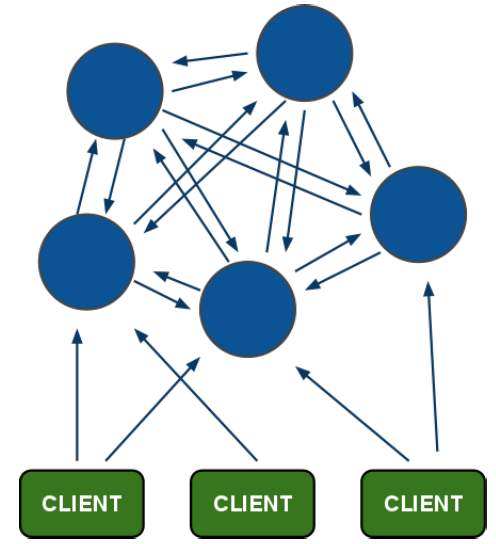
# **Redis Cluster 源码分析**

## 简介

Redis Cluster是由多个同时服务于一个数据集合的Redis实例组成的整体，对于用户来说，用户只关注这个数据集合，而整个数据集合的某个数据子集存储在哪个节点对于用户来说是透明的。Redis Cluster具有分布式系统的特点，也具有分布式系统如何实现高可用性与数据一致性的难点，由多个Redis实例组成的Redis Cluster结构通常如下：



Redis Cluster特点如下：

* 所有的节点相互连接；
* 集群消息通信通过集群总线通信，，集群总线端口大小为客户端服务端口+10000，这个10000是固定值；
* 节点与节点之间通过二进制协议进行通信；
* 客户端和集群节点之间通信和通常一样，通过文本协议进行；
* 集群节点不会代理查询；

## 槽（slot）概念

Redis Cluster中有一个16384长度的槽的概念，他们的编号为0、1、2、3……16382、16383。这个槽是一个虚拟的槽，并不是真正存在的。正常工作的时候，Redis Cluster中的每个Master节点都会负责一部分的槽，当有某个key被映射到某个Master负责的槽，那么这个Master负责为这个key提供服务，至于哪个Master节点负责哪个槽，这是可以由用户指定的，也可以在初始化的时候自动生成（redis-trib.rb脚本）。这里值得一提的是，在Redis Cluster中，只有Master才拥有槽的所有权，如果是某个Master的slave，这个slave只负责槽的使用，但是没有所有权。

## 数据分片

在Redis Cluster中，拥有16384个slot，这个数是固定的，存储在Redis Cluster中的所有的键都会被映射到这些slot中。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 。集群中的每个节点负责处理一部分哈希槽。

## 节点的槽指派信息

clusterNode结构的slots属性和numslot属性记录了节点负责处理那些槽：

struct clusterNode {

//…

unsigned char slots[16384/8];

};

Slots属性是一个二进制位数组(bit array)，这个数组的长度为16384/8=2048个字节，共包含16384个二进制位。

Master节点用bit来标识对于某个槽自己是否拥有。比如对于编号为1的槽，Master只要判断序列的第二位（索引从0开始）是不是为1即可。时间复杂度为O（1）。



## 集群所有槽的指派信息

通过将所有槽的指派信息保存在clusterState.slots数组里面，程序要检查槽i是否已经被指派，又或者取得负责处理槽i的节点，只需要访问clusterState.slots[i]的值即可，复杂度仅为O（1）。



