# Redis笔记

## 15.1非关系型数据库特点

1. 数据模型比较简单
2. 需要灵活性更强的IT系统
3. 对数据库性能要求高
4. 不需要高度的数据一致性
5. 对于给定的key，比较容易映射复杂的环境

## 15.2 Redis简介

redis是以key-value形式存储，和传统的关系型数据库不一样，不一定遵循传统数据库的一些基本要求（非关系型的，分布式的，开源的，水平可扩展的）

优点：对数据高并发读写

对海量数据的搞笑率存储和访问

对数据的可扩展性和高可用性

缺点：redis（ACID处理非常简单）

无法做到太复杂的关系型数据库模型

Redis是以key-value store存储，data structure service 数据结构服务器。键可以包含(string)字符串，哈希，(list)链表，(set)集合，(zset)有序集合。这些数据结合都支持push/pop、add/remove 及交集和并集以及更丰富的操作，redis支持各种不同的方式排序，为了保证效率，数据都是缓存在内存中，它也可以周期性的把更新的数据写入磁盘或者吧修改操作写入追加到文件中。

## 15.3 LinuxRedis安装

参考博客https://www.cnblogs.com/zuidongfeng/p/8032505.html

redis 默认占用6379端口

### 15.3.1 下载redis 包

在命令窗口中输入 wget http://download.redis.io/releases/redis-4.0.6.tar.gz

### 15.3.2 解压redis压缩包

tar -zxvf redis-4.0.6.tar.gz

### 15.3.3 yum安装gcc依赖

yum install gcc

### 15.3.4 编译安装

cd redis-4.0.6

编译安装

make MALLOC=libc

将redis-4.0.6/src目录下的文件加到/user/local/bin目录

cd src &&make install

### 15.3.5 进入src目录测试启动redis

输入指令 ./redis-server

### 15.3.6 设置后台进程访问

1.修改redis.conf文件

将daemonize no => 修改为 daemonize yes

2.指定redis.conf文件启动

./redis-server /sortware/redis-4.0.6/redis.conf

3.查看redis进程

ps -aux |grep redis

4.杀死进程

kill -9 进程id

### 15.3.7 开机自启动

1.进入 etc目录创建redis文件夹

mkdir redis

2.进入安装的redis目录将redis.conf文件考分到etc下的redis目录并且需要将名称更改为6379.conf

cp redis.conf /etc/redis/6379.conf

3.将redis的启动脚本复制到/etc/init.d目录下

进入安装包下的utils目录

cp redis\_init\_script /etc/init.d/redisd

4.设置开机自启动先进入init.d目录下

输入cp redis\_init\_script /etc/init.d/redisd

出现service redisd does not support chkconfig

redisd不支持chkconfig

解决：

vim编辑redisd文件在第一行加入两行注释保存退出

在第一行添加#chkconfig:2345 90 10

在第二行添加#description: Redis is a persistent key -value database

注释的意思是redis服务必须运行级2，3，4，5下被启动或者关闭，启动的优先级是90，关闭的优先级是10

#!/bin/sh

#chkconfig:2345 90 10

#description: Redis is a persistent key-val database

5.再次执行开启自启动命令

chkconfig redisd on

### 15.3.8 redis 开启关闭命令

service redisd start

service redisd stop

### 15.3.9 进入redis服务器

|  |
| --- |
| [root@localhost ~]# redis-cli  127.0.0.1:6379> |

## 15.4 redis基本数据类型

redis一共分为5中数据类型：String，Hash，LIst，Set，ZSet

String类型是包含很多种类型的特殊类型，并且是二进制安全的。比如序列化的对象进行存储，比如一张图片进行二进制存储，比如一个简单的字符串，数值等等。

set和get方法：

设置值：set name bhz 取值get name （说明设置name多次覆盖）

删除值：del name

## 15.5 String类型

### 15.4.1 keys

|  |
| --- |
| 127.0.0.1:6379> keys \* 输入keys \* 获取所有redis的键值  (empty list or set) |

### 15.4.2 设置键值

|  |
| --- |
| 127.0.0.1:6379> set user admin 设置user 的值为admin  OK |

### 15.4.3获取值

|  |
| --- |
| 127.0.0.1:6379> get user 使用get来获取key的值  "admin" |

### 15.4.4 删除key

|  |
| --- |
| 127.0.0.1:6379> del user 删除key 返回1  (integer) 1 |

### 15.4.5 setnx 不存在设置值，存在返回0

|  |
| --- |
| 127.0.0.1:6379> setnx user 123  (integer) 1  127.0.0.1:6379> setnx user 1234  (integer) 0 |

