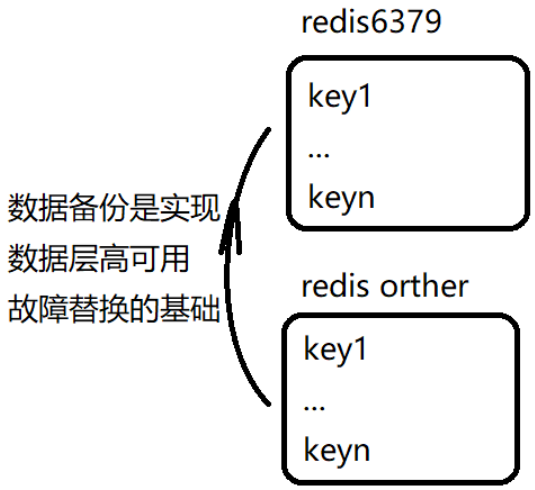
一　高可用的 redis 集群

如果一台系统能够不间断的提供服务，那么这台系统的可用性为100%。当某一个 redis 节点因为故障,无法继续处理数据进行读写操作时，我们需要考虑如何让这个的所有数据依然有效可被客户端使用——创建高可用的 redis 集群。

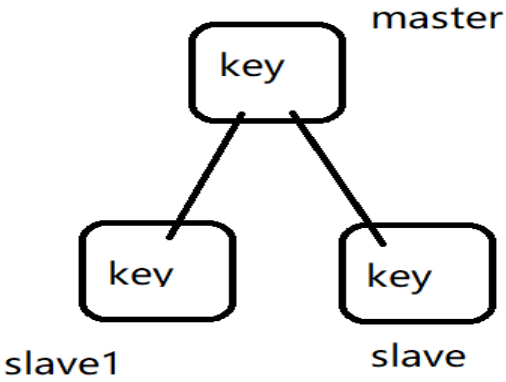
1　数据持续有效（redis 的主从复制）

通过多节点的故障顶替功能,实现高可用,但是顶替上来的节点必须具备原有节点的所有能力---redis 体现在数据的备份

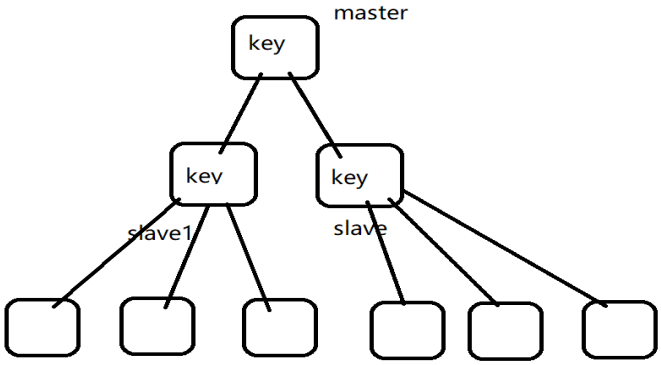


redis 支持高可用的,所以一定能实现主从的数据复制,最终主节点故障,从节点顶替可以完成高可用的故障转移工作.

redis 主从复制,支持一主多从:



redis 也支持多级主从:



根据企业经验,主从结构越复杂,redis 节点就要耗费更多的资源在数据同步上,越不稳定,一级主从 6 个从节点上限.

2 主从实现

2.A 启动三个 redis 节点

```
[root@10-42-175-170 redis-3.2.11]# start-redis.sh
[root@10-42-175-170 redis-3.2.11]# ps -ef|grep redis
root     1279     1  0 22:50 ?        00:00:00 redis-server *:6379
root     1285     1  0 22:50 ?        00:00:00 redis-server *:6380
root     1291     1  0 22:50 ?        00:00:00 redis-server *:6381
root     1295  1247  0 22:50 pts/0    00:00:00 grep redis
```

2.B 设置从节点的主节点

主从的搭建,实际上是通过从节点执行挂接完成.

