链滴

(转) Java虚拟机类加载顺序研究

作者：jaz

原文链接：https://ld246.com/article/1466237549584

来源网站：链滴

<p>当JVM（<a href="http://lib.csdn.net/base/17" class="replace_word" title="Java EE知识库"
target="_blank">Java虚拟机）启动时，会形成由三个类加载器组成的初始类加载器层次结构
</p>
<p>   Bootstrap Classloader
 &
bsp;   &
bsp; |
   &
bsp;   Extension Classloader
     &
bsp;   &
bsp; |
        S
stem Classloader</p>
<p>1.Bootstrap Classloader - 引导（也称为原始）类加载器,这个加载器的是非常特殊的，它实际
不是 java.lang.ClassLoader的子类，而是由JVM自身实现的</p>
<p>它所加载的有如下的资源：</p>
<p>D:/Program Files/Java/jre6/lib/resources.jar
D:/Program Files/Java/jre6/lib/rt.jar
D:/Program Files/Java/jre6/lib/sunrsasign.jar
D:/Program Files/Java/jre6/lib/jsse.jar
D:/Program Files/Java/jre6/lib/jce.jar
D:/Program Files/Java/jre6/lib/charsets.jar<br /
D:/Program Files/Java/jre6/classes</p>
<p>这时大家知道了为什么我们不需要在系统属性CLASSPATH中指定这些类库了吧，因为JVM在启
的时候就自动加载它们了。

2.Extension Classloader - 扩展类加载器 ExtClassLoader
它负责加载JRE的扩展目录（JAVA_HOME/jre/lib/ext或者由java.ext.dirs系统属性指定的）中JAR的
包，它已经属于最上层的加载器了。它所加载的有如下的资源：</p>
<p>D:/Program Files/Java/jre6/lib/ext;C:/WINDOWS/Sun/Java/lib/ext</p>
<p>
3. System Classloader - 系统（也称为应用）类加载器AppClassLoader 加载来自在命令
ava中的-classpath或者java.class.path系统属性或者 CLASSPATH操作系统属性所指定的JAR类包和
路径。也就是加载项目中用到的第三方包和项目中的类，它的父加载器为ExtClassLoader</p>
<p>它所加载的有如下的资源：</p>
<p>E:/works/encoder/lib/antlr-2.7.6.jar
E:/works/encoder/lib/cglib-2.2.jar
E:/work
/encoder/lib/cglib-nodep-2.1_3.jar
E:/works/encoder/lib/commons-codec-1.1.jar
:/works/encoder/lib/commons-collections-3.2.jar
E:/works/encoder/lib/commons-lang-
.1.jar
E:/works/encoder/lib/commons-logging-1.0.4.jar
E:/works/encoder/lib/co
mons-logging.jar
E:/works/encoder/lib/dom4j-1.4.jar
E:/works/encoder/lib/ehcac
e-0.9.jar
E:/works/encoder/lib/gameserver_core_1.0.1.jar
E:/works/encoder/lib/<a
href="http://lib.csdn.net/base/17" class="replace_word" title="Java EE知识库" target="_blank
>hibernate-annotations.jar
E:/works/encoder/lib/hibernate-commons-annotations.
ar
E:/works/encoder/lib/hibernate-entitymanager.jar
E:/works/encoder/lib/hibern
te-tools.jar
E:/works/encoder/lib/hibernate2.jar
E:/works/encoder/lib/hibernate3.j
r
E:/works/encoder/lib/jakarta-oro.jar
E:/works/encoder/lib/jasypt-1.5.jar
E:
works/encoder/lib/javassist-3.9.0.GA.jar
E:/works/encoder/lib/jta.jar
E:/works/enc
der/lib/jzlib-1.0.7.jar
E:/works/encoder/lib/libthrift.jar
E:/works/encoder/lib/log4j-
.2.15.jar
E:/works/encoder/lib/mina-integration-beans-2.0.0-M4.jar
E:/works/enc
der/lib/<a href="http://lib.csdn.net/base/14" class="replace_word" title="MySQL知识库" targe
="_blank">MySQL-connector-java-5.0.8-bin.jar
E:/works/encoder/lib/persistence.j
r
E:/works/encoder/lib/proxool-0.9.1.jar
E:/works/encoder/lib/proxool-cglib.jar
E:/works/encoder/lib/slf4j-api-1.5.6.jar
E:/works/encoder/lib/slf4j-log4j12-1.5.6.jar
E:/works/encoder/lib/<a href="http://lib.csdn.net/base/17" class="replace_word" title="Jav
 EE知识库" target="_blank">spring.jar
E:/works/encoder/lib/springside3-core-3.1.4.
ar
E:/works/encoder/lib/xerces-2.6.2.jar
E:/works/encoder/lib/xml-apis.jar</p>
<p>classloader 加载类用的是全盘负责委托机制。所谓全盘负责，即是当一个classloader加载一个C
ass的时候，这个Class所依赖的和引用的所有 Class也由这个classloader负责载入，除非是显式的使
另外一个classloader载入；委托机制则是先让parent（父）类加载器 (而不是super，它与parent cla
sloader类不是继承关系)寻找，只有在parent找不到的时候才从自己的类路径中去寻找。此外类加载
采用了cache机制，也就是如果 cache中保存了这个Class就直接返回它，如果没有才从文件中读取和
换成Class，并存入cache，这就是为什么我们修改了Class但是必须重新启动JVM才能生效的原因。<
r />类加载器的顺序是：
先是bootstrap classloader，然后是extension classloader，最后

