Java programming dynamics, Part 7: Bytecode engineering with BCEL

Apache BCEL lets you get to the details of JVM assembler language for classworking

Dennis Sosnoski

April 14, 2004

The Apache Byte Code Engineering Library (BCEL) lets you dig into the bytecode of Java classes. You can use it to transform existing class representations or construct new ones, and because BCEL works at the level of individual JVM instructions, it gives you the utmost power over your code. That power comes with a cost in complexity, though. In this article, Java consultant Dennis Sosnoski gives you the BCEL basics and guides you through an example BCEL application so you can decide for yourself if the power justifies the complexity.

In the last three articles of this series, I've shown you how to use the Javassist framework for classworking. This time I'm going to cover a very different approach to bytecode manipulation, using the Apache Byte Code Engineering Library (BCEL). BCEL operates at the level of actual JVM instructions, unlike the source code interface supported by Javassist. The low-level approach makes BCEL very good for when you really want to control every step of the program execution, but it also makes working with BCEL a lot more complex than using Javassist for cases where both will work.

I'm going to start out by covering the basic BCEL architecture, then devote most of this article to rebuilding my first Javassist classworking example with BCEL. I'll finish up with a quick look at some of the tools included in the BCEL package and some of the applications developers have built on top of BCEL.

BCEL class access

BCEL gives you all the same basic capabilities as Javassist to inspect, edit, and create Java binary classes. The obvious difference with BCEL is that everything is designed to work at the level of JVM assembler language, rather than the source code interface provided by Javassist. There are some deeper differences under the covers, including the use of two separate hierarchies of components within BCEL -- one for inspecting existing code and the other for creating new code.
I'm going to assume you're familiar with Javassist from the previous articles in this series (see the sidebar Don't miss the rest of this series). I'll therefore concentrate on the differences that are likely to confuse you when you start working with BCEL.

As with Javassist, the class inspection aspect of BCEL basically duplicates what's available directly in the Java platform through the Reflection API. This duplication is necessary in a classworking toolkit because you generally don't want to load the classes you're working with until after they've been modified.

BCEL provides some basic constant definitions in the org.apache.bcel package, but aside from these definitions all the inspection-related code is in the org.apache.bcel.classfile package. The starting point within this package is the JavaClass class. This class plays about the same role in accessing class information using BCEL as java.lang.Class does when using regular Java reflection. JavaClass defines methods to get the field and method information for the class, as well as structural information about superclass and interfaces. Unlike java.lang.Class, JavaClass also provides access to the internal information for the class, including the constant pool and attributes, and the complete binary class representation as a byte stream.

JavaClass instances are usually created by parsing the actual binary class. BCEL provides the org.apache.bcel.Repository class to handle the parsing for you. By default, BCEL parses and caches the representations of classes found in the JVM classpath, getting the actual binary class representations from an org.apache.bcel.util.Repository instance (note the difference in the package name). org.apache.bcel.util.Repository is actually an interface for a source of binary class representations. You can substitute other paths for looking up class files, or other ways of accessing class information, in place of the default source that uses the classpath.

### Changing classes

Besides reflection-style access to class components, org.apache.bcel.classfile.JavaClass also provides methods for altering the class. You can use these methods to set any of the class components to new values. They're not generally of much direct use, though, because the other classes in the package don't provide support for constructing new versions of the components in any reasonable manner. Instead, there's an entire separate set of classes in the
org.apache.bcel.generic package that provides editable versions of the same components represented by org.apache.bcel.classfile classes.

Just as org.apache.bcel.classfile.JavaClass is the starting point for using BCEL to inspect existing classes, org.apache.bcel.generic.ClassGen is your starting point for creating new classes. It also works for modifying existing classes -- to handle that case, there's a constructor that takes a JavaClass instance and uses it to initialize the ClassGen class information. Once you're done with your class modifications, you can get a usable class representation from the ClassGen instance by calling a method that returns a JavaClass, which can in turn be converted to a binary class representation.

