原文地址：<http://redisdoc.com/topic/cluster-tutorial.html>

# 集群教程

本文档翻译自 <http://redis.io/topics/cluster-tutorial> 。

本文档是 Redis 集群的入门教程， 从用户的角度介绍了设置、测试和操作集群的方法。

本教程不包含晦涩难懂的分布式概念， 也没有像 [Redis 集群规范](http://redisdoc.com/topic/cluster-spec.html#cluster-spec) 那样包含 Redis 集群的实现细节， 如果你打算深入地学习 Redis 集群的部署方法， 那么推荐你在阅读完这个教程之后， 再去看一看集群规范。

**Redis 集群目前仍处于 Alpha 测试版本**， 如果在使用过程中发现任何问题， 请到 [Redis 的邮件列表](https://groups.google.com/forum/?fromgroups#!forum/redis-db) 发贴， 或者到 [Redis 的 Github 页面](https://github.com/antirez/redis) 报告错误。

## 集群简介

Redis 集群是一个可以**在多个 Redis 节点之间进行数据共享**的设施（installation）。

Redis 集群不支持那些需要同时处理多个键的 Redis 命令， 因为执行这些命令需要在多个 Redis 节点之间移动数据， 并且在高负载的情况下， 这些命令将降低 Redis 集群的性能， 并导致不可预测的行为。

Redis 集群**通过分区（partition）来提供一定程度的可用性**（availability）： 即使集群中有一部分节点失效或者无法进行通讯， 集群也可以继续处理命令请求。

Redis 集群提供了以下两个好处：

* 将数据自动切分（split）到多个节点的能力。
* 当集群中的一部分节点失效或者无法进行通讯时， 仍然可以继续处理命令请求的能力。

## Redis 集群数据共享

Redis 集群使用数据分片（sharding）而非一致性哈希（consistency hashing）来实现： 一个 Redis 集群包含 16384 个哈希槽（hash slot）， 数据库中的每个键都属于这 16384 个哈希槽的其中一个， 集群使用公式 CRC16(key) % 16384 来计算键 key 属于哪个槽， 其中CRC16(key) 语句用于计算键 key 的 [CRC16 校验和](http://zh.wikipedia.org/wiki/%E5%BE%AA%E7%92%B0%E5%86%97%E9%A4%98%E6%A0%A1%E9%A9%97) 。

集群中的每个节点负责处理一部分哈希槽。 举个例子， 一个集群可以有三个哈希槽， 其中：

* 节点 A 负责处理 0 号至 5500 号哈希槽。
* 节点 B 负责处理 5501 号至 11000 号哈希槽。
* 节点 C 负责处理 11001 号至 16384 号哈希槽。

这种将哈希槽分布到不同节点的做法使得用户可以很容易地向集群中添加或者删除节点。 比如说：

* 如果用户将新节点 D 添加到集群中， 那么集群只需要将节点 A 、B 、 C 中的某些槽移动到节点 D 就可以了。
* 与此类似， 如果用户要从集群中移除节点 A ， 那么集群只需要将节点 A 中的所有哈希槽移动到节点 B 和节点 C ， 然后再移除空白（不包含任何哈希槽）的节点 A 就可以了。

因为将一个哈希槽从一个节点移动到另一个节点不会造成节点阻塞， 所以无论是添加新节点还是移除已存在节点， 又或者改变某个节点包含的哈希槽数量， 都不会造成集群下线。

## Redis 集群中的主从复制

为了使得集群在一部分节点下线或者无法与集群的大多数（majority）节点进行通讯的情况下， 仍然可以正常运作， Redis 集群对节点使用了主从复制功能： 集群中的每个节点都有 1 个至 N 个复制品（replica）， 其中一个复制品为主节点（master）， 而其余的 N-1个复制品为从节点（slave）。

在之前列举的节点 A 、B 、C 的例子中， 如果节点 B 下线了， 那么集群将无法正常运行， 因为集群找不到节点来处理 5501 号至11000 号的哈希槽。

另一方面， 假如在创建集群的时候（或者至少在节点 B 下线之前）， 我们为主节点 B 添加了从节点 B1 ， 那么当主节点 B 下线的时候， 集群就会将 B1 设置为新的主节点， 并让它代替下线的主节点 B ， 继续处理 5501 号至 11000 号的哈希槽， 这样集群就不会因为主节点 B 的下线而无法正常运作了。

不过如果节点 B 和 B1 都下线的话， Redis 集群还是会停止运作。

## Redis 集群的一致性保证（guarantee）

Redis 集群**不保证数据的强一致性**（strong consistency）： 在特定条件下， Redis 集群可能会丢失已经被执行过的写命令。

使用异步复制（asynchronous replication）是 Redis 集群可能会丢失写命令的其中一个原因。 考虑以下这个写命令的例子：

* 客户端向主节点 B 发送一条写命令。
* 主节点 B 执行写命令，并向客户端返回命令回复。
* 主节点 B 将刚刚执行的写命令复制给它的从节点 B1 、 B2 和 B3 。

