Java programming dynamics, Part 4: Class transformation with Javassist

Using Javassist to transform methods in bytecode

Dennis Sosnoski (dms@sosnoski.com), President, Sosnoski Software Solutions, Inc.

Date: 16 Sep 2003
Level: Intermediate
Also available in: Chinese Japanese Vietnamese

Activity: 20713 views
Comments: 0 (View | Add comment - Sign in)

Average rating (23 votes)
Rate this article

Summary: Bored with Java classes that execute just the way the source code was written? Then cheer up, because you're about to find out about twisting classes into shapes never intended by the compiler! In this article, Java consultant Dennis Sosnoski kicks his *Java programming dynamics* series into high gear with a look at Javassist, the bytecode manipulation library that's the basis for the aspect-oriented programming features being added to the widely used JBoss application server. You'll find out the basics of transforming existing classes with Javassist and see both the power and the limitations of this framework's source code approach to classworking.

View more content in this series

Tag this! Update My dW interests (Log in | What's this?)

After covering the basics of the Java class format and runtime access through reflection, it's time to move this series on to more advanced topics. This month I'll start in on the second part of the series, where the Java class information becomes just another form of data structure to be manipulated by applications. I'll call this whole topic area *classworking*.

I'll start classworking coverage with the Javassist bytecode manipulation library. Javassist isn't the only library for working with bytecode, but it does have one feature in particular that makes it a great starting point for experimenting with classworking: you can use Javassist to alter the bytecode of a Java class without actually needing to learn anything about bytecode or the Java virtual machine (JVM) architecture. This is a mixed blessing in some respects -- I don't generally advocate messing with technology you don't understand -- but it certainly makes bytecode manipulation much more accessible than with frameworks where you work at the level of individual instructions.

Javassist basics

Javassist lets you inspect, edit, and create Java binary classes. The inspection aspect mainly duplicates what's available directly in Java through the Reflection API, but having an alternative way to access this information is useful when you're actually modifying classes rather than just executing them. This is because the JVM design doesn't provide you any access to the raw class data after it's been loaded into the JVM. If you're going to work with classes as data, you need to do so outside of the JVM.

Javassist uses the `javassist.ClassPool` class to track and control the classes you're manipulating. This class works a lot like a JVM classloader, but with the important difference that rather than linking loaded classes for execution as part of your application, the class pool makes loaded classes usable as data through the Javassist API. You can use a default class pool that loads from the JVM search path, or define one that searches your own list of paths. You can even load binary classes directly from byte arrays or streams, and create new classes from scratch.

Don't miss the rest of this series

Part 1, "Classes and class loading" (April 2003)

Part 2, "Introducing reflection" (June 2003)

Part 3, "Applied reflection" (July 2003)

Part 5, "Transforming classes on-the-fly" (February 2004)

Part 6, "Aspect-oriented changes with Javassist" (March 2004)

Part 7, "Bytecode engineering with BCEL" (April 2004)

Part 8, "Replacing reflection with code generation" (June 2004)

Classes loaded in a class pool are represented by `javassist.CtClass` instances. As with the standard Java `java.lang.Class` class, `CtClass` provides methods for inspecting class data such as fields and methods. That's just the start for `CtClass`, though, which also defines methods for adding new fields, methods, and constructors to the class, and for altering the class name, superclass, and interfaces. Oddly, Javassist does not provide any way of deleting fields, methods, or constructors from a class.

Fields, methods, and constructors are represented by `javassist.CtField`, `javassist.CtMethod`, and `javassist.CtConstructor` instances, respectively. These classes define methods for modifying all aspects of the item represented by the class, including the actual bytecode body of a method or constructor.

The source of all bytecode

Javassist lets you completely replace the bytecode body of a method or constructor, or selectively add bytecode at the beginning or end of the existing body (along with a couple of other variations for constructors). Either way, the new bytecode is passed as a Java-like source code statement or block in a `String`. The Javassist methods effectively compile the source code you provide into Java bytecode, which they then insert into the body of the target method or constructor.

The source code accepted by Javassist doesn't exactly match the Java language, but the main difference is just the addition of some special identifiers used to represent the method or constructor parameters, method return value, and other items you may want to use in your inserted code. These special identifiers all start with the $ symbol, so they're not going to interfere with anything you'd otherwise do in your code.

