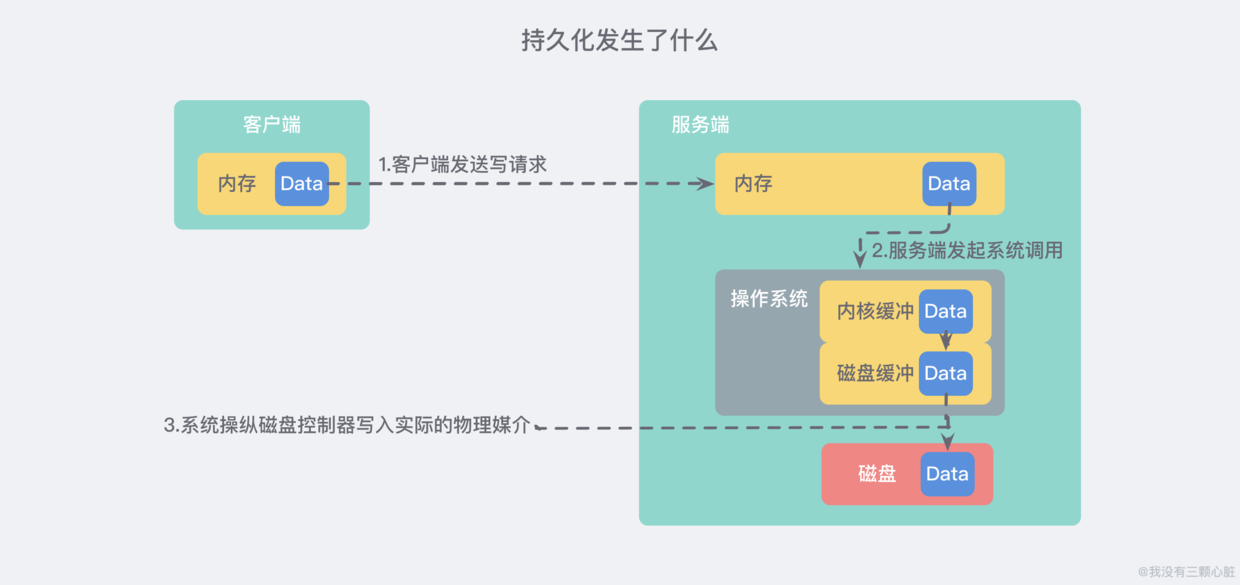
一、持久化简介

通常来说，从客户端发起请求开始，到服务器真实写入磁盘，需要发生如下几件事情：

1. 客户端向数据库发送写命令（数据在客户端的内存中）
2. 数据库接收到客户端的写请求（数据在服务器的内存中）
3. 数据库调用系统API将数据写入磁盘（数据库在内核缓冲区中）
4. 操作系统将写缓冲区传输到磁盘控制器（数据在磁盘缓存中）
5. 操作系统的磁盘控制器将数据写入实际的物理媒介中（数据在磁盘中）

上面的过程是精简的，实际的操作系统中，缓存和缓冲区会比这多得多。



二、如何尽可能保证持久化的安全

如果我们故障仅仅涉及到软件层面(该进程被管理员终止或程序崩溃)，并且没有接触到内核，那么在上述步骤3成功返回之后，我们就认为成功了。即使进程崩溃，操作系统仍然会帮助我们把数据正确地写入磁盘。

如果我们考虑停电/火灾等更具灾难性的事情，那么只有在完成了第5步之后，才是安全的。

所以我们可以总结得出数据安全最重要的阶段是：步骤三、四、五，即：

* 数据库软件调用写操作将用户空间的缓冲区转移到内核缓冲区的频率是多少？
* 内核多久从缓冲区取数据刷新到磁盘控制器？
* 磁盘控制器多久把数据写入物理媒介一次？
* 注意： 如果真的发生灾难性的事件，我们可以从上图的过程中看到，任何一步都可能被意外打断丢失，所以只能 尽可能地保证 数据的安全，这对于所有数据库来说都是一样的。

从第三步开始。Linux系统提供了清晰、易用的用于操作文件的POSIX file API。

int open(const char \*path, int oflag, .../\*,mode\_t mode \*/);

int close (int filedes);int remove( const char \*fname );

ssize\_t write(int fildes, const void \*buf, size\_t nbyte);

ssize\_t read(int fildes, void \*buf, size\_t nbyte);

