**一：主从复制**

在集群中，为了保证集群的健壮性，通常设置一部分集群节点为主节点，另一部分集群节点为这些主节点的从节点。一般情况下， 需要保证每个主节点至少有一个从节点。

集群初始化时，每个集群节点都是以独立的主节点角色而存在的，通过向集群节点发送”CLUSTER MEET <ip> <port>”命令，可以使集群节点间相互认识。节点间相互认识之后，可以通过向某些集群节点发送"CLUSTER REPLICATE <nodeID>"命令，使收到命令的集群节点成为<nodeID>节点的从节点。

在函数clusterCommand中，处理这部分的代码如下：

**else** **if** **(!**strcasecmp**(**c**->**argv**[**1**]->**ptr**,**"replicate"**)** **&&** c**->**argc **==** 3**)** **{**

*/\* CLUSTER REPLICATE <NODE ID> \*/*

clusterNode **\***n **=** clusterLookupNode**(**c**->**argv**[**2**]->**ptr**);**

*/\* Lookup the specified node in our table. \*/*

**if** **(!**n**)** **{**

addReplyErrorFormat**(**c**,**"Unknown node %s"**,** **(**char**\*)**c**->**argv**[**2**]->**ptr**);**

**return;**

**}**

*/\* I can't replicate myself. \*/*

**if** **(**n **==** myself**)** **{**

addReplyError**(**c**,**"Can't replicate myself"**);**

**return;**

**}**

*/\* Can't replicate a slave. \*/*

**if** **(**nodeIsSlave**(**n**))** **{**

addReplyError**(**c**,**"I can only replicate a master, not a slave."**);**

**return;**

**}**

*/\* If the instance is currently a master, it should have no assigned*

*\* slots nor keys to accept to replicate some other node.*

*\* Slaves can switch to another master without issues. \*/*

**if** **(**nodeIsMaster**(**myself**)** **&&**

**(**myself**->**numslots **!=** 0 **||** dictSize**(**server**.**db**[**0**].**dict**)** **!=** 0**))** **{**

addReplyError**(**c**,**

"To set a master the node must be empty and "

"without assigned slots."**);**

**return;**

**}**

*/\* Set the master. \*/*

clusterSetMaster**(**n**);**

clusterDoBeforeSleep**(**CLUSTER\_TODO\_UPDATE\_STATE**|**CLUSTER\_TODO\_SAVE\_CONFIG**);**

addReply**(**c**,**shared**.**ok**);**

**}**

"CLUSTER REPLICATE"命令的格式是"CLUSTER REPLICATE <nodeID>"；

首先，根据命令参数<nodeID>，从字典server.cluster->nodes中寻找对应的节点n；如果找不到n，或者，如果n就是当前节点，或者，n节点是个从节点，则回复客户端错误信息后返回；

如果当前节点为主节点，则当前节点不能有负责的槽位，当前节点的数据库也必须为空，如果不满足以上任一条件，则将不能置当前节点为从节点，因此回复客户端错误信息后，直接返回；

接下来，调用clusterSetMaster函数置当前节点为n节点的从节点，最后，回复客户端"OK"；

clusterSetMaster函数的代码如下：

void clusterSetMaster**(**clusterNode **\***n**)** **{**

redisAssert**(**n **!=** myself**);**

redisAssert**(**myself**->**numslots **==** 0**);**

**if** **(**nodeIsMaster**(**myself**))** **{**

myself**->**flags **&=** **~**REDIS\_NODE\_MASTER**;**

myself**->**flags **|=** REDIS\_NODE\_SLAVE**;**

clusterCloseAllSlots**();**

**}** **else** **{**

**if** **(**myself**->**slaveof**)**

clusterNodeRemoveSlave**(**myself**->**slaveof**,**myself**);**

**}**

myself**->**slaveof **=** n**;**

clusterNodeAddSlave**(**n**,**myself**);**

replicationSetMaster**(**n**->**ip**,** n**->**port**);**

resetManualFailover**();**

**}**

首先，必须保证n不是当前节点，而且当前节点没有负责任何槽位；

如果当前节点已经是主节点了，则将节点标志位中的REDIS\_NODE\_MASTER标记清除，并增加REDIS\_NODE\_SLAVE标记；然后调用clusterCloseAllSlots函数，置server.cluster->migrating\_slots\_to和server.cluster->importing\_slots\_from为空；

如果当前节点为从节点，并且目前已有主节点，则调用clusterNodeRemoveSlave函数，将当前节点从其当前主节点的slaves数组中删除，解除当前节点与其当前主节点的关系；

然后，置myself->slaveof为n，调用clusterNodeAddSlave函数，将当前节点插入到n->slaves中；

然后，调用replicationSetMaster函数，这里直接复用了主从复制部分的代码，相当于向当前节点发送了"SLAVE OF"命令，开始主从复制流程；

最后，调用resetManualFailover函数，清除手动故障转移状态；

**二：故障转移**

1：纪元(epoch)

理解Redis集群中的故障转移，必须要理解纪元(epoch)在分布式Redis集群中的作用，Redis集群使用RAFT算法中类似term的概念，在Redis集群中这被称之为纪元(epoch)。纪元的概念在介绍哨兵时已经介绍过了，在Redis集群中，纪元的概念和作用与哨兵中的纪元类似。Redis集群中的纪元主要是两种：currentEpoch和configEpoch。

a、currentEpoch

这是一个集群状态相关的概念，可以当做记录集群状态变更的递增版本号。每个集群节点，都会通过server.cluster->currentEpoch记录当前的currentEpoch。

集群节点创建时，不管是主节点还是从节点，都置currentEpoch为0。当前节点接收到来自其他节点的包时，如果发送者的currentEpoch（消息头部会包含发送者的currentEpoch）大于当前节点的currentEpoch，那么当前节点会更新currentEpoch为发送者的currentEpoch。因此，集群中所有节点的currentEpoch最终会达成一致，相当于对集群状态的认知达成了一致。

currentEpoch作用在于，当集群的状态发生改变，某个节点为了执行一些动作需要寻求其他节点的同意时，就会增加currentEpoch的值。目前currentEpoch只用于从节点的故障转移流程，这就跟哨兵中的sentinel.current\_epoch作用是一模一样的。

当从节点A发现其所属的主节点下线时，就会试图发起故障转移流程。首先就是增加currentEpoch的值，这个增加后的currentEpoch是所有集群节点中最大的。然后从节点A向所有节点发包用于拉票，请求其他主节点投票给自己，使自己能成为新的主节点。

