ASM4 使用指南

译者：贾洪峰 jxxy@263.net

本文档由OneAPM赞助翻译
对此技术感兴趣的同学可以投递简历给 mars@oneapm.com

第1章 引言

1.1 动机

程序分析、程序生成和程序转换都是非常有用的技术，可在许多应用环境下使用：

- 程序分析，既可能只是简单的语法分析（syntaxic parsing），也可能是完整的语义分析（sematic analysis），可用于查找应用程序中的潜在 bug、检测未被用到的代码、对代码实施逆向工程，等等。
- 程序生成，在编译器中使用。这些编译器不仅包括传统编译器，还包括用于分布式程序设计的 stub 编译器或 skeleton 编译器，以及 JIT（即时）编译器，等等。
- 程序转换可，用于优化或混淆（obfuscate）程序、向应用程序中插入调试或性能监视代码，用于面向方面的程序设计，等等。

所有这些技术都可针对任意程序设计语言使用，但对于不同语言，其使用的难易程度可能会有所不同。对于 Java 语言，它们可用于 Java 源代码或编译后的 Java 类。在使用经过编译的类时，其好处之一显然就是不需要源代码。因此，程序转换可用于任何应用程序，既包括保密的源代码，也包含商业应用程序。使用已编译类的另一个好处是，有可能在运行时，在马上就要将类加载到 Java 虚拟机之前，对类进行分析、生成或转换（在运行时生成和编译源代码也可以，但其速度很慢，而且需要一个完整的 Java 编译器）。其好处是，诸如 stub 编译器或方面编织器等工具对用户变为透明。

由于程序分析、生成和转换技术的用途众多，所以人们针对许多语言实现了许多用于分析、生成和转换程序的工具，这些语言中就包括 Java 在内。ASM 就是为 Java 语言设计的工具之一，用于进行**运行时**（也是脱机的）类生成与转换。于是，人们设计了 ASM[①]库，用于处理**经过编译的** Java 类。这个库的设计使其尽可能保持**快速**和**小型化**。对于那些在运行时使用 ASM 进行动态类生成或转换的应用程序来说，尽可能提高库的运行速度是非常重要的，这样可以保证这些应用程序的速度不致下降过多。而保持 ASM 库的小型化也非常重要，一方面是为了在内存有限的环境中使用，另一方面，也为了避免使那些使用 ASM 的小型应用程序或库增大过多。

ASM 并不是惟一可生成和转换已编译 Java 类的工具，但它是最新、最高效的工具之一，可从 http://asm.objectweb.org 下载。其主要优点如下：

[①] ASM 的名字没有任何含义：它只是引用 C 语言中的__asm__关键字，这个关键字允许执行一些用汇编语言编写的函数。

- 有一个简单的模块 API，设计完善、使用方便。
- 文档齐全，拥有一个相关的 Eclipse 插件。
- 支持最新的 Java 版本——Java 7。
- 小而快、非常可靠。
- 拥有庞大的用户社区，可以为新用户提供支持。
- 源许可开放，几乎允许任意使用。

1.2 概述

1.2.1 范围

ASM 库的目的是生成、转换和分析以字节数组表示的已编译 Java 类（它们在磁盘中的存储和在 Java 虚拟机中的加载都采用这种字节数组形式）。为此，ASM 提供了一些工具，使用高于字节级别的概念来读写和转换这种字节数组，这些概念包括数值常数、字符串、Java 标识符、Java 类型、Java 类结构元素，等等。注意，ASM 库的范围严格限制于类的读、写、转换和分析。具体来说，类的加载过程就超出了它的范围之外。

1.2.2 模型

ASM 库提供了两个用于生成和转换已编译类的 API，一个是核心 API，以**基于事件**的形式来表示类，另一个是树 API，以**基于对象**的形式来表示类。

在采用基于事件的模型时，类是用一系列事件来表示的，每个事件表示类的一个元素，比如它的一个标头、一个字段、一个方法声明、一条指令，等等。基于事件的 API 定义了一组可能事件，以及这些事件必须遵循的发生顺序，还提供了一个类分析器，为每个被分析元素生成一个事件，还提供一个类写入器，由这些事件的序列生成经过编译的类。

而在采用基于对象的模型时，类用一个对象树表示，每个对象表示类的一部分，比如类本身、一个字段、一个方法、一条指令，等等，每个对象都有一些引用，指向表示其组成部分的对象。基于对象的 API 提供了一种方法，可以将表示一个类的事件序列转换为表示同一个类的对象树，也可以反过来，将对象树表示为等价的事件序列。换言之，基于对象的 API 构建在基于事件的 API 之上。

这两个 API 可以与“用于 XML 的简单 API”（Simple API for XML，SAX）和用于 XML 文档的“文档对象模型（Document Object Model，DOM）API”相比较：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 可在 SAX 的上层提供。

ASM 之所以要提供两个 API，是因为没有哪种 API 是最佳的。实际上，每个 API 都有自己的优缺点：

- 基于事件的 API 要快于基于对象的 API，所需要的内存也较少，因为它不需要在内存中创建和存储用于表示类的对象树（SAX 与 DOM 之间也有同样的差异）。
- 但在使用基于事件的 API 时，类转换的实现可能要更难一些，因为在任意给定时刻，类中只有一个元素可供使用（也就是与当前事件对应的元素），而在使用基于对象的 API 时，可以在内存中获得整个类。

注意，这两个 API 都是仅能同时维护一个类，而且独立于其他类，也就是说，它们不会维护有关类层级结构的信息，如果类的转换影响到其他类，那其他这些类的修改应当由用户负责完成。

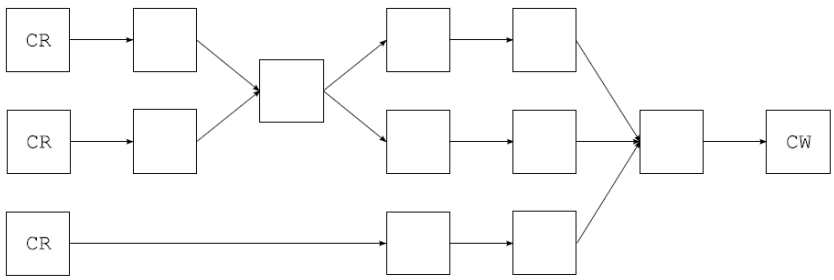
1.2.3 体系结构

ASM 应用程序拥有一个很强壮的体系结构方面（aspect）。事实上，对于基于事件的 API，其组织结构是围绕事件生成器（类分析器）、事件使用器（类写入器）和各种预定义的事件筛选器进行的，在这一结构中可以添加用户定义的生成器、使用器和筛选器。因此，这一 API 的使用分为两个步骤：

- 将事件生成器、筛选器和使用器组件组装为可能很复杂的体系结构，
- 然后启动事件生成器，以执行生成或转换过程。

基于对象的 API 也有一个体系结构方面：实际上，用于操作类树的类生成器或转换器组件是可以组成形成的，它们之间的链接代表着转换的顺序。

尽管典型 ASM 应用程序中的大多数组件体系结构都非常简单，但还是可以想象一下类似于如下所示的复杂体系结构，其中的箭头表示在类分析器、写入器或转换器之间进行的基于事件或基于对象的通信，在整个链中的任何位置，都可能会在基于事件与基于对象的表示之间进行转换：



1.3 组织形式

ASM 库划分为几个包，以几个 jar 文件的形式进行分发：

- **`org.objectweb.asm`** 和 **`org.objectweb.asm.signature`** 包定义了基于事件的 API，并提供了类分析器和写入器组件。它们包含在 asm.jar 存档文件中。
- **`org.objectweb.asm.util`** 包，位于 **`asm-util.jar`** 存档文件中，提供各种基于核心 API 的工具，可以在开发和调试 ASM 应用程序时使用。

- **org.objectweb.asm.commons** 包提供了几个很有用的预定义类转换器，它们大多是基于核心 API 的。这个包包含在 **asm-commons.jar** 存档文件中。
- **org.objectweb.asm.tree** 包，位于 **asm-tree.jar** 存档文件中，定义了基于对象的 API，并提供了一些工具，用于在基于事件和基于对象的表示方法之间进行转换。
- **org.objectweb.asm.tree.analysis** 包提供了一个类分析框架和几个预定义的类分析器，它们以树 API 为基础。这个包包含在 **asm-analysis.jar** 存档文件中。

本文档分为两部分。第一部分介绍核心 API，即 **asm**、**asm-util** 和 **asm-commons** 存档文件。第二部分介绍树 API，即 **asm-tree** 和 **asm-analysis** 存档文件。每部分至少包含该 API 与类相关的一章内容、该 API 与方法相关的一章内容、该 API 与注释、泛型等相关的一章内容。每章都会介绍编程接口及相关的工具与预定义组件。所有示例的源代码都可以从 ASM 网站上获得。

这种组织形式便于循序渐进地介绍类文件特征，但有时需要将同一个 ASM 类的介绍分散到几节中。因此，建议依次阅读本文档。如需有关 ASM API 的参考手册，请使用 Javadoc。

印刷约定

斜体用于强调句子中的元素。（译者注：中文中一般改为加粗显示）

定宽字体用于表示代码段。

粗体定宽字体用于强调代码元素。

斜体定宽字体用于表示标记和代码中的变量部分。

1.4 致谢

感谢 François Horn 在制作本文档期间提供的宝贵评论，这些意见极大地提升了本文档的结构和可读性。

第一部分 核心 API

第2章 类

本章说明如何使用核心 ASM API 来生成和转换经过编译的 Java 类。首先介绍已编译类，然后将利用大量说明性示例，介绍用于生成和转换已编译类的相应 ASM 接口、组件和工具。方法、注释和泛型的内容将在之后各章中说明。

2.1 结构

2.1.1 概述

已编译类的总体结构非常简单。实际上，与原生编译应用程序不同，已编译类中保留了来自源代码的结构信息和几乎所有符号。事实上，已编译类中包含如下各部分：

- 专门一部分，描述类的修饰符（比如 **`public`** 和 **`private`**）、名字、超类、接口和注释。
- 类中声明的每个字段各有一部分。每一部分描述一个字段的修饰符、名字、类型和注释。
- 类中声明的每个方法**及构造器**各有一部分。每一部分描述一个方法的修饰符、名字、返回类型与参数类型、注释。它还以 Java 字节代码指令的形式，包含了该方法的已编译代码。

但在源文件类和已编译类之间还是有一些差异：

- 一个已编译类仅描述一个类，而一个源文件中可以包含几个类。比如，一个源文件描述了一个类，这个类又有一个内部类，那这个源文件会被编译为两个类文件：主类和内部类各一个文件。但是，主类文件中包含对其内部类的**引用**，定义了内部方法的内层类会包含**引用**，引向其封装的方法。
- 已编译类中当然不包含注释（comment），但可以包含类、字段、方法和代码**属性**，可以利用这些属性为相应元素关联更多信息。Java 5 中引入可用于同一目的的注释（annotaion）以后，属性已经变得没有什么用处了。
- 编译类中不包含 **`package`** 和 **`import`** 部分，因此，所有类型名字都必须是完全限定的。

另一个非常重要的结构性差异是已编译类中包含**常量池**（constant pool）部分。这个池是一

个数组，其中包含了在类中出现的所有数值、字符串和类型常量。这些常量仅在这个常量池部分中定义一次，然后可以利用其索引，在类文件中的所有其他各部分进行引用。幸好，ASM 隐藏了与常量池有关的所有细节，所以我们不用再为它操心了。图 2.1 中总结了一个已编译类的整体结构。其确切结构在《Java 虚拟机规范》第 4 节中描述。

<table>
<tr><td colspan="2">修饰符、名字、超类、接口</td></tr>
<tr><td colspan="2">常量池：数值、字符串和类型常量</td></tr>
<tr><td colspan="2">源文件名（可选）</td></tr>
<tr><td colspan="2">封装的类引用</td></tr>
<tr><td colspan="2">注释*</td></tr>
<tr><td colspan="2">属性*</td></tr>
<tr><td>内部类*</td><td>名称</td></tr>
<tr><td rowspan="3">字段*</td><td>修饰符、名字、类型</td></tr>
<tr><td>注释*</td></tr>
<tr><td>属性*</td></tr>
<tr><td rowspan="4">方法*</td><td>修饰符、名字、返回类型与参数类型</td></tr>
<tr><td>注释*</td></tr>
<tr><td>属性*</td></tr>
<tr><td>编译后的代码</td></tr>
</table>

图 2-1 已编译类的整体结构（*表示零个或多个）

另一个重要的差别是 Java 类型在已编译类和源文件类中的表示不同。后面几节将解释它们在已编译类中的表示。

2.1.2 内部名

在许多情况下，一种类型只能是类或接口类型。例如，一个类的超类、由一个类实现的接口，或者由一个方法抛出的异常就不能是基元类型或数组类型，必须是类或接口类型。这些类型在已编译类中用**内部名字**表示。一个类的内部名就是这个类的完全限定名，其中的点号用斜线代替。例如，**String** 的内部名为 **java/lang/String**。

2.1.3 类型描述符

内部名只能用于类或接口类型。所有其他 Java 类型，比如字段类型，在已编译类中都是用**类型描述符**表示的（见图 2.2）。

Java 类型	类型描述符
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 一些 Java 类型的类型描述符

基元类型的描述符是单个字符：**Z** 表示 **boolean**，**C** 表示 **char**，**B** 表示 **byte**，**S** 表示 **short**，**I** 表示 **int**，**F** 表示 **float**，**J** 表示 **long**，**D** 表示 **double**。一个类类型的描述符是这个类的内部名，前面加上字符 **L**，后面跟有一个分号。例如，**String** 的类型描述符为 **Ljava/lang/String;**。而一个数组类型的描述符是一个方括号后面跟有该数组元素类型的描述符。

2.1.4 方法描述符

方法描述符是一个类型描述符列表，它用一个字符串描述一个方法的参数类型和返回类型。方法描述符以左括号开头，然后是每个形参的类型描述符，然后是一个右括号，接下来是返回类型的类型描述符，如果该方法返回 **void**，则是 **V**（方法描述符中不包含方法的名字或参数名）。

源文件中的方法声明	方法描述符
`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 方法描述符举例

一旦知道了类型描述符如何工作，方法描述符的理解就容易了。例如，**`(I)I`** 描述一个方法，它接受一个 **`int`** 类型的参数，返回一个 **`int`**。图 2.3 给出了几个方法描述符示例。

2.2 接口和组件

2.2.1 介绍

用于生成和变转已编译类的 ASM API 是基于 **`ClassVisitor`** 抽象类的（见图 2.4）。这个类中的每个方法都对应于同名的类文件结构部分（见图 2.1）。简单的部分只需一个方法调用就能访问，这个调用返回 **`void`**，其参数描述了这些部分的内容。有些部分的内容可以达到任意长度、任意复杂度，这样的部分可以用一个初始方法调用来访问，返回一个辅助的访问者类。**`visitAnnotation`**、**`visitField`** 和 **`visitMethod`** 方法就是这种情况，它们分别返回 **`AnnotationVisitor`**、**`FieldVisitor`** 和 **`MethodVisitor`**.

```
public abstract class ClassVisitor {
  public ClassVisitor(int api);
  public ClassVisitor(int api, ClassVisitor cv);
  public void visit(int version, int access, String name,
      String signature, String superName, String[] interfaces);
  public void visitSource(String source, String debug);
  public void visitOuterClass(String owner, String name, String desc);
  AnnotationVisitor visitAnnotation(String desc, boolean visible);
  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);
  public MethodVisitor visitMethod(int access, String name, String
desc,
     String signature, String[] exceptions);
  void visitEnd();
}
```

图 2.4 **`ClassVisitor`** 类

针对这些辅助类递归适用同样的原则。例如，**FieldVisitor** 抽象类中的每个方法（见图 2.5）对应于同名的类文件子结构，**visitAnnotation** 返回一个辅助的 **AnnotationVisitor**，和在 **ClassVisitor** 中一样。这些辅助访问者类的创建和使用在随后几章中解释：实际上，本章仅限于只需 **ClassVisitor** 类本身就能解决的简单问题。

```
public abstract class FieldVisitor {
  public FieldVisitor(int api);
  public FieldVisitor(int api, FieldVisitor fv);
  public AnnotationVisitor visitAnnotation(String desc, boolean
visible);
  public void visitAttribute(Attribute attr);
  public void visitEnd();
}
```

图 2.5 **FieldVisitor** 类

ClassVisitor 类的方法必须按以下顺序调用（在这个类的 Javadoc 中规定）：

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

这意味着必须首先调用 **visit**，然后是对 **visitSource** 的最多一个调用，接下来是对 **visitOuterClass** 的最多一个调用，然后是可按任意顺序对 **visitAnnotation** 和 **visitAttribute** 的任意多个访问，接下来是可按任意顺序对 **visitInnerClass**、**visitField** 和 **visitMethod** 的任意多个调用，最后以一个 **visitEnd** 调用结束。

ASM 提供了三个基于 **ClassVisitor** API 的核心组件，用于生成和变化类：

- **ClassReader** 类分析以字节数组形式给出的已编译类，并针对在其 **accept** 方法参数中传送的 **ClassVisitor** 实例，调用相应的 **visitXxx** 方法。这个类可以看作一个事件产生器。
- **ClassWriter** 类是 **ClassVisitor** 抽象类的一个子类，它直接以二进制形式生成编译后的类。它会生成一个字节数组形式的输出，其中包含了已编译类，可以用 **toByteArray** 方法来提取。这个类可以看作一个事件使用器。
- **ClassVisitor** 类将它收到的所有方法调用都委托给另一个 **ClassVisitor** 类。这个类可以看作一个事件筛选器。

接下来的各节将用一些具体示例来说明如何使用这些组件来生成和转换类。

2.2.2 分析类

在分析一个已经存在的类时，惟一必需的组件是 **ClassReader** 组件。让我们用一个例子来说明。假设希望打印一个类的内容，其方式类似于 **javap** 工具。第一步是编写 **ClassVisitor** 类的一个子类，打印它所访问的类的相关信息。下面是一种可能的实现方式，它有些过于简化了：

```
public class ClassPrinter extends ClassVisitor {
```

```
  public ClassPrinter() {
    super(ASM4);
  }
  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("}");
    }
  }
```

第二步是将这个**ClassPrinter**与一个**ClassReader**组件合并在一起，使**ClassReader**产生的事件由我们的**ClassPrinter**使用：

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

第二行创建了一个**ClassReader**，以分析**Runnable**类。在最后一行调用的**accept**方法分析**Runnable**类字节代码，并对**cp**调用相应的**ClassVisitor**方法。结果为以下输出：

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

}
```

注意，构建**ClassReader**实例的方式有若干种。必须读取的类可以像上面一样用名字指定，也可以像字母数组或**InputStream**一样用值来指定。利用**ClassLoader**的

getResourceAsStream 方法，可以获得一个读取类内容的输入流，如下：

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

2.2.3 生成类

为生成一个类，惟一必需的组件是 **ClassWriter** 组件。让我们用一个例子来进行说明。考虑以下接口：

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

可以对 **ClassVisitor** 进行六次方法调用来生成它：

```
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();
```

第一行创建了一个 **ClassWriter** 实例，它实际上将创建类的字节数组表示（构造器参数在下一章解释）。

对 **visit** 方法的调用定义了类的标头。**V1_5** 参数是一个常数，与所有其他 ASM 常量一样，在 ASM **Opcodes** 接口中定义。它指明了类的版本——Java 1.5。**ACC_*XXX*** 常量是与 Java 修饰符对应的标志。这里规定这个类是一个接口，而且它是 **public** 和 **abstract** 的（因为它不能被实例化）。下一个参数以内部形式规定了类的名字（见 2.1.2 节）。回忆一下，已编译类不包含 **Package** 和 **Import** 部分，因此，所有类名都必须是完全限定的。下一个参数对应于泛型（见 4.1 节）。在我们的例子中，这个参数是 **null**，因为这个接口并没有由类型变量进行参数化。第五个参数是内部形式的超类（接口类隐式继承自 **Object**）。最后一个参数是一个数组，其中是被扩展的接口，这些接口由其内部名指定。

接下来对 **visitField** 的三次调用是类似的，用于定义三个接口字段。第一个参数是一组标志，对应于 Java 修饰符。这里规定这些字段是 **public**、**final** 和 **static** 的。第二个参数是字段的名字，与它在源代码中的显示相同。第三个参数是字段的类型，采用类型描述符形式。这里，这些字段是 **int** 字段，它们的描述符是 **I**。第四个参数对应于泛型。在我们的例子中，它是 **null**，因为这些字段类型没有使用泛型。最后一个参数是字段的**常量**值：这个参数必须仅

用于真正的常量字段，也就是 **final static** 字段。对于其他字段，它必须为 **null**。由于此处没有注释，所以立即调用所返回的 **FieldVisitor** 的 **visitEnd** 方法，即对其 **visitAnnotation** 或 **visitAttribute** 方法没有任何调用。

visitMethod 调用用于定义 **compareTo** 方法，同样，第一个参数是一组对应于 Java 修饰符的标志。第二个参数是方法名，与其在源代码中的显示一样。第三个参数是方法的描述符。第四个参数对应于泛型。在我们的例子中，它是 **null**，因为这个方法没有使用泛型。最后一个参数是一个数组，其中包括可由该方法抛出的异常，这些异常由其内部名指明。它在这里为 null，因为这个方法没有声明任何异常。**visitMethod** 方法返回 **MethodVisitor**（见图 3.4），可用于定义该方法的注释和属性，最重要的是这个方法的代码。这里，由于没有注释，而且这个方法是抽象的，所以我们立即调用所返回的 **MethodVisitor** 的 **visitEnd** 方法。

对 **visitEnd** 的最后一个调用是为了通知 **cw**：这个类已经结束，对 **toByteArray** 的调用用于以字节数组的形式提取它。

1. 使用生成的类

前面的字节数组可以存储在一个 **Comparable.class** 文件中，供以后使用。或者，也可以用 **ClassLoader** 动态加载它。一种方法是定义一个 **ClassLoader** 子类，它的 **defineClass** 方法是公有的：

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

然后，可以用下面的代码直接调用所生成的类：

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

另一种加载已生成类的方法可能更清晰一些，那就是定义一个 **ClassLoader** 子类，它的 **findClass** 方法被重写，以在运行过程中生成所请求的类：

```
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);
  }
}
```

事实上，所生成类的使用方式取决于上下文，这已经超出了 ASM API 的范围。如果你正在编写编译器，那类生成过程将由一个抽象语法树驱动，这个语法树代表将要编译的程序，而生成的类将被存储在磁盘上。如果你正在编写动态代理类生成器或方面编织器，那将会以这种或那种方式使用一个 **ClassLoader**。

2.2.4 转换类

到目前为止，**ClassReader** 和 **ClassWriter** 组件都是单独使用的。这些事件是“人工”产生，并且由 **ClassWriter** 直接使用，或者与之对称地，它们由 **ClassReader** 产生，然后“人工”使用，也就是由自定义的 **ClassVisitor** 实现使用。当这些组件一同使用时，事情开始变得真正有意义起来。第一步是将 **ClassReader** 产生的事件转给 **ClassWriter**。其结果是，类编写器重新构建了由类读取器分析的类：

```
byte[] b1 = ...;
ClassWriter cw = new ClassWriter(0);
ClassReader cr = new ClassReader(b1);
cr.accept(cw, 0);
byte[] b2 = cw.toByteArray(); // b2 和 b1 表示同一个类
```

这本身并没有什么真正的意义（还有其他更简单的方法可以用来复制一个字节数组!），但等一等。下一步是在类读取器和类写入器之间引入一个 **ClassVisitor**：

```
byte[] b1 = ...;
ClassWriter cw = new ClassWriter(0);
// cv 将所有事件转发给 cw
ClassVisitor cv = new ClassVisitor(ASM4, cw) { };
ClassReader cr = new ClassReader(b1);
cr.accept(cv, 0);
byte[] b2 = cw.toByteArray(); // b2 与 b1 表示同一个类
```

图 2.6 给出了与上述代码相对应的体系结构，其中的组件用方框表示，事件用箭头表示（其中的垂直时间线与程序图中一样）。

图 2.6 转换链

但结果并没有改变，因为 **ClassVisitor** 事件筛选器没有筛选任何东西。但现在，为了能够转换一个类，只需重写一些方法，筛选一些事件就足够了。例如，考虑下面的 **ClassVisitor** 子类：

```
public class ChangeVersionAdapter extends ClassVisitor {
  public ChangeVersionAdapter(ClassVisitor cv) {
    super(ASM4, 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 **ClassVisitor** 类的一个方法。结果，所有调用都被不加改变地转发到传送给构造器的类访问器 **cv**，只有对 **visit** 方法的调用除外，在转发它时，对类版本号进行了修

改。相应的程序图在图 2.7 中给出。

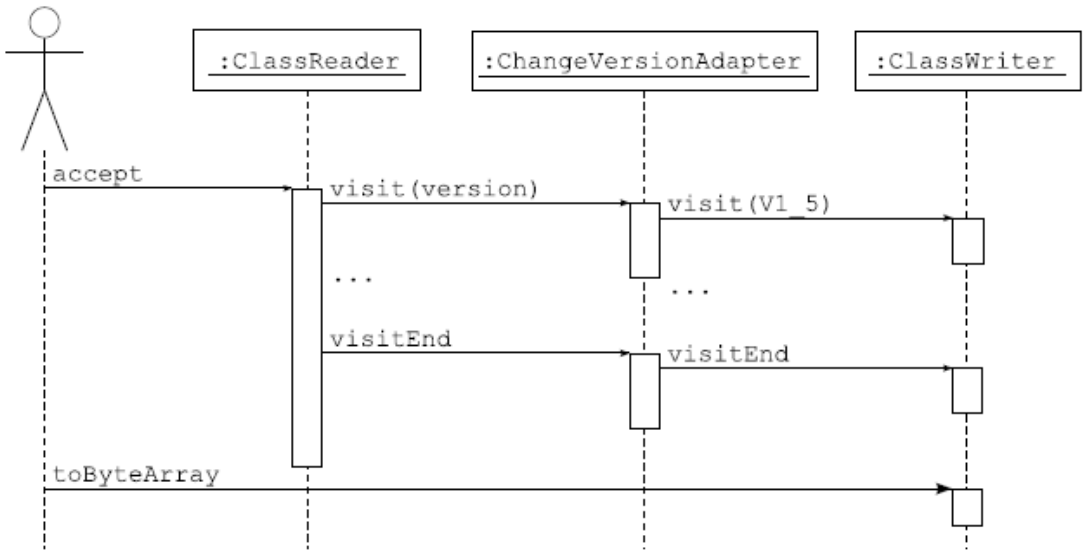


图 2.7 **ChangeVersionAdapter** 的程序图

通过修改 **visit** 方法的其他参数，可以实现其他转换，而不仅仅是修改类的版本。例如，可以向实现接口的列表中添加一个接口。还可以改变类的名字，**但进行这种改变所需要做的工作要多得多，不只是改变 visit 方法的 name 参数了**。实际上，类的名字可以出现在一个已编译类的许多不同地方，要真正实现类的重命名，必须修改类中出现的**所有**这些类名字。

