《Redis官方文档》sentinel

[原文链接](https://github.com/antirez/redis-doc/blob/master/topics/sentinel.md)

**Redis Sentinel 文档**

Redis Sentinel为Redis提供了高可用解决方案。实际上这意味着使用Sentinel可以部署一套Redis，在没有人为干预的情况下去应付各种各样的失败事件。

Redis Sentinel同时提供了一些其他的功能，例如：监控、通知、并为client提供配置。

下面是Sentinel的功能列表：

* 监控（Monitoring）：Sentinel不断的去检查你的主从实例是否按照预期在工作。
* 通知（Notification）：Sentinel可以通过一个api来通知系统管理员或者另外的应用程序，被监控的Redis实例有一些问题。
* 自动故障转移（Automatic failover）：如果一个主节点没有按照预期工作，Sentinel会开始故障转移过程，把一个从节点提升为主节点，并重新配置其他的从节点使用新的主节点，使用Redis服务的应用程序在连接的时候也被通知新的地址。
* 配置提供者（Configuration provider）：Sentinel给客户端的服务发现提供来源：对于一个给定的服务，客户端连接到Sentinels来寻找当前主节点的地址。当故障转移发生的时候，Sentinels将报告新的地址。

**Sentinel的分布式特性**  
Redis Sentinel是一个分布式系统，Sentinel运行在有许多Sentinel进程互相合作的环境下，它本身就是这样被设计的。有许多Sentinel进程互相合作的优点如下：

1. 当多个Sentinel同意一个master不再可用的时候，就执行故障检测。这明显降低了错误概率。
2. 即使并非全部的Sentinel都在工作，Sentinel也可以正常工作，这种特性，让系统非常的健康。

所有的Sentinels，Redis实例，连接到Sentinel和Redis的客户端，本身就是一个有着特殊性质的大型分布式系统。在这篇文章中，我将逐步地介绍这些概念，最开始是一些基本的信息来理解Sentinel的基本属性，后面是更复杂的信息来理解Sentinel是怎么工作的。

**快速开始**

**获取 Sentinel**

当前版本的Sentinel的被称为 **Sentinel 2**。它使用更强更简单的预测算法重写了Sentinel的初始化实现（文章的后面将会解释）。

Redis Sentinel 的一个稳定版本是随着Redis2.8和3.0一起的。这两个是Redis最新的稳定版。

新的进展在*unstable*分支下进行，一旦新的特性是稳定的，就会被合并到2.8和3.0分支。

和Redis 2.6一起的Redis Sentinel版本1，是过时的。我们不该使用它。

**运行Sentinel**

如果你使用redis-sentinel可执行文件，你可以使用下面的命令来运行Sentinel：

redis-sentinel /path/to/sentinel.conf

另外，你可以直接使用redis-server并以Sentinel模式来启动：

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

两种方式是一样的。

不管咋样，使用一个配置文件来运行Sentinel是必须的，这个文件被系统使用来存储当前状态，如果重启，这些状态会被重新载入。如果没有配置文件或者配置文件的路径不对，Sentinel将会拒绝启动。

默认情况下，Sentinels监听TCP端口26379，所以为了让Sentinels运行，你的机器的26379端口必须是打开的，用来接收其他Sentinel实例的连接，否则，Sentinels不能互相交流，也不知道该干什么，也不会执行故障转移。

**部署之前了解关于Sentinel的基本东西**

1. 一个健康的集群部署，至少需要三个Sentinel实例
2. 三个Sentinel实例应该被放在失败独立的电脑上或虚拟机中，比如说不同的物理机或者在不同的可用区域上执行的虚拟机。
3. Sentinel + Redis 分布式系统在失败期间并不确保写入请求被保存，因为Redis使用异步拷贝。可是有很多部署Sentinel的 方式来让窗口把丢失写入限制在特定的时刻，当然也有另外的不安全的方式来部署。
4. 如果你在开发环境中没有经常测试，或者在生产环境中也没有，那就没有高可用的设置是安全的。你或许有一个错误的配置而仅仅只是在很晚的时候才出现（凌晨3点你的主节点宕掉了）。
5. Sentinel，Docker ，其他的网络地址转换表，端口映射 使用应该很小心的使用：Docker执行端口重新映射，破坏Sentinel自动发现另外的Sentinel进程和一个主节点的从节点列表。在文章的稍后部分查看更过关于Sentinel和Docker的信息。

**Sentinel配置**

Redis源码中包含一个名为sentinel.conf的文件，是一个你可以用来配置Sentinel的示例配置文件。一个典型的最小配置文件像下面这样：

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

你仅仅只需要指定要监控的主节点，并给每个单独的主节点一个不同的名称。不需要指定从节点，从节点会被自动发现。Sentinel将会根据从节点额外的信息自动更新配置（为了在重启时保留信息）。在故障转移中每当一个从节点被提升为主节点或者当一个新的Sentinel被发现的时候，配置信息也被重新写入。

示例配置在上面，监控两个Redis实例集合，每个集合由一个主节点和不明确数量的从节点组成。一个集合叫做mymaster，另外一个叫做resque。

sentinel monitor参数的意思在下面

sentinel monitor <master-group-name> <ip> <port> <quorum>

为了更加清晰明了，让我们一行一行来检查配置选项的意思：

第一行用来告诉Redis监控一个叫做*mymaster*的主节点，地址是 127.0.0.1 端口号是6379，并且有2个仲裁机器。所有的意思都很明显，但是除了这个**quorum** 参数：

* **quorum**是 需要同意主节点不可用的Sentinels的数量
* 然而**quorum**仅仅只是用来检测失败。为了实际的执行故障转移，Sentinels中的一个需要被选定为leader并且被授权进行操作，这仅仅发生在大多数Sentinels进行投票的时候。

