深入理解Java引用(一)

# **什么是引用**

       我们知道Java是一门纯面向对象的语言，我们在使用Java语言编程时，到处都在使用对象(<<Thinking in Java>>一书中提到"Everything is object")。程序在运行时，对象是在堆内存(heap)中存储的，那么我们如何来访问对象呢？在C/C++中是通过指针，而是Java中是通过引用，引用指向了对象在内存(heap)中的地址。

       Java支持8种基本类型，而这8种基本类型并不是对象，但是有相应的8种包装类型，所以只有对8种基本类型对应的包装类访问才是基于引用。

那么为什么设计这8种基本类型呢？这个主要是基于内存和性能的考量。从内存上来看对于32位的JVM来说一个Integer数据占用**4个字节（对象头） + 4个字节（类型指针） +4个字节（实例数据） + 4个字节（对齐填充）**=16个字节，而一个int只占用4个字节。从性能上来看当JVM执行方法时，对于基本类型的数据直接从局部变量表中获取即可，而对象类型需要根据reference定位到heap，所以JVM的JIT(即时编译器)在会根据逃逸分析的标量替换的优化。

       引用本身也是占用内存空间的，一个引用值在32位JVM上占用32位(4个字节)，在64位JVM上占用64位(8个字节)，但通过指针压缩后占用4个字节。

# **引用的类型**

       根据Java语言规范，引用类型有：类类型(class types)，接口类型(interface types)，数组类型(array types)，类型变量类型(type variables)（实际上泛型类型，泛型擦除后运行时的实际类型可能是类类型，接口类型或者数组类型）。

根据JVM规范，引用类型有：类类型(class types)，接口类型(interface types)，数组类型(array types)。

public class ReferenceTypeTest {

@Test

public void test() {

//类类型

Object obj = new Object();

//数组类型

Object[] objArr = new Object[4];

// 接口类型

Serializable serializable = new Serializable() {

};

}

}

       所有引用类型有一个默认值：null，当一个引用不指向任何对象的时候，它的值就用null来表示。一个为null的引用，起初并不具备任何实际的运行期类型，但是它可转换为任意的引用类型。但是null又不属于任何类型。

public class ReferenceNullValueTest {

@Test

public void test() {

//没有指向任何对象

Object obj = null;

// null可强制转换成任何类型

String s = (String) obj;

Assert.assertNull(s);

Integer i = (Integer) obj;

Assert.assertNull(i);

// 但是null不属于任何类型

Assert.assertFalse(null instanceof Object);

Assert.assertFalse(null instanceof String);

Assert.assertFalse(null instanceof Integer);

}

}

# **值传递与引用传递**

        Java语言规范与JVM规范中并没有对值传递与引用传递有描述，值传递与引用传递都是在方法调用时的叫法，以方便理解，或者从某种程度上来说都是值传递，因为引用传递也是传递的引用值(内存地址)的copy。

每个方法在执行时都会生成一个栈帧，而局部变量表和操作数据栈就是栈帧的一部分，局部变量表可理解一个数组，这个数组的大小在编译后已经决定了，局部变量表依次存储了this(如果是实例方法)、方法入参、方法内部使用的局部变量，而操作数栈是存储运行字节码指令需要的操作数及操作结果。

理解方法执行时栈帧的局部变量表内存数据结构对这两个概念会有深入的理解。下边以这个例子来说明值传递与引用传递。通过“javap -verbose ReferenceTransmitTest.class”可以获取字节码。

import org.junit.Assert;

import org.junit.Test;

/\*\*

\* 测试值传递与引用传递

\*/

public class ReferenceTransmitTest {

private class Foo {

public int a;

}

/\*\*

\* 值传递。

\*/

@Test

public void testIntTransmit() {

// int a = 100 反编译后对应如下指令:

// 0: bipush 100 //将100压入操作数栈

// 2: istore\_1 //将操作数栈中的数据存入该方法栈桢的局部变量表下标为1的变量(即变量a)中

int a = 100;

System.out.println("before testIntTransmit() a = " + a);

// 参考addOne方法的相应注释

addOne(a);

// 反编译后对应指令如下：

// 48: iload\_1 //将局部变量表中下标为1的变量(即变量a)压入操作数栈

// 49: invokevirtual #7 // Method java/lang/StringBuilder.append:(I)Ljava/lang/StringBuilder;

