Redis2-主从+哨兵

configuraiton传播

7. 如何保证Redis的高并发和高可用?

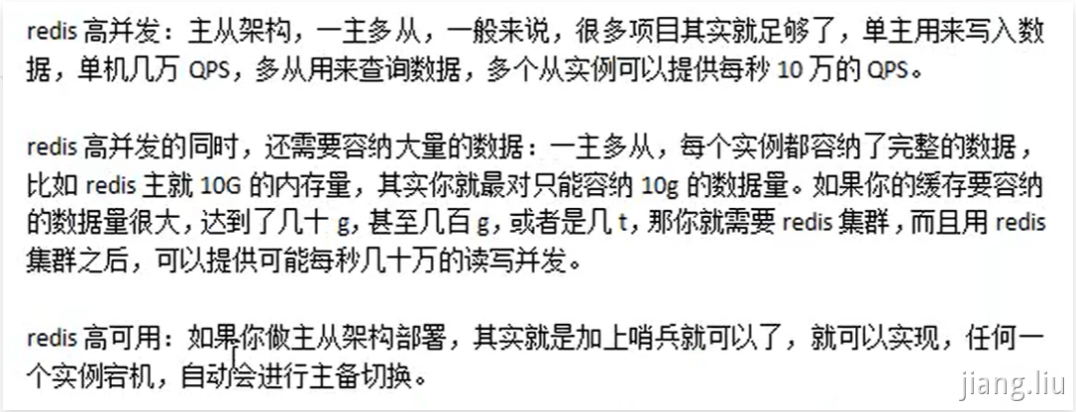
Redis不能支撑高并发的瓶颈在哪里？ 单机！单机的redis几乎不太可能说QPS超过10万+，单机的QPS在几万左右。除非一些特殊情况，比如机器配置特别高，物理机维护做的特别好，而且整体操作不是太复杂。而且对缓存，一般都是用来支撑读高并发的，写请求比较少，可能写请求也就一秒钟几千，一两千。大量的请求都是读，一秒钟二十万次读。

主从架构 -> 读写分离 -> 支撑10万+读QPS的架构。

一主多从，主节点负责写，并且将数据同步到其他slave节点，从节点负责读，所有的读请求全部走从节点。当读并发还增加，就水平扩容，增加slave节点就可以了。

redis replication 主从架构模式

redis主从架构 -> 读写分离架构 -> 可支持水平扩展的读高并发架构。



主从架构的核心原理

当启动一个slave从节点的时候，它会发送一个PSYNC命令给master主节点。

如果这是slave重新连接master，那么master 仅仅会复制给slave 部分缺少的数据，否则如果是slave 第一次连接master，那么会触发一次全量复制，将全量数据同步给slave节点。

开始全量复制的时候，master会启动一个后台线程，开始生成一份RDB快照文件，同时将从客户端收到的所有写命令缓存在内存中。RDB文件生成完毕之后，master会将这个RDB发送给slave，slave会先写入本地磁盘，然后再从本地磁盘加载到内存中。然后master会将内存中缓存的写命令发送给slave，slave也会同步这些数据。

slave node如果跟master node有网络故障，断开了连接，会自动重连。master如果发现有多个slave node都来重新连接，仅仅会启动一个rdb save操作，用一份数据服务所有slave node。

主从复制的断点续传

从redis 2.8开始，就支持主从复制的断点续传，如果主从复制过程中，网络连接断掉了，那么可以接着上次复制的地方，继续复制下去，而不是从头开始重新复制。

master node会在内存中创建一个backlog，master和slave都会保存一个replica offset还有一个master id，offset就是保存在backlog中的。如果master和slave网络连接断掉了，slave会让master从上次的replica offset开始继续复制。

但是如果没有找到对应的offset，那么就会执行一次全量复制。

无磁盘化复制

master在内存中直接创建rdb，然后发送给slave，不会在自己本地落地磁盘了。

配置参数：

> repl-diskless-sync no；无磁盘化复制默认是关闭的，使用的时候将no改为yes即可。
> repl-diskless-sync-delay 5，等待一定时长再开始复制，因为要等更多slave重新连接过来。默认是5秒。

过期key处理

slave不会过期key，只会等待master过期key。如果master过期了一个key，或者通过LRU淘汰了一个key，那么会模拟一条del命令发送给slave。

