现实项目中通常需要若干台Redis服务器的支持：

结构上，单个 Redis 服务器会发生单点故障，而且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；

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。

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。

**一：复制(replication)**

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，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。

1：配置

在复制的概念中，数据库分为两类，一类是主库，另一类是从库。主库可以进行读写操作，当写操作导致数据变化时，会自动将数据同步给从库。从库一般是只读的，并接受主库同步过来的数据。

一个主库可以拥有多个从库，而一个从库只能拥有一个主库，如下图所示：



在 Redis 中使用复制功能非常容易， 只需要在从库的配置文件中加入：

”slaveof *masterip masterport*”即可，而主库无需进行任何配置。

下面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，其中一个作为主库，另一个为从库。

首先不加任何参数来启动一个Redis实例作为主库，该实例默认监听6379端口。然后启动另一个Redis实例，加上”slaveof”参数作为从库，并让其监听6380端口：

# redis-server --port 6380 --slaveof 127.0.0.1 6379

此时在主库中的任何数据变化都会自动地同步到从库中。打开终端A执行redis-cli连接到主库：

root@localhost**:~**# redis**-**cli

127.0.0.1**:**6379**>**

再打开终端B执行redis-cli连接到从库：

root@localhost**:~**# redis**-**cli **-**p 6380

127.0.0.1**:**6380**>**

使用”info”命令来分别在终端A和终端B中获取Replication节点的相关信息，下面是终端A得到的信息：

127.0.0.1**:**6379**>**info replication

# Replication

role**:**master

connected\_slaves**:**1

slave0**:**ip**=**127.0.0.1**,**port**=**6380**,**state**=**online**,**offset**=**379**,**lag**=**1

**...**

可见，得到的角色是master，即主库，同时已连接的从库的个数为1。下面是终端B得到的信息：

127.0.0.1**:**6380**>**info replication

# Replication

role**:**slave

master\_host**:**127.0.0.1

master\_port**:**6379

**...**

可见，得到的角色是slave，即从库，同时其主库的IP地址为127.0.0.1，端口为 6379。

在终端A中使用set命令设置一个键的值：

127.0.0.1**:**6379**>** set foo heheh

OK

此时在终端B中就可以获得该值了：

127.0.0.1**:**6380**>** get foo

"heheh"

默认情况下，从库是只读的， 如果直接修改从库的数据会出现错误：

127.0.0.1**:**6380**>** set foo hi

**(**error**)** READONLY You can't write against a read only slave.

可以通过设置从库的配置文件中的“slave-read-only”为”no”，使从库可写，但是对从库的任何更改都不会同步给任何其他数据库，并且一旦主库中更新了对应的数据就会覆盖从库中的改动，所以通常的场景下，不应该设置从库可写。

配置多台从库的方法也一样，在所有从库的配置文件中都加上”slaveof”参数指向同一个主库即可。

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

127.0.0.1**:**6380**>**info replication

# Replication

role**:**master

connected\_slaves**:**0

**...**

127.0.0.1**:**6380**>** slaveof 127.0.0.1 6379

OK

127.0.0.1**:**6380**>**info replication

# Replication

role**:**slave

master\_host**:**127.0.0.1

master\_port**:**6379

**...**

如果该数据库已经是其他主库的从库了，则slaveof命令会停止和原来数据库的同步，转而和新数据库进行同步。此外对于从库来说，还可以使用”slave 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库。

2：原理

当一个从库启动后，会向主库发送”sync”命令。主库收到该命令后，开始在后台保存快照（即RDB持久化的过程），并将保存快照期间接收到的命令缓存起来。当快照完成后，Redis会将快照文件和所有缓存的命令发送给从库。从库收到后，会载入快照文件并执行收到的缓存命令。以上过程称为复制初始化。

复制初始化结束后，主库每当收到写命令时就会将命令同步给从库，从而保证主从库数据一致。

当主从库之间的连接断开重连后，Redis 2.6及之前的版本会重新进行复制初始化（即主库重新保存快照并传送给从库），即使从库可能仅有几条命令没有收到，主库也必须要将数据库里的所有数据重新传送给从库。这使得主从库断线重连后的数据恢复过程效率很低下，在网络环境不好的时候这一问题尤其明显。

