http://ifeve.com/java-code-to-byte-code-3/

从Java7开始，switch语句增加了对String类型的支持。不过字节码中的switch指令还是只支持int类型，并没有增加对其它类型的支持。事实上switch语句对String的支持是分成两个步骤来完成的。首先，将每个case语句里的值的hashCode和操作数栈顶的值（译注：也就是switch里面的那个值，这个值会先压入栈顶）进行比较。这个可以通过lookupswitch或者是tableswitch指令来完成。结果会路由到某个分支上，然后调用String.equlals来判断是否确实匹配。最后根据equals返回的结果，再用一个tableswitch指令来路由到具体的case分支上去执行。

```
public int simpleSwitch(String stringOne) {
   switch (stringOne) {
      case "a":
         return 0;
      case "b":
         return 2;
      case "c":
         return 3;
      default:
         return 4;
   }
}
```

这个字符串的switch语句会生成下面的字节码：

```
 0: aload_1
 1: astore_2
 2: iconst_m1
 3: istore_3
 4: aload_2
 5: invokevirtual #2                  // Method java/lang/String.hashCode:()I
 8: tableswitch   {
       default: 75
```

```
            min: 97
            max: 99
             97: 36
             98: 50
             99: 64
        }
36: aload_2
37: ldc           #3                  // String a
39: invokevirtual #4                  // Method java/lang/String.equals:
(Ljava/lang/Object;)Z
42: ifeq          75
45: iconst_0
46: istore_3
47: goto          75
50: aload_2
51: ldc           #5                  // String b
53: invokevirtual #4                  // Method java/lang/String.equals:
(Ljava/lang/Object;)Z
56: ifeq          75
59: iconst_1
60: istore_3
61: goto          75
64: aload_2
65: ldc           #6                  // String c
67: invokevirtual #4                  // Method java/lang/String.equals:
(Ljava/lang/Object;)Z
70: ifeq          75
73: iconst_2
74: istore_3
```

```
75: iload_3
76: tableswitch   {
          default: 110
            min: 0
            max: 2
              0: 104
              1: 106
              2: 108
        }
104: iconst_0
105: ireturn
106: iconst_2
107: ireturn
108: iconst_3
109: ireturn
110: iconst_4
111: ireturn
```

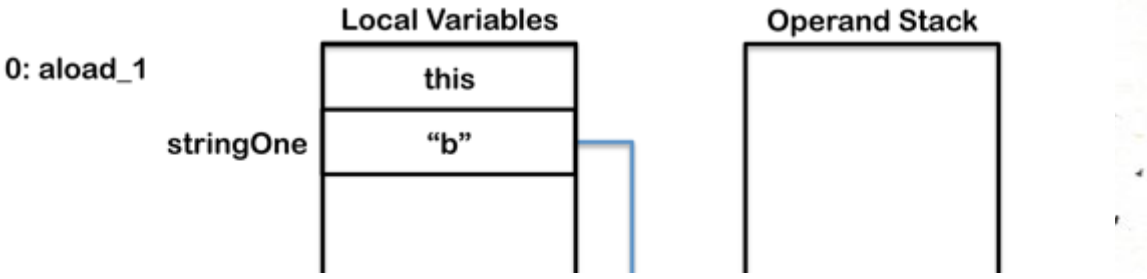
这段字节码所在的class文件里面，会包含如下的一个常量池。关于常量池可以看下JVM内部细节中的_运行时常量池_一节。

