[Java四种引用类型](https://www.cnblogs.com/liyutian/p/9690974.html)

**引用与对象**

每种编程语言都有自己操作内存中元素的方式，例如在 C 和 C++ 里是通过指针，而在 Java 中则是通过“引用”。  
在 Java 中一切都被视为了对象，但是我们操作的标识符实际上是对象的一个引用（reference）。

//创建一个引用，引用可以独立存在，并不一定需要与一个对象关联

String s;

通过将这个叫“引用”的标识符指向某个对象，之后便可以通过这个引用来实现操作对象了。

String str = new String("abc");

System.out.println(str.toString());

在 JDK1.2 之前，Java中的定义很传统：如果 reference 类型的数据中存储的数值代表的是另外一块内存的起始地址，就称为这块内存代表着一个引用。  
Java 中的垃圾回收机制在判断是否回收某个对象的时候，都需要依据“引用”这个概念。  
在不同垃圾回收算法中，对引用的判断方式有所不同：

* 引用计数法：为每个对象添加一个引用计数器，每当有一个引用指向它时，计数器就加1，当引用失效时，计数器就减1，当计数器为0时，则认为该对象可以被回收（目前在Java中已经弃用这种方式了）。
* 可达性分析算法：从一个被称为 GC Roots 的对象开始向下搜索，如果一个对象到GC Roots没有任何引用链相连时，则说明此对象不可用。

JDK1.2 之前，一个对象只有“已被引用”和"未被引用"两种状态，这将无法描述某些特殊情况下的对象，比如，当内存充足时需要保留，而内存紧张时才需要被抛弃的一类对象。

**四种引用类型**

所以在 JDK.1.2 之后，Java 对引用的概念进行了扩充，将引用分为了：强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）4 种，这 4 种引用的强度依次减弱。

**一，强引用**

Java中默认声明的就是强引用，比如：

Object obj = new Object(); //只要obj还指向Object对象，Object对象就不会被回收

obj = null; //手动置null

只要强引用存在，垃圾回收器将永远不会回收被引用的对象，哪怕内存不足时，JVM也会直接抛出OutOfMemoryError，不会去回收。如果想中断强引用与对象之间的联系，可以显示的将强引用赋值为null，这样一来，JVM就可以适时的回收对象了

**二，软引用**

软引用是用来描述一些非必需但仍有用的对象。**在内存足够的时候，软引用对象不会被回收，只有在内存不足时，系统则会回收软引用对象，如果回收了软引用对象之后仍然没有足够的内存，才会抛出内存溢出异常**。这种特性常常被用来实现缓存技术，比如网页缓存，图片缓存等。  
在 JDK1.2 之后，用java.lang.ref.SoftReference类来表示软引用。

下面以一个例子来进一步说明强引用和软引用的区别：  
在运行下面的Java代码之前，需要先配置参数 -Xms2M -Xmx3M，将 JVM 的初始内存设为2M，最大可用内存为 3M。

首先先来测试一下强引用，在限制了 JVM 内存的前提下，下面的代码运行正常

public class TestOOM {

public static void main(String[] args) {

testStrongReference();

}

private static void testStrongReference() {

// 当 new byte为 1M 时，程序运行正常

byte[] buff = new byte[1024 \* 1024 \* 1];

}

}

但是如果我们将

byte[] buff = new byte[1024 \* 1024 \* 1];

替换为创建一个大小为 2M 的字节数组

byte[] buff = new byte[1024 \* 1024 \* 2];

则内存不够使用，程序直接报错，强引用并不会被回收  


接着来看一下软引用会有什么不一样，在下面的示例中连续创建了 10 个大小为 1M 的字节数组，并赋值给了软引用，然后循环遍历将这些对象打印出来。

public class TestOOM {

private static List<Object> list = new ArrayList<>();

public static void main(String[] args) {

testSoftReference();

}

private static void testSoftReference() {

for (int i = 0; i < 10; i++) {

byte[] buff = new byte[1024 \* 1024];

SoftReference<byte[]> sr = new SoftReference<>(buff);

list.add(sr);

}

System.gc(); //主动通知垃圾回收

for(int i=0; i < list.size(); i++){

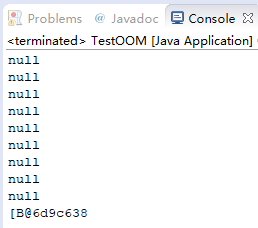
Object obj = ((SoftReference) list.get(i)).get();

System.out.println(obj);