如你所见， 主节点对命令的复制工作发生在返回命令回复之后， 因为如果每次处理命令请求都需要等待复制操作完成的话， 那么主节点处理命令请求的速度将极大地降低 —— 我们必须在性能和一致性之间做出权衡。

如果真的有必要的话， Redis 集群可能会在将来提供同步地（synchronou）执行写命令的方法。

Redis 集群另外一种可能会丢失命令的情况是， 集群出现网络分裂（[network partition](http://en.wikipedia.org/wiki/Network_partition)）， 并且一个客户端与至少包括一个主节点在内的少数（minority）实例被孤立。

举个例子， 假设集群包含 A 、 B 、 C 、 A1 、 B1 、 C1 六个节点， 其中 A 、B 、C 为主节点， 而 A1 、B1 、C1 分别为三个主节点的从节点， 另外还有一个客户端 Z1 。

假设集群中发生网络分裂， 那么集群可能会分裂为两方， 大多数（majority）的一方包含节点 A 、C 、A1 、B1 和 C1 ， 而少数（minority）的一方则包含节点 B 和客户端 Z1 。

在网络分裂期间， 主节点 B 仍然会接受 Z1 发送的写命令：

* 如果网络分裂出现的时间很短， 那么集群会继续正常运行；
* 但是， 如果网络分裂出现的时间足够长， 使得大多数一方将从节点 B1 设置为新的主节点， 并使用 B1 来代替原来的主节点 B ， 那么 Z1 发送给主节点 B 的写命令将丢失。

注意， 在网络分裂出现期间， 客户端 Z1 可以向主节点 B 发送写命令的最大时间是有限制的， 这一时间限制称为**节点超时时间**（node timeout）， 是 Redis 集群的一个重要的配置选项：

* 对于大多数一方来说， 如果一个主节点未能在节点超时时间所设定的时限内重新联系上集群， 那么集群会将这个主节点视为下线， 并使用从节点来代替这个主节点继续工作。
* 对于少数一方， 如果一个主节点未能在节点超时时间所设定的时限内重新联系上集群， 那么它将停止处理写命令， 并向客户端报告错误。

## 创建并使用 Redis 集群

Redis 集群由多个运行在集群模式（cluster mode）下的 Redis 实例组成， 实例的集群模式需要通过配置来开启， 开启集群模式的实例将可以使用集群特有的功能和命令。

以下是一个包含了最少选项的集群配置文件示例：

port 7000

cluster-enabled yes

cluster-config-file nodes.conf

cluster-node-timeout 5000

appendonly yes

文件中的 cluster-enabled 选项用于开实例的集群模式， 而 cluster-conf-file 选项则设定了保存节点配置文件的路径， 默认值为nodes.conf 。

节点配置文件无须人为修改， 它由 Redis 集群在启动时创建， 并在有需要时自动进行更新。

**要让集群正常运作至少需要三个主节点**， 不过在刚开始试用集群功能时， 强烈建议使用六个节点： 其中三个为主节点， 而其余三个则是各个主节点的从节点。

首先， 让我们进入一个新目录， 并创建六个以端口号为名字的子目录， 稍后我们在将每个目录中运行一个 Redis 实例：

mkdir cluster-test

cd cluster-test

mkdir 7000 7001 7002 7003 7004 7005

在文件夹 7000 至 7005 中， 各创建一个 redis.conf 文件， 文件的内容可以使用上面的示例配置文件， 但记得将配置中的端口号从 7000改为与文件夹名字相同的号码。

现在， 从 [Redis Github 页面](https://github.com/antirez/redis) 的 unstable 分支中取出最新的 Redis 源码， 编译出可执行文件 redis-server ， 并将文件复制到cluster-test 文件夹， 然后使用类似以下命令， 在每个标签页中打开一个实例：

cd 7000

../redis-server ./redis.conf

实例打印的日志显示， 因为 nodes.conf 文件不存在， 所以每个节点都为它自身指定了一个新的 ID ：

[82462] 26 Nov 11:56:55.329 \* No cluster configuration found, I'm 97a3a64667477371c4479320d683e4c8db5858b1

实例会一直使用同一个 ID ， 从而在集群中保持一个独一无二（unique）的名字。

每个节点都使用 ID 而不是 IP 或者端口号来记录其他节点， 因为 IP 地址和端口号都可能会改变， 而这个独一无二的标识符（identifier）则会在节点的整个生命周期中一直保持不变。

我们将这个标识符称为**节点 ID**。

## 创建集群

现在我们已经有了六个正在运行中的 Redis 实例， 接下来我们需要使用这些实例来创建集群， 并为每个节点编写配置文件。

通过使用 Redis 集群命令行工具 redis-trib ， 编写节点配置文件的工作可以非常容易地完成： redis-trib 位于 Redis 源码的 src 文件夹中， 它是一个 Ruby 程序， 这个程序通过向实例发送特殊命令来完成创建新集群， 检查集群， 或者对集群进行重新分片（reshared）等工作。