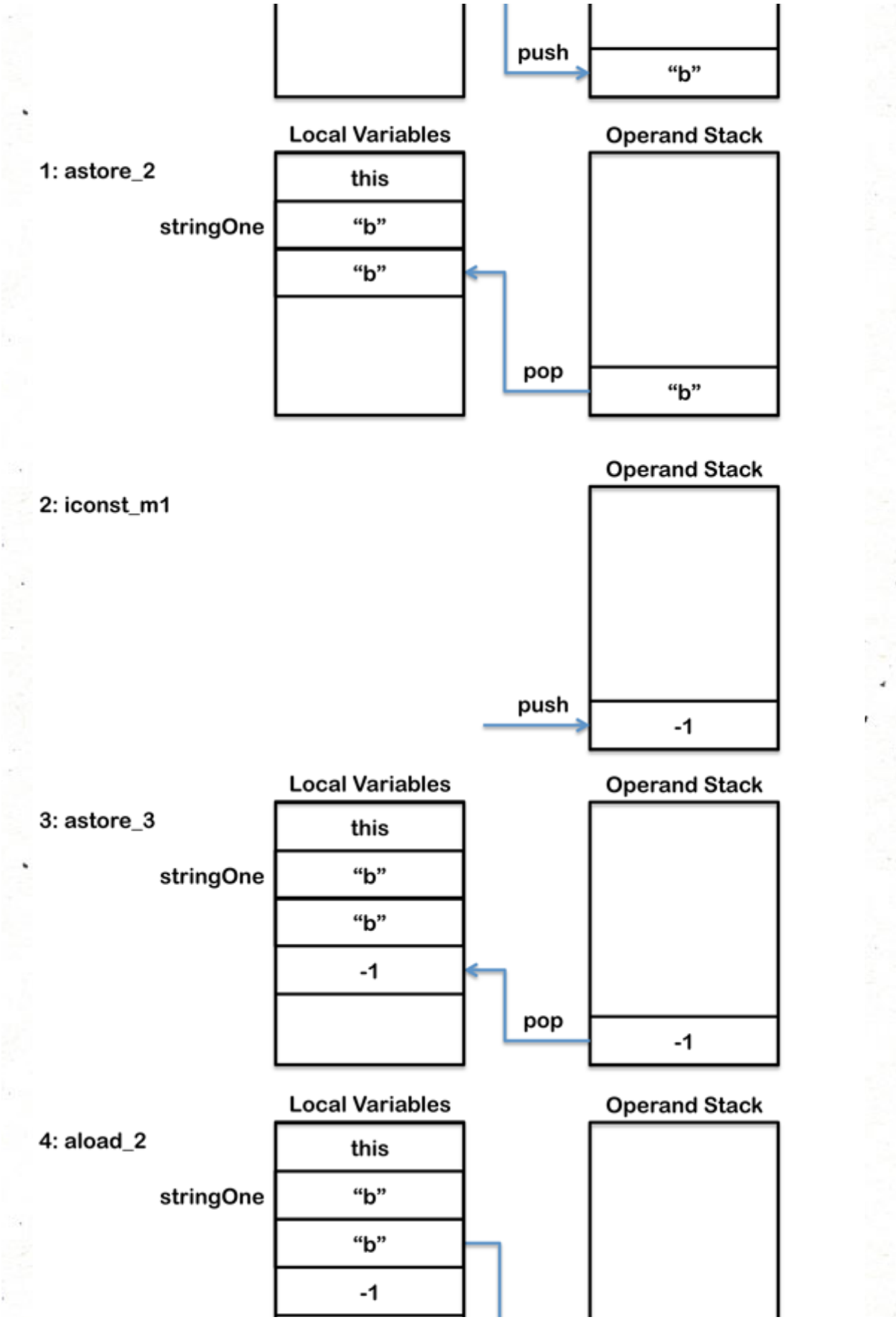
```
Constant pool:
   #2 = Methodref          #25.#26        //  java/lang/String.hashCode:()I
   #3 = String             #27            //  a
   #4 = Methodref          #25.#28        //  java/lang/String.equals:
(Ljava/lang/Object;)Z
   #5 = String             #29            //  b
   #6 = String             #30            //  c


  #25 = Class              #33            //  java/lang/String
  #26 = NameAndType        #34:#35        //  hashCode:()I
```

