Create and Read J2SE 5.0 Annotations with the ASM Bytecode Toolkit

by Eugene Kuleshov
10/20/2004

The previous article in this series showed how the ASM toolkit can be used to generate new code, and modify existing classes by adding or changing code. This is suitable for many cases, but there are times when it is necessary to stick some meta-information into the class bytecode and then access it later on. A typical example of such metadata is the annotation facility introduced in
J2SE 5.0.

Bytecode Attributes

Annotations are actually stored in bytecode with several special attributes. The binary format for these and all other standard attributes is described in the Java Virtual Machine (JVM) specification (which has been updated for JDK 1.5). Here is a short outline of attributes supported by ASM's `org.objectweb.asm.attrs` package.

- `EnclosingMethod`
  Used for anonymous or local classes.
- `LocalVariableTypeTable`
  Used by debuggers to determine the value of a given local variable during the execution of a method.

The following attributes have been introduced in the J2SE 5.0 VM.

- `Signature`
  Introduced in JSR-14 ("Adding Generics to the Java Programming Language") and used for classes, fields, and methods to carry generic type information in a backwards-compatible way.
- `SourceDebugExtension`
  Defined in JSR-45 ("Debugging Support for Other Languages") and used for classes only. This attribute allows debuggers keep a reference to the original non-Java source.
- `RuntimeInvisibleAnnotations`, `RuntimeInvisibleParameterAnnotations`, `RuntimeVisibleAnnotations`, `RuntimeVisibleParameterAnnotations`, `AnnotationDefault`
  Used to store annotations as defined in JSR-175 ("A Metadata Facility for the Java Programming Language").

There is also an attribute that hasn't been included in the J2SE 5.0 release but may be added in the future. For now this attribute is used for J2ME MIDlets and is generated by the CLDC preverifier tool.

- `StackMap`
  Contains information for the two-step bytecode verifier used by CDLC; its definition is given in the appendix "CLDC Byte Code Typechecker Specification" in the CLDC 1.1 specification.

All other nonstandard attributes will be ignored by Sun's JVM, although vendors may use proprietary attributes to implement additional features without breaking bytecode compatibility. For example, Microsoft's JVM used the attributes `ActualAccessFlags`, `Hidden`, `LinkUnsafe`, `NAT_L`, and `NAT_L_DCTS` to enable bindings to native libraries without using JNI. These attributes were generated by the MS `jvc` java compiler, based on instructions in its JavaDoc. For example, the code below will create the attribute `NAT_L`:

```
/**
 * @dll.import("SHELL32",auto)
 */
private native static boolean
    ShellExecuteEx(SHELLEXECUTEINFO pshex);
```

Nonstandard attributes could be also used by some application containers to persist proprietary metadata in the bytecode. However, the metadata attributes introduced in Java 5, as mentioned above, eliminate the need for such custom attributes.

Java 5 Annotations Support in ASM

ASM provides a generic API for bytecode attributes. All attributes supported by ASM extend the `Attribute` class and override the `read()` and `write()` methods in order to load and save attribute structures. Concrete attribute classes used to represent annotations are shown in Figure 1. All of them use the `Annotation` data object to store the actual values for annotations.

