五、类加载

- **把描述类的数据从class文件加载到内存中，并对数据进行格式转换、校验、解析、初始化并最终形成能够被JVM直接使用的java类型，这就是类加载的过程；**

1.加载

加载阶段要完成三件事：

- 获取定义此类或接口的二进制字节流；
- 将二进制字节流所代表的静态存储结构转换为方法区的运行时数据结构；
- 在方法区中生成一个代表这个类的java.lang.Class对象，作为方法区中这个类的各种数据的访问入口；

- "获取定义此类或接口的二进制字节流"并非由JVM内部完成，而是交由外部程序完成，完成这部分工作的代码成为类加载器。从JDK1.2开始就引入了三层类加载器和双亲委派模型的类加载架构并持续至今。

1.三层类加载器

- 启动类加载器；
- 应用程序类加载器；
- 扩展类加载器；
- 扩展类加载器之下还可以有一些用户自定义的类加载器（重写findClass()或loadClass()方法即可）；

2.双亲委派模型

- 当一个类加载器接收到一个类加载的请求时，首先不会去尝试加载这个类，而是把这个请求委派给它的父类加载器；
- 所有层次的类加载器都是如此；
- 一个类加载的请求最终会被委派到处于顶层的启动类加载器；
- 只有当父类加载器反馈无法完成这个类加载的请求时，子类加载器才会尝试去加载这个类。

优：

- java中的每一个类在任意类加载器中都能保证是同一个类；
- java中的所有类随着类加载器附带一种带有优先级的层次关系；

双亲委派模型被破坏的三次经历：

- 三层类加载器和双亲委派模型自JDK1.2引入，为了兼容JDK1.2之前用户自定义的类加载器，新增了一个使用protected修饰的findClass()方法；
- 为了解决双亲委派模型下基础类型无法回调用户代码的场景，引入了一个线程上下文类加载器Thread Context ClassLoader。实际是破坏了双亲委派模型的层次结构；
- 为了解决用户对程序代码动态性的追求（代码热替换、模块热部署），在JDK9中引入了一种OSGI技术，实现了模块化热部署；

2.验证

- 验证class文件中描述的字节流信息是否符合《JAVA虚拟机规范》、java语言规范。细分为：

1.文件格式验证：

- 验证该文件是否符合class文件格式规范的全部约束；

2.元数据验证：

- 对其描述的字节流信息进行语义分析，确保符合java语言规范；

3.字节码验证：

- 通过数据流分析和控制流分析，确保其描述的程序语义是合法的、符合逻辑的；
- 主要是对方法的方法体进行验证，确保其运行期间不会做出危害虚拟机自身安全的行为；
- 在JDK6中引入了StackMapTable属性表，属性表中记录了方法体的所有基本块开始时在局部变量表和操作数栈中的应有状态，从而将类型推到转变为类型检查，极大的节省了验证时间；

4.符号引用验证：

- 实际发生在解析阶段，主要是对类自身以外的各类信息进行匹配性校验；

3.准备

- 为class文件中的静态变量、实例变量分配内存空间并赋零值。比如整型对应0或0l，浮点型对应0.0f或0.0d，boolean型对应false；
- 如果一个静态变量使用final修饰，因为在前端编译器编译期间就将该变量值写入到class文件的字段表集合对应字段表的ConstantValue属性表中，所以准备阶段赋的初值就是ConstantValue属性表中的值；

4.解析

- 负责将符号引用转换为直接引用；
- 符号引用使用一组符号描述所引用的目标，被引用的目标不要求一定被加载到内存中；
- 直接引用是一个能够直接定位到目标的直接指针、相对偏移量或间接定位到目标的句柄，被引用的目标一定要求被加载到内存中；
- 对一个符号引用进行多次解析是很常见的事情，除invokeDynamic指令外，JVM会把第一次解析的结果进行缓存；
- invokeDynamic指令除外的原因是，该指令对应的引用称为"动态调用点限定符"，这里的"动态"要求程序必须加载到这条指令时，解析动作才能开始；
- 解析包括对类、接口、字段、类方法、接口方法、方法句柄、方法类型、动态常量八种类型的解析；

5.初始化

- 负责执行clinit()方法；
- clinit()方法由编译器自动收集类文件中静态变量的赋值动作、静态代码块语句合并而来； 如果一个类文件中没有静态代码块，也没有静态变量的赋值动作，就不会生成clinit()方法；

执行clinit()方法要求

- 在执行子类的clinit()方法之前，必须先执行父类的clinit()方法；
- 在执行子接口的clinit()方法之前，并不要求先执行父接口的clinit()方法；
- JVM保证clinit()方法在多线程环境中被正确的枷锁同步；

《JAVA虚拟机规范》严格规定当且仅当以下六种情况发生时，必须触发类型的初始化阶段：

遇到new、getStatic、putStatic、invokeStatic四条字节码指令时，如果对应的类型还没有初始化，必须触发类型的初始化阶段； 四条字节码指令对应的代码场景：

- 使用new关键字实例化对象；
- 访问静态方法；
- 访问或设置静态变量（不包括使用final修饰的静态变量）；

- 使用java.lang.reflect包中的方法对类型进行反射调用时，如果对应的类型还没有初始化，必须立即触发类型的初始化阶段；
- JVM启动时，需要指定一个主类，必须首先触发主类的初始化阶段；
- 接口中至少有一个方法使用default关键字修饰，触发接口实现类初始化阶段之前，必须触发接口的初始化阶段；
- 触发子类的初始化阶段之前，必须触发父类的初始化阶段；
- 使用JDK7新加入的动态语言支持时，如果一个java.lang.invoke.MethodHandler实例的最后解析结果为ref_getStatic、ref_putStatic、ref_invokStatic、ref_newInvokeSpecial四种类型的方法句柄时，如果对应的类型没有初始化，必须触发对应类型的初始化阶段；

6.使用

- 按照程序员的意愿正常使用；

7.卸载

类型的卸载发生在方法区，需要同时满足三个条件：

- 该类的类加载器已经被卸载；
- 该类的所有实例已经被卸载；
- 该类对应的java.lang.class对象没有在任何地方被引用，也无法通过反射访问该类的任何方法；

- 满足这三个条件仅能说明该类型可以被卸载，但并不保证一定会被卸载；