15 Replacing and Patching Core Java Classes

IN THIS CHAPTER

"A path without obstacles probably leads nowhere."

Defalque

Why Bother?

In Chapter 5, "Replacing and Patching Application Classes," we talked about the patching of Java classes to change or extend the underlying logic. The techniques presented in that chapter work for application and library classes loaded by the system or a custom class loader. However, attempting to apply the techniques to patch the core classes in a package whose name starts with `java` yields no results because the original version of the class continues to be used. Chapter 14, "Controlling Class Loading," provided a detailed discussion of how the classes are loaded, and with a little bit of reckoning, we can see why the system classes require a different approach. Recall that the system classes are loaded by the native bootstrap class loader, which does not use the `CLASSPATH` environment variable. Although the overall approach to system class patching is similar to application class patching, there are a few subtle differences, and they're the subject of this chapter.

Is there really a need to patch the core classes? In my career I have had to patch the application classes a lot more often than the system classes. One of the reasons might be that the core classes have been well designed and by now have matured into a form that suits most developers. However, every once in a while you can bump into a deficiency in a core class with no good workaround.

It is definitely not advisable to patch core Java classes as a permanent solution. This has legal consequences (the JDK license prohibits modifications to core classes) and can require additional work to migrate to a new version of JDK. However, this technique provides a lot more control to the developer. It can be used to insert traces into the JDK code and temporarily change the implementation of the core logic to suit the application needs. Last, but not least, it is just plain cool and being armed with this powerful technique would not hurt. Just be sure to read the license agreement before embarking on this path.

Patching Core Java Classes Using the Boot Class Path

As I have already mentioned, the approach to patching core classes is similar to the approach used to patch application classes. A source file needs to be obtained for a class that requires patching. JDK is conveniently distributed with the source code (thank you, Sun!), so most of the time you can just obtain the code from `src.jar`. Note that some of the system classes are shipped without the source code; this is true for the classes inside the `sun` package and other nonpublic packages. You can decompile the class files as described in Chapter 2, “Decompiling Classes,” although the license agreement must be observed.

I worked on a product called WebCream that is capable of running multiple virtual Swing clients inside the same JVM. While testing, it was observed that after running for a certain time the JVM would become locked and no new clients would be able to initialize themselves. Using the JVM thread dumps as described in Chapter 10, “Using Profilers for Application Runtime Analysis,” the examination revealed that the locking was occurring in a call to the `java.awt.Component`’s method `getTreeLock()`. The implementation of `getTreeLock()` simply returns a variable that is declared in `Component` as follows:

```
    static final Object LOCK = new AWTTreeLock();
```

Thus, AWT uses a global lock that is shared by *all* components and, if one thread fails to release the lock monitor in a timely fashion, no other thread can perform AWT and Swing operations. This was done by Java designers to prevent races when redoing a layout, but it is an absolute killer of scalability for a product such as WebCream. An immediate solution at that time was to patch the `java.awt.Component` class so that it uses a virtual client-specific lock instead of a global lock. With the patch in place, the locking of virtual clients was no longer reported.

After you get your hands on the source code, you can insert the new logic. Compile the class just like you would compile any other class, and please be sure to not add bugs. Now that you have a new version of the bytecode, the remaining task is to tell the JVM to use it instead of the original bytecode. This can be achieved by manipulating the boot class path, as was explained in Chapter 14. The bootstrap class loader uses the boot class path to locate the core classes. By default, it is set to include only `rt.jar` and possibly a few other system libraries. `rt.jar`, located in `JRE_HOME\lib`, contains most of the core classes, so if there is no source code for a class and you want to find its bytecode, check `rt.jar` first. The boot class path can be set using the `-Xbootclasspath` parameter to the Java launcher command line. Running `java -X` displays the following help:

```
C:\CovertJava\java -X
    -Xbootclasspath:<directories and zip/jar files separated by ;>
                     set search path for bootstrap classes and resources
    -Xbootclasspath/a:<directories and zip/jar files separated by ;>
                     append to end of bootstrap class path
    -Xbootclasspath/p:<directories and zip/jar files separated by ;>
                     prepend in front of bootstrap class path
    ...
```

Using the command-line parameter, we can set or augment the boot class path. Because we are interested in replacing an existing class, we use `-Xbootclasspath/p:` to prepend the directory that contains the patches in front of the default path. Running the JVM with this parameter results in the patched class being used instead of the original class.

