第7章 虚拟机类加载 机制

7.1 概述

虚拟机把描述类的数据从文件加载到==内存==，并对数据进行==校验、转换解析和初始化==，最终形成可以被虚拟机直接使用的Java类型，这就是 虚拟机的类加载机制。

在Java语言里，类型的加载、连接和初始化过程都是在程序运行期间完成的，虽然会令类加载时稍微增加一些性能开销为Java程序提供了高度的灵活性。Java里天生可以动态扩展的语言特性就是依赖运行期动态加载和动态连接这个特点实现的。

7.2 类加载的时机

类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括：==加载、验证、准备、解析、初始化、使用和卸载==7个阶段。==验证、准备、解析==3部分统称为==连接==。

==加载、验证、准备、初始化、卸载==这5个阶段的==顺序是确定==的。类的加载过程必须按照这种顺序按部就班地开始，而==解析==阶段不一定：==它在某些特殊情况下可以在初始化阶段之后再开始，这是为了支持Java语言的运行时绑定==，通常会在一个阶段执行的过程中调用、激活另外一个阶段。

Java虚拟机规范中==没有对加载阶段进行强制约束==，这点可以交给虚拟机的具体实现来自由把握，但对于初始化阶段，==虚拟机规定了5种情况必须立即对类进行“初始化”==：

1. 遇到new、getstatic、putstatic、invokestatic4条字节码指令时，如果没有进行过初始化，则需要先触发其初始化。
2. 使用java.lang.reflect包的方法对类进行反射调用的时候，如果类没有进行过初始化，则需要先触发其初始化。
3. 当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化。
4. 当虚拟机启动时，用户需要指定一个要执行的主类（包含main()方法的那个类），虚拟机会先初始化这个主类。
5. 当使用JDK1.7的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的解析结果REF_getStatic、REF_putStatic、REF_invokeStatic的方法句柄，并且这个方法句柄所对应的类没有进行过初始化，则需要先触发其初始化。

这5种场景中的行为称为对一个类进行==主动引用==。除此之外，所有引用类的方式都不会触发初始化，称为==被动引用==。

==对于静态字段，只有直接定义这个字段的类才会被初始化，通过子类来引用父类中定义的静态字段，只会触发父类的初始化而不会触发子类的初始化。==

==通过数组定义来引用类，不会触发此类的初始化。==

==常量在编译阶段会存入调用类的常量池中，本质上并没有直接引用到定义常量的类，因此不会触发定义常量的类的初始化。==

接口的加载过程和类加载过程有些不同，接口中不能使用静态块，但是编译器仍会为其产生“()”，雷雨接口真正的区别是前述5种必须加载情况中的第三种：当一个类在初始化时，要求其父类全部都已经初始化过了，但是一个接口在初始化时，并不需要其父接口全部都完成了初始化，只有在真正用到父接口的时候才会初始化。

7.3 类加载的过程

7.3.1 加载

==加载是类加载过程的一个阶段==，在加载阶段，虚拟机完成==如下3件事情==：

1. 通过一个类的全限定名来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的运行时数据结构
3. 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。

==可以获取到Class文件的二进制字节流==有：

1. 从ZIP包中读取
2. 从网络中获取
3. 运行时计算生成
4. 由其他文件生成
5. 从数据库中读取

一个非数组类的加载阶段是开发人员可控性最强的，开发人员可以通过定义自己的类加载器去控制字节流的获取方式（即重写一个类加载器的loadClass()方法）。

==一个数组类C的创建遵循如下规则==：

- 如果数组的组件类型（数组去掉一个维度的类型）是引用类型，那就递归采用本节中定义的加载过程去加载这个组件类型，数组C将在加载该组件类型的类加载器的类名称空间上被标识。
- 如果数组的组件类型不是引用类型，Java虚拟机将把数组C标记为域引导类加载器关联。
- 数组类的可见性与它的组件类型的可见性一致，如果组件类型不是引用类型，那数组类的可见性将默认为public。

方法区中的数据格式由虚拟机实现自行定义，虚拟机规范并未规定此区域的具体数据结构。然后在内存中实例化一个java.lang.Class对象（并==没有明确是在Java堆中==，对于HotSpot，Class对象创建在方法区）这个对象将作为程序访问方法区中的这些类型数据的外部接口。

