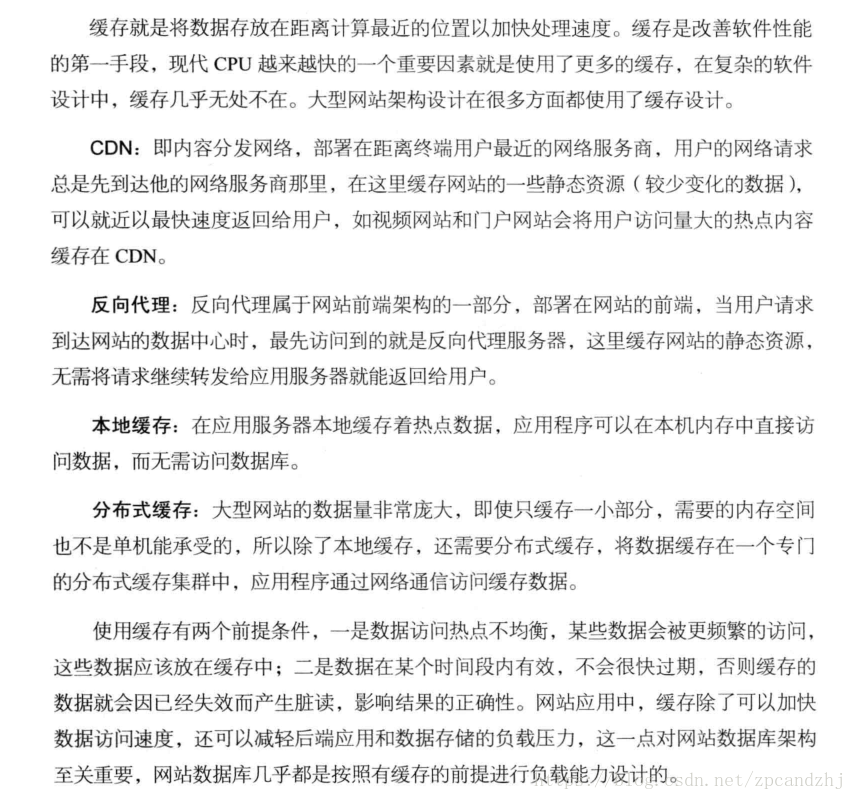
1.什么是缓存

解决方案：使用缓存。

1.1目前缓存的主流技术

1、Redis

2、Memcached

二者区别：

1、Memcache是多线程

2、Redis是单线程

1. redis用途

缓存 分布式集群架构中的session分离 任务队列(秒杀抢购) 应用排行榜 网站访问统计 数据过期处理等

2.11.Redis的基本命令

2.11.1.KEYS

exits key检测指定key是否存在，返回1表示存在，0不存在

del key1key2…keyN删除给定key,返回删除key的数目，0表示给定key都不存在

type key 返回给定 key 值的类型。返回 none 表示 key不存在,string,list,set,zset

keys pattern 返回匹配指定模式的所有 key。如：keys \* 返回所有key

randomkey 返回从当前数据库中随机选择的一个 key,如果当前数据库是空的，返回空串 rename oldkey

newkey 重命名一个 key,如果 newkey 存在，将会被覆盖，返回 1 表示成功， 0 失败。可能是 oldkey 不存在或者和 newkey 相同。

renamenx oldkey newkey 同上，但是如果 newkey 存在返回失败。

expire key seconds 为 key 指定过期时间，单位是秒。返回 1 成功，0 表示 key 已经设置过过 期时间或者不存在。

ttl key 返回设置过过期时间key的剩余过期秒数。-1表示key不存在或者未设置过期时间。 select db-index 通过索引选择数据库，默认连接的数据库是 0,默认数据库数是 16 个。返回 1 表示成功，0 失败。

Move key db-index将key从当前数据库移动到指定数据库。返回 1表示成功。0表示key 不存在或者已经在指定数据库中 。

3.Redis的字符串数据类型(String)

3.1.字符串类型

下表列出了一些用于在Redis中管理字符串的基本命令。

编号 命令 描述说明

1 SET key value 此命令设置指定键的值。

2 GET key 获取指定键的值。

3 GETRANGE key start end 获取存储在键上的字符串的子字符串。

4 GETSET key value 设置键的字符串值并返回其旧值。

5 GETBIT key offset 返回在键处存储的字符串值中偏移处的位值。

6 MGET key1 [key2…] 获取所有给定键的值

7 SETBIT key offset value 存储在键上的字符串值中设置或清除偏移处的位

8 SETEX key seconds value 使用键和到期时间来设置值

9 SETNX key value 设置键的值，仅当键不存在时

10 SETRANGE key offset value 在指定偏移处开始的键处覆盖字符串的一部分

11 STRLEN key 获取存储在键中的值的长度

12 MSET key value [key value …] 为多个键分别设置它们的值

13 MSETNX key value [key value …] 为多个键分别设置它们的值，仅当键不存在时

14 PSETEX key milliseconds value 设置键的值和到期时间(以毫秒为单位)

15 INCR key 将键的整数值增加1

16 INCRBY key increment 将键的整数值按给定的数值增加

17 INCRBYFLOAT key increment 将键的浮点值按给定的数值增加

18 DECR key 将键的整数值减1

19 DECRBY key decrement 按给定数值减少键的整数值

20 APPEND key value 将指定值附加到键

4. 列表类型(List)

Redis列表是链表是双向的，按照插入顺序排序。你可以添加一个元素到列表的头部（左边）或者尾部（右边）,一个列表最多可以包含 232 - 1 个元素 (4294967295, 每个列表超过40亿个元素)。

下表列出了列表相关的基本命令：

序号 命令及描述

1 BLPOP key1 [key2 ] timeout

移出并获取列表的第一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

2 BRPOP key1 [key2 ] timeout

移出并获取列表的最后一个元素， 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

3 BRPOPLPUSH source destination timeout

从列表中弹出一个值，将弹出的元素插入到另外一个列表中并返回它； 如果列表没有元素会阻塞列表直到等待超时或发现可弹出元素为止。

4 LINDEX key index

通过索引获取列表中的元素

5 LINSERT key BEFORE|AFTER pivot value

在列表的元素前或者后插入元素

6 LLEN key

获取列表长度

7 LPOP key

移出并获取列表的第一个元素

8 LPUSH key value1 [value2]

将一个或多个值插入到列表头部

9 LPUSHX key value

将一个值插入到已存在的列表头部

10 LRANGE key start stop

获取列表指定范围内的元素

11 LREM key count value

移除列表元素

12 LSET key index value

通过索引设置列表元素的值

13 LTRIM key start stop

对一个列表进行修剪(trim)，就是说，让列表只保留指定区间内的元素，不在指定区间之内的元素都将被删除。

14 RPOP key

移除并获取列表最后一个元素

15 RPOPLPUSH source destination

移除列表的最后一个元素，并将该元素添加到另一个列表并返回

16 RPUSH key value1 [value2]

在列表中添加一个或多个值

17 RPUSHX key value

为已存在的列表添加值

5.Redis 集合(Set)

