Q:什么是asm？

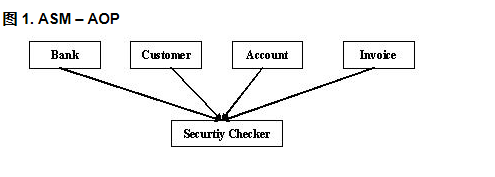
ASM 是一个 Java 字节码操控框架。它能被用来动态生成类或者增强既有类的功能。ASM 可以直接产生二进制 class 文件，也可以在类被加载入 Java 虚拟机之前动态改变类行为。Java class 被存储在严格格式定义的 .class 文件里，这些类文件拥有足够的元数据来解析类中的所有元素：类名称、方法、属性以及 Java 字节码（指令）。ASM 从类文件中读入信息后，能够改变类行为，分析类信息，甚至能够根据用户要求生成新类。

与 BCEL 和 SERL 不同，ASM 提供了更为现代的编程模型。对于 ASM 来说，Java class 被描述为一棵树；使用 “Visitor” 模式遍历整个二进制结构；事件驱动的处理方式使得用户只需要关注于对其编程有意义的部分，而不必了解 Java 类文件格式的所有细节：ASM 框架提供了默认的 “response taker”处理这一切。

<http://www.ibm.com/developerworks/cn/java/j-lo-asm30/>

Q: 为什么要动态生成 Java 类？

动态生成 Java 类与 AOP 密切相关的。AOP 的初衷在于软件设计世界中存在这么一类代码，零散而又耦合：零散是由于一些公有的功能（诸如著名的 log 例子）分散在所有模块之中；同时改变 log 功能又会影响到所有的模块。出现这样的缺陷，很大程度上是由于传统的 面向对象编程注重以继承关系为代表的“纵向”关系，而对于拥有相同功能或者说方面 （Aspect）的模块之间的“横向”关系不能很好地表达。例如，目前有一个既有的银行管理系统，包括 Bank、Customer、Account、Invoice 等对象，现在要加入一个安全检查模块， 对已有类的所有操作之前都必须进行一次安全检查。



然而 Bank、Customer、Account、Invoice 是代表不同的事务，派生自不同的父类，很难在高层上加入关于 Security Checker 的共有功能。对于没有多继承的 Java 来说，更是如此。传统的解决方案是使用 Decorator 模式，它可以在一定程度上改善耦合，而功能仍旧是分散的 —— 每个需要 Security Checker 的类都必须要派生一个 Decorator，每个需要 Security Checker 的方法都要被包装（wrap）。下面我们以 Account类为例看一下 Decorator：

首先，我们有一个 SecurityChecker类，其静态方法 checkSecurity执行安全检查功能：

public class SecurityChecker {

public static void checkSecurity() {

System.out.println("SecurityChecker.checkSecurity ...");

//TODO real security check

}

}

另一个是 Account类：

public class Account {

public void operation() {

System.out.println("operation...");

//TODO real operation

}

}

若想对 operation加入对 SecurityCheck.checkSecurity()调用，标准的 Decorator 需要先定义一个 Account类的接口：

public interface Account {

void operation();

}

然后把原来的 Account类定义为一个实现类：

public class AccountImpl extends Account{

public void operation() {

System.out.println("operation...");

//TODO real operation

}

}

定义一个 Account类的 Decorator，并包装 operation方法：

public class AccountWithSecurityCheck implements Account {

private Account account;

public AccountWithSecurityCheck (Account account) {

this.account = account;

}

public void operation() {

SecurityChecker.checkSecurity();

account.operation();

}

}

在这个简单的例子里，改造一个类的一个方法还好，如果是变动整个模块，Decorator 很快就会演化成另一个噩梦。动态改变 Java 类就是要解决 AOP 的问题，提供一种得到系统支持的可编程的方法，自动化地生成或者增强 Java 代码。这种技术已经广泛应用于最新的 Java 框架内，如 Hibernate，Spring 等。