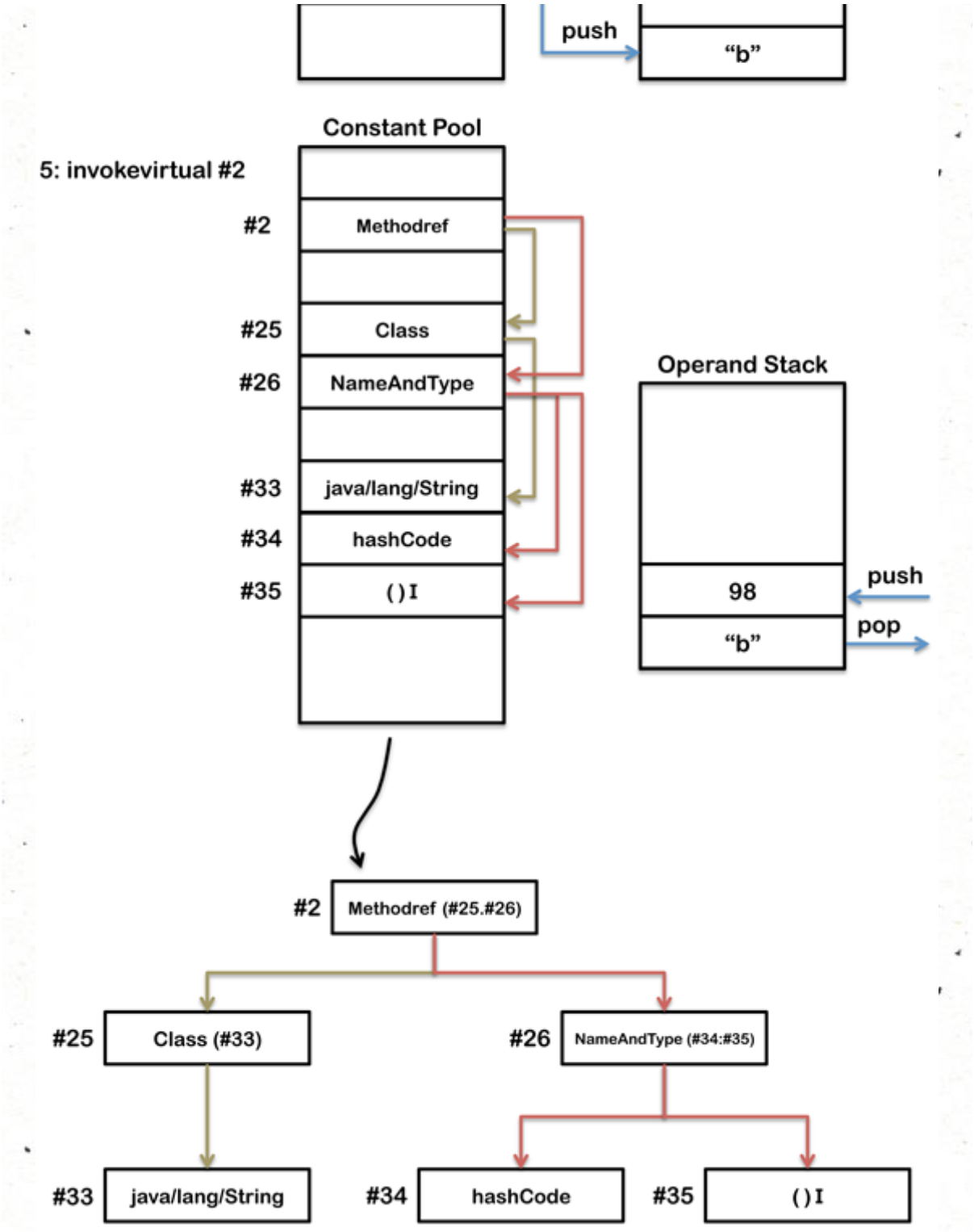
```
#27 = Utf8               a
#28 = NameAndType        #36:#37      // equals:(Ljava/lang/Object;)Z
#29 = Utf8               b
#30 = Utf8               c

#33 = Utf8               java/lang/String
#34 = Utf8               hashCode
#35 = Utf8               ()I
#36 = Utf8               equals
#37 = Utf8               (Ljava/lang/Object;)Z
```