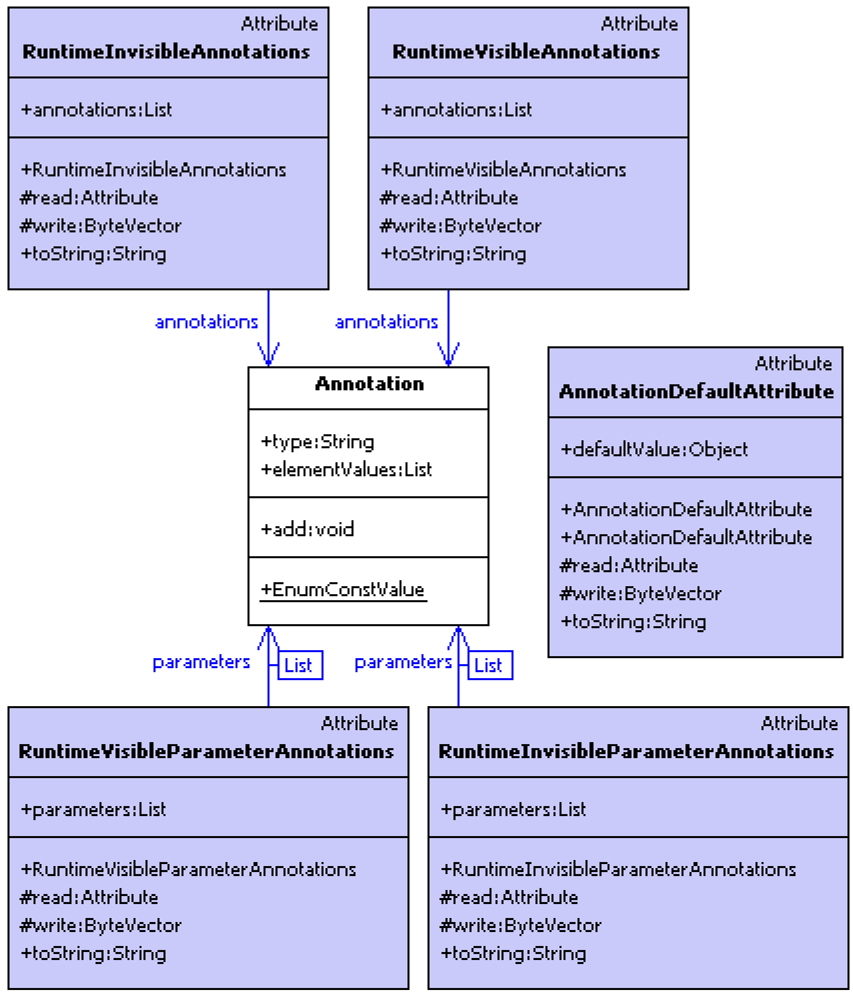

Figure 1. UML class diagram for ASM annotation attributes

`RuntimeVisibleAnnotations` and `RuntimeInvisibleAnnotations` contain `List`s of `Annotation`s.`RuntimeVisibleParameterAnnotations` and `RuntimeInvisibleParameterAnnotations` contain `List`s of `List`s of `Annotation`s in order to handle multiple annotations for each method parameter.

When parsing existing bytecode, `ClassReader` builds concrete attributes from the bytecode data and sends them as parameters to the appropriate `visit...` methods of `ClassVisitor` and `CodeVisitor`. The same events can be generated manually, as we will see later.

- The `ClassVisitor.visitAttribute()` method receives events for class-level attributes, and it is where annotations for class will be passed.
- The `ClassVisitor.visitField()` method receives events for fields, so all field annotations are passed as an `attrs` parameter of this method.

- The `ClassVisitor.visitMethod()` method receives events for every method, so both method and method parameter annotations are passed as an `attrs` parameter of this method.

Notice that the `ClassVisitor.visitAttribute()` and `CodeVisitor.visitAttribute()` methods are called for every attribute. Attributes in `ClassVisitor.visitField()` and `ClassVisitor.visitMethod()` are represented as a linked list.

The Java Virtual Machine specification defines structures and restrictions for all attributes, and I recommend keeping the "Class File Format" chapter handy. However, the `ASMifier` utility can help to implement required transformations with minimal knowledge of bytecode. Let's pick a simple class and apply a custom `Marker` annotation to see how it will be handled by the ASM API. Here's a trivial `Calculator1` class:

```
public class Calculator1 {
  private int result;

  private void sum( int i1, int i2) {
    result = i1 + i2;
  }
}
```

Here is the definition of the `Marker` annotation.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Marker {
  String value();
}
```

And this is an annotated version of `Calculator` class, called `Calculator2`.

```
@Marker("Class")
public class Calculator2 {
  @Marker("Field")
  private int result;

  @Marker("Method")
  private void sum( int i1, @Marker("Parameter") int i2) {
    result = i1 + i2;
  }
}
```

Now we can compile `Calculator1` and `Calculator2` and run `ASMifierClassVisitor` on both compiled classes, and then compare the results to see the ASM API calls required to add annotation attributes into bytecode. The comparison result is below. Red lines represent code without annotations and green lines represent code that has been added to generate annotation attributes in bytecode for the `Calculator2` class.

