1. **caffeine介绍**
2. **缓存使用场景**

在使用缓存之前，首先需要确认你的项目是否真的需要缓存。使用缓存会引入的一定的技术复杂度，后文也将会一一介绍这些复杂度。一般来说从两个方面来个是否需要使用缓存:

* CPU占用: 如果你有某些应用需要消耗大量的cpu去计算，比如正则表达式，如果你使用正则表达式比较频繁，而其又占用了很多CPU的话，那你就应该使用缓存将正则表达式的结果给缓存下来。
* 数据库或则网络IO占用: 如果你发现你的数据库连接池比较空闲，那么不应该用缓存。但是如果数据库连接池比较繁忙，甚至经常报出连接不够的报警，那么是时候应该考虑缓存了。

如果并没有上述两个问题，那么你不必为了增加缓存而缓存。

1. **选择合适的缓存**

缓存分为本地缓存和分布式缓存两种。

对于本地缓存来说，如果不需要淘汰算法则选择ConcurrentHashMap，如果需要淘汰算法和一些丰富的API，这里推荐选择Caffeine。

分布式缓存，这里就不介绍了。

实际应用系统一般都会有多级缓存。

1. **Caffeine 的使用**

Caffeine的API和Guava非常的相似，下面给出一个创建cache的例子：

public static void main(String[] args) throws

InterruptedException {  
 LoadingCache<String,String> loadingCache = Caffeine.*newBuilder*()  
 .maximumSize(3)  
 .build(new CacheLoader<String, String>() {  
 @Override  
 public String load(String key) throws Exception {  
 Thread.*sleep*(1000); //休眠1s，模拟加载数据  
 System.*out*.println(key + " is loaded from a cacheLoader!");  
 return key + "'s value";  
 }  
 @Override  
 public Map<String, String> loadAll(Iterable<? extends String> keys) throws Exception {  
 Thread.*sleep*(1000); //休眠1s，模拟加载数据  
 System.*out*.println(keys + " is loaded from a cacheLoader!");  
 Map<String, String> map = new HashMap<String, String>();  
 for(String key : keys) {  
 map.put(key, keys + "'s value");  
 }  
 return map;  
 }  
 });//在构建时指定自动加载器  
  
  
 try {  
 loadingCache.get("key1");  
 loadingCache.get("key2");  
 loadingCache.get("key3");  
 Thread.*sleep*(100000);  
 loadingCache.get("key1");  
 Thread.*sleep*(100000);  
 loadingCache.get("key2");  
 Thread.*sleep*(100000);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
}

1. **Caffeine原理[1]**
2. **lru和lfu的缺陷**

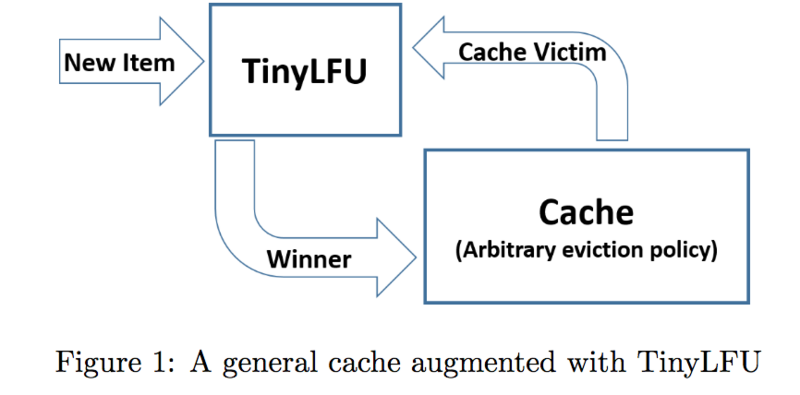
lru是最常见的缓存，认为最近访问的优先级最高，优点是存储空间小，缺点也很明显：可能被突发的稀疏流量冲击，导致真正缓存的数据被evict。其优化方案如Innodb和linux中的分代Lru，但是对缓存优先级定义仍然有缺陷。

lfu认为访问频次越高优先级越高，理论上来说对缓存的定义比lru更贴切，但是缺点也很明显：

* 无法应对数据过期，新来的数据由于频次不及过期的高频数据，无法加入到缓存中导致缓存失效；
* lfu具体实现上需要三个Map：key-value，key-count以及count-keys，占用很大内存空间，所以大部分缓存失效都不会用这种完备的Lfu。

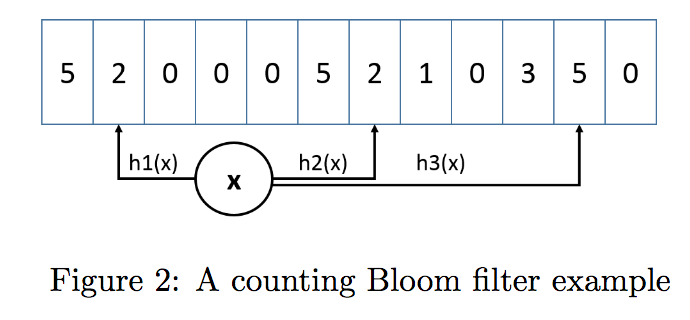
1. **lfu优化——tinyLFU**
2. **基本结构**

TinyLFU的淘汰策略是：一个新元素加入，判断能否替换一个旧元素，判断的依据是能否提升缓存命中率。显然，我们试图利用LFU来辅助Cache的缓存的LRU淘汰。



1. **countMin Sketch**

我们不希望记录LFU中每个key对应的count，那么可以参考布隆过滤器，利用k次hash，将对应的hash槽加1，那么对于key的多个hash槽的count，取最小值作为lfu近似count。这样可以大大减少空间复杂度。这种算法称为CountMin Sketch。



