# Redis删除策略

Redis中有个设置时间过期的功能，即对存储在 Redis 数据库中的值可以设置一个过期时间。作为一个缓存数据库， 这是非常实用的，比如一些 token 或者登录信息，尤其是短信验证码都是有时间限制的，按照传统的数据库处理方式，一般都是自己判断过期，这样无疑会严重影响项目性能。

我们 set key 的时候，都可以给一个 expire time，就是过期时间，通过过期时间我们可以指定这个 key 可以存活的时间，主要可采用**定期删除和惰性删除**两种方案。

* 定期删除：Redis默认是每隔100ms就随机抽取一些设置了过期时间的key，检查其是否过期，如果过期就删除。注意这里是随机抽取的。为什么要随机呢？你想一想假如Redis存了几十万个 key ，每隔100ms就遍历所有的设置过期时间的key的话，就会给CPU带来很大的负载！
* 惰性删除 ：定期删除可能会导致很多过期 key 到了时间并没有被删除掉。所以就有了惰性删除。它是指某个键值过期后,此键值不会马上被删除,而是等到下次被使用的时候,才会被检查到过期,此时才能得到删除,惰性删除的缺点很明显是浪费内存。 除非你的系统去查一下那个 key，才会被Redis给删除掉。这就是所谓的惰性删除！

# Redis如何保证数据一定能删除？

使用定期删除和惰性删除并不能保证一定删除，Redis有一个Redis 内存淘汰机制来确保数据一定会被删除。

在Redis.conf中有一行配置:maxmemory-policy volatile-lru

该配置就是配内存淘汰策略的，主要有以下六种方案：

**1. volatile-lru**：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰;

**2. volatile-ttl**：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰;

**3. volatile-random**：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰;

**4. allkeys-lru**：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰;

**5. allkeys-random**：从数据集（server.db[i].dict）中任意选择数据淘汰;

**6. no-enviction**（驱逐）：禁止驱逐数据，新写入操作会报错

# Redis持久化技术

# AOF日志

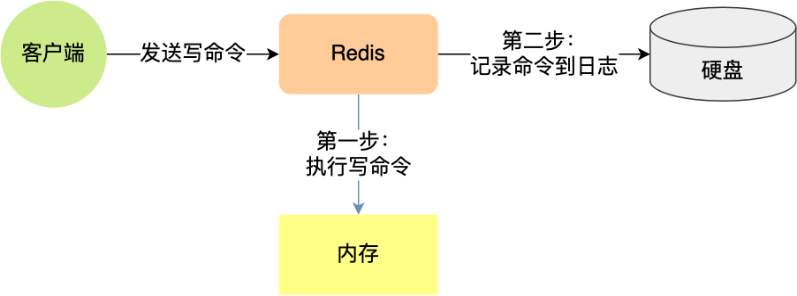
## 流程什么样？

保存写操作命令到日志的持久化方式叫AOF(Append Only File) 持久化功能

Redis 每执行一条写操作命令，就把该命令以追加的方式写入到一个文件里

然后重启 Redis 的时候，先去读取这个文件里的命令，并且执行它，相当于恢复了缓存数据

AOF 日志文件其实就是普通的文本，我们可以通过 cat 命令查看里面的内容



## 先写操作，后记日志的好处？

* 避免额外的检查开销
* 不会阻塞当前写操作命令的执行

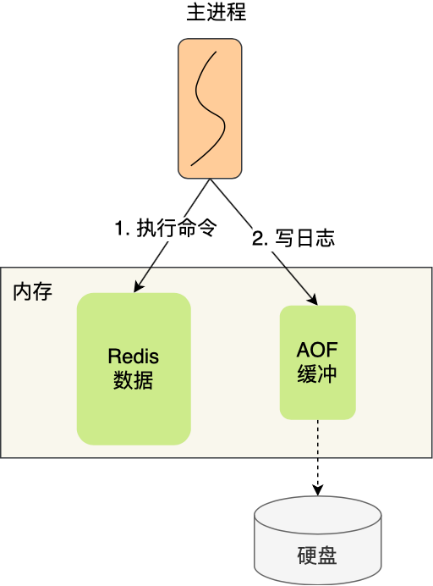
## AOF的问题

### 丢失数据的风险

* 执行写操作命令和记录日志是两个过程，那当 Redis 在还没来得及将命令写入到硬盘时，服务器发生宕机了，这个数据就会有丢失的风险。

### 可能会给「下一个」命令带来阻塞风险

* 因为将命令写入到日志的这个操作也是在主进程完成的（执行命令也是在主进程），也就是说这两个操作是同步的。



* 如果在将日志内容写入到硬盘时，服务器的硬盘的 I/O 压力太大，就会导致写硬盘的速度很慢，进而阻塞住了，也就会导致后续的命令无法执行。

## 写入日志流程

1. Redis 执行完写操作命令后，会将命令追加到 server.aof\_buf 缓冲区；

2. 然后通过 write() 系统调用，将 aof\_buf 缓冲区的数据写入到 AOF 文件，此时数据并没有写入到硬盘，而是拷贝到了内核缓冲区 page cache，等待内核将数据写入硬盘；

3. 具体内核缓冲区的数据什么时候写入到硬盘，由内核决定。

## Redis 提供了 3 种写回硬盘的策略

在 redis.conf 配置文件中的 appendfsync 配置项可以有以下 3 种参数可填：

* Always，这个单词的意思是「总是」，所以它的意思是每次写操作命令执行完后，同步将 AOF 日志数据写回硬盘；
* Everysec，这个单词的意思是「每秒」，所以它的意思是每次写操作命令执行完后，先将命令写入到 AOF 文件的内核缓冲区，然后每隔一秒将缓冲区里的内容写回到硬盘；
* No，意味着不由 Redis 控制写回硬盘的时机，转交给操作系统控制写回的时机，也就是每次写操作命令执行完后，先将命令写入到 AOF 文件的内核缓冲区，再由操作系统决定何时将缓冲区内容写回硬盘。

