http://ifeve.com/javacode2bytecode2/

条件语句

像if-else, switch这样的流程控制的条件语句，是通过用一条指令来进行两个值的比较，然后根据结果跳转到另一条字节码来实现的。

循环语句包括for循环，while循环，它们的实现方式也很类似，但有一点不同，它们通常都会包含一条goto指令，以便字节码实现循环执行。do-while循环不需要goto指令，因为它的条件分支是在字节码的末尾。更多细节请参考循环语句一节。

有一些指令可以用来比较两个整型或者两个引用，然后执行某个分支，这些操作都能在单条指令里面完成。而像double,float,long这些值需要两条指令。首先得去比较两个值，然后根据结果，会把1，0或者-1压到栈里。最后根据栈顶的值是大于，等于或者小于0来判断应该跳转到哪个分支。

我们先来介绍下if-else语句，然后再详细介绍下分支跳转用到的几种不同的指令。

if-else

下面的这个简单的例子是用来比较两个整数的：

```
1 public int greaterThen(int intOne, int intTwo) {
2 if (intOne > intTwo) {
3 return 0;
4 } else {
```

```
5  return 1;
6  }
7  }
```

方法最后会编译成如下的字节码:

```
1  0: iload_1
2  1: iload_2
3  2: if_icmple 7
4  5: iconst_0
5  6: ireturn
6  7: iconst_1
7  8: ireturn
```

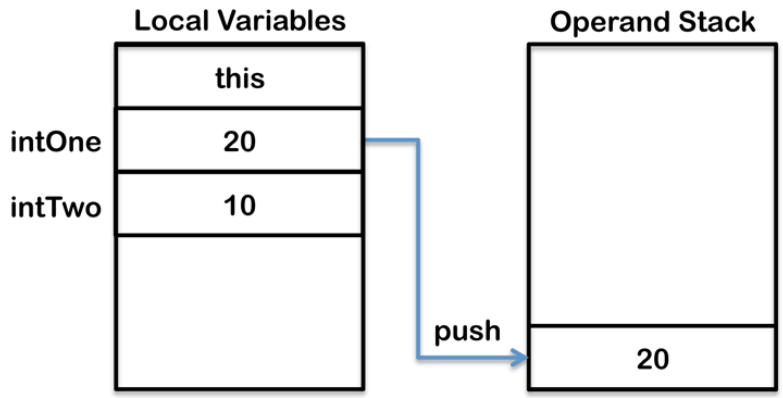
首先，通过iload_1, iload_2两条指令将两个入参压入操作数栈中。if_icmple会比较栈顶的两个值的大小。如果intOne小于或者等于intTwo的话，会跳转到第7行处的字节码来执行。可以看到这里和Java代码里的if语句的条件判断正好相反，这是因为在字节码里面，判断条件为真的话会跑到else分支里面去执行，而在Java代码里，判断为真会进入if块里面执行。换言之，if_icmple判断的是如果if条件不为真，然后跳过if块。if代码块里对应的代码是5，6处的字节码，而else块对应的是7，8处的。

```
public int greaterThen(int intOne, int intTwo) {
    if (intOne > intTwo) {
        return 0;
    } else {
        return 1;
    }
```