2. 优化

前面的转换只修改了原类的四个字节。但是，在使用上面的代码时，整个 **b1** 均被分析，并利用相应的事件从头从头构建了 **b2**，这种做法的效率不是很高。如果将 **b1** 中不被转换的部分直接复制到 **b2** 中，不对其分析，也不生成相应的事件，其效率就会高得多。ASM 自动为方法执行这一优化：

- 在 **ClassReader** 组件的 **accept** 方法参数中传送了 **ClassVisitor**，如果 **ClassReader** 检测到这个 **ClassVisitor** 返回的 **MethodVisitor** 来自一个 **ClassWriter**，这意味着这个方法的内容将不会被转换，事实上，应用程序甚至不会**看到**其内容。
- 在这种情况下，**ClassReader** 组件不会分析这个方法的内容，不会生成相应事件，只是复制 **ClassWriter** 中表示这个方法的字节数组。

如果 **ClassReader** 和 **ClassWriter** 组件拥有对对方的引用，则由它们进行这种优化，可设置如下：

```
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();
```

执行这一优化后，由于 **ChangeVersionAdapter** 没有转换任何方法，所以以上代码的速度可以达到之前代码的**两倍**。对于转换部分或全部方法的常见转换，这一速度提升幅度可能要小一些，但仍然是很可观的：实际上在 10%到 20%的量级。遗憾的是，这一优化需要将原类中定义的所有常量都复制到转换后的类中。对于那些**增加**字段、方法或指令的转换来说，这一点不成问题，但对于那些要**移除**或**重命名**许多类成员的转换来说，这一优化将导致类文件大于未优化时的情况。因此，建议仅对“增加性”转换应用这一优化。

3. 使用转换后的类

如上节所述，转换后的类 **b2** 可以存储在磁盘上，或者用 **ClassLoader** 加载。但在 **ClassLoader** 中执行的类转换只能转换由这个类加载器加载的类。如果希望转换**所有**类，则必须将转换放在 **ClassFileTransformer** 内部，见 **java.lang.instrument** 包中的定义（更多细节，请参阅这个软件包的文档）：

```
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

上一节用于转换类版本的方法当然也可用于 **ClassVisitor** 类的其他方法。例如，通过改变 **visitField** 和 **visitMethod** 方法的 **access** 或 **name** 参数，可以改变一个字段或一个方法的修饰字段或名字。另外，除了在转发的方法调用中使用经过修改的参数之外，还可以选择根本**不**转发该调用。其效果就是相应的类元素被**移除**。

例如，下面的类适配器移除了有关外部类及内部类的信息，还删除了一个源文件的名字，也就是由其编译这个类的源文件（所得到的类仍然具有全部功能，因为删除的这些元素仅用于调试目的）。这一移除操作是通过在适当的访问方法中不转发任何内容而实现的：

```
public class RemoveDebugAdapter extends ClassVisitor {
  public RemoveDebugAdapter(ClassVisitor cv) {
    super(ASM4, 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) {
  }
}
```

这一策略对于字段和方法是无效的，因为 **visitField** 和 **visitMethod** 方法必须返回一个结果。要移除字段或方法，不得转发方法调用，并向调用者返回 **null**。例如，下面的类适配器移除了一个方法，该方法由其名字及描述符指明（仅使用名字不足以标识一个方法，因为一个类中可能包含若干个具有不同参数的同名方法）：

```
public class RemoveMethodAdapter extends ClassVisitor {
  private String mName;
  private String mDesc;
  public RemoveMethodAdapter(
      ClassVisitor cv, String mName, String mDesc) {
    super(ASM4, 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)) {
      // 不要委托至下一个访问器 -> 这样将移除该方法
      return null;
    }
    return cv.visitMethod(access, name, desc, signature, exceptions);
  }
}
```

2.2.6 增加类成员

上述讨论的是少转发一些收到的调用，我们还可以多“转发”一些调用，也就是发出的调用数多于收到的调用，其效果就是**增加**了类成员。新的调用可以插在原方法调用之间的若干位置，只要遵守各个 **visitXxx** 必须遵循的调用顺序即可（见 2.2.1 节）。

例如，如果要向一个类中添加一个字段，必须在原方法调用之间添加对 **visitField** 的一个新调用，而且必须将这个新调用放在类适配器的一个访问方法中。比如，不能在 **visit** 方法中这样做，因为这样可能会导致对 **visitField** 的调用之后跟有 **visitSource**、**visitOuterClass**、**visitAnnotation** 或 **visitAttribute**，这是无效的。出于同样的原因，不能将这个新调用放在 **visitSource**、**visitOuterClass**、**visitAnnotation** 或 **visitAttribute** 方法中. 仅有的可能位置是 **visitInnerClass**、**visitField**、**visitMethod** 或 **visitEnd** 方法。

如果将这个新调用放在 **visitEnd** 方法中，那这个字段将总会被添加（除非增加显式条件），因为这个方法总会被调用。如果将它放在 **visitField** 或 **visitMethod** 中，将会添加**几个**字段：原类中的每个字段和方法各有一个相应的字段。这两种解决方案都可能发挥应有的作用；具体取决于你的需求。例如，可以仅添加一个计数器字段，用于计算对一个对象的调用次数，也可以为每个方法添加一个计数器，用于分别计算对每个方法的调用次数。

注意：事实上，惟一真正正确的解决方案是在 **visitEnd** 方法中添加更多调用，以添加新成员。实际上，一个类中不得包含重复成员，要确保一个新成员没有重复成员，惟一方法就是将它与所有已有成员进行对比，只有在 **visitEnd** 方法中访问了所有这些成员后才能完成这一工作。这种做法是相当受限制的。在实践中，使用程序员不大可能使用的生成名，比如**_counter$**或**_4B7F_ i** 就足以避免重复成员了，并不需要将它们添加到 **visitEnd** 中。注意，在第一章曾经讨论过，树 API 没有这一限制：可以在任意时刻向使用这个 API 的转换中添加新成员。

为了举例阐述以上讨论，下面给出一个类适配器，它会向类中添加一个字段，除非这个字段已经存在：

```
public class AddFieldAdapter extends ClassVisitor {
  private int fAcc;
  private String fName;
  private String fDesc;
  private boolean isFieldPresent;
  public AddFieldAdapter(ClassVisitor cv, int fAcc, String fName,
      String fDesc) {
    super(ASM4, 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();
  }
}
```

这个字段被添加在 **visitEnd** 方法中。**visitField** 方法未被重写为修改已有字段或删除一个字段，只是检测一下我们希望添加的字段是否已经存在。注意 **visitEnd** 方法中在调用 **fv.visitEnd()**之前的 **fv != null** 检测：这是因为一个类访问器可以在 **visitField** 中返回 **null**，在上一节已经看到这一点。

2.2.7 转换链

到目前为止，我们已经看到一些由 **ClassReader**、类适配器和 **ClassWriter** 组成的简单转换链。当然可以使用更为复杂的转换链，将几个类适配器链接在一起。将几个适配器链接在一起，就可以组成几个独立的类转换，以完成复杂转换。还要注意，转换链不一定是线性的。我们

可以编写一个 **ClassVisitor**，将接收到的所有方法调用同时转发给几个 **ClassVisitor**：

```
public class MultiClassAdapter extends ClassVisitor {
  protected ClassVisitor[] cvs;
  public MultiClassAdapter(ClassVisitor[] cvs) {
    super(ASM4);
    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);
    }
  }
  ...
}
```

反过来，几个类适配器可以委托至同一 **ClassVisitor**（这需要采取一些预防措施，确保比如 **visit** 和 **visitEnd** 针对这个 **ClassVisitor** 恰好仅被调用一次）。因此，诸如图 2.8 所示的这样一个转换链是完全可行的。

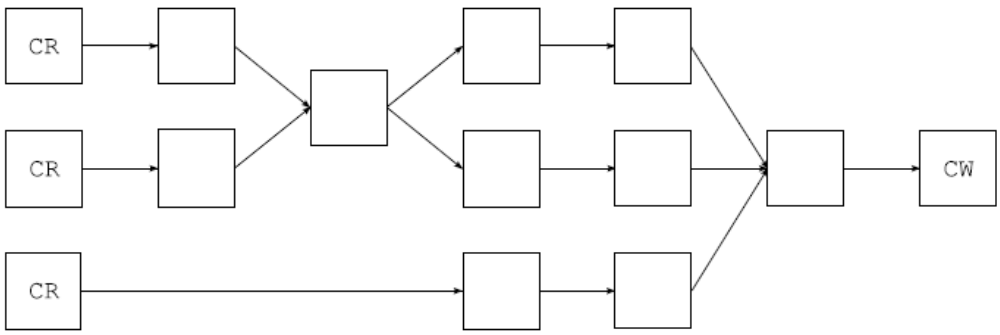


图 2.8 一个复杂转换

2.3 工具

除了 **ClassVisitor** 类和相关的 **ClassReader**、**ClassWriter** 组件之外，ASM 还在 **org.objectweb.asm.util** 包中提供了几个工具，这些工具在开发类生成器或适配器时可能非常有用，但在运行时不需要它们。ASM 还提供了一个实用类，用于在运行时处理内部名、类型描述符和方法描述符。所有这些工具都将在下面介绍。

2.3.1 Type

在前几节已经看到，ASM API 公开 Java 类型的形式就是它们在已编译类中的存储形式，也就是说，作为内部特性或类型描述符。也可以按照它们在源代码中的形式来公开它们，使代码更便于阅读。但这样就需要在 **ClassReader** 和 **ClassWriter** 中的两种表示形式之间进行系统转换，从而使性能降低。这就是为什么 ASM 没有透明地将内部名和类型描述符转换为它们等价

的源代码形式。但它提供了 **Type** 类，可以在必要时进行手动转换。

一个 **Type** 对象表示一种 Java 类型，既可以由类型描述符构造，也可以由 **Class** 对象构建。**Type** 类还包含表示基元类型的静态变量。例如，**Type.INT_TYPE** 是表示 **int** 类型的 **Type** 对象。

getInternalName 方法返回一个 **Type** 的内部名。例如，**Type.getType(String.class). getInternalName()** 给出 **String** 类的内部名，即 **"java/lang/String"**。这一方法只能对类或接口类型使用。

getDescriptor 方法返回一个 **Type** 的描述符。比如，在代码中可以不使用 **"Ljava/lang/String;"**，而是使用 **Type.getType(String.class). getDescriptor()**。或者，可以不使用 **I**，而是使用 **Type.INT_TYPE.getDescriptor()**。

Type 对象还可以表示方法类型。这种对象既可以从一个方法描述符构建，也可以由 **Method** 对象构建。**getDescriptor** 方法返回与这一类型对应的方法描述符。此外，**getArgumentTypes** 和 **getReturnType** 方法可用于获取与一个方法的参数类型和返回类型相对应的 **Type** 对象。例如，**Type.getArgumentTypes("(I)V")** 返回一个仅有一个元素 **Type.INT_TYPE** 的数组。与此类似，调用 **Type.getReturnType("(I)V")** 将返回 **Type.VOID_TYPE** 对象。

2.3.2 TraceClassVisitor

要确认所生成或转换后的类符合你的预期，**ClassWriter** 返回的字母数组并没有什么真正的用处，因为它对人类来说是不可读的。如果有文本表示形式，那使用起来就容易多了。这正是 **TraceClassVisitor** 类提供的东西。从名字可以看出，这个类扩展了 **ClassVisitor** 类，并生成所访问类的文本表示。因此，我们不是用 **ClassWriter** 来生成类，而是使用 **TraceClassVisitor**，以获得关于实际所生成内容的一个可读轨迹。甚至可以同时使用这两者，这样要更好一些。除了其默认行为之外，**TraceClassVisitor** 实际上还可以将对其方法的所有调用委托给另一个访问器，比如 **ClassWriter**：

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

这一代码创建了一个 **TraceClassVisitor**，将它自己接收到的所有调用都委托给 **cw**，然后将这些调用的一份文本表示打印到 **printWriter**。例如，如果在 2.2.3 节的例子中使用 **TraceClassVisitor**，将会得出：

```
// 类版本号 49.0 (49)
// 访问标志 1537
public abstract interface pkg/Comparable implements pkg/Mesurable {
  // 访问标志 25
  public final static I LESS = -1
```

```
  //访问标志 25
  public final static I EQUAL = 0
  //访问标志 25
  public final static I GREATER = 1
  //访问标志 1025
  public abstract compareTo(Ljava/lang/Object;)I
}
```

注意，可以在生成链或转换链的任意位置使用 **TraceClassVisitor**，以查看在链中这一点发生了什么，并非一定要恰好在 **ClassWriter** 之前使用。还要注意，有了这个适配器生成的类的文本表示形式，可能很轻松地用 **String.equals()** 来对比两个类。

2.3.3 CheckClassAdapter

ClassWriter 类并不会核实对其方法的调用顺序是否恰当，以及参数是否有效。因此，有可能会生成一些被 Java 虚拟机验证器拒绝的无效类。为了尽可能提前检测出部分此类错误，可以使用 **CheckClassAdapter** 类。和 **TraceClassVisitor** 类似，这个类也扩展了 **ClassVisitor** 类，并将对其方法的所有调用都委托到另一个 **ClassVisitor**，比如一个 **TraceClassVisitor** 或一个 **ClassWriter**。但是，这个类并不会打印所访问类的文本表示，而是验证其对方法的调用顺序是否适当，参数是否有效，然后才会委托给下一个访问器。当发生错误时，会抛出 **IllegalStateException** 或 **IllegalArgumentException**。

为核对一个类，打印这个类的文本表示形式，最终创建一个字节数组表示形式，应当使用类似于如下代码：

```
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，如果以不同顺序将这些类访问器链在一起，那它们执行的操作也将以不同顺序完成。例如，利用以下代码，这些核对工作将在轨迹**之后**进行：

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

和使用 **TraceClassVisitor** 时一样，也可以在一个生成链或转换链的任意位置使用 **CheckClassAdapter**，以查看该链中这一点的类，而不一定只是恰好在 **ClassWriter** 之前使用。

2.3.4 ASMifier

这个类为 **TraceClassVisitor** 工具提供了一种替代后端（该工具在默认情况下使用 **Textifier** 后端，生成如上所示类型的输出）。这个后端使 **TraceClassVisitor** 类的每个方

法都会打印用于调用它的 Java 代码。例如，调用 **visitEnd**() 方法将打印 **cv.visitEnd**();。其结果是，当一个具有 **ASMifier** 后端的 **TraceClassVisitor** 访问器访问一个类时，它会打印用 ASM 生成这个类的源代码。如果用这个访问器来访问一个已经存在的类，那这一点是很有用的。例如，如果你不知道如何用 ASM 生成某个已编译类，可以编写相应的源代码，用 **javac** 编译它，并用 **ASMifier** 来访问这个编译后的类。将会得到生成这个已编译类的 ASM 代码！

ASMifier 类也可以在命令行中使用。例如，使用以下命令，

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

将会生成一些代码，经过缩进后，这些代码就是如下模样：

```
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();
  }
}
```

第3章 方法

本章解释如何用核心 ASM API 生成和转换已编译方法。首先介绍编译后的方法，然后介绍用于生成和转换它们的相应 ASM 接口、组件和工具，并给出大量说明性示例。

3.1 结构

在编译类的内部，方法的代码存储为一系列的**字节码**指令。为生成和转换类，最根本的就是要了解这些指令，并理解它们是如何工作的。本节将对这些指令进行全面概述，这些内容足以开始编写简单的类生成器与转换器代码。如需完整定义，应当阅读 Java 虚拟机规范。

3.1.1 执行模型

在介绍字节代码指令之前，有必要先来介绍 Java 虚拟机执行模型。我们知道，Java 代码是在**线程**内部执行的。每个线程都有自己的执行栈，栈由**帧**组成。每个帧表示一个方法调用：每次调用一个方法时，会将一个新帧压入当前线程的执行栈。当方法返回时，或者是正常返回，或者是因为异常返回，会将这个帧从执行栈中弹出，执行过程在发出调用的方法中继续进行（这个方法的帧现在位于栈的顶端）。

每一帧包括两部分：一个局部变量部分和一个操作数栈部分。**局部变量**部分包含可根据索引以随机顺序访问的变量。由名字可以看出，**操作数栈**部分是一个栈，其中包含了供字节代码指令用作操作数的值。这意味着这个栈中的值只能按照“后入先出”顺序访问。不要将操作数栈和线程的执行栈相混淆：执行栈中的每一帧都包含**自己**的操作数栈。

局部变量部分与操作数栈部分的大小取决于方法的代码。这一大小是在编译时计算的，并随字节代码指令一起存储在已编译类中。因此，对于对应于某一给定方法调用的所有帧，其局部变量与操作数栈部分的大小相同，但对应于不同方法的帧，这一大小可能不同。

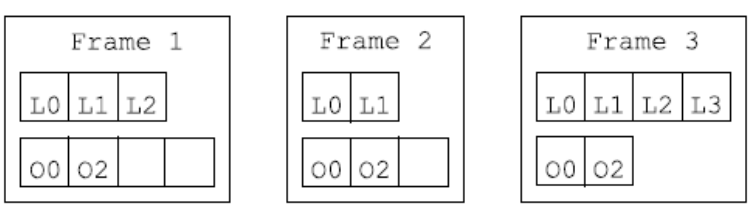


图 3.1 一个具有 3 帧的执行栈

图中文字：Frame，帧

图 3.1 给出了一个具有 3 帧的示例执行栈。第一帧包含 3 个局部变量，其操作数栈的最大值为 4，其中包含两个值。第二帧包含 2 个局部变量，操作数栈中有两个值。最后是第三帧，位于执行栈的顶端，包含 4 个局部变量和两个操作数。

在创建一个帧时，会将其初始化，提供一个空栈，并用目标对象 **this**（对于非静态方法）及该方法的参数来初始化其局部变量。例如，调用方法 **a.equals(b)** 将创建一帧，它有一个空栈，前两个局部变量被初始化为 **a** 和 **b**（其他局部变量未被初始化）。

局部变量部分和操作数栈部分中的每个槽（slot）可以保存除 **long** 和 **double** 变量之外的任意 Java 值。**long** 和 **double** 变量需要两个槽。这使局部变量的管理变得复杂：例如，第 *i* 个方法参数不一定存储在局部变量 *i* 中。例如，调用 **Math.max(1L, 2L)** 创建一个帧，**1L** 值位于前两个局部变量槽中，值 **2L** 存储在第三和第四个槽中。

3.1.2 字节代码指令

字节代码指令由一个标识该指令的操作码和固定数目的参数组成：

- **操作码**是一个无符号字节值——即字节代码名，由助记符号标识。例如，操作码 0 用助记符号 **NOP** 表示，对应于不做任何操作的指令。
- **参数**是静态值，确定了精确的指令行为。它们紧跟在操作码之后给出。比如 **GOTO 标记** 指令（其操作码的值为 167）以一个指明下一条待执行指令的标记作为参数**标记**。不要将指令参数与指令操作数相混淆：参数值是静态已知的，存储在编译后的代码中，而操作数值来自操作数栈，只有到运行时才能知道。

字节代码指令可以分为两类：一小组指令，设计用来在局部变量和操作数栈之间传送值；其他一些指令仅用于操作数栈：它们从栈中弹出一些值，根据这些值计算一个结果，并将它压回栈中。

ILOAD、**LLOAD**、**FLOAD**、**DLOAD** 和 **ALOAD** 指令读取一个局部变量，并将它的值压到操作数栈中。它们的参数是必须读取的局部变量的索引 *i*。**ILOAD** 用于加载一个 **boolean**、**byte**、**char**、**short** 或 **int** 局部变量。**LLOAD**、**FLOAD** 和 **DLOAD** 分别用于加载 **long**、**float** 或 **double** 值。（**LLOAD** 和 **DLOAD** 实际加载两个槽 *i* 和 *i*+1）。最后，**ALOAD** 用于加载任意非基元值，即对象和数组引用。与之对应，**ISTORE**、**LSTORE**、**FSTORE**、**DSTORE** 和 **ASTORE** 指令从操作数栈中弹出一个值，并将它存储在由其索引 i 指定的局部变量中。

可以看到，***x*LOAD** 和 ***x*STORE** 指令被赋入了类型（事实上，下面将要看出，几乎所有指令都被赋予了类型）。它用于确保不会执行非法转换。实际上，将一个值存储在局部变量中，然后再以不同类型加载它，是非法的。例如，**ISTORE 1 ALOAD 1** 序列是非法的——它允许将一个任意内存位置存储在局部变量 **1** 中，并将这个地址转换为对象引用！但是，如果向一个局部变量中存储一个值，而这个值的类型不同于该局部变量中存储的当前值，却是完全合法的。这意味

着一个局部变量的类型，即这个局部变量中所存值的类型可以在方法执行期间发生变化。

上面已经说过，所有其他字节代码指令都仅对操作数栈有效。它们可以划分为以下类别（见附件 A.1）：

栈 这些指令用于处理栈上的值：**POP** 弹出栈顶部的值，**DUP** 压入顶部栈值的一个副本，**SWAP** 弹出两个值，并按逆序压入它们，等等。

常量 这些指令在操作数栈压入一个常量值：**ACONST_NULL** 压入 **null**，**ICONST_0** 压入 **int** 值 0，**FCONST_0** 压入 **0f**，**DCONST_0** 压入 **0d**，**BIPUSH** *b* 压入字节值 *b*，**SIPUSH** *s* 压入 **short** 值 *s*，**LDC** *cst* 压入任意 **int**、**float**、**long**、**double**、**String** 或 **class**[①] 常量 **cst**，等等。

算术与逻辑 这些指令从操作数栈弹出数值，合并它们，并将结果压入栈中。它们没有任何参数。**xADD**、**xSUB**、**xMUL**、**xDIV** 和 **xREM** 对应于**+**、**-**、*****、**/**和**%**运算，其中 **x** 为 **I**、**L**、**F** 或 **D** 之一。类似地，还有其他对应于**<<**、**>>**、**>>>**、**|**、**&**和**^**运算的指令，用于处理 **int** 和 **long** 值。

类型变换 这些指令从栈中弹出一个值，将其转换为另一类型，并将结果压入栈中。它们对应于 Java 中的类型转换表达式。**I2F**, **F2D**, **L2D** 等将数值由一种数值类型转换为另一种类型。**CHECKCAST** *t* 将一个引用值转换为类型 *t*。

对象 这些指令用于创建对象、锁定它们、检测它们的类型，等等。例如，**NEW** *type* 指令将一个 *type* 类型的新对象压入栈中（其中 *type* 是一个内部名）。

字段 这些指令读或写一个字段的值。**GETFIELD** *owner name desc* 弹出一个对象引用，并压和其 *name* 字段中的值。**PUTFIELD** *owner name desc* 弹出一个值和一个对象引用，并将这个值存储在它的 *name* 字段中。在这两种情况下，该对象都必须是 *owner* 类型，它的字段必须为 *desc* 类型。**GETSTATIC** 和 **PUTSTATIC** 是类似指令，但用于静态字段。

方法 这些指令调用一个方法或一个构造器。它们弹出值的个数等于其方法参数个数加 1（用于目标对象），并压回方法调用的结果。**INVOKEVIRTUAL** *owner name desc* 调用在类 *owner* 中定义的 *name* 方法，其方法描述符为 *desc*。**INVOKESTATIC** 用于静态方法，**INVOKESPECIAL** 用于私有方法和构造器，**INVOKEINTERFACE** 用于接口中定义的方法。最后，对于 Java 7 中的类，**INVOKEDYNAMIC** 用于新动态方法调用机制。

数组 这些指令用于读写数组中的值。**xALOAD** 指令弹出一个索引和一个数组，并压入此索引处数组元素的值。**xASTORE** 指令弹出一个值、一个索引和一个数组，并将这个值存储在该数组的这一索引处。这里的 *x* 可以是 **I**、**L**、**F**、**D** 或 **A**，还可以是 **B**、**C** 或 **S**。

跳转 这些指令无条件地或者在某一条件为真时跳转到一条任意指令。它们用于编译 **if**、**for**、**do**、**while**、**break** 和 **continue** 指令。例如，**IFEQ** *label* 从栈中弹出一个 **int** 值，如果这个值为 0，则跳转到由这个 *label* 指定的指令处（否则，正常执行下一条指令）。还有许多其他跳转指令，比如 **IFNE** 或 **IFGE**。最后，**TABLESWITCH** 和

[①] 对应于 *identifier*.class Java 语法。

LOOKUPSWITCH 对应于 **switch** Java 指令。

返回 最后，***x*RETURN** 和 **RETURN** 指令用于终止一个方法的执行，并将其结果返回给调用者。**RETURN** 用于返回 **void** 的方法，***x*RETURN** 用于其他方法。

3.1.3 示例

让我们看一些基本示例，具体体会一下字节代码指令是如何工作的。考虑下面的 **bean** 类：

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

getter 方法的字节代码为：

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

第一条指令读取局部变量 0（它在为这个方法调用创建帧期间被初始化为 **this**），并将这个值压入操作数栈中。第二个指令从栈中弹出这个值，即 **this**，并将这个对象的 **f** 字段压入栈中，即 **this.f**。最后一条指令从栈中弹出这个值，并将其返回给调用者。图 3.2 中给出了这个方法执行帧的持续状态。