所以，我们有很好的可用的API来完成第三步，但是对于成功返回之前，我们对系统调用花费的时间没有太多的控制权。

然后我们来说说第四步。我们知道，除了早期对电脑特别了解那帮人(操作系统就这帮人搞的)，实际的物理硬件都不是我们能够直接操作的，都是通过操作系统调用来达到目的的。为了防止过慢的I/O操作拖慢整个系统的运行，操作系统层面做了很多的努力，譬如说上述第四步提到的写缓冲区，并不是所有的写操作都会被立即写入磁盘，而是要先经过一个缓冲区，默认情况下，Linux将在30秒后实际提交写入。

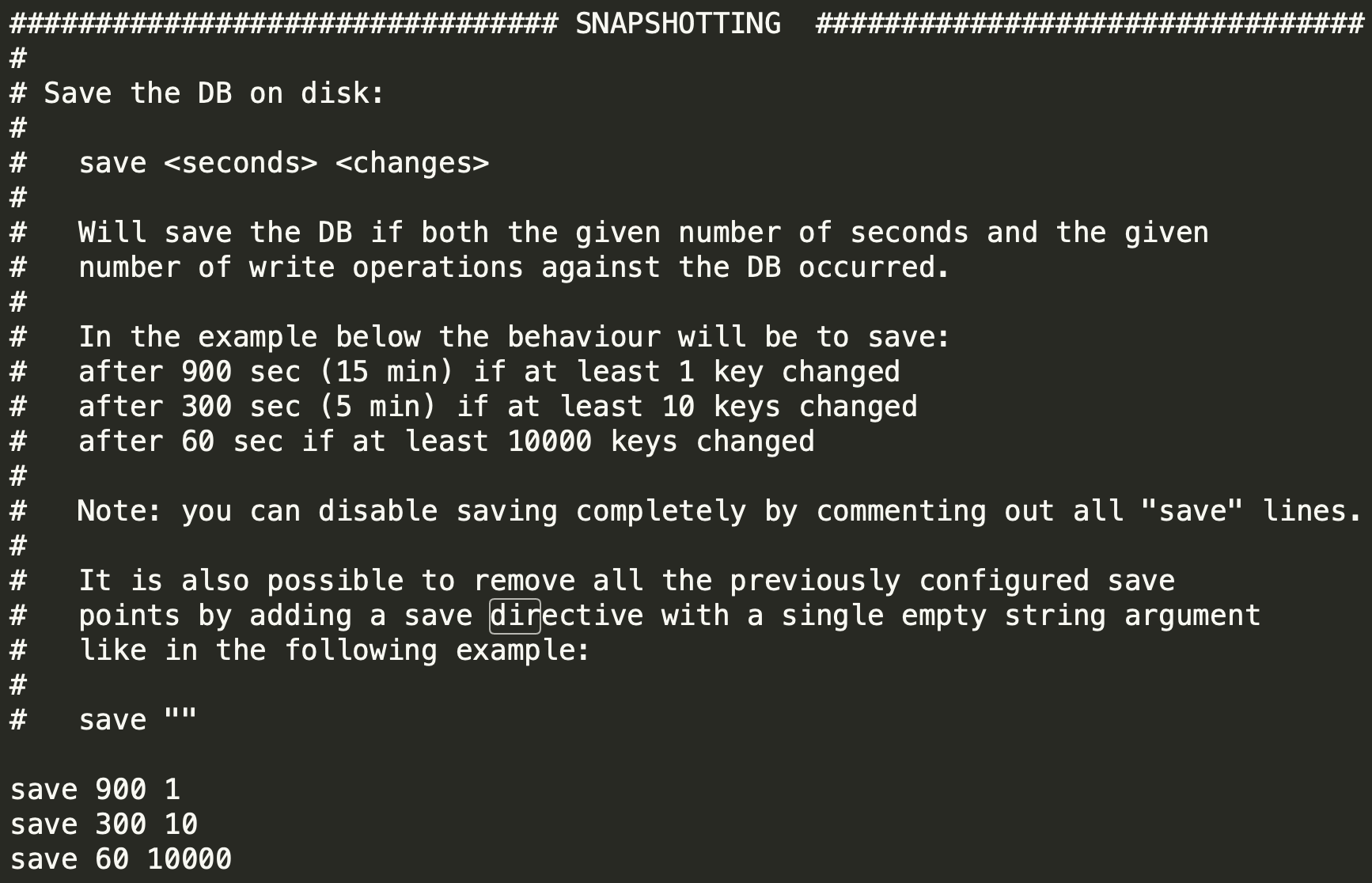
但是很明显，30秒并不是Redis能够承受的，这意味着，如果发生故障，那么最近30秒内写入的所有数据都可能会丢失。幸好PROSIX API提供了另一个解决方案：fsync，该命令会强制内核将缓冲区写入磁盘，但这是一个非常消耗性能的操作，每次调用都会阻塞等待直到设备报告IO完成，所以一般在生产环境的服务器中，Redis通常是每隔1s左右执行一次fsync操作。