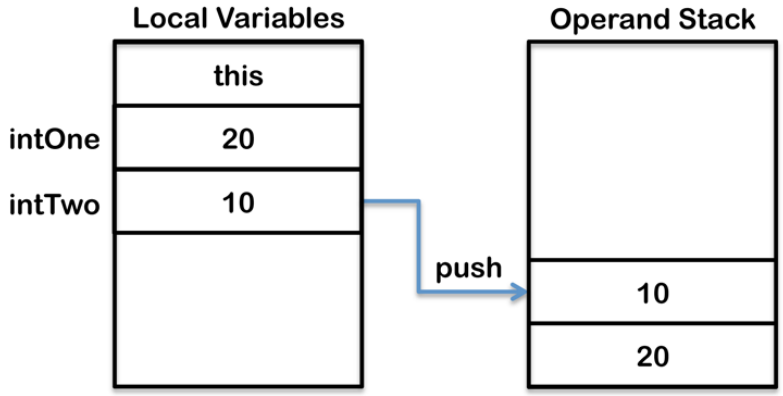
```
}

greaterThen(10, 20);
```

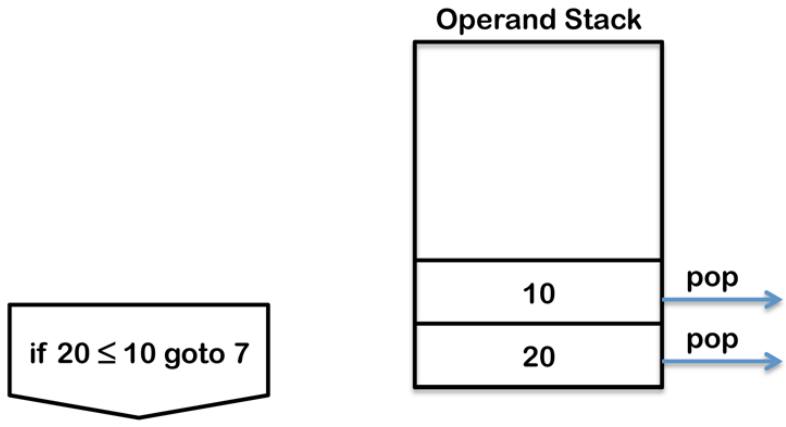
0: iload_1



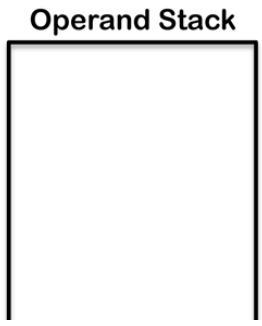
1: iload_2



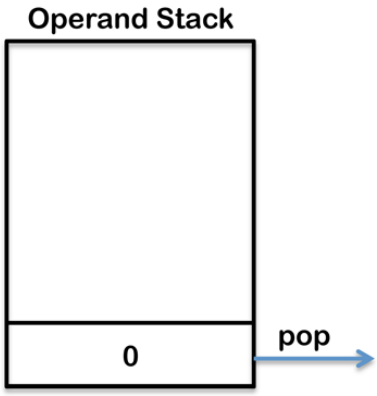
2: if_icmple 7



5: iconst_0



6: ireturn



下面的代码则稍微复杂了一点，它需要进行两次比较。

```
1 public int greaterThen(float floatOne, float floatTwo) {
2 int result;
3 if (floatOne > floatTwo) {
4 result = 1;
5 } else {
6 result = 2;
7 }
8 return result;
9 }
```

编译后会是这样：

```
01 0: fload_1
02 1: fload_2
```

```
03 2:
04 3: ifle 11
05 6: iconst_1
06 7: istore_3
07 8: goto 13
08 11: iconst_2
09 12: istore_3
10 13: iload_3
11 14: ireturn
```

在这个例子中，首先两个参数会被fload_1和fload_2指令压入栈中。和上面那个例子不同的是，这里需要比较两回。fcmple先用来比较栈顶的floatOne和floatTwo，然后把比较的结果压入操作数栈中。

```
1 * floatOne > floatTwo -> 1
2 * floatOne = floatTwo -> 0
3 * floatOne < floatTwo -> -1
```

