翻译自官方文档 [Redis cluster tutorial](http://redis.io/topics/cluster-tutorial)

http://blog.csdn.net/nsrainbow/article/details/49032337

Redis 集群教程

该文档是一篇关于redis集群的教程。该教程并不会让你去理解复杂分布式系统概念，只会告诉你如何安装、测试和操作一个集群。在这个过程中该文档只会从用户角度去描述系统的行为，并不会详细的探究[Redis集群手册](http://redis.io/topics/cluster-spec)中的内容。

该教程会努力的从最终用户的视角出发， 来介绍Redis集群的可用性和一致性这些特性。不过请放心，我们会用浅显易懂的方式来介绍这一切。

注意：该教程需要Redis 的版本高于或者等于 3.0

虽然我们不强制要求你去阅读手册，但是如果你打算部署一个正式的Redis集群（比如生产环境用的redis集群），建议你阅读更正式的手册。 不过最好还是从这篇文档开始玩redis，等你玩了一段时间redis之后再去阅读手册

Redis 集群101

Redis 集群提供了一个运行redis实例的方式，该方式下数据会被**自动的在多个reids节点中分享**。**在分区的时候**，Redis 集群还提供了**一定程度的可用性，**即在实际应用中，当几个节点挂掉或者无法通讯的时候，系统还可以持续运行。不过当大面积的节点出问题的时候集群还是会停止（比如当主要的master挂掉了的时候）

那么在实际应用中，Redis集群可以做到什么？

* **自动切分数据集到多个节点上**的能力
* 当部分节点宕机或无法通讯的情况下**仍可继续处理命令**

Redis 集群 TCP 端口

每一个redis集群的节点需要开通两个TCP端口。一个是用于客户端的Redis TCP，如6379。另一个由客户端加10000所得，如16379，用于Redis集群总线连接。 这是一个用户 节点对节点的 二进制协议通讯通道。集群总线是用来处理节点的失效检测，配置更新，灾备授权等事情。客户端应该连接redis普通命令端口（即之前提到的6370）而绝对不要去直接连接集群总线端口。不过还是要保证这两个端口在防火墙里面打开，否则redis集群内的节点无法互相通信。

集群总线端口总是比命令端口高10000。

注意：为了让redis集群运作正常，你需要在每个节点上：

1. 把客户端用来连接redis的普通客户端通讯端口（一般是6379）对所有客户端和其他节点开放（别的节点也会用这个端口来迁移数据）
2. 所有节点之间的集群总线端口（客户端口加上10000所得）必须互相开通

如果你没有同时打开这两个端口，集群就无法正常工作了

集群总线使用了一种不同的二进制协议，供节点和节点之间交换信息用。该协议可以让节点和节点之间以更小的流量和和更短的时间来交换信息。

Redis 集群数据分片

Redis集群用的不是基于哈希值的分片方式，用的是另一种不同的分片方式。在该分片方式下所有键在概念上都是我们称之为**哈希槽**的一部分。Redis集群有16384个哈希槽。当需要在 Redis 集群中放置一个 key-value 时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点。

Redis集群中的每个节点都存放了一些哈希槽。所以举例来说，比如你有3个节点：

* 节点A 保存了从 0 到 5500 的哈希槽
* 节点B 保存了从 5501 到 11000的哈希槽
* 节点C 保存了从 11001 到 16384 的哈希槽

这么做让集群增加或者减少节点变得很简单。比如我要增加一个节点D，我只需要从节点ABC移动一部分哈希槽到D。如果我要从节点中去除节点A，我只需要把节点A上的哈希槽移动到节点B或者C。当节点A的哈希槽被全部移走了之后，我就可以将它从节点中完全去除。

因为把哈希槽从一个节点移动到另外一个节点并不需要停止集群， 所以增加、删除节点或者在各节点间调整哈希槽的占有率的时候是不用停止集群的。Redis集群支持在一条命令里面对同一个哈希槽的多个键同时操作（或者在一个事务中，或者在一个lua脚本执行过程中）。用户可以通过哈希标签强制的把多个键放到一个哈希槽里面。在Redis集群手册中可以查到哈希标签的相关说明, 不过归纳成一句话就是：当有在key里面写上段包含在{...}中的文字的时候，之后大括号{...}中的文字会被计算成哈希键。比如有两个key一个名叫 this{foo}key 另一个名叫 another{foo}key ，这两个key会被归纳到同一个哈希槽里面。这样这两个key就可以在一个命令中同时进行操作了。