// 52: invokevirtual #8 // Method java/lang/StringBuilder.toString:()Ljava/lang/String;

// 55: invokevirtual #9 // Method java/io/PrintStream.println:(Ljava/lang/String;)V

System.out.println("after testIntTransmit() a = " + a);

Assert.assertEquals(100, a);

// 分析：

// testIntTransmit()与addOne()方法有各自的方法栈桢

// 调用addOne()方法传递的入参a的值为100，且该值copy到addOne()的方法栈桢中

// 变量a值在testIntTransmit()与addOne()的方法栈桢中互不影响

}

/\*\*

\* 引用传递。但是由于Integer不可变的特性，测试结果同值传递。

\*/

@Test

public void testIntegerTransmit() {

// Integer a = 100 反编译后对应如下指令:

// 0: bipush 100 // 将100压入操作数栈

// 2: invokestatic #15 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;

// 5: astore\_1 // 将操作数栈中的数据存入该方法栈桢的局部变量表下标为1的变量中

// 分析：这里做了一次装箱操作，变量a为指向堆内存的引用。也就是说该方法栈桢的局部变量表下标为1存储的是一个引用值

Integer a = 100;

System.out.println("before testIntTransmit() a = " + a);

// 参考addOne方法的相应注释

addOne(a);

// 51: aload\_1 //将局部变量表中下标为1的变量(即变量a)压入操作数栈

// 52: invokevirtual #16 // Method java/lang/StringBuilder.append:(Ljava/lang/Object;)Ljava/lang/StringBuilder;

// 55: invokevirtual #8 // Method java/lang/StringBuilder.toString:()Ljava/lang/String;

// 58: invokevirtual #9 // Method java/io/PrintStream.println:(Ljava/lang/String;)V

System.out.println("after testIntTransmit() a = " + a);

Assert.assertEquals(100, a.intValue());

// 分析：

// 由于Integer对象的不可变性(没有提供任何可以修改内部属性的途径)，

// 在调用addOne方法时，不会也不可能改变a指向内存中的对象实例的值

}

/\*\*

\* 引用传递。但是由于String不可变的特性，测试结果同值传递。

\*/

@Test

public void testStringTransmit() {

// String a = "Hello" 反编译后对应的指令。

// 0: ldc #19 // String Hello 从常量池中加载常量数据到操作数据栈中

// 2: astore\_1 // 将操作数据栈中的数据存储该方法栈桢的局部变量表下标为1中

// 此处可以看出"Hello"这个值存在于运行时常量池中中

String a = "Hello";

System.out.println("before testStringTransmit() a = " + a);

// 参考append()方法的注释

append(a);

System.out.println("after testStringTransmit() a = " + a);

Assert.assertEquals("Hello", a);

// 分析：

// 由于String对象的不可变性(没有提供任何可以修改内部属性的途径)，

// 在调用append方法时，不会也不可能改变a指向内存中的对象实例的值

}

/\*\*

\* 引用传递。

\*/

@Test

public void testObjectTransmit() {

// Foo foo = new Foo() 反编译后对应的指令如下：

// 0: new #24 // class ReferenceTransmitTest$Foo 申请一块Foo实例对象内存,此时所有字段默认为0，并将其引用值压入栈顶

// 3: dup // 复制栈顶数据值并将复制值压入栈顶，供后续JVM调用构造方法用

// 4: aload\_0

// 5: aconst\_null

// 6: invokespecial #25 // Method ReferenceTransmitTest$Foo."<init>":(LReferenceTransmitTest;LReferenceTransmitTest$1;)V 包括：初始化值、构造语句块、构造方法

// 9: astore\_1 // 将操作数据栈中的数据存储到该方法栈桢的局部变量表下标为1中

Foo foo = new Foo();

System.out.println("before testObjectTransmit() foo.a = " + foo.a);

// 参考addOne方法的注释

addOne(foo);

System.out.println("after testObjectTransmit() foo.a = " + foo.a);

Assert.assertEquals(1, foo.a);

//分析：

// testObjectTransmit()与addOne()方法访问及修改同一对象内存的值，所以在testObjectTransmit()中会看到addOne()方法对属性a值的修改

