1. **Guava cache介绍**
2. **基本介绍**

Guava Cache就是由Google大神开发的一个本地缓存组件。其可以指定缓存的容量大小或者指定缓存的缓存项大小，能够设置缓存项目的过期时间，能够设置key或者value为弱引用。与此同时Guava Cache能够根据这些设置在合适的时候对缓存项进行回收。

Guava Cache本身几乎没有主动回收的操作，即不会自己看线程来对不符合条件的（如过期）的缓存项进行回收。几乎都是被动回收，即当读取某个缓存项的时候才去判断它是否过期等，然后再回收，并重新加载新的缓存项。

其实Guava Cache本身也提供了invalidate方法来让使用者自己回收某些缓存项，但是笔者并不建议在生产环境中频繁地执行这样操作。因为invalidate操作与缓存读写有锁竞争，频繁的操作会影响线上缓存的读写性能。

1. **高效读写**

参考了jdk1.7的concurrent HashMap.

V get(K key, CacheLoader<? super K, V> loader) throws

ExecutionException {

int hash = hash(*checkNotNull*(key));  
 return segmentFor(hash).get(key, hash, loader);  
}

1. **简单使用**

public static void main(String[] args) throws

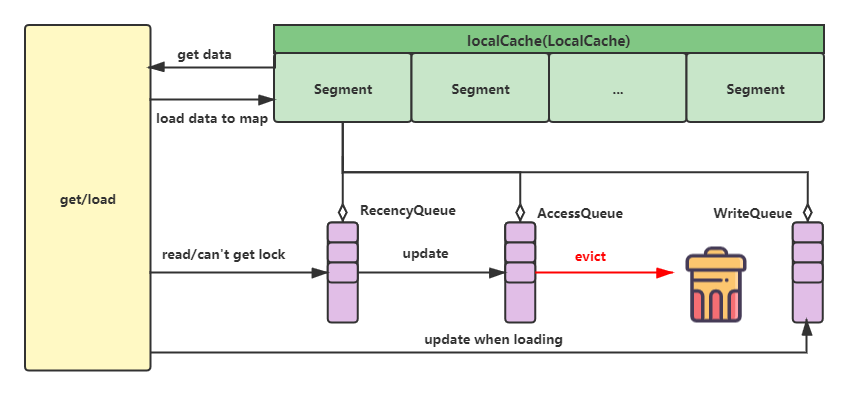
InterruptedException {  
 LoadingCache<String,String> loadingCache = CacheBuilder.*newBuilder*()  
 .maximumSize(3)  
 .build(new CacheLoader<String, String>() {  
 @Override  
 public String load(String key) throws Exception {  
 Thread.*sleep*(1000); //休眠1s，模拟加载数据  
 System.*out*.println(key + " is 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");  
 } catch (ExecutionException e) {  
 e.printStackTrace();  
 }  
}

1. **缓存原理**
2. **LRU基本实现**

设计LRU的最基本的解法就是双端链表+哈希表。基本思路如下：

1. 哈希表用于记录Key-value
2. 双端链表的head用于存储最新被get或put的key-value节点。Tail用于记录最近最不常用（LRU）的节点，如果下面有新的key-value插入，如果此时哈希表已存储数量等于初始设定的LRU的capacity，那么就将tail处的节点从哈希表中移除，将tail的前节点置为新的tail。
3. 反之亦然，我们也可以将最新的数据插入到队列尾部，老数据保留在队列head，guava就是使用这种思路。

代码可以参见[《95-leetcode-No146LRU设计》](../数据结构和算法/95-leetcode-No146LRU设计.docx)



1. **Guava Cache实现**

Guava Cache也是参照了双端链表+哈希表的设计，只是在具体实现时考虑了并发问题，引入了分段锁等提升并发读写性能的优化。

在Guava Cache的LRU实现中，它的双向链表并不是全局的(即这个那个Guava Cache只有一个)。而是每个Segment(ConcurrentHashMap中的概念)中都有。其中一共涉及到三个Queue其中包括：