注意，在执行这个switch语句的时候，用到了两个tableswitch指令，同时还有数个invokevirtual指令，这个是用来调用String.equals()方法的。在下一篇文章中关于方法调用的那节，会详细介绍到这个invokevirtual指令。下图演示了输入为”b”的情况下，这个swith语句是如何执行的。

```
public int simpleSwitch(String stringOne) {
    switch (stringOne) {
      case "a":
        return 0;
      case "b":
        return 2;
      case "c":
        return 3;
      default:
        return 4;
    }
}

simpleSwitch(“b”);
```




push
"b"
Local Variables
Operand Stack
1: astore_2
this
stringOne
"b"
"b"
pop
"b"
Operand Stack
2: iconst_m1
push
-1
Local Variables
Operand Stack
3: astore_3
this
stringOne
"b"
"b"
-1
pop
-1
Local Variables
Operand Stack
4: aload_2
this
stringOne
"b"
"b"
-1



如果有几个分支的hashcode是一样的话，比如说“FB”和”Ea”，它们的hashCode都是28，得简单的调整下equals方法的处理流程来进行处理。在下面的这个例子中，34行

处的字节码ifeg 42会跳转到另一个String.equals方法调用，而不是像前面那样执行lookupswitch指令，因为前面的那个例子中hashCode没有冲突。(译注：这里一般容易弄混淆，认为ifeq是字符串相等，为什么要跳到下一处继续比较字符串？其实ifeq是判断栈顶元素是否和0相等，而栈顶的值就是String.equals的返回值，而true,也就是相等，返回的是1，false返回的是0，因此ifeq为真的时候表明返回的是false，这会儿就应该继续进行下一个字符串的比较）

```
public int simpleSwitch(String stringOne) {
   switch (stringOne) {
      case "FB":
         return 0;
      case "Ea":
         return 2;
      default:
         return 4;
   }
}
```

这段代码会生成下面的字节码：

```
 0: aload_1
 1: astore_2
 2: iconst_m1
 3: istore_3
 4: aload_2
 5: invokevirtual #2                  // Method java/lang/String.hashCode:()I
 8: lookupswitch  {
       default: 53
         count: 1
          2236: 28
```

```
  }
28: aload_2
29: ldc           #3                  // String Ea
31: invokevirtual #4                  // Method java/lang/String.equals:
(Ljava/lang/Object;)Z
34: ifeq          42
37: iconst_1
38: istore_3
39: goto          53
42: aload_2
43: ldc           #5                  // String FB
45: invokevirtual #4                  // Method java/lang/String.equals:
(Ljava/lang/Object;)Z
48: ifeq          53
51: iconst_0
52: istore_3
53: iload_3
54: lookupswitch  {
       default: 84
         count: 2
             0: 80
             1: 82
  }
80: iconst_0
81: ireturn
82: iconst_2
83: ireturn
84: iconst_4
85: ireturn
```

###循环语句

if-else和switch这些条件流程控制语句都是先通过一条指令比较两个值，然后跳转到某个分支去执行。

for循环和while循环这些语句也类似，只不过它们通常都包含一个goto指令，使得字节码能够循环执行。do-while循环则不需要goto指令，因为它们的条件判断指令是放在循环体的最后来执行。

有一些操作码能在单条指令内完成整数或者引用的比较，然后根据结果跳转到某个分支继续执行。而比较double,long,float这些类型则需要两条指令。首先会将两个值进行比较，然后根据结果把1，-1，0压入操作数栈中。然后再根据栈顶的值是大于小于或者等于0，来决定下一步要执行的指令的位置。这些指令在上一篇文章中有详细的介绍。

####while循环

while循环包含条件跳转指令比如if_icmpge 或者if_icmplt（前面有介绍）以及goto指令。如果判断条件不满足的话，会跳转到循环体后的第一条指令继续执行，循环结束（译注：这里判断条件和代码中的正好相反，如代码中是i<2，字节码内是i>=2，从字节码的角度看，是满足条件后循环中止）。循环体的末尾是一条goto指令，它会跳转到循环开始的地方继续执行，直到分支跳转的条件满足才终止。

