

---

# JVM 详解

本文详细讲解了 JVM(Java Virtual Machine)的方方面面,首先由 java 的特性来描绘 JVM 的大致应用,再细细阐述了 JVM 的原理及内存管理机制和调优.最后讲述了与 JVM 密切相关的 Java GC 机制.

本文内容大多来自网络,但内容十分丰富,是学习 JVM 的好资料.

后面会再针对 JVM 的两大职责 class loader 和 execution engine 进行讲解

## 目录

---

Java 相关.....	2
1.1Java 定义 .....	3
1.2Java 的开发流程 .....	3
1.3Java 运行的原理 .....	4
1.4 半编译半解释 .....	5
1.5 平台无关性.....	6
JVM 内存模型.....	6
2.1 JVM 规范.....	6
2.2 Sun JVM .....	10
2.3 SUN JVM 内存管理(优化).....	12
2.4 SUN JVM 调优.....	15
2.5.JVM 简单理解 .....	18
2.5.1 Java 栈.....	18
2.5.2 堆 .....	19
2.5.3 堆栈分离的好处.....	22
2.5.4 堆(heap)和栈(stack) .....	22

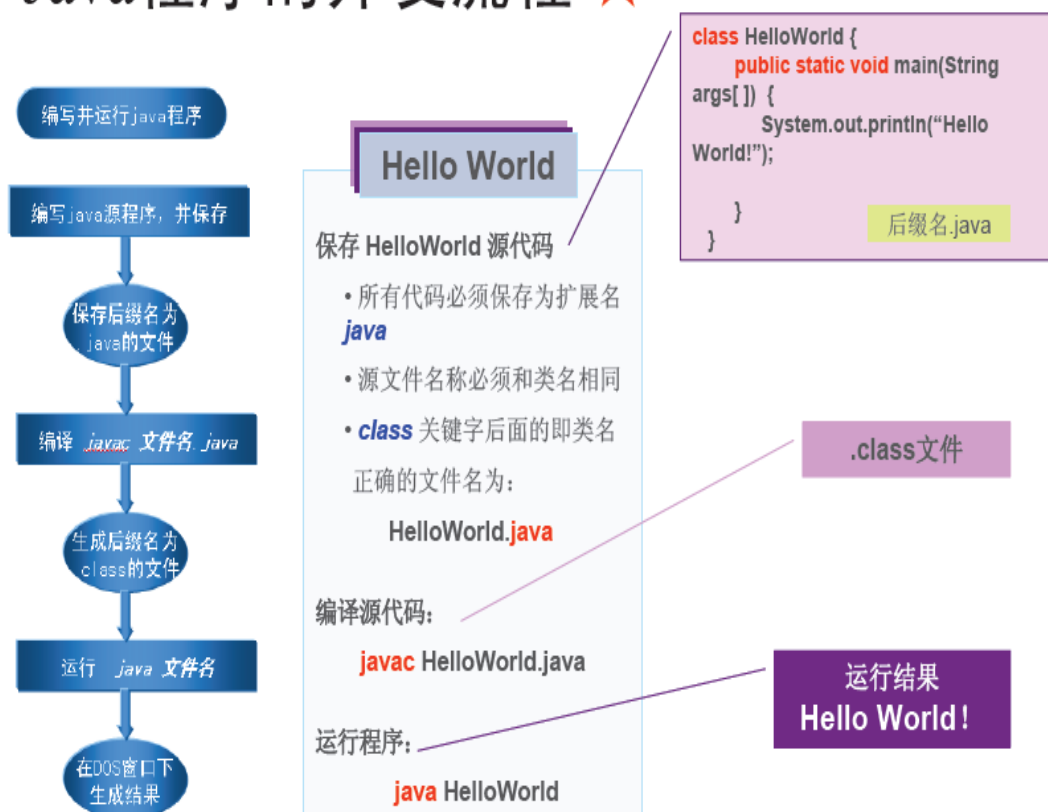
JAVA 垃圾收集器.....	23
3.1 垃圾收集简史 .....	23
3.2 常见的垃圾收集策略 .....	23
3.2.1 Reference Counting(引用计数 ) .....	24
3.2.2 跟踪收集器.....	24
3.3 JVM 的垃圾收集策略.....	30
3.3.1 Serial Collector .....	30
3.3.2 Parallel Collector .....	31
3.3.3 Concurrent Collector.....	32
Java 虚拟机（JVM）参数配置说明.....	33

## 1.1Java 定义

一种简单、面向对象、分布式、跨平台、半编译半解释、健壮、安全、高性能、多线程的动态的语言 —— Sun定义

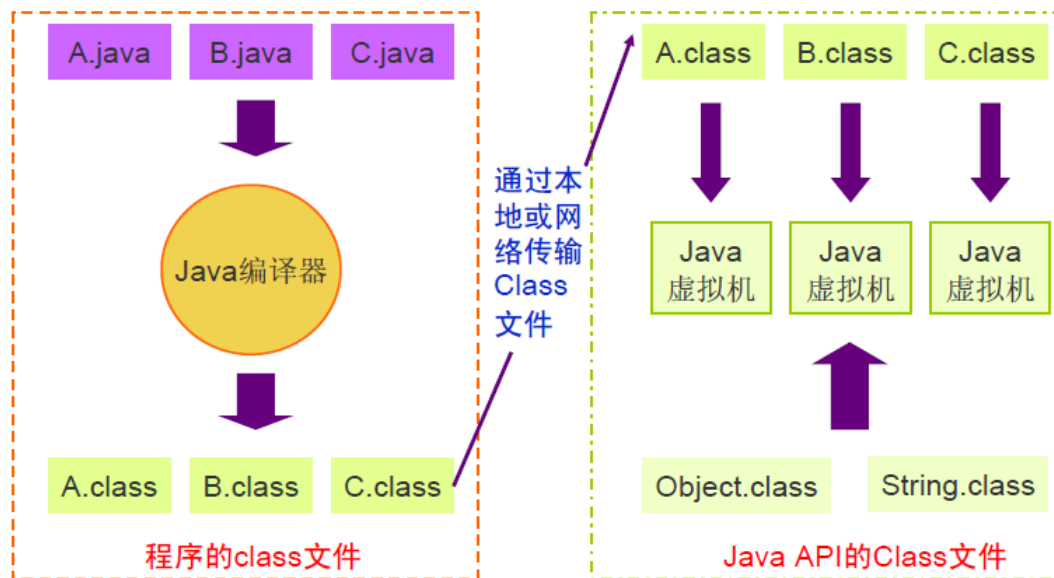
## 1.2Java 的开发流程

# Java程序的开发流程 ★

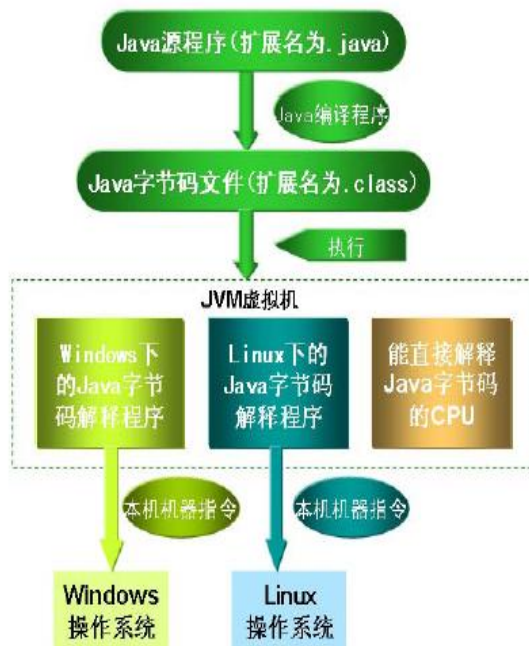


### 1.3Java 运行的原理

## Java运行的原理



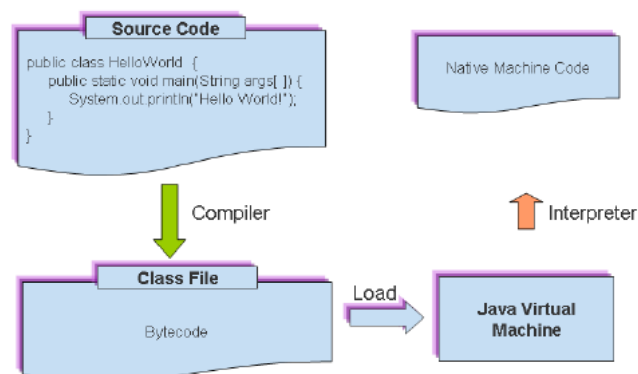
# Java运行的原理



- **JVM (Java Virtual Machine) — Java虚拟机**
  - ✓ 一个虚构出来的计算机
  - ✓ 通过在实际的计算机上仿真模拟各种计算机功能来实现的。
  - ✓ Java虚拟机有自己完善的硬件架构,如处理器、堆栈、寄存器等,还具有相应的指令系统。

