# 8·redis集群

单个Redis服务器会有如下问题：

（1）从结构上，单个 Redis 服务器会发生单点故障，同时一台服务器需要承受所有的请求负载。这就**需要为数据生成多个副本并分配在不同的服务器上**；

（2）从容量上，单个 Redis 服务器的内存非常容易成为存储瓶颈，所以**需要进行数据分片**。

现实中的项目通常需要若干台Redis服务器的支持，同时拥有多个Redis服务器后就会面临如何管理集群的问题，包括如何增加节点、故障恢复等操作。为此，本章将依次详细介绍Redis中的复制、哨兵（sentinel）和集群（cluster）的使用和原理。

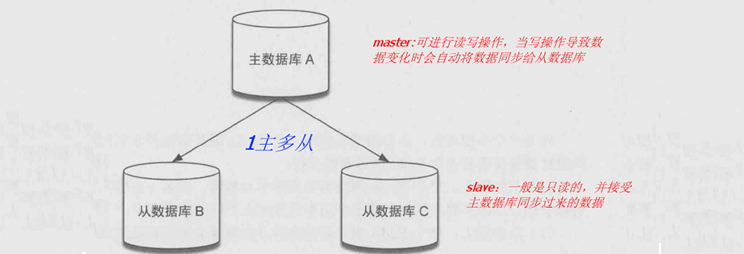
## 8.1·复制（replication）

通过持久化功能，Redis保证了即使在服务器重启的情况下也不会损失（或少量损失）数据。但是由于数据是存储在一台服务器上的，如果这台服务器出现硬盘故障等问题，也会导致数据丢失。**为了避免单点故障，通常的做法是将数据库复制多个副本以部署在不同的服务器上，这样即使有一台服务器出现故障，其他服务器依然可以继续提供服务**。

为此，Redis提供了复制功能，**可以实现当一台数据库中的数据更新后，自动将更新的数据同步到其他数据库上**。

### 8.1.1、配置

在复制的概念中，数据库分为master和slave两类。



在 Redis 中使用复制功能非常容易，配置如下：

|  |  |
| --- | --- |
| master配置 | 无需任何配置 |
| slave配置 | slaveof masterIp masterPort |

实践：实现一个最简化的复制系统，在一台服务器上启动两个 Redis 实例，监听不同端口，其中一个作为主数据库，另一个作为从数据库。

1、首先，不加任何参数启动一个 Redis 实例作为主数据库，该实例默认监听6379端口：

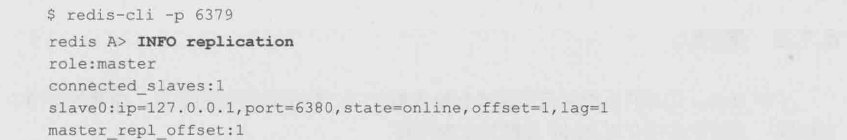


2、然后加上 slaveof 参数启动另一个 Redis 实例作为从数据库，并让其监听6380端口：

此时在主数据库中的任何数据变化都会自动地同步到从数据库中。



3、打开redis-cIi实例 A 并连接到主数据库，并使用INFO命令来获取Replication 节的相关信息：



4、再打开redis-cIi实例 B 并连接到从数据库，同样使用INFO命令来获取Replication 节的相关信息：



5、在实例 A 中使用 SET 命令设置一个键的值：



6、此时在实例 B 中就可以获得该值了：



7、默认情况下，从数据库是只读的，如果直接修改从数据库的数据会出现错误：



可以通过设置从数据库的配置文件中的slave-read-only为no以使从数据库可写，但是因为对从数据库的任何更改都不会同步给任何其他数据库，并且一旦主数据库中更新了对应的数据就会覆盖从数据库中的改动，所以通常的场景卜不应该设置从数据库可写，以免导致易被忽略的潜在应用逻辑错误。

