|  |
| --- |
| ASM 4.0 |
| 一个Java字节码工程库 |

|  |  |
| --- | --- |
| 作者 | Eric Bruneton |
| 译者 | 王俊超 |

# 目　录

# 介绍

## 动机

程序分析、生成和转换是非常有用的技术，它可以应用在许多的场景中。

* 程序分析，其范围可以从一个简单的句法分析到全语义分析，可以用来发现应用中潜在的错误，检测未用的代码，进行逆向工程代码等。
* 程序生成被用于编译器中。包括传统的编译器，也包括分布式编程中的存根（stub）或者骨架（skeleton）编译器，实时（Just in Time）编译器等。
* 程序转换可用于优化或混淆程序，将调试或性能监视代码到应用程序中，面向切面编程等。

所有这些技术可以用于任何编程语言，但这或多或少取决于使用的语言。就Java而言，它们可以在Java源代码或编译的Java类中使用。在编译后的class类中使用很明显的优点之一是，不需要代码。程序转换，因此可以应用于任何应用，包括闭源代码和商业的。另一个优点是，使用编译的代码，使得在运行时分析，生成或变换类成为可能，类在被装载到Java虚拟机（生成和编译源代码在运行时是可能的，但是这一过程非常缓慢，且需要一个完整的Java编译器）之前完成这些动作。其优点在于一些工具如存根编译器（stub compilers）或切面织入器（aspect weavers）变得对用户透明。

由于程序分析，生成和转换技术的多种可能的用法，对各种编程语言，许多工具已经被实现，用来分析、生成和转换程序，其中就包括Java。ASM就是这些工具中的一个，用于Java语言，它被设计成用于运行时（也可以用于脱机）类生成和转换。因此，ASM 1被设计工作于编译好的Java类，同进也尽可能的快和小。应用为了实现动态类生成或者转换而使用了ASM，为了不过多的拖慢应用的速度，ASM设计得尽可能快是非常重要的。为了让ASM能应于于内存受限的环境和避免应用或者类库膨涨，ASM设计得尽可能的小也是非常重要的。

ASM不仅仅是生成和转换编译后的Java类的的工具，它也是最新最有效的一个。可以从<http://asm.objectweb.org>上下载ASM，它包括以下几个方面的优点：

* 它有一个简单，精心设计和模块化的API，易于使用。
* 这是有据可查的，并有一个相关的Eclipse插件。
* 它提供了最新的Java版本，Java 7的支持。
* 它体积小，速度快，而且非常强大。
* 其广大的用户群体可以为新用户的支持。
* 它的开源许可，您可以在几乎任何你想要的方式使用它。

## 综述

### 应用范围

ASM库的目标是生成，转换和分析编译的Java类，这些Java类以字节数组形式表示（因为它们存储在磁盘上，并被Java虚拟机加载）。为此，ASM提供了工具来读，写和转换这此字节数组，通过高层概念而不是字节来实现，高层概念包括

使用更高级别的概念不是字节，如数字常量字符串，Java的标识符，Java类型，Java类结构元件等。数字常量、字符串、Java的标识符、Java类型、Java类结构元无素等。注意，ASM库的范围被严格限制于读，写，转化和分析类。尤其是类加载过程是超出ASM关注的范围。

### 模块划分

ASM库提供了两套API用于生成和转换编译后的类：核心API提供了类基于事件的表示，树形API提供类基于对象的表示。

基于事件的模型，一个类代表了一系列的事件，每一个事件代表了一个类元素，比如类的头部、字段、方法声明以及指令等等。基于事件的API定义了一组可能的事件以及这些事件必然发生的顺序，并提供了一类解析器(class parser)为每一个被解析的元素生成一个事件，同时也提供了一个类写者（class witer）从这一系列的事件中生成编译后的类。

基于对象的模型，一个类以一个对象树进行表示，每一个对象代表了类的一部分，比如类本身、字段、方法和指令等；同时每一个对象也包含代表它成分的引用对象。基于对象的API提供了一种转换方式，可以将代表一个类的一系列事件转换成代表同样一个类的对象树。反之亦然，即将一个对象树转换成与之等价的事件序列。换句话说，基于对象的API是建立在基于事件的API之上。

