错误处理

1164次阅读

在程序运行的过程中，如果发生了错误，可以事先约定返回一个错误代码，这样，就可以知道是否有错，以及出错的原因。在操作系统提供的调用中，返回错误码非常常见。比如打开文件的函数open()，成功时返回文件描述符（就是一个整数），出错时返回-1。

用错误码来表示是否出错十分不便，因为函数本身应该返回的正常结果和错误码混在一起，造成调用者必须用大量的代码来判断是否出错：

```
def foo():
    r = some_function()
    if r==(-1):
        return (-1)
    # do something
    return r

def bar():
    r = foo()
    if r==(-1):
        print 'Error'
    else:
        pass
```

一旦出错，还要一级一级上报，直到某个函数可以处理该错误（比如，给用户输出一个错误信息）。

所以高级语言通常都内置了一套try...except...finally...的错误处理机制，Python也不例外。

try

让我们用一个例子来看看try的机制：

```
try:
    print 'try...'
    r = 10 / 0
    print 'result:', r
except ZeroDivisionError, e:
    print 'except:', e
finally:
    print 'finally...'
print 'END'
```

当我们认为某些代码可能会出错时，就可以用try来运行这段代码，如果执行出错，则后续代码不会继续执行，而是直接跳转至错误处理代码，即except语句块，执行完except后，如果有finally语句块，则执行finally语句块，至此，执行完毕。

上面的代码在计算10 / 0时会产生一个除法运算错误：

```
try...
except: integer division or modulo by zero
finally...
END
```

从输出可以看到，当错误发生时，后续语句print 'result:', r不会被执行，except由于捕获到ZeroDivisionError，因此被执行。最后，finally语句被执行。然后，程序继续按照流程往下走。

如果把除数0改成2，则执行结果如下：

```
try...
result: 5
finally...
END
```

由于没有错误发生，所以except语句块不会被执行，但是finally如果有，则一定会被执行（可以没有finally语句）。

你还可以猜测，错误应该有很多种类，如果发生了不同类型的错误，应该由不同的except语句块处理。没错，可以有多个except来捕获不同类型的错误：

```
try:
    print 'try...'
    r = 10 / int('a')
    print 'result:', r
except ValueError, e:
    print 'ValueError:', e
except ZeroDivisionError, e:
    print 'ZeroDivisionError:', e
finally:
    print 'finally...'
print 'END'
```

int()函数可能会抛出ValueError，所以我们用一个except捕获ValueError，用另一个except捕获ZeroDivisionError。

此外，如果没有错误发生，可以在except语句块后面加一个else，当没有错误发生时，会自动执行else语句：

```
try:
    print 'try...'
    r = 10 / int('a')
    print 'result:', r
except ValueError, e:
    print 'ValueError:', e
except ZeroDivisionError, e:
    print 'ZeroDivisionError:', e
else:
    print 'no error!'
finally:
    print 'finally...'
print 'END'
```

Python的错误其实也是class，所有的错误类型都继承自BaseException，所以在使用except时需要注意的是，它不但捕获该类型的错误，还把其子类也“一网打尽”。比如：

```
try:
    foo()
except StandardError, e:
    print 'StandardError'
except ValueError, e:
    print 'ValueError'
```

第二个except永远也捕获不到ValueError，因为ValueError是StandardError的子类，如果有，也被第一个except给捕获了。

Python所有的错误都是从BaseException类派生的，常见的错误类型和继承关系看这里：

https://docs.python.org/2/library/exceptions.html#exception-hierarchy

使用try...except捕获错误还有一个巨大的好处，就是可以跨越多层调用，比如函数main()调用foo()，foo()调用bar()，结果bar()出错了，这时，只要main()捕获到了，就可以处理：

```
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except StandardError, e:
        print 'Error!'
    finally:
        print 'finally...'
```

也就是说，不需要在每个可能出错的地方去捕获错误，只要在合适的层次去捕获错误就可以了。这样一来，就大大减少了写try...except...finally的麻烦。

调用堆栈

如果错误没有被捕获，它就会一直往上抛，最后被Python解释器捕获，打印一个错误信息，然后程序退出。来看看err.py：

```
# err.py:
def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    bar('0')

main()
```

执行，结果如下：

```
$ python err.py
Traceback (most recent call last):
  File "err.py", line 11, in <module>
    main()
  File "err.py", line 9, in main
    bar('0')
  File "err.py", line 6, in bar
    return foo(s) * 2
  File "err.py", line 3, in foo
    return 10 / int(s)
ZeroDivisionError: integer division or modulo by zero
```

