
String user; // 用户名
String database; // 存放 user 集合的数据库名
char[] password; //密码字符数组
MongoCredential credential = MongoCredential.createCredential(user, database,
password);
MongoClient mongoClient = new MongoClient(new ServerAddress("host1", 27017),
                                  Arrays.asList(credential));
```

MongoDB 还提供了不同加密方式：SCRAM-SHA-1、MONGODB-CR、X.509、Kerberos (GSSAPI) 、LDAP (PLAIN)。

MongoDB 3.0 之后，默认的加密机制从 MONGODB-CR 变成了 SCRAM-SHA-1。

不同加密机制的连接方式参考官网链接：

```
http://mongodb.github.io/mongo-java-driver/3.4/driver/tutorials/authentication/
```

16.5.2　插入数据

（1）单条插入

```
Document document = new Document("name", "user0")
        .append("contact", new Document("phone", "228-555-0149")
        .append("email", "user@example.com")
.append("location",Arrays.asList(-73.92502, 40.8279556)))
            .append("age",18)
            .append("likeNum", Arrays.asList(8, 10));
collection.insertOne(document);
```

（2）多条插入

```
List<Document> datas = new ArrayList<Document>();
for (int i=1; i < 10; i++) {
String userName="user"+i;
Document document = new Document("name",userName)
            .append("contact", new Document("phone", "228-555-0149"+i)
                               .append("email", userName+"@example.com")
                               .append("location",Arrays.asList(-73.92502+i,
40.8279556)))
            .append("age",18+i)
            .append("likeNum", Arrays.asList(i, 8, 9));
  datas.add(document);
```

```
    }
collection.insertMany(datas);
```

16.5.3 查询数据

（1）查询所有

```
collection.find();
```

（2）与查询

查询年龄大于等于 20，小于 21，喜欢数字有 9 的 user：

```
collection.find(new Document("age", new Document("$gte", 20).append("$lt",
21)).append("likeNum", 9));
```

除了文档的形式还能使用内置静态方法：

```
collection.find(and(gte("age", 20), lt("age", 21), eq("likeNum", 9)));
```

注意使用静态内置方法需要手动添加引入：

```
import static com.mongodb.client.model.Filters.*;
```

（3）或查询

查询年龄是 18 或者 19 的 user：

```
collection.find(or(eq("age", 18), eq("age", 19)));
```

（4）模糊查询

查询 name 字段值有 user 的文档：

```
Pattern pattern = Pattern.compile("user");
BasicDBObject query = new BasicDBObject("name", pattern);
collection.find(query);
```

（5）地理信息查询

首先得创建地理位置索引：

```
collection.createIndex(Indexes.geo2dsphere("contact.location"));
```

查询位置-73.92505, 40.8279556 附近 2 米之外，5000 米以内的点，Filters.near 方法传入 Bson geometry 类型时单位是米。

```
Point refPoint = new Point(new Position(-73.92505, 40.8279556));
collection.find(Filters.near("contact.location", refPoint, 5000.0, 2.0));
```

16.5.4 更新数据

（1）单条更新

updateOne 方法第一个参数是查询条件，如果查出多条，也只修改第一条；第二个参数是修改条件。

把年龄为 18 的第一个文档中 contact 子文档的 phone 字段修改为 000-2231-1289：

```
collection.updateOne(new Document("age", 18), new Document("$set", new
Document("contact.phone", "000-2231-1289")));
```

（2）多条更新

把年龄为 18 的所有文档 likeNum 数据中都增加一个数值 0：

```
collection.updateMany(
                eq("age", 18),
            new Document("$push",new Document("likeNum",0))
                );
```

16.5.5　删除数据

（1）删除单条记录

删除年龄为 22 的第一条文档：

```
collection.deleteOne(eq("age",22));
```

（2）删除多条记录

删除年龄为 18 的所有记录：

```
collection.deleteMany(eq("age",18));
```

16.5.6　聚合方法执行

（1）执行 count

查询年龄为 19 的文档数量：

```
collection.count(new BasicDBObject("age", 19));
```

（2）执行 distinct

查询 age 有多少不同值：

```
collection.distinct("age", Integer.class);
```

因为 age 在数据库中存储的是 Int32 类型，所以在 Java 中使用整型 Integer；如果是其他数据类型的话，class 类型需要对应。

查询 likeNum 中有 9 的文档 age 有多少不同值：

```
collection.distinct("age", new Document("likeNum", 9),Integer.class);
```

（3）执行 mapreduce

筛选出 name 为 user3 的文档：

```
String map = "function Map(){if(this.name=='user3'){emit('result',this);}}";
String reduce = "function Reduce(key, values) {return values[0];}";
MapReduceIterable<Document> out = collection.mapReduce(map,reduce);
        for(Document u:out){
            System.out.println(u);
        }
```

（4）执行 aggregate

likeNum 中有 9 的文档根据 age 分组后统计数量：

```
collection.aggregate(
     Arrays.asList(
           Aggregates.match(Filters.eq("likeNum", 9)),
           Aggregates.group("$age", Accumulators.sum("count", 1))
     )
);
```

16.5.7 操作 GridFS

（1）新建 GridFS 集合

新建一个存放 GridFS 文件的集合 files：

```
MongoClient mongoClient = new MongoClient("192.168.199.8" , 27017 );
MongoDatabase myDatabase = mongoClient.getDatabase("test");
GridFSBucket gridFSBucket = GridFSBuckets.create(myDatabase, "files");
```

（2）上传文件

上传文件需要指定上传路径和文件名，比如在 F 盘中新建一个 txt 文件 mongodb.txt：

```
try {
   InputStream streamToUploadFrom = new FileInputStream(new
File("F:/mongodb.txt"));
   // Create some custom options
   GridFSUploadOptions options = new GridFSUploadOptions()
    .chunkSizeBytes(358400)
  .metadata(new Document("type", "presentation"));
   ObjectId fileId = gridFSBucket.uploadFromStream("mongodb-tutorial",
streamToUploadFrom, options);
} catch (FileNotFoundException e){
  // handle exception
}
```

（3）查询文件

```
gridFSBucket.find();
```

或者增加查询条件：

```
gridFSBucket.find(eq("metadata.type", "presentation"));
```

（4）下载文件

下载文件需要存储在 GridFS 中的文件的 fileId，以及设置下载文件存放的地址和文件名，这里设置为 F:/mongodb_download.txt。

