JVM

jenny

March 4, 2014

Contents

1 Java编译原理

Java 虚拟机(JVM)是可运行Java 代码的假想计算机。只要根据JVM规格描述将解释器移植到特定的计算机上，就能保证经过编译的任何Java代码能够在该系统上运行。本文首先简要介绍从Java文件的编译到最终执行的过程，随后对JVM规格描述作一说明。

1.1 Java源文件的编译、下载 、解释和执行

Java应用程序的开发周期包括编译、下载 、解释和执行几个部分。Java编译程序将Java源程序翻译为JVM可执行代码?字节码。这一编译过程同C/C++ 的编译有些不同。当C编译器编译生成一个对象的代码时，该代码是为在某一特定硬件平台运行而产生的。因此，在编译过程中，编译程序通过查表将所有对符号的引用转换为特定的内存偏移量，以保证程序运行。Java编译器却不将对变量和方法的引用编译为数值引用，也不确定程序执行过程中的内存布局，而是将这些符号引用信息保留在字节码中，由解释器在运行过程中创立内存布局，然后再通过查表来确定一个方法所在的地址。这样就有效的保证了Java的可移植性和安全性。

运行JVM字节码的工作是由解释器来完成的。解释执行过程分三部进行：代码的装入、代码的校验和代码的执行。装入代码的工作由"类装载器"（class loader）完成。类装载器负责装入运行一个程序需要的所有代码，这也包括程序代码中的类所继承的类和被其调用的类。当类装载器装入一个类时，该类被放在自己的名字空间中。除了通过符号引用自己名字空间以外的类，类之间没有其他办法可以影响其他类。在本台计算机上的所有类都在同一地址空间内，而所有从外部引进的类，都有一个自己独立的名字空间。这使得本地类通过共享相同的名字空间获得较高的运行效率，同时又保证它们与从外部引进的类不会相互影响。当装入了运行程序需要的所有类后，解释器便可确定整个可执行程序的内存布局。解释器为符号引用同特定的地址空间建立对应关系及查询表。通过在这一阶段确定代码的内存布局，Java很好地解决了由超类改变而使子类崩溃的问题，同时也防止了代码对地址的非法访问。 随后，被装入的代码由字节码校验器进行检查。校验器可发现操作数栈溢出，非法数据类型转化等多种错误。通过校验后，代码便开始执行了。

Java字节码的执行有两种方式:

1.即时编译方式：解释器先将字节码编译成机器码，然后再执行该机器码。

2.解释执行方式：解释器通过每次解释并执行一小段代码来完成Java字节码程 序的所有操作。

通常采用的是第二种方法。由于JVM规格描述具有足够的灵活性，这使得将字节码翻译为机器代码的工作具有较高的效率。对于那些对运行速度要求较高的应用程序，解释器可将Java字节码即时编译为机器码，从而很好地保证了Java代码的可移植性和高性能。

1.2 JVM规格描述

JVM的设计目标是提供一个基于抽象规格描述的计算机模型，为解释程序开发人员提很好的灵活性，同时也确保Java代码可在符合该规范的任何系统上运行。JVM对其实现的某些方面给出了具体的定义，特别是对Java可执行代码，即字节码(Bytecode)的格式给出了明确的规格。这一规格包括操作码和操作数的语法和数值、标识符的数值表示方式、以及Java类文件中的Java对象、常量缓冲池在JVM的存储 映象。这些定义为JVM解释器开发人员提供了所需的信息和开发环境。Java的设计者希望给开发人员以随心所欲使用Java的自由。

JVM定义了控制Java代码解释执行和具体实现的五种规格，它们是:

JVM指令系统

JVM寄存器

JVM栈结构

JVM碎片回收堆

JVM存储区

1.2.1 JVM指令系统

JVM指令系统同其他计算机的指令系统极其相似。Java指令也是由 操作码和操作数两部分组成。操作码为8位二进制数，操作数进紧随在操作码的后面，其长度根据需要而不同。操作码用于指定一条指令操作的性质（在这里我们采用汇编符号的形式进行说明），如iload表示从存储器中装入一个整数，anewarray表示为一个新数组分配空间，iand表示两个整数的"与"、ret用于流程控制，表示从对某一方法的调用中返回。当长度大于8位时，操作数被分为两个以上字节存放。JVM采用

了”big endian”的编码方式来处理这种情况，即高位bits存放在低字节中。这同 Motorola及其他的RISC CPU采用的编码方式是一致的，而与Intel采用的”little endian ”的编码方式即低位bits存放在低位字节的方法不同。

Java指令系统是以Java语言的实现为目的设计的，其中包含了用于调用方法和监视多先程系统的指令。Java的8位操作码的长度使得JVM最多有256种指令，目前已使用了160多种操作码。

1.2.2 JVM寄存器

所有的CPU均包含用于保存系统状态和处理器所需信息的寄存器组。如果虚拟机定义较多的寄存器，便可以从中得到更多的信息而不必对栈或内存进行访问，这有利于提高运行速度。然而，如果虚拟机中的寄存器比实际CPU的寄存器多，在实现虚拟机时就会占用处理器大量的时间来用常规存储器模拟寄存器，这反而会降低虚拟机的效率。针对这种情况，JVM只设置了4个最为常用的寄存器。它们是:

pc程序计数器
optop操作数栈顶指针
frame当前执行环境指针
vars指向当前执行环境中第一个局部变量的指针
所有寄存器均为32位。pc用于记录程序的执行。optop,frame和vars用于记录指向Java栈区的指针。

1.2.3 JVM栈结构

作为基于栈结构的计算机，Java栈是JVM存储信息的主要方法。当JVM得到一个Java字节码应用程序后，便为该代码中一个类的每一个方法创建一个栈框架，以保存该方法的状态信息。每个栈框架包括以下三类信息:

局部变量
执行环境
操作数栈

局部变量用于存储一个类的方法中所用到的局部变量。vars寄存器指向该变量表中的第一个局部变量。

执行环境用于保存解释器对Java字节码进行解释过程中所需的信息。它们是：上次调用的方法、局部变量指针和操作数栈的栈顶和栈底指针。执行环境是一个执行一个方法的控制中心。例如：如果解释器要执行iadd(整数加法)，首先要从frame寄存器中找到当前执行环境，而后便从执行环境中找到操作数栈，从栈顶弹出两个整数进行加法运算，最后将结果压入栈顶。

操作数栈用于存储运算所需操作数及运算的结果。

1.2.4 JVM碎片回收堆

Java类的实例所需的存储空间是在堆上分配的。解释器具体承担为类实例分配空间的工作。解释器在为一个实例分配完存储空间后，便开始记录对该实例所占用的内存区域的使用。一旦对象使用完毕，便将其回收到堆中。

在Java语言中，除了new语句外没有其他方法为一对象申请和释放内存。对内存进行释放和回收的工作是由Java运行系统承担的。这允许Java运行系统的设计者自己决定碎片回收的方法。在SUN公司开发的Java解释器和Hot Java环境中，碎片回收用后台线程的方式来执行。这不但为运行系统提供了良好的性能，而且使程序设计人员摆脱了自己控制内存使用的风险。

1.2.5 JVM存储区

JVM有两类存储区：常量缓冲池和方法区。常量缓冲池用于存储类名称、方法和字段名称以及串常量。方法区则用于存储Java方法的字节码。对于这两种存储区域具体实现方式在JVM规格中没有明确规定。这使得Java应用程序的存储布局必须在运行过程中确定，依赖于具体平台的实现方式。

JVM是为Java字节码定义的一种独立于具体平台的规格描述，是Java平台独立性的基础。目前的JVM还存在一些限制和不足，有待于进一步的完善，但无论如何，JVM的思想是成功的。