Ask the expert: Dennis Sosnoski on JVM and bytecode issues
For comments or questions about the material covered in this article series, as well as anything else that pertains to Java bytecode, the Java binary class format, or general JVM issues, visit the JVM and Bytecode discussion forum, moderated by Dennis Sosnoski.

Sound confusing? I think it is. In fact, going back and forth between the two packages is one of the most awkward aspects of working with BCEL. The duplicate class structures tend to get in the way, so if you're doing much with BCEL, it may be worthwhile to write wrapper classes that can hide some of these differences. For this article, I'll work mainly with the org.apache.bcel.generic package classes and avoid the use of wrappers, but it's something for you to keep in mind for your own work.

Besides ClassGen, the org.apache.bcel.generic package defines classes to manage the construction of various class components. These construction classes include ConstantPoolGen for handling the constant pool, FieldGen and MethodGen for fields and methods, and InstructionList for working with sequences of JVM instructions. Finally, the org.apache.bcel.generic package also defines classes to represent every type of JVM instruction. You can create instances of these classes directly, or in some cases by using the org.apache.bcel.generic.InstructionFactory helper class. The advantage of using InstructionFactory is that it handles many of the bookkeeping details of instruction building for you (including adding items to the constant pool as needed for the instructions). You'll see how to make all these classes play together in the next section.

Classworking with BCEL

For an example of applying BCEL, I'll use the same task I used as a Javassist example back in Part 4 -- measuring the time taken to execute a method. I'll even use the same approach I used with Javassist: I'll create a copy of the original method to be timed using a modified name, then replace the body of the original method with code that wraps timing calculations around a call to the renamed method.

Selecting a guinea pig

Listing 1 gives an example method I'll use for demonstration purposes: the buildString method of the StringBuilder class. As I said in Part 4, this method constructs a string of any requested
length by doing exactly what any Java performance guru will tell you not to do -- it repeatedly appends a single character to the end of a string to create a longer string. Because strings are immutable, this approach means a new string will be constructed each time through the loop, with the data copied from the old string and a single character added at the end. The net effect is that this method will run into more and more overhead as it's used to create longer strings.

Listing 1. Method to be timed

```java
public class StringBuilder {
    private String buildString(int length) {
        String result = "";
        for (int i = 0; i < length; i++) {
            result += (char)(i%26 + 'a');
        }
        return result;
    }

    public static void main(String[] argv) {
        StringBuilder inst = new StringBuilder();
        for (int i = 0; i < argv.length; i++) {
            String result = inst.buildString(Integer.parseInt(argv[i]));
            System.out.println("Constructed string of length " + result.length());
        }
    }
}
```

Listing 2 shows the source code equivalent to the classworking change I'll make with BCEL. Here the wrapper method just saves the current time, then calls the renamed original method and prints a time report before returning the result of the call to the original method.

Listing 2. Timing added to original method

```java
public class StringBuilder {
    private String buildString$impl(int length) {
        String result = "";
        for (int i = 0; i < length; i++) {
            result += (char)(i%26 + 'a');
        }
        return result;
    }

    private String buildString(int length) {
        long start = System.currentTimeMillis();
        String result = buildString$impl(length);
        System.out.println("Call to buildString$impl took " +
                            (System.currentTimeMillis()-start) + " ms.");
        return result;
    }

    public static void main(String[] argv) {
        StringBuilder inst = new StringBuilder();
        for (int i = 0; i < argv.length; i++) {
            String result = inst.buildString(Integer.parseInt(argv[i]));
            System.out.println("Constructed string of length " + result.length());
        }
    }
}
```
Coding the transform

Implementing the code to add method timing uses the BCEL APIs I outlined in the [BCEL class access](#) section. Working at the level of JVM instructions makes the code a lot longer than the Javassist example back in [Part 4](#), so here I'm going to walk through it a piece at a time before giving you the complete implementation. In the final code, all these pieces will make up a single method, one that takes a pair of parameters: `cgen`, an instance of the `org.apache.bcel.generic.ClassGen` class initialized with the existing information for the class being modified; and `method`, an `org.apache.bcel.classfile.Method` instance for the method I'm going to time.