配置多台从数据库的方法也一样，在所有的从数据库的配置文件中都加上 slaveof参数指向同一个主数据库即可。

除了通过配置文件或命令行参数设置 slaveof 参数之外，还可以在运行时使用 **SLAVEOF命令**修改：



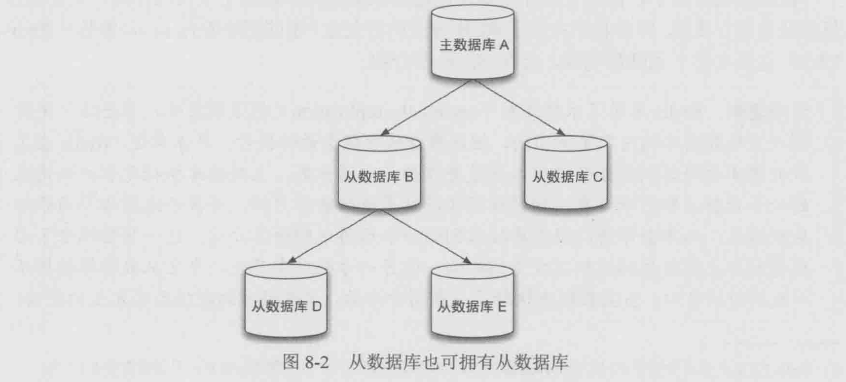
如果该数据库已经是其他主数据库的从数据库了，SLAVEOF命令会停止和原来数据库的同步转而和新数据库同步。此外对于从数据库来说，还可以使用**SLAVEOF NO ONE 命令**来使当前数据库停止接收其他数据库的同步并转换成为主数据库。

### 8.1.2、原理

### 8.1.3、图结构

从数据库不仅可以接收主数据库的同步数据，自己也可以同时作为主数据库存在，形

成类似图的结构。



### 8.1.4、读写分离与一致性

通过复制可以实现读写分离，以提高服务器的负载能力。

在常见的场景中（如电商网站），读的频率远大于写，当单机的 Redis 无法应付大量的读请求时（尤其是较耗资源的请求，如 S0RT 命令等）可以通过复制功能建立多个从数据库节点，主数据库只进行写操作，而从数据库负责读操作。这种一主多从的结构很适合读多写少的场景，而**当单个的主数据库不能够满足需求时，就需要使用 Redis 3.0 推出的集群功能， 8 . 3 节会详细介绍**。

### 8.1.5、从数据库持久化

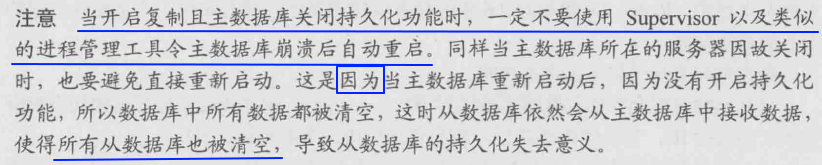
另一个相对耗时的操作是持久化，为了提高性能，可以通过复制功能建立一个（或若干个）从数据库，并**在从数据库中启用持久化，同时在主数据库禁用持久化**。

当从数据库崩溃重启后主数据库会自动将数据同步过来，所以无需担心数据丢失。

然而当主数据库崩溃时，情况就稍显复杂了。手工通过从数据库数据恢复主数据库数据时，需要严格按照以下两步进行。

1·在从数据库中使用**SLAVEOF NO ONE 命令**将从数据库提升成主数据库继续服务。

2·启动之前崩溃的主数据库，然后使用**SLAVEOF命令**将其设置成新的主数据库的从数据库，即可将数据同步回来。



无论哪种情况，手工维护从数据库或主数据库的重启以及数据恢复都相对麻烦，好在**Redis 提供了一种自动化方案哨兵来实现这一过程，避免了手工维护的麻烦和容易出错的问题，** **8.2 节会详细介绍哨兵**。

### 8.1.6、无硬盘复制

redis复制是基于RDB方式的持久化来实现的，这样实现的优点是可以复用已有代码，但也有以下缺点：

1·当主数据库禁用RDB 快照时（即删除了配置文件中所有的 save 语句），如果执行了复制初始化操作，Redis 依然会生成RDB快照，那么主数据库在下次启动后会以该快照来恢复数据，因为复制发生的时间不能确定，这使得恢复的数据可能是任何时间点的。