5.1.set

Redis的Set一个哈希表结构，它的内部会根据 hash 分子来存储和查找数据,Redis 中所以添加，删除，查找的复杂度都是O(1)。集合元素是不重复的，是无序的，集合的每个元素都是stirng结构类型。

集合中最大的成员数为 232 - 1 (4294967295, 每个集合可存储40多亿个成员)。

下表列出了 Redis 集合基本命令：

序号 命令及描述

1 SADD key member1 [member2] 向集合添加一个或多个成员

2 SCARD key 获取集合的成员数

3 SDIFF key1 [key2] 返回给定所有集合的差集

4 SDIFFSTORE destination key1 [key2] 返回给定所有集合的差集并存储在 destination 中

5 SINTER key1 [key2] 返回给定所有集合的交集

6 SINTERSTORE destination key1 [key2] 返回给定所有集合的交集并存储在 destination 中

7 SISMEMBER key member 判断 member 元素是否是集合 key 的成员

8 SMEMBERS key 返回集合中的所有成员

9 SMOVE source destination member 将 member 元素从 source 集合移动到 destination 集合

10 SPOP key 移除并返回集合中的一个随机元素

11 SRANDMEMBER key [count] 返回集合中一个或多个随机数

12 SREM key member1 [member2] 移除集合中一个或多个成员

13 SUNION key1 [key2] 返回所有给定集合的并集

14 SUNIONSTORE destination key1 [key2] 所有给定集合的并集存储在 destination 集合中

15 SSCAN key cursor [MATCH pattern] [COUNT count] 迭代集合中的元素

5.2.zset有序集合(sorted set)

Redis 有序集合和集合一样也是string类型元素的集合,且不允许重复的成员。不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。

有序集合的成员是唯一的,但分数(score)却可以重复。

集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。 集合中最大的成员数为 232 - 1 (4294967295, 每个集合可存储40多亿个成员)。

下表列出了 redis 有序集合的基本命令:

序号 命令及描述

1 ZADD key score1 member1 [score2 member2] 向有序集合添加一个或多个成员，或者更新已存在成员的分数

2 ZCARD key 获取有序集合的成员数ZCOUNT key min max min 为最小值，max 为最大值，默认为包含 min 和 max 值，采用数学区间表示的方法，如果需要不包含，则在分数前面加入“(”，注意不支持“[”表示

4 ZINCRBY key increment member 有序集合中对指定成员的分数加上增量 increment

5 ZINTERSTORE destination numkeys key [key …]

计算给定的一个或多个有序集的交集并将结果集存储在新的有序集合 key 中

6 ZLEXCOUNT key min max

在有序集合中计算指定字典区间内成员数量，“[”表示包含该值，“(”表示不包含该值

7 ZRANGE key start stop [WITHSCORES]

通过索引区间返回有序集合成指定区间内的成员，包含start和stop

8 ZRANGEBYLEX key min max [LIMIT offset count]

通过字典区间返回有序集合的成员

9 ZRANGEBYSCORE key min max [WITHSCORES] [LIMIT]

通过分数返回有序集合指定区间内的成员

10 ZRANK key member

返回有序集合中指定成员的索引

11 ZREM key member [member …]

移除有序集合中的一个或多个成员

12 ZREMRANGEBYLEX key min max

移除有序集合中给定的字典区间的所有成员

13 ZREMRANGEBYRANK key start stop

移除有序集合中给定的排名区间的所有成员

14 ZREMRANGEBYSCORE key min max

移除有序集合中给定的分数区间的所有成员

15 ZREVRANGE key start stop [WITHSCORES]

返回有序集中指定区间内的成员，通过索引，分数从高到底

16 ZREVRANGEBYSCORE key max min [WITHSCORES]

返回有序集中指定分数区间内的成员，分数从高到低排序

17 ZREVRANK key member

返回有序集合中指定成员的排名，有序集成员按分数值递减(从大到小)排序

18 ZSCORE key member

返回有序集中，成员的分数值

19 ZUNIONSTORE destination numkeys key [key …]

计算给定的一个或多个有序集的并集，并存储在新的 key 中

20 ZSCAN key cursor [MATCH pattern] [COUNT count]

迭代有序集合中的元素（包括元素成员和元素分值）

6.Redis的Hash数据结构

Redis hash 是一个string类型的field和value的映射表，hash特别适合用于存储对象。

Redis 中每个 hash 可以存储 232 - 1 键值对（40多亿）。

下表列出了 redis hash 基本的相关命令：

序号 命令及描述

1 HDEL key field2 [field2]

删除一个或多个哈希表字段

2 HEXISTS key field

查看哈希表 key 中，指定的字段是否存在。

3 HGET key field

获取存储在哈希表中指定字段的值。

4 HGETALL key

获取在哈希表中指定 key 的所有字段和值

5 HINCRBY key field increment

为哈希表 key 中的指定字段的整数值加上增量 increment 。

6 HINCRBYFLOAT key field increment

为哈希表 key 中的指定字段的浮点数值加上增量 increment 。

7 HKEYS key

获取所有哈希表中的字段

8 HLEN key

获取哈希表中字段的数量

9 HMGET key field1 [field2]

获取所有给定字段的值

10 HMSET key field1 value1 [field2 value2 ]

同时将多个 field-value (域-值)对设置到哈希表 key 中。

11 HSET key field value

将哈希表 key 中的字段 field 的值设为 value 。

12 HSETNX key field value

只有在字段 field 不存在时，设置哈希表字段的值。

13 HVALS key

获取哈希表中所有值

14 HSCAN key cursor [MATCH pattern] [COUNT count]

迭代哈希表中的键值对。

8.Redis事务

8.1 Redis事务

Redis 事务可以一次执行多个命令， 并且带有以下两个重要的保证：

事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。

事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。

一个事务从开始到执行会经历以下三个阶段：

开始事务。

命令入队。

执行事务。

下表列出了 redis 事务的相关命令：

序号 命令及描述

1 DISCARD

取消事务，放弃执行事务块内的所有命令。

2 EXEC

执行所有事务块内的命令。

3 MULTI

标记一个事务块的开始。

4 UNWATCH

取消 WATCH 命令对所有 key 的监视。

5 WATCH key [key …]

监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将会回滚。使用乐观锁机制

127.0.0.1:6379> watch book

OK

127.0.0.1:6379> multi

OK

127.0.0.1:6379> set book wuwen //另一个客户端set book tiyu，watch的key发生变化

QUEUED

127.0.0.1:6379> set color blue

QUEUED

127.0.0.1:6379> exec

(nil)