其他节点收到包后，发现发送者的currentEpoch比自己的currentEpoch大，就会更新自己的currentEpoch，并在尚未投票的情况下，投票给从节点A，表示同意使其成为新的主节点。

b、configepoch

这是一个集群节点配置相关的概念，每个集群节点都有自己独一无二的configepoch。所谓的节点配置，实际上是指节点所负责的槽位信息。

每一个主节点在向其他节点发送包时，都会附带其configEpoch信息，以及一份表示它所负责槽位的位数组信息。而从节点向其他节点发送包时，包中的configEpoch和负责槽位信息，是其主节点的configEpoch和负责槽位信息。节点收到包之后，就会根据包中的configEpoch和负责槽位信息，记录到相应节点属性中。

configEpoch主要用于解决不同的节点的配置发生冲突的情况。举个例子就明白了：节点A宣称负责槽位1，其向外发送的包中，包含了自己的configEpoch和负责槽位信息。节点C收到A发来的包后，发现自己当前没有记录槽位1的负责节点（也就是server.cluster->slots[1]为NULL），就会将A置为槽位1的负责节点（server.cluster->slots[1] = A），并记录节点A的configEpoch。后来，节点C又收到了B发来的包，它也宣称负责槽位1，此时，如何判断槽位1到底由谁负责呢？这就是configEpoch起作用的时候了，C在B发来的包中，发现它的configEpoch，要比A的大，说明B是更新的配置，因此，就将槽位1的负责节点设置为B（server.cluster->slots[1] = B）。

在从节点发起选举，获得足够多的选票之后，成功当选时，也就是从节点试图替代其下线主节点，成为新的主节点时，会增加它自己的configEpoch，使其成为当前所有集群节点的configEpoch中的最大值。这样，该从节点成为主节点后，就会向所有节点发送广播包，强制其他节点更新相关槽位的负责节点为自己。

2：故障转移概述

集群中，当某个从节点发现其主节点下线时，就会尝试在未来某个时间点发起故障转移流程。具体而言就是先向其他集群节点发送CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_REQUEST包用于拉票，集群主节点收到这样的包后，如果在当前选举纪元中没有投过票，就会向该从节点发送CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_ACK包，表示投票给该从节点。从节点如果在一段时间内收到了大部分主节点的投票，则表示选举成功，接下来就是升级为主节点，并接管原主节点所负责的槽位，并将这种变化广播给其他所有集群节点，使它们感知这种变化，并修改自己记录的配置信息。

接下来，就是故障转移中各个环节的详细描述：

3：从节点的选举和升级

3.1、从节点发起故障转移的时间

从节点在发现其主节点下线时，并不是立即发起故障转移流程，而是要等待一段时间，在未来的某个时间点才发起选举。这个时间点是这样计算的：

mstime() + 500ms + random()%500ms + rank\*1000ms

其中，固定延时500ms，是为了留出时间，使主节点下线的消息能传播到集群中其他节点，这样集群中的主节点才有可能投票；随机延时是为了避免两个从节点同时开始故障转移流程；rank表示从节点的排名，排名是指当前从节点在下线主节点的所有从节点中的排名，排名主要是根据复制数据量来定，复制数据量越多，排名越靠前，因此，具有较多复制数据量的从节点可以更早发起故障转移流程，从而更可能成为新的主节点。

rank主要是通过调用clusterGetSlaveRank得到的，该函数的代码如下：

int clusterGetSlaveRank**(**void**)** **{**

long long myoffset**;**

int j**,** rank **=** 0**;**

clusterNode **\***master**;**

redisAssert**(**nodeIsSlave**(**myself**));**

master **=** myself**->**slaveof**;**

**if** **(**master **==** **NULL)** **return** 0**;** */\* Never called by slaves without master. \*/*

myoffset **=** replicationGetSlaveOffset**();**

**for** **(**j **=** 0**;** j **<** master**->**numslaves**;** j**++)**

**if** **(**master**->**slaves**[**j**]** **!=** myself **&&**

master**->**slaves**[**j**]->**repl\_offset **>** myoffset**)** rank**++;**

**return** rank**;**

**}**

在该函数中，首先得到当前从节点的主节点master，如果master为NULL，则直接返回0；

然后调用replicationGetSlaveOffset函数，得到当前从节点的复制偏移量myoffset；接下来轮训master->slaves数组，只要其中从节点的复制偏移量大于myoffset，则增加排名rank的值；

在没有开始故障转移之前，每隔一段时间就会调用一次clusterGetSlaveRank函数，以更新当前从节点的排名。

3.2、从节点发起故障转移，开始拉票

从节点的故障转移，是在函数clusterHandleSlaveFailover中处理的，该函数在集群定时器函数clusterCron中调用。本函数用于处理从节点进行故障转移的整个流程，包括：判断是否可以发起选举；发起选举；判断选举是否超时；判断自己是否拉到了足够的选票；使自己升级为新的主节点这些所有流程。首先看一下升级流程之前的代码，如下：

void clusterHandleSlaveFailover**(**void**)** **{**

mstime\_t data\_age**;**

mstime\_t auth\_age **=** mstime**()** **-** server**.**cluster**->**failover\_auth\_time**;**

int needed\_quorum **=** **(**server**.**cluster**->**size **/** 2**)** **+** 1**;**

int manual\_failover **=** server**.**cluster**->**mf\_end **!=** 0 **&&**

server**.**cluster**->**mf\_can\_start**;**

mstime\_t auth\_timeout**,** auth\_retry\_time**;**

server**.**cluster**->**todo\_before\_sleep **&=** **~**CLUSTER\_TODO\_HANDLE\_FAILOVER**;**

*/\* Compute the failover timeout (the max time we have to send votes*

*\* and wait for replies), and the failover retry time (the time to wait*

*\* before trying to get voted again).*

*\**

*\* Timeout is MIN(NODE\_TIMEOUT\*2,2000) milliseconds.*

*\* Retry is two times the Timeout.*

*\*/*

auth\_timeout **=** server**.**cluster\_node\_timeout**\***2**;**

**if** **(**auth\_timeout **<** 2000**)** auth\_timeout **=** 2000**;**

auth\_retry\_time **=** auth\_timeout**\***2**;**

*/\* Pre conditions to run the function, that must be met both in case*

*\* of an automatic or manual failover:*

*\* 1) We are a slave.*

*\* 2) Our master is flagged as FAIL, or this is a manual failover.*

*\* 3) It is serving slots. \*/*

**if** **(**nodeIsMaster**(**myself**)** **||**

myself**->**slaveof **==** **NULL** **||**

**(!**nodeFailed**(**myself**->**slaveof**)** **&&** **!**manual\_failover**)** **||**

myself**->**slaveof**->**numslots **==** 0**)**

**{**

*/\* There are no reasons to failover, so we set the reason why we*

*\* are returning without failing over to NONE. \*/*

server**.**cluster**->**cant\_failover\_reason **=** REDIS\_CLUSTER\_CANT\_FAILOVER\_NONE**;**

