1、过期删除策略

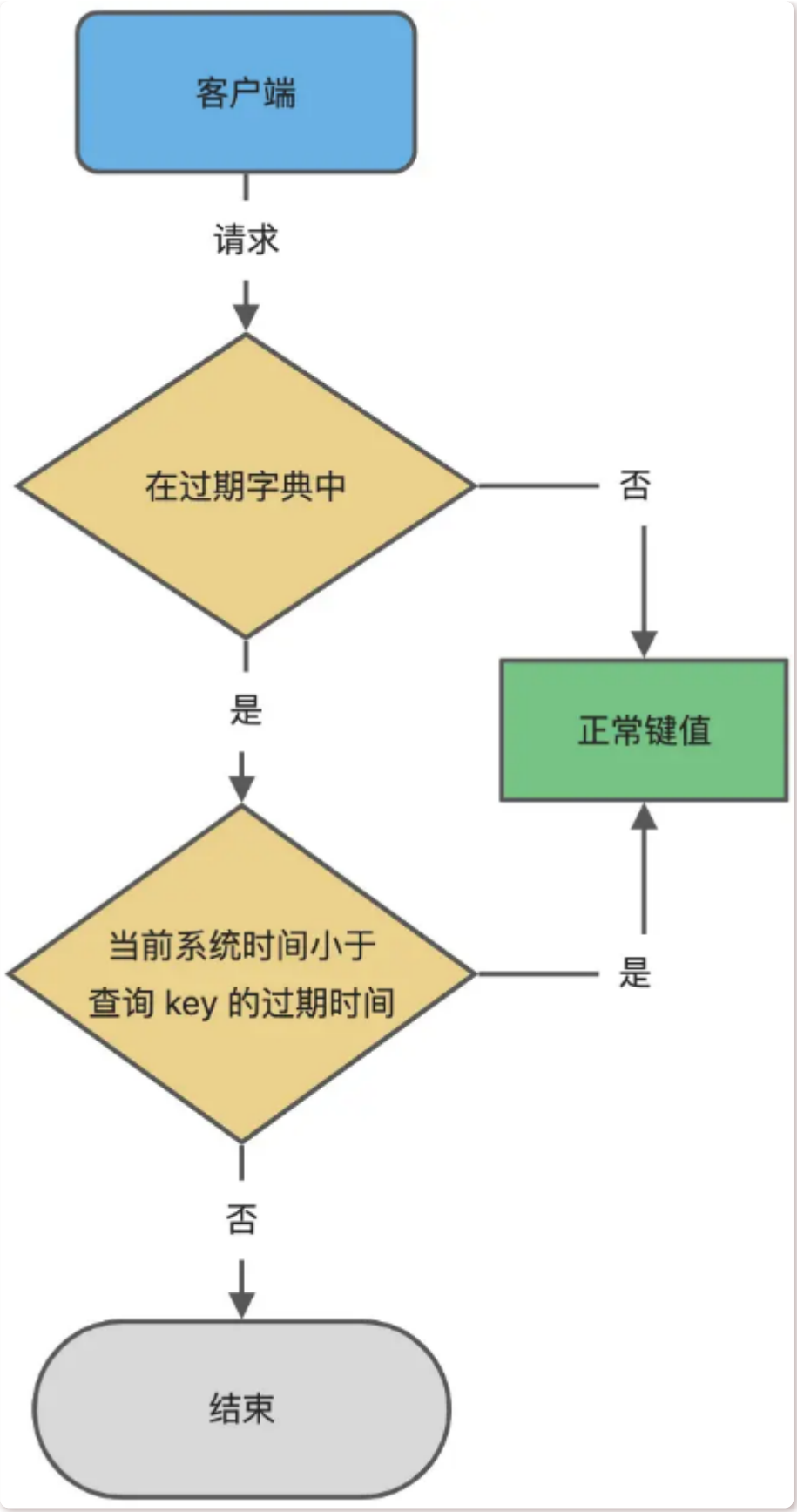
1.1、介绍

Redis 是可以对 key 设置过期时间的，因此需要有相应的机制将已过期的键值对删除，而做这个工作的就是过期键值删除策略。

每当我们对一个 key 设置了过期时间时，Redis 会把该 key 带上过期时间存储到一个**过期字典**（expires dict）中，也就是说「过期字典」保存了数据库中所有 key 的过期时间。

