<http://blog.csdn.net/dc_726/article/details/48552531>

版权声明：本文为博主原创文章，未经博主允许不得转载。欢迎访问 http://blog.csdn.net/dc\_726

目录[(?)[+]](http://blog.csdn.net/dc_726/article/details/48552531)

# 全面剖析Redis Cluster原理和应用

# 1.Redis Cluster总览

## 1.1 设计原则和初衷

在[官方文档Cluster Spec](http://redis.io/topics/cluster-spec)中，作者详细介绍了Redis集群为什么要设计成现在的样子。最核心的目标有三个：

1. **性能**：这是Redis赖以生存的看家本领，增加集群功能后当然不能对性能产生太大影响，所以Redis采取了P2P而非Proxy方式、异步复制、客户端重定向等设计，而牺牲了部分的一致性、使用性。
2. **水平扩展**：集群的最重要能力当然是扩展，文档中称可以线性扩展到1000结点。
3. **可用性**：在Cluster推出之前，可用性要靠Sentinel保证。有了集群之后也自动具有了Sentinel的监控和自动Failover能力。

## 1.2 架构变化与CAP理论

Redis Cluster集群功能推出已经有一段时间了。在单机版的Redis中，每个Master之间是没有任何通信的，所以我们一般在Jedis客户端或者Codis这样的代理中做Pre-sharding。按照CAP理论来说，**单机版的Redis属于保证CP(Consistency & Partition-Tolerancy)而牺牲A(Availability)**，也就说Redis能够保证所有用户看到相同的数据（一致性，因为Redis不自动冗余数据）和网络通信出问题时，暂时隔离开的子系统能继续运行（分区容忍性，因为Master之间没有直接关系，不需要通信），但是不保证某些结点故障时，所有请求都能被响应（可用性，某个Master结点挂了的话，那么它上面分片的数据就无法访问了）。

有了Cluster功能后，**Redis从一个单纯的NoSQL内存数据库变成了分布式NoSQL数据库，CAP模型也从CP变成了AP**。也就是说，通过自动分片和冗余数据，Redis具有了真正的分布式能力，某个结点挂了的话，因为数据在其他结点上有备份，所以其他结点顶上来就可以继续提供服务，保证了Availability。然而，也正因为这一点，Redis无法保证曾经的强一致性了。这也是CAP理论要求的，三者只能取其二。

关于CAP理论的通俗讲解，请参考我的译文[《可能是CAP理论的最好解释 》](http://blog.csdn.net/dc_726/article/details/42784237)。简单分析了Redis在架构上的变化后，咱们就一起来体验一下Redis Cluster功能吧！

# 2.Redis集群初探

Redis的安装很简单，以前已经介绍过，就不详细说了。关于Redis Cluster的基础知识之前也有过整理，请参考[《Redis集群功能预览》](http://blog.csdn.net/dc_726/article/details/43991905)。如果需要全面的了解，那一定要看[官方文档Cluster Tutorial](http://redis.io/topics/cluster-tutorial)，只看这一个就够了！

## 2.1 集群配置

要想开启Redis Cluster模式，有几项配置是必须的。此外为了方便使用和后续的测试，我还额外做了一些配置：

* **绑定地址**：bind 192.168.XXX.XXX。不能绑定到127.0.0.1或localhost，否则指导客户端重定向时会报”Connection refused”的错误。
* **开启Cluster**：cluster-enabled yes
* **集群配置文件**：cluster-config-file nodes-7000.conf。这个配置文件不是要我们去配的，而是Redis运行时保存配置的文件，所以我们也不可以修改这个文件。
* **集群超时时间**：cluster-node-timeout 15000。结点超时多久则认为它宕机了。
* **槽是否全覆盖**：cluster-require-full-coverage no。默认是yes，**只要有结点宕机导致16384个槽没全被覆盖，整个集群就全部停止服务**，所以一定要改为no
* 后台运行：daemonize yes
* 输出日志：logfile “./redis.log”
* 监听端口：port 7000

配置好后，根据我们的集群规模，拷贝出来几份同样的配置文件，唯一不同的就是监听端口，可以依次改为7001、7002… 因为Redis Cluster如果数据冗余是1的话，至少要3个Master和3个Slave，所以我们拷贝出6个实例的配置文件。为了避免相互影响，为6个实例的配置文件建立独立的文件夹。

[root@8gVm redis-3.0.4]# pwd

/root/Software/redis-3.0.4

[root@8gVm redis-3.0.4]# tree -I "\*log|nodes\*" cfg-cluster/

cfg-cluster/

├── 7000

│ └── redis.conf.7000

├── 7001

│ └── redis.conf.7001

├── 7002

│ └── redis.conf.7002

├── 7003

│ └── redis.conf.7003

├── 7004

│ └── redis.conf.7004

└── 7005

└── redis.conf.7005

6 directories, 6 files

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

## 2.2 redis-trib管理器

Redis作者应该是个Ruby爱好者，Ruby客户端就是他开发的。这次集群的管理功能没有嵌入到Redis代码中，于是作者又顺手写了个叫做redis-trib的管理脚本。redis-trib依赖Ruby和RubyGems，以及redis扩展。可以先用which命令查看是否已安装ruby和rubygems，用gem list –local查看本地是否已安装redis扩展。

