**JVM内存模型**

Java代码是运行在Java虚拟机(JVM)上的，Java虚拟机通过解释执行（解释器）或编译执行（编译器）来完成。

Java内存模型分为5个部分：方法区（Method Area），Java堆（Heap），Java栈（VM Stack），本地方法栈（Native Method Stack），程序计数器（PC 寄存器）

（图片来源：http://gityuan.com/images/jvm/jvm\_memory\_1.png）

线程共享区：

方法区（Method Area）：方法区是各个线程共享的区域，存放类信息，常量，静态常量，编译器编译后的代码等信息。

Java堆（Heap）：Java堆也是线程共享区域，类的实例存放在这里，一个系统会产生很多Java实例，因此Java堆的空间是最大的，如果Java堆的空间不足，就会抛出OutOfMemoryError异常。

线程私有区：

Java栈（VM Stack）：线程私有区域，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向栈里压一个元素，这个元素叫“栈帧”，栈帧中包含了方法中保存了该方法调用的参数、局部变量和返回地址等信息，如果栈空间不足了就会抛出StackOverflowError异常。

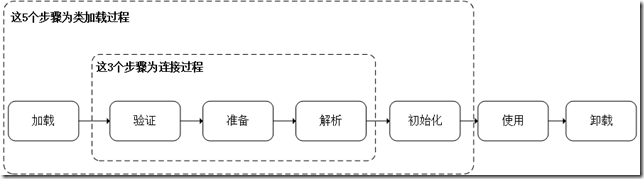
本地方法栈（Native Method Stack）：和Java栈类似，本地方法栈是用来执行本地方法的，存放的方法调用本地方法接口，最终调用本地方法库，实现与操作系统，硬件交互的目的。

程序计数器：这里对应的类以及加载，实例对象，方法，静态变量去了该去的地方，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。

**类加载机制**

 编写的java代码会通过编译器编译成字节编码的.class文件，再把字节编码加载到JVM中，映射到内存的各个区域中，程序就可以在内存中运行了。

类加载流程



（图片来源：<https://images2017.cnblogs.com/blog/352511/201708/352511-20170825174319746-900347526.png>）

加载

连接[验证 准备 解析 初始化 ]

使用

卸载

**1，加载**

加载是类装载的第一步，内存中生成一个代表这个类的java.lang.class对象，通过class文件的路径读取到二进制流，并解析二进制里的元数据（类型，常量等），作为方法区这个类的各种数据量的入口；这里的不一定从class文件获取，这里既可以从ZIP包（jar,war）包中获取，也在运行时动态生成（jsp转换成class文件，动态代理生成）。

**2，连接**

连接又可分为验证，准备，解析。

2.1，验证

验证主要是判断class文件的合法性，对版本号进行验证（例如如果使用java1.8编译后的class文件要再java1.6虚拟机上运行），还会对元数据，字节编码等进行验证，确保class文件里的字节流信息符合当前虚拟机的要求，不会危害虚拟机的安全。

2.2，准备

准备主要是分配内存，为变量分配初始值，即在方法区中分配这些变量所使用的内存空间，例如：

public static int i = 1；

在准备阶段i的值会被初始化为0，后面的类的初始化阶段才会赋值为1；

public static final int i = 1；

对应常量（static final）i，在准备阶段就会被赋值1；

2.3，解析

解析就是把代码中的符号引用替换为直接引用；例如某个类继承了java.lang.Object，原来的符号引用记录的是“java.lang.Object”，并不是java.lang,Object对象,直接引用就是找出对应的java.lang.Object对应的内存地址，建立直接引用关系；

**3，初始化**

初始化的过程包括执行类构造器方法，static变量赋值语句，static{}代码块，如果是一个子类进行初始化会先对其父类进行初始化，保证其父类在子类之前进行初始化；所以其实在java中初始化一个类，那么必然是先初始化java.lang.Object，因为所有的java类都继承自java.lang.Object。

以下几种情况不会执行类初始化：

* 通过子类引用父类的静态字段，只会触发父类的初始化，而不会触发子类的初始化。
* 定义对象数组，不会触发该类的初始化。
* 常量在编译期间会存入调用类的常量池中，本质上并没有直接引用定义常量的类，不会触发定义常量所在的类。
* 通过类名获取Class对象，不会触发类的初始化。
* 通过Class.forName加载指定类时，如果指定参数initialize为false时，也不会触发类初始化，其实这个参数是告诉虚拟机，是否要对类进行初始化。
* 通过ClassLoader默认的loadClass方法，也不会触发初始化动作。

**类加载器**

在JVM中有三中类加载器，BootStrap Classloader（启动Classloader）、Extension Classloader（扩展Classloader）和APP Classloader（应用Classloader）；

BootStrap ClassLoader主要加载JVM自身需要的类，这个加载器由C++编写是虚拟机的一部分，负责加载 JAVA\_HOME\lib 目录中的，或通过-Xbootclasspath参数指定路径中的，且被虚拟机认可（按文件名识别，如rt.jar）的类。

Extension Classloader是sun.misc.Launcher中的内部类ExtClassLoader，负责加载 JAVA\_HOME\lib\ext 目录中的，或通过java.ext.dirs系统变量指定路径中的类库。

APP ClassLoader是sun.misc.Launcher中的内部类AppClassLoader，负责加载用户路径上的类库。

用户也可以通过继承ClassLoader实现自己的类加载器。

**双亲委派模式**

当一个类加载器接收到类加载的任务时，会首先交给其父类加载器去加载，只有当父类加载器无法加载是其才会自己加载。其好处是可以避免一个类被重复加载。

即使两个类来源于相同的class文件，如果使用的类加载器不同，加载后的对象时完全不同的，这个不同反应在对象的 equals()、isAssignableFrom()、isInstance()等方法的返回结果，也包括了使用 instanceof 关键字对对象所属关系的判定结果。

**双亲委派模式的问题**

顶层ClassLoader，无法加载底层ClassLoader的类

Java框架(rt.jar)如何加载应用的类？

比如：javax.xml.parsers包中定义了xml解析的类接口  
Service Provider Interface SPI 位于rt.jar   
即接口在启动ClassLoader中。  
而SPI的实现类，在AppLoader。

这样就无法用BootstrapClassLoader去加载SPI的实现类。

**解决**

JDK中提供了一个方法：

1: Thread. setContextClassLoader()

用以解决顶层ClassLoader无法访问底层ClassLoader的类的问题；  
基本思想是，在顶层ClassLoader中，传入底层ClassLoader的实例。