# **[内存溢出的解决思路](https://www.cnblogs.com/200911/p/3965108.html)**

**内存溢出**是指应用系统中存在无法回收的内存或使用的内存过多，最终使得程序运行要用到的内存大于虚拟机能提供的最大内存。

   引起内存溢出的**原因**有很多种，常见的有以下几种：  
　　1.内存中加载的数据量过于庞大，如一次从数据库取出过多数据；  
　　2.集合类中有对对象的引用，使用完后未清空，使得JVM不能回收；  
　　3.代码中存在死循环或循环产生过多重复的对象实体；  
　　4.使用的第三方软件中的BUG；  
　　5.启动参数内存值设定的过小；

内存溢出的**解决方案**：  
**第一步**，修改JVM启动参数，直接增加内存。(-Xms，-Xmx参数一定不要忘记加。)

**第二步**，检查错误日志，查看“OutOfMemory”错误前是否有其它异常或错误。

**第三步**，对代码进行走查和分析，找出可能发生内存溢出的位置。

重点排查以下几点：  
　　1.检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。这个问题比较隐蔽，在上线前，数据库中数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。因此对于数据库查询尽量采用分页的方式查询。  
　　2.检查代码中是否有死循环或递归调用。

　　3.检查是否有大循环重复产生新对象实体。

　　4.检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。这个问题比较隐蔽，在上线前，数据库中   数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。因此对于数据库查询尽量采用分页的方式查询。

　　5.检查List、MAP等集合对象是否有使用完后，未清除的问题。List、MAP等集合对象会始终存有对对象的引用，使得这些对象不能被GC回收。

**第四步**，使用内存查看工具动态查看内存使用情况

**从内存溢出看Java 环境中的内存结构**

让我们看一下我们日常在开发过程中接触内存溢出的异常：

Exception in thread "main" [Full GCjava.lang.OutOfMemoryError: Java heap space

Exception in thread "main" java.lang.StackOverflowError

java.lang.OutOfMemoryError: PermGen space

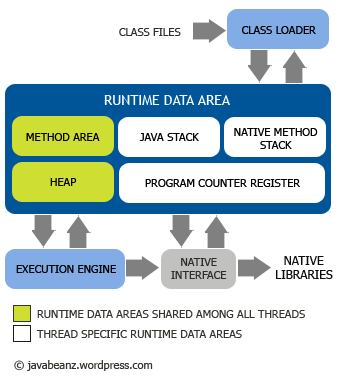
是不是有大家很熟悉的，遇见这样的问题解决起来可能不简单，但是如果现在让大家写个程序，故意让程序出现下面的异常，估计能很快写出来的也不是很多，这就要求开发人员对于java内存区域以及jvm规范有比较深的了解。

　　既然抛出了异常，首先我们肯定这些都是内存异常，只是内存异常中的不同种类，我们就试着了解一下为什么会出现以上的异常，可以看出有两种异常状况：：

　　OutOfMemoryError

　　StackOverflowError

　　其中OutOfMemoryError是在程序无法申请到足够的内存的时候抛出的异常，StackOverflowError是线程申请的栈深度大于虚拟机所允许的深度所抛出的异常。 可是从上面列出的异常内容也可以看出在OutOfMemoryError类型的一场中也存在这很多异常的可能。这是为什么？以为是在内存的不同结构中出现的错误，所以抛出的异常也就形形色色，说道这我们不得不介绍一下java的内存结构，请看下图（从网上摘的）：



　　在运行时的内存区域有5个部分，Method Area（方法区）,Java stack（java 虚拟机栈）,Native MethodStack（本地方法栈），Heap(堆)，Program Counter Regster(程序计数器)。从图中看出方法区和堆用黄色标记，和其他三个区域的不同点就是，方法区和堆是线程共享的，所有的运行在jvm上的程序都能访问这两个区域，堆，方法区和虚拟机的生命周期一样，随着虚拟机的启动而存在，而栈和程序计数器是依赖用户线程的启动和结束而建立和销毁。

**Program Counter Regster（程序计数器）**：每一个用户线程对应一个程序计数器，用来指示当前线程所执行字节码的行号。由程序计数器给文字码解释器提供吓一条要执行的字节码的的位置。根据jvm规范，在这个区域中不会抛出OutOfMemoryError的内存异常。