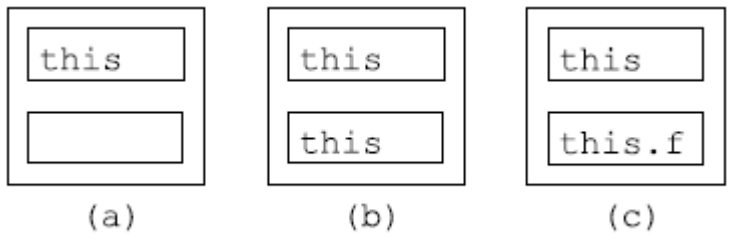


图 3.2 **getF** 方法的持续帧状态：a) 初始状态，b) 在 **ALOAD 0** 之后，c) 在 **GETFIELD** 之后

setter 方法的字节代码：

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

和之前一样，第一条指令将 **this** 压入操作数栈。第二条指令压入局部变量 **1**，在为这个方法调用创建帧期间，以 **f** 参数初始化该变量。第三条指令弹出这两个值，并将 **int** 值存储在被引用对象的 **f** 字段中，即存储在 **this.f** 中。最后一条指令在源代码中是隐式的，但在编译后

的代码中却是强制的，销毁当前执行帧，并返回调用者。这个方法执行帧的持续状态如图 3.3 所示。

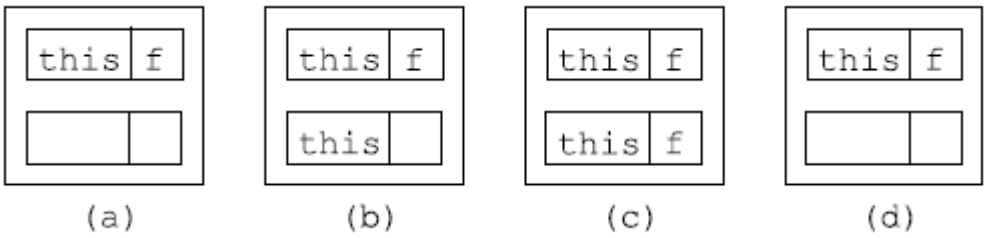


图 3.3 **setF** 方法的持续状态：a) 初始状态，b) 在 **ALOAD 0** 之后，c)在 **ILOAD 1** 之后，d) 在 **PUTFIELD** 之后

Bean 类还有一个默认的公有构造器，由于程序员没有定义显式的构造器，所以它是由编译器生成的。这个默认的公有构造器被生成为 **Bean() { super(); }**。这个构造器的字节代码如下：

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

第一条指令将 **this** 压入操作数栈中。第二条指令从栈中弹出这个值，并调用在 **Object** 对象中定义的**<init>**方法。这对应于 **super()** 调用，也就是对超类 **Object** 构造器的调用。在这里可以看到，在已编译类和源类中对构造器的命名是不同的：在编译类中，它们总是被命名为**<init>**，而在源类中，它们的名字与定义它们的类同名。最后一条指令返回调用者。

现在让我们考虑一个稍为复杂一点的 **setter** 方法：

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

这个新 **setter** 方法的字节代码如下：

```
  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
```

第一条指令将初始化为 **f** 的局部变量 **1** 压入操作数栈。**IFLT** 指令从栈中弹出这个值，并将

它与 0 进行比较。如果它**小于**（**LT**）0，则跳转到由 **label** 标记指定的指令，否则不做任何事情，继续执行下一条指令。接下来的三条指令与 **setF** 方法中相同。**GOTO** 指令无条件跳转到由 ***end*** 标记指定的指令，也就是 **RETURN** 指令。***label*** 和 ***end*** 标记之间的指令创建和抛出一个异常：**NEW** 指令创建一个异常对象，并将它压入操作数栈中。**DUP** 指令在栈中重复这个值。**INVOKESPECIAL** 指令弹出这两个副本之一，并对其调用异常构造器。最后，**ATHROW** 指令弹出剩下的副本，并将它作为异常抛出（所以不会继续执行下一条指令）。

3.1.4 异常处理器

不存在用于捕获异常的字节代码：而是将一个方法的字节代码与一个**异常处理器**列表关联在一起，这个列表规定了在某方法中一给定部分抛出异常时必须执行的代码。异常处理器类似于 **try catch** 块：它有一个范围，也就是与 **try** 代码块内容相对应的一个指令序列，还有一个处理器，对应于 **catch** 块中的内容。这个范围由一个起始标记和一个终止标记指定，处理器由一个起始标记指定。比如下面的源代码：

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

可被编译为

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

Try 和 **catch** 标记之间的代码对应于 **try** 块，而 ***catch*** 标记之后的代码对应于 **catch**。**TRYCATCHBLOCK** 行指定了一个异常处理器，覆盖了 ***try*** 和 ***catch*** 标记之间的范围，有一个开始于 ***catch*** 标记的处理器，用于处理一些异常，这些异常的类是 **InterruptedException** 的子类。这意味着，如果在 ***try*** 和 ***catch*** 之间抛出了这样一个异常，栈将被清空，异常被压入这个空栈中，执行过程在 **catch** 处继续。

3.1.5 帧

除了字节代码指令之外，用 Java 6 或更高版本编译的类中还包含一组**栈映射帧**，用于加快 Java 虚拟机中类验证过程的速度。栈映射帧给出一个方法的执行帧在执行过程中某一时刻的状态。更准确地说，它给出了在就要执行某一特定字节代码指令之前，每个局部变量槽和每个操作数栈槽中包含的值的**类型**。

例如，如果考虑上一节的 **getF** 方法，可以定义三个栈映射帧，给出执行帧在即将执行 **ALOAD**、即将执行 **GETFIELD** 和即将执行 **IRETURN** 之前的状态。这三个栈映射帧对应于图 3.2 给出的三种情况，可描述如下，其中第一个方括号中的类型对应于局部变量，其他类型对应于操作数栈：

如下代码**之前**的执行帧状态	指令
[pkg/Bean] []	**ALOAD 0**
[pkg/Bean] [pkg/Bean]	**GETFIELD**
[pkg/Bean] [I]	**IRETURN**

可以对 **checkAndSetF** 方法进行相同操作：

如下代码**之前**的执行帧状态	指令
[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] [未初始化（标记）]	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，它与前面的方法均类似。这是一种仅在栈映射帧中使用的特殊类型，它指定了一个对象，已经为其分配了内存，但还没有调用其构造器。参数规定了创建此对象的指令。对于这个类型的值，只能调用一种方法，那就是构造器。在调用它时，在帧中**出现的所有这一类型**都被代以一个实际类型，这里是 **IllegalArgumentException**。栈映射帧可使用三种其他特殊类型：**UNINITIALIZED_THIS** 是构造器中局部变量 0 的初始类型，**TOP** 对应于一个未定义的值，而 **NULL** 对应于 **null**。

上文曾经说过，从 Java 6 开始，除了字节代码之外，已编译类中还包含了一组栈映射帧。为节省空间，已编译方法中并没有为每条指令包含一个帧：事实上，它仅为那些对应于跳转目标或异常处理器的指令，或者跟在无条件跳转指令之后的指令包含帧。事实上，可以轻松、快速地由这些帧推断出其他帧。

在 **checkAndSetF** 方法的情景中，这意味着仅存储两个帧：一个用于 **NEW** 指令，因为它是 **IFLT** 指令的目标，还因为它跟在无条件跳转 **GOTO** 指令之后，另一个用于 **RETURN** 指令，因为它是 **GOTO** 指令的目标，还因为它跟在“无条件跳转”**ATHROW** 指令之后。

为节省更多空间，对每一帧都进行压缩：仅存储它与前一帧的差别，而初始帧根本不用存储，可以轻松地由方法参数类型推导得出。在 **checkAndSetF** 方法中，必须存储的两帧是相同的，都等于初始帧，所以它们被存储为单字节值，由 **F_SAME** 助记符表示。可以在与这些帧相关联的字节代码指令之前给出这些帧。这就给出了 **F_SAME** 方法的最终字节代码：

```
  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 接口与组件

3.2.1 介绍

用于生成和转换已编译方法的 ASM API 是基于 **MethodVisitor** 抽象类的（见图 3.4），它由 **ClassVisitor** 的 **visitMethod** 方法返回。除了一些与注释和调试信息有关的方法之外(这些方法在下一章解释)，这个类为每个字节代码指令类别定义了一个方法，其依据就是这些指令的参数个数和类型(这些类别**并非**对应于 3.1.2 节给出的类别)。这些方法必须按以下顺序调用(在 **MethodVisitor** 接口的 **Javadoc** 中还规定了其他一些约束条件)：

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

这就意味着，对于非抽象方法，如果存在注释和属性的话，必须首先访问它们，然后是该方法的字节代码。对于这些方法，其代码必须按**顺序**访问，位于对 **visitCode** 的调用（有且仅有一个调用）与对 **visitMaxs** 的调用（有且仅有一个调用）之间。

```
abstract class MethodVisitor{ // public accessors ommited
MethodVisitor(int api);
MethodVisitor(int api, MethodVisitor mv);
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 visitInvokeDynamicInsn(String name, String desc, Handle bsm,
Object... bsmArgs);
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** 类

于是，**visitCode** 和 **visitMaxs** 方法可用于检测该方法的字节代码在一个事件序列中的开始与结束。和类的情况一样，**visitEnd** 方法也必须在最后调用，用于检测一个方法在一个事件序列中的结束。

可以将 **ClassVisitor** 和 **MethodVisitor** 类合并，生成完整的类：

```
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，并不一定要在完成一个方法之后才能开始访问另一个方法。事实上，**MethodVisitor** 实例是完全独立的，可按任意顺序使用（只要还没有调用 **cv.visitEnd()**）：

```
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** API 的核心组件，用于生成和转换方法：

- **ClassReader** 类分析已编译方法的内容，在其 **accept** 方法的参数中传送了 **ClassVisitor**，**ClassReader** 类将针对这一 **ClassVisitor** 返回的 **MethodVisitor** 对象调用相应方法。
- **ClassWriter** 的 **visitMethod** 方法返回 **MethodVisitor** 接口的一个实现，它直接以二进制形式生成已编译方法。
- **MethodVisitor** 类将它接收到的所有方法调用委托给另一个 **MethodVisitor** 方法。可以将它看作一个事件筛选器。

1. ClassWriter 选项

在 3.1.5 节已经看到，为一个方法计算栈映射帧并不是非常容易：必须计算所有帧，找出与跳转目标相对应的帧，或者跳在无条件跳转之后的帧，最后压缩剩余帧。与此类似，为一个方法计算局部变量与操作数栈部分的大小要容易一些，但依然算不上非常容易。

幸好 ASM 能为我们完成这一计算。在创建 **ClassWriter** 时，可以指定必须自动计算哪些内容：

- 在使用 **new ClassWriter(0)** 时，不会自动计算任何东西。必须自行计算帧、局部变量与操作数栈的大小。
- 在使用 **new ClassWriter(ClassWriter.COMPUTE_MAXS)** 时，将为你计算局部变量与操作数栈部分的大小。还是必须调用 **visitMaxs**，但可以使用任何参数：它们将被忽略并重新计算。使用这一选项时，仍然必须自行计算这些帧。
- 在 **new ClassWriter(ClassWriter.COMPUTE_FRAMES)** 时，一切都是自动计算。不再需要调用 **visitFrame**，但仍然必须调用 **visitMaxs**（参数将被忽略并重新计算）。

这些选项的使用很方便，但有一个代价：**COMPUTE_MAXS** 选项使 **ClassWriter** 的速度降低 10%，而使用 **COMPUTE_FRAMES** 选项则使其降低一半。这必须与我们自行计算时所耗费的时间进行比较：在特定情况下，经常会存在一些比 ASM 所用算法更容易、更快速的计算方法，但 ASM 使用的算法必须能够处理所有情况。

注意，如果选择自行计算这些帧，可以让 **ClassWriter** 为你执行压缩步骤。为此，只需要用 **visitFrame(F_NEW, nLocals, locals, nStack, stack)** 访问未压缩帧，其中的 **nLocals** 和 **nStack** 是局部变量的个数和操作数栈的大小，**locals** 和 **stack** 是包含相应类型的数组（更多细节请参阅 **Javadoc**）。

还要注意，为了自动计算帧，有时需要计算两个给定类的公共超类。默认情况下，**ClassWriter** 类会在 **getCommonSuperClass** 方法中进行这一计算，它会将两个类加载到 JVM 中，并使用反射 API。如果我们正在生成几个相互引用的类，那可能会导致问题，因为被引用的类可能尚未存在。在这种情况下，可以重写 **getCommonSuperClass** 方法来解决这一问题。

3.2.2 生成方法

如果 **mv** 是一个 **MethodVisitor**，则 3.1.3 节定义的 **getF** 方法的字节代码可以用以下方法调用生成：

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

第一个调用启动字节代码的生成过程。然后是三个调用，生成这一方法的三条指令（可以看出，字节代码与 ASM API 之间的映射非常简单）。对 **visitMaxs** 的调用必须在已经访问了所有这些指令后执行。它用于为这个方法的执行帧定义局部变量和操作数栈部分的大小。在 3.1.3 节可以看出，这些大小为每部分 1 个槽，最后一次调用用于结束此方法的生成过程。

setF 方法和构造器的字节代码可以用一种类似方法生成。一个更有意义的示例是 **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，可以看到恰好映射到 3.1.5 节末尾所示字节代码的方法调用：每条指令、标记或帧分别有个调用（仅有的例外是 **label** 和 **end Label** 对象的声明和构造）。

注意：Label 对象规定了跟在这一标记的 **visitLabel** 之后的**指令**。例如，**end** 规定了 **RETURN** 指令，而不是随后马上要访问的帧，因为它不是一条指令。用几条标记指定同一指令是完全合法的，但一个标记只能恰好指定一条指令。换句话说，有可能用不同标记对 **visitLabel** 进行连续调用，但一条指令中的一个标记则必须用 **visitLabel** 恰好访问一次。最后一条约束是，标记不能共享，每个方法都必须拥有自己的标记。

3.2.3 转换方法

你现在应当已经猜到，方法可以像类一样进行转换，也就是使用一个方法适配器将它收到的方法调用转发出去，并进行一些修改：改变参数可用于改变各具体指令；不转发某一收到的调用将删除一条指令；在接收到的调用之间插入调用，将增加新的指令。**MethodVisitor** 类提供了这样一种方法适配器的基本实现，它只是转发它接收到的所有方法，而未做任何其他事情。

为了理解可以如何使用方法适配器，让我们考虑一种非常简单的适配器，删除方法中的 **NOP** 指令（因为它们不做任何事情，所以删除它们没有任何问题）：

```
public class RemoveNopAdapter extends MethodVisitor {
  public RemoveNopAdapter(MethodVisitor mv) {
    super(ASM4, mv);
  }
  @Override
  public void visitInsn(int opcode) {
    if (opcode != NOP) {
      mv.visitInsn(opcode);
    }
  }
}
```

这个适配器可以在一个类适配器内部使用，如下所示：

```
public class RemoveNopClassAdapter extends ClassVisitor {
  public RemoveNopClassAdapter(ClassVisitor cv) {
    super(ASM4, 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;
  }
}
```

换言之，类适配器只是构造一个方法适配器（封装链中下一个类访问器返回的方法访问器），并返回这个适配器。其效果就是构造了一个类似于类适配器链的方法适配器链（见图 3.5）。

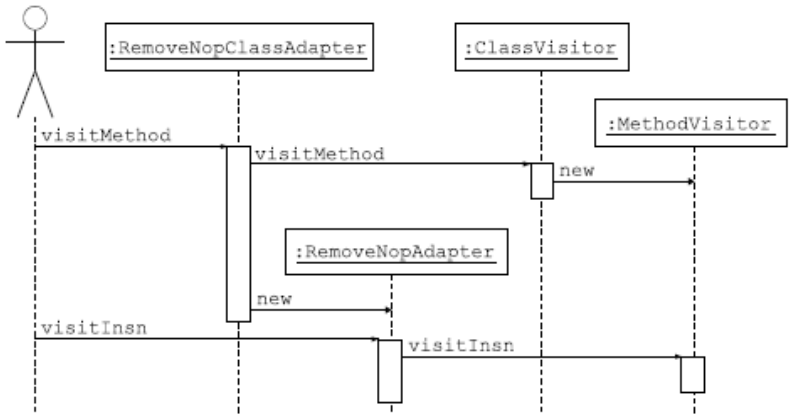


图 3.5 **RemoveNopAdapter** 的程序图

但注意，这种相似性并非强制的：完全有可能构造一个与类适配器链不相似的方法适配器链。每种方法甚至还可以有一个不同的方法适配器链。例如，类适配器可以选择仅删除方法中的 **NOP**，而不移除构造器中的该指令。可以执行如下：

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

在这种情况下，构造器的适配器链更短一些。与之相反，构造器的适配器链也可以更长一些，在 **visitMethod** 内部创建几个链接在一起的适配器。方法适配器链的拓扑结构甚至都可以不同于类适配器。例如，类适配器可能是线性的，而方法适配器链具有分支：

```
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);
}
```

现在已经明白了如何使用方法适配器，将它们合并在一个类适配器内部，现在就来看看如何实现一个比 **RemoveNopAdapter** 更有意义的适配器。

3.2.4 无状态转换

假设我们需要测量一个程序中的每个类所花费的时间。我们需要在每个类中添加一个静态计时器字段，并需要将这个类中每个方法的执行时间添加到这个计时器字段中。换句话说，有这样一个类 **C**：

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

我们希望将它转换为：

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

为了了解可以如何在 ASM 中实现它，可以编译这两个类，并针对这两个版本比较 **TraceClassVisitor** 的输出（或者是使用默认的 **Textifier** 后端，或者是使用 **ASMifier** 后端）。使用默认后端时，得到下面的差异之处（以粗体表示）：

```
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
```

可以看到，我们必须在方法的开头增加四条指令，在返回指令之前添加四条其他指令。还需要更新操作数栈的最大尺寸。此方法代码的开头部分用 **visitCode** 方法访问。因此，可以通过重写方法适配器的这一方法，添加前四条指令：

```
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");
}
```

其中的 **owner** 必须被设定为所转换类的名字。现在必须在任意 **RETURN** 之前添加其他四条指令，还要在任何 ***x*RETURN** 或 **ATHROW** 之前添加，它们都是终止该方法执行过程的指令。这些指令没有任何参数，因此在 **visitInsn** 方法中访问。于是，可以重写这一方法，以增加指令：

```
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，必须更新操作数栈的最大大小。我们添加的指令压入两个 **long** 值，因此需要操作数

栈中的四个槽。在此方法的开头，操作数栈初始为空，所以我们知道在开头添加的四条指令需要一个大小为 4 的栈。还知道所插入的代码不会改变栈的状态（因为它弹出的值的数目与压入的数目相同）。因此，如果原代码需要一个大小为 *s* 的栈，那转换后的方法所需栈的最大大小为 *max*(4, *s*)。遗憾的是，我们还在返回指令前面添加了四条指令，我们并不知道操作数栈恰在执行这些指令之前时的大小。只知道它小于或等于 *s*。因此，我们只能说，在返回指令之前添加的代码可能要求操作数栈的大小达到 *s*+4。这种最糟情景在实际中很少发生：使用常见编译器时，**RETURN** 之前的操作数栈仅包含返回值，即，它的大小最多为 0、1 或 2。但如果希望处理所有可能情景，那就需要考虑最糟情景。[①]必须重写 **visitMaxs** 方法如下：

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

当然，也可以不需要为最大栈大小操心，而是依赖 **COMPUTE_MAXS** 选项，此外，它会计算最优值，而不是最差情景中的值。但对于这种简单的转换，以人工更新 **maxStack** 并不需要花费太多精力。

现在就出现一个很有意义的问题：栈映射帧怎么样呢？原代码不包含任何帧，转换后的代码也没有包含，但这是因为我们用作示例的特定代码造成的吗？是否在某些情况下必须更新帧呢？答案是否定的，因为 1)插入的代码并没有改变操作数栈，2) 插入代码中没有包含跳转指令，3) 原代码的跳转指令（或者更正式地说，是控制流图）没有被修改。这意味着原帧没有发生变化，而且不需要为插入代码存储新帧，所以压缩后的原帧也没有发生变化。

现在可以将所有元素一起放入相关联的 ClassVisitor 和 MethodVisitor 子类中：

```
public class AddTimerAdapter extends ClassVisitor {
  private String owner;
  private boolean isInterface;
  public AddTimerAdapter(ClassVisitor cv) {
    super(ASM4, 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) {
```

[①] 幸好，并不一定要给出最优操作数栈大小。有可能给出任何大于或等于这个最优值的值，尽管这样可能会浪费该线程执行栈上的内存。

```
        fv.visitEnd();
      }
    }
    cv.visitEnd();
  }
  class AddTimerMethodAdapter extends MethodVisitor {
    public AddTimerMethodAdapter(MethodVisitor mv) {
      super(ASM4, 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);
    }
  }
}
```

这个类适配器用于实例化方法适配器（构造器除外），还用于添加计时器字段，并将被转换的类的名字存储在一个可以由方法适配器访问的字段中。

3.2.5 有状态转换

上一节看到的转换是局部的，不会依赖于在当前指令之前访问的指令：在开头添加的代码总是相同的，而且总会被添加，对于在每个 **RETURN** 指令之前添加的代码也是如此。这种转换称为**无状态**转换。它们的实现很简单，但只有最简单的转换具有这一性质。

更复杂的转换需要记忆在当前指令之前已访问指令的状态。例如，考虑这样一个转换，它将删除所有出现的 **ICONST_0 IADD** 序列，这个序列的操作就是加入 0，没有什么实际效果。显然，在访问一条 **IADD** 指令时，只有当上一条被访问的指令是 **ICONST_0** 时，才必须删除该指令。这就要求在方法适配器中存储状态。因此，这种转换被称为**有状态**转换。

让我们更仔细地研究一下这个例子。在访问 **ICONST_0** 时，只有当**下一条**指令是 **IADD** 时才必须将其删除。问题是，下一条指令还是未知的。解决方法是将是否删除它的决定推迟到下一条指令：如果下一指令是 **IADD**，则删除两条指令，否则，发出 **ICONST_0** 和当前指令。

要实现一些删除或替代某一指令序列的转换，比较方便的做法是引入一个 **MethodVisitor** 子类，它的 **visit*Xxx* Insn** 方法调用一个公用的 **visitInsn()** 方法：

```
public abstract class PatternMethodAdapter extends MethodVisitor {
  protected final static int SEEN_NOTHING = 0;
  protected int state;
  public PatternMethodAdapter(int api, MethodVisitor mv) {
    super(api, 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();
}
```

然后，上述转换可实现如下：

```
public class RemoveAddZeroAdapter extends PatternMethodAdapter {
  private static int SEEN_ICONST_0 = 1;
  public RemoveAddZeroAdapter(MethodVisitor mv) {
    super(ASM4, 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(int)方法首先判断是否已经检测到该序列。在这种情况下，它重新初始化**state**，并立即返回，其效果就是删除该序列。在其他情况下，它会调用公用的 **visitInsn** 方法，如果 **ICONST_0** 是**最后一条**被访问序列，它就会发出该指令。于是，如果**当前**指令是**ICONST_0**，它会记住这个事实并返回，延迟关于这一指令的决定。在所有其他情况下，当前指令都被转发到下一访问器。

1. 标记和帧

在前几节已经看到，对标记和帧的访问是恰在它们的相关指令之前进行。换句话说，尽管它们本身并不是指令，但它们是与指令同时受到访问的。这对于检测**指令**序列的转换会有影响，但这一影响实际上是一种优势。事实上，如果删除的指令之一是一条跳转指令的目标，会发生什么

情况呢？如果某一指令可能跳转到 **ICONST_0**，这意味着有一个指定这一指令的标记。在删除了这两条指令后，这个标记将指向跟在被删除 **IADD** 之后的指令，这正是我们希望的。但如果某一指令可能跳转到 **IADD**，我们就不能删除这个指令序列（不能确保在这一跳转之前，已经在栈中压入了一个 0）。幸好，在这种情况下，**ICONST_0** 和 **IADD** 之间必然有一个标记，可以很轻松地检测到它。

这一推理过程对于栈映射帧是一样的：如果访问介于两条指令之间的一个栈映射帧，那就不能删除它们。要处理这两种情况，可以将标记和帧看作是模型匹配算法中的指令。这一点可以在 **PatternMethodAdapter** 中完成（注意，**visitMaxs** 也会调用公用的 **visitInsn** 方法；它用于处理的情景是：方法的末尾是必须被检测序列的一个前缀）：

```
public abstract class PatternMethodAdapter extends MethodVisitor {
  ...
  @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);
  }
}
```

在下一章将会看到，编译后的方法中可能包含有关源文件行号的信息，比如用于异常栈轨迹。这一信息用 **visitLineNumber** 方法访问，它也与指令同时被调用。但是，在一个指令序列的中间给出行号，对于转换或删除该指令的可能性不会产生任何影响。解决方法是在模式匹配算法中完全忽略它们。

2. 一个更复杂的例子

上面的例子可以很轻松地推广到更复杂的指令序列。例如，考虑一个转换，它会删除对字段进行自我赋值的操作，这种操作通常是因为键入错误，比如 **f = f;**，或者是在字节代码中，**ALOAD 0 ALOAD 0 GETFIELD f PUTFIELD f**。在实现这一转换之前，最好是将状态机设计为能够识别这一序列（见图 3.6）。

