jvm

说明：做java开发的几乎都知道jvm这个名词，但是由于jvm对实际的简单开发的来说关联的还是不多，一般工作个一两年（当然不包括爱学习的及专门做性能优化的什么的），很少有人能很好的去学习及理解什么是jvm，以及弄清楚jvm的工作原理，个人认为这块还是非常有必要去认真了解及学习的，特别是刚入门或入门不久的java开发来说，这是java的基石。

JVM(Java Virtual Machine，Java虚拟机)

Java程序的跨平台特性主要是指字节码文件可以在任何具有Java虚拟机的计算机或者电子设备上运行，Java虚拟机中的Java解释器负责将字节码文件解释成为特定的机器码进行运行。因此在运行时，Java源程序需要通过编译器编译成为.class文件。众所周知java.exe是java class文件的执行程序，但实际上java.exe程序只是一个执行的外壳，它会装载jvm.dll（windows下，下皆以windows平台为例，linux下和solaris下其实类似，为：libjvm.so），这个动态连接库才是java虚拟机的实际操作处理所在。

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。Java语言最重要的特点就是跨平台运行。使用JVM就是为了支持与操作系统无关，实现跨平台。所以，JAVA虚拟机JVM是属于JRE的，而现在我们安装JDK时也附带安装了JRE(当然也可以单独安装JRE)。

内存空间：

JVM内存空间包含：方法区、java堆、java栈、本地方法栈。

方法区是各个线程共享的区域，存放类信息、常量、静态变量。

java堆也是线程共享的区域，我们的类的实例就放在这个区域，可以想象你的一个系统会产生很多实例，因此java堆的空间也是最大的。如果java堆空间不足了，程序会抛出OutOfMemoryError异常。

java栈是每个线程私有的区域，它的生命周期与线程相同，一个线程对应一个java栈，每执行一个方法就会往栈中压入一个元素，这个元素叫“栈帧”，而栈帧中包括了方法中的局部变量、用于存放中间状态值的操作栈，这里面有很多细节，我们以后再讲。如果java栈空间不足了，程序会抛出StackOverflowError异常，想一想什么情况下会容易产生这个错误，对，递归，递归如果深度很深，就会执行大量的方法，方法越多java栈的占用空间越大。

本地方法栈角色和java栈类似，只不过它是用来表示执行本地方法的，本地方法栈存放的方法调用本地方法接口，最终调用本地方法库，实现与操作系统、硬件交互的目的。

PC寄存器，说到这里我们的类已经加载了，实例对象、方法、静态变量都去了自己改去的地方，那么问题来了，程序该怎么执行，哪个方法先执行，哪个方法后执行，这些指令执行的顺序就是PC寄存器在管，它的作用就是控制程序指令的执行顺序。

执行引擎当然就是根据PC寄存器调配的指令顺序，依次执行程序指令。

JVM内存区域划分

粗略分来，JVM的内部体系结构分为三部分，分别是：类装载器（ClassLoader）子系统，运行时数据区，和执行引擎。

类装载器

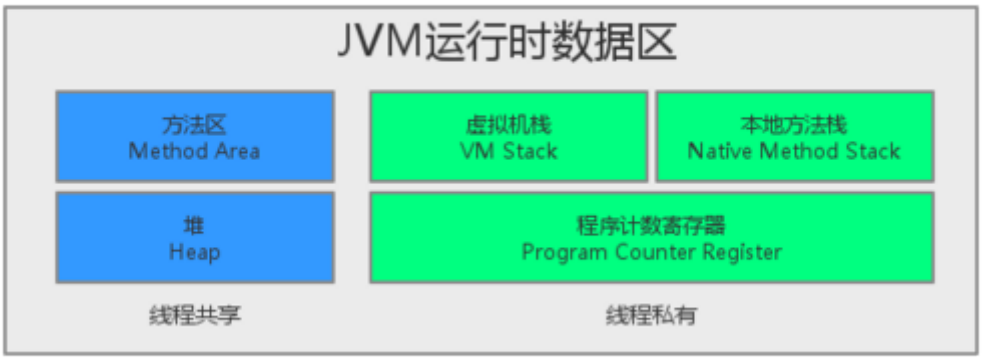
每一个Java虚拟机都由一个类加载器子系统（class loader subsystem），负责加载程序中的类型（类和接口），并赋予唯一的名字。每一个Java虚拟机都有一个执行引擎（execution engine）负责执行被加载类中包含的指令。JVM的两种类装载器包括：启动类装载器和用户自定义类装载器，启动类装载器是JVM实现的一部分，用户自定义类装载器则是Java程序的一部分，必须是ClassLoader类的子类。

执行引擎：它或者在执行字节码，或者执行本地方法

主要的执行技术有:解释，即时编译，自适应优化、芯片级直接执行其中解释属于第一代JVM，即时编译JIT属于第二代JVM，自适应优化（目前Sun的HotspotJVM采用这种技术）则吸取第一代JVM和第二代JVM的经验，采用两者结合的方式 。

