details about redis cluster specification

本文主要关注于 Redis Cluster 的实现细节的分析，本文档不是 Redis Cluster 的使用教程而是实现分析，所以你可能需要 Redis Cluster Specification 的基础才能更好地阅读本文档，所有源码分析基于 Redis-3.0.0

1. Introduction

A. Redis Cluster 的主要特点：
   a. Redis Cluster 是去中心的化的分布式存储，通过哈希槽的方式来分割数据到各个节点，每个 cluster 有 16384 个哈希槽,每个 key 通过 CRC16 校验后对 16384 取模来决定放置哪个槽.集群的每个节点负责一部分 hash 槽。
   b. Redis Cluster 的每个 Redis 的 Master 节点可以设置多个 Slave 节点，在 Master 节点挂掉后会从多个 Slave 节点选举一个新的 Master 节点出来。
   c. Redis Cluster 中各个节点在初始化后都是两两通过 Cluster Port 连到一起的，Cluster Port 数值上比 Server Port 大 10000。
   d. Redis Cluster 中节点感知采用 Gossip 协议，新节点加入后传播给整个 Cluster 中的所有节点。
   e. Redis Cluster 中 Master 节点的选举采用类 Raft 的协议，不过选举的参与方不是多个 Slave 节点而是多个其他的 Master 节点，这样做是为了节省资源，如果专门为了实现 Leader Election 而设置 2n+1 个 Slave 节点是极大浪费。
   f. Redis Cluster 不支持强一致性，原因就是 Master 节点和 Slave 节点之间的数据同步是弱同步的；还有就是，发生网络分区后原 Master 节点在此期间被写入的数据在网络恢复之后可能会发生丢失，因为另外一个分区发生了重新选举的，原 Master 节点在网络恢复后变成了 Slave 节点。
B. Redis Cluster 的实现概要：
   a. Redis Cluster 有两种状态

      REDIS_CLUSTER_OK，所有节点表现正常，在关闭 cluster-require-full-coverage 配置项的情况下允许不超过半数的主节点失效

      REDIS_CLUSTER_FAIL，在打开 cluster-require-full-coverage 配置项情况下存在失效的主节点或者超过半数的主节点失效
   b. Redis Cluster 添加了一个命令 CLUSTER 用于管理和维护 Redis Cluster，其中包括多个子命令，例如：

      `CLUSTER INFO` 打印集群的信息

      `CLUSTER NODES` 列出集群当前已知的所有节点（node），以及这些节点的相关信息。

      `CLUSTER FORGET <node_id>` 从集群中移除 `node_id` 指定的节点。

      。。。。。。

      Redis Cluster 相关的所有命令的处理在 clusterCommand 中(cluster.c:3800)

   c. Redis Cluster 实现的关键数据结构：

struct clusterNode;

保存的是单个节点的状态，包括节点的名字、节点的标识、ConfigEpoch、处理槽位数组、本节点处理的槽数量、主从关系以及连接信息等。

```
struct clusterNode {

    // 创建节点的时间
    mstime_t ctime; /* Node object creation time. */

    // 节点的名字，由 40 个十六进制字符组成
    // 例如 68eef66df23420a5862208ef5b1a7005b806f2ff
    char name[REDIS_CLUSTER_NAMELEN]; /* Node name, hex string, sha1-size */

    // 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_... */

    // 节点当前的配置纪元，用于实现故障转移
    uint64_t configEpoch; /* Last configEpoch observed for this 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 */

    // 该节点负责处理的槽数量
    int numslots;     /* Number of slots handled by this node */

    // 如果本节点是主节点，那么用这个属性记录从节点的数量
    int numslaves;    /* Number of slave nodes, if this is a master */

    // 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 */

    // 最后一次发送 PING 命令的时间
    mstime_t ping_sent;         /* Unix time we sent latest ping */

    // 最后一次接收 PONG 回复的时间戳
```

```
    mstime_t pong_received;    /* Unix time we received the pong */

    // 最后一次被设置为 FAIL 状态的时间
    mstime_t fail_time;          /* Unix time when FAIL flag was set */

    // 最后一次给某个从节点投票的时间
    mstime_t voted_time;         /* Last time we voted for a slave of this master */

    // 最后一次从这个节点接收到复制偏移量的时间
    mstime_t repl_offset_time;    /* Unix time we received offset for this node */

    // 这个节点的复制偏移量
    long long repl_offset;          /* Last known repl offset for this node. */

    // 节点的 IP 地址
    char ip[REDIS_IP_STR_LEN];    /* Latest known IP address of this node */

    // 节点的端口号
    int port;                       /* Latest known port of this node */

    // 保存连接节点所需的有关信息
    clusterLink *link;              /* TCP/IP link with this node */

    // 一个链表，记录了所有其他节点对该节点的下线报告
    list *fail_reports;             /* List of nodes signaling this as failing */

};
```

struct clusterLink;

clusterLink 结构保存了与其他节点进行通信所需的全部信息，包括连接创建时间、套接字、输入和输出缓冲区等信息。

```
typedef struct clusterLink {

    // 连接的创建时间
    mstime_t ctime;                 /* Link creation time */

    // TCP 套接字描述符
    int fd;                         /* TCP socket file descriptor */

    // 输出缓冲区，保存着等待发送给其他节点的消息（message）。
    sds sndbuf;                     /* Packet send buffer */

    // 输入缓冲区，保存着从其他节点接收到的消息。
    sds rcvbuf;                     /* Packet reception buffer */
```

```
    // 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;
```

struct clusterState;

每个节点都保存了一个这样的状态，该状态用于保存对于该节点本身视角下整个集群的状态。所以同一个集群中的不同节点在特定时刻，整个集群的状态可能是不一致的。

```
typedef struct clusterState {

    // 指向当前节点的指针
    clusterNode *myself;    /* This node */

    // 集群当前的配置纪元，用于实现故障转移
    uint64_t currentEpoch;

    // 集群当前的状态：是在线还是下线
    int state;                  /* REDIS_CLUSTER_OK, REDIS_CLUSTER_FAIL, ... */

    // 集群中至少处理着一个槽的节点的数量。
    int size;                   /* Num of master nodes with at least one slot */

    // 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 */

    // 节点黑名单，用于 CLUSTER FORGET 命令
    // 防止被 FORGET 的命令重新被添加到集群里面
    // （不过现在似乎没有在使用的样子，已废弃？还是尚未实现？）
    dict *nodes_black_list; /* Nodes we don't re-add for a few seconds. */

    // 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;

/* The following fields are used to take the slave state on elections. */
// 以下这些域被用于进行故障转移选举

// 上次执行选举或者下次执行选举的时间
mstime_t failover_auth_time; /* Time of previous or next election. */

// 节点获得的投票数量
int failover_auth_count;    /* Number of votes received so far. */

// 如果值为 1 ，表示本节点已经向其他节点发送了投票请求
int failover_auth_sent;     /* True if we already asked for votes. */

int failover_auth_rank;     /* This slave rank for current auth request. */

uint64_t failover_auth_epoch; /* Epoch of the current election. */
// 不能进行 failover 的原因
int cant_failover_reason;

/* Manual failover state in common. */
/* 共用的手动故障转移状态 */

// 手动故障转移执行的时间限制
mstime_t mf_end;            /* Manual failover time limit (ms unixtime).
                               It is zero if there is no MF in progress. */
/* Manual failover state of master. */
/* 主服务器的手动故障转移状态 */
clusterNode *mf_slave;      /* Slave performing the manual failover. */
/* Manual failover state of slave. */
/* 从服务器的手动故障转移状态 */
long long mf_master_offset; /* Master offset the slave needs to start MF
                               or zero if stil not received. */
// 指示手动故障转移是否可以开始的标志值
// 值为非 0 时表示各个主服务器可以开始投票
int mf_can_start;           /* If non-zero signal that the manual failover
                               can start requesting masters vote. */
```

```
    /* The followign fields are uesd by masters to take state on elections. */
    /*  以下这些域由主服务器使用，用于记录选举时的状态  */

    // 集群最后一次进行投票的纪元
    uint64_t lastVoteEpoch;         /* Epoch of the last vote granted. */

    // 在进入下个事件循环之前要做的事情，以各个 flag 来记录
    int todo_before_sleep; /* Things to do in clusterBeforeSleep(). */

    // 通过 cluster 连接发送的消息数量
    long long stats_bus_messages_sent;    /* Num of msg sent via cluster bus. */

    // 通过 cluster 接收到的消息数量
    long long stats_bus_messages_received; /* Num of msg rcvd via cluster bus.*/

} clusterState;
```

struct clusterMsg;
clusterMsg 结构是集群中消息的消息头，真正的数据部分由 clusterMsgData 结构保存。clusterMsgData 是一个 union 联合体，消息类型可以分为 PING、PONG 和 MEET 消息（由 clusterMsgGossip 结构定义），FAIL 消息，PUBLISH 消息，UPDATE 消息

