 本文是《[深入字节码 -- 使用 ASM 实现 AOP](http://my.oschina.net/u/1166271/blog/162796)》的后续博文。在上一篇文章中介绍了如何使用 ASM 动态安插代码到类中，从而简单实现 Aop。文章得到了广大朋友好评，我也希望可以不负众望继续写出可以得到大家认可的更多相关文章。本文主要讲解 ASM 核心接口方法和其参数意义。另外本文也可用做参考手册使用。

    ASM 4.0 核心包中包含几个关键类，这些类在ASM 3.0 时期是以接口形式提供。本文针对 ASM 4.0 编写,故不讨论 ASM 3.0环境。首先简单的将 “.class” 文件的内容看作是如下树形结构：

Class

Annotation

Annotation

...

Field

Annotation

...

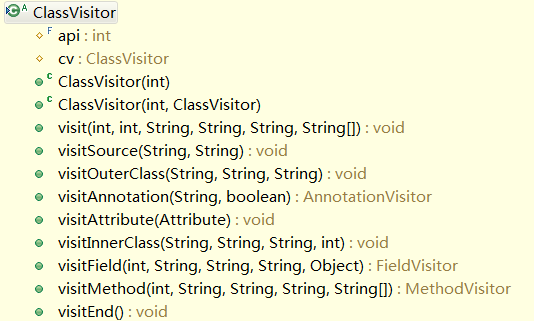
Method

Annotation

...

    ASM 在读取 “.class” 文件内容时也会按照递此顺序进行调用。每拜访一个结构中的成员都会使用相应的接口，具体关系如下：

|  |  |
| --- | --- |
| 树形关系 | 使用的接口 |
| Class | ClassVisitor |
| Field | FieldVisitor |
| Method | MethodVisitor |
| Annotation | AnnotationVisitor |

    ClassVisitor，在 ASM3.0 中是一个接口，到了 ASM4.0 与 ClassAdapter 抽象类合并。主要负责 “拜访” 类成员信息。其中包括（标记在类上的注解，类的构造方法，类的字段，类的方法，静态代码块）,它的完整接口如下：  


我将重点介绍其中几个关键方法：

**1.visit(int , int , String , String , String , String[])**  
    该方法是当扫描类时第一个拜访的方法，主要用于类声明使用。下面是对方法中各个参数的示意：**visit( 类版本 , 修饰符 , 类名 , 泛型信息 , 继承的父类 , 实现的接口)**。例如：

public class TestBean {

等价于：

visit(V1\_6, ACC\_PUBLIC | ACC\_SUPER , "org/more/test/asm/simple/TestBean",

null, "java/lang/Object", null)

**第一个参数：**表示类版本：V1\_6，表示 “.class” 文件的版本是 JDK 1.6。可用的其他版本有：V1\_1（JRE\_1.1）、V1\_2（J2SE\_1.2）、V1\_3（J2SE\_1.3）、V1\_4（J2SE\_1.4）、V1\_5（J2SE\_1.5）、V1\_6（JavaSE\_1.6）、V1\_7（JavaSE\_1.7）。我们所指的 JDK 6 或 JDK 7 实际上就是只 JDK 1.6 或 JDK 1.7。

**第二个参数：**表示类的修饰符：修饰符在 ASM 中是以 “ACC\_” 开头的常量进行定义。可以作用到类级别上的修饰符有：ACC\_PUBLIC（public）、ACC\_PRIVATE（private）、ACC\_PROTECTED（protected）、ACC\_FINAL（final）、ACC\_SUPER（extends）、ACC\_INTERFACE（接口）、ACC\_ABSTRACT（抽象类）、ACC\_ANNOTATION（注解类型）、ACC\_ENUM（枚举类型）、ACC\_DEPRECATED（标记了@Deprecated注解的类）、ACC\_SYNTHETIC。

**第三个参数：**表示类的名称：通常我们的类完整类名使用 “org.test.mypackage.MyClass” 来表示，但是到了字节码中会以路径形式表示它们 “org/test/mypackage/MyClass” 值得注意的是虽然是路径表示法但是不需要写明类的 “.class” 扩展名。

**第四个参数：**表示泛型信息，如果类并未定义任何泛型该参数为空。Java 字节码中表示泛型时分别对接口和类采取不同的定义。该参数的内容格式如下：

<泛型名:基于的类型....>Ljava/lang/Object;

<泛型名::基于的接口....>Ljava/lang/Object;

