JAVAWORLD

ode to your application

o changes at runtime

By Li Yang

JavaWorld | Jun 12, 2006 1:00 AM PT

JavaServer Pages (JSP) is a more flexible technology than servlets because it can respond to dynamic changes at runtime. Can you imagine a common Java class that has this dynamic capability too? It would be interesting if you could modify the implementation of a service without redeploying it and update your application on the fly.

The article explains how to write dynamic Java code. It discusses runtime source code compilation, class reloading, and the use of the Proxy design pattern to make modifications to a dynamic class transparent to its caller.

An example of dynamic Java code

Let's start with an example of dynamic Java code that illustrates what true dynamic code means and also provides some context for further discussions. Please find this example's complete source code in Resources.

The example is a simple Java application that depends on a service called Postman. The Postman service is described as a Java interface and contains only one method, `deliverMessage()`:

```
public interface Postman {
    void deliverMessage(String msg);
}
```

A simple implementation of this service prints messages to the console. The implementation class is the dynamic code. This class, `PostmanImpl`, is just a normal Java class, except it deploys with its source code instead of its compiled binary code:

```
ıts Postman {

    public PostmanImpl() {
        output = System.out;
    }

    public void deliverMessage(String msg) {
        output.println("[Postman] " + msg);
        output.flush();
    }
}
```

The application that uses the Postman service appears below. In the `main()` method, an infinite loop reads string messages from the command line and delivers them through the Postman service:

```
public class PostmanApp {

    public static void main(String[] args) throws Exception {
        BufferedReader sysin = new BufferedReader(new InputStreamReader(System.in));

        // Obtain a Postman instance
        Postman postman = getPostman();

        while (true) {
            System.out.print("Enter a message: ");
            String msg = sysin.readLine();
            postman.deliverMessage(msg);
        }
    }

    private static Postman getPostman() {
        // Omit for now, will come back later
    }
}
```

Execute the application, enter some messages, and you will see outputs in the console such as the following (you can download the example and run it yourself):

```
[DynaCode] Init class sample.PostmanImpl
Enter a message: hello world
[Postman] hello world
Enter a message: have a nice day!
[Postman] have a nice day!
Enter a message:
```

Everything is straightforward except for the first line, which indicates that the class `PostmanImpl` is compiled and loaded.

Now we are ready to see something dynamic. Without stopping the application, let's modify `PostmanImpl`'s source code. The new implementation delivers all the messages to a text file, instead of the console:

```
// MODIFIED VERSION
public class PostmanImpl implements Postman {

    private PrintStream output;

    // Start of modification
    public PostmanImpl() throws IOException {
        output = new PrintStream(new FileOutputStream("msg.txt"));
    }
    // End of modification

    public void deliverMessage(String msg) {
        output.println("[Postman] " + msg);

        output.flush();
    }
}
```

Shift back to the application and enter more messages. What will happen? Yes, the messages go to the text file now. Look at the console:

```
[DynaCode] Init class sample.PostmanImpl
rld
ice day!
!
Enter a message: I wanna go to the text file.
[DynaCode] Init class sample.PostmanImpl
Enter a message: me too!
Enter a message:
```

Notice `[DynaCode] Init class sample.PostmanImpl` appears again, indicating that the class `PostmanImpl` is recompiled and reloaded. If you check the text file msg.txt (under the working directory), you will see the following:

```
[Postman] I wanna go to the text file.
[Postman] me too!
```

Amazing, right? We are able to update the Postman service at runtime, and the change is completely transparent to the application. (Notice the application is using the same Postman instance to access both versions of the implementations.)

Four steps towards dynamic code

Let me reveal what's going on behind the scenes. Basically, there are four steps to make Java code dynamic:

- Deploy selected source code and monitor file changes
- Compile Java code at runtime
- Load/reload Java class at runtime
- Link the up-to-date class to its caller

Deploy selected source code and monitor file changes

To start writing some dynamic code, the first question we have to answer is, "Which part of the code should be dynamic—the whole application or just some of the classes?" Technically, there are few restrictions. You can load/reload any Java class at runtime. But in most cases, only part of the code needs this level of flexibility.

