《源码探秘 CPython》55. 虚拟机是如何捕获异常的?

原创 古明地觉 古明地觉的编程教室 2022-03-25 09:00

收录于合集
#系统与协议 1 #CPython 97

楔子

上一篇文章我们介绍了Python的异常是怎么实现的，抛出异常这个动作在虚拟机层面上是怎样的一个行为，以及虚拟机在处理异常时候的**栈帧展开**行为。

既然虚拟机内建的处理异常的动作我们已经了解了，那么接下来就看看异常捕获是如何实现的，还有它又是如何影响虚拟机的异常处理流程的。毕竟在大部分情况下，我们都不会将异常抛出去，而是将它捕获起来。

异常捕获语句

这里先来回顾一下异常捕获语句，首先一个完整的异常捕获语句如下：

```
try:
    pass
except IndexError as e:
    pass
except Exception as e:
    pass
else:
    pass
finally:
    pass
```

情况可以分为以下几种：

1）如果 try 里面的代码在执行时没有出现异常，那么会执行 else ，然后执行 finally；

```
try:
    print("我是 try")
except Exception as e:
    print("我是 except")
else:
    print("我是 else")
finally:
    print("我是 finally")
"""
我是 try
我是 else
我是 finally
"""
```

2）如果 try 里面的代码在执行时出现异常了（异常会被设置在线程状态对象中），那么会依次判断 except（可以有多个）能否匹配发生的异常。如果某个 except 将异常捕获了，那么会将异常给清空，然后执行finally；

```
try:
    raise IndexError("IndexError Occurred")
except ValueError as e:
    print("ValueError 匹配上了异常")
except IndexError as e:
    print("IndexError 匹配上了异常")
except Exception as e:
    print("Exception 匹配上了异常")
else:
    print("我是 else")
finally:
    print("我是 finally")
"""
IndexError 匹配上了异常
我是 finally
"""
```

except 子句可以有很多个，发生异常时会从上往下依次匹配。但是注意：多个 except 子句最多只有一个被执行，比如当前的 IndexError 和 Exception 都能匹配发生的异常，但是只会执行匹配上的第一个except子句。

> 只要发生异常了，else 就不会被执行了。不管 except 有没有将异常捕获到，都不会执行 else，因为 else 只有在 try 里面没有发生异常的时候才会执行。

3）如果 try 里面的代码在执行时出现异常了，但 except 没有将异常捕获掉，那么异常仍然被保存在线程状态对象中，然后执行finally。如果 finally 子句中没有出现 return、break、continue 等关键字，再将异常抛出来；

```
try:
    raise IndexError("IndexError Occurred")
except ValueError:
    print("ValueError 匹配上了异常")
finally:
    print("我是 finally")
"""
我是 finally
Traceback (most recent call last):
  File "......", line 2, in <module>
    raise IndexError("IndexError Occurred")
IndexError: IndexError Occurred
"""
```

except 没有将异常捕获掉，所以执行完 finally 之后，异常又被抛出来了。但如果 finally 里面出现 return、break、continue 等关键字，也不会抛出异常，因为会将异常丢弃掉。

```
def f():
    try:
        raise IndexError("IndexError Occurred")
    except ValueError:
        print("ValueError 匹配上了异常")
    finally:
        print("我是 finally")
        return

f()
"""
我是 finally
"""
```

由于 finally 里面出现了 return，所以异常并没有发生，而是被丢弃掉了。这个特性相信有很多小伙伴之前还是没有发现的。

然后 try、except、else、finally 这几个关键字不需要同时出现，可以有以下几种组

合：try...except、try...finally、try...except...else、try...except...else...finally。

这里需要提一下 try...finally 这种情况，显然它没有对异常进行捕获，那么当finally里面没有出现return、break、continue的时候，try 里面的异常一定是会抛出来的。一般这种情况主要用在异常发生时，你希望能够抛出来，但是又想做一些资源清理等善后工作之类的场景。

> finally 始终会被执行。

try、except、else、finally 的返回值

我们看看这四者的返回值之间的关系：

```
def retval():
    try:
        return 123
    except Exception:
        return 456

print(retval())  # 123
```