## 1.4 半编译半解释

- **半编译半解释?**
  - ✓ 系统先将用户输入的指令翻译成一种通用的, 比较规则的中间形式的代码, 保密性强, 运行时则由所在机器的解释器进行解释
  - ✓ java 语言的开发效率高, 但执行效率低。(相当于c++的55%)



## 1.5 平台无关性

---

- 平台无关性？

- ✓ 何谓平台：即一套特定的硬件再加上运行其上的操作系统，即硬件+软件。编程语言对不同平台的支持有所不同。（VB、C/C++ 、Java）
- ✓ Java完全不用修改任何源代码，也不用重新编译就可以直接移植到其他平台。
- ✓ Java的平台无关性给程序的部署带来了很大的灵活性，节约开发和升级成本。
- ✓ 怎样理解平台无关性呢？JVM (java Virtual Machine)起到了主要作用。JVM是运行在平台之上的程序，它能够虚拟出一台目标机，所有字节码就是在虚拟出的目标机上运行。
- ✓ 程序不可能在所有的平台上都可以运行：（1）因为不同平台的内存管理模式和CPU的指令集等都有很大的差别。（2）为了让java实现平台无关性，Sun公司在不同平台上用软件模拟出虚拟目标机，虚拟出CPU指令集和内存。（3）因此虽然平台间的差异比较大，但是虚拟出来的JVM是完全一样的。（4）Java的字节码仅仅运行在JVM上，不会和平台的底层直接打交道。（5）JVM根据平台的不同，把字节码解释成不同的本地代码（6）JVM就像翻译，把通用的普通话翻译成不同地方特色的方言。
- ✓ 但是有一个缺点：java代码必须要经过JVM解释才能运行，使得java运行的效率降低。
- ✓ WORA: Write Once, Run Anywhere(一次编写,到处运行),

## JVM 内存模型

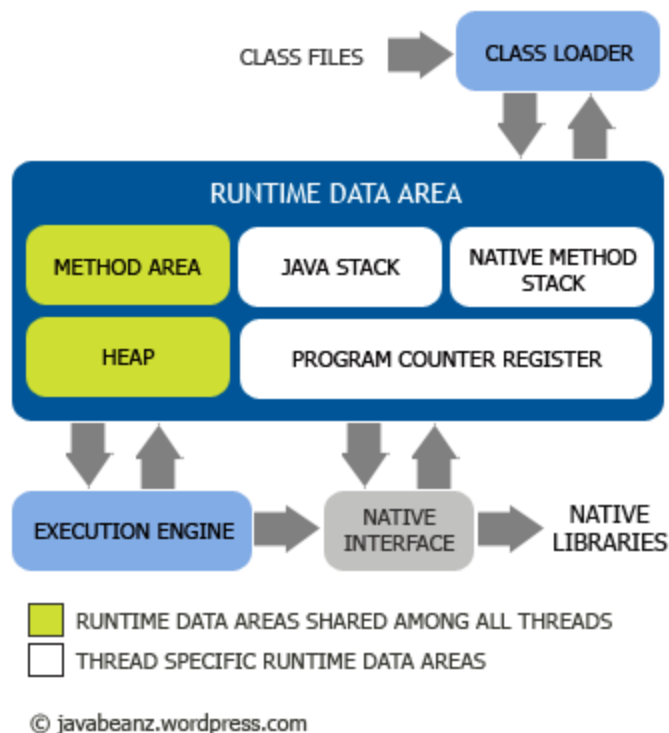
---

### 2.1 JVM 规范

---

### 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：



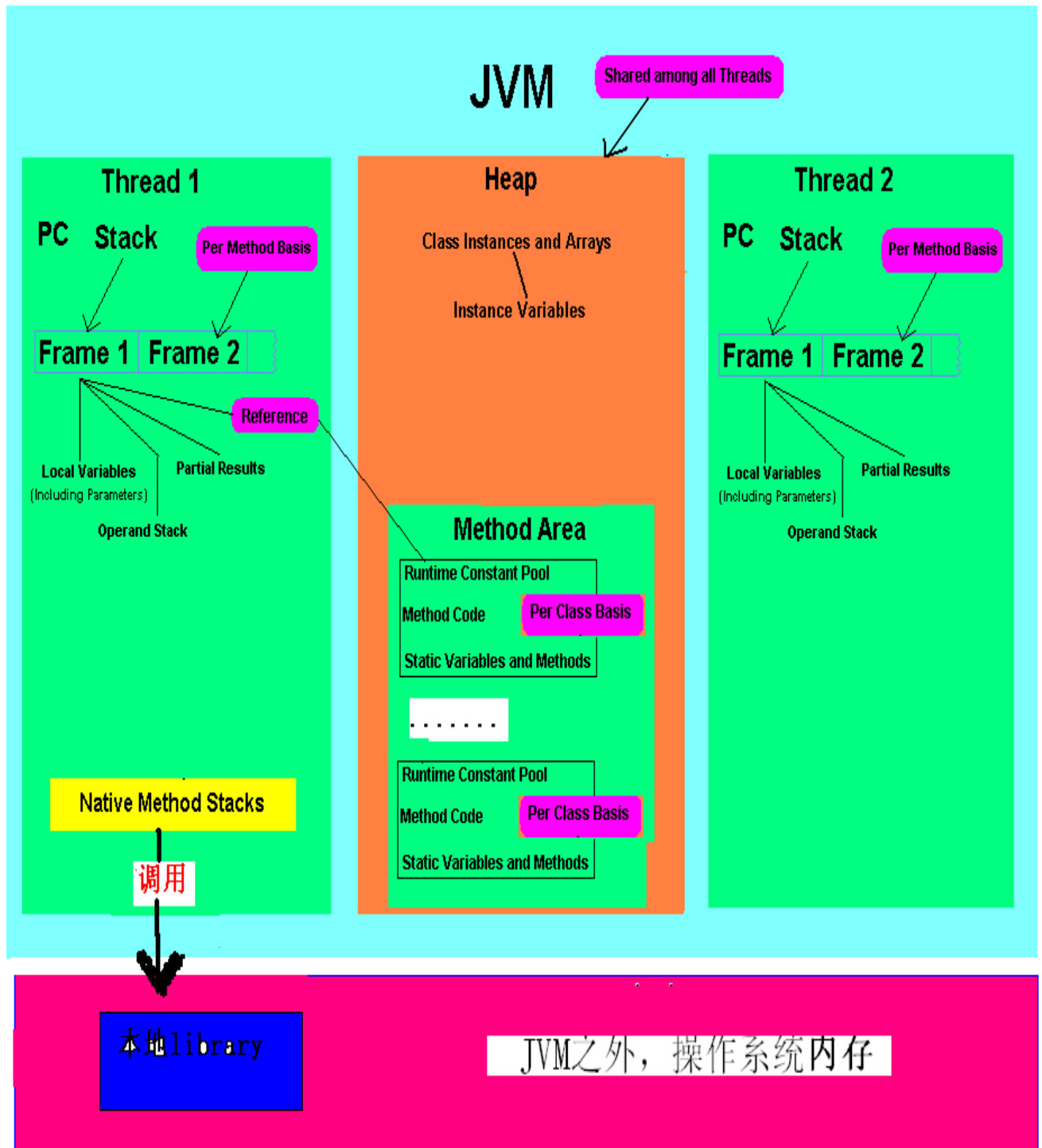
主要包括两个子系统和两个组件：**Class loader(类装载器)** 子系统，**Execution engine(执行引擎)** 子系统；**Runtime data area (运行时数据区域)** 组件，**Native interface(本地接口)** 组件。

**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)。Java 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)**, **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例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，它使用类装载机定位相应的 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“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，不止于此，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 )