最简便的方法就是用apt或yum包管理器安装RubyGems后执行gem install redis。如果网络或环境受限的话，可以手动安装RubyGems和redis扩展（国外链接可能无法下载，可以从[CSDN下载](http://download.csdn.net/download/lihcc/8722699)）：

[root@8gVm Software]# wget https://github.com/rubygems/rubygems/releases/download/v2.2.3/rubygems-2.2.3.tgz

[root@8gVm Software]# tar xzvf rubygems-2.2.3.tgz

[root@8gVm Software]# cd rubygems-2.2.3

[root@8gVm rubygems-2.2.3]# ruby setup.rb --no-rdoc --no-ri

[root@8gVm Software]# wget https://rubygems.org/downloads/redis-3.2.1.gem

[root@8gVm Software]# gem install redis-3.2.1.gem --local --no-rdoc --no-ri

Successfully installed redis-3.2.1

1 gem installed

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

## 2.3 集群建立

首先，启动我们配置好的6个Redis实例。

[root@8gVm redis-3.0.4]# for ((i=0; i<6; ++i))

> do

> cd cfg-cluster/700$i && ../../src/redis-server redis.conf.700$i && cd -

> done

* 1
* 2
* 3
* 4

此时6个实例还没有形成集群，现在用redis-trb.rb管理脚本建立起集群。可以看到，redis-trib默认用前3个实例作为Master，后3个作为Slave。**因为Redis基于Master-Slave做数据备份，而非像Cassandra或Hazelcast一样不区分结点角色，自动复制并分配Slot的位置到各个结点**。

[root@8gVm redis-3.0.4]# src/redis-trib.rb create --replicas 1 192.168.1.100:7000 192.168.1.100:7001 192.168.1.100:7002 192.168.1.100:7003 192.168.1.100:7004 192.168.1.100:7005

>>> Creating cluster

Connecting to node 192.168.1.100:7000: OK

Connecting to node 192.168.1.100:7001: OK

Connecting to node 192.168.1.100:7002: OK

Connecting to node 192.168.1.100:7003: OK

Connecting to node 192.168.1.100:7004: OK

Connecting to node 192.168.1.100:7005: OK

>>> Performing hash slots allocation on 6 nodes...

Using 3 masters:

192.168.1.100:7000

192.168.1.100:7001

192.168.1.100:7002

Adding replica 192.168.1.100:7003 to 192.168.1.100:7000

Adding replica 192.168.1.100:7004 to 192.168.1.100:7001

Adding replica 192.168.1.100:7005 to 192.168.1.100:7002

...

Can I set the above configuration? (type 'yes' to accept): yes

>>> Nodes configuration updated

>>> Assign a different config epoch to each node

>>> Sending CLUSTER MEET messages to join the cluster

Waiting for the cluster to join....

>>> Performing Cluster Check (using node 192.168.1.100:7000)

...

[OK] All nodes agree about slots configuration.

>>> Check for open slots...

>>> Check slots coverage...

[OK] All 16384 slots covered.

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28

至此，集群就已经建立成功了！“贴心”的Redis还在utils/create-cluster下提供了一个create-cluster脚本，能够创建出一个集群，类似我们上面建立起的3主3从的集群。

## 2.4 简单测试

我们连接到集群中的任意一个结点，**启动redis-cli时要加-c选项**，存取两个Key-Value感受一下Redis久违的集群功能。

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000

192.168.1.100:7000> set foo bar

-> Redirected to slot [12182] located at 192.168.1.100:7002

OK

192.168.1.100:7002> set hello world

-> Redirected to slot [866] located at 192.168.1.100:7000

OK

192.168.1.100:7000> get foo

-> Redirected to slot [12182] located at 192.168.1.100:7002

"bar"

192.168.1.100:7002> get hello

-> Redirected to slot [866] located at 192.168.1.100:7000

"world"

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

仔细观察能够注意到，redis-cli根据指示，不断在7000和7002结点之前重定向跳转。如果启动时不加-c选项的话，就能看到以错误形式显示出的MOVED重定向消息。

[root@8gVm redis-3.0.4]# src/redis-cli -h 192.168.1.100 -p 7000

192.168.1.100:7000> get foo

(error) MOVED 12182 192.168.1.100:7002

* 1
* 2
* 3

## 2.5 集群重启

目前redis-trib的功能还比较弱，需要重启集群的话先手动kill掉各个进程，然后重新启动就可以了。这也有点太… 网上有人重启后会碰到问题，我还比较幸运，这种“土鳖”的方式重启试了两次还没发现问题。

[root@8gVm redis-3.0.4]# ps -ef | grep redis | awk '{print $2}' | xargs kill

* 1

# 3.高级功能尝鲜

说是“高级功能”，其实在其他分布式系统中早就都有实现了，只不过在Redis世界里是比较新鲜的。本部分主要试验一下Redis Cluster中的数据迁移(Resharding)和故障转移功能。

## 3.1 数据迁移

本小节我们体验一下Redis集群的Resharding功能！

### 3.1.1 创建测试数据

首先保存foo1~10共10个Key-Value作为测试数据。

[root@8gVm redis-3.0.4]# for ((i=0; i<10; ++i))

