



微信扫一扫  
关注该公众号

收录于合集

#CPython

97个 >



前面我们介绍了虚拟机中常见的字节码指令，并且整个过程中，所有的指令都是从上往下顺序执行的，不涉及任何的跳转。但显然这是不够的，因为怎么能没有流程控制呢，像 if、for、while、try 等等，都属于流程控制，我们在编写代码的时候肯定也少不了它们的身影。

下面就先来看看 if 是怎么实现的吧。



if 语句算是最简单也是最常用的流程控制语句，那么它的字节码是怎么样的呢？当然我们这里的 if 语句指的是 if、elif、elif...、else 整体，里面的某个条件叫做该 if 语句的分支。

```
s = """
score = 90

if score >= 85:
    print("Good")

elif score >= 60:
    print("Normal")

else:
    print("Bad")
"""

if __name__ == '__main__':
    import dis
    dis.dis(compile(s, "<file>", "exec"))
```

古明地觉的 Python 小屋

反编译得到的字节码指令比较多，我们来慢慢分析。

```
# 为了阅读方便，源代码行号就不显示了
# 加载整数常量 90
0 LOAD_CONST           0 (90)
# 建立符号和对象的映射关系
2 STORE_NAME           0 (score)
# 加载变量 score
4 LOAD_NAME            0 (score)
# 加载整数常量 85
```

```

6 LOAD_CONST 1 (85)
# 将 score 和 85 进行 >= 操作
8 COMPARE_OP 5 (>=)
# 该指令的含义是如果为 False, 就进行跳转
# 后面的操作数 22 表示跳转到字节码偏移量为 22 的位置
# 显然就是 if 分支下面的 elif 分支
10 POP_JUMP_IF_FALSE 22
# 如果没有跳转, 能够走到这里, 说明 score >= 85 为真
# 然后执行该分支内的逻辑
# 这里要加载变量 print 和 字符串常量 "Good"
12 LOAD_NAME 1 (print)
14 LOAD_CONST 2 ('Good')
# 从栈里面弹出函数以及相关参数, 进行调用
# 后面的 1 表示传递的参数个数为 1
16 CALL_FUNCTION 1
# 将返回值从栈顶弹出去
18 POP_TOP
# if语句每次只会执行一个分支, 一旦执行了某个分支, 整个if语句就结束了
# 所以跳转到字节码偏移量为48的位置, 也就是整个 if 语句结束之后的位置
# 但需要注意这个 JUMP_FORWARD, 它表示向前跳转多少个偏移量
# 由于每个指令加上指令参数总共 2 字节, 所以 20 + 2 + 26 = 48
20 JUMP_FORWARD 26 (to 48)

# 显然这是 elif 分支, 它内部的逻辑和上面的 if 分支是类似的
>> 22 LOAD_NAME 0 (score)
24 LOAD_CONST 3 (60)
26 COMPARE_OP 5 (>=)
# 如果比较结果为 False, 直接跳转到下一个分支, 这里就是 else 了
28 POP_JUMP_IF_FALSE 40
# 如果比较结果为 True, 执行此分支内的代码
30 LOAD_NAME 1 (print)
32 LOAD_CONST 4 ('Normal')
34 CALL_FUNCTION 1
36 POP_TOP
# if 语句只会执行一个分支, 如果 elif 执行了, 那么整个语句就就结束了
# 此时同样要跳转, 至于跳转的位置, 显然每个分支都是一样的
# 都是 if 语句的结束位置, 或者说 if 语句的下一条语句的开始位置
38 JUMP_FORWARD 8 (to 48)

# else 分支, 上面的条件都不满足, 则执行 else
>> 40 LOAD_NAME 1 (print)
42 LOAD_CONST 5 ('Bad')
44 CALL_FUNCTION 1
46 POP_TOP
# 到此, 整个 if 语句就结束了
# 由于源码中 if 语句后面也没有内容了
# 所以再 return None 返回即可
>> 48 LOAD_CONST 6 (None)
50 RETURN_VALUE