## 3种策略的比较

* 3 种写回策略都无法能完美解决「主进程阻塞」和「减少数据丢失」的问题，因为两个问题是对立的，偏向于一边的话，就会要牺牲另外一边，原因如下：

• Always 策略的话，可以最大程度保证数据不丢失，但是由于它每执行一条写操作命令就同步将 AOF 内容写回硬盘，所以是不可避免会影响主进程的性能；

• No 策略的话，是交由操作系统来决定何时将 AOF 日志内容写回硬盘，相比于 Always 策略性能较好，但是操作系统写回硬盘的时机是不可预知的，如果 AOF 日志内容没有写回硬盘，一旦服务器宕机，就会丢失不定数量的数据。

• Everysec 策略的话，是折中的一种方式，避免了 Always 策略的性能开销，也比 No 策略更能避免数据丢失，当然如果上一秒的写操作命令日志没有写回到硬盘，发生了宕机，这一秒内的数据自然也会丢失。

* 如何选择？

• 如果要高性能，就选择 No 策略；

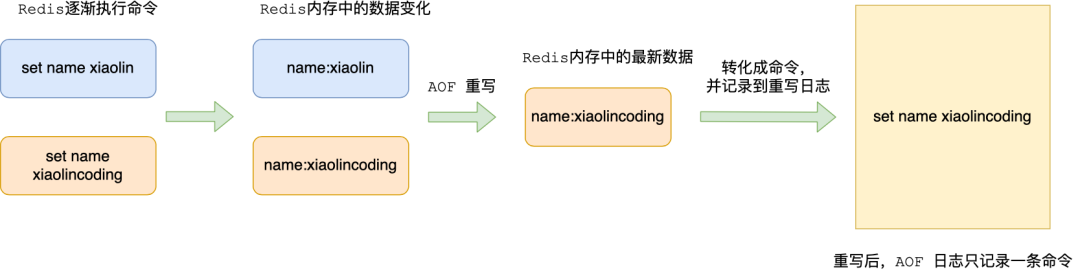
• 如果要高可靠，就选择 Always 策略；

• 如果允许数据丢失一点，但又想性能高，就选择 Everysec 策略。

## 重写机制

### 流程

在重写时，读取当前内存数据库中的所有键值对，然后将每一个键值对用一条命令记录到「新的 AOF 文件」，等到全部记录完后，就将新的 AOF 文件替换掉现有的 AOF 文件。



重写机制的妙处在于，尽管某个键值对被多条写命令反复修改，最终也只需要根据这个「键值对」当前的最新状态，然后用一条命令去记录键值对，代替之前记录这个键值对的多条命令，这样就减少了 AOF 文件中的命令数量。这就相当于压缩了 AOF 文件，使得 AOF 文件体积变小了。最后在重写工作完成后，将新的 AOF 文件覆盖现有的 AOF 文件。

## 重写方式

如果AOF文件本身过大时，重写的过程中是耗时的，所以重写过程不能放在Redis主进程中。所以，Redis 的重写 AOF 过程是由后台子进程 bgrewriteaof 来完成的

### 子进程重写的好处？

* 主进程可以继续处理命令请求，从而避免阻塞主进程
* 子进程带有主进程的数据副本，这里使用子进程而不是线程，因为如果是使用线程，多线程之间会共享内存，那么在修改共享内存数据的时候，需要通过加锁来保证数据的安全，而这样就会降低性能。而使用子进程，创建子进程时，父子进程是共享内存数据的，不过这个共享的内存只能以只读的方式，而当父子进程任意一方修改了该共享内存，就会发生「写时复制」，于是父子进程就有了独立的数据副本，就不用加锁来保证数据安全。

# RDB快照

## 如何使用

save

bgsave

* 执行了 bgsava 命令，会创建一个子进程来生成 RDB 文件，这样可以避免主线程的阻塞；

bgsava 命令：save 300 10 满足限定时间范围内key的变化数量达到指定数量即进行持久化

* 别看选项名叫 save，实际上执行的是 bgsave 命令，也就是会创建子进程来生成 RDB 快照文件。
* 命令的意思是300 秒之内，对数据库进行了至少 10 次修改；

## 特点

* 在 Redis 恢复数据时， RDB 恢复数据的效率会比 AOF 高些，因为直接将 RDB 文件中的数据读入内存就可以，不需要像 AOF 那样还需要额外执行操作命令的步骤才能恢复数据。
* RDB 文件的加载工作是在服务器启动时自动执行的，Redis 并没有提供专门用于加载 RDB 文件的命令。
* Redis 的快照是全量快照，也就是说每次执行快照，都是把内存中的「所有数据」都记录到磁盘中。

## 执行期间数据修改问题

手动保存，因为save命令会加入到指令的单线程任务执行序列中，执行了 save 命令，就会在主线程生成 RDB 文件，由于和执行操作命令在同一个线程，所以如果写入 RDB 文件的时间太长，会阻塞主线程；

执行 bgsave 过程中，由于是交给子进程来构建 RDB 文件，Redis主线程依然可以继续处理操作命令的，也就是数据是能被修改的。

同样使用写时复制技术（Copy-On-Write, COW）

没有数据修改的情况

所以，创建 bgsave 子进程后，由于共享父进程的所有内存数据，于是就可以直接读取主线程里的内存数据，并将数据写入到 RDB 文件。

有数据修改的情况

如果主线程要修改共享数据里的某一块数据（比如键值对 A）时，就会发生写时复制，于是这块数据的物理内存就会被复制一份（键值对 A'），然后主线程在这个数据副本（键值对 A'）进行修改操作。与此同时，bgsave 子进程可以继续把原来的数据（键值对 A）写入到 RDB 文件。