字典实际上是**哈希表**，哈希表的最大好处就是让我们可以用 `O(1)` 的时间复杂度来快速查找。当我们查询一个 key 时，Redis 首先检查该 key 是否存在于过期字典中：

- **如果不在**，则正常读取键值；
- **如果存在**，则会获取该 key 的过期时间，然后与当前系统时间进行比对，如果比系统时间大，那就没有过期，否则判定该 key 已过期。



1.2、定时删除策略

定时删除策略(TTL)的做法是，**在设置 key 的过期时间时，同时创建一个定时事件，当时间到达时，由事件处理器自动执行 key 的删除操作。**

- **优点**：可以保证过期 key 会被尽快删除，也就是内存可以被尽快地释放。因此，**定时删除对内存是最友好的**；

- **缺点**：在过期 key 比较多的情况下，删除过期 key 可能会占用相当一部分 CPU 时间，在内存不紧张但 CPU 时间紧张的情况下，将 CPU 时间用于删除和当前任务无关的过期键上，无疑会对服务器的响应时间和吞吐量造成影响。所以，**定时删除策略对 CPU 不友好**。

1.3、惰性删除策略

惰性删除策略(Lazy Expire)的做法是，**不主动删除过期键，每次从数据库访问 key 时，都检测 key 是否过期，如果过期则删除该 key。**

- **优点**：因为每次访问时，才会检查 key 是否过期，所以此策略只会使用很少的系统资源，因此，**惰性删除策略对 CPU 时间最友好**；
- **缺点**：如果一个 key 已经过期，而这个 key 又仍然保留在数据库中，那么只要这个过期 key 一直没有被访问，它所占用的内存就不会释放，造成了一定的内存空间浪费。所以，**惰性删除策略对内存不友好**。

1.4、定期删除策略

定期删除策略(Eviction)的做法是，**每隔一段时间「随机」从数据库中取出一定数量的 key 进行检查，并删除其中的过期key。**

- **优点**：通过限制删除操作执行的时长和频率，来减少删除操作对 CPU 的影响，同时也能删除一部分过期的数据减少了过期键对空间的无效占用；
- **缺点**：内存清理方面没有定时删除效果好，同时没有惰性删除使用的系统资源少，是一个**折中的策略**；
- **缺点**：难以确定删除操作执行的时长和频率。如果执行的太频繁，定期删除策略变得和定时删除策略一样，对CPU不友好；如果执行的太少，那又和惰性删除一样了，过期 key 占用的内存不会及时得到释放。

1.5、三者区别

过期删除策略	删除时间	CPU 资源消耗	内存开销
TTL（定时删除）	立即删除	较高	无
Lazy Expire（惰性删除）	访问时检查删除	较低	可能有过期键在内存中存留
Eviction（定期删除）	定期抽取删除	低	可能有过期键在内存中存留