我们需要执行以下命令来创建集群：

./redis-trib.rb create --replicas 1 127.0.0.1:7000 127.0.0.1:7001 \

127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005

命令的意义如下：

* 给定 redis-trib.rb 程序的命令是 create ， 这表示我们希望创建一个新的集群。
* 选项 --replicas 1 表示我们希望为集群中的每个主节点创建一个从节点。
* 之后跟着的其他参数则是实例的地址列表， 我们希望程序使用这些地址所指示的实例来创建新集群。

简单来说， 以上命令的意思就是让 redis-trib 程序创建一个包含三个主节点和三个从节点的集群。

接着， redis-trib 会打印出一份预想中的配置给你看， 如果你觉得没问题的话， 就可以输入 yes ， redis-trib 就会将这份配置应用到集群当中：

**>>>** Creating cluster

Connecting to node 127.0.0.1:7000: OK

Connecting to node 127.0.0.1:7001: OK

Connecting to node 127.0.0.1:7002: OK

Connecting to node 127.0.0.1:7003: OK

Connecting to node 127.0.0.1:7004: OK

Connecting to node 127.0.0.1:7005: OK

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

Using 3 masters:

127.0.0.1:7000

127.0.0.1:7001

127.0.0.1:7002

127.0.0.1:7000 replica #1 is 127.0.0.1:7003

127.0.0.1:7001 replica #1 is 127.0.0.1:7004

127.0.0.1:7002 replica #1 is 127.0.0.1:7005

M: 9991306f0e50640a5684f1958fd754b38fa034c9 127.0.0.1:7000

slots:0-5460 (5461 slots) master

M: e68e52cee0550f558b03b342f2f0354d2b8a083b 127.0.0.1:7001

slots:5461-10921 (5461 slots) master

M: 393c6df5eb4b4cec323f0e4ca961c8b256e3460a 127.0.0.1:7002

slots:10922-16383 (5462 slots) master

S: 48b728dbcedff6bf056231eb44990b7d1c35c3e0 127.0.0.1:7003

S: 345ede084ac784a5c030a0387f8aaa9edfc59af3 127.0.0.1:7004

S: 3375be2ccc321932e8853234ffa87ee9fde973ff 127.0.0.1:7005

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

输入 yes 并按下回车确认之后， 集群就会将配置应用到各个节点， 并连接起（join）各个节点 —— 也即是， 让各个节点开始互相通讯：

**>>>** Nodes configuration updated

**>>>** Sending CLUSTER MEET messages to join the cluster

Waiting for the cluster to join...

**>>>** Performing Cluster Check (using node 127.0.0.1:7000)

M: 9991306f0e50640a5684f1958fd754b38fa034c9 127.0.0.1:7000

slots:0-5460 (5461 slots) master

M: e68e52cee0550f558b03b342f2f0354d2b8a083b 127.0.0.1:7001

slots:5461-10921 (5461 slots) master

M: 393c6df5eb4b4cec323f0e4ca961c8b256e3460a 127.0.0.1:7002

slots:10922-16383 (5462 slots) master

M: 48b728dbcedff6bf056231eb44990b7d1c35c3e0 127.0.0.1:7003

slots: (0 slots) master

M: 345ede084ac784a5c030a0387f8aaa9edfc59af3 127.0.0.1:7004

slots: (0 slots) master

M: 3375be2ccc321932e8853234ffa87ee9fde973ff 127.0.0.1:7005

slots: (0 slots) master

[OK] All nodes agree about slots configuration.

如果一切正常的话， redis-trib 将输出以下信息：

**>>>** Check **for** open slots...

**>>>** Check slots coverage...

[OK] All 16384 slots covered.

这表示集群中的 16384 个槽都有至少一个主节点在处理， 集群运作正常。

## 集群的客户端

Redis 集群现阶段的一个问题是客户端实现很少。 以下是一些我知道的实现：

* redis-rb-cluster 是我（@antirez）编写的 Ruby 实现， 用于作为其他实现的参考。 该实现是对 redis-rb 的一个简单包装， 高效地实现了与集群进行通讯所需的最少语义（semantic）。
* redis-py-cluster 看上去是 redis-rb-cluster 的一个 Python 版本， 这个项目有一段时间没有更新了（最后一次提交是在六个月之前）， 不过可以将这个项目用作学习集群的起点。
* 流行的 Predis 曾经对早期的 Redis 集群有过一定的支持， 但我不确定它对集群的支持是否完整， 也不清楚它是否和最新版本的 Redis 集群兼容 （因为新版的 Redis 集群将槽的数量从 4k 改为 16k 了）。
* Redis unstable 分支中的 redis-cli 程序实现了非常基本的集群支持， 可以使用命令 redis-cli -c 来启动。

测试 Redis 集群比较简单的办法就是使用 redis-rb-cluster 或者 redis-cli ， 接下来我们将使用 redis-cli 为例来进行演示：

