Java字节码(Bytecode)与ASM简单说明

[4条回复](http://blog.hakugyokurou.net/?p=409#comments)

"自由有诸多困难并且民主也并不完美,但我们绝不会筑起一堵墙将我们的人民禁锢住."  
-John Fitzgerald Kennedy

这篇文章是为[MCMod教程的Extra编第三篇](http://www.hakugyokurou.net/wordpress/?p=333)配套写成的,用于简单说明Java字节码和ASM的使用,由于作者是苦逼高三党,平时只能在坐车回来路上用手机上网学Java字节码╮(╯▽╰)╭难免可能有不妥甚至有误的地方,求触手神犇不要碾我,指出错误之处就好...

最后还要记住,这个说明的目的是速成,而不是深造...

另外在阅读过程中请务必打开这个网页:<http://en.wikipedia.org/wiki/Javabytecodeinstruction_listings>,以便随时查阅信息.

什么是Java字节码?

自行百度/谷歌.

好吧我知道这么说很欠抽,但我确实不擅长解释它...简单地说,一切Java操作都可以通过200种指令来表示出来,每一种指令通过一个8位整数(即byte)来标识,因此这些指令被称为字节码(Bytecode),一个编译完毕的Java文件(.class)虽然并不是全部由字节码组成,但至少字节码占了很大一部分.  
200种指令听起来比Java甚至是C#的保留关键字还要多,但事实上有不少是重复的.

Class文件的构造?

自行百度/谷歌.

随便一个都讲的比我好,不过请务必仔细看看常量池和方法这两部分.

Java字节码的栈帧

终于到我可以说的了...

如果你仔细读了关于字节码的文章后,你应该会知道Java中当一个方法被调用时会产生一个栈帧(Stack Frame),可以理解为那个方法被包含在了这个栈帧里,栈帧包括3个部分,局部变量区,操作数栈区和帧数据区.接下来我们主要要用到的是局部变量区和操作数栈区.

局部变量区有大小限制,最大大小已在编译时被编译器计算并指定, 它的结构(从0开始)是:  
类的实例的引用(对于静态方法则没有这个)  
参数  
方法运行时创建的局部变量

举例,对于:

|  |  |
| --- | --- |
| 1  2  3  4 | void example1(int a,int b)  {  int c = a + b;  } |

编译成字节码后,它的局部变量区最大大小是4,即0~3号位置是可用的,其中0号位是它的类的实例,1和2号位分别是a,b.3号位开始空缺,运行时被用于存储c.

然而,局部变量区的基本单位是32位,这意味着如果你储存long和double时它们需要占2个位置.

举例:

|  |  |
| --- | --- |
| 1  2  3  4  5 | static long example2()  {  long l = 1000000L;  return l;  } |

编译成字节码后,它的局部变量区最大大小只有2,首先它是静态方法,意味着它不需要储存它的类实例,其次它没有参数.它只有一个long类型的局部变量,由于long要占2个空位,因此它的最大大小只有2.

局部变量区的访问没有顺序限制,可以随意访问.

然后是操作数栈区,操作数栈区是一个栈,主要用于存储调用方法时的参数,比如当你要调用example1时,首先要先向操作数栈区压入一个类的实例(或者说是实例的引用),然后压入2个int变量,最后再调用example1方法,调用时栈帧会从操作数栈区里提出方法需要的数据.调用完毕后再将结果压入操作数栈进行下一步的操作.

再上一个实例,以GameData类下的setName方法为例.  
它的代码是:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | static void setName(Item item, String name, String modId)  {  int id = item.itemID;  ItemData itemData = idMap.get(id);  itemData.setName(name,modId);  } |

编译为字节码后是:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | aload0  getfield net/minecraft/item/Item/itemID I  istore3  getstatic cpw/mods/fml/common/registry/GameData/idMap Ljava/util/Map;  iload3  invokestatic java/lang/Integer/valueOf(I)Ljava/lang/Integer;  invokeinterface java/util/Map/get(Ljava/lang/Object;)Ljava/lang/Object; 2  checkcast cpw/mods/fml/common/registry/ItemData  astore 4  aload 4  aload1  aload2  invokevirtual cpw/mods/fml/common/registry/ItemData/setName(Ljava/lang/String;Ljava/lang/String;)V  return |

每个字节码的具体含义我们待会再了解,你现在只要知道xloadi是将局部变量区中的第i个变量/引用压入操作数栈区,x是变量/引用的类型,xstorei是将操作数栈区栈顶的变量/引用提出栈并存入局部变量区的第i个位置.invokexxx是调用方法.getfield是获取非静态字段值.getstatic是获取静态字段值.  
setName是个静态方法,因此它的局部变量区无需储存类的实例,所以它的0~2号局部变量分别对应3个参数.

第一行代码int id = item.itemID创建了一个局部变量id,并用参数item的itemID字段为其赋值,故此我们的字节码代码是:  
aload0 (将item的引用压入操作数栈区)  
getfield (从操作数栈栈顶提出一个引用(或者理解为类的实例),并从引用中获取一个字段值然后压入操作数栈区)  
istore3 (将栈顶的值出栈然后存入局部变量区3号位)

第二行代码ItemData itemData = idMap.get(id)创建了一个局部变量itemData.  
getstatic (获取一个静态字段并压入操作数栈)  
iload3 (读取3号局部变量并压入操作数栈)  
invokestatic (从栈顶取出一个值,并调用Integer类的静态方法,然后将返回值压入操作数栈.)  
invokeinterface (调用被实现的接口方法,从栈顶取出两个值,第一个值作为目标方法所在类的实例,第二个值作为参数值.最后将结果压入操作数栈)  
checkcast (检查操作数栈顶的值的类型)  
astore 4 (将操作数栈栈顶的值出栈并写入4号局部变量)

第三行代码itemData.setName(name,modId)调用了局部变量itemData的setName方法.  
aload 4 (将4号局部变量的值压入操作数栈区)  
aload1 (将1号局部变量的值压入操作数栈区)  
aload2 (将2号局部变量的值压入操作数栈区)  
invokevirtual (调用一个类的非静态方法,从栈顶取出3个值,第一个值作为目标方法所在类的实例,第二三个值作为参数值.最后将结果压入操作数栈)

这个方法因为没有返回值,所以代码中没有写return.编译器在这里自动补上了.  
return (退出方法,不返回任何值)

最后还得说,操作数栈区也是有最大栈深度的限制,也就是说操作数栈能存进多少数据是有限制的,这个数是在编译时由编译器指定.