8.2 事务会滚

8.2.1 操作的数据类型错误会滚，发生错误的命令回滚，前后都回执行，只是和数据库不同的地方。

127.0.0.1:6379> multi

OK

127.0.0.1:6379> set book english

QUEUED

127.0.0.1:6379> incr book

QUEUED

127.0.0.1:6379> set color yellow

QUEUED

127.0.0.1:6379> exec

1) OK

2) (error) ERR value is not an integer or out of range

3) OK

127.0.0.1:6379> get book

"english"

127.0.0.1:6379> get color

"yellow"

8.2.2 操作的命令格式错误回滚，前后的操作都回滚。

127.0.0.1:6379> multi

OK

127.0.0.1:6379> set age 25

QUEUED

127.0.0.1:6379> incr

(error) ERR wrong number of arguments for 'incr' command

127.0.0.1:6379> set sex boy

QUEUED

127.0.0.1:6379> exec

(error) EXECABORT Transaction discarded because of previous errors.

127.0.0.1:6379> get age

(nil)

127.0.0.1:6379> get sex

(nil)

9.redis的流水线

在事务中 Redis 提供了队列，这是一个可以批量执行任务的队列，这样性能就比较高，但是使用 multi...exec 事务命令是有系统开销的，因为它会检测对应的锁和序列化命令。  
有时候我们希望在没有任何附加条件的场景下去使用队列批量执行一系列的命令，从而提高系统性能，这就是 Redis 的流水线（pipelined）技术

Java代码测试，30万条数据只要1秒多

JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();  
jedisPoolConfig.setMaxIdle(50);  
jedisPoolConfig.setMaxTotal(100);  
jedisPoolConfig.setMaxWaitMillis(2000);  
JedisPool jedisPool = new JedisPool(jedisPoolConfig,"localhost");  
Jedis jedis = jedisPool.getResource();  
long a = System.*currentTimeMillis*();  
Pipeline pipelined = jedis.pipelined();  
for (int i=0;i<300000;i++){  
 pipelined.set("name"+i,i+"");  
}  
pipelined.sync();  
long b = System.*currentTimeMillis*();  
System.*out*.println("流水线时间:"+(b-a));  
jedis.close();

9.Redis发布订阅模式

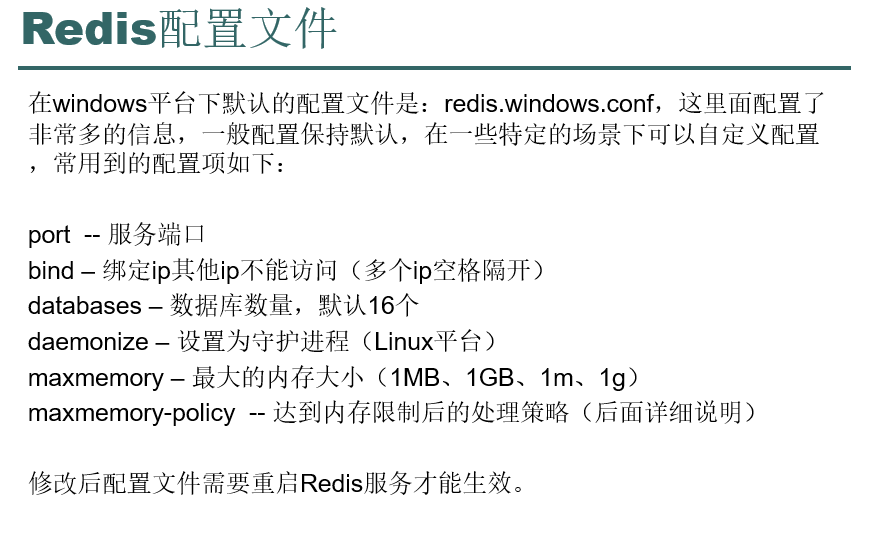
向通道发布消息，其他订阅了该通道的客户端接收到消息

命令参数如下：

SUBSCRIBE chat 客户端订阅chat通道

publish chat "let's go!!" 像chat通道发送消息

spring的例子见我的另一个博客。

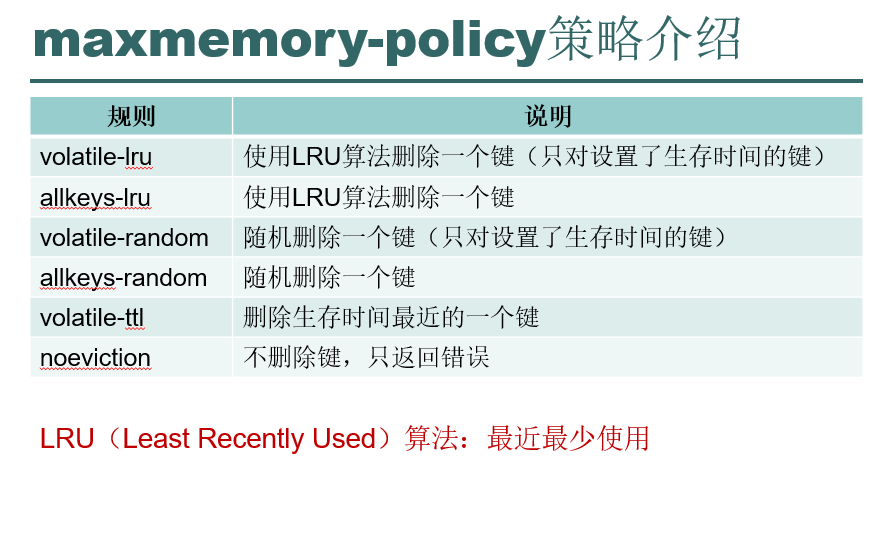


10. redis的垃圾回收策略

当键超时的时候，正如 [Java](http://c.biancheng.net/java/) 虚拟机，它提供了自动 GC（垃圾回收）的功能，

Redis 提供两种方式回收这些超时键值对，它们是定时回收和惰性回收。

* 定时回收是指在确定的某个时间触发一段代码，回收超时的键值对。数据量太大可能影响redis性能。
* 惰性回收则是当一个超时的键，被再次用 get 命令访问时，将触发 Redis 将其从内存中清空。

