在上一课时我详细介绍了ClassLoader的使用，包括它的主要作用就是用来将class字节码加载到内存中。那 JVM 加载 class 文件的具体过程是怎样的呢？本课时我们就来了解一下这一详细过程以及当中存在的问题。

一个 class 文件被加载到内存中需要经过 3 大步：装载、链接、初始化。其中链接又可以细分为：验证、准备、解析 3 小步。 因此用一张图来描述 class 文件加载到内存的步骤如下所示。

1. 装载

1.1 什么是装载

装载是指 Java 虚拟机查找 .class 文件并生成字节流，然后根据字节流创建 java.lang.Class 对象的过程。

这一过程主要完成以下 3 件事：

1. 1）ClassLoader通过一个类的全限定名（包名+类名）来查找.class文件，并生成二进制字节流：其中 class 字节码文件的来源不一定是 .class 文件，也可以是 jar 包、zip 包，甚至是来源于网络的字节流。
2. 把 .class 文件的各个部分分别解析（parse）为 JVM 内部特定的数据结构，并存储在方法区。

还记得在课时 03 中介绍的 .class 文件结构吗？在这里 JVM 会将这些 .class 文件的结构转化为 JVM 内部的运行时数据结构。这点同 JSON 解析过程有点类似：如果你做过 Android 开发，应该都使用过 GsonFormat 将后端开发返回的 JSON 结构转化为一个运行时 Bean 类，当程序运行时使用这个 Bean 类去解析处理 JSON 数据。

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。

1.2 加载时机

一个项目经过编译之后，往往会生成大量的 .class 文件。当程序运行时，JVM 并不会一次性的将这些 .class 文件全部加载到内存中。那 JVM 是什么时候加载某 .class 文件呢？对此，Java 虚拟机规范中并没有严格规定，不同的虚拟机实现会有不同实现。不过以下两种情况一般会对 class 进行装载操作。

- 隐式装载：在程序运行过程中，当碰到通过 new 等方式生成对象时，系统会隐式调用 ClassLoader 去装载对应的 class 到内存中；
- 显示装载：在编写源代码时，主动调用 Class.forName() 等方法也会进行 class 装载操作，这种方式通常称为显示装载。

2. 链接

链接过程分为 3 步：验证、准备、解析。

2.1 验证

验证是链接的第一步，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，并且不会危及虚拟机本身的安全。主要包含以下几个方面的检验。

1. 文件格式检验：

   检验字节流是否符合 class 文件格式的规范，并且能被当前版本的虚拟机处理。

2. 元数据检验：

   对字节码描述的信息进行语义分析，以保证其描述的内容符合 Java 语言规范的要求。

3. 字节码检验：

   通过数据流和控制流分析，确定程序语义是合法、符合逻辑的。

4. 符号引用检验：

   符号引用检验可以看作是对类自身以外（常量池中的各种符号引用）的信息进行匹配性校验。

实例分析：

我使用以下 Foo.java 来演示验证阶段的几种情况：

```
public class Foo{
    public static void main(String[] args){
        new Foo().print();
    }

    public void print(){
        int superCode = super.hashCode();
        System.out.println("superCode is " + superCode);

        int thisCode = hashCode();
        System.out.println("thisCode is " + thisCode);
    }

    public int hashCode(){
        return 111;
    }
}
```

使用 javac 编译 Foo.java 生成 Foo.class 字节码文件，然后使用 16 进制编辑器打开 Foo.class 文件，部分如下：

```
public class Foo{
    public static void main(String[] args){
        new Foo().print();
    }

    public void print(){
        int superCode = super.hashCode();
        System.out.println("superCode is " + superCode);

        int thisCode = hashCode();
        System.out.println("thisCode is " + thisCode);
    }

    public int hashCode(){
        return 111;
    }
}
```

正常情况下，使用 java Foo 执行结果如下：

```
➜  danny_folder
➜  danny_folder java Foo
superCode is 2018699554
thisCode is 111
➜  danny_folder
```