加载阶段和连接阶段的部分内容是交叉执行的。

7.3.2 验证

验证是连接状态的第一步，这阶段的目的是为了确保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。

如果验证到输入的字节流不符合Class文件格式的约束，虚拟机就应抛出一个java.lang.VerifyError异常或其子类异常。

验证阶段会完成下面4个阶段的检验动作：文件格式验证、元数据验证、字节码验证、符号引用验证。

1.文件格式验证

第一阶段要验证字节流是否符合Class文件格式的规范，并且能被当前版本的虚拟机所处理：

1. 是否以魔数0xCAFEBABE开头
2. 主、次版本号是否在当前虚拟机处理范围之内
3. 常量池的常量中是否有不被支持的常量类型
4. 指向常量的各种索引值中是否有不存在的常量或不符合类型的常量

5. CONSTANT_Utf8_info型的常量中是否有不符合UTF8编码的数据
6. Class文件中各个部分及文件本身是否有被删除或者附加的其他信息
7. ...

该阶段的主要目的是保证输入的字节流能正确地解析并存储于方法区之内，格式上符合描述一个Java类型信息的要求。这阶段的验证是基于二进制字节流进行的，只有通过了这个阶段的验证后，字节流才会进入内存的方法区进行存储，所以后面的3个验证阶段全部是基于方法区的存储结构进行的，不会再直接操作字节流。

2.元数据验证

对字节码描述的信息进行语义分析，以保证其描述信息符合Java语言规范的要求，这个阶段可能包含的验证点如下：

- 这个类是否有父类
- 这个类的父类是否继承了不允许被继承的类
- 如果这个类不是抽象类，是否实现了其父类或接口之中要求实现的所有方法
- 类中的字段、方法是否与父类产生矛盾
- ...

第二阶段的主要目的是对类的元数据信息进行语义校验，保证不存在不符合Java语法规范的元数据信息。

3.字节码验证

第三阶段的主要目的是通过数据流和控制流分析，确定程序语义是合法的、符合逻辑的。这个阶段将对类的方法体进行校验分析，保证被校验类的方法在运行时不会做出危害虚拟机安全的事件。例如：

- 保证任意时刻操作数栈的数据类型与指令代码序列都能配合工作
- 保证跳转指令不会跳转到方法体以为的字节码指令上
- 保证方法体中的类型转化是有效的
- ...

一个方法通过了字节码验证，也不能说明其一定就是安全的。

StackMapTable属性描述了方法体中所有的基本块，开始时本地变量表和操作栈应有的状态，在字节码验证期间，就不需要根据程序推导这些状态的合法性，只要检查StackMapTable属性中的记录是否合法。

HotSpot虚拟机中提供了-XX:-UserSplitVerifier选项来关闭这项优化，或者使用参数-XX:+FailOverToOldVerifier要求在类型校验失败的时候退回到就得类型推导方式进行校验。

4.符号引用验证

最后一个阶段的校验发生在虚拟机将符号引用转化为直接引用的时候，这个转化动作将在连接的第三阶段——解析阶段中发生。符号引用验证可以看做事对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验，通常需要校验如下内容：

- 符号引用中通过字符串描述全限定名是否能找到对应的类
- 在指定类中是否存在符合方法的字段描述符以及简单名称所描述的方法和字段。
- 符号引用中的类、字段、方法的访问性是否可以被当前类访问

符号引用验证的目的是确保解析动作能正常执行。

7.3.3 准备

准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些变量所使用的内存都将在方法区进行分配。这时候进行内存分配的仅包括类变量（static变量），而不包括实例变量，实例变量将会在对象实例化时随着对象一起分配在Java堆中，其次，这里所说的初始值“正常情况”下是指数据类型的零值。

如果类字段的字段属性表中存在ConstantValue属性，那在准备阶段变量value就会被初始化为ConstantValue属性所指定的值。

7.3.4 解析

解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。

符号引用：符号引用以一组符号来描述所引用的目标，符号可以是任何形式的字面量，只要使用时能无歧义地定位到目标即可。符号引用和虚拟机实现的内存布局无关，引用的目标并不一定已经加载到内存中。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都是一致的，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：直接引用可以是直接指向目标的指针、相对偏移量或是一二能间接定位到目标的句柄。直接引用是和虚拟机实现的内存布局相关的，同一个符号引用在不同虚拟机实例上翻译出来的直接引用一般会不同。如果有了直接引用，那引用的目标必定已经在内存中存在了。

