Redis单线程：

主要是指网络IO和键值对读写是由一个线程完成的；其他功能，如持久化、异步删除、集群数据同步是由其他线程完成的

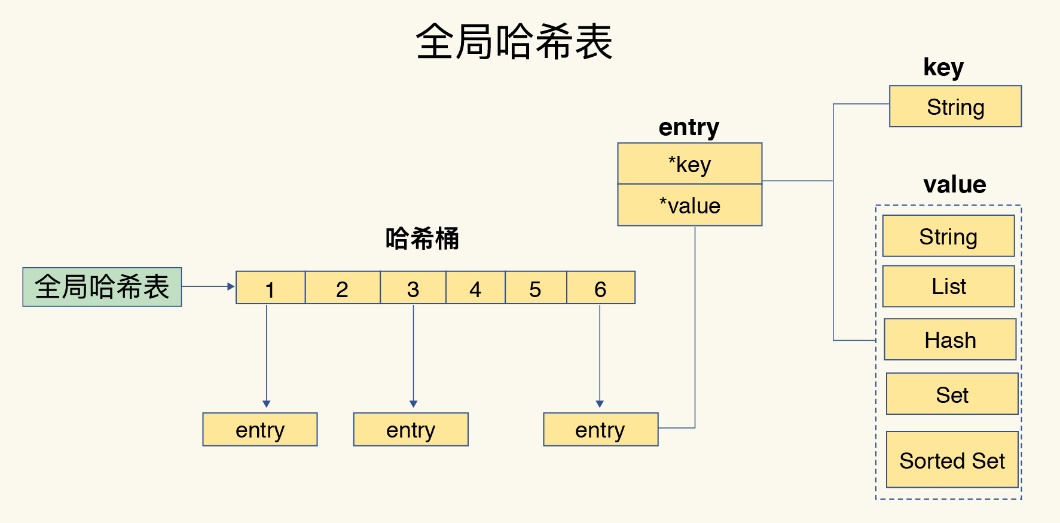
为什么使用单线程

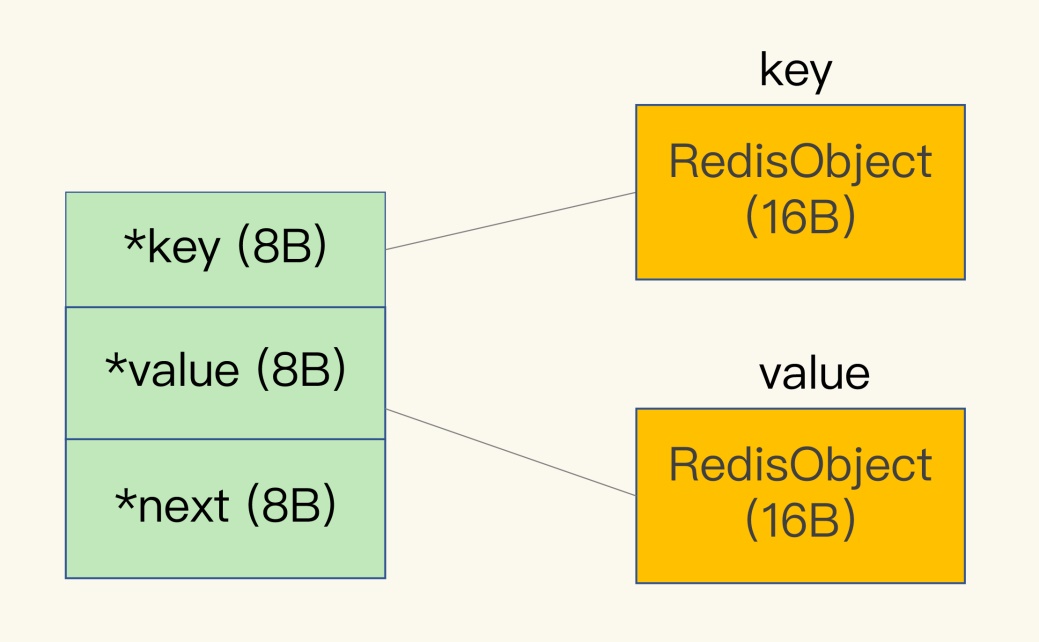
多线程编程模式存在共享数据并发访问控制问题，需要获取访问共享资源的互斥锁；

为什么这么快？

1. 数据结构

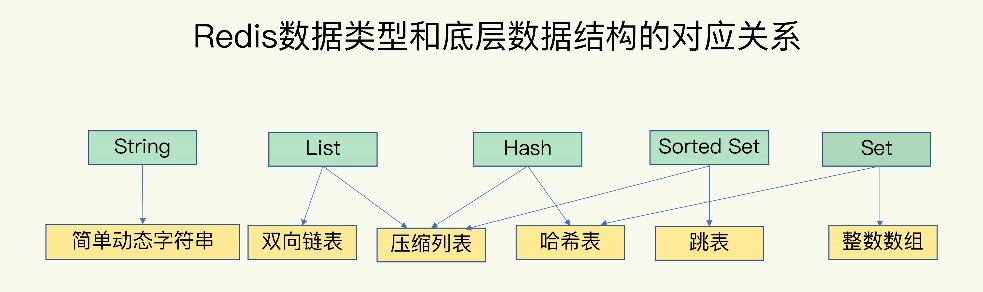
Redis使用**全局哈希表**保存所有键值对



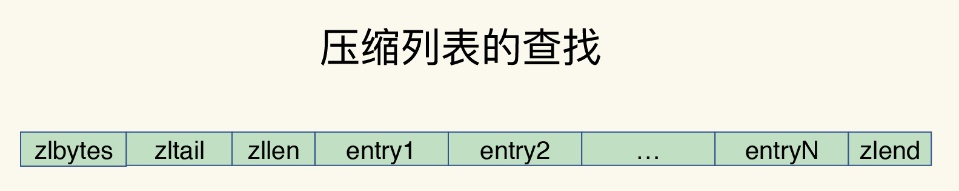


渐进式rehash：创建hash表2并分配2倍内存，将hash表1数据copy到hash表2中，释放hash表1；