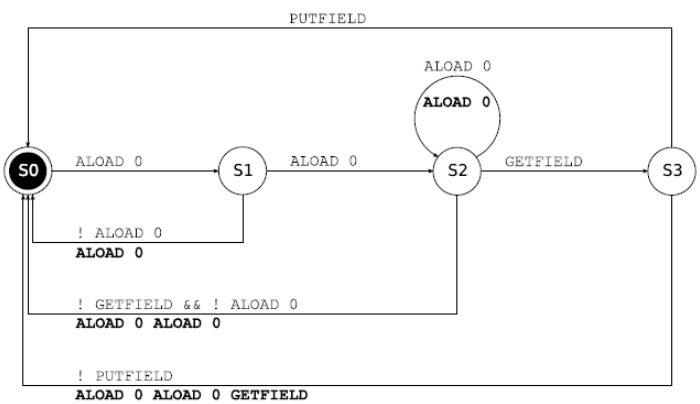


图 3.6 **ALOAD 0 ALOAD 0 GETFIELD f PUTFIELD f** 的状态机

每个转换都标有一个条件（当前指令的值）和一个操作（必须发出的指令序列，以粗体表示）。例如，如果当前指令不是 **ALOAD 0**，则由 **S1** 转换到 **S0**。在这种情况下，导致进入这一状态的 **ALOAD 0** 将被发出。注意从 **S2** 到其自身的转换：在发现三个或三个以上的连续 **ALOAD 0** 时会发生这一情况。在这种情况下，将停留在已经访问两个 **ALOAD 0** 的状态中，并发出第三个 **ALOAD 0**。找到状态机之后，相应方法适配器的编写就简单了。（8 种 Switch 情景对应于图中的 8 种转换）：

```
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;
      }
    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，出于和 3.2.4 节中 **AddTimerAdapter** 同样的原因，本节给出的有状态转换也不需要转换栈映射帧：原帧在转换后仍然有效。它们甚至不需要转换局部变量和操作数栈大小。最后，还必须注意，有状态转换并不限于检测和转换指令序列的转换。许多其他类型的转换也是有状态的。比如，下一节介绍的方法适配器就属于这种情景。

3.3 工具

org.objectweb.asm.commons 包中包含了一些预定义的方法适配器，可用于定义我们自己的适配器。这一节将介绍其中的三个，并用 3.2.4 节的 **AddTimerAdapter** 示例说明如何使用它们。我们还说说明，如何利用上一章看到的工具来简化方法生成或转换。

3.3.1 基本工具

2.3 节介绍的工具也可用于方法。

1. Type

许多字节代码指令，比如 ***x*LOAD**、***x*ADD** 或 ***x*RETURN** 依赖于将它们应用于哪种类型。**Type** 类提供了一个 **getOpcode** 方法，可用于为这些指令获取与一给定类型相对应的操作码。这一方法的参数是一个 **int** 类型的操作码，针对哪种类型调用该方法，则返回该哪种类型的操作码。例如 **t.getOpcode(IMUL)**，若 **t** 等于 **Type.FLOAT_TYPE**，则返回 **FMUL**。

2. TraceClassVisitor

这个类在上一章已经介绍过，它打印它所访问类的文本表示，**包括类的方法的文本表示**，其方式非常类似于这一章使用的方式。因此，可以将它用来跟踪在一个转换链中任意点处所生成或

所转换方法的内容。例如：

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

将输出：

```
// 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
}
```

它说明如何生成一个静态块 **static { ... }**，也就是用**<clinit>**方法（用于 **CLass INITializer**）。注意，如果希望跟踪某一个方法在链中某一点处的内容，而不是跟踪类的所有内容，可以用 **TraceMethodVisitor** 代替 **TraceClassVisitor**（在这种情况下，必须显式指定后端；这里使用了一个 **Textifier**）：

```
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 && ...) { // 如果必须跟踪此方法
    Printer p = new Textifier(ASM4) {
      @Override public void visitMethodEnd() {
        print(aPrintWriter); // 在其被访问后输出它
      }
    };
    mv = new TraceMethodVisitor(mv, p);
  }
  return new MyMethodAdapter(mv);
}
```

这一代码输出该方法经 **MyMethodAdapter** 转换过后的结果。

3. CheckClassAdapter

这个类也已经在上一章介绍过，它检查 **ClassVisitor** 方法的调用顺序是否适当，参数是否有效，所做的工作与 **MethodVisitor** 方法相同。因此，可用于检查 **MethodVisitor** API 在一个转换链中任意点的使用是否正常。和 **TraceMethodVisitor** 类似，可以用

CheckMethodAdapter 类来检查一个方法，而不是检查它的整个类：

```
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 && ...) { // 如果必须检查这个方法
    mv = new CheckMethodAdapter(mv);
  }
  return new MyMethodAdapter(mv);
}
```

这一代码验证 **MyMethodAdapter** 正确地使用了 **MethodVisitor** API。但要注意，这一适配器并**没有**验证字节代码是正确的：例如，它没有检测出 **ISTORE 1 ALOAD 1** 是无效的。实际上，如果使用 **CheckMethodAdapter** 的其他构造器（见 Javadoc），并且在 **visitMaxs** 中提供有效的 **maxStack** 和 **maxLocals** 参数，那这种错误是**可以**被检测出来的。

4. ASMifier

这个类已经在上一章介绍过，也用于处理方法的内容。利用它，可以知道如何用 ASM 生成一些编译后的代码：只需要用 Java 编写相应的源代码，用 javac 编译它，然后用 **ASMifier** 访问这个类。你会得到 ASM 代码，以生成与源代码相对应的字节代码。

3.3.2 AnalyzerAdapter

这个方法适配器根据 **visitFrame** 中访问的帧，计算每条指令之前的栈映射帧。实际上，如 3.1.5 节中的解释，**visitFrame** 仅在方法中的一些特定指令前调用，一方面是为了节省空间，另一方面也是因为“其他帧可以轻松快速地由这些帧推导得出”。这就是这个适配器所做的工作。当然，它仅对那些包含预计算栈映射帧的类有效，也就是对于用 Java 6 或更高版本编译的有效（或者用一个使用 **COMPUTE_FRAMES** 选项的 ASM 适配器升级到 Java 6）。

在我们的 **AddTimerAdapter** 示例中，这个适配器可用于获得操作数栈恰在 **RETURN** 指令之前的大小，从而允许为 **visitMaxs** 中的 **maxStack** 计算一个最优的已转换值（事实上，在实践中并不建议使用这一方法，因为它的效率要远低于使用 **COMPUTE_MAXS**）：

```
class AddTimerMethodAdapter2 extends AnalyzerAdapter {
  private int maxStack;
  public AddTimerMethodAdapter2(String owner, int access,
       String name, String desc, MethodVisitor mv) {
    super(ASM4, 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);
    }
  }
```

stack 字段在 **AnalyzerAdapter** 类中定义，包含操作数栈中的类型。更准确地说，在一个 **visit*Xxx* Insn** 中，且在调用被重写的方法之前，它会列出操作数栈正好在这条指令之前的状态。注意，**必须**调用被重写的方法，使 **stack** 字段被正确更新（因此，用 **super** 代替源代码中的 **mv**）。

或者，也可以通过调用超类中的方法来插入新指令：其方法就是这些指令的帧将由 **AnalyzerAdapter** 计算，由于这个适配器会根据它计算的帧来更新 **visitMaxs** 的参数，所以我们不需要自己来更新它们：

```
class AddTimerMethodAdapter3 extends AnalyzerAdapter {
  public AddTimerMethodAdapter3(String owner, int access,
      String name, String desc, MethodVisitor mv) {
    super(ASM4, 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

这个方法适配器将一个方法中使用的局部变量按照它们在这个方法中的出现顺序重新进行编号。例如，在一个有两个参数的方法中，第一个被读取或写入且索引大于或等于 3 的局部变量（前三个局部变量对应于 this 及两个方法参数，因此不会发生变化）被赋予索引 3，第二个被赋予索引 4，以此类推。在向一个方法中插入新的局部变量时，这个适配器很有用。没有这个适配

器，就需要在所有已有局部变量之后添加新的局部变量，但遗憾的是，在 **visitMaxs** 中，要直到方法的末尾处才能知道这些局部变量的编号。

为说明如何使用这个适配器，假定我们希望使用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;
  }
}
```

这一点很容易做到：只需扩展 **LocalVariablesSorter**，并使用这个类中定义的 **newLocal** 方法。

```
class AddTimerMethodAdapter4 extends LocalVariablesSorter{
  private int time;
  public AddTimerMethodAdapter4(int access, String desc,
      MethodVisitor mv) {
    super(ASM4, 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，在对局部变量重新编号后，与该方法相关联的原帧变为无效，在插入新局部变量后更不必说了。幸好，还是可能避免从头重新计算这些帧的：事实上，并不存在必须添加或删除的帧，只需对原帧中局部变量的内容进行重新排序，为转换后的方法获得帧就“足够”了。**LocalVariablesSorter** 会自动负责完成。如果还需要为你的方法适配器进行增量栈映射帧更新，可以由这个类的源代码中获得灵感。

前面曾经说过，这个类的原版本中存在关于最糟情景下 **maxStack** 取值的问题，在上面可以看出，使用局部变量并不能解决这个问题。如果希望用 **AnalyzerAdapter** 解决这个问题，除了 **LocalVariablesSorter** 之外，必须通过委托使用这些适配器，而不是通过继承（因为不可能存在多个继承）：

```
class AddTimerMethodAdapter5 extends MethodVisitor{
  public LocalVariablesSorter lvs;
  public AnalyzerAdapter aa;
  private int time;
  private int maxStack;
  public AddTimerMethodAdapter5(MethodVisitor mv) {
    super(ASM4, 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);
  }
}
```

为使用这个适配器，必须将一个 **LocalVariablesSorter** 链接到一个 **AnalyzerAdapter**，再将它自身连接到你的适配器：第一个适配器将对局部变量排序，并相应地更新帧，分析适配器将计算中间帧，在此过程中会考虑上一个适配器中完成的重新编号，你的适配器将可以访问这些重新编号的中间帧。这个链接可以在 **visitMethod** 中构造如下：

```
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类，可用于在一个方法的开头以及恰在任意 **RETURN** 或 **ATHROW** 指令之前插入代码。它的主要好处就是对于构造器也是有效的，在构造器中，不能将代码恰好插入到构造器的开头，而是插在对超构造器的调用之后。事实上，这个适配器的大多数代码都专门用于检测对这个超构造器的调用。

仔细研究 3.2.4 节中的 **AddTimerAdapter** 类将会看到，**AddTimerMethodAdapter** 因为这一原因而未被用于构造器。这一方法适配器从 **AdviceAdapter** 继承而来，可以对其进行改

进，以便对于构造器同样有效（注意，**AdviceAdapter** 继承自 **LocalVariablesSorter**，所以也可以轻松使用一个局部变量）：

```
class AddTimerMethodAdapter6 extends AdviceAdapter{
  public AddTimerMethodAdapter6(int access, String name, String desc,
      MethodVisitor mv) {
    super(ASM4, 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);
  }
}
```

第4章 元数据

本章解释如何用核心 API 生成和转换编译后的 Java 类元数据，比如注释。每一节都首先介绍一种元数据类型，然后给出用于生成和转换这些元数据的相应 ASM 接口、组件和工具，并给出一些说明性示例。

4.1 泛型

诸如 **`List<E>`**之类的泛型类，以及使用它们的类，包含了有关它们所声明或使用的泛型的信息。这一信息不是由字节代码指令在运行时使用，但可通过反射 API 访问。它还可以供编译器使用，以进行分离编译。

4.1.1 结构

出于后向兼容的原因，有关泛型的信息没有存储在类型或方法描述符中（它们的定义远早于 Java 5 中对泛型的引入），而是保存在称为类型、方法和类**签名**的类似构造中。在涉及泛型时，除了描述符之外，这些签名也会存储在类、字段和方法**声明**中（泛型不会影响方法的字节代码：编译器用它们执行静态类型检查，但会在必要时重新引入类型转换，就像这些方法未被使用一样进行编译）。

与类型和方法描述符不同，类型签名的语法非常复杂，这也是因为泛型的递归本质造成的（一个泛型可以将另一泛型作为参数——例如，考虑 **`List<List<E>>`**）。其语法由以下规则给出（有关这些规则的完整描述，请参阅《Java 虚拟机规范》）：

TypeSignature: **Z** | **C** | **B** | **S** | **I** | **F** | **J** | **D** | *FieldTypeSignature*
FieldTypeSignature: *ClassTypeSignature* | **[** *TypeSignature* | *TypeVar*
ClassTypeSignature: **L** *Id* (**/** *Id*)* *TypeArgs*? (**.** *Id TypeArgs*?)* **;**
TypeArgs: **<** *TypeArg*+ **>**
TypeArg: ***** | (**+** | **-**)? *FieldTypeSignature*
TypeVar: **T** *Id* **;**

第一条规则表明，**类型签名**或者是一个基元类型描述符，或者是一个字段类型签名。第二条规则将一个字段类型签名定义为一个类类型签名、数组类型签名或类型变量。第三条规则定义类类型签名：它们是类类型描述符，在主类名之后或者内部类名之后的尖括号中可能带有类型参数

（以点为前缀）。其他规则定义了类型参数和类型变量。注意，一个类型参数可能是一个完整的字段类型签名，带有它自己的类型参数：因此，类型签名可能非常复杂（见图 4.1）。

Java 类型和相应的类型签名
List<E> **Ljava/util/List<TE;>;**
List<?> **Ljava/util/List<*>;**
List<? extends Number> **Ljava/util/List<+Ljava/lang/Number;>;**
List<? super Integer> **Ljava/util/List<-Ljava/lang/Integer;>;**
List<List<String>[]> **Ljava/util/List<[Ljava/util/List<Ljava/lang/String;>;>;**
HashMap<K, V>.HashIterator<K> **Ljava/util/HashMap<TK;TV;>.HashIterator<TK;>;**

图 4.1 类型签名举例

方法签名扩展了方法描述符，就像类型签名扩展了类型描述符。**方法签名**描述了方法参数的类型签名及其返回类型的签名。与方法描述符不同的是，它还包含了该方法所抛出异常的签名，前面带有^前缀，还可以在尖括号之间包含可选的形式类型参数：

MethodTypeSignature:

TypeParams? **(** *TypeSignature** **)** (*TypeSignature* | **V**) *Exception**

Exception: ^ClassTypeSignature | *^TypeVar*

TypeParams: **<** *TypeParam*+ **>**

TypeParam: *Id* **:** *FieldTypeSignature*? (**:** *FieldTypeSignature*)*

比如以下泛型静态方法的方法签名，它以类型变量 **T** 为参数：

```
static <T> Class<? extends T> m (int n)
```

它是以下方法签名：

```
<T:Ljava/lang/Object;>(I)Ljava/lang/Class<+TT;>;
```

最后要说的是**类签名**，不要将它与类类型签名相混淆，它被定义为其超类的类型签名，后面跟有所实现接口的类型签名，以及可选的形式类型参数：

ClassSignature: *TypeParams*? *ClassTypeSignature ClassTypeSignature**

例如，一个被声明为 **C<E> extends List<E>** 的类的类签名就是 **<E:Ljava/lang/Object;>Ljava/util/List<TE;>;**。

4.1.2 接口与组件

和描述符的情况一样，也出于相同的效果原因（见 2.3.1 节），ASM API 公开签名的形式与它们在编译类中的存储形式相同（签名主要出现在 **ClassVisitor** 类的 **visit**、**visitField** 和 **visitMethod** 方法中，分别作为可选类、类型或方法签名参数）。幸好它还在

org.objectweb.asm.signature 包中提供了一些基于 **SignatureVisitor** 抽象类的工具，用于生成和转换签名（见图 4.2）。

```
public abstract class SignatureVisitor {
  public final static char EXTENDS = '+';
  public final static char SUPER = '-';
  public final static char INSTANCEOF = '=';
  public SignatureVisitor(int api);
  public void visitFormalTypeParameter(String name);
  public SignatureVisitor visitClassBound();
  public SignatureVisitor visitInterfaceBound();
  public SignatureVisitor visitSuperclass();
  public SignatureVisitor visitInterface();
  public SignatureVisitor visitParameterType();
  public SignatureVisitor visitReturnType();
  public SignatureVisitor visitExceptionType();
  public void visitBaseType(char descriptor);
  public void visitTypeVariable(String name);
  public SignatureVisitor visitArrayType();
  public void visitClassType(String name);
  public void visitInnerClassType(String name);
  public void visitTypeArgument();
  public SignatureVisitor visitTypeArgument(char wildcard);
  public void visitEnd();
}
```

图 4.2 **SignatureVisitor** 类

这个抽象类用于访问类型签名、方法签名和类签名。用于类型签名的方法以粗体显示，必须按以下顺序调用，它反映了前面的语法规则（注意，其中两个返回了 **SignatureVisitor**：这是因为类型签名的递归定义导致的）：

```
visitBaseType | visitArrayType | visitTypeVariable |
( visitClassType visitTypeArgument*
  ( visitInnerClassType visitTypeArgument* )* visitEnd ) )
```

用于访问方法签名的方法如下：

```
( visitFormalTypeParameter visitClassBound? visitInterfaceBound* )*
visitParameterType* visitReturnType visitExceptionType*
```

最后，用于访问类签名的方法为：

```
( visitFormalTypeParameter visitClassBound? visitInterfaceBound* )*
visitSuperClass visitInterface*
```

这些方法大多返回一个 **SignatureVisitor**：它是准备用来访问**类型**签名的。注意，不同于 **ClassVisitor** 返回的 **MethodVisitors**，**SignatureVisitor** 返回的 **SignatureVisitors** 不得为 **null**，而且必须顺序使用：事实上，在完全访问一个嵌套签名之前，不得访问父访问器的任何方法。

和类的情况一样，ASM API 基于这个 API 提供了两个组件：**SignatureReader** 组件分析一个签名，并针对一个给定的签名访问器调用适当的访问方法；**SignatureWriter** 组件基于它接收到的方法调用生成一个签名。

利用与类和方法相同的原理，这两个类可用于生成和转换签名。例如，假定我们希望对出现在某些签名中的类名进行重命名。这一效果可以用以下签名适配器完成，除 **visitClassType** 和 **visitInnerClassType** 方法之外，它将自己接收到的所有其他方法调用都不加修改地加以转发（这里假设 **sv** 方法总是返回 **this**，**SignatureWriter** 就属于这种情况）：

```
public class RenameSignatureAdapter extends SignatureVisitor {
  private SignatureVisitor sv;
  private Map<String, String> renaming;
  private String oldName;
  public RenameSignatureAdapter(SignatureVisitor sv,
      Map<String, String> renaming) {
    super(ASM4);
    this.sv = sv;
    this.renaming = renaming;
  }
  public void visitFormalTypeParameter(String name) {
    sv.visitFormalTypeParameter(name);
  }
  public SignatureVisitor visitClassBound() {
    sv.visitClassBound();
    return this;
  }
  public SignatureVisitor visitInterfaceBound() {
    sv.visitInterfaceBound();
    return this;
  }
  ...
  public void visitClassType(String name) {
    oldName = name;
    String newName = renaming.get(oldName);
    sv.visitClassType(newName == null ? name : newName);
  }
  public void visitInnerClassType(String name) {
    oldName = oldName + "." + name;
    String newName = renaming.get(oldName);
    sv.visitInnerClassType(newName == null ? name : newName);
  }
  public void visitTypeArgument() {
    sv.visitTypeArgument();
  }
  public SignatureVisitor visitTypeArgument(char wildcard) {
    sv.visitTypeArgument(wildcard);
    return this;
  }
  public void visitEnd() {
    sv.visitEnd();
  }
}
```

因此，以下代码的结果为**"LA<TK;TV;>.B<TK;>;"**：

```
String s = "Ljava/util/HashMap<TK;TV;>.HashIterator<TK;>;";
Map<String, String> renaming = new HashMap<String, String>();
renaming.put("java/util/HashMap", "A");
renaming.put("java/util/HashMap.HashIterator", "B");
SignatureWriter sw = new SignatureWriter();
SignatureVisitor sa = new RenameSignatureAdapter(sw, renaming);
SignatureReader sr = new SignatureReader(s);
sr.acceptType(sa);
sw.toString();
```

4.1.3 工具

2.3 节给出的**TraceClassVisitor**和**ASMifier**类以内部形式打印类文件中包含的签名。利用它们，可以通过以下方式找出与一个给定泛型相对应的签名：编写一个具有某一泛型的 Java 类，编译它，并用这些命令行工具来找出对应的签名。

4.2 注释

类、字段、方法和方法参数注释，比如**@Deprecated** 或**@Override**，只要它们的保留策略不是 **RetentionPolicy.SOURCE**，它们就会被存储在编译后的类中。这一信息不是在运行时供字节代码指令使用，但是，如果保留策略是 **RetentionPolicy.RUNTIME**，则可以通过反射 API 访问它。它还可以供编译器使用。

4.2.1 结构

源代码中的注释可以具有各种不同形式，比如 **@Deprecated** 、**@Retention(RetentionPolicy.CLASS)**或**@Task(desc="refactor", id=1)**。但在内部，所有注释的形式都是相同的，由一种注释类型和一组名称/值对规定，其中的取值仅限于如下几种：

- 基元，String 或 Class 值
- 枚举值
- 注释值
- 上述值的数组

注意，一个注释中可以包含其他注释，甚至可以包含注释数组。因此，注释可能非常复杂。

4.2.2 接口与组件

用于生成和转换注释的 ASM API 是基于 **AnnotationVisitor** 抽象类的（见图 4.3）。

```
public abstract class AnnotationVisitor {
  public AnnotationVisitor(int api);
  public AnnotationVisitor(int api, AnnotationVisitor av);
  public void visit(String name, Object value);
  public void visitEnum(String name, String desc, String value);
  public AnnotationVisitor visitAnnotation(String name, String desc);
  public AnnotationVisitor visitArray(String name);
  public void visitEnd();
}
```

图 4.3 **AnnotationVisitor** 类

这个类的方法用于访问一个注释的名称/值对（注释类型在访问这一类型的方法中访问，即

visitAnnotation 方法）。第一个方法用于基元、**String** 和 **Class** 值（后者用 **Type** 对象表示），其他方法用于枚举、注释和数组值。可以按任意顺序调用它们，**visitEnd** 除外：

```
(visit | visitEnum | visitAnnotation | visitArray )* visitEnd
```

注意，两个方法返回 **AnnotationVisitor**：这是因为注释可以包含其他注释。另外，与 **ClassVisitor** 返回的 **MethodVisitor** 不同，这两个方法返回的 **AnnotationVisitors** 必须顺序使用：事实上，在完全访问一个嵌套注释之前，不能调用父访问器的任何方法。

还要注意，**visitArray** 方法返回一个 **AnnotationVisitor**，以访问数组的元素。但是，由于数组的元素未被命名，因此，**name** 参数被 **visitArray** 返回的访问器的方法忽略，可以设定为 **null**。

1. 添加、删除和检测注释

与字段和方法的情景一样，可以通过在 **visitAnnotation** 方法中返回 **null** 来删除注释：

```
public class RemoveAnnotationAdapter extends ClassVisitor {
  private String annDesc;
  public RemoveAnnotationAdapter(ClassVisitor cv, String annDesc) {
    super(ASM4, cv);
    this.annDesc = annDesc;
  }
  @Override
  public AnnotationVisitor visitAnnotation(String desc, boolean vis) {
    if (desc.equals(annDesc)) {
      return null;
    }
    return cv.visitAnnotation(desc, vis);
  }
}
```

类注释的添加要更难一些，因为存在一些限制条件：必须调用 **ClassVisitor** 类的方法。事实上，所有可以跟在 **visitAnnotation** 之后的方法都必须重写，以检测什么时候已经访问了所有注释（因为 **visitCode** 方法的原因，方法注释的添加更容易一些）：

```
public class AddAnnotationAdapter extends ClassVisitor {
  private String annotationDesc;
  private boolean isAnnotationPresent;
  public AddAnnotationAdapter(ClassVisitor cv, String annotationDesc) {
    super(ASM4, cv);
    this.annotationDesc = annotationDesc;
  }
  @Override public void visit(int version, int access, String name,
    String signature, String superName, String[] interfaces) {
    cv.visit(v, access, name, signature, superName, interfaces);
  }
  @Override public AnnotationVisitor visitAnnotation(String desc,
      boolean visible) {
    if (visible && desc.equals(annotationDesc)) {
      isAnnotationPresent = true;
    }
    return cv.visitAnnotation(desc, visible);
  }
  @Override public void visitInnerClass(String name, String outerName,
      String innerName, int access) {
    addAnnotation();
    cv.visitInnerClass(name, outerName, innerName, access);
```

```
  }
  @Override
  public FieldVisitor visitField(int access, String name, String desc,
      String signature, Object value) {
    addAnnotation();
    return cv.visitField(access, name, desc, signature, value);
  }
  @Override
  public MethodVisitor visitMethod(int access, String name,
      String desc, String signature, String[] exceptions) {
    addAnnotation();
    return cv.visitMethod(access, name, desc, signature, exceptions);
  }
  @Override public void visitEnd() {
    addAnnotation();
    cv.visitEnd();
  }
  private void addAnnotation() {
    if (!isAnnotationPresent) {
      AnnotationVisitor av = cv.visitAnnotation(annotationDesc, true);
      if (av != null) {
        av.visitEnd();
      }
      isAnnotationPresent = true;
    }
  }
}
```

注意，如果类版本低于 1.5，这个适配器将其更新至该版本。这是必要地，因为对于版本低于 1.5 的类，JVM 会忽略其中的注释。

注释在类和方法适配器中的最后一种应用情景，也可能是最常见的应用情景，就是以注释实现转换的参数化。例如，你可能仅对于那些具有**@Persistent** 注释的字段来转换字段的访问，仅对于那些拥有**@Log** 注释的方法添加记录代码，如此等等。所有这些应用情景都可以很轻松地实现，因为注释是必须首先访问的：必须在字段和方法之前访问类注释，必须在代码之前访问方法和参数注释。因此，只需在检测到所需注释时设定一个标志，然后在后面的转换中使用，就像上面的例子用 **isAnnotationPresent** 标志所做的事情。

4.2.3 工具

2.3 节介绍的 **TraceClassVisitor**, **CheckClassAdapter** 和 **ASMifier** 类也支持注释（就像对于方法一样，还可能使用 **TraceAnnotationVisitor** 或 **CheckAnnotationAdapter**，在各个注释的级别工作，而不是在类级别工作）。它们可用于查看如何生成某个特定注释。例如，使用以下代码：

```
java -classpath asm.jar:asm-util.jar \1
   org.objectweb.asm.util.ASMifier \
   java.lang.Deprecated
```

将输出如下代码（经过微小的重构）：

```
package asm.java.lang;
import org.objectweb.asm.*;
public class DeprecatedDump implements Opcodes {
```

