Java Class 详解

基于栈和基于寄存器指令区别?

什么是直接引用和间接引用?

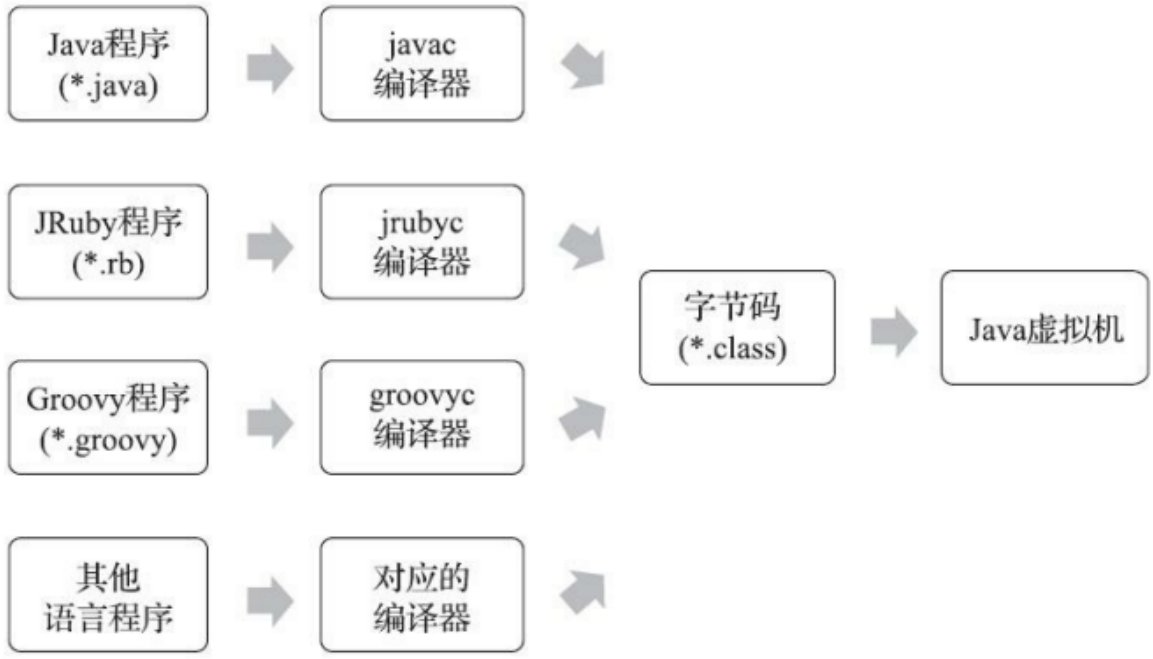
class文件怎么来的?

apt与AMS字节码插桩?

第一节 Class 文件介绍

1.1 背景

“计算机只认识0和1，所以我们写的程序需要被编译 器翻译成由0和1构成的二进制格式才能被计算机执行。”十多年过去了，今天的计算机仍然只能识别0和1，但由于最近十年内虚拟机以及大量建立在虚拟机之上的程序语言如雨后春笋般出现并蓬勃发展，把我们编写的程序编译成二进制本地机器码（Native Code）已不再是唯一的选择，越来越多的程序语言选择了与操作系统和机器指令集无关的、平台中立的格式作为程序编译后的存储格式。



Java 语言之所以能实现一次编译到处运行，就是因为使用所有平台都支持的字节码格式

第二节 Class类文件的结构

2.1 class文件格式

一个class文件是由下图描述出来的。我们可以按这张表的格式去解释一个class文件。

类　型	名　称	数　量
u4	magic	1
u2	minor_version	1
u2	major_version	1
u2	constant_pool_count	1
cp_info	constant_pool	constant_pool_count-1
u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count
u2	fields_count	1
field_info	fields	fields_count
u2	methods_count	1
method_info	methods	methods_count
u2	attributes_count	1
attribute_info	attributes	attributes_count

- 以u1、u2、u4、u8来分别代表1个字节、2个字节、4个字节和8个字节的无符号数，无符号数可以用来描述数字、索引引用、数量值或者按照UTF-8编码构成字符串值。