这两种API可以类比XML的SAX和DOM，基于事件的API与SAX类似，基于对象的API与DOM类似。基于对象的API是建立在基于事件的API之上，就如同DOM在SAX之上。

ASM提供了两种API，因为这两种API都不是最佳的。确实每一种API都有自己的优点和缺点。

* 基于事件的API比基于对象的API更快，需要更少的内存，因为不需要创建和存储代表一个类的对象树（同样的区别也存在于SAX和DOM中）
* 但是使用基于事件的API在实现类转换时更加困难，因为在任何时候可能获取到一个类的一个元素（即对应于当前事件的元素），使用基于对象的API在内存中可以获取整个类。

注意这两个API在同一时刻可能管理一个类，并且彼此独立。也不会维护类的继承信息。如如果一个类的转换影响到了其它的类，它取决于用户是否要修改这些影响到的类。

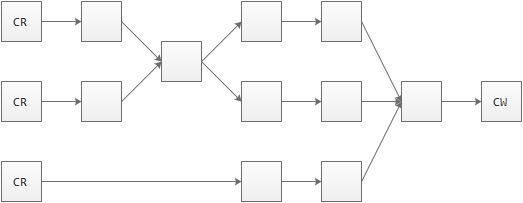
### 应用结构

ASM应用有一个强的结构化形态。事实上，基于事件的API是围绕事件生产者（类解析器），事件消费者（类写才）和各种预定义事件过滤器而组织的，可以向其中加入用户定义的生产者，消费者和过滤器。因此使用此API是一个两步过程：

* 将事件生产者，过滤器和消费组件装配到可能是复杂的架构当中，
* 然后启动事件生成器执行生成或者转换过程。

基于对象的API同样有一个强的结构化形态。实事上，操作在对象树之上的类生成器和转换器组件可以被组合在一起，它们之间的链接关系表示转换的顺序。

诚然，在大多数典型的ASM应用中组件的结构非常简单。可以想象如下图的复杂组件，箭头代表基于事件或者对象的的通信，这种通信在类解析器，类写者和类转换器之间。基于事件和基于对象之间的可能转换在任何地方都存在，以线条进行表示。



## 内容组织

ASM库由几个包组成，分布在几个jar文件中：

* org.objectweb.asm和org.objectweb.asm.signature包定义了基于事件的API，并提供类解析器和类写者组件。它们都包含在asm.jar归档库中。
* org.objectweb.asm.util包，在ASM-util.jar归档库中，提供基于核心API的工具，可以在ASM应用程序的开发与调试期间使用。
* org.objectweb.asm.commons包提供了一些有用的预定义类转换器，它们主要是基于核心API，这个包在ASM-commons.jar归档库中。
* org.objectweb.asm.tree包在ASM-tree.jar归档库中，定义了基于对象的API，提供一些工具用以基于事件和基于对象的类的表示之间进行转换。
* org.objectweb.asm.tree.analysis包提供了基于树型API的一个类分析框架和几个预定义的类分析器。它包含在ASM-analysis.jar归档库中。

本文档分为两个部分，第一部分包括核心API，即asm，asm-util和asm-util归档库。第二部分涵盖了树型API，即asm-tree和ASM-analysis归档库。每一部分至少包含四个章节。与类、方法、注解、泛型等相关的API各一个章节，还有其它章节等。每章包含了程序接口、相关的工具以及预定义的组件。所有样例的源代码可以从[ASM](http://asm.objectweb.org/)网站上获取。

这种内容组织形式可以更加容易的逐步介绍类文件特性，但有时在某些章节需要单独地介绍ASM类的特性。因此建议以序列方式读取此文档。有关ASM API的参考指南，请使用的Javadoc文档。

**印刷字体约定**

斜体用于句子强调。

等宽字体（使用Consolas）用于代码片段。

等宽粗体（使用**Consolas**）用于强调的代码元素。

等宽粗斜体（使用***Consolas***）用于代码变量和标签。

## 致谢

感谢François Horn在本文档成形过程中的宝贵建议，它改善了文档的结构和可读性。

# 第I部分 核心API

## 类

本章阐述如何使用核心ASM API去生成和转换一个已经编译好的Java类。首先会对编译好的类进行说明，之后会使用样例的方式介绍生成和转换这此类相关的ASM接口、组件和工具。方法内容、注解和泛型将在下章介绍。

