

精准测试技术之 Java概述

目录

- **精准回归**
- **AST简介**
- **ASM简介**
- **ASM原理**
- **Java字节码**

精准测试

- 基于代码更新的API调用关系达到精准回归测试，可以更大程度的减少测试用例回归和测试时间，避免回归所有用例
- 精准回归分为两个阶段
 - 一是获取API变更列表 **Java AST+SCM Diff**
 - 二是API调用关系 **Java ASM**

Java AST

- Eclipse JDT 提供了操纵 Java 源代码、检测错误、执行编译和启动程序的的 API
 - Eclipse JDT 有自己的文档对象模型(DOM) , 思想和XML 是一致的 : 抽象的语法树(AST)
 - **ASTParser**-AST语法树的解析器
 - **ASTNodes**-AST 层次结构
-
- 作用 :
 - ① 解析Java语法树
 - ② 生成Java源码

解析Java类

// 创建解析器 Java语言规范第3版

JLS3 J2SE 1.5.
JLS2 J2SE 1.4.

ASTParser parser=ASTParser.newParser(**AST.JLS3**);

//设置解析类型

parser

//解

parser

//获

Com

1. K_COMPILATION_UNIT: 一个编译单元,
2. K_STATEMENTS: Java statements, 比如赋值语句
3. K_EXPRESSION: Java expressions
4. K_CLASS_BODY_DECLARATIONS: Java class里的元素

parser.createAST(null);

源代码结构信息

//获取类型

```
List types = result.types();
```

// 取得类型声明

```
TypeDeclaration typeDec = (TypeDeclaration) types.get(0);
```

// 引用import

```
List importList = result.imports();
```

// 取得包名

```
PackageDeclaration packetDec = result.getPackage();
```

// 取得类名

```
String className = typeDec.getName().toString();
```

// 取得函数(Method)声明列表

```
MethodDeclaration methodDec[] = typeDec.getMethods();
```

// 取得函数(Field)声明列表

```
FieldDeclaration fieldDec[] = typeDec.getFields();
```

- **for (MethodDeclaration method : methodDec) {**
- **(method.getName());//函数名**
- Block block = method.getBody();
- System.out.println(method);
- ("Flags: " + method.getFlags());//**函数标记**
- ("Length: " + method.getLength());//**函数长度-位置长度**
- ("StartPosition: " + method.getStartPosition());//**函数开始位置**
- result.getLineNumber(method.getStartPosition()); //**函数开始行号**
- }

ASM简介

- **ASM 是一个 Java 字节码操控框架。**
- **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）。**

