# 概述

1. redis全称Remote Dictionary Server（远程字典服务器）；
2. Redis在TCP端口6379上监听到来的连接；
3. Value类型:string、hash、list、set、zset
4. Redis官网<https://redis.io/>
5. 中文版<http://www.redis.cn/>

# 数据类型

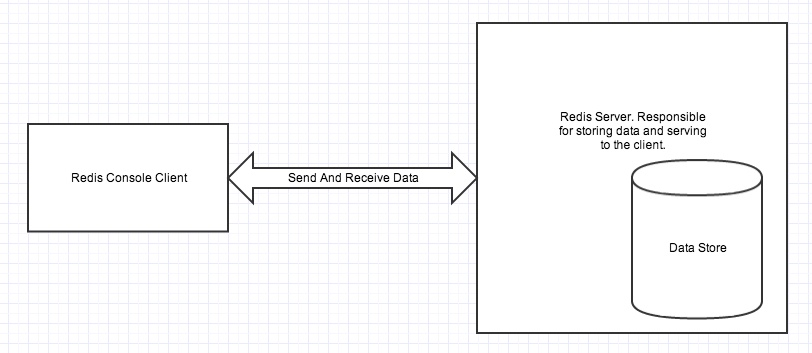
参考：<https://redis.io/topics/data-types-intro>

* String
* List：按照value的插入顺序存储；
* Set：所有元素都是独一无二的，并且是无序的；
* Zset：每一个元素都关联一个浮点类型的score，Zset通过score对所有的元素进行排序。和Set一样，Zset不允许重复的值（value），但是score运行重复；如果用户在往ZSet中添加一个已经存在的元素时，此时将修改该元素的score值，同时将重新排序；如果score值一样，ZSet将通过key的字典顺序排序；
* Hash：类似于java中的HashMap

# 架构

## 单服务结构

采用C/S架构；



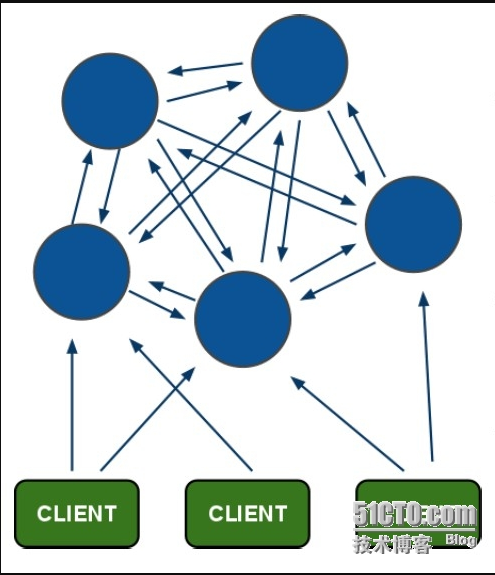
Client可以是控制台，也可以是基于Redis API的程序。

## 主从架构



主从结构的从节点可以构成层状结构。

## 集群架构



## 集群+主从

# 命令

参考中文手册<http://redisdoc.com/>

# 技术简介

## 管道

1. 参考

<https://redis.io/topics/pipelining>

1. 定义：

This way it is possible to send multiple commands to the server without waiting for the replies at all, and finally read the replies in a single step.

1. 理解：

客户端在一次请求中发送多个命令而无需等待每一个命令的执行结果，这些命令的执行结果最终由服务器一次性发送给客户端。

管道执行多个命令时，命令是在客户端一次性打包发送，而事务一次性执行多个命令时，每个命令都是逐个发送给服务器的。

1. 注意：

**IMPORTANT NOTE**: While the client sends commands using pipelining, the server will be forced to queue（排队） the replies, using memory. So if you need to send a lot of commands with pipelining, it is better to send them as batches having a reasonable number, for instance 10k commands, read the replies, and then send another 10k commands again, and so forth. The speed will be nearly the same, but the additional memory used will be at max the amount needed to queue the replies for this 10k commands.