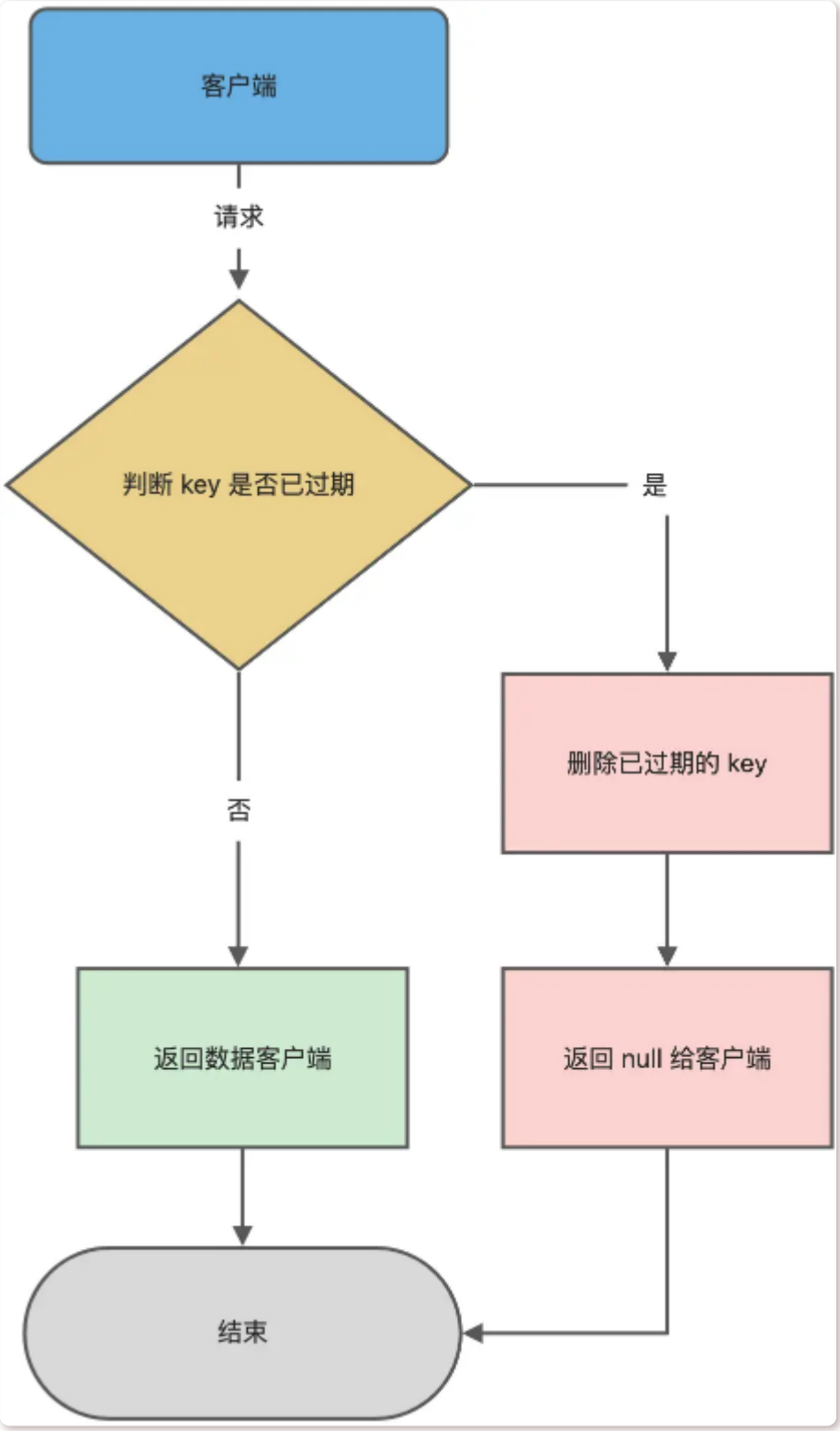
1.6、Redis实现

Redis使用的过期删除策略是「**惰性删除+定期删除**」这两种策略配和使用，以求**在合理使用CPU时间和避免内存浪费之间取得平衡**。

Redis 的惰性删除策略由 `db.c` 文件中的 `expireIfNeeded` 函数实现，Redis 在访问或者修改 key 之前，都会调用 `expireIfNeeded 函数对其进行检查，检查 key 是否过期：

- 如果过期，则删除该 key，至于选择异步删除，还是选择同步删除，根据 `lazyfree_lazy_expire` 参数配置决定（Redis 4.0版本开始提供参数），然后返回 null 客户端；

