http://ifeve.com/javacode2bytecode/

英文原文地址

http://blog.jamesdbloom.com/JavaCodeToByteCode_PartOne.html#variables

明白Java代码是如何编译成字节码并在JVM上运行的非常重要，这有助于理解程序运行的时候究竟发生了些什么。理解这点不仅能搞清语言特性是如何实现的，并且在做方案讨论的时候能清楚相应的副作用及权衡利弊。

本文介绍了Java代码是如何编译成字节码并在JVM上执行的。想了解JVM的内部结构以及字节码运行时用到的各个内存区域，可以看下我前面的一篇关于JVM内部细节的文章。

本文分为三部分，每一部分都分成几个小节。每个小节都可以单独阅读，不过由于一些概念是逐步建立起来的，如果你依次阅读完所有章节会更简单一些。每一节都会覆盖到Java代码中的不同结构，并详细介绍了它们是如何编译并执行的。

1. 第一部分， 基础概念

变量

局部变量

JVM是一个基于栈的架构。方法执行的时候（包括main方法），在栈上会分配一个新的帧，这个栈帧包含一组局部变量。这组局部变量包含了方法运行过程中用到的所有变量，包括this引用，所有的方法参数，以及其它局部定义的变量。对于类方法（也就是static方法）来说，方法参数是从第0个位置开始的. 而对于实例方法来说，第0个位置上的变量是this指针。

局部变量可以是以下这些类型:

* char

* long

* short

* int

* float

* double

* 引用

* 返回地址

除了long和double类型外，每个变量都只占局部变量区中的一个变量槽(slot)，而long及double会占用两个连续的变量槽，因为这些类型是64位的。

当一个新的变量创建的时候，操作数栈（operand stack）会用来存储这个新变量的值。然后这个变量会存储到局部变量区中对应的位置上。如果这个变量不是基础类型的话，本地变量槽上存的就只是一个引用。这个引用指向堆的里一个对象。

比如：

int i = 5;

编译后就成了

0: bipush 5

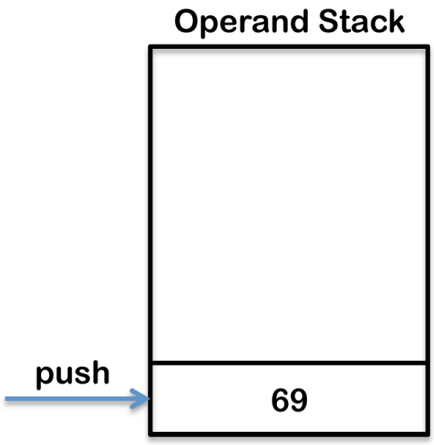
2: istore_0

bipush	用来将一个字节作为整型数字压入 这里5就会被压入操作数栈上。
istore_0	这是istore_这组指令集（译注：严 该叫做操作码，opcode ,指令是指 的操作数，oprand。不过操作码- 记符，这里统称为指令）中的一条 一个整型数字存储到本地变量中。 变量区中的位置，并且只能是0,1, 能用另一条指令istore了，这条指 作数，对应的是局部变量区中的位

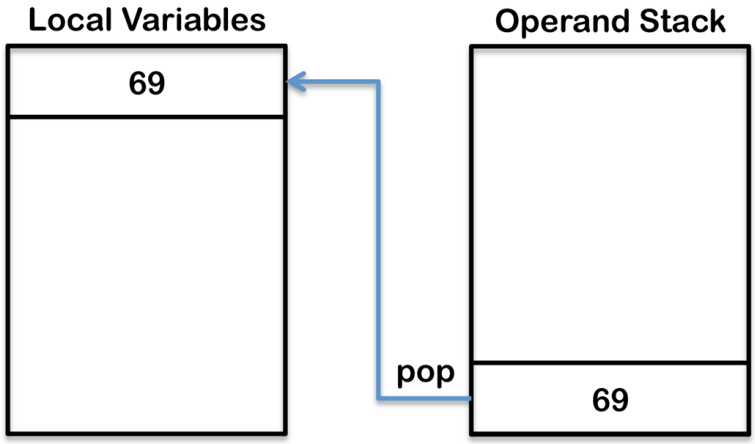
这条指令执行的时候，内存布局是这样的：

int i = 69;

