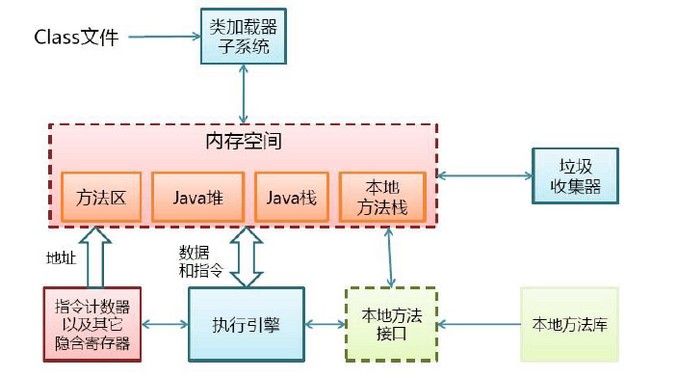
**JVM介绍及性能调优**

# 1、JVM介绍

JVM是Java Virtual Machine（Java虚拟机）的缩写，JVM是一种用于计算设备的规范，它是一个虚构出来的计算机，是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。

## 1.1 JVM内部结构

JVM的内部结构图如下：



下面来详细介绍下每部分结构：

**Class loader（类装载器子系统）：**负责查找并将Class文件装载到内存，最终形成可以被虚拟机直接使用的Java类型。

**Class Area（方法区）：**在类装载器将Class文件加载到内存的过程中，Java虚拟机会提取其中的类型信息，并将这些信息保存到方法区。方法区用于保存已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。由于所有线程都共享方法区，因此它们对方法去的访问线程应被设计成安全的。

**HEAP（堆）：**存储Java程序创建的类实例。所有线程共享，因此设计程序时也要考虑到多线程访问对象(堆数据)的同步问题。

**Stack（栈）：**Java栈是线程私有的。每当启动一个新线程时，Java虚拟机都会为它分配一个Java栈。Java栈以帧为单位保存线程的运行状态。虚拟机只会直接对Java栈执行两种操作：以帧为单位的压栈或出栈。当线程调用java方法时，虚拟机压入一个新的栈帧到该线程的java栈中。当方法返回时，这个栈帧被从java栈中弹出并抛弃。一个栈帧包含一个java方法的调用状态，它存储有局部变量表、操作栈、动态链接、方法出口等信息。栈内存随着线程生命周期结束而释放，意思就是线程一结束内存立即over。

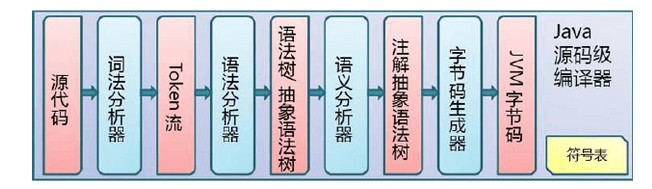
**PC Register（程序计数器/PC寄存器）：**一个运行中的Java程序，每当启动一个新线程时，都会为这个新线程创建一个自己的PC(程序计数器)寄存器。程序计数器的作用可以看做是当前线程所执行的字节码的行号指示器。字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。如果线程正在执行的是一个Java方法，这个计数器记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Natvie方法，这个计数器值则为空(Undefined)。

**Native Method Stack（本地方法栈）：**本地方法栈与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。任何本地方法接口都会使用某种本地方法栈。当线程调用Java方法时，虚拟机会创建一个新的栈帧并压入Java栈。然而当它调用的是本地方法时，虚拟机会保持Java栈不变，不再在线程的Java栈中压入新的帧，虚拟机只是简单地动态链接并直接调用指定的本地方法。如果某个虚拟机实现的本地方法接口是使用C连接模型的话，那么它的本地方法栈就是C栈。

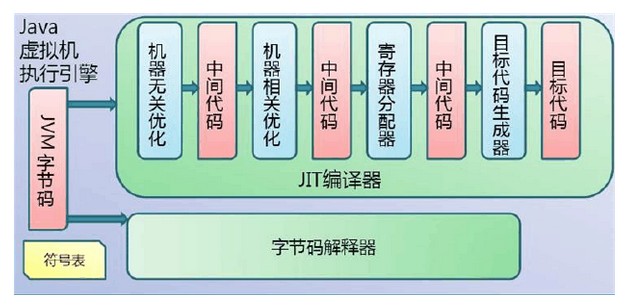
**Execution engine（执行引擎）：**负责执行字节码。方法的字节码是由Java虚拟机的指令序列构成的。每一条指令包含一个单字节的操作码，后面跟随0个或多个操作数。执行引擎执行字节码时，首先取得一个操作码，如果操作码有操作数，取得它的操作数。它执行操作码和跟随的操作数规定的动作，然后再取得下一个操作码。这个执行字节码的过程在线程完成前将一直持续。