复制的完整流程

1. slave node启动，仅仅保存master node的信息，包括master node的host和ip，但是复制流程没开始。
2. master host和ip是从哪儿来的，redis.conf里面的slaveof配置的。
3. slave node内部有个定时任务，每秒检查是否有新的master node要连接和复制，如果发现，就跟master node建立socket网络连接。
4. slave node发送ping命令给master node。
5. 口令认证，如果master设置了requirepass，那么salve node必须发送masterauth的口令过去进行认证。
6. master node第一次执行全量复制，将所有数据发给slave node
7. master node后续持续将写命令，异步复制给slave node。

数据同步相关的核心机制

指第一次slave连接msater的时候，执行的全量复制，那个过程里面一些细节的机制。

1）master和slave都会维护一个offset

master会在自身不断累加offset，slave也会在自身不断累加offset。slave每秒都会上报自己的offset给master，同时master也会保存每个slave的offset。这个倒不是说一定就用在全量复制，主要是master和slave都要知道各自的数据的offset，才能知道互相之间的数据不一致的情况。

2）backlog

master node有一个backlog，默认是1MB大小。master node给slave node复制数据时，也会将数据在backlog中同步写一份。backlog主要是用来做全量复制中断后的增量复制的。

3）master run id

通过info server命令，可以看到master run id。

```
[root@eshop-cache02 sentinal]# redis-cli -h 192.168.31.19 -p 6379
192.168.31.19:6379> info server
# Server
redis_version:3.2.8
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:91e51c5f07c551aa
redis_mode:standalone
os:Linux 2.6.32-431.el6.i686 i686
arch_bits:32
multiplexing_api:epoll
gcc_version:4.4.7
process_id:31500
run_id:c0b6c4145c79afa6955798e70e9f94e7ad17e2b5
tcp_port:6379
uptime_in_seconds:35962
uptime_in_days:0
hz:10
lru_clock:16618468
executable:/usr/local/bin/redis-server
config_file:/etc/redis/6379.conf
192.168.31.19:6379> \
```


如果根据host+ip定位master node，是不靠谱的，如果master node重启或者数据出现了变化，那么slave node应该根据不同的run id区分，run id不同就做全量复制。如果需要不更改run id重启redis，可以使用redis-cli debug reload命令。

> 因为可能master出问题了，恢复到20个小时前的数据了，slave比master的数据新，这时应该通过run_id判断，然后将最新的master全量数据，复制给slave，确保数据一致性。

4）psync

从节点使用psync从master node进行复制，psync runid offset。

master node会根据自身的情况返回响应信息，可能是FULLRESYNC runid offset触发全量复制，可能是CONTINUE触发增量复制。

全量复制

1）master执行bgsave，在本地生成一份RDB文件（快照）。

2）master node将rdb快照文件发送给salve node，如果复制时间超过repl-timeout所配置的值（默认60秒），从节点将放弃接受RDB文件并清理已经下载的临时文件，导致全量复制失败，可以适当调节大这个参数。对于千兆网卡的机器，一般每秒传输100MB，6G文件，很可能超过60s。

3）master node在生成rdb时，会将所有新的写命令缓存在内存中，在salve node保存了rdb之后，再将新的写命令复制给salve node

4）client-output-buffer-limit slave 256MB 64MB 60，如果在复制期间，内存缓冲区持续消耗超过64MB，或者一次性超过256MB（这里指的是新写入数据），那么停止复制，复制失败。

5）slave node接收到rdb之后，清空自己的旧数据，然后重新加载rdb到自己的内存中。并同时将复制过来的缓存命令也写入内存中。

6）如果slave node开启了AOF，那么会立即执行BGREWRITEAOF，重写AOF

> rdb生成、rdb通过网络拷贝、slave旧数据的清理、slave aof rewrite，很耗费时间，如果复制的数据量在4G~6G之间，那么很可能全量复制时间消耗到1分半到2分钟。

增量复制

1）如果全量复制过程中，master-slave网络连接断掉，那么salve重新连接master时，会触发增量复制

2）master直接从自己的backlog中获取部分丢失的数据，发送给slave node，默认backlog就是1MB

3）msater就是根据slave发送的psync中的offset来从backlog中获取数据的。

heartbeat 心跳

主从节点互相都会发送heartbeat心跳信息。master默认每隔10秒发送一次heartbeat，salve node每隔1秒发送一个heartbeat。

异步复制

master每次接收到写命令之后，先在内部写入数据，然后异步发送给slave node。

故障转移（主备切换）

master node 发生故障时，通过哨兵模式自动检测，并且将某个slave node自动切换为master node的过程，叫做主备切换，这个过程，实现了Redis的主从架构下的高可用性。

