LRU 是 Least Recently Used 最近最少使用算法。

1. (必填)你需要提供一个缓存容量作为构造参数。

2.（必填） 覆写 sizeOf 方法 ，自定义设计一条数据放进来的容量计算，如果不覆写就无法预知数据的容量，不能保证缓存容量限定在最大容量以内。

3.（选填） 覆写 entryRemoved 方法 ，你可以知道最少使用的缓存被清除时的数据（ evicted, key, oldValue, newVaule ）。

4.（记住）LruCache是线程安全的，在内部的 get、put、remove 包括 trimToSize 都是安全的（因为都上锁了）。

5.（选填） 还有就是覆写 create 方法 。

以下是 一个 LruCache 实现 Bitmap 小缓存的案例

private static final float ONE\_MIB = 1024 \* 1024;

// 7MB

private static final int CACHE\_SIZE = (int) (7 \* ONE\_MIB);

private LruCache<String, Bitmap> bitmapCache;

this.bitmapCache = new LruCache<String, Bitmap>(CACHE\_SIZE) {

protected int sizeOf(String key, Bitmap value) {

return value.getByteCount();

}

@Override

protected void entryRemoved(boolean evicted, String key, Bitmap oldValue, Bitmap newValue) {

...

}

};

LruCache 就是 利用 LinkedHashMap 的一个特性（ accessOrder＝true 基于访问顺序 ）再加上对 LinkedHashMap 的数据操作上锁实现的缓存策略。

LruCache 的数据缓存是内存中的。

1. 首先设置了内部 LinkedHashMap 构造参数 accessOrder=true， 实现了数据排序按照访问顺序。
2. 然后在每次 LruCache.get(K key)方法里都会调用LinkedHashMap.get(Object key)。
3. 如上述设置了 accessOrder=true 后，每次 LinkedHashMap.get(Object key) 都会进行 LinkedHashMap.makeTail(LinkedEntry<K, V> e)。
4. LinkedHashMap 是双向循环链表，然后每次 LruCache.get -> LinkedHashMap.get 的数据就被放到最末尾了。
5. 在 put 和 trimToSize 的方法执行下，如果发生数据量移除，会优先移除掉最前面的数据（因为最新访问的数据在尾部）。

LruCache的构造方法

/\*\*

\* LruCache的构造方法：需要传入最大缓存个数

\*/

public LruCache(int maxSize) {

...

this.maxSize = maxSize;

/\*

\* 初始化LinkedHashMap

\* 第一个参数：initialCapacity，初始大小

\* 第二个参数：loadFactor，负载因子=0.75f

\* 第三个参数：accessOrder=true，基于访问顺序；accessOrder=false，基于插入顺序

\*/

this.map = new LinkedHashMap<K, V>(0, 0.75f, true);

}

第一个参数 initialCapacity 用于初始化该 LinkedHashMap 的大小。

第二个参数 loadFactor，这个其实的 HashMap里的构造参数，涉及到扩容问题，比如 HashMap 的最大容量是100，那么这里设置0.75f的话，到75容量的时候就会扩容。

第三个参数 accessOrder=true ，这样的话 LinkedHashMap 数据排序就会基于数据的访问顺序，从而实现了 LruCache 核心工作原理。

LruCache.get(K key)

/\*\*

\* 根据 key 查询缓存，如果存在于缓存或者被 create 方法创建了。

\* 如果值返回了，那么它将被移动到双向循环链表的的尾部。

\* 如果如果没有缓存的值，则返回 null。

\*/

public final V get(K key) {

...