- **临时挂接**:执行临时挂接的命令,将节点重新启动,重新变成主节点
  从节点中执行命令>slaveof <masterip> <masterport>
  主节点：6379，从节点 6380、6381

```
[root@10-42-175-170 redis-3.2.11]# redis-cli -p 6380
127.0.0.1:6380> slaveof 10.42.175.170 6379
OK
 [root@10-42-175-170 redis-3.2.11]# redis-cli -p 6381
127.0.0.1:6381> slaveof 10.42.175.170 6379
OK
```

- **永久挂接**:通过从节点配置文件中 265 配置 slave 主节点 ip port 如果主节点设置了密码,配置 273 行 master 的 password

2.C 查看主从信息

info replication

```
127.0.0.1:6381> info replication
# Replication
role:slave
master_host:10.42.175.170
master_port:6379
master_link_status:up
master_last_io_seconds_ago:9
master_sync_in_progress:0
slave_repl_offset:71
slave_priority:100
slave_read_only:1
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
```

3 主从结构的简单测试

- 主从数据备份
  - 在 6379 主节点写入一些数据
  - 观察从节点能否读取这些数据
  - 主节点写入数据，从节点也会复制一份：

```
[root@10-42-175-170 redis-3.2.11]# redis-cli -p 6379
127.0.0.1:6379> set name shu
OK
127.0.0.1:6379> exit
[root@10-42-175-170 redis-3.2.11]# redis-cli -p 6380
127.0.0.1:6380> get name
"shu"
```

  - 从节点不能写入数据：

```
127.0.0.1:6381> set name shu
(error) READONLY You can't write against a read only slave.
```

  -
- 将主节点宕机(故障替换)
  - 观察从节点有何变化： 从节点不发生顶替,仅仅记录一下 down 的状态

```
[root@10-42-175-170 redis-3.2.11]# redis-cli -p 6379
127.0.0.1:6379> shutdown
not connected> exit
[root@10-42-175-170 redis-3.2.11]# redis-cli -p 6380
127.0.0.1:6380> info replication
# Replication
```

```
role:slave
master_host:10.42.175.170
master_port:6379
master_link_status:down
master_last_io_seconds_ago:-1
master_sync_in_progress:0
slave_repl_offset:728
master_link_down_since_seconds:22
slave_priority:100
slave_read_only:1
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
```

这种集群只能实现数据的复制，无法实现主节点故障子节点顶替功能---引入哨兵集群

二 哨兵集群(sentinel)

redis 中提供一个高可用结构里的特殊 redis 进程---哨兵.专门负责监听主从结构.实现判断宕机故障,完成故障转移,记录数据记录角色信息的一个集群.

1 原理

- 哨兵启动会连接主节点,同时监听多个主从,需要指定不同主从代号不相同
- 从主节点获取 info 返回的结果,包括从节点状态,记录到哨兵内存
- 每秒中向所有的集群节点发送一个心跳检测(rpc 远程通信协议)
- 发现宕机
    - 从节点宕机:哨兵仅仅会在内存记录中记录一份宕机状态
    - 主节点宕机:发起投票选举,让所有哨兵进程判断主节点宕机之后,进行过半的投票,如果选举结果没有任何一个过半情况,重新进行选举,为了防止重新选举
        - 可以设置从节点的优先级 slave_priority 在 redis.conf 配置文件设置,默认 100
        - 从配置结构上解决这个问题

- 一旦重新选举完成,哨兵将会修改从节点的 redis.conf 配置文件记录他是主节点,将其他的集群的从节点挂接到这个新的主节点上,一旦原有主节点恢复启动,哨兵将其挂接到新 master 下成为从节点

2 配置并启动哨兵的集群

2.A 配置文件

需要准备哨兵集群中每个节点的配置文件 sentinel.conf

2.A.a 某个哨兵配置文件示例

21 行:当前哨兵进程占用的端口号,如果同一个服务器启动多个哨兵,端口号不能相同

```
# The port that this sentinel instance will run on
port 26379

# sentinel announce-ip <ip>
# sentinel announce-port <port>
#
# The above two configuration directives are useful
ts where,
                                                21,1
```

60 行:哨兵监听主从核心配置， 可以写多行监听多个主从

```
#
sentinel monitor mymaster 10.9.100.26 6382 2
# Slaves are auto-discovered, so you don't need to sp
 in
# any way. Sentinel itself will rewrite this configur
dding
-- INSERT --                                    63,45
```

