ASM 3.0

Java 字节码引擎库

版本	时间	作者
0.1	2011/10/30	aswang

1 简介

1.1 动机

程序分析，生成以及转换是很有用的技术手段，可以应用在很多场景下：

- 程序分析，涉及的范围很广，从简单的语法解析到完整的语义分析，也可用来发现程序中潜在的 bug，检测未使用的代码，以及反向工程等。
- 用来帮助编译器生成代码。包括传统的编译器，用在分布式编程中的内嵌的编译器，以及即时编译器等。
- 程序转换可以用来优化程序或者对程序进行混淆，或者在应用中插入调试代码或者性能监控代码，再或者面向切面编程等。

所有的这些技术可以用在任何编程语言中，只是针对不同的编程语言，实现的难度也不尽相同。对应 java 语言，这些技术可以用在 java 源代码或者编译过的 java 类文件上。能够对编译过的 java 类文件进行处理，有一个明显的优势，那就是不需要源代码。程序转换也因此可以使用在任何应用程序上，包括闭源的和商业的。针对编译后的代码的处理，另外一个优势就是可以在运行时，也就是在这些代码被加载进入虚拟机之前（在运行时生成代码并编译源代码是可能的，但是速度很慢，并且需要一个全功能的 java 编译器），对类文件进行分析、生成或者转换。对用户而言，这些优势就是内嵌编译器或者切面编程对用户是不可见的。

由于程序分析、生成和转换存在很多应用需求，所以人们针对不同的编程语言，这也包括 java，实现了很多工具来进行程序分析、生成和转换。ASM 就是这些工具中的一员，其主要是面向 java。ASM 被设计用以在运行时对 java 类进行生成和转换，当然也包括离线处理。ASM 库主要是工作在编译好的 java 类之上。ASM 被设计的短小精悍，尽量保证其速度很快，同时库的容量又很小。ASM 快速的目标就是避免在运行时动态生成 class 和转换对应用程序的速度影响。由于 ASM 库容量很小，因此，它可以被用在很多内存受限的环境中，同时避免导致小应用程序或者库的容量变大。

ASM 不仅仅是生成和转换编译后的 java 类的工具，而且它也是最有效的。它可以从 http://asm.objectweb.org 上下载得到。它的主要优势包括如下几方面：

- 它有一个很小，但是设计良好并且模块化的 API，且易于使用。
- 它具有很好的文档，并且还有 eclipse 插件。
- 它支持最新的 java 版本，java 6。
- 它很小，很快，很健壮。
- 它有一个很大的用户社区，可以给新用户提供支持。
- 它的开原许可允许你几乎以任何方式来使用它。

1.2 概述

1.2.1 范围

ASM 库的目标是生成、转换和分析编译后的 java 类，在虚拟机中以字节数组表示（它们是保存在硬盘上，在运行的时候被加载入 java 虚拟机）。基于这个目标，ASM 提供了对这些字节数组进行读取，写入和转换的工具，这些工具使用更高级别的概念，而非字节，比如数字常量，字符串，java 标识符，java 类型，java 类结构元素等等。注意，ASM 库的范围严格限于对 java 类进行读取，写入，转换以及分析。Java 类文件的加载过程超出了该范围。

1.2.2 模型

ASM 库提供了两套 API 用来生成、转换编译后的 java 类：核心的 API 是基于事件的，而 Tree API 是基于对象的。

在基于事件的模型中，一个 java 类表现为一系列事件，每一个事件代表了类中的一个元素，如类的头部，字段，方法声明，或者指令等。基于事件的 API 定义了一系列的事件以及这些事件发生的顺序，并且提供了一个类解析器，这个解析器在解析到一个类元素时，就会生成一个事件，与此对应的，类写入器在发生这些事件的时候生成编译后的类。

在基于对象的模型中，一个 java 类表现为一个对象树，每一个对象代表了类中的一部分，入类本身，一个字段，一个方法或者指令等。每个对象都具有指向其组成成分的引用。基于对象的模型提供了一种方式，用来将代表类的一系列事件转换为一颗对象树，反之亦然，将一颗对象树转换为一系列对等事件。换句话说，基于对象的模型是构建在基于事件的模型之上。

这两套 API 可以和 java 中的 XML 解析库 SAX（the Simple API for XML）和 DOM（Document Object Model） API 来对比。基于事件的 API 类似于 SAX，而基于对象的 API 类似于 DOM。基于对象的 API 构建在基于事件的 API 之上，类似于 DOM 可以在 SAX 之上使用。

ASM 之所以提供了两套 API，那是因为没有最好的 API，这两套 API 具有它的优势和缺点：

- 基于事件的 API 速度更快，需要的内存空间要小于基于对象的 API，因为没必要在内存中保留一个代表类结构的对象树（这个不同同时也存在于 SAX 和 DOM 之间）。
- 但是，通过基于事件的 API 来实现类的转换比较困难，因为在给定的时间，只有一个与该事件对应的元素可用。但是，基于对象的 API 能够在内存中保留整个类。

注意，这两套 API 在一个时刻都只能处理一个类，并且不依赖于其他类：因为没有维护关于类结构的信息，因此如果一个类的转换影响了其他类，那么由用户来决定是否修改其他类。

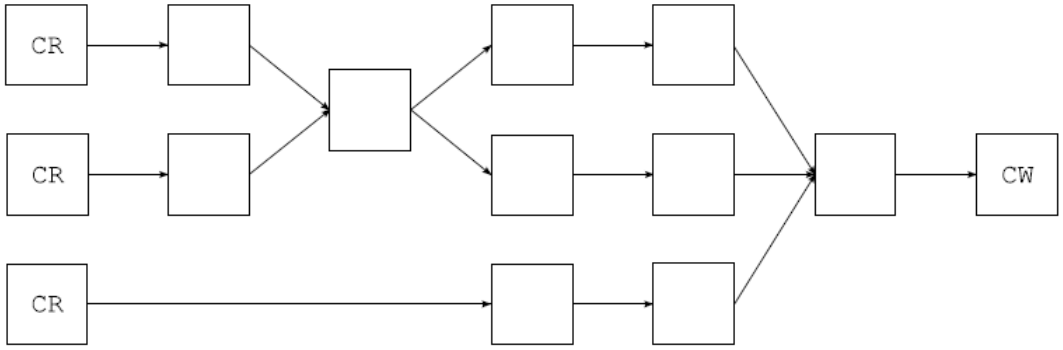
1.2.3 架构

ASM 应用有一个健壮的架构。基于事件的 API，在事件的产生者（类解析器）的周围，是事件的消费者（类写入器），以及一些预定义的事件过滤器，用户自定义的事件产生者、消费者和过滤器都可以被添加进去。可以按照以下的两个步骤来使用 API：

- 将事件的产生者、过滤器和消费者组件放置到架构的合适位置，
- 然后，启动事件的产生者进行类生产或者转换。

基于对象模型的 API 也有其架构：操作对象的类生成器和转换器可以被组合起来使用，它们之间的链接代表了转换的顺序。

尽管在典型的 ASM 应用中，大部分组件架构都是很简单的，但是可以想象一下像下面这幅图的复杂架构，箭头代表在链条中，基于事件或者对象模型中的类解析器、写入器和转换器之间的通信，在基于事件和对象模型之间需要经过合适的转换。



1.3 组织

ASM 库分为几个包，并且在不同的 jar 包中分发：

- org.objectweb.asm 和 org.objectweb.asm.signature 包定义了基于事件的 API，并且提供了类解析组件和写入组件。它们包含在 asm.jar 归档中。
- org.objectweb.asm.util 包，包含在 asm-util.jar 归档中，提供了一些基于核心 API 的工具，可以用来协助开发和调试 ASM 应用。
- org.objectweb.asm.commons 包提供了几个预定义类转换组件，大部分基于核心 API，它包含在 asm-common.jar 归档中。
- org.objectweb.asm.tree 包，包含在 asm-tree.jar 归档中，定义了基于对象模型的 API，同时也提供了一些工具对基于事件的和基于对象模型的展现进行转换。
- org.objectweb.asm.tree.analysis 包提供了一个类分析框架，以及几个预定义好的类分析器，基于 tree API。它包含在 asm-analysis.jar 归档中。

文档被组织为两部分，第一部分，覆盖了核心API，如asm，asm-util和asm-commons归档。第二部分覆盖了tree API（基于对象模型的API），如asm-tree和asm-analysis归档。每部分包含至少一章用来介绍与类相关的API内容，一章与方法相关的API，一章与注解、泛型等相关的API。每章都讲解了编程接口，以及相关的工具，和预定义好的组件等。所有这些例子的源

代码可在该地址下载：http://asm.objectweb.org/。

字体约定

Italic 斜体 用来强调句子中的元素。
等宽字体 表示代码片段
等宽粗体 表示强调代码元素
Italic 等宽斜体 表示代码中的变量部分和标签

1.4 致谢

我非常感谢 François Horn 在该文档拟定过程中的珍贵的评论，这帮助改善了文档的结构和可读性。

第一部　核心 API

2 类

这一章解释了如何使用 ASM 核心 API 来生成和转换编译后的 java 类。首先，结合一些说明性的例子，解释编译后的 java 类的相关结构，以及与之对应的 ASM 接口、组件和其他用于生成和转换的工具。关于方法，注解和泛型等内容，将在下一章节介绍。

2.1 结构

2.1.1 概述

一个编译后的 java 类的结构还是比较简单的。与被编译为本地代码的应用程序不同的是，编译后的 java 类保留了结构信息以及几乎所有的在源代码中定义的符号名称。事实上，编译后的 java 类包含以下几部分：

- 一个用于描述修饰符（public 或者 private），类名，父类名称，所实现的接口名称以及类注解的段。
- 一个用于描述类中字段信息的段。每个这样的段中都描述了字段的修饰符，名称，类型以及与该字段相关的注解。
- 一个用于描述方法和构造方法的段。每个段描述了方法的名称，返回值，参数类型以及方法的注解。同时也包含了方法编译后的字节码序列。

然而，在源代码和编译后的代码之间，还是存在一些不同：

- 一个编译后的 java 类仅仅只描述一个类信息，但是一个 java 源文件可以包含几个 java 类。例如，一个源文件可以定义一个包含内部类的 java 类，而编译后将会成为两个类文件，其中一个是主要的 java 类，另外一个是内部类。在主要的类中包含了指向内部类的引用，同时在主要的 java 类的方法中定义的内部类也会包含一个指向该 java 方法的引用。
- 一个编译后的 java 类不包含注释，当然，可以包含与类、字段、方法或者代码相关的属性，而这些属性可以用来关联一些额外的信息。在 java 5 中引入了注解以后，这些注解也可以实现同样的目的，因此，这些属性就变得不那么重要了。
- 一个编译后的 java 类不包含 package 和 import 段，因此，在编译后的类中，所有的类型名称都必须使用全路径。

另外在结构上，两者还有一个非常重要的不同，那就是一个编译后的 java 类包含一个常量池段。这个常量池是一个数组，它包含了在类中定义的所有数字，字符串以及类型常量。这些常量只在常量池中定义一次，在类的其它地方都是通过它们的索引来引用。让人高兴的是，ASM 隐藏了与常量池相关的细节，你就不必为此而烦恼了。图 2.1 总结了一个编译后的 java 类的整体结构。类的具体结构请参看 java 虚拟机规范第四段。

Modifiers, name, super class, interfaces	
Constant pool: numeric, string and type constants	
Source file name (optional)	
Enclosing class reference	
Annotation*	
Attribute*	
Inner class*	Name
Field*	Modifiers, name, type
	Annotation*
	Attribute*
Method*	Modifiers, name, return and parameter types
	Annotation*
	Attribute*
	Compiled code

图表 2.1 编译后的 java 类整体结构（*表示 0 或者更多）

另外一个不同，就是 java 的类型在源代码和编译后的类中是不同的，下一部分将介绍它们的表现方式。

2.1.2 内部名称

在很多情况下，一个类型限于一个 java 类或者结构表示的类型，例如，一个类的父类，一个类所实现的接口，一个方法所抛出的异常（不可能是基本类型）或者数组，这些都是类或者接口类型。这些类型在编译后的类中以内部名称表示。一个类的内部名称就是这个类的全路径名称，将包名中的点号替换为/。例如，String 的内部名称为 java/lang/String。

2.1.3 类型描述符

内部名称仅用作一个类或者接口的类型，所有其他的，如字段类型，java 基本类型都是以类型描述符来表示的，见图 2.2

Java type	Type descriptor
boolean	Z
char	C
byte	B
short	S
int	I
float	F
long	J
double	D
Object	Ljava/lang/Object;
int[]	[I
Object[][]	[[Ljava/lang/Object;

图 2.2 类型描述符

基本类型的描述符：Z 表示 boolean，C 表示 char，B 表示 byte，I 表示 int，F 表示 float，J 表示 long，D 表示 double。一个类的描述符就是这个类的内部名称，在前面加上一个 L，在后面加上一个分号即可。例如，String 的类型描述符就是 Ljava/lang/String.最后，一个数组的类型描述符就是一个中括号[后面跟上数组元素的类型描述符。

2.1.4 方法描述符

一个方法描述符就是一个包含参数类型的描述符，以及方法返回类型描述符的字符串。一个方法描述符以一个左括号开始，然后跟上每个参数的描述符，然后是一个右括号，最后就是返回值的类型描述符，如果一个方法的返回值是 void，那么返回值的类型描述符就是 V（一个方法描述符不包含这个方法的名称以及参数的名称）。

Method declaration in source file	Method descriptor
void m(int i, float f)	(IF)V
int m(Object o)	(Ljava/lang/Object;)I
int[] m(int i, String s)	(ILjava/lang/String;)[I
Object m(int[] i)	([I)Ljava/lang/Object;

图 2.3 方法描述符示例

一旦你知道了类型描述符如何工作，那么理解方法描述符很容易。例如，(I)I 描述了这样一个方法，它有一个 int 类型的参数，以及一个 int 返回值。图 2.3 给出了几个方法描述符的例子。