比如如果你有五个Sentinel进程，对于一个主节点quorum被设置为2，下面是发生的事情：

* 同时有两个Sentinels同意主节点不可用，其中的一个将会尝试开始故障转移。
* 如果至少有三个Sentinels是可用的，故障转移将会被授权并且开始

实际中，这意味着在失败时，如果大多数的Sentinel进程没有同意，Sentinel永远不会开始故障转移。

**其他的Sentinels选项**

其他的选项几乎都是如下形式：

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

用途如下：

down-after-milliseconds：当一个实例失去联系（要么不回复我们的请求，要么回复一个错误）超过了这个时间（毫秒为单位），Sentinel就开始认为这个实例挂掉了。

parallel-syncs：设置的从节点的数量，这些从节点在一次故障转移过后可以使用新的主节点进行重新配置。数量越少，完成故障转移过程将花费更多的时间，如果从节点为旧的数据提供服务，你或许不想所有的从节点使用主节点进行重新同步。复制进程对于从节点来说大部分是非阻塞的，还是有一个时刻它会停下来去从主节点加载数据。你或许想确保一次只有一个从节点是不可达的，可以通过设置这个选项的值为1来完成。

别的选项在文章的其他部分进行描述。

所有的配置参数都可以在运行时使用SENTINEL SET命令进行更改，查看 **Reconfiguring Sentinel at runtime**章节获取更多内容**。**

**Sentinel部署示例**

现在你已经知道了Sentinel的基本信息，你或许想知道哪里放置你的Sentinel进程，需要多少个Sentinel进程等等。这个章节给出了几个部署的例子。

为了以图形（*graphical* ）格式展示配置示例，我们使用ASCII艺术。下面是不同的符号的意思：

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

| 这是一个独立电脑 |

| 或者VM。我们称它为 |

| “box” |

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

我们把我们想要运行的东西写到boxes里：

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

| Redis master M1 |

| Redis Sentinel S1 |

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

不同的box之间通过一条线连接，表示他们之间可以互相交流：

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

| Sentinel S1 |---------------| Sentinel S2 |

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

中断的线条表示不同的网络分区：

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

| Sentinel S1 |------ // ------| Sentinel S2 |

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

同时还要注意：

* 主节点称为M1，M2，M3，…，Mn。
* 从节点称为R1，R2，R3，…，Rn。
* Sentinels称为S1，S2，S3，…，Sn。
* 客户端称为C1，C2，C3，…，Cn。
* 当一个实例因为Sentinels的行为转换角色，我们把它放在方括号里，所以[M1]表示一个实例现在是主节点。

注意永远不要设置只有两个Sentinels，因为开始一个故障转移，Sentinels总是需要和大多数Sentinels交流。

**示例1：仅仅只有两个Sentinels，永远不要这么做**

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

| M1 |---------| R1 |

| S1 | | S2 |

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

Configuration: quorum = 1

在这个设置中，如果M1宕掉了，R1将会被提升至主节点，因为两个Sentinels将会达成一致（显然把quorum设置为1），并且授权开始一个故障转移因为大多数是两个。显然，表面上可以工作，但是请检查下一个点来看看为什么这种设置是不可以的。

如果M1的box停止工作，M1也会停止。运行在另外一个box中的S2将不会被授权进行故障转移，所以系统将不可用。

注意，需要大多数是为了应付不同的故障，最新的配置稍后会传播给所有的Sentinels。同时注意在上述设置中单独一边的故障转移能力，没有任何协议，将是非常危险的：

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

| M1 |----//-----| [M1] |

| S1 | | S2 |

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

在上面的配置中，我们完美对称地创建了两个主节点（假设S2在没有授权的情况下可以进行故障转移），客户端或许会不确定写往哪一边，并且没有办法理解当分区治愈时候哪边的配置是正确的。

所以请至少部署三个Sentinels在三个不同的box当中。

**示例2：三个box的基本设置**

这是一个非常简单的设置，拥有更加安全的优点。它是基于三个boxes的，每个box运行一个Redis进程和Sentinel进程。

+----+

| M1 |

| S1 |

+----+

|

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

| R2 |----+----| R3 |

| S2 | | S3 |

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

Configuration: quorum = 2

如果M1挂掉，S2和S3将认同这次失败，并且能授权开始一次故障转移，这样使客户端可以继续使用。

在每一个Sentinel设置中，Redis是异步复制的，总是有丢失一些写入数据的危险，因为当一个从节点被提升为主节点的时候一个写入确认还没有到达。然而在上面的设置中，还有一种更加危险的情况，由于客户端和一个老的主节点在一个网络分区中，就像下面这样：

+----+

| M1 |

| S1 | <- C1 (writes will be lost)

+----+

|

/

/

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

| [M2] |----+----| R3 |

| S2 | | S3 |

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

在这种情况下，网络分区把旧的主节点[M1]给孤立了，所以从节点R2被提升为主节点。然而，像客户端C1，和旧的主节点在同一个网络分区中，或许继续像旧的主节点写入数据。当分区治愈，这些数据将永久丢失，这个旧得主节点将会被重新配置，作为新的主节点下的一个从节点，并丢弃它自己的数据。

可以使用下面的Redis复制特性减轻这个问题，如果一个主节点发现它不再能够把它的写入请求发送给指定数量的从节点，它就停止接受写入请求。

min-slaves-to-write 1

min-slaves-max-lag 10

当上面的配置应用于一个Redis实例。Redis发现它不能写入至少一个1从节点，作为主节点的Reids将会停止接受写入请求。由于复制是异步，不能写入也意味着从节点也是断开的，或者超过了指定的max-lag秒数没有发送异步回应。

在上面的示例中，使用这个配置的旧的主节点M1，在10秒过后就不可用了。当分区治愈，Sentinel配置将会统一为新的，客户端C1将获取到一个有效的配置并且继续。

然而天下没有免费的午餐，在这种改进下，如果两个从节点挂掉了，主节点将会停止接收写入请求，这就是一个权衡。

示例3：Sentinel在客户端所在的box中

有时候，我们只有两个Redis box是可用的，一个给主节点，一个给从节点。在那种情况下，示例2中的配置是不可行的，我们可以采取下面的方法，Sentinels被放置在客户端所在的地方：