**Java stack（java 虚拟机栈）：**这个区域是最容易出现内存异常的区域，每一个线程对应生成一个线程栈，线程每执行一个方法的时候，都会创建一个栈帧，用来存放方法的局部变量表，操作树栈，动态连接，方法入口，这和C#是不一样的，在C#CLR中没有栈帧的概念，都是在线程栈中通过压栈和出栈的方式进行数据的保存。jvm规范对这个区域定义了两种内存异常，OutOfMemoryError，StackOverflowError。

**Native MethodStack（本地方法栈）**：和虚拟机栈一样，不同的是处理的对象不一样，虚拟机栈处理java的字节码，而本地栈则是处理的Native方法。其他方面一致。

**Heap(堆)**：前面说了堆是所有线程都能访问的，随着虚拟机的启动而存在，这块区域很大，因为所有的线程都在这个区域保存实例化的对象，因为每一个类型中，每个接口实现类需要的内存不一样，一个方法内的多个分支需要的内存也不尽相同，我们只有在运行的时候才能知道要创建多少对象，需要分配多大的地址空间。GC关注的正是这样的部分内容，所以很多时候也将堆称为GC堆。堆中肯定不会抛出StackOverflowError类型的异常，所以只有OutOfMemoryError相关类型的异常。

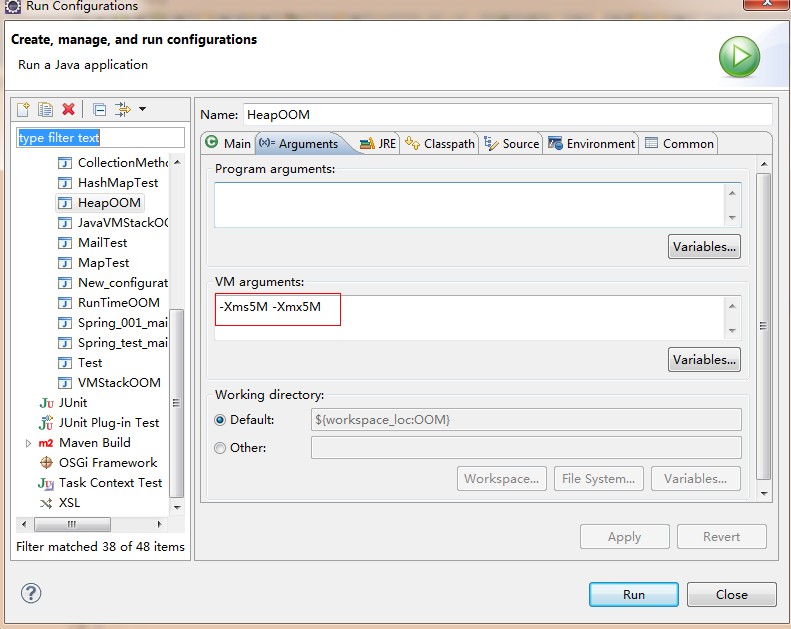
**Method Area（方法区）**：用于存放已被虚拟机加载的类信息，常量，静态方法，即使编译后的代码。同样只能抛出OutOfMemoryError相关类型的异常。

　　介绍完jvm内存结构中的常见区域，下面该是和我们主题呼应的时候了，在什么情况下，在那个区域，如何才能复现开始提到的异常信息？从第一个开始，异常信息的内容为：

Exception in thread "main" [Full GCjava.lang.OutOfMemoryError: Java heap space

可想而知是在堆中出现的问题，如何重现，由于是在堆中出现这个异常，那么就要处理好，不能被垃圾回收器给回收了，设置一下jvm中堆的最大值（这样才能够更快的出现错误），设置jvm值的方法是通过**-Xms(堆的最小值)，-Xmx(堆的最大值)。**下面动手试一下：

**然后在运行的时候设置jvm参数，如下：**



成功在java虚拟机堆中溢出。下面看第二个关于栈的异常，内容如下：

Exception in thread "main" java.lang.StackOverflowError

因为是与栈相关的话，那么我们在重现异常的时候就要相应的将栈内存容量设置的小一些，设置栈大小的方法是设置**-Xss参数**，看如下实现：

　运行时jvm参数的设置如下：

 　第三个异常是关于perm的异常内容，我们需要的是设置方法区的大小，实现方式是通过设置-XX:PermSize和-XX:MaxPermSize参数，内容如下：

java.lang.OutOfMemoryError: PermGen space

　　如果程序加载的类过多，例如tomcatweb容器，就会出现PermGen space异常，如果我将HeapOOM类的运行时的XX:PermSize设置为2M，如下：