由于没有发生异常，所以返回了try指定的返回值。

```
def retval():
    try:
        return 123
    except Exception:
        return 456
    else:
        return 789

print(retval())  # 123
```

虽然我们指定了 else，但是 try 里面已经执行 return 了，所以打印的仍是 try 的返回值。

```
def retval():
    try:
        1 / 0
        return 123
    except Exception:
        return 456

print(retval())  # 456
```

此时发生异常，所以返回了except指定的返回值。

```
def retval():
    try:
        1 / 0
        return 123
    except Exception:
        return 456
    else:
        return 789
```

```
9
10 print(retval())  # 456
```

一旦发生异常，else 就不可能被执行，所以此时仍然返回 456。

```
def retval():
    try:
        return 123
    except Exception:
        return 456
    finally:
        pass

print(retval())  # 123
```

因为finally中没有指定返回值，所以此时返回的是 123。

```
def retval():
    try:
        return 123
    except Exception:
        return 456
    finally:
        return

print(retval())  # None
```

一旦finally中出现了return，那么返回的都是finally指定的返回值。并且此时即便出现了没有捕获的异常，也不会报错，因为会将异常丢弃掉。

```
def retval():
    try:
        return 123
    except Exception:
        return 456
    finally:
        pass
    return 789

print(retval())  # 123
```

函数一旦return，就表示要返回了，但如果这个return是位于出现了finally的异常捕获语句中，那么会先执行finally，然后再返回。所以最后的一个 return 789 是不会执行的，因为已经出现 return 了，finally 执行完毕之后就直接返回了。

> 但要是finally里面也出现了return，那么会将返回值给替换掉，否则就还是之前的返回值。

```
def retval():
    try:
        pass
    except Exception:
        return 456
    finally:
        pass
    return 789

print(retval())  # 789
```

没有异常，所以except里的return没任何卵用，而try和finally中也没有return，因此返

回 789。

> 一个简单的异常捕获，总结起来还稍微有点绕呢。

从 Python 的层面理解完异常捕获之后，再来看看虚拟机是如何实现这一机制的？想要实现这一点，还是得从字节码入手。

异常捕获相关的字节码

来看一段代码：

```
s = """
try:
    raise Exception("raise an exception")
except Exception as e:
    print(e)
finally:
    print("finally code")
"""

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

抛异常有两种方式，一种是虚拟机执行的时候出现运行时错误抛出异常，另一种是使用 raise 关键字手动抛出异常。

这里我们就用第二种方式，来看一下它的字节码。

```
      0 SETUP_FINALLY           60 (to 62)
      2 SETUP_FINALLY           12 (to 16)

      4 LOAD_NAME                1 (Exception)
      6 LOAD_CONST               1 ('raise an exception')
      8 CALL_FUNCTION            1
     10 RAISE_VARARGS            1
     12 POP_BLOCK
     14 JUMP_FORWARD            42 (to 58)

>>   16 DUP_TOP
     18 LOAD_NAME                1 (Exception)
     20 COMPARE_OP              10 (exception match)
     22 POP_JUMP_IF_FALSE       56
     24 POP_TOP
     26 STORE_NAME               2 (e)
     28 POP_TOP
     30 SETUP_FINALLY           12 (to 44)

     32 LOAD_NAME                0 (print)
     34 LOAD_NAME                2 (e)
     36 CALL_FUNCTION            1
     38 POP_TOP
     40 POP_BLOCK
     42 BEGIN_FINALLY
>>   44 LOAD_CONST               2 (None)
     46 STORE_NAME               2 (e)
     48 DELETE_NAME              2 (e)
     50 END_FINALLY
```

```
30      52 POP_EXCEPT
31      54 JUMP_FORWARD             2 (to 58)
32  >>  56 END_FINALLY
33  >>  58 POP_BLOCK
34      60 BEGIN_FINALLY
35
36  >>  62 LOAD_NAME                0 (print)
37      64 LOAD_CONST               0 ('finally code')
38      66 CALL_FUNCTION            1
39      68 POP_TOP
40      70 END_FINALLY
41      72 LOAD_CONST               2 (None)
42      74 RETURN_VALUE
43
```

首先这个指令集比较复杂，因为要分好几种情况。try 里面没有出现异常；try里面出现了异常，但是 except 没有捕获到；try 里面出现了异常，except 捕获到了。但我们知道无论是哪种情况，都要执行finally。

先看上面的SETUP_FINALLY指令：