**return;**

**}**

*/\* Set data\_age to the number of seconds we are disconnected from*

*\* the master. \*/*

**if** **(**server**.**repl\_state **==** REDIS\_REPL\_CONNECTED**)** **{**

data\_age **=** **(**mstime\_t**)(**server**.**unixtime **-** server**.**master**->**lastinteraction**)**

**\*** 1000**;**

**}** **else** **{**

data\_age **=** **(**mstime\_t**)(**server**.**unixtime **-** server**.**repl\_down\_since**)** **\*** 1000**;**

**}**

*/\* Remove the node timeout from the data age as it is fine that we are*

*\* disconnected from our master at least for the time it was down to be*

*\* flagged as FAIL, that's the baseline. \*/*

**if** **(**data\_age **>** server**.**cluster\_node\_timeout**)**

data\_age **-=** server**.**cluster\_node\_timeout**;**

*/\* Check if our data is recent enough according to the slave validity*

*\* factor configured by the user.*

*\**

*\* Check bypassed for manual failovers. \*/*

**if** **(**server**.**cluster\_slave\_validity\_factor **&&**

data\_age **>**

**(((**mstime\_t**)**server**.**repl\_ping\_slave\_period **\*** 1000**)** **+**

**(**server**.**cluster\_node\_timeout **\*** server**.**cluster\_slave\_validity\_factor**)))**

**{**

**if** **(!**manual\_failover**)** **{**

clusterLogCantFailover**(**REDIS\_CLUSTER\_CANT\_FAILOVER\_DATA\_AGE**);**

**return;**

**}**

**}**

*/\* If the previous failover attempt timedout and the retry time has*

*\* elapsed, we can setup a new one. \*/*

**if** **(**auth\_age **>** auth\_retry\_time**)** **{**

server**.**cluster**->**failover\_auth\_time **=** mstime**()** **+**

500 **+** */\* Fixed delay of 500 milliseconds, let FAIL msg propagate. \*/*

random**()** **%** 500**;** */\* Random delay between 0 and 500 milliseconds. \*/*

server**.**cluster**->**failover\_auth\_count **=** 0**;**

server**.**cluster**->**failover\_auth\_sent **=** 0**;**

server**.**cluster**->**failover\_auth\_rank **=** clusterGetSlaveRank**();**

*/\* We add another delay that is proportional to the slave rank.*

*\* Specifically 1 second \* rank. This way slaves that have a probably*

*\* less updated replication offset, are penalized. \*/*

server**.**cluster**->**failover\_auth\_time **+=**

server**.**cluster**->**failover\_auth\_rank **\*** 1000**;**

*/\* However if this is a manual failover, no delay is needed. \*/*

**if** **(**server**.**cluster**->**mf\_end**)** **{**

server**.**cluster**->**failover\_auth\_time **=** mstime**();**

server**.**cluster**->**failover\_auth\_rank **=** 0**;**

**}**

redisLog**(**REDIS\_WARNING**,**

"Start of election delayed for %lld milliseconds "

"(rank #%d, offset %lld)."**,**

server**.**cluster**->**failover\_auth\_time **-** mstime**(),**

server**.**cluster**->**failover\_auth\_rank**,**

replicationGetSlaveOffset**());**

*/\* Now that we have a scheduled election, broadcast our offset*

*\* to all the other slaves so that they'll updated their offsets*

*\* if our offset is better. \*/*

clusterBroadcastPong**(**CLUSTER\_BROADCAST\_LOCAL\_SLAVES**);**

**return;**

**}**

*/\* It is possible that we received more updated offsets from other*

*\* slaves for the same master since we computed our election delay.*

*\* Update the delay if our rank changed.*

*\**

*\* Not performed if this is a manual failover. \*/*

**if** **(**server**.**cluster**->**failover\_auth\_sent **==** 0 **&&**

server**.**cluster**->**mf\_end **==** 0**)**

**{**

int newrank **=** clusterGetSlaveRank**();**

**if** **(**newrank **>** server**.**cluster**->**failover\_auth\_rank**)** **{**

long long added\_delay **=**

**(**newrank **-** server**.**cluster**->**failover\_auth\_rank**)** **\*** 1000**;**

server**.**cluster**->**failover\_auth\_time **+=** added\_delay**;**

server**.**cluster**->**failover\_auth\_rank **=** newrank**;**

redisLog**(**REDIS\_WARNING**,**

"Slave rank updated to #%d, added %lld milliseconds of delay."**,**

newrank**,** added\_delay**);**

**}**

**}**

*/\* Return ASAP if we can't still start the election. \*/*

**if** **(**mstime**()** **<** server**.**cluster**->**failover\_auth\_time**)** **{**

clusterLogCantFailover**(**REDIS\_CLUSTER\_CANT\_FAILOVER\_WAITING\_DELAY**);**

**return;**

**}**

*/\* Return ASAP if the election is too old to be valid. \*/*

**if** **(**auth\_age **>** auth\_timeout**)** **{**

clusterLogCantFailover**(**REDIS\_CLUSTER\_CANT\_FAILOVER\_EXPIRED**);**

**return;**

**}**

*/\* Ask for votes if needed. \*/*

**if** **(**server**.**cluster**->**failover\_auth\_sent **==** 0**)** **{**

server**.**cluster**->**currentEpoch**++;**

server**.**cluster**->**failover\_auth\_epoch **=** server**.**cluster**->**currentEpoch**;**

redisLog**(**REDIS\_WARNING**,**"Starting a failover election for epoch %llu."**,**

**(**unsigned long long**)** server**.**cluster**->**currentEpoch**);**

clusterRequestFailoverAuth**();**

server**.**cluster**->**failover\_auth\_sent **=** 1**;**

clusterDoBeforeSleep**(**CLUSTER\_TODO\_SAVE\_CONFIG**|**

CLUSTER\_TODO\_UPDATE\_STATE**|**

CLUSTER\_TODO\_FSYNC\_CONFIG**);**

**return;** */\* Wait for replies. \*/*

**}**

**...**

**}**

server.cluster->failover\_auth\_time属性，表示从节点可以开始进行故障转移的时间。集群初始化时该属性置为0，一旦满足开始故障转移的条件后，该属性就置为未来的某个时间点，在该时间点，从节点才开始进行拉票。

函数中，首先计算auth\_age，该变量表示距离发起故障转移流程，已经过去了多少时间；然后计算needed\_quorum，该变量表示当前从节点必须至少获得多少选票，才能成为新的主节点；manual\_failover表示是否是管理员手动触发的故障转移流程；

然后计算auth\_timeout，该变量表示故障转移流程(发起投票，等待回应)的超时时间，超过该时间后还没有获得足够的选票，则表示本次故障转移失败；

计算auth\_retry\_time，该变量表示判断是否可以开始下一次故障转移流程的时间，只有距离上一次发起故障转移时，已经超过auth\_retry\_time之后，才表示可以开始下一次故障转移了（auth\_age > auth\_retry\_time）；

