《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？

原创 古明地觉 古明地觉的编程教室 2022-03-23 08:30

收录于合集
#CPython
97个 >

楔子

程序在运行的过程中，总是会不可避免地产生异常，此时为了让程序不中断，必须要将异常捕获掉。如果能提前得知可能会发生哪些异常，则建议使用精确捕获；如果不知道会发生哪些异常，则使用 Exception。

另外异常也可以用来传递信息，比如生成器：

```
def gen():
    yield 1
    yield 2
    return "result"

g = gen()
next(g)
next(g)
try:
    next(g)
except StopIteration as e:
    print(f"返回值: {e.value}")   # result
```

如果想要拿到生成器的返回值，我们需要让它抛出 StopIteration，然后进行捕获，再调用 value 属性拿到返回值。所以，Python是将生成器的返回值封装到了异常里面。

之所以举这个例子，目的是想说明，异常并非是让人嗤之以鼻的东西，它也可以作为信息传递的载体。特别是在 Java 语言中，引入了 checked exception，方法的所有者还可以声明自己会抛出什么异常，然后调用者对异常进行处理。在 Java 程序启动时，抛出大量异常都是司空见惯的事情，并在相应的调用堆栈中将信息完整地记录下来。至此，Java 的异常不再是异常，而是一种很普遍的结构，从良性到灾难性都有所使用，异常的严重性由调用者来决定。

虽然在 Python 里面，异常还没有达到像 Java 异常那么高的地位，但使用频率也是很高的，下面我们就来剖析一下异常是怎么实现的？

Python的异常机制

如果想要产生异常，可以有两种方式：一种是虚拟机自身抛出异常，另一种是通过 raise 关键字。

如果是虚拟机自身抛异常的话，那么可以有很多种方式，比如索引越界、除以零、调用对象不存在的方法等等。下面我们就以除以零为例，看看异常是怎么产生的？整个流程是什么样子的？

```
s = "1 / 0"

```

```
if __name__ == '__main__':
    import dis
    dis.dis(compile(s, "<file>", "exec"))
"""
  1           0 LOAD_CONST               0 (1)
              2 LOAD_CONST               1 (0)
              4 BINARY_TRUE_DIVIDE
              6 POP_TOP
              8 LOAD_CONST               2 (None)
             10 RETURN_VALUE
"""
```

我们看第3条字节码指令，异常正是在执行这条指令的时候触发的。

```
case TARGET(BINARY_TRUE_DIVIDE): {
    //从栈顶弹出元素 0
    PyObject *divisor = POP();
    //获取新的栈顶元素 1
    PyObject *dividend = TOP();
    //调用 __truediv__
    PyObject *quotient = PyNumber_TrueDivide(dividend, divisor);
    //减少引用计数
    Py_DECREF(dividend);
    Py_DECREF(divisor);
    //将结果设置在栈顶
    SET_TOP(quotient);
    //但是结果 quotient 一定对吗？答案是不一定
    //如果除数是 0，那么说明出错了，此时会返回 NULL
    if (quotient == NULL)
        //因此会跳转到 error标签
        goto error;
    DISPATCH();
}
```

逻辑很简单，就是获取两个值，然后进行除法运算。正常情况下肯定会得到一个浮点数，而如果不能相除则返回 NULL。当接收的quotient是NULL，那么进入 error 标签。

下面看一下PyNumber_TrueDivide都干了些啥?

```
//longobject.c
//PyNumber_TrueDivide的核心在于 long_true_divide
static PyObject *
long_true_divide(PyObject *v, PyObject *w)
{
    //...
    a = (PyLongObject *)v;
    b = (PyLongObject *)w;
    //获取b_size，就是b对应的ob_size
    //如果 ob_size 为 0，说明对应的整数为 0
    //当除数为 0 时，抛出PyExc_ZeroDivisionError
    if (b_size == 0) {
        PyErr_SetString(PyExc_ZeroDivisionError,
                        "division by zero");
        goto error;
    }
    //...
}
```

