JVM INTERNALS

Real-World Bytecode Handling with ASM

Scan, inspect, generate, and transform bytecodes on the fly with the ASM library.

by Ben Evans

August 1, 2018

The ASM library is a production-quality open source library for reading, writing, and manipulating JVM bytecode. It is used as part of many projects (including Gradle and the Kotlin compiler) and is used in shaded form (that is, as copied code with renamed packages to avoid namespace collisions) inside the JDK. In fact, it is used as the code-generation engine to enable runtime support of lambda expressions. Note that when you are working with ASM, you should use the external version, not the shaded version present inside the JDK.

In this article, I explain how to use ASM to perform some useful operations. In what follows, I assume that the reader is already familiar with some basics of JVM bytecode and the structure of class files. You can find the code from this article on the *Java Magazine* download page.

A “Hello World” Example

Let’s take a look at a very traditional example, namely creating a class that will print “Hello World!” I will use ASM’s `ClassWriter` API for this exercise. It is a simple API that makes heavy use of the Visitor pattern to achieve its goals.

My example produces a new class file, `HelloWorld.class`, completely from scratch. This class will not have any Java source code representation—that is, it will exist only as a compiled class.

The `HelloWorld.class` file will be created by another class, `MakeHelloWorld`, which will use the ASM libraries to assemble `HelloWorld.class` as output. However, the generated output class will run completely standalone and will not need ASM or any other JAR as a runtime dependency.

Within `MakeHelloWorld`, the overall structure of the class creation is to use a `ClassWriter` field, referred to as `cw`, to build up the class by visiting these aspects of the class in turn:

- Overall metadata
- Constructor body
- Definition of the `main` method and its bytecode

After all aspects of the class have been visited, you can make the writer object ready for serialization by calling `visitEnd()` and then convert it to a byte array that can be written to disk.

In code, this overall driver method looks like the following, and it only needs to be called with the name of the output class:

```
public byte[] serializeToBytes(String outputClazzNar
    cw.visit(V1_8, ACC_PUBLIC + ACC_SUPER, outputCla
        null, "java/lang/Object", null);
    addStandardConstructor();
    addMainMethod();
    cw.visitEnd();
    return cw.toByteArray();
}
```

The serialization method starts by visiting the top-level metadata (class file version, flags, class name, and superclass name) and then calls methods to add a constructor and the main method, before finishing the class and converting it to a frozen byte array.

You create the constructor like this:

```
void addStandardConstructor() {
    MethodVisitor mv =
        cw.visitMethod(ACC_PUBLIC, "<init>", "()V",
    mv.visitVarInsn(ALOAD, 0);
    mv.visitMethodInsn(
        INVOKESPECIAL, "java/lang/Object", "<init>"
    mv.visitInsn(RETURN);
    mv.visitMaxs(1, 1);
    mv.visitEnd();
}
```

This code works with a `MethodVisitor` that is created from the `ClassWriter` field before visiting each instruction in turn. After that, you must finish the method by noting how many stack slots the code uses. You do this by calling `visitMaxs()`.

The main method is added using another `MethodVisitor`:

```
void addMainMethod() {
    MethodVisitor mv =
        cw.visitMethod(ACC_PUBLIC + ACC_STATIC,
            "main", "([Ljava/lang/String;)V", null,
    mv.visitCode();
    mv.visitFieldInsn(GETSTATIC, "java/lang/System"
        "out", "Ljava/io/PrintStream;");
    mv.visitLdcInsn("Hello World!");
    mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/Print
        "println", "(Ljava/lang/String;)V", false);
    mv.visitInsn(RETURN);
    mv.visitMaxs(3, 3);
    mv.visitEnd();
}
```

The code here is a little more complex, because objects need to be retrieved from static fields (via a `GETSTATIC` opcode and then the method must be called).

When I run `MakeHelloWorld`, I see `HelloWorld.class` appear in the file system. I can run the generated class in the usual way—`java HelloWorld`—and when I do, I see the familiar message appear.

The visitor API for ASM is easier to understand than some of the alternative APIs offered by the library. The general principle is that the different sections of the class file must be visited in the correct order (or skipped if there's nothing required for that section). The `MethodVisitor` interface is quite general.