8. Redis哨兵架构

哨兵介绍

sentinal，中文名是哨兵。

哨兵是redis集群架构中非常重要的一个组件，主要功能如下：

（1）**集群监控**，负责监控redis master和slave进程是否正常工作。

（2）**消息通知**，如果某个redis实例有故障，那么哨兵负责发送消息作为报警通知给管理员。

（3）**故障转移**，如果master node挂掉了，会自动转移到slave node上。

（4）**配置中心**，如果发生了故障转移，通知client客户端新的master地址。

哨兵本身也是分布式的，作为一个哨兵集群去运行，互相协同工作

（1）故障转移时，判断一个master node是宕机了，需要大部分的哨兵都同意才行，涉及到了分布式选举的问题

（2）即使部分哨兵节点挂掉了，哨兵集群还是能正常工作的，因为如果一个作为高可用机制重要组成部分的故障转移系统本身是单点的，那就很坑爹了。

> 目前采用的是sentinal 2版本，sentinal 2相对于sentinal 1来说，重写了很多代码，主要是让故障转移的机制和算法变得更加健壮和简单。

哨兵的核心知识

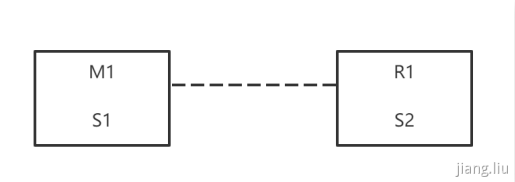
1）哨兵至少需要3个实例，来保证自己的健壮性

2）哨兵 + redis主从的部署架构，是不会保证数据零丢失的，只能保证redis集群的高可用性

3）对于哨兵 + redis主从这种复杂的部署架构，尽量在测试环境和生产环境，都进行充足的测试和演练。

为什么redis哨兵集群只有2个节点无法正常工作？

哨兵集群必须部署**2个以上节点**，如果哨兵集群仅仅部署了个2个哨兵实例，quorum=1



这里有两个参数，quorum是设置的，majority是默认的。

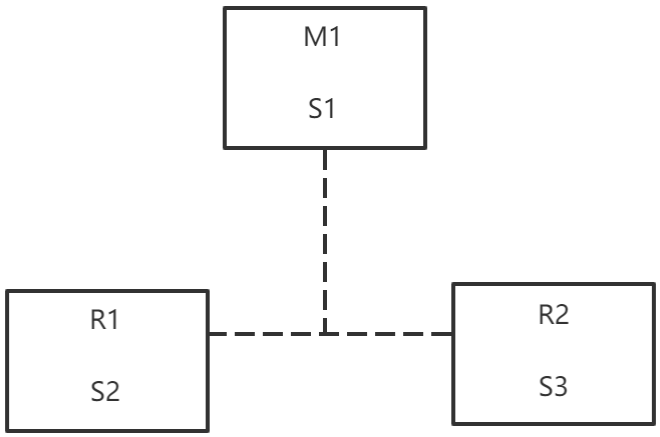
> **quorum**：确认odown（客观宕机）的最少的哨兵数量
>
> **majority**：授权进行主从切换的最少的哨兵数量
>
> 每次一个哨兵要做主备切换，首先需要quorum数量的哨兵认为odown，然后选举出一个哨兵来做切换，这个哨兵还得得到majority哨兵的授权，才能正式执行切换。

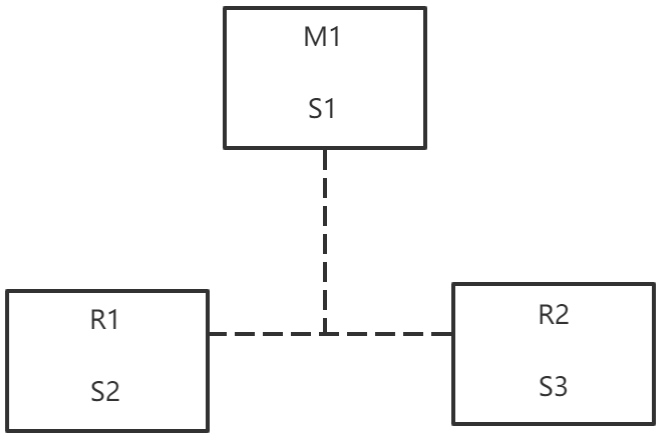
> 如果quorum < majority，比如5个哨兵，majority就是3，quorum设置为2，那么就3个哨兵授权就可以执行切换，但是如果quorum >= majority，那么必须quorum数量的哨兵都授权，比如5个哨兵，quorum是5，那么必须5个哨兵都同意授权，才能执行切换。