V mapValue;

synchronized (this) {

// 关键点：LinkedHashMap每次get都会基于访问顺序来重整数据顺序

mapValue = map.get(key);

// 计算 命中次数

if (mapValue != null) {

hitCount++;

return mapValue;

}

// 计算 丢失次数

missCount++;

}

/\*

\* 官方解释：

\* 尝试创建一个值，这可能需要很长时间，并且Map可能在create()返回的值时有所不同。如果在create()执行的时

\* 候，一个冲突的值被添加到Map，我们在Map中删除这个值，释放被创造的值。

\*/

V createdValue = create(key);

if (createdValue == null) {

return null;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* 不覆写create方法走不到下面 \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* 正常情况走不到这里

\* 走到这里的话 说明 实现了自定义的 create(K key) 逻辑

\* 因为默认的 create(K key) 逻辑为null

\*/

synchronized (this) {

// 记录 create 的次数

createCount++;

// 将自定义create创建的值，放入LinkedHashMap中，如果key已经存在，会返回 之前相同key 的值

mapValue = map.put(key, createdValue);

// 如果之前存在相同key的value，即有冲突。

if (mapValue != null) {

/\*

\* 有冲突

\* 所以 撤销 刚才的 操作

\* 将 之前相同key 的值 重新放回去

\*/

map.put(key, mapValue);

} else {

// 拿到键值对，计算出在容量中的相对长度，然后加上

size += safeSizeOf(key, createdValue);

}

}

// 如果上面 判断出了 将要放入的值发生冲突

if (mapValue != null) {

/\*

\* 刚才create的值被删除了，原来的 之前相同key 的值被重新添加回去了

\* 告诉 自定义 的 entryRemoved 方法

\*/

entryRemoved(false, key, createdValue, mapValue);

return mapValue;

} else {

// 上面 进行了 size += 操作 所以这里要重整长度

trimToSize(maxSize);

return createdValue;

}

}

LinkedHashMap.get(Object key)

/\*\*

\* Returns the value of the mapping with the specified key.

\*

\* @param key

\* the key.

\* @return the value of the mapping with the specified key, or {@code null}

\* if no mapping for the specified key is found.

\*/

@Override public V get(Object key) {

/\*

\* This method is overridden to eliminate the need for a polymorphic

\* invocation in superclass at the expense of code duplication.

\*/

if (key == null) {

HashMapEntry<K, V> e = entryForNullKey;

if (e == null)

return null;

if (accessOrder)

makeTail((LinkedEntry<K, V>) e);

return e.value;

}

int hash = Collections.secondaryHash(key);

HashMapEntry<K, V>[] tab = table;

for (HashMapEntry<K, V> e = tab[hash & (tab.length - 1)];

e != null; e = e.next) {

K eKey = e.key;

if (eKey == key || (e.hash == hash && key.equals(eKey))) {

if (accessOrder)

makeTail((LinkedEntry<K, V>) e);

return e.value;

}

}

return null;

}

如果 accessOrder=true 那么每次 get 都会执行 N 次 makeTail(LinkedEntry<K, V> e) 。

LinkedHashMap.makeTail(LinkedEntry e)

/\*\*

\* Relinks the given entry to the tail of the list. Under access ordering,

\* this method is invoked whenever the value of a pre-existing entry is

\* read by Map.get or modified by Map.put.

\*/

private void makeTail(LinkedEntry<K, V> e) {

// Unlink e

e.prv.nxt = e.nxt;

e.nxt.prv = e.prv;

// Relink e as tail

LinkedEntry<K, V> header = this.header;

LinkedEntry<K, V> oldTail = header.prv;

e.nxt = header;

