Android AOP之ASM技术研发

|  |  |  |
| --- | --- | --- |
| 版本号 | 描述 | 日期(更新日期)/更新人 |
| 1.0 | Android AOP之ASM技术 | 2018/08/28 刘志保 |
|  |  |  |

预备知识:

<0> : 文档参考https://asm.ow2.io/developer-guide.html

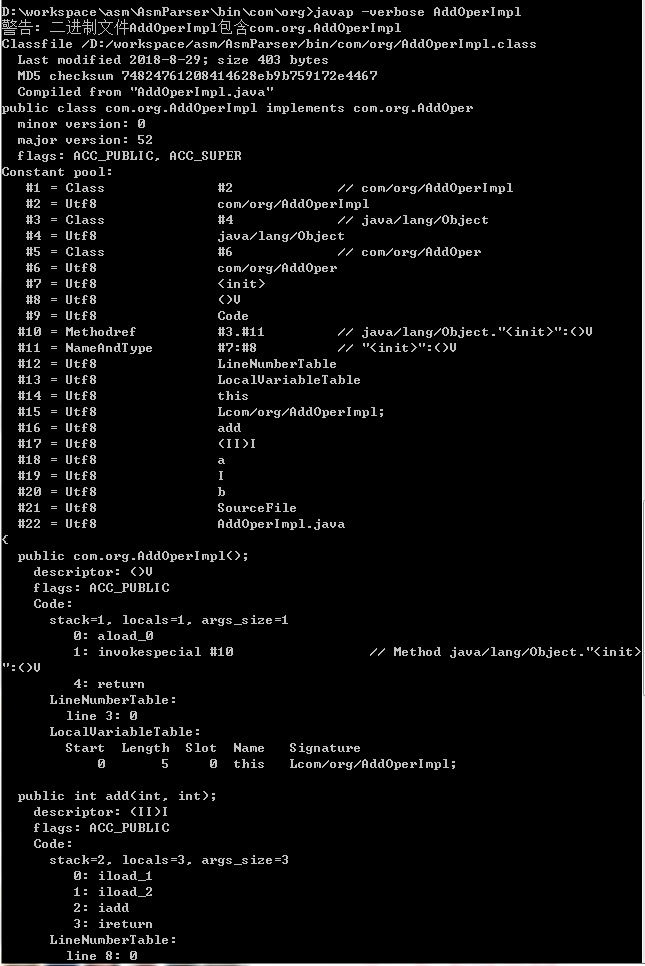
<1> : 先下载asm-util-3.2.jar和asm- 3.2.jar两个包,其中asm- 5.0.3.jar版本略有不同;

<http://www.java2s.com/Code/Jar/a/Downloadasmutil32jar.htm>

[asm-3.2.jar](http://files.cnblogs.com/liuling/asm-3.2.rar).

<2> : 下载Java Bytecode Editor工具软件[JBE],下载地址 : http://set.ee/jbe/

如果仅仅只是查看,也可以通过命令 : javap -verbose 你的class名字,不带.class扩展名.例如使用如下:



<3> : java字节码知识: https://segmentfault.com/a/1190000008606277

<4> : asm.jar包API;

<5> : 反编译工具 : <http://jd.benow.ca/> 名字 : JD-GUI,这个工具使用的时候需要注意,如果class有依赖性,那么依赖的包也要放在一起,不能够把class文件单独拿出来反编译,不然没有结果的.

下面正式开始介绍:

<1> : 先预定要生成一个什么样的类,然后再去生成,比如预定生成如下类的字节码:

AddOper.java

**package** com.org;

**public** **interface** AddOper {

**public** **static** **final** String ***SYMBOL*** = "+";

**public** **int** add(**int** a, **int** b);

}

AddOperImpl.java

**package** com.org;

**public** **class** AddOperImpl **implements** AddOper{

@Override

**public** **int** add(**int** a, **int** b) {

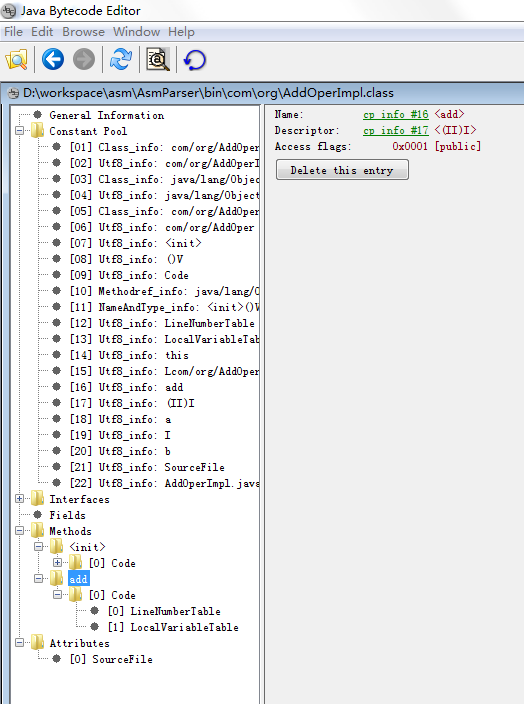
// **TODO** Auto-generated method stub

**return** a + b;

}

}

先把这两个类编译生成class文件,然后用JBE工具解析AddOperImpl.class文件,信息如下:



根据这个信息,新建一个java工程,导入上面两个jar包.新建CreateAddOperImpl.java类,如下:

**package** com.org;

**import** java.io.File;

**import** java.io.FileOutputStream;

**import** java.io.PrintWriter;

**import** java.lang.reflect.InvocationTargetException;

**import** java.lang.reflect.Method;

**import** org.objectweb.asm.ClassWriter;

**import** org.objectweb.asm.MethodVisitor;

**import** org.objectweb.asm.Opcodes;

**import** org.objectweb.asm.util.TraceClassVisitor;

**public** **class** CreateAddOperImpl {

**static** ClassWriter *cw* = **new** ClassWriter(0);

**public** **static** **void** createImpl() **throws** Exception {

PrintWriter printWriter = **new** PrintWriter(System.***out***);

TraceClassVisitor visitor = **new** TraceClassVisitor(*cw*, printWriter);

visitor.visit(Opcodes.***V1\_7***, Opcodes.***ACC\_PUBLIC***, "asm/demo/AddOperImpl", **null**, "java/lang/Object", **new** String[]{"asm/demo/AddOper"});

//添加构造方法

/\*

\* 0 aload\_0

\* 1 invokespecial #10 <java/lang/Object/<init>()V>

\* 4 return

\* \*/

MethodVisitor mv = visitor.visitMethod(Opcodes.***ACC\_PUBLIC***, "<init>", "()V", **null**, **null**);

mv.visitCode();

mv.visitVarInsn(Opcodes.***ALOAD***, 0);

mv.visitMethodInsn(Opcodes.***INVOKESPECIAL***, "java/lang/Object", "<init>", "()V");

mv.visitInsn(Opcodes.***RETURN***);

mv.visitMaxs(1, 1);

mv.visitEnd();

// 添加add方法

/\*

\* 0 iload\_1

\* 1 iload\_2

\* 2 iadd

\* 3 ireturn

\* \*/

mv = visitor.visitMethod(Opcodes.***ACC\_PUBLIC***, "add", "(II)I", **null**, **null**);

mv.visitCode();

mv.visitVarInsn(Opcodes.***ILOAD***, 1);

mv.visitVarInsn(Opcodes.***ILOAD***, 2);

mv.visitInsn(Opcodes.***IADD***);

mv.visitInsn(Opcodes.***IRETURN***);

mv.visitMaxs(2, 3);

mv.visitEnd();

visitor.visitEnd();

FileOutputStream fos = **new** FileOutputStream(**new** File("D:/AddOperImpl.class"));

fos.write(*cw*.toByteArray());

fos.close();

}

**public** **int** usingImpl() **throws** IllegalAccessException, IllegalArgumentException, InvocationTargetException, InstantiationException {

MyClassLoader classLoader = **new** MyClassLoader();

Class<?> clazz = classLoader.defineClassForName("asm.demo.AddOperImpl", *cw*.toByteArray());

Method addMethod;

**try** {

addMethod = clazz.getMethod("add", **int**.**class**, **int**.**class**);

Object result = addMethod.invoke(clazz.newInstance(), 10, 20);

**if**(result != **null** && result **instanceof** Integer) {

System.***out***.println((Integer) result);

**return** (Integer)result;

}

} **catch** (NoSuchMethodException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

} **catch** (SecurityException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** -1;

}

}

分析上面的程序.

<1> : 首先创建类AddOperImpl.java,如下:

PrintWriter printWriter = **new** PrintWriter(System.***out***);

TraceClassVisitor visitor = **new** TraceClassVisitor(*cw*, printWriter);

visitor.visit(Opcodes.***V1\_7***, Opcodes.***ACC\_PUBLIC***, "asm/demo/AddOperImpl", **null**, "java/lang/Object", **new** String[]{"asm/demo/AddOper"});

visit方法:

第一个参数为jdk版本,根据自己电脑安装配置的jdk;

第二个是访问权限,设为公共;

第三个参数类的全路径+类名;

第四个参数;

第五个参数,类都是Object;

第六个为该类实现的接口数组;

<2> : 然后构建类的构造方法:

//添加构造方法

/\*

\* 0 aload\_0

\* 1 invokespecial #10 <java/lang/Object/<init>()V>

\* 4 return

\* \*/

MethodVisitor mv = visitor.visitMethod(Opcodes.***ACC\_PUBLIC***, "<init>", "()V", **null**, **null**);

mv.visitCode();

mv.visitVarInsn(Opcodes.***ALOAD***, 0);

mv.visitMethodInsn(Opcodes.***INVOKESPECIAL***, "java/lang/Object", "<init>", "()V");