Redis 集群主从模型

了在部分节点失败或者无法通信的情况下集群仍然可用，Redis 集群采用了一种 主从模型。在该模型下每一个哈希槽都会被从master端复制N份到slave节点。在我们的例子中有三个节点分别是ABC，如果节点B挂掉了，集群就无法继续工作，因为从5501到11000的哈希槽就没了。

不过如果当集群被创建的时候（迟些时候也可以）我们给每一个master节点增加一个slave从节点。模型变成这样：集群中有三个节点A，B，C，以及他们各自的从节点 A1，B1，C1，当节点B挂掉的时候，系统还可以正常运行。

节点B1是用来做为节点B的镜像的。当节点B挂掉了，集群会选举B1作为新的master节点，并继续运行下去。

不过要注意当节点B和节点B1都挂掉的时候，redis集群还是无法继续运行。

Redis 集群一致性保证

Redis 集群并不能保证数据的**强一致性**。在实际应用中这意味着在特定的情况下，就算Redis 集群告知客户端已经收到了写请求，这个写请求仍然有可能丢失。Redis集群之所以会丢失写请求的首要原因是：它采用了异步的复制机制。在写的时候会经历以下的步骤:

1. 你的客户端发送了一个写请求给master B 节点
2. master B 节点回复了一个OK给你的客户端。
3. master B 节点把这个写请求传播到它的 slave B1, B2, B3 节点上去。

正如你所见， B节点并不会等到B1，B2，B3节点都回复它之后才回复OK给客户端。因为这样会造成redis集群过高的延迟度。所以如果你的客户端正在写些什么东西，节点B又告知你的客户端它收到了写请求，但是在它把这个写请求发送给它的slave节点们之前，节点B挂了，那么其中一个slave节点（假设它还没收到写请求）被选举为master那么你这个写请求就永久的丢失了。

很多数据库被配置成每秒刷新一次数据到磁盘，他们都会发生非常类似的事情。所以你可以根据以往使用传统数据库（这些数据库都不是基于分布式的）的经验很容易的推导出这种场景。同样的你也可以强制让数据库每次都等到写入了磁盘才回复客户端，这样就可以保证一致性，但是这往往导致了系统的性能急剧下降。同样的如果你把Reids集群设计成同步复制机制也会造成性能低下。

从根本上说这是一种用一致性来换取性能的交易。

当非常有必要的时候，Redis集群也支持同步写入。它通过实现WAIT命令来实现。这样一来基本不会丢失写操作。但是请注意就算你使用了同步复制，Redis集群也不能达到强一致性，因为：总是会遇到某些更复杂的错误场景，在这些场景下slave节点在被选举为master的时候还没收到写请求。

还有一个需要注意的会丢失数据的情况。当进行一次网络网络分裂的时候某个客户端被分配到一个拥有很少节点的区域中的情况。

就拿我们的6节点例子（master是ABC，slave是A1，B1，C1），此时有一个客户端，我们称之为Z1。当网络分裂后，有可能有这种情况：现在有2方，一方是 A,C,A1，B1，C1，另一方是B和Z1。Z1依然可以写入B，而且B也会接受来自Z1的写请求。如果这次网络分裂在很短的时间内被修复， 集群依然会保持正常运行。 然而如果浙西网络分裂持续了较长时间，长到足够B1在多数方被选举为master。那么Z1发送给B的写请求都会丢失。

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。

Redis 集群配置参数

我们来做一个redis集群的部署例子。在继续后面的步骤之前我先介绍一下配置在redis.conf文件中的Redis集群参数。有些参数很容易懂，有些你必须接着读以下的内容才会懂。