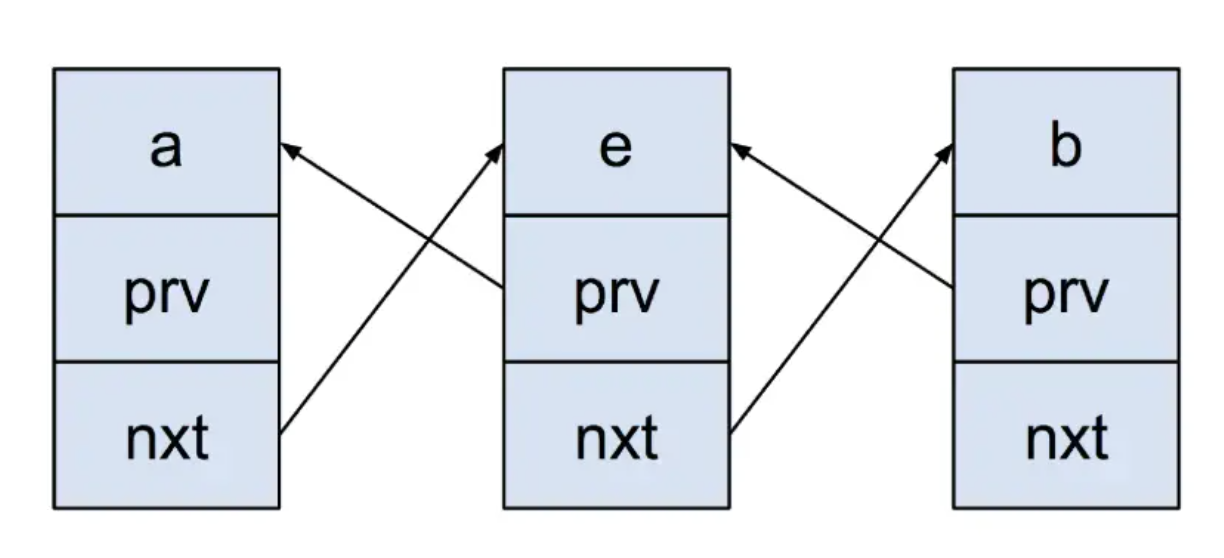
e.prv = oldTail;

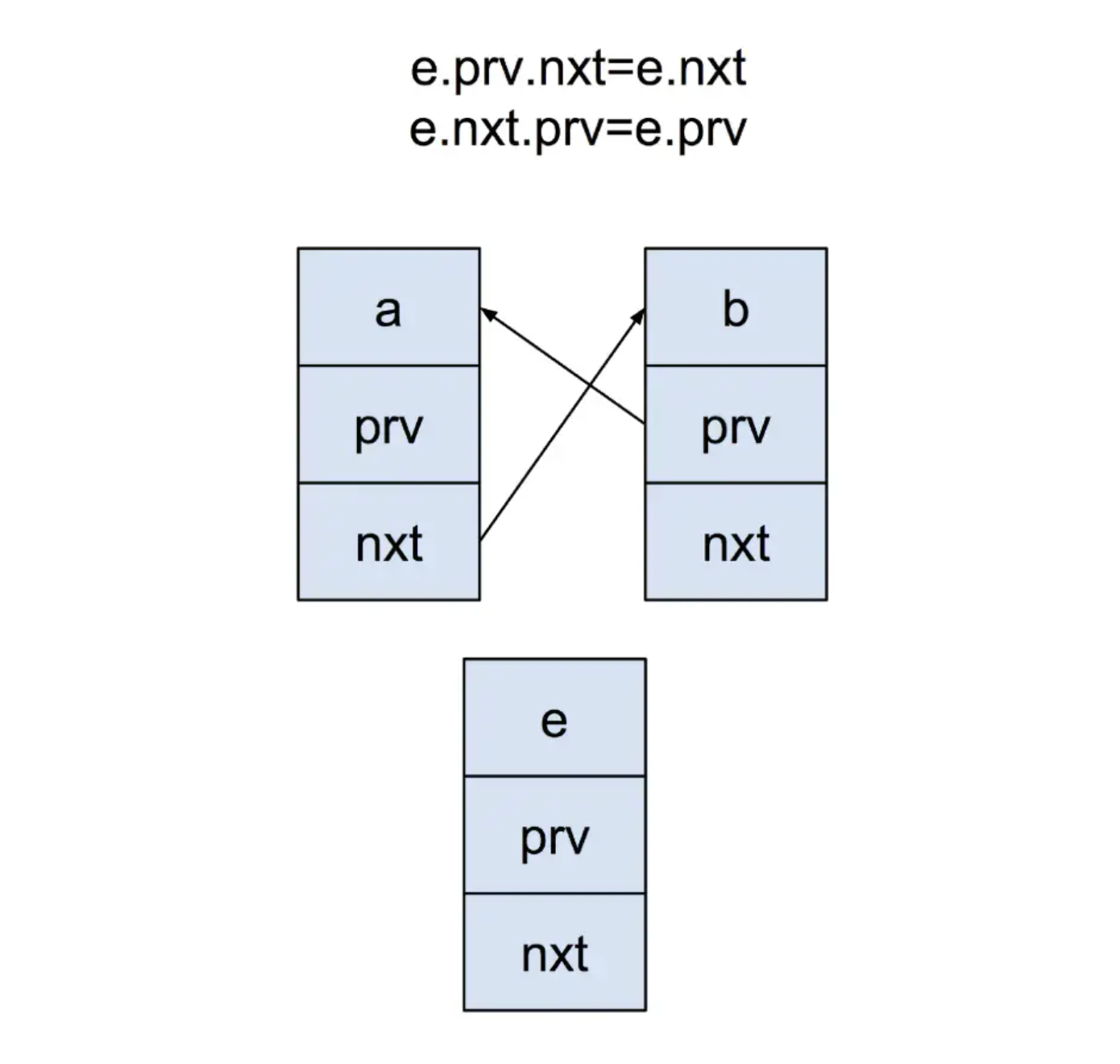
oldTail.nxt = header.prv = e;