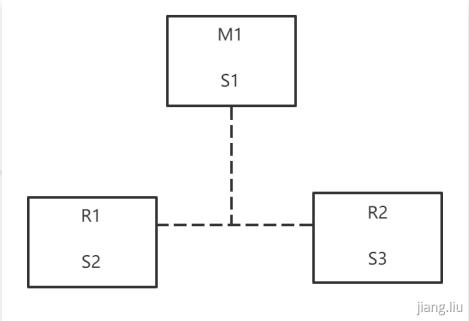
master宕机，s1和s2中只要有1个哨兵认为master宕机，就可以进行切换了，同时s1和s2中会选举出一个哨兵来执行具体的故障转移操作。同时这个时候，需要majority这个参数，也就是大多数哨兵都是运行的，2个哨兵的majority就是2（2的majority=2，3的majority=2，5的majority=3，4的majority=2），2个哨兵都运行着，就可以允许执行故障转移。

但是如果整个M1和S1运行的机器宕机了，那么哨兵只有1个了，此时就没有majority来允许执行故障转移，虽然另外一台机器还有一个R1，但是故障转移不会执行。

经典的3节点哨兵集群







比如设置：quorum=2，

如果M1所在机器宕机了，那么三个哨兵还剩下2个，S2和S3可以一致认为master宕机，然后选举出一个来执行故障转移，同时3个哨兵的majority是2，所以还剩下的2个哨兵运行着，就可以允许执行故障转移。

9.redis哨兵主备切换的数据丢失问题：异步复制、集群脑裂

redis主备切换的过程，可能会导致数据丢失。

两种数据丢失的情况

1）异步复制导致的数据丢失

因为master -> slave的复制是异步的，请求到达master之后，master写成功先返回成功状态给客户端，然后异步复制数据给其他slave，这时候master node挂了的话，其他slave升为master之后，刚才这小部分数据没有同步到其他机器，就发生数据丢失了。

2）脑裂导致的数据丢失

脑裂，也就是说，某个master所在机器突然脱离了正常的网络，跟其他slave机器不能连接，但是实际上master还运行着。此时哨兵可能就会认为master宕机了，然后开启选举，将其他slave切换成了master，这个时候，集群里就会有两个master，也就是所谓的脑裂。

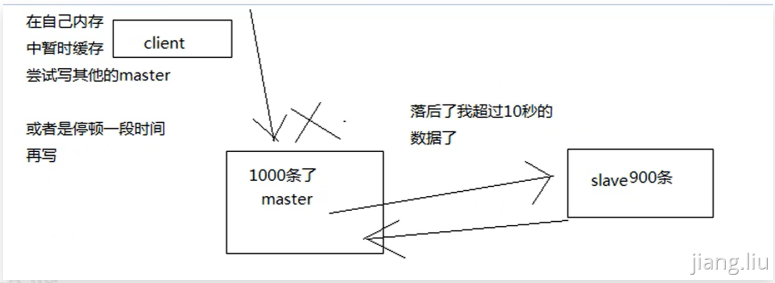
此时虽然某个slave被切换成了master，但是可能client还没来得及切换到新的master，还继续写向旧master写数据，这部分数据就会丢失。当旧master再次恢复的时候，会被作为一个slave挂到新的master上去，自己的数据会清空，重新全量同步新的master的数据。

解决异步复制和脑裂导致的数据丢失

就靠下面这两参数:

```Java
min-slaves-to-write: 1   // 要求至少有1个slave,
min-slaves-max-lag: 10   // 数据复制和同步的延迟不能超过10秒
```