## CLUSTER MEET



需要组建一个真正的可工作的集群，我们必须将各个独立的节点连接起来，构成一个包含多个节点的集群。

连接各个节点的工作使用CLUSTER MEET命令来完成。

CLUSTER MEET <ip> <port>

**CLUSTER MEET命令实现：**

1. 节点 A 会为节点 B 创建一个 clusterNode 结构， 并将该结构添加到自己的 clusterState.nodes 字典里面。
2. 节点A根据CLUSTER MEET命令给定的IP地址和端口号，向节点B发送一条MEET消息。
3. 节点B接收到节点A发送的MEET消息，节点B会为节点A创建一个clusterNode结构，并将该结构添加到自己的clusterState.nodes字典里面。
4. 节点B向节点A返回一条PONG消息。
5. 节点A将受到节点B返回的PONG消息，通过这条PONG消息节点A可以知道节点B已经成功的接收了自己发送的MEET消息。
6. 之后，节点A将向节点B返回一条PING消息。
7. 节点B将接收到的节点A返回的PING消息，通过这条PING消息节点B可以知道节点A已经成功的接收到了自己返回的PONG消息，握手完成。
8. 之后，节点A会将节点B的信息通过Gossip协议传播给集群中的其他节点，让其他节点也与节点B进行握手，最终，经过一段时间后，节点B会被集群中的所有节点认识。

## 数据结构

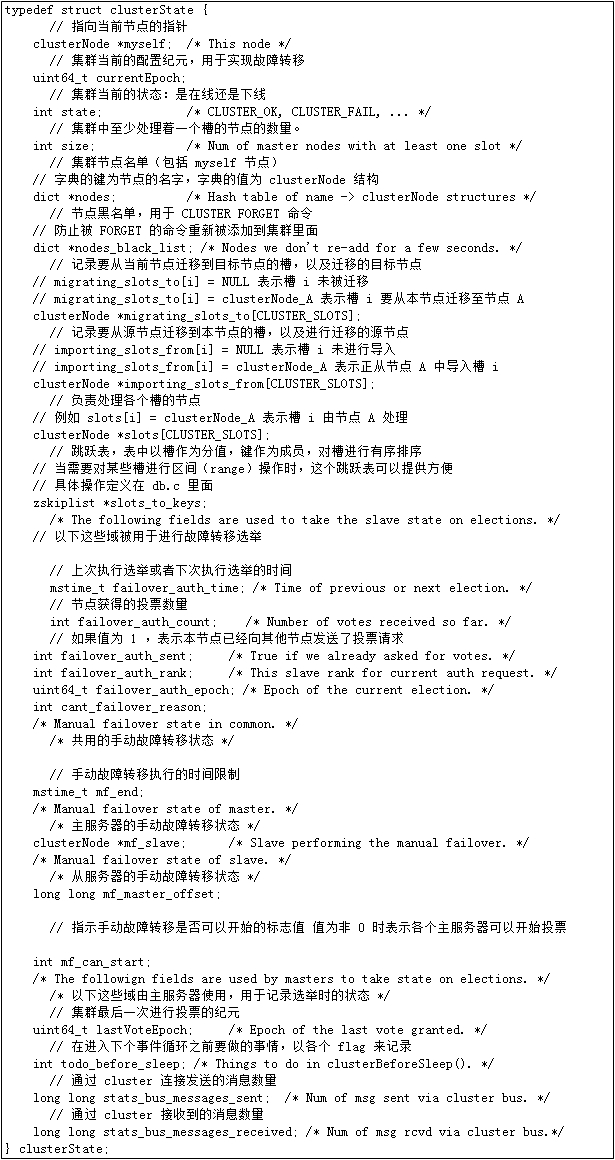
clusterNode 结构保存了一个节点的当前状态， 比如节点的创建时间， 节点的名字， 节点当前的配置纪元， 节点的 IP 和地址， 等等。

1. slots：位图，由当前clusterNode负责的slot为1
2. salve, slaveof：主从关系信息
3. ping\_sent, pong\_received：心跳包收发时间
4. clusterLink \*link：Node间的连接
5. list \*fail\_reports：收到的节点不可达投票



