集群

Redis集群是redis提供的分布式数据库方案，集群通过分片（sharding）来进行数据共享，并提供复制和故障转移的功能。

节点

一个redis集群通常由多个节点（node）组成。刚开始时，每个节点相互独立处于一个仅包含自己的一个集群中。要组建一个真正的集群，必须将各个节点连接起来，构成一个包含多个节点的集群。

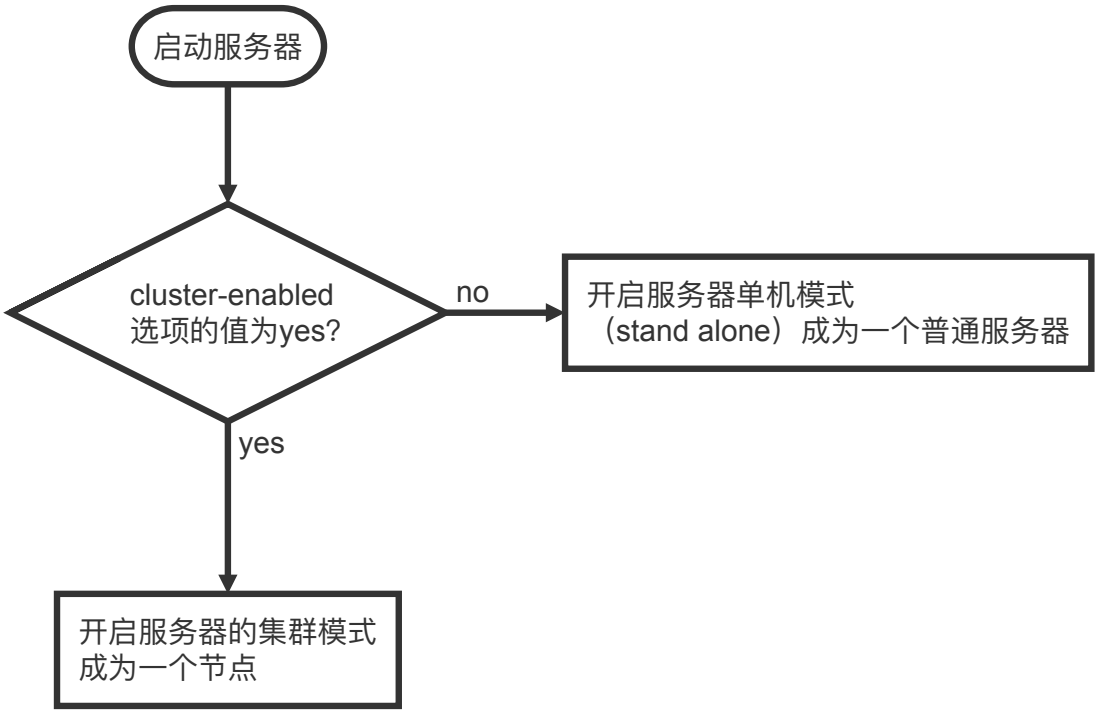
连接各个节点可以使用命令：cluster meet 来完成。

> cluster meet

向一个节点发送cluster meet命令时，可以让node节点与 ip和port 所制定的节点进行握手（handshake），当握手成功后，node节点会将ip和port所指定的节点添加到当前所在集群中。

启动节点

一个节点就是运行在集群模式下的一个redis服务器，Redis服务器再启动时会根据cluster-enabled配置选项是否为yes来决定是否开启服务器的集群模式。



节点（运行在集群模式下的Redis服务器）会继续使用所有在单机模式中使用的服务器组件，比如说：

- 节点会继续使用文件事件处理器来处理 命令请求和命令回复
- 节点会继续使用时间事件处理器来执行serverCron函数，而serverCron函数又会调用集群模式特有的clusterCron函数。clusterCron函数负责执行在集群模式下需要执行的常规操作，例如想集群

中的其他节点发送Gossip消息，检查节点是否短线，或者检查是否需要对下线节点进行自动故障转移

- 节点会继续使用数据库来保存键值对数据，键值对依然会是各种不同类型的对象
- 节点会继续使用RDB持久化模块和AOF持久化模块来执行持久化工作
- 节点会继续使用发布与订阅模块来执行PUBLISH、SUBSCRIBE等命令
- 节点会继续使用复制模块来进行节点的复制工作
- 节点会继续使用Lua脚本环境来执行客户端输入的Lua脚本

除此以外，节点会继续使用redisServer结构来保存服务器的状态，使用redisClient结构来保存客户端的状态，至于那些只有在集群模式下才会用到的数据，节点将它们保存到了cluster.h/clusterNode结构、cluster.h/clusterLink结构，以及cluster.h/clusterState结构里面。

集群数据结构

clusterNode 结构保存了一个节点的当前状态，比如节点的创建时间、节点名字、节点当前的配置纪元、节点的ip与port等等。

每个节点都会使用一个clusterNode结构来记录自己的状态，并为集群中所有其他的节点（包括主节点和从节点）都创建一个响应的clusterNode结构，以记录其他节点的状态：

```
typedef struct clusterNode {
    mstime_t ctime; /* Node object creation time. */
    // 节点名称，由40个十六进制字符组成
    char name[CLUSTER_NAMELEN]; /* Node name, hex string, sha1-size */
    // 节点标识
    // 使用各种不容的标识值记录节点的角色（主节点、从节点）
    // 以及节点当前所处状态（在线、下线）
    int flags;      /* CLUSTER_NODE_... */
    // 节点当前配置纪元，用于故障转移
    uint64_t configEpoch; /* Last configEpoch observed for this node */
    unsigned char slots[CLUSTER_SLOTS/8]; /* slots handled by this node */
    int numslots;   /* Number of slots handled by this node */
    int numslaves;  /* Number of slave nodes, if this is a master */
    struct clusterNode **slaves; /* pointers to slave nodes */
    struct clusterNode *slaveof; /* pointer to the master node. Note that it
                                    may be NULL even if the node is a slave
                                    if we don't have the master node in our
                                    tables. */
    mstime_t ping_sent;      /* Unix time we sent latest ping */
    mstime_t pong_received;  /* Unix time we received the pong */
    mstime_t fail_time;      /* Unix time when FAIL flag was set */
    mstime_t voted_time;     /* Last time we voted for a slave of this master
*/
    mstime_t repl_offset_time;  /* Unix time we received offset for this node
*/
    mstime_t orphaned_time;     /* Starting time of orphaned master condition
*/
    long long repl_offset;      /* Last known repl offset for this node. */
    char ip[NET_IP_STR_LEN];  /* Latest known IP address of this node */
```

```
    int port;                   /* Latest known clients port of this node */
    int cport;                  /* Latest known cluster port of this node. */
    clusterLink *link;          /* TCP/IP link with this node */
    // 保存连接节点所需的有关信息
    list *fail_reports;         /* List of nodes signaling this as failing */
} clusterNode;
```

clusterNode结构的link属性时一个clusterLink结构，该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：

```
typedef struct clusterLink {
    mstime_t ctime;             /* Link creation time */
    int fd;                     /* TCP socket file descriptor */
    // 发送给其他节点的消息
    sds sndbuf;                 /* Packet send buffer */
    // 从其他节点接收的消息
    sds rcvbuf;                 /* Packet reception buffer */
    // 与这个连接相关联的node，没有则为null
    struct clusterNode *node;   /* Node related to this link if any, or NULL
*/
} clusterLink;
```

> ### redisClient结构和clusterLink结构的相同与不同之处
>
> redisClient结构和clusterLink结构都有自己的套接字描述符与输入、输出缓冲区，这两个结构的区别在于，redisClient结构中的套接字和缓冲区是用来连接客户端的，而clusterLink结构中的套接字和缓冲区则用户连接节点的

每个节点都保存着一个clusterState结构，这个结构记录了在当前节点的视角下，集群目前所处的状态，例如集群时在线还是下线，集群包含多少个节点，集群当前的配置纪元，诸如此类：

```
typedef struct clusterState {
    // 指向当前节点的指针
    clusterNode *myself;  /* This node */
    // 集群当前纪元，用于实现故障转移
    uint64_t currentEpoch;
    // 当前集群状态，上线还是下线
    int state;            /* CLUSTER_OK, CLUSTER_FAIL, ... */
    // 集群中至少处理着一个槽点节点的数量
    int size;             /* Num of master nodes with at least one slot */
    // 集群节点名单
    // 字典的键为节点的名字，字典的值为节点对应的clusterNode结构
    dict *nodes;          /* Hash table of name -> clusterNode structures */
    dict *nodes_black_list; /* Nodes we don't re-add for a few seconds. */
    clusterNode *migrating_slots_to[CLUSTER_SLOTS];
    clusterNode *importing_slots_from[CLUSTER_SLOTS];
    clusterNode *slots[CLUSTER_SLOTS];
    uint64_t slots_keys_count[CLUSTER_SLOTS];
    rax *slots_to_keys;
```

```
    /* The following fields are used to take the slave state on elections. */
    mstime_t failover_auth_time; /* Time of previous or next election. */
    int failover_auth_count;    /* Number of votes received so far. */
    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. */
    int cant_failover_reason;   /* Why a slave is currently not able to
                                   failover. See the CANT_FAILOVER_* macros.
*/
    /* Manual failover state in common. */
    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. */
    clusterNode *mf_slave;      /* Slave performing the manual failover. */
    /* Manual failover state of slave. */
    long long mf_master_offset; /* Master offset the slave needs to start MF
                                   or zero if stil not received. */
    int mf_can_start;           /* If non-zero signal that the manual failover
                                   can start requesting masters vote. */
    /* The followign fields are used by masters to take state on elections. */
    uint64_t lastVoteEpoch;     /* Epoch of the last vote granted. */
    int todo_before_sleep; /* Things to do in clusterBeforeSleep(). */
    /* Messages received and sent by type. */
    long long stats_bus_messages_sent[CLUSTERMSG_TYPE_COUNT];
    long long stats_bus_messages_received[CLUSTERMSG_TYPE_COUNT];
    long long stats_pfail_nodes;    /* Number of nodes in PFAIL status,
                                       excluding nodes without address. */
} clusterState;
```