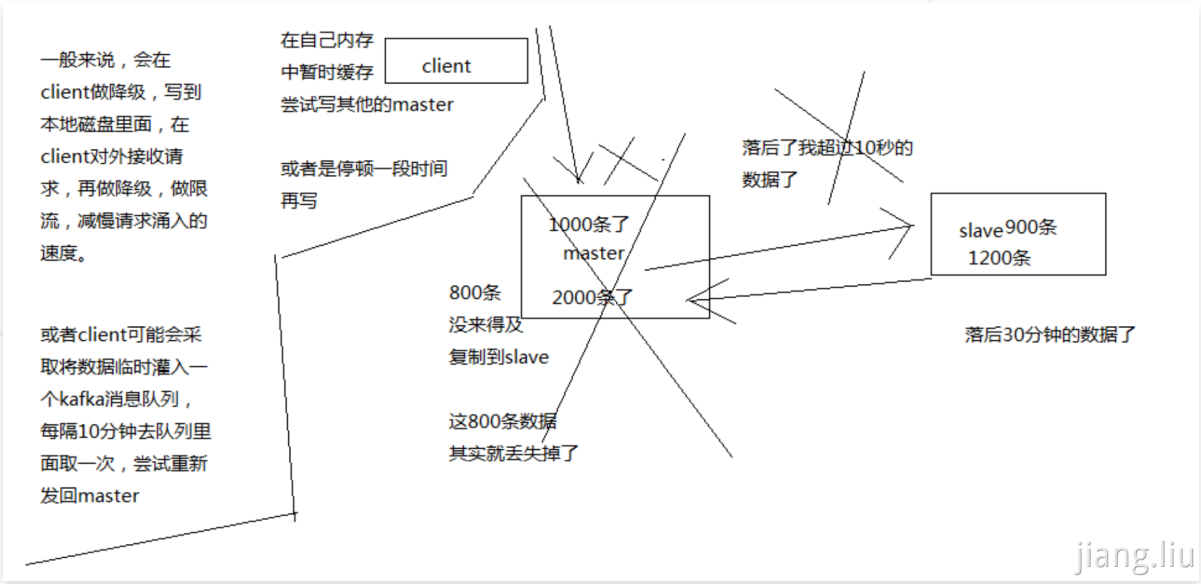
如果说一旦所有的slave，数据复制和同步的延迟都超过了10秒钟，那么这个时候，master就不会再接收任何请求了



上面两个配置可以减少异步复制和脑裂导致的数据丢失：

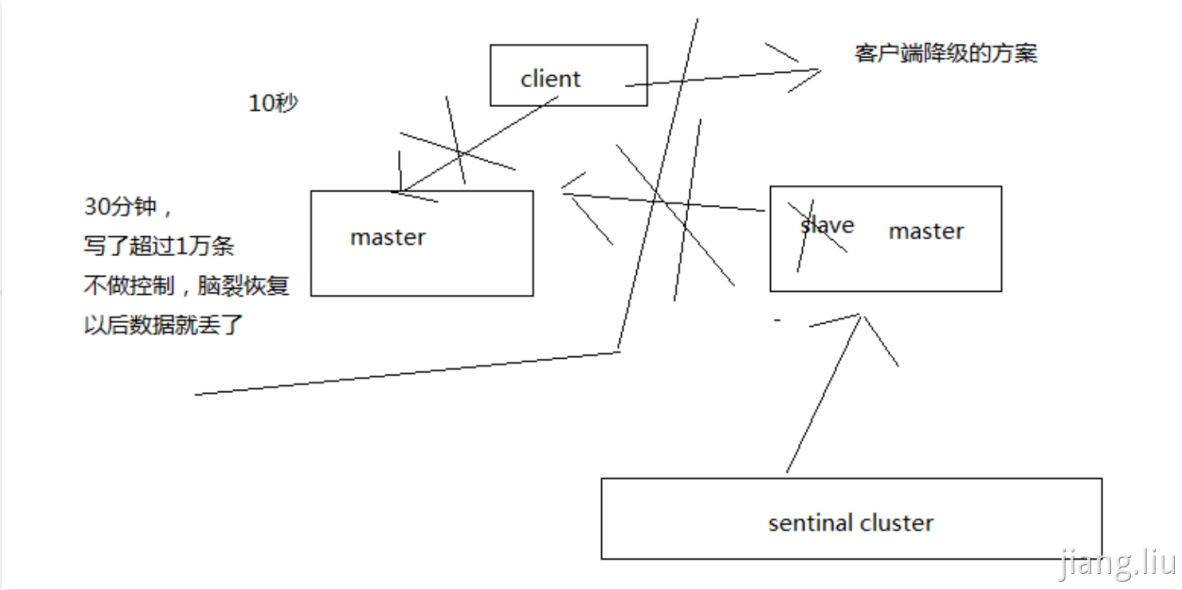
1）减少异步复制的数据丢失

有了`min-slaves-max-lag`这个配置，就可以确保，一旦slave复制数据和ack延时太长，就认为可能master宕机后损失的数据太多了，那么就拒绝写请求，这样可以把master宕机时由于部分数据未同步到slave导致的数据丢失降低的可控范围内。



