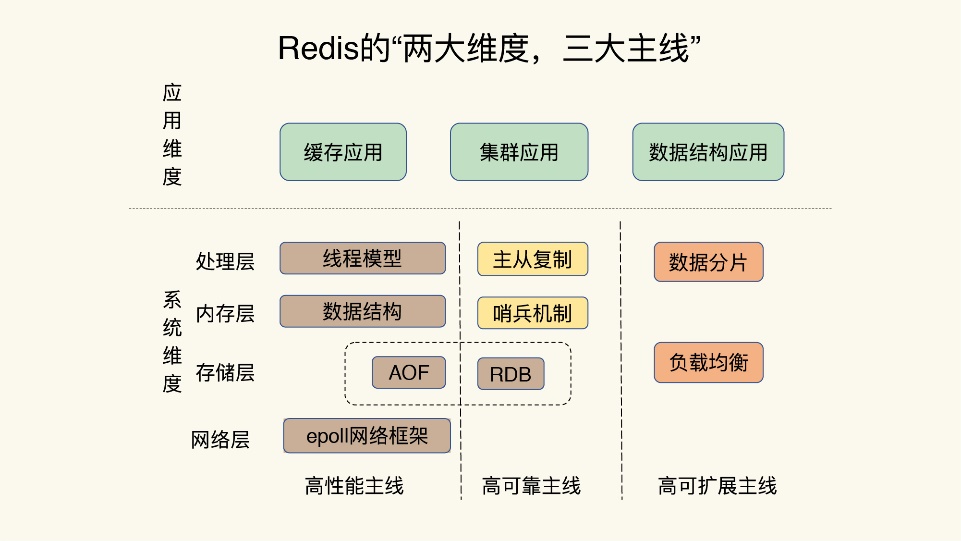
## 开篇部分

* 1. Redis知识全景图： 两大维度，三大主线



两大维度： 系统维度 和 应用维度

系统维度上 连接 Redis关键技术设计原理

应用维度：

三大主线： 高性能，高可靠，高可扩展

高性能：

线程模型，数据结构，持久化，网络框架

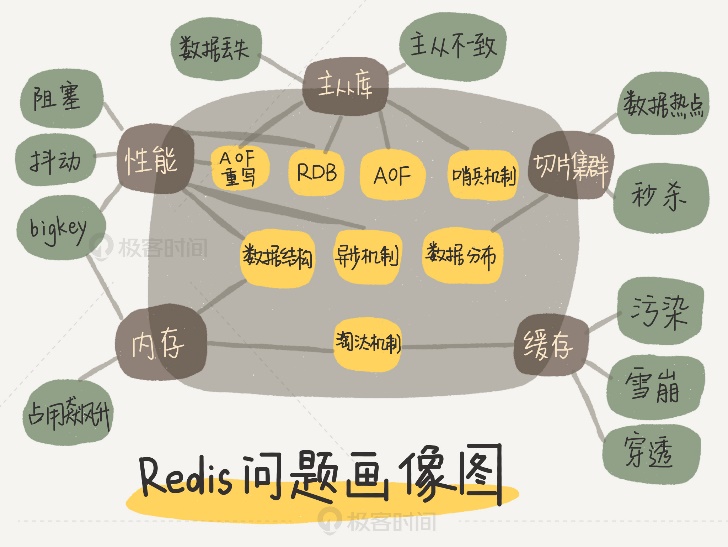
高性能：

主从复制， 哨兵机制

高可扩展：

数据分片，负载均衡

* 1. Redis 的问题画像图：典型问题与技术点



## 基础部分

Redis是典型的键值数据库，非关系性数据库

#### 2.1 基本框架：简易版本的KV数据库SimpeKV设计

##### 2.1.1 数据模型：存什么样的数据

对键值对数据库而言，基本数据类型是 key-value模型， 不同的键值对数据库所支持的key值差异不大，而value支持的类型差别较大， 比如Memcached支持的value的类型只有string， Redis支持的类型包括string，哈希，列表，集合等。在选择键值对数据库时，value支持的类型也成为一个重要的选择。

##### 2.1.2操作接口：对数据可做什么样的操作

SimpleKV是一个简单的kv键值对数据库，基本操作为增删查改，支持三种基本操作： PUT， GET， DELETE。

PUT ：新写入或更新一个 key-value 对

GET ： 根据一个key读取对应的value值

DELETE：根据key值删除整个key-value对

SCAN: 根据一段 key 的范围返回相应的value值

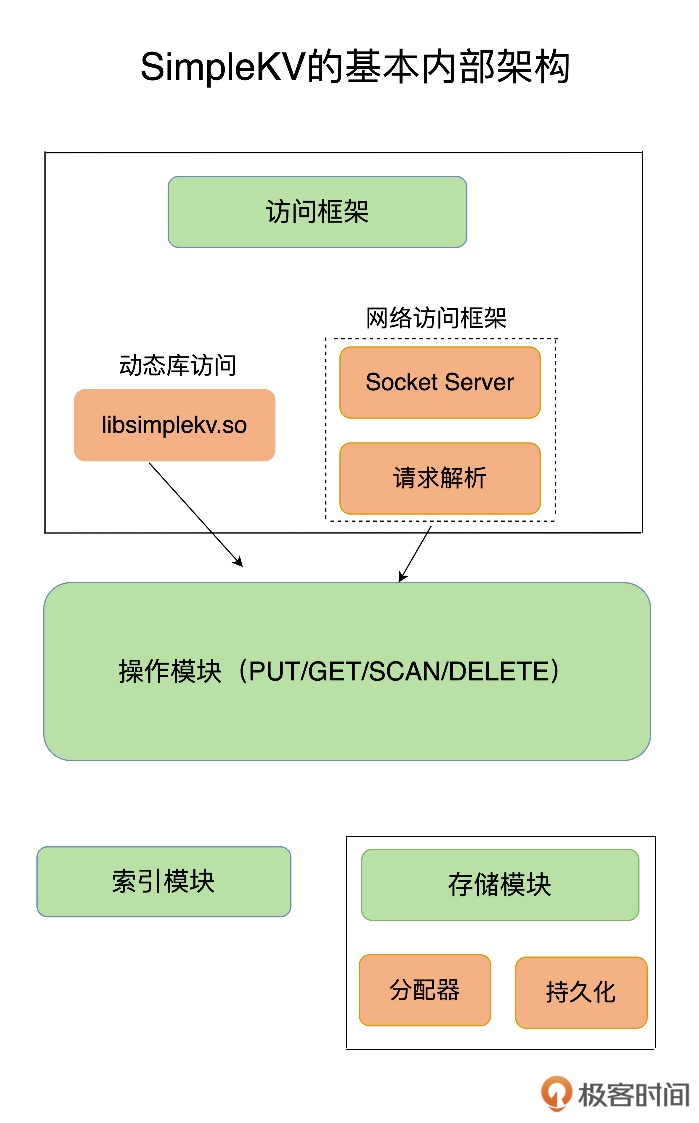
##### 2.1.3 键值对保存位置：内存 or 外存

保存在内存： 读写块，速度约在 百ns级别 但是存在潜在风险，一旦断电，所有数据将丢失

保存在外存： 数据可以避免丢失，但是磁盘读写速度慢(通常几ms级别)，会拉低kv数据库的整体性能。

因此，如何设计选择，通过根据kv数据库的主要应用场景。 Memcached和redis都是属于内存键值数据库。

##### 2.1.4 键值数据库主要部分： 访问框架 、 索引模块、 操作模块 、 存储模块



###### 采用什么访问模式

* + - * 1. 通过函数库调用的方式供外部应用使用（本地端）
        2. 通过网络框架以 socket 通信的形式对外提供键值对访问（Memcached和Redis）

