**一：手动故障转移**

Redis集群支持手动故障转移。也就是向从节点发送”CLUSTER FAILOVER”命令，使其在主节点未下线的情况下，发起故障转移流程，升级为新的主节点，而原来的主节点降级为从节点。

为了不丢失数据，向从节点发送”CLUSTER FAILOVER”命令后，流程如下：

a：从节点收到命令后，向主节点发送CLUSTERMSG\_TYPE\_MFSTART包；

b：主节点收到该包后，会将其所有客户端置于阻塞状态，也就是在10s的时间内，不再处理客户端发来的命令；并且在其发送的心跳包中，会带有CLUSTERMSG\_FLAG0\_PAUSED标记；

c：从节点收到主节点发来的，带CLUSTERMSG\_FLAG0\_PAUSED标记的心跳包后，从中获取主节点当前的复制偏移量。从节点等到自己的复制偏移量达到该值后，才会开始执行故障转移流程：发起选举、统计选票、赢得选举、升级为主节点并更新配置；

”CLUSTER FAILOVER”命令支持两个选项：FORCE和TAKEOVER。使用这两个选项，可以改变上述的流程。

如果有FORCE选项，则从节点不会与主节点进行交互，主节点也不会阻塞其客户端，而是从节点立即开始故障转移流程：发起选举、统计选票、赢得选举、升级为主节点并更新配置。

如果有TAKEOVER选项，则更加简单粗暴：从节点不再发起选举，而是直接将自己升级为主节点，接手原主节点的槽位，增加自己的configEpoch后更新配置。

因此，使用FORCE和TAKEOVER选项，主节点可以已经下线；而不使用任何选项，只发送”CLUSTER FAILOVER”命令的话，主节点必须在线。

在clusterCommand函数中，处理”CLUSTER FAILOVER”命令的部分代码如下：

**else** **if** **(!**strcasecmp**(**c**->**argv**[**1**]->**ptr**,**"failover"**)** **&&**

**(**c**->**argc **==** 2 **||** c**->**argc **==** 3**))**

**{**

*/\* CLUSTER FAILOVER [FORCE|TAKEOVER] \*/*

int force **=** 0**,** takeover **=** 0**;**

**if** **(**c**->**argc **==** 3**)** **{**

**if** **(!**strcasecmp**(**c**->**argv**[**2**]->**ptr**,**"force"**))** **{**

force **=** 1**;**

**}** **else** **if** **(!**strcasecmp**(**c**->**argv**[**2**]->**ptr**,**"takeover"**))** **{**

takeover **=** 1**;**

force **=** 1**;** */\* Takeover also implies force. \*/*

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

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

**return;**

**}**

**}**

*/\* Check preconditions. \*/*

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

addReplyError**(**c**,**"You should send CLUSTER FAILOVER to a slave"**);**

**return;**

**}** **else** **if** **(**myself**->**slaveof **==** **NULL)** **{**

addReplyError**(**c**,**"I'm a slave but my master is unknown to me"**);**

**return;**

**}** **else** **if** **(!**force **&&**

**(**nodeFailed**(**myself**->**slaveof**)** **||**

myself**->**slaveof**->**link **==** **NULL))**

**{**

addReplyError**(**c**,**"Master is down or failed, "

"please use CLUSTER FAILOVER FORCE"**);**

**return;**

**}**

resetManualFailover**();**

server**.**cluster**->**mf\_end **=** mstime**()** **+** REDIS\_CLUSTER\_MF\_TIMEOUT**;**

**if** **(**takeover**)** **{**

*/\* A takeover does not perform any initial check. It just*

*\* generates a new configuration epoch for this node without*

*\* consensus, claims the master's slots, and broadcast the new*

*\* configuration. \*/*

redisLog**(**REDIS\_WARNING**,**"Taking over the master (user request)."**);**

clusterBumpConfigEpochWithoutConsensus**();**

clusterFailoverReplaceYourMaster**();**

**}** **else** **if** **(**force**)** **{**

*/\* If this is a forced failover, we don't need to talk with our*

*\* master to agree about the offset. We just failover taking over*

*\* it without coordination. \*/*

redisLog**(**REDIS\_WARNING**,**"Forced failover user request accepted."**);**

server**.**cluster**->**mf\_can\_start **=** 1**;**

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

redisLog**(**REDIS\_WARNING**,**"Manual failover user request accepted."**);**

clusterSendMFStart**(**myself**->**slaveof**);**

**}**

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

**}**

首先检查命令的最后一个参数是否是FORCE或TAKEOVER；

如果当前节点是主节点；或者当前节点是从节点，但没有主节点；或者当前从节点的主节点已经下线或者断链，并且命令中没有FORCE或TAKEOVER参数，则直接回复客户端错误信息后返回；

