**八：判断实例是否客观下线**

当前哨兵一旦监测到某个主节点实例主观下线之后，就会向其他哨兵发送”is-master-down-by-addr”命令，询问其他哨兵是否也认为该主节点主观下线了。如果有超过quorum个哨兵（包括当前哨兵）反馈，都认为该主节点主观下线了，则当前哨兵就将该主节点实例标记为客观下线。

注意，客观下线的概念只针对主节点实例，而与从节点和哨兵实例无关。

1：发送”is-master-down-by-addr”命令

”is-master-down-by-addr”命令有两个作用：一是询问其他哨兵是否认为某个主节点已经主观下线；二是开始故障迁移时，当前哨兵向其他哨兵实例进行"拉票"，让其选自己为领导节点。

本节只关注该命令的第一个作用，此时，该命令的格式是：

*"SENTINEL is-master-down-by-addr <masterip> <masterport> <sentinel.current\_epoch> \*";*

在哨兵的“主函数”sentinelHandleRedisInstance中，通过调用函数sentinelAskMasterStateToOtherSentinels来向其他哨兵发送”is-master-down-by-addr”命令。该函数的代码如下：

void sentinelAskMasterStateToOtherSentinels**(**sentinelRedisInstance **\***master**,** int flags**)** **{**

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

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

di **=** dictGetIterator**(**master**->**sentinels**);**

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

sentinelRedisInstance **\***ri **=** dictGetVal**(**de**);**

mstime\_t elapsed **=** mstime**()** **-** ri**->**last\_master\_down\_reply\_time**;**

char port**[**32**];**

int retval**;**

*/\* If the master state from other sentinel is too old, we clear it. \*/*

**if** **(**elapsed **>** SENTINEL\_ASK\_PERIOD**\***5**)** **{**

ri**->**flags **&=** **~**SRI\_MASTER\_DOWN**;**

sdsfree**(**ri**->**leader**);**

ri**->**leader **=** **NULL;**

**}**

*/\* Only ask if master is down to other sentinels if:*

*\**

*\* 1) We believe it is down, or there is a failover in progress.*

*\* 2) Sentinel is connected.*

*\* 3) We did not received the info within SENTINEL\_ASK\_PERIOD ms. \*/*

**if** **((**master**->**flags **&** SRI\_S\_DOWN**)** **==** 0**)** **continue;**

**if** **(**ri**->**flags **&** SRI\_DISCONNECTED**)** **continue;**

**if** **(!(**flags **&** SENTINEL\_ASK\_FORCED**)** **&&**

mstime**()** **-** ri**->**last\_master\_down\_reply\_time **<** SENTINEL\_ASK\_PERIOD**)**

**continue;**

*/\* Ask \*/*

ll2string**(**port**,sizeof(**port**),**master**->**addr**->**port**);**

retval **=** redisAsyncCommand**(**ri**->**cc**,**

sentinelReceiveIsMasterDownReply**,** **NULL,**

"SENTINEL is-master-down-by-addr %s %s %llu %s"**,**

master**->**addr**->**ip**,** port**,**

sentinel**.**current\_epoch**,**

**(**master**->**failover\_state **>** SENTINEL\_FAILOVER\_STATE\_NONE**)** **?**

server**.**runid **:** "\*"**);**

**if** **(**retval **==** REDIS\_OK**)** ri**->**pending\_commands**++;**

**}**

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

**}**

在函数中，轮训字典master->sentinels，针对其中的每一个哨兵实例ri：

属性ri->last\_master\_down\_reply\_time表示上次收到该哨兵实例ri对于"SENTINEL IS-MASTER-DOWN-BY-ADDR"命令回复的时间，如果该时间距离当前时间已经超过了5倍的SENTINEL\_ASK\_PERIOD，则清除其对于master的过时的状态记录：将SRI\_MASTER\_DOWN标记从实例标志位中清除；释放实例中的leader属性并置为NULL；

接下来开始向哨兵实例ri发送命令，但是在发送命令之前需要满足一定的条件，这些条件分别是：主节点master已经被标记为主观下线了；该哨兵实例处于连接状态；参数flags中设置了SENTINEL\_ASK\_FORCED标记，或者距离上次收到该哨兵实例的命令回复已超过SENTINEL\_ASK\_PERIOD；

