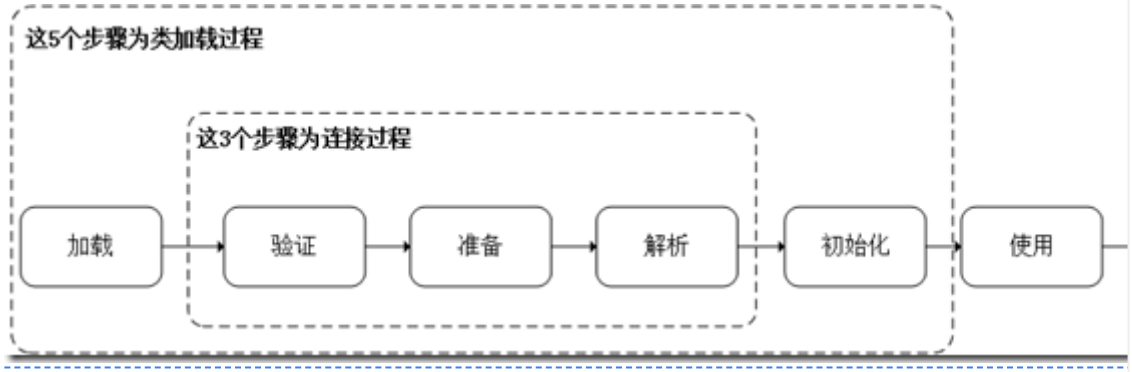
飘扬的红领巾

深入理解JVM（六）——类加载器原理

2017-08-25 17:41 by 飘扬的红领巾, 3637 阅读, 0 评论, 收藏, 编辑

我们知道我们编写的java代码，会经过编译器编译成字节码文件（class文件），再把字节码文件装载到JVM中，映射到各个内存区域中，我们的程序就可以在内存中运行了。那么字节码文件是怎样装载到JVM中的呢？中间经过了哪些步骤？常说的双亲委派模式又是怎么回事？本文主要搞清楚这些问题。

类装载流程



1、加载

About

李平，目前在一家O2O互联网公司从事设计、开发工作。业余时间喜欢跑步、看书、游戏。

喜欢简单而高效的工作环境，熟悉JavaEE、SOA、数据库架构、优化、系统运维，有大型门户网站，金融系统建设经验。RHCE、MySQL OCP。MyCAT开源项目成员。

我的开源项目：
mycat-eye
nosql-eye

昵称：飘扬的红领巾
园龄：6年9个月
荣誉：推荐博客
粉丝：910
关注：0
+加关注

加载是类装载的第一步，首先通过class文件的路径读取到二进制流，并解析二进制流将里面的元数据（类型、常量等）载入到方法区，在java堆中生成对应的java.lang.Class对象。

2、连接

连接过程又分为3步，验证、准备、解析

2.1、验证

验证的主要目的就是判断class文件的合法性，比如class文件一定是以0xCAFEBABE开头的，另外对版本号也会做验证，例如如果使用java1.8编译后的class文件要再java1.6虚拟机上运行，因为版本问题就会验证不通过。除此之外还会对元数据、字节码进行验证，具体的验证过程就复杂的多了，可以专门查看相关资料去了解。

2.2、准备

准备过程就是分配内存，给类的一些字段设置初始值，例如：

```
public static int v=1;
```

这段代码在准备阶段v的值就会被初始化为0，只有到后面类初始化阶段时才会被设置为1。

但是对于static final（常量），在准备阶段就会被设置成指定的值，例如：

```
public static final   int v=1;
```

这段代码在准备阶段v的值就是1。

2.3、解析

解析过程就是将符号引用替换为直接引用，例如某个类继承java.lang.object，原来的符号引用记录的是“java.lang.object”这个符号，凭借这个符号并不能找到java.lang.object这个对象在哪里？而直接引用就是要找到java.lang.object所在的内存地址，建立直接引用关系，这样就方便查询到具体对象。

3、初始化

初始化过程，主要包括执行类构造方法、static变量赋值语句，staic{}语句块，需要注意的是如果一个子类进行初始化，那么它会事先初始化其父类，保证父类在子类之前被初始化。所以其实在java中初始化一个类，那么必然是先初始化java.lang.Object，因为所有的java类都继承自java.lang.Object。