```
  ...
  ClassWriter cw = new ClassWriter(false);
  CodeVisitor cv;

  cw.visit(V1_5, ACC_PUBLIC + ACC_SUPER,
-     "Calculator1", "java/lang/Object", null,
-     "Calculator1.java");
+     "Calculator2", "java/lang/Object", null,
+     "Calculator2.java");
+ // FIELD ATTRIBUTES
+ RuntimeInvisibleAnnotations fAtt1 =
+       new RuntimeInvisibleAnnotations();
+ Annotation fAtt1ann0 = new Annotation("LMarker;");
+ fAtt1ann0.add( "value", "Field");
+ fAtt1.annotations.add( fAtt1ann0);
  cw.visitField(ACC_PRIVATE, "result", "I", null,
-     null);
+     fAtt1);

  {
  cv = cw.visitMethod(ACC_PUBLIC, "<init>", "()V",
      null, null);
  cv.visitVarInsn(ALOAD, 0);
  cv.visitMethodInsn(INVOKESPECIAL,
      "java/lang/Object", "<init>", "()V");
  cv.visitInsn(RETURN);
  cv.visitMaxs(1, 1);
  }
  {
+ // METHOD ATTRIBUTES
+ RuntimeInvisibleParameterAnnotations mAtt1 =
+       new RuntimeInvisibleParameterAnnotations();
+ List mAtt1p0 = new ArrayList();
+ mAtt1.parameters.add( mAtt1p0);
+
+ List mAtt1p1 = new ArrayList();
+ Annotation mAtt1p1a0 = new Annotation("LMarker;");
+ mAtt1p1a0.add( "value", "Parameter");
+ mAtt1p1.add( mAtt1p1a0);
+ mAtt1.parameters.add( mAtt1p1);
+
+ RuntimeInvisibleAnnotations mAtt2 =
+       new RuntimeInvisibleAnnotations();
+ Annotation mAtt2a0 = new Annotation("LMarker;");
+ mAtt2a0.add( "value", "Method");
+ mAtt2.annotations.add( mAtt2a0);
+
+ mAtt1.next = mAtt2;
+
  cv = cw.visitMethod(ACC_PRIVATE, "sum", "(II)V",
-     null, null);
```

```
+     null, mAtt1);
  cv.visitVarInsn(ALOAD, 0);
  cv.visitVarInsn(ILOAD, 1);
  cv.visitVarInsn(ILOAD, 2);
  cv.visitInsn(IADD);
- cv.visitFieldInsn(PUTFIELD, "Calculator1",
+ cv.visitFieldInsn(PUTFIELD, "Calculator2",
      "result", "I");
  cv.visitInsn(RETURN);
  cv.visitMaxs(3, 3);
  }
+ {
+ // CLASS ATRIBUTE
+ RuntimeInvisibleAnnotations attr =
+     new RuntimeInvisibleAnnotations();
+ Annotation attrann0 = new Annotation("LMarker;");
+ attrann0.add( "value", "Class");
+ attr.annotations.add( attrann0);
+ cw.visitAttribute(attr);
+ }
  cw.visitEnd();
  ...
```

It's common practice to generate or transform classes at runtime using a custom `ClassLoader`. We can also use this technique to add Java 5 annotations. A `ClassLoader` implementation may use the following code to do the required transformation on loaded classes.

```
ClassWriter cw = new ClassWriter(false);
try {
    ClassReader cr =
      new ClassReader(url.openStream());
    cr.accept(new MarkerClassVisitor(cw),
      Attributes.getDefaultAttributes(), false);

    byte[] b = cw.toByteArray();
    return defineClass( name, b, 0, b.length);
} catch( Exception ex) {
    throw new ClassNotFoundException(
        "Unable to load class "+name);
}
```

The actual transformation is done by `MarkerClassVisitor`. It changes the bytecode version in the `visit()` method and adds a class-level `Marker` annotation using the code from the above comparison, before delegating the call to the `visitEnd()` method of the chained `ClassVisitor`.

```
public static class MarkerClassVisitor
    extends ClassAdapter {

  public MarkerClassVisitor(ClassVisitor cv) {
    super(cv);
  }
```

```
  public void visit( int version, int access,
      String name, String superName,
      String[] interfaces, String sourceFile) {
    super.visit(Constants.V1_5, access, name,
        superName, interfaces, sourceFile);
  }

  public void visitEnd() {
    String t = Type.getDescriptor(Marker.class);
    Annotation ann = new Annotation(t);
    ann.add("value", "Class");

    RuntimeVisibleAnnotations attr =
      new RuntimeVisibleAnnotations();
    attr.annotations.add(ann);
    cv.visitAttribute(attr);

    super.visitEnd();
  }

}
```

Below is a simple JUnit test case that uses the Java 5 reflection API to verify that the `Marker` annotation has been created. You can find the complete source code in the Resources section below.

