ASMSupport使用手册

*——java字节码操作类库*

版本：0.4

作者:温斯群

第1章 关于ASMSupport

1.1 简介

ASMSupport是一个java的字节码操作类库，底层采用的是ASM。那么相比于直接是用ASM，ASMSupport的优势在于它让开发者不需要了解底层指令的情况下能够快速的开发出动态生成字节码的业务。通过实践归结出以下几个asmsupport的优势：

1. 屏蔽了JVM底层指令。
2. 自动堆栈管理，无需像使用asm一般操作堆栈。
3. 采用匿名类的方式模拟java代码的各程序块，使得变量操作更安全。
4. 使用简单，上手快，可以快速的创建class

1.2 Maven·

<dependency>

<groupId>cn.wensiqun</groupId>

<artifactId>asmsupport</artifactId>

<version>0.4</version>

</dependency>

第2章 从Hello World开始

2.1 第一个ASMSupport程序

在介绍ASMSupport的一大篇长篇大论之前，先喝口咖啡，通过一个Hello World实例来感受下ASMSupport是如何创建一个类的。虽然你现在还不明白是如何使用asmsupport，但是Hello World嘛，你懂的。

先看下我们想要生成的代码具体是什么样的：

**public** **class** HelloWorldExample

{

**public** **static** **void** main(String[] args)

{

System.*out*.println("Hello World");

}

}

代码2-1

很简单的一段代码. 首先分析下操作流程，和我们常规编写代码一样，首先我需创建一个类，那么我们就需要一个类创建器，在ASMSupport中他是cn.wensiqun.asmsupport.creator.ClassCreator，我们通过new操作获取一个类创建器；随之而来的就是创建一个main方法，由于main方法是一个static类型的，那么我们将会调用他的createStaticMethod方法来帮助我们创建main方法；光有方法名没有方法的实际内容可不行，如果没有方法体，那么就是创建一个接口来，而这里我们还希望能够打印出Hello World呢，所以在调用createStaticMethod的时候我们会传递一个方法体对象cn.wensiqun.asmsupport.block.method.common.StaticMethodBody，这个对象才是核心，我们需要通过匿名类的方式来实现具体的方法体中的内容。那么如何使用ASMSupport去实现呢，代码送上：

ClassCreator creator = **new** ClassCreator(

Opcodes.*V1\_5*, Opcodes.*ACC\_PUBLIC* ,

"generated.helloworld.HelloWorldExample", **null**, **null**);

creator.createStaticMethod("main",

**new** AClass[]{AClassFactory.*getProductClass*(String[].**class**)},

**new** String[]{"args"}, **null**, **null**,

Opcodes.*ACC\_PUBLIC* + Opcodes.*ACC\_STATIC*,

**new** StaticMethodBody(){

@Override

**public** **void** generateBody(LocalVariable... argus) {

invoke(*systemOut*, "println", Value.*value*("Hello World"));

//don't forget return.

runReturn();

}

});

代码2-2

2.2 原理分析

上面我们分析过ASMSupport的操作流程，接下来我们从字节码层面上分析下ASMSupport的实现原理。首先我们先来看看上面main方法中的内容在字节码层面上是如何表示的：

GETSTATIC System.out

LDC "HelloWorld"

INVOKEIRTUAL PrintStream.println(String): void

RETURN

代码2-3

首先是获取一个System的一个静态成员变量out.然后再将”Hello World”压入操作栈，再调用out的println方法，并且将栈顶的”Hello World”传入println方法，最后再调用一个RETURN操作跳出。整个过程很简单，而前面三条指令在我们代码中表现的行为其实就是一句话System.out.println("Hello World!"),而后一条指令RETURN实际上就是main方法中没有显示的写明的return操作。所以在ASMSupport中为了更符合代码编写的习惯，将前三条指令封装成一个类cn.wensiqun.asmsupport.operators.method.StaticMethodInvo-ker。RETURN指令也封装成一个类cn.wensiqun.asmsupport.operators.Return.然后将这两个类放到当前方法的一个执行队列中。然后按照顺序执行队列中的各个对象，进而调用底层的ASM原始字节码操作实现方法中字节码的动态编写。下图能够简单描述这一过程。