然后调用resetManualFailover，重置手动强制故障转移的状态；

置mf\_end为当前时间加5秒，该属性表示手动强制故障转移流程的超时时间，也用来表示当前是否正在进行手动强制故障转移；

如果命令最后一个参数为TAKEOVER，这表示收到命令的从节点无需经过选举的过程，直接接手其主节点的槽位，并成为新的主节点。因此首先调用函数clusterBumpConfigEpochWithoutConsensus，产生新的configEpoch，以便后续更新配置；然后调用clusterFailoverReplaceYourMaster函数，转变成为新的主节点，并将这种转变广播给集群中所有节点；

如果命令最后一个参数是FORCE，这表示收到命令的从节点可以直接开始选举过程，而无需达到主节点的复制偏移量之后才开始选举过程。因此置mf\_can\_start为1，这样在函数clusterHandleSlaveFailover中，即使在主节点未下线或者当前从节点的复制数据比较旧的情况下，也可以开始故障转移流程；

如果最后一个参数不是FORCE或TAKEOVER，这表示收到命令的从节点，首先需要向主节点发送CLUSTERMSG\_TYPE\_MFSTART包，因此调用clusterSendMFStart函数，向其主节点发送该包；

主节点收到CLUSTERMSG\_TYPE\_MFSTART包后，在clusterProcessPacket函数中，是这样处理的：

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

*/\* This message is acceptable only if I'm a master and the sender*

*\* is one of my slaves. \*/*

**if** **(!**sender **||** sender**->**slaveof **!=** myself**)** **return** 1**;**

*/\* Manual failover requested from slaves. Initialize the state*

*\* accordingly. \*/*

resetManualFailover**();**

server**.**cluster**->**mf\_end **=** mstime**()** **+** REDIS\_CLUSTER\_MF\_TIMEOUT**;**

server**.**cluster**->**mf\_slave **=** sender**;**

pauseClients**(**mstime**()+(**REDIS\_CLUSTER\_MF\_TIMEOUT**\***2**));**

redisLog**(**REDIS\_WARNING**,**"Manual failover requested by slave %.40s."**,**

sender**->**name**);**

**}**

如果字典中找不到发送节点，或者发送节点的主节点不是当前节点，则直接返回；

调用resetManualFailover，重置手动强制故障转移的状态；

然后置mf\_end为当前时间加5秒，该属性表示手动强制故障转移流程的超时时间，也用来表示当前是否正在进行手动强制故障转移；

然后设置mf\_slave为sender，该属性表示要进行手动强制故障转移的从节点；

然后调用pauseClients，使所有客户端在之后的10s内阻塞；

主节点在发送心跳包时，在构建包头时，如果发现当前正处于手动强制故障转移阶段，则会在包头中增加CLUSTERMSG\_FLAG0\_PAUSED标记：

void clusterBuildMessageHdr**(**clusterMsg **\***hdr**,** int type**)** **{**

**...**

*/\* Set the message flags. \*/*

**if** **(**nodeIsMaster**(**myself**)** **&&** server**.**cluster**->**mf\_end**)**

hdr**->**mflags**[**0**]** **|=** CLUSTERMSG\_FLAG0\_PAUSED**;**

**...**

**}**

从节点在clusterProcessPacket函数中处理收到的包，一旦发现主节点发来的，带有CLUSTERMSG\_FLAG0\_PAUSED标记的包，就会将该主节点的复制偏移量记录到server.cluster->mf\_master\_offset中：

int clusterProcessPacket**(**clusterLink **\***link**)** **{**

**...**

*/\* Check if the sender is a known node. \*/*

sender **=** clusterLookupNode**(**hdr**->**sender**);**

**if** **(**sender **&&** **!**nodeInHandshake**(**sender**))** **{**

**...**

*/\* Update the replication offset info for this node. \*/*

sender**->**repl\_offset **=** ntohu64**(**hdr**->**offset**);**

sender**->**repl\_offset\_time **=** mstime**();**

*/\* If we are a slave performing a manual failover and our master*

*\* sent its offset while already paused, populate the MF state. \*/*

**if** **(**server**.**cluster**->**mf\_end **&&**

nodeIsSlave**(**myself**)** **&&**

myself**->**slaveof **==** sender **&&**

hdr**->**mflags**[**0**]** **&** CLUSTERMSG\_FLAG0\_PAUSED **&&**

server**.**cluster**->**mf\_master\_offset **==** 0**)**

**{**

server**.**cluster**->**mf\_master\_offset **=** sender**->**repl\_offset**;**

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

"Received replication offset for paused "