以下展示以7000节点创建的clusterState结构，这个结构从节点7000点角度记录了集群以及集群包含的三个节点的当前状态（省略部分属性）：

- 结构的currentEpoch属性为0，表示集群当前的配置纪元为0
- 结构的size属性的值为0 ，表示集群目前没有任何节点在处理槽，因此结构的state属性的值为REDIS_CLUSTER_FALL，这表示集群目前处于下线状态
- 结构的nodes字典记录了集群目前包含的三个节点，这三个节点分别由三个clusterNode结构来表示，
- 三个节点的clusterNode结构的flags属性都是REDIS_NODE_MASTER，说明三个节点都是主节点

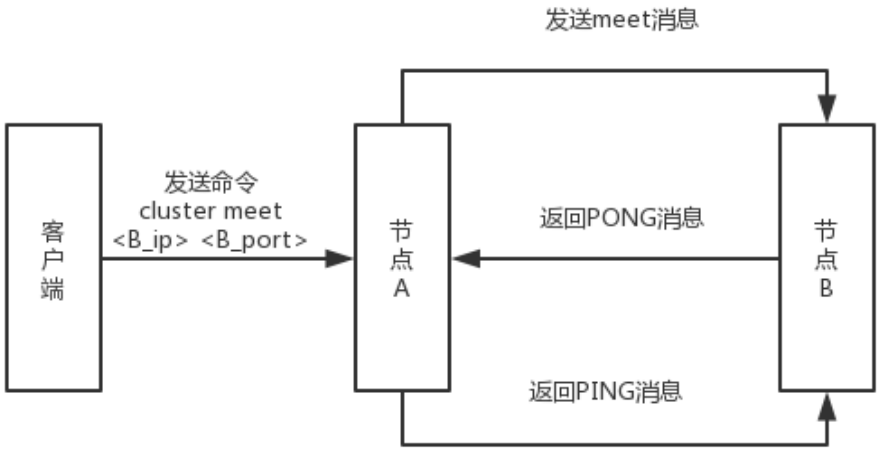
CLUSTER MEET命令实现

通过向节点A发送cluster meet 命令，客户端可以让接受命令等节点A将另一个节点B添加到A当前所在的集群里面：

```
cluster meet <ip> <port>
```

收到命令的节点A将于节点B进行握手（handshake），以此来确认彼此的存在，并为将来的进一步通信打好基础：

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.node字典中。
2. 之后，节点A根据cluster meet命令给定的ip与port，向节点B发送一条meet消息（message）。
3. 如果一切顺利，节点B将接收到节点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.node字典中。
4. 之后，节点B向节点A返回一条pong消息。
5. 如果一切顺利，节点A将接收到节点B返回到pong消息，通过这条pong消息节点A可以知道节点B已经成功地接收到了自己发送到meet消息。
6. 之后，节点A将向节点B发送一条PING消息
7. 如果一切顺利，节点B将接收到节点A返回到ping消息，通过这条ping消息节点B可以知道节点A已经成功地收到了自己返回到pong消息，握手完成。



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，经过一段时间之后，节点B会被集群中所有节点认识。

槽指派

redis集群通过分片的方式来保存数据库中的键值对：集群的整个数据库被氛围16384个槽（slot），数据库中每个键都属于这16384个槽中的一个，集群中的每个节点可以处理0个或最多16384个槽

当数据库中16384个槽都有节点在处理时，集群处于上线状态（OK）；若存在一个槽没有得到处理，则集群处于下线状态（fail）。

通过节点发送 cluster addslots 命令，我们可以将一个或多个槽指派（assign）给节点负责：

```
127.0.0.1:7000> cluster addslots 0 1 2 3 4 ... 7
```

同理，可以将剩余的槽分配给其他节点。

将所有的槽分配给其他节点的时候，集群处于上线状态，可以使用以下命令查看：

```
# 查看集群状态
127.0.0.1:7000> cluster info
cluster_state:ok
cluster_slots_assigned:16384
cluster_slots_ok:16384
cluster_slots_pfail:0
cluster_slots_fail:0
cluster_known_nodes:3
cluster_size:3
cluster_current_epoch:0
cluster_stats_messages_sent:2699
cluster_stats_messages_received:2617

# 查看节点信息
127.0.0.1:7000> cluster nodes
```

记录节点的槽指派信息

clusterNode 结构的slots属性和numslots属性记录了节点负责处理哪些槽：

```
struct clusterNode {
  // ...
  unsigned char slots[16384/8];

  int numslots;
  // ...
};
```

slots 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，共包含了16384个二进制位。

Redis以0为索引，16383为终止索引，对slots数组中的16384个二进制位进行编号，并根据索引i上的二进制位的值来判断节点是否负责处理槽i。

字节	slots[0]								slots[1]~slots[2047]							
索引	0	1	2	3	4	5	6	7	8	9	10	11	...	16381	16382	16383
值	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0

上图展示了，当前节点仅处理0～7号槽。

因为取出和设置slots数组中任意一个二进制位的值的时间复杂度位O(1)，所以对于一个给定节点的slots数组来说，程序检查节点是否负责处理某个槽，又或者将某个槽指派给节点负责，这两个动作的时间复杂度都是O(1)。

传播节点的槽指派信息

一个节点除了会将自己负责处理的槽记录在clusterNode结构的slots属性和numslots属性外，它还会将自己的slots数组通过消息发送给集群中其他节点，以此来告知其他节点自己目前负责处理哪些槽。

因此集群中的每个节点都知道数据库中的16384个槽分别被指派给了集群中的哪些节点

记录集群所有槽点指派信息

clusterState结构中的slots数组记录了集群中所有16384哥槽点指派信息：

```
typedef struct clusterState{
  // ...
  clusterNode *slots[16384]
  // ...
}clusterState;
```

slots数组包含了16384个项，每个数组项都是一个指向clusterNode 结构的指针：

- 如果 slots[i] 指针指向null，那么表示槽i尚未指派给任何节点
- 如果 slots[i] 指针指向一个clusterNode结构，那么表示槽 i 已经指派给了clusterNode结构所代表的节点

以上为例，slots[0] ~ slots[7] 指向 上图图示中 clusterNode ，而其他的数组各自指向对应的 clusterNode 。

如果只将槽指派信息保存在各个节点的clusterNode.slots数组里，会出现一些无法高效地解决的问题，而clusterState.slots 数组的存在解决了这些问题：

- 如果节点只使用 clusterNode.slots 数组 来记录槽的指派信息，那么为了知道槽 i 是否已经被指派，或者槽 i 被指派给了哪个节点，程序需要遍历clusterState.nodes 字典中的所有 clusterNode 结构，检查这些结构的slots 数组，知道找到负责处理 槽 i 的节点位置，整个过程的时间复杂度是 O(n) ，其中n为clusterState.nodes 字典保存的 clusterNode 结构的数量。
- 而通过将所有槽点指派信息保存在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，时间复杂度为 O(1)。

虽然 clusterState.slots 数组记录了集群中所有槽点指派信息，但使用 clusterNode 结构的 slots 数组来记录单个节点的槽指派信息仍然是必要的：

- 因为当程序的要将 某个节点 的槽指派信息通过消息发送给其他节点时，程序只需要将相应节点的 clusterNode.slots 数组整个发送出去就可以了。
- 另一方面，如果redis 不是用 clusterNode.slots 数组，而单独使用 clusterState.slots 数组的话，那么每次要将节点A 的槽指派信息传播给其他节点时， 程序必须先遍历整个clusterState.slots 数组，记录节点A 负责处理哪些槽，然后才能发送节点A 的槽指派信息。

clusterState.slots 记录了集群中所有槽点指派信息，而 clusterNode.slots 数组只记录了 clusterNode 结构所代表的槽指派信息。

cluster addslots 命令的实现

cluster addslots 命令接受一个或多个槽作为参数，并将所有输入的槽指派给接收该命令的节点负责：

```
cluster addslots <slots> [slot ...]
```