There are also some restrictions on what you can do in the source code you pass to Javassist. The first restriction is the actual format, which must be a single statement or block. This isn't much of a restriction for most purposes, because you can put any sequence of statements you want in a block. Here's an example using the special Javassist identifiers for the first two method parameter values to show how this works:

```
{
  System.out.println("Argument 1: " + $1);
  System.out.println("Argument 2: " + $2);
}
```

A more substantial limitation on the source code is that there's no way to refer to local variables declared outside the statement or block being added. This means that if you're adding code at both the start and end of a method, for instance, you generally won't be able to pass information from the code added at the start to the code added at the end. There are ways around this limitation, but the workarounds are messy -- you generally need to find a way to merge the separate code inserts into a single block.

Back to top

Classworking with Javassist

For an example of applying Javassist, I'll use a task I've often handled directly in source code: measuring the time taken to execute a method. This is easy enough to do in the source; you just record the current time at the start of the method, then check the current time again at the end of the method and find the difference between the two values. If you don't have source code, it's normally much more difficult to get this type of timing information. That's where classworking comes in handy -- it lets you make changes like this for any method, without needing source code.

Listing 1 shows a (bad) example method that I'll use as a guinea pig for my timing experiments: the `buildString` method of the `StringBuilder` class. 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 in order 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.

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.

Listing 1. Method to be timed

```
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());
```

More Less
ajax and android apache api application application_a... application_... architecture architecture_... atom business_model... business_proce... business_proc... business_servi... cloud cloud_computi... components configuration_... databases_and... design development dojo dom_(document_... eclipse frameworks general_progr... google green_it html html5 integration interoperabil... j2ee_(java_2_e... java java_technol... javascript jquery jsf json linux mashups messaging on_demand_busi... open_source open_technolog... parsing patterns performance perl php php_(hypertex... programming python rest schemas security service services soa soa_(service... soap_(simple_... spring standards standards_and... tips virtualization web web_2.0 web_authoring

```
        }
    }
}
```

Adding method timing

Because I have the source code available for this method, I'll show you how I would add the timing information directly. This will also serve as the model for what I want to do using Javassist. Listing 2 shows just the `buildString()` method, with timing added. This doesn't amount to much of a change. The added code just saves the start time to a local variable, then computes the elapsed time at the end of the method and prints it to the console.

Listing 2. Method with timing

```
    private String buildString(int length) {
        long start = System.currentTimeMillis();
        String result = "";
        for (int i = 0; i < length; i++) {
            result += (char)(i%26 + 'a');
        }
        System.out.println("Call to buildString took " +
            (System.currentTimeMillis()-start) + " ms.");
        return result;
    }
```

Doing it with Javassist

Getting the same effect by using Javassist to manipulate the class bytecode seems like it should be easy. Javassist provides ways to add code at the beginning and end of methods, after all, which is exactly what I did in the source code to add timing information for the method.

There's a hitch, though. When I described how Javassist lets you add code, I mentioned that the added code could not reference local variables defined elsewhere in the method. This limitation blocks me from implementing the timing code in Javassist the same way I did in the source code; in that case, I defined a new local variable in the code added at the start and referenced that variable in the code added at the end.

So what other approach can I use to get the same effect? Well, I *could* add a new member field to the class and use that instead of a local variable. That's a smelly kind of solution, though, and suffers from some limitations for general use. Consider what would happen with a recursive method, for instance. Each time the method called itself, the saved start time value from the last call would be overwritten and lost.

Fortunately there's a cleaner solution. I can keep the original method code unchanged and just change the method name, then add a new method using the original name. This *interceptor* method can use the same signature as the original method, including returning the same value. Listing 3 shows what a source code version of this approach would look like:

web_developme...
web_services websphere
wsdl_(web_ser...
xhtml_(extensi... xml
xpath_(xml_pat...
xquery_(xml_qu... xslt
xslt_(xsl_tra...

View as cloud | list

Dig deeper into Java on developerWorks

- Overview
- New to Java programming
- Downloads and products
- Open source projects
- Standards
- Technical library (articles, tutorials, and more)
- Training
- Forums
- Events
- Newsletter

Listing 3. Adding an interceptor method in the source

```
    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 took " +
            (System.currentTimeMillis()-start) + " ms.");
        return result;
    }
```

This approach of using an interceptor method works well with Javassist. Because the entire body of the method is a single block, I can define and use local variables within the body without any problems. Generating the source code for the interception method is also easy -- it only needs a few substitutions to work for any possible method.

Running the interception

Implementing the code to add method timing uses some of the Javassist APIs described in Javassist basics. Listing 4 shows this code, in the form of an application that takes a pair of command-line arguments giving the class name and method name to be timed. The `main()` method body just finds the class information and then passes it to the `addTiming()` method to handle the actual modifications. The `addTiming()` method first renames the existing method by appending `"$impl"` to the end of the name, then creates a copy of the method using the original name. It then replaces the body of the copied method with timing code wrapping a call to the renamed original method.

