 AOP 的概念已经不是什么新鲜事物，所以我在这里就不在介绍 Aop 的概念。目前市面上要做到 Aop 是一件十分简单的事情。Spring、AspectJ、CGLib等等都可以帮助你达到目的，但是它们也只不过是一些泛生品。

    上面提到了一些开源的 Aop 实现技术选型，但是我敢说无论你尝试使用上面哪种技术选型都没有我将要介绍的这种方式的运行效率最高。不过读者不要高兴的太早，读完本文想必你就知道是什么原因了。

    介绍一款工具ASM，下面是(<http://www.ibm.com/developerworks/cn/java/j-lo-asm30/>)内容的一个节选。

    ASM 是一个 Java 字节码操控框架。它能被用来动态生成类或者增强既有类的功能。ASM 可以直接产生二进制 class 文件，也可以在类被加载入 Java 虚拟机之前动态改变类行为。Java class 被存储在严格格式定义的 .class 文件里，这些类文件拥有足够的元数据来解析类中的所有元素：类名称、方法、属性以及 Java 字节码（指令）。ASM 从类文件中读入信息后，能够改变类行为，分析类信息，甚至能够根据用户要求生成新类。

    可以负责任的告诉大家，ASM只不过是通过 “Visitor” 模式将 “.class” 类文件的内容从头到尾扫描一遍。因此如果你抱着任何更苛刻的要求最后都将失望而归。上面我们介绍的那些 Aop 框架它们几乎都属于 ASM 框架的泛生品。

    众所周知，Aop 无论概念有多么深奥。它无非就是一个“Propxy模式”。被代理的方法在调用前后作为代理程序可以做一些预先和后续的操作。这一点想必读者都能达到一个共识。因此要想实现 Aop 的关键是，如何将我们的代码安插到被调用方法的相应位置。

    而要追求 Aop 最快的效率的方法也正式将我们要执行的代码直接安插到相应的位置。先看一段最简单的代码。

public class TestBean {

public void halloAop() {

System.out.println("Hello Aop");

}

}

    接下来我要为 halloAop 这个方法加装一个Aop，使在它之前和之后各打印一段字符串。最后的代码执行起来看上去应该是这个样子的：

public class TestBean {

public void halloAop() {

System.out.println("before");

System.out.println("Hello Aop");

System.out.println("after");

}

}

    首先使用 javac 上面类，然后通过javap -c 查看它们的代码：

$ javap -c TestBean

Compiled from "TestBean.java"

public class org.more.test.asm.TestBean extends java.lang.Object{

public org.more.test.asm.TestBean();

Code:

0: aload\_0

1: invokespecial #8; //Method java/lang/Object."<init>":()V

4: return

public void halloAop();

Code:

0: getstatic #15; //Field java/lang/System.out:Ljava/io/PrintStream;

3: ldc #21; //String Hello Aop

5: invokevirtual #23; //Method java/io/PrintStream.println:(Ljava/lang/String;)V

8: return

}

    第 04 行：这一行开始到第 08 行 表示的是一个的默认构造方法，虽然我们没有在 TestBean 类中编写任何构造方法，但是作为 Java 类都应当有一个默认的无参构造方法，而这个构造方法是编译器为我们自动添加的。  
    第 10 行：从这行开始到结束就是我们编写的 halloAop 方法，下面将会介绍一下上面出现的几个字节码指令。

    下面如果我们可以在上面字节码的12行和15行动态的插入代码那么我们的AOP目的就达到了。

        在介绍指令之前我先简单说明一下 JVM 的运行机制。首先可以简单的将 JVM 虚拟机看作是一个 CPU。作为 CPU 都要有一个入口程序。在我们的电脑中主板的 Bioss 程序就是充当这个角色，而在JVM 中 Main方法来充当这一角色。CPU 在运行程序的时会将程序数据放入它的几个固定存储器，我们称它们为寄存器。CPU 对数据的所有计算都针对寄存器。而 JVM 并不具备这一特征，它采用的是堆结构。   
        比方说计算 “a + b”，在 CPU 中需要两个寄存器。首先将“1”载入第一个寄存器，其次将另外一个“1”载入第二个寄存器，然后调用相应的加法指令将两个寄存器中的数据相加。相加的结果会保存在另外一个寄存器上。而在 JVM 中首先将第一个“1”push到堆栈中，其次在将另外一个“1”push到堆栈中，紧接着调用ADD指令。这个指令会取出这两个数字相加然后将结果再次放入堆栈中。经过运算之后堆栈中的两个“1”已经不存在了，在堆栈顶端有一个新的值“2”。JVM 所有计算都是在此基础之上完成的。

    在 Java 中每一个方法在执行的时候 JVM 都会为其分配一个“帧”，帧是用来存储方法中计算所需要的所有数据。其中第 0 个元素就是 “this”，如果方法有参数传入会排在它的后面。