有 9 中消息类型，但实际上只有 4 种消息体，union 里面有 4 种类型的 struct

```
union clusterMsgData {

    /* PING, MEET and PONG */
    struct {
        /* Array of N clusterMsgDataGossip structures */
        // 每条消息都包含两个 clusterMsgDataGossip 结构
        clusterMsgDataGossip gossip[1];
    } ping;

    /* FAIL */
    struct {
        clusterMsgDataFail about;
    } fail;

    /* PUBLISH */
    struct {
        clusterMsgDataPublish msg;
    } publish;

    /* UPDATE */
```

```
    struct {
        clusterMsgDataUpdate nodecfg;
    } update;

};

typedef struct {
……
union clusterMsgData data;
} clusterMsg;
```

PING 消息或者 MEET 消息：如果当前节点是第一次遇见这个节点，并且消息类型是 MEET 消息，则将该节点加入集群的节点列表中，并在发送 PING 后返回的 PONG。对于 PING 消息，节点会调用 clusterSendingPing 发送一个 PONG 消息。

PING 消息、PONG 消息或 MEET 消息：对于 PING 消息更新发送者信息，对于 PONG 消息则更新本节点对发送者节点的认识，收到 PONG 后可以移除对该发送者的 PFAIL 状态并判断是否可以移除发送者的 FAIL 状态。

FAIL 消息：如果收到 FAIL 消息，则从消息中获取到下线节点的信息，如果下线节点不是当前节点也没有处于 FAIL 状态，则将对该下线节点标记为 FAIL 状态，并移除 PFAIL 状态。

PUBLISH 消息：向订阅者发送消息。

FAILOVER_AUTH_REQUEST 消息：这是一条请求获得 Failover 授权的消息，发送者请求当前节点为支持它进行故障转移投票。在满足条件的情况下，为请求进行故障转移的节点进行投票，支持它的故障迁移。

FAILOVER_AUTH_ACK 消息：这是一条 Failover 授权回复消息，用于本节点决定自己是否可以进行故障转移操作。

MFSTART 消息：该类型消息是本节点（Master 节点）的 Slave 节点发送，用于 Slave 节点请求进行 Manual Failover 操作，该操作会暂停客户端，让服务器在指定的时间内不再接受被暂停客户端发来的命令。

UPDATE 消息：因为其哈希槽配置已经变化，消息发送者要求消息接收者进行更新

d. Redis Cluster 启动流程：

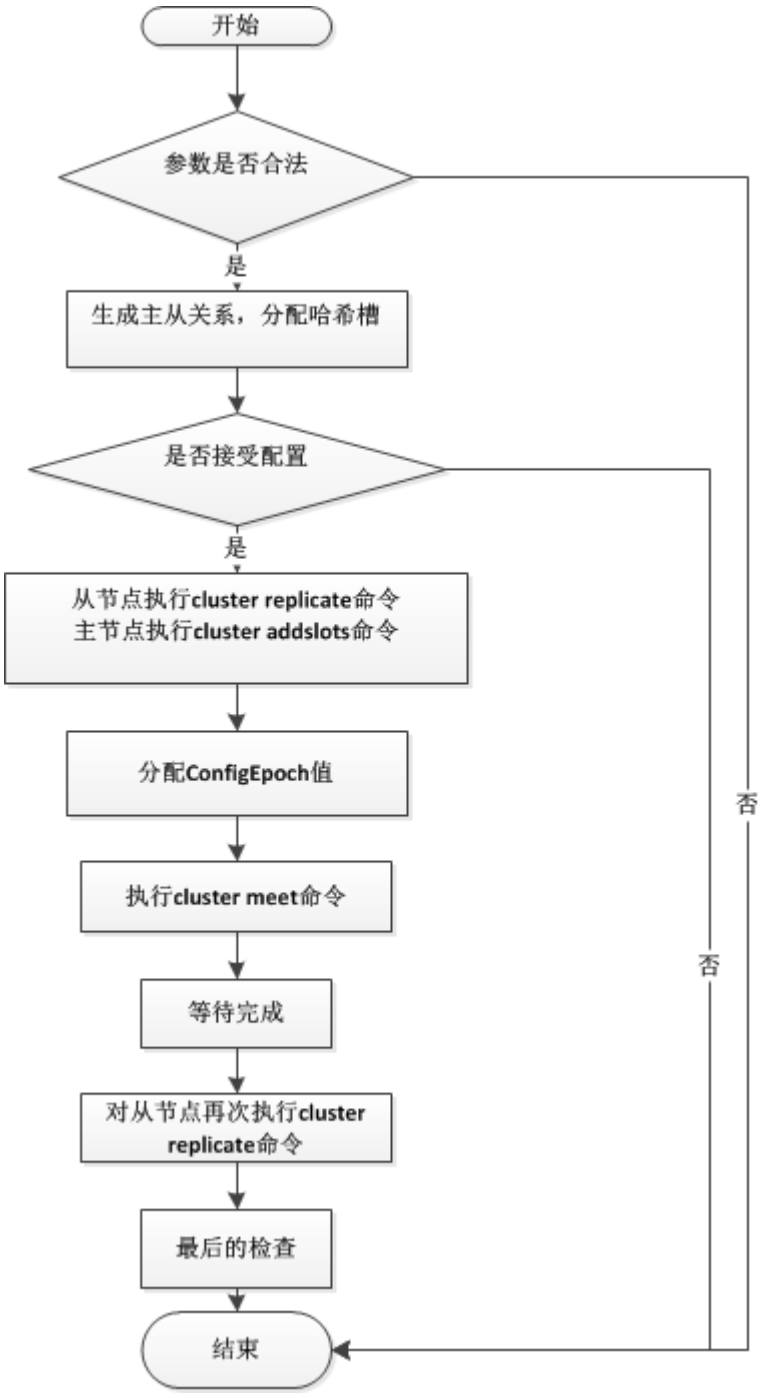
1) .initServer(redis.c:1837)会设置定时器任务 serverCron
2) .serverCron(redis.c:1243)会调用 Redis Cluster 定时任务 clusterCron，100ms 执行一次，里面工作主要包括和 Cluster 中的节点建立链接(createClusterLink)；随机 5 个节点发送 gossip 消息；检查所有节点，设置下线标记；更新 Cluster 状态
3) .initServer(redis.c:1876)会调用 clusterInit 进行初始化
4) .clusterInit(cluster.c:391)主要进行相关的参数初始化
5) .clusterInit 里面主要进行 Cluster Node 配置文件的锁定，读取，保存，然后监听 Cluster Port 对应的地址，设置 callback clusterAcceptHandler 和 callback clusterReadHandler 在这个 callback 里面进行节点消息的处理，然后申请一个用于保存哈希槽的 sorted set.
6) .clusterInit 最后进行 resetManualFailover 重置手动容错的相关参数