Listing 4. Adding the interceptor method with Javassist

```
public class JassistTiming
{
    public static void main(String[] argv) {
        if (argv.length == 2) {
            try {

                // start by getting the class file and method
                CtClass clas = ClassPool.getDefault().get(argv[0]);
                if (clas == null) {
                    System.err.println("Class " + argv[0] + " not found");
                } else {

                    // add timing interceptor to the class
                    addTiming(clas, argv[1]);
                    clas.writeFile();
                    System.out.println("Added timing to method " +
                        argv[0] + "." + argv[1]);

                }

            } catch (CannotCompileException ex) {
                ex.printStackTrace();
            } catch (NotFoundException ex) {
                ex.printStackTrace();
            } catch (IOException ex) {
                ex.printStackTrace();
            }

        } else {
            System.out.println("Usage: JassistTiming class method-name");
        }
    }

    private static void addTiming(CtClass clas, String mname)
        throws NotFoundException, CannotCompileException {

        //  get the method information (throws exception if method with
        //  given name is not declared directly by this class, returns
        //  arbitrary choice if more than one with the given name)
        CtMethod mold = clas.getDeclaredMethod(mname);

        //  rename old method to synthetic name, then duplicate the
        //  method with original name for use as interceptor
        String nname = mname+"$impl";
        mold.setName(nname);
        CtMethod mnew = CtNewMethod.copy(mold, mname, clas, null);

        //  start the body text generation by saving the start time
        //  to a local variable, then call the timed method; the
        //  actual code generated needs to depend on whether the
        //  timed method returns a value
        String type = mold.getReturnType().getName();
        StringBuffer body = new StringBuffer();
        body.append("{\nlong start = System.currentTimeMillis();\n");
        if (!"void".equals(type)) {
            body.append(type + " result = ");
        }
        body.append(nname + "($$);\n");

        //  finish body text generation with call to print the timing
        //  information, and return saved value (if not void)
        body.append("System.out.println(\"Call to method " + mname +
            " took \" +\n (System.currentTimeMillis()-start) + " +
            "\" ms.\");\n");
        if (!"void".equals(type)) {
            body.append("return result;\n");
        }
        body.append("}");

        //  replace the body of the interceptor method with generated
        //  code block and add it to class
        mnew.setBody(body.toString());
        clas.addMethod(mnew);

        //  print the generated code block just to show what was done
        System.out.println("Interceptor method body:");
        System.out.println(body.toString());
    }
}
```

The construction of the interceptor method body uses a `java.lang.StringBuffer` to accumulate the body text (showing the proper way to handle `String` construction, as opposed to the approach used in `StringBuilder`). This varies depending on whether the original method returns a value or not. If it *does* return a value, the constructed code saves that value in a local variable so it can be returned at the end of the interceptor method. If the original method is of type `void`, there's nothing to be saved and nothing to be returned from the interceptor method.

The actual body text looks just like standard Java code except for the call to the (renamed) original method. This is the `body.append(nname + "($$);\n");` line in the code, where nname is the modified name for the original method. The `$$` identifier used in the call is the way Javassist represents the list of parameters to the method under construction. By using this identifier in the call to the original method, all the arguments supplied in the call to the interceptor method are passed on to the original method.

Listing 5 shows the results of first running the `StringBuilder` program in unmodified form, then running the `JassistTiming` program to add timing information, and finally running the `StringBuilder` program after it's been modified. You can see how the `StringBuilder` run after modification reports execution times, and how the times increase much faster than the length of the constructed string because of the inefficient string construction code.

Listing 5. 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 javassist.jar:. JassistTiming StringBuilder buildString
Interceptor method body:
{
long start = System.currentTimeMillis();
java.lang.String result = buildString$impl($$);
System.out.println("Call to method buildString took " +
 (System.currentTimeMillis()-start) + " ms.");
return result;
}
Added timing to method StringBuilder.buildString