* AccessQueue：非线程安全的LRU本身双向链表，负责存储对元素的读取行为记录
* WriteQueue：非线程安全的LRU本身双向链表，WriteAccess则负责对元素的写入行为进行记录
* RecencyQueue：普通CocurrentLinkedQueue。在不需要或者无法获取锁的读场景下记录LRU的顺序，在真正获取锁时将状态更新到到accessQueue，从而保证读缓存不需要加锁或加锁等待，提高读并发的能力

1. **AccessQueue**

AccessQueue实现非常简单，就是一个普通的双端链表，每次add(offer)则将entry插入到链表tail（head之前），遍历从head开始，在LRU移除时会首先移除先插入队列（head之后）的节点。但需要注意的是，AccessQueue不是线程安全，需要获取锁才能进行读写，但对于读的情况，这样的效率显然不能接受，所以才会有RecencyQueue

static final class AccessQueue<K, V> extends

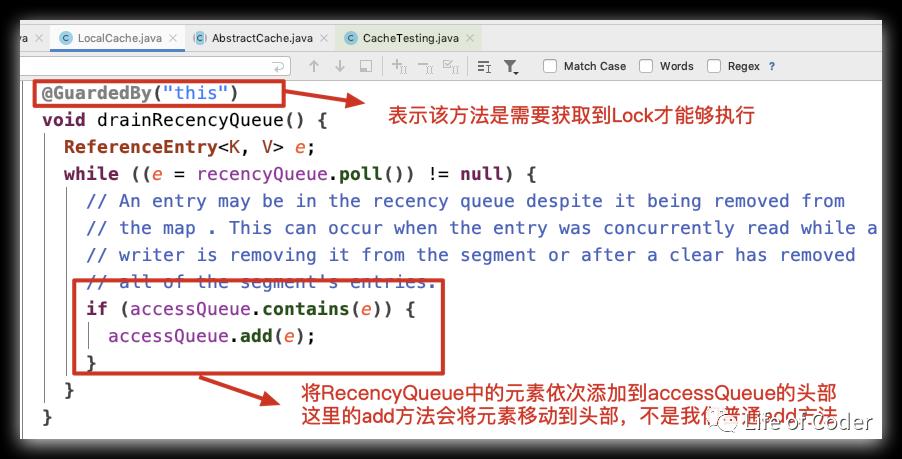
AbstractQueue<ReferenceEntry<K, V>> {  
 final ReferenceEntry<K, V> head =  
 new AbstractReferenceEntry<K, V>() {  
  
 @Override  
 public long getAccessTime() {  
 return Long.*MAX\_VALUE*;  
 }  
  
 @Override  
 public void setAccessTime(long time) {}  
  
 @Weak ReferenceEntry<K, V> nextAccess = this;  
  
 @Override  
 public ReferenceEntry<K, V> getNextInAccessQueue() {  
 return nextAccess;  
 }  
  
 @Override  
 public void setNextInAccessQueue(ReferenceEntry<K, V> next) {  
 this.nextAccess = next;  
 }  
  
 @Weak ReferenceEntry<K, V> previousAccess = this;  
  
 @Override  
 public ReferenceEntry<K, V> getPreviousInAccessQueue() {  
 return previousAccess;  
 }  
  
 @Override  
 public void setPreviousInAccessQueue(ReferenceEntry<K, V> previous) {  
 this.previousAccess = previous;  
 }  
 };  
  
 // implements Queue  
  
 @Override  
 public boolean offer(ReferenceEntry<K, V> entry) {  
 // unlink  
 *connectAccessOrder*(entry.getPreviousInAccessQueue(), entry.getNextInAccessQueue());  
  
 // add to tail  
 *connectAccessOrder*(head.getPreviousInAccessQueue(), entry);  
 *connectAccessOrder*(entry, head);  
  
 return true;  
 }  
  
 @Override  
 public ReferenceEntry<K, V> peek() {  
 ReferenceEntry<K, V> next = head.getNextInAccessQueue();  
 return (next == head) ? null : next;  
 }  
  
