sentinel(哨兵)是redis的高可用解决方案。由一个或多个sentinel实例组成的分布式系统，可以监控任意多个主节点，以及它们属下的所有从节点。当某个主节点下线时，sentinel可以将下线主节点属下的某个从节点升级为新的主节点。

**一：哨兵进程**

哨兵，本质上是redis服务器的一种运行模式。也就是说它们共用大部分的代码，只是哨兵模式中有部分代码是自己特有的。

在Makefile中，哨兵的编译和安装，实际上跟redis服务器是一模一样的：

REDIS\_SERVER\_NAME**=**redis**-**server

REDIS\_SENTINEL\_NAME**=**redis**-**sentinel

**...**

$**(**REDIS\_SENTINEL\_NAME**):** $**(**REDIS\_SERVER\_NAME**)**

**...**

install**:** all

**...**

@ln **-**sf $**(**REDIS\_SERVER\_NAME**)** $**(**INSTALL\_BIN**)/**$**(**REDIS\_SENTINEL\_NAME**)**

因此，哨兵实际上就是redis服务器的软连接而已：

# ll /usr/local/bin/redis-sentinel

lrwxrwxrwx 1 root root 12 May 21 10**:**50 **/**usr**/**local**/**bin**/**redis**-**sentinel **->** redis**-**server

在代码中，使用全局变量server.sentinel\_mode，来决定当前的进程是哨兵模式，还是redis服务器进程：

int main**(**int argc**,** char **\*\***argv**)** **{**

**...**

server**.**sentinel\_mode **=** checkForSentinelMode**(**argc**,**argv**);**

initServerConfig**();**

*/\* We need to init sentinel right now as parsing the configuration file*

*\* in sentinel mode will have the effect of populating the sentinel*

*\* data structures with master nodes to monitor. \*/*

**if** **(**server**.**sentinel\_mode**)** **{**

initSentinelConfig**();**

initSentinel**();**

**}**

**...**

**}**

checkForSentinelMode的代码非常简单质朴，就是扫描命令行参数中是否包含"--sentinel"，或者程序名是否为"redis-sentinel"，来决定当前进程是否为哨兵模式的：

*/\* Returns 1 if there is --sentinel among the arguments or if*

*\* argv[0] is exactly "redis-sentinel". \*/*

int checkForSentinelMode**(**int argc**,** char **\*\***argv**)** **{**

int j**;**

**if** **(**strstr**(**argv**[**0**],**"redis-sentinel"**)** **!=** **NULL)** **return** 1**;**

**for** **(**j **=** 1**;** j **<** argc**;** j**++)**

**if** **(!**strcmp**(**argv**[**j**],**"--sentinel"**))** **return** 1**;**

**return** 0**;**

**}**

因此，哨兵的下面两种启动方法，本质上是一样的：

redis**-**sentinel **/**path**/**to**/**your**/**sentinel**.**conf

redis**-**server **/**path**/**to**/**your**/**sentinel**.**conf **--**sentinel

在main函数中，得到server.sentinel\_mode的值之后，接下来就是调用initServerConfig初始化全局服务器结构：struct redisServer server。

哨兵模式下也会使用该结构，但是在哨兵模式中，接下来就会调用initSentinelConfig和initSentinel来初始化哨兵自己的结构和属性。还会覆盖掉server中某些属性：

/\* This function overwrites a few normal Redis config default with Sentinel

\* specific defaults. \*/

void initSentinelConfig**(**void**)** **{**

server**.**port **=** REDIS\_SENTINEL\_PORT**;**

**}**

*/\* Perform the Sentinel mode initialization. \*/*

void initSentinel**(**void**)** **{**

unsigned int j**;**

*/\* Remove usual Redis commands from the command table, then just add*

*\* the SENTINEL command. \*/*

dictEmpty**(**server**.**commands**,NULL);**

**for** **(**j **=** 0**;** j **<** **sizeof(**sentinelcmds**)/sizeof(**sentinelcmds**[**0**]);** j**++)** **{**

int retval**;**

struct redisCommand **\***cmd **=** sentinelcmds**+**j**;**

retval **=** dictAdd**(**server**.**commands**,** sdsnew**(**cmd**->**name**),** cmd**);**