###### 如何定位键值对的位置

SimpleKV依赖于 索引模块 来定位。

索引的作用在于让键值kv数据库根据 key 找到相应的 value 的存储位置，进而执行操作。

索引的类型：哈希表 ， B+树， 字典树， 跳表，不同的索引结构在性能，空间消耗，并发控制具有不同的特征。

哈希表： Memcached和Redis采用 key-value索引，采用哈希表作为索引的原因主要是键值数据保存在内存中，而内存的高性能可以和哈希表的O(1)操作复杂度想匹配。

RoscksDB：采用调表作为内存中的key-value索

###### 不同操作的具体逻辑

GET/SCAN: 根据 value的存储位置返回value的值即可。

PUT 会涉及到内存空间分配

DELETE 操作，SimpleKV 需要删除键值对，需要释放相应的空间，由 分配器 完成。

以上操作都涉及到 SimpleKV的存储模块

###### 存储模块

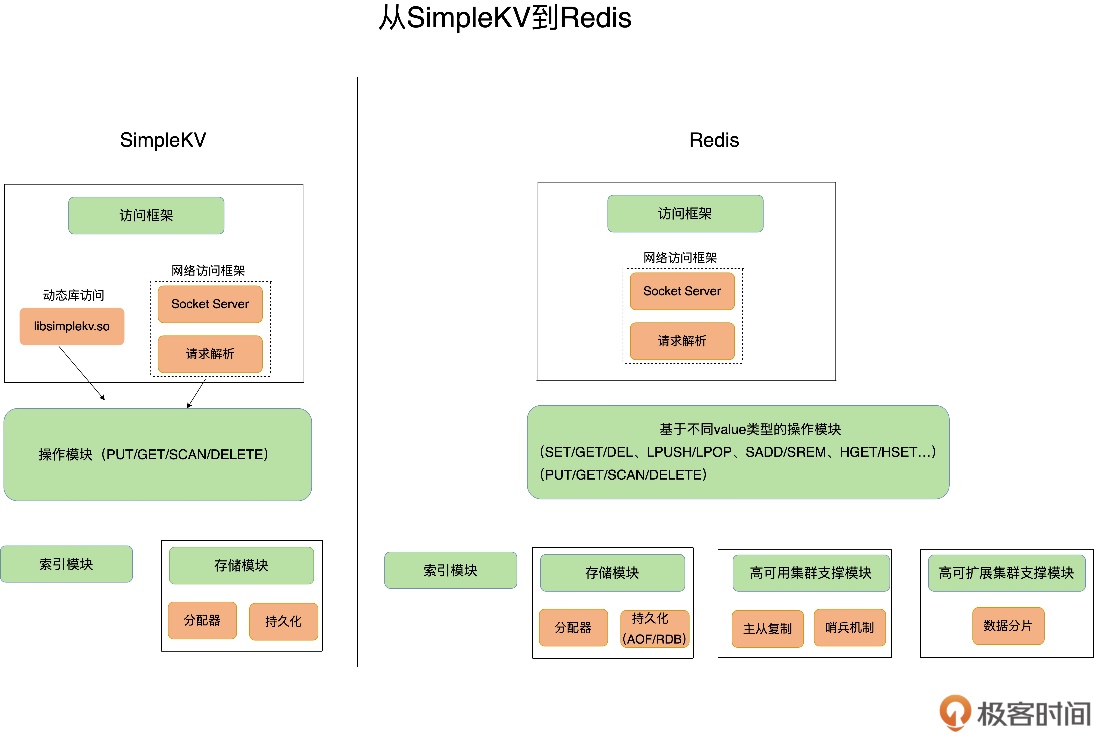
SimpleKV采用内存分配器glibc的malloc和free， 因此，SimpleKV不需要考虑内存管理问题，但是键值数据库的键值对大小通常不一样，而glibc的分配器在处理随机大小的内存块分配时，表现并不好，一旦数据过大，很容易出现严重的内存碎片问题。因此，分配器 是KV数据库的一个关键因数。 Redis 提供了多种内存分配器的选择。

一般而言，都需要重启后快速提供服务 涉及到 持久化

SimpleKV的存储模块增加了持久化功能，这里采用 文件形式，将键值数据通过调用本地文件系统接口保存到磁盘。

SimpleKV两种 持久化 方式：

1. SimpleKV直接对键值对落盘保存(每次都保存)，它的可靠性强，但是每次都要写盘，性能会略低。
2. SimpleKV只是周期性的把内存中的键值保存到文件中，避免频繁写盘操作，但SimpleKV数据有丢失的可能。



SimpleKV 与 Redis对比图

#### 2.2 数据结构： Redis的数据结构

##### 2.2.1 Redis速度快的原因

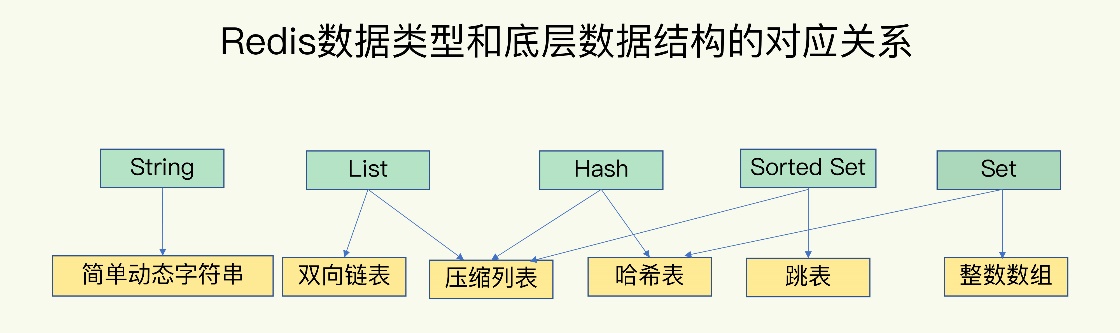
1. Redis是内存数据库，所有操作都是在内存上完成的，而内存访问速度本来就快。
2. Redis自身的数据结构，高效的数据结构是Redis快速处理数据的基础。