sentinel monitor:开启监听

mymaster:为当前监听的主从启的一个名字

10.9.100.26:主节点的 ip

6382:主节点端口

2:主观下限票数--哨兵集群最少投票数量必须是 2 才能继续判断过半

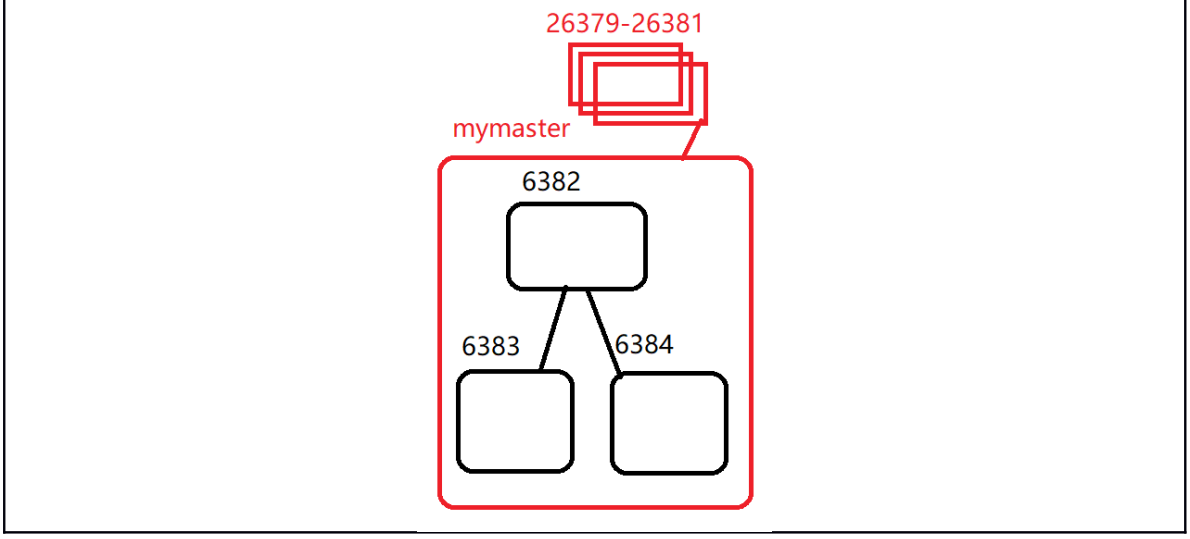
2.B 启动编写好的脚本开启集群

start-sentinel.sh：就是根据配置好的三个哨兵配置文件，启动三个哨兵的集群

```
for port in {6382,6383,6384}
do
rm -f $port/redis.conf
cp $port/redis-ms.conf $port/redis.conf
redis-cli -p $port shutdown
rm -f $port/dump*
rm -f $port/*log
redis-server $port/redis.conf
done;
./reset-sentinel.sh
redis-sentinel sentinel01.conf
redis-sentinel sentinel02.conf
redis-sentinel sentinel03.conf
~
```

启动脚本，6382 为主节点，6383、6384 为从节点
另外 26379、26380、26381 为哨兵，都监听的同一个主从结构

```
[root@10-42-175-170 redis-3.2.11]# start-sentinel.sh
Could not connect to Redis at 127.0.0.1:6382: Connection refused
Could not connect to Redis at 127.0.0.1:6383: Connection refused
Could not connect to Redis at 127.0.0.1:6384: Connection refused
process provider not exist
[root@10-42-175-170 redis-3.2.11]# ps -ef|grep redis
root      1632     1  0 18:41 ?        00:00:00 redis-server *:6382
root      1641     1  0 18:41 ?        00:00:00 redis-server *:6383
root      1651     1  0 18:41 ?        00:00:00 redis-server *:6384
root      1665     1  0 18:41 ?        00:00:00 redis-sentinel *:26379 [sentinel]
root      1669     1  0 18:41 ?        00:00:00 redis-sentinel *:26380 [sentinel]
root      1673     1  0 18:41 ?        00:00:00 redis-sentinel *:26381 [sentinel]
```



3 观察日志理解原理

- 到/tmp/sentinel26379.log sentinel26380.log sentinel26381.log

```
1333:X 30 May 15:09:02.495 # +monitor master mymaster 10.9.100.26 6382
quorum 2
1333:X 30 May 15:09:02.495 * +slave slave 10.9.100.26:6383 10.9.100.26
6383 @ mymaster 10.9.100.26 6382
1333:X 30 May 15:09:02.498 * +slave slave 10.9.100.26:6384 10.9.100.26
6384 @ mymaster 10.9.100.26 6382
1333:X 30 May 15:09:04.510 * +sentinel sentinel
94a5edc7ff7004c15fcebe725d0acf4fe5d7a0fd 10.9.100.26 26381 @ mymaster
10.9.100.26 6382
1333:X 30 May 15:09:04.523 * +sentinel sentinel
8670d1669989c87f169d79d3bf07fb13718a1a4b 10.9.100.26 26380 @ mymaster
10.9.100.26 6382
```

连接主节点,获取从节点信息

- 将从节点宕机 6383

哨兵记录宕机事件(+sdown)

```
1333:X 30 May 15:15:54.961 # +sdown slave 10.9.100.26:6383 10.9.100.26
6383 @ mymaster 10.9.100.26 6382
```