满足以上所有条件之后，调用redisAsyncCommand向ri异步发送命令，命令的回调函数是sentinelReceiveIsMasterDownReply。

2：其他哨兵收到”is-master-down-by-addr”命令后的处理

当哨兵收到其他哨兵发来的”SENTINEL is-master-down-by-addr”命令后，调用函数sentinelCommand进行处理。该函数中处理”is-master-down-by-addr”的部分代码是：

void sentinelCommand**(**redisClient **\***c**)** **{**

**...**

**else** **if** **(!**strcasecmp**(**c**->**argv**[**1**]->**ptr**,**"is-master-down-by-addr"**))** **{**

*/\* SENTINEL IS-MASTER-DOWN-BY-ADDR <ip> <port> <current-epoch> <runid>\*/*

sentinelRedisInstance **\***ri**;**

long long req\_epoch**;**

uint64\_t leader\_epoch **=** 0**;**

char **\***leader **=** **NULL;**

long port**;**

int isdown **=** 0**;**

**if** **(**c**->**argc **!=** 6**)** **goto** numargserr**;**

**if** **(**getLongFromObjectOrReply**(**c**,**c**->**argv**[**3**],&**port**,NULL)** **!=** REDIS\_OK **||**

getLongLongFromObjectOrReply**(**c**,**c**->**argv**[**4**],&**req\_epoch**,NULL)**

**!=** REDIS\_OK**)**

**return;**

ri **=** getSentinelRedisInstanceByAddrAndRunID**(**sentinel**.**masters**,**

c**->**argv**[**2**]->**ptr**,**port**,NULL);**

*/\* It exists? Is actually a master? Is subjectively down? It's down.*

*\* Note: if we are in tilt mode we always reply with "0". \*/*

**if** **(!**sentinel**.**tilt **&&** ri **&&** **(**ri**->**flags **&** SRI\_S\_DOWN**)** **&&**

**(**ri**->**flags **&** SRI\_MASTER**))**

isdown **=** 1**;**

*/\* Vote for the master (or fetch the previous vote) if the request*

*\* includes a runid, otherwise the sender is not seeking for a vote. \*/*

**if** **(**ri **&&** ri**->**flags **&** SRI\_MASTER **&&** strcasecmp**(**c**->**argv**[**5**]->**ptr**,**"\*"**))** **{**

leader **=** sentinelVoteLeader**(**ri**,(**uint64\_t**)**req\_epoch**,**

c**->**argv**[**5**]->**ptr**,**

**&**leader\_epoch**);**

**}**

*/\* Reply with a three-elements multi-bulk reply:*

*\* down state, leader, vote epoch. \*/*

addReplyMultiBulkLen**(**c**,**3**);**

addReply**(**c**,** isdown **?** shared**.**cone **:** shared**.**czero**);**

addReplyBulkCString**(**c**,** leader **?** leader **:** "\*"**);**

addReplyLongLong**(**c**,** **(**long long**)**leader\_epoch**);**

**if** **(**leader**)** sdsfree**(**leader**);**

**}**

**...**

**}**

首先从命令参数中取出master的port，以及req\_epoch。然后根据参数中的master的ip和port信息，调用函数getSentinelRedisInstanceByAddrAndRunID得到主节点实例ri；

如果当前哨兵没有处于TILT模式，并且找到的主节点实例ri确实是主节点，并且该主节点实例已经被标记为主观下线了，则设置isdown为1，否则isdown为0；

如果命令参数中的第5个参数不是"\*"，说明该命令是用于"拉票"的，因此调用函数sentinelVoteLeader进行投票，该函数返回本哨兵所选择的领导节点的运行ID，以及该领导的epoch，也就是leader和leader\_epoch；

最后，回复给哨兵消息，回复消息中包含：isdown，leader和leader\_epoch（如果该命令不是用来"拉票"，则leader字段为"\*"，leader\_epoch为0）；