说完了类加载过程，我们来介绍一下这个过程当中的主角：类加载器。

类加载器

类加载器ClassLoader，它是一个抽象类，ClassLoader的具体实例负责把java字节码读取到JVM当中，ClassLoader还可以定制以满足不同字节码流的加载方式，比如从网络加载、从文件加载。ClassLoader的负责整个类装载流程中的“加载”阶段。

ClassLoader的重要方法：

```
1:  public Class<?> loadClass(String name) throws
ClassNotFoundException
```

载入并返回一个类。

并发(1)
并发 乐观锁 悲观锁(1)
大型网站(1)
代码质量 Checkstyle PMD JDepend Eclemma Metric(1)
更多

随笔分类

Apache Mina(1)
Eclipse(1)
Hibernate(2)
Java(19)
JVM(8)
MongoDB(2)
MySQL(4)
RCP/SWT/Jface(1)
SOA(1)
Spring(3)
持续集成(4)
大型网站(3)
多线程(1)
开源项目(2)
敏捷(1)
其他(7)
设计模式(1)
数据结构/算法(1)
系统架构(3)
支付(1)

2012年12月(1)
2012年11月(1)
2012年9月(1)
2012年6月(2)
2012年5月(4)
2012年3月(1)

```
1:  protected final Class<?> defineClass(byte[] b, int off, int
len)
```

定义一个类，该方法不公开被调用。

```
1:  protected Class<?> findClass(String name) throws
ClassNotFoundException
```

查找类，loadClass的回调方法

```
1:  protected final Class<?> findLoadedClass(String name)
```

查找已经加载的类。

系统中的ClassLoader

BootStrap Classloader（启动ClassLoader）

Extension ClassLoader （扩展ClassLoader）

App ClassLoader(应用 ClassLoader)

Custom ClassLoader(自定义ClassLoader)

每个ClassLoader都有另外一个ClassLoader作为父ClassLoader，BootStrap Classloader除外，它没有父Classloader。

ClassLoader加载机制如下：

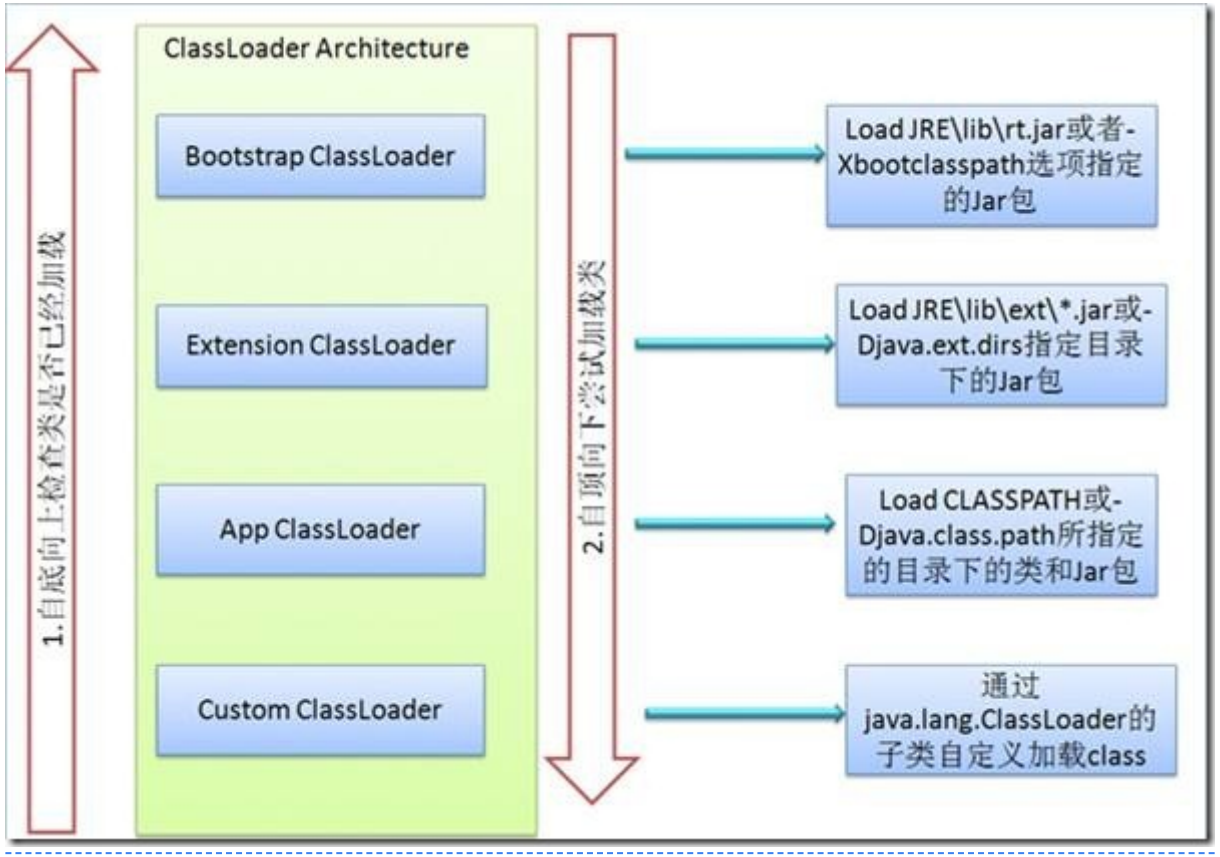
重构(1)

