# 7.redis

## 7.1.介绍

**什么是redis**

是一个支持nosql，key-value数据类型，内存，可以持久化，作为缓存，分布式的非关系型数据库

**nosql、key-value、非关系型数据库**

结构化数据、非结构化数据、半结构化数据

结构化数据：数据库数据

非结构化数据：图片、视频、音频

半结构化数据：json、xml、html

nosql（not only sql）：

mysql、oracel是典型的关系型数据库，使用的是sql语言操作，一旦涉及到非、半结构化数据，无能为力

使用非关系型数据库（nosql）存储非、半结构化数据，不能使用sql，常用结构就是key-value，非关系型数据库例如：redis、HBase、MongoDB

**内存**

redis在运行期间，使用内存存储数据

优点：非常快

缺点：容量小，断电丢失

**可持久化**

存储在redis内存中的数据，可以设置持久化策略，在断电内存数据丢失时，重启redis，可以从磁盘文件重新读取数据。

**分布式**

可以使用redis搭建分布式集群，3.0之后的cluster（集群）支持1000个节点

**redis支持的功能**

1.缓存，提示系统的访问性能（减少层直接的调用）

2.低配的消息队列

3.实现各种逻辑锁

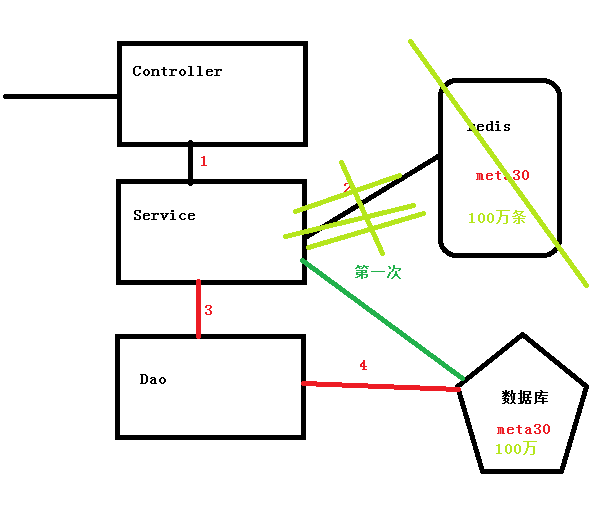
4.解决session共享

**同类产品**

redis和memoryCache对比

|  |  |  |  |
| --- | --- | --- | --- |
|  | redis | memoryCache | 总结 |
| 性能 | 单实例，单线程，非阻塞技术  优点：cpu开销小，省去切换的开销，和线程资源争抢的开销  缺点：海量数据性能稍比memoryCache低 | 多线程技术，数据量大时，性能占优，但是缺点是cpu消耗比redis高 | 吞吐量（单位时间内的数据读写量）都很大，redis是秒级万条数据（由于网络原因，一般达不到） |
| 数据结构 | redis支持String，hash，list，set，set，Stream | 只支持String | memoryCache对数据的局限性大 |
| 持久化 | 支持持久化 | 不支持持久化，很难解决缓存雪崩的情况 |  |

缓存雪崩\缓存击穿：



海量请求访问系统时，由于缓存服务器宕机（缓存数据未命中）（数据丢失），大量请求涌入数据库，造成数据库压力暴增，数据库无法承受这种压力，导致数据库宕机，重启（缓存未命中）-宕机-重启-宕机，导致系统崩溃

缓存雪崩一旦出现，后果不堪设想（很大的损失）

## 7.2.redis安装和启动

### 7.2.1.redis安装

1.上传解压安装包

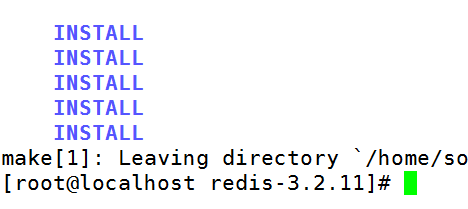
[root@localhost ~]# cp /home/resources/redis-3.2.11.tar.gz /home/software/

[root@localhost software]# tar -xvf redis-3.2.11.tar.gz

2.进入redis根目录，执行编译和编译安装

[root@localhost software]# cd redis-3.2.11/

[root@localhost redis-3.2.11]# make && make install



### 7.2.2.redis的启动与登录

**启动服务端**

启动redis的单节点服务

语法：redis-server 配置文件（默认可以不加）

[root@localhost redis-3.2.11]# redis-server



启动端口默认6379

**启动客户端**

语法：redis-cli -p（服务端的端口号，默认6379） -h（服务端ip，默认127.0.0.1）

由于xshell当前窗口被服务端占用，再开一个窗口

[root@localhost ~]# redis-cli

127.0.0.1:6379>

**简单测试redis命令**

增：

127.0.0.1:6379> set name zhangsan

OK

查：

127.0.0.1:6379> get name

"zhangsan"

改：

127.0.0.1:6379> set name lisi

OK

删：

127.0.0.1:6379> del name

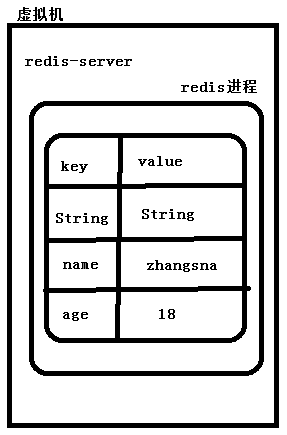
(integer) 1

## 7.3.redis命令

操作redis的数据，redis具备五种数据类型，针对key-value中的value，应对不同的使用场景

**String、Hash、List**、Set、Zset

### 7.3.1.String类型（包括基础命令）



课前资料预习笔记中

**keys \*** :查看当前redis节点（启动的每一个redis服务都叫redis节点。）所有已存在的key值。\*是一个pattern name\*,haha\* \*.block

127.0.0.1:6379> keys \*

(empty list or set)

127.0.0.1:6379> set name hanlaoshi

OK

127.0.0.1:6379> set location beijing

OK

127.0.0.1:6379> keys \*

1) "location"

2) "name"

127.0.0.1:6379>

**set key value:**存储一个key-value结构的数据（**String**）,覆盖一个同名key值得数据

**EX seconds – 设置键key的过期时间，单位时秒**

**PX milliseconds – 设置键key的过期时间，单位时毫秒**

**NX – 只有键key不存在的时候才会设置key的值**

**XX – 只有键key存在的时候才会设置key的值**

127.0.0.1:6379> set age 22

OK

127.0.0.1:6379> keys \*

1) "location"

2) "age"

3) "name"

127.0.0.1:6379> get age

"22"

127.0.0.1:6379> set age 33

OK

127.0.0.1:6379> get age

"33"

127.0.0.1:6379>

**get key:**获取当前key对应的value数据

127.0.0.1:6379> get age

"22"

127.0.0.1:6379> set age 33

OK

127.0.0.1:6379> get age

"33"

127.0.0.1:6379>

**select 整数：**redis中默认存在16个数据分库（database），index号0-15，在一个服务器节点上，可以区分多种类型，多种功能的数据库仓库，默认登录的是0号分库(其他库使用不多)

127.0.0.1:6379> select 1

OK

127.0.0.1:6379[1]> keys \*

(empty list or set)

127.0.0.1:6379[1]> select 0

OK

127.0.0.1:6379> keys \*

1) "location"

2) "age"

3) "name"

127.0.0.1:6379>

**exists key：**判断当前节点是否包含key的数据。

exists比get效率高

get也可以完成这个任务。redis在key-value结构的value存储容量是1G。使用get判断存在会先读在判断数据是否为空，浪费资源

127.0.0.1:6379> exists haha kaka age

(integer) 1

127.0.0.1:6379> exists age

(integer) 1

127.0.0.1:6379> exists haha

(integer) 0

127.0.0.1:6379>

**del key:**删除key值对应的key-value数据；

127.0.0.1:6379> del age

(integer) 1

127.0.0.1:6379> del haha

(integer) 0

127.0.0.1:6379> keys \*

1) "location"

2) "name"

127.0.0.1:6379>

**type key:查看当前key对应value的类型**

127.0.0.1:6379> type name

string

127.0.0.1:6379>

**save：将当前的内存数据保存到磁盘文件（根目录dump.rdb）**

127.0.0.1:6379> keys \*

1) "location"

2) "name"

127.0.0.1:6379> save

OK

127.0.0.1:6379>

**save前，save后观察redis根目录的dump.rdb大小可以发现已经存储到持久化文件**

**flushall:将当前redis所有数据清空，包括持久化文件，内存数据（创建集群的时候使用）**

127.0.0.1:6379> keys \*

1) "location"

2) "name"

127.0.0.1:6379> select 1

OK

127.0.0.1:6379[1]> set name wanglaoshi

OK

127.0.0.1:6379[1]> set age 11

OK

127.0.0.1:6379[1]> keys \*

1) "age"

2) "name"

127.0.0.1:6379[1]> save

OK

127.0.0.1:6379[1]> flushall

OK

127.0.0.1:6379[1]> keys \*

(empty list or set)

127.0.0.1:6379[1]> select 0

OK

127.0.0.1:6379> keys \*

(empty list or set)

127.0.0.1:6379>

**flushdb:清空一个分库的数据，不删除持久化文件内容（进行测试使用）**

127.0.0.1:6379[1]> flushdb

OK

127.0.0.1:6379[1]> keys \*

(empty list or set)

127.0.0.1:6379[1]> select 0

OK

127.0.0.1:6379> keys (

(empty list or set)

127.0.0.1:6379> keys \*

1) "name"

127.0.0.1:6379>

redis中string字符串如果是纯数字，也有一些简单的数字操作命令

**incr\decr key：**自增1，自减1；计步器

127.0.0.1:6379> set num 100

OK

127.0.0.1:6379> incr num

(integer) 101

127.0.0.1:6379> decr num

(integer) 100

127.0.0.1:6379>

**多步计步器：incrby\decrby key 整数：对key的value数字自增自减去整数步数**

**127.0.0.1:6379> incrby num 10**

(integer) 110

127.0.0.1:6379> decrby num 10

(integer) 100

127.0.0.1:6379>

**mset mget：**单节点批量操作（不支持多节点分布式，只能对本节点数据进行处理）(测试时确定当前节点拥有的数据)

127.0.0.1:6379> mset num1 200 num2 300 num3 400

OK

127.0.0.1:6379> keys \*

1) "num3"

2) "name"

3) "num"

4) "100"

5) "num1"

6) "num2"

127.0.0.1:6379> mget num1 num2 num3

1) "200"

2) "300"

3) "400"

127.0.0.1:6379>

**redis作为内存缓存数据库虽然速度快,还是有缺点的,内存容量,不可能比硬盘容量大的.redis最常见的保护内存的机制(热点数据保存,冷点数据超时清除)**

**expire key:对key添加超时过期的设定**

**配合ttl key可以查看 key值剩余时间,ttl剩余时间是-2,表示已经超时的内容删除，-1表示永久数据**

127.0.0.1:6379> ttl num1

(integer) -2

127.0.0.1:6379> keys \*

1) "num3"

2) "name"

3) "num"

4) "100"

5) "num2"

127.0.0.1:6379> ttl name

(integer) -1

127.0.0.1:6379>

**pexpire:设置超时的毫秒数**

**append key value** :对key对应的value进行追加数据操作;

127.0.0.1:6379> get name

"zhangsan"

 127.0.0.1:6379> append name feng

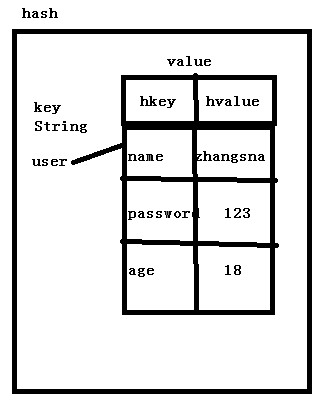
(integer) 12

127.0.0.1:6379> get name

"zhangsanfeng"

### 7.3.2.Hash数据类型

对应value为hash结构的数据都是面向对象的数据结构



**hset key field value:key 是 相当于变量名，filed是value中的key-value对中的key，但是面向对象时表示一个对象的属性名称， value表示属性的值**

127.0.0.1:6379> hset user username hanlaoshi

(integer) 1

127.0.0.1:6379> hset user age 18

(integer) 1

127.0.0.1:6379> hset user location beijing

(integer) 1

127.0.0.1:6379> type user

hash

127.0.0.1:6379>

**hget key field**

127.0.0.1:6379> hget user username

"hanlaoshi"

127.0.0.1:6379>

**hexists key field:判断hash类型中的属性值是否存在**

127.0.0.1:6379> hexists user haha

(integer) 0

127.0.0.1:6379> hexists user age

(integer) 1

127.0.0.1:6379>

**hmset和hmget：批量设置，不支持分布式**

127.0.0.1:6379> hmset student age 19 name wangxiaoxiao gender male

OK

127.0.0.1:6379> hmget student age name gender

1) "19"

2) "wangxiaoxiao"

3) "male"

127.0.0.1:6379>

**Hdel key field:删除属性和值**

127.0.0.1:6379> hdel student age

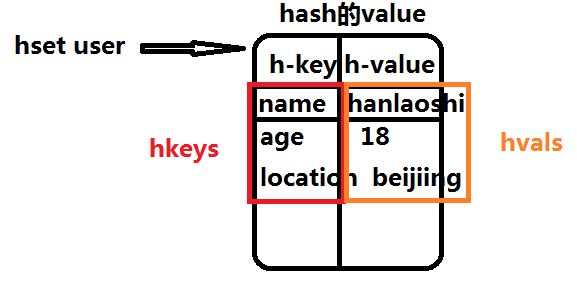
(integer) 1

127.0.0.1:6379> hget student age

(nil)

127.0.0.1:6379>

**Hkeys Hvals:单独获取对象的属性名，或者属性的值**



127.0.0.1:6379> hkeys user

1) "name"

2) "username"

3) "age"

4) "location"

127.0.0.1:6379> hvals user

1) "haha"

2) "hanlaoshi"

3) "18"

4) "beijing"

127.0.0.1:6379>

**Hlen key：获取属性个数（长度）**

127.0.0.1:6379> hlen user

(integer) 4

127.0.0.1:6379> hlen student

(integer) 2

127.0.0.1:6379>

**Hincrby:和String类型特性区别的一个体现**

String对应存储对象数据：需要在redis中修改对象某个属性的数值（10-15）

{"id":1623125,"age"：10,“name”:"hanlaoshi"}

不能直接在redis中获取age将10变成15，

从redis读出来，转化成对象，对象set属性，重新放回redis

和hash存储对象数据

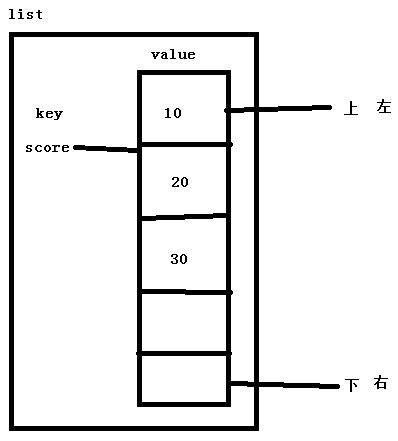
user id 1623125

age 10

name hanlaoshi

Hincreby user name 5

### 7.3.3.List链表数据结构



**list01--{“200”,"300","400"}**

查看链表list和设置链表数据Lpush（左表示上，右表示下）

**lpush key value [vlaues]:对于一个链表,从左(上)开始插入数据;**

 127.0.0.1:6379> lpush score 100 200 300

**lrange key 起始下标 结束下标（展示范围，结束下标是-1表示到尾部）**

127.0.0.1:6379> lpush mylist01 100 200 300

(integer) 3

127.0.0.1:6379> lrange mylist 0 1

(empty list or set)

127.0.0.1:6379> lrange mylist01 0 1

1) "300"

2) "200"

127.0.0.1:6379>

**Rpush key value：**从下向上插入数据

127.0.0.1:6379> rpush mylist01 400

(integer) 4

127.0.0.1:6379> lrange mylist01 0 -1

1) "300"

2) "200"

3) "100"

4) "400"

127.0.0.1:6379>

**Linsert：**从链表中间操作数据,从左侧找到第一个相同元素插入数据（before after）

linsert mylist01 before 100 one

在mylist01链表中，找到100这个元素，从前插入数据one 300 200 one 100 400

127.0.0.1:6379> linsert mylist01 before 100 one

(integer) 5

127.0.0.1:6379> lrange mylist01 0 -1

1) "300"

2) "200"

3) "one"

4) "100"

5) "400"

127.0.0.1:6379> lpush mylist01 100

(integer) 6

127.0.0.1:6379> lrange mylist01 0 -1

1) "100"

2) "300"

3) "200"

4) "one"

5) "100"

6) "400"

127.0.0.1:6379> linsert mylist01 after 100 two

(integer) 7

127.0.0.1:6379> lrange mylist01 0 -1

1) "100"

2) "two"

3) "300"

4) "200"

5) "one"

6) "100"

7) "400"

127.0.0.1:6379>

**lset key index value将下标对应的数据修改**

127.0.0.1:6379> lset mylist01 1 2

OK

127.0.0.1:6379> lrange mylist01 0 -1

1) "100"

2) "2"

3) "300"

4) "200"

5) "one"

6) "100"

7) "400"

127.0.0.1:6379>

**Lrem:**这个没有rrem，内部逻辑涉及到了从下往上删除

lrem key count value：从左向右寻找count个value相同值元素删除。

count>0从左往右删

count=0 全删

count<0从右往左删

127.0.0.1:6379> lpush mylist01 200 200 200

(integer) 9

127.0.0.1:6379> lrange mylist01 0 -1

1) "200"

