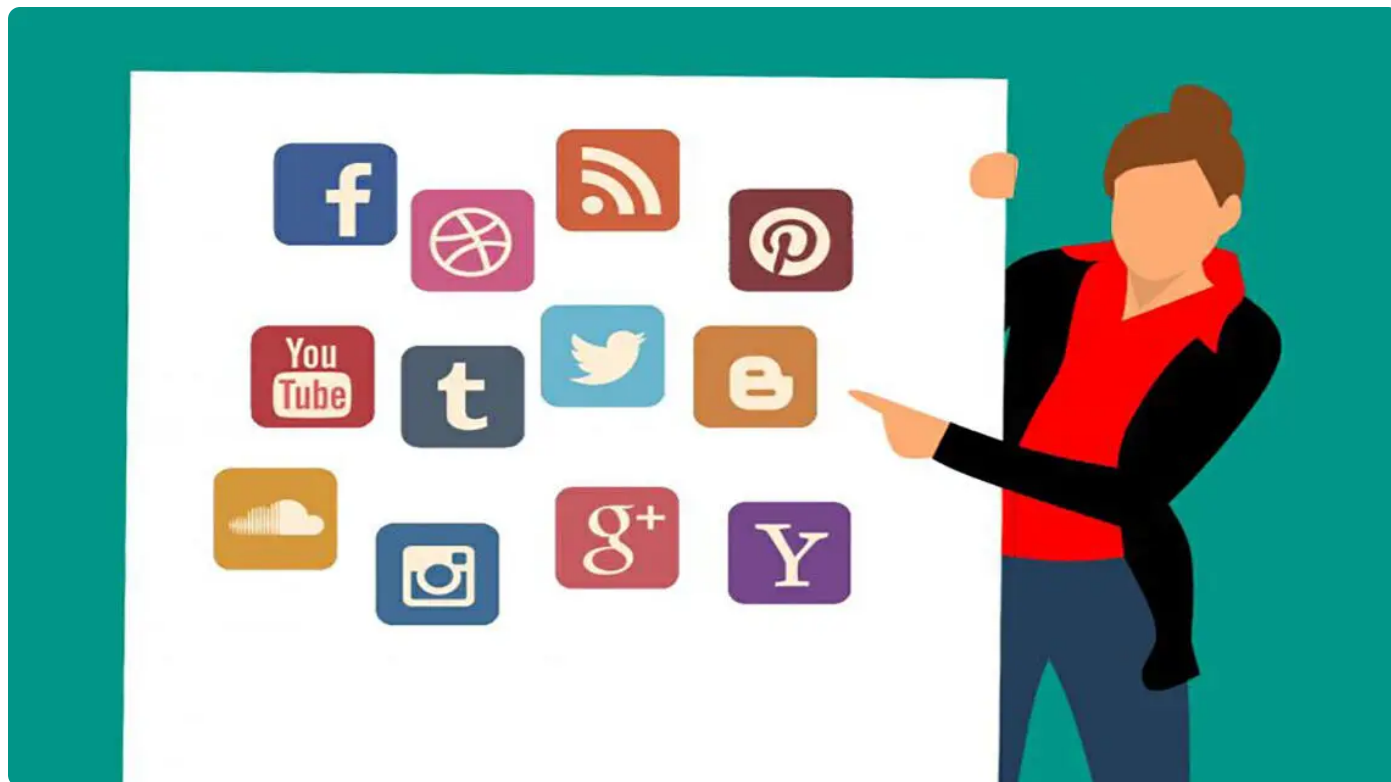


## 32 | 缓存淘汰策略：怎么淘汰缓存命中率才不会下降？

2023-9-1 邓明



你好，我是大明。今天我们来聊一个使用缓存的时候绕不开的话题——如何淘汰键值对？

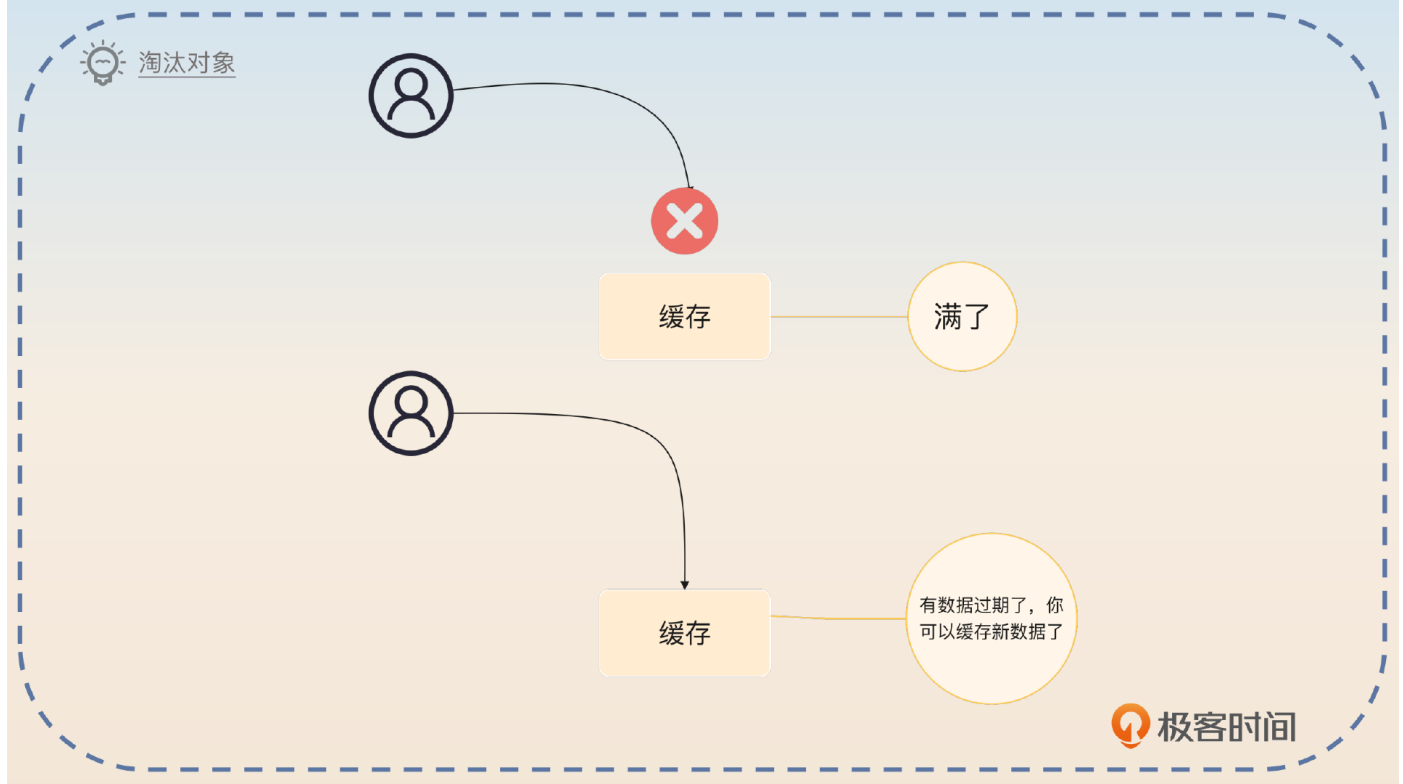
这个问题在面试中也很常见。大多数人都能把八股文背好，也就是常见的 LRU 之类的策略都能回答出来。而它在面试中的独特优势，就是能够根据业务设计出花样繁多的解决方案。如果你能在面试中用上这些方案，就能给面试官留下很深刻的印象。

那么我们先从“为什么要淘汰”这个问题开始学习。

### 为什么要淘汰？

我们日常在使用缓存的时候，都会尝试控制整个缓存的开销，尤其是本地缓存的内存开销。比如说线上偶尔会遇到本地缓存了太多数据，导致应用内存不足的问题。如果是 Java 这种垃圾回收的语言，那么你就会遇到频繁地垃圾回收甚至 full GC 之类的问题。

所以用缓存肯定要控制住缓存的内存使用量。而这就会引出一个问题，万一我达到了内存使用上限，但是我又需要加入新的键值对，怎么办？最保守的做法就是直接报错，那么你就没有办法缓存新的数据了。后续如果缓存中已有的数据过期了，你就能缓存新的数据了。



