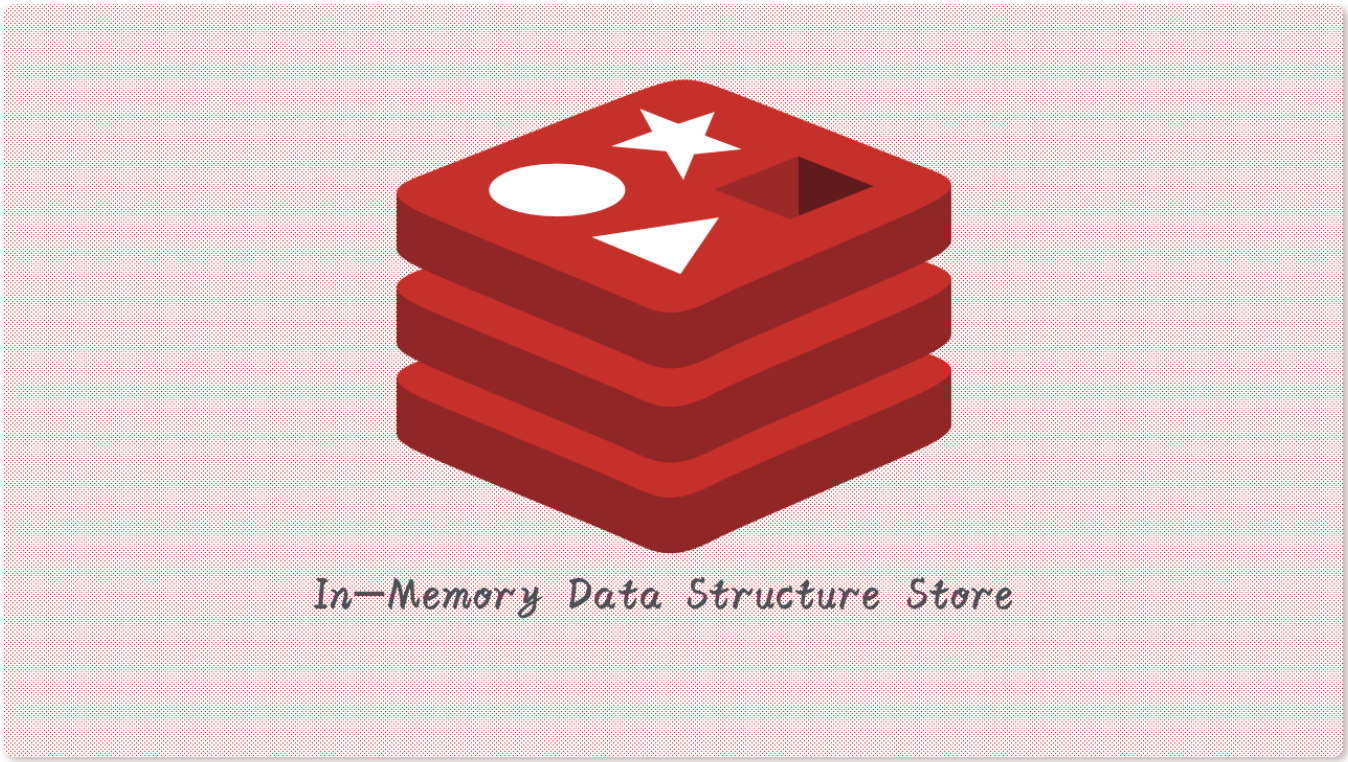
前言
1、AOF日志
1.1、概述
1.2、日志文件
1.3、写回策略
1.4、策略实现原理
1.5、重写机制
1.6、AOF 后台重写
1.6.1、介绍
1.6.2、实现原理
1.7、优缺点
2、RDB快照
2.1、概述
2.2、实现方式
2.3、实现原理
2.4、极端情况
2.5、优缺点
3、混合体实现
4、大Key问题
4.1、何为大key
4.2、负面影响
4.2.1、持久化影响
4.2.2、其他影响
4.2、如何避免大key
5、应用场景

前言

Redis是一个内存数据库，当机器重启之后内存中的数据都会丢失。众所周知，数据在很多情况下都是最最最重要的一部分，所以对Redis来说，持久化显得尤为重要。



1、AOF日志

1.1、概述

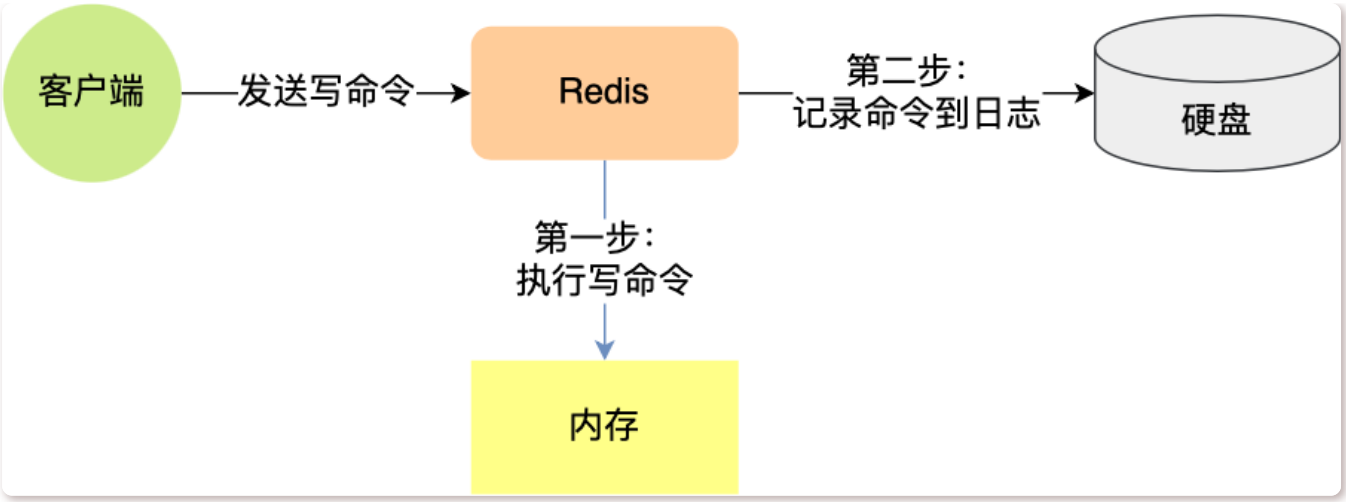
在Redis中提供了两种持久化的方式，分别是 *AOF日志* 和 *RDB快照* 。这两种技术都会用各用一个日志文件来记录信息，但是记录的内容是不同的。

- AOF 文件的内容是操作命令；
- RDB 文件的内容是二进制数据。

AOF持久化方式是**以日志的形式**记录每个写操作的指令，将操作追加到AOF文件的末尾。当Redis重新启动时，可以通过重新执行AOF文件中的指令来恢复数据。因此对于AOF持久化方式，最重要的便是AOF的日志文件。

1.2、日志文件

试想一下，如果 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，这不就相当于恢复了缓存数据了吗?