在copy过程中，Redis仍然正常处理客户端请求，每处理一个请求时，就会处理一个索引下的所有entry到表2中；避免大量copy造成的开销，保证请求访问速



压缩列表



Zlbytes：列表长度

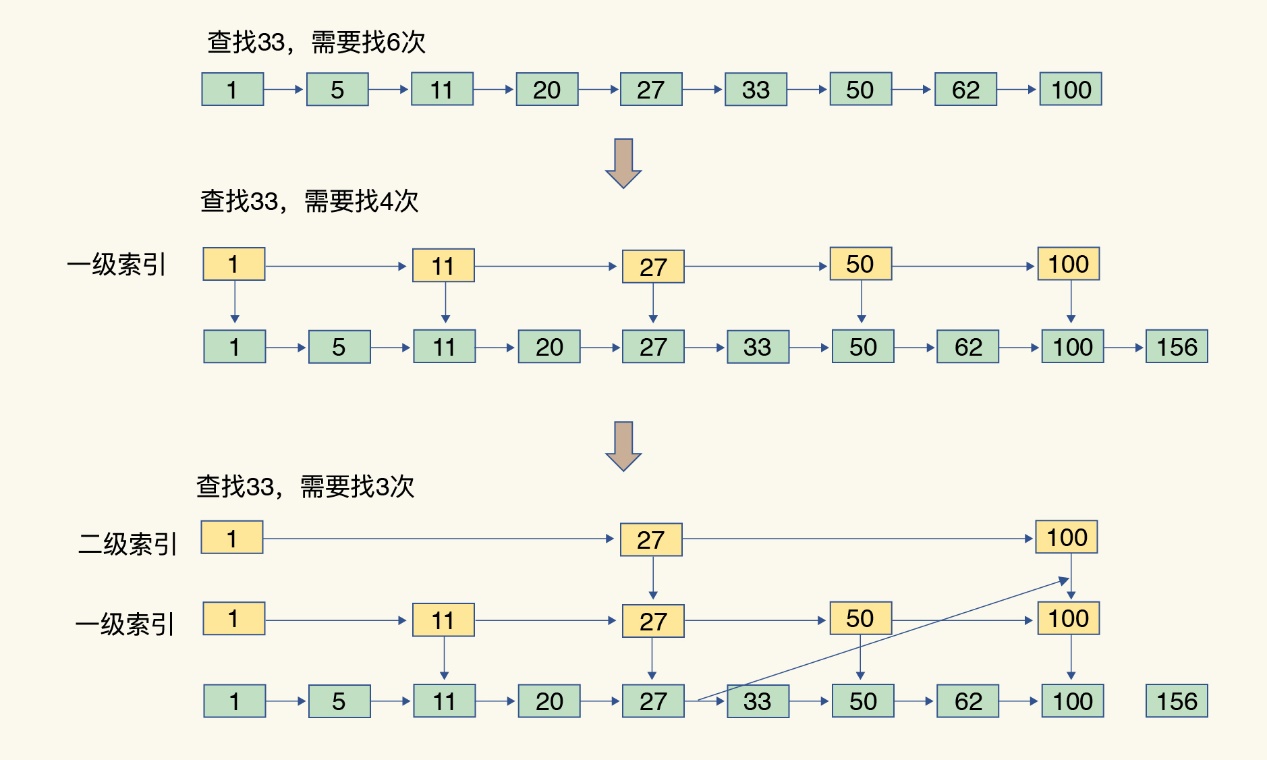
Zltail：列表尾部偏移量

Zllen：entry个数

Zlend：表示列表结束

优势：查找第一位元素和最后一个元素，可以通过表头信息直接O(1)查询

跳跃表：

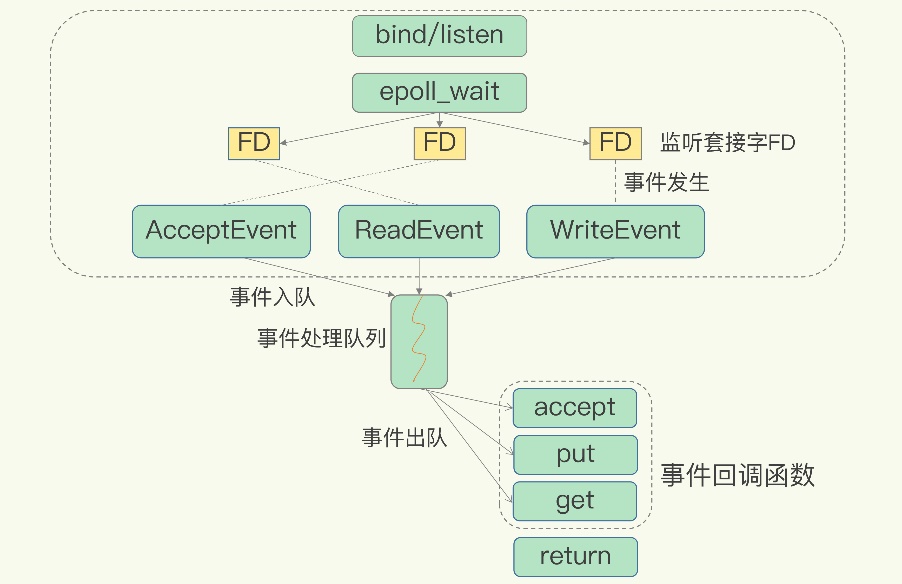


增加了多级索引，通过索引位置的几个跳转，实现数据快速定位



1. 多路复用机制（select/epoll）：在网络IO中能够并发处理大量的客户端请求，实现高吞吐率；该机制允许内核中，同时存在多个监听套接字和已连接套接字。