|  |
| --- |
| 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。

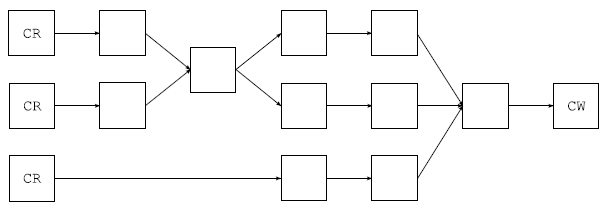
### 应用结构

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"); |

#### 类生成

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。