- 如果没有过期，不做任何处理，然后返回正常的键值对给客户端；



再回忆一下，定期删除策略的做法：**每隔一段时间「随机」从数据库中取出一定数量的 key 进行检查，并删除其中的过期key。**

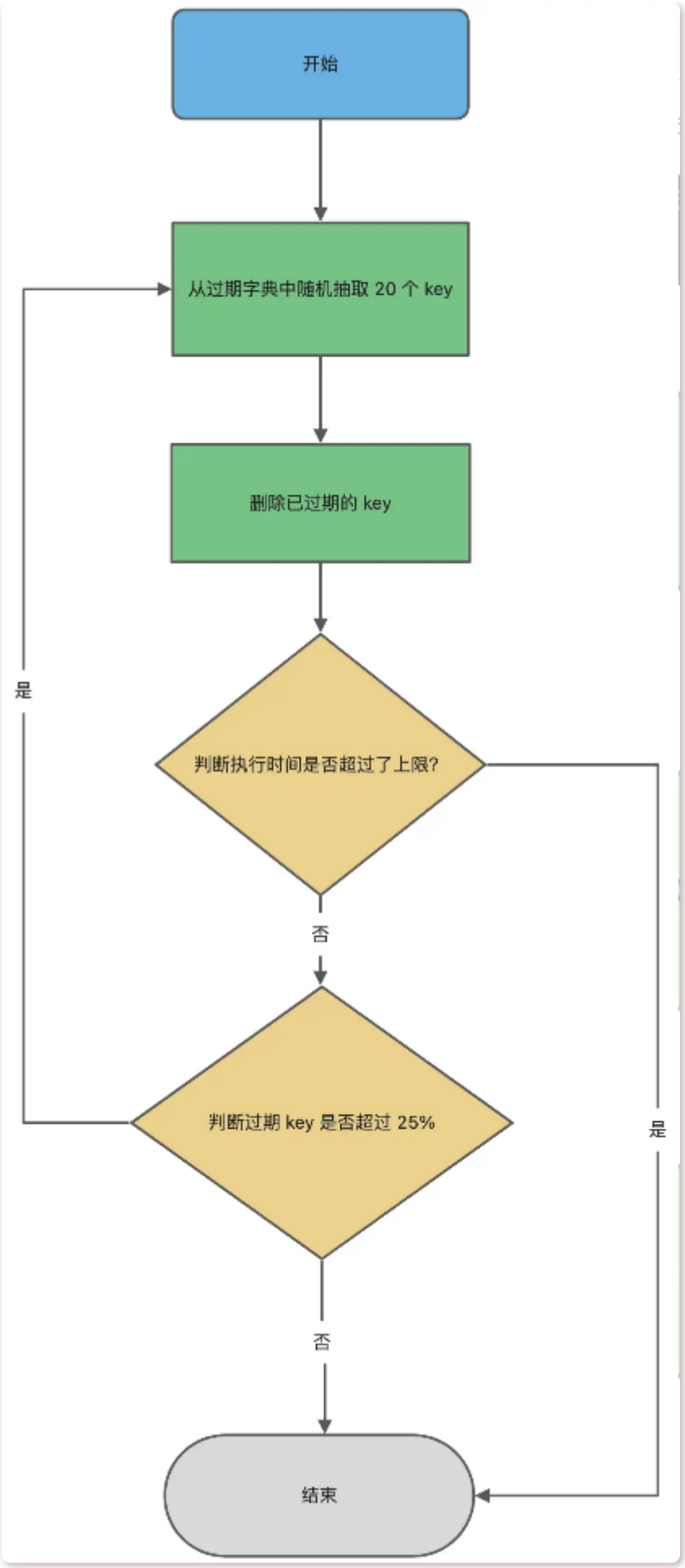
在 Redis 中，默认**每秒进行10次过期检查**一次数据库，此配置可通过 Redis 的配置文件 `redis.conf` 进行配置，配置键为 `hz`，它的默认值是 `hz 10`。

> 值得注意的是，每次检查数据库并不是遍历过期字典中的所有 key，而是从数据库中**随机抽取一定数量**的 key 进行过期检查。这个一定数量在源码中是写死的，并未提供对应的参数进行自定义配置，**数值固定为20**。

Redis 定期删除的流程为：

1. 从过期字典中随机抽取 `20` 个 key；
2. 检查这 `20` 个 key 是否过期，并删除已过期的 key；
3. 如果本轮检查的**已过期 key 的数量** 超过`5个(20*0.25)`，也就是「**已过期 key 的数量**」占比「**随机抽取 key 的数量**」大于 `25%`，则继续重复步骤 1，即重复执行删除策略；如果已过期的 key 比例小于 25%，则停止继续删除过期 key，然后等待下一轮再检查。

