Bytecode generation tips and tricks

Abstract

This article introduces readers to bytecode generation and also shows how to inject generated bytecode into JVM runtime. After reading this article generating a Java class shall be no more difficult for you than creating an XML document with DOM API.

Introduction

Over last couple of years bytecode generation gained significant momentum. Many tools generate bytecode instead of source code to obviate compilation step and simplify injection of generated code at runtime.
There is a number of bytecode generation libraries with BCEL[2] being the most renown and, probably, most elaborated. It is used by such tools as Xalan stylesheet compiler[9] and Mercury Interactive's Topaz J2EE Probe. Sooth to say, BCEL API is a bit cumbersome. While working on SQLC[8.2], I've created several helper classes[8.1] to make bytecode generation easier. Examples in this article use both BCEL and my classes.

Bytecode vs. source generation

You may ask: "Why to bother with bytecode generation if I can produce Java sources?" Ultimately, it's your call what technique to use. For example some people create XML documents with println(), other use DOM API. Keep the following considerations in mind:

- With Jad[6] and Jadclipse[7] .class file is as an open book as .java file. But read-only. So there is no need in (and actually no place for) formidable headers *"Generated by XXX - Do not edit!"*
- There is an additional compilation step if you produce sources instead of bytecode. Compilation at runtime and then injection of compiled files to JVM is a real big headache, especially if you do not control target JVM.
- Amount of Java code you need to write to produce Java sources will not be significantly less than amount of Java code for bytecode generator. Actually, it may be more.
- Use of different template engines has its own vices. First of all generation logic gets distributed between Java code and template code, which makes the whole solution less manageable and more fragile, because templates aren't easy to debug. Secondly, as you start adding more and more to your templates they will pretty soon become unreadable. Take a look at Xdoclet templates – they remind me cuneiform on the Hammurabi stella.

Usage scenarios

BCEL can be used for code generation (this is what XSLTC and SQLC do) and for code modification (this is what Mercury Topaz J2EE Probe does). This article covers the first scenario – code generation. In MDA parlance it describes how to build **T** in **PIM+T -> PSM** equation where **PSM** is Java bytecode and **PIM** is XSL stylesheet in case of XSLTC and SQL statements in case of SQLC

Generating interfaces

Generating interfaces is the simplest task because interfaces' methods are all abstract. First of all we need to instantiate `com.pavelvlasov.codegen.Interface`:

```
1. Interface myInterface=new Interface("public
   interface com.myorg.myapp.MyGeneratedInterface
   extends java.io.Serializable", "My Generated
   interface", null);
```

Then add a method. This is as straightforward as creating an interface itself:

```
1. myInterface.addMethod("void setMyValue
   (java.lang.String str)", null, "My generated
   method");
```

If the method parameters are not known at coding time then they can be supplied in the second parameter of `addMethod()`, which shall be either **null** or a collection of `com.pavelvlasov.util.Parameter` implementations.

Adding a field is also an one-liner:

```
1. myInterface.addField("MY_CONSTANT",
   "java.lang.String");
```

But fields in interfaces are static final and thus shall be initialized in the static initializer:

```
1. InstructionList il=new InstructionList();
2. il.append(new LDC(myInterface.getClassGen().
   getConstantPool().addString("My constant
   value")));
3. il.append(myInterface.createPutField
   ("MY_CONSTANT"));
4. il.append(new RETURN());
5. myInterface.addStaticInitializer(il,
   "Initializes myInterface");
```

Once an interface is created and methods added it should be either saved to file for future use or injected into JVM runtime. Interface can be saved to file by invoking its save (File) method. Another way is to obtain BCEL JavaClass object using `getJavaClass()` method and do whatever needed with that object.

In case the generated interface needs to be injected into Java runtime `com.pavelvlasov.codegen.InjectingClassLoader` comes into play.

```
1. ClassLoader parentClassLoader = ...;
2. InjectingClassLoader icl=new
   InjectingClassLoader(parentClassLoader);
3. icl.consume(myInterface);
4. Class myInterfaceClass=icl.loadClass
   ("com.myorg.myapp.MyGeneratedInterface");
5. ...
```

Generating classes

Generating classes is a bit more complicated task comparing to interfaces generation because of the need to generate method implementations.