* **cluster-enabled <yes/no>**: 该项如果设置成yes，该实例支持redis集群。否则该实例会像往常一样以独立模式启动。
* **cluster-config-file <filename>:** 必须注意到尽管该项是可选的，这并不是一个用户可以编辑的配置文件，这是redis集群节点自动生成的配置文件，每次一旦配置有修改它都通过该配置文件来持久化配置（基本上都是状态），这样在下次启动的时候可以重新读取这些配置。该文件中列出了该集群中的其他节点的状态，持久化变量等信息。 当节点收到一些信息的时候该文件就会被冲重写。
* **cluster-node-timeout <milliseconds>**: redis集群节点的最大超时时间。响应超过这个时间的话该节点会被认为是挂掉了。如果一个master节点超过一定的时候无法访问，它会被它的slave取代。 该参数在redis集群配置中很重要。很明显，当节点无法访问大部分master节点超过一定时间后，它会停止接受查询请求。
* **cluster-slave-validity-factor <factor>**:如果将该项设置为0，不管slave节点和master节点间失联多久都会一直尝试failover（设为正数，失联大于一定时间（factor\*节点TimeOut），不再进行FailOver）。比如，如果节点的timeout设置为5秒，该项设置为10，如果master跟slave之间失联超过50秒，slave不会去failover它的master（意思是不会去把master设置为挂起状态，并取代它）。注意：任意非0数值都有可能导致当master挂掉又没有slave去failover它，这样redis集群不可用。在这种情况下只有原来那个master重新回到集群中才能让集群恢复工作。
* **cluster-migration-barrier <count>**: 一个master可以拥有的最小slave数量。该项的作用是，当一个master没有任何slave的时候，某些有富余slave的master节点，可以自动的分一个slave给它。具体参见[手册](http://redis.io/topics/cluster-spec)中的replica migration章节
* **cluster-require-full-coverage <yes/no>**: 如果该项设置为yes（默认就是yes） 当一定比例的键空间没有被覆盖到（就是某一部分的哈希槽没了，有可能是暂时挂了）集群就停止处理任何查询炒作。如果该项设置为no，那么就算请求中只有一部分的键可以被查到，一样可以查询（但是有可能会查不全）

创建并使用 Redis 集群

注意: 手动部署一个redis集群前学习这些操作很重要。但是如果你只是想最快速的搭建一个集群，你可以跳过这节和下一节直接看 **用create-cluster脚本搭建redis集群**。创建集群之前首要的一件事情是我们需要有一些运行在**集群模式**下的空节点。集群是不能在普通redis实例上创建的。我们必须让节点运行在集群模式下才能开启一些集群的特性和使用集群的命令。

以下是集群的最小配置文件:

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. port 7000
2. cluster-enabled yes
3. cluster-config-file nodes.conf
4. cluster-node-timeout 5000
5. 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：  
命令如下:

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. mkdir cluster-test
2. cd cluster-test
3. 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。  
从**Redis Github 页面 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：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. cd 7000
2. ../redis-server ./redis.conf

你可以从实例打印的日志中看出来， 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 ：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. [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唯一（unique）的名字。每个节点通过这个名字来记忆其他节点，我们把这个字符串称之为**Node ID**

创建一个集群

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。

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. ./redis-trib.rb create --replicas 1 127.0.0.1:7000 127.0.0.1:7001 \
2. 127.0.0.1:7002 127.0.0.1:7003 127.0.0.1:7004 127.0.0.1:7005

这个命令在这里用于创建一个新的集群, 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这个集群实例的地址列表,3个master3个slave  
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,让各个节点开始互相通讯,最后可以得到如下信息：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. [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。

**用create-cluster脚本搭建redis集群**

如果你不想像上面提到的那样手动配置每个节点，这里提供了一个更简单的系统（但是你不会学习到那么多的选项）。只需要去查看下redis发布版本中的utils/create-cluster 文件夹。里面有一个叫 create-cluster 的脚本。该脚本可以通过以下命令启动一个含有6个节点（3master 和 3slave）的集群：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. create-cluster start
2. create-cluster create

在第2步，redis-trib utility 需要你接受集群方案的时候记得回答yes。  
你现在可以跟集群交互了。第一个节点会默认监听300001。如果你想停集群用以下命令：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. create-cluster stop

让我们开始玩集群吧

现阶段redis集群有一个问题，那就是缺乏客户端库。

以下是我知道的客户端实现：

* [redis-rb-cluster](https://github.com/antirez/redis-rb-cluster) 是我写的一个ruby客户端实现(这边指的是作者@antirez) 。这个库对原生的 redis-rb 进行了一个简单的封装，用最小代码量实现了高效的对集群的操作。
* [redis-py-cluster](https://github.com/Grokzen/redis-py-cluster)：这个客户端用python对redis-rb-cluster进行了转接。支持大部分 redis-py 的功能。
* [Predis](https://github.com/nrk/predis) ： 这个库最近很活跃，更新很快。基于PHP
* [Jedis](https://github.com/xetorthio/jedis)：最流行的java客户端，最近也支持redis集群了。你可以在项目的README里面的集群段落看到相关介绍。
* [StackExchange.Redis](https://github.com/StackExchange/StackExchange.Redis) ： C# 的客户端
* [thunk-redis](https://github.com/thunks/thunk-redis)：nodejs和io.js 的客户端
* Redis库的不稳定分支里面的 redis-cli 工具也提供了非常基本的集群支持。具体使用的方式是用 -c 来启动该工具可以切换到集群模式。

