第 18 章

◂ 分片部署 ▸

我们在第 7 章了解 MongoDB 分片中学习了 MongoDB 分片的简介和自动分片的原理，MongoDB 分片对 MongoDB 存储的数据做分流，可以分担单台服务器的压力，本章就尝试部署 MongoDB 分片。

18.1 总体思路

部署一个分片集群需要 3 个部分：

（1）Shard Server

Shard Server 是实际存储数据的数据库，就是传说中的分片，每个分片保存所有数据的一部分。每个 Shard 可以是一个 MongoDB 实例，也可以是一组 MongoDB 实例构成集群——副本集（Replica Set）。MongoDB 官方建议每个 Shared 最好是一组副本集，这样具有更好的容灾性。我们将在第 19 章把分片与副本集结合部署。本章先使用单例 MongoDB 实例作为分片。我们有三个虚拟机，初步设计为做三个分片。

（2）Config Server

Config Server 是配置服务器，存储所有 Shard 节点的配置信息、每个 Chunk（块）的 Shard Key 范围、Chunk 在各个 Shard 的位置，以及分片集群中所有数据库 db 和所有集合（collection）的 sharding（分片）配置信息。mongos 本身没有物理存储分片服务器和数据路由信息，只是缓存在内存里；配置服务器则实际存储这些数据。mongos 第一次启动或者关掉重启就会从 Config Server 加载配置信息，以后如果配置服务器信息变化会通知到所有的 mongos 更新自己的状态，这样 mongos 就能继续准确路由。在生产环境通常需要多个 Config Server，因为它存储了分片路由的元数据，这个可不能丢失！多个 Config Server 保存的是一样的信息，就算挂掉其中一台，只要还有其他 Config Server， MongoDB 分片集群就能保持正常工作。MongoDB 3.2 版本之后 Config Server 支持部署成副本集，更多了一层保障，MongoDB 3.2 版本之前可以水平部署多个 Config Server，但是 MongoDB 3.4 版本多个 Config Server 必须配置成副本集。所以我们需要配置 3 个 Config Server 组成的副本集。

（3）Route Process

mongos 是一个 Route Process，为数据路由器、应用程序与数据库集群交互的入口，所有的请求都通过 mongos 进行协调，不需要在应用程序添加一个路由选择器，mongos 自己就是

一个请求分发中心，它负责把对应的数据请求转发到对应的 shard 服务器上。应用程序使用分片集群与使用单例 MongoDB 服务器是一样的用法，应用程序只需要将查询、保存、更新请求原封不动地发送给 mongos，mongos 会去询问 Config Server 是否需要在 Shard 上查询、保存、更新，然后再连接到相应的 Shard 进行操作，最后将操作结果返回给应用程序。应用程序本身不要具体把操作发给 Shard，只需要把操作请求发送给 mongos 就可以了。应用程序在生产环境通常用多 mongos 作为请求的入口，防止其中一个挂掉，所有的 mongodb 请求都没有办法操作。

图 18-1 分片基本架构是最基本的架构，但是单个 mongos 路由服务和单个 Config Server 存在致命的问题，如果它们中有一个挂了，集群就不可用了。所以我们在本章节采用如图 18-2 所示的分片架构。

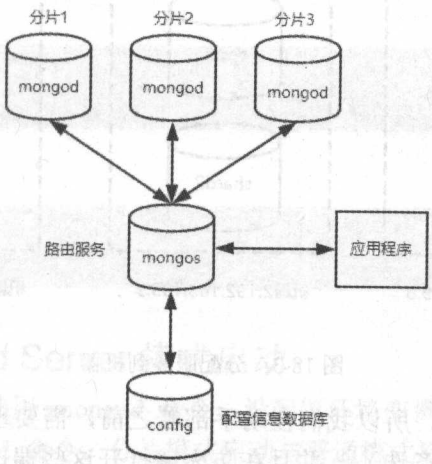


图 18-1　分片基本架构

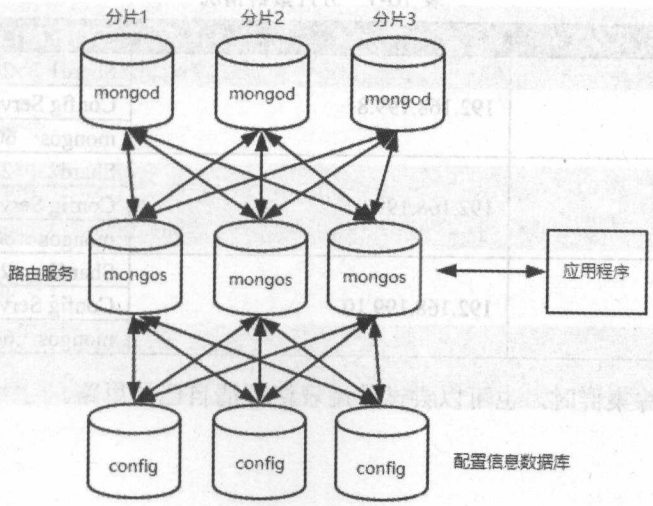


图 18-2　生产环境可用的分片架构

也就是部署 3 个单例 MongoDB 作为 Shard，部署 3 个 mongos 作为路由，部署三个

Config Server 作为配置信息的数据库服务。

理想情况下我们需要 9 台计算机，但是这样成本太高了，而且 mongos 和 Config Server 本身并不存储应用程序的数据，所以 mongos 和 Config Server 可以和 Shard 共用一台计算机，使用不同端口即可（同一台计算机以不同端口可以启动多个 MongoDB 实例）。我们这里刚好有三个虚拟机，它们的 ip 分别为 192.168.199.8、192.168.199.9、192.168.199.10。所以我们可以采用如图 18-3 所示的分配情况。

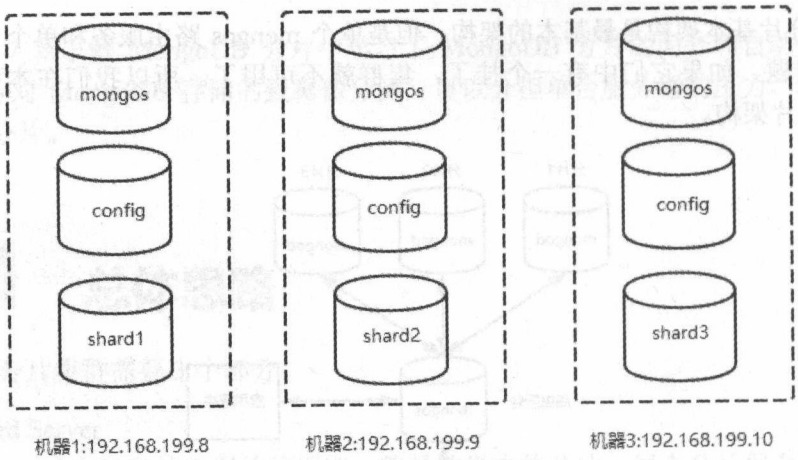


图 18-3　分配服务到机器

由于会使用不同的端口，所以我们在动手部署之前，需要理清楚使用哪些端口（只要不与其他端口冲突，端口号随意选），并且在防火墙打开这些端口，防火墙打开端口的具体步骤可以参考 17.2 MongoDB 环境准备中的网络配置。我的分片集群情况如表 18-1 所示。

表 18-1　分片集群情况

主机	IP	服务和端口
mongodb0	192.168.199.8	Shard1　28000
		Config Server　40000
		mongos　60000
mongodb1	192.168.199.9	Shard2　28000
		Config Server　40000
		mongos　60000
mongodb2	192.168.199.10	Shard3　28000
		Config Server　40000
		mongos　60000

大家在部署分片集群时，也可以尝试使用表格理清自己的思路。

18.2 创建 3 个 Shard Server

我们第一步就来先部署 Shard Server，也就是 3 个单例 MongoDB 实例。

18.2.1 创建目录

mongodb0 使用命令：

```
mkdir  -p  /data/shard1
mkdir  -p  /data/shard1/log
```

mongodb1 使用命令：

```
mkdir  -p  /data/shard2
mkdir  -p  /data/shard2/log
```

mongodb2 使用命令：

```
mkdir  -p  /data/shard3
mkdir  -p  /data/shard3/log
```

18.2.2 以分片 Shard Server 模式启动

已配置环境变量可直接使用 mongod 命令，没配置环境变量需要进入 mongod 执行文件所在 bin 目录，使用./mongod 命令。分片模式启动与普通模式启动的区别在于使用--shardsvr 参数。

mongodb0 使用命令：

```
mongod --dbpath=/data/shard1 --port 28000 --
logpath=/data/shard1/log/shard1.log --logappend --fork  --shardsvr
```

mongodb1 使用命令：

```
mongod --dbpath=/data/shard2 --port 28000 --
logpath=/data/shard2/log/shard2.log --logappend --fork  --shardsvr
```

mongodb2 使用命令：

```
mongod --dbpath=/data/shard3 --port 28000 --
logpath=/data/shard3/log/shard3.log --logappend --fork  --shardsvr
```

18.3 启动 Config Server

部署 Config Server，Config Server 也是使用 mongod 启动，需要使用--configsvr 参数作为 Config Server 的识别。因为 MongoDB 3.4 多个 Config Server 需要配置成副本集，所以还需要加上参数--replSet replsetConfig。

18.3.1 创建目录

mongodb0 使用命令：

```
mkdir -p /data/shard/configdb1
```

mongodb1 使用命令：

```
mkdir -p /data/shard/configdb2
```

mongodb2 使用命令：

```
mkdir -p /data/shard/configdb3
```

18.3.2 以分片 Config Server 模式启动

mongodb0 使用命令：

```
mongod --dbpath /data/shard/configdb1 --port 40000 --
logpath=/data/shard1/log/config1.log --fork --configsvr --
replSet replsetConfig
```

mongodb1 使用命令：

```
mongod --dbpath /data/shard/configdb2 --port 40000 --
logpath=/data/shard2/log/config2.log --fork --configsvr --
replSet replsetConfig
```

mongodb2 使用命令：

```
mongod --dbpath /data/shard/configdb3 --port 40000 --
logpath=/data/shard3/log/config3.log --fork --configsvr --
replSet replsetConfig
```

MongoDB 3.4 版本多个 Config Server 只能使用 Config Server 副本集配置，所以我们还需要把多个 Config Server 初始化成副本集。

在 mongodb0 的 mongo 客户端中初始化副本集，需要注意的是，连接 mongo 时应该连接端口 40000 的 mongod 实例，使用命令：

```
mongo --port 40000
```

然后初始化副本集，使用命令：

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

稍等主节点启动成功后分别添加其他两个 Config Server 节点为副节点，使用命令：

```
rs.add("192.168.199.9:40000");
rs.add("192.168.199.10:40000");
```

18.4 启动 Route Process

Route Process 路由服务是使用 mongos 来启动的，需要设置--configdb 参数和 chunkSize 参数。--configdb 则是指定与 mongos 交互的 Config Server，多个 Config Server 时用逗号隔开即可。在 MongoDB 3.2 和之前的版本中，可以使用 --configdb 192.168.199.8:40000,192.168.199.9:40000,192.168.199.10:40000 命令指定多个非副本集的 Config Server，但 MongoDB 3.4 版本多个 Config Server 只能使用 Config Server 副本集配置了，使用参数--configdb 副本集名/ip:端口，ip:端口...我这里副本集名是 replsetConfig，所以使用参数为--configdb replsetConfig/192.168.199.8:40000,192.168.199.9:40000,192.168.199.10:40000，否则会报错：

```
FailedToParse: mirrored config server connections are not supported; for
config server replica sets be sure to use the replica set connection string。
```

--chunkSize 参数是可选参数，MongoDB 3.4 版本默认为 64MB，我们这里为了测试分片和块的拆分，把--chunkSize 设置为 1MB，生产环境保持默认即可。旧的版本中 chunkSize 的设置是在 mongos 启动时加上--chunkSize 1 即可，但是 MongoDB 3.4 在 mongos 启动时已经去掉了这个配置参数，会报错 Error parsing command line: unrecognised option '--chunkSize'。需要在启动 mongos 成功之后再设置。

mongos 启动命令分别如下。

mongodb0 使用命令：

```
mongos --configdb  replsetConfig/192.168.199.8:40000,192.168.199.9:40000,192.
168.199.10:40000 --port  60000  --logpath=/data/shard1/log/mongos.log --fork
```

mongodb1 使用命令：

```
mongos --configdb  replsetConfig/192.168.199.8:40000,192.168.199.9:40000,192.
168.199.10:40000 --port  60000  --logpath=/data/shard2/log/mongos.log --fork
```

mongodb2 使用命令：

```
mongos --configdb
```

```
replsetConfig/192.168.199.8:40000,192.168.199.9:40000,192.168.199.10:40000 --
port  60000  --logpath=/data/shard3/log/mongos.log -fork
```

启动成功后会输出：

```
about to fork child process, waiting until server is ready for connections.
forked process: 2300
child process started successfully, parent exiting
```

如果一直卡着没有输出，请检查启动语句，尤其是 Config Server 副本集名称。

为了测试方便，启动成功后，我们把 chunkSize 修改成 1MB，MongoDB 3.4 版本的 chunkSize 不在启动参数中设置，需要在 mongo 客户端设置，mongo 客户端进入 mongos 服务所在的端口，使用命令：

```
mongo --port 60000
use config
db.settings.save( { _id:"chunksize", value: 1 } )
```

三台机子都执行上述的命令，也就是三个 mongos 服务都修改 chunkSize 的大小。

查看 chunkSize 值的大小使用如下命令：

```
use config
db.settings.find()
```

18.5 配置 sharding

mongos 路由启动好了之后，就可以进行 sharding 的配置了，也就是添加分片。mongo 客户端进入 mongos 服务所在的端口，在其中一台机子中进入 mongos>使用命令：

```
mongo --port 60000
```

需要先切换到 admin 数据库，才能进行分片的添加，使用如下命令：

```
use admin
db.runCommand({addshard:"192.168.199.8:28000" })
db.runCommand({addshard:"192.168.199.9:28000" })
db.runCommand({addshard:"192.168.199.10:28000" })
```

或者使用命令：

```
sh.addShard( "192.168.199.8:28000")
sh.addShard( "192.168.199.9:28000")
sh.addShard( "192.168.199.10:28000")
```

添加成功输出如下：

```
mongos> use admin
switched to db admin
mongos> db.runCommand({addshard:"192.168.199.8:28000" })
{ "shardAdded" : "shard0000", "ok" : 1 }
mongos> db.runCommand({addshard:"192.168.199.9:28000" })
{ "shardAdded" : "shard0001", "ok" : 1 }
mongos> db.runCommand({addshard:"192.168.199.10:28000" })
{ "shardAdded" : "shard0002", "ok" : 1 }
```

到这里我们对分片的部署就完成了。

在没启用安装认证的情况下，我们可以使用第三方工具来连接查看分片集群中的数据，这样可以更好地了解分片的工作机制。

例如，我使用 Robomongo 工具根据 shard 和 Config Server 以及 mongos 的 ip 和端口来连接它们，可以看到如图 18-4 所示的情况。

