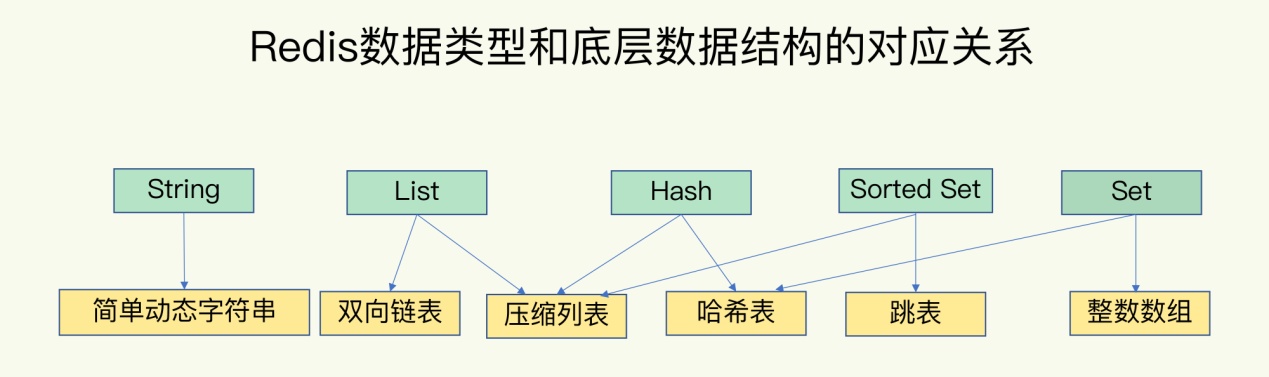
**Reids为什么这么快?**

1. **由于其内存数据库的天然属性。**
2. **归功于他的数据结构。**

**底层的数据机构总共有6种，分别是简单动态字符串、双向链表、压缩链表、哈希表、跳表和整数数组。**

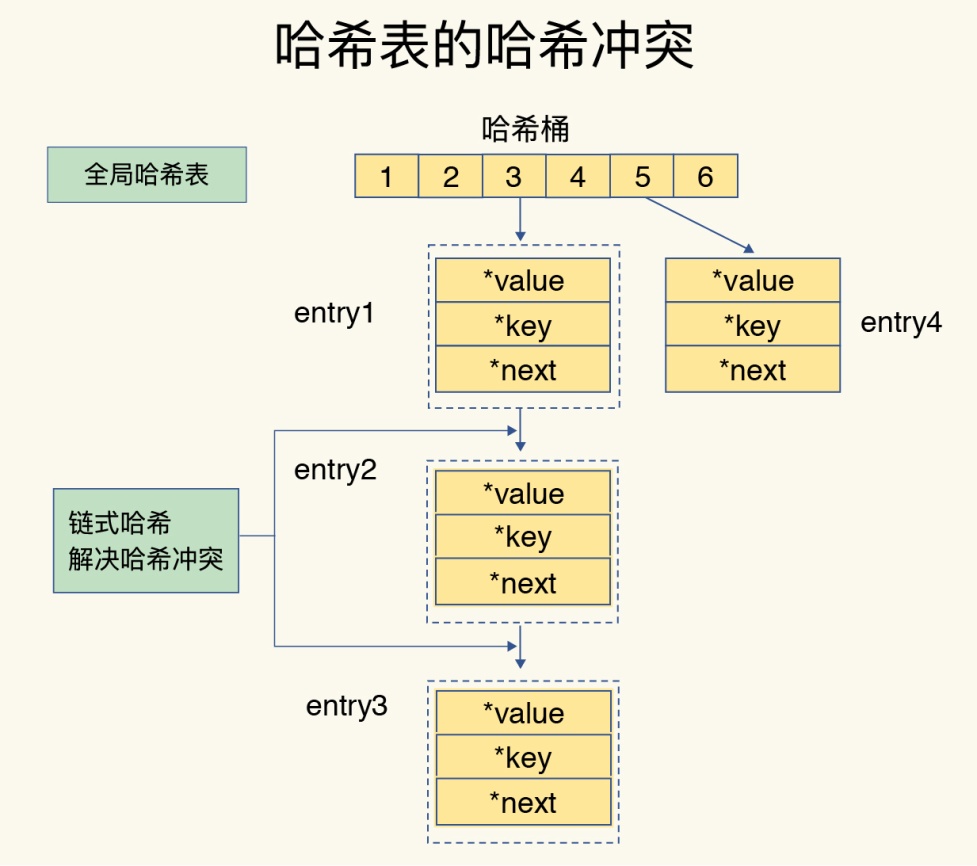


String 的底层类型只有一种数据结构，也就是简单动态字符串。

List、Hash、Set、sortedSet 这四种数据类型都有两种数据结构。

**为什么hash表的操作变慢了？**

哈希冲突，同一个hash桶中的多个元素用一个链表来保存，他们之间以此用指针连接。



如果哈希表里写入的数据越来越多，哈希冲突可能也会越来越多，这就会导致某些哈希冲突链过长，进而导致这个链上的元素查找耗时长，效率降低。对于追求“快”的 Redis 来说，这是不太能接受的。