```

古明地觉的Python小屋

我们看到字节码偏移量之前有几个>>这样的符号, 这是什么呢? 显然它是if语句中的每一个分支开始的地方, 当然最后的>>是返回值。

经过分析, 整个 if 语句的字节码指令还是很简单的。就是从上到下依次判断每一个分支, 如果某个分支条件成立, 就执行该分支的代码, 执行完毕后结束整个 if 语句; 分支不成立, 那么就跳转到下一个分支。

而核心指令就在于COMPARE\_OP、POP\_JUMP\_IF\_FALSE和JUMP\_FORWARD, 从结构上我们不难分析, 当前这三个指令的作用如下:

- COMPARE\_OP: 进行比较操作;
- POP\_JUMP\_IF\_FALSE: 跳转到下一个分支;
- JUMP\_FORWARD: 跳转到整个 if 语句结束后的第一条指令;

我们来分别解释一下。



首先是 COMPARE\_OP，我们看到 COMPARE\_OP 是有参数的，这里是 5，那么这个 5 代表啥呢？

```
// Include/object.h
#define Py_LT 0 //小于
#define Py_LE 1 //小于等于
#define Py_EQ 2 //等于
#define Py_NE 3 //不等于
#define Py_GT 4 //大于
#define Py_GE 5 //大于等于

// Include/opcode.h
enum cmp_op {PyCmp_LT=Py_LT, PyCmp_LE=Py_LE,
             PyCmp_EQ=Py_EQ, PyCmp_NE=Py_NE,
             PyCmp_GT=Py_GT, PyCmp_GE=Py_GE,
             PyCmp_IN, PyCmp_NOT_IN, PyCmp_IS,
             PyCmp_IS_NOT, PyCmp_EXC_MATCH, PyCmp_BAD};

// 可以看出
/*
 * PyCmp_IN = 6
 * PyCmp_NOT_IN = 7
 * PyCmp_IS = 8
 * PyCmp_IS_NOT = 9
 * PyCmp_EXC_MATCH = 10
 * PyCmp_BAD = 11
 */
```

古明地觉的 Python小屋

然后 COMPARE\_OP 内部主要是调用了 `cmp_outcome` 函数，这个函数我们之前简单看过，这里再来详细地介绍一下。

```
1 static PyObject *
2 cmp_outcome(int op, PyObject *v, PyObject *w)
3 {
4     int res = 0;
5     //op是操作数、或者说指令参数
6     //对于当前指令而言，用于判断"比较操作符"的种类
7     switch (op) {
8         //python的is, 等价于 C 的指针是否相等
9         case PyCmp_IS:
10             res = (v == w);
11             break;
12         //python的is not, 等价于 C 的指针是否不相等
13         case PyCmp_IS_NOT:
14             res = (v != w);
15             break;
16         //python的in, 等价于调用PySequence_Contains
17         //v in w <=> w.__contains__(v)
18         case PyCmp_IN:
19             res = PySequence_Contains(w, v);
20             if (res < 0)
21                 return NULL;
22             break;
23         //python的not in, 等价于调用PySequence_Contains再取反
24         case PyCmp_NOT_IN:
25             res = PySequence_Contains(w, v);
26             if (res < 0)
27                 return NULL;
28             res = !res;
29             break;
30         //python的异常匹配
31         case PyCmp_EXC_MATCH:
```

```

32 //这里判断给定的类是不是异常类(继承 BaseException)
33 //异常类一定要继承BaseException
34 //比如我们肯定不能except int as e
35 //如果是不是异常类, 那么一定是异常类组成的元组
36 if (PyTuple_Check(w)) {
37     Py_ssize_t i, length;
38     length = PyTuple_Size(w);
39     //如果是元组, 那么里面的每一个元素都要是异常类
40     for (i = 0; i < length; i += 1) {
41         PyObject *exc = PyTuple_GET_ITEM(w, i);
42         //如果出现了不是异常类的元素
43         //直接报错
44         if (!PyExceptionClass_Check(exc)) {
45             PyErr_SetString(PyExc_TypeError,
46                             CANNOT_CATCH_MSG);
47             return NULL;
48         }
49     }
50 }
51 //走到这里说明 w 不是元组
52 //如果不是元组, 那么就必须是异常类
53 else {
54     //进行检测, 如果不是则报错
55     if (!PyExceptionClass_Check(w)) {
56         PyErr_SetString(PyExc_TypeError,
57                         CANNOT_CATCH_MSG);
58         return NULL;
59     }
60 }
61 //上面相当于对 w 进行检测, 走到这里说明 w 是合法的
62 //然后判断指定的 w 能否匹配上实际发生的异常 v
63 res = PyErr_GivenExceptionMatches(v, w);
64 break;
65 default:
66     //当上面所有的case都不满足时, 执行 default
67     //显然它是专门用于大小的比较的
68     //传入两个对象、以及操作符, 即上面的Py_LT、Py_LE... 之一
69     return PyObject_RichCompare(v, w, op);
70 }
71 //Py_True和Py_False就相当于Python中的True和False
72 //本质上是一个PyLongObject
73 //这里根据res在C里面是否为真, 返回True和False
74 v = res ? Py_True : Py_False;
75 Py_INCREF(v);
76 return v;
77 }