```
String fileName= "mongodb-tutorial";
//下载文件数据库中 files.files 的 filename
       try {
          FileOutputStream streamToDownloadTo = new
FileOutputStream("F:/mongodb_download.txt");
```

```
            gridFSBucket.downloadToStream(fileName, streamToDownloadTo);
            streamToDownloadTo.close();
            System.out.println(streamToDownloadTo.toString());
        } catch (IOException e) {
            // handle exception
        }
```

（5）重命名文件

```
ObjectId fileId=new ObjectId("58e0f8db6464ee74a4a25b96");
//下载文件数据库中 files.files 的_Id
gridFSBucket.rename(fileId, "newFileName");
```

（6）删除文件

```
ObjectId fileId=new ObjectId("58e0f8db6464ee74a4a25b96");
//下载文件数据库中 files.files 的_Id
gridFSBucket.delete (fileId);
```

16.5.8　运行示例

学习操作语法之后我们尝试运行它们，我们把需要执行的操作放在一个 Java 文件的 main 方法中运行。然后在 Eclipse 的 console 控制台以及客户端连接数据库查看结果。

对着项目 src 目录右击，选择 NEW→新建 Class 文件→命令为 Test→Finish。

Test.java 的代码如下：

```
package test;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import org.bson.Document;

import com.mongodb.MongoClient;
import com.mongodb.client.FindIterable;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import static com.mongodb.client.model.Filters.*;
public class Test {
    public static void main(String[] args) {
        //连接 test 数据库
        MongoClient mongoClient = new MongoClient("192.168.199.8" , 27017 );
        MongoDatabase database = mongoClient.getDatabase("test");
        MongoCollection<Document> collection = database.getCollection("user");
        //插入数据
        List<Document> datas = new ArrayList<Document>();
        for (int i=1; i < 10; i++) {
        String userName="user"+i;
        Document document = new Document("name",userName)
                    .append("contact", new Document("phone", "228-555-0149"+i)
```

```
    .append("email", userName+"@example.com")
    .append("location",Arrays.asList(-73.92502+i, 40.8279556)))
    .append("age",18+i)
    .append("likeNum", Arrays.asList(i, 8, 9));
    datas.add(document);
            }
        collection.insertMany(datas);
        //与查询并输出结果
        System.out.println("与查询:");
        FindIterable<Document> result=collection.find(new Document("age", new
Document("$gte", 20).append("$lt", 21)).append("likeNum", 9));
        System.out.println(result.first());


    }

}
```

Console 控制台输出结果为：

```
与查询:
Document{{_id=58e0e4ee6464ee61a05b5fab, name=user2,
contact=Document{{phone=228-555-01492, email=user2@example.com, location=[-
71.92502, 40.8279556]}}, age=20, likeNum=[2, 8, 9]}}
```

客户端中查看 user 集合已经有写入的 9 个文档。Robomongo 客户端中查看 user 集合界面如图 16-6 所示。

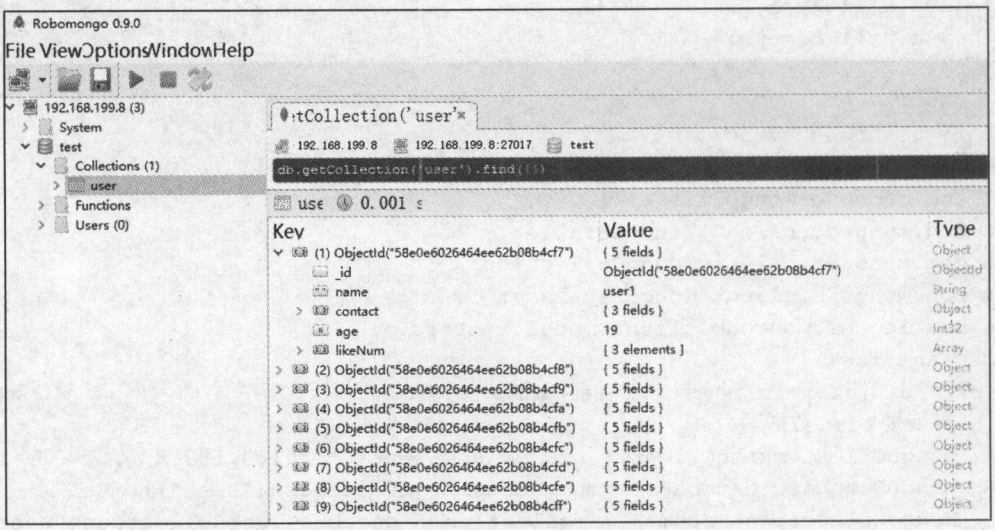

图 16-6　Robomongo 客户端中查看 user 集合

如果要执行其他操作，再修改 main 方法中的代码即可。

第三部分

管理与开发进阶篇

第 17 章

◂ 副本集部署 ▸

我们在第 6 章已经学习了有关副本集的理论原理，也知道了主从复制其实是副本集的一种，主从复制有比较明显的缺陷：当主节点出现故障停电或者死机等情况发生时，整个 MongoDB 服务集群就不能正常运作了，需要人工地去处理这种情况。所以在工作中很少使用主从复制了，一般都用副本集。所以本章节主要记录如何部署副本集，在最后小节会简单给出主从复制部署的运行命令。

17.1 总体思路

当副本集的总可投票数为偶数时，可能出现无法选举出主节点的情况，mongod 会提示：