The Postman example demonstrates a typical pattern on selecting dynamic classes. No matter how a system is composed, in the end, there will be building blocks such as services, subsystems, and components. These building blocks are relatively independent, and they expose functionalities to each other via predefined interfaces. Behind an interface, it is the implementation that is free to change as long as it conforms to the contract defined by the interface. This is exactly the quality we need for dynamic classes. So simply put: Choose the implementation class to be the dynamic class.

For the rest of the article, we'll make the following assumptions about the chosen dynamic classes:

- mplements some Java interface to expose functionality
- implementation does not hold any stateful information about its client
- n bean), so the instances of the dynamic class can replace each other

Please note that these assumptions are not prerequisites. They exist just to make the realization of dynamic code a bit easier so we can focus more on the ideas and mechanisms.

With the selected dynamic classes in mind, deploying the source code is an easy task. Figure 1 shows the file structure of the Postman example.

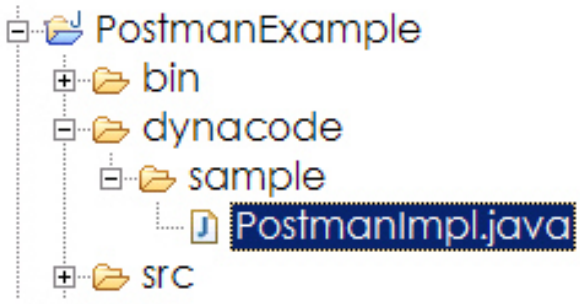

Figure 1. The file structure of the Postman example

We know "src" is source and "bin" is binary. One thing worth noting is the dynacode directory, which holds the source files of dynamic classes. Here in the example, there's only one file—PostmanImpl.java. The bin and dynacode directories are required to run the application, while src is not necessary for deployment.

Detecting file changes can be achieved by comparing modification timestamps and file sizes. For our example, a check to PostmanImpl.java is performed every time a method is invoked on the `Postman` interface. Alternatively, you may spawn a daemon thread in the background to regularly check the file changes. That may result in better performance for large-scale applications.

Compile Java code at runtime

After a source code change is detected, we come to the compilation issue. By delegating the real job to an existing Java compiler, runtime compilation can be a piece of cake. Many Java compilers are available for use, but in this article, we use the Javac compiler included in Sun's Java Platform, Standard Edition (Java SE is Sun's new name for J2SE).

At the minimum, you can compile a Java file with just one statement, providing that the tools.jar, which contains the Javac compiler, is on the classpath (you can find the tools.jar under <JDK_HOME>/lib/):

```
int errorCode = com.sun.tools.javac.Main.compile(new String[] {
          "-classpath", "bin",
          "-d", "/temp/dynacode_classes",
          "dynacode/sample/PostmanImpl.java" });
```

The class `com.sun.tools.javac.Main` is the programming interface of the Javac compiler. It provides static methods to compile Java source files. Executing the above statement has the same effect as

ıd line with the same arguments. It compiles the source file ava using the specified classpath bin and outputs its class file to the ıcode_classes. An integer returns as the error code. Zero means success; any other number indicates something has gone wrong.

The `com.sun.tools.javac.Main` class also provides another `compile()` method that accepts an additional `PrintWriter` parameter, as shown in the code below. Detailed error messages will be written to the `PrintWriter` if compilation fails.

```
// Defined in com.sun.tools.javac.Main
 public static int compile(String[] args);
 public static int compile(String[] args, PrintWriter out);
```

I assume most developers are familiar with the Javac compiler, so I'll stop here. For more information about how to use the compiler, please refer to Resources.

Load/reload Java class at runtime

The compiled class must be loaded before it takes effect. Java is flexible about class loading. It defines a comprehensive class-loading mechanism and provides several implementations of classloaders. (For more information on class loading, see Resources.)