　　第四个异常估计遇到的人就不多了，是DirectMemory内存相关的，内容如下：

Exception in thread "main" java.lang.OutOfMemoryError

at sun.misc.Unsafe.allocateMemory(Native Method)

at oom.DirectMemoryOOM.main(DirectMemoryOOM.java:23)

DirectMemoruSize可以通过设置 -XX:MaxDirectMemorySize参数指定容量大小，如果不指定的话，那么就跟堆的最大值一致,下面是代码实现：

运行时设置的jvm参数如下：

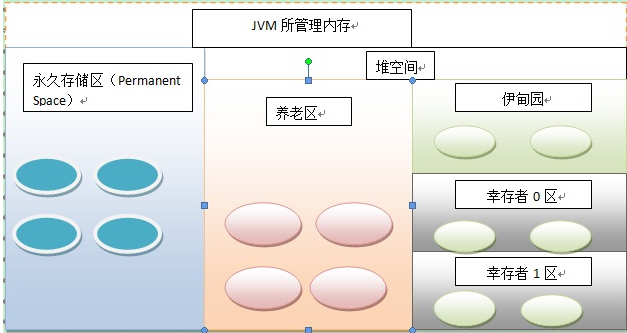
**关于JAVA中内存溢出的解决办法**

J2ee应用系统是运行在J2EE应用服务器上的，而j2ee应用服务器又是运行在JVM上的，

生成环境中JVM参数的优化和设置对于J2EE应用系统性能有着决定性的作用。要优化系统，则需要对JVM参数进行合理的设置，所以我们需要了解究竟在什么地方进行设置、有哪些参数以及各参数的意义分别是什么，并且我们还得了解JVM的内存管理机制究竟是个什么玩意儿？其实我们在网上搜索引擎上，一搜就有可以获取到一大把相关信息，关键是我们如何深入的理解它们。那么下面我们就简单的介绍一下究竟什么是JVM的内存管理机制吧~！

JVM的早期版本并没有进行分区管理；这样的后果是JVM进行垃圾回收时，不得不扫描JVM所管理的整片内存，所以搜集垃圾是很耗费资源的事情，也是早起JAVA程序的性能低下的主要原因。随着JVM的发展，JVM引进了分区管理的机制。

JVM所管理的所有内存资源分为2个大的部分。永久存储区（Permanent Space） 和堆空间（The Heap Space）。其中对空间又分为新生区和养老区，新生区又分为伊甸园，幸存者0区、幸存1区。如下图：



关于个分区的用途，大家可以参考其他相关文档。本教程所要处理的问题是如何解决内存溢出的问题。接下来以tomcat服务器为例：

我们首先得找到内存管理所要设置的参数在哪个文件：＜CATALINA\_HOME＞/bin/catalina.bat。

需要添加一行代码：

JAVA\_OPTS="-Xms512m-Xmx512m -Xss1024K -XX:PermSize=256m -XX:MaxPermSize=256m"

下面分别对各参数进行介绍和解释：

JVM 相关参数：

参数名参数说明

-**server** 启用能够执行优化的编译器, 显著提高服务器的性能，但使用能够执行优化的编译器时，服务器的预备时间将会较长。生产环境的服务器强烈推荐设置此参数。

**-Xss** 单个线程堆栈大小值；JDK5.0 以后每个线程堆栈大小为1M，以前每个线程堆栈大小为256K。在相同物理内存下，减小这个值能生成更多的线程。但是操作系统对一个进程内的线程数还是有限制的，不能无限生成，经验值在3000~5000左右。

-**XX:**+UseParNewGC 可用来设置年轻代为并发收集【多CPU】，如果你的服务器有多个CPU，你可以开启此参数；开启此参数，多个CPU 可并发进行垃圾回收，可提高垃圾回收的速度。此参数和+UseParallelGC，-XX:ParallelGCThreads搭配使用。

+**UseParallelGC**选择垃圾收集器为并行收集器。此配置仅对年轻代有效。即上述配置下，年轻代使用并发收集，而年老代仍旧使用串行收集。可提高系统的吞吐量。

-**XX:**ParallelGCThreads 年轻代并行垃圾收集的前提下（对并发也有效果）的线程数，增加并行度，即：同时多少个线程一起进行垃圾回收。此值最好配置与处理器数目相等。永久存储区相关参数：参数名参数说明

-**Xnoclassgc**每次永久存储区满了后一般GC 算法在做扩展分配内存前都会触发一次FULL GC，除非设置了-Xnoclassgc.