种保存写操作命令到日志的持久化方式，就是 Redis 里的 **AOF(Append Only File)** 持久化功能。**但值得注意只会记录写操作命令，读操作命令是不会被记录的**，因为没意义。

在Redis中AOF持久化功能**默认是不开启**的，需要我们修改 `redis.conf` 配置文件中的以下参数：

```
// 标识是否开启AOF持久化，默认no为关闭
appendonly  yes
// AOF持久化文件的名称
appendfilename  "appendonly.aof"
```

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。但aof日志文件有着自己的一套规范，举个例子，执行 `set name xbaozi` **完成后**，在aof文件中存放的记录如下：

```
*3
$3
set
$4
name
$6
xbaozi
```

「 `*3` 」表示当前命令有三个部分，每部分都是以「 `$+数字` 」开头，后面紧跟着具体的命令、键或值。然后，这里的「 `数字` 」表示这部分中的命令、键或值一共有多少字节。例如，「 `$3 set` 」表示这部分有3个字节，也就是「 `set` 」命令这个字符串的长度。

不知道大家有没有注意到前面说到的写入AOF日志是在**命令完成后才写入**的，这有两个原因：

1. **避免额外的检查开销**。因为如果先将写操作命令记录到 AOF 日志里，再执行该命令的话，**如果当前的命令语法有问题**，那么如果不进行命令语法检查，该错误的命令记录到 AOF 日志里后，Redis **在使用日志恢复数据时，就可能会出错**。而如果先执行写操作命令再记录日志的话，只有在该命令执行成功后，才将命令记录到 AOF 日志里，这样就不用额外的检查开销，保证记录在 AOF 日志里的命令都是可执行并且正确的。
2. **不会阻塞当前写操作命令的执行**。因为当写操作命令执行成功后，才会将命令记录到 AOF 日志。

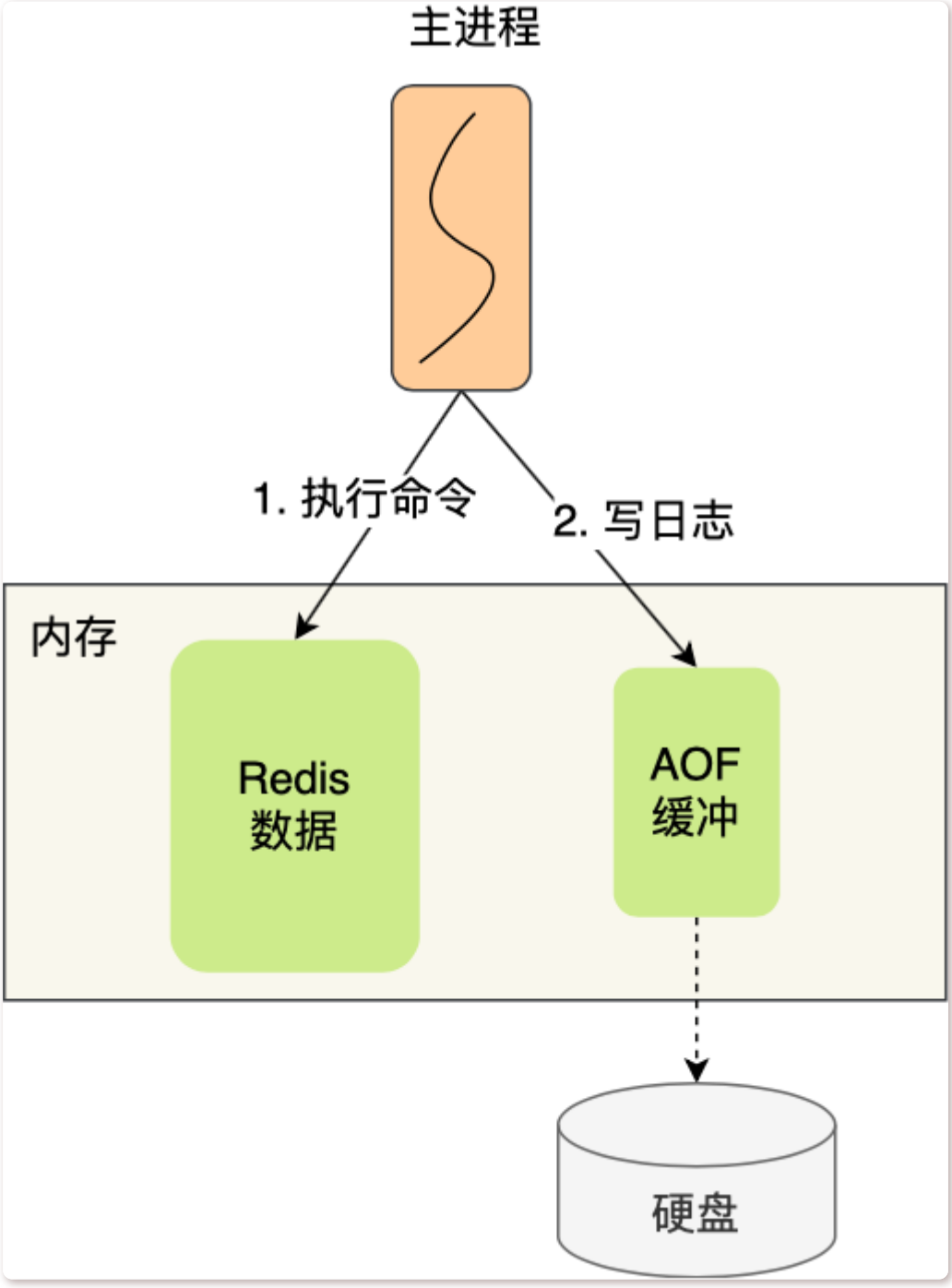
> 当然，任何事物都有两面性。

使用AOF实现持久化也是有着一定的风险的：

1. **数据丢失**。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**。
2. **阻塞服务**。由于写操作命令执行成功后才记录到 AOF 日志，所以不会阻塞当前写操作命令的执行，但是**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

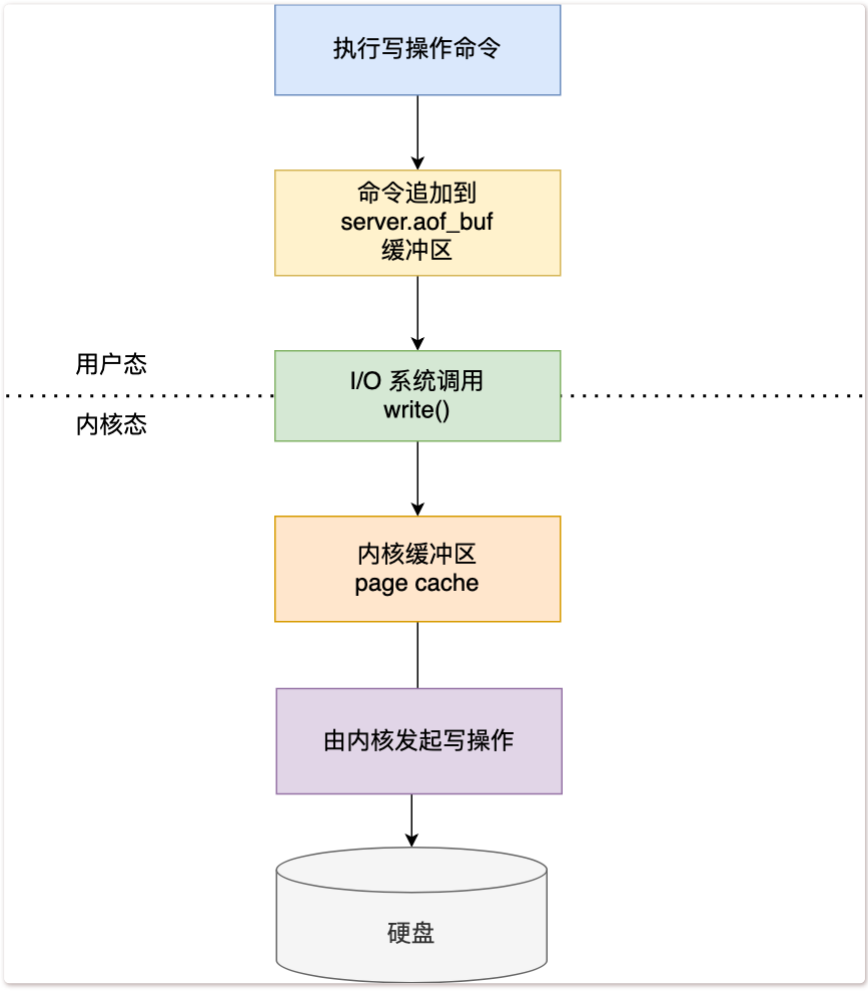
如果在将日志内容写入到硬盘时，服务器的硬盘的 I/O 压力太大，就会导致写硬盘的速度很慢，进而阻塞住了，也就会导致后续的命令无法执行。



