Bytecode Manipulation in the Real World

Geert Bevin
Terracotta Inc. - http://terracotta.org
Who am I?

• developer at Terracotta - http://terracotta.org
• founder of Uwyn - http://uwyn.com
• founder of RIFE - http://rifers.org
• contributor to many open-source projects: Terracotta, RIFE, OpenLaszlo, Gentoo, ...
• Sun Java Champion
• creator of native Java language continuations
• biker and gamer
Agenda

• What is bytecode manipulation?
• Some popular projects using it
• Don't be afraid
• Plug in the manipulation
• Best practices
• Q & A
Agenda

• What is bytecode manipulation?
• Some popular projects using it
• Don't be afraid
• Plug in the manipulation
• Best practices
• Q & A
What is bytecode manipulation?

• Tools can read bytecode without loading classes (FindBugs)
  – Bytecode inspection

• Languages compile to class files with bytecode (Java, Groovy, Scala, ...)
  – Bytecode generation

• You can modify bytecode just as you can edit other files (AspectJ)
  – Bytecode modification

▷ All this is bytecode manipulation
Bytecode is not assembler

• I used to think that bytecode == assembler
• JVM makes working with bytecode easy
• JDK ships with tools to look at bytecode
• Bytecode is a lot like Java
Let's take a quick look

Hello World
Quick Look

source code HelloWorld.java

```java
public class HelloWorld {
    public HelloWorld() {
    }

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

javac -g:none HelloWorld.java
javap -p -v HelloWorld

```java
public class HelloWorld extends java.lang.Object {
    public HelloWorld();
    Code:
        Stack=1, Locals=1, Args_size=1
        0: aload_0: this
        1: invokespecial Object.<init>() : void
        4: return
    public static void main(java.lang.String[]);
    Code:
        Stack=2, Locals=1, Args_size=1
        0: getstatic System.out : PrintStream
        3: ldc "Hello World!"
        5: invokevirtual PrintStream.println(String) : void
        8: return
}
```
Agenda

• What is bytecode manipulation?
• Some popular projects using it
• Don't be afraid
• Plug in the manipulation
• Best practices
• Q & A
Agenda

• What is bytecode manipulation?
• Some popular projects using it
• Don't be afraid
• Plug in the manipulation
• Best practices
• Q & A
Persistence

• Hibernate
  – track field changes
  – lazily load many-to-many and many-to-one associations
  – optimize reflection performance
  – ...

Inversion of Control (IoC)

• Spring
  – read class meta data without loading it
  – read annotations without loading classes
  – scan the classpath with inclusion filters
  – discover parameter names for auto wiring
  – …
Aspect Oriented Programming

• AspectJ / Spring
  – generate proxies if no interfaces are present for dynamic proxies
  – look for pointcuts where AOP advices need to be applied
  – weave advices into the target classes
  – ...

Geert Bevin

Bytecode Manipulation in the Real World

http://terracotta.org
Static code analysis

• FindBugs
  – check for misunderstood API methods
  – check for common typos
  – check for bad practices
  – check for multi-threaded correctness
  – check for performance problems
  – ...
Clustering

• Terracotta
  – cluster several JVMs to create one super JVM out of multiple nodes
  – opcodes like GETFIELD, PUTFIELD, MONITORENTER and MONITOREXIT become coordinated over the cluster
  – inspect and instrument code that works with cluster-wide data structures
  – replace and enhance library classes to be better citizens in a clustered application
  – ...
Conclusion
You're probably already using bytecode manipulation all over the place
Agenda

• What is bytecode manipulation?
• Some popular projects using it
• Don't be afraid
• Plug in the manipulation
• Best practices
• Q & A
Agenda

• What is bytecode manipulation?
• Some popular projects using it
• Don't be afraid
• Plug in the manipulation
• Best practices
• Q & A
The JVM has a verifier
Why is there a verifier?

- Java compiler produces valid bytecode
- JVM can't assume that the bytecode is valid because it could have been modified
  - it could perform dangerous operations
  - crash your applications
  - even crash the JVM
- As a protection the JVM verifies the bytecode when a class is loaded
- Verification also improves performance since less runtime checks are needed
What is verified?

• The verification process is very extensive

• Notable steps:
  – Check format of the class file to ensure that the bytes are valid and well structured
  – Check coherence of the class structure
    • presence of superclass
    • ensure 'final' is honored
    • valid constant pool
    • valid field and method references
What is verified?

– Code inspection with data-flow analysis ensures the following:
  • operand stack is always the same size and contains the same types of values
  • local variables only accessed while containing a value of an appropriate type
  • methods invocations have appropriate arguments
  • fields assignment have appropriate types
  • opcodes have appropriate type arguments on the operand stack and in the local variable array
  • much more ...
Libraries hide the hard stuff
Libraries hide the hard stuff

• Libraries are there to make your life easy
  – ASM :
    http://asm.objectweb.org/
  – BCEL :
    http://jakarta.apache.org/bcel/
  – CGlib :
    http://cglib.sourceforge.net/
  – Javassist :
    http://www.csg.is.titech.ac.jp/~chiba/javassist/
Libraries hide the hard stuff

• My preference goes to ASM
  – Very lightweight
  – Low memory usage: visitor pattern
  – Abstracts enough to make your life easy
  – Close enough to the basics to educate you
  – High-level utilities for common operations
  – Tree-based API for non-linear manipulations
  – Increasingly popular
  – Great documentation
  – Helpful and passionate community
Example of using ASM

Generating Hello World
HelloWorld in Java with ASM