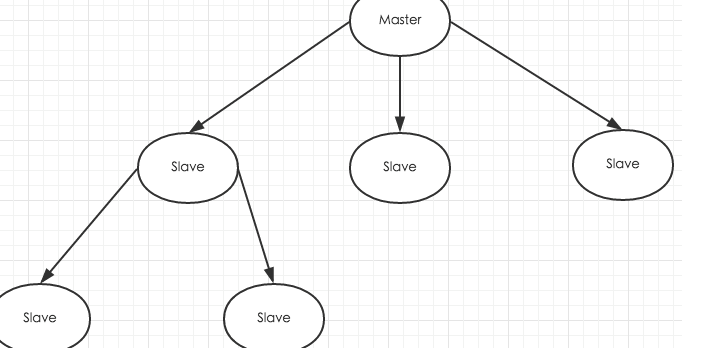
在redis.conf 里面有个配置策略 maxmemory-policy ，它有几个可选值：

11. redis的主从复制

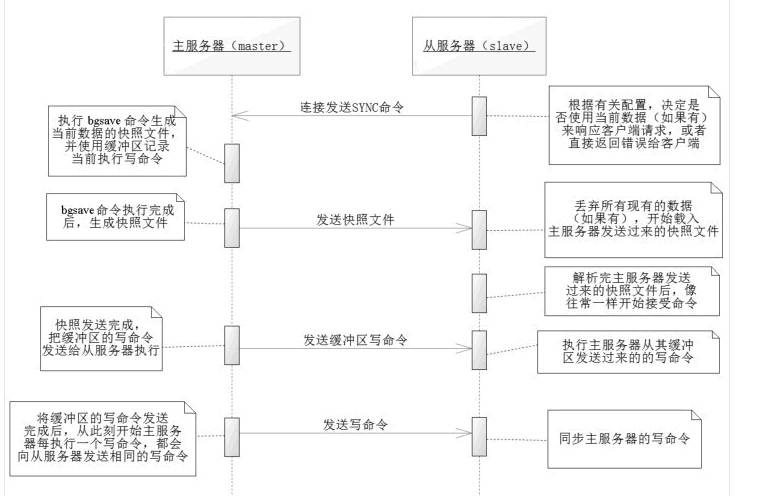
和Mysql主从复制的原因一样，Redis虽然读取写入的速度都特别快，但是也会产生读压力特别大的情况。为了分担读压力，Redis支持主从复制，主服务器负责写操作，从服务器负责读操作，redis使用异步复制，不影响客户端的读写操作。

Redis主从复制可以根据是否是全量分为全量同步和增量同步。

全量同步：salve初始化阶段，发送sync全量同步，增量同步：salve正常工作之后，值同步master的写操作，slave 在任何时候都可以发起全量同步。redis 策略是，无论如何，首先会尝试进行增量同步，如不成功，要求从机进行全量同步。Slave掉线之后，重启之后会请求全量同步。



11.1主从同步步骤



1）无论如何要先保证主服务器的开启，开启主服务器后，从服务器通过命令或者重启配置项可以同步到主服务器。  
2）当从服务器启动时，读取同步的配置，根据配置决定是否使用当前数据响应客户端，然后发送 SYNC 命令。  
当主服务器接收到同步命令的时候，就会执行 bgsave 命令备份数据，但是主服务器并不会拒绝客户端的读/写，而是将来自客户端的写命令写入缓冲区。从服务器未收到主服务器备份的快照文件的时候，会根据其配置决定使用现有数据响应客户端或者拒绝。  
3）当 bgsave 命令被主服务器执行完后，开始向从服务器发送备份文件，这个时候从服务器就会丢弃所有现有的数据，开始载入发送的快照文件。  
4）当主服务器发送完备份文件后，从服务器就会执行这些写入命令。此时就会把 bgsave 执行之后的缓存区内的写命令也发送给从服务器，从服务完成备份文件解析，就开始像往常一样，接收命令，等待命令写入。  
5）缓冲区的命令发送完成后，当主服务器执行一条写命令后，就同时往从服务器发送同步写入命令，从服务器就和主服务器保持一致了。而此时当从服务器完成主服务器发送的缓冲区命令后，就开始等待主服务器的命令了。

11.3 linux环境搭建主从复制、

1）修改redis.conf 中的配置

bind 0.0.0.0#任意ip都可以连接

protected-mode 为no不要保护模式

2）拷贝住服务器的配置，称为两个从服务器配置

cp redis.conf redis-slvae6481.conf

cp redis.conf redis-slvae6482.conf

3)修改slave从配置

修改slave从机的配置 指定主机ip端口replicaof 192.168.0.103 6379

4)启动主从服务器

[root@localhost /]# cd /usr/local/redis/bin 进入到redis的配置

[root@localhost /]# ./redis-server redis.conf 启动主服务器

[root@localhost /]# ./redis-server redis-slave6481.conf 启动从服务器

[root@localhost /]# ./redis-server redis-slave6481.conf启动从服务器

[root@localhost /]# ./redis-cli –p 6379 info replication 查看主从的情况

**12 redis持久化的两种策略，redis支持两种模式一起使用。**

12.1 RDB

对于快照模式的备份而言，它的配置项如下：

* save 900 1 当 900 秒执行 1 个写命令时，启用快照备份。  
  save 300 10 当 300 秒执行 10 个写命令时，启用快照备份。  
  save 60 10000 当 60 秒内执行 10000 个写命令时，启用快照备份。

**Redis实现快照的过程**  
-  Redis使用fork函数复制一份当前进程（父进程）的副本（子进程）；  
-  父进程继续接收并处理客户端发来的命令，而子进程开始将内存中的数据写入硬盘中的临时文件；  
-  当子进程写入完所有数据后会用该临时文件替换旧的RDB文件，至此一次快照操作完成。  
-  在执行fork的时候操作系统（类Unix操作系统）会使用写时复制（copy-on-write）策略，即fork函数发生的一刻父子进程共享同一内存数据，当父进程要更改其中某片数据时（如执行一个写命令 ），操作系统会将该片数据复制一份以保证子进程的数据不受影响，所以新的RDB文件存储的是执行fork一刻的内存数据。