redisAssert**(**retval **==** DICT\_OK**);**

**}**

*/\* Initialize various data structures. \*/*

sentinel**.**current\_epoch **=** 0**;**

sentinel**.**masters **=** dictCreate**(&**instancesDictType**,NULL);**

sentinel**.**tilt **=** 0**;**

sentinel**.**tilt\_start\_time **=** 0**;**

sentinel**.**previous\_time **=** mstime**();**

sentinel**.**running\_scripts **=** 0**;**

sentinel**.**scripts\_queue **=** listCreate**();**

sentinel**.**announce\_ip **=** **NULL;**

sentinel**.**announce\_port **=** 0**;**

**}**

在initSentinelConfig函数中，使用REDIS\_SENTINEL\_PORT(26379)覆盖掉server.port属性。也就是说，哨兵进程默认的监听端口是26379。

在initSentinel函数中，除了初始化全局哨兵结构struct sentinelState sentinel之外，还会使用sentinelcmds，重新初始化server.commands，该结构中记录了redis支持的命令以及命令处理函数等内容。sentinelcmds的内容如下：

struct redisCommand sentinelcmds**[]** **=** **{**

**{**"ping"**,**pingCommand**,**1**,**""**,**0**,NULL,**0**,**0**,**0**,**0**,**0**},**

**{**"sentinel"**,**sentinelCommand**,-**2**,**""**,**0**,NULL,**0**,**0**,**0**,**0**,**0**},**

**{**"subscribe"**,**subscribeCommand**,-**2**,**""**,**0**,NULL,**0**,**0**,**0**,**0**,**0**},**

**{**"unsubscribe"**,**unsubscribeCommand**,-**1**,**""**,**0**,NULL,**0**,**0**,**0**,**0**,**0**},**

**{**"psubscribe"**,**psubscribeCommand**,-**2**,**""**,**0**,NULL,**0**,**0**,**0**,**0**,**0**},**

**{**"punsubscribe"**,**punsubscribeCommand**,-**1**,**""**,**0**,NULL,**0**,**0**,**0**,**0**,**0**},**

**{**"publish"**,**sentinelPublishCommand**,**3**,**""**,**0**,NULL,**0**,**0**,**0**,**0**,**0**},**

**{**"info"**,**sentinelInfoCommand**,-**1**,**""**,**0**,NULL,**0**,**0**,**0**,**0**,**0**},**

**{**"role"**,**sentinelRoleCommand**,**1**,**"l"**,**0**,NULL,**0**,**0**,**0**,**0**,**0**},**

**{**"client"**,**clientCommand**,-**2**,**"rs"**,**0**,NULL,**0**,**0**,**0**,**0**,**0**},**

**{**"shutdown"**,**shutdownCommand**,-**1**,**""**,**0**,NULL,**0**,**0**,**0**,**0**,**0**}**

**};**

也就是说，在哨兵模式下，支持的命令要比redis服务器要少很多，而且大部分命令的处理函数也不同于redis服务器中的命令处理函数。

**二：数据结构**

在哨兵模式中，最主要的数据结构就是sentinelState。该结构中保存维护了哨兵模式下的所有状态和属性。它的定义如下：

/\* Main state. \*/

struct sentinelState **{**

uint64\_t current\_epoch**;** */\* Current epoch. \*/*

dict **\***masters**;** */\* Dictionary of master sentinelRedisInstances.*

*Key is the instance name, value is the*

*sentinelRedisInstance structure pointer. \*/*

int tilt**;** */\* Are we in TILT mode? \*/*

int running\_scripts**;** */\* Number of scripts in execution right now. \*/*

mstime\_t tilt\_start\_time**;** */\* When TITL started. \*/*

mstime\_t previous\_time**;** */\* Last time we ran the time handler. \*/*

list **\***scripts\_queue**;** */\* Queue of user scripts to execute. \*/*

char **\***announce\_ip**;** */\* IP addr that is gossiped to other sentinels if*