Redis 为了保证定期删除不会出现循环过度，导致线程卡死现象，为此增加了定期删除循环流程的时间上限，默认不会超过 `25ms`。


开始
从过期字典中随机抽取 20 个 key
删除已过期的 key
判断执行时间是否超过了上限?
是
否
判断过期 key 是否超过 25%
是
否
结束

1.7、持久化时过期键处理

Redis 持久化文件有两种格式：RDB（Redis Database）和 AOF（Append Only File），下面我们分别来看过期键在这两种格式中的呈现状态。

RDB 文件分为两个阶段，RDB **文件生成**阶段和**加载**阶段：

- 「**RDB 文件生成阶段**」从内存状态持久化成 RDB（文件）的时候，会对 key 进行**过期检查**，过期的键**不会**被保存到新的 RDB 文件中，因此 Redis 中的过期键**不会**对生成新 RDB 文件产生任何影响；
- 「**RDB 加载阶段**」如果 Redis 是**主服务器运行模式**的话，在载入 RDB 文件时，程序会对文件中保存的键进行**过期检查**，过期键**不会**被载入到数据库中。所以过期键**不会**对载入 RDB 文件的主服务器造成影响；
- 「**RDB 加载阶段**」如果 Redis 是**从服务器运行模式**的话，在载入 RDB 文件时，不论键**是否过期都会被载入**到数据库中。但由于主从服务器在进行**数据同步**时，从服务器的数据会**被清空**。所以**一般来说**，过期键对载入 RDB 文件的从服务器也**不会**造成影响。

AOF 文件分为两个阶段，AOF **文件写入**阶段和**重写**阶段。

- 「**AOF 文件写入阶段**」当 Redis 以 AOF 模式持久化时，如果数据库某个过期键还**没被删除**，那么 AOF 文件会保留此过期键，当此过期键**被删除后**，Redis 会向 AOF 文件追加一条 DEL 命令来**显式地删除**该键值；
- 「**AOF 重写阶段**」执行 AOF 重写时，会对 Redis 中的键值对进行**过期检查**，已过期的键不会被保存到重写后的 AOF 文件中，因此**不会**对 AOF 重写造成任何影响。

1.8、主从模式过期键处理

当 Redis 运行在主从模式下时，**从库不会进行过期扫描，从库对过期的处理是被动的**。也就是即使从库中的 key 过期了，如果有客户端访问从库时，依然可以得到 key 对应的值，像未过期的键值对一样返回。

从库的过期键处理依靠主服务器控制，**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。

2、内存淘汰机制

2.1、介绍

Redis的内存淘汰机制是为了解决内存占用过高的问题。在 Redis 的运行内存达到了某个阈值，就会触发**内存淘汰机制**，根据一定的策略来选择一些键值对进行删除，从而释放部分内存。这个阈值就是我们设置的最大运行内存，此值在 Redis 的配置文件中可以找到，配置项为 *maxmemory*。

常见的内存淘汰策略有：

1. *LRU（Least Recently Used，最近最少使用）*：淘汰整个键值中最久未使用的键值；

2. `LFU（Least Frequently Used，最不经常使用）`：淘汰整个键值中最少使用的键值；
3. `Random（随机）`：随机选择键值对进行淘汰。
4. `noeviction（不进行数据淘汰）`：Redis3.0之后，默认的内存淘汰策略。它表示当运行内存超过最大设置内存时，不淘汰任何数据，而是不再提供服务，直接返回错误。

> 🌈
>
> 虽然虽然，但是后面就不介绍后两个策略了，主要介绍前两个策略：
>
> - 随机策略：想介绍也没东西介绍，就随缘抓几个起来噶掉这玩意
> - 不进行数据淘汰策略：想介绍也没东西介绍，就直接把门关了这玩意