$ redis-cli -c -p 7000

redis 127.0.0.1:7000> set foo bar

-> Redirected to slot [12182] located at 127.0.0.1:7002

OK

redis 127.0.0.1:7002> set hello world

-> Redirected to slot [866] located at 127.0.0.1:7000

OK

redis 127.0.0.1:7000> get foo

-> Redirected to slot [12182] located at 127.0.0.1:7002

"bar"

redis 127.0.0.1:7000> get hello

-> Redirected to slot [866] located at 127.0.0.1:7000

"world"

redis-cli 对集群的支持是非常基本的， 所以它总是依靠 Redis 集群节点来将它转向（redirect）至正确的节点。

一个真正的（serious）集群客户端应该做得比这更好： 它应该用缓存记录起哈希槽与节点地址之间的映射（map）， 从而直接将命令发送到正确的节点上面。

这种映射只会在集群的配置出现某些修改时变化， 比如说， 在一次故障转移（failover）之后， 或者系统管理员通过添加节点或移除节点来修改了集群的布局（layout）之后， 诸如此类。

## 使用 redis-rb-cluster 编写一个示例应用

在展示如何使用集群进行故障转移、重新分片等操作之前， 我们需要创建一个示例应用， 了解一些与 Redis 集群客户端进行交互的基本方法。

在运行示例应用的过程中， 我们会尝试让节点进入失效状态， 又或者开始一次重新分片， 以此来观察 Redis 集群在真实世界运行时的表现， 并且为了让这个示例尽可能地有用， 我们会让这个应用向集群进行写操作。

本节将通过两个示例应用来展示 redis-rb-cluster 的基本用法， 以下是本节的第一个示例应用， 它是一个名为 [example.rb](https://github.com/antirez/redis-rb-cluster/blob/master/cluster.rb) 的文件， 包含在[redis-rb-cluster 项目里面](https://github.com/antirez/redis-rb-cluster)：

|  |  |
| --- | --- |
| 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 | require './cluster'  startup\_nodes = [  {:host => "127.0.0.1", :port => 7000},  {:host => "127.0.0.1", :port => 7001}  ]  rc = RedisCluster.new(startup\_nodes,32,:timeout => 0.1)  last = false  **while** **not** last  **begin**  last = rc.get("\_\_last\_\_")  last = 0 **if** !last  **rescue** => e  puts "error *#{*e.to\_s*}*"  sleep 1  **end**  **end**  ((last.to\_i+1)..1000000000).each{|x|  **begin**  rc.set("foo*#{*x*}*",x)  puts rc.get("foo*#{*x*}*")  rc.set("\_\_last\_\_",x)  **rescue** => e  puts "error *#{*e.to\_s*}*"  **end**  sleep 0.1  } |

这个应用所做的工作非常简单： 它不断地以 foo<number> 为键， number 为值， 使用 [SET](http://redisdoc.com/string/set.html#set) 命令向数据库设置键值对。

如果我们执行这个应用的话， 应用将按顺序执行以下命令：

* SET foo0 0
* SET foo1 1
* SET foo2 2
* 诸如此类。。。

代码中的每个集群操作都使用一个 begin 和 rescue 代码块（block）包裹着， 因为我们希望在代码出错时， 将错误打印到终端上面， 而不希望应用因为异常（exception）而退出。

代码的**第七行**是代码中第一个有趣的地方， 它创建了一个 Redis 集群对象， 其中创建对象所使用的参数及其意义如下：

* 第一个参数是记录了启动节点的 startup\_nodes 列表， 列表中包含了两个集群节点的地址。
* 第二个参数指定了对于集群中的各个不同的节点， Redis 集群对象可以获得（take）的最大连接数 （maximum number of connections this object is allowed to take）。
* 第三个参数 timeout 指定了一个命令在执行多久之后， 才会被看作是执行失败。

记住， 启动列表中并不需要包含所有集群节点的地址， 但这些地址中至少要有一个是有效的（reachable）： 一旦 redis-rb-cluster 成功连接上集群中的某个节点时， 集群节点列表就会被自动更新， 任何真正的（serious）的集群客户端都应该这样做。

现在， 程序创建的 Redis 集群对象实例被保存到 rc 变量里面， 我们可以将这个对象当作普通 Redis 对象实例来使用。

在**十一至十九行**， 我们先尝试阅读计数器中的值， 如果计数器不存在的话， 我们才将计数器初始化为 0 ： 通过将计数值保存到 Redis 的计数器里面， 我们可以在示例重启之后， 仍然继续之前的执行过程， 而不必每次重启之后都从 foo0 开始重新设置键值对。

为了让程序在集群下线的情况下， 仍然不断地尝试读取计数器的值， 我们将读取操作包含在了一个 while 循环里面， 一般的应用程序并不需要如此小心。

**二十一至三十行**是程序的主循环， 这个循环负责设置键值对， 并在设置出错时打印错误信息。

程序在主循环的末尾添加了一个 sleep 调用， 让写操作的执行速度变慢， 帮助执行示例的人更容易看清程序的输出。

执行 example.rb 程序将产生以下输出：

ruby ./example.rb

1

2

3

4

5

6

7

8

9

...