##### 2.2.2 Redis数据结构 和 底层数据实现

###### 2.2.2.1 Redis数据类型(5种)

1. String(字符串) 底层是 sds
2. List(列表) 底层是 ziplist 和 doublelist
3. Hash(哈希) 底层是 ziplist 和 hashtable
4. Set(集合) 底层是 ziplist 和 intset
5. Sorted Set(有序集合) 底层是 ziplist 和 skiplist

其中 list，hash，Set 和 SortSet统称为 集合类型，特点是一个键对应一个集合的数据

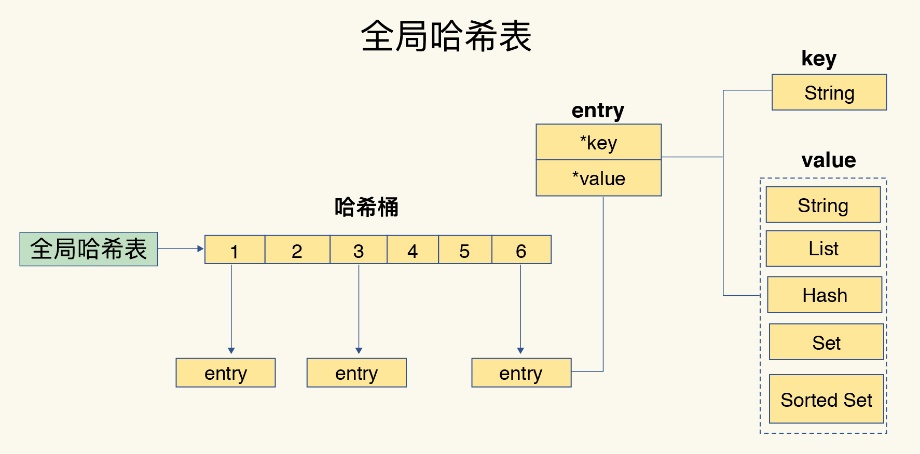


###### 2.2.2.2 Redis底层数据结构(6种)

1. 简单动态字符串 simple dynamic string
2. 双向链表 doublelist O(N)
3. 压缩列表 ziplist O(N)
4. 哈希表 hashtable O(1)
5. 跳表 sliplist O(logN)
6. 整数数组 inset O(N)

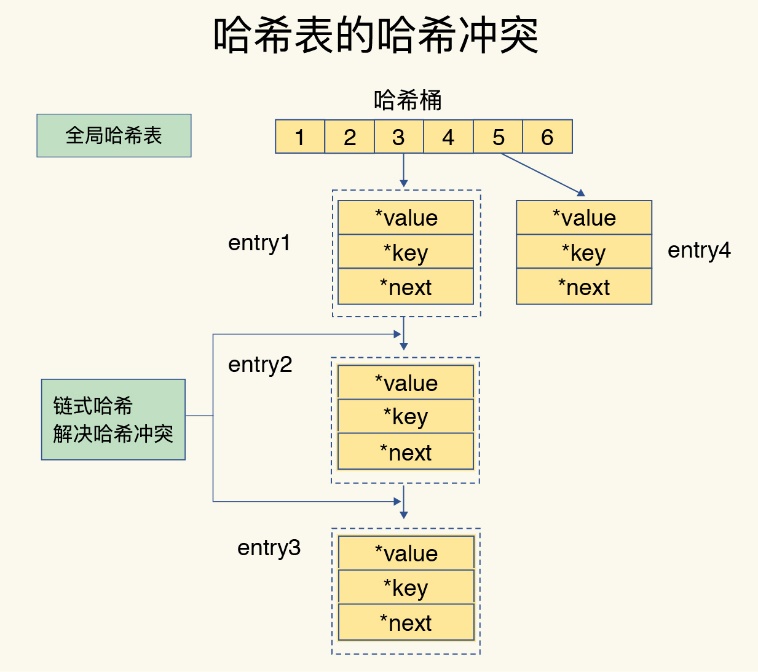
###### 2.2.2.3 键和值组织结构：

Redis使用了一个 哈希表 来保存所有的键值对： 一个哈希表，实际上就是一个数组，数组的每个元素称位一个哈希桶，也即是哈希表由多个哈希桶组成，每个哈希桶保存了键值对数据，哈希桶中的元素保存的并不是值本身，而是指向值的指针。即不管是string还是集合，哈希桶中的元素都是指向他们的指针。



##### 2.2.3 Redis中哈希冲突和rehash

Redis中，采用 链地址法(链式哈希) 来处理哈希冲突：同一个哈希桶中的多个元素用一个链表来保存，它们之间依次用指针连接



若哈希填充过多，就可能导致某些哈希冲突链过长，进而导致链表上元素查找过长，效率降低，从而影响效率。因此，Redis会对哈希表做rehash 操作，rehash也即是增加现有哈希桶的数量，逐个让增多的entry元素能够在更多的桶之间分散保存。

Redis默认使用了两个全局哈希表：哈希表1 和 哈希表2 ；一开始，当插入数据时，默认使用哈希表1，此时哈希表2没有被分配空间，随着数据逐步增多，Redis开始执行rehash。主要分为三步：

1. 给哈希表2分配更大的空间， 比如时哈希表1大小的两倍
2. 把哈希表1的数据重新映射并拷贝到哈希表2中（涉及到大量数据的拷贝，若一次性拷贝则可能导致Redis线程拥塞，为避免此问题，Redis采用了渐进式rehash）
3. 释放哈希表1的空间。

##### 2.2.4 集合数据操作效率

###### 2.2.4.1 Redis集合数据

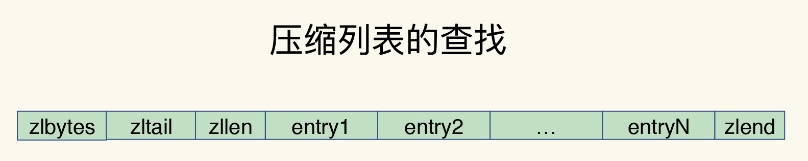
* + - * 1. 对于集合类型的value，第一步时通过全局哈希表找到对应的哈希桶位置，第二步时在集合中再增删改查。
        2. 集合操作的效率与集合首先是底层数据结构相关，比如使用哈希表实现的集合比用链表实现的集合的访问效率要高，其次是这些操作效率与操作本身有关，比如读写一个元素比读写所有元素的操作效率要高。

###### 2.2.4.2 Redis集合底层数据结构

1. 主要有 5 种， 整数数组(intset), 双向链表(doublelist), 哈希表(hashtable), 压缩列表(ziplist), 跳表(skiplist)

