【经典案例】JVM内存分配担保机制

<https://cloud.tencent.com/developer/article/1082730>

在现实社会中，借款会指定担保人，就是当借款人还不起钱，就由担保人来还钱。

在JVM的内存分配时，也有这样的**内存分配担保机制**。就是当在新生代无法分配内存的时候，把新生代的对象转移到老生代，然后把新对象放入腾空的新生代。

现在假设，我们的新生代分为三个区域，分别为eden space，from space和to space。

现在是尝试分配三个2MB的对象和一个4MB的对象，然后我们通过JVM参数 -Xms20M、-Xmx20M、-Xmn10M 把Java堆大小设置为20MB，不可扩展。

其中10M分配给新生代，另外10M分配给老生代。

然后我们通过-XX：SurvivorRatio=8来分配新生代各区的比例，设置为8，表示eden与一个survivor区的空间比例为8:1。



**图1 新生代内存分配**

JVM参数配置：

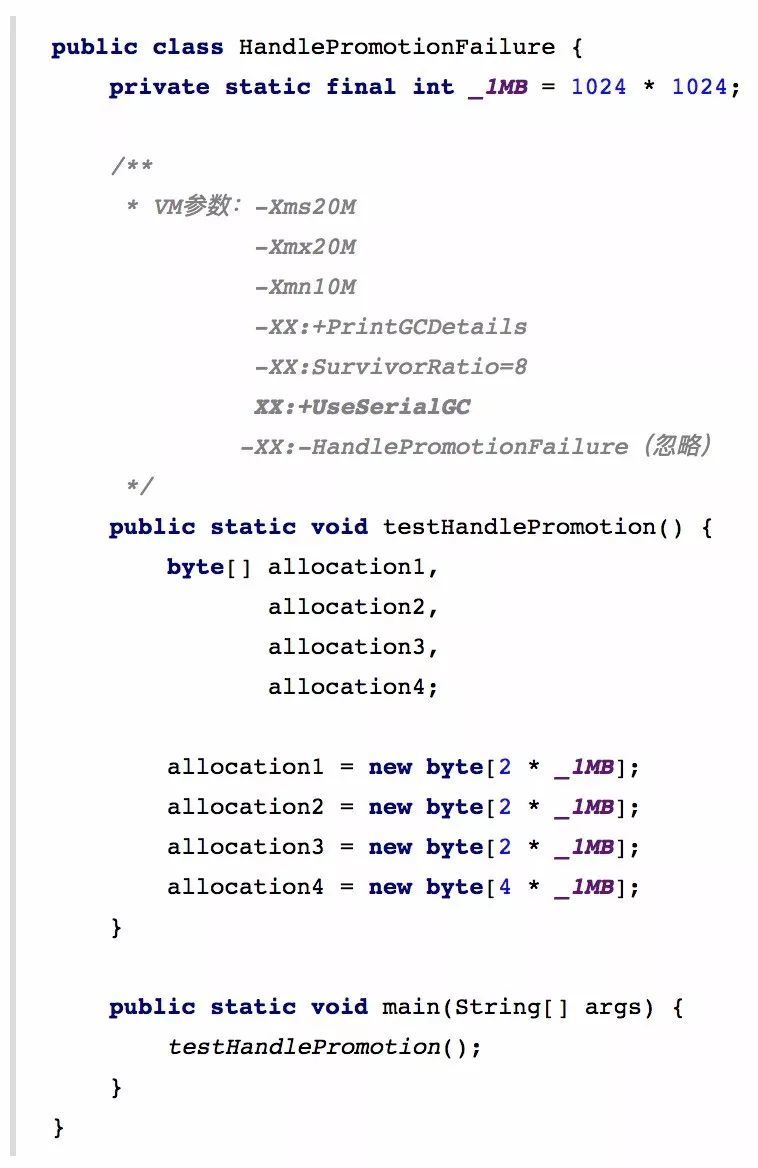
-Xms20M -Xmx20M -Xmn10M -XX:+PrintGCDetails -XX:SurvivorRatio=8 -**XX:+UseSerialGC**

这里我们先手动指定垃圾收集器为客户端模式下的Serial+Serial Old的收集器组合进行内存回收。

由于不同的收集器的收集机制不同，为了呈现出内存分配的担保效果，我们这里需要手动指定为Serial+Serial Old模式。

另外担保机制在JDK1.5以及之前版本中默认是关闭的，需要通过HandlePromotionFailure手动指定，JDK1.6之后就默认开启。这里我们使用的是JDK1.8，所以不用再手动去开启担保机制。