1.3、写回策略

Redis 写入 AOF 日志的过程主要如下：

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**，控制的就是上面说的第三步的过程。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**。

这 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**：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；

- **Everysec**：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；
- **No**：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。

写回策略	安全性	性能	写回时机	优点	缺点	适用场景
always	高	低	每个写命令都立即写入	- 数据安全性高 - 持久化数据完整性好 - 故障发生时丢失的数据量较少	- 性能较低 - 每个写操作都需要同步写磁盘，影响性能	对数据安全性要求较高的场景
everysec	中	中	每秒写入一次	- 数据安全性较高 - 性能较好 - 故障发生时丢失的数据量较少	- 写回时机不是实时的，可能会丢失最近的写命令 - 在发生故障时可能会丢失最后一次写操作	对性能和数据安全性有平衡要求
no	低	高	依赖操作系统/文件系统刷盘	- 性能最佳 - 不进行同步写磁盘操作，减少磁盘写入次数 - 写操作不会阻塞	- 数据安全性较低 - 故障发生时丢失的数据量较多 - 可能会导致数据丢失或损坏	对性能要求较高的场景

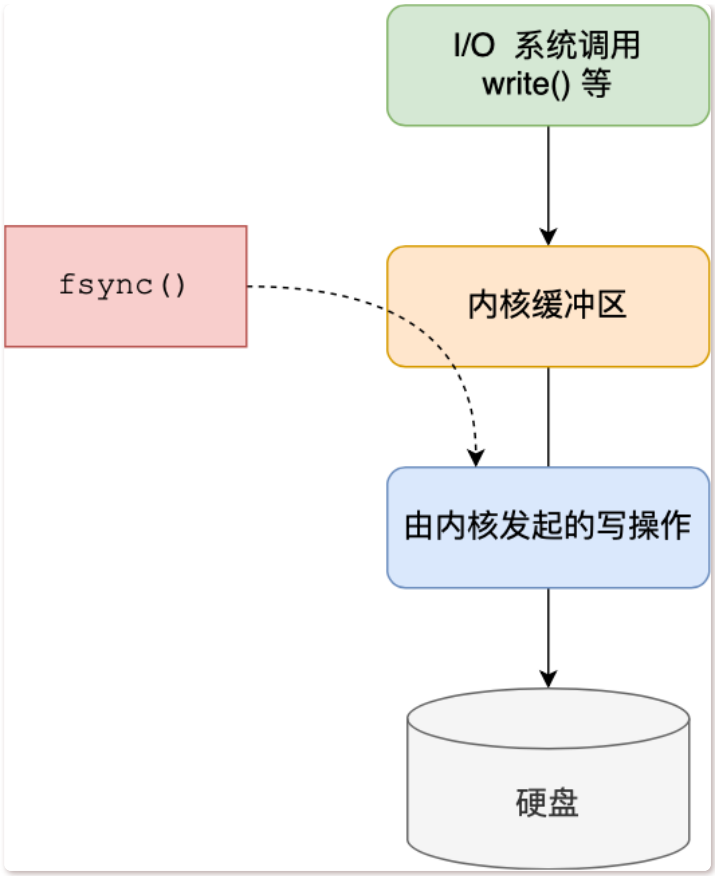
1.4、策略实现原理

深入到源码后就会发现这三种策略只是在控制 `fsync()` 函数的调用时机。

当应用程序向文件写入数据时，**内核通常先将数据复制到内核缓冲区中，然后排入队列，然后由内核决定何时写入硬盘。**

如果想要应用程序向文件写入数据后，能立马将数据同步到硬盘，就可以调用 `fsync()` 函数，这样内核就会将内核缓冲区的数据直接写入到硬盘，等到硬盘写操作完成后，该函数才会返回。

- Always 策略就是每次写入 AOF 文件数据后，就执行 fsync() 函数；
- Everysec 策略就会创建一个异步任务来执行 fsync() 函数；
- No 策略就是永不执行 fsync() 函数；