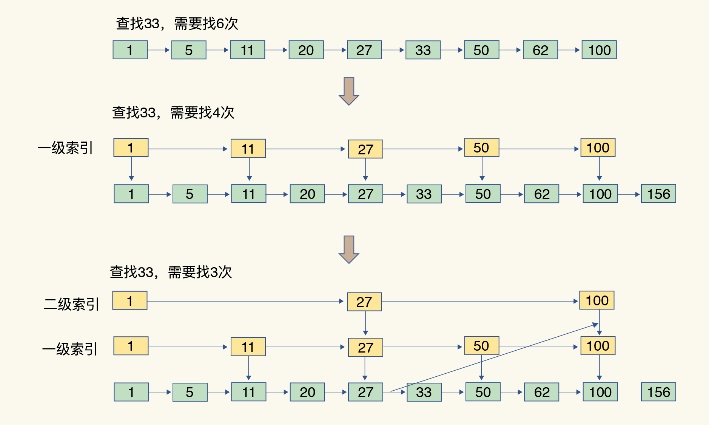


1. 压缩列表(ziplist)类似一个数组，数组中每个元素对应保存一个数据，每个数据有三个字段 zlbytes， zltail， zllen，分别表示列表长度，列表尾，列表中entry个数；压缩列表在表尾还有一个zlend，表示列表结束。



也即是 表尾和表头的查找为O(1)，其余元素O(N)

1. 跳表(ziplist)是在链表基础上，增加了多级索引，通过索引位置的跳转，实现数据的快速定位，跳表的查找复杂度是O(lonN)



跳表的查找过程

每日一问:整数数组(intset)和压缩列表(ziplist)在查找时间复杂度方面并没有太大优势，那为什么Redis还会把它们作为底层数据结构呢？

两个原因：

1. 内存利用率，数组和压缩列表都是非常紧凑的数据结构，它们占用内存比链表更少。Redis是内存数据库，大量的数据存储在内存中，因此需要尽可能的优化，提高内存利用率。
2. 数组对CPU高速缓存支持更加友好，所以Redis在设计时，集合数据较少情况下，默认采用内存紧凑排列的方式进行存储，同时利用CPU高速缓存不会降低访问速度。当元素超过设定的阈值后，避免查询时间复杂度过高，转为哈希和跳表数据结构存储，保证查询效率。

#### 2.3 高性能IO模型 ： 为什么单线程Redis能够那么快？

Redis 的网络模型 ：单 Reactor 模式， non-blocking io + epoll

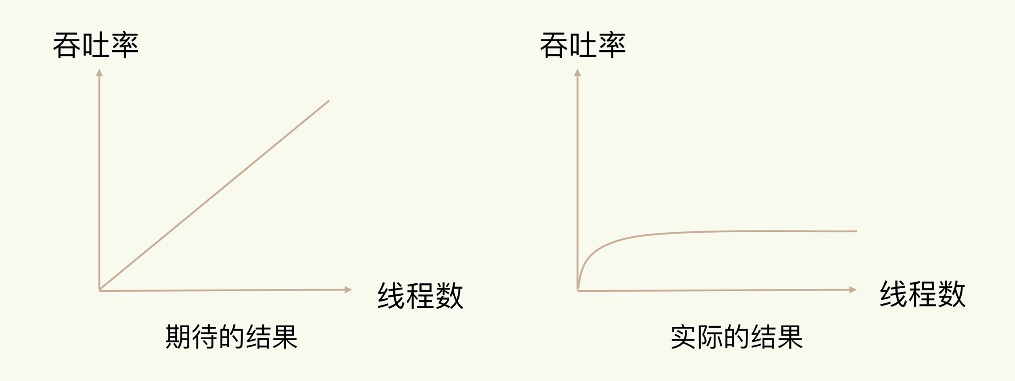
##### 2.3.1 Redis真的是单线程吗？

通常所说的 Redis单线程 主要是指Redis的网IO和键值对读写是由一个线程来完成的，这也是Redis对外提供键值存储服务的主要线程，其他功能，如持久化，异步删除，集群数据同步等，其实是由额外的线程执行的。

##### 2.3.2 Redis 为什么用单线程

###### 2.3.2.1 多线程的开销

1. 多线程可以增加系统吞吐率或增加系统可扩展性，在合理的资源分配情况下，可以增加系统中处理请求操作的的资源实体，进而提升系统能够同时处理的请求数，即吞吐率。



线程数与系统吞吐率

1. 原因在于：系统中通常存在被多线程访问的共享资源，比如临界资源。当多个线程需要修改访问临界资源时，为了保证共享资源的正确性，就需要额外的机制进行保证(比如锁)，而这个额外的机制，会带来额外的开销(加解锁的开销)。
2. 多线程编程模式面临的共享资源的并发访问控制问题。
3. 并发访问控制是多线程编程开发的难点，如果没有精细的设计，比如只是简单采用粗粒度的互斥锁，往往会导致不理想的结果，即使增加了线程，大部分线程也在等待获取访问共享资源的互斥锁，导致并行变串行，使得系统吞吐率没有随着线程的增加而增加。
4. 多线程开发中一般都会引入 同步原语 来保护共享资源的并发访问， 但这会降低易调试性和可维护性。

**为避免以上这些问题，Redis直接采用了单线程模式。**

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。

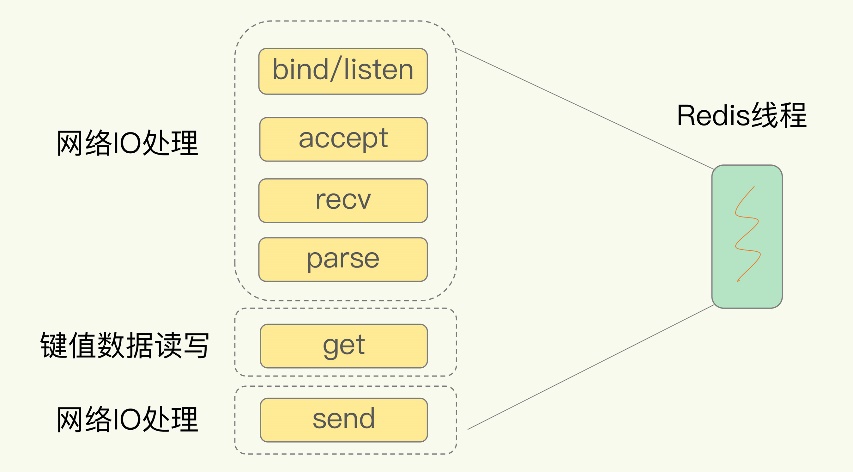
单线程的Redis也可以获取高性能，这与多路复用的IO模型密切相关，因为这避免了accept(),recv()/send()等潜在的网络IO阻塞点。

##### 2.3.3 单线程Redis为什么那么快(每秒十万级别的处理能力)

* + - 1. Redis的大部分操作直接在内存中完成，而内存的速度本来就快。
      2. Redis的高效数据结构，Redis采用全局哈希表，通过key值可以很快获取value的位置。比如哈希表和跳表。
      3. Redis采用了**多路复用机制**，使其在网络IO操作中能够并发处理大量的客户请求，实现高吞吐率。