所以，Redis 会对哈希表做 rehash 操作。rehash 也就是增加现有的哈希桶数量，让逐渐增多的 entry 元素能在更多的桶之间分散保存，减少单个桶中的元素数量，从而减少单个桶中的冲突。那具体怎么做呢？

**Redis 采用了渐进式 rehash**

**Redis为什么使用单线程？**

Redis 单线程是指它对网络 IO 和数据读写的操作采用了一个线程，而采用单线程的一个核心原因是避免多线程开发的并发控制问题。单线程的 Redis 也能获得高性能，跟多路复用的 IO 模型密切相关，因为这避免了 accept() 和 send()/recv() 潜在的网络 IO 操作阻塞点。

**单线程 Redis 为什么那么快？**

（1）redis的操作大部分在内存上完成，再加上采用高效的数据结构（hash表，跳表）

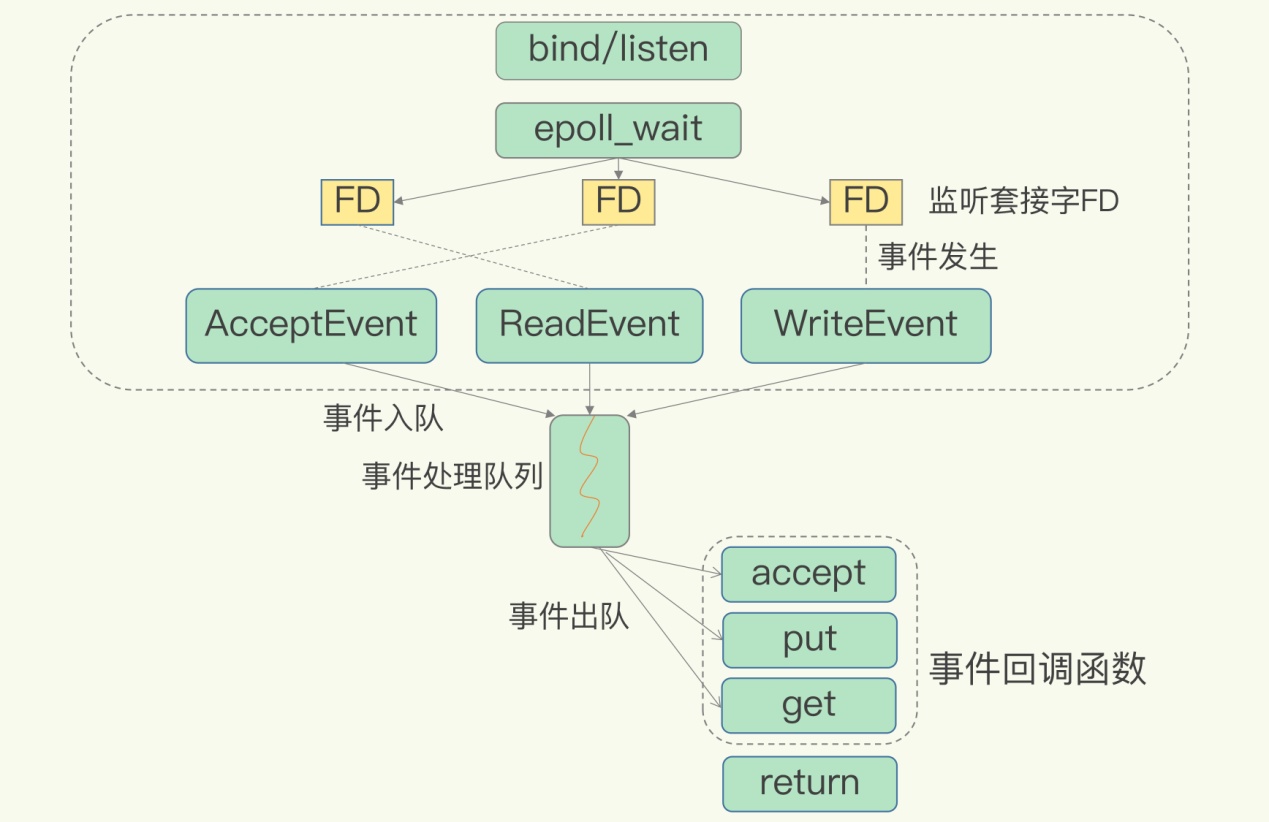
（2）redis 采用多路复用机制，使其在网络操作中能并发处理大量的客户端请求，实现高吞吐率。

**基于多路复用的I/O模型**

Linux中的IO 多路复用机制是指一个线程处理多个IO流，就是我们常听到select/epoll机制，简单的说：在redis运行单线程的情况下，该机制允许内核中，同时存在多个监听socket和已连接socket，kernel会一直监听这些socket上的连接请求或者数据请求，