*not NULL. \*/*

int announce\_port**;** */\* Port that is gossiped to other sentinels if*

*non zero. \*/*

**}** sentinel**;**

在sentinelState结构中，最主要的成员就是字典masters。该字典中记录当前哨兵所要监控和交互的所有实例。这些实例包括主节点、从节点和其他哨兵。

masters字典以主节点的名字为key，以主节点实例结构sentinelRedisInstance为value。主节点的名字通过解析配置文件得到。而sentinelRedisInstance结构的定义如下：

**typedef** struct sentinelRedisInstance **{**

int flags**;** */\* See SRI\_... defines \*/*

char **\***name**;** */\* Master name from the point of view of this sentinel. \*/*

char **\***runid**;** */\* run ID of this instance. \*/*

uint64\_t config\_epoch**;** */\* Configuration epoch. \*/*

sentinelAddr **\***addr**;** */\* Master host. \*/*

redisAsyncContext **\***cc**;** */\* Hiredis context for commands. \*/*

redisAsyncContext **\***pc**;** */\* Hiredis context for Pub / Sub. \*/*

int pending\_commands**;** */\* Number of commands sent waiting for a reply. \*/*

mstime\_t cc\_conn\_time**;** */\* cc connection time. \*/*

mstime\_t pc\_conn\_time**;** */\* pc connection time. \*/*

**...**

*/\* Master specific. \*/*

dict **\***sentinels**;** */\* Other sentinels monitoring the same master. \*/*

dict **\***slaves**;** */\* Slaves for this master instance. \*/*

**...**

*/\* Slave specific. \*/*

**...**

struct sentinelRedisInstance **\***master**;** */\* Master instance if it's slave. \*/*

**...**

*/\* Failover \*/*

**...**

struct sentinelRedisInstance **\***promoted\_slave**;**

**...**

**}** sentinelRedisInstance**;**

在哨兵模式中，所有的主节点、从节点以及哨兵实例，都是由sentinelRedisInstance结构表示的。

在该结构中，首先是公共部分，也就是所有实例都会用到的属性，比如：

flags是实例标志位，该标志位中的标记，表示实例的类型或者所处的状态等；

name是实例的名字：每个实例都有一个名字，相当于实例的索引，不同实例具有不同的名字。主节点实例的名字从配置文件中得到，从节点和哨兵实例的名字是由ip和port组成的；

runid记录实例的运行ID；

addr记录实例的地址，其中包含了ip地址和port端口号；

哨兵会与其监控的所有主节点、该主节点下属的所有从节点，以及与之监控相同主节点的其他哨兵之间建立TCP连接。哨兵与主节点和从节点之间会建立两个TCP连接，分别用于发送命令和订阅HELLO频道；哨兵与其他哨兵之间只建立一个发送命令的TCP连接（因为哨兵本身不支持订阅模式）；

哨兵与其他节点进行通信，使用的是Hiredis中的异步方式。因此，sentinelRedisInstance结构中的cc，就是用于命令连接的异步上下文；而其中的pc，就是用于订阅连接的异步上下文；

除了公共部分，不同类型的实例还会有自己特有的属性。比如对于主节点实例而言，它的特有属性有：

sentinels字典：用于记录监控相同主节点其他哨兵实例。该字典以哨兵名字为key，以哨兵实例sentinelRedisInstance结构为key；

slaves字典：用于记录该主节点实例的所有从节点实例。该字典以从节点名字为key，以从节点实例sentinelRedisInstance结构为key；

因此总结而言就是：sentinelState结构中的字典masters中，记录了本哨兵要监控的所有主节点实例，而在表示每个主节点实例的sentinelRedisInstance结构中，字典sentinels中记录了监控该主节点的其他哨兵实例，字典slaves记录了该主节点的所有下属从节点。