1. **lfu的保鲜机制**

TinyLFU使用reset机制来保证sketch中的数据尽可能最新。每增加一个新的元素到approximation sketches，会增加一个计数值size，一旦计数值size达到了一个预设的采样尺寸（W）：

* 就会将频率采样（CBF）维护的所有计数值除以2（可以使用高效的寄存器位移来实现）
* 同时，size=size/2

论文也花了较大的篇幅通过数学归纳法来证明了这种Reset机制的正确性，且评估了其存在的截断错误（3会被reset为1，而非1.5），并且得出了以下结论：

* reset在constant distribution频率分布下完全正确，且可以应对流量频率的变化（数学归纳法证明，感兴趣的可以参考原文3.3.1）。
* 采样数W越大，截断错误的带来的影响越小。

1. **tinyLFU的内存优化**
2. **counter**

对tinyLFU，其reset采样大小为w，那么显然counter就不会无限扩大，其大小最大为w，对应数据大小为log(w) bit。设缓存大小为c，对于新的需要进入cache的元素，其计数器大小本身不需要达到log(w)，因为我们只需要比较在缓存中需要丢弃元素的计数（频率大于等于1/c才能保持在缓存里，而reset又会降低一半的频率），与新元素计数即可，而新元素的计数器只需要保证最大为w/c即可。例如w=100, c= 10，那么新元素的计数器只需要限制在10以内即可。

1. **doorKeeper**

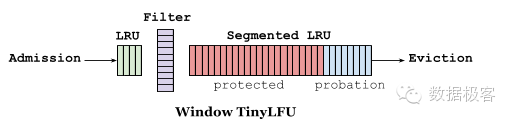
通常会有一些大量只出现一次的对象进入tinyLFU计数，显然这些对象利用count min sketch计算也会增加counter的长度（counter的数据类型可以动态增加）。那么我们可以在进入tinyLFU计算前，加入一个名为doorKeeper的标准布隆过滤器。只有在doorKeeper的元素，才会真正计算器tinyLFU，否则只停留在doorKeeper中，即count为1。

reset会清空doorKeeper。

1. **w-tinyLFU——基于Linux的缓存页优化**

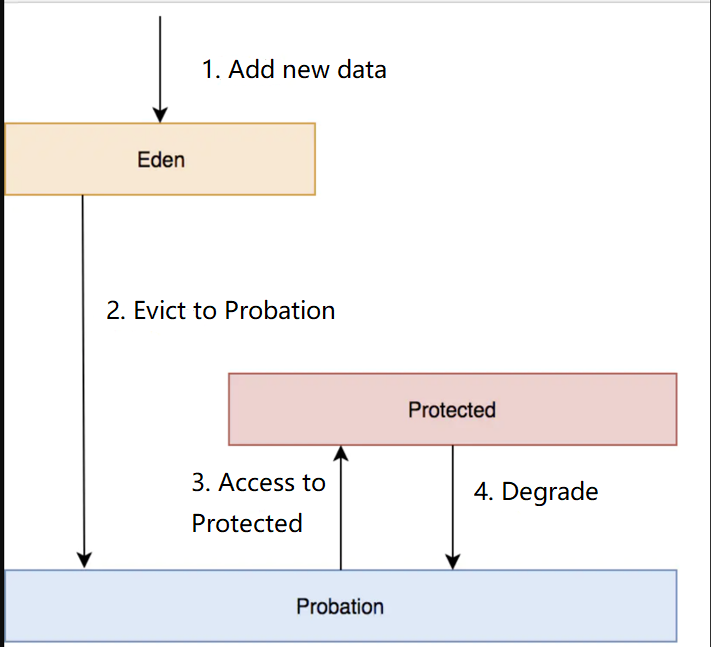
有了tinyLFU，我们就可以使用其作为Lru过期淘汰策略。同时为了保证不会有突发稀疏流量，这部分流量可能无法累计足够的count而进入tinyLFU缓存，从而导致缓存失败。所以在tinyLFU之前我们加入了一个window cache（lru，占用整个w-tinyLFU内存1%）用于处理LRU适合的burst场景，tinyLFU作为filter来执行淘汰策略，主缓存则是一个分段缓存：segmented cache（lru，占用整个w-tinyLFU内存99%）。这个分段缓存分为两个区域，和linux中的active和inactive链表类似。

* probation A1，占用整个分段缓存20%的存储空间，主要用于**执行淘汰**。
  + 如果这里的元素被访问，则将其升级至protected段，如果protected满了则而将protected段最老元素会降级到probation段。
  + 如果从window cache有淘汰元素进入，则需要将probation中的最老元素与其进行count比较来判断到底淘汰谁。
* protected A2，占用整个分段缓存80%的存储空间，主要存储有价值的缓存数据。



1. **基本流程**

* 新元素首先插入window cache。
* window cache满了则执行降级到probation队尾。如果probation满了，则执行**淘汰**。
* 如果访问了probation中的元素，则将其升级到protected；
* protected满了则将其最老元素降级到probation；