其中 “泛型名:基于的类型” 内容可以无限的写下去，例如：

public class TestBean<T,V,Z> {

泛型参数为：<T:Ljava/lang/Object;V:Ljava/lang/Object;Z:Ljava/lang/Object;>Ljava/lang/Object;

分析结构如下：

<

T:Ljava/lang/Object;

V:Ljava/lang/Object;

Z:Ljava/lang/Object;

>

Ljava/lang/Object;

再或者：

public class TestBean<T extends Date, V extends ArrayList> {

泛型参数为：<T:Ljava/util/Date;V:Ljava/util/ArrayList;>Ljava/lang/Object;

分析结构如下：

<

T:Ljava/util/Date;

V:Ljava/util/ArrayList;

>

Ljava/lang/Object;

    以上内容只是针对泛型内容是基于某个具体类型的情况，如果泛型是基于接口而非类型则定义方式会有所不同，这一点需要注意。例如：

public class TestBean<T extends Serializable, V> {

泛型参数为：<T::Ljava/io/Serializable;V:Ljava/lang/Object;>Ljava/lang/Object;

分析结构如下：

<

T::Ljava/io/Serializable; //比类型多出一个“:”

V:Ljava/lang/Object;

>

Ljava/lang/Object;

**第五个参数：**表示所继承的父类。由于 Java 的类是单根结构，即所有类都继承自 java.lang.Object 因此可以简单的理解为任何类都会具有一个父类。虽然在编写 Java 程序时我们没有去写 extends 关键字去明确继承的父类，但是 JDK在编译时 总会为我们加上 “ extends Object”。所以倘若某一天你看到这样一份代码也不要过于紧张。

**第六个参数：**表示类实现的接口，在 Java 中类是可以实现多个不同的接口因此此处是一个数组例如：

public class TestBean implements Serializable , List {

该参数会以 “[java/io/Serializable, java/util/List]” 形式出现。

    这里需要补充一些内容，如果类型其本身就是接口类型。对于该方法而言，接口的父类类型是 “java/lang/Object”，接口所继承的所有接口都会出现在第六个参数中。例如：

public inteface TestBean implements Serializable , List {

最后两个参数对应为:

"java/lang/Object", ["java/io/Serializable","java/util/List"]

**2.visitAnnotation(String , boolean)**    该方法是当扫描器扫描到类注解声明时进行调用。下面是对方法中各个参数的示意：**visitAnnotation(注解类型 , 注解是否可以在 JVM 中可见)**。例如：

@Bean({ "" })

public class TestBean {

@Bean等价于：

visitAnnotation("Lnet/hasor/core/gift/bean/Bean;", true);

下面是 @Bean 的源代码：

@Retention(RetentionPolicy.RUNTIME)

@Target({ ElementType.TYPE })

public @interface Bean {

/\*\* Bean名称。\*/

public String[] value();

}

**第一个参数：**表示的是，注解的类型。它使用的是（“L” + “类型路径” + “;”）形式表述。

**第二个参数：**表示的是，该注解是否在 JVM 中可见。这个参数的具体含义可以理解为：如果为 true 表示虚拟机可见，我们可以通过下面这样的代码获取到注解类型：

testBeanType.getAnnotation(TestAnno.class);

    谈到这里就需要额外说明一下在声明注解时常见的 “@Retention(RetentionPolicy.RUNTIME)” 标记。RetentionPolicy 是一个枚举它具备三个枚举元素其每个含义可以理解为：

    1.RetentionPolicy.SOURCE：声明注解只保留在 Java 源程序中，在编译 Java 类时注解信息不会被写入到 Class。如果使用的是这个配置 ASM 也将无法探测到这个注解。  
    2.RetentionPolicy.CLASS：声明注解仅保留在 Class 文件中，JVM 运行时并不会处理它，这意味着 ASM 可以在 visitAnnotation 时候探测到它，但是通过Class 反射无法获取到注解信息。  
    3.RetentionPolicy.RUNTIME：这是最常用的一种声明，ASM 可以探测到这个注解，同时 Java 反射也可以取得注解的信息。所有用到反射获取的注解都会用到这个配置，就是这个原因。

**3.visitField(int , String , String , String , Object)**    该方法是当扫描器扫描到类中字段时进行调用。下面是对方法中各个参数的示意：**visitField(修饰符 , 字段名 , 字段类型 , 泛型描述 , 默认值)**。例如：

public class TestBean {

private String stringData;

stringData字段等价于：

visitField(ACC\_PRIVATE, "stringData", "Ljava/lang/String;", null, null)