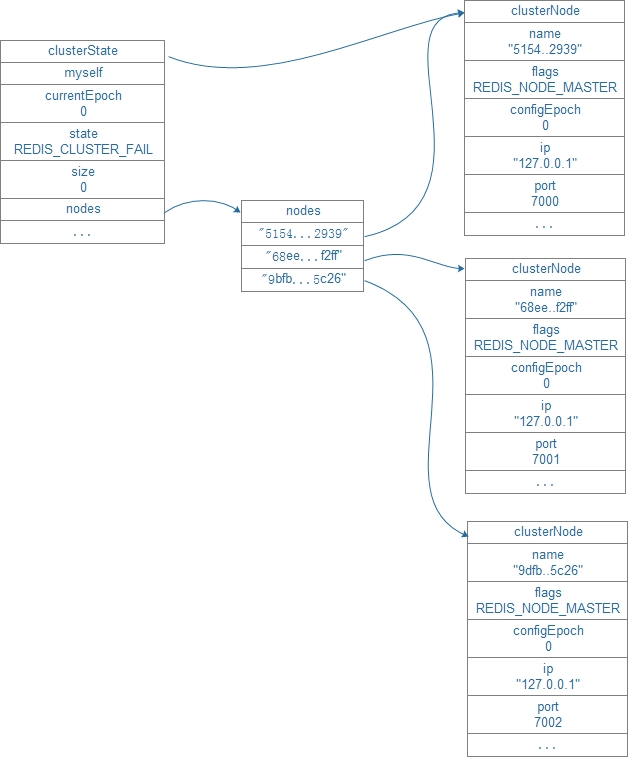
clusterState 结构记录了在当前节点的视角下， 集群目前所处的状态。

1. myself：指针指向自己的clusterNode
2. currentEpoch：当前节点见过的最大epoch，可能在心跳包的处理中更新
3. nodes：当前节点感知到的所有节点，为clusterNode指针数组
4. slots：slot与clusterNode指针映射关系
5. migrating\_slots\_to, importing\_slots\_from：记录slots的迁移信息
6. failover\_auth\_time, failover\_auth\_count, failover\_auth\_sent, failover\_auth\_rank, failover\_auth\_epoch：Failover相关