## 1.2 JVM工作原理

Java代码编译是由Java源码编译器来完成，流程图如下所示：



Java字节码的执行是由JVM执行引擎来完成，流程图如下所示：

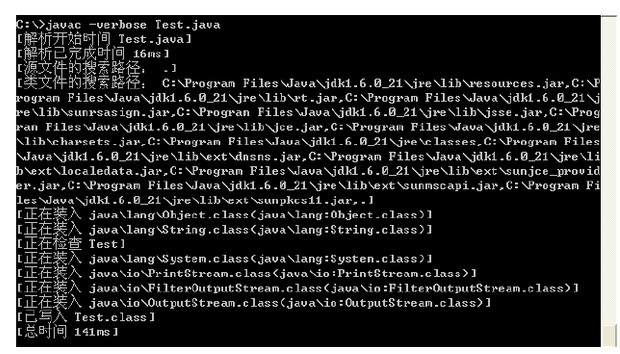


Java代码编译和执行的过程包括以下3个重要的机制：

* Java源代码编译机制
* 类加载机制
* 类执行机制

### 1.2.1 Java源代码编译机制

Java源码编译由以下3个过程组成：分析和输入到符号表、注释处理、语义分析和生成class文件。

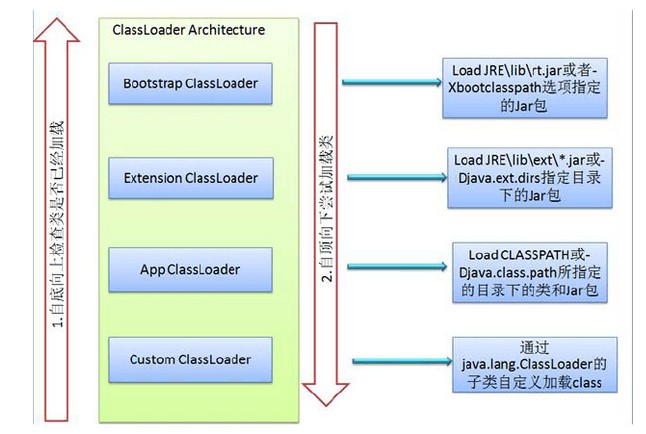


最后生成的class文件包括以下三个部分：

* 结构信息。包括class文件格式版本号及各部分的数量与大小的信息；
* 元数据。对应于Java源码中声明与常量的信息。包含类/继承的超类/实现的接口的声明信息、域与方法声明信息和常量池；
* 方法信息。对应Java源码中语句和表达式对应的信息。包含字节码、异常处理器表、求值栈与局部变量区大小、求值栈的类型记录、调试符号信息。

### 1.2.2 类加载机制

JVM的类加载是通过ClassLoader及其子类来完成的，类的层次关系和加载顺序可以由下图来描述：



* Bootstrap ClassLoader /启动类加载器：$JAVA\_HOME中jre/lib/rt.jar里所有的class，由C++实现，不是ClassLoader子类；
* Extension ClassLoader/扩展类加载器：负责加载java平台中扩展功能的一些jar包，包括$JAVA\_HOME中jre/lib/\*.jar或-Djava.ext.dirs指定目录下的jar包；
* App ClassLoader/ 系统类加载器：负责记载Classpath中指定的jar包及目录中class
* Custom ClassLoader/用户自定义类加载器(java.lang.ClassLoader的子类)：属于应用程序根据自身需要自定义的ClassLoader，如tomcat、jboss都会根据j2ee规范自行实现ClassLoader。

加载过程中会先检查类是否被已加载，检查顺序是自底向上，从Custom ClassLoader到BootStrap ClassLoader逐层检查，只要某个Classloader已加载就视为已加载此类，保证此类只所有ClassLoader加载一次。而加载的顺序是自顶向下，也就是由上层来逐层尝试加载此类。

### 1.2.3 类执行机制

JVM是基于栈的体系结构来执行class字节码的。线程创建后，都会产生程序计数器（PC）和栈（Stack），程序计数器存放下一条要执行的指令在方法内的偏移量，栈中存放一个个栈帧，每个栈帧对应着每个方法的每次调用，而栈帧又是有局部变量区和操作数栈两部分组成，局部变量区用于存放方法中的局部变量和参数，操作数栈中用于存放方法执行过程中产生的中间结果。

## 1.3 JVM、JDK、JRE的区别

**JDK(Java开发工具包)。**JDK是整个JAVA的核心，包括了Java运行环境（Java Runtime Envirnment），一堆Java工具（javac/java/jdb等）和Java基础的类库（即Java API 包括rt.jar）

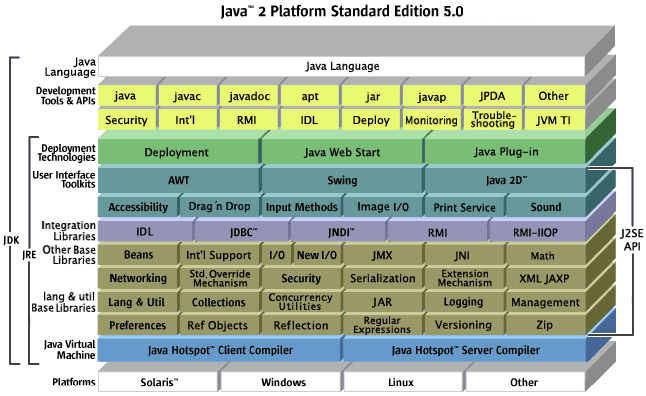
JDK有以下三种版本：J2SE（standard edition）标准版，是我们通常用的一个版本；J2EE（enterpsise edtion），使用这种JDK开发J2EE应用程序；J2ME（micro edtion）主要用于移动设备、嵌入式设备上的java应用程序