接下来判断当前节点是否可以进行故障转移：当前节点是主节点；当前节点是从节点但是没有主节点；当前节点的主节点不处于下线状态并且不是手动强制进行故障转移；当前节点的主节点没有负责的槽位。满足以上任一条件，则不能进行故障转移，直接返回即可；

接下来计算，现在距离当前从节点与主节点最后交互的时间data\_age，也就是当前从节点与主节点已经断链了多长时间。如果data\_age大于server.cluster\_node\_timeout，则从data\_age中减去server.cluster\_node\_timeout，因为经过server.cluster\_node\_timeout时间没有收到主节点的PING回复，才会将其标记为PFAIL，因此data\_age实际上表示：在主节点下线之前，当前从节点有多长时间没有与其交互过了。data\_age主要用于判断当前从节点的数据新鲜度；如果data\_age超过了一定时间，表示当前从节点的数据已经太老了，不能替换掉下线主节点，因此在不是手动强制故障转移的情况下，直接返回；

如果auth\_age大于auth\_retry\_time，表示可以开始进行下一次故障转移了。如果之前没有进行过故障转移，则auth\_age等于mstime，肯定大于auth\_retry\_time；如果之前进行过故障转移，则只有距离上一次发起故障转移时，已经超过auth\_retry\_time之后，才表示可以开始下一次故障转移。满足该条件后，设置故障转移流程的开始时间：server.cluster->failover\_auth\_time为mstime() + 500 + random()%500 + rank\*1000，该属性的计算原理之前已经讲过，不再赘述；

注意如果是管理员发起的手动强制执行故障转移，则设置server.cluster->failover\_auth\_time为当前时间，表示会立即开始故障转移流程；最后，调用clusterBroadcastPong，向该下线主节点的所有从节点发送PONG包，包头部分带有当前从节点的复制数据量，因此其他从节点收到之后，可以更新自己的排名；最后直接返回；

如果还没有开始故障转移，则调用clusterGetSlaveRank，取得当前从节点的最新排名。因为在开始故障转移之前，可能会收到其他从节点发来的心跳包，因而可以根据心跳包中的复制偏移量更新本节点的排名，获得新排名newrank，如果newrank比之前的排名靠后，则需要增加故障转移开始时间的延迟，然后将newrank记录到server.cluster->failover\_auth\_rank中；

如果当前时间还不到开始故障转移的时候，则直接返回即可；

如果auth\_age大于auth\_timeout，说明之前的故障转移超时了，因此直接返回；

走到这里，说明可以开始故障转移了。因此，首先增加当前节点的currentEpoch的值，表示要开始新一轮选举了。此时该从节点的currentEpoch就是所有集群节点中最大的；然后将该currentEpoch记录到server.cluster->failover\_auth\_epoch中；

然后调用clusterRequestFailoverAuth，向所有集群节点发送CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_REQUEST包用于拉票；然后，置server.cluster->failover\_auth\_sent为1，表示已发起了故障转移流程；最后直接返回；

3.3、主节点投票

集群中所有节点收到用于拉票的CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_REQUEST包后，只有负责一定槽位的主节点能投票，其他没资格的节点直接忽略掉该包。

在clusterProcessPacket中，判断收到的是CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_REQUEST包后，就会调用clusterSendFailoverAuthIfNeeded函数，在满足条件的基础上，给发送者投票。该函数的代码如下：

void clusterSendFailoverAuthIfNeeded**(**clusterNode **\***node**,** clusterMsg **\***request**)** **{**

clusterNode **\***master **=** node**->**slaveof**;**

uint64\_t requestCurrentEpoch **=** ntohu64**(**request**->**currentEpoch**);**

uint64\_t requestConfigEpoch **=** ntohu64**(**request**->**configEpoch**);**

unsigned char **\***claimed\_slots **=** request**->**myslots**;**

int force\_ack **=** request**->**mflags**[**0**]** **&** CLUSTERMSG\_FLAG0\_FORCEACK**;**

int j**;**

*/\* IF we are not a master serving at least 1 slot, we don't have the*

*\* right to vote, as the cluster size in Redis Cluster is the number*

*\* of masters serving at least one slot, and quorum is the cluster*

*\* size + 1 \*/*

**if** **(**nodeIsSlave**(**myself**)** **||** myself**->**numslots **==** 0**)** **return;**

*/\* Request epoch must be >= our currentEpoch.*

*\* Note that it is impossible for it to actually be greater since*

*\* our currentEpoch was updated as a side effect of receiving this*

*\* request, if the request epoch was greater. \*/*

**if** **(**requestCurrentEpoch **<** server**.**cluster**->**currentEpoch**)** **{**

redisLog**(**REDIS\_WARNING**,**

"Failover auth denied to %.40s: reqEpoch (%llu) < curEpoch(%llu)"**,**

node**->**name**,**

**(**unsigned long long**)** requestCurrentEpoch**,**

**(**unsigned long long**)** server**.**cluster**->**currentEpoch**);**

**return;**

**}**

*/\* I already voted for this epoch? Return ASAP. \*/*

**if** **(**server**.**cluster**->**lastVoteEpoch **==** server**.**cluster**->**currentEpoch**)** **{**

redisLog**(**REDIS\_WARNING**,**

"Failover auth denied to %.40s: already voted for epoch %llu"**,**

node**->**name**,**

**(**unsigned long long**)** server**.**cluster**->**currentEpoch**);**

**return;**

**}**

*/\* Node must be a slave and its master down.*

*\* The master can be non failing if the request is flagged*

*\* with CLUSTERMSG\_FLAG0\_FORCEACK (manual failover). \*/*

**if** **(**nodeIsMaster**(**node**)** **||** master **==** **NULL** **||**

**(!**nodeFailed**(**master**)** **&&** **!**force\_ack**))**

**{**

**if** **(**nodeIsMaster**(**node**))** **{**

redisLog**(**REDIS\_WARNING**,**

"Failover auth denied to %.40s: it is a master node"**,**

node**->**name**);**

**}** **else** **if** **(**master **==** **NULL)** **{**

redisLog**(**REDIS\_WARNING**,**

"Failover auth denied to %.40s: I don't know its master"**,**

node**->**name**);**

**}** **else** **if** **(!**nodeFailed**(**master**))** **{**

redisLog**(**REDIS\_WARNING**,**

"Failover auth denied to %.40s: its master is up"**,**

node**->**name**);**

**}**

**return;**

**}**

*/\* We did not voted for a slave about this master for two*

*\* times the node timeout. This is not strictly needed for correctness*

*\* of the algorithm but makes the base case more linear. \*/*

**if** **(**mstime**()** **-** node**->**slaveof**->**voted\_time **<** server**.**cluster\_node\_timeout **\*** 2**)**

**{**

redisLog**(**REDIS\_WARNING**,**

"Failover auth denied to %.40s: "