如果使用 16 进制编辑器修改 class 文件中的魔数，如下所示：

```
cafe babb 0000 0034 0036 0a00 1000 1c07
001d 0a00 0200 1c0a 0002 001e 0a00 1000
1f09 0020 0021 0700 220a 0007 001c 0800
```

将“cafe babe”修改为“cafe babb”，重新运行则会报如下错误：

```
➜  danny_folder java Foo
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.ClassFormatError: Incompatible magic value 1667327589 in class file Foo
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:763)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
        at java.net.URLClassLoader.defineClass(URLClassLoader.java:467)
        at java.net.URLClassLoader.access$100(URLClassLoader.java:73)
```

class 文件中魔数后的“0034”为版本号，如果将其修改为“0035”则会报如下错误：

```
→  danny_folder java Foo
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.UnsupportedClassVersionError: Foo has been compiled by a more recent version of
 the Java Runtime (class file version 53.0), this version of the Java Runtime only recognizes class file versions up
 to 52.0
```

版本号“0034”之后的“0036”是常量池计数器，表示常量池中有 54 个常量。如果将这个值进行修改也有可能造成运行时错误，比如我将“0036”改为“0032”：

```
cafe babe 0000 0034 0032 0a00 1000 1c07
001d 0a00 0200 1c0a 0002 001e 0a00 1000
1f09 0020 0021 0700 220a 0007 001c 0800
230a 0007 0024 0a00 0700 250a 0007 0026
```

重新执行 java Foo，则会报如下错误：

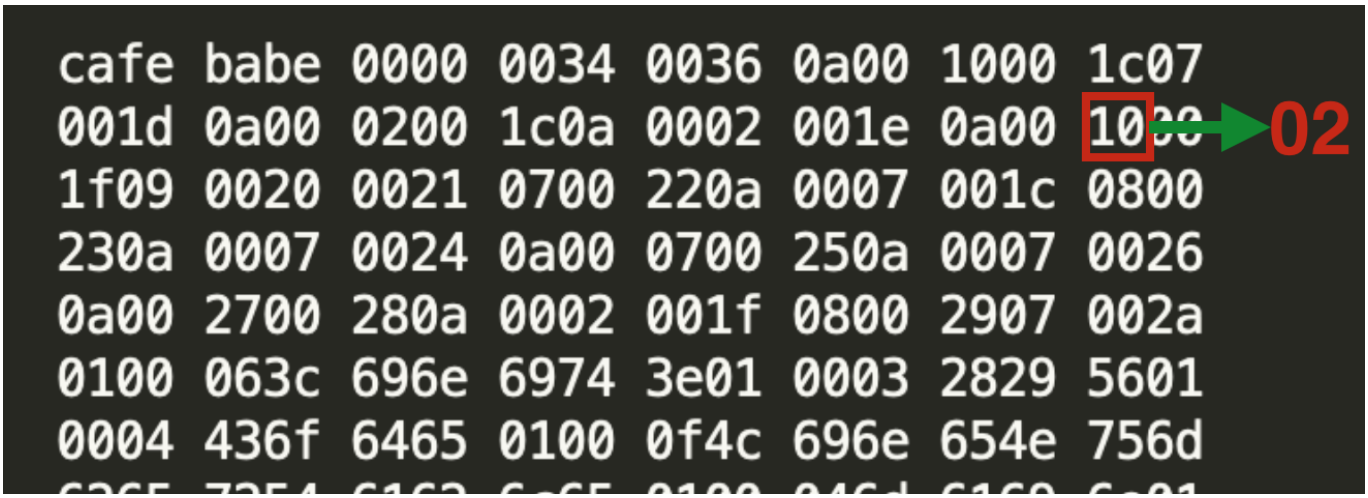
```
→  danny_folder java Foo
Error: A JNI error has occurred, please check your installation and try again
Exception in thread "main" java.lang.ClassFormatError: Invalid constant pool index 50 in class file Foo
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:763)
        at java.security.SecureClassLoader.defineClass(SecureClassLoader.java:142)
```