Listing 3 has the first piece of code for the transform method. As you can see from the comments, the first part just initializes the basic BCEL components I'm going to use, which includes initializing a new `org.apache.bcel.generic.MethodGen` instance using the information for the method to be timed. I set an empty instruction list for this `MethodGen`, which I'll later fill in with the actual timing code. In the second part, I create a second `org.apache.bcel.generic.MethodGen` instance from the original method, then remove the original method from the class. On this second `MethodGen` instance, I just change the name to use a "$impl" suffix, then call `getMethod()` to convert the modifiable method information to a fixed form as an `org.apache.bcel.classfile.Method` instance. I then use the `addMethod()` call to add the renamed method to the class.

**Listing 3. Adding the interception method**

```java
// set up the construction tools
InstructionFactory ifact = new InstructionFactory(cgen);
InstructionList ilist = new InstructionList();
ConstantPoolGen pgen = cgen.getConstantPool();
String cname = cgen.getClassName();
MethodGen wrapgen = new MethodGen(method, cname, pgen);
wrapgen.setInstructionList(ilist);

// rename a copy of the original method
MethodGen methgen = new MethodGen(method, cname, pgen);
cgen.removeMethod(method);
String iname = methgen.getName() + "$impl";
methgen.setName(iname);
cgen.addMethod(methgen.getMethod());
```

Listing 4 gives the next piece of code for the transform method. The first part here computes the space occupied by the method call parameters on the stack. This piece is needed because to store the start time on the stack frame before calling the wrapped method I need to know what offset can be used for a local variable (note that I could use BCEL's local variable handling to get the same effect, but for this article I prefer an explicit approach). The second part of this code generates the call to `java.lang.System.currentTimeMillis()` to get the start time, saving it to the computed local variable offset in the stack frame.

You might wonder why I check whether the method is static at the start of my parameter size calculation, then initialize the stack frame slot to zero if it is (as opposed to one if it is not). This approach relates to how the Java language handles method calls. For non-static methods, the first (hidden) parameter on every call is the `this` reference for the target object, which I need to take into account when computing the complete parameter set size on the stack frame.
Listing 4. Setting up for the wrapped call

```java
// compute the size of the calling parameters
Type[] types = methgen.getArgumentTypes();
int slot = methgen.isStatic() ? 0 : 1;
for (int i = 0; i < types.length; i++) {
    slot += types[i].getSize();
}

// save time prior to invocation
ilist.append(ifact.createInvoke("java.lang.System",
    "currentTimeMillis", Type.LONG, Type.NO_ARGS,
    Constants.INVOKESTATIC));
ilist.append(InstructionFactory.createStore(Type.LONG, slot));
```

Listing 5 shows the code to generate the call to the wrapped method and save the result (if any). The first part of this piece again checks whether the method is static. If the method is not static, I generate code to load the `this` object reference to the stack, and also set the method call type to virtual (rather than static). The `for` loop then generates code to copy all call parameter values to the stack, the `createInvoke()` method generates the actual call to the wrapped method, and the final `if` statement saves the result value to another local variable position in the stack frame (if the result type is not void).

Listing 5. Calling the wrapped method

```java
// call the wrapped method
int offset = 0;
short invoke = Constants.INVOKESTATIC;
if (!methgen.isStatic()) {
    ilist.append(InstructionFactory.createLoad(Type.OBJECT, 0));
    offset = 1;
    invoke = Constants.INVOKEVIRTUAL;
}
for (int i = 0; i < types.length; i++) {
    Type type = types[i];
    ilist.append(InstructionFactory.createLoad(type, offset));
    offset += type.getSize();
}
Type result = methgen.getReturnType();
ilist.append(ifact.createInvoke(cname,
    iname, result, types, invoke));

// store result for return later
if (result != Type.VOID) {
    ilist.append(InstructionFactory.createStore(result, slot+2));
}
```