ASM总览

ASM是啥我就不介绍了...具体看Extra编第三篇.

对于字节码的修改,ASM提供了2种模型,对象模型和事件模型.

对象模型

对象模型的本质其实是一个经过封装了的事件模型,它通过一个树状图来描述一个类,一个类包含多个节点,比如方法节点,字段节点,每个节点又有子节点,比如方法节点有操作码子节点...

树状图模式适宜处理简单的类修改,它的优点是便于使用和学习,而且节省了代码量.缺点是若要处理大量信息则会使代码变得复杂,并且难以做到代码复用.

获取一个类节点(ClassNode)

ASM使用ClassReader来解析字节码,ClassReader在解析完字节码后可以通过accept方法来将结果写入一个ClassNode.  
下面这个代码可以将一个字节码解析为一个ClassNode.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | static public ClassNode getClassNode(byte[] bytes)  {  ClassNode classNode = new ClassNode();  ClassReader classReader = new ClassReader(bytes);  classReader.accept(classNode, 0);  return classNode;  } |

类节点结构

类节点包括如下信息:

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 说明 |
| int | version | class文件的major版本,即编译它的java版本. |
| int | access | 访问级 |
| String | name | 类名,采用全地址,如java/lang/Object |
| String | signature | 签名,通常是null |
| String | superName | 基类类名,采用全地址. |
| List<String> | interfaces | 实现的接口,采用全地址 |
| String | sourceFile | 源代码文件,可能为null |
| String | sourceDebug | debug源,可能为null |
| String | outerClass | <似乎和外部类有关,没研究过...> |
| String | outerMethod | <似乎和匿名类有关,没研究过...> |
| String | outerMethodDesc | <似乎和匿名类有关,没研究过...> |
| List<AnnotationNode> | visibleAnnotations | 可见的注解 |
| List<AnnotationNode> | invisibleAnnotations | 不可见的注解 |
| List<Attribute> | attrs | 类的Attribute |
| List<InnerClassNode> | innerClasses | 类的内部类 |
| List<FieldNode> | fields | 类的字段 |
| List<MethodNode> | methods | 类的方法 |

它们的访问级全部为public,因此你能获得任何你想要的信息.

字段节点(FieldNode)

字段节点被储存在类节点的fields字段中,由于它是一个ArrayList,因此我们只能以遍历集合的形式来搜索我们需要的字段节点,如下列代码:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | for(FieldNode fieldNode : (List)classNode.fields)  {  if(fieldNode.name.equals("password")) //判断字段名是否匹配目标字段  {  //进行操作  }  } |

字段节点包括如下信息:

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 说明 |
| int | access | 访问级 |
| String | name | 字段名 |
| String | signature | 签名,通常是null |
| String | desc | 类型,如"Ljava/lang/Object;"或"F" |
| Object | value | 初始值,很可能是null |
| List<AnnotationNode> | visibleAnnotations | 可见的注解 |
| List<AnnotationNode> | invisibleAnnotations | 不可见的注解 |
| List<Attribute> | attrs | 字段的Attribute |

开发者可以在此修改字段,如下列代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | for(FieldNode fieldNode : (List)classNode.fields)  {  if(fieldNode.name.equals("password"))  {  fieldNode.access = Opcodes.ACC\_PUBLIC;  }  } |

这个代码可以将password字段的访问级设为public.

关于访问级的具体说明请看附录.

开发者还可以为类添加字段,如:

|  |  |
| --- | --- |
| 1  2 | FieldNode newFieldNode = new FieldNode(Opcodes.ACC\_PUBLIC | Opcodes.ACC\_STATIC, "tempInt", "I", null, null);  classNode.fields.add(newFieldNode); |

这个代码是添加一个类型为int的静态公共(static public)字段.注意这里的类型使用的是Java类型描述符,如int的类型描述符是I,boolean的类型描述符是Z.具体的说明看附录.

方法节点(MethodNode)

字段节点被储存在类节点的methods字段中,由于它也是一个ArrayList,所以我们最好也还是用遍历集合的形式来获取方法节点.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | for(MethodNode methodNode : (List)classNode.methods)  {  if(methodNode.name.equals("getName")) //判断方法名是否匹配目标方法  {  //进行操作  }  } |

注:构造函数的名字一定是,静态构造函数的名字一定是

方法节点包含如下信息:

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 说明 |
| int | access | 访问级 |
| String | name | 方法名 |
| String | desc | 方法的返回值和参数 |
| String | signature | 签名,通常是null |
| List<String> | exceptions | 可能返回的异常 |
| List<AnnotationNode> | visibleAnnotations | 可见的注解 |
| List<AnnotationNode> | invisibleAnnotations | 不可见的注解 |
| List<Attribute> | attrs | 方法的Attribute |
| Object | annotationDefault | 默认的注解 |
| List[]<AnnotationNode> | visibleParameterAnnotations | 可见的参数注解 |
| List[]<AnnotationNode> | invisibleParameterAnnotations | 不可见的参数注解 |
| InsnList | instructions | 操作码列表 |
| List<TryCatchBlockNode> | tryCatchBlocks | try-catch块列表 |
| int | maxStack | 最大栈深度 |
| int | maxLocals | 最大局部变量数量 |
| List<LocalVariableNode> | localVariables | 局部变量节点 |

开发者对方法节点最长进行的操作是插入/修改/删去指令.这里就涉及到了Java字节码和栈帧.