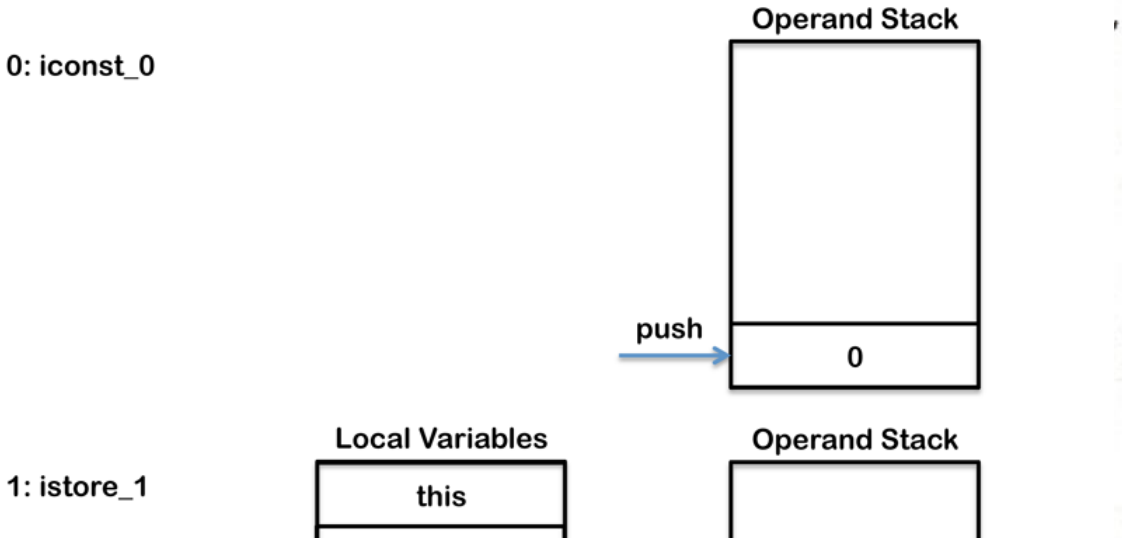
```
public void whileLoop() {
    int i = 0;
    while (i < 2) {
        i++;
    }
}
```

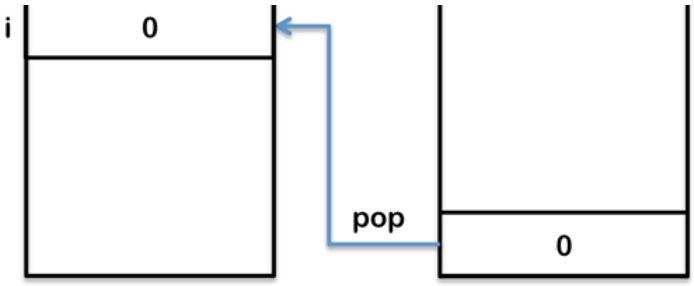
编译完后是：

```
0: iconst_0
1: istore_1
2: iload_1
3: iconst_2
4: if_icmpge 13
7: iinc 1, 1
10: goto 2
13: return
```

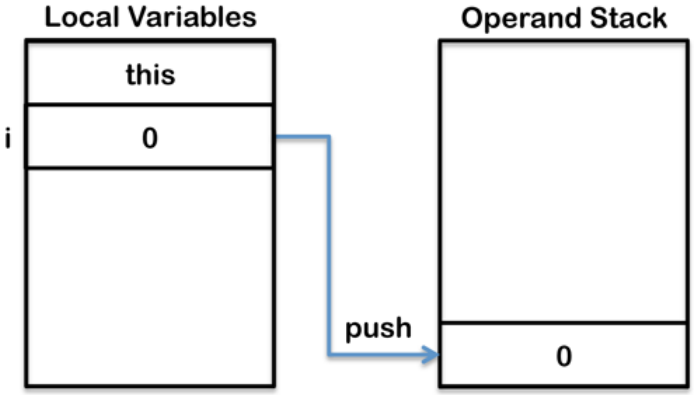
if_icmpge指令会判断局部变量区中的1号位的变量（也就是i，译注：局部变量区从0开始计数，第0位是this)是否大于等于2，如果不是继续执行，如果是的话跳转到13行处，结束循环。goto指令使得循环可以继续执行，直到条件判断为真，这个时候会跳转到紧挨着循环体后边的return指令处。iinc是少数的几条能直接更新局部变量区里的变量的指令之一，它不用把值压到操作数栈里面就能直接进行操作。这里iinc指令把第1个局部变量（译注：第0个是this）自增1。

```
public void whileLoop() {
    int i = 0;
    while (i < 10) {
        i++;
    }
}
```