我们常常用JDK来代指Java API，Java API是Java的应用程序接口，其实就是前辈们写好的一些java Class，包括一些重要的语言结构以及基本图形，网络和文件I/O等等 ，我们在自己的程序中，调用前辈们写好的这些Class，来作为我们自己开发的一个基础。当然，现在已经有越来越多的性能更好或者功能更强大的第三方类库供我们使用。

**JRE(java运行时环境)。**也就是我们说的JAVA平台，所有的Java程序都要在JRE下才能运行。包括JVM和JAVA核心类库和支持文件。与JDK相比，它不包含开发工具——编译器、调试器和其它工具。

**JVM(JAVA虚拟机)。**JVM是JRE的一部分，它是一个虚构出来的计算机，是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。JVM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。JVM 的主要工作是解释自己的指令集（即字节码）并映射到本地的 CPU 的指令集或 OS 的系统调用。Java语言是跨平台运行的，其实就是不同的操作系统，使用不同的JVM映射规则，让其与操作系统无关，完成了跨平台性。JVM 对上层的 Java 源文件是不关心的，它关注的只是由源文件生成的类文件（ class file ）。类文件的组成包括 JVM 指令集，符号表以及一些补助信息。

下图很好的表明了JDK、JRE、JVM三者之间的关系：



我们利用JDK（调用JAVA API）开发了属于我们自己的JAVA程序后，通过JDK中的编译程序（Javac）将我们的文本java文件编译成JAVA字节码，在JRE上运行这些JAVA字节码，JVM解析这些字节码，映射到CPU指令集或OS的系统调用。

# 2、GC机制

## 2.1 JVM的内存管理机制

在程序运行过程当中，会创建大量的对象，这些对象，大部分是短周期的对象，小部分是长周期的对象，对于短周期的对象，需要频繁地进行垃圾回收以保证无用对象尽早被释放掉，对于长周期对象，则不需要频率垃圾回收以确保无谓地垃圾扫描检测。为解决这种矛盾，Sun JVM的内存管理采用分代的策略。

年轻代(Young Gen)：年轻代主要存放新创建的对象，内存大小相对会比较小，垃圾回收会比较频繁。年轻代分成1个Eden Space和2个Suvivor Space（命名为A和B）

* 当对象在堆创建时，将进入年轻代的Eden Space。
* 垃圾回收器进行垃圾回收时，扫描Eden Space和A Suvivor Space，如果对象仍然存活，则复制到B Suvivor Space，如果B Suvivor Space已经满，则复制 Old Gen
* 扫描A Suvivor Space时，如果对象已经经过了几次的扫描仍然存活，JVM认为其为一个Old对象，则将其移到Old Gen。
* 扫描完毕后，JVM将Eden Space和A Suvivor Space清空，然后交换A和B的角色（即下次垃圾回收时会扫描Eden Space和BSuvivor Space）。

我们可以看到：Young Gen垃圾回收时，采用将存活对象复制到空的Suvivor Space的方式来确保不存在内存碎片，采用空间换时间的方式来加速内存垃圾回收。针对年轻代的垃圾回收即Young GC。

年老代(Tenured Gen)：年老代主要存放JVM认为比较old的对象（经过几次的Young Gen的垃圾回收后仍然存在），内存大小相对会比较大，垃圾回收也相对没有那么频繁（譬如可能几个小时一次）。年老代主要采用压缩的方式来避免内存碎片（将存活对象移动到内存片的一边），当然，有些垃圾回收器（譬如CMS垃圾回收器）出于效率的原因，可能会不进行压缩。针对年老代的垃圾回收即Full GC。

持久代(Perm Gen)：持久代主要存放类定义、字节码和常量等很少会变更的信息。

## 2.2 JVM的内存回收机制

在介绍回收机制之前，先看几个问题：

* 首先哪些对象需要被回收？
* 谁来回收？
* 什么时候回收？
* 在哪里回收？
* 怎么回收？

①对于要回收的对象，系统会对其作出标记

GC算法:

* 引用计数算法：引用计数是一种简单但速度很慢的垃圾回收计数。每个对象都含有一个引用计数器，当有引用链接至对象时，引用计数加1，当引用离开作用于或者被设置为null时，引用计数减1。缺陷：如果兑现之间存在循环引用，就会出现麻烦。
* 标记-清除算法：算法分成标记和清除两个阶段。首先标记出所有需要回收的对象，然后回收所有需要回收的对象。缺点：标记和清理两个过程效率不高；标记清理之后会产生大量不连续的内存碎片，空间碎片太多可能会导致后续使用中无法找到足够的连续内存而提前触发另一次的垃圾回收操作。
* 复制收集算法：解决效率而产生的算法。将内存分成两块，每次只使用其中的一块，当半区内存用完时，仅将存活的对象复制到另一块内存中，然后清理掉原先的内存。这样使得每次内存回收都是对整个半区的回收，内存分配时也不用考虑内存碎片等复杂情况，实现简单，运行高效。缺点则是内存缩减为一半。
* 分代收集算法：根据对象不同的存货周期将内存分为几块。然后根据各个年代的特点采用最适合的收集算法，譬如年轻代的每次GC都会有大批的对象死去，这样采用复制算法则只需要付出少量存活对象的复制就可以完成收集操作。

②JVM垃圾回收器具体解决回收的问题。

③对于回收的地方一般在堆上完成，栈上也可以进行回收不过条件比较苛刻。

④在内存不够用的时候，JVM开始回收。

⑤垃圾回收基本的思想是：stop-the-world再回收，就如：不能在打扫卫生的时候同时再仍新的垃圾。但是CMS、与将出世的 G1会打破这个界限，实现 边打扫卫生，还边扔垃圾。

怎么回收：一般有三个基本的算法：

* 标记清除：直接标记，再清除掉需要回收的内存。（产生大量的内存碎片）
* 标记复制：用两个一样的大小的内存，总只有一块再用，回收时没有回收的部分直接复制到另一块上去。
* 标记整理：先标记，让所有存活的对象向一段移动，然后直接清除掉边界外的内存。

评估垃圾回收策略的两个重要度量是：**吞吐量和暂停时间**。

* 吞吐量(Throughput )：JVM花费在垃圾回收上的时间越长，则吞吐量越低。
* 暂停时间(Pause time)：JVM垃圾回收过程当中有一个暂停期，在暂停期间，应用程序不能运行，暂停时间是暂停期的长度。

非常遗憾的是，一般这两个指标是相互冲突的，改善其中一个会影响到另外一个，根据情景的不同我们决定是优先考虑吞吐量还是暂停时间，对于需要实时响应的应用，我们需要优先考虑暂停时间，对于后台运行应用，我们需要优先考虑吞吐量。

内存回收有以下几种策略：

* 并行(Parallel)：并行表示使用多个线程同时进行垃圾回收的工作，此策略一般会从同时改善暂停时间和吞吐量，在有多CPU内核的服务器上，这是基本上我们要使用的策略。吞吐量优先
* 并发(Concurrent)：并行表示垃圾回收器的一些工作（譬如垃圾标记）与应用程序同时进行，这将更进一步缩短暂停时间，需要注意的是，同时垃圾回收器的复杂性会大大增大，基本上是会降低吞吐量，响应时间优先。
* 内存碎片处理：有不压缩、压缩和拷贝三种策略，从空间上讲，拷贝将花费更多的内存（譬如如上内存管理的Young Gen，需要维持一个额外的Suvivor空间），从时间上来讲，不压缩会减低创建对象时的内存分配效率，在垃圾回收上，拷贝策略会比压缩策略更高效。

## 2.3 垃圾回收器选择

JVM给出了4种选择：序列垃圾回收器、并发垃圾回收器、并行压缩垃圾回收器、并发标志清除垃圾回收器。序列垃圾回收器只适用于小数据量的情况，所以生产环境的选择主要是并行收集器和并发收集器。

### 2.3.1 串行垃圾回收器

Serial Collector：串行垃圾回收器，垃圾回收器对Young Gen和Tenured Gen都是使用单线的垃圾回收方式，对Young Gen，会使用拷贝策略避免内存碎片，对Old Gen，会使用压缩策略避免内存碎片。基本上，在对内核的服务器上应该避免使用这种方式。

在JVM启动参数中使用-XX:+UseSerialGC启用Serial Collector。

新生代、老年代使用串行回收；新生代使用复制算法，老年代使用标记-压缩算法。

### 2.3.2 并发垃圾回收器

Parallel Collector：并发垃圾回收器，垃圾回收器对Young Gen和Tenured Gen都是使用多线程并行垃圾回收的方式，对Young Gen，会使用拷贝策略避免内存碎片，对Old Gen，会使用压缩策略避免内存碎片。

在JVM启动参数中使用-XX:+UseParallelGC启用Parallel Collector。

新生代并行，老年代串行。复制算法，多线程需要多核支持。-XX:ParallelGCThreads限制线程数量。

### 2.3.3 并行压缩垃圾回收器（吞吐量优先）

Parallel Compacting Collector：并行压缩垃圾回收器，与Parallel Collector垃圾回收类似，但对Tenured Gen会使用一种更有效的垃圾回收策略，此垃圾回收器在暂停时间上会更短。

