Java中一共有4种引用类型(其实还有一些其他的引用类型比如FinalReference)：

强引用、软引用、弱引用、虚引用。

其中强引用就是我们经常使用的Object a = new Object(); 这样的形式，在Java中并没有对应的Reference类。

本篇文章主要是分析软引用、弱引用、虚引用的实现，这三种引用类型都是继承于Reference这个类，主要逻辑也在Reference中。

### 1、问题

1.网上大多数文章对于软引用的介绍是：在内存不足的时候才会被回收，那内存不足是怎么定义的？什么才叫内存不足？

2.网上大多数文章对于虚引用的介绍是：形同虚设，虚引用并不会决定对象的生命周期。主要用来跟踪对象被垃圾回收器回收的活动。真的是这样吗？

3.虚引用在Jdk中有哪些场景下用到了呢？

### 2、Reference

// 引用对象

private T referent; /\* Treated specially by GC \*/

// 回收队列，由使用者在Reference的构造器中指定

volatile ReferenceQueue<? super T> queue;  
  
// 当该引用被加入到queue中的时候，该字段被设置为queue中的下一个元素，以形成链表结构  
@SuppressWarnings("rawtypes")  
volatile Reference next;  
  
/\*

\* 在GC时，JVM底层会维护一个叫DiscoveredList的链表，存放的是Reference

\* 对象，discovered字段指向的就是链表中的下一个元素，由JVM设置  
 \*/  
transient private Reference<T> discovered; /\* used by VM \*/  
  
  
// 进行线程同步的锁对象  
static private class Lock { }  
private static Lock *lock* = new Lock();  
  
  
/\*

\* 等待加入queue的Reference对象，在GC时由JVM设置，会有一个java层

\* 的线程(ReferenceHandler)源源不断的从pending中提取元素加入到queue

\*/  
private static Reference<Object> *pending* = null;

一个Reference对象的生命周期如下：



主要分为Native层和Java层两个部分。

Native层在GC时将需要被回收的Reference对象加入到DiscoveredList中（代码在referenceProcessor.cpp中process\_discovered\_references方法），然后将DiscoveredList的元素移动到PendingList中（代码在referenceProcessor.cpp中enqueue\_discovered\_ref\_helper方法）,PendingList的队首就是Reference类中的pending对象。

Java层代码：

private static class ReferenceHandler extends Thread {

···

public void run() {  
 while (true) {  
 *tryHandlePending*(true);  
 }  
 }  
}

static boolean tryHandlePending(boolean waitForNotify) {  
 Reference<Object> r;  
 Cleaner c;  
 try {  
 synchronized (*lock*) {  
 if (*pending* != null) {  
 r = *pending*;  
 // 如果是Cleaner对象，则记录下来，下面做特殊处理  
 c = r instanceof Cleaner ? (Cleaner) r : null;  
 // 指向PendingList的下一个对象  
 *pending* = r.discovered;  
 r.discovered = null;  
 } else {  
 // 如果PendingList为null就先等待，当有对象加入到PendingList中时，jvm会执行notify  
 if (waitForNotify) {  
 *lock*.wait();  
 }  
 // retry if waited  
 return waitForNotify;  
 }  
 }  
 } catch (OutOfMemoryError x) {  
 Thread.*yield*();  
 return true;  
 } catch (InterruptedException x) {  
 return true;  
 }  
  
 // 如果是Cleaner对象，则调用clean方法进行资源回收  
 if (c != null) {  
 c.clean();  
 return true;  
 }  
  
 // 将Reference加入到ReferenceQueue，开发者可以通过从ReferenceQueue中poll元素感知到对象被回收的事件

ReferenceQueue<? super Object> q = r.queue;  
 if (q != ReferenceQueue.*NULL*) q.enqueue(r);  
 return true;  
}

流程比较简单：就是源源不断的从PendingList中提取出元素，然后将其加入到ReferenceQueue中去，开发者可以通过从ReferenceQueue中poll元素感知到对象被回收的事件。

另外需要注意的是，对于Cleaner类型（继承自虚引用）的对象会有额外的处理：在其指向的对象被回收时，会调用clean方法，该方法主要是用来做对应的资源回收，在堆外内存DirectByteBuffer中就是用Cleaner进行堆外内存的回收，这也是虚引用在java中的典型应用。

看完了Reference的实现，再看看几个实现类里，各自有什么不同。

SoftReference：

public class SoftReference<T> extends Reference<T> {

*/\*\*  
 \* 静态变量，每次GC时都会将该字段设置成当前时间。  
 \*/* static private long *clock*;  
  
 */\*\*  
 \* timestamp字段则会在每次调用get方法时将其赋值为clock（如果不相等且对象没被回收）。  
 \*/* private long timestamp;  
public SoftReference(T referent) {  
 super(referent);  
 this.timestamp = *clock*;  
 }  
public SoftReference(T referent, ReferenceQueue<? super T> q) {  
 super(referent, q);  
 this.timestamp = *clock*;  
 }  
public T get() {  
 T o = super.get();  
 if (o != null && this.timestamp != *clock*)  
 this.timestamp = *clock*;  
 return o;  
 }  
  
}

那这两个字段的作用是什么呢？这和软引用在内存不够的时候才被回收，又有什么关系呢？

这些还得看JVM的源码才行，因为决定对象是否需要被回收都是在GC中实现的。



<https://mp.weixin.qq.com/s/vIAXODcnf-5dbEdWqssPlg>