该命令的实现可由以下伪码表示：

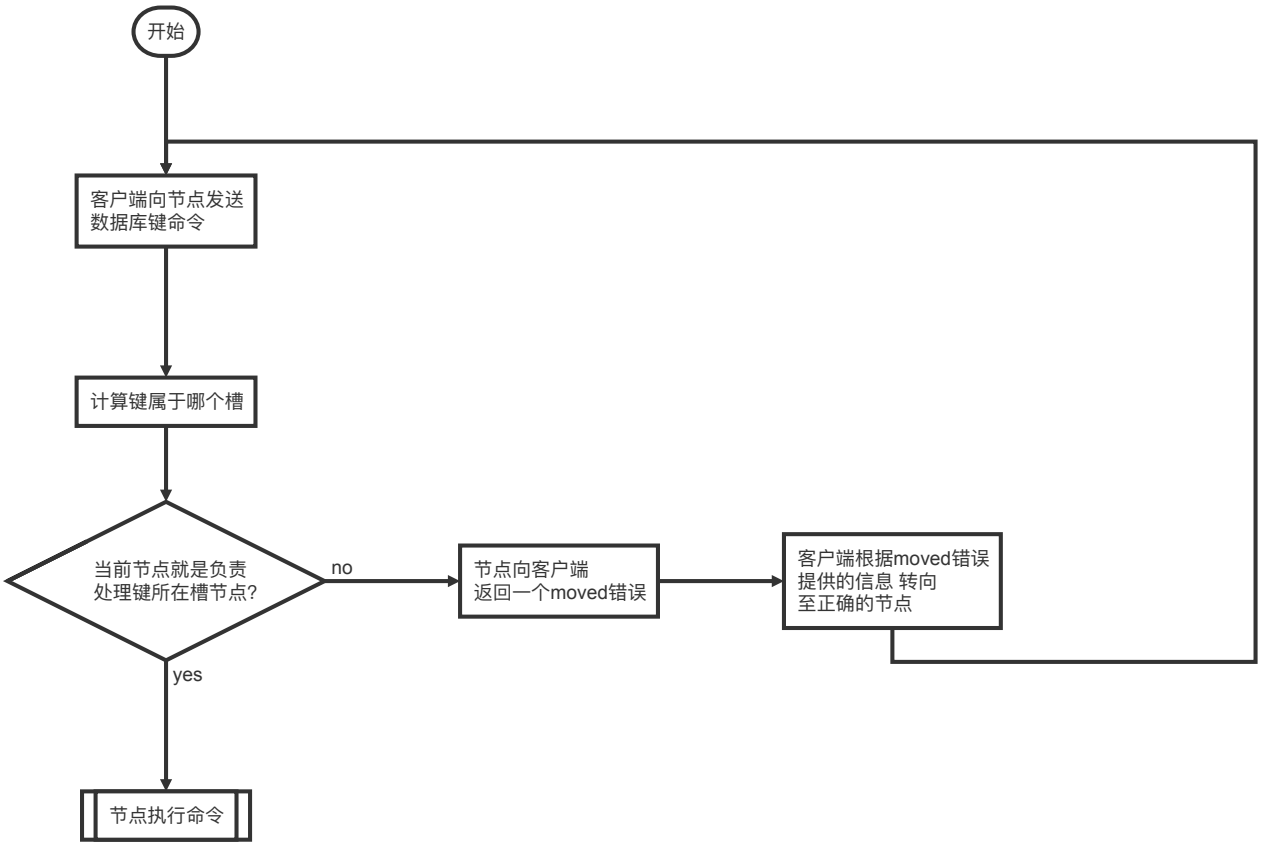
```
def CLUSTER_ADDSLOTS(*all_input_slots):
  // 遍历输入的所有槽
  for i in all_input_slots :
      // 如果有槽已经被指派，则向客户端返回错误，并终止命令执行
      if clusterState.slots[i] != null:
          reply_error()
          return

  // 若均未指派，则将这些槽指派给当前节点
  for i in all_input_slots :
    // 设置clusterState结构的slots数组
    // 将slots[i] 的指针指向代表当前节点的 clusterNode 结构
    clusterState.slots[i] = clusterState.myself

    // 访问代表当前节点的clusterNode结构的slots数组
    // 将数组在索引i上的二进制位设置为1
    setSlotsBit(clusterState.myself.slot,i)
```

最后，当该命令执行完毕后，节点会通过发送消息告知集群中的其他节点，自己目前正在负责处理的槽。

在集群中执行命令

当客户端向节点发送与数据库键有关的命令时，接收命令的节点回计算出命令要处理的数据库键属于哪个槽，并检查这个槽是否指派给了自己：

- 如果键所在的槽正好就指派给了当前节点，那么节点直接执行命令
- 如果键所在的槽并没有指派给当前节点，那么节点会向客户端返回一个MOVED错误，指引客户端转向（redirect）至正确的节点，并再次发送之前想要执行的命令。



计算键属于哪个槽

节点使用以下算法来计算给定键key属于哪个槽：

```
def slot_number(key):
  return CRC16(key) & 16383
```

其中 CRC16(key) 语句用于计算键key的CRC-16校验和，而 & 16383 语句则用于取模计算出 一个 介于 0～16383 之间的槽号。

可以使用以下命令来查看给定键属于的槽：

```
127.0.0.1:7000 > cluster keyslot "msg"
```

该命令也是由以上算法实现的，以下为伪码实现：

```
def cluster_keyslot(key) :
  # 计算槽号
  slot = slot_number(key)
  # 将槽号返回客户端
  reply_client(slot)
```

判断槽是否由当前节点负责

当节点计算出键所属槽 i 之后，节点就会检查自己在clusterState.slots 数组中的项 i ，判断键所在的槽是否由自己负责：

1. 如果clusterState.slots[i] 等于 clusterState.myself ，那么说明槽 i 由当前节点负责，节点可以执行客户端发送的命令
2. 如果 clusterState.slots[i] 不等于 clusterState.myself ，那么说明槽 i 并非由当前节点负责，阶段会根据 clusterState.slots[i] 指向的 clusterNode 结构所记录的节点 IP 与 port ， 会向客户端返回 moved 错误，指引客户端转向至正在处理槽 i 的节点。

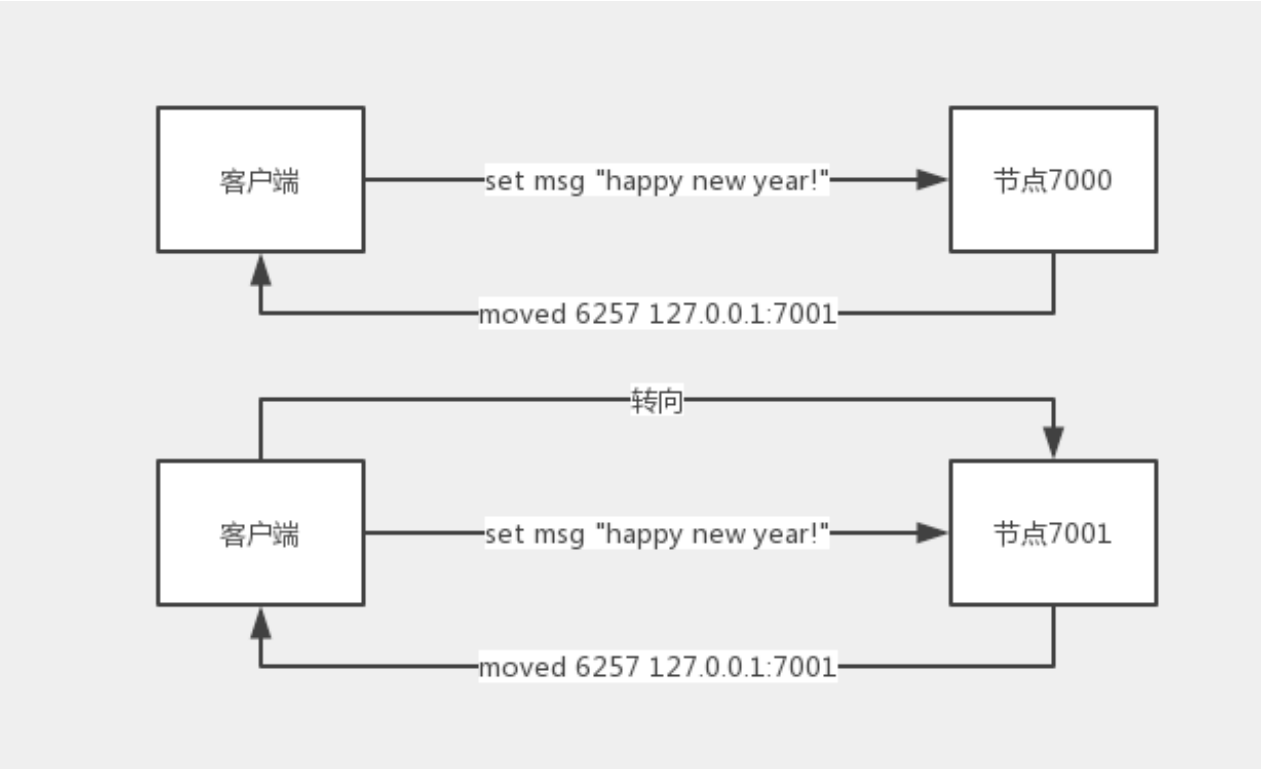
MOVED 错误

当客户端发现键所在的槽并非由自己处理时， 节点就会向客户端返回一个 MOVED 错误，指引客户端转向至正在负责槽的节点。

MOVED错的的格式为：

```
MOVED <slot> <ip>:<port>
```

当客户端接收到节点返回的moved错误时，客户端会根据moved错误中提供的IP地址和端口号，转向至负责处理槽slot节点，并向该节点重新发送之前想要执行的命令。



