# [Redis Sentinel机制与用法说明](http://www.cnblogs.com/zhoujinyi/p/5569462.html)(哨兵模式)

**概述**

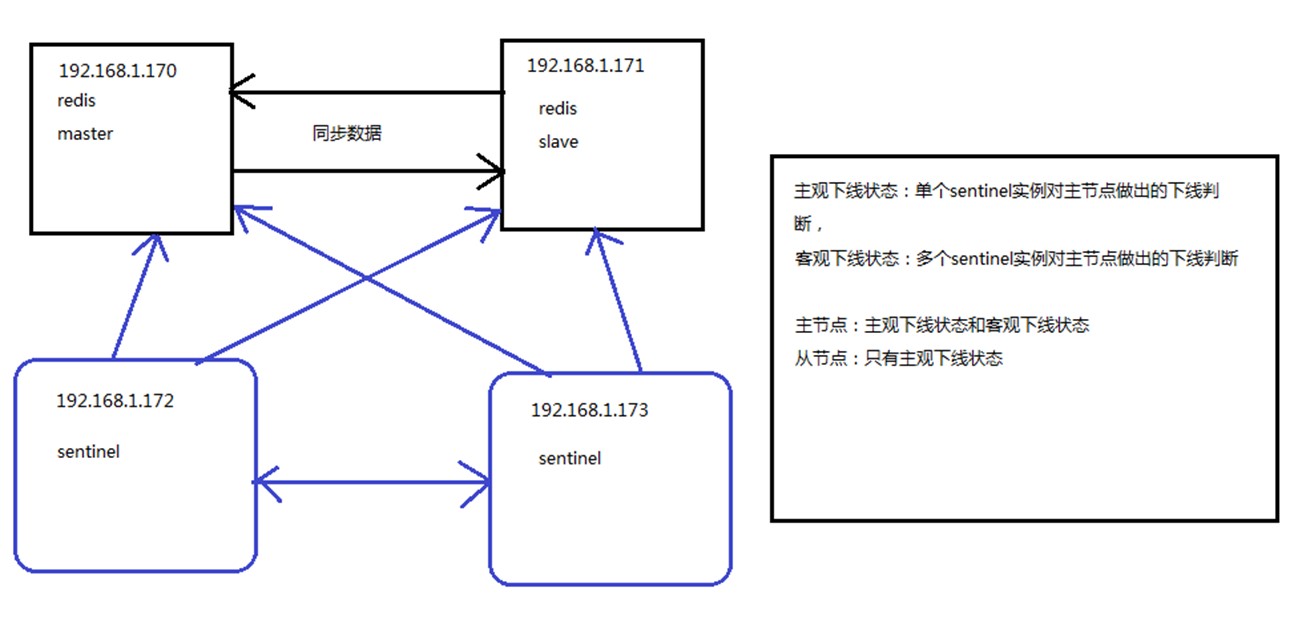
1、主Redis与各个从Redis之间能够实现高可用(监控、自动故障迁移等)，这个委托给redis-sentinel来实现。   
2、业务代码中为分散redis服务的压力，全部写操作走主库，部分读操作需要走从服务器的业务需求。(就目前jedis 2.6.2版本中的JedisSentinelPool做不到这点)   
3、对多个Master-Slaves组成的高可用单元做sharding集群

