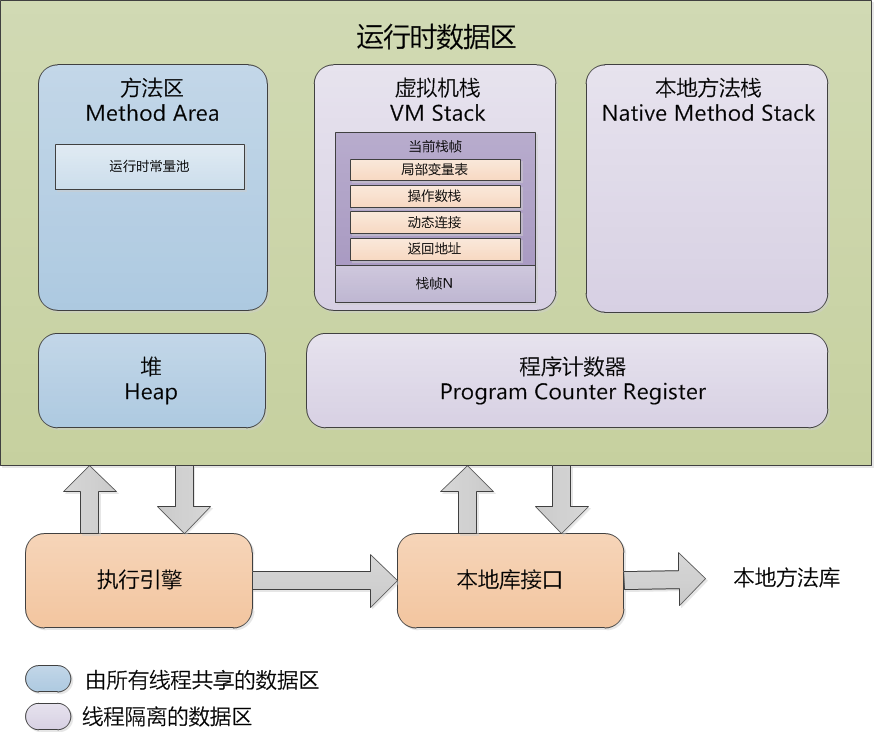
**概述**

JVM内存模型是理解JVM各种机制的基础，了解JVM内存模型也是Java程序员的基本功之一。大多数JVM内存模型的文章停留在对各种概念的说明，而涉及到具体的过程更多的是语焉不详，让人云里雾里。本文就以普通对象为例子，来说明JVM模型创建对象的过程，让大家对JVM内存模型有初级的认识，为理解JVM的各种参数调试、JVM垃圾管理等打下基础。

**内存结构**

本文仅仅对普通的java对象做说明，不涉及更深层的如堆内存管理（垃圾回收机制），也不涉及直接内存的内容。下面是JDK1.7的JVM内存结构图：



图一

以下引自java虚拟机规范：

1）方法区（Method Area）

方法区（Method Area）是可供各条线程共享的运行时内存区域。它存储了每一个类的结构信息，例如运行时常量池（Runtime Constant Pool）、字段和方法数据、构造函数和普通方法的字节码内容、还包括一些在类、实例、接口初始化时用到的特殊方法。

2）运行时常量池（Runtime Constant Pool）

运行时常量池是每一个类或接口的常量池（Constant Pool）的运行时表示形式，它包括了若干种不同的常量：从编译期可知的数值字面量到必须运行期解析后才能获得的方法或字段引用。每一个运行时常量池都分配在Java虚拟机的方法区之中，在类和接口被加载到虚拟机后，对应的运行时常量池就被创建出来。

3）虚拟机栈（VM Stack）

每一条Java虚拟机线程都有自己私有的Java虚拟机栈，这个栈与线程同时创建，用于存储栈帧。Java虚拟机栈的作用，就是用于存储局部变量与一些过程结果的地方。另外，它在方法调用和返回中也扮演了很重要的角色。

4）堆（Heap）

在Java虚拟机中，堆（Heap）是可供各条线程共享的运行时内存区域，也是供所有类实例和数组对象分配内存的区域。Java堆在虚拟机启动的时候就被创建，它存储了被自动内存管理系统（垃圾收集器）所管理的各种对象，这些受管理的对象无需，也无法显式地被销毁。

5）程序计数器（Program Counter Register）

每一条Java虚拟机线程都有自己的程序计数器。在任意时刻，一条Java虚拟机线程只会执行一个方法的代码，这个正在被线程执行的方法称为该线程的当前方法。如果这个方法不是native的，那PC寄存器就保存Java虚拟机正在执行的字节码指令的地址，如果该方法是native的，那PC寄存器的值是undefined。