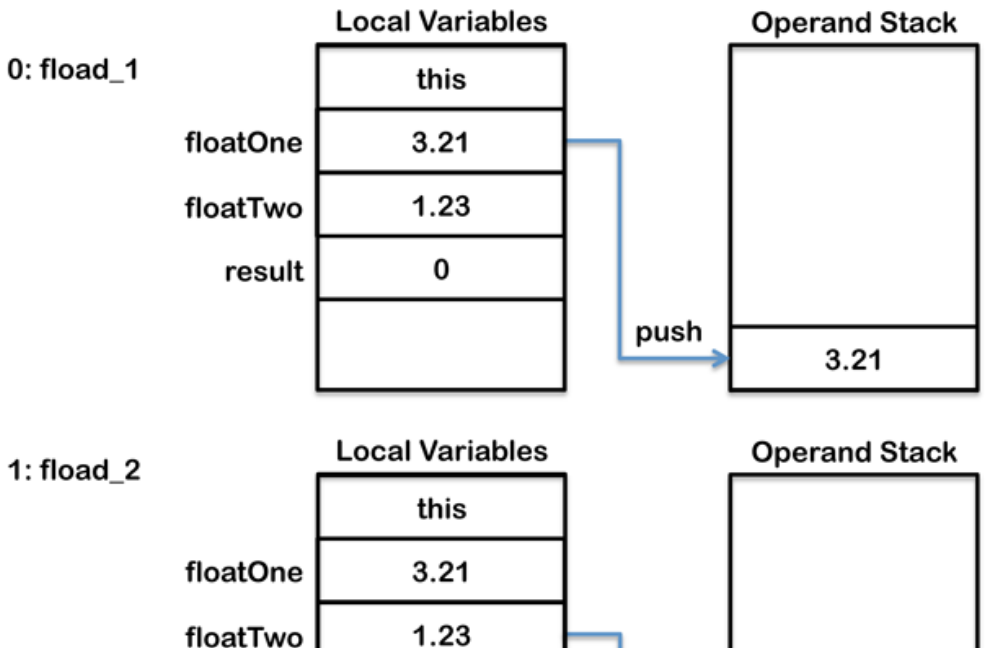
```
4 *
  floatOn
  e or
  floatTw
  o = NaN
```

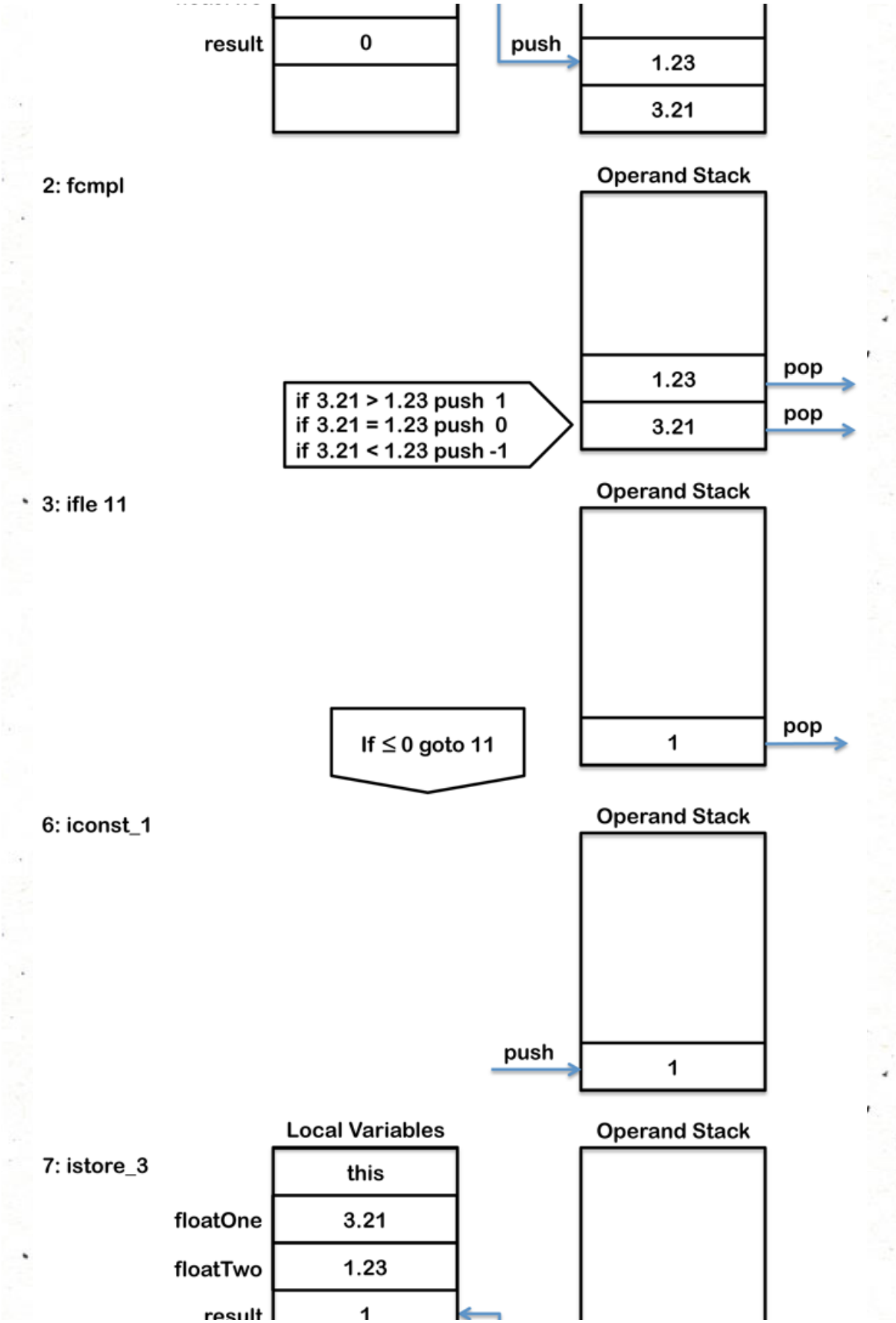
然后通过ifle进行判断，如果前面fcmpl的结果是< =0的话，则跳转到11行处的字节码去继续执行。