#### 一、sentinel介绍

Redis Sentinel   
Sentinel(哨兵)是用于监控redis集群中Master状态的工具，其已经被集成在redis2.4+的版本中

**Sentinel作用：**   
1)：Master状态检测   
2)：如果Master异常，则会进行Master-Slave切换，将其中一个Slave作为Master，将之前的Master作为Slave   
3)：Master-Slave切换后，master\_redis.conf、slave\_redis.conf和sentinel.conf的内容都会发生改变，即master\_redis.conf中会多一行slaveof的配置，sentinel.conf的监控目标会随之调换   
**Sentinel工作方式：**   
1)：每个Sentinel以每秒钟一次的频率向它所知的Master，Slave以及其他 Sentinel 实例发送一个 PING 命令   
2)：如果一个实例（instance）距离最后一次有效回复 PING 命令的时间超过 down-after-milliseconds 选项所指定的值， 则这个实例会被 Sentinel 标记为主观下线。   
3)：如果一个Master被标记为主观下线，则正在监视这个Master的所有 Sentinel 要以每秒一次的频率确认Master的确进入了主观下线状态。   
4)：当有足够数量的 Sentinel（大于等于配置文件指定的值）在指定的时间范围内确认Master的确进入了主观下线状态， 则Master会被标记为客观下线   
5)：在一般情况下， 每个 Sentinel 会以每 10 秒一次的频率向它已知的所有Master，Slave发送 INFO 命令   
6)：当Master被 Sentinel 标记为客观下线时，Sentinel 向下线的 Master 的所有 Slave 发送 INFO 命令的频率会从 10 秒一次改为每秒一次   
7)：若没有足够数量的 Sentinel 同意 Master 已经下线， Master 的客观下线状态就会被移除。   
若 Master 重新向 Sentinel 的 PING 命令返回有效回复， Master 的主观下线状态就会被移除。

**主观下线和客观下线**   
主观下线：Subjectively Down，简称 SDOWN，指的是当前 Sentinel 实例对某个redis服务器做出的下线判断。   
客观下线：Objectively Down， 简称 ODOWN，指的是多个 Sentinel 实例在对Master Server做出 SDOWN 判断，并且通过 SENTINEL is-master-down-by-addr 命令互相交流之后，得出的Master Server下线判断，然后开启failover.

**通俗来讲就是：**   
redis的sentinel系统用来管理多个redis服务器，可以实现一个功能上实现HA的集群。该系统主要执行三个任务：   
①监控（ Monitoring ）： Redis Sentinel实时监控主服务器和从服务器运行状态。   
②提醒（notification）： 当被监控的某个 Redis 服务器出现问题时， Redis Sentinel 可以向系统管理员发送通知， 也可以通过 API 向其他程序发送通知   
一个简单的主从结构加sentinel集群的架构图如下：   
  
**上图是一主一从节点，加上两个部署了sentinel的集群，sentinel集群之间会互相通信，沟通交流redis节点的状态，做出相应的判断并进行处理，这里的主观下线状态和客观下线状态是比较重要的状态，它们决定了是否进行故障转移**可以 通过订阅指定的频道信息，当服务器出现故障得时候通知管理员   
客户端可以将 Sentinel 看作是一个只提供了订阅功能的 Redis 服务器，你不可以使用 PUBLISH 命令向这个服务器发送信息，但你可以用 SUBSCRIBE 命令或者 PSUBSCRIBE 命令， 通过订阅给定的频道来获取相应的事件提醒。   
一个频道能够接收和这个频道的名字相同的事件。 比如说， 名为 +sdown 的频道就可以接收所有实例进入主观下线（SDOWN）状态的事件。

Redis-Sentinel是Redis官方推荐的高可用性(HA)解决方案，**当用Redis做Master-slave的高可用方案时**，假如master宕机了，Redis本身(包括它的很多客户端)都没有实现自动进行主备切换，而**Redis-sentinel本身也是一个独立运行的进程，它能监控多个master-slave集群，发现master宕机后能进行自动切换。**

它的**主要功能**有以下几点

* 不时地**监控redis**是否按照预期良好地运行;
* 如果发现某个redis节点运行出现状况，能够通知另外一个进程(例如它的客户端);
* 能够进行自动切换。当一个master节点不可用时，能够选举出master的多个slave(如果有超过一个slave的话)中的一个来作为新的master,其它的slave节点会将它所追随的master的地址改为被提升为master的slave的新地址。

**Sentinel支持集群**

很显然，只使用单个sentinel进程来监控redis集群是不可靠的，当sentinel进程宕掉后(sentinel本身也有单点问题，single-point-of-failure)整个集群系统将无法按照预期的方式运行。