Redis 2.8版的一个重要改进就是断线重连后，支持有条件的增量数据传输，当从库重新连接上主库后，主库只需要将断线期间执行的命令传送给从库，从而大大提高Redis复制的实用性。后续会详细介绍增量复制的实现原理以及应用条件。

下面从具体协议角度，详细介绍复制初始化的过程。Redis服务器使用 TCP协议通信，可以使用 telnet 工具伪装成一个从库来与主库通信。首先在命令行中连接主库：

root@localhost**:~**# telnet 127.0.0.1 6379

Trying 127.0.0.1...

Connected to 127.0.0.1.

Escape character is '^]'**.**

作为从库，先要发送”ping”命令确认主库是否可以连接：

ping

**+**PONG

而后向主库发送”replconf”命令说明自己的端口号：

replconf listening**-**port 6381

**+**OK

这时就可以开始同步的过程了，向主库发送”sync”命令开始同步，此时主库发送回快照文件和缓存的命令。目前主库中只有一个foo键，所以收到的内容如下（快照文件是二进制 格式，从第三行开始）：

sync

$29

REDIS0006**?**foobar**?**6\_**?**"

从库将收到的内容写入到硬盘上的临时文件中，当写入完成后从库会用该临时文件替换RDB快照文件，之后的操作就和RDB持久化时，启动恢复的过程一样了。在同步的过程中从库并不会阻塞，而是可以继续处理客户端发来的命令。

默认情况下，从库会用同步前的数据对命令进行响应。可以配置”slaveserve-stale-data”参数为”no”来使从库在同步完成前对所有命令（除了info和 slaveof）都回复错误：

”SYNC with master in progress.“

复制初始化阶段结束后，主库执行的任何导致数据变化的命令都会异步地传送给从库，这一过程为“复制同步阶段”。同步的内容和Redis通信协议一样。复制同步阶段会贯穿整个主从同步过程的始终，直到主从关系终止为止。

在复制的过程中，快照无论在主库还是从库中都起了很大的作用，只要执行复制就会进行快照，即使关闭了RDB方式的持久化（通过删除所有save参数）。Redis 2.8.18 之后支持了无硬盘复制，会在下面介绍。

Redis采用了乐观复制(optimistic replication)的复制策略，容忍在一定时间内主从库的内容是不同的，但是两者的数据会最终同步。具体来说，Redis在主从库之间复制数据的过程本身是异步的，这意味着，主库执行完客户端请求的命令后会立即将命令在主库的执行结果返回给客户端，并异步地将命令同步给从库，而不会等待从库接收到该命令后再返回给客户端。

这一特性保证了启用复制后主库的性能不会受到影响，但另一方面也会产生一个主从库数据不一致的时间窗口，当主库执行了一条写命令后，主库的数据已经发生的变动，然而在主库将该命令传送给从库之前，如果两个数据库之间的网络连接断开了，此时二者之间的数据就会是不一致的。

从这个角度来看，主库是无法得知某个命令最终同步给了多少个从库的，不过 Redis 提供了两个配置选项，来限制只有当数据至少同步给指定数量的从库时，主库才是可写的：

min-slaves-to-write 3

min-slaves-max-lag 10

“min-slaves-to-write”表示只有当3个（或以上）的从库连接到主库时，主库才是可写的，否则会返回错误：

“NOREPLICAS Not enough good slaves to write.”