首先,ASM的目的是简化开发者操作字节码的难度,因此不可能让用户直接对着byte变量撸,所以ASM在使用时用操作码来代替字节码,用最简单的话说,操作码是ASM对字节码的一个包装.如果你不想对着一个byte撸的话,就来对着操作码撸吧.  
操作码列表是方法节点中用于存储操作码的地方,这个集合中每一个元素都代表一行操作码.  
ASM将一行字节码封装为一个xxxInsnNode(某某操作码节点),如ALOAD被封装入VarInsnNode(变量操作码节点),INVOKEVIRTUAL被封装入MethodInsnNode(方法操作码节点).  
ASM之所以将字节码封装的如此详细,是为了简化开发者的操作,比如ALOAD指令除了它本身以外,后面还要附加一个参数值,用来指示局部变量位置.VarInsnNode便原生支持快速创建或修改一个带有此参数值的字节码.

所有xxxInsnNode都有一个共性,就是它们都是AbstractInsnNode(抽象操作码节点)的派生类.AbstractInsnNode只关心字节码的类型,而不关心参数值.

除AbstractInsnNode以外,所有的xxxInsnNode为:

|  |  |  |
| --- | --- | --- |
| 名称 | 说明 | 额外参数 |
| FieldInsnNode | 用于GETFIELD和PUTFIELD之类的字段操作的字节码. | String owner 字段所在的类 String name 字段的名称 String desc 字段的类型 |
| FrameNode | 栈映射帧的字节码 | 因为不了解,所以不敢乱说... |
| IincInsnNode | 用于IINC变量自加操作的字节码. | int var 目标局部变量的位置 int incr 要增加的数 |
| InsnNode | 一切无参数值操作的字节码.如ALOAD\_0,DUP 这里有个疑问,POP也是无参数值的字节码,但Javadoc 里没提到InsnNode支持这个操作, 笔误还是Bug? | 无 |
| IntInsnNode | 用于BIPUSH,SIPUSH和NEWARRAY这三个直接操作 整数的操作. | int operand 操作的整数值 |
| InvokeDynamicInsnNode | 用于Java7新增的INVOKEDYNAMIC操作的字节码 | String name 方法的名称 String desc 方法的描述 Handle bsm 句柄 Object[] bsmArgs 参数常量 |
| JumpInsnNode | 用于IFEQ或GOTO等转跳操作字节码 | LabelNode lable 目标lable |
| LabelNode | 一个用于表示转跳点的Label节点 | 无 |
| LdcInsnNode | 用于LDC等插入引用值的字节码 | Object cst 引用值 |
| LineNumberNode | 一个表示行号的节点 | int line 行号 LabelNode start 对应的 第一个Label |
| LookupSwitchInsnNode | 用于实现LOOKUPSWITCH操作的字节码 | LabelNode dflt default块对 应的Lable List<Integer> keys 值 List<LabelNode> labels 对应的Label |
| MethodInsnNode | 用于INVOKEVIRTUAL等传统方法调用操作的字节码. 不适用于Java7新增的INVOKEDYNAMIC | String owner 方法所在的类 String name 方法的名称 String desc 方法的描述 |
| MultiANewArrayInsnNode | 用于MULTIANEWARRAY操作的字节码 | String desc 类型 int dims 维数 |
| TableSwitchInsnNode | 用于实现TABLESWITCH操作的字节码 | int min 键的最小值 int max 键的最大值 LabelNode dflt default块对 应的Lable List<LabelNode> labels 对应的Label |
| TypeInsnNode | 用于实现NEW,ANEWARRAY和CHECKCAST等 类型相关的操作的字节码 | String desc 类型 |
| VarInsnNode | 用于实现ALOAD,ASTORE等局部变量操作的字节码 | int var 局部变量的编号 |

定位操作码的位置

如果你明确知道你想要的操作码在InsnList中的位置的话,你可以使用get方法来直接获取一个操作码,如:

|  |  |
| --- | --- |
| 1  2 | MethodNode methodNode = ...;  AbstractInsnNode ainNode = methodNode.instructions.get(4); |

这个代码是获得第5个操作码.

但在大多数情况下我们都无法确定操作码的具体位置,因此我们主要还是通过遍历操作码列表然后判断其关键特征来定位操作码,例如这个代码:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | MethodNode methodNode = ...;  for(AbstractInsnNode ainNode : methodNode.instructions.toArray())  {  if(ainNode.getOpcode() == Opcodes.BIPUSH && ((IntInsnNode)ainNode).operand == 64)  {  ....//进行操作  }  } |

它能定位一个"BIPUSH 64"字节码.有时一个程序会有多个相同的指令,这时开发者只能靠判断前后文识别其特征来定位,或者记下其命中次数然后设定在某一次进行操作.

(遍历操作码列表其实有2种方法,具体的讨论参见附录E)

替换一个操作码

替换操作简单到无脑,调用InsnList的set方法就能替换一个特定的操作码实例,如:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | MethodNode methodNode = ...;  for(AbstractInsnNode ainNode : methodNode.instructions.toArray())  {  if(ainNode.getOpcode() == Opcodes.BIPUSH && ((IntInsnNode)ainNode).operand == 64)  {  methodNode.instructions.set(ainNode, new VarInsnNode(Opcodes.ALOAD, 1));  }  } |

我们在定位了"BIPUSH 64"字节码的位置后,将封装它的操作码替换为一个新的VarInsnNode操作码,这个新操作码封装了"ALOAD 1"字节码.实现了将原程序中"将值设为64"替换为"将值设为局部变量1".