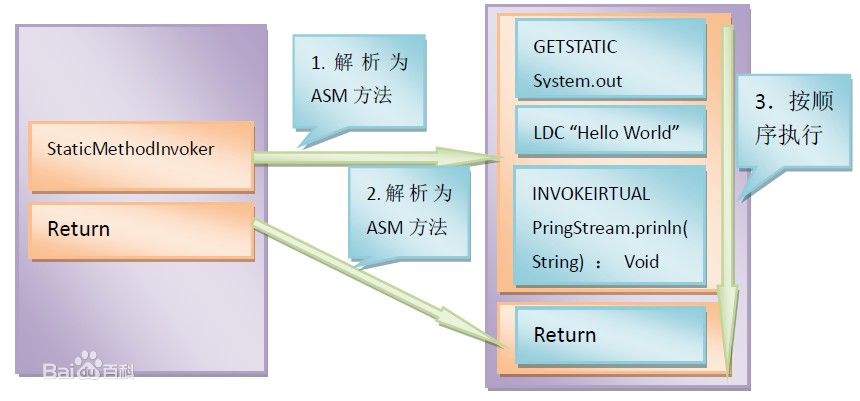


图2-1

第3章 概念篇

这一章主要介绍ASMSupport中的常用类和包结构。是一个比较枯燥的章节，可以将这一章当作是一个工具篇，在实际开发过程中遇到不明白的类能在此章找到相对应的解决方法。如果是初学者也可以在粗略了解本章中介绍的AClass,Value, ClassCreator, ProgramBlock之后直接进入第三章实战篇，在实战篇遇到不明白的类之后进入本章节再细节上了解该类。

3.1 cn.wensiqun.asmsupport包

asmsupport所有的类都在这个包或者其子包下。那么下面我们看看这个包下的结构：

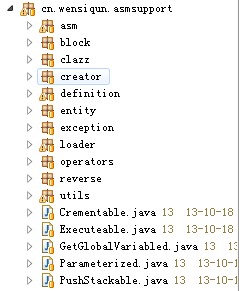


图3-1

在这里我们主要简单的介绍主要的几个包的定义。

* clazz包：asmsupport是模拟我们平时编写java的方式来实现的，所以在asmsupport中最基本的一个内容就是对class的描述。而这个包下就是存放着asmsupport对class的描述的类，包括了AClass,ArrayClass等等，将在后面详细介绍。
* definition包：这个包下我们定义了使用asmsupport的一些基本组件，包括方法，变量以及常量值等等。
* block包：此包下是我们对程序块的抽象，比如方法块，静态方法块，try...catch程序块，条件控制程序块等等。
* creator包：这个包我们定义了类创建器，类修改器，枚举类型创建器，方法创建器等等，这是我们使用asmsupport的入口。
* operators包：这个包下我们将所有的java操作都进行了抽象，比如算数运算，条件运算，方法调用，赋值等等。
* asm包：这个包下主要是存放一些asm底层操作的辅助类以及asm 3.3.1版本转向4.2版本的适配器，作为asmsupport的使用者可以不用了解这个包
* utils包：这里主要就是asmsupport的一些工具类。

3.2 clazz包

在我们平时编写java的时候我们有使用各种类，对于一般所使用的class我们将其抽象成cn.wensiqun.asmsupport.clazz.AClass。当然AClass仅仅是一个通用的类，在asmsupport中我们将其细分了如下一下类，如下图：

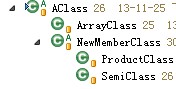


图3-2

我们可以看到AClass只是作为一个父类，其子类下还有ArrayClass，这是对数组类的抽象。NewMemberClass表示这个类是可以通过asmsupport添加新的成员变量或者方法的，也可以被修改的，正常情况下，一些已存在的类是能够被修改的。比如我们需要动态修改某个类的时候，而这些已经存在的类我们称之为ProductClass，叫做成品类。被classloader加载过的类都是ProductClass类，与之对应的就是SemiClass，称之为半成品类，这就是我们使用asmsupport创建出来的类，当这个类被创建之后再次使用的时候就变成ProductClass类了。下面我们将介绍每个类。