2·因为复制初始化时需要在硬盘中创建 RDB 快照文件，所以如果硬盘性能很慢（如网络硬盘）时这一过程会对性能产生影响。举例来说，当使用 Redis 做缓存系统时，因为不需要持久化，所以服务器的硬盘读写速度可能较差。但是当该缓存系统使用一主多从的集群架构时，每次和从数据库同步，Redis都会执行一次快照，同时对硬盘进行读写，导致性能降低。

因此从2.8.18 版本开始， Redis引入了无硬盘复制选项，开启该选项时，Redis 在与从数据库进行复制初始化时将不会将快照内容存储到硬盘上，而是直接通过网络发送给从数据库，避免了硬盘的性能瓶颈。可以在配置文件中使用如下配置来开启该功能：



### 8.1.7、增量复制

8.1.2节在介绍复制的原理时提到当主从数据库连接断开后，从数据库会发送 SYNC命令来重新进行一次完整复制操作，即使断开期间数据库的变化很小（甚至没有），也需要将数据库中的所有数据重新快照并传送一次。这种实现方式显然不太理想，Redis 2.8 相对2.6的最重要的更新之一就是实现了**主从断线重连情况下的增量复制**。

增量复制是基于如下 3 点实现的：

1·从数据库会存储主数据库的运行 ID（run id）。每个 Redis 运行实例均会拥有一个唯一的运行 ID ，每当实例重启后，就会自动生成一个新的运行 ID 。

2·在复制同步阶段，主数据库每将一个命令传送给从数据库时，都会同时把该命令存放到一个积压队列（backlog）中，并记录下当前积压队列中存放的命令的偏移量范围。

3·同时，从数据库接收到主数据库传来的命令时，会记录下该命令的偏移浪。

这3点是实现增量复制的基础。回到8.1.2节的主从通信流程，可以看到，当主从连接准备就绪后，从数据库会发送一条SYNC命令来告诉主数据库可以开始把所有数据同步过来了。而2.8版之后，不再发送SYNC命令，取而代之的是发送 **PSYNC命令** ，格式为“PSYNC 主数据库的运行 ID 断开前最新的命令偏移量”。主数据库收到PSYNC命令后，会执行以下判断来决定此次重连是否可以执行增量复制：

1·首先主数据库会判断从数据库传送来的运行 ID 是否和自己的运行 ID 相同。这一步骤的意义在于确保从数据库之前确实是和自己同步的，以免从数据库拿到错误的数据（比如主数据库在断线期间重启过，会造成数据的不一致）。

2·然后判断从数据库最后同步成功的命令偏移量是否在积压队列中，如果在则可以执行增量复制，并将积压队列中相应的命令发送给从数据库。

如果此次重连不满足增量复制的条件，主数据库会进行一次全部同步（即与Redis2.6的过程相同）。

大部分情况下，增量复制的过程对开发者来说是完全透明的，开发者不需要关心增量复制的具体细节。2.8版本的主数据库也可以正常地和旧版本的从数据库同步（通过接收SYNC命令），同样2.8版本的从数据库也可以与旧版本的主数据库同步（通过发送SYNC命令）。**唯一需要开发者设置的就是积压队列的大小了**。

积压队列在本质上是一个固定长度的循环队列，默认情况下积压队列的大小为 1 MB ，可以通过配置文件的repl-backlog-size选项来调整。很容易理解的是，积压队列越大，其允许的主从数据库断线的时间就越长。根据主从数据库之间的网络状态，设置一个合理的积压队列很重要。因为积压队列存储的内容是命令本身，如 set foo bar ，所以估算积压队列的大小只需要估计主从数据库断线的时间中主数据库可能执行的命令的大小即可。

与积压队列相关的另一个配置选项是repl-backlog-ttl，即当所有从数据库与主数据库断开连接后，经过多久时间可以释放积压队列的内存空间。默认时问是1小时。

