JavaClassLoader

http://www.xthinking.net/Wiki.jsp?page=JavaClassLoader&version=1

ClassLoader

ClassLoader 主要对类的请求提供服务，当 JVM 需要某类时，它根据名称向 ClassLoader 要求这个类，然后由 ClassLoader 返回这个类的 class 对象。

几个相关概念

1. ClassLoader 负责载入系统的所有 Resources（Class，文件，来自网络的字节流等），通过 ClassLoader 从而将资源载入 JVM
2. 每个 class 都有一个 reference，指向自己的 ClassLoader: Class.getClassLoader()
3. array 的 ClassLoader 就是其元素的 ClassLoader，若是基本数据类型，则这个 array 没有 ClassLoader

主要方法和工作过程

Java1.1 及从前版本中，ClassLoader 主要方法：

1. Class loadClass(String name, boolean resolve);
   ClassLoader.loadClass() 是 ClassLoader 的入口点
1. defineClass 方法是 ClassLoader 的主要诀窍。该方法接受由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。
2. findSystemClass 方法从本地文件系统装入文件。它在本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，以将该文件转换成类。当运行 Java 应用程序时，这是 JVM 正常装入类的缺省机制。
3. resolveClass 可以不完全地（不带解析）装入类，也可以完全地（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 loadClass 的 resolve 参数的值
4. findLoadedClass 充当一个缓存：当请求 loadClass 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类所造成的麻烦。应首先调用该方法

一般 load 方法过程如下：

- 调用 findLoadedClass 来查看是否存在已装入的类。
- 如果没有，那么采用某种特殊的神奇方式来获取原始字节。（通过 IO 从文件系统，来自网络的字节流等）
- 如果已有原始字节，调用 defineClass 将它们转换成 Class 对象。
- 如果没有原始字节，然后调用 findSystemClass 查看是否从本地文件系统获取类。
- 如果 resolve 参数是 true，那么调用 resolveClass 解析 Class 对象。
- 如果还没有类，返回 ClassNotFoundException。
- 否则，将类返回给调用程序。

委托模型

自从JDK1.2 以后，ClassLoader做了改进，使用了**委托模型**，所有系统中的ClassLoader组成一棵树，ClassLoader在载入类库时先让Parent寻找，Parent找不到才自己找。

JVM 在运行时会产生三个 ClassLoader，Bootstrap ClassLoader、Extension ClassLoader 和 App ClassLoader。其中，Bootstrap ClassLoader 是用 C++编写的，在 Java 中看不到它，是 null。它用来加载核心类库，就是在 lib 下的类库，Extension ClassLoader 加载 lib/ext 下的类库，App ClassLoader 加载 Classpath 里的类库，三者的关系为:App ClassLoader 的 Parent 是 Extension ClassLoader，而 Extension ClassLoader 的 Parent 为 Bootstrap ClassLoader。加载一个类时，首先 BootStrap 进行寻找，找不到再由 Extension ClassLoader 寻找，最后才是 App ClassLoader。

将 ClassLoader 设计成委托模型的一个重要原因是出于安全考虑，比如在 Applet 中，如果编写了一 个 java.lang.String 类并具有破坏性。假如不采用这种委托机制，就会将这个具有破坏性的 String 加载到了用户机器上，导致破坏用户安全。但采用这种委托机制则不会出现这种情况。因为要加载 java.lang.String 类时，系统最终会由 Bootstrap 进行加载，这个具有破坏性的 String 永远没有机会加载。

委托模型还带来了一些问题，在某些情况下会产生混淆，如下是 Tomcat 的 ClassLoader 结构图:

```
                Bootstrap
                    |
                 System
                    |
                 Common
                  /   \
            Catalina Shared
                    /   \
                 Webapp1 Webapp2 ...
```

由 Common 类装入器装入的类决不能（根据名称）直接访问由 Web 应用程序装入的类。使这些类联系在一起的唯一方法是通过使用这两个类集都可见的接口。在这个例子中，就是包含由 Java servlet 实现的 javax.servlet.Servlet。

