**十：故障转移流程中的状态转换**

当哨兵针对某个主节点进行故障转移时，该主节点的故障转移状态master->failover\_state，要依次经历下面六个状态：

SENTINEL\_FAILOVER\_STATE\_WAIT\_START

SENTINEL\_FAILOVER\_STATE\_SELECT\_SLAVE

SENTINEL\_FAILOVER\_STATE\_SEND\_SLAVEOF\_NOONE

SENTINEL\_FAILOVER\_STATE\_WAIT\_PROMOTION

SENTINEL\_FAILOVER\_STATE\_RECONF\_SLAVES

SENTINEL\_FAILOVER\_STATE\_UPDATE\_CONFIG

在哨兵的“主函数”sentinelHandleRedisInstance中，通过sentinelFailoverStateMachine函数进行故障转移状态的转换。它的代码如下：

void sentinelFailoverStateMachine**(**sentinelRedisInstance **\***ri**)** **{**

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

**if** **(!(**ri**->**flags **&** SRI\_FAILOVER\_IN\_PROGRESS**))** **return;**

**switch(**ri**->**failover\_state**)** **{**

**case** SENTINEL\_FAILOVER\_STATE\_WAIT\_START**:**

sentinelFailoverWaitStart**(**ri**);**

**break;**

**case** SENTINEL\_FAILOVER\_STATE\_SELECT\_SLAVE**:**

sentinelFailoverSelectSlave**(**ri**);**

**break;**

**case** SENTINEL\_FAILOVER\_STATE\_SEND\_SLAVEOF\_NOONE**:**

sentinelFailoverSendSlaveOfNoOne**(**ri**);**

**break;**

**case** SENTINEL\_FAILOVER\_STATE\_WAIT\_PROMOTION**:**

sentinelFailoverWaitPromotion**(**ri**);**

**break;**

**case** SENTINEL\_FAILOVER\_STATE\_RECONF\_SLAVES**:**

sentinelFailoverReconfNextSlave**(**ri**);**

**break;**

**}**

**}**

下面分别讲解每个状态及其处理函数。

1：SENTINEL\_FAILOVER\_STATE\_WAIT\_START

上一章讲过，在哨兵的“主函数”sentinelHandleRedisInstance中，调用sentinelStartFailoverIfNeeded函数判断是否可以开始一次故障转移流程。当条件满足后，就会调用sentinelStartFailover函数，开始新一轮的故障转移流程。在该函数中，就会将该主节点的故障转移状态置为SENTINEL\_FAILOVER\_STATE\_WAIT\_START。

一旦哨兵开始一次故障转移流程时，该哨兵第一件事就是向其他所有哨兵发送”is-master-down-by-addr”命令进行拉票。然后就是调用sentinelFailoverWaitStart函数处理当前状态。

sentinelFailoverWaitStart函数的代码如下：

void sentinelFailoverWaitStart**(**sentinelRedisInstance **\***ri**)** **{**

char **\***leader**;**

int isleader**;**

*/\* Check if we are the leader for the failover epoch. \*/*

leader **=** sentinelGetLeader**(**ri**,** ri**->**failover\_epoch**);**

isleader **=** leader **&&** strcasecmp**(**leader**,**server**.**runid**)** **==** 0**;**

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

*/\* If I'm not the leader, and it is not a forced failover via*

*\* SENTINEL FAILOVER, then I can't continue with the failover. \*/*

**if** **(!**isleader **&&** **!(**ri**->**flags **&** SRI\_FORCE\_FAILOVER**))** **{**

int election\_timeout **=** SENTINEL\_ELECTION\_TIMEOUT**;**

*/\* The election timeout is the MIN between SENTINEL\_ELECTION\_TIMEOUT*

*\* and the configured failover timeout. \*/*

**if** **(**election\_timeout **>** ri**->**failover\_timeout**)**

election\_timeout **=** ri**->**failover\_timeout**;**

*/\* Abort the failover if I'm not the leader after some time. \*/*

**if** **(**mstime**()** **-** ri**->**failover\_start\_time **>** election\_timeout**)** **{**

sentinelEvent**(**REDIS\_WARNING**,**"-failover-abort-not-elected"**,**ri**,**"%@"**);**

sentinelAbortFailover**(**ri**);**

**}**

**return;**

**}**

sentinelEvent**(**REDIS\_WARNING**,**"+elected-leader"**,**ri**,**"%@"**);**

ri**->**failover\_state **=** SENTINEL\_FAILOVER\_STATE\_SELECT\_SLAVE**;**

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

sentinelEvent**(**REDIS\_WARNING**,**"+failover-state-select-slave"**,**ri**,**"%@"**);**

**}**

当前哨兵，在调用sentinelStartFailover函数发起故障转移流程时，会将当前选举纪元sentinel.current\_epoch记录到ri->failover\_epoch中。因此，本函数首先根据ri->failover\_epoch，调用函数sentinelGetLeader得到本界选举的结果leader。如果本界选举尚无人获得超过半数的选票，则leader为NULL；

如果当前哨兵还没有赢得选举，并且主节点标志位中没有设置SRI\_FORCE\_FAILOVER标记，说明当前哨兵还没有获得足够的选票，暂时不能继续进行接下来的故障转移流程，需要直接返回。

但是如果超过一定时间之后，当前哨兵还是没有赢得选举，则会终止当前的故障转移流程，因此如果当前距离开始故障转移的时间超过election\_timeout，则调用函数sentinelAbortFailover，终止本次故障转移流程。