2) "200"

3) "200"

4) "2"

5) "300"

6) "200"

7) "one"

8) "100"

9) "400"

127.0.0.1:6379> lrem mylist01 -2 200

(integer) 2

127.0.0.1:6379> lrange mylist01 0 -1

1) "200"

2) "200"

3) "2"

4) "300"

5) "one"

6) "100"

7) "400"

127.0.0.1:6379> lrem mylist01 0 200

(integer) 2

127.0.0.1:6379>

**ltrim** 保留链表的范围内的数据元素

ltrim key 起始位置 结束位置（-1表示到尾部）

127.0.0.1:6379> ltrim mylist01 4 6

OK

127.0.0.1:6379> lrange mylist01 0 -1

1) "400"

127.0.0.1:6379>

**lpop** 从list头部删除元素，返回删除结果（类似remove） 秒杀逻辑,可以利用list对象实现阻止超卖现象

127.0.0.1:6379> lrange mylist01 0 -1

1) "six"

2) "five"

3) "four"

4) "three"

5) "two"

6) "one"

7) "400"

127.0.0.1:6379> lpop mylist01

"six"

127.0.0.1:6379> lrange mylist01 0 -1

1) "five"

2) "four"

3) "three"

4) "two"

5) "one"

6) "400"

127.0.0.1:6379>

两个链表的数据交互

**rpoplpush**：从第一个list的尾部删除数据，移动到第二个list的头部添加,这个命令的过程是原子级的

127.0.0.1:6379> rpoplpush mylist01 mylist02

"400"

127.0.0.1:6379> lrange mylist02 0 -1

1) "400"

2) "5"

3) "4"

4) "3"

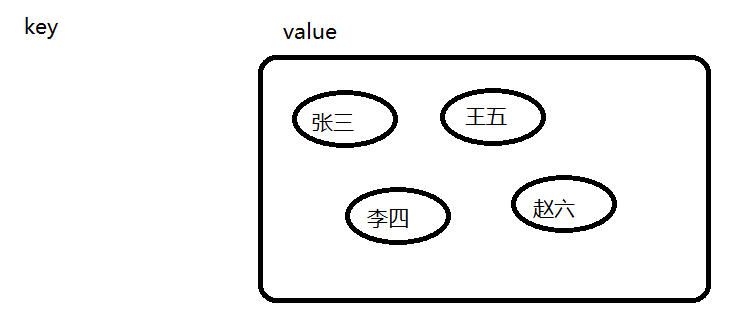
5) "2"

6) "1"

127.0.0.1:6379>

### 7.3.4.Set结构

集合用来保存多个字符串元素，和列表不同的是不允许有重复元素，并且集合中元素是无序的



**最常见的应用场景，比如爱好，兴趣，社交平台中关注人存在一个集合中，将其所有粉丝存在一个集合**

**单集合操作**

**sadd key member添加**

127.0.0.1:6380> sadd favor histroy english math

(integer) 3

127.0.0.1:6380>

**srem key member删除**

127.0.0.1:6380> srem favor histroy

(integer) 1

**scard key** 返回元素个数

127.0.0.1:6380> scard favor

(integer) 2

**sismember key element判断元素是否在集合中**

127.0.0.1:6380> sismember favor histroy

(integer) 0

127.0.0.1:6380> sismember favor math

(integer) 1

127.0.0.1:6380>

**srandmember key 随机抽取几个元素**

127.0.0.1:6380> sismember favor math

(integer) 1

127.0.0.1:6380> srandmember favor 1

1) "math"

127.0.0.1:6380> srandmember favor 1

1) "english"

**smembers key获取所有元素**

127.0.0.1:6380> sadd favor english histry

(integer) 2

127.0.0.1:6380> smembers favor

1) "english"

2) "histry"

3) "math"

127.0.0.1:6380>

**集合间的操作**

**sinter set1 set2 交集**

127.0.0.1:6380> sadd favor1 english histroy math

(integer) 3

127.0.0.1:6380> sadd favor2 english math chinese

(integer) 3

127.0.0.1:6380> sinter favor1 favor2

1) "english"

2) "math"

127.0.0.1:6380>

**sunion set1 set2 并集**

127.0.0.1:6380> sunion favor1 favor2

1) "english"

2) "chinese"

3) "histroy"

4) "math"

127.0.0.1:6380>

**sdiff set1 set2 差集属于set1不属于set2的元素为差集**

127.0.0.1:6380> sdiff favor1 favor2

1) "histroy"

127.0.0.1:6380> sdiff favor2 favor1

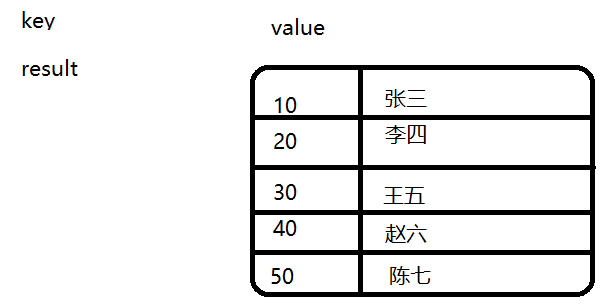
1) "chinese"

127.0.0.1:6380>

### 7.3.5.ZSet结构（sorted set）

在set的基础上绑定一个score作为排序的依据

比较典型的使用场景是排行榜系统。例如视频网站需要对用户上传的视频做排行榜榜



单集合（集合内）

**zadd key score memeber添加成员**

127.0.0.1:6380> zadd result 10 lilaoshi

(integer) 1

127.0.0.1:6380> zadd result 20 wanglaoshi

(integer) 1

127.0.0.1:6380> zadd result 30 chenlaoshi

(integer) 1

127.0.0.1:6380>

**zscore key member计算某个成员的分数**

127.0.0.1:6380> zadd result 30 chenlaoshi

(integer) 1

127.0.0.1:6380> zscore result chenlaoshi

"30"

127.0.0.1:6380> zadd result 40 chenlaoshi

(integer) 0

127.0.0.1:6380> zscore result chenlaoshi

"40"

127.0.0.1:6380>

计算成员的排名 zrank key member

127.0.0.1:6380> zrank result lilaoshi

(integer) 0

127.0.0.1:6380> zrank result wanglaoshi

(integer) 1

127.0.0.1:6380>

**删除成员 zrem key member**

127.0.0.1:6380> zrem result wanglaoshi

(integer) 1

127.0.0.1:6380> zrank result wanglaoshi

(nil)

127.0.0.1:6380>

**增加成员的 分数zincrby key increment** member

127.0.0.1:6380> zincrby result 50 lilaoshi

"60"

127.0.0.1:6380>

**返回指定排名范围的成员 zrange key start end**

127.0.0.1:6380> zrange result 0 1

1) "chenlaoshi"

2) "lilaoshi"

127.0.0.1:6380>

**返回指定分数范围的成员 zrangebyscore key min max**

返回指定分数范围成员个数 zcount key min max

**删除指定排名内的升序元素 zremrangebyrank key start end**

127.0.0.1:6380> zremrangebyrank result 0 1

(integer) 2

127.0.0.1:6380> zrange result 0 1

(empty list or set)

127.0.0.1:6380>

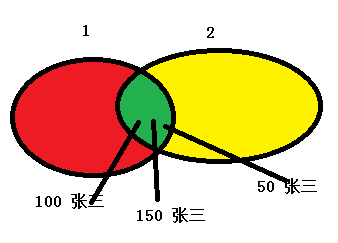
**删除指定分数范围的成员 zremrangebyscore key min max**

**2 集合间的操作**

**交集 zinterstore destination numkeys key**

制定一个交集并集查询结果的输出集合desitination

指定查询交集并集的总zset个数numkeys ,给定符合数量的多个zset集合



127.0.0.1:6380> zadd result1 10 li

(integer) 1

127.0.0.1:6380> zadd result1 20 wang

(integer) 1

127.0.0.1:6380> zadd result2 30 li

(integer) 1

127.0.0.1:6380> zadd result2 40 wang

(integer) 1

127.0.0.1:6380> zinterstore out1 2 result1 result2

(integer) 2

127.0.0.1:6380> zrange out1 0 -1 withscores

1) "li"

2) "40"

3) "wang"

4) "60"

127.0.0.1:6380>

并集 zunionstore destionation numkeys key

## 7.4.多实例部署

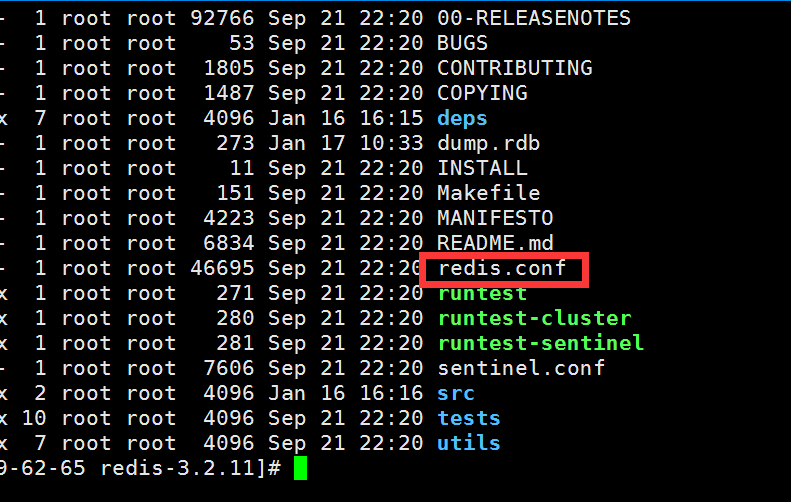
### 7.4.1.多实例部署意义

redis是单实例,单线程的技术,一个服务器中只启动一个redis的server不足以充分发挥一个服务器的性能的;所以一般在同一台服务器上,根据服务器性能多启动几个redis实例同时运行,至少端口不能冲突,所以,需要利用对redis的配置文件了解,搭建一下3个节点的redis

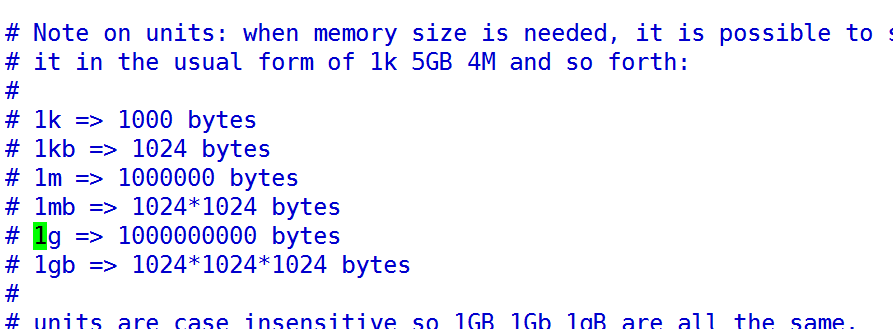
### 7.4.2.redis的配置文件修改

**配置文件学习**

/redis根目录/redis.conf 是配置文件的模板

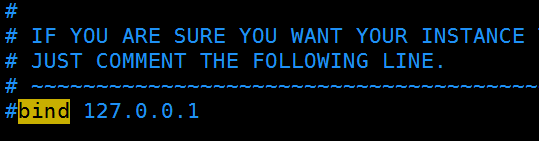


p11行



redis中的数据单位解释

p61 bind 用#注释掉



如果需要绑定监听的ip（客户端只有通过被绑定的ip才可以利用

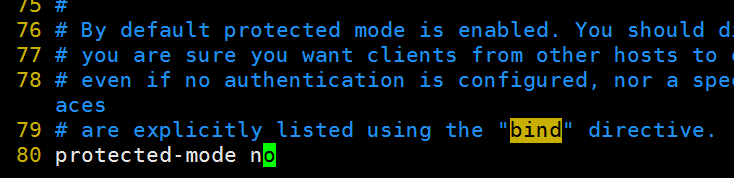
redis-cli -h ip地址链接服务器）

bing 127.0.0.1 106.75.101.219(外网可访问当前服务器的ip)

一旦用#注释bind，没有任何限制，只要可以链接服务器，都允许使用

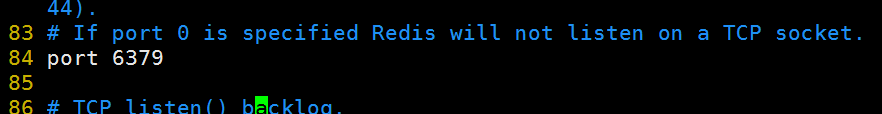
redis 建议一旦关闭保护模式,配置一个requirepass的授权密码,否则不安全

p80 保护模式不启动



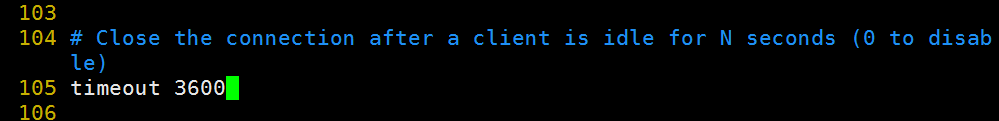
保护模式开启，需要登录密码，改成no

p84 6379是默认端口(要启动其他的redis实例需要修改端口)



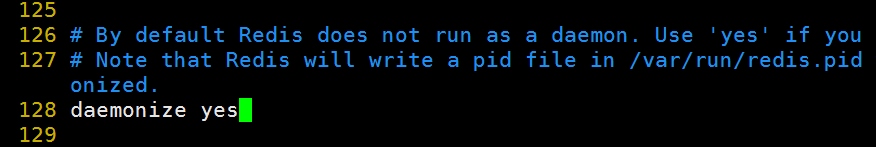
p105 当客户端空闲时间达到一小时,就会自动断开连接,0秒表示

不启用超时配置



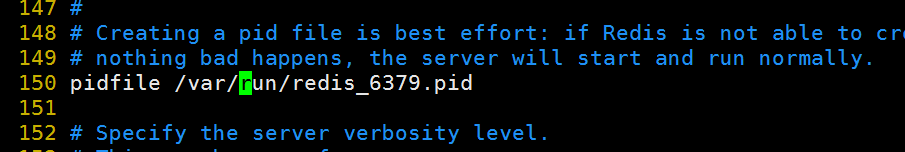
p128 daemonize 设置成yes让redis服务器启动有守护进程管理

(后台执行)

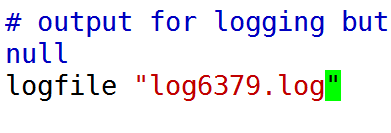


p150 对应不同的redis实例,pid的文件名称需要和端口同名

每个进程在linux或者其他操作系统中都会占用pid号，当系统中的进程过多时，需要查找redis进程号可能比较麻烦，直接打开pid文件查看即可



P163logfile 需要指定，利用端口号命名，放到redis根目录，日志输出的文件



202行

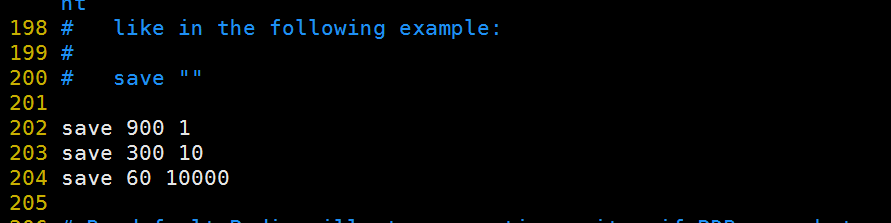
save 900(秒) 1(变动的数据条数)

当900以内,至少有1条数据变动,看是flush保存数据到文件

save 300 10

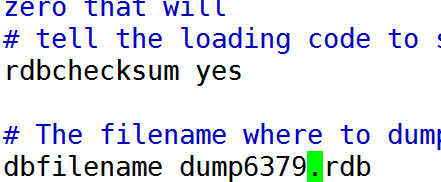
300秒以内至少10条数据变动,保存文件

save 60 10000



P237左右，指定dump的持久化文件，每个服务单独指向一个文件，

重启时，数据不会错乱



  保存退出

**启动三个redis实例**

拷贝模板文件为三分

[root@10-9-100-26 redis-3.2.11]# cp redis.conf redis01.conf

[root@10-9-100-26 redis-3.2.11]# cp redis.conf redis02.conf

[root@10-9-100-26 redis-3.2.11]# cp redis.conf redis03.conf

* 修改redis02,redis03的端口号(与端口号相关的配置一并修改) 进入vim编辑模式直接输入命令

:%s/6379/6380/g 意思是用6380替换所有叫6379

Redis03修改为6381

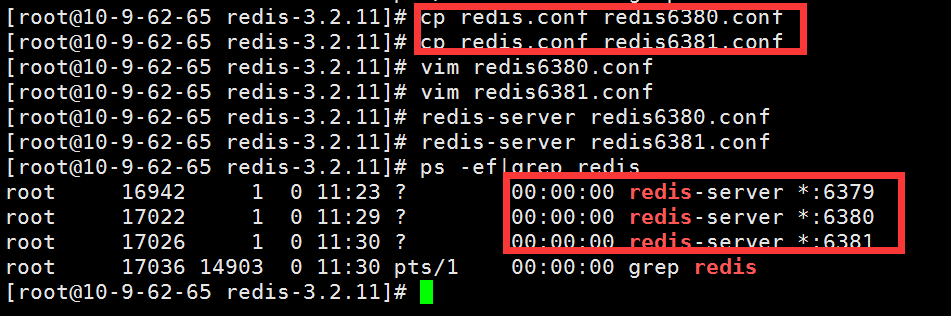
* 启动三个节点

#redis-server redis01.conf

#redis-server redis02.conf

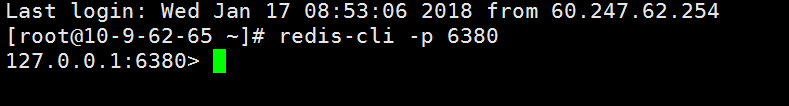
#redis-server redis03.conf

#ps -ef|grep redis

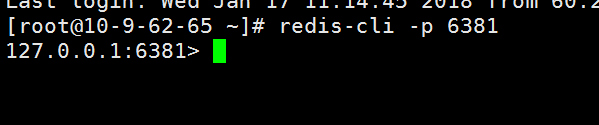


指定端口登录客户端redis-cli -p [端口号] -h [ip]