0: bipush 69



2: istore_0



class文件中的每一个方法都会包含一个局部变量表，如果这段代码在一个方法里面的话，你会在类文件的局部变量表中发现如下的一条记录。

LocalVariableTable:

Start	Length	Slot	Name	Signature
0	1	1	i	I

字段

Java类里面的字段是作为类对象实例的一部分，存储在堆里面的（类变量对应存储在类对象里面）。关于字段的信息会添加到类文件里的field_info数组里，像下面这样：

```
ClassFile {
    u4 magic;
    u2 minor_version;
```

```
    u2 major_version;
    u2 constant_pool_count;
    cp_info contant_pool[constant_pool_count – 1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

另外，如果变量被初始化了，那么初始化的字节码会加到构造方法里。

下面这段代码编译了之后：

```
public class SimpleClass {

    public int simpleField = 100;

}
```

如果你用javap进行反编译，这个被添加到了field_info数组里的字段会多出一段描述信息。

```
public int simpleField;
    Signature: I
    flags: ACC_PUBLIC
```

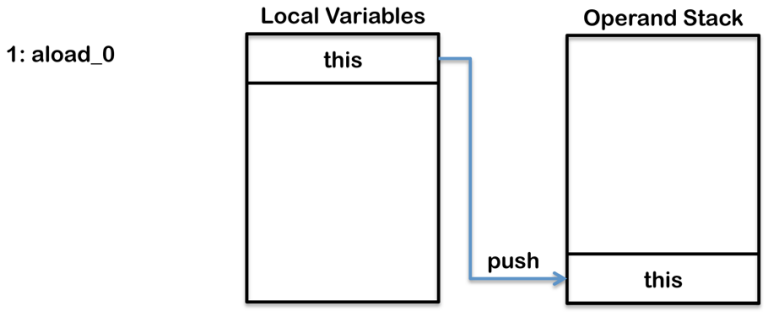
初始化变量的字节码会被加到构造方法里，像下面这样：

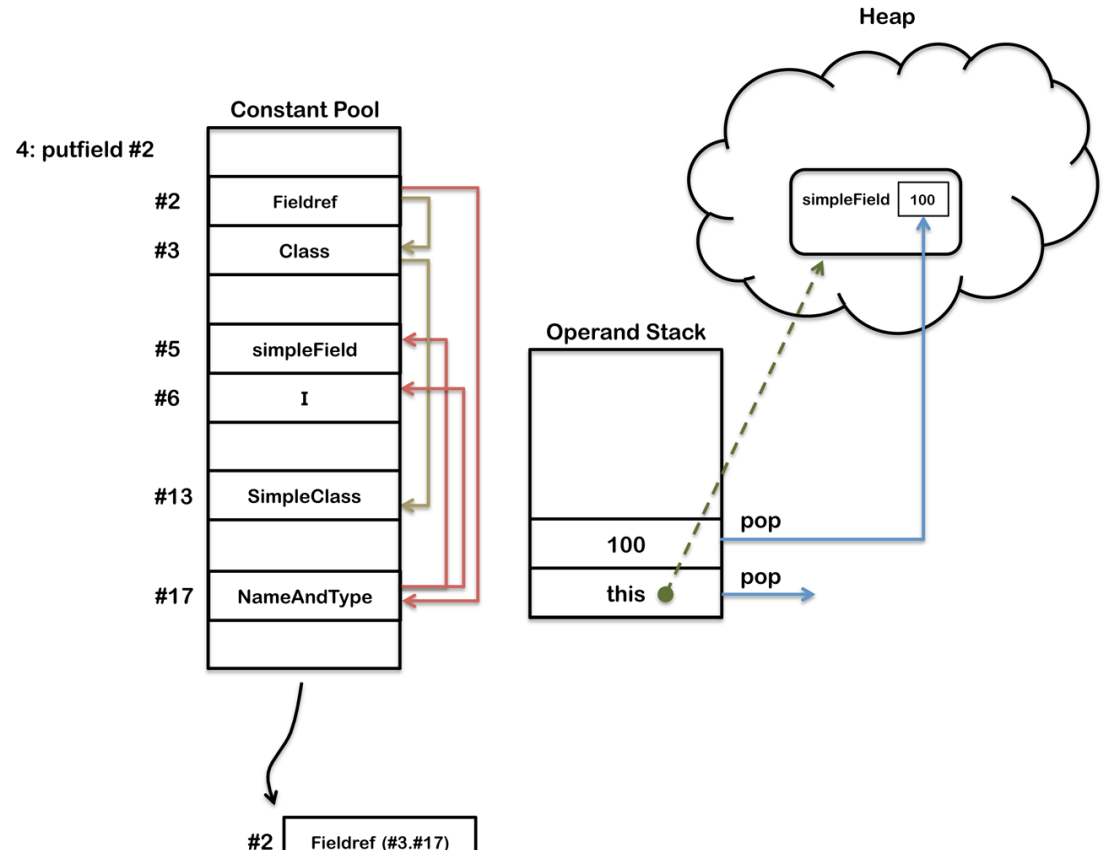
```
public SimpleClass();
  Signature: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=1, args_size=1
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: aload_0
       5: bipush        100