 @Override  
 public ReferenceEntry<K, V> poll() {  
 ReferenceEntry<K, V> next = head.getNextInAccessQueue();  
 if (next == head) {  
 return null;  
 }  
  
 remove(next);  
 return next;  
 }  
  
 @Override  
 @SuppressWarnings("unchecked")  
 public boolean remove(Object o) {  
 ReferenceEntry<K, V> e = (ReferenceEntry<K, V>) o;  
 ReferenceEntry<K, V> previous = e.getPreviousInAccessQueue();  
 ReferenceEntry<K, V> next = e.getNextInAccessQueue();  
 *connectAccessOrder*(previous, next);  
 *nullifyAccessOrder*(e);  
  
 return next != NullEntry.*INSTANCE*;  
 }  
  
 @Override  
 @SuppressWarnings("unchecked")  
 public boolean contains(Object o) {  
 ReferenceEntry<K, V> e = (ReferenceEntry<K, V>) o;  
 return e.getNextInAccessQueue() != NullEntry.*INSTANCE*;  
 }  
  
 @Override  
 public boolean isEmpty() {  
 return head.getNextInAccessQueue() == head;  
 }  
  
 @Override  
 public int size() {  
 int size = 0;  
 for (ReferenceEntry<K, V> e = head.getNextInAccessQueue();  
 e != head;  
 e = e.getNextInAccessQueue()) {  
 size++;  
 }  
 return size;  
 }  
  
 @Override  
 public void clear() {  
 ReferenceEntry<K, V> e = head.getNextInAccessQueue();  
 while (e != head) {  
 ReferenceEntry<K, V> next = e.getNextInAccessQueue();  
 *nullifyAccessOrder*(e);  
 e = next;  
 }  
  
 head.setNextInAccessQueue(head);  
 head.setPreviousInAccessQueue(head);  
 }  
  
 @Override  
 public Iterator<ReferenceEntry<K, V>> iterator() {  
 return new AbstractSequentialIterator<ReferenceEntry<K, V>>(peek()) {  
 @Override  
 protected ReferenceEntry<K, V> computeNext(ReferenceEntry<K, V> previous) {  
 ReferenceEntry<K, V> next = previous.getNextInAccessQueue();  
 return (next == head) ? null : next;  
 }  
 };  
 }  
}

1. **recencyQueue:** 并发读优化

既然已经有了AccessQueue我们就能够知道元素的访问顺序了，从而很容易实现LRU算法了。为什么还需要RecentQueue这个CocurrentLinkedQueue呢？

因为上面我们已经提到到过，AccessQueue被设计成了线程不安全，因此必须要在获取到Segment中的Lock的时候才能访问。设想一下当我们访问元素的时候需要怎么操作才能够确保被访问的元素在AccessQueue中能够被移动到前面去？很明显为了实现着功能，我们必须在每次访问元素的时候都需要获取Segment中的Lock，然后才能够安全地将元素移动AccessQueue的前面去。这样功能我们是实现了，但是每次访问元素的时候我们都需要获取锁。这样就破坏了ConcurrentHashMap的分段锁的思想（ConcurrentHashMap分段锁思想中get是不需要获取锁的，这样才能够提供高效的读取性能），导致元素的读取变得很慢，性能很低。

因此为了确保Guava Cache的性能，它引入了RecencyQueue这个同步队列。在读取元素的时候，将所有被访问元素添加到RecencyQueue中。因为其是同步队列所以支持并发插入。这样就确保了高性能的读取能力。当在某些场景下获取到锁的时候，就再将RecencyQueue中的元素移动到AccessQueue中。



那么什么场景下能够获取到锁？主要有如下两种情况：

* 对缓存的修改（写入，新增，修改、删除）是必须要获取锁的
* 每次get元素的时候都会尝试获取一下锁(tryLock)，没有竞争的情况下就能够获取成功

如下图：每次get完元素之后都会执行一下如下代码（调用链：segment.get->postReadCleanup->cleanUp->runLockedCleanup）:

所以Guava Cache通过RecentQueue和AccessQueue的结合就实现了在确保get的高性能的场景下还能记录对元素的访问，从而实现LRU算.

