链滴

Java 四种引用类型

作者：xumiao

原文链接：https://ld246.com/article/1588002097178

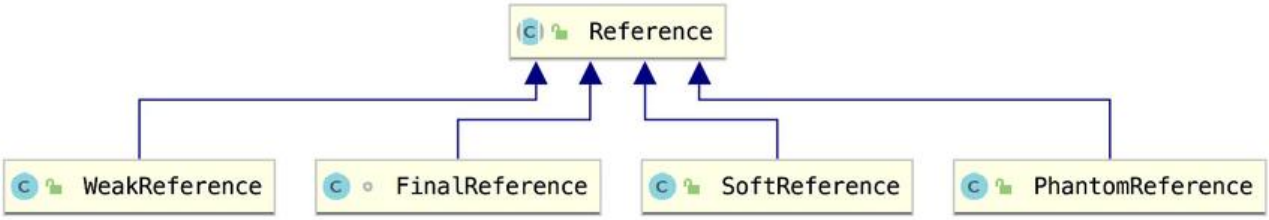
来源网站：链滴

介绍

Java中提供了四种引用类型，分别如下：

- FinalReference（强引用）
- SoftReference（软引用）
- WeakReference（弱引用）
- PhantomReference（虚引用）

其中FinalReference是包权限无法使用，其它三种引用类型都是公共的可以在应用中使用，下面是Ref
rence的类结构。

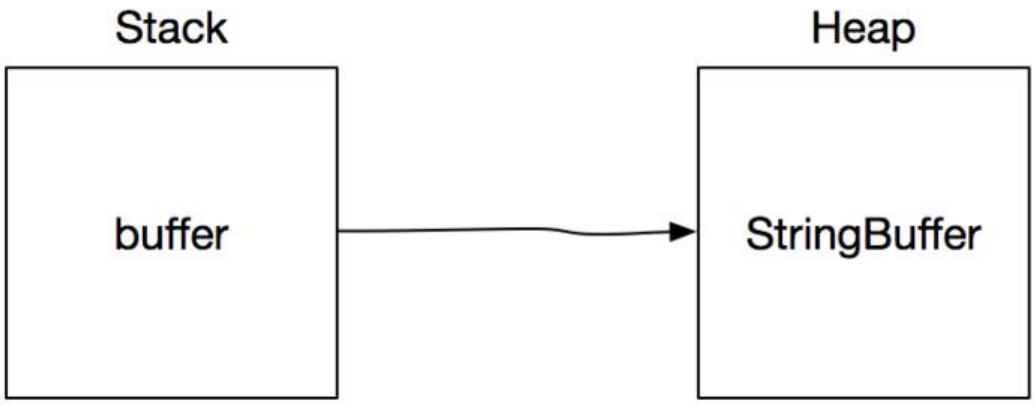


FinalReference

Java中的强引用其实就是new对象，可以通过引用操作堆中的对象（和C中的指针类似），例如：

```
StringBuffer buffer = new StringBuffer("HelloWorld!");
```

变量buffer指向StringBuffer所在的堆空间，通过buffer来进行操作。



FinalReference的特性

1.可以直接访问目标对象；
2.指向的对象不会被GC回收□□recycle，当J

原文链接：Java 四种引用类型

M内存不足时会抛出OOM异常终端程序；

3.基于上面第2点，当其它需要释放的代码块持FinalReference会造成内存泄露；

SoftReference

SoftReference即软引用，当JVM堆空间使用率到达阈值的时候会触发GC回收，我们可以用它来实现内存敏感的缓存，SoftReference的特性是可以保留对Java对象软引用的实例，软引用的实例并不会止GC进行回收，在GC线程进行回收之前我们可以通过它的get方法获取到对象的强引用，一旦对象被C回收后get方法将返回null。

下面我们通过代码来实践，通过设置JVM堆内存大小为2M来模拟：

```
java -Xms2M -Xmx2M -verbose:gc -XX:+PrintGCDetails [class文件]
```

- -Xms2M

堆大小固定为2M

- -verbose:gc

输出虚拟机中GC的详细情况

- -XX:+PrintGCDetails

在控制台上打印出GC具体细节

```
public static void testSoftRef() {
    SoftReference<byte[]> softRef1 = new SoftReference<>(new byte[1024 * 300]);
    SoftReference<byte[]> softRef2 = new SoftReference<>(new byte[1024 * 300]);
    SoftReference<byte[]> softRef3 = new SoftReference<>(new byte[1024 * 300]);
    SoftReference<byte[]> softRef4 = new SoftReference<>(new byte[1024 * 300]);
    SoftReference<byte[]> softRef5 = new SoftReference<>(new byte[1024 * 300]);

    System.out.println(softRef1.get());
    System.out.println(softRef2.get());
    System.out.println(softRef3.get());
    System.out.println(softRef4.get());
    System.out.println(softRef5.get());
}
```

