链滴

Redis 笔记一

作者：matthewhan

原文链接：https://ld246.com/article/1625797312179

来源网站：链滴

<h2 id="Q-Redis-变慢的原因一">Q：Redis 变慢的原因一</h2>

但是，这里依然存在一个问题，哈希冲突链上的元素只能通过指针逐一查找再操作。如果哈希表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ntry 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。那具体么做呢？
其实，为了使 rehash 操作更高效，Redis 默认使用了两个全局哈希表：哈希表 1 和哈希表 2。开始，当你刚插入数据时，默认使用哈希表 1，此时的哈希表 2 并没有被分配空间。随着数据逐步增，Redis 开始执行 rehash，这个过程分为三步：
给哈希表 2 分配更大的空间，例如是当前哈希表 1 大小的两倍；
把哈希表 1 中的数据重新映射并拷贝到哈希表 2 中；
释放哈希表 1 的空间。
到此，我们就可以从哈希表 1 切换到哈希表 2，用增大的哈希表 2 保存更多数据，而原来的哈希 1 留作下一次 rehash 扩容备用。
<h2 id="Q-渐进式-rehash-拷贝">Q：渐进式 rehash 拷贝</h2>

简单来说就是在第二步拷贝数据时，Redis 仍然正常处理客户端请求，每处理一个请求时，从哈表 1 中的第一个索引位置开始，顺带着将这个索引位置上的所有 entries 拷贝到哈希表 2 中；等处理一个请求时，再顺带拷贝哈希表 1 中的下一个索引位置的 entries。
因为在进行渐进式 rehash 的过程中， 字典会同时使用 ht[0] 和 ht[1] 两个哈希表， 所以在渐进 rehash 进行期间， 字典的删除（delete）、查找（find）、更新（update）等操作会在两个哈希表进行： 比如说， 要在字典里面查找一个键的话， 程序会先在 ht[0] 里面进行查找， 如果没找到的话，就会继续到 ht[1] 里面进行查找， 诸如此类。
另外， 在渐进式 rehash 执行期间， 新添加到字典的键值对一律会被保存到 ht[1] 里面， 而 ht[0 则不再进行任何添加操作： 这一措施保证了 ht[0] 包含的键值对数量会只减不增， 并随着 rehash 作的执行而最终变成空表。
<h2 id="Q-Redis-的几种数据结构">Q：Redis 的几种数据结构</h2>

大类型下对应多种实现转换规则是基于一个 key 的数据大小和元素个数，配置文件中配。
<h2 id="Q-为什么说-Redis-单线程这么快">Q：为什么说 Redis 单线程这么快</h2>
我们通常说，Redis 是单线程，主要是指 Redis 的网络 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流程。但 Redis 的其他功能，比如持久化、异步删除、集数据同步等，其实是由额外的线程执行的。
多线程的坏处：
<ol start="0">
为了保证共享资源的正确性，就需要有额外的机制进行保证，而这个额外的机制，就会带来额外开销。
并发访问控制也是难点，降低系统代码的易调试性和可维护性

单线程快的原因：
<ol start="0">
高效的数据结构（哈希表、跳表）
多路复用机制

<h3 id="什么是多路复用机制-">什么是多路复用机制？</h3>
基本 IO 模型与阻塞点：以 Get 请求为例，为了处理一个 Get 请求，需要监听客户端请求（bind/isten），和客户端建立连接（accept），从 socket 中读取请求（recv），解析客户端发送请求（pare），根据请求类型读取键值数据（get），最后给客户端返回结果，即向 socket 中写回数据（send。
但是，在这里的网络 IO 操作中，有潜在的阻塞点，分别是 accept() 和 recv()。当 Redis 监听到个客户端有连接请求，但一直未能成功建立起连接时，会阻塞在 accept() 函数这里，导致其他客户无法和 Redis 建立连接。类似的，当 Redis 通过 recv() 从一个客户端读取数据时，如果数据一直没到达，Redis 也会一直阻塞在 recv()。
这就导致 Redis 整个线程阻塞，无法处理其他客户端请求，效率很低。不过，幸运的是，socket 网络模型本身支持非阻塞模式 。
针对监听套接字，我们可以设置非阻塞模式：当 Redis 调用 accept() 但一直未有连接请求到达，Redis 线程可以返回处理其他操作，而不用一直等待。但是，你要注意的是，调用 accept() 时，已存在监听套接字了。
Linux 中的 IO 多路复用机制是指一个线程处理多个 IO 流，就是我们经常听到的 select/epoll 制。简单来说，在 Redis 只运行单线程的情况下，该机制允许内核中，同时存在多个监听套接字和已接套接字。内核会一直监听这些套接字上的连接请求或数据请求。一旦有请求到达，就会交给 Redis 程处理，这就实现了一个 Redis 线程处理多个 IO 流的效果。
也就是说，不会阻塞在某一个特定的客户端请求处理上。正因为此，Redis 可以同时和多个客户连接并处理请求，从而提升并发性。
现在，我们知道了，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。
<h2 id="Q-AOF">Q：AOF</h2>
AOF 写日志是主线程发起，在命令执行后才记录日志，所以不会阻塞当前的写操作。
其次，AOF 虽然避免了对当前命令的阻塞，但可能会给下一个操作带来阻塞风险。这是因为，AF 日志也是在主线程中执行的，如果在把日志文件写入磁盘时，磁盘写压力大，就会导致写盘很慢，而导致后续的操作也无法执行了。
<h3 id="三种写回策略">三种写回策略</h3>
也就是 AOF 配置项 appendfsync 的三个可选值。