内核会一直监听这些套接字上的连接请求或数据请求。Redis线程不会阻塞在某一个特定的监听或者套接字上（不会阻塞在某一客户端的请求处理上）；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）

优化点：避免accept/send/recv潜在的网络IO阻塞点

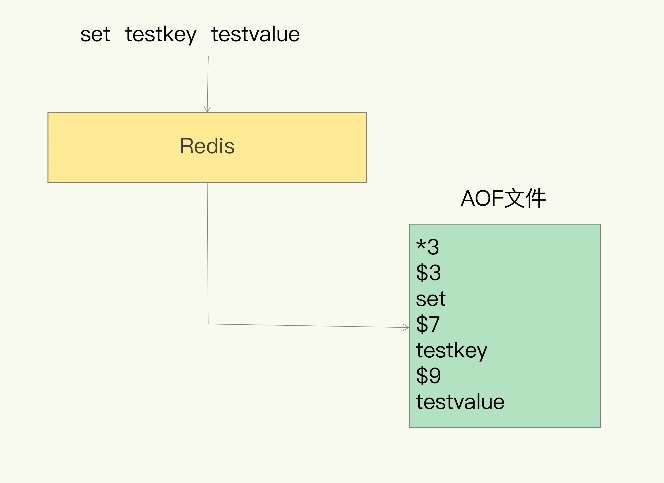


Redis持久方式

AOF

在主线程中执行

文本形式，记录内容是每一条命令



采用写后日志（与数据库相反）：

Redis先执行命令，把数据写入内存中，然后才写日志

原因：

1. 为了避免额外的开销，Redis写AOF时，不会做语法检查，所以如果写前日志的形式，日志中就可能会有错误的命令，导致无法使用；
2. 不会造成写操作的阻塞（数据直接存到了redis中）