### 15.4.6 setex 设置有效期，有效期过后返回nil（即null）

|  |
| --- |
| 127.0.0.1:6379> setex fliay 10 22222 设置fliay的值为22222有效期为10秒钟  OK  127.0.0.1:6379> get fliay 10秒过后获取值返回结果为nil  (nil) |

### 15.4.6 setrange 替换字符串

|  |
| --- |
| 127.0.0.1:6379> set fliay 100  OK  127.0.0.1:6379> setrange fliay 2 a 2表示需要替换的位置此处是将2替换成2  (integer) 3  127.0.0.1:6379> get fliay  "10a" 获取值返回结果10a |

### 15.4.7 mset mget 一次性设置多个值和一次性获取多个值

|  |
| --- |
| 127.0.0.1:6379> mset a 1 b 2 c 3 d 4  OK  127.0.0.1:6379> mget a b c d e  1) "1"  2) "2"  3) "3"  4) "4"  5) (nil) |

### 15.4.8 getset 先获取旧值再获取新值

|  |
| --- |
| 127.0.0.1:6379> getset fliay 520  "10a"  127.0.0.1:6379> get fliay  "520" |

15.4.9 incr decr 递增递减

|  |
| --- |
| 127.0.0.1:6379> get fliay  "520"  127.0.0.1:6379> incr fliay  (integer) 521  127.0.0.1:6379> decr fliay  (integer) 520 |

### 15.4.10 incrby decrby 指定值递增 递减

|  |
| --- |
| 127.0.0.1:6379> get fliay  "520"  127.0.0.1:6379> incrby fliay 3  (integer) 523  127.0.0.1:6379> decrby fliay 10  (integer) 513 |

### 15.4.11 append 字符串追加

|  |
| --- |
| 127.0.0.1:6379> get fliay  "513"  127.0.0.1:6379> append fliay hello  (integer) 8  127.0.0.1:6379> get fliay  "513hello" |

### 15.4.12 strlen 获取字符串长度

|  |
| --- |
| 127.0.0.1:6379> get fliay  "513hello"  127.0.0.1:6379> strlen fliay  (integer) 8 |

## 15.6 Hash 类型

Hash类型是String的field和value的映射，或者说是一个String集合，它特别适合存储对象，相比较而言，将一个对象类型存储在Hash类型里要比存储在String类型里占用更少的内存空间，并且方便存取整个对象。

### 15.6.1 hset hget hash键值存储

hset myhset field hello) hset集合，myhash是集合名字 field是字段名 hello是值

|  |
| --- |
| 127.0.0.1:6379> hset user name fliay  (integer) 1  127.0.0.1:6379> hset user age 28  (integer) 1  127.0.0.1:6379> hset user id 1  (integer) 1  127.0.0.1:6379> hget user name  "fliay" |

### 15.6.2 hmset hmget hash键值批量存取

|  |
| --- |
| 127.0.0.1:6379> hmset student name fliay  OK  127.0.0.1:6379> hmset student age 8  OK  127.0.0.1:6379> hmset student id 1  OK  127.0.0.1:6379> hmget student id age name  1) "1"  2) "8"  3) "fliay" |

### 15.6.3 hsetnx 与setnx 大同小异（不存在则添加，存在则返回0）

|  |
| --- |
| 127.0.0.1:6379> hsetnx user id 1  (integer) 1  127.0.0.1:6379> hsetnx user id 2  (integer) 0 |

### 15.6.4 hincrby 和hdecrby 集合递增和递减

|  |
| --- |
| 127.0.0.1:6379> hset teacher age 33  (integer) 1  127.0.0.1:6379> hget teacher age  "33"  127.0.0.1:6379> hincrby teacher age 1  (integer) 34  127.0.0.1:6379> hget teacher age  "34" |

### 15.6.5 hexists 判断是否存在如果存在返回，不存在返回0

|  |
| --- |
| 127.0.0.1:6379> hexists user name  (integer) 0  127.0.0.1:6379> hexists teacher age  (integer) 1 |

### 15.6.6 hlen返回hash集合里的所有键值数

|  |
| --- |
| 127.0.0.1:6379> hlen student  (integer) 3 |

### 15.6.7 hdel删除指定hash的field

|  |
| --- |
| 127.0.0.1:6379> hdel student age  (integer) 1  127.0.0.1:6379> hget student age  (nil) |