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

| M1 |----+----| R1 |

| S1 | | | S2 |

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

|

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

| | |

| | |

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

| C1 | | C2 | | C3 |

| S1 | | S2 | | S3 |

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

Configuration: quorum = 2

在这种设置下，Sentinels的视角和客户端是 一样的：如果大部分的客户端认为一个主节点是可用的，它就是可用的。这里的C1，C2，C3是一般的客户端， 并不意味着C1是连接到Redis的单个客户端，它更像一个应用服务器，一个Redis app，或者类似的东西。

如果M1和S1所在的box挂掉了，故障转移将会进行，但是很明显的看到不同的网络分区将导致不同的行为。比如说，如果客户端和Redis服务断开连接，Sentinel将不会被设置，因为Redis的主节点和从节点都是不可用的。

注意如果C3和M1在一个分区，我们有了一个和示例2中描述的类似的问题，不同的是，这里我们没有办法打破对称，因为只有一个主节点和从节点，所以主节点不会停止接收请求。

所以这是一个有效的设置，但是实例2中的设置更有优势，比如Redis高可用系统，Redis运行在同一个box中，更容易被管理，并且可以限制在小部分的分区中主节点接收写入请求的时间。

**示例4：Sentinel 客户端 这一边少于三个客户端**

示例3描述的设置中，如果客户端这一边的box少于不够三个，这个 设置就不能使用。在这种情况下，我们需要借助混合设置，像下面这样：

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

| M1 |----+----| R1 |

| S1 | | | S2 |

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

|

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

| |

| |

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

| C1 | | C2 |

| S3 | | S4 |

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

Configuration: quorum = 3

这和示例3中的设置非常相似，但是这里我们在可用的四个box中运行了四个Sentinel。如果主节点M1变成不可用节点，其他三个Sentinel将执行故障转移。

理论上，当移除S2和S4正在运行的box，这个设置可以工作，把quorum设置为2。然而，在应用层没有高可用的系统，想在Redis这一边得到高可用是不太可能的。

**Sentinel，Docker,NAT 和可能的问题**

Docker使用被称为端口映射的技术：与一个程序认为他使用的端口相比，运行在Docker容器里面的程序可能被暴露在不同的端口上。为了运行多个容器在相同的服务器上同时使用同一个端口，这是非常有用的。

Docker不是唯一会发生这件事情的软件系统，也有其他的网络地址转换设置导致端口是被重映射，并且有时候没有端口，只有IP地址。

端口和地址重映射在两个方面制造了与Sentinel有关的问题：

1. Sentinel的自动发现服务将停止工作，因为它使基于每个Sentinel 往它监听的端口和IP地址广播*hello*消息来实现的。但是Sentinels没有办法来理解端口和IP地址被重映射了，所以他会宣布它和其他的Sentinels的连接是不正常的。
2. 在一个主节点的INFO输出中，从节点 被列出来也是类似的方式：主节点检查远端对等的TCP连接来发现地址，在握手过程中，从节点自己广告他的端口，然而由于相同的原因，端口或许是错误的。

因为Sentinels自动发现从节点使用主节点的INFO输出信息，发现的从节点是不可达的，并且Sentinel将永远不会开始故障转移，因为从系统的观点来看，没有好的从节点，所以目前没有方式监控使用Docker部署的主节点和从节点实例，除非你通知Docker以1:1映射端口。

对于第一个问题，万一你想使用Docker运行一堆Sentinel实例，你可以使用下面的两个Sentinel配置，为了强迫Sentinel宣布一个指定的端口和IP：

sentinel announce-ip <ip>

sentinel announce-port <port>

注意，Docker可以运行*host networking*模式。这就不会有问题因为端口不会被重新映射。

**快速教程**

在文章接下来的部分中，所有的说明都是关于Sentinel API，配置和语义。对于想尽快上手的人，这部分的教程展示了三个Sentinel怎么配置和交互。

现在我假设三个实例分别在端口5000、5001、5002上。我也假设你在6379上有一个主节点Redis实例，6380上有一个从节点实例。在本教程中我们将使用IPV4回调地址127.0.0.1，假设你在你的电脑上运行了 模拟环境。

三个Sentinel配置文件应该看起来像下面这样：

port 5000

sentinel monitor mymaster 127.0.0.1 6379 2

sentinel down-after-milliseconds mymaster 5000

sentinel failover-timeout mymaster 60000

sentinel parallel-syncs mymaster 1

另外的两个配置文件也是相同的，但是使用5001,5002作为端口号。

上面的配置中需要注意的一些事情：

* 主节点集群称为mymaster，它定义了主节点和它的从节点。因为每个*master set*有一个不同的名称，Sentinel能同时监控不同的主节点和从节点的集合。
* quorum被设置为2。
* down-after-milliseconds的值是5000毫秒，就是5秒钟，所以在这个时间内一旦我们不能收到回复，主节点将发现失败。

一旦你启动了三个Sentinels，可以看到他们打印的一些信息：

+monitor master mymaster 127.0.0.1 6379 quorum 2

这是一个Sentinel事件，如果你SUBSCRIBE 了指定名称的事件，你可以收到这种事件通过发布/订阅。

Sentinel在故障检测和故障转移中生成和打印不同的事件。

**询问Sentinel关于主节点的状态**

Sentinel开始启动的时候，要做的事情是检查主节点的监控是否正常：

$ redis-cli -p 5000

127.0.0.1:5000> sentinel master mymaster

1) "name"

2) "mymaster"

3) "ip"

4) "127.0.0.1"

5) "port"

6) "6379"

7) "runid"

8) "953ae6a589449c13ddefaee3538d356d287f509b"

9) "flags"

10) "master"

11) "link-pending-commands"

12) "0"

13) "link-refcount"

14) "1"

15) "last-ping-sent"

16) "0"

17) "last-ok-ping-reply"

18) "735"

19) "last-ping-reply"

20) "735"

21) "down-after-milliseconds"

22) "5000"

23) "info-refresh"

