从这篇文章开始，我们开始分类型讲解字节码指令。这篇要讲的是控制转移指令，也就是 if-else、三目表达式、switch-case 背后的指令。

0x01 概述

根据字节码的不同用途，可以大概分为如下几类

- 加载和存储指令，比如 iload 将一个整形值从局部变量表加载到操作数栈
- 控制转移指令，比如条件分支 ifeq
- 对象操作，比如创建类实例的指令 new
- 方法调用，比如 invokevirtual 指令用于调用对象的实例方法
- 运算指令和类型转换，比如加法指令 iadd
- 线程同步，monitorenter 和 monitorexit 两条指令来支持 synchronized 关键字的语义
- 异常处理，比如 athrow 显式抛出异常

我们首先来介绍控制转移指令

0x02 控制转移指令

控制转移指令根据条件进行分支跳转，我们常见的 if-then-else、三目表达式、for 循环、异常处理等都属于这个范畴。对应的指令集包括：

- 条件分支：ifeq、iflt、ifle、ifne、ifgt、ifge、ifnull、ifnonnull、if_icmpeq、 if_icmpne、if_icmplt, if_icmpgt、if_icmple、if_icmpge、if_acmpeq 和 if_acmpne。
- 复合条件分支：tableswitch、lookupswitch
- 无条件分支：goto、goto_w、jsr、jsr_w、ret

看一个 for 循环的例子

```
public class MyLoopTest {
    public static int[] numbers = new int[]{1, 2,
3};
    public static void main(String[] args) {
        ScoreCalculator calculator = new
ScoreCalculator();
        for (int number : numbers) {
            calculator.record(number);
        }
    }
}
```

对应的字节码

```
public static void main(java.lang.String[]);
Code:
   0: new           #2                  // class
ScoreCalculator
   3: dup
   4: invokespecial #3                  // Method
ScoreCalculator."<init>":()V
   7: astore_1

   8: getstatic     #4                  // Field
numbers:[I
  11: astore_2
  12: aload_2
  13: arraylength
  14: istore_3
  15: iconst_0
  16: istore       4
```

```
  18: iload          4
  20: iload_3
  21: if_icmpge      43
  24: aload_2
  25: iload          4
  27: iaload
  28: istore         5
  30: aload_1
  31: iload          5
  33: i2d
  34: invokevirtual #5  // Method
ScoreCalculator.record:(D)V

  37: iinc           4, 1
  40: goto           18

  43: return
```

先把局部变量表的示意图放出来便于理解

0	1	2	3	4	5
args	calculator	$array	$len	$i	

- 0 ~ 7：new、dup 和一个 的 invokespecial 表示创建新类实例，下一节会重点介绍
- 8 ~ 16：是初始化循环控制变量的一个过程。加载静态变量数组引用，存储到局部变量下标为 2 的位置上，记为$array，aload_2 加载$array到栈顶，调用 arraylength 指令 获取数组长度存储到栈顶，随后调用 istore_3 将数组长度存储到局部变量表中第 3 个位置，记为$len。

Java 虚拟机指令集使用不同的字节码来区分不同的操作数类型，比如 iconst_0、istore_1、iinc、if_icmplt 都只针对于 int 数据类型。

同时注意到 istore_3 和 istore 4 使用了不同形式的指令类型，它们的作用都是把栈顶元素存入到局部变量表的指定位置。对于 3 采用了 istore_3，它属于istore_<i>指令组，其中 i 只能是0 1 2 3。其实把istore_3 写成 istore 3 也能获取正确的结果，但是编译的字节码会变长，在字节码执行时也需要获取和解析 3 这个额外的操作数。

类似的做法还有 iconst_<i>，可以将 -1 0 1 2 3 4 5 压入操作数栈。这么做的目的是把使用最频繁的几个操作数融入到指令中，使得字节码更简洁高效

iconst_0 将整型值 0 加载到栈顶，随后将它存储到局部变量表第 4 个位置，记为 $i，写成伪代码就是

```
$array = numbers;
$len = $array.arraylength
$i = 0
```