### 15.6.8 hkeys 获取hash里面所有的hash

|  |
| --- |
| 127.0.0.1:6379> hkeys student  1) "name"  2) "id" |

### 15.6.9 hvals 获取hash里面所有的value

|  |
| --- |
| 127.0.0.1:6379> hvals student  1) "fliay"  2) "2" |

### 15.6.10 hgetall 获取hash里面所有的key和value

|  |
| --- |
| 127.0.0.1:6379> hgetall student  1) "name"  2) "fliay"  3) "id"  4) "2" |

## 15.7 List 类型

List类型是一个链表结构的集合，主要功能有push,pop,获取元素等，更详细的说，List类型是一个双端链表的结构，我们可以通过相关操作进行集合的头部或者尾部添加删除元素，List的设计非常简单精巧，既可以做栈又可以做队列，满足绝大数需求。

### 15.7.1 lpush 从头部加速元素（栈）先进后出

|  |
| --- |
| 127.0.0.1:6379> lpush list1 1 2 3 4 5 6 7  (integer) 7 |

### 15.7.2 lrange 从结合中取数据

|  |
| --- |
| 127.0.0.1:6379> lrange list1 0 5  1) "7"  2) "6"  3) "5"  4) "4"  5) "3"  6) "2" |

### 15.7.3 rpush 从尾部加入元素（队列）先进先出（取数据同上）

|  |
| --- |
| 127.0.0.1:6379> lpush list1 1 2 3 4 5 6 7  (integer) 7 |

### 15.7.4 linsert 插入元素

|  |
| --- |
| 127.0.0.1:6379> linsert list before 5 999  (integer) 0  127.0.0.1:6379> linsert list1 before 5 999  (integer) 9  127.0.0.1:6379> lrange list1 0 5  1) "7"  2) "6"  3) "999"  4) "5" |

### 15.7.5 lset 将指定下标的元素替换掉

|  |
| --- |
| 127.0.0.1:6379> lset list1 3 2  OK  127.0.0.1:6379> LRANGE list1 0 5  1) "7"  2) "6"  3) "999"  4) "2"  5) "4"  6) "3" |

### 15.7.6 lrem 删除元素，返回删除个数

|  |
| --- |
| 127.0.0.1:6379> LREM list1 1 7  (integer) 1  127.0.0.1:6379> LRANGE list1 0 5  1) "6"  2) "999"  3) "2"  4) "4"  5) "3"  6) "2" |

### 15.7.7 ltrim 保留指定key的值范围内的数据

|  |
| --- |
| 127.0.0.1:6379> ltrim list1 0 3  OK  127.0.0.1:6379> LRANGE list1 0 8  1) "6"  2) "999"  3) "2"  4) "4"  127.0.0.1:6379> |

### 15.7.8 lpop 从list的头部删除元素，并返回删除元素

|  |
| --- |
| 127.0.0.1:6379> LPOP list1  "6" |

### 15.7.9 rpop 从list的尾部删除元素，并返回删除元素

|  |
| --- |
| 127.0.0.1:6379> rpop list1  "4" |

### 15.7.10 rpoplpush 先从尾部删除元素，然后从头部加入元素

|  |
| --- |
| 127.0.0.1:6379> RPOPLPUSH list1 1231  "999"  127.0.0.1:6379> LRANGE list1 0 -1  1) "7"  2) "6"  3) "5"  4) "4"  5) "3"  6) "2"  7) "1" |

### 15.7.11 lindex 返回名称为key的list中index位置的元素

|  |
| --- |
| 127.0.0.1:6379> lindex list1 1  "6" |

### 15.7.12 llen 返回元素个数

|  |
| --- |
| 127.0.0.1:6379> llen list1  (integer) 7 |

## 15.8 set类型和zset类型

set集合是string类型的无序集合，set是通过hashtable实现的，对集合我们可以取交集、并集、差集

set是无序的集合，zset是有序的集合

### 15.8.1 sadd 向名称为key的set集合中添加元素

|  |
| --- |
| 127.0.0.1:6379> sadd user1 aaa  (integer) 1  127.0.0.1:6379> sadd user1 bbb  (integer) 1  127.0.0.1:6379> sadd user1 ccc  (integer) 1  127.0.0.1:6379> SMEMBERS user1 使用SMEMBERS 查询  1) "aaa"  2) "bbb"  3) "ccc" |