### 结构

#### 概览

编译好的类的整个结构非常简单。确实，与本地编译的应用不一样，一个编译好的Java类保留了结构信息以及来自源代码的几乎所有的符号。事实上一个编译好的类包含：

一个部分包含一个描述类的修饰符（如public或者private）、名称、父类、接口和注解。

* 一个部分包含一个类中声明的一个字段，每一个字段都有一个这样部分。每一个这样的部分描述了字段的修饰符、名称、类型和注解。
* 一个部分包含一个类中声明的方法和*构造函数*，每一个方法和构造函数都有一个这样部分。每一个样的部分描述了方法的修饰符、名称、返回类型和参数类型。它也包括方法编译后的代码，以一种Java字节码的形式存在

然而编译好的类和源代码之间也存在不一样的地方：

* 一个编译好的类仅仅描述一个类的信息，但是源文件可以包含几个类。例如一个源文件描述了一个类，这个类有一个内部类，在编译的时候会产生两个类文件：一个主类和一个内部类。但是主类包含了对内部类的*引用*。定义在方法中的内部类包含了指向包含方法的*引用*。
* 一个编译好的类也不公包含注释。但是会包含类、字段、方法和代码*属性*。它们可以用于关联这些元素的附加信息。由于Java 5中引入了注解，其可被用于同样的目的，属性已基本无用。
* 一个编译好的类不会包含package和import部分，所有的类型都会使用全限定名。

另一个巨大的结构化差异是一个编译好的类会包含一个常量池（constant pool）部分。常量池是一个数组，包含了类中出现的所有数字、字符串和类型常量。这些常量在常量池部分只被定义过一次，并且在类文件中的其它部分按照索引被引用。所幸的是ASM隐藏了常量池相关的细节，因而你无需要为他苦恼。图2.1总结了编译后的类的整个结构。其细节结构在Java虚拟机规范中的第四部分有阐述。

|  |  |
| --- | --- |
| 修饰符、名称、父类、接口 | |
| 常量池：数字、字符串和类型常量 | |
| 源文件名（可选） | |
| 封闭类引用 | |
| 注解\* | |
| 属性\* | |
| 内部类\* | 名称 |
| 字段\* | 修饰符、名称、类型 |
| 注解\* |
| 属性\* |
| 方法 | 修饰符、名称、返回类型和参数类型 |
| 注解\* |
| 属性\* |
| 编译后的代码 |

图2.1 编译后的类的整体结构（\*表示一个或者多个）

另一个显著的不是的Java类型的表示在编译后的类和源代码不一样。下一节介绍他们在编译后的类中是如何表示的。

#### 内部名称

在许多场景中，一个类型被约束为一个类或者一个接口。例如一个类的父类、被其它类实现的接口或者一个方法中抛出的异常，他们不是无始类型或者数组类型，必须是类类型或者接口类型。这些类型在编译好的类中以内部名称（internal name）进行表示。一个类的内部名称就是一个类的全限定名，点使用正斜杠替代。例如，String的内部名称就是java/lang/String

#### 类型描述符

内部名称仅用于被约束为类型为类或者接口的情况。在其它的情况，比如字段类型、Java类型在编译好的类中以类型描述符（type descriptors）的方式进行表示（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 |
| doube | 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

元始类型的描述符是单个符号：Z代表boolean，C代表char，B代表byte，S代表short，I代表int，F代表float，J代表long以及D代表double。类中类类型的类型描述符以L打头，以分号结尾，中间是类的内部名称。例如，String的类型描述符是Ljava/lang/String;。最后数组类型的类型描述符是以左方括号开始，后面接数组元素的类型描述符。

#### 方法描述符

方法描述符（method descriptor）是一个类型列表，它以一个字符串的形式描述了一个方法的参数类型和返回类型。一个方法的描述符是以一个左圆括号开始，接下来是参数的类型描述符，接下来是右圆括号，最后是返回类型的类型描述符，如果方法返回是void就使用V代替（一个方法的描述符不包括方法的名称和参数名称）。

一旦你知道了类型描述符是如何工作的，明白方法描述符是非常容易的事情了。例如(I)I描述一个方法，它具有一个int类型的参数，返回一个int结果。图2.3给出了几个方法描述符的例子。

