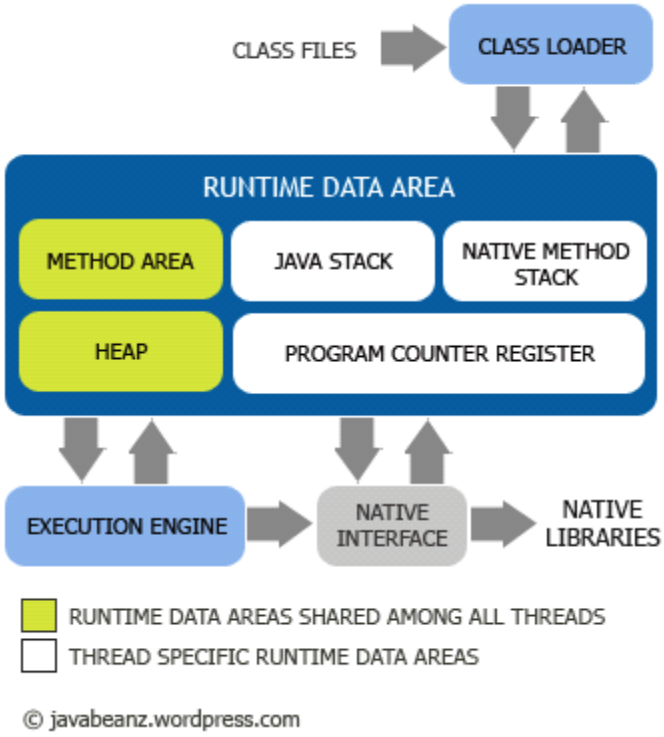
本文分为两个部分：
1，JVM specification s(JVM 规范) 对 JVM 内存的描述
2，Sun 的 JVM 的内存机制。

JVM specification 对 JVM 内存的描述

首先我们来了解 JVM specification 中的 JVM 整体架构。如下图：



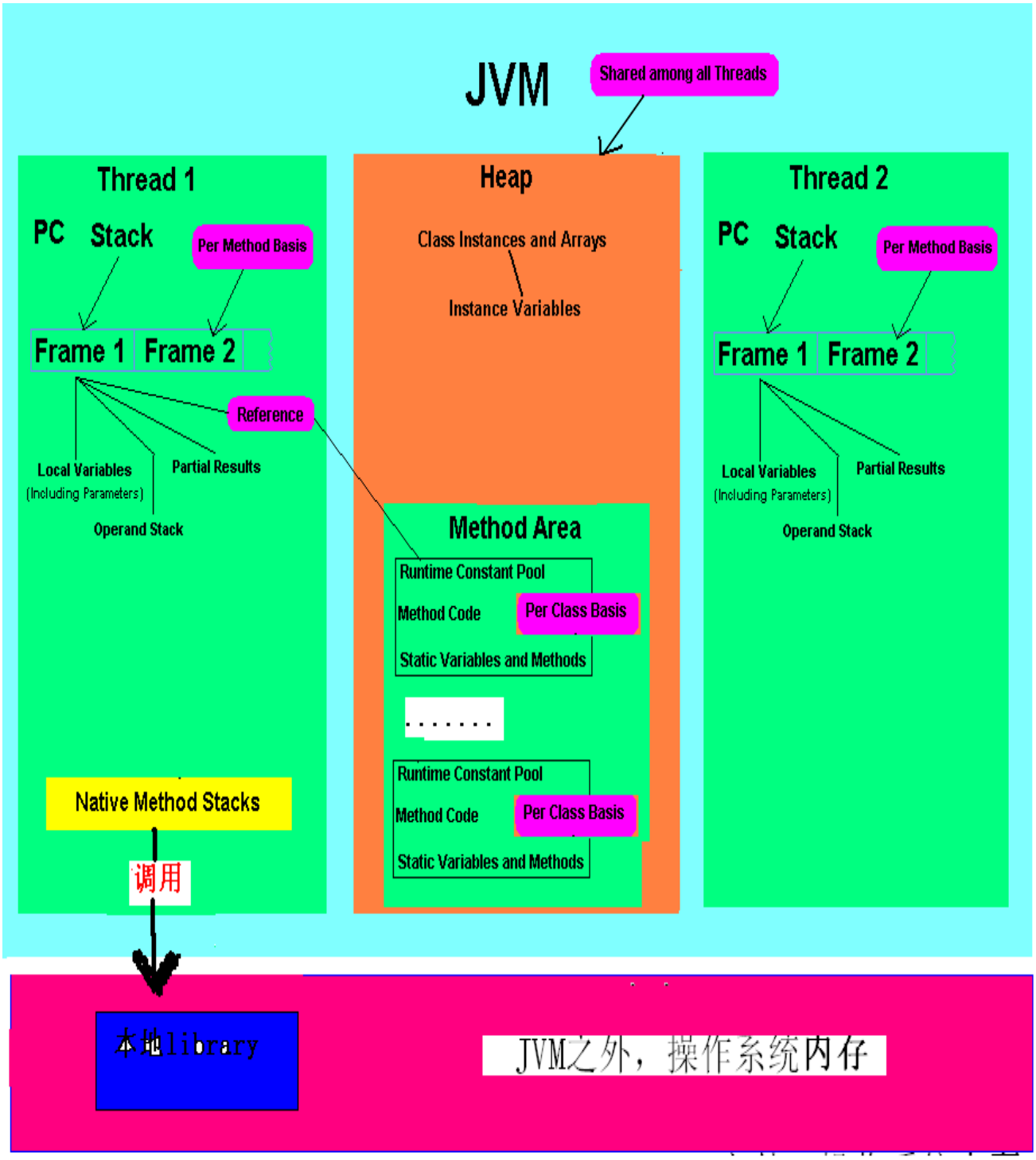
主要包括两个子系统和两个组件： Class loader(类装载器) 子系统，Execution engine(执行引擎) 子系统；Runtime data area (运行时数据区域) 组件， Native interface(本地接口)组件。