虽说 JVM 会检查各种对 class 字节码文件的篡改行为，但是依然无法百分百保证 class 文件的安全性。比如我还是用 Foo.java 举例，在 Foo.java 中的 print 方法中，分别打印出父类的自身类的 hashCode 值，分别是：2018699554 和 111。我们可以在 class 字节码的基础上进行篡改，将父类的 hashCode 也返回 111。

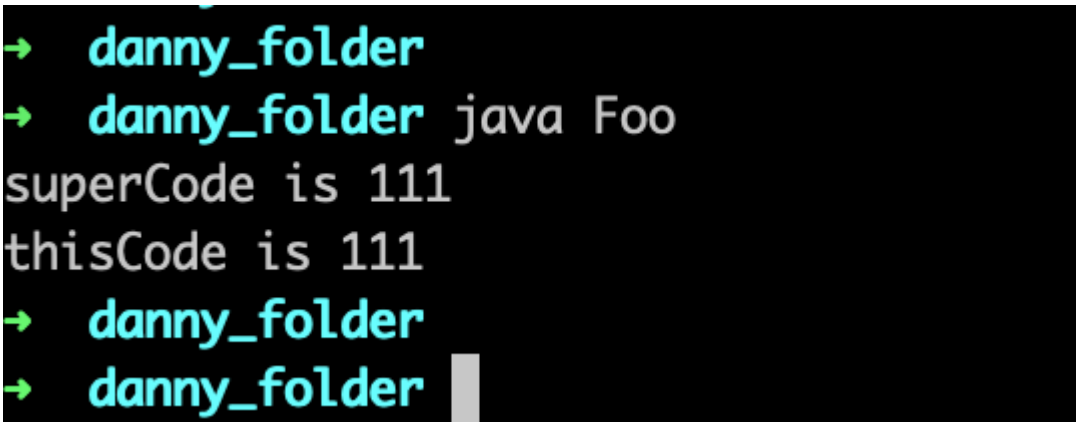
因为篇幅原因，我只截取了部分常量池的内容。图中 1 处指向了父类 Object的hashCode 方法，图中 2 处指向了 Foo 的 hashCode 方法。在课时 03 中，我们了解已经了解了 CONSTANT_Methodref_info 结构如下：

```
CONSTANT_Methodref_info {
    u1 tag = 10;
    u2 class_index;      指向此方法的所属类
    u1 name_type_index;      指向此方法的名称和类型
}
```

其中class_index就是指向方法的所属类（图中为16，转化为16进制为0X10），因此只需要使用 16 进制编辑器将指向 Object 的 class_index 改为执行 Foo 的 class_index 即可。具体修改如下：



将图中 0X10 改为 0X02 并保存，重新运行 java Foo 效果如下：



可以看出，虽然在 Java 源文件中调用的是 super.hashCode() 方法，但是经过篡改之后，Foo.class 文件成功通过 JVM 的校验，并成功执行最终打印出我们想要的结果。

注意：上面的实例也说明即使没有Java源文件，在某种程度上，工程师还是可以对编译之后的class字节码文件进行篡改。这也是为什么我们在项目中经常会使用混淆，甚至是使用一些三方的加固软件，来保证我们所编写的代码的安全性。

2.2 准备

准备是链接的第 2 步，这一阶段的主要目的是 **为类中的静态变量分配内存**，并为其设置"0值"。比如：

```
public static int value = 100;
```

在准备阶段，JVM 会为 value 分配内存，并将其设置为 0。而真正的值 100 是在初始化阶段设置。并且此阶段进行内存分配的仅包括类变量，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）。