### 接口和组件

#### 表示

用于生成和转换编译后的类的ASM API是基于ClassVisitor抽类（见图2.4）。ClassVisitor类的每一个方法与类文件结构相同名字的部分相对应（见图2.1）。简单的部分使用简单的方法（简单的方法就是它的参数描述了它的内容，返回类型是void）调用进行访问。内容可以是任意长度并且非常复杂的部分以一个初始方法进行访问，这个初始方法返回一个辅助访问者类。比如visitAnnotation、visitField和visitMethod方法分别返回AnnotationVisitor，FieldVisitor和MethodVisitor。

同样了规则也递归地用于这些辅助类。例如，在ClassVisitor中，FieldVisitor抽象类中的每一个方法与类文件同名的主干结构相对应，visitAnnotation返回一个辅助对象AnnotationVisitor，创建和使用这些辅助类将在下一章节进行阐述：事实上，本章仅限于可单独用ClassVisitor类来解决简单的问题。

|  |
| --- |
| **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

|  |
| --- |
| **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类中的方法必须以如下方式的顺序进行调用，具体的细节请见这个类的Javadoc文档。

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

这意味着visit必须首先被调用，然后最多调用一次visitSource，之后最多调用一次visitOuterClass，接下来可以以任意顺序和次序调用visitAnnotation和visitAttribut，接下来可以以任意顺序和次序调用visitInnerClass、visitField和visitMethod，最终以一次visitEnd调用结束。

ASM提供了基于ClassVisitor API的三个核心组件用于生成和转换类：

* ClassReader类解析字节数组形式的编译后的类，并且在ClassVisitor实例（它以参加的形式传递到accept方法）上调用相应的visit*Xxx*方法。它可以看作是事件的生产者。
* ClassWriter类是ClassVisitor抽象类的一个字类，ClassVisitor类直接以进制的形式构建编译后的类。ClassWriter产生的输出就是包含编译后的类的字节数组。编译后的类可以通过toByteArray方法恢复。它可以看作是事件的消费者
* ClassVisitor类代理它接收到的其它的ClassVisitor实例的所有的方法调用。它可以看作是一个事件的过滤器。

下一节将展示一个具体的实例，解释如何使用这些组件来生成和转换一个类。

#### 解析类

唯一需要的组件来解析现有类的是ClassReader组件。让我们举个例子来说明这一点。假设我们想要打印的一个类的内容，如同javap工具相同的方式。第一步就是写一个ClassVisitor的子类，它打印所访问的类的信息。以下是可能的，极简的实现：

|  |
| --- |
| **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(**"}"**);  } } |

第二步就是使用ClassReader组件来组合ClassPrinter，使得ClassReader产生的事件可以被ClassPrinter消费：

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

第二行创建一个ClassReader对象来解析Runnable类。最后一行调用的accept方法解析Runnable类的字节码，解析时使用cp上与ClassVisitor相对应的方法。如果是如下的输出：

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

注意：有几种方式可以创建一个ClassReader实例，这个必须被读取的类可以通过名称指定、或者值、或者是字节数组以及InputStream流。读取一个类内容的输入流可以通过ClassLoader的getResourceAsStream方法获取，如：

|  |
| --- |
| *cl*.getResourceAsStream(*classname*.replace(’.’, ’/’) + ".class"); |

#### 生成类

生成一个类仅需要的组件是ClassWirter组件。让我们以一个例子来进行说明，考虑下面一个接口：

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

只需要ClassVisitor的六个方法调用就可以生成：

|  |
| --- |
| ClassWriter cw = **new** ClassWriter(**0**); cw.visit(***V1\_5***, ***ACC\_PUBLIC*** + ***ACC\_ABSTRACT*** + ***ACC\_INTERFACE***,  **"pkg/Comparable"**, **null**, **"java/lang/Object"**,  **new** String[] { **"pkg/Mesurable"** }); cw.visitField(***ACC\_PUBLIC*** + ***ACC\_FINAL*** + ***ACC\_STATIC***, **"LESS"**, **"I"**,  **null**, **new** Integer(-**1**)).visitEnd(); cw.visitField(***ACC\_PUBLIC*** + ***ACC\_FINAL*** + ***ACC\_STATIC***, **"EQUAL"**, **"I"**,  **null**, **new** Integer(**0**)).visitEnd(); cw.visitField(***ACC\_PUBLIC*** + ***ACC\_FINAL*** + ***ACC\_STATIC***, **"GREATER"**, **"I"**,  **null**, **new** Integer(**1**)).visitEnd(); cw.visitMethod(***ACC\_PUBLIC*** + ***ACC\_ABSTRACT***, **"compareTo"**,  **"(Ljava/lang/Object;)I"**, **null**, **null**).visitEnd(); cw.visitEnd(); **byte**[] b = cw.toByteArray(); |