“min-slaves-max-lag”表示允许从库最长失去连接的时间，如果从库最后与主库联系（即发送 “replconf ack”命令）的时间小于这个值，则认为从库还在保持与主库的连接。

举个例子，按上面的配置，假设主库与3个从库相连，其中一个从库上一次与主库联系是 9 秒前，这时主库可以正常接受写入，一旦1秒过后这台从库依旧没有活动，则主库则认为目前连接的从库只有2个，从而拒绝写入。这一特性默认是关闭的，在分布式系统中，打开并合理配置该选项后可以降低主从架构中因为网络分区导致的数据不一致的问题。

从库不仅可以接收主库的同步数据，自己也可以同时作为主库存在，形成类似下图的结构：



数据库A的数据会同步到B和C中， 而B中的数据会同步到D和E中。向B中写入数据不会同步到A或C中，只会同步到 D和E中。

3：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（如电子商务网站），读的频率大于写，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 sort命令等），可以通过复制功能建立多个从库节点，主库只进行写操作，而从库负责读操作。

这种一主多从的结构很适合读多写少的场景，而当单个主库不能够满足需求时，就需要使用Redis 3.0 推出的集群功能，后续会详细介绍。

4：从库持久化

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）从库，并在从库中启用持久化，同时在主库禁用持久化。当从库崩溃重启后，主库会自动将数据同步过来，所以无需担心数据丢失。 然而当主库崩溃时，情况就稍显复杂了。

手工通过从库数据恢复主库数据时，需要严格按照以下两步进行：

a：在从库中使用 “slaveof no one”命令将从库提升成主库继续服务；

b：启动之前崩溃的主库，然后使用”slaveof”命令将其设置成新主库的从库，即可将数据同步回来。

注意，当开启复制且主库关闭持久化功能时，一定不要使用 Supervisor 以及类似的进程管理工具令主库崩溃后自动重启。同样当主库所在的服务器因故关闭时，也要避免直接重新启动。这是因为当主库重新启动后，因为没有开启持久化功能，所以数据库中所有数据都被清空，这时从库依然会从主库中接收数据，使得所有从库也被清空，导致从库的持久化失去意义。

无论哪种情况，手工维护从库或主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，后续会详细介绍“哨兵”。

5：无硬盘复制

介绍Redis复制的工作原理时，介绍了复制是基于RDB方式的持久化实现的，即主库端在后台保存 RDB 快照，从库端则接收并载入快照文件。这样的实现优点是可以显著地简化 逻辑，复用已有的代码，但是缺点也很明显：

a：当主库禁用RDB快照时（即删除了所有的配置文件中的save语句），如果执行了复制初始化操作，Redis依然会生成RDB快照，所以下次启动后主库会以该快照恢复数据。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。

b：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，则这一过程会对性能产生影响。

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库，避免了硬盘的性能瓶颈。

目前无硬盘复制的功能还在试验阶段，可以在配置文件中使用如下配置来开启该功能：

repl-diskless-sync yes

6：增量复制

在介绍复制原理时提到，当主从库连接断开后，从库会发送sync命令来重新进行一次完整复制操作。这样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版相对2.6版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功能。增量复制是基于如下3点实现的：

a：从库会保存主库的运行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。

b：在复制同步阶段，主库每将一个命令传送给从库时，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。

c：同时，从库接收到主库传来的命令时，会记录下该命令的偏移量。

注意，主库和所有从库都记录了命令的偏移量。以上3点是实现增量复制的基础。当主从连接准备就绪后，从库会发送一条sync命令来告诉主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，格式为“psync *主库的运行ID 断开前最新的命令偏移量*”。

主库收到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：

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

b：然后判断从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库。如果此次重连不满足增量复制的条件，主库会进行一次全部同步（即与Redis 2.6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 版本的主库也可以正常地和旧版本的从库同步（通过接收sync命令），同样 2.8 版本的从库也可以与旧版本的主库同步（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。积压队列越大，其允许的主从库断线的时间就越长。

根据主从库之间的网络状态，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库断线的时间中，主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”，即当所有从库与主库断开连接后，经过多久时间可以释放积压队列的内存空间。默认时间是1小时。

**二：哨兵**

在一个典型的一主多从的Redis 系统中，从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库来升级为主库，以使得系统能继续提供服务。

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，Redis 2.8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。

注意 Redis 2.6 版也提供了哨兵工具，但此时的哨兵是1.0版，存在非常多的问题，任何情况下都不应该使用这个版本的哨兵。本章介绍的哨兵都是Redis 2.8提供的哨兵