有一种情况比较特殊--静态常量，比如：

```
public static final int value = 100;
```

以上代码会在准备阶段就为 value 分配内存，并设置为 100。

Java 中基本类型的默认"0值"如下：

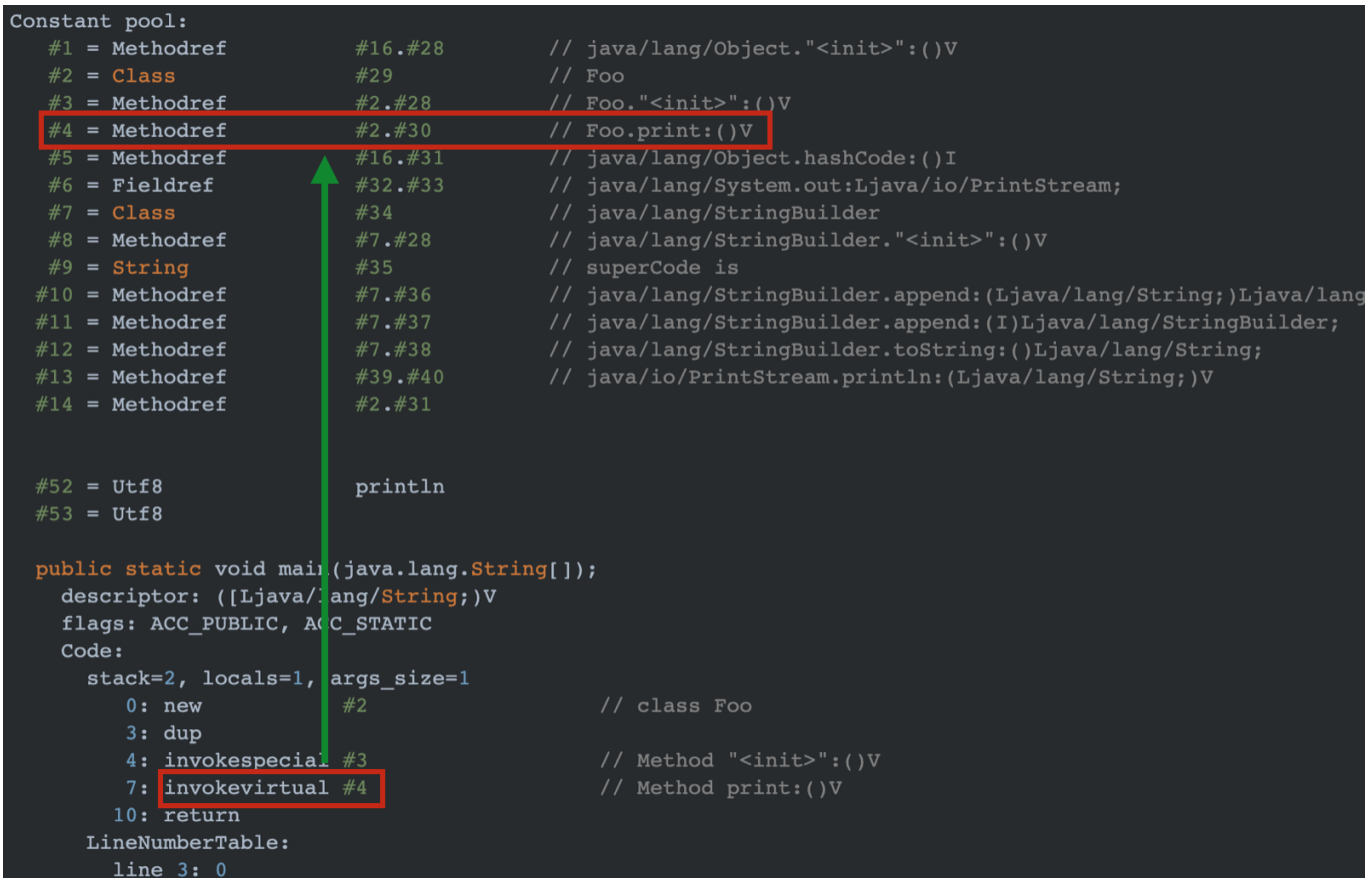
- 基本类型（int、long、short、char、byte、boolean、float、double）的默认值为 0；

- 引用类型默认值是 null；

2.3 解析

解析是链接的最后一步，这一阶段的任务是 **把常量池中的符号引用转换为直接引用，也就是具体的内存地址。在这一阶段，JVM 会将常量池中的类、接口名、字段名、方法名等转换为具体的内存地址。**