如果当前哨兵最终赢得了选举，则更新故障转移的状态，置ri->failover\_state属性为下一个状态：SENTINEL\_FAILOVER\_STATE\_SELECT\_SLAVE，并更新ri->failover\_state\_change为当前时间；

2：SENTINEL\_FAILOVER\_STATE\_SELECT\_SLAVE

当故障转移状态转换为SENTINEL\_FAILOVER\_STATE\_SELECT\_SLAVE时，就需要在下线主节点的所有下属从节点中，按照一定的规则，选择一个从节点使其成为新的主节点。

该状态下的处理函数为sentinelFailoverSelectSlave，该函数的代码如下：

void sentinelFailoverSelectSlave**(**sentinelRedisInstance **\***ri**)** **{**

sentinelRedisInstance **\***slave **=** sentinelSelectSlave**(**ri**);**

*/\* We don't handle the timeout in this state as the function aborts*

*\* the failover or go forward in the next state. \*/*

**if** **(**slave **==** **NULL)** **{**

sentinelEvent**(**REDIS\_WARNING**,**"-failover-abort-no-good-slave"**,**ri**,**"%@"**);**

sentinelAbortFailover**(**ri**);**

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

sentinelEvent**(**REDIS\_WARNING**,**"+selected-slave"**,**slave**,**"%@"**);**

slave**->**flags **|=** SRI\_PROMOTED**;**

ri**->**promoted\_slave **=** slave**;**

ri**->**failover\_state **=** SENTINEL\_FAILOVER\_STATE\_SEND\_SLAVEOF\_NOONE**;**

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

sentinelEvent**(**REDIS\_NOTICE**,**"+failover-state-send-slaveof-noone"**,**

slave**,** "%@"**);**

**}**

**}**

该函数首先调用函数sentinelSelectSlave选择一个符合条件的从节点；

如果没有合适的从节点，则调用sentinelAbortFailover直接终止本次故障转移流程；

如果找到了合适的从节点slave，则首先将标记SRI\_PROMOTED增加到该从节点的标志位中；并使主节点实例的ri->promoted\_slave指针指向该从节点实例，并将故障转移状态置为SENTINEL\_FAILOVER\_STATE\_SEND\_SLAVEOF\_NOONE；然后更新ri->failover\_state\_change\_time为当前时间；

函数sentinelSelectSlave用于在下线主节点的所有从节点实例中，按照一定的规则选择一个从节点。该函数的代码如下：

sentinelRedisInstance **\***sentinelSelectSlave**(**sentinelRedisInstance **\***master**)** **{**

sentinelRedisInstance **\*\***instance **=**

zmalloc**(sizeof(**instance**[**0**])\***dictSize**(**master**->**slaves**));**

sentinelRedisInstance **\***selected **=** **NULL;**

int instances **=** 0**;**

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

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

mstime\_t max\_master\_down\_time **=** 0**;**

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

max\_master\_down\_time **+=** mstime**()** **-** master**->**s\_down\_since\_time**;**

max\_master\_down\_time **+=** master**->**down\_after\_period **\*** 10**;**

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

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

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

mstime\_t info\_validity\_time**;**

**if** **(**slave**->**flags **&** **(**SRI\_S\_DOWN**|**SRI\_O\_DOWN**|**SRI\_DISCONNECTED**))** **continue;**

**if** **(**mstime**()** **-** slave**->**last\_avail\_time **>** SENTINEL\_PING\_PERIOD**\***5**)** **continue;**

**if** **(**slave**->**slave\_priority **==** 0**)** **continue;**

*/\* If the master is in SDOWN state we get INFO for slaves every second.*

*\* Otherwise we get it with the usual period so we need to account for*

*\* a larger delay. \*/*

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

info\_validity\_time **=** SENTINEL\_PING\_PERIOD**\***5**;**

**else**

info\_validity\_time **=** SENTINEL\_INFO\_PERIOD**\***3**;**

**if** **(**mstime**()** **-** slave**->**info\_refresh **>** info\_validity\_time**)** **continue;**

**if** **(**slave**->**master\_link\_down\_time **>** max\_master\_down\_time**)** **continue;**

instance**[**instances**++]** **=** slave**;**

**}**

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

**if** **(**instances**)** **{**

qsort**(**instance**,**instances**,sizeof(**sentinelRedisInstance**\*),**

compareSlavesForPromotion**);**

selected **=** instance**[**0**];**

**}**

zfree**(**instance**);**

**return** selected**;**

**}**

首先创建数组instance，它将用于保存所有状态良好的从节点；

然后计算max\_master\_down\_time，他表示所允许的从节点与主节点断链时间的最大值。它的值是主节点客观下线的时间加上10倍的master->down\_after\_period的值：

接下来，轮训字典master->slaves，针对其中的每一个从节点，判断其状态是否良好。从节点状态良好的条件是：

从节点没有处于主观下线、客观下线或者断链状态；

距离上一次收到该从节点对于"PING"命令的正常回复的时间，不超过5倍的SENTINEL\_PING\_PERIOD；

该从节点的优先级不是0；

距离上一次收到该从节点对于"INFO"命令的回复的时间，不超过3倍或5倍（根据主节点是否客观下线而定）的SENTINEL\_PING\_PERIOD；

从节点与主节点的断链时间（该时间值根据从节点的"INFO"命令回复中得到）不超过max\_master\_down\_time；

满足以上条件的从节点，就认为是状态良好的从节点，将其记录到数组instance中；

所有从节点都轮训完毕之后，使用qsort快速排序算法，对数组instance进行排序。这里使用的比较函数compareSlavesForPromotion；排好序的instance数组，状态越好的从节点，其位置越靠前，因此，返回instance[0]作为选中的从节点；