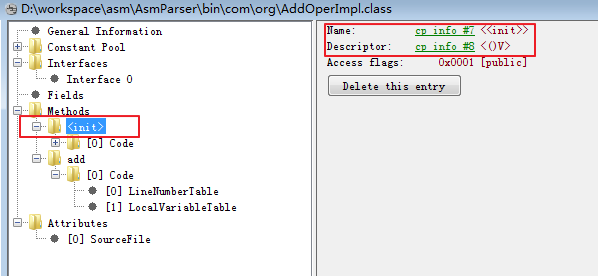
mv.visitInsn(Opcodes.***RETURN***);

mv.visitMaxs(1, 1);

mv.visitEnd();

其中:

visitor.visitMethod(Opcodes.***ACC\_PUBLIC***, "<init>", "()V", **null**, **null**);信息通过下面获取:



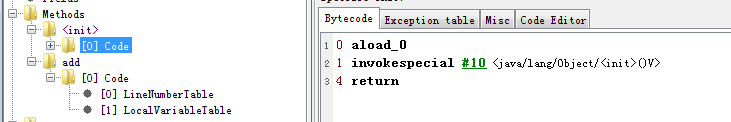
其中mv.visitCode();

mv.visitVarInsn(Opcodes.***ALOAD***, 0);

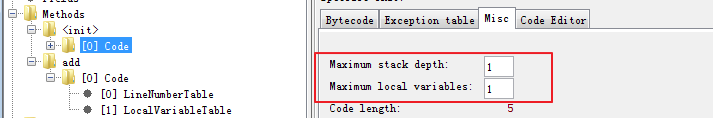
mv.visitMethodInsn(Opcodes.***INVOKESPECIAL***, "java/lang/Object", "<init>", "()V");

mv.visitInsn(Opcodes.***RETURN***);

通过下面获知:



其中mv.visitMaxs(1, 1);如下:



这样就把构造函数的字节码编辑好了.

<3> : 通add方法:

// 添加add方法

/\*

\* 0 iload\_1

\* 1 iload\_2

\* 2 iadd

\* 3 ireturn

\* \*/

mv = visitor.visitMethod(Opcodes.***ACC\_PUBLIC***, "add", "(II)I", **null**, **null**);

mv.visitCode();

mv.visitVarInsn(Opcodes.***ILOAD***, 1);

mv.visitVarInsn(Opcodes.***ILOAD***, 2);

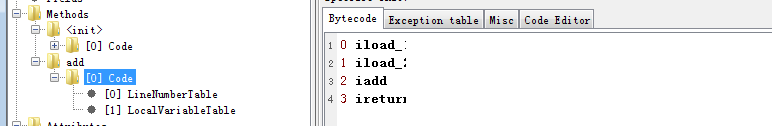
mv.visitInsn(Opcodes.***IADD***);

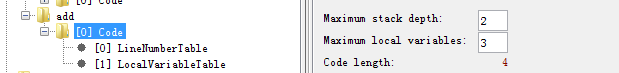
mv.visitInsn(Opcodes.***IRETURN***);

mv.visitMaxs(2, 3);

mv.visitEnd();

visitor.visitEnd();





<4> : 然后把它写入文件:

FileOutputStream fos = **new** FileOutputStream(**new** File("D:/AddOperImpl.class"));

fos.write(*cw*.toByteArray());

fos.close();

这个.class就可以被使用了.

上面有了一个初步的了解,下面看看ASM提供的三个核心组件: ClassReader, ClassWriter, ClassAdapter.

ClassReader 用来解析编译过的 class 的字节数组。 然后，调用 ClassVisitor 实例的  
visitXxx方法，其中 ClassVisitor实例作为 ClassReader.accept方法的参数传递进去的。  
ClassReader 可以被看作是一个事件产生者.

ClassWriter 是 ClassVisitor 接口的一个实现，用来以二进制方式构建编译后的类。它  
产生一个包含编译后的类的字节数组，可以通过它的 toByteArray 方法来或得。它  
可以被看作是一个事件消费者.

ClassAdapter 也是 ClassVisitor 接口的一个实现，它将对它的方法调用委托给另一个  
ClassVisitor。它可以被认为是一个事件过滤器.

整个核心类core API,具体参看: http://www.blogjava.net/DLevin/archive/2014/06/25/414292.html

上面有点混乱,我总结一下:

<1> : 字节码预备知识: 类字节码格式可以具体参考：[《Java字节码格式详解1》](http://www.blogjava.net/DLevin/archive/2011/09/05/358033.html)、[《Java字节码格式详解2》](http://www.blogjava.net/DLevin/archive/2011/09/05/358034.html)、[《Java字节码格式详解3》](http://www.blogjava.net/DLevin/archive/2011/09/05/358035.html)

<2> : 几个核心类:[ http://blog.hakugyokurou.net/?p=409]

ClassWriter

修改,更新,插入字节码全靠它

ClassReader

读取当前的字节码和信息全靠它

ClassAdapter

打桩用的,真正实施插入,删除等具体业务的实现

TraceClassVisitor[工具类]

字节码信息打印显示查看等,工具类有很多,不只是这一个

ClassLoader

加载字节码,然后通过反射机制进行调用

ClassVisitor---这个接口看asm源码理解

整个过程,多写代码测试,基本的操作就完全没问题了.