"master manual failover: %lld"**,**

server**.**cluster**->**mf\_master\_offset**);**

**}**

**}**

**}**

从节点在集群定时器函数clusterCron中，会调用clusterHandleManualFailover函数，判断一旦当前从节点的复制偏移量达到了server.cluster->mf\_master\_offset，就会置server.cluster->mf\_can\_start为1。这样在接下来要调用的clusterHandleSlaveFailover函数中，就会立即开始故障转移流程了。

clusterHandleManualFailover函数的代码如下：

void clusterHandleManualFailover**(**void**)** **{**

*/\* Return ASAP if no manual failover is in progress. \*/*

**if** **(**server**.**cluster**->**mf\_end **==** 0**)** **return;**

*/\* If mf\_can\_start is non-zero, the failover was already triggered so the*

*\* next steps are performed by clusterHandleSlaveFailover(). \*/*

**if** **(**server**.**cluster**->**mf\_can\_start**)** **return;**

**if** **(**server**.**cluster**->**mf\_master\_offset **==** 0**)** **return;** */\* Wait for offset... \*/*

**if** **(**server**.**cluster**->**mf\_master\_offset **==** replicationGetSlaveOffset**())** **{**

*/\* Our replication offset matches the master replication offset*

*\* announced after clients were paused. We can start the failover. \*/*

server**.**cluster**->**mf\_can\_start **=** 1**;**

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

"All master replication stream processed, "

"manual failover can start."**);**

**}**

**}**

不管是从节点，还是主节点，在集群定时器函数clusterCron中，都会调用manualFailoverCheckTimeout函数，一旦发现手动故障转移的超时时间已到，就会重置手动故障转移的状态，表示终止该过程。manualFailoverCheckTimeout函数代码如下：

/\* If a manual failover timed out, abort it. \*/

void manualFailoverCheckTimeout**(**void**)** **{**

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

redisLog**(**REDIS\_WARNING**,**"Manual failover timed out."**);**

resetManualFailover**();**

**}**

**}**

**二：从节点迁移**

在Redis集群中，为了增强集群的可用性，一般情况下需要为每个主节点配置若干从节点。但是这种主从关系如果是固定不变的，则经过一段时间之后，就有可能出现孤立主节点的情况，也就是一个主节点再也没有可用于故障转移的从节点了，一旦这样的主节点下线，整个集群也就不可用了。

因此，在Redis集群中，增加了从节点迁移的功能。简单描述如下：一旦发现集群中出现了孤立主节点，则某个从节点A就会自动变成该孤立主节点的从节点。该从节点A满足这样的条件：A的主节点具有最多的附属从节点；A在这些附属从节点中，节点ID是最小的（The *acting slave* is the slave among the masters with the maximum number of attached slaves, that is not in FAIL state and has the smallest node ID）。