## 2.2 Sun JVM

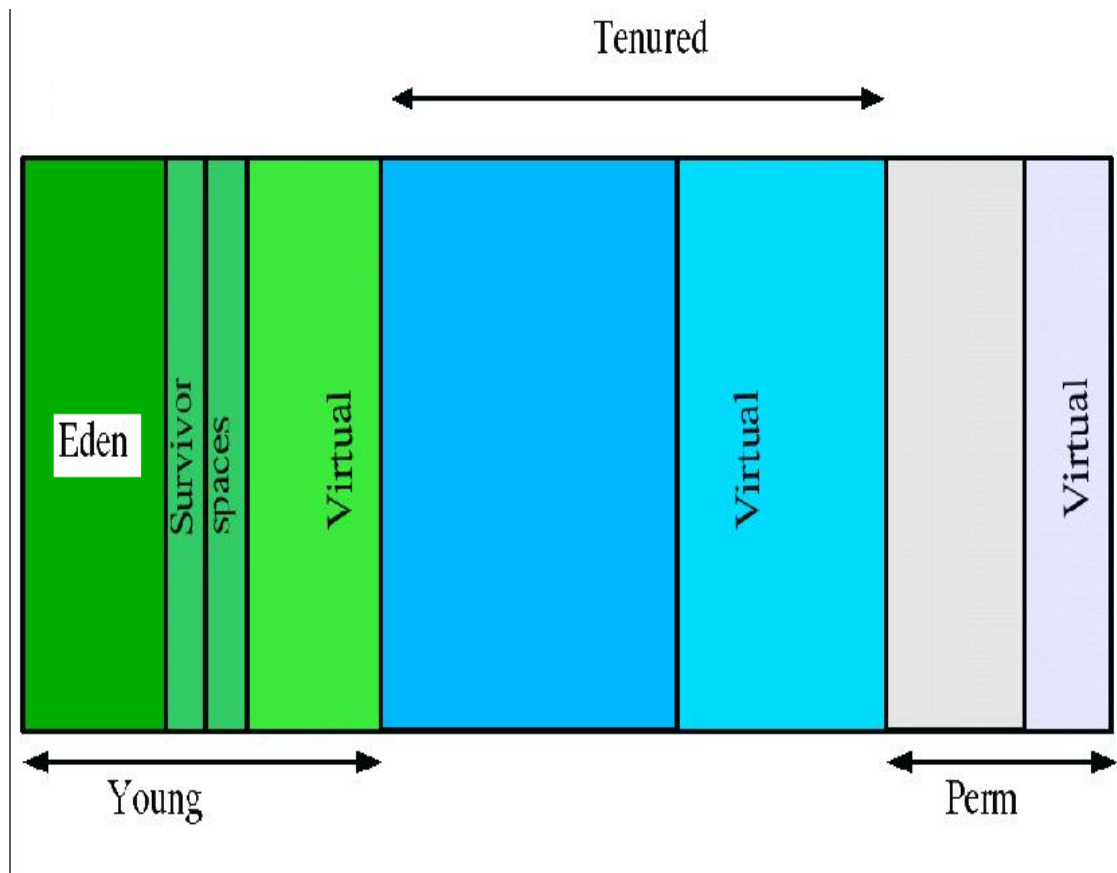
### Sun JVM 中对 JVM Specification 的实现（内存部分）

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 区也满了的时候，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 区过来的对象。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。

## 2.3 SUN JVM 内存管理(优化)

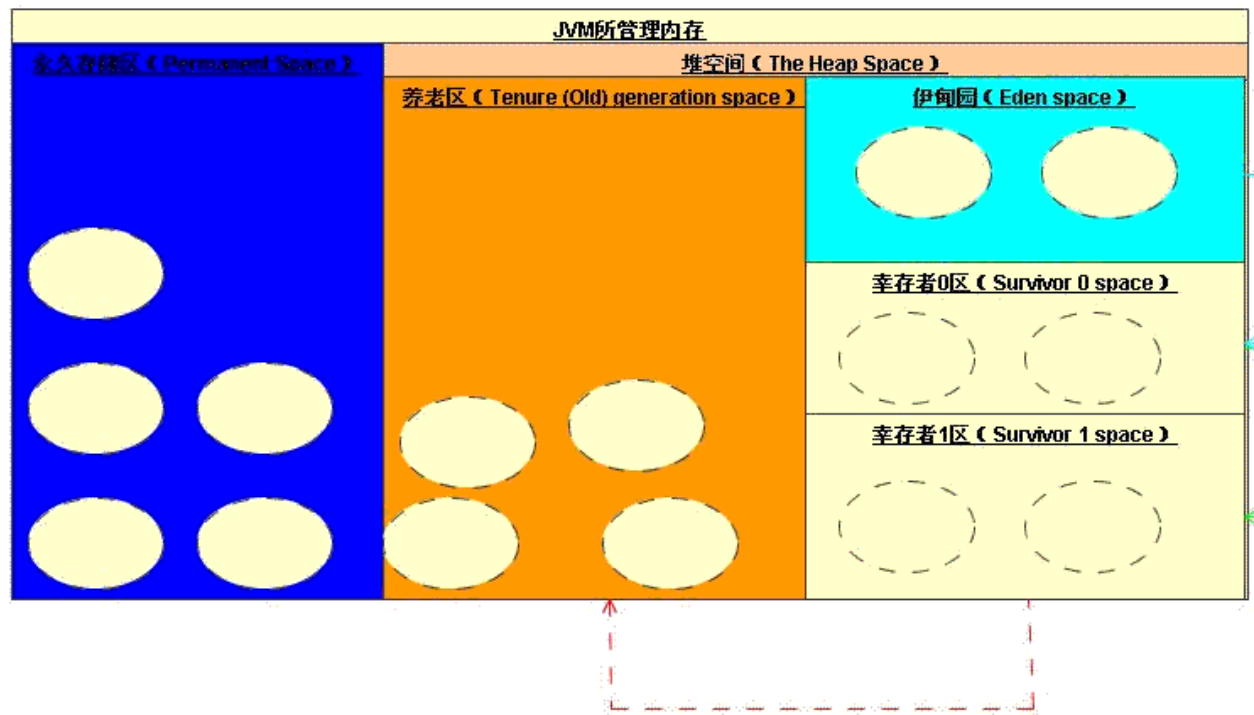
在我做 J2EE 系统开发的工作生涯中，经常遇到技术人员或客户发出诸如此类的感慨：我的 J2EE 应用系统处理的数据量不大，系统体积也不大，技术架构也没有问题，我的应用服务器的内存有 4G 或 8G；系统运行起来很慢，还经常出现内存溢出错误。真是无奈！每次遇到这样的情况，我心中都会忍不住窃笑之。

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,现在我们来查看 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以及编译后的 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。

## 2.4 SUN JVM 调优

---

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 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

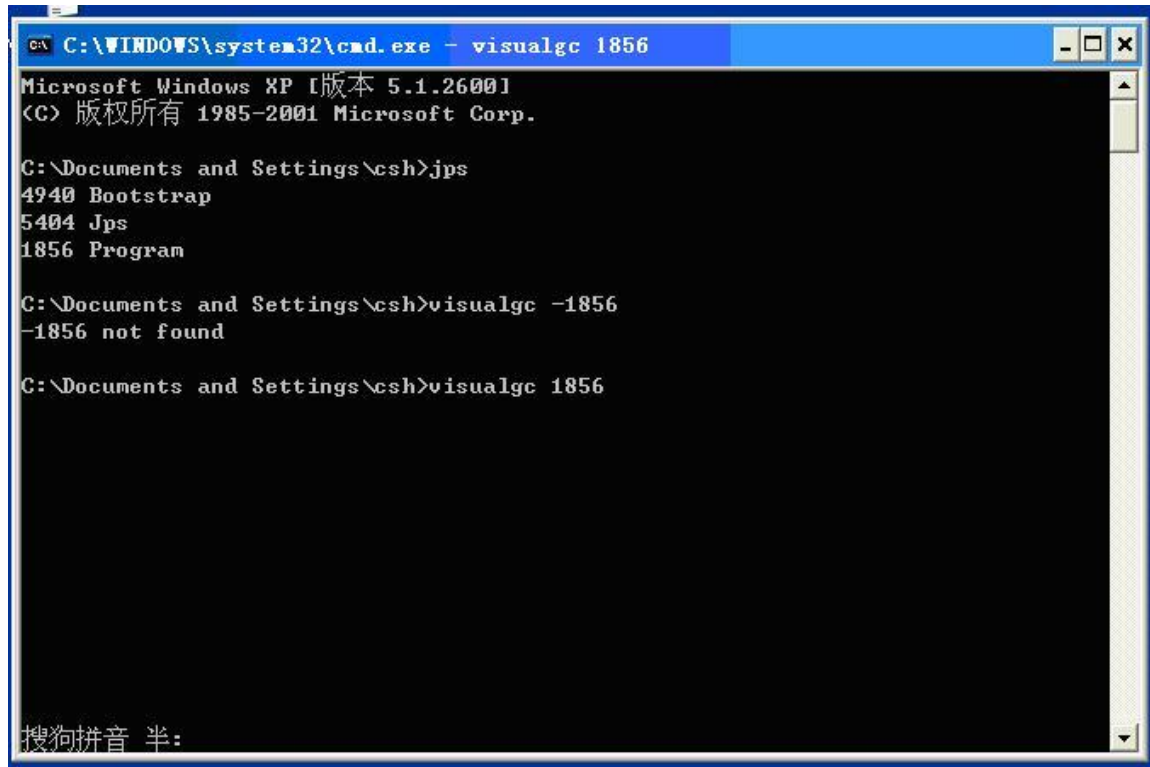
(java.lang.OutOfMemoryError: Java Permanent Space)”和“JVM 堆空间溢出 (java.lang.OutOfMemoryError: Java heap space)”这两种溢出错误。现在大家都知道答案了:“永久存储区溢出 (java.lang.OutOfMemoryError: Java Permanent Space)”乃是永久存储区设置太小,不能满足系统需要的大小,此时只需要调整-XX:PermSize 和-XX:MaxPermSize 这两个参数即可。“JVM 堆空间溢出 (java.lang.OutOfMemoryError: Java 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





```
C:\WINDOWS\system32\cmd.exe - visualgc 1856
Microsoft Windows XP [版本 5.1.2600]
(C) 版权所有 1985-2001 Microsoft Corp.

C:\Documents and Settings\csh>jps
4940 Bootstrap
5404 Jps
1856 Program

C:\Documents and Settings\csh>visualgc -1856
-1856 not found

C:\Documents and Settings\csh>visualgc 1856

搜狗拼音 半:
```