3：哨兵收到其他哨兵的”is-master-down-by-addr”命令回复信息后的处理

之前在sentinelAskMasterStateToOtherSentinels函数中，发送”is-master-down-by-addr”命令时，设置的回调函数是sentinelReceiveIsMasterDownReply。当收到其他哨兵对于”is-master-down-by-addr”命令的回复信息时，就调用该函数进行处理。该函数的代码如下：

void sentinelReceiveIsMasterDownReply**(**redisAsyncContext **\***c**,** void **\***reply**,** void **\***privdata**)** **{**

sentinelRedisInstance **\***ri **=** c**->**data**;**

redisReply **\***r**;**

REDIS\_NOTUSED**(**privdata**);**

**if** **(**ri**)** ri**->**pending\_commands**--;**

**if** **(!**reply **||** **!**ri**)** **return;**

r **=** reply**;**

*/\* Ignore every error or unexpected reply.*

*\* Note that if the command returns an error for any reason we'll*

*\* end clearing the SRI\_MASTER\_DOWN flag for timeout anyway. \*/*

**if** **(**r**->**type **==** REDIS\_REPLY\_ARRAY **&&** r**->**elements **==** 3 **&&**

r**->**element**[**0**]->**type **==** REDIS\_REPLY\_INTEGER **&&**

r**->**element**[**1**]->**type **==** REDIS\_REPLY\_STRING **&&**

r**->**element**[**2**]->**type **==** REDIS\_REPLY\_INTEGER**)**

**{**

ri**->**last\_master\_down\_reply\_time **=** mstime**();**

**if** **(**r**->**element**[**0**]->**integer **==** 1**)** **{**

ri**->**flags **|=** SRI\_MASTER\_DOWN**;**

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

ri**->**flags **&=** **~**SRI\_MASTER\_DOWN**;**

**}**

**if** **(**strcmp**(**r**->**element**[**1**]->**str**,**"\*"**))** **{**

*/\* If the runid in the reply is not "\*" the Sentinel actually*

*\* replied with a vote. \*/*

sdsfree**(**ri**->**leader**);**

**if** **((**long long**)**ri**->**leader\_epoch **!=** r**->**element**[**2**]->**integer**)**

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

"%s voted for %s %llu"**,** ri**->**name**,**

r**->**element**[**1**]->**str**,**

**(**unsigned long long**)** r**->**element**[**2**]->**integer**);**

ri**->**leader **=** sdsnew**(**r**->**element**[**1**]->**str**);**

ri**->**leader\_epoch **=** r**->**element**[**2**]->**integer**;**

**}**

**}**

**}**

首先，如果回复中的第一个参数值为1，说明发送回复的哨兵也认为主节点实例主观下线了，因此增加SRI\_MASTER\_DOWN标记到该哨兵实例的标志位中；否则，将哨兵实例标志位中的SRI\_MASTER\_DOWN标记清除；

如果回复中的第二个参数不是"\*"，说明发送回复的哨兵返回了其选择的领导节点及其epoch，分别将其选择的领导节点的运行ID和epoch记录到ri->leader和ri->leader\_epoch中；

4：判断实例是否客观下线

在哨兵的“主函数”sentinelHandleRedisInstance中，调用sentinelCheckObjectivelyDown函数检测实例是否客观下线。该函数的代码如下：

void sentinelCheckObjectivelyDown**(**sentinelRedisInstance **\***master**)** **{**

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

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

unsigned int quorum **=** 0**,** odown **=** 0**;**

**if** **(**master**->**flags **&** SRI\_S\_DOWN**)** **{**

*/\* Is down for enough sentinels? \*/*

quorum **=** 1**;** */\* the current sentinel. \*/*

*/\* Count all the other sentinels. \*/*

di **=** dictGetIterator**(**master**->**sentinels**);**

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

sentinelRedisInstance **\***ri **=** dictGetVal**(**de**);**

**if** **(**ri**->**flags **&** SRI\_MASTER\_DOWN**)** quorum**++;**

**}**

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

**if** **(**quorum **>=** master**->**quorum**)** odown **=** 1**;**

**}**

*/\* Set the flag accordingly to the outcome. \*/*

**if** **(**odown**)** **{**

**if** **((**master**->**flags **&** SRI\_O\_DOWN**)** **==** 0**)** **{**

sentinelEvent**(**REDIS\_WARNING**,**"+odown"**,**master**,**"%@ #quorum %d/%d"**,**