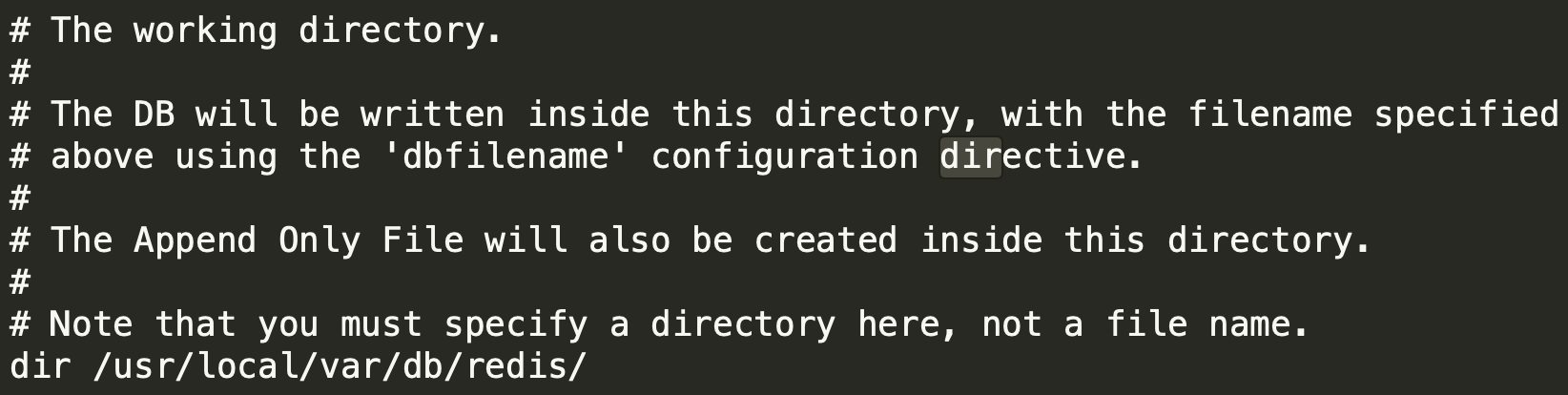
到目前为止，我们了解到了如何控制第三步和第四步，但是对于第五步，我们 完全无法控制。也许一些内核实现将试图告诉驱动实际提交物理介质上的数据，或者控制器可能会为了提高速度而重新排序写操作，不会尽快将数据真正写到磁盘上，而是会等待几个多毫秒。这完全是我们无法控制的。

三、Redis中的两种持久化方式

#### 1、快照RDB

Redis快照是最简单的Redis持久性模式。当满足特定条件时，它将生成数据集的时间点快照，例如，如果先前的快照是在2分钟前创建的，并且现在已经至少有100次新写入，则将创建一个新的快照。Redis实例来控制，也可以在运行时修改而无需重新启动服务器。快照作为包含整个数据集的单个.rdb文件生成。