但是大多数的业务是不能接受这种方案的，因为这一类的业务认为已经在缓存中的数据可能用不上了，虽然还没有过期，但是还是可以考虑淘汰掉，腾出空间来存放新的数据。这些新的数据比老的数据有更大的可能性被使用。

## 淘汰算法

最有名的淘汰算法是 LRU 和 LFU。除了这两种，还有最佳置换算法（OPT）和先进先出置换算法（FIFO）等，但是用得都不如 LRU 和 LFU 多，所以这里主要聊 LRU 和 LFU 这两种。

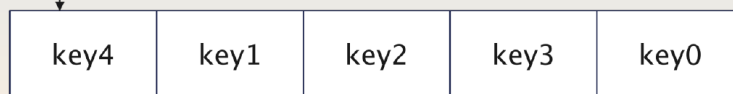
### LRU

LRU（Least Recently Used）是指最近最少使用算法。也就是说，缓存容量不足的时候，就从所有的 key 里面挑出一个最近一段时间最长时间未使用的 key。

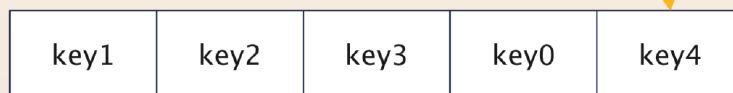
这个算法从实现上来说很简单，只需要把 key 用额外的链表连起来，然后每次被访问到的 key 都挪到队尾，那么队首就是最近最长时间未访问过的 key。也可以反过来，每次访问过的挪到队首，那么队尾就是最近最久未访问过的 key。比如说你可以借助 Java 的 LinkedHashMap 轻易实现 LRU 算法。