6）本地方法栈（Native Method Stack）

Java虚拟机实现可能会使用到传统的栈来支持native方法（指使用Java以外的其他语言编写的方法）的执行，这个栈就是本地方法栈（Native Method Stack）。当Java虚拟机使用其他语言（例如C语言）来实现指令集解释器时，也会使用到本地方法栈。如果Java虚拟机不支持native方法，并且自己也不依赖传统栈的话，可以无需支持本地方法栈，如果支持本地方法栈，那这个栈一般会在线程创建的时候按线程分配。

本地方法栈的内容不在本文的讨论范围之内。

**实例说明**

下面来看一个最简单的Java Bean：

**public** **class** **User** {

**private** **int** id;

**private** String name;

**private** Object ob = **new** Object();

**public** **int** **getId**() {

**return** id;

}

**public** **void** **setId**(**int** id) {

**this**.id = id;

}

**public** String **getName**() {

**return** name;

}

**public** **void** **setName**(String name) {

**this**.name = name;

}

**public** Object **getOb**() {

**return** ob;

}

**public** **void** **setOb**(Object ob) {

**this**.ob = ob;

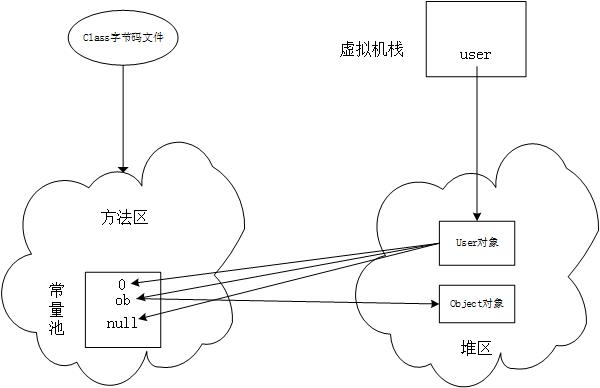
}

}

当我们在main方法中通过new创建一个User的实例，它的过程是怎样的呢？首先，JVM会启动一个main方法的线程，然后初始化好线程所需要的资源，包括栈帧信息等；然后JVM会通过类加载器把Class源码加载到方法区；接着，根据类信息为User类在堆内开辟一块内存，将User实例化，把包括对象头信息在内的一系列对象的信息存到堆当中，这个过程会对类成员变量进行初始化，引用类型未初始化的赋值为null，基本数据类型初始化值见表一；已经初始化的如果是字符串引用就将其分配到常量池，如果是对象引用会将引用地址存储到常量池中，将引用对象实例化到堆中，过程和User实例化过程相同。

严格意义上讲，JVM的内存中的堆栈等概念和C的完全不同，C语言分为代码段、数据段、BSS段、堆段、栈段。Java是线程式的管理方式，其方法区、堆和栈都是动态管理的，每一个区域都是一个或者几个线程，共享同一个进程的代码段、数据段等。例如栈，JVM的栈不但包含了方法执行的临时变量，还包含了其他一些如栈帧等数据。

User加载并初始化完成后我们就发现，数据在JVM的内存分布如下图：



图二

我们可以看到，常量池中有个0、ob和null，对应的存放的就是user的成员变量id的值、ob的引用地址以及为存name引用开辟的地址。Java类在初始化的时候，如果成员变量没有赋值，那么会类在初始化的时候会给其默认赋一个值，引用类型默认就为null。在该实例中，Object对象是没有成员变量的，所以在常量池中并没有Object的成员变量值，Object类的加载和初始化过程和User类相同。

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 类型 | boolean | char | byte | short | int | long | float | double | 引用 |
| 默认值 | false | 0 | 0 | 0 | 0 | 0L | 0.0f | 0.0 | null |

表一

当成员变量为static修饰的时候怎么办？Java的static关键字和c语言的几乎完全不同，当JVM加载字节码文件时候遇见static关键字修饰的变量，就会在加载的时候把该变量初始化，变量值或者引用类型地址会放到常量池中，该部分常量池又叫静态常量池，也是运行时常量池的一部分，垃圾回收时候会和方法区一起回收。因为这时候并没有实例化类对象，调用static变量的时候也没法通过对象调用，也没法使用this调用，而是直接通过方法区调用。