测试 Redis 集群最简单的方法莫过于使用上面提到的任意一种客户端或者直接使用 redis-cli 命令行工具。以下例子演示了怎样使用命令行工具进行测试:

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. $ redis-cli -c -p 7000
2. redis 127.0.0.1:7000> set foo bar
3. -> Redirected to slot [12182] located at 127.0.0.1:7002
4. OK
5. redis 127.0.0.1:7002> set hello world
6. -> Redirected to slot [866] located at 127.0.0.1:7000
7. OK
8. redis 127.0.0.1:7000> get foo
9. -> Redirected to slot [12182] located at 127.0.0.1:7002
10. "bar"
11. redis 127.0.0.1:7000> get hello
12. -> Redirected to slot [866] located at 127.0.0.1:7000
13. "world"

**注意**: 如果你用之前提到的简易脚本来创建你的集群，你的集群中的节点可能会监听不同的端口。这些端口默认是从30001开始递增。

redis-cli 的集群功能只提供了非常基本的功能，所以他总是假定：客户端知道数据在哪个节点之上，并准确的连接数据所在的节点。但是一个实际使用的客户端应该应该要缓存哈希槽和节点之间的映射关系，通过这个映射关系来引导客户单连接指定的节点。该映射关系只有集群配置改变的时候才刷新。比如在一次failover之后或者系统管理员通过增加或者删除节点来改变集群的分布之后，该映射关系才刷新。

用 redis-rb-cluster 来写一个例子app

在展示如何使用redis集群做失效备援或者重新分片之前。我们需要建立一些例子工程，或者至少理解一个简单的redis集群客户端跟集群交互的一些基本语法。

通过这种方式我们可以运行一个例子，在运行例子的同时我们尝试让一些节点挂掉或者启动一个重新分片过程，以此来观察redis集群如何响应真实情况下的突发情况的。 如果不写点什么到集群里面去没法看出什么门道。

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 的文件， 包含在redis-rb-cluster 项目里面

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. 1  require './cluster'
2. 2
3. 3  startup\_nodes = [
4. 4      {:host => "127.0.0.1", :port => 7000},
5. 5      {:host => "127.0.0.1", :port => 7001}
6. 6  ]
7. 7  rc = RedisCluster.new(startup\_nodes,32,:timeout => 0.1)
8. 8
9. 9  last = false
10. 10
11. 11  while not last
12. 12      begin
13. 13          last = rc.get("\_\_last\_\_")
14. 14          last = 0 if !last
15. 15      rescue => e
16. 16          puts "error #{e.to\_s}"
17. 17          sleep 1
18. 18      end
19. 19  end
20. 20
21. 21  ((last.to\_i+1)..1000000000).each{|x|
22. 22      begin
23. 23          rc.set("foo#{x}",x)
24. 24          puts rc.get("foo#{x}")
25. 25          rc.set("\_\_last\_\_",x)
26. 26      rescue => e
27. 27          puts "error #{e.to\_s}"
28. 28      end
29. 29      sleep 0.1
30. 30  }

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 命令向数据库设置键值对：

* SET foo0 0
* SET foo1 1
* SET foo2 2
* 以此类推

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 集群对象可以获得的最大连接数 ，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。

启动的节点列表不需要包含集群的所有节点。但这些地址中至少要有一个是有效的： 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。

从**第11行到第19行**发生了以下事情: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。

**21至30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：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. ruby ./example.rb
2. 1
3. 2
4. 3
5. 4
6. 5
7. 6
8. 7
9. 8
10. 9
11. ^C (我把程序给停了)

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。

集群重新分片

现在我们可以来尝试集群重新分片了。做分片的时候请保持集群运行，这样如果分片对程序有什么影响你就可以观察的到了。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。

重分片意思就是把一些哈希槽从一些节点移动到另一些节点中取。正如我们集群创建的时候那样做的，重新分片也可以使用redis-trib 工具来做。

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

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

你只需要指定一个节点就可以了，redis-trib 会自动找到其他的节点。目前 redis-trib 的重新分片只能通过管理功能实现，比如你不能做到从这个节点自动的移动5%的哈希槽到其他节点去（虽然这个功能正在实现中）。所以由这引出了几个问题，首要的一个就是你究竟想做一次多大范围的分片？

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. 你想移动多少哈希槽 (从 1 到 16384)?

