

Dynamic Class Loading & Swapping

2006-09-27 18:44:43

Das Austauschen von Klassen während der Laufzeit scheint nicht immer ein triviales Problem zu sein. Die meisten Anwendungen, die Plugins benutzen, wie z. B. Firefox, Eclipse oder jEdit, benötigen einen Neustart nach dem Update von Plugins. Dabei geht das in Java recht einfach.

Auf die Idee, wie man es machen könnte, brachte mich [Andrej](#): Er schlug vor, einen eigenen Classloader und Wrapper für die Plugins zu schreiben.

Doch das ganze ist wesentlich unkomplizierter. Eine kurze Recherche brachte mich auf den Artikel [Dynamically Reloading a Modified Class](#); mit der Lösung, auch ohne einen eigenen Classloader zu schreiben: URLClassLoader.

Grundsätzlich kann man mit einem Classloader jegliche Klasse aus dem Classpath laden, indem man das bekannte Class.forName benutzt.

Doch der System Classloader benutzt intern einen Cache, wonach jede einmal geladene Klasse vermerkt und bei erneutem Aufruf von Class.forName aus dem Cache, also nicht von der Platte, geladen wird.

Der URLClassLoader arbeitet da anders. Er holt sich eine Klasse explizit von der angegebenen URL.

Um nun Klassen im Speicher austauschbar zu halten, muss man lediglich dafür sorgen, dass keine Referenzen mehr auf diese Klasse existieren, um den Tausch on the fly zu machen. Dies kann man durch ein Proxy bzw. Wrapperobjekt machen.

Angenommen wir definieren ein PluginInterface, in etwa wie:

```
public interface PluginInterface { public String getName(); public long getVersion(); public void process(PluginActionEvent event) throws PluginProcessingException; public void activate(Object parameter) throws PluginActivationException; public void deactivate(Object parameter) throws PluginDeactivationException; }
```

 Ein einfaches Plugin könnte dann etwa so aussehen:

```
public class DemoPlugin1 implements PluginInterface { public String getName() { return &quot;demo1&quot;; } public long getVersion() { return 0; } public void process(PluginActionEvent event) throws PluginProcessingException { System.out.println(&quot;process: &quot;+ getName()+&quot; version=&quot;+getVersion()); } public void activate(Object parameter) throws PluginActivationException { System.out.println(&quot;activate: &quot;+getName()+ &quot; version=&quot;+getVersion()); } public void deactivate(Object parameter) throws PluginDeactivationException { System.out.println(&quot;deactivate: &quot;+getName()+ &quot; version=&quot;+getVersion()); } }
```

 Um nun direkte Verweise auf dieses Plugin im Speicher zu vermeiden, benutzt man nun den Wrapper:

```
public class PluginWrapper implements PluginInterface{ private PluginInterface plugin; private Class loadedClass; private String loadedClassString; public PluginWrapper(){ } public String getName() { if (plugin==null) return null; return plugin.getName(); } public long getVersion() { if (plugin==null) return -1; return plugin.getVersion(); } public void loadPlugin(URL pluginLocation, String loadedClassString) throws ClassNotFoundException{ if (this.loadedClassString!=null){ unloadPlugin(); } this.loadedClassString = loadedClassString; URLClassLoader cl = new URLClassLoader(new URL[]{pluginLocation}); }
```

```
loadedClass = cl.loadClass(loadedClassString); } public void unloadPlugin(){ plugin = null; loadedClass = null; } public void instantiatePlugin() throws ClassNotFoundException, InstantiationException, IllegalAccessException{ if (plugin!=null) throw new IllegalStateException(&quot;Plugin already instantiated&quot;); if (loadedClass!=null) throw new IllegalStateException(&quot;Plugin must first be loaded&quot;); PluginInterface pi = (PluginInterface) loadedClass.newInstance(); plugin = pi; } public void process(PluginActionEvent event) throws PluginProcessingException{ if (plugin!=null) plugin.process(event); } public void activate(Object parameter) throws PluginActivationException{ if (plugin==null){ try { instantiatePlugin(); } catch (Exception e) { throw new PluginActivationException(e); } } plugin.activate(parameter); } public void deactivate(Object parameter) throws PluginDeactivationException{ if (plugin!=null){ plugin.deactivate(parameter); } } }
```

Die Funktionsweise ist recht einfach:

Mit `plugin.loadPlugin` übergibt man der Wrapperinstanz eine URL, an der die zu ladende Klasse zu finden ist (eine Jar Datei, eine http/ftp URL oder ein Verzeichnis) und mit einem String den voll qualifizierten Klassennamen.

Anschließend wird mit `plugin.instantiate` eine Instanz des Plugins erzeugt.

Die Methode `unload` sorgt dafür, dass die Referenzen des Wrappers auf das Plugin selber auf null gesetzt werden und damit die Klasse später vom Garbagecollector entfernt werden kann.

Alle Plugin-Methoden werden an das Plugin delegiert. Somit benötigt man in der restlichen Anwendung keine direkte Referenz mehr auf das dynamisch geladene Plugin.

Der Wrapper hingegen wird ständig weiter benutzt und einmal pro Plugin instanziiert.

Ich habe mir einen einfachen Deskriptor gebaut, über den man Plugins definiert, in etwa wie:

```
-----PLUGIN DESCRIPTOR START-----  
class=xyz.DemoPlugin1  
version=0  
name=demo1  
descriptor-update-url=http://narcanti.keyboardsamurais.de/mp/demo1-descriptor.txt  
jar-update-url=http://narcanti.keyboardsamurais.de/mp/demo1.jar  
note=This Version of the plugin needs log4j v.1.2.14  
-----PLUGIN DESCRIPTOR END-----
```

Über diesen Deskriptor kann man nach Updates prüfen und bei Bedarf das entsprechende Jar herunterladen.

Es sei angemerkt, dass der Name eines Plugins eine Art eindeutiger Plugin-Identifizier ist, über den der entsprechende Wrapper identifiziert wird.

Im übrigen ist der Code nur eine zusammengekürzte Version, da ich das ganze etwas aufwändiger, komfortabler und konsistenter umgesetzt habe.

Weitere interessante Artikel zu diesem Thema:

- [“Determining from Where a Class Was Loaded”](#)
- [“Loading a Class That Is Not on the Classpath”](#)