**第一个参数：**表示字段的修饰符，修饰符在 ASM 中是以 “ACC\_” 开头的常量进行定义。可以作用到字段级别上的修饰符有：ACC\_PUBLIC（public）、ACC\_PRIVATE（private）、ACC\_PROTECTED（protected）、ACC\_STATIC（static）、ACC\_FINAL（final）、ACC\_VOLATILE（volatile）、ACC\_TRANSIENT（transient）、ACC\_ENUM（枚举）、ACC\_DEPRECATED（标记了@Deprecated注解的字段）、ACC\_SYNTHETIC。

**第二个参数：**表示字段的名称。

**第三个参数：**表示字段的类型，其格式为：（“L” + 类型路径 + “;”）。

**第四个参数：**表示泛型信息， 泛型类型描述是使用（“T” + 泛型名 + “;”）加以说明。例如：“private T data;” 字段的泛型描述将会是 “ TT; ”， “ private V data; ” 字段的泛型描述将会是 “ TV; ”。泛型名称将会在 “visit(int , int , String , String , String , String[])” 方法中第五个参数中加以表述。例如：

public class TestBean<T, V> {

private T data;

private V value;

等价于：

visit(V1\_6, ACC\_PUBLIC | ACC\_SUPER , "org/more/test/asm/simple/TestBean",

"<T:Ljava/lang/Object;V:Ljava/lang/Object;>Ljava/lang/Object;", //定义了两个泛型类型 T 和 V

"java/lang/Object", null)

visitField(ACC\_PRIVATE, "data", "Ljava/lang/Object;", "TT;", null) //data 泛型名称为 T

visitField(ACC\_PRIVATE, "value", "Ljava/lang/Object;", "TV;", null) // value 泛型名称为 V

还有一种情况，倘若类在定义泛型时候已经基于某个类型那么生成的代码将会是如下形式：

public class TestBean<T extends Serializable, V> {

private T data;

private V value;

等价于：

visit(V1\_6, ACC\_PUBLIC | ACC\_SUPER , "org/more/test/asm/simple/TestBean",

"<T::Ljava/io/Serializable;V:Ljava/lang/Object;>Ljava/lang/Object;", //定义了两个泛型类型 T 和 V

"java/lang/Object", null)

visitField(ACC\_PRIVATE, "data", "Ljava/io/Serializable;", "TT;", null)  //data 泛型名称为 T

visitField(ACC\_PRIVATE, "value", "Ljava/lang/Object;", "TV;", null)  // value 泛型名称为 V

**第五个参数：**表示的是默认值， 由于默认值是 Object 类型大家可能以为可以是任何类型。这里要澄清一下，默认值中只能用来表述 Java 基本类型这其中包括了（byte、sort、int、long、float、double、boolean、String）其他所有类型都不不可以进行表述。并且只有标有 “final” 修饰符的字段并且该字段赋有初值时这个参数才会有值。例如类：

public class TestBean {

private final String data;

public TestBean() {

data = "aa";

}

....

在执行 “visitField” 方法时候，这个参数的就是 null 值，下面这种代码也会是 null 值：

public class TestBean {

private final Date data;

public TestBean() {

data =new Date();

}

....

此外如果字段使用的是基本类型的包装类型，诸如：Integer、Long...也会为空值：

public class TestBean {

private final Integer intData = 12;

...

能够正确得到默认值的代码应该是这个样子的：

public class TestBean {

private final String data = "ABC";

private final int intData = 12;

...

**4.visitMethod(int , String , String , String , String[])**    该方法是当扫描器扫描到类的方法时进行调用。下面是对方法中各个参数的示意：**visitMethod(修饰符 , 方法名 , 方法签名 , 泛型信息 , 抛出的异常)**。例如：

public class TestBean {

public int halloAop(String param) throws Throwable {

等价于：

visit(V1\_6, ACC\_PUBLIC | ACC\_SUPER , "org/more/test/asm/simple/TestBean",

null, "java/lang/Object", null)

visitMethod(ACC\_PUBLIC, "<init>", "()V", null, null)

visitMethod(ACC\_PUBLIC, "halloAop", "(Ljava/lang/String;)I", null, [java/lang/Throwable])