void runLockedCleanup(long now) {

if (tryLock()) {  
 try {  
 drainReferenceQueues();  
 expireEntries(now); // calls drainRecencyQueue  
 readCount.set(0);  
 } finally {  
 unlock();  
 }  
 }  
}

1. **工作流程**

在一.3中的测试我们使用get方法，会进入如下LocalCache.get中。

1. **缓存命中**

如果缓存中有值，那么首先判断其有没有过期，就像redis那样的被动回收策略。

V get(K key, int hash, CacheLoader<? super K, V> loader) throws

ExecutionException {  
 *checkNotNull*(key);  
 *checkNotNull*(loader);  
 try {  
 if (count != 0) { // read-volatile  
 // don't call getLiveEntry, which would ignore loading values  
 ReferenceEntry<K, V> e = getEntry(key, hash);  
 if (e != null) {  
 long now = map.ticker.read();  
 V value = getLiveValue(e, now);  
 if (value != null) {  
 recordRead(e, now);  
 statsCounter.recordHits(1);  
 return scheduleRefresh(e, key, hash, value, now, loader);  
 }  
 ValueReference<K, V> valueReference = e.getValueReference();  
 if (valueReference.isLoading()) {  
 return waitForLoadingValue(e, key, valueReference);  
 }  
 }  
 }  
  
 // at this point e is either null or expired;  
 return lockedGetOrLoad(key, hash, loader);  
 } catch (ExecutionException ee) {  
 Throwable cause = ee.getCause();  
 if (cause instanceof Error) {  
 throw new ExecutionError((Error) cause);  
 } else if (cause instanceof RuntimeException) {  
 throw new UncheckedExecutionException(cause);  
 }  
 throw ee;  
 } finally {  
 postReadCleanup();  
 }  
}

命中的缓存会调用recordLockedRead用于更新accessQueue的这个双端队列，用于维护LRU。

void recordLockedRead(ReferenceEntry<K, V> entry, long now) {

if (map.recordsAccess()) {  
 entry.setAccessTime(now);  
 }  
 accessQueue.add(entry);  
}

accessQueue的add方法实现如下，我们可以看到会将其插入到head的前面，也就是队列尾部，而每次执行evict则会从head开始。

@Override

public boolean offer(ReferenceEntry<K, V> entry) {  
 // unlink  
 *connectAccessOrder*(entry.getPreviousInAccessQueue(), entry.getNextInAccessQueue());  
  
 // add to tail  
 *connectAccessOrder*(head.getPreviousInAccessQueue(), entry);  
 *connectAccessOrder*(entry, head);  
  
 return true;  
}

@Override  
public ReferenceEntry<K, V> poll() {  
 ReferenceEntry<K, V> next = head.getNextInAccessQueue();  
 if (next == head) {  
 return null;  
 }  
  
 remove(next);  
 return next;  
}

1. **缓存未命中**

如果没有值会进入 lockedGetOrLoad方法。此时会再次判断hashMap中是否有缓存（或正在加载的缓存），如果任然没有则使用loadSync从我们重写的CacheLoader.load或loadAll方法获取缓存。

V lockedGetOrLoad(K key, int hash, CacheLoader<? super K, V>

loader) throws ExecutionException {  
 ReferenceEntry<K, V> e;  
 ValueReference<K, V> valueReference = null;  
 LoadingValueReference<K, V> loadingValueReference = null;  
 boolean createNewEntry = true;  
  
 lock();  
 try {  
 // re-read ticker once inside the lock  
 long now = map.ticker.read();  
 preWriteCleanup(now);  
  
 int newCount = this.count - 1;  
 AtomicReferenceArray<ReferenceEntry<K, V>> table = this.table;  
 int index = hash & (table.length() - 1);  
 ReferenceEntry<K, V> first = table.get(index);  
  