1. 测试

参看本文档目录下的PipelineTest代码

## 事务

Redis 事务可以一次执行多个命令， 并且带有以下两个重要的保证：

* 批量操作在发送 EXEC 命令前被放入队列缓存。
* 收到 EXEC 命令后进入事务执行，事务中任意命令执行失败，其余的命令依然被执行。
* 在事务执行过程，其他客户端提交的命令请求不会插入到事务执行命令序列中。

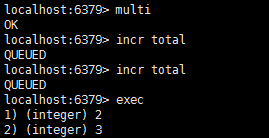
一个事务从开始到执行会经历以下三个阶段：

* 开始事务。
* 命令入队。
* 执行事务。

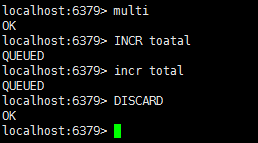
参考

<http://redisdoc.com/topic/transaction.html>

举例



如果输入multi命令后想要放弃本次事务，通过discard命令即可取消本次事务

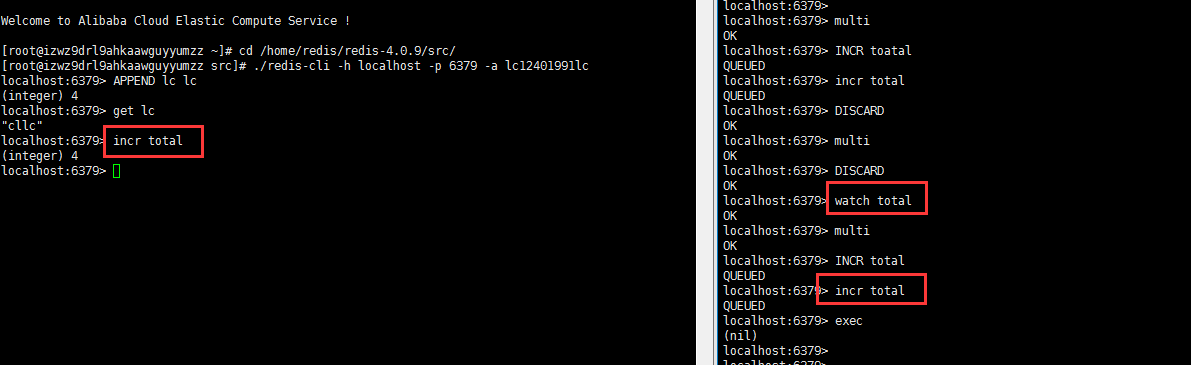


在上面的举例中，如果有两个客户端同时执行incr操作，由于redis不会对key加锁，因此可能造成两次incr的结果出现误差(两个client同时获取total的值，然后分别incr)。

为了解决这个问题，redis提供watch命令：

1. Watch 命令用于监视一个(或多个) key ，如果在事务执行之前这个(或这些) key 被其他命令所改动，那么事务将被打断。

举例



本例子中两个client同时连接同一个redis服务器，右边的客户端watch total，然后输入两次自增操作，在输入exec命令前，左边的client对total进行incr，然后右边的client执行exec出现错误。

## 发布订阅

1. 命令

Subscribe、Publish、Unsubscribe

1. 订阅对象
   * 频道

频道是一个具体的对象，比如如下命令就是订阅了foo和bar两个频道

Subscribe foo bar

如果某一个客户端向foo、bar频道中发布消息message时，订阅了这些频道的所有订阅者都会接收到该消息、

* + 模式

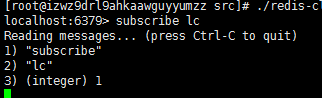
1. 消息格式
   * 此处的消息是指订阅者接收到的消息。根据redis规范定义，频道转发的每一条消息message都是一个三元组，其中第一个元素表示message的类型(也就是本次操作的命令)，第二个元素表示，第三个元素表示接收该消息的客户端目前订阅频道的总数；
   * 举例

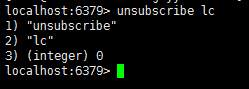
输入命令

Redis>Subscribe first second

接收到的返回消息如下：

“subscribe”





1. 额

## 哨兵Sentinel机制

简单的说，Sentinel就是Redis集群的监控工具；

### 角色

* 监控Monitoring：定时检测Master和Slave服务器是否正常工作；
* 通知Notification：Sentinel可以发送通知提醒管理员，或者通过接口调用其他程序通知系统出现问题；
* 自动迁移Automatic Failover：如果Master没有按照预期执行，Sentinel会在Slave中选择一个从服务器作为新的Master，并将其他Slave转移给新Master，同时client如果连接原有的Master，Sentinel会将新的Master地址返回给Client以便下次连接；
* 配置提供者：Configuration Provider：Client可以连接Sentinel来获取最新Master的地址，如果发生故障，哨兵将报告新地址。

### Sentinel事实

* 为了部署一个健壮系统需要至少3个Redis Sentinel实例；
* 三个Sentinel实例应该部署在独立的环境，比如不同的物理机或者不同的虚拟机中，以便因为故障导致所有的Sentinel异常；
* Sentinel+Redis的系统不能保证在发生错误期间，已经被确认过（Ack）的写操作会被保留；
* 客户端需要支持Sentinel机制；