**第一个参数：**表示方法的修饰符，修饰符在 ASM 中是以 “ACC\_” 开头的常量进行定义。可以作用到方法级别上的修饰符有：ACC\_PUBLIC（public）、ACC\_PRIVATE（private）、ACC\_PROTECTED（protected）、ACC\_STATIC（static）、ACC\_FINAL（final）、ACC\_SYNCHRONIZED（同步的）、ACC\_VARARGS（不定参数个数的方法）、ACC\_NATIVE（native类型方法）、ACC\_ABSTRACT（抽象的）、ACC\_DEPRECATED（标记了@Deprecated注解的方法）、ACC\_STRICT、ACC\_SYNTHETIC。

**第二个参数：**表示方法名，在 ASM 中 “visitMethod” 方法会处理（构造方法、静态代码块、私有方法、受保护的方法、共有方法、native类型方法）。在这些范畴中构造方法的方法名为 “<init>”，静态代码块的方法名为 “<clinit>”。列如：

public class TestBean {

public int halloAop(String param) throws Throwable {

return 0;

}

static {

System.out.println();

}

...

等价于：

visit(V1\_6, ACC\_PUBLIC | ACC\_SUPER , "org/more/test/asm/simple/TestBean",

null, "java/lang/Object", null)

visitMethod(ACC\_PUBLIC, "<clinit>", "()V", null, null)

visitMethod(ACC\_PUBLIC, "<init>", "()V", null, null)

visitMethod(ACC\_PUBLIC, "halloAop", "(Ljava/lang/String;)I", null, [java/lang/Throwable])

**第三个参数：**表示方法签名，方法签名的格式如下：“(参数列表)返回值类型”。在字节码中不同的类型都有其对应的代码，如下所示：

"I" = int

"B" = byte

"C" = char

"D" = double

"F" = float

"J" = long

"S" = short

"Z" = boolean

"V" = void

"[...;" = 数组

"[[...;" = 二维数组

"[[[...;" = 三维数组

"L....;" = 引用类型

    下面是一些方法签名对应的方法参数列表。

|  |  |
| --- | --- |
| String[] | [Ljava/lang/String; |
| String[][] | [[Ljava/lang/String; |
| int, String, String, String, String[] | ILjava/lang/String;Ljava/lang/String;Ljava/lang/String;[Ljava/lang/String; |
| int, boolean, long, String[], double | IZJ[Ljava/lang/String;D |
| Class<?>, String, Object... paramType | Ljava/lang/Class;Ljava/lang/String;[Ljava/lang/Object; |
| int[] | [I |

**第四个参数：**凡是具有泛型信息的方法，该参数都会有值。并且该值的内容信息基本等于第三个参数的拷贝，只不过不同的是泛型参数被特殊标记出来。例如：

public class TestBean<T, V extends List> {

public T halloAop(V abc, int aaa) throws Throwable {

方法签名：(Ljava/util/List;I)Ljava/lang/Object;

泛型签名：(TV;I)TT;

public class TestBean<T, V extends List> {

public String halloAop(V abc, int aaa) throws Throwable {

方法签名：(Ljava/util/List;I)Ljava/lang/String;

泛型签名：(TV;I)Ljava/lang/String;

    可以看出泛型信息中用于标识泛型类型的结构是（“T” + 泛型名 + “;”），还有一种情况就是。泛型是声明在方法上。例如：

public class TestBean {

public <T extends List> String halloAop(T abc, int aaa) throws Throwable {

方法签名：(Ljava/util/List;I)Ljava/lang/String;

泛型签名：<T::Ljava/util/List;>(TT;I)Ljava/lang/String; //泛型类型基于接口

public class TestBean {

public <T> String halloAop(T abc, int aaa) throws Throwable {

方法签名：(Ljava/lang/Object;I)Ljava/lang/String;

泛型签名：<T:Ljava/lang/Object;>(TT;I)Ljava/lang/String; //泛型类型基于类型

**第五个参数：**用来表示将会抛出的异常，如果方法不会抛出异常。则该参数为空。这个参数的表述形式比较简单，举一个例子：

public class TestBean {

public <T> String halloAop(T abc, int aaa) throws Throwable,Exception {

异常参数为：[java/lang/Throwable, java/lang/Exception]

**5.visitEnd()**    该方法是当扫描器完成类扫描时才会调用，如果想在类中追加某些方法。可以在该方法中实现。在后续文章中我们会用到这个方法。

    提示：ACC\_SYNTHETIC、ACC\_STRICT：这两个修饰符我也不清楚具体功能含义是什么，如果有知道的朋友还望补充说明。

    下一篇文章将讲解，使用 ASM 如何实一个 Aop 现代理类，届时将不在详细讲解 ClassVisitor 类中各个方法的作用。