Example of Patching `java.lang.Integer`

To put the theory in practice, let's write a simple patch to `java.lang.Integer`. For reasons unknown to the Java community, the `Integer` object is immutable. After the value is set, it cannot be changed. The idea was probably to make `Integer` objects behave like `String` objects, in that if you need to change a value that is represented by an `Integer` object, you should create a new instance and use it instead of the old value. The problem with this approach is that it results in inefficient memory usage for applications that need dynamic collections of integers. Java does not provide collection classes for primitive types, so the only way to get a dynamic array of integers is to use a `java.util.Array` of `Integer` instances. If the value of the stored integer needs to change, you must create a new instance of `Integer` and place it in the array where the old value used to be. Of course, the allocations and subsequent garbage collections produce significant overhead. A much better approach is to change the internal value of the `Integer` object. However, because `java.lang.Integer` is immutable, the only legitimate workaround is to create and use your own class that mimics the `Integer` and give it a `setValue()` method.

We, nevertheless, are going to patch the existing `java.lang.Integer` class and grant it a `setValue()` method. We will do this from a purely academic interest and to practice what we preach because we do not want to commit violations to the Java license agreement. Examining the source code for `java.lang.Integer` reveals that the value of the object is stored in a private field, `value`. Thus we must copy the source file to the `CovertJava\src\java\lang` directory and insert a method called `setValue` (see Listing 15.1).

LISTING 15.1 `setValue()` Method Source Code

```
public void setValue(int value) {
    this.value = value;
}
```

The next step is to create a test class, `CorePatchTest`, that accesses the newly inserted `setValue()` method. The code for the test class is shown in Listing 15.2.

LISTING 15.2 Using the Patched `java.lang.Integer`

```
package covertjava.patching;

public class CoreClassTest {
    public static void main(String[] args) {
        Integer i = new Integer(10);
        System.out.println("Old value = " + i);
        i.setValue(100);
        System.out.println("New value = " + i);
    }
}
```

Compiling the classes that use the patched versions of the core classes can be a little bit tricky if the public interface of the core class has changed. Trying to run `javac` on our test class results in an error because the JDK implementation of `Integer` does not have the `setValue()` method. Because of this, we cannot use Ant to compile the patched `java.lang.Integer`. The easiest workaround is to compile our patched `Integer` manually using `javac` and then copy the class file to the `CovertJava/distrib/patches` directory. We can now configure the compiler to use our patched version of `Integer` for our project, which we can do by placing the patched class on the boot class path before the original version. `javac` takes a `-bootclasspath` parameter that enables overriding the default boot class path, as does Ant's `javac` task. However, if we try to override the boot class path for `javac`, we must specify the location of `rt.jar` and all the other system libraries. That makes the build scripts dependent on the path to the JDK installation or environment variables. A simpler way is to pass `-Xbootclasspath/p:` to the JVM that runs Ant, so that instead of overriding the default path we just add an item in front of it. The `ant.bat` script uses the `ANT_OPTS` environment variable for passing command-line options to the Java invocation line. We will take advantage of this by adding the following line to `CovertJava\bin\build.bat`:

```
ANT_OPTS=-Xbootclasspath/p:..\distrib\patches
```

Now we can use Ant to build the project and the distribution libraries (the release target). Our final task after building the project is to create a batch file called `corePatchTest.bat` in the `CovertJava\bin` directory that executes `CorePatchTest`. Once again, to ensure that the patched version of `Integer` is used, we pass the `-Xbootclasspath` parameter to `java`. The relevant source code for `corePatchTest.bat` is shown in Listing 15.3.

LISTING 15.3 Executing a Test of a Core Class Patch

```
set JAVA_OPTS=-Xbootclasspath/p:..\distrib\patches
java %JAVA_OPTS% covertjava.patching.CoreClassTest
```

`corePatchTest.bat` produces the following output:

```
Old value: 10
New value: 100
```

Voilá! One more technique is added to our bag of tricks.

Quick Quiz

1. Can you think of a case in which you would want to patch a core class?
2. How is the process of patching core classes different from patching application classes?
3. Why do we need to alter the boot class path?

In Brief

- Patching core Java classes can help in debugging and understanding the JVM.
- Core classes are always loaded by the bootstrap class loader, which uses the boot class path to locate the bytecode.
- To patch a core class, the new version must be placed in the boot class path in front of the old version.
- To compile a class that uses the patched version of the core class that has changed its public interface, the patched version must be specified on the boot class path of the Java compiler.