#redis-cli -p 6380



#redis-cli -p 6381



## 7.5.jedis客户端代码

### 7.5.1.依赖

1.quickstart（springboot-redis-test01）

2.pom文件引入依赖

springboot（继承easymall-parent）、redis依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-redis</artifactId>

<version>1.4.7.RELEASE</version>

</dependency>

### 7.5.2.jedis连接

|  |
| --- |
| **package** cn.tedu.jedis.test;  **import** org.junit.Test;  **import** redis.clients.jedis.Jedis;  /\*\*  \* jedis测试类  \*/  **public** **class** JedisTest {  /\*  \* java代码连接redis客户端  \*/  @Test  **public** **void** test01(){  //创建jedis对象,相当于在linux执行redis-cli -p 6379 -h 192.168.171.138  Jedis jedis = **new** Jedis("192.168.171.138",6379);  //调用api存、取等操作  //jedis.set("name", "xiaomin");  System.***out***.println(jedis.get("name"));  }  } |

### 7.5.3.模拟缓存逻辑

查询商品的缓存

service逻辑：

1.判断缓存是否存在

2. 如果存在：读取数据，封装返回

如果不存在：去数据库查询

存入缓存，并将数据封装返回

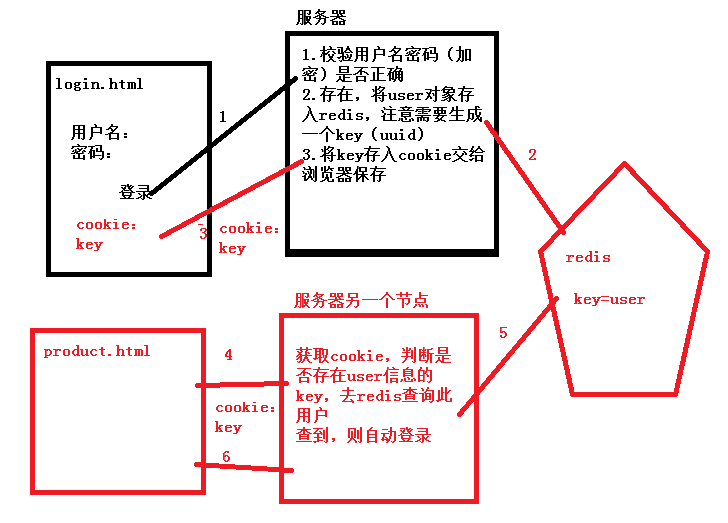
key的设计：通常有业务意义，可以利用数据本身的id

例如：key=”product\_”+productId

|  |
| --- |
| @Test  **public** **void** test02(){  //模拟ProductId  String priductId = "5783692";  System.***out***.println("用户发起请求访问商品：product\_"+priductId);  //去缓存查数据  //拿到缓存的key  String key = "product\_"+priductId;  //生成jedis客户端  Jedis jedis = **new** Jedis("192.168.171.138",6379);  //判断缓存中是否有数据  **if**(jedis.exists(key)){//存在  System.***out***.println("从缓存中获取数据："+jedis.get(key));  }**else**{//缓存无数据  //从数据库查  String productJson = "{'name':'phone','price':500}";  System.***out***.println("从数据库查到数据："+productJson);  //将数据存入缓存  jedis.set(key, productJson);  }  } |

作业：将缓存逻辑迁移到product商品中心的单个商品查询

## 7.6.easymall用户登录系统（User系统）



1.pom文件添加依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-redis</artifactId>

<version>1.4.7.RELEASE</version>

</dependency>

2.业务逻辑

|  |  |
| --- | --- |
| js请求地址 | http://www.easymall.com/user/login |
| 后台接收 | /user/manage/login |
| 请求方式 | Post |
| 请求参数 | User user 只有用户名和密码,查询需要加密password |
| 返回数据 | 返回SysResult对象的json,其结构:  Integer status; 200表示成功,其他表示失败  String msg;成功返回 “ok”,失败返回其他信息  Object data;根据需求携带其他数据 |
| 备注1 | 需要使用redis存储查询到的user数据,并且利用cookie传递回浏览器,以便下次访问携带cookie完成用户状态的请求 |
| 备注2 | Redis存储的用户数据,key=ticket生成公式:”EM\_TICKET”+currentTime+userId;value就是userJson |
| 备注3 | 可以利用这里的登录逻辑,控制一个用户最多的登录个数 |

UserController

|  |
| --- |
| @RequestMapping("/login")  **public** SysResult doLogin(User user,HttpServletRequest request,HttpServletResponse response){  //交给业务层登录  String ticket = userService.doLogin(user);  //判断key是否存在  **if**("".equals(ticket) || ticket == **null**){//不存在  //登录失败  **return** SysResult.*build*(201, "失败", **null**);  }**else**{  //将ticket交给cookie返回  CookieUtils.*setCookie*(request, response, "EM\_TICKET",ticket);  **return** SysResult.*ok*();  }  } |

UserService

|  |
| --- |
| **public** String doLogin(User user) {  //密码加密  user.setUserPassword(MD5Util.*md5*(user.getUserPassword()));  //数据库查询user对象  User getUser = userMapper.queryUserByUsernameAndPassword(user);  **if**(getUser == **null**){//用户名密码错误  **return** "";  }**else**{//用户名密码正确  //创建一个存入redis的key  String ticket = UUIDUtil.*getUUID*();  //将查到的user对象存入redis  //将user对象转换为json  ObjectMapper mapper = **new** ObjectMapper();  String userJson;  **try** {  userJson = mapper.writeValueAsString(getUser);  } **catch** (JsonProcessingException e) {  e.printStackTrace();  **return** "";  }  //创建jedis对象  Jedis jedis = **new** Jedis("192.168.171.138",6379);  jedis.set(ticket, userJson);  **return** ticket;  }  } |

UserMapper

|  |
| --- |
| User queryUserByUsernameAndPassword(User user); |

UserMapper.xml

|  |
| --- |
| <select id=*"queryUserByUsernameAndPassword"*  parameterType=*"User"* resultType=*"User"*>  select \* from t\_user where user\_name=#{userName}  and user\_password=#{userPassword}  </select> |

zuul的cookie拦截

zuul网关不能接受cookie，默认对cookie的头敏感，所以会将cookie过滤掉，只需要在zuul的配置文件中添加敏感头为空的配置（没有value）

zuul.sensitive-headers=

测试

**登录功能-状态显示**

|  |  |
| --- | --- |
| js请求地址 | http://www.easymall.com/user/query/{ticket} |
| 后台接收 | /user/manage/query/{ticket} |
| 请求方式 | Get |
| 请求参数 | String ticket 就是用户登录时生成的rediskey值 |
| 返回数据 | 返回SysResult对象的json,其结构:  Integer status; 200表示成功,其他表示失败  String msg;成功返回 “ok”,失败返回其他信息  Object data;封装从redis获取的userJson |
| 备注 | 可以在这个逻辑中完成续租用户登录状态的逻辑的续租 |

UserController

|  |
| --- |
| @RequestMapping("/query/{ticket}")  **public** SysResult checkLogin(@PathVariable String ticket){  //交给service根据key（ticket）查询redis中是否存在userJson  String userJson = userService.checkLogin(ticket);  //判断非空  **if**(StringUtils.*isNotEmpty*(userJson)){  **return** SysResult.*build*(200, "ok", userJson);  }**else**{  **return** SysResult.*build*(201, "", **null**);  }  } |

UserService

|  |
| --- |
| **public** String checkLogin(String ticket) {  Jedis jedis = **new** Jedis("192.168.171.138",6379);  String userJson = jedis.get(ticket);  **return** userJson;  } |

## 7.7.分片自定义算法

单个redis内存容量有上限，很容易达到瓶颈，即使配置超时逻辑

那么，需要利用redis分布式集群

问题：如何划分数据

可以引入hash取余

### 7.7.1.hash取余算法

1.取hashcode

int hashcode = key.hashCode();这里的返回值可正可负

只要key不变，hash值一定不变

2.取正

由于取余后，只能是正整数，所以需要对hash值取正

方式一：int absHash = Math.abs(hashcode);

方式二：**int** result = hashcode&Integer.***MAX\_VALUE***;

经过方式二（效率高），任意数都会变为正

3.取余

**int** num = result%3;

得到一个0、1、2的任意数，此余数和redis的编号匹配

### 7.7.2.利用hash算法模拟海量数据

|  |
| --- |
| @Test  **public** **void** test04(){  Jedis jedis1 = **new** Jedis("192.168.171.138",6379);  Jedis jedis2 = **new** Jedis("192.168.171.138",6380);  Jedis jedis3 = **new** Jedis("192.168.171.138",6381);  //模拟海量数据  **for** (**int** i = 0; i < 10000; i++) {  //对keyhash取余  String key = UUID.*randomUUID*().toString();  **int** hashcode = key.hashCode()&Integer.***MAX\_VALUE***;  **int** num = hashcode % 3;  //根据取余结果，放入不同的redis  **if**(num == 0){  jedis1.set(key, "value"+i);  }  **if**(num == 1){  jedis2.set(key, "value"+i);  }  **if**(num == 2){  jedis3.set(key, "value"+i);  }  }  } |

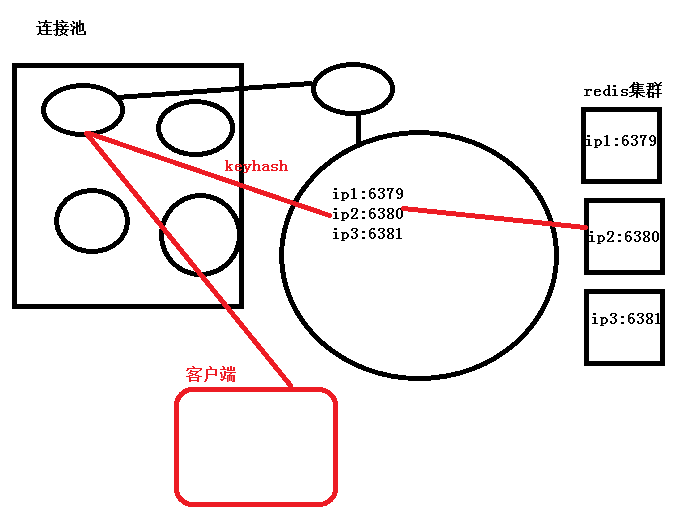
### 7.7.3.jedis自带分片对象

jedis也可以实现底层的算法（hash一致性），利用一个对象收集多个节点信息，通过底层hash一致性算法封装，可以像jedis对象调用方法意义，但底层已经做了hash一致性计算

|  |
| --- |
| @Test  **public** **void** test05(){  //利用List收集所有节点信息  List<JedisShardInfo> infoList = **new** ArrayList<JedisShardInfo>();  infoList.add(**new** JedisShardInfo("192.168.171.138",6379));  infoList.add(**new** JedisShardInfo("192.168.171.138",6380));  infoList.add(**new** JedisShardInfo("192.168.171.138",6381));  //创建shardedJedis对象  ShardedJedis sJedis = **new** ShardedJedis(infoList);  //sJedis.set("name", "zhangsan");  //sJedis.set("age", "18");  //sJedis.set("gender", "m");  System.***err***.println(sJedis.get("name"));  } |

### 7.7.4.分片连接池

和jdbc同理，jedis连接对象交给连接池维护



|  |
| --- |
| @Test  **public** **void** test06(){  //利用List收集所有节点信息  List<JedisShardInfo> infoList = **new** ArrayList<JedisShardInfo>();  infoList.add(**new** JedisShardInfo("192.168.171.138",6379));  infoList.add(**new** JedisShardInfo("192.168.171.138",6380));  infoList.add(**new** JedisShardInfo("192.168.171.138",6381));  //创建配置对象，专门用来存储配置信息  GenericObjectPoolConfig config = **new** GenericObjectPoolConfig();  config.setMaxTotal(10);  config.setMaxIdle(5);  config.setMinIdle(3);  //创建连接池，将配置信息和节点信息交给连接池  ShardedJedisPool pool = **new** ShardedJedisPool(config,infoList);  //从连接池获取连接  ShardedJedis sJedis = pool.getResource();  //sJedis.set("location", "猪窝");  System.***out***.println(sJedis.get("location"));  //还连接  pool.~~returnResource~~(sJedis);  } |

## 7.8.连接池整合easymall

有了连接池，可以高效使用多个redis节点的集群完成redis功能，可以将连接池交给springboot框架维护，哪里使用，哪里注入

### 7.8.1.操作步骤

1.pom依赖（已经依赖）

2.配置文件

spring.redis.shardedpool.nodes=192.168.171.138:6379,192.168.171.138:6380,192.168.171.138:6381

spring.redis.shardedpool.maxTotal=100

spring.redis.shardedpool.maxIdle=10

spring.redis.shardedpool.minIdel=5

3.配置类读取配置文件并初始化连接池对象

|  |
| --- |
| **package** cn.tedu.user.config;  **import** java.util.ArrayList;  **import** java.util.List;  **import** org.apache.commons.pool2.impl.GenericObjectPoolConfig;  **import** org.springframework.boot.context.properties.ConfigurationProperties;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** redis.clients.jedis.JedisShardInfo;  **import** redis.clients.jedis.ShardedJedisPool;  @Configuration  @ConfigurationProperties(prefix="spring.redis.shardedpool")  **public** **class** ShardedJedisPoolConfig {  //只要配置文件的value以","\";"间隔，可以自动封装到list  **private** List<String> nodes;  **private** Integer maxTotal;  **private** Integer maxIdle;  **private** Integer minIdle;  @Bean  **public** ShardedJedisPool initShardedJedisPool(){  //1.所有节点信息的list  List<JedisShardInfo> list = **new** ArrayList<JedisShardInfo>();  **for** (String node : nodes) {  //192.168.171.138:6379  list.add(**new** JedisShardInfo(  node.split(":")[0],node.split(":")[1]));  }  //2.保存配置信息的config对象  GenericObjectPoolConfig config = **new** GenericObjectPoolConfig();  config.setMaxTotal(maxTotal);  config.setMaxIdle(maxIdle);  config.setMinIdle(minIdle);  //3.创建连接池，将上述两个对象传进去，并返回  **return** **new** ShardedJedisPool(config,list);  }  get&set  } |

### 7.8.2.修改登录系统（第二版）

UserController（无需改动）

UserService

|  |
| --- |
| @Autowired  **private** ShardedJedisPool pool;  **public** String doLogin(User user) {  ShardedJedis jedis = pool.getResource();  **try** {  //密码加密  user.setUserPassword(MD5Util.*md5*(user.getUserPassword()));  //数据库查询user对象  User getUser = userMapper.queryUserByUsernameAndPassword(user);  **if**(getUser == **null**){//用户名密码错误  **return** "";  }**else**{//用户名密码正确  //创建一个存入redis的key  String ticket = UUIDUtil.*getUUID*();  //将查到的user对象存入redis  //将user对象转换为json  ObjectMapper mapper = **new** ObjectMapper();  String userJson;    userJson = mapper.writeValueAsString(getUser);    //创建jedis对象  //Jedis jedis = new Jedis("192.168.171.138",6379);  jedis.set(ticket, userJson);  **return** ticket;  }  } **catch** (JsonProcessingException e) {  e.printStackTrace();  **return** "";  }**finally**{  pool.~~returnResource~~(jedis);  }    }  **public** String checkLogin(String ticket) {  ShardedJedis jedis = pool.getResource();  **try** {  String userJson = jedis.get(ticket);  **return** userJson;  } **catch** (Exception e) {  e.printStackTrace();  **return** "";  }**finally**{  pool.~~returnResource~~(jedis);  }  } |