这种设计方式非常巧妙，以主节点为核心，将当前哨兵所监控的实例进行分组，每个主节点及其属下的从节点和哨兵，组成一个监控单位，不同监控单位之间的流程是相互隔离的。

对于从节点实例而言，sentinelRedisInstance结构中也有一些它所各有的属性，比如master指针，就指向了它的主节点的sentinelRedisInstance结构；

sentinelRedisInstance结构中还包含与故障转移相关的属性，这在分析哨兵的故障转移流程的代码时会介绍。

**三：初始化**

在哨兵模式下，启动时必须指定一个配置文件，这也是哨兵模式和redis服务器不同的地方，哨兵模式不支持命令行方式的参数配置。

除了redis服务器的配置选项之外，哨兵还需要自己特有的一些配置选项。比如最基本的，就是在配置文件中，需要制定哨兵要监控的主节点：

sentinel monitor **<**mastername**>** **<**masterip**>** **<**masterport**>** **<**quorum**>**

该配置选项中，制定了要监控的主节点的名字、ip、port和quorum值。只有主节点的名字是需要在配置文件中指定，后续所有该主节点的配置选项都以该名字为索引，比如下面就是一个实际的配置内容：

sentinel monitor mymaster 127.0.0.1 6379 2

sentinel down**-**after**-**milliseconds mymaster 60000

sentinel failover**-**timeout mymaster 180000

sentinel parallel**-**syncs mymaster 1

sentinel monitor resque 192.168.1.3 6380 4

sentinel down**-**after**-**milliseconds resque 10000

sentinel failover**-**timeout resque 180000

sentinel parallel**-**syncs resque 5

与哨兵相关的配置选项，第一个单词必须是”sentinel”。上面的配置文件，监控的主节点名字分别是mymaster和resque。

在配置文件中，只需要指定主节点的名字、ip和port信息，而从节点和其他哨兵的信息，都是在信息交互的过程中自动发现的。

在源代码sentinel.c中，函数sentinelHandleConfiguration就是用于解析哨兵配置选项的函数。比如用于解析”sentinel monitor”选项的部分代码如下：

char **\***sentinelHandleConfiguration**(**char **\*\***argv**,** int argc**)** **{**

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

**if** **(!**strcasecmp**(**argv**[**0**],**"monitor"**)** **&&** argc **==** 5**)** **{**

*/\* monitor <name> <host> <port> <quorum> \*/*

int quorum **=** atoi**(**argv**[**4**]);**

**if** **(**quorum **<=** 0**)** **return** "Quorum must be 1 or greater."**;**

**if** **(**createSentinelRedisInstance**(**argv**[**1**],**SRI\_MASTER**,**argv**[**2**],**

atoi**(**argv**[**3**]),**quorum**,NULL)** **==** **NULL)**

**{**

**switch(**errno**)** **{**

**case** EBUSY**:** **return** "Duplicated master name."**;**

**case** ENOENT**:** **return** "Can't resolve master instance hostname."**;**

**case** EINVAL**:** **return** "Invalid port number"**;**

**}**

**}**

**}**

**...**

**}**

”sentinel monitor”选项中，参数个数必须为5个：以”monitor”为第一个参数，剩下的分别是主节点名字、主节点ip，主节点端口，以及quorum值。

上面的代码，就是根据参数值，直接调用createSentinelRedisInstance函数，创建一个SRI\_MASTER标记的主节点实例。

createSentinelRedisInstance函数的代码如下：

sentinelRedisInstance **\***createSentinelRedisInstance**(**char **\***name**,** int flags**,** char **\***hostname**,** int port**,** int quorum**,** sentinelRedisInstance **\***master**)** **{**

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

sentinelAddr **\***addr**;**

dict **\***table **=** **NULL;**

char slavename**[**128**],** **\***sdsname**;**

redisAssert**(**flags **&** **(**SRI\_MASTER**|**SRI\_SLAVE**|**SRI\_SENTINEL**));**