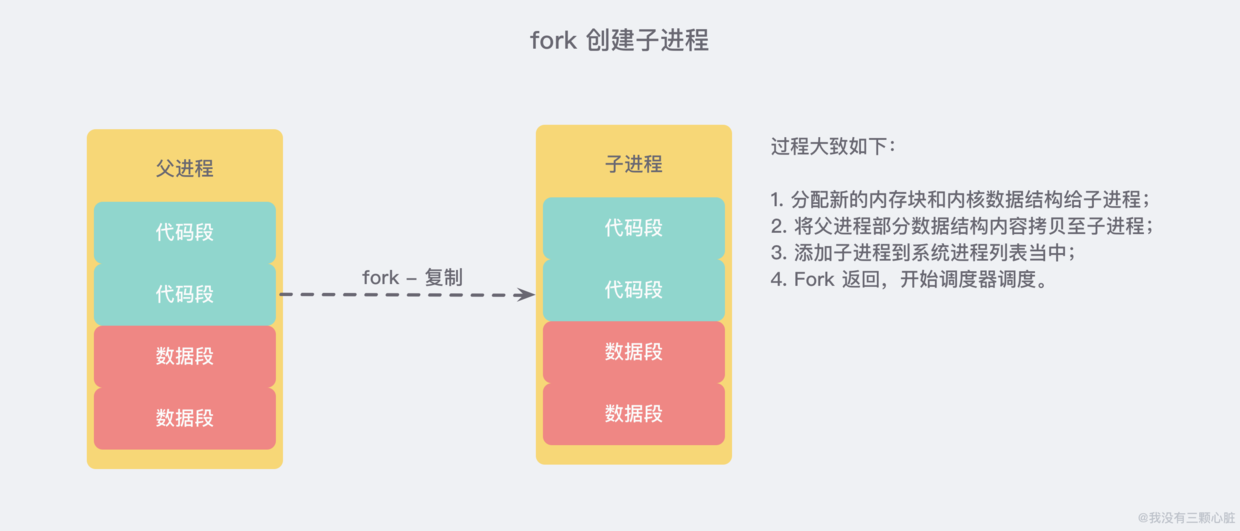




但我们知道，Redis是一个单线程的程序，这意味着，我们不仅仅要响应用户的请求，还需要进行内存快照。而后者要求Redis必须进行IO操作，这会严重拖累服务器的性能。

还有一个重要的问题是，我们在持久化的同时，内存数据结构还可能在变化，比如一个大型的hash字典正在持久化，结果一个请求过来把它删除了，可是这才刚持久化结束，咋办？

操作系统多进程COW(Copy On Write)机制拯救了我们。Redis在持久化时会调用glibc的函数fork产生一个子进程，简单理解也就是基于当前进程复制了一个进程，主进程和子进程会共享内存里面的代码块和数据段：



这里多说一点，为什么fork成功调用后会有两个返回值呢？因为子进程在复制时复制了父进程的堆栈段，所以两个进程都停留在了fork函数中(都在同一个地方往下继续"同时"执行)，等待返回，所以\*一次在父进程中返回子进程的 pid，另一次在子进程中返回零，系统资源不够时返回负数\*\*。\*(伪代码如下)