```
case TARGET(SETUP_FINALLY): {
    PyFrame_BlockSetup(f, SETUP_FINALLY, INSTR_OFFSET() + oparg,
                       STACK_LEVEL());
    DISPATCH();
}
```

该指令内部仅仅是调用了PyFrame_BlockSetup函数，但是参数我们需要解释一下。

首先 f 指的是当前的栈帧对象；`SETUP_FINALLY`是指令本身，一个整数，值为 122。这两个比较简单，没什么可说的。

然后`INSTR_OFFSET()`表示下一条待执行指令的偏移量，它是一个宏，定义在 ceval.c 中：

```
#define INSTR_OFFSET()  \
    (sizeof(_Py_CODEUNIT) * (int)(next_instr - first_instr))
```

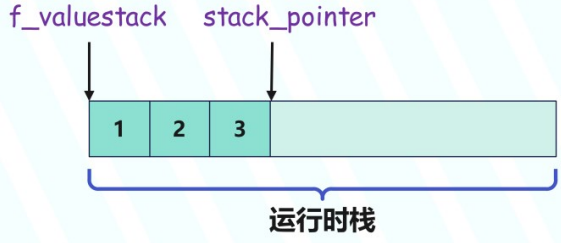
如果再加上 oparg，那么会对应 finally 子句（或 except 字句）。

最后是 `STACK_LEVEL()`，它也是一个宏，定义在 ceval.c 中：

```
#define STACK_LEVEL()     ((int)(stack_pointer - f->f_valuestack))
```

这里可能有人忘记了`f->f_valuestack`是干什么的了，我们来回顾一下。

栈帧对象有一个 f_localsplus 成员，它是一个数组，但是这段连续内存被分成了四份，分别用于局部变量、cell 对象、free 对象、运行时栈。这四部分虽然是连续的，但彼此之间互不打扰，都只会在属于自己的内存区域里玩耍。正所谓鸡犬相闻，但又老死不相往来。



然后是运行时栈，虽然叫做栈，但它本质上是一个数组。而 f_valuestack 指向运行时栈的栈底位置，从栈底到栈顶的地址又是增大的，因此它位于 free 对象和运行时栈的交界处。

stack_pointer 表示栈指针，用于操作栈里面元素的，比如 PUSH、POP、TOP、SECOND 等等这些宏，都是操作的 stack_pointer。比如当前栈里面有三个元素，如果想弹出一个元素，那么 *--stack_pointer 即可；如果想压入一个元素 v，那么 *stack_pointer++ = v 即可。所以随着元素的压入和弹出，栈顶也在不断变化，但无论如何，stack_pointer 始终指向运行时栈的栈顶。

另外栈帧还有一个成员叫 f_stacktop，它也指向运行时栈的栈顶，因此 stack_pointer 后续还会赋值给 f_stacktop。

最后，由于栈里面的元素都是 PyObject *，所以 f_stacktop、f_valuestack、stack_pointer 都是二级指针 PyObject **。只不过为了清晰，我们图上画的是 1 2 3，但其实它们都是指针，指向的是 1 2 3。

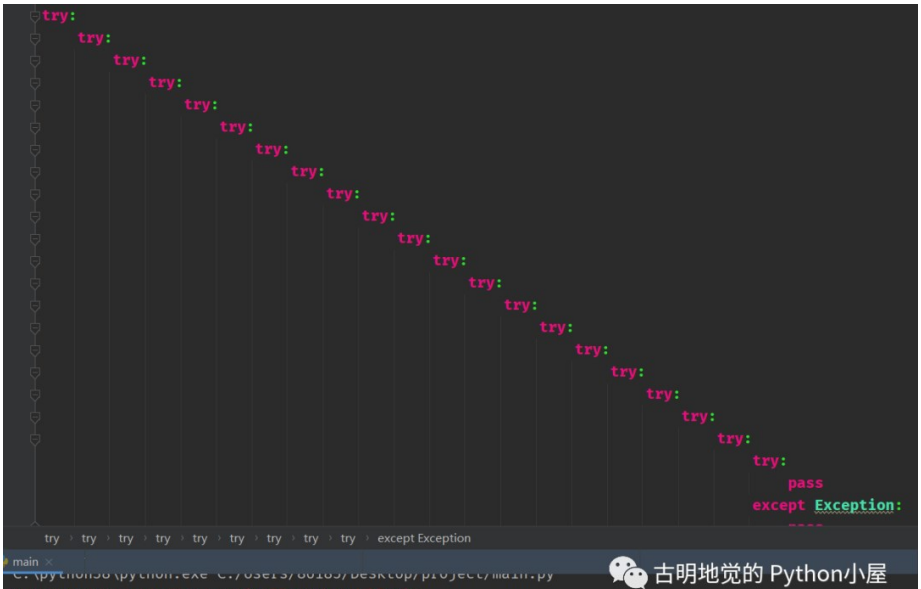
以上我们回顾了一下栈帧相关的知识，此时这个 `STACK_LEVEL()` 的含义应该很好理解了，它表示栈顶和栈底之间差了多少字节。再除以 8、也就是指针大小的话，那么不就是栈里面的元素数量吗？同理，如果 `STACK_LEVEL()` 等于0，这意味着什么呢？没错，表示此时的运行时栈是空的，而ceval.c里面也正好有这么一个宏。