第一行创建了一个ClassWriter实例，它实质会创建表示类的字节数组（构造函数的参数将下一部分进说明）。

visit方法调用确定了类的头部，如同其它的ASM常量一样，V1\_5参数是一个定义在ASM Opcodes接口中的常量。它指明了类的版本是Java 1.5。ACC\_XXX常量是一个标志，与Java修饰符相对应。在这里我们指定了这个类是一个接口，它是public和abstruct的（因为它不能被实例化）。下一个参数以内部名称（见章节2.1.1.2）的形式指定类的名称。回想，一个编译后的类不会包含一个包或者引入的部分，因此所有的类必须是全限定名。下一个参数对应的是泛型（见XXX）。在我们这个例子中它是null，因为这个接口不是没有使用类型参数进行参数化。第一个参数是以内部名称（接口类隐式地继承与Object类）表示的父类。最后一个参加是以内部名称表示的一组需要扩展的接口。

三个visitFiled方法是类似的，用于定义接口的三个字段。第一个参数设置了与Java修饰符相对应的一系列标志。在这里我们将字段设置为public、final和static。第二个参数是字段的名字，如同它在源码中的那样。第三个参数是字段的类型，以类型描述符的形式。这里字段的类型是int，描述符是I。第四个参数是与之对应的泛型，在本示例中是null，因为未使用泛型。最后一个参数是字段的常量值（constant value），这个参数必须用于常量字段，即：final static字段。其它的参数这个字段必须为null。因为没有使用注解，我们使用visitEnd方法立即返回一个FieldVisitor对象，即不需要任何的visitAnnotation和visitAttribute方法调用。

visiteMethod调用用来定义compareTo方法。第一个参数是与Java修饰符相对应的一系列标志。第二个参数是方法的名称，如同它在源码中的那样。第三个参数是方法的描述符。第四个参数是泛型类型。在本示例中是null，因为未使用泛型。最后一个参数是方法抛出的一系列异常，通过内部名称指定这些异常。在这里是null因为方法没有声明任何异常。visitMethod方法返回一个MehtodVisitor对象（见图XXXX），它用于定义一个方法的注解和属性，以及最重要的方法代码。在这里因为没有注解以及方法是抽象的，我们调用visitEnd方法立即返回了一个MethodVisitor对象，

最后使用visitEnd调用通知道cw类的创建已经完成，并且使用toByteArray方法返回一个代表类的字节数组。

##### 使用生成的类

上一节生成的字节数组可以存储为Compare.class文件，以备将来使用。一种方式是，字节数组被ClasssLoader动态加载。ClassLoader子类中要定义一个defineClass方法，这个方法是公有的：

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

然后这个生成的类可以直接被加载：

|  |
| --- |
| Class c = myClassLoader.defineClass(**"pkg.Comparable"**, b); |

另一种更清爽的方法去加载一个类就是定义一个一个ClassLoader子类，它的findClass被重写，以便凭空生成这个请求的类。

|  |
| --- |
| **public class** StubClassLoader **extends** ClassLoader {  @Override  **protected** Class findClass(String name) **throws** ClassNotFoundException {  **if** (!name.endsWith(**"\_Stub"**)) { *// 与原文档的代码不一样，多了一个非（!）* ClassWriter cw = **new** ClassWriter(**0**);   *// ...*  **byte**[] b = cw.toByteArray();  **return** defineClass(name, b, **0**, b.**length**);  }  **return super**.findClass(name);  } } |

事实上生成类的方式依赖于上下文，也超了ASM API的范围。假如你在写一个编译器，类的生成过程将通过抽象语法树驱动，抽象语法树代表了编译过程。生成的类会被保存到磁盘上。如果你在写一个动态代理类生成器或者切面织入器，你将以一种形式或者另一种形式使用ClassLoader。