quorum**,** master**->**quorum**);**

master**->**flags **|=** SRI\_O\_DOWN**;**

master**->**o\_down\_since\_time **=** mstime**();**

**}**

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

**if** **(**master**->**flags **&** SRI\_O\_DOWN**)** **{**

sentinelEvent**(**REDIS\_WARNING**,**"-odown"**,**master**,**"%@"**);**

master**->**flags **&=** **~**SRI\_O\_DOWN**;**

**}**

**}**

**}**

变量quorum表示认为主节点主观下线的哨兵实例的个数。如果master的标志位中设置了SRI\_S\_DOWN，则将其置为1，表明本哨兵实例认为其主观下线了；然后轮训字典master->sentinels，针对其中的每一个哨兵实例，只要其标志位中设置了SRI\_MASTER\_DOWN标记，说明已经收到过该哨兵对于"IS-MASTER-DOWN-BY-ADDR"命令的回复，并且它也认为该master主观下线了，因此将quorum加1；

轮训完所有哨兵实例之后，如果quorum的值大于等于master->quorum，则认为该主节点客观下线了，置变量odown为1；

如果odown为1，并且主节点之前没有被置为客观下线过，则将SRI\_O\_DOWN标记增加到主节点实例的标志位中，表示该主节点客观下线了；

如果odown为0，并且主节点之前已经被置为客观下线了，则将SRI\_O\_DOWN标记从主节点实例的标志位中清除；

**九：故障转移流程之选举领导节点**

1：故障转移流程

当哨兵监测到某个主节点客观下线之后，就会开始故障转移流程。具体步骤就是：

a：在所有哨兵中发起一次“选举”，让其他哨兵选择“我”（当前哨兵）为领导节点；

b：如果“我”能赢得大部分的选票，也就是在共有n个哨兵节点的情况下，如果有超过n/2个哨兵都将选票投给了“我”，则“我”就赢得了本界选举，成为领导节点，从而可以继续下面的流程。如果我没有赢得本界选举，则不能进行下面的流程了，而是随机等待一段时间后，开始下一轮选举；

c：“我”赢得选举后，就会从客观下线主节点的所有下属从节点中，按照一定规则选择一个从节点，使其升级为新的主节点；

d：当选中的从节点升级为主节点之后，“我”就会向剩下的从节点发送”SLAVEOF”命令，使它们与新的主节点进行同步；

e：最后，更新新主节点的信息，并通过”PUBLISH”命令，将新主节点的信息传播给其他哨兵。

2：选举领导节点原理

故障转移流程中，最难理解的部分就是选举领导节点的过程。因为多个哨兵实际上是组成了一个分布式系统，它们之间需要相互协作，通过交换信息，最终选出一个领导节点。

sentinel选举的过程，借鉴了分布式系统中的Raft协议。Raft协议是用来解决分布式系统一致性问题的协议，在很长一段时间，Paxos被认为是解决分布式系统一致性的代名词。但是Paxos难于理解，更难以实现。而Raft协议设计的初衷就是容易实现，保证对于普遍的人群都可以十分舒适容易的去理解。

有关Raft算法，可以参考官网https://raft.github.io/中的介绍。如果想要以最快的速度了解Raft算法的基本原理，可以参考这个PPT，非常形象且容易理解：http://thesecretlivesofdata.com/raft/

要理解哨兵的选举过程，关键就在于理解选举纪元(epoch)的概念。所谓的选举纪元，直白的解释就是“第几届选举”。

选举纪元实际上就是一个计数器。当哨兵进程启动时，其选举纪元就被初始化，默认的初始化值为0，不过该值也可以在配置文件中进行配置。

哨兵运行起来之后，哨兵之间通过HELLO消息来交换信息。HELLO消息中，除了有主节点信息之外，还包含哨兵本地的选举纪元值（sentinel.current\_epoch）。当哨兵收到其他哨兵发布的HELLO消息后，解析其中的选举纪元值，如果该值大于“我”本地的选举纪元值，则会用它的选举纪元更新“我”的选举纪元。

因此，同一个监控单位内的所有哨兵，他们的选举纪元最终就会达成一个统一的值，这也就是Raft中，最终一致性的意思。

当哨兵A发现某个主节点客观下线后，它就会发起新一届的选举。第一件事就是将本地的选举纪元加1，这个加1的意思，实际上就是表示“发起新一届选举”。之后，哨兵A就会向其他哨兵发送”is-master-down-by-addr”命令，用于拉票，其中就包含了A的选举纪元。