```
  public static byte[] dump() throws Exception {
    ClassWriter cw = new ClassWriter(0);
    AnnotationVisitor av;
    cw.visit(V1_5, ACC_PUBLIC + ACC_ANNOTATION + ACC_ABSTRACT
        + ACC_INTERFACE, "java/lang/Deprecated", null,
        "java/lang/Object",
        new String[] { "java/lang/annotation/Annotation" });
    {
      av = cw.visitAnnotation("Ljava/lang/annotation/Documented;",
         true);
      av.visitEnd();
    }
    {
      av = cw.visitAnnotation("Ljava/lang/annotation/Retention;", true);
      av.visitEnum("value", "Ljava/lang/annotation/RetentionPolicy;",
         "RUNTIME");
      av.visitEnd();
    }
    cw.visitEnd();
    return cw.toByteArray();
  }
}
```

此代码说明如何用 **ACC_ANNOTATION** 标志创建一个注释类，并说明如何创建两个类注释，一个没有值，一个具有枚举值。方法注释和参数注释可以采用 **MethodVisitor** 类中定义的 **visitAnnotation** 和 **visitParameterAnnotation** 方法以类似方式创建。

4.3 调试

以 **javac -g** 编译的类中包含了其源文件的名字、源代码行编号与字节代码指令之间的映射、源代码中局部变量名与字节代码中局部变量槽之间的映射。当这一可选信息可用时，会在调试器中和异常栈轨迹中使用它们。

4.3.1 结构

一个类的源文件名存储在一个专门的类文件结构部分中（见图 2.1）。

源代码行编号与字节代码指令之间的映射存储为一个由（*line number, label*）对组成的列表中，放在方法的已编译代码部分中。例如，如果 *l1*、*l2* 和 *l3* 是按此顺序出现的三个标记，则下面各对：

```
(n1, l1)
(n2, l2)
(n3, l3)
```

意味着 *l1* 和 *l2* 之间的指令来自行 **n1**，*l2* 和 *l3* 之间的指令来自 **n2**，*l3* 之后的指令来自行 **n3**。注意，一个给定行号可以出现在几个对中。这是因为，对于出现在一个源代码行中的表达式，其在字节代码中的相应指令可能不是连续的。例如，**for** (*init; cond; incr*) *statement;*通常是按以下顺序编译的：

init statement incr cond

源代码中局部变量名与字节代码中局部变量槽之间的映射，以*(name, type descriptor, type signature, start, end, index)*等多元组列表的形式存储在该方法的已编译代码节中。这样一个多元组的含义是：在两个标记 *start* 和 *end* 之间，槽 *index* 中的局部变量对应于源代码中的局部变量，其名字和类型由多元组的前三个元素组出。注意，编译器可以使用相同的局部变量槽来存储具有不同作用范围的不同源局部变量。反之，同一个源代码局部变量可能被编译为一个具有非连续作用范围的局部变量槽。例如，有可能存在一种类似如下的情景：

```
l1:
... // 这里的槽 1 包含局部变量 i
l2:
... // 这里的槽 1 包含局部变量 j
l3:
... // 这里的槽 1 再次包含局部变量 i
end:
```

相应的多元组为：

```
("i", "I", null, l1, l2, 1)
("j", "I", null, l2, l3, 1)
("i", "I", null, l3, end, 1)
```

4.3.2 接口和组件

调试信息用 **ClassVisitor** 和 **MethodVisitor** 类的三个方法访问：

- 源文件名用 **ClassVisitor** 类的 **visitSource** 方法访问；
- 源代码行号与字节代码指令之间的映射用 **MethodVisitor** 类的 **visitLineNumber** 方法访问，每次访问一对；
- 源代码中局部变量名与字节代码中局部变量槽之间的映射用 **MethodVisitor** 类的 **visitLocalVariable** 方法访问，每次访问一个多元组。

visitLineNumber 方法必须在已经访问了作为参数传送的标记之后进行调用。在实践中，就是在访问这一标记后立即调用它，从而可以非常容易地知道一个方法访问器中当前指令的源代码行：

```
public class MyAdapter extends MethodVisitor {
  int currentLine;
  public MyAdapter(MethodVisitor mv) {
    super(ASM4, mv);
  }
  @Override
  public void visitLineNumber(int line, Label start) {
    mv.visitLineNumber(line, start);
    currentLine = line;
  }
  ...
}
```

类似地，**visitLocalVariable** 方法方法必须在已经访问了作为参数传送的标记之后调用。下面给出一些方法调用示例，它们对应于上一节给出的名称值对和多元组：

```
visitLineNumber(n1, l1);
visitLineNumber(n2, l2);
visitLineNumber(n3, l3);
visitLocalVariable("i", "I", null, l1, l2, 1);
visitLocalVariable("j", "I", null, l2, l3, 1);
visitLocalVariable("i", "I", null, l3, end, 1);
```

1. 忽略调试信息

为了访问行号和局部变量名，**ClassReader** 类可能需要引入“人为”**Label** 对象，也就是说，跳转指令并不需要它们，它们只是为了表示调试信息。这可能会在诸如 3.2.5 节介绍的情景中导致错误判断，在该情景中，指令序列中部的一个 **Label** 被认为是一个跳转目标，因此禁止这一序列被删除。

为避免这种误判，可以在 **ClassReader.accept** 方法中使用 **SKIP_DEBUG** 选项。有了这一选项，类读取器不会访问调试信息，不会为它创建人为标记。当然，调试信息会从类中删除，因此，只有在不会为应用程序造成问题时才能使用这一选项。

> 注意：**ClassReader** 类提供了其他一些选项，比如：**SKIP_CODE**，用于跳过对已编译代码的访问（如果只需要类的结构，那这个选项是很有用的）；**SKIP_FRAMES**，用于跳过栈映射帧；**EXPAND_FRAMES**，用于解压缩这些帧。

4.3.3 工具

和泛型与注释的情景一样，可以使用 **TraceClassVisitor**、**CheckClassAdapter** 和 **ASMifier** 类来了解如何使用调试信息。

第5章 后向兼容

5.1 引言

过去已经在类文件格式中引入了新的元素，未来还将继续添加新元素（例如，用于模块化、Java 类型的注释，等等）。到 ASM 3.x，这样的每一次变化都会导致 ASM API 中的后向不兼容变化，这不是件好事情。为解决这些问题，ASM 4.0 中已经引入了一种新机制。它的目的是确保未来所有 ASM 版本都将与之前直到 ASM 4.0 的任意版本保持后向兼容，即使向类文件格式中引入了新的功能时也能保持这种兼容性。这意味着，从 4.0 开始，为一个 ASM 版本编写的类生成器、类分析器或类适配器，将可以在任何未来 ASM 版本中使用。**但是**，仅靠 ASM 自身是不能确保这一性质的。它需要用户在编写代码时遵循一些简单的准则。本章的目的就是介绍这些准则，并大致介绍一下 ASM 核心 API 中用于确保后向兼容性的内部机制。

> 注意：ASM 4.0 中引入的后向兼容机制要求将 **ClassVisitor**、**FieldVisitor**、**MethodVisitor** 等由接口变为抽象类，具有一个以 ASM 版本为参数的构造器。如果你的代码是为 ASM 3.x 实现的，可以将其升级至 ASM 4.0：将代码分析器和适配器中的 **implements** 用 **extends** 替换，并在它们的构造器中指定一个 ASM 版本。此外，**ClassAdapter** 和 **MethodAdapter** 还被合并到 **ClassVisitor** 和 **MethodVisitor** 中。要转换代码，只需用 **ClassVisitor** 代替 **ClassAdapter**，用 **MethodVisitor** 代替 **MethodAdapter**。另外，如果定义了自定义的 **FieldAdapter** 或 **AnnotationAdapter** 类，现在可以用 **FieldVisitor** 和 **AnnotationVisitor** 代替它们。

5.1.1 后向兼容约定

在给出用以确保后向兼容性的规则之前，首先给出“后向兼容”的更准确定义。

首先，研究一下新的类文件特征如何影响代码生成器、分析器和适配器是非常重要的。也就是说，在不受任何实现和二进制兼容问题影响时，在引入这些新特征之前设计的类生成器、分析器或适配器在进行这些修改之后还是否有效？换言之，如果有一个在引入这些新功能之前设计的转换链，假定这些新功功直接被忽略，原封不动地通过转换链，那这个转换链还是否依然有效？事实上，类生成器、分析器和适配器受到的影响是不同的：

- 类生成器不受影响：它们生成具有某一固定类版本的代码，这些生成的类在未来的 JVM 版本中依然有效，因为 JVM 确定了后向二进制兼容。
- 类分析器可能受到影响，也可能不受影响。例如，有一段用于分析字节代码指令的代码，

它是为 Java 4 编写的，它也许能够正常处理 Java 5 类，尽管 Java 5 中引入了注释。但同一段代码也许不再能处理 Java 7 类，因为它不能忽略新的动态调用指令。

- 类适配器可能受到影响，也可能不受影响。死代码清除工具不会因为引入注释而受到影响，甚至不会受到新的动态调用指令的影响。但另一方面，这两种新特性可能都会影响到为类进行重命名的工具。

这表明，新的类文件特性可能会对已有的类分析器或适配器产生不可预测的影响。如果新的特性直接被忽略，原封不动地通过一个分析链或转换链，这个链在某些情况下可以运行，不产生错误，并给出有效结果，而在某些情况下，也可以运行，不产生错误，但却给出无效结果，而在另外一些情况下，可能会在执行期间失败。第二种情景的问题尤其严重，因为它会在用户不知晓的情况下破坏分析链或转换链的语义，从而导致难以找出 Bug。为解决这一问题，我们认为最好不要忽略新特性，而是只要在分析链或转换链中遇到未知特性，就产生一条错误。这种错误发出信号：这个链也许能够处理新的类格式，也许不能，链的编写者必须分析具体情景，并在必要时进行更新。

所有上述内容引出了后向兼容性约定的如下定义：

- ASM 版本 X 是为版本号低于小等于 x 的 Java 类编写的。它不能生成版本号 y>x 的类，如果在 **`ClassReader.accept`** 中，以一个版本号大于 x 的类作为输入，它必须**失败**。
- 对于为 ASM X 编写且遵循了以下所述规则的代码，当输入类的版本不超过 x，对于 ASM 未来任意大于 X 的版本 Y，该代码都能不加修改地正常工作。
- 对于为 ASM X 编写且遵循了以下所述规则的代码，当输入类的声明版本为 y，但仅使用了在不晚于版本 x 中定义的功能，则在使用 ASM Y 或任意未来版本时，该代码能够不加修改地正常工作。
- 对于为 ASM X 编写且遵循了以下所述规则的代码，当输入类使用了在版本号为 y>x 的类中定义的功能时，对于 ASM X 或任意其他未来版本，该代码都必须**失败**。

注意，最后三点与类生成器无关，因为它没有类输入。

5.1.2 一个例子

为说明这些用户规则及用于保证后向兼容性的内部 ASM 机制，本章假定将向 Java 8 类中添加两个新的假设属性，一个用于存储类的作者，另一个用于存储它的许可。还假设这些新的属性在 ASM 5.0 中通过 **`ClassVisitor`** 的两个新方法公开，一个是：

```
void visitLicense(String license);
```

用于访问许可，还有一个是 **`visitSource`** 的新版本，用于在访问源文件名和调试信息的同时访问作者[①]：

```
@Deprecated void visitSource(String source, String debug);
```

[①] 事实上，可能仅添加一个**`visitLicense(String author, String license)`**方法，因为修改一个方法签名要比添加一个方法更复杂，如下所示。这里的做法只是出于说明目的。

作者和许可属性是可选的，即对 **visitLicense** 的调用并非强制的，在一个 **visitSource** 调用中，**author** 可能是 **null**。

5.2 规则

本节给出一些规则，在使用 ASM API 时，要想确保你的代码在所有未来 ASM 版本中都有效（其意义见上述约定），就必须遵循这些规则。

首先，如果编写一个类生成器，那不需要遵循任何规则。例如，如果正在为 ASM 4.0 编写一个类生成器，它可能包含一个类似于 **visitSource(mySource, myDebug)** 的调用，当然不包含对 **visitLicense** 的调用。如果不加修改地用 ASM 5.0 运行它，它将会调用过时的 **visitSource** 方法，但 ASM 5.0 **ClassWriter** 将会在内部将它重定向到 **visitSource(null, mySource, myDebug)**，生成所期望的结果（但其效率要稍低于直接将代码升级为调用这个新方法）。同理，缺少对 **visitLicense** 的调用也不会造成问题（所生成的类版本也没有变化，人们并不指望这个版本的类中会有一个许可属性）。

另一方面，如果编写一个类分析器或类适配器，也就是说，如果重写 **ClassVisitor** 类（或者任何其他类似的类，比如 **FieldVisitor** 或 **MethodVisitor**），就必须遵循一些规则，如下所述。

5.2.1 基本规则

这里考虑一个类的简单情况：直接扩展 **ClassVisitor**（讨论和规则对于其他访问器类都是相同的；间接子类的情景在下一节讨论）。在这种情况下，只有一条规则：

规则 1：要为 ASM X 编写一个 **ClassVisitor** 子类，就以这个版本号为参数，调用 **ClassVisitor** 构造器，在这个版本的 **ClassVisitor** 类中，**绝对不要重写或调用弃用的方法**（或者将在之后版本引入的方法）。

就这么多。在我们的示例情景中（见 5.1.2 节），为 ASM 4.0 编写的类适配器必须看起来类似于如下所示：

```
class MyClassAdapter extends ClassVisitor {
  public MyClassAdapter(ClassVisitor cv) {
    super(ASM4, cv);
  }
  ...
  public void visitSource(String source, String debug) { // optional
    ...
    super.visitSource(source, debug); // optional
  }
}
```

一旦针对 **ASM5**.0 升级之后，必须删除 **visitSource(String, String)**，这个类看起来必须类似于如下所示：

```
class MyClassAdapter extends ClassVisitor {
```

```
  public MyClassAdapter(ClassVisitor cv) {
    super(ASM5, cv);
  }
  ...
  public void visitSource(String author,
      String source, String debug) { // optional
    ...
    super.visitSource(author, source, debug); // optional
  }
  public void visitLicense(String license) { // optional
    ...
    super.visitLicense(license); // optional
  }
}
```

它是如何工作的呢？在 ASM 4.0 中，**ClassVisitor** 的内部实现如下：

```
public abstract class ClassVisitor {
  int api;
  ClassVisitor cv;
  public ClassVisitor(int api, ClassVisitor cv) {
    this.api = api;
    this.cv = cv;
  }
  ...
  public void visitSource(String source, String debug) {
    if (cv != null) cv.visitSource(source, debug);
  }
}
```

在 ASM 5.0 中，这一代码变为：

```
public abstract class ClassVisitor {
  ...
  public void visitSource(String source, String debug) {
    if (api < ASM5) {
      if (cv != null) cv.visitSource(source, debug);
    } else {
      visitSource(null, source, debug);
    }
  }
  public void visitSource(Sring author, String source, String debug) {
    if (api < ASM5) {
      if (author == null) {
        visitSource(source, debug);
      } else {
        throw new RuntimeException();
      }
    } else {
      if (cv != null) cv.visitSource(author, source, debug);
    }
  }
  public void visitLicense(String license) {
    if (api < ASM5) throw new RuntimeException();
    if (cv != null) cv.visitSource(source, debug);
  }
}
```

如果 **MyClassAdapter** 4.0 扩展了 **ClassVisitor** 4.0，那一切都将如预期中一样正常工作。如果升级到 ASM 5.0，但没有修改代码，**MyClassAdapter** 4.0 现在将扩展 **ClassVisitor** 5.0。但 **api** 字段仍将是 **ASM4** <**ASM5**，容易看出，在这种情况下，在调用 **visitSource(String,**

String)时，**ClassVisitor** 5.0 的行为特性类似于 **ClassVisitor** 4.0。此外，如果用一个 **null** 作者一访问新的 **visitSource** 方法，该调用将被重定向至旧版本。最后，如果在输入类中找到非 **null** 作者或许可，执行过程将会失败，与约定中的规定一致（或者是在新的 **visitSource** 方法中，或者是在 **visitLicense** 中）。

如果升级到 ASM 5.0，并同时升级代码，现在将拥有扩展了 **ClassVisitor** 5.0 的 **MyClassAdapter** 5.0。**api** 字段现在是 **ASM5**，**visitLicense** 和新的 **visitSource** 方法的行为就是直接将调用委托给下一个访问者 **cv**。此外，旧的 **visitSource** 方法现在将调用重定向至新的 **visitSource** 方法，这样可以确保：如果在转换链中，在我们自己的类适配器之前使用了一个旧类适配器，那 **MyClassAdapter** 5.0 不会错过这个访问事件。

ClassReader 将总是调用每个访问方法的最新版本。因此，如果随 ASM 4.0 使用 **MyClassAdapter** 4.0，或者随 ASM 5.0 使用 **MyClassAdapter 5.0**，将不会产生重定向。只有在随 ASM 5.0 使用 **MyClassAdapter 4.0** 时，才会在 **ClassVisitor** 中发生重定向（在新 **visitSource** 方法的第 3 行）。因此，尽管旧代码在新 ASM 版本中仍能正常使用，但它的运行速度要慢一些。将其升级为使用新的 API，将恢复其性能。

5.2.2 继承规则

上述规则对于 **ClassVisitor** 或任意其他类似类的直接子类都足够了。对于间接子类，也就是说，如果定义了一个扩展 **ClassVisitor** 的子类 **A1**，而它本身又由 **A2** 扩展，……它本身又由 **An** 扩展，则**必须为同一 ASM 版本编写所有这些子类**。事实上，在一个继承链中混用不同版本将导致同时重写同一方法的几个版本，比如 **visitSource(String,String)** 和 **visitSource(String,String,String)**，它们的行为可能不同，导致错误或不可预测的结果。如果这些类的来源不同，每个来源被独立升级、单独发布，那几乎不可能保证这一性质。这就引出第二条规则：

规则 2：不要使用访问器的继承，而要使用委托（即访问器链）。一种好的做法是让你的访问器类在默认情况为 final 的，以确保这一特性。

事实上，这一规则有两个例外：

- 如果能够完全由自己控制继承链，并同时发布层次结构中的所有类，那就可以使用访问器的继承。但必须确保层次结构中的所有类都是为同一 ASM 版本编写的。仍然要让层次结构的叶类是 final 的。
- 如果除了叶类之外，没有其他类重写任何访问方法（例如，如果只是为了引入方便的方法而在 ClassVisitor 和具体访问类之间使用了中间类），那就可以使用“访问器”的继承。仍然要让层次结构的叶类是 final 的（除非它们也没有重写任何访问方法；在这种情况下，提供一个以 ASM 版本为参数的构造器，使子类可以指定它们是为哪个版本编写的）。

第二部分 树 API

第6章 类

本章解释如何用 ASM 树 API 来生成和转换类。首先介绍树 API 本身，然后解释如何用核心 API 来组成它。用于方法、注释和泛型内容的树 API 将在随后各章介绍。

6.1 接口和组件

6.1.1 介绍

用于生成和转换已编译 **Java** 类的 ASM 树 API 是基于 **ClassNode** 类的（见图 6.1）。

```
public class ClassNode ... {
public int version;
public int access;
public String name;
public String signature;
public String superName;
public List<String> interfaces;
public String sourceFile;
public String sourceDebug;
public String outerClass;
public String outerMethod;
public String outerMethodDesc;
public List<AnnotationNode> visibleAnnotations;
public List<AnnotationNode> invisibleAnnotations;
public List<Attribute> attrs;
public List<InnerClassNode> innerClasses;
public List<FieldNode> fields;
public List<MethodNode> methods;
}
```

图 6.1 ClassNode 类（仅给出了字段）

可以看出，这个类的公共字段对应于图 2.1 中给出的类文件结构部分。这些字段的内容与核心 API 相同。例如，**name** 是一个内部名字，**signature** 是一个类签名（见 2.1.2 节和 4.1 节）。一些字段包含其他 ***Xxx*Node** 类：这些类将在随后各章详细介绍，它们拥有一种类似的结构，即拥有一些字段，对应于类文件结构的子部分。例如，**FieldNode** 类看起来是这样的：

```
public class FieldNode ... {
```

```
  public int access;
  public String name;
  public String desc;
  public String signature;
  public Object value;
  public FieldNode(int access, String name, String desc,
       String signature, Object value) {
    ...
  }
  ...
}
```

MethodNode 类是类似的：

```
public class MethodNode ... {
  public int access;
  public String name;
  public String desc;
  public String signature;
  public List<String> exceptions;
  ...
  public MethodNode(int access, String name, String desc,
  String signature, String[] exceptions)
  {
    ...
  }
}
```

6.1.2 生成类

用树 API 生成类的过程就是：创建一个 **ClassNode** 对象，并初始化它的字段。例如，2.2.3 节的 **Comarable** 接口可用如下代码生成（其代码数量大体与 2.2.3 节相同）：

```
ClassNode cn = new ClassNode();
cn.version = V1_5;
cn.access = ACC_PUBLIC + ACC_ABSTRACT + ACC_INTERFACE;
cn.name = "pkg/Comparable";
cn.superName = "java/lang/Object";
cn.interfaces.add("pkg/Mesurable");
cn.fields.add(new FieldNode(ACC_PUBLIC + ACC_FINAL + ACC_STATIC,
    "LESS", "I", null, new Integer(-1)));
cn.fields.add(new FieldNode(ACC_PUBLIC + ACC_FINAL + ACC_STATIC,
    "EQUAL", "I", null, new Integer(0)));
cn.fields.add(new FieldNode(ACC_PUBLIC + ACC_FINAL + ACC_STATIC,
    "GREATER", "I", null, new Integer(1)));
cn.methods.add(new MethodNode(ACC_PUBLIC + ACC_ABSTRACT,
    "compareTo", "(Ljava/lang/Object;)I", null, null));
```

使用树 API 生成类时，需要多花费大约 30%的时间（见附录 A.1），占用的内存也多于使用核心 API。但可以按任意顺序生成类元素，这在一些情况下可能非常方便。

6.1.3 添加和删除类成员

添加和删除类就是在 **ClassNode** 对象的 **fields** 或 **methods** 列表中添加或删除元素。例如，如果像下面这样定义了 **ClassTransformer** 类，以便能够轻松地编写类转换器：

```
public class ClassTransformer {
  protected ClassTransformer ct;
  public ClassTransformer(ClassTransformer ct) {
    this.ct = ct;
  }
  public void transform(ClassNode cn) {
    if (ct != null) {
      ct.transform(cn);
    }
  }
}
```

则 2.2.5 节中的 **RemoveMethodAdapter** 可实现如下：

```
public class RemoveMethodTransformer extends ClassTransformer {
  private String methodName;
  private String methodDesc;
  public RemoveMethodTransformer(ClassTransformer ct,
      String methodName, String methodDesc) {
    super(ct);
    this.methodName = methodName;
    this.methodDesc = methodDesc;
  }
  @Override public void transform(ClassNode cn) {
    Iterator<MethodNode> i = cn.methods.iterator();
    while (i.hasNext()) {
      MethodNode mn = i.next();
      if (methodName.equals(mn.name) && methodDesc.equals(mn.desc)) {
        i.remove();
      }
    }
    super.transform(cn);
  }
}
```

可以看出，它与核心 API 的主要区别是需要迭代所有方法，而在使用核心 API 时是不需要这样做的（这一工作会在 **ClassReader** 中为你完成）。事实上，这一区别对于几乎所有基于树的转换都是有效的。例如，在用树 API 实现 2.2.6 节的 **AddFieldAdapter** 时，它还需要一个迭代器：

```
public class AddFieldTransformer extends ClassTransformer {
  private int fieldAccess;
  private String fieldName;
  private String fieldDesc;
  public AddFieldTransformer(ClassTransformer ct, int fieldAccess,
      String fieldName, String fieldDesc) {
    super(ct);
    this.fieldAccess = fieldAccess;
    this.fieldName = fieldName;
    this.fieldDesc = fieldDesc;
  }
  @Override public void transform(ClassNode cn) {
    boolean isPresent = false;
    for (FieldNode fn : cn.fields) {
      if (fieldName.equals(fn.name)) {
        isPresent = true;
        break;
      }
    }
    if (!isPresent) {
      cn.fields.add(new FieldNode(fieldAccess, fieldName, fieldDesc,
```

```
            null, null));
        }
        super.transform(cn);
      }
    }
```

和生成类的情景一样，使用树 API 转换类时，所花费的时间和占用的内存也要多于使用核心 API 的时候。但使用树 API 有可能使一些转换的实现更为容易。比如有一个转换，要向一个类中添加注释，包含其内容的数字签名，就属于上述情景。在使用核心 API 时，只有在访问了整个类之后才能计算数字签名，但这时再添加包含其内容的注释就太晚了，因为对注释的访问必须位于类成员之前。而在使用树 API 时，这个问题就消失了，因为这时不存在此种限制。

事实上，有可能用核心 API 实现 **AddDigitialSignature** 示例，但随后，必须分两**遍**来转换这个类。第一遍，首先用一个 **ClassReader**（没有 **ClassWriter**）来访问这个类，以根据类的内容来计算数字签名。在第二遍，重复利用同一个 **ClassReader** 对类进行第一次访问，这一次是向一个 **ClassWriter** 链接一个 **AddAnnotationAdapter**。通过推广这一论述过程，我们可以看出，事实上，任何转换都可以仅用核心 API 来实现，只需在必要时分几遍完成。但这样就提高了转换代码的复杂性，要求在各遍之间存储状态（这种状态可能非常复杂，需要一个完整的树形表示！），而且对一个类进行多次分析是有成本的，必需将这一成本与构造相应 **ClassNode** 的成本进行比较。

结论是：**树 API 通常用于那些不能由核心 API 一次实现的转换。**但当然也存在例外。例如一个混淆器不能由核心 API 一遍实现，因为必须首先在原名称和混淆后的名字之间建立了完整的映射之后，才可能转换类，而这个映射的建立需要对所有类进行分析。但树 API 也不是一个好的解决方案，因为它需要将**所有**待混淆类的对象表示保存在内存中。在这种情况下，最好是分两遍使用核心 API：一遍用于计算原名与混淆后名称之间的映射（一个简单的散列表，它需要的内存要远少于所有类的完整对象表示），另一遍用于根据这一映射来转换类。

6.2 组件合成

到现在为止，我们只是看到了如何创建和转换 **ClassNode** 对象，但还没有看到如何由一个类的字节数组表示来构造一个 **ClassNode**，或者反过来，由 **ClassNode** 构造这个字节数组。事实上，这一功能可以通过合成核心 API 和树 API 组件来完成，本节就来解释这一内容。

6.2.1 介绍

除了图 6.1 所示的字段之外，**ClassNode** 类扩展了 **ClassVisitor** 类，还提供了一个 **accept** 方法，它以一个 **ClassVisitor** 为参数。**Accept** 方法基于 **ClassNode** 字段值生成事件，而 **ClassVisitor** 方法执行逆操作，即根据接到的事件设定 **ClassNode** 字段：