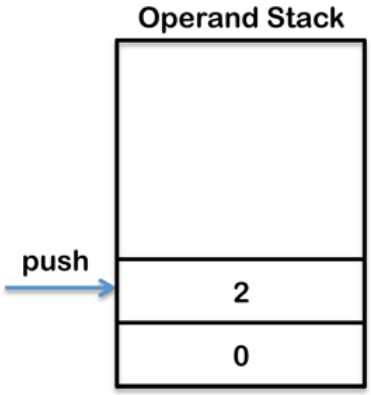




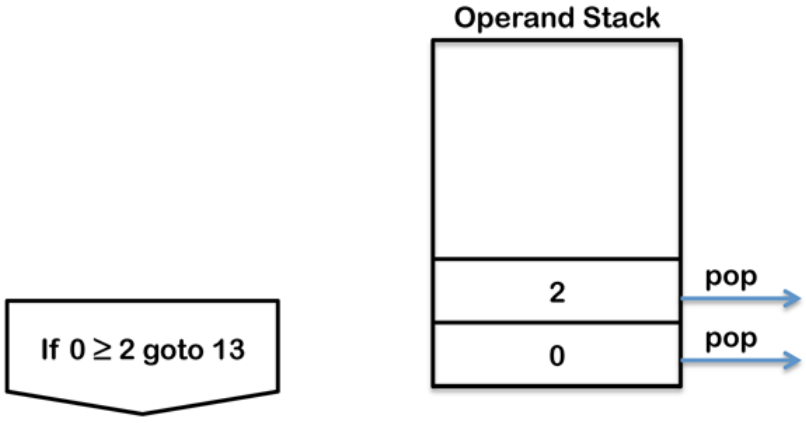
2: iload_1



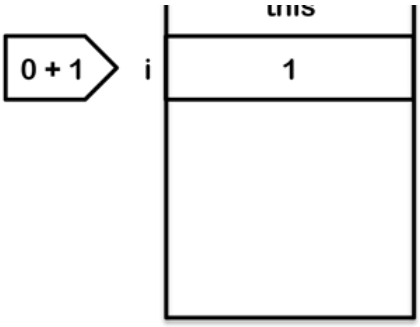
3: iconst_2



4: if_icmpge 13



7: iinc 1, 1



for循环和while循环在字节码里的格式是一样的。这并不奇怪，因为每个while循环都可以很容易改写成一个for循环。比如上面的while循环就可以改写成下面的for循环，当然了它们输出的字节码也是一样的：

```
public void forLoop() {
    for(int i = 0; i < 2; i++) {

    }
}
```