当除数为 0 时，虚拟机就知道要抛异常了，所以会设置异常信息。

> Python提供了大量的异常，可以在pyerrors.h里面查看。另外我们说Python一切皆对象，因此异常也是一个对象。

而设置异常我们看到是通过`PyErr_SetString`实现的，该函数接收三个参数，分别是：线程状态对象、异常类型、异常值，然后会在线程状态对象中记录异常信息(线程的知识后续会说)。

当然啦，`PyErr_SetString`内部会调用`PyErr_SetObject`，在`PyErr_SetObject`内部又会调用`PyErr_Restore`，记录异常信息实际上是在`PyErr_Restore`里面实现的，我们来看一下这个函数。

```
// Python/errors.c
void
PyErr_Restore(PyObject *type, PyObject *value, PyObject *traceback)
{
    //获取线程对象
    PyThreadState *tstate = _PyThreadState_GET();
    _PyErr_Restore(tstate, type, value, traceback);
}

void
_PyErr_Restore(PyThreadState *tstate, PyObject *type, PyObject *value,
               PyObject *traceback)
{
    //异常类型、异常值、异常的回溯栈
    //对应Python中sys.exc_info()返回的元组里面的3个元素
    PyObject *oldtype, *oldvalue, *oldtraceback;

    //如果traceback不为空并且不是回溯栈
    //那么将其设置为NULL
    if (traceback != NULL && !PyTraceBack_Check(traceback)) {
        Py_DECREF(traceback);
        traceback = NULL;
    }

    //获取以前的异常信息
    oldtype = tstate->curexc_type;
    oldvalue = tstate->curexc_value;
    oldtraceback = tstate->curexc_traceback;

    //设置当前的异常信息
    tstate->curexc_type = type;
    tstate->curexc_value = value;
    tstate->curexc_traceback = traceback;

    //将之前的异常信息的引用计数分别减1
    Py_XDECREF(oldtype);
    Py_XDECREF(oldvalue);
    Py_XDECREF(oldtraceback);
}
```

最后线程状态对象tstate的curexc_type保存了PyExc_ZeroDivisionError，而cur_value中保存了异常值，curexc_traceback保存了回溯栈。

```
import sys

try:
    1 / 0
except ZeroDivisionError as e:
    exc_type, exc_value, exc_tb = sys.exc_info()
    # <class 'ZeroDivisionError'>
    print(exc_type)
    # division by zero
    print(exc_value)
    # <traceback object at 0x000001C43F29F4C0>
    print(exc_tb)
```

```
13
14      # exc_tb也可以通过e.__traceback__获取
15      print(e.__traceback__ is exc_tb)  # True
```

我们再来看看PyThreadState对象，它是与线程相关的，但它只是线程信息的一个抽象描述，而真实的线程及状态肯定是由操作系统来维护和管理的。

因为虚拟机在运行的时候总需要另外一些与线程相关的状态和信息，比如是否发生了异常等等，这些信息显然操作系统是没有办法提供的。而PyThreadState对象正是Python为线程准备的、在虚拟机层面保存线程状态信息的对象(后面简称线程状态对象、或者线程对象)。

在这里，当前活动线程(OS原生线程)对应的PyThreadState对象可以通过PyThreadState_GET获得，在得到了线程状态对象之后，就将异常信息存放在里面。

> 关于线程相关的内容，后续会详细说。

目前我们知道异常已经被记录在线程状态对象当中了，现在可以回头看看，在跳出了分派字节码指令的 switch 块所在的 for 循环之后，发生了什么动作。

> 我们知道在ceval.c里面有一个 _PyEval_EvalFrameDefault 函数，它是负责执行字节码指令的。里面有一个for循环，会依次遍历每一条字节码，而在这个for循环里面又有一个巨型switch，里面case了所有指令出现的情况。当全部的指令都执行完毕之后，这个for循环就结束了。