24) "126"

25) "role-reported"

26) "master"

27) "role-reported-time"

28) "532439"

29) "config-epoch"

30) "1"

31) "num-slaves"

32) "1"

33) "num-other-sentinels"

34) "2"

35) "quorum"

36) "2"

37) "failover-timeout"

38) "60000"

39) "parallel-syncs"

40) "1"

像你所见的，它打印了主节点的一些信息。有几个是我们特别有兴趣的：

1. num-other-sentinels 是2，所以我们知道对于这个主节点Sentinel已经发现了两个以上的Sentinels。如果你检查日志，你可以看到+sentinel事件发生。
2. flags是master。如果主节点挂掉了，我们可以看到s\_down或者o\_down标志。
3. num-slaves现在是1，所以Sentinel发现有一个从节点。

为了探测关于这个实例更多的信息，你可以尝试下面的两个命令：

SENTINEL slaves mymaster

SENTINEL sentinels mymaster

第一个将提供关于从节点类似的信息，第二个是关于另外的Sentinels。

**获取当前主节点的地址**

Sentinel也作为一个配置提供者，提供给客户端它们想连接的主节点和从节点的集群。因为可能的故障转移和重配置，客户端不知道一个集群实例内当前的活着的主节点，所以Sentinel提供了一个API：

127.0.0.1:5000> SENTINEL get-master-addr-by-name mymaster

1) "127.0.0.1"

2) "6379"

**故障转移测试**

现在我们部署Sentinel可以被测试了。我们可以杀死主节点然后查看配置变化。做我们可以做的：

redis-cli -p 6379 DEBUG sleep 30

这个命令让我们的主节点变为不可达，睡眠30秒，它基本上模拟了主节点挂掉的一些原因。

如果你检查Sentinel的日志，你应该能看到许多动作：

1. 每个Sentinel发现了主节点挂掉了并有一个+sdown事件
2. 这个事件稍候升级到+odown，意味着大多数Sentinel已经同意了主节点是不可达的。
3. Sentinels开始投票一个Sentinel开始并尝试故障转移
4. 故障转移开始

如果你重新询问mymaster的当前主节点的地址，这次我们会得到一个不同的回复：

127.0.0.1:5000> SENTINEL get-master-addr-by-name mymaster

1) "127.0.0.1"

2) "6380"

目前为止一切都很顺利，现在你可以创建你自己的Sentinel部署或者阅读更多来理解Sentinel的命令和内部原理。

**Sentinel API**

Sentinel提供了一个API，可以用来检查它的状态，检查主节点和从节点的健康，订阅具体的通知并在运行时改变Sentinel的配置。

默认情况下Sentinel使用TCP端口号26379。Sentinels接收使用Redis的协议命令，所以你可以使用redis-cli或者其他未修改的Redis客户端来和Sentinel交流。

直接查询一个Sentinel来检查所监控的Redis实例的状态，看看另外的Sentinels所知道是可能的。有两种方式，使用发布/订阅，每当一些事件发生，比如说一次故障转移，或一个实例发生错误等，都可能接收到一个从Sentinels推送过来的通知。

**Sentinel命令**

下面是可以接收的命令列表，没有覆盖到那些用来改变Sentinel配置的命令：

* PING 这个命令仅仅返回PONG。
* SENTINEL masters  展示监控的主节点和它们的状态列表
* SENTINEL master <master name> 展示指定的主节点的信息
* SENTINEL salves <master name> 展示这个主节点的从节点，以及它们的状态
* SENTINEL  sentinels <master name> 展示这个主节点的sentinel实例，以及它们的状态
* SENTINEL  get-master-addr-by-name  <master name> 返回主节点的IP和端口号。如果这个主节点的一次故障转移正在进行，就返回提升的从节点的IP和端口号
* SENTINEL reset <pattern> 这个命令将会根据匹配的名称重置主节点，pattern参数是通配符（glob-style）类型，重置进程清除主节点中之前的所有状态，并且移除主节点发现和关联的从节点和sentinel。
* SENTINEL failover <master name> 如果主节点不可达，强制开始故障转移，不需要另外的Sentinels同意。
* SENTINEL ckquorum <master name> 检查当前的Sentinel配置对于主节点的故障转移是否能达到仲裁人数，并且大多数是需要的来授权故障转移。这个命令应该在监控系统中使用来检查一个Sentinel部署是否正常。
* SENTINEL flushconfig  强制Sentinel重新写入它的配置到磁盘上，包括当前Sentinel状态。通常，每次当它状态里的一些东西改变，Sentinel就会重写配置信息。然而有时候配置文件会丢失，由于错误的操作、磁盘故障、包升级脚本、或配置管理。在那种情况下，强制Sentinel重写它的配置文件是容易的。甚至之前的配置文件完全丢失，这个命令也能很好的工作。

**运行时重新配置Sentinel**

从Redis 2.8.4开始，Sentinel提供了一个API为了增加、移除或者改变一个给定的主节点的配置。注意如果你有多个sentinels，为了工作正常，你应该改变所有的Redis Sentinel 实例。这意味着改变单个Sentinel的配置不会把变化发送给在网络中另外的Sentinels.