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。可以随时应用服务器一直保持开启状态，如果系统发生内存溢出，可以一眼就看出是哪个区发生了溢出。根据观察结果进行进一步优化。

## 2.5.JVM 简单理解

### 2.5.1 Java 栈

Java 栈是与每一个线程关联的，JVM 在创建每一个线程的时候，会分配一定的栈空间给线程。它主要用来存储线程执行过程中的局部变量，方法的返回值，以及方法调用上下文。栈空间随着线程的终止而释放。

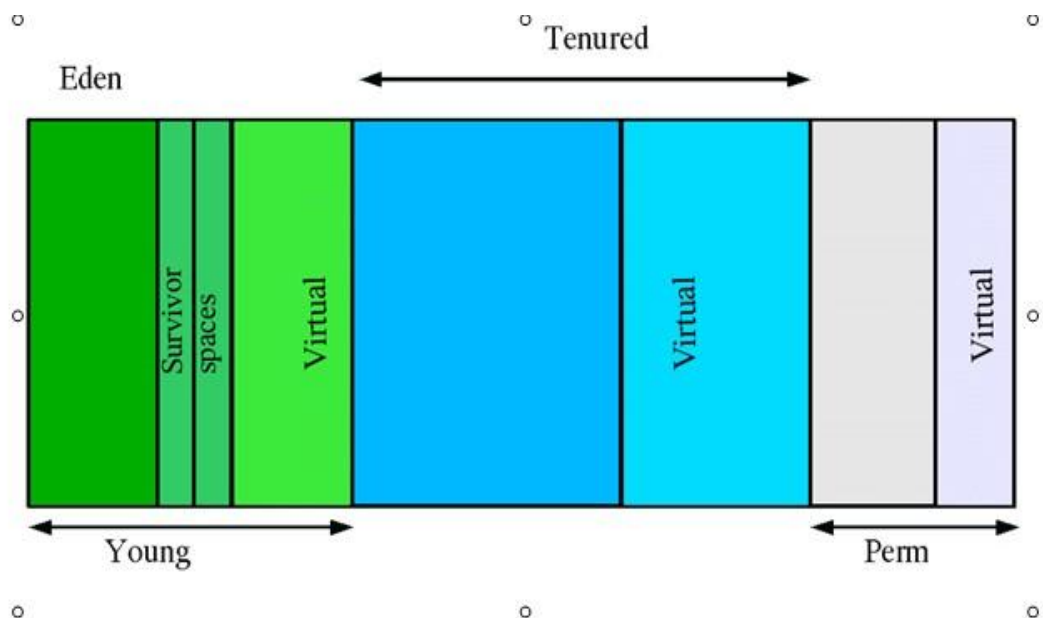
**StackOverflowError**：如果在线程执行的过程中，栈空间不够用，那么 JVM 就会抛出此异常，这种情况一般是死递归造成的。

## 2.5.2 堆

Java 中堆是由所有的线程共享的一块内存区域，堆用来保存各种 JAVA 对象，比如数组，线程对象等。

### 2.5.2.1 Generation

JVM 堆一般又可以分为以下三部分：



#### ➤ Perm

Perm 代主要保存 class, method, filed 对象，这部门的空间一般不会溢出，除非一次性加载了很多的类，不过在涉及到热部署的应用服务器的时候，有时候会遇到 `java.lang.OutOfMemoryError: PermGen space` 的错误，造成这个错误的很大原因就有可能是每次都重新部署，但是重新部署后，类的 class 没有被卸载掉，这样就造成了大量的 class 对象保存在了 perm 中，这种情况下，一般重新启动应用服务器可以解决问题。

### ➤•Tenured

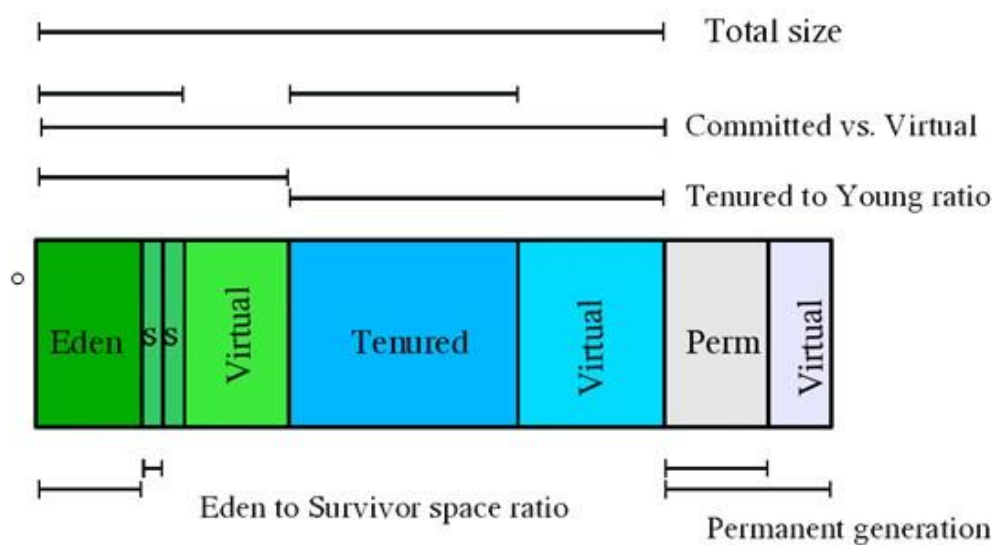
Tenured 区主要保存生命周期长的对象，一般是一些老的对象，当一些对象在 Young 复制转移一定的次数以后，对象就会被转移到 Tenured 区，一般如果系统中用了 application 级别的缓存，缓存中的对象往往会被转移到这一区间。

### ➤•Young

Young 区被划分为三部分，Eden 区和两个大小严格相同的 Survivor 区，其中 Survivor 区间中，某一时刻只有其中一个是被使用的，另外一个留做垃圾收集时复制对象用，在 Young 区间变满的时候，minor GC 就会将存活的对象移到空闲的 Survivor 区间中，根据 JVM 的策略，在经过几次垃圾收集后，任然存活于 Survivor 的对象将被移动到 Tenured 区间。

## 2.5.2.2 Sizing the Generations

JVM 提供了相应的参数来对内存大小进行配置。



正如上面描述，JVM 中堆被分为了 3 个大的区间，同时 JVM 也提供了一些选项对 Young, Tenured 的大小进行控制。

#### ➤•Total Heap

-Xms : 指定了 JVM 初始启动以后初始化内存

-Xmx : 指定 JVM 堆得最大内存，在 JVM 启动以后，会分配-Xmx 参数指定大小的内存给 JVM，但是不一定全部使用，JVM 会根据-Xms 参数来调节真正用于 JVM 的内存

-Xmx -Xms 之差就是三个 Virtual 空间的大小

#### ➤•Young Generation

-XX:NewRatio=8 意味着 tenured 和 young 的比值 8 : 1，这样  $\text{eden} + 2 * \text{survivor} = 1/9$

堆内存

-XX:SurvivorRatio=32 意味着 eden 和一个 survivor 的比值是 32 : 1，这样一个 Survivor 就占 Young 区的 1/34.