下面我们新建四个byte数组，前三个分别为2MB大小的内存分配，第四个是4MB的内存分配。代码如下：



然后运行程序，看看GC日志：

[GC (Allocation Failure) [DefNew: **7836K->472K(9216K)**, 0.0120087 secs] **7836K->6616K(19456K)**, 0.0123203 secs] [Times: user=0.01 sys=0.01, real=0.01 secs]

Heap

def new generation total **9216K**, used 4732K [0x00000007bec00000, 0x00000007bf600000, 0x00000007bf600000)

**eden space 8192K**, 52% used [0x00000007bec00000, 0x00000007bf0290f0, 0x00000007bf400000)

**from space 1024K**, 46% used [0x00000007bf500000, 0x00000007bf576018, 0x00000007bf600000)

**to space 1024K**, 0% used [0x00000007bf400000, 0x00000007bf400000, 0x00000007bf500000)

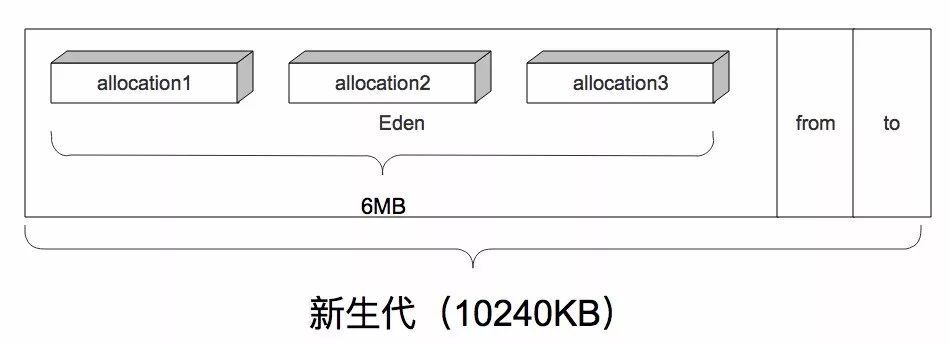
tenured generation total 10240K, used 6144K [0x00000007bf600000, 0x00000007c0000000, 0x00000007c0000000)

**the space 10240K**, **60%** used [0x00000007bf600000, 0x00000007bfc00030, 0x00000007bfc00200, 0x00000007c0000000)

Metaspace used 3160K, capacity 4494K, committed 4864K, reserved 1056768K

class space used 341K, capacity 386K, committed 512K, reserved 1048576K

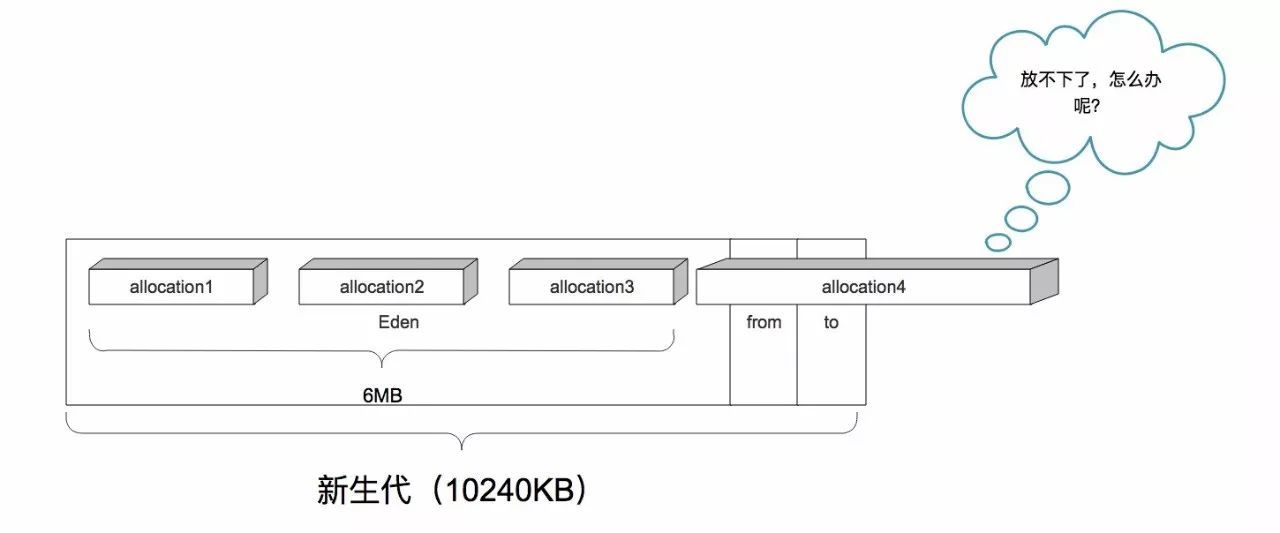
通过GC日志我们发现在分配allocation4的时候，发生了一次Minor GC，让**新生代从7836K变为了472K**，但是你发现整个堆的占用并没有多少变化。这是因为前面三个2MB的对象都还存活着，所以回收器并没有找到可回收的对象。但为什么会出现这次GC呢？