读/写



挪到后面



这里访问是一个含糊的说法，你可以认为读写都是访问，也可以认为只有写是访问。所以有个 LRU 的变种是只有在写的时候，才会挪动这个 key，读并不会，也就是它倾向于保留写频繁的数据。

## LFU

LFU (Least Frequently Used) 是最不经常使用算法，它是根据对象的使用次数来确定淘汰对象的，每次都是将使用次数最低的淘汰掉。所以基本的思路就是按照访问频率来给所有的对象排序。每次要淘汰的时候，就从使用次数最少的对象里面找出一个淘汰掉。

如果有好几个对象的访问频次恰好相等，而且又是最低的，那么你可以自由决策如何淘汰。标准做法是淘汰最先插入的，不过也有一些实现就是随机删一个，又或者删掉排序位置最小的那个。实现的基本思路就是每次读写的时候，对象上面的次数都加 1，然后调整位置。



读/写

key1 10	key2 10	key3 13	key0 100	key4 20
------------	------------	------------	-------------	------------

key2 10	key1 11	key3 13	key0 100	key4 20
------------	------------	------------	-------------	------------

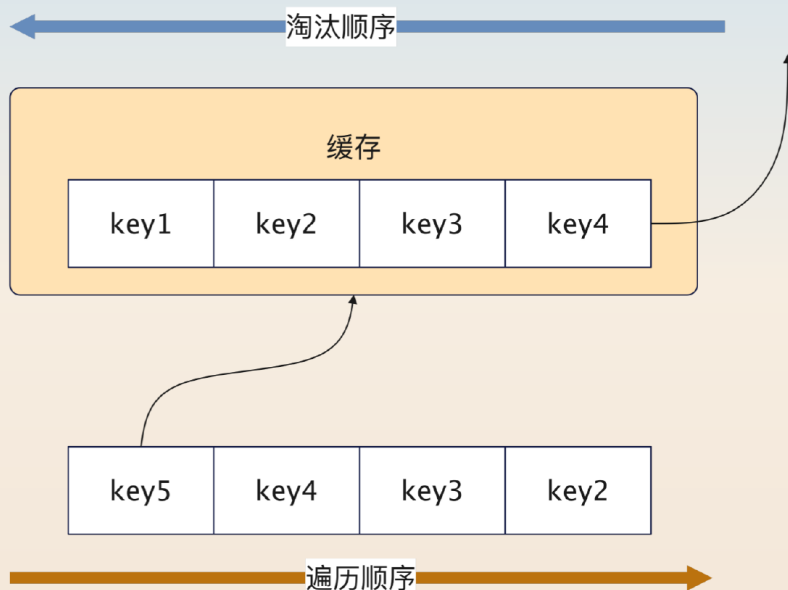


这个算法也有一些变种。最主要的变种是统计一段时间内的访问次数而不是整个生命周期的次数。比如说每个对象都只统计近一个小时内的访问次数。但是这种变种的实现复杂度就要高很多。

## 小结

在实践中，一般是优先考虑 LRU。这主要是因为 LRU 对时间局部性突出的应用非常友好，而大多数的应用场景都满足时间局部性的要求。

但是 LRU 在一些特殊的场景下，表现也不好。最典型的场景就是访问历史记录，因为越是历史悠久的，越有可能已经被你淘汰了。另外一个场景是遍历，遍历的时候，当次被遍历到对象总是不会被淘汰掉，而实际上，已经被遍历的对象反而应该被淘汰掉，腾出空间给尚未遍历的对象。



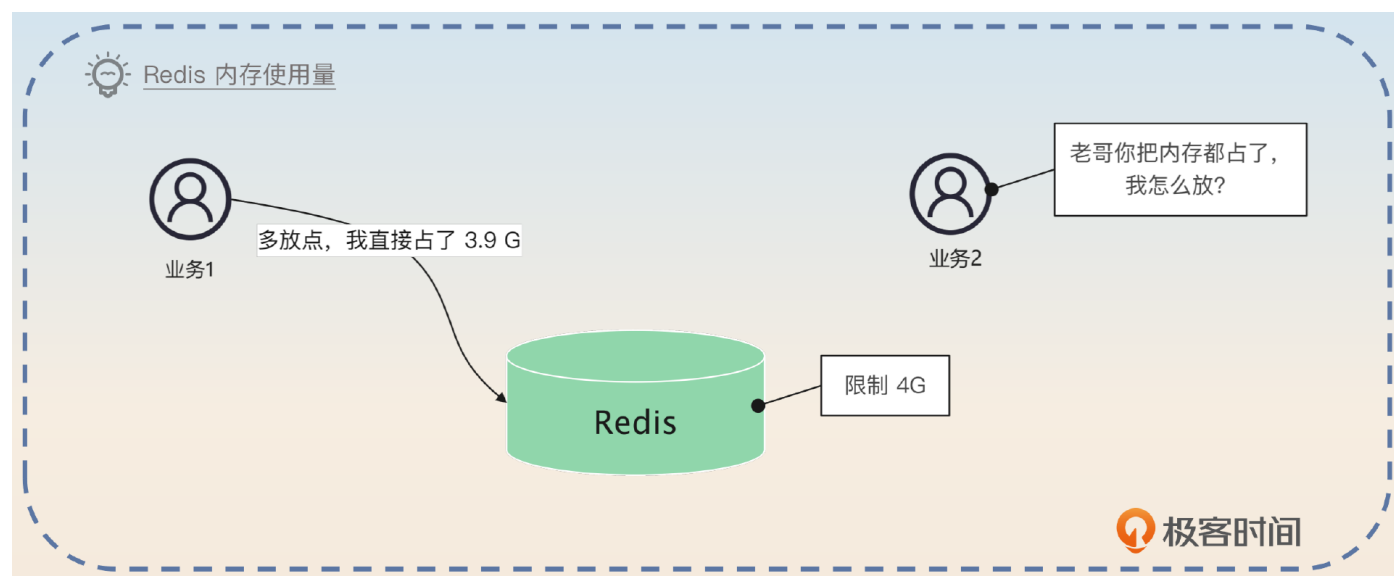
例如，图中遍历到 key5 的时候会触发淘汰，把 key4 淘汰了。紧接着遍历 key4，会把 key3 淘汰了。以此类推，最终结果就是缓存完全没命中。