如果在 lib 或者 lib/ext 等类库有与应用中同样的类，那么应用中的类将无法被载入。通常在 jdk 新版本出现有类库移动时会出现问题，例如最初我们使用自己的 xml 解析器，而在 jdk1.4 中 xml 解析器变成标准类库，load 的优先级也高于我们自己的 xml 解析器，我们自己的 xml 解析器永远无法找到，将可能导致我们的应用无法运行。

相同的类，不同的 ClassLoader，将导致 ClassCastException 异常。

线程中的 ClassLoader

每个运行中的线程都有一个成员 contextClassLoader，用来在运行时动态地载入其它类，可以使用方法

```
Thread.currentThread().setContextClassLoader(...);
```

更改当前线程的 contextClassLoader，来改变其载入类的行为，也可以通过方法

```
Thread.currentThread(). getContextClassLoader()
```

来获得当前线程的 ClassLoader。

实际上，在 Java 应用中所有程序都运行在线程里，如果在程序中没有手工设置过 ClassLoader，对于一般的 java 类如下两种方法获得的 ClassLoader 通常都是同一个

1. this.getClass.getClassLoader()；
2. Thread.currentThread().getContextClassLoader()；

方法一得到的Classloader是静态的，表明类的载入者是谁；方法二得到的Classloader是动态的，谁执行（某个线程），就是那个执行者的Classloader。对于单例模式的类，静态类等，载入一次后，这个实例会被很多程序（线程）调用，对于这些类，载入的Classloader和执行线程的Classloader通常都不同。

Web 应用中的 ClassLoader

回到上面的例子，在 Tomcat 里，WebApp 的 ClassLoader 的工作原理有点不同，它先试图自己载入类（在 ContextPath/WEB-INF/...中载入类），如果无法载入，再请求父 ClassLoader 完成。

由此可得：

- 对于 WEB APP 线程，它的 contextClassLoader 是 WebAppClassLoader
- 对于 Tomcat Server 线程，它的 contextClassLoader 是 CatalinaClassLoader

获得 ClassLoader 的几种方法

可以通过如下 3 种方法得到 ClassLoader

1. this.getClass.getClassLoader(); //使用当前类的 ClassLoader
2. Thread.currentThread().getContextClassLoader(); // 使用当前线程的 ClassLoader
3. ClassLoader.getSystemClassLoader(); // 使用系统 ClassLoader，即系统的入口点所使用的 ClassLoader。（注意，system ClassLoader 与根 ClassLoader 并不一样。JVM 下 system ClassLoader 通常为 App ClassLoader）

几种扩展应用

用户定制自己的 ClassLoader 可以实现以下的一些应用

- 安全性。类进入 JVM 之前先经过 ClassLoader，所以可以在这边检查是否有正确的数字签名等
- 加密。java 字节码很容易被反编译，通过定制 ClassLoader 使得字节码先加密防止别人下载后反编译，这里的 ClassLoader 相当于一个动态的解码器
- 归档。可能为了节省网络资源，对自己的代码做一些特殊的归档，然后用定制的 ClassLoader 来解档
- 自展开程序。把 java 应用程序编译成单个可执行类文件，这个文件包含压缩的和加密的类文件数据，同时有一个固定的 ClassLoader，当程序运行时它在内存中完全自行解开，无需先安装
- 动态生成。可以生成应用其他还未生成类的类，实时创建整个类并可在任何时刻引入 JVM

资源载入

所有资源都通过 ClassLoader 载入到 JVM 里，那么在载入资源时当然可以使用 ClassLoader，只是对于不同的资源还可以使用一些别的方式载入，例如对于类可以直接 new，对于文件可以直接做 IO 等。

载入类的几种方法

假设有类 A 和类 B，A 在方法 amethod 里需要实例化 B，可能的方法有 3 种。对于载入类的情况，用户需要知道 B 类的完整名字（包括包名，例如"com.rain.B"）。

1. 使用 Class 静态方法 Class.forName

```
Class cls = Class.forName("com.rain.B");
B b = (B)cls.newInstance();
```

2. 使用 ClassLoader

```
/* Step 1. Get ClassLoader */
ClassLoader cl; // 如何获得ClassLoader参考 1.6
/* Step 2. Load the class */
Class cls = cl.loadClass("com.rain.B");
/* Step 3. new instance */
B b = (B)cls.newInstance(); // 有B的类得到一个B的实例
```