删去一个操作码

无脑到让人肾爆,只要你知道一个操作码实例,就能调用InsnList的remove方法将它隙间掉.

|  |  |
| --- | --- |
| 1 | methodNode.instructions.remove(xxx); |

插入操作码

插入操作码还稍微像回事,InsnList提供了以下几类方法用于插入字节码.  
add(AbstractInsnNode insn) 将一个操作码添加到这个InsnList的末尾  
insert(AbstractInsnNode insn) 将一个操作码插入到这个InsnList的开头  
insert(AbstractInsnNode location,AbstractInsnNode insn) 将一个操作码插入到另一个已知的操作码的下面  
insertBefore(AbstractInsnNode location,AbstractInsnNode insn) 将一个操作码插入到另一个已知的操作码的上面

例如下面的代码:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | MethodNode methodNode = ...;  for(AbstractInsnNode ainNode : methodNode.instructions.toArray())  {  if(ainNode.getOpcode() == Opcodes.BIPUSH && ((IntInsnNode)ainNode).operand == 64)  {  methodNode.instructions.insert(ainNode, new MethodInsnNode(Opcodes.INVOKEVIRTUAL, "java/awt/image/BufferedImage", "getWidth", "(Ljava/awt/image/ImageObserver;)I"));  methodNode.instructions.insert(ainNode, new InsnNode(Opcodes.ACONSTNULL));  methodNode.instructions.set(ainNode, new VarInsnNode(Opcodes.ALOAD, 1));  }  } |

这个代码的作用是将:  
BIPUSH 64  
替换为:  
ALOAD 1  
ACONSTNULL  
INVOKEVIRTUAL java/awt/image/BufferedImage.getWidth(Ljava/awt/image/ImageObserver;)I

输出字节码

当我们修改完一个类节点后,就该将它输出为字节码了.ASM通过ClassWriter类来输出字节码.  
下面这个代码可以将一个ClassNode解析为字节码.

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | static public byte[] getBytecode(ClassNode classNode)  {  ClassWriter classWriter = new ClassWriter(ClassWriter.COMPUTEFRAMES);//让ClassWriter自行计算最大栈深度和栈映射帧等信息  classNode.accept(classWriter);  return classWriter.toByteArray();  } |

至此,对象模型的使用介绍差不多就到这了...接下来是事件模型.

(PS:有时修改字节码会遇到栈映射帧(StackMap Frame)的验证问题,因为这个问题并非总是会遇到,所以我就写在了附录里.)

事件模型

我们之前提到,对象模型是一个经过封装的事件模型,换句话说事件模型才是本体,那我为什么还要把事件模型放在第二位?因为事件模型使用起来难度较大,不易理解.而且有些事件模型的操作还要依赖对象模型的思路(比如对操作码节点的操作).所以我放到了第二位.  
事件模型采用了设计模式中的访问者模式,访问者模式被誉为"最难学的设计模式"...  
故此,我先简单解释一下访问者模式,GoF在<设计模式>一书中对访问者模式的定义是"表示一个作用于某对象结构中的各元素的操作.它是你可以在不改各元素的类的前提下定义作用于这些元素的新操作".其实后半句对一流程序员来说才算萌点,我等蒟蒻看前半句就行了...  
访问者模式设定了这样的一个故事背景:"有M个元素和N种算法,每个算法都能作用于任意一个元素,并且在不同的元素上有不同的运行方式."在过去你可能会想到在每一个元素的类中添加N个方法,一个方法实现一个算法.但这样做会导致维护困难,代码混乱和耦合度过高等问题.因此有了访问者模式,现在有N个访问者,每个访问者拥有一个算法以及它的M种运行方式,当需要调用一个算法时,就让对应的访问者去访问元素,然后让访问者根据被访问对象自行选择算法.

依然无法理解的话就再举一个例子.以Minecraft为例,Block类的onBlockDestroyedByPlayer方法就是上文提到的算法,这个算法在不同的Block派生类中都有不同的运行方式(通过重写方法来实现),如果将它改造成访问者模式的话,那Block类的onBlockDestroyedByPlayer就会被去掉,取而代之的是增加一个accept方法,用来接收一个访问者的来访.我们会添加一个BlockDestroyedByPlayerVisitor类,并为它添加一个visit方法,用来回应一个Block类的同意访问请求.算法的实现全在visit方法中实现.  
你可能注意到我们没有直接让访问者去操作元素,而是先让元素接受(accept)访问者,然后访问者开始访问(visit)元素,这是为了避免暴露不必要的内部细节.因为算法位于元素类的外部,这意味着它无法访问元素的私有(private)字段,所以我们让元素通过调用visit方法的方式将必要的私有参数传递过去.以下是个代码例子:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Block block = ...;  BlockDestroyedByPlayerVisitor visitor = BlockDestroyedByPlayerVisitor.getInstance(); //我采用了单例模式,因为没必要大量创建算法的实例.  block.accept(visitor);  accept的内容为:  public void accept(BlockVisitor visitor) //通常来说,访问者都会实现同一个接口,或者统一派生自一个类  {  if(visitor instanceof BlockDestroyedByPlayerVisitor)  {  (BlockDestroyedByPlayerVisitor)visitor.visit(world,x,y,z,id);  }  } |

这仅仅只是个例子而已,事实上它有不少问题所以不能投入实用,在这里仅仅只是用于展示访问者模式的实例而已...

访问者模式的另一个好处就是可以用它来模拟事件系统,这也是这个操作模型被称为事件模型的原因.一个被访问者可以向访问者通过调用各种visit的方式引发各种事件,以此推进程序流程.