所以 Redis 在使用 bgsave 快照过程中，如果主线程修改了内存数据，不管是否是共享的内存数据，RDB 快照都无法写入主线程刚修改的数据，因为此时主线程的内存数据和子线程的内存数据已经分离了，子线程写入到 RDB 文件的内存数据只能是原本的内存数据，只能交由下一次的 bgsave 快照。极端情况下，如果所有的共享内存都被修改，则此时的内存占用是原先的 2 倍。所以，针对写操作多的场景，要留意快照过程中内存的变化，防止内存被占满。

## RDB的缺点

在服务器发生故障时，丢失的数据会比 AOF 持久化的方式更多，因为 RDB 快照是全量快照的方式，因此执行的频率不能太频繁，否则会影响 Redis 性能，而 AOF 日志可以以秒级的方式记录操作命令，所以丢失的数据就相对更少。

如果频率太低，两次快照间一旦服务器发生宕机，就可能会比较多的数据丢失。

如果频率太高，频繁写入磁盘和创建子进程会带来额外的性能开销。

通常可能设置至少 5 分钟才保存一次快照，这时如果 Redis 出现宕机等情况，则意味着最多可能丢失 5 分钟数据。

有没有什么方法不仅有 RDB 恢复速度快的优点和，又有 AOF 丢失数据少的优点呢？

## AOF和RDB混合持久化

混合持久化工作在 AOF 日志重写过程

持久化格式

* AOF 文件的前半部分是 RDB 格式的全量数据，后半部分是 AOF 格式的增量数据。



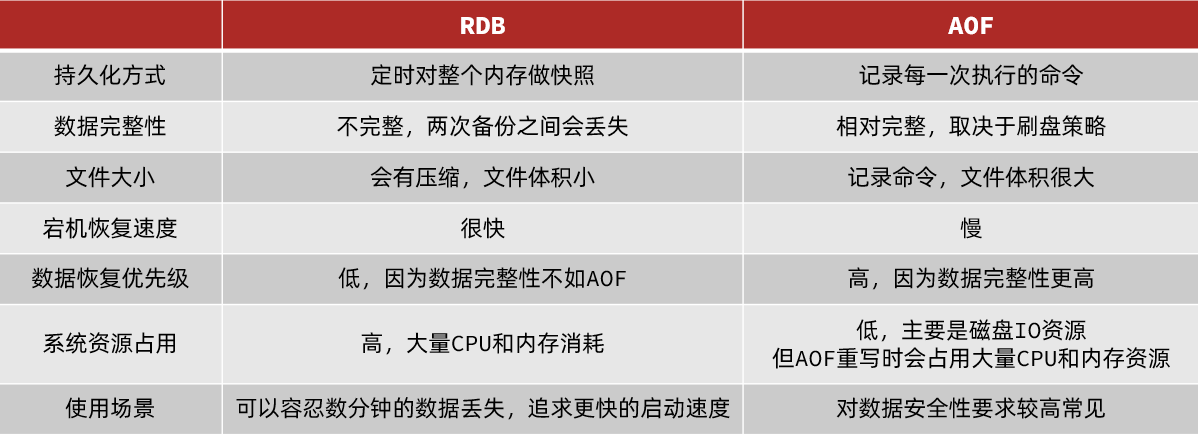
流程

* 在 AOF 重写日志时，fork 出来的重写子进程会先将与主线程共享的内存数据以 RDB 方式写入到 AOF 文件
* 然后主线程处理的操作命令会被记录在重写缓冲区里，重写缓冲区里的增量命令会以 AOF 方式写入到 AOF 文件
* 写入完成后通知主进程将新的含有 RDB 格式和 AOF 格式的 AOF 文件替换旧的的 AOF 文件。

优点

* 重启 Redis 加载数据的时候，由于前半部分是 RDB 内容，这样加载的时候速度会很快。
* 加载完 RDB 的内容后，才会加载后半部分的 AOF 内容，由于存储的是主线程处理的操作命令，可以使得数据更少的丢失。

# RDB与AOF对比



# AOF和RDB如何选择？

对数据非常敏感，建议使用默认的AOF持久化方案

AOF持久化策略使用everysecond，每秒钟fsync一次。该策略redis仍可以保持很好的处理性能，当出现问题时，最多丢失0-1秒内的数据。

注意：由于AOF文件存储体积较大，且恢复速度较慢

数据呈现阶段有效性，建议使用RDB持久化方案

数据可以良好的做到阶段内无丢失（该阶段是开发者或运维人员手工维护的），且恢复速度较快，阶段点数据恢复通常采用RDB方案

注意：利用RDB实现紧凑的数据持久化会使Redis降的很低，慎重总结

双保险策略，同时开启 RDB 和 AOF，重启后，Redis优先使用 AOF 来恢复数据，降低丢失数据的量

# Redis缓存雪崩

## 定义

此时有大量的用户请求，都无法在 Redis 中处理，于是全部请求都直接访问数据库，从而导致数据库的压力骤增，严重的会造成数据库宕机，造成整个系统崩溃

## 原因

### 大量缓存数据在同一时间过期（缓存集中失效）

### Redis 故障宕机

## 解决策略

### 缓存集中失效

* 均匀设置过期时间

• 在对缓存数据设置过期时间时，给这些数据的过期时间加上一个随机数，这样就保证数据不会在同一时间过期。

* 使用互斥锁

• 通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和构建缓存，当缓存构建完成后，再释放锁。未能获取互斥锁的请求，要么等待锁释放后重新读取缓存，要么就返回空值或者默认值。

• 实现互斥锁的时候，最好设置超时时间，不然第一个请求拿到了锁，然后这个请求发生了某种意外而一直阻塞，一直不释放锁，这时其他请求也一直拿不到锁，整个系统就会出现无响应的现象。

• 加锁排队只是为了减轻数据库的压力，并没有提高系统吞吐量。假设在高并发下，缓存重建期间key是锁着的，这是过来1000个请求999个都在阻塞的。同样会导致用户等待超时，这是个治标不治本的方法。

* 使用双缓存策略

• 一个主缓存，一个备用缓存（不设置过期时间），他们只是key不一样，value的值是一样的。当业务线程访问不到「主 key 」的缓存数据时，就直接返回「备 key 」的缓存数据，然后在更新缓存的时候，同时更新「主 key 」和「备 key 」的数据。