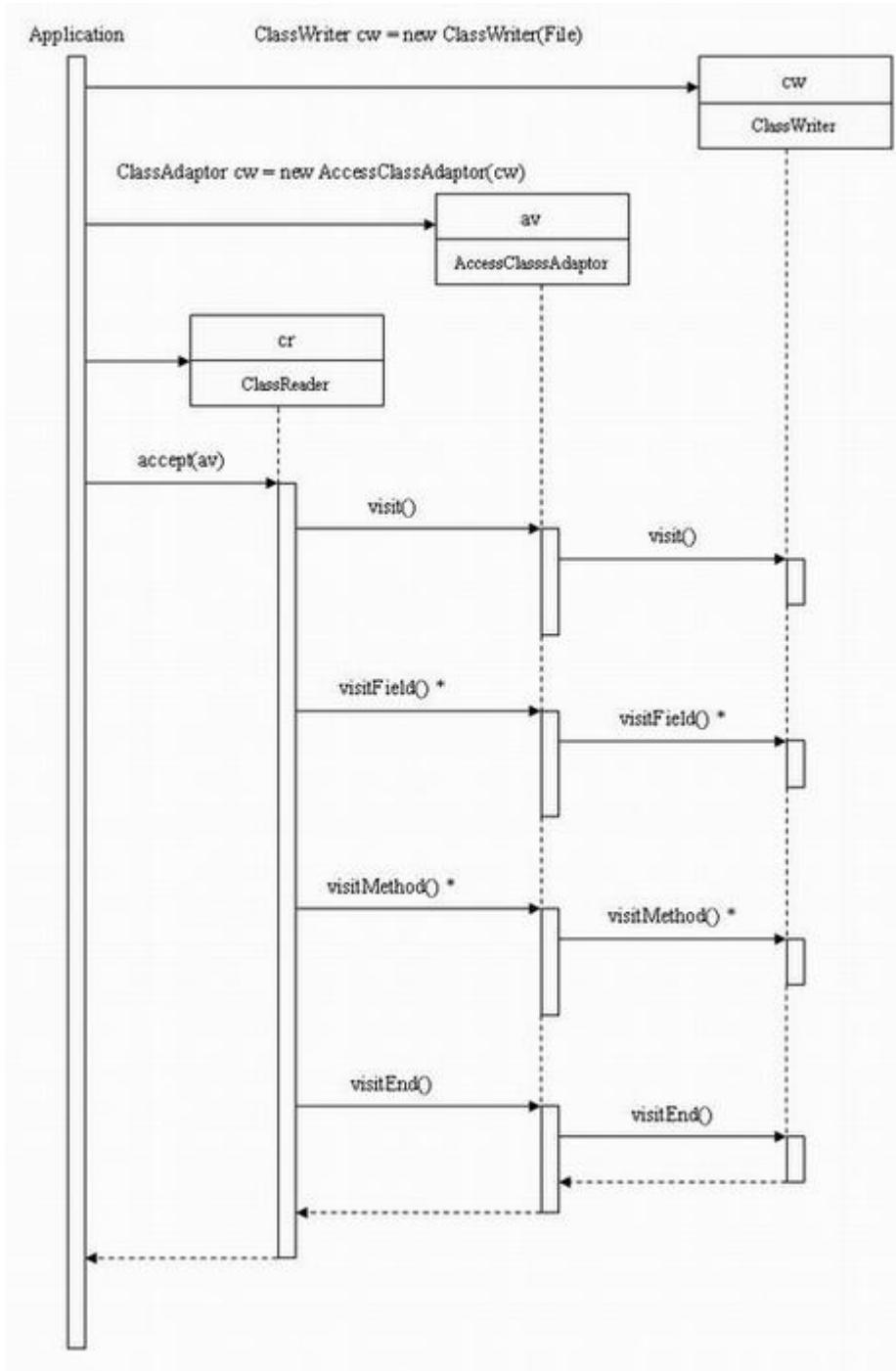
ASM编程框架

- ASM 通过 “**Tree**” 来表示复杂的字节码结构，并利用 Push 模型来对树进行遍历，在遍历过程中对字节码进行修改。
- 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

ASM编程原理

- 调用ClassReader类，它能正确的分析字节码，构建出抽象的树在内存中表示字节码。
- ClassReader会调用 accept方法，接受一个实现了 ClassVisitor接口的对象实例作为参数，然后依次调用 ClassVisitor接口的各个方法。
- 各个 ClassVisitor通过**职责链模式**，可以非常简单的封装对字节码的各种修改。

A S M 时 序 逻 辑



ASM编程举例

```
// 根据虚拟机的加载，设置className
String className = "ASMTTest.Shopping";
ClassReader cr = new ClassReader(className);
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
ClassAdapter classAdapter = new ByteCodeClassAdapter(cw);
ClassAdapter classDeleteAdapter = new DeleteFunctionAdapter(
    classAdapter);
cr.accept(classDeleteAdapter, ClassReader.SKIP_DEBUG);
byte[] data = cw.toByteArray();
File file = new File("bin/" + className.replace(".", "/") + ".class");
FileOutputStream fout = new FileOutputStream(file);
fout.write(data);
fout.close();
```

ASM编程举例

```
public class Shopping {  
  
    public void shopping() {  
        System.out.println("shopping in China");  
    }  
  
    public void deleteFunction() {  
        System.err.println("Can not be invoked");  
    }  
}
```

Shopping对应的Bytecode

```
public void shopping();
```

Code:

```
Stack=2, Locals=1, Args_size=1
0: getstatic      #15; //Field java/lang/System.out:Ljava/io/PrintStream;
3: ldc      #21; //String shopping in China
5: invokevirtual #23; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
8: return
```

LineNumberTable:

```
line 14: 0
```

```
line 15: 8
```

LocalVariableTable:

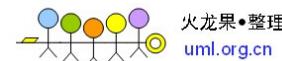
Start	Length	Slot	Name	Signature
0	9	0	this	LASMTest/Shopping;