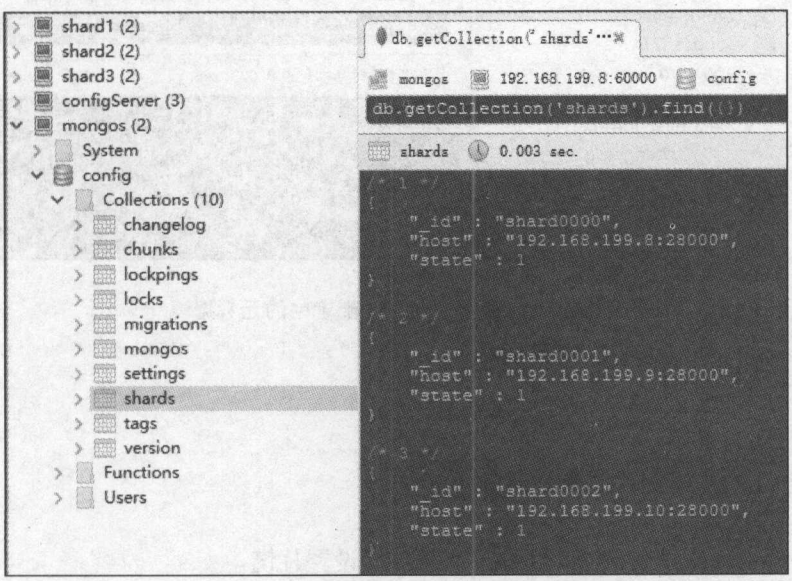


图 18-4　Robomongo 工具中查看分片集群

18.6 对数据库 mytest 启用分片

在任意机子中进入 mongos，比如在 mongodb0 机子中使用命令：

```
mongo  --port  60000
```

然后切换到 admin 数据库，对 mytset 数据库设置分片，因为 MongoDB 中的数据库都会自动创建，所以我们这里不需要先新建 mytest 数据库。

```
use admin
db.runCommand({enablesharding:"mytest"})
```

或者：

```
sh.enableSharding("mytest")
```

设置成功输出：

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

此时分片集群 config 数据库中多了一个 databases 集合，记录了数据库分片的元数据如图 18-5 所示。

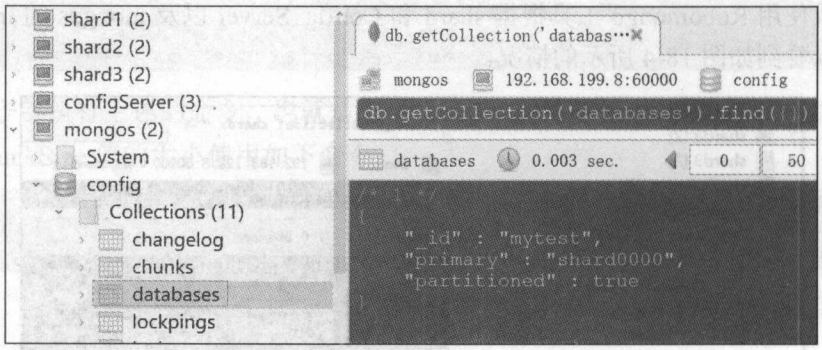


图 18-5　数据库分片在配置中的元数据

18.7 集合启用分片

对数据库 mytest 中的集合 student 启用分片，设置片键。

在任意机子中进入 mongos，比如在 mongodb0 机子中使用命令：

```
mongo  --port  60000
```

对 mytest 数据库中的 student 集合启用分片和设置片键，因为 MongoDB 中的集合会自动创建，所以我们这里不需要先新建 student 集合。MongoDB 3.4 版本中设置片键有三种策略：哈希、区间和标签。

片键可以根据集合字段的情况进行选择，一般来说可以使用_id。我们这里以_id 字段为例。

1. 区间片键方式

把 mytest 数据库 student 集合的_id 字段设置为区间片键，使用命令：

```
sh.shardCollection( "mytest.student", { _id:1 } )
```

2. 哈希片键方式

把 mytest 数据库 student 集合的_id 字段设置为哈希片键，使用命令：

```
sh.shardCollection( "mytest.student", { _id: "hashed" } )
```

3. 标签片键方式

标签片键比较特别，通过对分片节点打标签，再将片键按范围对应到这些标签上，对应片键范围的集合中的数据就会落在这个分片节点上。

首先需要对分片打标签，对分片打标签需要知道分片的_id，在 mongos>中使用命令查看：

```
sh.status()
```

输出内容为：

```
--- Sharding Status ---
  sharding version: {
        "_id" : 1,
        "minCompatibleVersion" : 5,
        "currentVersion" : 6,
        "clusterId" : ObjectId("58d4fa662b45e2a349b15aab")
}
  shards:
        { "_id" : "shard0000", "host" : "192.168.199.8:28000", "state" : 1 }
        { "_id" : "shard0001", "host" : "192.168.199.9:28000", "state" : 1 }
        { "_id" : "shard0002", "host" : "192.168.199.10:28000", "state" : 1 }
  active mongoses:
        "3.4.2" : 3
 autosplit:
        Currently enabled: yes
  balancer:
        Currently enabled:  yes
        Currently running:  no
              Balancer lock taken at Fri Mar 24 2017 18:52:23 GMT+0800 (CST) by
ConfigServer:Balancer
        Failed balancer rounds in last 5 attempts:  0
        Migration Results for the last 24 hours:
              No recent migrations
  databases:
        { "_id" : "mytest", "primary" : "shard0000", "partitioned" : true }
```

这里分片_id 分别为 shard0000、shard0001、shard0002,，则对分片打标签命令如下：

```
sh.addShardTag("shard0000", "tag1")
sh.addShardTag("shard0001", "tag2")
sh.addShardTag("shard0002", "tag2")
```

然后 student 的学号字段 code 设置标签片键范围，使用命令：

```
sh.addTagRange("mytest.student", { code: "1" }, { code: "600" }, "tag2")
sh.addTagRange("mytest.student", { code: "601" }, { code: "5000" }, "tag1")
```

这样设置后，学号 code 为 1~600 的会存储在分片 shard0001 或者 shard0002 中，学号为 601~5000 的会存储在分片 shard0000 中。

查看带有某个标签的分片，在 mongos>中使用命令：

```
use config
db.shards.find({ tags: "tag2" })
```

查看标签的片键值范围，使用命令：

```
use config
db.tags.find({ tags: "tag1" })
```

移除某个分片的标签使用命令如下：

```
sh.removeShardTag("shard0002", "tag2")
```

移除集合数据的标签片键范围使用命令如下：

```
use config
db.tags.remove({ _id: { ns: "mytest.student", min: { code: "1" }}, tag:
"tag2" })
```

18.8 分片集群插入数据测试

在 mongos>中使用命令：

```
use mytest
for(var i=1; i<=600000; i++)
db.student.insert({age:i,name:"mary",addr:"guangzhou",country:"China"})
```

执行可能需要一小会时间，我们直接在工具中查看 config 集合中的 chunks 元数据信息，就可以看到随着数据的写入分了多少片，以及片键的最小到最大的范围，如图 18-6 所示。

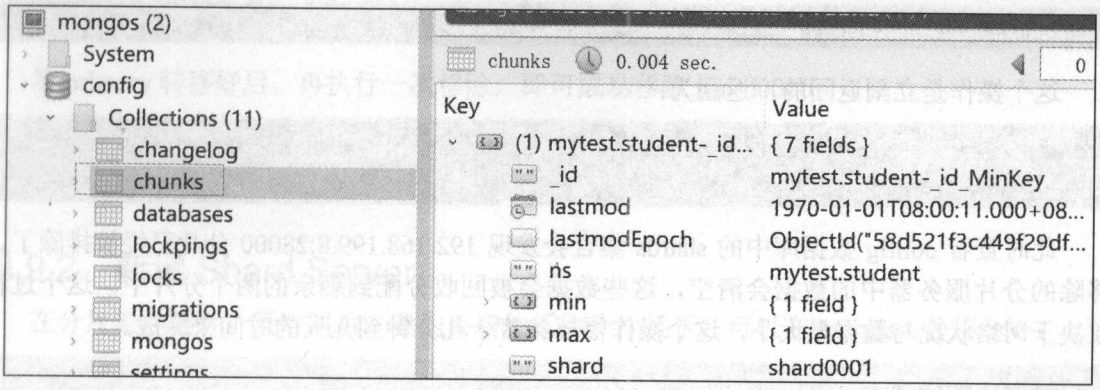


图 18-6　分片集群中 chunk 在 config 数据库中的元数据

3 个 shard 服务器里可以看到都有了 mytest 数据库以及 student 集合，存储数据成功后界面如图 18-7 所示。

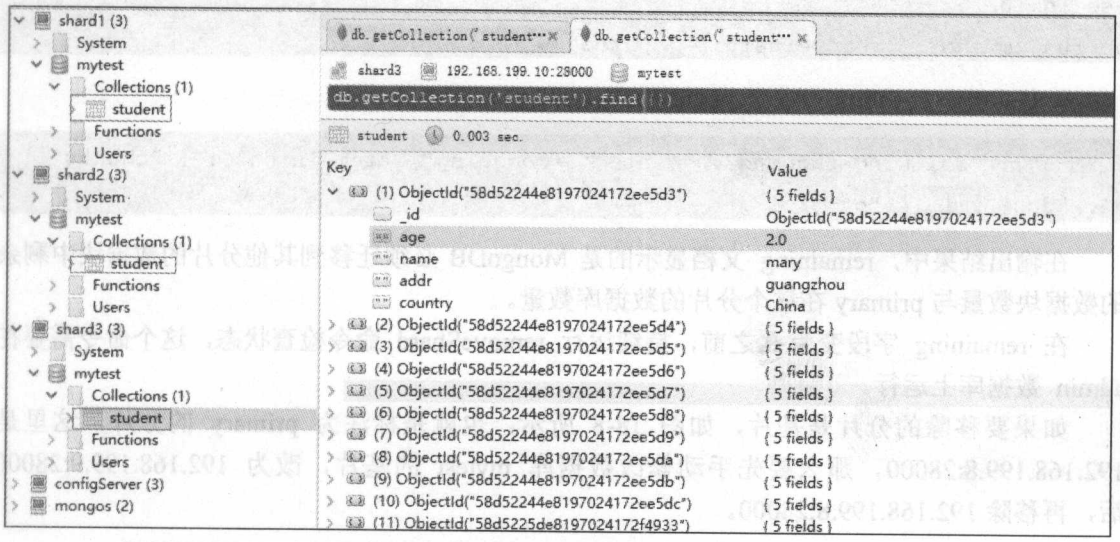


图 18-7　分片存储数据成功

到这里分片就部署成功，功能测试通过了。

18.9 分片的管理

18.9.1 移除 Shard Server，回收数据

在 mongos>中使用命令：

```
use admin
```

```
db.runCommand({"removeshard" : "192.168.199.8:28000"})
```

这个操作是立刻返回的，返回为：

```
{ "msg" : "draining started successfully", "state" : "started", "shard" :
"shard0000", "ok" : 1}
```

此时查看 config 数据库中的 shards 集合会发现 192.168.199.8:28000 分片已经被排除了。移除的分片服务器中的数据会清空，这些数据会被回收分配到剩余的两个分片中。这个过程取决于网络状况与数据量大小，这个操作需要花费十几分钟到几天的时间来完成。

检查迁移的状态

检查迁移的状态，再次在 admin 数据库运行 removeShard 命令，比如，对一个命名为 shard0000 的分片，运行命令：

```
use admin
db.runCommand( { removeShard: " shard0000" } )
```

这条命令返回类似如下的输出：

```
{ "msg" : "draining ongoing", "state" : "ongoing", "remaining" : { "chunks" :
42, "dbs" : 1 }, "ok" : 1}
```

在输出结果中，remaining 文档显示的是 MongoDB 必须迁移到其他分片的数据块中剩余的数据块数量与 primary 在这个分片的数据库数量。

在 remaining 字段变为 0 之前，持续运行 removeShard 命令检查状态，这个命令需要在 admin 数据库上运行。

如果要移除的分片是基片，如图 18-8 所示，也就是标注为 primary 的分片，这里是 192.168.199.8:28000，那么要先手动修改数据库 mytest 的基片，改为 192.168.199.9:28000 后，再移除 192.168.199.8:28000。

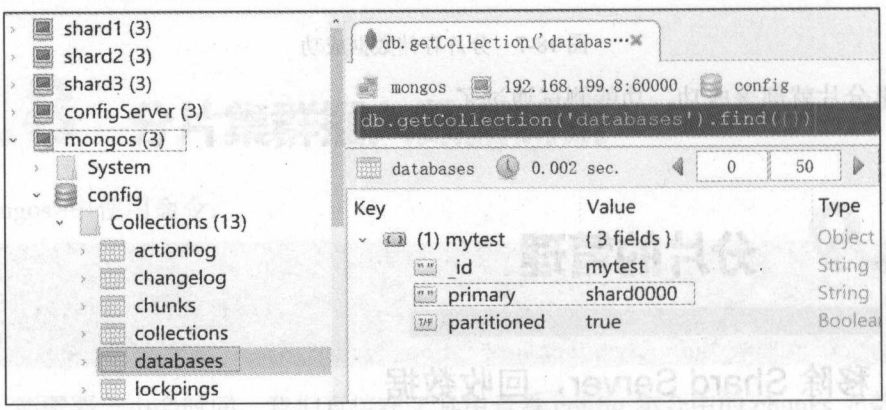


图 18-8　mytest 数据库的基片

在 mongos>中执行：

```
db.runCommand({"moveprimary" : "mytest","to" : "192.168.199.9:28000"})
```

等 primary 转移好后，再执行一次移除，即可成功移除 shard0000：

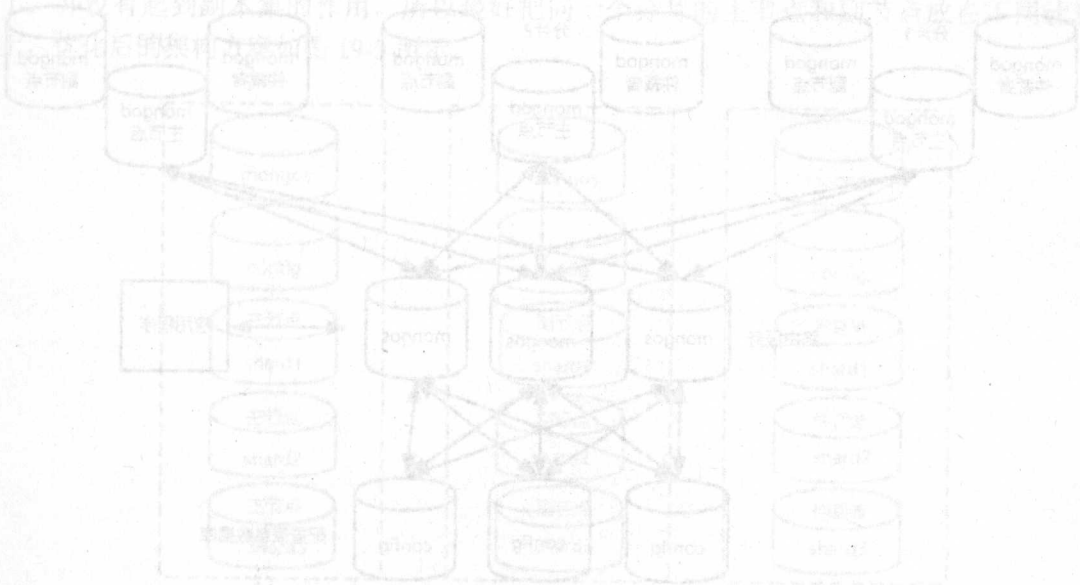
```
use admin
db.runCommand( { removeShard: " shard0000" } )
```

18.9.2　新增 Shard Server

在分片集群运行一段时间后，我们要增加新的分片机子，可以在 mongos>使用命令：

```
use admin
db.runCommand({"addshard" : "192.168.0.188:38011"})
```