* 后台线程更新缓存

• 业务线程不再负责更新缓存，缓存也不设置有效期，而是让缓存“永久有效”，并将更新缓存的工作交由后台线程定时更新。

• 第一种方式，后台线程不仅负责定时更新缓存，而且也负责频繁地检测缓存是否有效，检测到缓存失效了，原因可能是系统紧张而被淘汰的，于是就要马上从数据库读取数据，并更新到缓存。这种方式的检测时间间隔不能太长，太长也导致用户获取的数据是一个空值而不是真正的数据。

• 第二种方式，在业务线程发现缓存数据失效后（缓存数据被淘汰），通过消息队列发送一条消息通知后台线程更新缓存，后台线程收到消息后，在更新缓存前可以判断缓存是否存在，存在就不执行更新缓存操作；不存在就读取数据库数据，并将数据加载到缓存。这种方式相比第一种方式缓存的更新会更及时，用户体验也比较好。

### 故障宕机

* 服务熔断或请求限流

• 启动服务熔断机制，暂停业务应用对缓存服务的访问，直接返回错误，不用再继续访问数据库，从而降低对数据库的访问压力，保证数据库系统的正常运行，然后等到 Redis 恢复正常后，再允许业务应用访问缓存服务。

• 启用请求限流机制，只将少部分请求发送到数据库进行处理，再多的请求就在入口直接拒绝服务，等到 Redis 恢复正常并把缓存预热完后，再解除请求限流的机制。

* 构建Redis缓存高可用集群

• 最好通过主从节点的方式构建 Redis 缓存高可靠集群。

• 如果 Redis 缓存的主节点故障宕机，从节点可以切换成为主节点，继续提供缓存服务，避免了由于 Redis 故障宕机而导致的缓存雪崩问题。

# Redis缓存击穿

## 定义

如果缓存中的某个热点数据过期了，此时大量的请求访问了该热点数据，就无法从缓存中读取，直接访问数据库，数据库很容易就被高并发的请求冲垮

## 和雪崩的区别

缓存击穿与缓存雪崩的区别是这里针对的是某一热门key缓存，而雪崩针对的是大量缓存的集中失效。

## 解决策略

### 二级缓存

* 设置不同的失效时间，保障不会被同时淘汰就行

### 互斥锁方案

* 保证同一时间只有一个业务线程更新缓存，未能获取互斥锁的请求，要么等待锁释放后重新读取缓存，要么就返回空值或者默认值。

### 后台刷新数据

* 由后台异步更新缓存，或者在热点数据准备要过期前，提前通知后台线程更新缓存以及重新设置过期时间；

# Redis缓存穿透

## 定义

当用户访问的数据，既不在缓存中，也不在数据库中，导致请求在访问缓存时，发现缓存缺失，再去访问数据库时，发现数据库中也没有要访问的数据，没办法构建缓存数据。那么当有大量这样的请求到来时，数据库的压力骤增，这就是缓存穿透的问题。

## 原因

### 业务误操作，缓存中的数据和数据库中的数据都被误删除了，所以导致缓存和数据库中都没有数据；

### 黑客恶意攻击，故意大量访问某些读取不存在数据的业务；

## 解决策略

### 非法请求的限制

* 当有大量恶意请求访问不存在的数据的时候，也会发生缓存穿透，因此在 API 入口处我们要判断求请求参数是否合理，请求参数是否含有非法值、请求字段是否存在，如果判断出是恶意请求就直接返回错误，避免进一步访问缓存和数据库。

### 缓存空值或者默认值

* 当我们线上业务发现缓存穿透的现象时，可以针对查询的数据，在缓存中设置一个空值或者默认值，这样后续请求就可以从缓存中读取到空值或者默认值，返回给应用，而不会继续查询数据库。

### 使用布隆过滤器判断数据是否在数据库

* 我们可以在写入数据库数据时，使用布隆过滤器做个标记，然后在用户请求到来时，业务线程确认缓存失效后，可以通过查询布隆过滤器快速判断数据是否存在，如果不存在，就不用通过查询数据库来判断数据是否存在。
* 布隆过滤器由「初始值都为 0 的位图数组」和「 N 个哈希函数」两部分组成。