### Sentinel搭建

1. 阅读配置文件

Sentinel.conf

1. 制定配置文件

启动一个Sentinel所需最小配置项如下：

sentinel monitor mymaster 127.0.0.1 6379 2

sentinel down-after-milliseconds mymaster 60000

sentinel failover-timeout mymaster 180000

sentinel parallel-syncs mymaster 1

参数说明：

sentinel monitor <master-group-name> <ip> <port> <quorum>

其他参数都一目了然，只有quorum，quorum表示指明一个master失效时需要的协商数量（投票数），举个例子，如果有5个Sentinel，如果quorum为2，表示只要有两个Sentinel同意Master不可达或者失效，则该Master被判断为无效；接下来这两个Sentinel中的一个将执行故障转移（Automatic Failover）

### Sentinel部署场景

参考官方文档<https://redis.io/topics/sentinel>中给出的例子

## 持久化

有关Redis数据持久化的策略参考:

<https://redis.io/topics/persistence>

### 持久化策略

* RDB（Redis Database）机制：该策略采用定期备份（写快照snapshot）机制，每隔特定时间将内存中的数据写入外存RDB文件。这种方式的缺点就是无法完全恢复最后一次备份到异常发生期间的数据。
* AOF（Append Only File）机制：该策略将所有在server端的write操作（包括写的数据）记录到一个AOF日志文件，该文件是一个append-only类型。如果发生异常可以通过log恢复。这种策略的弊端就是log日志太大，恢复耗时。
* 关闭持久化：如果用户只希望redis的数据存活时间和Redis的运行时间保持一致就可以关闭持久化。
* RDB+AOF

（这种方式RDB和AOF都是单独一体的，只是AOF的优先级要高，如果出现故障优先采用AOF恢复）通过这两种策略恢复数据，如果发生异常，Redis重启时会优先选择AOF文件进行数据恢复，因为AOF保存的数据更完整。

* RDB和AOF混合

Redis 4以后的版本支持这种方式。通过aof-use-rdb-preamble参数进行配置，有关该参数的解释详见redis.conf文件。

这种持久化能够通过 AOF 重写操作创建出一个同时包含 RDB 数据和 AOF 数据的 AOF 文件， 其中 RDB 数据位于 AOF 文件的开头， 它们储存了服务器开始执行重写操作时的数据库状态： 至于那些在重写操作执行之后执行的 Redis 命令， 则会继续以 AOF 格式追加到 AOF 文件的末尾， 也即是 RDB 数据之后。

混合机制通过命令BGREWRITEAOF触发。

### RDB持久化

#### 优点

* RDB文件紧凑并且单一，保存了某个时间点的数据集
* 创建RDB文件时，Redis主进程只需要fork一个子进程来处理RDB文件的生成即可，主进程无需做任何IO操作；
* 和AOF相比，RDB的回复较快；

#### 缺点

* RDB的持久化丢失的数据可能很大
* RDB的生成需要fork一个子进程，创建子进程比较耗时，同时如果dataset比较多，IO操作较多就比较耗时

### AOF持久化

#### 优点

* 具有多种fsync策略，

#### 缺点

* AOF文件通常比RDB文件大；

### 混合持久化

## 集群

### 集群规范

参考

<http://www.redis.cn/topics/cluster-spec.html>

<https://redis.io/topics/cluster-spec>

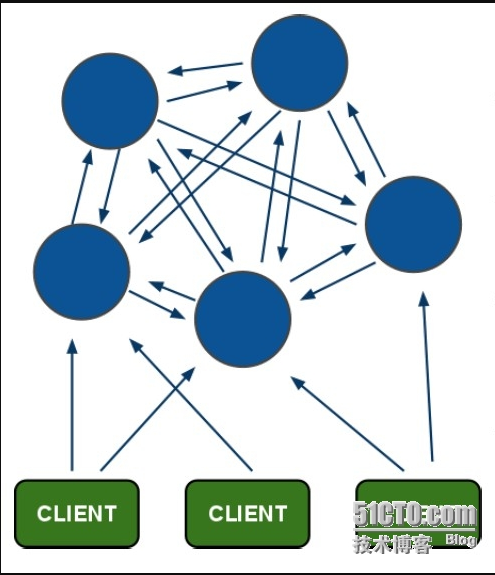
极客学院

<https://wiki.jikexueyuan.com/project/redis-guide/cluster-a.html>

### 目标

* 高性能（High Performance）
* 线性可扩展（Linear scalability）
* 写安全（write safety）
* 可用性（availability）