Class loader 子系统的作用 ：根据给定的全限定名类名(如 java.lang.Object)来装载 class 文件的内容到 Runtime data area 中的 method area(方法区域)。Javsa 程序员可以 extends java.lang.ClassLoader 类来写自己的 Class loader。

Execution engine 子系统的作用 ：执行 classes 中的指令。任何 JVM specification 实现(JDK)的核心是 Execution engine， 换句话说：Sun 的 JDK 和 IBM 的 JDK 好坏主要取决于他们各自实现的 Execution engine 的好坏。每个运行中的线程都有一个 Execution engine 的实例。

Native interface 组件 ：与 native libraries 交互，是其它编程语言交互的接口。

Runtime data area 组件：这个组件就是 JVM 中的内存。 下面对这个部分进行详细介绍。



Runtime data area 的整体架构图

Runtime data area 主要包括五个部分：Heap (堆)，Method Area(方法区域)，Java Stack(java 的栈)，Program Counter(程序计数器)，Native method stack(本地方法栈)。Heap 和 Method Area 是被所有线程的共享使用的；而 Java stack，Program counter 和 Native method stack 是以线程为粒度的，每个线

程独自拥有。

Heap

Java 程序在运行时创建的所有类实或数组都放在同一个堆中。而一个 Java 虚拟实例中只存在一个堆空间，因此所有线程都将共享这个堆。每一个 java 程序独占一个 JVM 实例，因而每个 java 程序都有它自己的堆空间，它们不会彼此干扰。但是同一 java 程序的多个线程都共享着同一个堆空间，就得考虑多线程访问对象(堆数据)的同步问题。（这里可能出现的异常 java.lang.OutOfMemoryError: Java heap space）

Method area

在 Java 虚拟机中，被装载的 class 的信息存储在 Method area 的内存中。当虚拟机装载某个类型时，它使用类装载器定位相应的 class 文件，然后读入这个 class 文件内容并把它传输到虚拟机中。紧接着虚拟机提取其中的类型信息，并将这些信息存储到方法区。该类型中的类（静态）变量同样也存储在方法区中。与 Heap 一样，method area 是多线程共享的，因此要考虑多线程访问的同步问题。比如，假设同时两个线程都企图访问一个名为 Lava 的类，而这个类还没有内装载入虚拟机，那么，这时应该只有一个线程去装载它，而另一个线程则只能等待。 （这里可能出现的异常 java.lang.OutOfMemoryError: PermGen full）