风险点：

1. 刚执行完一个命令就宕机，造成aof日志缺失。
2. 因为Aof主线程中执行，如果磁盘写压力大，导致写磁盘慢，后续操作无法执行。

回写策略：控制一个写命令执行完后AOF日志写回磁盘的时机

AppendFsync可选值：

Always：同步回写，每次命令执行完，马上回写

Everysec：每秒回写，每个命令执行完，先把日志写到AOF的内存缓冲区，每个1s写入磁盘

No：操作系统控制，每个命令执行完，先把日志写到AOF的内存缓冲区，操作系统控制何时写入磁盘中



日志文件过大处理

AOF重写机制：在重写时，Redis针对现状，创建一个新AOF文件，读取所有数据库中键值对，写入新文件中；通过另一个文件，保证在此期间的写操作不会丢失

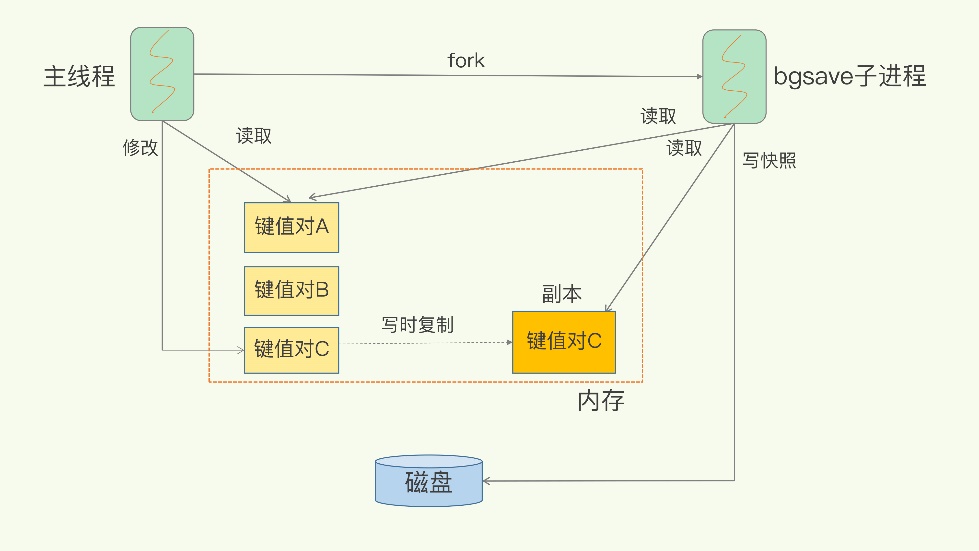
由后台子进程bgrewriteaof完成，避免阻塞主线程；

RDB：快照

Bgsave操作：创建一个子进程，专门用于写入RDB文件，避免阻塞主线程

写时复制（COW：copy on write），在执行快照同时，正常处理写操作。写时复制机制保证快照期间数据可修改

Bgsave子进程由主线程fork出来，可以共享主线程所有数据。



Redis4.0使用AOF和RDB混合方式。

**主从数据同步：**

一，全量同步：

**第一阶段**：主从库建立连接、协商同步过程，为全量同步做准备。从库给主库发送psync命令，表示要进行数据同步，主库根据psync命令启动复制。psync命令包含了主库的runId和复制进度的offset

runId：每个redis实例启动时自动生产的一个随机Id，第一次复制不知道则为？

offset：此时为-1， 因为全量

主库收到命令后，会用FULLRESTNC响应命令带上两个参数，主库runId和复制进度offset。

**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。

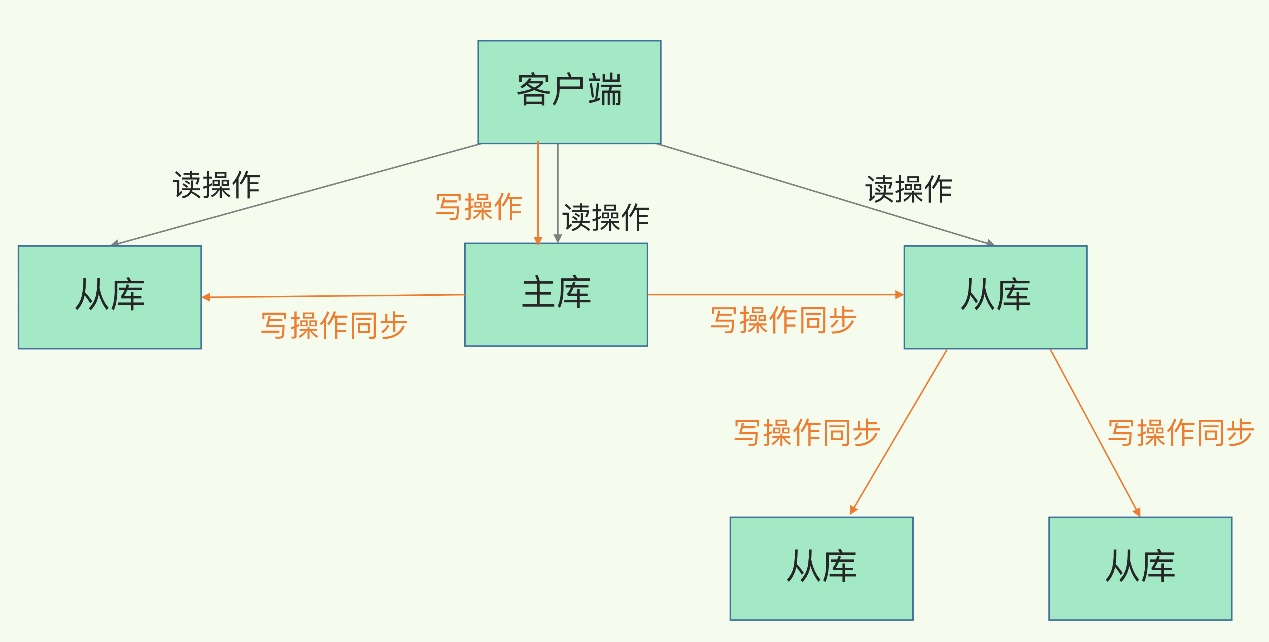
主库执行 bgsave 命令，生成 RDB 文件，接着将文件发给从库。从库接收到 RDB 文件后，会先清空当前数据库，然后加载 RDB 文件。这是因为从库在通过 replicaof 命令开始和主库同步前，可能保存了其他数据。为了避免之前数据的影响，从库需要先把当前数据库清空。