<http://www.ibm.com/developerworks/cn/java/j-lo-asm30/>

Q: 为什么选择 ASM ？

最直接的改造 Java 类的方法莫过于直接改写 class 文件。Java 规范详细说明了 class 文件的格式，直接编辑字节码确实可以改变 Java 类的行为。直到今天，还有一些 Java 高手们使用最原始的工具，如 UltraEdit 这样的编辑器对 class 文件动手术。是的，这是最直接的方法，但是要求使用者对 Java class 文件的格式了熟于心：小心地推算出想改造的函数相对文件首部的偏移量，同时重新计算 class 文件的校验码以通过 Java 虚拟机的安全机制。

Java 5 中提供的 Instrument 包也可以提供类似的功能：启动时往 Java 虚拟机中挂上一个用户定义的 hook 程序，可以在装入特定类的时候改变特定类的字节码，从而改变该类的行为。但是其缺点也是明显的：

* Instrument 包是在整个虚拟机上挂了一个钩子程序，每次装入一个新类的时候，都必须执行一遍这段程序，即使这个类不需要改变。
* 直接改变字节码事实上类似于直接改写 class 文件，无论是调用 ClassFileTransformer. transform(ClassLoader loader, String className, Class classBeingRedefined, ProtectionDomain protectionDomain, byte[] classfileBuffer)，还是 Instrument.redefineClasses(ClassDefinition[] definitions)，都必须提供新 Java 类的字节码。也就是说，同直接改写 class 文件一样，使用 Instrument 也必须了解想改造的方法相对类首部的偏移量，才能在适当的位置上插入新的代码。

尽管 Instrument 可以改造类，但事实上，Instrument 更适用于监控和控制虚拟机的行为。

一种比较理想且流行的方法是使用 java.lang.ref.proxy。我们仍旧使用上面的例子，给 Account类加上 checkSecurity 功能 :

首先，Proxy 编程是面向接口的。下面我们会看到，Proxy 并不负责实例化对象，和 Decorator 模式一样，要把 Account定义成一个接口，然后在 AccountImpl里实现 Account接口，接着实现一个 InvocationHandlerAccount方法被调用的时候，虚拟机都会实际调用这个 InvocationHandler的 invoke方法：

class SecurityProxyInvocationHandler implements InvocationHandler {

private Object proxyedObject;

public SecurityProxyInvocationHandler(Object o) {

proxyedObject = o;

}

public Object invoke(Object object, Method method, Object[] arguments)

throws Throwable {

if (object instanceof Account && method.getName().equals("opertaion")) {

SecurityChecker.checkSecurity();

}

return method.invoke(proxyedObject, arguments);

}

}

最后，在应用程序中指定 InvocationHandler生成代理对象：

public static void main(String[] args) {

Account account = (Account) Proxy.newProxyInstance(

Account.class.getClassLoader(),

new Class[] { Account.class },

new SecurityProxyInvocationHandler(new AccountImpl())

);

account.function();

}

其不足之处在于：

* Proxy 是面向接口的，所有使用 Proxy 的对象都必须定义一个接口，而且用这些对象的代码也必须是对接口编程的：Proxy 生成的对象是接口一致的而不是对象一致的：例子中 Proxy.newProxyInstance生成的是实现 Account接口的对象而不是 AccountImpl的子类。这对于软件架构设计，尤其对于既有软件系统是有一定掣肘的。
* Proxy 毕竟是通过反射实现的，必须在效率上付出代价：有实验数据表明，调用反射比一般的函数开销至少要大 10 倍。而且，从程序实现上可以看出，对 proxy class 的所有方法调用都要通过使用反射的 invoke 方法。因此，对于性能关键的应用，使用 proxy class 是需要精心考虑的，以避免反射成为整个应用的瓶颈。

ASM 能够通过改造既有类，直接生成需要的代码。增强的代码是硬编码在新生成的类文件内部的，没有反射带来性能上的付出。同时，ASM 与 Proxy 编程不同，不需要为增强代码而新定义一个接口，生成的代码可以覆盖原来的类，或者是原始类的子类。它是一个普通的 Java 类而不是 proxy 类，甚至可以在应用程序的类框架中拥有自己的位置，派生自己的子类。

相比于其他流行的 Java 字节码操纵工具，ASM 更小更快。ASM 具有类似于 BCEL 或者 SERP 的功能，而只有 33k 大小，而后者分别有 350k 和 150k。同时，同样类转换的负载，如果 ASM 是 60% 的话，BCEL 需要 700%，而 SERP 需要 1100% 或者更多。