一个集群客户端通常会与集群中的多个节点创建套接字连接，而所谓的节点转向实际上就是换一个套接字发送命令。

如果客户端尚未与想要转向的节点创建套接字连接，那么客户端会先根据MOVED错误提供的IP地址和端口号来连接节点，然后再进行转向。

> ### 被隐藏的MOVED错误
>
> 集群模式的 redis-clie 客户端在接收到MOVED错误时，并不会打印出MOVED错误，而是根据MOVED错误自动进行节点转向，并打印出转向信息，所以我们时看不见节点返回的MOVED错误的：
>
> ```
> $ redis-cli -c -p 7000 # 集群模式
> 127.0.0.1:7000> set msg "happy new year!"
> -> Redirected to slot [6275] located at 127.0.0.1:7001
>
> 127.0.0.1:7001>
> ```
>
> 但是，如果我们使用单机（stand alone）模式d reds-cli客户端，再次向节点7000 发送相同的命令，那么MOVED 错误就会被客户端打印出来：
>
> ```
> $ redis-cli -p 7000 # 单机模式
> 127.0.0.1:7000> set msg "happy new year!"
> (error) MOVED 6257 127.0.0.1:7001
>
> 127.0.0.1:7000>
> ```
>
> 这是因为单机模式的redis-cli客户端不清楚 moved 错误的作用，所以只会直接转向，而不会自动转向

节点数据库的实现

集群节点保存键值对以及键值对过期时间的方式，与单机redis服务器保存键值对以及过期方式完全相同。

节点与单机服务器在数据库方面的一个区别是，节点只能使用0号数据库，而单机服务器没有这个限制。

另外，出了将键值对保存在数据库里之外，节点还会用clusterState 结构中的 slots_to_keys 跳跃表来保存槽和键之间的关系

```
typedef struct clusterState{
  // ...
  zskiplist *slots_to_keys;
  // ...
}clusterState;
```

slots_to_keys **跳跃表**每个节点的分值（score）都是一个槽号，而每个节点的成员（member）都是一个数据库键：

- 每当节点往数据库中添加一个新的键值对时，节点就会将这个键以及这个键所在的槽关联到slots_to_keys中
- 当节点删除数据库中的某个键值对时，节点就会在slots_to_keys 中解除关联

重新分片

Redis 集群的重新分片操作可以将任意数量已经指派给某个节点（源节点）的槽改为指派给另一个节点（目标节点），并且相关槽所属的键值对也会从源节点被移动到目标节点。

重新分片操作可以在线（online）操作，在重新分片的过程中，集群不需要下线，并且源节点和目标节点都可以继续处理命令请求。

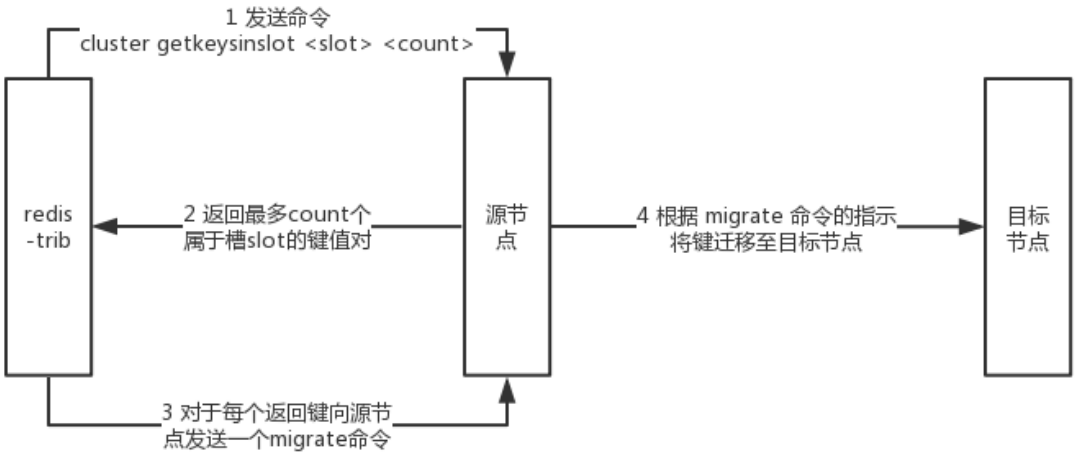
重新分片实现原理

redis集群的重新分片操作是由redis的集群管理软件redis-trib负责执行的，redis提供了进行重新分片所需的所有命令，而redis-trib则通过向源节点与目标节点发送命令来进行重新分片的操作

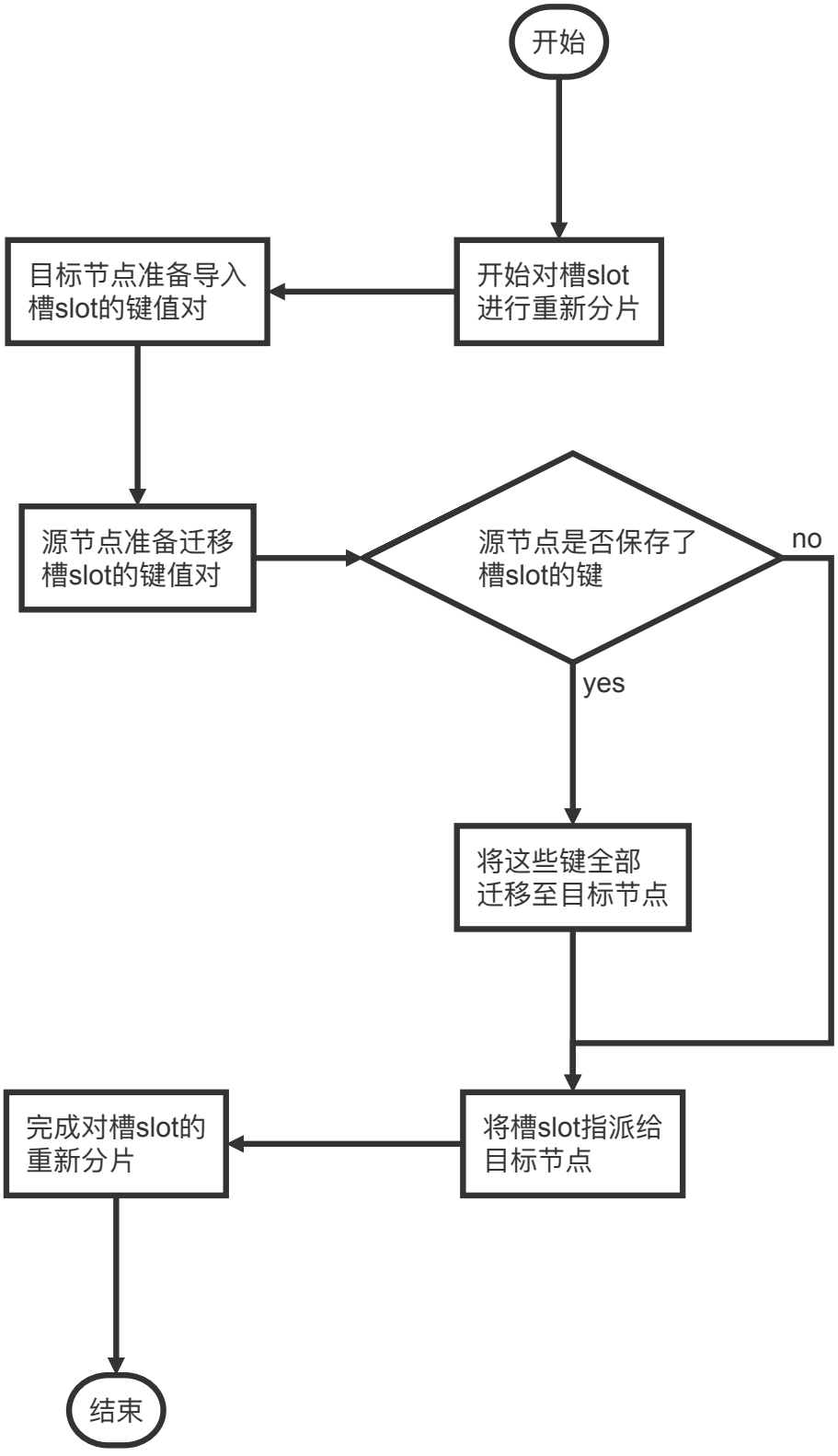
Redis-trib 对集群中单个槽slot进行重新分片的步骤如下：

1. Redis-trib 对目标节点发送 cluster setslot importing <source_id> 命令，让目标节点准备好从源节点倒入（import）属于槽 slot 的键值对。
2. redis-trib 对源节点发送 cluster setslot migrating <target_id> 命令，让源节点准备好将属于槽slot的键值对迁移（migrate）至目标节点。
3. redis-trib 向源节点发送 cluster getkeysinslot 命令，获得最多count个 属于 槽slot的键值对的键名（key name）。
4. 对于步骤3获得的每个键名，redis-trib都想源节点发送一个 migrate <target_id> <target_port> <key_name> 0 命令，将被选中的键值对从源节点迁移至目标节点
5. 重复执行步骤3与步骤4，知道源节点保存的所有属于槽slot的键值对都被迁移至目标节点为止。迁移过程如下图所示
6. redis-trib 向集群中的任意一个节点发送cluster setslot NODE <targe_id> 命令，将槽 slot 指派给