一旦有请求到达就交给redis处理，这样就实现了一个redis线程处理多个IO流的效果。

下图多路复用的Io模型



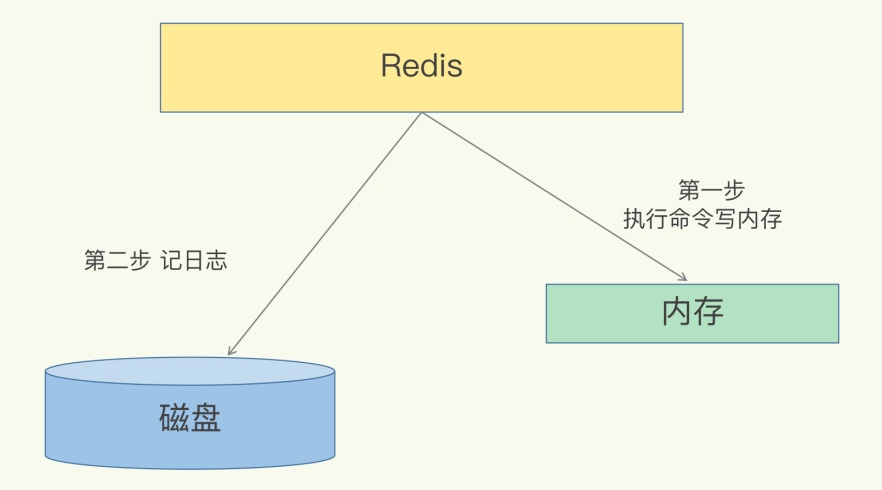
**图中的fd就是如上说述的多个socket，redis网络框架调用epoll机制，让内核监听这些socket，此时redis线程不会阻塞在某个特定监听的socket或者已连接的socket上，也就是说不会阻塞在某个特定的客户端请求处理上。正如此，redis可以同时和多个客户端连接并处理请求，从而提升并发性。**

为了请求到达时通知到redis线程，select/epoll机制提供了基于事件的回调机制，针对不同事件的发生，调用相应的处理函数。

**Redis数据的持久化**

Redis的持久化机制主要有两大机制，AOF日志和RDB快照；

**AOF日志：他是写后日志，redis先执行命令，把数据写入内存，然后才记录日志，如下图所示：**

****

**AOF为什么先执行命令然后写日志呢？**

AOF记录的是收到的每条指令，用文本记录，避免出现记录错误命令的情况出现，另外不会阻挡当期的写操作。

**三种会写策略：**

**AOF给我们提供了三个选择，也就是AOF的配置项appendfsync的三个可选值：**

Always，同步写回，每个写命令执行完，立马同步的把日志写会磁盘。

Everysec，每秒写回：每个写命令执行完，只是先把日志写到AOF文件的内存缓冲区，每隔一秒把缓冲区的内容写入磁盘。

No，操作系统控制的写回，每个写命令执行完，只是先把日志文件写到AOF

文件的内存缓冲区，每隔一秒把缓冲区的内容写入磁盘。

**三种策略的回写时机图表：**

**日志文件太大了怎么办？**

AOF的重写机制，直接根据数据库里的数据最新状态，生成这些数据的插入命令，作为新的日志，这个过程通过后台线程完成。

**Reids 宕机后快速恢复的手段2：**

内存快照：就是内存中的数据在某一时刻状态的记录，类似照片，对于redis就是把某一时刻的状态以文件的形式写到磁盘中，也就是快照，这样即使宕机快照文件也不会丢失，数据的可靠性也得到了保证。这个快照文件就称作RDB文件，Rdb就是redis database的缩写。