1：什么是哨兵

哨兵的作用就是监控 Redis系统的运行状况。它的功能包括：

a：监控主库和从库是否正常运行；

b：主库出现故障时自动将从库转换为主库；

哨兵是一个独立的进程，使用哨兵的一个典型架构如下图所示，其中虚线表示主从复制关系，实线表示哨兵的监控路径：



在一主多从的Redis系统中，可以使用多个哨兵进行监控，以保证系统足够稳健，此时，不仅哨兵会监控主库和从库，哨兵之间也会互相监控。如下图所示：



2：实践

在理解哨兵的原理前，首先实际使用一下哨兵，来了解哨兵是如何工作的。为了简单起见，在同一个主机上建立3个Redis实例，包括一个主库和两个从库。主库的端口为6379，两个从库的端口分别为6380和6381。使用Redis-cli获取复制状态，以保证复制配置正确。 首先是主库：

127.0.0.1**:**6379**>**info replication

# Replication

role**:**master

connected\_slaves**:**2

slave0**:**ip**=**127.0.0.1**,**port**=**6380**,**state**=**online**,**offset**=**57**,**lag**=**1

slave1**:**ip**=**127.0.0.1**,**port**=**6381**,**state**=**online**,**offset**=**57**,**lag**=**0

master\_repl\_offset**:**57

...

可见其连接了两个从库。然后用同样的方法查看两个从库的配置：

127.0.0.1**:**6380**>**info replication

# Replication

role**:**slave

master\_host**:**127.0.0.1

master\_port**:**6379

master\_link\_status**:**up

**...**

127.0.0.1**:**6381**>**info replication

# Replication

role**:**slave

master\_host**:**127.0.0.1

master\_port**:**6379

master\_link\_status**:**up

**...**

接下来开始配置哨兵。建立一个配置文件sentinel.conf，内容为：

sentinel monitor mymaster 127.0.0.1 6379 1

其中mymaster表示要监控的主库的名字，可以自己定义。这个名字必须仅由大小写字母、数字和”.-\_”这3个字符组成。后两个参数表示主库的IP地址和端口号。最后的1表示最低通过票数，后面会介绍。

接下来启动 Sentinel进程，并将上述配置文件的路径传递给哨兵：

# redis-sentinel /root/sentinel.conf

配置哨兵监控一个系统时，只需要配置其监控主库即可，哨兵会自动发现所有复制该主库的从库，具体原理后面会详细介绍。

启动哨兵后，哨兵输出如下内容：

**...**

2112**:**X 06 Dec 08**:**28**:**12.550 # Sentinel runid is ba22decd98aff96e61faca44d7bfd1b4a911cc26

2112**:**X 06 Dec 08**:**28**:**12.550 # **+**monitor master mymaster 127.0.0.1 6379 quorum 1

2112**:**X 06 Dec 08**:**28**:**13.550 **\*** **+**slave slave 127.0.0.1**:**6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6379

2112**:**X 06 Dec 08**:**28**:**13.568 **\*** **+**slave slave 127.0.0.1**:**6381 127.0.0.1 6381 @ mymaster 127.0.0.1 6379

其中”+slave”表示新发现了从库，可见哨兵成功地发现了两个从库。

现在哨兵已经在监控这3个Redis实例了，这时将主库关闭（杀死进程或使用 shutdown 命令），等待指定时间后（可配置，默认为 30 秒），哨兵会输出如下内容：

2112**:**X 06 Dec 08**:**33**:**40.709 # **+**sdown master mymaster 127.0.0.1 6379

2112**:**X 06 Dec 08**:**33**:**40.711 # **+**odown master mymaster 127.0.0.1 6379 #quorum 1**/**1

**...**

其中”+sdown”表示哨兵主观认为主库停止服务了，而”+odown”则表示哨兵客观认为主库停止服务了，关于主观和客观的区别后文会详细介绍。此时哨兵开始执行故障恢复，即挑选一个从库，将其升级为主库。输出如下内容：

2112**:**X 06 Dec 08**:**33**:**40.713 # **+**try**-**failover master mymaster 127.0.0.1 6379