所以有必要将sentinel集群，这样有几个好处：

* 即使有一些sentinel进程宕掉了，依然可以进行redis集群的主备切换；
* 如果只有一个sentinel进程，如果这个进程**运行出错，或者是网络堵塞**，那么将无法实现redis集群的主备切换**（单点问题）**;
* 如果有多个sentinel，redis的客户端可以**随意地连接任意一个sentinel来获得关于redis集群中的信息**。

**Sentinel版本**

Sentinel当前最新的稳定版本称为Sentinel 2(与之前的Sentinel 1区分开来）。随着redis2.8的安装包一起发行。安装完Redis2.8后，可以在redis2.8/src/里面找到Redis-sentinel的启动程序。

***强烈建议*：**  
如果你使用的是redis2.6(sentinel版本为sentinel 1)，你最好应该使用redis2.8版本的sentinel 2，因为sentinel 1有很多的Bug，已经被官方弃用，所以强烈建议使用redis2.8以及sentinel 2。

**运行Sentinel**

运行sentinel有两种方式：

* 第一种

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

* 第二种

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

以上两种方式，都必须指定一个sentinel的配置文件sentinel.conf，如果不指定，将无法启动sentinel。sentinel默认监听26379端口，所以运行前必须确定该端口没有被别的进程占用。

**Sentinel的配置**

Redis源码包中包含了一个sentinel.conf文件作为sentinel的配置文件，配置文件自带了关于各个配置项的解释。典型的配置项如下所示：

[复制代码](javascript:void(0);)

**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

[复制代码](javascript:void(0);)

上面的配置项配置了两个名字分别为mymaster和resque的master，配置文件只需要配置master的信息就好啦，不用配置slave的信息，因为slave能够被自动检测到(master节点会有关于slave的消息)。需要注意的是，配置文件在sentinel运行期间是会被动态修改的，例如当发生主备切换时候，配置文件中的master会被修改为另外一个slave。这样，之后sentinel如果重启时，就可以根据这个配置来恢复其之前所监控的redis集群的状态。

接下来我们将**一行一行地解释上面的配置项：**

**sentinel monitor mymaster 127.0.0.1 6379 2**

这一行代表sentinel监控的master的名字叫做mymaster,地址为127.0.0.1:6379，行尾最后的一个2代表什么意思呢？我们知道，网络是不可靠的，有时候一个sentinel会因为网络堵塞而误以为一个master redis已经死掉了，当sentinel集群式，解决这个问题的方法就变得很简单，只需要多个sentinel互相沟通来确认某个master是否真的死了，这个2代表，**当集群中有2个sentinel认为master死了时，才能真正认为该master已经不可用了。**（sentinel集群中各个sentinel也有互相通信，通过gossip协议）。

除了第一行配置，我们发现剩下的配置都有一个统一的格式:

**sentinel <option\_name> <master\_name> <option\_value>**

接下来我们根据上面格式中的option\_name一个一个来解释这些配置项：

* **down-after-milliseconds**  
  sentinel会向master发送心跳*PING*来确认master是否存活，如果master在“一定时间范围”内不回应*PONG* 或者是回复了一个错误消息，那么这个sentinel会主观地(单方面地)认为这个master已经不可用了(subjectively down, 也简称为**SDOWN**)。而这个down-after-milliseconds就是用来指定这个“一定时间范围”的，单位是毫秒。

不过需要注意的是，这个时候sentinel并不会马上进行failover（故障切换）主备切换，这个sentinel还需要参考sentinel集群中其他sentinel的意见，如果超过某个数量的sentinel也主观地认为该master死了，那么这个master就会被客观地(注意哦，这次不是主观，是客观，与刚才的subjectively down相对，这次是objectively down，简称为**ODOWN**)认为已经死了。需要一起做出决定的sentinel数量在上一条配置中进行配置。

* **parallel-syncs**  
  在发生failover主备切换时，这个选项指定了最多可以有多少个slave同时对新的master进行同步，这个数字越小，完成failover所需的时间就越长，但是如果这个数字越大，就意味着越多的slave因为replication而不可用。可以通过将这个值设为 1 来保证每次只有一个slave处于不能处理命令请求的状态。