在JVM启动参数中使用-XX:+UseParallelOldGC启用Parallel Compacting Collector。

* -XX:+UseParallelGC：设置为并行收集器。此配置仅对年轻代有效。即年轻代使用并行收集，而年老代仍使用串行收集。
* -XX:ParallelGCThreads=20：配置并行收集器的线程数，即：同时有多少个线程一起进行垃圾回收。此值建议配置与CPU数目相等。
* -XX:+UseParallelOldGC：配置年老代垃圾收集方式为并行收集。JDK6.0开始支持对年老代并行收集。
* -XX:MaxGCPauseMillis=100：设置每次年轻代垃圾回收的最长时间（单位毫秒）。如果无法满足此时间，JVM会自动调整年轻代大小，以满足此时间。
* -XX:+UseAdaptiveSizePolicy：设置此选项后，并行收集器会自动调整年轻代Eden区大小和Survivor区大小的比例，以达成目标系统规定的最低响应时间或者收集频率等指标。此参数建议在使用并行收集器时，一直打开。

### 2.3.4 并发标志清除垃圾回收器（响应时间优先）

Concurrent Mark-Sweep (CMS) Collector：并发标志清除垃圾回收器，对Young Gen会使用与Parallel Collector同样的垃圾回收策略，对Tenured Gen，垃圾回收的垃圾标志线程与应用线程同时进行，而垃圾清除则需要暂停应用线程，但暂停时间会大大缩减，需要注意的是，由于垃圾回收过程更加复杂，会降低总体的吞吐量，与用户线程一起执行。

* -XX:+UseConcMarkSweepGC：即CMS收集，设置年老代为并发收集。CMS收集是JDK1.4后期版本开始引入的新GC算法。它的主要适合场景是对响应时间的重要性需求大于对吞吐量的需求，能够承受垃圾回收线程和应用线程共享CPU资源，并且应用中存在比较多的长生命周期对象。CMS收集的目标是尽量减少应用的暂停时间，减少Full GC发生的几率，利用和应用程序线程并发的垃圾回收线程来标记清除年老代内存。
* -XX:+UseParNewGC：设置年轻代为并发收集。可与CMS收集同时使用。JDK5.0以上，JVM会根据系统配置自行设置，所以无需再设置此参数。
* -XX:CMSFullGCsBeforeCompaction=0：由于并发收集器不对内存空间进行压缩和整理，所以运行一段时间并行收集以后会产生内存碎片，内存使用效率降低。此参数设置运行0次Full GC后对内存空间进行压缩和整理，即每次Full GC后立刻开始压缩和整理内存。
* -XX:+UseCMSCompactAtFullCollection：打开内存空间的压缩和整理，在Full GC后执行。可能会影响性能，但可以消除内存碎片。
* -XX:+CMSIncrementalMode：设置为增量收集模式。一般适用于单CPU情况。
* -XX:CMSInitiatingOccupancyFraction=70：表示年老代内存空间使用到70%时就开始执行CMS收集，以确保年老代有足够的空间接纳来自年轻代的对象，避免Full GC的发生。

## 2.4 其他辅助信息参数

* -XX:-CITime：打印消耗在JIT编译的时间。
* -XX:ErrorFile=./hs\_err\_pid.log：保存错误日志或数据到指定文件中。
* -XX:HeapDumpPath=./java\_pid.hprof：指定Dump堆内存时的路径。
* -XX:-HeapDumpOnOutOfMemoryError：当首次遭遇内存溢出时Dump出此时的堆内存。
* -XX:OnError=";"：出现致命ERROR后运行自定义命令。
* -XX:OnOutOfMemoryError=";"：当首次遭遇内存溢出时执行自定义命令。
* -XX:-PrintClassHistogram：按下 Ctrl+Break 后打印堆内存中类实例的柱状信息，同JDK的 jmap -histo 命令。
* -XX:-PrintConcurrentLocks：按下 Ctrl+Break 后打印线程栈中并发锁的相关信息，同JDK的 jstack -l 命令。
* -XX:-PrintCompilation：当一个方法被编译时打印相关信息。
* -XX:-PrintGC：每次GC时打印相关信息。
* -XX:-PrintGCDetails：每次GC时打印详细信息。
* -XX:-PrintGCTimeStamps：打印每次GC的时间戳。
* -XX:-TraceClassLoading：跟踪类的加载信息。
* -XX:-TraceClassLoadingPreorder：跟踪被引用到的所有类的加载信息。
* -XX:-TraceClassResolution：跟踪常量池。
* -XX:-TraceClassUnloading：跟踪类的卸载信息

# 3、JVM性能分析

**-Xmn：**新生代内存大小的的最大值，包括E区和两个S区的总和。使用方法：-Xmn65535，-Xmn1024k，-Xmn512m，-Xmn1g (-Xms，-Xmx也是种写法)

**-Xms：**初始堆的大小，也是堆大小的最小值，默认值是总共的物理内存/64（且小于1G），默认情况下，当堆中可用内存小于40%(这个值可以用-XX: MinHeapFreeRatio 调整，如-X:MinHeapFreeRatio=30)时，堆内存会开始增加，一直增加到-Xmx的大小；