投票采用先到先得的策略，因此当哨兵B收到A发来的”is-master-down-by-addr”命令之后，得到A的选举纪元，如果其值大于本地的选举纪元，说明本界选举中还没有投过票，则会更新本地的选举纪元，同时把票投给A。

现实当然不会这么简单，分布式系统因为涉及多个机器，就会有各种可能的情况发生。比如哨兵C几乎同时也发起了新一届的选举，它也会把本地的选举纪元加1，并发送”is-master-down-by-addr”命令。当B收到C发来的命令之后，得到C的选举纪元，发现其值并不大于本地的选举纪元（因为刚才已经根据A的选举纪元更新了），因此就不会再次投票了，而是将之前投票给A的结果反馈给C。

通过上面的介绍可知，在同一届选举（同一个选举纪元的值）中，每个哨兵只会投一次票。因此，在一界选举中，只可能有一个哨兵能获得超过半数的投票，从而赢得选举。

当然，也有可能产生选举失败的情况。也就是没有一个哨兵能获得超过半数的投票。比如有4个哨兵节点A、B、C、D。哨兵A和C几乎同时发起了新的选举，最终B和C将选票投给了A，而A和D将选票投给了C。因此，A和C都只得到了2票，没有超过半数，因此都不能成为新的领导节点。这种情况下，A和C都会随机等待一段时间之后，重新发起新的选举。这种随机性能减少下一轮选举的冲突，从而降低选举失败的可能。

3：判断是否开始故障转移

在哨兵的“主函数”sentinelHandleRedisInstance中，调用sentinelStartFailoverIfNeeded函数，判断是否开始一次新的故障转移流程。该函数的代码如下：

int sentinelStartFailoverIfNeeded**(**sentinelRedisInstance **\***master**)** **{**

*/\* We can't failover if the master is not in O\_DOWN state. \*/*

**if** **(!(**master**->**flags **&** SRI\_O\_DOWN**))** **return** 0**;**

*/\* Failover already in progress? \*/*

**if** **(**master**->**flags **&** SRI\_FAILOVER\_IN\_PROGRESS**)** **return** 0**;**

*/\* Last failover attempt started too little time ago? \*/*

**if** **(**mstime**()** **-** master**->**failover\_start\_time **<**

master**->**failover\_timeout**\***2**)**

**{**

**if** **(**master**->**failover\_delay\_logged **!=** master**->**failover\_start\_time**)** **{**

time\_t clock **=** **(**master**->**failover\_start\_time **+**

master**->**failover\_timeout**\***2**)** **/** 1000**;**

char ctimebuf**[**26**];**

ctime\_r**(&**clock**,**ctimebuf**);**

ctimebuf**[**24**]** **=** '\0'**;** */\* Remove newline. \*/*

master**->**failover\_delay\_logged **=** master**->**failover\_start\_time**;**

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

"Next failover delay: I will not start a failover before %s"**,**

ctimebuf**);**

**}**

**return** 0**;**

**}**

sentinelStartFailover**(**master**);**

**return** 1**;**

**}**

是否能开始一次新的故障转移流程，需要满足下面三个条件：

a：主节点master被标记为客观下线了；

b：当前没有针对该master进行故障转移流程；

c：最重要的条件是，针对该master，当前时间与master->failover\_start\_time之间的时间差，已经超过了master->failover\_timeout\*2。也就是说，当前距离上次进行故障转移流程的开始时间，或者是距离上次投票给其他哨兵的时间，已经等待了足够长的时间；

当创建实例时，master->failover\_start\_time属性值为0，这样第一次进行故障转移时就可以立即开始。

该属性会在两个地方更新，一个是开始一次新的故障转移流程时；一个是当前哨兵收到其他哨兵发来的用于拉票的”is-master-down-by-addr”命令，并且当前哨兵把票投给了其他哨兵，而不是自己时。

更新该属性的方法是master->failover\_start\_time=mstime()+rand()%1000，因此该属性中具有随机性，这就相当于将下次故障转移开始的时间随机化，从而可以减少冲突的发生（比如两个哨兵针对同一个主节点，同时开始进行故障转移，但是因为都没有获得足够的选票。因此这两个哨兵会等待一段时间后再次进行故障转移流程，因此master->failover\_start\_time属性的随机化，实际上就是等待时间的随机化）；