其他配置项在sentinel.conf中都有很详细的解释。  
所有的配置都可以在运行时用命令SENTINEL SET command动态修改。

**Sentinel的“仲裁会”**

前面我们谈到，当一个master被sentinel集群监控时，需要为它指定一个参数，这个参数指定了当需要判决master为不可用，并且进行failover（主备切换）时，所需要的sentinel数量，**本文中我们暂时称这个参数为票数(投票给master)**

不过，当failover主备切换真正被触发后，failover并不会马上进行，还需要sentinel中的**大多数sentinel授权后**才可以进行failover。  
当ODOWN（主观授权）时，failover被触发。failover一旦被触发，尝试去进行failover的**（想获得执行权）**sentinel会去获得“大多数”sentinel的授权（如果票数比大多数还要大的时候，则询问更多的sentinel)**则授权的票数要比认为它DIE的票数多**  
这个区别看起来很微妙，但是很容易理解和使用。例如，集群中有5个sentinel，票数被设置为2，当2个sentinel认为一个master已经不可用了以后，将会触发failover，但是，进行**failover的那个sentinel必须先获得至少3个sentinel的授权才可以实行failover。**如果票数被设置为5，要达到ODOWN状态，必须所有5个sentinel都主观认为master为不可用，要进行failover，那么得获得所有5个sentinel的授权。

**配置版本号**

为什么要先获得大多数sentinel的认可时才能真正去执行failover呢？

当一个sentinel被授权后，它将会获得宕掉的master的一份最新配置版本号，当failover执行结束以后，这个版本号将会被用于最新的配置。

因为大多数sentinel都已经知道该版本号已经被要执行failover的sentinel拿走了，所以其他的sentinel都不能再去使用这个版本号。这意味着，每次failover都会附带有一个独一无二的版本号。我们将会看到这样做的重要性。

而且，sentinel集群都遵守一个规则：如果sentinel A推荐sentinel B去执行failover，B会等待一段时间后，自行再次去对同一个master执行failover，这个等待的时间是通过failover-timeout配置项去配置的。从这个规则可以看出，sentinel集群中的sentinel不会再同一时刻并发去failover同一个master，第一个进行failover的sentinel如果失败了，另外一个将会在一定时间内进行重新进行failover，以此类推。

redis sentinel保证了活跃性：如果大多数sentinel能够互相通信，最终将会有一个被授权去进行failover.  
redis sentinel也保证了安全性：每个试图去failover同一个master的sentinel都会得到一个独一无二的版本号。

**配置传播**

一旦一个sentinel成功地对一个master进行了failover，它将会把关于master的最新配置通过广播形式通知其它sentinel，其它的sentinel则更新对应master的配置。

一个faiover要想被成功实行，sentinel必须能够向选为master的slave发送**SLAVEOF NO ONE**命令，然后能够通过INFO命令看到新master的配置信息。

当将一个slave选举为master并发送**SLAVEOF NO ONE**后，即使其它的slave还没针对新master重新配置自己，failover也被认为是成功了的，然后所有sentinels将会发布新的配置信息。

新配在集群中相互传播的方式，就是为什么我们需要当一个sentinel进行failover时必须被授权一个版本号的原因。

每个sentinel使用##发布/订阅##的方式持续地传播master的配置版本信息，配置传播的##发布/订阅##管道是：\_\_sentinel\_\_:hello。

因为每一个配置都有一个版本号，所以以版本号最大的那个为标准。

举个栗子：假设有一个名为mymaster的地址为192.168.1.50:6379。一开始，集群中所有的sentinel都知道这个地址，于是为mymaster的配置打上版本号1。一段时候后mymaster死了，有一个sentinel被授权用版本号2对其进行failover。如果failover成功了，假设地址改为了192.168.1.50:9000（从的为9000），此时配置的版本号为2，进行failover的sentinel会将新配置广播给其他的sentinel，由于其他sentinel维护的版本号为1，发现新配置的版本号为2时，版本号变大了，说明配置更新了，于是就会采用最新的版本号为2的配置。