**图2 正常流程把前三个对象放入了新生代Eden区**

如果你算一笔账就知道了，前面三个对象**2MB+2MB+2MB=6MB**。

虚拟机分配内存优先会分配到新生代的eden space，通过**图1**我们知道新生代可用内存一共只有**9216KB**，现在新生代已经被用去了**6MB**，还剩下**9216KB-6144KB=3072KB**，然而第四个对象是**4MB**，显然在新生代已经装不下了。



**图3 第四个对象此时无法放入Eden区**

**于是发生了一次Minor GC！**

而且本次GC期间，虚拟机发现eden space的三个对象（6MB）又无法全部放入Survivor空间(Survivor可用内存只有1MB)。

这时候该怎么办呢？第四个对象还要不要分配呢？

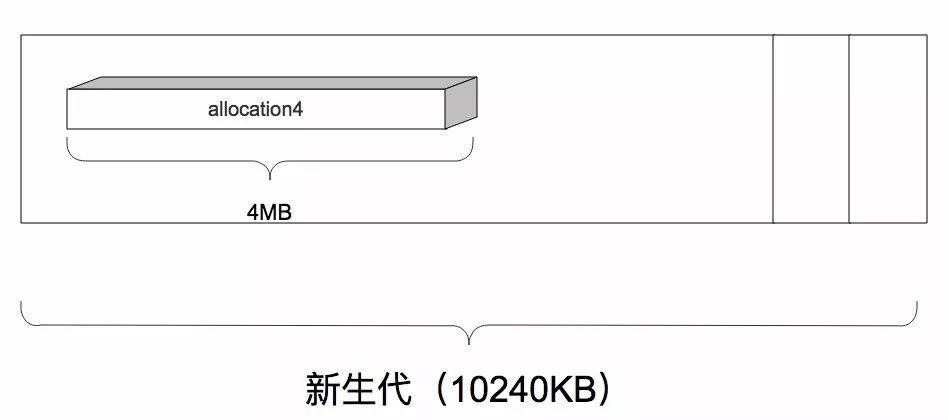
此时，JVM就启动了内存分配的担保机制，把这6MB的三个对象直接转移到了老年代。

此时就把新生代的空间腾出来了，然后把第四个对象（4MB）放入了Eden区中，所以你看到的结果是4096/8192=0.5，也就是约50%：

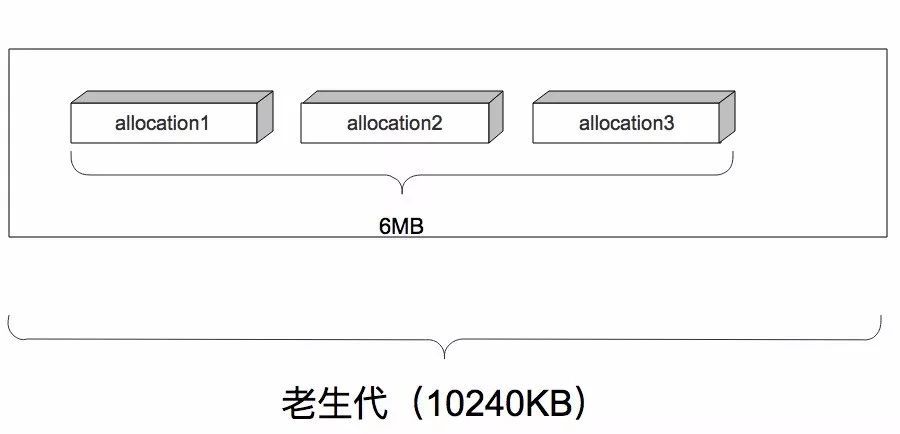
eden space 8192K, 52% used [0x00000007bec00000, 0x00000007bf0290f0, 0x00000007bf400000)