modCount++;

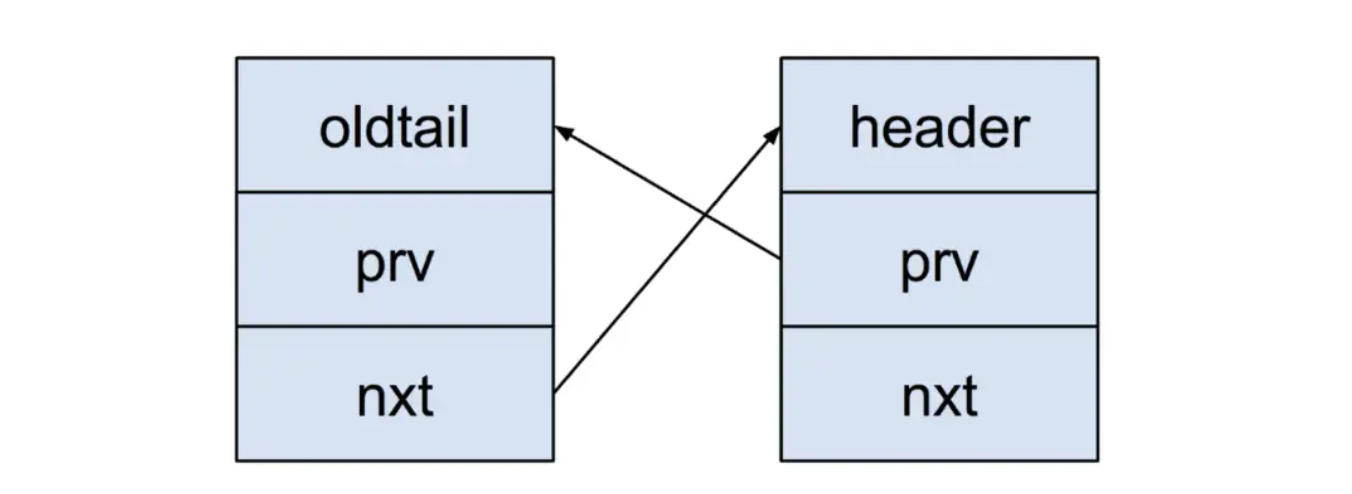
}

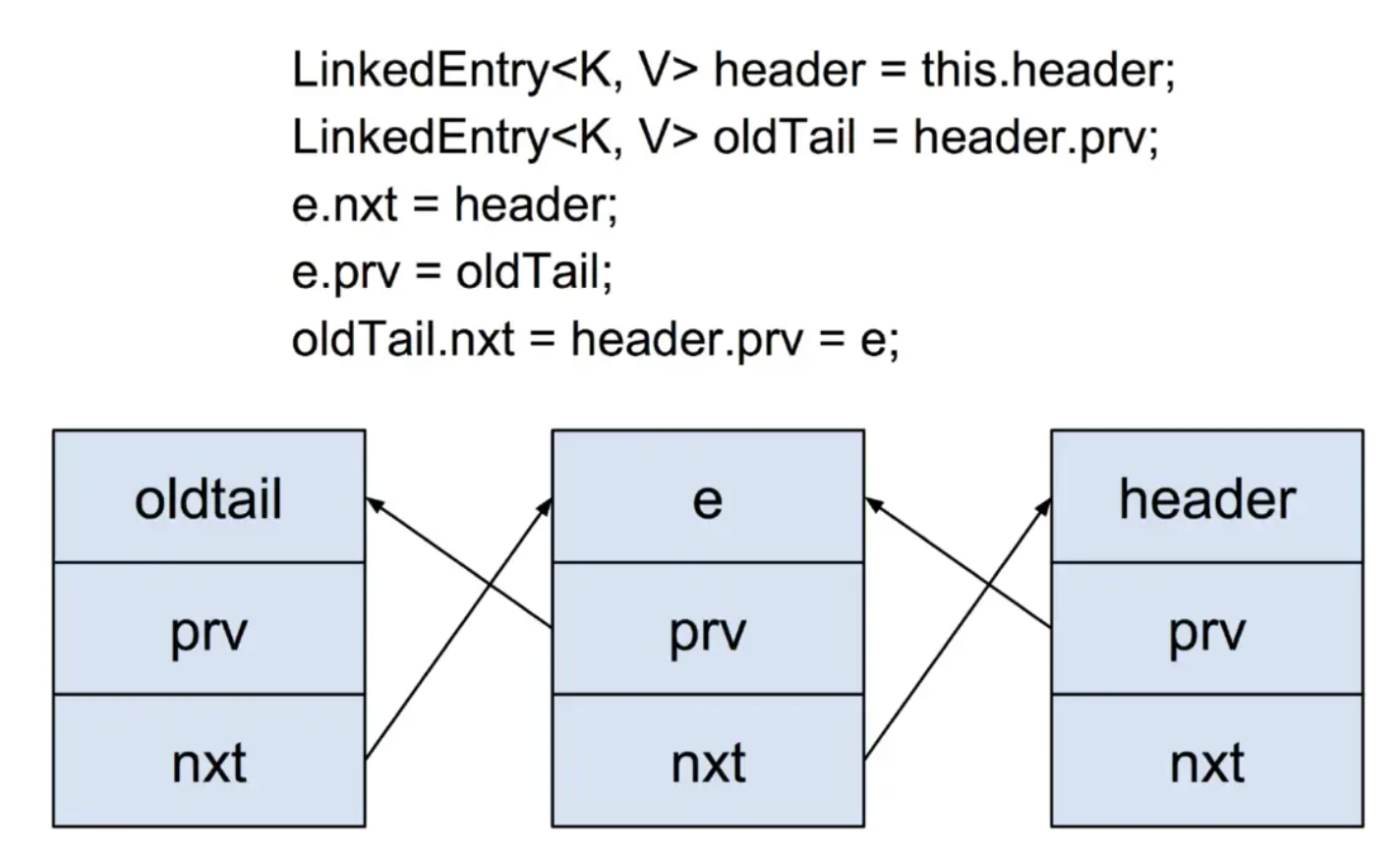
Unlink e





Relink e as tail





LinkedHashMap 是双向循环链表，然后此次 LruCache.get -> LinkedHashMap.get 的数据就被放到最末尾了。

LruCache.put(K key, V value)

public final V put(K key, V value) {

...

synchronized (this) {

...

// 拿到键值对，计算出在容量中的相对长度，然后加上

size += safeSizeOf(key, value);

...