这意味着sentinel集群保证了第二种活跃性：一个能够互相通信的sentinel集群最终会采用版本号最高且相同的配置。

**SDOWN和ODOWN的更多细节**

sentinel对于不可用有两种不同的看法，一个叫主观不可用(SDOWN),另外一个叫客观不可用(ODOWN)。SDOWN是sentinel自己主观上检测到的关于master的状态，ODOWN需要一定数量的sentinel达成一致意见才能认为一个master客观上已经宕掉，各个sentinel之间通过命令SENTINEL is\_master\_down\_by\_addr来获得其它sentinel对master的检测结果。

从sentinel的角度来看，如果发送了*PING*心跳后，在一定时间内没有收到合法的回复，就达到了SDOWN的条件。这个时间在配置中通过is-master-down-after-milliseconds参数配置。

当sentinel发送*PING*后，以下回复之一都被认为是合法的：

PING replied with +PONG.

PING replied with -LOADING error.

PING replied with -MASTERDOWN error.

其它任何回复（或者根本没有回复）都是不合法的。

从SDOWN切换到ODOWN不需要任何一致性算法，只需要一个gossip协议：如果一个sentinel收到了足够多的sentinel发来消息告诉它某个master已经down掉了，SDOWN状态就会变成ODOWN状态。如果之后master可用了，这个状态就会相应地被清理掉。

正如之前已经解释过了，真正进行failover需要一个授权的过程，但是所有的failover都开始于一个ODOWN状态。

ODOWN状态只适用于master，对于不是master的redis节点sentinel之间不需要任何协商，slaves和sentinel不会有ODOWN状态。

**Sentinel之间和Slaves之间的自动发现机制**

虽然sentinel集群中各个sentinel都互相连接彼此来检查对方的可用性以及互相发送消息。但是你不用在任何一个sentinel配置任何其它的sentinel的节点。**因为sentinel利用了master的发布/订阅机制去自动发现其它也监控了统一master的sentinel节点。**

通过向名为\_\_sentinel\_\_:hello的管道中发送消息来实现。

同样，你也不需要在sentinel中配置某个master的所有slave的地址，**sentinel会通过询问master来得到这些slave的地址的。**

每个sentinel通过向每个master和slave的发布/订阅频道\_\_sentinel\_\_:hello每秒发送一次消息，来宣布它的存在。  
每个sentinel也订阅了每个master和slave的频道\_\_sentinel\_\_:hello的内容，来发现未知的sentinel，当检测到了新的sentinel，则将其加入到自身维护的master监控列表中。  
每个sentinel发送的消息中也包含了其当前维护的最新的master配置。如果某个sentinel**发现自己的配置版本低于接收到的配置版本，则会用新的配置更新自己的master配置。**

在为一个master添加一个新的sentinel前，sentinel总是检查是否已经有sentinel与新的sentinel的进程号或者是地址是一样的。如果是那样，这个sentinel将会被删除，而把新的sentinel添加上去。

（**即每个master都有相关的其他sentinel和slave的信息，每个sentinel都通过了发布/订阅频道取得了master相关的sentinel和slave信息）**

**网络隔离时的一致性**

redis sentinel集群的配置的一致性模型为最终一致性，集群中每个sentinel最终都会采用最高版本的配置。然而，在实际的应用环境中，有三个不同的角色会与sentinel打交道：

* Redis实例.
* Sentinel实例.
* 客户端.