### 集群架构图



* 集群内的节点通过Ping-Pong机制互联；
* 集群不支持多命令的操作（因为集群中的每个节点负责一部分slot，多命令的操作可能存在key值分配到不同redis节点的情况，而多个命令之间的运行结果会相互依赖，从而导致需要数据迁移带来大量的数据传输开销）；
* 集群分区就是通过slot来实现的，每个区负责部分slot；

### 特点

* 自动分区。也就是将hash槽分配到不同的节点；
* 部分容错。如果部分节点宕机或者无法与其他节点连接通信，集群仍能提供服务；
* 数据非强一致性。如果主节点具有备份节点，可能会出现数据不一致的现象，比如一个客户端连接主节点并发送写命令（write），主节点执行命令后向client发送结果信息，此时主节点突然fail，导致无法向备份节点发送write命令，接下来备份节点之一被选举为主节点，因此刚才的write命令就从此消失。根源是redis采用异步的主从复制。

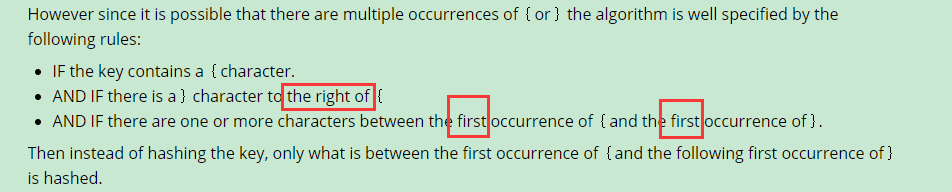
### 集群互联端口

* 服务端和客户端的连接端口是TCP协议，默认为6379端口，俗称命令端口；
* 集群节点之间互联端口，俗称集群总线端口，每一个节点的总线端口与命令端口的差值一直未10000，比如命令端口为6379，则该节点的总线端口为16739；
* 搭建集群时，需要保证每个节点的命令端口对所有授权的client是可达的，总线端口对集群中所有的接口是可达的；
* 集群内部节点通过binary protocol协议进行通信；

### 分片

Redis采用hash slot来达到分片效果。为了人为将一类key分片到同一个redis，Redis提供一种Hash Tags的分片方法，能够将复合型key人为的分片同一个节点，比如user:user1:ids和user:user1:content这两个key，如果通过集群字段分片，可能会将这两个key分到不同的节点上（比如，每个节点只有一个slot），为了人为的将这两个key分到同一个redis节点，可以将key设计为user:{user1}:ids和user:{user1}:content。Hash tags的算法如下：

* 如果key中存在{,同时存在}，并且}位于{的右边，此时需要执行Hash tags算法，否则将使用整个key进行计算hash slot
* 集群从左往右截取第一个{和第一个}之间的字符进行计算hash slot
* 如果第一个{和第一个}之间为空，整个key将用于计算hash slot



举例：

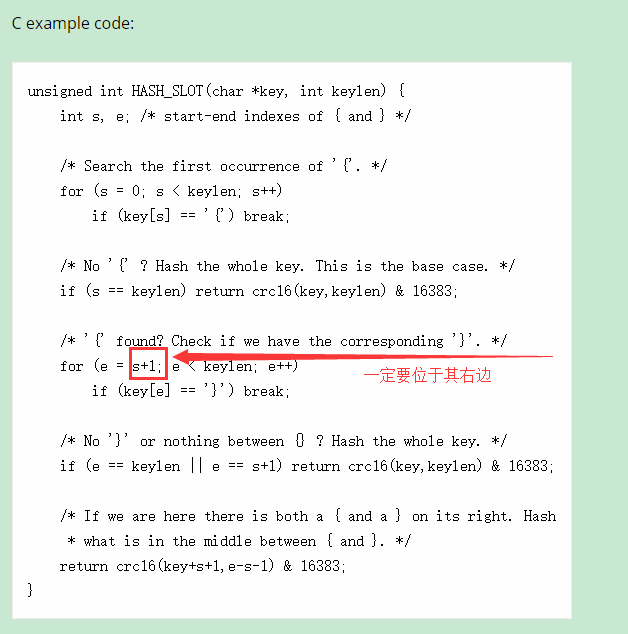
{foo}.example1中foo被用于计算hash slot

Foo.{}.test中整个key被用于计算hash slot

{{foo}}中{foo被用于计算hash slot.(注意这个例子中第二个{要被作为hash slot的计算值)