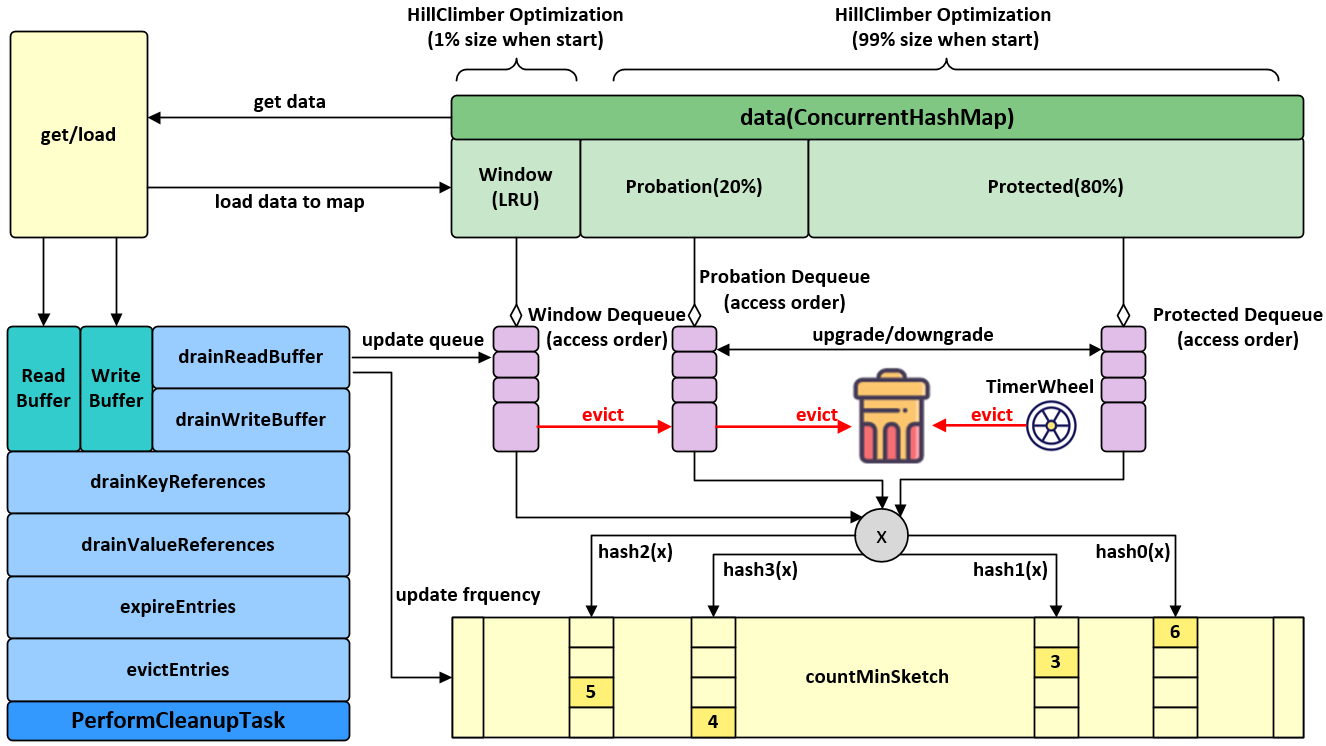
1. **淘汰算法**

probation队列中队首是最Lru中最老的元素，队尾是刚从window cache淘汰来的元素，那么会根据tinyLFU的频度来判断到底淘汰谁（只有这种情况会出现淘汰）：

* 如果队尾元素的频度大于队首，那么就直接淘汰队首，
* 当队尾频度小于等于队首，且频度小于5的时候，直接将队尾淘汰
* 当队尾频度小于等于队首，且频度大于5的时候，通过随机的方式进行淘汰任意一个键。

1. **源码实现**
2. **基本组成**

这里我们以版本2.3.1进行分析。由于是早期版本，BoundedLocalCache并没有doorkeeper和用于调整window cache（在源码里称为Eden）占比大小的Pacer。一般我们使用强引用关系的实现类SSMS，代表S(Strong keys) S(Strong values) M(evicts) S(not weighted).



|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 类型 | 名称 | 含义 |
|  | ConcurrentHashMap<Object,Node<K, V>> | data | map存储数据 |
|  | AccessOrderDeque | accessOrderEdenDeque | eden缓存按照访问顺序的双向队列，队首是最早访问的节点 |
|  | accessOrderProbationDeque | probation缓存按照访问顺序的双向队列，队首是最早访问的节点 |
|  | accessOrderProtectedDeque | protected缓存按照访问顺序的双向队列，队首是最早访问的节点 |
|  | WriteOrderDeque | writeOrderDeque | 按照写顺序记录的双向队列，队首是最早写的节点。用于drainBuffersTask中写后清理过期节点，因为此时可能刚执行完write，没有将数据更新到前面三个accessOrderDequeue中，所以必须手动执行 |
|  | PerformCleanupTask | drainBuffersTask | 异步处理buffer并执行evict |
|  | Consumer<Node<K, V>> | accessPolicy | 处理get请求，默认是onAccess方法 |
|  | Buffer<Node<K, V>> | readBuffer | drainBuffersTask中异步批量处理读请求 |
|  | WriteBuffer | writeBuffer | drainBuffersTask中异步处理写请求 |
|  | NodeFactory | nodeFactory | 生成map中的node节点，默认是eden节点 |
|  | Weigher<K, V> | weigher | 计算节点权重以用于判断是否需要evict |
|  | Lock | evictionLock | 执行evict时的对整个caffeineCache的全局锁，因为各种access queue都不是线程安全的 |