For the case of `MakeHelloWorld`, I've obtained a visitor from the `ClassWriter`, and the actual implementation of the interface is `MethodWriter`. This keeps a reference back to the `ClassWriter` that created it and allows metadata about the method to be built up as the various `visit` methods are called.

The method represented by a `MethodWriter` needs to be sealed up when it is completed, and so `mv.visitEnd()` is called as the final action of the methods that create the methods in `HelloWorld`.

Let's decompile the generated class via `javap -c HelloWorld.class` and look at the bytecode that results from the ASM class generation:

```
public class HelloWorld {
  public HelloWorld();
    Code:
       0: aload_0
       1: invokespecial #8   // Method java/lang/Ob
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #16  // Field
                             //  java/lang/System.ou
       3: ldc           #18  // String Hello World!
       5: invokevirtual #24  // Method
                 //  java/io/PrintStream.println:(Lja
       8: return
}
```

The correspondence between the Java bytecode instructions and the calls to the visitor API is clear, especially in the `main()` method. On the whole, ASM tries to stay very close to the bytecode format, while still providing enough of a high-level API to allow you to be productive.

Let's consider two more examples with a bit more complexity:

- Exploring the "lost update" problem, as illustrated by trying to increment a counter safely
- Exploring a prototype of a "safe class loader" that tries to prevent any user code from executing any native methods

Defeating Lost-Update Protection

I'll start with some simple code to demonstrate the lost-update effect, which I'll describe in a moment. One of the classic ways to introduce the effect is via an incrementing class:

```
public class Counter {
    private int i = 0;

    public int increment() {
        return i = i + 1;
    }
}
```

This class needs a driver:

```
int MAX_INC = 10_000_000;
Counter c = new Counter();
Runnable r = () -> {
    for (int i = 0; i < MAX_INC; i++) {
        c.increment();
    }
};
Thread t1 = new Thread(r);
Thread t2 = new Thread(r);
t1.start();
t2.start();
t1.join();
t2.join();
int disc = 2 * MAX_INC - c.increment() + 1;
System.out.println("Discrepancy: "+ disc);
```

The code is incrementing 10 million times on each thread, so `increment()` is called a total of 20 million times. However, when you run this code, you can clearly see that the code reports a discrepancy—not all calls to `increment()` appear to have been recorded in `c`.

This is the Lost Update antipattern, and even code as simple as `increment()` can exhibit it. This pattern is one of the classic pitfalls of concurrent programming in modern environments.

The lost update is caused by the operating system scheduler running both threads on CPU cores at the same time. Each thread increments the value of `i` as it sees it in the local CPU cache but does not flush the result to main memory. This results in an indeterminate number of updates being performed by both threads before the CPU flushes the cache line to main memory. These cache-only writes are then lost from the overall total being recorded in main memory.

The solution, of course, is to add the `synchronized` keyword to `increment()`, and then the discrepancy is always zero; all updates to i are flushed to main memory before being reread.

To see this, let's start with a synchronized counter and write a tool using ASM that switches off all synchronization in a class. Then, the transformed class will suffer from the lost-update problem even though the original code was safe.

The `OfflineUnsynchronizer` code will operate in the following way:

- Read in the class file using a `ClassReader`.
- Walk through the ASM representation of the class, using a custom `ClassVisitor`.
- Write the Java bytecode back out as a `byte[]`, using a `ClassWriter`.
- Save the bytecode as a transformed class file.

I need to know some details of Java bytecode to carry out the transformation.

For example, in Java bytecode, a `synchronized` method is represented by a flag called `ACC_SYNCHRONIZED` on the method, so I need to remove that flag from any method that I visit. However, to be really sure that all the synchronization is gone, I also need to know that the block form of synchronization is represented slightly differently. If I have some code like this:

```
Object o = ...
synchronized (o) {
    // ...
}
```

It will be turned into a sequence of bytecodes that looks a bit like this:

```
[Sequence that leaves o on top of the stack]
monitorenter

// ...

[Reload o]
monitorexit
```

Both `monitorenter` and `monitorexit` bytecodes consume the top of the stack and lock or unlock the object that they find there. So if these opcodes were replaced with a basic pop, this would strip the synchronization out of any method body that is encountered.

