**[深入理解JVM(九)——类加载的过程](https://blog.csdn.net/u010425776/article/details/51254858)**

<https://blog.csdn.net/u010425776/article/details/51254858>

通过之前的介绍可知，类加载过程共有5个步骤，分别是：加载、验证、准备、解析、初始化。其中，验证、准备、解析称为连接。下面详细介绍这5个过程JVM所做的工作。

**加载**

注意：“加载”是“类加载”过程的第一步，千万不要混淆。

**1. 加载的过程**

在加载过程中，JVM主要做3件事情：

1. 通过一个类的全限定名来获取这个类的二进制字节流，即class文件：   
   在程序运行过程中，当要访问一个类时，若发现这个类尚未被加载，并满足类初始化时机的条件时，就根据要被初始化的这个类的全限定名找到该类的二进制字节流，开始加载过程。
2. 将二进制字节流的存储结构转化为特定的数据结构，存储在方法区中；
3. 在内存中创建一个java.lang.Class类型的对象：   
   接下来程序在运行过程中所有对该类的访问都通过这个类对象，也就是这个Class类型的类对象是提供给外界访问该类的接口。

**2. 从哪里加载？**

JVM规范对于加载过程给予了较大的宽松度。一般二进制字节流都从已经编译好的本地class文件中读取，此外还可以从以下地方读取：

* 从压缩包中读取   
  如：Jar、War、Ear等。
* 从其它文件中动态生成   
  如：从JSP文件中生成Class类。
* 从数据库中读取   
  将二进制字节流存储至数据库中，然后在加载时从数据库中读取。有些中间件会这么做，用来实现代码在集群间分发。
* 从网络中获取   
  从网络中获取二进制字节流。典型就是Applet。

**3. 类 和 数组加载过程的区别？**

数组也有类型，称为“数组类型”。如：

String[] str = new String[10];

* 1

这个数组的数组类型是Ljava.lang.String，而String只是这个数组中元素的类型。

当程序在运行过程中遇到new关键字创建一个数组时，由JVM直接创建数组类，再由类加载器创建数组中的元素类。

而普通类的加载由类加载器完成。既可以使用系统提供的引导类加载器，也可以使用用户自定义的类加载器。

**4. 加载过程的注意点**

1. JVM规范并未给出类在方法区中存放的数据结构   
   类完成加载后，二进制字节流就以特定的数据结构存储在方法区中，但存储的数据结构是由虚拟机自己定义的，JVM规范并没有指定。
2. JVM规范并没有指定Class对象存放的位置   
   在二进制字节流以特定格式存储在方法区后，JVM会创建一个java.lang.Class类型的对象，作为本类的外部接口。既然是对象就应该存放在堆内存中，不过JVM规范并没有给出限制，不同的虚拟机根据自己的需求存放这个对象。HotSpot将Class对象存放在方法区。
3. 加载阶段和连接阶段是交叉的   
   通过之前的介绍可知，类加载过程中每个步骤的开始顺序都有严格限制，但每个步骤的结束顺序没有限制。也就是说，类加载过程中，必须按照如下顺序开始：   
   加载、连接、初始化，但结束顺序无所谓，因此由于每个步骤处理时间的长短不一就会导致有些步骤会出现交叉。

**验证**

验证阶段比较耗时，它非常重要但不一定必要，如果所运行的代码已经被反复使用和验证过，那么可以使用-Xverify:none参数关闭，以缩短类加载时间。

**1. 验证的目的是什么？**

验证是为了保证二进制字节流中的信息符合虚拟机规范，并没有安全问题。

**2. 为什么需要验证？**

虽然Java语言是一门安全的语言，它能确保程序猿无法访问数组边界以外的内存、避免让一个对象转换成任意类型、避免跳转到不存在的代码行，如果出现这些情况，编译无法通过。也就是说，Java语言的安全性是通过编译器来保证的。