1. **分段缓存大小**

在BoundedLocalCache初始化时会首先设定其eden、probation、protected段的各自大小

@GuardedBy("evictionLock")

void setMaximum(long maximum) {  
 Caffeine.*requireArgument*(maximum >= 0);  
  
 long max = Math.*min*(maximum, *MAXIMUM\_CAPACITY*);  
 long eden = max - (long) (max \* *PERCENT\_MAIN*);  
 long mainProtected = (long) ((max - eden) \* *PERCENT\_MAIN\_PROTECTED*);  
  
 lazySetMaximum(max);  
 lazySetEdenMaximum(eden);  
 lazySetMainProtectedMaximum(mainProtected);  
  
 if ((frequencySketch() != null) && !isWeighted() && (weightedSize() >= (max >>> 1))) {  
 // Lazily initialize when close to the maximum size  
 frequencySketch().ensureCapacity(max);  
 }  
}

1. **缓存高性能读写——WAL和mysql的insertBuffer**

缓存进行读写之后（读的话，已经存在则直接返回，不存在则load数据，保存，再返回；写的话，则直接插入或更新），需要维护一些淘汰策略相关的额外数据，诸如：

* 计算和比较数据的是否过期
* 统计频率（像LFU或其变种）
* 维护read queue和write queue
* 淘汰符合条件的数据

缓存数据的读写都伴随着缓存状态的变更，Guava Cache的做法是把缓存状态的变更和读写操作一起同步处理，在一个同步加锁的操作中完成。虽然Guava Cache巧妙地利用了JDK的ConcurrentHashMap（JDk7的分段锁或者JDK8的无锁CAS）来降低锁的密度，达到提高并发度的目的，同时使用recencyQueue来避免读操作的锁。但是，对于一些热点数据，这种做法还是避免不了频繁的锁竞争。

Caffeine借鉴了数据库系统的**WAL**（Write-Ahead Logging）思想，即先写日志再执行操作，这种思想同样适合缓存，例如**mysql**中的insertbuffer。在执行读写操作时：

1. 先将数据写入map，map缓存优先级最高必须**同步**完成；
2. 然后**同步**先把操作记录在缓冲区；
3. 最后在合适的时机**异步**、批量地执行缓冲区中的内容。

在执行缓冲区的内容时，也是需要在缓冲区加上同步锁的，不然存在并发问题，只不过这样就可以把 对锁的竞争从缓存数据转移到对缓冲区上。

cache的访问主要是读写。所以buffer主要包含：ReadBuffer和WriteBuffer。

1. **数据读取**

我们执行cache.get()方法读取数据，使用了LocalCache接口的computeIfAbsent

@Override

public V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction,  
 boolean recordStats, boolean recordLoad) {  
 *requireNonNull*(key);  
 *requireNonNull*(mappingFunction);  
 long now = expirationTicker().read();  
  
 // An optimistic fast path to avoid unnecessary locking  
 Node<K, V> node = data.get(nodeFactory.newLookupKey(key));  
 if (node != null) {  
 V value = node.getValue();  
 if ((value != null) && !hasExpired(node, now)) {  
 afterRead(node, now, /\* recordHit \*/ true);  
 return value;  
 }  
 }  
 if (recordStats) {  
 mappingFunction = statsAware(mappingFunction, recordLoad);  
 }  
 Object keyRef = nodeFactory.newReferenceKey(key, keyReferenceQueue());  
 return doComputeIfAbsent(key, keyRef, mappingFunction, now);  
}

如果数据存在，我们就将先执行afterRead，将异步任务插入readBuffer中返回结果即可。此需要更新该节点的访问顺序和计算频率，但由于没有涉及缓存溢出，所以可以在drainStatus等于IDLE时，不需要立即执行drainbuffer.

void afterRead(Node<K, V> node, long now, boolean recordHit) {

if (recordHit) {  
 statsCounter().recordHits(1);  
 }  
 node.setAccessTime(now);  
  
 boolean delayable = skipReadBuffer() || (readBuffer.offer(node) != Buffer.*FULL*);  
 if (shouldDrainBuffers(delayable)) {  
 scheduleDrainBuffers();  
 }  
 refreshIfNeeded(node, now);  
}

boolean shouldDrainBuffers(boolean delayable) {  
 switch (drainStatus()) {  
 case *IDLE*:  
 return !delayable;  
 case *REQUIRED*:  
 return true;  
 case *PROCESSING\_TO\_IDLE*:  
 case *PROCESSING\_TO\_REQUIRED*:  
 return false;  
 default:  
 throw new IllegalStateException();  
 }  
}

1. **数据加载**

数据加载主要执行doComputeIfAbsent方法，这里会执行ConcurrentHashMap的data的compute方法，其中会调用我们在一.3中重写的load和loadAll方法，加载完后，注意这里会立即写入到data的map中，而不是等待drainBuffersTask，这是由于缓存优先级最高，而其访问、写入顺序和频率更新相比不那么重要，特别是在不需要evict时。

默认新生成的节点都是eden节点。

V doComputeIfAbsent(K key, Object keyRef,

Function<? super K, ? extends V> mappingFunction, long now) {  
 @SuppressWarnings("unchecked")  
 V[] oldValue = (V[]) new Object[1];  
 @SuppressWarnings("unchecked")  
 V[] newValue = (V[]) new Object[1];  
 @SuppressWarnings("unchecked")  
 K[] nodeKey = (K[]) new Object[1];  
 @SuppressWarnings({"unchecked", "rawtypes"})  
 Node<K, V>[] removed = new Node[1];  
  
 int[] weight = new int[2]; // old, new  
 RemovalCause[] cause = new RemovalCause[1];  
 Node<K, V> node = data.compute(keyRef, (k, n) -> {  
 if (n == null) {