但这里还存在一个问题，就是导致跳出那个巨大switch块所在的for循环的原因可以有两种：

- 1. 执行完所有的字节码指令之后正常跳出；
- 2. 发生异常后跳出；

那么虚拟机是如何区分到底是哪一种呢？很简单，通过 error 标签实现。

```
1  PyObject* _Py_HOT_FUNCTION
2  _PyEval_EvalFrameDefault(PyFrameObject *f, int throwflag)
3  {
4      for (;;) {
5          switch (opcode) {
6              //一个超大的switch语句
7          }
8
9  //一旦出现异常，会使用goto语句跳转到error标签这里
10 //否则不会执行到这里
11 error:
12 #ifdef NDEBUG
13         if (!_PyErr_Occurred(tstate)) {
14             _PyErr_SetString(tstate, PyExc_SystemError,
15                              "error return without exception set");
16         }
17 #else
18         assert(_PyErr_Occurred(tstate));
19 #endif
20
21         //创建traceback对象
22         PyTraceBack_Here(f);
23         //c_tracefunc是用户自定义的追踪函数
24         //主要用于编写Python的debugger
25         //但通常情况下这个值都是NULL，所以不用考虑它
26         if (tstate->c_tracefunc != NULL)
27             call_exc_trace(tstate->c_tracefunc, tstate->c_traceobj,
```

```
                        tstate, f);

    }
}
```

如果在执行switch语句的时候出现了异常，那么会跳转到error这里，否则会跳转到其它地方。因此当跳转到error标签的时候就代表出现异常了，这里我们来看一下。

上面说了，当出现异常时，会在线程状态对象中将异常信息记录下来，包括异常类型、异常值、回溯栈（traceback）。那么问题来了，这个traceback是在什么地方创建的呢？显然是通过error标签中调用的`PyTraceBack_Here`创建的。

另外可能有人不清楚这个 traceback 是做什么的，我们举个Python的例子。

```
def h():
    1 / 0

def g():
    h()

def f():
    g()

f()

"""
Traceback (most recent call last):
  File "D:/satori/main.py", line 10, in <module>
    f()
  File "D:/satori/main.py", line 8, in f
    g()
  File "D:/satori/main.py", line 5, in g
    h()
  File "D:/satori/main.py", line 2, in h
    1 / 0
ZeroDivisionError: division by zero
"""
```

这是脚本运行时产生的错误输出，我们看到了函数调用的信息：比如在源代码的哪一行调用了哪一个函数，那么这些信息是从何而来的呢？

没错，显然是traceback对象。虚拟机在处理异常的时候，会创建traceback对象，在该对象中记录栈帧的信息。虚拟机利用该对象来将栈帧链表中每一个栈帧的状态进行可视化，可视化的结果就是上面输出的异常信息。

而且我们发现输出的信息也是一个链状的结构，因为每一个栈帧都会创建一个traceback对象，这些traceback对象之间也会组成一个链表。

所以当虚拟机开始处理异常的时候，它首先的动作就是创建一个traceback对象，用于记录异常发生时活动栈帧的状态。创建方式是通过`PyTraceBack_Here`函数，接收一个栈帧作为参数。

```
//Python/traceback.c
int
PyTraceBack_Here(PyFrameObject *frame)
{
    PyObject *exc, *val, *tb, *newtb;
    //获取保存线程状态的traceback对象
    PyErr_Fetch(&exc, &val, &tb);
    //_PyTraceBack_FromFrame创建新的traceback对象
    //此时新的traceback对象和老的traceback对象会组成链表
    newtb = _PyTraceBack_FromFrame(tb, frame);
    if (newtb == NULL) {
```

```
        _PyErr_ChainExceptions(exc, val, tb);
        return -1;
    }
    //将新的traceback对象交给线程状态对象
    PyErr_Restore(exc, val, newtb);
    Py_XDECREF(tb);
    return 0;
}
```