2.2 接口和组件

2.2.1 表现（Presentation）

生成和转换编译后的类的 ASM API 是基于 ClassVisitor 接口的（见图 2.4）。在这个接口中的每一个方法都与类文件中有着相同名称的段相对应(见图 2.1)。在访问类结构中简单的段时，是通过调用一个独立的方法来实现的，该方法的参数就是该段相关的内容，该方法的返回值为 void。对长度任意并且较复杂的段进行访问时，是通过一个初始化方法返回一个辅助的 visitor 接口来实现，例如 visitAnnotation，visitField 以及 visitMethod，它们都返回与之对应的接口 AnnotationVisitor，FieldVisitor 以及 MethodVisitor。

这些规则也同样适用于这些辅助接口。例如，在 FieldVisitor 接口中的每个方法，都与类文件结构中与该名称（Field）对应的子结构对应（见图 2.5），并且 visitAnnotation 并会一个辅助的 AnnotationVisitor 接口，与 ClassVisitor 中的 AnnotationVisitor 相同。关于这些辅助接口的创建和使用，将在下一章节介绍，这一章主要限于那些简单的问题，使用 ClassVisitor 接口就可以解决的。

```
public interface ClassVisitor {
  void visit(int version, int access, String name, String signature,
      String superName, String[] interfaces);
  void visitSource(String source, String debug);
  void visitOuterClass(String owner, String name, String desc);
  AnnotationVisitor visitAnnotation(String desc, boolean visible);
  void visitAttribute(Attribute attr);
  void visitInnerClass(String name, String outerName, String innerName,
      int access);
  FieldVisitor visitField(int access, String name, String desc,
      String signature, Object value);
  MethodVisitor visitMethod(int access, String name, String desc,
      String signature, String[] exceptions);
  void visitEnd();
}
```

图 2.4 ClassVisitor 接口

```
public interface FieldVisitor {
  AnnotationVisitor visitAnnotation(String desc, boolean visible);
  void visitAttribute(Attribute attr);
  void visitEnd();
}
```

图 2.5 FieldVisitor 接口

对 ClassVisitor 接口中方法的调用必须遵循下面文档定义的顺序，该文档定义在 ClassVisitor

接口的 Javadoc 中。

visit visitSource? visitOuterClass? (visitAnnotation | visitAttribute)*(visitInnerClass | visitField | visitMethod)*

visitEnd

这就意味着 visit 必须被第一个调用，然后调用 visitSource 方法，最多调用一次，再接着是 visitOuterClass，然后再调用任意次数的 visitAnnotation 或者 visitAttribute 方法，接着可以调用任意次数的 visitInnerClass，visitiField 或者 visitMethod，顺序不限，在最后，调用 visitEnd 方法。

在 ClassVisitor 接口的基础上，ASM 提供了三个组件来生成和转换类：

- 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 可以被看做是一个事件产生者。
- 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。它可以被看做是一个事件消费者。
- 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。

接下来，将结合具体的例子来展示如何使用这些组件来生成和转换类。

2.2.2 解析类

解析一个已存在的类仅需要 ClassReader 这个组件。下面让我们以一个实例来展示如何解析类。假设，我们想要打印一个类的内容，我们可以使用 javap 这个工具。第一步，实现 ClassVisitor 这个接口，用来打印类的信息。下面是一个简单的实现：

```
public class ClassPrinter implements ClassVisitor {
    public void visit(int version, int access, String name,
        String signature, String superName, String[] interfaces) {
        System.out.println(name + " extends " + superName + " {");
    }
    public void visitSource(String source, String debug) {
    }
    public void visitOuterClass(String owner, String name, String desc) {
    }
    public AnnotationVisitor visitAnnotation(String desc,
        boolean visible) {
        return null;
    }
    public void visitAttribute(Attribute attr) {

    }
    public void visitInnerClass(String name, String outerName,
        String innerName, int access) {
```

```
    }
    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 visitEnd() {
        System.out.println("}");
    }
}
```

第二步，将 ClassPrinter 和 ClassReader 结合起来，这样，ClassReader 产生的事件就可以被我们的 ClassPrinter 消费了：

```
ClassPrinter cp = new ClassPrinter();
ClassReader cr = new ClassReader("java.lang.Runnable");
cr.accept(cp, 0);
```

上面的第二行代码创建了一个 ClassReader 来解析 Runnable 类。最后一行代码中的 accept 方法解析 Runnable 类的字节码，并且调用 cp 上对应的方法。结果如下：

```
java/lang/Runnable extends java/lang/Object {
run()V
}
```

注意，这里有多种方式来构造一个 ClassReader 的实例。可以通过类名，例如上面的例子，或者通过类的字节数组。或者类的输入流。类的输入流可以通过 ClassLoader 的 getResourceAdStream 方法：

```
cl.getResourceAsStream(classname.replace('.', '/') + ".class");
```

2.2.3 生成类

生成一个类只需要 ClassWriter 组件即可。下面将使用一个例子来展示。考虑下面的接口:

```
package pkg;
public interface Comparable extends Mesurable {
    int LESS = -1;
    int EQUAL = 0;
    int GREATER = 1;
    int compareTo(Object o);
}
```

上面的类可以通过调用 ClassVisitor 的 6 个方法来生成：

```
ClassWriter cw = new ClassWriter(0);
cw.visit(V1_5, ACC_PUBLIC + ACC_ABSTRACT + ACC_INTERFACE,
    "pkg/Comparable", null, "java/lang/Object",
    new String[] { "pkg/Mesurable" });
cw.visitField(ACC_PUBLIC + ACC_FINAL + ACC_STATIC, "LESS", "I",
    null, new Integer(-1)).visitEnd();
cw.visitField(ACC_PUBLIC + ACC_FINAL + ACC_STATIC, "EQUAL", "I",
    null, new Integer(0)).visitEnd();
cw.visitField(ACC_PUBLIC + ACC_FINAL + ACC_STATIC, "GREATER", "I",
    null, new Integer(1)).visitEnd();
cw.visitMethod(ACC_PUBLIC + ACC_ABSTRACT, "compareTo",
    "(Ljava/lang/Object;)I", null, null).visitEnd();
cw.visitEnd();
byte[] b = cw.toByteArray();
```

第一行代码用于创建一个 ClassWriter 实例，由它来构建类的字节数组（构造方法中的参数将在后面章节介绍）。

首先，通过调用 visit 方法来定义类的头部。其中，V1_5 是一个预先定义的常量（与定义在 ASM Opcodes 接口中的其它常量一样），它定义了类的版本，Java 1.5.ACC_XX 常量与 java 中的修饰符对应。在上面的代码中，我们指定了类是一个接口，因此它的修饰符是 public 和 abstract（因为它不能实例化）。接下来的参数以内部名称形式定义了类的名称，（见 2.1.2 章节）。因为编译过的类中不包含 package 和 import 段，因此，类名必须使用全路径。接下来的参数与泛型对应（见 4.1 章节）。在上面的例子中，它的值为 null，因为这个接口没有使用泛型。第五个参数指定了父类，也是以内部形式（接口隐式地继承自 Object）。最后一个参数指定了该接口所继承的所有接口，该参数是一个数组。

接下来三次调用 visitField 方法，都是用来定义接口中三个字段的。visitField 方法的第一个参数是描述字段的访问修饰符。在这里，我们指定这些字段为 public，final 和 static。第二个参数是字段的名称，与在源代码中的名称一样。第三个参数，以类型描述符的形式指定了字段的类型。上面的字段是 int 类型，因此它的类型描述符是 I。第四个参数与该字段的泛型对应，在这里为空，因为这个字段没有使用泛型。最后一个参数是这些字段的常量值，这个参数只能针对常量字段使用，如 final static 类型的字段。对于其他字段，它必须为空。因为这里没有使用注解，所以没有调用任何 visitAnnotation 和 visitAttribute 方法，而是直接调用返回的 FieldVisitor 的 visitEnd 方法。

visitMethod 方法是用来定义 compareTo 方法的。该方法的第一个参数也是定义访问修饰符的，第二个参数是方法的名称，在源代码中指定的。第三个参数是该方法的描述符，第三个参数对应泛型，这里仍然为空，因为没有使用泛型。最后一个参数是指定该方法所声明的异常类型数组，在这个方法中为 null，因为 compareTo 方法没有声明任何异常。visitMethod 方法返回一个 MethodVisitor（参见图 3.4），它可以用来定义方法的注解和属性，以及方法的代码。在这里没有注解，因为这个方法是抽象的，因此我们直接调用了 MethodVisitor 的 visitEnd 方法。

最后，调用 ClassWriter 的 visitEnd 方法来通过 cw 类已经完成，然后调用 toByteArray 方法，返回该类的字节数组形式。

使用生成的类

前面获取的字节数组可以保存到 Comparable.class 文件中，以便在以后使用。此外它也可以被 ClassLoader 动态加载。可以通过继承 ClassLoader，并重写该类的 defineClass 方法来实现自己的 ClassLoader：

```
class MyClassLoader extends ClassLoader {
    public Class defineClass(String name, byte[] b) {
        return defineClass(name, b, 0, b.length);
    }
}
```

然后可以通过下面的代码来加载：

```
Class c = myClassLoader.defineClass("pkg.Comparable", b);
```

另一种加载生成的类的方法是通过定义一个 ClassLoader 的子类，并重写其中的 findClass 方法来生成需要的类：

```
class StubClassLoader extends ClassLoader {
    @Override
    protected Class findClass(String name)
        throws ClassNotFoundException {
        if (name.endsWith("_Stub")) {
            ClassWriter cw = new ClassWriter(0);
            ...
            byte[] b = cw.toByteArray();
            return defineClass(name, b, 0, b.length);
        }
        return super.findClass(name);
    }
}
```

实际上使用生成类的方式取决于使用的上下文，它超出了 ASM API 的范围。如果你打算写一个编译器，那么类的生成过程将被一个即将被编译的类的抽象的语法树驱动，并且生成的类将被保存到磁盘上。如果你打算编写一个动态的类代理生成工具或者在面向切面编程中使用，那么选择 ClassLoader 比较合适。

2.2.4 转换类

到目前为止，ClassReader 和 ClassWriter 都是独立使用。手工产生事件，然后被 ClassWriter 直接消费，或者对称地，事件由 ClassReader 产生，然后手工地消费，如通过一个自定义的 ClassVisitor 来实现。当把这些组件组合在一起使用时，将变得很有趣。第一步，将 ClassReader 产生的事件导入到 ClassWriter，结果就是类将被 ClassReader 解析，然后再由 ClassWriter 重组为 Class。

```
byte[] b1 = ...;
ClassWriter cw = new ClassWriter();
```

```
ClassReader cr = new ClassReader(b1);
cr.accept(cw, 0);
byte[] b2 = cw.toByteArray(); // b2 represents the same class as b1
```

当然，有趣的并不是这个过程本身（因为有更简单的方式来复制一个字节数组）。但是，接下来介绍的 ClassAdapter，它处于 ClassReader 和 ClassWriter 之间，将会带来变化：

```
byte[] b1 = ...;
ClasssWriter cw = new ClassWriter();
ClassAdapter ca = new ClassAdapter(cw); // ca forwards all events to cw
ClassReader cr = new ClassReader(b1);
cr.accept(ca, 0);
byte[] b2 = cw.toByteArray(); // b2 represents the same class as b1
```

与上面代码对应的结构图如如 2.6.在下面的图中，组件以方形表示，事件以箭头表示（在序列图中是一个垂直的时间线）。

图 2.6 转换链

执行结果没有任何变化，因为这里使用的 ClassAdapter 事件过滤器没有过滤任何东西。但是，现在可以重写这个类来过滤一些事件，以实现转换类。例如，考虑下面这个 ClassAdapter 的子类：

```
public class ChangeVersionAdapter extends ClassAdapter {
    public ChangeVersionAdapter(ClassVisitor cv) {
        super(cv);
    }
    @Override
    public void visit(int version, int access, String name,
        String signature, String superName, String[] interfaces) {
        cv.visit(V1_5, access, name, signature, superName, interfaces);
    }
}
```

这个类仅重写了 ClassAdapter 的一个方法。因此，所有的调用都未经过改变直接传递给了 ClassVisitor 实例 cv，cv 通过构造方法传递给自定义的 ClassAdapter，除了 visit 方法，visit 方法修改了类的版本号。对应的序列图如下：

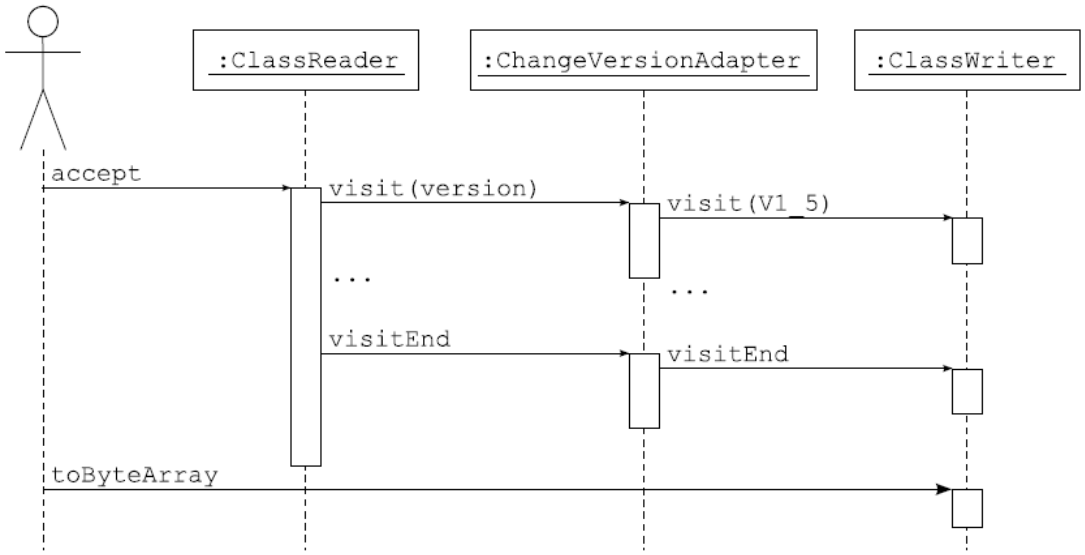


图 2.7

可以通过修改 visit 方法的其它参数来实现其它转换，而不仅仅是修改类的版本号。例如，你可以给类增加一个借口。当然也可以修改类的名称，但是这需要修改很多东西，而不只是修改 visit 方法中类的名称。实际上，类名可能在很多地方存在，所有这些出现的地方都需要修改。