2. Scalability

在探讨 Redis Cluster 的伸缩性之前，先看看利用 redis-trib.rb 脚本创建 Redis Cluster 集群发生了什么，例如：

./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

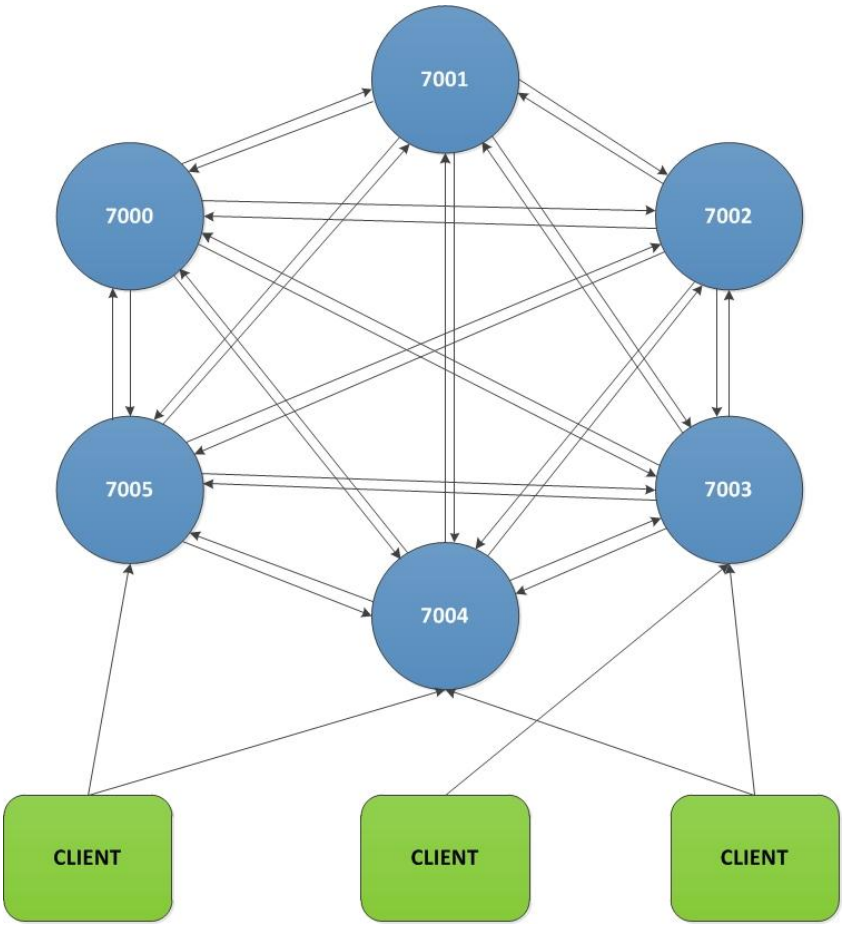
会发生如下流程：



1. 调用 create_cluster_cmd(redis-trib.rb)处理 create 命令,会发生如下调用
2. 检查参数合法性 check_create_parameters

3. 生成主从关系，并给主节点分配哈希槽，直到这步还没有真正发命令给 redis 节点进行配置，alloc_slots，show_nodes
4. yes_or_die，交互命令，是否接受以上的配置，yes 或者退出
5. flush_nodes_config 刷新每个节点的配置，对从节点则设置 cluster replicate 命令(会失败，因为还不知道主节点的存在)，对主节点则设置 cluster addslots 命令，添加负责的哈希槽
6. assign_config_epoch 给每个节点分配递增的 ConfigEpoch 值，每次递增 1，通过 cluster set-config-epoch 命令设置 ConfigEpoch 值，因为前面对哈希槽进行更新了所以要更新 ConfigEpoch 来保证哈希槽配置是全局一致的
7. join_cluster 通过 cluster meet 命令来互相认识对方，这里通过介绍第一个节点给其他 5 个节点来达到相互认识的目的
8. wait_cluster_join 因为相互认识对方有一定迟延，故这里需要等待相互认识工作完成
9. 再次调用 flush_nodes_config，因为这时候从节点知道主节点的存在了，所以 cluster replicate 会成功
10. check_cluster 检查集群配置是否正常，检查包括 check_config_consistency 是否每个主节点的哈希槽配置是一样的，check_open_slots 是否有 importing 或 migrating slots，check_slots_converage 是否左右哈希槽都被覆盖了

这时候 Redis Cluster 便初始化成功了



Redis Cluster 初始化完成后，无论主从节点都是两两建立了 TCP 链接的

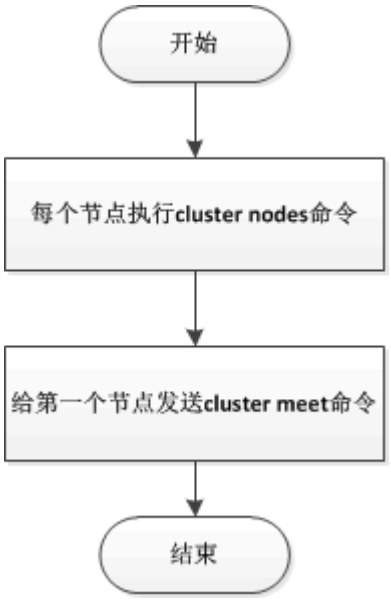
关于 Redis Cluster 伸缩性，主要就是探讨增减节点的实现，增加一个节点，通过命令：

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

其中第一个参数是新节点地址，第二参数可以是集群中任意一个节点地址 redis-trib.rb add-node 命令实际发送了 cluster meet 命令给集群里某一个节点，该节点会将新节点加入它的集群配置，并与新节点进行握手，以确保可以连上新节点，最后通过定时发 ping 包给集群其他节点以告之有新节点的存在，其他节点也因此将新节点加入它的集群配置，并与新节点进行握手，以确保可以连上新节点