对比分析：如果把Java原程序想象成我们的C++ 原程序，Java原程序编译后生成的字节码就相当于C++原程序编译后的80x86的机器码（二进制程序文件），JVM虚拟机相当于80x86计算机系统,Java解释器相当于80x86CPU。在80x86CPU上运行的是机器码，在Java解释器上运行的是Java字节码。

Java解释器相当于运行Java字节码的“CPU”,但该“CPU”不是通过硬件实现的，而是用软件实现的。Java解释器实际上就是特定的平台下的一个应用程序。只要实现了特定平台下的解释器程序，Java字节码就能通过解释器程序在该平台下运行，这是Java跨平台的根本。当前，并不是在所有的平台下都有相应Java解释器程序，这也是Java并不能在所有的平台下都能运行的原因，它只能在已实现了Java解释器程序的平台下运行。

2 Java和JVM运行原理揭秘

这里和大家简单分享一下JAVA和JVM运行的原理，Java语言写的源程序通过Java编译器，编译成与平台无关的‘字节码程序’(.class文件，也就是0，1二进制程序)，然后在OS之上的Java解释器中解释执行，而JVM是java的核心和基础，在java编译器和os平台之间的虚拟处理器

2.1 Java语言运行的过程

Java语言写的源程序通过Java编译器，编译成与平台无关的‘字节码程序’(.class文件，也就是0，1二进制程序)，然后在OS之上的Java解释器中解释执行。

./f0.gif

也相当与

./f1.gif

注：JVM（java虚拟机）包括解释器，不同的JDK虚拟机是相同的，解释器不同。

2.2 JVM：

JVM是java的核心和基础，在java编译器和os平台之间的虚拟处理器。它是一种利用软件方法实现的抽象的计算机基于下层的操作系统和硬件平台，可以在上面执行java的字节码程序。

java编译器只要面向JVM，生成JVM能理解的代码或字节码文件。Java源文件经编译成字节码程序，通过JVM将每一条指令翻译成不同平台机器码，通过特定平台运行。

JVM执行程序的过程：

I.加载。class文件

II.管理并分配内存

III.执行垃圾收集

./f2.gif

3 简单介绍Inside JVM体系结构

当Inside JVM运行程序时，字节码，创建的对象，传递给方法的参数,返回值，局部变量以及运算的中间结果保存在运行时数据区中。规范本身对运行时数据区只有抽象的描述，也使得JVM可以容易的在各种计算机和设备上实现。

3.1 Inside JVM运行时数据区

3.1.1 方法区：

JVM中被装载的类型信息存储在一个逻辑上被称为方法区的内存中，JVM在装载完CLASS文件后提取其中的类型信息并将之存储在方法区。该类型的静态变量同样也存储在方法区中。由于所有的线程共享方法区，所以对方法区数据的访问必须考虑到线程的同步。

方法区的大小是不固定的，JVM可以通过装载新类型或者卸载已有类型来动态的调整方法区的大小，即可以改变方法区所占用的内存。方法区不一定是连续的，方法区可以在一个堆中自由分配。

JVM保存的在方法区中的存储类型以下信息

此类型的全限定名

此类型的直接超类的全限定名（除非这个类型为java.lang.object，它没有超类）

此类型为接口还是类

此类型的访问修饰符号（public,abstact或final的某个子集）

除以上基本信息，还得存储以下类型的具体信息

此类型的常量池

字段信息

方法信息

除常量以外的所有静态变量

一个到类classLoader的引用

一个到Class类的引用

常量池可以理解为本地指针数组，在JAVA的动态连接中起核心的左右，后边再做详细的介绍。

为了尽可能的提高访问的效率，必须优化存储在方法区中的类型信息的数据结构。所以，实现中还可以加如其他数据结构以加快访问原始数据的速度，如方法表。JVM为每个装载的非抽象类，都生成一个方法表，把他作为类信息的一部分保存在方法区。方法表也是一个本地指针数组，其元素为方法的入口地址。方法表所指向实例方法的数据包括以下信息：