除了自动快照，还可以手动发送SAVE或BGSAVE命令让Redis执行快照，两个命令的区别在于，前者是由主进程进行快照操作，会阻塞住其他请求，后者会通过fork子进程进行快照操作。

12.2 AOF

默认情况下Redis没有开启AOF(append only file)方式的持久化，可以在redis.conf中通过appendonly参数开启：

appendonly yes

配置redis自动重写AOF文件的条件

auto-aof-rewrite-percentage 100  # 当目前的AOF文件大小超过上一次重写时的AOF文件大小的百分之多少时会再次进行重写，如果之前没有重写过，则以启动时的AOF文件大小为依据

auto-aof-rewrite-min-size 64mb   # 允许重写的最小AOF文件大小  
配置写入AOF文件后，要求系统刷新硬盘缓存的机制

# appendfsync always   # 每次执行写入都会执行同步，最安全也最慢  
appendfsync everysec   # 每秒执行一次同步操作

# appendfsync no       # 不主动进行同步操作，而是完全交由操作系统来做（即每30秒一次），最快也最不安全

12.3两种模式的优缺点

### redis两种持久化的方式

* RDB持久化可以在指定的时间间隔内生成数据集的时间点快照
* AOF持久化记录服务器执行的所有写操作命令,并在服务器启动时,通过重新执行这些命令来还原数据集,AOF文件中全部以redis协议的格式来保存,新命令会被追加到文件的末尾,redis还可以在后台对AOF文件进行重写,文件的体积不会超出保存数据集状态所需要的实际大小,
* redis还可以同时使用AOF持久化和RDB持久化,在这种情况下,当redis重启时,它会有限使用AOF文件来还原数据集,因为AOF文件保存的数据集通常比RDB文件所保存的数据集更加完

### ＲＤＢ的优点

1. RDB 是一个非常紧凑（compact）的文件，它保存了 Redis 在某个时间点上的数据集。 这种文件非常适合用于进行备份： 比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也备份一个 RDB 文件。 这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。
2. RDB 非常适用于灾难恢复（disaster recovery）：它只有一个文件，并且内容都非常紧凑，可以（在加密后）将它传送到别的数据中心，或者亚马逊 S3 中。
3. RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。
4. RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快

### ＲＤＢ的缺点

1. 如果你需要尽量避免在服务器故障时丢失数据，那么 RDB 不适合你。 虽然 Redis 允许你设置不同的保存点（save point）来控制保存 RDB 文件的频率， 但是， 因为RDB 文件需要保存整个数据集的状态， 所以它并不是一个轻松的操作。 因此你可能会至少 5 分钟才保存一次 RDB 文件。 在这种情况下， 一旦发生故障停机， 你就可能会丢失好几分钟的数据。

* 每次保存 RDB 的时候，Redis 都要 fork() 出一个子进程，并由子进程来进行实际的持久化工作。 在数据集比较庞大时， fork()可能会非常耗时，造成服务器在某某毫秒内停止处理客户端； 如果数据集非常巨大，并且 CPU 时间非常紧张的话，那么这种停止时间甚至可能会长达整整一秒。 虽然 AOF 重写也需要进行 fork() ，但无论 AOF 重写的执行间隔有多长，数据的耐久性都不会有任何损失。

### AOF 的优点

* 使用 AOF 持久化会让 Redis 变得非常耐久（much more durable）：你可以设置不同的 fsync 策略，比如无 fsync ，每秒钟一次 fsync ，或者每次执行写入命令时 fsync 。 AOF 的默认策略为每秒钟 fsync 一次，在这种配置下，Redis 仍然可以保持良好的性能，并且就算发生故障停机，也最多只会丢失一秒钟的数据（ fsync 会在后台线程执行，所以主线程可以继续努力地处理命令请求）。
* AOF 文件是一个只进行追加操作的日志文件（append only log）， 因此对 AOF 文件的写入不需要进行 seek ， 即使日志因为某些原因而包含了未写入完整的命令（比如写入时磁盘已满，写入中途停机，等等）， redis-check-aof 工具也可以轻易地修复这种问题。
* Redis 可以在 AOF 文件体积变得过大时，自动地在后台对 AOF 进行重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。 整个重写操作是绝对安全的，因为 Redis 在创建新 AOF 文件的过程中，会继续将命令追加到现有的 AOF 文件里面，即使重写过程中发生停机，现有的 AOF 文件也不会丢失。 而一旦新 AOF 文件创建完毕，Redis 就会从旧 AOF 文件切换到新 AOF 文件，并开始对新 AOF 文件进行追加操作。
* AOF 文件有序地保存了对数据库执行的所有写入操作， 这些写入操作以 Redis 协议的格式保存， 因此 AOF 文件的内容非常容易被人读懂， 对文件进行分析（parse）也很轻松。 导出（export） AOF 文件也非常简单： 举个例子， 如果你不小心执行了 FLUSHALL 命令， 但只要 AOF 文件未被重写， 那么只要停止服务器， 移除 AOF 文件末尾的 FLUSHALL 命令， 并重启 Redis ， 就可以将数据集恢复到 FLUSHALL 执行之前的状态。

### AOF 的缺点

* 对于相同的数据集来说，AOF 文件的体积通常要大于 RDB 文件的体积。
* 根据所使用的 fsync 策略，AOF 的速度可能会慢于 RDB 。 在一般情况下， 每秒 fsync 的性能依然非常高， 而关闭 fsync 可以让 AOF 的速度和 RDB 一样快， 即使在高负荷之下也是如此。 不过在处理巨大的写入载入时，RDB 可以提供更有保证的最大延迟时间（latency）。
* AOF 在过去曾经发生过这样的 bug ： 因为个别命令的原因，导致 AOF 文件在重新载入时，无法将数据集恢复成保存时的原样。 （举个例子，阻塞命令 BRPOPLPUSH 就曾经引起过这样的 bug 。） 测试套件里为这种情况添加了测试： 它们会自动生成随机的、复杂的数据集， 并通过重新载入这些数据来确保一切正常。 虽然这种 bug 在 AOF 文件中并不常见， 但是对比来说， RDB 几乎是不可能出现这种 bug 的