这个程序并不是十分有趣， 稍后我们就会看到一个更有趣的集群应用示例， 不过在此之前， 让我们先使用这个示例来演示集群的重新分片操作。

## 对集群进行重新分片

现在， 让我们来试试对集群进行重新分片操作。

在执行重新分片的过程中， 请让你的 example.rb 程序处于运行状态， 这样你就会看到， 重新分片并不会对正在运行的集群程序产生任何影响， 你也可以考虑将 example.rb 中的 sleep 调用删掉， 从而让重新分片操作在近乎真实的写负载下执行。

重新分片操作基本上就是将某些节点上的哈希槽移动到另外一些节点上面， 和创建集群一样， 重新分片也可以使用 redis-trib 程序来执行。

执行以下命令可以开始一次重新分片操作：

$ ./redis-trib.rb reshard 127.0.0.1:7000

你只需要指定集群中其中一个节点的地址， redis-trib 就会自动找到集群中的其他节点。

目前 redis-trib 只能在管理员的协助下完成重新分片的工作， 要让 redis-trib 自动将哈希槽从一个节点移动到另一个节点， 目前来说还做不到 （不过实现这个功能并不难）。

执行 redis-trib 的第一步就是设定你打算移动的哈希槽的数量：

$ ./redis-trib.rb reshard 127.0.0.1:7000

Connecting to node 127.0.0.1:7000: OK

Connecting to node 127.0.0.1:7002: OK

Connecting to node 127.0.0.1:7005: OK

Connecting to node 127.0.0.1:7001: OK

Connecting to node 127.0.0.1:7003: OK

Connecting to node 127.0.0.1:7004: OK

>>> Performing Cluster Check (using node 127.0.0.1:7000)

M: 9991306f0e50640a5684f1958fd754b38fa034c9 127.0.0.1:7000

slots:0-5460 (5461 slots) master

M: 393c6df5eb4b4cec323f0e4ca961c8b256e3460a 127.0.0.1:7002

slots:10922-16383 (5462 slots) master

S: 3375be2ccc321932e8853234ffa87ee9fde973ff 127.0.0.1:7005

slots: (0 slots) slave

M: e68e52cee0550f558b03b342f2f0354d2b8a083b 127.0.0.1:7001

slots:5461-10921 (5461 slots) master

S: 48b728dbcedff6bf056231eb44990b7d1c35c3e0 127.0.0.1:7003

slots: (0 slots) slave

S: 345ede084ac784a5c030a0387f8aaa9edfc59af3 127.0.0.1:7004

slots: (0 slots) slave

[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)? 1000

我们将打算移动的槽数量设置为 1000 个， 如果 example.rb 程序一直运行着的话， 现在 1000 个槽里面应该有不少键了。

除了移动的哈希槽数量之外， redis-trib 还需要知道重新分片的目标（target node）， 也即是， 负责接收这 1000 个哈希槽的节点。

指定目标需要使用节点的 ID ， 而不是 IP 地址和端口。 比如说， 我们打算使用集群的第一个主节点来作为目标， 它的 IP 地址和端口是 127.0.0.1:7000 ， 而节点 ID 则是 9991306f0e50640a5684f1958fd754b38fa034c9 ， 那么我们应该向 redis-trib 提供节点的 ID ：

$ ./redis-trib.rb reshard 127.0.0.1:7000

...

What is the receiving node ID? 9991306f0e50640a5684f1958fd754b38fa034c9

redis-trib 会打印出集群中所有节点的 ID ， 并且我们也可以通过执行以下命令来获得节点的运行 ID ：

$ ./redis-cli -p 7000 cluster nodes | grep myself

9991306f0e50640a5684f1958fd754b38fa034c9 :0 myself,master - 0 0 0 connected 0-5460

接着， redis-trib 会向你询问重新分片的源节点（source node）， 也即是， 要从哪个节点中取出 1000 个哈希槽， 并将这些槽移动到目标节点上面。

如果我们不打算从特定的节点上取出指定数量的哈希槽， 那么可以向 redis-trib 输入 all ， 这样的话， 集群中的所有主节点都会成为源节点， redis-trib 将从各个源节点中各取出一部分哈希槽， 凑够 1000 个， 然后移动到目标节点上面：

$ ./redis-trib.rb reshard 127.0.0.1:7000

...

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

输入 all 并按下回车之后， redis-trib 将打印出哈希槽的移动计划， 如果你觉得没问题的话， 就可以输入 yes 并再次按下回车：

$ ./redis-trib.rb reshard 127.0.0.1:7000

...

Moving slot 11421 from 393c6df5eb4b4cec323f0e4ca961c8b256e3460a

Moving slot 11422 from 393c6df5eb4b4cec323f0e4ca961c8b256e3460a

Moving slot 5461 from e68e52cee0550f558b03b342f2f0354d2b8a083b

Moving slot 5469 from e68e52cee0550f558b03b342f2f0354d2b8a083b

...

Moving slot 5959 from e68e52cee0550f558b03b342f2f0354d2b8a083b

Do you want to proceed with the proposed reshard plan (yes/no)? yes

输入 yes 并使用按下回车之后， redis-trib 就会正式开始执行重新分片操作， 将指定的哈希槽从源节点一个个地移动到目标节点上面：

$ ./redis-trib.rb reshard 127.0.0.1:7000

...