下面就是快速排序算法中，使用的比较函数compareSlavesForPromotion的代码：

int compareSlavesForPromotion**(**const void **\***a**,** const void **\***b**)** **{**

sentinelRedisInstance **\*\***sa **=** **(**sentinelRedisInstance **\*\*)**a**,**

**\*\***sb **=** **(**sentinelRedisInstance **\*\*)**b**;**

char **\***sa\_runid**,** **\***sb\_runid**;**

**if** **((\***sa**)->**slave\_priority **!=** **(\***sb**)->**slave\_priority**)**

**return** **(\***sa**)->**slave\_priority **-** **(\***sb**)->**slave\_priority**;**

*/\* If priority is the same, select the slave with greater replication*

*\* offset (processed more data frmo the master). \*/*

**if** **((\***sa**)->**slave\_repl\_offset **>** **(\***sb**)->**slave\_repl\_offset**)** **{**

**return** **-**1**;** */\* a < b \*/*

**}** **else** **if** **((\***sa**)->**slave\_repl\_offset **<** **(\***sb**)->**slave\_repl\_offset**)** **{**

**return** 1**;** */\* b > a \*/*

**}**

*/\* If the replication offset is the same select the slave with that has*

*\* the lexicographically smaller runid. Note that we try to handle runid*

*\* == NULL as there are old Redis versions that don't publish runid in*

*\* INFO. A NULL runid is considered bigger than any other runid. \*/*

sa\_runid **=** **(\***sa**)->**runid**;**

sb\_runid **=** **(\***sb**)->**runid**;**

**if** **(**sa\_runid **==** **NULL** **&&** sb\_runid **==** **NULL)** **return** 0**;**

**else** **if** **(**sa\_runid **==** **NULL)** **return** 1**;** */\* a > b \*/*

**else** **if** **(**sb\_runid **==** **NULL)** **return** **-**1**;** */\* a < b \*/*

**return** strcasecmp**(**sa\_runid**,** sb\_runid**);**

**}**

该函数用于比较两个从节点的状态：如果a的状态要好于b，则返回-1，表示a小于b，否则返回0或1，表示a等于或大于b；

首先比较a和b的优先级：优先级越小（0除外），则状态越好；如果a和b的优先级相同，则比较它们的复制偏移量：复制偏移量越大，则状态越好；

如果以上的比较结果都是相同的，则比较a和b的运行ID的字母循序，另外如果某个从节点的运行ID为NULL，则它的状态更差。

3：SENTINEL\_FAILOVER\_STATE\_SEND\_SLAVEOF\_NOONE

当选择好一个从节点之后，接下来在状态为SENTINEL\_FAILOVER\_STATE\_SEND\_SLAVEOF\_NOONE时，要做的就是向该从节点发送”SLAVEOF NO ONE”命令。

该状态下的处理函数为sentinelFailoverSendSlaveOfNoOne，该函数的代码如下：

void sentinelFailoverSendSlaveOfNoOne**(**sentinelRedisInstance **\***ri**)** **{**

int retval**;**

*/\* We can't send the command to the promoted slave if it is now*

*\* disconnected. Retry again and again with this state until the timeout*

*\* is reached, then abort the failover. \*/*

**if** **(**ri**->**promoted\_slave**->**flags **&** SRI\_DISCONNECTED**)** **{**

**if** **(**mstime**()** **-** ri**->**failover\_state\_change\_time **>** ri**->**failover\_timeout**)** **{**

sentinelEvent**(**REDIS\_WARNING**,**"-failover-abort-slave-timeout"**,**ri**,**"%@"**);**

sentinelAbortFailover**(**ri**);**

**}**

**return;**

**}**

*/\* Send SLAVEOF NO ONE command to turn the slave into a master.*

*\* We actually register a generic callback for this command as we don't*

*\* really care about the reply. We check if it worked indirectly observing*

*\* if INFO returns a different role (master instead of slave). \*/*

retval **=** sentinelSendSlaveOf**(**ri**->**promoted\_slave**,NULL,**0**);**

**if** **(**retval **!=** REDIS\_OK**)** **return;**

sentinelEvent**(**REDIS\_NOTICE**,** "+failover-state-wait-promotion"**,**

ri**->**promoted\_slave**,**"%@"**);**

ri**->**failover\_state **=** SENTINEL\_FAILOVER\_STATE\_WAIT\_PROMOTION**;**

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

**}**

代码很简单。首先如果选中的从节点当前处于断链状态，则因无法向其发送命令，因此直接返回；如果该状态已经持续超过ri->failover\_timeout的时间，则调用函数sentinelAbortFailover终止本次故障转移流程；

然后调用sentinelSendSlaveOf函数，向从节点发送"SLAVEOF NO ONE"命令，然后置故障转移状态为SENTINEL\_FAILOVER\_STATE\_WAIT\_PROMOTION，并且更新ri->failover\_state\_change\_time为当前时间；