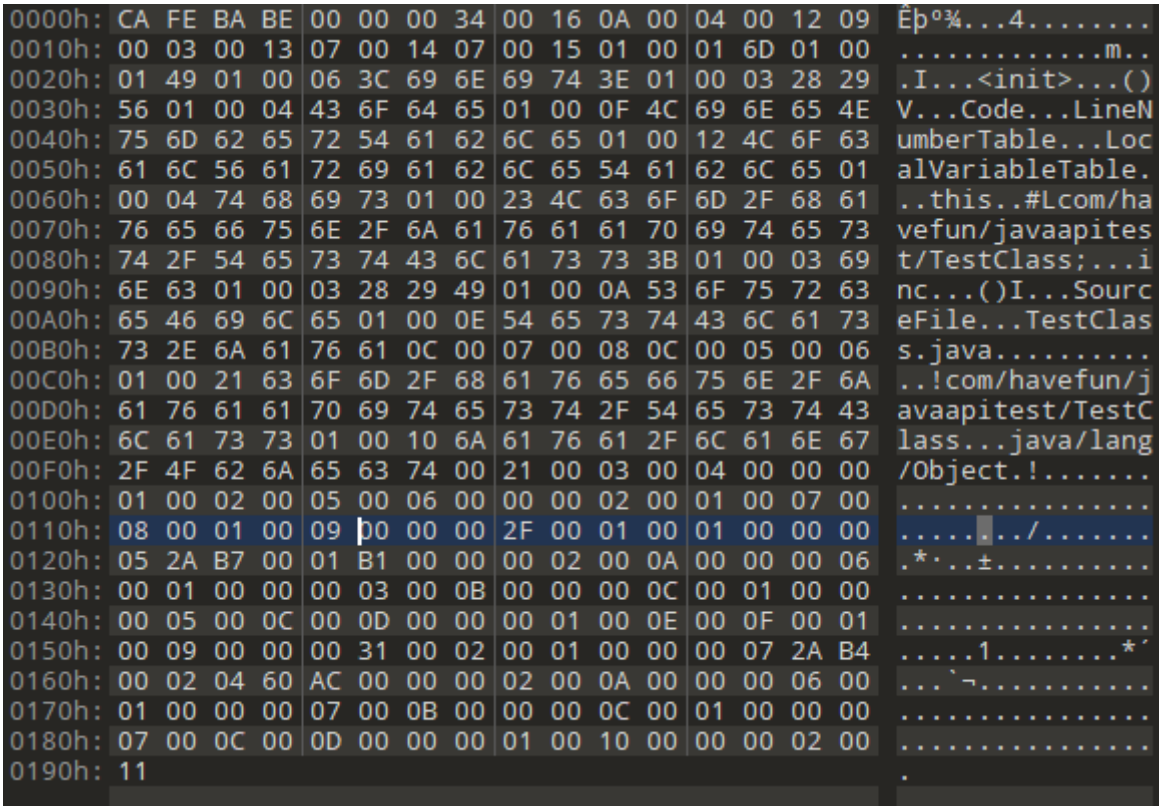
接下来我们用这一小段样本代码来说明class文件的具体内容。再复杂的java源文件都是可以通过这样的方式分析出来。

```java
public class TestClass {
    private int m;

    public int inc() {
        return m + 1;
    }
}
```

我们将上面的代码用编译器进行编译得到一个TestClass.class文件。通过Windows工具“010Editor”对这个class文件进行阅读。

下面是010Editor上面class二进制内容：



0A FE BA BE : 魔数（它的唯一作用是确定这个文件是否为一个能被虚拟机接受的Class文件）

---

00 00 00 34 : 次版本号与主版本号 次版本号为0，主版本号为52（只能被jdk1.1~1.8 识别）

class主版本与jdk版本关系（部分）

JDK 6	不带（默认为 -target 6 ）	1.1～6	50.0
JDK 7	不带（默认为 -target 7 ）	1.1～7	51.0
JDK 8	不带（默认为 -target 8 ）	1.1～8	52.0
JDK 9	不带（默认为 -target 9 ）	6～9 ⊖	53.0
JDK 10	不带（默认为 -target 10 ）	6～10	54.0
JDK 11	不带（默认为 -target 11 ）	6～11	55.0
JDK 12	不带（默认为 -target 12 ）	6～12	56.0
JDK 13	不带（默认为 -target 13 ）	6～13	57.0

2.2 常量池

00 16 : 常量池数量 22，索引是1-21，

为什么常量池的索引不从0开始?