redisAssert**((**flags **&** SRI\_MASTER**)** **||** master **!=** **NULL);**

*/\* Check address validity. \*/*

addr **=** createSentinelAddr**(**hostname**,**port**);**

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

*/\* For slaves and sentinel we use ip:port as name. \*/*

**if** **(**flags **&** **(**SRI\_SLAVE**|**SRI\_SENTINEL**))** **{**

snprintf**(**slavename**,sizeof(**slavename**),**

strchr**(**hostname**,**':'**)** **?** "[%s]:%d" **:** "%s:%d"**,**

hostname**,**port**);**

name **=** slavename**;**

**}**

*/\* Make sure the entry is not duplicated. This may happen when the same*

*\* name for a master is used multiple times inside the configuration or*

*\* if we try to add multiple times a slave or sentinel with same ip/port*

*\* to a master. \*/*

**if** **(**flags **&** SRI\_MASTER**)** table **=** sentinel**.**masters**;**

**else** **if** **(**flags **&** SRI\_SLAVE**)** table **=** master**->**slaves**;**

**else** **if** **(**flags **&** SRI\_SENTINEL**)** table **=** master**->**sentinels**;**

sdsname **=** sdsnew**(**name**);**

**if** **(**dictFind**(**table**,**sdsname**))** **{**

releaseSentinelAddr**(**addr**);**

sdsfree**(**sdsname**);**

errno **=** EBUSY**;**

**return** **NULL;**

**}**

*/\* Create the instance object. \*/*

ri **=** zmalloc**(sizeof(\***ri**));**

*/\* Note that all the instances are started in the disconnected state,*

*\* the event loop will take care of connecting them. \*/*

ri**->**flags **=** flags **|** SRI\_DISCONNECTED**;**

ri**->**name **=** sdsname**;**

ri**->**runid **=** **NULL;**

ri**->**config\_epoch **=** 0**;**

ri**->**addr **=** addr**;**

ri**->**cc **=** **NULL;**

ri**->**pc **=** **NULL;**

**...**

dictAdd**(**table**,** ri**->**name**,** ri**);**

**return** ri**;**

**}**

参数hostname可以是实例的IP地址，也可以是实例的域名；参数flags表示该实例的类型，共有三种类型：SRI\_MASTER，表示要创建的实例是主节点；SRI\_SLAVE，表示要创建的实例是从节点；SRI\_SENTINEL，表示要创建的实例是哨兵；

首先调用函数createSentinelAddr，根据参数hostname和port，创建地址结构addr；hostname有可能是域名，因此createSentinelAddr中，会首先对域名进行解析，如果能解析出IP地址，并且port在范围(0, 65535]中，则将ip和port记录到addr中；如果解析不了hostname，则createSentinelAddr中，会设置errno为ENOENT，并返回NULL；如果port超出了合法范围，则设置errno为EINVAL，并返回NULL；如果createSentinelAddr返回NULL，则createSentinelRedisInstance也直接返回NULL，表示创建实例失败；

参数name表示该实例的名字，主节点的名字在配置文件中配置的；从节点和哨兵的名字由hostname和port组成；

如果该实例为主节点，则参数master为NULL，最终会将该实例存放到字典sentinel.masters中；如果该实例为从节点或哨兵，则参数master不能为NULL，将该实例存放到字典master->slaves或master->sentinels中；如果字典中已经存在同名实例，则设置errno为EBUSY，并且返回NULL，表示创建实例失败；

新创建的实例中，标志位中设置SRI\_DISCONNECTED标记，表示尚未与实例建链；剩下的代码，就是初始化实例的一系列属性，不再赘述；

最后，将该实例插入到相应的字典中；

因此，解析完哨兵的配置文件之后，就已经把所有要监控的主节点实例插入到字典sentinel.masters中了。下一步，就是开始向主节点进行TCP建链了。

**四：哨兵进程的“主函数”**

在介绍哨兵进程的各种流程之前，需要先了解一下哨兵进程的“主函数”。

在redis服务器中的定时器函数serverCron中，每隔100ms就会调用一次sentinelTimer函数。该函数就是哨兵进程的主要处理函数，哨兵中的所有流程都是在该函数中处理的。

void sentinelTimer**(**void**)** **{**

sentinelCheckTiltCondition**();**

sentinelHandleDictOfRedisInstances**(**sentinel**.**masters**);**

sentinelRunPendingScripts**();**

sentinelCollectTerminatedScripts**();**

sentinelKillTimedoutScripts**();**

*/\* We continuously change the frequency of the Redis "timer interrupt"*

*\* in order to desynchronize every Sentinel from every other.*

*\* This non-determinism avoids that Sentinels started at the same time*

*\* exactly continue to stay synchronized asking to be voted at the*

*\* same time again and again (resulting in nobody likely winning the*

*\* election because of split brain voting). \*/*

server**.**hz **=** REDIS\_DEFAULT\_HZ **+** rand**()** **%** REDIS\_DEFAULT\_HZ**;**