Java stack

Java stack 以帧为单位保存线程的运行状态。虚拟机只会直接对 Java stack 执行两种操作：以帧为单位的压栈或出栈。每当线程调用一个方法的时候，就对当前状态作为一个帧保存到 java stack 中(压栈)；当一个方法调用返回时，从 java stack 弹出一个帧(出栈)。栈的大小是有一定的限制，这个可能出现 StackOverFlow 问题。 下面的程序可以说明这个问题。

```
public class TestStackOverFlow {

        public static void main(String[] args) {

                Recursive r = new Recursive();
                r.doit(10000);
                // Exception in thread "main"
java.lang.StackOverflowError
        }

}

class Recursive {

        public int doit(int t) {
                if (t <= 1) {
```

```
                return 1;
            }
            return t + doit(t - 1);
        }

}
```

Program counter
每个运行中的 Java 程序，每一个线程都有它自己的 PC 寄存器，也是该线程启动时创建的。PC 寄存器的内容总是指向下一条将被执行指令的饿“地址”，这里的“地址”可以是一个本地指针，也可以是在方法区中相对应于该方法起始指令的偏移量。

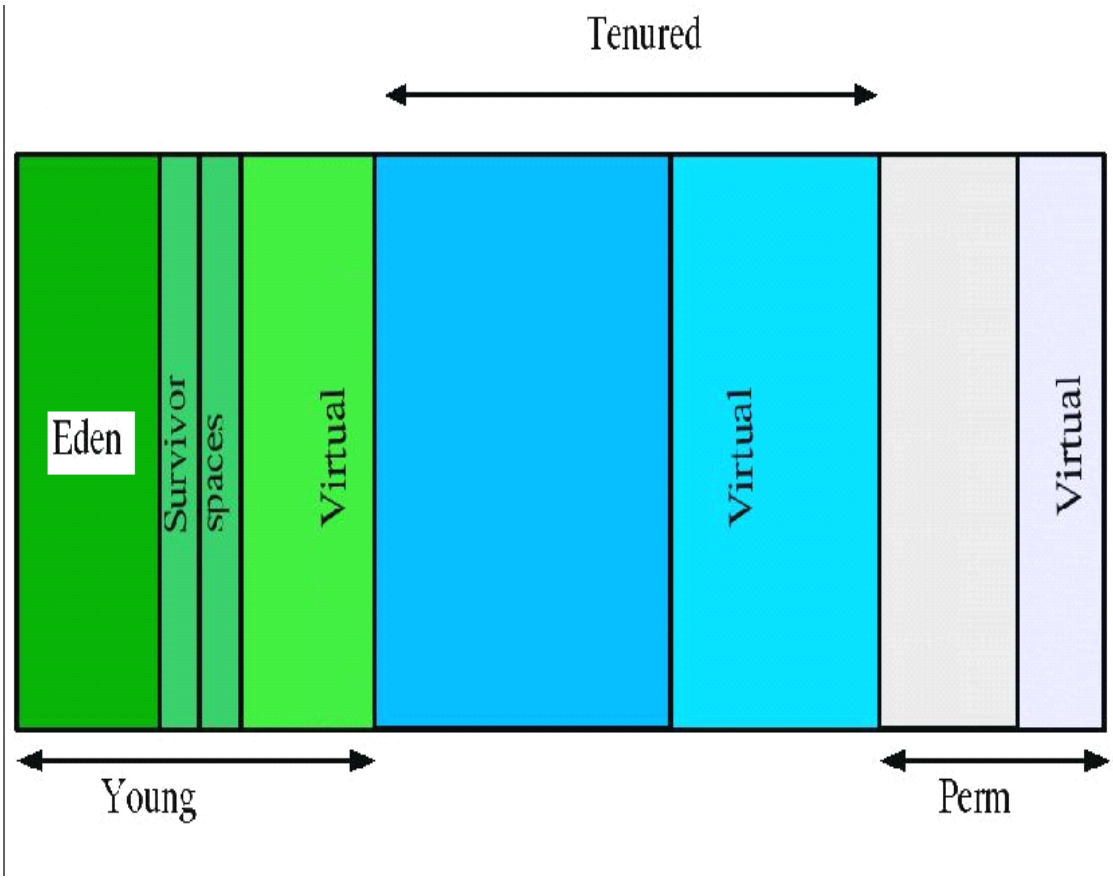
Native method stack
对于一个运行中的 Java 程序而言，它还能会用到一些跟本地方法相关的数据区。当某个线程调用一个本地方法时，它就进入了一个全新的并且不再受虚拟机限制的世界。本地方法可以通过本地方法接口来访问虚拟机的运行时数据区，不止与此，它还可以做任何它想做的事情。比如，可以调用寄存器，或在操作系统中分配内存等。总之，本地方法具有和 JVM 相同的能力和权限。（这里出现 JVM 无法控制的内存溢出问题 native heap OutOfMemory ）