#### 转换类

到目前为止，ClassReader和ClassWriter组都是单独使用的，事件是手动生成的，被ClassWriter直接消费，也就是，事件由ClassReader生成，手动消费。即通过ClassVisitor实现。当这些组件组件在一起使用的时候，事件会变得非常有趣。第一步就是指派CassReader产生的事件到ClassWriter。结果就是由读者解析的类由类写者改造。

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

这本身一点都不有趣（有一种更容易的字节数组拷贝的方式），但是请稍候。下一步将介绍类读者和类写才之间的ClassVisitor：

|  |
| --- |
| **byte**[] b1 = ...;  ClassWriter cw = new ClassWriter(0);  *// cv forwards all events to cw* ClassVisitor cv = **new** ClassVisitor(***ASM4***, cw) { }; ClassReader cr = **new** ClassReader(b1); cr.accept(cw, **0**); **byte**[] b2 = cw.toByteArray(); *// b2 represents the same class as b1* |

与上面代码相对应的结构如图2.6所示：

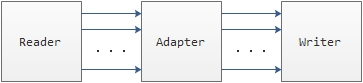


图2.6 转换链

结果并不会改变，但是，因为ClassVisitor事件求过滤器没有过滤任何内容。但是现在为了能够转换一个类，通过重写一些方法进行实现，它已经足够过滤一些事件。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);  } } |

这个类只重写了ClassVisitor类的一个方法。结果是所有的调用无改变的转发到类写者cv，然后传递给构造函数。除了调用某些特定的访问方法，这些方法带有类版本号进行转发。相应的顺序图如图2.7所示。

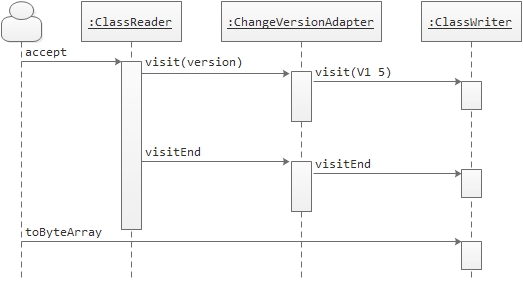


图2.7 ChangeVersionAdapter顺序图

通过修改visit方法的其它参数，你可以实现另外一些转换，而不是仅仅只改变类的版本。例如，你可以在实现的接口列表中添加一个接口。也可以改变一个类的名称，*但是这个需要的不仅仅是改变****visit****方法的****名称****参数*。事实上，在一编译好的类中，一个类的名称可以出现在许多不同的地方，**所有**这些出现的地方必须同样被改变。

#### 移除类成员

在上一节中用于转换类的方法同样可以应用于ClassVisitor类的其它方法。例如通过改变visitFiled的access或者name参数，你可以修改一个字段或者方法的修饰符或者名称。此外，除了转发具有修改参数的方法调用，您可以选择不转发此调用。结果就与之相对应的类元素就会被删除掉。

例如，下面的类适配器移除了外部类和内部类的信息，同样也移除了编译这个类的源文件名（结果类仍保持完全正常，因为这些元素仅用于调试目的）。这是通过不转发任何匹配的访问方法实现的：

|  |
| --- |
| **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) {  } } |

这种方式在字段和方法上并不能实现，因为visitField和visitMethod方法必须返回一个结果。国了移除一个字段或者方法，你必须不转发方法调用，并返回一个null给调用者。例如下面的类适配器移除了一个方法，通过名字和描述符进行了指定（单纯的名字不足以确定一个方法，因为一个类可以包含几个相同名字的方法但是参数不同的方法）。

|  |
| --- |
| **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**)) {  *// do not delegate to next visitor -> this removes the method* **return null**;  }  **return cv**.visitMethod(access, name, desc, signature, exceptions);  } } |

#### 添加类成员

与少转发你接收到的调用相反，你可以做更多的转发，它的影响就是会**增加**类的元素。新的方法调用可以插入到原来方法调用的几个位置，只要遵从visitXxx方法调用的先后顺序（见章节2.1.2.1）。