我们尝试将1000个槽重新分片， 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， 也即是， 负责接收这 1000 个哈希槽的节点。我会使用第一个master节点： 127.0.0.1:70000 。 但是我需要在实例中指定Node ID。用redis-trib可以输出节点列表。但是我也可以通过以下命令找到节点ID：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. $ redis-cli -p 7000 cluster nodes | grep myself
2. 97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 connected 0-5460

ok，现在我知道我的目标是节点的ID是97a3a64667477371c4479320d683e4c8db5858b1。

现在需要指定从哪写节点来移动keys到目标。我输入的是all ，这样就会从其他每个master上取一些哈希槽。最后确认后你将会看到每个redis-trib移动的槽的信息，每个key的移动的信息也会打印出来。在重新分片的过程中，你的例子程序是不会受到影响的,你可以停止或者重新启动多次。

在重新分片结束后你可以通过如下命令检查集群状态

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

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

所有的节点都会被该操作覆盖到。不过此时127.0.0.1:7000这个节点会拥有更多的哈希槽，大概会有6461个。

将重新分片操作做成一个脚本

重新分片操作可以做成自动的，这样我们就不用在交互模式下手动的输入一个个参数了。可以通过以下的命令行去实现：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. ./redis-trib.rb reshard <host>:<port> --from <node-id> --to <node-id> --slots --yes

如果您经常重分片，那么可以用这个脚本来实现分片自动化。不过当前没有办法让 redis-trib 自动检测集群的键分配，由此来智能判断是否需要重新分片。该功能未来会加入。

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 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 项目包含了一个名为 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://redis.io/commands/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 命令更新一个计数器时， 应用会记录下计数器执行 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 命令， 那么计数器的值应该是 50 。
* 在每次发送 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 命令的过程中， 丢失了某条 INCR 命令， 又或者多执行了某条客户端没有确认到的 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：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. $ ruby consistency-test.rb
2. 925 R (0 err) | 925 W (0 err) |
3. 5030 R (0 err) | 5030 W (0 err) |
4. 9261 R (0 err) | 9261 W (0 err) |
5. 13517 R (0 err) | 13517 W (0 err) |
6. 17780 R (0 err) | 17780 W (0 err) |
7. 22025 R (0 err) | 22025 W (0 err) |
8. 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，那么 consistency-test.rb 将向我们报告不一致情况：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. $ redis 127.0.0.1:7000> set key\_217 0
2. OK
4. (in the other tab I see...)
6. 94774 R (0 err) | 94774 W (0 err) |
7. 98821 R (0 err) | 98821 W (0 err) |
8. 102886 R (0 err) | 102886 W (0 err) | 114 lost |
9. 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 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 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。

失效备援(failover)测试

注意：在执行本节操作的过程中， 请一直运行 consistency-test 程序。为了触发失效备援，我们要做的最简单的事情（这也是在一个分布式系统中最简单的一种故障）就是把一个redis进程搞挂，在我们的例子中就是一个master节点进程。

我们可以定义一个集群，并通过以下命令让其崩溃：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. $ redis-cli -p 7000 cluster nodes | grep master
2. 3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 1385482984082 0 connected 5960-10921
3. 2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 master - 0 1385482983582 0 connected 11423-16383
4. 97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 connected 0-5959 10922-11422

ok，现在7000，7001和7002 是master节点了。让我们用 **DEBUG SEGFAULT**命令搞挂7002 节点:

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. $ redis-cli -p 7002 debug segfault
2. Error: Server closed the connection

现在我们可以来观察下 consistency test 的输出。

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. 18849 R (0 err) | 18849 W (0 err) |
2. 23151 R (0 err) | 23151 W (0 err) |
3. 27302 R (0 err) | 27302 W (0 err) |
5. ... many error warnings here ...
7. 29659 R (578 err) | 29660 W (577 err) |
8. 33749 R (578 err) | 33750 W (577 err) |
9. 37918 R (578 err) | 37919 W (577 err) |
10. 42077 R (578 err) | 42078 W (577 err) |

你可以看到在失效备援的时候系统拒绝了578个读请求和577个写请求，但是数据库中没有引发任何一个的不一致问题。这可能跟教程刚开始部分所说的不同。在教程刚开始的时候我们说到redis 集群之所以在失效备援的时候会丢失写请求是因为它使用的是异步复制机制。我在教程开始的时候没有提到这点：其实丢失写请求的情况是很少发生的。因为把请求的响应返回给客户端和发送复制命令给slave这两件事情几乎是同时发生的。所以丢失数据的时间窗口非常小。然而非常难发生并不意味这不可能发生。所以这并没有改变redis集群无法实现强一致性的事实。

