深入JVM内核——原理、诊断与优化 第6周

DATAGURU专业数据分析社区

法律声明

DATAGURU 炼数成金

【声明】本视频和幻灯片为炼数成金网络课程的教学资料，所有资料只能在课程内使用，不得在课程以外范围散播，违者将可能被追究法律和经济责任。

课程详情访问炼数成金培训网站

智能程序员www.zncxy.com免费提供下载

类装载器

- class装载验证流程
- 什么是类装载器ClassLoader
- JDK中ClassLoader默认设计模式
- 打破常规模式
- 热替换

- **智能程序员www.zncxy.com免费提供下载**

class装载验证流程

- 加载
- 链接
  - 验证
  - 准备
  - 解析
- 初始化

- **智能程序员www.zncxy.com免费提供下载**

class装载验证流程 -加载

- 装载类的第一个阶段
- 取得类的二进制流
- 转为方法区数据结构
- 在Java堆中生成对应的java.lang.Class对象

class装载验证流程 -链接 验证

- 链接 -> 验证
  - 目的：保证Class流的格式是正确的
    - 文件格式的验证
      - 是否以0xCAFEBABE开头
      - 版本号是否合理
    - 元数据验证
      - 是否有父类
      - 继承了final类？
      - 非抽象类实现了所有的抽象方法
    - 字节码验证 (很复杂)
      - 运行检查
      - 栈数据类型和操作码数据参数吻合
      - 跳转指令指定到合理的位置
    - 符号引用验证
      - 常量池中描述类是否存在
      - 访问的方法或字段是否存在且有足够的权限
      -

class装载验证流程 -链接 准备

- 链接 -> 准备
  - 分配内存，并为类设置初始值（方法区中）
    - public static int v=1;
    - 在准备阶段中，v会被设置为0
    - 在初始化的<clinit>中才会被设置为1
    - 对于static final类型，在准备阶段就会被赋上正确的值
    - public static final int v=1;

class装载验证流程 - 链接 解析

- 链接 -> 解析
  - 符号引用替换为直接引用

class装载验证流程 – 初始化

- 执行类构造器<clinit>
  - static变量 赋值语句
  - static{}语句
- 子类的<clinit>调用前保证父类的<clinit>被调用
- <clinit>是线程安全的

class装载验证流程

- Java.lang.NoSuchFieldError错误可能在什么阶段抛出

什么是类装载器ClassLoader

- ClassLoader是一个抽象类
- ClassLoader的实例将读入Java字节码将类装载到JVM中
- ClassLoader可以定制，满足不同的字节码流获取方式
- ClassLoader负责类装载过程中的加载阶段

JDK中ClassLoader默认设计模式

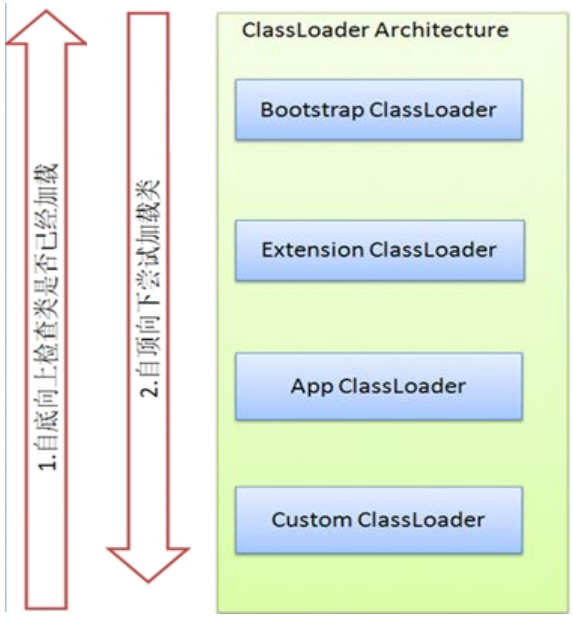
- ClassLoader的重要方法
  - public Class<?> loadClass(String name) throws ClassNotFoundException
    - 载入并返回一个Class
  - protected final Class<?> defineClass(byte[] b, int off, int len)
    - 定义一个类，不公开调用
  - protected Class<?> findClass(String name) throws ClassNotFoundException
    - loadClass回调该方法，自定义ClassLoader的推荐做法
  - protected final Class<?> findLoadedClass(String name)
    - 寻找已经加载的类