该功能是在集群定时器函数clusterCron中实现的。这部分的代码如下：

void clusterCron**(**void**)** **{**

**...**

orphaned\_masters **=** 0**;**

max\_slaves **=** 0**;**

this\_slaves **=** 0**;**

di **=** dictGetSafeIterator**(**server**.**cluster**->**nodes**);**

**while((**de **=** dictNext**(**di**))** **!=** **NULL)** **{**

clusterNode **\***node **=** dictGetVal**(**de**);**

now **=** mstime**();** */\* Use an updated time at every iteration. \*/*

mstime\_t delay**;**

**if** **(**node**->**flags **&**

**(**REDIS\_NODE\_MYSELF**|**REDIS\_NODE\_NOADDR**|**REDIS\_NODE\_HANDSHAKE**))**

**continue;**

*/\* Orphaned master check, useful only if the current instance*

*\* is a slave that may migrate to another master. \*/*

**if** **(**nodeIsSlave**(**myself**)** **&&** nodeIsMaster**(**node**)** **&&** **!**nodeFailed**(**node**))** **{**

int okslaves **=** clusterCountNonFailingSlaves**(**node**);**

*/\* A master is orphaned if it is serving a non-zero number of*

*\* slots, have no working slaves, but used to have at least one*

*\* slave. \*/*

**if** **(**okslaves **==** 0 **&&** node**->**numslots **>** 0 **&&** node**->**numslaves**)**

orphaned\_masters**++;**

**if** **(**okslaves **>** max\_slaves**)** max\_slaves **=** okslaves**;**

**if** **(**nodeIsSlave**(**myself**)** **&&** myself**->**slaveof **==** node**)**

this\_slaves **=** okslaves**;**

**}**

**...**

**}**

**...**

**if** **(**nodeIsSlave**(**myself**))** **{**

**...**

*/\* If there are orphaned slaves, and we are a slave among the masters*

*\* with the max number of non-failing slaves, consider migrating to*

*\* the orphaned masters. Note that it does not make sense to try*

*\* a migration if there is no master with at least \*two\* working*

*\* slaves. \*/*

**if** **(**orphaned\_masters **&&** max\_slaves **>=** 2 **&&** this\_slaves **==** max\_slaves**)**

clusterHandleSlaveMigration**(**max\_slaves**);**

**}**

**...**

**}**

轮训字典server.cluster->nodes，只要其中的节点不是当前节点，没有处于REDIS\_NODE\_NOADDR或者握手状态，就对该node节点做相应的处理：

如果当前节点是从节点，并且node节点是主节点，并且node未被标记为下线，则首先调用函数clusterCountNonFailingSlaves，计算node节点未下线的从节点个数okslaves，如果node主节点的okslaves为0，并且该主节点负责的插槽数不为0，说明该node主节点是孤立主节点，因此增加orphaned\_masters的值；如果该node主节点的okslaves大于max\_slaves，则将max\_slaves改为okslaves，因此，max\_slaves记录了所有主节点中，拥有最多未下线从节点的那个主节点的未下线从节点个数；如果当前节点正好是node主节点的从节点之一，则将okslaves记录到this\_slaves中，以上都是为后续做从节点迁移做的准备；

轮训完所有节点之后，如果存在孤立主节点，并且max\_slaves大于等于2，并且当前节点刚好是那个拥有最多未下线从节点的主节点的众多从节点之一，则调用函数clusterHandleSlaveMigration，满足条件的情况下，进行从节点迁移，也就是将当前从节点置为某孤立主节点的从节点。

clusterHandleSlaveMigration函数的代码如下：

void clusterHandleSlaveMigration**(**int max\_slaves**)** **{**

int j**,** okslaves **=** 0**;**

clusterNode **\***mymaster **=** myself**->**slaveof**,** **\***target **=** **NULL,** **\***candidate **=** **NULL;**