```
[rsMgr] replSet total number of  votes is even - add artiber or give one member an extra vote。
```

2 个节点组成副本集是不合理的，因为这样的副本集不具备故障切换能力：

- 当 SECONDARY 节点挂掉，剩下一个 PRIMARY，此时副本集运行不会出问题，但不具备副本集的功能了，相当于单实例 MongoDB 服务。
- 当 PRIMARY 节点挂掉，此时副本集只剩下一个 SECONDARY，它只有 1 票，不超过总节点数的半数，不会成功选举自己为 PRIMARY 节点。会报错误提示：[rsMgr]replSet can't see a majority,will not try to elect self。

我们有两种方案，一是设置 MongoDB 集群的节点数量为奇数，二是当 MongoDB 集群的节点数量为偶数时，适当增加仲裁节点，增加集群的稳定性。

由此可知，算上仲裁节点，至少需要三个节点才能有效组成副本集。三个节点组成的副本集，当 PRIMARY 节点挂掉了，可以顺利选出一个 PRIMARY 节点，此时要马上修复挂掉的节点，因为不修复的话，如果当前的 PRIMARY 节点又挂了，剩下一个 SECONDARY 节点是不能选出 PRIMARY 节点的。

所以尝试部署副本集的可行方案是：3 个数据节点或 2 个数据节点+1 个仲裁节点。

我们这里记录 2 个数据节点+1 个仲裁节点的部署步骤，学会了之后就可以同理部署出 3 个数据节点甚至更多节点的副本集。

17.2 MongoDB 环境准备

根据我们的部署思路，我们需要有三台计算机作为 MongoDB 的服务器，2 台配置为数据节点，1 台配置为仲裁节点不保存数据。在生产环境中当时使用三台实体计算机是最好的，我们利用虚拟机来学习部署，实体机中的部署副本集步骤与虚拟机中是一样的。我们在 8.3 Linux 系统安装 MongoDB 小节中学习了虚拟机 Linux 系统的搭建，并且在上面安装了 MongoDB。我们已经有了一个虚拟机，我们可以用这个方法再搭建出两个虚拟机，或者使用克隆的方式快速复制出两个虚拟机。我这里就使用克隆的方式，在 VM 工具栏上单击虚拟机→管理→克隆，如图 17-1 所示，根据引导完成克隆。

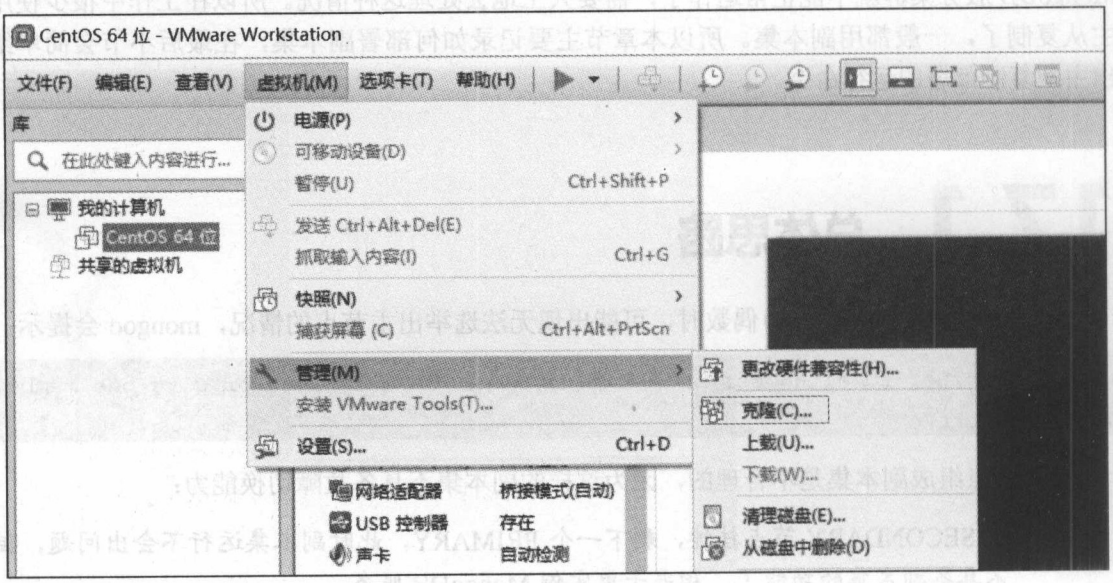

图 17-1　克隆虚拟机

（1）选择克隆自虚拟机中的当前状态。

（2）选择创建完整的克隆。

（3）填写虚拟机名称和选择虚拟机系统文件存放的路径。

（4）单击完成，稍等片刻就克隆完成了。

如果启动虚拟机时出现报错：The VMware Authorization Service is not running，需要在控制面板→管理工具→服务中找到 VMware Authorization Service 服务，启动 VMware Authorization Service 服务，并把它设置为自动。

这里一共准备了三个虚拟机。因为我的主机 Windows 系统的 ip 为 192.168.199.217，三个虚拟机应该与主机在同一个网段。修改静态 ip 时注意新克隆的虚拟机 MAC 地址（对着虚拟机右击，选择设置→网络适配器→高级可以查看）需要与 HWADDR 的值对应，如图 17-2 所示，同时 UUID 需要删除（重启后会自动生成）。它们的静态 ip 分别设置为：

- 192.168.199.8

- 192.168.199.9
- 192.168.199.10

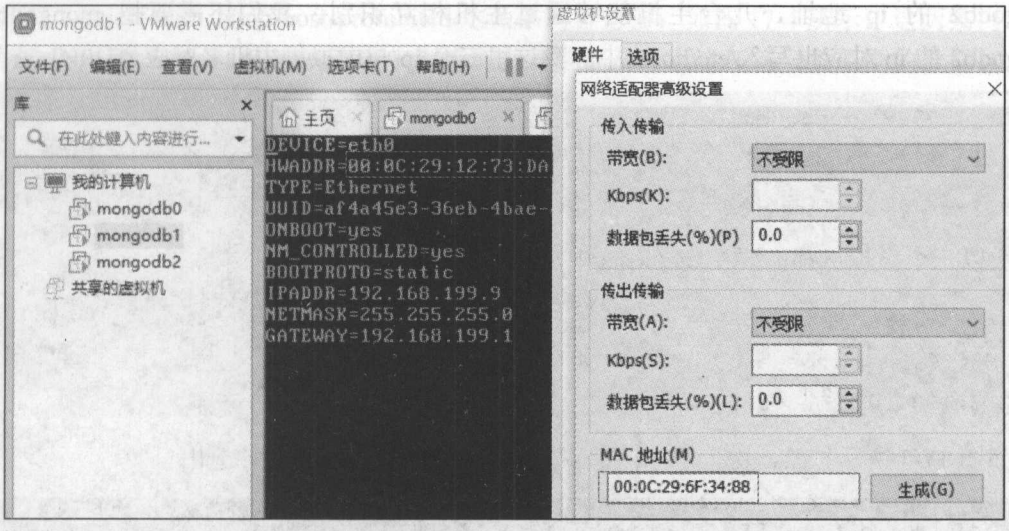

图 17-2　克隆的虚拟机 MAC 地址需要与 HWADDR 的值对应

设置好 ip 后还需要把 ip 与主机名对应起来。

查看主机名使用命令：

