[重温java之classloader体系结构（含hotswap）](http://www.iteye.com/topic/136427)

**jvm classLoader architecture** :

a, Bootstrap ClassLoader/启动类加载器  
主要负责jdk\_home/lib目录下的核心 api 或 -Xbootclasspath 选项指定的jar包装入工作.

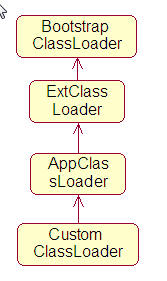
b, Extension ClassLoader/扩展类加载器  
主要负责jdk\_home/lib/ext目录下的jar包或 -Djava.ext.dirs 指定目录下的jar包装入工作

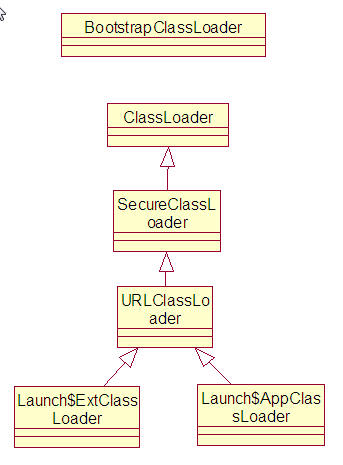
c, System ClassLoader/系统类加载器  
主要负责java -classpath/-Djava.class.path所指的目录下的类与jar包装入工作.

b, User Custom ClassLoader/用户自定义类加载器(java.lang.ClassLoader的子类)  
在程序运行期间, 通过java.lang.ClassLoader的子类动态加载class文件, 体现java动态实时类装入特性.

**类加载器的特性：**

1， 每个ClassLoader都维护了一份自己的名称空间， 同一个名称空间里不能出现两个同名的类。  
2， 为了实现java安全沙箱模型顶层的类加载器安全机制, java默认采用了 ” 双亲委派的加载链 ” 结构.

如下图:  


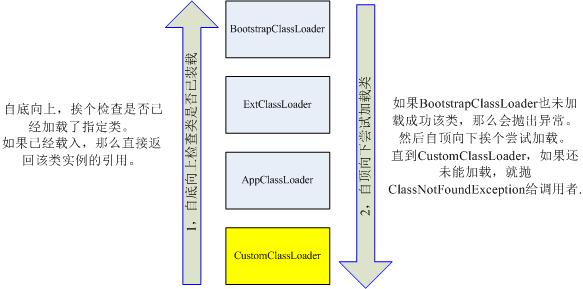
Class Diagram：  


类图中， BootstrapClassLoader是一个单独的java类， 其实在这里， 不应该叫他是一个java类。  
因为， 它已经完全不用java实现了。

它是在jvm启动时， 就被构造起来的， 负责java平台核心库。（具体上面已经有介绍）

启动类加载实现 (其实我们不用关心这块， 但是有兴趣的， 可以研究一下 ):  
[bootstrap classLoader 类加载原理探索](http://www.iteye.com/blog/136885)

**自定义类加载器加载一个类的步骤 :**



ClassLoader 类加载逻辑分析， 以下逻辑是除 BootstrapClassLoader 外的类加载器加载流程:

**Java代码 [收藏代码](javascript:void())**

1. // 检查类是否已被装载过
2. Class c = findLoadedClass(name);
3. **if** (c == **null** ) {
4. // 指定类未被装载过
5. **try** {
6. **if** (parent != **null** ) {
7. // 如果父类加载器不为空， 则委派给父类加载
8. c = parent.loadClass(name, **false** );
9. } **else** {
10. // 如果父类加载器为空， 则委派给启动类加载加载
11. c = findBootstrapClass0(name);
12. }
13. } **catch** (ClassNotFoundException e) {
14. // 启动类加载器或父类加载器抛出异常后， 当前类加载器将其
15. // 捕获， 并通过findClass方法， 由自身加载
16. c = findClass(name);
17. }
18. }

**用Class.forName加载类**  
Class.forName使用的是被调用者的类加载器来加载类的.  
这种特性, 证明了java类加载器中的名称空间是唯一的, 不会相互干扰.

即在一般情况下, 保证同一个类中所关联的其他类都是由当前类的类加载器所加载的.

**Java代码 [收藏代码](javascript:void())**

1. **public** **static** Class forName(String className)
2. **throws** ClassNotFoundException {
3. **return** forName0(className, **true** , ClassLoader.getCallerClassLoader());
4. }
6. /\*\* Called after security checks have been made. \*/
7. **private** **static** **native** Class forName0(String name, **boolean** initialize,
8. ClassLoader loader)
9. **throws** ClassNotFoundException;

上图中 ClassLoader.getCallerClassLoader 就是得到调用当前forName方法的类的类加载器

**线程上下文类加载器**  
java默认的线程上下文类加载器是 系统类加载器(AppClassLoader).

**Java代码 [收藏代码](javascript:void())**

1. // Now create the class loader to use to launch the application
2. **try** {
3. loader = AppClassLoader.getAppClassLoader(extcl);
4. } **catch** (IOException e) {
5. **throw** **new** InternalError(
6. "Could not create application class loader" );
7. }
9. // Also set the context class loader for the primordial thread.
10. Thread.currentThread().setContextClassLoader(loader);

以上代码摘自sun.misc.Launch的无参构造函数Launch()。

使用线程上下文类加载器, 可以在执行线程中, 抛弃双亲委派加载链模式, 使用线程上下文里的类加载器加载类.  
典型的例子有, 通过线程上下文来加载第三方库jndi实现, 而不依赖于双亲委派.