MongoDB 规定分片集群加入的新 mongod 不能含有相同的数据库，如果有的话会报错，先把同名的数据库删除之后，才能新增为 Shard Server。



第 19 章

◂ 分片+副本集部署 ▸

19.1 总体思路

我们在第 18 章完成了单例 MongoDB 实例作为分片的 MongoDB 分片集群。但是这样的分片集群是存在风险的，因为每个分片都保存着应用程序所有数据的一部分数据，如果其中一个分片节点挂掉了，这部分数据就缺失了。MongoDB 官方建议每个 Shared 最好是一组 Replica Set，这样具有更好的容灾性。所以本章我们尝试分片+副本集部署。根据我们在第 18 章中的分片架构，把 MongoDB 单例替换成 MongoDB 副本集，架构如图 19-1 所示。

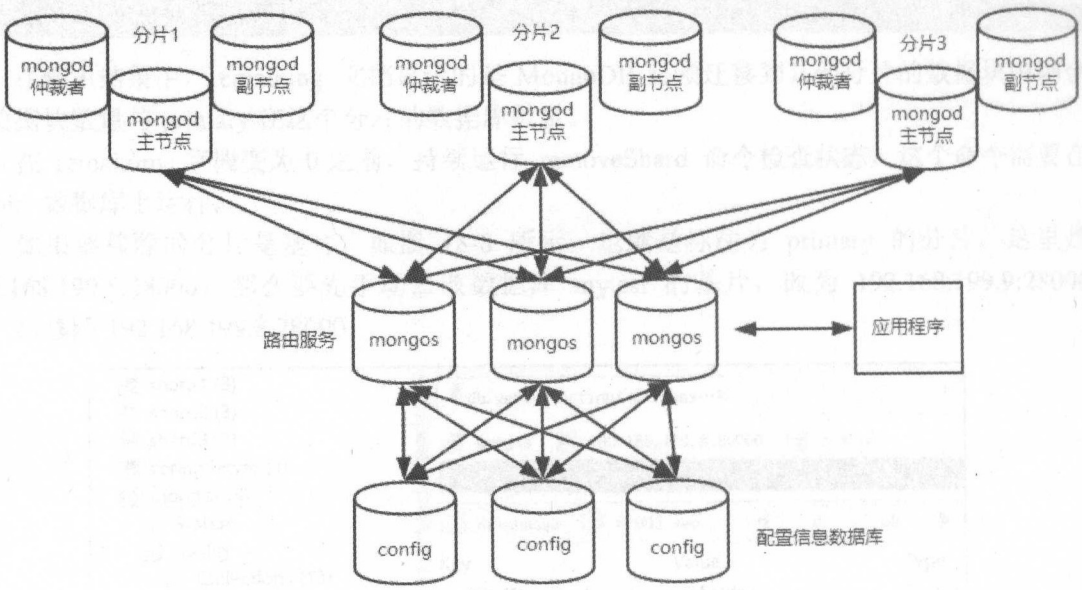


图 19-1　分片+副本集架构

理想情况下，这种架构需要 15 台计算机，mongos 和 Config Server 不保存应用程序数据，不会消耗太多性能，所以可以与 Shard 分片部署在同一台计算机。副本集中的 MongoDB 实例主节点、副节点以及仲裁节点最好是单独使用一台计算机，这样才能达到最好的容灾性。但是，如果是条件有限的情况，也可以把它们与 mongos 和 Config Server 放在一起，如图 19-2 所示。

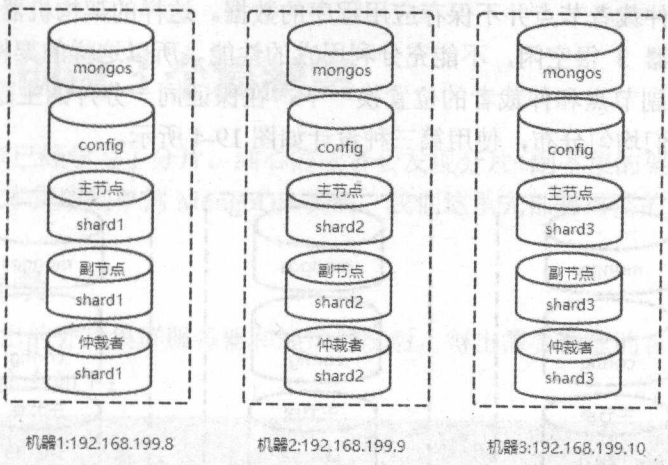


图 19-2　分片+副本集架构第一种设计

我们有三个虚拟机，所以还是做三个分片，可以在虚拟机上用不同端口启动三个 MongoDB 实例，分别作为主节点、副节点、仲裁者，组成副本集。但是，这种架构有一个致命缺点，它的一个分片中的主节点、副节点和仲裁者都放在同一台计算机下，如果只是主节点挂了，它的副本集还是能起作用的；如果这台计算机挂了，那么这个分片的数据就丢失了，并没有起到副本集的作用。所以最好把同一个分片的主节点和副节点放在不同计算机上，优化后的架构方案如图 19-3 所示。

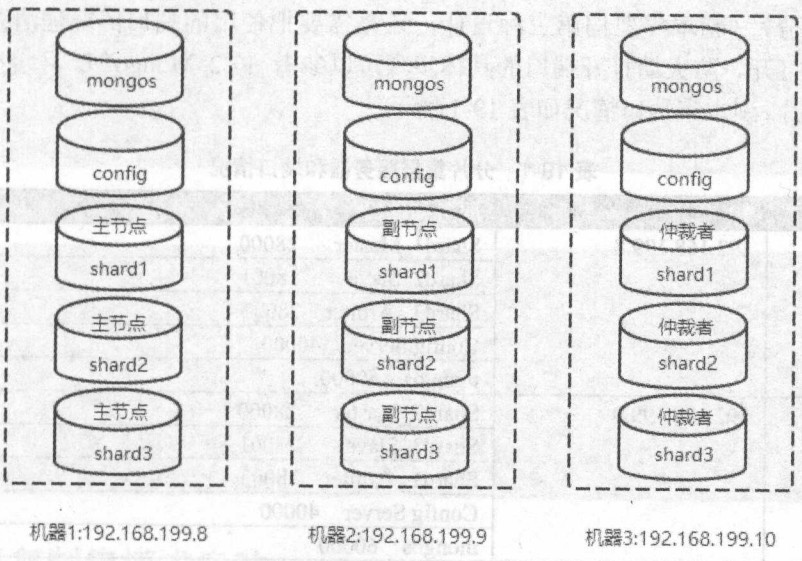


图 19-3　分片+副本集架构第二种设计

在分片+副本集架构第二种设计中，我们看到同一个分片的主节点和副节点分开放置，例如 shard1 的主节点放在机器 1 中，它的副节点放在机器 2 中，这样无论哪台机子挂掉了，分片都能正常工作。但是这种架构并不是最优的，因为我们看到机器 3 中都是放置的仲裁者

节点，我们都知道仲裁者节点并不保存应用程序的数据。这样的架构机器 3 几乎没有起到分流数据的作用，机器 3 很空闲，不能充分利用它的性能。所以这样的架构还得继续优化，我们需要把主节点、副节点和仲裁者的位置换一下，在保证同一分片的主副节点放置在不同机器上的同时，使它们均匀分布，使用第三种设计如图 19-4 所示。

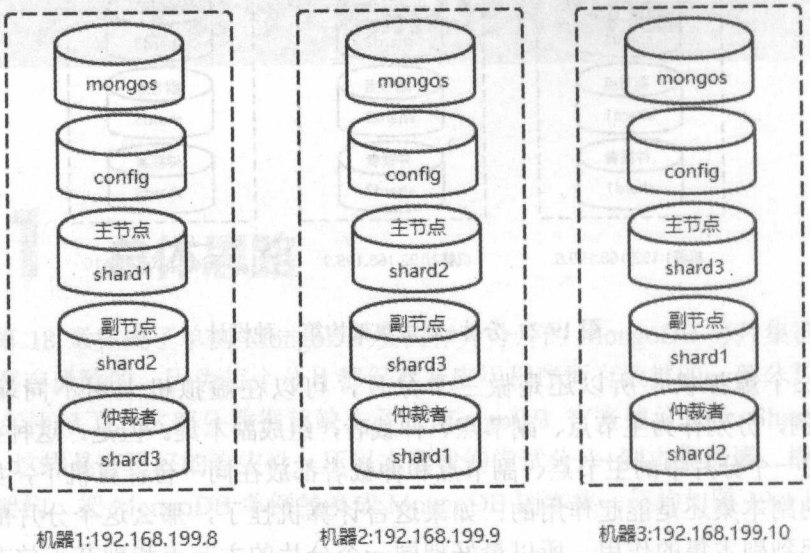


图 19-4　分片+副本集架构第三种设计

我们使用分片+副本集架构第三种设计，还是需要把使用的端口情况理清楚，并在防火墙把相应端口打开。防火墙打开端口的具体步骤可以参考 17.2 MongoDB 环境准备中的网络配置。我的分片+副本集端口情况如表 19-1 所示。

表 19-1　分片集群服务器和端口情况

主机	IP	服务和端口
mongodb0	192.168.199.8	Shard1_Master　28000
		Shard2_Slave　28001
		Shard3_Arbiter　28002
		Config Server　40000
		mongos　60000
mongodb1	192.168.199.9	Shard2_Master　28000
		Shard3_Slave　28001
		Shard1_Arbiter　28002
		Config Server　40000
		mongos　60000
mongodb2	192.168.199.10	Shard3_Master　28000
		Shard1_Slave　28001
		Shard2_Arbiter　28002
		Config Server　40000
		mongos　60000

19.2 创建 3 个复制集

我们在第 18 章已经学习了分片，细心的读者会发现分片+副本集的架构与分片的架构相差就在于 shard 是副本集还是单例 MongoDB 实例。我们这里先部署需要的三个副本集。

19.2.1 创建目录

对照表 18-1 给出的分片集群服务器和端口情况后，得出需要新建的目录。

mongodb0 使用命令如下：

```
mkdir -p /data/shard1/master
mkdir -p /data/shard1/master/log
mkdir -p /data/shard2/slave
mkdir -p /data/shard2/slave/log
mkdir -p /data/shard3/arbiter
mkdir -p /data/shard3/arbiter/log
```

mongodb1 使用命令如下：

```
mkdir -p /data/shard1/arbiter
mkdir -p /data/shard1/arbiter/log
mkdir -p /data/shard2/master
mkdir -p /data/shard2/master/log
mkdir -p /data/shard3/slave
mkdir -p /data/shard3/slave/log
```

mongodb2 使用命令如下：

```
mkdir -p /data/shard1/slave
mkdir -p /data/shard1/slave/log
mkdir -p /data/shard2/arbiter
mkdir -p /data/shard2/arbiter/log
mkdir -p /data/shard3/master
mkdir -p /data/shard3/master/log
```

19.2.2 以复制集模式启动

三个副本集分别取名为 shard1、shard2 和 shard3，每台机器需要使用不同端口启动三个 MongoDB 实例，命令说明如下。

mongodb0 使用命令：

```
mongod --dbpath=/data/shard1/master --port 28000 --
```

```
logpath=/data/shard1/master/log/shard.log --logappend --fork  --shardsvr  --
replSet shard1

mongod --dbpath=/data/shard2/slave --port 28001 --
logpath=/data/shard2/slave/log/shard.log --logappend --fork  --shardsvr  --
replSet shard2

mongod --dbpath=/data/shard3/arbiter --port 28002 --
logpath=/data/shard3/arbiter/log/shard.log --logappend --fork  --shardsvr  --
replSet shard3
```

mongodb1 使用命令：

```
mongod --dbpath=/data/shard2/master --port 28000 --
logpath=/data/shard2/master/log/shard.log --logappend --fork  --shardsvr  --
replSet shard2

mongod --dbpath=/data/shard3/slave --port 28001 --
logpath=/data/shard3/slave/log/shard.log --logappend --fork  --shardsvr  --
replSet shard3

mongod --dbpath=/data/shard1/arbiter --port 28002 --
logpath=/data/shard1/arbiter/log/shard.log --logappend --fork  --shardsvr  --
replSet shard1
```

mongodb2 使用命令：

```
mongod --dbpath=/data/shard3/master --port 28000 --
logpath=/data/shard3/master/log/shard.log --logappend --fork  --shardsvr  --
replSet shard3

mongod --dbpath=/data/shard1/slave --port 28001 --
logpath=/data/shard1/slave/log/shard.log --logappend --fork  --shardsvr  --
replSet shard1

mongod --dbpath=/data/shard2/arbiter --port 28002 --
logpath=/data/shard2/arbiter/log/shard.log --logappend --fork  --shardsvr  --
replSet shard2
```

19.2.3 初始化复制集

mongodb0 登录到 mongo 客户端初始化 shard1 副本集，使用命令如下：

```
mongo --port 28000
```

进入 mongos>后输入

```
rs.initiate()
rs.add("192.168.199.9:28002")
rs.add("192.168.199.10:28001")
```

mongodb1 登录到 mongo 客户端初始化 shard2 副本集，使用命令如下：

```
mongo --port 28000
```

进入 mongos>后输入：

```
rs.initiate()
rs.add("192.168.199.8:28001")
rs.add("192.168.199.10:28002")
```

mongodb2 登录到 mongo 客户端初始化 shard3 副本集，使用命令如下：

```
mongo --port 28000
```

进入 mongos>后输入：

```
rs.initiate()
rs.add("192.168.199.8:28002")
rs.add("192.168.199.9:28001")
```

19.3 创建分片需要的 Config Server 与 Route Process

19.3.1 创建目录

mongodb0 使用命令如下：

```
mkdir -p /data/shard/configdb
mkdir -p /data/shard/log
```

mongodb1 使用命令如下：

```
mkdir -p /data/shard/configdb
mkdir -p /data/shard/log
```

mongodb2 使用命令如下：

```
mkdir -p /data/shard/configdb
mkdir -p /data/shard/log
```

19.3.2 启动 Config Server、Route Process

mongodb0 使用命令启动 Config Server：

```
mongod --dbpath=/data/shard/configdb --port 40000 --
logpath=/data/shard/log/config.log --fork --configsvr --replSet configReplSet
```

mongodb1 使用命令启动 Config Server：

```
mongod --dbpath=/data/shard/configdb --port 40000 --
logpath=/data/shard/log/config.log --fork --configsvr --replSet configReplSet
```

mongodb2 使用命令启动 Config Server：

```
mongod --dbpath=/data/shard/configdb --port 40000 --
logpath=/data/shard/log/config.log --fork --configsvr --replSet configReplSet
```

如果未启动成功报错了可以查看日志，使用命令：

```
cat /data/shard/log/config.log
```

显示报错信息为：

```
bind() failed Address already in use for socket: 0.0.0.0:40000
```

说明有其他进程占用了 40000 端口。

可以使用 kill -2 命令退出占用的进程，查看命令如下：

```
ps -ef|grep 40000
```

输出信息为：

```
root      2480     1  0 Mar24 ?       00:00:00 mongos --configdb
cfgReplSet/192.168.199.8:40000,192.168.199.9:40000,192.168.199.10:40000 --port
60000 --logpath=/data/shard1/log/mongos.log --fork
root      2481   2480  0 Mar24 ?       00:01:19 mongos --configdb
cfgReplSet/192.168.199.8:40000,192.168.199.9:40000,192.168.199.10:40000 --port
60000 --logpath=/data/shard1/log/mongos.log --fork
root      3217   1236  0 01:37 pts/0    00:00:00 grep 40000
```