那么这个traceback对象究竟长什么样呢?

```
//Include/cpython/traceback.h
typedef struct _traceback {
    PyObject_HEAD
    // 指向下一个traceback
    struct _traceback *tb_next;
    // 指向对应的栈帧
    struct _frame *tb_frame;
    int tb_lasti;
    int tb_lineno;
} PyTracebackObject;
```

里面有一个tb_next，所以很容易想到这个traceback也是一个链表结构。其实traceback对象的链表结构跟栈帧对象的链表结构是同构的、或者说一一对应的，即一个栈帧对象对应一个traceback对象。

再来看一下这个链表是怎么产生的，在PyTraceBack_Here函数中我们看到它是通过_PyTraceBack_FromFrame创建的，那么秘密就隐藏在这个函数中：

```
//Python/traceback.h
PyObject*
_PyTraceBack_FromFrame(PyObject *tb_next, PyFrameObject *frame)
{
    assert(tb_next == NULL || PyTraceBack_Check(tb_next));
    assert(frame != NULL);

    //底层调用了tb_create_raw，参数如下：
    //下一个traceback、当前栈帧、当前f_lasti、以及源代码行号
    return tb_create_raw((PyTracebackObject *)tb_next, frame, frame->f_l
asti,
                         PyFrame_GetLineNumber(frame));
}

static PyObject *
tb_create_raw(PyTracebackObject *next, PyFrameObject *frame, int lasti,
              int lineno)
{
    PyTracebackObject *tb;
    if ((next != NULL && !PyTraceBack_Check(next)) ||
                    frame == NULL || !PyFrame_Check(frame)) {
        PyErr_BadInternalCall();
        return NULL;
    }
    //申请内存
    tb = PyObject_GC_New(PyTracebackObject, &PyTraceBack_Type);
    if (tb != NULL) {
        //建立链表
        Py_XINCREF(next);
        //让tb_next指向下一个traceback
        tb->tb_next = next;
        Py_XINCREF(frame);
        //设置栈帧
        //所以我们可以通过e.__traceback__.tb_frame获取栈帧
        tb->tb_frame = frame;
```

```
        //执行完毕时字节码的偏移量
        tb->tb_lasti = lasti;
        //源代码行号
        tb->tb_lineno = lineno;
        //加入GC追踪，参与垃圾回收
        PyObject_GC_Track(tb);
    }
    return (PyObject *)tb;
}
```

tb_next将两个traceback连接了起来，不过这个和栈帧里面f_back正好相反。f_back指向的是上一个栈帧，而tb_next指向的是下一个traceback。

另外在新创建的对象中，还使用tb_frame和对应的PyFrameObject对象建立了联系，当然还有最后执行完毕时的字节码偏移量、以及其在源代码中对应的行号。话说还记得PyCodeObject对象中的那个co_lnotab吗，这里的tb_lineno就是通过co_lnotab获取的。

目前信息量可能有点大，我们还以上面这段代码为例，来解释一下：

```
def h():
    1 / 0

def g():
    h()

def f():
    g()

f()
```

当执行到函数 h 的 1/0 这行代码时，底层会调用long_true_divide函数，由于除数为0，那么会通过PyErr_SetString设置一个异常进去，最终将异常类型、异常值、traceback 保存到线程状态对象中。但此时traceback实际上是为空的，因为目前还没有涉及到traceback的创建，那么它是什么时候创建的呢？继续往下看。

由于出现了异常，那么long_true_divide会返回NULL。

```
case TARGET(BINARY_TRUE_DIVIDE): {
    PyObject *divisor = POP();
    PyObject *dividend = TOP();
    PyObject *quotient = PyNumber_TrueDivide(dividend, divisor);
    Py_DECREF(dividend);
    Py_DECREF(divisor);
    SET_TOP(quotient);
    if (quotient == NULL)
        goto error;
    DISPATCH();
}
```