**给哪些内存数据做快照？**

全量快照。

**Redis提供了两个命令来生成RDB文件，分别是save和bgsave**

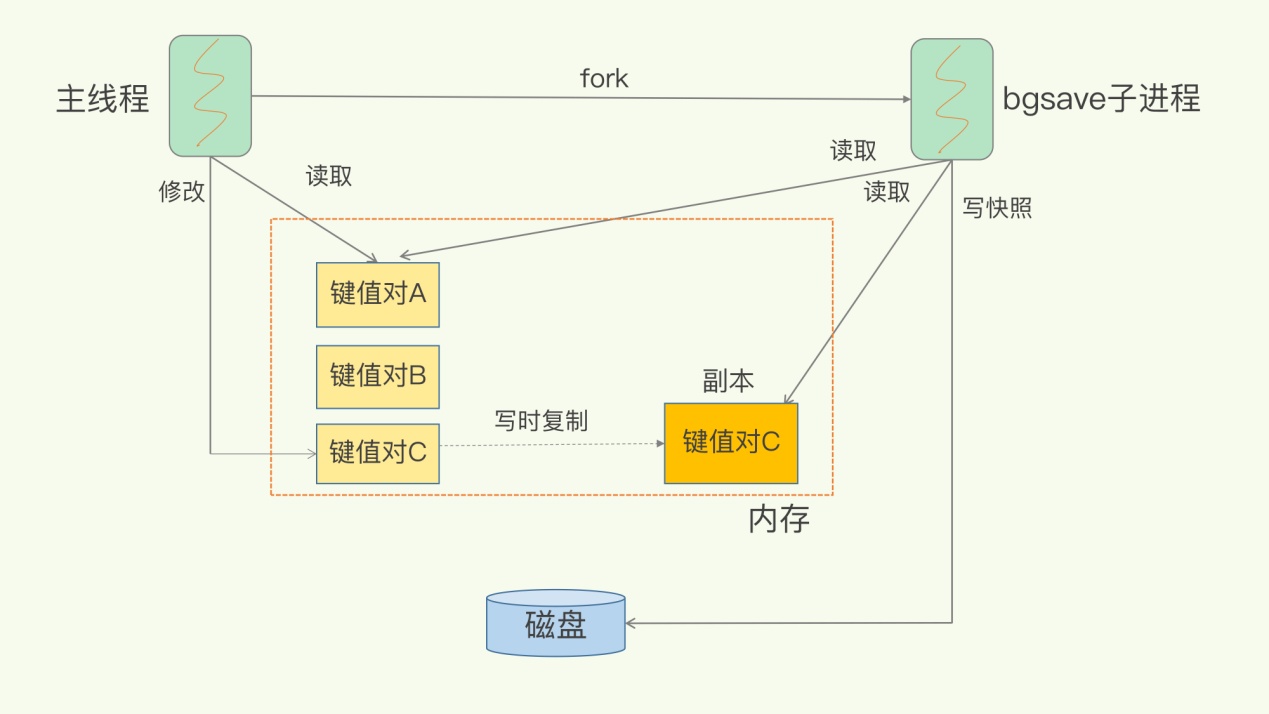
Save在主线程上执行，会导致阻塞。

Bgsave：创建个子进程，专门用于写入RDB文件，避免了主线程的阻塞，这也是redis rdb文件的默认生产模式。

**快照时候数据还可以修改吗？**

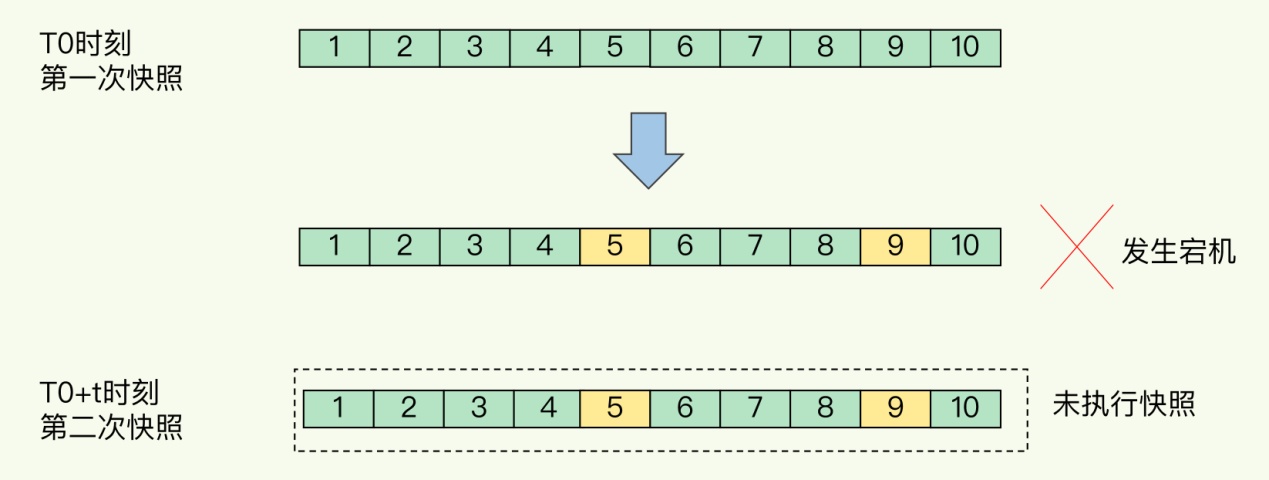
不可以修改,为了快照而暂停写操作肯定是无法接受的，这时redis会借助操作系统提供的写时复制技术，（copy on write, COW ）,在执行快照的同时，也可以正常处理写操作。

简单来说，bgsave 子进程是由主线程 fork 生成的，可以共享主线程的所有内存数据。bgsave 子进程运行后，开始读取主线程的内存数据，并把它们写入 RDB 文件。此时，如果主线程对这些数据也都是读操作（例如图中的键值对 A），那么，主线程和 bgsave 子进程相互不影响。但是，如果主线程要修改一块数据（例如图中的键值对 C），那么，这块数据就会被复制一份，生成该数据的副本。然后，bgsave 子进程会把这个副本数据写入 RDB 文件，而在这个过程中，主线程仍然可以直接修改原来的数据。