**}**

哨兵中记录的所有实例，随着时间的流逝，在各种状态间进行转换，不同的状态下就有不同的处理方式。

该函数中，首先调用sentinelCheckTiltCondition判断当前是否处于TILT模式下，有关TILT模式后续会介绍；

然后调用函数sentinelHandleDictOfRedisInstances处理哨兵中的所有实例；

剩下的就是跟执行脚本相关：sentinelRunPendingScripts用于执行列表sentinel.scripts\_queue中的任务；sentinelCollectTerminatedScripts用于得到执行任务的子进程的退出状态；sentinelKillTimedoutScripts用于杀死执行时间过程的任务；

最后，修改server.hz，增加其随机性，以避免投票选举时发生冲突；

sentinelHandleDictOfRedisInstances函数，是处理该哨兵中保存的所有实例的函数。它的代码如下：

void sentinelHandleDictOfRedisInstances**(**dict **\***instances**)** **{**

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

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

sentinelRedisInstance **\***switch\_to\_promoted **=** **NULL;**

*/\* There are a number of things we need to perform against every master. \*/*

di **=** dictGetIterator**(**instances**);**

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

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

sentinelHandleRedisInstance**(**ri**);**

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

sentinelHandleDictOfRedisInstances**(**ri**->**slaves**);**

sentinelHandleDictOfRedisInstances**(**ri**->**sentinels**);**

**if** **(**ri**->**failover\_state **==** SENTINEL\_FAILOVER\_STATE\_UPDATE\_CONFIG**)** **{**

switch\_to\_promoted **=** ri**;**

**}**

**}**

**}**

**if** **(**switch\_to\_promoted**)**

sentinelFailoverSwitchToPromotedSlave**(**switch\_to\_promoted**);**

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

**}**

sentinelHandleDictOfRedisInstances函数会扫描字典instances ，针对其中的每一个实例，都调用函数sentinelHandleRedisInstance进行处理。而且如果实例是主节点的话，还会递归调用本函数，接着处理字典ri->slaves中的所有从节点实例，以及字典ri->sentinels中的所有哨兵实例。

函数的最后，如果针对某个主节点，发起了故障转移流程，并且流程已经到了最后一步，则会调用函数sentinelFailoverSwitchToPromotedSlave进行处理；

sentinelHandleRedisInstance函数，就是相当于哨兵进程的“主函数”。有关实例的几乎所有动作，都在该函数中进行的。该函数的代码如下：

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

*/\* ========== MONITORING HALF ============ \*/*

*/\* Every kind of instance \*/*

sentinelReconnectInstance**(**ri**);**

sentinelSendPeriodicCommands**(**ri**);**

*/\* ============== ACTING HALF ============= \*/*

*/\* We don't proceed with the acting half if we are in TILT mode.*

*\* TILT happens when we find something odd with the time, like a*

*\* sudden change in the clock. \*/*

**if** **(**sentinel**.**tilt**)** **{**

**if** **(**mstime**()-**sentinel**.**tilt\_start\_time **<** SENTINEL\_TILT\_PERIOD**)** **return;**

sentinel**.**tilt **=** 0**;**

sentinelEvent**(**REDIS\_WARNING**,**"-tilt"**,NULL,**"#tilt mode exited"**);**

**}**

*/\* Every kind of instance \*/*

sentinelCheckSubjectivelyDown**(**ri**);**

*/\* Masters and slaves \*/*

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

*/\* Nothing so far. \*/*

**}**

*/\* Only masters \*/*

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

sentinelCheckObjectivelyDown**(**ri**);**

**if** **(**sentinelStartFailoverIfNeeded**(**ri**))**

sentinelAskMasterStateToOtherSentinels**(**ri**,**SENTINEL\_ASK\_FORCED**);**

sentinelFailoverStateMachine**(**ri**);**

sentinelAskMasterStateToOtherSentinels**(**ri**,**SENTINEL\_NO\_FLAGS**);**

**}**

**}**

本函数用于处理实例的所有状态。在哨兵中，每个实例在某一时刻都处于某种状态，随着时间的流逝，实例在不同的状态之间转换，本函数就是在实例处于不同状态下，进行不同的处理；