**...**

2112**:**X 06 Dec 08**:**33**:**42.861 # **+**failover**-**end master mymaster 127.0.0.1 6379

2112**:**X 06 Dec 08**:**33**:**42.862 # **+switch-**master mymaster 127.0.0.1 6379 127.0.0.1 6381

2112**:**X 06 Dec 08**:**33**:**42.864 **\*** **+**slave slave 127.0.0.1**:**6380 127.0.0.1 6380 @ mymaster 127.0.0.1 6381

2112**:**X 06 Dec 08**:**33**:**42.864 **\*** **+**slave slave 127.0.0.1**:**6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6381

2112**:**X 06 Dec 08**:**34**:**12.932 # **+**sdown slave 127.0.0.1**:**6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6381

“+try-failover”表示哨兵开始进行故障恢复，”+failover-end”表示哨兵完成故障恢复，期间涉及的内容比较复杂，包括领头哨兵的选举、备选从库的选择等，放到后面介绍，此处只需要关注最后4条输出。”+switch-master”表示主库从6379端口迁移到6381端口，即6381端口的从库升级为主库，两个”+slave”则列出了新主库的两个从库，端口分别为6380和6379。其中6379就是之前停止服务的主库。

哨兵并没有彻底清除停止服务实例的信息，这是因为停止服务的实例可能会在之后的某个时间恢复服务，这时哨兵会让其重新加入进来，所以当实例停止服务后，哨兵会更新该实例的信息，使得当其重新加入后可以按照当前信息继续对外提供服务。此例中6379端口的主库实例停止服务了，而6381 端口的从库已经升级为主库，当6379端口的实例恢复服务后，会转变为6381端口实例的从库来运行，所以哨兵将6379端口实例的信息修改成了 6381端口实例的从库。

故障恢复完成后，可以使用”info replication”重新检查6380和6381两个端口上的实例的复制信息：

127.0.0.1**:**6380**>**info replication

# Replication

role**:**slave

master\_host**:**127.0.0.1

master\_port**:**6381

master\_link\_status**:**up

**...**

127.0.0.1**:**6381**>**info replication

# Replication

role**:**master

connected\_slaves**:**1

slave0**:**ip**=**127.0.0.1**,**port**=**6380**,**state**=**online**,**offset**=**59391**,**lag**=**0

可见6381端口上的实例已经确实升级为主库了，同时6380端口上的实例是其从库。整个故障恢复过程就此完成。

如果此时将6379端口上的实例重新启动，会发生什么情况呢？首先哨兵会监控到这一变化，并输出：

2112**:**X 06 Dec 08**:**57**:**17.572 # **-**sdown slave 127.0.0.1**:**6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6381

2112**:**X 06 Dec 08**:**57**:**27.546 **\*** **+**convert**-**to**-**slave slave 127.0.0.1**:**6379 127.0.0.1 6379 @ mymaster 127.0.0.1 6381

“-sdown”表示实例6379已经恢复了服务（与+sdown相反），同时”+convert-to-slave”表示将6379端口的实例设置为6381端口实例的从库。这时使用”info replication”查看6379端口实例的复制信息为：

127.0.0.1**:**6379**>**info replication

# Replication

role**:**slave

master\_host**:**127.0.0.1

master\_port**:**6381

master\_link\_status**:**up

同时6381端口实例的复制信息为：

127.0.0.1**:**6381**>**info replication

# Replication

role**:**master

connected\_slaves**:**2

slave0**:**ip**=**127.0.0.1**,**port**=**6380**,**state**=**online**,**offset**=**105334**,**lag**=**1

slave1**:**ip**=**127.0.0.1**,**port**=**6379**,**state**=**online**,**offset**=**105334**,**lag**=**1