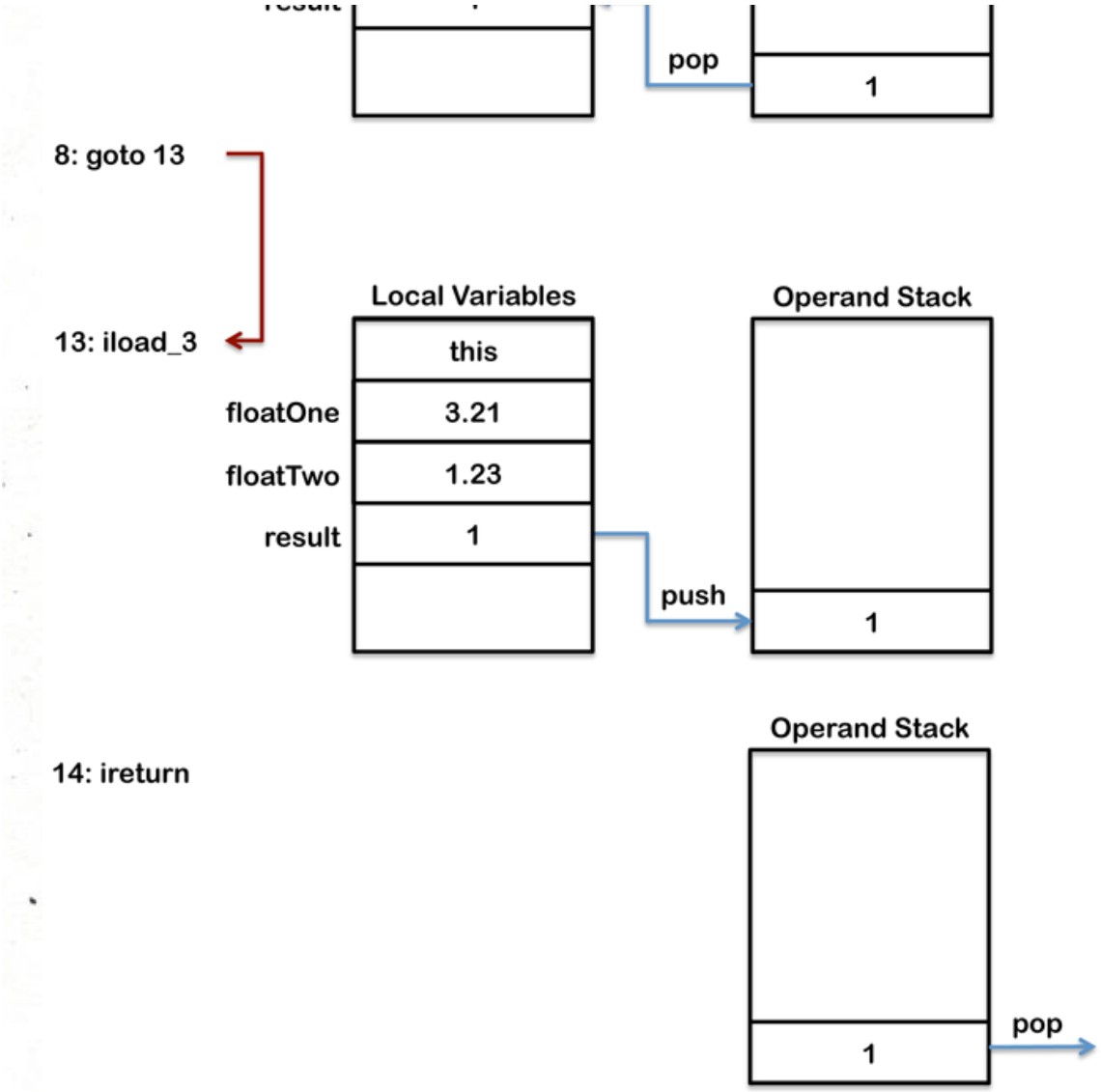
这个例子还有一个地方和前面不同的是，它只在方法末有一个return语句，因此在if代码块的最后，会有一个goto语句来跳过else块。goto语句会跳转到第13条字节码处，然后通过iload_3将存储在局部变量区第三个位置的结果压入栈中，然后就可以通过return指令将结果返回了。

```
public int greaterThen(float floatOne, float floatTwo) {
    int result;
    if (floatOne > floatTwo) {
       result = 1;
    } else {
       result = 2;
    }
    return result;
}

greateThen(3.21, 1.23);
```