如果后面某些指向常量池的索引值的数据在特定情况下需要表达“不引用任何一个常量池项目”的含义，可以把索引值设置为0来表示。

CONSTANT_Utf8_info	1	UTF-8 编码的字符串
CONSTANT_Integer_info	3	整型字面量
CONSTANT_Float_info	4	浮点型字面量
CONSTANT_Long_info	5	长整型字面量
CONSTANT_Double_info	6	双精度浮点型字面量
CONSTANT_Class_info	7	类或接口的符号引用
CONSTANT_String_info	8	字符串类型字面量
CONSTANT_Fieldref_info	9	字段的符号引用
CONSTANT_Methodref_info	10	类中方法的符号引用
CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
CONSTANT_NameAndType_info	12	字段或方法的部分符号引用
CONSTANT_MethodHandle_info	15	表示方法句柄
CONSTANT_MethodType_info	16	表示方法类型
CONSTANT_Dynamic_info	17	表示一个动态计算常量

（续）

类　　型	标　　志	描　　述
CONSTANT_InvokeDynamic_info	18	表示一个动态方法调用点
CONSTANT_Module_info	19	表示一个模块
CONSTANT_Package_info	20	表示一个模块中开放或者导出的包

0A 00 04 00 12(常量索引：1):

0A: -> 10 通过查表 表示一个Methodref_info

CONSTANT_Methodref_info	tag	u1	值为 10
	index	u2	指向声明方法的类描述符 CONSTANT_Class_info 的索引项
	index	u2	指向名称及类型描述符 CONSTANT_NameAndType 的索引项

04: 找到索引为4的常量 -> java/lang/Object

12 转十进制得到18， 这里找到常量池里18的常量代表 ()V

得到结果： java/lang/Object () V

---

09 00 03 00 13(常量索引：2):

09: -> 09 表示一个Fieldref_info

CONSTANT_Fieldref_info	tag	u1	值为 9
	index	u2	指向声明字段的类或者接口描述符 CONSTANT_Class_info 的索引项
	index	u2	指向字段描述符 CONSTANT_NameAndType 的索引项

最终得到：com/havefun/javaapitest/TestClass 和 m i

---

07 00 14(常量索引：3):

CONSTANT_Class_info	tag	u1	值为 7
	index	u2	指向全限定名常量项的索引

最终结果： com/havefun/javaapitest/TestClass

---

07 00 15(常量索引：4):

07 表示类信息

15-> 21 是在常量的索引 -> java/lang/Object

---

01 00 01 6D(常量索引：5)： m

CONSTANT_Utf8_info	tag	u1	值为 1
	length	u2	UTF-8 编码的字符串占用了字节数
	bytes	u1	长度为 length 的 UTF-8 编码的字符串

---

01 00 01 49(常量索引：6): I

---

01 00 06 3C 69 6E 69 74 3E(常量索引：7):

---

01 00 03 28 29 56(常量索引：8): ()V

---

01 00 04 43 6F 64 65(常量索引：9):　　Code

---

01 00 0F 4C 69 6E 65 4E 75 6D 62 65 72 54 61 62 6C 65(常量索引：10): LineNumberTable

---

01 00 12 4C 6F 63 61 6C 56 61 72 69 61 62 6C 65 54 61 62 6C 65(常量索引：11):

LocalVariableTable

---

01 00 04 74 68 69 73(常量索引：12): ----> this

---

01 00 23 4C 63 6F 6D 2F 68 61 76 65 66 75 6E 2F 6A 61 \

76 61 61 70 6974 65 73 74 2F 54 65 73 74 43 6C 61 73 73 3B(常量索引：13):

Lcom/havefun/javaapitest/TestClass;