"can't vote about this master before %lld milliseconds"**,**

node**->**name**,**

**(**long long**)** **((**server**.**cluster\_node\_timeout**\***2**)-**

**(**mstime**()** **-** node**->**slaveof**->**voted\_time**)));**

**return;**

**}**

*/\* The slave requesting the vote must have a configEpoch for the claimed*

*\* slots that is >= the one of the masters currently serving the same*

*\* slots in the current configuration. \*/*

**for** **(**j **=** 0**;** j **<** REDIS\_CLUSTER\_SLOTS**;** j**++)** **{**

**if** **(**bitmapTestBit**(**claimed\_slots**,** j**)** **==** 0**)** **continue;**

**if** **(**server**.**cluster**->**slots**[**j**]** **==** **NULL** **||**

server**.**cluster**->**slots**[**j**]->**configEpoch **<=** requestConfigEpoch**)**

**{**

**continue;**

**}**

*/\* If we reached this point we found a slot that in our current slots*

*\* is served by a master with a greater configEpoch than the one claimed*

*\* by the slave requesting our vote. Refuse to vote for this slave. \*/*

redisLog**(**REDIS\_WARNING**,**

"Failover auth denied to %.40s: "

"slot %d epoch (%llu) > reqEpoch (%llu)"**,**

node**->**name**,** j**,**

**(**unsigned long long**)** server**.**cluster**->**slots**[**j**]->**configEpoch**,**

**(**unsigned long long**)** requestConfigEpoch**);**

**return;**

**}**

*/\* We can vote for this slave. \*/*

clusterSendFailoverAuth**(**node**);**

server**.**cluster**->**lastVoteEpoch **=** server**.**cluster**->**currentEpoch**;**

node**->**slaveof**->**voted\_time **=** mstime**();**

redisLog**(**REDIS\_WARNING**,** "Failover auth granted to %.40s for epoch %llu"**,**

node**->**name**,** **(**unsigned long long**)** server**.**cluster**->**currentEpoch**);**

**}**

首先得到包头中，发送节点的currentEpoch和configEpoch；注意，如果发送节点为从节点，则该configEpoch是其主节点的configEpoch；

如果当前节点为从节点，或者当前节点虽然为主节点，但是没有负责的槽位，则没有投票资格，因此直接返回；

如果发送者的currentEpoch小于当前节点的currentEpoch，则拒绝为其投票。因为发送者的状态与当前集群状态不一致，可能是长时间下线的节点刚刚上线，这种情况下，直接返回即可；

如果当前节点lastVoteEpoch，与当前节点的currentEpoch相等，说明本界选举中，当前节点已经投过票了，不在重复投票，直接返回（因此，如果有两个从节点同时发起拉票，则当前节点先收到哪个节点的包，就只给那个节点投票。注意，即使这两个从节点分属不同主节点，也只能有一个从节点获得选票）；

如果发送节点是主节点；或者发送节点虽然是从节点，但是找不到其主节点；或者发送节点的主节点并未下线并且这不是手动强制开始的故障转移流程，则根据不同的条件，记录日志后直接返回；

针对同一个下线主节点，在2\*server.cluster\_node\_timeout时间内，只会投一次票，这并非必须的限制条件（因为之前的lastVoteEpoch判断，已经可以避免两个从节点同时赢得本界选举了），但是这可以使得获胜从节点有时间将其成为新主节点的消息通知给其他从节点，从而避免另一个从节点发起新一轮选举又进行一次没必要的故障转移；

接下来，判断发送节点，对其宣称要负责的槽位，是否比之前负责这些槽位的节点，具有相等或更新的配置纪元configEpoch：针对16384个槽位，只要发送节点宣称要负责该槽位，就判断当前节点记录的，该槽位当前的负责节点的configEpoch，是否比发送节点的configEpoch要大，若是，说明发送节点的配置信息不是最新的，可能是一个长时间下线的节点又重新上线了，这种情况下，不能给他投票，因此直接返回；

走到这里，说明当前节点可以给发送节点投票了。因此，调用clusterSendFailoverAuth函数向发送节点发送CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_ACK包表示投票；然后将server.cluster->currentEpoch记录到server.cluster->lastVoteEpoch，表示本界选举，当前节点已经投过票了；最后记录当前投票时间到node->slaveof->voted\_time中；

3.4、从节点统计投票、赢得选举

从节点收到CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_ACK包后，就会统计投票。这部分逻辑是在函数clusterProcessPacket中处理的。这部分的代码如下：

**else** **if** **(**type **==** CLUSTERMSG\_TYPE\_FAILOVER\_AUTH\_ACK**)** **{**

**if** **(!**sender**)** **return** 1**;** */\* We don't know that node. \*/*

*/\* We consider this vote only if the sender is a master serving*

*\* a non zero number of slots, and its currentEpoch is greater or*

*\* equal to epoch where this node started the election. \*/*

**if** **(**nodeIsMaster**(**sender**)** **&&** sender**->**numslots **>** 0 **&&**

senderCurrentEpoch **>=** server**.**cluster**->**failover\_auth\_epoch**)**

**{**

server**.**cluster**->**failover\_auth\_count**++;**

*/\* Maybe we reached a quorum here, set a flag to make sure*

*\* we check ASAP. \*/*

clusterDoBeforeSleep**(**CLUSTER\_TODO\_HANDLE\_FAILOVER**);**

**}**

**}**

如果发送节点是主节点，并且该主节点有负责的槽位，并且发送节点的CurrentEpoch，大于等于当前节点发起选举时的CurrentEpoch（否则，可能是当前节点之前发起过一轮选举，失败后，又发起了新一轮选举；而现在收到的包，是针对之前那一轮选举的投票（有可能在网络中迷路了一段时间）），满足以上条件，表示选票有效，因此增加server.cluster->failover\_auth\_count的值；

在clusterHandleSlaveFailover函数的最后一部分，从节点判断收到了大部分主节点的投票之后，就会开始升级为主节点。这部分的代码如下：

void clusterHandleSlaveFailover**(**void**)** **{**

int needed\_quorum **=** **(**server**.**cluster**->**size **/** 2**)** **+** 1**;**