### 15.8.2 srem 删除set集合中的元素

### 15.8.3 spop 随机返回删除的key

### 15.8.4 sdiff 返回两个集合的不同元素（哪个集合在前面就以哪个集合为标准）

### 15.8.5 sdiffstore 将返回的不同元素存储到另外一个集合里

### 15.8.6 smove 从一个set集合移动到另一个set集合

### 15.8.7 scard 查看集合里元素个数

### 15.8.8 sismember 判断某元素是否为集合中的元素（1代表集合中的元素 0代表不是）

### 15.8.9 srandmember 随机返回一个元素

### 15.8.10 zadd 有序的集合

|  |
| --- |
| 127.0.0.1:6379> zadd u 1 1  (integer) 1  127.0.0.1:6379> zadd u 2 2  (integer) 1  127.0.0.1:6379> zadd u 5 5  (integer) 1  127.0.0.1:6379> zadd u 4 4  (integer) 1  127.0.0.1:6379> zrange u 0 -1 withscores  1) "1"  2) "1"  3) "2"  4) "2"  5) "4"  6) "4"  7) "5"  8) "5" |

## 15.9 Redis高级命令

### 15.9.1 expire 设置过期时间 使用ttl查看还有多久过期

|  |
| --- |
| 127.0.0.1:6379> set name fliay  OK  127.0.0.1:6379> EXPIRE name 20 设置过期时间为20秒  (integer) 1  127.0.0.1:6379> ttl name  (integer) 5 返回还有5秒过期 |

### 15.9.2 psersist 取消过期时间

### 15.9.3 select 选择数据库

redis一共有16个数据库

|  |
| --- |
| 127.0.0.1:6379> select 1 选择1号数据库  OK |

### 15.9.4 move 将数据移动到其它数据库

|  |
| --- |
| 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]> set age 11  OK  127.0.0.1:6379[1]> keys \*   1. "age"   127.0.0.1:6379> move user 1 将数据库0中的user移动到数据库1中  (integer) 1  127.0.0.1:6379> select 1 选择数据库1  OK  127.0.0.1:6379[1]> keys \*  1) "user"  2) "age" |

### 15.9.5 randomkey 随机返回数据里面的一个key

|  |
| --- |
| 127.0.0.1:6379[1]> randomkey  "user" |

15.9.6 rename 重命名key

|  |
| --- |
| 127.0.0.1:6379[1]> rename user users  OK  127.0.0.1:6379[1]> keys \*  1) "users"  2) "age" |

## 15.10 主从复制

### 15.10.1 主从复制

1.一个master可以拥有多个slave

2.多个slave可以连接同一个master之外还可以连接到其他的slave

3.主从复制不会阻塞在同步数据时master可以继续处理client请求

4.提供系统的伸缩性

## 15.11 主从配置

### 15.11.1 配置slaveof从服务器

|  |
| --- |
| vim redis.conf 找到如下行配置  主服务器需要添加密码  requirepass 123456 （主从都要设置）  masterauth 123456 （主从都要设置）  # slaveof <masterip> <masterport>  slaveof 192.168.1.149 6379  配置完成 |

## 15.12 启动添加密码以及关闭修改shell

|  |
| --- |
| service redis start -a 123456  修改redis服务脚本，加入如下所示的红色授权信息即可：  vi /etc/init.d/redis  $CLIEXEC -a "password" -p $REDISPORT shutdown  redis-cli -a 123456 |

## 15.13 哨兵

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 有了主从复制的实现以后，我们如果想对主从服务器进行监控，那么在redis2.5以后提供了一个“哨兵”的机制，顾名思义，哨兵的意思就是监控Redis系统的运行状况，主要功能有两点：   1. 监控著数据库和从数据库是否正常运行 2. 主数据库出现故障的时候，可以自动将从数据库转换为主数据库，实现自动切换。(在其中一台从服务器进行配置) 3. copy文件sentinel.conf 到 /etc/redis中 4. 修改sentinel.conf文件   sentinel monltor mymaster 192.168.1.149 6379 #名称、IP、端口、投票选举次数  sentinel down-after-millisenconds mymaster 5000 #默认1s监测一次，这里配置超时5000毫秒为宕机。  sentinel failover-timeout mymaster 900000  sentinel parallel-syncs mymaster 2  sentinel can-failover mymaster yes   1. 启动sentinel哨兵   redis-server /etc/redis/sentinel.conf --sentinel&   1. 查看哨兵相关信息命令   redis-cli -h 192.168.1.155 -p 26379 info Sentinel   1. 关闭朱富强查看集群信息   redis-cli -h 192.168.1.149 -p 6379 shutdown   1. 实现步骤 2. 进入redis安装的目录将sentinel.conf 拷贝到/etc/redis  |  | | --- | | cp sentinel.conf /etc/redis/ |  1. 查看sentinel.conf 哨兵的端口号是port 26379   将所有日志，数据输出到/etc/redis/目录下   |  | | --- | | dir /etc/redis |  1. 配置哨兵  |  | | --- | | sentinel monitor mymaster 192.168.1.149 6379 2 #名称、IP、端口、投票选举次数 |  1. 配置宕机时间  |  | | --- | | sentinel down-after-milliseconds mymaster 30000 #默认1秒检测一次，这里配置30秒检测一次 |  1. 配置超时时间  |  | | --- | | sentinel failover-timeout mymaster 180000 #配置故障转移时间为18秒 |  1. 配置从节点数量  |  | | --- | | sentinel parallel-syncs mymaster 2 #配置从节点数量 |  1. 启动  |  | | --- | | redis-server sentinel.conf --sentinel & #添加shutdown为关闭 |  1. 查看哨兵命令  |  | | --- | | redis-cli -h 192.168.1.155 -p 26379 info Sentinel | |