**可以每秒做一次快照吗？**

如下图所示，我们先在 T0 时刻做了一次快照，然后又在 T0+t 时刻做了一次快照，在这期间，数据块 5 和 9 被修改了。如果在 t 这段时间内，机器宕机了，那么，只能按照 T0 时刻的快照进行恢复。此时，数据块 5 和 9 的修改值因为没有快照记录，就无法恢复了

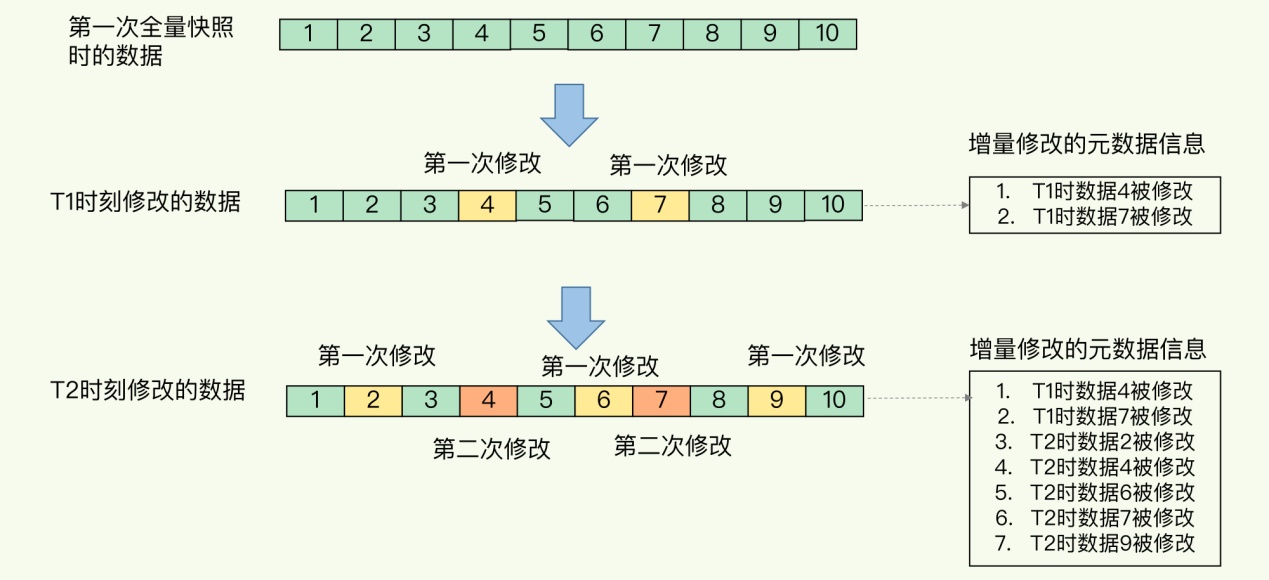


**如果频繁的执行全量快照，也会带来两方面的开销。**

（1）会导致主线程的阻塞

（2）会给磁盘带来压力

**那么增量修改可行吗？**

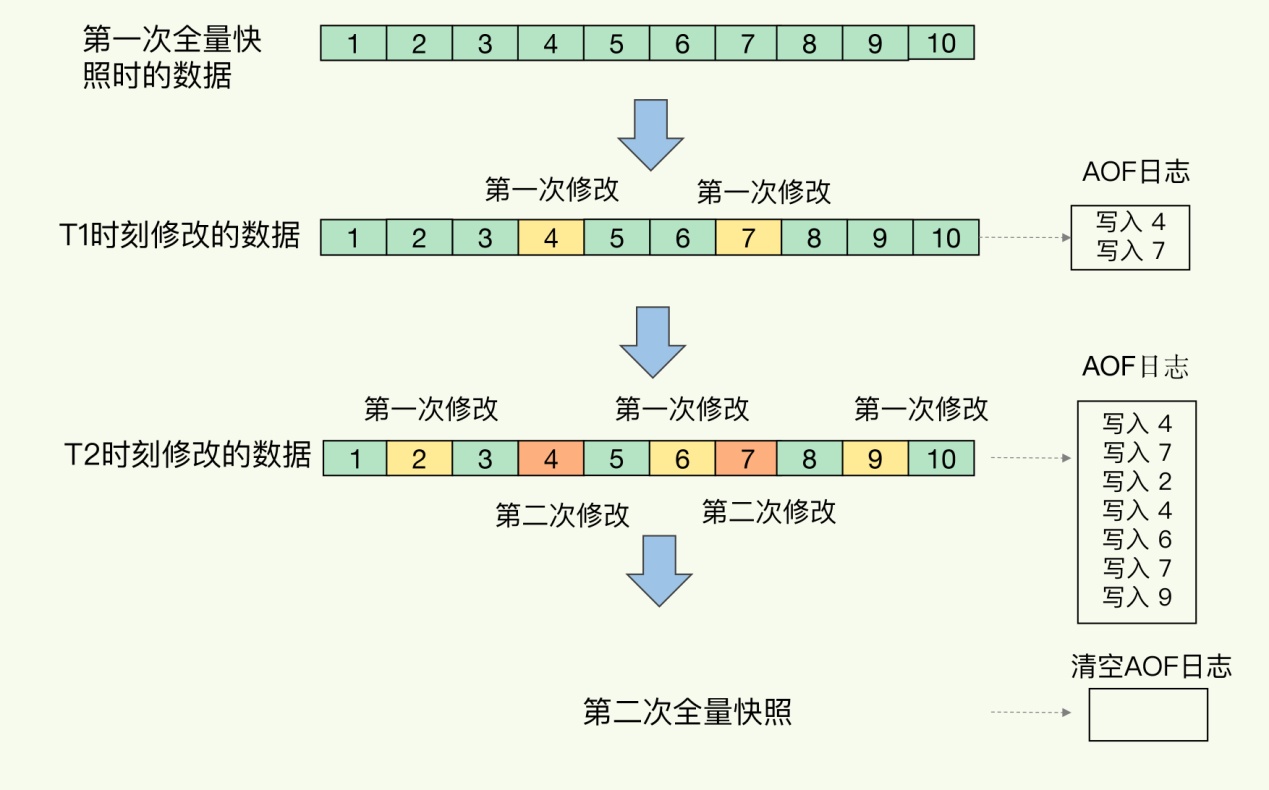


虽然跟 AOF 相比，快照的恢复速度快，但是，快照的频率不好把握，如果频率太低，两次快照间一旦宕机，就可能有比较多的数据丢失。如果频率太高，又会产生额外开销，那么，还有什么方法既能利用 RDB 的快速恢复，又能以较小的开销做到尽量少丢数据呢？

**混合使用 AOF 日志和内存快照：**

内存快照以一定的频率执行，在两次快照之间，使用 AOF 日志记录这期间的所有命令操作。

如下图所示，T1 和 T2 时刻的修改，用 AOF 日志记录，等到第二次做全量快照时，就可以清空 AOF 日志，因为此时的修改都已经记录到快照中了，恢复时就不再用日志了



这个方法既能享受到 RDB 文件快速恢复的好处，又能享受到 AOF 只记录操作命令的简单优势，颇有点“鱼和熊掌可以兼得”的感觉，建议你在实践中用起来。

**AOF 和 RDB 的选择问题：**

1. 数据不能丢失时，内存快照和 AOF 的混合使用是一个很好的选择；
2. 如果允许分钟级别的数据丢失，可以只使用 RDB；
3. 如果只用 AOF，优先使用 everysec 的配置选项，因为它在可靠性和性能之间取了一个平衡。