首先调用sentinelReconnectInstance函数，如果实例处于断链状态，则调用该函数进行TCP建链；

然后调用sentinelSendPeriodicCommands函数，向实例发送"PING"，"INFO"或"PUBLISH"消息。

接下来，如果哨兵当前处于TILT模式下，并且处于该模式下的时间还不到SENTINEL\_TILT\_PERIOD毫秒，则直接返回，而不在进行后续的处理；如果处于TILT模式下已经超过了SENTINEL\_TILT\_PERIOD毫秒，则退出TILT模式；

然后，调用函数sentinelCheckSubjectivelyDown检查实例是否主观下线；

最后，如果当前实例为主节点，则检查该实例是否客观下线，并在必要的情况下，发起故障转移流程；

**五：建链**

哨兵对于其所监控的所有主节点，及其属下的所有从节点，都会建立两个TCP连接。一个用于发送命令，一个用于订阅其HELLO频道。而哨兵对于监控同一主节点的其他哨兵实例，只建立一个命令连接。

哨兵向其他实例建立的命令连接，主要用于发送”PING”、”INFO”或”PUBLISH”命令。哨兵会根据实例对于这些命令的回复时间和回复内容，修改该实例的状态；

哨兵向主节点和从节点建立的订阅连接，主要是为了监控同一主节点的所有哨兵之间，能够相互发现，以及交换信息。

哨兵与其他实例之间的交互，主要是通过Hiredis的异步方式进行的。关于Hiredis的异步方式，可以参考之前的文章。

TCP连接的建立，就是通过函数sentinelReconnectInstance实现的。该函数的代码如下：

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

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

*/\* Commands connection. \*/*

**if** **(**ri**->**cc **==** **NULL)** **{**

ri**->**cc **=** redisAsyncConnectBind**(**ri**->**addr**->**ip**,**ri**->**addr**->**port**,**REDIS\_BIND\_ADDR**);**

**if** **(**ri**->**cc**->**err**)** **{**

sentinelEvent**(**REDIS\_DEBUG**,**"-cmd-link-reconnection"**,**ri**,**"%@ #%s"**,**

ri**->**cc**->**errstr**);**

sentinelKillLink**(**ri**,**ri**->**cc**);**

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

ri**->**cc\_conn\_time **=** mstime**();**

ri**->**cc**->**data **=** ri**;**

redisAeAttach**(**server**.**el**,**ri**->**cc**);**

redisAsyncSetConnectCallback**(**ri**->**cc**,**

sentinelLinkEstablishedCallback**);**

redisAsyncSetDisconnectCallback**(**ri**->**cc**,**

sentinelDisconnectCallback**);**

sentinelSendAuthIfNeeded**(**ri**,**ri**->**cc**);**

sentinelSetClientName**(**ri**,**ri**->**cc**,**"cmd"**);**

*/\* Send a PING ASAP when reconnecting. \*/*

sentinelSendPing**(**ri**);**

**}**

**}**

*/\* Pub / Sub \*/*

**if** **((**ri**->**flags **&** **(**SRI\_MASTER**|**SRI\_SLAVE**))** **&&** ri**->**pc **==** **NULL)** **{**

ri**->**pc **=** redisAsyncConnectBind**(**ri**->**addr**->**ip**,**ri**->**addr**->**port**,**REDIS\_BIND\_ADDR**);**