虚拟机只要求了在执行anewarray、checkcast、getfield、getstatic、instanceof、invokedynamic、invokeinterface、invokespecial、invokestatic、invokevirtual、lde、ldc_w、multianewarray、new、putfield和putstatic这16个用于操作符号引用的字节码指令之前，先对它们所使用的符号引用进行解析。

对同一个符号引用进行多次解析请求是很常见的事，除invokedynamic指令之外，虚拟机实现可以对第一次解析的结果进行缓存，从而避免解析动作重复执行。但需要保证的是在同一实体中，如果一个符号引用之间已经被成功解析过，那么后续的引用解析请求就应当一直成功；同样的，如果第一次解析失败了，那么其他指令对这个符号的解析请求也应当收到相同的异常。

对于invokedynamic指令，其本身就是用于动态语言支持，它所对应的引用称为“动态调用点限定符”，这里“动态”的含义是必须等到程序实际运行到这条指令的时候，解析动作才能执行。相对的，其余可触发解析的指令都是“静态”的，可以在刚刚完成加载阶段，还没有开始执行代码时就进行解析。

解析动作主要针对类或接口、字段、类方法、接口方法、方法类型、方法句柄和调用点限定符7类符号引用。

1.类或接口的解析

假设当前代码所处的类为D，如果要把一个从未解析过的符号引用N解析为一个类或接口C的直接引用，那虚拟机完成整个接续的过程需要以下3个步骤：

1. 如果C不是一个数组类型，那虚拟机将会把代表N的全限定名传递给D的类加载器去加载这个类C。
2. 如果C是一个数组类型，并且数组的元素类型为对象，那将会按照第一点加载数组元素类型，接着由虚拟机生成一个代表此数组维度和元素的数组对象。
3. 如果上面的步骤没有出现任何异常，那么C在虚拟机中实际上已经成为一个有效的类或接口了，但在解析完成之前还要进行符号引用验证，确认D是否具备对C的访问权限，如果发现不具备访问权限，将抛出java.lang.IllegalAccessError异常。

2.字段解析

要解析一个未被解析过的字段符号引用，首先将会对字段表内class_index项中索引CONSTANT_Class_info符号引用进行解析，也就是字段所属的类或接口的符号引用。将这个字段所属的类或接口用C表示，虚拟机规范要求按照如下步骤对C进行后续字段的搜索：

1. 如果C本身就包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。
2. 否则，如果在C中实现了接口，将会按照继承关系从下往上递归搜索各个接口和它的父接口，如果接口中包含了简单名称和字段描述都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。
3. 否则，如果C不是java.lang.Object的话，将会按照继承关系从下往上递归搜索其父类，如果在弗雷中包含了简单名称和字段描述符都与目标相匹配的字段，则返回这个字段的直接引用，查找结束。
4. 否则，查找失败，抛出java.langNoSuchFieldError异常。

3.类方法解析

类方法解析的第一个步骤与字段解析一样，也需要先解析出类方法表的class index项中索引的方法所述的类或接口的符号引用。如果解析成功用C表示这个类，按照接下来的步骤进行后续的类方法搜索：

1. 类方法和接口方法符号引用的常量类型定义是分开的，如果在类方法表中发现class_index中索引的C是个接口，那就直接抛出java.lang.IncompatibleClassChangeError异常。
2. 如果通过了第一步，在类C中查找是否有简答名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接已用，查找结束。
3. 否则在类C的父类中递归查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。
4. 否则在类C实现的接口列表及它们的父接口之中递归查找是否有简单名称和描述符都和目标相匹配的方法，如果存在匹配的方法，说明类C是一个抽象类，这时查找结束，抛出java.lang.IlleagalAccessError异常。
5. 否则宣告方法查找失败，抛出了java.lang.NoSuchMethodError异常。