 for (e = first; e != null; e = e.getNext()) {  
 K entryKey = e.getKey();  
 if (e.getHash() == hash  
 && entryKey != null  
 && map.keyEquivalence.equivalent(key, entryKey)) {  
 valueReference = e.getValueReference();  
 if (valueReference.isLoading()) {  
 createNewEntry = false;  
 } else {  
 V value = valueReference.get();  
 if (value == null) {  
 enqueueNotification(  
 entryKey, hash, value, valueReference.getWeight(), RemovalCause.*COLLECTED*);  
 } else if (map.isExpired(e, now)) {  
 // This is a duplicate check, as preWriteCleanup already purged expired  
 // entries, but let's accommodate an incorrect expiration queue.  
 enqueueNotification(  
 entryKey, hash, value, valueReference.getWeight(), RemovalCause.*EXPIRED*);  
 } else {  
 recordLockedRead(e, now);  
 statsCounter.recordHits(1);  
 // we were concurrent with loading; don't consider refresh  
 return value;  
 }  
  
 // immediately reuse invalid entries  
 writeQueue.remove(e);  
 accessQueue.remove(e);  
 this.count = newCount; // write-volatile  
 }  
 break;  
 }  
 }  
  
 if (createNewEntry) {  
 loadingValueReference = new LoadingValueReference<>();  
  
 if (e == null) {  
 e = newEntry(key, hash, first);  
 e.setValueReference(loadingValueReference);  
 table.set(index, e);  
 } else {  
 e.setValueReference(loadingValueReference);  
 }  
 }  
 } finally {  
 unlock();  
 postWriteCleanup();  
 }  
  
 if (createNewEntry) {  
 try {  
 // Synchronizes on the entry to allow failing fast when a recursive load is  
 // detected. This may be circumvented when an entry is copied, but will fail fast most  
 // of the time.  
 synchronized (e) {  
 return loadSync(key, hash, loadingValueReference, loader);  
 }  
 } finally {  
 statsCounter.recordMisses(1);  
 }  
 } else {  
 // The entry already exists. Wait for loading.  
 return waitForLoadingValue(e, key, valueReference);  
 }  
}

1. **缓存新增与丢弃**

在完成缓存加载后，我们需要判断是否达到缓存最大大小或最大weight，如果达到则需要执行evict

boolean storeLoadedValue(

K key, int hash, LoadingValueReference<K, V> oldValueReference, V newValue) {  
 lock();  
 try {  
 long now = map.ticker.read();  
 preWriteCleanup(now);  
  
 int newCount = this.count + 1;  
 if (newCount > this.threshold) { // ensure capacity  
 expand();  
 newCount = this.count + 1;  
 }  
  
 AtomicReferenceArray<ReferenceEntry<K, V>> table = this.table;  
 int index = hash & (table.length() - 1);  
 ReferenceEntry<K, V> first = table.get(index);  
  
 for (ReferenceEntry<K, V> e = first; e != null; e = e.getNext()) {  
 K entryKey = e.getKey();  
 if (e.getHash() == hash  
 && entryKey != null  
 && map.keyEquivalence.equivalent(key, entryKey)) {  
 ValueReference<K, V> valueReference = e.getValueReference();  
 V entryValue = valueReference.get();  
 // replace the old LoadingValueReference if it's live, otherwise  
 // perform a putIfAbsent  
 if (oldValueReference == valueReference  
 || (entryValue == null && valueReference != *UNSET*)) {  
 ++modCount;  
 if (oldValueReference.isActive()) {  
 RemovalCause cause =  
 (entryValue == null) ? RemovalCause.*COLLECTED* : RemovalCause.*REPLACED*;  
 enqueueNotification(key, hash, entryValue, oldValueReference.getWeight(), cause);  
 newCount--;  
 }  
 setValue(e, key, newValue, now);  
 this.count = newCount; // write-volatile  
 evictEntries(e);  
 return true;  
 }  
  
 // the loaded value was already clobbered  
 enqueueNotification(key, hash, newValue, 0, RemovalCause.*REPLACED*);  
 return false;  
 }  
 }  
  
 ++modCount;  
 ReferenceEntry<K, V> newEntry = newEntry(key, hash, first);  
 setValue(newEntry, key, newValue, now);  
 table.set(index, newEntry);  
 this.count = newCount; // write-volatile  
 evictEntries(newEntry);  
 return true;  
 } finally {  
 unlock();  
 postWriteCleanup();  
 }  
}