正如预期一样，6381端口实例的从库变为了两个，6379成功恢复服务。

3：实现原理

哨兵进程启动时读取配置文件的内容，通过如下的配置找出需要监控的主库：

sentinel monitor master**-**name ip redis-port quorum

master-name 是主库的名字，因为考虑到故障恢复后当前监控的系统的主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主库的名字获取当前系统的主库的IP地址和端口号。

ip表示当前系统中主库的地址，redis-port则表示端口号。

quorum表示执行故障恢复操作前，至少需要几个哨兵节点同意，后文会详细介绍。

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：

sentinel monitor mymaster 127.0.0.1 6379 2

sentinel monitor othermaster 192.168.1.3 6380 4

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。

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库的名字使得监控不同主库时可以使用不同的配置参数。例如：

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

sentinel down**-**after**-**milliseconds othermaster 10000

上面的两行配置分别配置了mymaster 和othermaster的”down-after-milliseconds”选项，分别为60000和10000。

哨兵启动后，会与要监控的主库建立两条连接，这两个连接的建立方式与普通的Redis客户端无异。其中一条用来订阅该主库的”\_\_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库发送info等命令来获取主库本身的信息，之前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，所以这时哨兵会使用另外一条连接来发送这些命令。

和主库的连接建立完成后，哨兵会定时执行下面3个操作。

a：每10秒向主库和从库发送info命令；

b：每2秒向主库和从库的”\_\_sentinel\_\_:hello”频道发送自己的信息，也就是说哨兵不但订阅了该频道，而且还会向该频道发布信息，以使其他哨兵得到自己的信息；

c：每1秒向主库、从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。下面分别详细介绍。

首先，发送info命令使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。配置哨兵监控 Redis 主从系统时只需要指定主库的信息即可，因为哨兵正是借助info命令来获取所有复制该主库的从库信息的。

启动后，哨兵向主库发送info命令得到从库列表，而后对每个从库同样建立两个连接，两个连接的作用和与主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库发送info命令来获取更新信息，并进行相应操作。比如对新增的从库建立连接并加入监控列表，对主从库的角色变化（由故障恢复操作引起）进行信息更新等。

接下来哨兵向主从库的”\_\_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：

<哨兵的地址>, <哨兵的端口>, <哨兵运行ID>, <哨兵的配置版本>, <主库的名字>, <主库的地址>, <主库的端口>, <主库的配置版本>

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，则将其加入已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库的配置版本，如果该版本比当前记录的主库的版本高，则更新主库的数据。配置版本的作用会在后面详细介绍。

实现了自动发现从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。

发送ping的时间间隔与”down-after-milliseconds”选项有关，最长间隔为1秒。当”down-after-milliseconds”的值小于1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1秒时，哨兵会每隔1秒发送一 次ping命令。

如果超过”down-after-milliseconds”指定时间后，被ping的节点仍未回复，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，该节点已经下线。如果该节点是主库，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：

sentinel monitor mymaster 127.0.0.1 6379 2

该配置表示只有当至少两个哨兵节点（包括当前节点）认为该主库主观下线时，当前哨兵节点才会认为该主库客观下线。

接下来开始进行领头哨兵的选举。虽然当前哨兵节点发现了主库客观下线，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。

a：发现主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。

b：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。

c： 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。

d：若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。

具体过程可以参考Raft算法的过程 http://raftconsensus.github.io/。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。

选出领头哨兵后，领头哨兵开始对主库进行故障恢复。故障恢复的过程相对简单，具体如下：

首先领头哨兵将从停止服务的主库的从库中挑选一个来充当新的主库。挑选的依据如下：

a：所有在线的从库中，选择优先级最高的从库。优先级可以通过”slave-priority”选项来设置；

b：如果有多个最高优先级的从库，则复制的命令偏移量越大（即复制越完整）越优先；

c：如果以上条件都一样，则选择运行ID较小的从库。

选出一个从库后，领头哨兵将向从库发送”slaveof no one”命令，使其升级为主库。而后领头哨兵向其他从库发送slaveof命令来使其成为新主库的从库。

最后一步则是更新内部的记录，将已经停止服务的，旧的主库更新为新的主库的从库，使得当其恢复服务时自动以从库的身份继续服务。

4：哨兵的部署

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：

a：为每个节点（无论是主库还是从库）部署一个哨兵；

b：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。