```

所以`cmp_outcome`里面包含了很多的判断逻辑, 不同的操作执行不同的分支。而对于当前比较大小而言, 核心都隐藏在default分支调用的 `PyObject_RichCompare` 当中。

```

1 PyObject *
2 PyObject_RichCompare(PyObject *v, PyObject *w, int op)
3 {
4     PyObject *res;
5     //首先会对op进行判断, 要确保Py_LT <= op <= Py_GE
6     //即0 <= op <= 5, 要保证op是几个操作符中的一个
7     //因为如果不满足 0 <= op <= 5, 上面根本不会进入 default 分支
8     assert(Py_LT <= op && op <= Py_GE);
9     //首先 v 和 w 不能是 C 的空指针
10    //要确保它们都指向一个具体的 PyObject
11    //但是说实话, 底层的这些检测, 我们在Python的层面基本不会触发
12    if (v == NULL || w == NULL) {
13        if (!PyErr_Occurred())
14            PyErr_BadInternalCall();

```

```

15     return NULL;
16 }
17 //所以核心是下面的do_richcompare
18 //但是我们看到这里先调用了Py_EnterRecursiveCall
19 //原因和函数的递归有关, 比如我们在__eq__中又对self使用了==
20 //那么会不断调用__eq__, 这会引发无限递归
21 if (Py_EnterRecursiveCall(" in comparison"))
22     //所以Py_EnterRecursiveCall是让解释器追踪递归的深度的
23     //如果递归层数过多, 超过了指定限制
24     //那么能够及时抛出异常, 从递归中摆脱出来
25     return NULL;
26
27 //调用do_richcompare, 还是这三个参数, 得到比较的结果
28 res = do_richcompare(v, w, op);
29 //离开递归调用
30 Py_LeaveRecursiveCall();
31 //返回res
32 return res;
33 }

```

显然, PyObject\_RichCompare 也只是做一些类型检测而已, 大小比较的真正逻辑, 其实位于do\_richcompare里面, 我们继续看:

```

1 static PyObject *
2 do_richcompare(PyObject *v, PyObject *w, int op)
3 {
4     //比较函数, 对应__eq__、__le__、...
5     richcmpfunc f;
6     //比较结果
7     PyObject *res;
8     int checked_reverse_op = 0;
9
10    //如果type(v)和type(w)不一样、
11    //并且type(w)是type(v)的子类、
12    //并且type(w)中定义了tp_richcompare
13    if (v->ob_type != w->ob_type &&
14        PyType_IsSubtype(w->ob_type, v->ob_type) &&
15        (f = w->ob_type->tp_richcompare) != NULL) {
16        checked_reverse_op = 1;
17        //那么直接调用type(w)的tp_richcompare进行比较
18        res = (*f)(w, v, _Py_SwappedOp[op]);
19        if (res != Py_NotImplemented)
20            return res;
21        Py_DECREF(res);
22    }
23    //如果type(v)定义了tp_richcompare
24    if ((f = v->ob_type->tp_richcompare) != NULL) {
25        //调用type(v)的tp_richcompare方法
26        res = (*f)(v, w, op);
27        if (res != Py_NotImplemented)
28            return res;
29        Py_DECREF(res);
30    }
31
32    //type(w) 不是 type(v) 的子类、
33    //并且type(v)没有定义tp_richcompare、
34    //并且type(w)定义了tp_richcompare
35    if (!checked_reverse_op && (f = w->ob_type->tp_richcompare) != NULL)
36    {
37        //那么执行w的tp_richcompare
38        res = (*f)(w, v, _Py_SwappedOp[op]);
39        if (res != Py_NotImplemented)
40            return res;
41        Py_DECREF(res);