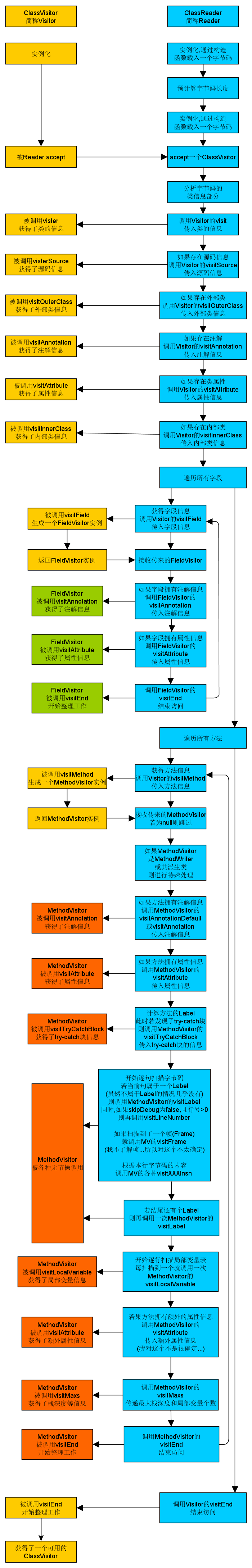
ASM的事件模型采用的访问者模式结合了这些优点,虽然学起来有些麻烦,但使用起来并不难.

类访问者(ClassVisitor)

ClassVisitor被定义为一个能接收并解析ClassReader发来的信息的类.当你让ClassVisitor访问ClassReader时,ClassReader会开始字节码的解析工作,并将结果源源不断通过调用各种方法输入ClassVisitor.其中只有visit方法只会并且一定会被调用一次,其它都不定.比如visitMethod方法,每当ClassReader解析出一个方法的字节码时,都会调用一次visitMethod方法.由它生成一个MethodVisitor(方法访问者).这个MethodVisitor不会消停下来,而是会被反馈回ClassReader,由ClassReader向MethodVisitor输入更多的信息(如操作码内容).这也是访问者模式的另一个优势,将工作分隔开,ClassVisitor只处理和类相关的事,方法的事情被外包给MethodVisitor处理.  
我们之前说过对象模型是事件模型的一个封装.其实,ClassNode就是ClassVisitor的一个派生类.它将ClassReader发来的信息分类储存.MethodNode也是MethodVisitor的一个派生类,它将ClassReader发来的操作码信息串成一个列表来储存.  
ClassWriter也是ClassVisitor的一个派生类,不过它不会储存信息,它会将收到的信息立即转译成字节码,并随时输出它们.  
ClassReader则与一切都无关系,如果你再深入了解访问者模式的话,你会知道访问者模式中还存在一个叫ObjectStructure的东西,可以看做所有元素的集合以及事件源.在这里ClassReader一定程度上充当了它的作用.

ClassReader解析过程

了解ClassReader解析过程有助于我们深入学习ASM的事件模型的使用方法.

(抱歉,这张图右上角有个错误,"实例化"因为手滑出现了两次...不过并不影响阅读)  
[](http://www.hakugyokurou.net/wordpress/wp-content/uploads/2013/03/graph2big.gif)

(大图下载地址:<http://sdrv.ms/14O1dM9>)

基本上,这张图解答了关于ClassVisitor,FieldVistor,MethodVisitor的大部分问题.比如各个方法的作用,何时被调用等.

如果使用事件模型来修改字节码,我们可以先创建一个ClassVisitor的派生类,建议继承ClassNode而不是直接继承ClassVisitor,这样可以复用一些ClassNode的功能.之后再创建新的MethodVisitor派生类(同样建议继承MethodNode).  
然后重写新的ClassVisitor的visitMethod方法,判断传来的方法的名字,如果是我们要修改的方法,就实例化一个新的MethodVisitor返回过去(别忘了把你的MethodVisitor添加入methods列表里).如果不是就调用基类方法.  
最后根据你要修改的代码,在新的MethodVisitor中重写相关的方法,继续用上文的BIPUSH 64举例,BIPUSH指令属于IntInsn,因此我们要重写visitIntInsn方法.判断传来的字节码类型(毕竟BIPUSH,SIPUSH和NEWARRAY都是IntInsn)以及参数(是否是64),之后如果满足条件就改为新的代码,不符合条件就交给基类方法处理.  
比如下列代码:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | /<em>\*</em>  \* 有时代码中存在多个BIPUSH 64,而我们却只要修改特定的一个.  \* 由于事件模型无法方便地分析代码的前后文(只能分析前文),  \* 所以这里采用的是计算命中次数.  \* 假设代码中存在2个以上的BIPUSH 64,而我们只修改第二个.  \*/  private int hit = 0;    @Override  public void visitIntInsn(final int opcode, final int operand)  {  if(opcode == Opcodes.BIPUSH && operand == 64)  {  if(hit++ != 2)  {  super.visitIntInsn(opcode, operand);  return;  }  instructions.add(new VarInsnNode(Opcodes.ALOAD, 1));  instructions.add(new InsnNode(Opcodes.ACONST\_NULL));  instructions.add(new MethodInsnNode(Opcodes.INVOKEVIRTUAL, "java/awt/image/BufferedImage", "getWidth", "(Ljava/awt/image/ImageObserver;)I"));  return;  }  super.visitIntInsn(opcode, operand);  } |

这个是MethodVisitor中的代码,ClassVisitor中的代码就没有什么可写的了...仅仅是个判断名字而已.

对于事件模型我能说的也只有这么多...因为我更喜欢对象模型,虽然说大触手们都爱用事件模型吧-w-...对于事件模型的使用,Aswang翻译的ASM官方手册中有更详细的解释.

另外那些代码没有经过实测= =如果你发现了问题请反馈给我...