Sun JVM 中对 JVM Specification 的实现（内存部分）

JVM Specification 只是抽象的说明了 JVM 实例按照子系统、内存区、数据类型以及指令这几个术语来描述的，但是规范并非是要强制规定 Java 虚拟机实现内部的体系结构，更多的是为了严格地定义这些实现的外部特征。

Sun JVM 实现中：Runtime data area(JVM 内存) 五个部分中的 Java Stack ，Program Counter, Native method stack 三部分和规范中的描述基本一致；但对 Heap 和 Method Area 进行了自己独特的实现。这个实现和 Sun JVM 的 Garbage collector（垃圾回收）机制有关，下面的章节进行详细描述。

垃圾分代回收算法（Generational Collecting）
基于对对象生命周期分析后得出的垃圾回收算法。把对象分为年青代、年老代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收。现在的垃圾回收器（从 J2SE1.2 开始）都是使用此算法的。



如上图所示，为 Java 堆中的各代分布。

1. Young（年轻代）JVM specification 中的 Heap 的一部份

年轻代分三个区。一个 Eden 区，两个 Survivor 区。大部分对象在 Eden 区中生成。当 Eden 区满时，还存活的对象将被复制到 Survivor 区（两个中的一个），当这个 Survivor 区满时，此区的存活对象将被复制到另外一个 Survivor 区，当这个 Survivor 去也满了的时候，从第一个 Survivor 区复制过来的并且此时还存活的对象，将被复制“年老区(Tenured)”。需要注意，Survivor 的两个区是对称的，没先后关系，所以同一个区中可能同时存在从 Eden 复制过来对象，和从前一个 Survivor 复制过来的对象，而复制到年老区的只有从第一个 Survivor 去过来的对象。而且，Survivor 区总有一个是空的。

2. Tenured（年老代）JVM specification 中的 Heap 的一部份

年老代存放从年轻代存活的对象。一般来说年老代存放的都是生命期较长的对象。

3. Perm（持久代） JVM specification 中的 Method area

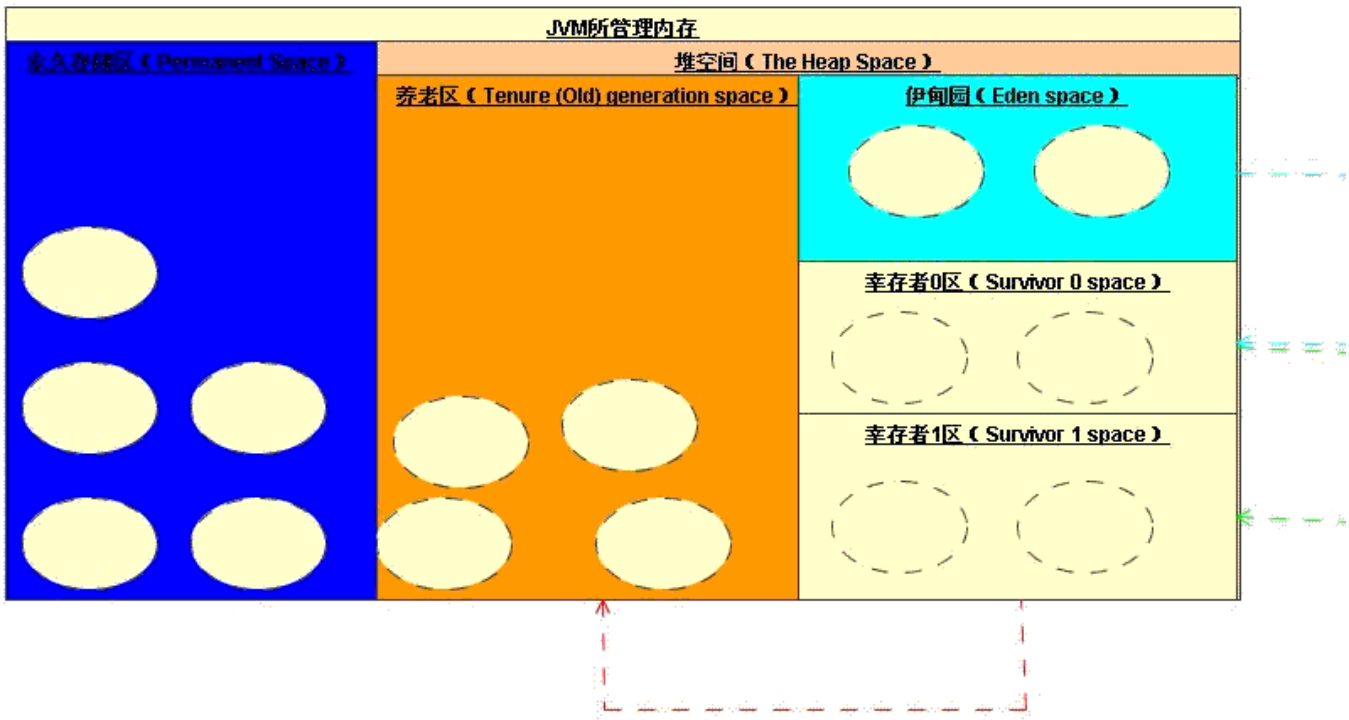
用于存放静态文件，如今 Java 类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些 class，例如 Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。持久代大小通过-XX:MaxPermSize=进行设置。