```
hostname
```

因为有两个虚拟机是克隆的，所以三个虚拟机的主机名一样，都为 localhost.mongodb0，我这里使用 vi 命令编辑/etc/sysconfig/network 和/etc/hosts 文件修改主机名，命令如下：

```
vi  /etc/sysconfig/network
```

把 HOSTNAME=localhost.mongodb0 中的 localhost.mongodb0 修改成我们设置的主机名。例如，mongodb0 机子这里修改为 HOSTNAME= mongodb0，按 Esc 键，输入:wq，按回车键，保存/etc/sysconfig/network 并退出。

```
vi  /etc/hosts
```

可以看到一列内容为：

```
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
```

需要把 localhost.localdomain 修改为我们设置的主机名，例如 mongodb0 机子这里修改为：

```
127.0.0.1   localhost  mongodb0  localhost4 localhost4.localdomain4
```

同时还需要把 ip 对应写入/etc/hosts 文件中，例如 mongodb0 机子在/etc/hosts 最后加入一行内容为：

```
192.168.199.8  mongodb0
```

mongo0 与自己的 ip 对应，为了保证 mongodb0 能够通过主机名找到 mongodb1 和 mongodb2 的 ip 地址，几台主机能够通过主机相互识别，我们还需要把 mongodb1 和 mongodb2 的 ip 对应也写入/etc/hosts 中。我这里三个 ip 与虚拟机主机名对应为：

```
192.168.199.8  mongodb0
192.168.199.9  mongodb1
192.168.199.10  mongodb2
```

所以三个虚拟机都需要在/etc/hosts 中加入内容：

```
192.168.199.8  mongodb0
192.168.199.9  mongodb1
192.168.199.10  mongodb2
```

如图 17-3 所示按 Esc 键，输入:wq，按回车键，保存/etc/hosts 并退出。

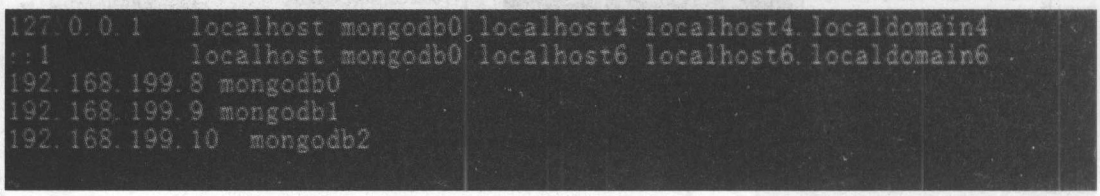

图 17-3　设置 ip 与主机名的映射

重启服务器后生效，使用 hostname 命令验证主机名，使用 ping 命令验证三个虚拟机相互之间是否可以使用主机名 ping 通。

然后还需要把 MongoDB 服务启动使用的端口在防火墙中放开。如果对安全不严格的测试环境或者内网环境可以关闭防火墙，如果是生产环境下不能关闭防火墙则需要打开 MongoDB 服务启动使用的端口，例如这里是默认的 27017。使用的命令如下：

（1）永久性生效，重启后不会复原

开启：chkconfig iptables on

关闭：chkconfig iptables off

（2）即时生效，重启后复原

开启：　service iptables start

关闭：　service iptables stop

需要说明的是，对于 Linux 下的其他服务都可以用以上命令执行开启和关闭操作。

在开启了防火墙时，做如下设置：开启相关端口，修改/etc/sysconfig/iptables 文件：

```
vi /etc/sysconfig/iptables
```

添加以下内容：

```
-A INPUT -m state --state NEW -m tcp -p tcp --dport 27017 -j ACCEPT
```

同时需要注意的是，-A INPUT -m state --state NEW -m tcp -p tcp --dport 27017 -j ACCEPT 这条内容应该放在-A INPUT -j REJECT --reject-with icmp-host-prohibited 和-A FORWARD -j REJECT --reject-with icmp-host-prohibited 这两条内容之前，如图 17-4 所示。这两条内容的意思是在 INPUT 表和 FORWARD 表中拒绝所有其他不符合上述任何一条规则的数据包。并且发送一条 host prohibited 的消息给被拒绝的主机。

```
# Firewall configuration written by system-config-firewall
# Manual customization of this file is not recommended.
*filter
:INPUT ACCEPT [0:0]
:FORWARD ACCEPT [0:0]
:OUTPUT ACCEPT [0:0]
-A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
-A INPUT -p icmp -j ACCEPT
-A INPUT -i lo -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT
-A INPUT -m state --state NEW -m tcp -p tcp --dport 27017 -j ACCEPT
-A INPUT -j REJECT --reject-with icmp-host-prohibited
-A FORWARD -j REJECT --reject-with icmp-host-prohibited
COMMIT
```

图 17-4　设置打开端口需放在 icmp-host-prohibited 之前

这个是 iptables 的默认策略，你也可以删除这两条内容确保打开端口有效。

然后重启防火墙：

```
service iptables restart
```

查看防火墙状态：

```
service iptables status
```

注意三个虚拟机的防火墙都需要打开 27017 端口或者直接关闭防火墙，然后确保三个虚拟机中的 MongoDB 能正常启动即可。

17.3 创建目录

mongodb0 使用命令：

```
mkdir -p /data/replset/r0   //机子0号存放数据的目录
mkdir -p /data/replset/key  //存放密钥的目录
mkdir -p /data/replset/log  //存放日记的目录
```

mongodb1 使用命令：

```
mkdir -p /data/replset/r1   //机子1号存放数据的目录
mkdir -p /data/replset/key  //存放密钥的目录
```