老年代则被占用了6MB，也就是前三个对象，1024\*2\*3=6144KB，6144KB/10240KB=0.6也就是60%：

the space 10240K, 60% used [0x00000007bf600000, 0x00000007bfc00030, 0x00000007bfc00200, 0x00000007c0000000)



**图4：担保后，allocation4放入到新生代eden区**



**图5：担保后，之前在新生代的三个对象转移到了老生代**

**服务端模式下的担保机制实现**

上面我们演示的在客户端模式（Serial+Serial Old）的场景下的结果，接下来我们使用服务端模式（**Parallel Scavenge+Serial Old的组合**）来看看担保机制的实现。

修改GC组合为：-XX:+UseParallelGC

然后我们运行程序看看GC日志。

* **第四个对象是4MB的情况下：**

[GC (Allocation Failure) [PSYoungGen: **6156K->592K(9216K)**] **6156K->4696K(19456K)**, 0.0032059 secs] [Times: user=0.01 sys=0.01, real=0.01 secs]

Heap

**PSYoungGen** total **9216K**, used 7057K [0x00000007bf600000, 0x00000007c0000000, 0x00000007c0000000)

**eden space 8192K**, 78% used [0x00000007bf600000,0x00000007bfc505f8,0x00000007bfe00000)

**from space 1024K**, 57% used [0x00000007bfe00000,0x00000007bfe94010,0x00000007bff00000)

**to space 1024K**, 0% used [0x00000007bff00000,0x00000007bff00000,0x00000007c0000000)

**ParOldGen** total **10240K**, **used 4104K** [0x00000007bec00000, 0x00000007bf600000, 0x00000007bf600000)

object space 10240K, **40%** used [0x00000007bec00000,0x00000007bf002020,0x00000007bf600000)

Metaspace used 3299K, capacity 4494K, committed 4864K, reserved 1056768K

class space used 357K, capacity 386K, committed 512K, reserved 1048576K

* **第四个对象是3MB的情况下：**

[GC (Allocation Failure) [PSYoungGen: **8192K->544K**(9216K)] 8192K->6688K(19456K), 0.0052943 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]

[Full GC (Ergonomics) [PSYoungGen: 544K->0K(9216K)] [ParOldGen: 6144K->6627K(10240K)] 6688K->6627K(19456K), [Metaspace: 3286K->3286K(1056768K)], 0.0063048 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]

Heap

**PSYoungGen** total 9216K, used 3238K [0x00000007bf600000, 0x00000007c0000000, 0x00000007c0000000)

**eden space 8192K, 39%** used [0x00000007bf600000,0x00000007bf929918,0x00000007bfe00000)

**from space 1024K, 0%** used [0x00000007bfe00000,0x00000007bfe00000,0x00000007bff00000)

**to space 1024K, 0%** used [0x00000007bff00000,0x00000007bff00000,0x00000007c0000000)

**ParOldGen total 10240K, used 6627K** [0x00000007bec00000, 0x00000007bf600000, 0x00000007bf600000)

object space 10240K, **64%** used [0x00000007bec00000,0x00000007bf278f70,0x00000007bf600000)

Metaspace used 3294K, capacity 4494K, committed 4864K, reserved 1056768K

class space used 356K, capacity 386K, committed 512K, reserved 1048576K

发现**当我们使用Server模式下的ParallelGC收集器组合（Parallel Scavenge+Serial Old的组合）下，担保机制的实现和之前的Client模式下（SerialGC收集器组合）有所变化。在GC前还会进行一次判断，如果要分配的内存>=Eden区大小的一半，那么会直接把要分配的内存放入老年代中。否则才会进入担保机制。**

这里我们的第四个对象是4MB的时候，也就是(1024KB\*4)/8192KB=0.5，刚好一半，于是就这第四个对象分配到了老年代。

第二次，我们把第四个对象由4MB，改为3MB，此时3MB/8192KB=0.37，显然不到一半，此时发现3MB还是无法放入，那么就执行担保机制，把前三个对象转移到老生代，然后把第四个对象（3MB）放入eden区。

**总结**

内存分配是在JVM在内存分配的时候，新生代内存不足时，把新生代的存活的对象搬到老生代，然后新生代腾出来的空间用于为分配给最新的对象。这里老生代是担保人。在不同的GC机制下，也就是不同垃圾回收器组合下，担保机制也略有不同。在Serial+Serial Old的情况下，发现放不下就直接启动担保机制；在Parallel Scavenge+Serial Old的情况下，却是先要去判断一下要分配的内存是不是**>=Eden区大小的一半**，如果是那么直接把该对象放入老生代，否则才会启动担保机制。