## 8.2·哨兵

8.1节介绍了Redis中复制的原理和使用方式，在一个典型的一主多从的Redis系统中，从数据库在整个系统中起到了数据冗余备份和读写分离的作用。当主数据库遇到异常中断服务后，开发者可以通过手动的方式选择一个从数据库来升格为主数据库，以使得系统能够继续提供服务。然而整个过程相对麻烦且需要人工介入，难以实现自动化。

为此，Redis2.8中提供了哨兵工具来实现自动化的系统监控和故障恢复功能。

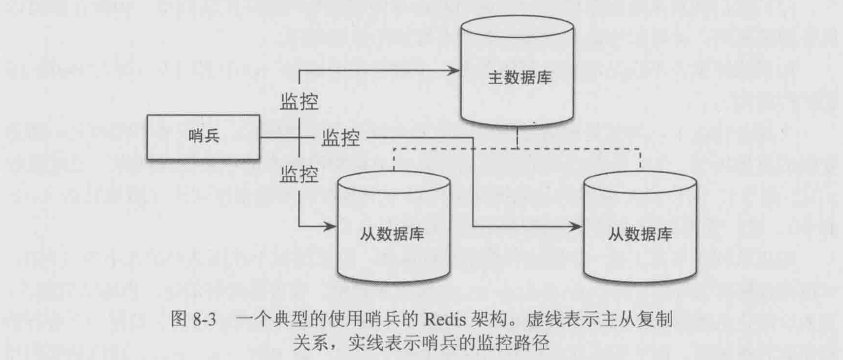
### 8.2.1·什么是哨兵

顾名思义，哨兵的作用就是监控Redis系统的运行状况。它的功能包括以下两个。

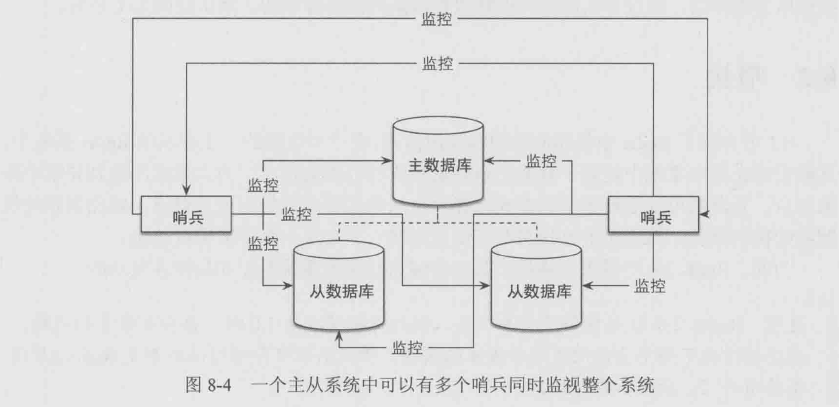
（1）监控主数据库和从数据库是否正常运行。

（2）主数据库出现故障时自动将从数据库转换为主数据库。

哨兵是一个独立的进程，使用哨兵的一个典型架构如下图8-3所示。



在一个一主多从的Redis系统中，可以使用多个哨兵进行监控任务以保证系统足够稳健，如图8-4所示。注意，此时哨兵不仅会同时监控主数据库和从数据库，哨兵之间也会互相监控。