{foo}.{bar}中foo被用于计算hash slot

算法代码是最好的解释



### 集群节点属性

* 节点名字是一个160位的随机数
* 节点名字构造好之后将不再更改，并将该id记录到配置文件；

## 集群搭建

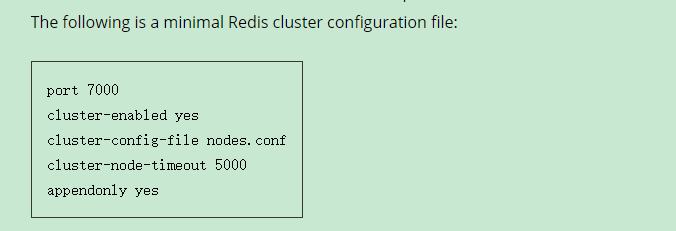
### 搭建步骤

参考<http://blog.51cto.com/zhoushouby/1560400>

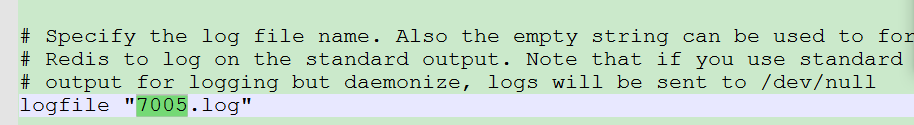
<https://redis.io/topics/cluster-tutorial>