是system classloader。大家会发现加载的Class越是重要的越在靠前面。这样做的原因是出于安全性考虑，试想如果system classloader“亲自”加载了一个具有破坏性的“java.lang System”类的后果吧。这种委托机制保证了用户即使具有一个这样的类，也把它加入到了类径中，但是它永远不会被载入，因为这个类总是由bootstrap classloader来加载的。

System.out.println("bootstrap classloader - 引导（也称为原始）类加载器");
 &bsp;URL[] urls=sun.misc.Launcher.getBootstrapClassPath().getURLs();
  &nsp; for (int i = 0; i < urls.length; i++) {
     &nsp; System.out.println(urls[i].toExternalForm());
     }
&nsp; System.out.println("extension classloader - 扩展类加载器");
  System.out.println(System.getProperty("java.ext.dirs"));
  System.out.println("system classloader - 系统（也称为应用）类加载器");
  System.out.println(System.getProperty("java.class.path").replace(";", "/n"));

==========
JAVA中的每一个类都是通过类加载器加载到内存中的。对于类加载器的工作流程如下表示：
1.searchfile()
找到我所要加载的类文件。（抛除JAR包的概念，现在只是要加载一个.class文件）
2.loadDataClass()
读取这个类文件的字节码。
3.defineClass()
加载类文件。（加载的过程其实很复杂，我们现在先不研究它。）
从这个过程中我们能很清楚发现，自定义的类加载能够很轻松的控制每个类文件的加载过程。这样在第二步（loadDataClass）第三步（difineClass）之间，我们将会有自己的空间灵活的控制这个过程。
我们加密解密的技就应用到这里。
了解ClassLoader
1， 什么是 ClassLoader?
Java 程序并不是一个可执行件，是需要的时候，才把装载到 JVM中。ClassLoader 做的工作就是 JVM 中将类装入内存。 而且，Java ClassLoader 就是用 Java 语言编写的。这意味着您可以创建自己的 ClassLoader
ClassLoader 的基本目标是对类的请求提供服务。当 JVM 需要使用类时，它根据名称向 ClassLoader 求这个类，然后 ClassLoader 试图返回一个表示这个类的 Class 对象。 通过覆盖对应于这个过程不阶段的方法，可以创建定制的 ClassLoader。
2， 一些重要的方法
A) 方法 loadClass&bsp;
ClassLoader.loadClass() 是 ClassLoader 的入口点。该方法的定义如下：
lass loadClass(String name, boolean resolve ;
name JVM 需要的类的名称，如 Foo 或 java.ang.Object。
resolve 参数告诉方法是否需要解析类。在准备执行类之前，应考虑类解析。并总是需要解析。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。
B) 方法 defineClass
defineClass 方法是 ClassLoader 的主要诀窍。该方法接受原始字节组成的数组并把它转换成 Class 对象。原始数组包含如从文件系统或网络装入的数据。defin Class 管理 JVM 的许多复杂、神秘和倚赖于实现的方面 -- 它把字节码分析成运行时数结构、校验有效性等等。不必担心，您无需亲自编写它。事实上，即使您想要这么做也不能覆它，因为它已被标记成final的。
