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

原创 古明地觉 古明地觉的编程教室 2022-03-22 08:30

收录于合集
#CPython
97个 >

楔子

在介绍 if 语句的时候，我们已经见识了最基本的控制，但是我们发现 if 只能向前跳转，不管是哪个分支，都是通过 JUMP_FORWARD。而下面介绍的 for、while 循环，指令是可以回退的，也就是向后跳转。

另外，在 if 语句的分支中，无论哪个分支，其指令的跳跃距离通常都是当前指令与目标指令的距离，相当于向前跳了多少步。那么指令回退时，是不是相当于向后跳了多少步呢？带着疑问，我们来往下看。

for

我们来看一个简单的 for 循环的字节码。

```
s = """
lst = [1, 2]
for item in lst:
    print(item)
"""


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


反编译之后，字节码指令如下：

```
  # 加载整数常量 1
  0 LOAD_CONST               0 (1)
  # 加载整数常量 2
  2 LOAD_CONST               1 (2)
  # 构建 PyListObject 对象，容量为 2
  4 BUILD_LIST               2
  # 使用符号 "lst" 保存
  6 STORE_NAME               0 (lst)

  # 加载变量 lst
  8 LOAD_NAME                0 (lst)

  # 获取对应的迭代器
 10 GET_ITER

  # 开始 for 循环，将里面的元素依次迭代出来
  # 循环结束后跳转到字节码偏移量为26的位置，也就是第14条指令
  # 但是参数是 12，因此我们看到 for 循环结束之后，使用的也是相对跳转
  # 12 + 2 + 12 = 26
>>   12 FOR_ITER                12 (to 26)
  # 元素迭代出来之后，使用符号 "item" 保存
```

```
14 STORE_NAME               1 (item)

 # 加载函数 print，参数 item，当然它们本质上都是变量
16 LOAD_NAME                2 (print)
18 LOAD_NAME                1 (item)
 # 函数调用，参数个数是 1
20 CALL_FUNCTION            1
 # 将 print 的返回值从栈顶弹出，由于这里没有使用变量接收，所以是 POP_TOP
 # 如果使用变量接收了，比如 res = print(item)，那么该指令就是 STORE_NAME
22 POP_TOP
 # 到此 for 循环的一次遍历就结束了，然后采用绝对跳转
 # 跳转到偏移量为 12 的位置，开始下一次循环
24 JUMP_ABSOLUTE            12
 # 当循环结束时，整个程序就执行结束了
 # 最后 return None
>> 26 LOAD_CONST               2 (None)
28 RETURN_VALUE
```


我们直接从 10 GET_ITER 开始看起，首先 for 循环遍历一个对象的时候，要满足后面的对象是一个可迭代对象，然后会先调用这个对象的__iter__方法，把它变成一个迭代器。再不断地调用这个迭代器的__next__方法，一步一步将里面的值全部迭代出来，当出现StopIteration异常时，for循环捕捉，最后退出。

另外，我们说 Python 里面是先有值，后有变量，for 循环也不例外。循环的时候，先将 lst 对应的迭代器中的元素迭代出来，然后再让变量 item 指向。所以字节码中先是 12 FOR_ITER，然后才是 14 STORE_NAME。

因此包含10个元素的迭代器，需要迭代11次才能结束。因为 for 循环事先是不知道迭代10次就能结束的，它需要再迭代一次发现没有元素可以迭代、从而抛出StopIteration异常、再进行捕捉，之后才能结束。

> for 循环遍历可迭代对象时，会先拿到对应的迭代器，那如果遍历的就是一个迭代器呢？答案是依旧调用 __iter__，只不过由于本身就是一个迭代器，所以返回的还是其本身。

将元素迭代出来之后，就开始执行 for 循环体的逻辑了。

执行完之后，通过 JUMP_ABSOLUTE 跳转到字节码偏移量为12、也就是 FOR_ITER 的位置开始下一次循环。这里我们发现它没有跳到 GET_ITER 那里，所以可以得出结论，for 循环在遍历的时候只会创建一次迭代器。

下面来看指令对应的具体逻辑：

```
case TARGET(GET_ITER): {
    //在 GET_ITER之前，LOAD_NAME已经将 lst 压入运行时栈
    //此处会从运行时栈的顶端获取压入的 lst
    PyObject *iterable = TOP();
    //调用PyObject_GetIter，获取对应的迭代器
    //这个函数我们在介绍迭代器的时候已经说过了
    PyObject *iter = PyObject_GetIter(iterable);
    Py_DECREF(iterable);
    //将变量 iter 设置为新的栈顶元素
    SET_TOP(iter);
    if (iter == NULL)
        goto error;
    //指令预测，Python 认为 GET_ITER 的下一条指令
    //很有可能是 FOR_ITER，其次是 CALL_FUNCTION
    PREDICT(FOR_ITER);
    PREDICT(CALL_FUNCTION);
    //continue
    DISPATCH();
}
```

当创建完迭代器之后，就正式进入 for 循环了。所以从 FOR_ITER 开始，进入了Python虚拟机层面上的 for循环。

> 源代码中的 for循环，在虚拟机层面也一定对应着一个相应的循环控制结构。因为无论进行怎样的变换，都不可能在虚拟机层面利用顺序结构来实现源码层面上的循环结构，这也可以看成是程序的拓扑不变性。

我们来看一下 FOR_ITER 指令对应的具体实现：