下面是SENTINEL自命令列表，用来更新一个Sentinel实例的配置：

* SENTINEL MONITOR <name> <ip> <port> <quorum> 这个命令告诉Sentinel开始监控一个指定名称、IP、端口号、quorum的主节点，它和sentinel.conf配置文件中的sentinel monitor配置指令是完全相同的，不同的是这里不能使用主机名作为IP，需要提供一个IPV4或IPV6地址。
* SENTINEL REMOVE <name> 用来移除指定的主节点：主节点不再被监控，并且将被从Sentinel的内部状态中被完全移除，所以不会被SENTINEL masters列出。
* SENTINEL SET <name> <option> <value> SET命令和Reids的CONFIG SET指令非常相似，被用来改变一个指定主节点的配置参数。多个选项-值可以被指定。所有通过sentinel.conf配置的参数可以使用SET命令重新配置。

下面是SENTINEL SET命令的一个例子，为了修改一个名为objects-cache的主节点的down-after-milliseconds配置：

SENTINEL SET objects-cache-master down-after-milliseconds 1000

正如我们提到的，SENTINEL SET可以被用来设置所有的在启动配置文件中被设置的参数。而且，还可以仅仅改变主节点的quorum配置，而不需要使用SENTINEL REMOVE和SENTINEL MONITOR来删除或者增加主节点，只需要使用：

SENTINEL SET objects-cache-master quorum 5

注意，没有等价的GET命令，因为SENTINEL MASTER以一种易于解析的格式提供了所有的配置参数。

**添加和移除sentinels**

添加一个新的sentinel到你的部署中是很容易的一个过程，因为Sentinel有自动发现机制。所有的你需要做的事情是开启一个新的Sentinel来监控当前的主节点。10秒过后，Sentinel将获取到其他的Sentinels列表和当前主节点的从节点。

如果你想一次性增加多个Sentinels，建议你一个接一个的增加，等所有的Sentinels已经知道第一个再添加另一个。在添加的新的Sentinels过程中错误有可能发生，在这时候保证在一次网络分区内中大部分是可用是很有用的。

在没有网络分区时，通过在30秒后增加每个新的节点，这是很容易实现的。

最后，可以使用SENTINEL MASTER mastername命令来检查是否全部Sentinels都同意了监控主节点的Sentinels的总数。

移除一个Sentinel稍微复杂一点：**Sentinels永远不会忘记已经看到的Sentinels，**甚至他们在相当长的一段时间内不可达，因为我们不想动态的改变授权一次故障转移和创建新的配置所需要的大多数。在没有网络分区的说话，需要执行下面的步骤来移除一个Sentinel：

1. 停止你想要移除的Sentinel的进程
2. 发送一个SENTINEL RESET \*命令到其他的Sentinel实例，相继的，两次发送到实例之间至少等待30秒
3. 检查所有的Sentinels赞同的当前存活的Sentinels的数量，通过检查每个SENTINEL MASTER mastername的输出。

**移除旧的主节点或不可达的从节点**

Sentinels永远不会忘记一个主节点的从节点，甚至当他们很长时间都不可达。这是很有用的，因为在一次网络分区或失败事件发生后，Sentinels应该能正确地重新配置一个返回的从节点。

而且，在故障转移发生之后，被故障转移的主节点实际上被添加为新的主节点的从节点，一旦它可用的时候，这种方式将重新配置来复制新的主节点。

然而有时候你想从Sentinels监控的从节点列表中永久的移除一个从节点。

为了做这件事，你需要发送一个SENTINEL RESET mastername命令给所有的Sentinels：它们将在十秒后刷新从节点列表，只添加当前主节点的INFO输出中正确的复制列表。

**发布/订阅消息**

一个客户端能使用一个Sentinel作为一个Redis兼容的发布/订阅服务器，为了SUBSCRIBE或者PSUBSCRIBE到指定频道，获取指定事件通知。

频道的名称和事件的名称是一样的。比如说名称为+sdown的频道将收到所有的关于实例进入SDOWN 条件的通知。

使用 PSUBSCRIBE \* 订阅来获取所有的消息。

下面是一个频道列表，以及使用API，你可以接收到的消息格式。第一个词是频道/事件名称，剩余部分是数据格式。

注意，指定*instance details*的地方意味着提供了下面的参数用于表示目标实例：

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

标识主节点的部分（从@开始到结束）是可选的，只有实例本身不是主节点的时指定。

* **+reset-master** <instance details> — 主节点被重置。
* **+slave** <instance details> — 一个新的从节点被发现和关联。
* **+failover-state-reconf-slaves** <instance details> — 故障转移状态被转换为reconf-slaves状态。
* **+failover-detected** <instance details> — 另一个Sentinel开始了故障转移或者其他的外部实体被发现（一个关联的从节点变为主节点）。
* **+slave-reconf-sent** <instance details> — 为了给新的从节点重新配置，sentinel 中的leader发送SLAVEOF命令到这个实例。
* **+slave-reconf-inprog** <instance details> –从节点被重新配置展示一个主节点的从节点，但是同步过程尚未完成。
* **+slave-reconf-done** <instance details> — 从节点现在和主节点是同步的。
* **-dup-sentinel** <instance details> –指定的主节点，一个或者多个sentinels被 移除，因为是重复的。
* **+sentinel** <instance details> — 这个主节点的一个新的sentinel被发现和关联。
* **+sdown** <instance details> — 指定的实例现在处于主观下线状态。
* **-sdown** <instance details> — 指定的实例不再处于主观下线状态。
* **+odown** <instance details> — 指定的实例现在处于客观下线状态。
* **-odown** <instance details> — 指定的实例现在不处于客观下线状态。
* **+new-epoch** <instance details> — 当前时间被更新。
* **+try-failover** <instance details> — 准备新的故障转移，等待大多数的选举。
* **+elected-leader** <instance details> — 赢得了选举，开始故障转移。
* **+failover-state-select-slave** <instance details> — 新的故障转移状态是select-slave：我们 正在寻找合适提升为主节点的从节点。
* **no-good-slave** <instance details> — 没有合适进行提升的从节点。一般会在稍后重试，但是这或许会改变并且终止故障转移。
* **selected-slave** <instance details> — 我们找到了指定的从节点来进行提升。
* **failover-state-send-slaveof-noone** <instance details> — 我们尝试重新配置这个提升后的主节点，等待它切换。
* **failover-end-for-timeout** <instance details> — 故障转移由于超时而停止，无论如何从节点最后被配置为复制新的主节点。
* **failover-end** <instance details> — 故障转移由于成功而停止，所有的从节点被配置为复制新的主节点。
* **switch-master** <master name> <oldip> <oldport> <newip> <newport> — 配置改变后，主节点新的IP和地址都是指定的。这是大多数外部用户感兴趣的消息。
* **+tilt** — 进入Tilt模式。
* **-tilt** — 退出Tilt模式。