---

01 00 03 69 6E 63(常量索引：14): inc

---

01 00 03 28 29 49(常量索引：15):　　()I

---

01 00 0A 53 6F 75 72 63 65 46 69 6C 65(常量索引：16): SourceFile

---

01 00 0E 54 65 73 74 43 6C 61 73 73 2E 6A 61 76 61(常量索引：17): TestClass.java

---

0C 00 07 00 08(常量索引：18):

CONSTANT_NameAndType_info	tag	u1	值为 12
	index	u2	指向该字段或方法名称常量项的索引
	index	u2	指向该字段或方法描述符常量项的索引

0C 表示字段或方法的部分引用;

07 ->

05 -> ()V

最终得到： // "":()V

0C 00 05 00 06(常量索引：19):　最终得到： // m:I

---

01 00 21 63 6F 6D 2F 68 61 76 65 66 75 6E 2F 6A 61 \

76 61 61 70 69 74 95 73 74 2F 54 65 73 76 43 6C 61 73 73(常量索引：20):

最终得到：com/havefun/javaapitest/TestClass

---

01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74(常量索引：21):

最终得到：java/lang/Object

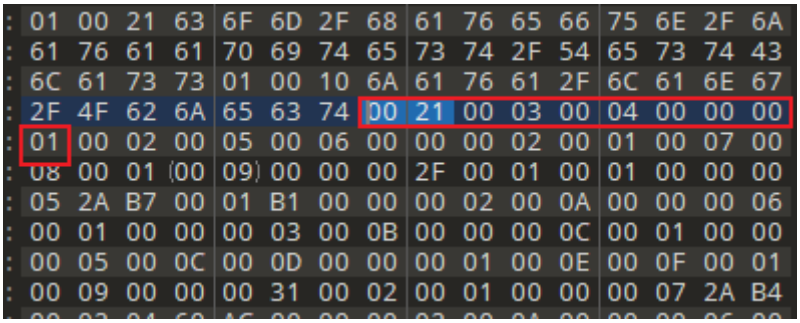
Javap -v 生成的内容，通过上面的分析就很容易看懂这个反编译过后的常量池要表达的内容了！

```
Constant pool:
   #1 = Methodref          #4.#18         // java/lang/Object."<init>":()V
   #2 = Fieldref           #3.#19         //
com/havefun/javaapitest/TestClass.m:I
   #3 = Class              #20            // com/havefun/javaapitest/TestClass
   #4 = Class              #21            // java/lang/Object
   #5 = Utf8               m
   #6 = Utf8               I
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               LocalVariableTable
  #12 = Utf8               this
  #13 = Utf8               Lcom/havefun/javaapitest/TestClass;
  #14 = Utf8               inc
  #15 = Utf8               ()I
  #16 = Utf8               SourceFile
  #17 = Utf8               TestClass.java
  #18 = NameAndType        #7:#8          // "<init>":()V
  #19 = NameAndType        #5:#6          // m:I
  #20 = Utf8               com/havefun/javaapitest/TestClass
  #21 = Utf8               java/lang/Object
```

访问标识符、类索引、父类索引与接口索引集合

下图是class文件结构表里面的一部分，描述了访问标识，类索引，父类索引与接口集合等。

u2	access_flags	1
u2	this_class	1
u2	super_class	1
u2	interfaces_count	1
u2	interfaces	interfaces_count



00 21: ACC_PUBLIC | ACC_SUPER

下面是截取的常量池部分内容，类索引和父类索引都能在上面找到。

```
#1 = Methodref          #4.#18         // java/lang/Object."<init>":()V
#2 = Fieldref           #3.#19         // com/havefun/javaapitest/TestClass.m:I
#3 = Class              #20            // com/havefun/javaapitest/TestClass
#4 = Class              #21            // java/lang/Object
```