```
public class ClassNode extends ClassVisitor {
  ...
  public void visit(int version, int access, String name,
      String signature, String superName, String[] interfaces[]) {
    this.version = version;
```

```
    this.access = access;
    this.name = name;
    this.signature = signature;
    ...
  }
  ...
  public void accept(ClassVisitor cv) {
    cv.visit(version, access, name, signature, ...);
    ...
  }
}
```

要由字节数组构建 **ClassNode**，可以将它与 **ClassReader** 合在一起，使 **ClassReader** 生成的事件可供 **ClassNode** 组件使用，从而初始化其字段（由上述代码可以看出）：

```
ClassNode cn = new ClassNode();
ClassReader cr = new ClassReader(...);
cr.accept(cn, 0);
```

反过来，可以将 **ClassNode** 转换为其字节数组表示，只需将它与 **ClassWriter** 合在一起即可，从而使 **ClassNode** 的 **accept** 方法生成的事件可供 **ClassWriter** 使用：

```
ClassWriter cw = new ClassWriter(0);
cn.accept(cw);
byte[] b = cw.toByteArray();
```

6.2.2 模式

要用树 API 转换类，可以将这些元素放在一起：

```
ClassNode cn = new ClassNode(ASM4);
ClassReader cr = new ClassReader(...);
cr.accept(cn, 0);
... // 可以在这里根据需要转换 cn
ClassWriter cw = new ClassWriter(0);
cn.accept(cw);
byte[] b = cw.toByteArray();
```

还可能与核心 API 一起使用基于树的类转换器，比如类适配器。有两种常见模式可用于此种情景。第一种模式使用继承：

```
public class MyClassAdapter extends ClassNode {
  public MyClassAdapter(ClassVisitor cv) {
    super(ASM4);
    this.cv = cv;
  }
  @Override public void visitEnd() {
    // put your transformation code here
    accept(cv);
  }
}
```

当这个类适配器用在一个经典的转换链时：

```
ClassWriter cw = new ClassWriter(0);
ClassVisitor ca = new MyClassAdapter(cw);
ClassReader cr = new ClassReader(...);
cr.accept(ca, 0);
```

```
byte[] b = cw.toByteArray();
```

cr 生成的事件供 **ClassNode ca** 使用，从而初始化这个对象的字段。最后，在使用 **visitEnd** 事件时，**ca** 执行转换，并通过调用其 **accept** 方法，生成与所转换类对应的新事件，然后由 **cw** 使用。如果假定 **ca** 改变了类版本，则相应原程序图如图 6.2 所示。

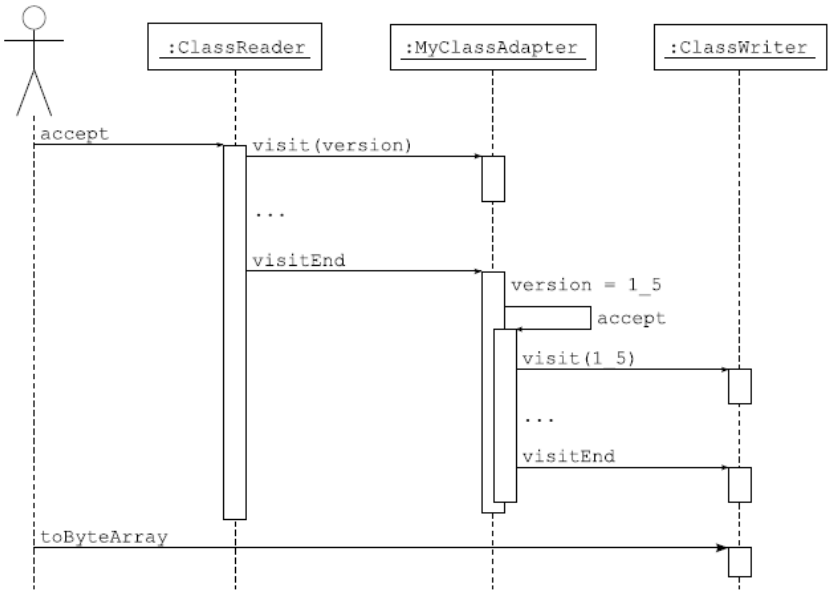


图 6.2 **MyClassAdapter** 的程序图

与图 2.7 中 **ChangeVersionAdapter** 的程序图进行对比，可以看出，**ca** 和 **cw** 之间的事件发生在 **cr** 和 **ca** 之间的事件之后，而不是像正常类适配器一样同时进行。事实上，对于所有基于树的转换都是如此，同时还解释了为什么它们受到的限制要少于基于事件的转换。

第二种模式可用于以类似程序图获得相同结果，它使用的是委托而非继承：

```
public class MyClassAdapter extends ClassVisitor {
  ClassVisitor next;
  public MyClassAdapter(ClassVisitor cv) {
    super(ASM4, new ClassNode());
    next = cv;
  }
  @Override public void visitEnd() {
    ClassNode cn = (ClassNode) cv;
    // 将转换代码放在这里
    cn.accept(next);
  }
}
```

这一模式使用两个对象而不是一个，但其工作方式完全与第一种模式相同：接收到的事件用于构造一个 **ClassNode**，它被转换，并在接收到最后一个事件后，变回一个基于事件的表示。

这两种模式都允许用基于事件的适配器来编写基于树的类适配器。它们也可用于将基于树的适配器组合在一起，但如果只需要组合基于树的适配器，那这并非最佳解决方案：在这种情况下，使用诸如 **ClassTransformer** 的类将会避免在两种表示之间进行不必要的转换。

第7章 方法

本章解释如何用 ASM 树 API 生成和转换方法。首先介绍树 API 本身，给出一些说明性示例，然后说明如何用核心 API 编写它。用于泛型和注释的树 API 在下一章介绍。

7.1 接口和组件

7.1.1 介绍

用于生成和转换方法的 ASM 树 API 是基于 **MethodNode** 类的（见图 7.1）。

```
public class MethodNode ... {
  public int access;
  public String name;
  public String desc;
  public String signature;
  public List<String> exceptions;
  public List<AnnotationNode> visibleAnnotations;
  public List<AnnotationNode> invisibleAnnotations;
  public List<Attribute> attrs;
  public Object annotationDefault;
  public List<AnnotationNode>[] visibleParameterAnnotations;
  public List<AnnotationNode>[] invisibleParameterAnnotations;
  public InsnList instructions;
  public List<TryCatchBlockNode> tryCatchBlocks;
  public List<LocalVariableNode> localVariables;
  public int maxStack;
  public int maxLocals;
}
```

图 7.1 **MethodNode** 类（仅给出字段）

这个类的大多数字段都类似于 **ClassNode** 的对应字段。最重要的是从 **instructions** 字段开始的最后几个。这个 **instructions** 字段是一个指令列表，用一个 **InsnList** 对象管理，它的公共 API 如下：

```
public class InsnList { // public accessors omitted
  int size();
  AbstractInsnNode getFirst();
  AbstractInsnNode getLast();
  AbstractInsnNode get(int index);
  boolean contains(AbstractInsnNode insn);
  int indexOf(AbstractInsnNode insn);
  void accept(MethodVisitor mv);
```

```
  ListIterator iterator();
  ListIterator iterator(int index);
  AbstractInsnNode[] toArray();
  void set(AbstractInsnNode location, AbstractInsnNode insn);
  void add(AbstractInsnNode insn);
  void add(InsnList insns);
  void insert(AbstractInsnNode insn);
  void insert(InsnList insns);
  void insert(AbstractInsnNode location, AbstractInsnNode insn);
  void insert(AbstractInsnNode location, InsnList insns);
  void insertBefore(AbstractInsnNode location, AbstractInsnNode insn);
  void insertBefore(AbstractInsnNode location, InsnList insns);
  void remove(AbstractInsnNode insn);
  void clear();
}
```

InsnList 是一个由指令组成的双向链表，**它们的链接存储在 AbstractInsnNode 对象本身中**。这一点极为重要，因为它对于必须如何使用指令对象和指令列表的方式有许多影响：

- 一个 **AbstractInsnNode** 对象在一个指令列表中最多出现一次。
- 一个 **AbstractInsnNode** 对象不能同时属于多个指令列表。
- 一个结果是：如果一个 **AbstractInsnNode** 属于某个列表，要将它添加到另一列表，必须先将其从原列表中删除。
- 另一结果是：将一个列表中的所有元素都添加到另一个列表中，将会清空第一个列表。

AbstractInsnNode 类是表示字节代码指令的类的超类。它的公共 API 如下：

```
public abstract class AbstractInsnNode {
  public int getOpcode();
  public int getType();
  public AbstractInsnNode getPrevious();
  public AbstractInsnNode getNext();
  public void accept(MethodVisitor cv);
  public AbstractInsnNode clone(Map labels);
}
```

它的子类是 ***Xxx*** **InsnNode** 类，对应于 **MethodVisitor** 接口的 **visit*Xxx*** **Insn** 方法，而且其构造方式完全相同。例如，**VarInsnNode** 类对应于 **visitVarInsn** 方法，且具有以下结构：

```
public class VarInsnNode extends AbstractInsnNode {
  public int var;
  public VarInsnNode(int opcode, int var) {
    super(opcode);
    this.var = var;
  }
  ...
}
```

标记与帧，还有行号，尽管它们并不是指令，但也都用 **AbstractInsnNode** 类的子类表示，即 **LabelNode**、**FrameNode** 和 **LineNumberNode** 类。这样就允许将它们恰好插在列表中对应的真实指令之前，与核心 API 中一样（在核心 API 中，就是恰在相应的指令之前访问标记和帧）。因此，很容易使用 **AbstractInsnNode** 类提供的 **getNext** 方法找到跳转指令的目标：这是目标标记之后第一个是真正指令的 **AbstractInsnNode**。另一个结果是：与核心 API 一样，只要标记保持不变，删除指令并不会破坏跳转指令。

7.1.2 生成方法

用树 API 生成一个方法包括：创建一个 **MethodNode**，初始化其字段。最重要的部分是方法代码的生成。比如，3.1.5 节的 **checkAndSetF** 方法可生成如下：

```
MethodNode mn = new MethodNode(...);
InsnList il = mn.instructions;
il.add(new VarInsnNode(ILOAD, 1));
LabelNode label = new LabelNode();
il.add(new JumpInsnNode(IFLT, label));
il.add(new VarInsnNode(ALOAD, 0));
il.add(new VarInsnNode(ILOAD, 1));
il.add(new FieldInsnNode(PUTFIELD, "pkg/Bean", "f", "I"));
LabelNode end = new LabelNode();
il.add(new JumpInsnNode(GOTO, end));
il.add(label);
il.add(new FrameNode(F_SAME, 0, null, 0, null));
il.add(new TypeInsnNode(NEW, "java/lang/IllegalArgumentException"));
il.add(new InsnNode(DUP));
il.add(new MethodInsnNode(INVOKESPECIAL,
   "java/lang/IllegalArgumentException", "<init>", "()V"));
il.add(new InsnNode(ATHROW));
il.add(end);
il.add(new FrameNode(F_SAME, 0, null, 0, null));
il.add(new InsnNode(RETURN));
mn.maxStack = 2;
mn.maxLocals = 2;
```

和类的情景一样，使用树 API 来生成方法时，花费的时间和占用的内存都要多于使用核心 API 的情况。但可以按照任意顺序来生成其内容。具体来说，这些指令可按非顺序方式生成，这在一些情况下是很有用的。

比如，考虑一个压缩编译器。通常，要编译表达式 e_1+e_2，首先发送 e_1 的代码，然后发出 e_2 的代码，然后发出将这两个值相加的代码。但如果 e_1 和 e_2 不是同一基元类型，必须恰在 e_1 的代码之后插入一个转换操作，恰在 e_2 的代码之后插入另一个。但是究竟发出哪些转换操作取决于 e_1 和 e_2 的类型。

现在，如果表达式的类型是由发出已编译代码的方法返回的，那在使用核心 API 时就会存在一个问题：只有在已经编译了 e_2 之后才能知道必须插在 e_1 之后的转换，但这时已经太晚了，因为我们不能在之前访问的指令之间插入指令。[①]在使用树 API 时不存在这一问题。例如，一种可能性是使用比如下面所示的 **compile** 方法：

```
public Type compile(InsnList output) {
  InsnList il1 = new InsnList();
  InsnList il2 = new InsnList();
  Type t1 = e1.compile(il1);
  Type t2 = e2.compile(il2);
  Type t = ...; // 计算 t1 和 t2 的公共超类型
  output.addAll(il1); // 在常量时间内完成
  output.add(...); // 由 t1 到 t 的转换指令
  output.addAll(il2); // 在常量时间内完成
  output.add(...); // 由 t2 到 t 的转换指令
```

[①] 解决方案是分两遍编译表达式：一遍用于计算表达式类型和必须插入的转换，另一遍发出编译后的代码。

```
    output.add(new InsnNode(t.getOpcode(IADD)));
    return t;
  }
```

7.1.3 转换方法

用树 API 转换方法只需要修改一个 **MethodNode** 对象的字段，特别是 **instructions** 列表。尽管这个列表可以采用任意方式修改，但常见做法是通过迭代修改。事实上，与通用 **ListIterator** 约定不同，**InsnList** 返回的 **ListIterator** 支持许多并发列表修改[①]。事实上，可以使用 **InsnList** 方法删除包括当前元素在内的一或多个元素，删除下一个元素之后的一或多个元素（也就是说，不是紧随当今元素之后的元素，而是它后面一个元素之后的元素），或者在当前元素之前或其后续者之后插入一或多个元素。这些修改将反映在迭代器中，即在下一元素之后插入（或删除）的元素将在迭代器中被看到（或不被看到）。

如果需要在一个列表的指令 *i* 之后插入几条指令，那另一种修改指令列表的常见做法是将这些新指令插入一个临时指令列表中，再在一个步骤内将这个临时列表插到主列表中：

```
InsnList il = new InsnList();
il.add(...);
...
il.add(...);
mn.instructions.insert(i, il);
```

逐条插入指令也是可行的，但却非常麻烦，因为必须在每次插之后更新插入点。

7.1.4 无状态转换和有状态转换

让我们用一些示例来具体看看如何用树 API 转换方法。为了看出核心 API 和树 API 之间的区别，重新实现 3.2.4 节的 **AddTimerAdapter** 示例和 3.2.5 节的 **RemoveGetFieldPutFieldAdapter** 是有意义的。计时器示例可实现如下：

```
public class AddTimerTransformer extends ClassTransformer {
  public AddTimerTransformer(ClassTransformer ct) {
    super(ct);
  }
  @Override public void transform(ClassNode cn) {
    for (MethodNode mn : (List<MethodNode>) cn.methods) {
      if ("<init>".equals(mn.name) || "<clinit>".equals(mn.name)) {
        continue;
      }
      InsnList insns = mn.instructions;
      if (insns.size() == 0) {
        continue;
      }
      Iterator<AbstractInsnNode> j = insns.iterator();
      while (j.hasNext()) {
        AbstractInsnNode in = j.next();
        int op = in.getOpcode();
        if ((op >= IRETURN && op <= RETURN) || op == ATHROW) {
```

[①] 即，这些修改与对 Iterator.next 的调用交织在一起。多线程并发是不受支持的。

```
          InsnList il = new InsnList();
          il.add(new FieldInsnNode(GETSTATIC, cn.name, "timer", "J"));
          il.add(new MethodInsnNode(INVOKESTATIC, "java/lang/System",
            "currentTimeMillis", "()J"));
          il.add(new InsnNode(LADD));
          il.add(new FieldInsnNode(PUTSTATIC, cn.name, "timer", "J"));
          insns.insert(in.getPrevious(), il);
        }
      }
      InsnList il = new InsnList();
      il.add(new FieldInsnNode(GETSTATIC, cn.name, "timer", "J"));
      il.add(new MethodInsnNode(INVOKESTATIC, "java/lang/System",
        "currentTimeMillis", "()J"));
      il.add(new InsnNode(LSUB));
      il.add(new FieldInsnNode(PUTSTATIC, cn.name, "timer", "J"));
      insns.insert(il);
      mn.maxStack += 4;
    }
    int acc = ACC_PUBLIC + ACC_STATIC;
    cn.fields.add(new FieldNode(acc, "timer", "J", null, null));
    super.transform(cn);
  }
}
```

在这里可以看出上一节讨论的用于在指令列表中插入若干指令的模式，其中包含了使用临时指令列表。这个示例还表明，有可能在迭代一个指令表的时候向当前指令之前插入指令。注意，在使用核心 API 和树 API 时，实现这一适配器所需要的代码数量大体相同。

（如果假定 **MethodTransformer** 类似于上一章的 **MethodTransformer** 类，）删除了字段自我赋值的方法适配器（见 3.2.5 节）可实现如下：

```
public class RemoveGetFieldPutFieldTransformer extends
    MethodTransformer {
  public RemoveGetFieldPutFieldTransformer(MethodTransformer mt) {
    super(mt);
  }
  @Override public void transform(MethodNode mn) {
    InsnList insns = mn.instructions;
    Iterator<AbstractInsnNode> i = insns.iterator();
    while (i.hasNext()) {
      AbstractInsnNode i1 = i.next();
      if (isALOAD0(i1)) {
        AbstractInsnNode i2 = getNext(i1);
        if (i2 != null && isALOAD0(i2)) {
          AbstractInsnNode i3 = getNext(i2);
          if (i3 != null && i3.getOpcode() == GETFIELD) {
            AbstractInsnNode i4 = getNext(i3);
            if (i4 != null && i4.getOpcode() == PUTFIELD) {
              if (sameField(i3, i4)) {
                while (i.next() != i4) {
                }
                insns.remove(i1);
                insns.remove(i2);
                insns.remove(i3);
                insns.remove(i4);
              }
            }
          }
        }
      }
    }
```

```
    super.transform(mn);
  }
  private static AbstractInsnNode getNext(AbstractInsnNode insn) {
    do {
      insn = insn.getNext();
      if (insn != null && !(insn instanceof LineNumberNode)) {
      break;
    }
  } while (insn != null);
  return insn;
  }
  private static boolean isALOAD0(AbstractInsnNode i) {
    return i.getOpcode() == ALOAD && ((VarInsnNode) i).var == 0;
  }
  private static boolean sameField(AbstractInsnNode i,
      AbstractInsnNode j) {
    return ((FieldInsnNode) i).name.equals(((FieldInsnNode) j).name);
  }
}
```

在这里再次看到，有可能在对一个指令清单迭代时从中删除指令。但要注意 **while (i.next() != i4)**循环：必须将迭代器放在必须删除的指令**之后**（因为不可能删除恰在当前指令之后的指令）。基于访问器和基于树的实现都可以在被检测序列的中部检测到标记和帧，在这种情况下，不要删除它。但要忽略序列中的行号（见 **getNext** 方法），使用基于树的 API 时的代码数量要多于使用核心 API 的情况。但是，这两种实现之间的主要区别是：在使用树 API 时，不需要状态机。特别是有三个或更多个连续 **ALOAD 0** 指令的特殊情景（它很容易被忽视），不再成为问题了。

利用上述实现，一条给定指令可能会被查看多次，这是因为在 **while** 循环中的每一步，**i2**、**i3** 和 **i4** 也可能会在这一迭代中被查看（在未来迭代中还会查看它们）。事实上，有可能使用一种更高效的实现，使每条指令最多被查看一次：

```
public class RemoveGetFieldPutFieldTransformer2 extends
    MethodTransformer {
  ...
  @Override public void transform(MethodNode mn) {
    InsnList insns = mn.instructions;
    Iterator i = insns.iterator();
    while (i.hasNext()) {
      AbstractInsnNode i1 = (AbstractInsnNode) i.next();
      if (isALOAD0(i1)) {
        AbstractInsnNode i2 = getNext(i);
        if (i2 != null && isALOAD0(i2)) {
          AbstractInsnNode i3 = getNext(i);
          while (i3 != null && isALOAD0(i3)) {
            i1 = i2;
            i2 = i3;
            i3 = getNext(i);
          }
          if (i3 != null && i3.getOpcode() == GETFIELD) {
            AbstractInsnNode i4 = getNext(i);
              if (i4 != null && i4.getOpcode() == PUTFIELD) {
                if (sameField(i3, i4)) {
                  insns.remove(i1);
                  insns.remove(i2);
                  insns.remove(i3);
                  insns.remove(i4);
                }
```

```
            }
          }
        }
      }
    }
    super.transform(mn);
  }
  private static AbstractInsnNode getNext(Iterator i) {
    while (i.hasNext()) {
      AbstractInsnNode in = (AbstractInsnNode) i.next();
      if (!(in instanceof LineNumberNode)) {
        return in;
      }
    }
    return null;
  }
  ...
}
```

与上一个实现的区别在于 **getNext** 方法，它现在是对列表迭代器进行操作。当序列被识别出来时，迭代器恰好位于它的后面，所以不再需要 **while (i.next() != i4)** 循环。但这里再次出现了三个或多个连续 **ALOAD 0** 指令的特殊情况（见 **while (i3 != null)** 循环）。

7.1.5 全局转换

到目前为止，我们看到的所有方法转换都是**局部**的，甚至有状态的转换也是如此，所谓“局部”是指，一条指令 *i* 的转换仅取决于与 *i* 有固定距离的指令。但还存在一些**全局**转换，在这种转换中，指令 *i* 的转换可能取决于与 *i* 有任意距离的指令。对于这些转换，树 API 真的很有帮助，也就是说，使用核心 API 实现它们将会非常非常复杂。

下面的转换就是这样一个例子：用向 **label** 的跳转代替向 GOTO **label** 指令的跳转，然后用一个 **RETURN** 指令代替指向这个 **RETURN** 指令的 **GOTO**。实际中，一个跳转指令的目标与这条指令的距离可能为任意远，可能在它的前面，也可能在其之后。这样一个转换可实现如下：

```
public class OptimizeJumpTransformer extends MethodTransformer {
  public OptimizeJumpTransformer(MethodTransformer mt) {
    super(mt);
  }
  @Override public void transform(MethodNode mn) {
    InsnList insns = mn.instructions;
    Iterator<AbstractInsnNode> i = insns.iterator();
    while (i.hasNext()) {
      AbstractInsnNode in = i.next();
      if (in instanceof JumpInsnNode) {
        LabelNode label = ((JumpInsnNode) in).label;
        AbstractInsnNode target;
        // 当 target == goto l, 用 l 代替 label
        while (true) {
          target = label;
          while (target != null && target.getOpcode() < 0) {
            target = target.getNext();
          }
          if (target != null && target.getOpcode() == GOTO) {
            label = ((JumpInsnNode) target).label;
          } else {
```

```
            break;
          }
        }
        // 更新目标
         ((JumpInsnNode) in).label = label;
        // 在可能时，用目标指令代替跳转
        if (in.getOpcode() == GOTO && target != null) {
          int op = target.getOpcode();
          if ((op >= IRETURN && op <= RETURN) || op == ATHROW) {
            // replace 'in' with clone of 'target'
            insns.set(in, target.clone(null));
          }
        }
      }
    }
    super.transform(mn);
  }
}
```

此代码的工作过程如下：当找到一条跳转指令 **in** 时，它的目标被存储在 **label** 中。然后用最内层的 **while** 循环查找紧跟在这个标记之后出现的指令（不代表实际指令的 **AbstractInsnNode** 对象，比如 **FrameNode** 或 **LabelNode**，其“操作码”为负）。只要这条指令是 **GOTO**，就用这条指令的目标代替 **label**，然后重复上述步骤。最后，用这个更新后的 **label** 值来代替 **in** 的目标标记，如果 **in** 本身是一个 **GOTO**，并且其更新后的目标是一条 **RETURN** 指令，则 **in** 用这个返回指令的**克隆副本**代替（回想一下，一个指令对象在一个指令列表中不能出现一次以上）。

对于 3.1.5 节定义的 **checkAndSetF** 方法，这一转换的效果如下：

```
// 之前                        // 之后
  ILOAD 1                      ILOAD 1
  IFLT label                   IFLT label
  ALOAD 0                      ALOAD 0
  ILOAD 1                      ILOAD 1
  PUTFIELD ...                 PUTFIELD ...
  GOTO end                     RETURN
label:                       label:
F_SAME                       F_SAME
  NEW ...                      NEW ...
  DUP                          DUP
  INVOKESPECIAL ...            INVOKESPECIAL ...
  ATHROW                       ATHROW
end:                         end:
F_SAME                       F_SAME
  RETURN                       RETURN
```

注意，尽管这个转换改变了跳转指令（更正式地说，是改变了控制流图），但它**不**需要更新方法的帧。事实上，在每条指令处，执行帧的状态保持不变，而且由于没有引用新的跳转目标，所以并不需要访问新的帧。但是，可能会出现不再需要某个帧的情况。例如在上面的例子中，转换后不再需要 **end** 标记，它后面的 **F_SAME** 帧和 **RETURN** 指令也是如此。幸好，访问帧数超出必需数量是完全合法的，在方法中包含未被使用的代码（称为**死**代码或**不可及**代码）也是合法的。因此，上述方法适配器是正确的，尽管可对其进行改进，删除死代码和帧。

7.2 组件合成

到目前为止，我们仅看到了如何创建和转换 **MethodNode** 对象，却还没有看到与类的字节数组表示进行链接。和类的情景一样，这一链接过程也是通过合成核心 API 和树 API 组件完成的，本节就来进行解释。

7.2.1 介绍

除了图 7.1 显示的字段之外，**MethodNode** 类扩展了 **MethodVisitor** 类，还提供了两个 **accept** 方法，它以一个 **MethodVisitor** 或一个 **ClassVisitor** 为参数。**accept** 方法基于 **MethodNode** 字段值生成事件，而 **MethodVisitor** 方法执行逆操作，即根据接收到的事件设定 **MethodNode** 字段。

7.2.2 模式

和类的情景一样，有可能与核心 API 使用一个基于树的方法转换器，比如一个方法适配器。用于类的两种模式实际上对于方法也是有效的，其工作方式完全相同。基于继承的模式如下：