result
0
push
1.23
3.21
2: fcmpl
Operand Stack
1.23
3.21
pop
pop
if 3.21 > 1.23 push 1
if 3.21 = 1.23 push 0
if 3.21 < 1.23 push -1
3: ifle 11
Operand Stack
If ≤ 0 goto 11
1
pop
6: iconst_1
Operand Stack
push
1
7: istore_3
Local Variables
Operand Stack
this
floatOne
3.21
floatTwo
1.23
result
1



除了比较数值的指令外，还有比较引用是否相等的(==)，以及引用是否等于null的（== null或者!=null)，以及比较对象的类型的（instanceof)。

if_icmp<cond>	这组指令用来比较操作数栈顶的两 转到新的位置去执行。<cond>可 ne-不等于，lt-小于,le-小于等于， 于等于。
if_acmp<cond>	这两个指令用来比较对象是否相等 数指定的位置进行跳转。
ifnonnull ifnull	这两个指令用来判断对象是否为n 作数指定的位置进行跳转。
lcmp	这个指令用来比较栈顶的两个长整 值压入栈中： 如果value1>value2 value1==value2,压入0，如果valu 入-1.
fcmp<cond> l g dcomp<cond>	这组指令用来比较两个float或者d 然后然后将结果值压入栈中：如果 压入1,如果value1==value2,压入0 value1<value2压入-1. 指令可以以 同之处在于它们是如何处理NaN的 dcmpg指令把整数1压入操作数栈 dcmpl把-1压入操作数栈。这确保 时候，如果其中一个不是数字（N

switch语句

Java switch表达式的类型只能是char,byte,short,int,Character, Byte, Short,Integer,String或者enum。JVM为了支持switch语句，用了两个特殊的指令，叫做tableSwitch和lookupswitch，它们都只能操作整型数值。只能使用整型并不影响，因为char,byte,short和enum都可以提升成int类型。Java7开始支持String类型，下面我们会介绍到。tableswitch操作会比较快一些，不过它消耗的内存会更多。

tableswitch会列出case分支里面最大值和最小值之间的所有值，如果判断的值不在这个范围内则直接跳转到default块执行，case中没有的值也会被列出，不过它们同样指向的是default块。拿下面的这个switch语句作为例子：

```
01 public int simpleSwitch(int intOne) {
```

```
03 case 0:
04 return 3;
05 case 1:
06 return 2;
07 case 4:
08 return 1;
09 default:
10 return -1;
11 }
12 }
```