2）减少脑裂的数据丢失

如果一个master出现了脑裂，跟其他slave丢了连接，那么上面两个配置可以确保，如果不能继续给指定数量的slave发送数据，而且slave超过10秒没有给自己ack消息，那么就直接拒绝客户端的写请求。这样脑裂后的旧master就不会接受client的新数据，也就避免了数据丢失。因此在脑裂场景下，最多就丢失10秒的数据。



10. redis哨兵底层原理分析

sdown和odown转换机制

sdown：主观宕机，就一个哨兵如果自己觉得一个master宕机了，那么就是主观宕机。

odown：客观宕机，如果quorum数量的哨兵都觉得一个master宕机了，那么就是客观宕机。

sdown达成的条件很简单，如果一个哨兵ping一个master，超过了`is-master-down-after-milliseconds`指定的毫秒数之后，就主观认为master宕机。

sdown到odown转换的条件很简单，如果一个哨兵在指定时间内，收到了quorum指定数量的其他哨兵也认为那个master是sdown了，那么就认为是odown了，大家都认为宕机了，所以是客观认为master宕机。

哨兵集群的自动发现机制

哨兵互相之间的发现，是通过redis的pub/sub系统实现的，每个哨兵都会往`__sentinel__:hello`这个channel里发送一个消息，这时候所有其他哨兵都可以消费到这个消息，并感知到其他的哨兵的存在。

每隔两秒钟，每个哨兵都会往自己监控的某个master+slaves对应的`__sentinel__:hello channel`里发送一个消息，内容是自己的host、ip和runid还有对这个master的监控配置。每个哨兵也会去监听自己监控的每个master+slaves对应的`__sentinel__:hello channel`，然后去感知到同样在监听这个master+slaves的其他哨兵的存在。每个哨兵还会跟其他哨兵交换对master的监控配置，互相进行监控配置的同步。

slave配置的自动纠正

哨兵会负责自动纠正slave的一些配置，比如slave如果要成为潜在的master候选人，哨兵会确保slave在复制现有master的数据; 如果slave连接到了一个错误的master上，比如故障转移之后，那么哨兵会确保它们连接到正确的master上。

slave->master选举算法

如果一个master被认为odown了，而且majority哨兵都允许主备切换，那么某个哨兵就会执行主备切换操作，此时首先要选举一个slave做为新的master，这时候会考虑slave的四个因素：

- 跟master断开连接的时长
- slave优先级
- 复制offset
- run id

如果一个slave跟master断开连接已经超过了：down-after-milliseconds的10倍，外加master宕机的时长，，那么slave就被认为不适合选举为master。

公式为：

```
Java | 复制代码
(down-after-milliseconds * 10) + milliseconds_since_master_is_in_SDOWN_state
```

接下来会对slave进行排序：

（1）按照slave优先级进行排序，slave priority（可配参数，默认100）越小，优先级越高。

（2）如果slave priority相同，那么看replica offset，哪个slave复制了越多的数据，offset越靠后，优先级就越高。

（3）如果上面两个条件都相同，那么选择一个run id比较小的那个slave。

quorum和majority

每次一个哨兵要做主备切换，首先需要quorum数量的哨兵认为odown，然后选举出一个哨兵来做切换，这个哨兵还得得到majority哨兵的授权，才能正式执行切换。

如果quorum < majority，比如5个哨兵，majority就是3，quorum设置为2，那么就3个哨兵授权就可以执行切换。

但是如果quorum >= majority，那么必须quorum数量的哨兵都授权，比如5个哨兵，quorum是5，那么必须5个哨兵都同意授权，才能执行切换。

configuration epoch

哨兵会对一套redis master+slave进行监控，有相应的监控的配置。

执行切换的那个哨兵，会从要切换到的新master（salve->master）那里得到一个`configuration epoch`，这就是一个version号，每次切换的version号都必须是唯一的。

如果第一个选举出的哨兵切换失败了，那么其他哨兵，会等待`failover-timeout`时间，然后接替继续执行切换，此时会重新获取一个新的`configuration epoch`，作为新的version号。

configuraiton传播

哨兵完成切换之后，会在自己本地更新生成最新的master配置，然后同步给其他的哨兵，就是通过之前说的pub/sub消息机制。

这里之前的version号就很重要了，因为各种消息都是通过一个channel去发布和监听的，所以一个哨兵完成一次新的切换之后，新的master配置是跟着新的version号的。其他的哨兵都是根据版本号的大小来更新自己的master配置的。