走进 JVM 之字节码与类加载

作者：matthewhan
原文链接：https://ld246.com/article/1601016020278
来源网站：链滴

字节码

JVM主流还是HotSpot，是OpenJDK最主流的一种。

跨平台需要一个中间层，那就是字节码（ByteCode），JVM会将字节码编译执行，热点代码还会通过JT动态编译成机器码提高效率。

一个Java类文件，用十六进制表示二进制流。其中，起始的4个直接非常特殊：cafe babe。是Goslin定义的一个魔法数，标志该文件是一个Java类文件。

JVM在字节码上设计一套操作码助记符，用特殊的单词来标记。

一个 .java文件转换成字节码甚至是机器码文件需要以下几步：

Java ==> 词法解析 = token流 => 语法解析 ==> 语义分析 ==> 生成字节码 ==> 字节码

其中，语法解析的目的是组装成一颗语法树，再通过 语义分析阶段检查类型、关键字、作用域是否合。

字节码必须通过**类加载过程**加载到JVM环境后才可以执行。执行有3种模式

- 解释执行
- JIT编译执行
- JIT编译执行与解释混合执行

主流的JVM就是混合执行，在控制台输入 java -version也可以看到 mixed mode的字样，表示是混执行。

类加载过程

类是在运行期间第一次使用时动态加载的，而不是一次性加载所有类。因为如果一次性加载，那么会用很多的内存。

流程

主要分成**三大步**：

1. 加载

只是类加载的第一个阶段，不要和类加载混淆。主要干三件事情：

- 通过类的完全限定名称获取定义该类的二进制字节流。
- 将该字节流表示的静态存储结构转换为方法区的运行时存储结构。
- 在内存中生成一个代表该类的 Class 对象，作为方法区中该类各种数据的访问入口。

2. 链接（验证、准备、解析）

3. 初始化

分为主动引用和被动应用。

- 主动引用：

new关键字、main类等

- 被动引用：

静态字段、常量、SuperClass[] sca = new SuperClass[10];这样一个数组的初始化。

通过数组定义来引用类，不会触发此类的初始化。该过程会对数组类进行初始化，数组类是一个由虚机自动生成的、直接继承自 Object 的子类，其中包含了数组的属性和方法。

初始化的过程：读取字节码的二进制数据到内存中，在JVM的方法区内，然后利用字节码文件创建一Class对象作用在堆区。

所以说类加载是一个将.class字节码文件实例化成Class对象并进行相关初始化的过程。

类加载器

双亲委派模型，或者叫「溯源委派加载模型」更合适。因为类加载器类似原始部落，存在权利等级制，最高的一层是BootStrap。

低层次的当前类加载器，不能覆盖更高层的加载器加载的类。**所以低层的加载器想要加载某个类时，要向上逐级询问**：该类加载了吗？高一级的父加载器往往都是懒狗，收到下级的请求会转发该请求给身的上级。

所以一次询问一定会传递到BootStrap ClassLoader。加载不到时，才会逐级向下尝试加载。如果类都加载不了，就会允许当前类加载器加载，不然肯定是优先父类加载器加载。

所以是按需加载。

父子关系是通过组合关系实现，非继承关系。

这个模型的好处：

1. 避免重复加载

2. 安全性问题，防止核心的加载器已加载的类被「覆盖、篡改」

例子：

比如在自己开发的环境中，不要定义和核心API同名的包名。

1.根加载器（Bootstrap）

最底层的加载器，由C++实现，没有父加载器所以没有继承 java.lang.ClassLoader，负责将存放在 <RE_HOME>\lib目录中的，或者被 -Xbootclasspath参数所指定的路径中的。

负责装载最核心的类：Object、System、String等。

根类加载器加载的类，打印他的 Classloader只会是null，因为不是在JVM体系内。

例如：System.out.println(Object.class.getClassLoader());

2.扩展类加载器（Platform）

纯Java语言编写。用于加载一些扩展的系统类（也就是安全性和重要性比上者稍差一点），比如：XM、加密、压缩。

这个类加载器是由 ExtClassLoader（sun.misc.Launcher$ExtClassLoader）实现。它负责将 <JAVA HOME>/lib/ext或者被 java.ext.dir系统变量所指定路径中的所有类库加载到内存中，开发者可以直使用扩展类加载器。

Java9之前是ExtClass，之后是PlatformClass。

3.应用类加载器（Application）

这个类加载器是由 AppClassLoader（sun.misc.Launcher$AppClassLoader）实现的。

负责加载用户类路径（ClassPath）上所指定的类库，开发者可以直接使用这个类加载器。比如自己的 TestFuck类，该类就是被该加载器所加载。

4.自定义加载器

ClassLoader

该类和Class都位于 java.lang这个包下，主要是对双亲委派模型的实现。

loadClass方法

经典递归实现

1. 先检查当前类是否已经在当前加载器加载了，如果没有，向上请求（上级也是这个模式）
2. 直到父类加载器抛出ClassNotFoundException，此时尝试自己去加载。

一般不覆写该方法，因为还是要遵循双亲委派模型的机制。

findClass方法

```
protected Class<?> findClass(String name) throws ClassNotFoundException {
    throw new ClassNotFoundException(name);
}
```

源码很简单，该方法需要覆写，因为上面说了，**loadClass是递归父类加载器直到抛出该异常。**那么要覆写该方法实现自己的自定义的类。

defineClass方法

通常与 findClass方法一起使用，在 findClass里面调用该方法返回。

继承 ClassLoader类，覆写 findClass方法。

什么时候需要自定义类加载器?

1. 隔离加载类：中间件不同的jar包相互影响
2. 修改类的加载方式
3. 扩展加载源：网络层面、数据库、电视机顶盒也能加载
4. 防止源码泄露

5.热部署

当我们的一个类已经被加载后，通过双亲委派模型，他并不会重新加载。但是我们在编译器中开发的候可以选择热部署的方式。他又是怎么实现的呢?

双亲委派模型的核心在于 loadClass方法，**如果直接略过或者覆写该方法**，就不会出现无限向上级请加载的情况了（递归倒了，哭cry）。

破坏双亲委派模型的2种方式:

1. 不走　loadClass方法
2. 将上级类加载器下本应该（交给父加载器加载）加载的jar包删了，这样上级就会抛出异常，直到逐返回下级加载「虚假的类」

第二种方式有点嗯了。。

6.线程上下文加载器

我们根加载器属实位高权重，但是保不准它也有求于小弟的时候。比如 mysql-connect-java.jar，每计科学生都知道的经典数据库驱动。

在BootStrap加载器下有个 rt.jar中有个类叫做 java.sql.DriverManager，会去加载 Driver.class，而class是一个接口，并没有实现类，它的实现类在第三方的jar包中，也就是驱动中的类，相当于反向载了。

```
public Void run() {
    ServiceLoader<Driver> loadedDrivers = ServiceLoader.load(Driver.class);
```

这里的Void是定义的一个类，不是void关键字。

双亲委派模型可不允许反向加载，所以这里采用的是线程上下文加载器，该方式虽然也破坏了双亲委模型，但是更为灵活。

```
ClassLoader cl = Thread.currentThread().getContextClassLoader();
return ServiceLoader.load(service, cl);
```