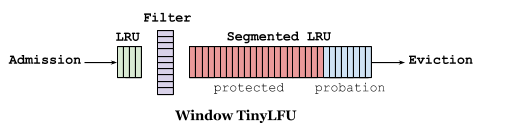
Caching with Caffeine[1]

## Enviction策略

Least Recently Used（LRU）简单，高效且通常可以满足命中率要求。

**现代Cache考虑Recent past并且考虑Recency和频率**。一种保留历史的方法是使用popularity sketch方法，如**CountMin Sketch**(使用矩阵计数和多hash函数)。

Window TinyLFU(W-TinyLFU)用CountMin Sketch作为过滤器，如果新entry比将被移除的entry有更高的访问频率，那么就会将新entry加入Cache。准入窗口给新entry机会积累数据，这样可以避免应对突发流量时连续发生miss（Cache中如果保留的是高频访问数据，那么就会发生这样的情况），为了刷新历史，启动一个进程（也可能不是）周期性对计数进行减半。



1. Window TinyLFU

**备注：LRU保留最近访问；Filter保留频次（次数）；Segmented LRU保存数据。**

W-TinyLFU使用分段LRU（SLRU）策略长期缓存数据。数据首先进入probationary段，随着后续访问进入protected段。当protected段满时，数据会进入probationary段。触发probationary段中数据剔除出Cache。

其他优化如Climbing mountain，可类比最小二乘法这样的迭代优化算法，可参见。

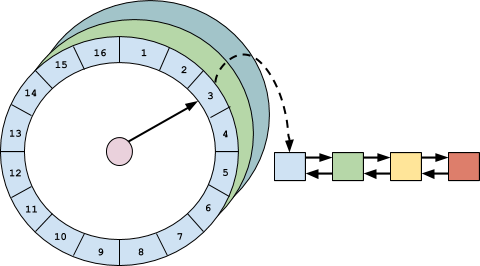
## 过期策略

使用独立线程周期性对cache进行过期处理（回收内存）往往优于对数据进行过期时间排序回收内存。

Caffeine使用不同策略（固定周期处理过期即可）。

写顺序队列和访问顺序队列。过期性能优化，如Hierarchical timing wheel参见图2。文件参见[4]。

注：最低层wheel最高的时间精度。如最小时间刻度是u，大小是n，那么高一级的最小时间刻度就是n2 u，依次类推。具体例子参见[5]。



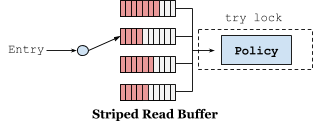
1. Hierarchical timing wheel

## 并发

并发访问往往要写某些共享状态，传统上用一个独立的锁保护整个cache。继而将cache分成小的独立区域，优化性能。但是这样的优化有限，因为热数据使得某些锁竞争更激烈。当锁竞争成为瓶颈时，下一步经典的解决方法是使用**随机采样**或者**基于FIFO**的剔除策略。这些技术拥有好的读性能，弱的写性能，并且不易选择需要剔除的数据。

此外，通过借鉴数据库理论可以使用commit log来扩展写性能。Commit log不会立即修改数据结构，只把更新写入log并异步方式批量重放。同样的思想可以应用于Cache，即执行hash表操作，将操作记录到buffer，必要时调度执行重放。这种策略同样需要锁保护，或者更精确的讲是try lock，但是锁竞争转移到了追加log buffer。

Caffeine中，读写使用不同的commit log buffer。每个访问都记录到条纹环形buffer中，而选择哪个条纹则由线程相关的hash函数计算，竞争发生时增加条纹数。环形buffer满时，会丢弃后续新的添加请求。但Caffeine仍旧会返回缓存值给调用者。W-TinyLFU能够确认希望保留的热数据，因此丢失访问记录不会有什么影响。通过使用线程相关的hash函数，热数据引起的竞争也可以通过扩展访问负载方式避免（有点疑惑，最终数据不就是在一个位置吗？这个是用于统计数据？应该是的）。



1. 条纹读Buffer

由于数据丢失是不可接受的，所以对于写请求则使用传统的并发队列，如果ring buffer满会立刻调度drain操作。

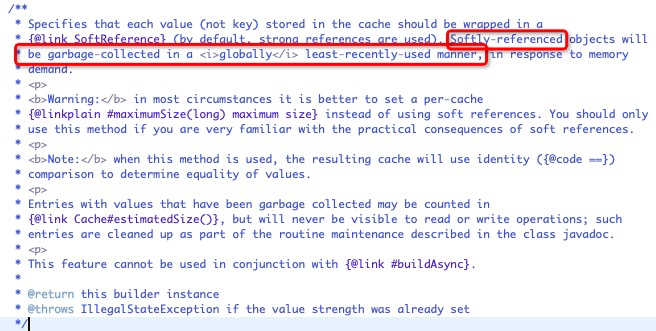
两类buffers都使用多生产者/单生产者模式以简单地实现高效算法。

区分读写buffer以及对应的优化可能引起对某一entry的操作记录失序。如果以错误的方式重放可能会保存dangling引用。这可以通过定义entry生命周期的状态机解决。



1. Entry状态机

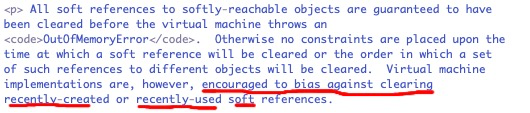
其他：



1. gc与软引用

Caffeine.java中有个注释，具体见红框部分。

疑问：gc和软引用，要具体看看gc。



1. SoftReference.class注释
2. <http://highscalability.com/blog/2016/1/25/design-of-a-modern-cache.html>
3. <http://highscalability.com/blog/2019/2/25/design-of-a-modern-cachepart-deux.html>
4. <https://en.wikipedia.org/wiki/Hill_climbing>
5. <http://www.cs.columbia.edu/~nahum/w6998/papers/sosp87-timing-wheels.pdf>
6. <https://www.confluent.io/blog/apache-kafka-purgatory-hierarchical-timing-wheels/>