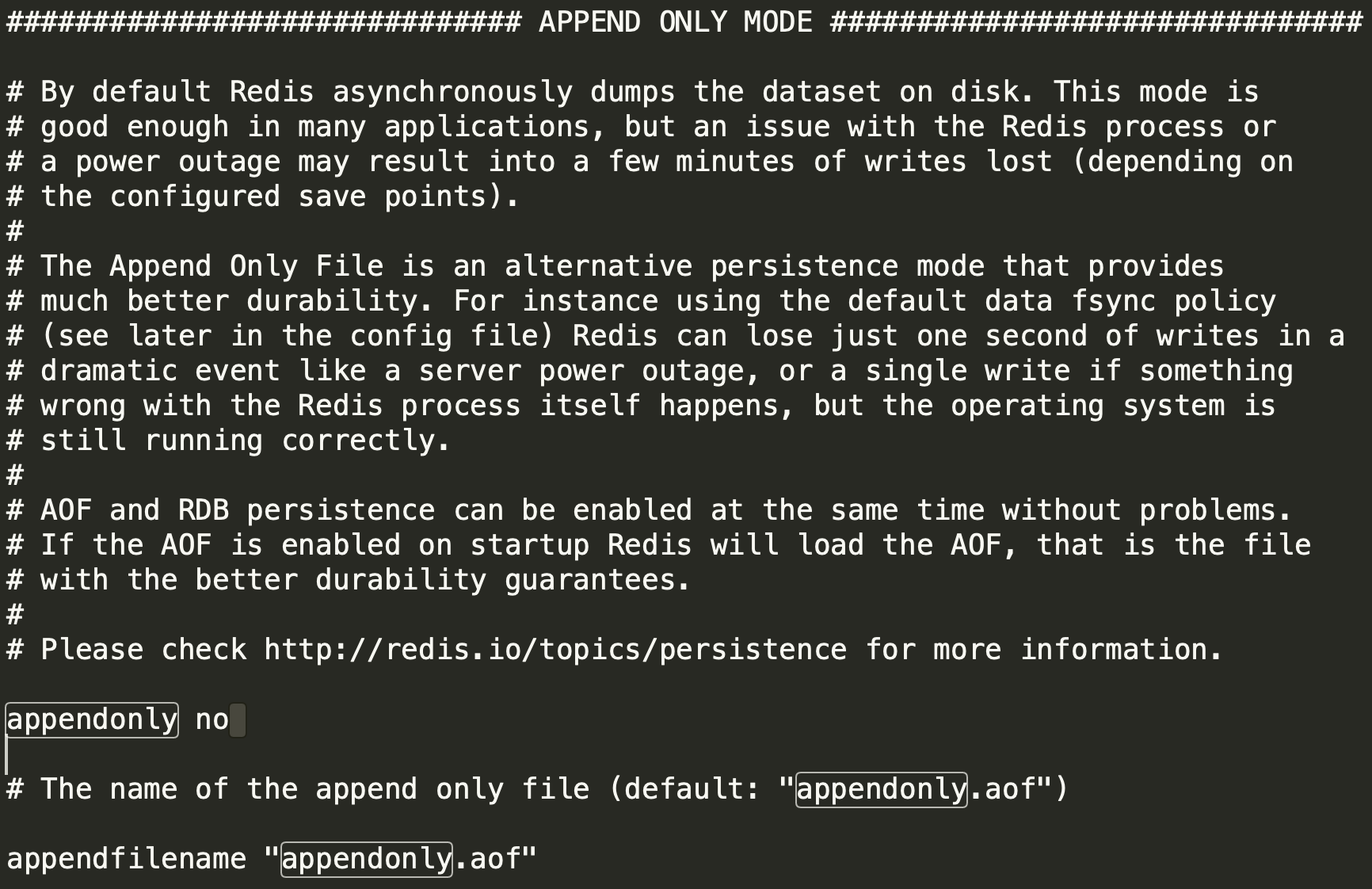
所以快照持久化可以完全交给子进程来处理，父进程则继续处理客户端请求。子进程做数据持久化，它不会修改现有的内存数据结构，它只是对数据结构进行遍历读取，然后序列化写到磁盘中。但是父进程不一样，它必须持续服务客户端请求，然后对内存数据结构进行不间断的修改。

这个时候就会使用操作系统的COW机制来进行数据段页面的分离。数据段是由很多操作系统的页面组合而成，当父进程对其中一个页面的数据进行修改时，会将被共享的页面复制一份分离出来，然后对这个复制的页面进行修改。这时 子进程相应的页面是没有变化的，还是进程产生时那一瞬间的数据。

子进程因为数据没有变化，它能看到的内存里的数据在进程产生的一瞬间就凝固了，再也不会改变，这也是为什么Redis的持久化叫「快照」的原因。接下来子进程就可以非常安心的遍历数据了进行序列化写磁盘了。

#### 2、仅追加文件AOF

快照不是很持久。如果运行Redis的计算机停止运行，电源线出现故障或者您 kill -9的实例意外发生，则写入Redis的最新数据将丢失。



AOF(Append Only File - 仅追加文件)它的工作方式非常简单：每次执行修改内存中数据集的写操作时，都会记录该操作。假设AOF日志记录了自Redis实例创建以来所有的修改性指令序列，那么就可以通过对一个空的Redis实例顺序执行所有的指令，来恢复Redis当前实例的内存数据结构的状态。