浅谈 SUN JVM 内存管理与应用服务器的优化 之 SUN JVM 内存管理

文章分类: Java 编程 关键字: j2ee
作者：Jason S.H.Chen
名词解释：
JVM（Java Virtual Machine）：Java 虚拟机，所有的 Java 程序都在 Java 虚拟机中运行。
元数据：在本文中指用于描述类和接口定义的数据。

在我做 J2EE 系统开发的工作生涯中，经常遇到技术人员或客户发出诸如此类的感慨：我的 J2EE 应用系统处理的数据量不大，系统体积也不大，技术架构也没有问题，我的应用服务器的内存有 4G 或 8G；系统运行起来很慢，还经常出现内存溢出错误。真是无奈！每次遇到这样的情况，我心中都会忍不住窃笑之。

其实他们所遇到这种情况，不是技术架构上的问题，不是系统本身的问题，也不是应用服务器的问题，也可能不是服务器的内存资源真的不足的问题。他们花了很多时间在 J2EE 应用系统本身上找问题（当然一般情况下，这种做法是对的；当出现问题时，在自身上多找找有什么不足），结果还是解决不了问题。他们却忽略了很重要的一点：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（Java Virtual Machine）上的。
其实在生产环境中 JVM 参数的优化和设置对 J2EE 应用系统性能有着决定性的作用。本篇我们就来分析 JAVA 的创建者 SUN 公司的 JVM 的内存管理机制（在现实中绝大多数的应用服务器是运行在 SUN 公司的 JVM 上的，当然除了 SUN 公司的 JVM，还有 IBM 的 JVM，Bea 的 JVM 等）；下篇咱们具体讲解怎样优化 JVM 的参数以达到优化 J2EE 应用的目的。
咱们先来看 JVM 的内存管理制吧，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，也是早期 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。
采用分区管理机制的 JVM 将 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）。其中堆空间又分为新生区（Young (New) generation space）和养老区（Tenure (Old) generation space），新生区又分为伊甸园（Eden space），幸存者 0 区（Survivor 0 space）和幸存者 1 区（Survivor 1 space）。具体分区如下图：



那 JVM 他的这些分区各有什么用途，请看下面的解说。
永久存储区（Permanent Space）：永久存储区是 JVM 的驻留内存，用于存放 JDK 自身所携带的 Class,Interface 的元数据，应用服务器允许必须的 Class,Interface 的元数据和 Java 程序运行时需要的 Class 和 Interface 的元数据。被装载进此区域的数据是不会被垃圾回收器回收掉的，关闭 JVM 时，释放此区域所控制的内存。

堆空间（The Heap Space）：是 JAVA 对象生死存亡的地区，JAVA 对象的出生，成长，死亡都在这个区域完成。堆空间又分别按 JAVA 对象的创建和年龄特征分为养老区和新生区。

新生区（Young (New) generation space）：新生区的作用包括 JAVA 对象的创建和从 JAVA 对象中筛选出能进入养老区的 JAVA 对象。

伊甸园（Eden space）：JAVA 对空间中的所有对象在此出生，该区的名字因此而得名。也即是说当你的 JAVA 程序运行时，需要创建新的对象，JVM 将在该区为你创建一个指定的对象供程序使用。创建对象的依据即是永久存储区中的元数据。