}

public void addOne(int a) {

//JVM调用该方法时，会将参数a放在该方法栈桢的局部变量表下标为1的位置

System.out.println("before addOne() a = " + a);

//++a 反编译后得到如下指令，它的意思是将该方法栈桢的局部变量表下标为1的值加1，然后写回局部变量表下标为1中

//25: iinc 1, 1

++a;

System.out.println("after addOne() a = " + a);

}

public void addOne(Integer a) {

//JVM调用该方法时，会将参数a放在该方法栈桢的局部变量表下标为1的位置

System.out.println("before addOne() a = " + a);

// ++a 反编译后得到如下指令

// 25: aload\_1 //将该方法栈桢的局部变量表下标为1的值加载到操作数栈是

// 26: invokevirtual #18 // Method java/lang/Integer.intValue:()I //这里实际上做了个拆箱操作，获取a的int值并且压入操作数据栈

// 29: iconst\_1 // 将常量1压入操作数据栈

// 30: iadd // 将操作数栈中的两个操作数相加，把结果压入操作数栈

// 31: invokestatic #15 // Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer; //对操作数栈中的数据做了一次装箱操作，并且把返回结果(是一个引用地址)压入操作数栈

// 34: astore\_1 // 将操作数栈中的值存储到该方法栈桢的局部变量表下标为1中

++a;

// 上述指令等同于如下代码

// int b = a.intValue();

// int b = b + 1;

// a = Integer.valueOf(b);

System.out.println("after addOne() a = " + a);

//分析:

// 由于Integer的不可变性(没有提供任何可以修改内部属性的途径)，

// 编译器通过拆箱之后获取int值，然后对int值做了加1的操作，

// 最后通过装箱，生成了一个新的Integer对象，同时重新修改引用类型a的值指向新的Integer对象的内存地址。

}

public void append(String a) {

//JVM调用该方法时，会将参数a放在该方法栈桢的局部变量表下标为1的位置

System.out.println("before append() a = " + a);

// a += " World!" 反编译后得到如下指令

// 25: new #3 // class java/lang/StringBuilder

// 28: dup

// 29: invokespecial #4 // Method java/lang/StringBuilder."<init>":()V

// 32: aload\_1

// 33: invokevirtual #6 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;

// 36: ldc #33 // String World!

// 38: invokevirtual #6 // Method java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;

// 41: invokevirtual #8 // Method java/lang/StringBuilder.toString:()Ljava/lang/String;

// 44: astore\_1

a += " World!";

// 上述指令等同于如下代码：

// StringBuilder sb = new StringBuilder();

// sb.append(a);

// sb.append(" World");

// a = sb.toString();

System.out.println("after append() a = " + a);

//分析:

// 由于String的不可变性(没有提供任何可以修改内部属性的途径)，

// 编译器对String的连接操作，通过生成StringBuilder来完成的

// 最后通过StringBuilder的toString()方法生成了一个新的字符串对象，同时重新修改引用类型a的值指向新的String对象的内存地址。。

}

public void addOne(Foo foo) {

//JVM调用该方法时，会将参数foo放在该方法栈桢的局部变量表下标为1的位置

System.out.println("before addOne() foo.a = " + foo.a);

// foo.a++ 反编译后对应的指令如下：

// 28: aload\_1 // 将该方法栈桢的局部变量表下标为1(即foo)的值加载到操作数据栈

// 29: dup // 复制栈顶数据值并将复制值压入栈顶

// 30: getfield #27 // Field ReferenceTransmitTest$Foo.a:I 获取对象属性a的值并且压入操作数栈

// 33: iconst\_1 // 将常量1压入操作数栈

// 34: iadd // 将操作数栈中数据相加然后把结果压入操作数栈

// 35: putfield #27 // Field ReferenceTransmitTest$Foo.a:I 将操作数栈的值写入对象foo的属性a中

foo.a++;

System.out.println("after addOne() foo.a = " + foo.a);

}

}

上述单元测试的结论：

* 基本数据类型参数传值为值传递，对传入调用方法的入参变量修改不会影响原始值。
* 包装器类型及String类型参数值值为引用传递，但由于包装器类与String类的不变性，编译器对其进行了特殊处理，并不会改变原始值，所以也可以理解为值传递。
* 引用传递，对传入调用方法的入参变量和原始变量指向同一个内存地址（同一个对象），所以对参数的修改会影响到实际的对象。