占用 40000 端口的进程有两个，进程号为 2480 和 2481，所以我们使用命令：

```
kill -2 2480
kill -2 2481
```

因为 MongoDB 3.4 版本多个 Config Server 必须以副本集的形式才能添加，所以

mongodb0 登录到 mongo 客户端初始化 config 副本集，使用命令如下：

```
mongo --port 40000
```

进入 mongos>后输入：

```
rs.initiate()
rs.add("192.168.199.9:40000")
rs.add("192.168.199.10:40000")
```

mongodb0 使用命令启动 mongos：

```
mongos  --configdb
configReplSet/192.168.199.8:40000,192.168.199.9:40000,192.168.199.10:40000  --
port 60000  --logpath=/data/shard/log/mongos.log --fork
```

mongodb1 使用命令启动 mongos：

```
mongos  --configdb
configReplSet/192.168.199.8:40000,192.168.199.9:40000,192.168.199.10:40000  --
port 60000  --logpath=/data/shard/log/mongos.log --fork
```

mongodb2 使用命令启动 mongos：

```
mongos  --configdb
configReplSet/192.168.199.8:40000,192.168.199.9:40000,192.168.199.10:40000  --
port 60000  --logpath=/data/shard/log/mongos.log --fork
```

注意 configReplSet 是 Config Server 副本集的名称，读者需要对应自己起的名称。

19.4 配置分片

配置分片也就是把复制集添加为分片节点，这是分片+副本集区别于分片集群的重要步骤，在其中一台机子中进入 mongos>执行配置命令，使用命令如下：

```
mongo --port 60000
```

进入 mongos>后输入：

```
use admin
db.runCommand({addshard:"shard1/mongodb0:28000,192.168.199.9:28002,192.168.199
.10:28001" })
db.runCommand({addshard:"shard2/192.168.199.8:28001,mongodb1:28000,192.168.199
.10:28002" })
db.runCommand({addshard:"shard3/192.168.199.8:28002,192.168.199.9:28001,mongod
```

```
b2:28000" })
```

添加成功后输出信息为：

```
{ "shardAdded" : "shard1", "ok" : 1 }
{ "shardAdded" : "shard2", "ok" : 1 }
{ "shardAdded" : "shard3", "ok" : 1 }
```

注意，shard1 和 shard2 以及 shard3 分别对应我们在 19.2 节“三个复制集”中创建的副本集名称以及下属的 name。

需要注意的是，name 中 ip 与 mongodb0 主机名不能自动识别替换，所以 name 要与副本集中 rs.status()命令查看的一致，具体的名称可以用如下命令查看：

```
mongo --port 28000
```

进入 PRIMARY>，输入命令：

```
rs.status()
```

如果不对应输出的错误信息如下：

```
{
        "code" : 96,
        "ok" : 0,
        "errmsg" : "in seed list
shard3/192.168.199.8:28002,192.168.199.9:28001,192.168.199.10:28000, host
192.168.199.10:28000 does not belong to replica set shard3; found { hosts:
[ \"mongodb2:28000\", \"192.168.199.8:28002\", \"192.168.199.9:28001\" ],
setName: \"shard3\", setVersion: 3, ismaster: true, secondary: false, primary:
\"mongodb2:28000\", me: \"mongodb2:28000\", electionId:
ObjectId('7fffffff0000000000000001'), lastWrite: { opTime: { ts: Timestamp
1490377813000|1, t: 1 }, lastWriteDate: new Date(1490377813000) },
maxBsonObjectSize: 16777216, maxMessageSizeBytes: 48000000, maxWriteBatchSize:
1000, localTime: new Date(1490377813902), maxWireVersion: 5, minWireVersion: 0,
readOnly: false, ok: 1.0 }"
}
```

到此分片+副本集的架构就部署成功了，相关分片和副本集的功能测试可参考第 17 章和第 18 章，这里就不重复叙述了。

第 20 章

springMVC+maven+MongoDB 框架搭建

我们在第 16 章“Java 操作 MongoDB”中学习了使用 Java 驱动操作 MongoDB，但是其中的操作都比较直白，没有经过封装，而且每次使用前都要先写数据库名和 ip 端口。我们在开发 Web 时，如果也直接使用 Java 驱动无疑会增加很多代码量。目前比较主流的 Web 开发框架 springMVC+maven 中已经增加了对 MongoDB 的支持，提供的 spring-data-mongodb 驱动包对原生的 MongoDB 官方 Java 驱动进行了一些封装，让我们在 Web 项目中能够很方便地操作 MongoDB。本章我们就来搭建 Mongodb 的 Web 框架。

20.1 SpringMVC 和 Maven 简介

SpringMVC 是一种基于 Java 的实现了 Web MVC 设计模式的轻量级 Web 框架，使用了 MVC 架构模式的思想，将 Web 层进行职责解耦，并对传统的 Web 请求响应进行了封装，框架的目的就是帮助我们简化开发，SpringMVC 能够帮助我们高效地开发 Java 的 Web 应用。

Maven 是 Apache 的一个开源项目，主要服务于基于 Java 平台的项目构建、依赖管理、项目信息管理。Maven 主要能为我们提供以下几个服务：

（1）自动编译。
（2）自动下载管理依赖 jar 包。
（3）获取项目信息。

20.2 Eclipse 安装 Maven 插件

在 16.1 节 Eclipse 安装中我们已经安装了 Eclipse，版本为 Version: Neon.3 Release (4.6.3)，它已经自带了 Maven 插件。如果使用其他的 IDE，需要检查是否已经自带了 Maven 插件，因为每种 IDE 的插件安装方式不同，这里就不详细说明了。

20.3 新建 Maven 类型的 Web 项目

已经安装 Maven 插件的 Eclipse 可以新建 Maven 类型的 Web 项目，步骤为在 Eclipse 选项卡中单击 File→New→Maven Project，如图 20-1 所示。如果 New 中没有 Maven Project，则在最后一列 Other 中找到 Maven Project。

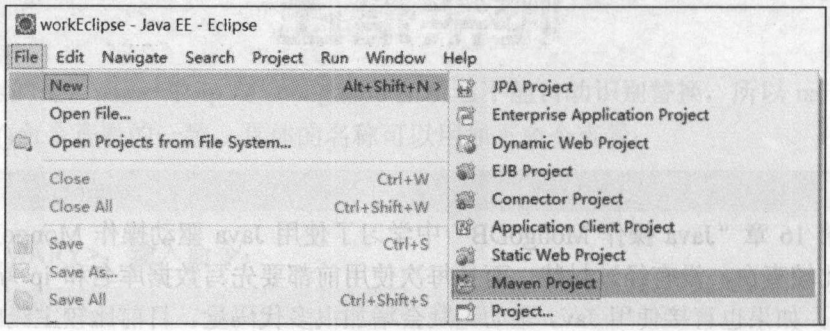


图 20-1 新建 Maven 项目

然后对项目进行快速配置，选中 Create a simple project(skip archetype selection) →Next→输入项目命名和信息，选择项目打包为 war，如图 20-2 所示。

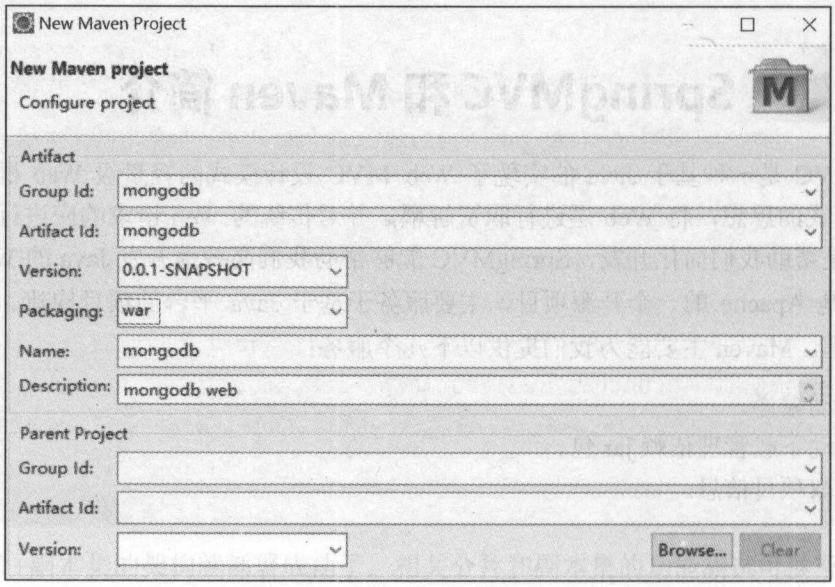


图 20-2 输入项目命名和信息

此时创建的 Maven 项目是 Java 项目，我们需要把它设置成 Web 项目。

对着项目右击，选择 Properties，进入属性页面，选择 Project Facets 菜单，Dynamic Web Module 选择为版本 3.0 和 Java 选择为版本 1.7，如图 20-3 所示。

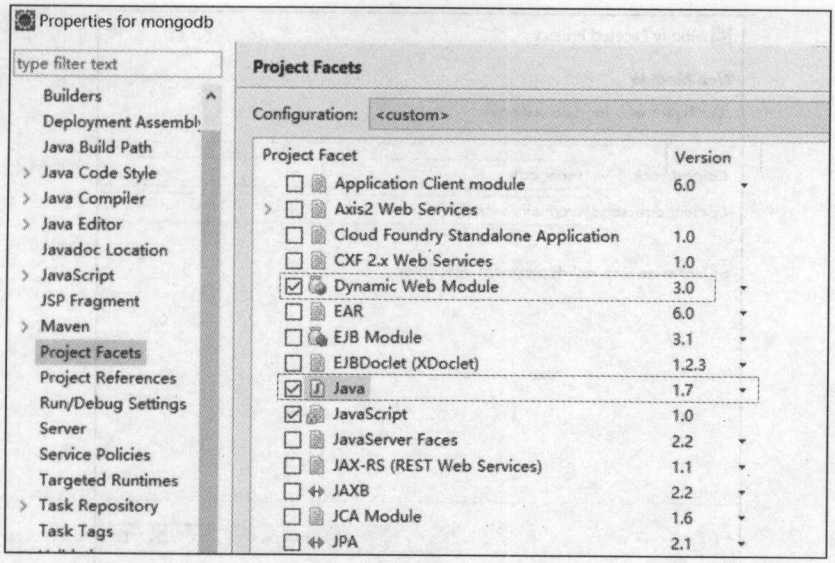


图 20-3　Project Facets 菜单选项

单击右下角的 Apply→OK。再次对着项目右击，选择 Properties，进入属性页面，选择 Project Facets 菜单，去掉勾选 Dynamic Web Module，单击右下角的 Apply→OK，然后再进入一次 Project Facets，勾选 Dynamic Web Module，这时候看到下方出现了 Further configuration available...选项。这里取消勾选再勾选就是为了触发这个选项的显示，如图 20-4 所示。单击 Further configuration available...，进入 Web Module 的设置。

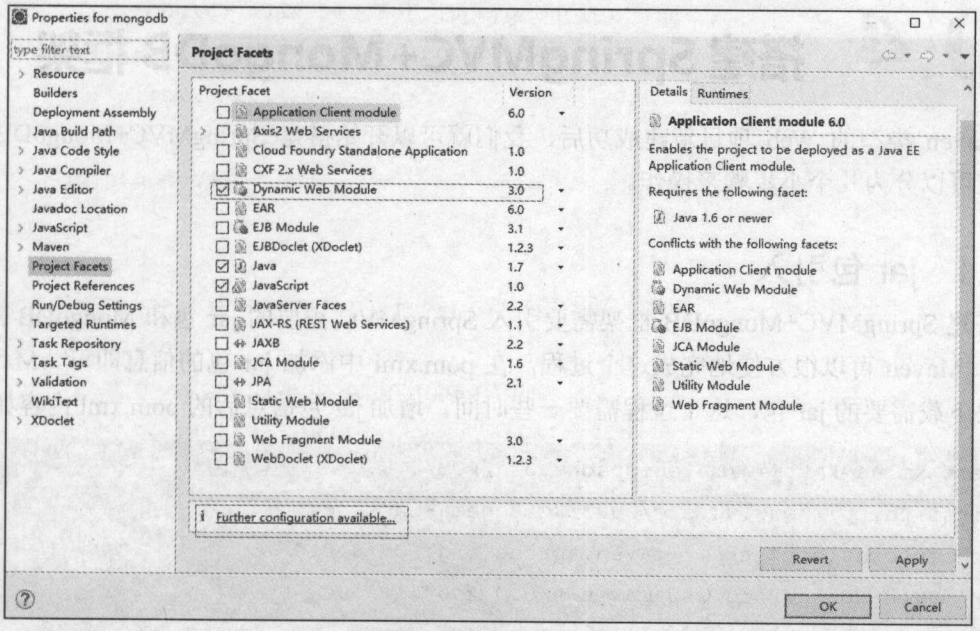


图 20-4　Further configuration available...选项

配置 Content directory 为 src/main/webapp，并勾选生成 web.xml 的选项，如图 20-5 所示。

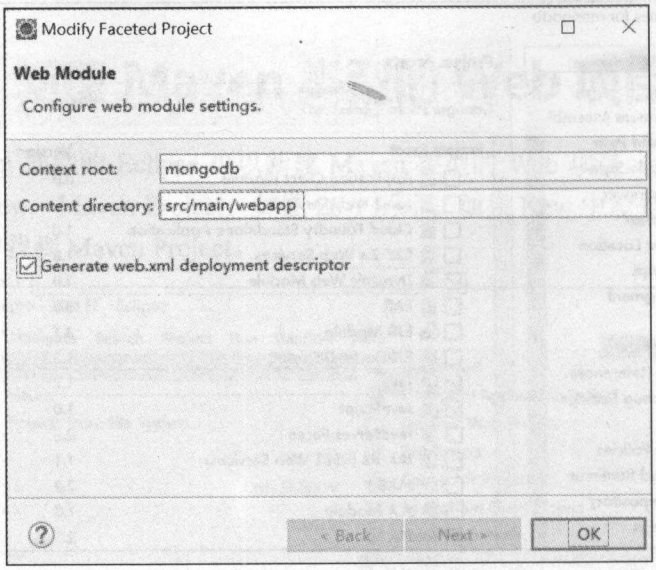


图 20-5　配置 Web Module

单击 OK 后会返回到 Project Facets 菜单，单击 Project Facets 菜单右下角的 Apply→OK。

这时候我们可以看到 mongodb 项目中 src/main/webapp 的 WEB-INF 目录下已经有 web.xml 文件了。这样我们的 Maven 类型的 Web 项目就新建成功了。

20.4 搭建 SpringMVC+MongoDB 框架

Maven 类型的 Web 项目新建成功后，我们就可以开始搭建 SpringMVC+MongoDB 框架了，它可以分为几个小步骤来操作。

20.4.1 jar 包引入

搭建 SpringMVC+MongoDB 框架需要引入 SpringMVC 需要的 jar 包和 MongoDB 需要的 jar 包，Maven 可以很方便地完成这个过程，在 pom.xml 中添加 jar 包的信息即可，Maven 会自动去下载需要的 jar 包，这个过程需要一些时间。增加 jar 包信息后的 pom.xml 内容如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>mongodb</groupId>
  <artifactId>mongodb</artifactId>
  <version>0.0.1-SNAPSHOT</version>
```

```
 <packaging>war</packaging>
 <name>mongodb</name>
 <description>mongodb web</description>
   <properties>
       <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
   </properties>
<dependencies>
          <dependency>
          <groupId>org.apache.openejb</groupId>
          <artifactId>javaee-api</artifactId>
          <version>5.0-1</version>
          <scope>provided</scope>
       </dependency>
       <dependency>
           <groupId>javax.servlet</groupId>
           <artifactId>jstl</artifactId>
           <version>1.2</version>
           <scope>provided</scope>
       </dependency>
       <dependency>
           <groupId>javax.servlet.jsp</groupId>
           <artifactId>jsp-api</artifactId>
           <version>2.1</version>
           <scope>provided</scope>
       </dependency>