> do

> src/redis-cli -c -h 192.168.1.100 -p 7000 set foo$i bar

> done

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000

192.168.1.100:7000> keys \*

1) "foo6"

2) "foo7"

3) "foo3"

4) "foo2"

192.168.1.100:7000> get foo4

-> Redirected to slot [9426] located at 192.168.1.100:7001

"bar"

192.168.1.100:7001> keys \*

1) "foo4"

2) "foo8"

192.168.1.100:7001> get foo5

-> Redirected to slot [13555] located at 192.168.1.100:7002

"bar"

192.168.1.100:7002> keys \*

1) "foo5"

2) "foo1"

3) "foo10"

4) "foo9"

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25

### 3.1.2 启动新结点

参照之前的方法新拷贝出两份redis.conf配置文件redis.conf.7010和7011，与之前结点的配置文件做一下区分。启动新的两个Redis实例之后，通过redis-trib.rb脚本添加新的Master和Slave到集群中。

[root@8gVm redis-3.0.4]# cd cfg-cluster/7010 && ../../src/redis-server redis.conf.7010 && cd -

[root@8gVm redis-3.0.4]# cd cfg-cluster/7011 && ../../src/redis-server redis.conf.7011 && cd -

* 1
* 2

### 3.1.3 添加到集群

使用redis-trib.rb add-node分别将两个新结点添加到集群中，一个作为Master，一个作为其Slave。

[root@8gVm redis-3.0.4]# src/redis-trib.rb add-node 192.168.1.100:7010 192.168.1.100:7000

>>> Adding node 192.168.1.100:7010 to cluster 192.168.1.100:7000

Connecting to node 192.168.1.100:7000: OK

Connecting to node 192.168.1.100:7001: OK

Connecting to node 192.168.1.100:7002: OK

Connecting to node 192.168.1.100:7005: OK

Connecting to node 192.168.1.100:7003: OK

Connecting to node 192.168.1.100:7004: OK

>>> Performing Cluster Check (using node 192.168.1.100:7000)

...

[OK] All nodes agree about slots configuration.

>>> Check for open slots...

>>> Check slots coverage...

[OK] All 16384 slots covered.

Connecting to node 192.168.1.100:7010: OK

>>> Send CLUSTER MEET to node 192.168.1.100:7010 to make it join the cluster.

[OK] New node added correctly.

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000 cluster nodes

0d1f9c979684e0bffc8230c7bb6c7c0d37d8a5a9 192.168.1.100:7010 master - 0 1442452249525 0 connected

...

[root@8gVm redis-3.0.4]# src/redis-trib.rb add-node --slave --master-id 0d1f9c979684e0bffc8230c7bb6c7c0d37d8a5a9 192.168.1.100:7011 192.168.1.100:7000

>>> Adding node 192.168.1.100:7011 to cluster 192.168.1.100:7000

Connecting to node 192.168.1.100:7000: OK

Connecting to node 192.168.1.100:7010: OK

Connecting to node 192.168.1.100:7001: OK

Connecting to node 192.168.1.100:7002: OK

Connecting to node 192.168.1.100:7005: OK

Connecting to node 192.168.1.100:7003: OK

Connecting to node 192.168.1.100:7004: OK

>>> Performing Cluster Check (using node 192.168.1.100:7000)

...

[OK] All nodes agree about slots configuration.

>>> Check for open slots...

>>> Check slots coverage...

[OK] All 16384 slots covered.

Connecting to node 192.168.1.100:7011: OK

>>> Send CLUSTER MEET to node 192.168.1.100:7011 to make it join the cluster.

Waiting for the cluster to join.

>>> Configure node as replica of 192.168.1.100:7010.

[OK] New node added correctly.

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42

### 3.1.4 Resharding

通过redis-trib.rb reshard可以交互式地迁移Slot。下面的例子将5000个Slot从7000~7002迁移到7010上。也可以通过./redis-trib.rb reshard <host>:<port> --from <node-id> --to <node-id> --slots --yes在程序中自动完成迁移。

[root@8gVm redis-3.0.4]# src/redis-trib.rb reshard 192.168.1.100:7000

Connecting to node 192.168.1.100:7000: OK

Connecting to node 192.168.1.100:7010: OK

Connecting to node 192.168.1.100:7001: OK

Connecting to node 192.168.1.100:7002: OK

Connecting to node 192.168.1.100:7005: OK

Connecting to node 192.168.1.100:7011: OK

Connecting to node 192.168.1.100:7003: OK

Connecting to node 192.168.1.100:7004: OK

>>> Performing Cluster Check (using node 192.168.1.100:7000)

M: b2036adda128b2eeffa36c3a2056444d23b548a8 192.168.1.100:7000

slots:0-5460 (4128 slots) master

1 additional replica(s)

M: 0d1f9c979684e0bffc8230c7bb6c7c0d37d8a5a9 192.168.1.100:7010

slots:0 (4000 slots) master

1 additional replica(s)

...

[OK] All nodes agree about slots configuration.

>>> Check for open slots...

>>> Check slots coverage...

[OK] All 16384 slots covered.