出现异常，返回 NULL

返回值为 NULL，跳转到 error 标签

古明地觉的 Python小屋

当返回值为 NULL 时，虚拟机就意识到发生异常了，这时候会跳转到 error 标签。在里面会先取出线程状态对象中已有的traceback对象（此时为空），然后以函数 h 的栈帧为参数，创建一个新的traceback对象，将两者通过 tb_next 关联起来。最后，再替换掉线程状态对象里面的traceback对象。

在虚拟机意识到到有异常抛出，并创建了traceback对象之后，它会在当前栈帧中寻找try except语句，来执行开发人员指定的捕捉异常的动作。如果没有找到，那么虚拟机将退出当前的活动栈帧，并沿着栈帧链回退到上一个栈帧（这里是函数 g 的栈帧），在上一个栈帧中寻找try except语句。

就像我们之前说的，出现函数调用会创建栈帧，当函数执行完毕或者出现异常的时候，会回退到上一级栈帧。一层一层创建、一层一层返回。至于回退的这个动作，则是在_PyEval_EvalFrameDefault的最后完成。

如果开发人员没有任何的捕获异常的动作，那么将通过标签exception_unwind里面的break跳出虚拟机执行字节码的那个for循环。最后，由于没有捕获到异常， 其返回值retval被设置为NULL，同时将当前线程状态对象中的活动栈帧，设置为上一级栈帧，从而完成栈帧回退的动作。

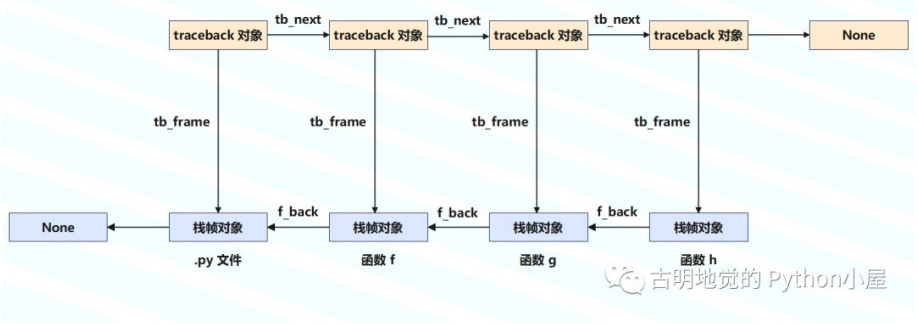
```
    for(;;){
        switch(opcode){
            //......
            case TARGET(BINARY_TRUE_DIVIDE): {
                //...
                PyObject *quotient = PyNumber_TrueDivide(dividend, divisor);
                //...
                if (quotient == NULL)
                    // 除数为 0，返回的 quotient 为 NULL，表示出现异常
                    // 并在线程状态对象中设置 异常类型、异常值、traceback
                    // 但是我们知道当前栈帧对应的 traceback 实际上并没有被创建
                    // 于是跳转到 error 标签
                    goto error;
                DISPATCH();
            }
            //......
        }

// 当执行某条指令出现异常时，会跳转到 error 标签这里
// 并从线程状态对象中拿到traceback对象
// 然后基于当前栈帧创建新的 traceback 对象，并将两者通过 tb_next 关联起来
error:
    //...
    PyTraceBack_Here(f);
    //...

// traceback 创建完毕之后，再去检测是否有异常捕获
// 所以我们看到不管有没有异常捕获，只要出现异常了，都会先将异常设置进去
// 然后如果发生的异常能够被捕获，那么再将异常给清空
// 所以 exception_unwind 就是检测是否有异常捕获逻辑
exception_unwind:
    //如果发生了异常，这里会将异常进行展开，然后试图捕获
    //至于异常捕获逻辑是怎么实现的，下一篇会单独分析，这里我们就让异常抛出去
        while (f->f_iblock > 0) {
            //...
        }
        break;
    }  // 注意：到这里，大大的for循环就结束了

    //retval表示函数的返回值
    //当发生异常时，retval 为 NULL，否则它一定是一个 PyObject *
    assert(retval == NULL);
    assert(_PyErr_Occurred(tstate));
    //......
exit_eval_frame:
    if (PyDTrace_FUNCTION_RETURN_ENABLED())
        dtrace_function_return(f);
    Py_LeaveRecursiveCall();
    f->f_executing = 0;
    //将线程状态对象中的活动栈帧设置为上一个栈帧，完成栈帧回退的动作
    tstate->frame = f->f_back;
    //在回退到上一个栈帧时，还要告诉它当前栈帧的返回值
    //因为要上一个要通过返回值是否为 NULL，来判断当前栈帧是否出异常了
    return _Py_CheckFunctionResult(NULL, retval, "PyEval_EvalFrameEx");
```