• 第一步，使用 N 个哈希函数分别对数据做哈希计算，得到 N 个哈希值；

• 第二步，将第一步得到的 N 个哈希值对位图数组的长度取模，得到每个哈希值在位图数组的对应位置。

• 第三步，将每个哈希值在位图数组的对应位置的值设置为 1；

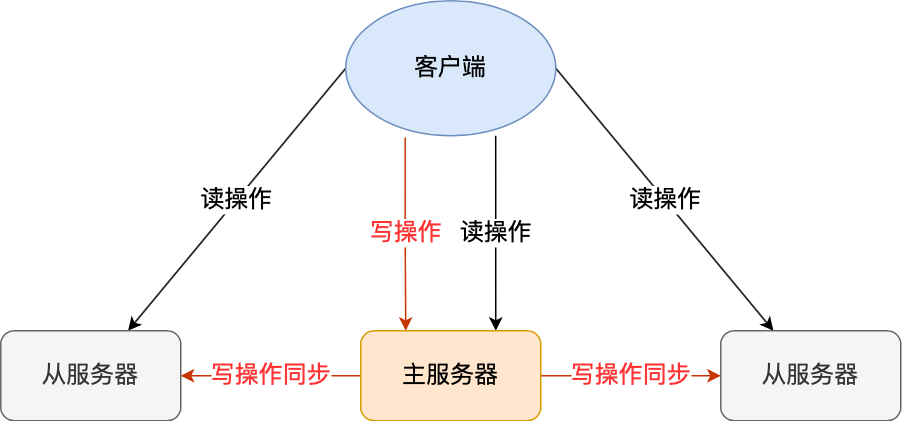
• 布隆过滤器由于是基于哈希函数实现查找的，高效查找的同时存在哈希冲突的可能性

• 查询布隆过滤器说数据存在，并不一定证明数据库中存在这个数据，但是查询到数据不存在，数据库中一定就不存在这个数据。

# Redis主从复制

Redis 提供了主从复制模式，来解决高可用问题，这样可以保证多台服务器的数据一致性，且主从服务器之间采用的是「读写分离」的方式。

所有的数据修改只在主服务器上进行，然后将最新的数据同步给从服务器，这样就使得主从服务器的数据是一致的。



# 主从复制两种模式

全量复制

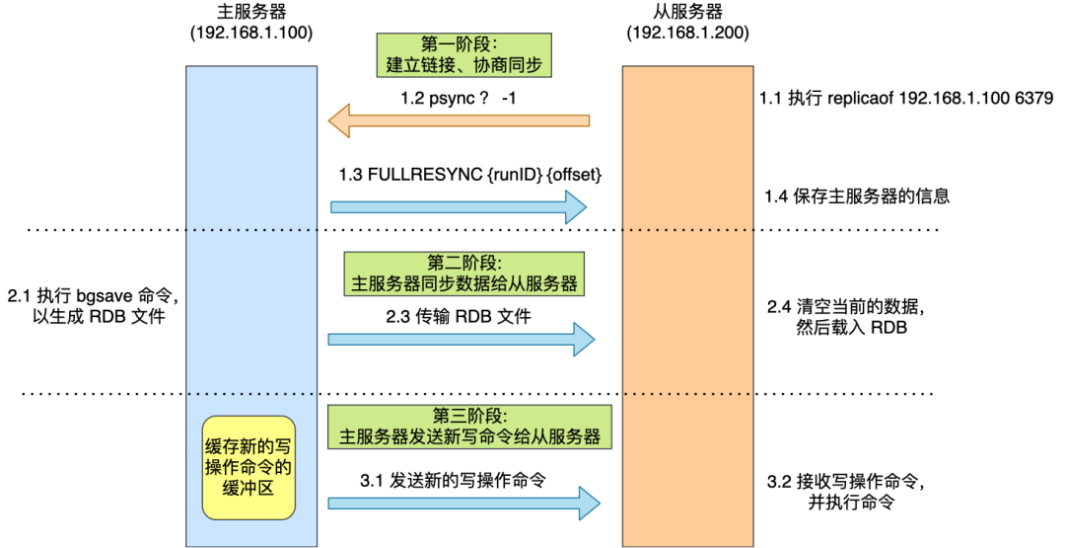
增量复制

# 第一次同步（全量复制）

可以使用 replicaof（Redis 5.0 之前使用 slaveof）命令形成主服务器和从服务器的关系。

服务器B上执行，replicaof <服务器 A 的 IP 地址> <服务器 A 的 Redis 端口号>，服务器 B 就会变成服务器 A 的「从服务器」，然后与主服务器进行第一次同步。

## 可分为哪三个阶段？



### 第一阶段是建立链接、协商同步、全量复制的准备工作

* 执行了 replicaof 命令后，从服务器就会给主服务器发送 psync 命令，表示要进行数据同步。
* psync 命令包含两个参数，分别是主服务器的 runID 和复制进度 offset。
  + runID，每个 Redis 服务器在启动时都会自动生产一个随机的 ID 来唯一标识自己。当从服务器和主服务器第一次同步时，因为不知道主服务器的 run ID，所以将其设置为 "?"。
  + offset，偏移量，随着记录在repl\_baklog中的数据增多而逐渐增大。slave完成同步时也会记录当前同步的offset。如果slave的offset小于master的offset，说明slave数据落后于master，需要更新。第一次同步时，其值为 -1。
* 主服务器收到 psync 命令后，会用 FULLRESYNC 作为响应命令返回给对方。并且这个响应命令会带上两个参数：主服务器的 runID 和主服务器目前的复制进度 offset。从服务器收到响应后，会记录这两个值。
* FULLRESYNC 响应命令的意图是采用全量复制的方式，也就是主服务器准备把所有的数据都同步给从服务器。

### 第二阶段是主服务器同步数据给从服务器

* 主服务器会执行 bgsave 命令来生成 RDB 文件，然后把文件发送给从服务器。主服务器生成 RDB 这个过程是不会阻塞主线程的，也就是说 Redis 依然可以正常处理命令
* 从服务器收到 RDB 文件后，会先清空当前的数据，然后载入 RDB 文件。
* 但是这期间的写操作命令并没有记录到刚刚生成的 RDB 文件中，这时主从服务器间的数据就不一致了。
* 那么为了保证主从服务器的数据一致性，主服务器会将在 RDB 文件生成后收到的写操作命令，写入到 replication buffer 缓冲区里。

