JVM 类加载

作者：helly
原文链接：https://ld246.com/article/1499508612502
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

一、类的生命周期

1. 加载

加载时机

Java虚拟机规范没有严格规定，看具体的Java虚拟机实现，可以预先加载。

加载过程（先到缓存找）

1. 根据类的全限定名找到类的二进制字节流；
2. 将二进制字节流的静态存储结构转化为方法区中的运行时数据结构；
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。

2. 验证- 确保被加载类的二进制字节流的正确性和安全性

1. 文件格式验证 - CAFEBABE、主次版本号、常量池中的数据类型是否正确等。
2. 元数据验证 - 语义分析，如是否有父类等。
3. 字节码验证 - 语义分析，数据流和控制流是否正确等。
4. 符号引用验证 - 符号引用是否正确，保证后面解析动作能正常执行。

3. 准备 - 为类的静态字段分配内存并赋初始值

同时被static和final修饰的字段在编译后会有ConstantValue属性，这使得其在准备阶段就会赋值为ConstantValue指定的值。

4. 解析 - 将符号引用替换为直接引用

符号引用：就是描述相对位置的符号，可以是任何字面量。

直接引用：地址指针或偏移量。

invokestatic - 用于调用类（静态）方法

invokespecial - 用于调用实例方法，特化于super方法调用、private方法调用与构造器调用

invokevirtual - 用于调用虚方法和final修饰的方法

invokeinterface - 用于调用接口方法

Java中支持多态（包括重载和重写）的只有虚方法，虚方法是指非static的、非final的并且非private方法，不包括构造器。

参数的数量、类型等信息组成了函数的signature。在不同语言中，函数的signature不仅可以包含参的数量、类型，也可能包含参数的结构/模式，甚至可能包括返回类型的数量和类型。

使用同一个名字来命名signature不同的函数，称为函数重载（function overloading），重载是编时概念。

方法分派（Method Dispatch）：单一分派（single-dispatch）（又叫静态分派）和多分派（multipe-dispatch）（又叫动态分派）。

重载的实现-单一分派

Java编译器在重载的时候通过参数的类型（静态类型）而不是实际类型来确定使用哪个重载的版本。用invokeVirtual字节码指令。

多态的实现-多分派

方法的第一个参数，也就是隐含参数this（称为接受者（Reciever））的实际类型会参与到多分派，他参数可能参与单一分派也可能参与多分派。

静态方法不能被重写，因为不参与多态，但能继承。

super关键字：使用super调用父类方法的话用的是invokespecial指令，这个指令是不支持多态的，找父类的方法执行。

5. 初始化 - 为类的静态变量赋予指定的初始值

初始化时机

虚拟机规范中并没强行约束什么时候开始类加载过程的第一个阶段：加载。但对于初始化阶段虚拟机范是严格规定了如下几种情况：

1. 遇到new、getstatic、putstatic、invokestatic这四条字节码指令时（对应new一个对象、读取或置类的静态字段（static final修饰的字段除外）、调用类的静态方法）；
2. 反射调用时 - Class.forName()；
3. 初始化一个类时如果其父类还未初始化先初始化其父类；
4. 虚拟机启动时先初始化包含mian()方法的那个类；
5. 还不懂。