2.2、LRU

LRU（Least Recently Used，最近最少使用）是Redis3.0之前默认的内存淘汰策略，它是淘汰整个键值中最久未使用的键值。

传统 LRU 算法的实现是基于「**链表**」结构，链表中的元素**按照操作顺序从前往后排列**，最新操作的键会被移动到表头，当需要**内存淘汰**时，只需要**删除链表尾部**的元素即可，因为链表尾部的元素就代表最久未被使用的元素。

Redis 并没有使用这样的方式实现 LRU 算法，因为传统的 LRU 算法存在两个问题：

- 需要用链表管理所有的缓存数据，这会带来**额外的空间开销**；
- 当有数据被访问时，需要在链表上把该**数据移动**到头端，如果有大量数据被访问，就会带来很多链表移动操作，会很**耗时**，进而会**降低 Redis 缓存性能**。

> 🌈
>
> Redis 实现的是一种**近似 LRU 算法**，目的是为了更好的节约内存，它的实现方式是在 Redis 的**对象结构体中添加一个额外的字段，用于记录此数据的最后一次访问时间**。

当 Redis 进行内存淘汰时，会使用**随机采样**的方式来淘汰数据，它是默认随机取 5 个值（此值可配置），然后**淘汰最久没有使用的那个**。

Redis 实现的 LRU 算法的优点：

- 不用为所有的数据维护一个大链表，**节省了空间占用**；
- 不用在每次数据访问时都移动链表项，**提升了缓存的性能**；

但是 LRU 算法有一个问题，由于是**随机采样**的方式来淘汰数据，因此**无法解决缓存污染问题**。比如应用一次读取了大量的数据，而这些数据只会被读取这一次，如果运气炸裂每次随机采样都采不到它们，那么这些数据会留存在 Redis 缓存中很长一段时间，造成缓存污染。

2.3、LFU

LFU 全称是 Least Frequently Used 翻译为**最近最不常用的**，是在Redis4.0新增的一种内存淘汰策略。

LFU 算法是根据**数据访问次数**来淘汰数据的，它的核心思想是“如果数据过去被访问多次，那么将来被访问的频率也更高”。

> 其实严格来说，LFU算法是根据**数据访问频率**来淘汰数据的。

所以， LFU 算法会记录每个数据的访问次数。当一个数据被再次访问时，就会增加该数据的访问次数。这样就解决了偶尔被访问一次之后，数据留存在缓存中很长一段时间的问题，相比于 LRU 算法也更合理一些。

LFU 算法相比于 LRU 算法的实现，多记录了「数据的访问频次」的信息。Redis 对象的结构如下：

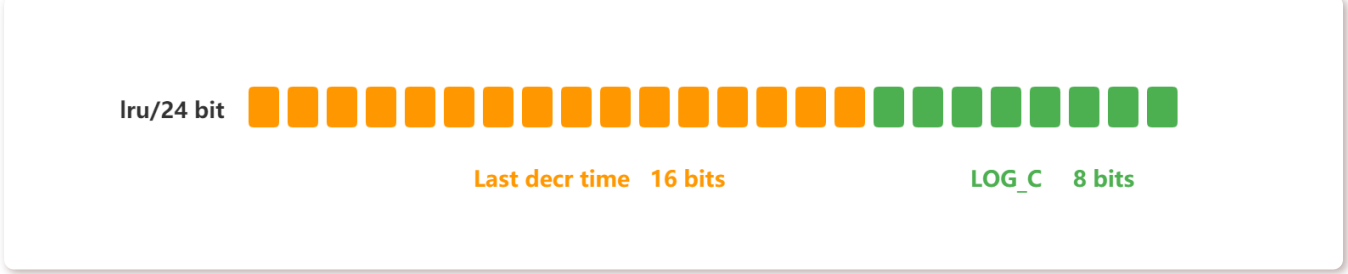
```
typedef struct redisObject {
    ...

    // 24 bits，用于记录对象的访问信息
    unsigned lru:24;
    ...
} robj;
```