```

```

42     }
43
44     //上面的逻辑可能有点绕, 我们可以精炼一下, 当 v 和 w 在比较时:
45     //如果type(w)是type(v) 的子类, 那么优先调用type(w)的tp_richcompare
46     //否则按照顺序, type(v)和type(w)的tp_richcompare, 谁不为空就调用谁的
47     //如果都为空, 那么就走下面的逻辑了
48     switch (op) {
49         //比较两者是否相等
50         case Py_EQ:
51             res = (v == w) ? Py_True : Py_False;
52             break;
53         //两者是否不等
54         case Py_NE:
55             res = (v != w) ? Py_True : Py_False;
56             break;
57         //显然此时的两个对象只能判断相等或者不等
58         //如果其它操作那么显然是报错的, 下面的信息你一定很熟悉
59         default:
60             PyErr_Format(PyExc_TypeError,
61                          "'%s' not supported between instances of '%.100s' and '%.100s'",
62                          opstrings[op],
63                          v->ob_type->tp_name,
64                          w->ob_type->tp_name);
65             return NULL;
66     }
67     Py_INCREF(res);
68     //返回
69     return res;
70 }

```

虽然在Python里面用于比较的魔法方法有多个, 比如 `__eq__`、`__le__`、`__gt__` 等等, 但在底层, 它们都对应`tp_richcompare`, 只是执行时的操作符不同。所以我们实现任意一个用于比较的魔法方法, 底层都会实现`tp_richcompare`。

至于`tp_richcompare`具体支持多少种操作符, 就取决于我们实现了几个魔法方法, 比如我们只实现了 `__eq__`, 那么`tp_richcompare`的操作符就只能选择 `Py_EQ`, 否则就会抛出 `Py_NotImplemented`。

## POP\_JUMP\_IF\_FALSE

`COMPARE_OP` 执行完之后会将比较的结果压入运行时栈, 而该指令则是将结果从栈顶弹出并判断真假。如果为假, 那么跳到下一个分支, 否则执行此分支的代码。

同理还有`POP_JUMP_IF_TRUE`, 它表示当比较结果为真时, 跳到下一个分支, 否则执行此分支的代码。

我们来看一下`POP_JUMP_IF_FALSE` 的具体实现吧。

```

1 case TARGET(POP_JUMP_IF_FALSE): {
2     PREDICTED(POP_JUMP_IF_FALSE);
3     //从栈顶将比较结果弹出
4     PyObject *cond = POP();
5     int err;
6     //如果为真, 直接执行下一条指令
7     if (cond == Py_True) {
8         Py_DECREF(cond);

```



```

9     FAST_DISPATCH();
10 }
11 //如果为假, 跳到下一个分支
12 //从这里我们看到, 指令跳转是由JUMPTO实现的
13 if (cond == Py_False) {
14     Py_DECREF(cond);
15     JUMPTO(oparg);
16     FAST_DISPATCH();
17 }
18 //都不满足, 说明cond不是一个布尔类型
19 //调用PyObject_IsTrue获取布尔值
20 //此处算是解释了为什么 if cond 和 if bool(cond)是等价的
21 //因为当 cond 不是布尔类型时, 会自动获取布尔值
22 err = PyObject_IsTrue(cond);
23 Py_DECREF(cond);
24 //err > 0 表示为真, 那么什么也不做, 执行下一条指令
25 if (err > 0)
26     ;
27 // err == 0 表示为假, 那么跳到下一个分支
28 else if (err == 0)
29     JUMPTO(oparg);
30 // 否则说明出错了
31 else
32     goto error;
33 // continue, 会跳转到 for 循环所在位置
34 // 除此之外还有一个 FAST_DISPATCH
35 // 它表示跳转到标签 fast_next_opcode 所在位置
36 // 不管哪一种, 都表示要执行下一条字节码指令
37 DISPATCH();
38 }