而且，该属性还能防止当哨兵A已经开始故障转移时，另一个哨兵B开始针对同一个主节点进行故障转移（因为哨兵B收到了A的"拉票"命令，并且B把票投给了A，因此，B中会更新master->failover\_start\_time的值，因此B在开始故障转移时，会等待足够长的时间）；

如果不满足以上任何一个条件，则返回0。如果满足以上条件的情况下，则调用sentinelStartFailover函数，开始故障转移流程，然后返回1。

4：开始新一轮的故障转移流程

在sentinelStartFailoverIfNeeded函数中，一旦满足条件后，就会调用函数sentinelStartFailover，开始新一轮的故障转移流程。sentinelStartFailover函数的代码如下：

void sentinelStartFailover**(**sentinelRedisInstance **\***master**)** **{**

redisAssert**(**master**->**flags **&** SRI\_MASTER**);**

master**->**failover\_state **=** SENTINEL\_FAILOVER\_STATE\_WAIT\_START**;**

master**->**flags **|=** SRI\_FAILOVER\_IN\_PROGRESS**;**

master**->**failover\_epoch **=** **++**sentinel**.**current\_epoch**;**

sentinelEvent**(**REDIS\_WARNING**,**"+new-epoch"**,**master**,**"%llu"**,**

**(**unsigned long long**)** sentinel**.**current\_epoch**);**

sentinelEvent**(**REDIS\_WARNING**,**"+try-failover"**,**master**,**"%@"**);**

master**->**failover\_start\_time **=** mstime**()+**rand**()%**SENTINEL\_MAX\_DESYNC**;**

master**->**failover\_state\_change\_time **=** mstime**();**

**}**

该函数实际上就是修改主节点实例的一些状态：

将主节点的master->failover\_state属性置为SENTINEL\_FAILOVER\_STATE\_WAIT\_START，这是故障转移流程的第一个状态；

将SRI\_FAILOVER\_IN\_PROGRESS标记增加到主节点标志位中，表示该主节点进入故障转移流程；

将选举纪元sentinel.current\_epoch加1，并赋值给master->failover\_epoch，表示马上开始新一轮的选举；

将master->failover\_start\_time属性设置为当前时间加上一个1000（1s）内的随机数；将master->failover\_state\_change\_time置为当前时间戳；

5：发送”is-master-down-by-addr”命令进行拉票

在哨兵的“主函数”sentinelHandleRedisInstance中，sentinelStartFailoverIfNeeded函数返回1，表示开始了一次新的故障转移流程。接下来就会调用函数sentinelAskMasterStateToOtherSentinels(ri,SENTINEL\_ASK\_FORCED)，向所有哨兵发送”is-master-down-by-addr”命令进行拉票，请求其他哨兵投票给自己。

sentinelAskMasterStateToOtherSentinels函数的代码，之前已经讲过，不再赘述。这里只需要知道，用于拉票的”is-master-down-by-addr”命令格式是：

*"SENTINEL is-master-down-by-addr <masterip> <masterport> <sentinel.current\_epoch> <server.runid>";*

其中的sentinel.current\_epoch，就是当前哨兵的选举纪元。

6：其他哨兵收到”is-master-down-by-addr”命令后进行投票

当哨兵收到其他哨兵发来的”SENTINEL is-master-down-by-addr”命令后，调用函数sentinelCommand进行处理。该函数中处理”is-master-down-by-addr”的部分代码之前已经讲过，不再赘述，这里需要注意的是，在这部分代码中，调用sentinelVoteLeader函数进行投票。

sentinelVoteLeader函数的代码如下：

char **\***sentinelVoteLeader**(**sentinelRedisInstance **\***master**,** uint64\_t req\_epoch**,** char **\***req\_runid**,** uint64\_t **\***leader\_epoch**)** **{**