重新启动 6383(+reboot),哨兵记录事件 reboot,删除宕机事件（-sdown）

```
1333:X 30 May 15:16:47.512 * +reboot slave 10.9.100.26:6383
10.9.100.26 6383 @ mymaster 10.9.100.26 6382
1333:X 30 May 15:16:47.589 # -sdown slave 10.9.100.26:6383 10.9.100.26
6383 @ mymaster 10.9.100.26 6382
```

- 将主节点宕机 6382

```
1333:X 30 May 15:18:27.790 # +new-epoch 1
1333:X 30 May 15:18:27.793 # +vote-for-leader
8670d1669989c87f169d79d3bf07fb13718a1a4b 1
1333:X 30 May 15:18:27.809 # +sdown master mymaster 10.9.100.26 6382
1333:X 30 May 15:18:27.880 # +odown master mymaster 10.9.100.26 6382
#quorum 3/2
```

epoch 逻辑时钟值,随着主从替换将会做自增操作.

vote-for-leader 重新执行选举,从已有的 2 个从节点头片选举.

发现主节点宕机+sdown

所有哨兵同意宕机结果 +odown

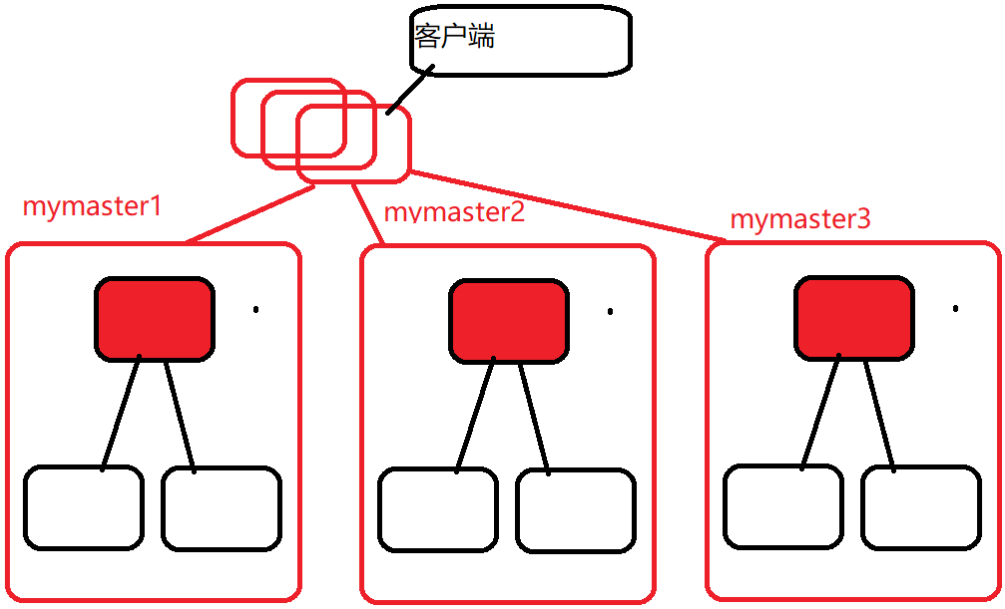
```
1333:X 30 May 15:18:28.473 # +switch-master mymaster 10.9.100.26 6382
10.9.100.26 6384
1333:X 30 May 15:18:28.473 * +slave slave 10.9.100.26:6383 10.9.100.26
6383 @ mymaster 10.9.100.26 6384
1333:X 30 May 15:18:28.473 * +slave slave 10.9.100.26:6382 10.9.100.26
6382 @ mymaster 10.9.100.26 6384
1333:X 30 May 15:18:58.526 # +sdown slave 10.9.100.26:6382 10.9.100.26
6382 @ mymaster 10.9.100.26 6384
```

switch-master:故障转移 6382 换成了 6384

对新的主从进行更改新的监听 6382 挂接到 6384 6383 重新挂接到 6384,其中 6382 刚宕机记录宕机.在把 6382 启动回来,即使 6382 自己认为自己还是 master,sentinel 也会强制给 6382 执行一次 slaveof 命令,挂接到 6384

```
1341:X 30 May 15:23:17.475 # -sdown slave 10.9.100.26:6382 10.9.100.26
6382 @ mymaster 10.9.100.26 6384
1341:X 30 May 15:23:27.389 * +convert-to-slave slave 10.9.100.26:6382
10.9.100.26 6382 @ mymaster 10.9.100.26 6384
```