```java
ClassWriter cw = new ClassWriter(0); MethodVisitor mv;

cw.visit(V_1_5, ACC_PUBLIC + ACC_SUPER, "HelloWorld", null,
        "java/lang/Object", null);

mv = cw.visitMethod(ACC_PUBLIC, "<init>", "()V", null, null);
mv.visitCode();
mv.visitVarInsn(ALOAD, 0);
mv.visitMethodInsn(INVOKEPECIAL, "java/lang/Object", "<init>", "()V");
mv.visitInsn(RETURN);
mv.visitMaxs(1, 1);
mv.visitEnd();

mv = cw.visitMethod(ACC_PUBLIC + ACC_STATIC, "main",
        "([Ljava/lang/String;)V", null, null);
mv.visitCode();
mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out",
        "Ljava/io/PrintStream;");
mv.visitLdcInsn("Hello World!");
mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println",
        "(Ljava/lang/String;)V");
mv.visitInsn(RETURN);
mv.visitMaxs(2, 1);
mv.visitEnd();

cw.visitEnd();
```
Modifying a class with ASM

Logging method invocations
Method invocation logging with ASM

• Detect the beginning of each method
• Modify any existing class file
• Insert print statement before any of the method's code
  `System.out.println(class + "." + method + desc)`
• This statically changes the class so that this functionality is permanently added
public class MethodLoggerAdapter extends ClassAdapter implements Opcodes {
    private String classname;

    public MethodLoggerAdapter(ClassVisitor cv) { super(cv); }

    public void visit(int ver, int access, String name, String sig, String superName, String[] interfaces) {
        this.classname = name;
        super.visit(ver, access, name, sig, superName, interfaces);
    }

    public MethodVisitor visitMethod(int acc, final String name, final String desc, String sig, String[] ex) {
        return new MethodAdapter(super.visitMethod(acc, name, desc, sig, ex)) {
            public void visitCode() {
                mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;"));
                mv.visitLdcInsn(classname + "." + name + desc);
                mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V"));
                super.visitCode();
            }
        };
    }
}
Tools make your life easy
JDK's javap command

• Class file disassembler
• By default print out the structure of a class based on accessibility modifiers (package, protected, public, private)
• Using the verbose option, the bytecode of methods is displayed
ASM tools

• **TraceClassVisitor**
  – Prints out a readable trace of the bytecode that is actually generated
  – Hooked up in your code's visitor chain

• **CheckClassAdapter**
  – Detect errors that wouldn't pass the verifier
  – Hooked up in your code's visitor chain

• **ASMifierClassVisitor**
  – Prints the Java source code to generate a particular class with ASM
  – Used from the command line
Eclipse Bytecode Outline Plug-in
Eclipse Bytecode Outline Plug-in

• Automatically convert Java source code to readable bytecode
• Alternatively convert to ASM API instructions that generate the bytecode
• Bytecode analysis pane that shows local variable array and operand stack
• Contains a bytecode reference for each instruction
Jad - fast Java decompiler