优化

前面的转换只改变了原始类中的四个字节。尽管如此，通过上面的代码，b1 被完整的解析，产生的事件被用来从头构造 b2，尽管这样做不高效。另一种高效的方式是直接复制不需要转换的部分到 b2，这样就不需要解析这部分同时也不产生对应的事件。ASM 会自动地对下面的方法进行优化：

- 如果 ClassReader 检测到一个 MethodVisitor 直接被 ClassVisitor 返回，而这个 ClassVisitor（如 ClassWriter）是通过 accept 的参数直接传递给 ClassReader，这就意味着这个方法的内容将不会被转换，并且对应用程序也是不可见的。
- 在上面的情形中，ClassReader 组件不会解析这个方法的内容，也不会产生对应的事件，而只是在 ClassWriter 中复制该方法的字节数组。

这个优化由 ClassReader 和 ClassWriter 来执行，如果它们拥有彼此的引用，就像下面的代码：

```
byte[] b1 = ...
ClassReader cr = new ClassReader(b1);
ClassWriter cw = new ClassWriter(cr, 0);
ChangeVersionAdapter ca = new ChangeVersionAdapter(cw);
cr.accept(ca, 0);
byte[] b2 = cw.toByteArray();
```

经过优化，上面的代码将比前面例子中的代码快两倍。因为 ChangeVersionAdapter 没有转换任何方法。对于转换部分或者所有方法而言，这种对速度的提高虽然很小，但确实显著的，可以达到 10%到 20%。不幸地是，这种优化需要复制在原始类中定义的所有常量到转换后的类中。这对于在转换中增加字段，方法或者指令什么的不是一个问题，但是相对于未优化的情形，这会导致在大的类转换过程中删除或者重命名很多类的元素。因此，这种优化适合于

需要添加代码的转换。

使用转换后的类

转换后的类 b2 可以保存到磁盘或者被 ClassLoader 加载，如前面章节描述的。但是在一个 ClassLoader 中只能转换被该 ClassLoader 加载的类。如果你想转换所有的类，你需要把转换的代码放置到一个 ClassFileTransformer 中，该类定义在 java.lang.instrment 包中（可以参看该报的文档获得详细信息）：

```
public static void premain(String agentArgs, Instrumentation inst) {
    inst.addTransformer(new ClassFileTransformer() {
        public byte[] transform(ClassLoader l, String name, Class c,
            ProtectionDomain d, byte[] b)throws IllegalClassFormatException {
            ClassReader cr = new ClassReader(b);
            ClassWriter cw = new ClassWriter(cr, 0);
            ClassVisitor cv = new ChangeVersionAdapter(cw);
            cr.accept(cv, 0);
            return cw.toByteArray();
            }
    });
}
```

2.2.5 移除类成员

前面例子中用来修改类版本号的方法也可以用在 ClassVisitor 接口中的其它方法上。例如，通过修改 visitField 和 visitMethod 方法中的 access 核 name，你可以修改一个字段或者方法的访问修饰符和名称。更进一步，除了转发修改该参数的方法调用，你也可以选择不转发该方法调用，这样做的效果就是，对应的类元素将被移除。

例如，下面的类适配器将移除外部类和内部类，同时移除源文件的名称（修改过的类仍然是功能完整的，因为这些元素仅用作调试）。这主要是通过保留 visit 方法为空来实现。

```
public class RemoveDebugAdapter extends ClassAdapter {
    public RemoveDebugAdapter(ClassVisitor cv) {
        super(cv);
    }
    @Override
    public void visitSource(String source, String debug) {
    }
    @Override
    public void visitOuterClass(String owner, String name, String desc) {
    }
    @Override
    public void visitInnerClass(String name, String outerName,
        String innerName, int access) {
```

```
    }
}
```

上面的策略对字段和方法不起作用，因为 visitField 和 visitMethod 方法必须返回一个结果。为了移除一个字段或者一个方法，你不能转发方法调用，而是返回一个 null。下面的例子，移除一个指定了方法名和修饰符的方法（单独的方法名是不足以确定一个方法，因为一个类可以包含多个相同方法名的但是参数个数不同的方法）：

```
public class RemoveMethodAdapter extends ClassAdapter {
    private String mName;
    private String mDesc;
    public RemoveMethodAdapter(
        ClassVisitor cv, String mName, String mDesc) {
        super(cv);
        this.mName = mName;
        this.mDesc = mDesc;
    }
    @Override
    public MethodVisitor visitMethod(int access, String name,
        String desc, String signature, String[] exceptions) {
        if (name.equals(mName) && desc.equals(mDesc)) {
            // do not delegate to next visitor -> this removes the method
            return null;
        }
        return cv.visitMethod(access, name, desc, signature, exceptions);
    }
}
```

2.2.6 增加类成员

除了传递较少的方法调用，你也可以传递更多的方法调用，这样可以实现增加类元素。新的方法调用可以插入到原始方法调用之间，同时 visitXxx 方法调用的顺序必须保持一致（参看 2.2.1）。

例如，如果你想给类增加一个字段，你需要在原始方法调用之间插入一个 visitField 调用，并且你需要将这个新的调用放置到类适配器的其中一个 visit 方法之中（这里的 visit 是指以 visit 打头的方法）。你不能在方法名为 visit 的方法中这样做，因为这样会导致后续对 visitSource，visitOuterClass，visitAnnotation 或者 visitAttribute 方法的调用，这样做是无效的。同样，你也不能将对 visitField 方法的调用放置到 visitSource，visitOuterClass，visitAnnotation 或者 visitAttribute 方法中。可能的位置是 visitInnerClass，visitField，visitMethod 和 visitEnd 方法。

如果你将这个调用放置到 visitEnd 中，字段总会被添加，除非你添加了显示的条件，因为这个方法总是会被调用。如果你把它放置到 visitField 或者 visitMethod 中，将会添加好几个字段，因为对原始类中每个字段或者方法的调用都会导致添加一个字段。两种方案都能实现，如何使用取决于你的需要。例如，你恶意增加一个单独的 counter 字段，用来统计对某个对象的调用次数，或者针对每个方法，添加一个字段，来分别统计对每个方法的调用。

注意：事实上，添加成员的唯一正确的方法是在 visitEnd 方法中增加额外的调用。同时，一个类不能包含重复的成员，而确保新添加的字段是唯一的方法就是比较它和已经存在的成员，这只能在所有成员都被访问之后来操作，例如在 visitEnd 方法中。程序员一般不大可能会使用自动生成的名字，如_counter$或者_4B7F_可以避免出现重复的成员，这样就不需要在 visitEnd 中添加它们。注意，如在第一章中讲的，tree API 就不会存在这样的限制，使用 tree API 就可以在转换的任何时间点添加新成员。

为了展示上面的讨论，下面是一个类适配器，用来给一个类增加一个字段，除非这个字段已经存在：

```
public class AddFieldAdapter extends ClassAdapter {
    private int fAcc;
    private String fName;
    private String fDesc;
    private boolean isFieldPresent;
    public AddFieldAdapter(ClassVisitor cv, int fAcc, String fName,
        String fDesc) {
        super(cv);
        this.fAcc = fAcc;
        this.fName = fName;
        this.fDesc = fDesc;
    }
    @Override
    public FieldVisitor visitField(int access, String name, String desc,
        String signature, Object value) {
        if (name.equals(fName)) {
            isFieldPresent = true;
        }
        return cv.visitField(access, name, desc, signature, value);
    }
    @Override
    public void visitEnd() {
        if (!isFieldPresent) {
            FieldVisitor fv = cv.visitField(fAcc, fName, fDesc, null, null);
            if (fv != null) {
                fv.visitEnd();
            }
        }
        cv.visitEnd();
    }
}
```

这个字段是在 visitEnd 方法中添加的。重写 visitField 方法不是为了修改已经存在的字段，而是为了检测我们希望添加的字段是否已经存在。注意，在调用 fv.visitEnd 之前，我们测试了 fv 是否为空，如我们前面所讲，一个 class visitor 的 visitField 方法可以返回 null。

2.2.6 转换链

到目前为止，我们看到了一些有 ClassReader，一个类适配器和 ClassWriter 组成的转换链。当然，也可以将多个类适配器连接在一起，来实现更复杂的转换链。链接多个类适配器运行你组合多个独立的类转换，以实现更复杂的转换。注意，一个转换链条没必要是线性的，你可以编写一个 ClassVisitor，然后同时转发所有的方法调用给多个 ClassVisitor：

```
public class MultiClassAdapter implements ClassVisitor {
    protected ClassVisitor[] cvs;
    public MultiClassAdapter(ClassVisitor[] cvs) {
        this.cvs = cvs;
    }
    @Override public void visit(int version, int access, String name,
        String signature, String superName, String[] interfaces) {
        for (ClassVisitor cv : cvs) {
            cv.visit(version, access, name, signature, superName, interfaces);
        }
    }
    ...
}
```

相对地，多个类适配器也可以将方法调用都委托给相同的 ClassVisitor（这需要额外的小心，以确保 visit 和 visitEnd 方法只被调用一次）。如图 2.8 这样的转换链也是可能地。

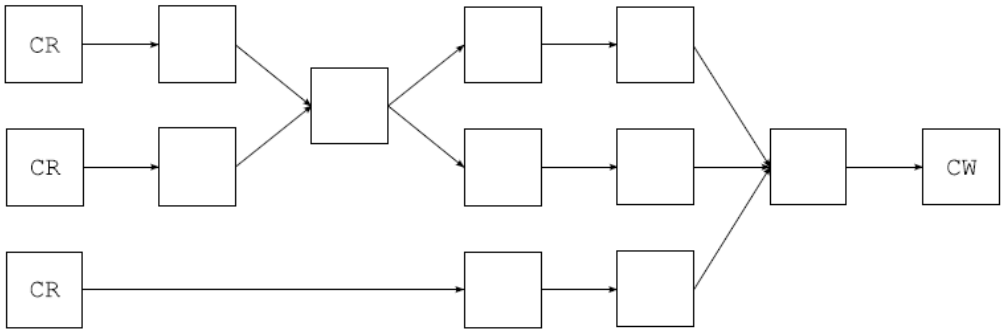


图 2.8：一个复杂的转换链

2.3 工具

除了 ClassVisitor 接口，以及与之相关的三个组件 ClassReader ClassAdapter 和 ClassWriter，ASM 在 org.objectweb.asm.util 包中提供了一些工具用来帮助开发类生成器或者适配器，这些工具在运行时并不需要。ASM 提供了一些实用类用来在运行时操作内部名称，类型描述符以及方法描述符。所有的这些工具将在下面介绍。

2.3.1 类型

就像在前面章节中见到的，ASM API 将编译后的 class 中的 java 类型以内部名称或者类型描述符的方式展现出来。当然，也可以将这些类型还原成源代码中定义的样子，这样就更便于阅读。但是，这需要在 ClassReader 和 ClassWriter 之间进行系统转换，但是这会降低性能。这就是为什么 ASM 没有透明地转换内部名称和类型描述符为源代码中对等的形式。尽管如此，ASM 还是提供了 Type 类用来在需要的时候手工地进行转换。

一个 Type 对象代表了一个 java 类型，它可以通过类型描述符或者 Class 对象来构造。这个 Type 类也包含了一些静态变量，用来表示基本类型，例如 Type.INT_TYPE 是 int 类型的 Type 对象。

getInternalName 方法返回一个 Type 的内部名称，例如，Type.getType(String.class).getInternalName()返回了 String 类的内部名称”java/lang/String”.这个方法只能用于类或者借口类型。

getDescriptor 方法返回一个 Type 的描述符，例如，你可以使用 Type.getType(String.class).getDescriptor() 来代替 ”Ljava/lang/String;” 。或者，使用 Type.INT_TYPE.getDescriptor()来代替 I。

Type 类也提供了一些静态方法用来获取一个方法的参数的 Type 对象和返回值的 Type 对象，主要是通过它的类型描述符或者 java.lang.reflect.Method 对象来获得。例如，Type.getArgumentTypes(“(I)V”) 返回一个包含 Type.INT_TYPE 的数组，同样地 Type.getReturnType(“(I)V”)返回一个 Type.VOID_TYPE 对象。

2.3.2 TraceClassVisitor

为了检查类的生成或者转换是否如你期望，单靠 ClassWriter 返回的字节数组是没有多大帮助的，因为它不可读。相比较而言，一个文本表示更易于阅读和使用，而这就是 TraceClassVisitor 提供的。这个类，就如它的名字暗示的一样，实现了 ClassVisitor 借口，并且构造解析过的类的文本表示。因此，你可以使用 TraceClassVisitor 来替代 ClassWriter，这样你可以跟踪真正生成的是什么。更好的办法是同时使用这两者，TraceClassVisitor 可以跟踪代码生成，除此之外，它也可以将所有的调用委托给另外一个 visitor，如 ClassWriter:

```
ClassWriter cw = new ClassWriter(0);
TraceClassVisitor cv = new TraceClassVisitor(cw, printWriter);
cv.visit(...);
...
cv.visitEnd();
byte b[] = cw.toByteArray();
```

上面的代码创建了一个 TraceClassVisitor，然后委托所有对它的调用给 cw，并且将对这些方法的调用以文本方式交给 printWriter 来打印。例如，在 2.2.3 章节中的例子使用 TraceClassVisitor 会输出如下内容：

```
// class version 49.0 (49)
// access flags 1537
public abstract interface pkg/Comparable implements pkg/Mesurable {
  // access flags 25
  public final static I LESS = -1
  // access flags 25
  public final static I EQUAL = 0
  // access flags 25
  public final static I GREATER = 1
  // access flags 1025
  public abstract compareTo(Ljava/lang/Object;)I
}
```