3.2.1 AClass

1. **public** **abstract** **boolean** isArray() ：判断当前类是否是一个数组，这是一个抽象方法。返回类型是boolean类型，true表示是一个数组类。
2. **public** **abstract** **int** getDimension() ：如果当前类是个数组则返回这个数组类的维度，如果不是一个数组类则返回的是-1.
3. **public** String getPackage() ： 获取当前类的包名，返回的是个string类型。
4. **public** String getName() ：获取当前类的全名，包括了包名。
5. **public** **int** getModifiers() ：获取当前类的修饰符的值。可参考org.objectweb.asm.Opcodes类。
6. **public** **int** getVersion() ：获取当前被创建的类所使用的asm的版本。这个方法只适用于SemiClass类中。
7. **public** Class<?> getSuperClass() ： 获取当前类的父类。
8. **public** Class<?>[] getInterfaces() ：获取当前类的接口。
9. **public** **abstract** String getDescription() ： 获取当前类的在字节码层面上的描述，这是个抽象类，在其子类中实现。
10. **public** **boolean** isChildOrEqual(AClass cls) ： 判断当前类是否与传入的类相等或者是其子类.
11. **public** Type getType() ： 获取当前类的Type，Type是ASM的特有类。
12. **public** **abstract** GlobalVariableEntity getGlobalVariableEntity(-String name) : 获取当前类中的全局静态变量的Entity。
13. **public** **final** MethodEntity availableConstructor(AClass aclass, AClass[] parameterTypes) : 获取当前类中可调用的构造方法。由于访问控制修饰符，在不同的类中可能无法调用当前类中某个构造方法。这个方法就是可访问的构造方法，第一个参数表示在哪个类中调用构造方法，第二个参数表示构造方法的参数类型列表。
14. **public** **final** MethodEntity availableMethod(AClass where, String name, AClass[] parameterTypes) ： 获取可访问的方法。效果和之availableConstructor是一样的，第一个参数表示在哪个类中调用构造方法，第二个参数表示方法名，第二个参数表示构造方法的参数类型列表。
15. **public** MethodEntity getSuperMethod(String methodName, AClass[] parameterTypes) ： 获取可访问的父类中的方法，第一个参数表示方法名，第二个参数表示调用方法的参数类型列表。
16. **public** MethodEntity getSuperConstructor(AClass[] paraTypes) ： 获取可访问的父类中的构造方法。参数表示调用方法的参数类型列表。
17. **public** **abstract** **boolean** isPrimitive() : 判断当前类是否是基本类型。
18. **public** **final** **boolean** isInterface() ： 判断当前类是否是接口类型。
19. **public** **final** **boolean** isAbstract() ： 判断当前类是否是抽象类。
20. **public** **abstract** **int** getCastOrder() : 获取当前类的强制转换顺序。
21. **public** **final** Value getDefaultValue() ： 获取当前类的默认值，比如Object则返回的null，int类型则返回0. 但是其返回类型是Value。这是asmsupport对常量的特有封装，具体Value将在以后介绍。
22. **public** **abstract** **boolean** existStaticInitBlock() ： 获取当前类是否已经有了静态程序块。

在这里我们介绍下上面方法中的一些概念，首先是getDescription方法，上面介绍的是“获取当前类的在字节码层面上的描述”，那么第一个概念就是这个description。

这个概念其实是class格式中(可参考[jvms4.3](http://www.wensiqun.com/jvms/se6/ClassFile.doc.html#1169))的一个小的部分，它其实是用一个字符串的方式去表示一个字段或者方法的类型，而这种表示方式就是jvm层面所认可的。下面个表格就是表示各种不同类型的description。

|  |  |
| --- | --- |
| Java 类型 | Description |
| 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; |

从这个表格我们可以看到对应对象类型的description其实是将对象的全名中的”.”改成了”/”同时在前面加上大写的”L”,在后面加上分号”;”。而对于数字类型就是在类型的description前面加上”[”. 多少维就加多少个”[”。

现在开始介绍第二个概念getModifiers，这里的modifiers是java中的修饰符，比如public，static。那么这里我们注意到这个方法返回的是一个int类型。这是因为jvm是通过按位与的方式来判断包括了哪些修饰符。假设public的修饰符的2进制的值是1, static的二进制值是10. 那么如果某个方法的修饰符是public static. 那么他的modifier的值就是11. 这时候我如何判断11是否是包含了public修饰符呢，只要将11按位和1与得到值1. 这是不等于0的说明包括了public修饰符，同样11和10与得到10也是非0值说明也是包含了static修饰符。

接下来是第三个概念就是getCastOrder方法，那什么是CastOrder呢。根据字面意思就是强制转换的顺序，我们知道在我们编写java的时候有些基本类型java编译器会为我们自动进行类型转换。比如：”int i=10;double=i”，这段代码java虚拟机就会将i转换成double类型。那么我们如何判断是否可以自动类型转换呢。就是靠这个CastOrder。顺序低的可以转换成顺序高的，比如上面的例子中int类型的顺序是4，而double类型的顺序是7，当然这是对于基本类型来说的，对于数组类型和对象类型就不是这个逻辑的，但是这两种类型同样也是有CastOrder的。

3.2.1 ArrayClass

ArrayClass是ASMSupport对java中数组类型的一个封装,比如int[], String[][]。对应的在ArrayClass中我们也需要保存数组的相关的信息，比如数组的维度，数组的基本类型等等。接下来我们看下ArrayClass代码, 大部分的方法都定义在了父类AClass中，所以这里我们只显示ArrayClass部分内容：

**public** **class** ArrayClass **extends** AClass {

**private** AClass aclass;

**private** String desc;

**protected** **int** dim;

ArrayClass(AClass cls, **int** dim) {

...