**-BUSY状态的处理**

当一个Lua脚本的运行时间超过了配置中指定的Lua脚本时间限制，Redis实例将返回 -BUSY错误。当这个发生的时候，在触发故障转移之前 Redis Sentinel将尝试发送SCRIPT KILL命令，如果脚本是只读的，就会成功。

如果在这个尝试后，实例仍然处于失败情况，它最后会开始故障转移。

**从节点优先**

Redis实例有个配置参数叫slave-priority。这个信息在Redis从节点实例的INFO输出中展示出来，并且Sentinel使用它来选择一个从节点在一次故障转移中：

1. 如果从节点的优先级被设置为0，这个从节点永远不会被提升为主节点。
2. Sentinel首选一个由更低（ *lower*）优先级的从节点。

比如在当前主节点的同一个数据中心有一个从节点S1，并且有另外的从节点S2在另外的数据中心，可以将S1优先级设置为10，S2优先级设置为100，如果主节点挂掉了并且S1和S2都是可用的，S1将是首选的。

查看关于从节点选举的更多信息，请查看本文章的**slave selection and priority章节**。

**Sentinel和Redis权限**

当主节点被配置为从客户端需要密码，作为一个安全措施，从节点也需要知道这个密码为了主节点认证并且创建主-从连接用于异步复制协议。

使用下列的配置选项来实现：

* requirepass 在主节点中，为了设置认证密码，并且确保实例不会处理来自没有认证的客户端的请求。
* masterauth 在从节点中，为了取得主节点的认证，来从主节点正确的复制 数据。

当Sentinel使用的时候，没有一个单独的主节点，因为一次故障转移过后，从节点将扮演主节点的角色，并且老的主节点被重新配置作为一个从节点，所以你要做的是在全部的实例中设置上面的选项，包括主节点和从节点。

这通常是一个理智的设置，因为你不想要仅仅在主节点中保护你的数据，在从节点中有同样的数据。

然而，在罕见的情况下，你需要一个从节点是可进入的而不需要认证，你可以设置一个优先级为0的从节点来实现，阻止这个从节点被提升为主节点，配置这个从节点的masterauth选项，不要使用requirepass选项，以便数据可以被读在没有认证的情况下。

**Sentinel 客户端实现**

Sentinel需要显式的客户端支持，除非系统配置为执行脚本来执行一个透明的重定向对于所有的主节点实例的请求（虚拟IP或类似的系统）。可以参考文档[Sentinel clients guidelines](https://github.com/antirez/redis-doc/blob/master/topics/sentinel-clients)。

**更高级的概念**

下面的章节是关于Sentinel怎么工作的一些细节，没有付诸于实现的想法和算法在文章的最后章节。

**SDOWN和ODOWN失败状态**

Redis Sentine有两个不同概念的下线，一个被称为*主观下线（Subjectively Down ）*条件（SDOWN），是一个本地Sentinel实例下线条件。另一个被称为*客观下线（Objectively Down ）*条件（ODOWN），是当足够的Sentinels具有SDOWN条件就满足ODOWN，并且从其他的Sentinels使用SENTINEL is-master-down-by-addr命令得到反馈。

从一个Sentinel的角度来看，满足一个SDOWN条件就是在指定的时间内对于PING请求不能收到有效的回复，这个时间在配置文件中是is-master-down-after-milliseconds参数。

一个PING请求可接受的回复是下列之一：

* 回复+PONG。
* 回复 -LOADING错误。
* 回复-MASTERDOWN错误。

其他的回复（或根本没有回复）被认为是无效的。注意一个合理的主节点在INFO输出中通知他自己是一个从节点被认为是下线的。

注意SDOWN需要在配置中整个的时间间隔都没有收到有效的回复，因此对于实例如果时间间隔是30000毫秒，并且我们每隔29秒收到有效的回复，这个实例就被认为在工作。

SDOWN还不够触发故障转移：它仅仅意味着一个单独的Sentinel相信一个Redis实例不可达。要触发故障转移，必须达到ODOWN状态。

从SDOWN转换到ODOWN，没有使用强一致性算法，而仅仅是gossip的形式：如果一个Sentinel在一个给定的时间范围内从足够的Sentinels 得到一个报告说一个主节点没有在工作，SDOWN被提升为ODOWN。如果这个确认稍候消失，这个标识也会清除。

一个更加严格的授权是使用大多数需要为了真正的开始故障转移，但是在达到ODOWN状态之前不会触发故障转移。

ODOWN条件只适用于主节点。对于其他类型的实例，Sentinel不需要采取行动，所以对于从节点和其他的sentinels来说ODOWN状态永远不可能达到，而仅仅只有SDOWN状态。

然而SDOWN也有语义的影响，比如一个从节点在SDOWN状态不会被选举来提升来执行一个故障转移。

**Sentinels和从节点自动发现**

Sentinels和其他的Sentinels保持连接为了互相之间检查是否可达和交换消息。然而你不需要在每个运行的Sentinel 实例中配置其他的Sentinel地址列表，Sentinel使用Redis实例的发布/订阅能力来发现其他的监控相同的主节点和从节点的Sentinels。

通过往名称为\_\_sentinel\_\_:hello的通道发送*hello消息（hello messages）*来实现这个特性。

同样的，你不需要配置一个主节点关联的从节点的列表，Sentinel也会自动发现这个列表通过问询Redis：

* 每隔两秒，每个Sentinel向每个监控的主节点和从节点的发布/订阅通道\_\_sentinel\_\_:hello来公布一个消息，宣布它自己的IP，端口，id。
* 每个Sentinel都订阅每个主节点和从节点的发布/订阅通道\_\_sentinel\_\_:hello，寻找未知的sentinels。当新的sentinels被检测到，他们增加这个主节点的sentinels。
* Hello消息也包含主节点的全部配置信息，如果接收的Sentinel有一个更旧的配置，它会立即更新它的配置。
* 在增加一个主节点的新的sentinel之前，Sentinel总是要检查是否已经有一个有相同的id、地址的sentinel。在这种情况下，所有匹配的sentinels被移除，新的被增加。