结尾

"То робостью, то ревностью томим.  
Я вас любил так искренно, так нежно,  
Как дай вам бог любимой быть другим."  
-Александр Сергеевич Пушкин

这么快就到结尾了吗...这只是一个简单说明而已,仅仅只是一个Java字节码和ASM的简单说明,如果这两个东西用寥寥几千字就能说清楚的话,就不需要那些长篇的指导和手册了.  
无论如何,任何事情的学习都源于需求,实现于实践,别说是熟练,仅仅只是入门便需要数十小时的实践操作,学编程最好的途径是读别人的代码和写自己的代码,继续努力吧...

附录A - 访问级

访问级准确说是类/字段/方法的修饰名的集合.比如一个公共类的修饰符是public,一个公共接口的修饰符是public interfact.

ASM使用一个18位二进制数字来表示一个完整的访问级,即000000000000000000,其中第1位(即最右边的0)表示public修饰符,第4位表示static修饰符,因此一个static public类的访问级是000000000000001001.  
(也有可能是000000000000101001...高三党时间苦逼啊,无力详细测试了)

当然,ASM不会让我们用这么蠢的方式来设置访问级,ASM的Opcodes类已经定义了各个修饰符的常量.

|  |  |  |  |
| --- | --- | --- | --- |
| 名称(已略去ACC\_前缀) | 说明 | 16进制大小 | 适用 |
| PUBLIC | 公共 | 0x0001 | 类,方法,字段 |
| PRIVATE | 私有 | 0x0002 | 类,方法,字段 |
| PROTECTED | 内部保护 | 0x0004 | 类,方法,字段 |
| STATIC | 静态 | 0x0008 | 方法,字段 |
| FINAL | Final... | 0x0010 | 类,方法,字段 |
| SUPER | 将一个类标示为"超类", 定义看这里: <http://t.cn/zYm7PMv> | 0x0020 | 类 |
| SYNCHRONIZED | 需要资源锁方法 | 0x0020 | 方法 |
| VOLATILE | 需要同步的字段 | 0x0040 | 字段 |
| BRIDGE | 不了解...和泛型有关吧 | 0x0040 | 方法 |
| VARARGS | 参数不定的方法 | 0x0080 | 方法 |
| TRANSIENT | 不会被串行化的临时字段 | 0x0080 | 字段 |
| NATIVE | JNI引用的外部方法 | 0x0100 | 方法 |
| INTERFACE | 表明此为接口 | 0x0200 | 类 |
| ABSTRACT | 抽象 | 0x0400 | 类,方法 |
| STRICT | 让浮点数采用精确浮点 | 0x0800 | 方法 |
| SYNTHETIC | 和内部类以及匿名类有关 | 0x1000 | 类,方法,字段 |
| ANNOTATION | 表明此为注解 | 0x2000 | 类 |
| ENUM | 表明此为枚举 | 0x4000 | 似乎是字段? |
| DEPRECATED | ASM专用的一个东西? | 0x20000 | 类,方法,字段 |

当我们需要计算一个访问级时,简单地用这些修饰符常量相加就可以获得一个访问级了.

比如当我需要一个公共+超类+抽象的访问级时,用Opcodes.ACC\_PUBLIC + Opcodes.ACC\_SUPER + Opcodes.ACC\_ABSTRACT就可以获得一个访问级了.

分析访问级就需要用到位操作,这个就自行百度吧...

附录B - Java类型描述符

变量类型在字节码中有特殊的写法,比如int会被写为I,float会被写为F,long会被写为J等等...

以下为Java代码中的变量类型和其在Java字节码中的类型描述符.

|  |  |  |  |
| --- | --- | --- | --- |
| 变量类型 | 类型描述符 | 包装类 | 包装类类型描述符 |
| int | I(大写i) | Integer | Ljava/lang/Integer; |
| short | S | Short | Ljava/lang/Short; |
| long | J | Long | Ljava/lang/Long; |
| boolean | Z | Boolean | Ljava/lang/Boolean; |
| char | C | Character | Ljava/lang/Character; |
| byte | B | Byte | Ljava/lang/Byte; |
| float | F | Float | Ljava/lang/Float; |
| double | D | Double | Ljava/lang/Double; |
| void | V | Void | Ljava/lang/Void; |
| Object | L+类名(使用'/'作为 分隔符)+; 如: Ljava/lang/Object; Lorg/objectweb/asm/MethodVisitor; | / | / |
| String | Ljava/lang/String; | / | / |
|  | 数组写法: |  |  |
| X的N维数组 | N个[+X的类型描述符 | / | / |
| int[] | [I | / | / |
| byte[][] | [[B | / | / |
| String[] | [Ljava/lang/String; | / | / |

附录C - Java字节码分类

坑中...

附录D - 栈映射帧与字节码错误

栈映射帧(StackMap Frame)是Java6以后引入的一种验证机制,用于检验Java字节码的正确性.它的方式是记录每一个关键步骤时操作数栈的理论状态,并在实际运行时将实际状态和理论状态对比,如果状态不一样则代表出现了错误.具体的介绍可以[看这里](http://justfansty.i.sohu.com/blog/view/211545704.htm).