clusterLink 结构保存了连接节点所需的有关信息， 比如套接字描述符， 输入缓冲区和输出缓冲区。

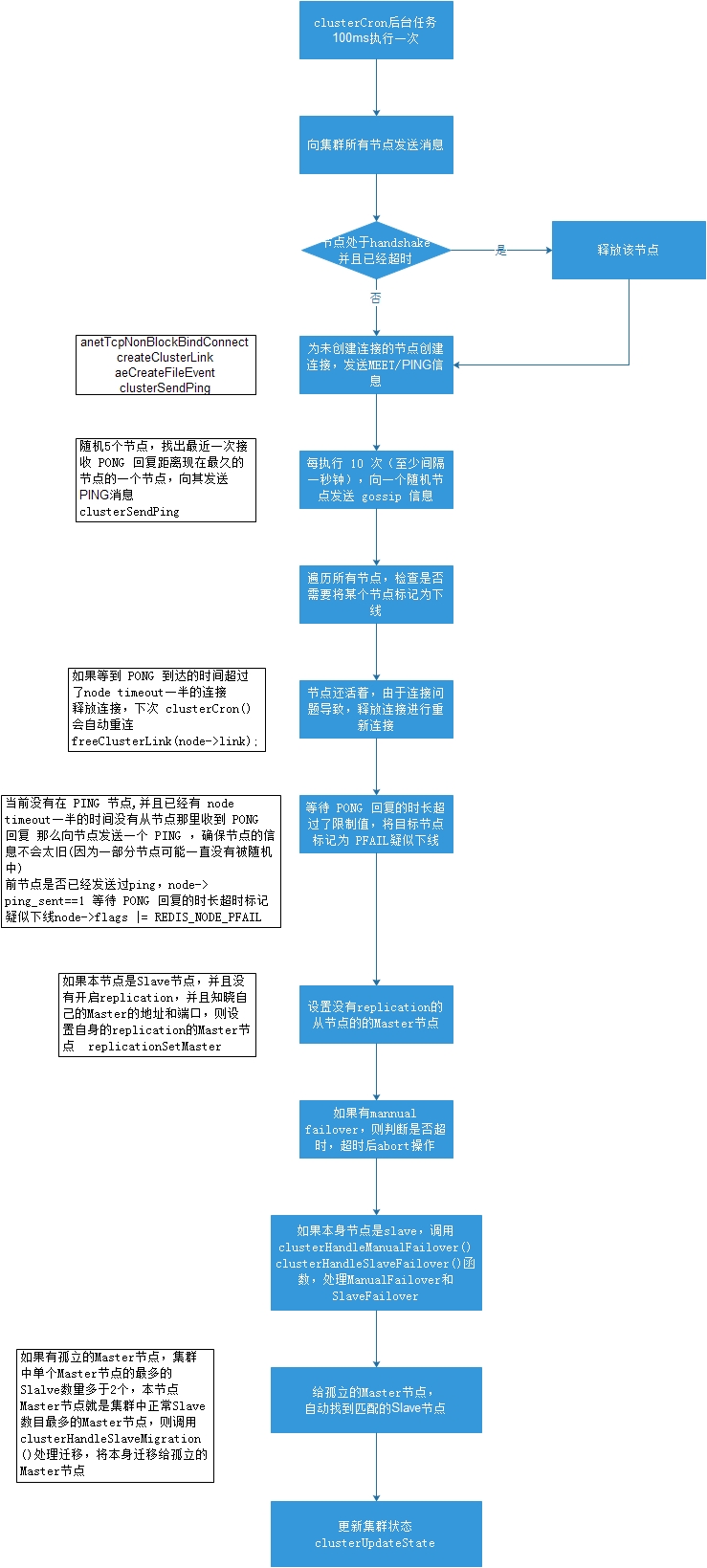




## 定时任务clusterCron

**定时任务 clusterCron**

1. 对handshake节点建立Link，发送Ping或Meet
2. 选择合适的clusterNode发送Ping
3. 如果是从查看是否需要做Failover
4. 统计并决定是否进行slave的迁移，来平衡不同master的slave数
5. 判断所有pfail报告数是否过半数



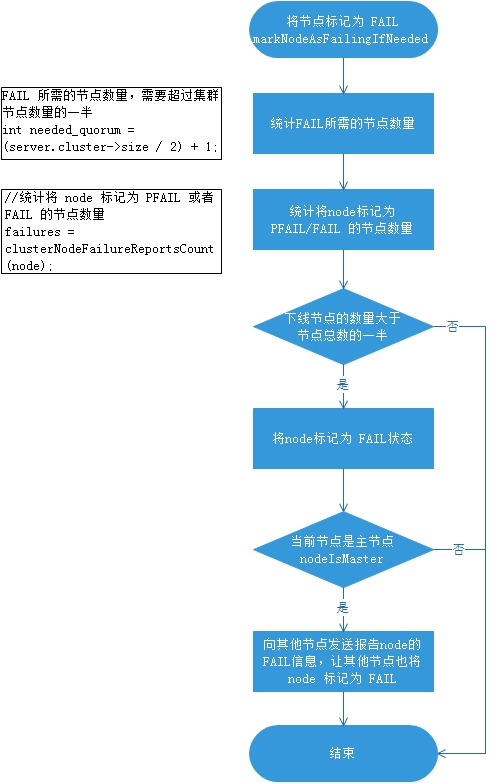
## 故障转移

当slave发现自己的master变为FAIL状态时，便尝试进行Failover，以期成为新的master。由于挂掉的master可能会有多个slave。Failover的过程需要经过类Raft协议的过程在整个集群内达到一致， 其过程如下：

* slave发现自己的master变为FAIL
* 将自己记录的集群currentEpoch加1，并广播Failover Request信息
* 其他节点收到该信息，只有master响应，判断请求者的合法性，并发送FAILOVER\_AUTH\_ACK，对每一个epoch只发送一次ack
* 尝试failover的slave收集FAILOVER\_AUTH\_ACK
* 超过半数后变成新Master
* 广播Pong通知其他集群节点

将 node 标记为 FAIL 需要满足以下两个条件：

1. 有半数以上的主节点将 node 标记为 PFAIL 状态。
2. 当前节点也将 node 标记为 PFAIL 状态。



## 请求重定向

由于每个节点只负责部分slot，以及slot可能从一个节点迁移到另一节点，造成客户端有可能会向错误的节点发起请求。因此需要有一种机制来对其进行发现和修正，这就是请求重定向。有两种不同的重定向场景：