**故障转移之外重新配置**

即使没有故障转移，Sentinels将尝试设置当前的配置到监控的实例上面。

特别的：

* 从节点声称为主节点，将被作为从节点配置来复制当前的主节点。
* 从节点连接了一个错误的主节点，也会被重新配置来复制正确的主节点。

Sentinels重新配置从节点，错误的配置在一段时间内应该被观察到，比在广播新的配置的时候要好得多。

这个阻止了有一个过时配置（比如说从一个分区中重新加入）的Sentinels 在收到更新之前去交换从节点的配置。

同样注意：

* 主节点的故障转移被重新配置作为从节点当他们返回可用的时候
* 在一个网络分区中，从节点一旦可达，被重新配置。

本章最重要的教训就是：Sentinels是每个进程总是尝试去把最后的配置施加到监控的实例上的一个系统。

**从节点选举和优先级**

当一个Sentinel实例准备执行故障转移，因为主节点在ODOWN状态下并且Sentinel从大多数已知的Sentinel实例中收到了授权开始故障转移，一个合适的从节点要被选举出来。

从节点选举过程评估从节点的下列信息：

1. 与主节点断开的时间
2. 从节点优先级
3. 复制偏移处理
4. 运行ID

一个从节点被发现从主节点断开超过主节点配置超时（down-after-milliseconds 选项）时间十倍以上，加上从正在执行故障转移的Sentinel的角度看主节点不可用的时间，将被认为是不合适的并且会被跳过。

在更严格的条件下，一个从节点的INFO输出建议了从主节点断开超过：

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

被认为是不可靠的并且会被无视。

从节点选举只会考虑通过了上述测试的从节点，并根据上面的条件进行排序，以下列顺序：

1. 根据Redis实例中的redis.conf文件中配置的slave-priority进行排序，更低的优先级会被优先。
2. 如果优先级相同，检查复制偏移处理，从主节点收到更加新的数据的从节点会被选择。
3. 如果多个从节点有相同的优先级和数据偏移，执行进一步检查，选择有着更小运行ID的从节点。有一个更小的ID并不是具有正真的优点，但是对于从节点选举来说更确定，而不是随机选择一个从节点。

如果有机器是首选，Redis主节点、从节点必须被配置一个slave-priority。否则，所有的实例都有一个默认的ID。

一个Redis实例可以被配置指定slave-priority为0为了永远不被Sentinels 选择为新的主节点。然而一个这样配置的从节点会被Sentinels重新配置，为了在一次故障转移后复制新的主节点，唯一不同的是，它永远不会成为主节点。

**算法和内部结构**

下面的章节，我们将会探索Sentinel特性的细节。对于使用者来说，并不需要知道全部的细节，但是一个更深入的理解可能会帮助部署和操作Sentinel以一个更加有效的方式。

**Quorum**

前面的章节展示了每个被Sentinel监控的主节点和一个配置的quorum相关联。它指定了需要同意主节点是不可达或者错误的Sentinel进程的数量为了触发一次故障转移。

可是，在故障转移触发后，为了真正地执行故障转移，至少大多数的Sentinels必须授权一个Sentinel开始故障转移。当只有小部分的Sentinels存在的一个网络分区中，故障转移永远不会执行。

我们尝试让这件事更加清晰：

* Quorum：为了把一个主节点标记成**ODOWN，**需要的Sentinel进程数量来发现错误条件。
* **ODOWN**状态触发故障转移。
* 一旦故障转移被触发，Sentinel尝试向大多数的Sentinels请求授权。

不同之处看起来很微妙，但是实际上很简单地理解和使用。如果你又5个Sentinel实例，quorum被设置为2，一旦2个Sentinel认为主节点不可达，故障转移就会被触发。然而2个Sentinels中的一个得到3个Sentinels的授权才开始故障转移。

把quorum设置为5，必须所有的Sentinels同意主节点失败，并为了开始故障转移，需要得到所有Sentinels的授权。

这意味着quorum在两方面可以被用来调整Sentinel：

1. 如果quorum被设置为小于我们部署的Sentinels大多数，我们使Sentinel对主节点失败更加敏感，并一旦少数的Sentinels不再和主节点交流就会触发故障转移。
2. 如果quorum被设置为大于我们部署的Sentinels大多数，仅仅当大多数连接良好的Sentinels同意主节点挂掉的时候，Sentinel才能开始故障转移。

**配置epochs**

为了开始故障转移，Sentinels需要从大多数得到授权，有下面几个重要的原因：

当一个Sentinel被授权，它为故障转移的主节点获得一个独一无二的配置epoch。这将用来标识新的配置的版本在故障转移完成之后。因为大多数同意一个版本被分配给指定的Sentinel，没有其他的Sentinel可以使用它。这意味着，每次故障转移的配置都有一个独一无二的版本号。我们将看到这为什么是很重要的。

此外Sentinels有一个规则：如果一个Sentinel投票给其他的Sentinel在一次故障转移中，它将等待一段时间再次尝试故障转移这个主节点，你可以在sentinel.conf中配置这个延迟时间failover-timeout。这意味着Sentinels在相同的时间内不会尝试故障转移相同的主节点，第一次请求授权的将会尝试，如果失败了，另一个将会在一段时间后尝试，等等。

Redis Sentinel保证了*活性（liveness）*性质，如果大多数Sentinels 能够交流。如果主节点挂了，最后将有一个会被授权开始故障转移。

Redis Sentinel同样也保证了*安全（safety ）*性质，每个Sentinel将使用不同的*配置epoch*（*configuration epoch*）来故障转移同一个主节点。

**配置传播**

一旦一个Sentinel能成功的故障转移一个主节点，它将开始广播新的配置以便其他的Sentinels更新他们关于主节点的信息。

为了认定一次故障转移是成功的，它需要Sentinel能发送SLAVEOF NO ONE指令给选举的从节点，并且切换为主节点，稍后能在主节点的INFO输出中观察到。