##### 2.3.4基于多路复用的高性能 I/O 模型

###### 2.3.4.1 基本IO模型与阻塞点

****

**Redis基本IO模型**

由于 Redis 在网络IO处理采用的是单线程，若线程被阻塞，则无法处理其他客户端。但是，**socket网络模型本身支持非阻塞模式**。

1. 阻塞模式：accept()、recv()都可能阻塞：
   1. 当redis监听到一共客户端有连接请求时，但一直未能成功建立连接，则会阻塞到accept()上，导致其他客户端无法与Redis建立连接。
   2. 类似，Reids通过recv()从一个客户端读取数据时，如果数据一直没有到达，Redis会阻塞在recv()。
2. 非阻塞模式：accept(),recv()等函数本身是阻塞的，但是可以将监听的fd设置为非阻塞的，来实现非阻塞监听模式。



Redis套接字类型与非阻塞设置

设置为非阻塞模式后，Redis线程可以不用继续等待，由某种机制继续监听套接字连接请求，若有请求时则会通知Redis。（accept(),recv()同理），通过这种方式保证Redis线程，既不会像基本IO模型中一直在阻塞点等待，也不会导致Redis无法处理实际到达的连接请求或数据，

###### 2.3.4.2 基于多路复用的高性能I/O模型

I/O多路复用是一种同步I/O模型(一个线程处理多个IO流)，功能是一个线程同时监听多个文件描述符的功能。多路指的是网络连接，复用指的是复用同一个线程。

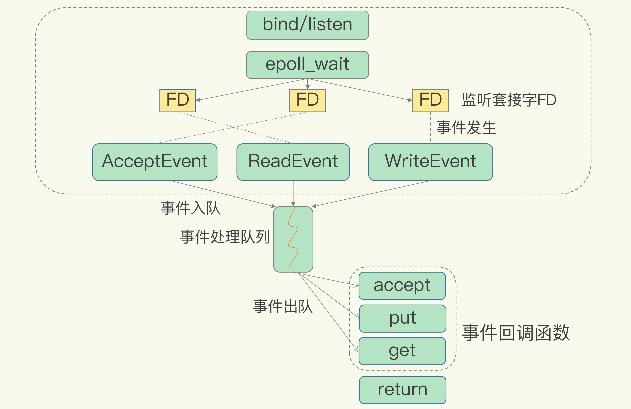
Redis的IO多路复用机制：

基于Linux的select/epoll

这些机制允许内核中，同时存在多个监听套接字和已连接的套接字。

一旦内核监听到套接字上有数据就会交给Redis线程处理。

通过以上方式，就实现了一个Redis线程处理多个IO流的效果。



基于多路复用的Redis高性能IO模型

###### 2.3.4.3 Redis高性能IO模型

* + - 1. 为了在请求到达时能够通知Redis线程，select/epoll提供了基于事件的回调函数机制，即**针对不同的事件的发生，调用相应的处理函数**。
      2. select/epoll一旦监听到FD上有请求到达，就会触发相应的事件。这些事件被放入一个就绪事件队列，Redis的单线程对该事件队列不断进行处理，从而避免轮询是否有请求实际发生，避免造成CPU资源的浪费。
      3. Redis对事件队列中的事件处理时，会调用相应的处理函数，即实现了基于事件的回调。因为Redis一直对事件队列进行处理，所以能够及时响应客户端请求，从而提升Redis的响应性能。

不同的操作系统多路复用机制也是适用的，Linux下的select/epoll， 基于FreeBSD的kqueue，以及基于Soolaris的evport。

2020年5月，Redis 6.0中提出了多线程模型。Redis的瓶颈不在CPU,而在内存和网络，内存不够可以增加内存或通过数据结构等方式进行优化，但是Redis的网络IO的读写会占用CPU,因此，可以把网络处理改为多线程的方式，来提升性能。

Redis 6.0引入多线程主要有两个原因：

1. 充分利用服务器的多核资源
2. 多线程分摊Redis的同步IO读写负荷。

每日一问：你觉得还有哪些潜在的性能瓶颈吗？

Redis单线程处理IO请求性能瓶颈主要包括2个方面：  
1、任意一个请求在server中一旦发生耗时，都会影响整个server的性能，也就是说后面的请求都要等前面这个耗时请求处理完成，自己才能被处理到。耗时的操作包括以下几种：  
 a、操作bigkey：写入一个bigkey在分配内存时需要消耗更多的时间，同样，删除bigkey释放内存同样会产生耗时；  
 b、使用复杂度过高的命令：例如SORT/SUNION/ZUNIONSTORE，或者O(N)命令，但是N很大，例如lrange key 0 -1一次查询全量数据；  
 c、大量key集中过期：Redis的过期机制也是在主线程中执行的，大量key集中过期会导致处理一个请求时，耗时都在删除过期key，耗时变长(定时删除用的是主线程，每100ms一次)；  
 d、淘汰策略：淘汰策略也是在主线程执行的，当内存超过Redis内存上限后，每次写入都需要淘汰一些key，也会造成耗时变长；  
 e、AOF刷盘开启always机制：每次写入都需要把这个操作刷到磁盘，写磁盘的速度远比写内存慢，会拖慢Redis的性能；  
 f、 主从全量同步生成RDB：虽然采用fork子进程生成数据快照，但fork这一瞬间也是会阻塞整个线程的，实例越大，阻塞时间越久；  
2、并发量非常大时，单线程读写客户端IO数据存在性能瓶颈，虽然采用IO多路复用机制，但是读写客户端数据依旧是同步IO，只能单线程依次读取客户端的数据，无法利用到CPU多核。  
  
针对问题1，一方面需要业务人员去规避，一方面Redis在4.0推出了lazy-free机制，把bigkey释放内存的耗时操作放在了异步线程中执行，降低对主线程的影响。  
  
针对问题2，Redis在6.0推出了多线程，可以在高并发场景下利用CPU多核多线程读写客户端数据，进一步提升server性能，当然，只是针对客户端的读写是并行的，每个命令的真正操作依旧是单线程的。

#### 2.4 Redis持久化之 AOF日志：宕机了，Redis 如何避免数据丢失？

Redis 应用场景，作为 缓存，作为内存数据库等。作为缓存时，若一旦宕机，内存中的数据将全部丢失。

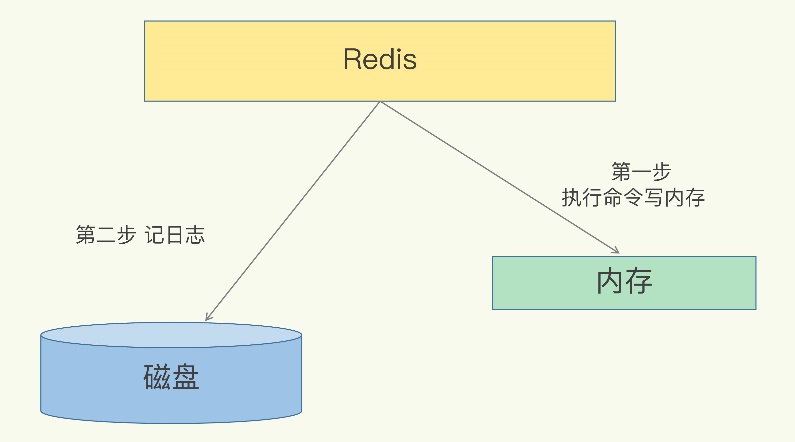
Redis 持久化，避免从后端数据库中进行恢复。Redis的两大持久化机制：AOF日志和RDB快照。