**-XX:PermSize** 应用服务器启动时，永久存储区的初始内存大

-**XX:MaxPermSize** 应用运行中，永久存储区的极限值。为了不消耗扩大JVM 永久存储区分配的开销，将此参数和-XX:PermSize这个两个值设为相等。堆空间相关参数参数名参数说明

-**Xms** 启动应用时，JVM 堆空间的初始大小值。

**-Xmx** 应用运行中，JVM 堆空间的极限值。为了不消耗扩大JVM 堆控件分配的开销，将此参数和-Xms 这个两个值设为相等，考虑到需要开线程，讲此值设置为总内存的80%.

**-Xmn** 此参数硬性规定堆空间的新生代空间大小，推荐设为堆空间大小的1/4。

上面所列的JVM 参数关系到系统的性能，而其中-XX:PermSize，

-XX:MaxPermSize，-Xms，-Xmx 和-Xmn 这5 个参数更是直接关系到系统的性能，系统是否会出现内存溢出。

-XX:PermSize 和-XX:MaxPermSize 分别设置应用服务器启动时，永久存储区的初始大小和极限大小；在生成环境中强烈推荐将这个两个值设置为相同的值，以避免分配永久存储区的开销，具体的值可取系统“疲劳测试”获取到的永久存储区的极限值；如果不进行设置-XX:MaxPermSize 默认值为64M,一般来说系统的类定义文件大小都会超过这个默认值。

-Xms 和-Xmx 分别是服务器启动时，堆空间的初始大小和极限值。-Xms的默认值是物理内存的1/64 但小于1G，-Xmx 的默认值是物理内存的1/4 但小于1G.在生产环境中这些默认值是肯定不能满足我们的需要的。也就是你的服务器有8g 的内存，不对JVM 参数进行设置优化，应用服务器启动时还是按默认值来分配和约束JVM 对内存资源的使用，不会充分的利用所有的内存资源。

结论：“永久存储区溢出（java.lang.OutOfMemoryError:Java Permanent Space）”乃是永久存储区设置太小，不能满足系统需要的大小，此时只需要调整-XX:PermSize 和-XX:MaxPermSize 这两个参数即可。“JVM 堆空间溢出（java.lang.OutOfMemoryError: Java heap space）”错误是JVM 堆空间不足，此时只需要调整-Xms 和-Xmx 这两个参数即可。

到此我们知道了，当系统出现内存溢出时，是哪些参数设置不合理需要调整。但我们怎么知道服务器启动时，到底JVM 内存相关参数的值是多少呢？

这个问题其实Sun公司早已经意料到了，所以给我们开发了内存使用监控工具jvmstat.

大家可以到ORACLE官网进行下载。用它可以很方便的看到我们的服务器内存使用情况。

将下载的jvmstat包解压到如“C:\ProgramFiles\Java\jvmstat”（这是我本地java路径，大家可以根据自己所安装的java环境的路径进行解压）。启动完之后我们就可以使用visualgc命令了，cmd进入命令符窗口，输入tasklist（windows下查看进程任务PID）查找到你要检测进程PID.然后直接输入visuglgc PID 就会弹出三个可见视图。

如下图：

内存溢出与数据库锁表的问题，可以说是开发人员的噩梦，一般的程序异常，总是可以知道在什么时候或是在什么操作步骤上出现了异常，而且根据堆栈信息也很容易定位到程序中是某处出现了问题。内存溢出与锁表则不然，一般现象是操作一般时间后系统越来越慢，直到死机，但并不能明确是在什么操作上出现的，发生的时间点也没有规律，查看日志或查看数据库也不能定位出问题的代码。