为了考察整个系统的行为我们必须同时考虑到这三个角色。

下面有个简单的例子，有三个主机，每个主机分别运行一个redis和一个sentinel:

+-------------+

| Sentinel 1 | <--- Client A

| Redis 1 (M) |

+-------------+

|

|

+-------------+ | +------------+

| Sentinel 2 |-----+-- / partition / ----| Sentinel 3 | <--- Client B

| Redis 2 (S) | | Redis 3 (M)|

+-------------+ +------------+

在这个系统中，初始状态下**redis3是master**, redis1和redis2是slave。之后**redis3所在的主机网络不可用了**，sentinel1和sentinel2启动了failover并把redis1选举为master。

Sentinel集群的特性保证了sentinel1和sentinel2得到了关于master的最新配置。但是sentinel3依然持着的是**旧的配置**，因为它与外界隔离了。

当网络恢复以后，我们知道sentinel3将会更新它的配置。但是，如果客户端所连接的master被网络隔离，会发生什么呢？

客户端将依然可以向redis3写数据，但是当网络恢复后，redis3就会变成redis的一个slave，那么，在网络隔离期间，客户端向redis3写的数据将会丢失。

也许你不会希望这个场景发生：

* 如果你把redis当做缓存来使用，那么你也许能容忍这部分数据的丢失。
* 但如果你把redis当做一个存储系统来使用，你也许就无法容忍这部分数据的丢失了。

因为redis采用的是异步复制，在这样的场景下，没有办法避免数据的丢失。然而，你可以通过以下**配置来配置redis3和redis1，使得数据不会丢失。**

min-slaves-to-write 1

min-slaves-max-lag 10

通过上面的配置，当一个redis是master时，**如果它不能向至少一个slave写数据**(上面的min-slaves-to-write指定了slave的数量)，**它将会拒绝接受客户端的写请求。**由于复制是异步的，master无法向slave写数据意味着slave要么断开连接了，要么**不在指定时间内向master发送同步数据的请求了**(上面的min-slaves-max-lag指定了这个时间)。

**Sentinel状态持久化**

**snetinel的状态会被持久化地写入sentinel的配置文件中**。每次当收到一个新的配置时，或者新创建一个配置时，配置会被持久化到硬盘中，并带上配置的版本戳。这意味着，可以安全的停止和重启sentinel进程。

**无failover时的配置纠正**

即使当前没有failover正在进行，sentinel依然会使用当前配置去设置监控的master。特别是：

1.根据最新配置确认为slaves的节点却声称自己是master(上文例子中被网络隔离后的的redis3)，**这时它们会被重新配置为当前master的slave**。

2.如果slaves连接了一个错误的master，将会被改正过来，连接到正确的master。

**Slave选举与优先级**

当一个sentinel准备好了要进行failover，并且收到了其他sentinel的授权，那么就**需要选举出一个合适的slave来做为新的master**。

slave的选举主要会评估slave的以下几个方面：

* **与master断开连接的次数**
* **Slave的优先级**
* **数据复制的下标(用来评估slave当前拥有多少master的数据)**
* **进程ID**

如果一个slave与master失去联系超过10次，并且每次都超过了配置的最大失联时间(down-after-milliseconds)，如果sentinel在进行failover时发现slave失联，那么这个slave就会被sentinel认为不适合用来做新master的。

更严格的定义是，如果一个slave持续断开连接的时间超过

(down-after-milliseconds \* 10) + milliseconds\_since\_master\_is\_in\_SDOWN\_state

就会**被认为失去选举资格。（超时，或者断开次数太多）**符合上述条件的slave才会被列入master候选人列表，并根据以下顺序来进行排序：

1. sentinel首先会根据slaves的优先级来进行排序，优先级越小排名越靠前。
2. 如果优先级相同，则查看复制的下标，哪个从master接收的复制数据多，哪个就靠前。
3. 如果优先级和下标都相同，就选择进程ID较小的那个。

一个redis无论是master还是slave，都必须在配置中指定一个slave优先级。要注意到master也是有可能通过failover变成slave的。

如果一个redis的slave优先级配置为0，那么它将永远不会被选为master。但是它依然会从master哪里复制数据。

**Sentinel和Redis身份验证**

当一个master配置为需要密码才能连接时，客户端和slave在连接时都需要提供密码。

master通过requirepass设置自身的密码，不提供密码无法连接到这个master。  
slave通过masterauth来设置访问master时的密码。