```
case TARGET(FOR_ITER): {
    // 指令预测，预测成功会直接跳转到此处
    PREDICTED(FOR_ITER);
    /* 从栈顶获取 iterator 对象(指针) */
    PyObject *iter = TOP();
    //调用迭代器类型对象的 tp_iternext方法
    //将迭代器内的元素迭代出来
    PyObject *next = (*iter->ob_type->tp_iternext)(iter);
    //如果next不为NULL，那么将元素压入运行时栈
    //等待被赋值给 for循环的变量
    if (next != NULL) {
        PUSH(next);
        //依旧是指令预测
        //对于我们当前这个例子来说，显然预测失败了
        //这里是 STORE_FAST，而例子中是 STORE_NAME
        //原因是Python虚拟机认为 for循环更有可能在局部作用域中出现
        //而我们当前的for循环位于全局作用域
        PREDICT(STORE_FAST);
        PREDICT(UNPACK_SEQUENCE);
        //continue
        DISPATCH();
    }
    //tstate指的是线程状态对象，我们会后面分析
    //这里表示如果出现了异常
    if (_PyErr_Occurred(tstate)) {
        //并且还不是StopIteration
        //那么证明执行的时候出错了
        if (!_PyErr_ExceptionMatches(tstate, PyExc_StopIteration)) {
            goto error;
        }
        //否则说明是StopIteration，意味着迭代就结束了
        else if (tstate->c_tracefunc != NULL) {
            call_exc_trace(tstate->c_tracefunc, tstate->c_traceobj, tsta
te, f);
        }
        //_PyErr_Clear会将异常回溯栈清空
        //从Python的层面来看，等于是for循环将StopIteration自动捕捉了
        _PyErr_Clear(tstate);
    }
    //走到这里说明循环结束了
    STACK_SHRINK(1);
    Py_DECREF(iter);
    JUMPBY(oparg);
    PREDICT(POP_BLOCK);
    DISPATCH();
}
```

在将迭代出来的元素压入运行时栈之后(预测失败)，会执行 STORE_NAME。然后虚拟机将沿着字节码指令的顺序一条一条地执行下去，从而完成输出的动作。

但是我们知道，for循环中肯定会有指令回退的动作。从字节码中也看到了，for循环遍历一次之后，会再次跳转到FOR_ITER，而跳转所使用的指令就是JUMP_ABSOLUTE。这个在介绍 if 的时候，我们已经说过了，它表示绝对跳转。

```
case TARGET(JUMP_ABSOLUTE): {
    PREDICTED(JUMP_ABSOLUTE);
    //显然这里的oparg表示字节码偏移量
    //表示直接跳转到偏移量为oparg的位置上
    JUMPTO(oparg);
#if FAST_LOOPS
    FAST_DISPATCH();
```

```
#else
    DISPATCH();
#endif
}

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

可以看到和 if 不一样，for循环使用的是绝对跳转。JUMP_ABSOLUTE 是强制设置 next_instr 的值，将 next_instr 设定到距离f->f_code->co_code开始地址的某一特定偏移量的位置。这个偏移量由JUMP_ABSOLUTE的指令参数决定，所以该参数就成了for循环中指令回退动作的最关键的一点。

天下没有不散的宴席，随着迭代的进行，for循环总有退出的那一刻，而这个退出的动作只能落在 FOR_ITER 的身上。在 FOR_ITER 指令执行的过程中，如果遇到了StopIteration，就意味着迭代结束了。

这个结果将导致Python虚拟机会将迭代器从运行时栈中弹出，同时执行一个 JUMPBY 的动作，向前跳跃，在字节码的层面是向下，也就是偏移量增大的方向。

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

case TARGET(FOR_ITER): {
    /*
    ...
    ...
    ...
    */
    //走到这里说明循环结束了
    //STACK_SHRINK会对栈进行收缩
    //此处就相当于将迭代器(指针)从运行时栈中弹出
    STACK_SHRINK(1);
    //减少迭代器的引用计数
    Py_DECREF(iter);
    //进行跳转，此时采用相对跳转
    //显然会跳转到整个for循环下面的第一条语句的位置
    JUMPBY(oparg);
    PREDICT(POP_BLOCK);
    DISPATCH();
}
```

所以 for 循环结束时采用的是相对跳转，进入下一次循环时采用的是绝对跳转。

以上就是 for 循环的原理，从字节码的角度去理解它，是不是别有一番风味呢?

看完了 for，再来看 while 就简单了。不仅如此，我们还要分析两个关键字：break、continue，当然goto就别想了。