此方法的操作数栈和局部变量的大小

此方法的字节码

异常表

JVM可以为每个对象生成一份方法表的copy（这样比较耗内存，但能提高访问的速度）或只在对象里保存到方法区中方法表的引用。这和C++中的VTBL很象，在C++中，对象有实例数据和一组指向对象可以调用的虚拟函数指针组成。

3.1.2 堆

Java程序在运行时创建的所有类室例或数组都放在同一个堆中，而一个JVM实例只有一个堆空间，所有线程都共享这个堆。堆空间可以自由的伸缩，也不必是连续的。

1. 常见的堆空间的设计：

(a) 把堆空间分为两部分： 句柄池和对象池，对象的引用为指向句柄池的本地指针，句柄池里的每个条目分为两部分，一部分为指向对象池的本地指针，一部分为指向方法区类数据区的本地指针。对象池里保存的是实例对象的数据，此数据是实例私有的。这种设计的好处有利于内存碎片的整理，当移动对象池中的对象时，句柄部分只需要修改指向对象池条目的地址。缺点就是两级指针的访问。

(b) 使对象指针直接的指向对象数据， 该数据包括指向方法区方法区数据类类型的指针和对象的实例数据。这样的优缺点正好和前边的方法相反。当移动堆中的对象时，对象的指针也得跟着改变，这就必须在整个运行时数据区中更新被移动对象的引用。

以上两种方法的思想可以类比为对链表和数组的删除和加入操作。

在Java中，数组和其他对象一样，总是存储在堆中并拥有一个与他们的类向关联的CLASS实例，所有具有相同维度和类型的数组都是一个类的实例，而不管数组的长度。

3.1.3 程序记数器

每个线程都有自己的程序记数器，它的内容总是下一条将被执行指令的地址。

3.1.4 Java栈

当一个线程被创建时都将得到自己的程序记数器和Java栈，Java栈以帧为单位保存调用信息。当线程调用一个方法时，JVM会压如一新的栈帧到Java栈，反之则弹出。也就是说，JVM只会对Java栈执行两种操作：以帧为单位的压栈和弹栈。帧的大小根据调用信息是可变的，后边做详细的介绍。由于Java栈上的数据是此线程私有的，因此不需考虑多线程下的栈数据的线程安全问题。

1. Inside JVM栈帧由三部分组成：局部变量区，操作数栈和帧数据区。

(a) 局部变量区

局部变量区的大小由调用方法的参数和方法的局部变量所决定。编译器按声名顺序将他们放到局部变量数组，此数组以字长为单位，从0开始记数。如果是实例方法，数组的第一个元素为实例的this指针。

在Java中，所有的对象都按引用传递，并且对象存储在堆中，在局部变量或操作数栈中不会有对象的COPY，只有对象引用。

(b) 操作数栈

操作数栈也是以字长为单位的数组，但不同于局部变量数组以索引去访问，它是通过标准的栈操作，压栈和弹栈来访问的。JVM没有寄存器,程序记数器也无法被程序指令直接访问。JVM的运行方式是基于栈的而非基于寄存器的，JVM的指令是从操作数栈中而不是寄存器中取得操作数的。虽然指令也可以从其他地方取得操作数，比如从字节码流中跟随在操作码之后的字节中或从常量池中，但主要还是从操作数栈中获取操作数。

JVM把操作数栈作为它的工作区，大多数指令都要从这里弹出数据，执行运算，然后把结果压回操作数栈，然后等相关的指令将结果再次弹出。操作数栈扮演了暂存操作数的角色。

(c) 帧数据区

除局部变量区和操作数栈外，Java栈帧还数据来支持常量池的解析、正常方法的返回以及异常派发机制。这些信息都保存在Java栈帧的帧数据区中。JVM可以通过帧数据区中指向常量池的指针来执行某个需要用到常量池数据的指令。