##### 2.4.1 AOF日志如何实现？

写前日志(Write Ahead Log,WAL) ：在实际写数据前，先把修改的数据写入到日志文件中，以便故障时进行恢复，然后再执行操作。

写后日志: 与写前日志相反，先执行命令，把数据写入内存，然后才记录日志。

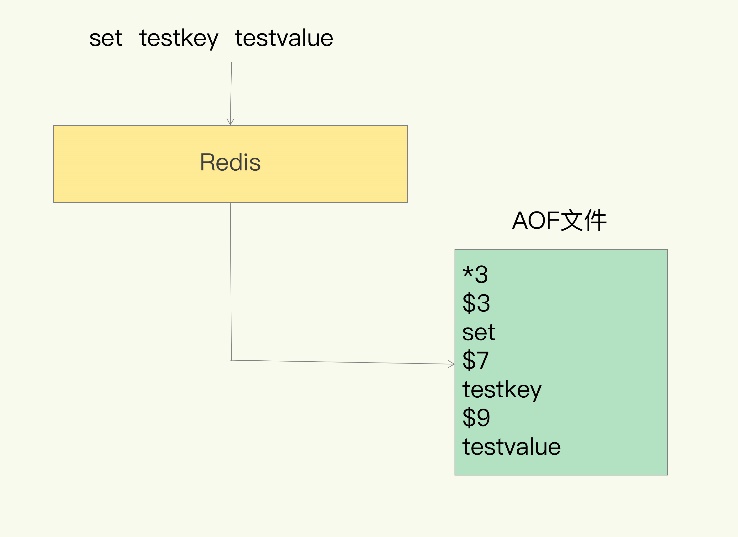
AOF日志是写后日志，也即是Redis先执行写命令，把数据写入内存，然后才记录日志。



Redis AOF操作过程

传统数据库的日志比如redo log(重写日志) 记录的是修改后的数据。

AOF记录的是Redis收到的每一条命令，这些命令是以文本形式保存的。Redis的AOF日志内容示例：



Redis AOF日志内容

写后日志这种方式，让系统先执行命令，只有执行成功的命令才会被记录到日志中，否则，系统直接向客户端报错。

Redis使用写后日志的好处

1. 可以避免记录错误命令的情况
2. 它是在命令执行后才记录日志，所以不会阻塞当前的写操作。

AOF潜在风险：

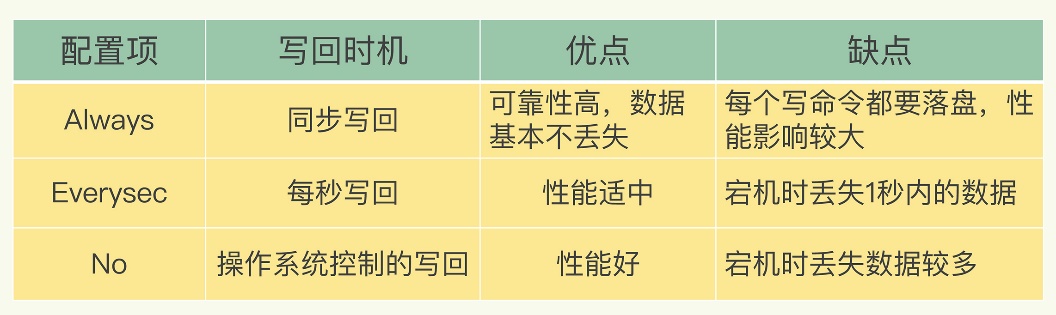
1. 数据丢失风险，若Redis直接作为数据库，则可能出现执行完一个命令还没来得及记录日志宕机的情况，数据就会丢失。
2. AOF日志是在主线程中执行的，若把日志文件写入磁盘，IO很慢，进而影响后续操作。

这两个风险都是和AOF写回磁盘的时机相关。

##### 2.4.2 AOF 三种写回策略

###### 2.4.2.1 AOF提供三种选择，即是AOF配置项 appendfsync 的三个可选值

* + - 1. Always 同步写回：每个写命令执行完，立即同步地将日志写回磁盘；基本可以做到数据不丢失，但是每一个写命令后都有一个慢速的落盘操作，不可避免地会影响主线程性能。
      2. 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回磁盘；采用一秒写回一次的频率。避免同步写回的性能开销，可以减少对性能的影响，但是若发生宕机，则上一秒未落盘的命令操作仍然会丢失。
      3. No 操作系统控制写回：把每个写命令执行完，只是先把日志写到AOF文件的内存缓冲区，由操作系统决定何时将缓冲区内容写回磁盘；只要AOF记录没有写回磁盘，一旦宕机对应的数据就丢失。



AOF三种写回策略

三种写回策略体现了系统设计的一个重要原则，即trade-off (取舍原则)，在性能与可靠性之间做取舍。

###### 2.4.2.2 AOF文件过大带来的性能问题：

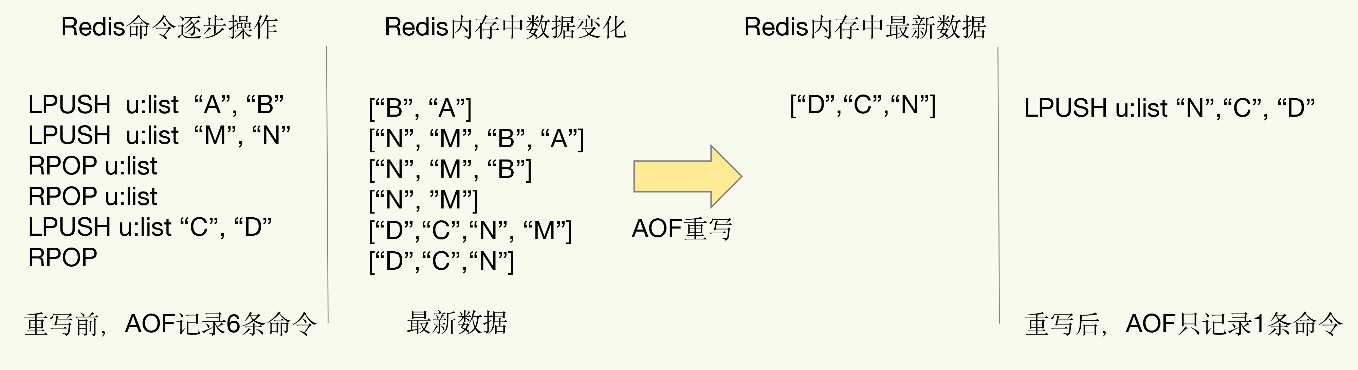
1. 文件系统本身对文件大小有限制，无法保存过大的文件。
2. 若文件过大，在往里面追加记录时，效率会变低。
3. 若发生宕机，AOF中记录将被逐个执行，用于故障恢复，若日志文件过大，整个恢复过程非常缓慢，这会影响到Redis的正常使用。