-Xmn 参数设置了年轻代的大小

#### ➤•Perm Generation

-XX:PermSize=16M -XX:MaxPermSize=64M

Thread Stack

-XX:Max=128K

### 2.5.3 堆栈分离的好处

---

呵 呵，其它的先不说了，就来说说面向对象的设计吧，当然除了面向对象的设计带来的维护性，复用性和扩展性方面的好处外，我们看看面向对象如何巧妙的利用了堆 栈分离。如果从 JAVA 内存模型的角度去理解面向对象的设计，我们就会发现对象它完美的表示了堆和栈，对象的数据放在堆中，而我们编写的那些方法一般都是 运行在栈中，因此面向对象的设计是一种非常完美的设计方式，它完美的统一了**数据存储和运行**。

### 2.5.4 堆(heap)和栈(stack)

---

什么叫堆？你用十几个麻将牌竖直叠成一摞这叫堆，你可以从上面、下面、中间任意抽出一张牌，也可以任意插入一张。

什么叫栈？**AK-47**的弹匣就是一个栈，在上面的子弹没被取出之前，你无法取出下面的子弹——尽管你可以从边上的透明部分读出里面装的是什么型号、颜色的子弹。

堆很灵活，但是不安全。对于对象，我们要动态地创建、销毁，不能说后创建的对象没有销毁，先前创建的对象就不能销毁，那样的话我们的程序就寸步难行，所以 **Java** 中用堆来存储对象。而一旦堆中的对象被销毁，我们继续引用这个对象的话，就会出现著名的 **NullPointerException**，这就是堆的缺点——错误的引用逻辑只有在运行时才会被发现。

栈不灵活，但是很严格，是安全的，易于管理。因为只要上面的引用没有销毁，下面引用就一定还在，所以，在栈中，上面引用永远可以通过下面引用来查找对象，同时如果确认某一区间的内容会一起存在、一起销毁，也可以上下互相引用。在大部分程序中，都是先定义的变量、引用先进栈，后定义的后进栈，同时，区块内部的变量、引用在进入区块时压栈，区块结束时出栈，理解了这种机制，我们就可以很方便地理解各种编程语言的作用域的概念了，同时这也是栈的优点——错误的引用逻辑在编译时就可以被发现。

### 举例说明

简单的说 其实 栈 就是存放变量引用的一个地方， 堆 就是存放实际对象的地方 也就是。

比如： `int i = 7;` 这个 其实是存在栈里边的。内容为 `i = 7`。

`Apple app = new Apple();` 这个 `app` 是在栈里边的 他对应的是一个内存地址也在堆里边，而这个内存地址对应的是堆里边存放 `Apple` 实例的地址。

`String s = "Hello World!";` 这个其实是存在另外一块静态代码区。

总体来说： 栈--主要存放引用 和基本数据类型。

堆--用来存放 `new` 出来的对象实例。

## JAVA 垃圾收集器

---

### 3.1 垃圾收集简史

---

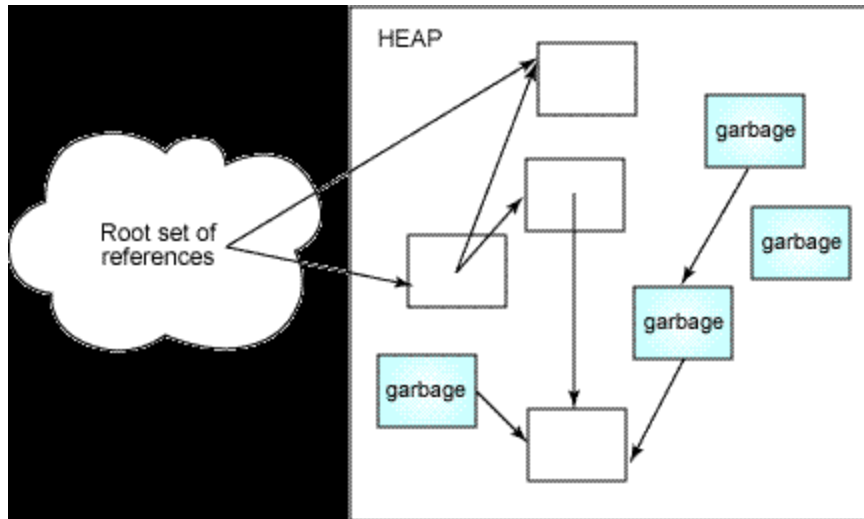
垃圾收集提供了内存管理的机制，使得应用程序不需要在关注内存如何释放，内存用完后，垃圾收集会进行收集，这样就减轻了因为人为的管理内存而造成的错误，比如在 C++ 语言里，出现内存泄露时很常见的。

Java 语言是目前使用最多的依赖于垃圾收集器的语言，但是垃圾收集器策略从 20 世纪 60 年代就已经流行起来了，比如 Smalltalk, Eiffel 等编程语言也集成了垃圾收集器的机制。

### 3.2 常见的垃圾收集策略

---

所有的垃圾收集算法都面临同一个问题，那就是找出应用程序不可到达的内存块，将其释放，这里面得不可到达主要是指应用程序已经没有内存块的引用了，而在 JAVA 中，某个对象对应用程序是可到达的是指：这个对象被根（根主要是指类的静态变量，或者活跃在所有线程栈的对象的引用）引用或者对象被另一个可到达的对象引用。



### 3.2.1 Reference Counting(引用计数 )

---

引用计数是最简单直接的一种方式，这种方式在每一个对象中增加一个引用的计数，这个计数代表当前程序有多少个引用引用了此对象，如果此对象的引用计数变为 0，那么此对象就可以作为垃圾收集器的目标对象来收集。

优点：

简单，直接，不需要暂停整个应用

缺点：

1.需要编译器的配合，编译器要生成特殊的指令来进行引用计数的操作，比如每次将对象赋值给新的引用，或者者对象的引用超出了作用域等。

2.不能处理循环引用的问题

### 3.2.2 跟踪收集器

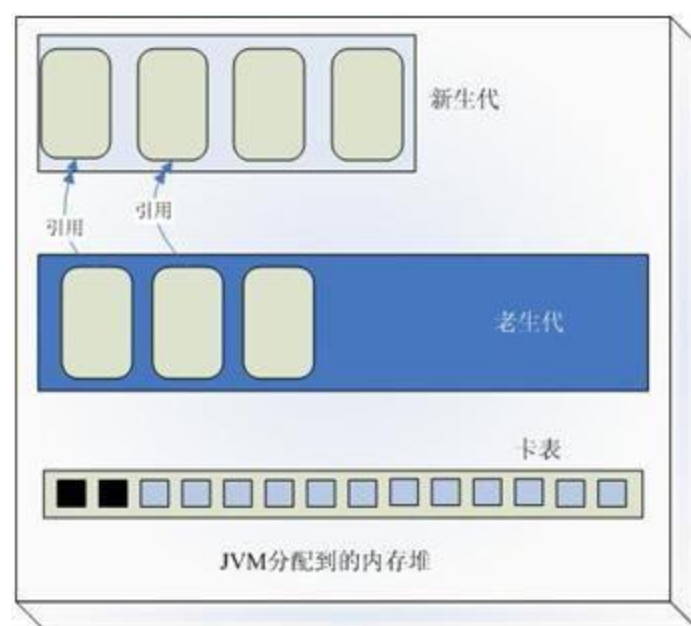
---

跟踪收集器首先要暂停整个应用程序，然后开始从根对象扫描整个堆，判断扫描的对象是否有对象引用，这里面有三个问题需要搞清楚：



1. 如果每次扫描整个堆，那么势必让 GC 的时间变长，从而影响了应用本身的执行。因此在 JVM 里面采用了分代收集，在新生代收集的时候 minor gc 只需要扫描新生代，而不需要扫描老生代。

2. JVM 采用了分代收集以后，minor gc 只扫描新生代，但是 minor gc 怎么判断是否有老生代的对象引用了新生代的对象，JVM 采用了卡片标记的策略，卡片标记将老生代分成了一块一块的，划分以后的每一个块就叫做一个卡片，JVM 采用卡表维护了每一个块的状态，当 JAVA 程序运行的时候，如果发现老生代对象引用或者释放了新生代对象的引用，那么就 JVM 就将卡表的状态设置为脏状态，这样每次 minor gc 的时候就会只扫描被标记为脏状态的卡片，而不需要扫描整个堆。具体如下图：



3. GC 在收集一个对象的时候会判断是否有引用指向对象，在 JAVA 中的引用主要有四种：

Strong reference, Soft reference, Weak reference, Phantom reference.