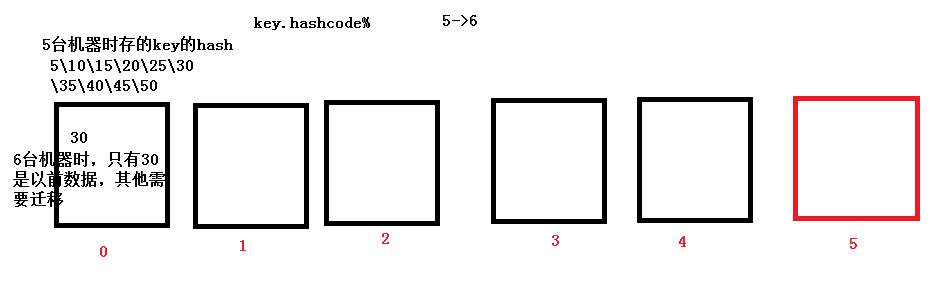
## 7.9.hash一致性

### 7.9.1.hash取余算法问题

1.数据倾斜问题

2.扩容缩容问题（节点变化导致取余因数变化，数据取不到而需要迁移数据）

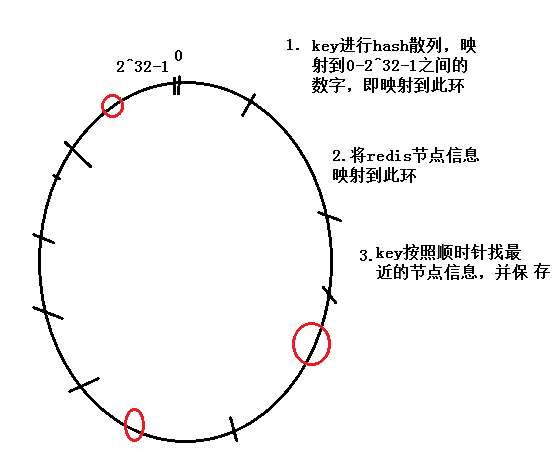
如果集群节点较大，可能接近100%的数据需要迁移

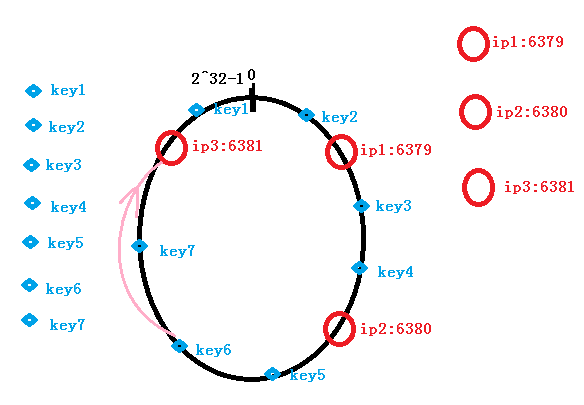


### 7.9.2.hash一致性介绍

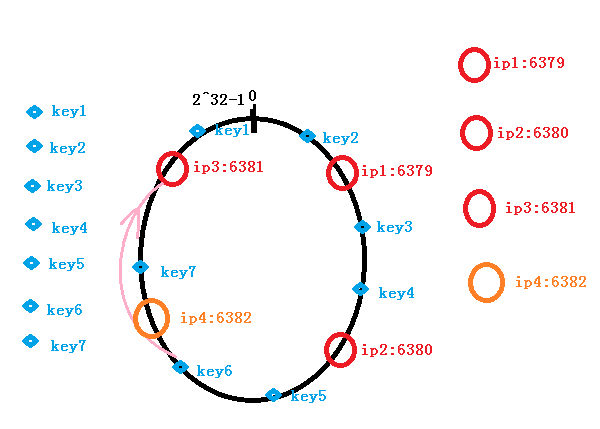
hash一致性是一种数学模型，1997年美国麻省理工学院，大二学生研究的数学模型为基石-hash散列

### 7.9.3.hash一致性逻辑



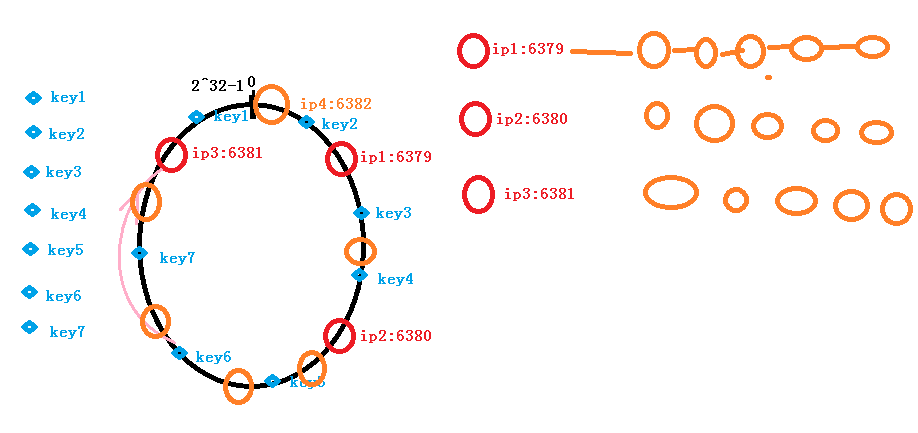


### 7.9.4.hash一致性解决扩、缩容问题



当新来一个节点，也映射到hash环，只有这个节点之前的内容需要迁移到此节点，其他无需改动

### 7.9.5.解决数据倾斜问题



引入虚拟节点的概念，每个真实节点可以有若干个虚拟节点，这些虚拟节点也映射到hash环，这样，hash环上的节点数可以变得很大，节点越多，分配越平均

当数据顺时针找到任意节点（虚拟节点），就可以获取对应的真实节点的信息，就可以将数据存入对应的真实节点，极大的解决数据倾斜问题（数据分配不均匀）

jedis中，每个节点的虚拟节点个数为160\*weight

虚拟节点+weight测试

### 7.9.6.hash一致性总结

1.基于hash散列数学模型算法

2.hash散列结果区间为2^32-1

3.所有key散列映射到环的某个整数

4.每个节点（包括虚拟节点）也都映射到hash环

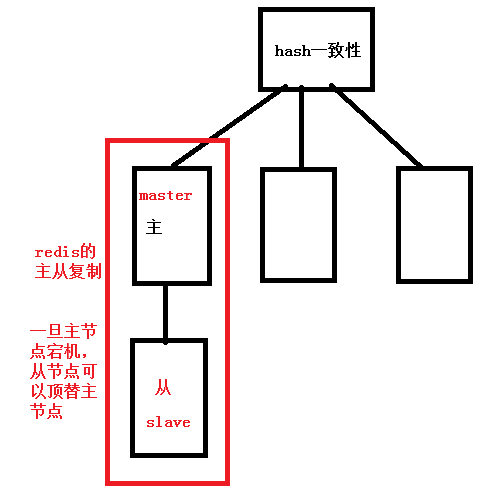
5.key顺时针寻找最近的节点

6.将数据保存到此节点中（如果是虚拟节点，找到对应的真实节点）

7.概念：jedis中每个节点的虚拟节点默认160，可以分配权重：160\*weight

## 7.10.redis高可用

### 7.10.1.高可用结构



### 7.10.2.redis主从复制

redis支持数据的主从复制，支持多级主从，也支持一主多从，当结构过于复杂时，数据同步效率会降低，企业经验，最多两级主从，最多1主6从。

**（1）配置搭建一个redis的一主二从关系**

关闭redis：通过进程号kill -9 进程号

或者在客户端调用shutdown命令

1.准备3个节点6382,6383,6384

[root@10-9-100-26 redis-3.2.11]# cp redis.conf redis6382.conf

[root@10-9-100-26 redis-3.2.11]# cp redis.conf redis6383.conf

[root@10-9-100-26 redis-3.2.11]# cp redis.conf redis6384.conf

2.挨个修改端口,启动3个节点

:%s/6379/替换的端口/g

[root@10-9-100-26 redis-3.2.11]# redis-server redis6382.conf

[root@10-9-100-26 redis-3.2.11]# redis-server redis6383.conf

[root@10-9-100-26 redis-3.2.11]# redis-server redis6384.conf

3.主从关系的创建

6382主节点,6383,6384两个从节点

观察节点状态的命令

* 6382>info replication

role:master

connected\_slaves:0

master\_repl\_offset:0

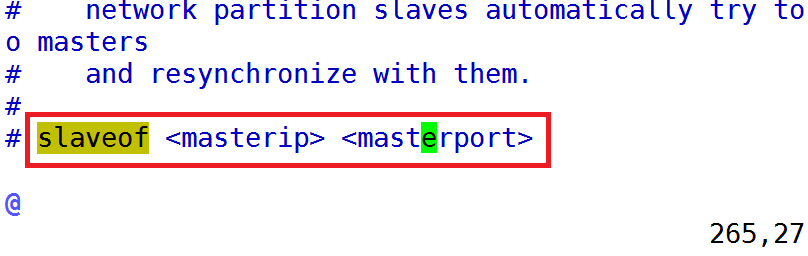
repl\_backlog\_active:0

repl\_backlog\_size:1048576

repl\_backlog\_first\_byte\_offset:0

repl\_backlog\_histlen:0

* 通过配置文件做定义主从



* 通过命令挂接主节点

从节点>slaveof masterip masterport

127.0.0.1:6383> slaveof 10.9.100.26 6382

127.0.0.1:6384> slaveof 10.9.100.26 6382

4.查看挂接状态

主节点显示从节点信息

从节点显示主节点.主从关系各种数据状态

**（2）主从关系测试**

1.数据同步

主节点6382写数据,set

6382操作

127.0.0.1:6382> set name haha

OK

127.0.0.1:6382> get name

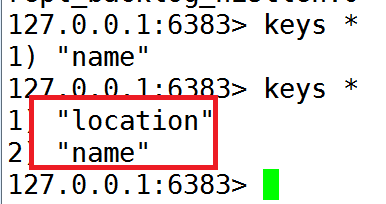
"zhangsan"

127.0.0.1:6382> set location beijing

OK

127.0.0.1:6382>

6383/6384观察数据



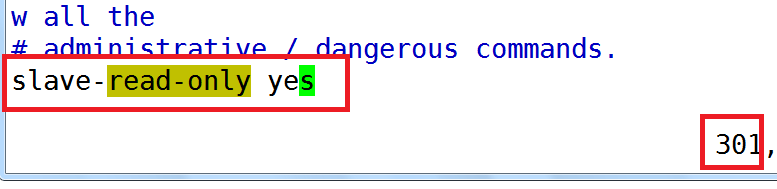
2.从节点写入数据

127.0.0.1:6384> set age 18

(error) READONLY You can't write against a read only slave.

127.0.0.1:6384>

默认状态(配置文件默认值),从节点只读,如果需要在从节点写数据(多级主从关系);打开配置文件的readonly的配置



3.高可用替换

将6382干掉.观察6383和6384能否有一个自动将slave角色替换成master.

127.0.0.1:6382> shutdown

not connected> quit

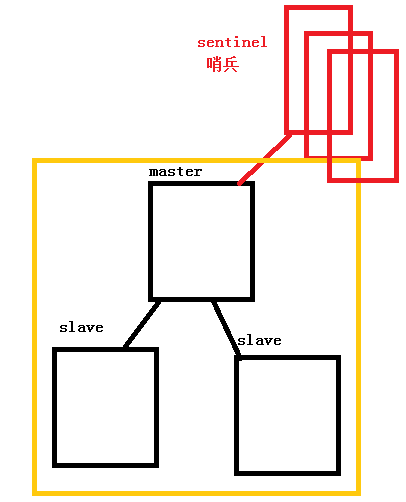
在单纯的主从结构下.主节点宕机,从节点什么都不做，不会替换主节点;

## 7.11.哨兵集群模式

### 7.11.1.哨兵集群介绍

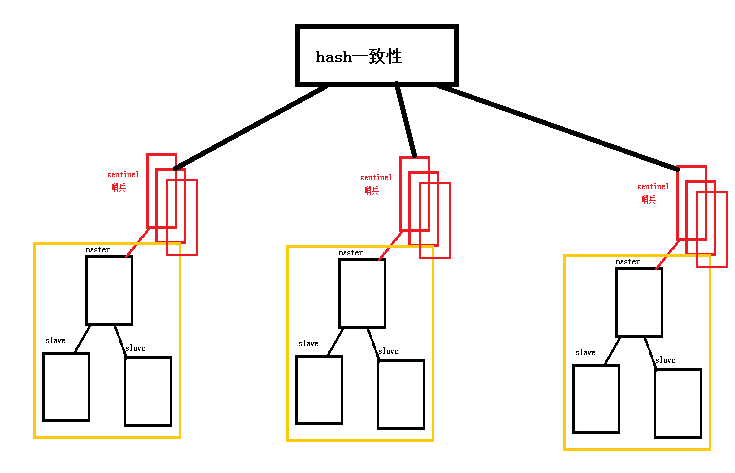
redis-cluster（redis3.0）出现之前，绝大部分redis使用的是哨兵集群，利用哨兵（redis进程）对主从结构监听，监听主，从主节点获取（info replication）从节点信息，保存在内存中，后续每一秒都会发起一次心跳检测（rpc远程协议），一旦发现主节点宕机，哨兵集群就会发起投票机制，通过过半选举出新的master

**哨兵集群结构**



一个哨兵集群只能维护一个数据分片的高可用，不能完成分布式

分布式结构：



### 7.11.2.哨兵投票机制

为什么要投票，为什么要过半

哨兵监听的主从关系，角色是变化的，每次变化，都需要哨兵监控，对所有的决策都需要投票

哨兵连接主从结构需要网络（网络通信可能有波动），如果只有一个哨兵投票，则结论不可信，所以引入过半机制，一半以上节点必须投票一致

**集群容忍度：**

哨兵集群总，可以允许宕机的数量

5个哨兵，至少3个节点投票，至少活三个，集群容忍度为2

6个哨兵，至少4个节点投票，至少活四个，集群容忍度为2

由于2n和2n-1的集群容忍度相同，一半集群数量为奇数。

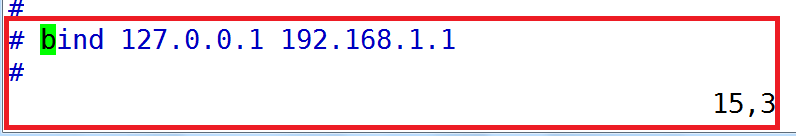
### 7.11.3.搭建哨兵集群

3个哨兵节点个数,管理6382/6383/6384的主从结构

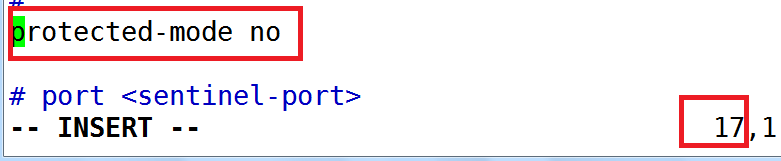
1.修改哨兵的模板配置文件(在根目录sentinel.conf)

配置三个哨兵的端口26379 26380 26381

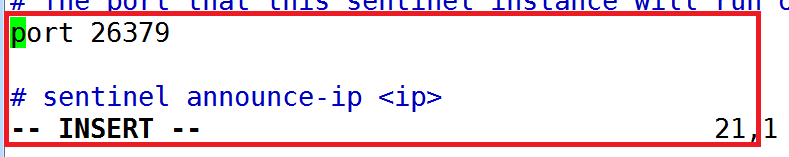
P15 bind的ip地址绑定--保持注释



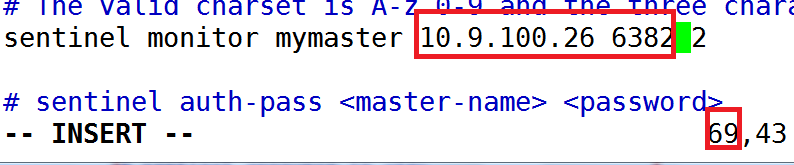
P17 释放注释掉的protected-mode



P21 保持模板文件中26379默认端口号



P69 监听主节点的配置核心内容



sentinel monitor 配置命令起始,表示开始进行对\*\*的监听工作

mymaster 自定义的名称,表示当前哨兵监听的一个主从的结构代号,也是其他哨兵判断是否是同一个集群的依据

host 主节点的ip地址

port 主节点的端口号

2 主观下限票数,当集群中某些哨兵节点宕机有关,当集群哨兵不断宕机之后,最少剩下的节点数量

2.拷贝三分,修改各自的端口号为26379 26380 26381

[root@10-9-100-26 redis-3.2.11]# cp sentinel.conf sentinel01.conf

[root@10-9-100-26 redis-3.2.11]# cp sentinel.conf sentinel02.conf

[root@10-9-100-26 redis-3.2.11]# cp sentinel.conf sentinel03.conf

3.启动三个哨兵集群(注意前提是配置的主从正常运转，即已经启动)

[root@10-9-100-26 redis-3.2.11]# redis-sentinel sentinel01.conf

[root@10-9-100-26 redis-3.2.11]# redis-sentinel sentinel02.conf

[root@10-9-100-26 redis-3.2.11]# redis-sentinel sentinel03.conf

### 7.11.4.对哨兵集群高可用测试

1. 6382kill掉,观察日志内容(哨兵日志)

哨兵发现宕机.开始投票,通过,确定宕机.

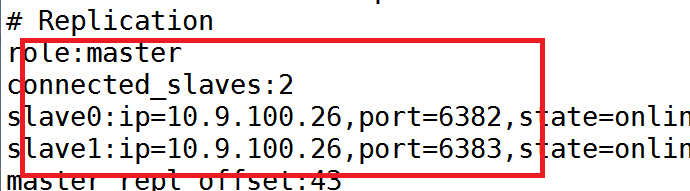
开发投票选举新的主节点;

主节点一旦恢复,重新添加到集群以从节点角色继续提供服务.