ALOAD\_0：  
    这个指令是LOAD系列指令中的一个，它的意思表示装载当前第 0 个元素到堆栈中。代码上相当于“this”。而这个数据元素的类型是一个引用类型。这些指令包含了：ALOAD，ILOAD，LLOAD，FLOAD，DLOAD。区分它们的作用就是针对不用数据类型而准备的LOAD指令，此外还有专门负责处理数组的指令 SALOAD。

invokespecial：  
    这个指令是调用系列指令中的一个。其目的是调用对象类的方法。后面需要给上父类的方法完整签名。“#8”的意思是 .class 文件常量表中第8个元素。值为：“java/lang/Object."<init>":()V”。结合ALOAD\_0。这两个指令可以翻译为：“super()”。其含义是调用自己的父类构造方法。

GETSTATIC：  
    这个指令是GET系列指令中的一个其作用是获取静态字段内容到堆栈中。这一系列指令包括了：GETFIELD、GETSTATIC。它们分别用于获取动态字段和静态字段。

IDC：  
    这个指令的功能是从常量表中装载一个数据到堆栈中。

invokevirtual：  
    也是一种调用指令，这个指令区别与 invokespecial 的是它是根据引用调用对象类的方法。这里有一篇文章专门讲解这两个指令：“<http://wensiqun.iteye.com/blog/1125503>”。

RETURN：  
    这也是一系列指令中的一个，其目的是方法调用完毕返回：可用的其他指令有：IRETURN，DRETURN，ARETURN等，用于表示不同类型参数的返回。

    讲了这么多指令想必已经有很多同学开始打退堂鼓了。没错，ASM 就是让我们直接面对底层字节码。要追求最快的 AOP 执行效率也要从字节码入手。不过为了方便开发，我再介绍一个工具，ASM-Bytecode。它是一个Eclipse插件，专门用于 ASM 框架下开发的辅助工具。它可以帮助我们生成一些繁琐的代码，从而让我们尽量绕开对底层组合虚拟机指令的关心。插件更新地址：“<http://andrei.gmxhome.de/eclipse/>”，项目首页：“<http://andrei.gmxhome.de/bytecode/index.html>”

    下面设计一个简单的架构。让任何一个被代理的类在其方法调用之前和返回之后都要调用我们的一个静态方法。为了区别它们分别使用两个方法来代表 Aop 不同的切点，分别是调用前和调用后，拦截器代码如下：

public class AopInterceptor {

public static void beforeInvoke() {

System.out.println("before");

};

public static void afterInvoke() {

System.out.println("after");

};

}

    接下来只需要在代理类的方法中插入对这两个方法的调用即可。首先设想被代理的方法最终执行的代码应该是下面这个样子的：

public class TestBean {

public void halloAop() {

AopInterceptor.beforeInvoke();

System.out.println("Hello Aop");

AopInterceptor.afterInvoke();

}

}

使用 ASM-Bytecode 工具，将这段代码转变为 ASM 代码。下面是上面这段代码的转换结果：

{

mv = cw.visitMethod(ACC\_PUBLIC, "halloAop", "()V", null, null);

mv.visitCode();

Label l0 = new Label();

mv.visitLabel(l0);

mv.visitLineNumber(24, l0);

mv.visitMethodInsn(INVOKESTATIC, "org/more/test/asm/AopInterceptor", "beforeInvoke", "()V");

Label l1 = new Label();

mv.visitLabel(l1);

mv.visitLineNumber(25, l1);

mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");

mv.visitLdcInsn("Hello Aop");

mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V");

Label l2 = new Label();

mv.visitLabel(l2);

mv.visitLineNumber(26, l2);

mv.visitMethodInsn(INVOKESTATIC, "org/more/test/asm/AopInterceptor", "afterInvoke", "()V");

Label l3 = new Label();

mv.visitLabel(l3);

mv.visitLineNumber(27, l3);

mv.visitInsn(RETURN);

Label l4 = new Label();

mv.visitLabel(l4);

mv.visitLocalVariable("this", "Lorg/more/test/asm/TestBean;", null, l0, l4, 0);

mv.visitMaxs(2, 1);

mv.visitEnd();

}

    上面这段代码的作用是用 ASM 输出整个 helloAop 的字节码部分，因此这是一段参考代码。  
    第 02 行：表示准备输出一个共有方法 “halloAop”，ACC\_PUBLIC 表示共有，相当于 public 修饰符。“()V” 是方法的参数包括返回值签名。“V” 是 void 的缩写。表示无返回值。后面两个 null 分别是方法的异常抛出信息和属性信息。  
    第 03 行：表示开始正式输出方法的执行代码。  
    第 07 行：表示调用静态方法，这行代码相当于“AopInterceptor.beforeInvoke();”，这个代码是我们需要的。同理第 17 行代码也是我们需要的。它相当于“AopInterceptor.afterInvoke();”。