####do-while循环

do-while循环和for循环，while循环非常类似，除了一点，它是不需要goto指令的，因为条件跳转指令在循环体的末尾，可以用它来跳转回循环体的起始处。

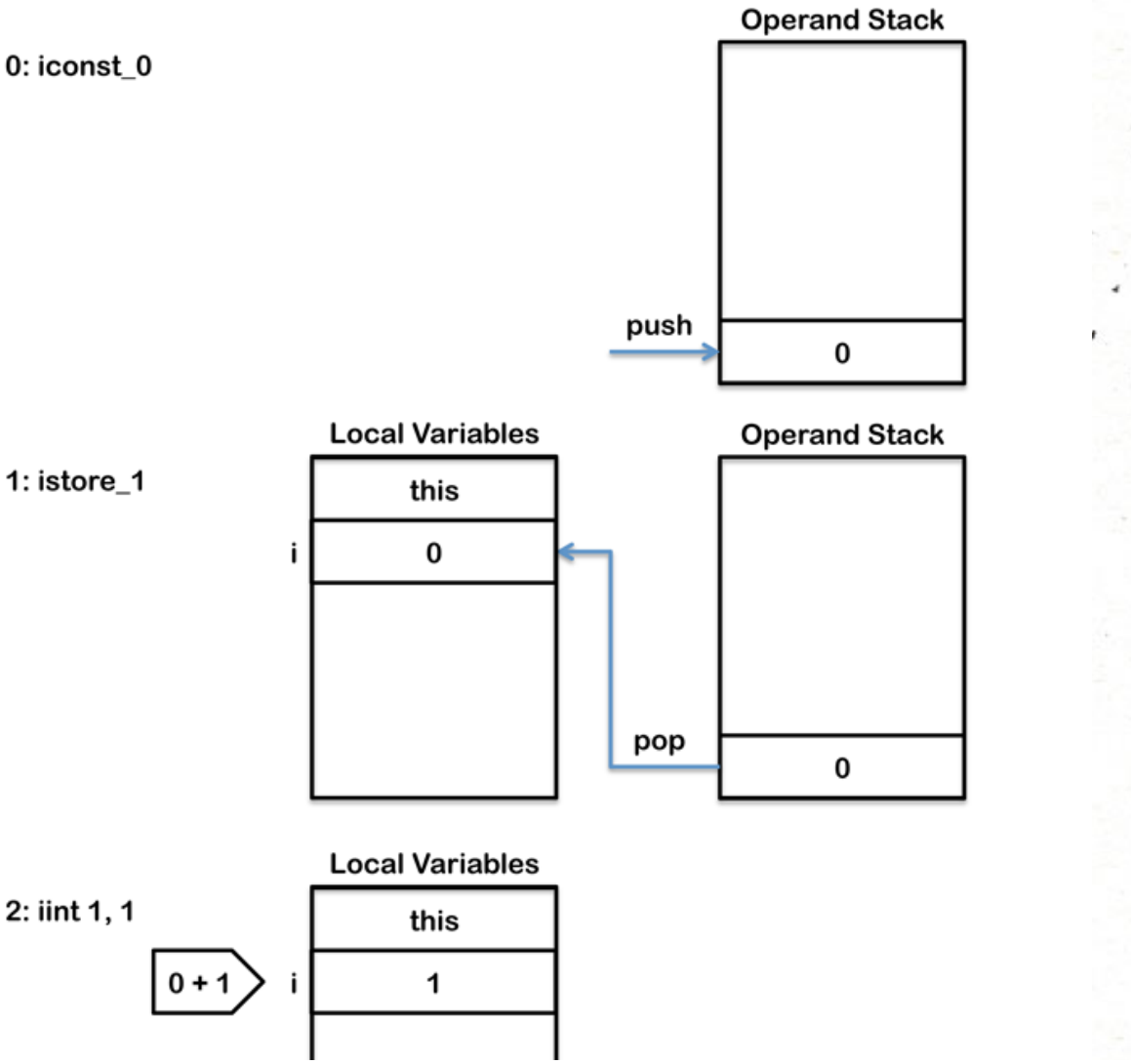
```
public void doWhileLoop() {
    int i = 0;
    do {
        i++;
    } while (i < 2);
}
```