       <!-- aspectjweaver.jar 这是 Spring AOP 所要用到的包 -->
       <dependency>
           <groupId>org.aspectj</groupId>
           <artifactId>aspectjweaver</artifactId>
           <version>1.7.1</version>
       </dependency>

           <!-- spring mvc -->
       <dependency>
           <groupId>org.springframework</groupId>
           <artifactId>spring-webmvc</artifactId>
           <version>4.0.0.RELEASE</version>
```

```
    </dependency>

    <!-- spring3 -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>4.0.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>4.0.0.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-beans</artifactId>
        <version>4.0.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-web</artifactId>
        <version>4.0.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-expression</artifactId>
        <version>4.0.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-orm</artifactId>
        <version>4.0.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-test</artifactId>
        <version>4.0.0.RELEASE</version>
    </dependency>
```

```
        <!--jsp 页面使用的 jstl -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
            <version>1.2</version>
        </dependency>



        <!-- mongodb -->
         <dependency>
           <groupId>org.springframework.data</groupId>
           <artifactId>spring-data-mongodb</artifactId>
           <version>1.5.5.RELEASE</version>
       </dependency>
</dependencies>
<build>
    <plugins>
       <!-- define the project compile level -->
       <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>2.3.2</version>
          <configuration>
             <source>1.7</source>
             <target>1.7</target>
          </configuration>
       </plugin>
    </plugins>
</build>
</project>
```

如果报错 Failed to read artifact descriptor，说明包的下载出错，主要是 Maven 的默认仓库是用的国外的，感兴趣的读者可以自行学习设置 Maven 仓库成国内第三方仓库镜像，这里就不细说了。遇到下载不下来 jar 包时，可以强制重新下载，对着项目右击→maven→update project →勾选 Forces updated releases /snapshots。

包下载完成后查看项目的 Libraries，发现 Maven Dependencies 中已经有了我们需要的 jar 包，如图 20-6 所示。

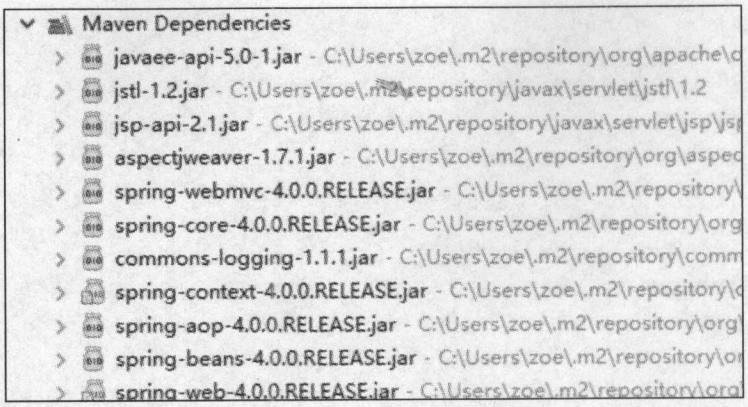


图 20-6　Maven 自动下载的 jar 包

20.4.2　新建 SpringMVC 配置文件

SpringMVC 配置文件是 xml 文件，里面主要设置包的路径和视图模式，让 SpringMVC 框架知道 Web 项目的 java 代码和 jsp 页面代码分明放置在哪一个地方。新建步骤如下：

Eclipse 选项卡中单击 File→New→Other→搜索 xml→选中 XML File→选中/src/main/resources 路径----》→输入 xml 名称为：springMVC.xml→Finish，如图 20-7 所示。

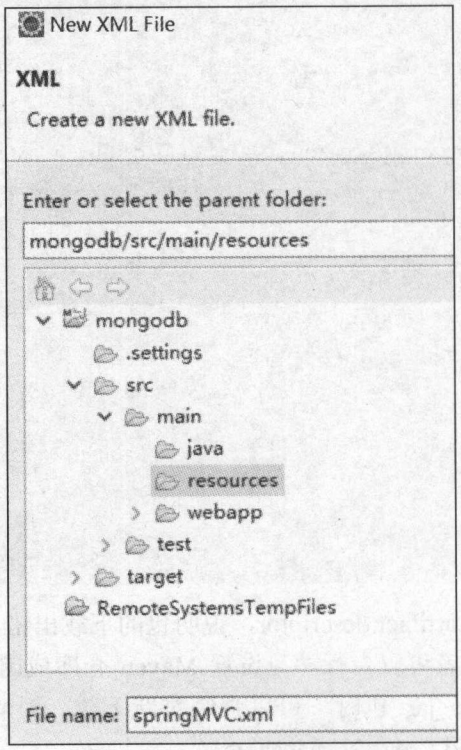


图 20-7　新建 xml 文件

springMVC.xml 的完整内容修改如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:context="http://www.springframework.org/schema/context"
xmlns:p="http://www.springframework.org/schema/p"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context-3.1.xsd
      http://www.springframework.org/schema/mvc
      http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

 <!-- 支持注解 -->
    <mvc:annotation-driven />


    <!--自动装配 DefaultAnnotationHandlerMapping 和 AnnotationMethodHandlerAdapter
-->
<mvc:default-servlet-handler />


<!-- 设置自动扫描的路径,用于自动注入 bean    这里的路径与自己的项目目录对应-->
<!-- 扫描 controller 路由控制器  -->
<context:component-scan base-package="com.mongodb" />

    <!-- 设置静态资源可访问 -->
    <mvc:resources location="/" mapping="/**"/>

    <!-- 视图解析器 -->
<bean id="viewResolver"

class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="viewClass"
value="org.springframework.web.servlet.view.JstlView" />
<property name="suffix" value=".jsp" />  <!-- 视图文件类型 -->
<property name="prefix" value="/WEB-INF/views" /> <!-- 视图文件的文件夹路径 -->
</bean>
</beans>
```

我们这里设置 Java 代码的路径是 com.mongodb，JSP 的路径是/WEB-INF/views。所以需要在/src/main/java 中新建 Package，命名为 com.mongodb。在/src/main/webapp/ WEB-INF 中新建 views 文件夹，如图 20-8 所示。

图 20-8　新建目录

如果有报错请对着项目右击，选中 properties→Java Compiler，将 Compiler compliance level 修改为 1.7。

20.4.3　新建 MongoDB 配置文件

MongoDB 配置文件也是 xml 文件，主要是配置 Web 项目使用的 MongoDB 实例的 ip 和端口以及使用的数据库。例如，我这里使用 192.168.199.8 端口 27017 中的 test 数据库，在/src/main/resources 路径下新建 mongodb.xml 文件后，输入内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!-- 指定 Spring 配置文件的 Schema 信息 -->
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:p="http://www.springframework.org/schema/p"
xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:mongo="http://www.springframework.org/schema/data/mongo"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
```

```
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.1.xsd">

   <bean id="mongo"
class="org.springframework.data.mongodb.core.MongoFactoryBean">
       <property name="host" value="192.168.199.8"/>
       <property name="port" value="27017"/>
    </bean>

    <bean id="mongoTemplate"
class="org.springframework.data.mongodb.core.MongoTemplate">
       <constructor-arg name="mongo" ref="mongo" />
       <constructor-arg name="databaseName" value="test"/>
    </bean>

</beans>
```

20.4.4　配置 web.xml

我们需要把前面新建的 springMVC.xml、mongodb.xml（名称与自己的前面的命名统一）在 web.xml 中配置引用。

完整 web.xml 内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <display-name></display-name>
<!-- spring mongodb -->
    <context-param>
       <param-name>contextConfigLocation</param-name>
       <param-value>classpath:mongodb.xml</param-value>
    </context-param>
    <!--spring mvc 配置 -->
    <servlet>
       <servlet-name>springMVC</servlet-name>
       <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
       <init-param>
```

```
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:springMVC.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>springMVC</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

  <!-- encodeing -->
  <filter>
    <filter-name>encodingFilter</filter-name>
    <filter-
class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
    <init-param>
      <param-name>encoding</param-name>
      <param-value>UTF-8</param-value>
    </init-param>
    <init-param>
      <param-name>forceEncoding</param-name>
      <param-value>true</param-value>
    </init-param>
  </filter>
  <!-- encoding filter for jsp page -->
  <filter-mapping>
    <filter-name>encodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  <listener>          <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
</web-app>
```

20.4.5 创建 index.jsp 和 IndexController

测试 SpringMVC 框架需要一个路由控制器，我们在 com.mongodb 下新建 IndexController.java，对着 com.mongodb 包右击，单击 New→Class，在/WEB-INF/views 路径下新建一个 index.jsp。对着 WEB-INF/views 文件夹右击，单击 New→JSP File。

IndexController.java 内容如下：

```
package com.mongodb;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
```

```
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class IndexController {
      @RequestMapping(value={"","/","/index"})
        public String  index(Model model)  {
                    return "/index";
        }
    }
```

index.jsp 页面内容如下：

```
<%@ page language="java" contentType="text/html; charset=utf-8"
   pageEncoding="utf-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>测试 mongodb</title>
</head>
<body>
欢迎您
</body>
</html>
```

创建完之后，页面可能会报错：The superclass javax.servlet.http.HttpServlet was not found on the Java Build Path，这是缺少库导致的，在下一小节安装 Tomcat 后中会解决这个问题。

20.4.6　启动 Web 项目

经过上面的部署，我们的 Maven 类型的 Web 项目 SpringMVC+MongoDB 框架就搭建完成了，可以尝试一下能不能成功运行起来。运行 Web 项目需要 Web 服务器，Java Web 项目比较常用的服务器是 Tomcat，我们这里安装 Tomcat，然后设置 Eclipse 使用 Tomcat。

1. 安装 Tomcat

Tomcat 官网是 http://tomcat.apache.org/，Tomcat 下载下来是一个压缩文件，解压即可用。我这里下载了 apache-tomcat-8.5.12-windows-x64.zip，将压缩文件解压至自定义路径（我的路径: D:\ apache-tomcat-8.5.12）。注意，在安装 Tomcat 时，在其字母周围不要存在空格，否则可能导致配置不成功。

在系统变量里配置环境变量（不区分大小写）：

（1）新增变量名：CATALINA_HOME，变量值：D:\apache-tomcat-8.5.12。

（2）新增变量名：JRE_HOME （与安装 JDK 的路径对应），变量值：C:\Program Files\Java\jre1.8.0_121。

（3）编辑变量名：path，在最后加上变量值;%CATALINA_HOME% \lib; %CATALINA_HOME% \lib\servlet-api.jar; %CATALINA_HOME%\lib\jsp-api.jar。

配置完成后可以进入 tomcat 安装路径的 bin 目录下，单击 startup.bat 测试是否能启动成功。tomcat 启动 startup.bat，一闪而过说明报错了。右击 startup.bat，单击编辑，在文本的最后加上 pause，保存后重新运行 startup.bat，这时候窗口不会再一闪而过，而是停留在桌面上，这时候可以看到报错信息，一般是缺少了什么配置变量，根据提示去配置即可（调试成功后，把 pause 去掉就能正常使用 tomcat 了）。

2. Eclipse 配置 Tomcat

在 Eclipse 选项卡中，选择 Window→Preferences→Server→Runtime Environments→Add。

选择对应版本，我们这里选择 Tomcat v8.5 Server，新建一个本机服务，再进行下一步。

选择 Tomcat 安装路径，选择 JRE 版本，完成配置。

然后将 Tomcat 添加到 Build Path 的库中，对着项目右击，执行 Build Path→Configure Build Path→Add Library... →Server Runtime→选中 Apache Tomcat v8.5→Finish。

3. 启动 tomcat

对着项目名右击，执行 Run as→Run on Server 即可。此时，Console 控制台会输出启动信息。输出信息 Server startup in xxx ms 时，启动成功。

启动成功后可以在浏览器中访问项目：http://localhost:8080/mongodb/，如图 20-9 所示。

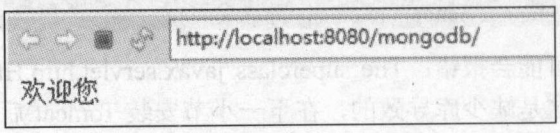


图 20-9 启动 mongodb 项目成功

第 21 章

◂ 注册登录功能的实现 ▸

第 20 章已经搭建好了工作中常用的 Web 框架。本章就结合 Spring Data MongoDB 的用法来完成一个 Web 应用程序中常见的功能：注册和登录。通过这个实例，我们就能掌握实际工作中 MongoDB 在 Web 应用开发中的使用方式了。Spring Data MongoDB 驱动调用 MongoDB 的使用方式跟第 16 章中官方支持 Java 的 MongoDB 驱动有些区别，我们将在本章最后一节给出常见的操作用法。

21.1 UI 框架 Bootstrap

21.1.1 简介

Bootstrap 是 Twitter 推出的一个用于前端开发的开源工具包。它由 Twitter 的设计师 Mark Otto 和 Jacob Thornton 合作开发，是一个 CSS/HTML 框架，用于开发响应式布局、移动设备优先的 Web 项目。Bootstrap 让前端开发更快速、简单，让所有开发者都能快速上手、所有设备都可以适配、所有项目都适用。这里使用 Bootstrap 作为我们的 UI 框架。Bootstrap 的官网是 http://www.bootcss.com/，更多 Bootstrap 的信息可查看官网。

21.1.2 应用 Bootstrap

把 Bootstrap 应用到我们的 Web 中很简单，只需要在官网下载 Bootstrap（如图 21-1 所示）的资源包，放到 Web 中，然后引用相关 CSS 和 JS 即可。

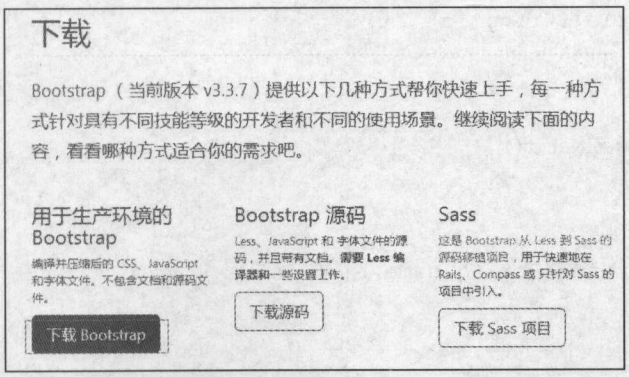


图 21-1　下载 Bootstrap

下载完毕后得到 bootstrap-3.3.7-dist.zip，解压后得到文件夹 bootstrap-3.3.7-dist，把这个文件放到 Web 项目的 src/main/webapp 目录下即可。我们在编写 JSP 页面时可以引用里面的资源。

21.2 新建用户实体

用户注册需要保存用户的信息，登录时则需要验证用户的信息，这些信息数据都是需要保存到 MongoDB 的。MongoDB 的好处就是不需要提前新建集合和设置字段的数据类型，只需要新建好 Java 中的实体 class，通过 Spring Data MongoDB 的方式保存到 MongoDB 即可。

MongoDB 会自动把 class 实体转化为 BSON 文档，自动对应数据类型。

对着 com.mongodb 右击，单击新建，Package 命名为 entity，在 entity 下新建的用户实体 class 命名为 User.java。

User.java 代码如下：

```
package com.mongodb.entity;

import javax.persistence.Entity;

@Entity
public class User {
    private String id;//用户 id
    private String name;//用户名称
    private String username;//登录账户
    private String password;//登录密码
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getUsername() {
        return username;
```

```
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }

}
```

21.3 注册功能编写

注册功能需要完成两个部分，一是注册页面，二是注册后端。注册页面 JSP 让用户输入信息，单击注册按钮后把数据发送到后端，后端 Java 代码中把注册信息保存到数据库。