```
mkdir -p /data/replset/log  //存放日记的目录
```

mongodb2 使用命令：

```
mkdir -p /data/replset/r2   //机子2号存放数据的目录
mkdir -p /data/replset/key  //存放密钥的目录
mkdir -p /data/replset/log  //存放日记的目录
```

17.4 创建 Key

我们在第 13 章“安全和访问控制”中已经学习了单例的 MongoDB 服务想要开启安全认证只需要启动时加上--auth 参数即可，但是副本集中则需要设置 KeyFile 之后--auth 参数才有效。副本集认证的总体思路是用户名、密码和 KeyFile 文件，KeyFile 需要各个副本集服务启动时加载而且 KeyFile 文件的内容必须一致，然后在操作库时需要用户名、密码。

KeyFile 文件必须满足条件：

（1）至少 6 个字符，小于 1024 字节。

（2）认证的时候不考虑文件中空白字符。

（3）连接副本集成员的 KeyFile 和启动 mongos 进程的 KeyFile 文件内容必须一样。

（4）必须是 base64 编码，但是不能有等号。

（5）文件权限必须是 x00，也就是说，不能分配任何权限给 group 成员和 other 成员。

Keyfile 不是必选项，如果不需要 auth 认证功能，可以不带--keyfile 参数启动。如果有其中一个虚拟机用--keyfile 的方式启动，那其他机子没有这个 KeyFile 文件就无法加入副本集。

开启了 KeyFile，隐含就开启了 auth，这个时候连接副本集就需要进行认证了，否则只能通过本地例外方式操作数据库。

在副本集中添加用户需要在服务未加--keyFile 参数启动的情况下，按照单实例方法添加（访问任意一个副本器操作，其他副本集会自动同步），账户添加、授权成功后重新加入 keyFile 启动服务，即可完成并使用。

客户端连接 MongoDB 集群服务时进行认证与连接单服务器环境时认证步骤是一样的，密钥文件只是集群中服务器进行内部沟通时使用，不影响客户端连接 MongoDB 的认证。密钥文件基本上是一个明文的文件，Hash 计算后被当做集群的内部密码。

mongodb0 使用命令：

```
echo "replset1 key" > /data/replset/key/r0
chmod 600 /data/replset/key/r*
```

mongodb1 使用命令：

```
echo "replset1 key" > /data/replset/key/r1
chmod 600 /data/replset/key/r*
```

mongodb2 使用命令：

```
echo "replset1 key" > /data/replset/key/r2
chmod 600 /data/replset/key/r*
```

replset1 key 就是我们设置的明文密码，只要内容满足 Keyfile 的条件，读者可以随意设置，例如还可以设置成 my secret key 等。chmod 600 是把 Keyfile 文件的权限修改成仅用户可使用，防止其他程序改写此 KEY。不同版本的 MongoDB 对权限要求也不同，如果启动时报错 keyFilers0.key are too open，则是因为 mongo key 文件权限过大造成的。可以试试 chmod 400 /data/replset/key/r* 或者 chmod 300 /data/replset/key/r*。

17.5 初始化副本集

准备好了目录和 KeyFile 之后就可以启动了（如果之前 MongoDB 处于启动状态，则需要先把它关闭）。进入 mongod 可执行文件的目录下，使用命令如下。

Mongodb0 使用命令：

```
../mongod --dbpath=/data/replset/r0 --replSet replset1 --
logpath=/data/replset/log/r0.log --logappend --fork
```

mongodb1 使用命令：

```
./mongod --dbpath=/data/replset/r1 --replSet replset1 --
logpath=/data/replset/log/r1.log --logappend --fork
```

mongodb2 使用命令：

```
./mongod --dbpath=/data/replset/r2 --replSet replset1 --
logpath=/data/replset/log/r2.log --logappend --fork
```

参数--replSet 设置的是副本集的名称，我这里设置为 replset1，读者也可以起其他名字。如果需要加安全认证则需要分别加上参数--keyFile，参数如下：

Mongodb0 使用命令：

```
./mongod --dbpath=/data/replset/r0 --replSet replset1 --
logpath=/data/replset/log/r0.log --logappend --fork --keyFile
/data/replset/key/r0
```

mongodb1 使用命令：

```
./mongod --dbpath=/data/replset/r1 --replSet replset1 --
logpath=/data/replset/log/r1.log --logappend --fork --keyFile
/data/replset/key/r1
```

mongodb2 使用命令：

```
./mongod --dbpath=/data/replset/r2 --replSet  replset1 --
logpath=/data/replset/log/r2.log  --logappend  --fork  --keyFile
/data/replset/key/r2
```

使用--keyFile 模式启动，默认会启用--auth 认证模式。一般需要先在单例模式下创建好用户，再使用--keyFile 模式启动，认证后访问数据库。否则，在 mongo 客户端访问数据时会报错：

```
Error: error: {
        "ok" : 0,
        "errmsg" : "not authorized on test to execute command { find: \"say\",
filter: {} }",
        "code" : 13,
        "codeName" : "Unauthorized"
}
```

创建用户以及认证模式登录详见第 13 章“安全和访问控制”，我们这里就不使用--keyFile 模式了。

启动成功时输出信息为：

```
about to fork child process, waiting until server is ready for connections.
forked process: 1464
child process started successfully, parent exiting
```

进程数 forked process 可能不同，关键是要看到 started successfully。

如果出错，需要检查启动命令参数是否正确、标点符号是否正确，以及日志数据库文件存放路径是否存在等原因，或者可以尝试 9.7 节中给出的修复未正常关闭 MongoDB 的方法。

启动三个 MongoDB 服务之后，开始把它们初始化为副本集。在想要设置为 PRIMARY 节点的机子运行客户端 mongo。我们这里把 mongodb0 的机子设置为 PRIMARY 节点，在 mongodb0 的终端上 mongo 可执行文件所在的 bin 目录下输入命令：

```
mongo
```

进入 mongo 客户端之后输入副本集初始化命令：

```
rs.initiate()
```

只在 mongodb0 的机子上需要执行 rs.initiate()。

成功初始化显示输出如下：

```
{
        "info2" : "no configuration specified. Using a default configuration for
the set",
```

```
        "me" : "mongodb0:27017",
        "ok" : 1
}
```

如果初始化报错 No host described in new configuration 1 for replica set replset1 maps to this node 说明 ip 没有对应主机名，请参考 17.2 节“MongoDB 环境准备”中设置 ip 与主机名对应。完成报错信息显示如下：

```
{
        "info2" : "no configuration specified. Using a default configuration for
the set",
        "me" : "localhost.mongodb0:27017",
        "ok" : 0,
        "errmsg" : "No host described in new configuration 1 for replica set
replset1 maps to this node",
        "code" : 93,
        "codeName" : "InvalidReplicaSetConfig"
}
```

初始化成功之后可以查看副本集配置，在 mongo 客户端输入命令：

```
rs.conf()
```

输出信息为：

```
{
        "_id" : "replset1",
        "version" : 1,
        "protocolVersion" : NumberLong(1),
        "members" : [
                {
                        "_id" : 0,
                        "host" : "mongodb0:27017",
                        "arbiterOnly" : false,
                        "buildIndexes" : true,
                        "hidden" : false,
                        "priority" : 1,
                        "tags" : {