➤•Strong Reference

强引用是 JAVA 中默认采用的一种方式，我们平时创建的引用都属于强引用。如果一个对象没有强引用，那么对象就会被回收。

```
public void testStrongReference() {  
  
    Object referent = new Object();  
  
    Object strongReference = referent;  
  
    referent = null;  
  
    System.gc();  
  
    assertNotNull(strongReference);  
  
}
```

#### ➤•Soft Reference

软引用的对象在 GC 的时候不会被回收，只有当内存不够用的时候才会真正的回收，因此软引用适合缓存的场合，这样使得缓存中的对象可以尽量在内存中待长久一点。

```
Public void testSoftReference(){  
  
    String str = "test";  
  
    SoftReference<String> softreference = new SoftReference<String>(str);  
  
    str=null;
```

```
        System.gc();

        assertNotNull(softreference.get());

    }
```

#### ➤Weak reference

弱引用有利于对象更快的被回收，假如一个对象没有强引用只有弱引用，那么在 GC 后，这个对象肯定会被回收。

```
Public void testWeakReference(){

    String str = "test";

    WeakReference<String> weakReference = new WeakReferenc
e<String>(str);

    str=null;

    System.gc();

    assertNull(weakReference.get());

}
```

➤•Phantom reference

### 3.2.2.1 Mark-Sweep Collector(标记-清除收集器)

---

标记清除收集器最早由 Lisp 的发明人于 1960 年提出，标记清除收集器停止所有的工作，从根扫描每个活跃的对象，然后标记扫描过的对象，标记完成以后，清除那些没有被标记的对象。

优点：

- 1 解决循环引用的问题
- 2 不需要编译器的配合，从而就不执行额外的指令

缺点：

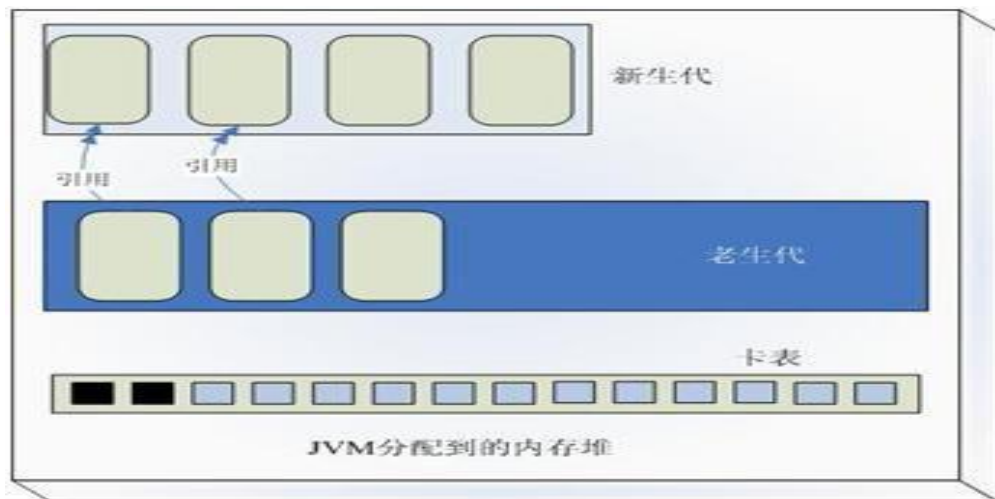
1. 每个活跃的对象都要进行扫描，收集暂停的时间比较长。

### 3.2.2.2 Copying Collector(复制收集器)

---

复制收集器将内存分为两块一样大小空间，某一个时刻，只有一个空间处于活跃的状态，当活跃的空间满的时候，GC 就会将活跃的对象复制到未使用的空间中去，原来不活跃的空间就变为了活跃的空间。

复制收集器具体过程可以参考下图：



优点：

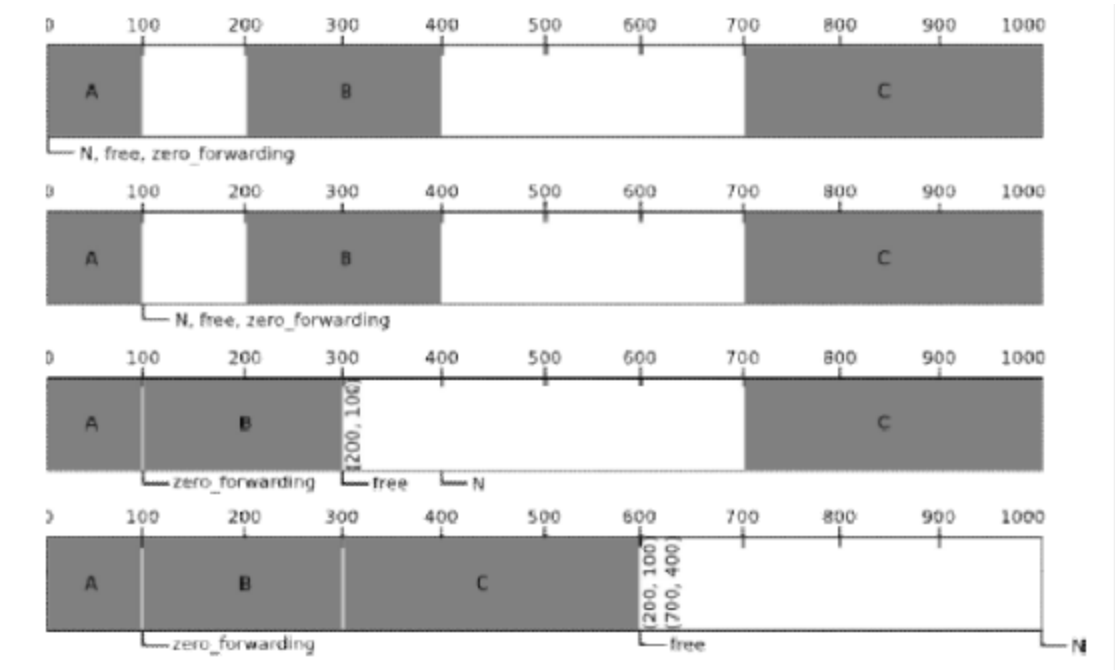
1 只扫描可以到达的对象，不需要扫描所有的对象，从而减少了应用暂停的时间

缺点：

1. 需要额外的空间消耗，某一个时刻，总是有一块内存处于未使用状态
2. 复制对象需要一定的开销

### 3.2.2.3 Mark-Compact Collector(标记-整理收集器)

标记整理收集器汲取了标记清除和复制收集器的优点，它分两个阶段执行，在第一个阶段，首先扫描所有活跃的对象，并标记所有活跃的对象，第二个阶段首先清除未标记的对象，然后将活跃的的对象复制到堆得底部。标记整理收集器的过程示意图请参考下图：



Mark-compact 策略极大的减少了内存碎片，并且不需要像 Copy Collector 一样需要两倍的空间。

### 3.3 JVM 的垃圾收集策略

GC 的执行时要耗费一定的 CPU 资源和时间的，因此在 JDK1.2 以后，JVM 引入了分代收集的策略，其中对新生代采用"Mark-Compact"策略，而对老生代采用了"Mark-Sweep"的策略。其中新生代的垃圾收集器命名为"minor gc"，老生代的 GC 命名为"Full Gc 或者 Major GC"。其中用 `System.gc()` 强制执行的是 Full Gc。

#### 3.3.1 Serial Collector

Serial Collector 是指任何时刻都只有一个线程进行垃圾收集，这种策略有一个名字"stop the whole world", 它需要停止整个应用的执行。这种类型的收集器适合于单 CPU 的机器。

#### Serial Copying Collector

此种 GC 用 `-XX:UseSerialGC` 选项配置，它只用于 **新生代** 对象的收集。

1.5.0 以后.

`-XX:MaxTenuringThreshold` 来设置对象复制的次数。当 eden 空间不够的时候，GC 会将 eden 的活跃对象和一个名叫 From survivor 空间中尚不够资格放入 Old 代的对象复制到另外一个名字叫 To Survivor 的空间。而此参数就是用来说明到底 From survivor 中的哪些对象不够资格，假如这个参数设置为 31，那么也就是说只有对象复制 31 次以后才算是合格的对象。

这里需要注意几个问题：

- From Survivor 和 To survivor 的角色是不断变化的，同一时间只有一块空间处于使用状态，这个空间就叫做 From Survivor 区，当复制一次后角色就发生了变化。
- 如果复制的过程中发现 To survivor 空间已经满了，那么就直接复制到 old generation.
- 比较大的对象也会直接复制到 Old generation, 在开发中，我们应该尽量避免这种情况的发生。