当变量位于方法内则该变量是局部变量，局部变量存在栈的局部变量表中，例如，我们把getName()方法改为：

**public** String **getName**() {

String myName = **this**.name

**return** myName;

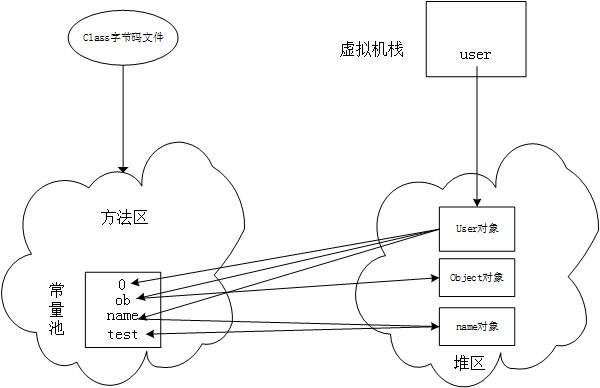
}

局部变量表是一组变量值存储空间，以变量槽为最小单位，用于存放方法参数和方法内部定义的局部变量。当执行getName()方法时候，myName变量会被放到栈的一个slot内，当方法执行结束后，超出变量所在作用域，那么该变量槽将会被废弃，下次调用新的方法的时候会覆盖重用，同理，方法内的Object对象会将其引用放在变量槽中，而对象依然会在堆里面开辟空间存放。栈区中的操作数栈和C语言中的栈类似，加减乘除的临时变量会放在操作数栈中，所以又称之为表达式栈，表达式执行结束后操作数栈中的数据就已经被弹出栈，不在赘述。

当我们使用==对对象作比较时候，是检查如果两个操作数的值是否相等，如果相等则条件为真。而当对象是引用类型时候，常量池、栈中存放的是存放的是该引用对象的地址，若对象是两个对象，则对象在堆内的地址不同，故引用对象栈中的数据也是不同，也就不相等。虽然String类型也是引用类型，但是String类型和Java的八种基本数据类型一样，都实现了常量池技术，除非特别声明的包装类型，否则变量都会直接取自常量池，而不是通过堆中的对象获取。如果String是包装类型，改成如下代码：

**private** **String** name = new String("test");

我们可以看到内存分布会变成如下：



图三

Java自动装箱中还有一种“特殊”情况，之所以加引号，因为只是在使用时候看起来是相等，这个就是Integer类，例如：

Integer i1 = 127;

Integer i2 = 127;

Integer i3 = 128;

Integer i4 = 128;

System.out.println(i1==i2); *//true*

System.out.println(i3==i4); *//false*

i1、i2、i3、i4在赋值的时候会自动进行装箱，调用Integer的valueOf()方法但是valueOf()在(-128,128]内的时候会到IntegerCache.cache[]数组中取值，所以，当在这个范围内值是相等的，即i1==i2，因为是从数组中取的同一对象。而超出这个范围的值赋值会执行Integer i = new Integer(128)，故i3和i4不相等。

**SpringMVC和Struts**

SpringMVC和其他Spring Bean相同，默认是单例创建，所有方法“共享”成员变量， 基于此种设计，SpringMVC的请求是基于方法的。Spring这种管理方式的好处是我们不用重复创建对象，增加了对象创建的时间消耗和垃圾回收器的负担，但缺点就是一旦涉及到对成员变量的操作我们就必须考虑并发和锁的问题。而Struts刚好相反，Struts的创建默认是基于类的，每一个请求都会创建一个对象，但由于每个对象都是独立的，故我们在使用Struts时候不需要考虑对象成员变量的锁问题。同样的，Servlet也是单例模式，生命周期是一旦创建就存在，随着整个Servlet的存销。但这种单例模式我们在方法内只有局部变量的时候却无需考虑并发和锁问题，因为每个线程都有独立的栈帧，方法内的局部变量放在栈帧内，栈帧会随着线程的创建和销毁，方法的并发运行通常情况下并不会相互影响。所以我们在写SpringMVC和Servlet的时候为了避免导致问题更加复杂而应该尽量把变量控制在方法之内，尽量少的使用成员变量，而Struts则不存在这个问题，所有线程中对象的方法一般情况下都是顺序执行的。

**JDK1.8的改进**

在JDK1.8中，元空间替换永久代（方法区），1.7永久代默认的空间大小为64M，通过MaxPermSize参数调整。永久代的主要存在以下几个问题：

1. 永久代大小难以确定，很难进行调优，而且因为一旦在启动时候确定大小就固定下来，而且很容易导致溢出。
2. 永久代内存耗尽会触发Full GC，永久代的回收效率本来就较堆回收低，这样增加了GC的复杂度，也导致整个垃圾回收效率变低。

在1.8中移除了永久代的概念，用元空间代替，元空间将大部分数据都在本地内存中分配，默认情况下，类元数据只受可用的本地内存限制，新参数（MaxMetaspaceSize）用于限制本地内存分配给类元数据的大小。如果没有指定这个参数，元空间会在运行时根据需要动态调整。元空间的垃圾回收和内存管理由元空间虚拟机来完成，类及相关的元数据的生命周期与类加载器的一致，元空间虚拟机不会单独回收某个类，而是当GC发现某个类加载器不再存活了，会把相关的空间整个回收掉。