## Redis 支持的淘汰算法

Redis 支持很多种淘汰算法，你可以通过 `maxmemory` 选项来控制 Redis 的整个内存使用量，还可以通过指定 `maxmemory_policy` 来设置淘汰算法。

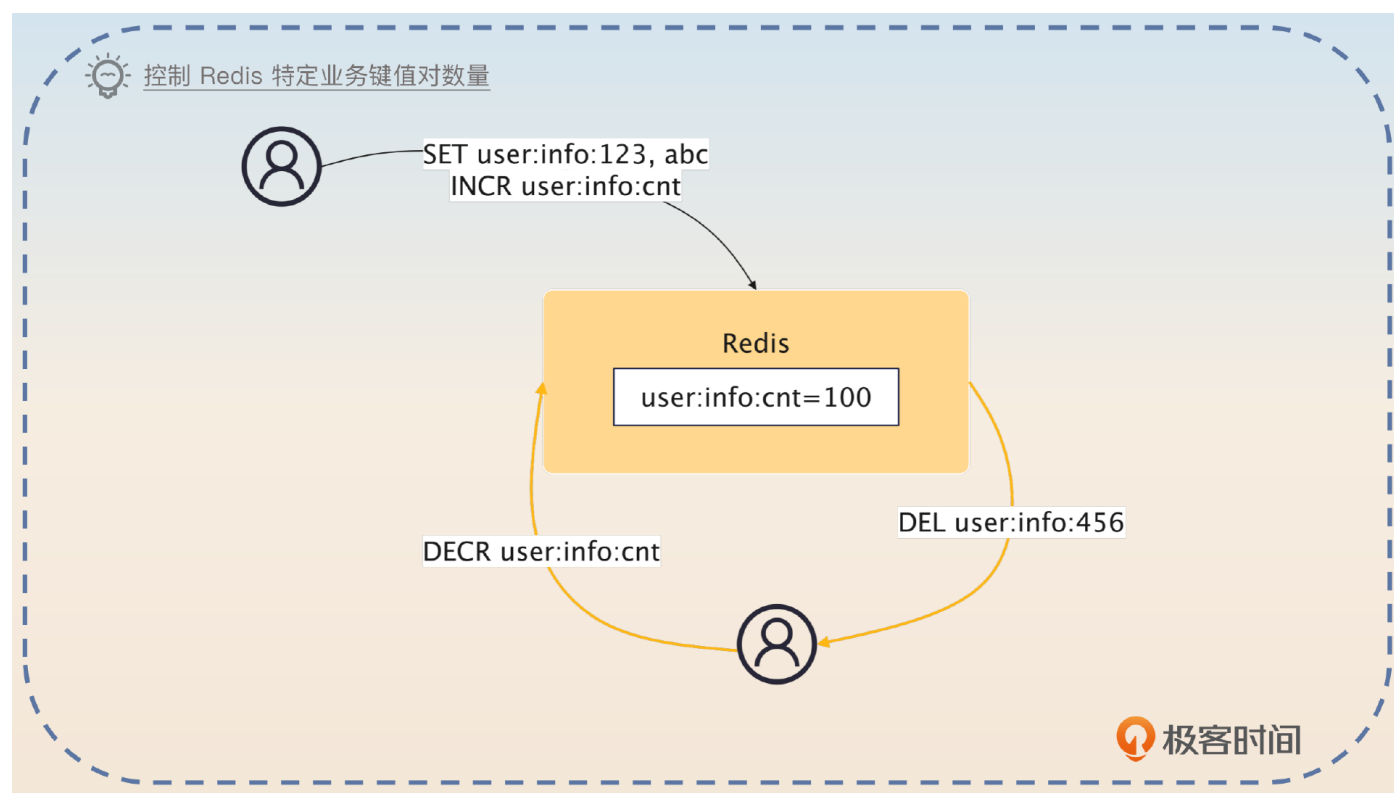
<code>volatile-lru</code>	在设置了过期时间的键值对中执行 LRU 算法
<code>volatile-lfu</code>	在设置了过期时间的键值对中执行 LFU 算法
<code>volatile-random</code>	在设置了过期时间的键值对中执行随机淘汰算法
<code>volatile-ttl</code>	将过期时间最早的键值对清除掉
<code>allkeys-lru</code>	在所有 key 中执行 LRU 算法
<code>allkeys-lfu</code>	在所有 key 中执行 LFU 算法
<code>allkeys-random</code>	在所有 key 中执行随机淘汰算法
<code>noeviction</code>	不执行淘汰算法，如果超过内存使用量，则返回错误