//执行重写的load或loadAll  
 newValue[0] = mappingFunction.apply(key);  
 if (newValue[0] == null) {  
 return null;  
 }  
 weight[1] = weigher.weigh(key, newValue[0]);  
 return nodeFactory.newNode(key, keyReferenceQueue(),  
 newValue[0], valueReferenceQueue(), weight[1], now);  
 }  
  
 synchronized (n) {  
 nodeKey[0] = n.getKey();  
 weight[0] = n.getWeight();  
 oldValue[0] = n.getValue();  
 if ((nodeKey == null) || (oldValue[0] == null)) {  
 cause[0] = RemovalCause.*COLLECTED*;  
 } else if (hasExpired(n, now)) {  
 cause[0] = RemovalCause.*EXPIRED*;  
 n.setAccessTime(now);  
 n.setWriteTime(now);  
 } else {  
 return n;  
 }  
  
 writer.delete(nodeKey[0], oldValue[0], cause[0]);  
 newValue[0] = mappingFunction.apply(key);  
 if (newValue[0] == null) {  
 removed[0] = n;  
 n.retire();  
 return null;  
 }  
 weight[1] = weigher.weigh(key, newValue[0]);  
 n.setValue(newValue[0], valueReferenceQueue());  
 n.setWeight(weight[1]);  
 return n;  
 }  
 });  
  
 if (node == null) {  
 if (removed[0] != null) {  
 afterWrite(null, new RemovalTask(removed[0]), now);  
 }  
 return null;  
 }  
 if (cause[0] != null) {  
 if (hasRemovalListener()) {  
 notifyRemoval(nodeKey[0], oldValue[0], cause[0]);  
 }  
 statsCounter().recordEviction(weight[0]);  
 }  
 if (newValue[0] == null) {  
 afterRead(node, now, /\* recordHit \*/ true);  
 return oldValue[0];  
 }  
 if ((oldValue[0] == null) && (cause[0] == null)) {

//执行drainBuffersTask  
 afterWrite(node, new AddTask(node, weight[1]), now);  
 } else {  
 int weightedDifference = (weight[1] - weight[0]);  
 afterWrite(node, new UpdateTask(node, weightedDifference), now);  
 }  
  
 return newValue[0];  
}

将节点插入data后就执行afterWrite，此时由于写操作比较重要可能会涉及缓存的evict，需要尽快执行，所以只要插入writeBuffer成功就会异步执行drainBuffersTask

void afterWrite(@Nullable Node<K, V> node, Runnable task, long

now) {  
 if (node != null) {  
 node.setAccessTime(now);  
 node.setWriteTime(now);  
 }  
 if (buffersWrites()) {  
 for (int i = 0; i < *WRITE\_BUFFER\_RETRIES*; i++) {  
 if (writeBuffer().offer(task)) {  
 scheduleAfterWrite();  
 return;  
 }  
 scheduleDrainBuffers();  
 }  
  
 // The maintenance task may be scheduled but not running due to all of the executor's threads  
 // being busy. If all of the threads are writing into the cache then no progress can be made  
 // without assistance.  
 try {  
 performCleanUp(task);  
 } catch (RuntimeException e) {  
 *logger*.log(Level.*SEVERE*, "Exception thrown when performing the maintenance task", e);  
 }  
 }  
}

void scheduleAfterWrite() {  
 for (;;) {  
 switch (drainStatus()) {  
 case *IDLE*:  
 casDrainStatus(*IDLE*, *REQUIRED*);  
 scheduleDrainBuffers();  
 return;  
 case *REQUIRED*:  
 scheduleDrainBuffers();  
 return;  
 case *PROCESSING\_TO\_IDLE*:  
 if (casDrainStatus(*PROCESSING\_TO\_IDLE*, *PROCESSING\_TO\_REQUIRED*)) {  
 return;  
 }  
 continue;  
 case *PROCESSING\_TO\_REQUIRED*:  
 return;  
 default:  
 throw new IllegalStateException();  
 }  
 }  
}

1. **异步整理任务**
2. **组成**

异步任务主要包括：

* drainReadBuffer()：处理readBuffer中的任务
* drainWriteBuffer()： 处理writeBuffer中的任务
* drainKeyReferences(): 处理非强引用的Key
* drainValueReferences()：处理非强引用的Value
* expireEntries()：对缓存节点执行过期操作，如果过期则removalTask
* evictEntries()：清理缓存，主要针对过期和缓存满了情况

其调用如下：

void scheduleDrainBuffers() {

if (drainStatus() >= *PROCESSING\_TO\_IDLE*) {  
 return;  
 }  
 if (evictionLock.tryLock()) {  
 try {  
 int drainStatus = drainStatus();  
 if (drainStatus >= *PROCESSING\_TO\_IDLE*) {  
 return;  
 }  
 lazySetDrainStatus(*PROCESSING\_TO\_IDLE*);  
 executor().execute(drainBuffersTask);  
 } catch (Throwable t) {  
 *logger*.log(Level.*WARNING*, "Exception thrown when submitting maintenance task", t);  
 performCleanUp(/\* ignored \*/ null);  
 } finally {  
 evictionLock.unlock();  
 }  
 }  
}

主要内容源码如下，可见第一件事就是加evictionLock以保证各种accessOrderDequeue的修改的线程安全。

void performCleanUp(@Nullable Runnable task) {

evictionLock.lock();  
 try {  
 lazySetDrainStatus(*PROCESSING\_TO\_IDLE*);  
 if (task != null) {  
 task.run();  
 }  
 maintenance();  
 } finally {  
 if ((drainStatus() != *PROCESSING\_TO\_IDLE*) || !casDrainStatus(*PROCESSING\_TO\_IDLE*, *IDLE*)) {  
 lazySetDrainStatus(*REQUIRED*);  
 }  
 evictionLock.unlock();  
 }  
}