注意，为了弄清楚在转换链中到底发生了什么，你可以在生成类或者转换链过程的任何点使用 TraceClassVisitor，而不仅仅是在 ClassWriter 之前使用。注意，通过这个类生成的类的文本表示可以通过 String.equals()很容易地进行比较。

2.3.3 CheckClassAdapter

ClassWriter 并不会检查它的方法调用是否按照合适的顺序以及参数是否有效。这样就可能生成无效的代码，而被 java 虚拟机的验证工具所拒绝。为了尽可能地检测出这些错误，可以使用 CheckClassAdapter。和 TraceClassVisitor 一样，这个类也实现了 ClassVisitor 接口，它也会将对它的方法调用委托给其他的 ClassVisitor，例如一个 TraceClassVisitor 或者 ClassWriter。尽管如此，除了打印类的文本表示，这个类会在将方法调用委托给下一个 ClassVisitor 之前，检查对它的方法调用顺序是否合理，以及参数是否有效。如果发生错误，将会抛出 IllegalStateException 或者 IllegalArgumentException。

为了检查一个类，并且打印它的文本表示，最终创建一个字节数组，你可以参考下面的代码：

```
ClassWriter cw = new ClassWriter(0);
TraceClassVisitor tcv = new TraceClassVisitor(cw, printWriter);
CheckClassAdapter cv = new CheckClassAdapter(tcv);
cv.visit(...);
...
cv.visitEnd();
byte b[] = cw.toByteArray();
```

注意，如果这些 ClassVisitor 的顺序不同，那么它们会以不同的顺序执行。例如，下面的代码会导致在跟踪代码以后再检查类。

```
ClassWriter cw = new ClassWriter(0);
CheckClassAdapter cca = new CheckClassAdapter(cw);
TraceClassVisitor cv = new TraceClassVisitor(cca, printWriter);
```

就像 TraceClassVisitor 一样，为了检查类是否有效，你也可以在生成类或者转换类的链的任

何节点使用 CheckClassAdapter，而不仅仅是在 ClassWriter 之前。

2.3.4 ASMifierClassVisitor

这个类也实现了 ClassVisitor 接口，它的每个方法会打印出调用它的 java 代码。例如，调用 visitEnd 会打印出 cv.visitEnd();，结果是，当这个 visitor 解析一个类时，它会打印出使用 ASM 来生成这个类的源代码。当你使用这个类去解析一个存在的类时，你会发现它很有用。例如，如果你不知道如何使用 ASM 来生成一些编译后的类，那么你可以先写出这些类的源代码，然后使用 javac 来编译，再然后使用 ASMifierClassVisitor 来解析，这样就能够得到使用 ASM 来生成这些类的源代码了。

ASMifierClassVisitor 可以通过命令行直接使用，如下面的例子：

```
java -classpath asm.jar:asm-util.jar \
      org.objectweb.asm.util.ASMifierClassVisitor \
      java.lang.Runnable
```

生成的代码经过缩进以后，就是下面的代码：

```
package asm.java.lang;
import org.objectweb.asm.*;
public class RunnableDump implements Opcodes {
      public static byte[] dump() throws Exception {
            ClassWriter cw = new ClassWriter(0);
            FieldVisitor fv;
            MethodVisitor mv;
            AnnotationVisitor av0;
            cw.visit(V1_5, ACC_PUBLIC + ACC_ABSTRACT + ACC_INTERFACE,
                  "java/lang/Runnable", null, "java/lang/Object", null);
            {
                  mv = cw.visitMethod(ACC_PUBLIC + ACC_ABSTRACT, "run", "()V",
                        null, null);
                  mv.visitEnd();
            }
            cw.visitEnd();
            return cw.toByteArray();
      }
}
```

3 方法

这一章节将解释如何使用 ASM 核心 API 生成和转换编译后的方法。首先介绍编译后的方法

的表现形式，然后介绍相关的 ASM 接口，组件以及工具，结合很多说明性的示例讲解如何生成和转换方法。

3.1 结构

在编译后的类中，方法的代码是以字节码指令序列保存的。为了生成和转换这些类，我们必须了解这些指令以及它们如何工作。这个章节将粗略的讲解一下这些指令，但是这已足够我们去编写一些简单的类（我认为这里应该是方法）生成工具和转换工具。如果希望查看这些指令的完整定义，请参看 java 虚拟机规范。

3.1.1 执行模型

在介绍字节码指令之前，需要先介绍一下 java 虚拟机的执行模型。如你所知，java 代码是在线程中被执行的。每个线程都有它自己的执行栈，这个栈由很多帧组成。每个帧代表了一个方法调用：每当一个方法被调用时，就会创建一个新的帧，然后将这个帧放到当前线程的执行栈的栈顶。当这个方法正常返回时，或者发生了异常，这个帧就会从执行栈顶弹出，然后虚拟机会接着执行下一个位于执行栈栈顶的帧。

每个帧都包含两部分：一个局部变量区和一个操作数栈区。局部变量区包含了方法中定义的局部变量，可以通过索引值来随即访问。操作数栈区，如它的名字暗示，它是一个包含操作数的栈，这些数据被字节码指令所使用。这意味着这个栈中的数据只能以后进先出的顺序访问。注意，不要把操作数栈和线程的执行栈混淆：在执行栈中的每个帧都包含它自己的操作数栈。

局部变量区和操作数栈的大小是由方法中的代码决定的。它们的大小是在编译时计算的，然后随着字节码指令一起存储在编译后的类中。结果是，对同一个方法的调用的所有帧的局部变量和操作数栈大小都相同，不同的方法调用的帧的局部变量和操作数栈大小不同。

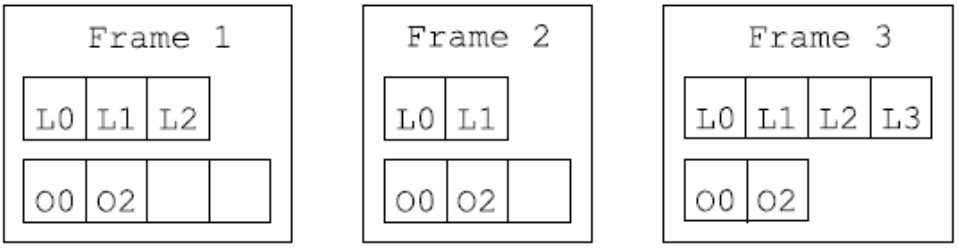


图 3.1 一个包含 3 个帧的执行栈

图 3.1 展示了一个简单执行栈，包含 3 个帧。第一个帧包含 3 个局部变量，操作数栈大小为 4，包含 2 个值。第二个帧包含 2 个局部变量，操作数栈中包含 2 个值。第三个帧位于执行栈的栈顶，包含 4 个局部变量和 2 个操作数。

当帧被创建的时候，操作数栈初始化为空，局部变量包含当前对象的引用 this（非静态方法）以及方法的参数。例如 a.equals(b)会创建一个帧，它包含一个空的操作数栈，第一个和第二个局部变量为 a 和 b（其它的局部变量未被初始化）。

局部变量区和操作数栈区的每一个插槽（图 3.1 中的方框）都能存储 java 中的任何值，除了 long 型和 double 型的值。因为 long 和 double 值需要 2 个插槽，这增加了局部变量管理的复杂性：例如 i^{th} 方法的参数就没必要保存在局部变量 i 中（有疑问）。Math.max(1L,2L)会创建一个帧，第一个局部变量值为 1L，占用前两个插槽，第二个局部变量值为 2L，占据第三个和第四个插槽。

3.1.2 字节码指令

一个字节码指令由一个指示指令的操作码和固定数量的参数：

- opcode 是一个无符号字节数值 – 字节码的名称是由助记符表示。例如，操作码 0 的助记符为 NOP，其对应的指令不做任何事情。
- 参数都是静态的值，定义了精确的指令行为。它们放置在操作码之后。例如，GOTO 标签指令，它的操作码是 167，需要一个参数 label，这个 label 指定了下一条将被执行的指令。在这里不要将指令参数和指令操作数混淆：这里的参数是静态已知的并且被保存在编译后的代码中，而操作数的值来自操作数栈，它们在运行时才可知。

字节码指令可以划分为两类：将局部变量的值传递到操作数栈的一小部分指令集，另外一部分指令在操作数栈上进行操作：它们取得栈顶上的一些数据，进行计算，获得结果，然后将结果放回栈顶。

ILOAD,LLOAD,FLOAD,DLOAD 和 ALOAD 指令读取一个局部变量的值，然后将它放置到操作数栈，这些指令的参数就是需要读取的局部变量的索引值 i。ILOAD 被用来加载一个 boolean，char，short 以及 int 局部变量。LLOAD，FLOAD 和 DLOAD 用来加载 long,float 或者 double 值，注意 LLOAD 和 DLOAD 实际加载了 i 和 i+1 插槽上的值。最后，ALOAD 是用来加载那些非基本类型的值，如对象和数组引用。相对地，ISTORE,LSTORE,FSTORE,DSTORE 和 ASTORE 指令是用来从操作数栈取得相应的数据然后保存会局部变量，它们的参数也是索引值 i。

如你前面所见，XLOAD 和 XSTORE 指令是区分类型的（事实上，包括你即将在下面看到的，几乎所有的指令都是区分类型的）。这样做主要是为了确保不进行非法的类型转换。当然，以不同的类型从局部变量中加载一个值也是合法的。例如，ISTORE 1 ALOAD 1 这样的序列是合法的。当然，允许保存一个任意的内存地址到一个索引为 1 局部变量中，然后转换这个地址为一个对象引用。能够以不同于当前存储在局部变量中的类型来保存到另一个变量中，这样做是很完美的。这意味着一个局部变量的类型，或者存储在这个局部变量中值的类型，可以在方法的执行过程中改变。

如上面所说，其它的指令都是仅针对操作数栈进行操作的。它们可以分为以下几类（参看附录 A.1）：

Stack 这些指令是用来操作栈中的数据：POP 弹出栈顶的数据，DUP 复制栈顶的数据然后放回栈顶，SWAP 交换位于栈顶上的两个数据。

Constants 这些指令用来将一个常量值放置到操作数栈：ACONST_NULL 放置 null 值到栈顶，ICONST_0 放置 int 类型的 0 到栈顶，FCONST_0 放置 float 类型的 0 值到栈顶，DCONST_0 放置 double 类型的 0 值，BIPUSH b 放置字节值 b 到栈顶，SIPUSH s 放置 short 类型的值 s 到栈顶。LDC cst 放置 int，float，long，double，String 或者 class 类型的常量 cst 到栈顶。

算术和逻辑操作 这些指令从操作数栈弹出数值进行计算，然后将结果放回栈顶。它们没有任何参数。XADD,xSUB，Xmul，xDIV，和 xREM 与算术操作符+,-,*,/和%对应，这里的 x 为 I,L,F 或者 D。同样地，这里还包含其它的指令，如<<,>>,|,&,^等针对 int 和 long 值的指令。

Casts 这些指令从栈弹出一个数据，然后将其转换为另外一个类型，然后返回栈顶。它们与 java 中的类型转换对应。I2F,F2D,L2D 等，将数值从一种类型转换为另外一种类型。CHECKCAST t 转换一个引用值类型为 t。

Objects 这些指令是用来创建对象，以及锁定对象，测试它们的类型等。例如，NEW type 指令创建一个新对象，然后放回栈（type 表示内部名称）。

Fields 这些指令用来读取或者写入字段的值。GETFIELD owner name desc 弹出一个对象的引用，然后将该对象字段名称为 name 的值放置到栈中。PUTFIELD owner name desc 从栈中弹出一个值和一个对象引用，然后将这个值保存到这个对象的 name 字段中。在这两种情形中，对象必须是 owner 类型，它的字段必须是 desc 描述的类型。GETSTATIC 和 PUTSTATIC 指令与上面指令类型，只是它们用作静态字段。

Methods 这些指令用来调用方法或者构造方法。它们从栈顶弹出方法的参数，以及目标对象，然后将方法执行结果放回栈顶。INVOKEVIRTUAL owner name desc 调用类型为 owner 方法名为 name，并且方法描述符为 desc 的方法。INVOKESTATIC 用来调用静态方法。INVOKESPECIAL 用来调用私有方法和构造方法。INVOKEINTERFACE 用来调用定义在接口中的方法。

Arrays 这些指令用来读取和写入数组中的值。xALOAD 指令从栈上弹出一个索引值和一个数组，然后将索引值所对应的数组元素放回栈。xSTORE 指令从站上弹出一个值，一个索引以及一个数组，然后将值保存到索引对应的数组元素中。这里的 x 可以是 I,L,F,D 或者 A，也可以是 B,C 或者 S。

Jumps 这些指令在某些条件为真或者为假的时候，跳转到任意一条指令。它们是用来编译 if,for,do,while,break 和 continue 等指令的。例如，IFEQ label 从栈上弹出一个 int 值，如果这个值为 0，就跳转到 label 表示的指令（否则执行流程将按正常流程执行下一条指令）。这里还有其它的一些跳转指令，如 IFNE,或者 IFGE 等。最后，TABLESWITCH 和 LOOKUPSWITCH 指令与 java 中的 switch 对应。

Return xRETURN 和 RETURN 指令是用来终止一个方法的执行，并将结果返回给调用者。RETURN 针对无返回值的方法，而 xRETURN 针对其它有返回值的方法。

3.1.3 示例

接下来我们看看一些基本的例子，以获取关于字节码指令如何工作的具体印象。考虑下面这个 Bean 类：

```
package pkg;
public class Bean {
	private int f;
	public int getF() {
		return this.f;
	}
	public void setF(int f) {
		this.f = f;
	}
}
```

其中 getter 方法的字节码为：

```
ALOAD 0
GETFIELD pkg/Bean f I
IRETURN
```

第一条指令读取索引位置为 0 的局部变量 this，这个局部变量是在这个方法调用时创建的帧的过程中被初始化的，然后将这个局部变量放置到操作数栈栈顶。第二条指令从栈顶弹出这个值，this，然后将字段 f 的值放置到栈顶，this.f。最后一条指令从栈顶弹出得到的字段 f 的值，将它返回给调用者。这个方法执行过程中帧的状态如图 3.2：

