ClassLoader 分析

http://www.blogjava.net/landy/archive/2006/05/21/47391.html

前言

ClassLoader 是 Java 虚拟机 (JVM) 的类装载子系统，它负责将 Java 字节码装载到 JVM 中，并使其成为 JVM 一部分。JVM 的类动态装载技术能够在运行时刻动态地加载或者替换系统的某些功能模块，而不影响系统其他功能模块的正常运行。本文将分析 JVM 中的类装载系统，探讨 JVM 中类装载的原理、实现以及应用。

装载过程简介

类装载就是寻找一个类或是一个接口的字节码文件并通过解析该字节码来构造代表这个类或是这个接口的 Class 对象的过程。在 Java 中，类装载器把一个类装入 Java 虚拟机中，要经过三个步骤来完成：装载、链接和初始化，其中链接又可以分成校验、准备和解析三步，除了解析外，其它步骤是严格按照顺序完成的，各个步骤的主要工作如下：

a) 装载：查找和导入类或接口的字节码；

b) 链接：执行下面的校验、准备和解析步骤，其中解析步骤是可以选择的；

   i. 校验：检查导入类或接口的二进制数据的正确性；

   ii. 准备：给类的静态变量分配并初始化存储空间；

   iii. 解析：将符号引用转成直接引用；

c) 初始化：激活类的静态变量的初始化 Java 代码和静态 Java 代码块。

至于在类装载和虚拟机启动的过程中的具体细节和可能会抛出的错误，请参看《Java 虚拟机规范》以及《深入 Java 虚拟机》。由于本文的讨论重点不在此就不再多叙述。

装载的实现

JVM 中类的装载是由 ClassLoader 和它的子类来实现的。Java ClassLoader 是一个重要的 Java 运行时系统组件，它负责在运行时查找和装入 Java 字节码。

在 Java 中，ClassLoader 是一个抽象类，它在包 java.lang 中。可以这样说，只要了解了 ClassLoader 中的一些重要的方法，再结合上面所介绍的 JVM 中类装载的具体的过程，对动态装载类这项技术就有了一个比较大概的掌握，这些重要的方法包括以下几个：

1. loadCass 方法 loadClass(String name, boolean resolve) 其中 name 参数指定了 JVM 需要的类的名称，该名称以类的全限定名表示，如 Java.lang.Object ； resolve 参数告诉方法是否需要解析类，在初始化类之前，应考虑类解析，并不是所有的类都需要解析，如果 JVM 只需要知道该类是否存在或找出该类的超类，那么就不需要解析。这个方法是 ClassLoader 的入口点。
2. defineClass 方法 这个方法接受类文件的字节数组并把它转换成 Class 对象。字节数组可以是从本地文件系统或网络装入的数据。它把字节码分析成运行时数据结构、校验有效性等等。
3. findSystemClass 方法 findSystemClass 方法从本地文件系统装入 Java 字节码。它在本地文件系统中寻找类文件，如果存在，就使用 defineClass 将字节数组转换成 Class 对象。当运行 Java 应用程序时，这是 JVM 正常装入类的缺省机制。

4. resolveClass 方法 resolveClass(Class c) 方法解析装入的类，如果该类已经被解析过那么将不做处理。当调用 loadClass 方法时 ,通过它的 resolve 参数决定是否要进行解析。
5. findLoadedClass 方法 当调用 loadClass 方法装入类时，调用 findLoadedClass 方法来查看 ClassLoader 是否已装入这个类，如果已装入，那么返回 Class 对象，否则返回 NULL。如果强行装载已存在的类，将会抛出链接错误。

装载的应用

一般来说，我们实现自定义的 ClassLoader 需要继承抽象类 java.lang.ClassLoader，其中必须实现的方法是 loadClass(String name) ，对于这个方法需要实现如下操作：