**if** **(**req\_epoch **>** sentinel**.**current\_epoch**)** **{**

sentinel**.**current\_epoch **=** req\_epoch**;**

sentinelFlushConfig**();**

sentinelEvent**(**REDIS\_WARNING**,**"+new-epoch"**,**master**,**"%llu"**,**

**(**unsigned long long**)** sentinel**.**current\_epoch**);**

**}**

**if** **(**master**->**leader\_epoch **<** req\_epoch **&&** sentinel**.**current\_epoch **<=** req\_epoch**)**

**{**

sdsfree**(**master**->**leader**);**

master**->**leader **=** sdsnew**(**req\_runid**);**

master**->**leader\_epoch **=** sentinel**.**current\_epoch**;**

sentinelFlushConfig**();**

sentinelEvent**(**REDIS\_WARNING**,**"+vote-for-leader"**,**master**,**"%s %llu"**,**

master**->**leader**,** **(**unsigned long long**)** master**->**leader\_epoch**);**

*/\* If we did not voted for ourselves, set the master failover start*

*\* time to now, in order to force a delay before we can start a*

*\* failover for the same master. \*/*

**if** **(**strcasecmp**(**master**->**leader**,**server**.**runid**))**

master**->**failover\_start\_time **=** mstime**()+**rand**()%**SENTINEL\_MAX\_DESYNC**;**

**}**

**\***leader\_epoch **=** master**->**leader\_epoch**;**

**return** master**->**leader **?** sdsnew**(**master**->**leader**)** **:** **NULL;**

**}**

哨兵调用本函数进行投票选举领导节点。参数master表示要进行故障转移的主节点；req\_epoch表示选举纪元，也就是"第几届选举"；req\_runid表示进行拉票的哨兵实例的运行ID；leader\_epoch是输出参数，返回当前哨兵最新投票的选举纪元。该函数返回当前哨兵最新一次投票选择的领导节点的运行ID；

首先如果req\_epoch大于当前哨兵的当前选举纪元，则将当前哨兵的sentinel.current\_epoch属性更新为req\_epoch；

然后，如果master->leader\_epoch小于req\_epoch，并且sentinel.current\_epoch小于等于req\_epoch的话，说明当前哨兵实例，针对第req\_epoch界选举，尚未投票。因此可以将选票投给req\_runid所表示的哨兵。因此，这种情况下，将master->leader更新为req\_runid，并且将master->leader\_epoch赋值为sentinel.current\_epoch，表示对于主节点master，当前哨兵最新的一次投票投给了master->leader，并且将本次投票的选举纪元记录到master->leader\_epoch中；

这里，如果”我"选择的领导节点不是我自己，则更新master->failover\_start\_time属性为当前时间加1s内的随机时间，这样，针对同一个主节点，可以推迟"我"进行故障转移的时间；

最后，将leader\_epoch赋值为master->leader\_epoch，并且返回master->leader的值。

7：哨兵收到其他哨兵的”is-master-down-by-addr”命令回复信息后的处理

当收到其他哨兵对于”is-master-down-by-addr”命令的回复信息时，哨兵调用函数sentinelReceiveIsMasterDownReply进行处理。该函数之前已经介绍过了，不再赘述。只需要知道，当收到回复后，会把其他哨兵的投票结果记录到哨兵实例的leader和leader\_epoch属性中。

8：统计投票

当故障转移流程处于SENTINEL\_FAILOVER\_STATE\_WAIT\_START状态时，会调用sentinelFailoverWaitStart函数进行处理，而在该函数中，第一件事就是调用sentinelGetLeader函数，统计本界选举的投票结果。

sentinelGetLeader函数的代码如下：

char **\***sentinelGetLeader**(**sentinelRedisInstance **\***master**,** uint64\_t epoch**)** **{**

dict **\***counters**;**

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

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

unsigned int voters **=** 0**,** voters\_quorum**;**

char **\***myvote**;**

char **\***winner **=** **NULL;**

uint64\_t leader\_epoch**;**

uint64\_t max\_votes **=** 0**;**

redisAssert**(**master**->**flags **&** **(**SRI\_O\_DOWN**|**SRI\_FAILOVER\_IN\_PROGRESS**));**

counters **=** dictCreate**(&**leaderVotesDictType**,NULL);**

voters **=** dictSize**(**master**->**sentinels**)+**1**;** */\* All the other sentinels and me. \*/*