       7: putfield      #2                  // Field simpleField:I
      10: return
```

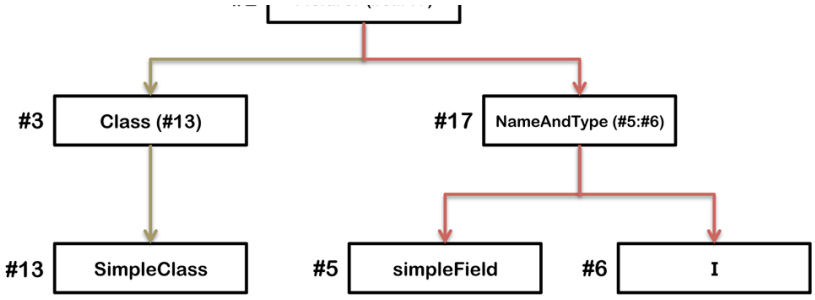
aload_0	从局部变量数组中加载一个对象引 栈顶。尽管这段代码看起来没有构 编译器生成的默认的构造方法里， 始化的代码。第一个局部变量正好 是aload_0把this引用压到操作数栈 aload_指令集中的一条，这组指令 操作数栈中。n对应的是局部变量 并且也只能是0,1,2,3。还有类似的 加载的并不是对象引用，比如iload 和dload_, 这里i代表int,l代表long, double。局部变量的在数组中的位 通过iload,lload,fload,dload,和aloa 指令都接受一个操作数，它代表的 变量的在数组中的位置。
invokespecial	这条指令可以用来调用对象实例的 方法和父类中的方法。它是方法调 条，其它的还有invokedynamic, in invokespecial, invokestatic, invok invokespecial指令调用的是父类也 java.lang.Object的构造方法。
bipush	它是用来把一个字节作为整型压至 在这里100会被压到操作数栈里。
putfield	它接受一个操作数，这个操作数引 量池里的一个字段，在这里这个字 simpleField。赋给这个字段的值， 段的对象引用，在执行这条指令的 作数栈顶上pop出来。前面的aloa 含这个字段的对象压到操作数栈上 bipush又把100压到栈里。最后pu 两个值从栈顶弹出。执行完的结果 simpleField这个字段的值更新成了

上述代码执行的时候内存里面是这样的：

```
public class SimpleClass {
   public int i = 100;
}
```







这里的putfield指令的操作数引用的是常量池里的第二个位置。JVM会为每个类型维护一个常量池，运行时的数据结构有点类似一个符号表，尽管它包含的信息更多。Java中的字节码操作需要对应的数据，但通常这些数据都太大了，存储在字节码里不适合，它们会被存储在常量池里面，而字节码包含一个常量池里的引用 。当类文件生成的时候，其中的一块就是常量池：