21.3.1 注册页面代码

在/WEN-INF/views 中新建 register.jsp。

register.jsp 代码如下：

```
<%@ page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<%
String path = request.getContextPath();
String basePath =
request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+
path+"/";
%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html lang="zh-CN">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
```

```
    <!-- 上述3个 meta 标签*必须*放在最前面，任何其他内容都*必须*跟随其后！ -->
    <meta name="description" content="">
    <meta name="author" content="">
    <title>注册页</title>
    <!-- Bootstrap core CSS -->
    <link href="<%=basePath%>/bootstrap-3.3.7-dist/css/bootstrap.min.css"
rel="stylesheet">
  </head>
  <style type="text/css">
  body {
  padding-top: 40px;
  padding-bottom: 40px;
  background-color: #eee;
}

.form-signin {
  max-width: 330px;
  padding: 15px;
  margin: 0 auto;
}
.form-signin .form-signin-heading,
.form-signin .checkbox {
  margin-bottom: 10px;
}
.form-signin .checkbox {
  font-weight: normal;
}
.form-signin .form-control {
  position: relative;
  height: auto;
  -webkit-box-sizing: border-box;
     -moz-box-sizing: border-box;
          box-sizing: border-box;
  padding: 10px;
  font-size: 16px;
}
.form-signin .form-control:focus {
  z-index: 2;
}
```

```
.form-signin input[type="email"] {
  margin-bottom: -1px;
  border-bottom-right-radius: 0;
  border-bottom-left-radius: 0;
}
.form-signin input[type="password"] {
  margin-bottom: 10px;
  border-top-left-radius: 0;
  border-top-right-radius: 0;
}
</style>
  <body>

    <div class="container">
      <form class="form-signin" id="register" method="post" action="saveUser">
        <h2 class="form-signin-heading">注册</h2>
        <label for="inputName" class="sr-only">Name</label>
        <input type="name" name="name" class="form-control" placeholder="Name"
required autofocus>
         <label for="inputUsername" class="sr-only">Username</label>
        <input type="username" name="username" class="form-control"
placeholder="Username" required>
        <label for="inputPassword" class="sr-only">Password</label>
        <input type="password" name="password" class="form-control"
placeholder="Password" required>
         <button class="btn btn-lg btn-primary btn-block" type="submit">注册
</button>
      </form>
    </div> <!-- /container -->
  </body>
</html>
```

这里需要注意的是，input 中的 id 值要与后端代码中的@RequestParam 参数命名对应，form 中的 action 路径需要与后端代码中的注册方法的@RequestMapping 对应。

21.3.2　注册后端代码

注册后端代码负责接受 JSP 页面传递过来的参数，组装成 User 实体保存入库。保存入库时，需要借助 Spring Data MongoDB 包中的 MongoTemplate 类。在 MongoTemplate 类的声明时加上@Autowired 注解，SpringMVC 框架就会把我们之前在 mongodb.xml 中定义的 bean 类

型的 mongoTemplate 信息关联起来。mongoTemplate 的 bean 里记录了我们要连接的数据库的地址和数据库名，这样就能把数据保存到数据库里了。生产环境中的注册，一般都会把密码经过加密后再保存入库，而不是存储明文，登录验证时，把用户输入的密码加密一次再与数据库中的密码作对比。一般的加密方法有 MD5 加密等，有兴趣的读者可以尝试，这里就不细说了。

在 com.mongodb 下新建一个 UserController.java。

UserController.java 的代码如下：

```
package com.mongodb;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

import com.mongodb.entity.User;

@Controller
public class UserController {
    @Autowired
    MongoTemplate mongoTemplate;

    @RequestMapping(value = { "register" })
    public String register() {
        //注册页面的路由，跳转到注册页面 register.jsp
        return "/register";
    }

    @RequestMapping(value = { "saveUser" })
    public String saveUser(Model model, @RequestParam String name,
@RequestParam String username,
            @RequestParam String password) {
        User user = new User();
        user.setName(name);
        user.setPassword(password);
        user.setUsername(username);
        mongoTemplate.save(user);//保存 User 到数据库
```

```
        return "/login";
    }
}
```

21.4 登录功能编写

登录功能也需要两部分，一是登录页面，二是登录后端代码，登录页面让用户输入账号密码，单击登录按钮后把用户密码传到登录后端，后端代码根据用户名密码去数据库中查询用户信息。如果查到了说明有该用户，用户登录成功。查不到用户，则登录失败。

21.4.1 登录页面代码

登录页面与注册页面比较类似，区别主要在于 form 的 action 路径，以及只需要填写 username 和 password。在/WEB-INF/views 中新建 login.jsp。

login.jsp 的代码为：

```
<%@ page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<%
String path = request.getContextPath();
String basePath =
request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+
path+"/";
%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html lang="zh-CN">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <!-- 上述3个meta标签*必须*放在最前面，任何其他内容都*必须*跟随其后！ -->
    <meta name="description" content="">
    <meta name="author" content="">
    <title>登录页</title>
    <!-- Bootstrap core CSS -->
    <link href="<%=basePath%>/bootstrap-3.3.7-dist/css/bootstrap.min.css"
rel="stylesheet">
```

```
  </head>
  <style type="text/css">
  body {
  padding-top: 40px;
  padding-bottom: 40px;
  background-color: #eee;
}

.form-signin {
  max-width: 330px;
  padding: 15px;
  margin: 0 auto;
}
.form-signin .form-signin-heading,
.form-signin .checkbox {
  margin-bottom: 10px;
}
.form-signin .checkbox {
  font-weight: normal;
}
.form-signin .form-control {
  position: relative;
  height: auto;
  -webkit-box-sizing: border-box;
     -moz-box-sizing: border-box;
          box-sizing: border-box;
  padding: 10px;
  font-size: 16px;
}
.form-signin .form-control:focus {
  z-index: 2;
}
.form-signin input[type="email"] {
  margin-bottom: -1px;
  border-bottom-right-radius: 0;
  border-bottom-left-radius: 0;
}
.form-signin input[type="password"] {
  margin-bottom: 10px;
```

```
  border-top-left-radius: 0;
  border-top-right-radius: 0;
}
</style>
  <body>
    <div class="container">
      <form class="form-signin" id="login" method="post" action="loginUser">
        <h2 class="form-signin-heading">登录</h2>
       <label for="inputUsername" class="sr-only">Username</label>
        <input type="username" name="username" class="form-control"
placeholder="Username" required>
        <label for="inputPassword" class="sr-only">Password</label>
        <input type="password" name="password" class="form-control"
placeholder="Password" required>
        <button class="btn btn-lg btn-primary btn-block" type="submit">登录
</button>
      </form>
    </div> <!-- /container -->
  </body>
</html>
```

21.4.2　登录后端代码

登录后端代码主要负责接收用户输入的 username 和 password，然后在 MongoDB 库 User 集合中查询，能查到则认证通过，登录成功。

在 com.mongodb 下新建 LoginController.java。

LoginController.java 代码如下：

```
package com.mongodb;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.query.Criteria;
import org.springframework.data.mongodb.core.query.Query;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

import com.mongodb.entity.User;
```

```
@Controller
public class LoginController {
    @Autowired
    MongoTemplate mongoTemplate;

    @RequestMapping(value = { "login" })
    public String login() {
        //登录页面的路由，跳转到登录页面 login.jsp
        return "/login";
    }

    @RequestMapping(value = { "loginUser" })

    public String loginUser(Model model, @RequestParam String username,
            @RequestParam String password) {
        Query query=new Query();
        query.addCriteria(Criteria.where("username").is(username));
        query.addCriteria(Criteria.where("password").is(password));
        if(mongoTemplate.count(query,User.class)>0){
            //根据账号密码去 MongoDB 数据库中查询 User 集合,数量大于0，则登录成功
            return "/index";//登录成功后进入 index.jsp 页面
        }
        return "/login";//登录失败返回 login.jsp 页面
    }
}
```

21.5 运行测试

完成注册登录后 mongodb 项目目录如图 21-2 所示。

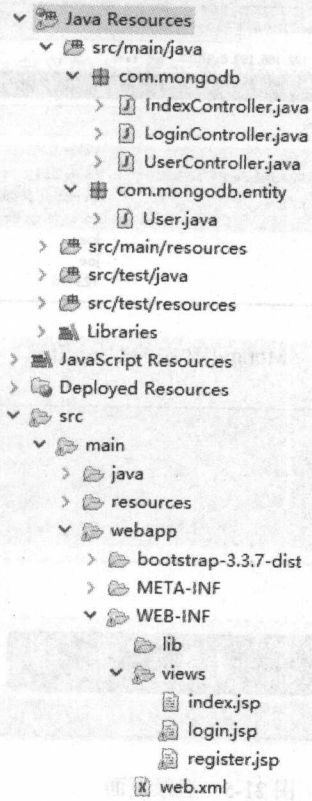


图 21-2　mongodb 项目目录

对着 mongodb 项目名右击，选择 Run As→Run on Server。

使用路径 http://localhost:8080/mongodb/register 在浏览器中访问注册页面，并填入信息，单击注册，如图 21-3 所示。

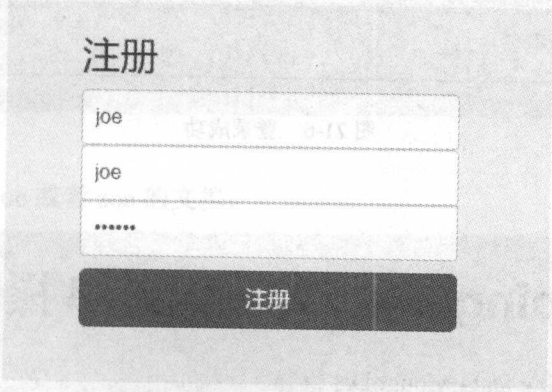


图 21-3　注册用户

注册成功后页面会跳转到登录页面，这时我们查看数据库中发现已经保存有 User 信息了，如图 21-4 所示。登录页面如图 21-5 所示。

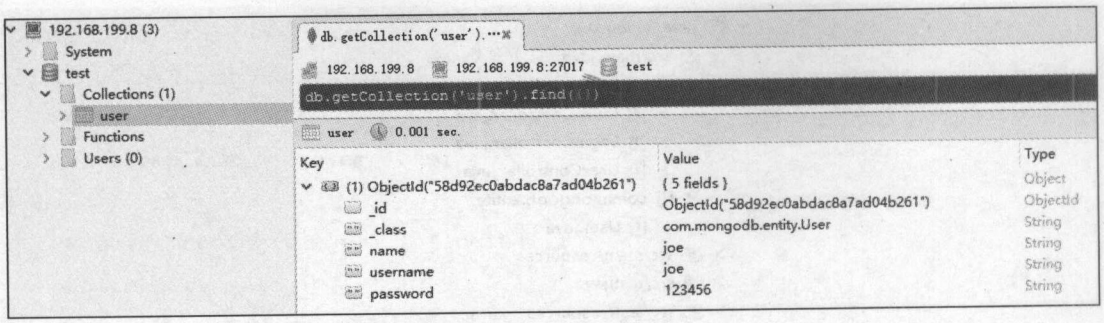


图 21-4　MongoDB 中的 User 信息

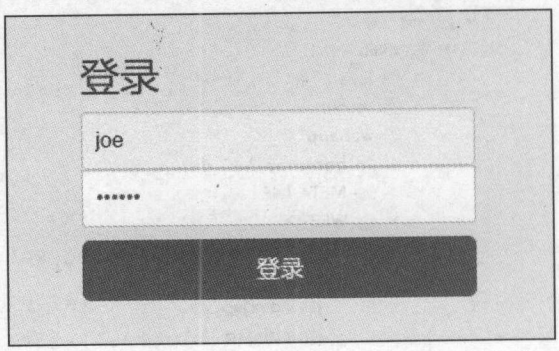


图 21-5　登录页面

使用刚才注册的 username 和 password 登录，登录成功后自动跳转到 Index.jsp 页面，如图 21-6 所示。

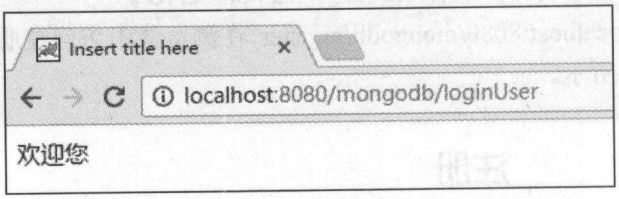


图 21-6　登录成功

21.6 Sping Data MongoDB 操作

本节记录常用的 Sping Data MongoDB 操作。

更多 Spring Data MongoDB 的用法和信息可以在搜索引擎中搜索关键词 Spring Data MongoDB query，或者参考 Spring Data MongoDB 官网文档链接：

```
http://docs.spring.io/spring-data/mongodb/docs/current/reference/html/
```

官网 API 文档链接：

```
http://docs.spring.io/spring-data/data-mongo/docs/2.0.0.M1/api/
```

需要注入 mongodb.xml 中定义的 bean 元素 MongoTemplate，MongoTemplate 中包含了数据库的连接和数据库名等信息：

```
@Autowired
MongoTemplate mongoTemplate;
```

21.6.1　插入数据

```
User user = new User();
user.setName("joe");
user.setPassword("123456");
user.setUsername("joe");
mongoTemplate.save(user);//保存 User 到数据库
```

21.6.2　查询数据

（1）查询第一条记录：

```
mongoTemplate.findOne(new Query(), User.class);
```

（2）查询所有：

```
mongoTemplate.find (new Query(), User.class);
```

（3）与查询：

查询 username 字段为 joe，并且 password 字段为 123456 的文档：

```
Query query = new Query();
query.addCriteria(Criteria.where("username").is("joe"));
query.addCriteria(Criteria.where("password").is("123456"));
mongoTemplate.find(query,User.class);
```

（4）或查询

查询 name 字段为 joe 或者 zoe 的文档：

```
Criteria criteria = new Criteria();
   criteria.orOperator(
              Criteria.where("name").is("joe"),
              Criteria.where("name").is("zoe")
              );
Query query = Query.query(criteria);
mongoTemplate.find(query,User.class);
```

（5）模糊查询

查询 name 字段值有 jo 的文档：

```
String regexName="jo";
Query query = new Query();
query.addCriteria(Criteria.where("name").regex(".*" + regexName + ".*", "i"));
mongoTemplate.find(query,User.class);
```

（6）增加限制条件和排序

查询 name 为 joe 的文档，跳过第 1 条数据，只返回 2 条数据，并且按照_id 降序（Direction.DESC）排序，升序使用 Direction.ASC。

```
Query query = new Query();
query.addCriteria(Criteria.where("name").is("joe"));
query.skip(1).limit(2).with(new Sort(new Sort.Order(Direction.DESC, "_id")));
mongoTemplate.find(query,User.class);
```

（7）地理信息查询

User.java 增加字段 location 如下：

```
private double[] location;
public double[] getLocation() {
        return location;
    }
    public void setLocation(double[] location) {
        this.location = location;
    }
```

赋值时使用普通坐标对：

```
double[] xy=new double[]{-73.92505, 40.8279556};
user.setLocation(xy);
```

保存到数据库后，创建地理位置索引：

```
mongoTemplate.indexOps(User.class).ensureIndex(new
GeospatialIndex("location"));
```

GeospatialIndex 方法默认是使用平面的 2d 类型索引，如果使用的是 GeoJSON 格式的地理位置，建立 2DSPHERE 类型的索引可以使用：

```
mongoTemplate.indexOps(User.class).ensureIndex(new
GeospatialIndex("location").typed(GeoSpatialIndexType.GEO_2DSPHERE));
```

查询位置-72.92505, 40.8279556 附近 2 度以内的点，单位的转换可以参考 10.12.2 小节“管道操作器”中$geoNear 的相关内容，或者附录 A“ MongoDB 地理位置距离单位”。

```
Point point = new Point(-72.92505, 40.8279556);
List<User> us =mongoTemplate.find(new
Query(Criteria.where("location").near(point).maxDistance(2)),User.class);
```