3.1.5 本地方法栈

当线程调用本地方法时，JVM会保持Java栈不变，不再在线程的Java栈中压入新的帧，Inside JVM只是简单地动态连接并直接调用本地方法。但是，本地方法有可能回调JVM中的Java方法，此时该线程会保存本地方法栈的状态并进入Java栈，在Java栈压入新的栈帧。

4 深入Java虚拟机：JVM中的Stack和Heap

在JVM中，内存分为两个部分，Stack（栈）和Heap（堆），这里，我们从JVM的内存管理原理的角度来认识Stack和Heap，并通过这些原理认清Java中静态方法和静态属性的问题。

4.1 一般，JVM的内存分为两部分：Stack和Heap。

Stack（栈） 是JVM的内存指令区。Stack管理很简单，push一定长度字节的数据或者指令，Stack指针压栈相应的字节位移；pop一定字节长度数据或者指令，Stack指针弹栈。Stack的速度很快，管理很简单，并且每次操作的数据或者指令字节长度是已知的。所以Java 基本数据类型，Java 指令代码，常量都保存在Stack中。

Heap（堆） 是JVM的内存数据区。Heap 的管理很复杂，每次分配不定长的内存空间，专门用来保存对象的实例。在Heap 中分配一定的内存来保存对象实例，实际上也只是保存对象实例的属性值，属性的类型和对象本身的类型标记等，并不保存对象的方法（方法是指令，保存在Stack中）,在Heap 中分配一定的内存保存对象实例和对象的序列化比较类似。而对象实例在Heap 中分配好以后，需要在Stack中保存一个4字节的Heap 内存地址，用来定位该对象实例在Heap 中的位置，便于找到该对象实例。

由于Stack的内存管理是顺序分配的，而且定长，不存在内存回收问题；而Heap 则是随机分配内存，不定长度，存在内存分配和回收的问题；因此在JVM中另有一个GC进程，定期扫描Heap ，它根据Stack中保存的4字节对象地址扫描Heap ，定位Heap 中这些对象，进行一些优化（例如合并空闲内存块什么的），并且假设Heap 中没有扫描到的区域都是空闲的，统统refresh（实际上是把Stack中丢失了对象地址的无用对象清除了），这就是垃圾收集的过程；关于垃圾收集的更深入讲解请参考51CTO之前的文章《JVM内存模型及垃圾收集策略解析》。 ./f3.gif JVM的体系结构

我们首先要搞清楚的是什么是数据以及什么是指令。然后要搞清楚对象的方法和对象的属性分别保存在哪里。

1）方法本身是指令的操作码部分，保存在Stack中；

2）方法内部变量作为指令的操作数部分，跟在指令的操作码之后，保存在Stack中（实际上是简单类型保存在Stack中，对象类型在Stack中保存地址，在Heap 中保存值）；上述的指令操作码和指令操作数构成了完整的Java 指令。

3）对象实例包括其属性值作为数据，保存在数据区Heap 中。

非静态的对象属性作为对象实例的一部分保存在Heap 中，而对象实例必须通过Stack中保存的地址指针才能访问到。因此能否访问到对象实例以及它的非静态属性值完全取决于能否获得对象实例在Stack中的地址指针。

4.2 非静态方法和静态方法的区别:

非静态方法 有一个和静态方法很重大的不同：非静态方法有一个隐含的传入参数，该参数是JVM给它的，和我们怎么写代码无关，这个隐含的参数就是对象实例在Stack中的地址指针。因此非静态方法（在Stack中的指令代码）总是可以找到自己的专用数据（在Heap 中的对象属性值）。当然非静态方法也必须获得该隐含参数，因此非静态方法在调用前，必须先new一个对象实例，获得Stack中的地址指针，否则JVM将无法将隐含参数传给非静态方法。

静态方法 无此隐含参数，因此也不需要new对象，只要class文件被ClassLoader load进入JVM的Stack，该静态方法即可被调用。当然此时静态方法是存取不到Heap 中的对象属性的。