2.重启哨兵集群

确定主从结构

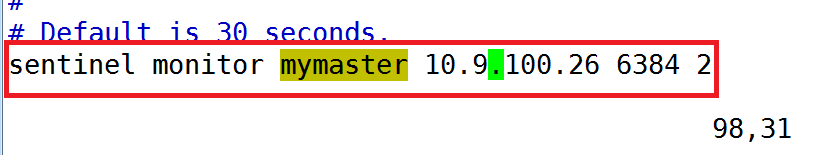
主从节点启动了起来,观察结构是否正确;都要看到正确的info replication的信息



哨兵进程

初始化配置文件

* 确定sentinel monitor mymaster 主节点ip 主节点port 2



* 尾部的日志给删除
* 重新启动sentinel

### 7.11.5.jedis连接操作sentinel集群的api

|  |
| --- |
| //连接哨兵集群操作主从,高可用结构  @Test  **public** **void** test08(){  //收集哨兵的集群节点信息  Set<String> sentinelSet=**new** HashSet<String>();  sentinelSet.add(**new** HostAndPort("192.168.171.138", 26379).toString());  sentinelSet.add(**new** HostAndPort("192.168.171.138", 26380).toString());  sentinelSet.add(**new** HostAndPort("192.168.171.138", 26381).toString());  //配置连接的配置对象  GenericObjectPoolConfig config=**new** GenericObjectPoolConfig();  config.setMaxTotal(200);  //构建哨兵的管理对象  JedisSentinelPool pool=**new** JedisSentinelPool  ("mymaster", sentinelSet, config);  //获取一个当前主从结构的主节点资源  Jedis jedis = pool.getResource();  System.***out***.println("当前正在服役的主节点:"+pool.getCurrentHostMaster());  //jedis操作主节点  jedis.set("name", "lisi");  //String name = jedis.get("name");  //System.out.println("name:"+name);  } |

### 7.11.6.哨兵整合到springboot

1.pom文件引入依赖（如果需要）

2.配置文件（哨兵节点信息、哨兵管理的redis高可用集群名称）

3.配置类读配置信息，初始化连接池（将配置信息设置到连接池），即交给框架维护

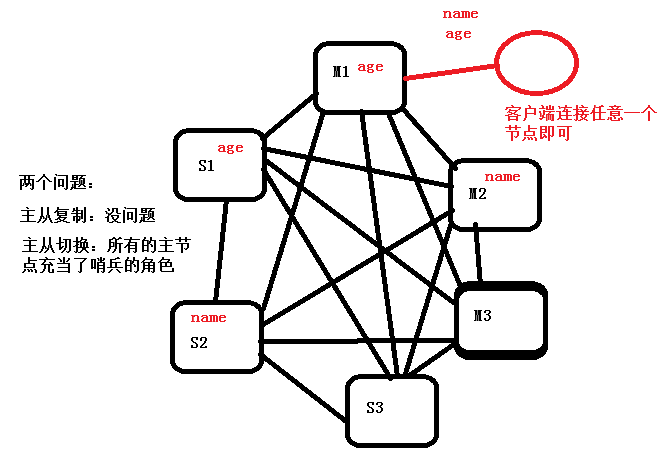
4.哪里使用连接池，哪里注入

## 7.12.redis-cluster

### 7.12.1.redis-cluster介绍

redis3.0出现的一个新的redis集群结构，引入了很多新的概念和逻辑，将之前学的主从复制，哨兵监听整合到了这个集群

**redis-cluster结构**



### 7.12.2.集群结构特点

1.集群两两互联

2.集群监听机制：主节点充当哨兵角色，一旦有主节点宕机，其他主节点进行投票选举

3.客户端可以连接任意一个节点（主、从都可以），客户端向集群传输数据时，连接的节点会判断数据存储的位置，然后将数据传输到真正存数据的节点

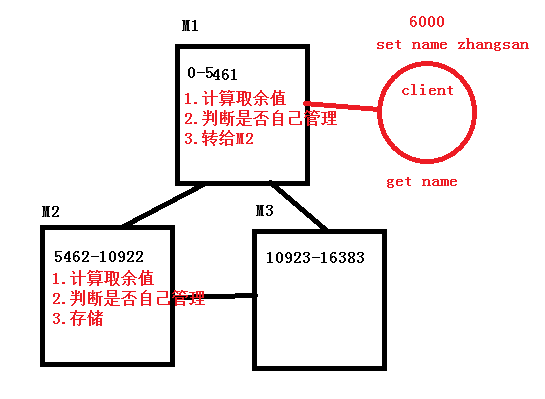
4.引入了新的分布式计算逻辑--hash取模（CRC16()，即hashcode%16384）散列计算规则（取余结果0-16383），这个区间称为槽道号，每个master负责管理一批槽道号，根据key的取余（取模）结果判断key归哪个节点管理，从而做到解耦

### 7.12.3.槽道号逻辑

redis-cluster要求集群中至少有3个master主节点（因为最少容忍度是1）

槽道号（0-16383）：理论上redis主节点最多可以有16384个，但是官方最多1000左右

每个主节点都可以具备槽道号的管理权，但是也可以不管理槽道号，这样只具备投票权



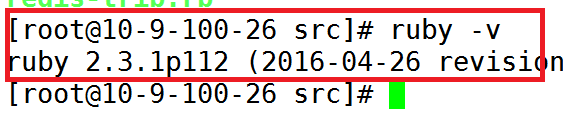
问题一：如何判断是否自己管理

问题二：如果不是自己管理，如何判断交给谁

## 7.13.redis-cluster安装（ruby安装）

### 7.13.1.安装ruby环境

* ruby语言已经安装完毕

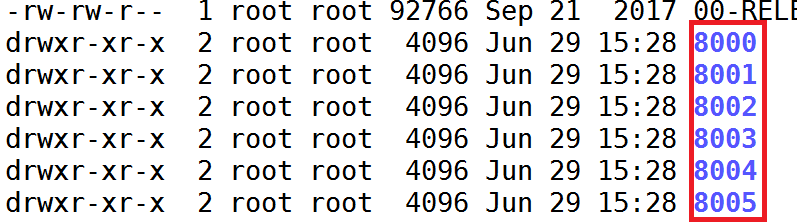


### 7.13.2.准备一下搭建集群的配置文件环境

1.每一个配置文件单独由一个端口号命名的文件夹管理 8000/redis.conf 8001/redis.conf...

2.在redis根目录创建出搭建集群需要的6个节点对应的文件夹8000-8005

[root@10-9-100-26 redis-3.2.11]# mkdir 8000 8001 8002 8003 8004 8005



3.修改一个redis-cluster节点启动的模板文件

[root@10-9-100-26 redis-3.2.11]# cp redis.conf redis-cluster.conf

4.打开模板文件修改内容

bind,protected-mode daimonize dump log 等不变(6379)

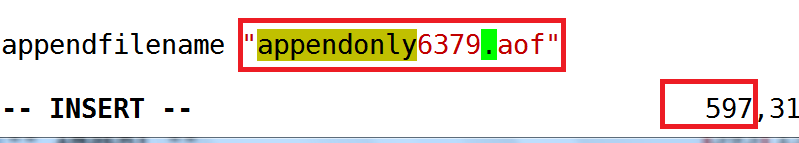
继续配置与集群节点启动有关的内容.

P593行追加模式 aof的持久化策略

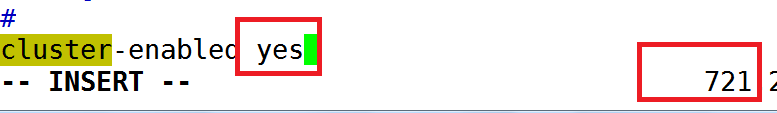


开启aof模式的持久化策略

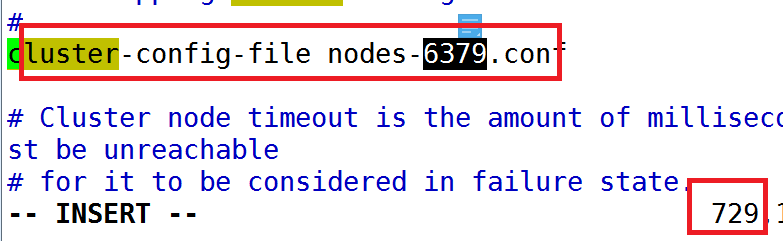
P597 指定一个端口号对应的持久化文件



P721 必须将节点启动定义为集群模式



P729 打开指定一个集群节点运行状态的记录文件conf



5.将模板文件拷贝到8000-8005的文件夹中,挨个修改端口号

[root@10-9-100-26 redis-3.2.11]# cp redis-cluster.conf 8000/

[root@10-9-100-26 redis-3.2.11]# cp redis-cluster.conf 8001/

[root@10-9-100-26 redis-3.2.11]# cp redis-cluster.conf 8002/

[root@10-9-100-26 redis-3.2.11]# cp redis-cluster.conf 8003/

[root@10-9-100-26 redis-3.2.11]# cp redis-cluster.conf 8004/

[root@10-9-100-26 redis-3.2.11]# cp redis-cluster.conf 8005/

修改端口

:%s/6379/800\*/g

6.启动所有节点

[root@10-9-100-26 redis-3.2.11]# redis-server 8000/redis-cluster.conf

[root@10-9-100-26 redis-3.2.11]# redis-server 8001/redis-cluster.conf

[root@10-9-100-26 redis-3.2.11]# redis-server 8002/redis-cluster.conf

[root@10-9-100-26 redis-3.2.11]# redis-server 8003/redis-cluster.conf

[root@10-9-100-26 redis-3.2.11]# redis-server 8004/redis-cluster.conf

[root@10-9-100-26 redis-3.2.11]# redis-server 8005/redis-cluster.conf

查看启动进程redis

[root@10-9-100-26 redis-3.2.11]# ps -ef|grep redis

root 2118 1 0 16:10 ? 00:00:00 redis-server \*:8000 [cluster]

root 2122 1 0 16:10 ? 00:00:00 redis-server \*:8001 [cluster]

root 2126 1 0 16:10 ? 00:00:00 redis-server \*:8002 [cluster]

root 2130 1 0 16:10 ? 00:00:00 redis-server \*:8003 [cluster]

root 2134 1 0 16:10 ? 00:00:00 redis-server \*:8004 [cluster]

root 2139 1 0 16:10 ? 00:00:00 redis-server \*:8005 [cluster]

7.以集群模式启动的节点在搭建分配槽道之前(16384全部分配),无法直接操作

127.0.0.1:8000> set name zhangsan

(error) CLUSTERDOWN Hash slot not served

### 7.13.3.两种redis的持久化策略对比

|  |  |
| --- | --- |
| rdb | aof |
| 使用dump.rdb为默认的持久化文件,内存储的是数据内容key-value | 一种追加模式的日志文件，存储的是命令的日志 |
| 可以一次性存储多条数据(执行效率高) | 默认每秒同步执行的写命令(执行效率低) |
| save时间间隔长,对数据的备份能力要求高,不推荐使用rdb | 备份数据时间比较短,最多最多,在极大的高并发先只会丢失1-2秒的数据 |
| 一次性存储数据过大时,有可能造成阻塞,影响使用 | 细水长流 |
| 数据量并发少的时候rdb性能更高 | 数据量并发到达一定程度aof返回性能高于rdb |

如果同时开启aof和rdb,aof优先级高于rdb;重启之后最先加载的appendOnly8000.aof;

### 7.13.4.利用ruby命令创建一个集群

最终可以使用ruby脚本命令创建一个3主各自有一从的集群结构

1.redis-trib.rb是使用ruby语言编写的具备很多操作集群底层命令的代码,利用它来实现创建集群的过程

[root@10-9-100-26 src]# redis-trib.rb

Usage: redis-trib <command> <options> <arguments ...>

create host1:port1 ... hostN:portN

--replicas <arg>

check host:port

info host:port

fix host:port

--timeout <arg>

reshard host:port

--from <arg>

--to <arg>

--slots <arg>

--yes

--timeout <arg>

--pipeline <arg>

rebalance host:port

--weight <arg>

--auto-weights

--use-empty-masters

--timeout <arg>

--simulate

--pipeline <arg>

--threshold <arg>

add-node new\_host:new\_port existing\_host:existing\_port

--slave

--master-id <arg>

del-node host:port node\_id

set-timeout host:port milliseconds

call host:port command arg arg .. arg

import host:port

--from <arg>

--copy

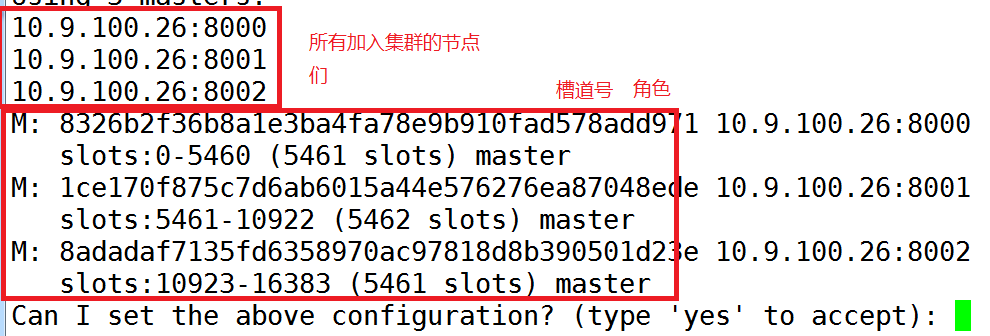
--replace

help (show this help)

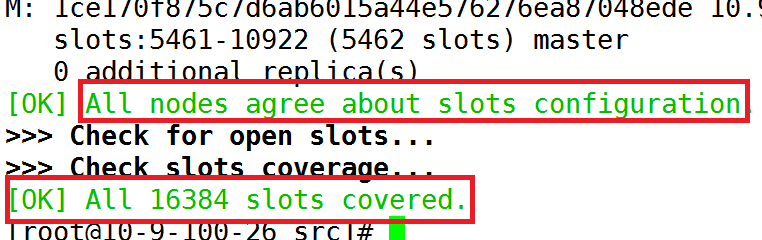
For check, fix, reshard, del-node, set-timeout you can specify the host and port of any working node in the cluster.

2.ruby脚本搭建一个3主的结构

[root@10-9-100-26 src]# redis-trib.rb create 10.9.100.26:8000 10.9.100.26:8001 10.9.100.26:8002



输入yes搭建集群(3个主节点)完毕,显示如下内容



3.初次体验一下槽道的逻辑

操作集群节点,需要使用集群的登录模式

#redis-cli -c -p 8000,-c:以集群模式登录节点

登录8000,进行命令的输入

#cluster info



#cluster nodes(显示的内容会保存在节点的node-800\*.conf文件中)

8adadaf7135fd6358970ac97818d8b390501d23e 10.9.100.26:8002 master - 0 1561796894215 3 connected 10923-16383

1ce170f875c7d6ab6015a44e576276ea87048ede 10.9.100.26:8001 master - 0 1561796895218 2 connected 5461-10922

8326b2f36b8a1e3ba4fa78e9b910fad578add971 10.9.100.26:8000 myself,master - 0 0 1 connected 0-5460

节点信息包含一下内容

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 8adadaf7135fd6358970ac97818d8b390501d23e | 10.9.100.26:8002 | master | - | 0 1561796894215 | 3 | connected | 10923-16383 |
| 节点id | 节点host地址:节点端口 | 角色 | 节点对应的主节点id | 操作时间相关 | 序号 | 连接状态 | 管理槽道号 |

myself 当前查看信息的客户端信息

4.将name=wang的key-value值set到8000的节点上

127.0.0.1:8000> set name wang

-> Redirected to slot [5798] located at 10.9.100.26:8001

OK

10.9.100.26:8001>

### 7.13.5.添加节点

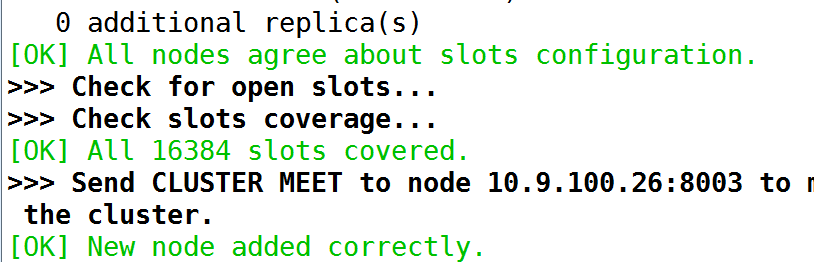
1.动态添加主节点

已经搭建的3个主节点的集群,8000 8001 8002 正常运行,可以直接新增一个节点为主节点;

调用

add-node 新节点ip:端口 已存在集群的任意节点ip:端口

[root@10-9-100-26 src]# redis-trib.rb add-node 10.9.100.26:8003 10.9.100.26:8000



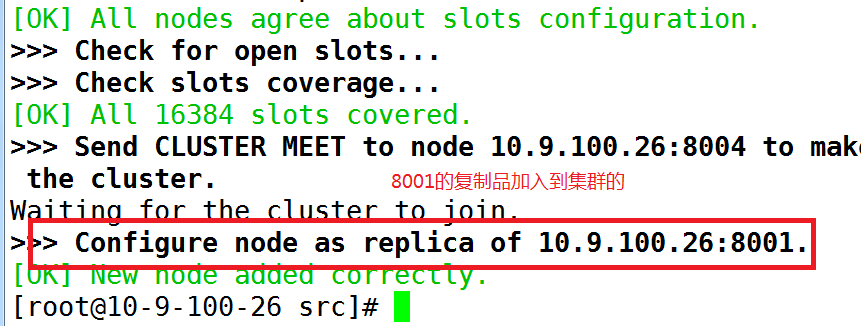
但是8003并没有获得任何槽道管理权

2.动态添加从节点

add-node --slave(以从节点角色加入集群) --master-id 参数(主节点的id) 新节点ip:端口 已存在节点ip:端口

8004挂接到8001;

[root@10-9-100-26 src]# redis-trib.rb add-node --slave --master-id 1ce170f875c7d6ab6015a44e576276ea87048ede 10.9.100.26:8004 10.9.100.26:8003



登录8004验证主从复制结构是否正确

正确挂接在8001节点下

3.直接验证集群结构的高可用

将主节点8001干掉shutdown,观察8004的顶替情况

顶替后是否能获取name 和location

顶替后是否正确继承槽道管理权5461-10922

8004正确顶替,正确运行功能

4.8001重启观察角色(slave)

重新挂接到8004作为从节点继续提供服务

### 7.13.6.将已经分配好的槽道交给8003管理

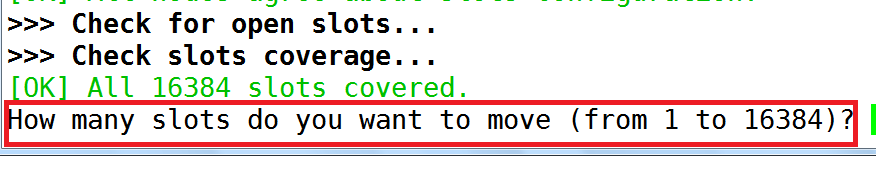
做槽道迁移;200个槽道交给8003

reshard 已存在集群的节点ip:端口

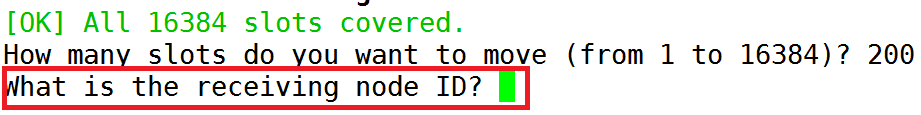
根据问题,作答.