Advices on code generation:

- Minimize amount of code to be generated by moving functionality to superclass and helper classes
- Create a template method in Java, compile it.
- Run `org.apache.bcel.util.Class2HTML` to generate class HTML documentation.
- Run `org.apache.bcel.util.BCELifier`. It will create code, which would generate template class. There is a bug in BCELifier shipped in BCEL 5.1 and it doesn't work on all classes. See “Resources” for a download link of fixed version.
- Use instructions produced by BCELifier as a starting point. You can also copy bytecode instructions from generated HTML documentation to your generator method; comment them out and then write code using commented instructions as guidelines. The most convenient method, at least for me, is to use -a option in Jad – it produces already commented JVM instructions. There is an option “Generate JVM instructions as comments” in Jadclipse as well.
- Run `org.apache.bcel.verifier.Verifier` or `com.pavelvlasov.codegen.ClassGeneratorBase.verify()` on the generated classes.
- Use Jad and/or Jadclipse to decompile generated files and verify method logic.

Writing linear bytecode is very simple, as we've seen from the previous section. Branches (if, while, ...) and exception handlers are the things which require attention. Let's see how to generate code, which has both branches and exception handlers using the advices above. This is the code we are going to generate:

```
1. public int getMyInt(String str) {
2.     if (str==null) {
3.             return 0;
4.     } else {
5.             try {
6.                     return Integer.parseInt
   (str);
7.             } catch (NumberFormatException e) {
8.                     return -1;
9.             }
10.    }
11.}
```

this is the output of BCELifier:

```
1. InstructionList il = new InstructionList();
2. MethodGen method = new MethodGen(ACC_PUBLIC,
   Type.INT, new Type[] { Type.STRING }, new
   String[] { "arg0" }, "getMyInt",
   "com.pavelvlasov.codegen.samples.TemplateClass"
   , il, _cp);
3.
4. InstructionHandle ih_0 = il.append
   (_factory.createLoad(Type.OBJECT, 1));
5. BranchInstruction ifnonnull_1 =
   _factory.createBranchInstruction
   (Constants.IFNONNULL, null);
6. il.append(ifnonnull_1);
7. InstructionHandle ih_4 = il.append(new PUSH
   (_cp, 0));
8. il.append(_factory.createReturn(Type.INT));
9. InstructionHandle ih_6 = il.append
   (_factory.createLoad(Type.OBJECT, 1));
10.il.append(_factory.createInvoke
   ("java.lang.Integer", "parseInt", Type.INT, new
   Type[] { Type.STRING },
   Constants.INVOKESTATIC));
11.InstructionHandle ih_10 = il.append
   (_factory.createReturn(Type.INT));
12.InstructionHandle ih_11 = il.append
   (_factory.createStore(Type.OBJECT, 2));
13.InstructionHandle ih_12 = il.append(new PUSH
   (_cp, -1));
14.InstructionHandle ih_13 = il.append
   (_factory.createReturn(Type.INT));
15.ifnonnull_1.setTarget(ih_6);
16.method.addExceptionHandler(ih_6, ih_10, ih_11,
   new ObjectType
   ("java.lang.NumberFormatException"));
17.method.setMaxStack();
18.method.setMaxLocals();
19._cg.addMethod(method.getMethod());
20.il.dispose();
```

Now, we change it to be less cryptic:

```
1. MethodPrototype mp=new MethodPrototype(myClass,
   "public int getMyInt(java.lang.String str)",
   null);
2. InstructionList il = new InstructionList();
3. ExceptionHandler eh=new ExceptionHandler
   ("java.lang.NumberFormatException");
4. il.append(mp.createVariableLoad("str"));
5. BranchInstruction ifnonnull =
   InstructionFactory.createBranchInstruction
   (Constants.IFNONNULL, null);
6. il.append(ifnonnull);
7. il.append(new ICONST(0));
8. il.append(mp.createReturn());
9. InstructionHandle ih = il.append
   (mp.createVariableLoad("str"));
10.ifnonnull.setTarget(ih);
11.eh.setFrom(ih);
12.il.append(myClass.createInvoke
   ("java.lang.Integer", "int parseInt
   (java.lang.String)", null,
   Constants.INVOKESTATIC));
13.eh.setTo(il.append(mp.createReturn()));
14.eh.setHandler(il.append
   (InstructionFactory.createStore(Type.OBJECT,
   2)));
15.il.append(new ICONST(-1));
16.il.append(mp.createReturn());
17.Collection ehc=new ArrayList();
18.ehc.add(eh);
19.mp.addMethod(il, ehc, "My generated method");
```

After that we run `org.apache.bcel.verifier.Verifier` to verify the generated class. Then we run Jad and compare decompiled code with the original.

