jvm系列 (五) ---类加载机制

<http://www.cnblogs.com/-new/p/7514885.html>

# [jvm系列 (五) ---类加载机制](http://www.cnblogs.com/-new/p/7514885.html)

# 类的加载机制

## 目录

* [jvm系列（一）：jvm内存区域与溢出](http://www.cnblogs.com/-new/p/7244460.html)
* [jvm系列（二）：垃圾收集器与内存分配策略](http://www.cnblogs.com/-new/p/7284368.html)
* [jvm系列（三）：锁的优化](http://www.cnblogs.com/-new/p/7427036.html)
* [jvm系列 (四) ---强、软、弱、虚引用](http://www.cnblogs.com/-new/p/7491269.html)
* [我的博客目录](http://www.cnblogs.com/-new/p/7396648.html)

## 什么是类的加载机制

* 简单的说，就是虚拟机把类的数据从Class文件加载到内存，对数据进行检验，转换解析和初始化，最终形成可以被虚拟机直接使用的java类型

# 类加载的过程

* 类加载的过程：加载(注意和类加载区别开来)，验证，准备，解析，初始化

## 加载

* 通过一个类的全限定名来获取其定义的二进制流
* 将这个字节流所代表的静态存储结构（类信息，静态常量、常量）转化成方法区的运行时数据结构
* 在内存中生成一个代表这个类的Class对象，作为方法区中这些数据的访问入口

## 验证

* 目的：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，并且不会危害虚拟机的安全。Class文件可以从任何途径来，包括我们手动编辑一个Class文件，如果我们不对他进行验证，那么就可能会出现安全问题。
* 文件格式验证：是否符合Class文件的格式的规范，能够被当前版本的虚拟机处理。通过这个验证，可以保证字节流能够正确解析并存储于方法区，格式上符合一个Java类型信息的要求。只有通过这个验证后，字节流才会进入内存的方法区进行存储。
* 元数据验证，对字节码描述的信息进行语义分析，比如这个类是否有父类，是否继承了final修饰的类，保证描述的信息符合java语言规范。
* 字节码验证，通过数据流和控制流分析，确定程序语义是和合法的，符合逻辑的。这个阶段对类的方法体进行验证分析，保证在类的方法运行时不会危害虚拟机。比如说验证类型转换是否合理，比如说父类不能转为子类
* 符号引用验证，对常量池中的各种符号引用的信息进行匹配性校验，比如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。确保解析动作能够正常执行。

## 准备

* 为类的静态变量分配内存并设置类常量的初始值

public static int value = 123

* 这段代码经过准备阶段后，value的值是0

public static final int value = 123

* 被static和final同时修饰的，经过准备阶段后，value是123

## 解析

* 虚拟机将常量池内的符号引用（类和接口的全限定名、字段的名称和描述符、方法的名称和描述符）替换为直接引用（可以是直接指向目标的指针、相对偏移量或是一个能间接定位到目标的句柄）的过程

## 初始化

* 准备阶段变量已经赋过系统要求的初始值，而在初始化阶段，则根据程序员通过程序制定的主观计划去初始化类变量和其他资源。说白了就是初始化静态变量指定的值和执行静态代码块

### 必须初始化的情况

* 用new创建对象（准确来说是new指令）的时候
* 读取设置一个类的静态字段，调用类的静态方法
* 使拥reflect包的方法对类进行反射调用的时候
* 初始一个类的时候，如果一个类的父类还没有初始化的时候，触发父类的初始化
* 主类（main方法的类）

public class SuperClass

{

public static int value = 123;

static

{

System.out.println("SuperClass init");

}

}

public class SubClass extends SuperClass

{

static

{

System.out.println("SubClass init");

}

}

public class Test

{

public static void main(String[] args)

{

System.out.println(SubClass.value);

}

}

//输出：SuperClass init

123

* 通过子类来获取父类的静态字段，只会触发父类的初始化不会触发子类的初始化

public static void main(String[] args)

{

SuperClass s[]=new SuperClass[10];

}

* 并没有输出SuperClass init，也就是没有初始化这个类，通过反编译发现用的指令是newarray，所以并没有触发初始化，如果不是创建数组而是创建对象的话，会输出SuperClass init，通过反编译发现指令是new而不是newarray

class ConstClass

{

public static final String HELLOWORLD = "Hello World";

static

{

System.out.println("ConstCLass init");

}

}

public class TestMain

{

public static void main(String[] args)

{

System.out.println(ConstClass.HELLOWORLD);

}

}

* 并没有输出ConstClass init，也就是说明并没有触发该类初始化，这是因为在编译阶段通过常量传播优化，这个字符串已经存储在NotInitialization类的常量池中，也就是实际上对这个常量的引用变为NotInitialization类对自身常量池的引用了