更严重的是内存溢出与数据库锁表在系统开发和单元测试阶段并不容易被发现，当系统正式上线一般时间后，操作的并发量上来了，数据也积累了一些，系统就容易出现内存溢出或是锁表的现象，而此时系统又不能随意停机或重启，为修正BUG带来很大的困难。

本文以笔者开发和支持的多个项目为例，与大家分享在开发过程中遇到的Java内存溢出和数据库锁表的检测和处理解决过程。

**2．内存溢出的分析**  
内存溢出是指应用系统中存在无法回收的内存或使用的内存过多，最终使得程序运行要用到的内存大于虚拟机能提供的最大内存。为了解决Java中内存溢出问题，我们首先必须了解Java是如何管理内存的。Java的内存管理就是对象的分配和释放问题。在Java中，内存的分配是由程序完成的，而内存的释放是由垃圾收集器(Garbage Collection，GC)完成的，程序员不需要通过调用GC函数来释放内存，因为不同的JVM实现者可能使用不同的算法管理GC，有的是内存使用到达一定程度时，GC才开始工作，也有定时执行的，有的是中断式执行GC。但GC只能回收无用并且不再被其它对象引用的那些对象所占用的空间。Java的内存垃圾回收机制是从程序的主要运行对象开始检查引用链，当遍历一遍后发现没有被引用的孤立对象就作为垃圾回收。

引起内存溢出的原因有很多种，常见的有以下几种：

**l         内存中加载的数据量过于庞大，如一次从数据库取出过多数据；**

**l         集合类中有对对象的引用，使用完后未清空，使得JVM不能回收；**

**l         代码中存在死循环或循环产生过多重复的对象实体；**

**l         使用的第三方软件中的BUG；**

**l         启动参数内存值设定的过小；**

**3．内存溢出的解决**  
内存溢出虽然很棘手，但也有相应的解决办法，可以按照从易到难，一步步的解决。

**第一步，就是修改JVM启动参数，直接增加内存**。这一点看上去似乎很简单，但很容易被忽略。JVM默认可以使用的内存为64M，Tomcat默认可以使用的内存为128MB，对于稍复杂一点的系统就会不够用。在某项目中，就因为启动参数使用的默认值，经常报“OutOfMemory”错误。因此，-Xms，-Xmx参数一定不要忘记加。

**第二步，检查错误日志，查看“OutOfMemory”错误前是否有其它异常或错误。**在一个项目中，使用两个数据库连接，其中专用于发送短信的数据库连接使用DBCP连接池管理，用户为不将短信发出，有意将数据库连接用户名改错，使得日志中有许多数据库连接异常的日志，一段时间后，就出现“OutOfMemory”错误。经分析，这是由于DBCP连接池BUG引起的，数据库连接不上后，没有将连接释放，最终使得DBCP报“OutOfMemory”错误。经过修改正确数据库连接参数后，就没有再出现内存溢出的错误。

查看日志对于分析内存溢出是非常重要的，通过仔细查看日志，分析内存溢出前做过哪些操作，可以大致定位有问题的模块。

**第三步，安排有经验的编程人员对代码进行走查和分析，找出可能发生内存溢出的位置。**重点排查以下几点：

l         检查代码中是否有死循环或递归调用。

l         检查是否有大循环重复产生新对象实体。

l         检查对数据库查询中，是否有一次获得全部数据的查询。一般来说，如果一次取十万条记录到内存，就可能引起内存溢出。这个问题比较隐蔽，在上线前，数据库中数据较少，不容易出问题，上线后，数据库中数据多了，一次查询就有可能引起内存溢出。因此对于数据库查询尽量采用分页的方式查询。

l         检查List、MAP等集合对象是否有使用完后，未清除的问题。List、MAP等集合对象会始终存有对对象的引用，使得这些对象不能被GC回收。

**第四步，使用内存查看工具动态查看内存使用情况。**某个项目上线后，每次系统启动两天后，就会出现内存溢出的错误。这种情况一般是代码中出现了缓慢的内存泄漏，用上面三个步骤解决不了，这就需要使用内存查看工具了。

内存查看工具有许多，比较有名的有：Optimizeit Profiler、JProbe Profiler、JinSight和Java1.5的**Jconsole**等。它们的基本工作原理大同小异，都是监测Java程序运行时所有对象的申请、释放等动作，将内存管理的所有信息进行统计、分析、可视化。开发人员可以根据这些信息判断程序是否有内存泄漏问题。一般来说，一个正常的系统在其启动完成后其内存的占用量是基本稳定的，而不应该是无限制的增长的。持续地观察系统运行时使用的内存的大小，可以看到在内存使用监控窗口中是基本规则的锯齿形的图线，如果内存的大小持续地增长，则说明系统存在内存泄漏问题。通过间隔一段时间取一次内存快照，然后对内存快照中对象的使用与引用等信息进行比对与分析，可以找出是哪个类的对象在泄漏。