3. 直接 new

```
B b = new B();
```

文件载入（例如配置文件等）

假设在 com.rain.A 类里想读取文件夹 /com/rain/config 里的文件 sys.properties，读取文件可以通过绝对路径或相对路径，绝对路径很简单，在 Windows 下以盘号开始，在 Unix 下以"/"开始。

对于相对路径，其相对值是相对于 ClassLoader 的，因为 ClassLoader 是一棵树，所以这个相对路径和 ClassLoader 树上的任何一个 ClassLoader 相对比较后可以找到文件，那么文件就可以找到，当然，读取文件也使用委托模型。

1. 直接 IO

```
/**
 * 假设当前位置是 "C:/test"，通过执行如下命令来运行 A "java com.rain.A"
 * 1. 在程序里可以使用绝对路径，Windows 下的绝对路径以盘号开始，Unix 下以"/"开始
 * 2. 也可以使用相对路径，相对路径前面没有"/"
 * 因为我们在 "C:/test" 目录下执行程序，程序入口点是"C:/test"，相对路径就
 * 是 "com/rain/config/sys.properties"
 * （例子中，当前程序的 ClassLoader 是 AppClassLoader，SystemClassLoader = 当前的
 * 程序的 ClassLoader，入口点是"C:/test"）
 * 对于 ClassLoader 树，如果文件在 jdk lib 下，如果文件在 jdk lib/ext 下，如果文件在环
 * 境变量里，都可以通过相对路径"sys.properties"找到，lib 下的文件最先被找到
 */
File f = new File("C:/test/com/rain/config/sys.properties"); // 使用绝对路径
//File f = new File("com/rain/config/sys.properties"); // 使用相对路径
InputStream is = new FileInputStream(f);
```

如果是配置文件，可以通过 java.util.Properties.load(is)将内容读到 Properties 里，Properties 默认认为 is 的编码是 ISO-8859-1，如果配置文件是非英文的，可能出现乱码问题。

2. 使用 ClassLoader

```
/**
 * 因为有 3 种方法得到 ClassLoader，对应有如下 3 种方法读取文件
 * 使用的路径是相对于这个 ClassLoader 的那个点的相对路径，此处只能使用相对路径
```

```
     */
    InputStream is = null;
    is = this.getClass().getClassLoader().getResourceAsStream(
          "com/rain/config/sys.properties");
    //is = Thread.currentThread().getContextClassLoader().getResourceAsStream(
          "com/rain/config/sys.properties");
    //is =
ClassLoader.getSystemResourceAsStream("com/rain/config/sys.properties");
```

如果是配置文件，可以通过 java.util.Properties.load(is)将内容读到 Properties 里，这里要注意编码问题。

3. 使用 ResourceBundle

```
    ResourceBundle bundle = ResourceBundle.getBoundle("com.rain.config.sys");
```

这种用法通常用来载入用户的配置文件，关于 ResourceBunlde 更详细的用法请参考其他文档。

总结：有如下 3 种途径来载入文件

1. 绝对路径 ---> IO
2. 相对路径 ---> IO
   ---> ClassLoader
3. 资源文件 ---> ResourceBundle

如何在 web 应用里载入资源

在 web 应用里当然也可以使用 ClassLoader 来载入资源，但更常用的情况是使用 ServletContext，如下是 web 目录结构

```
       ContextRoot
          |- JSP、HTML、Image 等各种文件
           |- [WEB-INF]
                |- web.xml
                |- [lib] Web 用到的 JAR 文件
                  |- [classes] 类文件
```

用户程序通常在 classes 目录下，如果想读取 classes 目录里的文件，可以使用 ClassLoader，如果想读取其他的文件，一般使用 ServletContext.getResource()

如果使用 ServletContext.getResource(path)方法，路径必须以"/"开始，路径被解释成相对于 ContextRoot 的路 径，此处载入文件的方法和 ClassLoader 不同，举例 "/WEB-INF/web.xml", "/download/WebExAgent.rar" 。