[dennis]$ java StringBuilder 1000 2000 4000 8000 16000
Call to method buildString took 37 ms.
Constructed string of length 1000
Call to method buildString took 59 ms.
Constructed string of length 2000
Call to method buildString took 181 ms.
Constructed string of length 4000
Call to method buildString took 863 ms.
Constructed string of length 8000
Call to method buildString took 4154 ms.
Constructed string of length 16000
```

Back to top

Trust in the source, Luke?

Javassist does a great job of making classworking easy by letting you work with source code rather than actual bytecode instruction lists. But this ease of use comes with some drawbacks. As I mentioned back in The source of all bytecode, the source code used by Javassist is not exactly the Java language. Besides recognizing special identifiers in the code, Javassist implements much looser compile-time checks on the code than required by the Java language specification. Because of this, it will generate bytecode from the source in ways that may have surprising results if you're not careful.

As an example, Listing 6 shows what happens when I change the type of the local variable used for the method start time in the interceptor code from `long` to `int`. Javassist accepts the source code and converts it into valid bytecode, but the resulting times are garbage. If you tried compiling this assignment directly in a Java program, you'd get a compile error because it violates one of the rules of the Java language: a narrowing assignment requires a cast.

Listing 6. Storing a long in an int

```
[dennis]$ java -cp javassist.jar:. JassistTiming StringBuilder buildString
Interceptor method body:
{
int start = System.currentTimeMillis();
java.lang.String result = buildString$impl($$);
System.out.println("Call to method buildString took " +
 (System.currentTimeMillis()-start) + " ms.");
return result;
}
Added timing to method StringBuilder.buildString
[dennis]$ java StringBuilder 1000 2000 4000 8000 16000
Call to method buildString took 1060856922184 ms.
Constructed string of length 1000
Call to method buildString took 1060856922172 ms.
Constructed string of length 2000
```

```
Call to method buildString took 1060856922382 ms.
Constructed string of length 4000
Call to method buildString took 1060856922809 ms.
Constructed string of length 8000
Call to method buildString took 1060856926253 ms.
Constructed string of length 16000
```

Depending on what you do in the source code, you can even get Javassist to generate invalid bytecode. Listing 7 shows an example of this, where I've patched the `JassistTiming` code to always treat the timed method as returning an `int` value. Javassist again accepts the source code without complaint, but the resulting bytecode fails verification when I try to execute it.

Listing 7. Storing a String in an int

```
[dennis]$ java -cp javassist.jar:. JassistTiming StringBuilder buildString
Interceptor method body:
{
long start = System.currentTimeMillis();
int result = buildString$impl($$);
System.out.println("Call to method buildString took " +
 (System.currentTimeMillis()-start) + " ms.");
return result;
}
Added timing to method StringBuilder.buildString
[dennis]$ java StringBuilder 1000 2000 4000 8000 16000
Exception in thread "main" java.lang.VerifyError:
 (class: StringBuilder, method: buildString signature:
 (I)Ljava/lang/String;) Expecting to find integer on stack
```

This type of issue isn't a problem as long as you're careful with the source code you supply to Javassist. It's important to realize that Javassist won't necessarily catch any errors in the code, though, and that the results of an error may be difficult to predict.

Back to top

Looking ahead

There's a lot more to Javassist than what we've covered in this article. Next month, we'll delve a little deeper with a look at some of the special hooks Javassist provides for bulk modification of classes and for on-the-fly modification as classes are loaded at runtime. These are the features that make Javassist a great tool for implementing aspects in your applications, so make sure you catch the follow-up for the full story on this powerful tool.

Resources

- Javassist was originated by Shigeru Chiba of the Department of Mathematics and Computing Sciences, Tokyo Institute of Technology. It's recently joined the open source JBoss application server project where it's the basis for the addition of new aspect-oriented programming features. Javassist is distributed under the Mozilla Public License (MPL) and the GNU

Lesser General Public License (LGPL) open source licenses.

- Learn more about the Java bytecode design in "Java bytecode: Understanding bytecode makes you a better programmer" (*developerWorks*, July 2001).
- Want to find out more about aspect-oriented programming? Try "Improve modularity with aspect-oriented programming" (*developerWorks*, January 2002) 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.
- Find hundreds more Java technology resources on the *developerWorks* Java technology zone.

About the author

Dennis Sosnoski is the founder and lead consultant of Seattle-area Java consulting company Sosnoski Software Solutions, Inc., specialists in J2EE, XML, and Web services support. His professional software development experience spans over 30 years, with the last several years focused on server-side Java technologies. Dennis is a frequent speaker on XML and Java technologies at conferences nationwide, and chairs the Seattle Java-XML SIG. Contact Dennis at dms@sosnoski.com.

Rate this article

★★★★☆ Average rating (23 votes)

○ ★☆☆☆☆ 1 star
○ ★★☆☆☆ 2 stars
○ ★★★☆☆ 3 stars
○ ★★★★☆ 4 stars
○ ★★★★★ 5 stars

Submit

Comments

Add comment:

Back to top