通过以上四个步骤的分析与处理，基本能处理内存溢出的问题。当然，在这些过程中也需要相当的经验与敏感度，需要在实际的开发与调试过程中不断积累。

总体上来说，产生内存溢出是由于代码写的不好造成的，因此提高代码的质量是最根本的解决办法。有的人认为先把功能实现，有BUG时再在测试阶段进行修正，这种想法是错误的。正如一件产品的质量是在生产制造的过程中决定的，而不是质量检测时决定的，软件的质量在设计与编码阶段就已经决定了，测试只是对软件质量的一个验证，因为测试不可能找出软件中所有的BUG。

--------------------------------------------------------------------------------------------------------------------------------

原因有很多种，比如：

1.数据量过于庞大；死循环 ；静态变量和静态方法过多；递归；无法确定是否被引用的对象；

2.虚拟机不回收内存（内存泄漏）；

    说白了就是程序运行要用到的内存大于虚拟机能提供的最大内存就发生内存溢出了。 内存溢出的问题要看业务和系统大小而定，对于某些系统可能内存溢出不常见，但某些系统还是很常见的解决的方法，

一个是优化程序代码，如果业务庞大，逻辑复杂，尽量减少全局变量的引用，让程序使用完变量的时候释放该引用能够让垃圾回收器回收，释放资源。  
二就是物理解决，增大物理内存，然后通过：-Xms256m -Xmx256m -XX:MaxNewSize=256m -XX:MaxPermSize=256m的修改

**一、内存溢出类型**   
1 、 java.lang.OutOfMemoryError: PermGen space

JVM 管理两种类型的内存，堆和非堆。堆是给开发人员用的上面说的就是，是在 JVM 启动时创建；非堆是留给 JVM 自己用的，用来存放类的信息的。它和堆不同，运行期内 GC 不会释放空间。如果 web app 用了大量的第三方 jar 或者应用有太多的 class 文件而恰好 MaxPermSize 设置较小，超出了也会导致这块内存的占用过多造成溢出，或者 tomcat 热部署时侯不会清理前面加载的环境，只会将 context 更改为新部署的，非堆存的内容就会越来越多。

2 、 java.lang.OutOfMemoryError: Java heap space

第一种情况是个补充，主要存在问题就是出现在这个情况中。其默认空间 ( 即 -Xms) 是物理内存的 1/64 ，最大空间 (-Xmx) 是物理内存的 1/4 。如果内存剩余不到 40 ％， JVM 就会增大堆到 Xmx 设置的值，内存剩余超过 70 ％， JVM 就会减小堆到 Xms 设置的值。所以服务器的 Xmx 和 Xms 设置一般应该设置相同避免每次 GC 后都要调整虚拟机堆的大小。假设物理内存无限大，那么 JVM 内存的最大值跟操作系统有关，一般 32 位机是 1.5g 到 3g 之间，而 64 位的就不会有限制了。

注意：如果 Xms 超过了 Xmx 值，或者堆最大值和非堆最大值的总和超过了物理内存或者操作系统的最大限制都会引起服务器启动不起来。

垃圾回收 GC 的角色

JVM 调用 GC 的频度还是很高的，主要两种情况下进行垃圾回收：

当应用程序线程空闲；另一个是 java 内存堆不足时，会不断调用 GC ，若连续回收都解决不了内存堆不足的问题时，就会报 out of memory 错误。因为这个异常根据系统运行环境决定，所以无法预期它何时出现。

根据 GC 的机制，程序的运行会引起系统运行环境的变化，增加 GC 的触发机会。

为了避免这些问题，程序的设计和编写就应避免垃圾对象的内存占用和 GC 的开销。显示调用 System.GC() 只能建议 JVM 需要在内存中对垃圾对象进行回收，但不是必须马上回收，