How many slots do you want to move (from 1 to 16384)? 5000

What is the receiving node ID? 0d1f9c979684e0bffc8230c7bb6c7c0d37d8a5a9

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:all

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000 cluster nodes

0d1f9c979684e0bffc8230c7bb6c7c0d37d8a5a9 192.168.1.100:7010 master - 0 1442455872019 7 connected 0-1332 5461-6794 10923-12255

b2036adda128b2eeffa36c3a2056444d23b548a8 192.168.1.100:7000 myself,master - 0 0 1 connected 1333-5460

b5ab302f5c2395e3c8194c354a85d02f89bace62 192.168.1.100:7001 master - 0 1442455875022 2 connected 6795-10922

0c565e207ce3118470fd5ed3c806eb78f1fdfc01 192.168.1.100:7002 master - 0 1442455874521 3 connected 12256-16383

...

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34

迁移完成后，查看之前保存的foo1~10的分布情况，可以看到部分Key已经迁移到了新的结点7010上。

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000 keys "\*"

1) "foo3"

2) "foo7"

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7001 keys "\*"

1) "foo4"

2) "foo8"

3) "foo0"

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7002 keys "\*"

1) "foo1"

2) "foo9"

3) "foo5"

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7010 keys "\*"

1) "foo6"

2) "foo2"

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

## 3.2 故障转移

在高可用性方面，Redis可算是能够”Auto”一把了！**Redis Cluster重用了Sentinel的代码逻辑，不需要单独启动一个Sentinel集群，Redis Cluster本身就能自动进行Master选举和Failover切换**。

下面我们故意kill掉7010结点，之后可以看到结点状态变成了fail，而Slave 7011被选举为新的Master。

[root@8gVm redis-3.0.4]# kill 43637

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000 cluster nodes

0d1f9c979684e0bffc8230c7bb6c7c0d37d8a5a9 192.168.1.100:7010 master,fail - 1442456829380 1442456825674 7 disconnected

b2036adda128b2eeffa36c3a2056444d23b548a8 192.168.1.100:7000 myself,master - 0 0 1 connected 1333-5460

b5ab302f5c2395e3c8194c354a85d02f89bace62 192.168.1.100:7001 master - 0 1442456848722 2 connected 6795-10922

0c565e207ce3118470fd5ed3c806eb78f1fdfc01 192.168.1.100:7002 master - 0 1442456846717 3 connected 12256-16383

5a3c67248b1df554fbf2c93112ba429f31b1d3d1 192.168.1.100:7005 slave 0c565e207ce3118470fd5ed3c806eb78f1fdfc01 0 1442456847720 6 connected

99bff22b97119cf158d225c2b450732a1c0d3c44 192.168.1.100:7011 master - 0 1442456849725 8 connected 0-1332 5461-6794 10923-12255

cd305d509c34842a8047e19239b64df94c13cb96 192.168.1.100:7003 slave b2036adda128b2eeffa36c3a2056444d23b548a8 0 1442456848220 4 connected

64b544cdd75c1ce395fb9d0af024b7f2b77213a3 192.168.1.100:7004 slave b5ab302f5c2395e3c8194c354a85d02f89bace62 0 1442456845715 5 connected

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

尝试查询之前保存在7010上的Key，可以看到7011顶替上来继续提供服务，整个集群没有受到影响。

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000 get foo6

"bar"

[root@8gVm redis-3.0.4]#

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000 get foo2

"bar"

* 1
* 2
* 3
* 4
* 5

# 4.内部原理剖析

前面我们已经学习过，用Redis提供的redis-trib或create-cluster脚本能几步甚至一步就建立起一个Redis集群。这一部分我们为了深入学习，所以要暂时抛开这些方便的工具，完全手动建立一遍上面的3主3从集群。

## 4.1 集群发现：MEET

最开始时，每个Redis实例自己是一个集群，我们通过cluster meet让各个结点互相“握手”。这也是Redis Cluster目前的一个欠缺之处：**缺少结点的自动发现功能**。

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000 cluster nodes

33c0bd93d7c7403ef0239ff01eb79bfa15d2a32c :7000 myself,master - 0 0 0 connected

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000 cluster meet 192.168.1.100 7001

OK

...

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000 cluster meet 192.168.1.100 7005

OK

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000 cluster nodes

7b953ec26bbdbf67179e5d37e3cf91626774e96f 192.168.1.100:7003 master - 0 1442466369259 4 connected

5d9f14cec1f731b6477c1e1055cecd6eff3812d4 192.168.1.100:7005 master - 0 1442466368659 4 connected

33c0bd93d7c7403ef0239ff01eb79bfa15d2a32c 192.168.1.100:7000 myself,master - 0 0 1 connected

63162ed000db9d5309e622ec319a1dcb29a3304e 192.168.1.100:7001 master - 0 1442466371262 3 connected

45baa2cb45435398ba5d559cdb574cfae4083893 192.168.1.100:7002 master - 0 1442466372264 2 connected

cdd5b3a244761023f653e08cb14721f70c399b82 192.168.1.100:7004 master - 0 1442466370261 0 connecte

