<https://mp.weixin.qq.com/s/pA0K3xOBdE-nXNxOYFbGrg>

## Java中当对象不再使用时，不赋值为null会导致什么后果 ？

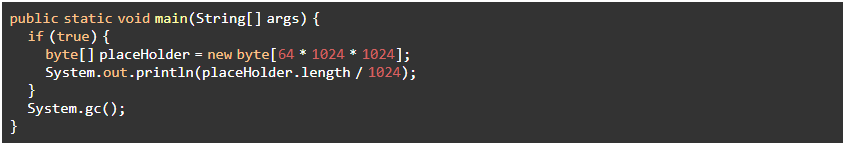
## 前言

许多Java开发者都曾听说过“不使用的对象应手动赋值为null“这句话，而且好多开发者一直信奉着这句话；问其原因，大都是回答“有利于GC更早回收内存，减少内存占用”，但再往深入问就回答不出来了。

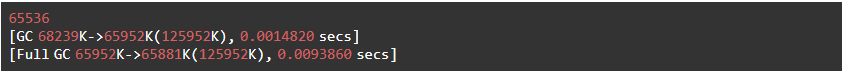
鉴于网上有太多关于此问题的误导，本文将通过实例，深入JVM剖析“对象不再使用时赋值为null”这一操作存在的意义，供君参考。本文尽量不使用专业术语，但仍需要你对JVM有一些概念。

## 示例代码

我们来看看一段非常简单的代码：

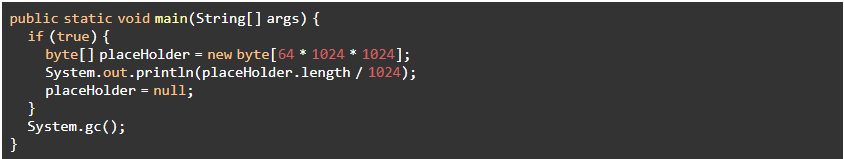


我们在if中实例化了一个数组placeHolder，然后在if的作用域外通过System.gc();手动触发了GC，其用意是回收placeHolder，因为placeHolder已经无法访问到了。来看看输出：



Full GC 65952K->65881K(125952K)代表的意思是：本次GC后，内存占用从65952K降到了65881K。意思其实是说GC没有将placeHolder回收掉，是不是不可思议？

下面来看看遵循“不使用的对象应手动赋值为null“的情况：



其输出为：



这次GC后内存占用下降到了345K，即placeHolder被成功回收了！对比两段代码，仅仅将placeHolder赋值为null就解决了GC的问题，真应该感谢“不使用的对象应手动赋值为null“。

等等，为什么例子里placeHolder不赋值为null，GC就“发现不了”placeHolder该回收呢？这才是问题的关键所在。

## 运行时栈

### 典型的运行时栈

如果你了解过编译原理，或者程序执行的底层机制，你会知道方法在执行的时候，方法里的变量（局部变量）都是分配在栈上的；当然，对于Java来说，new出来的对象是在堆中，但栈中也会有这个对象的指针，和int一样。

比如对于下面这段代码：



其运行时栈的状态可以理解成：

| **索引** | **变量** |
| --- | --- |
| 1 | a |
| 2 | b |
| 3 | c |

“索引”表示变量在栈中的序号，根据方法内代码执行的先后顺序，变量被按顺序放在栈中。

再比如：



这时运行时栈就是：

| **索引** | **变量** |
| --- | --- |
| 1 | a |
| 2 | b |
| 3 | c |
| 4 | d |

容易理解吧？其实仔细想想上面这个例子的运行时栈是有优化空间的。

### Java的栈优化

上面的例子，main()方法运行时占用了4个栈索引空间，但实际上不需要占用这么多。当if执行完后，变量a、b和c都不可能再访问到了，所以它们占用的1～3的栈索引是可以“回收”掉的，比如像这样：

| **索引** | **变量** |
| --- | --- |
| 1 | a |
| 2 | b |
| 3 | c |
| 1 | d |

变量d重用了变量a的栈索引，这样就节约了内存空间。

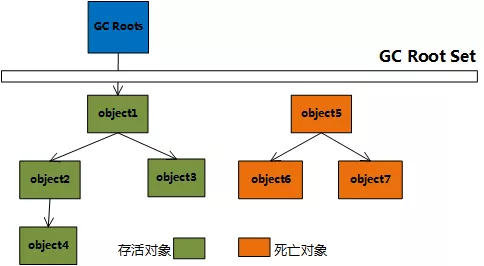
### 提醒

上面的“运行时栈”和“索引”是为方便引入而故意发明的词，实际上在JVM中，它们的名字分别叫做“局部变量表”和“Slot”。而且局部变量表在编译时即已确定，不需要等到“运行时”。