一个是并不能解决内存资源耗空的局面，另外也会增加 GC 的消耗。

**二、 JVM 内存区域组成**   
简单的说 java中的堆和栈

java把内存分两种：一种是栈内存，另一种是堆内存

1。在函数中定义的基本类型变量和对象的引用变量都在函数的栈内存中分配；

2。堆内存用来存放由 new创建的对象和数组

在函数（代码块）中定义一个变量时， java就在栈中为这个变量分配内存空间，当超过变量的作用域后， java会自动释放掉为该变量所分配的内存空间；在堆中分配的内存由 java虚拟机的自动垃圾回收器来管理

堆的优势是可以动态分配内存大小，生存期也不必事先告诉编译器，因为它是在运行时动态分配内存的。缺点就是要在运行时动态分配内存，存取速度较慢；

栈的优势是存取速度比堆要快，缺点是存在栈中的数据大小与生存期必须是确定的无灵活 性。

java 堆分为三个区： New 、 Old 和 Permanent

GC 有两个线程：

新创建的对象被分配到 New 区，当该区被填满时会被 GC 辅助线程移到 Old 区，当 Old 区也填满了会触发 GC 主线程遍历堆内存里的所有对象。 Old 区的大小等于 Xmx 减去 -Xmn

java栈存放

栈调整：参数有 +UseDefaultStackSize -Xss256K，表示每个线程可申请 256k的栈空间

每个线程都有他自己的 Stack

**三、 JVM如何设置虚拟内存**提示：在 JVM中如果 98％的时间是用于 GC且可用的 Heap size 不足 2％的时候将抛出此异常信息。

提示： Heap Size 最大不要超过可用物理内存的 80％，一般的要将 -Xms和 -Xmx选项设置为相同，而 -Xmn为 1/4的 -Xmx值。

提示： JVM初始分配的内存由 -Xms指定，默认是物理内存的 1/64； JVM最大分配的内存由 -Xmx指定，默认是物理内存的 1/4。

默认空余堆内存小于 40%时， JVM就会增大堆直到 -Xmx的最大限制；空余堆内存大于 70%时， JVM会减少堆直到 -Xms的最小限制。因此服务器一般设置 -Xms、 -Xmx相等以避免在每次 GC 后调整堆的大小。

提示：假设物理内存无限大的话， JVM内存的最大值跟操作系统有很大的关系。

简单的说就 32位处理器虽然可控内存空间有 4GB,但是具体的操作系统会给一个限制，

这个限制一般是 2GB-3GB（一般来说 Windows系统下为 1.5G-2G， Linux系统下为 2G-3G）， 而 64bit以上的处理器就不会有限制了

提示：注意：如果 Xms超过了 Xmx值，或者堆最大值和非堆最大值的总和超过了物理内 存或者操作系统的最大限制都会引起服务器启动不起来。

提示：设置 NewSize、 MaxNewSize相等， “new”的大小最好不要大于 “old”的一半，原因是 old区如果不够大会频繁的触发 “主 ” GC ，大大降低了性能

JVM使用 -XX:PermSize设置非堆内存初始值，默认是物理内存的 1/64；

由 XX:MaxPermSize设置最大非堆内存的大小，默认是物理内存的 1/4。

解决方法：手动设置 Heap size

修改 TOMCAT\_HOME/bin/catalina.bat

在“ echo “Using CATALINA\_BASE: $CATALINA\_BASE””上面加入以下行：

1. JAVA\_OPTS=”-server -Xms800m -Xmx800m -XX:MaxNewSize=256m”

**四、性能检查工具使用**   
定位内存泄漏：

JProfiler 工具主要用于检查和跟踪系统（限于 Java 开发的）的性能。 JProfiler 可以通过时时的监控系统的内存使用情况，随时监视垃圾回收，线程运行状况等手段，从而很好的监视 JVM 运行情况及其性能。

1. 应用服务器内存长期不合理占用，内存经常处于高位占用，很难回收到低位；

2. 应用服务器极为不稳定，几乎每两天重新启动一次，有时甚至每天重新启动一次；

3. 应用服务器经常做 Full GC(Garbage Collection)，而且时间很长，大约需要 30-40秒，应用服务器在做 Full GC的时候是不响应客户的交易请求的，非常影响系统性能。