00 03: 类索引-> 常量池索引3

00 04: 父类索引-> 常量池索引4

00 00: 接口数量0

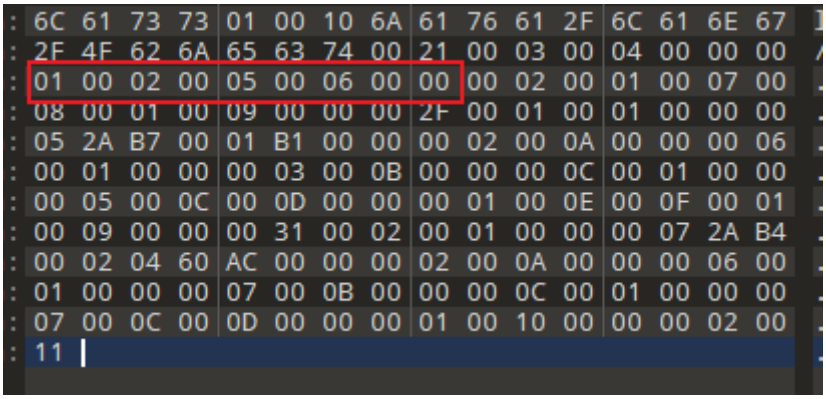
2.2 字段信息

字段表结构如下：

类　型	名　称	数　量	类　型	名　称	数　量
u2	access_flags	1	u2	attributes_count	1
u2	name_index	1	attribute_info	attributes	attributes_count
u2	descriptor_index	1			

字段访问标志：

标志名称	标志值	含　义	标志名称	标志值	含　义
ACC_PUBLIC	0x0001	字段是否 public	ACC_VOLATILE	0x0040	字段是否 volatile
ACC_PRIVATE	0x0002	字段是否 private	ACC_TRANSIENT	0x0080	字段是否 transient
ACC_PROTECTED	0x0004	字段是否 protected	ACC_SYNTHETIC	0x1000	字段是否由编译器自动产生
ACC_STATIC	0x0008	字段是否 static	ACC_ENUM	0x4000	字段是否 enum
ACC_FINAL	0x0010	字段是否 final			



字段表信息：

00 01：字段数量 1

通过字段表结构读取6个字节：00 02 00 05 00 06 00 00

00 02 访问描述符：代表了private

00 05 字段名称在常量池的索引：m

00 06 描述符在常量池的索引：I

00 00 属性数量为0

结合起来字段就很容易知道这个是 private 的int类型的字段m。

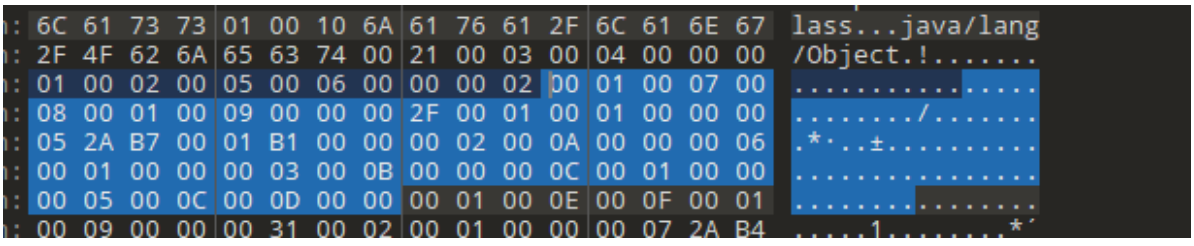
2.3 方法表信息

继续读class文件后面的内容：00 02 表示有两个方法。

方法表的结构：

类　型	名　称	数　量	类　型	名　称	数　量
u2	access_flags	1	u2	attributes_count	1
u2	name_index	1	attribute_info	attributes	attributes_count
u2	descriptor_index	1			

向后读方法表第一个方法：



00 01: 代表public方法　　00 07：方法名　　00 08：方法签名()V

上面这小部分可以得到如下信息：

```
public com.havefun.javaapitest.TestClass();
    descriptor: ()V
    flags: ACC_PUBLIC
```