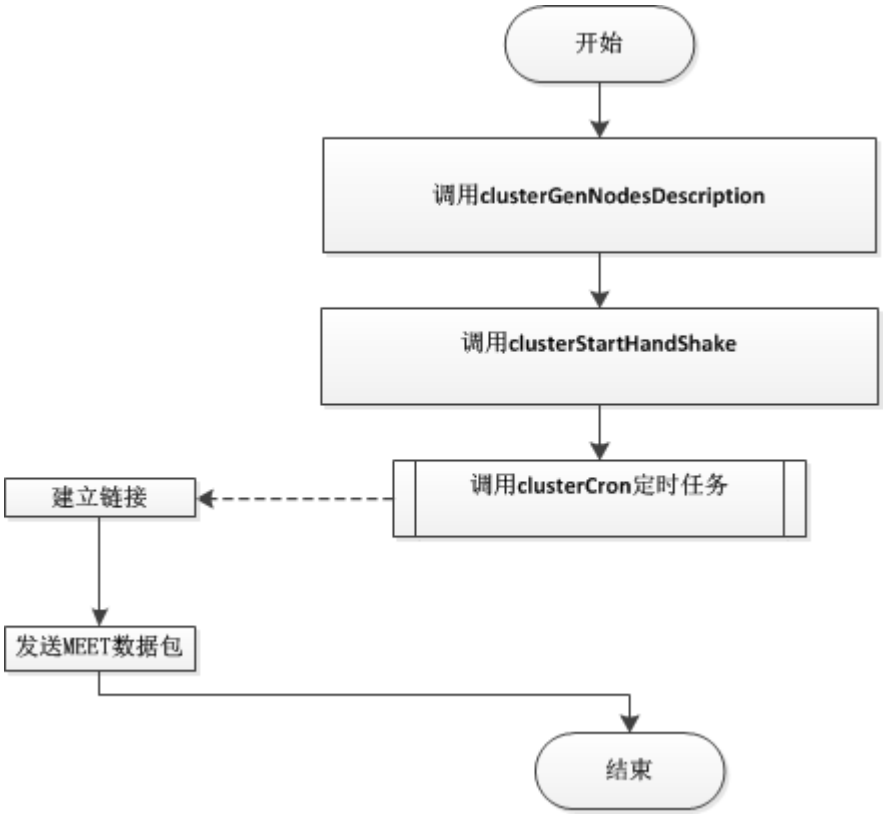
这个命令发生了如下过程：

客户端：



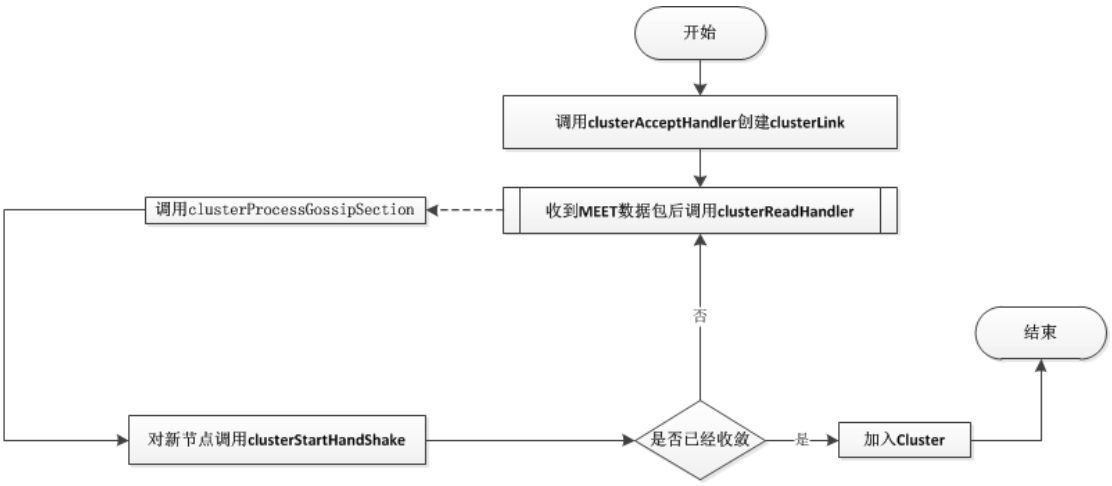
1. 调用 addnode_cluster_cmd 来进行添加节点的操作
2. 在 addnode_cluster_cmd 里面首先会调用 load_cluster_info_from_node 利用第二个参数连上 Redis Cluster，对每个节点发送 cluster nodes 命令，加载现有的节点配置信息列表
3. 然后给客户端拿到的第一个节点发送 cluster meet 命令，把新的节点加入到 Cluster 中。

当前节点服务端：



1. 在客户端的第 2 步中，第二个参数指定的节点会首先收到 cluster nodes 命令，进入到 clusterCommand 的”nodes”分支里面调用 clusterGenNodesDescription 返回 Cluster 的节点配置信息，这里面的内容是和 nodes.conf(这个文件的名字取决于 redis.conf 里面的配置)是一样的
2. 在客户端的第 3 步中，给 Cluster 的节点发送 cluster meet 命令，进入到 clusterCommand 的”meet”分支，在检查参数合法了之后，调用 clusterStartHandshake
3. 在 clusterStartHandshake 里面会调用 createClusterNode 来创建新的节点，并且设置 flag 值为 REDIS_NODE_HANDSHAKE 和 REDIS_NODE_MEET，意思分别是该节点还没和当前节点完成第一次 PING-PONG 通讯；当前节点还未与该节点进行过接触，带有这个标识会让当前节点发送 MEET 数据包而不是 PING 数据包
4. 然后给新节点设置 IP 和 PORT，然后调用 clusterAddNode 把新节点添加到当前节点的节点 dict 里面，通过全局的 redisServer 对象 server(redis.c:69)的 clusterState 属性 cluster 可以访问到，也就是 server.cluster->nodes，这个就是当前节点保存的 Redis Cluster 所有节点的数据，包括自身
5. 注意！在第 2 步中的 clusterStartHandShake 中并没有建立和新节点建立 TCP 连接，在完成第 3 和 4 步之后即退出了这个函数，这是因为新节点已经添加到当前节点 dict 中并且设置了 flag，随后的事情就交给定时任务 clusterCron 来完成了。
6. clusterCron 会每秒执行 10 次，在遍历 server.cluster->nodes 的时候(cluster.c:3046)发现 TCP 连接还没创建的话就会调用 anetTcpNonBlockBindConnect(cluster.c:3063)来创建 TCP 连接，调用 createClusterLink(cluster.c:3078)来创建 Redis Cluster 的内部链接的封装 struct clusterLink，注册 clusterReadHandler,这个函数就是用来处理 Redis Cluster 之间交互的命令的。
7. clusterSendPing 发送 MEET 数据包。

新节点服务端：



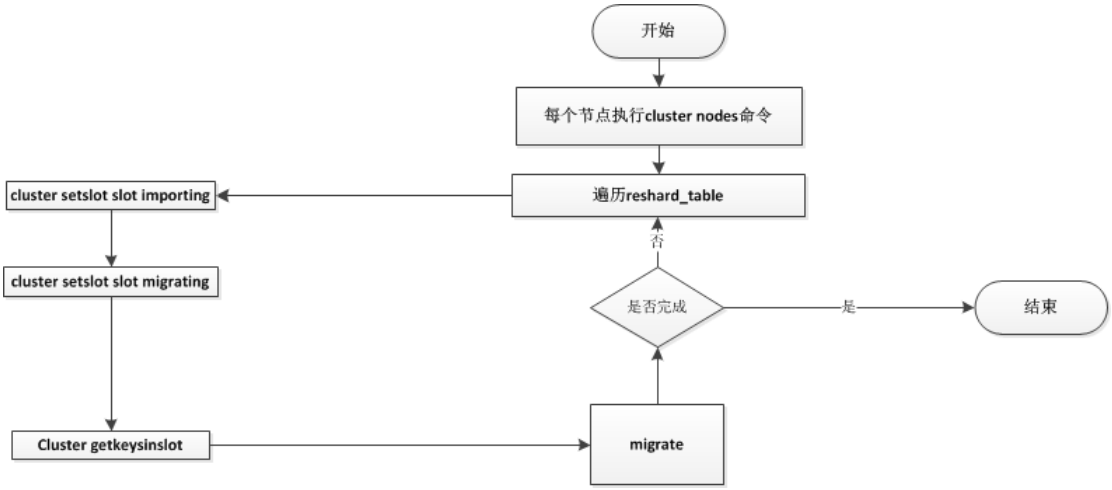
1. 新节点收到 TCP connect 请求后会回调 clusterAcceptHandler(cluster.c:566)进行处理，同样地会注册 clusterReadHandler 来进行 Redis Cluster 之间交互的命令。
2. 新节点收到“当前节点服务端“在第 7 步发送的 MEET 数据包，然后开始进入 clusterReadHandler 处理，真正的处理函数的是内部的 clusterProcessPacket(cluster.c:2050)
3. 在 clusterProcessPacket 内会设置 myself->ip(cluster.c:1636)，因为 myself 一开始是不知道自己的“官方“IP 的所以要通过 MEET 数据包设置
4. 然后创建 clusterNode 到 dict 中 clusterAddNode(cluster.c:1650),因为前面的“当前节点”对于新节点也是新的，在 dict 里面是查不到的，所以需要新创建一个加进去
5. 调用 clusterProcessGossipSection(cluster.c:1661)，里面对 MEET 数据包的数据体进行遍历，先得到 count 的数值，这个 clusterMsg 里面的 count 在发送 MEET，PING，PONG 这 3 种 Gossip 协议消息的时候使用，意思是指正文包含的节点数据数量，也就是 ClusterMsgDataGossip 的数量，每次遍历对于新的节点调用 clusterStartHandshake 进行握手，和前面的“当前节点”的流程是一样的；对于旧的节点进行失效判断，检查 PFail 或者 Fail 的状态
6. 就这样前面的“当前节点“发送 MEET 数据包给新节点，然后新节点也对发送过来的节点列表发送 MEET 数据包，在多轮交互后，新节点就和原先 Redis Cluster 的每个节点建立了连接，进而完成了加入 Cluster 的过程