[root@10-9-100-26 src]# redis-trib.rb reshard 10.9.100.26:8004

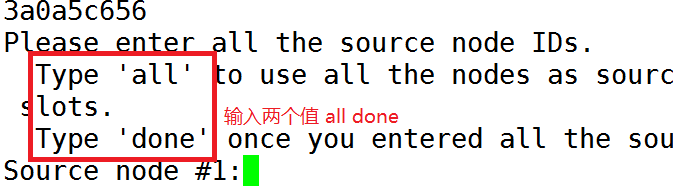
回答问题1:迁移的槽道数量(没法迁移非空槽道 5798 7307)



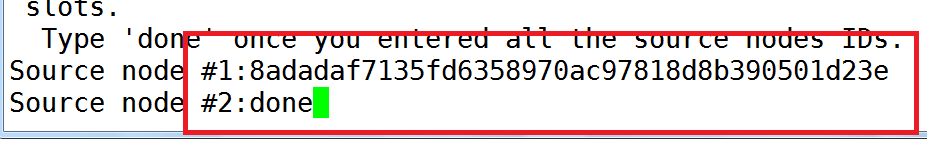
回答问题2:接收迁移槽道的节点id(8003的id)



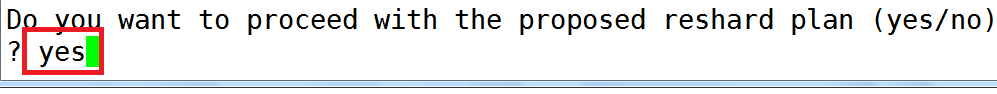
回答问题3:选择槽道的来源(所有槽道均分all,指定节点提供槽道输入节点id,完毕后输入done)



这里使用8002做槽道来源(200都从8002迁移)



回答问题4:是否同意迁移的设置



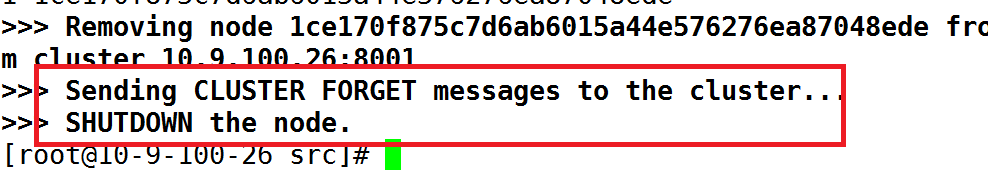
reshard不支持非空迁移(有数据的槽道),可以使用cluster集群提供的底层命令一步一步完成有数据的迁移(数据微调)

### 7.13.7.删除节点

只能删除从节点或者没有槽道管理权的主节点,一旦删除,为了不影响集群使用,删除的节点自动关闭;删除的节点重新启动重新添加到集群(做不到);

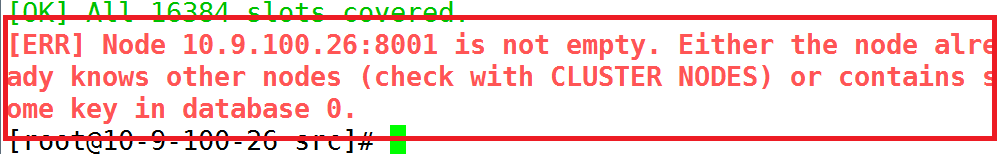
del-node 任意集群节点ip:端口 删除节点的id值

[root@10-9-100-26 src]# redis-trib.rb del-node 10.9.100.26:8001 1ce170f875c7d6ab6015a44e576276ea87048ede



8001被关闭的原因,是因为删除的节点不会修改配置文件node-8001.conf,如果将节点启动,自己以为自己还是集群中的节点，但是其他节点并不知道这个节点。

在添加节点是碰到的问题:



提示的问题:当前节点无法加入集群,不是节点有数据.有是节点已经被集群加入了.

删除8001,想以新节点角色加入;

加入之前,删除node文件.删除dump文件，aof也删除，重新启动

### 7.13.8.重新搭建集群(测试环境)

停/删/启/建

* 将所有节点kill/shutdown
* 删除所有持久化文件
  + node\*
  + dump\*
  + append\*
* 重新启动各自节点
* redis-trib.rb create 重新搭建 --replicas 1

以主从结构的集群来搭建cluster,1每个主节点下至少配置一个从节点(6个节点 正好三主各自有一从)

### 7.13.9.通过脚本重建集群

提供一个脚本shell（restart.sh）

修改其中的ip地址为自己节点的ip即可使用

内容如下：

for port in {8000..8005};do redis-cli -c -p $port shutdown;done;

rm -rf dump\*;

rm -rf appendonly\*;

rm -rf nodes\*;

for port in {8000..8005};do redis-server $port/redis-cluster.conf;done;

echo yes|./src/redis-trib.rb create --replicas 1 192.168.171.138:8000 192.168.171.138:8001 192.168.171.138:8002 192.168.171.138:8003 192.168.171.138:8004 192.168.171.138:8005;

replicas 1：表示每个主节点下有一个从节点

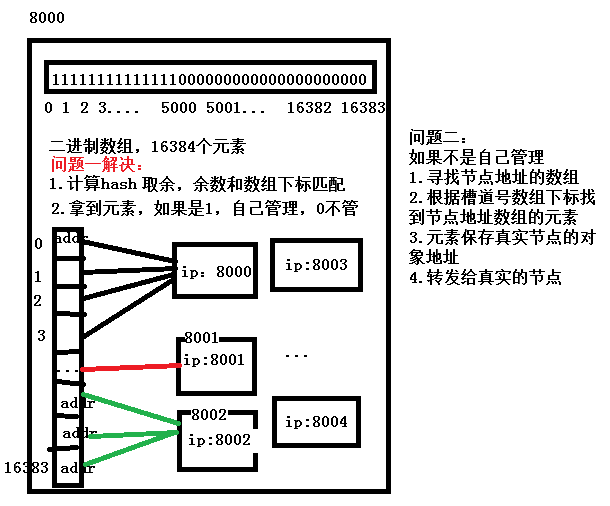
## 7.14.槽道原理

### 7.14.1.槽道两个问题

1.怎么判断是不是自己的

2.怎么知道给谁

### 7.14.2.槽道的组成部分



1.判断数组

16384个二进制组成的数据，本质上是2048个byte数组，当数据进来时，首先经过crc16()计算规则得到一个0-16383之间的数字（槽道号），通过位移取到槽道号对应的元素（非1即0），如果是1，则说明自己管理，如果是0 ，则不是自己管理

2.节点指向（索引）数组

16384个元素，和槽道号一一对应，每个元素存储的是真实节点对象的内存地址，当第一步判断不是自己管理时，从此数组中找到真正管理数据的节点对象，然后根据节点信息转发出去

### 7.14.3.二进制值的特点

主节点：有槽道管理权的，二进制数组每个都不同，同一个槽道号的二进制值，只有一个是1，剩下的都是0，没有槽道管理权的，二进制数组全是0；

从节点：从节点没有槽道管理权，所有的槽道号计算完毕，直接去索引数组获取节点信息并转发，但是，从节点会备份一个主节点的二进制数组，这个数组只是用来备份的，不使用

### 7.14.4.二进制的赋值

创建集群时，将节点管理的槽道号对应的二进制值从0变为1。

### 7.14.5.索性数组

节点相互握手过程中，每个节点都会记录集群的所有节点信息对象

索引数组的每个元素，保存的不仅仅是节点对象的地址，还要保存一些其他信息，例如槽道管理的所有的key

## 7.15.手动搭建redis

### 7.15.1.将集群清空

将已经创建的集群.恢复到初始状态

**（1）停**

[root@10-9-100-26 redis-3.2.11]# redis-cli -c -p 8000 shutdown

[root@10-9-100-26 redis-3.2.11]# redis-cli -c -p 8001 shutdown

[root@10-9-100-26 redis-3.2.11]# redis-cli -c -p 8002 shutdown

[root@10-9-100-26 redis-3.2.11]# redis-cli -c -p 8003 shutdown

[root@10-9-100-26 redis-3.2.11]# redis-cli -c -p 8004 shutdown

[root@10-9-100-26 redis-3.2.11]# redis-cli -c -p 8005 shutdown

**（2）删除持久化文件**

node文件,dump文件,append文件

[root@10-9-100-26 redis-3.2.11]# rm -rf appendonly800\*

[root@10-9-100-26 redis-3.2.11]# rm -rf dump800\*

[root@10-9-100-26 redis-3.2.11]# rm -rf nodes-800\*

**（3）启动所有节点(初始化的6个节点)**

[root@10-9-100-26 redis-3.2.11]# redis-server 8000/redis-cluster.conf

[root@10-9-100-26 redis-3.2.11]# redis-server 8001/redis-cluster.conf

[root@10-9-100-26 redis-3.2.11]# redis-server 8003/redis-cluster.conf

[root@10-9-100-26 redis-3.2.11]# redis-server 8002/redis-cluster.conf

[root@10-9-100-26 redis-3.2.11]# redis-server 8004/redis-cluster.conf

[root@10-9-100-26 redis-3.2.11]# redis-server 8005/redis-cluster.conf

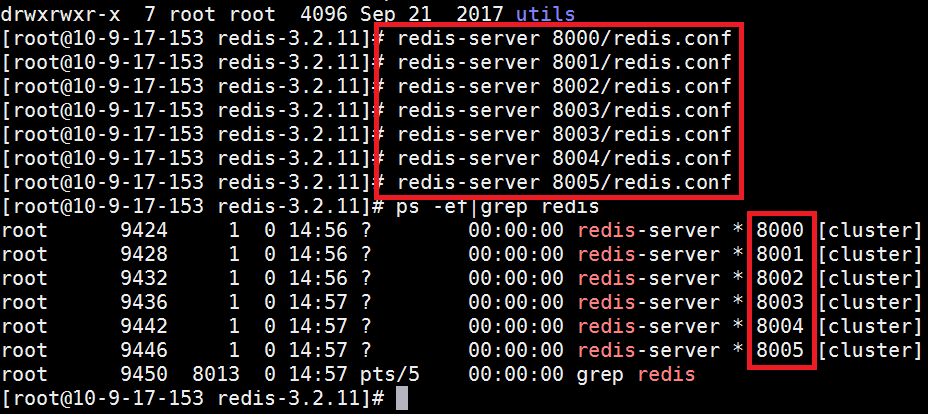
### 7.15.2.手动搭建集群（和ruby无关）

**（1）meet节点**

手动搭建:调用集群命令cluster meet 节点碰撞

注意:手动搭建集群,登录时用-h携带对外访问地址

1.启动节点 8000-8005全启动



2.节点握手

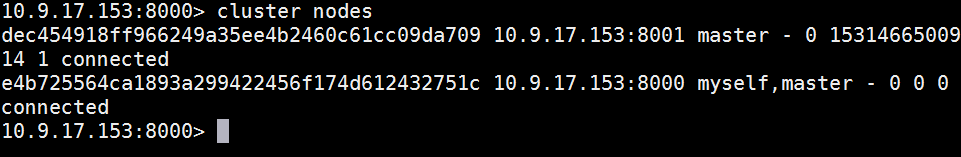
登录任何启动的节点; >cluster meet ip port(ip 端口是其他节点信息)

注意:跨服务器执行集群的搭建,需要登录时使用对外访问ip地址;

[root@10-9-17-153 redis-3.2.11]# redis-cli -c -p 8000 -h 10.9.17.153

10.9.17.153:8000>cluster meet 10.9.17.153 8001

初始化情况下,所有握手的节点都是master 8001属于有数据记录没清干净



添加其他节点到集群(槽道是没有分配的)

10.9.17.153:8000> cluster meet 10.9.17.153 8002

OK

10.9.17.153:8000> cluster meet 10.9.17.153 8003

OK

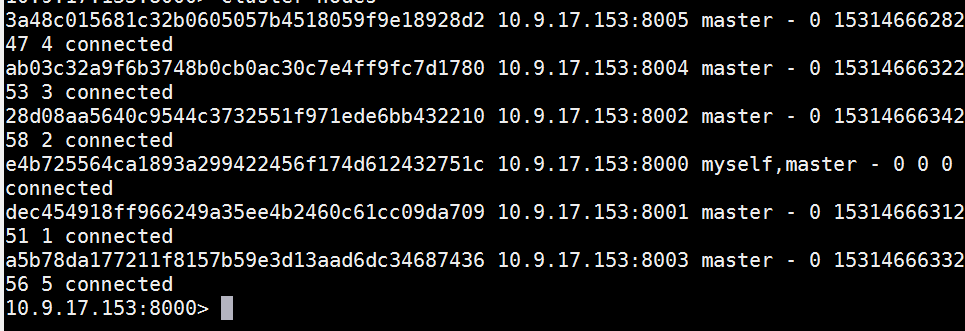
10.9.17.153:8000> cluster meet 10.9.17.153 8004

OK

10.9.17.153:8000> cluster meet 10.9.17.153 8005

OK

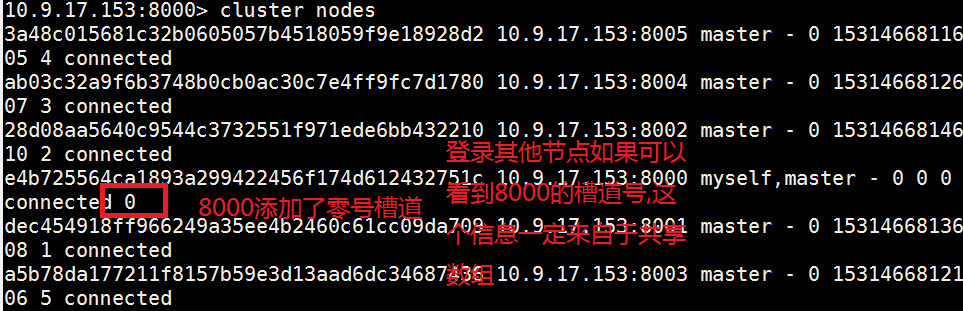
观察集群节点



8000-8002是主节点,8003-8005是从节点

3.先为8000-8002三个主节点分配槽道

8000>cluster addslots 槽道号 :可以1个,也可以多个用空格隔开



addslot执行完成后,当前节点和其他节点的共享数组发生变动,0号元素内容替换为8000的节点信息,和其他

所有节点的二进制,除了8000的第0位从0变成1,其他没变化

分配了一个槽道后,查看集群info状态,在16384个槽道全部分配完毕前,fail

分配其他槽道的脚本;这里使用到shell脚本;例如如下脚本内容;

[root@10-9-17-153 redis-3.2.11]# vim 8000slots0\_5461.sh

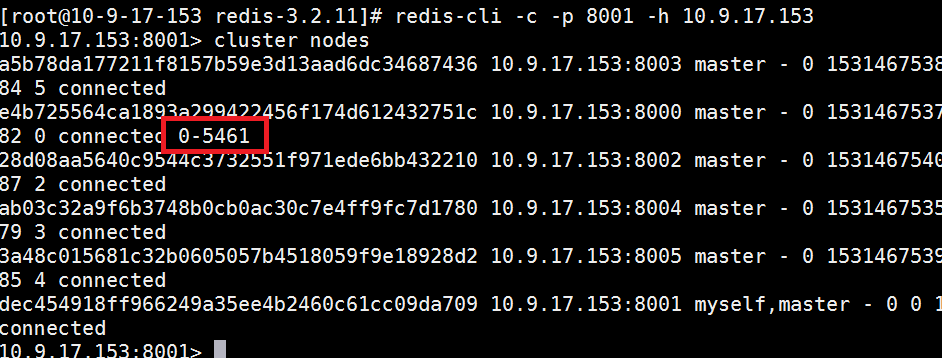
for port in {0..5461}; do redis-cli -h 10.9.17.153 -c -p 8000 cluster addslots $port;done(不要在文件中拷贝)