我们现在可以查看失效备援之后集群的布局（注意我同时重启了崩溃的实例，这样可以把这个节点作为slave重新加入到系统中）:

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. $ redis-cli -p 7000 cluster nodes
2. 3fc783611028b1707fd65345e763befb36454d73 127.0.0.1:7004 slave 3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 0 1385503418521 0 connected
3. a211e242fc6b22a9427fed61285e85892fa04e08 127.0.0.1:7003 slave 97a3a64667477371c4479320d683e4c8db5858b1 0 1385503419023 0 connected
4. 97a3a64667477371c4479320d683e4c8db5858b1 :0 myself,master - 0 0 0 connected 0-5959 10922-11422
5. 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 127.0.0.1:7005 master - 0 1385503419023 3 connected 11423-16383
6. 3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 1385503417005 0 connected 5960-10921
7. 2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 0 1385503418016 3 connected

现在 7000， 7001 和 7005 是master节点。7002之前是master节点，现在是7005的一个slave节点。  
[CLUSTER NODES](http://redis.io/commands/cluster-nodes) 命令的结果可能看起来很复杂，但是它实际上是很简单的，并且是一下token的组合:

* 节点 ID
* ip:port
* flags ：例如 master 、 slave 、 myself 、fail
* 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
* 集群最近一次向节点发送 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 。
* 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。

手动失效备援

有时候就算master不是真的出问题了，也需要强制引发一次失效备援。比如为了升级Redis中的一个master节点，又想尽量减小对系统可用性的影响，我们就可以用失效备援来把这个master节点转换为slave节点。

redis集群可以用[CLUSTER FAILOVER](http://redis.io/commands/cluster-failover)命令来进行手动失效备援。这个命令必须要在你想切换的目标**slave**上执行。

手动失效备援比master实际出错引发的失效备援更安全。因为手动失效备援是在系统正常运行并且新的master节点持续不断的接收来自master的复制请求的情况下将客户端的连接从原来的master移动到新的master之上的，这样可以保证在切换的过程中不丢失数据。

以下是你做失效备援的时候slave产生的日志：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. # Manual failover user request accepted.
2. # Received replication offset for paused master manual failover: 347540
3. # All master replication stream processed, manual failover can start.
4. # Start of election delayed for 0 milliseconds (rank #0, offset 347540).
5. # Starting a failover election for epoch 7545.
6. # Failover election won: I'm the new master.

基本上之前客户端连接的那个master已经被我们用失效备援停止了。与此同时，master节点发送跟slave之间的复制位移量（就是现在还差多少没有复制）。slave会停止下来等待复制位移量被消除。当达到复制位移量的时候才开始失效备援，然后旧master被告知要进行配置切换。当客户端从旧master解锁的时候他们已经重定向到新master节点了。

添加新节点

添加一个新节点其实就是以下过程：添加一个空白节点，移动一些数据到这个节点里面或者告诉它作为一个现有节点的备份节点，即一个slave节点。

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：

2种情况第一个步骤都是**添加一个空白节点**。

照启动其他节点的配置（我们已经照这个配置启动了7000到7005这6个节点了）来启动一个监听7006的节点其实很简单，唯一不一样的就是端口号.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.

现在我们像平时一样使用**redis-trib**来添加一个节点到集群中

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

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

正如你所见我使用 add-node 命令的时候第1个参数用来指定新节点的地址，第2个参数可以随便使用集群中的任何一个节点。

在实际情况下 redis-trib 其实没有帮我们做很多事情，它只是发送了一个 CLUSTER MEET 信息给节点，这件事情手动也可以完成。 然而redis-trib 还在操作之前检查了集群的状态，所以就算你知道redis-trib 是如何工作的你也最好使用redis-trib来执行这些操作。  
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

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. redis 127.0.0.1:7006> cluster nodes
2. 3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 127.0.0.1:7001 master - 0 1385543178575 0 connected 5960-10921
3. 3fc783611028b1707fd65345e763befb36454d73 127.0.0.1:7004 slave 3e3a6cb0d9a9a87168e266b0a0b24026c0aae3f0 0 1385543179583 0 connected
4. f093c80dde814da99c5cf72a7dd01590792b783b :0 myself,master - 0 0 0 connected
5. 2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 0 1385543178072 3 connected
6. a211e242fc6b22a9427fed61285e85892fa04e08 127.0.0.1:7003 slave 97a3a64667477371c4479320d683e4c8db5858b1 0 1385543178575 0 connected
7. 97a3a64667477371c4479320d683e4c8db5858b1 127.0.0.1:7000 master - 0 1385543179080 0 connected 0-5959 10922-11422
8. 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。