结果如下，我们可以看到上面分配的byte长度是超出年轻代大小的，当内存不足时的确触发了GC进回收，正如上面所说的那样。

```
[GC (Allocation Failure) [PSYoungGen: 510K->320K(1024K)] 510K->320K(1536K), 0.0037082 s
cs] [Times: user=0.01 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 825K->432K(1024K)] 825K->432K(1536K), 0.0016384 s
cs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 942K->496K(1024K)] 942K->512K(1536K), 0.0113468 s
cs] [Times: user=0.01 sys=0.00, real=0.01 secs]
[GC (Allocation Failure) --[PSYoungGen: 984K->984K(1024K)] 1300K->1493K(1536K), 0.00147
4 secs] [Times: user=0.00 sys=0.01, real=0.00 secs]
[Full GC (Ergonomics) [PSYoungGen: 984K->34K(1024K)] [ParOldGen: 509K->493K(512K)] 14
3K->527K(1536K), [Metaspace: 3282K->3282K(1056768K)], 0.0131428 secs] [Times: user=0.01
sys=0.00, real=0.02 secs]
```

```
[Full GC (Ergonomics) [PSYoungGen: 334K->334K(1024K)] [ParOldGen: 493K->492K(512K)] 8
7K->827K(1536K), [Metaspace: 3282K->3282K(1056768K)], 0.0106514 secs] [Times: user=0.00
sys=0.00, real=0.01 secs]
[Full GC (Allocation Failure) [PSYoungGen: 334K->0K(1024K)] [ParOldGen: 492K->397K(512K)]
827K->397K(1536K), [Metaspace: 3282K->3282K(1056768K)], 0.0042212 secs] [Times: user=0.
1 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 300K->332K(1024K)] 697K->729K(1536K), 0.0002648 s
cs] [Times: user=0.00 sys=0.00, real=0.00 secs]
null
null
null
[B@610455d6
[B@511d50c0
Heap
 PSYoungGen      total 1024K, used 646K [0x00000007bfe80000, 0x00000007c0000000, 0x000
0007c0000000)
  eden space 512K, 61% used [0x00000007bfe80000,0x00000007bfecea50,0x00000007bff000
0)
  from space 512K, 64% used [0x00000007bff80000,0x00000007bffd3010,0x00000007c00000
0)
  to   space 512K, 0% used [0x00000007bff00000,0x00000007bff00000,0x00000007bff80000)
 ParOldGen       total 512K, used 397K [0x00000007bfe00000, 0x00000007bfe80000, 0x000000
7bfe80000)
  object space 512K, 77% used [0x00000007bfe00000,0x00000007bfe63570,0x00000007bfe80
00)
 Metaspace       used 3291K, capacity 4500K, committed 4864K, reserved 1056768K
  class space    used 363K, capacity 388K, committed 512K, reserved 1048576K
```

WeakReference

WeakReference即弱引用，当触发GC时，无论JVM堆内存是否足够对象都会被回收，下面进行测试：

```
    public static void testWeakRef() {
        byte[] buffer = new byte[1024 * 500];
        WeakReference<byte[]> weakReference = new WeakReference<>(buffer);
        System.out.println("GC前：" + weakReference.get());
        buffer = null;
        //手动触发GC
        System.gc();
        System.out.println("GC后：" + weakReference.get());
    }
```

结果如下：

```
GC前：[B@610455d6
GC后：null
```

SoftReference和WeakReference都适用于保存可选的缓存数据，在系统内存不足时，将回收缓存的据不会导致OOM，并且缓存也能存在很长一段时间。

PhantomReference

PhantomReference即虚引用，是所有类型中最弱的，它几乎是没有引用因为随时会被GC回收，当

用它的get方法获取强引用时始终都是返回null，它必须要和ReferenceQueue一起使用，用来跟踪垃回收过程。

当GC要回收对象时，如果发现PhantomReference后将进行GC然后销毁该对象，并将PhantomReference添加到ReferenceQueue中，判断是否向ReferenceQueue添加了PhantomReference可以确定否需要对引用的对象进行回收，如果ReferenceQueue中存在PhantomReference那么在回收引用对之前可以进行一些额外的操作。

```
    public static void testPhantomRef() {
        byte[] buffer = new byte[1024 * 500];
        ReferenceQueue<Object> referenceQueue = new ReferenceQueue<>();
        PhantomReference<byte[]> phantomReference = new PhantomReference<>(buffer,refe
enceQueue);
        buffer = null;
        System.out.println("GC前：" + phantomReference.get());
        System.gc();
        System.out.println("GC后：" + phantomReference.get());
    }
```

结果如下：

```
GC前：null
GC后：null
```

关于PhantomReference的get方法总是返回null。

```
    /**
     * 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;
    }
```

WeakHashMap

顾名思义，它和HashMap一样都是实现了Map接口，只不过它使用的是WeakReference作为存储，eakHashMap是典型的弱引用例子。

```
public class WeakHashMap<K,V>
    extends AbstractMap<K,V>
    implements Map<K,V> {}

private static class Entry<K,V> extends WeakReference<Object> implements Map.Entry<K,V>
{}
```

需要注意的是如果WeakHashMap的key在系统中是FinalReference强引用的, 那么WeakHashMap退化为一个普通的HashMap，因为它不能被GC回收。

参考资料

再谈四种引用状态

常用JVM命令参数

Do You Really Know the 4 Reference Types in Java?

原文链接：Java 四种引用类型