4.执行shell脚本

#sh 8000slots0\_5461.sh

调用执行shell脚本中的内容,将0-5461槽道挨个添加到当前8000节点中

查看集群节点信息,发现槽道分配完毕

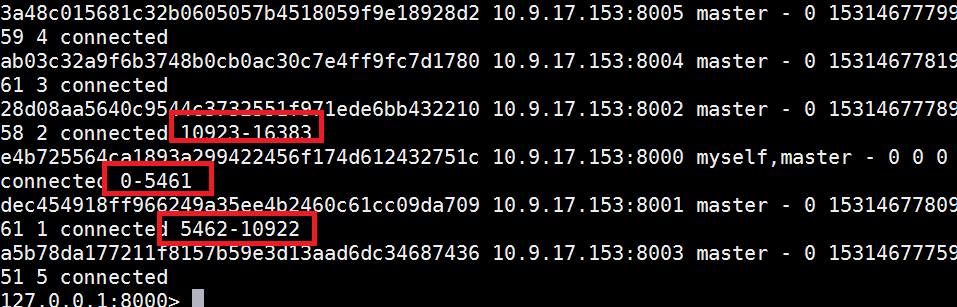


编辑其他两个shell脚本,各自执行脚本,完成 8001(5462-10922) 8002(10923-16383)的槽道分配

对应的shell脚本为

for i in {5462..10922}; do redis-cli -h 10.9.17.153 -c -p 8001 cluster addslots $i;done

for i in {10923..16383}; do redis-cli -h 10.9.17.153 -c -p 8002 cluster addslots $i;done



5.查看集群状态,ok

>cluster info

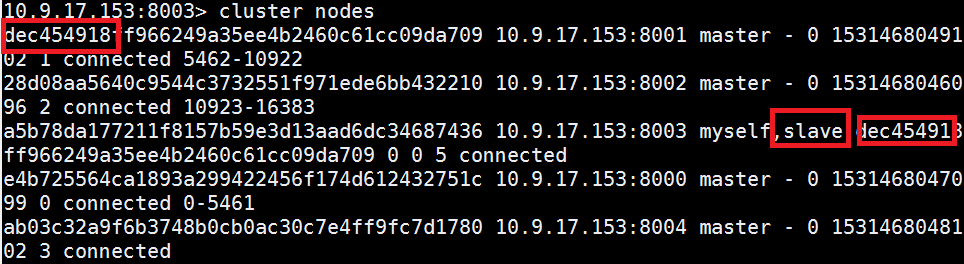
**（2）从节点的角色转化**

利用集群命令cluster来添加的,必须先meet进来

将8003登录,然后调用命令添加为8001的从节点

[root@10-9-17-153 redis-3.2.11]# redis-cli -h 10.9.17.153 -c -p 8003

10.9.17.153:8003> cluster replicate dec454918ff966249a35ee4b2460c61cc09da709(8001id)

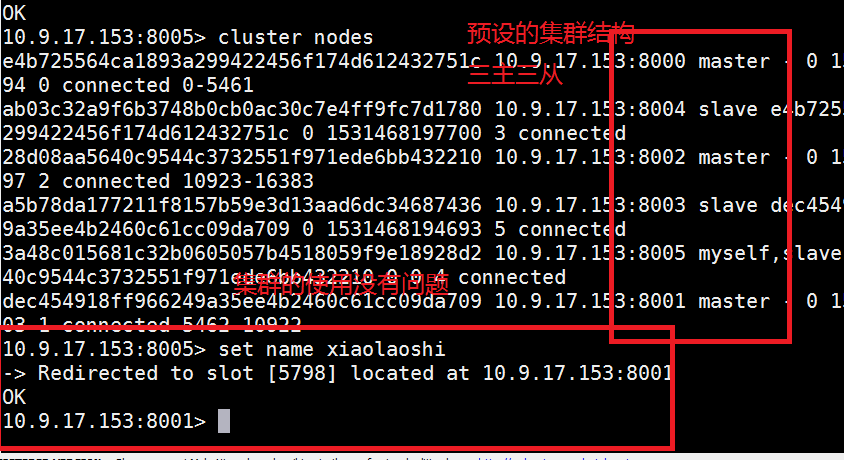


8004 8005的操作一样

登录8004,8005,

执行挂接到主节点id的cluster命令

测试集群



总结:redis-trib.rb 命令文件,就是利用ruby语言完成上述步骤的封装;对外提供方便的调用命令;

### 7.15.3.槽道的迁移(数据微调)

**（1）空槽道迁移(10923槽道)**

1.判断10923是否为空槽道,就是检查10923map的key是否对应一批key值

127.0.0.1:8002> cluster getkeysinslot 10923 500

(empty list or set)

2.目标节点(8001)导入槽道

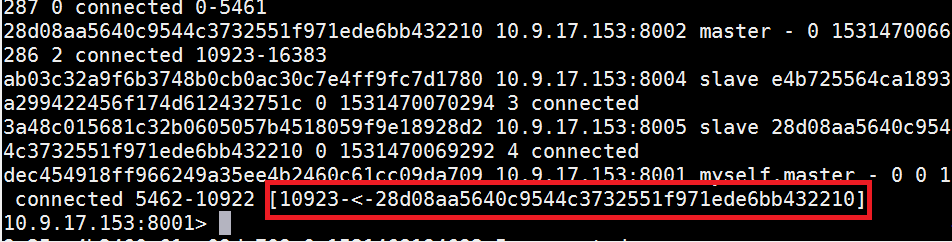
(目标节点中二进制不变, 数组当前槽道的状态变为导入) 正常状态是stable,槽道有四种状态,stable,importing导入状态,migrating导出状态,deleted删除状态

cluster setslot 10923 importing 源节点id

在8001上登录,执行命令,10923就是操作的槽道,importing将8001上的数组的10923元素中的槽道状态修改为导入importing 跟着的是源节点id 8002的id

10.9.17.153:8001> cluster setslot 10923 importing 28d08aa5640c9544c3732551f971ede6bb432210

从8001检查当前集群的节点状态



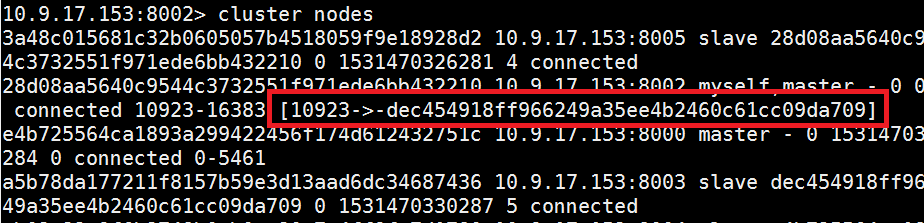
3.源节点(8002)导出槽道(源节点二进制不变,数组槽道状态变为导出)

登录8002执行迁出操作

cluster setslot 10923 migrating 目标节点id

10.9.17.153:8002> cluster setslot 10923 migrating dec454918ff966249a35ee4b2460c61cc09da709(8001的id)

通过cluster nodes观察到 8002的信息也发生了变化



槽道的状态是正常,导入,导出,都不影响数据的插入

数据微调的时候,集群不能对外提供服务;

数据迁移工作,在维护内容属于重大维护事项;

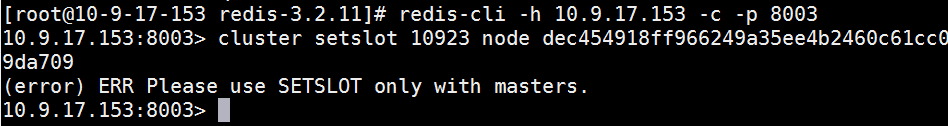
4.通知当事人节点槽道迁移了(所有的节点,数组发生变化)

cluster setslot 10923 node 8001节点id 所有集群节点都执行这个操作

将所有集群节点通知到,10923从现在开始,归8001所有

10.9.17.153:8001> cluster setslot 10923 node dec454918ff966249a35ee4b2460c61cc09da709 (8001二进制10923下标的二进制从0变成1,数组变了)

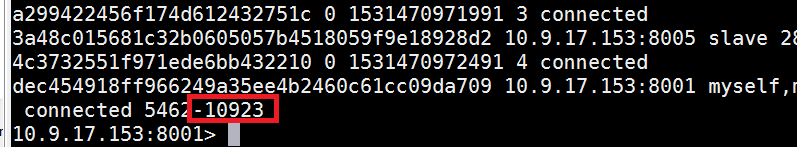
10.9.17.153:8002> cluster setslot 10923 node dec454918ff966249a35ee4b2460c61cc09da709 (8002二进制从1变成0,数组变了)



在8003上执行操作,提示只能再master执行更新

从节点同步主节点的信息,包括数组,和数据的备份

检查集群节点状态



空槽道迁移成功了

**（2）非空槽道迁移(reshard命令不支持)**

1.判断5798槽道是否非空

127.0.0.1:8001> cluster getkeysinslot 5798 500

1) "name"

按照以上的内容,有数据的槽道可以迁移,但是数组保存着 map{槽道号:[keys]};所以,要迁移非空槽道,需要同步的将数据keys一并迁移;

name 存储在5798上,迁移5798(源节点8001)目标节点8002

2.目标节点导入槽道

10.9.17.153:8002> cluster setslot 5798 importing dec454918ff966249a35ee4b2460c61cc09da709(源节点id)

3.源节点导出槽道

10.9.17.153:8001> cluster setslot 5798 migrating 28d08aa5640c9544c3732551f971ede6bb432210(目标节点id)

**4.将槽道的数据迁移到目标节点(登录源节点迁移数据)**

* + **确定当前槽道包含的所有key(5798)**

8001>cluster getkeysinslot 5798 500

查看5798上500个范围内的所有key;

[root@10-9-17-153 redis-3.2.11]# redis-cli -h 10.9.17.153 -c -p 8001 (登录8001)

10.9.17.153:8001> cluster getkeysinslot 5798 5 (获取槽道保存的数组信息)

1) "name"

* 将获取的key值,进行迁移,从8001迁移到8002

8001>migrate 目标节点ip 目标端口 "" 0 500 keys name

10.9.17.153:8001> migrate 10.9.17.153 8002 "" 0 500 keys name

"": 表示key的匹配; 可以使用正则(有待考证)

0: 表示迁移到目标节点的database(0-15)

500:连接超时毫秒

keys:标签确定迁移的key值名称

name:key

如果有多个key:keys name1 name2 name3 name4

除了key-value执行迁移,map记录{槽道:[key集合]}的内容一并迁移

5.所有当事人节点通知更新数据(槽道迁移)

10.9.17.153:8001> cluster setslot 5798 node 28d08aa5640c9544c3732551f971ede6bb432210

10.9.17.153:8002> cluster setslot 5798 node 28d08aa5640c9544c3732551f971ede6bb432210

6.最后测试

### 7.15.4.redis集群命令（与ruby无关）

**集群**

cluster info ：打印集群的信息

cluster nodes ：列出集群当前已知的所有节点（ node），以及这些节点的相关信息。

**节点**

cluster meet <ip> <port> ：将 ip 和 port 所指定的节点添加到集群当中，让它成为集群的一份子。

cluster forget <node\_id> ：从集群中移除 node\_id 指定的节点(保证空槽道)。

cluster replicate <node\_id> ：将当前节点设置为 node\_id 指定的节点的从节点。

cluster saveconfig ：将节点的配置文件保存到硬盘里面。

**槽(slot)**

cluster addslots <slot> [slot ...] ：将一个或多个槽（ slot）指派（ assign）给当前节点。

cluster delslots <slot> [slot ...] ：移除一个或多个槽对当前节点的指派。

cluster flushslots ：移除指派给当前节点的所有槽，让当前节点变成一个没有指派任何槽的节点。

cluster setslot <slot> node <node\_id> ：将槽 slot 指派给 node\_id 指定的节点，如果槽已经指派给

**另一个节点，那么先让另一个节点删除该槽>，然后再进行指派。**

cluster setslot <slot> migrating <node\_id> ：将本节点的槽 slot 迁移到 node\_id 指定的节点中。

cluster setslot <slot> importing <node\_id> ：从 node\_id 指定的节点中导入槽 slot 到本节点。

cluster setslot <slot> stable ：取消对槽 slot 的导入（ import）或者迁移（ migrate）。

**键**

cluster keyslot <key> ：计算键 key 应该被放置在哪个槽上。

cluster countkeysinslot <slot> ：返回槽 slot 目前包含的键值对数量。

cluster getkeysinslot <slot> <count> ：返回 count 个 slot 槽中的键

## 7.16.将JedisCluster整合到springboot

### 7.16.1.测试JedisCluster

JedisCluster自带连接池

|  |
| --- |
| @Test  **public** **void** test09(){  //1.收集节点信息  Set<HostAndPort> clusterSet = **new** HashSet<HostAndPort>();  //至少一个  clusterSet.add(**new** HostAndPort("192.168.171.138",8000));  //2.配置信息的对象  GenericObjectPoolConfig config = **new** GenericObjectPoolConfig();  config.setMaxTotal(100);  //3.创建JedisCluster对象（自带连接池）  JedisCluster jedis = **new** JedisCluster(clusterSet,config);  jedis.set("locationg","18");  } |

### 7.16.2.将JedisCluster交给springboot维护

以测试工程为例（springboot-redis-test01）

1.pom引入（redis，之前引入过）

2.配置文件

server.port=8095

spring.redis.cluster.nodes=192.168.171.138:8000,192.168.171.138:8001

spring.redis.cluster.maxTotal=100

spring.redis.cluster.maxIdle=10

spring.redis.cluster.minIdle=2

3.配置类读取配置文件内容，并通过@Bean创建对象和初始化参数

|  |
| --- |
| **package** cn.tedu.config;  **import** java.util.HashSet;  **import** java.util.List;  **import** java.util.Set;  **import** org.apache.commons.pool2.impl.GenericObjectPoolConfig;  **import** org.springframework.boot.context.properties.ConfigurationProperties;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** redis.clients.jedis.HostAndPort;  **import** redis.clients.jedis.JedisCluster;  @Configuration  @ConfigurationProperties(prefix="spring.redis.cluster")  **public** **class** RedisClusterConfig {  **private** List<String> nodes;  **private** Integer maxTotal;  **private** Integer maxIdle;  **private** Integer minIdle;  //初始化JedisCluster对象  @Bean  **public** JedisCluster initJedisCluster(){  //节点信息  Set<HostAndPort> set = **new** HashSet<HostAndPort>();  **for** (String node : nodes) {  set.add(**new** HostAndPort(node.split(":")[0],  Integer.*parseInt*(node.split(":")[1])));  }  //配置信息  GenericObjectPoolConfig config = **new** GenericObjectPoolConfig();  config.setMaxTotal(maxTotal);  config.setMaxIdle(maxIdle);  config.setMinIdle(minIdle);  //返回JedisCluster对象  **return** **new** JedisCluster(set,config);  }  get&set  } |

4.启动类+Controller（注入JedisCluster）

|  |
| --- |
| **package** cn.tedu;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  **import** redis.clients.jedis.JedisCluster;  @SpringBootApplication  @RestController  **public** **class** StarterRedisCluster {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(StarterRedisCluster.**class**, args);  }  @Autowired  **private** JedisCluster jedis;  @RequestMapping("cluster")  **public** String getandsetcluster(String key,String value){  //将传过来的key、value存入redis集群中  jedis.set(key, value);  //从集群中通过key获取value  **return** jedis.get(key);  }  } |