编译后会生成如下的字节码

```
01 0: iload_1
02 1: tableswitch {
03 default: 42
04 min: 0
05 max: 4
06 0: 36
07 1: 38
08 2: 42
09 3: 42
10 4: 40
11 }
```

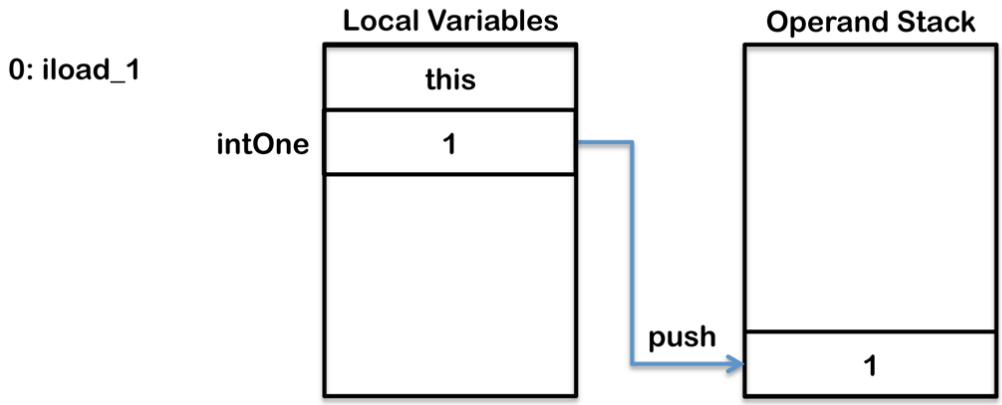
```
12 36: iconst
13 37: ireturn
14 38: iconst_2
15 39: ireturn
16 40: iconst_1
17 41: ireturn
18 42: iconst_m1
19 43: ireturn
```

tableswitch指令里0，1，4的值和代码里的case语句一一对应，它们指向的是对应代码块的字节码。tableswitch指令同样有2,3的值，但代码中并没有对应的case语句，它们指向的是default代码块。当这条指令执行的时候，会判断操作数栈顶的值是否在最大值和最小值之间。如果不在的话，直接跳去default分支，也就是上面的42行处的字节码。为了确保能找到default分支，它都是出现在tableswitch指令的第一个字节（如果需要内存对齐的话，则在补齐了之后的第一个字节）。如果栈顶的值在最大最小值的范围内，则用它作为tableswtich内部的索引，定位到应该跳转的分支。比如1的话，就会跳转至38行处继续执行。下图会演示这条指令是如何执行的：

```
public int simpleSwitch(int intOne) {
    switch (intOne) {
      case 0:
        return 3;
      case 1:
        return 2;
      case 4:
        return 1;
      default:
```

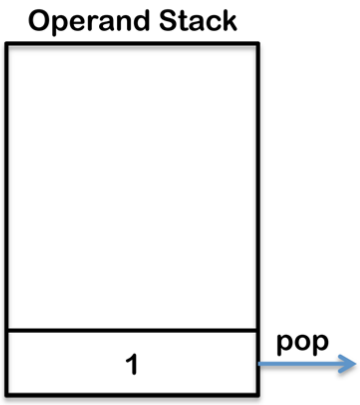
```
        return -1;
    }
}

simpleSwitch(1);
```