Now into the wrap up. Listing 6 generates the code to actually compute the number of milliseconds elapsed since the start time, and to print it out as a nicely formatted message. This part looks very complex, but most of the operations are actually just writing individual pieces of the output message. It does illustrate several types of operations I didn't use in the earlier code, including a field access (to `java.lang.System.out`) and a few different instruction types. Most of these should be easy to understand if you think in terms of the JVM as a stack-based processor, so I won't go into details here.
Listing 6. Computing and printing time used

```java
// print time required for method call
ilist.append(ifact.createFieldAccess("java.lang.System", "out",
    new ObjectType("java.io.PrintStream"), Constants.GETSTATIC));
ilist.append(InstructionConstants.DUP);
ilist.append(InstructionConstants.DUP);
String text = "Call to method " + methgen.getName() + " took ";
ilist.append(new PUSH(pgen, text));
ilist.append(ifact.createInvoke("java.io.PrintStream", "print",
    Type.VOID, new Type[] { Type.STRING }, Constants.INVOKEVIRTUAL));
ilist.append(ifact.createInvoke("java.lang.System",
    "currentTimeMillis", Type.LONG, Type.NO_ARGS,
    Constants.INVOKESTATIC));
ilist.append(InstructionFactory.createLoad(Type.LONG, slot));
ilist.append(InstructionConstants.LSUB);
ilist.append(ifact.createInvoke("java.io.PrintStream", "print",
    Type.VOID, new Type[] { Type.LONG }, Constants.INVOKEVIRTUAL));
ilist.append(new PUSH(pgen, " ms.");
ilist.append(ifact.createInvoke("java.io.PrintStream", "println",
    Type.VOID, new Type[] { Type.STRING }, Constants.INVOKEVIRTUAL));
```

After the timing message code is generated, all that's left for Listing 7 is the completion of the wrapper method code with a return of the saved result value (if any) from the wrapped method call, followed by the finalizing of the constructed wrapper method. This last part involves several steps. The call to `stripAttributes(true)` just tells BCEL not to generate debug information for the constructed method, while the `setMaxStack()` and `setMaxLocals()` calls calculate and set the stack usage information for the method. After that's been done, I can actually generate the finalized version of the method and add it to the class.

Listing 7. Completing the wrapper

```java
// return result from wrapped method call
if (result != Type.VOID) {
    ilist.append(InstructionFactory.createLoad(result, slot+2));
}
ilist.append(InstructionFactory.createReturn(result));
```

Listing 8 shows the complete code (slightly reformatted to fit the width), including a `main()` method that takes the name of the class file and method to be transformed:

Listing 8. The complete transform code

```java
public class BCELTiming {
    private static void addWrapper(ClassGen cgen, Method method) {
        // set up the construction tools
        InstructionFactory ifact = new InstructionFactory(cgen);
        InstructionList ilist = new InstructionList();
        ConstantPoolGen pgen = cgen.getConstantPool();
        String cname = cgen.getClassName();
        MethodGen wrapgen = new MethodGen(method, cname, pgen);
```
wrapgen.setInstructionList(ilist);

// rename a copy of the original method
MethodGen methgen = new MethodGen(method, cname, pgen);
cgen.removeMethod(method);
String iname = methgen.getName() + "$impl$);
methgen.setName(iname);
cgen.addMethod(methgen.getMethod());
Type result = methgen.getReturnType();

// compute the size of the calling parameters
type[] types = methgen.getArgumentTypes();
int slot = methgen.isStatic() ? 0 : 1;
for (int i = 0; i < types.length; i++) {
    slot += types[i].getSize();
}

// save time prior to invocation
ilist.append(ifact.createInvoke("java.lang.System",
    "currentTimeMillis", Type.LONG, Type.NO_ARGS,
    Constants.INVOKESTATIC));
ilist.append(InstructionFactory.
    createStore(Type.LONG, slot));

// call the wrapped method
int offset = 0;
short invoke = Constants.INVOKESTATIC;
if (!methgen.isStatic()) {
    ilist.append(InstructionFactory.
        createLoad(Type.OBJECT, 0));
    offset = 1;
    invoke = Constants.INVOKEVIRTUAL;
}
for (int i = 0; i < types.length; i++) {
    Type type = types[i];
ilist.append(InstructionFactory.
        createLoad(type, offset));
    offset += type.getSize();
}
ilist.append(ifact.createInvoke(cname,
iname, result, types, invoke));