Always，同步写回：每个写命令执行完，立马同步地将日志写回磁盘；

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中的内容写入磁盘；
No，操作系统控制的写回：每个写命令执行完，只是先把日志写到 AOF 文件的内存冲区，由操作系统决定何时将缓冲区内容写回磁盘。

我们一定要小心 AOF 文件过大带来的性能问题。如何解决？利用 AOF 重写机制</strng>
重写机制具有“多变一”功能。所谓的“多变一”，也就是说，旧日志文件中的多条令，在重写后的新日志中变成了一条命令。

<h3 id="AOF-重写会阻塞吗-">AOF 重写会阻塞吗？</h3>
和 AOF 日志由主线程写回不同，重写过程是由后台子进程 bgrewriteaof 来完成的，这也是为避免阻塞主线程，导致数据库性能下降。
我把重写的过程总结为“一个拷贝，两处日志”。
“一个拷贝”就是指，每次执行重写时，主线程 fork 出后台的 bgrewriteaof 子进程。此时，for 会把主线程的内存拷贝一份给 bgrewriteaof 子进程，这里面就包含了数据库的最新数据。然后，bgrwriteaof 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。
“两处日志”又是什么呢？
因为主线程未阻塞，仍然可以处理新来的操作。此时，如果有写操作，第一处日志就是指正在使

的 AOF 日志，Redis 会把这个操作写到它的缓冲区。这样一来，即使宕机了，这个 AOF 日志的操作然是齐全的，可以用于恢复。
而第二处日志，就是指新的 AOF 重写日志。这个操作也会被写到重写日志的缓冲区。这样，重日志也不会丢失最新的操作。等到拷贝数据的所有操作记录重写完成后，重写日志记录的这些最新操也会写入新的 AOF 文件，以保证数据库最新状态的记录。此时，我们就可以用新的 AOF 文件替代旧件了。
主线程 fork 子线程的一瞬间是会发生阻塞的

<h2 id="Q-RDB-Redis-DataBase--快照">Q：RDB（Redis DataBase） 快照</h2>
<h3 id="RDB-重写会阻塞吗-如何做快照-">RDB 重写会阻塞吗？如何做快照？</h3>
Redis 提供了两个命令来生成 RDB 文件，分别是 save 和 bgsave。

save：在主线程中执行，会导致阻塞；
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，这也是 Redis RDB 文件生成的默认配置。

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。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，生成该据的副本（键值对 C’）。然后，主线程在这个数据副本上进行修改。同时，bgsave 子进程可以继把原来的数据（键值对 C）写入 RDB 文件。

<blockquote>
这里对数据的快照，我觉得很妙。
当存在在「快照备份的过程」中（T 时刻），如果是读操作无所谓，写操作的话，需先将该数据复制一份副本，然后主线程在该副本上进行修改，这样子线程存储的数据则是「老」的数，这样的话，在「快照备份的过程中」能保证快照的数据都是 T 时刻的数据了，无论该过程结束的时是 T + N 秒。
</blockquote>
<h3 id="多久做快照-">多久做快照？</h3>
虽然 bgsave 执行时不阻塞主线程，但是，如果频繁地执行全量快照，也会带来两方面的开销。
一方面，频繁将全量数据写入磁盘，会给磁盘带来很大压力，多个快照竞争有限的磁盘带宽，前个快照还没有做完，后一个又开始做了，容易造成恶性循环。
另一方面，bgsave 子进程需要通过 fork 操作从主线程创建出来。虽然，子进程在创建后不会再塞主线程，但是，fork 这个创建过程本身会阻塞主线程，而且主线程的内存越大，阻塞时间越长。如频繁 fork 出 bgsave 子进程，这就会频繁阻塞主线程了（所以，在 Redis 中如果有一个 bgsave 在行，就不会再启动第二个 bgsave 子进程）
<blockquote>
原文中「所以，在 Redis 中如果有一个 bgsave 在运行，就不会再启动第二个 bgsave 子进程」指的是 Redis 实际的机制，并不会出现多个 bgsave 子进程来用于快照。
</blockquote>
优化
Redis 4.0 中提出了一个混合使用 AOF 日志和内存快照的方法。简单来说，内存快照以一定的频执行，在两次快照之间，使用 AOF 日志记录这期间的所有命令操作。混
合使用 RDB 和 AOF，正好可以取两者之长，避两者之短，以较小的性能开销保证数据可靠性和能。

<h3 id="关于混合使用个人感想">关于混合使用个人感想</h3>
关于 AOF 和 RDB，AOF 有点像一个软件的小版本升级，version 1.1 ==> version 1.2，可能只有几条数据的更新，此时采用 AOF 进行数据更新比较好，但是到了一个大的版本，比如 ersion 3.8，此时最好重新下一个最新版本的客户端了，所以对于 Redis 来说利用 RDB 比较好。</st ong>
很多人要说了，诶？AOF 不是会重写操作记录实现 All In One 吗？那这样理解的话 无论数据怎么更新、增加、删除， AOF 的操作都应该比 RDB 这种全量的少啊？我看评论区 Kaito 的 言：
<blockquote>
RDB 文件内容是经过压缩的二进制数据（不同数据类型数据做了针对性优化），文件 小。
</blockquote>
觉得应该是如果一段时间后数据几乎全改了，此时的 AOF 文件一定是比 RDB 大的， 复速度肯定也比不上 RDB。而且删除操作多的话，RDB 更占优势，AOF 应该会记录删除的操作记录
[poll1625797274157]