这里我们详细查看下新增缓存，此时recordWrite会将该entry插入accessQueue和writeQueue队尾。

void setValue(ReferenceEntry<K, V> entry, K key, V value, long

now) {

ValueReference<K, V> previous = entry.getValueReference();  
 int weight = map.weigher.weigh(key, value);  
 *checkState*(weight >= 0, "Weights must be non-negative");  
  
 ValueReference<K, V> valueReference =  
 map.valueStrength.referenceValue(this, entry, value, weight);  
 entry.setValueReference(valueReference);  
 recordWrite(entry, weight, now);  
 previous.notifyNewValue(value);  
}

void recordWrite(ReferenceEntry<K, V> entry, int weight, long now) {  
 // we are already under lock, so drain the recency queue immediately  
 drainRecencyQueue();  
 totalWeight += weight;  
  
 if (map.recordsAccess()) {  
 entry.setAccessTime(now);  
 }  
 if (map.recordsWrite()) {  
 entry.setWriteTime(now);  
 }  
 accessQueue.add(entry);  
 writeQueue.add(entry);  
}

以及丢弃缓存，此时我们会从队首取出entry进行移除，具体是从accessQueue中取出需要移除的entry，然后分别删除accessQueue和writeQueue。

void evictEntries(ReferenceEntry<K, V> newest) {

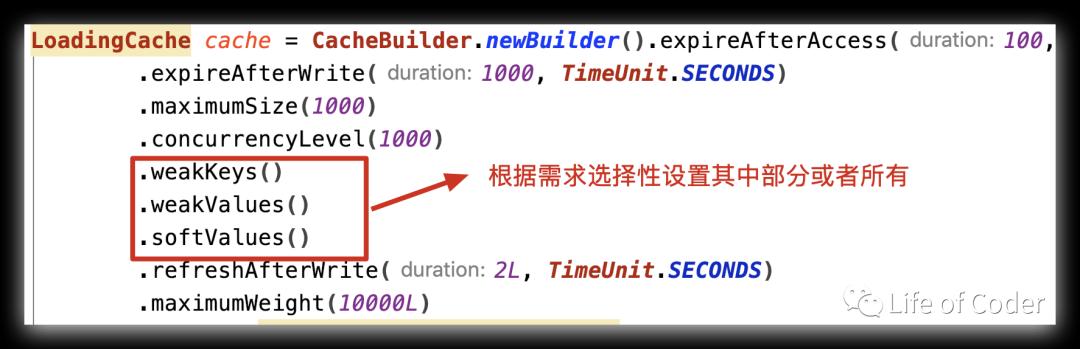
if (!map.evictsBySize()) {  
 return;  
 }  
  
 drainRecencyQueue();  
  
 // If the newest entry by itself is too heavy for the segment, don't bother evicting  
 // anything else, just that  
 if (newest.getValueReference().getWeight() > maxSegmentWeight) {  
 if (!removeEntry(newest, newest.getHash(), RemovalCause.*SIZE*)) {  
 throw new AssertionError();  
 }  
 }  
  
 while (totalWeight > maxSegmentWeight) {  
 ReferenceEntry<K, V> e = getNextEvictable();  
 if (!removeEntry(e, e.getHash(), RemovalCause.*SIZE*)) {  
 throw new AssertionError();  
 }  
 }  
}

@GuardedBy("this")  
ReferenceEntry<K, V> getNextEvictable() {  
 for (ReferenceEntry<K, V> e : accessQueue) {  
 int weight = e.getValueReference().getWeight();  
 if (weight > 0) {  
 return e;  
 }  
 }  
 throw new AssertionError();  
}  
boolean removeEntry(ReferenceEntry<K, V> entry, int hash, RemovalCause cause) {  
 int newCount = this.count - 1;  
 AtomicReferenceArray<ReferenceEntry<K, V>> table = this.table;  
 int index = hash & (table.length() - 1);  
 ReferenceEntry<K, V> first = table.get(index);  
  