1.5、重写机制

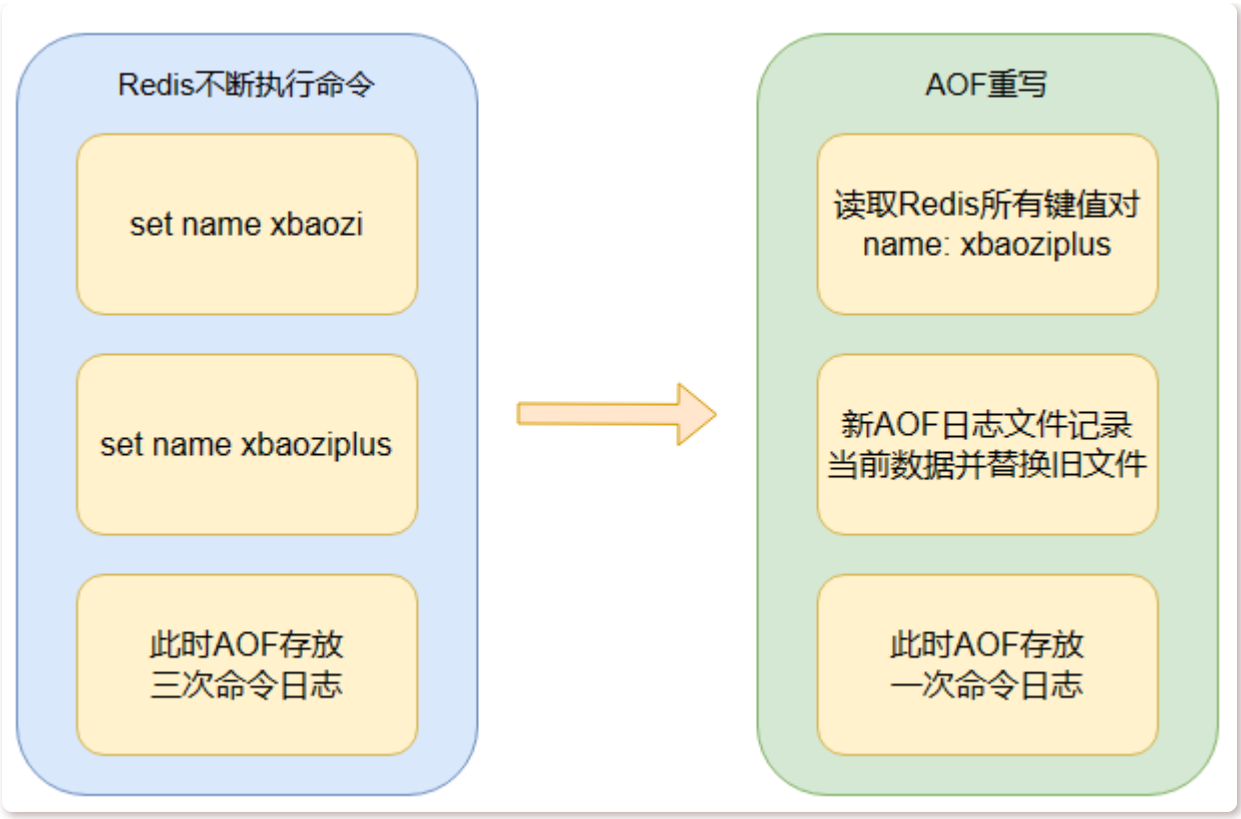
AOF 日志是一个文件，随着执行的写操作命令越来越多，文件的大小会越来越大。

如果当 AOF 日志文件过大就会带来性能问题，比如重启 Redis 后，需要读 AOF 文件的内容以恢复数据，如果文件过大，整个恢复的过程就会很慢。

所以，Redis 为了避免 AOF 文件越写越大，提供了 **AOF 重写机制**，当 AOF 文件的大小超过所设定的阈值后，Redis 就会启用 AOF 重写机制，来压缩 AOF 文件。

AOF 重写机制是在重写时，读取当前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。

举个例子，在没有使用重写机制前，假设前后执行了「*set name xbaozi*」和「*set name xbaoziplus*」这两个命令的话，就会将这两个命令记录到 AOF 文件。



但是**在使用重写机制后，就会读取 name 最新的 value（键值对），然后用一条「set name xbaoziplus」命令记录到新的 AOF 文件**，之前的第一个命令就没有必要记录了，因为它属于「历史」命令，没有作用了。这样一来，一个键值对在重写日志中只用一条命令就行了。

重写工作完成后，就会将新的 AOF 文件覆盖现有的 AOF 文件，这就相当于压缩了 AOF 文件，使得 AOF 文件体积变小了。然后，在通过 AOF 日志恢复数据时，只用执行这条命令，就可以直接完成这个键值对的写入了。

所以，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。最后在重写工作完成后，将新的 AOF 文件覆盖现有的 AOF 文件。

> 那么为什么不直接复用AOF文件进行修改，而是选择多开一个文件耗费空间去重写AOF呢?

因为**如果 AOF 重写过程中失败了，现有的 AOF 文件就会造成污染**，可能无法用于恢复使用。所以 AOF 重写过程，先重写到新的 AOF 文件，重写失败的话，就直接删除这个文件就好，不会对现有的 AOF 文件造成影响。

1.6、AOF 后台重写

1.6.1、介绍

写入 AOF 日志的操作虽然是在主进程完成的，因为它写入的内容不多，所以一般不太影响命令的操作。

但是在触发 AOF 重写时，比如当 AOF 文件大于 `64M` 时，就会对 AOF 文件进行重写，这时是需要**读取所有缓存的键值对数据**，并为每个键值对生成一条命令，然后将其写入到新的 AOF 文件，重写完后，就把现在的 AOF 文件替换掉。

这个过程其实是很耗时的，所以重写的操作不能放在主进程里。

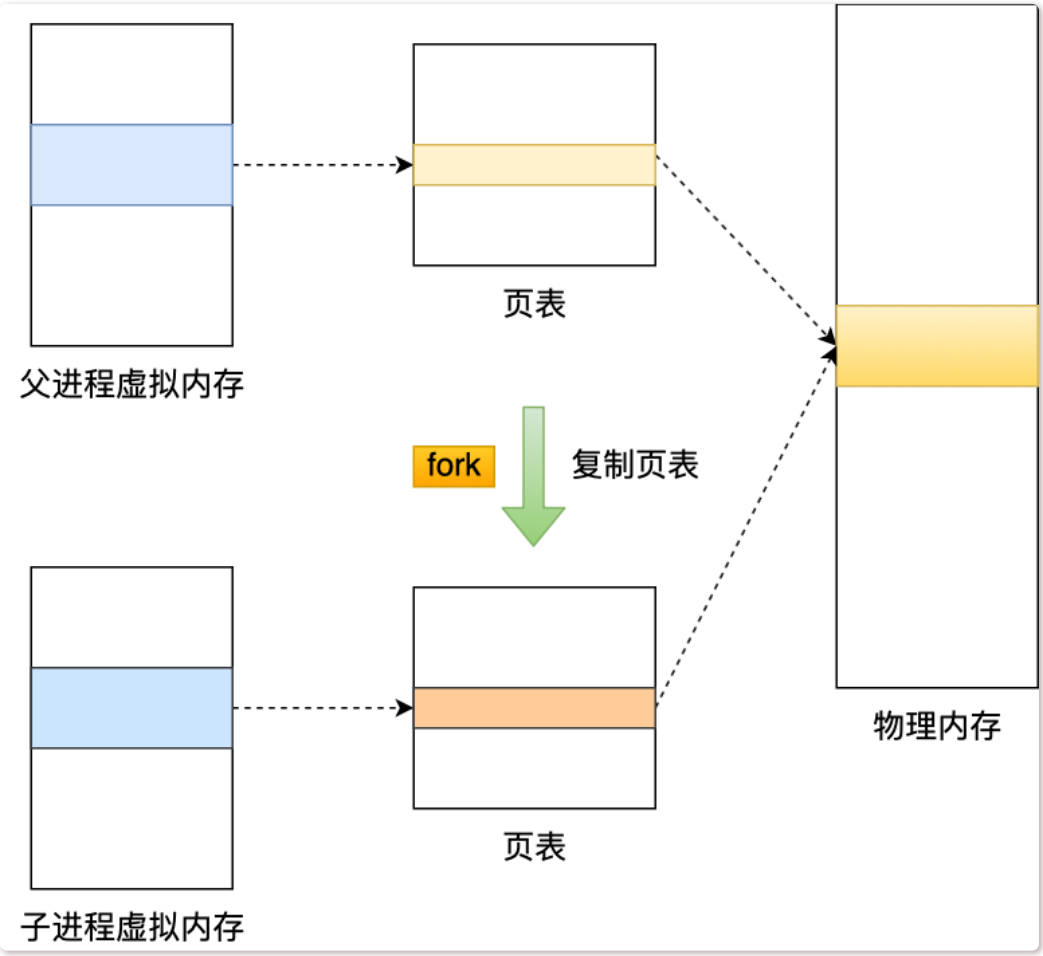
所以，Redis 的**重写 AOF 过程是由后台子进程 `bgrewriteaof` 来完成的**，这么做可以达到两个好处：

