Techniques of Java Programming: Dynamic Class Loading

Manuel Oriol

May 15, 2006

1 Introduction

Dynamic class loading is one of the key features of the Java language. It concentrates around the notion of `ClassLoader`. The modularity of the system has been originally motivated by the need of having a unified mechanism for both loading mobile code (e.g. applets) and regular code.

2 Default Dynamic Class Loading

The default mechanism for loading classes is coded within the default class loader. This class loader can be accessed through the `static` method `getSystemClassLoader` from `ClassLoader`.

Table 1: Method from ClassLoader

java.lang.ClassLoader	
ClassLoader()	Creates a loader
ClassLoader(ClassLoader parent)	Creates a loder with given parent.
Class defineClass(String name, byte[] b, int off, int len)	Defines a class, given its bytecode definition.
Class <?> findClass(String Name)	Finds a class.
Class <?> findLoadedClass(String name)	Checks if a class has been loaded.
protected Class <?> findSystemClass(String name)	Loads a class using the system Class loader.
ClassLoader getParent()	Returns the parent loader.
Class loadClass(String name)	Loads a class and resolves it.
Class loadClass(String name, boolean resolve)	Loads a class and can resolve it depending on resolve.
void resolveClass(Class c)	Links a class

When willing to load a class within a `ClassLoader`, the default behavior of the Java Virtual Machine is to call the method `loadClass` descibed in Table 1.

By default a sequence of operations are performed (this is called the delegation model for loading classes).

`loadClass` is called:

1. `findLoadedClass(className)` checks if the class has already been loaded. If it is not the case...
2. `loadClass` is called on the parent ClassLoader (if none the system one is used), if it is not possible...
3. `findClass` is invoked and tries to load the class once the class has been loaded in a byte[] it needs to actually be loaded in the VM and linked into the system:
   (a) `defineClass` is invoked and the byte[] is transformed into a Class object.
4. `resolveClass` is called to actually link the class into the ClassLoader.

As we can see, the only real steps to load a class and use it are steps 3a and 4. It is however needed to implement `loadClass` in order to allow on-demand class loading to proceed as needed. Due to the `protected` keyword, these metods are only accessible from subclasses of `ClassLoader`.

3 Custom Class Loaders

In order to create a custom class loader, programmers can either redefine `loadClass` or `findClass`. In the APIs the preference is given to `findClass`, nevertheless it implies that the consistency of classes is ensured all over the application. In the case of class loaders containing several versions of the same class in the same JVM, it is not possible to rely on the delegation mechanism. Moreover in numerous other cases where the loading of classes on-demand is not the best option (e.g. when doing prefetching), it is also desirable to change the implementation of `loadClass` to load a group of classes rather than one class only.

Besides the custom loading, the use of class loaders is the only way to have code garbalge collected in a JVM. Once no thread is running through the classloader and no reference on any class or instance is still valid from outside the class loader, the code and instances can be garbage collected.

4 Example: `PrefetchingLoader`

As a simple example,it can be usefull to show a class loader that prefetches classes and loads them in the JVM when it accesses a directory. As a simple example of use, it would be very usful to prefetch all classes that a plugin uses when the plugin is actually loaded.

To achieve this goal, we override `loadClass` and use a method that finds all classes from a directory.

```
import javax.swing.*;
import java.io.*;
```

```
import java.util.*;

public class PrefetchingLoader extends ClassLoader{
   public class MyFileFilter extends javax.swing.filechooser.FileFilter{
      String name;
      public MyFileFilter(String name){
         this.name=name;
      }
      public boolean accept(File f){
         return (f.getName()).endsWith(name+".class");
      }
      public String getDescription(){
         return ".class chooser";
      }
   }

   public Class loadClass(String name)throws ClassNotFoundException{
      return loadClass(name,true);
   }
   public Class loadClass(String name,boolean resolve)
      throws ClassNotFoundException{

      Vector<Class> v;
      Class c;
      c=findLoadedClass(name);
      if (c!=null) return c;
      if (name.startsWith("java.") || name.startsWith("javax.")||
          name.startsWith("sun.")) {
         c=findSystemClass(name);
         resolveClass(c);
         return c;
      }else {
         v=findClasses(name);
         if (resolve)
         for (Class c0:v) {
            System.err.println("Resolving "+c0.getName());
            resolveClass(c0);
         }
         return v.firstElement();
      }
   }

   public Class readClass(File file,String name){
      Class c;
      try {
         FileReader fr=new FileReader(file);
         long l=file.length();
         char[] cBuf=new char[(int)l+1];
         fr.read(cBuf,0,(int)l);
         byte[] bBuf=(new String(cBuf)).getBytes();
         c=defineClass(name, bBuf,0,(int)l);
         return c;
      } catch (Throwable e){
         System.err.println(name+" not loaded");
```

```
      return null;
   }
}

public Vector<Class> findClasses(String name) throws ClassNotFoundException {
   File f=null;
   File dir=null;
   String s0=null;
   System.err.println("trying to find: "+name);
   Vector<Class> v = new Vector<Class>();
   String s=System.getProperty("java.class.path");
   System.err.println("classpath: "+s);
   int index0;
   index0=s.indexOf(System.getProperty("path.separator"));
   if (index0<-1)
      s0=s;
   else {
      s0=s.substring(0,index0);
      s=s.substring(index0+1);
   }

   while (s0!=null){
      f=new File(s0+System.getProperty("file.separator")+name+".class");
      if (f.exists()) {
      String dirString=null;
      if (name.indexOf('.')>-1){
         dirString=s0+System.getProperty("file.separator")+name;
         dirString=dirString.substring(0,dirString.lastIndexOf('.'));
      } else
         dirString=s0;

      v.add(readClass(f,name));
      dir=new File(dirString);
      for (File f0:dir.listFiles()){
         String path=f0.getPath();
         if (path.endsWith(".class")){
            String className=path.substring(dirString.length()+1);
            className=className.substring(0,className.length()-6);

            if (!className.equals(name)){
               System.err.println("reading "+className);
               Class c=readClass(f0,className);
               if (c!=null)
                  v.add(c);
            }
         }
      }
      s0=null;
      } else{
         try{
            index0=s.indexOf(System.getProperty("path.separator"));
            if (index0<-1) {
               s0=s;
               s=null;
```

```
            }
            else {
               s0=s.substring(0,index0);
               s=s.substring(index0+1);
            }
          } catch (Exception e){ s0 =null;}
        }
      }
      if (v.isEmpty())
         throw new ClassNotFoundException("The class "+name+" could
              not be found");
      return v;
   }

   public static void main(String[] args) throws Exception{
      ClassLoader cl = new PrefetchingLoader();
      Class t=cl.loadClass("T");
      t.newInstance();
   }
}
```

5 Exercise

1. Imagine several improvements to te `PrefetchingClassLoader`.
2. Code a `NetworkClassLoader` and its associated `NetworkClassFinde`.
3. Give some applications in which you would use class loaders and why.