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

## 4.2 角色设置：REPLICATE

结点全部“握手”成功后，就可以用cluster replicate命令为结点指定角色了，默认每个结点都是Master。

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7003 cluster replicate 33c0bd93d7c7403ef0239ff01eb79bfa15d2a32c

OK

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7004 cluster replicate 63162ed000db9d5309e622ec319a1dcb29a3304e

OK

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7005 cluster replicate 45baa2cb45435398ba5d559cdb574cfae4083893

OK

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000 cluster nodes

7b953ec26bbdbf67179e5d37e3cf91626774e96f 192.168.1.100:7003 slave 33c0bd93d7c7403ef0239ff01eb79bfa15d2a32c 0 1442466812984 4 connected

5d9f14cec1f731b6477c1e1055cecd6eff3812d4 192.168.1.100:7005 slave 45baa2cb45435398ba5d559cdb574cfae4083893 0 1442466813986 5 connected

33c0bd93d7c7403ef0239ff01eb79bfa15d2a32c 192.168.1.100:7000 myself,master - 0 0 1 connected

63162ed000db9d5309e622ec319a1dcb29a3304e 192.168.1.100:7001 master - 0 1442466814987 3 connected

45baa2cb45435398ba5d559cdb574cfae4083893 192.168.1.100:7002 master - 0 1442466811982 2 connected

cdd5b3a244761023f653e08cb14721f70c399b82 192.168.1.100:7004 slave 63162ed000db9d5309e622ec319a1dcb29a3304e 0 1442466812483 3 connected

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

## 4.3 槽指派：ADDSLOTS

设置好主从关系之后，就可以用cluster addslots命令指派16384个槽的位置了。有点恶心的是，**ADDSLOTS命令需要在参数中一个个指明槽的ID，而不能指定范围**。这里用Bash 3.0的特性简化了，不然就得用Bash的循环来完成了：

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7000 cluster addslots {0..5000}

OK

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7001 cluster addslots {5001..10000}

OK

[root@8gVm redis-3.0.4]# src/redis-cli -c -h 192.168.1.100 -p 7001 cluster addslots {10001..16383}

OK

[root@8gVm redis-3.0.4]# src/redis-trib.rb check 192.168.1.100:7000

Connecting to node 192.168.1.100:7000: OK

...

>>> Performing Cluster Check (using node 192.168.1.100:7000)

...

[OK] All nodes agree about slots configuration.

>>> Check for open slots...

>>> Check slots coverage...

[OK] All 16384 slots covered.

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

这样我们就通过手动执行命令得到了与之前一样的集群。

## 4.4 数据迁移：MIGRATE

真正开始Resharding之前，redis-trib会先在源结点和目的结点上执行cluster setslot <slot> importing和cluster setslot <slot> migrating命令，将要迁移的槽分别标记为迁出中和导入中的状态。然后，执行cluster getkeysinslot获得Slot中的所有Key。最后就可以对每个Key执行migrate命令进行迁移了。槽迁移完成后，执行cluster setslot命令通知整个集群槽的指派已经发生变化。

关于迁移过程中的数据访问，客户端访问源结点时，如果Key还在源结点上就直接操作。**如果已经不在源结点了，就向客户端返回一个ASK错误，将客户端重定向到目的结点**。

## 4.5 内部数据结构

Redis Cluster功能涉及三个核心的数据结构clusterState、clusterNode、clusterLink都在cluster.h中定义。这三个数据结构中最重要的属性就是：**clusterState.slots、clusterState.slots\_to\_keys和clusterNode.slots了，它们保存了三种映射关系**：

* clusterState：集群状态   
  + nodes：所有结点
  + migrating\_slots\_to：迁出中的槽
  + importing\_slots\_from：导入中的槽
  + **slots\_to\_keys：槽中包含的所有Key，用于迁移Slot时获得其包含的Key**
  + **slots：Slot所属的结点，用于处理请求时判断Key所在Slot是否自己负责**
* clusterNode：结点信息   
  + **slots：结点负责的所有Slot，用于发送Gossip消息通知其他结点自己负责的Slot**。通过位图方式保存节省空间，**16384/8恰好是2048字节，所以槽总数16384不是随意定的**。
* clusterLink：与其他结点通信的连接

// 集群状态，每个节点都保存着一个这样的状态，记录了它们眼中的集群的样子。

// 另外，虽然这个结构主要用于记录集群的属性，但是为了节约资源，

// 有些与节点有关的属性，比如 slots\_to\_keys 、 failover\_auth\_count

// 也被放到了这个结构里面。

typedef struct clusterState {

...