另外，你没有办法控制某些 key 的最大内存使用量，比如说某个业务最多缓存 100M 的数据，你只能控制整个 Redis 实例的内存使用量。



但是我们可以通过控制某个业务允许的键值对数量和单一键值对最大内存来间接控制业务在 Redis 上的内存使用量。比如说某个业务只允许有 10000 个键值对，每个键值对不能超过 1KB，那么它的整个内存使用量就不会超过 10MB。

你可以考虑在 Redis 中额外记录一个键值对数量，同时监听键值对的删除命令。当新加入某个键值对的时候，键值对数量加一。当某个键值对被删除的时候，键值对减一，那么就算键值对自然过期被淘汰，你还是能够得到通知。



注意，你要区分一下你这个键值对是一个全新的键值对，还是已有这个键但是值被修改了。在后一种情况下，你并不需要更新键值对的数量。

这种机制很复杂，如果你想把我在后面提到的那些算法用在 Redis 上，就得编写一些比较复杂的 lua 脚本。

## 面试准备

在面试缓存淘汰之前，你需要了解清楚公司内的一些情况。

公司的 Redis 淘汰策略是什么？中间有没有调整过？如果有，那么为什么调整？

你有没有用过本地缓存？如果用过本地缓存，那么这个本地缓存是如何控制内存使用量的？它支持什么样的缓存淘汰策略？

有没有出现过因为缓存占用内存太大而引起的线上故障？后来是怎么解决的？

有没有使用 Redis 不当引起的线上故障？后来是怎么解决的？

我非常建议你抽空的时候写一个本地缓存。对于大多数语言来说，实现一个本地缓存所需的数据结构大多数都已经有了，所以实际上难度并不大。

在这个过程中，你就可以尝试设计不同的淘汰策略，这样你就会有更加深刻的理解，面试的时候也就不会害怕面试官问细节了。

除此以外，当面试官问到以下问题，你也可以用这节课的内容来回答。

在使用缓存的时候有什么注意事项？

为什么要控制缓存的内存使用量？

你使用缓存遇到过什么问题？

你优化过缓存吗？怎么提高缓存的命中率？

Redis 支持哪些淘汰策略？你们公司用的是是什么？

## 基本思路

最好的面试思路还是把优化缓存淘汰策略作为保证系统高性能的一环。这里就可以用对比说明策略，就是在优化之前缓存命中率或者响应时间有多差，而在优化之后，缓存命中率提高了多少或者响应时间下降了多少。

这里我用优先淘汰代价低的案例，给你展示如何回答。

为了进一步提高系统的性能，我还尝试过优化缓存。早期我们有一个业务，用到了一个本地缓存。这个本地缓存使用的淘汰算法是 LRU，最开始我们都觉得这个算法没什么问题。后来业务反馈，说有几个大客户一直抱怨自己的查询时快时慢。一听到时快时慢，我就可以确定应该是缓存出了问题。

经过排查我们发现原来这个缓存执行 LRU 的时候有时会把大客户的数据淘汰掉。而偏偏大客户的数据实时计算很慢，所以一旦没有命中缓存，响应时间就会暴增。

后来我进一步梳理业务，一方面考虑进一步增大缓存的可用内存。另外一方面，设计了灵活的淘汰策略，在淘汰的时候优先淘汰小客户的数据。

这样做的好处就是优先保证了大客户的用户体验，平均响应时间下降了40%。而小客户因为本身计算就很快，所以影响也不是很大。

当然，从实践角度来说，针对不同业务调整淘汰策略取得的收益是比较小的。但是站在面试的角度，这也算是一个比较新奇的优化点，倒是可以用用。

除了你主动提起，面试官也可能主动询问，比如从怎么安全使用缓存这个点切入。“安全”强调的就是你用缓存不能影响到别人，那么控制缓存的内存使用量就是一个关键的环节。所以这种时候你应该这样回答：