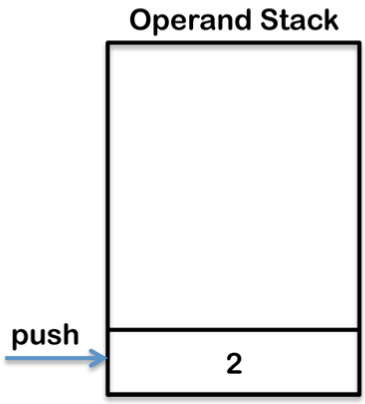


1: tableswitch

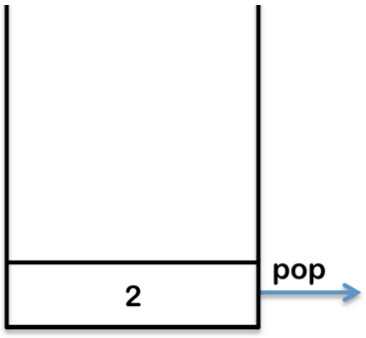
default	42
lowbyte	0
highbyte	4
0 jump	36
1 jump	38
2 jump	42
3 jump	42
4 jump	40



38: iconst_2



39: ireturn



如果case语句里面的值取值范围太广了（也就是太分散了）这个方法就不太好了，因为它占用的内存太多了。因此当switch的case条件里面的值比较分散的时候，就会使用lookupswitch指令。这个指令会列出case语句里的所有跳转的分支，但它没有列出所有可能的值。当执行这条指令的时候，栈顶的值会和lookupswitch里的每个值进行比较，来确定要跳转的分支。执行lookupswitch指令的时候，JVM会在列表中查找匹配的元素，这和tableswitch比起来要慢一些，因为tableswitch直接用索引就定位到正确的位置了。当switch语句编译的时候，编译器必须去权衡内存的使用和性能的影响，来决定到底该使用哪条指令。下面的代码，编译器会生成lookupswitch语句：

```
01 public int simpleSwitch(int intOne) {
02 switch (intOne) {
03 case 10:
04 return 1;
05 case 20:
06 return 2;
```

```
07 case
08 return 3;
09 default:
10 return -1;
11 }
12 }
```

生成后的字节码如下:

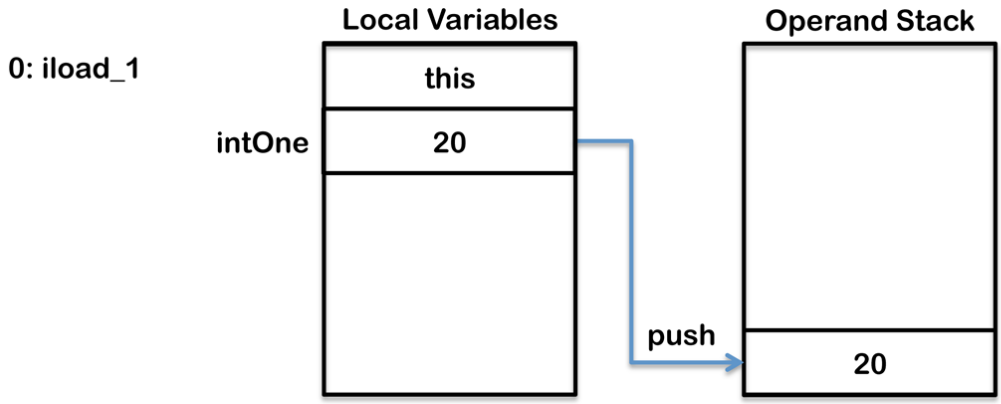
```
01 0: iload_1
02 1: lookupswitch {
03 default: 42
04 count: 3
05 10: 36
06 20: 38
07 30: 40
08 }
09 36: iconst_1
10 37: ireturn
11 38: iconst_2
12 39: ireturn
13 40: iconst_3
```