函数sentinelSendSlaveOf用于发送”SLAVEOF”命令，它的代码如下：

int sentinelSendSlaveOf**(**sentinelRedisInstance **\***ri**,** char **\***host**,** int port**)** **{**

char portstr**[**32**];**

int retval**;**

ll2string**(**portstr**,sizeof(**portstr**),**port**);**

*/\* If host is NULL we send SLAVEOF NO ONE that will turn the instance*

*\* into a master. \*/*

**if** **(**host **==** **NULL)** **{**

host **=** "NO"**;**

memcpy**(**portstr**,**"ONE"**,**4**);**

**}**

*/\* In order to send SLAVEOF in a safe way, we send a transaction performing*

*\* the following tasks:*

*\* 1) Reconfigure the instance according to the specified host/port params.*

*\* 2) Rewrite the configuraiton.*

*\* 3) Disconnect all clients (but this one sending the commnad) in order*

*\* to trigger the ask-master-on-reconnection protocol for connected*

*\* clients.*

*\**

*\* Note that we don't check the replies returned by commands, since we*

*\* will observe instead the effects in the next INFO output. \*/*

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

sentinelDiscardReplyCallback**,** **NULL,** "MULTI"**);**

**if** **(**retval **==** REDIS\_ERR**)** **return** retval**;**

ri**->**pending\_commands**++;**

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

sentinelDiscardReplyCallback**,** **NULL,** "SLAVEOF %s %s"**,** host**,** portstr**);**

**if** **(**retval **==** REDIS\_ERR**)** **return** retval**;**

ri**->**pending\_commands**++;**

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

sentinelDiscardReplyCallback**,** **NULL,** "CONFIG REWRITE"**);**

**if** **(**retval **==** REDIS\_ERR**)** **return** retval**;**

ri**->**pending\_commands**++;**

*/\* CLIENT KILL TYPE <type> is only supported starting from Redis 2.8.12,*

*\* however sending it to an instance not understanding this command is not*

*\* an issue because CLIENT is variadic command, so Redis will not*

*\* recognized as a syntax error, and the transaction will not fail (but*

*\* only the unsupported command will fail). \*/*

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

sentinelDiscardReplyCallback**,** **NULL,** "CLIENT KILL TYPE normal"**);**

**if** **(**retval **==** REDIS\_ERR**)** **return** retval**;**

ri**->**pending\_commands**++;**

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

sentinelDiscardReplyCallback**,** **NULL,** "EXEC"**);**

**if** **(**retval **==** REDIS\_ERR**)** **return** retval**;**

ri**->**pending\_commands**++;**

**return** REDIS\_OK**;**

**}**

如果参数host为NULL，则需要发送"SLAVEOF NO ONE"命令，否则，"SLAVEOF"后跟具体的ip和port信息；

为了安全的发送"SLAVEOF"命令，这里使用事务的方式进行发送。首先发送"MULTI"命令；然后发送"SLAVEOF"命令；然后发送"CONFIG REWRITE"命令，这样从节点会重写配置文件；然后发送"CLIENT KILL TYPE normal"命令，从节点收到该命令后，会断开所有与之连接的normal客户端，包括与所有哨兵的命令连接；最后发送"EXEC"命令；

以上的命令，都不关心它们的回复，而是会在该实例的"INFO"命令回复中判断命令的执行结果；

4：SENTINEL\_FAILOVER\_STATE\_WAIT\_PROMOTION

该状态下的处理函数为sentinelFailoverWaitPromotion，代码如下：

void sentinelFailoverWaitPromotion**(**sentinelRedisInstance **\***ri**)** **{**

*/\* Just handle the timeout. Switching to the next state is handled*

*\* by the function parsing the INFO command of the promoted slave. \*/*

**if** **(**mstime**()** **-** ri**->**failover\_state\_change\_time **>** ri**->**failover\_timeout**)** **{**

sentinelEvent**(**REDIS\_WARNING**,**"-failover-abort-slave-timeout"**,**ri**,**"%@"**);**

sentinelAbortFailover**(**ri**);**

**}**

**}**

本函数中，只是判断处于SENTINEL\_FAILOVER\_STATE\_WAIT\_PROMOTION状态的时间是否超过了阈值ri->failover\_timeout。如果确实已经超过了，则调用函数sentinelAbortFailover终止本次故障转移流程；

从节点执行完"SLAVEOF NO ONE"命令之后，会在其发送的"INFO"命令回复中体现出来。因此相应的状态转换动作也就在"INFO"回复的回调函数sentinelRefreshInstanceInfo中执行。

在sentinelRefreshInstanceInfo中，处理这部分的代码为：

void sentinelRefreshInstanceInfo**(**sentinelRedisInstance **\***ri**,** const char **\***info**)** **{**

**...**

*/\* Handle slave -> master role switch. \*/*

**if** **((**ri**->**flags **&** SRI\_SLAVE**)** **&&** role **==** SRI\_MASTER**)** **{**

*/\* If this is a promoted slave we can change state to the*

*\* failover state machine. \*/*

**if** **((**ri**->**flags **&** SRI\_PROMOTED**)** **&&**

**(**ri**->**master**->**flags **&** SRI\_FAILOVER\_IN\_PROGRESS**)** **&&**

**(**ri**->**master**->**failover\_state **==**

SENTINEL\_FAILOVER\_STATE\_WAIT\_PROMOTION**))**

**{**

*/\* Now that we are sure the slave was reconfigured as a master*

*\* set the master configuration epoch to the epoch we won the*

*\* election to perform this failover. This will force the other*

*\* Sentinels to update their config (assuming there is not*

*\* a newer one already available). \*/*

ri**->**master**->**config\_epoch **=** ri**->**master**->**failover\_epoch**;**

ri**->**master**->**failover\_state **=** SENTINEL\_FAILOVER\_STATE\_RECONF\_SLAVES**;**