 for (ReferenceEntry<K, V> e = first; e != null; e = e.getNext()) {  
 if (e == entry) {  
 ++modCount;  
 ReferenceEntry<K, V> newFirst =  
 removeValueFromChain(  
 first,  
 e,  
 e.getKey(),  
 hash,  
 e.getValueReference().get(),  
 e.getValueReference(),  
 cause);  
 newCount = this.count - 1;  
 table.set(index, newFirst);  
 this.count = newCount; // write-volatile  
 return true;  
 }  
 }  
  
 return false;  
}

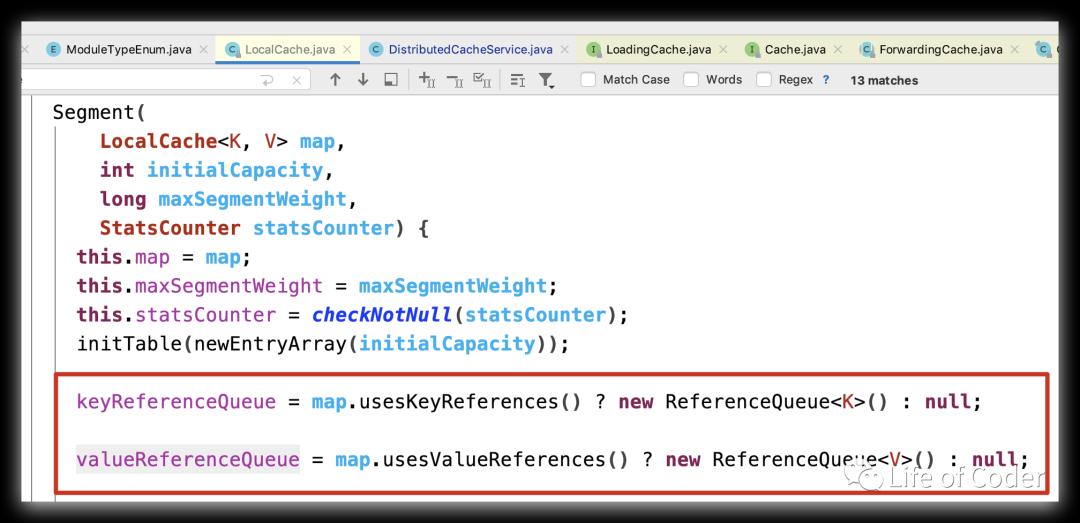
ReferenceEntry<K, V> removeValueFromChain(  
 ReferenceEntry<K, V> first,  
 ReferenceEntry<K, V> entry,  
 @Nullable K key,  
 int hash,  
 V value,  
 ValueReference<K, V> valueReference,  
 RemovalCause cause) {  
 enqueueNotification(key, hash, value, valueReference.getWeight(), cause);  
 writeQueue.remove(entry);  
 accessQueue.remove(entry);  
  
 if (valueReference.isLoading()) {  
 valueReference.notifyNewValue(null);  
 return first;  
 } else {  
 return removeEntryFromChain(first, entry);  
 }  
}

1. **引用类型回收**
2. **使用与基本实现**

Guava可以帮助我们实现引用类型的回收，引用基本类型参见[《jvm-10-4种引用类型》](../jvm/jvm-10-4种引用类型.docx)：



使用过引用类型的同学都知道，在构建引用类型的时候需要传递一个Queue队列，当引用类型被回收后就会被JVM放到这个Queue中。在Guava Cache中，在构建每个Segment的时候就会构建出对应的Queue（如下图）。后面在当前Segment中创建引用类型的时候就使用这个对应的Queue。



下面是软引用对象的构建：

*SOFT* {

@Override  
 <K, V> ValueReference<K, V> referenceValue(  
 Segment<K, V> segment, ReferenceEntry<K, V> entry, V value, int weight) {  
 return (weight == 1)  
 ? new SoftValueReference<K, V>(segment.valueReferenceQueue, value, entry)  
 : new WeightedSoftValueReference<K, V>(  
 segment.valueReferenceQueue, value, entry, weight);  
 }  
  