4 哨兵集群实现的分布式



5 哨兵高可用测试

```
//测试连接哨兵,不能连接 6382 6384 6383 任何一个人
@Test
public void test10(){
    System.out.println(new
HostAndPort("10.9.100.26",26379).toString());
    //收集哨兵的连接信息
    Set<String> sentinelInfo =new HashSet();
    sentinelInfo.add("10.9.100.26:26379");
    sentinelInfo.add("10.9.100.26:26380");
    sentinelInfo.add("10.9.100.26:26381");
    //sentinelInfo.add(new
HostAndPort("10.9.100.26",26379).toString())
    //构建一个哨兵连接池
    JedisSentinelPool pool=new
JedisSentinelPool("mymaster",sentinelInfo);
    //这是在 pool 记录的各种属性就会根据主从的变化而变化
    System.out.println("当前正在使用的 master 信
息:"+pool.getCurrentHostMaster());
    //获取主节点连接对象
    Jedis jedis = pool.getResource();//6384
    jedis.set("name","王老师");
    System.out.println(jedis.get("name"));
}
```

三　redis 的结构演化总结

1　单节点 redis

缺点:

内存容量小

并发上限小

2　多个 redis 节点

优点:

解决了容量小

解决单节点并发小

缺点:

没有数据分片的高可用,一旦某个节点宕机,整个数据缺少一部分

3　哨兵集群

优点:

保证单个数据分片的数据高可用,高可靠

缺点:

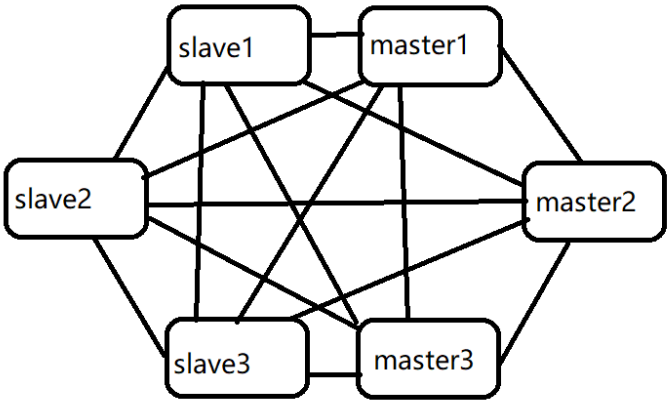
- 很难实现分布式,分布式计算逻辑很难使用一致性 hash
- 做不到这种结构的分布式中 key 和 node 解耦.
  - 当某一个 key 值,一致性 hash 计算分片,和某个主从绑定,想把 key 迁移到别的主从无法实现的.不容易解决数据倾斜.