}

@Override

**public** **boolean** existStaticInitBlock() {

**return** **false**;

}

@Override

**public** String getDescription() {

**return** desc;

}

@Override

**public** GlobalVariable getGlobalVariable(String name) {

**return** aclass.getGlobalVariable(name);

}

@Override

**public** **boolean** isArray() {

**return** **true**;

}

@Override

**public** **int** getDimension() {

**return** dim;

}

@Override

**public** GlobalVariableEntity getGlobalVariableEntity(String name) {

**return** aclass.getGlobalVariableEntity(name);

}

@Override

**public** **boolean** isPrimitive() {

**return** **false**;

}

@Override

**public** **int** getCastOrder() {

**return** 9;

}

**public** AClass getNextDimType(){

**if**(dim > 1){

**return** **new** ArrayClass(aclass, dim - 1);

}**else**{

**return** aclass;

}

}

**public** AClass getRootComponentClass(){

**return** aclass;

}

@Override

**public** **boolean** isChildOrEqual(AClass cls) {

**if**(cls **instanceof** ArrayClass){

String clsName = cls.getName();

**if** (getName().equals(clsName)) {

**return** **true**;

}

}

**if**(cls **instanceof** ProductClass){

**if**(((ProductClass)cls).getReallyClass().equals(Object.**class**)){

**return** **true**;

}

}

**return** **false**;

}

}

代码3-1

通过上面的代码我们能看到的在ArrayClass中我们加入了aclass，desc，dim三个属性，分表表示这个数组类型的最基本类型，描述以及这个数组的维度。接下来我们来介绍每个方法。

1. **public** **boolean** existStaticInitBlock() : 这个重写的父类方法，由于数组类型是运行时的类型，他是没有静态程序块的，这里直接返回fasle
2. **public** **boolean** isArray() : 重写父类方法，返回true。
3. **public** **int** getDimension() ： 获取当前数组类型维度。
4. **public** AClass getNextDimType() ： 获取下一维度的AClass。比如当前的ArrayClass表示的是int[][]，那么下一个维度就是int[]的ArrayClass。如果当前的ArrayClass表示，那么下一维度就是int类型的ProductClass。
5. **public** AClass getRootComponentClass() : 获取基本类型.比如当前ArrayClass表示的是int[][]，那么通过这个方法获取的就是int类型的AClassProduct。
6. **public** **boolean** isChildOrEqual(AClass cls) ：这个方法也是重写父类的方法，需要注意的是数组类型仅仅值是Object类型的子类。所以至于传入Object类型的AClass或者是和当前ArrayClass相同的类型这个方法才返回的是true。

3.2.1 NewMemberClass

在ASMSupport中我们不仅能够在运行时新创建一个类，而且还能够修改一个类。这些类都将继承与一个类就是NewMemberClass表示这个AClass是我们新创建的或者能够被修改的。比如我们上面讲的ArrayClass就不能修改，所以它不继承自这个类。

如果我们希望能够动态的编写一个类，那么我们肯定希望为这个class添加一个或修改一个方法。那么在方法中我们或多或少的会用到this和super这两个关键字。所以在NewMemberClass中我们需要添加两个成员变量分别是thisVariable和superVariable来表示this和super。这两个属性的类型分别是ThisVariable和SuperVariable，这两种类型将在后面介绍。

如果我们修改或者新创建一个类，必然会为这个类添加方法和成员变量。我们用两个List来保存我们添加的方法和成员变量分别是methods，globalVariables。相应的还有constructors，这个存储的是新添加的构造方法。下面我们来看下我们需要添加哪些成员变量：

**private** SuperVariable superVariable;

**private** ThisVariable thisVariable;

**private** List<AMethod> methods;

**private** List<AMethod> bridgeMethod;

**private** List<AMethod> constructors;

**private** AMethod staticBlock;

**private** List<GlobalVariable> globalVariables;

**private** **int** enumNum;

代码3-2

通过上面的代码我们发现我们这里还添加了staticBlock，bridgeMethod和enumNum。下面来继续介绍这几个成员变量：

1. staticBlock：如果我们修改或者新建的class中需要有静态程序块，那么这个程序块将存储在这个成员变量中
2. bridgeMethod ：这个地方存储的是拥有bridge修饰符的方法。
3. enumNum ：只有我们在创建一个新枚举类的时候这个属性才有效。