21.6.3　更新数据

注意 Update 引入的包是：

```
import org.springframework.data.mongodb.core.query.Update;
```

而不是 com.mongodb.Update。

（1）单条更新

updateFirst 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。

更新单个字段，把 name 为 joe 的第一条数据 username 修改为 joe009：

```
mongoTemplate.updateFirst(new Query(Criteria.where("name").is("joe")),
            Update.update("username", "joe009"), User.class);
```

更新多个字段，把 name 为 joe 的第一条数据 username 修改为 joe009，password 修改为 23456：

```
Update update = new Update();
update.set("username","joe009");
update.set("password","23456");
mongoTemplate.updateFirst(new
Query(Criteria.where("name").is("joe")),update,User.class);
```

（2）多条更新

更新单个字段，把 name 为 joe 的全部文档 username 修改为 joe009：

```
mongoTemplate.updateMulti(new Query(Criteria.where("name").is("joe")),
            Update.update("username", "joe009"), User.class);
```

更新多个字段，把 name 为 joe 的全部文档 username 修改为 joe009，password 修改为 23456：

```
Update update = new Update();
update.set("username","joe009");
update.set("password","23456");
mongoTemplate.updateMulti(new
Query(Criteria.where("name").is("joe")),update,User.class);
```

21.6.4 删除数据

删除 username 为 joe009 的数据：

```
mongoTemplate.remove(new
Query(Criteria.where("username").is("joe009")),User.class);
```

21.6.5 聚合方法执行

（1）执行 count

查询 name 为 joe 的文档数量：

```
mongoTemplate.count(new Query(Criteria.where("name").is("joe")),User.class);
```

（2）执行 distinct

User 集合中 username 有多少不同值：

```
mongoTemplate. getCollection("user").distinct("username");
```

User 集合中 name 为 joe 的文档 username 有多少不同值：

```
Query query=new Query(Criteria.where("name").is("joe"));
mongoTemplate.getCollection("user").distinct("username",query.getQueryObject()
);
```

（3）执行 mapreduce

根据 name 分组后统计每组的数量：

```
String mapFunction = "function(){ emit(this.name,{'count':1});}";
String reduceFunction = "function(key, values){var sum = 0;
    values.forEach(function(doc){sum += doc.count; }); return
    {groupname:key,total:sum} };";
MapReduceOutput mapReduceOutput =
mongoTemplate.getCollection("user").mapReduce(mapFunction, reduceFunction,
                "resultCollection", null);
DBCollection resultColl = mapReduceOutput.getOutputCollection();
DBCursor cursor = resultColl.find();
while (cursor.hasNext()) {
    System.out.println(cursor.next());
}
```

（4）执行 aggregate

注意使用 match 等静态内置方法需要手动添加引入：

```
import static org.springframework.data.mongodb.core.aggregation.Aggregation.*;
```

Aggregation 在较新的 mongo-java-driver 中才有，所以 pom.xml 中 spring-data-mongodb 的<dependency>前需要加上：

```
<!-- mongo db 驱动-->
<dependency>
   <groupId>org.mongodb</groupId>
   <artifactId>mongo-java-driver</artifactId>
   <version>3.4.2</version>
</dependency>
```

查询出 name 为 joe 的文档按照 username 分组后计算个数赋值给 totalNum，根据 totalNum 降序排序返回结果。

```
TypedAggregation<User> agg = Aggregation.newAggregation(
                    User.class
                    ,match(Criteria.where("name").is("joe"))
                    ,group("username").count().as("totalNum")
                    ,sort(Sort.Direction.DESC, "totalNum")
                    ,project("totalNum")
);

AggregationResults<String> result = mongoTemplate.aggregate(agg,String.class);
for(String dbo : result){
   System.out.println(dbo.toString());
}
```

21.6.6　操作 GridFS

操作 GridFS 需要在 mongodb.xml 中增加 GridFsTemplate 的 bean 声明，同时在需要的地方注入 GridFsTemplate。

mongodb.xml 的头部 beans 中需要加入：

```
xmlns:mongo="http://www.springframework.org/schema/data/mongo"
```

xsi:schemaLocation 参数中需要加入：

```
http://www.springframework.org/schema/data/mongo
http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd
```

如果报错，有可能是版本不匹配，需要检查 spring-context.xsd 是否带有版本号，去掉 spring-context.xsd 的版本号。

mongodb.xml 中<beans></beans>增加：

```
<mongo:db-factory id="mongoDbFactory" dbname="test"  mongo-ref="mongo" />
 <mongo:mapping-converter id="converter" />
   <bean id="gridFsTemplate"
class="org.springframework.data.mongodb.gridfs.GridFsTemplate">
     <constructor-arg ref="mongoDbFactory" />
     <constructor-arg ref="converter" />
   </bean>
```

然后在需要的 class 中注入：

```
@Autowired
GridFsTemplate gridFsTemplate;
```

（1）上传文件

上传文件需要指定文件的路径，我们在 src/main/resources 路径下放一个 test.png 图片，发布到 tomcat 时，它的路径与类的路径一样为 classes，所以使用 this.getClass().getResource("/").getPath()获取到 class 的路径即可找到图片。上传文件使用如下方法：

```
InputStream  inputStream = new
FileInputStream(this.getClass().getResource("/").getPath()+"test.png");
String id =gridFsTemplate.store(inputStream, "test.png",
"image/png").getId().toString();
```

（2）查询文件

```
String id = "58e1eea46464ee7be8ac60aa";
GridFSDBFile gridFsdbFile = gridFsTemplate.findOne(new
Query(Criteria.where("_id").is(id)));
```

（3）下载文件

下载文件需要存储在 GridFS 中的文件的_id，最终可以得到 InputStream，用户再自行处理。

```
String id = "58e1eea46464ee7be8ac60aa";
Query query = new Query(Criteria.where("_id").is(id));
GridFSDBFile gridFsDBFile = this.gridFsTemplate.findOne(query);
InputStream inputStream =gridFsDBFile.getInputStream();
```

（4）删除文件

```
String id = "58e1eea46464ee7be8ac60aa";
gridFsTemplate.delete(new Query(Criteria.where("_id").is(id)));
```

21.6.7　运行示例

学习了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 IndexController 的某个路由方法中，然后在浏览器访问项目调用这个路由路径，查看结果。

对着 com.mongodb 包右击，选择 NEW→新建 Class 文件→命令为 IndexController。

IndexController.java 的代码如下：

```
package com.mongodb;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.query.Criteria;
import org.springframework.data.mongodb.core.query.Query;
import org.springframework.data.mongodb.gridfs.GridFsTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import com.mongodb.entity.User;

@Controller
public class IndexController {
    @Autowired
    GridFsTemplate gridFsTemplate;

    @Autowired
    MongoTemplate mongoTemplate;

    @RequestMapping(value = { "/index" })
    public String index(Model model) {

        User user = new User();
        user.setName("joe");
        user.setPassword("123456");
        user.setUsername("joe");
        mongoTemplate.save(user);// 保存 User 到数据库

        Query query = new Query();
        query.addCriteria(Criteria.where("username").is("joe"));
        query.addCriteria(Criteria.where("password").is("123456"));
        System.out.println(mongoTemplate.find(query,
User.class).get(0).getName());
```

```
        return "/index";
    }

}
```

对着项目右击，选择 Run As→Run on Server，启动项目后，通过浏览器访问下面链接：

```
http://localhost:8080/mongodb/index
```

Console 中输出：

```
joe
```

如果测试其他操作，替换 public String index(Model model) {}方法体中的代码即可。

第四部分

管理与开发经验篇

第 22 章

◂MongoDB开发的经验▸

22.1 尽量选取稳定新版本 64 位的 MongoDB

MongoDB 32 位的版本受到虚拟内存地址大小的限制，单个实例最大数据空间仅为 2GB，64 位基本无限制（128T），故建议使用 64 位计算机部署 MongoDB。MongoDB 的 32 位版本只用于在 32 位的系统上部署测试和开发，不能在正式生产环境中使用。MongoDB 老版本存在一些问题：全局的写入锁、没有集合连接（多集合查询需要查询多次）、数据丢失、安全方面有漏洞等。2015 年一篇博客《别再用 MongoDB 了！》细数了 MongoDB 的诸多罪证，闹得沸沸扬扬。后来 MongoDB 的联合创始人出来澄清了很多问题的原因是由于用户没有正确地配置 MongoDB。MongoDB 在新版本中优化了那些容易造成问题的默认配置，随着版本发布，全局锁也进化到了文档级的锁，关于多表查询也提供了管道聚合左连接，而且优化了引擎，提供了数据压缩比等。所以，选取稳定新版本的 MongoDB，会有更好的用户体验，更少的坑。

22.2 数据结构的设计

MongoDB 弱化了数据结构的模型，也就是我们不需要先设计好每个集合的结构就能使用它，MongoDB 会根据我们的数据自动创建结构，而且提供了内嵌文档等存储方式，非常方便灵活。但是，需要注意的是，数据结构模型的弱化不等于没有数据结构模型。如果要写出好的应用程序以及得到更好的 MongoDB 的性能支持，必须要思考如何来存储数据。

我们来看几个例子，如果我们有一个页面要展示用户的信息数据，最方便的用法就是把所有用户的信息组合成一个实体，保存在一个集合中。如下：

```
{
  _id:<ObjectId>,
  username:"123xyz",
  phone:"123-456-7890",
  email:"xyz@example.com"
}
```

但是现实使用场景并不都是那么简单的，因为数据之间会产生关联。比如我们做购买功能时，需要有订单和产品两个实体。订单实体是需要知道购买了哪些产品实体的。按照我们只使用一个实体做展示的思路，就需要把产品内嵌到订单实体中如下：

```
{
_id:<ObjectId>,
name:"订单1",
status:1,
product:{
    _id:<productaId>,
    name:"产品1",
    price:66
  }
}
```

这样保存数据读取订单展示时很方便，只需要查询一次。但是如果产品 1 的信息有变动，比如名称需要变成产品一，就需要修改库中所有包含产品 1 的订单文档。如果订单数量很多，无疑会造成很大的性能消耗，修改速度慢，容易出问题。 针对这种情况 MongoDB 提供了文档引用功能 DBref。详情可查看 2.5.5 小节自动关联内嵌文档 DBRef 相关内容。在 Spring-Data-MongoDB 中使用 DBref 也很方便，使用@DBref 标签标明字段即可。使用 DBref 保存后的订单文档为：

```
{
_id:<ObjectId>,
name:"订单1",
status:1,
product:{
    "$id":<productaId>,
    "$ref ":"product"
  }
}
```

还需要把产品实体保存在产品集合中：

```
{
    _id:<productaId>,
    name:"产品1",
    price:66
}
```

这时候修改产品名只需要修改产品集合中的 name 即可。订单文档在使用 product 字段时会自动提取最新的数据。DBref 本来是很好的数据结构方式，但是它比较占据数据库的空

间，因为它在订单文档中虽然只保存了引用信息，但是需要分配 product 文档的空间，相当于 product 文档在数据库中存储了 2 份（product 集合中有一份），如果 10000 个订单引用了 product，就占据了 10000 个 product 的文档需要的空间。按照这样的增长方式，是很消耗空间的，所以不要引用不断增加的数据。比如用户的评论，如果把一个用户的评论内容全部用引用的方式内嵌在用户 user 集合中，随着用户的评论数量越来越多，消耗的空间是惊人的，对数据库的性能影响也会越来越大，这时候我们就需要重新组织数据的结构了。

我们可以把订单和产品拆分之后只用它们的 id 做弱关联。订单如下：

```
{
 _id:<ObjectId>,
 name:"订单1",
 status:1,
 productId:productaId
 }
```

产品实体保存在产品集合中：

```
{
    _id:<productaId>,
    name:"产品1",
    price:66
}
```

订单和产品通过 productaId 字段和_id 字段对应起来，这样的数据结构可以很方便地修改产品信息，也不会随着订单量的增加消耗太多的空间。只是查询时有些不方便，需要先查询出订单，再根据关联 id 去查询产品信息才能得到完整的信息。当然，也可以通过管道聚合进行左连接查询。

以上的几种数据结构方式各有优劣，读者需要根据自己应用程序的场景进行设计和选择。

MongoDB 官网中关于数据结构的链接如下：

```
https://docs.mongodb.com/manual/core/data-modeling-introduction/
```

22.3 查询的技巧

（1）限定返回结果条数和字段

MongoDB 提供了 limit 限制返回的条数，并 find 查询时可以设置参数限制只返回哪些字段。

例如：

```
db.user.find({"myName":"joe"},{"age":1}).limit(10)
```

查询 user 集合中 myName 为 joe 的文档，且只返回 age 字段，只返回 10 条记录。

我们在开发过程中如果每次都返回所有文档和所有字段会比较慢，合理地限制返回条数以及只返回需要的字段可以很大程度地提高性能，尤其是单个文档比较大的情况下。

（2）避免使用 skip 跳过大量结果

skip 一般与 limit 配合使用做分页，用 skip 跳过少量的文档是没有问题的。但是，如果文档数量非常多的话，skip 就会变得很慢。因为 skip 会一条一条跳过数据，所以跳过的数据也是需要加载到内存中的，所以会影响性能。这时候我们就要做优化了，文档数量很多的情况下尽量避免使用 skip。通常可以给文档本身内置查询条件，来避免过大的 skip，或者利用上次的结果来计算下一次查询。比如，我们可以根据创建时间排序之后取前 10 个作为第一页。要获取第二页的文档时，把第一页最后的创建时间作为查询条件，就可以获取第二页的文档了，以此类推，我们总可以找到一种方法实现不用 skip 的分页。

（3）避免使用$where

$where 操作符功能强大且灵活，它可以将 JavaScript 表达式和 JavaScript 函数作为查询语句的一部分。在 JavaScript 表达式和函数中，可以使用 this 或 obj 来引用当前操作的文档，所以可以实现非常多的功能。

查询时，$where 操作符不能使用索引，每个文档需要从 BSON 对象转换成 JavaSript 对象后，才可以通过$where 表达式来运行。因此，它比常规查询要慢很多，一般情况下，要避免使用$where 查询。

（4）MapReduce 不能作实时查询

MapReduce 有很好的聚合功能，用来进行统计，非常灵活且易于使用，它可以很好地与分片（sharding）结合使用，并允许大规模输出。但是 MapReduce 在执行过程中需要一定的交互，所以会比较慢。它适合处理大数据量的离线统计分析，不适合需要实时结果的场景。对于大数据的处理，MongoDB 现在可以很好地跟 Hadoop 进行配合使用，有兴趣的读者可以学习。

MapReduce 跟 Hadoop 的结合参见官网链接：

```
https://docs.mongodb.com/ecosystem/use-cases/hadoop/
```

和

```
https://docs.mongodb.com/ecosystem/tools/hadoop/
```

（5）AND 型查询先小后大

假设要查询满足条件 A、B、C 的文档。满足 A 条件的有 60000 个文档，满足 B 条件的有 6000 个，满足 C 条件的有 100 个。如果按照先 ABC 的查询顺序，效率是不高的，如图 22-1 所示。

