日常开发中，我们使用Redis存储Key时通常会设置一个过期时间,但是Redis是怎么删除过期的key，而且Redis是单线程的，删除key会不会造成阻塞。要搞清楚这些，就要了解Redis的过期策略和内存淘汰机制。

Redis采用的是定期删除 + 懒惰删除策略

#### 定期删除策略

Redis会将每个设置了过期时间的key放入到一个独立的字典中，默认每100ms进行一次过期扫描：

1. 随机抽取20个key
2. 删除这20个key中过期的key
3. 如果过期的key比例超过1/4，就是重复步骤1，继续删除。

**为什么不扫描所有的key？**

Redis是单线程，全部扫描岂不是卡死了。而且为了防止每次扫描过期的key比例都超过1/4，导致不停循环卡死线程，Redis每次扫描添加了上限时间，默认是25ms。

如果客户端将超时时间设置的比较短，比如10ms，那么就会出现大量的链接因为超时而关闭，业务端就会出现很多异常。而且这时你还无法从Redis的slowlog中看到慢查询记录，因为慢查询指的是逻辑处理福欧恒慢，不包含等待时间。

如果在同一时间出现大面积key过期，Redis循环多次扫描过期词典，直到过期的key比例小于1/4。这会导致卡顿，而且在高并发道德情况下，可能会导致缓存雪崩。

**为什么Redis为每次扫描添的上限时间是25ms，还会出现上面的情况？**

因为Redis是单线程，每个请求处理都需要排队，而且由于Redis每次扫描都是25ms，也就是每个请求最多25ms，100个请求就是2500ms。

如果有大批量的key过期，要给过期时间设置一个随机范围，而不宜全部在同一时间过期、分散过期处理的压力。

#### 从库的过期策略

从库不会进行过期扫描，从库对过期的处理是被动的。主库在key到期时，会在AOF文件里增加一条del指令，同步到所有的从库，从库通过执行这条del指令来删除过期的key。

因为指令同步是异步进行的，所以主库过期的key的del指令没有及时同步到从库的话，会出现主从数据的不一致，主库没有的数据在从库里还存在。

#### 懒惰删除策略

**Redis为什么要懒惰删除（lazy free）？**

删除指令del会直接释放对象的内存，大部分情况，这个指令非常快，没有明显延迟。不过如果删除的key是一个非常大的对象，比如一个包含了千万元素的hash，又或者在使用FLUSHDB和FLUSHALL删除包含大量键的数据库时，那么删除操作就会导致单线程卡顿。

redis 4.0引入了lazyfree的机制，它可以将删除键或者数据库的操作放在后台线程里执行，从而尽可能地避免服务器阻塞。

#### unlink

unlink指令，它能对删除操作进行懒处理，丢给后台线程来异步回收内存。

> unlink key

OK

#### flush

flushdb 和 flushall 指令，用来清空数据库，这也是极其缓慢的操作。Redis 4.0 同样给这两个指令也带来了异步化，在指令后面增加 async 参数就可以将整棵大树连根拔起，扔给后台线程慢慢焚烧。

> flushall async

OK

#### 异步队列

主线程将对象的引用从「大树」中摘除后，会将这个 key 的内存回收操作包装成一个任务，塞进异步任务队列，后台线程会从这个异步队列中取任务。任务队列被主线程和异步线程同时操作，所以必须是一个线程安全的队列。

不是所有的 unlink 操作都会延后处理，如果对应 key 所占用的内存很小，延后处理就没有必要了，这时候 Redis 会将对应的 key 内存立即回收，跟 del 指令一样。

#### 更多异步删除点

Redis 回收内存除了 del 指令和 flush 之外，还会存在于在 key 的过期、LRU 淘汰、rename 指令以及从库全量同步时接受完 rdb 文件后会立即进行的 flush 操作。

Redis4.0 为这些删除点也带来了异步删除机制，打开这些点需要额外的配置选项。

* slave-lazy-flush 从库接受完 rdb 文件后的 flush 操作
* lazyfree-lazy-eviction 内存达到 maxmemory 时进行淘汰
* lazyfree-lazy-expire key 过期删除
* lazyfree-lazy-server-del rename 指令删除 destKey

#### 内存淘汰机制

Redis 的内存占用会越来越高。Redis 为了限制最大使用内存，提供了 redis.conf 中的配置参数 maxmemory。当内存超出 maxmemory，Redis 提供了几种内存淘汰机制让用户选择，配置 maxmemory-policy：

