类的加载流程

作者：vcjmhg
原文链接：https://ld246.com/article/1618387419746
来源网站：链滴

概述

什么是类加载呢?

我们知道一个Class文件编译完成之后是存在于磁盘的一个普通文件，如果想要执行，必然需要将Clas文件加载到内存中，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型**，这个过程其实就是类的加载机制。

当然上述过程说的比较抽象，**具体来说**其实Class文件从开始加载到内存中开始到被卸载回收为止，个过程分为**五大步**：

加载

加载是类加载过程的第一步，主要完成三件事情：

1. 通过类的 **全限定名**来获取定义此类的**二进制流（获取二进制流）**
2. 将字节流所代表的静态存储结构转换成方法区的运行时数据结构（ **转换存储结构**）
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**各种数据的访问入口（生成代表自己的Class对象）**

《Java 虚拟机规范》中其实对这三点的要求并不太具体，留给虚拟机的实现和Java应用的灵活性比较。比如：**"通过全类名获取定义此类的二进制字节流" 并没有指明具体从哪里获取、怎样获取。** 因而实际获取二进制字节流的时候就有许多方式，比如：比较常见的就是从 ZIP 包中读取（日后出现的JA、EAR、WAR格式的基础）、其他文件生成（典型应用就是JSP）等等。

并且相对于类加载过程的其他阶段，**非数组类型**在加载阶段（在加载阶段获取二进制字节流的阶段）的**控性是最强的。**在该阶段，既可以使用Java虚拟机内置的引导类加载器来完成，也可以由**用户自定义类**加载器去完成，具体可看"类加载器的加载原理"。

此处为何强调是非数组类呢？ 主要因为数组类本身是**不通过**类加载器创建的，它是由Java虚拟机直接**内存中构建出来的**，但数组类和类加载器仍然还是有着**密切的关系**，因为数组类中的元素类型（Element Type，指的是数组去掉所有维度的类型）最终**还是要通过类加载器**来加载完成。

一个数组类（简称C）创建过程遵循如下流程：

1. 在加载时首选判断数组的 组件类型（Component Type 数组去掉一个维度之后的类型）是否是用类型，如果是则按照本小节定义的"加载过程" 所来进行加载，并将**数组类的可访问性**设置成与**组件型**的可访问性相同

2. 如果不是引用类型（比如int[]中的int）则将其与"引导类加载器" 相关联，将数组类的可访问性设成public

这边可能某些小伙伴会有疑问，此处的元素类型和组件类型有什么具体区别吗？

此处我们写一个小例子来做一个说明，具体代码如下所示：

```
public class TestComponentType {
    public static void main(String[] args) {
        String[] str1 = new String[0];
        String[][] str2= new String[0][0];
        /**
         * 获取数组变量的组件类型
         */
        System.out.println("str1的组件类型为:"+str1.getClass().getComponentType());
        System.out.println("str2的组件类型为： "+str2.getClass().getComponentType());
    }
}
```

运行结果如下：

```
str1的组件类型为:class java.lang.String
str2的组件类型为： class [Ljava.lang.String;
```

我们可以看到一维数组和二维数组的组件类型是不同的，str1去掉一个维度的的类型之后为String，而其组件类型就是java.lang.String而str2去掉一个维度的类型之后变成一维数组[Ljava.lang.String。str1和str2它们的元素类型是相同的都是java.lang.String。

总结来说就是组件类型是**数组去掉一个维度之后的类型；** 元素类型是去**掉所有维度信息**之后的数据类。

好了前边我们讲了类加载阶段所做的一些事情，但我们思考这样一个问题，**什么情况下需要开始类加的第一个过程？或者说类加载的一个时机**

实际上第一个阶段“加载”的一个具体时机，《Java虚拟机规范》中并没有强制要求，而是严格规定有且只有六种情况下需要对类进行"初始化",因而在**初始化**之前的**加载**和**连接**过程必须提前完成。

连接

连接过程展开来说分成了三个步骤：验证、准备和解析。

验证

验证是连接阶段的第一步，主要用来确保Class文件的**字节流中包含的信息**符合《Java虚拟机规范》全部约束要求。

首先我们要考虑**为什么要有验证这一步骤呢**?