@GuardedBy("evictionLock")  
void maintenance() {  
 drainReadBuffer();  
  
 drainWriteBuffer();  
 drainKeyReferences();  
 drainValueReferences();  
  
 expireEntries();  
 evictEntries();  
}

1. **处理读操作任务与主缓存升降级**

对于read操作，我们需要更新：

* 访问频率
* accessOrderQueue

@GuardedBy("evictionLock")

void onAccess(Node<K, V> node) {  
 if (evicts()) {  
 K key = node.getKey();  
 if (key == null) {  
 return;  
 }  
 frequencySketch().increment(key);  
 if (node.inEden()) {  
 *reorder*(accessOrderEdenDeque(), node);  
 } else if (node.inMainProbation()) {  
 reorderProbation(node);  
 } else {  
 *reorder*(accessOrderProtectedDeque(), node);  
 }  
 } else if (expiresAfterAccess()) {  
 *reorder*(accessOrderEdenDeque(), node);  
 }  
}

同时如果当前节点是probation，我们就可以将该节点存入protected列表，这里的思想和linux 页缓存类似。当然如果protected满了则将其最早访问的元素放到probation中

@GuardedBy("evictionLock")

void reorderProbation(Node<K, V> node) {  
 if (!accessOrderProbationDeque().contains(node)) {  
 // Ignore stale accesses for an entry that is no longer present  
 return;  
 } else if (node.getPolicyWeight() > mainProtectedMaximum()) {  
 return;  
 }  
  
 long mainProtectedWeightedSize = mainProtectedWeightedSize() + node.getPolicyWeight();  
 accessOrderProbationDeque().remove(node);  
 accessOrderProtectedDeque().add(node);  
 node.makeMainProtected();  
  
 long mainProtectedMaximum = mainProtectedMaximum();  
 while (mainProtectedWeightedSize > mainProtectedMaximum) {  
 Node<K, V> demoted = accessOrderProtectedDeque().pollFirst();  
 if (demoted == null) {  
 break;  
 }  
 demoted.makeMainProbation();  
 accessOrderProbationDeque().add(demoted);  
 mainProtectedWeightedSize -= node.getPolicyWeight();  
 }  
  
 lazySetMainProtectedWeightedSize(mainProtectedWeightedSize);  
}

1. **处理写操作任务**

对于write操作，我们这里执行AddTask

* 更新各个dequeue的weight（缓存节点总数量）
* 更新访问频率
* 更新writeOrderQueue，以方便后面expireEntries
* 更新accessOrderEdenDeque，因为是新插入节点
* 更新节点的访问和写入时间

void drainWriteBuffer() {  
 if (!buffersWrites()) {  
 return;  
 }  
 for (int i = 0; i < *WRITE\_BUFFER\_MAX*; i++) {  
 Runnable task = writeBuffer().poll();  
 if (task == null) {  
 break;  
 }  
 task.run();  
 }  
}

@Override

@GuardedBy("evictionLock")  
public void run() {  
 if (evicts()) {  
 node.setPolicyWeight(weight);  
 long weightedSize = weightedSize();  
 lazySetWeightedSize(weightedSize + weight);  
 lazySetEdenWeightedSize(edenWeightedSize() + weight);  
  
 long maximum = maximum();  
 if (weightedSize >= (maximum >>> 1)) {  
 // Lazily initialize when close to the maximum  
 long capacity = isWeighted() ? data.mappingCount() : maximum;  
 frequencySketch().ensureCapacity(capacity);  
 }  
  
 K key = node.getKey();  
 if (key != null) {  
 frequencySketch().increment(key);  
 }  
 }  
  
 // ignore out-of-order write operations  
 boolean isAlive;  
 synchronized (node) {  
 isAlive = node.isAlive();  
 }  
 if (isAlive) {  
 if (expiresAfterWrite()) {  
 writeOrderDeque().add(node);  
 }  
 if (evicts() || expiresAfterAccess()) {  
 accessOrderEdenDeque().add(node);  
 }  
 }  
  
 // Ensure that in-flight async computation cannot expire  
 if (isComputingAsync(node)) {  
 node.setAccessTime(Long.*MAX\_VALUE*);  
 node.setWriteTime(Long.*MAX\_VALUE*);  
 ((CompletableFuture<?>) node.getValue()).thenRun(() -> {  
 long now = expirationTicker().read();  
 node.setAccessTime(now);  
 node.setWriteTime(now);  
 });  
 }  
}

1. **对缓存节点执行过期操作**

这里会对accessOrderDequeue和writeOrderDequeue分别寻找并移除过期缓存节点。

*/\*\* Expires entries that have expired in the access and write*

*queues. \*/*@GuardedBy("evictionLock")  
void expireEntries() {  
 long now = expirationTicker().read();  
 expireAfterAccessEntries(now);  
 expireAfterWriteEntries(now);  
}

首先来看accessOrderDequeue

@GuardedBy("evictionLock")

void expireAfterAccessEntries(long now) {  
 if (!expiresAfterAccess()) {  
 return;  
 }  
  
 long expirationTime = (now - expiresAfterAccessNanos());  
 expireAfterAccessEntries(accessOrderEdenDeque(), expirationTime, now);  
 if (evicts()) {  
 expireAfterAccessEntries(accessOrderProbationDeque(), expirationTime, now);  
 expireAfterAccessEntries(accessOrderProtectedDeque(), expirationTime, now);  
 }  
}