1. 创建6个redis实例

创建6个路径，分别复制一份redis存放在对应的路径下，并修改其配置文件，包括如下几项。（6个实例的端口号分别为7000~7005）

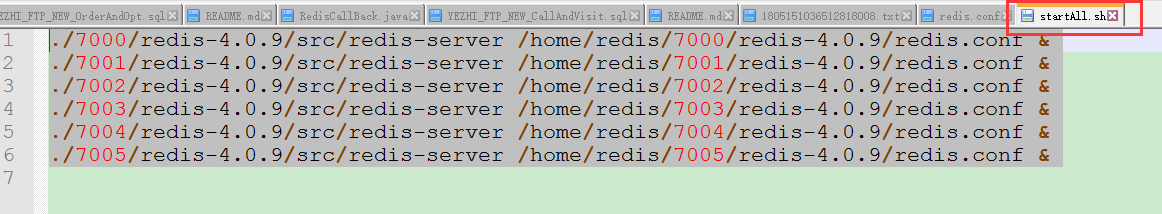


为了便于将日志记录，修改了日志输出文件

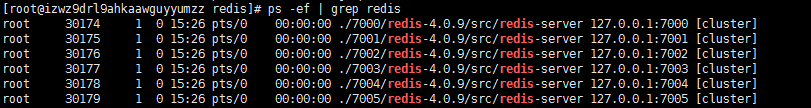


1. 逐个启动所有的redis

批量脚本



执行结果



1. 接下来将多个实例做出主从结构

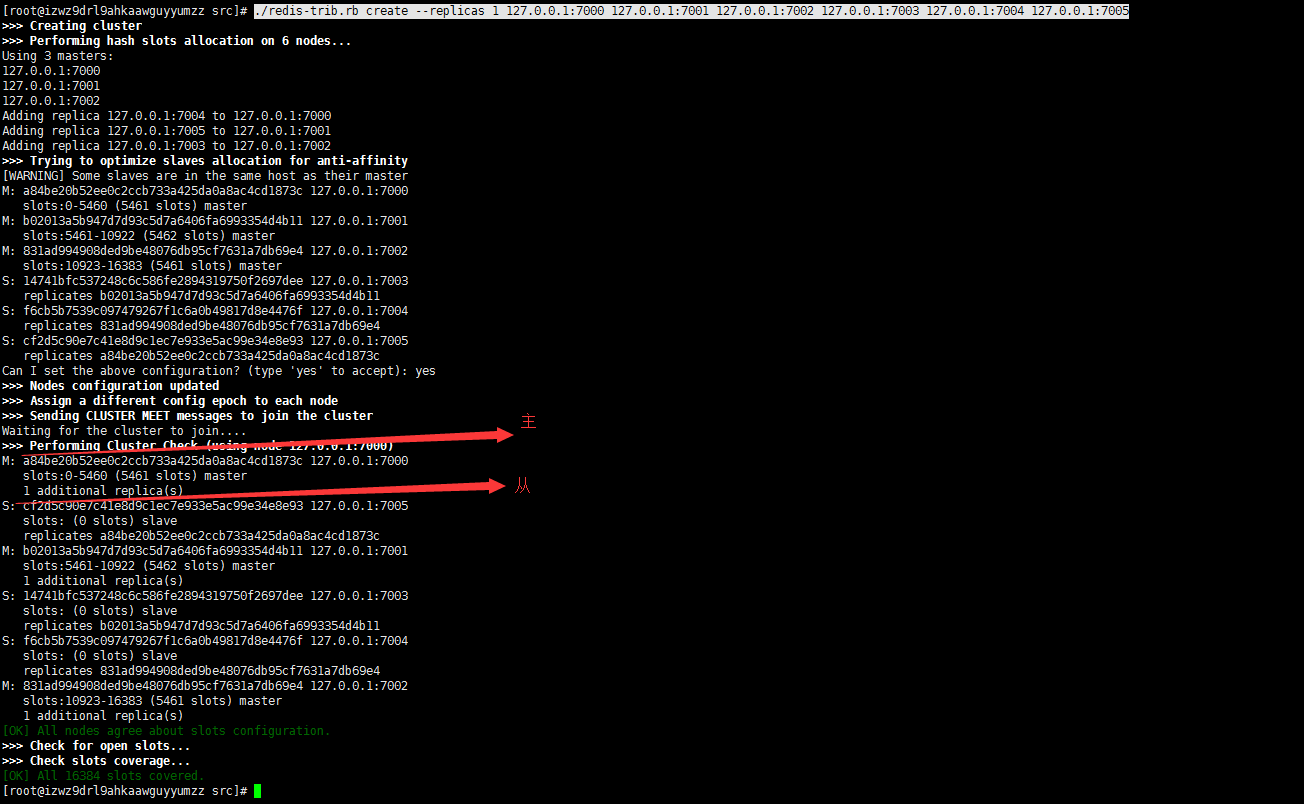
命令（只需要在其中一台服务器操作下面内容）：

* 安装ruby：（已安装就忽略此步骤）

yum -y install ruby ruby-rdoc

* 构建集群

./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



构建完成后，每主redis负责一部分哈希槽slot，通过上图可以得出三个主redis各自负责的slot编号如下：

|  |  |
| --- | --- |
| Redis端口 | Slot编号 |
| 7000 | 0~5460 |
| 7001 | 5461~10922 |
| 7002 | 10923~16383 |