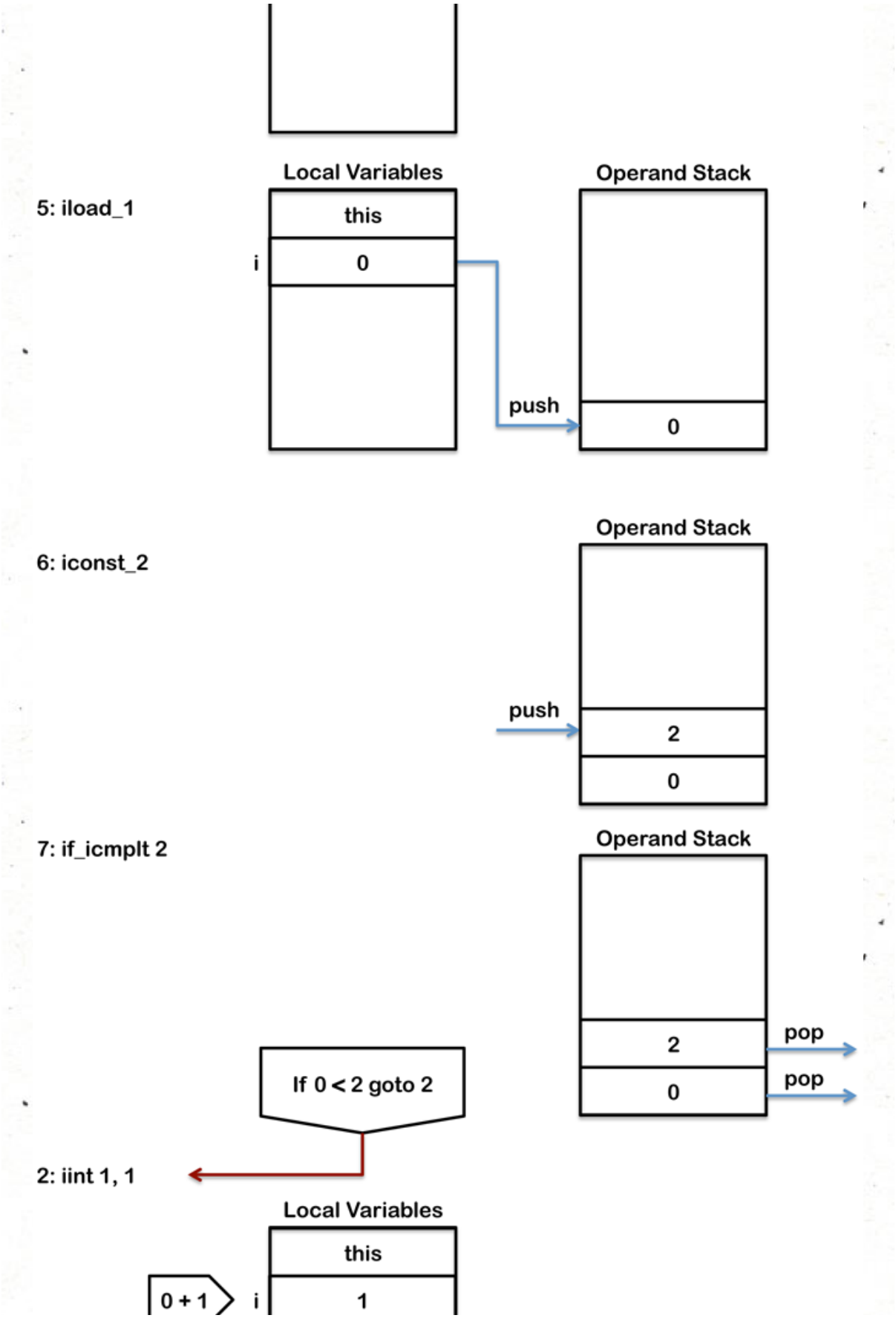
这会生成如下的字节码：

```
0: iconst_0
```

```
1: istore_1
2: iinc          1, 1
5: iload_1
6: iconst_2
7: if_icmplt     2
10: return
```

```
public void doWhileLoop() {
    int i = 0;
    do {
       i++;
    } while (i < 2);
}
```




Local Variables
Operand Stack
5: iload_1
this
i
0
push
0
Operand Stack
6: iconst_2
push
2
0
Operand Stack
7: if_icmplt 2
2
pop
0
pop
If 0 < 2 goto 2
2: iinc 1, 1
Local Variables
this
0 + 1
i
1