推荐排行榜

1. 大型网站系统架构的演化(211)
2. 大型网站的灵魂——性能(63)
3. 电商系统中的商品模型的分析与设计一续(51)
4. 电商系统中的商品模型的分析与设计(47)
5. 做了两款数据库监控工具，打算在近期开源(39)

阅读排行榜

1. 大型网站系统架构的演化(51206)
2. 深入理解JVM（一）——基本原理(34635)
3. 电商系统中的商品模型的分析与设计(16810)
4. 大型网站的灵魂——性能(15776)
5. 使用Maven+Nexus+Jenkins+Svn+Tomcat+Sonar搭建持续集成环境（一）(15209)



自下向上检查类是否被加载，一般情况下，首先从App ClassLoader中调用findLoadedClass方法查看是否已经加载，如果没有加载，则会交给父类，Extension ClassLoader去查看是否加载，还没加载，则再调用其父类，BootstrapClassLoader查看是否已经加载，如果仍然没有，自顶向下尝试加载类，那么从 Bootstrap ClassLoader到 App ClassLoader依次尝试加载。

值得注意的是即使两个类来源于相同的class文件，如果使用不同的类加载器加载，加载后的对象是完全不同的，这个不同反应在对象的 equals()、isAssignableFrom()、isInstance()等方法的返回结果，也包括了使用instanceof 关键字对对象所属关系的判定结果。

```
protected Class<?> loadClass(String name, boolean resolve)
    throws ClassNotFoundException
{
    synchronized (getClassLoadingLock(name)) {
        // First, check if the class has already been loaded
        Class<?> c = findLoadedClass(name);
        if (c == null) {
            long t0 = System.nanoTime();
            try {
                if (parent != null) {
                    c = parent.loadClass(name, false);
                } else {
                    c = findBootstrapClassOrNull(name);
                }
            } catch (ClassNotFoundException e) {
                // ClassNotFoundException thrown if class not found
                // from the non-null parent class loader
            }
```

从代码上可以看出，首先查看这个类是否被加载，如果没有则调用父类的loadClass方法，直到BootstrapClassLoader（没有父类），我们把这个过程叫做双亲模式，

双亲模式的问题

顶层ClassLoader，无法加载底层ClassLoader的类

Java框架(rt.jar)如何加载应用的类？

比如：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，在AppLoader。

这样就无法用BootstrapClassLoader去加载SPI的实现类。

解决

JDK中提供了一个方法：

```
1:  Thread. setContextClassLoader()
```

用以解决顶层ClassLoader无法访问底层ClassLoader的类的问题；
基本思想是，在顶层ClassLoader中，传入底层ClassLoader的实例。

双亲模式的破坏

双亲模式是默认的模式，但不是必须这么做；
Tomcat的WebappClassLoader 就会先加载自己的Class，找不到再委托parent；
OSGi的ClassLoader形成网状结构，根据需要自由加载Class。

小结

本文介绍了类加载的流程，以及ClassLoader工作机制，最后分析双亲模式的缺陷，以及如何弥补该缺陷，介绍了tomcat、OSGI如何自定义类加载流程。

参考资料：

《实战Java虚拟机》 葛一鸣

《深入理解Java虚拟机（第2版）》 周志明

飘扬的红领巾
关注 - 0
粉丝 - 910

0 0

荣誉：推荐博客

+加关注

« 上一篇：深入理解JVM（五）——垃圾回收器
» 下一篇：深入理解JVM（七）——性能监控工具

分类: Java,JVM