**JVM中的Stack(栈)和Heap(堆)**

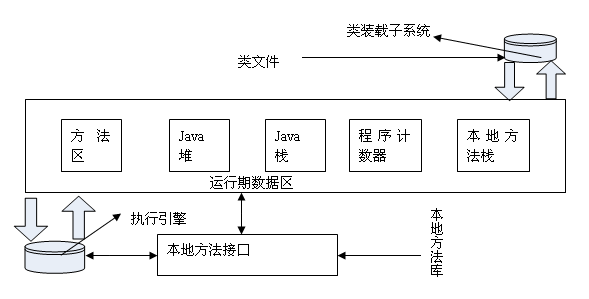
      在JVM中，静态属性保存在Stack指令内存区，动态属性保存在Heap数据内存区。本文将从JVM的角度来讲解Java虚拟机的这一机制。

     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。

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。



**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。

**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。

**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。

**注1：**在JVM加载方法f()所属的类的时候，会在内存的方法区中开辟一个空间来存放**常量池**的信息。这里有一个隐藏的步骤：对于String字面值，JVM会在堆中分配一个叫做inner String的对象(相同的字面值只有唯一的一个Inner String)。然后在常量池中存放指向这个inner String的引用地址。这个过程是常量池解析的一部分。因此String s = new String("hello");实际上是有两个字符串对象在堆中存在的。  
       最后，当f()方法结束之后，栈中f()所对应的栈帧内所有的指令处理的中间数据全部执行完毕，这是栈帧会被回收，对象引用s的内存区域将不会存在，而堆中的s对象空间仍然会存在，当然，如果s对象以后再也不可能在其他地方引用到的话，JVM的垃圾回收机制会在某一个不确定的时候回收这一部分的对象空间。

**注2：**栈 Java虚拟机的栈有三个区域:局部变量区、运行环境区、操作数区。

(1)局部变量区　每个Java方法使用一个固定大小的局部变量集。它们按照与vars寄存器的字偏移量来寻址。局部变量都是32位的。长整数和双精度浮点数占据了两个局部变量的空间,却按照第一个局部变量的索引来寻址。(例如,一个具有索引n的局部变量,如果是一个双精度浮点数,那么它实际占据了索引n和n+1所代表的存储空间。)虚拟机规范并不要求在局部变量中的64位的值是64位对齐的。虚拟机提供了把局部变量中的值装载到操作数栈的指令,也提供了把操作数栈中的值写入局部变量的指令。

(2)运行环境区　在运行环境中包含的信息用于动态链接,正常的方法返回以及异常传播。  
◆动态链接   
 运行环境包括对指向当前类和当前方法的解释器符号表的指针,用于支持方法代码的动态链接。方法的class文件代码在引用要调用的方法和要访问的变量时使用符号。动态链接把符号形式的方法调用翻译成实际方法调用,装载必要的类以解释还没有定义的符号,并把变量访问翻译成与这些变量运行时的存储结构相应的偏移地址。动态链接方法和变量使得方法中使用的其它类的变化不会影响到本程序的代码。  
◆正常的方法返回  
 如果当前方法正常地结束了,在执行了一条具有正确类型的返回指令时,调用的方法会得到一个返回值。执行环境在正常返回的情况下用于恢复调用者的寄存器,并把调用者的程序计数器增加一个恰当的数值,以跳过已执行过的方法调用指令,然后在调用者的执行环境中继续执行下去。  
◆异常和错误传播  
 异常情况在Java中被称作Error(错误)或Exception(异常),是Throwable类的子类,在程序中的原因是:①动态链接错,如无法找到所需的class文件。②运行时错,如对一个空指针的引用

 (3)操作数栈区　机器指令只从操作数栈中取操作数,对它们进行操作,并把结果返回到栈中。选择栈结构的原因是:在只有少量寄存器或非通用寄存器的机器(如Intel486)上,也能够高效地模拟虚拟机的行为。操作数栈是32位的。它用于给方法传递参数,并从方法接收结果,也用于支持操作的参数,并保存操作的结果。例如,iadd指令将两个整数相加。相加的两个整数应该是操作数栈顶的两个字。这两个字是由先前的指令压进堆栈的。这两个整数将从堆栈弹出、相加,并把结果压回到操作数栈中。

       每个原始数据类型都有专门的指令对它们进行必须的操作。每个操作数在栈中需要一个存储位置,除了long和double型,它们需要两个位置。操作数只能被适用于其类型的操作符所操作。例如,压入两个int类型的数,如果把它们当作是一个long类型的数则是非法的。在Sun的虚拟机实现中,这个限制由字节码验证器强制实行。但是,有少数操作(操作符dupe和swap),用于对运行时数据区进行操作时是不考虑类型的。

**堆--用new建立，垃圾自动回收负责回收**

1、堆是一个"运行时"数据区，类实例化的对象就是从堆上去分配空间的；

2、在堆上分配空间是通过"new"等指令建立的；

3、Java针对堆的操作和C++的区别就是，Java不需要在空间不用的时候来显式的释放；

4、Java的堆是由Java的垃圾回收机制来负责处理的，堆是动态分配内存大小，垃圾收集器可以自动回收不再使用的内存空间。

5、但缺点是，因为在运行时动态分配内存，所以内存的存取速度较慢。

例如：

|  |
| --- |
| String str = new String("abc"); |

**栈--存放基本数据类型，速度快**

1、栈中主要存放一些基本类型的变量（int, short, long, byte, float, double, boolean, char）和对象句柄；

2、栈的存取速度比堆要快；

3、栈数据可以共享；

4、栈的数据大小与生存期必须是确定的，缺乏灵活性。

例如：

|  |
| --- |
| int a = 3; |