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

sentinelFlushConfig**();**

sentinelEvent**(**REDIS\_WARNING**,**"+promoted-slave"**,**ri**,**"%@"**);**

sentinelEvent**(**REDIS\_WARNING**,**"+failover-state-reconf-slaves"**,**

ri**->**master**,**"%@"**);**

sentinelCallClientReconfScript**(**ri**->**master**,**SENTINEL\_LEADER**,**

"start"**,**ri**->**master**->**addr**,**ri**->**addr**);**

sentinelForceHelloUpdateForMaster**(**ri**->**master**);**

**}**

**...**

**}**

**...**

**}**

属性ri->flags表示该实例原来的类型，而role表示该实例在”INFO”命令回复中，报告的自己当前的角色。

如果ri->flags中为从节点，但是role为主节点。这种情况下：如果当前实例确实是哨兵在进行故障转移流程中选中的新主节点，并且目前的故障转移状态为SENTINEL\_FAILOVER\_STATE\_WAIT\_PROMOTION，说明已经向其发送了"SLAVEOF NO ONE"，这里收到该节点的"INFO"回复中，它已经报告自己为主节点，因此"SLAVEOF"命令执行成功了。

因此：更新ri->master中的config\_epoch属性值，更新故障迁移状态为SENTINEL\_FAILOVER\_STATE\_RECONF\_SLAVES，更新failover\_state\_change\_time属性为当前时间；并且更新配置文件，记录日志，发布消息，调用sentinelForceHelloUpdateForMaster函数，强制引发向所有实例节点发送"PUBLISH"命令；

5：SENTINEL\_FAILOVER\_STATE\_RECONF\_SLAVES

当故障转移状态变为SENTINEL\_FAILOVER\_STATE\_RECONF\_SLAVES时，选中的从节点已经升级为主节点，接下来要做的就是向其他从节点发送”SLAVEOF”命令，使它们与新的主节点进行同步。

该状态下的处理函数是sentinelFailoverReconfNextSlave，代码如下：

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

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

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

int in\_progress **=** 0**;**

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

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

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

**if** **(**slave**->**flags **&** **(**SRI\_RECONF\_SENT**|**SRI\_RECONF\_INPROG**))**

in\_progress**++;**

**}**

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

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

**while(**in\_progress **<** master**->**parallel\_syncs **&&**

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

**{**

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

int retval**;**

*/\* Skip the promoted slave, and already configured slaves. \*/*

**if** **(**slave**->**flags **&** **(**SRI\_PROMOTED**|**SRI\_RECONF\_DONE**))** **continue;**

*/\* If too much time elapsed without the slave moving forward to*

*\* the next state, consider it reconfigured even if it is not.*

*\* Sentinels will detect the slave as misconfigured and fix its*

*\* configuration later. \*/*

**if** **((**slave**->**flags **&** SRI\_RECONF\_SENT**)** **&&**

**(**mstime**()** **-** slave**->**slave\_reconf\_sent\_time**)** **>**

SENTINEL\_SLAVE\_RECONF\_TIMEOUT**)**

**{**

sentinelEvent**(**REDIS\_NOTICE**,**"-slave-reconf-sent-timeout"**,**slave**,**"%@"**);**

slave**->**flags **&=** **~**SRI\_RECONF\_SENT**;**

slave**->**flags **|=** SRI\_RECONF\_DONE**;**

**}**

*/\* Nothing to do for instances that are disconnected or already*

*\* in RECONF\_SENT state. \*/*

**if** **(**slave**->**flags **&** **(**SRI\_DISCONNECTED**|**SRI\_RECONF\_SENT**|**SRI\_RECONF\_INPROG**))**

**continue;**

*/\* Send SLAVEOF <new master>. \*/*

retval **=** sentinelSendSlaveOf**(**slave**,**

master**->**promoted\_slave**->**addr**->**ip**,**

master**->**promoted\_slave**->**addr**->**port**);**

**if** **(**retval **==** REDIS\_OK**)** **{**

slave**->**flags **|=** SRI\_RECONF\_SENT**;**

slave**->**slave\_reconf\_sent\_time **=** mstime**();**

sentinelEvent**(**REDIS\_NOTICE**,**"+slave-reconf-sent"**,**slave**,**"%@"**);**

in\_progress**++;**

**}**

**}**

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

*/\* Check if all the slaves are reconfigured and handle timeout. \*/*

sentinelFailoverDetectEnd**(**master**);**

**}**

因为从节点在与主节点进行同步时，有可能无法响应客户端的查询。因此为了避免过多从节点因为同步而无法响应的问题，一个时间段内，最多只能允许master->parallel\_syncs个从节点正在进行同步操作；

因此，首先轮训字典master->slaves，统计当前正在进行同步的从节点之和；只要从节点标志位中设置了SRI\_RECONF\_SENT或者SRI\_RECONF\_INPROG标记，就说明该从节点正在进行同步，将计数器in\_progress加1；