这时候，即使从节点的重新配置正在进行，故障转移也被认为是成功的，并且所有的Sentinels需要开始报告新的配置。

一个新的配置广播的方式，就是为什么我们需要每次Sentinel被授权故障转移时有一个不同的版本号的原因。

每个Sentinel使用Redis发布/订阅消息来连续不断的广播它的一个主节点的配置的版本号，所有的从节点和主节点。同时，所有的Sentinels等待消息来查看其他的Sentinels广播的配置。

配置在\_\_sentinel\_\_:hello发布/订阅频道中被广播。

因为每个配置都有一个不同的版本号，大的版本号总是赢得小的版本号。

例如 一开始所有的Sentinels认为主节点mymaster的配置为192.168.1.50:6379。这个配置的版本号为1。一段时间后，一个被授权开始故障转移有版本号2，如果故障转移成功，它将广播新的配置，它说是192.168.1.50:9000，版本号为2,。所有其他的实例将看到这个配置并更新他们的配置，因为新的配置有更高的版本号。

这意味着Sentinel保证第二个活性属性：一个Sentinels集合能互相交流并且把配置信息收敛到一个更高的版本号。

基本上，如果网络是分区的，每个分区将收敛到一个更高的本地配置。没有网络分区的特殊性情况下，只有一个分区并且每个Sentinel将同意配置。

**分区下的一致性**

Redis Sentinel 配置是最终一致的，所以每个分区将收敛到更高的可用的配置。然而在使用Sentinel的真实世界的系统中，有三个不同的角色：

1. Redis实例
2. Sentinel实例
3. 客户端

为了定义系统的行为，我们考虑所有的三种。

下面是一个简单的有三个节点的网络，每个都运行一个Redis实例和一个Sentinel实例：

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

| Sentinel 1 |----- Client A

| Redis 1 (M) |

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

|

|

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

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

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

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

在这个系统中，原始状态是Redis3是主节点，Redis1和Redis2是从节点。一个网络分区发生隔离了旧的主节点。Sentinels1 和 2开始一个故障转移过程提升Sentinel 1为新的主节点。

Sentinel的属性保证Sentinel 1和 2 现在有了一个主节点的新的配置。可是Sentinel 3依然有旧的配置因为它在一个不同的分区中存活。

我们知道Sentinel 3将得到他的配置更新当网络分区治愈的时候，但是如果有客户端和旧的主节点在一起，分区时会发生什么呢？

客户端仍然可以向Redis 3写入数据。当网络分区治愈，Redis 3变成Reids 1的一个从节点，在分区期间写入的数据都会丢失。

根据你的配置，你可以想或不想让这种情况发生：

* 如果你使用Redis作为缓存，客户端B仍然可以向旧的主节点写入数据是很方便的，即使数据将会丢失。
* 如果你使用Redis作为存储，这是不好的，你需要配置系统为了部分的阻止这个问题。

因为Redis是异步复制的，这种情况下，没有办法完全阻止数据丢失，但是你可以使用下面的Redis配置选项来限制Redis 3 和 Redis 1之间的分歧：

min-slaves-to-write 1

min-slaves-max-lag 10

当一个Redis有上面的配置，当作为主节点的时候，如果他不能向至少一个从节点写入数据，将会停止接受写入请求。因为复制是异步的，不*能写（not being able to write ）*意味着从节点都是分离的，或者没有发送异步确认超过了指定的max-lag的时间。

使用这个配置，上面的例子中的Redis 3将会在10秒之后变得不可用。当分区治愈，Sentinel 3的配置将会是新的，Client B能获取到一个有效的配置并继续工作。

总之， Redis + Sentinel是一个最终一致性系统（ eventually consistent system），功能是最后一个故障转移获胜（ last failover  wins）。旧节点中的数据会被丢弃，从当前主节点复制数据，所以总有一个丢失确认写的窗口。这是由于Redis的异步复制和系统的“虚拟”合并功能的丢弃性质。注意，Sentinel本身没有限制，如果你把故障转移编排起来，相同的属性仍然适用，仅仅有两种方式来避免丢失写入确认：

1. 使用同步复制
2. 使用一个最终一致的系统，相同物体的不同版本能被合并

Redis现在不能使用上面的任何系统，是目前的发展目标。可是有一个代理实现解决方案2在Redis存储之上，比如说SoundCloud  [Roshi](https://github.com/soundcloud/roshi)，或者Netflix  [Dynomite](https://github.com/Netflix/dynomite)。

**Sentinel持久化状态**

Sentinel状态保存在sentinel配置文件中。例如，每次一个收到一个新的配置，主节点，配置和配置epoch一起被保存在磁盘上。这意味着停止和重启Sentinel进程是很安全的。

**TILT模式**

Redis Sentinel严重依赖电脑时间：例如为了推断一个实例是否可达，它会记住最后一次成功回复PING命令的时间，并和当前时间比较来推断哪个是旧的。

可是，如果电脑时间意外改变了，或者电脑非常繁忙，或进程由于某些原因阻塞。Sentinel或许开始表现意外的行为。

TILT模式是一个特殊的“保护”模式，当发现奇怪的事情可能降低系统的可靠性，一个Sentinel可以进入这个模式。Sentinel定时中断调用每秒10次，所以我们期待定时中断调用之间间隔100毫米左右。

Sentinel所做的就是登记之前的中断调用时间，并和当前的调用时间比较：如果结果是负数或意外的数，将会进入TILT模式。

当处于Sentinel模式Sentinel将会继续监控每件事，但是：

* 停止一切动作
* 开始回复负数给SENTINEL is-master-down-by-addr请求让检测失败不再有信

如果30秒内每件事都表现正常，将退出TILT模式。

注意某些情况下，使用许多内核提供的单调时钟API代替TILT模式。可是它仍然是不清晰的如果这是一个很好的解决方案，因为在进程只是仅仅挂起或调度很长时间没有执行的情况下，当前的系统会避免这个问题。