## 15.14 事务

redis的事务是非常简单的，使用方法如下：

首先是hi用multi方法打开事务，然后进行设置，这时设置的数据都会放入列表里进行保存，最后使用exec执行，吧数据一次存储到redis中，使用discard方法取消事务。

|  |
| --- |
| 127.0.0.1:6379> multi  OK  127.0.0.1:6379> set a 1  QUEUED  127.0.0.1:6379> set b 2  QUEUED  127.0.0.1:6379> set c 3  QUEUED  127.0.0.1:6379> exec 进行事务提交  1) OK  2) OK  3) OK  127.0.0.1:6379> multi 执行取消事务  OK  127.0.0.1:6379> set aa 1  QUEUED  127.0.0.1:6379> set bb2  (error) ERR wrong number of arguments for 'set' command  127.0.0.1:6379> set bb 2  QUEUED  127.0.0.1:6379> set cc 3  QUEUED  127.0.0.1:6379> discard  OK |

## 15.15 Redis 持久化机制

redis是一个支持持久化的内存数据库，也就是说redis需要经常讲内存中的数据同步到硬盘中保证持久化。

### 15.15.1 snapshotting （快照）默认方式

将内存中以快照的方式写入到二进制文件中，默认为dump.rdb.可以通过配置自动做快照持久化的方式。我们可以配置redis在n秒内如果超过m个key则修改就自动做快照。

snapshotting配置

save 900 1 #900秒内如果超过1个key被修改，则发起快照保存

save 300 10 #300 秒内如果超过10个key被修改，则发起快照保存

save 60 10000 。。。。。。。。。。

### 15.15.2 append-only file（缩写aof）的方式

（有点类似于oracle日志）由于快照方式是在一定时间间隔做一次，所以可能发生redis意外down的情况就会丢失最后一次快照的所有修改的数据、aof对比快照方式有更好的持久化性，是由于在使用aof时，redis会将每一个收到的写明了都通过write函数追加到命令中。当redis重新启动的时候会重置执行文件中保存的写命令来在内存中重建这个数据库的内容。这个文件在bin目录下：appendonly.aof。aof不是立即写到硬盘上，可以通过配置文件修改强制写到硬盘中。

aof设置：

appendonly yes //启动aof持久化方式有三种修改方式

#appendfsync always //收到写命令就立即写入到磁盘

#appendfsync everysec//每秒钟写入磁盘一次，在性能和持久化方面做了很好的折中

#appendfsync no //完全依赖os性能最好持久化没保证

|  |
| --- |
| redis配置文件配置：  appendonly yes 需要将appendonly设置成yes  # The name of the append only file (default: "appendonly.aof")  appendfilename "appendonly.aof" 将日志输出到appendonly.aof中 |

## 15.16 Redis发布订阅

|  |
| --- |
| 多个客户端可以通过subscribe [频道] 进行订阅  服务端可以通过publish [频道] 发布的内容进行发布消息  127.0.0.1:6379> publish cmcc hello  (integer) 0  127.0.0.1:6379> subscribe cmcc  Reading messages... (press Ctrl-C to quit)  1) "subscribe"  2) "cmcc"  3) (integer) 1  1) "message"  2) "cmcc"  3) "hello" |

## 15.17 虚拟内存的使用

redis暂时会把不经常访问的数据从内存交换到磁盘中，腾出宝贵的空间，用于其他需要范围的数据，这需要对vm相关进行配置。（3.0版本是不带vm特性的配置无效）

|  |
| --- |
| 修改配置文件：redis.conf  vm-enabled yes #开启vm功能  vm-swap-file /temp/redis.swap #交换处理的value保存的文件路径  vm-max-mernory 1000000 #redis使用的最大内存上限  vm-page-size 32 #每个页面的大小为32个字节  vm-max-threads 4 #用于执行value对象还如缓存的工作线程数量 |