* noeviction：当内存超出 maxmemory，写入请求会报错，但是删除和读请求可以继续。（使用这个策略，疯了吧）
* allkeys-lru：当内存超出 maxmemory，在所有的 key 中，移除最少使用的key。只把 Redis 既当缓存是使用这种策略。（推荐）。
* allkeys-random：当内存超出 maxmemory，在所有的 key 中，随机移除某个 key。（应该没人用吧）
* volatile-lru：当内存超出 maxmemory，在设置了过期时间 key 的字典中，移除最少使用的 key。把 Redis 既当缓存，又做持久化的时候使用这种策略。
* volatile-random：当内存超出 maxmemory，在设置了过期时间 key 的字典中，随机移除某个key。
* volatile-ttl：当内存超出 maxmemory，在设置了过期时间 key 的字典中，优先移除 ttl 小的。

#### LRU算法

实现LRU算法除了需要key/value字典外，还需要附加一个链表，链表中的元素按照一定的顺序进行排列。当空间满的时候，会踢掉链表尾部的元素。当字典的某个元素被访问时，它在链表中的位置会被移动到表头。所以链表的元素排列顺序就是元素最近被访问的时间顺序。

使用Python的OrderedDict（双向链表+字典）来实现一个简单的LRU算法：

#### 近似LRU算法

Redis 使用的并不是完全 LRU 算法。不使用 LRU 算法，是为了节省内存，Redis 采用的是随机LRU算法，Redis 为每一个 key 增加了一个24 bit的字段，用来记录这个 key 最后一次被访问的时间戳。

注意 Redis 的 LRU 淘汰策略是懒惰处理，也就是不会主动执行淘汰策略，当 Redis 执行写操作时，发现内存超出 maxmemory，就会执行 LRU 淘汰算法。这个算法就是随机采样出5(默认值)个 key，然后移除最旧的 key，如果移除后内存还是超出 maxmemory，那就继续随机采样淘汰，直到内存低于 maxmemory 为止。

如何采样就是看 maxmemory-policy 的配置，如果是 allkeys 就是从所有的 key 字典中随机，如果是 volatile 就从带过期时间的 key 字典中随机。每次采样多少个 key 看的是 maxmemory\_samples 的配置，默认为 5。

#### LFU

Redis 4.0 里引入了一个新的淘汰策略 —— LFU（Least Frequently Used） 模式，作者认为它比 LRU 更加优秀。

LFU 表示按最近的访问频率进行淘汰，它比 LRU 更加精准地表示了一个 key 被访问的热度。

如果一个 key 长时间不被访问，只是刚刚偶然被用户访问了一下，那么在使用 LRU 算法下它是不容易被淘汰的，因为 LRU 算法认为当前这个 key 是很热的。而 LFU 是需要追踪最近一段时间的访问频率，如果某个 key 只是偶然被访问一次是不足以变得很热的，它需要在近期一段时间内被访问很多次才有机会被认为很热。

**Redis对象的热度**

Redis 的所有对象结构头中都有一个 24bit 的字段，这个字段用来记录对象的热度。

// redis 的对象头

typedef struct redisObject {

unsigned type:4; // 对象类型如 zset/set/hash 等等

unsigned encoding:4; // 对象编码如 ziplist/intset/skiplist 等等

unsigned lru:24; // 对象的「热度」

int refcount; // 引用计数

void \*ptr; // 对象的 body

} robj;

**LRU模式**

在 LRU 模式下，lru 字段存储的是 Redis 时钟 server.lruclock，Redis 时钟是一个 24bit 的整数，默认是 Unix 时间戳对 2^24 取模的结果，大约 97 天清零一次。当某个 key 被访问一次，它的对象头的 lru 字段值就会被更新为 server.lruclock。

**LFU模式**

在 LFU 模式下，lru 字段 24 个 bit 用来存储两个值，分别是 ldt(last decrement time) 和 logc(logistic counter)。

logc 是 8 个 bit，用来存储访问频次，因为 8 个 bit 能表示的最大整数值为 255，存储频次肯定远远不够，所以这 8 个 bit 存储的是频次的对数值，并且这个值还会随时间衰减。如果它的值比较小，那么就很容易被回收。为了确保新创建的对象不被回收，新对象的这 8 个 bit 会初始化为一个大于零的值，默认是 LFU\_INIT\_VAL=5。

ldt 是 16 个位，用来存储上一次 logc 的更新时间，因为只有 16 位，所以精度不可能很高。它取的是分钟时间戳对 2^16 进行取模，大约每隔 45 天就会折返。

同 LRU 模式一样，我们也可以使用这个逻辑计算出对象的空闲时间，只不过精度是分钟级别的。图中的 server.unixtime 是当前 redis 记录的系统时间戳，和 server.lruclock 一样，它也是每毫秒更新一次。