```

我们看到该指令能够实现跳转, 是因为内部调用JUMPTO。那么问题来了, 它和上面一开始的 JUMP\_FORWARD 有什么区别呢? 下面来进行介绍。

## JUMP\_FORWARD



先来看一看字节码:

```

1 case TARGET(JUMP_FORWARD): {
2     JUMPBY(oparg);
3     FAST_DISPATCH();
4 }

```

我们看到又出现了一个 JUMPBY, 后续在介绍for循环的时候还会看到一个 JUMP\_ABSOLUTE, 那么它们之间有什么关系呢? 这里直接说结论:

- JUMP\_FORWARD: 这是一个指令, 本质上就是一个整数。执行该指令对应的 case 分支时会调用 JUMPBY, 表示从当前位置跳转指定的偏移量;
- JUMP\_ABSOLUTE: 也是一个指令, 执行时会调用 JUMPTO, 表示跳转到某一指定位置。当然, POP\_JUMP\_IF\_FALSE内部也调用了JUMPTO。

所以核心就在于 JUMPTO 和 JUMPBY 的区别, 它们都是宏:

```

1 #define JUMPTO(x) (next_instr = first_instr + (x) / sizeof(_Py_CODEUNIT))
2 #define JUMPBY(x) (next_instr += (x) / sizeof(_Py_CODEUNIT))

```

如果我们将 JUMPBY 换一种写法, 区别就更明显了。

```

1 #define JUMPTO(x) (next_instr = first_instr + (x) / sizeof(_Py_CODEUNIT))

```

```
2 # Fine JUMPBY(x) (next_instr = next_instr + (x) / sizeof(_Py_CODEUNIT))
```

当 JUMPTO 的参数为 50 时，它表示跳转到 co\_code 中偏移量为 50 的位置，也就是第 26 条指令；并且 JUMPTO 的执行结果，与当前指令的偏移量无关，它属于绝对跳转。无论当前指令的偏移量为多少，执行效果始终是跳转到第 26 条指令。

而当 JUMPBY 的参数是 50 时，它表示从当前指令的结束位置、或者说下一条指令的开始位置向前跳转 50 个偏移量。显然 JUMPBY 的执行结果，与当前指令的偏移量是有关的，它属于相对跳转。

并且 JUMPTO 既可以向前跳转（偏移量增大），也可以向后跳转（偏移量减小）；而 JUMP\_FORWARD 只能向前跳转。

假设参数为  $n$ ，当前指令的偏移量为  $m$ 。对于 JUMPTO 而言，跳转之后的偏移量始终是  $n$ ，如果  $m < n$  就是向前跳转， $m > n$  就是向后跳转；对于 JUMP\_FORWARD 而言，跳转之后的偏移量等于  $m + 2 + n$ ，它只能向前跳转，并且跳转之后的偏移量不仅取决于  $n$ ，还取决于  $m$ 。

以上就是指令的跳转，在后续介绍循环的时候，我们还会见到。然后可能有人好奇，为什么 JUMPTO 和 JUMPBY 里面的  $x$  要除以 `sizeof(_Py_CODEUNIT)`？

首先 `_Py_CODEUNIT` 是一个 `uint16_t`，而 `next_instr` 的类型是 `_Py_CODEUNIT *`，这意味着 `next_instr` 每自增一次，就会跳两个字节，所以还要将  $x / 2$ 。假设偏移量为 50 字节，那么只需要加上 25 即可，如果不除以 2，偏移量就变成 100 了。

由于每条指令都会带有一个参数，所以 `next_instr` 每次需要跳两个字节，才能指向下一条指令。因此 `next_instr` 是一个 `_Py_CODEUNIT *`，这样只需自增一次，便可跳两个字节。

然后通过 `*next_instr` 便可取出待执行的指令，由于是两个字节，所以此时也包含了指令参数。通过宏 `_Py_OPCODE` 可以拿到指令，通过宏 `_Py_OPARG` 可以拿到指令参数。

```
1 #ifdef WORDS_BIGENDIAN
2 #   define _Py_OPCODE(word) ((word) >> 8)
3 #   define _Py_OPARG(word) ((word) & 255)
4 #else
5 #   define _Py_OPCODE(word) ((word) & 255)
6 #   define _Py_OPARG(word) ((word) >> 8)
7 #endif
```