@GuardedBy("evictionLock")  
void expireAfterAccessEntries(AccessOrderDeque<Node<K, V>> accessOrderDeque,  
 long expirationTime, long now) {  
 for (;;) {  
 Node<K, V> node = accessOrderDeque.peekFirst();  
 if ((node == null) || (node.getAccessTime() > expirationTime)) {  
 return;  
 }  
 evictEntry(node, RemovalCause.*EXPIRED*, now);  
 }  
}

然后来看writeOrderDequeue。 因为此时可能刚执行完write，没有将数据更新到前面三个accessOrderDequeue中，所以必须手动执行

@GuardedBy("evictionLock")

void expireAfterWriteEntries(long now) {  
 if (!expiresAfterWrite()) {  
 return;  
 }  
 long expirationTime = now - expiresAfterWriteNanos();  
 for (;;) {  
 final Node<K, V> node = writeOrderDeque().peekFirst();  
 if ((node == null) || (node.getWriteTime() > expirationTime)) {  
 break;  
 }  
 evictEntry(node, RemovalCause.*EXPIRED*, now);  
 }  
}

1. **数据淘汰**

对于缓存满了的情况我们需要考虑将不那么重要的数据淘汰。

@GuardedBy("evictionLock")

void evictEntries() {  
 if (!evicts()) {  
 return;  
 }  
 int candidates = evictFromEden();  
 evictFromMain(candidates);  
}

淘汰的顺序是先从eden以LRU算法淘汰最早访问的节点至probation。

@GuardedBy("evictionLock")

int evictFromEden() {  
 int candidates = 0;  
 Node<K, V> node = accessOrderEdenDeque().peek();  
 while (edenWeightedSize() > edenMaximum()) {  
 // The pending operations will adjust the size to reflect the correct weight  
 if (node == null) {  
 break;  
 }  
  
 Node<K, V> next = node.getNextInAccessOrder();  
 if (node.getWeight() != 0) {  
 node.makeMainProbation();  
 accessOrderEdenDeque().remove(node);  
 accessOrderProbationDeque().add(node);  
 candidates++;  
  
 lazySetEdenWeightedSize(edenWeightedSize() - node.getPolicyWeight());  
 }  
 node = next;  
 }

如果probation也满了，就会执行淘汰策略。probation队列中队首是最Lru中最老的元素，队尾是刚从window cache淘汰来的元素，那么会根据tinyLFU的频度来判断到底淘汰谁（只有这种情况会出现淘汰）：

* 如果队尾元素的频度大于队首，那么就直接淘汰队首，
* 当队尾频度小于等于队首，且频度小于5的时候，直接将队尾淘汰
* 当队尾频度小于等于队首，且频度大于5的时候，通过随机的方式进行淘汰任意一个键。

@GuardedBy("evictionLock")

void evictFromMain(int candidates) {  
 int victimQueue = *PROBATION*;  
 Node<K, V> victim = accessOrderProbationDeque().peekFirst();  
 Node<K, V> candidate = accessOrderProbationDeque().peekLast();  
 while (weightedSize() > maximum()) {  
 // Stop trying to evict candidates and always prefer the victim  
 if (candidates == 0) {  
 candidate = null;  
 }  
  
 // Try evicting from the protected and eden queues  
 if ((candidate == null) && (victim == null)) {  
 if (victimQueue == *PROBATION*) {  
 victim = accessOrderProtectedDeque().peekFirst();  
 victimQueue = *PROTECTED*;  
 continue;  
 } else if (victimQueue == *PROTECTED*) {  
 victim = accessOrderEdenDeque().peekFirst();  
 victimQueue = *EDEN*;  
 continue;  
 }  
  
 // The pending operations will adjust the size to reflect the correct weight  
 break;  
 }  
  
 // Skip over entries with zero weight  
 if ((victim != null) && (victim.getPolicyWeight() == 0)) {  
 victim = victim.getNextInAccessOrder();  
 continue;  
 } else if ((candidate != null) && (candidate.getPolicyWeight() == 0)) {  
 candidate = candidate.getPreviousInAccessOrder();  
 candidates--;  
 continue;  
 }  
  
 // Evict immediately if only one of the entries is present  
 if (victim == null) {  
 candidates--;  
 Node<K, V> evict = candidate;  
 candidate = candidate.getPreviousInAccessOrder();  
 evictEntry(evict, RemovalCause.*SIZE*, 0L);  
 continue;  
 } else if (candidate == null) {  
 Node<K, V> evict = victim;  
 victim = victim.getNextInAccessOrder();  
 evictEntry(evict, RemovalCause.*SIZE*, 0L);  
 continue;  
 }  
  
 // Evict immediately if an entry was collected  
 K victimKey = victim.getKey();  
 K candidateKey = candidate.getKey();  
 if (victimKey == null) {  
 Node<K, V> evict = victim;  
 victim = victim.getNextInAccessOrder();  
 evictEntry(evict, RemovalCause.*COLLECTED*, 0L);  
 continue;  
 } else if (candidateKey == null) {  
 candidates--;  
 Node<K, V> evict = candidate;  
 candidate = candidate.getPreviousInAccessOrder();  
 evictEntry(evict, RemovalCause.*COLLECTED*, 0L);  
 continue;  
 }  
  
 // Evict immediately if the candidate's weight exceeds the maximum  
 if (candidate.getPolicyWeight() > maximum()) {  
 candidates--;  
 Node<K, V> evict = candidate;  
 candidate = candidate.getPreviousInAccessOrder();  
 evictEntry(evict, RemovalCause.*SIZE*, 0L);  
 continue;  
 }  
  