## GC一瞥

这里来简单讲讲主流GC里非常简单的一小块：**如何确定对象可以被回收。**另一种表达是，**如何确定对象是存活的。**

仔细想想，Java的世界中，对象与对象之间是存在关联的，我们可以从一个对象访问到另一个对象。如图所示。



再仔细想想，这些对象与对象之间构成的引用关系，就像是一张大大的图；更清楚一点，是众多的树。

如果我们找到了所有的树根，那么从树根走下去就能找到所有存活的对象，那么那些没有找到的对象，就是已经死亡的了！这样GC就可以把那些对象回收掉了。

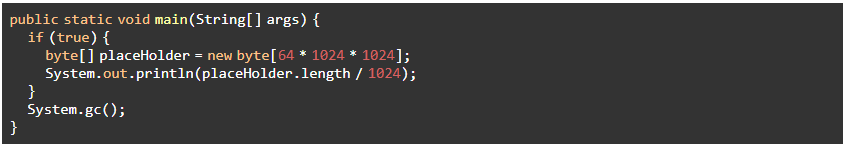
现在的问题是，怎么找到树根呢？JVM早有规定，其中一个就是：**栈中引用的对象。也就是说，只要堆中的这个对象，在栈中还存在引用，就会被认定是存活的。**

### 提醒

上面介绍的确定对象可以被回收的算法，其名字是“可达性分析算法”。

## JVM的“bug”

我们再来回头看看最开始的例子：



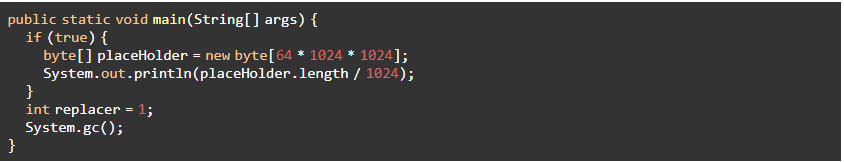
看看其运行时栈：



栈中第一个索引是方法传入参数args，其类型为String[]；第二个索引是placeHolder，其类型为byte[]。

联系前面的内容，我们推断placeHolder没有被回收的原因：System.gc();**触发GC时，main()方法的运行时栈中，还存在有对args和placeHolder的引用，GC判断这两个对象都是存活的，不进行回收。**也就是说，代码在离开if后，虽然已经离开了placeHolder的作用域，但在此之后，没有任何对运行时栈的读写，placeHolder所在的索引还没有被其他变量重用，所以GC判断其为存活。

为了验证这一推断，我们在System.gc();之前再声明一个变量，按照之前提到的“Java的栈优化”，这个变量会重用placeHolder的索引。



看看其运行时栈：



不出所料，replacer重用了placeHolder的索引。来看看GC情况：



placeHolder被成功回收了！我们的推断也被验证了。

再从运行时栈来看，加上int replacer = 1;和将placeHolder赋值为null起到了同样的作用：断开堆中placeHolder和栈的联系，让GC判断placeHolder已经死亡。

现在算是理清了“不使用的对象应手动赋值为null“的原理了，一切根源都是来自于JVM的一个“bug”：代码离开变量作用域时，并不会自动切断其与堆的联系。为什么这个“bug”一直存在？你不觉得出现这种情况的概率太小了么？算是一个tradeoff了。

## 总结

希望看到这里你已经明白了“不使用的对象应手动赋值为null“这句话背后的奥义。我比较赞同《深入理解Java虚拟机》作者的观点：**在需要“不使用的对象应手动赋值为null“时大胆去用，但不应当对其有过多依赖，更不能当作是一个普遍规则来推广。**