参考：

JVM 类加载机制详解

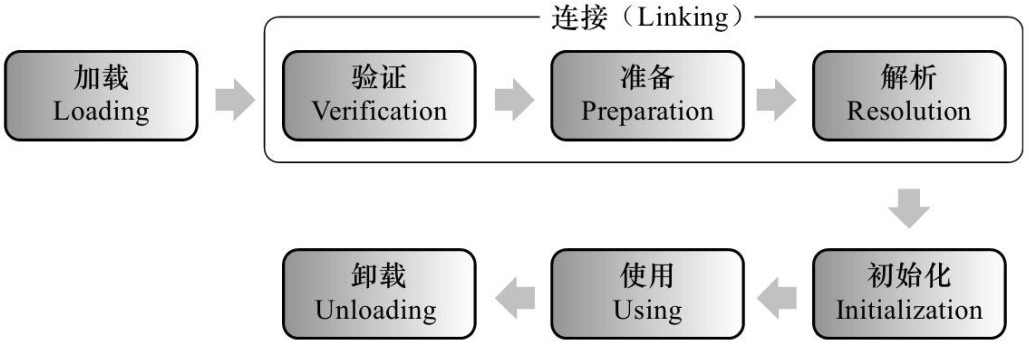
《深入理解Java虚拟机_JVM高级特性与最佳实践 第2版》

前言

类加载机制算是JVM层面最重要的东西了，我们的类在JVM使用都是过的这一关。那么这个JVM的层面的东西为什么要去了解呢。通俗的来说，大多数Java开发人员在较长的时间周期内可能都不会在实际开发中真正有能力或有机会能去接触JVM的调优。但是呢，Java开发人员怎么能没有梦想呢，像JVM这样的东西肯定得去克服的啦。嗯，其实只是面试喜欢问而已，汗。

类加载过程

先上张图。



这整个过程也就是类的生命周期了，类加载过程就是由加载起步到初始化结束。接下来就以图示过程依次讲解吧。

加载

我们的机制是类加载，这边只是其中的一个步骤加载，注意不要混淆了。

加载阶段JVM干了三件事情：

1.通过类的全限定类名获取定义此类的二进制字节流；

2.将这个字节流所代表的静态存储结构转化为方法区（概念参考上篇文章）的运行时数据结构；

3.在内存中生成一个代表此类的java.lang.Class对象，作为方法区这个类的各种数据结构的访问入口。

注意第一件事情说二进制字节流，却未指明具体来源，所以开放性很大，可以从注入zip包（jar包，war包），网络（aaplet），运行时计算生成（动态代理），其它文件生成（jsp），数据库等等地方获取。

验证

验证阶段是连接这个大阶段中的一个小阶段，这一阶段是为了确保Class文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。

从整体上看大致分为4个阶段的校验动作：文件格式验证、元数据验证、字节码验证、符号引入验证。

从这个阶段的作用来看，这是非常重要的一步，但从它的功能来说，也不是必不可少的，如果全部代码已经反复使用和验证过，那么说明不更改就不会有问题，是可以略过这一步的，通过-Xverify:none可以关闭大部分的类验证措施。

准备

准备阶段正式为类变量（被static修饰的变量）分配内存并设置类变量初始值，这些变量使用的内存都在方法区进行分配，初始值通常情况是对应数据类型的零值。例如：

```
public static int value = 123;
```

这时候初始值为0而不是123，因为此时准备阶段还并未执行任何Java方法，把value赋值123的是putstatic指令，存放在类构造器<clinit>方法中，所以将value赋值123是在初始化阶段才进行。当然也有一种特殊情况，value在准备阶段就被赋值123，例如：

```
public static final int value = 123;
```

这种情况我们通俗讲就是常量，在JVM层面怎么理解它在准备阶段就初始为123了呢？修饰final后类字段的字段属性就存在了ConstantValue属性，那么准备阶段就会将变量value初始化为ConstantValue所指定的值。

解析（待深入）

解析阶段是JVM将常量池内的符号引用替换为直接引用的过程。符号引用就是class文件中CONSTANT_Class_info，CONSTANT_Field_info，CONSTANT_Method_info等类型的常量。

直接引用和符号引用又有什么关联呢？

符号引用与虚拟机实现的布局无关，引用的目标并不一定要已经加载到内存中。各种虚拟机实现的内存布局可以各不相同，但是它们能接受的符号引用必须是一致的，因为符号引用的字面量形式明确定义在Java虚拟机规范的Class文件格式中。

