**[Java中的软引用，弱引用和虚引用](http://www.blogjava.net/qileilove/archive/2012/02/02/369217.html)**

　在[**Android**](http://www.51testing.com/html/26/n-806726.html)的图片处理中，碰到的一个非常普遍的问题便是OOM错误 为此网上也有很多例子，而在之前的一篇转载里 提到了ListView中加载图片的ImageLoader，而其中有一处，使用到了名为SoftPreference的类 这是[**Java**](http://www.51testing.com/html/26/n-806726.html)中的一个类 也就是所谓的软引用 在查询了相关的资料以后 会发现SoftPreference的特性，非常适合用来处理OOM引起的问题 下面是[**百度**](http://www.51testing.com/html/26/n-806726.html)文库的一篇转载：

　　SoftReference、Weak Reference和PhantomRefrence分析和比较

　　本文将谈一下对SoftReference（软引用）、WeakReference（弱引用）和PhantomRefrence（虚引用）的理解，这三个类是对heap中java对象的应用，通过这个三个类可以和gc做简单的交互。

**强引用：**

　　除了上面提到的三个引用之外，还有一个引用，也就是最长用到的那就是强引用。例如：

|  |
| --- |
| Object o=new Object();    Object o1=o; |

　　上面代码中第一句是在heap堆中创建新的Object对象通过o引用这个对象，第二句是通过o建立o1到new Object()这个heap堆中的对象的引用，这两个引用都是强引用.只要存在对heap中对象的引用，gc就不会收集该对象.如果通过如下代码：

|  |
| --- |
| o=null;    o1=null; |

　　如果显式地设置o和o1为null，或超出范围，则gc认为该对象不存在引用，这时就可以收集它了。可以收集并不等于就一会被收集，什么时候收集这要取决于gc的算法，这要就带来很多不确定性。例如你就想指定一个对象，希望下次gc运行时把它收集了，那就没办法了，有了其他的三种引用就可以做到了。其他三种引用在不妨碍gc收集的情况下，可以做简单的交互。

　　heap中对象有强可及对象、软可及对象、弱可及对象、虚可及对象和不可到达对象。应用的强弱顺序是强、软、弱、和虚。对于对象是属于哪种可及的对象，由他的最强的引用决定。如下：

|  |
| --- |
| String abc=new String("abc");  //1    SoftReference<String> abcSoftRef=new SoftReference<String>(abc);  //2    WeakReference<String> abcWeakRef = new WeakReference<String>(abc); //3    abc=null; //4    abcSoftRef.clear();//5 |

　　第一行在heap对中创建内容为“abc”的对象，并建立abc到该对象的强引用,该对象是强可及的。

　　第二行和第三行分别建立对heap中对象的软引用和弱引用，此时heap中的对象仍是强可及的。

　　第四行之后heap中对象不再是强可及的，变成软可及的。同样第五行执行之后变成弱可及的。

**SoftReference（软引用）**

　　软引用是主要用于内存敏感的高速缓存。在jvm报告内存不足之前会清除所有的软引用，这样以来gc就有可能收集软可及的对象，可能解决内存吃紧问题，避免内存溢出。什么时候会被收集取决于gc的算法和gc运行时可用内存的大小。当gc决定要收集软引用是执行以下过程,以上面的abcSoftRef为例：

　　1、首先将abcSoftRef的referent设置为null，不再引用heap中的new String("abc")对象。

　　2、将heap中的new String("abc")对象设置为可结束的(finalizable)。

　　3、当heap中的new String("abc")对象的finalize()方法被运行而且该对象占用的内存被释放， abcSoftRef被添加到它的ReferenceQueue中。

　　注:对ReferenceQueue软引用和弱引用可以有可无，但是虚引用必须有，参见：

|  |
| --- |
| Reference(T paramT, ReferenceQueue<? super T>paramReferenceQueue) |