```
#define EMPTY()  (STACK_LEVEL() == 0)
```

PyFrame_BlockSetup接收的参数我们知道了，再来看看函数的本体。

```
//Objects/frameobject.c
void
PyFrame_BlockSetup(PyFrameObject *f, int type, int handler, int level)
{
    //创建一个PyTryBlock *
    PyTryBlock *b;
    //这个f_iblock表示当前指令在f_blockstack上的索引
    //还记得这个f_blockstack吗？在栈帧中定义如下：
    //PyTryBlock f_blockstack[CO_MAXBLOCKS];
    //我们在介绍栈帧的时候说过，它可以用于try代码块
    //f_blockstack是一个数组，内部存储了多个PyTryBlock对象
    //CO_MAXBLOCKS是一个宏，为 20
    if (f->f_iblock >= CO_MAXBLOCKS)
        Py_FatalError("XXX block stack overflow");
    //这里得到的b显然就是PyTryBlock结构体实例的指针
    b = &f->f_blockstack[f->f_iblock++];
    //设置属性
    b->b_type = type;
    b->b_level = level;
    b->b_handler = handler;
}
```

try代码块在底层会对应一个`PyTryBlock`，它位于`f_blockstack`中。然后这个数组的长度是 20，这意味着在Python里面try代码块嵌套不能超过20层。