# 类加载器

* 通过一个类的全限定名来获取其定义的二进制字节流的代码模块称为类的加载器
* 一个类需要加载他的类加载器和这个类本身来确定在虚拟机的唯一性，也就是说两个类来源一个Class文件，如果加载他们的类加载器不同，那么这两个类不同
* public static void main(String[]args) throws Exception{
* ClassLoader myLoader=new ClassLoader(){
* @Override
* public Class<?>loadClass(String name)throws ClassNotFoundException{
* try{
* String fileName=name.substring(name.lastIndexOf(".")+1)+".class";
* InputStream is=getClass().getResourceAsStream(fileName);
* if(is==null){
* return super.loadClass(name);
* }
* byte[]b=new byte[is.available()];
* is.read(b);
* return defineClass(name,b,0,b.length);
* }
* catch(IOException e){
* throw new ClassNotFoundException(name);
* }
* }
* }
* ;
* Object obj=myLoader.loadClass("Person").newInstance();
* System.out.println(obj.getClass());
* System.out.println(obj instanceof Person);//false

}

* 在这里自己定义一个类加载器，而在这里实际上是存在两个Person类，一个是我们自己定义的类加载器加载的，一个是系统应用程序类加载器加载的
* 从虚拟机的角度，只有两种不同的类加载器，一种是启动类加载器，C++实现，是虚拟机的一部分。另一部分就是其他的类加载器，有java实现，继承自抽象类ClassLoader，属于虚拟机之外。
* 从java开发人员的角度看，分为启动类加载器，扩张类加载器，应用类加载器。

## 启动类加载器（Bootstrap ClassLoader）

* 加载的是JAVA\_HOME/lib下的类库，启动类加载器无法被java程序直接引用

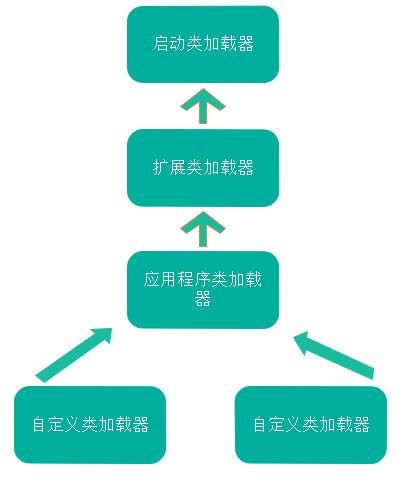
## 扩展类加载器（Extension ClassLoader）

* 这个类加载器由sun.misc.Launcher$ExtClassLoader实现，它负责用于加载JAVA\_HOME/lib/ext目录中的，或者被java.ext.dirs系统变量指定所指定的路径中所有类库，开发者可以直接使用扩展类加载器。

## 应用程序类加载器（Application ClassLoader）

* 由sun.misc.Launcher$AppClassLoader实现，负责加载用户类路径（java.class.path）上指定的类库，开发者可以直接使用这个类加载器

## 双亲委派模型



* 图描述的就是所谓的双亲委托模型，除了顶层的启动类加载器外，其他加载器都有他的父加载器，但这不是继承关系，而是使用组合来复用父加载器的代码
* 工作过程：如果一个类收到类加载的请求，它首先不会自己去尝试去加载这个类，而是把这个请求委托给父类加载器去完成，所有的加载请求最后都会传授到顶层的启动类加载器，只有当父类加载器在他的搜索范围找不到所需的类的时候，也就是无法完成加载请求的时候，子加载器才会尝试去加载。

## 双亲委托模型的实现

protected synchronized Class<?> loadClass(String name, boolean resolve)

307 throws ClassNotFoundException

308 {

309 // First, check if the class has already been loaded检查是否已经被加载

310 Class c = findLoadedClass(name);

//如果还没被加载

311 if (c == null) {

312 try {

//父加载器是否为null，因为启动类加载器是C++代码编写的，所以父加载器为null的时候，说明此时父加载器是启动类加载器。如果父加载器不为空，让父加载器去加载

313 if (parent != null) {

314 c = parent.loadClass(name, false);

315 } else {

316 c = findBootstrapClass0(name);

317 }

318 } catch (ClassNotFoundException e) {

319 // If still not found, then invoke findClass in order如果父加载器无法加载，调用本身的方法的来进行类加载器

320 // to find the class.

321 c = findClass(name);

322 }

323 }

324 if (resolve) {

325 resolveClass(c);

326 }

327 return c;

328 }

## 为什么要双亲委派模型

* 例如java.lang.Object，存放于rt.jar中，无论哪一个类加载器要去加载这个类，最终都是由Bootstrap ClassLoader去加载，因此Object类在程序的各种类加载器环境中都是一个类。相反，如果没有双亲委派模型，由各个类加载器自己去加载的话，如果用户自己编写了一个java.lang.Object，并放在CLASSPATH下，那么系统会出现多个不同的Objcet类
* 如果两个类加载器都要加载一个Class，如果没有双亲委托，各自去加载，那么会加载两份Class文件