A Look at the Java Class Loader

At 8:03 AM on Oct 15, 2005, Kumar Matcha

http://www.javalobby.org/java/forums/t18345.html

Class loading is one of the most powerful mechanisms provided by the Java Language Specification. All Java programmers should know how the class loading mechanism works and what can be done to suit their needs. By understanding the class loading mechanism you can save time that would otherwise be spent on debugging ***ClassNotFoundException***, ***ClassCastException***, etc.

Class Loaders

In a Java Virtual Machine (JVM), each and every class is loaded by some instance of a java.lang.ClassLoader. The ClassLoader class is located in the java.lang package and you can extend it to add your own functionality to class loading.

When a new JVM is started by "java HelloWorld", the "**bootstrap class loader**" is responsible for loading key java classes like **java.lang.Object** and other runtime code into memory. The runtime classes are packaged inside **jre/lib/rt.jar** file. We cannot find the details of the **bootstrap class loader** in the java language specification, since this is a native implementation. For this reason the behavior of the bootstrap class loader will differ across JVM's.

Maze Behind Class Loaders

All class loaders are of the type **java.lang.ClassLoader**. Other than the bootstrap class loader all class loaders have a **parent class loader**. These two statements are different and are important for the correct working of any class loaders written by a developer. The most important aspect is to correctly set the parent class loader. The parent class loader for any class loader is the class loader instance that loaded that class loader.

We have two ways to set the parent class loader:

```
public class CustomClassLoader extends ClassLoader{
  public CustomClassLoader(){
    super(CustomClassLoader.class.getClassLoader());
  }
}
```

Or

```
public class CustomClassLoader extends ClassLoader {
  public CustomClassLoader(){
    super(getClass().getClassLoader());
  }
}
```

The first constructor is the preferred one, because calling the method **getClass()** from within a constructor should be discouraged, since the object initialization will be complete only at the exit of the constructor code. Thus if the parent class loader is correctly set, whenever a class is requested out of a ClassLoader instance using **loadClass(String name)** method, if it cannot find the class, it should ask the parent first. If the parent cannot find the class, the **findClass(String name)** method is invoked. The default implementation of

findClass(String name) will throw **ClassNotFoundException** and developers are expected to implement this method when they subclass **java.lang.ClassLoader** to make custom class loaders.

Inside the **findClass(String name)** method, the class loader needs to fetch the byte codes from some arbitrary source. The source may be a file system, a network URL, another application that can spit out byte codes on the fly, or any similar source that is capable of generating byte code compliant with the Java byte code specification. Once the byte code is retrieved, the method should call the **defineClass()** method, and the runtime is very particular about which instance of the ClassLoader is calling the method. Thus if two ClassLoader instances define byte codes from the same or different sources, the defined classes are different.

For example, let's say I've a main class called **MyProgram**. **MyProgram** is loaded by the application class loader, and it created instances of two class loaders **CustomClassLoader1** and **CustomClassLoader2** which are capable of finding the byte codes of another class **Student** from some source. This means the class definition of the **Student** class is not in the application class path or extension class path. In such a scenario the **MyProgram** class asks the custom class loaders to load the **Student** class, **Student** will be loaded and **Student.class** will be defined independently by both **CustomClassLoader1** and **CustomClassLoader2**. This has some serious implications in java. In case some static initialization code is put in the **Student** class, and if we want this code to be executed one and only once in a JVM, the code will be executed **twice** in the JVM with our setup, once each when the class is separately loaded by both CustomClassLoaders. If we have two instances of **Student** class loaded by these CustomClassLoaders say **student1** and **student2**, then **student1** and **student2** are not type-compatible. In other words,

```
    Student student3 = (Student) student2;
```

will throw **ClassCastException**, because the JVM sees these two as separate, distinct class types, since they are defined by different ClassLoader instances.

The Need For Your Own Class loader

For those who wish to control the JVM's class loading behavior, the developers need to write their own class loader. Let us say that we are running an application and we are making use of a class called **Student**. Assuming that the **Student** class is updated with a better version on the fly, i.e. when the application is running, and we need to make a call to the updated class. If you are wondering that the **bootstrap class loader** that has loaded the application will do this for you, then you are wrong. Java's class loading behavior is such that, once it has loaded the classes, it will not reload the new class. How to overcome this issue has been the question on every developers mind. The answer is simple. **Write your own class loader and then use your class loader to load the classes.** When a class has been modified on the run time, then you need to create a new instance of your class loader to load the class. Remember, that once you have created a new instance of your class loader, then **you should make sure that you no longer make reference to the old class loader**, because when two instances of the same object is loaded by different class loaders then they are treated as incompatible types.

Writing Your Own Class loader

The solution to control class loading is to implement custom class loaders. Any custom class loader should have **java.lang.ClassLoader** as its direct or distant super class. Moreover you need to set the parent class loader in the constructor. Then you have to override the **findClass()** method. Here is an implementation of a custom class loader.

```
    import java.io.DataInputStream;
    import java.io.File;
    import java.io.FileInputStream;
    import java.io.IOException;
    import java.util.Enumeration;
    import java.util.Hashtable;

    public class CustomClassLoader extends ClassLoader {

        public CustomClassLoader() {
            super(CustomClassLoader.class.getClassLoader());
        }

        public Class loadClass(String className) throws ClassNotFoundException {
             return findClass(className);
        }

        public Class findClass(String className){
            byte classByte[];
            Class result = null;
            result = (Class)classes.get(className);
            if(result != null){
                return result;
            }

            try{
                return findSystemClass(className);
            }catch(Exception e){
            }

            try{
               String classPath =
((String)ClassLoader.getSystemResource(className.replace('.', '/') +
".class").getFile()).substring(1);
               classByte = loadClassData(classPath);
                result = defineClass(className, classByte, 0, classByte.length,
null);
                classes.put(className, result);
                return result;
            }catch(Exception e){
                return null;
            }
        }

        private byte[] loadClassData(String className) throws IOException{
            File f = new File(className);
            int size = (int)f.length();
            byte buff[] = new byte[size];
            FileInputStream fis = new FileInputStream(f);
            DataInputStream dis = new DataInputStream(fis);
            dis.readFully(buff);
            dis.close();
            return buff;
        }

        private Hashtable classes = new Hashtable();

    }
```

Here is how to use the CustomClassLoader.

```
    public class CustomClassLoaderTest {

        public static void main(String [] args) throws Exception{
          CustomClassLoader test = new CustomClassLoader();
          test.loadClass(“com.test.HelloWorld”);
        }
    }
```

Summary

Many J2EE application servers have a “**hot deployment** ” capability, where we can reload an application with a new version of class definition, without bringing the server VM down. Such application servers make use of custom class loaders. Even if we don’t use an application server, we can create and use custom class loaders to fine-control class loading mechanisms in our Java applications. So have fun with custom loaders.