```
SyntaxError: too many statically nested blocks
```

这里我们嵌套了21层，所以报错了，这是一个在编译阶段就能检测出的错误。当然啦，如果不是恶意代码，我个人认为不会存在要嵌套超过20层的异常捕获。

然后`f_iblock`表示索引，用于获取某一个`PyTryBlock`。在拿到指针之后，就对里面的属性进行设置。那么来看一下这个`PyTryBlock`长什么样？

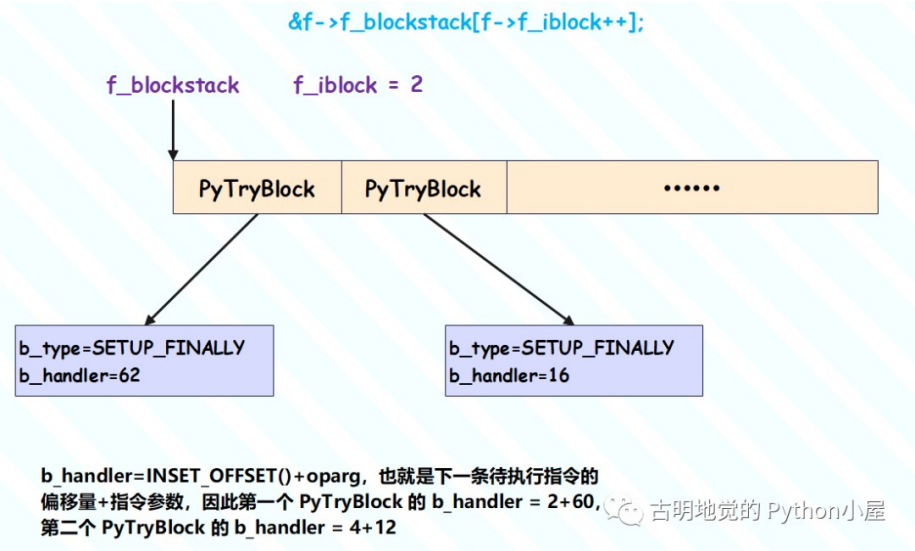
```
//frameobject.h
typedef struct {
    //block的种类，因为存在着多种用途的PyTryBlock对象
    //而在PyFrame_BlockSetup中我们看到它被设置成了参数type
    //参数type就是当前Python虚拟机正在执行的字节码指令
    //因此PyTryBlock的具体用途是以字节码指令作为区分的
    int b_type;
    //它被设置成了 handler
    //对于当前而言就是INSTR_OFFSET() + oparg
    //所以它表示要跳转的指令所在的偏移量
    int b_handler;
    //STACK_LEVEL()，运行时栈的深度
    int b_level;
} PyTryBlock;
```

栈帧对象的f_blockstack成员是一个由PyTryBlock对象组成的数组，而SETUP_FINALLY指令所做的事情就是从这个数组中获取一块PyTryBlock对象。

然后在这个对象中存放一些虚拟机当前的状态信息，比如当前执行的字节码指令，当前运行时栈的深度等等。那么这个对象在 try 控制结构中起着什么样的作用呢？我们后面就会知晓。

不过这里可能会有人好奇，为什么会出现两个SETUP_FINALLY，答案是：在3.8之前，第二条指令其实是SETUP_EXCEPT，第一条用于finally、第二条用于except。但从3.8开始，两条指令都换成了SETUP_FINALLY，但是指令参数（操作数）不同。

```
      # 跳转到 finally
      0 SETUP_FINALLY           60 (to 62)
      # 跳转到 except
      2 SETUP_FINALLY           12 (to 16)
```



取出两块PyTryBlock，在捕捉异常的时候用。至于怎么捕捉的一会儿再说，我们先回到抛异常的地方看看。抛异常是通过指令RAISE_VARARGS实现的，但是在这之前，已经先通过LOAD_NAME、LOAD_CONST 以及 CALL_FUNCTION构造了一个异常对象，并压入到了栈中。

> 尽管Exception是一个类，但调用的指令也同样是CALL_FUNCTION。至于这个指令的剖析和对象的创建后面会介绍，这里只需要知道一个异常对象已经被创建出来了。

而RAISE_VARARGS指令的工作就从把这个异常对象从运行时栈中取出开始。

```
case TARGET(RAISE_VARARGS): {
    PyObject *cause = NULL, *exc = NULL;
    switch (oparg) {
    case 2:
        cause = POP(); /* cause */
        /* fall through */
    case 1:
        exc = POP(); /* exc */
        /* fall through */
    case 0:
        if (do_raise(tstate, exc, cause)) {
            goto exception_unwind;
        }
        break;
    default:
        _PyErr_SetString(tstate, PyExc_SystemError,
                         "bad RAISE_VARARGS oparg");
        break;
    }
    goto error;
}
```

因为RAISE_VARARGS指令的参数是 1，所以 case 1 成立，于是将异常从运行时栈中弹出，并赋值给变量 exc，然后调用do_raise函数。

在do_raise中，最终会调用之前说过的PyErr_Restore函数，将异常对象存储到当前的线程状态对象中。在经过了一系列繁复的动作之后(比如创建并设置traceback)，虚拟机将携带着信息(f_iblock=2)抵达真正捕捉异常的代码，也就是跳转到标签为exception_unwind的地方进行异常捕获。