```
public class MarkerClassLoaderTest extends TestCase {

  public void testLoadClass() throws Exception {
    MarkerClassLoader cl =
        new MarkerClassLoader(getClass());
    Class c = cl.loadClass( "asm.Calculator1");
    Annotation a = c.getAnnotation(Marker.class);
    assertNotNull( "Expecting Marker", a);
  }

}
```

Reading J2SE 5.0 Annotations

As shown above, annotations can be generated and accessed from Java 5 code; however, it would be interesting to access these annotations from older JVMs. Let's see how an adapter class, similar to the Java 5 reflection API, could use the ASM toolkit to access this information.

Here is the public part of the `AnnotatedClass` adapter.

```
public class AnnotatedClass {
  private AnnReader r;

  public AnnotatedClass(Class c) {
    try {
      URL u = c.getResource("/"+
          c.getName().replace('.', '/')+".class");
      r = new AnnReader(u.openStream());
    } catch(IOException ex) {
      throw new RuntimeException(ex.toString());
    }
  }

  public AnnotatedClass(InputStream is) {
    try {
      r = new AnnotationReader(is);
    } catch(IOException ex) {
      throw new RuntimeException(ex.toString());
    }
  }

  public Ann[] getAnnotations() {
    List anns = r.getClassAnnotations();
    return (Ann[]) anns.toArray(new Ann[0]);
  }

  ...
}
```

The method `getAnnotations()` substitutes for a new method with the same name in the Java 5 API. However, because`java.lang.annotation.Annotation` class can't be used, our method return the marker interface `Ann`.

```
public static interface Ann {
}
```

Client code that uses the above class would cast the received `Ann` instance into the corresponding interface.

```
Class c =  Calculator2.class;
AnnotatedClass ac = new AnnotatedClass(c);
Ann[] anns = ac.getAnnotations();
if( anns[0] instanceof Marker) {
    String value = ((Marker)anns[0]).value();
```

```
    ...
}
```

The tricky part is that the `Marker` annotation class can't be used directly with older JREs, because its bytecode version is only accepted by Java 5 VM and it contains a few additional flags not recognized by the older JVMs. However, it is easy to transform it on the fly and make it a plain Java interface by comparing the results produced by the `ASMifierClassVisitor` utility or just manually creating and compiling such an interface to be used with old JREs.

Annotation data is loaded by the `AnnReader` class, which extends ASM's `ClassAdapter` and redefines the `visitAttribute()`, `visitField()`, and `visitMethod()` methods.

```
public class AnnReader
      extends ClassAdapter {
  private List classAnns = new ArrayList();
  private Map fieldAnns = new HashMap();
  private Map methodAnns = new HashMap();
  private Map methodParamAnns = new HashMap();


  public AnnReader(InputStream is)
      throws IOException {
    super(null);
    ClassReader r = new ClassReader(is);
    r.accept(this,
        Attributes.getDefaultAttributes(), true);
  }

  public void visitAttribute(Attribute attr) {
    classAnns.addAll(loadAnns(attr));
  }

  public void visitField(int access,
        String name, String desc, Object value,
        Attribute attrs) {
    fieldAnns.put(name+desc, loadAnns(attrs));
  }

  public CodeVisitor visitMethod(int access,
        String name, String desc,
        String[] exceptions, Attribute attrs) {
    methodAnns.put(name+desc, loadAnns(attrs));
    methodParamAnns.put(name+desc,
        loadParamAnns(attrs));
    return null;
  }

  ...
```

The `loadAnns()` and `loadParamAnns()` methods are very straightforward. They just iterate through annotations and collect all values into a `List`, using the `loadAnn()` method. Each element in the `List` would be a dynamic proxy that implements the `Ann` interface and the interface declared by the annotation (e.g., `Marker`).