##### 2.4.3 AOF重写机制：目的是为避免日志文件过大

AOF重写机制是直接根据数据库的最新状态，生成这些数据的插入命令，作为新日志；这个过程由后台线程 bgrewriteof 来完成，避免对主线程的阻塞。

###### 2.4.3.1 AOF重写：多变一

* + 1. AOF重写机制就是在重写时，Redis根据数据库的现状创建一个新的AOF文件：读取数据库的所有键值对，然后对每一个键值对使用一条命令记录它的写入。
    2. AOF的重写机制可以把AOF日志文件变小，因为重写机制具有 多变一 的功能，也即是将旧日志文件中的多条命令，在重写后的新日志中变为一条命令。
    3. 在AOF文件中，是以追加的方式，逐一记录收到的写命令。当一个键值被多条写命令反复修改时，AOF文件会记录相应的多条命令。在重写时，会根据键值对的最新状态，为它生成对应的写入命令。因此，在重写时，一个键值对只要一条命令即可。在日志恢复时，只用执行该命令就可以完成该键值对的写入。[重写时将AOF的多条命令变为一条命令]



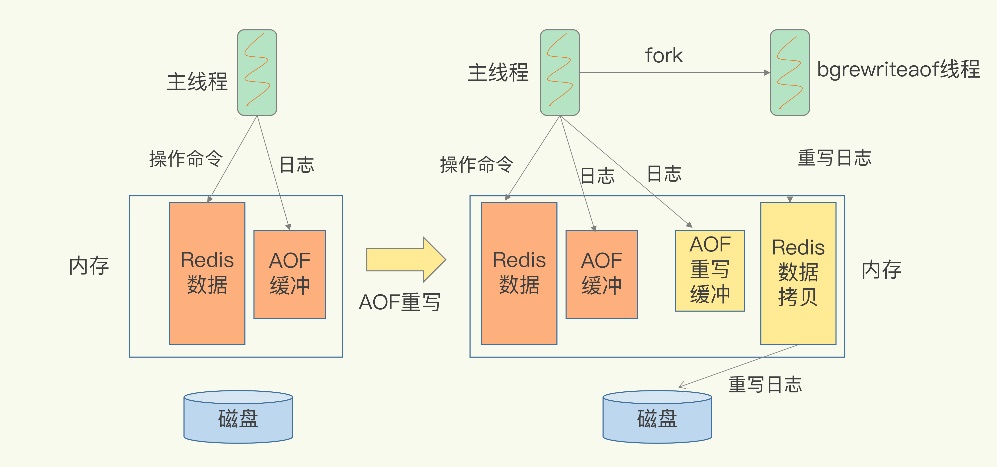
AOF重写减少日志大小

##### 2.4.4 AOF重写会阻塞吗？

不会，AOF日志由主线程写回，而重写过程是由后台线程 bgrewriteof 来完成，从而避免出现阻塞主线程，导致数据库性能下降的情况。

##### 2.4.4.1重写的过程： 一个拷贝，两处日志

1. 一个拷贝： 每次执行重写时，主线程fork出后台的 bgrewriteof 子进程， fork会把主线程的数据拷贝一份给 bgrewriteof子进程(fork时子进程会拷贝父进程的页表，即虚实映射关系，而不会拷贝物理内存，子进程复制父进程的页表，也能共享父进程的内存数据。)， 这里面包含了最新的数据， 然后， bgrewriteof 子进程就可以在不影响主线程的情况下，逐一把拷贝的数据写成操作，记入重写日志。
2. 两处日志：
   1. 第一处日志指的是正在使用的AOF日志，Redis会把写操作写入它的缓冲区，万一宕机，已有的AOF日志仍然是齐全的，可用于恢复。
   2. 第二处日志指的是AOF重写日志，AOF日志内容会被重写到AOF重写日志中，重写日志完成后，就可以用AOF文件替换旧日志文件。



AOF 非阻塞阻塞的重写过程

每次AOF重写时，Redis会先执行一个内存拷贝(拷贝的是页表)，用于重写；然后，使用两个日志保证在重写过程中，新写入的数据不会丢失。因为Redis采用的是额外的线程 bgwriteof 来完成数据的重写，所有该过程不会阻塞主线程。

落盘时机和重写机制都是在 记日志 过程中 发挥作用， 落盘时机的选择可以避免记录日志时阻塞主线程，重写可以避免日志文件过大。

使用AOF日志进行故障恢复时，则是在 用日志 ;需要将日志的所有操作都运行一遍，而Redis采用的是单线程设计，只能逐条记录进行执行，类似 重放 的过程，因而很慢。

#### 2.5 Redis持久化之 内存快照：

##### 2.5.1 内存快照RDB

* 1. 内存快照： 内存中的数据在某一时刻的状态记录(类似照片将一瞬间记录下来，相当于Linux的ps命令)。
  2. Redis的内存快照是将某一时刻的状态以文件的形式写入磁盘。这样即使宕机，快照文件也不会丢失，从而保证数据的可靠性。这个快照文件就称位RDB，其中RDB是Redis DataBase 的缩写。
  3. 与AOF 相比， RDB记录的是某一时刻的数据，而不是操作；因此在数据恢复是，直接把RDB读入文件，很快的完成恢复。
  4. 需要考虑的问题：

1. 对哪些数据做快照？这关系涉及到快照的执行效率。类似拍照的选景问题，把哪些人，那些物拍进去。
2. 做快照时，数据还能修改吗？这关系到Redis是否被阻塞，能否同时正常处理请求。类似按下快门前，提示大家不要乱动，否则拍出来的照片就模糊了。