自适应优化：开始对所有的代码都采取解释执行的方式，并监视代码执行情况，然后对那些经常调用的方法启动一个后台线程，将其编译为本地代码，并进行仔细优化。若方法不再频繁使用，则取消编译过的代码，仍对其进行解释执行。

运行时数据区：主要包括：方法区，堆，Java栈，PC寄存器，本地方法栈



jvm结构

- 方法区和堆由所有线程共享

堆：存放所有程序在运行时创建的对象

方法区：当JVM的类装载器加载.class文件，并进行解析，把解析的类型信息放入方法区。

- Java栈和PC寄存器由线程独享

JVM栈是线程私有的，每个线程创建的同时都会创建JVM栈，JVM栈中存放的为当前线程中局部基本类型的变量（java中定义的八种基本类型：boolean、char、byte、short、int、long、float、double）、部分的返回结果以及Stack Frame，非基本类型的对象在JVM栈上仅存放一个指向堆上的地址

- 本地方法栈：存储本地方法调用的状态

JVM运行时数据区

因为jvm运行时的数据区对我们开发来说还是特别重要要掌握的知识所以单拎开来西说下。

- 方法区域（Method Area）

在Sun JDK中这块区域对应的为PermanetGeneration，又称为持久代。

方法区域存放了所加载的类的信息（名称、修饰符等）、类中的静态变量、类中定义为final类型的常量、类中的Field信息、类中的方法信息，当开发人员在程序中通过Class对象中的getName、isInterface等方法来获取信息时，这些数据都来源于方法区域，同时方法区域也是全局共享的，在一定的条件下它也会被GC，当方法区域需要使用的内存超过其允许的大小时，会抛出OutOfMemory的错误信息。

- 堆（Heap）

它是JVM用来存储对象实例以及数组值的区域，可以认为Java中所有通过new创建的对象的内存都在此分配，Heap中的对象的内存需要等待GC进行回

收。

堆是JVM中所有线程共享的，因此在其上进行对象内存的分配均需要进行加锁，这也导致了new对象的开销是比较大的

Sun Hotspot JVM为了提升对象内存分配的效率，对于所创建的线程都会分配一块独立的空间TLAB（Thread Local Allocation Buffer），其大小由JVM根据运行的情况计算而得，在TLAB上分配对象时不需要加锁，因此JVM在给线程的对象分配内存时会尽量的在TLAB上分配，在这种情况下JVM中分配对象内存的性能和C基本是一样高效的，但如果对象过大的话则仍然是直接使用堆空间分配

TLAB仅作用于新生代的Eden Space，因此在编写Java程序时，通常多个小的对象比大的对象分配起来更加高效。

- JavaStack(java的栈)：虚拟机只会直接对Javastack执行两种操作：以帧为单位的压栈或出栈

每个帧代表一个方法，Java方法有两种返回方式，return和抛出异常，两种方式都会导致该方法对应的帧出栈和释放内存。

帧的组成：局部变量区（包括方法参数和局部变量，对于instance方法，还要首先保存this类型，其中方法参数按照声明顺序严格放置，局部变量可以任意放置），操作数栈，帧数据区（用来帮助支持常量池的解析，正常方法返回和异常处理）。

- ProgramCounter(程序计数器)

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。

若thread执行Java方法，则PC保存下一条执行指令的地址。若thread执行native方法，则Pc的值为undefined

- Nativemethodstack(本地方法栈)：保存native方法进入区域的地址

依赖于本地方法的实现，如某个JVM实现的本地方法借口使用C连接模型，则本地方法栈就是C栈，可以说某线程在调用本地方法时，就进入了一个不受JVM限制的领域，也就是JVM可以利用本地方法来动态扩展本身。

JVM垃圾回收

Sun的JVMGenerationalCollecting(垃圾回收)原理是这样的：把对象分为年青代(Young)、年老代(Tenured)、持久代(Perm)，对不同生命周期的对象使用不同的算法。(基于对对象生命周期分析)

通常我们说的JVM内存回收总是在指堆内存回收，确实只有堆中的内容是动态申请分配的，所以以上对象的年轻代和年老代都是指的JVM的Heap空间，而持久代则是之前提到的MethodArea，不属于Heap。

GC的基本原理：将内存中不再被使用的对象进行回收，GC中用于回收的方法称为收集器，由于GC需要消耗一些资源和时间，Java在对对象的生命周期特征进行分析后，按照新生代、旧生代的方式来对对象进行收集，以尽可能的缩短GC对应用造成的暂停