- 子进程进行 AOF 重写期间，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。

1.6.2、实现原理

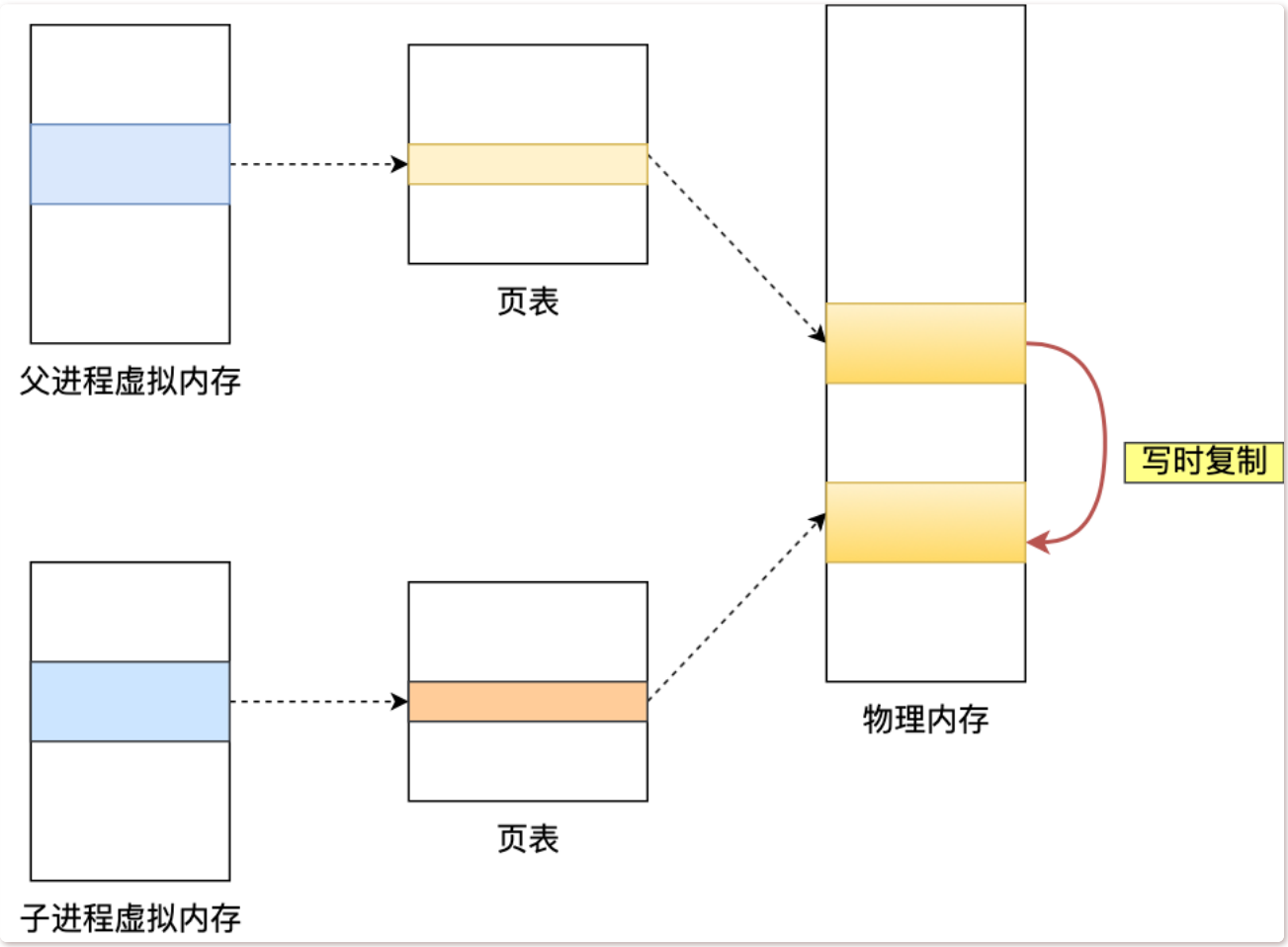
前面提及到子进程带有主进程的数据副本，那么是怎么拥有主进程一样的数据副本的呢？

主进程在通过 `fork` 系统调用生成 `bgrewriteaof` 子进程时，操作系统会把主进程的「**页表**」复制一份给子进程，这个页表记录着虚拟地址和物理地址映射关系，而不会复制物理内存，也就是说，**两者的虚拟空间不同，但其对应的物理空间是同一个**。



这样一来，子进程就共享了父进程的物理内存数据了，这样能够**节约物理内存资源**，页表对应的页表项的属性会标记该物理内存的权限为**只读**。

不过，当父进程或者子进程在向这个内存发起写操作时，CPU 就会触发**写保护中断**，这个写保护中断是由于违反权限导致的，然后操作系统会在「写保护中断处理函数」里进行**物理内存的复制**，并重新设置其内存映射关系，将父子进程的内存读写权限设置为**可读写**，最后才会对内存进行写操作，这个过程被称为「**写时复制(Copy On Write)**」。



写时复制顾名思义，**在发生写操作的时候，操作系统才会去复制物理内存**，这样是为了防止 `fork` 创建子进程时，由于物理内存数据的复制时间过长而导致父进程长时间阻塞的问题。

当然，操作系统复制父进程页表的时候，父进程也是阻塞中的，不过页表的大小相比实际的物理内存小很多，所以通常复制页表的过程是比较快的。

不过，如果父进程的内存数据非常大，那自然页表也会很大，这时父进程在通过 `fork` 创建子进程的时候，阻塞的时间也越久。所以，有两个阶段会导致阻塞父进程：

- **创建子进程的途中**，由于要**复制父进程的页表**等数据结构，阻塞的时间跟页表的大小有关，页表越大，阻塞的时间也越长；
- **创建完子进程后**，如果子进程或者父进程修改了共享数据，就会**发生写时复制**，这期间会拷贝物理内存，如果内存越大，自然阻塞的时间也越长；

触发重写机制后，主进程就会创建重写 AOF 的子进程，此时父子进程共享物理内存，重写子进程只会对这个内存进行只读，重写 AOF 子进程会读取数据库里的所有数据，并逐一把内存数据的键值对转换成一条命令，再将命令记录到重写到新的 AOF 文件，最后再替换掉原有的AOF文件。