- 18 ~ 34：是真正的循环体。首先加载 $i和 $len到栈顶，然后调用 if_icmpge 进行比较，如果 $i >= $len，直接跳转到指令 43，也就是 return，函数结束。如果$i < $len，执行循环体，加载$array、$i，然后 iaload 指令把下标为 $i 的数组元素加载到操作数栈上，随后存储到局部变量表下标为 5 的位置上，记为$item。随后调用 invokevirtual 指令来执行 record 方法
- 37 ~ 40：执行循环后的 $i 自增操作。

iinc 这个指令比较特殊，之前介绍的指令都是基于操作数栈来实现功能，iinc 是一个例外，它直接对局部变量进行自增操作，不要先入栈、加一、再出栈，因此效率非常高，适合循环结构。

这一段写成伪代码就是

```
@start: if ($i >= $len) return;
$item = $array[$i]
++ $i
goto @start
```

整段代码是不是非常熟悉，看看下面这个代码

```
for (int i = 0; i < numbers.length; i++) {
    calculator.record(numbers[i]);
}
```

由此可见，for(item : array) 就是一个语法糖，javac 会让它现出原形，回归到它的本质。

0x02 你不一定知道的 switch 底层实现

如果让你来设计一个 switch-case 的底层实现，你会如何来实现？是一个个 if-else 来判断吗？
实际上编译器将使用 tableswitch 和 lookupswitch 两个指令来生成 switch 语句的编译代码。为什么会有两个呢？这充分体现了效率上的考量。

```
int chooseNear(int i) {
    switch (i) {
        case 100: return 0;
        case 101: return 1;
        case 104: return 4;
        default: return -1;
    }
}
```

字节码如下：

```
0: iload_1
1: tableswitch   { // 100 to 104
          100: 36
          101: 38
          102: 42
          103: 42
          104: 40
      default: 42
}

36: iconst_0   // return 0
37: ireturn
38: iconst_1   // return 1
39: ireturn
40: iconst_4   // return 4
41: ireturn
42: iconst_m1  // return -1
43: ireturn
```

细心的同学会发现，代码中的 case 中并没有出现 102、103，为什么字节码中出现了呢?
编译器会对 case 的值做分析，如果 case 的值比较紧凑，中间有少

量断层或者没有断层，会采用 tableswitch 来实现 switch-case，有断层的会**生成一些虚假的 case 帮忙补齐连续**，这样可以实现 O(1) 时间复杂度的查找：因为 case 已经被补齐为连续的，通过游标就可以一次找到。

伪代码如下

```
int val = pop();                    // pop an int
from the stack
if (val < low || val > high) {  // if its less
than <low> or greater than <high>,
    pc += default;                  // branch to
default
} else {                            // otherwise
    pc += table[val - low];     // branch to
entry in table
}
```

我们来看一个 case 值断层严重的例子

```
int chooseFar(int i) {
    switch (i) {
        case 1: return 1;
        case 10: return 10;
        case 100: return 100;
        default: return -1;
    }
}
```

对应字节码

```
0: iload_1
1: lookupswitch  { // 3
             1: 36
            10: 38
           100: 41
       default: 44
}
```

如果还是采用上面那种 tableswitch 补齐的方式，就会生成上百个假 case，class 文件也爆炸式增长，这种做法显然不合理。lookupswitch应运而生，它的键值都是经过排序的，在查找上可以采用二分查找的方式，时间复杂度为 O(log n)

结论是：switch-case 语句 在 case 比较稀疏的情况下，编译器会使用 lookupswitch 指令来实现，反之，编译器会使用 tableswitch 来实现

0x03 小结

这篇文章以 for 和 switch-case 语句为例讲解了控制转移指令的实现细节，一起来回顾一下要点：

- 第一，for(item : array)语法糖实际上会改写为for (int i = 0; i < numbers.length; i++)的形式；
- 第二，switch-case 语句 在 case 稀疏程度不同的情况下会分别采用 lookupswitch 和 tableswitch 指令来实现。

0x04 思考

最后，给你留一个道作业题，switch-case 语句支持枚举类型，你能通过分析字节码写出其底层的实现原理吗?

欢迎你在留言区留言，和我一起讨论。