当栈帧回退时，会进入函数 g 的栈帧，由于 retval 为NULL，所以知道自己调用的函数 h 内部发生异常了（如果没有发生异常，返回值一定是一个PyObject *），那么继续寻找异常捕获语句。对于当前这个例子来说，显然是找不到的，于是会从线程状态对象中取出已有的traceback对象（此时是函数 h 的栈帧对应的traceback），然后以函数 g 的栈帧为参数，创建新的traceback对象，再将两者通过 tb_next 关联起来，并重新设置到线程状态对象中。

异常会沿着栈帧链进行反向传播，函数 h 出现的异常被传播到了函数 g 中，显然接下来函数 g 要将异常传播到函数 f 中。因为函数 g 在无法捕获异常时，也会将retval设置为NULL，而函数 f 看到返回值为NULL时，同样会去寻找异常捕获语句。但是找不到，于是会从线程状态对象中取出已有的traceback对象（此时是函数 g 的栈帧对应的traceback），然后以函数 f 的栈帧为参数，创建新的traceback对象，再将两者通过 tb_next 关联起来，并重新设置到线程状态对象中。

最后再传播到模块对应的栈帧中，如果还无法捕获发生的异常，那么虚拟机就要将异常抛出来了。

这个沿着栈帧链不断回退的过程我们称之为栈帧展开，在这个栈帧展开的过程中，虚拟机不断地创建与各个栈帧对应的traceback，并将其链接成链表。



由于没有异常捕获，那么接下来会调用PyErr_Print。然后在PyErr_Print中，虚拟机取出其维护的 traceback链表，并进行遍历，将里面的信息逐个输出到stderr当中，最终就是我们在Python中看到的异常信息。

并且打印顺序是：.py文件、函数f、函数g、函数h。因为每一个栈帧对应一个traceback，而栈帧又是往后退的，因此显然会从 .py文件对应的traceback开始打印，然后通过tb_next找到函数f 对应的traceback，依次下去……。当异常信息全部输出完毕之后，解释器就结束运行了。

> Python的异常也是一个对象，所谓的异常抛出，对于 C 而言，本质上就是将一段字符串输出到 stderr 中，然后中止程序运行。

因此从链路的开始位置到结束位置，将整个调用过程都输出出来，可以很方便地定位问题出现在哪里。

```
Traceback (most recent call last):
  File "D:/satori/main.py", line 10, in <module>
    f()
  File "D:/satori/main.py", line 8, in f
    g()
  File "D:/satori/main.py", line 5, in g
    h()
  File "D:/satori/main.py", line 2, in h
    1 / 0
ZeroDivisionError: division by zero
```

另外，虽然traceback一直在更新（因为要对整个调用链路进行追踪），但是异常类型和异常值是始终不变的，就是函数 h 中抛出的 `ZeroDivisionError: division by zero`。

以上就是虚拟机抛异常的过程，下一篇我们来分析异常捕获机制是如何实现的？

收录于合集 #CPython 97

‹ 上一篇
《源码探秘 CPython》55. 虚拟机是如何捕获异常的?

下一篇 ›
《源码探秘 CPython》53. 流程控制语句 for、while 是怎么实现的?