Redis 对象头中的 lru 字段，在 LRU 算法下和 LFU 算法下使用方式并不相同。

在 LRU 算法中，Redis 对象头的 *24 bits* 的 *lru* 字段是用来记录 key 的**访问时间戳**，因此在 LRU 模式下，Redis可以根据对象头中的 *lru* 字段记录的值，来比较最后一次 key 的访问时间长，从而淘汰最久未被使用的 key。

在 LFU 算法中，Redis对象头的 *24 bits* 的 *lru* 字段被分成两段来存储， *高16bit* 存储 *ldt*(Last Decrement Time)， *低8bit* 存储 *logc*(Logistic Counter)：

- *ldt* 是用来记录 key 的**访问时间戳**；
- *logc* 是用来记录 key 的**访问频次**，它的值越小表示使用频率越低，越容易淘汰，每个新加入的 key 的**logc 初始值为 5**。



> 注意，`logc` 并不是单纯的访问次数，而是访问频次（访问频率），因为 **`logc` 会随时间推移而衰减的**。

在每次 key 被访问时，会先对 `logc` 做一个衰减操作，**衰减的值跟前后访问时间的差距有关系**.如果上一次访问的时间与这一次访问的时间差距很大，那么衰减的值就越大，这样实现的 LFU 算法是根据**访问频率**来淘汰数据的，而不只是访问次数。

访问频率需要考虑 key 的**访问是多长时间段内发生的**。key 的先前访问距离当前时间越长，那么这个 key 的访问频率相应地也就会降低，这样被淘汰的概率也会更大。

对 `logc` 做完衰减操作后，就开始对 `logc` 进行增加操作，增加操作并不是单纯的自增，而是**根据概率增加**，如果 `logc` 越大的 key，它的 `logc` 就越难再增加。

所以，Redis 在访问 key 时，对于 `logc` 是这样变化的：

1. 先按照**上次访问距离当前的时长**，来对 `logc` 进行衰减；
2. 然后，再**按照一定概率**增加 `logc` 的值

`redis.conf` 提供了两个配置项，用于调整 LFU 算法从而控制 `logc` 的增长和衰减：

- `lfu-decay-time` 用于调整 `logc` 的衰减速度，它是一个**以分钟为单位**的数值，默认值为1，`lfu-decay-time` 值越大，衰减越慢；
- `lfu-log-factor` 用于调整 `logc` 的增长速度，`lfu-log-factor` 值越大，`logc` 增长越慢。

2.4、区别

策略	LRU（最近最少使用）	LFU（最不经常使用）
淘汰原则	最久未使用的键值对	访问次数最少的键值对
更新机制	当键被访问时更新使用时间	当键被访问时增加访问次数
适用场景	**处理热数据**，最近被访问频繁的数据	**处理稳定数据**，访问次数相对固定的数据

3、区分过期删除和内存淘汰

内存淘汰机制：

- 当 Redis 的内存使用达到配置的最大内存限制时，内存淘汰机制会根据预先设置的策略来删除一些键值对，以释放内存空间。
- 内存淘汰机制并不关心键是否设置了过期时间，它主要根据某种算法选择要淘汰的键值对，以腾出更多的内存空间，使得新的键值对可以被存储在内存中。
- 常见的内存淘汰策略有 LRU（最近最少使用）、LFU（最不经常使用）、随机等。

过期删除机制：

- Redis 允许为键设置过期时间，过期删除机制是在键设置了过期时间后，当键过期时自动将其删除。
- 过期删除机制并不是为了释放内存，而是为了使 Redis 中的数据始终保持最新的状态。过期的键值对将不再对外提供服务，直到下次有读或写操作访问该键时，Redis 会发现键已经过期，然后将其删除。

特点	内存淘汰机制	过期删除机制
目的	释放内存空间	维护数据的时效性
触发时机	当 Redis 内存达到上限时，触发淘汰策略	当键设置了过期时间后，等待键过期触发过期删除策略
依赖键的过期时间	不依赖键的过期时间，所有键都有可能被淘汰	只对设置了过期时间的键起作用
策略	1. LRU 2. LFU 3. 随机	1. 定时删除 2. 惰性删除 3. 定期删除