// 指向当前节点的指针

clusterNode \*myself; /\* This node \*/

// 集群当前的状态：是在线还是下线

int state; /\* REDIS\_CLUSTER\_OK, REDIS\_CLUSTER\_FAIL, ... \*/

// 集群节点名单（包括 myself 节点）

// 字典的键为节点的名字，字典的值为 clusterNode 结构

dict \*nodes; /\* Hash table of name -> clusterNode structures \*/

// 记录要从当前节点迁移到目标节点的槽，以及迁移的目标节点

// migrating\_slots\_to[i] = NULL 表示槽 i 未被迁移

// migrating\_slots\_to[i] = clusterNode\_A 表示槽 i 要从本节点迁移至节点 A

clusterNode \*migrating\_slots\_to[REDIS\_CLUSTER\_SLOTS];

// 记录要从源节点迁移到本节点的槽，以及进行迁移的源节点

// importing\_slots\_from[i] = NULL 表示槽 i 未进行导入

// importing\_slots\_from[i] = clusterNode\_A 表示正从节点 A 中导入槽 i

clusterNode \*importing\_slots\_from[REDIS\_CLUSTER\_SLOTS];

// 负责处理各个槽的节点

// 例如 slots[i] = clusterNode\_A 表示槽 i 由节点 A 处理

clusterNode \*slots[REDIS\_CLUSTER\_SLOTS];

// 跳跃表，表中以槽作为分值，键作为成员，对槽进行有序排序

// 当需要对某些槽进行区间（range）操作时，这个跳跃表可以提供方便

// 具体操作定义在 db.c 里面

zskiplist \*slots\_to\_keys;

...

} clusterState;

// 节点状态

struct clusterNode {

...

// 节点标识

// 使用各种不同的标识值记录节点的角色（比如主节点或者从节点），

// 以及节点目前所处的状态（比如在线或者下线）。

int flags; /\* REDIS\_NODE\_... \*/

// 由这个节点负责处理的槽

// 一共有 REDIS\_CLUSTER\_SLOTS / 8 个字节长

// 每个字节的每个位记录了一个槽的保存状态

// 位的值为 1 表示槽正由本节点处理，值为 0 则表示槽并非本节点处理

// 比如 slots[0] 的第一个位保存了槽 0 的保存情况

// slots[0] 的第二个位保存了槽 1 的保存情况，以此类推

unsigned char slots[REDIS\_CLUSTER\_SLOTS/8]; /\* slots handled by this node \*/

// 指针数组，指向各个从节点

struct clusterNode \*\*slaves; /\* pointers to slave nodes \*/

// 如果这是一个从节点，那么指向主节点

struct clusterNode \*slaveof; /\* pointer to the master node \*/

...

};

/\* clusterLink encapsulates everything needed to talk with a remote node. \*/

// clusterLink 包含了与其他节点进行通讯所需的全部信息

typedef struct clusterLink {

...

// TCP 套接字描述符

int fd; /\* TCP socket file descriptor \*/

// 与这个连接相关联的节点，如果没有的话就为 NULL

struct clusterNode \*node; /\* Node related to this link if any, or NULL \*/

...

} clusterLink;

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72

## 4.6 处理流程全梳理

在单机模式下，Redis对请求的处理很简单。Key存在的话，就执行请求中的操作；Key不存在的话，就告诉客户端Key不存在。然而在集群模式下，**因为涉及到请求重定向和Slot迁移，所以对请求的处理变得很复杂**，流程如下：

1. 检查Key所在Slot是否属于当前Node？   
   2.1 计算crc16(key) % 16384得到Slot   
   2.2 **查询clusterState.slots负责Slot的结点指针**   
   2.3 与myself指针比较
2. 若不属于，则响应MOVED错误重定向客户端
3. 若属于且Key存在，则直接操作，返回结果给客户端
4. 若Key不存在，检查该Slot是否迁出中？(**clusterState.migrating\_slots\_to**)
5. 若Slot迁出中，返回ASK错误重定向客户端到迁移的目的服务器上
6. 若Slot未迁出，检查Slot是否导入中？(**clusterState.importing\_slots\_from**)
7. 若Slot导入中且有ASKING标记，则直接操作
8. 否则响应MOVED错误重定向客户端

# 5.应用案例收集

## 5.1 有道：Redis Cluster使用经验

详情请参见[原文](http://www.open-open.com/lib/view/open1429583033948.html)，关键内容摘录如下：

### 5.1.1 两个缺点

“redis cluster的设计在这块有点奇葩，跟集群相关的操作需要一个外部的ruby脚本来协助（当然可能是为了让主程序的代码足够简洁？），然后那个脚本还只支持填实例的ip不支持host，还不告诉你不支持让你用host之后各种莫名其妙。”

“第一个缺点就是严格依赖客户端driver的成熟度。如果把redis cluster设计成类似Cassandra，请求集群中任何一个节点都可以负责转发请求，client会好写一些。”

“第二个缺点完全是设计问题了，就是一个redis进程既负责读写数据又负责集群交互，虽然设计者已经尽可能简化了代码和逻辑，但还是让redis从一个内存NoSQL变成了一个分布式NoSQL。分布式系统很容易有坑，一旦有坑必须升级redis。”

### 5.1.2 去中心化 vs. Proxy

“关于redis cluster的设计，Gossip/P2P的去中心化架构本身不是问题，但一旦有了中心节点，能做的事情就多了，比如sharding不均匀是很容易自动rebalance的，而无中心的只能靠外界来搞。然后redis cluster又是slot的形式而非C\*式的一致性哈希，新节点分slot又不自动，依赖外界（ruby脚本）来分配显得不方便更不优美和谐。而且因为是master-slave的系统而非W+R>N的那种，master挂掉之后尽快发现是比较重要的，gossip对于节点挂掉的发现终究没有中心节点/zookeeper方便快速。”