The resulting code is represented by the following two simple classes: an `UnsynchronizingClassVisitor` and an `UnsynchronizingMethodVisitor`, both of which extend ASM framework classes:

```
public class UnsynchronizingClassVisitor extends Cla

    public UnsynchronizingClassVisitor(int api, Clas
        super(Opcodes.ASM5, cv);
    }

    @Override
    public MethodVisitor visitMethod(int flags, Stri
      String desc, String signature, String[] excep
        int maskedFlags = flags & (~ACC_SYNCHRONIZEI

        MethodVisitor baseMethodVisitor =
            super.visitMethod(maskedFlags, name, des
                              signature, exceptions

        return new UnsynchronizingMethodVisitor(base
    }
}
```

The `UnsynchronizingClassVisitor` class uses a Decorator pattern: it takes the `baseMethodVisitor` and wraps it by adding functionality that is called only when a no-argument opcode is encountered in the body of the method, as shown in this code:

```
public class UnsynchronizingMethodVisitor extends Me
    public UnsynchronizingMethodVisitor(MethodVisito
        super(Opcodes.ASM5, mv);
    }

    @Override
    public void visitInsn(final int opcode) {
        switch (opcode) {
            case Opcodes.MONITORENTER:
            case Opcodes.MONITOREXIT:
                super.visitInsn(Opcodes.POP);
                return;
        }

        super.visitInsn(opcode);
    }
}
```

I use the following bit of code to drive this transformation:

```
try (InputStream in =
        Files.newInputStream(Paths.get(fName))) {
    ClassReader classReader = new ClassReader(in);
    ClassWriter writer =
       new ClassWriter(classReader, ClassWriter.COMI

    ClassVisitor unsynchronizer =
       new UnsynchronizingClassVisitor(writer);
    classReader.accept(unsynchronizer,
       ClassReader.SKIP_FRAMES | ClassReader.SKIP_DI

   Path newClazz = Paths.get(transformName(fName));
   Files.write(newClazz, writer.toByteArray());
} catch (Exception ex) {
    System.err.println(
        "Exception whilst reading class: " + fName)
    ex.printStackTrace(System.err);
}
```

Now, if I take a synchronized version of the `Counter` class, I can run it through the `OfflineUnsynchronizer`, and the resulting transformed class will suffer the lost-update problem even though the original code was safe.

Ruling Out Native Code

Java bytecode is platform-independent, so it cannot call operating system libraries directly (for example, to handle I/O operations). Instead, Java programs (including the JDK) call out to native methods (written in C) that in turn call the relevant parts of the operating system.

Suppose you have a use case where you want to allow users to execute unknown code as part of a framework or container. Such a capability has obvious security concerns, so you might want to reduce the risk by disallowing certain actions—such as running native methods—in the users' classes. Fortunately, the Java security model relies on class loading, and it allows you to hook into the loading process to customize how (and whether) new code is loaded. [For more on how class loading works, see the article "How the JVM Locates, Loads, and Runs Libraries" by Oleg Selajev, which you can download as a PDF. —*Ed.*]

The overall scheme could look like this:

- Write a custom class loader.
- During class loading, inspect every "call site" where a method is called.
- Check to see whether the metadata for the method indicates that the method is native.
- If it is, reject the class and fail class loading.
- If you reach the end without failing, the class is good and can be loaded.

Here's how to write a class loader that will reject any non-pure Java classes it is asked to load:

```
public final class PureJavaClassLoader extends Class
    private final List<String> auxClasspath = new A

    public PureJavaClassLoader(ClassLoader parent) 
        super(parent);
    }

    public void setupClasspath(final String auxilia
        for (String entry : auxiliaryClassPath.spli
            if (entry.startsWith("/")) {
                auxClasspath.add(entry);
            } else {
                System.err.println(
                    "Bad classpath entry seen: " +
                     entry + ", ignoring");
            }
        }
    }

    Path findClassFile(String qualifiedClassName)th
        final String fileName =
            qualifiedClassName.replaceAll("/", "\\.
        for (String s : auxClasspath) {
            Path trial = Paths.get(s, fileName);
            if (trial.toFile().exists())
                return trial;
        }

        throw new IOException("Class "+ qualifiedCl
                              " not found on classp
    }
}
```