为了保证主从库的数据一致性，主库会在内存中用专门的 replication buffer，记录 RDB 文件生成后收到的所有写操作。

**第三阶段**：主库将replication buffer中的修改操作发给从库，从库再执行。

如果从库过多，主从数据同步时主库fork生成rdb时间过多，fork操作会阻塞主线程处理正常请求，如何分担主库压力？

解决：使用“主——从——从”模式



二，基于长连接的命令传播，可以避免频繁建立连接的开销

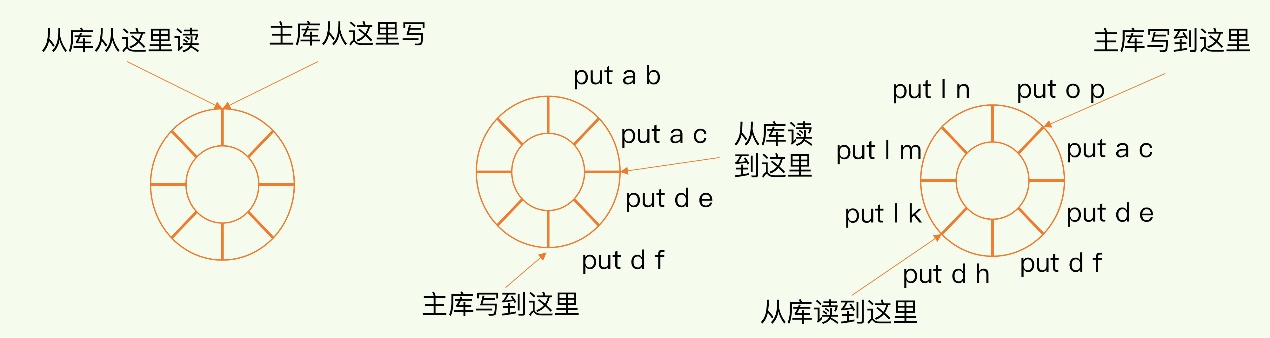
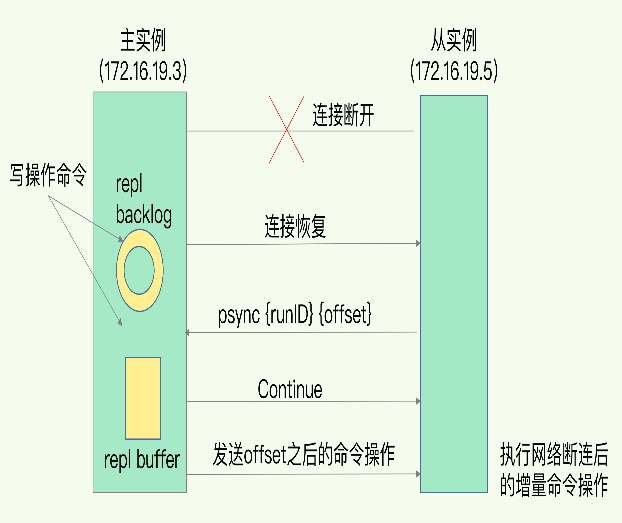
三，增量同步

Redis2.8后，网络断开，主从会采用增量复制的方式同步。

**repl\_backlog\_buffer 环形缓冲区：主库会记录自己写到的位置，从库会记录自己读到的位置**

主从断开连接后，主库会把期间的写操作命令，写入replication buffer中，同时也会写在repl\_backlog\_buffer中。

主从库的连接恢复之后，从库首先会给主库发送 psync 命令，并把自己当前的 slave\_repl\_offset 发给主库，主库会判断自己的 master\_repl\_offset 和 slave\_repl\_offset 之间的差距。在网络断连阶段，主库可能会收到新的写操作命令，所以，一般来说，master\_repl\_offset 会大于 slave\_repl\_offset。此时，主库只用把 master\_repl\_offset 和 slave\_repl\_offset 之间的命令操作同步给从库就行。