Moving slot 5934 from 127.0.0.1:7001 to 127.0.0.1:7000:

Moving slot 5935 from 127.0.0.1:7001 to 127.0.0.1:7000:

Moving slot 5936 from 127.0.0.1:7001 to 127.0.0.1:7000:

Moving slot 5937 from 127.0.0.1:7001 to 127.0.0.1:7000:

...

Moving slot 5959 from 127.0.0.1:7001 to 127.0.0.1:7000:

在重新分片的过程中， example.rb 应该可以继续正常运行， 不会出现任何问题。

在重新分片操作执行完毕之后， 可以使用以下命令来检查集群是否正常：

$ ./redis-trib.rb check 127.0.0.1:7000

Connecting to node 127.0.0.1:7000: OK

Connecting to node 127.0.0.1:7002: OK

Connecting to node 127.0.0.1:7005: OK

Connecting to node 127.0.0.1:7001: OK

Connecting to node 127.0.0.1:7003: OK

Connecting to node 127.0.0.1:7004: OK

>>> Performing Cluster Check (using node 127.0.0.1:7000)

M: 9991306f0e50640a5684f1958fd754b38fa034c9 127.0.0.1:7000

slots:0-5959,10922-11422 (6461 slots) master

M: 393c6df5eb4b4cec323f0e4ca961c8b256e3460a 127.0.0.1:7002

slots:11423-16383 (4961 slots) master

S: 3375be2ccc321932e8853234ffa87ee9fde973ff 127.0.0.1:7005

slots: (0 slots) slave

M: e68e52cee0550f558b03b342f2f0354d2b8a083b 127.0.0.1:7001

slots:5960-10921 (4962 slots) master

S: 48b728dbcedff6bf056231eb44990b7d1c35c3e0 127.0.0.1:7003

slots: (0 slots) slave

S: 345ede084ac784a5c030a0387f8aaa9edfc59af3 127.0.0.1:7004

slots: (0 slots) slave

[OK] All nodes agree about slots configuration.

>>> Check for open slots...

>>> Check slots coverage...

[OK] All 16384 slots covered.

根据检查结果显示， 集群运作正常。

需要注意的就是， 在三个主节点中， 节点 127.0.0.1:7000 包含了 6461 个哈希槽， 而节点 127.0.0.1:7001 和节点 127.0.0.1:7002 都只包含了 4961 个哈希槽， 因为后两者都将自己的 500 个哈希槽移动到了节点 127.0.0.1:7000 。

## 一个更有趣的示例应用

我们在前面使用的示例程序 example.rb 并不是十分有趣， 因为它只是不断地对集群进行写入， 但并不检查写入结果是否正确。 比如说， 集群可能会错误地将 example.rb 发送的所有 [SET](http://redisdoc.com/string/set.html#set) 命令都改成了 SET foo 42 ， 但因为 example.rb 并不检查写入后的值， 所以它不会意识到集群实际上写入的值是错误的。

因为这个原因， [redis-rb-cluster 项目](https://github.com/antirez/redis-rb-cluster)包含了一个名为 [consistency-test.rb](https://github.com/antirez/redis-rb-cluster/blob/master/consistency-test.rb) 的示例应用， 这个应用比起 example.rb 有趣得多： 它创建了多个计数器（默认为 1000 个）， 并通过发送 [INCR](http://redisdoc.com/string/incr.html#incr) 命令来增加这些计数器的值。

在增加计数器值的同时， consistency-test.rb 还执行以下操作：

* 每次使用 [INCR](http://redisdoc.com/string/incr.html#incr) 命令更新一个计数器时， 应用会记录下计数器执行 [INCR](http://redisdoc.com/string/incr.html#incr) 命令之后应该有的值。 举个例子， 如果计数器的起始值为 0 ， 而这次是程序第 50 次向它发送 [INCR](http://redisdoc.com/string/incr.html#incr) 命令， 那么计数器的值应该是 50 。
* 在每次发送 [INCR](http://redisdoc.com/string/incr.html#incr) 命令之前， 程序会随机从集群中读取一个计数器的值， 并将它与自己记录的值进行对比， 看两个值是否相同。

换句话说， 这个程序是一个一致性检查器（consistency checker）： 如果集群在执行 [INCR](http://redisdoc.com/string/incr.html#incr) 命令的过程中， 丢失了某条 [INCR](http://redisdoc.com/string/incr.html#incr) 命令， 又或者多执行了某条客户端没有确认到的 [INCR](http://redisdoc.com/string/incr.html#incr) 命令， 那么检查器将察觉到这一点 —— 在前一种情况中， consistency-test.rb 记录的计数器值将比集群记录的计数器值要大； 而在后一种情况中， consistency-test.rb 记录的计数器值将比集群记录的计数器值要小。

运行 consistency-test 程序将产生类似以下的输出：

$ ruby consistency-test.rb

925 R (0 err) | 925 W (0 err) |

5030 R (0 err) | 5030 W (0 err) |

9261 R (0 err) | 9261 W (0 err) |

13517 R (0 err) | 13517 W (0 err) |

17780 R (0 err) | 17780 W (0 err) |

22025 R (0 err) | 22025 W (0 err) |

25818 R (0 err) | 25818 W (0 err) |

每行输出都打印了程序执行的读取次数和写入次数， 以及执行操作的过程中因为集群不可用而产生的错误数。

如果程序察觉了不一致的情况出现， 它将在输出行的末尾显式不一致的详细情况。

比如说， 如果我们在 consistency-test.rb 运行的过程中， 手动修改某个计数器的值：

$ redis 127.0.0.1:7000> set key\_217 0

OK

那么 consistency-test.rb 将向我们报告不一致情况：

(in the other tab I see...)