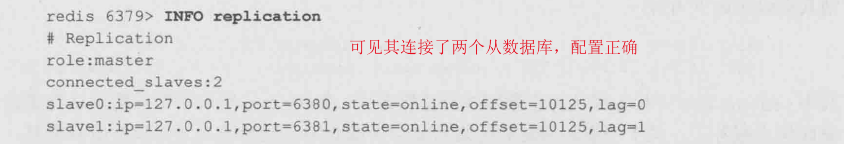


### 8.2.2·马上上手

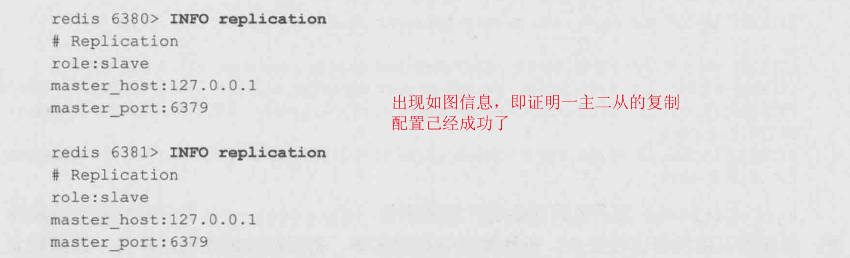
在理解哨兵的原理前，我们首先实际使用一下哨兵，来了解哨兵是如何工作的。

首先按照8.1节介绍的方式建立起3个Redis实例，其中包括一个主数据库和两个从数据库。主数据库的端口为6379，两个从数据库的端口分别为6380和6381。我们使用Redis命令行客户端来获取复制状态，以保证复制配置正确。