幸存者 0 区（Survivor 0 space）和幸存者 1 区（Survivor1 space）：当伊甸园的控件用完时，程序又需要创建对象；此时 JVM 的垃圾回收器将对伊甸园区进行垃圾回收，将伊甸园区中的不再被其他对象所引用的对象进行销毁工作。同时将伊甸园中的还有其他对象引用的对象移动到幸存者 0 区。幸存者 0 区就是用于

存放伊甸园垃圾回收时所幸存下来的 JAVA 对象。当将伊甸园中的还有其他对象引用的对象移动到幸存者0区时,如果幸存者0区也没有空间来存放这些对象时,JVM 的垃圾回收器将对幸存者 0 区进行垃圾回收处理，将幸存者 0 区中不在有其他对象引用的 JAVA 对象进行销毁，将幸存者 0 区中还有其他对象引用的对象移动到幸存者 1 区。幸存者 1 区的作用就是用于存放幸存者 0 区垃圾回收处理所幸存下来的 JAVA 对象。

养老区（Tenure (Old) generation space）：用于保存从新生区筛选出来的 JAVA 对象。

上面我们看了 JVM 的内存分区管理,现在我们来看 JVM 的垃圾回收工作是怎样运作的。首先当启动 J2EE 应用服务器时，JVM 随之启动，并将 JDK 的类和接口，应用服务器运行时需要的类和接口以及 J2EE 应用的类和接口定义文件也及编译后的 Class 文件或 JAR 包中的 Class 文件装载到 JVM 的永久存储区。在伊甸园中创建 JVM，应用服务器运行时必须的 JAVA 对象，创建 J2EE 应用启动时必须创建的 JAVA 对象；J2EE 应用启动完毕，可对外提供服务。

JVM 在伊甸园区根据用户的每次请求创建相应的 JAVA 对象，当伊甸园的空间不足以用来创建新 JAVA 对象的时候，JVM 的垃圾回收器执行对伊甸园区的垃圾回收工作，销毁那些不再被其他对象引用的 JAVA 对象（如果该对象仅仅被一个没有其他对象引用的对象引用的话，此对象也被归为没有存在的必要，依此类推），并将那些被其他对象所引用的 JAVA 对象移动到幸存者 0 区。

如果幸存者 0 区有足够控件存放则直接放到幸存者 0 区;如果幸存者 0 区没有足够空间存放，则 JVM 的垃圾回收器执行对幸存者 0 区的垃圾回收工作，销毁那些不再被其他对象引用的 JAVA 对象（如果该对象仅仅被一个没有其他对象引用的对象引用的话，此对象也被归为没有存在的必要，依此类推），并将那些被其他对象所引用的 JAVA 对象移动到幸存者 1 区。

如果幸存者 1 区有足够控件存放则直接放到幸存者 1 区;如果幸存者 0 区没有足够空间存放，则 JVM 的垃圾回收器执行对幸存者 0 区的垃圾回收工作，销毁那些不再被其他对象引用的 JAVA 对象（如果该对象仅仅被一个没有其他对象引用的对象引用的话，此对象也被归为没有存在的必要，依此类推），并将那些被其他对象所引用的 JAVA 对象移动到养老区。

如果养老区有足够控件存放则直接放到养老区；如果养老区没有足够空间存放，则 JVM 的垃圾回收器执行对养老区区的垃圾回收工作，销毁那些不再被其他对象引用的 JAVA 对象（如果该对象仅仅被一个没有其他对象引用的对象引用的话，此对象也被归为没有存在的必要，依此类推），并保留那些被其他对象所引用的 JAVA 对象。如果到最后养老区，幸存者 1 区，幸存者 0 区和伊甸园区都没有空间的话，则 JVM 会报告“JVM 堆空间溢出（java.lang.OutOfMemoryError: Java heap space）”，也即是在堆空间没有空间来创建对象。

这就是 JVM 的内存分区管理，相比不分区来说；一般情况下，垃圾回收的速度要快很多；因为在没有必要的时候不用扫描整片内存而节省了大量时间。

通常大家还会遇到另外一种内存溢出错误“永久存储区溢出（java.lang.OutOfMemoryError: Java Permanent Space）”。

好，本篇对 SUN 的 JVM 内存管理机制讲解就到此为止，下一篇我们将详细讲解怎样优化 SUN 的 JVM 让我们的 J2EE 系统运行更快，不出现内存溢出等问题。