// store result for return later
if (result != Type.VOID) {
    ilist.append(InstructionFactory.
        createStore(result, slot+2));
}

// print time required for method call
ilist.append(ifact.createFieldAccess("java.lang.System",
    "out", new ObjectType("java.io.PrintStream"),
    Constants.GETSTATIC));
ilist.append(InstructionConstants.DUP);
ilist.append(InstructionConstants.DUP);
String text = "Call to method " + methgen.getName() + " took ";
ilist.append(new PUSH(pgen, text));
ilist.append(ifact.createInvoke("java.io.PrintStream",
    "print", Type.VOID, new Type[] { Type.STRING },
    Constants.INVOKEVIRTUAL));
ilist.append(ifact.createInvoke("java.lang.System",
    "currentTimeMillis", Type.LONG, Type.NO_ARGS,
    Constants.INVOKESTATIC));
ilist.append(InstructionFactory.
    createLoad(Type.LONG, slot));
ilist.append(InstructionConstants.LSUB);
ilist.append(ifact.createInvoke("java.io.PrintStream",};
"print", Type.VOID, new Type[] { Type.LONG },
        Constants.INVOKEVIRTUAL));
    ilist.append(new PUSH(pgen, " ms.");
    ilist.append(ifact.createInvoke("java.io.PrintStream",
        "println", Type.VOID, new Type[] { Type.STRING },
        Constants.INVOKEVIRTUAL));

    // return result from wrapped method call
    if (result != Type.VOID) {
        ilist.append(InstructionFactory.createLoad(result, slot+2));
    }
    ilist.append(InstructionFactory.createReturn(result));

    // finalize the constructed method
    wrapgen.stripAttributes(true);
    wrapgen.setMaxStack();
    wrapgen.setMaxLocals();
    cgen.addMethod(wrapgen.getMethod());
    ilist.dispose();
}

public static void main(String[] argv) {
    if (argv.length == 2 && argv[0].endsWith(".class")) {
        try {
            JavaClass jclas = new ClassParser(argv[0]).parse();
            ClassGen cgen = new ClassGen(jclas);
            Method[] methods = jclas.getMethods();
            int index;
            for (index = 0; index < methods.length; index++) {
                if (methods[index].getName().equals(argv[1])) {
                    break;
                }
            }
            if (index < methods.length) {
                addWrapper(cgen, methods[index]);
                FileOutputStream fos = new FileOutputStream(argv[0]);
                cgen.getJavaClass().dump(fos);
                fos.close();
            } else {
                System.err.println("Method " + argv[1] +
                    " not found in " + argv[0]);
            }
            } catch (IOException ex) {
                ex.printStackTrace(System.err);
            }
        } else {
            System.out.println("Usage: BCELTiming class-file method-name");
        }
    }
}

Taking it out for a spin

Listing 9 shows the results of first running the StringBuilder program in unmodified form, then running the BCELTiming program to add timing information, and finally running the StringBuilder program after it's been modified. You can see how StringBuilder starts reporting execution times after it's been modified, and how the times increase much faster than the length of the constructed string because of the inefficient string construction code.
Listing 9. Running the programs

[dennis]$ java StringBuilder 1000 2000 4000 8000 16000
Constructed string of length 1000
Constructed string of length 2000
Constructed string of length 4000
Constructed string of length 8000
Constructed string of length 16000

[dennis]$ java -cp bcel.jar:. BCELTiming StringBuilder.class buildString

[dennis]$ java StringBuilder 1000 2000 4000 8000 16000
Call to method buildString$impl took 20 ms.
Constructed string of length 1000
Call to method buildString$impl took 79 ms.
Constructed string of length 2000
Call to method buildString$impl took 250 ms.
Constructed string of length 4000
Call to method buildString$impl took 879 ms.
Constructed string of length 8000
Call to method buildString$impl took 3875 ms.
Constructed string of length 16000

Wrapping up BCEL

There's more to BCEL than just the basic classworking support I've shown in this article. It also includes a full verifier implementation to make sure that a binary class is valid according to the JVM specification (see org.apache.bcel.verifier.VerifierFactory), a disassembler that generates a nicely framed and linked JVM-level view of a binary class, and even a BCEL program generator that outputs source code for a BCEL program to build a class you provide. (The org.apache.bcel.util.BCELifier class is not included in the Javadocs, so look to the source code for usage. This feature is intriguing, but the output is probably too cryptic to be of use to most developers).