使用缓存一定要注意控制缓存的内存使用量，不能因为某一个业务而直接把所有的内存都耗尽。

要控制住内存使用量，就需要在内存用完的时候淘汰一些键值对。

实际上，解决缓存淘汰的最佳思路，就是给缓存足够的内存，不触发淘汰。但这也可以说是正确但是无用的废话，因为现实中不可能你需要多少内存就给你多少内存。所以最佳还是要



考虑怎么淘汰键值对。

解决缓存淘汰的问题，应该优先考虑增加内存，降低缓存淘汰的几率。不过毕竟内存也不是无限的，最终都还是要选择合适的淘汰策略。比如说我们公司的 Redis 使用了 volatile-lru 淘汰策略。

这里你可以将 volatile-lru 淘汰策略换成你们实际的策略。有些时候面试官也可能会直接问你 Redis 支持哪些淘汰策略，这时候你按照前置知识里面罗列出来的内容回答就可以了，同时你不要忘记补充你的 Redis 使用的是什么策略。

最后，你记得补充一句话，用来引导话题。

这些 LRU 或者 LFU 之类的算法都是普适性很强的算法，但是我也用过一些更加针对业务的淘汰算法。比如说按照优先级淘汰，大对象优先淘汰、小对象优先淘汰、代价低优先淘汰。大多数时候，不论是 Redis 还是本地缓存，这种业务针对性特别强的算法，都得自己实现。

我相信面试官会对针对业务特性来设计淘汰策略感兴趣，因为它能够体现你对业务的理解，以及灵活设计方案的能力。

## 亮点方案

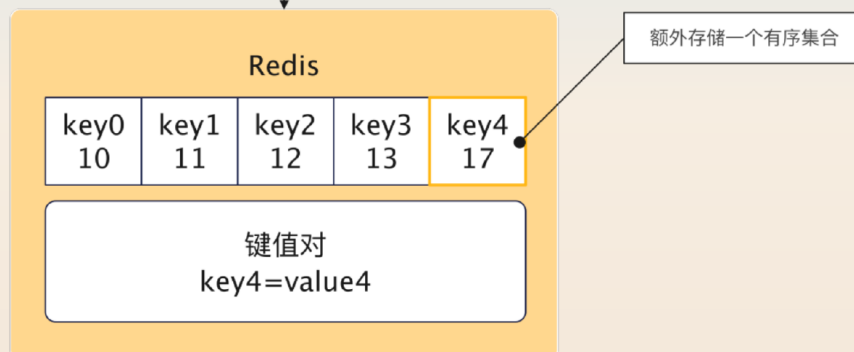
在实践中，同一个公司内部业务的重要性都是有区别的。就算是同一个业务内部，不同用户的重要性也是有区别的。所以针对业务设计淘汰策略的一个核心思路就是根据业务的重要性来设计。

比如说某个服务同时服务于 VIP 用户和普通用户，那么完全可以在缓存触发淘汰的时候，先把普通用户的数据淘汰了。所以可以考虑为每一个键值对绑定一个优先级，每次缓存要执行淘汰的时候，就从先淘汰优先级最低的数据。

这个方案在 Redis 落地基本上就是在前面提到的控制键值对数量的基础上，引入一个按照优先级进行排序的有序集合，那么有序集合里元素个数就是键值对个数。在每次触发淘汰的时候，从有序集合里面取出来一个 key，再把对应的键值对删除。