因为开发环境和产品环境会有不同，导致该问题发生有时会在产品环境中发生， 通常可以使用工具跟踪系统的内存使用情况，在有些个别情况下或许某个时刻确实 是使用了大量内存导致 out of memory，这时应继续跟踪看接下来是否会有下降，

如果一直居高不下这肯定就因为程序的原因导致内存泄漏。

**五、不健壮代码的特征及解决办法**   
1 、尽早释放无用对象的引用。好的办法是使用临时变量的时候，让引用变量在退出活动域后，自动设置为 null ，暗示垃圾收集器来收集该对象，防止发生内存泄露。

对于仍然有指针指向的实例， jvm 就不会回收该资源 , 因为垃圾回收会将值为 null 的对象作为垃圾，提高 GC 回收机制效率；

2 、我们的程序里不可避免大量使用字符串处理，避免使用 String ，应大量使用 StringBuffer ，每一个 String 对象都得独立占用内存一块区域；

1. String str = “aaa”;
3. String str2 = “bbb”;
5. String str3 = str + str2;// 假如执行此次之后 str ,str2 以后再不被调用 , 那它就会被放在内存中等待 Java 的 gc 去回收 , 程序内过多的出现这样的情况就会报上面的那个错误 , 建议在使用字符串时能使用 StringBuffer 就不要用 String, 这样可以省不少开销；

3 、尽量少用静态变量，因为静态变量是全局的， GC 不会回收的；

4 、避免集中创建对象尤其是大对象， JVM 会突然需要大量内存，这时必然会触发 GC 优化系统内存环境；显示的声明数组空间，而且申请数量还极大。

这是一个案例想定供大家警戒:

使用jspsmartUpload作文件上传,现在运行过程中经常出现java.outofMemoryError的错误，用top命令看看进程使用情况，发现内存不足2M，花了很长时间，发现是jspsmartupload的问题。把jspsmartupload组件的源码文件（class文件）反编译成Java文件，如梦方醒：

1. m\_totalBytes = m\_request.getContentLength();
2. m\_binArray = **new byte[m\_totalBytes];**

变量m\_totalBytes表示用户上传的文件的总长度，这是一个很大的数。如果用这样大的数去声明一个byte数组，并给数组的每个元素分配内存空间，而且m\_binArray数组不能马上被释放，JVM的垃圾回收确实有问题，导致的结果就是内存溢出。

jspsmartUpload为什末要这样作，有他的原因，根据RFC1867的http上传标准，得到一个文件流，并不知道文件流的长度。设计者如果想文件的长度，只有操作servletinputstream一次才知道，因为任何流都不知道大小。只有知道文件长度了，才可以限制用户上传文件的长度。为了省去这个麻烦，jspsmartUpload设计者直接在内存中打开文件，判断长度是否符合标准，符合就写到服务器的硬盘。这样产生内存溢出，这只是我的一个猜测而已。

所以编程的时候，不要在内存中申请大的空间，因为web服务器的内存有限，并且尽可能的使用流操作，例如

1. **byte[] mFileBody = new byte[512];**
2. Blob vField= rs.getBlob("FileBody");
3. InputStream instream=vField.getBinaryStream();
4. FileOutputStream fos=**new FileOutputStream(saveFilePath+CFILENAME);**
5. **int b;**
6. **while( (b =instream.read(mFileBody)) != -1){**
7. fos.write(mFileBody,0,b);
8. }
9. fos.close();
10. instream.close();

5 、尽量运用对象池技术以提高系统性能；生命周期长的对象拥有生命周期短的对象时容易引发内存泄漏，例如大集合对象拥有大数据量的业务对象的时候，可以考虑分块进行处理，然后解决一块释放一块的策略。

6 、不要在经常调用的方法中创建对象，尤其是忌讳在循环中创建对象。可以适当的使用 hashtable ， vector 创建一组对象容器，然后从容器中去取那些对象，而不用每次 new 之后又丢弃

7 、一般都是发生在开启大型文件或跟数据库一次拿了太多的数据，造成 Out Of Memory Error 的状况，这时就大概要计算一下数据量的最大值是多少，并且设定所需最小及最大的内存空间值。