### 第三阶段是主服务器发送新写操作命令给从服务器

* 在主服务器生成的 RDB 文件发送后，然后将 replication buffer 缓冲区里所记录的写操作命令发送给从服务器，然后从服务器重新执行这些操作。

## 从服务器过多怎么办？

如果从服务器数量非常多，而且都与主服务器进行全量同步的话，就会带来两个问题：

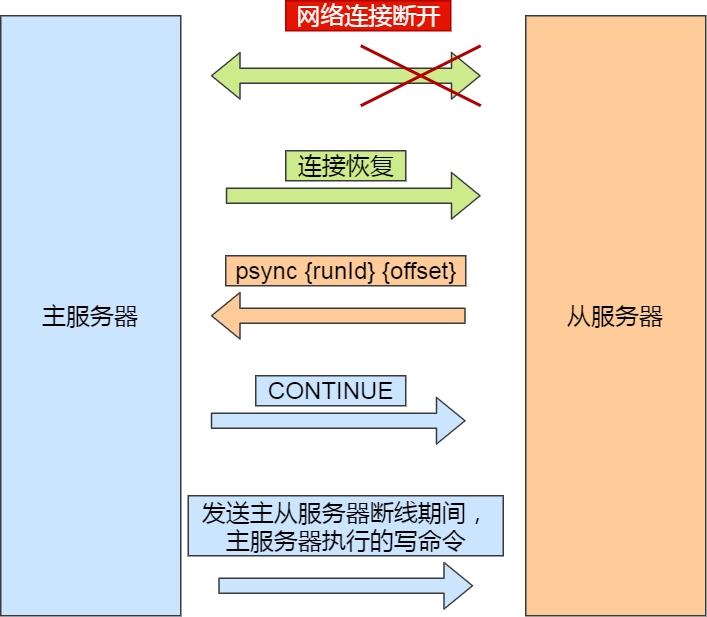
* 由于是通过 bgsave 命令来生成 RDB 文件的，那么主服务器就会忙于使用 fork() 创建子进程，如果主服务器的内存数据非大，在执行 fork() 函数时是会阻塞主线程的，从而使得 Redis 无法正常处理请求；
* 传输 RDB 文件会占用主服务器的网络带宽，会对主服务器响应命令请求产生影响。

Redis 从服务器可以有自己的从服务器，我们可以把拥有从服务器的从服务器当作【一级从服务器】，它不仅可以接收主服务器的同步数据，自己也可以同时作为主服务器的形式将数据同步给自己的从服务器【二级从服务器】

具体做法在【二级从服务器】上执行 replicaof <目标服务器的IP> 6379 命令。使它作为目标服务器的从服务器。

# 增量复制

主从服务器间的网络断开又恢复或者重启后，从主从服务器会采用增量复制的方式继续同步，也就是只会把网络断开期间主服务器接收到的写操作命令，同步给从服务器。



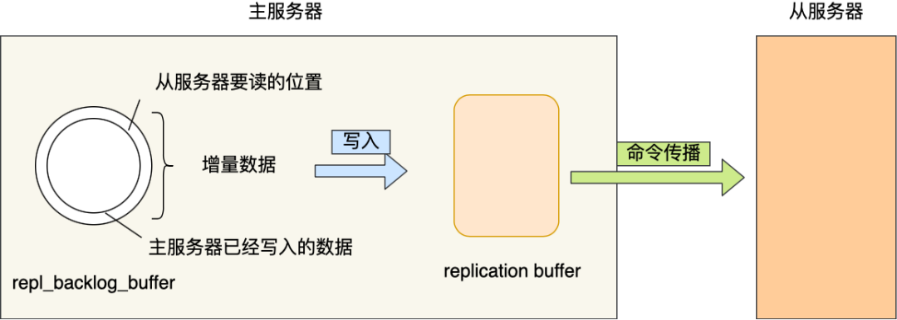
## 可分为哪三步？

从服务器在恢复网络后，会发送 psync 命令给主服务器，此时的 psync 命令将自己的复制偏移量 slave\_repl\_offset 发送给主服务器

主服务器收到该命令后，然后用 CONTINUE 响应命令告诉从服务器接下来采用增量复制的方式同步数据

主服务器根据自己的 master\_repl\_offset 和 slave\_repl\_offset 之间的差距，然后来决定对从服务器执行哪种同步操作：

* 如果判断出从服务器要读取的数据还在 repl\_backlog\_buffer 缓冲区里，就会将增量的数据写入到 replication buffer 缓冲区，这个缓冲区我们前面也提到过，它是缓存将要传播给从服务器的命令。



* 相反，如果判断出从服务器要读取的数据已经不存在 repl\_backlog\_buffer 缓冲区里，那么主服务器将采用全量同步的方式

## 哪些是增量数据呢？

增量数据在repl\_backlog\_buffer里，它是一个「环形」缓冲区，用于主从服务器断连后，从中找到差异的数据。该缓冲区中还包含一个偏移量replication offset， 它标记上面那个缓冲区的同步进度，主从服务器都有各自的偏移量，主服务器使用 master\_repl\_offset 来记录自己「写」到的位置，从服务器使用 slave\_repl\_offset 来记录自己「读」到的位置。

## 缓冲区的增量数据什么时候添加的？

在主服务器进行命令传播时，不仅会将写命令发送给从服务器，还会将写命令写入到 repl\_backlog\_buffer 缓冲区里，因此 这个缓冲区里会保存着最近传播的写命令。

## 缓冲区大小的问题

repl\_backlog\_buffer 缓行缓冲区的默认大小是 1M，并且由于它是一个环形缓冲区，所以当缓冲区写满后，主服务器的写入速度远超于从服务器的读取速度，缓冲区的数据就会被覆盖。一旦从服务器想读的数据已经被覆盖了，主服务器就会采用全量同步，这个方式比增量同步的性能损耗要大很多。

我们应该调整下 repl\_backlog\_buffer 缓冲区大小，尽可能的大一些，减少出现从服务器要读取的数据被覆盖的概率，从而使得主服务器采用增量同步的方式。

repl\_backlog\_buffer 最小的大小可以根据这上面这个公式估算

* + second 为从服务器断线后重新连接上主服务器所需的平均 时间(以秒计算)。
  + write\_size\_per\_second 则是主服务器平均每秒产生的写命令数据量大小。
  + 为了应对一些突发的情况，可以将 缓冲区的大小设置为此基础上的 2 倍，也就是 10 MB。

# 什么是哨兵

哨兵(sentinel) 是一个分布式系统，用于对主从结构中的每台服务器进行监控，当出现故障时通过投票机制选择新的master并将所有slave连接到新的master。

哨兵也是一台redis服务器，只是不提供数据服务

# 哨兵的作用

## 监控

Sentinel 会不断检查master和slave是否按预期工作

* 获取各个sentinel的状态（是否在线）
* 获取master的状态
  + master属性
  + 各个slave的详细信息
* 获取所有slave的状态（根据master中的slave信息）
  + slave属性