Generating documentation

If you generate classes, which implement some interface then you probably don't need to document them. E.g. Xalan XSLTC produces a bunch of classes but you don't need documentation for these classes, all you need to know is translet class name. On the other hand SQLC (see Resources) generates classes and interfaces from SQL statements and in this case documentation is necessary. The good thing about classes from `com.pavelvlasov.codegen`

package is that if you use them to generate bytecode then effort needed to generate documentation is close to zero. What you need to do is to use class `com.pavelvlasov.codegen.HtmlDocConsumer`

```
1. HtmlDocConsumer consumer=
2.     new HtmlDocConsumer(
3.             new File("generated"),
4.             new File("generated_doc"));
5.
6. com.pavelvlasov.codegen.Class myClass=
7.     new com.pavelvlasov.codegen.Class(
8.             "public class
   com.myorg.myapp.MyGeneratedClass",
9.             "My Generated class",
10.            consumer.getListener());
11....
12.// Saving to file.
13.consumer.consume(myClass.getJavaClass());
14.consumer.close();
```

Runtime generation

Classes can be generated at build time and runtime. The first case is a trivial one – generated classes can be use as any other Java classes.

Runtime code generation is more interesting theme. It has already been shown how to inject generated classes into JVM runtime. How to use them? If generated classes implement interfaces or extend classes known at compile time then the answer is obvious – instantiate and cast.

What if generated classes do not fall in the category mentioned above? How to use classes not known at compile time? Well, the answer is that information about generated classes can be obtained through reflection. Scripting environments such as JSP, JSTL, Velocity and script interpreters will use generated classes as happily any other class loaded from classpath.

An important note about runtime generation: BCEL is not threadsafe. If you are going to generate classes in multithrreaded environment then each generating thread shall be provided its own classloader with BCEL classes loaded into this particular classloader. This will result in increased memory footprint because BCEL classes will be presented in memory one time per generating thread.

Conclusion

I hope, dear reader, that after reading this article bytecode generation will become part of your skillset.

There are many cases when there is a model/data structure in non-Java software system., which shall be used in Java. It can be XML schema, mainframe map, database metadata, ... There is probably no reason to write a generator for XML schema because it has already been done many times, just select XML-Java mapping solution which fits your need. But for not so common cases bytecode engineering is a good choice to generate bridges between software components.

Resources

1. Source code - see donwnload page on http://www.hammurapi.biz. You need to download BCEL, Antlr and PvCommons libraries (see below) to run the samples.
2. BCEL (http://jakarta.apache.org/bcel/) - Byte Code Engineering Library.
3. BCEL with fixed bug http://www.pavelvlasov.com/bcel-5.1-fixed.zip
4. JVM instructions reference http://cat.nyu.edu/~meyer/jvmref/
5. Antlr (http://www.antlr.org) – Parser generator.
6. Jad (http://kpdus.tripod.com/jad.html) – fast decompiler
7. Jadclipse (http://sourceforge.net/projects/jadclipse/) - Eclipse plugin for Jad
8. Common library (http://www.hammurapi.biz)
   8.1. `com.pavelvlasov.codegen package` – Code generation classes.
   8.2. `com.pavelvlasov.sqlc package` – SQL compiler. Uses codegeneration classes to compile SQL statements to Java classes.
   8.3. `com.pavelvlasov.cache.sql` – Example of classes compiled by SQLC
9. Xalan (http://xml.apache.org/xalan-j/) - Java XSL transformer. Ships with XSLT Compiler, which uses BCEL to compile xsl stylesheets into “translets”.