}

}

}

打印结果：  


我们发现无论循环创建多少个软引用对象，打印结果总是只有最后一个对象被保留，其他的obj全都被置空回收了。  
这里就说明了在内存不足的情况下，软引用将会被自动回收。  
值得注意的一点 , 即使有 byte[] buff 引用指向对象, 且 buff 是一个strong reference, 但是 SoftReference sr 指向的对象仍然被回收了，这是因为Java的编译器发现了在之后的代码中, buff 已经没有被使用了, 所以自动进行了优化。  
如果我们将上面示例稍微修改一下：

private static void testSoftReference() {

byte[] buff = null;

for (int i = 0; i < 10; i++) {

buff = new byte[1024 \* 1024];

SoftReference<byte[]> sr = new SoftReference<>(buff);

list.add(sr);

}

System.gc(); //主动通知垃圾回收

for(int i=0; i < list.size(); i++){

Object obj = ((SoftReference) list.get(i)).get();

System.out.println(obj);

}

System.out.println("buff: " + buff.toString());

}

则 buff 会因为强引用的存在，而无法被垃圾回收，从而抛出OOM的错误。  


如果一个对象惟一剩下的引用是软引用，那么该对象是软可及的（softly reachable）。垃圾收集器并不像其收集弱可及的对象一样尽量地收集软可及的对象，相反，它只在真正 “需要” 内存时才收集软可及的对象。

**三，弱引用**

弱引用的引用强度比软引用要更弱一些，**无论内存是否足够，只要 JVM 开始进行垃圾回收，那些被弱引用关联的对象都会被回收**。在 JDK1.2 之后，用 java.lang.ref.WeakReference 来表示弱引用。  
我们以与软引用同样的方式来测试一下弱引用：

private static void testWeakReference() {

for (int i = 0; i < 10; i++) {

byte[] buff = new byte[1024 \* 1024];

WeakReference<byte[]> sr = new WeakReference<>(buff);

list.add(sr);

}

System.gc(); //主动通知垃圾回收

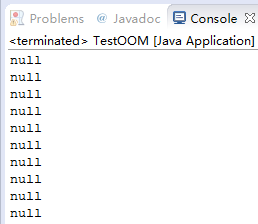
for(int i=0; i < list.size(); i++){

Object obj = ((WeakReference) list.get(i)).get();

System.out.println(obj);

}

}

打印结果：  


可以发现所有被弱引用关联的对象都被垃圾回收了。

**四，虚引用**

虚引用是最弱的一种引用关系，如果一个对象仅持有虚引用，那么它就和没有任何引用一样，它随时可能会被回收，在 JDK1.2 之后，用 PhantomReference 类来表示，通过查看这个类的源码，发现它只有一个构造函数和一个 get() 方法，而且它的 get() 方法仅仅是返回一个null，也就是说将永远无法通过虚引用来获取对象，虚引用必须要和 ReferenceQueue 引用队列一起使用。

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

/\*\*

\* Returns this reference object's referent. Because the referent of a

\* phantom reference is always inaccessible, this method always returns

\* <code>null</code>.

\*

\* @return <code>null</code>

\*/

public T get() {

return null;

}

public PhantomReference(T referent, ReferenceQueue<? super T> q) {

super(referent, q);

}

}

那么传入它的构造方法中的 ReferenceQueue 又是如何使用的呢？

**五，引用队列（ReferenceQueue）**

引用队列可以与软引用、弱引用以及虚引用一起配合使用，当垃圾回收器准备回收一个对象时，如果发现它还有引用，那么就会在回收对象之前，把这个引用加入到与之关联的引用队列中去。程序可以通过判断引用队列中是否已经加入了引用，来判断被引用的对象是否将要被垃圾回收，这样就可以在对象被回收之前采取一些必要的措施。

与软引用、弱引用不同，虚引用必须和引用队列一起使用。

引用的分类

引用计数算法和根搜索算法都需要通过判断引用的方式判断对象是否可回收。

在 JDK 1.2 之后，Java 将引用分成四种，按照引用强度从高到低的顺序依次是：强引用、软引用、弱引用、虚引用。

强引用是指在程序代码中普遍存在的引用。垃圾回收器永远不会回收被强引用关联的对象。

软引用描述还有用但并非必需的对象。只有在系统将要发生内存溢出异常时，被软引用关联的对象才会被回收。在 JDK 1.2 之后，提供了 SoftReference 类实现软引用。

弱引用描述非必需的对象，其强度低于软引用。被弱引用关联的对象只能存活到下一次垃圾回收发生之前，当垃圾回收器工作时，被弱引用关联的对象一定会被回收。在 JDK 1.2 之后，提供了 WeakReference 类实现弱引用。

虚引用是最弱的引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用取得一个对象实例。为一个对象设置虚引用关联的唯一目的就是能在这个对象被回收时收到一个系统通知。在 JDK 1.2 之后，提供了 PhantomReference 类实现虚引用。

作者：力扣 (LeetCode)

链接：https://leetcode-cn.com/leetbook/read/java-interview-highlights/en7epe/

来源：力扣（LeetCode）

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。