直接引用可以是指向目标的指针，相对偏移量或是一个能间接定位到目标的句柄。如果有了直接引用，那引用的目标必定已经在内存中存在。

初始化

初始化阶段是类加载过程的最后一步，这一步才算真正开始执行类中定义的Java程序代码（或者说字节码）。

初始化阶段实则就是执行类构造器<clinit>方法的过程。<clinit>方法是由编译器自动收集类中的所有类变量的赋值动作和静态语句块（static{}块）中的语句合并产生的，静态语句块只能访问定义在它之前的变量。JVM会保证子类<clinit>方法被执行前，父类的<clinit>方法已经执行过了，所以父类的静态语句块要优先于子类的。但<clinit>方法并不是必须的，若类和接口中没有静态语句块和对变量赋值（当然，接口没有静态语句块），那么就不会为这个类和接口生成<clinit>方法。JVM能够保证一个类的<clinit>方法在多线程环境中被正确加锁、同步。

类加载器

这边又得提到上述的加载阶段了，加载阶段的第一步就是通过类的全限定类名获取定义此类的二进制字节流，这一动作是放在JVM外部实现的，以便让应用程序自己决定如何去获取所需要的类，然后就要涉及到类加载器这个概念了。

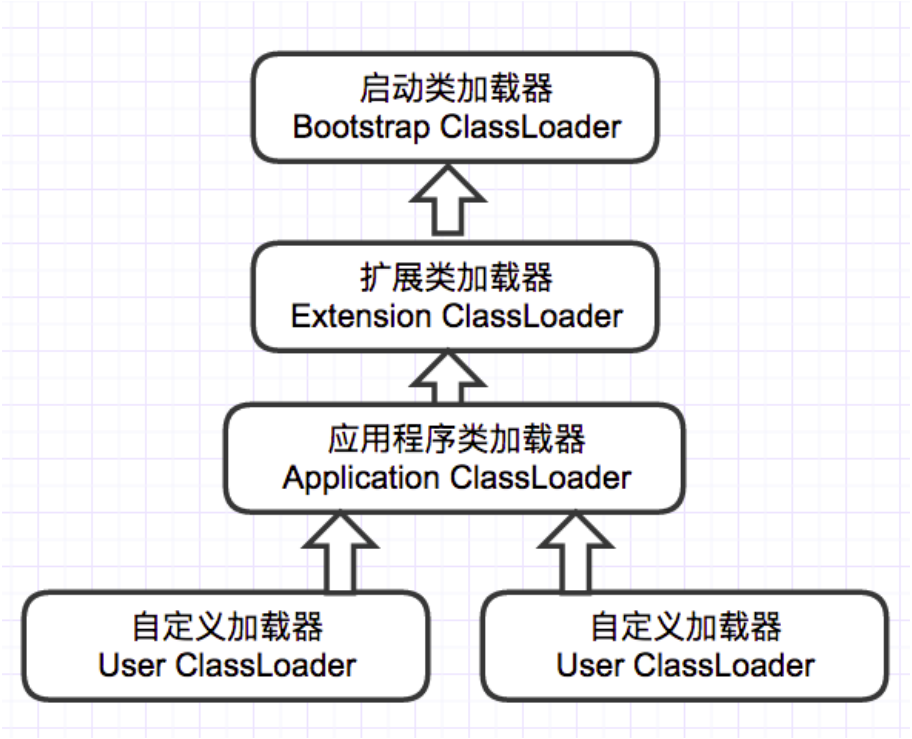
从开发人员角度来看，系统提供了三种类加载器：

启动类加载器（Bootstrap ClassLoader）：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。

扩展类加载器（Extension ClassLoader）：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。

应用程序类加载器(Application ClassLoader)：负责加载用户路径（classpath）上的类库。

当然，有需求的话还可以自定义类加载器。这么多加载器，他们是个怎么样的联系呢？



图片是这样，但千万不要误会成这是继承的关系，他们的联系只是层级，这种联系我们有个术语来描述，双亲委托模型。那么，这边的双亲委托模型到底起了什么作用呢，我们调用任何类加载器，首先会让上层的类加载器去加载，若加载不了才会自己去加载，比如我们现在调用了自定义加载器，也只有自定义加载器能够加载，但这里会先依次委托到启动类加载器，然后一层层下来依次尝试加载不成，最后自定义加载器才会自己加载。

这有什么好处呢？比如我们要加载Object类，如果用了双亲委托模型，总是会由启动类加载器去加载，但是若没有用的话，用各个类加载器自行加载，此时若是编写了一个Object类，那么系统中就会有多个Object类，这就会一片混乱。