For simplicity, I'll manage an *auxiliary class path* of directories that I want to search for classes to load, rather than using the main class path. The `findClassFile` method is a helper that locates the file corresponding to a qualified class name. The real action is in the `findClass` method to which class loaders delegate from `loadClass()`. This is where I implement the check for native code:

```
@Override
public Class<?> findClass(final String qualifiedCla
  ClassNotFoundException {

    Class<?> cls = null;
    try {
```

```
            return super.findClass(qualifiedClassName);
        } catch (ClassNotFoundException ignored) {
            try (final InputStream in = Files.newInputS
                    findClassFile(qualifiedClassName)))
                        final byte[] allClassBytes = in
                        final ClassReader classReader =
                            new ClassReader(allClassByt
                        final PureJavaCheckingClassVisi
                            classVisitor =
                              new PureJavaCheckingClass

                    // If there's debug info in the cla
                    // don't look at it
                        classReader.accept(
                            classVisitor, ClassReader.S

                        if (classVisitor.containsNative
                            throw new ClassNotFoundExce
                           "Class cannot be loaded - co
                        } else {
                            return defineClass(null, al
                                                allClass
                        }
            } catch (IOException e) {
                    throw new ClassNotFoundException(
                        "Error finding and opening clas
            }
        }
    }
```

In the previous code, I call `in.readAllBytes()` directly, rather than passing `in` to the `ClassReader` constructor. This is because the ASM class `ClassReader` consumes input streams, so I can't reuse `in` after it's been used to create a class reader.

Next, I create an instance of our custom class visitor, `PureJavaCheckingClassVisitor`. This visitor simply visits the metadata for each method in the class being considered and records whether any method is native. It is defined as the following:

```
public class PureJavaCheckingClassVisitor extends Cl
    private boolean containsNative = false;

    public PureJavaCheckingClassVisitor() {
        super(Opcodes.ASM5);
    }

    @Override
    public MethodVisitor visitMethod(int flags, Str
      String desc, String signature, String[] excep
        if ((flags & ACC_NATIVE) > 0) {
            containsNative = true;
        }

        return new MethodVisitor(Opcodes.ASM5) {};
    }

    public boolean containsNative() {
        return containsNative;
    }
}
```

If the class visitor ever sees a native method, it sets a flag. The flag is read by the `PureJava-ClassLoader`, which rejects the class with a `ClassNotFoundException` if the flag has been set. This exception is used, rather than the alternative natural choice (`SecurityException`), because the contract of `ClassLoader` (which is the supertype of this class) uses the checked exception `ClassNotFoundException`. In this circumstance, use of a runtime exception (such as `Security-Exception`) could violate some expectations of clients of the classloader.

Assuming that an exception has not been thrown, the bytes of the class file are fed to `defineClass()`, which is a protected method defined on `ClassLoader` so it is accessible only to subclasses—effectively custom class loaders. This returns the `Class<?>` object that I return from `findClass()`, and the class is successfully loaded.

Conclusion

A word of caution: the previous example will indeed prevent any classes with native methods from being loaded. However, in a real environment, you would also have to take into account other cases, such as the following:

- Code that calls a native method of an already-loaded class (the transitive case)
- Reflective access to native methods
- Invocation of native methods via the `MethodHandles` interface

Not only that, but some native methods are essential for proper functioning of virtually all Java programs (such as `getClass()` or `Object::hashCode`).

A full discussion of what would be required to fully restrict native code from running is too far afield for this article. In practice, some sort of approved list of core native methods within the JDK would have to be used. Nevertheless, note the things I did with ASM in the example: I read through bytecodes for a given release of Java, skipped over debugging data, and identified specific bytecodes. And earlier, I transformed bytecodes on the fly.

Ben Evans

Ben Evans (@kittylyst) is a Java Champion, a tech fellow and founder at jClarity, an organizer for the London Java Community (LJC), and a member of the Java SE/EE Executive Committee. He has written four books on programming, including the recent *Optimizing Java* (O'Reilly).