# **GC算法**

## **4.1 引用计数算法**

**1、基本思想**

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；

当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。

**2、优缺点**

优点：实现简单，判定效率高。

缺点：很难解决对象之间相互循环引用的问题。（因此主流的java虚拟机中没有选用此方法）

## **4.2 可达性分析算法**

reachable通过一系列被称为"GC Roots" 的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，当一个对象到GC Roots没有任何引用链相(**对象不可达**)时，则证明此对象不可用。

在主流的商用程序语言的主流实现中，都是通过可达性分析算法来判断对象是否存活的。

可作为GC Roots的对象包括：

* **虚拟机栈（栈帧中的本地变量表）中引用的对象**
* **方法区中类静态属性应用的对象。**
* **方法区中常量引用的对象**
* **本地方法栈中JNI引用的对象**

# **基于强弱区分引用**

       我们平时开发大部分都在用强引用，如:Object obj = new Object()。在JDK1.2之前，只有强引用，对于强引用，即使发生OOM（java.lang. OutOfMemoryError），垃圾回收器也是不会回收它的，但是对于某些场景当内存不足时，希望垃圾回收器能清理这些对象，譬如缓存，连接池等，对这些对象就显得无能为力了。所在以JDK1.2及以后，在API层面及JVM根据引用的强弱程度将引用分为：强引用(Strong Reference)、软引用(Soft Reference)、弱引用(Weak Reference)、虚引用(Phantom Reference)，这四种引用强度依次逐渐减弱。

       那么垃圾回收器是如何根据强弱程度对对象进行回收的呢？垃圾收集器实际上会选出一些对象作为root，若对象与这些root对象之间有路径存在，那么这些对象是可达的，根据对象引用的引用强弱程度，依次可分为强可达(Strong Reachable)，软可达(Soft Reachable)，弱可达(Weak  Reachable)，虚可达(Phantom  Reachable)，其中软引用、弱引用、虚引用在一定条件下，都是可以被回收的。不同的可达性状态影响着垃圾回收器的回收策略，这个就是目前JVM采用的可达性分析算法。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 可达性 | 垃圾回收 | 如何使用 | 使用场景 |
| 强引用 | 强可达 | 即使OOM,GC也不会回收强引用对象 |  | 一般开发都是在使用强引用 |
| 软引用 | 软可达 | GC会在OOM前回收软引用对象 | 基于SoftReference和ReferenceQueue | 多用于框架中，如缓存、连接池 |
| 弱引用 | 弱可达 | GC发生时会回收弱引用对象 | 基于WeakReference和ReferenceQueue | 多用于框架中，如缓存、连接池 |
| 虚引用 | 虚可达 | 随时可能会被回收 | 基于PhantomReference和ReferenceQueue | 跟踪对象的回收 |

## **5.1 强引用**

       强引用就是程序中一般使用的引用类型。强引用具备如下特点：

* 强引用可以直接访问目标对象
* 强引用所指向的对象在任何时候都不会被系统回收，虚拟机宁愿抛出OOM异常，也不会回收强引用所指向的对象
* 强引用可能导致内存泄露

public class StrongReferenceTest {

/\*\*

\* JVM参数：

\* -Xms10m -Xmx10m -XX:+PrintHeapAtGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+HeapDumpOnOutOfMemoryError

\*/

@Test(expected = OutOfMemoryError.class)

public void testStrongReference() {

int size = 1000000;

//用list保持强引用，即使发生OOM，垃圾回收器也不会回收list中的对象

List<Integer> list = new ArrayList(size);

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

list.add(i);

}

}

}

## **5.2 软引用**

       软引用是比强引用弱一些的引用类型。一个对象只持有软引用，那么当堆空间不足时，就会回收该对象。软引用使用java.lang.ref.SoftReference类实现。使用场景：缓存

import org.junit.Assert;

import org.junit.Test;

import java.lang.ref.ReferenceQueue;

import java.lang.ref.SoftReference;

import java.util.ArrayList;

import java.util.List;

public class SoftReferenceTest {

private class Student {

private int id;

private String name;

public Student(int id, String name) {

this.id = id;

this.name = name;

}

@Override

public String toString() {

return "Student{" + "id=" + id + ", name='" + name + '\'' + '}';