JDK中ClassLoader默认设计模式 – 分类

- BootStrap ClassLoader（启动ClassLoader）
- Extension ClassLoader（扩展ClassLoader）
- App ClassLoader（应用ClassLoader/系统ClassLoader）
- Custom ClassLoader(自定义ClassLoader)

- 每个ClassLoader都有一个Parent作为父亲

JDK中ClassLoader默认设计模式 – 协同工作



rt.jar /-Xbootclasspath

%JAVA_HOME%/lib/ext/*.jar

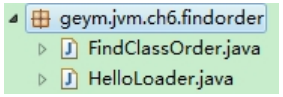
Classpath下

完全自定义路径

JDK中ClassLoader默认设计模式 – 协同工作

```
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
            }
```

JDK中ClassLoader默认设计模式




```
public class HelloLoader {
        public void print(){
            System.out.println("I am in apploader");
        }
}
```

```
public class HelloLoader {
        public void print(){
            System.out.println("I am in bootloader");
        }
}
```


```
public class FindClassOrder {
        public static void main(String args[]){
            HelloLoader loader=new HelloLoader();
            loader.print();
        }
}
```

JDK中ClassLoader默认设计模式

- 直接运行以上代码：
  - I am in apploader
- 加上参数 **-Xbootclasspath/a:D:/tmp/clz**
  - I am in bootloader
  - 此时AppLoader中不会加载HelloLoader
    - I am in apploader 在classpath中却没有加载
    - 说明类加载是从上往下的

JDK中ClassLoader默认设计模式

强制在apploader中加载

```
public static void main(String args[]) throws Exception {
        ClassLoader cl=FindClassOrder2.class.getClassLoader();
        byte[] bHelloLoader=loadClassBytes("geym.jvm.ch6.findorder.HelloLoader");
        Method md_defineClass=ClassLoader.class.getDeclaredMethod("defineClass", byte[].class,int.class,int.class);
        md_defineClass.setAccessible(true);
        md_defineClass.invoke(cl, bHelloLoader,0,bHelloLoader.length);
        md_defineClass.setAccessible(false);

        HelloLoader loader = new HelloLoader();
        System.out.println(loader.getClass().getClassLoader());
        loader.print();
}
```

-Xbootclasspath/a:D:/tmp/clz

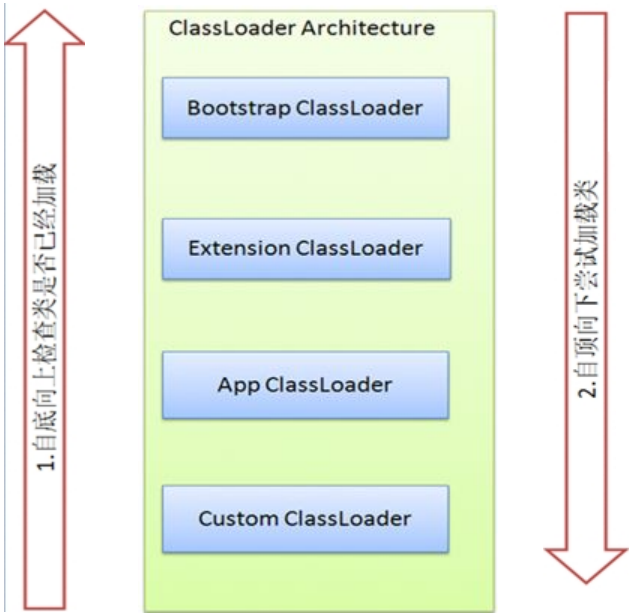
I am in apploader

在查找类的时候，先在底层的Loader查找，是从下往上的。Apploader能找到，就不会去上层加载器加载

JDK中ClassLoader默认设计模式

能否只用反射，仿照上面的写法，将类注入启动ClassLoader

JDK中ClassLoader默认设计模式 – 问题



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？

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。

JDK中ClassLoader默认设计模式 – 解决

- Thread. setContextClassLoader()
  - 上下文加载器
  - 是一个**角色**
  - 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

