**虚拟机类加载机制**

JVM把描述类的数据从class文件加载到内存中，并对数据进行验证、准备、解析和初始化，最终形成可以被JVM使用的Java类型，就是JVM的类加载机制。Java里面，类型的加载、连接和初始化都是在程序运行期间完成的。



* Java中的静态绑定和动态绑定

绑定：把一个方法与其所在的类/对象 关联起来叫做方法的绑定。

静态绑定：程序运行前方法已被绑定。在将java文件编译成class文件时，已经对其中的方法和域根据类信息进行了一次绑定，即Java中编译期进行的绑定。在Java中，final、private、static修饰的方法以及构造函数都是静态绑定的

动态绑定：在程序运行期间方法根据对象类型进行了绑定。是多态实现的前提。它通过方法表来实现：每个类被加载到虚拟机时，在方法区保存元数据，其中，包括一个叫做 方法表（method table）的东西，表中记录了这个类定义的方法的指针，每个表项指向一个具体的方法代码。如果这个类重写了父类中的某个方法，则对应表项指向新的代码实现处。从父类继承来的方法位于子类定义的方法的前面。

动态绑定的过程：

1.虚拟机提取对象实际类型的方法表

2.虚拟机搜索方法签名

3.调用方法

什么时候开始类的加载？

JVM规范并没有明确规定何时进行类的加载，但是碰到下面5种情况必须进行类的初始化。

* 遇到new(实例化对象)、getstatic(读取静态字段)、putstatic(设置静态字段)和invokestatic(调用静态方法)4条指令时，如果类没有进行初始化，则先触发初始化。
* 使用java.lang.reflect包的方法对类进行反射调用时，如果类没有进行初始化，则先触发其初始化。
* 当初始化一个类的时候，如果发现其父类还没有进行过初始化，则先触发其父类的初始化。
* JVM启动时标明的启动类，JVM会首先初始化这个主类。
* 使用JDK7的动态语言时，如果REF\_getStatic、REF\_putStatic、REF\_invokeStatic的方法句柄对应的类没有进行初始化，则必须先初始化。

1. 加载

加载是“类加载”（Class Loading）过程的一个阶段。主要做3件事：

* 通过一个类的全限定名来获取定义此类的二进制字节流。
* 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
* 在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。

3、验证

验证是链接阶段的第一步，目的是保证Class文件中的字节流数据符合当前JVM的要求，并且不会对JVM产生恶意的攻击。从整体看，验证阶段主要要下面4个阶段：

* 文件格式验证：验证Class文件的格式是否规范，保证输入的字节流能正确地被JVM解析并且存储于方法区内，其后的验证都是在方法区内对类的数据进行验证。
* 元数据验证：对字节码描述的类的元数据信息进行语义检查，保证其描述的信息符合Java语言规范的要求（例如这个类是否有父类，是否实现了所继承接口的所有方法，是否继承了final修饰的类等……）。
* 字节码验证：通过数据流和控制流分析，分析程序的执行过程是合法并且符合逻辑的。第二阶段是对类的元数据信息进行数据类型验证，这一阶段是对类的方法体进行检验分析。保证被检验类的方法在运行时不会脱离控制。
* 符号引用验证：该阶段发生在JVM将符号引用转换为直接引用的时候，即链接的解析阶段。目的是保证解析动作能正常秩序。（例如通过字符串全限定名是否能够找到该类，符号引用中的类、字段、方法的访问性质是否能被当前类访问等…….）

4、准备

为类变量分配内存并为类变量（static修饰的变量）设置初始值，初识值指的是数据类型的零值，因为这个时候还没有开始执行任何Java方法，而static修饰的变量的赋值是在类构造器<clinit>()方法中，所以赋值操作在初识化过程中。这些变量使用的内存将在方法区中分配。

5、解析

JVM将常量池中的符号引用替换为直接引用，java文件编译为class文件的过程中并没有链接步骤，方法调用在class文件中都是符号引用，而不是方法在运行时内存布局中的入口地址（直接引用）。

* 符号引用：用一组符号来描述所引用的目标，这些符号代表class文件中的常量池中记录了用来描述类、方法、接口等的字面量。
* 直接引用：可以直接指向目标的指针或者间接定位到目标的句柄，也就是目标在方法区里的入口地址。

6、初始化

类初始化是“类加载”的最后一个阶段，在准备阶段类变量已经被初始化为系统默认的零值，而在初始化阶段则是按照程序员的意愿为类变量赋值。这一阶段就是执行类构造器<clinit>()方法的过程。

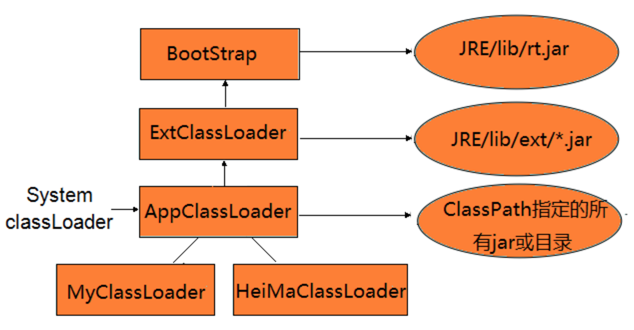
<clinit>()方法是由编译器自动收集类中所有类变量的赋值操作和静态语句块（static{}）中的语句合并产生的。

7、类加载器

类加载器用来实现类的加载过程，为所有被载入内存中的类生成一个java.lang.Class实例对象。一旦一个类被加载进内存，就不会再被加载。在Java中，一个类用其全限定类名（包括包名和类名）作为标识；但在JVM中，一个类用其全限定类名和其类加载器作为其唯一标识。系统提供的类加载器有三种：启动类加载器、扩展类加载器和系统类加载器。

* 启动类加载器(Bootstrap ClassLoader): 用来加载<JAVA\_HOME>\lib目录中并且能被JVM识别的类库到JVM内存中，如果名称不符合的类库即使放在lib目录中也不会被加载。该类加载器由C++(针对于HotSpot)实现，它不是ClassLoader子类，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。
* 扩展类加载器(Extension ClassLoader): 用来加载<JAVA\_HOME>\lib\ext或者被java.ext.dirs系统变量所指定的路径中的所有类库，由Java语言实现，开发者可以直接使用。
* 应用程序加载器(Application ClassLoader): 负责加载用户类路径(classpath)上的指定类库，我们可以直接使用这个类加载器。一般情况，如果我们没有自定义类加载器，程序默认就是用这个加载器。如果没有特别指定，则用户自定义的类加载器都以此类加载器作为父加载器。

8、双亲委派模型



工作原理：

如果一个类加载器收到了类加载请求，它并不会自己先去加载，而是把这个请求委托给父类的加载器去执行，如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归。因此，所有的加载请求最终都会到达顶层的启动类加载器，如果父类加载器可以完成类加载任务，就成功返回，倘若父类加载器无法完成此加载任务，子类加载器才会尝试自己去加载，这就是双亲委派模式。

优点：

双亲委派模式使得类的加载有了层级优先级，通过这种层级的优先级来保证加载过的类不会被重复加载，父类已经加载过的类，子类没有必要再去加载一次。

该模式可以防止核心API库被随意篡改，更加安全。因为核心类库只能由启动类加载器加载，假设通过其它路径向程序传递了一个java.lang.String类，通过双亲委派模式，这个类最终传递到启动类加载器，启动类在核心API库中找到了这个这个类，如果发现这个类以及被加载了，就不会加载这个类，而是直接返回一个Class对象，如果这个类还没有被加载，启动类加载器会从核心API库的路径中去加载这个类，然后返回一个Class对象。