    在上面生成的代码中 4，5，6，8，9，10，14，15，16，18，19，20 行看到如下内容：

Label l2 = new Label();

mv.visitLabel(l2);

mv.visitLineNumber(26, l2);

这些内容表示 Java 代码的行号标记，可以删除不用。在方法的最后部分代码：

Label l4 = new Label();

mv.visitLabel(l4);

mv.visitLocalVariable("this", "Lorg/more/test/asm/TestBean;", null, l0, l4, 0);

也是可以被删除不用的，这部分代码表示向 class 文件中写入方法本地变量表的名称以及类型。经过精简之后就是下面的代码了：

mv = cw.visitMethod(ACC\_PUBLIC, "halloAop", "()V", null, null);

mv.visitCode();

mv.visitMethodInsn(INVOKESTATIC, "org/more/test/asm/AopInterceptor", "beforeInvoke", "()V");

mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");

mv.visitLdcInsn("Hello Aop");

mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V");

mv.visitMethodInsn(INVOKESTATIC, "org/more/test/asm/AopInterceptor", "afterInvoke", "()V");

mv.visitInsn(RETURN);

mv.visitMaxs(2, 1);

mv.visitEnd();

逐句对应解释：  
    01 行：相当于 public void halloAop() 方法声明。  
    02 行：正式开发方法内容的填充。  
    03 行：调用静态方法，相当于：“AopInterceptor.beforeInvoke();”。  
    04 行：取得一个静态字段将其放入堆栈，相当于“System.out”。“Ljava/io/PrintStream;”是字段类型的描述，翻译过来相当于：“java.io.PrintStream”类型。在字节码中凡是引用类型均由“L”开头“;”结束表示，中间是类型的完整名称。  
    05 行：将字符串“Hello Aop”放入堆栈，此时堆栈中第一个元素是“System.out”，第二个元素是“Hello Aop”。  
    06 行：调用PrintStream类型的“println”方法。签名“(Ljava/lang/String;)V”表示方法需要一个字符串类型的参数，并且无返回值。  
    07 行：调用静态方法，相当于：“AopInterceptor.afterInvoke();”。  
    08 行：是 JVM 在编译时为方法自动加上的“return”指令。该指令必须在方法结束时执行不可缺少。  
    09 行：表示在执行这个方法期间方法的堆栈空间最大给予多少。  
    10 行：表示方法输出结束。

下面就是安插 Aop 实现的 ASM 代码：

class AopClassAdapter extends ClassVisitor implements Opcodes {

public AopClassAdapter(int api, ClassVisitor cv) {

super(api, cv);

}

public void visit(int version, int access, String name,

String signature, String superName, String[] interfaces) {

//更改类名，并使新类继承原有的类。

super.visit(version, access, name + "\_Tmp", signature, name, interfaces);

{//输出一个默认的构造方法

MethodVisitor mv = super.visitMethod(ACC\_PUBLIC, "<init>",

"()V", null, null);

mv.visitCode();

mv.visitVarInsn(ALOAD, 0);

mv.visitMethodInsn(INVOKESPECIAL, name, "<init>", "()V");

mv.visitInsn(RETURN);

mv.visitMaxs(1, 1);

mv.visitEnd();

}

}

public MethodVisitor visitMethod(int access, String name,

String desc, String signature, String[] exceptions) {

if ("<init>".equals(name))

return null;//放弃原有类中所有构造方法

if (!name.equals("halloAop"))

return null;// 只对halloAop方法执行代理

//

MethodVisitor mv = super.visitMethod(access, name,

desc, signature, exceptions);

return new AopMethod(this.api, mv);

}

}

class AopMethod extends MethodVisitor implements Opcodes {

public AopMethod(int api, MethodVisitor mv) {

super(api, mv);

}

public void visitCode() {

super.visitCode();

this.visitMethodInsn(INVOKESTATIC, "org/more/test/asm/AopInterceptor", "beforeInvoke", "()V");

}

public void visitInsn(int opcode) {

if (opcode == RETURN) {//在返回之前安插after 代码。

mv.visitMethodInsn(INVOKESTATIC, "org/more/test/asm/AopInterceptor", "afterInvoke", "()V");

}

super.visitInsn(opcode);

}

}

接下来就是使用 ASM 改写 Java 类：

ClassWriter cw = new ClassWriter(0);

//

InputStream is = Thread.currentThread().getContextClassLoader()

.getResourceAsStream("org/more/test/asm/TestBean.class");

ClassReader reader = new ClassReader(is);

reader.accept(new AopClassAdapter(ASM4, cw), ClassReader.SKIP\_DEBUG);

//

byte[] code = cw.toByteArray();

接下来编写一个 ClassLoader 加载我们的新类就可以了，新类的名称后面多了“\_Tmp”。本文的所有代码可以在下面这个地址中得到：

<http://www.oschina.net/code/snippet_1166271_24995>

|  |
| --- |
|  |