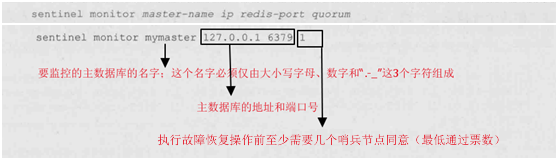
首先是主数据库：



然后用同样的方法查看两个从数据库的配置：



接下来开始配置哨兵。建立一个配置文件，如sentinel.conf，内容为：

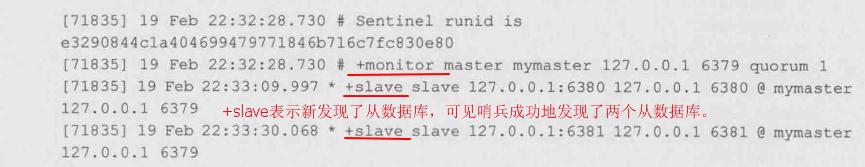


需要注意的是，配置哨兵监控一个系统时，只需要配置其监控的主数据库即可，哨兵会自动发现所有复制该主数据库的从数据库，具体原理后面会详细介绍。

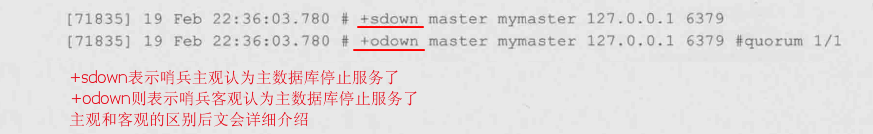
接下来启动Sentinel进程，并将上述配置文件的路径传递给哨兵：



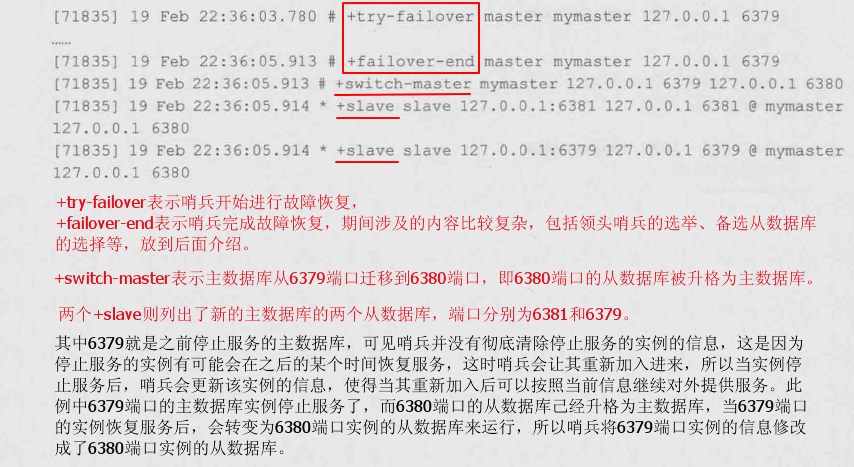
启动哨兵后，哨兵输出如下内容：



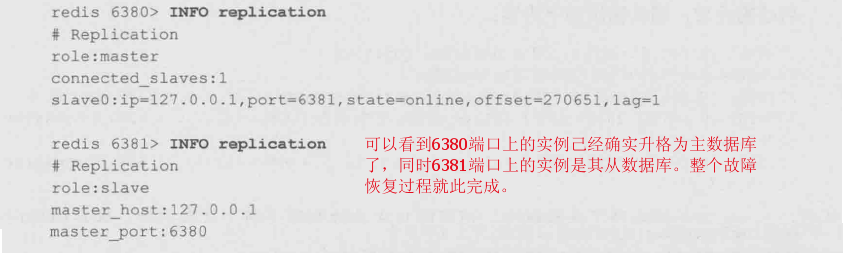
现在哨兵已经在监控这3个Redis实例了，这时我们将主数据库（即运行在6379端口上的Redis实例）关闭（杀死进程或使用shutdown命令），等待指定时问后（可以配置，默认为30秒）,哨兵会输出如下内容：



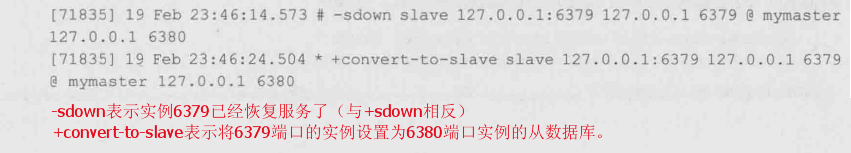
接着哨兵开始执行故障恢复，即挑选一个从数据库，将其升格为主数据库，同时输出如下内容：



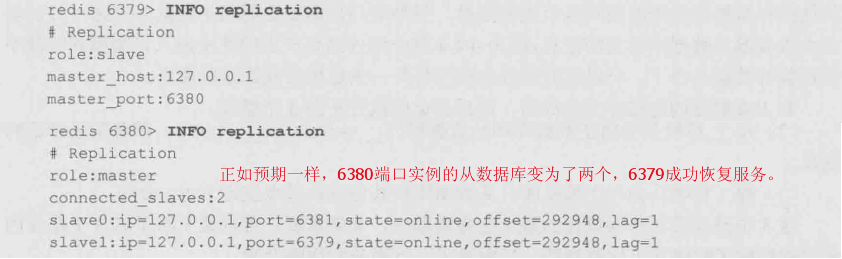
故障恢复完成后，可以使用Redis命令行客户端重新检查6380和6381两个端口上的实例的复制信息：



那么此时我们将6379端口上的实例重新启动，会发生什么情况呢？首先峭兵会监控到这一变化，并输出：



这时使用Redis命令行客户端查看6379端口实例和6380端口实例的复制信息为：



### 8.2.3·实现原理

#### 8.2.3.1·哨兵配置文件

一个哨兵进程启动时会读取配置文件的内容，

通过文件中如下配置可以找出需要监控的主数据库信息，如名称、地址、端口号。（注：因为考虑到故障恢复后当前监控的系统的主数据库的地址和端口会产生变化，所以哨兵提供了命令可以通过master-name获取当前系统的主数据库的地址和端口号。）



一个哨兵节点可以同时监控多个Redis主从系统，只需要提供多个sentinel monitor配置即可，例如：



配置文件中还可以定义其他监控相关的参数，每个配置选项都包含主数据库的名字使得监控不同主数据库时可以使用不同的配置参数。例如：



#### 8.2.3.2·哨兵启动后会做的工作

**哨兵启动后，会与要监控的主数据库建立两条连接**，这两个连接的建立方式与普通的Redis客户端无异。其中一条连接用来订阅该主数据库的\_\_sentinel\_\_:hello频道以获取其他同样监控该数据库的哨兵节点的信息；另外哨兵也需要定期向主数据库发送INFO等命令来获取主数据库本身的信息，这需要使用另外一条连接来发送这些命令（因为4.4.4节介绍过，当客户端的连接进入订阅模式时就不能再执行其他命令了）。