**...**

*/\* Check if we reached the quorum. \*/*

**if** **(**server**.**cluster**->**failover\_auth\_count **>=** needed\_quorum**)** **{**

*/\* We have the quorum, we can finally failover the master. \*/*

redisLog**(**REDIS\_WARNING**,**

"Failover election won: I'm the new master."**);**

*/\* Update my configEpoch to the epoch of the election. \*/*

**if** **(**myself**->**configEpoch **<** server**.**cluster**->**failover\_auth\_epoch**)** **{**

myself**->**configEpoch **=** server**.**cluster**->**failover\_auth\_epoch**;**

redisLog**(**REDIS\_WARNING**,**

"configEpoch set to %llu after successful failover"**,**

**(**unsigned long long**)** myself**->**configEpoch**);**

**}**

*/\* Take responsability for the cluster slots. \*/*

clusterFailoverReplaceYourMaster**();**

**}** **else** **{**

clusterLogCantFailover**(**REDIS\_CLUSTER\_CANT\_FAILOVER\_WAITING\_VOTES**);**

**}**

**}**

计算needed\_quorum，该变量表示当前从节点必须至少获得多少选票，才能成为新的主节点；

如果server.cluster->failover\_auth\_count的值大于needed\_quorum，表明当前从节点已经受到了大部分节点的支持，可以成为新的主节点了。

因此，首先更新myself->configEpoch为server.cluster->failover\_auth\_epoch，这样当前节点的configEpoch就成为所有集群节点中最大的了，方便后续更新配置。这种产生新configEpoch的方式是经过协商过的，因为只有从节点赢得大部分主节点投票的时候，才会产生新的configEpoch；最后，调用clusterFailoverReplaceYourMaster函数，取代下线主节点，成为新的主节点，并向其他节点广播这种变化。

clusterFailoverReplaceYourMaster函数的代码如下：

void clusterFailoverReplaceYourMaster**(**void**)** **{**

int j**;**

clusterNode **\***oldmaster **=** myself**->**slaveof**;**

**if** **(**nodeIsMaster**(**myself**)** **||** oldmaster **==** **NULL)** **return;**

*/\* 1) Turn this node into a master. \*/*

clusterSetNodeAsMaster**(**myself**);**

replicationUnsetMaster**();**

*/\* 2) Claim all the slots assigned to our master. \*/*

**for** **(**j **=** 0**;** j **<** REDIS\_CLUSTER\_SLOTS**;** j**++)** **{**

**if** **(**clusterNodeGetSlotBit**(**oldmaster**,**j**))** **{**

clusterDelSlot**(**j**);**

clusterAddSlot**(**myself**,**j**);**

**}**

**}**

*/\* 3) Update state and save config. \*/*

clusterUpdateState**();**

clusterSaveConfigOrDie**(**1**);**

*/\* 4) Pong all the other nodes so that they can update the state*

*\* accordingly and detect that we switched to master role. \*/*

clusterBroadcastPong**(**CLUSTER\_BROADCAST\_ALL**);**

*/\* 5) If there was a manual failover in progress, clear the state. \*/*

resetManualFailover**();**

**}**

首先调用clusterSetNodeAsMaster，将当前从节点从其主节点的slaves数组中删除，将当前节点的标志位中，清除REDIS\_NODE\_SLAVE标记，增加REDIS\_NODE\_MASTER标记，并置当前节点的主节点为NULL，因此，调用该函数之后，当前节点在集群中的的角色就是主节点了；

然后调用replicationUnsetMaster，取消主从复制过程，将当前节点升级为主节点；

然后轮训16384个槽位，当前节点接手老的主节点负责的槽位；

然后调用clusterUpdateState和clusterSaveConfigOrDie，看是否需要修改集群状态（由下线转为上线），然后将配置保存到本地配置文件中；

然后，向所有集群节点广播PONG包，使得"当前节点成为新主节点并接手相应槽位"的消息，尽快通知给其他节点；

最后，调用resetManualFailover，重置手动强制故障转移的状态。

4：更新配置

经过故障转移之后，某个从节点升级成了主节点，并接手原主节点所负责的槽位。接下来就需要更新配置信息，使得其他节点能感知到，这些槽位现在由新的节点负责了。此时就是configEpoch发挥作用的时候了。

在上一节中，从节点成为主节点，接手下线主节点的槽位后，会向所有集群节点广播PONG包，使得所有集群节点能够更新关于这几个槽位负责节点的配置信息。更新配置这部分逻辑是在clusterProcessPacket函数中处理的，这部分的代码如下：

**if** **(**type **==** CLUSTERMSG\_TYPE\_PING **||** type **==** CLUSTERMSG\_TYPE\_PONG **||**

type **==** CLUSTERMSG\_TYPE\_MEET**)**

**{**

**...**

*/\* Check for role switch: slave -> master or master -> slave. \*/*

**if** **(**sender**)** **{**

**if** **(!**memcmp**(**hdr**->**slaveof**,**REDIS\_NODE\_NULL\_NAME**,**

**sizeof(**hdr**->**slaveof**)))**

**{**

*/\* Node is a master. \*/*

clusterSetNodeAsMaster**(**sender**);**

**}** **else** **{**

*/\* Node is a slave. \*/*

**...**

**}**

**}**

*/\* Update our info about served slots.*

*\**

*\* Note: this MUST happen after we update the master/slave state*

*\* so that REDIS\_NODE\_MASTER flag will be set. \*/*

*/\* Many checks are only needed if the set of served slots this*

*\* instance claims is different compared to the set of slots we have*

*\* for it. Check this ASAP to avoid other computational expansive*

*\* checks later. \*/*

clusterNode **\***sender\_master **=** **NULL;** */\* Sender or its master if slave. \*/*

int dirty\_slots **=** 0**;** */\* Sender claimed slots don't match my view? \*/*

**if** **(**sender**)** **{**

sender\_master **=** nodeIsMaster**(**sender**)** **?** sender **:** sender**->**slaveof**;**

**if** **(**sender\_master**)** **{**

dirty\_slots **=** memcmp**(**sender\_master**->**slots**,**

hdr**->**myslots**,sizeof(**hdr**->**myslots**))** **!=** 0**;**

**}**

**}**

*/\* 1) If the sender of the message is a master, and we detected that*

*\* the set of slots it claims changed, scan the slots to see if we*

*\* need to update our configuration. \*/*

**if** **(**sender **&&** nodeIsMaster**(**sender**)** **&&** dirty\_slots**)**

clusterUpdateSlotsConfigWith**(**sender**,**senderConfigEpoch**,**hdr**->**myslots**);**