总结一下该过程：当一个class文件被ClassLoader load进入JVM后，方法指令保存在Stack中，此时Heap 区没有数据。然后程序技术器开始执行指令，如果是静态方法，直接依次执行指令代码，当然此时指令代码是不能访问Heap 数据区的；如果是非静态方法，由于隐含参数没有值，会报错。因此在非静态方法执行前，要先new对象，在Heap 中分配数据，并把Stack中的地址指针交给非静态方法，这样程序技术器依次执行指令，而指令代码此时能够访问到Heap 数据区了。

4.2.1 静态属性和动态属性:

前面提到对象实例以及动态属性都是保存在Heap 中的，而Heap 必须通过Stack中的地址指针才能够被指令（类的方法）访问到。因此可以推断出：静态属性是保存在Stack中的，而不同于动态属性保存在Heap 中。正因为都是在Stack中，而Stack中指令和数据都是定长的，因此很容易算出偏移量，也因此不管什么指令（类的方法），都可以访问到类的静态属性。也正因为静态属性被保存在Stack中，所以具有了全局属性。

在JVM中，静态属性保存在Stack指令内存区，动态属性保存在Heap数据内存区。

5 Java虚拟机（JVM）中的内存设置详解

在一些规模稍大的应用中，Java虚拟机（JVM）的内存设置尤为重要，想在项目中取得好的效率，GC（垃圾回收）的设置是第一步。

PermGen space：全称是Permanent Generation space.就是说是永久保存的区域,用于存放Class和Meta信息,Class在被Load的时候被放入该区域Heap space：存放Instance。

GC(Garbage Collection)应该不会对PermGen space进行清理,所以如果你的APP会LOAD很多CLASS的话,就很可能出现PermGen space错误

5.1 Java Heap分为3个区

1.Young

2.Old

3.Permanent

Young保存刚实例化的对象。当该区被填满时，GC会将对象移到Old区。Permanent区则负责保存反射对象，本文不讨论该区。

JVM的Heap分配可以使用-X参数设定，

-Xms 初始Heap大小

-Xmx java heap最大值

-Xmn young generation的heap大小

5.2 JVM有2个GC线程

第一个线程负责回收Heap的Young区

第二个线程在Heap不足时，遍历Heap，将Young 区升级为Older区

Older区的大小等于-Xmx减去-Xmn，不能将-Xms的值设的过大，因为第二个线程被迫运行会降低JVM的性能。

为什么一些程序频繁发生GC?

有如下原因：

1.程序内调用了System.gc()或Runtime.gc()。

2.一些中间件软件调用自己的GC方法，此时需要设置参数禁止这些GC。

3.Java的Heap太小，一般默认的Heap值都很小。

4.频繁实例化对象，Release对象 此时尽量保存并重用对象，例如使用StringBuffer()和String()。

如果你发现每次GC后，Heap的剩余空间会是总空间的50%，这表示你的Heap处于健康状态,许多Server端的Java程序每次GC后最好能有65%的剩余空间

经验之谈：

1．Server端JVM最好将-Xms和-Xmx设为相同值。为了优化GC，最好让-Xmn值约等于-Xmx的1/3。

2．一个GUI程序最好是每10到20秒间运行一次GC，每次在半秒之内完成。

注意：

1．增加Heap的大小虽然会降低GC的频率，但也增加了每次GC的时间。并且GC运行时，所有的用户线程将暂停，也就是GC期间，Java应用程序不做任何工作。

2．Heap大小并不决定进程的内存使用量。进程的内存使用量要大于-Xmx定义的值，因为Java为其他任务分配内存，例如每个线程的Stack等。

Stack的设定

每个线程都有他自己的Stack。

-Xss 每个线程的Stack大小

Stack的大小限制着线程的数量。如果Stack过大就好导致内存溢漏。-Xss参数决定Stack大小，例如-Xss1024K。如果Stack大小，也会导致Stack溢漏。