在增加一个节点之后，往往需要 reshard，把哈希槽均衡到各个节点从而达到负载均衡和伸缩性的目标，那么 reshard 可以通过以下命令来进行：

./redis-trib.rb reshard 127.0.0.1:7000

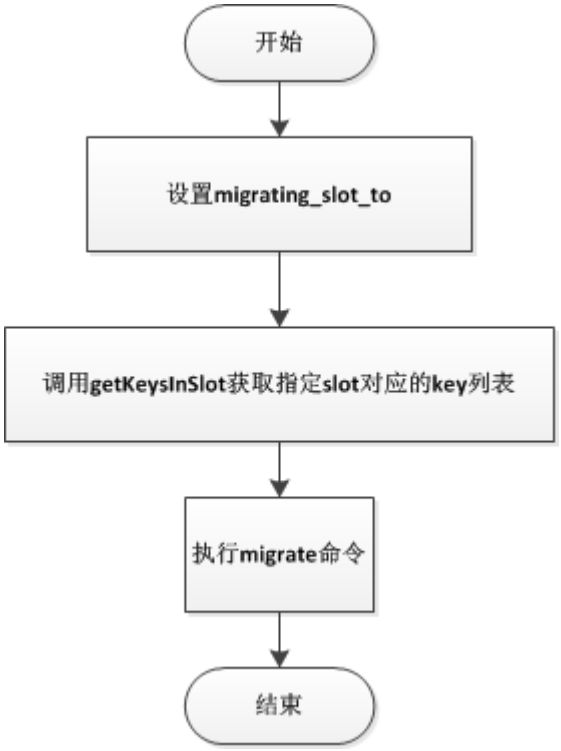
唯一的一个参数是 Redis Cluster 任意一个节点的地址，这个命令发生了如下的过程：

客户端：



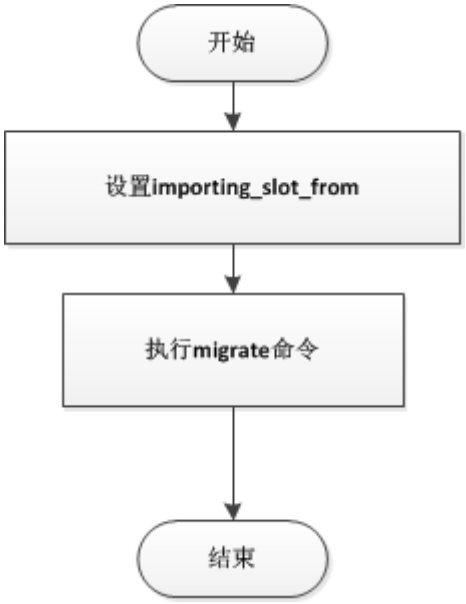
1. 调用 reshard_cluster_cmd 来操作 reshard 的过程
2. 首先会利用第 1 个参数调用 load_cluster_info_from_node 连上 Redis Cluster，对每个节点发送 cluster nodes 命令，加载现有的节点配置信息列表
3. 然后依次输入要 reshard 的哈希槽的数目以及目标节点 ID 和源节点 ID 还有 yes 的交互应答，然后客户端会计算 reshard_table,计算 reshard_table 的策略是在尽量均匀的基础上对于源节点的哈希槽的个数做考量，如果源节点的哈希槽比较多的话那么该源节点就会被移走较多数目的哈希槽
4. 最后遍历 reshard_table，调用 move_slot 对每一个 slot 对应的 key 进行迁移
5. 在 move_slot 里面，首先会对 target 执行命令 cluster setslot slot importing node_id 将 node_id 上面的 slot 导入到 target 上面去；然后对 source 执行命令 cluster setslot slot migrating node_id 将 source 上面的 slot 导入到 node_id 指定的节点上面去，这两个步骤不能调转过来，因为这个命令是为了查询的重定向，如果互调了顺序那么在对 target 执行 cluster setslot slot importing 之前会有 client 重定向到 target，但是 target 却不知道这个情况
6. 在 move_slot 里面，执行完上面的 cluster setslot 之后就开始通过 cluster getkeysinslot 来获取 source 里面的对应 slot 的 key 列表，然后顺序执行 migrate 命令进行迁移
7. 循环执行第 6 步直到 slot 对应的 key 全部被迁移
8. 循环执行 5,6,7 这 3 步直到 reshard_table 遍历完成，这时候整个 reshard 操作就已经完成了

源节点服务端：



1. 在客户端的第 5 步对源节点服务端发送 cluster setslot slot migrating node_id，进入到 clusterCommand 的 setslot 分支，以及下一步的 migrating 子分支，设置数组 server.cluster->migrating_slot_to 的 slot 索引的值为 node_id 指定的 clusterNode，这个数组的意思是记录要迁移到目标节点的槽，以及要迁移的目标节点；clusterState 上面另外一个相近的属性是 importing_slot_from，这个数组的意思是记录要从源节点迁移到本节点的槽，以及要迁移的节点。做这些记录的原因主要是为了在查询的时候可以告诉客户端可以重定向到新节点去
2. 客户端的第 6 步发送 cluster getkeysinslot 会进入 clusterCommand 的 getkeysinslot 分支进行处理，主要就是调用 getKeysInSlot 获取指定 slot 对应的 key 列表
3. 然后就是执行 migrate 命令进行 source 节点和 target 节点的数据迁移了

目标节点服务端：



1. 在客户端的第 5 步对目标节点服务端发送 cluster setslot importing node_id，进入到 clusterCommand 的 setslot 分支，以及下一步的 importing 子分支，设置数组 server.cluster->importing_slot_from 的 slot 索引的值为 node_id 指定的 clusterNode，这个数组的意义上面已经提到，设置了这个值那么该节点在收到相应的 slot 的请求的时候就会进行处理
2. 在源节点每一次执行完 cluster getkeysinslot 后，目标 target 服务端和 source 节点配合执行 migrate 命令
3. 对应 migrate 命令的实现不作赘述

删除从节点或空主节点通过命令：

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

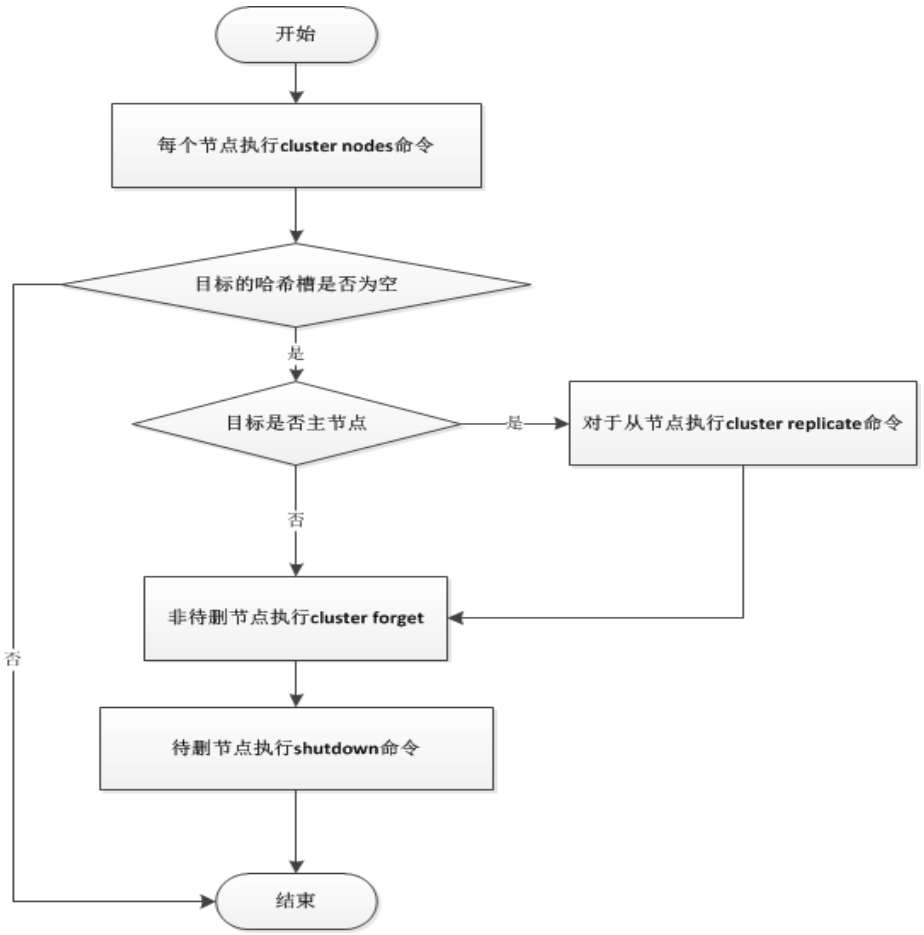
其中第一个参数可以是集群中任意一个节点地址，第二参数是要删除节点 id

删除非空节点的话，先要通过 manual failover 使得该节点变成从节点，然后通过上面的删除从节点的命令来删除，下一个 chapter 再探讨 manual failover，这里先讨论删除一个从节点的实现