## 15.18 集群搭建

redis3.0以前，提供了Sentine工具来监控Master的状态，如果Master异常，则会做主从切换，将salve作为master，将master作为salve。其配置也是稍微的负责，并且各方面表现一般。现在redis3.0已经支持集群的容错功能，并且非常简单。集群搭建至少需要三个master

|  |  |
| --- | --- |
| 第一步：创建一个文件夹 redis-cluster 然后在其下面分别创建6个文件夹如下：  1、midkr -p mkdir -p /etc/redis/redis-cluster  2、cd /redis-cluster  3、mkdir 7001、mkdir 7002、 mkdir 7003、 mkdir 7004、mkdir 7005、mkdir 7006  第二步：把之前的redis.conf配置文件分别copy到700\*下，进行修改各个文件内容，也就是对700\*的每一个copy的redis.conf文件进行修改！如下：  1、daemonize yes (后台启动)  2、port 700\* （分别对每个机器的端口号进行设置）  3、bind 192.168.1.149 （必须绑定当前机器的ip，不然入坑）  4、dir /etc/redis/redis-cluster/700\*(指定数据文件存放位置，必须要指定不同的目录位置，不然会丢失数据)  5、cluster-enabled yes 启动集群模式  6、cluster-config-file nodes 700\*.conf(这里700\*最好和port对应上)  7、cluster-node-timeout 5000  8、appendonly yes  第三步：把修改后的配置文件，分别拷贝到各个文件夹下，注意每个文件夹修改端口好，并且nodes文件也要不相同  第四步：由于redis集群需要使用ruby命令，所以我们需要安装ruby  1、yum install ruby  2、yum install rubygems  3、yum install redis（安装redis和ruby的接口）  第五步：分别启动6个redis实例，然后槛车是否启动成功  1、redis-server /etc/redis/700\*/redis.conf  2、ps -el|grep redis 查看是否启动成功  3、netstat -tunpl |grep redis 监控redis  第六步：首先到redis3.0的安装目录下，然后执行redis-trib.rb命令  1、cd /sortware/redis-4.0.6/src  2、./redis-trib.rb create --replicas 1 192.168.1.149:7001 192.168.1.149:7002 192.168.1.149:7003 192.168.1.149:7004 192.168.1.149:7005 192.168.1.149:7006  注意：如果在配置文件中设置了密码需要注释掉否则入坑  如果启动出现错误如果是由于ruby版本过老的话需要重新安装ruby，安装说明详情查看  <https://www.cnblogs.com/carryping/p/7447823.html>  第七步：到此为止集群搭建成功进行验证：  1、连接任意一个客户端即可：./redis-cli -c -h -p(-c 表示集群模式，指定ip地址和端口号)如：  /redis-cli -c -h 127.0.0.1 -p 700\*   |  | | --- | | [root@localhost redis-cluster]# redis-cli -c -h 127.0.0.1 -p 7001  127.0.0.1:7001> keys \*  (empty list or set)  127.0.0.1:7001> set a 123  -> Redirected to slot [15495] located at 127.0.0.1:7003  OK |   2、进行验证：cluster info（查看集群信息）、cluster nodes（查看节点列表）  3、进行数据操作验证  4、关闭集群则需要逐个关闭使用命令：  /redis-cli -c -h 127.0.0.1 -p 700\* shutdown |

## 15.19 集群操作

### 15.19.1 添加节点（一个master 一个slave）

|  |
| --- |
| 1.创建7007与7008两个文件夹  2.将7006中的redis.conf拷贝到7007与7008目录下并且修改对应的Ip  3.修改内容如下：  port:7007/7008  dir /ect/redis/redis-cluster/7007/(7008/)  cluster-config-file nodes 7007(7008).conf |

### 15.19.2 ./redis-trib.rb 中的操作指令

|  |
| --- |
| 1. create ：创建一个集群环境 host1:port1 .... hostN:portN  2.cail ：可以执行redis命令  3.add-node ：将一个节点添加到集群中，第一个参数为新节点的IP，第二个参数为急群众任意一个已经存在的节点port （ip:port）  4.del-node ：删除一个节点  5.reshard ：重新分片  6. check ：检查集群 |