dictIterator **\***di**;**

dictEntry **\***de**;**

*/\* Step 1: Don't migrate if the cluster state is not ok. \*/*

**if** **(**server**.**cluster**->**state **!=** REDIS\_CLUSTER\_OK**)** **return;**

*/\* Step 2: Don't migrate if my master will not be left with at least*

*\* 'migration-barrier' slaves after my migration. \*/*

**if** **(**mymaster **==** **NULL)** **return;**

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

**if** **(!**nodeFailed**(**mymaster**->**slaves**[**j**])** **&&**

**!**nodeTimedOut**(**mymaster**->**slaves**[**j**]))** okslaves**++;**

**if** **(**okslaves **<=** server**.**cluster\_migration\_barrier**)** **return;**

*/\* Step 3: Idenitfy a candidate for migration, and check if among the*

*\* masters with the greatest number of ok slaves, I'm the one with the*

*\* smaller node ID.*

*\**

*\* Note that this means that eventually a replica migration will occurr*

*\* since slaves that are reachable again always have their FAIL flag*

*\* cleared. At the same time this does not mean that there are no*

*\* race conditions possible (two slaves migrating at the same time), but*

*\* this is extremely unlikely to happen, and harmless. \*/*

candidate **=** myself**;**

di **=** dictGetSafeIterator**(**server**.**cluster**->**nodes**);**

**while((**de **=** dictNext**(**di**))** **!=** **NULL)** **{**

clusterNode **\***node **=** dictGetVal**(**de**);**

int okslaves**;**

*/\* Only iterate over working masters. \*/*

**if** **(**nodeIsSlave**(**node**)** **||** nodeFailed**(**node**))** **continue;**

*/\* If this master never had slaves so far, don't migrate. We want*

*\* to migrate to a master that remained orphaned, not masters that*

*\* were never configured to have slaves. \*/*

**if** **(**node**->**numslaves **==** 0**)** **continue;**

okslaves **=** clusterCountNonFailingSlaves**(**node**);**

**if** **(**okslaves **==** 0 **&&** target **==** **NULL** **&&** node**->**numslots **>** 0**)**

target **=** node**;**

**if** **(**okslaves **==** max\_slaves**)** **{**

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

**if** **(**memcmp**(**node**->**slaves**[**j**]->**name**,**

candidate**->**name**,**

REDIS\_CLUSTER\_NAMELEN**)** **<** 0**)**

**{**

candidate **=** node**->**slaves**[**j**];**

**}**

**}**

**}**

**}**

dictReleaseIterator**(**di**);**

*/\* Step 4: perform the migration if there is a target, and if I'm the*

*\* candidate. \*/*

**if** **(**target **&&** candidate **==** myself**)** **{**

redisLog**(**REDIS\_WARNING**,**"Migrating to orphaned master %.40s"**,**

target**->**name**);**

clusterSetMaster**(**target**);**

**}**

**}**

如果当前集群状态不是REDIS\_CLUSTER\_OK，则直接返回；如果当前从节点没有主节点，则直接返回；

接下来计算，当前从节点的主节点，具有未下线从节点的个数okslaves；如果okslaves小于等于迁移阈值server.cluster\_migration\_barrier，则直接返回；

接下来，开始轮训字典server.cluster->nodes，针对其中的每一个节点node：

如果node节点是从节点，或者处于下线状态，则直接处理下一个节点；如果node节点没有配置从节点，则直接处理下一个节点；

调用clusterCountNonFailingSlaves函数，计算该node节点的未下线主节点数okslaves；如果okslaves为0，并且该node节点的numslots大于0，说明该主节点之前有从节点，但是都下线了，因此找到了一个孤立主节点target；

如果okslaves等于参数max\_slaves，说明该node节点就是具有最多未下线从节点的主节点，因此将当前节点的节点ID，与其所有从节点的节点ID进行比较，如果当前节点的名字更大，则将candidate置为具有更小名字的那个从节点；（其实从这里就可以直接退出返回了）

轮训完所有节点后，如果找到了孤立节点，并且当前节点拥有最小的节点ID，则调用clusterSetMaster，将target置为当前节点的主节点，并开始主从复制流程。

**三：configEpoch冲突问题**

在集群中，负责不同槽位的主节点，具有相同的configEpoch其实是没有问题的，但是有可能因为人为介入的原因或者BUG的问题，导致具有相同configEpoch的主节点都宣称负责相同的槽位，这在分布式系统中是致命的问题；因此，Redis规定集群中的所有节点，必须具有不同的configEpoch。

当某个从节点升级为新主节点时，它会得到一个大于当前所有节点的configEpoch的新configEpoch，所以不会导致具有重复configEpoch的从节点（因为一次选举中，不会有两个从节点同时胜出）。但是在管理员发起的重新分片过程的最后，迁入槽位的节点会自己更新自己的configEpoch，而无需其他节点的同意；或者手动强制故障转移过程，也会导致从节点在无需其他节点同意的情况下更新configEpoch，以上的情况都可能导致出现多个主节点具有相同configEpoch的情况。

因此，就需要一种算法，保证集群中所有节点的configEpoch都不相同。这种算法是这样实现的：当某个主节点收到其他主节点发来的心跳包后，发现包中的configEpoch与自己的configEpoch相同，就会调用clusterHandleConfigEpochCollision函数，解决这种configEpoch冲突的问题。

clusterHandleConfigEpochCollision函数的代码如下：

void clusterHandleConfigEpochCollision**(**clusterNode **\***sender**)** **{**

*/\* Prerequisites: nodes have the same configEpoch and are both masters. \*/*

**if** **(**sender**->**configEpoch **!=** myself**->**configEpoch **||**

**!**nodeIsMaster**(**sender**)** **||** **!**nodeIsMaster**(**myself**))** **return;**

*/\* Don't act if the colliding node has a smaller Node ID. \*/*

**if** **(**memcmp**(**sender**->**name**,**myself**->**name**,**REDIS\_CLUSTER\_NAMELEN**)** **<=** 0**)** **return;**

*/\* Get the next ID available at the best of this node knowledge. \*/*

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

myself**->**configEpoch **=** server**.**cluster**->**currentEpoch**;**

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

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

"WARNING: configEpoch collision with node %.40s."

" configEpoch set to %llu"**,**

sender**->**name**,**

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

**}**

如果发送节点的configEpoch不等于当前节点的configEpoch，或者发送节点不是主节点，或者当前节点不是主节点，则直接返回；

如果相比于当前节点的节点ID，发送节点的节点ID更小，则直接返回；

因此，较小名字的节点能获得更大的configEpoch，接下来首先增加自己的currentEpoch，然后将configEpoch赋值为currentEpoch。

这样，即使有多个节点具有相同的configEpoch，最终，只有具有最大节点ID的节点的configEpoch保持不变，其他节点都会增加自己的configEpoch，而且增加的值会不同，具有最小NODE ID的节点，最终具有最大的configEpoch。

参考：

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