```
Constant pool:
   #1 = Methodref          #4.#16         //  java/lang/Object."<init>":()V
   #2 = Fieldref           #3.#17         //  SimpleClass.simpleField:I
   #3 = Class              #13            //  SimpleClass
   #4 = Class              #19            //  java/lang/Object
   #5 = Utf8               simpleField
   #6 = Utf8               I
   #7 = Utf8               <init>
   #8 = Utf8               ()V
   #9 = Utf8               Code
  #10 = Utf8               LineNumberTable
  #11 = Utf8               LocalVariableTable
  #12 = Utf8               this
  #13 = Utf8               SimpleClass
  #14 = Utf8               SourceFile
  #15 = Utf8               SimpleClass.java
  #16 = NameAndType        #7:#8          //  "<init>":()V
  #17 = NameAndType        #5:#6          //  simpleField:I
```

```
#18 = Utf8               LSimpleClass;
#19 = Utf8               java/lang/Object
```

常量字段（类常量）

带有final标记的常量字段在class文件里会被标记成ACC_FINAL.

比如

```
public class SimpleClass {

    public final int simpleField = 100;

}
```

字段的描述信息会标记成ACC_FINAL:

```
public static final int simpleField = 100;
  Signature: I
  flags: ACC_PUBLIC, ACC_FINAL
  ConstantValue: int 100
```

对应的初始化代码并不变：

```
4: aload_0
5: bipush        100
7: putfield      #2                  // Field simpleField:I
```

静态变量

带有static修饰符的静态变量则会被标记成ACC_STATIC：

```
public static int simpleField;
  Signature: I
  flags: ACC_PUBLIC, ACC_STATIC
```

不过在实例的构造方法中却再也找不到对应的初始化代码了。因为static变量会在类的构造方法中进行初始化，并且它用的是putstatic指令而不是putfiled。

```
static {};
  Signature: ()V
```

flags: ACC_STATIC

Code:

stack=1, locals=0, args_size=0

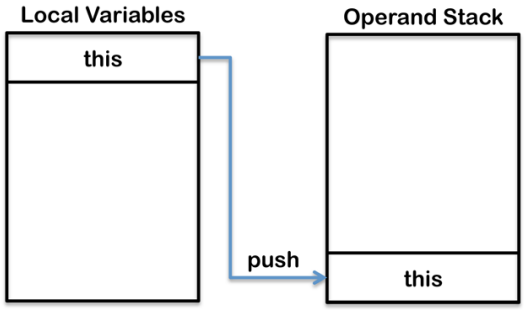
0: bipush 100

2: putstatic #2 // Field simpleField:I

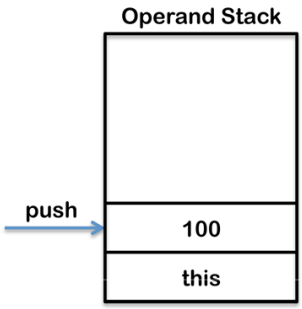
5: return