 @Override  
 Equivalence<Object> defaultEquivalence() {  
 return Equivalence.*identity*();  
 }  
},

1. **回收**

当引用类型（非强引用）被回收后，其对应的引用类型对象会被放到其初始化的队列中。于是我们只要通过不断从Queue中读取数据就知道哪些key或者value被回收了。

在Guava Cache中，引用类型的回收和之前的LRU回收一样，有两个地方会触发，其分别是：

* 在get的时候，如果tryLock获取到锁了，则进行回收。
* 在modify的时候（put|add|delete），此时肯定获取到了锁，直接进行回收。

具体回收都是调用drainReferenceQueues方法，如下：

@GuardedBy("this")

void drainReferenceQueues() {  
 if (map.usesKeyReferences()) {  
 drainKeyReferenceQueue();  
 }  
 if (map.usesValueReferences()) {  
 drainValueReferenceQueue();  
 }  
}

@GuardedBy("this")

void drainValueReferenceQueue() {  
 Reference<? extends V> ref;  
 int i = 0;  
 while ((ref = valueReferenceQueue.poll()) != null) {  
 @SuppressWarnings("unchecked")  
 ValueReference<K, V> valueReference = (ValueReference<K, V>) ref;  
 map.reclaimValue(valueReference);  
 if (++i == *DRAIN\_MAX*) {  
 break;  
 }  
 }  
}

@GuardedBy("this")  
void drainKeyReferenceQueue() {  
 Reference<? extends K> ref;  
 int i = 0;  
 while ((ref = keyReferenceQueue.poll()) != null) {  
 @SuppressWarnings("unchecked")  
 ReferenceEntry<K, V> entry = (ReferenceEntry<K, V>) ref;  
 map.reclaimKey(entry);  
 if (++i == *DRAIN\_MAX*) {  
 break;  
 }  
 }  
}

1. **SoftKeys无意义**

前文中提到，softKeys以及被废弃，本质原因就是没有意义。因为我们要试图从map中获取key，那么一定得是强引用到key，一旦强引用到key断开，无论是软引用还是弱引用，对于key来说只不过是存活时间长短的问题。没有强引用就不再需要保存，所以我们直接使用弱引用来代替软引用，思路可以参考[《java并发-25-threadlocal原理和使用》。](../java并发编程/java并发-25-threadlocal原理和使用.docx)下面是原文：

I wrote the question because, initially, I did genuinely wonder why (as I had existing code that used softKeys). However, the reason was obvious on reflection and I decided to post it here, in case someone else also uses softKeys and was wondering the same thing.

In short, the reason was that softKeys never made any sense in the first place. Thus, its initial inclusion was in itself a mistake, one which the Guava developers are rectifying via deprecation.

In general, you use soft references if you want the object to stick around for a little after all the strong references are gone; in contrast, with weak references, the object usually gets collected soon once there are no strong or soft references left. This is useful for cached values that you want to hold on to temporarily, so that a lookup using the corresponding key will "revive" a strong reference for the value.

However, this behaviour doesn't make any sense for keys:

Since softKeys and weakKeys maps use an identity-based lookup, the only way to get at an entry of interest is to posess a strong reference to its key.† Thus, once there are no strong key references left, the entry is effectively dead (with no possibility of revival).

The only practical difference between softKeys and weakKeys is how long an entry remains in the map after all the strong references to its key are gone. Since such entries are dead anyway, using softKeys instead of weakKeys only delays the entry's eviction, for no good reason.

Thus, most times when coming across code that uses softKeys, a much more suitable replacement is weakKeys.

I am not considering the case of fetching the entry via iteration, or anything other than key-based lookup, since maps are primarily about key-based operations.

**参考**

1. [guava cache入门](https://www.pianshen.com/article/96171073786/)
2. [Guava Cache实现原理——LRU回收实现](https://www.pianshen.com/article/89831117984/)
3. [Guava Cache实现原理——引用类型回收](https://www.pianshen.com/article/15431191444/)
4. jvm-10-4种引用类型
5. java并发-25-threadlocal原理和使用