*/\* Count other sentinels votes \*/*

di **=** dictGetIterator**(**master**->**sentinels**);**

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

sentinelRedisInstance **\***ri **=** dictGetVal**(**de**);**

**if** **(**ri**->**leader **!=** **NULL** **&&** ri**->**leader\_epoch **==** sentinel**.**current\_epoch**)**

sentinelLeaderIncr**(**counters**,**ri**->**leader**);**

**}**

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

*/\* Check what's the winner. For the winner to win, it needs two conditions:*

*\* 1) Absolute majority between voters (50% + 1).*

*\* 2) And anyway at least master->quorum votes. \*/*

di **=** dictGetIterator**(**counters**);**

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

uint64\_t votes **=** dictGetUnsignedIntegerVal**(**de**);**

**if** **(**votes **>** max\_votes**)** **{**

max\_votes **=** votes**;**

winner **=** dictGetKey**(**de**);**

**}**

**}**

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

*/\* Count this Sentinel vote:*

*\* if this Sentinel did not voted yet, either vote for the most*

*\* common voted sentinel, or for itself if no vote exists at all. \*/*

**if** **(**winner**)**

myvote **=** sentinelVoteLeader**(**master**,**epoch**,**winner**,&**leader\_epoch**);**

**else**

myvote **=** sentinelVoteLeader**(**master**,**epoch**,**server**.**runid**,&**leader\_epoch**);**

**if** **(**myvote **&&** leader\_epoch **==** epoch**)** **{**

uint64\_t votes **=** sentinelLeaderIncr**(**counters**,**myvote**);**

**if** **(**votes **>** max\_votes**)** **{**

max\_votes **=** votes**;**

winner **=** myvote**;**

**}**

**}**

voters\_quorum **=** voters**/**2**+**1**;**

**if** **(**winner **&&** **(**max\_votes **<** voters\_quorum **||** max\_votes **<** master**->**quorum**))**

winner **=** **NULL;**

winner **=** winner **?** sdsnew**(**winner**)** **:** **NULL;**

sdsfree**(**myvote**);**

dictRelease**(**counters**);**

**return** winner**;**

**}**

本函数用于得到：针对master主节点，选举纪元为epoch的选举结果。如果已经有某个哨兵实例赢得了超过半数的选票，则返回该实例的运行ID，否则，返回NULL；

首先创建字典counters，它用于统计每个哨兵实例的选票。它以哨兵的运行ID为key，以得到的选票数为value；然后取值voters为监控master主节点的所有哨兵个数，包括"我"自己；

接下来轮训字典master->sentinels，针对其中的每一个哨兵实例，如果其leader属性不为空，并且其leader\_epoch属性等于当前选举纪元的话，说明该哨兵实例在本界选举中将选票投给了ri->leader。因此，在字典counters中增加ri->leader的选票数；

轮训完所有哨兵实例后，开始轮训字典counters进行"唱票"，最终得到获得票数最多的哨兵实例winner，以及其获得的票数max\_votes；

接下来是统计"我"的选票。如果得到winner的话，则调用sentinelVoteLeader：如果在选举纪元epoch中，"我"之前还没有投过票，则"我"也投给winner；如果"我"之前已经投过票了，则返回"我"选择的领导节点。

类似的，如果winner为NULL，说明其他哨兵没有投过选票，则调用函数sentinelVoteLeader：如果在选举纪元epoch中，"我"之前还没有投过票，则"我"将票投给我自己；如果"我"之前已经投过票了，则返回"我"选择的领导节点。

不管"我"之前有没有投过票，函数sentinelVoteLeader的返回值myvote，都是"我"所选择的领导节点，leader\_epoch都是"我"投票时的选举纪元；如果sentinelVoteLeader返回的选举纪元leader\_epoch就是当前纪元的话，则增加myvote的选票，并且更新winner及其票数max\_votes；

要想真正赢得选举，winner必须得到超过半数的哨兵的支持，也就是其票数必须大于等于voters/2+1；而且其票数还必须大于等于master->quorum；

满足以上条件的话，winner就是选举纪元为epoch时，最终选出的领导节点，因此返回winner；不满足以上条件，说明选举纪元为epoch时，还没有人赢得选举，因此返回NULL。

参考：

https://github.com/maemual/raft-zh\_cn/blob/master/raft-zh\_cn.md

http://weizijun.cn/2015/04/30/Raft%E5%8D%8F%E8%AE%AE%E5%AE%9E%E6%88%98%E4%B9%8BRedis%20Sentinel%E7%9A%84%E9%80%89%E4%B8%BELeader%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90/