　　被 Soft Reference 指到的对象，即使没有任何 Direct Reference，也不会被清除。一直要到 JVM 内存不足且 没有 Direct Reference 时才会清除，SoftReference 是用来设计 object-cache 之用的。如此一来 SoftReference 不但可以把对象 cache 起来，也不会造成内存不足的错误 （OutOfMemoryError）。我觉得 Soft Reference 也适合拿来实作 pooling 的技巧。

|  |
| --- |
| A obj = new A(); SoftRefenrence sr = new SoftReference(obj);      //引用时     if(sr!=null){         obj = sr.get();     }else{         obj = new A();         sr = new SoftReference(obj);     } |

**弱引用**

　　当gc碰到弱可及对象，并释放abcWeakRef的引用，收集该对象。但是gc可能需要对此运用才能找到该弱可及对象。通过如下代码可以了明了的看出它的作用：

|  |
| --- |
| String abc=new String("abc");     WeakReference<String> abcWeakRef = new WeakReference<String>(abc);     abc=null;     System.out.println("before gc: "+abcWeakRef.get());     System.gc();     System.out.println("after gc: "+abcWeakRef.get()); |

　　运行结果：

　　before gc: abc  
　　after gc: null

　　gc收集弱可及对象的执行过程和软可及一样，只是gc不会根据内存情况来决定是不是收集该对象。

　　如果你希望能随时取得某对象的信息，但又不想影响此对象的垃圾收集，那么你应该用 Weak Reference 来记住此对象，而不是用一般的 reference。

|  |
| --- |
| A obj = new A();      WeakReference wr = new WeakReference(obj);      obj = null;      //等待一段时间，obj对象就会被垃圾回收 　　...  　　if (wr.get()==null) {  　　System.out.println("obj 已经被清除了 ");  　　} else {  　　System.out.println("obj 尚未被清除，其信息是 "+obj.toString()); 　　} 　　... } |

　　在此例中，透过 get() 可以取得此 Reference 的所指到的对象，如果返回值为 null 的话，代表此对象已经被清除。

　　这类的技巧，在设计 Optimizer 或 Debugger 这类的程序时常会用到，因为这类程序需要取得某对象的信息，但是不可以 影响此对象的垃圾收集。

**PhantomRefrence（虚引用）**

　　虚顾名思义就是没有的意思，建立虚引用之后通过get方法返回结果始终为null，通过源代码你会发现，虚引用通向会把引用的对象写进referent，只是get方法返回结果为null。先看一下和gc交互的过程在说一下他的作用。

　　1 不把referent设置为null，直接把heap中的new String("abc")对象设置为可结束的(finalizable).

　　2 与软引用和弱引用不同，先把PhantomRefrence对象添加到它的ReferenceQueue中，然后在释放虚可及的对象。

　　你会发现在收集heap中的new String("abc")对象之前，你就可以做一些其他的事情。通过以下代码可以了解他的作用。

|  |
| --- |
| import java.lang.ref.PhantomReference;     import java.lang.ref.Reference;     import java.lang.ref.ReferenceQueue;     import java.lang.reflect.Field;         public class Test {         public static boolean isRun = true;             public static void main(String[] args) throws Exception {             String abc = new String("abc");             System.out.println(abc.getClass() + "@" + abc.hashCode());             final ReferenceQueue referenceQueue = new ReferenceQueue<String>();             new Thread() {                 public void run() {                     while (isRun) {                         Object o = referenceQueue.poll();                         if (o != null) {                             try {                                 Field rereferent = Reference.class                                        .getDeclaredField("referent");                                 rereferent.setAccessible(true);                                 Object result = rereferent.get(o);                                 System.out.println("gc will collect:"                                        + result.getClass() + "@"                                        + result.hashCode());                             } catch (Exception e) {                                     e.printStackTrace();                             }                         }                     }                 }             }.start();             PhantomReference<String> abcWeakRef = new PhantomReference<String>(abc,                     referenceQueue);             abc = null;             Thread.currentThread().sleep(3000);             System.gc();             Thread.currentThread().sleep(3000);             isRun = false;         }         } |

　　结果为：

　　class java.lang.String@96354  
　　gc will collect:class java.lang.String@96354