00 01: 表示属性表有一个属性

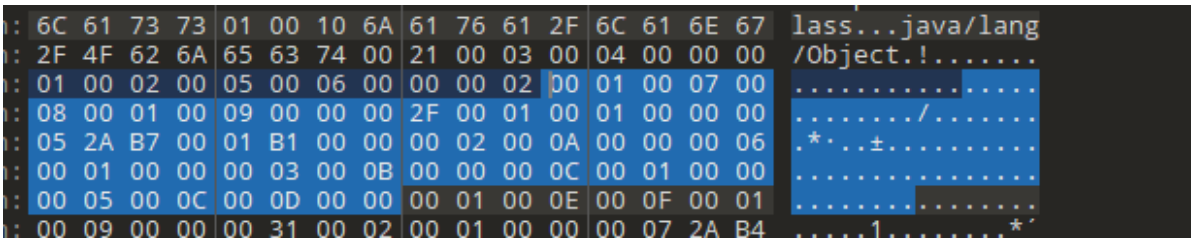
属性表结构：

类　　型	名　　称	数　　量
u2	attribute_name_index	1
u4	attribute_length	1
u1	info	attribute_length

00 09 00 00 00 2F: 通过常量池09表示Code（Code 的含义是Java代码编译成字节码的指令），后面4个字节表示接下来的属性长度，2F转十进制等于47。

Code对应的结构：

类　　型	名　　称	数　　量
u2	attribute_name_index	1
u4	attribute_length	1
u2	max_stack	1
u2	max_locals	1
u4	code_length	1
u1	code	code_length
u2	exception_table_length	1
exception_info	exception_table	exception_table_length
u2	attributes_count	1
attribute_info	attributes	attributes_count



接下来的字节码是：00 01 00 01 表示操作数栈最大深度为1；max_locals代表了局部变量表所需的存储空间；

再接下来4个字节：00 00 00 05（表示代码长度）；

再向后读5个字节表示代码：2A B7 00 01 B1；

- 2A：对应指令aload_0。是将第0个变量槽中为reference类型的本地变量推送到操作数栈顶。
- B7：指令为invokespecial。指令的作用是以栈顶的reference类型的数据所指向的对象作为方法接收者，调用此对象的实例构造器方法、private方法或者它的父类的方法。这个方法有一个u2类型的参数说明具体调用哪一个方法，它指向常量池中的一个CONSTANT_Methodref_info类型常量，即此方法的符号引用。

  这里 00 01 也就是代表了常量池里面#1号常量 =>（// java/lang/Object."":()V）这是一个构造方法；

  因为Java默认在每个方法插入一个默认参数this，并且放在变量槽0的位置。上面两条指令可以理解为 this = new Object()； 把这个this给实例化了。
- B1：对应指令为return

说明：这里一个字节表示一条指令操作，那么也就说明Java虚拟机最多不会超过256条指令；

00 00 ：异常表长度为0

00 02：属性列表数量为2；

那么上面可以得到如下信息：

```
Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1                  // Method java/lang/Object."
<init>":()V
         4: return
```

2.3.1 属性表信息

00 0A 00 00 00 06 00 01 00 00 00 03：

通过查表0A对应的常量池里面的：LineNumberTable；LineNumberTable属性用于描述Java源码行号与字节码行号（字节码的偏移量）之间的对应关系。00 00 00 06 表示属性长度为6个字节；00 01表示有一个line_number_table；00 00表示是字节码行号，00 03表示是Java源码行号

LineNumberTable对应的结构：

类　　型	名　　称	数　　量
u2	attribute_name_index	1
u4	attribute_length	1
u2	line_number_table_length	1
line_number_info	line_number_table	line_number_table_length

那么这可以得到如下信息：

```
LineNumberTable:
    line 3: 0
```

---

00 0B 00 00 00 0C 00 01 00 00 00 05 00 0C 00 0D 00 00：

通过常量池得到0B代表的是 LocalVariableTable，

LocalVariableTable的属性结构：

类　　型	名　　称	数　　量
u2	attribute_name_index	1
u4	attribute_length	1
u2	local_variable_table_length	1
local_variable_info	local_variable_table	local_variable_table_length

local_variable_info结构

类　型	名　称	数　量
u2	start_pc	1
u2	length	1
u2	name_index	1
u2	descriptor_index	1
u2	index	1

属性长度0C转十进制为12；00 01局部变量表长度为1；

00 00 00 05：表示start_pc和length属性分别代表了这个局部变量的生命周期开始的字节码偏移量及其作用范围覆盖的长度，两者结合起来就是这个局部变量在字节码之中的作用域范围。

