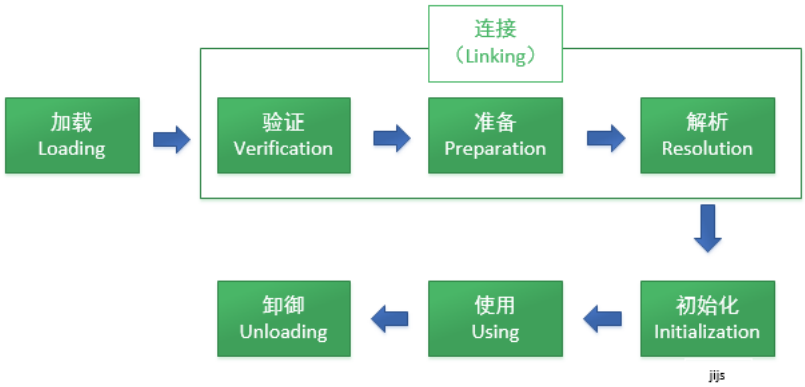
JVM 类加载机制深入浅出


作者 jijs (/u/1f0067e24ff8) + 关注
2017.05.04 23:14* 字数 2400 阅读 857 评论 2 喜欢 15 赞赏 1
(/u/1f0067e24ff8)

从类被加载到虚拟机内存中开始，到卸御出内存为止，它的整个生命周期分为7个阶段，加载(Loading)、验证(Verification)、准备(Preparation)、解析(Resolution)、初始化(Initialization)、使用(Using)、卸御(Unloading)。其中验证、准备、解析三个部分统称为连接。
7个阶段发生的顺序如下：



类的生命周期

1. 加载

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。

> **注意：JVM中的ClassLoader类加载器加载Class发生在此阶段**

2. 验证

2.1 文件格式的验证

1. 主要验证字节流是否符合Class文件格式的规范，如果符合则把字节流加载到方法区中进行存储。
2. 文件头、主次版本验证等等

2.2 元数据验证

主要对字节码描述的信息进行语义分析，保证其描述符合Java语言的要求。

1. 类是否有父类
2. 是否继承了不允许被继承的类（final修饰过的类）
3. 如果这个类不是抽象类，是否实现其父类或接口中所有要求实现的方法

4. 类中的字段、方法是否与父类产生矛盾（如：覆盖父类final类型的字段，或者不符合个则的方法）

2.3 字节码验证

最复杂的一个阶段。主要目的是通过数据量和控制流分析，确定程序语义是合法的，符合逻辑的。
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。

2.4 符号引用验证

符号引用中通过字符串描述的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（private、protected、public、default）是否可被当前类访问。

3、准备

准备阶段正式为类变量分配内存并设置初始值阶段。
public **static** int value=123; 初始后为 value=0;
对于static final类型，在准备阶段会被赋予正确的值
public **static final** value=123;初始化为 value=123;

如果是boolean值默认赋值为：false
如果是对象引用默认赋值为：null
...

> **注意：**
> **只设置类中的静态变量（方法区中），不包括实例变量（堆内存中），实例变量是在对象实例化的时候初始化分配值的**

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。

1. 符号引用：简单的理解就是字符串，比如引用一个类，java.util.ArrayList 这就是一个符号引用，字符串引用的对象不一定被加载。
2. 直接引用：指针或者地址偏移量。引用对象一定在内存（已经加载）。

5、初始化

1. 执行类构造器<clinit>
2. 初始化静态变量、静态块中的数据等（一个类加载器只会初始化一次）
3. 子类的<clinit>调用前保证父类的<clinit>被调用

> **注意：**
> **<clinit>是线程安全的，执行<clinit>的线程需要先获取锁才能进行初始化操作，保证只有一个线程能执行<clinit>(利用此特性可以实现线程安全的懒汉单例模式)。**

什么是类装载器ClassLoader

1. ClassLoader是一个抽象类
2. ClassLoader的实例将读入Java字节码将类装载到JVM中

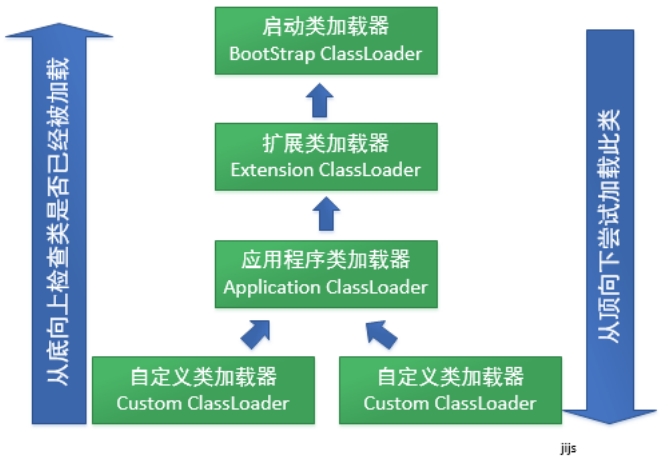
3. ClassLoader可以定制，满足不同的字节码流获取方式
4. ClassLoader负责类装载过程中的加载阶段。