接下来，只要in\_progress还没超过master->parallel\_syncs，就轮训字典master->slaves，向尚未发送过"SLAVEOF"命令的从节点发送该命令。在轮训过程中：

如果该从节点实例的标志位中设置了SRI\_PROMOTED，说明它是"我"选中的新的主节点，因此直接跳过；

如果该从节点实例的标志位中设置了SRI\_RECONF\_DONE，说明该从节点已经完成了同步，因此直接跳过；

如果从节点处于SRI\_RECONF\_SENT状态的时间已经超过了SENTINEL\_SLAVE\_RECONF\_TIMEOUT，则将该从节点的状态直接置为SRI\_RECONF\_DONE，当做其已经完成了同步。后续收到该从节点的"INFO"回复时，如果信息不正确，到时候会采取相应的动作；

如果从节点实例已经断链，则直接跳过；

如果从节点实例的状态为SRI\_RECONF\_SENT或SRI\_RECONF\_INPROG，说明该从节点正在进行同步，直接跳过；

经过以上判断之后，剩下的从节点就是还没有发送过"SLAVEOF"命令的节点，因此调用sentinelSendSlaveOf函数向其发送命令，发送成功之后，将其状态置为SRI\_RECONF\_SENT；

在函数的最后，调用函数sentinelFailoverDetectEnd，检查是否所有从节点实例都已经完成了同步；

在向从节点发送”SLAVEOF”命令之后，该从节点实例的状态会经过SRI\_RECONF\_SENT、SRI\_RECONF\_INPROG和SRI\_RECONF\_DONE这三种状态的转换。

当向从节点发送完”SLAVEOF”命令之后，该从节点实例的状态为SRI\_RECONF\_SENT，剩下的状态转换是根据该从节点发来的”INFO”命令回复中的信息进行判断的。

在收到从节点的”INFO”命令回复的回调函数sentinelRefreshInstanceInfo中，处理这部分的代码如下：

void sentinelRefreshInstanceInfo**(**sentinelRedisInstance **\***ri**,** const char **\***info**)** **{**

**...**

*/\* Detect if the slave that is in the process of being reconfigured*

*\* changed state. \*/*

**if** **((**ri**->**flags **&** SRI\_SLAVE**)** **&&** role **==** SRI\_SLAVE **&&**

**(**ri**->**flags **&** **(**SRI\_RECONF\_SENT**|**SRI\_RECONF\_INPROG**)))**

**{**

*/\* SRI\_RECONF\_SENT -> SRI\_RECONF\_INPROG. \*/*

**if** **((**ri**->**flags **&** SRI\_RECONF\_SENT**)** **&&**

ri**->**slave\_master\_host **&&**

strcmp**(**ri**->**slave\_master\_host**,**

ri**->**master**->**promoted\_slave**->**addr**->**ip**)** **==** 0 **&&**

ri**->**slave\_master\_port **==** ri**->**master**->**promoted\_slave**->**addr**->**port**)**

**{**

ri**->**flags **&=** **~**SRI\_RECONF\_SENT**;**

ri**->**flags **|=** SRI\_RECONF\_INPROG**;**

sentinelEvent**(**REDIS\_NOTICE**,**"+slave-reconf-inprog"**,**ri**,**"%@"**);**

**}**

*/\* SRI\_RECONF\_INPROG -> SRI\_RECONF\_DONE \*/*

**if** **((**ri**->**flags **&** SRI\_RECONF\_INPROG**)** **&&**

ri**->**slave\_master\_link\_status **==** SENTINEL\_MASTER\_LINK\_STATUS\_UP**)**

**{**

ri**->**flags **&=** **~**SRI\_RECONF\_INPROG**;**

ri**->**flags **|=** SRI\_RECONF\_DONE**;**

sentinelEvent**(**REDIS\_NOTICE**,**"+slave-reconf-done"**,**ri**,**"%@"**);**

**}**

**}**

**}**

如果该从节点标志位中设置了SRI\_RECONF\_SENT标记，并且它的"INFO"回复中"master\_host:"和"master\_port:"的信息与新主节点的ip和port相同，则将从节点标志中的SRI\_RECONF\_SENT标记清除，并增加SRI\_RECONF\_INPROG标记；

如果该从节点的标志位中设置了SRI\_RECONF\_INPROG标记，并且它的"INFO"回复中的"master\_link\_status:"的信息为"up"，则说明该从节点已经完成了与新主节点间的同步，因此，将将从节点标志中的SRI\_RECONF\_INPROG标记清除，并增加SRI\_RECONF\_DONE标记。

在函数sentinelFailoverReconfNextSlave的最后，会调用函数sentinelFailoverDetectEnd，检查是否所有从节点实例都已经完成了同步。该函数的代码如下：