6. 使用

7. 卸载

1. 正常结束
2. 遇到异常或错误结束
3. 执行了System.exit()方法
4. 操作系统错误时虚拟机进程终止

三、类加载器

1 启动类加载器（没有父类加载器）

加载JDK\jre\lib或-Xbootclasspath参数指定的路径下的Java类。

虚拟机按照文件名识别加载jar包，如果文件名不被虚拟机识别，即使将jar包放进lib目录也没用。

c++实现。

2 扩展类加载器（父类加载器为null）

加载JDK\jre\ext或-Djava.ext.dirs指定的路径下的Java类。

3 应用程序类加载器

加载用户类路径（java -classpath 或 -D java.class.path指定的路径）下所指定的类。

程序中默认的类加载器，因此可以通过ClassLoader.getSystemClassLoader()方法获得应用类加载器。

4 用户自定义类加载器

继承ClassLoader类，重写findClass()方法。

或继承URLClassLoader类。

四、类加载机制

1. 全盘负责
2. 父类委托
3. 缓存机制

Java虚拟机加载过的类的二进制字节流都会被缓存，下次要加载类的时候优先到缓存看一看有没有，就是为什么修改类后要重启Java虚拟机进程修改才能生效。

五、类加载的方式（3）

1 虚拟机进程启动时加载

2 Class.forName()动态加载

3 ClassLoader.loadClass()动态加载 - 不会执行static中的内容

六、双亲委派模型（JDK1.2引入）

如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把请求委托给父加器去完成，依次向上，因此，所有的类加载请求最终都应该被传递到顶层的启动类加载器中，只有当加载器在它的搜索范围中没有找到所需的类时，即无法完成该加载，子加载器才会尝试自己去加载该。

2 优点

① 避免类的重复加载，保证每个类只会被加载一次。

② 安全。案例：网络传过来一个java.lang.Integer类（本地加载过了）；自定义一个类丢进java.lan包（没权限）

3 双亲委派模型的父子关系并不是Java中的继承关系，而是通过组合来实现的。

4 实现

ClassLoader（抽象类）

```
/*
 * 先在缓存找class对象，找不到就加载
 * 有父类委托给父类
 * 没父类委托给启动类加载器
 * 都没找到通过自定义的findClass方法去找
 */
```

loadClass(String) // ClassLoader实现，双亲委派模型的实现
loadClass(String name, boolean resolve) // 是否在生成class对象的同时进行相关解析操作
findClass(String) // 用来被loadClass方法调用，ClassLoader抽象类中没有具体实现findClass，而直接抛出异常，各子类需要自己去实现
defineClass(byte[] b, int off, int len) // 将字节流转化为为JVM能够识别的Class对象，常与findClas方法一起使用，被findClass方法调用以生成class对象，ClassLoader抽象类已经实现它
resolve(Class<?> c) // 对class对象进行解析

SecureClassLoader（extends ClassLoader）

URLClassLoader（extends SecureClassLoader）

// 存在一个URLClassPath类，根据传进来的目录名或Jar包名创建相应的加载器（有3种）去加载对路径下的class文件，这3个加载器就是真正加载字节码流的
findClass(String name) // 实现
findResource()

ExtClassLoader（extends URLClassLoader）

AppClassLoader（extends URLClassLoader）

ExtClassLoader和AppClassLoader都是sun.misc.Launcher类的静态内部类。Lanunche初始化的时先创建ExtClassLoader类加载器，然后再创建AppClassLoader并把ExtClassLoader传递给它作为父加载器。

5 自定义类加载器

① 继承ClassLoader类就必须重写findClass()方法，因为ClassLoader类没有实现它，只是在里面抛个异常；

② 继承URLClassLoader

编写自定义类加载器场景：

1 class文件不在ClassPath路径下；

2 class文件通过网络传输过来，需要解密；

3 热部署功能（通过不同类加载器产生不同class对象）

六、JVM中如何判断两个class对象是否为同一个类对象？

1 类的全限定名必须一样；

2 必须由同一个ClassLoader对象加载。

因为不同ClassLoader对象拥有独立的类名称空间，所以加载的class对象也会存在于不同的类名空间。

但现实不会出现这种情况，因为有缓存，加载之前都会到缓存看一看加载没有，所以要重新加载class象必须绕过缓存查询，或直接调用findClass()方法

以“类的全限定名+ClassLoader实例ID”来标明Class对象。

七、显示加载与隐式加载

1 显示加载

Class.forName()

obj.getClass().getClassLoader().loadClass()

2 隐式加载

JVM加载

双亲委派模型的破坏者-线程上下文类加载器

参考：

《深入理解Java虚拟机》

JVM（1）：Java 类的加载机制

深入理解Java类加载器(ClassLoader)