*/\* 2) We also check for the reverse condition, that is, the sender*

*\* claims to serve slots we know are served by a master with a*

*\* greater configEpoch. If this happens we inform the sender.*

*\**

*\* This is useful because sometimes after a partition heals, a*

*\* reappearing master may be the last one to claim a given set of*

*\* hash slots, but with a configuration that other instances know to*

*\* be deprecated. Example:*

*\**

*\* A and B are master and slave for slots 1,2,3.*

*\* A is partitioned away, B gets promoted.*

*\* B is partitioned away, and A returns available.*

*\**

*\* Usually B would PING A publishing its set of served slots and its*

*\* configEpoch, but because of the partition B can't inform A of the*

*\* new configuration, so other nodes that have an updated table must*

*\* do it. In this way A will stop to act as a master (or can try to*

*\* failover if there are the conditions to win the election). \*/*

**if** **(**sender **&&** dirty\_slots**)** **{**

int j**;**

**for** **(**j **=** 0**;** j **<** REDIS\_CLUSTER\_SLOTS**;** j**++)** **{**

**if** **(**bitmapTestBit**(**hdr**->**myslots**,**j**))** **{**

**if** **(**server**.**cluster**->**slots**[**j**]** **==** sender **||**

server**.**cluster**->**slots**[**j**]** **==** **NULL)** **continue;**

**if** **(**server**.**cluster**->**slots**[**j**]->**configEpoch **>**

senderConfigEpoch**)**

**{**

redisLog**(**REDIS\_VERBOSE**,**

"Node %.40s has old slots configuration, sending "

"an UPDATE message about %.40s"**,**

sender**->**name**,** server**.**cluster**->**slots**[**j**]->**name**);**

clusterSendUpdate**(**sender**->**link**,**

server**.**cluster**->**slots**[**j**]);**

*/\* TODO: instead of exiting the loop send every other*

*\* UPDATE packet for other nodes that are the new owner*

*\* of sender's slots. \*/*

**break;**

**}**

**}**

**}**

**}**

**}**

如果发送节点的slaveof为空，说明发送节点为主节点。则调用clusterSetNodeAsMaster函数。函数中，如果发送节点已经是主节点则直接返回，如果发送节点之前是从节点，则该函数会将其置为主节点；

接下来，判断发送节点宣称负责的槽位，是否与当前节点记录的不同，若不同，则置dirty\_slots为1。

一种情况是，如果该节点之前是从节点，则其负责的槽位为空，因此像刚刚完成故障转移的新主节点，它在当前节点的视角中，负责的槽位就为空，但是该节点在其发送的PONG包中，会宣称其负责原主节点的槽位，所以这里的dirty\_slots会被置为1；

如果发送节点宣称负责的槽位与当前节点记录的不同，并且发送节点是主节点，则说明该发送节点可能是刚刚完成故障转移，新上任的主节点。这种情况下，调用函数clusterUpdateSlotsConfigWith，更新当前节点关于发送节点的配置信息（这是处理senderConfigEpoch大于server.cluster->slots[j]->configEpoch的情况）。

还有一种情况就是，之前下线的主节点，经过一段时间之后又重新上线了。而此时在其他节点眼中，它所负责的槽位已经被其他节点所接手了。因此，在重新上线的主节点发来的心跳包中，所宣称负责的槽位与当前节点记录的不同，所以这里的dirty\_slots也会被置为1。

这种情况下，轮训16384个槽位，只要发送节点宣称负责的一个槽位，与当前节点记录的负责该槽位的节点不一致，并且发送节点的配置纪元configepoch更小，说明发送节点的配置信息需要更新，因此向该节点发送UPDATE包，包中带有最新负责该槽位的节点信息；（这是处理senderConfigEpoch小于server.cluster->slots[j]->configEpoch的情况）

节点在收到UPDATE包后，在clusterProcessPacket函数中，相应的处理逻辑是：

**else** **if** **(**type **==** CLUSTERMSG\_TYPE\_UPDATE**)** **{**

clusterNode **\***n**;** */\* The node the update is about. \*/*

uint64\_t reportedConfigEpoch **=**

ntohu64**(**hdr**->**data**.**update**.**nodecfg**.**configEpoch**);**

**if** **(!**sender**)** **return** 1**;** */\* We don't know the sender. \*/*

n **=** clusterLookupNode**(**hdr**->**data**.**update**.**nodecfg**.**nodename**);**

**if** **(!**n**)** **return** 1**;** */\* We don't know the reported node. \*/*

**if** **(**n**->**configEpoch **>=** reportedConfigEpoch**)** **return** 1**;** */\* Nothing new. \*/*

*/\* If in our current config the node is a slave, set it as a master. \*/*

**if** **(**nodeIsSlave**(**n**))** clusterSetNodeAsMaster**(**n**);**

*/\* Update the node's configEpoch. \*/*

n**->**configEpoch **=** reportedConfigEpoch**;**

clusterDoBeforeSleep**(**CLUSTER\_TODO\_SAVE\_CONFIG**|**

CLUSTER\_TODO\_FSYNC\_CONFIG**);**

*/\* Check the bitmap of served slots and update our*

*\* config accordingly. \*/*

clusterUpdateSlotsConfigWith**(**n**,**reportedConfigEpoch**,**

hdr**->**data**.**update**.**nodecfg**.**slots**);**

**}**

首先取得包中，宣称负责槽位的节点的configEpoch：reportedConfigEpoch；

如果在字典server.cluster->nodes中找不到发送节点，说明还不认识发送节点，则直接返回；然后在字典server.cluster->nodes中寻找宣称负责槽位的节点n；如果找不到n，则直接返回；如果当前节点记录的n的configEpoch，比reportedConfigEpoch大，则不能更新配置，直接返回；

如果当前节点记录的n为从节点，则调用clusterSetNodeAsMaster，将其标记为主节点；然后更新n节点的configEpoch；

最后，调用clusterUpdateSlotsConfigWith函数，更新当前节点关于槽位的负责节点信息，并在满足条件的情况下，使当前节点成为节点n的从节点；

因此，更新配置信息，不管哪种情况，最终都是通过clusterUpdateSlotsConfigWith函数实现的。该函数的代码如下：

void clusterUpdateSlotsConfigWith**(**clusterNode **\***sender**,** uint64\_t senderConfigEpoch**,** unsigned char **\***slots**)** **{**

int j**;**

clusterNode **\***curmaster**,** **\***newmaster **=** **NULL;**

*/\* The dirty slots list is a list of slots for which we lose the ownership*

*\* while having still keys inside. This usually happens after a failover*

*\* or after a manual cluster reconfiguration operated by the admin.*

*\**

*\* If the update message is not able to demote a master to slave (in this*

*\* case we'll resync with the master updating the whole key space), we*

*\* need to delete all the keys in the slots we lost ownership. \*/*

uint16\_t dirty\_slots**[**REDIS\_CLUSTER\_SLOTS**];**

int dirty\_slots\_count **=** 0**;**

*/\* Here we set curmaster to this node or the node this node*

*\* replicates to if it's a slave. In the for loop we are*

*\* interested to check if slots are taken away from curmaster. \*/*

curmaster **=** nodeIsMaster**(**myself**)** **?** myself **:** myself**->**slaveof**;**