“基于proxy做转发意味着屏蔽了下层存储，完全可以根据前缀/tag/冷热程度，来把部分甚至大多数数据放在磁盘从而节约成本又保证一致性，这都是有中心节点所带来的好处。”

## 5.2 奇虎360：Redis Cluster浅析和Bada对比

详情请参见[原文](http://chuansong.me/n/1271889)，关键内容摘录如下：

### 5.2.1 负载均衡问题

“redis cluster的主备是以节点为单位，而bada则是以partition为单位，这样，同样是3个节点，1024个partition的情况下，redis cluster的主节点负责整个1024个partition的服务，**而两个从节点则只负责异步备份，导致集群负载不均**，再看bada，将1024个partition的主均分到3个节点中，每个节点各有主备，主对外提供服务，这样均分了访问压力，有效的利用了资源。”

### 5.2.2 一致性的保证

“**redis cluster与bada一样，最终一致性**，读写都只请求主节点，当一条写请求在对应的主节点写成功后，会立刻返回给客户端成功，然后主节点通过异步的方式将新的数据同步到对应的从节点，这样的方式减少了客户端多个节点写成功等待的时间，不过在某些情况下会造成写丢失：

1）当主节点接受一条写请求，写入并返回给客户端成功后不幸宕掉，此时刚才的写还未同步给其对应的从节点，而从节点在发现主节点挂掉并重新选主后，新的主节点则永久丢失了之前老的主节点向用户确认的写

2）当网络发生割裂，将集群分裂成少数派与多数派，这样在客户端不知情的情况下，会将写继续写入到少数派中的某些主节点中，而当割裂超过一定时长后，集群感知到异常，此时少数派中的所有主节点会停止响应所有的写请求，多数派的其对应的从节点则会发起选举成为新的主节点，假设过了一会后割裂恢复，老的主节点发现有更新的主存在，自动变成其从节点，而新的主节点中则会永久丢失掉网络割裂至集群感知异常进行切主这个阶段老主节点确认的所有写

**相对于redis cluster的永久丢失，bada通过binlog merge有效的解决了这一问题**。所有partition的主节点在响应客户端的写请求时，都会在本地记录binlog，binlog实质就是带有时间戳的KV对。当老主以从节点的身份重新加入集群时，会触发binlog merge操作，新主会比较并且合并二者的binlog，这样就可以将之前丢失掉得写再补回来。”

### 5.2.3 请求重定向问题

“bada服务端节点在收到本不该由自己负责的Partition请求后，不会向客户端返回重定向信息，而是通过代理的方式，直接在集群内部向正确节点转发客户端的请求，并将结果同meta信息再转发回客户端。”

“再看multi key操作，**redis cluster为了追求高性能，支持multi key的前提是所有的key必须在同一个节点中**, 不过这样的处理需要交给用户，对需要进行multi key操作的所有key，在写入前人为的加上hash tags。当redis cluster进行resharding的时候，也就是将某些slot从一个节点迁移到另一个节点时，此时的multi key操作可能会失败，因为在迁移的slot中的key此时存在于两个节点。

bada怎么做呢？用户如果对multi key操作性能很在乎时，可以采用与redis cluster同样的方式，给这些key加上hash tags来让它们落在同一个节点，如果可以接受性能的稍微损耗而解放用户的处理逻辑，则可以像single key操作一样，**请求任一bada节点，它会代理所有的key请求并将结果返回给用户。并且在multi key操作在任何时候都可以，即使在进行partition的迁移**，bada也会提前进行切主，保证服务的正常提供。”

## 5.3 芒果TV：Redis服务解决方案

详情请参见[原文](http://my.oschina.net/neuront/blog/377568)，关键内容摘录如下：

芒果TV在Redis Cluster基础上进行开发，主要增加了两个组件：

* **监控管理**：以Python为主要开发框架的Web应用程序[Redis-ctl](https://github.com/HunanTV/redis-ctl)
* **请求代理**：以C++11为开发语言的轻量数据代理程序[cerberus](https://github.com/HunanTV/redis-cerberus)。其作用和优点为：   
  + 集群代理程序的自动请求分发/重试机制使得应用不必修改自身代码或更新Redis库
  + 代理节点为所有Redis节点加上统一管理和状态监测, 可以查阅历史数据, 或在发生任何问题之后快速响应修复
  + 代理进程的无状态性使之可在故障后快速恢复, 不影响后端集群数据完整性

这两个组件都已开源到GitHub上，大家可以关注一下！

# 6.Pros & Cons总结

关于Redis Cluster带来的种种优势就不说了，在这里主要是“鸡蛋里挑骨头”，总结一下现阶段集群功能的欠缺之处和可能的“坑”。

## 6.1 无中心化架构

### 6.1.1 Gossip消息

Gossip消息的网络开销和时延是决定Redis Cluster能够线性扩展的因素之一。关于这个问题，在[《redis cluster百万QPS的挑战》](http://itindex.net/detail/54098-redis-cluster-%E7%99%BE%E4%B8%87)一文中有所提及。

### 6.1.2 结点粒度备份

此外，Redis Cluster也许是为了简化设计采用了Master-Slave复制的数据备份方案，并没有采取如Cassandra或IMDG等对等分布式系统中常见的Slot粒度（或叫Partition/Bucket等）的自动冗余和指派。