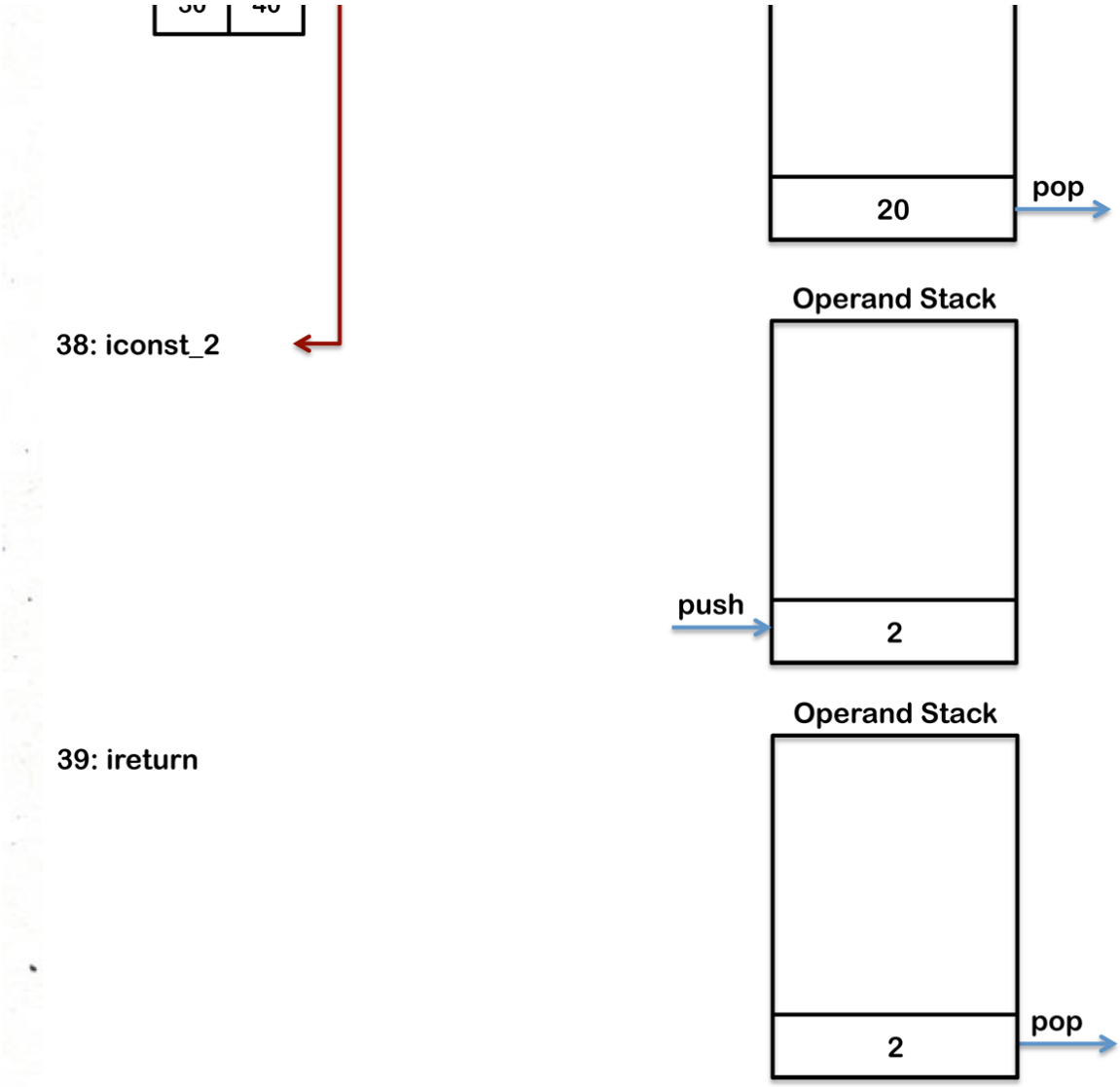
```
14 41: ireturn
15 42: iconst_m1
16 43: ireturn
```

为了确保搜索算法的高效（得比线性查找要快），这里会提供列表的长度，同时匹配的元素也是排好序的。下图演示了lookupswitch指令是如何执行的。

```
public int simpleSwitch(int intOne) {
    switch (intOne) {
      case 10:
        return 1;
      case 20:
        return 2;
      case 30:
        return 3;
      default:
        return -1;
    }
}

simpleSwitch(20);
```




20
pop
Operand Stack
38: iconst_2
push
2
Operand Stack
39: ireturn
2
pop

未完待续。