lua: key4, 17, value4

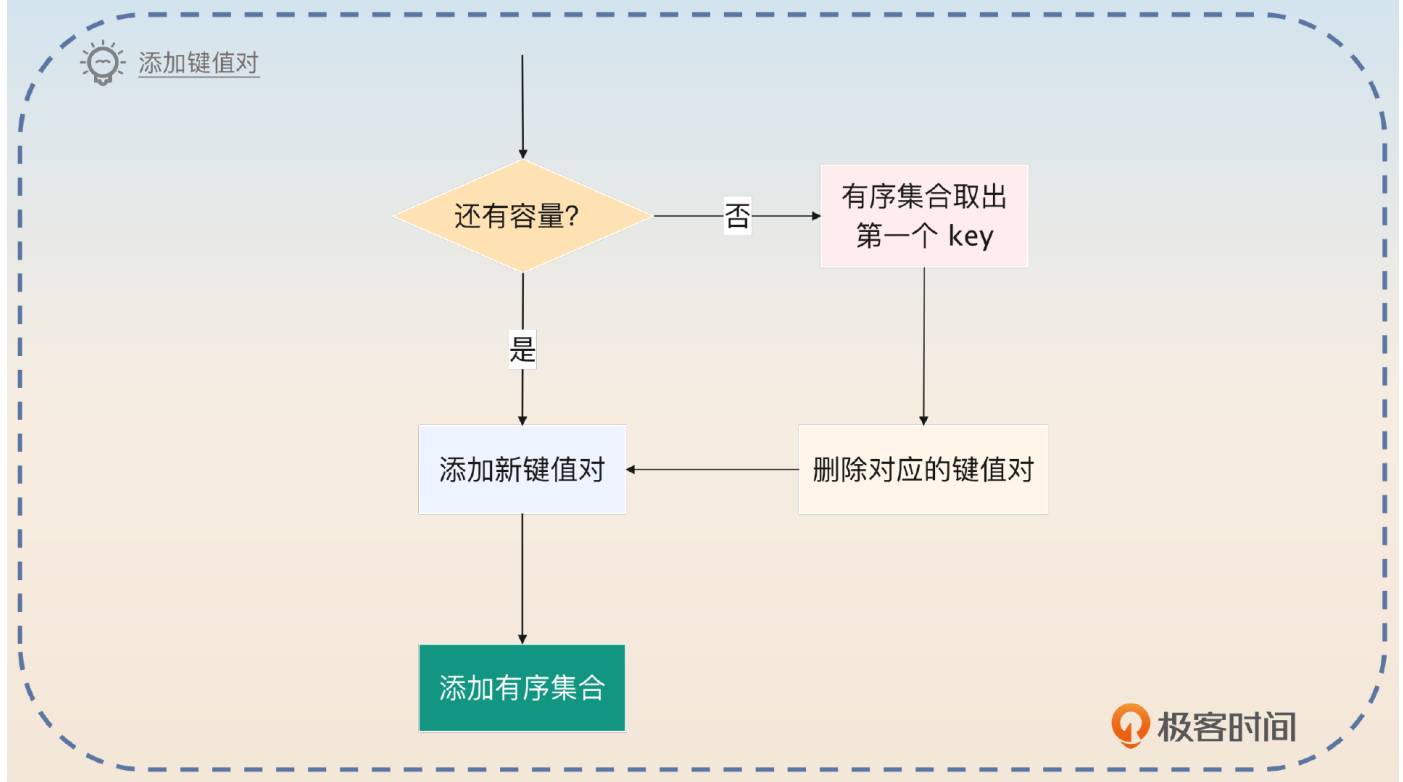


你可以先简单介绍一下这个方案。

之前我在业务里面使用过一个按照优先级来淘汰的策略。我们的业务有一个特点，就是数据有很鲜明的重要性之分。所以对于我们来说，应该尽可能保证优先级高的数据都在缓存中。所以在触发了淘汰的时候，我们希望先淘汰优先级比较低的缓存。所以我在 Redis 上利用有序集合设计了一个控制键值对数量，并且按照优先级来淘汰键值对的机制。这个有序集合是使用数据的优先级来排序的，也就是用优先级作为 score。

这里，你要仔细解释每一个部分。首先是增加键值对的时候是如何操作的。

增加一个键值对就要执行一个 lua 脚本。在这个脚本里面，它会先检测有序集合里面的元素个数有没有超过允许的键值对数量上限，如果没有超过，就写入键值对，再把 key 加入有序集合。如果超过了上限，那么就有序集合里面拿出第一个 key，删除这个 key 对应的键值对。



然后补充怎么在键值对过期的时候，维护有序集合。

同时监听 Redis 上的删除事件，每次收到删除事件，就把有序集合中对应的 key 删除。

那么在这个基础上，根据你计算优先级的不同方式，你可以将这个机制用于不同的场景。所以需要总结拔高一下。

在这个基础上，我可以根据不同的业务特征来计算优先级，从而实现大对象先淘汰、小对象先淘汰、热度低先淘汰等算法。

这里我稍微解释一下这个总结里面提到的几种淘汰策略，你在面试官问到的时候可以补充说明。

## 先淘汰大对象

在一些场景下，如果你的缓存有大有小，那么在淘汰的时候你会希望尽可能先把大的对象淘汰了，因为这样可以一次性腾出比较多的内存。这种策略在本地缓存实现上经常使用。与之对应的另一些场景下，你可以考虑小对象优先淘汰。

## 先淘汰小对象

比如说你的大对象都是经过复杂计算之后得出来的，而小对象的计算逻辑就非常简单。那么显然淘汰大对象是不合适的，因为一旦缓存未命中，那么响应时间就会大幅度提高。而淘汰小对象，即便下一次触发了重复计算，响应时间也没多少波动。进一步抽象就可以说成是代价低的优先淘汰。

所谓的代价高低是一种逻辑上的概念，你可以把任何你认为会对你的业务产生关键性影响的资源认为是“代价”，而消耗很多这个资源的，就是代价高昂。不怎么消耗这个资源的，就可以认为是代价低廉。

那么小对象先淘汰也是基于这样一个假设，小对象对应的计算步骤都是很快的，代价是比较低的，所以可以先淘汰。但是淘汰小对象也有一个缺陷，就是如果新加入的键值对数据比较大，那么你需要淘汰好几个小对象才能腾出足够的空间。