## **Serial     Mark-Compact   Collector**

串行的标记-整理收集器是 JDK5 update6 之前默认的老生代的垃圾收集器，此收集使得内存碎片最少化，但是它需要暂停的时间比较长

### 3.3.2 Parallel Collector

---

Parallel Collector 主要是为了应对多 CPU，大数据量的环境。

Parallel Collector 又可以分为以下两种：

#### **Parallel Copying Collector**

此种 GC 用-XX:UseParNewGC 参数配置,它主要用于新生代的收集,此 GC 可以配合 CMS 一起使用。1.4.1 以后

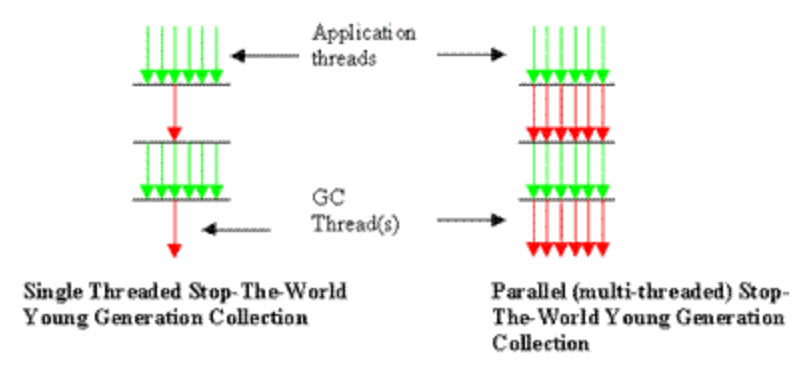
### Parallel Mark-Compact Collector

此种 GC 用-XX:UseParallelOldGC 参数配置，此 GC 主要用于老年代对象的收集。1.6.0

### Parallel scavenging Collector

此种 GC 用-XX:UseParallelGC 参数配置，它是对新生代对象的垃圾收集器，但是它不能和 CMS 配合使用，它适合于比较大新生代的情况，此收集器起始于 jdk 1.4.0。它比较适合于对吞吐量高于暂停时间的场合。

Serial gc 和 Parallel gc 可以用如下的图来表示：

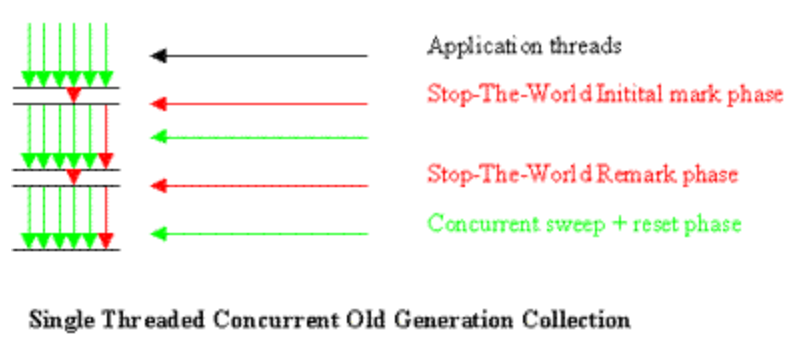


### 3.3.3 Concurrent Collector

Concurrent Collector 通过并行的方式进行垃圾收集，这样就减少了垃圾收集器收集一次的时间，这种 GC 在实时性要求高于吞吐量比较有用。

此种 GC 可以用参数-XX:UseConcMarkSweepGC 配置，此 GC 主要用于老年代和 Perm 代的收集。





## Java 虚拟机（JVM）参数配置说明

在 Java、J2EE 大型应用中，JVM 非标准参数的配置直接关系到整个系统的性能。

JVM 非标准参数指的是 JVM 底层的一些配置参数，这些参数在一般开发中默认即可，不需要任何配置。但是在生产环境中，为了提高性能，往往需要调整这些参数，以求系统达到最佳性能。

另外这些参数的配置也是影响系统稳定性的一个重要因素，相信大多数 Java 开发人员都见过“OutOfMemory”类型的错误。呵呵，这其中很可能就是 JVM 参数配置不当或者就没有配置没意识到配置引起的。

为了说明这些参数，还需要说说 JDK 中的命令行工具一些知识做铺垫。

### 首先看如何获取这些命令配置信息说明：

假设你是 windows 平台，你安装了 J2SDK，那么现在你从 cmd 控制台窗口进入 J2SDK 安装目录下的 bin 目录，然后运行 java 命令，出现如下结果，这些就是包括 java.exe 工具的和 JVM 的所有命令都在里面。

```
-----  
D:\j2sdk15\bin>java
```

```
Usage: java [-options] class [args...]
```

(to execute a class)

or `java [-options] -jar jarfile [args...]`

(to execute a jar file)

where options include:

`-client` to select the "client" VM

`-server` to select the "server" VM

`-hotspot` is a synonym for the "client" VM [deprecated]

The default VM is client.

`-cp <class search path of directories and zip/jar files>`

`-classpath <class search path of directories and zip/jar files>`

A ; separated list of directories, JAR archives,  
and ZIP archives to search for class files.

`-D<name>=<value>`

set a system property

`-verbose[:class|gc|jni]`

enable verbose output

`-version` print product version and exit

`-version:<value>`

require the specified version to run

`-showversion` print product version and continue

`-jre-restrict-search | -jre-no-restrict-search`

include/exclude user private JREs in the version search

`-? -help` print this help message

`-X` print help on non-standard options

`-ea[:<packagename>...]:<classname>]`

`-enableassertions[:<packagename>...]:<classname>]`

enable assertions

`-da[:<packagename>...]:<classname>]`

-disableassertions[: <packagename>...| : <classname>]  
disable assertions

-esa | -enablesystemassertions  
enable system assertions

-dsa | -disablesystemassertions  
disable system assertions

-agentlib: <libname>[ = <options>]  
load native agent library <libname>, e.g. -agentlib:hprof  
see also, -agentlib:jdwp=help and -agentlib:hprof=help

-agentpath: <pathname>[ = <options>]  
load native agent library by full pathname

-javaagent: <jarpath>[ = <options>]  
load Java programming language agent, see java.lang.instrume

nt

-----  
在控制台输出信息中，有个-X（注意是大写）的命令，这个正是查看 JVM 配置参数的命令。

其次，用 **java -X** 命令查看 JVM 的配置说明：

运行后如下结果，这些就是配置 JVM 参数的秘密武器，这些信息都是英文的，为了方便阅读，我根据自己的理解翻译成中文了（不准确的地方还请各位博友斧正）

-----  
D:\j2sdk15\bin>java -X

-Xmixed            mixed mode execution (default)

-Xint            interpreted mode execution only

-Xbootclasspath:<directories and zip/jar files separated by ;>  
set search path for bootstrap classes and resources

-Xbootclasspath/a:<directories and zip/jar files separated by ;>  
append to end of bootstrap class path

-Xbootclasspath/p:<directories and zip/jar files separated by ;>  
prepend in front of bootstrap class path

- Xnoclassgc      disable class garbage collection
- Xincgc          enable incremental garbage collection
- Xloggc:<file>    log GC status to a file with time stamps
- Xbatch          disable background compilation
- Xms<size>        set initial Java heap size
- Xmx<size>        set maximum Java heap size
- Xss<size>        set java thread stack size
- Xprof            output cpu profiling data
- Xfuture          enable strictest checks, anticipating future default
- Xrs              reduce use of OS signals by Java/VM (see documentation)
- Xcheck:jni        perform additional checks for JNI functions
- Xshare:off        do not attempt to use shared class data
- Xshare:auto      use shared class data if possible (default)
- Xshare:on        require using shared class data, otherwise fail.

The -X options are non-standard and subject to change without notice.

-----

JVM 配置参数中文说明:

-----

1、-Xmixed          mixed mode execution (default)

混合模式执行

2、-Xint            interpreted mode execution only

解释模式执行

3、-Xbootclasspath:<directories and zip/jar files separated by ;>

set search path for bootstrap classes and resources

设置 zip/jar 资源或者类（.class 文件）存放目录路径

3、-Xbootclasspath/a:<directories and zip/jar files separated by ;>

append to end of bootstrap class path

追加 zip/jar 资源或者类（.class 文件）存放目录路径

4、-Xbootclasspath/p:<directories and zip/jar files separated by ;>

prepend in front of bootstrap class path

预先加载 zip/jar 资源或者类（.class 文件）存放目录路径

5、-Xnoclassgc        disable class garbage collection

关闭类垃圾回收功能

6、-Xincgc            enable incremental garbage collection

开启类的垃圾回收功能

7、-Xloggc:<file>    log GC status to a file with time stamps

记录垃圾回日志到一个文件。

8、-Xbatch            disable background compilation

关闭后台编译

9、-Xms<size>        set initial Java heap size

设置 JVM 初始化堆内存大小

10、-Xmx<size>        set maximum Java heap size

设置 JVM 最大的堆内存大小

11、-Xss<size>        set java thread stack size

设置 JVM 栈内存大小

12、-Xprof            output cpu profiling data

输入 CPU 概要表数据

13、-Xfuture            enable strictest checks, anticipating future default

执行严格的代码检查，预测可能出现的情况

14、-Xrs                reduce use of OS signals by Java/VM (see documentation)

通过 JVM 还原操作系统信号

15、-Xcheck:jni        perform additional checks for JNI functions

对 JNI 函数执行检查

16、-Xshare:off        do not attempt to use shared class data

尽可能不去使用共享类的数据

17、-Xshare:auto       use shared class data if possible (default)

尽可能的使用共享类的数据

18、-Xshare:on         require using shared class data, otherwise fail.