**if** **(**ri**->**pc**->**err**)** **{**

sentinelEvent**(**REDIS\_DEBUG**,**"-pubsub-link-reconnection"**,**ri**,**"%@ #%s"**,**

ri**->**pc**->**errstr**);**

sentinelKillLink**(**ri**,**ri**->**pc**);**

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

int retval**;**

ri**->**pc\_conn\_time **=** mstime**();**

ri**->**pc**->**data **=** ri**;**

redisAeAttach**(**server**.**el**,**ri**->**pc**);**

redisAsyncSetConnectCallback**(**ri**->**pc**,**

sentinelLinkEstablishedCallback**);**

redisAsyncSetDisconnectCallback**(**ri**->**pc**,**

sentinelDisconnectCallback**);**

sentinelSendAuthIfNeeded**(**ri**,**ri**->**pc**);**

sentinelSetClientName**(**ri**,**ri**->**pc**,**"pubsub"**);**

*/\* Now we subscribe to the Sentinels "Hello" channel. \*/*

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

sentinelReceiveHelloMessages**,** **NULL,** "SUBSCRIBE %s"**,**

SENTINEL\_HELLO\_CHANNEL**);**

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

*/\* If we can't subscribe, the Pub/Sub connection is useless*

*\* and we can simply disconnect it and try again. \*/*

sentinelKillLink**(**ri**,**ri**->**pc**);**

**return;**

**}**

**}**

**}**

*/\* Clear the DISCONNECTED flags only if we have both the connections*

*\* (or just the commands connection if this is a sentinel instance). \*/*

**if** **(**ri**->**cc **&&** **(**ri**->**flags **&** SRI\_SENTINEL **||** ri**->**pc**))**

ri**->**flags **&=** **~**SRI\_DISCONNECTED**;**

**}**

如果实例标志位中没有SRI\_DISCONNECTED标记，则表示该实例已经建链，直接返回；

实例中的异步上下文cc，用于向该实例发送命令。如果cc为NULL，则首先调用函数redisAsyncConnectBind创建异步上下文，并发起非阻塞的TCP建链；

创建异步上下文cc成功之后，调用redisAeAttach将该上下文与AE事件loop结合；然后调用redisAsyncSetConnectCallback，设置建链回调函数sentinelLinkEstablishedCallback，并注册可写事件；然后调用redisAsyncSetDisconnectCallback，设置断链回调函数sentinelDisconnectCallback；然后调用sentinelSendAuthIfNeeded，向该实例异步发送"AUTH XXX"命令，进行密码认证；然后调用sentinelSetClientName，向该实例异步发送命令"CLIENT SETNAME sentinel-<runid>-cmd"，设置该客户端连接的名字；最后调用函数sentinelSendPing，向实例异步发送"PING"命令；

实例中的异步上下文pc，用于订阅实例的HELLO频道。如果pc为NULL，并且实例ri不是哨兵的话，则首先调用redisAsyncConnectBind创建异步上下文，并发起非阻塞的TCP建链；

创建异步上下文pc成功之后，调用redisAeAttach将该上下文与AE事件loop结合；然后调用redisAsyncSetConnectCallback，设置建链回调函数sentinelLinkEstablishedCallback，并注册可写事件；然后调用redisAsyncSetDisconnectCallback，设置断链回调函数sentinelDisconnectCallback；然后调用sentinelSendAuthIfNeeded，向该实例异步发送"AUTH XXX"命令，进行密码认证；然后调用sentinelSetClientName，向该实例异步发送命令"CLIENT SETNAME sentinel-<runid>-pubsub"，设置该客户端连接的名字；最后向实例异步发送"SUBSCRIBE \_\_sentinel\_\_:hello"命令，订阅实例的HELLO频道，回调函数为sentinelReceiveHelloMessages，当收到该频道发布的消息时，就会调用该函数；

如果异步上下文cc和pc都创建好了，则可以从实例标志位中去除SRI\_DISCONNECTED

标记。如果实例为哨兵，则无需创建异步上下文pc。

注意：以上两个连接的建连回调函数，只是会在TCP连接建立成功或失败时被调用，用于打印一些信息；而断连回调函数，是在TCP断连时被调用，用于将断连的异步上下文置为NULL，并且将标记SRI\_DISCONNECTED增加到实例标志位中，这样，下次调用sentinelReconnectInstance函数时，就会重新建连了。