                        },
                        "slaveDelay" : NumberLong(0),
                        "votes" : 1
```

```
            }
        ],
        "settings" : {
                "chainingAllowed" : true,
                "heartbeatIntervalMillis" : 2000,
                "heartbeatTimeoutSecs" : 10,
                "electionTimeoutMillis" : 10000,
                "catchUpTimeoutMillis" : 2000,
                "getLastErrorModes" : {

                },
                "getLastErrorDefaults" : {
                        "w" : 1,
                        "wtimeout" : 0
                },
                "replicaSetId" : ObjectId("58d30b74fdced25c4fca25fd")
        }
}
```

初始化成功之后稍等一会就会发现 mongodb0:OTHER>变成了 mongodb0: PRIMARY>，说明 mongodb0 现在是 PRIMARY 节点的角色。

接着我们把两个节点加入进来。mongodb1 作为 SECONDARY 节点使用命令如下，在 mongo 客户端输入命令：

```
rs.add("mongodb1:27017")
```

需要两个参数：一个是主机名 mongodb1，另一个是 MongoDB 服务的端口号，我们这里启动时没有设置端口号，所以是默认的 27017。

如果报错 Quorum check failed because not enough voting nodes responded 说明 mongodb0 与 mongodb1 之间的 MongoDB 服务不能连通，原因有很多种，如下：

```
{
        "ok" : 0,
        "errmsg" : "Quorum check failed because not enough voting nodes
responded; required 2 but only the following 1 voting nodes responded:
mongodb0:27017; the following nodes did not respond affirmatively:
mongodb1:27017 failed with No route to host",
        "code" : 74,
        "codeName" : "NodeNotFound"
}
```

或者：

```
"errmsg" : "Quorum check failed because not enough voting nodes responded;
required 2 but only the following 1 voting nodes responded: mongodb0:27017;
the following nodes did not respond affirmatively:
mongodb1:27017 failed with Failed attempt to connect to mongodb1:27017;
couldn't connect to server mongodb1:27017,
connection attempt failed"
```

或者：

```
"errmsg" : "Quorum check failed because not enough voting nodes responded;
required 2 but only the following 1 voting nodes responded: mongodb0:27017;
the following nodes did not respond affirmatively:
mongodb1:27017 failed with not running with --replSet"
```

首先可以在 mongodb0 使用 ping 命令 ping 192.168.199.9 确认网络能连通，然后使用 ping mongodb1 命令确认域名能对应，防火墙需要关闭或者打开 MongoDB 服务所在的端口 27017。详细步骤请参照 17.2 节“MongoDB 环境准备”中设置 ip 与主机名对应。如果确认网络没问题，请确保 MongoDB 正常启动和端口的正确性。不想设置主机名的话，其实使用 ip 代替主机名作为添加参数也可以。例如这里可以使用命令 rs.add("192.168.199.9:27017")。

成功添加后输出信息：

```
{ "ok" : 1 }
```

mongodb2 作为 ARBITER 节点使用命令如下，在 PRIMARY 节点的 mongo 客户端输入命令：

```
rs.addArb("mongodb2:27017")
```

成功添加后输出信息：

```
{ "ok" : 1 }
```

才添加完 mongodb1 作为 SECONDARY 节点时需要启动的时间，mongodb0 会暂时降为 SECONDARY 节点。这时，如果运行 rs.addArb 命令，例如 replset1:SECONDARY> rs.addArb("mongodb2:27017")，会报错如下：

```
{
        "ok" : 0,
        "errmsg" : "replSetReconfig should only be run on PRIMARY, but my state
is SECONDARY; use the \"force\" argument to override",
        "code" : 10107,
        "codeName" : "NotMaster"
}
```

这种情况可以使用 rs.status()命令查看集群情况，新成员初始状态为"stateStr" : "(not

reachable/healthy)"，然后变成"stateStr" : "STARTUP"，然后变成"stateStr" : "RECOVERING"，最后当数据同步成功后，状态变为"stateStr" : "SECONDARY"。等待启动完毕再执行 rs.addArb 命令即可。如果一直处于 STARTUP 状态有可能是该节点到原 PRIMARY 节点的网络不通，请参考 17.2 节“MongoDB 环境准备”中的网络配置。

正确添加后查看副本集状态输出如下：

```
replset1:PRIMARY> rs.status()
{
        "set" : "replset1",
        "date" : ISODate("2017-03-23T21:18:51.624Z"),
        "myState" : 1,
        "term" : NumberLong(3),
        "heartbeatIntervalMillis" : NumberLong(2000),
        "optimes" : {
                "lastCommittedOpTime" : {
                        "ts" : Timestamp(1490303922, 1),
                        "t" : NumberLong(3)
                },
                "appliedOpTime" : {
                        "ts" : Timestamp(1490303922, 1),
                        "t" : NumberLong(3)
                },
                "durableOpTime" : {
                        "ts" : Timestamp(1490303922, 1),
                        "t" : NumberLong(3)
                }
        },
        "members" : [
                {
                        "_id" : 0,
                        "name" : "mongodb0:27017",
                        "health" : 1,
                        "state" : 1,
                        "stateStr" : "PRIMARY",
                        "uptime" : 71809,
                        "optime" : {
                                "ts" : Timestamp(1490303922, 1),
                                "t" : NumberLong(3)
                        },
```

```
        "optimeDate" : ISODate("2017-03-23T21:18:42Z"),
        "infoMessage" : "could not find member to sync from",
        "electionTime" : Timestamp(1490303835, 1),
        "electionDate" : ISODate("2017-03-23T21:17:15Z"),
        "configVersion" : 3,
        "self" : true
    },
    {
        "_id" : 1,
        "name" : "mongodb1:27017",
        "health" : 1,
        "state" : 2,
        "stateStr" : "SECONDARY",
        "uptime" : 69756,
        "optime" : {
              "ts" : Timestamp(1490303922, 1),
              "t" : NumberLong(3)
        },
        "optimeDurable" : {
              "ts" : Timestamp(1490303922, 1),
              "t" : NumberLong(3)
        },
        "optimeDate" : ISODate("2017-03-23T21:18:42Z"),
        "optimeDurableDate" : ISODate("2017-03-23T21:18:42Z"),
        "lastHeartbeat" : ISODate("2017-03-23T21:18:50.717Z"),
        "lastHeartbeatRecv" : ISODate("2017-03-23T21:18:47.719Z"),
        "pingMs" : NumberLong(0),
        "configVersion" : 3
    },
    {
        "_id" : 2,
        "name" : "mongodb2:27017",
        "health" : 1,
        "state" : 7,
        "stateStr" : "ARBITER",
        "uptime" : 8,
        "lastHeartbeat" : ISODate("2017-03-23T21:18:50.717Z"),
        "lastHeartbeatRecv" : ISODate("2017-03-23T21:18:47.761Z"),
        "pingMs" : NumberLong(0),
```