```
public class SimpleClass {
   public int i = 100;
}
```

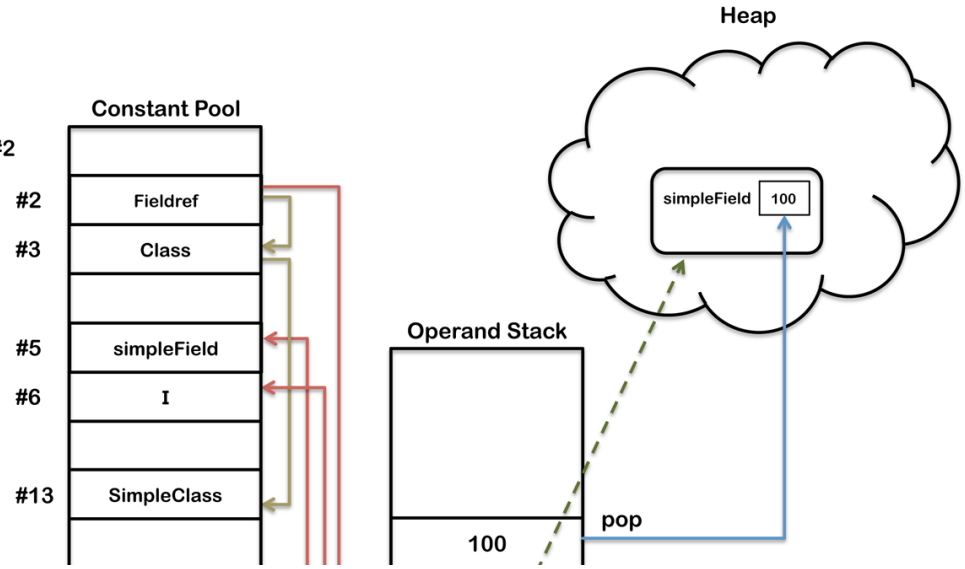
1: aload_0

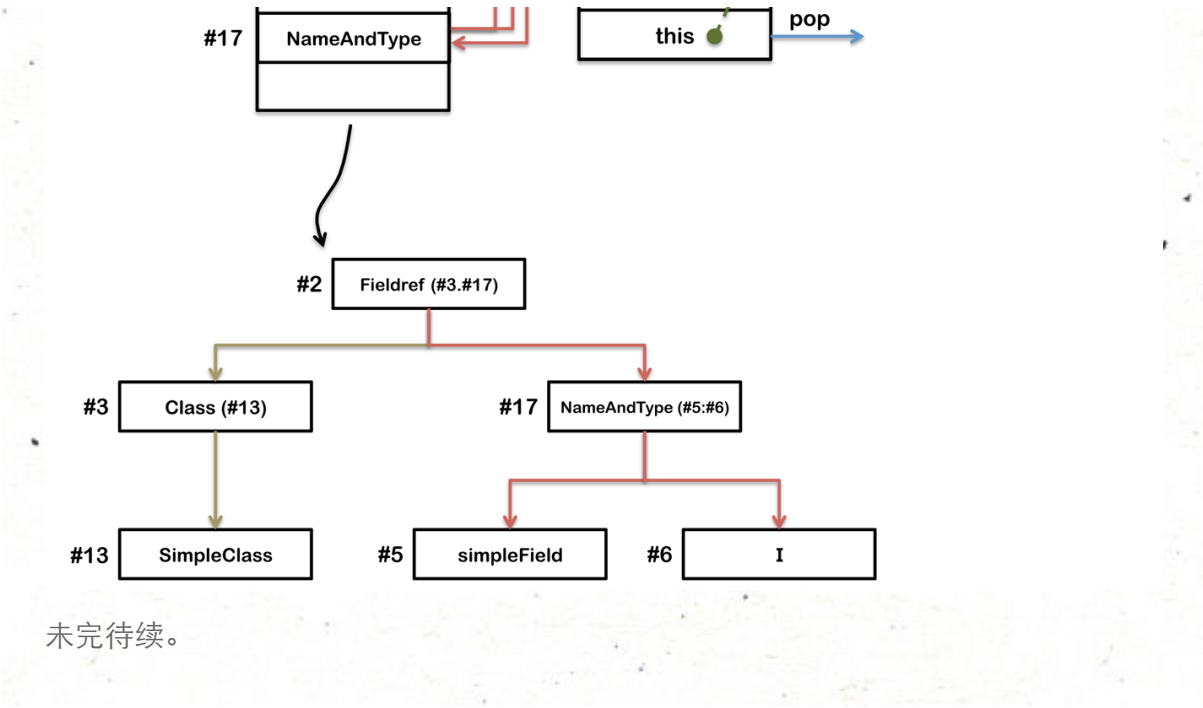


2: bipush 100



4: putfield #2




#17
NameAndType
this
pop
#2
Fieldref (#3.#17)
#3
Class (#13)
#17
NameAndType (#5:#6)
#13
SimpleClass
#5
simpleField
#6
I

未完待续。