```
s = """
a = 0
while a < 10:
    a += 1
    if a == 5:
        continue
```

```
    if a == 7:
        break
    print(a)
"""


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


反编译之后，指令如下，和 for有很多是类似的。

```
    # 加载整数常量 0
    0 LOAD_CONST               0 (0)
    # 使用变量 a 存储
    2 STORE_NAME               0 (a)
    # 进入 while 循环了，加载变量a
>>  4 LOAD_NAME                0 (a)
    # 加载整数常量 10
    6 LOAD_CONST               1 (10)
    # 比较操作
    8 COMPARE_OP               0 (<)
    # 为 False 直接结束循环
    # 跳转到字节码偏移量为 50 的位置，也就是第 26 条指令
    # 或者说整个 while 语句结束后的位置
   10 POP_JUMP_IF_FALSE       50

    # 如果为 True 则进入循环
    # 首先加载变量 a
   12 LOAD_NAME                0 (a)
    # 加载整数常量 1
   14 LOAD_CONST               2 (1)
    # 执行 a += 1 操作
    # 会先执行 a + 1
   16 INPLACE_ADD
    # 然后再让变量 a 指向相加之后的结果
   18 STORE_NAME               0 (a)

    # 下面执行 if 语句
    # 加载变量 a 和常量 5，进行 == 比较操作
   20 LOAD_NAME                0 (a)
   22 LOAD_CONST               3 (5)
   24 COMPARE_OP               2 (==)
    # 如果为False，那么直接跳转到偏移量为 30 的位置
    # 也就是当前 if 语句的下一条指令，然后又会进入新的 if 语句
   26 POP_JUMP_IF_FALSE       30
    # 如果a == 5成立，那么绝对跳转，跳到字节码偏移量为 4 的位置
    # 所以 continue 本质上是一个绝对跳转，目标是循环开始的地方
   28 JUMP_ABSOLUTE            4

    # 走到这里说明 a == 5 不成立，于是判断 a == 7
    # 加载变量 a、常量 7，进行 == 比较
>>  30 LOAD_NAME                0 (a)
   32 LOAD_CONST               4 (7)
   34 COMPARE_OP               2 (==)
    # 如果为False，跳转到偏移量为 40 的位置，也就是 print(a)
   36 POP_JUMP_IF_FALSE       40
    # 如果 a == 5 成立，那么跳转到字节码偏移量为50的地方
    # 因为是 break，所以相当于结束循环了
   38 JUMP_ABSOLUTE           50

    # 到这里，说明上面的两个 if 都不成立
    # 加载变量 print、变量 a，进行函数调用
>>  40 LOAD_NAME                1 (print)
```

```
 42 LOAD_NAME                0 (a)
 44 CALL_FUNCTION            1
  # 从栈顶弹出返回值
 46 POP_TOP
  # 走到这里说明 while 循环执行一圈了
  # 那么再度跳转到 while a < 10 的地方
 48 JUMP_ABSOLUTE            4

  # 最后 while 循环结束，return None
>>   50 LOAD_CONST               5 (None)
 52 RETURN_VALUE
```


有了 for 循环，再看 while 循环就简单多了，整体逻辑和 for 高度相似，当然里面还结合了 if。

另外我们看到break和continue本质上都是一个JUMP_ABSOLUTE，都是通过绝对跳转实现的。break 是跳转到 while循环结束后的第一条指令；continue 则是跳转到 while 循环的开始位置。

然后执行一圈之后，再通过JUMP_ABSOLUTE跳转回去，显然此时的指令参数和 continue 是一样的，都是 while 循环的开始位置。

当循环条件不满足的时候，指令 POP_JUMP_IF_FALSE 发现结果为 False，直接结束循环，此时的指令参数和 break 是一样的，都是跳转到 while 循环的结束位置，或者说 while 循环的下一条指令的开始位置。

所以 while 事实上比 for 还是要简单一些的。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》54. 异常是怎么实现的？虚拟机是如何将异常抛出去的？

下一篇 ›
《源码探秘 CPython》52. 流程控制语句 if 是怎么实现的？

喜欢此内容的人还喜欢

python 7天进阶之路-对象和json转换
缪斯之子

[系列]微服务·深入理解 gRPC - Part2
走向架构师的每一天

Abaqus python脚本开发 第三章 各类指令的方法对象变量 (3)
山石结构