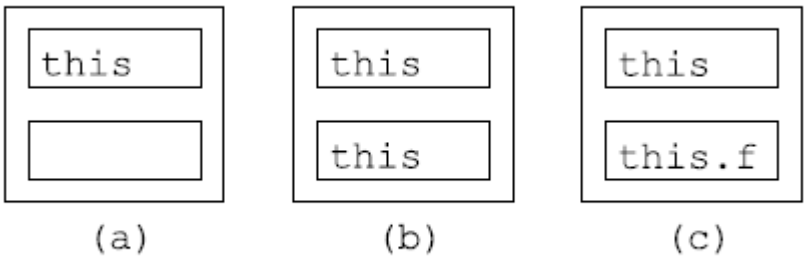


图 3.2 getF 方法帧的状态：a)初始化，b)ALOAD 0 以后，c）GETFIELD 以后

Setter 方法的字节码如下：

```
ALOAD 0
ILOAD 1
PUTFIELD pkg/Bean f I
RETURN
```

第一条指令将 this 放置到操作数栈栈顶。第二条指令将索引为 1 的局部变量值放置到栈顶，这个索引的值为方法的参数，在方法调用创建帧的过程中初始化的。第三个指令从栈顶弹出这两个值，并将 int 值存贮到 this 对象的字段 f 中，this.f。最后一条指令，在源代码中是隐

式定义的，但是在编译后的代码中是必须的，它负责销毁执行帧并将调用返回给调用者。这个方法的执行帧的状态如图 3.3：

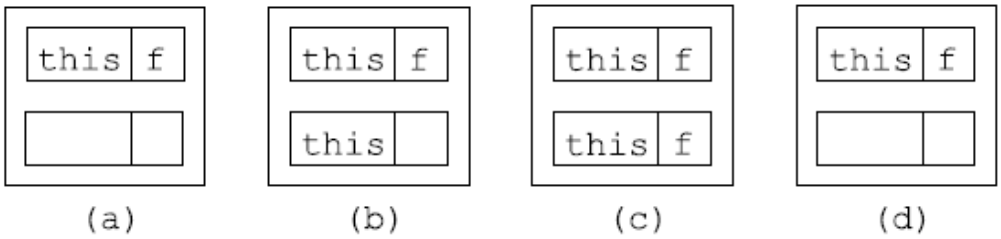


图 3.3 setF 方法执行帧的连续状态：a)初始化，b)ALOAD 0 之后，
c)ILOAD 1 以后，d)PUTFIELD 以后

这个 Bean 类也有一个缺省的共有的构造方法，它是由编译器生成的，因为程序员没有显示的定义构造方法。缺省的构造方法的代码是 Bean(){ super();}。它的字节码如下：

```
ALOAD 0
INVOKESPECIAL java/lang/Object <init> ()V
RETURN
```

第一条指令将 this 放置到操作数栈栈顶。第二个指令从栈顶弹出这个值，然后调用定义在 Object 类中的<init>方法。这对应着 super()方法调用，就是调用父类 Object 的构造方法。在这里可以看出这里的名称在源代码和编译后的代码中是不一样的：在编译后的类中一直为<init>，而在源代码中它们的名称和类名一样。最后一条指令返回到方法调用者。

下面让我们考虑一个更复杂的 setter 方法：

```
public void checkAndSetF(int f) {
    if (f >= 0) {
        this.f = f;
    } else {
        throw new IllegalArgumentException();
    }
}
```

这个 setter 方法的字节码如下：

```
    ILOAD 1
    IFLT label
    ALOAD 0
    ILOAD 1
    PUTFIELD pkg/Bean f I
    GOTO end
label:
    NEW java/lang/IllegalArgumentException
    DUP
```

```
    INVOKESPECIAL java/lang/IllegalArgumentException <init> ()V
    ATHROW
end:
    RETURN
```

第一条指令将索引为 1 的局部变量放置到操作数栈栈顶，这个局部变量就是方法参数 f。IFLT 指令从栈顶弹出这个值，然后将它和 0 比较，如果结果小于 0（LT），执行就跳转到 label 表示的指令处接着执行，否则不做任何事，接着往下面执行。接下来的三条指令与之前 setF 中的指令相同。GOTO 指令无条件地跳转到 end 表示的指令处，也即是 RETURN 指令。在 label 和 end 标签之间的指令，创建并且抛出一个异常：NEW 指令创建异常对象，并将它放置到栈顶，DUP 指令复制栈顶元素，并重新放置到栈顶。INVOKESPECIAL 指令从栈顶取得异常的一个拷贝，然后调用这个异常对象的构造方法。最后，ATHROW 指令弹出剩余的异常对象拷贝，然后抛出（这样执行流程就不会传递到下一条指令）。

3.1.4 异常处理

没有字节码指令用于捕获异常：只有与方法关联的异常处理程序（exception handler）列表，它们指定了在方法执行的某部分发生异常时应该执行的代码。一个异常处理程序类似于 try catch 块：它有一个范围，就是 try 包含的块所对应的指令序列，异常处理程序就对应着 catch 块内容。范围是由 start 和 end 标签指定，异常处理程序以 start 开始。示例代码如下：

```
public static void sleep(long d) {
    try {
        Thread.sleep(d);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

编译之后：

```
TRYCATCHBLOCK try catch catch java/lang/InterruptedException
try:
    LLOAD 0
    NVOKESTATIC java/lang/Thread sleep (J)V
    RETURN
catch:
    INVOKEVIRTUAL java/lang/InterruptedException printStackTrace ()V
    RETURN
```

在 try 和 catch 之间的指令就对应着源码中的 try 代码块，catch 后面的指令就对应着源码中的 catch 代码块。TRYCATCHBLOCK 行指定了一个异常处理程序，覆盖的范围就是 try 和 catch 之间的代码，对应的处理程序从 catch 标签开始，对应的异常是 InterruptedException 的子类。这意味着在 try 和 catch 之间的任何地方发生了这样的异常，栈将被清空，异常就被放入这个空栈中，执行流程就传递到 catch 指令。

3.1.5 帧

使用 java6 或者更高版本的 jdk 编译的类，除了包含字节码指令之外，还包含一个栈映射帧（stack map frames）集合，它们用来加速虚拟机内部的类验证流程。一个栈映射帧描述了在一个方法执行时，在一些点的执行帧状态。更精确地说法就是，它给出了在字节码指令执行前，局部变量区和操作数栈区每个值的类型（type）。

例如，我们来考虑之前的 getF 方法，我们可以定义三个栈映射帧来分别描述 ALOAD,GETFIELD 和 IRETURN 这三个指令之前的执行帧状态。这三个栈映射帧对应了三种情形，见图 3.2，可以如下描述，第一个括号中描述的是局部变量的类型，其它的则是操作数栈中数据的类型：

```
State of the execution frame before     Instruction
[pkg/Bean] []                           ALOAD 0
[pkg/Bean] [pkg/Bean]                   GETFIELD
[pkg/Bean] [I]                              IRETURN
[pkg/Bean I] []                         ILOAD 1
[pkg/Bean I] [I]                        IFLT label
[pkg/Bean I] []                         ALOAD 0
[pkg/Bean I] [pkg/Bean]                 ILOAD 1
[pkg/Bean I] [pkg/Bean I]               PUTFIELD
[pkg/Bean I] []                         GOTO end
[pkg/Bean I] []                         label:
[pkg/Bean I] []                         NEW
[pkg/Bean I] [Uninitialized(label)] DUP
[pkg/Bean I] [Uninitialized(label)  Uninitialized(label)] INVOKESPECIAL
[pkg/Bean I] [java/lang/IllegalArgumentException] ATHROW
[pkg/Bean I] []                         end:
[pkg/Bean I] []                         RETURN
```

除了 Uninitialized(label)类型之外，这个和之前的方法没什么区别。这个 Uninitialized(label)类型是一个特殊的类型，只用在栈映射帧中，它表示一个对象的内存已经分配，但是其构造方法还未被调用。参数表示创建这个对象的指令。对于这个值所能调用的唯一的方法就是构造方法。当构造方法调用以后，所有的 Uninitialized(label)就会被替换为真实的类型，在这里就是 IllegalArgumentException。栈映射帧还可以使用另外的三个特殊类型：UNINITIALIZED_THIS 是构造方法中索引值为 0 的局部变量的初始化类型，TOP 对应着未定义的值，NULL 对应着 null。

就如上面所说，从 java6 开始，编译的类中包含栈映射帧的集合。为了节省空间，一个编译的方法并不是每个指令都对应着一个帧：事实上只有那些对应着判断指令的目标或者异常处理程序，或者无条件跳转指令才有帧。因为其它的帧可以很容易地很快地从这些帧继承而来。

在 checkAndSetF 方法的示例中，这意味着只有两个帧被存储：一个是 NEW 指令，因为它是 IFLT 指令的目标，同时它紧跟这无条件跳转指令 GOTO，另外一个是 RETURN 指令，因为它

是 GOTO 指令的目标，同时它紧跟着无调教跳转指令 ATHROW 指令。

为了节省更多的空间，每个帧都会被压缩，然后存储它与之前帧的不同，初始帧不被保存，因为它可以从方法的参数类型推断出来。在 checkAndSetF 的例子中，那两个必须被保存的帧是和初始帧相同的，因此它们使用 F_SAME 助记符来存储。这些帧可以在与它们关联的字节码指令之前被助记符表示。下面给出了 checkAndSetF 方法的最终字节码指令：

```
    ILOAD 1
    IFLT label
    ALOAD 0
    ILOAD 1
    PUTFIELD pkg/Bean f I
    GOTO end
label:
F_SAME
    NEW java/lang/IllegalArgumentException
    DUP
    INVOKESPECIAL java/lang/IllegalArgumentException <init> ()V
    ATHROW
end:
F_SAME
    RETURN
```

3.2 接口和组件

3.2.1 表现

ASM API 中关于方法的生成和转换的部分是基于 MethodVisitor 接口的（见图 3.4），该接口由 ClassVisitor 的 visitMethod 方法返回。除了一些与注解和调试相关的方法之外，这些将在后面章节中介绍，MethodVisitor 接口还针对每个字节码指令分类定义了一个方法，这个分类是基于这些指令的编号以及参数的类型（这些分类与 3.1.2 章节中出现的分类并不对应）。这些方法必须按照下面给定的顺序来调用（还包含一些定义在 MethodVisitor 接口的 JavaDoc 文档中的限制）：

```
visitAnnotationDefault?
( visitAnnotation | visitParameterAnnotation | visitAttribute )*
( visitCode
    ( visitTryCatchBlock | visitLabel | visitFrame | visitXxxInsn |
    visitLocalVariable | visitLineNumber )*
    visitMaxs )?
    visitEnd
```

这就意味着，如果存在注解和属性，那么它们必须首先被调用，然后才是方法的字节码，这只针对非抽象方法。位于 visitCode 和 visitMax 方法之间的这些方法，必须按照顺序来调用，

并且 visitCode 和 visitMax 只能调用一次。

```
public interface MethodVisitor {
    AnnotationVisitor visitAnnotationDefault();
    AnnotationVisitor visitAnnotation(String desc, boolean visible);
    AnnotationVisitor visitParameterAnnotation(int parameter,
    String desc, boolean visible);
    void visitAttribute(Attribute attr);
    void visitCode();
    void visitFrame(int type, int nLocal, Object[] local, int nStack,
    Object[] stack);
    void visitInsn(int opcode);
    void visitIntInsn(int opcode, int operand);
    void visitVarInsn(int opcode, int var);
    void visitTypeInsn(int opcode, String desc);
    void visitFieldInsn(int opc, String owner, String name, String desc);
    void visitMethodInsn(int opc, String owner, String name, String desc);
    void visitJumpInsn(int opcode, Label label);
    void visitLabel(Label label);
    void visitLdcInsn(Object cst);
    void visitIincInsn(int var, int increment);
    void visitTableSwitchInsn(int min, int max, Label dflt,
    Label labels[]);
    void visitLookupSwitchInsn(Label dflt, int keys[], Label labels[]);
    void visitMultiANewArrayInsn(String desc, int dims);
    void visitTryCatchBlock(Label start, Label end, Label handler,
    String type);
    void visitLocalVariable(String name, String desc, String signature,
    Label start, Label end, int index);
    void visitLineNumber(int line, Label start);
    void visitMaxs(int maxStack, int maxLocals);
    void visitEnd();
}
```

图 3.4 MethodVisitor 接口

在一系列的事件中，visitCode 和 visitMaxs 可以被用来检测方法字节码的开始和结束。就像类一样，visitEnd 方法必须最后被调用，同时可以用来检测类的结束。

ClassVisitor 和 MethodVisitor 可以一定的顺序合并起来以生成更复杂的类：

```
ClassVisitor cv = ...;
cv.visit(...);
MethodVisitor mv1 = cv.visitMethod(..., "m1", ...);
mv1.visitCode();
mv1.visitInsn(...);
...
mv1.visitMaxs(...);
```

```
mv1.visitEnd();
MethodVisitor mv2 = cv.visitMethod(..., "m2", ...);
mv2.visitCode();
mv2.visitInsn(...);
...
mv2.visitMaxs(...);
mv2.visitEnd();
cv.visitEnd();
```

注意，这里没有必要结束一个方法的访问之后再去访问另外一个，事实上，MethodVisitor 接口是完全独立的，它们可以任何顺序来使用，只要 cv.visitEnd()方法还未被调用：