4.接口方法解析

接口方法也需要县解析出接口方法表的class_index项中索引的方法所属的类或接口的符号引用，如果解析成功，依然用C表示这个接口，接下来按照后续步骤查找：

1. 与类方法解析不同，如果在接口方法表中发现class_index中的索引C是个类而不是接口，那就直接抛出java.lang.IncompatibleClassChangeError异常
2. 否则，在接口C中查找是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。
3. 否则，在接口C的父接口中递归查找，直到java.lang.Object类为止，看看是否有简单名称和描述符都与目标相匹配的方法，如果有则返回这个方法的直接引用，查找结束。
4. 否则，宣告方法查找失败，抛出java.lang.NoSuchMethodError异常。

7.3.5 初始化

类初始化阶段是类加载阶段的最后一步。到了初始化阶段，才真正开始执行类中定义的Java程序代码。

在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则根据程序员通过程序制定的主观计划去初始化类变量和其他资源。初始化阶段是执行类构造器()方法的过程。

()方法是由编译器自动收集类中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，编译器收集的顺序是由语句在源文件中出现的顺序所决定的，静态语句块中只能访问到定义在静态语句块之前的变量，定义在它之后的变量，在前面的静态语句块可以赋值，但不能访问。

()方法与类的构造函数不同，它不需要显示调用父类构造器，虚拟机会保证在子类的()方法执行之前，父类的()方法已经执行完毕。因此在虚拟机中第一个被执行的()方法的类肯定是java.lang.Object。

由于父类的()方法先执行，也就意味着父类中定义的静态语句块要优先于子类的变量赋值操作。

()方法对于类或接口来说并不是必须的，如果一个类中没有静态语句块，也没有对变量的赋值操作，那么编译器可以不为这个类生成()方法。

接口中不能使用静态语句块，但仍然有变量初始化的赋值操作，因此接口与类一样都会生成()方法。但接口与类不同的是，执行接口的()方法不需要先执行父接口的()方法。只有当父接口中定义的变量使用时，父接口才会初始化。另外，接口的实现类在初始化时也一样不会执行接口的()方法。

虚拟机会保证一个类的()方法在多线程环境中被正确地加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的()方法，其它线程都需要阻塞等待，知道活动线程执行()方法完毕。如果在一个类的()方法中有耗时很长的操作，就可额能造成多个进行阻塞。

7.4 类加载器

类加载器通过一个类的全限定名来获取描述此类的二进制字节流。

7.4.1 类和类加载器

对于任何一个类，都需要由加载它的类加载器和这个类本身一同确立其在Java虚拟机的唯一性，每一个类加载器，都拥有一个独立的类名称空间。

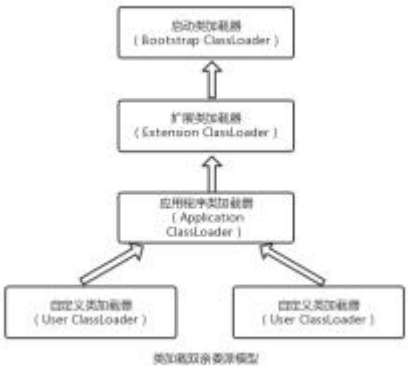
7.4.2 双亲委派模型

从Java虚拟机的角度来说，只存在两种不同的类加载器：

1. 启动类加载器(Bootstrap ClassLoader)，这个类加载器使用C++实现，是虚拟机自身的一部分。
2. 其他的类加载器，这个类加载器都由Java语言实现，独立于虚拟机外部，并且全部继承自抽象类java.lang.ClassLoader。

绝大部分Java程序都会使用到如下3种系统提供的类加载器：

- 启动类加载器，负责将存放在<JAVA_HOME>\lib目录中的，或者被-Xbootclasspath参数所指定的路径中的，并且被虚拟机识别的类库加载到虚拟机内存中。
- 扩展类加载器，负责加载<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，开发者可以直接使用扩展类加载器。
- 应用程序类加载器，一般也称为系统类加载器。负责加载用户类路径上所指定的类库。如果应用程序中没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。



这种层次关系称为类加载器的双亲委培模型。双亲委派模型要求除了顶层的启动类加载器以外，其余的类加载器都应该有自己的父类加载器。

双亲委派模型的工作过程是：如果一个类加载器收到了类加载请求，首先刽自己去尝试加载这个类，而是吧这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求时，子加载器才会去尝试自己加载。

7.4.3 破坏双亲委派模型