四 REDIS-CLUSTER

1 redis-cluster 概括

redis3.0 版本,出现了 redis-cluster 的集群结构,这个结构完成了即是分布式,又是高可用的

1.A 结构



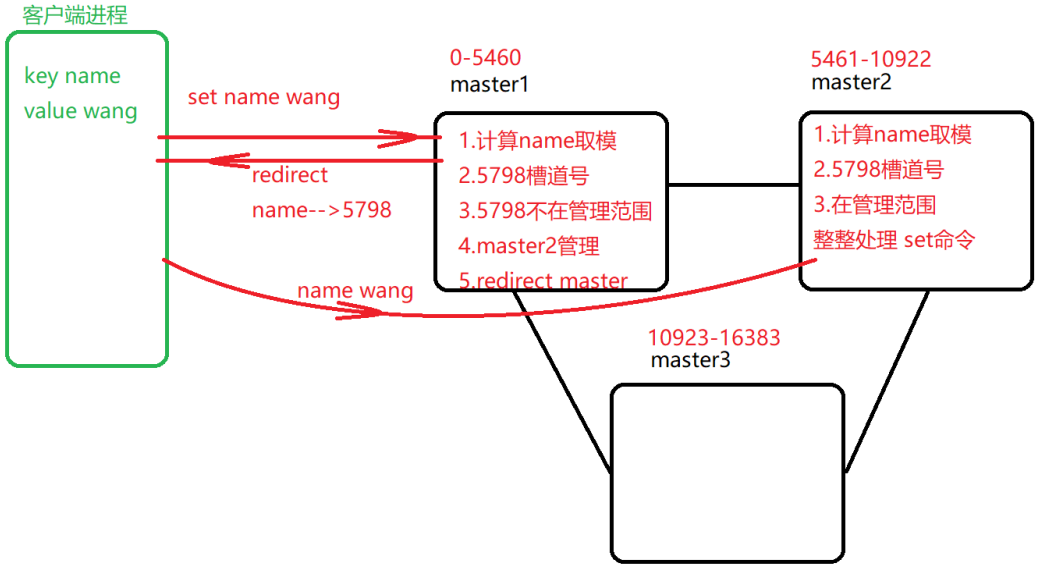
1.B 特性

- 集群所有节点之间是两两互联的,并且通信使用二进制协议优化传输速度(redis-cluster 结构的基础)
- 哨兵进程消失了,但是哨兵高可用监听功能,整合到了 master 节点,通过 master 节点来实现过半选举,监听整个集群功能.集群最小结构就是保证 3 个 master
    - 因为哨兵逻辑整合到 master 设计到集群的过半选举
    - 只有 3 个 master 时,允许集群宕机一个 master(集群容忍度最小是 1)
    - 最大的集群扩展上限(官方数据 redis 节点 1000 个)
- redis-cli 进程客户端访问集群任何一个节点,都可以实现集群的分布式数据管理.因为集群内部引入了分布式的计算逻辑,可以将发送的数据进行计算之后帮助客户端重定向到正确节点来最终处理数据
- 集群中使用了新的分布式计算逻辑---hash 槽.引入了 16384 个槽道号概念[0,...,16383].key 值做为计算数据,先进行计算 CRC16,key 值不变对应整数取余结果不变 对

应槽道号 slot.又由于 master 管理的不同的槽道号,得到与节点的对应关系,松耦合因为槽道可以迁移 key-->slot-->node

1.C 槽道使用示意图

3 个 master 每个主节点管理一份槽道区间



客户端进程无论连接哪个 redis 集群的节点,都会实现数据的分布式计算.

2 测试集群测试效果

在启动的集群中测试新特性,测试具不具备之前学习的逻辑 高可用,数据备份

2.A 启动集群

- 修改 start-cluster.sh 所有节点 ip 地址

  镜像 sh 脚本没有自动读取动态的 ip,使用了固定 10.9.151.60.就需要将这个 ip 修改成你的云主的 ip

```
#create cluster
echo yes|src/redis-trib.rb create --replicas 1 10.9.151.60 8000 1
0.9.151.60:8001 10.9.151.60:8002 10.9.151.60:8003 10.9.151.60:800
4 10.9.151.60:8005
                                                    14,52         Bot
```

vim 的命令修改替换相同内容

:%s/原文内容/替换后新内容/g

```
#create cluster
echo yes|src/redis-trib.rb create --replicas 1 10.9.100.26 8000 1
0.9.100.26:8001 10.9.100.26:8002 10.9.100.26:8003 10.9.100.26:800
4 10.9.100.26:8005
                                                    13,15         Bot
```

shell 脚本运行,将会在云主机中创建一个 3 主,并且各自有一个从节点的 redis-cluster 集群.

2.B 登录测试

- **掌握集群操作的基础命令**
  - 登录客户端命令 redis-cli 多一个选项 c 以集群模式登录

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

  - 集群的一些基础命令
    - >cluster info

      可以在登录一个节点之后,直接调用查看集群的状态
    - >cluster nodes

      展示当前集群所有节点的详细信息

b423b10fe8322a043e04eb0c7823e0a2a65a1478	10.9.104.184:8003	slave	e8cdfc02c5d379becb5eb4be600d757a129a1c42	0 1587196202567	4	connected (槽道号)
节点 id	节点 host:port	角色	当前从节点的主节点 id,如果没有主节点-	创建和操作时间	序号	连接状态和管理槽道

- **体会槽道的效果**

  登录任何一个节点,即使是从节点也可以进行 set 数据操作.

```
[root@10-9-100-26 redis-3.2.11]# redis-cli -c -p 8005
127.0.0.1:8005> set name wanglaoshi
-> Redirected to slot [5798] located at 10.9.100.26:8001
OK
10.9.100.26:8001>
```

- **测试高可用数据复制**

  8000m-->8003s

  8001m-->8004s

  8002m-->8005s

  登录主节点 8001 通过 set name 写入了一个数据,只需要观察 8004 作为从节点有没有同步到这个数据

```
10.9.100.26:8001> keys *
1) "name"
127.0.0.1:8004> keys *
1) "name"
```

  高可用测试,找到一个主节点,将主节点宕机,观察集群节点状态变化,有没有故障转移功能

```
[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
127.0.0.1:8001> cluster nodes
4c1cf304c71447361f8faac5c111747fe3c21176 10.9.100.26:8000
aster - 1590829535746 1590829534544 1 disconnected 0-5460
```

集群已经记录 8000 宕机的状态 disconnected(其他主节点过半 2 个,不断心跳检测判断宕机),会有一段时间留给集群 master 进行选举,从节点没有顶替