**-Xmx：**堆的最大值，默认值是总共的物理内存/64（且小于1G），如果Xms和Xmx都不设置，则两者大小会相同，默认情况下，当堆中可用内存大于70%（这个值可以用-XX: MaxHeapFreeRatio 调整，如-X:MaxHeapFreeRatio=60）时，堆内存会开始减少，一直减小到-Xms的大小；整个堆的大小=年轻代大小+年老代大小，堆的大小不包含持久代大小，如果增大了年轻代，年老代相应就会减小，官方默认的配置为年老代大小/年轻代大小=2/1左右（使用-XX:NewRatio可以设置-XX:NewRatio=5，表示年老代/年轻代=5/1）；

建议在开发测试环境可以用Xms和Xmx分别设置最小值最大值，但是在线上生产环境，Xms和Xmx设置的值必须一样，原因与年轻代一样——防止抖动；

**-Xss：**这个参数用于设置每个线程的栈内存，默认1M，一般来说是不需要改的。除非代码不多，可以设置的小点，另外一个相似的参数是-XX:ThreadStackSize，这两个参数在1.6以前，都是谁设置在后面，谁就生效；1.6版本以后，-Xss设置在后面，则以-Xss为准，-XXThreadStackSize设置在后面，则主线程以-Xss为准，其它线程以-XX:ThreadStackSize为准。

**实例1：**

发现部分开发测试机器出现异常：java.lang.OutOfMemoryError: GC overhead limit exceeded，这个异常代表：GC为了释放很小的空间却耗费了太多的时间，其原因一般有两个：1，堆太小，2，有死循环或大对象；  
笔者首先排除了第2个原因，因为这个应用同时是在线上运行的，如果有问题，早就挂了。所以怀疑是这台机器中堆设置太小；  
使用ps -ef |grep "java"查看，发现：

http://images.cnitblog.com/blog/406312/201312/31180232-b35faf9a89dd48a184de58bc3a7e26ca.png  
该应用的堆区设置只有768m，而机器内存有2g，机器上只跑这一个java应用，没有其他需要占用内存的地方。另外，这个应用比较大，需要占用的内存也比较多；  
笔者通过上面的情况判断，只需要改变堆中各区域的大小设置即可，于是改成下面的情况：

http://images.cnitblog.com/blog/406312/201312/31180253-523e7c956672437cb0203cae27232059.png  
跟踪运行情况发现，相关异常没有再出现；

**实例2：**

一个服务系统，经常出现卡顿，分析原因，发现Full GC时间太长：  
jstat -gcutil:  
S0     S1    E     O       P        YGC YGCT FGC FGCT  GCT  
12.16 0.00 5.18 63.78 20.32  54   2.047 5     6.946  8.993   
分析上面的数据，发现Young GC执行了54次，耗时2.047秒，每次Young GC耗时37ms，在正常范围，而Full GC执行了5次，耗时6.946秒，每次平均1.389s，数据显示出来的问题是：Full GC耗时较长，分析该系统的是指发现，NewRatio=9，也就是说，新生代和老生代大小之比为1:9，这就是问题的原因：

1，新生代太小，导致对象提前进入老年代，触发老年代发生Full GC；

2，老年代较大，进行Full GC时耗时较大；

优化的方法是调整NewRatio的值，调整到4，发现Full GC没有再发生，只有Young GC在执行。这就是把对象控制在新生代就清理掉，没有进入老年代（这种做法对一些应用是很有用的，但并不是对所有应用都要这么做）

# 4、Jprofiler介绍

# 5、调优实例

## 5.1 JVM调优实例

**一：JVM调优之串行垃圾回收**

也就是默认配置，完成10万request用时153秒。JVM参数配置如下：

1. $JAVA\_ARGS.="-Dresin.home=$SERVER\_ROOT-server
2. -Xms2048M-Xmx2048M-Xmn512M
3. -XX:PermSize=256M-XX:MaxPermSize=256M
4. -XX:MaxTenuringThreshold=7-XX:GCTimeRatio=19
5. -Xnoclassgc-Xloggc:log/gc.log
6. -XX:+PrintGCDetails-XX:+PrintGCTimeStamps";

这种配置一般在resin启动24小时内似乎没有大问题，网站可以正常访问，但查看日志发现，在接近24小时时，FullGC执行越来越频繁，大约每隔3分钟就有一次FullGC，每次FullGC系统会停顿6秒左右，作为一个网站来说，用户等待6秒恐怕太长了，所以这种方式有待改善。MaxTenuringThreshold=7表示一个对象如果在救助空间移动7次还没有被回收就放入年老代，GCTimeRatio=19表示java可以用5%的时间来做垃圾回收，1/(1+19)=1/20=5%。

**二：JVM调优之并行回收**

完成10万request用时117秒，配置如下：

1. $JAVA\_ARGS.="-Dresin.home=$SERVER\_ROOT-server-Xmx2048M
2. -Xms2048M-Xmn512M-XX:PermSize=256M-XX:MaxPermSize=256M
3. -Xnoclassgc-Xloggc:log/gc.log-XX:+PrintGCDetails
4. -XX:+PrintGCTimeStamps-XX:+UseParallelGC-XX:ParallelGCThreads=20
5. -XX:+UseParallelOldGC-XX:MaxGCPauseMillis=500
6. -XX:+UseAdaptiveSizePolicy-XX:MaxTenuringThreshold=7
7. -XX:GCTimeRatio=19";