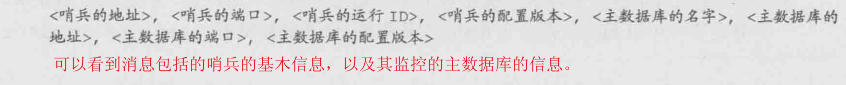
**和主数据库的连接建立完成后，哨兵会定时执行下面3个操作**。

|  |
| --- |
| （1）每10秒哨兵会向主数据库和从数据库发送INFO命令。  （2）每2秒哨兵会向主数据库和从数据库的\_\_sentinel\_\_:hello频道发送自己的信息。  （3）每1秒哨兵会向主数据库、从数据库和其他哨兵节点发送PING命令。 |

这3个操作贯穿哨兵进程的整个生命周期中，非常重要，可以说了解了这3个操作的意义就能够了解哨兵工作原理的一半内容了。下面分别详细介绍。

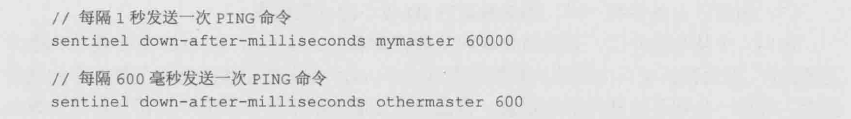
（1）首先，发送INFO命令使得哨兵可以获得当前数据库的相关信息（包括运行ID、复制信息等）从而实现新节点的自动发现。前面说配置哨兵监控Redis主从系统时只需要指定主数据库的信息即可，因为哨兵正是借助INFO命令来获取所有复制该主数据库的从数据库信息的。启动后，哨兵向主数据库发送INFO命令，通过解析返回结果来得知从数据库列表，而后对每个从数据库同样建立两个连接，两个连接的作用和前文介绍的与主数据库建立的两个连接完全一致。在此之后，哨兵会每10秒定时向已知的所有主从数据库发送INFO命令来获取信息更新并进行相应操作，比如对新增的从数据库建立连接并加入监控列表，对主从数据库的角色变化（由故障恢复操作引起）进行信息更新等。

（2）接下来哨兵向主从数据库的\_\_sentinel\_\_:hello频道发送信息来与同样监控该数据库的哨兵分享自己的信息。发送的消息内容为：



前文介绍过，哨兵会订阅每个其监控的数据库的\_\_sentinel\_\_:hello频道，所以当其他哨兵收到消息后，会判断发消息的哨兵是不是新发现的哨兵。如果是则将其加入己发现的哨兵列表中并创建一个到其的连接（与数据库不同，哨兵与哨兵之间只会创建一条连接用来发送PING命令，而不需要创建另外一条连接来订阅频道，因为哨兵只需要订阅数据库的频道即可实现自动发现其他哨兵）。同时哨兵会判断信息中主数据库的配置版本，如果该版本比当前记录的主数据库的版本高，则更新主数据库的数据。配置版本的作用会在后面详细介绍。

（3）实现了自动发现从数据库和其他哨兵节点后，哨兵要做的就是定时监控这些数据库和节点有没有停止服务。这是通过每隔一定时间向这些节点发送PING命令实现的。时间间隔与down-after-milliseconds选项有关，当down-after-milliseconds的值小于l秒时，哨兵会每隔down-after-milliseconds指定的时间发送一次PING命令，当down-after-milliseconds的值大于l秒时，哨兵会每隔1秒发送一次PING命令。例如：



当超过down-after-milliseconds选项指定时间后，如果被PING的数据库或节点仍然未进行回复，则哨兵认为其**主观下线（subjectively down）**。主观下线表示从当前的哨兵进程看来，该节点己经下线。如果该节点是主数据库，则哨兵会进一步判断是否需要对其进行故障恢复：哨兵发送SENTINEL is-master-down- by-addr命令询问其他哨兵节点以了解他们是否也认为该主数据库主观下线，如果达到指定数量时，哨兵会认为其**客观下线（objectively down）**，并选举领头的哨兵节点对主从系统发起故障恢复。这个指定数量即为前文介绍的quorum参数。例如，下面的配置：