前面我们说到由于到Class文件来源的宽泛性要求，因而如果对加载的Class文件完全信任的话，可能因为载入了**有错误或者有恶意企图**的字节码流而导致整个系统受到攻击甚至崩溃，所以验证字节码是Java虚拟机保护自身的一项**必要措施。**

既然说验证的过程这么重要，**那jvm虚拟机是如何进行验证的？或者说在验证阶段它做了哪些工作呢?**

从整体上看，验证阶段主要完成了四个阶段的工作：

- 验证阶段的工作
  - 文件格式验证
  - 元数据验证
  - 字节码验证
  - 符号引用验证

文件格式验证

该阶段主要验证字节流是否符合**Class文件格式的规范**，并且能够被当前版本的虚拟机处理。该阶段证点有很多比如：

- 是否以魔数 0XCAFFEBABE
- 主、次版本号是否在java虚拟机可接受的范围之内
- 常量池中的常量是否有不被支持的常量类型（检查常量tag标志）
- ……

该阶段的主要**目的**是保证字节流能够正确的解析并存储在"方法区" 之内，格式上符合一个Java类型信的要求，只有通过了这个阶段的验证之后，这段字节流才**被允许进入Java虚拟机内存的方法区**中进行储。

元数据验证

该阶段是对字节码描述信息进行**语义分析**，保证其描述信息符合《Java虚拟机规范》的要求。

验证点可能包含如下内容：

- 这个类是否有父类（除了 java.lang.Object类之外，所有的类都应该有父类）
- 这个类是否继承了不允许被继承的类（比如 final修饰的类）
- 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的实现的所有方法。
- ……

该阶段主要目的对类的元数据信息进行**语义校验。**

字节码校验

第三个阶段是整个验证过程中最复杂的一个阶段，主要目的是通过数据流分析和控制流分析，确定程语义是**合法的**、**符合逻辑**的，不会做出危害虚拟机的行为。

该阶段会对类的方法体（Class中的"Code属性"）进行**校验分析**保证被**校验方法**在运行期间不会对虚机的安全造成威胁，比如：

- 保证任意时刻操作数栈的数据类型与指令代码序列都能够配合工作，例如不会出现类似于“在操作中放置了一个int类型的数据，使用时却按照long类型来加载本地变量表。
- 保证任何跳转指令都不能跳转到 **方法体之外**的字节指令上，
- 保证方法体中类型的转换总是有效的，例如不能把父类对象复制给子类变量
- .....

那进行了这么严密的验证，是否就说明我们的代码**绝对就是安全**的呢?

这个答案肯定是否定的，即使字节码验证阶段进行了再严密、再大量的检查，仍然无法保证其**绝对**安性，这就涉及到离散数学中一个著名问题-停机问题，简单来说就是，我们无法通过程序检查出程序**否**在有限时间内结束运行。这是一个悖论，无法被解决。

另外，前边我们提到这个字节码校验是加载阶段最为耗时，最为复杂的操作，但它由是我们类加载过中必不可少的过程，如果耗时太长会对程序的实时性带来很大的影响，因此需要通过一些措施来减少该阶段的耗时。

那我们考虑能不能把其中一些校验操作放到类加载之前呢?

在JDK6之后的Javac编译器和Java虚拟机里进行一项联合优化，把**尽可能多的校验辅助措施**挪到Java编译器里进行。

具体做法就是在方法体"的Code属性"中增加一项StackMapTable的新属性，这项属性描述了方法体有的基本块（Basic Block，按照控制流拆分的代码块）开始时候本地变量表和操作栈应有的状态，在节码验证期间，Java虚拟机就不需要根据程序推导这些状态，只需要检查StackMapTable属性中记录否合法即可。

符号引用验证

最后一个阶段工作是在虚拟机符号引用转化成直接引用的时候即在"解析"阶段 发生的，所做的事情主是对类自身以外的各类信息进行匹配性校验，通俗来讲，要校验这个类缺少所需要的外部类、方法、段等资源，是否访问了本该被没有访问权限的一些外部类、方法等资源。具体来说：

- 符号引用中通过权限定名能不能找到对应的类
- 在指定类中是否存在符合方法的字段描述符
- 可访问性是否合法
- .....

这一步骤的**主要目的**是确保解析行为是否能够正常执行。

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，这些内存都将在方法去中进行分配，在该阶段我们需要注意一下两点：

1. 该阶段进行内存分配的只包含 **类变量**，不包含**实例变量**，实例变量将在"对象的创建过程"中分配Java堆中。
2. 类变量的初始化通常情况下是 **零值**，但如果是常量则会直接对常量进行初始化，比如public static

final int value = 123;常量value在该阶段会直接被初始化成123。

基本类型的初始化表如下所示：

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。

其中符号引用可以是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**，只要能够**无歧义**位到目标即可。直接引用则是可以直接指向目标的指针，相对偏移量，或者**能够间接定位到目标的句柄**这里的直接引用可能和操作系统中的直接寻址不同，直接寻址里边直接放的是地址，一步到位。而这的**直接引用**分成了三类，但它们总体特点就是**在程序运行中**，**虚拟机可以通过该引用找到目标**即可。

解析动作主要是针对七类符号引用进行的：

- 解析的种类
  - 类或接口解析
  - 字段解析
  - 方法解析
  - 接口方法解析
  - 方法类型解析
  - 方法句柄解析
  - 调用点限定符解析