```
                "configVersion" : 3
            }
        ],
        "ok" : 1
}
```

到这里我们的副本集就启动好了，如果还要增加更多的 SECONDARY 节点，依旧使用 rs.add 命令即可，格式为 rs.add("主机名:端口")或者 rs.add("ip:端口")。如果还要增加更多的 ARBITER 节点，依旧使用 rs.addArb 命令即可，格式为 rs.addArb ("主机名:端口")或者 rs.addArb ("ip:端口")。

副本集的初始化除了先 rs.initiate()命令初始化，再使用 rs.add 等命令添加成员这种方式，还可以先新建一个 config 配置变量，然后使用 rs.initiate(config)命令初始化。使用 config 配置变量的好处是可以带参数，比如配置节点优先级等。例如在副本集未初始化之前，在 mongodb0 机子上进入 mongo 客户端，如下代码可以实现跟第一种方式初始化一样的效果，首先直接输入 config 配置变量如下：

```
config_replset1 =
 {
 _id:"replset1",
members:
[ {_id:0,host:"192.168.199.8:27017",priority:4},
{_id:1,host:"192.168.199.9: 27017",priority:2},
{_id:2,host:" 192.168.199.10: 27017",arbiterOnly : true}
]
}
```

然后使用带参数初始化命令：

```
rs.initiate(config_replset1);
```

17.6 数据同步测试

副本集部署好了之后，我们可以进行数据同步测试，测试副本集的数据同步是否生效，步骤如下：

首先向 PRIMARY（主节点）写入一条数据，如图 17-5 所示。在 PRIMARY 节点运行 mongo 客户端进入 replset1:PRIMARY>。

输入命令：

```
use test
```

```
db.say.insert({"text":"Hello World"})
db.say.find()
```

```
replset1:PRIMARY> use test
switched to db test
replset1:PRIMARY> db.say.insert({"text":"hello world"})
replset1:PRIMARY> db.say.findall()
Wed Feb 27 11:05:10 TypeError: db.say.findall is not a function (shell):1
replset1:PRIMARY> db.say.find()
{ "_id" : ObjectId("512d77dc7c6ad07653d0b308"), "text" : "hello world" }
replset1:PRIMARY>
```

图 17-5　PRIMARY 节点写入数据

进入 SECONDARY（副节点）查看数据是否同步。

默认情况下 SECONDARY 节点不能读写，要设定 slaveOk 为 true 才可以从 SECONDARY 节点读取数据。

replSet 里只能有一个 Primary 节点，只能在 Primary 写数据，不能在 SECONDARY 写数据。

首先需要在 SECONDARY 节点设置 slaveOk 为 true，在 SECONDARY 节点运行 mongo 客户端进入 replset1: SECONDARY>。

使用命令如下：

```
db.getMongo().setSlaveOk()
```

或者：

```
rs.slaveOk()
```

然后查看 say 集合，使用命令：

```
use test
db.say.find()
```

结果输出了我们在 PRIMARY 节点中写入的数据如下：

```
replset1:SECONDARY> db.say.find()
{ "_id" : ObjectId("58d4451f76daae4a7dde1607"), "text" : "Hello World" }
```

说明副本集数据同步有效。

如果未设置 slaveOk 为 true，会看到报错信息：

```
replset1:SECONDARY> db.say.find()
Error: error: {
        "ok" : 0,
        "errmsg" : "not master and slaveOk=false",
        "code" : 13435,
        "codeName" : "NotMasterNoSlaveOk"
}
```

我们也可以在 ARBITER 节点上测试数据的写入和读取，会发现 ARBITER 节点不能写入数据，也不能读取数据，因为仲裁节点只负责投票。

17.7 故障切换测试

副本集还有个很重要的功能就是故障切换，我们这里可以做故障测试，把主节点关闭，看看副集点是否能接替主节点进行工作。

在 PRIMARY 节点运行 mongo 客户端进入 replset1:PRIMARY>，输入命令：

```
use admin
db.shutdownServer()
```

这个时候我们在原 SECONDARY 节点 mongodb1 机子上进入 mongo 客户端，发现它已经变成了 replset1:PRIMARY>，说明故障切换成功。

使用 rs.status()查看副本集状态如下：

```
replset1:PRIMARY> rs.status()
{
        "set" : "replset1",
        "date" : ISODate("2017-03-23T22:18:33.302Z"),
        "myState" : 1,
        "term" : NumberLong(4),
        "heartbeatIntervalMillis" : NumberLong(2000),
        "optimes" : {
                "lastCommittedOpTime" : {
                        "ts" : Timestamp(1490307105, 1),
                        "t" : NumberLong(3)
                },
                "appliedOpTime" : {
                        "ts" : Timestamp(1490307512, 1),
                        "t" : NumberLong(4)
                },
                "durableOpTime" : {
                        "ts" : Timestamp(1490307512, 1),
                        "t" : NumberLong(4)
                }
        },
        "members" : [
                {
```