13.redis哨兵模式

13.1这里的哨兵有两个作用：

* 通过发送命令，让 Redis 服务器返回监测其运行状态，包括主服务器和从服务器。
* 当哨兵监测到 master 宕机，通过选举会自动将 slave 切换成 master，然后通过发布订阅模式通知到其他的从服务器，修改配置文件，让它们切换主机。

13.2 linux环境搭建

1）把哨兵配置文件copy到redis中，方便以后使用

[root@localhost redis-5.0.3]# cp sentinel.conf /usr/local/redis/bin

2）进入刚复制到的文件夹中

cd /usr/local/redis/bin

3）、：[root@localhost bin]# vi sentinel.conf

编辑如下键值对

　# bind 127.0.0.1 192.168.1.1#注释掉或者值为0.0.0.0

　　　　protected-mode no#关闭保护模式

　　　　port 26001#端口号 默认是26739

　　　　daemonize yes#后台运行

　　　　sentinel monitor mymaster 192.168.194.131 6381 1#设置 主名称 ip地址 端口号 参入选举的哨兵数

　　　　sentinel down-after-milliseconds mymaster 3000#sentinel心跳检测主3秒内无响应，视为挂掉，开始切换其他从为主

　　　　sentinel parallel-syncs mymaster 1#每次最多可以有1个从同步主。一个从同步结束，另一个从开始同步。

　　　　sentinel failover-timeout mymaster 18000#主从切换超时时间

4）测试

启动主从rdis，在启动哨兵

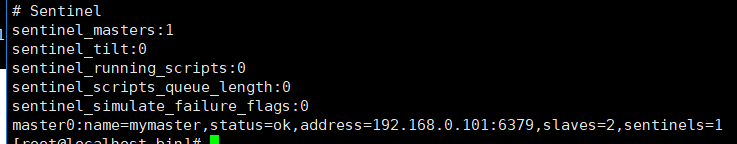
[root@localhost /]# ./redis-server redis.conf 启动主服务器

[root@localhost /]# ./redis-server redis-slave6481.conf 启动从服务器

[root@localhost /]# ./redis-server redis-slave6481.conf启动从服务器

[root@localhost bin]# ./redis-sentinel sentinel.conf 启动哨兵

[root@localhost /]# ./redis-cli –p 26379 info 查看哨兵的情况



5）设置哨兵自动启动（过程和redis自启动设置差不多）

**2.编辑start.sh**

**[root@localhost system]# cd /usr/local/redis/script/**

　　　　vi start.sh

增加以下：

/usr/local/redis/src/redis-sentinel /usr/local/redis /sentinel.conf

**3.编辑stop.sh**

　　　　vi stop.sh

增加以下：

/usr/local/redis/src/redis-cli -p 26739 shutdown

**4.编辑restart.sh**

　　　　vi restart.sh

增加以下：

systemctl stop redis-sentinel

systemctl start redis-sentinel

**5.编写redis-sentinel.service**

　　　　cd /usr/lib/systemd/system/

　　　　vi redis-sentinel.service

[Unit]

Description=redis-sentinel

After=redis.service

[Service]

Type=forking

ExecStart=/usr/local/redis/script/start.sh

ExecStop=/usr/local/redis/script/stop.sh

ExecReload=/usr/local/redis/script/restart.sh

[Install]

WantedBy=multi-user.target

　　　　#建议启动redis服务后，再启动哨兵

改变权限

　　　　　　chmod 777 redis-sentinel.service

　　　　　　chmod 777 /usr/local/redis/script/\*

　　　　进程服务重加载

　　　　　　systemctl daemon-reload

　　　　开机启动哨兵

　　　　　　systemctl enable redis-sentinel.service

　　　　启动哨兵

　　　　　　systemctl start redis-sentinel.service

　　　　关闭哨兵

　　　　　　systemctl stop redis-sentinel.service

　　　　重启哨兵

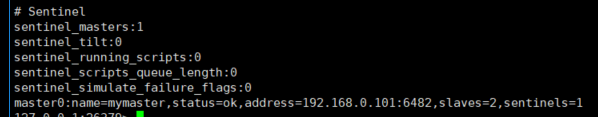
　　　　　　systemctl restart redis-sentinel.service

测试关闭redis主机：

./redis-cli –p 6379

Shutdown

可以看到sentinel的info显示选举了6482作为主机。



哨兵集群，和创建一个哨兵类似，这里就不写了。

14 cluster集群模式

14.1哨兵模式性能和高可用性等各方面表现一般，特别是在主从切换的瞬间存在访问瞬断的情况，而且哨兵模式只有一个主节点对外提供服务，没法支持很高的并发，且单个主节点内存也不宜设置得过大，否则会导致持久化文件过大，影响数据恢复或主从同步的效率。

cluster模式就支持动态扩容，可以在线增加或删除节点，而且客户端可以连接任何一个主节点进行读写，不过此时的从节点仅仅只是备份的作用。至于为何能做到动态扩容，主要是因为Redis集群没有使用一致性hash,而是使用的哈希槽。Redis集群会有16384个哈希槽，每个key通过CRC16校验后对16384取模来决定放置哪个槽，而集群的每个节点负责一部分hash槽。

**14.2，故障转移**

**1，故障转移机制详解**

集群中的节点会向其它节点发送PING消息（该PING消息会带着当前集群和节点的信息），如果在规定时间内，没有收到对应的PONG消息，就把此节点标记为疑似下线。当被分配了slot槽位的主节点中有超过一半的节点都认为此节点疑似下线（就是其它节点以更高的频次，更频繁的与该节点PING-PONG），那么该节点就真的下线。其它节点收到某节点已经下线的广播后，把自己内部的集群维护信息也修改为该节点已事实下线。

节点资格审查：然后对从节点进行资格审查，每个从节点检查最后与主节点的断线时间，如果该值超过配置文件的设置，那么取消该从节点的资格。准备选举时间：这里使用了延迟触发机制，主要是给那些延迟低的更高的优先级，延迟低的让它提前参与被选举，延迟高的让它靠后参与被选举。（延迟的高低是依据之前与主节点的最后断线时间确定的）

选举投票：当从节点获取选举资格后，会向其他带有slot槽位的主节点发起选举请求，由它们进行投票，优先级越高的从节点就越有可能成为主节点，当从节点获取的票数到达一定数值时（如集群内有N个主节点，那么只要有一个从节点获得了N/2+1的选票即认为胜出），就会替换成为主节点。