### 15.19.3 新增一个主节点 7007

|  |
| --- |
| 1.使用add-node命令：绿色问新增的节点，红色为已经存在的节点  [root@localhost redis-cluster]# ./redis-trib.rb add-node 192.168.1.149:7007 192.168.1.149:7001  >>> Adding node 192.168.1.149:7007 to cluster 192.168.1.149:7001  >>> Performing Cluster Check (using node 192.168.1.149:7001)  >>> Send CLUSTER MEET to node 192.168.1.149:7007 to make it join the cluster.  [OK] New node added correctly.  查看7007是否已经添加成功  redis-cli -c -h 192.168.1.149 -p 7007  输入 cluster nodes 查看节点信息。  2.为7007分配slot槽 使用redis-trib命令，找到集群中的任意一个主节点（红色位置表现急群众的任意一个主节点对其进行重新分片工作）  ./redis-trib.rb reshard 192.168.1.149:7001  How many slots do you want to move (from 1 to 16384)? 500 分配的槽数量  What is the receiving node ID? f8b5e53085f40ba2e39d96c77bc2f77a2e0ef939 分配槽的ID  Please enter all the source node IDs.  Type 'all' to use all the nodes as source nodes for the hash slots. 所有节点合力分配  Type 'done' once you entered all the source nodes IDs. 只通过当前选择的节点来分配  3.需要执行replicate命令来指定当前节点（从节点）首先需要登录新加的几点7008节点的客户端，然后使用集群命令进行操作，把当前的7008（slave）节点指定到下一个主节点下（这里使用之前创建的7007主节点，红色表示节点Id）  首先需要将7008加入集群 ./redis-trib.rb add-node 192.168.1.149:7008 192.168.1.149:7001  [root@localhost redis-cluster]# redis-cli -c -h 192.168.1.149 -p 7008  192.168.1.149:7008> cluster replicate f8b5e53085f40ba2e39d96c77bc2f77a2e0ef939 （id是7007的id）  OK |

### 15.19.4 移除一个节点

|  |
| --- |
| 删除一个节点7008 slave,输入del-node命令，指定删除节点ip和端口，以及节点id（红色为7008节点id）  ./redis-trib.rb del-node 192.168.1.149:7008 8da2abf5bd7f9f6ac182bbe8c0a7989e6f906e50 |

### 15.19.5 主节点移除

|  |
| --- |
| 1. 删除7007 mastsr 主节点之前，我们需要先把其全部的数据（slot槽）移动到其它节点上去。否则会出现数据丢失。目前删除节点的槽无法做到平均分配，只能分配到一个节点上去。   [root@localhost redis-cluster]#./redis-trib.rb reshard 192.168.1.149:7007  M: f8b5e53085f40ba2e39d96c77bc2f77a2e0ef939 192.168.1.149:7007  slots:0-165,5461-5627,10923-11088 (499 slots) master  How many slots do you want to move (from 1 to 16384)? 499将所有的槽还回去  What is the receiving node ID? 306409c0695403d4f45ffb7c74d55d2b2913db49  Please enter all the source node IDs.  Type 'all' to use all the nodes as source nodes for the hash slots.  Type 'done' once you entered all the source nodes IDs.  输入返还的节点（7007）id 306409c0695403d4f45ffb7c74d55d2b2913db49  输入done否则会入坑！  下一步就可以通过按照 15.19.4的操作将节点进行删除 |

## 15.20 Redis 问题

### 15.20.1.在redis中遇到存储较慢的情况：

原因是因为redis开启了AOF记录日志，为了体现redis的高可靠性。多线程并发的情况下去访问redis 写的性能 也相对来说降低了

### 15.20.2 redis4.0.6版本需要使用Jedis2.8.1

在使用的过程中redis服务器使用了4.0.6的版本，如果jedis使用的版本老了会出现无法连接的情况，版本过于新的话也会出现各种错误，建议使用dedis2.8.1

# Java&redis

Jedis就是redis支持java的第三方类库，我们可以使用Jedis类库操作redis数据库。大体上在3.0之前我们使用Jedis数据库的api比较全面，但是目前java第三方可用库更新比较

## 16.1 需要用到的jar包

|  |
| --- |
| <!-- java使用redis 需要的jar-->  <!-- https://mvnrepository.com/artifact/redis.clients/jedis -->  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>2.7.2</version>  </dependency> |

## 16.2 连接案例（单节点）

|  |
| --- |
| // 创建单个节点  Jedis jedis = new Jedis("192.168.1.149", 6379);  jedis.auth("123456");// 设置密码  String a = jedis.get("aaa");  System.out.println(a); |