```
for (;;) {  // for 循环开始位置
    //......
exception_unwind:
    /* 如果异常发生了，栈帧展开 */
    //对于当前而言，f_iblock 等于 2
    while (f->f_iblock > 0) {
        //弹出 PyTryBlock，由于 try 代码块肯定是从内往外执行的
        //所以 f_blockstack 中的 block 也是从后往前获取的
        PyTryBlock *b = &f->f_blockstack[--f->f_iblock];

        //EXCEPT_HANDLER定义在opcode.h 中，值为 257
        //用于异常处理，但比较特殊的是，它不是一个指令（操作码）
        if (b->b_type == EXCEPT_HANDLER) {
          //这是一个宏，所做事情如下
          //从线程状态对象中拿到 exc_info，和 Python 里面 sys.exc_info 等价
          //然后从栈顶弹出 exc_type、exc_value、exc_traceback，并设置在 exc_info中
          //这里需要搭配下面的代码一起阅读，我们先往下看
            UNWIND_EXCEPT_HANDLER(b);
            continue;
        }
        //对于当前的代码而言，第一次弹出的PyTryBlock用于except
        //它的 b_type = SETUP_FINALLY、 b_handler=16
        if (b->b_type == SETUP_FINALLY) {
            //异常类型、异常值、回溯栈
            PyObject *exc, *val, *tb;
            //拿到 b_handler，也就是 except 对应的指令的偏移量
            int handler = b->b_handler;
            //从线程状态对象中拿到 exc_info
            _PyErr_StackItem *exc_info = tstate->exc_info;
            //将当前 PyTryBlock 的 b_type 设置为 EXCEPT_HANDLER
            PyFrame_BlockSetup(f, EXCEPT_HANDLER, -1, STACK_LEVEL());
            //将exc_traceback、exc_value、exc_type 压入运行时栈
            PUSH(exc_info->exc_traceback);
            PUSH(exc_info->exc_value);
            if (exc_info->exc_type != NULL) {
                PUSH(exc_info->exc_type);
```

```
            }
            else {
                Py_INCREF(Py_None);
                PUSH(Py_None);
            }
            //获取当前线程状态对象中存储的异常类型、异常值、回溯栈
            _PyErr_Fetch(tstate, &exc, &val, &tb);
            //对异常规范化处理，这里不需要关注
            _PyErr_NormalizeException(tstate, &exc, &val, &tb);
            //我们之前一直说 exc_type为异常类型、exc_value为异常值
            //但这个异常值其实指的就是异常对象本身，举个栗子：Exception("xxx")
            //那么exc_value指的就是Exception("xxx")本身
            //只是在打印的时候，会打印出 xxx。
            //至于 exc_type，显然就是 Exception 这个类型了
            //然后exc_value 内部有一个属性 __traceback__，拿到的就是 exc_traceback
            //所以这个PyException_SetTraceback就是用于检测__traceback__是否存在
            if (tb != NULL)
                PyException_SetTraceback(val, tb);
            else
                PyException_SetTraceback(val, Py_None);
            Py_INCREF(exc);
            exc_info->exc_type = exc;
            Py_INCREF(val);
            exc_info->exc_value = val;
            exc_info->exc_traceback = tb;
            if (tb == NULL)
                tb = Py_None;
            Py_INCREF(tb);
            PUSH(tb);
            PUSH(val);
            PUSH(exc);
            //绝对跳转，跳到偏移量为 handler 的指令
            JUMPTO(handler);
            /* Resume normal execution */
            goto main_loop;
        }
    }
    break;
} //for循环结束位置

assert(retval == NULL);
assert(_PyErr_Occurred(tstate));
```


我们看到虚拟机调用PUSH将tb、val、exc分别压入运行时栈中，并且知道此时开发者已经为异常处理做好了准备，所以接下来的异常处理工作，则需要交给开发者指定的代码来解决。

这个动作通过JUMP_FORWARD来完成，内部调用了JUMPTO(b->b_handler)。JUMPTO其实仅仅是进行了一下指令的跳跃，将虚拟机将要执行的下一条指令设置为异常处理代码编译后所得到的第一条字节码指令。

因为第一个弹出的是PyTryBlock中b_handler为16的SETUP_FINALLY，那么虚拟机将要执行的下一条指令就是偏移量为16的那条指令，而这条指令就是DUP_TOP，异常处理代码对应的第一条字节码指令。