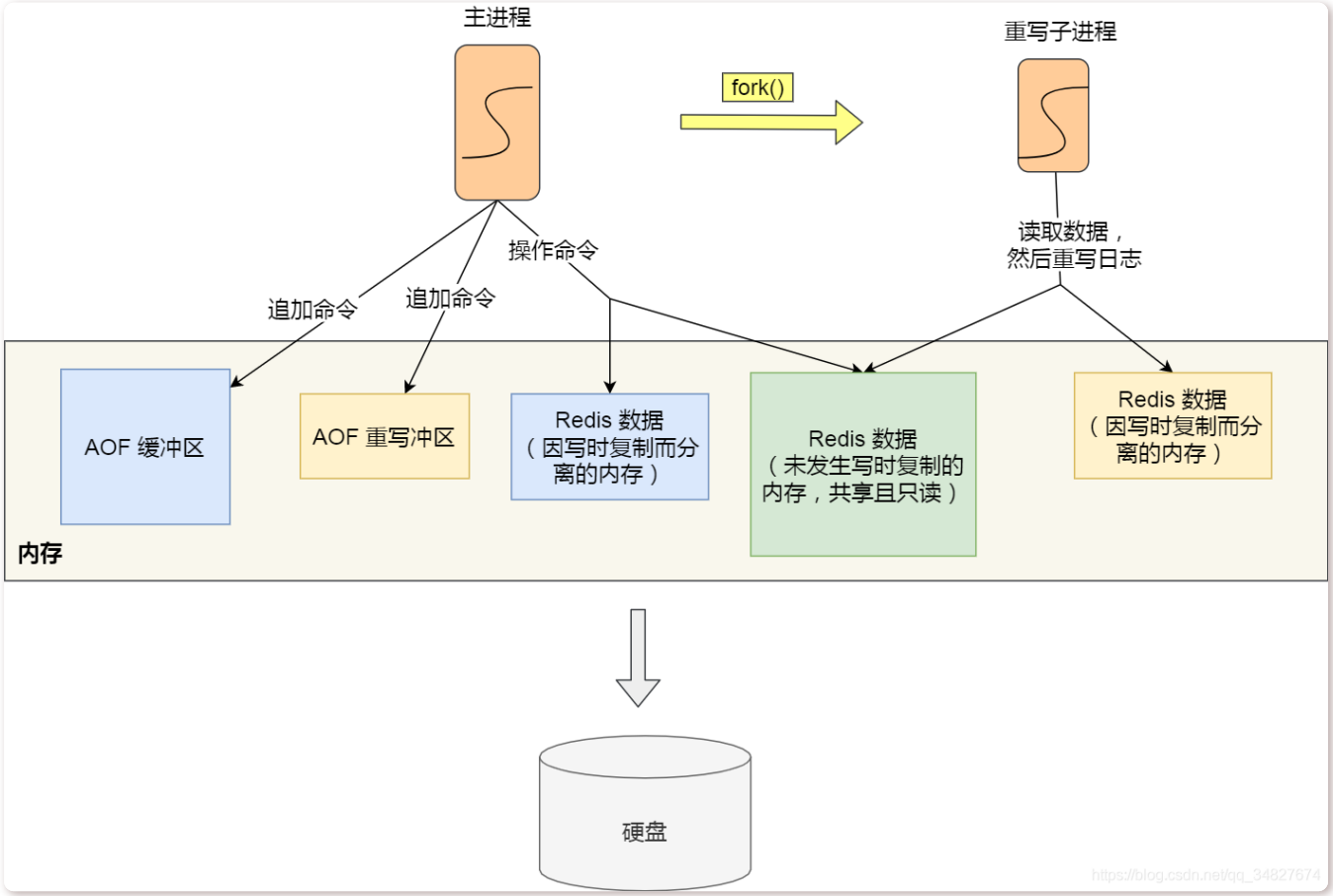
> 但是子进程重写过程中，主进程依然可以正常处理命令。

如果此时**主进程修改了已经存在 key-value，就会发生写时复制，注意这里只会复制主进程修改的物理内存数据，没修改物理内存还是与子进程共享的**。

所以如果这个阶段修改的是一个 bigkey，也就是数据量比较大的 key-value 的时候，这时复制的物理内存数据的过程就会比较耗时，有阻塞主进程的风险。

还有个问题，重写 AOF 日志过程中，如果主进程修改了已经存在 key-value，此时这个 key-value 数据在子进程的内存数据就跟主进程的内存数据不一致了，这时要怎么办呢？

为了解决这种数据不一致问题，Redis 设置了一个 **AOF 重写缓冲区**，这个缓冲区在创建 `bgrewriteaof` 子进程之后开始使用。在重写 AOF 期间，当 Redis 执行完一个写命令之后，它会**同时将这个写命令写入到「AOF 缓冲区」和「AOF 重写缓冲区」**。



也就是说，在 `bgrewriteaof` 子进程执行 AOF 重写期间，主进程需要执行以下三个工作：

- 执行客户端发来的命令；
- 将执行后的写命令追加到 「**AOF 缓冲区**」；
- 将执行后的写命令追加到 「**AOF 重写缓冲区**」；

当子进程**完成 AOF 重写工作**，即扫描数据库中所有数据，逐一把内存数据的键值对转换成一条命令，再将命令记录到重写日志后，会向**主进程发送一条信号**，信号是进程间通讯的一种方式，且是异步的。

主进程收到该信号后，会调用一个信号处理函数，该函数主要做以下工作：

- 将 **AOF 重写缓冲区**中的所有内容**追加**到新的 AOF 的文件中，使得新旧两个 AOF 文件所保存的数据库状态一致；
- 新的 AOF 的文件进行**改名**，**覆盖**现有的 AOF 文件。

信号函数执行完后，主进程就可以继续像往常一样处理命令了。

在整个 AOF **后台重写过程**中，除了**发生写时复制会对主进程造成阻塞**，还有**信号处理函数执行时也会对主进程造成阻塞**，在其他时候，AOF 后台重写都不会阻塞主进程。

1.7、优缺点

优点	缺点
简单：AOF日志是一个简单的文本文件，易于理解和操作	**文件尺寸**：相较于RDB持久化方式，AOF日志文件通常较大
易于恢复：在AOF文件中，每个写操作都以追加的方式记录	**写入性能**：AOF持久化方式相对RDB方式更耗费写入性能
可读性：AOF文件是可读的，可以用于手动分析和恢复数据	**内存占用**：相较于RDB持久化方式，AOF需要更多的内存
灵活：AOF提供不同级别的持久化选项，如每秒同步或追加	**恢复时间**：AOF恢复时间相对较长，因为需要重新执行操作
	启动时间：AOF恢复需要执行大量写操作，启动时间可能较长

2、RDB快照

2.1、概述

前面提及到，在Redis中提供了两种持久化的方式，分别是 *AOF日志* 和 *RDB快照* 。这两种技术都会用各用一个日志文件来记录信息，但是记录的内容是不同的。

- AOF 文件的内容是操作命令；
- RDB 文件的内容是二进制数据。

> 那么到底什么是快照呢?

所谓的快照，就是记录某一个瞬间东西，比如当我们给风景拍照时，那一个瞬间的画面和信息就记录到了一张照片。

所以，RDB 快照就是**记录某一个瞬间的内存数据**，记录的是实际数据，而 AOF 文件记录的是命令操作的日志，而不是实际的数据。

因此在 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。

2.2、实现方式

Redis 提供了两个命令来生成 RDB 文件，分别是 `save` 和 `bgsave`，他们的区别就在于是否在「主线程」里执行：

- 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**；
- 执行了 `bgsave` 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**；

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 还可以通过配置文件的选项来实现每隔一段时间自动执行一次 `bgsave` 命令，默认会提供以下配置：

```
# 900 秒之内，对数据库进行了至少1次修改时执行bgsave
save 900 1
# 300 秒之内，对数据库进行了至少10次修改时执行bgsave
save 300 10
# 60 秒之内，对数据库进行了至少10000次修改时执行bgsave
save 60 10000
```