The sample code below shows how to load and reload a class. The basic idea is to load the dynamic class using our own `URLClassLoader`. Whenever the source file is changed and recompiled, we discard the old class (for garbage collection later) and create a new `URLClassLoader` to load the class again.

```
sses.
/dynacode_classes/");
// The parent classloader.
ClassLoader parentLoader = Postman.class.getClassLoader();

// Load class "sample.PostmanImpl" with our own classloader.
URLClassLoader loader1 = new URLClassLoader(
        new URL[] { classesDir.toURL() }, parentLoader);
Class cls1 = loader1.loadClass("sample.PostmanImpl");
Postman postman1 = (Postman) cls1.newInstance();

/*
 * Invoke on postman1 ...
 * Then PostmanImpl.java is modified and recompiled.
 */

// Reload class "sample.PostmanImpl" with a new classloader.
URLClassLoader loader2 = new URLClassLoader(
        new URL[] { classesDir.toURL() }, parentLoader);
Class cls2 = loader2.loadClass("sample.PostmanImpl");
Postman postman2 = (Postman) cls2.newInstance();

/*
 * Work with postman2 from now on ...
 * Don't worry about loader1, cls1, and postman1
 * they will be garbage collected automatically.
 */
```

Pay attention to the `parentLoader` when creating your own classloader. Basically, the rule is that the parent classloader must provide all the dependencies the child classloader requires. So in the sample code, the dynamic class `PostmanImpl` depends on the interface `Postman`; that's why we use `Postman`'s classloader as the parent classloader.

We are still one step away to completing the dynamic code. Recall the example introduced earlier. There, dynamic class reload is transparent to its caller. But in the above sample code, we still have to change the service instance from `postman1` to `postman2` when the code changes. The fourth and final step will remove the need for this manual change.

Link the up-to-date class to its caller

How do you access the up-to-date dynamic class with a static reference? Apparently, a direct (normal) reference to a dynamic class's object will not do the trick. We need something between the client and the dynamic class—a proxy. (See the famous book Design Patterns for more on the Proxy pattern.)

Here, a proxy is a class functioning as a dynamic class's access interface. A client does not invoke the dynamic class directly; the proxy does instead. The proxy then forwards the invocations to the backend

dynamic class. Figure 2 shows the collaboration.

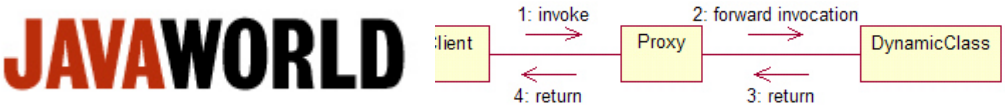

Figure 2. Collaboration of the proxy

When the dynamic class reloads, we just need to update the link between the proxy and the dynamic class, and the client continues to use the same proxy instance to access the reloaded class. Figure 3 shows the collaboration.

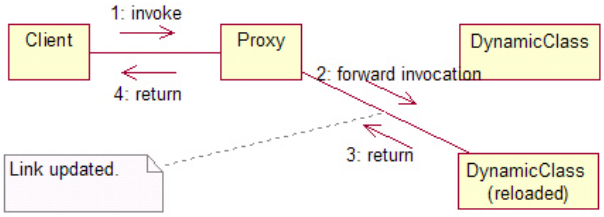

Figure 3. Collaboration of the proxy with reloaded dynamic class

In this way, changes to the dynamic class become transparent to its caller.

The Java reflection API includes a handy utility for creating proxies. The class `java.lang.reflect.Proxy` provides static methods that let you create proxy instances for any Java interface.

The sample code below creates a proxy for the interface `Postman`. (If you aren't familiar with `java.lang.reflect.Proxy`, please take a look at the Javadoc before continuing.)

```
InvocationHandler handler = new DynaCodeInvocationHandler(...);
 Postman proxy = (Postman) Proxy.newProxyInstance(
             Postman.class.getClassLoader(),
             new Class[] { Postman.class },
             handler);
```

The returned `proxy` is an object of an anonymous class that shares the same classloader with the `Postman` interface (the `newProxyInstance()` method's first parameter) and implements the `Postman` interface (the second parameter). A method invocation on the `proxy` instance is dispatched to the `handler`'s `invoke()` method (the third parameter). And `handler`'s implementation may look like the following:

JAVAWORLD