尽可能的使用共享类的数据，否则运行失败

The -X options are non-standard and subject to change without notice.

-----

怎么用这这些参数呢？其实所有的命令行都是这么一用，下面我就给出一个最简单的 **Hello Worl** 的例子来演示这个参数的用法，非常的简单。

HelloWorld.java

-----

```
public class HelloWorld
```

```
{  
    public static void main(String[] args)  
    {  
        System.out.println("Hello World!");  
    }  
}
```

编译并运行：

```
D:\j2sdk15\bin>javac HelloWorld.java
```

```
D:\j2sdk15\bin>java -Xms256M -Xmx512M HelloWorld
```

Hello World!

呵呵，这下满足了吧！

### 实践：在大型系统或者应用中配置 JVM 参数

比如你配置 IDE 工具的参数，常见的有 IDEA、Eclipse，这个是在一个配置文件中指定即可。

如果你要在 J2EE 环境中配置这些参数，那么你需要在 J2EE 应用服务器或者 Servlet 容器相关启动参数设置处指定，其启动文件中来配置，Tomcat 是在 catalina.bat 中配置，weblogic 和 websphere 是在其他地方，具体我就说了，相信玩过的这些大型服务器的人都知道，没玩过的看看这篇文章，玩玩就知道了，呵呵。

另外常常有人问到 jdk 的一些相关命令用法，其实，当你看到这里的时候，你应该知道如何获取这些命令的用法了。如果你还不会，那么，建议你去学学 DOS，我是没辙了。如果你会这些，还是没有看明白，那么你赶紧学学英语吧，这样你就能看懂了。

另外：我在最后给出常用的几个 Java 命令行说明，以供参考：

(1)、javac

用法: `javac <选项> <源文件>`

其中, 可能的选项包括:

- `-g` 生成所有调试信息
- `-g:none` 不生成任何调试信息
- `-g:{lines,vars,source}` 只生成某些调试信息
- `-nowarn` 不生成任何警告
- `-verbose` 输出有关编译器正在执行的操作的消息
- `-deprecation` 输出使用已过时的 API 的源位置
- `-classpath <路径>` 指定查找用户类文件的位置
- `-cp <路径>` 指定查找用户类文件的位置
- `-sourcepath <路径>` 指定查找输入源文件的位置
- `-bootclasspath <路径>` 覆盖引导类文件的位置
- `-extdirs <目录>` 覆盖安装的扩展目录的位置
- `-endorseddirs <目录>` 覆盖签名的标准路径的位置
- `-d <目录>` 指定存放生成的类文件的位置
- `-encoding <编码>` 指定源文件使用的字符编码
- `-source <版本>` 提供与指定版本的源兼容性
- `-target <版本>` 生成特定 VM 版本的类文件
- `-version` 版本信息
- `-help` 输出标准选项的提要
- `-X` 输出非标准选项的提要
- `-J<标志>` 直接将 <标志> 传递给运行时系统

## (2)、jar

用法: `jar {ctxu}[vfm0Mi] [jar-文件] [manifest-文件] [-C 目录] 文件名 ...`

选项:

- `-c` 创建新的存档
- `-t` 列出存档内容的列表
- `-x` 展开存档中的命名的 (或所有的) 文件
- `-u` 更新已存在的存档



- v 生成详细输出到标准输出上
- f 指定存档文件名
- m 包含来自标明文件的标明信息
- O 只存储方式；未用 ZIP 压缩格式
- M 不产生所有项的清单（manifest）文件
- i 为指定的 jar 文件产生索引信息
- C 改变到指定的目录，并且包含下列文件：

如果一个文件名是一个目录，它将被递归处理。

清单（manifest）文件名和存档文件名都需要被指定，按 'm' 和 'f' 标志指定的相同顺序。

示例 1：将两个 class 文件存档到一个名为 'classes.jar' 的存档文件中：

```
jar cvf classes.jar Foo.class Bar.class
```

示例 2：用一个存在的清单（manifest）文件 'mymanifest' 将 foo/ 目录下的所有

文件存档到一个名为 'classes.jar' 的存档文件中：

```
jar cvfm classes.jar mymanifest -C foo/ .
```

### (3)、javadoc

javadoc：错误 - 未指定软件包或类。

用法：javadoc [选项] [软件包名称] [源文件] [@file]

- overview <文件> 读取 HTML 文件的概述文档
- public 仅显示公共类和成员
- protected 显示受保护/公共类和成员（默认）
- package 显示软件包/受保护/公共类和成员
- private 显示所有类和成员
- help 显示命令行选项并退出
- doclet <类> 通过替代 doclet 生成输出
- docletpath <路径> 指定查找 doclet 类文件的位置
- sourcepath <路径列表> 指定查找源文件的位置
- classpath <路径列表> 指定查找用户类文件的位置
- exclude <软件包列表> 指定要排除的软件包的列表

- subpackages <子软件包列表> 指定要递归装入的子软件包
- breakiterator 使用 **BreakIterator** 计算第 1 句
- bootclasspath <路径列表> 覆盖引导类加载器所装入的  
类文件的位置
- source <版本> 提供与指定版本的源兼容性
- extdirs <目录列表> 覆盖安装的扩展目录的位置
- verbose 输出有关 **Javadoc** 正在执行的操作的消息
- locale <名称> 要使用的语言环境, 例如 **en\_US** 或 **en\_US\_WIN**
- encoding <名称> 源文件编码名称
- quiet 不显示状态消息
- J<标志> 直接将 <标志> 传递给运行时系统

通过标准 **doclet** 提供:

- d <目录> 输出文件的目标目录
- use 创建类和软件包用法页面
- version 包含 **@version** 段
- author 包含 **@author** 段
- docfilessubdirs 递归复制文档文件子目录
- splitindex 将索引分为每个字母对应一个文件
- windowtitle <文本> 文档的浏览器窗口标题
- doctitle <html 代码> 包含概述页面的标题
- header <html 代码> 包含每个页面的页眉文本
- footer <html 代码> 包含每个页面的页脚文本
- bottom <html 代码> 包含每个页面的底部文本
- link <url> 创建指向位于 <url> 的 **javadoc** 输出的链接
- linkoffline <url> <url2> 利用位于 <url2> 的软件包列表链接至位于 <url>  
的文档
- excludedocfilessubdir <名称 1>:...排除带有给定名称的所有文档文件子目录。
- group <名称> <p1>:<p2>.. 在概述页面中, 将指定的软件包分组
- nocomment 抑止描述和标记, 只生成声明。

-nodeprecated	不包含 @deprecated 信息
-noqualifier <名称 1>:<名称 2>:...从输出中排除限定符的列表。	
-nosince	不包含 @since 信息
-notimestamp	不包含隐藏时间戳
-nodeprecatedlist	不生成已过时的列表
-notree	不生成类分层结构
-noindex	不生成索引
-nohelp	不生成帮助链接
-nonavbar	不生成导航栏
-serialwarn	生成有关 @serial 标记的警告
-tag <名称>:<位置>:<标题>	指定单个变量自定义标记
-taglet	要注册的 Taglet 的全限定名称
-tagletpath	Taglet 的路径
-charset <字符集>	用于跨平台查看生成的文档的字符集。
-helpfile <文件>	包含帮助链接所链接到的文件
-linksource	以 HTML 格式生成源
-sourcetab <制表符长度>	指定源中每个制表符占据的空格数
-keywords	使软件包、类和成员信息附带 HTML 元标记
-stylesheetfile <路径>	用于更改生成文档的样式的文件
-docencoding <名称>	输出编码名称

#### (4)、rmid

rmid: 非法选项: -?

用法: rmid <option>

其中, <option> 包括:

- port <option> 指定供 rmid 使用的端口
- log <directory> 指定 rmid 将日志写入的目录
- stop 停止当前的 rmid 调用 (对指定端口)
- C<runtime 标记> 向每个子进程传递参数 (激活组)

-J<runtime 标记> 向 java 解释程序传递参数

参考资料:

1. [JVM 的内存机制介绍](#)
2. [JVM 内存模型以及垃圾收集策略解析](#)
3. [Java 虚拟机简介](#)
4. [Java 虚拟机\(JVM\)参数配置说明](#)