比如上面 Foo.java 中编译之后 main 方法的字节码如下：



在 main 方法中通过 invokevirtual 指令调用了 print 方法，`Foo.print:()V`就是一个 **符号引用**，当 main 方法执行到此处时，会将 **符号引用** `Foo.print:()V` 解析（resolve）成直接引用，可以将直接引用理解为方法真正的内存地址。

对于符号引用和直接引用，可以将其与生活中的微信聊天进行类比，在微信好友列表中，保存的是好友的名称或者别名（也就是符号引用），当我们真正给某个好友发消息时，计算机（JVM）会根据好友的名称找到对象计算机的 IP 地址（直接引用）并成功将消息发送给这一地址。

3. 初始化

这是 class 加载的最后一步，这一阶段是执行类构造器方法的过程，并真正初始化类变量。比如：

```
public static int value = 100;
```

在准备阶段 value 被分配内存并设置为 0，在初始化阶段 value 就会被设置为 100。

3.1 初始化的时机

对于装载阶段，JVM 并没有规范何时具体执行。但是对于初始化，JVM 规范中严格规定了 class 初始化的时机，主要有以下几种情况会触发 class 的初始化：

1. 虚拟机启动时，初始化包含 main 方法的主类；
2. 遇到 new 指令创建对象实例时，如果目标对象类没有被初始化则进行初始化操作；
3. 当遇到访问静态方法或者静态字段的指令时，如果目标对象类没有被初始化则进行初始化操作；
4. 子类的初始化过程如果发现其父类还没有进行过初始化，则需要先触发其父类的初始化；
5. 使用反射 API 进行反射调用时，如果类没有进行过初始化则需要先触发其初始化；
6. 第一次调用 java.lang.invoke.MethodHandle 实例时，需要初始化 MethodHandle 指向方法所在的类。

3.2 初始化类变量

在初始化阶段，只会初始化与类相关的静态赋值语句和静态语句，也就是有 static 关键字修饰的信息，而没有 static 修饰的语句块在实例化对象的时候才会执行。

比如以下代码：

```
public class ClassInit{
    public static int value = 1;

    // 静态语句块在初始化阶段执行
    static{
        System.out.println("ClassInit static block!");
    }

    // 非静态语句块只在创建对象实例时被执行
    {
        System.out.println("ClassInit non-static block!");
    }
}
```

然后在 ClassInitTest.java 中访问 ClassInit 的 value 值，如下：

```
public class ClassInitTest{
    public static void main(String[] ags){
        ClassInit.value = 2;
    }
}
```

执行上述代码，打印日志如下：

```
➜  course06
➜  course06 java ClassInitTest
ClassInit static block!
➜  course06
➜  course06
```

可以看出，非静态代码块并没有被执行。如果将 ClassInitTest.java 修改如下：

```
public class ClassInitTest{
    public static void main(String[] ags){
        ClassInit.value = 2;
        ClassInit ci = new ClassInit();
    }
}
```

加了一行代码，使用 new 创建 ClassInit 对象实例。再次执行后非静态代码块也将会被执行，如下：

```
➜  course06
➜  course06 java ClassInitTest
ClassInit static block!
ClassInit non-static block!
➜  course06
➜  course06
```

3.3 被动引用

上述的 6 种情况在 JVM 中被称为主动引用，除此 6 种情况之外所有引用类的方式都被称为被动引用。**被动引用并不会触发 class 的初始化**。

最典型的就是在子类中调用父类的静态变量，比如有以下两个类：

```
class Parent {
    public static int value = 1;

    static {
        System.out.println("this is Parent!");
    }
}

class Child extends Parent {
    static {
        System.out.println("this is Child!");
    }
}
```

可以看出 Child 继承自 Parent 类，如果直接使用 Child 来访问 Parent 中的 value 值，则不会初始化 Child 类，比如如下代码：

```
public class NonInitTest{
    public static void main(String[] args){
        Child.value = 2;
    }
}
```

执行上述代码，打印如下效果：

```
➜  danny_folder
➜  danny_folder java NonInitTest
this is Parent!
➜  danny_folder
➜  danny_folder
```