Shopping对应的Bytecode

```
// access flags 0x1
public shopping()V
L0
LINENUMBER 14 L0
GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
LDC "shopping in China"
INVOKEVIRTUAL java/io/PrintStream.println(Ljava/lang/String;)V
L1
LINENUMBER 15 L1
RETURN
L2
LOCALVARIABLE this LASMTest/Shopping; L0 L2 0
MAXSTACK = 2
MAXLOCALS = 1
```

Shopping的ASM解析树

```
{  
mv = cw.visitMethod(ACC_PUBLIC, "shopping", "()V", null, null);  
mv.visitCode();  
Label l0 = new Label();  
mv.visitLabel(l0);  
mv.visitLineNumber(14, l0);  
mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");  
mv.visitLdcInsn("shopping in China");  
mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V");  
Label l1 = new Label();  
mv.visitLabel(l1);  
mv.visitLineNumber(15, l1);  
mv.visitInsn(RETURN);  
Label l2 = new Label();  
mv.visitLabel(l2);  
mv.visitLocalVariable("this", "LASMTest/Shopping;", null, l0, l2, 0);  
mv.visitMaxs(2, 1);  
mv.visitEnd();  
}
```



ByteCodeMethodAdapter

- 继承 MethodAdapter，实现 MethodVisitor 的所有接口
----**接口的适配器模式**

- ## • 覆蓋visitCode方法

Shopping的bytecode变化

```
public void shopping();
```

Code:

增加的字节码

```
Stack=2, Locals=1, Args_size=1
```

```
0: invokestatic #12; //Method java/lang/Thread.currentThread:()Ljava/lang/Thread;
3: invokevirtual #16; //Method java/lang/Thread.getStackTrace:()[[Ljava/lang/StackTraceElement;
6: invokestatic #22; //Method ASMTest/Depends.depends:([Ljava/lang/StackTraceElement;)V
9: getstatic #31; //Field java/lang/System.out:Ljava/io/PrintStream;
12: ldc #33; //String shopping in China
14: invokevirtual #39; //Method java/io/PrintStream.println:([Ljava/lang/String;)V
```

ASM解析树的变化

```
{  
mv = cw.visitMethod(ACC_PUBLIC, "shopping", "()V", null, null);  
mv.visitCode();  
mv.visitMethodInsn(INVOKESTATIC, "java/lang/Thread", "currentThread", "()Ljava/lang/Thread;");  
mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Thread", "getStackTrace", "[Ljava/lang/StackTraceElement;");  
mv.visitMethodInsn(INVOKESTATIC, "ASMTest/Depends", "depends", "([Ljava/lang/StackTraceElement;)V");  
mv.visitFieldInsn(GETSTATIC, "java/lang/System", "out", "Ljava/io/PrintStream;");  
mv.visitLdcInsn("shopping in China");  
mv.visitMethodInsn(INVOKEVIRTUAL, "java/io/PrintStream", "println", "(Ljava/lang/String;)V");  
mv.visitInsn(RETURN);  
mv.visitMaxs(2, 1);  
mv.visitEnd();
```

运行结果

```
java.lang.Thread.getStackTrace
ASMTest.Shopping.<init>
ASMTest.User.main
```

```
java.lang.Thread.getStackTrace
ASMTest.Shopping.shopping
ASMTest.User.doShopping
ASMTest.User.main
```

```
shopping in China
```

职责链模式

```
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
ClassAdapter classAdapter = new ByteCodeClassAdapter(cw);
ClassAdapter classDeleteAdapter = new DeleteFunctionAdapter(
    classAdapter);
cr.accept(classDeleteAdapter, ClassReader.SKIP_DEBUG);
```

DeleteFunctionAdapter

- 继承 ClassAdapter，实现 ClassVistor 的所有接口
--- 接口的适配器模式

- 覆盖 visitMethod 方法（删除 deleteFunction 方法）

```
public MethodVisitor visitMethod(int access,
        String name, String[] exceptions) {
    if (name.equals("deleteFunction")) {
        return null;
    }
    return super.visitMethod(access, name, exceptions);
}
```

运行结果

```
public class Shopping {  
  
    public void shopping() {  
        System.out.println("shopping in China");  
    }  
  
    public void deleteFunction() {  
        System.err.println("Can not be invoked");  
    }  
}
```

```
public static void main(String[] args) {  
    User user = new User(new Shopping());  
    user.doShopping();  
    user.deleteFunction();  
}
```