94774 R (0 err) | 94774 W (0 err) |

98821 R (0 err) | 98821 W (0 err) |

102886 R (0 err) | 102886 W (0 err) | 114 lost |

107046 R (0 err) | 107046 W (0 err) | 114 lost |

在我们修改计数器值的时候， 计数器的正确值是 114 （执行了 114 次 [INCR](http://redisdoc.com/string/incr.html#incr) 命令）， 因为我们将计数器的值设成了 0 ， 所以consistency-test.rb 会向我们报告说丢失了 114 个 [INCR](http://redisdoc.com/string/incr.html#incr) 命令。

因为这个示例程序具有一致性检查功能， 所以我们用它来测试 Redis 集群的故障转移操作。

## 故障转移测试

在执行本节操作的过程中， 请一直运行 consistency-test 程序。

要触发一次故障转移， 最简单的办法就是令集群中的某个主节点进入下线状态。

首先用以下命令列出集群中的所有主节点：

$ redis-cli -p 7000 cluster nodes | grep master

3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 1385482984082 0 connected 5960-10921

2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 master - 0 1385482983582 0 connected 11423-16383

97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 connected 0-5959 10922-11422

通过命令输出， 我们知道端口号为 7000 、 7001 和 7002 的节点都是主节点， 然后我们可以通过向端口号为 7002 的主节点发送 [DEBUG SEGFAULT](http://redisdoc.com/server/debug_segfault.html#debug-segfault) 命令， 让这个主节点崩溃：

$ redis-cli -p 7002 debug segfault

Error: Server closed the connection

现在， 切换到运行着 consistency-test 的标签页， 可以看到， consistency-test 在 7002 下线之后的一段时间里将产生大量的错误警告信息：

18849 R (0 err) | 18849 W (0 err) |

23151 R (0 err) | 23151 W (0 err) |

27302 R (0 err) | 27302 W (0 err) |

... many error warnings here ...

29659 R (578 err) | 29660 W (577 err) |

33749 R (578 err) | 33750 W (577 err) |

37918 R (578 err) | 37919 W (577 err) |

42077 R (578 err) | 42078 W (577 err) |

从 consistency-test 的这段输出可以看到， 集群在执行故障转移期间， 总共丢失了 578 个读命令和 577 个写命令， 但是并没有产生任何数据不一致。

这听上去可能有点奇怪， 因为在教程的开头我们提到过， Redis 使用的是异步复制， 在执行故障转移期间， 集群可能会丢失写命令。

但是在实际上， 丢失命令的情况并不常见， 因为 Redis 几乎是同时执行将命令回复发送给客户端， 以及将命令复制给从节点这两个操作， 所以实际上造成命令丢失的时间窗口是非常小的。

不过， 尽管出现的几率不高， 但丢失命令的情况还是有可能会出现的， 所以我们对 Redis 集群不能提供强一致性的这一描述仍然是正确的。

现在， 让我们使用 cluster nodes 命令， 查看集群在执行故障转移操作之后， 主从节点的布局情况：

$ redis-cli -p 7000 cluster nodes

3fc783611028b1707fd65345e763befb36454d73 127.0.0.1:7004 slave 3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 0 1385503418521 0 connected

a211e242fc6b22a9427fed61285e85892fa04e08 127.0.0.1:7003 slave 97a3a64667477371c4479320d683e4c8db5858b1 0 1385503419023 0 connected

97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 connected 0-5959 10922-11422

3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 127.0.0.1:7005 master - 0 1385503419023 3 connected 11423-16383

3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 1385503417005 0 connected 5960-10921

2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 0 1385503418016 3 connected

我重启了之前下线的 127.0.0.1:7002 节点， 该节点已经从原来的主节点变成了从节点， 而现在集群中的三个主节点分别是 127.0.0.1:7000、 127.0.0.1:7001 和 127.0.0.1:7005 ， 其中 127.0.0.1:7005 就是因为 127.0.0.1:7002 下线而变成主节点的。

cluster nodes 命令的输出有点儿复杂， 它的每一行都是由以下信息组成的：

* 节点 ID ：例如 3fc783611028b1707fd65345e763befb36454d73 。
* ip:port ：节点的 IP 地址和端口号， 例如 127.0.0.1:7000 ， 其中 :0 表示的是客户端当前连接的 IP 地址和端口号。
* flags ：节点的角色（例如 master 、 slave 、 myself ）以及状态（例如 fail ，等等）。
* 如果节点是一个从节点的话， 那么跟在 flags 之后的将是主节点的节点 ID ： 例如 127.0.0.1:7002 的主节点的节点 ID 就是3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 。
* 集群最近一次向节点发送 [PING](http://redisdoc.com/connection/ping.html#ping) 命令之后， 过去了多长时间还没接到回复。
* 节点最近一次返回 PONG 回复的时间。
* 节点的配置纪元（configuration epoch）：详细信息请参考 [Redis 集群规范](http://redisdoc.com/topic/cluster-spec.html#cluster-spec) 。
* 本节点的网络连接情况：例如 connected 。
* 节点目前包含的槽：例如 127.0.0.1:7001 目前包含号码为 5960 至 10921 的哈希槽。