• Generates Java source from bytecode
• Handy when looking at classes whose bytecode has already been manipulated
• Used on the command line, for example: jad HelloWorld.class
• Generates HelloWorld.jad
public class HelloWorld {
    public HelloWorld() {
    }

    public static void main(String args[]) {
        System.out.println("Hello World!");
    }
}
Agenda

• What is bytecode manipulation?
• Some popular projects using it
• Don't be afraid
• Plug in the manipulation
• Best practices
• Q & A
Agenda

• What is bytecode manipulation?
• Some popular projects using it
• Don't be afraid
• Plug in the manipulation
• Best practices
• Q & A
Different options

• Compiler
• Static transformer
• JDK instrumentation agent
• Classloader
Compiler

• Generate class files by parsing source
• Used by Java, Groovy, Scala, ...
• Unless you're writing your own language you'll probably never use this option
• Don't de facto dismiss it though
• For example, RIFE's template engine uses a simple parser and writing a compiler for it was trivial
Static transformer (pre-processor)

- Class files can be read as byte arrays
- This can be provided as an argument to ASM's visitors
- Resulting in another byte array
- Can be written to another file

- This can be easily integrated with an existing build process
JDK instrumentation agent

- Available since JDK 1.5
- Can register transformers that are called when classes are defined or re-defined
- Class' byte array is provided by JDK
- Cleanly separates the class definition phase from the class loading phase
- Activated through JVM command-line option `-javaagent`
- JDK 1.6 adds support for self-activating agents through jar manifest option
Classloader

• Last resort when:
  – JDK 1.4 support is needed
  – JDK 1.5 support is needed and JVM command-line can't be modified
  – manipulation requires state of running system

• Writing your own classloader is difficult

• Have to respect the classloader hierarchy

• Classpath has to be correctly setup

• Difficult to debug class compatibility issues

• Not trivial to obtain byte arrays quickly
Agenda

• What is bytecode manipulation?
• Some popular projects using it
• Don't be afraid
• Plug in the manipulation
• Best practices
• Q & A
Agenda

• What is bytecode manipulation?
• Some popular projects using it
• Don't be afraid
• Plug in the manipulation
• Best practices
• Q & A
The trivial ones
Trivial best practices

• Use the available tools and libraries
• Prefer agents over classloaders
• Start from Java to know which opcodes you need, don't write them from scratch
Document the purpose of your opcodes
Document the purpose of your opcodes

- Even more important than Java source code comments since there's a lot of noise
- Structure blocks in logical sections
- Generating opcodes is easy, understanding what really goes on after the fact is much harder
- Non trivial instrumentations require a lot of effort to comprehend without context
Document the purpose of your opcodes

• For example, look at the following trivial snippet and imagine something difficult:

```java
int hashEntryLocal = newLocal(Type.getObjectType("java/util/concurrent/ConcurrentHashMap$HashEntry"));
mv.visitVarInsn(ALOAD, hashEntryLocal);
Label labelStart = new Label();
Label labelEnd = new Label();
Label labelFinally = new Label();
mv.visitTryCatchBlock(labelStart, labelEnd, labelFinally, null);
mv.visitVarInsn(ALOAD, hashEntryLocal);
super.visitInsn(opcode, owner, name, desc);
int valueLocal = newLocal(Type.getObjectType("java/lang/Object"));
mv.visitInsn(ASTORE, valueLocal);
```
/ store the hash entry in a local variable
int hashEntryLocal = newLocal(Type.getObjectType(
    "java/util/concurrent/ConcurrentHashMap$HashEntry"));
mv.visitVarInsn(ASTORE, hashEntryLocal);

// setup try catch with finally to release the monitor
// on the hash entry that will be entered below
Label labelStart = new Label();
Label labelEnd = new Label();
Label labelFinally = new Label();
mv.visitTryCatchBlock(labelStart, labelEnd, labelFinally, null);

// enter a monitor on the hash entry
mv.visitVarInsn(ALOAD, hashEntryLocal);
mv.visitMethodInsn(MONITORENTER);

// start try catch block
mv.visitLabel(labelStart);

// obtain the value of the entry
mv.visitVarInsn(ALOAD, hashEntryLocal);
super.visitFieldInsn(opcode, owner, name, desc);
int valueLocal = newLocal(Type.getObjectType("java/lang/Object"));
mv.visitVarInsn(ASTORE, valueLocal);
Prefer visitor API over tree API
Prefer visitor API over tree API

• Compare to XML
  – visitor API == SAX
    • event driven, no in memory tree
    • fast
    • less memory
  – tree API == DOM
    • in memory tree of the entire class
    • can be easier when doing class transformations
public interface ClassVisitor {

    void visit(int version, int access, String name, String sig, String superName, String[] interfaces);

    void visitSource(String source, String debug);

    void visitOuterClass(String owner, String name, String desc);

    AnnotationVisitor visitAnnotation(String desc, boolean visible);

    void visitAttribute(Attribute attr);

    void visitInnerClass(String name, String outerName, String innerName, int access);

    FieldVisitor visitField(int access, String name, String desc, String sig, Object value);

    MethodVisitor visitMethod(int access, String name, String desc, String sig, String[] exceptions);

    void visitEnd();
}

Visitor API excerpt
public class ClassNode {
    public int version;
    public int access;
    public String name;
    public String signature;
    public String superName;
    public List<String> interfaces;
    public String sourceFile;
    public String sourceDebug;
    public String outerClass;
    public String outerMethod;
    public String outerMethodDesc;
    public List<AnnotationNode> visibleAnnotations;
    public List<AnnotationNode> invisibleAnnotations;
    public List<Attribute> attrs;
    public List<InnerClassNode> innerClasses;
    public List<FieldNode> fields;
    public List<MethodNode> methods;
}

Tree API excerpt
Use LocalVariablesSorter
ASM method adapter
LocalVariableSorter method adapter

• In bytecode, local variables are referred to as indexes in an array
• All opcodes within the same method use those numeric indexes
• If you introduce local variables, all later ones need to be renumbered everywhere

» LocalVariableSorter does this for you
Prefer delegating to real Java code
Prefer delegating to real Java code

• Don't forget that you can call your own classes and methods from within bytecode

• In-lining functionality as opcodes is less readable and less maintainable

• For example, in the method logging example consider this helper method:

```java
public static class ByteCodeUtil {
    public static void systemOutPrintln(MethodVisitor mv, String msg) {
        mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");
        mv.visitLdcInsn(msg);
        mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V");
    }
}
```
Prefer delegating to real Java code

• Now the adapter can be simplified to:

```java
public class MethodLoggerAdapter extends ClassAdapter implements Opcodes {
    private String classname;

    public MethodLoggerAdapter(ClassVisitor cv) { super(cv); }

    public void visit(int ver, int access, String name, String sig, String superName, String[] interfaces) {
        this.classname = name;
        super.visit(ver, access, name, sig, superName, interfaces);
    }

    public MethodVisitor visitMethod(int acc, final String name, final String desc, String sig, String[] ex) {
        return new MethodAdapter(super.visitMethod(acc, name, desc, sig, ex)) {
            @Override
            public void visitCode() {
                ByteCodeUtil.systemOutPrintln(mv, classname + "." + name + desc);
                super.visitCode();
            }
        };
    }
}
```

Geert Bevin
Don't copy and modify, inspect and instrument
Don't copy and modify

• Let's say that you want to modify a method of an existing library class
• The wrong approach is:
  – Generate the ASM calls from existing method
  – Modify the calls
  – Replace the entire method
• Why?
  – Licensing and copyright
  – Not resilient to class changes
  – Very unmaintainable
Inspect and instrument

• The correct approach is:
  – Generate the ASM calls from existing method
  – Look for patterns that need modification
  – Write the ASM visitors to check for these
  – Insert or omit bytecode when the patterns are detected

• Advantages:
  – You copy none of the existing code
  – You instrument one particular behavior
  – The rest of the original method can change
Remove unnecessary opcodes
Remove unnecessary opcodes

• Using the Eclipse plugin or the ASMifierClassVisitor, it's very easy to obtain bytecode generation instructions
• Read through the results carefully
• Some opcodes might have to be removed
• For example:
  – line numbers
  – unnecessary labels
  – local variable table
  – max stack and max locals
Remove unnecessary opcodes

• Imagine the bytecode of this code has to be added to an existing method:

```java
StringBuilder msg = new StringBuilder(
    new Date().toString());
msg.append(" : Hello World!");
System.out.println(msg);
```
This is the generated bytecode
L0
 LINENUMBER 8 L0
 NEW StringBuilder
 DUP
 NEW Date
 DUP
 INVOKEESPECIAL Date.<init>() : void
 INVOKEVIRTUAL Date.toString() : String
 INVOKEESPECIAL StringBuilder.<init>(String) : void
 ASTORE 1
 L1
 LINENUMBER 9 L1
 ALOAD 1: msg
 LDC " : Hello World!"
 INVOKEVIRTUAL StringBuilder.append(String) : StringBuilder
 POP
 L2
 LINENUMBER 10 L2
 GETSTATIC System.out : PrintStream
 ALOAD 1: msg
 INVOKEVIRTUAL PrintStream.println(Object) : void
 L3
 LINENUMBER 11 L3
 RETURN
 L4
 LOCALVARIABLE args String[] L0 L4 0
 LOCALVARIABLE msg StringBuilder L1 L4 1
 MAXSTACK = 4
 MAXLOCALS = 2
This is what you should remove
L0
LINENUMBER 8 L0
NEW StringBuilder
DUP
NEW Date
DUP
INVOKESPECIAL Date.<init>() : void
INVOKEVIRTUAL Date.toString() : String
INVOKESPECIAL StringBuilder.<init>(String) : void
ASTORE 1

L1
LINENUMBER 9 L1
ALOAD 1: msg
LDC " : Hello World!"
INVOKEVIRTUAL StringBuilder.append(String) : StringBuilder
POP

L2
LINENUMBER 10 L2
GETSTATIC System.out : PrintStream
ALOAD 1: msg
INVOKEVIRTUAL PrintStream.println(Object) : void

L3
LINENUMBER 11 L3
RETURN

L4
LOCALVARIABLE args String[] L0 L4 0
LOCALVARIABLE msg StringBuilder L1 L4 1
MAXSTACK = 4
MAXLOCALS = 2
Modularize by chaining visitors
Modularize by chaining visitors

• Do not write one big visitor with every manipulation
• The visitor pattern is perfect for delegation (for example, ClassAdapter ASM class)
• Create independent visitors for single, well-defined tasks
• Chain them together

▶ Easier to comprehend, easier to maintain and often even easier to implement
Agenda

• What is bytecode manipulation?
• Some popular projects using it
• Don't be afraid
• Plug in the manipulation
• Best practices
• Q & A
Q & A

Cluster your JVM
http://terracotta.org