```
ClassVisitor cv = ...;
cv.visit(...);
MethodVisitor mv1 = cv.visitMethod(..., "m1", ...);
mv1.visitCode();
mv1.visitInsn(...);
...
MethodVisitor mv2 = cv.visitMethod(..., "m2", ...);
mv2.visitCode();
mv2.visitInsn(...);
...
mv1.visitMaxs(...);
mv1.visitEnd();
...
mv2.visitMaxs(...);
mv2.visitEnd();
cv.visitEnd();
```

ASM 提供了三个基于 MethodVisitor 接口的组件来生成和转换方法：

- ClassReader 解析编译后方法的内容，然后调用 MethodVisitor 接口中对应的方法，其中 MethodVisitor 实例由 ClassVisitor 返回，而 ClassVisitor 实例被当做参数传递给 ClassReader 的 accept 方法。
- ClassWriter 的 visitMethod 方法返回 MethodVisitor 接口的一个实现，由它来直接以二进制的方式构建编译后的方法。
- MethodAdapter 是 MethodVisitor 的一个实现，它将自己接受的方法调用委托给其他的 MethodVisitor 实例。

ClassWriter 的选项

如我们在 3.1.5 章节中所见，计算一个方法的栈映射帧并不容易：你必须计算所有的帧，找到与跳转目标以及那些无条件跳转想对应的帧，最后压缩这些帧。同样地，计算一个方法的局部变量和操作数栈的大小也不容易，虽然相对而言容易点。

令人高兴地是，ASM 可以为你计算这些东西。当你创建一个 ClassWriter 时，你可以指定哪

些是需要自动计算的：

- **new ClassWriter(0)** 将不会自动进行计算。你必须自己计算帧、局部变量和操作数栈的大小。
- **new ClassWriter(ClassWriter.COMPUTE_MAXS)** 局部变量和操作数栈的大小就会自动计算。但是，你仍然需要自己调用 visitMaxs 方法，尽管你可以使用任何参数：实际上这些参数会被忽略，然后重新计算。使用这个选项，你仍然需要计算帧的大小。
- **new ClassWriter（ClassWriter.COMPUTE_FRAMES）** 所有的大小都将自动为你计算。你也不许要调用 visitFrame 方法，但是你仍然需要调用 visitMaxs 方法（参数将被忽略然后重新计算）。

使用这些选项是方便很多，但是会带来一些损失：COMPUTE_MAXS 选项将使得 ClassWriter 慢 10%，使用 COMPUTE_FRAMES 选项将使得 ClassWriter 慢两倍。你必须将它与自己计算花费的时间相比较：与 ASM 提供的算法相比较，在某些特定的情况下，这里有一些更容易和快速的算法来进行计算，但是必须处理所有的情形。

如果你选择自己计算这些帧的大小，你可以让 ClassWriter 来帮你进行帧压缩。这样的话，你只能使用 visitFrame(F_NEW,nLocals,locals,nStack,stack) 方法去访问那些未被压缩的帧,nLocals 和 nStack 代表局部变量和操作数栈的大小，locals 和 stack 是一些包含对应类型的数组（参看 JavaDoc）。

注意，为了自动计算帧的大小，有时必须计算两个类共同的父类。缺省情况下，ClassWriter 将会在 getCommonSuperClass 方法中计算这些，通过在加载这两个类进入虚拟机时，使用反射 API 来计算。但是，如果你将要生成的几个类相互之间引用，这将会带来问题，因为引用的类可能还不存在。在这种情况下，你可以重写 getCommonSuperClass 方法来解决这个问题。

3.2.2 生成方法

在 3.1.3 节中定义的 getF 方法字节码可以通过下面的方法调用来生成，其中 mv 为 MethodVisitor：

```
mv.visitCode();
mv.visitVarInsn(ALOAD, 0);
mv.visitFieldInsn(GETFIELD, "pkg/Bean", "f", "I");
mv.visitInsn(IRETURN);
mv.visitMaxs(1, 1);
mv.visitEnd();
```

从第一个方法开始进行字节码生成，紧跟着的三个方法调用用来生成方法中的代码（字节码和 ASM API 之间的映射还是很简单的）。visitMaxs 方法必须在其他方法被调用之后再调用，它是用来这个方法的执行帧中局部变量和操作数栈的大小的。如在 3.1.3 节中见到的，一个方框表示一部分。最后的方法调用是用来结束方法的生成。

相应地，setF 方法和构造方法也可以相似的方式来生成。一个比较有意思的例子是 checkAndSetF 方法：

```
mv.visitCode();
mv.visitVarInsn(ILOAD, 1);
Label label = new Label();
mv.visitJumpInsn(IFLT, label);
mv.visitVarInsn(ALOAD, 0);
mv.visitVarInsn(ILOAD, 1);
mv.visitFieldInsn(PUTFIELD, "pkg/Bean", "f", "I");
Label end = new Label();
mv.visitJumpInsn(GOTO, end);
mv.visitLabel(label);
mv.visitFrame(F_SAME, 0, null, 0, null);
mv.visitTypeInsn(NEW, "java/lang/IllegalArgumentException");
mv.visitInsn(DUP);
mv.visitMethodInsn(INVOKESPECIAL,
    "java/lang/IllegalArgumentException", "<init>", "()V");
mv.visitInsn(ATHROW);
mv.visitLabel(end);
mv.visitFrame(F_SAME, 0, null, 0, null);
mv.visitInsn(RETURN);
mv.visitMaxs(2, 2);
mv.visitEnd();
```

在 visitCode 和 visitEnd 调用之间的方法调用和 3.1.5 节最后展示的字节码是精确映射的。每个调用对应一个指令，标签或者帧（唯一的例外就是 label 和 end label 的声明和构造）。

注意 一个 label 对象指定的指令就是那些紧跟在这个 label 的 visitLabel 方法调用之后。例如，end 指定了 RETURN 指令，不是其后的 frame，因为它不是一个指令。可以使用几个标签来指定相同的指令，但是一个标签必须指定一个指令。换句话说，可以针对不同的 label 连续调用 visitLabel，但是在一个指令中使用的标签必须只能被 visitLabel 访问一次。最后一个限制就是标签不能共享，每个方法都有自己的标签。

3.2.3 转换方法

现在，你已经猜到了，方法可以像类一样就行转换，例如，通过使用一个方法适配器来转发那些带有修改的方法调用：改变参数可以被用来变更指令，不转发某个方法调用可以删除一个指令，插入新的调用可以添加新的指令。MethodAdapter 类提供了这样的基本实现，它仅仅转发它收到的所有方法调用。

为了弄清楚方法适配器如何使用，让我们考虑一个简单的例子，删除方法中的 NOP 指令（删除该指令不会带来任何问题，因为这个指令不做任何事情）：

```
public class RemoveNopAdapter extends MethodAdapter {
    public RemoveNopAdapter(MethodVisitor mv) {
        super(mv);
    }
```

```
    @Override
    public void visitInsn(int opcode) {
        if (opcode != NOP) {
            mv.visitInsn(opcode);
        }
    }
}
```

这个适配器可以用在一个类适配器中，像下面一样：

```
public class RemoveNopClassAdapter extends ClassAdapter {
    public RemoveNopClassAdapter(ClassVisitor cv) {
        super(cv);
    }
    @Override
    public MethodVisitor visitMethod(int access, String name,
        String desc, String signature, String[] exceptions) {
        MethodVisitor mv;
        mv = cv.visitMethod(access, name, desc, signature, exceptions);
        if (mv != null) {
            mv = new RemoveNopAdapter(mv);
        }
        return mv;
    }
}
```

这个类适配器主要用来构建方法适配器，该适配器封装链上的下一个 Class Visitor 返回的方法适配器，然后返回该适配器。这个方法适配器链构造的效果类似于之前的类适配器链（参看图 3.5）。

尽管上面的要求不是强制性的，但是仍然可能构造一个与类适配器链不同的方法适配器链。每个方法甚至都可以有一个不同的方法适配器链。例如，类适配器可以选择在方法中而不是构造方法中移除 NOP 指令。可以像下面这样来实现：

```
...
mv = cv.visitMethod(access, name, desc, signature, exceptions);
if (mv != null && !name.equals("<init>")) {
mv = new RemoveNopAdapter(mv);
}
...
```

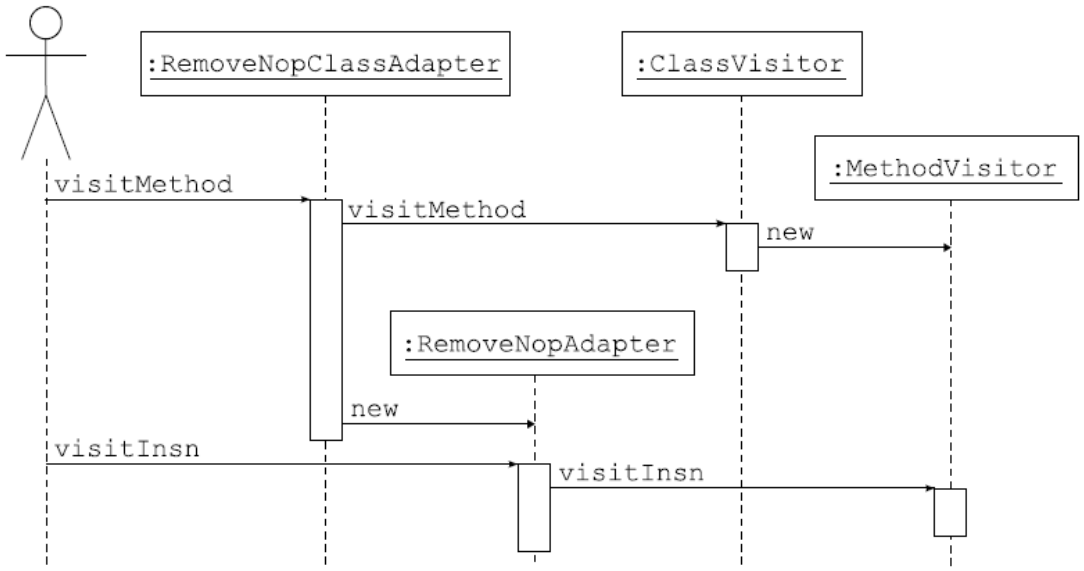


图 3.5 RemoveNopAdapter 序列图

在这个例子中，针对构造方法的适配器链要短一些。相反地，针对构造方法的适配器链也可以更长，可以将几个方法适配器在 visitMethod 方法内部链接在一起。方法适配器链也可以拥有与类适配器链不同的拓扑图结构。例如，类适配器链应该是线性的，而方法适配器可以有分支：

```
public MethodVisitor visitMethod(int access, String name,
    String desc, String signature, String[] exceptions) {
    MethodVisitor mv1, mv2;
    mv1 = cv.visitMethod(access, name, desc, signature, exceptions);
    mv2 = cv.visitMethod(access, "_" + name, desc, signature, exceptions);
    return new MultiMethodAdapter(mv1, mv2);
}
```

现在，我们已经了解了如何在一个类适配器中使用方法适配器，以及如何组装它们，下面让我们来看看如何实现比 RemoveNopAdapter 更有趣的适配器。

3.2.4 无状态转换

假定，我们想要测量一个程序中花费在每个类上的时间，那么我们需要在每个类上添加一个静态的计时字段，并且我们需要将每个方法的执行之间加到这个字段上。也就是说我们想转换类似于 C 这样的一个类：

```
public class C {
    public void m() throws Exception {
        Thread.sleep(100);
    }
}
```

转换之后：

```
public class C {
```

```
    public static long timer;
    public void m() throws Exception {
        timer -= System.currentTimeMillis();
        Thread.sleep(100);
        timer += System.currentTimeMillis();
    }
}
```

为了弄清楚 ASM 是如何实现的，我们将编译这两个类，然后对比它们的 TraceClassVisitor 或者 ASMifierClassVisitor 的输出。通过 TraceClassVisitor 我们可以发现以下不同（粗体表示）：

GETSTATIC C.timer : J
INVOKESTATIC java/lang/System.currentTimeMillis()J
LSUB
PUTSTATIC C.timer : J
LDC 100
INVOKESTATIC java/lang/Thread.sleep(J)V
GETSTATIC C.timer : J
INVOKESTATIC java/lang/System.currentTimeMillis()J
LADD
PUTSTATIC C.timer : J
RETURN
MAXSTACK = **4**
MAXLOCALS = 1

通过上面我们看到了，我们必须在这个方法的最开始添加四条指令，以及在这个方法返回之前添加另外四条指令。最后 我们需要更新操作数栈的大小。因此，我们可以在方法适配器中重写这个方法以增加最开始的四条指令：

```
public void visitCode() {
    mv.visitCode();
    mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
        "currentTimeMillis", "()J");
    mv.visitInsn(LSUB);
    mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
}
```

在这里 owner 必须设置为即将转换的类的名字。现在我们需要添加另外四条位于 RETURN 之前的指令，同时这四条指令也必须位于 xRETURN 或者 ATHROW 之前，因为这些指令都会结束方法的执行。这些指令没有任何参数，并且都是通过 visitInsn 方法来访问。因此，我们可以重写这个方法来添加这四条指令：

```
public void visitInsn(int opcode) {
    if ((opcode >= IRETURN && opcode <= RETURN) || opcode == ATHROW) {
        mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
        mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
```

```
            "currentTimeMillis", "()J");
            mv.visitInsn(LADD);
            mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
        }
        mv.visitInsn(opcode);
    }
```

最后，我们仍然需要更新操作数栈的大小。这些指令中，我们往操作数栈中放置了两个 long 行数据，因此需要四个方框。在这个方法开始的时候，这个操作数栈是被初始化为空的，因此我们知道最开始的四条指令需要的栈大小为 4.我们也知道，我们插入的代码对这个栈大小没有改变（因为从栈中弹出的数据与入栈的数据一样多）。结果是，如果最初的代码需要的栈大小为 s，那么转换方法之后栈的大小的最大值为 max(4,s)。不幸地是，我们还需要在 RETURN 指令之前添加四条指令，在这里我们不知道这个操作数栈的大小。我们只知道它的大小应该小于等于 s。最后，我们可以认为在 RETURN 指令之前添加的代码需要的操作数栈的大小最大为 s+4。这种最坏的情形在实际中很少发生：就一般的编译器而言，RETURN 指令之前，操作数栈中一般只包含方法的返回值，那么它的大小就可能是 0,1 或者最大为 2. 但是，如果我们希望处理所有可能的情况，那么我们应该使用最坏的这种情况。所以，我们需要重写 visitMaxs 方法：