首先发送 cluster forget 命令给除了删除节点外其他集群节点，它会使收到该命令的节点将待删除节点从集群节点配置中删除；如果待删除节点是空的主节点，会通过 cluster replicate 命令将它的从节点设为目前从节点数最少的主节点的从节点；最后发送 shutdown 命令给待删除节点，使其自动关闭进程

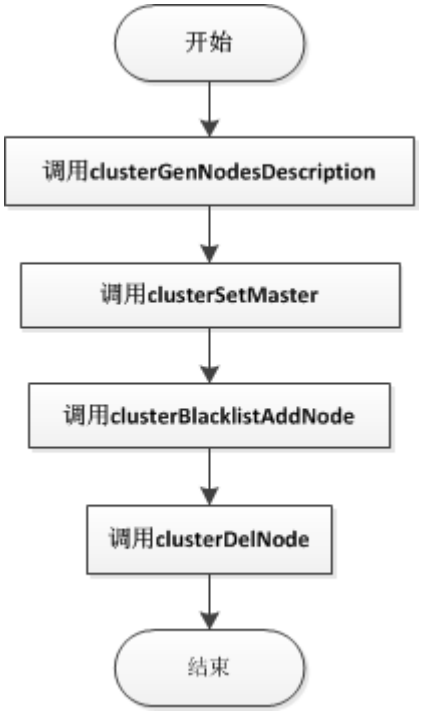
这个命令发生了如下的过程：

客户端：



1. 调用 delnode_cluster_cmd 来进行删除节点的操作
2. 在 delnode_cluster_cmd 首先会利用第一个参数调用 load_cluster_info_from_node，也就是发送 cluster nodes 命令得到现有的节点信息列表
3. 然后根据第 2 个参数节点 ID 从列表里面得到相应的节点信息，然后根据节点信息判断它的哈希槽是不是为空的，如果不是空的哈希槽的话就提示错误然后退出客户端
4. 遍历所有节点，如果被删除的节点是主节点，那么就把它的从节点通过 cluster replicate 命令设置为拥有最少从节点的非空主节点的从节点，然后除了待删节点的每个节点都发送 cluster forget 命令
5. 最后待删节点执行 shutdown 命令，关闭待删节点的 redis 进程

非待删节点服务端：



1. 对于客户端第 2 步的 cluster nodes 命令前面已经讨论过这里不再重复
2. 客户端第 4 步的 cluster replicate 在服务端会进入 clusterCommand 的 replicate 分支里面，在检查了参数合法之后调用 clusterSetMaster(cluster.c:4138)把客户端第 4 步指定节点设置为当前节点的主节点
3. 在 clusterSetMaster 里面首先调用 clusterNodeAddSlave 把自身加到指定节点的 Slave 列表里面去
4. 然后调用 replicationSetMaster 设置当前节点的 Master 节点的 IP 和 PORT
5. 最后调用 resetMannualFailover 重置手动容错的相关参数
6. 在第 4 步设置好 Master 节点的地址后，剩下 replicate 的工作就交给 replicationCron(replication.c:1938)去完成了
7. 非待删节点服务端在收到 cluster forget 命令后会进入 clusterCommand 的 forget 分支里面，在检查参数合法之后调用 clusterBlacklistAddNode，把待删节点加入到当前节点的 back_list 里面，设置超时时间为 REDIS_CLUSTER_BLACKLIST_TTL，即系 60 秒，也就是说 redis-trib 脚本有 60 秒的时间把 cluster forget 命令发往所有非待删节点而不用担心这些节点在这段时间内重新把待删节点添加到 Cluster 内

8. 然后调用 clusterDelNode，里面主要的工作就是做一些清理，包括 clusterState 的两个属性，长度为 REDIS_CLUSTER_SLOTS 的 clusterNode*的数组 migrating_slots_to 和 importing_slots_from,数组的索引表示哈希槽，数组的元素的值分别表示从当前节点迁移到目标节点的相应的哈希槽的 clusterNode 的指针和从源节点迁移到当前节点相应的哈希槽的 clusterNode 的指针，如果这两个数组某些槽位的值是待删节点的 clusterNode*的值那么就把它置为 NULL；然后清除 clusterState 里面的 nodes 里面的每个 clusterNode*的 fail_reports 属性，这个是个 list，那么清除掉关于待删节点的值；最后就是断开 TCP，释放内存的操作了

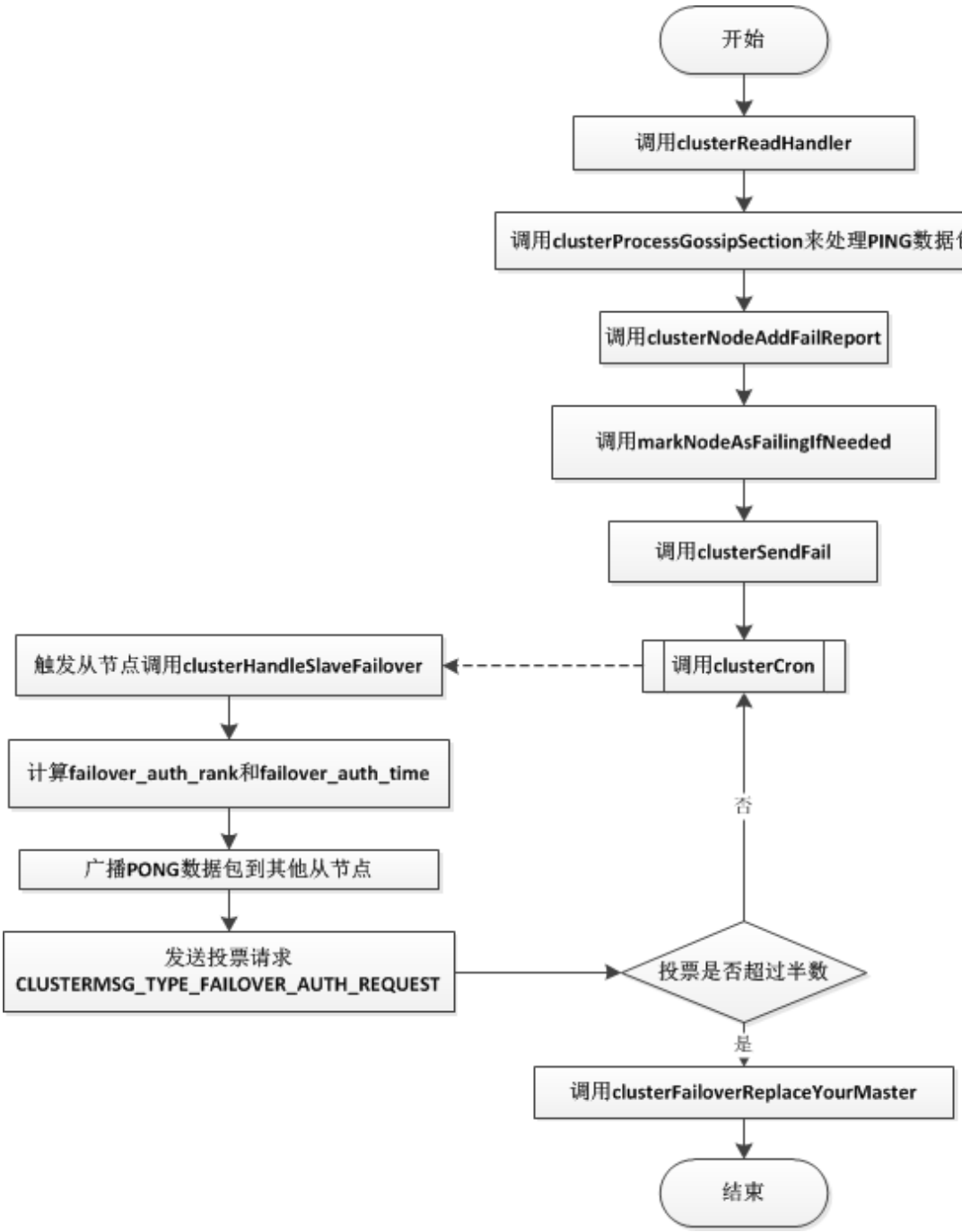
待删节点服务端：

1. 执行 shutdown 命令，和非 cluster 版本的 redis 一样，不做赘述

3. Fault Tolerance

Failover 是 Redis Cluster 的容错机制，是 Redis Cluster 的最核心的功能之一；它允许在某些节点失效情况下，Cluster 还能正常提供服务。容错机制本质上就是主从切换机制，即主节点失效后选举一个从节点作为新的主节点；另外 Redis Cluster 也支持手动 Failover，将从节点变为主节点即使主节点还活着，Redis Cluster 的容错的实现依赖于 Gossip 协议和类 Raft 的协议，Gossip 用于节点间感知和失败判定类 Raft 协议用于节点选举，Failover 主要分为主节点失效产生的自动 Failover 和 Manual Failover