目标节点，这一指派信息会通过消息发送至整个集群，最终集群中的所有节点都会知道槽slot 已经指派给了目标节点。



如果重新分片涉及到了多个槽，那么redis-trib将对每个给定的槽分别执行上面给出的步骤。流程图如下：



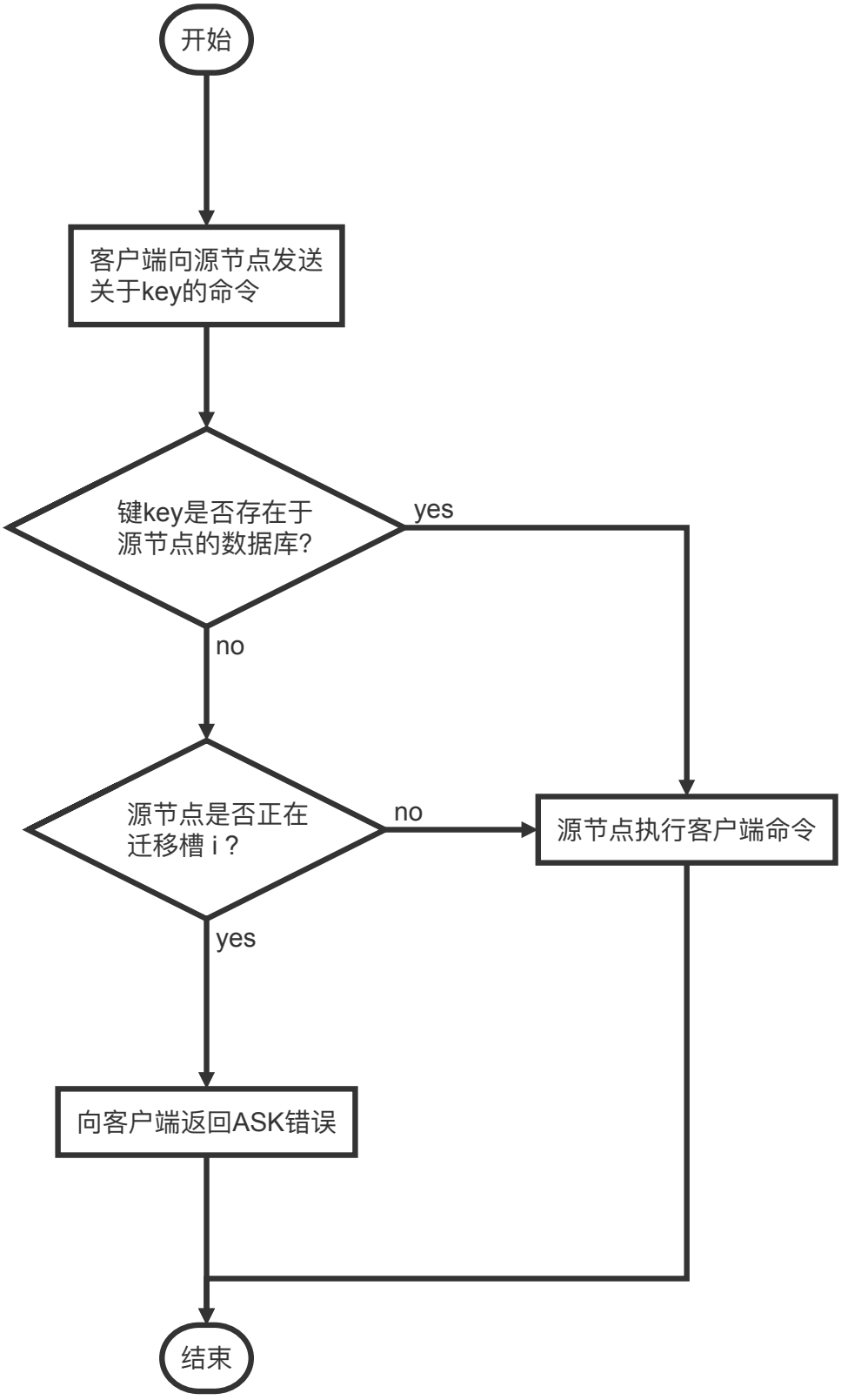
ASK错误

在进行重新分片期间，源节点向目标节点迁移一个槽的过程中，可能会出现这样一种情况：属于被迁移槽点一部分键值对保存在源节点中，而另一部分键值对保存在目标节点中。

当客户端向源节点发送一个与数据库键有关的命令，并且命令要处理的数据库键恰好就属于正在被迁移的槽时：

- 源节点会现在自己的数据库中查找制定的键，如果找到的话，就直接执行客户端发送的命令。
- 相反地，如果源节点没能在自己的数据库里面找到制定的键，那么这个键有可能已经被迁移到了目标节点，源节点将向客户端返回一个ASK错误，指引客户端转向正在被倒入槽点目标节点，并再次发送之前想要执行的命令。

如下图所示：



其中，源节点是否正在迁移槽 i ，若为no，则证明，该键不存在于源节点中；若为yes，则证明，该键可能存在于目标节点。

> **被隐藏的ASK错误**

> 和接到MOVED错误时的情况类似，集群模式的redis-cli在接到ASK错误也不会打印错误，而是自动根据错误提供的IP与PORT进行转向动作。如果想看到节点发送的ASK错误的话，可以使用单机模式的redis-cli客户端：
>
> ```
> $ redis-cli -p 7000
> 127.0.0.1:7000> get "love"
> (error)ASK 16198 127.0.0.1:7003
> ```

cluster setslot importing 命令的实现

clusterState 结构的 importing_slots_from 数组记录了当前节点正在从其他节点导入的槽：

```
typedef struct clusterState {
  // ...
  clusterNode *importing_slots_from[16384];
  // ...
}clusterState;
```

如果importing_slots_from[i] 的值不为null，而是志向一个clusterNode 结构，那么表示当前节点正在从clusterNode所代表的节点导入槽 i 。

在对集群进行重新分片的时候，向目标节点发送命令：

```
cluster setslot <i> importing <source_id>
```

可以讲目标节点 clusterState.importing_slots_from[i] 的值设置为 source_id 所代表的 clusterNode 结构。

cluster setslot migrating 命令的实现

clusterState 结构的 migrating_slots_to 数组记录了当前节点正在迁移至其他节点的槽：

```
typedef struct clusterState {
  // ...
  clusterNode *migrating_slots_to[16384];
  // ...
}clusterState ;
```

如果 migrating_slots_to[i] 的值不为null，而是指向一个 clusterNode的结构，那么表示当前节点正在将槽 i 迁移至clusterNode 所代表的节点。

在对集群进行重新分片的时候，向源节点发送命令：

```
cluster setslot <i> migrating <source_id>
```

可以将源节点 clusterState.migrating_slots_to[i] 的值设置为 target_id 所代表节点的 clusterNode 结构。

ASK错误

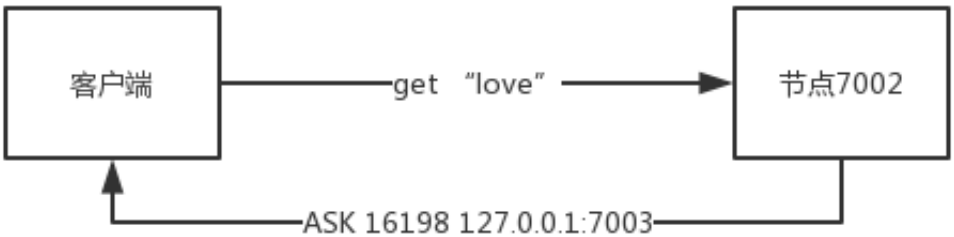
如果一个节点收到一个关于键key的命令请求，并且键key 所属于的槽 i 正好就指派给了这个节点，那么节点会尝试在自己的数据库里查找键key，如果找到了的话， 节点就直接执行客户端发送的命令。

于此相反，如果节点没有在自己的数据库中找到键key，那么节点会键擦好自己的clusterState.migrating_slots_to[i] ， 看键 key 所属的槽 i 是否正在进行迁移，如果槽 i 正在迁移的话，那么节点会想客户端发送一个ASK错误，引导客户端到正在导入 槽 i 的节点去查到键 key 。

例子如下：

> 假设节点7002 向节点 7003 迁移槽16198，有一个客户端向节点7002 发送命令：
>
> ```
> 127.0.01:7002> get "love"
> ```
>
> 因为键 love 刚好属于槽 16198 （所以不会发生MOVED错误），所以节点7002会首先在自己的数据库中查找键“love”，但并没有找到，通过检查自己的 clusterState.migrating_slots_to[16198] ，发现节点7002 正在将槽 16198 迁移至 节点7003 ， 于是 它向客户端返回错误：
>
> ```
> (error) ASK 16198 127.0.0.1:7003
> ```

这个错误表示客户端可以尝试到IP为127.0.0.1，端口号为7003 的节点去执行和槽16198 相关的操作，如下图示：



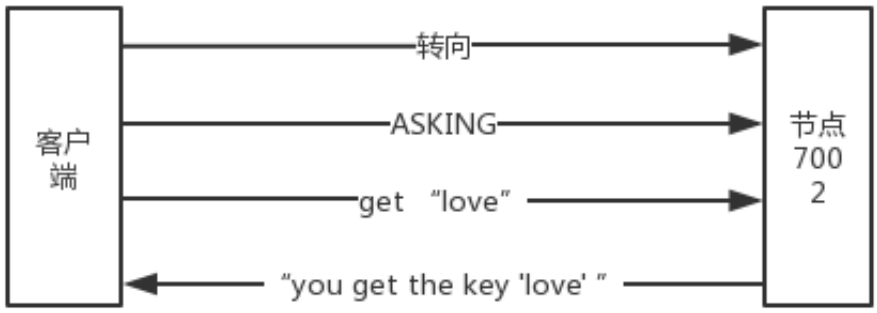
接收到ASK错误的客户端会根据错误提供的IP和PORT，转向至正在导入槽点目标节点，然后首先向目标节点发送一个ASKING命令，之后再重新发送原本想要执行的命令。