**Redis的高可靠性：**

**数据尽量少丢失，**

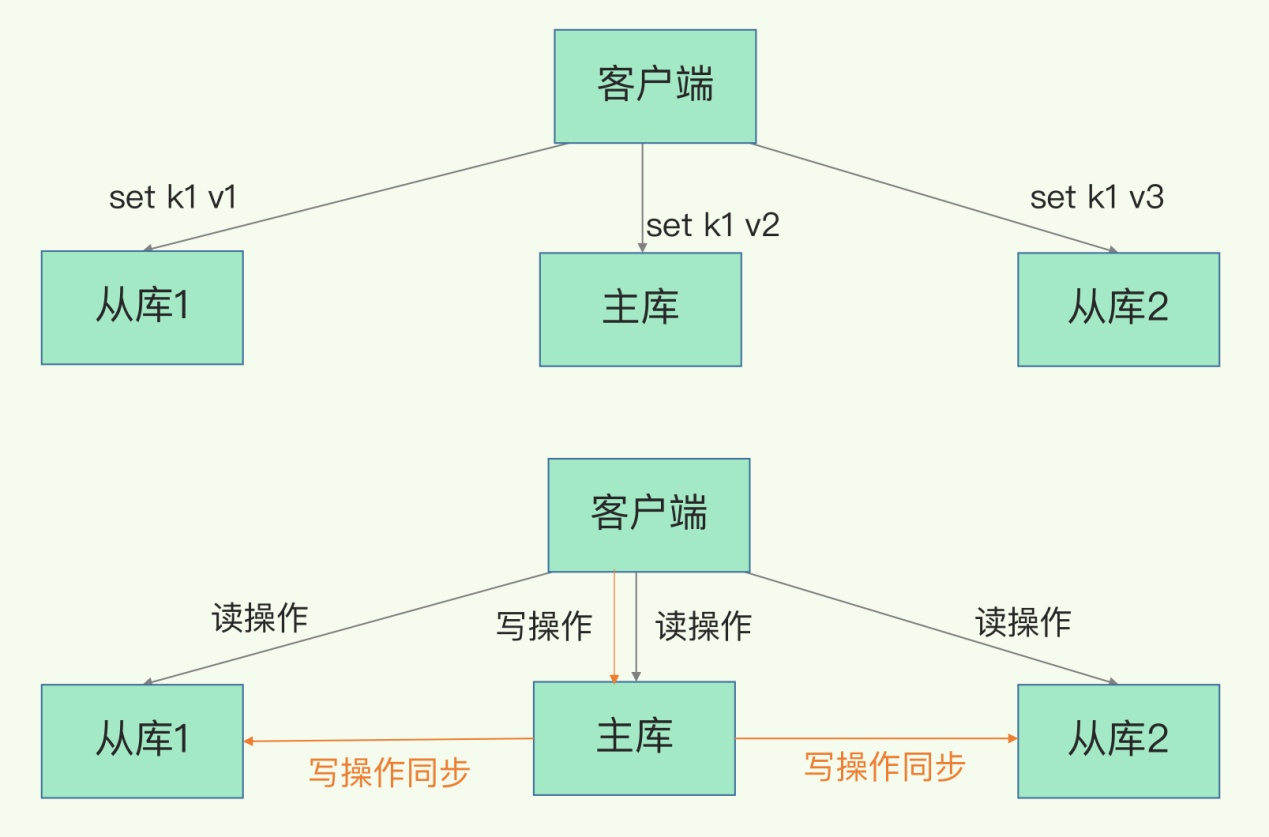
**服务尽量少中断。**

**Redis的做法就是增加副本的冗余量：**

将一份数据同时保存在多个实例上。即使有一个实例出现了故障，需要过一段时间才能恢复，其他实例也可以对外提供服务，不会影响业务使用

Redis 提供了主从库模式，以保证数据副本的一致，主从库之间采用的是读写分离的方式。**读操作**：主库、从库都可以接收；

**写操作：**首先到主库执行，然后，主库将写操作同步给从库。

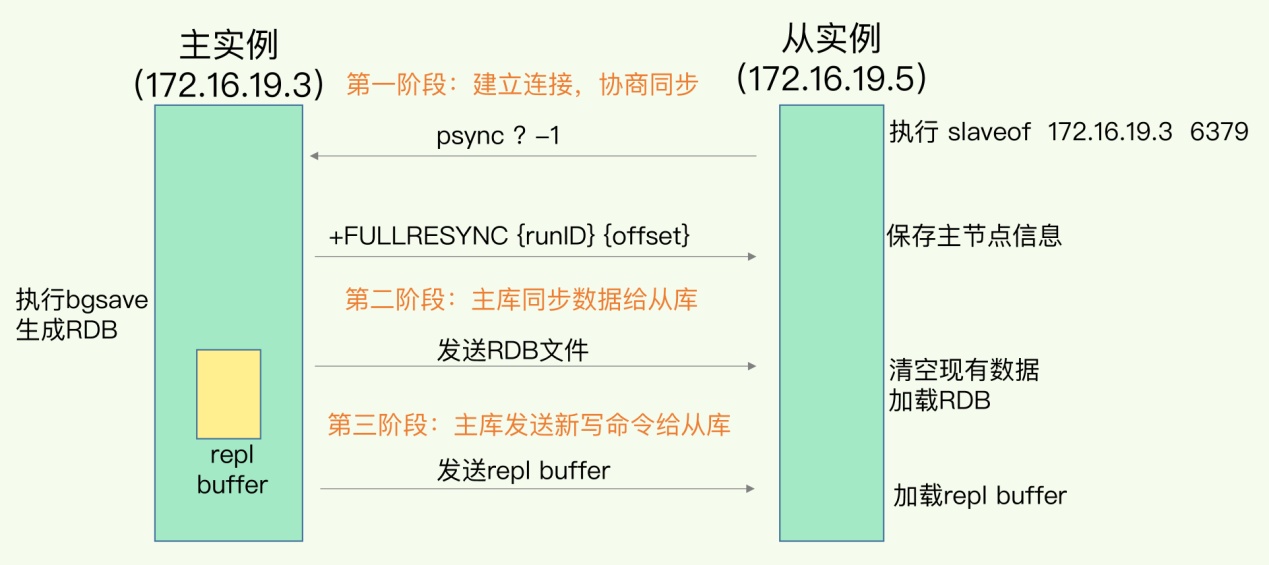


**为什么要采用读写分离的方式呢？**

主从库模式一旦采用了读写分离，所有数据的修改只会在主库上进行，不用协调三个实例。主库有了最新的数据后，会同步给从库，这样，主从库的数据就是一致的。

**主从库如何进行第一次同步？**

例如，现在有实例 1（ip：172.16.19.3）和实例 2（ip：172.16.19.5），我们在实例 2 上执行以下这个命令后，实例 2 就变成了实例 1 的从库，并从实例 1 上复制数据：



第一阶段是主从库间建立连接、协商同步的过程，主要是为全量复制做准备。在这一步，从库和主库建立起连接，并告诉主库即将进行同步，主库确认回复后，主从库间就可以开始同步了。