1. 主节点失效产生的 Failover：
   a. 集群中的节点会通过定时任务 clusterCron 向其他节点发送 PING 数据包，并且接受来自 Cluster Port 的连接请求，同时对接收的 PING 数据包进行回复
      当一个节点向另外一个节点发送 PING 数据包，但是目标节点未能在 server.cluster_node_timeout 内回复，那么发送 PING 数据包的节点会将目标节点标记为 PFail(possible fail)
      在发送 PING 数据包的同时，会告诉目标节点集群中已经被标记 PFail 或者 Fail 的节点(cluster.c:2228)
      相应地，当节点收到其他节点发来的 PING 消息时，它会记下那些被其他节点标记为失效的节点(cluster:1314)，调用 clusterNodeAddFailReport
      如果某个节点被当前节点标记为 PFail，当前节点在调用 clusterNodeAddFailReport 之后会调用 markNodeAsFailingIfNeeded(cluster.c:1319)来检查检查是否超过半数的节点把该节点标记为 PFail，如果已经超过了半数的节点这样认为那么该节点会被标记为 Fail 并且该节点失效的信息会被广播到整个集群 clusterSendFail(cluster.c:1169)，所有收到这个信息的节点回将该节点标记为 Fail
   b. 一旦某主节点进入 Fail 状态，集群变为 Fail 状态，同时会触发 Failover(通过 clusterCron 里面的 clusterHandleSlaveFailover)，重新选出新的主节点，使得集群恢复正常继续提供服务：



a) 当前节点会检查自身是否属于已下线主节点的从节点；已下线主节点的哈希槽是否非空；主从节点的最后交互时间是否超时

b) 计算 failover_auth_rank 和 failover_auth_time，然后广播 PONG 数据包到这个被标记为 FAIL 的主节点的从节点去(CLUSTER_BROADCAST_LOCAL_SLAVES)，这些从节点就可以得到当前节点的 offset，下一次进入 clusterHandleSlaveFailover 的时候就会进行这样的操作 server.cluster->currentEpoch++ 然后发送 CLUSTERMSG_TYPE_FAILOVER_AUTH_REQUEST(cluster.c:2821)也就是发送投票请求

c) 每次调用定时任务 clusterCron 进入 clusterHandleSlaveFailover 都会检查 auth_age 是否超时，如果超时了就会重置 failover_auth_time

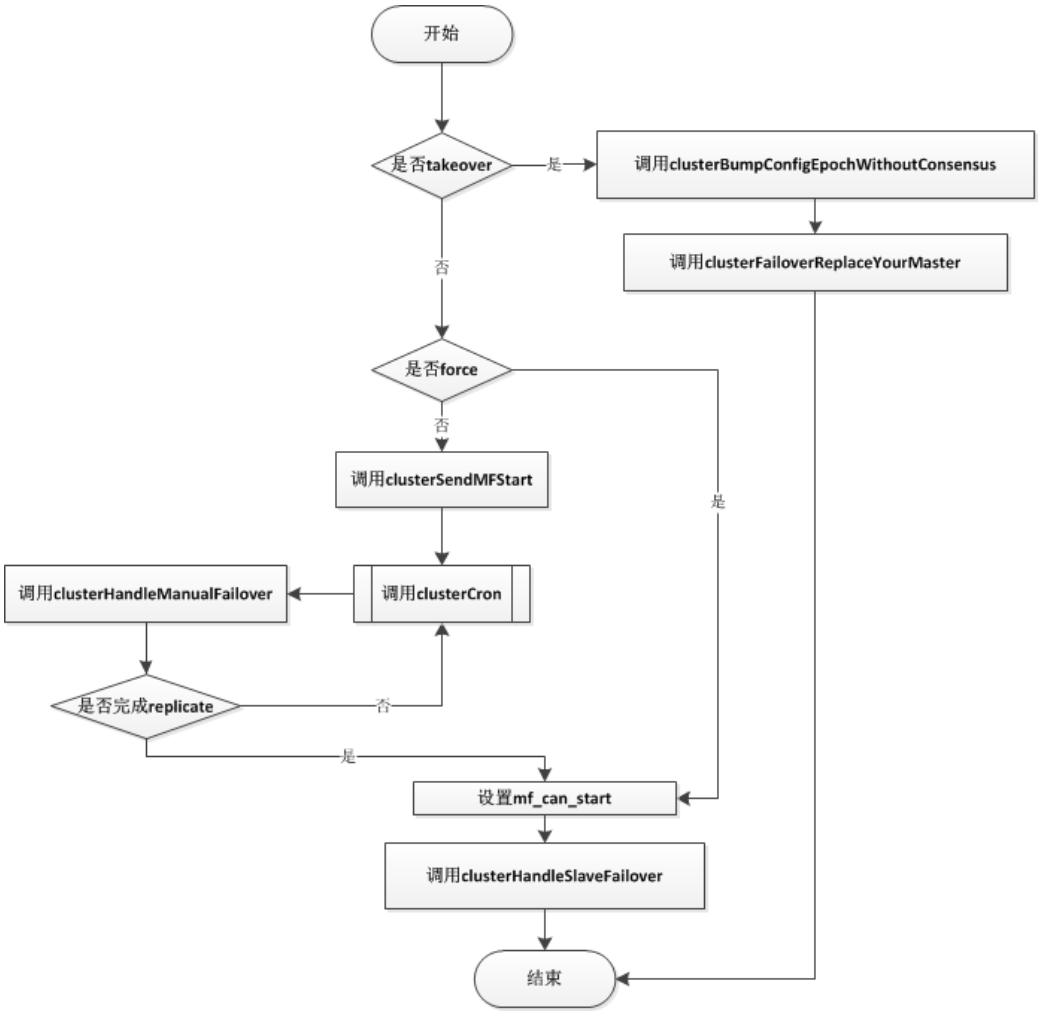
d) 当发现收集到的票数超过了半数(cluster.c:2830)就会把自己置为新的主节点也

就是调用 clusterFailoverReplaceYourMaster(cluster.c:2845)

c. 集群中的其他主节点在收到 VOTE 请求的时候，会进行判定是否接受请求：
   a) 当前节点是否主节点；是否分配有哈希槽；请求带来的 ConfigEpoch 是否大于等于节点保存的 ConfigEpoch；对于 currentEpoch 是否已经投过票了
   b) 检查发送请求的节点是否是主节点；是否可以找到发送节点的主节点；发送节点的主节点是否已经恢复
   c) 遍历请求带来的哈希槽列表，确保它们所对应的节点的 ConfigEpoch 都比请求带来的 ConfigEpoch 早
   d) 给发送请求的从节点投票调用 clusterSendFailoverAuth(cluster.c:2543)；更新投票的 Epoch 和时间戳

2. Manual Failover：
   a. Manual Failover 是一种运维功能，允许手动设置从节点为新的主节点，即使主节点还活着，Manual Failover 和上面讨论的主节点失效产生的 Failover 有两点不同
      a) Manual Failover 通过在客户端发送 cluster failover 来触发，而且接收命令的节点只能是从节点
      b) Manual Failover 不需要主节点失效，Failover 的时长为 5 秒，只有收到命令的从节点才会开始申请
   b. Manual Failover 通过命令 cluster failover 触发，会发生如下流程：