但是我们知道，编译器和虚拟机是两个独立的东西，虚拟机只认二进制字节流，它不会管所获得的二进制字节流是哪来的，当然，如果是编译器给它的，那么就相对安全，但如果是从其它途径获得的，那么无法确保该二进制字节流是安全的。通过上文可知，虚拟机规范中没有限制二进制字节流的来源，那么任意来源的二进制字节流虚拟机都能接受，为了防止字节流中有安全问题，因此需要验证！

**3. 验证的过程**

1. 文件格式验证   
   这个阶段主要验证输入的二进制字节流是否符合class文件结构的规范。二进制字节流只有通过了本阶段的验证，才会被允许存入到方法区中。   
   本验证阶段是基于二进制字节流的，而后面的三个验证阶段都是在方法区中进行，并基于类特定的数据结构的。   
     
   通过上文可知，加载开始前，二进制字节流还没进方法区，而加载完成后，二进制字节流已经存入方法区。而在文件格式验证前，二进制字节流尚未进入方法区，文件格式验证通过之后才进入方法区。也就是说，加载开始后，立即启动了文件格式验证，本阶段验证通过后，二进制字节流被转换成特定数据结构存储至方法区中，继而开始下阶段的验证和创建Class对象等操作。这个过程印证了：加载和验证是交叉进行的。
2. 元数据验证   
   本阶段对方法区中的字节码描述信息进行语义分析，确保其符合Java语法规范。
3. 字节码验证   
   本阶段是验证过程的最复杂的一个阶段。   
   本阶段对方法体进行语义分析，保证方法在运行时不会出现危害虚拟机的事件。
4. 符号引用验证   
   本阶段验证发生在解析阶段，确保解析能正常执行。

**准备**

准备阶段完成两件事情：   
1. 为已经在方法区中的类中的静态成员变量分配内存   
类的静态成员变量也存储在方法区中。   
2. 为静态成员变量设置初始值   
初始值为0、false、null等。

示例1：

public static String name = "柴毛毛";

* 1

在准备阶段，JVM会在方法区中为name分配内存空间，并赋上初始值null。   
给name赋上”柴毛毛”是在初始化阶段完成的。

示例2:

public static final String name = "柴毛毛";

* 1

被final修饰的常量如果有初始值，那么在编译阶段就会将初始值存入constantValue属性中，在准备阶段就将constantValue的值赋给该字段。 

**解析**

解析阶段是虚拟机将常量池中的符号引用替换为直接引用的过程。 

**初始化**

初始化阶段就是执行类构造器clinit()的过程。   
clinit()方法由编译器自动产生，收集类中static{}代码块中的类变量赋值语句和类中静态成员变量的赋值语句。在准备阶段，类中静态成员变量已经完成了默认初始化，而在初始化阶段，clinit()方法对静态成员变量进行显示初始化。

**初始化过程的注意点**

1. clinit()方法中静态成员变量的赋值顺序是根据Java代码中成员变量的出现的顺序决定的。
2. 静态代码块能访问出现在静态代码块之前的静态成员变量，无法访问出现在静态代码块之后的成员变量。
3. 静态代码块能给出现在静态代码块之后的静态成员变量赋值。
4. 构造函数init()需要显示调用父类构造函数，而类的构造函数clinit()不需要调用父类的类构造函数，因为虚拟机会确保子类的clinit()方法执行前已经执行了父类的clinit()方法。
5. 如果一个类/接口中没有静态代码块，也没有静态成员变量的赋值操作，那么编译器就不会生成clinit()方法。
6. 接口也需要通过clinit()方法为接口中定义的静态成员变量显示初始化。
7. 接口中不能使用静态代码块。
8. 接口在执行clinit()方法前，虚拟机不会确保其父接口的clinit()方法被执行，只有当父接口中的静态成员变量被使用到时才会执行父接口的clinit()方法。
9. 虚拟机会给clinit()方法加锁，因此当多条线程同时执行某一个类的clinit()方法时，只有一个方法会被执行，其它的方法都被阻塞。并且，只要有一个clinit()方法执行完，其它的clinit()方法就不会再被执行。因此，在同一个类加载器下，同一个类只会被初始化一次。