浅谈 SUN JVM 内存管理与应用服务器的优化之 服务器内存分配与优化

文章分类:Java 编程

作者：Jason S.H.Chen

上篇(http://cshbbrain.javaeye.com/blog/526356)给大家介绍了 SUN JVM 的内存管理机制。本篇主要讲解与性能相关的 JVM 参数，怎样使用工具监控 JVM 的内存分配使用情况和怎样调整 JVM 参数让系统在特定硬件配置下达到最优化的性能。

通过上篇 SUN JVM 内存管理机制的介绍，大家都知道了 SUN JVM 内存分为永久存储区，伊甸园，幸存者 0 区，幸存者 1 区和养老区等几个区域。他们的作用以及垃圾回收处理过程在上篇也做了详细介绍。下面我们就来看看和这些内存分区相关的 JVM 参数。

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.

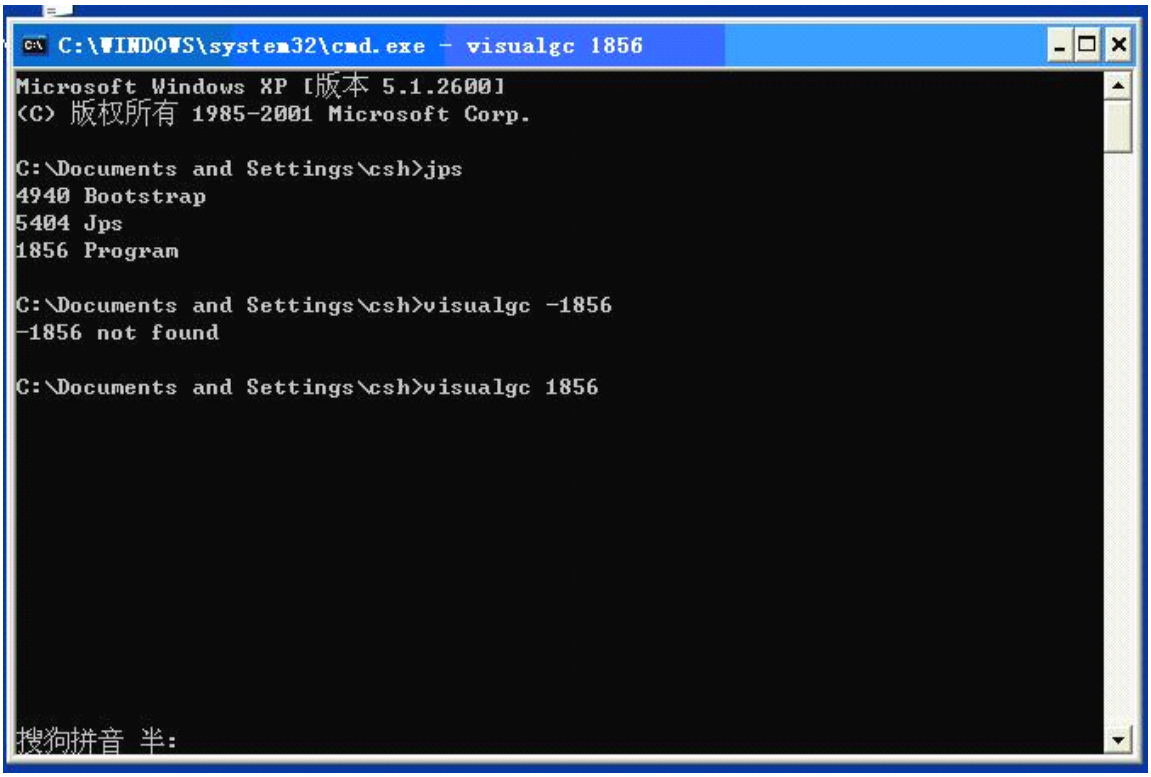
-XX:PermSize 应用服务器启动时，永久存储区的初始内存大

-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。

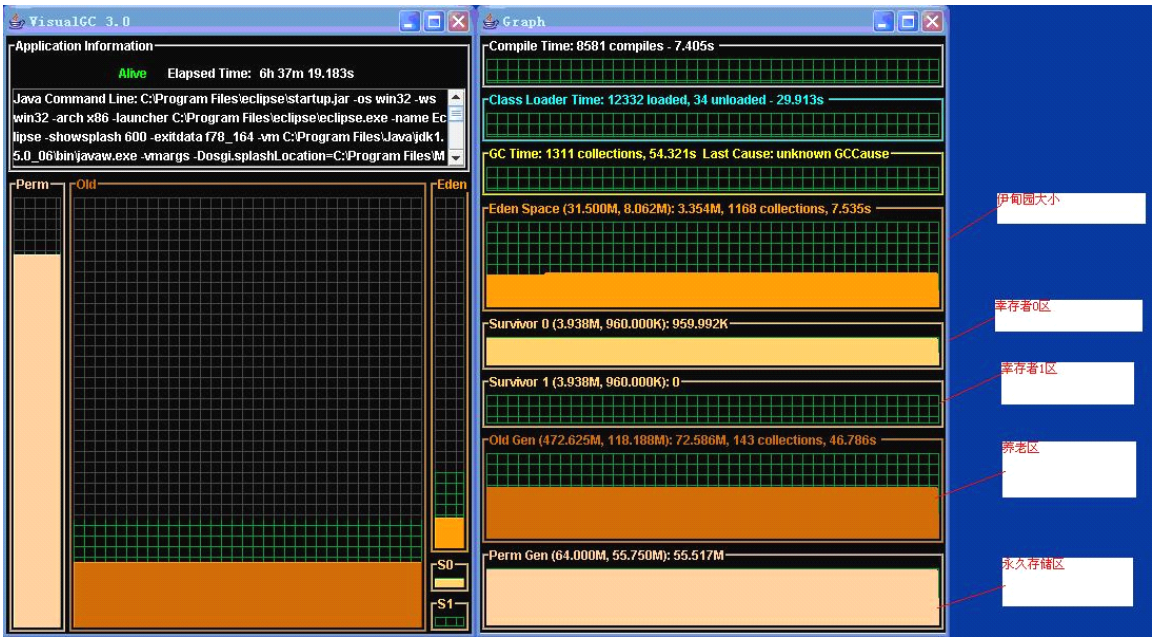
堆空间相关参数

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。
到此我们就不难理解上文提到的“我的服务器有 8g 内存，系统也就 100M 左右，居然出现内存溢出”这个“怪现象”了。在上文我曾提到“永久存储区溢出（java.lang.OutOfMemoryError: Java Permanent Space）”和“JVM 堆空间溢出（java.lang.OutOfMemoryError: Java heap space）”这两种溢出错误。