并行回收我尝试过多种组合配置，似乎都没什么用，resin启动3小时左右就会停顿，时间超过10秒。也有可能是参数设置不够好的原因，MaxGCPauseMillis表示GC最大停顿时间，在resin刚启动还没有执行FullGC时系统是正常的，但一旦执行FullGC，MaxGCPauseMillis根本没有用，停顿时间可能超过20秒，之后会发生什么我也不再关心了，赶紧重启resin，尝试其他回收策略。

**三：JVM调优之并发回收**

完成10万request用时60秒，比并行回收差不多快一倍，是默认回收策略性能的2.5倍，配置如下：

1. $JAVA\_ARGS.="-Dresin.home=$SERVER\_ROOT-server
2. -Xms2048M-Xmx2048M-Xmn512M-XX:PermSize=256M
3. -XX:MaxPermSize=256M-XX:+UseConcMarkSweepGC
4. -XX:MaxTenuringThreshold=7-XX:GCTimeRatio=19
5. -Xnoclassgc-Xloggc:log/gc.log-XX:+PrintGCDetails
6. -XX:+PrintGCTimeStamps-XX:+UseCMSCompactAtFullCollection
7. -XX:CMSFullGCsBeforeCompaction=0";

这个配置虽然不会出现10秒连不上的情况，但系统重启3个小时左右，每隔几分钟就会有5秒连不上的情况，查看gc.log，发现在执行ParNewGC时有个promotionfailed错误，从而转向执行FullGC，造成系统停顿，而且会很频繁，每隔几分钟就有一次，所以还得改善。UseCMSCompactAtFullCollection是表是执行FullGC后对内存进行整理压缩，免得产生内存碎片，CMSFullGCsBeforeCompaction=N表示执行N次FullGC后执行内存压缩。

**四：JVM调优之增量回收**

完成10万request用时171秒，太慢了，配置如下：

1. $JAVA\_ARGS.="-Dresin.home=$SERVER\_ROOT-server
2. -Xms2048M-Xmx2048M-Xmn512M-XX:PermSize=256M
3. -XX:MaxPermSize=256M-XX:MaxTenuringThreshold=7
4. -XX:GCTimeRatio=19-Xnoclassgc-Xloggc:log/gc.log
5. -XX:+PrintGCDetails-XX:+PrintGCTimeStamps-Xincgc";

似乎回收得也不太干净，而且也对性能有较大影响，不值得试。

**五：JVM调优之并发回收的I-CMS模式**

和增量回收差不多，完成10万request用时170秒。配置如下：

1. $JAVA\_ARGS.="-Dresin.home=$SERVER\_ROOT-server
2. -Xms2048M-Xmx2048M-Xmn512M-XX:PermSize=256M
3. -XX:MaxPermSize=256M-XX:MaxTenuringThreshold=7
4. -XX:GCTimeRatio=19-Xnoclassgc-Xloggc:log/gc.log
5. -XX:+PrintGCDetails-XX:+PrintGCTimeStamps
6. -XX:+UseConcMarkSweepGC-XX:+CMSIncrementalMode
7. -XX:+CMSIncrementalPacing
8. -XX:CMSIncrementalDutyCycleMin=0
9. -XX:CMSIncrementalDutyCycle=10-XX:-TraceClassUnloading";

采用了sun推荐的参数，回收效果不好，照样有停顿，数小时之内就会频繁出现停顿，什么sun推荐的参数，照样不好使。

**六：JVM调优之递增式低暂停收集器**

又叫什么火车式回收，完成10万request用时153秒，配置如下：

1. $JAVA\_ARGS.="-Dresin.home=$SERVER\_ROOT-server
2. -Xms2048M-Xmx2048M-Xmn512M-XX:PermSize=256M
3. -XX:MaxPermSize=256M-XX:MaxTenuringThreshold=7
4. -XX:GCTimeRatio=19-Xnoclassgc-Xloggc:log/gc.log
5. -XX:+PrintGCDetails-XX:+PrintGCTimeStamps-XX:+UseTrainGC";

该配置效果也不好，影响性能，所以没试。

七：相比之下，还是并发回收比较好，性能比较高，只要能解决ParNewGC（并行回收年轻代）时的promotionfailed错误就一切好办了，查了很多文章，发现引起promotionfailed错误的原因是CMS来不及回收（CMS默认在年老代占到90%左右才会执行），年老代又没有足够的空间供GC把一些活的对象从年轻代移到年老代，所以执行FullGC。CMSInitiatingOccupancyFraction=70表示年老代占到约70%时就开始执行CMS，这样就不会出现FullGC了。SoftRefLRUPolicyMSPerMB这个参数也是我认为比较有用的，官方解释是softlyreachableobjectswillremainaliveforsomeamountoftimeafterthelasttime  
theywerereferenced.Thedefaultvalueisonesecondo  
flifetimeperfreemegabyteintheheap，我觉得没必要等1秒，所以设置成0。配置如下