为避免主库写速度大于从库读速度，可调整参数repl\_backlog\_size，设置缓冲空间。

**哨兵机制：**

**监控**：哨兵进程在运行时，周期性给所有主从发送PING命令。如果**从库**没有在规定时间内响应，哨兵会把他标记为“下线状态”；同样，如果主库也没有响应，哨兵会判定主库下线，然后开始**自动切换主库**流程。

**通知**：哨兵把新主库的连接信息发给其他从库，让他们执行replicaof命令，和新主库建立连接，并进行数据复制。同时，哨兵会把新主库的链接信息通知给客户端，让他们把请求操作发送到新主库上。

**主库真的挂了吗？**

主观下线：哨兵会使用ping命令检测他自己和主从库的网络连接情况，用来判断实例的状态。如果响应超时，哨兵会先将从库标记为“主观下线”。若是主库需要哨兵集群来判断。当大多数的哨兵实例都判断主库已经“主观下线”，主库才会被标记为“客观下线”。同时这会触发主从切换。

客观下线：当有N个哨兵实例时，最好要有N/2+1个实例判断主库为“主观下线”，才能判定主库为“客观下线”。

**该选择哪个从库当主库？**

筛选+打分

1. 筛选：在选主时，除了要检查从库的在线状态，还要判断他之前的网络连接状态。

通过配置项：down-after-milliseconds \* 10（主从断连的最大连接超时时间），如果在此时间内（down-after-milliseconds）主从节点没有通过网络联系上，可认为主从断连。如果超过10次，说明从库网络状态不好，不适合做新主库。

1. 打分：
   1. 优先级，通过slave-priority设置优先级
   2. 和旧主库同步程度最接近的从库得分高：通过slave\_repl\_offset接近master\_repl\_offset判断
   3. ID号小的从库得分高

主从切换过程中，客户端能否正常进行请求操作？

如果客户端采取读写分离，那么读请求可以在从库上正常执行，不会有影响。但由于哨兵没有选出新主库，写请求会失败，失败持续时间=哨兵切换主从的时间+客户端感知新主库的时

如果不想让业务感知异常，客户端可以将写失败的请求缓存起来或写入消息队列中，等哨兵切换完主从后，再把这些请求发给新的主库。Ps：这种场景适合对写请求不敏感的业务，且业务层需要适配。

当哨兵完成主从切换后，客户端需要及时感知主库发生变化。具体方式如下：

1. 哨兵将新主库地址写去自己实例的pubsub中，客户端需要订阅此pubsub，当此pubsub有数据时，客户端就能感知主库发生变化。这种属于哨兵主动通知客户端。
2. 客户端访问主从库时，不能直接写死主从库地址，而是从哨兵集群中获取最新的地址（sentinel get-master-addr-by-name命令），当实例异常时，客户端可从哨兵集群中拿到最新的实例地址。
3. PS：一般Redis的SDK都提供了通过哨兵拿到实例地址，再访问实例的方式。

**哨兵集群**

**Redis性能**

IO性能

1. 任意一个请求在server中一旦耗时，就会影响整个server的性能，后面的请求要等待，包括：
   1. 操作bigkey：写入bigkey分配内存时需消耗更多时间，删除bigkey释放空间一样
   2. 使用复杂度过高的命令：sort/sunion/zunionsort，或者基数很大的范围查询lrange key
   3. 大量key集中过期：Redis过期机制在主线程中执行，执行任务可能会耗时在删除过期key的过程中
   4. 淘汰策略：Redis淘汰策略在主线程执行，当redis内存达到上限后，每次写入都需要删除一些过期key
   5. AOF刷盘开启always机制：写磁盘速度远比写内存慢
   6. 主从全量同步生成RDB：虽然采用了fork子进程生成快照数据，但fork一瞬间会造成整个线程阻塞，且实例越大，阻塞时间越长
2. 并发量非常大的时候