具体的解析过程，限于篇幅原因，此处暂不详述，具体可参考《深入理解JVM虚拟机第三版》第七章容。

初始化

前边我们说了《Java虚拟机规范》中有且只有六种情况必须对类进行**初始化**：

1. 遇到new、getstatic、putstatic或者invokestatic这四条字节码指令时，如果类型没有进行初始化则需要先触发其初始化阶段。

能够生成这四种字节码指令的**典型场景**如下：

1. 使用new创建新对象

2. 读取或者设置一个类型的静态字段（用 final修饰的常量或者已在编译期把结果放入常量池的态字段除外）的时候

3. 调用一个类型的静态方法

2. 使用 java.lang.reflect包中的方法对类型进行反射调用的时候

3. 当初始化类的时候， **发现其父类没有被初始化，需要先触发其父类的初始化。**

4. 虚拟机启动时，用户指定的主类（含main）没有被初始化时，需要被初始化

5. 当使用JDK7新加入的动态语言支持时，如果一个java.lang.invoke.MethodHandle实例最后的执结果为REF_getStatic、REF_putStatic、REF_invokeStati、REF_newInvokeSpecial这四种类型的句方法，并且这个方法的句柄对应的类没有进行**初始化**，则需要先触发其初始化。

6. 当一个接口定义了JDK8中新加入的默认方法（被 **default关键字**修饰的接口方法）时，如果这个口的实现类发生了初始化，那该接口要在其之前被初始化

类的初始化阶段是类加载过程的最后一个步骤，也是**真正开始执行**类中编写的Java程序代码的过程（他过程除加载过程外其他过程完全由虚拟机主导），本质上是讲，就是执行**类构造器**<clinit>()方法过程。

但我们需要**注意**的是<clinit>()方法本身并不是程序员直接在**程序代码中编写**的，而是由Javac编译器**动生成的。**既然这样为何前边又说初始化类会真正开始执行**类中编写的Java程序代码** question uestion

要解决这个问题，我们有必要了解一下<Clinit>()方法的生成过程：

clinit()方法是由编译器自动收集类中所有**类变量**的**赋值语句**和**静态语句块**(static{}块)合并生成。**收集序**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，义**在它之后的变量**，**可以赋值但不能访问。**

例如下边程序：

```
public class TestStaticVariable {
    static {
        a = 2; //给变量赋值可以正常通过编译通过
        //System.out.println(a); //报错Cannot reference a field before it is defined
    }
    static int a = 1;

    public static void main(String[] args) {
        System.out.println(a);
    }
}
```

运行结果为：**1**

但 **运行结果似乎在静态代码块中的赋值操作根本没起作用，那这样的设计的目的是啥？** 为什么不直设计成编译错误呢?

同时思考下边一段代码：

```
public class TestStaticVariable {
    static {
        a = 2;
    }
    static int a;

    public static void main(String[] args) {
        System.out.println(a);
    }
}
```

运行结果为：**2**

为什么它又起作用了？？？

这两个问题暂未想明白。。thinking "#Java
为何定义在它之后的变量可以被赋值，但不能被访问呢？"

这里的<clinit>()方法似乎和实例对象的<init>()方法很相似呀，但实际上它们也有很多**不同点**：

clinit()方法它不需要**显示**的调用父类构造器，Java虚拟机会保证在子类的<clinit>()方法在执行之前**类**的<clinit()>方法**已经执行完成**。因而在Java虚拟机中第一个被执行<clinit>()的方法类型肯定是java.ang.Object。

同时关于<clinit>()我们还要注意一下几点：

1. **并不是所有类都会生成**<clinit>()方法，如果一个类没有静态语句块，也没有对变量的赋值操作，么编译器可以不为这个类生成<clinit>()方法。

2. 在接口中，虽然不能使用静态语句块，但由于仍然存在变量的初始化操作，因而在这种情况下，接也会生成<clinit>()方法，但与类不同的时，执行接口的<clinit>()方法时，不需要先执行父类的clinit()法，因为**只有当父接口中定义的变量被使用时**，父接口才会被初始化。

3. 因为 <clinit>()方法是java虚拟自动生成并加载的，因而java虚拟机会自动保证其加锁同步。但可能也会导致一个问题，如果一个类的<clinit>()方法中有耗时很长的操作时，可能会造成多个线程塞。

使用

使用过程实际上就是通过类模板，创建对象的过程。具体参考"对象的创建过程"

卸载

卸载类，实际上就是指该类的Class对象被"GC" 。当类被**卸载**时需要满足三个要求：

1. 该类的 **所有的实例对象**都已被GC，也就是说堆不存在该类的实例对象。
2. 该类没有 **在其他任何地方被引用**
3. 该类的 **类加载器的实例**已被GC

总结

本文主要讲了类文件从加载到内存、连接、初始化、使用和卸载完整生命周期中Java虚拟机所做的工以及每一步操作的必要性，希望能给读者以帮助。

参考

1. 类加载过程
2. 《深入理解Java虚拟机》