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

int not\_reconfigured **=** 0**,** timeout **=** 0**;**

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

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

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

*/\* We can't consider failover finished if the promoted slave is*

*\* not reachable. \*/*

**if** **(**master**->**promoted\_slave **==** **NULL** **||**

master**->**promoted\_slave**->**flags **&** SRI\_S\_DOWN**)** **return;**

*/\* The failover terminates once all the reachable slaves are properly*

*\* configured. \*/*

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

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

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

**if** **(**slave**->**flags **&** **(**SRI\_PROMOTED**|**SRI\_RECONF\_DONE**))** **continue;**

**if** **(**slave**->**flags **&** SRI\_S\_DOWN**)** **continue;**

not\_reconfigured**++;**

**}**

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

*/\* Force end of failover on timeout. \*/*

**if** **(**elapsed **>** master**->**failover\_timeout**)** **{**

not\_reconfigured **=** 0**;**

timeout **=** 1**;**

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

**}**

**if** **(**not\_reconfigured **==** 0**)** **{**

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

master**->**failover\_state **=** SENTINEL\_FAILOVER\_STATE\_UPDATE\_CONFIG**;**

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

**}**

*/\* If I'm the leader it is a good idea to send a best effort SLAVEOF*

*\* command to all the slaves still not reconfigured to replicate with*

*\* the new master. \*/*

**if** **(**timeout**)** **{**

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

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

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

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

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

int retval**;**

**if** **(**slave**->**flags **&**

**(**SRI\_RECONF\_DONE**|**SRI\_RECONF\_SENT**|**SRI\_DISCONNECTED**))** **continue;**

retval **=** sentinelSendSlaveOf**(**slave**,**

master**->**promoted\_slave**->**addr**->**ip**,**

master**->**promoted\_slave**->**addr**->**port**);**

**if** **(**retval **==** REDIS\_OK**)** **{**

sentinelEvent**(**REDIS\_NOTICE**,**"+slave-reconf-sent-be"**,**slave**,**"%@"**);**

slave**->**flags **|=** SRI\_RECONF\_SENT**;**

**}**

**}**

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

**}**

**}**

首先，如果"我"选中的新主节点目前处于主观下线的状态，则直接返回；

接下来，轮训字典master->slaves，查看当前尚未完成同步的从节点的个数not\_reconfigured：如果该从节点的标志位中还没有设置SRI\_RECONF\_DONE标记，则表示它还没有完成同步操作；

如果故障转移流程处于当前状态的时间，已经超过了master->failover\_timeout的时间，则将not\_reconfigured置为0，表示接下来会强制进入下一状态；并且置timeout为1，表示接下来会重新发送一次"SLAVEOF"命令；

接下来，如果not\_reconfigured为0，要么表示所有从节点已经完成了与新主节点间的同步，要么表示超时了。不管哪种情况，都将故障转移状态置为SENTINEL\_FAILOVER\_STATE\_UPDATE\_CONFIG，表示进入故障转移流程的最后状态；

接下来，如果timeout为1，表示发生了超时。向所有未完成同步的从节点发送一次"SLAVEOF"命令：轮训字典master->slaves，只要从节点标志位中没有设置SRI\_RECONF\_DONE，SRI\_RECONF\_SENT或SRI\_DISCONNECTED标记，就调用sentinelSendSlaveOf函数重新向从节点发送一次"SLAVEOF"命令；

6：SENTINEL\_FAILOVER\_STATE\_UPDATE\_CONFIG

故障转移流程的最后一个状态，就是要更新当前哨兵节点中的主节点实例，及其下属从节点实例的信息。

需要注意的是，该状态的处理并非在sentinelFailoverStateMachine函数中完成的。而是在sentinelHandleDictOfRedisInstances函数中，轮训完所有实例之后，一旦发现某个主节点的故障转移状态为SENTINEL\_FAILOVER\_STATE\_UPDATE\_CONFIG，则调用函数sentinelFailoverSwitchToPromotedSlave进行处理。

sentinelFailoverSwitchToPromotedSlave函数的代码很简单，就是调用函数sentinelResetMasterAndChangeAddress，将主节点的信息更新为选中的从节点的信息。sentinelResetMasterAndChangeAddress函数的代码如下：

int sentinelResetMasterAndChangeAddress**(**sentinelRedisInstance **\***master**,** char **\***ip**,** int port**)** **{**

sentinelAddr **\***oldaddr**,** **\***newaddr**;**

sentinelAddr **\*\***slaves **=** **NULL;**

int numslaves **=** 0**,** j**;**

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

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

newaddr **=** createSentinelAddr**(**ip**,**port**);**

**if** **(**newaddr **==** **NULL)** **return** REDIS\_ERR**;**

*/\* Make a list of slaves to add back after the reset.*

*\* Don't include the one having the address we are switching to. \*/*

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

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

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

**if** **(**sentinelAddrIsEqual**(**slave**->**addr**,**newaddr**))** **continue;**

slaves **=** zrealloc**(**slaves**,sizeof(**sentinelAddr**\*)\*(**numslaves**+**1**));**

slaves**[**numslaves**++]** **=** createSentinelAddr**(**slave**->**addr**->**ip**,**

slave**->**addr**->**port**);**

**}**

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

*/\* If we are switching to a different address, include the old address*

*\* as a slave as well, so that we'll be able to sense / reconfigure*

*\* the old master. \*/*

**if** **(!**sentinelAddrIsEqual**(**newaddr**,**master**->**addr**))** **{**

slaves **=** zrealloc**(**slaves**,sizeof(**sentinelAddr**\*)\*(**numslaves**+**1**));**

slaves**[**numslaves**++]** **=** createSentinelAddr**(**master**->**addr**->**ip**,**

master**->**addr**->**port**);**

**}**

*/\* Reset and switch address. \*/*

sentinelResetMaster**(**master**,**SENTINEL\_RESET\_NO\_SENTINELS**);**

oldaddr **=** master**->**addr**;**

master**->**addr **=** newaddr**;**

master**->**o\_down\_since\_time **=** 0**;**

master**->**s\_down\_since\_time **=** 0**;**

*/\* Add slaves back. \*/*

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

sentinelRedisInstance **\***slave**;**

slave **=** createSentinelRedisInstance**(NULL,**SRI\_SLAVE**,**slaves**[**j**]->**ip**,**