}

...

trimToSize(maxSize);

return previous;

}

1. put 开始的时候确实是把值放入 LinkedHashMap 了，不管超不超过你设定的缓存容量。2.然后根据 safeSizeOf 方法计算 此次添加数据的容量是多少，并且加到 size里 。说到 safeSizeOf 就要讲到 `sizeOf(K key, V value) 会计算出此次添加数据的大小 。
2. 直到 put 要结束时，进行了 trimToSize 才判断 size 是否 大于 maxSize 然后进行最近很少访问数据的移除。

LruCache.trimToSize(int maxSize)

public void trimToSize(int maxSize) {

/\*

\* 这是一个死循环，

\* 1.只有 扩容 的情况下能立即跳出

\* 2.非扩容的情况下，map的数据会一个一个删除，直到map里没有值了，就会跳出

\*/

while (true) {

K key;

V value;

synchronized (this) {

// 在重新调整容量大小前，本身容量就为空的话，会出异常的。

if (size < 0 || (map.isEmpty() && size != 0)) {

throw new IllegalStateException(

getClass().getName() + ".sizeOf() is reporting inconsistent results!");

}

// 如果是 扩容 或者 map为空了，就会中断，因为扩容不会涉及到丢弃数据的情况

if (size <= maxSize || map.isEmpty()) {

break;

}

Map.Entry<K, V> toEvict = map.entrySet().iterator().next();

key = toEvict.getKey();

value = toEvict.getValue();

map.remove(key);

// 拿到键值对，计算出在容量中的相对长度，然后减去。

size -= safeSizeOf(key, value);

// 添加一次收回次数

evictionCount++;

}

/\*

\* 将最后一次删除的最少访问数据回调出去

\*/

entryRemoved(true, key, value, null);

}

}

会判断之前 size 是否小于 maxSize 。是的话，直接跳出后什么也不做。不是的话，证明已经溢出容量了。由 makeTail 图已知，最近经常访问的数据在最末尾。拿到一个存放 key 的 Set，然后一直一直从头开始删除，删一个判断是否溢出，直到没有溢出。

entryRemoved 的作用

entryRemoved被LruCache调用的场景

1. （put） put发生 key 冲突时被调用，evicted=false，key=此次put 的 key，oldValue=被覆盖的冲突 value，newValue=此次 put 的 value。
2. （trimToSize） trimToSize 的时候，只会被调用一次，就是最后一次被删除的最少访问数据带回来。evicted=true，key=最后一次被删除的 key，oldValue=最后一次被删除的 value，newValue=null（此次没有冲突，只是 remove）。
3. （remove） remove的时候，存在对应 key，并且被成功删除后被调用。evicted=false，key=此次 put的 key，oldValue=此次删除的 value，newValue=null（此次没有冲突，只是 remove）。
4. （get后半段，查询丢失后处理情景，不过建议忽略） get 的时候，正常的话不实现自定义 create 的话，代码上看 get 方法只会走一半，如果你实现了自定义的 create(K key) 方法，并且在 你 create 后的值放入 LruCache 中发生 key 冲突时被调用，evicted=false，key=此次 get 的 key，oldValue=被你自定义 create(key)后的 value，newValue=原本存在 map 里的 key-value。

因为 HashMap 在数据量大情况下，拿数据可能造成丢失，导致前半段查不到，你自定义的 create(key)放入的时候发现又查到了（有冲突）。然后又急忙把原来的值放回去，此时你就白白create一趟，无所作为，还要走一遍entryRemoved。

/\*\*

\* 1.当被回收或者删掉时调用。该方法当value被回收释放存储空间时被remove调用

\* 或者替换条目值时put调用，默认实现什么都没做。

\* 2.该方法没用同步调用，如果其他线程访问缓存时，该方法也会执行。

\* 3.evicted=true：如果该条目被删除空间 （表示 进行了trimToSize or remove） evicted=false：put冲突后 或 get里成功create后

\* 导致

\* 4.newValue!=null，那么则被put()或get()调用。

\*/

protected void entryRemoved(boolean evicted, K key, V oldValue, V newValue) {

}

LruCache 局部同步锁

在 get, put, trimToSize, remove 四个方法里的 entryRemoved 方法都不在同步块里。因为 entryRemoved 回调的参数都属于方法域参数，不会线程不安全。

本地方法栈和程序计数器是线程隔离的数据区。