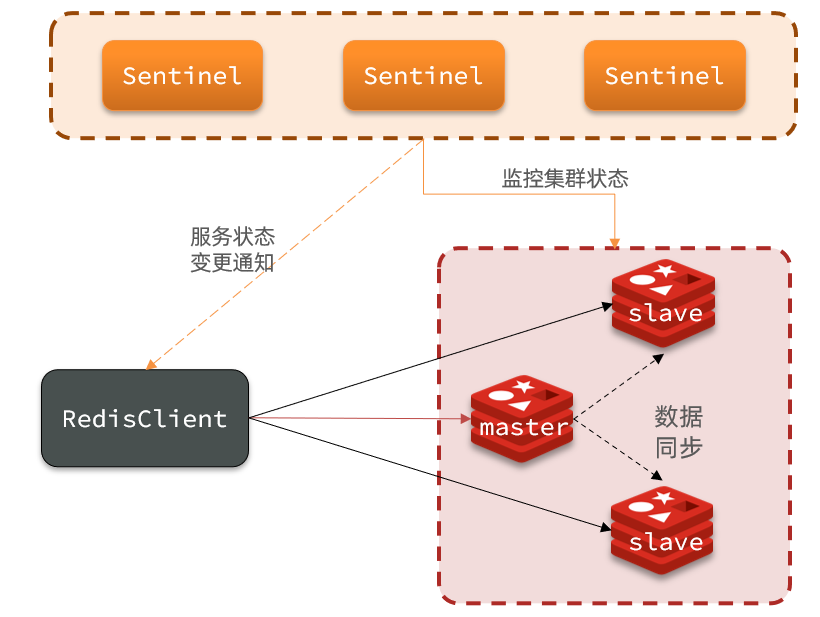
## 自动故障恢复

如果master故障，Sentinel会将一个slave提升为master。当故障实例恢复后也以新的master为主

## 通知

Sentinel充当Redis客户端的服务发现来源，当集群发生故障转移时，会将最新信息推送给Redis的客户端

# 哨兵结构



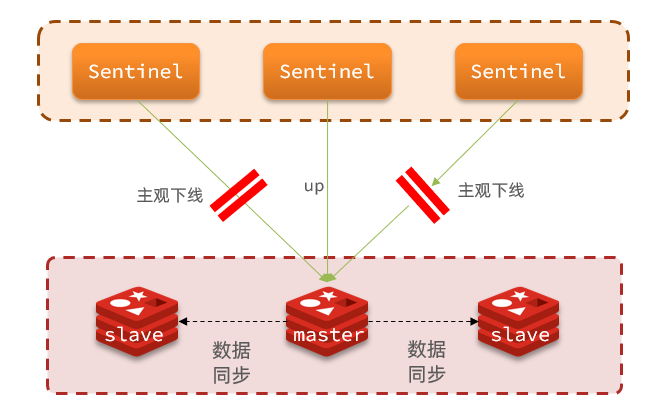
# 哨兵如何监控节点状态？

Sentinel基于心跳机制监测服务状态，每隔1秒向集群的每个实例发送ping命令

虽然哨兵检测的是主节点的状态，但是也能够获取到从节点的状态。从节点的主从关系能够从主节点上的info replication命令中获取。

主观下线：如果某sentinel节点发现某实例未在规定时间响应，则认为该实例主观下线。

客观下线：若超过指定数量（quorum）的sentinel都认为该实例主观下线，则该实例客观下线。quorum值最好超过Sentinel实例数量的一半。



# 哨兵如何选择新的master？

首先会判断slave节点与master节点断开时间长短，如果超过指定值（down-after-milliseconds \* 10）则会排除该slave节点

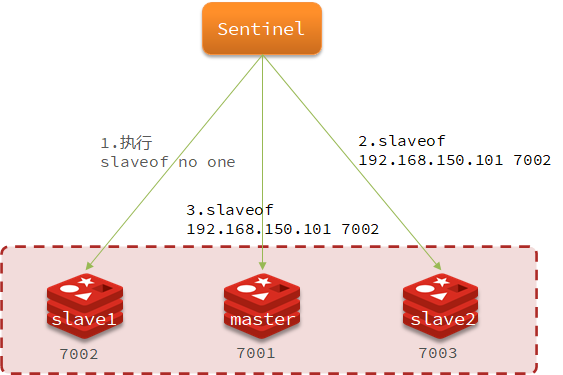
然后判断slave节点的slave-priority值，越小优先级越高，如果是0则永不参与选举