1. **MOVED错误**

* 请求的key对应的槽不在该节点上，节点将查看自身内部所保存的哈希槽到节点 ID 的映射记录， 节点回复一个 MOVED 错误。
* 需要客户端进行再次重试。



1. **ASK错误**
   * 请求的key对应的槽目前的状态属于MIGRATING状态，并且当前节点找不到这个key了，节点回复ASK错误。ASK会把对应槽的IMPORTING节点返回给你，告诉你去IMPORTING的节点尝试找找。
   * 客户端进行重试 首先发送ASKING命令，节点将为客户端设置一个一次性的标志（flag），使得客户端可以执行一次针对 IMPORTING 状态的槽的命令请求，然后再发送真正的命令请求。
   * 不必更新客户端所记录的槽至节点的映射。

## 心跳数据

* Header，发送者自己的信息
  + 所负责slots的信息
  + 主从信息
  + ip port信息
  + 状态信息
* Gossip，发送者所了解的部分其他节点的信息
  + ping\_sent, pong\_received
  + ip, port信息
  + 状态信息，比如发送者认为该节点已经不可达，会在状态信息中标记其为PFAIL或FAIL



clusterMsg结构的currentEpoch、sender、myslots等属性记录了发送者自身的节点信息，接收者会根据这些信息，在自己的clusterState.nodes字典里找到发送者对应的clusterNode结构，并对结构进行更新。



Redis集群中的各个节点通过Gossip协议来交换各自关于不同节点的状态信息，其中Gossip协议由MEET、PING、PONG三种消息实现，这三种消息的正文都由两个clusterMsgDataGossip结构组成。

每次发送MEET、PING、PONG消息时，发送者都从自己的已知节点列表中随机选出两个节点(可以是主节点或者从节点),并将这两个被选中节点的信息分别保存到两个结构中。

当接收者收到消息时，接收者会访问消息正文中的两个结构，并根据自己是否认识clusterMsgDataGossip结构中记录的被选中节点进行操作：

1. 如果被选中节点不存在于接收者的已知节点列表，那么说明接收者是第一次接触到被选中节点，接收者将根据结构中记录的IP地址和端口号等信息，与被选择节点进行握手。
2. 如果被选中节点已经存在于接收者的已知节点列表，那么说明接收者之前已经与被选中节点进行过接触，接收者将根据clusterMsgDataGossip结构记录的信息，对被选中节点对应的clusterNode结构进行更新。



## 槽迁移

当槽x从一个Node A向另一个Node B迁移时，Node A和Node B都会有这个槽x，Node A上槽x的状态设置为MIGRATING，Node B上槽x的状态被设置为IMPORTING。

MIGRATING状态

key存在则成功处理

key不存在，则返回客户端ASK，仅当这次请求会转向另一个节点，并不会刷新客户端中node的映射关系，也就是说下次该客户端请求该key的时候，还会选择Node A节点

key包含多个命令，如果都存在则成功处理，如果都不存在，则返回客户端ASK，如果一部分存在，则返回客户端TRYAGAIN，通知客户端稍后重试，这样当所有的key都迁移完毕的时候客户端重试请求的时候回得到ASK，然后经过一次重定向就可以获取这批键

IMPORTING状态

正常命令会被MOVED重定向，如果是ASKING命令则命令会被执行，从而key没有在老的节点已经被迁移到新的节点的情况可以被顺利处理

Key不存在则新建

没有ASKING的请求和正常请求一样被MOVED，这保证客户端node映射关

系出错的情况下不会发生写错

1. **假如k1属于槽x，并且k1还在Node A**



1. **假如k1属于槽x，并且k1不在Node A**



## 集群消息处理 clusterProcessPacket

* 根据收到的消息更新自己的epoch和slave的offset信息
* 处理MEET消息，使加入集群
* 从goosip中发现未知节点，发起handshake
* 对PING，MEET回复PONG
* 根据收到的心跳信息更新自己clusterState中的master-slave，slots信息
* 对FAILOVER\_AUTH\_REQUEST消息，检查并投票
* 处理FAIL，FAILOVER\_AUTH\_ACK，UPDATE信息