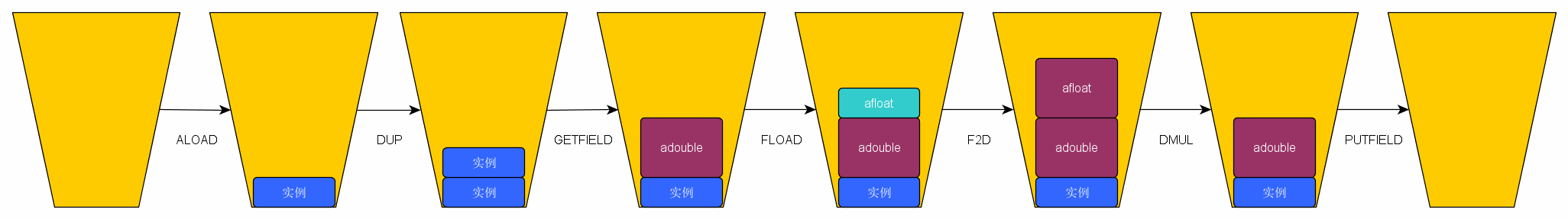
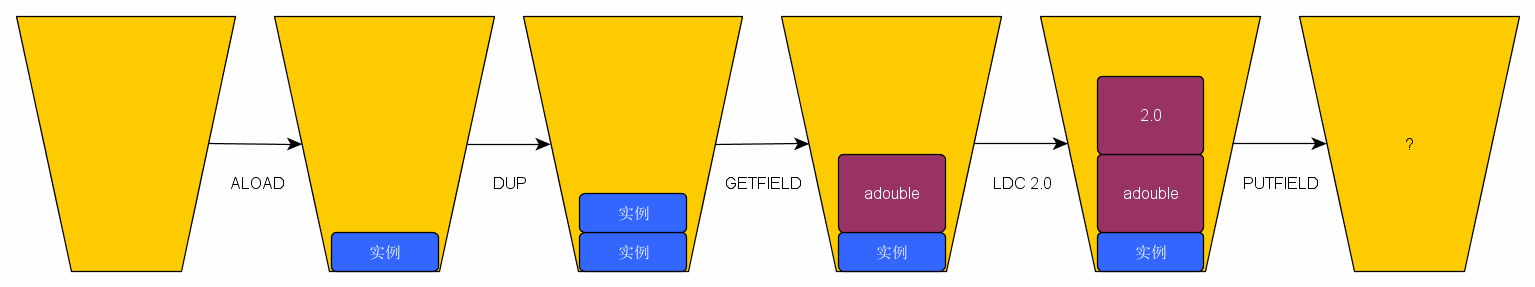
ASM可以自动计算栈映射帧,通常来说是极为有效的(尽管有人指出ASM自动计算栈映射帧会增加50%的额外运算,但谁在乎呢...),但偶尔你会遇到栈映射帧验证失败,比如:VerifyError: Inconsistent stackmap frames at branch target错误.

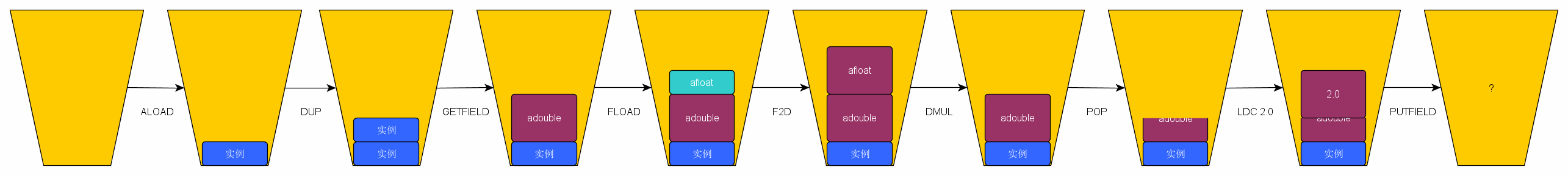
通常来说是有这几个原因导致:  
错误1.字节码编写错误,导致ASM计算失败  
ASM测不出你的错误,但JVM可以,比如代码:

|  |  |
| --- | --- |
| 1 | this.adouble \*= (double)afloat; |

其中,adouble是一个double类型字段,afloat是一个flaot类型的局部变量,位置为1.  
编译为字节码后是:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | ALOAD 0  DUP  GETFIELD apackage/Aclass.adouble : D  FLOAD 1  F2D  DMUL  PUTFIELD apackage/Aclass.adouble : D |

我们要把它修改成this.adouble = 2.0D,也许你乐观地以为只要把FLOAD 1换成LDC 2.0,然后去掉F2D和DMUL就可以了,但实际上如果你这样做的话,会在运行时导致VerifyError错误.  
这是因为对字节码操作指令不熟悉而导致的,ALOAD是读入一个实例引用(占1个字节的空间),DUP是复制栈顶的数据,GETFIELD是以栈顶的数据为实例,读取那个实例的字段(占1~2个字节,视类型而定),FLOAD是读入float(占1个字节),F2D是将栈顶的数据(假设为float类型)转换为double类型(占2个字节).DMUL是将栈顶的4个字节视为2个double数据并相乘,并将结果压回栈顶.PUTFIELD是将栈顶的2~3个字节视为2个数据(第一个视为实例,第二个视为值),并将值赋给实例中的字段.  
也许你现在看出问题了,原先的操作数栈的状态变化过程是:  
[](http://www.hakugyokurou.net/wordpress/wp-content/uploads/2013/03/graph3.gif)  
然而,我们修改后的字节码的运行状态是:  
[](http://www.hakugyokurou.net/wordpress/wp-content/uploads/2013/03/graph4.gif)  
感谢JVM的验证机制将我们拦了下来,PUTFIELD会把adouble的数值的后半部分视为一个实例引用,然后试图将2.0D写进里面,只有神知道会发生什么,ASM也不知道会发生什么,因此它算不出此时的栈映射图,我们运行时也自然会遇到错误.

