**内存溢出**

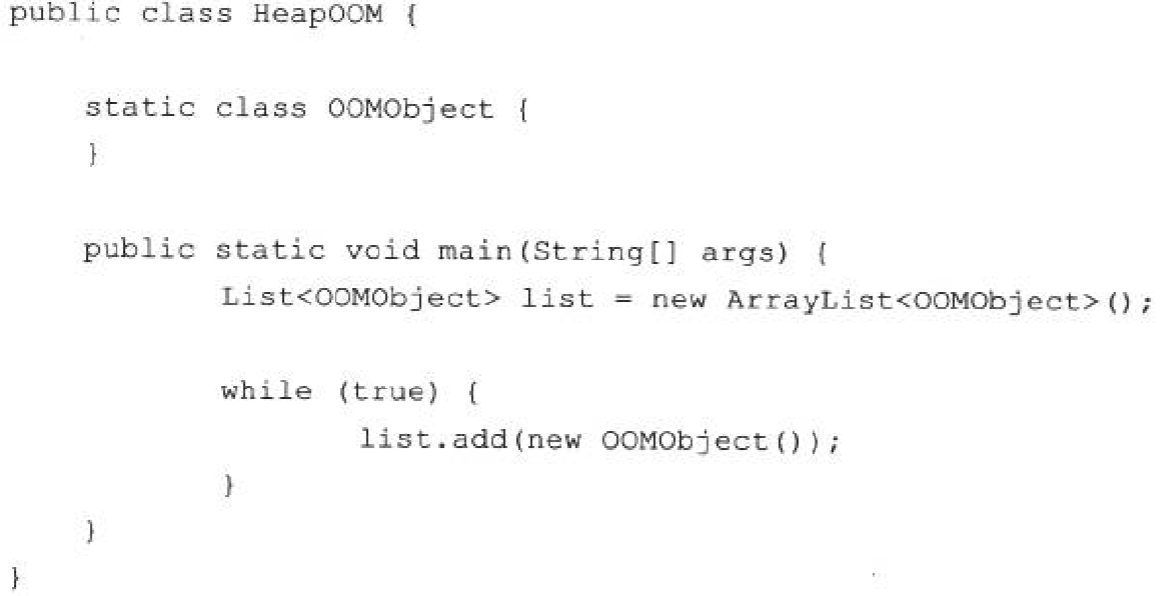
在Java虚拟内存分析中，除了程序计数器不会出现内存溢出外，其它的几个内存都会发生内存溢出异常OutOfMemoryError（简称OOM）。下面就分析各个内存区域中的OOM异常。

* **Java堆溢出**

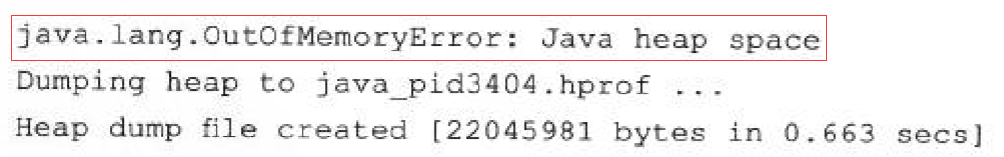
Java堆用于存储Java对象，只要在Java堆上不停地创建对象，并且保证GC Root到对象之间有可达路径（避免垃圾收集器回收这些对象），当对象到达最大的容量限制（-Xmx）后就会产生堆内存溢出异常。

**参数**： -Xms：代表堆的最小值。-Xmx：代表堆的最大值。只要将这两个值设置为一样就可以避免虚拟机自动扩展堆内存。

**示例：**



上面运行结果会出现Java堆内存溢出异常，当出现堆内存溢出异常时异常堆栈打印的信息“java.lang.OutOfMemoryError”会跟着进一步提示“Java heap space”，如下：



**解决：**解决对内存溢出异常需要使用内存映像分析工具进行分析。重点在于分析出是出现了内存泄露还是内存溢出。

如果是内存泄露就通过工具查看GC Roots的引用链。于是就能找到泄露的对象是通过什么路径与GC Roots相关联而导致无法被垃圾回收器回收。

如果是内存溢出，第一、需要检查虚拟机堆参数（-Xms和-Xmx）相对于机器物理内存是否还可以调大。第二、检查内存中是否有生命周期的对象，并尝试减少运行期间的内存消耗。

* **虚拟机栈和本地方法栈溢出**

虚拟机规范中给出了两个异常：

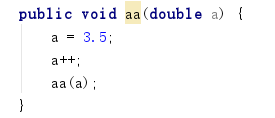
1. 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出StackOverflowError异常。
2. 如果虚拟机在扩展栈是无法申请到足够的内存空间，将抛出OutOfMermoryErrory异常。

对《深入Java虚拟机》一书中所做的解释感觉有点不理解，所以个人认为虚拟机给出的上面两个规范可以这么理解：

针对第一条，假如虚拟机的内存空间足够，当线程的栈的栈帧所占的内存一旦大于线程栈的最大限制（这里的最大限制是说线程栈已经扩展到最大），此时虚拟机还有空余的空间，则会抛出StackOverflowError异常。这里是栈溢出。

针对第二条，假如虚拟机的内存空间全部使用，没有空闲的空间时，这时候当某个线程的线程栈需要扩展空间，向虚拟机申请内存空间时，申请不到任何内存空间，则抛出OutOfMermoryErrory异常。此时并不是栈溢出，而是内存溢出。

示例，执行如下方法，这是一个无限递归：



**多线程内存溢出浅析**

通过不断的创建线程，也会引起OutOfMermoryErrory异常，这里并不是单纯的Java堆OOM。

操作系统分配给每个进程的空间是有限制的，譬如32位的Windows限制为2GB。虚拟机提供了参数来控制Java堆和方法区最大值。所以，当不计算进程本身耗费的内存，2GB减去Java堆最大值，再减去方法区最大容量，剩下的就是线程可以使用的内存空间，而线程计数器所占的内存很小可以忽略不计，所以剩下的内存就全部被线程的本地方法栈和虚拟机栈所用。当无限制的创建线程时，也就是无限制的创建本地方法栈和虚拟机栈，导致虚拟机内存越来越小，最终会导致内存溢出。

通过上面分析，要想在虚拟机中创建更多的线程，再不能减少线程数和更换64位虚拟机的情况下，只能通过减少最大堆和减少栈容量来创建更多线程。

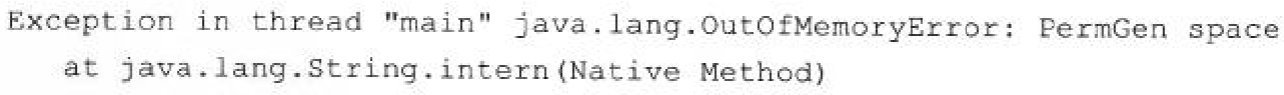
* **方法区和运行时常量池溢出**

方法区用于存放Class的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。对于这些区域的测试，基本的思路就是运行时产生大量的类去填充方法区（如反射和动态代理），直到溢出。

在JDK1.6之前的版本中，由于常量池分配在永久代中，我们可以通过-XX:PermSize和-MaxPermSize限制方法区的大小，从而间接限制运行时常量池的大小。

**

上面代码中String.intern()是一个Native方法，它的作用是：如果一个字符常量池中已经包含一个等于此字符对象的字符串，则返回代表池中这个字符串的对象；否则，将此字符串对象中的字符添加到字符串常量池中，并返回此String对象的引用。所以上面代码运行结果如下：



从结果中看出，运行时常量溢出，在OutOfMemoryError后面会跟随“PermGen space”。