```
public class MyMethodAdapter extends MethodNode {
  public MyMethodAdapter(int access, String name, String desc,
      String signature, String[] exceptions, MethodVisitor mv) {
    super(ASM4, access, name, desc, signature, exceptions);
    this.mv = mv;
  }
  @Override public void visitEnd() {
    // 将你的转换代码放在这儿
    accept(mv);
  }
}
```

而基于委托的模式为：

```
public class MyMethodAdapter extends MethodVisitor {
  MethodVisitor next;
  public MyMethodAdapter(int access, String name, String desc,
      String signature, String[] exceptions, MethodVisitor mv) {
    super(ASM4,
      new MethodNode(access, name, desc, signature, exceptions));
    next = mv;
  }
  @Override public void visitEnd() {
    MethodNode mn = (MethodNode) mv;
    //将你的转换代码放在这儿
    mn.accept(next);
  }
}
```

第一种模式的一种变体是直接在 **ClassAdapter** 的 **visitMethod** 中将它与一个匿名内部类一起使用：

```
public MethodVisitor visitMethod(int access, String name,
      String desc, String signature, String[] exceptions) {
  return new MethodNode(ASM4, access, name, desc, signature, exceptions)
  {
    @Override public void visitEnd() {
      //将你的转换代码放在这儿
      accept(cv);
    }
  };
}
```

这些模式表明，可以将树 API 仅用于方法，将核心 API 用于类。**在实践中经常使用这一策略。**

第8章 方法分析

本章介绍用于分析方法代码的 ASM API，它是基于树 API 的。首先介绍代码分析算法，然后以一些示例介绍相应的 ASM API。

8.1 介绍

代码分析是一个很大的主题，存在许多代码分析算法。我们不可能在这里介绍所有这些算法，也超出了本文档的范围。事实上，这一节的目的只是概述 ASM 中使用的算法。关于这一主题的更好介绍，可以在有关编译器的书中找到。接下来的几节将介绍代码分析技术的两个重要类型，即数据流和控制流分析：

- **数据流**分析包括：对于一个方法的每条指令，计算其执行帧的状态。这一状态可能采用一种多少有些抽象的方式来表示。例如，引用值可能用一个值来表示，可以每个类一个值，可以是{**null**, 非 **null**，可为 **null**}集合中的三个可能值表示，等等。
- **控制流**分析包括计算一个方法的控制流图，并对这个图进行分析。**控制流图**中的节点为指令，如果指令 *j* 可以紧跟在 *i* 之后执行，则图的有向边将连接这两条指令 *i→j*。

8.1.1 数据流分析

有两种类型的数据流分析可以执行：

- **正向分析**是指对于每条指令，根据执行帧在执行此指令之前的状态，计算执行帧在这一指令之后的状态。
- **反向分析**是指对于每条指令，根据执行帧在执行此指令之后的状态，计算执行帧在这一指令**之前**的状态。

正向数据流分析的执行是对于一个方法的每个字节代码指令，模拟它在其执行帧上的执行，通常包括：

- 从栈中弹出值，
- 合并它们，
- 将结果压入栈中。

这看起来似乎就是解释器或 Java 虚拟机做的事情，但事实上，它是完全不同的，因为其目

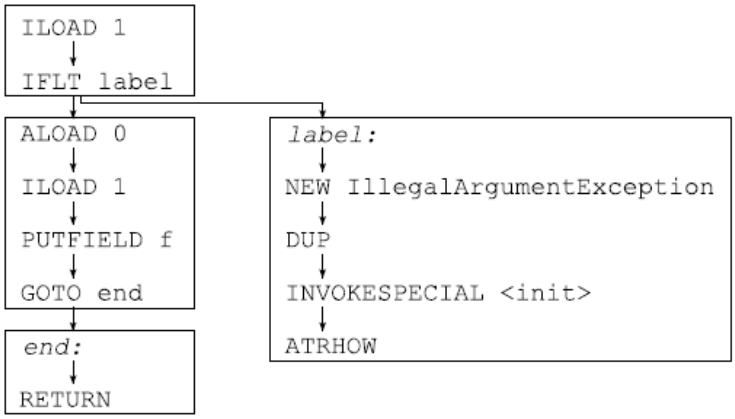
标是**对于所有可能出现的参数值**，模拟一个方法中的**所有可能执行路径**，而不是由某一组特定方法参数值所决定的单一执行路径。一个结果就是，对于分支指令，**两个**路径都将被模拟（而实际解释器将会根据实际条件值，仅沿一条分支执行）。

另一个结果是，所处理的值实际上是由可能取值组成的**集合**。这些集合可能非常大，比如“所有可能值”，“所有整数”，“所有可能对象”或者“所有可能的 **String** 对象”，在这些情况下，可以将它们称为**类型**。它们也可能更为准确，比如“所有正整数”，“所有介于 0 到 10 之间的整数”，或者“所有不为 **null** 的可能对象”。要模拟指令 *i* 的执行，就是要对于其操作数取值集合中的所有组合形式，找出 *i* 的所有可能结果集。例如，如果整数由以下三个集合表示：**P**=“正整数或 **null**”，**N**=“负整数或 **null**”，**A**=“所有整数”，要模拟 **IADD** 指令，就意味着当两个操作数均为 **P** 时返回 P，当两个操作数均为 N 时返回 N，在所有其他情况下返回 **A**。

最后一个后果是需要计算取值的**并**集：例如，与 **(b ? e1 : e2)** 对应的可能值集是 **e1** 的可能值与 **e2** 的可能值的并集。更一般地说，每当控制流图包含两条或多条具有同一目的地的边时，就需要这一操作。在上面的例子中，整数由三个集合 **P**、**N** 和 **A** 表示，可以很容易地计算出这些集合中两个集合的并集：除非这两个集合相等，否则总是 **A**。

8.1.2 控制流分析

控制流分析的基础是方法的控制流图。举个例子，3.1.3 节 **checkAndSetF** 方法的控制流图给出如下（图中包含的标记类似于实际指令）：



这个图可以分解为四个**基本**模块（如图中的矩形所示），一个基本模块就是这样一个指令序列：除最后一条指令外，每个指令都恰有一个后继者，而且除第一条外，所有其他指令都不是跳转的目标。

8.2 接口与组件

用于代码分析的 ASM API 在 **org.objectweb.asm.tree.analysis** 包中。由包的名字可以看出，它是基于树 API 的。事实上，这个包提供了一个进行正向数据流分析的框架。

为了能够以准确度不一的取值进行各种数据流分析，数据流分析算法分为两部分：一种是固定的，由框架提供，另一种是变化的，由用户提供。更准确地说：

- 整体数据流分析算法、将适当数量的值从栈中弹出和压回栈中的任务仅实现一次，用于 **Analyzer** 和 **Frame** 类中的所有内容。
- 合并值的任何和计算值集并集的任务由用户定义的 **Interpreter** 和 **Value** 抽象类的子类提供。提供了几个预定义的子类，下面几节将进行介绍。

尽管框架的主要目的是执行数据流分析，但 **Analyzer** 类也可构造所分析方法的控制流图。为此，可以重写这个类的 **newControlFlowEdge** 和 **newControlFlowExceptionEdge** 方法，它们默认情况下不做任何事情。其结果可用于进行控制流分析。

8.2.1 基本数据流分析

Interpreter 类是抽象类中预定义的 **Interpreter** 子类之一。它利用在 **BasicValue** 类中定义的七个值集来模拟字节代码指令的效果：

- **UNINITIALIZED_VALUE** 指“所有可能值”。
- **INT_VALUE** 指“所有 **int**、**short**、**byte**、**boolean** 或 **char** 值”。
- **FLOAT_VALUE** 指“所有 **float** 值”。
- **LONG_VALUE** 指“所有 **long** 值”。
- **DOUBLE_VALUE** 指“所有 **double** 值”。
- **REFERENCE_VALUE** 指“所有对象和数组值”。
- **RETURNADDRESS_VALUE** 用于子例程（见附录 A.2）

这个解释器本身不是非常有用（方法帧中已经提供了这一信息，而且更为详细——见 3.1.5 节），但它可以用作一个“空的”**Interpreter** 实现，以构建一个 **Analyzer**。这个分析器可用于检测方法中的不可及代码。事实上，即使是沿着跳转指令的两条分支，也不可能到达那些不能由第一条指令到达的代码。其结果是：在分析之后，无论什么样的 **Interpreter** 实现，由 **Analyzer.getFrames** 方法返回的计算帧，对于不可到达的指令都是 **null**。这一特性可用于非常轻松地实现一个 **RemoveDeadCodeAdapter** 类（还有一些更高效的方法，但它们需要编写的代码也更多）：

```
public class RemoveDeadCodeAdapter extends MethodVisitor {
  String owner;
  MethodVisitor next;
  public RemoveDeadCodeAdapter(String owner, int access, String name,
      String desc, MethodVisitor mv) {
    super(ASM4, new MethodNode(access, name, desc, null, null));
    this.owner = owner;
    next = mv;
  }
  @Override public void visitEnd() {
    MethodNode mn = (MethodNode) mv;
    Analyzer<BasicValue> a =
        new Analyzer<BasicValue>(new BasicInterpreter());
    try {
      a.analyze(owner, mn);
```

```
        Frame<BasicValue>[] frames = a.getFrames();
        AbstractInsnNode[] insns = mn.instructions.toArray();
        for (int i = 0; i < frames.length; ++i) {
          if (frames[i] == null && !(insns[i] instanceof LabelNode)) {
            mn.instructions.remove(insns[i]);
          }
        }
      } catch (AnalyzerException ignored) {
      }
      mn.accept(next);
    }
  }
```

结合 7.1.5 节的 **OptimizeJumpAdapter**，由跳转优化器引入的死代码被移除。例如，对 **checkAndSetF** 方法应用这个适配器链将给出：

```
// 在 OptimizeJump 之后
  ILOAD 1
  IFLT label
  ALOAD 0
  ILOAD 1
  PUTFIELD ...
  RETURN
label:
F_SAME
  NEW ...
  DUP
  INVOKESPECIAL ...
  ATHROW
end:
F_SAME
  RETURN
```

```
// 在 RemoveDeadCode 之后
  ILOAD 1
  IFLT label
  ALOAD 0
  ILOAD 1
  PUTFIELD ...
  RETURN
label:
F_SAME
  NEW ...
  DUP
  INVOKESPECIAL ...
  ATHROW
 end:
```

注意，死标记未被移除。这是故意的：它实际上没有改变最终代码，但避免删除一个尽管不可及但可能会在比如 **LocalVariableNode** 中引用的标记。

8.2.2 基本数据流验证器

BasicVerifier 类扩展 **BasicInterpreter** 类。它使用的事件集相同，但不同于 **BasicInterpreter** 的是，它会验证对指令的使用是否正确。例如，它会验证 **IADD** 指令的操作数为 **INTEGER_VALUE** 值（而 **BasicInterpreter** 只是返回结果，即 **INTEGER_VALUE**）。这个类可在开发类生成器或适配器时进行调试，见 3.3 节的解释。例如，这个类可以检测出 **ISTORE 1 ALOAD 1** 序列是无效的。它可以包含在像下面这样一个实用工具适配器中（在实践中，使用 **CheckMethodAdapter** 类要更简单一些，可以将其配置为使用 **BasicVerifier**）：

```
public class BasicVerifierAdapter extends MethodVisitor {
  String owner;
  MethodVisitor next;
  public BasicVerifierAdapter(String owner, int access, String name,
      String desc, MethodVisitor mv) {
    super(ASM4, new MethodNode(access, name, desc, null, null));
    this.owner = owner;
    next = mv;
  }
  @Override public void visitEnd() {
```

```
    MethodNode mn = (MethodNode) mv;
    Analyzer<BasicValue> a =
      new Analyzer<BasicValue(new BasicVerifier());
    try {
      a.analyze(owner, mn);
    } catch (AnalyzerException e) {
      throw new RuntimeException(e.getMessage());
    }
    mn.accept(next);
  }
}
```

8.2.3 简单的数据流验证器

SimpleVerifier 类扩展了 **BasicVerifier** 类。它使用更多的集合来模拟字节代码指令的执行：事实上，每个类都由它自己的集合表示，这个集合表示了这个类的所有可能对象。因此，它可以检测出更多的错误，比如如下情况：一个对象的可能值为“所有 **Thread** 类型的对象”，却对这个对象调用在 **String** 类中定义的方法。

这个类使用 Java 反射 API，以执行与类层次结构有关的验证和计算。然后，它将一个方法引用的类加载到 **JVM** 中。这一默认行为可以通过重写这个类的受保护方法来改变。

和 **BasicVerifier** 一样，这个类也可以在开发类生成器或适配器时使用，以便更轻松地找出 Bug。但它也可以用于其他目的。下面这个转换就是一个例子，它会删除方法中不必要的类型转换：如果这个分析器发现 **CHECKCAST to** 指令的操作数是“所有 **from** 类型的对象”值集，如果 **to** 是 **from** 的一个超类，那 **CHECKCAST** 指令就是不必要的，可以删除。这个转换的实现如下：

```
public class RemoveUnusedCastTransformer extends MethodTransformer {
  String owner;
  public RemoveUnusedCastTransformer(String owner,
      MethodTransformer mt) {
    super(mt);
    this.owner = owner;
  }
  @Override public MethodNode transform(MethodNode mn) {
    Analyzer<BasicValue> a =
      new Analyzer<BasicValue>(new SimpleVerifier());
    try {
      a.analyze(owner, mn);
      Frame<BasicValue>[] frames = a.getFrames();
      AbstractInsnNode[] insns = mn.instructions.toArray();
      for (int i = 0; i < insns.length; ++i) {
        AbstractInsnNode insn = insns[i];
        if (insn.getOpcode() == CHECKCAST) {
          Frame f = frames[i];
          if (f != null && f.getStackSize() > 0) {
            Object operand = f.getStack(f.getStackSize() - 1);
            Class<?> to = getClass(((TypeInsnNode) insn).desc);
            Class<?> from = getClass(((BasicValue) operand).getType());
            if (to.isAssignableFrom(from)) {
              mn.instructions.remove(insn);
            }
          }
        }
```

```
      }
    } catch (AnalyzerException ignored) {
    }
    return mt == null ? mn : mt.transform(mn);
  }
  private static Class<?> getClass(String desc) {
    try {
      return Class.forName(desc.replace('/', '.'));
    } catch (ClassNotFoundException e) {
      throw new RuntimeException(e.toString());
    }
  }
  private static Class<?> getClass(Type t) {
    if (t.getSort() == Type.OBJECT) {
      return getClass(t.getInternalName());
    }
    return getClass(t.getDescriptor());
  }
}
```

但对于 Java 6 类（或者用 **COMPUTE_FRAMES** 升级到 Java 6 的类），用 **AnalyzerAdapter** 以核心 API 来完成这一任务要更简单一些，效率要高得多：

```
public class RemoveUnusedCastAdapter extends MethodVisitor {
  public AnalyzerAdapter aa;
  public RemoveUnusedCastAdapter(MethodVisitor mv) {
    super(ASM4, mv);
  }
  @Override public void visitTypeInsn(int opcode, String desc) {
    if (opcode == CHECKCAST) {
      Class<?> to = getClass(desc);
      if (aa.stack != null && aa.stack.size() > 0) {
        Object operand = aa.stack.get(aa.stack.size() - 1);
        if (operand instanceof String) {
          Class<?> from = getClass((String) operand);
          if (to.isAssignableFrom(from)) {
            return;
          }
        }
      }
    }
    mv.visitTypeInsn(opcode, desc);
  }
  private static Class getClass(String desc) {
    try {
      return Class.forName(desc.replace('/', '.'));
    } catch (ClassNotFoundException e) {
      throw new RuntimeException(e.toString());
    }
  }
}
```

8.2.4 用户定义的数据流分析

假定我们希望检测出一些字段访问和方法调用的对象可能是 **null**，比如在下面的源代码段中（其中，第一行防止一些编译器检测 Bug，否则它可能会被认作一个“**o** 可能尚未初始化”错误）：

```
Object o = null;
while (...) {
o = ...;
}
o.m(...); // 潜在的 NullPointerException!
```

于是我们需要一个数据流分析，它能告诉我们，在对应于最后一行的 **INVOKEVIRTUAL** 指令处，与 **o** 对应的底部栈值可能为 **null**。为此，我们需要为引用值区分三个集合：包含 **null** 值的 **NULL** 集，包含所有非 **null** 引用值的 **NONNULL** 集，以及包含所有引用值的 **MAYBENULL** 集。于是，我们只需要考虑 **ACONST_NULL** 将 **NULL** 集压入操作数栈，而所有其他在栈中压入引用值的指令将压入 **NONNULL** 集（换句话说，我们考虑任意字段访问或方法调用的结果都不是 **null**，如果不对程序的所有类进行全局分析，那就不可能得到更好的结果）。为表示 **NULL** 和 **NONNULL** 集的并集，**MAYBENULL** 集合是必需的。

上述规则必须在一个自定义的 **Interpreter** 子类中实现。完全可以从头实现它，但也可以通过扩展 **BasicInterpreter** 类来实现它，而且这种做法要容易得多。事实上，如果我们考虑 **BasicValue.REFERENCE_VALUE** 对应于 **NONNULL** 集，那只需重写模拟 **ACONST_NULL** 执行的方法，使它返回 **NULL**，还有计算并集的方法：

```
class IsNullInterpreter extends BasicInterpreter {
  public final static BasicValue NULL = new BasicValue(null);
  public final static BasicValue MAYBENULL = new BasicValue(null);
  public IsNullInterpreter() {
    super(ASM4);
  }
  @Override public BasicValue newOperation(AbstractInsnNode insn) {
    if (insn.getOpcode() == ACONST_NULL) {
      return NULL;
    }
    return super.newOperation(insn);
  }
  @Override public BasicValue merge(BasicValue v, BasicValue w) {
    if (isRef(v) && isRef(w) && v != w) {
      return MAYBENULL;
    }
    return super.merge(v, w);
  }
  private boolean isRef(Value v) {
      return v == REFERENCE_VALUE || v == NULL || v == MAYBENULL;
  }
}
```

于是，可以很容易地利用这个 **IsNullnterpreter** 来检测那些可能导致潜在 **null** 指针异常的指令：

```
public class NullDereferenceAnalyzer {
  public List<AbstractInsnNode> findNullDereferences(String owner,
      MethodNode mn) throws AnalyzerException {
    List<AbstractInsnNode> result = new ArrayList<AbstractInsnNode>();
    Analyzer<BasicValue> a =
      new Analyzer<BasicValue>(new IsNullInterpreter());
    a.analyze(owner, mn);
    Frame<BasicValue>[] frames = a.getFrames();
    AbstractInsnNode[] insns = mn.instructions.toArray();
    for (int i = 0; i < insns.length; ++i) {
      AbstractInsnNode insn = insns[i];
      if (frames[i] != null) {
```

```
        Value v = getTarget(insn, frames[i]);
        if (v == NULL || v == MAYBENULL) {
          result.add(insn);
        }
      }
    }
    return result;
  }
  private static BasicValue getTarget(AbstractInsnNode insn,
       Frame<BasicValue> f) {
    switch (insn.getOpcode()) {
    case GETFIELD:
    case ARRAYLENGTH:
    case MONITORENTER:
    case MONITOREXIT:
      return getStackValue(f, 0);
    case PUTFIELD:
      return getStackValue(f, 1);
    case INVOKEVIRTUAL:
    case INVOKESPECIAL:
    case INVOKEINTERFACE:
      String desc = ((MethodInsnNode) insn).desc;
      return getStackValue(f, Type.getArgumentTypes(desc).length);
    }
    return null;
  }
  private static BasicValue getStackValue(Frame<BasicValue> f,
      int index) {
    int top = f.getStackSize() - 1;
    return index <= top ? f.getStack(top - index) : null;
  }
}
```

findNullDereferences 方法用一个 **IsNullInterpreter** 分析给定方法节点。然后，对于每条指令，检测其引用操作数（如果有的话）的可能值集是不是 **NULL** 集或 **NONNULL** 集。若是，则这条指令可能导致一个 **null** 指针异常，将它添加到此类指令的列表中，该列表由这一方法返回。

getTarget 方法在帧 **f** 中返回与 **insn** 对象操作数相对应的 **Value**，如果 insn 没有对象操作数，则返回 **null**。它的主要任务就是计算这个值相对于操作数栈顶端的偏移量，这一数量取决于指令类型。

8.2.5 控制流分析

控制流分析可以有许多应用。一个简单的例子就是计算方法的“圆复杂度”。这一度量定义为控制流图的边数减去节点数，再加上 2。例如，**checkAndSetF** 方法的控制流图如 8.1.2 节所示，它的圈复杂度为 11-12+2=1。这个度量很好地表征了一个方法的“复杂度”（在这个数字与方法的平均 bug 数之间存在一种关联）。它还给出了要“正确”测试一个方法所需要的建议测试情景数目。

用于计算这一度量的算法可以用 ASM 分析框架来实现（还有仅基于核心 API 的更高效方法，只是它们需要编写更多的代码）。第一步是构建控制流图。我们在本章开头曾经说过，可以通过重写 **Analyzer** 类的 **newControlFlowEdge** 方法来构建。这个类将节点表示为 **Frame** 对象。

如果希望将这个图存储在这些对象中，则需要扩展 **Frame** 类：

```
class Node<V extends Value> extends Frame<V> {
  Set< Node<V> > successors = new HashSet< Node<V> >();
  public Node(int nLocals, int nStack) {
    super(nLocals, nStack);
  }
  public Node(Frame<? extends V> src) {
    super(src);
  }
}
```

随后，可以提供一个 **Analyzer** 子类，用来构建控制流图，并用它的结果来计算边数、节点数，最终计算出圈复杂度：

```
public class CyclomaticComplexity {
  public int getCyclomaticComplexity(String owner, MethodNode mn)
      throws AnalyzerException {
    Analyzer<BasicValue> a =
        new Analyzer<BasicValue>(new BasicInterpreter()) {
      protected Frame<BasicValue> newFrame(int nLocals, int nStack) {
        return new Node<BasicValue>(nLocals, nStack);
      }
      protected Frame<BasicValue> newFrame(
          Frame<? extends BasicValue> src) {
        return new Node<BasicValue>(src);
      }
      protected void newControlFlowEdge(int src, int dst) {
        Node<BasicValue> s = (Node<BasicValue>) getFrames()[src];
        s.successors.add((Node<BasicValue>) getFrames()[dst]);
      }
    };
    a.analyze(owner, mn);
    Frame<BasicValue>[] frames = a.getFrames();
    int edges = 0;
    int nodes = 0;
    for (int i = 0; i < frames.length; ++i) {
      if (frames[i] != null) {
        edges += ((Node<BasicValue>) frames[i]).successors.size();
        nodes += 1;
      }
    }
    return edges - nodes + 2;
  }
}
```

第9章 元数据

本章介绍用于编译 Java 类元数据（比如注释）的树 API。本章非常短，因为这些元数据已经在第 4 章介绍过了，而且在了解了相应的核心 API 之后，树 API 就很简单了。

9.1 泛型

树 API 没有提供对泛型的任何支持！事实上，它用签名表示泛型，这一点与核心 API 中一样，但却没有提供与 **SignatureVisitor** 对应的 **SignatureNode** 类，尽管这也是可能的（事实上，至少使用几个 **Node** 类来区分类型、方法和类签名会很方便）。

9.2 注释

注释的树 API 都基于 **AnnotationNode** 类，它的公共 API 如下：

```
public class AnnotationNode extends AnnotationVisitor {
  public String desc;
  public List<Object> values;
  public AnnotationNode(String desc);
  public AnnotationNode(int api, String desc);
  ... // AnnotationVisitor 接口的方法
  public void accept(AnnotationVisitor av);
}
```

desc 字段包含了注释类型，而 **values** 字段包含了名称/值对，其中每个名字后面都跟有相关联的值（值的表示在 Javadoc 中描述）。

可以看出，**AnnotationNode** 类扩展了 **AnnotationVisitor** 类，还提供了一个 **accept** 方法，它以一个这种类型的对象为参数，比如具有这个类和方法访问器类的 **ClassNod** 和 **MethodNode** 类。我们前面已经看到用于类和方法的模式也可用于合成处理注释的核心与树 API 组件。例如，对于基于继承的模式（见 7.2.2 节），可进行“匿名内部类”的变体，使其适用于注释，给出如下：

```
public AnnotationVisitor visitAnnotation(String desc, boolean visible) {
  return new AnnotationNode(ASM4, desc) {
    @Override public void visitEnd() {
      // 将注释转换代码放在这里
      accept(cv.visitAnnotation(desc, visible));
    }
  };
```