（1）对新生代的对象的收集称为minor GC；

（2）对旧生代的对象的收集称为Full GC；

（3）程序中主动调用System.gc()强制执行的GC为Full GC。

不同的对象引用类型， GC会采用不同的方法进行回收，JVM对象的引用分为了四种类型：

（1）强引用：默认情况下，对象采用的均为强引用（这个对象的实例没有其他对象引用，GC时才会被回收）

（2）软引用：软引用是Java中提供的一种比较适合于缓存场景的应用（只有在内存不够用的情况下才会被GC）

（3）弱引用：在GC时一定会被GC回收

（4）虚引用：由于虚引用只是用来得知对象是否被GC

- Young（年轻代）

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。

- Tenured（年老代）

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。

- Perm（持久代）

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。

GC如何判断一个对象为”垃圾”的

答案：1.引用计数算法(已被淘汰的算法)：给对象中添加一个引用计数器,每当有一个地方引用它时,计数器值就加1;当引用失效时,计数器值就减1;任何时刻计数器为0的对象就是不可能再被使用的。

2.可达性分析算法

目前主流的编程语言(java,C#等)的主流实现中,都是称通过可达性分析(Reachability Analysis)来判定对象是否存活的。这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点,从这些节点开始向下搜索,搜索所走过的路径称为引用链(Reference Chain),当一个对象到GC Roots没有任何引用链相连(用图论的话来说,就是从GC Roots到这个对象不可达)时,则证明此对象是不可用的。

在Java语言中,可作为GC Roots的对象包括下面几种:

- 虚拟机栈(栈帧中的本地变量表)中引用的对象。
- 方法区中类静态属性引用的对象。
- 方法区中常量引用的对象。
- 本地方法栈中JNI(即一般说的Native方法)引用的对象

二.被GC判断为”垃圾”的对象一定会回收吗

即使在可达性分析算法中不可达的对象,也并非是“非死不可”的,这时候它们暂时处于“缓刑”阶段,要真正宣告一个对象死亡,至少要经历两次标记过程:如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链,那它将会被第一次标记并且进行一次筛选,筛选的条件是此对象是否有必要执行finalize()方法。当对象没有覆盖finalize()方法,或者finalize()方法已经被虚拟机调用过,虚拟机将这两种情况都视为“没有必要执行”。(即意味着直接回收)

如果这个对象被判定为有必要执行finalize()方法,那么这个对象将会放置在一个叫做F-Queue的队列之中,并在稍后由一个由虚拟机自动建立的、低优先级的Finalizer线程去执行它。这里所谓的“执行”是指虚拟机会触发这个方法,但并不承诺会等待它运行结束,这样做的原因是,如果一个对象在finalize()方法中执行缓慢,或者发生了死循环(更极端的情况),将很可能会导致F-Queue队列中其他对象永久处于等待,甚至导致整个内存回收系统崩溃。

finalize()方法是对象逃脱死亡命运的最后一次机会,稍后GC将对F-Queue中的对象进行第二次小规模的标记,如果对象要在finalize()中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可,譬如把自己(this关键字)赋值给某个类变量或者对象的成员变量,那在第二次标记时它将被移除出“即将回收”的集合;如果对象这时候还没有逃脱,那基本上它就真的被回收了。

SAVE_HOOK对象的finalize()方法确实被GC收集器触发过,并且在被收集前成功逃脱了。另外一个值得注意的地方是,代码中有两段完全一样的代码片段,执行结果却是一次逃脱成功,一次失败,这是因为任何一个对象的finalize()方法都只会被系统自动调用一次,如果对象面临下一次回收,它的finalize()方法不会被再次执行,因此第二段代码的自救行动失败了。因为finalize()方法已经被虚拟机调用过,虚拟机都视为“没有必要执行”。(即意味着直接回收)

你能不能谈谈，java GC是在什么时候，对什么东西，做了什么事情？

什么时候：程序员不能具体控制时间，系统在不可预测的时间调用System.gc()函数的时候；当然可以通过调优，用NewRatio控制newObject和oldObject的比例，用MaxTenuringThreshold 控制进入oldObject的次数，使得oldObject 存储空间延迟达到full gc,从而使得计时器引发gc时间延迟OOM的时间延迟，以延长对象生存期。
对什么东西：超出了作用域或引用计数为空的对象；从gc root开始搜索找不到的对象，而且经过一次标记、清理，仍然没有复活的对象。
做什么：删除不使用的对象，回收内存空间；运行默认的finalize,当然程序员想立刻调用就用dipose调用以释放资源如文件句柄，JVM用from survivor、to survivor对它进行标记清理，对象序列化后也可以使它复活。

关于JVM内存管理的一些建议

1、手动将生成的无用对象，中间对象置为null，加快内存回收。

2、对象池技术如果生成的对象是可重用的对象，只是其中的属性不同时，可以考虑采用对象池来较少对象的生成。如果有空闲的对象就从对象池中取出使用，没有再生成新的对象，大大提高了对象的复用率。

3、JVM调优通过配置JVM的参数来提高垃圾回收的速度，如果在没有出现内存泄露且上面两种办法都不能保证JVM内存回收时，可以考虑采用JVM调优的方式来解决，不过一定要经过实体机的长期测试，因为不同的参数可能引起不同的效果。如-Xnoclassgc参数等。