JDK中ClassLoader默认设计模式 – 解决

```
static private Class getProviderClass(String className, ClassLoader cl,
        boolean doFallback, boolean useBSClsLoader) throws ClassNotFoundException
{
    try {
        if (cl == null) {
            if (useBSClsLoader) {
                return Class.forName(className, true, FactoryFinder.class.getClassLoader());
            } else {
                cl = ss.getContextClassLoader();
                if (cl == null) {
                    throw new ClassNotFoundException();
                }
                else {
                    return cl.loadClass(className); //使用上下文ClassLoader
                }
            }
        }
        else {
            return cl.loadClass(className);
        }
    }
    catch (ClassNotFoundException e1) {
        if (doFallback) {
            // Use current class loader - should always be bootstrap CL
            return Class.forName(className, true, FactoryFinder.class.getClassLoader());
        }
.....
```

代码来自于javax.xml.parsers.FactoryFinder
展示如何在启动类加载器加载AppLoader的类

上下文ClassLoader可以突破双亲模式的局限性

JDK中ClassLoader默认设计模式

- 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

JDK中ClassLoader默认设计模式

- 破坏双亲模式例子- 先从底层ClassLoader加载

OrderClassLoader的部分实现

```
protected synchronized Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException {
    // First, check if the class has already been loaded
    Class re=findClass(name);
    if(re==null){
        System.out.println( “无法载入类:” +name+ “ 需要请求父加载器");
        return super.loadClass(name,resolve);
    }
    return re;
}
```

JDK中ClassLoader默认设计模式

```
protected Class<?> findClass(String className) throws ClassNotFoundException {
Class clazz = this.findLoadedClass(className);
if (null == clazz) {
    try {
        String classFile = getClassFile(className);
        FileInputStream fis = new FileInputStream(classFile);
        FileChannel fileC = fis.getChannel();
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        WritableByteChannel outC = Channels.newChannel(baos);
        ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
         省略部分代码
        fis.close();
        byte[] bytes = baos.toByteArray();

        clazz = defineClass(className, bytes, 0, bytes.length);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
return clazz;
}
```

JDK中ClassLoader默认设计模式

```
OrderClassLoader myLoader=new OrderClassLoader("D:/tmp/clz/");
Class clz=myLoader.loadClass("geym.jvm.ch6.classloader.DemoA");
System.out.println(clz.getClassLoader());

System.out.println("==== Class Loader Tree ====");
ClassLoader cl=myLoader;
while(cl!=null){
    System.out.println(cl);
    cl=cl.getParent();
}
```

因为先从OrderClassLoader加载，找不到Object，之后使用appLoader加载Object

DemoA在ClassPath中，但由OrderClassLoader加载

```
java.io.FileNotFoundException: D:\tmp\clz\java\lang\Object.class (系统找不到指定的路径。)
        at java.io.FileInputStream.open(Native Method)
        .....
        at geym.jvm.ch6.classloader.ClassLoaderTest.main(ClassLoaderTest.java:7)
无法载入类:java.lang.Object需要请求父加载器
geym.jvm.ch6.classloader.OrderClassLoader@18f5824
==== Class Loader Tree ====
geym.jvm.ch6.classloader.OrderClassLoader@18f5824
sun.misc.Launcher$AppClassLoader@f4f44a
sun.misc.Launcher$ExtClassLoader@1d256fa
```

JDK中ClassLoader默认设计模式

如果OrderClassLoader不重载loadClass()，只重载findClass，那么程序输出为

```
sun.misc.Launcher$AppClassLoader@b23210
==== Class Loader Tree ====
geym.jvm.ch6.classloader.OrderClassLoader@290fbc
sun.misc.Launcher$AppClassLoader@b23210
sun.misc.Launcher$ExtClassLoader@f4f44a
```

DemoA由AppClassLoader加载

热替换

- 含义：
  - –当一个class被替换后，系统无需重启，替换的类立即生效
  - –例子：
    - geym.jvm.ch6.hot.CVersionA

```
public class CVersionA {
        public void sayHello() {
                System.out.println("hello world! (version A)");
        }
}
```

热替换

- DoopRun 不停调用CVersionA . sayHello()方法，因此有输出：
  - hello world! (version A)
- 在DoopRun 的运行过程中，替换CVersionA 为：

```
public class CVersionA {
        public void sayHello() {
                System.out.println("hello world! (version B)");
        }
}
```

- 替换后， DoopRun 的输出变为
  - hello world! (version B)

没有做不到，只有想不到

Thanks

FAQ时间