执行命令的从节点服务端：



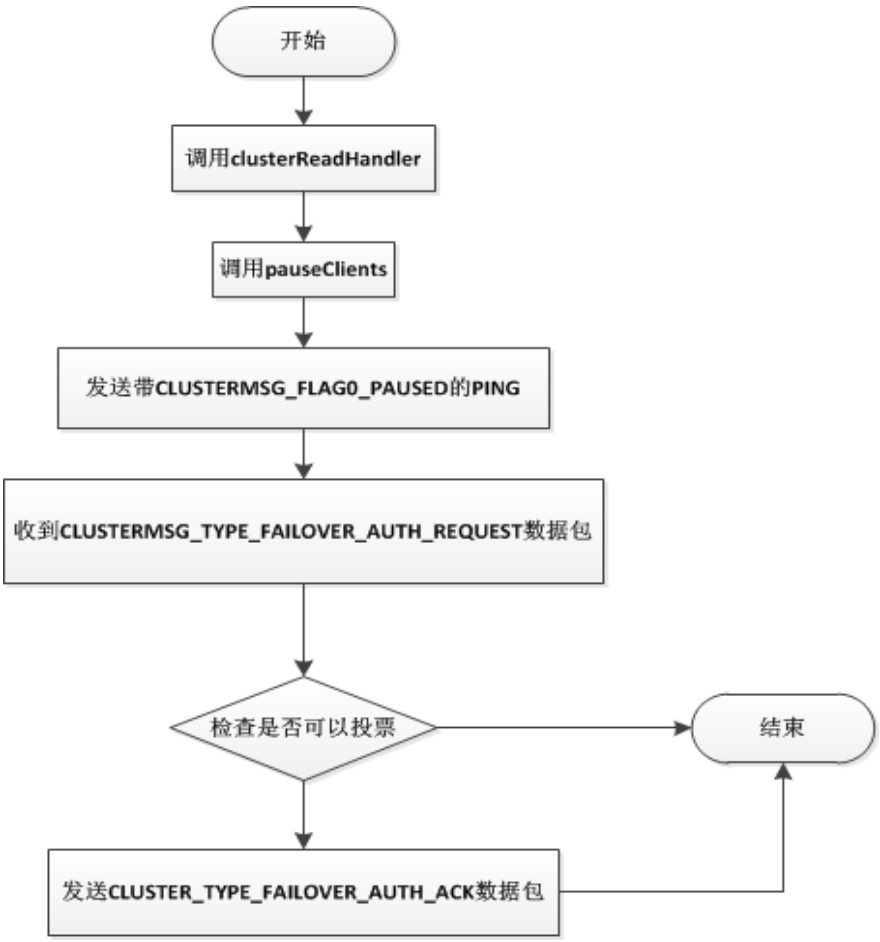
a) 从节点服务端会在 clusterCommand 里面转到 failover 分支(cluster.c:4175)，将 server.cluster->mf_end 设置为超时时间，也就是当前时间戳加上 REDIS_CLUSTER_MF_TIMEOUT(5s)，检查：

b) 是否直接 takeover 接管主节点，在当次事件循环中调用 clusterBumpConfigEpochWithoutConsensus 来更新 ConfigEpoch 然后调用 clusterFailoverReplaceYourMaster 广播 PONG 消息来宣告接管了主节点，需要用户来保证 ConfigEpoch 的一致性

c) 还是 force 启动 Manual Failover，忽略是否 replicate 完成，随后进入 clusterHandleSlaveFailover 进行和“主节点失效产生的 Failover”一样的流程，广播 offset 到主节点的其他从节点，发送选举请求，等待选举完成，更新 ConfigEpoch，接管主节点

d) 抑或是执行一个完整 Manual Failover 流程，先调用 clusterSendMFStart(cluster.c:4227)给主节点发送 CLUSTERMSG_TYPE_MFSTART 数据包通知主节点当前从节点要开始 Failover，之后的从节点会收到来自主节点带了 CLUSTERMSG_FLAG0_PAUSED 的 PING 数据包，然后更新 server.cluster->mf_master_offset，当从节点的 clusterHandleManualFailover 发现主从节点的 replication 的 offset 相等 (cluster.c:3008) 的时候设置 server.cluster->mf_can_start 为 1，然后下一步会进入 clusterHandleSlaveFailover 进行和“主节点失效产生的 Failover”一样的处理

主节点的服务端(最完整的的 Manual Failover 流程，不带 force 或者 takeover 选项)：



a) 收到 CLUSTERMSG_TYPE_MFSTART 后，会设置 server.cluster->mf_end 为当前时

间戳加上 REDIS_CLUSTER_MF_TIMEOUT(5s)，设置 server.cluster->mf_slave 为发送该消息的 clusterNode，调用 pauseClients(cluster.c:1932)停止对客户端服务 2 倍 REDIS_CLUSTER_MF_TIMEOUT 的时间

b) 在 clusterCron 中，向从节点发送带 CLUSTERMSG_FLAG0_PAUSED 标记的 PING 数据包(在 clusterBuildMessageHdr 里面设置该标记 cluster.c:2140)，向该从节点通知其停止对客户端服务时的 replication 偏移量 offset
c) 等待主从复制完成
d) 主节点收到该从节点的投票请求，也就是 CLUSTERMSG_TYPE_FAILOVER_AUTH_REQUEST 数据包，包括其他的主节点也会收到这个投票请求，都会检查是否满足投票条件，也就是调用 clusterSendFailoverAuthIfNeeded，如果满足投票条件就会发送 CLUSTER_TYPE_FAILOVER_AUTH_ACK 给发起投票的从节点，前面“主节点失效产生的 Failover”的第“c”点已经探讨过进行投票需要遵循哪些限制，这里不再赘述

4. Consensus

Redis Cluster 的全局相关的配置是描述整个集群的元数据，例如节点地址，主节点所负责的哈希槽，主从关系等等需要各个节点保持一致，Redis Cluster 只保证配置数据最终一致性，它为了达到这个目标采取了两种措施：

1. 每个节点都有 ConfigEpoch，用来表示该节点当前哈希槽配置的版本，如果哈希槽配置发生变化则 ConfigEpoch 也会发生变化，而且是递增的，这可以保证其他节点根据 ConfigEpoch 来同步更新为最新的哈希槽配置；另外从节点跟随主节点，自身是没有哈希槽配置的
2. 每个节点定时发送 PING 数据包给其他节点，PING 数据包第一部分内容包括发送者自身的节点地址，名字状态，哈希槽配置，主从关系，ConfigEpoch 等等；第二部分内容是其他节点信息，包括节点地址，名字，状态等等，为了保证 PING 数据包的不太大，所以每次只会随机选取 3 个节点。每次会随机 5 个节点来发送这样的数据包

通过这两种措施来保证 Redis Cluster 的配置信息是最新的并且最终一致

哈希槽的更新需要 ConfigEpoch 配合，如果收到的 ConfigEpoch 大于本地的 ConfigEpoch 那么就更新哈希槽(cluster.c:1807)；PING 数据包里面对其他节点的处理，通过 clusterProcessGossipSection 进行处理，如果地址变了或者是新节点则需要调用 clusterStartHandShake，状态信息用来判断 FAIL 或者 PFAIL 状态

5. Compare to others

下面从一致性，容错性，可用性，扩展性，复制效率等等方面来比较一下 Redis Cluster 和 RamCloud

	Redis Cluster	Hbase
一致性	只支持最终一致性，会有丢失数据的情况发生	
容错性	在主节点失效后可以自动选举新的主节点出来也可以手动切换主从节点	
可用性	可以通过对每个主节点配置多个从节点，每个从节点部署在不同的机架的机器上来提供可用性，主节点至少会产生 cluster_node_timeout 时长的失效才会开始进行 Failover	
扩展性	Redis Cluster 可以轻易扩展到 1k 左右的节点，添加节点后可以通过脚本工具 redis-trib.rb 把哈希槽 reshard，还没法做到自动负载均衡，在 reshard 的时候不会中断服务	
复制效率	Redis Cluster 主从节点的复制建议开启 AOF 关闭 BGSAVE，原因应该是尽量减少 IO 避免产生 Cluster 方面的通讯延迟，Redis Cluster 的复制效率和单机 Redis 的没区别	