(1) 确认类的名称；

(2) 检查请求要装载的类是否已经被装载；

(3) 检查请求加载的类是否是系统类；

(4) 尝试从类装载器的存储区获取所请求的类；

(5) 在虚拟机中定义所请求的类；

(6) 解析所请求的类；

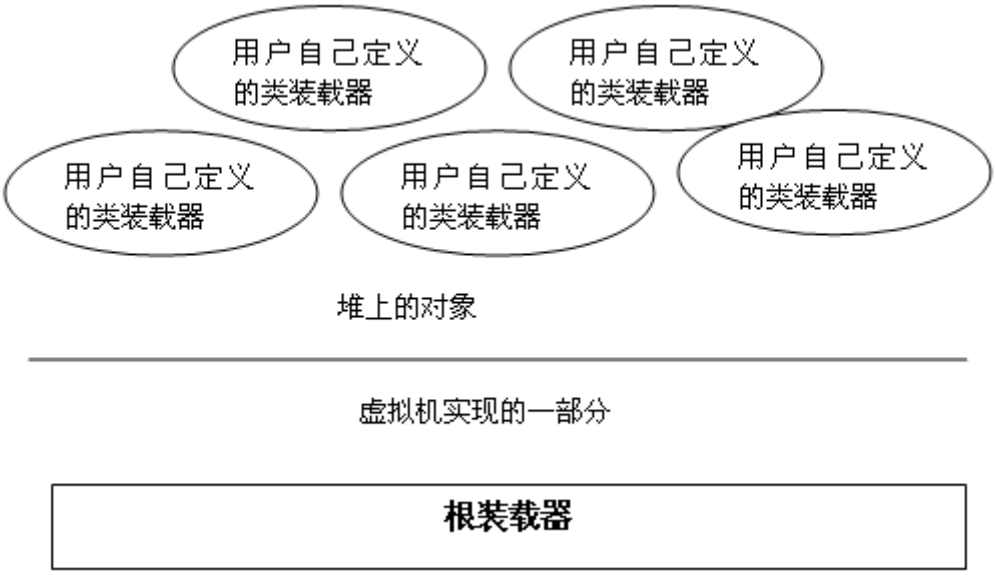
(7) 返回所请求的类。

所有的 Java 虚拟机都包括一个内置的类装载器，这个内置的类库装载器被称为根装载器 (bootstrap ClassLoader) 。根装载器的特殊之处是它只能够装载基本的 Java 类，如 rt.jar 中的 class。当应用程序可以使用用户自定义的 ClassLoader 来加载特定 ClassPath 下的类。下面的例子是 JDK5.0 的 URLClassLoader 的实现。

```
    public class URLClassLoader extends SecureClassLoader {
     ......
     protected Class<?> findClass(final String name)
             throws ClassNotFoundException {
             try {
                 return (Class)AccessController.doPrivileged(new
PrivilegedExceptionAction() {
                     public Object run() throws ClassNotFoundException {
                             // 由类的全限定名得到物理路径
                             String path = name.replace('.',
'/').concat(".class");
                             // 从 URLs ClassPath 中取得相应的字节码
                             Resource res = ucp.getResource(path, false);
                             if (res != null) {
                                 try {
                                     // 由 Java 字节码创建一个 Class 对象
                                     return defineClass(name, res);
                                  } catch (IOException e) {
                                     throw new ClassNotFoundException(name, e);
                                  }
                              } else {
                                  throw new ClassNotFoundException(name);
                              }
                         }
                     }, acc);
             } catch (java.security.PrivilegedActionException pae) {
                 throw (ClassNotFoundException) pae.getException();
             }
         }
     ......
    }
```

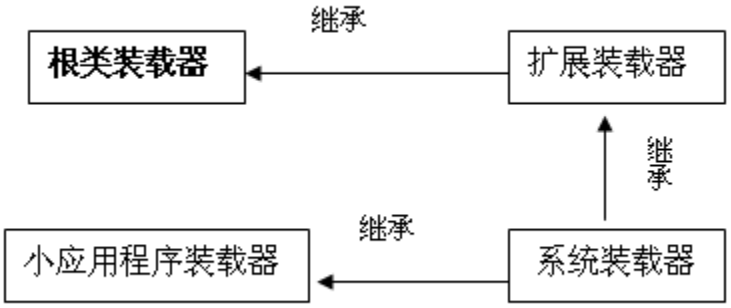
Java 虚拟机的类装载原理

前面我们已经知道，一个 Java 应用程序使用两种类型的类装载器：根装载器(bootstrap) 和用户定义的装载器 (user-defined) 。根装载器是 Java 虚拟机实现的一部分。根装载器以某种默认的方式将类装入，包括 Java API 的类。在运行期间，一个 Java 程序能使用用户自己定义的类装载器。根装载器是虚拟机固有的一部分，而用户定义的类装载器则不是，它是用 Java 语言写的，被编译成 class 文件之后然后再被装入到虚拟机，并像其它的任何对象一样可以被实例化。Java 类装载器的体系结构如下所示：