> 🌈
>
> 这里有一点值得一提：虽然配置文件里面的配置使用的是save，但是实际执行的命令其实是 `bgsave`。

这里提一点，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。

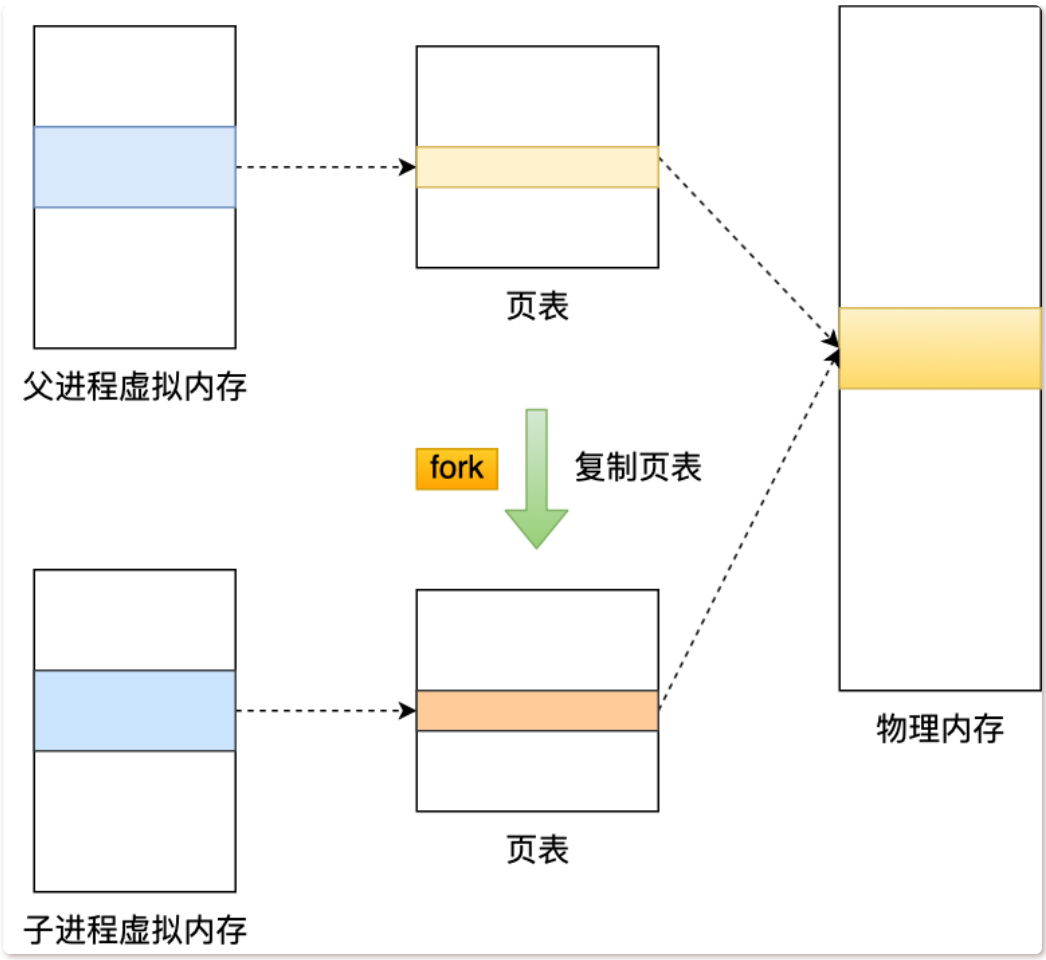
所以可以认为，执行快照是一个**比较重**的操作，如果频率太频繁，可能会对 Redis 性能产生影响。如果频率太低，服务器故障时，丢失的数据会更多。

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。

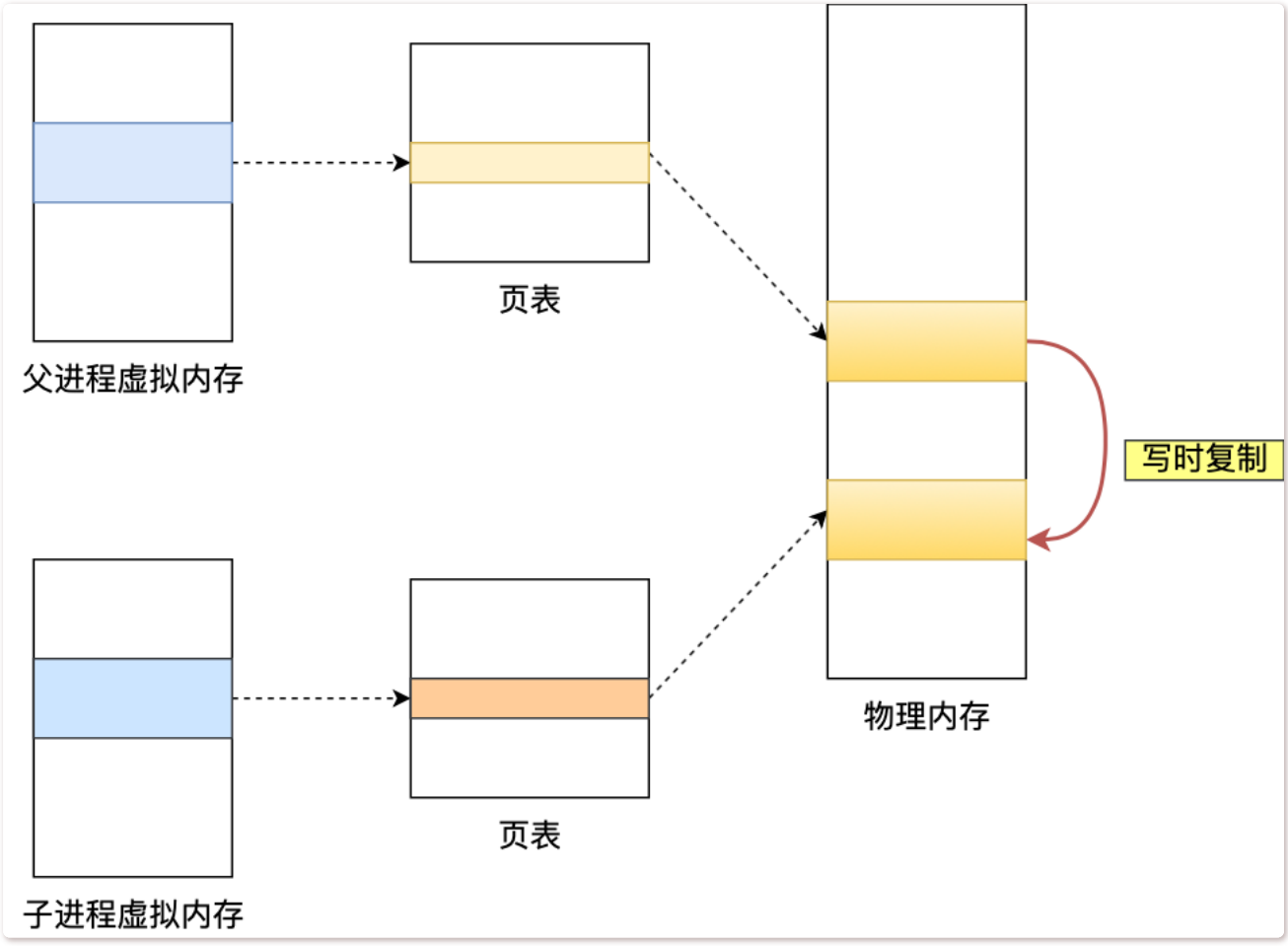
这就是 RDB 快照的缺点，**在服务器发生故障时，丢失的数据会比 AOF 持久化的方式更多**，因为 RDB 快照是全量快照的方式，因此执行的频率不能太频繁，否则会影响 Redis 性能，而 AOF 日志可以以秒级的方式记录操作命令，所以丢失的数据就相对更少。

2.3、实现原理

执行 `bgsave` 命令的时候，同样使用到了**写时复制技术(Copy-On-Write, COW)**，会通过 `fork()` 创建子进程，此时子进程和父进程是共享同一片内存数据的，因为创建子进程的时候，会复制父进程的页表，但是页表指向的物理内存还是一个。



只有在发生修改内存数据的情况时，物理内存才会被复制一份。



这样在执行 `bgsave` 命令时，Redis 便**可以继续处理操作命令**，同时也是为了减少创建子进程时的性能损耗，从而加快创建子进程的速度，毕竟创建子进程的过程中，是会阻塞主线程的。

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（父进程）里的内存数据，并将数据写入到 RDB 文件。

当主线程（父进程）对这些共享的内存数据也都是只读操作，那么，主线程（父进程）和 `bgsave` 子进程相互不影响。

但是，如果**主线程（父进程）要修改共享数据**里的某一块数据（比如键值对 `A:xbaozi` ）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:xbaozi` ），然后**主线程**在这个数据副本（键值对 `A:xbaozi` ）进行修改操作（键值对 `A:xbaoziplus` ）。与此同时， `bgsave` **子进程**可以继续把**原来的数据**（键值对 `A:xbaozi` ）写入到 RDB 文件。