## 添加新节点到集群

根据新添加节点的种类， 我们需要用两种方法来将新节点添加到集群里面：

* 如果要添加的新节点是一个主节点， 那么我们需要创建一个空节点（empty node）， 然后将某些哈希桶移动到这个空节点里面。
* 另一方面， 如果要添加的新节点是一个从节点， 那么我们需要将这个新节点设置为集群中某个节点的复制品（replica）。

本节将对以上两种情况进行介绍， 首先介绍主节点的添加方法， 然后再介绍从节点的添加方法。

无论添加的是那种节点， 第一步要做的总是添加一个空节点。

我们可以继续使用之前启动 127.0.0.1:7000 、 127.0.0.1:7001 等节点的方法， 创建一个端口号为 7006 的新节点， 使用的配置文件也和之前一样， 只是记得要将配置中的端口号改为 7000 。

以下是启动端口号为 7006 的新节点的详细步骤：

1. 在终端里创建一个新的标签页。
2. 进入 cluster-test 文件夹。
3. 创建并进入 7006 文件夹。
4. 将 redis.conf 文件复制到 7006 文件夹里面，然后将配置中的端口号选项改为 7006 。
5. 使用命令 ../../redis-server redis.conf 启动节点。

如果一切正常， 那么节点应该会正确地启动。

接下来， 执行以下命令， 将这个新节点添加到集群里面：

./redis-trib.rb add-node 127.0.0.1:7006 127.0.0.1:7000

命令中的 add-node 表示我们要让 redis-trib 将一个节点添加到集群里面， add-node 之后跟着的是新节点的 IP 地址和端口号， 再之后跟着的是集群中任意一个已存在节点的 IP 地址和端口号， 这里我们使用的是 127.0.0.1:7000 。

通过 cluster nodes 命令， 我们可以确认新节点 127.0.0.1:7006 已经被添加到集群里面了：

redis 127.0.0.1:7006> cluster nodes

3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 1385543178575 0 connected 5960-10921

3fc783611028b1707fd65345e763befb36454d73 127.0.0.1:7004 slave 3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 0 1385543179583 0 connected

f093c80dde814da99c5cf72a7dd01590792b783b :0 myself,master - 0 0 0 connected

2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 0 1385543178072 3 connected

a211e242fc6b22a9427fed61285e85892fa04e08 127.0.0.1:7003 slave 97a3a64667477371c4479320d683e4c8db5858b1 0 1385543178575 0 connected

97a3a64667477371c4479320d683e4c8db5858b1 127.0.0.1:7000 master - 0 1385543179080 0 connected 0-5959 10922-11422

3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 127.0.0.1:7005 master - 0 1385543177568 3 connected 11423-16383

新节点现在已经连接上了集群， 成为集群的一份子， 并且可以对客户端的命令请求进行转向了， 但是和其他主节点相比， 新节点还有两点区别：

* 新节点没有包含任何数据， 因为它没有包含任何哈希桶。
* 尽管新节点没有包含任何哈希桶， 但它仍然是一个主节点， 所以在集群需要将某个从节点升级为新的主节点时， 这个新节点不会被选中。

接下来， 只要使用 redis-trib 程序， 将集群中的某些哈希桶移动到新节点里面， 新节点就会成为真正的主节点了。

因为使用 redis-trib 移动哈希桶的方法在前面已经介绍过， 所以这里就不再重复介绍了。

现在， 让我们来看看， 将一个新节点转变为某个主节点的复制品（也即是从节点）的方法。

举个例子， 如果我们打算让新节点成为 127.0.0.1:7005 的从节点， 那么我们只要用客户端连接上新节点， 然后执行以下命令就可以了：

redis 127.0.0.1:7006> cluster replicate 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e

其中命令提供的 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 就是主节点 127.0.0.1:7005 的节点 ID 。

执行 cluster replicate 命令之后， 我们可以使用以下命令来确认 127.0.0.1:7006 已经成为了 ID 为 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e的节点的从节点：

$ redis-cli -p 7000 cluster nodes | grep slave | grep 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e

f093c80dde814da99c5cf72a7dd01590792b783b 127.0.0.1:7006 slave 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 0 1385543617702 3 connected

2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 0 1385543617198 3 connected

3c3a0c... 现在有两个从节点， 一个从节点的端口号为 7002 ， 而另一个从节点的端口号为 7006 。

## 移除一个节点

未完待续。