Java 的类装载的体系结构

Java 的类装载模型是一种代理 (delegation) 模型。当 JVM 要求类装载器 CL(ClassLoader) 装载一个类时，CL 首先将这个类装载请求转发给他的父装载器。只有当父装载器没有装载并无法装载这个类时，CL 才获得装载这个类的机会。这样，所有类装载器的代理关系构成了一种树状的关系。树的根是类的根装载器 (bootstrap ClassLoader) ，在 JVM 中它以 "null" 表示。除根装载器以外的类装载器有且仅有一个父装载器。在创建一个装载器时，如果没有显式地给出父装载器，那么 JVM 将默认系统装载器为其父装载器。 Java 的基本类装载器代理结构如图 2 所示：



Java 类装载的代理结构

下面针对各种类装载器分别进行详细的说明。

根 (Bootstrap) 装载器：该装载器没有父装载器，它是 JVM 实现的一部分，从 sun.boot.class.path 装载运行时库的核心代码。

扩展 (Extension) 装载器：继承的父装载器为根装载器，不像根装载器可能与运行时的操作系统有关，这个类装载器是用纯 Java 代码实现的，它从 java.ext.dirs (扩展目录) 中装载代码。

系统 (System or Application) 装载器：装载器为扩展装载器，我们都知道在安装 JDK 的时候要设置环境变量 (CLASSPATH)，这个类装载器就是从 java.class.path(CLASSPATH 环境变量) 中装载代码的，它也是用纯 Java 代码实现的，同时还是用户自定义类装载器的缺省父装载器。

小应用程序 (Applet) 装载器：父装载器为系统装载器，它从用户指定的网络上的特定目录装载小应用程序代码。

在设计一个类装载器的时候，应该满足以下两个条件：

对于相同的类名，类装载器所返回的对象应该是同一个类对象

如果类装载器 CL1 将装载类 C 的请求转给类装载器 CL2，那么对于以下的类或接口，CL1 和 CL2 应该返回同一个类对象：

a) S 为 C 的直接超类；

b) S 为 C 的直接超接口；

c) S 为 C 的成员变量的类型；

d) S 为 C 的成员方法或构建器的参数类型；

e) S 为 C 的成员方法的返回类型。

每个已经装载到 JVM 中的类对象都含有装载它的类装载器的信息。Class 类的方法 getClassLoader可以得到装载这个类的类装载器。一个类装载器认识的类包括它的父装载器认识的类和它自己装载的类，由此可见**类装载器认识的类是它自己装载的类的超集**。注意，我们可以得到类装载器的有关的信息，但是**已经装载到 JVM 中的类是不能更改它的类装载器的**。

Java 中的类的装载过程也就是代理装载的过程。比如： Web 浏览器中的 JVM 需要装载一个小应用程序 TestApplet。JVM 调用小应用程序装载器 ACL(Applet ClassLoader)来完成装载。 ACL 首先请求它的父装载器，即系统装载器装载 TestApplet 是否装载了这个类，由于 TestApplet 不在系统装载器的装载路径中，所以系统装载器没有找到这个类，也就没有装载成功。接着 ACL 自己装载 TestApplet。 ACL 通过网络成功地找到了 TestApplet.class 文件并将它导入到了 JVM 中。在装载过程中，JVM 发现 TestAppet 是从超类 java.applet.Applet 继承的。所以 JVM 再次调用 ACL 来装载 java.applet.Applet 类。 ACL 又再次按上面的顺序装载 Applet 类，结果 ACL 发现他的父装载器已经装载了这个类，所以 ACL 就直接将这个已经装载的类返回给了 JVM，完成了 Applet 类的装载。接下来，Applet 类的超类也一样处理。最后， TestApplet 及所有有关的类都装载到了 JVM 中。

总结

类的动态装载机制是 JVM 的一项核心技术，也是容易被忽视而引起很多误解的地方。本文介绍了 JVM 中类装载的原理、实现以及应用，分析了 ClassLoader 的结构、用途以及如何利用自定义的 ClassLoader 装载并执行 Java 类，希望能使大家对 JVM 中的类装载有一个比较深入的理解。