slaves**[**j**]->**port**,** master**->**quorum**,** master**);**

releaseSentinelAddr**(**slaves**[**j**]);**

**if** **(**slave**)** sentinelEvent**(**REDIS\_NOTICE**,**"+slave"**,**slave**,**"%@"**);**

**}**

zfree**(**slaves**);**

*/\* Release the old address at the end so we are safe even if the function*

*\* gets the master->addr->ip and master->addr->port as arguments. \*/*

releaseSentinelAddr**(**oldaddr**);**

sentinelFlushConfig**();**

**return** REDIS\_OK**;**

**}**

因为某个从节点实例升级为主节点了。因此首先遍历字典master->slaves，根据其中的每一个从节点实例，只要它的ip或port与新主节点的ip或port不同，就将其ip和port记录到数组slaves中；

并且，当前主节点的ip和port与新的主节点的ip和port不同的情况下，也把当前主节点的地址记录到数组slaves中（因为该主节点后续上线后，会转换成从节点）；

然后，调用sentinelResetMaster函数，重置主节点实例的信息，比如释放并重建从节点字典ri->slaves；断开异步上下文cc和pc上的连接；重置实例结构中的各个属性等；

最后，轮训数组slaves，根据其中记录的每一个ip和port信息，创建从节点实例，增加到字典master->slaves中；

另外，如果哨兵A收到其他哨兵发布的HELLO消息后，发现HELLO消息中的主节点信息，与本地的不一致。说明其他哨兵刚刚完成了一次故障转移流程，并升级了某个从节点使其成为了新的主节点。因此，哨兵A也会调用sentinelResetMasterAndChangeAddress函数，重置主节点信息。

最后，当前处于下线状态的旧的主节点B，已经被放到新的主节点的master->slaves字典中了。因此哨兵会不断尝试向其建链。一旦B恢复上线后，哨兵与其的命令连接和订阅连接就会建立。在向其发送”INFO”命令，并得到其回复后，就会发现它的角色还是主节点，因此需要向其发送”SLAVEOF”命令，使其成为从节点。

这是在收到”INFO”命令回复的回调函数sentinelRefreshInstanceInfo中进行处理的。这部分的代码如下：

void sentinelRefreshInstanceInfo**(**sentinelRedisInstance **\***ri**,** const char **\***info**)** **{**

**...**

**if** **((**ri**->**flags **&** SRI\_SLAVE**)** **&&** role **==** SRI\_MASTER**)** **{**

*/\* If this is a promoted slave we can change state to the*

*\* failover state machine. \*/*

**if** **((**ri**->**flags **&** SRI\_PROMOTED**)** **&&**

**(**ri**->**master**->**flags **&** SRI\_FAILOVER\_IN\_PROGRESS**)** **&&**

**(**ri**->**master**->**failover\_state **==**

SENTINEL\_FAILOVER\_STATE\_WAIT\_PROMOTION**))**

**{**

**...**

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

*/\* A slave turned into a master. We want to force our view and*

*\* reconfigure as slave. Wait some time after the change before*

*\* going forward, to receive new configs if any. \*/*

mstime\_t wait\_time **=** SENTINEL\_PUBLISH\_PERIOD**\***4**;**

**if** **(!(**ri**->**flags **&** SRI\_PROMOTED**)** **&&**

sentinelMasterLooksSane**(**ri**->**master**)** **&&**

sentinelRedisInstanceNoDownFor**(**ri**,**wait\_time**)** **&&**

mstime**()** **-** ri**->**role\_reported\_time **>** wait\_time**)**

**{**

redisLog**(**REDIS\_WARNING**,** "[%s]%s report it is master, send SLAVEOF %s %d"**,**

\_\_func\_\_**,** getinstanceinfo**(**ri**),** ri**->**master**->**addr**->**ip**,** ri**->**master**->**addr**->**port**);**

int retval **=** sentinelSendSlaveOf**(**ri**,**

ri**->**master**->**addr**->**ip**,**

ri**->**master**->**addr**->**port**);**

**if** **(**retval **==** REDIS\_OK**)**

sentinelEvent**(**REDIS\_NOTICE**,**"+convert-to-slave"**,**ri**,**"%@"**);**

**}**

**}**

**}**

**...**

**}**

如果ri->flags中为从节点，但是role为主节点，但是该实例不是在在进行故障转移流程中选中的新主节点。这种情况一般是，之前下线的老的主节点又重新上线了。

因此，在调用sentinelMasterLooksSane函数判断当前主节点状态正常，并且该实例在近期并未主观下线或客观下线，并且该实例上报自己是主节点已经有一段时间了，则调用函数sentinelSendSlaveOf，向该实例发送"SLAVE OF"命令，使其成为从节点。

至此，故障转移流程就介绍完了。但是，因为sentinel是分布式系统，涉及到多个主机，以及网络环境的不稳定等因素，现实中肯定会有很多边界情况的发生，sentinel的代码也肯定是踩过很多坑之后才更新到现在的样子。所以，这里只是介绍了一些主体流程，剩下的，只能在实际的场景中去感受代码的巧妙。