**1.技术点**

**1.1 java中的引用**

引用类型主要分为4种：①强引用；②软引用；③弱引用；④虚引用。强引用就是永远不会回收掉被引用的对象，比如说我们代码中new出来的对象。软引用表示有用但是非必需的，如果系统内存资源紧张，可能就会被回收；弱引用表示非必需的对象，只能存活到下一次垃圾回收发生之前；虚引用是最弱的，这个引用无法操作对象。

**1.2引用队列ReferenceQueue**

|  |
| --- |
| Object o1 = new Object();  Integer o2 = new Integer((int) o1);  如果o1对象除了在o2中有引用之外没有别的地方存在引用，那么就可以回收o1。然后当这个o1被回收之后，我们就需要把o2放入引用队列中，所以引用队列（ReferenceQueue）就是Reference的监听器。在WeakHashMap中就是通过ReferenceQueue来反向处理map中的数据，如果对象被回收了，那么就需要把map中的对应数据移除。  **public** **class** ReferenceTest {  **private** **static** **final** **int** ***\_1MB*** = 1024\*1024;//设置大小为1MB  **public** **static** **void** main(String[] args) **throws** InterruptedException {  ReferenceQueue<Object> referenceQueue = **new** ReferenceQueue<Object>();//用引用队列进行监控引用的回收情况  Object value = **new** Object();  Map<Object, Object> map = **new** HashMap<Object, Object>();  **for** (**int** i = 0; i < 100; i++) {//循环100次把数据插入到弱引用中（WeakReference）， 同时把弱引用作为key存入HashMap  **byte**[] bytes = **new** **byte**[***\_1MB***];  //每个弱引用中都有关联引用队列（referenceQueue），由于bytes只被weakReference引用，因此每次weakReference中的bytes被回收之后，那么这个weakReference对象就会放入引用队列  WeakReference<**byte**[]> weakReference = **new** WeakReference<**byte**[]>(bytes, referenceQueue);  //通过map存储弱引用对象  map.put(weakReference, value);  }  Thread thread = **new** Thread(**new** Runnable() {//线程通过调用引用队列的情况查看那些对象被回收  @SuppressWarnings("unchecked")  **public** **void** run() {  **try** {  **int** cnt = 0;  WeakReference<**byte**[]> k;  **while** ((k = (WeakReference<**byte**[]>) referenceQueue.remove()) != **null**) {//返回被回收对象的引用（注意本例中被回收的是bytes）  System.***out***.println((cnt++)+"回收了"+k);  System.***out***.println("map的size = " + map.size());//用于监控map的存储数量有没有发生变化  }  } **catch** (Exception e) {  // **TODO**: handle exception  }  }  });  thread.start();  }  } |

流程：

①bytes对象存入weakReference对象中

②weakReference作为key存入HashMap中

③GC回收了bytes对象，这时，引用这个对象的weakReference对象就会被存储到referenceQueue中

④循环输出referenceQueue，输出被回收的bytes关联的weakReference对象

下面用WeakHashMap做测试

|  |
| --- |
| **import** java.util.Map;  **import** java.util.WeakHashMap;  **public** **class** ReferenceTest2 {  **private** **static** **final** **int** ***\_1MB*** = 1024\*1024;//设置大小为1MB  **public** **static** **void** main(String[] args) **throws** InterruptedException {  Object value = **new** Object();  Map<Object, Object> map = **new** WeakHashMap<Object, Object>();  **for** (**int** i = 0; i < 100; i++) {//循环100次把数据插入WeakHashMap中  **byte**[] bytes = **new** **byte**[***\_1MB***];  map.put(bytes, value);  }  **while** (**true**) {//死循环监控map大小变化  Thread.*sleep*(500);//稍稍停顿，效果更直观  System.***out***.println(map.size());//打印WeakHashMap的大小  System.*gc*();//建议系统进行GC  }  }  } |

因为在插入过程中就存在了GC，会把WeakHashMap的内容清除，因为一旦GC发生，弱引用就会被清除，导致WeakHashMap大小为0；

在你触发一些操作的时候，WeakHashMap里面的数据就会被移除了

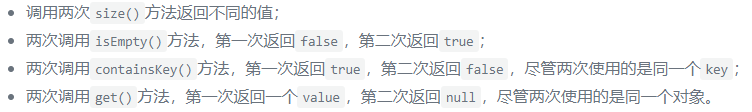
|  |
| --- |
| **public** **int** size() {  **if** (size == 0)  **return** 0;  expungeStaleEntries();从表中清除掉不存在的引用的方法。  **return** size;  } |