C) 方法 findSystemClass
findSystemClass 方法从本地文件系统装入文件。它在地文件系统中寻找类文件，如果存在，就使用 defineClass 将原始字节转换成 Class 对象，以将该文转换成类。当运行 Java 应用程序时，这是 JVM 正常装入类的缺省机制。（Java 2 中 ClassLoader 变动提供了关于 Java 版本 1.2 这个过程变动的详细信息。） 对于定制的 ClassLoader，只有在尝试它方法装入类之后，再使用 findSystemClass。原因很简单：ClassLoader 是负责执行装入类的特殊骤，不是负责所有类。例如，即使 ClassLoader 从远程的 Web 站点装入了某些类，仍然需要在本地器上装入大量的基本 Java 库。而这些类不是我们所关心的，所以要 JVM 以缺省方式装入它们：从本文件系统。这就是 findSystemClass 的用途。
D) 方法 resolveClass
正如前面所提到的，可以不完全地（不带解析）装入类，也以完全地（带解析）装入类。当编写我们自己的 loadClass 时，可以调用 resolveClass，这取决于 loClass 的 resolve 参数的值。

E) 方法 findLoadedClass
findLoadedClass 充当一个缓存：当请求 loadClss 装入类时，它调用该方法来查看 ClassLoader 是否已装入这个类，这样可以避免重新装入已存在所造成的麻烦。应首先调用该方法。
3， 怎么组装这些方法
1） 调用 findLoadedClass 来查看是否存在已装入的类。<r />2） 如果没有，那么采用那种特殊的神奇方式来获取原始字节。
3） 如果已有原始字节，

用 defineClass 将它们转换成 Class 对象。
4） 如果没有原始字节，然后调用 findSystemClas 查看是否从本地文件系统获取类。
5） 如果 resolve 参数是 true，那么调用 resolveClass 解析 Class 对象。
6） 如果还没有类，返回 ClassNotFoundException。
4，Java 2 中 ClassLoader 的变动
1）loadClass 的缺省实现
定制编写的 loadClass 方法一般尝试几种方式来装入所请求的类，如果您编写许多类，会发现一次次地在相同的、很杂的方法上编写变量。 在 Java 1.2 中 loadClass 的实现嵌入了大多数查找类的一般方法，并使您通覆盖 findClass 方法来定制它，在适当的时候 findClass 会调用 loadClass。 这种方式的好处是您可不一定要覆盖 loadClass；只要覆盖 findClass 就行了，这减少了工作量。
2）新方法：findClass
loadClass 的缺省实现调用这个新方法。findClass 的用途包含您的 ClassLoader 的所有特殊代码，而无需要复制其它代码（例如，当专门的方法失败时，调用系统 Classoader）。
3） 新方法：getSystemClassLoader
如果覆盖 findClass 或 loadClass，getSystmClassLoader 使您能以实际 ClassLoader 对象来访问系统 ClassLoader（而不是固定的从 findSystmClass 调用它）。
4） 新方法：getParent
为了将类请求委托给父代 ClassLoader，这个新方法允许lassLoader 获取它的父代 ClassLoader。当使用特殊方法，定制的 ClassLoader 不能找到类时，可使用这种方法。
父代 ClassLoader 被定义成创建该 ClassLoader 所包含代码的对象的 ClassLoder。