例如，如果你想添加一个字段到一个类中，你必须在原来方法调用之间插入一个新的visitField方法，并且你必须把这个新的方法调用放入到你的类适配器中的访问方法的某一个中。你不能在visit方法中这么做，例如，这可能会导致visitSource、visitOuterClass、visitAnnotation或者visitAttribute方法后面会接着调用visitField，这是不合法的。同样的原因，你不能把新的方法调用入到visitSource、visitOuterClass、visitAnnotation或者visitAttribute方法中。唯一可以放入的方法是visitInnerClass、visitField、visitMethod或者visitEnd方法。

假如你把新的调用放入visitEnd方法中，这个字段总是会被添加（除非你们添加了明确的条件），因为这个方法总是会被调用。如果你把方法放入到visitField或才visitMethod中，则会添加**几个**字段，原来类中有几个字段或者方法就会添加几个。两种方法都是有意义的，这取决于你的需求。例如，你可以添加一个计数器字段来计算的对象的调用；或每个方法一个计数器，分别计算每个方法的调用。

**注意**：事实上唯一正确的添加新成员的方法就是在visitEnd方法中调用额外的方法。事实上，一个类不能包含重复的成员，保证一个新成员是唯一的方法是将它与已经存在的成员进行比较，一旦他们被访问了，这个只能被执行一次，即：在visitEnd方法中。这相当具有局限性，使用不太可能被程序员使用的生成的名字，比如\_counter$或者\_4B7F\_，在实践当中是足够避免重复成员出现，而且不需要在visitEnd方法加添加他们。注意：正如第一章讨论的，树形API没有这方面的限制，使用树形API的转换中，任何时候都可以添加一个新的成员。

为了说明上面的讨论，下面是给一个类添加字段的类适配器，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();  } } |

这个字段在visitEnd方法中被添加。visitField方法没有被重写，而修改或者删除已经存在的字段，仅仅是判断我们要添加的这个字段是否存在。注意：fv != null 方法在调用fv.visitEnd()前进行检测，这是因为正如我们之前几节看到的，一个类的访问器在visitField方法中可以返回null。

#### 转换链

目前我们们看到了由ClassReader、类适配器和ClassWriter组成的简单转换链。当用几个类配置器链接在一起时，可以用于更复杂的转换链。为了做一些复杂的转换，链接几个适配器使得你可以组合几个独立的类转换器。同时也要注意：一个转换链不必是线性的。你可以写一个ClassVisitor同时转发所有收到的ClassVisitor方法调用：

|  |
| --- |
| **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);  }  } } |

对称地，多个类适配器可以委托给同一个ClassVisitor（这需要一些条件要保证，例如，visit和visitEnd方法在ClassVisitor类中仅能调用一次）。因此，一个转换链，例图2.8所示的，是完全可能的。

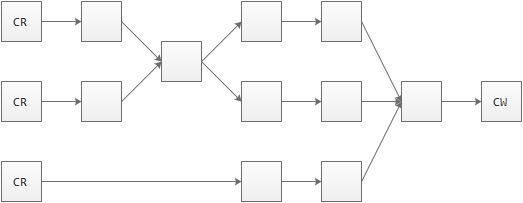


图2.8 复杂转换链

### 工具

除了ClassVisitor类和相关的ClassReader以以及ClassWriter组件，ASM在og.objectweb.asm.util包中，提供了几个工具，在开发一个类生成器或者适配器过程中非常有用，在运行期，这些工具是不必要的。ASM同时也提供了一个工具类在运行时来操作内部名称、类型描述符和方法描述符。所有这些工具在接下来会一一介绍。

#### Type

正如你在前几节看到的，ASM API暴露了存储在编译好的类中的Java类型，即内部名称或者类型描述符。如同它们出现在源代码中一样，暴露它们使得代码更具有可读性。但是这需要在ClassReader和ClassWriter中的两种表示之间做系统的转的转换，也会降低性能。这就是为什么ASM没有透明地将内部名称和类描述符转成与之相等的源代码形式。但是，当需要的时候ASM API提供了Type类来进行手动操作。