替换主节点：被选举出来的从节点会执行slaveof no one把自己的状态从slave变成master，然后执行clusterDelSlot操作撤销故障主节点负责的槽，并执行 clusterAddSlot把这些槽分配给自己，之后向集群广播自己的pong消息，通知集群内所有的节点，当前从节点已变为主节点。接管相关操作：新的主节点接管了之前故障的主节点的槽信息，接收和处理与自己槽位相关的命令请求。

14.3 linux环境搭建（三主三从）

1）local下创建6个文件夹，放置6个节点

[root@localhost local]# mkdir redis-cluster

2）redis-cluster下创建7001-7006等6个文件夹，放置节点

[root@localhost redis-cluster]# mkdir 7001 7002 7003 7004 7005 7006

3）把redis的redis.confcopy到7001到7006中

[root@localhost redis-cluster]# cp /usr/local/redis-5.0.3/redis.conf ./7001

。。。。。。。

[root@localhost redis-cluster]# cp /usr/local/redis-5.0.3/redis.conf ./7006

4）先重命名个文件，方便管理7001到7006都要

[root@localhost redis-cluster]# mv ./7002/redis.conf ./7002/redis-7002.conf

5）[root@localhost 7001]# vi redis-7001.conf

修改以下的配置：

bind  0.0.0.0   (0.0.0.0表示所有节点都可以访问该redis)

    protected-mode no

    daemonize yes (设置后台运行redis)

    cluster-enabled yes（开启集群，把#去掉）

    cluster-node-timeout 15000 (设置请求超时时间，默认为15秒，可以自行修改)

    appendonly yes （aop日志开启，会每次进行写操作都记录一条日志）

    --根据不同的端口需要设置的地方

        port 7000

        pidfile /var/run/redis\_7000.pid

        dbfilename dump\_7000.rdb

        appendfilename "appendonly\_7000.aof"

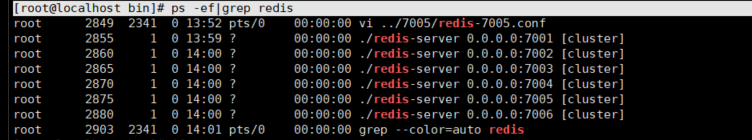
        cluster-config-file nodes\_7000.conf

6）启动6个实例，这里只写一个例子，其他同样方式启动

[root@localhost bin]# ./redis-server /usr/local/redis-cluster/7001/redis-7001.conf

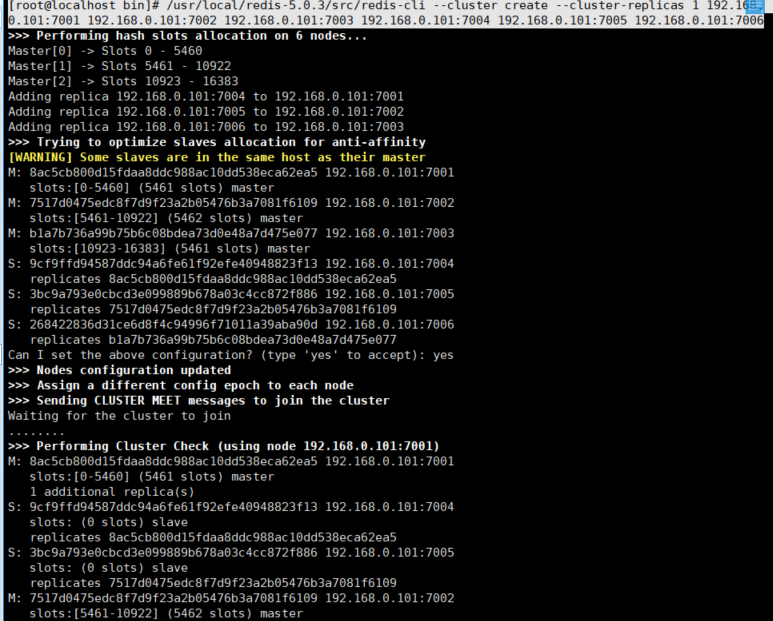
7）查看启动的redis

[root@localhost bin]# ps -ef|grep redis



8）用redis-cli创建整个redis集群(redis5以前的版本集群是依靠ruby脚本redis-trib.rb实现)

[root@localhost bin]# /usr/local/redis-5.0.3/src/redis-cli --cluster create --cluster-replicas 1 192.168.0.101:7001 192.168.0.101:7002 192.168.0.101:7003 192.168.0.101:7004 192.168.0.101:7005 192.168.0.101:7006

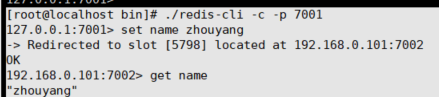


Can I set the above configuration? (type 'yes' to accept):yes代表为每个创建的主服务器节点创建一个从服务器节点

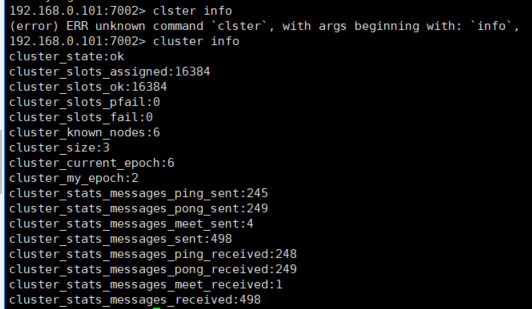
9）验证集群

随便连接那个redis.-c带便连接集群

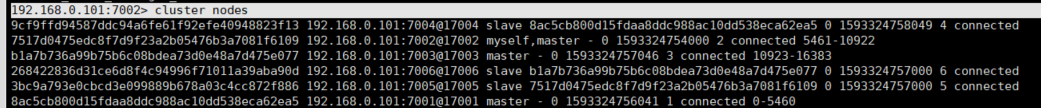
[root@localhost bin]# ./redis-cli -c -p 7001，这里可以看到7001中设置了name，其他节点都可以读取到name，集群搭建完成。