现在可以用redis-trib 的重新分片功能移动一些哈希槽到这个节点了。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。

添加一个slave节点

通过2步操作可以添加一个新的slave。 第一步是再次使用 redis-trib ，不过这回要带上 --slave 选项，如下：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

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

注意到这条命令跟我们之前添加master的命令非常像。我们并不用具体指定要添加slave到哪个master。这样redis-trib 会在有比较少slave的master节点中随机的找一个master来挂载slave节点。

不过你也可以精确的指定你要挂载这个slave及诶单到哪个master上：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. ./redis-trib.rb add-node --slave --master-id 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 127.0.0.1:7006 127.0.0.1:7000

这就是我们添加一个slave到指定master的方法。  
一个更手动添加slave到指定master的方式是：添加一个空节点然后通过 [CLUSTER REPLICATE](http://redis.io/commands/cluster-replicate) 命令来将其转化为一个slave节点。这个方法也同样适用于当一个节点已经是slave节点的时候你想将它转换为另一个master的slave。

比如我现在想给127.0.0.1:7005节点添加一个复制节点（就是slave），该节点现在的哈希槽范围是 11423-16383，Node ID 是 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e。我所要做的只不过是连上新节点（在此之前该节点已经被作为空master节点添加到集群里面了）并执行以下命令：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. redis 127.0.0.1:7006> cluster replicate 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e

这样就搞定了。现在我们有了一个新的复制节点，该节点复制了上面我们提到的哈希槽，并且集群中的其他节点都被通知到了（配置改变后需要几秒钟的时间来同步通知到其他节点）。我们可以用以下命令来确认一下情况是否正如我们所说：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. $ redis-cli -p 7000 cluster nodes | grep slave | grep 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e
2. f093c80dde814da99c5cf72a7dd01590792b783b 127.0.0.1:7006 slave 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 0 1385543617702 3 connected
3. 2938205e12de373867bf38f1ca29d31d0ddb3e46 127.0.0.1:7002 slave 3c3a0c74aae0b56170ccb03a76b60cfe7dc1912e 0 1385543617198 3 connected

节点 3c3a0c... 现在拥有了两个slave节点，分别是 7002 （之前就有的） 和 7006 （我们现在加上去的）。

移除一个节点

通过 redis-trib 提供的 del-node 命令可以移除一个slave节点:

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. ./redis-trib del-node 127.0.0.1:7000 `<node-id>`

可以用集群中随便一个节点作为第1个参数。第2个参数是你要移除的节点ID.  
你也可以用这条命令来移除master节点，但是**在移除master节点之前必须确保它是空的**。如果你要移除的master节点不是空的，你需要先用重新分片命令来把数据移到其他的节点。另外一个移除master节点的方法是先进行一次手动的失效备援，等它的slave被选举为新的master，并且它被作为一个新的slave被重新加到集群中来之后再移除它。很明显，如果你是想要减少集群中的master数量，这种做法没什么用。在这种情况下你还是需要用重新分片来移除数据后再移除它。

复制迁移

虽然在redis集群中通过以下命令是可以将一个slave节点重新配置为另外一个master的slave：

**[plain]** [view plain](http://blog.csdn.net/nsrainbow/article/details/49032337) [copy](http://blog.csdn.net/nsrainbow/article/details/49032337)

1. CLUSTER REPLICATE <master-node-id>

然而有时候你不想找系统管理员来帮忙，又想自动的将一个复制节点从一个master下移动到另外一个master下。 这种情况下的复制节点的自动重配置被称为复制迁移。复制迁移可以提升系统的可靠性。

注意： 你可以从 [Redis集群手册](http://redis.io/topics/cluster-spec) 中读到复制迁移的细节。但是在这篇教程里面我们只介绍大概的思路和究竟你可以从中得到什么好处。

在某种情况下，你想让集群的复制节点从一个master迁移到另一个master的原因可能是：集群的抗崩溃能力总是跟集群中master 拥有的平均slave数量成正比。  
比如，如果一个集群中每个master只有一个slave，当master和slave都挂掉的时候这个集群就崩溃了。因为此时有一些哈希槽无法找到了。虽然网络分裂会把一堆节点从集群中孤立出来（这样你一下就会知道集群出问题了），但是其他的更常见的硬件或者软件的问题并不会在多台机器上同时发生，所以很可能在你的这个集群（平均每个master只有一个slave）有一个slave在早上4点挂掉，然后他的master在随后的早上6点挂掉。这样依然会导致集群崩溃。