**2.源码分析**

|  |
| --- |
| ①引用队列  private final ReferenceQueue<Object> queue = new ReferenceQueue<>();  ②静态内部类entry  **private** **static** **class** Entry<K,V> **extends** WeakReference<Object> **implements** Map.Entry<K,V> {  V value;  **final** **int** hash;  Entry<K,V> next;  Entry(Object key, V value,  ReferenceQueue<Object> queue,  **int** hash, Entry<K,V> next) {  **super**(key, queue);  **this**.value = value;  **this**.hash = hash;  **this**.next = next;  }  }  如果上面第一个代码所示，key值和referenceQueue作为weakReference的参数封装进了WeakReference中，GC的时候回收掉弱引用引用的对象，如上面的bytes，  ③核心移除map中K，V方法  解释一下为什么对象回收之后，map中对应的K，V也会被移除  **private** **void** expungeStaleEntries() {  GC之后，queue里面就会存着对应的reference的子类，这里是WeakReference的子类Entry。，  **for** (Object x; (x = queue.poll()) != **null**; ) {  **synchronized** (queue) {//线程同步，锁定队列  @SuppressWarnings("unchecked")  Entry<K,V> e = (Entry<K,V>) x;  得到对应元素在table中的下标  **int** i = *indexFor*(e.hash, table.length);  prev指向table[i]即链表头结点  Entry<K,V> prev = table[i];  Entry<K,V> p = prev;用来表示p的上一个节点  **while** (p != **null**) {如果p节点存在  Entry<K,V> next = p.next;定义next执行p的next  **if** (p == e) {**如果当前队列的节点e==p**  **if** (prev == e)如果第一次prev就是e,说明e就是头结点，直接指向next  table[i] = next;  **else否则prev的next指向p的next节点**  prev.next = next;  // Must not null out e.next;  // stale entries may be in use by a HashIterator  e.value = **null**; // Help GC--e的value值置空  size--;  **break**;  }  不相等，就继续沿着链表往下找  prev = p;  p = next;  }  }  }  }  通过hash值与length-1找到对应元素在table中的下标  **private** **static** **int** indexFor(**int** h, **int** length) {  **return** h & (length-1);  } |

那在什么时候会调用这个expungeStaleEntries方法

|  |
| --- |
| //size方法  public int size() {  if (size == 0)  return 0;  expungeStaleEntries();//去除被回收的对象  return size;  }  //getTable方法（这个方法是put和get方法的辅助方法）    private Entry<K,V>[] getTable() {  expungeStaleEntries();//去除被回收的对象  return table;  }  //resize扩容方法  void resize(int newCapacity) {  Entry<K,V>[] oldTable = **getTable();**  int oldCapacity = oldTable.length;  if (oldCapacity == MAXIMUM\_CAPACITY) {  threshold = Integer.MAX\_VALUE;  return;  }  Entry<K,V>[] newTable = newTable(newCapacity);  transfer(oldTable, newTable);  table = newTable;  if (size >= threshold / 2) {  threshold = (int)(newCapacity \* loadFactor);  } else {  expungeStaleEntries();//去除被回收的对象  transfer(newTable, oldTable);  table = oldTable;  }  }  //get方法（基于上面说的getTable方法）  public V get(Object key) {  Object k = maskNull(key);  int h = hash(k);  Entry<K,V>[] tab = getTable();//getTable中包装了expungeStaleEntries方法  int index = indexFor(h, tab.length);  Entry<K,V> e = tab[index];  while (e != null) {  if (e.hash == h && eq(k, e.get()))  return e.value;  e = e.next;  }  return null;  }  //put方法（基于上面说的getTable方法）  public V put(K key, V value) {  Object k = maskNull(key);  int h = hash(k);  Entry<K,V>[] tab = getTable();//getTable中包装了expungeStaleEntries方法  int i = indexFor(h, tab.length);  for (Entry<K,V> e = tab[i]; e != null; e = e.next) {  if (h == e.hash && eq(k, e.get())) {  V oldValue = e.value;  if (value != oldValue)  e.value = value;  return oldValue;  }  } |



**WeekHashMap 的这个特点特别适用于需要缓存的场景，**在缓存场景下，由于内存是有限的，不能缓存所有对象；对象缓存命中可以提高系统效率，不过并不是说这个用于缓存一定合理，因为如果你只put了，没有做其他get或者会触发expungeStaleEntries操作的方法， 那么该值就会一直存在了。建议还是使用LRU缓存会比较好