硬件环境

硬件环境也影响GC的效率，例如机器的种类，内存，swap空间，和CPU的数量。 如果你的程序需要频繁创建很多transient对象，会导致JVM频繁GC。这种情况你可以增加机器的内存，来减少Swap空间的使用。

4种GC

1、第一种为单线程GC，也是默认的GC，该GC适用于单CPU机器。

2、第二种为Throughput GC，是多线程的GC，适用于多CPU，使用大量线程的程序。第二种GC与第一种GC相似，不同在于GC在收集Young区是多线程的，但在Old区和第一种一样，仍然采用单线程。-XX:+UseParallelGC参数启动该GC。

3、第三种为Concurrent Low Pause GC，类似于第一种，适用于多CPU，并要求缩短因GC造成程序停滞的时间。这种GC可以在Old区的回收同时，运行应用程序。-XX:+UseConcMarkSweepGC参数启动该GC。

4、第四种为Incremental Low Pause GC，适用于要求缩短因GC造成程序停滞的时间。这种GC可以在Young区回收的同时，回收一部分Old区对象。-Xincgc参数启动该GC。

单文件的JVM内存进行设置

默认的java虚拟机的大小比较小，在对大数据进行处理时java就会报错：java.lang.OutOfMemoryError。

设置jvm内存的方法，对于单独的.class，可以用下面的方法对Test运行时的jvm内存进行设置。

java -Xms64m -Xmx256m Test

-Xms是设置内存初始化的大小

-Xmx是设置最大能够使用内存的大小（最好不要超过物理内存大小）

tomcat启动jvm内存设置

Linux：

在/usr/local/apache-tomcat-5.5.23/bin目录下的catalina.sh添加：JAVA$_{OPTS}$='-Xms512m -Xmx1024m'要加“m”说明是M否则就是KB了，在启动tomcat时会报内存不足。

-Xms：初始值

-Xmx：最大值

-Xmn：最小值Windows

在catalina.bat最前面加入set JAVA$_{OPTS}$=-Xms128m -Xmx350m 如果用startup.bat启动tomcat,OK设置生效.够成功的分配200M内存.但是如果不是执行startup.bat启动tomcat而是利用windows的系统服务启动tomcat服务,上面的设置就不生效了,就是说set JAVA$_{OPTS}$=-Xms128m -Xmx350m 没起作用.上面分配200M内存就OOM了..windows服务执行的是bin.exe.他读取注册表中的值,而不是catalina.bat的设置.解决办法:

修改注册表HKEY$_{LOCALMACHINE}$ Software FoundationService Manager5

原值为

-Dcatalina.home=”C:5.0”

-Djava.endorsed.dirs=”C:5.0”

-Xrs加入 -Xms300m -Xmx350m

重起tomcat服务,设置生效

weblogic启动jvm内存设置

在weblogic中，可以在startweblogic.cmd中对每个domain虚拟内存的大小进行设置，默认的设置是在commEnv.cmd里面。

JBoss

默认可以使用的内存为64MB $JBOSSDIR$/bin/run.config JAVA$_{OPTS}$ = ”-server -Xms128 -Xmx512”

Eclipse

在所在目录下，键入 eclipse.exe -vmargs -Xms256m -Xmx512m 256m表示JVM堆内存最小值 512m表示JVM堆内存最大

Websphere

进入控制台去设置：应用程序服务器 ¿ server1 ¿ 进程定义 ¿ Java 虚拟机

6 深入学习JVM内存设置原理和调优

这里向大家描述一下JVM内存设置原理和内存调优，设置jvm内存的方法，对于单独的.class，可以用下面的方法对Test运行时的jvm内存进行设置。

6.1 JVM内存设置原理

默认的java虚拟机的大小比较小，在对大数据进行处理时java就会报错：java.lang.OutOfMemoryError。设置jvm内存的方法，对于单独的.class，可以用下面的方法对Test运行时的jvm内存进行设置。

java-Xms64m-Xmx256mTest

-Xms是设置内存初始化的大小

-Xmx是JVM内存设置中设置最大能够使用内存的大小（最好不要超过物理内存大小）

在weblogic中，可以在startweblogic.cmd中对每个domain虚拟内存的大小进行设置，默认的设置是在commEnv.cmd里面。简单介绍了JVM内存设置，下面我们看一下JVM内存的调优。