192.168.0.101:7002> cluster info 查看集群信息，



192.168.0.101:7002> cluster nodes,可以看到主从节点的关系



14.4 集群的水平扩展

1）新建8007 8008；两个文件夹，放置两个节点配置信息

[root@localhost redis-cluster]# mkdir 7007

[root@localhost redis-cluster]# mkdir 7008

[root@localhost redis-cluster]# cp 7006/redis-7006.conf 7008/redis-7008.conf

[root@localhost redis-cluster]# cp 7006/redis-7006.conf 7007/redis-7007.conf

2）修改这两个的配置文件，和上面差不多

[root@localhost redis-cluster]# vi 7007/redis-7007.conf

[root@localhost redis-cluster]# vi 7008/redis-7008.conf

bind  0.0.0.0   (0.0.0.0表示所有节点都可以访问该redis)

    protected-mode no

    daemonize yes (设置后台运行redis)

    cluster-enabled yes（开启集群，把#去掉）

    cluster-node-timeout 15000 (设置请求超时时间，默认为15秒，可以自行修改)

    appendonly yes （aop日志开启，会每次进行写操作都记录一条日志）

    --根据不同的端口需要设置的地方

        port 7000

        pidfile /var/run/redis\_7007.pid

        dbfilename dump\_7007.rdb

        appendfilename "appendonly\_7007.aof"

        cluster-config-file nodes\_7007.conf

3）启动这两个实例

[root@localhost bin]# ./redis-server /usr/local/redis-cluster/7007/redis-7007.conf

[root@localhost bin]# ./redis-server /usr/local/redis-cluster/7008/redis-7008.conf

查看7001的集群，发现并没有7007和7008加入

[root@localhost bin]# ./redis-cli -c –h 192.168.0.101 -p 7001

127.0.0.1:7001> cluster info

127.0.0.1:7001> cluster nodes

4)把7007加入到集群,把7007节点加入已知7001的集群中。加入了但是这是还没有分配槽点给7007

[root@localhost bin]# ./redis-cli --cluster add-node 192.168.0.101:7007 192.168.0.101:7001

》》查看当前集群节点，发现7007已经加入称为master

[root@localhost bin]# ./redis-cli -c -p 7001

127.0.0.1:7001> cluster nodes

9cf9ffd94587ddc94a6fe61f92efe40948823f13 192.168.0.101:7004@17004 slave 8ac5cb800d15fdaa8ddc988ac10dd538eca62ea5 0 1593332349511 4 connected

3bc9a793e0cbcd3e099889b678a03c4cc872f886 192.168.0.101:7005@17005 slave 7517d0475edc8f7d9f23a2b05476b3a7081f6109 0 1593332351522 5 connected

7517d0475edc8f7d9f23a2b05476b3a7081f6109 192.168.0.101:7002@17002 master - 0 1593332350000 2 connected 5461-10922

b1a7b736a99b75b6c08bdea73d0e48a7d475e077 192.168.0.101:7003@17003 master - 0 1593332350000 3 connected 10923-16383

f04b4ef9e33636f0cf5c0ad1d839300094c73012 192.168.0.101:7007@17007 master - 0 1593332352529 0 connected

268422836d31ce6d8f4c94996f71011a39aba90d 192.168.0.101:7006@17006 slave b1a7b736a99b75b6c08bdea73d0e48a7d475e077 0 1593332350517 6 connected

8ac5cb800d15fdaa8ddc988ac10dd538eca62ea5 192.168.0.101:7001@17001 myself,master - 0 1593332350000 1 connected 0-5460

5）为新节点手工分配hash槽

[root@localhost bin]# ./redis-cli --cluster reshard 192.168.0.101:7001

会出现询问分配多少槽点，给哪个节点分配？all是把集群原来配置好的借点平均分配给新节点，done是把指定的节点分配给新节点。

How many slots do you want to move (from 1 to 16384)? 1000

What is the receiving node ID? f04b4ef9e33636f0cf5c0ad1d839300094c73012

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.

Source node #1: 662809cf2d5bb138912dea7fb1e452f6e0f149da

Source node #2: done

6）查看分配好的集群信息，这样新加入的节点就分配好了槽点。

[root@localhost bin]# ./redis-cli -c -p 7007

127.0.0.1:7007> cluster nodes

7）把7008加入到集群并成为7007的从节点，这时就完成了集群的配置

加入到集群，这时是master节点：[root@localhost bin]# ./redis-cli --cluster add-node 192.168.0.101:7008 192.168.0.101:7007

打来7008客户端:[root@localhost bin]# ./redis-cli -c -p 7008

查看集群节点信息：127.0.0.1:7008> cluster nodes

551230eebef1f499c5962db11868cadb9b555b11 192.168.0.101:7008@17008 myself,master - 0 1593333269000 0 connected

b1a7b736a99b75b6c08bdea73d0e48a7d475e077 192.168.0.101:7003@17003 master - 0 1593333272319 3 connected 11256-16383

268422836d31ce6d8f4c94996f71011a39aba90d 192.168.0.101:7006@17006 slave b1a7b736a99b75b6c08bdea73d0e48a7d475e077 0 1593333271314 3 connected

9cf9ffd94587ddc94a6fe61f92efe40948823f13 192.168.0.101:7004@17004 slave 8ac5cb800d15fdaa8ddc988ac10dd538eca62ea5 0 1593333270000 1 connected

3bc9a793e0cbcd3e099889b678a03c4cc872f886 192.168.0.101:7005@17005 slave 7517d0475edc8f7d9f23a2b05476b3a7081f6109 0 1593333271000 2 connected

7517d0475edc8f7d9f23a2b05476b3a7081f6109 192.168.0.101:7002@17002 master - 0 1593333269303 2 connected 5795-10922

8ac5cb800d15fdaa8ddc988ac10dd538eca62ea5 192.168.0.101:7001@17001 master - 0 1593333269000 1 connected 333-5460

f04b4ef9e33636f0cf5c0ad1d839300094c73012 192.168.0.101:7007@17007 master - 0 1593333268000 7 connected 0-332 5461-5794 10923-11255

指定主节点：127.0.0.1:7008> cluster replicate f04b4ef9e33636f0cf5c0ad1d839300094c73012

8）删除从节点方式，这样节点7008就从集群移除，并停止服务

[root@localhost bin]# ./redis-cli --cluster del-node 192.168.0.101:7008 551230eebef1f499c5962db11868cadb9b555b11

9）删除主节点方式，先把槽点分给其他主节点，再删除该节点

[root@localhost bin]# ./redis-cli --cluster reshard 192.168.0.101:7001

移动7008的1000个槽点：How many slots do you want to move (from 1 to 16384)? 1000

移动到哪？（7001）What is the receiving node ID? 8ac5cb800d15fdaa8ddc988ac10dd538eca62ea5

指定7008节点：Source node #1: fea53768189af3e3e4849038af13607f59ec84b0

Source node #2: done

删除7008节点：

[root@localhost bin]# ./redis-cli --cluster del-node 192.168.5.100:8007 fea53768189af3e3e4849038af13607f59ec84b0