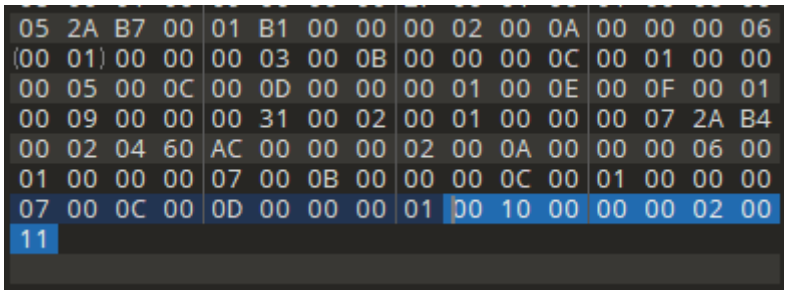
0C：在常量池查询是表示 this；0D：是这个变量的描述符对应的：
Lcom/havefun/javaapitest/TestClass;

最后的00 00表示：index是这个局部变量在栈帧的局部变量表中变量槽的位置。

通过上面这一小节可以得到如下信息：

```
LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0       5     0  this   Lcom/havefun/javaapitest/TestClass;
```

2.4 属性信息



SourceFile属性结构

类　型	名　称	数　量
u2	attribute_name_index	1
u4	attribute_length	1
u2	sourcefile_index	1

00 10：对应常量池的SourceFile　00 00 00 02：对应的属性长度为2；

作用：如果不生成这项属性，当抛出异常时，堆栈中将不会显示出错代码所属的文件名。

11：转十进制得到17，sourcefile_index数据项是指向常量池中CONSTANT_Utf8_info型常量的索引，常量值是源码文件的文件名。通过常量池得知17对应常量为：TestClass.java