}

}

/\*\*

\* JVM参数：

\* -Xms10m -Xmx10m -XX:+PrintHeapAtGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+HeapDumpOnOutOfMemoryError

\*/

@Test

public void testSoftReferenceWithoutQueue() {

int softRefSize = 100000;

List<SoftReference> list = new ArrayList<SoftReference>(softRefSize);

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

// 注意此处创建的Student对象除了SoftReference没有其它强引用

// 若此处Student还被其它GC root对象关联，那么它仍然是强引用对象

list.add(new SoftReference<Student>(new Student(i, "张三")));

}

// 可用内存充足，没有发生GC

Assert.assertEquals(0, getGcObjectSize(list));

// 强制执行gc,大部分情况下垃圾回收器都会立即执行，若没有执行，请在gc后sleep一段时间

System.gc();

// 可用内存充足，垃圾回收器不会回收软引用对象

Assert.assertEquals(0, getGcObjectSize(list));

// 创建一个大对象，让heap内存紧张，发生full gc，清除软引用

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

System.out.println("getGcObjectSize(list) = " + getGcObjectSize(list));

//被垃圾回收器回收的软引用对象的数量>0

Assert.assertTrue(getGcObjectSize(list) > 0);

}

/\*\*

\* JVM参数：

\* -Xms10m -Xmx10m -XX:+PrintHeapAtGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+HeapDumpOnOutOfMemoryError

\*/

@Test

public void testSoftReferenceWithQueue() throws InterruptedException {

// 定义一个软引用的队列

final ReferenceQueue<Student> softQueue = new ReferenceQueue<Student>();

class StudentSoftReference extends SoftReference<Student> {

private int id;

public StudentSoftReference(Student referent, ReferenceQueue<? super Student> q) {

super(referent, q);

id = referent.id;

}

}

// 启动一个线程，实时追踪对象回收的情况

Thread t = new Thread(new Runnable() {

public void run() {

while (true) {

try {

StudentSoftReference sr = (StudentSoftReference) softQueue.remove();

System.out.println("Student id " + sr.id + " has been deleted by GC!");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

});

t.setDaemon(true); //设置为守护线程

t.start();

int softRefSize = 10;

List<SoftReference> list = new ArrayList<SoftReference>(softRefSize);

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

// 若垃圾回收器准备回收一个对象时，如果发现它还有弱引用，就会在回收对象后，将这个弱引用加入引用队列,

// 以通知应用程序的回收情况

list.add(new StudentSoftReference(new Student(i, "张三"), softQueue));

}

// 可用内存充足，没有发生GC

Assert.assertEquals(0, getGcObjectSize(list));

// 强制执行gc,大部分情况下垃圾回收器都会立即执行，若没有执行，请在gc后sleep一段时间

System.gc();

// 可用内存充足，垃圾回收器不会回收软引用对象

Assert.assertEquals(0, getGcObjectSize(list));

// 创建一个大对象，让内存资源紧张，发生full gc，清除软引用

byte[] bArr = new byte[1024 \* 1010 \* 6];

System.out.println("getGcObjectSize(list) = " + getGcObjectSize(list));

Assert.assertTrue(getGcObjectSize(list) > 0);

// 避免主线程退出时，守护线程还没结束

Thread.sleep(2000);

}

/\*\*

\* 获取被垃圾回收器清除的对象数量

\*

\* @param list

\* @return

\*/

private int getGcObjectSize(List<SoftReference> list) {

int gcStuSize = 0;

for (SoftReference sr : list) {

if (sr.get() == null) {

gcStuSize++;

}

}

return gcStuSize;

}

}

## **5.3 弱引用**

       弱引用比软引用弱一些，在系统GC时，只要发现弱引用，总将将对象进行回收。弱引用使用java.lang.ref.WeakReference类实现。使用场景：缓存

import org.junit.Assert;

import org.junit.Test;

import java.lang.ref.WeakReference;

import java.util.ArrayList;

import java.util.List;

public class WeakReferenceTest {

private class Student {

private int id;

private String name;

public Student(int id, String name) {

this.id = id;

this.name = name;

}

@Override

public String toString() {

return "Student{" + "id=" + id + ", name='" + name + '\'' + '}';