## 低热度优先

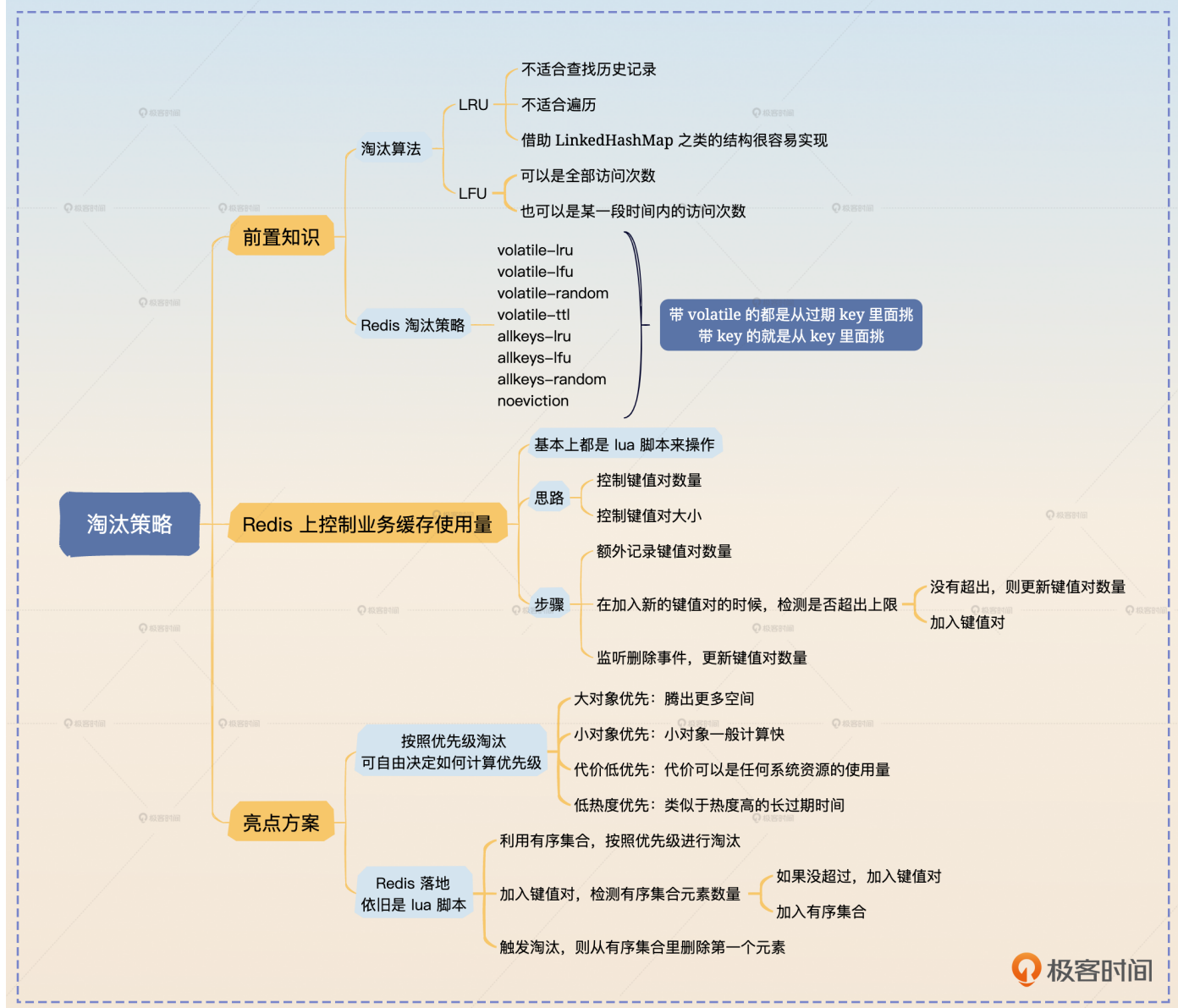
这种是根据数据冷热来淘汰，比如说存在博主粉丝关系的内容平台上，在淘汰数据的时候完全可以先将小博主的数据淘汰了，留下大博主的数据。因为我们知道大博主粉丝更多，所以他们的数据留下来有更大的概率会被访问到。这个和上节课提到的热点数据缓存时间会更长的思路是一样的。

## 面试思路总结

最后我们来总结一下这节课的内容。

你需要先了解为什么会有淘汰这种说法，原因就是我们希望控制住内存使用量。常见的淘汰算法LRU、LFU，以及 Redis 支持的各种淘汰算法比较重要，在准备面试的过程中，记得提前了解你们公司 Redis 使用的算法。

最后我提出了一个针对业务特性来设计的按照优先级淘汰的算法，根据怎么计算优先级又可以进一步细分成大对象先淘汰、小对象先淘汰、根据热度来淘汰。这些方案在 Redis 上落地的时候，都需要借助 Redis 的有序集合，而且为了避免并发问题，你操作 Redis 的时候都是要通过 lua 脚本封装一系列操作的。



## 思考题

最后请你来思考两个问题。

我在这一节课里面只说了，可以在 Redis 上控制键值对数量来间接控制内存使用量。实际上要直接控制某个业务的内存使用量也不是不行，就是非常麻烦，你觉得可以怎么做？

在 Redis 控制内存使用量的方案中，我一直说要使用 lua 脚本，因为有并发问题。那么你能想到有什么样的并发问题吗？

欢迎你把你的思考分享到评论区和我互动，也欢迎你把这节课的内容分享给需要的朋友，我们下节课再见！

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

## 精选留言 3