6.2 JVM内存的调优

6.2.1 Heap设定与垃圾回收

JavaHeap分为3个区，Young，Old和Permanent。Young保存刚实例化的对象。当该区被填满时，GC会将对象移到Old区。Per则负责保存反射对象，本文不讨论该区。JVM的Heap分配可以使用-X参数设定，

JVM有2个GC线程。

第一个线程负责回收Heap的Young区。

第二个线程在Heap不足时，遍历Heap，将Young区升级为Older区。Older区的大小等于-Xmx减去-Xmn，不能将-Xms的值设的过大，因为第二个线程被迫运行会降低JVM的性能。

为什么一些程序频繁发生GC？有如下原因：

- 程序内调用了System.gc()或Runtime.gc()。
- 一些中间件软件调用自己的GC方法，此时需要设置参数禁止这些GC。
- Java的Heap太小，一般默认的Heap值都很小。
- 频繁实例化对象，Release对象。此时尽量保存并重用对象，例如使用StringBuffer()和String()。

如果你发现每次GC后，Heap的剩余空间会是总空间的50%，这表示你的Heap处于健康状态。许多Server端的Java程序每次GC后最好能有65%的剩余空间。

经验之谈：

1．Server端JVM最好将-Xms和-Xmx设为相同值。为了优化GC，最好让-Xmn值约等于-Xmx的1/3[1]。

2．一个GUI程序最好是每10到20秒间运行一次GC，每次在半秒之内完成[1]。

注意：

1．增加Heap的大小虽然会降低GC的频率，但也增加了每次GC的时间。并且GC运行时，所有的用户线程将暂停，也就是GC期间，Java应用程序不做任何工作。

2．Heap大小并不决定进程的内存使用量。进程的内存使用量要大于-Xmx定义的值，因为Java为其他任务分配内存，例如每个线程的Stack等。

6.2.2 Stack的设定

每个线程都有他自己的Stack。

Xss每个线程的Stack大小

Stack的大小限制着线程的数量。如果Stack过大就好导致内存溢漏。-Xss参数决定Stack大小，例如-Xss1024K。如果Stack大小，也会导致Stack溢漏。

6.2.3 硬件环境

硬件环境也影响GC的效率，例如机器的种类，内存，swap空间，和CPU的数量。

如果你的程序需要频繁创建很多transient对象，会导致JVM频繁GC。这种情况你可以增加机器的内存，来减少Swap空间的使用[1]。

6.2.4 4种GC

第一种为单线程GC，也是默认的GC。，该GC适用于单CPU机器。

第二种为ThroughputGC，是多线程的GC，适用于多CPU，使用大量线程的程序。第二种GC与第一种GC相似，不同在于GC在收集Young区是多线程的，但在Old区和第一种一样，仍然采用单线程。-XX:+UseParallelGC参数启动该GC。

第三种为ConcurrentLowPauseGC，类似于第一种，适用于多CPU，并要求缩短因GC造成程序停滞的时间。这种GC可以在Old区的回收同时，运行应用程序。-XX:+UseConcMarkSweepGC参数启动该GC。

第四种为IncrementalLowPauseGC，适用于要求缩短因GC造成程序停滞的时间。这种GC可以在Young区回收的同时，回收一部分Old区对象。-Xincgc参数启动该GC。

4种GC的具体描述参见[2]。关于JVM内存设置的内容就介绍到这里，请关注本节其
他相关报道。

[1]DEFINITION NOT FOUND.
[2]DEFINITION NOT FOUND.