```
private List loadAnns(Attribute a) {
  List anns = new ArrayList();
  while(a!=null) {
    if(a instanceof
        RuntimeVisibleAnnotations) {
      RuntimeVisibleAnnotations ra =
          (RuntimeVisibleAnnotations) a;
      addAnns(anns, ra.annotations);
    } else if(a instanceof
        RuntimeInvisibleAnnotations) {
      ...
    }
    a = a.next;
  }
  return anns;
}

private List loadParamAnns(Attribute a) {
  List anns = new ArrayList();
  while(a!=null) {
    if(a instanceof
        RuntimeVisibleParameterAnnotations) {
      RuntimeVisibleParameterAnnotations ra =
        (RuntimeVisibleParameterAnnotations) a;
      addParamAnns( anns, ra.parameters);
    } else if(a instanceof
        RuntimeInvisibleParameterAnnotations) {
      ...
    }
    a = a.next;
  }
  return anns;
}

private void addParamAnns( List anns, List params) {
  for(Iterator it = params.iterator(); it.hasNext();) {
    List paramAttrs = (List) it.next();
    List paramAnns = new ArrayList();
    addAnns(paramAnns, paramAttrs);
    anns.add(paramAnns);
  }
}

private void addAnns(List anns, List attr) {
  for(int i = 0; i<attr.size(); i++) {
    anns.add(loadAnn((Annotation) attr.get(i)));
  }
}
```

Method `loadAnn()` is responsible for creating a dynamic proxy from the values retrieved from an `Annotation` object. The proxy is created using `AnnInvocationHandler`, which tries to find a value in the map with the same key as the method name. It is also creates a summary in case `toString()` is called, and throws a `RuntimeException` otherwise.

```
private Object loadAnn(Annotation annotation) {
  String type = annotation.type;
  List vals = annotation.elementValues;
  List nvals = new ArrayList(vals.size());
  for(int i = 0; i < vals.size(); i++) {
    Object[] element = (Object[]) vals.get(i);
    String name = (String) element[0];
    Object value = getValue(element[1]);
    nvals.add(new Object[] { name, value});
  }

  try {
    Type t = Type.getType(type);
    String cname = t.getClassName();
    Class typeClass = Class.forName(cname);
    ClassLoader cl = getClass().getClassLoader();
    return Proxy.newProxyInstance(cl,
        new Class[] { Ann.class, typeClass},
        new AnnInvocationHandler(type, nvals));

  } catch(ClassNotFoundException ex) {
    throw new RuntimeException(ex.toString());

  }
}
```

Finally, the `getValue()` method recursively converts annotation values into Java types. It also wraps nested annotations into dynamic proxies using the `loadAnn()` method.

```
private Object getValue(Object value) {
  if (value instanceof EnumConstValue) {
    // TODO convert to java.lang.Enum adapter
    return value;
  }
  if (value instanceof Type) {
    String cname = ((Type)value).getClassName();
    try {
      return Class.forName(cname);
    } catch(ClassNotFoundException e) {
      throw new RuntimeException(e.toString());
    }
  }
  if (value instanceof Annotation) {
    return loadAnn(((Annotation) value));
  }
  if (value instanceof Object[]) {
    Object[] values = (Object[]) value;
```

```
      Object[] o = new Object[ values.length];
      for(int i = 0; i < values.length; i++) {
        o[ i] = getValue(values[ i]);
      }
      return o;
    }

    return value;
  }
```

In fact, the above code allows you to read annotation data that is not available through the Java 5 reflection API. For example, you can retrieve annotations with `RetentionPolicy.CLASS`.

Conclusion

J2SE 5's annotation facility opens new possibilities for declarative component configuration. Some scenarios may require the dynamic manipulation of annotations in the runtime, and this is provided by the ASM toolkit, which offers complete support for bytecode attributes used to persist Java 5 annotation data. It also allows access to those attributes from older JREs and can even read non-visible annotations at runtime.

Resources

- Source code for this article
- ASM project page
- Java Virtual Machine Specification
- "Revised Class File Format" (Chapter 4 of the JVM specification; PDF). Includes modifications for J2SE 5.0 to support changes mandated by JSR-14, JSR-175, and JSR-201, as well as minor corrections and adjustments.

Eugene Kuleshov is a senior Java developer. He is working for a middleware market and implementing B2B applications on the J2EE platform. He also does research in security and cryptography for the Java platform and XML.

Return to ONJava.com.

Related Articles:

Using the ASM Toolkit for Bytecode Manipulation
ASM is making inroads in the Java bytecode manipulation community--it's used by Groovy, AspectWerkz, BeanShell, and others--because of its light weight and good performance. Eugene Kuleshov shows how to get started with ASM.

Is bytecode manipulation a practical means of programming with attributes?