可以看出，Child 中的静态代码块并没有被执行。也就是说 JVM 并没有对 Child 执行初始化操作。

对于静态字段，只有直接定义这个字段的类才会被初始化，因此通过子类Child来引用父类Parent中定义的静态字段，只会触发父类Parent的初始化而不会触发子类Child的初始化。至于是否要触发子类的加载和验证，在虚拟机规范中并未明确规定，可以通过 XX:+TraceClassLoading 参数来查看，比如使用如下命令再次执行 NonInitTest：

```
java -XX:+TraceClassLoading NonInitTest
```

查看部分打印日志如下：

```
[Loaded java.security.BasicPermissionCollection from /Library/Java/JavaVirtualMachines/jdk1.8.0_151.jdk/Contents/Home/jre/lib/r
[Loaded NonInitTest from file:/Users/axing/work_dir/jvm/danny_folder/]
[Loaded sun.launcher.LauncherHelper$FXHelper from /Library/Java/JavaVirtualMachines/jdk1.8.0_151.jdk/Contents/Home/jre/lib/rt.j
[Loaded java.lang.Class$MethodArray from /Library/Java/JavaVirtualMachines/jdk1.8.0_151.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.Void from /Library/Java/JavaVirtualMachines/jdk1.8.0_151.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded Parent from file:/Users/axing/work_dir/jvm/danny_folder/]
[Loaded Child from file:/Users/axing/work_dir/jvm/danny_folder/]
this is Parent!
[Loaded java.lang.Shutdown from /Library/Java/JavaVirtualMachines/jdk1.8.0_151.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.Shutdown$Lock from /Library/Java/JavaVirtualMachines/jdk1.8.0_151.jdk/Contents/Home/jre/lib/rt.jar]
→  danny_folder
→  danny_folder
```

可以看出，虽然只有 Parent 被初始化，但是 Parent 和 Child 都经过了装载和验证阶段，并被加载到内存中。

3.4 class 初始化和对象的创建顺序

关于 class 的初始化还有一点经常会在面试中被提及，那就是对象的初始化顺序。当我们在代码中使用 new 创建一个类的实例对象时，类中的静态代码块、非静态代码块、构造函数之间的执行顺序是怎样的。

```
public class InitOrder {
    public static void main(String[] args){
        Parent p = new Child();
        System.out.println("~~~~~~~~~~~~~~~~~~~~~~~~");
        p = new Child();
    }

    static class Child extends Parent{
        static {
            System.out.println("Child static block!");
        }

        {
            System.out.println("Child non-static block!");
        }

        public Child(){
            System.out.println("Child constructor!");
        }
    }

    static class Parent{
        static {
            System.out.println("Parent static block!");
        }

        {
            System.out.println("Parent non-static block!");
        }

        public Parent(){
            System.out.println("Parent constructor!");
        }
    }
}
```

在 main 方法中执行了 2 次 new Child() 的操作，执行上述代码结果如下：

```
➜  course06 java InitOrder
Parent static block!
Child static block!
Parent non-static block!
Parent constructor!
Child non-static block!
Child constructor!
~~~~~~~~~~~~~~~~~~~~~~~
Parent non-static block!
Parent constructor!
Child non-static block!
Child constructor!
➜  course06
```

总结一下对象的初始化顺序如下：

静态变量/静态代码块 -> 普通代码块 -> 构造函数

1. 父类静态变量和静态代码块；
2. 子类静态变量和静态代码块；
3. 父类普通成员变量和普通代码块；
4. 父类的构造函数；
5. 子类普通成员变量和普通代码块；
6. 子类的构造函数。

5. 总结

这节课主要介绍了 .class 文件被加载到内存中所经过的详细过程，主要分 3 大步：装载、链接、初始化。其中链接中又包含验证、准备、解析 3 小步。

1. 装载：指查找字节流，并根据此字节流创建类的过程。装载过程成功的标志就是在方法区中成功创建了类所对应的 Class 对象。
2. 链接：指验证创建的类，并将其解析到 JVM 中使之能够被 JVM 执行。
3. 初始化：则是将标记为 static 的字段进行赋值，并且执行 static 标记的代码语句 。