这种设计虽然避免比较复杂的分布式技术，但也带来了一些问题：

* **Slave完全闲置**：即便是读请求也不会被重定向到Slave结点上，Slave属于“冷备”
* **写压力无法分摊**：Slave闲置导致的另一个问题就是写压力也都在Master上

## 6.2 客户端的挑战

由于Redis Cluster的设计，客户端要担负起一部分责任：

* **Cluster协议支持**：不管Dummy还是Smart模式，都要具备解析Cluster协议的能力
* **网络开销**：Dummy客户端不断重定向的网络开销
* **连接维护**：Smart客户端对连接到集群中每个结点Socket的维护
* **缓存路由表**：Smart客户端Slot路由表的缓存和更新
* **内存消耗**：Smart客户端上述维护的信息都是有内存消耗的
* **MultiOp有限支持**：对于MultiOp，由客户端通过KeyTag保证所有Key都在同一Slot。而即便如此，迁移时也会导致MultiOp失败。同理，对Pipeline和Transaction的支持也受限于必须操作同一Slot内的Key。

## 6.3 Redis实现问题

尽管属于无中心化架构一类的分布式系统，但不同产品的细节实现和代码质量还是有不少差异的，就比如Redis Cluster有些地方的设计看起来就有一些“奇葩”和简陋：

* **不能自动发现**：无Auto Discovery功能。集群建立时以及运行中新增结点时，都要通过手动执行MEET命令或redis-trib.rb脚本添加到集群中
* **不能自动Resharding**：不仅不自动，连Resharding算法都没有，要自己计算从哪些结点上迁移多少Slot，然后还是得通过redis-trib.rb操作
* **严重依赖外部redis-trib**：如上所述，像集群健康状况检查、结点加入、Resharding等等功能全都抽离到一个Ruby脚本中了。还不清楚上面提到的缺失功能未来是要继续加到这个脚本里还是会集成到集群结点中？redis-trib也许要变成Codis中Dashboard的角色
* **无监控管理UI**：即便未来加了UI，像迁移进度这种信息在无中心化设计中很难得到
* **只保证最终一致性**：写Master成功后立即返回，**如需强一致性，自行通过WAIT命令实现**。但对于“脑裂”问题，目前Redis没提供网络恢复后的Merge功能，“脑裂”期间的更新可能丢失

## 6.4 性能损耗

由于之前手头没有空闲的物理机资源，所以只在虚拟机上做了简单的单机测试，在单独的一台压力机使用YCSB测试框架向虚拟机产生读写负载。虚拟机的配置为8核Intel Xeon CPU X5650@2.67GHz，16GB内存，分别搭建了4结点的单机版Redis和集群版Redis，测试一下Redis Cluster的性能损耗。由于不是最近做的测试，所以Jedis用的2.6.2版本。注：当然Redis Cluster可以通过多机部署获得水平扩展带来的性能提升，这里只是由于环境有限所以做的简单单机测试。

由于YCSB本身仅支持Redis单机版，所以需要我们自己增加扩展插件，具体方法请参照[《YCSB性能测试工具使用》](http://blog.csdn.net/dc_726/article/details/43991871)。通过YCSB产生2000w随机数据，Value大约100Byte左右。然后通过YCSB测试Read-Mostly(90% Read)和Read-Write-Mixed(50% Read)两种情况：

* **数据加载**：吞吐量上有约18%的下降。
* **Read-Mostly**：吞吐量上有约3.5%~7.9%的下降。
* **Read-Write-Mixed**：吞吐量上有约3.3%~5.5%下降。
* **内存占用**：Jedis客户端多占用380MB内存。

## 6.5 最后的总结

从现阶段看来，相比Sentinel或Codis等方案，Redis Cluster的优势还真是有限，个人觉得最大的优点有两个：

1. 官方提供的Slot实现而不用像Codis那样去改源码了；
2. 不用额外的Sentinel集群或类似的代码实现了。

同其他分布式系统，如Cassandra，或内存型的IMDG如Hazelcast和GridGain，除了性能方面外，从功能上Redis Cluster简直被爆得体无完肤… 看看我之前总结过的GridGain介绍[《开源IMDG之GridGain》](http://blog.csdn.net/dc_726/article/details/42615299)：

* 结点自动发现和Rebalance
* 分区粒度的备份
* 故障时分区角色自动调整
* 结果聚合（不会重定向客户端）
* “脑裂”恢复后的Merge（Hazelcast支持多种合并策略）
* 多Primary分区写操作（见Replicated模式）

**这些都是Redis Cluster没有或者要手动完成的。当然这也不足为奇，因为这与Redis的设计初衷有关**，毕竟作者都已经说了，最核心的设计目标就是性能、水平伸缩和可用性。

从Redis Cluster的环境搭建使用到高级功能和内部原理剖析，再到应用案例收集和优缺点的分析罗列，讲了这么多，关于Redis集群到底如何，相信大家根据自己切身和项目的具体情况一定有了自己的结论。不管是评估测试也好，二次开发也好，还是直接上线使用也好，相信随着官方的不断迭代更新和大家的力量，Redis Cluster一定会逐渐完善成熟的！