JVM中的类加载器

1. **启动类加载器（BootStrap ClassLoader）：** 引导类装入器是用本地代码实现的类装入器，它负责将 jdk中jre/lib下面的核心类库或-Xbootclasspath选项指定的jar包加载到内存中。由于引导类加载器涉及到虚拟机本地实现细节，开发者无法直接获取到启动类加载器的引用，所以不允许直接通过引用进行操作。
2. **扩展类加载器（Extension ClassLoader）：** 扩展类加载器是由Sun的ExtClassLoader（sun.misc.Launcher$ExtClassLoader）实现的。它负责将jdk中jre/lib/ext或者由系统变量-Djava.ext.dir指定位置中的类库加载到内存中。开发者可以直接使用标准扩展类加载器。
3. **系统类加载器（System ClassLoader）：** 系统类加载器是由 Sun的AppClassLoader（sun.misc.Launcher$AppClassLoader）实现的。它负责将系统类路径java -classpath或-Djava.class.path变量所指的目录下的类库加载到内存中。开发者可以直接使用系统类加载器。

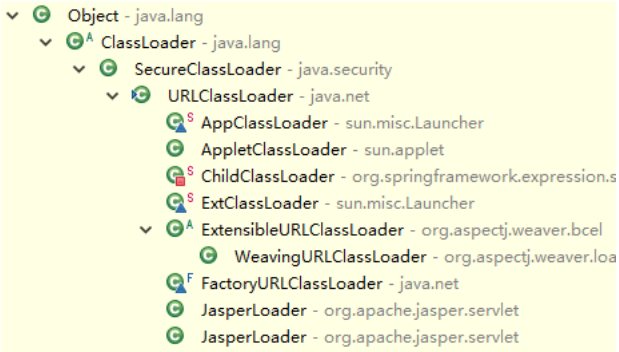
双亲委派模型

下图中展示了类加载器直接的关系和双亲委派模型



类加载器双亲委派模型

从图中我们发现除启动类加载器外，每个加载器都有父的类加载器。
双亲委派机制：如果一个类加载器在接到加载类的请求时，它首先不会自己尝试去加载这个类，而是把这个请求任务委托给父类加载器去完成，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。



ClassLoader类关系图

从类的继承关系来看，ExtClassLoader和AppClassLoader都是继承URLClassLoader，都是ClassLoader的子类。而BootStrapClassLoader是有C写的，不再java的ClassLoader子类中。

> **注意：**
> **从图中可以看到类加载器间的父子关系不是以继承的方式实现的，而是以组合关系的方式来复用父加载器的代码。**
> **如果一个类加载器收到了类加载的请求，它首先会把这个请求委派给父加载器去完成，每一个层次的类加载器都是如此。**

双亲委派模型的好处

Java类随着加载它的类加载器一起具备了一种带有优先级的层次关系。比如，Java中的Object类，它存放在rt.jar之中,无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此Object在各种类加载环境中都是同一个类。如果不采用双亲委派模型，那么由各个类加载器自己取加载的话，那么系统中会存在多种不同的Object类。

破坏双亲委派模型

案例一

双亲委派模型的问题：顶层ClassLoader，无法加载底层ClassLoader的类。
JDK的javax.xml.parsers包中定义了xml解析的类接口
Service Provider Interface SPI 位于rt.jar 即接口在启动ClassLoader中。而SPI的实现类，可能由第三方提供，AppClassLoader进行加载。
解决思路：可以在线程中放入底层的ClassLoader到Thread. setContextClassLoader()中，然后在顶层ClassLoader中使用Thread.getContextClassLoader()获得底层的ClassLoader进行加载第三方实现。

案例二

Tomcat中使用了自定ClassLoader，并且也破坏了双亲委托机制。
每个应用使用WebAppClassloader进行单独加载，他首先使用WebAppClassloader进行类加载，如果加载不了再委托父加载器去加载，这样可以保证每个应用中的类不冲突。每个tomcat中可以部署多个项目，每个项目中存在很多相同的class文件（很多相同的jar包），他们加载到jvm中可以做到互不干扰。

案例三：

利用破坏双亲委派来java的**类热部署**实现（每次修改类文件，不需要重启服务）。
因为一个Class只能被一个ClassLoader加载一次，否则会报java.lang.LinkageError。当我们想要实现代码热部署时，可以每次都new一个自定义的ClassLoader来加载新的Class文件。JSP的实现动态修改就是使用此特性实现。

Class加密实现思路

ClassLoader加载.class文件的方式不仅限于从jar包中读取，还可以从种地方读取，因为ClassLoader加载时需要的是byte[]数组.

> **ClassLoader加载Class文件方式：**
>
> 1. 从本地系统中直接加载
> 2. 通过网络下载.class文件
> 3. 从zip，jar等归档文件中加载.class文件
> 4. 从专有数据库中提取.class文件
> 5. 将Java源文件动态编译为.class文件

加密实现思路：加载Class文件的方式灵活，我们可以自定义ClassLoader，把加密后的Class文件，在加载Class前先进行解密，然后在通过ClassLoader进行加载。