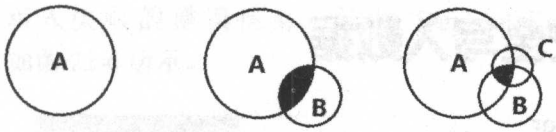


图 22-1　由大到小的 AND 查询

如果把 C 条件放在最前，然后是 B，最后是 A，则只需要查看 100 多个文档即可查询出所需文档，如图 22-2 所示。

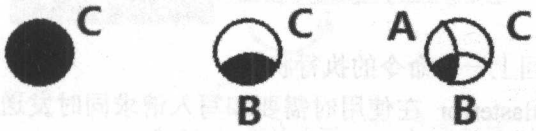


图 22-2　由小到大的 AND 查询

根据条件的先后顺序不同，占用的空间不同，MongoDB 的工作量也不同，所以如果已知某个查询条件比较苛刻，可以放在最前面。

（6）OR 型查询先大后小

OR 型查询与 AND 查询正好相反，匹配多的查询条件应该放在最前面，因为 MongoDB 每次都要匹配不在结果集中的文档。

如果按照 C 或者 B 或者 A 的查询顺序，如图 22-3 所示，深色部分为 MongoDB 下一步要搜索的空间。

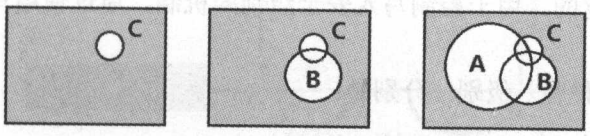


图 22-3　由小到大的 OR 查询

如果先大后小，则可以缩小后续查询条件要搜索的空间，如图 22-4 所示。

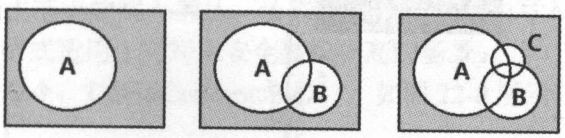


图 22-4　由大到小的 OR 查询

22.4 安全写入数据

（1）使用 getlasterror

为了提高 MongoDB 的工作效率，它对用户发出的更新、插入、删除这些命令都是异步处理的，用户不需要等待返回确认信息。也就是说 MongoDB 的写入操作不返回任何数据库响应，驱动程序也得不到是否成功执行了这个命令的信息。

虽然大多数情况下都是能够执行成功的，在一般的应用开发中不需要关注这个问题，但是有些场景是不能这样处理的。

比如比较敏感的金额操作等，需要得到数据执行操作的确认。针对这种情况，可以使用 getlasterror 命令。

getlasterror 命令返回上一个命令的执行状态。

需要注意的是，getlasterror 在使用时需要和写入请求同时发送才能确保连续执行，期间不会有其他操作插队。驱动程序会自动处理好这些，所以使用者不用特别关心。可以这样理解，如果需要某个写入操作确保操作成功就在使用它时带上 getlasterror 命令，把这样的使用方式当成安全写入即可。

如果有非常重要的数据确保要写入成功，可以使用 getlasterror 搭配 fsync 参数阻塞应用程序的请求，确保数据写入成功。fsync 模式会等待数据都成功写入（至多 100 毫秒），然后才返回成功。要注意的是 fsync 并不是立即将数据写入磁盘，而且阻塞其他请求，直到数据被写入磁盘，所以每次写入数据都使用 fsync，则每平均 100 毫秒才能写入一次。这会让 MongoDB 的性能严重降低，因此要尽量少用 fsync。

（2）MongoDB 的写安全机制

还有另外一种方式确保写入的安全，就是使用 Write Concern（写入安全）机制。写入安全是一种由客户端设置的，用于控制写入安全级别的机制，通过使用写入安全机制可以提高数据的可靠性。

MongoDB 提供 4 种写入级别，分别是：

（1）非确认式写入（Unacknowledged）

默认为非确认式写入，使用命令：{ writeConcern:{w:0}}。

非确认式写入不返回响应结果，如图 22-5 所示。

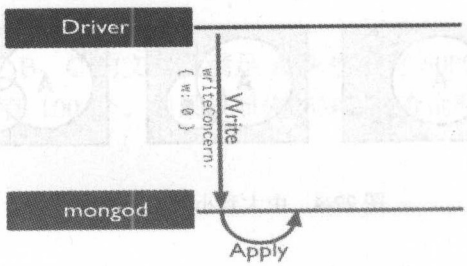


图 22-5　非确认式写入

（2）确认式写入（Acknowledged）

确认式写入返回写入失败的错误信息，比如 DuplicateKey Error。使用命令：{ writeConcern:{w:1}}，如图 22-6 所示。

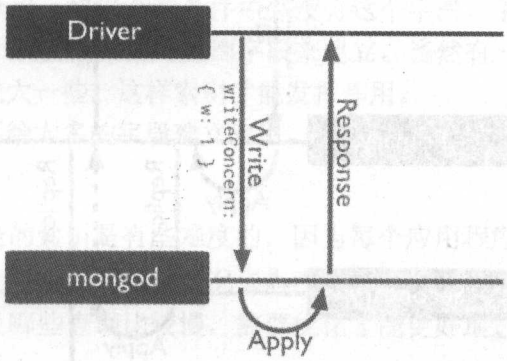


图 22-6　确认式写入

（3）日志写入（Journaled）

一般的写入完成只是写入到内存中，并没有持久化到硬盘，日志写入模式会写入完成之后把记录保存到 journal 日志后才返回响应结果，这种写入方式能够承受服务器突然断电崩溃，更有效地保障数据的安全。

日志写入使用命令：{ writeConcern:{w:1,j:true}}，如图 22-7 所示。

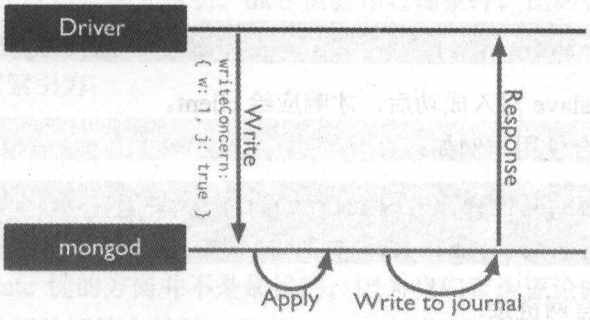


图 22-7　日志写入

（4）复制集确认式写入（Replica Acknowledged）

写操作不仅要得到主节点的写入确认，还需要得到从节点的写入确认，这里还可以设置写入节点的个数。这种方式适用于对写入安全要求更高的场景。

复制集确认式写入命令：{ writeConcern:{w:2}}，如图 22-8 所示。

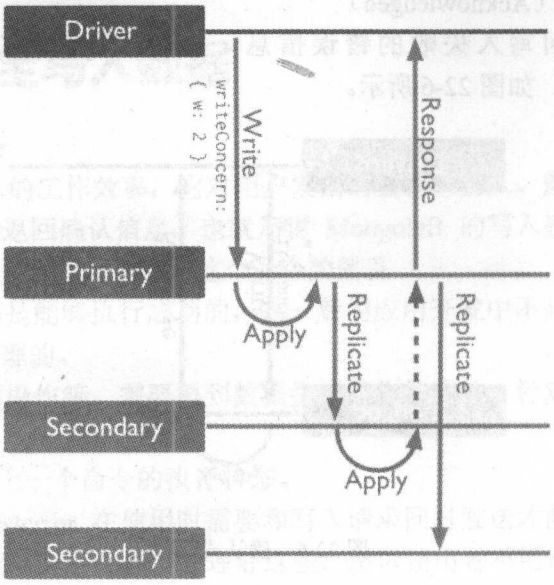


图 22-8　复制集确认式写入

还可以使用以下配置：

```
{ writeConcern:{wtimeout:5000}}
```

用于设置超时处理，本命令设置 5000 毫秒未完成写操作，则报超时错误。

```
{writeConcern:{w:"majority"}}
```

复制集中大多数 slave 写入成功后，才响应给 client。

写入级别可以组合使用，例如：

```
{writeConcern:{w:"majority",wtimeout:5000}})
{writeConcern:{w:1,j:true,wtimeout:5}}
```

更多信息可参考官网链接：

```
https://docs.mongodb.com/v2.4/core/write-concern/
```

22.5 索引设置的技巧

（1）索引的重要

如果某个不用索引，MongoDB 会做全集合扫描，逐个扫描文档，遍历整个集合，才能找到结果。如果数据量非常大的时候，会很慢。比如 260GB 的数据，MongoDB 会把它们加载到内存中扫描（如果操作系统内存是 16GB，系统会自动将旧的内存页面换出去），但是如

果有正确的索引，只需要加载索引数据进来（260GB 的数据索引可能只有 60GB），需要加载的数据就小了很多，而且索引有顺序，可以快速定位查询到文档，返回结果。

（2）复合索引

给某个字段建立了索引，如果查询条件中并没有这个字段，那索引文件就白加载了，所以索引的选择很重要，一般要根据常用查询字段来建立。当然有一个技巧是建立复合索引，字段对应上索引的概率就大一些，这样索引才能发挥作用。

需要注意的是，不要给太多的字段建立索引，否则索引文件本身就非常大。

（3）索引的选择

要掌握如何设置最佳的索引是有些难度的，因为每个应用程序的查询场景不同，所以没有一个通用的最佳方案，只能根据实际情况去思考。我们需要知道今后会做哪些查询，哪些内容需要快速查找，以及哪些查询比较慢，需要优化才能更好地选择索引。所以，建立索引时应该考虑以下几个问题：

- 会做什么样的查询?其中哪些键需要索引?
- 每个键的索引方向是怎样的?
- 有没有不同的键排列可以使常用的数据更多地保留在内存中？

如果三个问题都有明确的答案，说明我们能建立一个比较优质的索引。举例来说：我们有一个用户评论集合，里面有字段 username、content 和 date。我们会按照 username 去查询某个用户的评论，username 需要设置索引。对于 content 评论内容字段，我们几乎不会使用它做查询条件，所以 content 不需要建立索引。date 也会作查询条件，比如找到最新的评论，或者会根据 date 作排序，所以 date 需要设置索引。

所以我们可以设置索引为：

```
db.usercontent.ensureIndex({"username":1,"date":1})
```

索引 username 和 date 按升序排序。先按 username 升序排，同组 username 的按 date 升序排。

但是这样的设置 date 键的方向并不是最优的，因为我们查询评论时一般是只展示比较新的评论，date 升序排最新的评论会被排在最后，date 应该按降序排，这样才能更快地查询到我们需要的数据，所以使用的索引应该是：

```
db.usercontent.ensureIndex({"username":1,"date":-1})
```

这样的索引是最优的吗？我们还应该思考键的前后位置。比如我们要查询用户和日期，取出某一用户最近的评论。MongoDB 按照我们索引键的前后顺序先查询到对应的用户名，再在同组用户名中去查询出最近的评论。如果我们要查询的用户每次都不相同，则内存中存放的索引页每次都需要替换掉。如果调整一下索引的前后顺序，把 date 放在前面，使用索引：

```
db.usercontent.ensureIndex({"date":-1,"username":1})
```

这种情况下内存中会优先保存有最近几天的用户索引，无论我们查询哪个用户的最近评

论，都不需要替换内存中的索引页，可以有效地减少内存交换，这样查询任何用户的最新评论都会快很多。

（4）建索引导致数据库阻塞的解决

建索引就是一个容易引起长时间写锁的问题，MongoDB 在前台建索引时需要占用一个写锁（而且不会临时放弃），如果集合的数据量很大，建索引通常要花比较长的时间，特别容易引起问题。

解决的方法很简单，MongoDB 提供了两种建立索引的方式：一种是 background 方式，不需要长时间占用写锁；另一种是非 background 方式，需要长时间占用锁。使用 background 方式就可以解决问题。

例如，为超大表 posts 建立索引， 千万不要使用如下代码：

```
db.posts.ensureIndex({user_id: 1})
```

而应该使用如下代码：

```
db.posts.ensureIndex({user_id: 1}, {background: 1})
```

（5）索引调优

索引设置好了之后，还需要对它进行跟踪优化。explain 是一个很好用的命令，在 find 查询时使用 explain 会输出查询的细节，包括索引的情况、耗时以及扫描文档数的统计信息。有时候我们设置了一个索引，并不知道某个查询会不会使用到它，就需要使用 explain 查看。

设置索引使用命令如下：

```
db.usercontent.ensureIndex({"username":1})
```

设置索引 username。

查询分析使用命令如下：

```
db.usercontent.find({"username":"joe"}).sort({"username":1}).explain();
```

输出如下：

```
{
        "queryPlanner" : {
                "plannerVersion" : 1,
                "namespace" : "test.usercontent",
                "indexFilterSet" : false,
                "parsedQuery" : {
                        "username" : {
                                "$eq" : "joe"
                        }
                },
                "winningPlan" : {
                        "stage" : "FETCH",
```

```
                "inputStage" : {
                        "stage" : "IXSCAN",
                        "keyPattern" : {
                                "username" : 1
                        },
                        "indexName" : "username_1",
                        "isMultiKey" : false,
                        "multiKeyPaths" : {
                                "username" : [ ]
                        },
                        "isUnique" : false,
                        "isSparse" : false,
                        "isPartial" : false,
                        "indexVersion" : 2,
                        "direction" : "forward",
                        "indexBounds" : {
                                "username" : [
                                        "[\"joe\", \"joe\"]"
                                ]
                        }
                }
        },
        "rejectedPlans" : [ ]
    },
    "serverInfo" : {
        "host" : "mongodb0",
        "port" : 27017,
        "version" : "3.4.2",
        "gitVersion" : "3f76e40c105fc223b3e5aac3e20dcd026b83b38b"
    },
    "ok" : 1
}
```

indexBounds 结果参数中说明了当前查询具体使用的索引。

如果发现 MongoDB 使用了非预期的索引，或者读者觉得某种情况下使用另一个索引效果会更好，可以使用 hint 参数强制使用某个索引。

设置索引使用命令如下：

```
db.usercontent.ensureIndex({"date":1})
```

需要注意的是，强制使用索引前必须保证已经建立了该索引，否则会报 planner returned

error: bad hint 错误。

强制使用索引使用命令如下：

```
db.usercontent.find({"username":"joe"}).sort({"username":1}).hint({"date":1});
```

强制使用{"date":1}索引。

相关 explain 结果参数和更多信息可查看官网链接：

```
https://docs.mongodb.com/manual/reference/explain-results/
```

（6）不适合使用索引的情况

设置索引的好处显而易见，但是下面两种情况最好不使用索引：

- 集合中的数据每次查询都需要返回大部分的文档。

比如数据集合的大小是 260GB，我们需要查询出 90%的文档，如果使用了索引，则需要把 60GB 的索引先加载到内存中，然后按照索引的指针去加载 230GB 的集合，一共需要加载 290GB 的数据才能返回所需文档。所以，索引一般用在返回结果只是总体数据的一部分的情况。根据经验，一旦要返回大约集合一半的数据时就不要使用索引了。如果已经对某个字段建立了索引，又想在大规模查询时不使用它，可以在查询时加参数禁用索引。使用自然排序{"$natural":1}即可禁用索引：

```
db.test.find().sort({"$natural":1});
```

- 写比读多

如果对该集合的写操作比读操作多时，就尽量不要添加索引，因为索引越多，写的操作就会越慢。比如操作日志集合，我们不要经常去读，只是做一个记录，这种情况不需要设置索引。如果集合读的量非常大，才需要通过创建索引来提高查询效率。

22.6 不要用 GridFS 处理小的二进制文件

GridFS 取文件时需要做两次查询，先查出文件信息的元数据，然后根据元数据去查询出内容。所以 GridFS 比普通的二进制存储会慢一些，GridFS 是用来存放大文件的，至少一个文档（16MB）存放不下的情况适合使用 GridFS。小于文档大小限制的文件，使用一般的二进制存储到 MongoDB 即可，比如图片、声音，甚至小的视频需要一次性加载在页面中的小文件，都应该使用二进制的存储方式以利于加载。而需要用户下载的大文件则适合使用 GridFS 存储，等用户发出下载请求时再去访问 GridFS。