```
}
```

9.3 调试

作为被编译类来源的源文件存储在 **ClassNode** 中的 **sourceFile** 字段中。关于源代码行号的信息存储在 **LineNumberNode** 对象中，它的类继承自 **AbstractInsnNode**。在核心 API 中，关于行号的信息是与指令同时受访问的，与此类似，**LineNumberNode** 对象是指令列表的一部分。最后，源局部变量的名字和类型存储在 **MethodNode** 的 **localVariables** 字段中，它是 **LocalVariableNode** 对象的一个列表。

第10章 后向兼容

10.1 介绍

与核心 **API** 的情景一样，在 **ASM 4.0** 的树 **API** 中已经引入了一种新机制，用于确保未来 ASM 版本的后向兼容性。但要再次强调，仅靠 **ASM** 自身不能保证这一性质。它要求用户在编写代码时遵循一些简单的规则。本章的目标就是介绍这些规则，并大致介绍 **ASM** 树 **API** 中用于确保后向兼容的内部机制。

10.2 规则

本节给出一些规则，在使用 **ASM** 树 **API** 时，要想确保你的代码在所有未来 **ASM** 版本中都保持有效（其意义见 5.1.1 节定义的约定），就必须遵循这些规则。

首先，如果使用树 **API** 编写一个类生成器，那就不需要遵循什么规则（和核心 API 一样）。可以用任意构造器创建 **ClassNode** 和其他元素，可以使用这些类的任意方法。

另一方面，如果要用树 API 编写类分析器或类适配器，也就是说，如果使用 **ClassNode** 或其他直接或间接地通过 **ClassReader.accept()** 填充的类似类，或者如果重写这些类中的一个，则必须遵循下面给出的规则。

10.2.1 基本规则

1. 创建类节点

考虑这样一种情景，我们创建一个 **ClassNode**，通过一个 **ClassReader** 填充它，然后分析或转换它，最终根据需要用 **ClassWriter** 写出结果（这一讨论及相关规则同样适用于其他节点类；对于由别人创建的 **ClassNode**，其分析或转换在下一节讨论）。在这种情况下，仅有一条规则：

规则 3：要用 **ASM** 版本 X 的树 **API** 编写类分析器或适配器，则使用以这一确切版本为参数的构造器创建 **ClassNode**（而不是使用没有参数的默认构造器）。

本规则的目的是在通过一个 **ClassReader** 填充 **ClassNode** 时，如果遇到未知特性，则抛出一个错误（根据后向兼容性约定的定义）。如果不遵循这一规则，在以后遇到未知元素时，你

的分析或转换代码可能会失败，也许能够成功运行，但却因为没有忽略这些未知元素而给出错误结果。换言之，如果不遵循这一规则，可能无法保证约定的最后一项条款。

如何做到呢？**ASM 4.0** 内部对 **ClassNode** 的实现如下（这里重复使用 5.1.2 节的示例）：

```
public class ClassNode extends ClassVisitor {
  public ClassNode() {
    super(ASM4, null);
  }
  public ClassNode(int api) {
    super(api, null);
  }
  ...
  public void visitSource(String source, String debug) {
    // 将 source 和 debug 存储在局部字段中...
  }
}
```

在 ASM 5.0 中，这一代码变为：

```
public class ClassNode extends ClassVisitor {
  ...
  public void visitSource(String source, String debug) {
    if (api < ASM5) {
      // 将 source 和 debug 存储在局部字段中...
    } else {
      visitSource(null, source, debug);
    }
  }
  public void visitSource(Sring author, String source, String debug) {
    if (api < ASM5) {
      if (author == null)
        visitSource(source, debug);
      else
        throw new RuntimeException();
    } else {
      // 将 author、source 和 debug 存储在局部字段中...
    }
  }
  public void visitLicense(String license) {
    if (api < ASM5) throw new RuntimeException();
    // 将 license 存储在局部字段中
  }
}
```

如果使用 **ASM 4.0**，那创建 **ClassNode(ASM4)** 没有什么特别之处。但如果升级到 **ASM 5.0**，但不修改代码，那就会得到一个 **ClassNode** 5.0，它的 **api** 字段将为 **ASM4** < **ASM5**。于是容易看出，如果输入类包含一个非 **null** 作者或许可属性，那通过 **ClassReader** 填充 **ClassNode** 时将会失败，如约定中的定义。如果还升级你的代码，将 **api** 字段改为 **ASM5**，并升级剩余代码，将这些新属性考虑在内，那在填充代码时就不会抛出错误。

注意，**ClassNode 5.0** 代码非常类似于 **ClassVisitor 5.0** 代码。这是为了确保在定义 **ClassNode** 的子类时能够拥有正确的语义（类似于 **ClassVisitor** 的子类——见 10.2.2 节）。

2. 使用现有类代码

如果你的类分析器或适配器收到别人创建的 **ClassNode**，那你就不能肯定在创建它时传送

给其构造器的 ASM 版本。当然可以自行检查 **api** 字段，但如果发现这个版本高于你支持的版本，直接拒绝这个类可能太过保守了。事实上，这个类中可能没有包含任何未知特性。另一方面，你不能检查是否存在未知特性（在我们的示例情景中，在为 ASM 4.0 编写代码时，你如何判断你的 **ClassNode** 中不存在未知的 **license** 字段呢？因为你在这里还不知道未来会添加这样一个字段）。于是设计了 **ClassNode.check()** 方法来解决这个问题。这就引出了以下规则：

规则 4：要用 ASM 版本 X 的树 API 编写一个类分析器或适配器，使用别人创建的 **ClassNode**，在以任何方式使用这个 **ClassNode** 之前，都要以这个确切版本号为参数，调用它的 **check()** 方法。

其目的与规则 3 相同：如果不遵循这一规则，可能无法保证约定的最后一项条款。如何做到的呢？这个检查方法在 **ASM 4.0** 内部的实现如下：

```
public class ClassNode extends ClassVisitor {
  ...
  public void check(int api) {
    // 不做任何事
  }
}
```

在 ASM 5.0 中，这一代码变为：

```
public class ClassNode extends ClassVisitor {
  ...
  public void check(int api) {
    if (api < ASM5 && (author != null || license != null)) {
      throw new RuntimeException();
    }
  }
}
```

如果你的代码是为 **ASM 4.0** 编写的，而且如果得到一个 **ClassNode** 4.0，它的 **api** 字段将为 **ASM4**，这样不会有问题，**check** 也不做任何事情。但如果你得到一个 **ClassNode** 5.0，如果这个节点实际上包含了非 **null author** 或 **license**，也就是说，它包含了 ASM 4.0 中未知的新特性，那 **check(ASM4)** 方法将会失败。

> 注意：如果你自己创建 **ClassNode**，也可以使用这一规则。那就不需要遵循规则 3，也就是说，不需要在 **ClassNode** 构造器中指明 ASM 版本。这一检查将在 **check** 方法中进行（但在填充 **ClassNode** 时，这种做法的效率要低于在之前进行检查）。

10.2.2 继承规则

如果希望提供 **ClassNode** 的子类或者其他类似节点类，那么规则 1 和 2 都是适用的。注意，在一个 **MethodNode** 匿名子类的一个常用特例中，**visitEnd()** 方法被重写：

```
class MyClassVisitor extends ClassVisitor {
  ...
  public MethodVisitor visitMethod(...) {
    final MethodVisitor mv = super.visitMethod(...);
```

```
      if (mv != null) {
        return new MethodNode(ASM4) {
          public void visitEnd() {
            // perform a transformation
            accept(mv);
          }
        }
      }
      return mv;
    }
  }
```

那就自动适用规则 2（匿名类不能被重写，尽管没有明确将它声明为 **final** 的）。你只需要遵循规则 3，也就是说，在 **MehtodNode** 构造器中指定 ASM 版本（或者遵循规则 4，也就是在执行转换之前调用 **check(ASM4)**）。

10.2.3 其他包

asm.util 和 **asm.commons** 中的类都有两个构造函数变体：一个有 ASM 版本参数，一个没有。

如果只是希望像 **asm.util** 中的 **ASMifier**、**Textifier** 或 **Check*Xxx* Adapter** 类或者 **asm.commons** 包中的任意类一样，加以实例化和应用，那可以用没有 ASM 版本参数的构造器来实例化它们。也可以使用带有 ASM 版本参数的构造器，那就会不必要地将这些组件限制于特定的 ASM 版本（而使用无参数构造器相当于在说“使用最新的 ASM 版本”）。这就是为什么使用 ASM 版本参数的构造器被声明为 **protected**。

另一方面，如果希望重写 **asm.util** 中的 **ASMifier**、**Textifier** 或 **CheckXxx Adapter** 类或者 **asm.commons** 包中的任意类，那适用规则 1 和 2。具体来说，你的构造器**必须**以你希望用作参数的 ASM 版本来调用 **super(…)**。

最后，如果希望使用或重写 **asm.tree.analysis** 中的 **Interpreter** 类或其子类，必须做出同样的区分。还要注意，在使用这个分析包之前，创建一个 **MethodNode** 或者从别人那里获取一个，那在将这一代码传送给 **Analyzer** 之前必须使用规则 3 和 4。

A. 附录

A.1 字节代码指令

本节对字节代码指令进行简要描述。如需全面描述，请参阅 Java 虚拟机规范。

约定：a 和 **b** 表示 **int**, **float**, **long** 或 **double** 值（比如，它们对于 **IADD** 表示 **int**，而对于 **LADD** 则表示 **long**），**o** 和 **p** 表示对象引用，**v** 表示任意值（或者，对于栈指令，表示大小为 1 的值），**w** 表示 **long** 或 **double**，**i**、**j** 和 **n** 表示 **int** 值。

局部变量

指令	之前的栈	之后的栈
ILOAD, LLOAD, FLOAD, DLOAD var	...	... , a
ALOAD var	...	... , o
ISTORE, LSTORE, FSTORE, DSTORE var	... , a	...
ASTORE var	... , o	...
IINC var incr	...	...

栈

POP	... , v	...
POP2	... , v1 , v2	...
	... , w	...
DUP	... , v	... , v , v
DUP2	... , v1 , v2	... , v1 , v2 , v1 , v2
	... , w	... , w, w
SWAP	... , v1 , v2	... , v2 , v1
DUP_X1	... , v1 , v2	... , v2 , v1 , v2
DUP_X2	... , v1 , v2 , v3	... , v3 , v1 , v2 , v3
	... , w , v	... , v , w , v
DUP2_X1	... , v1 , v2 , v3	... , v2 , v3 , v1 , v2 , v3
	... , v , w	... , w , v , w
DUP2_X2	... , v1 , v2 , v3 , v4	... , v3 , v4 , v1 , v2 , v3 , v4
	... , w , v1 , v2	... , v1 , v2 , w , v1 , v2

	.... , v1 , v2 , w	... , w , v1 , v2 , w
	... , w1 , w2	... , w2 , w1 , w2

常量

ICONST_n (-1 _ n _ 5)	...	... , n
LCONST_n (0 _ n _ 1)	...	... , nL
FCONST_n (0 _ n _ 2)	...	... , nF
DCONST_n (0 _ n _ 1)	...	... , nD
BIPUSH b, -128 _ b < 127	...	... , b
SIPUSH s, -32768 _ s < 32767	...	... , s
LDC cst (int, float, long, double, String 或 Type)	...	... , cst
ACONST_NULL	...	... , null

算术与逻辑

IADD, LADD, FADD, DADD	... , a , b	... , a + b
ISUB, LSUB, FSUB, DSUB	... , a , b	... , a - b
IMUL, LMUL, FMUL, DMUL	... , a , b	... , a * b
IDIV, LDIV, FDIV, DDIV	... , a , b	... , a / b
IREM, LREM, FREM, DREM	... , a , b	... , a % b
INEG, LNEG, FNEG, DNEG	... , a	... , -a
ISHL, LSHL	... , a , n	... , a <_< n
ISHR, LSHR	... , a , n	... , a >_> n
IUSHR, LUSHR	... , a , n	... , a >_>_> n
IAND, LAND	... , a , b	... , a & b
IOR, LOR	... , a , b	... , a | b
IXOR, LXOR	... , a , b	... , a ^ b
LCMP	... , a , b	... , a == b ? 0 : (a < b ? -1 : 1)
FCMPL, FCMPG	... , a , b	... , a == b ? 0 : (a < b ? -1 : 1)
DCMPL, DCMPG	... , a , b	... , a == b ? 0 : (a < b ? -1 : 1)

类型转换

I2B	... , i	... , (byte) i
I2C	... , i	... , (char) i
I2S	... , i	... , (short) i
L2I, F2I, D2I	... , a	... , (int) a
I2L, F2L, D2L	... , a	... , (long) a
I2F, L2F, D2F	... , a	... , (float) a
I2D, L2D, F2D	... , a	... , (double) a
CHECKCAST *class*	... , o	... , (*class*) o

对象、字段和方法

NEW class ...	... , new class	

GETFIELD c f t	... , o	... , o.f
PUTFIELD c f t	... , o , v	...
GETSTATIC c f t	...	... , c.f
PUTSTATIC c f t	... , v	...
INVOKEVIRTUAL c m t	... , o , v1 , ... , vn	... , o.m(v1, ... vn)
INVOKESPECIAL c m t	... , o , v1 , ... , vn	... , o.m(v1, ... vn)
INVOKESTATIC c m t	... , v1 , ... , vn	... , c.m(v1, ... vn)
INVOKEINTERFACE c m t	... , o , v1 , ... , vn	... , o.m(v1, ... vn)
INVOKEDYNAMIC m t bsm	... , o , v1 , ... , vn	... , o.m(v1, ... vn)
INSTANCEOF class	... , o	... , o instanceof class
MONITORENTER	... , o	...
MONITOREXIT	... , o	...

数组

NEWARRAY type （用于任意基元类型）	... , n	... , new type[n]
ANEWARRAY class	... , n	... , new class[n]
MULTIANEWARRAY [...[t n	... , i1 ,... , in	... , new t[i1]...[in]...
BALOAD, CALOAD, SALOAD	... , o , i	... , o[i]
IALOAD, LALOAD, FALOAD, DALOAD	... , o , i	... , o[i]
AALOAD	... , o , i	... , o[i]
BASTORE, CASTORE, SASTORE	... , o , i , j	...
IASTORE, LASTORE, FASTORE, DASTORE	... , o , i , a	...
AASTORE	... , o , i , p	...
ARRAYLENGTH	... , o	... , o.length

跳转

IFEQ	... , i	... i == 0 时跳转
IFNE	... , i	... i != 0 时跳转
IFLT	... , i	... i < 0 时跳转
IFGE	... , i	... i >= 0 时跳转
IFGT	... , i	... i > 0 时跳转
IFLE	... , i	... i <= 0 时跳转
IF_ICMPEQ	... , i , j	... i == j 时跳转
IF_ICMPNE	... , i , j	... i != j 时跳转
IF_ICMPLT	... , i , j	... i < j 时跳转
IF_ICMPGE	... , i , j	... i >= j 时跳转
IF_ICMPGT	... , i , j	... i > j 时跳转
IF_ICMPLE	... , i , j	... i <= j 时跳转

IF_ACMPEQ	... , o , p	... o == p 时跳转
IF_ACMPNE	... , o , p	... o != p 时跳转
IFNULL	... , o	... o == null 时跳转
IFNONNULL	... , o	... o != null 时跳转
GOTO	...	... 总是跳转
TABLESWITCH	... , i	... 总是跳转
LOOKUPSWITCH	... , i	... 总是跳转

返回

IRETURN, LRETURN, FRETURN, DRETURN	... , a
ARETURN	... , o
RETURN	...
ATHROW	... , o

A.2 子例程

除了上一节给出的字节代码指令，版本号**低于**或等于 **V1_5** 的类还可以包含 **JSR** 和 **RET** 指令，用于子例程（**JSR** 表示 Jump to SubRoutine，即跳转至子例程，RET 表示 RETurn from subroutine,，即从子例程返回）。版本高于或等于 **V1_6** 的类**不能**包含这些指令（移除它们就是为了 Java 6 中引入的新验证器体系结构；因为它们不是严格必需的，所以才可能删除它们）。

JSR 指令以一个标记为参数，无条件跳转到这个标记。但在跳转之前，它会在操作数栈中压入一个**返回地址**，它是紧跟在 **JSR** 之后的指令的索引。这个返回地址只能由诸如 **POP**、**DUP** 或 **SWAP** 之类的栈指令、**ASTORE** 指令和 **RET** 指令处理。

RET 指令以一个局部变量索引为参数。它加载包含在这个槽中的返回地址，并无条件跳转至相应的指令。由于返回地址可以有几个可能值，所以 RET 指令可以返回到几个可能指令。

让我们用一个例子来说明。考虑以下代码：

```
  JSR sub
  JSR sub
  RETURN
sub:
  ASTORE 1
  IINC 0 1
  RET 1
```

第一条指令将第二条指令的索引作为返回地址压入栈中，并跳转到 **ASTORE** 指令。这个指令将返回地址存储局部变量 **1** 中。然后，局部变量 0 增 1。最后，**RET** 指令载入在局部变量 **1** 中包含的返回地址，并跳转到相应的指令，即第二条指令。

这个第二指令又是一个 **JSR** 指令：它将第三条指令的索引作为返回地址压入栈中，并跳转到 **ASTORE** 指令。当再次到达 **RET** 指令时，返回地址现在对应于 **RETURN** 指令，所以执行过程

跳转到这个 **RETURN**，并停止。

sub 标记之后的指令定义了一个所谓的子例程。它有点像“方法”，可以从一个正常方法的不同地方“调用”。在 Java 6 之前，子例程用于编译 Java 中的 **finally** 块。但事实上，子例程并非必不可少的：实际上，有可能用相应子例程的主体来代替每个 **JSR** 指令。这种**内联**生成了重复代码，但删除了 **JSR** 和 **RET** 指令。对于上面的例子，结果非常简单：

```
IINC 0 1
IINC 0 1
RETURN
```

ASM 在 **org.objectweb.asm.commons** 包中提供了一个 **JSRInlinerAdapter** 类，它可以自动执行这一转换。可以用它来删除 **JSR** 和 **RET** 指令，以简化代码分析，或者将类从 1.5 或更低版本转换为 1.6 或更高版本。

A.3 属性

2.1.1 节曾经解释过，有可能将任意**属性**关联到类、字段和方法。在引入新特性时，这种可扩展机制对于扩展类文件格式非常有用。例如，它已经被用于扩展这一格式，以支持注释、泛型、栈映射帧等。这一机制还可由用户使用，而不只是由 Sun 公司使用，但自从在 Java 5 中引入注释之后，**注释的使用就比属性容易得多**。也就是说，如果你**真的**需要使用自己的属性，或者必须管理由别人定义的非标准属性，可以在 ASM 用 **Attribute** 类完成。

默认情况下，**ClassReader** 类会为它找到的每个标准属性创建一个 **Attribute** 实例，并以这个实例为参数，调用 **visitAttribute** 方法（至于是 **ClassVisitor**、**FieldVisitor**，还是 MethodVisitor 类的该方法，则取决于上下文）。这个实例中包含了属性的原始内容，其形式为私有字节数组。在访问这种未知属性时，**ClassWriter** 类就是将这个原始字节数组复制到它构建的类中。**这一默认行为只有在使用 2.2.4 节介绍的优化时才是安全的**（除了提高性能外，这是使用该优化的另一原因）。没有这一选项，原内容可能会与类编写器创建的新常量池不一致，从而导致类文件被损坏。

默认情况下，非标准属性会以它在已转换类中的形式被复制，它的内容对 ASM 和用户来说是完全不透明的。如果需要访问这一内容，必须首先定义一个 **Attribute** 子类，能够对原内容进行解码并重新编码。还必须在 **ClassReader.accept** 方法中传送这个类的一个原型实例，使这个类可以解码这一类型的属性。让我们用一个例子来说明这一点。下面的类可用于运行一个设想的“注释”特性，它的原始内容是一个 **short** 值，引用存储在常量池中的一个 UTF8 字符串：

```
class CommentAttribute extends Attribute {
  private String comment;
  public CommentAttribute(final String comment) {
    super("Comment");
    this.comment = comment;
  }
  public String getComment() {
    return comment;
  }
  @Override
  public boolean isUnknown() {
```

```
      return false;
    }
    @Override
    protected Attribute read(ClassReader cr, int off, int len,
        char[] buf, int codeOff, Label[] labels) {
      return new CommentAttribute(cr.readUTF8(off, buf));
    }
    @Override
    protected ByteVector write(ClassWriter cw, byte[] code, int len,
        int maxStack, int maxLocals) {
      return new ByteVector().putShort(cw.newUTF8(comment));
    }
  }
```

最重要的方法是 **read** 和 **write** 方法。**read** 方法对这一类型的属性的原始内容进行解码，**write** 方法执行逆操作。注意，**read** 方法必须返回一个**新的**属性实例。为了在读取一个类时实现这种属性的解码，必须使用：

```
ClassReader cr = ...;
ClassVisitor cv = ...;
cr.accept(cv, new Attribute[] { new CommentAttribute("") }, 0);
```

这个“注释”属性将被识别，并为它们中的每一个都创建一个 **CommentAttribute** 实例（而未知属性仍将用 **Attribute** 实例表示）。

A.4 规则

现在回忆一下为确保你的代码能与较早的 ASM 版本保持后向兼容性而必须遵循的规则（见第 5 章和第 10 章）。

规则 1：要为 ASM X 编写一个 **ClassVisitor** 子类，就以这个版本号为参数，调用 **ClassVisitor** 构造器，在这个版本的 **ClassVisitor** 类中，**绝对不要重写或调用被弃用的方法**（或者将在之后版本引入的方法）。

规则 2：不要使用访问器的继承，而要使用委托（即访问器链）。一种好的做法是让你的访问器类在默认情况下成为 final 的，以确保这一特性。

规则 3：要用 ASM 版本 X 的树 API 编写类分析器或适配器，则使用以这一确切版本为参数的构造器创建 **ClassNode**（而不是使用没有参数的默认构造器）。

规则 4：要用 ASM 版本 X 的树 API 编写一个类分析器或适配器，使用别人**创建**的 **ClassNode**，在以任何方式使用这个 **ClassNode** 之前，都要以这个确切版本号为参数，调用它的 **check()** 方法。

规则 1 和 2 还适用于 **ClassNode**、**MethodNode** 等的子类，**asm.tree.analysis** 中 **Interpreter** 及其子类的子类，**asm.util** 中 **ASMifier**、**Texifier** 或 **Check*Xxx*Adapter** 类的子类，**asm.commons** 包中任意类的子类。最后，规则 2 有两个例外：

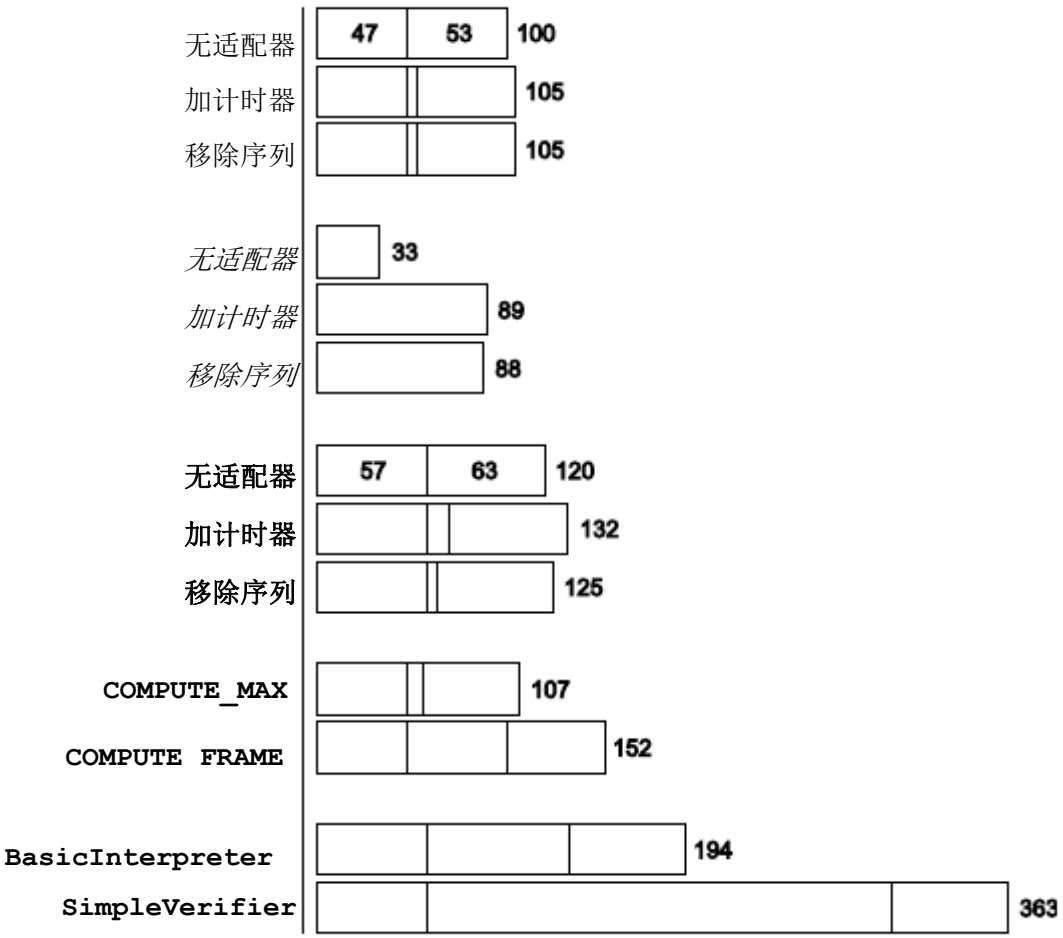
- 如果能够完全由自己控制继承链，并同时发布层次结构中的所有类，那就可以使用访问器的继承。然后必须确保层次结构中的所有类都是为同一 ASM 版本编写的。仍然要让

层次结构的叶类是 final 的。

- 如果除了叶类之外，没有其他类重写任何访问方法（例如，如果只是为了引入方便的方法而在 **ClassVisitor** 和具体访问类之间使用了中间类），那就可以使用“访问器”的继承。仍然要让层次结构的叶类是 final 的（除非它们也没有重写任何访问方法；在这种情况下，提供一个以 ASM 版本为参数的构造器，使子类可以指定它们是为哪个版本编写的）。

A.5 性能

下图给出核心与树 API、**ClassWriter** 选项和分析框架的相对性能（越短越快）：



引用时间 100 对应于直接链接到 **ClassWriter** 的 **ClassReader**。“加计时器”和“移除序列”测试对应于 **AddTimerAdapter** 和 **RemoveGetFieldPutFieldAdapter**（*斜体*表示使用 2.2.4 节所述的优化，**粗体**表示使用树 API）。总转换时间分解为三个部分：类分析（下）、类转换或分析（中间）和类的写入（上）。对于每个测试，测量值都是分析、转换和写入一个**字节数组**所需要的时间，也就是从磁盘加载类并将它们加载到 JVM 所需要的时间**未**考虑在内。为

获得这些结果，将每个测试对于 **JDK 7 rt.jar** 上的 18600 多个类运行 10 次，并采用最佳运行时获得的性能。

快速分析一下这些结果表明：

- 90%的转换时间用于类分析和写入。
- “复制常量池”优化可提速 15-20%。
- 基于树的转换要比基于访问器的慢大约 25%。
- **COMPUTE_MAXS** 选项不会耗时太多。
- **COMPUTE_FRAMES** 选项耗时很多⇒进行增量帧更新。
- 分析包的成本非常高！