在 Python 中，有一些字节码指令通常都是按照顺序出现的，通过上一个字节码指令直接预测下一个字节码指令是可能的。比如 `COMPARE_OP` 的后面通常都会紧跟着 `POP_JUMP_IF_TRUE` 或者 `POP_JUMP_IF_FALSE`，这在上面的字节码中可以很清晰的看到。

然后再来回顾一下 `COMPARE_OP` 的具体实现：

```
1 case TARGET(COMPARE_OP): {
2     PyObject *right = POP();
3     PyObject *left = TOP();
4     PyObject *res = cmp_outcome(tstate, oparg, left, right);
5     Py_DECREF(left);
6     Py_DECREF(right);
7     SET_TOP(res);
8     // 上面的部分应该不需要解释了
9     if (res == NULL)
10         goto error;
```



```

11 //重点是这里, PREDICT 表示指令预测
12 PREDICT(POP_JUMP_IF_FALSE);
13 PREDICT(POP_JUMP_IF_TRUE);
14 DISPATCH();
15 }

```

为什么要有这样的一个预测功能呢？答案是当预测成功时，能够省去很多无谓的操作，使得执行效率大幅提高。

**PREDICTED(POP\_JUMP\_IF\_FALSE)**；实际上就是检查下一条待处理的字节码指令是否是 POP\_JUMP\_IF\_FALSE。如果是，那么程序会直接跳转到 PRED\_POP\_JUMP\_IF\_FALSE 那里。

```

1 #if defined(DYNAMIC_EXECUTION_PROFILE) || USE_COMPUTED_GOTOS
2 #define PREDICT(op)          if (0) goto PRED_##op
3 #else
4 #define PREDICT(op) \
5     do{ \
6         _Py_CODEUNIT word = *next_instr; \
7         opcode = _Py_OPCODE(word); \
8         if (opcode == op){ \
9             oparg = _Py_OPARG(word); \
10            next_instr++; \
11            goto PRED_##op; \
12        } \
13    } while(0)
14 #endif
15 #define PREDICTED(op)        PRED_##op:

```

我们可以将这些宏简化一下：

```

1 if (_Py_OPCODE(*next_instr) == POP_JUMP_IF_FALSE)
2     goto PRED_POP_JUMP_IF_FALSE;
3 if (_Py_OPCODE(*next_instr) == POP_JUMP_IF_TRUE)
4     goto PRED_POP_JUMP_IF_TRUE

```

指令跳跃的目的是为了绕过一些无谓的操作，直接进入 POP\_JUMP\_IF\_TRUE 或者 POP\_JUMP\_IF\_FALSE 指令对应的 case 语句。

```

case TARGET(POP_JUMP_IF_FALSE): {
    PREDICTED(POP_JUMP_IF_FALSE);
    PyObject *cond = POP();
    int err;
    if (cond == Py_True) {
        Py_DECREF(cond);
        FAST_DISPATCH();
    }
    if (cond == Py_False) {
        Py_DECREF(cond);
        JUMPTO(oparg);
        FAST_DISPATCH();
    }
    err = PyObject_IsTrue(cond);
    Py_DECREF(cond);
    if (err > 0)
        ;
}

```

预测成功，直接跳转到这里

古明地觉的 Python 小屋

我们看到当预测成功时，直接就跳转到下一条指令执行了。而 COMPARE\_OP 后面一般都会跟着 POP\_JUMP\_IF\_FALSE 或 POP\_JUMP\_IF\_TRUE，所以这就是指令预测。

收录于合集 #CPython 97

[← 上一篇](#)

《源码探秘 CPython》53. 流程控制语句  
for、while 是怎么实现的？

[下一篇 →](#)

《源码探秘 CPython》51. 虚拟机的一般表  
达式

喜欢此内容的人还喜欢

A tour of gRPC: 01 - 基础理论  
BUG侦探



监控操作系统指标(node\_exporter)  
AmCoder



PASTIS: 从HiC矩阵推断染色体3D结构  
生信杂记