解决：

1. 业务人员规避，Redis4.0推出lazy-free机制，把bigkey释放内存放在异步线程处理
2. Redis在6.0推出了多线程（但我还没有用过）

Redis实例：

数据库实例不宜太大，几个GB级别合适。减少RDB文件生成，传输，和重新加载的开销。

1. redis阻塞点
   1. 和客户端交互时的阻塞点

IO慢，但是采用多路复用模型，避免主线程一直等待网络连接或请求到来，所以网络IO不是阻塞点；

**集合全量查询**和**聚合操作**，例如HGETALL，SMEMBERS

**删除操作**，本质是释放键值对占用的内存空间。Bigkey删除

**清空数据库**

* 1. 和磁盘交互的阻塞点

RDB文件生成、AOF日志重写采用redis子进程，不会成为阻塞点；

AOF日志同步写。一个同步写磁盘操作大约1~2ms，如果有大量数据操作需要记录在AOF中，并同步写，会阻塞。

* 1. 主从节点交互时阻塞

主从同步时，从库需要使用flushDB清空数据，删除大量key的阻塞点；

从库加载RDB文件到内存，RDB文件越大，加载越慢，成为阻塞点；

* 1. 切片集群

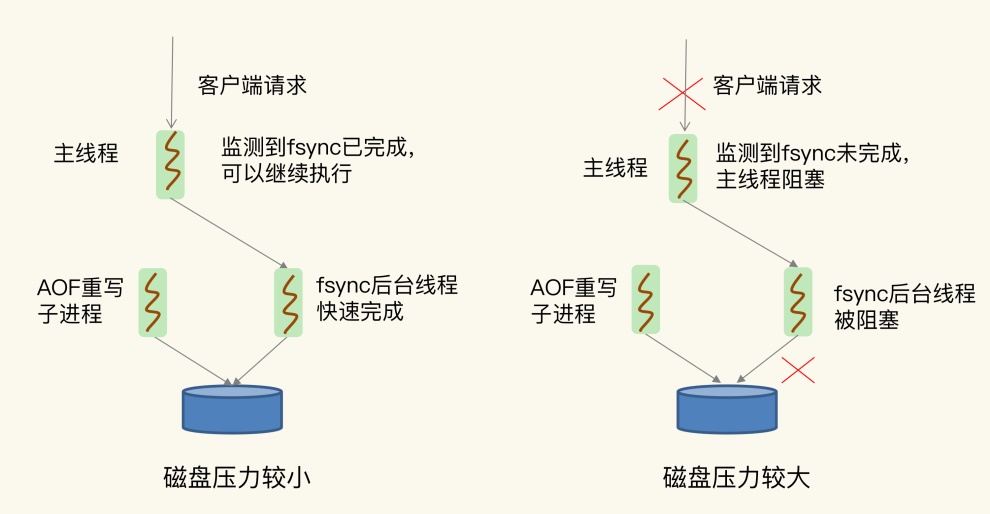
使用Redis Cluster方案，迁移bigkey，会造成阻塞；

1. Redis变慢
   1. 自身操作影响
      1. 慢查询
         1. 用其他高效的命令代替。比如，查询set集合，不要使用SMEMBERS，而是使用SSCAN多次迭代查询，避免一次返回大量数据，造成线程阻塞
         2. 在客户端（内存）完成排序，交集等操作，不要使用Redis命令。
      2. 过期key操作

Redis自动删除过期key机制，会引起Redis操作阻塞，导致性能变慢。默认每100ms会删除一些过期key。（Redis 4.0 后可以用异步线程机制来减少阻塞影响）

* + - 1. 使用EXPIREAT设置过期时间的时候，看是否很多key使用同样的UNIX 时间戳
  1. 文件系统
     1. AOF模式
        1. 当写回策略配置为 everysec 时，Redis 会使用后台的子线程异步完成 fsync 的操作。
        2. always 策略并不使用后台子线程来执行
        3. AOF 重写会对磁盘进行大量 IO 操作，fsync 又需要等到数据写到磁盘后才能返回。当 AOF 重写的压力比较大时，就会导致 fsync 被阻塞。虽然 fsync 是由后台子线程负责执行的，但是，主线程会监控 fsync 的执行进度





* 1. 操作系统
     1. Swap

内存 swap 是操作系统里将内存数据在内存和磁盘间来回换入和换出的机制，涉及到磁盘的读写，所以，一旦触发 swap，无论是被换入数据的进程，还是被换出数据的进程，其性能都会受到慢速磁盘读写的影响。

swap 触发后影响的是 Redis 主 IO 线程，这会极大地增加 Redis 的响应时间。

触发Swap原因（物理机器内存不足）：

·Redis实例自身使用了大量的内存，导致物理机器的可用内存不足

·和 Redis 实例在同一台机器上运行的其他进程，在进行大量的文件读写操作。文件读写本身会占用系统内存，这会导致分配给 Redis 实例的内存量变少，进而触发 Redis 发生 swap。

* + 1. 内存大页

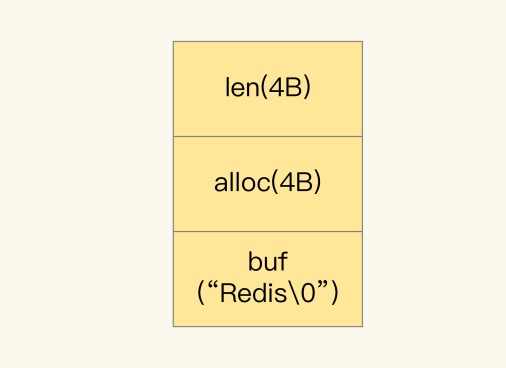
Linux 内核从 2.6.38 开始支持内存大页机制，该机制支持 2MB 大小的内存页分配，而常规的内存页分配是按 4KB 的粒度来执行的。