我们可以通过给每个master都再多加一个slave节点来改进系统的可靠性，但是这样很昂贵。复制迁移允许只给某些master增加slave。比方说你的集群有20个节点，10个master，每个master都有1个slave。然后你增加3个slave到集群中并把他们分配给某几个master节点，这样某些master就会拥有多于1个slave。

当某个master失去了slave的时候，复制迁移可以将slave节点从拥有富余slave的master旗下迁移给没有slave的master。所以当你的slave在早上4点挂掉的时候，另一个slave会被迁移过来取代它的位置，这样当master节点在早上5点挂掉的时候，依然有一个slave可以被选举为master，集群依然可以正常运行。

所以简而言之你应该了解关于复制迁移的哪些方面？

* 集群在迁移的时候会尝试去迁移拥有最多slave数量的master旗下的slave。
* 想利用复制迁移特性来增加系统的可用性，你只需要增加一些slave节点给单个master（哪个master节点并不重要）。
* 复制迁移是由配置项cluster-migration-barrier控制的： 你可以从Redis集群提供的默认配置文件 redis.conf 样例中了解到更多关于复制迁移的知识。

在Redis集群中升级节点

升级一个slave节点非常简单，因为你只需要停止节点，升级它，然后启动节点就好了。如果此时这个slave有客户端在连接也没关系，在这个slave停止的时候客户端会被重定向到别的slave去。

升级一个master就有点复杂了，推荐使用以下步骤升级master：

* 使用 CLUSTER FAILOVER 命令来使用手动失效备援，这样来把master切换为slave
* 等待master切换为slave完成
* 就像你升级普通slave一样升级它
* 如果你希望刚刚升级好的节点再次作为master在集群中运行，那就再触发一次手动失效备援让这个及节点重新成为master

照这些步骤你就可以一个一个的升级集群中的节点了。

迁移到redis集群

希望迁移到redis集群的用户可能只有一个master节点。他也可能正在使用一个现有的分片设置，该分片设置中的key已经被切分到N个节点去，这些节点是使用客户端自己实现的或者redis代理实现的一些分片算法。

在这些情况下迁移到redis集群都是很容易的，然而最终要的是如果应用使用了多key操作。以下是三种不同的情况：

1. 不使用多key操作或者事务操作或者Lua脚本(涉及到多key)。对key的访问都是独立的。
2. 使用多key操作，事务或者lua脚本(涉及到多key)，但是只作用于相同的哈希槽，即这些key都有一个{...}包裹起来的部分相同。比如以下的多key操作都是在同一个哈希标签下的：SUNION {user:1000}.foo {user:1000}.bar.
3. 使用了多key操作，事务或者lua脚本(涉及到多key)，操作的key并没有相同的哈希标签。

Redis集群无法处理第3种情况：如果不想用多key操作就要修改一下应用，或者只在相同哈希槽的情况下使用

第一种和第二种情况是适用的，所以我们重点关注前两种情况。这两种情况是采用同一个方法解决的：

假设你已经有一些数据了，这些数据本分割到N个节点上，并且如果没有数据分片的话这个N=1。你可以采用以下步骤将你的数据迁移到redis集群上：

1. 停止你的客户端。目前redis集群还没有动态迁移功能。
2. 通过BGREWRITEAOF 命令生成一个AOF(append only file)文件。并等待该AOF文件生成完毕
3. 把AOF文件命名为 aof-1 到 aof-N 。此时你可以停止你的旧实例 (在实际情况下一般会用相同的机器来跑新集群)
4. 建一个有N个master节点但没有slave节点的集群。你可以吃些添加slave。确保你所有的节点都使用AOF。
5. 停止所有节点把它们的aof文件替换成之前保存的aof文件，aof-1对应第1节点，aof-n对应第n个节点。
6. 重启你的redis集群。
7. 用 redis-trib 命令修复集群,让key可以被迁移过来
8. 最后，用 redis-trib 来检查你的集群是否迁移成功。
9. 重启客户端。

还有另外一种方法来把外部数据导入到redis集群里面，就是用 redis-trib import 命令。该命令可以把所有key从一个运行中的实例迁移到redis集群中（这些key会被从源实例中删除）。不过请注意如果源实例用的是2.8版本该操作可能会很慢，因为2.8版本还没有实现迁移连接的缓存，所以你可能需要升级源实例到3.x之后重启你的实例，之后再迁移。