现在大家都知道答案了：“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。在实践中，经常遇到对 JVM 参数进行设置了，并且自己心里觉得应该不会出现内存溢出了；但不幸的是内存溢出还是发生了。很多人百思不得其解，那我可以肯定地告诉你，你设置的 JVM 参数并没有起作用（本文咱不探讨没有起作用的原因）。不信我们就去看看，下面介绍如何使用 SUN 公司的内存使用监控工具 jvmstat.
本文只介绍如何使用 jvmstat 查看内存使用，不介绍其安装配置。有兴趣的读者，可到 SUN 公司的官方网站下载一个，他本身已经带有非常详细的安装配置文档了。这里假设你已经在你的应用服务器上配置好了 jvmstat 了。那我们就开始使用他来看看我们的服务器到底是有没有按照我们设置的参数启动。
首先启动服务器，等服务器启动完。开启 DOS 窗口（此例子是在 windows 下完成，linux 下同样），在 dos 窗口中输入 jps 这个命令。如下图



窗口中会显示所有 JAVA 应用进程列表，列表的第一列为应用的进程 ID,第二列为应用的名字。在列表中找到你的应用服务器的进程 ID,比如我这里的应用服务器进程 ID 为 1856.在命令行输入 visualgc 1856 回车。进入 jvmstat 的主界面,如下图：



上图分别标注了伊甸园，幸存者 0 区，幸存者 1 区，养老区和永久存储区。图上直观的反应出各存储区的大小，已经使用的大小，剩下的空间大小，并用数字标出了各区的大小；如果你这上面的数字和你设置的 JVM 参数相同的话，那么恭喜你，你设置的参数已经起作用，如果和你设置的不一致的话，那么你设置的参数没有起作用（可能是服务器的启动方式没有载入你的 JVM 参数设置。）
在优化服务器的时候，这个工具很有用，他占用资源少。可以随应用服务器一直保持开启状态，如果系统发生内粗溢出，可以一眼就看出是那个区发生了溢出。根据观察结果进行进一步优化。