一个Type对象代表了一个Java类型，可以从类型描述符或者Class对象中创建。Type类也包含了静态变量，代表了原始类型。例如，Type.INT\_TYPE是Type对象，代表了int类型。

getInternalName方法返回Type的内部名称。例如：Type.getType(String.class).getInternalName()返回了String类的内部名称。即：“java/lang/String”。这个方法仅用于类或者接口类型。

getDescriptor方法返回Type的描述符。因而，例如，与使用“Ljava/lang/String;”不同，在你的代码中，你可能使用Type.getType(String.class).getDescriptor()。或者与使用I不同，你可以使用Type.INT\_TYPE.getDescriptor()。

一个Type对象同样也可以代表一个方法类型。这个对象可以从一个方法描述符或者Method对象中创建。getDescriptor方法返回与该类型对应的方法描述符。加之，getArgumentTypes和getReturnType方法可以用于获取与方法的参数类型和返回类型一致的Type对象。例如：Type.getArgumentTypes(“(I)V”)返回一个包含一个元素Type.INT\_TYPE的数组。类似地，Type.getReturnType(“(I)V”)返回Type.VOID\_TYPE对象。

#### TraceClassVisitor

为了检查一个生成或者转换的类是否符合你所期待。ClassWriter返回的字节数组并不是非常有用的，因为对人们来说，它并不具体很好的可读性。文本表示更加容易使用。这就是为什么要提交TraceClassVisitor类。这个类，如同它的名称所暗示的，拓展了ClassVisitor类，并且构建了被访问类的文体表示。因此，为了获取实际生成的可读的内容，你可以使用TraceClassVisitor，而不是使用ClassWriter生成类。或者，甚至更好，你可以同时使用这两个。事实上，除了默认的行为这外，TraceClassVisitor可以将它方法上所有调用代理到另一个访问者上，例如：ClassWriter。

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

这段代码创建了一个TraceClassVisitor对象，它代理了从cw上接收到的所有调用，同时把这些调用的文本表示输出到了printWriter，例如，使用2.1.3.3节的TraceClassVisitor示例将会输出：

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

注意：为了查看转换链上某一点到底发生了什么，你可以在生成或者转换链的任何一点使用TraceClassVisitor，不仅仅在ClassWriter之前。这个适配器生成的类的文本表示可以容易地使用String.equals()用于类比较。

#### CheckClassAdapter

ClassWriter类不会检查方法是否以合适的序列和正确的参数被调用。因此可能会产生被Java虚拟机验证器拒绝的非法类。为了尽快地检查这些错误，可以使用CheckClassAdapter。和TraceClassVisitor类似，它扩展了ClassVisitor类的功能，并且代理它的所有方法调用到另一个ClassVisitor上，比如TraceClassVisitor或者ClassWriter。然而，与输出一个访问过的类的文本表示不同，在将方法代理到下一个访问者对象前，这个类检查方法的调用是否以合适的顺序以及正确的参数。假如出错，会抛出IllegalStateException或者IllegalArgumentException异常。

为了检查一个类，输出类的文本表示，并最终创建一个类的字节数组的表示，你应当这样操作：

|  |
| --- |
| 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：如果你以不同的顺序链接这些操作，这些操作的执行也会以不同的顺序。比如：如下代码，检查发生在跟踪*之后*。

|  |
| --- |
| ClassWriter cw = **new** ClassWriter(**0**); **CheckClassAdapter cca = new CheckClassAdapter(cw);** TraceClassVisitor cv = **new** TraceClassVisitor(cca, printWriter); |

与TraceClassVisitor类似，为了检查链上某一点类的情况，你可以在生成或者转换链上的任意一点使用CheckClassAdapter，不仅仅在ClassVisitor之前。

#### ASMifier

这个类提供了TraceClassVisitor（它默认使用Textifier后端，生产的输出如上一节所示）工具的另一个后端。这个后端使得每个TraceClassVisitor的被调用的方法输出Java代码。结果就是，当一个具有Asmifier后端的TraceClassVisitor访问器访问了一个类，它使用ASM来生成这个类的源代码。当你使用这个访问者去访问一个已经存在的类时，这个方法非常有用。例如：如果你不知道怎么使用ASM去生成一个已经编译好的类，编写相应的源代码，用javac编译，使用ASMifier访问已经编译后的类。你将会得到ASM代码来生成编译好的类。

ASM类以命令行的形式使用，例如下面的使用：

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

生成代码，经过排序后，如下：

|  |
| --- |
| **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(); } |

## 方法