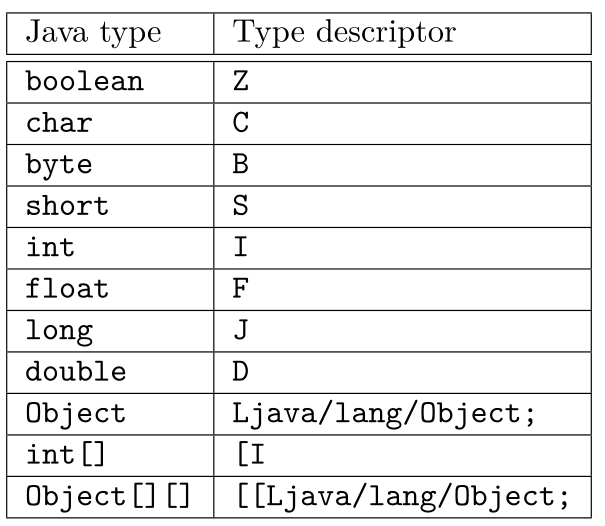
ASM 已经被广泛应用于一系列 Java 项目：AspectWerkz、AspectJ、BEA WebLogic、IBM AUS、OracleBerkleyDB、Oracle TopLink、Terracotta、RIFE、EclipseME、Proactive、Speedo、Fractal、EasyBeans、BeanShell、Groovy、Jamaica、CGLIB、dynaop、Cobertura、JDBCPersistence、JiP、SonarJ、Substance L&F、Retrotranslator 等。Hibernate 和 Spring 也通过 cglib，另一个更高层一些的自动代码生成工具使用了 ASM。

<http://www.ibm.com/developerworks/cn/java/j-lo-asm30/>

Q:怎么获得import？

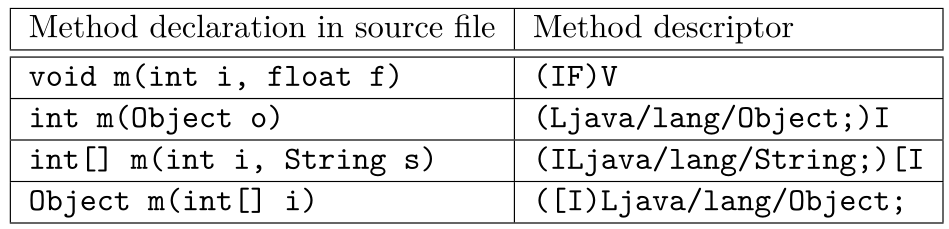
因为被编译的class不包括import所以如果要获得import获得里面的所有调用

Q:java类型与字节码类型的关系



这只是部分

Q:方法的表示是什么样子？



Q:怎么访问一个class

是通过ClassReader 来访问

package com.kga.q.q1;

import java.io.IOException;

import org.objectweb.asm.ClassReader;

public class Q1 {

private int f1;

private int f2;

public int getF1() {

return f1;

}

public void setF1(int f1) {

this.f1 = f1;

}

public int getF2() {

return f2;

}

public void setF2(int f2) {

this.f2 = f2;

}

public static void main(String[] args) {

ClassReader cr = null;

try {

cr = new ClassReader(Q1.class.getName());

Q1ClassVisitor qv=new Q1ClassVisitor();

cr.accept(qv, 0);

} catch (IOException e) {

throw new RuntimeException(e);

}

}

}

package com.kga.q.q1;

import org.objectweb.asm.ClassVisitor;

import org.objectweb.asm.FieldVisitor;

import org.objectweb.asm.MethodVisitor;

import org.objectweb.asm.Opcodes;

public class Q1ClassVisitor extends ClassVisitor {

public Q1ClassVisitor() {

super(Opcodes.ASM4);

}

public FieldVisitor visitField(int access, String name, String desc, String signature, Object value) {

System.out.println("" + desc + " " + name);

return null;

}

public MethodVisitor visitMethod(int access, String name, String desc, String signature, String[] exceptions) {

System.out.println("" + name + desc);

return null;

}

public void visit(int version, int access, String name, String signature, String superName, String[] interfaces) {

System.out.println(name + " extends " + superName + " {");

}

public void visitEnd() {

System.out.println("}");

}

}

Q:怎样修改一个class？