以上述例子来说，接收到 ASK 16198 127.0.0.1:7003 时，会执行以下步骤：

- 转向至7003节点
- 发送asking命令
- 发送 get “love” 命令

- 获得回复 "you get the key 'love' "

如下图示：

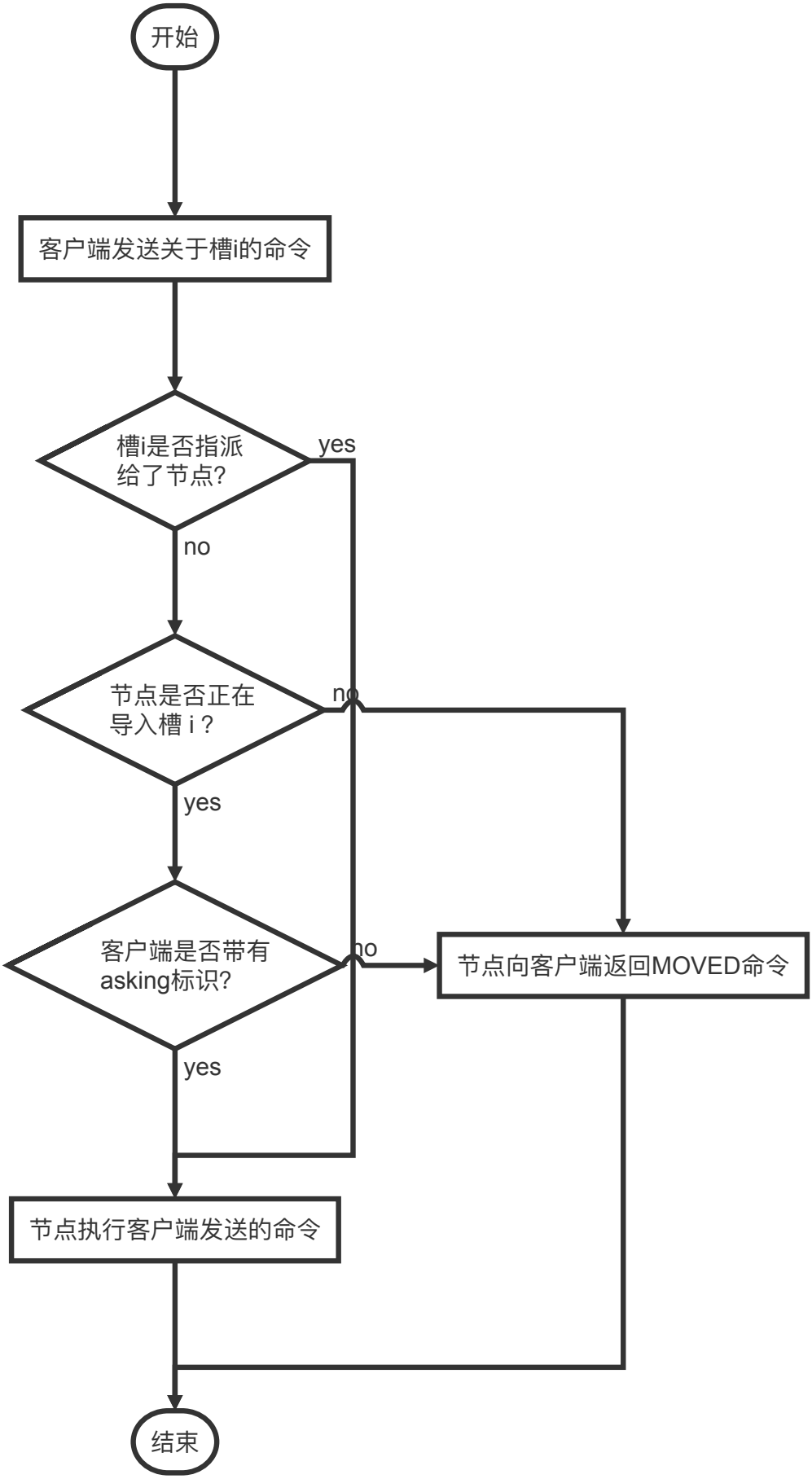


ASKING 命令

ASKING命令的唯一要做的就是打开发送该命令的客户端REDIS_ASKING标识，以下为伪码实现：

```
def asking();

// 打开标识
client.flags |= REDIS_ASKING

// 向客户端返回OK回复
reply("OK")
```

在一般的情况下，如果客户端向节点发送一个槽 i 的命令，而 槽 i 又没有指派给这个节点的话，那么节点将向客户端返回一个MOVED错误；但是，如果节点的clusterState.importing_slots_from[i] 显示节点正在导入槽 i ，并且发送命令的客户端带有 REDIS_ASKING 标识，那么节点将破例执行这个关于槽 i 的命令一次。流程如下：


开始
客户端发送关于槽i的命令
槽i是否指派给了节点?
yes
no
节点是否正在导入槽 i ?
no
yes
客户端是否带有asking标识?
no
节点向客户端返回MOVED命令
yes
节点执行客户端发送的命令
结束

当客户端接收到ask错误并转向至正才导入槽的节点时，客户端回先向节点发送一个asking命令，然后才重新发送想要执行的命令。如果不先发送asking命令的话，客户端发送的命令将被节点拒绝执行，并返回moved错误。

另外需要注意的是，客户端的REDIS_ASKING 标识是一个一次性标识，当节点执行了一个带有redis_asking表示的客户端发送的命令之后，客户端的redis_asking标识就会被移除。

ASK错误与MOVED错误的区别

ask错误与moved错误都会导致客户端转向，它们的区别在于：

- moved错误代表槽的负责权已经从一个节点转移到另一个节点：在客户端接收到关于槽i的moved错误之后，客户端每次遇到关于槽 i 的命令请求时，都可以直接将命令请求发送至moved错误所指向的节点。
- ask错误只是在两个节点在迁移过程中出现的临时解决方案：在客户端收到关于槽i的ask错误之后，客户端只会在接下来的一次命令请求中将关于槽i的命令请求发送至ask错误所指示的节点，但这种转向不会对客户端今后发送关于槽 i 对命令请求造成任何影响，客户端仍然会将关于槽 i 对命令请求发送至目前负责处理槽 i的节点，除非ask错误再次出现。

复制与故障转移

Redis集群中分为主节点（master）和从节点（slave），其中主节点用于处理槽，而从节点则用于复制某个主节点，并在复制的主节点下线时，代替下线主节点继续处理命令请求。

设置从节点

向一个节点发送命令：

```
clustaer replicate <node_id>
```

可以让接受命令的节点称为 node_id 所指定节点的从节点，并开始对主节点进行复制：

- 接收到该命令的节点首先会在自己的clusterState.nodes 字典中找到 node_id 所对应节点的clusterNode结构，一次来记录这个节点正在复制的主节点：

```
struct clusterNode {
  // ...
  // 如果这是一个从节点，那么指向主节点
  struct clusterNode *slaveof;
  // ...
}
```

- 然后节点会修改自己在clusterState.myself.flags 中的属性，关闭原本的 REDIS_NODE_MATER 标识，打开 REDIS_NODE_SLAVE 标识，表示这个节点已经从主节点变为了从节点
- 最后，节点会调用复制代码，并根据 clusterState.myself.slaveof 执行的 clusterNode 结构所保存的ip与port，对主节点进行复制。因为节点的复制功能与单机复制使用了相同的代码，所以让从节点复制主节点相当于向从节点发送命令 slaveof <master_ip> <master_port> 。

一个节点称为从节点，并开始复制某个主节点这一信息会通过消息发送给集群中的其他节点，最终集群中的所有节点都会知道某个从节点正在复制某个主节点。

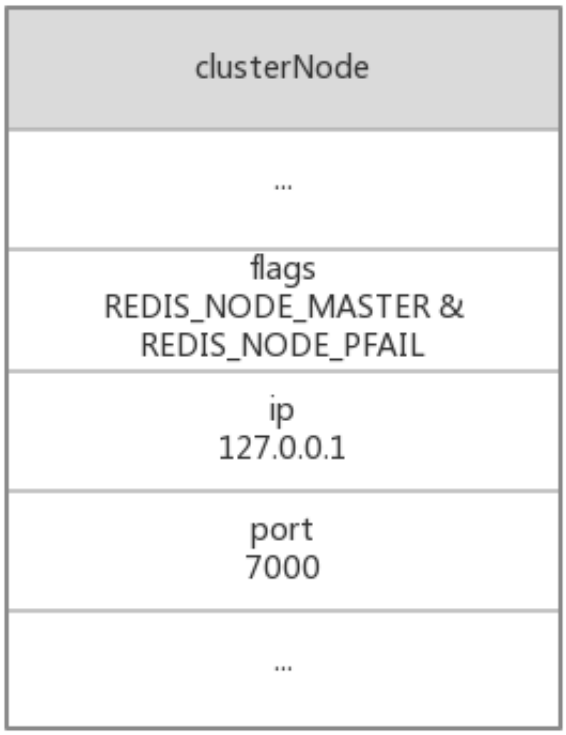
集群中所有节点都会在白哦是主节点的clusterNode结构的slaves属性和numslaves属性中记录正在复制这个主节点的从节点名单：

```
struct clusterNode {
  // ...
  // 复制当前节点的从节点数量
  int numslaves ;
  // 数组
  // 每个数组的项指向一个正在复制当前节点的从节点的clusterNode结构
  struct clusetNode **slaves ;
} ;
```