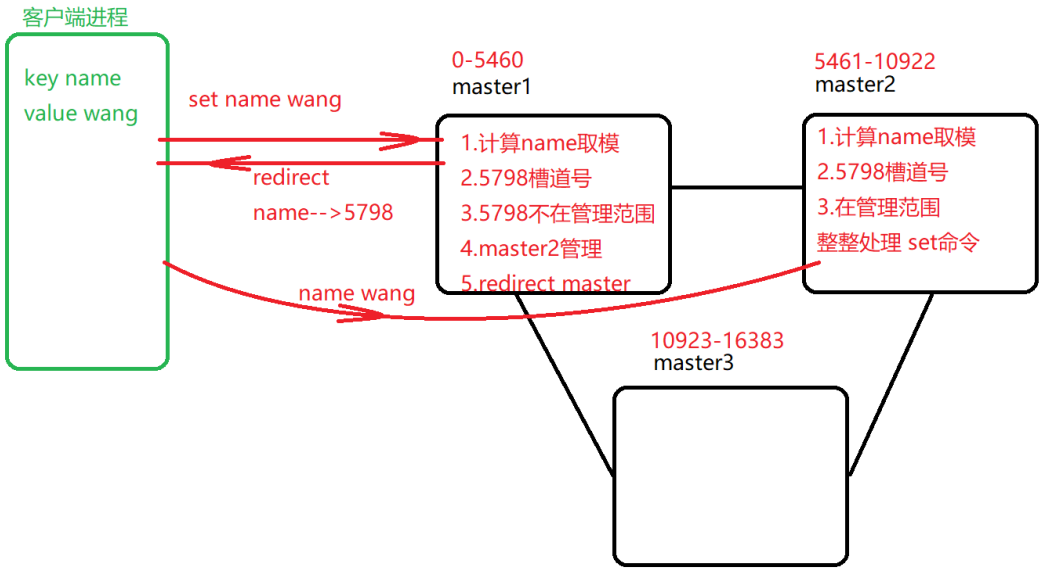
```
e07a56c7898aa89296f8177a785b30584f2bf02e 10.9.100.26:8003 slave 4c1cf304c71447361f8faac5c111747fe3c21176 0 1590829539555 4 connected
```

等待一段时间,经过替换选举,8003 成为新的主节点

```
e07a56c7898aa89296f8177a785b30584f2bf02e 10.9.100.26:8003 master - 0 1590829698940 7 connected 0-5460
```

将 8000 重新启动,加入集群并且以 8003 的从节点身份提供功能

3 槽道原理



- 节点如何判断槽道号归属

  进程接收命令 set name wang,进程调用方法 CRC16 得到 0-16383 之间的整数(槽道号 name-->5798 当前节点凭什么判断 5798 槽道号是不是我来管理?

- 如何获取正确的槽道管理者信息

  节点可以正确判断管理权的,name-->5798(槽道号)在众多集群节点中由谁来管理?

3.A 槽道组成结构

在 redis-cluster 存在分布式计算原则--hash 槽,引入 16384 个槽道.

- 16384 位的二进制,以 2048 个元素的 byte(2048*8bit=16384 位)数组存储在每个节点内存里---**位序列**
- 16384 个元素的数组,元素值,指向的是一个内存的节点对象 node---**共享数组/索引数组**

3.B 位序列

每个节点在集群创建之初都会根据槽道管理权的分配创建一个二进制数据—位序列

- **主节点的二进制:**将管理的槽道号和二进制中的 bit 值做对应关系

16384 位的二进制,定义,从左到右,位数 0,1,2,...16383 对应槽道号;如果二进制的第 0 位 bit 值为 1,表示该槽道号在本节点有管理权,如果是 0 没有管理权.

- **从节点的二进制**:因为从节点没有槽道管理权的,二进制的值 0 （16384 位都为 0）

- **使用位序列判断所属权**
  - 在 8002 中调用 set name haha
    - 计算 name 的槽道号 name-->5798
    - 到位序列中找 5798 下标对应的 bit-->0
    - 判断 5798 这个槽道 8002 没有所属权
- **通信的作用**(为什么槽道号 16384 个?)

  集群节点内部通信时,需要封装一个通信槽道状态数据,用到了这个 2 进制,byte[2048]大小刚好是 2KB(16384bit),通信头中有一个头携带这个二进制,如果槽道 16385 至少 2 个 2KB 的头才能写到当前节点想要传送出去的槽道数据.