}

}

/\*\*

\* JVM参数：

\* -Xms10m -Xmx10m -XX:+PrintHeapAtGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+HeapDumpOnOutOfMemoryError

\*/

@Test

public void testWeakReferenceWithoutQueue() {

int weakRefSize = 1000;

List<WeakReference> list = new ArrayList<WeakReference>(weakRefSize);

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

// 注意此处创建的Student对象除了WeakReference没有其它强引用

// 若此处Student还没其它GC root对象关联，那么它仍然是强引用对象

list.add(new WeakReference<Student>(new Student(i, "张三")));

}

// 可用内存充足，没有发生GC

Assert.assertEquals(0, getGcObjectSize(list));

// 强制执行gc,大部分情况下垃圾回收器都会立即执行，若没有执行，请在gc后sleep一段时间

System.gc();

// 不管内存情况是否充足，软引用对象都会被垃圾回收器回收

Assert.assertEquals(weakRefSize, getGcObjectSize(list));

}

// 基于队列的测试同SoftReferenceTest#testPhantomReferenceWithQueue

/\*\*

\* 获取被垃圾回收器清除的对象数量

\*

\* @param list

\* @return

\*/

private int getGcObjectSize(List<WeakReference> list) {

int gcStuSize = 0;

for (WeakReference sr : list) {

if (sr.get() == null) {

gcStuSize++;

}

}

return gcStuSize;

}

}

## **5.4 虚引用**

       虚引用是所有引用类型中最弱的一个。一个持有虚引用的对象，和没有引用几乎是一样的，随时都可能被垃圾回收器回收。当试图通过虚引用的get()方法取得强引用时，总是会失败。软引用使用java.lang.ref.PhantomReference类实现。使用场景：对象回收跟踪。

import org.junit.Assert;

import org.junit.Test;

import java.lang.ref.PhantomReference;

import java.lang.ref.ReferenceQueue;

public class PhantomReferenceTest {

private class Student {

private int id;

private String name;

public Student(int id, String name) {

this.id = id;

this.name = name;

}

// 可以解注下这段代码，看下打印日志。下一篇源码剖析解释下这个。

//@Override

//protected void finalize() {

// System.out.println("Student id " + id + " has been finalize()");

//}

@Override

public String toString() {

return "Student{" + "id=" + id + ", name='" + name + '\'' + '}';

}

}

/\*\*

\* JVM参数：

\* -XX:+PrintHeapAtGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+HeapDumpOnOutOfMemoryError

\*/

@Test

public void testPhantomReferenceWithQueue() {

final ReferenceQueue<Student> phantomQueue = new ReferenceQueue<Student>();

class StudentPhantomReference extends PhantomReference<Student> {

private int id;

public StudentPhantomReference(Student referent, ReferenceQueue<? super Student> q) {

super(referent, q);

id = referent.id;

}

}

// 启动一个线程，实时追踪对象回收的情况

Thread t = new Thread(new Runnable() {

public void run() {

while (true) {

try {

StudentPhantomReference sr = (StudentPhantomReference) phantomQueue.remove();

System.out.println("Student id " + sr.id + " has been deleted");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

});

t.setDaemon(true);//设置为守护线程

t.start();

StudentPhantomReference phantomReference = new StudentPhantomReference(new Student(1, "张三"), phantomQueue);

// 通过get()方法获取到的Student对象为null

Assert.assertNull(phantomReference.get());

// 强制执行gc,大部分情况下垃圾回收器都会立即执行，若没有执行,请在gc后sleep一段时间

System.gc();

//无论内存是否充足，Student对象都被回收，观察日志打印

}

}

# 6.总结

       Java中的类型分两类，一是基本类型，另一类就是引用类型，引用类型的值就是指向对象内存的内存地址。

引用类型主要分为三类：类类型，接口类型，数组类型。还有一种特殊的引用类型值null, 它既可以转换为任意类型，但又不属于任意类型。

在Java中传递分为值传递和引用传递，值传递不会改变原始值，但引用传递会，可通过跟踪执行字节码过程的本地变量表和操作数栈来理解这个概念。

垃圾回收器在回收对象时，会根据引用对象的可达性进行进行不同的回收，其中强引用对应强可达，软引用对应软可达，弱引用对应弱可达，虚引用对应虚可达。Java提供了一套API来支持软引用，弱引用和虚引用.