故障检测

集群中的每个节点都会定期地向集群中的其他节点发送ping消息，以此来检测对方是否在线，如果接收ping消息的节点没有在规定的时间内，向发送ping消息的节点返回pong消息，那么发送ping消息的节点就会将接收ping消息的节点标记为疑似下线（probable fail，PFAIL）。

举例如下：如果节点7001向节点7000发送了一条ping消息，但是节点7000没有在规定的时间内向节点7001返回一条pong消息，那么节点7001就会在自己的clusterState.nodes 字典中找到节点7000所对应的clusterNode结构，并在结构的flags属性中打开 REDIS_NODE_PFALL 标识，以此表示节点7000进入了疑似下线的状态，如下图所示：



集群中各个节点会通过互相发送消息的方式来疾患集群中各个节点的状态信息。例如某个节点是处于在线状态，疑似下线状态（PFAIL）还是已下线状态（FAIL）。

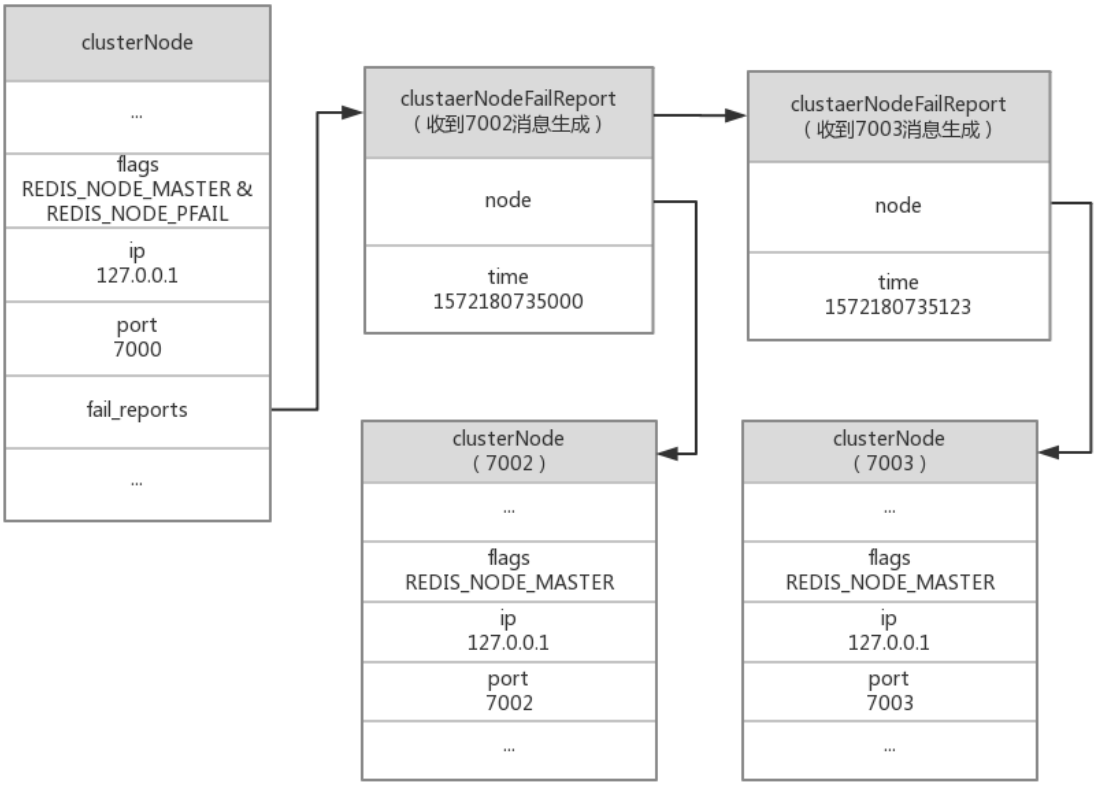
当一个主节点A通过消息得知主节点B认为主节点C进入了疑似下线状态时，主节点A会在自己的clusterState.node字典中找到主节点C对应的clusterNode结构，并将主节点B的下线报告（failure report）添加到clusterNode结构的fail_reports链表里面：

```
struct clusterNode{
  // ...
  // 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;
  // ...
};
```

每个下线报告都是由一个 clusterNodeFailReport 结构表示：

```
struct clusterNodeFailReport {
  // 报告目标节点 已经下线的的节点
  struct clusterNode *node;
  // 最后一次从node节点收到下线报告的时间
  // 程序使用这个时间戳来检查下线报告是否过期
  //  （与当前时间相差太大的下线报告会被删除）
  mstime_t time;
}typedef clusterNodeFailReport;
```

举个例子，如果主节点7001在收到主节点7002和7003发送的消息后得知，主节点7002和7003都认为主节点7000进入了疑似下线阶段，那么主节点7001将为主节点7000创建如下图的下线报告：



如果在一个集群里，半数以上负责处理槽的主节点都将某个主节点x报告为疑似下线，那么这个主节点x将被标记为已下线（FAIL），将主节点x标记为已下线的节点会向集群广播一条关于主节点x的FAIL消息，所有收到这条FAIL消息的节点都会立即将主节点x标记为已下线。

这里有一个疑问！因为通知"疑似下线"状态的消息是广播的，所以存在一定的可能性，主节点A与主节点B同时得出结论：主节点C 变成 已下行线状态，此时主节点A与B同时向集群广播 C 已下线的消息吗?

故障转移

当一个节点发现自己正在复制的主节点进入了一下线状态时，从节点将开始对下线主节点进行故障转移，以下是故障转移步骤：

1. 复制下线主节点的所有从节点里面，会有一个节点会被选中。
2. 被选中的从节点会执行slaveof no one 命令，成为新的主节点。
3. 新的主节点会撤销所有对已下线主节点的槽指派，并将这些槽全部指派给自己。
4. 新的主节点向集群中广播一条pong消息，这条pong消息可以让集群中的其他节点立即知道这个节点已经由从节点变成了主节点，并且这个主节点已经接管了原本由已下线节点负责处理的槽。
5. 新的主节点开始接收自己负责处理的槽有关的命令请求，故障转移完成。
6. 新问题，难道这里不需要，其他从节点，slaves of 新的主节点吗?

选取新的主节点

新的主节点是通过选举产生的。

以下是集群选举新的主节点的方法：

1. 集群的配置机缘是一个字增计数器，它的初始值为0。
2. 当集群中的某个节点开始一次故障转移操作时，集群配置纪元的值都会被 +1 。
3. 对于每个配置纪元，集群里每个负责处理槽的主节点都有一次投票的机会，而第一个向出节点要求投票的从节点将获得主节点的投票。
4. 当从节点发现自己正在复制的主节点进入已下线状态时，从节点会向集群广播一条 clustermsg_type_failover_auth_request 消息，要求所有收到这条消息，并且具有投票权的主节点向这个从节点投票。
5. 如果一个主节点具有投票权，并且这个主节点尚未投票给其他从节点，那么主节点将向要求投票的从节点返回一条 clustermsg_type_failover_auth_ack 消息，表示这个主节点支持这个从节点成为新的主节点
6. 每个参与选举的从节点都会接收 clustermsg_type_failover_auth_ack 消息，并根据自己收到了多少条这种消息来统计自己获得了多少主节点的支持。
7. 如果集群里有个N个具有投票权利的主节点，那么当一个从节点收集到大于等于N/2+1张支持票时，这个从节点就会当选为新的主节点。
8. 因为在每一个配置纪元里面，每个具有投票权的主节点只能投一次票，所以如果有N个主节点进行投票，那么具有大于等于 N/2+1 张支持票的从节点只会有一个，这个确保了新的主节点 只会有一个。
9. 如果在一个配置纪元中没有从节点能收集到足够多的支持票，那么集群将进入一个新的配置纪元，并在此进行选举，知道选出新的主节点为止。

这个选举新主节点的方法和sentinel方法非常相似，因为两者都是基于Raft算法的领头选举（leader election）方法实现的。

消息

集群中的各个节点通过发送和接收消息（message）来进行通信，我们称发送消息的节点为发送着（sender），接收消息的节点为接收者（reveiver）。

节点发送的消息主要有以下5种：

- meet消息：当发送者接到客户端发送的cluster meet命令时，发送者会向接受者发送meet消息，请求接收者加入到发送者当前所处的集群中。
- ping消息：集群里的每个节点默认每隔一秒钟就会从已知节点列表中随机选出5个节点，然后对这5个节点中最长时间没有发送过ping消息的节点发送ping消息，以此来检测被选中节点是否在线。除此之外，如果节点A最后一次收到节点B发送的pong消息的时间，举例当前时间已经超过了cluster_node_timeout 选项设置时长的一半，那么节点A也会向节点B发送PING消息，这可以防止节点A因为长时间没有随机选中节点B作为ping消息的发送对象而导致对节点B的信息更新滞后。
- pong消息：当接收者收到发送者发来的meet消息或者ping消息时，为了向发送者确认这条meet消息或者ping消息已送达，接收者会向发送者返回一条pong消息。另外，一个节点也可以通过向集群广播自己的pong消息来让集群中的其他节点立即刷新关于这个节点的认识，例如当一次故障转移操作成功执行之后，新的主节点会向集群广播一条pong消息，以此来让集群中的其他节点立即知道这个节点已经变成了主节点，并且接管了已下线节点负责的槽。
- fail消息：当一个主节点A判断另一个主节点B已经进入了FAIL状态时，节点A会向集群广播一条关于节点B的FAIL消息，所有收到这条消息的节点都会立即将B标记为已下线。
- publish消息，当节点收到一个publish命令时，节点会执行这个命令，并向集群广播一条publish消息，所有接收到这条publish消息的节点都会执行相同的publish命令