解决方式：关闭内存大页

**String类型**

短板：消耗内存空间较多

String除了消耗内存记录实际数据，还要额外的空间记录**数据长度，空间使用等 元数据**，当保存实际数据较小的时候，元数据的开销显得比较大。



简单动态字符串 sds结构

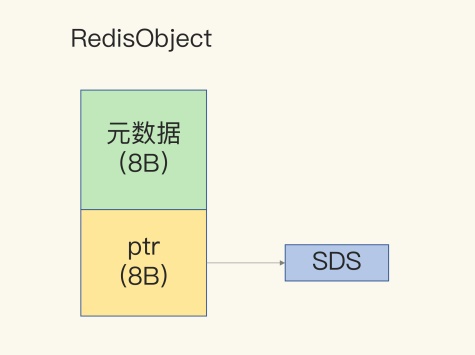
Buf：字节数组，保存实际数据。为了表示数组结束，redis会自动在后添加“\0”，额外占用一个字节。

Len：占4字节，表示buf用的长度。

Alloc：占4字节，表示buf实际分配的长度，一般大于len

Len和Alloc占用SDS的额外开销

RedisObject开销



RedisObject结构

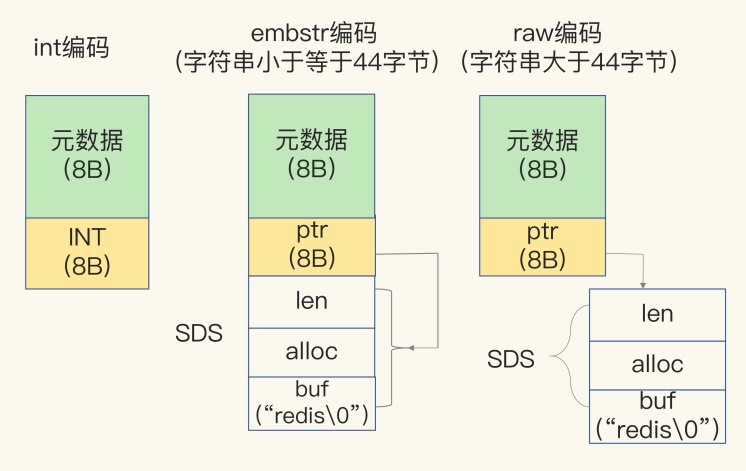
元数据：占8字节

指针：8字节；例如，指向String类型的sds结构所在的内存地址

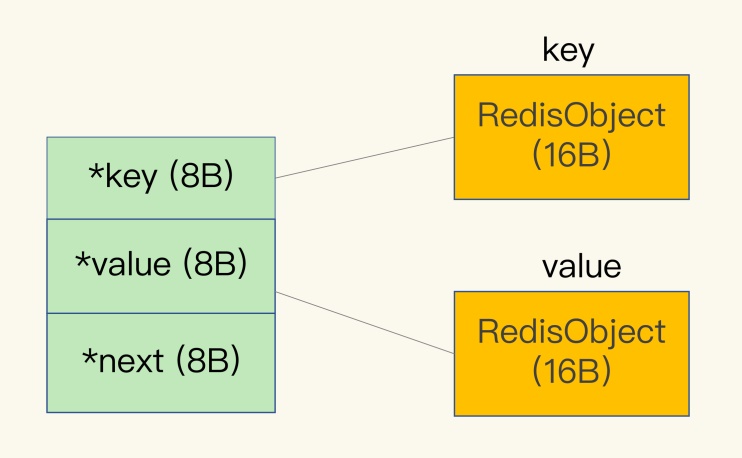
优化：当保存的是Long型整数，指针直接赋值为整数数据，这样就不用额外的指针指向整数了，节省指针的开销。

当保存字符串时，如果字符串小于44字节时，RedisObject种的元数据、指针和SDS是在一片连续的内存区域，避免内存碎片。emdstr编码方式。

当超过44字节时，会给sds分配独立空间，并用指针指向sds结构。Raw编码方式。



编码方式



Redis全局hash表每一项的dictEntry结构体

Redis内存分配库jemalloc：分配内存会分配2的幂次数的内存空间，这样可以减少频繁分配内存空间的次数。

**用什么数据结构可以节省内存？**

**压缩列表（ziplist）**

**Hash类型底层结构：**压缩列表和Hash表。为了能充分使用压缩列表的精简内存布局，我们一般要控制保存在 Hash 集合中的元素个数

Hash 类型设置了用压缩列表保存数据时的两个阈值，一旦超过了阈值，Hash 类型就会用哈希表来保存数据了。

这两个阈值分别对应以下两个配置项：

hash-max-ziplist-entries：表示用压缩列表保存时哈希集合中的最大元素个数。

hash-max-ziplist-value：表示用压缩列表保存时哈希集合中单个元素的最大长度。