```
case TARGET(DUP_TOP): {
    PyObject *top = TOP();
    Py_INCREF(top);
    PUSH(top);
    FAST_DISPATCH();
}
```

然后except Exception，毫无疑问要LOAD_NAME，把这个异常加载进来，然后调用指令COMPARE_OP，比较我们指定的异常和运行时栈中存在的异常是否匹配。

POP_JUMP_IF_FALSE如果为Py_True表示匹配，那么继续往下执行；如果为Py_False表示不匹配，然后直接跳转到了56 END_FINALLY。因为异常不匹配的话，那么异常的相关信息还是要重新放回线程状态对象当中，让虚拟机重新引发异常，而这个动作就

由END_FINALLY完成，通过PyErr_Restore函数将异常信息重新写回线程对象中。

```
case TARGET(END_FINALLY): {
    PREDICTED(END_FINALLY);
    PyObject *exc = POP();
    if (exc == NULL) {
        FAST_DISPATCH();
    }
    else if (PyLong_CheckExact(exc)) {
        int ret = _PyLong_AsInt(exc);
        Py_DECREF(exc);
        if (ret == -1 && _PyErr_Occurred(tstate)) {
            goto error;
        }
        JUMPTO(ret);
        FAST_DISPATCH();
    }
    else {
        assert(PyExceptionClass_Check(exc));
        PyObject *val = POP();
        PyObject *tb = POP();
        //将异常信息又写入到了线程状态对象当中
        _PyErr_Restore(tstate, exc, val, tb);
        goto exception_unwind;
    }
}
```

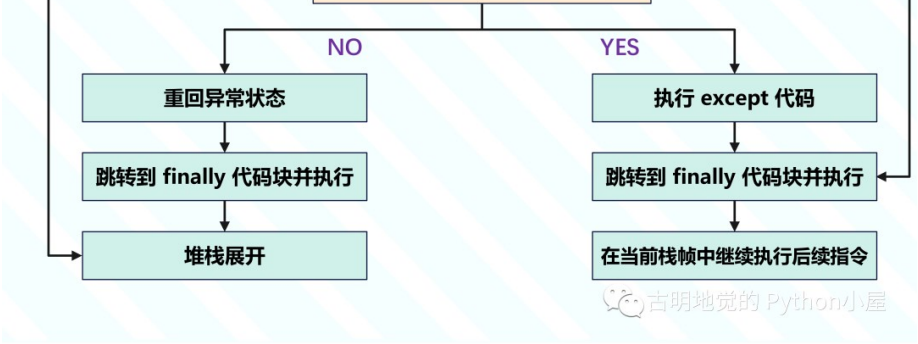
然而不管异常是否匹配，最终处理异常的两条岔路都会在58 POP_BLOCK处汇合。

```
case TARGET(POP_BLOCK): {
    // 将当前栈帧的f_blockstack中还剩下的那个、
    // 与SETUP_FINALLY对应的PyTryBlock对象弹出
    // 然后虚拟机的流程就进入了与finally表达式对应的字节码指令了
    PREDICTED(POP_BLOCK);
    PyFrame_BlockPop(f);
    DISPATCH();
}
```

因此在异常机制的实现中，最重要的就是虚拟机状态以及栈帧对象中f_blockstack里存放的PyTryBlock对象了。

首先根据虚拟机状态可以判断当前是否发生了异常，而PyTryBlock对象则告诉虚拟机，开发者是否为异常设置了except和finally，虚拟机异常处理的流程就是在虚拟机所处的状态和PyTryBlock的共同作用下完成的。

异常捕获涉及的内容比较多，源码就不一点一点的分析了，下面再用一张图来描述一下大致流程：



总之Python中一旦出现异常了，那么会将异常类型、异常值、异常回溯栈设置在线程状态对象中，然后栈帧一步一步地回退，寻找异常捕获代码(从内向外)。如果退到了模块级别还没有发现异常捕获，那么从外向内打印traceback中的信息，当走到最后一层的时候再将线程中设置的异常类型和异常值打印出来。