当Redis收到客户端修改指令后，会先进行参数校验、逻辑处理，如果没问题，就立即将该指令文本存储到AOF日志中，也就是说，先执行指令再将日志存盘。这一点不同于MySQL、LevelDB、HBase等存储引擎，如果我们先存储日志再做逻辑处理，这样就可以保证即使宕机了，我们仍然可以通过之前保存的日志恢复到之前的数据状态，但是Redis为什么没有这么做呢？

#### 3、AOF重写

Redis在长期运行的过程中，AOF的日志会越变越长。如果实例宕机重启，重放整个AOF日志会非常耗时，导致长时间Redis无法对外提供服务。所以需要对 AOF日志"瘦身"。

Redis提供了bgrewriteaof指令用于对AOF日志进行瘦身。其原理就是开辟一个子进程对内存进行遍历转换成一系列Redis的操作指令，序列化到一个新的 AOF日志文件中。序列化完毕后再将操作期间发生的增量AOF日志追加到这个新的AOF日志文件中，追加完毕后就立即替代旧的AOF日志文件了，瘦身工作就完成了。

fsync

AOF日志是以文件的形式存在的，当程序对AOF日志文件进行写操作时，实际上是将内容写到了内核为文件描述符分配的一个内存缓存中，然后内核会异步将脏数据刷回到磁盘的。

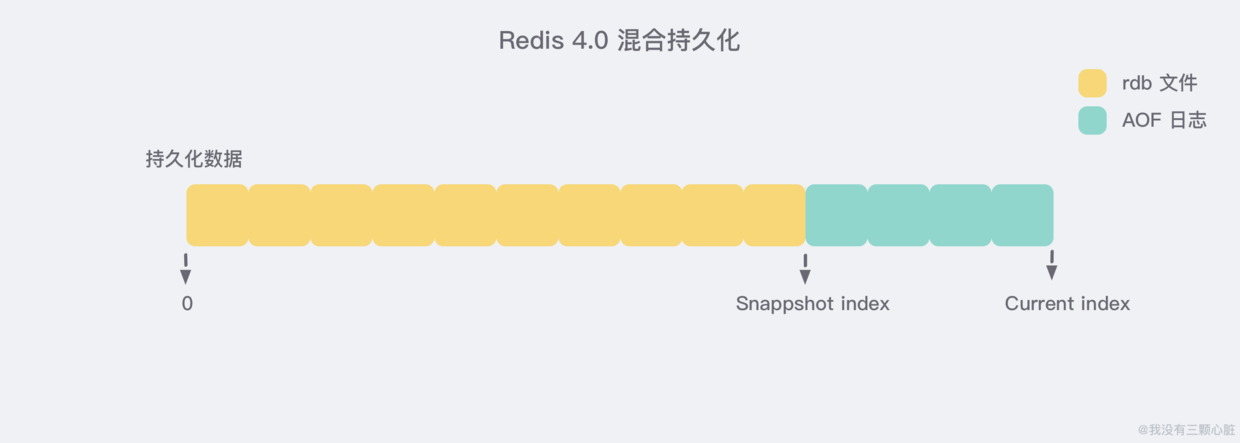
就像我们上方第四步描述的那样，我们需要借助glibc提供的fsync(int fd)函数来讲指定的文件内容强制从内核缓存刷到磁盘。但"强制开车"仍然是一个很消耗资源的一个过程，需要"节制"！通常来说，生产环境的服务器，Redis每隔1s左右执行一次fsync操作就可以了。

Redis同样也提供了另外两种策略，一个是永不fsync，来让操作系统来决定合适同步磁盘，很不安全，另一个是来一个指令就fsync一次，非常慢。但是在生产环境基本不会使用，了解一下即可。

四、Redis 4.0混合持久化

重启Redis时，通常使用AOF日志重放，但是重放AOF日志性能相对rdb来说要慢很多，这样在Redis实例很大的情况下，启动需要花费很长的时间。

Redis 4.0为了解决这个问题，带来了一个新的持久化选项——混合持久化。将rdb文件的内容和增量的AOF日志文件存在一起。这里的AOF日志不再是全量的日志，而是自持久化开始到持久化结束的这段时间发生的增量AOF日志，通常这部分AOF日志很小：



于是在Redis重启的时候，可以先加载rdb的内容，然后再重放增量AOF日志就可以完全替代之前的AOF全量文件重放，重启效率因此大幅得到提升。