```
            "_id" : 0,
            "name" : "mongodb0:27017",
            "health" : 0,
            "state" : 8,
            "stateStr" : "(not reachable/healthy)",
            "uptime" : 0,
            "optime" : {
                  "ts" : Timestamp(0, 0),
                  "t" : NumberLong(-1)
            },
            "optimeDurable" : {
                  "ts" : Timestamp(0, 0),
                  "t" : NumberLong(-1)
            },
            "optimeDate" : ISODate("1970-01-01T00:00:00Z"),
            "optimeDurableDate" : ISODate("1970-01-01T00:00:00Z"),
            "lastHeartbeat" : ISODate("2017-03-23T22:18:31.922Z"),
            "lastHeartbeatRecv" : ISODate("2017-03-23T22:11:51.702Z"),
            "pingMs" : NumberLong(0),
            "lastHeartbeatMessage" : "Connection refused",
            "configVersion" : -1
      },
      {
            "_id" : 1,
            "name" : "mongodb1:27017",
            "health" : 1,
            "state" : 1,
            "stateStr" : "PRIMARY",
            "uptime" : 75384,
            "optime" : {
                  "ts" : Timestamp(1490307512, 1),
                  "t" : NumberLong(4)
            },
            "optimeDate" : ISODate("2017-03-23T22:18:32Z"),
            "electionTime" : Timestamp(1490307121, 1),
            "electionDate" : ISODate("2017-03-23T22:12:01Z"),
            "configVersion" : 3,
            "self" : true
      },
```

```
        {
                "_id" : 2,
                "name" : "mongodb2:27017",
                "health" : 1,
                "state" : 7,
                "stateStr" : "ARBITER",
                "uptime" : 3591,
                "lastHeartbeat" : ISODate("2017-03-23T22:18:31.827Z"),
                "lastHeartbeatRecv" : ISODate("2017-03-23T22:18:32.775Z"),
                "pingMs" : NumberLong(0),
                "configVersion" : 3
        }
    ],
    "ok" : 1
}
```

可以看到 mongodb0 的状态为"stateStr" : "(not reachable/healthy)"，mongodb1 已经成为 PRIMARY。

17.8 Java 程序连接 MongoDB 副本集测试

要充分利用 MongoDB 副本集的性能，则需要在 Java 程序使用时把副本集的节点都进行设置，如下代码测试了副本集的写入，三个节点有一个节点挂掉也不会影响应用程序客户端对整个副本集的读写。Java 代码如下：

```
public class TestMongoDBReplSet {
public static void main(String[] args) {
try {
List<ServerAddress> addresses = new ArrayList<ServerAddress>();
ServerAddress address1 = new ServerAddress("192.168.199.8" , 27017);
ServerAddress address2 = new ServerAddress("192.168.199.9" , 27017);
ServerAddress address3 = new ServerAddress("192.168.199.10" , 27017);
addresses.add(address1);
addresses.add(address2);
addresses.add(address3);
MongoClient client = new MongoClient(addresses);
DB db = client.getDB( "test");
DBCollection coll = db.getCollection( "testdb");
```

```
// 写入
BasicDBObject object = new BasicDBObject();
object.append( "test2", "testval2" );
coll.insert(object);
DBCursor dbCursor = coll.find();
while (dbCursor.hasNext()) {
DBObject dbObject = dbCursor.next();
System. out.println(dbObject.toString());
}
} catch (Exception e) {
e.printStackTrace();
}
}
}
```

上述写入副本集的代码其实还是主节点负责接受写的操作，副节点是不接受写操作的。副节点一般来说只负责默认地同步数据，如果副节点要用来读取的话，需要设置 SlaveOk=true。设置副本节点负责读操作，Java 代码如下：

```
public class TestMongoDBReplSetReadSplit {
public static void main(String[] args) {
try {
List<ServerAddress> addresses = new ArrayList<ServerAddress>();
ServerAddress address1 = new ServerAddress("192.168.199.8" , 27017);
ServerAddress address2 = new ServerAddress("192.168.199.9" , 27017);
ServerAddress address3 = new ServerAddress("192.168.199.10" , 27017);
addresses.add(address1);
addresses.add(address2);
addresses.add(address3);
MongoClient client = new MongoClient(addresses);
DB db = client.getDB( "test" );
DBCollection coll = db.getCollection( "testdb" );
BasicDBObject object = new BasicDBObject();
object.append( "test2" , "testval2" );
//读操作从副本节点读取
ReadPreference preference = ReadPreference. secondary();
DBObject dbObject = coll.findOne(object, null , preference);
System. out .println(dbObject);
} catch (Exception e) {
e.printStackTrace();
```

```
}
}
}
```

读参数除了 secondary 一共还有五个参数：primary、primaryPreferred、secondary、secondaryPreferred、nearest，分析如下。

- primary：默认参数，只从主节点上进行读取操作。
- primaryPreferred:大部分从主节点上读取数据，只有主节点不可用时从 secondary 节点读取数据。
- secondary：只从 secondary 节点上进行读取操作，存在的问题是 secondary 节点的数据会比 primary 节点数据“旧”。
- secondaryPreferred：优先从 secondary 节点进行读取操作，secondary 节点不可用时从主节点读取数据。
- nearest：不管是主节点、secondary 节点，从网络延迟最低的节点上读取数据。

读写分离做好了能够实现数据分流，减轻服务器的压力。

17.9 主从复制部署

MongoDB 的主从复制其实很简单，就是在运行主节点的服务器上开启 mongod 进程时，加入参数--master 即可；在运行从节点的服务器上开启 mongod 进程时，加入--slave 和--source 参数指定主节点即可。这样，在主数据库更新时，数据被复制到从数据库中。

详细部署步骤如下。

创建目录，mongodb0 使用命令：

```
mkdir -p  /mongodb/db/master
mkdir -p  /mongodb/log
```

mongodb1 使用命令：

```
mkdir -p  /mongodb/db/slave
mkdir -p  /mongodb/log
```

mongodb0 作为主节点启动使用命令：

```
mongod --dbpath=/mongodb/db/master  --fork --logpath=/mongodb/log/master.log  --master
```

mongodb1 作为从节点启动使用命令：

```
mongod --dbpath=/mongodb/db/slave  --fork --logpath=/mongodb/log/slave.log  --
```

```
slave  --source= mongodb0:27017
```

测试主从数据库数据同步，在主节点插入数据，在从节点查看数据是否同步成功。

在主节点进入 mongo 客户端运行命令：

```
use test
db.say.save({line:"hello world"})
```

在从节点进入 mongo 客户端运行命令：

```
use test
db.getMongo().setSlaveOk()
db.say.find()
```

输出信息如下：

```
> use test
switched to db test
> db.say.find()
Error: error: {
        "ok" : 0,
        "errmsg" : "not master and slaveOk=false",
        "code" : 13435,
        "codeName" : "NotMasterNoSlaveOk"
}
> db.getMongo().setSlaveOk()
> db.say.find()
{ "_id" : ObjectId("58d45046025492ae07eb36ca"), "line" : "hello world" }
```

主从部署成功。