出错并不可怕，可怕的是不知道哪里出错了。解读错误信息是定位错误的关键。我们从上往下可以看到整个错误的调用函数链：

错误信息第1行：

```
Traceback (most recent call last):
```

告诉我们这是错误的跟踪信息。

第2行：

```
  File "err.py", line 11, in <module>
    main()
```

调用main()出错了，在代码文件err.py的第11行代码，但原因是第9行：

```
  File "err.py", line 9, in main
    bar('0')
```

调用bar('0')出错了，在代码文件err.py的第9行代码，但原因是第6行：

```
  File "err.py", line 6, in bar
    return foo(s) * 2
```

原因是return foo(s) * 2这个语句出错了，但这还不是最终原因，继续往下看：

```
  File "err.py", line 3, in foo
    return 10 / int(s)
```

原因是return 10 / int(s)这个语句出错了，这是错误产生的源头，因为下面打印了：

```
ZeroDivisionError: integer division or modulo by zero
```

根据错误类型ZeroDivisionError，我们判断，int(s)本身并没有出错，但是int(s)返回0，在计算10 / 0时出错，至此，找到错误源头。

记录错误

如果不捕获错误，自然可以让Python解释器来打印出错误堆栈，但程序也被结束了。既然我们能捕获错误，就可以把错误堆栈打印出来，然后分析错误原因，同时，让程序继续执行下去。

Python内置的logging模块可以非常容易地记录错误信息：

```
# err.py
import logging

def foo(s):
    return 10 / int(s)

def bar(s):
    return foo(s) * 2

def main():
    try:
        bar('0')
    except StandardError, e:
        logging.exception(e)

main()
print 'END'
```

同样是出错，但程序打印完错误信息后会继续执行，并正常退出：

```
$ python err.py
ERROR:root:integer division or modulo by zero
Traceback (most recent call last):
  File "err.py", line 12, in main
    bar('0')
  File "err.py", line 8, in bar
    return foo(s) * 2
  File "err.py", line 5, in foo
    return 10 / int(s)
```

```
ZeroDivisionError: integer division or modulo by zero
END
```

通过配置，logging还可以把错误记录到日志文件里，方便事后排查。

抛出错误

因为错误是class，捕获一个错误就是捕获到该class的一个实例。因此，错误并不是凭空产生的，而是有意创建并抛出的。Python的内置函数会抛出很多类型的错误，我们自己编写的函数也可以抛出错误。

如果要抛出错误，首先根据需要，可以定义一个错误的class，选择好继承关系，然后，用raise语句抛出一个错误的实例：

```
# err.py
class FooError(StandardError):
    pass

def foo(s):
    n = int(s)
    if n==0:
        raise FooError('invalid value: %s' % s)
    return 10 / n
```

执行，可以最后跟踪到我们自己定义的错误：

```
$ python err.py
Traceback (most recent call last):
  ...
__main__.FooError: invalid value: 0
```

只有在必要的时候才定义我们自己的错误类型。如果可以选择Python已有的内置的错误类型（比如ValueError，TypeError），尽量使用Python内置的错误类型。

最后，我们来看另一种错误处理的方式：

```
# err.py
def foo(s):
    n = int(s)
    return 10 / n

def bar(s):
    try:
        return foo(s) * 2
    except StandardError, e:
        print 'Error!'
        raise

def main():
    bar('0')

main()
```

在bar()函数中，我们明明已经捕获了错误，但是，打印一个Error!后，又把错误通过raise语句抛出去了，这不有病么？

其实这种错误处理方式不但没病，而且相当常见。捕获错误目的只是记录一下，便于后续追踪。但是，由于当前函数不知道应该怎么处理该错误，所以，最恰当的方式是继续往上抛，让顶层调用者去处理。

raise语句如果不带参数，就会把当前错误原样抛出。此外，在except中raise一个Error，还可以把一种类型的错误转化成另一种类型：

```
try:
    10 / 0
except ZeroDivisionError:
    raise ValueError('input error!')
```

只要是合理的转换逻辑就可以，但是，决不应该把一个IOError转换成毫不相干的ValueError。

小结

Python内置的try...except...finally用来处理错误十分方便。出错时，会分析错误信息并定位错误发生的代码位置才是最关键的。

程序也可以主动抛出错误，让调用者来处理相应的错误。但是，应该在文档中写清楚可能会抛出哪些错误，以及错误产生的原因。