第三节 基于栈指令简介

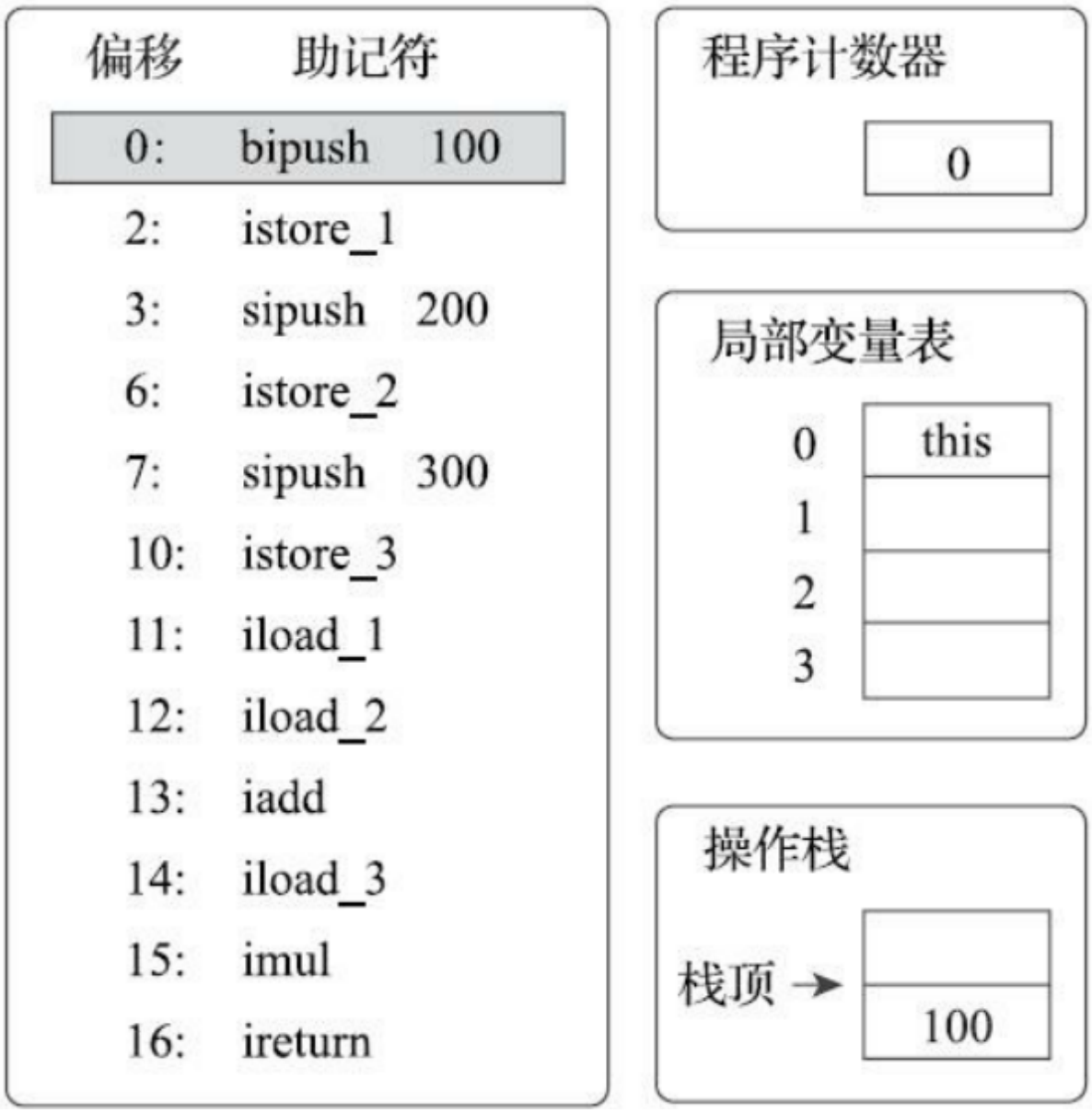
3.1 基于栈的解释器执行过程

以一段代码作为例子说明演示字节码执行过程

```
public int calc() {
    int a = 100;
    int b = 200;
    int c = 300;
    return (a + b) * c;
}
```

编译成字节码指令如下：

```
 public int calc();
    descriptor: ()I
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=4, args_size=1
         0: bipush        100   // 将100 推到操作数栈
         2: istore_1            // 将操作数栈顶的整型值出栈并存放到第1个局部变量槽中
         3: sipush        200   // 将200 推到操作数栈
         6: istore_2            // 将操作数栈顶的整型值出栈并存放到第2个局部变量槽中
         7: sipush        300   // 将300 推到操作数栈
        10: istore_3            // 将操作数栈顶的整型值出栈并存放到第3个局部变量槽中
        11: iload_1             // 将局部变量槽1的变量放入操作数栈
        12: iload_2             // 将局部变量槽2的变量放入操作数栈
        13: iadd                // 将操作数栈中头两个栈顶元素出栈，做整型加法，然后把结果
重新入栈
        14: iload_3             // 将局部变量槽3的变量放入操作数栈
        15: imul                // 将操作数栈中头两个栈顶元素出栈，做整型乘法，然后把结果
重新入栈
        16: ireturn             // 将结束方法执行并将操作数栈顶 的整型值返回给该方法的调用
者
```



3.2 基于栈与基于寄存器指令集区别?

以同样的1+1这个计算来进行举例

基于栈的指令集如下:

```
iconst_1
iconst_1
iadd
istore_0
```

基于寄存器指令集如下:

```
mov eax, 1
add eax, 1
```

这两种指令集的优势与劣势:

- 基于栈的指令集主要优点是可移植

- 基于寄存器的指令会比基于栈的指令少，但是每条指令会边长。
- 基于栈指令集的主要缺点是理论上执行速度相对来说会稍慢一些

结尾

class文件的结构分析就到这里了，通过一个简单的类去探索编译器如何实现类的编写，那么再复杂的类我们也能一步一步分析出来，只是需要我们更加细心，我们了解了这些文件的生成过程，个人认为有如下好处：

- 知道javap -v 反编译class文件的输出类容到底是怎么来的。
- class文件怎么描述一个Java方法或者一个变量

运用方向比如字节码增强，动态修改或者生成等都是能够实现的。

参考书籍

《深入理解Java虚拟机》