这是一种情况,此外还有一种容易犯错的地方,就是忽视了数据的体积(也就是占的字节数).  
假如我们打算换一种方案,在DMUL后面加入POP(粗俗理解为弹出栈顶数据)和LDC 2.0,来达到写入的目的,但这样做的后果同样是验证错误.  
这也是因为对字节码的认识不充分,将POP理解为弹出栈顶数据实在是俗不可耐,它的正确定义是弹出栈顶的1个字节.double变量占2个字节,你只弹出了半个数据,还残留下半个double占在栈中,PUTFIELD又把那半个double当做实例,结果自然又是惨剧.  
[](http://www.hakugyokurou.net/wordpress/wp-content/uploads/2013/03/graph5.gif)  
正确的办法是使用POP2,它是从栈顶弹出2个字节,正好是1个double或long变量.

错误2:JDK版本错误  
VerifyError错误在JDK7较为常见,据说可以在启动时加入虚拟机参数-XX:-UseSplitVerifier来取消验证;也可以将编译环境换为JDK6.  
但我认为这种情况少之又少,ASM4.0的changelog中明确说明支持Java7.

错误3:ASM脑抽  
这个问题无解了,自行修正吧...  
我认为这个情况几乎不可能存在,十有九九九是你的字节码写错了...

附录E - Iterator VS ToArray&Foreach

遍历MethodNode的instructions有两种方法,一种是使用它自带的Iterator(遍历器),另一种是将它输出为一个数组,然后用foreach遍历一遍数组.

大多数大触都会下意识地选择Iterator,然而看下列代码:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | for(Iterator iterator = methodNode.instructions.iterator();iterator.hasNext();)  {  AbstractInsnNode ainNode = iterator.next();  if(ainNode.getOpcode() == Opcodes.GETFIELD && ((FieldInsnNode)ainNode).name.equal("afloat"))  {  //中心代码  methodNode.instructions.set(ainNode.getNext(), new LdcInsnNode(Float.valueOf(1.00F)));  }  } |

如果程序中有3个afloat = xxx,或者xxx = afloat.那么你可能认为中心代码会被运行3次,实际上它只会运行1次.

这是由于InsnList的set方法和内部的InsnListIterator的特性造成的(其实我认为与其说是特性,不如说是Bug),每一个AbstractInsnNode(简称节点)都有2个字段,上一个节点(prev)和下一个节点(next),set的原理是将记住目标节点的上一个节点(简称P节点)和下一个节点(简称N节点),然后将P节点的next设为新节点,新节点的prev设为P节点,N节点的prev设为新节点,新节点的next设为N节点,最后把旧节点的prev和next都设为null,这样新节点就连入了链表,旧节点就成了一个无法到达的变量,可以坐等被GC回收了,这听力来很美好,但也仅仅是听起来.  
事实上当set和Iterator用在一起时,set可以处理几乎所有的节点,但惟独不包括下一个节点(当前被遍历的节点的next).因为Iterator内部维持着2个字段,它们也叫next和prev,分别是当前被遍历节点的下一个节点和上一个节点.这时问题就来了,假设我们的Iterator正位于A节点上,A节点的下一个节点是B节点,我们通过methodNode.instructions.set(A.getNext(),ab)来将B节点修改为ab节点,修改结果很成功,ab节点指向了C节点(假设B的下一个节点是C节点),A节点指向了ab节点,B节点被孤立了出去.但我们忽略了一个问题,Iterator的next依然指向B节点,set方法忘了通知它.  
结果就是当Iterator继续开始遍历时,开入了一个死胡同,这个死胡同就是B节点,它不指向任何节点,也不被任何节点指向(除了闷在鼓里的Iterator),结果就是Iterator误以为已经遍历到尽头,从而停止遍历.

所以我们可以知道InsnList的Iterator实现的很失败,至少是set方法很失败.如果我们一定要用遍历器,那只能通过一个变量记住要修改的节点,然后在遍历结束后统一修改.

这个事情在数组中完全没有问题,因为我们是在遍历一个InsnList的深层副本,对本体的操作完全不会影响到深层副本,比如:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | for(AbstractInsnNode ainNode : methodNode.instructions.toArray())  {  if(ainNode.getOpcode() == Opcodes.GETFIELD && ((FieldInsnNode)ainNode).name.equal("afloat"))  {  //中心代码  methodNode.instructions.set(ainNode.getNext(), new LdcInsnNode(Float.valueOf(1.00F)));  }  } |

这个代码完全没有问题,它可以正常地被执行3次.

总之就是这个样子,我比较喜欢toArray转为数组后再遍历,但这也不代表我们就可以完全否定Iterator,它总会有用的...

附录F - Keep you stupid

ASM的目的是简化程序员操作字节码的难度,因此它绝对不可能再为你多添麻烦(ASM自身的bug除外...),众所周知Java字节码为了提高效率,将一种指令拆分成多种形式,比如LDC拥有三种形式,分别拥有不同的寻址能力,xLOAD有xLOAD\_0等形式,其速度比xLOAD 0要快,但在ASM中你无需考虑这个,所有形式的LDC都被简单视为LDC,所有xLOAD\_0都被视为xLOAD 0,ASM会在输出字节码时自动优化.

更令人遗憾的是,手工优化不但是无用功,而且反而会导致ASM错误.比如如果你将  
VarInsnNode(ALOAD,0)  
写为  
InsnNode(42) //42=ALOAD\_0  
看上去两者确实是等效的,在实际运行中也确实如此,但ASM的ClassWriter可不这么认为,如果你让ClassWriter自动计算栈映射帧(即加入COMPUTE\_FRAMES参数)那么ClassWriter会因为无法识别那个InsnNode(42)而出现谜のNPE,最终导致无法输出正确的字节码.

附录G - ClassCircularityError

这个Error的解释是"类循环引用",即A的父类是B,B的父类是A...这个看似不可能的事情却可能在ClassWriter的getCommonSuperClass方法中出现,其发生的原因和机理我现在也没发现...一个取巧的解决方式是自己创建个ClassWriter的派生类并重写getCommonSuperClass方法,自行判断其公共基类并返回值.