ASMT test.User.main

Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method deleteFunction() is undefined for the type Shopping

at ASMT test.User.deleteFunction(User.java:34)
at ASMT test.User.main(User.java:40)

ASM的应用

➤ Languages and AOP tools

AspectWerkz | AspectJ | BeanShell | CGLIB | dynaop | Clojure | [Groovy](#) | Jamaica | [JRuby](#) | [Jython](#) | NetLogo | Open Quark | WebSphere sMash | Coroutines | fun4j

➤ Java ME

EclipseME | MicroEmulator Sun Java ME emulation for Java SE

➤ Tools and frameworks

Fractal | Dr. Garbage | Proactive | Retrotranslator | RIFE | R-OSGi | Terracotta | Substance L&F | WindowBuilder | Javeleon

➤ Persistence

EasyBeans | Ebean | JDBCersistence | JPOX | [OpenEJB](#) | [Oracle BerkleyDB](#) | Oracle TopLink | Speedo

➤ Monitoring

[BEA WebLogic](#) | BTrace | Byteman | JiP | ByCounter | Limpid Log

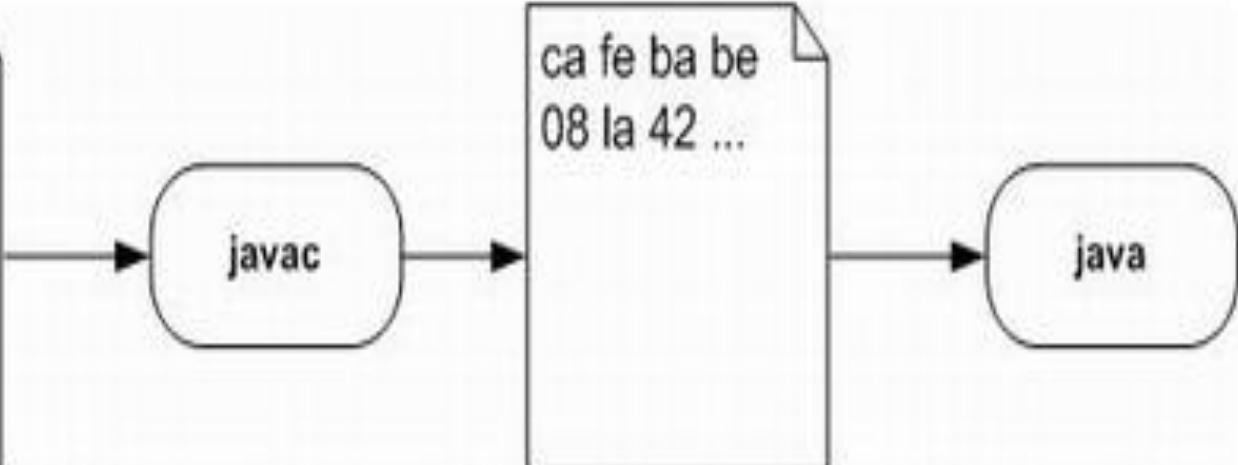
➤ Testing and code analysis

Agitar | [Cobertura](#) | Eclipse | JCarder | SemmleCode | Structure101 | [SonarJ](#) | TamiFlex

Java字节码

```
public class HelloWorld {  
    void hello() {  
        .....  
    }  
}
```

HelloWorld.java



HelloWorld.class

Java字节码结构

Magic	该项存放了一个 Java 类文件的魔数（magic number）和版本信息
Version	该项存放了 Java 类文件的版本信息
Constant Pool	该项存放了类中各种文字字符串、类名、方法名和接口名称、final 变量以及对外部类的引用信息等常量
Access_flag	该项指明了该文件中定义的是类还是接口，访问标志
This Class	指向表示该类全限定名称的字符串常量的指针
Super Class	指向表示父类全限定名称的字符串常量的指针
Interfaces	一个指针数组，存放了该类或父类实现的所有接口名称的字符串常量的指针
Fields	该项对类或接口中声明的字段进行了细致的描述
Methods	该项对类或接口中声明的方法进行了细致的描述
Class attributes	该项存放了在该文件中类或接口所定义的属性的基本信息

Shopping方法分析

```
public void shopping();
```

Code:

```
Stack=2, Locals=1, Args_size=1
0: getstatic #15; //Field java/lang/System.out:Ljava/io/PrintStream;
3: ldc #21; //String shopping in China
5: invokevirtual #23;
//Method java/io/PrintStream.println:(Ljava/lang/String;)V
8: return
```