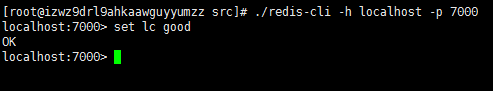
### 测试

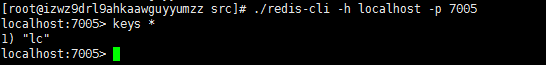
接下来做一些测试

* 显示集群节点信息

./redis-trib.rb check localhost:7000

* 在主redis存放数据，从redis查询数据





* 在一个主redis上写入一个不属于该redis的slot

因为lclc的crc16计算得到的slot（1625）编号不属于7001

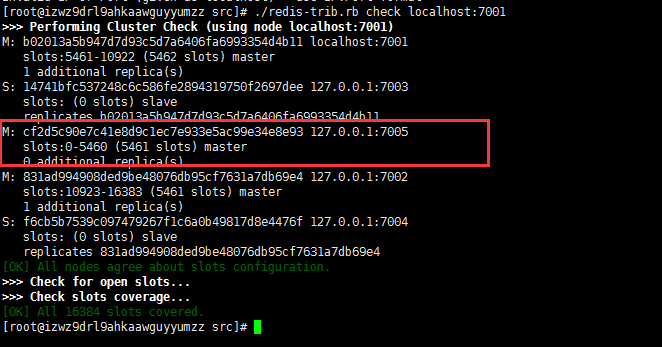


* 主redis宕机

将7000的redis进程kill，然后通过命令查看集群状态（指定集群任意一个ip:port即可）；

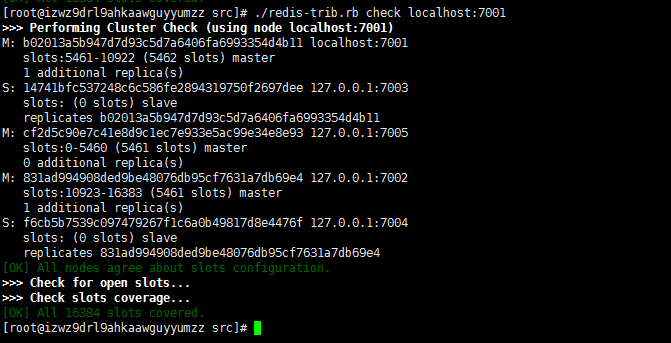
从redis 7005被选举为主redis，命令如下：

./redis-trib.rb check localhost:7001



* 恢复已经宕机的redis

重启redis 7000，通过图片可知7000并没有被加入到集群

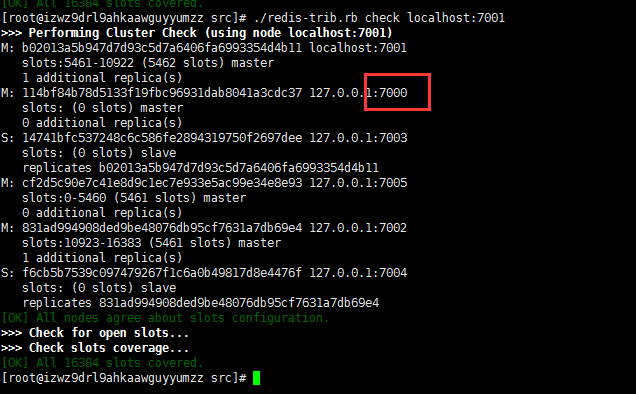


可以通过如下命令把新的节点加入到集群（加入主redis或者从redis）

主redis：./redis-trib.rb add-node 127.0.0.1:7000 127.0.0.1:7001

第二个参数是集群中任意一个redis实例

添加的主redis此时并没有slot

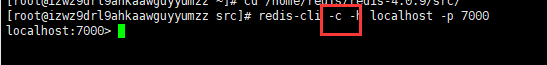


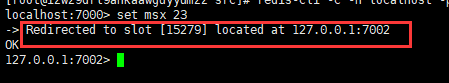
从redis

./redis-trib.rb add-node --slave 127.0.0.1:7000 127.0.0.1:7001

命令中第二个参数也是集群中某个redis实例，并不是指定主redis，redis系统会随机选择一个主redis作为新加入的redis实例的主redis

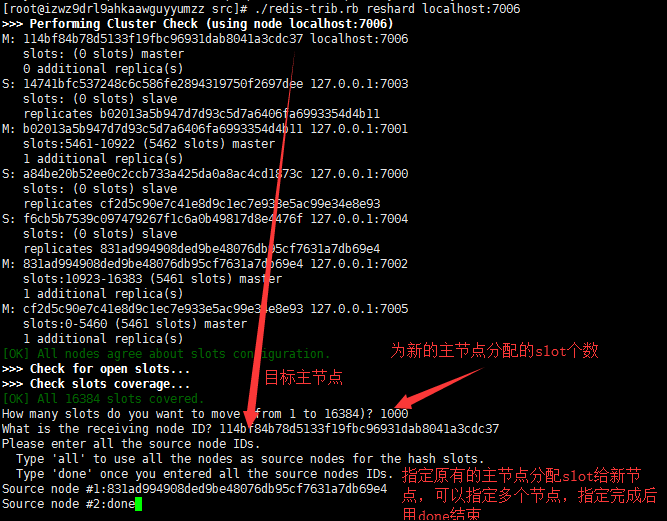
* 通过参数连接集群

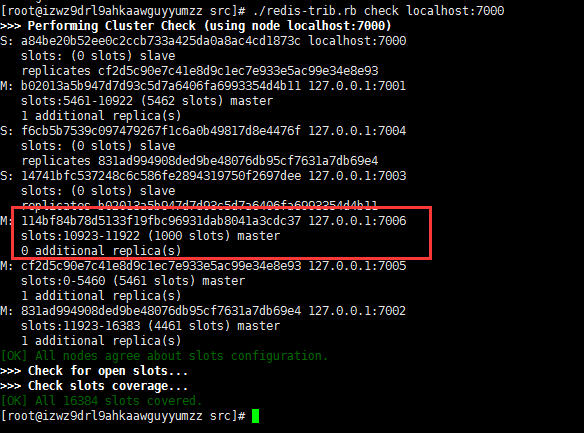




* 集群增加主节点并分配slot

./redis-trib.rb reshard localhost:7006





* 删除主节点

如果主节点不是empty，必须将该主节点服务的slot全部转移到其他主节点之后才能删除。

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

## 主从复制

### 复制机制

* 主节点Master和从节点slave正常连接时，主节点将发送一串命令流（a stream of command）给从节点，以便从节点保存主节点上的数据更新（这些命令只涉及具有写动作的命令，比如write，设置key过期时间等等命令）；
* 当Master和Slave之间的连接被破坏时（网络原因或者主从节点的一方感知到连接超时），Slave将尝试重新连接Master，连接上Master之后，将进行部分再同步（partial resynchronization），部分再同步也就是将重新执行连接失效期间丢失的命令流；
* 当部分再同步无法正确恢复，从节点将进行全量再同步（full resynchronization），这个过程更加复杂，首先主节点将创建一个所有数据的快照（snapshot）并将快照发给从节点（此时主从节点数据是一致的），然后将从创建快照时刻之后执行的命令流发送给从节点执行；