如果slave-prority一样，则判断slave节点的offset值，越大说明数据越新，优先级越高

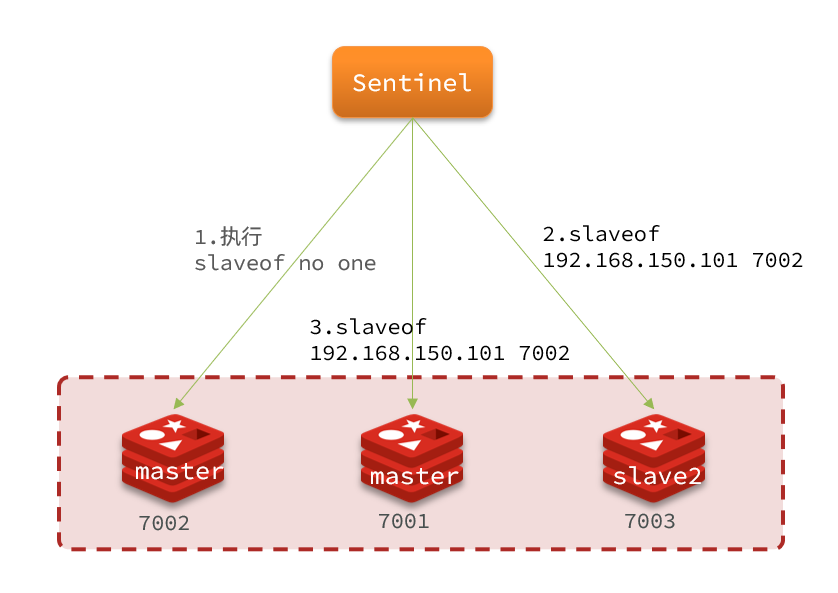
最后是判断slave节点的运行id大小，越小优先级越高。

# 哨兵如何实现故障转移？

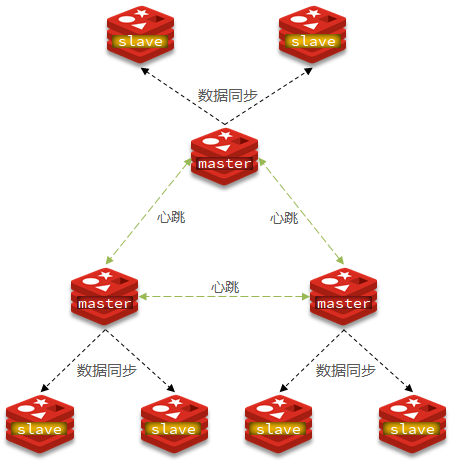
当选中了其中一个slave为新的master后（例如slave1），故障的转移的步骤如下：



1. sentinel给备选的slave1节点发送slaveof no one命令，让该节点成为master
2. sentinel给所有其它slave发送slaveof 192.168.150.101 7002 命令，让这些slave成为新master的从节点，开始从新的master上同步数据。
3. 最后，sentinel将故障节点标记为slave，当故障节点恢复后会自动成为新的master的slave节点
4. 最后的状态



# 什么是分片集群？



在主从集群中，是由一个主服务器，多个从服务器构成的。虽然主从和哨兵可以解决高可用、高并发读的问题。但是依然有两个问题没有解决：

1. 海量数据存储的问题

2. 高并发写问题。

分片集群就是在集群中有多个主服务器，每个主服务器可以有自己的从服务器，这样的集群构成方式叫做分片集群。

分片集群中可以不使用哨兵集群，每台主服务器之间可以传递心跳包来检测各个主从服务器之间的运行状态。

# 分片集群的特点

1. 集群中有多个master，每个master保存不同数据
2. 每个master都可以有多个slave节点
3. master之间通过ping监测彼此健康状态
4. 客户端请求可以访问集群任意节点，最终都会被转发到正确节点（转发槽）

# 搭建分片集群

通过命令来管理集群：

redis-cli --cluster create --cluster-replicas 1 192.168.150.101:7001 192.168.150.101:7002 192.168.150.101:7003 192.168.150.101:8001 192.168.150.101:8002 192.168.150.101:8003

* redis-cli --cluster：代表集群操作命令
* create：代表是创建集群
* --replicas 1：指定集群中每个master的副本个数为1，此时`节点总数 ÷ (replicas + 1)` 得到的就是master的数量。因此节点列表中的前n个就是master，其它节点都是slave节点，随机分配到不同master

创建完集群之后，每台主机自动分配完槽

通过命令可以查看集群状态：

redis-cli -p 7001 cluster nodes

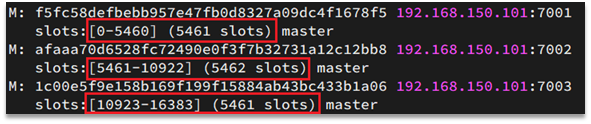
使用集群主机的客户端：

需要给redis-cli加上-c参数才可以

redis-cli -c -p 7001

# 散列插槽

Redis会把每一个master节点映射到0~16383共16384个插槽（hash slot）上，查看集群信息时就能看到：



数据key不是与节点绑定，而是与插槽绑定。redis会根据key的有效部分计算插槽值

* key中包含"{}"，且“{}”中至少包含1个字符，“{}”中的部分是有效部分
* key中不包含“{}”，整个key都是有效部分

计算方式是使用有效key利用CRC16算法得到一个hash值，然后对16384取余，得到的结果就是slot值。

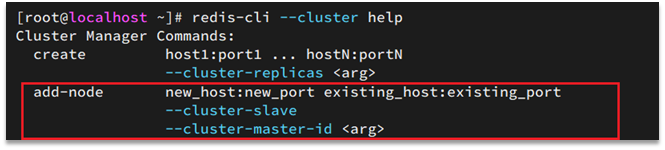
如何将同一类数据固定的保存在同一个Redis实例？

使这一类数据使用相同的有效部分，例如key都以{typeId}为前缀

# 集群伸缩

添加一个节点到集群

redis-cli --cluster提供了很多操作集群的命令，比如，添加节点的命令：



如何添加？

向集群中添加一个新的master节点，并向其中存储 num = 10

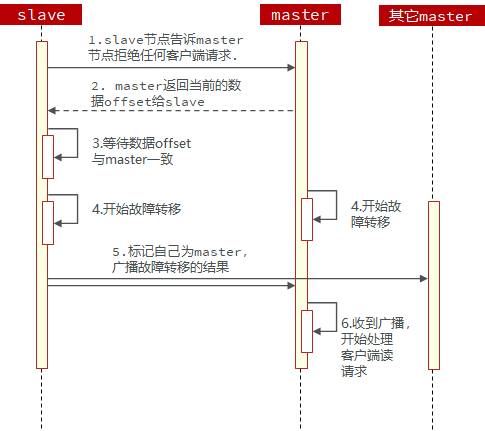
* 启动一个新的redis实例，端口为7004
* 添加7004到之前的集群，并作为一个master节点
* 给7004节点分配插槽，使得num这个key可以存储到7004实例

# 故障转移

如果发现有一台主服务器宕机，那么集群中会自动发现，再确定下线后，自动将其slave服务器提升为master服务器，不使用哨兵的功能

数据迁移

利用cluster failover命令可以手动让集群中的某个master宕机，切换到执行cluster failover命令的这个slave节点，实现无感知的数据迁移。其流程如下：



Failover支持三种不同模式：

* 缺省：默认的流程，如图1~6歩
* force：省略了对offset的一致性校验
* takeover：直接执行第5歩，忽略数据一致性、忽略master状态和其它master的意见