 // Evict the entry with the lowest frequency  
 candidates--;  
 if (admit(candidateKey, victimKey)) {  
 Node<K, V> evict = victim;  
 victim = victim.getNextInAccessOrder();  
 evictEntry(evict, RemovalCause.*SIZE*, 0L);  
 candidate = candidate.getPreviousInAccessOrder();  
 } else {  
 Node<K, V> evict = candidate;  
 candidate = candidate.getPreviousInAccessOrder();  
 evictEntry(evict, RemovalCause.*SIZE*, 0L);  
 }  
 }  
}

淘汰策略代码如下：

boolean admit(K candidateKey, K victimKey) {

int victimFreq = frequencySketch().frequency(victimKey);  
 int candidateFreq = frequencySketch().frequency(candidateKey);  
 if (candidateFreq > victimFreq) {  
 return true;  
 } else if (candidateFreq <= 5) {  
 // The maximum frequency is 15 and halved to 7 after a reset to age the history. An attack  
 // exploits that a hot candidate is rejected in favor of a hot victim. The threshold of a warm  
 // candidate reduces the number of random acceptances to minimize the impact on the hit rate.  
 return false;  
 }  
 int random = ThreadLocalRandom.*current*().nextInt();  
 return ((random & 127) == 0);  
}

1. **TinyLFU近似频率**
2. **频率计算**

这里为了能够尽可能少的占用内存，FrequencySketch借鉴了bloomFilter的思路通过4次hash来近似得到频率。

@Nonnegative

public int frequency(@Nonnull E e) {  
 if (isNotInitialized()) {  
 return 0;  
 }  
  
 int hash = spread(e.hashCode());  
 int start = (hash & 3) << 2;  
 int frequency = Integer.*MAX\_VALUE*;  
 for (int i = 0; i < 4; i++) {  
 int index = indexOf(hash, i);  
 int count = (int) ((table[index] >>> ((start + i) << 2)) & 0xfL);  
 frequency = Math.*min*(frequency, count);  
 }  
 return frequency;  
}

1. **频率更新**

这里为了能够尽可能少的占用内存，FrequencySketch借鉴了bloomFilter的思路通过4次hash来近似得到频率。

public void increment(@Nonnull E e) {

if (isNotInitialized()) {  
 return;  
 }  
  
 int hash = spread(e.hashCode());  
 int start = (hash & 3) << 2;  
  
 // Loop unrolling improves throughput by 5m ops/s  
 int index0 = indexOf(hash, 0);  
 int index1 = indexOf(hash, 1);  
 int index2 = indexOf(hash, 2);  
 int index3 = indexOf(hash, 3);  
  
 boolean added = incrementAt(index0, start);  
 added |= incrementAt(index1, start + 1);  
 added |= incrementAt(index2, start + 2);  
 added |= incrementAt(index3, start + 3);  
  
 if (added && (++size == sampleSize)) {  
 reset();  
 }  
}

1. **reset**

LFU还需解决的问题就是历史数据的过期，这里采用reset算法，即新增数量size超过门限sampleSize，就执行位运算对所有count除以2，并让size=size/2，从而避免较高频率数据无法被淘汰的问题。

*/\*\* Reduces every counter by half of its original value. \*/*

void reset() {  
 int count = 0;  
 for (int i = 0; i < table.length; i++) {  
 count += Long.*bitCount*(table[i] & *ONE\_MASK*);  
 table[i] = (table[i] >>> 1) & *RESET\_MASK*;  
 }  
 size = (size >>> 1) - (count >>> 2);  
}

sampleSize的设定其实和我们初始化cache相关，这里以强引用SSMS cache为例，可以看出sampleSize在大多数情况下等于min(设定的最大缓存数量，设定的初始缓存数量) \* 10

SSMS(Caffeine<K, V> builder, CacheLoader<? super K, V>

acheLoader, boolean async) {  
 super(builder, (CacheLoader<K, V>) cacheLoader, async);  
 this.sketch = new FrequencySketch<K>();  
 if (builder.hasInitialCapacity()) {  
 long capacity = Math.*min*(builder.getMaximum(), builder.getInitialCapacity());  
 this.sketch.ensureCapacity(capacity);  
 }  
 this.accessOrderEdenDeque = new AccessOrderDeque<Node<K, V>>();  
 this.accessOrderProbationDeque = new AccessOrderDeque<Node<K, V>>();  
 this.accessOrderProtectedDeque = new AccessOrderDeque<Node<K, V>>();  
 this.writeBuffer = new WriteBuffer<>(*WRITE\_BUFFER\_MIN*, *WRITE\_BUFFER\_MAX*);  
}

public void ensureCapacity(@Nonnegative long maximumSize) {  
 Caffeine.*requireArgument*(maximumSize >= 0);  
 int maximum = (int) Math.*min*(maximumSize, Integer.*MAX\_VALUE* >>> 1);  
 if ((table != null) && (table.length >= maximum)) {  
 return;  
 }  
  
 table = new long[(maximum == 0) ? 1 : *ceilingNextPowerOfTwo*(maximum)];  
 tableMask = Math.*max*(0, table.length - 1);  
 sampleSize = (maximumSize == 0) ? 10 : (10 \* maximum);  
 if (sampleSize <= 0) {  
 sampleSize = Integer.*MAX\_VALUE*;  
 }  
 size = 0;  
}

**参考**

1. [mysql之innodb-9-lru优化](mysql之innodb-9-lru优化.docx)
2. [深入剖析来自未来的缓存-Caffeine](https://cloud.tencent.com/developer/article/1520182)
3. [Caffeine高性能缓存设计](https://blog.csdn.net/u010853261/article/details/105587699/)