## 16.3 创建多个分片

|  |
| --- |
| //主从，哨兵 使用  private static ShardedJedis shard;  //连接池  private static ShardedJedisPool pool;  @BeforeClass  public static void setUpBeforeClass() {  // 分片  List<JedisShardInfo> shards = Arrays.asList(new JedisShardInfo("192.168.1.159", 6379));  shard = new ShardedJedis(shards);  GenericObjectPoolConfig goConfig = new GenericObjectPoolConfig();  goConfig.setMaxTotal(100);  goConfig.setMaxIdle(20);  goConfig.setMaxWaitMillis(-1);  goConfig.setTestOnBorrow(true);  pool = new ShardedJedisPool(goConfig, shards);  } |

## 16.4 Spring-data-redis 整合

Spring-data-redis集群配置

|  |
| --- |
| <!-- 加载Properties文件 -->  <context:property-placeholder location="classpath:\*.properties"  ignore-unresolvable="true" />  <!-- 配置扫描包 -->  <context:component-scan base-package="com.java.\*,org.redis.\*" />  <!-- 配置集群 -->  <bean id="redisClusterConfiguration"  class="org.springframework.data.redis.connection.RedisClusterConfiguration">  <property name="maxRedirects" value="3"></property>  <!-- 节点 配置 -->  <property name="clusterNodes">  <set>  <bean class="org.springframework.data.redis.connection.RedisClusterNode">  <constructor-arg name="host" value="192.168.1.149" />  <constructor-arg name="port" value="7001" />  </bean>  <bean class="org.springframework.data.redis.connection.RedisClusterNode">  <constructor-arg name="host" value="192.168.1.149" />  <constructor-arg name="port" value="7002" />  </bean>  <bean class="org.springframework.data.redis.connection.RedisClusterNode">  <constructor-arg name="host" value="192.168.1.149" />  <constructor-arg name="port" value="7003" />  </bean>  <bean class="org.springframework.data.redis.connection.RedisClusterNode">  <constructor-arg name="host" value="192.168.1.149" />  <constructor-arg name="port" value="7004" />  </bean>  <bean class="org.springframework.data.redis.connection.RedisClusterNode">  <constructor-arg name="host" value="192.168.1.149" />  <constructor-arg name="port" value="7005" />  </bean>  <bean class="org.springframework.data.redis.connection.RedisClusterNode">  <constructor-arg name="host" value="192.168.1.149" />  <constructor-arg name="port" value="7006" />  </bean>  </set>  </property>  </bean>  <bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">  <property name="maxIdle" value="20" />  <property name="maxTotal" value="100" />  <property name="maxWaitMillis" value="6000" />  <property name="testOnBorrow" value="true" />  </bean>  <!--<bean id="jedisPoolConfig" class="redis.clients.jedis.JedisPoolConfig">  <property name="maxIdle" value="${redis.maxIdle}" /> <property name="maxTotal"  value="${redis.maxActive}" /> <property name="maxWaitMillis" value="${redis.maxWait}"  /> <property name="testOnBorrow" value="${redis.testOnBorrow}" /> </bean> -->    <!-- 配置redis连接 -->  <bean id="jeidsConnectionFactory"  class="org.springframework.data.redis.connection.jedis.JedisConnectionFactory">  <constructor-arg ref="redisClusterConfiguration" />  <constructor-arg ref="jedisPoolConfig" />  </bean>  <!-- redis Template -->  <bean id="redisTemplate" class="org.springframework.data.redis.core.RedisTemplate">  <property name="connectionFactory" ref="jeidsConnectionFactory"></property>  <property name="keySerializer">  <bean  class="org.springframework.data.redis.serializer.StringRedisSerializer" />  </property>  <property name="valueSerializer">  <bean  class="org.springframework.data.redis.serializer.StringRedisSerializer" />  </property>  <property name="hashKeySerializer">  <bean  class="org.springframework.data.redis.serializer.StringRedisSerializer" />  </property>  <property name="hashValueSerializer">  <bean  class="org.springframework.data.redis.serializer.StringRedisSerializer" />  </property>  </bean> |

Test

|  |
| --- |
| @RunWith(SpringJUnit4ClassRunner.class)  @ContextConfiguration(locations="classpath:applicationContext-context.xml")  public class SpringDataRedisTest {  @Autowired  private RedisTemplate redisTemplate;  @Autowired  private RedisServiceImpl redisService ;  @Test  public void test1(){  //redisTemplate.opsForValue().set("studentname", "jay");  String name = (String) redisTemplate.opsForValue().get("age");  System.out.println(name);  }  } |