```
def h():
    1 / 0

def g():
    h()

def f():
    g()

f()

# traceback 回溯栈
Traceback (most recent call last):
  # 打印模块的 traceback
  File "D:/satori/main.py", line 13, in <module>
    f()

  # 打印 f 的traceback
  File "D:/satori/main.py", line 10, in f
    g()

  # 打印 g 的traceback
  File "D:/satori/main.py", line 6, in g
    h()

  # 打印 h 的traceback
  File "D:/satori/main.py", line 2, in h
    1 / 0

# h 的 traceback->tb_next 为 None，证明是在 h 中发生了错误
# 或者说 h 的 traceback 已经是最后一层了
# 所以再将之前设置在线程状态对象中异常类型和异常值打印出来即可
ZeroDivisionError: division by zero
```


模块中调用了 f、f 调用了 g、g 调用了 h，在 h 中执行出错了、但又没有异常捕获，那么会将执行权交给 g 对应的栈帧；但是 g 也没有异常捕获，那么再将执行权交给 f 对应的栈帧。所以调用的时候栈帧一层一层创建，执行完毕、或者出现异常时，栈帧一层一层回退。

因此 h 的 f_back指向g、g 的f_back指向 f、f 的 f_back指向模块、模块的 f_back为 None。但是对应的traceback则是模块的 tb_next指向 f、f 的 tb_next指向 g、g 的 tb_next指向 h、h 的 tb_next为 None。

因此栈帧的遍历顺序是**从 h 到模块**，traceback 的遍历顺序是**从模块到 h**。

思考题

最后再看一个思考题：

```
e = 2.718
try:
    raise Exception("我要引发异常了")
except Exception as e:
    print(e)  # 我要引发异常了

print(e)
# NameError: name 'e' is not defined
```

why？我们发现在外面打印 e 的时候，告诉我们 e 没有被定义。这是为什么呢？首先可以肯定的是，原因是由**except Exception as e**导致的，因为我们as 的也是 e，和外面的 e 重名了。那如果我们 as 的是 e1 呢？

```
e = 2.718
try:
    raise Exception("我要引发异常了")
except Exception as e1:
    print(e1)  # 我要引发异常了

print(e)  # 2.718
```

可以看到 as 的是 e1 就没有问题了，但是为什么呢？很容易推测出，因为外面的变量叫 e，而我们捕获异常 as 的也是 e，此时 e 的指向就变了。而当异常处理结束的时候，e 这个变量就被销毁了，所以外面就找不到了。然而事实上也确实如此。

我们可以看一下字节码，就能很清晰地看出端倪了。

```
             50 DELETE_NAME              0 (e)
             52 END_FINALLY
             54 POP_EXCEPT
             56 JUMP_FORWARD             2 (to 60)
        >>   58 END_FINALLY
        >>   60 LOAD_CONST               2 (None)
             62 RETURN_VALUE
```

字节码很长，但是我们只需要看偏移量为 50 的那个字节码即可。你看到了什么，DELETE_NAME 直接把 e 这个变量给删了，所以我们就找不到了，因此代码相当于下面这样：

```
e = 2.718
try:
    raise Exception("我要引发异常了")
except Exception as e:
    try:
        print(e)
    finally:
        del e
```

因此在异常处理的时候，如果把异常赋予了一个变量，那么这个变量在异常处理结束时会被删掉，因此只能在 except 里面使用，这就是原因。但是原因有了，可动机呢？Python这么做的动机是什么？根据官网文档解释：

当使用 as 将异常赋值给一个变量时，该变量将在 except 子句结束时被清除，这意味着异常必须赋值给一个不同的名称(不同于外部指定的变量)，才能在 except 子句之后引用它(外部指定的变

量)。

而变量被清除是因为在附加了回溯信息的情况下，它们会形成堆栈帧的循环引用，在下一次垃圾回收执行之前，会使所有局部变量都保持存活。

说人话就是，如果不将 as 后面的变量删除，那么原本一次 GC 就能搞定的事情会需要两次才能搞定。

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》56. 函数的底层结构

下一篇 ›
《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？