从库给主库发送 psync 命令，表示要进行数据同步，主库根据这个命令的参数来启动复制。psync 命令包含了主库的 runID 和复制进度 offset 两个参数。

runID，是每个 Redis 实例启动时都会自动生成的一个随机 ID，用来唯一标记这个实例。当从库和主库第一次复制时，因为不知道主库的 runID，所以将 runID 设为“？”。offset，此时设为 -1，表示第一次复制。

offset，此时设为 -1，表示第一次复制

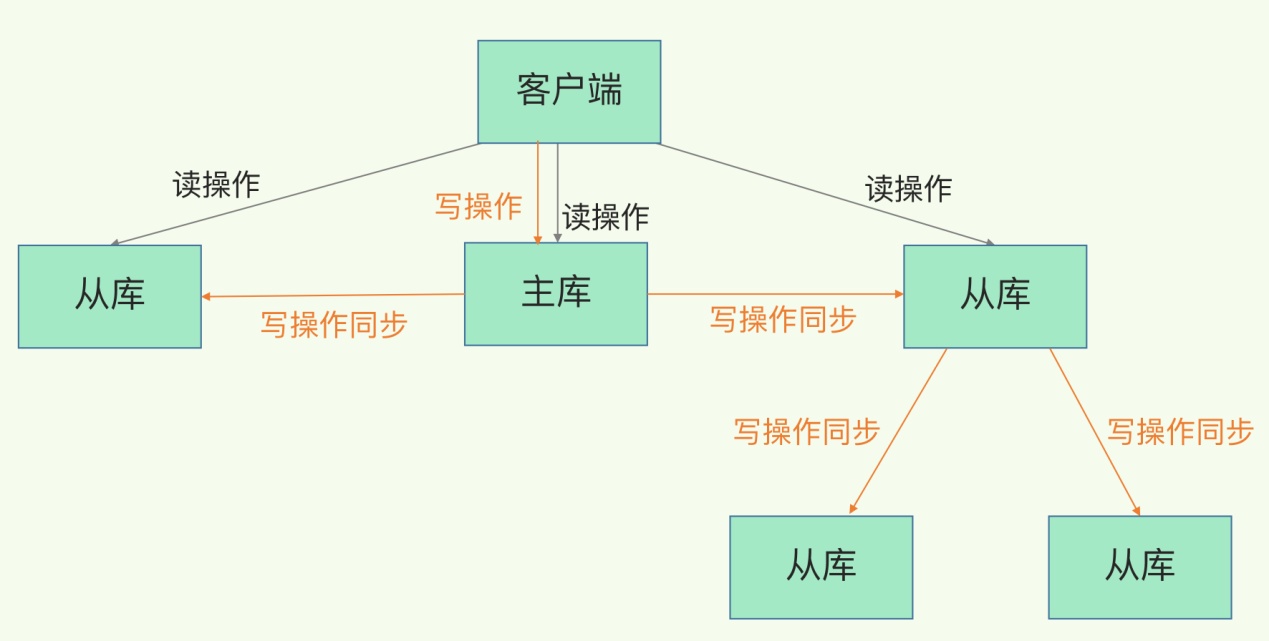
FULLRESYNC 响应表示第一次复制采用的全量复制，也就是说，主库会把当前所有的数据都复制给从库。在第二阶段，主库将所有数据同步给从库。从库收到数据后，在本地完成数据加载。这个过程依赖于内存快照生成的 RDB 文件。

**主从级联模式分担全量复制时的主库压力**

一次全量复制中，对于主库来说，需要完成两个耗时的操作：生成 RDB 文件和传输 RDB 文件。

通过“主 - 从 - 从”模式将主库生成 RDB 和传输 RDB 的压力，以级联的方式分散到从库上。

我们在部署主从集群的时候，可以手动选择一个从库（比如选择内存资源配置较高的从库），用于级联其他的从库。然后，我们可以再选择一些从库（例如三分之一的从库），在这些从库上执行如下命令，让它们和刚才所选的从库，建立起主从关

 **了 Redis 的主从库同步的基本原理，总结来说，有三种模式：全量复制、基于长连接的命令传播，以及增量复制。**

一个 Redis 实例的数据库不要太大，一个实例大小在几 GB 级别比较合适，这样可以减少 RDB 文件生成、传输和重新加载的开销。另外，为了避免多个从库同时和主库进行全量复制，给主库过大的同步压力，我们也可以采用“主 - 从 - 从”这一级联模式，来缓解主库的压力。

我特别建议你留意一下 **repl\_backlog\_size** 这个配置参数。如果它配置得过小，在增量复制阶段，可能会导致从库的复制进度赶不上主库，进而导致从库重新进行全量复制。所以，通过调大这个参数，可以减少从库在网络断连时全量复制的风险。

**题目：**

主从库间的数据复制同步使用的是 RDB 文件，前面我们学习过，AOF 记录的操作命令更全，相比于 RDB 丢失的数据更少。那么，为什么主从库间的复制不使用 AOF 呢？