1. $JAVA\_ARGS.="-Dresin.home=$SERVER\_ROOT-server-Xms2048M
2. -Xmx2048M-Xmn512M-XX:PermSize=256M-XX:MaxPermSize=256M
3. -XX:SurvivorRatio=8-XX:MaxTenuringThreshold=7
4. -XX:GCTimeRatio=19-Xnoclassgc-XX:+DisableExplicitGC
5. -XX:+UseParNewGC-XX:+UseConcMarkSweepGC
6. -XX:+CMSPermGenSweepingEnabled
7. -XX:+UseCMSCompactAtFullCollection
8. -XX:CMSFullGCsBeforeCompaction=0
9. -XX:+CMSClassUnloadingEnabled-XX:-CMSParallelRemarkEnabled
10. -XX:CMSInitiatingOccupancyFraction=70
11. -XX:SoftRefLRUPolicyMSPerMB=0-XX:+PrintClassHistogram
12. -XX:+PrintGCDetails-XX:+PrintGCTimeStamps
13. -XX:+PrintGCApplicationConcurrentTime
14. -XX:+PrintGCApplicationStoppedTime
15. -Xloggc:log/gc.log";

上面这个配置内存上升的很慢，24小时之内几乎没有停顿现象，最长的只停滞了0.8s，ParNewGC每30秒左右才执行一次，每次回收约0.2秒，看来问题应该暂时解决了。

参数不明白的可以上网查，本人认为比较重要的几个参数是：

-Xms-Xmx-XmnMaxTenuringThresholdGCTimeRatioUse

ConcMarkSweepGCCMSInitiatingOccupancyFractionSoftRefLRUPolicyMSPerMB

eclipse中配置JVM参数:-Xmx1024M-Xms1000M-server-XX:PermSize=64M-XX:MaxPermSize=128m

# 6、常见问题

一、JVM内存溢出现象提示一：java.lang.OutOfMemoryError:Javaheapspace

解释：

**Heapsize设置**

JVM堆的设置是指java程序运行过程中JVM可以调配使用的内存空间的设置.JVM在启动的时候会自动设置Heapsize的值，其初始空间(即-Xms)是物理内存的1/64，最大空间(-Xmx)是物理内存的1/4。可以利用JVM提供的-Xmn-Xms-Xmx等选项可进行设置。Heapsize的大小是YoungGeneration和TenuredGeneraion之和。

提示：在JVM中如果98％的时间是用于GC且可用的Heapsize不足2％的时候将抛出此异常信息。

提示：HeapSize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。

解决方法：

**手动设置Heapsize**

修改TOMCAT\_HOME/bin/catalina.bat，在“echo"UsingCATALINA\_BASE:$CATALINA\_BASE"”上面加入以下行：  
Java代码

1. setJAVA\_OPTS=%JAVA\_OPTS%-server
2. -Xms800m-Xmx800m-XX:MaxNewSize=256m
3. setJAVA\_OPTS=%JAVA\_OPTS%-server
4. -Xms800m-Xmx800m-XX:MaxNewSize=256m

或修改catalina.sh

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

1. JAVA\_OPTS="$JAVA\_OPTS-server-Xms800m
2. -Xmx800m-XX:MaxNewSize=256m"

二、JVM内存溢出现象提示二：java.lang.OutOfMemoryError:PermGenspace

原因：  
PermGenspace的全称是PermanentGenerationspace,是指内存的永久保存区域，这块内存主要是被JVM存放Class和Meta信息的,Class在被Loader时就会被放到PermGenspace中，它和存放类实例(Instance)的Heap区域不同,GC(GarbageCollection)不会在主程序运行期对PermGenspace进行清理，所以如果你的应用中有很CLASS的话,就很可能出现PermGenspace错误，这种错误常见在web服务器对JSP进行precompile的时候。如果你的WEBAPP下都用了大量的第三方jar,其大小超过了jvm默认的大小(4M)那么就会产生此错误信息了。

解决方法：

**1.手动设置MaxPermSize大小**

修改TOMCAT\_HOME/bin/catalina.bat（Linux下为catalina.sh），在Java代码  
“echo"UsingCATALINA\_BASE:$CATALINA\_BASE"”上面加入以下行：

1. setJAVA\_OPTS=%JAVA\_OPTS%-server
2. -XX:PermSize=128M-XX:MaxPermSize=512m

“echo"UsingCATALINA\_BASE:$CATALINA\_BASE"”上面加入以下行：

1. setJAVA\_OPTS=%JAVA\_OPTS%-server
2. -XX:PermSize=128M-XX:MaxPermSize=512m

catalina.sh下为：

Java代码

1. JAVA\_OPTS="$JAVA\_OPTS-server
2. -XX:PermSize=128M-XX:MaxPermSize=512m"
3. JAVA\_OPTS="$JAVA\_OPTS-server
4. -XX:PermSize=128M-XX:MaxPermSize=512m"