该配置表示只有当至少两个Sentinel节点（包括当前节点）认为该主数据库主观下线时，当前哨兵节点才会认为该主数据库客观下线。进行接下来的选举领头哨兵步骤。

#### 8.2.3.3·选举领头哨兵的过程

虽然当前哨兵节点发现了主数据库客观下线，需要故障恢复，但是故障恢复需要由领头的哨兵来完成，这样可以保证同一时间只有一个哨兵节点来执行故障恢复。**选举领头哨兵的过程使用了Raft算法**，具体过程如下。

|  |
| --- |
| （1）发现主数据库客观下线的哨兵节点（下面称作A）向每个哨兵节点发送命令，要求对方选自己成为领头哨兵。  （2）如果目标哨兵节点没有选过其他人，则会同意将A设置成领头哨兵。  （3）如果A发现有超过半数且超过quorum参数值的哨兵节点同意选自己成为领头哨兵，则A成功成为领头哨兵。  （4）当有多个哨兵节点同时参选领头哨兵，则会出现没有任何节点当选的可能。此时每个参选节点将等待一个随机时间重新发起参选请求，进行下一轮选举，直到选举成功。 |

因为要成为领头哨兵必须有超过半数的哨兵节点支持，所以每次选举最多只会选出一个领头哨兵。

#### 8.2.3.4·故障恢复的过程

选出领头哨兵后，领头哨兵将会开始对主数据库进行故障恢复。**故障恢复的过程**相对简单，具体如下。

首先，领头哨兵会从停止服务的主数据库的从数据库中挑选一个出来（后续将被升格为新的主数据库），挑选的依据如下。

|  |
| --- |
| （1）所有在线的从数据库中，选择优先级最高的从数据库。优先级可以通过slave-priority选项来设置。  （2）如果有多个最高优先级的从数据库，则复制的命令偏移量（见8.1.7节）越大（即复制越完整）越优先。  （3）如果以上条件都一样，则选择运行ID较小的从数据库。 |

而后，领头哨兵将向挑选出来的那个从数据库发送SLAVEOF NO ONE命令使其升格为主数据库。

再而后，领头哨兵向其他从数据库发送SLAVEOF命令来使其成为新主数据库的从数据库。

最后，更新内部的记录，将己经停止服务的旧的主数据库更新为新的主数据库的从数据库，使得当其恢复服务时自动以从数据库的身份继续服务。

### 8.2.4·哨兵的部署

同时多个哨兵节点也可以同时监控同一个Redis主从系统，从而形成网状结构。具体实践时如何协调哨兵与主从系统的数量关系会在8.2.4节介绍。

哨兵以独立进程的方式对一个主从系统进行监控，监控的效果好坏与否取决于哨兵的视角是否有代表性。如果一个主从系统中配置的哨兵较少，哨兵对整个系统的判断的可靠性就会降低。极端情况下，当只有一个哨兵时，哨兵本身就可能会发生单点故障。整体来讲，相对稳妥的哨兵部署方案是使得哨兵的视角尽可能地与每个节点的视角一致，即：

（1）为每个节点（无论是主数据库还是从数据库）部署一个哨兵；

（2）使每个哨兵与其对应的节点的网络环境相同或相近。

这样的部署方案可以保证哨兵的视角拥有较高的代表性和可靠性。举例一个例子：当网络分区后，如果哨兵认为某个分区是主要分区，即意味着从每个节点观察，该分区均为主分区。

同时设置quorum的值为N/2+1（其中N为哨兵节点数量），这样使得只有当大部分哨兵节点同意后才会进行故障恢复。

当系统中的节点较多时，考虑到每个哨兵都会和系统中的所有节点建立连接，为每个节点分配一个哨兵会产生较多连接，尤其是当进行客户端分片时使用多个哨兵节点监控多个主数据库会因为Redis不支持连接复用而产生大量冗余连接，具体可以见此issue（<https://github.com/antirez/redis/issues/2257>）；同时如果Redis节点负载较高，会在一定程度上影响其对哨兵的回复以及与其同机的哨兵与其他节点的通信。所以配置哨兵时还需要根据实际的生产环境情况进行选择。