3.C 索引/共享数组

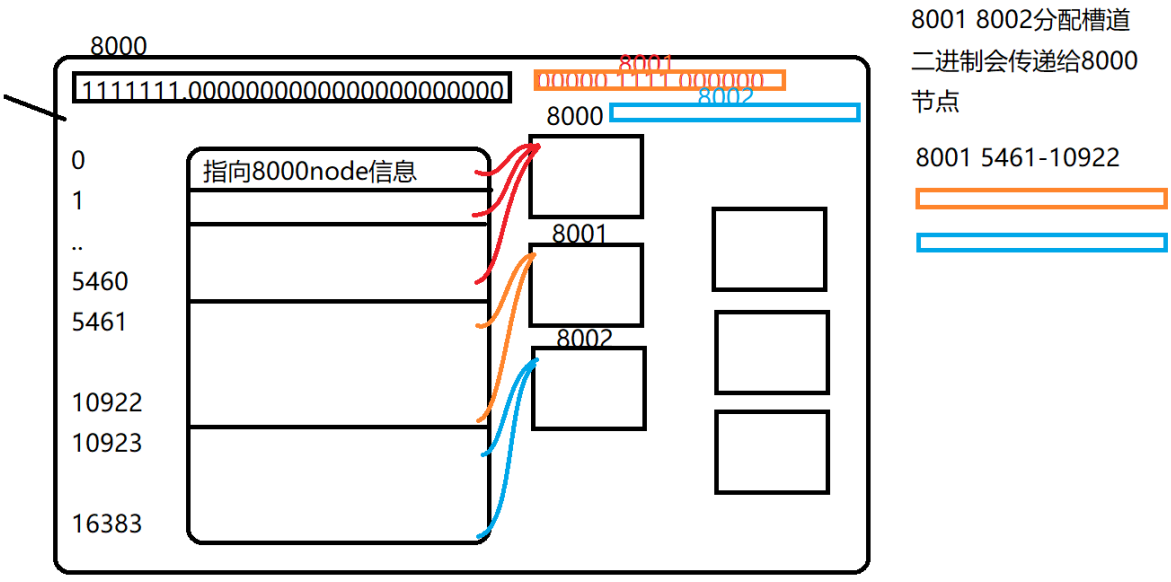
- **集群的创建阶段经过两两互联通信**

经过两两互联通信,每个节点中的节点信息对象,都会通信给其他所有节点,所以登录一个创建好的集群中任何一个节点,能够通过 cluster nodes 查看到所有节点信息.

集群的节点越多,每个节点保存的所有节点信息数据量越大.扩展上限 1000 个节点原因之一.理论上分片个数可以 16384,每个主节点管理一个槽道.不管理槽道的主节点可以有无数个.

- **创建数组**

每个集群节点中,只要集群创建完毕都会保存一个完全一样的数组对象（索引数组）.他是在一开始就创建的,但是元素的赋值是在分配槽道时完成的



分配槽道时,二进制在变化,两两互联中携带相互传递,所有节点都能获取集群槽道的所有分配结果.所有节点将会对这个数组进行赋值.

元素个数 16384 个,下标刚好对应槽道号.对应元素的值,引用变量指向该槽道的正确管理者.

- **数组创建所属权判断完毕之后寻找正确管理**
  - 8002 调用 set name
  - name-->5798-->bit 0 没有管理权
  - 到数组中拿到 5798 下标的元素引用变量--->8001 详细信息包括 ip port
  - 让客户端重定向到 8001

3.D 扩展问题

- 能否使用数组直接判断所属权?
  - 理论可以的,但是比二进制判断效率低的多的多
- 不判断所属权,直接利用数组进行重定向?
  - 不可以,进入死循环
- 二进制进行通信的
  - 没有通信,相互之间无法获取集群的槽道变化

4 JAVA 测试（JedisCluster）

JedisCluster 已经封装了客户端代码的 hash 槽的分布式计算逻辑
只要给这个对象一个 key-value 总能直接算出来 key 对应槽道,从而连接正确节点实现数据的发送

JedisCluster 不会让服务端给客户端重定向的机会.底层封装了 jedisPool,连接所有节点的客户端对象.

```
@Test
public void test01(){
    //收集集群节点信息,由于节点连接任何一个都能获取到集群所有节点信息
    //所以只需要提供一个必定连接的节点就可以,1 个,若干个
    Set<HostAndPort> info=new HashSet<>();
    info.add(new HostAndPort("10.9.100.26",8000));
    //创建这个 JedisCluster 对象
    //控制连接池属性
    JedisPoolConfig config=new JedisPoolConfig();
    config.setMaxIdle(10);
    config.setMinIdle(2);
    config.setMaxTotal(100);
    JedisCluster cluster=new JedisCluster(info,config);
    cluster.set("name","王老师");
    System.out.println(cluster.get("name"));
}
```