**if** **(**sender **==** myself**)** **{**

redisLog**(**REDIS\_WARNING**,**"Discarding UPDATE message about myself."**);**

**return;**

**}**

**for** **(**j **=** 0**;** j **<** REDIS\_CLUSTER\_SLOTS**;** j**++)** **{**

**if** **(**bitmapTestBit**(**slots**,**j**))** **{**

*/\* The slot is already bound to the sender of this message. \*/*

**if** **(**server**.**cluster**->**slots**[**j**]** **==** sender**)** **continue;**

*/\* The slot is in importing state, it should be modified only*

*\* manually via redis-trib (example: a resharding is in progress*

*\* and the migrating side slot was already closed and is advertising*

*\* a new config. We still want the slot to be closed manually). \*/*

**if** **(**server**.**cluster**->**importing\_slots\_from**[**j**])** **continue;**

*/\* We rebind the slot to the new node claiming it if:*

*\* 1) The slot was unassigned or the new node claims it with a*

*\* greater configEpoch.*

*\* 2) We are not currently importing the slot. \*/*

**if** **(**server**.**cluster**->**slots**[**j**]** **==** **NULL** **||**

server**.**cluster**->**slots**[**j**]->**configEpoch **<** senderConfigEpoch**)**

**{**

*/\* Was this slot mine, and still contains keys? Mark it as*

*\* a dirty slot. \*/*

**if** **(**server**.**cluster**->**slots**[**j**]** **==** myself **&&**

countKeysInSlot**(**j**)** **&&**

sender **!=** myself**)**

**{**

dirty\_slots**[**dirty\_slots\_count**]** **=** j**;**

dirty\_slots\_count**++;**

**}**

**if** **(**server**.**cluster**->**slots**[**j**]** **==** curmaster**)**

newmaster **=** sender**;**

clusterDelSlot**(**j**);**

clusterAddSlot**(**sender**,**j**);**

clusterDoBeforeSleep**(**CLUSTER\_TODO\_SAVE\_CONFIG**|**

CLUSTER\_TODO\_UPDATE\_STATE**|**

CLUSTER\_TODO\_FSYNC\_CONFIG**);**

**}**

**}**

**}**

*/\* If at least one slot was reassigned from a node to another node*

*\* with a greater configEpoch, it is possible that:*

*\* 1) We are a master left without slots. This means that we were*

*\* failed over and we should turn into a replica of the new*

*\* master.*

*\* 2) We are a slave and our master is left without slots. We need*

*\* to replicate to the new slots owner. \*/*

**if** **(**newmaster **&&** curmaster**->**numslots **==** 0**)** **{**

redisLog**(**REDIS\_WARNING**,**

"Configuration change detected. Reconfiguring myself "

"as a replica of %.40s"**,** sender**->**name**);**

clusterSetMaster**(**sender**);**

clusterDoBeforeSleep**(**CLUSTER\_TODO\_SAVE\_CONFIG**|**

CLUSTER\_TODO\_UPDATE\_STATE**|**

CLUSTER\_TODO\_FSYNC\_CONFIG**);**

**}** **else** **if** **(**dirty\_slots\_count**)** **{**

*/\* If we are here, we received an update message which removed*

*\* ownership for certain slots we still have keys about, but still*

*\* we are serving some slots, so this master node was not demoted to*

*\* a slave.*

*\**

*\* In order to maintain a consistent state between keys and slots*

*\* we need to remove all the keys from the slots we lost. \*/*

**for** **(**j **=** 0**;** j **<** dirty\_slots\_count**;** j**++)**

delKeysInSlot**(**dirty\_slots**[**j**]);**

**}**

**}**

本函数处理这样的场景：节点sender宣称自己的configEpoch和负责的槽位信息slots，但是这些槽位之前由其他节点负责，而sender节点的configEpoch更大，说明需要更新这些槽位的负责节点为sender。

本函数会由多种角色的节点执行，比如是之前下线的主节点，经过一段时间后，又重新上线了，该重新上线主节点会收到其他节点发来的UPDATE包，会通过该函数更新自己的配置信息，并成为其他节点的从节点；也可以是已下线主节点的其他从节点，收到新主节点发来的心跳包之后，通过该函数更新自己的配置信息，并成为新上任主节点的从节点；又可以是集群中的其他节点，收到新主节点发来的心跳包后，仅仅更新自己的配置信息；或者是集群刚建立时，当前节点收到其他节点宣称负责某些槽位的包后，更新自己的配置信息；

函数中，首先得到当前节点的主节点curmaster，如果当前节点是主节点，则curmaster就是myself，否则，curmaster是myself->slaveof；

如果sender就是当前节点，则直接返回；

接下来轮训16384个槽位，只要sender宣称负责该槽位，则进行处理：

如果该槽位的负责节点已经是sender，则直接处理下一个槽位；如果当前节点正在迁入该槽位，则直接处理下一个槽位；

如果该槽位尚未有负责节点（可能是集群刚建立时），或者该槽位的负责节点的configEpoch小于sender节点的configEpoch，则需要将该槽位改为由sender负责：如果该槽位目前是由当前节点负责，并且槽位中尚有key，这种情况，说明当前节点是下线后又重新上线的旧主节点。因此，将该槽位记录到dirty\_slots数组中；如果该槽位现在由curmaster负责，说明当前节点要么是下线后又重新上线的节点，要么是下线主节点的其他从节点，两种情况，都需要当前节点成为新主节点的从节点，因此置newmaster为sender；接下来就是调用clusterDelSlot和clusterAddSlot，将该槽位的负责节点改为sender；

轮训完所有槽位之后，如果设置了newmaster，并且curmaster负责的槽位已清空，则可以将当前节点置为sender的从节点了，因此调用clusterSetMaster置sender为当前节点的主节点；

如果不满足上面的条件，并且dirty\_slots\_count不为0，则轮训数组dirty\_slots，将其中每个槽位的所有key，从数据库中删除。

这种情况的场景是：下线主节点A，原来负责槽位1,2,3，经过很长一段时间，现在A又重新上线了，但是现在槽位1,2由B节点负责，而槽位3由C节点负责。A现在收到的UPDATE包，其中只有节点B负责槽位1,2的信息（因为其他节点D收到A的包之后，发现A宣称负责的1,2,3槽位，现在由其他节点负责了。节点D轮训16384个槽位，只要发现槽位1的负责节点B的configEpoch大于A的configEpoch，它就只会发出一个UPDATE包，其中只带有节点B的信息（函数clusterProcessPacket中发送UPDATE包的逻辑）），因此节点A收到该UPDATE包之后，只能先将槽位1和2删除，并将其中的KEY从数据库中删除，只有下次收到关于节点C负责槽位3的UPDATE包之后，把槽位3也清除了，此时就符合curmaster->numslots == 0的条件了，才能把自己置为C的从节点。

参考：

http://redis.io/topics/cluster-spec