这里我们接触到两个概念：1.bridge方法，2.enumNum，首先介绍bridge方法：

bridge是在JVM层面的一个修饰符，我们在编写java代码的时候并不需要使用它。在java字节码层面中每个方法都有各自的一个签名，它的具体格式这里不做详细介绍，这里只要知道这个签名是由方法返回类型，方法参数类型列表组成的就可以了（比如方法”public String test(int I, boolean b)”，它的签名就是”(IZ)Ljava/lang/String;”），也就是这样的一个签名是java虚拟机判断方法是否唯一的标准。这样问题就来了，当我们编写java代码的时候我们可以重写父类的一个方法，并且让其返回值是父类方法中返回值的子类。我们用java实例来描述这种情况：

存在一个类A，它有一个子类B。现在父类中有一个方法public A test()。这时候我们在子类重写这个方法public B test()，这种情况是允许的。但是这样的话我们重写的方法的签名和父类就不一样了(父类中的是：“()LA”，子类中的是“()LB”)，这样的话由于签名不同JVM不认为它是重写的一个方法。但是为什么我们在真实编写java代码的时候这中情况可以编译通过并且正常运行呢。通过查看子类的class文件我们能看到Java 编译器自动生成一个方法名方法参数和“public B test()”方法相同的一个方法，这个方法有个修饰符bridge, 并且返回类型和父类中的方法是一样的。我们用代码来描述下这个bridge方法

public bridge A test(){

return this.test();//调用public B test()

}

终于知道了，在这个桥接方法中调用我们之前重写的方法起一个桥梁作用就解决了签名不同的问题了。

--------------------------------------------------------------------------------------------------------------------------------------------------

接着我们介绍第二个概念enumNum，称之为枚举类型计数器。这个属性只有我们在新建一个枚举类型的class时候才起作用，我们知道所有的枚举类型都是java.lang.Enum的一个子类，而java.lang.Enum拥有一个构造方法protected Enum(String name, int ordinal). 我们创建一个枚举类型class的时候枚举类型的每一个值都是通过这个构造方法来创建的，每一个值的名称被转换成一个字符串也就是构造方法的第一个参数，而ordinal就是这个枚举类型的被创建的顺序数。而enumNum就是顺序数的计数器，这样asmsupport在为每一个枚举类型添加常量值的时候就可以根据enumNum来获得ordinal，我们通过创建一个枚举类class以及分析这个class的字节码来解释着一个概念：

Java代码:

**public** **enum** EnumNumber {

*ONE*,*TWO*,*THREE*

}

部分字节码内容:

public final enum EnumNumber {

public static final enum EnumNumber ONE;

public static final enum EnumNumber TWO;

public static final enum EnumNumber THREE;

private static final synthetic EnumNumber[] ENUM$VALUES;

static {};

0 new EnumNumber [1]

3 dup

4 ldc <String "ONE"> [14]

6 iconst\_0

7 invokespecial EnumNumber(java.lang.String, int) [15]

10 putstatic EnumNumber.ONE : EnumNumber [19]

13 new EnumNumber [1]

16 dup

17 ldc <String "TWO"> [21]

19 iconst\_1

20 invokespecial EnumNumber(java.lang.String, int) [15]

23 putstatic EnumNumber.TWO : EnumNumber [22]

26 new EnumNumber [1]

29 dup

30 ldc <String "THREE"> [24]

32 iconst\_2

33 invokespecial EnumNumber(java.lang.String, int) [15]

36 putstatic EnumNumber.THREE : EnumNumber [25]

39 iconst\_3

40 anewarray EnumNumber [1]

43 dup

44 iconst\_0

45 getstatic EnumNumber.ONE : EnumNumber [19]

48 aastore

49 dup

50 iconst\_1

51 getstatic EnumNumber.TWO : EnumNumber [22]

54 aastore

55 dup

56 iconst\_2

57 getstatic EnumNumber.THREE : EnumNumber [25]

60 aastore

61 putstatic EnumNumber.ENUM$VALUES : EnumNumber[] [27]

64 return

通过这段字节码可以看出每一个枚举类型常量实际上是一个静态成员变量，而这些成员变量的初始化是在static块中完成的。Ldc<String “ONE”>就是将常量名作为参数传入构造方法，iconst\_1就是传入序号1告诉JVM我是第一个被创建的，而enumNum的任务就是统计已经创建了多少个常量了，这样下次在创建的时候只需要传入enumNum+1就可以了。