就是这样，Redis 使用 bgsave 对当前内存中的所有数据做快照，这个操作是由 bgsave 子进程在后台完成的，执行时不会阻塞主线程，这就使得主线程同时可以修改数据。

2.4、极端情况

不知道大家有没有发现， `bgsave` 快照过程中，如果主线程修改了共享数据，**发生了写时复制后，RDB 快照保存的是原本的内存数据**，而主线程刚修改的数据，是没办法在这一时间写入 RDB 文件的，只能交由下一次的 `bgsave` 快照。

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（父进程）的内存数据和子进程的内存数据已经分离了，子进程写入到 RDB 文件的内存数据只能是原本的内存数据。

如果系统恰好在 RDB 快照文件创建完毕后崩溃了，那么 Redis 将会丢失主线程在快照期间修改的数据。

另外，写时复制的时候会出现这么个极端的情况。

在 Redis 执行 RDB 持久化期间，刚 `fork` 时，主进程和子进程共享同一物理内存，但是途中主进程处理了写操作，修改了共享内存，于是当前被修改的数据的物理内存就会被复制一份。

那么极端情况下，**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，我们要留意下快照过程中内存的变化，防止内存被占满了。

> 🌈
>
> 这里补上一嘴，使用AOF日志做持久化的时候也会存在这个问题，因为AOF日志持久化中也用到了写时复制技术进行持久化实现。

2.5、优缺点

优点	详细描述	缺点	详细描述
快速保存和加载	RDB文件格式简洁，加载速度快，可以快速保存和加载数据。	**大Key问题**	RDB在保存和加载时，存在大Key问题，可能会导致性能下降。
数据压缩	RDB支持数据压缩，可以减少数据存储空间，提高Redis的性能。	**数据一致性问题**	在RDB保存期间，如果发生故障或中断，可能会导致数据不一致的问题。
快照备份	RDB可以作为Redis的快照备份，提高Redis的可靠性和恢复能力。	**频繁写入问题**	如果Redis需要频繁写入数据，而没有足够的时间进行RDB保存，可能会导致数据丢失。

3、混合体实现

尽管 RDB 比 AOF 的数据恢复速度快，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。

那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？

当然有，那就是将 RDB 和 AOF 混合使用，这个方法是在 Redis4.0 提出的，该方法叫**混合使用 AOF 日志和内存快照**，也叫混合持久化。

如果想要开启混合持久化功能，可以在 Redis 配置文件将下面这个配置项设置成 yes：

```
aof-use-rdb-preamble yes
```

混合持久化工作在 **AOF 日志重写过程**。

当开启了混合持久化时，在 AOF **重写日志**时，`fork` 出来的重写子进程会**先将与主进程共享的内存数据以 RDB 方式写入到 AOF 文件**，然后**主进程处理的操作命令**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**。

也就是说，使用了混合持久化，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**。

这样的好处在于，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，这里的内容是 Redis 后台子进程重写 AOF 期间，主进程处理的操作命令，可以使得**数据更少的丢失**。

4、大Key问题

4.1、何为大key

在Redis中，大Key问题是**指存储在Redis中的某个Key所对应的value过大**，导致Redis性能下降或崩溃的问题。对于不同类型的数据结构，大Key的定义是不同的。对于string类型来说，一般认为超过 `10KB` 的Value是大Key；对于set、zset、hash等类型来说，一般认为数据量超过 `5000条` 的Key是大Key。

当Redis存储的Key过大时，会对Redis的性能产生负面影响，如内存使用率过高、写入和读取数据的速度变慢等。这可能会导致Redis阻塞或崩溃。

4.2、负面影响

4.2.1、持久化影响

当 AOF 写回策略配置了 **Always** 策略，如果写入是一个大 Key，主线程在执行 `fsync()` 函数的时候，阻塞的时间会比较久，因为当写入的数据量很大的时候，数据同步到硬盘这个过程是很耗时的。

AOF 重写机制和 RDB 快照（`bgsave` 命令）的过程，都会分别通过 `fork()` 函数创建一个子进程来处理任务。会有两个阶段会导致阻塞父进程（主线程）：

- 创建子进程的途中，由于要复制父进程的页表等数据结构，阻塞的时间跟页表的大小有关，页表越大，阻塞的时间也越长；
- 创建完子进程后，如果父进程修改了共享数据中的大 Key，就会发生写时复制，这期间会拷贝物理内存，由于大 Key 占用的物理内存会很大，那么在复制物理内存这一过程，就会比较耗时，所以有可能会阻塞父进程。

4.2.2、其他影响

大 key 除了会影响持久化之外，还会有以下的影响。

- **客户端超时阻塞**。由于 Redis 执行命令是单线程处理，然后在操作大 key 时会比较耗时，那么就会阻塞 Redis，从客户端这一视角看，就是很久很久都没有响应。
- **引发网络阻塞**。每次获取大 key 产生的网络流量较大，如果一个 key 的大小是 1 MB，每秒访问量为 1000，那么每秒会产生 1000MB 的流量，这对于普通千兆网卡的服务器来说是灾难性的。
- **阻塞工作线程**。如果使用 del 删除大 key 时，会阻塞工作线程，这样就没办法处理后续的命令。
- **内存分布不均**。集群模型在 slot 分片均匀情况下，会出现数据和查询倾斜情况，部分有大 key 的 Redis 节点占用内存多，QPS 也会比较大。

4.2、如何避免大key

在开发过程中为了避免大key问题，通常需要采取一些必要的措施：

1. **定期清理或拆分大Key**：对于存储在Redis中的大Key，可以定期进行清理或拆分，以避免单个Key过大导致的问题。
2. **优化数据结构**：对于需要存储大量数据的Key，可以尝试使用更优化的数据结构，如使用Hashes或Lists代替普通的String类型，以减少内存占用和读写时间。
3. **限制Key的大小**：可以根据实际情况，限制Key的大小，以避免过大Key的出现。
4. **使用分布式缓存**：如果需要存储大量的数据，可以考虑使用分布式缓存，将数据分散存储在多个节点上，以避免单节点性能问题。
5. **使用Redis Cluster**：Redis Cluster可以将数据分布在多个节点上，可以提高Redis的可用性和性能，避免大Key问题。

5、应用场景

截止现在接触到个严格来说应该算是有三种持久化方式了：

1. **AOF持久化**：
    - 优点：数据完整性较好，能够提供更高的数据安全性。适合对数据安全性要求较高的场景。
    - 缺点：写入性能相对较低，因为每个写命令都需要同步写入磁盘。
    - 适用场景：对数据完整性要求较高，且可以容忍一定性能损耗的场景。
2. **RDB持久化**：
    - 优点：写入性能较好，因为是定期快照的方式进行持久化。
    - 缺点：可能会有数据丢失，因为是定期快照，定期写入磁盘。
    - 适用场景：对性能要求较高，可以容忍一定数据丢失的场景。
3. **混合使用**：
    - AOF和RDB持久化的混合使用可以兼顾两者的优势，同时也可以减轻各自的缺点。
    - 可以根据具体的业务需求和场景选择定期执行RDB快照，以及每秒执行AOF日志写入的方式。

持久化技术	优点	缺点	适用场景
AOF	数据完整性好	写入性能较低	**对数据完整性要求较高**，且可以容忍一定性能损耗的场景
RDB	写入性能好	可能有数据丢失	**对性能要求较高**，可以容忍一定数据丢失的场景
混合使用	兼顾AOF和RDB的优点	兼顾AOF和RDB的缺点	对性能和数据安全性**都有一定要求**的场景