```
public void visitMaxs(int maxStack, int maxLocals) {
    mv.visitMaxs(maxStack + 4, maxLocals);
}
```

当然了，我们也不必为操作数栈的大小而烦恼，我们可以依赖于 COMPUTE_MAXS 选项，它会为我们计算最优的值而不是最坏的值。同时，就这么一个简单的转换，也没必须花费这么经理来手动更新 maxStack。

现在，一个有趣的问题是：如何处理栈映射帧？原始的代码不包含任何帧，转换后的代码页不包含，难道这是因为我们使用的例子很特殊吗？这里是否有一些情形帧必须被更新吗？答案是否定的，因为，第一，插入的代码没有改变操作数栈，第二，插入的代码没有包含跳转指令，第三，这些跳转指令，或者更正式的，程序的控制流没有修改。这意味着原始的帧没有改变，既然新插入的代码没有增加帧，那么原始帧压缩后也没有改变。

现在，我们可以将这些元素和 ClassAdapter 以及 MethodAdapter 结合起来：

```
public class AddTimerAdapter extends ClassAdapter {
    private String owner;
    private boolean isInterface;
    public AddTimerAdapter(ClassVisitor cv) {
        super(cv);
    }
    @Override public void visit(int version, int access, String name,
        String signature, String superName, String[] interfaces) {
        cv.visit(version, access, name, signature, superName, interfaces);
        owner = name;
        isInterface = (access & ACC_INTERFACE) != 0;
```

```
    }
    @Override public MethodVisitor visitMethod(int access, String name,
        String desc, String signature, String[] exceptions) {
        MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
        exceptions);
        if (!isInterface && mv != null && !name.equals("<init>")) {
            mv = new AddTimerMethodAdapter(mv);
        }
    return mv;
    }
    @Override public void visitEnd() {
        if (!isInterface) {
            FieldVisitor fv = cv.visitField(ACC_PUBLIC + ACC_STATIC, "timer",
            "J", null, null);
            if (fv != null) {
                fv.visitEnd();
            }
        }
    cv.visitEnd();
}

class AddTimerMethodAdapter extends MethodAdapter {
    public AddTimerMethodAdapter(MethodVisitor mv) {
        super(mv);
    }
    @Override public void visitCode() {
        mv.visitCode();
        mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
        mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
        "currentTimeMillis", "()J");
        mv.visitInsn(LSUB);
        mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
    }
    @Override public void visitInsn(int opcode) {
        if ((opcode >= IRETURN && opcode <= RETURN) || opcode == ATHROW) {
            mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
            "currentTimeMillis", "()J");
            mv.visitInsn(LADD);
            mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
        }
        mv.visitInsn(opcode);
    }
    @Override public void visitMaxs(int maxStack, int maxLocals) {
```

```
            mv.visitMaxs(maxStack + 4, maxLocals);
        }
        }
}
```

这里的类适配器是用来实例化方法适配器（除了构造方法），同时也用来添加计时字段和保存将被转换的类名到一个字段中，这样在方法适配器中可以访问到这个类名。

3.2.5 有状态转换

在前面章节看到的转换都是局部的，它们都没有依赖于之前已经访问过的代码：在方法最开始添加的代码始终是一样的并且总是会被添加，同样，在 RETURN 之前添加的代码也是如此。这样的转换称之为无状态转换。这些转换较容易实现，但是也只有这种最简单的转换才满足这个属性。

更复杂的转换，需要记住之前访问过代码的一些状态。考虑这样的一个转换，移除所有 ICONST_0 IADD 指令序列，这些指令的效果类似于加零。当一个 IADD 指令被访问时，只有当最后一个指令是 ICONST_0 时，这个指令序列才可以被移除。这需要在方法适配器内部保存一些状态。类似于这样的转换称之为有状态转换。

让我们详细地了解下这个示例。当一个 ICONST_0 指令被访问时，如果其下一条指令是 IADD 时，才可以将这两条指令删除。问题是在访问 ICONST_0 指令时，下一条指令还未知。解决方案就是推迟最初这个判断知道下一条指令被访问：如果访问到 IADD 指令，那么就移除这两个指令，否则忽略 ICONST_0 指令和当前的指令。

为了实现这样的一个转换：移除或者替换某些指令序列，简便的方式就是引入一个方法适配器 MethodAdapter，这个类的 visitXxxInsn 方法会调用一个共同的方法 visitInsn 方法：

```
public abstract class PatternMethodAdapter extends MethodAdapter {
    protected final static int SEEN_NOTHING = 0;
    protected int state;
    public PatternMethodAdapter(MethodVisitor mv) {
        super(mv);
    }
    @Overrid public void visitInsn(int opcode) {
        visitInsn();
        mv.visitInsn(opcode);
    }
    @Override public void visitIntInsn(int opcode, int operand) {
        visitInsn();
        mv.visitIntInsn(opcode, operand);
    }
    ...
    protected abstract void visitInsn();
}
```

上面的转换可以像下面这样实现：

```
public class RemoveAddZeroAdapter extends PatternMethodAdapter {
    private static int SEEN_ICONST_0 = 1;
    public RemoveAddZeroAdapter(MethodVisitor mv) {
        super(mv);
    }
    @Override public void visitInsn(int opcode) {
        if (state == SEEN_ICONST_0) {
            if (opcode == IADD) {
                state = SEEN_NOTHING;
                return;
            }
        }
        visitInsn();
        if (opcode == ICONST_0) {
            state = SEEN_ICONST_0;
            return;
        }
        mv.visitInsn(opcode);
    }
    @Override protected void visitInsn() {
        if (state == SEEN_ICONST_0) {
            mv.visitInsn(ICONST_0);
        }
        state = SEEN_NOTHING;
    }
}
```

visitInsn 方法首先会判断指令序列是否被检测到。在这个例子中，它初始化 state 然后立即返回，它具有删除指令序列的效果。在另外的情形中，它会调用 visitInsn 方法，这个方法会忽略 ICONST_0 指令。如果当前的指令是 ICONST_0,它会记录下来，然后返回，这样就可以在下一条指令到来时判断了。在所有其它的情形中，当前的指令直接被传递给了下一个 visitor。

标签和帧

如我们在前面章节中见到的一样，标签和帧会先于它们关联的指令被访问。换句话说，它们会和指令同时被访问，尽管它们本身不是指令。这会对检测指令序列这样的转换造成一定的影响，但这种影响事实上是有利的。如果我们移除的指令是跳转指令的目标，那会发生什么了？如果有指令会跳转到 ICONST_0 指令，这意味着这里有一个标签来代表这个指令。在移除这两个指令后，这个标签就会代表 IADD 之后的指令，这正是我们想要的。但是如果一些指令要跳转到 IADD 指令，那我们就不能移除这个指令序列了（我们不能确保在跳转之前，一个 0 值入栈了）。让人有希望的是，在这种情况下，在 ICONST_0 和 IADD 之间必须有一个

标签，并且这很容易检测到。

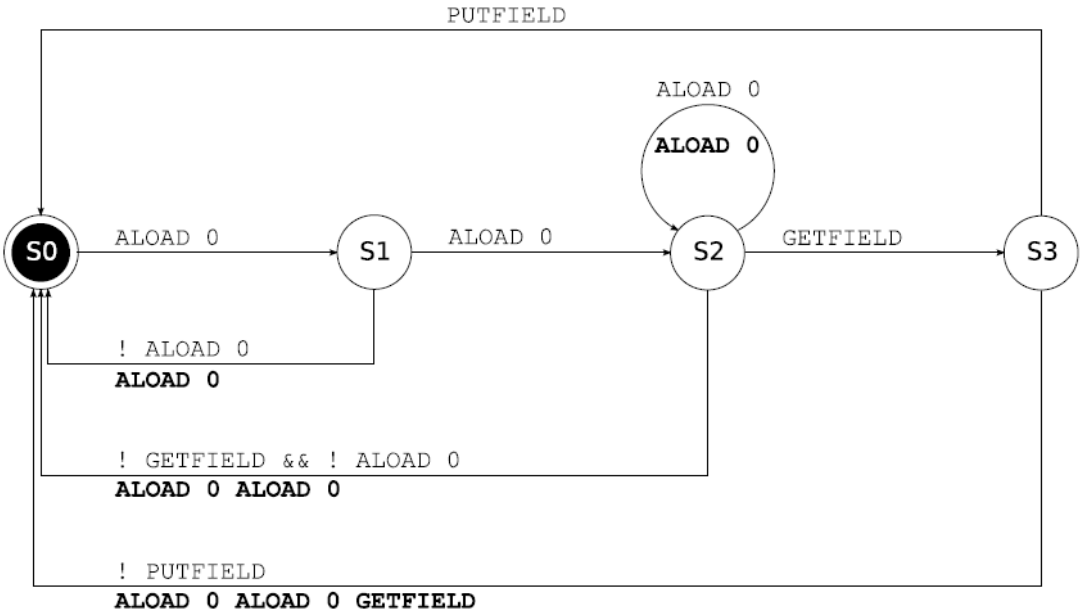
这里的原因也同样适用于栈映射帧：如果一个栈映射帧在这两个指令之间被访问，我们也不能移除这两个指令。这两种情形，我们都可以把标签和帧当做指令使用模式匹配算法来处理。这可以使用 PatternMethodAdapter 类完成（注意 visitMaxs 方法也会调用 visitInsn 方法，这是用来处理这样的情形：方法的结尾可能是这个序列的前缀，它们也应该被检测到）：

```
public abstract class PatternMethodAdapter extends MethodAdapter {
    ...
    @Override public void visitFrame(int type, int nLocal, Object[] local,
    int nStack, Object[] stack) {
        visitInsn();
        mv.visitFrame(type, nLocal, local, nStack, stack);
    }
    @Override public void visitLabel(Label label) {
        visitInsn();
        mv.visitLabel(label);
    }
    @Override public void visitMaxs(int maxStack, int maxLocals) {
        visitInsn();
        mv.visitMaxs(maxStack, maxLocals);
    }
}
```

我们在下一章节中会看到，一个编译后的类可能会包含源文件行号等信息，可以用来进行异常追踪。这些信息通过 visitLineNumber 方法来访问，它们也会在指令被访问的同时被调用。尽管如此，出现在指令序列中间的行号也不会对转换或者删除行号有任何影响。相应的解决办法就是在模式匹配算法中完全忽略它们。

一个更复杂的例子

之前的例子可以很容易地扩展到更复杂的指令序列的情形。考虑这样转换示例，删除所有字段的自我赋值，如 f=f，或者，以字节码的形式就是，ALOAD 0 ALOAD 0　GETFIELD f　PUTFIELD f。在实现这个转换之前，我们最好来设计一个识别这种序列的状态机：



每个转换都会以一个条件作为标签（就是当前的指令）以及一个动作（一个必须被执行的指令序列，以粗体表示）。例如从 S1 到 S0 的转换，只有当前的指令不是 ALOAD 0 时才进行。在这种情形下，ALOAD 0 会被访问然到达初始状态。来看看 S2 到自身的转换：如果找到 3 个或者更多的连续的 ALOAD 0 指令，那么将进行 S2 到自身的转换。在这种情形下，我们会停留在这样的状态：两条 ALOAD 0 指令已经被访问，同时我们将再执行一个 ALOAD 0 指令。一旦这个状态机模型被找到，那么编写对应的方法适配器就很直接了（下面的 8 个 switch case 就对应着上面的 8 个转换）：

```
class RemoveGetFieldPutFieldAdapter extends PatternMethodAdapter {
    private final static int SEEN_ALOAD_0 = 1;
    private final static int SEEN_ALOAD_0ALOAD_0 = 2;
    private final static int SEEN_ALOAD_0ALOAD_0GETFIELD = 3;
    private String fieldOwner;
    private String fieldName;
    private String fieldDesc;

    public RemoveGetFieldPutFieldAdapter(MethodVisitor mv) {
        super(mv);
    }
    @Override
    public void visitVarInsn(int opcode, int var) {
        switch (state) {
        case SEEN_NOTHING: // S0 -> S1
            if (opcode == ALOAD && var == 0) {
            state = SEEN_ALOAD_0;
            return;
            }
            break;
```

```
        case SEEN_ALOAD_0: // S1 -> S2
            if (opcode == ALOAD && var == 0) {
            state = SEEN_ALOAD_0ALOAD_0;
            return;
            }
            break;
        case SEEN_ALOAD_0ALOAD_0: // S2 -> S2
            if (opcode == ALOAD && var == 0) {
            mv.visitVarInsn(ALOAD, 0);
            return;
            }
            break;
        }
        visitInsn();
        mv.visitVarInsn(opcode, var);
}
@Override
public void visitFieldInsn(int opcode, String owner, String name,
        String desc) {
        switch (state) {
        case SEEN_ALOAD_0ALOAD_0: // S2 -> S3
            if (opcode == GETFIELD) {
            state = SEEN_ALOAD_0ALOAD_0GETFIELD;
            fieldOwner = owner;
            fieldName = name;
            fieldDesc = desc;
            return;
            }
            break;
        case SEEN_ALOAD_0ALOAD_0GETFIELD: // S3 -> S0
            if (opcode == PUTFIELD && name.equals(fieldName)) {
            state = SEEN_NOTHING;
            return;
            }
            break;
        }
        visitInsn();
        mv.visitFieldInsn(opcode, owner, name, desc);
}
@Override protected void visitInsn() {
        switch (state) {
        case SEEN_ALOAD_0: // S1 -> S0
            mv.visitVarInsn(ALOAD, 0);
            break;
```