Constant pool:

```
const #15 = Field #16.#18;
const #16 = class #17; // java/lang/System
const #18 = NameAndType #19:#20;// out:Ljava/io/PrintStream;
const #21 = String #22; // shopping in China
const #22 = Asciz shopping in China;
const #23 = Method #24.#26;
const #24 = class #25; // java/io/PrintStream
const #25 = Asciz java/io/PrintStream;
const #26 = NameAndType #27:#28;// println:(Ljava/lang/String;)V
const #27 = Asciz println;
const #28 = Asciz (Ljava/lang/String;)V;
```

域描述符

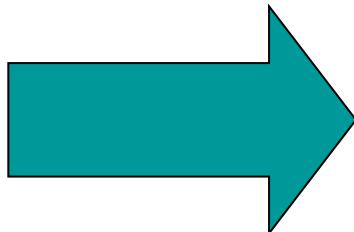
B	byte	有符号字节
C	char	字符
D	double	双精度 IEEE 754浮点数
F	float	单精度 IEEE 754浮点数
I	int	整数
J	long	长整数
L<classname>;	...	类实例
S	short	有符号 short
Z	boolean	true 或者 false
[...	一个数组维

方法描述符

- 字符V指示该方法没有返回值 (void)

- 方法描述符不论是实例方法还是static，都是一样的

```
public
String[][]
getShoppingList
(int i,
double d,
float[] f,
String string)
```



```
(  
ID[FLjava/lang/String;  
)  
[[Ljava/lang/String;
```

Java字节码指令

- **xLOAD和xSTORE**
- **stack操作符**
- **常量操作符**
- **运算符指令**
- **类型转换指令**
- **对象操作和装载指令**
- **控制转移指令**
- **方法调用返回**

xLOAD和xSTORE

- xLOAD-读取本地变量，并将其push压栈到操作数栈
- xSTORE-从操作数栈pop出栈一个变量并将其存到本地变量

ILOAD/ISTORE	boolean,byte, char, short, or int
LLOAD/LSTORE	long
FLOAD/FSTORE	float
DLOAD/DSTORE	double
ALOAD/ASTORE	任何非保留类型如object和array

stack操作符

- POP 出栈
- DUP 压栈栈首元素的拷贝
- SWAP 交换栈首两个元素的位置

装载常量操作符

- ACONST_NULL pushes null,
- ICONST_0 pushes the int value 0,
- FCONST_0 pushes 0f,
- DCONST_0 pushes 0d,
- BIPUSH b pushes the byte value b,
- SIPUSH s pushes the short value s,
- LDC cst pushes the arbitrary int,float, long, double, String, or class1 constant

运算符

- xADD +
- xSUB -
- xMUL *
- xDIV /
- xREM %
- xNEG -
- x is either I, L, F or D

- 移位: ishr,ishl,iushr,lshr,lshr,lushr
- 按位或: ior,lor
- 按位域: iand,land
- 按位异或: ixor,lxor

类型转换

放宽数值转换

- i2l
- i2f
- i2d
- l2f
- l2d
- f2d

缩窄数值转换

- i2b
- i2c
- i2s
- f2i
- f2l
- d2l
- d2i
- d2f

对象操作

- getfield,putfield 获取设置非static域
- getstatic,putstatic 获取设置static域
- xaload,xastore 装载和存储数组
- x is either b,c,s,i,l,f,d,a
- instanceof,chkcast 检查类实例属性

控制转移指令

- 条件转移

ifeq,iflt,ifle,ifne,ifgt,ifge,ifnull,ifnonnull
if_icmpne,if_icmpne,if_icmplt,if_icmpgt,if_icmple,if_icmpge
if_acmpne,if_acmpne,lcmp,fcmpl,fcmpg,dcmpl,dcmpg

- 复合条件转移

tableswitch,lookupswitch

- 无条件转移

goto, goto_w, jsr, jsr_w, ret

方法调用指令

- **invokevirtual**
调用对象的实例方法和类型
- **invokeinterface**
调用接口实现的方法，多态
- **invokespecial**
调用需要特殊处理的方法-实例初始化方法，`private`方法或超类方法
- **invokestatic**
调用类的类（`static`）方法

返回指令

- ireturn byte,char,short,int
- areturn class
- freturn float
- dreturn double
- lreturn long
- return void

Thanks