默认情况Redis采用异步复制进行主从数据的同步，这种同步模式保证了Redis的High Latency和High Performance。

### 全量复制步骤

* 主节点创建一个后台进程用于生成RDB文件（当前时刻的快照）；
* 同时将接下来的所有写命令缓存在内存中；
* RDB文件生成完成后，发送给需要全量复制的从节点；
* 从节点将RDB文件保存在本地磁盘，然后加载到内存中；
* 接下来主节点将缓存下来的所有写命令发送给从节点（以命令流的方式）；

注：Redis 2.8.18版本之后的支持直接发送RDB文件给从节点的方式，如此一来无需将RDB文件保存在本地磁盘。

### 主从关系描述

* Redis采用异步机制进行复制，同时从节点也采用异步方式确认同步的数据量；
* 主节点和从节点之间是一对多的关系，也就是一个主节点可以有多个从节点，但是一个从节点只能对应一个主节点；
* 主从节点之间可以构成多层结构。也就是一个从节点也可以接收其他从节点的连接，从而构成多层结构。从Redis 4.0之后，所有从节点都接收主节点的复制流。
* 在主节点处，复制是非阻塞的（Non-Blocking），也就是说当一个或者多个从节点在进行初始化复制或者部分再同步操作时，主节点仍然可以接收命令操作；
* 在从节点处，复制也是非阻塞的。
* 可以使用复制来避免 master 将全部数据集写入磁盘造成的开销：一种典型的技术是配置你的 master Redis.conf 以避免对磁盘进行持久化，然后连接一个 slave ，其配置为不定期保存或是启用 AOF。但是，这个设置必须小心处理，因为重新启动的 master 程序将从一个空数据集开始：如果一个 slave 试图与它同步，那么这个 slave 也会被清空。

### 数据安全性

采用主从复制时，强烈建议在主节点和从节点端都开启数据持久化，并且避免Redis实例自动重启（比如Redis自身检测到读写磁盘缓慢，导致延时增大而重启）。

自动重启的危害：

* 假设A实例是Master，A实例开启了自动重启，B，C是其Slave；
* A实例crash，A将重启，同时A没有开启数据持久化，此时A实例的数据为空；
* B，C接收A实例的复制，由于A此时为空，B，C将摧毁并清空原有的副本；

### 实践

# 内存布局

# 传输协议

参考<https://redis.io/topics/protocol>

# 启动模式

bind参数

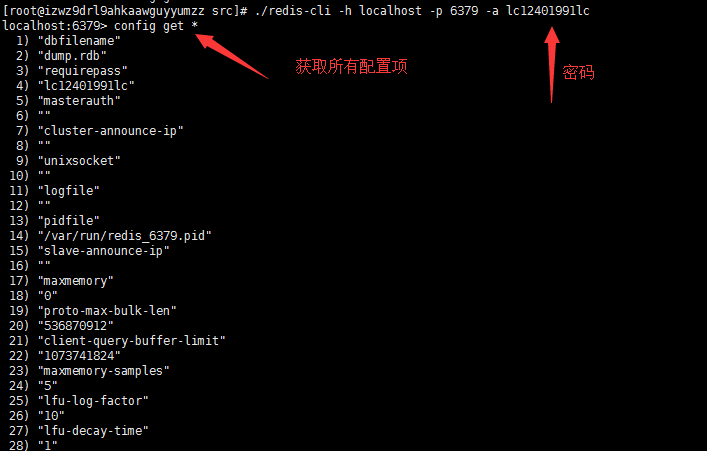
# By default, if no "bind" configuration directive is specified, Redis listens # for connections from all the network interfaces available on the server. # It is possible to listen to just one or multiple selected interfaces using # the "bind" configuration directive, followed by one or more IP addresses.

    这句话的意思是说，如果没有设置bind参数，则该redis服务器会监听该服务器所在机器上的所有网络接口，如果该机器有多个网卡，可以通过bind参数来指定需要监听的网络接口的ip。

protected-mode参数

如果protected-mode 为yes，则只允许局域网内的客户端连接该服务器

如果protected-mode 为No，则允许公网内的客户端连接该服务器，如果采用这种方式，需要设置requirepass来设置连接密码以保证安全性



# 参考资料