一条消息分别由消息头（header）和消息正文（data）组成，接下来的内容将首先介绍消息头，然后再分别介绍上面提到的物种不同类型的消息正文。

消息头

节点发送的所有消息都是由一个消息头报告，消息头除了包含消息正文之外，还记录了消息发送者自身的一些信息，因为这些信息也会被消息接收者用到，所以严格来讲，我们还可以认为消息头本身也是消息的一部分。

结构如下：

```
typedef struct {
    char sig[4];        /* Signature "RCmb" (Redis Cluster message bus). */
    uint32_t totlen;    /* 消息的长度 */
    uint16_t ver;       /* Protocol version, currently set to 1. */
    uint16_t port;      /* 端口号 */
    uint16_t type;      /* 消息类型 */
    uint16_t count;     /* 消息正文包含的节点信息数量，只有meet，ping，pong三种有用 */
    uint64_t currentEpoch;  /* 发送者所处的配置纪元 */
    uint64_t configEpoch;   /* 如果是主节点，则为发送者配置纪元；从节点，发送者正在复制
的主节点配置纪元*/
    uint64_t offset;    /* Master replication offset if node is a master or
                           processed replication offset if node is a slave. */
    char sender[CLUSTER_NAMELEN]; /* 发送者名字 ID */
    unsigned char myslots[CLUSTER_SLOTS/8]; /* 发送者目前的槽指派信息 */
    char slaveof[CLUSTER_NAMELEN]; /* 从节点的话，这里记录主节点；若主节点，则为
REDIS_NODE_NULL_NAME */
    char myip[NET_IP_STR_LEN];    /* Sender IP, if not all zeroed. */
    char notused1[34];  /* 34 bytes reserved for future usage. */
    uint16_t cport;      /* Sender TCP cluster bus port */
    uint16_t flags;      /* 发送者标识值 */
    unsigned char state; /* 发送者所处集群状态 */
    unsigned char mflags[3]; /* Message flags: CLUSTERMSG_FLAG[012]_... */
    union clusterMsgData data; /* 消息正文 */
} clusterMsg;
```

clusterMsg.data 属性指向联合 cluster.h/clusterMsgData ，这个联合就是消息的正文：

```
union clusterMsgData {
    /* PING, MEET and PONG */
    struct {
        /* Array of N clusterMsgDataGossip structures */
        clusterMsgDataGossip gossip[1];
    } ping;

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

    /* PUBLISH */
    struct {
```

```
        clusterMsgDataPublish msg;
    } publish;

    /* UPDATE */
    struct {
        clusterMsgDataUpdate nodecfg;
    } update;

    /* MODULE */
    struct {
        clusterMsgModule msg;
    } module;
};
```

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。

举例，通过对比接收者为发送者记录的槽指派信息，以及发送者在消息头的myslots属性记录的槽指派信息，接收者可以知道发送者的槽指派信息是否发生了变化。

又或者说，通过对比接收者为发送者记录的标识值，以及发送者在消息头的flags属性记录的标识值，接收者可以知道发送者的状态和角色是否发生了变化，例如节点状态由原来的在线变成了下线，或者由主节点变成从节点等。

meet、ping、pong消息的实现

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 cluster.h/clusterMsgDataGossip 结构组成：

```
union clusterMsgData {
    /* PING, MEET and PONG */
    struct {
        /* Array of N clusterMsgDataGossip structures */
        clusterMsgDataGossip gossip[1];
    } ping;

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

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

    /* UPDATE */
```

```
    struct {
        clusterMsgDataUpdate nodecfg;
    } update;

    /* MODULE */
    struct {
        clusterMsgModule msg;
    } module;
};
```

因为meet、ping、pong 三种消息都是用相同的消息正文，所遇你节点通过下消息头的type 属性来判断一条消息时meet消息，ping消息还是pong消息。

每次发送meet、ping、pong消息时，发送者都从自己已知的节点列表中随机选出两个节点（可以是主节点或者是从节点），并将这两个选中节点的信息分别保存到两个clusterMsgDataGossip 结构里面。

clusterMsgDataGossip结构记录了被选中节点的名字，发送者与被选中节点最后一次发送和接收ping消息和pong消息的时间戳，被选中节点ip地址和port，以及被选中节点的标识值：

```
typedef struct {
    // 节点的名字
    char nodename[CLUSTER_NAMELEN];
    // 最后一次向该节点发送ping消息的时间戳
    uint32_t ping_sent;
    // 最后一次从该节点接收到pong消息的时间戳
    uint32_t pong_received;
    // 节点ip地址 NET_IP_STR_LEN == 16
    char ip[NET_IP_STR_LEN];
    // 节点端口号
    uint16_t port;
    uint16_t cport;
    // 节点标识值
    uint16_t flags;
    uint32_t notused1;
} clusterMsgDataGossip;
```

当接收者收到meet、ping、pong消息时，接收者会访问消息正文中的两个clusterMsgDataGossip结构，并根据自己是否认识clusterMsgDataGossip结构中记录的被选中节点来选择进行哪种操作：

- 如果被选中节点不存在于接收者已知节点列表，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和port等信息，与被选中节点进行握手
- 如果被选中节点已经存在于接收者已知节点列表，那么说明接收者之前已经于被选中节点进行接触，接收者将根据clusterMsgDataGossip结构记录的信息，对被选中节点所对应的clusterNode结构进行更新。

例如，发送ping于pong消息的例子，假设在一个包含A、B、C、D、E、F六个节点的集群里：

- 节点A向节点D发送ping消息，并且消息里面包含了节点B和节点C的信息，当节点D收到这条PING消息时，它将更新自己对节点B和节点C的认识。
- 之后，节点D将向节点A返回一条pong消息，并且消息里包含了节点E和节点F的信息，当节点A接

收到这条pong消息时，它将更新自己对节点E和节点F的认识。

fail消息的实现

当集群里的主节点A将主节点B标记为已下线（FAIL）时，主节点A将向集群广播一条关于主节点B的FAIL消息，所有接收到这条FAIL消息的节点都会将主节点B标记为已下线。

在集群的节点数量比较大时，淡出你使用Gossip协议来传播节点的已下线信息会给节点信息更新带来一定的延迟，因为Gossip协议消息通常需要一段时间才能传播至整个集群，而发送FAIL消息可以让集群里的所有节点立即知道某个主节点已下线，从而尽快判断是否需要将集群标记为下线，又或者对下线主节点进行故障转移。

> 是什么协议？如何立即？

FAIL消息的正文由cluster.h/clusterMsgDataFail结构表示，这个结构只包含一个nodename属性，该属性记录了已下线节点的名字：

```
typedef struct {
    char nodename[CLUSTER_NAMELEN];
} clusterMsgDataFail;
```

因为集群中的所有节点都有一个独一无二的名字，所以FAIL消息里面只需要保存下线节点的名字，接收到消息的节点就可以根据这个名字来判断是哪个节点下线了。

publish消息的实现

当客户端向集群中的某个节点发送命令：

publish

的时候，接收到publish命令的节点不仅会向channel频道发送消息message，它还会向集群广播一条publish消息，所有接收到这条publish消息的节点都会向channel频道发送message消息。

换句话说，向集群中的某个节点发送命令：

publish

将导致所有节点都向channel频道发送mesage消息。

消息正文由 cluster.h/clusterMsgDataPublish 结构表示：

```
typedef struct {
    // channel 的长度
    uint32_t channel_len;
    // mesg的长度
    uint32_t message_len;
    // 定义为8字节，目的是为了对齐其他消息结构
    // 实际长度由保存内容决定
    unsigned char bulk_data[8];
} clusterMsgDataPublish;
```

其中 0 ～ channel_len-1 是channel参数，channel_len ~ message_len 是mesg参数。

> **为什么不直接向节点广播PUBLISH命令**
>
> 实际上，要让集群的所有节点都执行相同的PUBLISH 命令，最简单的方法就是向所有节点广播相同的publish 命令，这也是redis 在复制publish 命令时所使用的方法，不过因为这种做法并不符合redis集群的“各个节点通过发送和接收消息来进行通信” 这一规则，所以节点没有采取广播PUBLISH命令的做法。

回顾

- 节点通过握手来将其他节点添加到自己所处的集群当中
- 集群中的16384个槽可以分别指派给集群中的各个节点，每个节点都会记录哪些槽指派给了自己，而哪些槽又被指派给了其他节点
- 节点在接到一个命令请求时，会先检查这个命令请求要处理的槽是否由自己负责，如果不是的话，节点将向客户端返回一个moved错误，moved错误携带的信息可以指引客户端专向至正在负责相关槽点节点
- 对Redis集群的重新分片工作是由redis-trib负责执行的，重新分片的关键是将属于某个槽所有的键值对从一个节点转至另一个节点。
- 如果节点A正在迁移槽i至节点B，那么当节点A没能在自己的数据库中找到命令指定的数据库键时，节点A会向客户端放回一个ASK错误，指引客户端到节点B继续查找指定的数据库键
- MOVED错误表示槽点负责权已经从一个节点转移到另一个节点，而ASK错误只是两个节点在迁移槽的过程中使用的一种临时措施
- 集群里的从节点用于复制主节点，并在主节点下线时，代替主节点继续处理命令请求
- 集群中的节点通过发送和接收消息来进行通信，常见的消息包括MEET、PING、PONG、PUBLISH、FALL五种