##### 给哪些内存数据做快照？

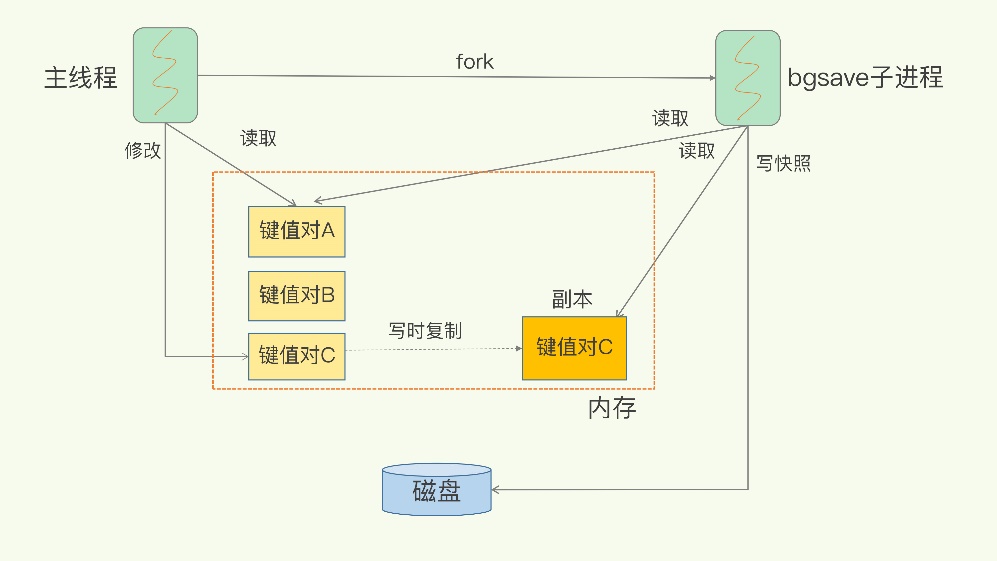
1. Redis的数据都保存在内存中，为所有数据的提供可靠性保证，因此执行的 **全量快照** ，也即是将内存中所有数据都记录到磁盘，。好处是一个性记录所有数据，一个都不少，不足之处就是全部写入内存比较耗时，开销大。
2. Redis灵魂之问： 它会阻塞主线程吗?这关系到是否会降Redis的性能。
3. 两个命令：
   1. save : 在主线程中执行，会导致阻塞。
   2. bgsave :创建一个子进程，专门用于写入RDB文件，避免了主线程的阻塞。因此bgsave 是 Redis RDB文件生成的默认配置。

bgsave 命令执行全量快照，既提供了数据的可靠性保证，也避免了对Redis性能的影响。

##### 快照时数据能够修改吗？

不能修改，但是可以通过写时复制技术来实现快照期间数据可修改。

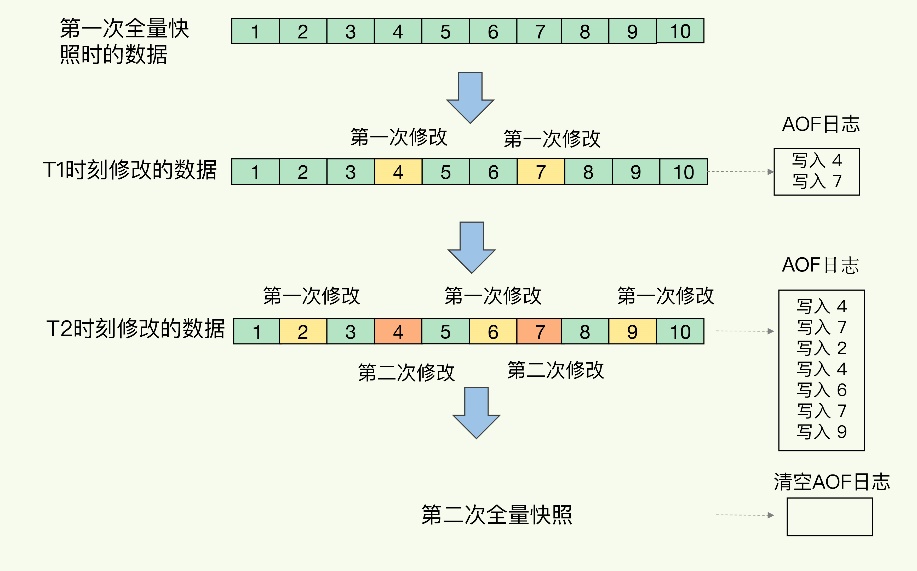
1. 但是不能修改会存在潜在的问题，会影响Redis的业务处理。
2. **避免阻塞和正常处理写操作并不是一回事**，为了保证快照的完整性，它只能处理读操作，因为不能修改正在执行的快数据。
3. 为了能够在执行快照过程中，正常处理写操作，Redis借助了操作系统提供的写时复制技术(Copy-On-Write,COW)。
4. bgsave子进程由主线程fork生成的，（通过拷贝主线程的页表）可以共享主线程的所有内存数据，bgsave子进程运行时，读取主线程的内存数据，并把它们写入RDB文件。
5. 若主线程也是读操作，则主线程和 bgsave子进程互不影响。
6. 若主线程要修改某一块数据，则这块数据会被复制一份，生成该数据的副本。bgsave子进程会把这个副本数据写入RDB文件，而主线程则可以直接修改原来的数据。

写时复制COW机制保证快照期间数据可修改

通过 写时复制COW机制 既保证了快照的完整性，也允许主线程同时对数据进行修改，避免了对正常业务的影响。

##### 多久做一次快照？可以每秒做一次快照吗？

1. 频繁地执行全量快照，会带来两方面的开销
   1. 频繁地将全量数据写入磁盘，会给磁盘带来很大的压力，多个快照竞争优先的磁盘带宽，前一个快照还未写完，后一个又开始了，导致恶行循环。
   2. bgsave子进程需要通过fork操作从主线程创建处理，虽然bgsave子进程在创建后不会阻塞主线程，但是 fork创建过程本身是会阻塞主线程的，主线程内存越大，阻塞越久。若频繁fork出bgsave子进程，会频繁阻塞主线程。
2. 可以采用 增量快照
   1. 增量快照指的是做了一次全量快照后，后续的快照只对修改的数据进行快照记录，从而避免每次全量快照的开销。
   2. 增量快照的前提是：需要记住哪些数据被修改了。但记录这些元数据信息，又需要额外的空间开销。
   3. 与AOF相比，快照RDB恢复速度快，但是频率不好把握。
      * 1. 频率太低，两次快照键一旦宕机，就可能丢失较多数据。
        2. 频率太高，会导致额外开销。
   4. Redis 4.0提出了一个 混合使用AOF日志和内存快照 的方法
      * 1. 内存快照以一定的频率执行执行。避免频繁执行快照带来的开销。
        2. 在两次快照之间，使用AOF日志记录这期间的所有命令操作。AOF日志只记录两次快照间的操作，而不是所有操作，避免出现日志文件过大的情况，同时避免重写开销。



内存快照RDB和AOF混合使用

RDB内存快照的优势在于，可以快速恢复数据库，即只需要把RDB文件读入内存内存，避免AOF需要顺序、逐一执行每条命令带来的低效性问题。

AOF 和 RDB 的选择

##### **AOF和RDB 的选择问题**

1. 若数据不能丢失，内存快照和AOF的混合使用比较合适
2. 若允许分钟级别的数据丢失，可以只使用RDB
3. 若只用AOF,优先使用 everysec 的配置选项， 因为它折中了可靠性和性能。

## 实践部分

## 加餐部分