```
            case SEEN_ALOAD_0ALOAD_0: // S2 -> S0
                mv.visitVarInsn(ALOAD, 0);
                mv.visitVarInsn(ALOAD, 0);
                break;
            case SEEN_ALOAD_0ALOAD_0GETFIELD: // S3 -> S0
                mv.visitVarInsn(ALOAD, 0);
                mv.visitVarInsn(ALOAD, 0);
                mv.visitFieldInsn(GETFIELD, fieldOwner, fieldName, fieldDesc);
                break;
            }
            state = SEEN_NOTHING;
        }
    }
```

注意，因为和 3.2.4 节中 AddTimerAdapter 相似的原因，本节中出现的有状态转换不需要转换栈映射帧：原始的帧仍在转换之后然有效。也不需要转换局部变量和操作数栈的大小。最后，注意有状态转换不局限于检测和转换指令序列，还有很多其它类型的转换也是状态转换。下一节出现的有关方法适配器就是这样的例子。

3.3 工具

在 org.objectweb.asm.commons 包中预先定义了一些方法适配器，它们可以辅助你定义你自己的适配器。这个章节将介绍其中的三个适配器，展示如何在 AddTimerAdapter 示例中使用它们（3.2.4）。它也展示了如何使用前面章节中提到的工具来删除方法或者转换。

3.3.1 基本工具

在 2.3 节中出现的工具也可以针对方法使用。

Type

很多字节码指令，如 Xload，xADD 或者 xRETURN，都依赖于它所处理的类型。Type 这个类提供了一个 getOpcode 方法，针对这些指令，它可以获取一个给定类型的 opcode。这个方法以一个 int 类型的 opcode 作为参数，然后返回针对调用该方法对象的类型的 opcode。例如 t.getOpcode(IMUL)返回 FMUL，其中 t 为 Type.FLOAT_TYPE.

TraceClassVisitor

这个类在前面章节已经介绍过了，主要是打印它所访问类的字节码的文本表示，当然也包括方法的文本表示。这个类也可以用来追踪转换链中方法生成和转换的内容。例如：

```
java -classpath asm.jar:asm-util.jar \
    org.objectweb.asm.util.TraceClassVisitor \
    java.lang.Void
```

将会打印出一下内容：

```
// class version 49.0 (49)
```

```
// access flags 49
public final class java/lang/Void {
    // access flags 25
    // signature Ljava/lang/Class<Ljava/lang/Void;>;
    // declaration: java.lang.Class<java.lang.Void>
    public final static Ljava/lang/Class; TYPE
    // access flags 2
    private <init>()V
        ALOAD 0
        INVOKESPECIAL java/lang/Object.<init> ()V
        RETURN
        MAXSTACK = 1
        MAXLOCALS = 1
    // access flags 8
    static <clinit>()V
        LDC "void"
        INVOKESTATIC java/lang/Class.getPrimitiveClass (...)...
        PUTSTATIC java/lang/Void.TYPE : Ljava/lang/Class;
        RETURN
        MAXSTACK = 1
        MAXLOCALS = 0
}
```

上面的代码展示了如何生成一个静态块 static { … }，即<clinit>方法。注意，如果你想在链的某个点追踪一个方法的内容，你可以选择使用 TraceMethodVisitor，而不是用 TraceClassVisitor 来追踪所有的内容：

```
public MethodVisitor visitMethod(int access, String name,
    String desc, String signature, String[] exceptions) {
    MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
    exceptions);
    if (debug && mv != null && ...) { // if this method must be traced
        mv = new TraceMethodVisitor(mv) {
            @Override public void visitEnd() {
                print(aPrintWriter); // print it after it has been visited
            }
        };
    }
    return new MyMethodAdapter(mv);
}
```

上面的代码将会打印出使用 MyMethodAdapter 转换后的内容。

CheckClassAdapter

这个类在前面的章节中也介绍过，用来检查 ClassVisitor 方法是否按照合适的顺序来调用，如果给以合适的参数，它可以用来检查 MethodVisitor 中的方法。也就是说它可以用来检查

MethodVisitor 接口在转换链中是否被正确使用。就像 TraceMethodVisitor 一样，你可以选择使用 CheckMethodAdapter 来检查单个方法而不是所有方法：

```
public MethodVisitor visitMethod(int access, String name,
String desc, String signature, String[] exceptions) {
    MethodVisitor mv = cv.visitMethod(access, name, desc, signature,
    exceptions);
    if (debug && mv != null && ...) { // if this method must be checked
        mv = new CheckMethodAdapter(mv);
    }
    return new MyMethodAdapter(mv);
}
```

上面的代码用来检查 MyMethodAdapter 是否正确使用了 MethodVisitor 接口。但是它不能检查字节码是否正确：如它无法检测 ISTORE 1 ALOAD 1 是无效的。

ASMifierClassVisitor

这个类同样在前面章节中介绍过了，它同样也可以用在方法上。它可以用来了解如何使用 ASM 来生成一些编译后的代码：编写对应的源代码，然后使用 javac 编译，在使用 ASMifierClassVisitor 来访问这个类即可。这样你就可以得到 ASM 代码来生成源码对应的字节码。

3.3.2 AnalyzerAdapter

这个方法适配器基于 visitFrame 访问过的帧，来对比每个指令的栈映射帧。如在 3.1.5 节中解释的一样，visitFrame 方法只能在一个方法中的某些指令之前被调用，这样可以节省空间，并且“其它的帧可以很容易很快地从这些帧推断出来”。这就是这个章节即将做的事情。这个适配器只能工作在那些包含预先计算过的栈映射帧的类上，并且这些类需要使用 java6 或者更高编译器编译。

在 AddTimerAdapter 示例中，这个适配器可以获得在 RETURN 指令之前的操作数栈的大小，从而可以在 visitMaxs 方法中为 maxStack 计算一个最佳的值（事实上，在实际操作中这个方法不推荐使用，因为它没有 COMPUTE_MAXS 有效 ：

```
class AddTimerMethodAdapter2 extends AnalyzerAdapter {
    private int maxStack;
    public AddTimerMethodAdapter2(String owner, int access,
        String name, String desc, MethodVisitor mv) {
        super(owner, access, name, desc, mv);
    }
    @Override public void visitCode() {
        super.visitCode();
        mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
        mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
            "currentTimeMillis", "()J");
        mv.visitInsn(LSUB);
```

```
            mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
            maxStack = 4;
        }
        @Override public void visitInsn(int opcode) {
            if ((opcode >= IRETURN && opcode <= RETURN) || opcode == ATHROW) {
                mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
                mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
                    "currentTimeMillis", "()J");
                mv.visitInsn(LADD);
                mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
                maxStack = Math.max(maxStack, stack.size() + 4);
            }
            super.visitInsn(opcode);
        }
        @Override public void visitMaxs(int maxStack, int maxLocals) {
            super.visitMaxs(Math.max(this.maxStack, maxStack), maxLocals);
        }
}
```

其中 stack 字段是在 AnalyzerAdapter 类中定义的，包含了操作数栈中的类型。更经确定地讲，在一个 visitXxxInsn 方法中，并且在被重写的方法调用之前，这个列表包含了操作数栈的状态。注意，被重写的那个方法也必须被调用，这样 stack 字段才会被正确的更新（以后将使用 super 来代替原始代码中的 mv）

此外，可以通过调用父类的方法来插入新的指令：效果就是 AnalyzerAdapter 将会计算这些指令帧大小。因此，因为这个适配器会基于自己的计算来更新 visitMaxs 方法的参数，我们就不需要自己更新它们了：

```
class AddTimerMethodAdapter3 extends AnalyzerAdapter {
    public AddTimerMethodAdapter3(String owner, int access,
        String name, String desc, MethodVisitor mv) {
        super(owner, access, name, desc, mv);
    }
    @Override public void visitCode() {
        super.visitCode();
        super.visitFieldInsn(GETSTATIC, owner, "timer", "J");
        super.visitMethodInsn(INVOKESTATIC, "java/lang/System",
            "currentTimeMillis", "()J");
        super.visitInsn(LSUB);
        super.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
    }
    @Override public void visitInsn(int opcode) {
        if ((opcode >= IRETURN && opcode <= RETURN) || opcode == ATHROW) {
            super.visitFieldInsn(GETSTATIC, owner, "timer", "J");
            super.visitMethodInsn(INVOKESTATIC, "java/lang/System",
            "currentTimeMillis", "()J");
```

```
                super.visitInsn(LADD);
                super.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
            }
            super.visitInsn(opcode);
        }
}
```

3.3.3 LocalVariablesSorter

这个方法适配器对方法中局部变量以它们出现的顺序进行重新编号。例如，一个包含两个参数的方法，第一个局部变量的索引值大于或者等于 3,那么前面还应该有三个局部变量，以及两个方法参数，它们是不能改变的，因此第一个局部变量的索引值应该是 3，第二个就应该是 4（有疑问？），后面依次类推.这个适配器对于想在方法中插入新的变量会有帮助。如果没有这个适配器，虽然也可以实现添加新的局部变量到代码的最后，但是不幸地是，它们的编号直到 visitMaxs 方法的末尾才可知。

为了展示如何使用这个适配器，假设我们希望通过添加一个局部变量来实现 AddTimerAdapter：

```
public class C {
    public static long timer;
    public void m() throws Exception {
        long t = System.currentTimeMillis();
        Thread.sleep(100);
        timer += System.currentTimeMillis() - t;
    }
}
```

可以通过继承 LocalVariablesSorter，并使用其中的 newLocal 方法，很容易就实现上面的功能。

```
class AddTimerMethodAdapter4 extends LocalVariablesSorter {
    private int time;
    public AddTimerMethodAdapter4(int access, String desc,
        MethodVisitor mv) {
        super(access, desc, mv);
    }
    @Override public void visitCode() {
        super.visitCode();
        mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
            "currentTimeMillis", "()J");
        time = newLocal(Type.LONG_TYPE);
        mv.visitVarInsn(LSTORE, time);
    }
    @Override public void visitInsn(int opcode) {
        if ((opcode >= IRETURN && opcode <= RETURN) || opcode == ATHROW) {
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
```

```
                    "currentTimeMillis", "()J");
            mv.visitVarInsn(LLOAD, time);
            mv.visitInsn(LSUB);
            mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
            mv.visitInsn(LADD);
            mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
        }
        super.visitInsn(opcode);
    }
    @Override public void visitMaxs(int maxStack, int maxLocals) {
        super.visitMaxs(maxStack + 4, maxLocals);
    }
}
```

注意，当局部变量被重新编号以后，之前与方法关联的帧就变为无效了，更不用说插入新的局部变量了。让人感到有希望的是，我们可以避免从头计算这些帧：事实上没有帧被删除或者被添加，只需要对原始帧中的局部变量进行重排序就能获得转换后的帧了。**LocalVariablesSorter** 将会自动进行这些操作。如果你需要对栈映射帧进行增量更新，你可以从这个类的源码中获取灵感。

如你所见，使用一个局部变量并不能解决在之前版本中遇到的问题，就是计算 maxStack 在最坏情况下的值。如果你希望使用 AnalyzerAdapter 来解决这个问题，那么除了 LocalVariablesSorter，你必须通过委托的方式来使用这些适配器而不是继承（因为多继承是不可能的）：

```
class AddTimerMethodAdapter5 extends MethodAdapter {
    public LocalVariablesSorter lvs;
    public AnalyzerAdapter aa;
    private int time;
    private int maxStack;
    public AddTimerMethodAdapter5(MethodVisitor mv) {
        super(mv);
    }
    @Override public void visitCode() {
        mv.visitCode();
        mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
            "currentTimeMillis", "()J");
        time = lvs.newLocal(Type.LONG_TYPE);
        mv.visitVarInsn(LSTORE, time);
        maxStack = 4;
    }
    @Override public void visitInsn(int opcode) {
        if ((opcode >= IRETURN && opcode <= RETURN) || opcode == ATHROW) {
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
```

```
                    "currentTimeMillis", "()J");
            mv.visitVarInsn(LLOAD, time);
            mv.visitInsn(LSUB);
            mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
            mv.visitInsn(LADD);
            mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
            maxStack = Math.max(aa.stack.size() + 4, maxStack);
        }
        mv.visitInsn(opcode);
    }
    @Override public void visitMaxs(int maxStack, int maxLocals) {
        mv.visitMaxs(Math.max(this.maxStack, maxStack), maxLocals);
    }
}
```

为了使用上面的适配器，你必要将 LocalVariablesSorter 和 AnalyzerAdapter 链接起来，然后链接你的适配器：第一个适配器将对局部变量进行排序并更新帧，AnalyzerAdapter 适配器将根据前一步骤中执行结果计算中间帧，这样你的适配器就可以访问那些已经重编号的帧了。这个链可以在 visitMethod 方法中如下构造：

```
mv = cv.visitMethod(access, name, desc, signature, exceptions);
if (!isInterface && mv != null && !name.equals("<init>")) {
    AddTimerMethodAdapter5 at = new AddTimerMethodAdapter5(mv);
    at.aa = new AnalyzerAdapter(owner, access, name, desc, at);
    at.lvs = new LocalVariablesSorter(access, desc, at.aa);
    return at.lvs;
}
```

3.3.4 AdviceAdapter

这个方法适配器是一个抽象的类，可以用来在方法的开始插入代码，也可以在 **RETURN** 和 **ATHROW** 指令之前。它最大的优势就是可以适用于构造方法，在构造方法中，代码不能直接插入到方法的开始，而是插入到调用父类构造方法之后。事实上，这个适配器的大部分代码都是用来检测调用父类的构造方法的。

如果你仔细的查看 3.2.4 章节中的 AddTimerAdapter 类，你会发现 AddTimerMethodAdapter 适配器没有用在构造方法上，就是因为这个问题。通过继承 AdviceAdapter 这个适配器，可以让这个适配器也能够很好的工作在构造方法上（注意，AdviceAdapter 是继承自 LocalVariablesSorter，因此我们也可以很容易地使用一个局部变量）：

```
class AddTimerMethodAdapter6 extends AdviceAdapter {
    public AddTimerMethodAdapter6(int access, String name, String desc,
        MethodVisitor mv) {
        super(mv, access, name, desc);
    }
    @Override protected void onMethodEnter() {
```

```
            mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
                  "currentTimeMillis", "()J");
            mv.visitInsn(LSUB);
            mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
      }
      @Override protected void onMethodExit(int opcode) {
            mv.visitFieldInsn(GETSTATIC, owner, "timer", "J");
            mv.visitMethodInsn(INVOKESTATIC, "java/lang/System",
                  "currentTimeMillis", "()J");
            mv.visitInsn(LADD);
            mv.visitFieldInsn(PUTSTATIC, owner, "timer", "J");
      }
      @Override public void visitMaxs(int maxStack, int maxLocals) {
            super.visitMaxs(maxStack + 4, maxLocals);
      }
}
```