In my own use of BCEL, I've found the HTML disassembler especially useful. To try it out, just execute the org.apache.bcel.util.Class2HTML class from the BCEL JAR, with the path to the class file you want to disassemble as a command line argument. It'll generate the HTML files in the current directory. For example, here I'll disassemble the StringBuilder class I used for my timing example:

[dennis]$ java -cp bcel.jar org.apache.bcel.util.Class2HTML StringBuilder.class
Processing StringBuilder.class...Done.

Figure 1 is a screen capture of the framed output generated by the disassembler. In this shot the large frame in the upper right shows the disassembly of the timing wrapper method added to the StringBuilder class. The full HTML output is included in the download files -- just open the StringBuilder.html file in a browser window if you'd like to view this live."
Currently, BCEL is probably the most widely used framework for Java classworking. It lists a number of other projects that use BCEL on the Web site, including the Xalan XSLT compiler, the AspectJ extension to the Java programming language, and several JDO implementations. Many other unlisted projects are also using BCEL, including my own JiBX XML data binding project. However, several of the projects listed by BCEL have since switched to other libraries, so don't take the length of the list as an absolute guide to BCEL's popularity.

The big advantages of BCEL are its commercial-friendly Apache licensing and its extensive JVM instruction-level support. These features, combined with its stability and longevity, have made it a very popular choice for classworking applications. BCEL does not seem all that well designed for either speed or ease of use, though. Javassist offers a much friendlier API for most purposes, with equivalent (or perhaps even better) speed, at least in my simple tests. If your projects can make use of software using the Mozilla Public License (MPL) or GNU Lesser General Public License (LGPL), Javassist may be a better choice right now (it's available under either of these licenses).

Up next

Now that I've introduced you to both Javassist and BCEL, my next article in this series will dig into a more useful application of classworking than what you've seen so far. Back in Part 2, I demonstrated how reflection calls to methods are much slower than direct calls. In Part 8, I'll show how you can use both Javassist and BCEL to replace reflection calls with dynamically generated
code at runtime -- with a dramatic improvement in performance. Check back next month for another dose of *Java programming dynamics* to find out the details.
## Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>j-dyn7.zip</td>
<td>462KB</td>
</tr>
</tbody>
</table>
Related topics

- Get all the details on the open source Byte Code Engineering Library at the Apache project page.
- Learn more about the Java bytecode design in "Java bytecode: Understanding bytecode makes you a better programmer" (developerWorks, July 2001) by Peter Haggar.
- For an excellent reference to the JVM architecture and instruction set, see Inside the Java Virtual Machine, by Bill Venners (Artima Software, Inc., 2004). You can view some sample chapters online to get a look at it before you purchase.
- You can purchase or view the official Java Virtual Machine Specification online for the definitive word on all aspects of JVM operation.
- AspectJ extends the Java language with aspect-oriented features, using BCEL to weave code into classes generated by the compiler. Learn all about it at the Eclipse project page.
- For some other projects making use of BCEL, check out the Apache Xalan XSLTC compiler for XSL stylesheets, the Hansel JUnit extension that monitors code coverage in tests, and the author's own JiBX framework for fast XML data binding.
- Want to find out more about aspect-oriented programming? Try "Improve modularity with aspect-oriented programming" (developerWorks January 2002) by Nicholas Lesiecki for an overview of working with the AspectJ language.
- The open source Jikes Project provides a very fast and highly compliant compiler for the Java programming language. Use it to generate your bytecode the old fashioned way -- from Java source code.

© Copyright IBM Corporation 2004
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)