但是当使用了sentinel时，由于一个master可能会变成一个slave，一个slave也可能会变成master，所以需要同时设置上述两个配置项。

**Sentinel API**

Sentinel默认运行在26379端口上，sentinel支持redis协议，所以可以使用redis-cli客户端或者其他可用的客户端来与sentinel通信。

有两种方式能够与sentinel通信：

* 一种是直接使用客户端向它发消息
* 另外一种是使用**发布/订阅模式来订阅sentinel事件**，比如说failover，或者某个redis实例运行出错，等等。

**Sentinel命令**

**sentinel支持的合法命令如下：**

* PING sentinel回复PONG.
* **SENTINEL masters**显示被监控的所有master以及它们的状态.
* **SENTINEL master <master name>** 显示指定master的信息和状态；
* **SENTINEL slaves <master name>** 显示指定master的所有slave以及它们的状态；
* **SENTINEL get-master-addr-by-name <master name>**返回指定master的ip和端口，如果正在进行failover或者failover已经完成，将会显示被提升为master的slave的ip和端口。
* **SENTINEL reset <pattern>** 重置名字匹配该正则表达式的所有的master的状态信息，清楚其之前的状态信息，以及slaves信息。
* **SENTINEL failover <master name>** 强制sentinel执行failover，并且不需要得到其他sentinel的同意。但是failover后会将最新的配置发送给其他sentinel。

**动态修改Sentinel配置**

从redis2.8.4开始，sentinel提供了一组API用来添加，删除，修改master的配置。

需要注意的是，如果你通过API修改了一个sentinel的配置，sentinel不会把修改的配置告诉其他sentinel。你需要自己手动地对多个sentinel发送修改配置的命令。

以下是一些**修改sentinel配置**的命令：

* **SENTINEL MONITOR <name> <ip> <port> <quorum>** 这个命令告诉sentinel去监听一个新的master
* **SENTINEL REMOVE <name>**命令sentinel放弃对某个master的监听
* **SENTINEL SET <name> <option> <value>** 这个命令很像Redis的CONFIG SET命令，用来改变指定master的配置。支持多个<option><value>。例如以下实例：
* **SENTINEL SET objects-cache-master down-after-milliseconds 1000**

只要是配置文件中存在的配置项，都可以用**SENTINEL SET**命令来设置。这个还可以用来设置master的属性，比如说**quorum**(票数)，而不需要先删除master，再重新添加master。例如：

**SENTINEL SET objects-cache-master quorum 5**

**增加或删除Sentinel**

**由于有sentinel自动发现机制，所以添加一个sentinel到你的集群中非常容易，你所需要做的只是监控到某个Master上，然后新添加的sentinel就能获得其他sentinel的信息以及master所有的slaves。**

如果你需要添加多个sentinel，建议你一个接着一个添加，这样可以预防网络隔离带来的问题。你可以每个30秒添加一个sentinel。最后你可以用SENTINEL MASTER mastername来检查一下是否所有的sentinel都已经监控到了master。

删除一个sentinel显得有点复杂：因为sentinel永远不会删除一个已经存在过的sentinel，即使它已经与组织失去联系很久了。  
要想删除一个sentinel，应该遵循如下步骤：

1. 停止所要删除的sentinel
2. **发送一个SENTINEL RESET \* 命令给所有其它的sentinel实例**，如果你想要重置指定master上面的sentinel，只需要把\*号改为特定的名字，注意，需要一个接一个发，每次发送的间隔不低于30秒。
3. 检查一下所有的sentinels是否都有一致的当前sentinel数。使用SENTINEL MASTER mastername 来查询。

**删除旧master或者不可达slave**

sentinel永远会记录好一个Master的slaves，即使slave已经与组织失联好久了。这是很有用的，因为sentinel集群必须有能力把一个恢复可用的slave进行重新配置。