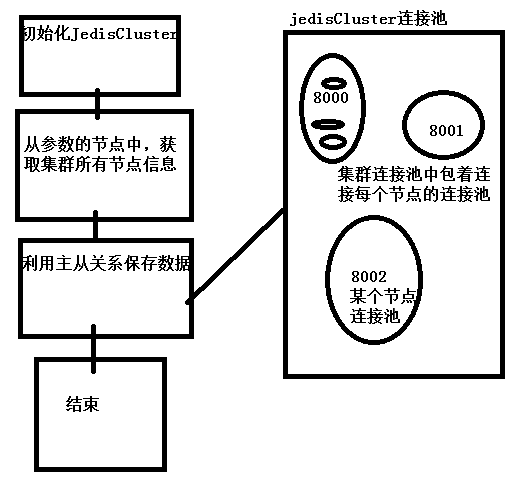
启动测试

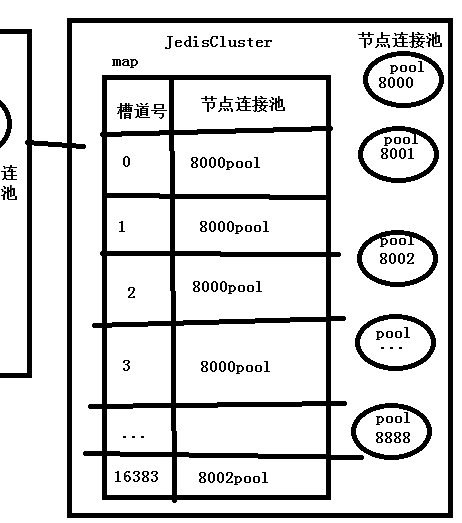
这里报错由于继承的是easymall，所以需要eureka，可以不用管

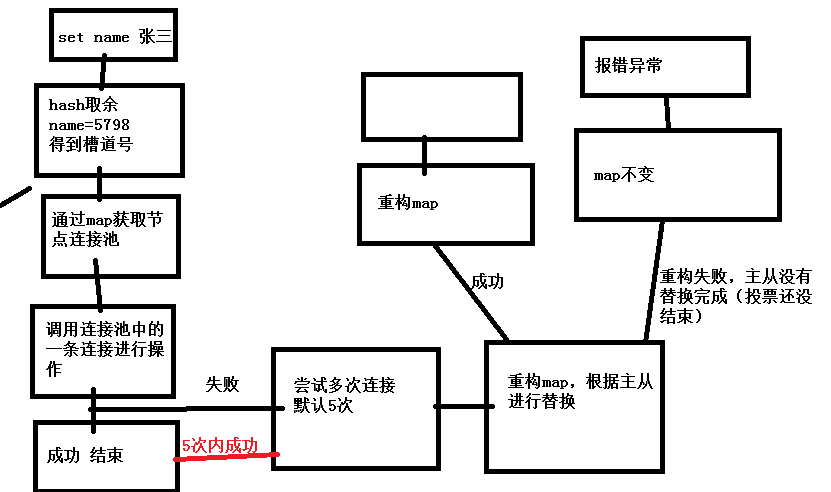
浏览器访问

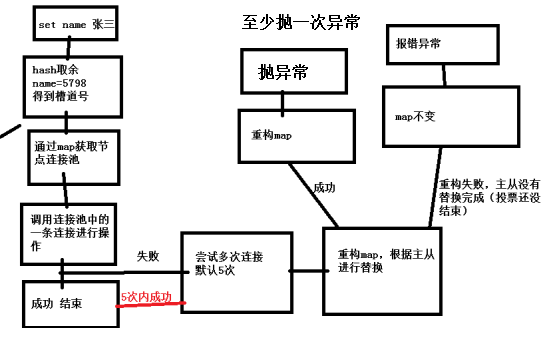
## 7.17.JedisCluster原理

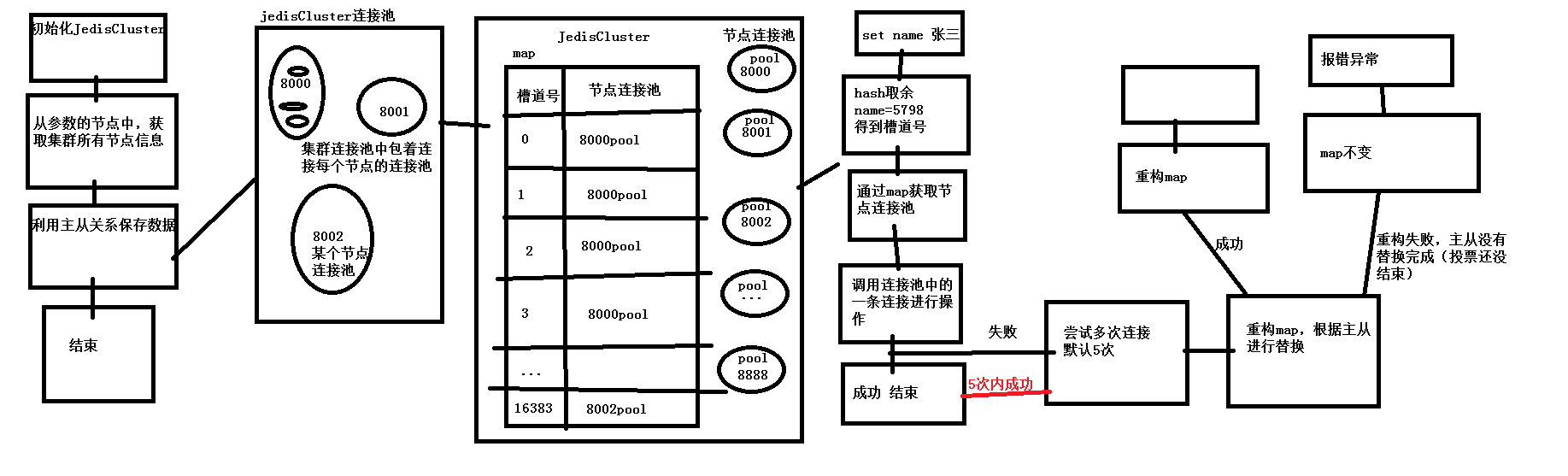
### 7.17.1.对象创建过程











### 7.17.2.JedisCluster依赖系统

1.quickstart(easymall-common-redisCluster)

2.pom引入easymall-parent、redis

<parent>

<groupId>cn.tedu</groupId>

<artifactId>easymall-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-redis</artifactId>

<version>1.4.7.RELEASE</version>

</dependency>

3.配置类读取配置文件并初始化JedisCluster对象交给框架

|  |
| --- |
| **package** cn.tedu.redis.cluster;  **import** java.util.HashSet;  **import** java.util.List;  **import** java.util.Set;  **import** org.apache.commons.pool2.impl.GenericObjectPoolConfig;  **import** org.springframework.boot.context.properties.ConfigurationProperties;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** redis.clients.jedis.HostAndPort;  **import** redis.clients.jedis.JedisCluster;  @Configuration  @ConfigurationProperties(prefix="spring.redis.cluster")  **public** **class** RedisClusterConfig {  **private** List<String> nodes;  **private** Integer maxTotal;  **private** Integer maxIdle;  **private** Integer minIdle;  @Bean  **public** JedisCluster initJedisCluster(){  //节点信息  Set<HostAndPort> set = **new** HashSet<HostAndPort>();  //将节点信息保存到set中  **for** (String node : nodes) {  set.add(**new** HostAndPort(node.split(":")[0],  Integer.*parseInt*(node.split(":")[1])));  }  //配置信息对象  GenericObjectPoolConfig config = **new** GenericObjectPoolConfig();  config.setMaxTotal(maxTotal);  config.setMaxIdle(maxIdle);  config.setMinIdle(minIdle);  //返回JedisCluster对象  **return** **new** JedisCluster(set,config);  }  get&set  } |

## 7.18.用户系统功能

### 7.18.1.引入RedisCluster

1.pom依赖easymall-common-redisClusert

2.配置文件

添加：

|  |
| --- |
| #RedisCluster  spring.redis.cluster.nodes=192.168.171.138:8000,192.168.171.138:8001  spring.redis.cluster.maxTotal=100  spring.redis.cluster.maxIdle=10  spring.redis.cluster.minIdle=5 |

3.哪用哪注入

### 7.18.2.登录功能改造

UserService

|  |
| --- |
| /\*@Autowired  private ShardedJedisPool pool;\*/  @Autowired  **private** JedisCluster jedis;  **public** String doLogin(User user) {  //ShardedJedis jedis = pool.getResource();  **try** {  //密码加密  user.setUserPassword(MD5Util.*md5*(user.getUserPassword()));  //数据库查询user对象  User getUser = userMapper.queryUserByUsernameAndPassword(user);  **if**(getUser == **null**){//用户名密码错误  **return** "";  }**else**{//用户名密码正确  //创建一个存入redis的key  String ticket = UUIDUtil.*getUUID*();  //将查到的user对象存入redis  //将user对象转换为json  ObjectMapper mapper = **new** ObjectMapper();  String userJson;    userJson = mapper.writeValueAsString(getUser);    //创建jedis对象  //Jedis jedis = new Jedis("192.168.171.138",6379);  jedis.set(ticket, userJson);  **return** ticket;  }  } **catch** (JsonProcessingException e) {  e.printStackTrace();  **return** "";  }/\*finally{  pool.returnResource(jedis);  }\*/    }  **public** String checkLogin(String ticket) {  //ShardedJedis jedis = pool.getResource();  **try** {  String userJson = jedis.get(ticket);  **return** userJson;  } **catch** (Exception e) {  e.printStackTrace();  **return** "";  }/\*finally{  pool.returnResource(jedis);  }\*/  } |

### 7.18.3.登录超时改造

问题一：存入redis的用户信息需要设置超时时间（30分钟）

解决：设置超时时间

问题二：登录突然断开，由于30分钟到期，用户访问时无法从redis拿到数据，用户登录断开

解决：方式一：每次访问刷新redis剩余时间

方式二：续约，每次访问判断剩余时间，达到某个阈值，再延长一段时间

方式一和方式二对比：

方式一：每次访问都需要对redis进行写操作，浪费redis的资源（效率偏低）

方式二：只有在剩余时间达到阈值才进行写操作，效率高

利用方式二解决登录突然断开问题

**1.设置超时时间**

|  |
| --- |
| **public** String doLogin(User user) {  ...  //jedis.set(ticket, userJson);  jedis.setex(ticket, 60\*30, userJson);  **return** ticket;  }  } **catch** (JsonProcessingException e) {  ...  } |

**2.登录突然断开（剩余10分，延长5分）**

|  |
| --- |
| **public** String checkLogin(String ticket) {  //ShardedJedis jedis = pool.getResource();  **try** {  //判断超时剩余时间  Long ttl = jedis.ttl(ticket);  //如果少于10分，在延长5分  **if**(ttl < 60\*10 && ttl>0){  //给key设置超时时间  jedis.expire(ticket, (**int**)(ttl+5\*60));  }  String userJson = jedis.get(ticket);  **return** userJson;  } **catch** (Exception e) {  e.printStackTrace();  **return** "";  }/\*finally{  pool.returnResource(jedis);  }\*/  } |

启动测试

问题，无法启动报错

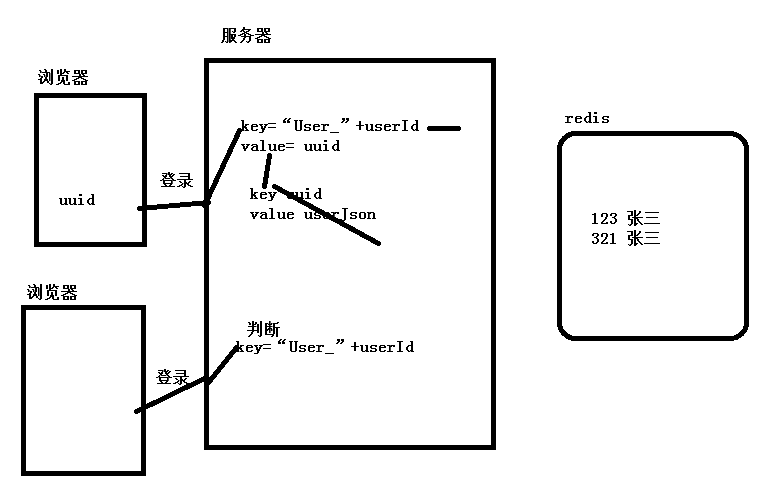
由于ShardedJedisPoolConfig配置类的存在，需要读取配置文件

解决方式一：配置文件正常配置

方式二：删除类或者删除类的注解

### 7.18.4.登录顶替

只允许一个用户在一个客户端登录



每个用户登录时，在redis保存两组数据

第一组：key：用户id相关，value：uuid（第二组key的）

第二组：key：uuid，value：userJson

登录逻辑：判断第一组key是否存在，如果不存在，则无人登录，直接登录逻辑

如果存在，说明有人登录，通过用户id，删除之前的两组数据，放入新的两组数据

UserService

|  |
| --- |
| **public** String doLogin(User user) {  //ShardedJedis jedis = pool.getResource();  **try** {  //密码加密  user.setUserPassword(MD5Util.*md5*(user.getUserPassword()));  //数据库查询user对象  User getUser = userMapper.queryUserByUsernameAndPassword(user);  **if**(getUser == **null**){//用户名密码错误  **return** "";  }**else**{//用户名密码正确  //创建一个存入redis的key  String ticket = UUIDUtil.*getUUID*();  //将查到的user对象存入redis  //将user对象转换为json  ObjectMapper mapper = **new** ObjectMapper();  String userJson;    userJson = mapper.writeValueAsString(getUser);    //创建jedis对象  //Jedis jedis = new Jedis("192.168.171.138",6379);  //jedis.set(ticket, userJson);  //判断之前是否有人登陆  String exisTicket = jedis.get("user\_loginId"+getUser.getUserId());  **if**(StringUtils.*isNotEmpty*(exisTicket)){//不空，之前已经有人登录  //删除前人的数据  jedis.del(exisTicket);  }  //放入判断逻辑数据和真实数据  jedis.setex("user\_loginId"+getUser.getUserId(), 60\*30, ticket);  jedis.setex(ticket, 60\*30, userJson);  **return** ticket;  }  } |

如何做到一个账户最多三人登录

提示，利用redis的list结构，将UserId相关的key存入此list中，每次登录进行判断，如果list.size<3直接登录，else踢出去一个（lpush和rpop）

## 7.19.商品系统

### 7.19.1.整合redis

1.pom依赖redis-cluster系统

2.配置文件（拷贝User系统的）

#RedisCluster

spring.redis.cluster.nodes=192.168.171.138:8000,192.168.171.138:8001

spring.redis.cluster.maxTotal=100

spring.redis.cluster.maxIdle=10

spring.redis.cluster.minIdle=5

3.哪用哪注入（使用时）

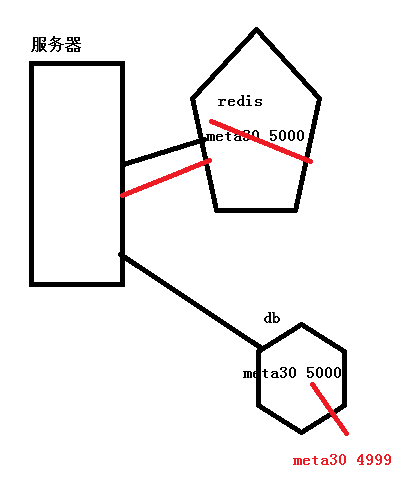
### 7.19.2.商品的缓存

ProductService

|  |
| --- |
| @Autowired  JedisCluster jedis;  **public** Product queryById(String productId) {  **try** {  //**TODO** 1.去redis查  //根据key去redis中查  String productKey = "product\_query"+productId;  //判断redis中是否存在  **if**(jedis.exists(productKey)){//缓存有数据  //从缓存获取数据  String productJson = jedis.get(productKey);  //将json转换为对象  Product product = MapperUtil.***MP***.readValue(productJson, Product.**class**);  **return** product;  }**else**{  //2.去数据库  //每次去数据库查，非常耗费资源，尤其是热门商品，去之前，先去redis查  Product prod = productMapper.queryById(productId);  //**TODO** 3.将数据放入redis  //将从数据库查的对象转换成json存入redis  String prodJson = MapperUtil.***MP***.writeValueAsString(prod);  //设置超时存储  jedis.setex(productKey,60\*60\*24\*2, prodJson);  **return** prod;  }  } **catch** (Exception e) {  e.printStackTrace();  **return** **null**;  }  } |

### 7.19.3.并发锁

**（1）商品修改而缓存未修改问题**



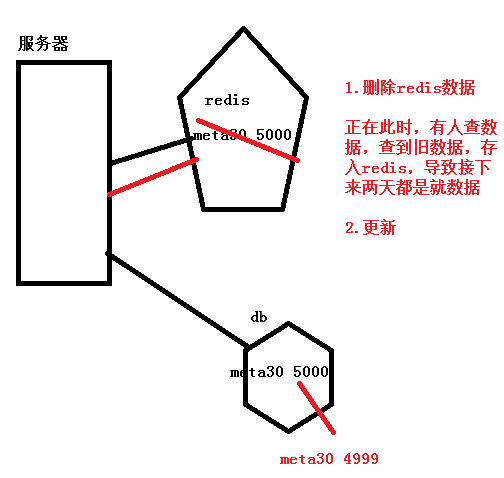
数据库修改商品属性时，需要删除redis中的对应数据，否则用户看到的数据与数据库数据不一致

修改商品更新逻辑，更新同时删除redis对应数据

ProductService

|  |
| --- |
| **public** **void** productUpdate(Product product) {  //必须更新前删，否则万一删除失败，读的还是旧数据  jedis.del("product\_query"+product.getProductId());  productMapper.productUpdate(product);  } |

**（2）高并发缓存问题**



高并发下执行修改逻辑

1.删除redis

2.用户查询商品，redis没有，数据库查询（旧数据），存入redis（高并发，第三步还没来得及执行）

3.更新数据库

结论，即使数据库更新前删除redis数据，也可能导致拿到2天旧数据

解决办法：更新过程加锁

ProductService

|  |
| --- |
| **public** **void** productUpdate(Product product) {  //更新之前加锁  String lock = "product\_query"+product.getProductId()+".lock";  //将锁设置到Redis中，并设置超时时间为：商品的剩余时间  //获取商品剩余时间  Long leftTime = jedis.ttl("product\_query"+product.getProductId());  //加锁  jedis.setex(lock, Integer.*parseInt*(leftTime+""), "");  //必须更新前删，否则万一删除失败，读的还是旧数据  jedis.del("product\_query"+product.getProductId());  productMapper.productUpdate(product);  //解锁-即从redis中删除锁  jedis.del(lock);  } |

ProductService

每次查询判断此商品是否有锁

|  |
| --- |
| **public** Product queryById(String productId) {  **try** {  //**TODO** 1.去redis查  //根据key去redis中查  String productKey = "product\_query"+productId;  //判断是否有锁  **if**(jedis.exists(productKey+".lock")){//有锁  **return** productMapper.queryById(productId);  }  //判断redis中是否存在  **if**(jedis.exists(productKey)){//缓存有数据  ...  }  } **catch** (Exception e) {  e.printStackTrace();  **return** **null**;  }  } |

### 7.19.4.发送验证码

**逻辑：**

系统生成验证码，发送给用户，用户输入验证码，系统进行比对是否一致。

可以将验证码存入redis

1.生成逻辑key：”valString\_”+userPhone value:5678（验证码）（需要设置超时时间）

2.将验证码发给用户，调用运营商的接口

3.用户输入验证码并提交

4.获取提交的验证码和手机号，去redis查找对应的验证码数据，进行比较

以上逻辑存在问题：如果黑客通过代码无限请求，则需要无限次给用户发送验证码（很浪费钱）

思考题：如何解决

提示：判断是否已发送，也可以实现几分钟只能最多发几次