大部分java app服务器(jboss, tomcat..)也是采用contextClassLoader来处理web服务。  
还有一些采用 hotswap 特性的框架, 也使用了线程上下文类加载器, 比如 seasar (full stack framework in japenese).

线程上下文从根本解决了一般应用不能违背双亲委派模式的问题.

使java类加载体系显得更灵活.

随着多核时代的来临, 相信多线程开发将会越来越多地进入程序员的实际编码过程中. 因此,  
在编写基础设施时， 通过使用线程上下文来加载类, 应该是一个很好的选择.

当然, 好东西都有利弊. 使用线程上下文加载类, 也要注意, 保证多根需要通信的线程间的类加载器应该是同一个,  
防止因为不同的类加载器, 导致类型转换异常(ClassCastException).

**自定义的类加载器实现**  
defineClass(String name, byte[] b, int off, int len,ProtectionDomain protectionDomain)  
是java.lang.Classloader提供给开发人员, 用来自定义加载class的接口.

使用该接口, 可以动态的加载class文件.

例如,  
在jdk中, URLClassLoader是配合findClass方法来使用defineClass, 可以从网络或硬盘上加载class.

而使用类加载接口, 并加上自己的实现逻辑, 还可以定制出更多的高级特性.

比如,

一个简单的hot swap 类加载器实现:

**Java代码 [收藏代码](javascript:void())**

1. **import** java.io.File;
2. **import** java.io.FileInputStream;
3. **import** java.lang.reflect.Method;
4. **import** java.net.URL;
5. **import** java.net.URLClassLoader;
7. /\*\*
8. \* 可以重新载入同名类的类加载器实现
9. \*
11. \* 放弃了双亲委派的加载链模式.
12. \* 需要外部维护重载后的类的成员变量状态.
13. \*
14. \* @author ken.wu
15. \* @mail ken.wug@gmail.com
16. \* 2007-9-28 下午01:37:43
17. \*/
18. **public** **class** HotSwapClassLoader **extends** URLClassLoader {
20. **public** HotSwapClassLoader(URL[] urls) {
21. **super** (urls);
22. }
24. **public** HotSwapClassLoader(URL[] urls, ClassLoader parent) {
25. **super** (urls, parent);
26. }
28. **public** Class load(String name)
29. **throws** ClassNotFoundException {
30. **return** load(name, **false** );
31. }
33. **public** Class load(String name, **boolean** resolve)
34. **throws** ClassNotFoundException {
35. **if** ( **null** != **super** .findLoadedClass(name))
36. **return** reload(name, resolve);
38. Class clazz = **super** .findClass(name);
40. **if** (resolve)
41. **super** .resolveClass(clazz);
43. **return** clazz;
44. }
46. **public** Class reload(String name, **boolean** resolve)
47. **throws** ClassNotFoundException {
48. **return** **new** HotSwapClassLoader( **super** .getURLs(), **super** .getParent()).load(
49. name, resolve);
50. }
51. }
53. **public** **class** A {
54. **private** B b;
56. **public** **void** setB(B b) {
57. **this** .b = b;
58. }
60. **public** B getB() {
61. **return** b;
62. }
63. }
65. **public** **class** B {}

这个类的作用是可以重新载入同名的类， 但是， 为了实现hotswap, 老的对象状态  
需要通过其他方式拷贝到重载过的类生成的全新实例中来。(A类中的b实例)

而新实例所依赖的B类如果与老对象不是同一个类加载器加载的， 将会抛出类型转换异常(ClassCastException).

为了解决这种问题， HotSwapClassLoader自定义了load方法. 即当前类是由自身classLoader加载的， 而内部依赖的类

还是老对象的classLoader加载的.

**Java代码 [收藏代码](javascript:void())**

1. **public** **class** TestHotSwap {
2. **public** **static** **void** main(String args[]) {
3. A a = **new** A();
4. B b = **new** B();
5. a.setB(b);
7. System.out.printf("A classLoader is %s n" , a.getClass().getClassLoader());
8. System.out.printf("B classLoader is %s n" , b.getClass().getClassLoader());
9. System.out.printf("A.b classLoader is %s n" , a.getB().getClass().getClassLoader());
11. HotSwapClassLoader c1 = **new** HotSwapClassLoader( **new** URL[]{ **new** URL( "file:\e:\test\")} , a.getClass().getClassLoader());
12. Class clazz = c1.load(" test.hotswap.A ");
13. Object aInstance = clazz.newInstance();
15. Method method1 = clazz.getMethod(" setB ", B.**class**);
16. method1.invoke(aInstance, b);
18. Method method2 = clazz.getMethod(" getB ", **null**);
19. Object bInstance = method2.invoke(aInstance, **null**);
21. System.out.printf(" reloaded A.b classLoader is %s n", bInstance.getClass().getClassLoader());
22. }
23. }

输出

A classLoader is sun.misc.Launcher$AppClassLoader@19821f  
B classLoader is sun.misc.Launcher$AppClassLoader@19821f  
A.b classLoader is sun.misc.Launcher$AppClassLoader@19821f  
reloaded A.b classLoader is sun.misc.Launcher$AppClassLoader@19821f