并且，failover后，失效的master将会被标记为新master的一个slave，这样的话，当它变得可用时，就会从新master上复制数据。

然后，有时候你想要永久地删除掉一个slave(有可能它曾经是个master)，你只需要发送一个SENTINEL RESET master命令给所有的sentinels，它们将会更新列表里能够正确地复制master数据的slave。

**发布/订阅**

客户端可以向一个sentinel发送订阅某个频道的事件的命令，当有特定的事件发生时，sentinel会通知所有订阅的客户端。需要注意的是**客户端只能订阅，不能发布。**

订阅频道的名字与事件的名字一致。例如，频道名为sdown 将会发布所有与SDOWN相关的消息给订阅者。

如果想要订阅所有消息，只需简单地使用PSUBSCRIBE \*

以下是所有你可以收到的消息的消息格式，如果你订阅了所有消息的话。第一个单词是频道的名字，其它是数据的格式。

注意：以下的instance details的格式是：

<instance-type> <name> <ip> <port> @ <master-name> <master-ip> <master-port>

如果这个redis实例是一个master，那么@之后的消息就不会显示。

[复制代码](javascript:void(0);)

+reset-master <instance details> -- 当master被重置时.

+slave <instance details> -- 当检测到一个slave并添加进slave列表时.

+failover-state-reconf-slaves <instance details> -- Failover状态变为reconf-slaves状态时

+failover-detected <instance details> -- 当failover发生时

+slave-reconf-sent <instance details> -- sentinel发送SLAVEOF命令把它重新配置时

+slave-reconf-inprog <instance details> -- slave被重新配置为另外一个master的slave，但数据复制还未发生时。

+slave-reconf-done <instance details> -- slave被重新配置为另外一个master的slave并且数据复制已经与master同步时。

-dup-sentinel <instance details> -- 删除指定master上的冗余sentinel时 (当一个sentinel重新启动时，可能会发生这个事件).

+sentinel <instance details> -- 当master增加了一个sentinel时。

+sdown <instance details> -- 进入SDOWN状态时;

-sdown <instance details> -- 离开SDOWN状态时。

+odown <instance details> -- 进入ODOWN状态时。

-odown <instance details> -- 离开ODOWN状态时。

+new-epoch <instance details> -- 当前配置版本被更新时。

+try-failover <instance details> -- 达到failover条件，正等待其他sentinel的选举。

+elected-leader <instance details> -- 被选举为去执行failover的时候。

+failover-state-select-slave <instance details> -- 开始要选择一个slave当选新master时。

no-good-slave <instance details> -- 没有合适的slave来担当新master

selected-slave <instance details> -- 找到了一个适合的slave来担当新master

failover-state-send-slaveof-noone <instance details> -- 当把选择为新master的slave的身份进行切换的时候。

failover-end-for-timeout <instance details> -- failover由于超时而失败时。

failover-end <instance details> -- failover成功完成时。

switch-master <master name> <oldip> <oldport> <newip> <newport> -- 当master的地址发生变化时。通常这是客户端最感兴趣的消息了。

+tilt -- 进入Tilt模式。

-tilt -- 退出Tilt模式。

[复制代码](javascript:void(0);)

**TILT 模式**

redis sentinel非常依赖系统时间，例如它会使用系统时间来判断一个PING回复用了多久的时间。  
然而，假如系统时间被修改了，或者是系统十分繁忙，或者是进程堵塞了，sentinel可能会出现运行不正常的情况。  
当系统的稳定性下降时，TILT模式是sentinel可以进入的一种的保护模式。当进入TILT模式时，sentinel会继续监控工作，但是它不会有任何其他动作，它也不会去回应is-master-down-by-addr这样的命令了，因为它在TILT模式下，检测失效节点的能力已经变得让人不可信任了。  
如果系统恢复正常，持续30秒钟，sentinel就会退出TITL模式。

**-BUSY状态**

注意：该功能还未实现。

当一个脚本的运行时间超过配置的运行时间时，sentinel会返回一个-BUSY 错误信号。如果这件事发生在触发一个failover之前，sentinel将会发送一个SCRIPT KILL命令，如果script是只读的话，就能成功执行。