

## 08 | 异常处理：如何提高程序的稳定性？

2019-05-27 景霄

Python核心技术与实战

[进入课程 >](#)



讲述：冯永吉

时长 08:47 大小 8.06M



你好，我是景霄。

今天这节课，我想和你聊聊 Python 的异常处理。和其他语言一样，异常处理是 Python 中一种很常见，并且很重要的机制与代码规范。

我在实际工作中，见过很多次这样的情况：一位工程师提交了代码，不过代码某处忘记了异常处理。碰巧这种异常发生的频率不低，所以在代码 push 到线上后没多久，就会收到紧急通知——服务器崩溃了。


如果事情严重，对用户的影响也很大，这位工程师还得去专门的会议上做自我检讨，可以说是很惨了。这类事件层出不穷，也告诉我们，正确理解和处理程序中的异常尤为关键。

## 错误与异常

首先要了解，Python 中的错误和异常是什么？两者之间又有什么联系和区别呢？

通常来说，程序中的错误至少包括两种，一种是语法错误，另一种则是异常。


所谓语法错误，你应该很清楚，也就是你写的代码不符合编程规范，无法被识别与执行，比如下面这个例子：

 复制代码

```
1 if name is not None
2     print(name)
```

If 语句漏掉了冒号，不符合 Python 的语法规则，所以程序就会报错 `invalid syntax`。

而异常则是指程序的语法正确，也可以被执行，但在执行过程中遇到了错误，抛出了异常，比如下面的 3 个例子：

 复制代码

```
1 10 / 0
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 ZeroDivisionError: integer division or modulo by zero
5
6 order * 2
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9 NameError: name 'order' is not defined
10
11 1 + [1, 2]
12 Traceback (most recent call last):
13   File "<stdin>", line 1, in <module>
14 TypeError: unsupported operand type(s) for +: 'int' and 'list'
```


它们语法完全正确，但显然，我们不能做除法时让分母为 0；也不能使用未定义的变量做运算；而让一个整型和一个列表相加也是不可取的。

于是，当程序运行到这些地方时，就抛出了异常，并且终止运行。例子中的 `ZeroDivisionError` `NameError` 和 `TypeError`，就是三种常见的异常类型。

当然，Python 中还有很多其他异常类型，比如 `KeyError` 是指字典中的键找不到；`FileNotFoundError` 是指发送了读取文件的请求，但相应的文件不存在等等，我在此不一一赘述，你可以自行参考[相应文档](#)。

## 如何处理异常


刚刚讲到，如果执行到程序中某处抛出了异常，程序就会被终止并退出。你可能会问，那有没有什么办法可以不终止程序，让其照样运行下去呢？答案当然是肯定的，这也就是我们所说的异常处理，通常使用 `try` 和 `except` 来解决，比如：

 复制代码

```
1 try:
2     s = input('please enter two numbers separated by comma: ')
3     num1 = int(s.split(',')[0].strip())
4     num2 = int(s.split(',')[1].strip())
5     ...
6 except ValueError as err:
7     print('Value Error: {}'.format(err))
8
9 print('continue')
10 ...
```

这里默认用户输入以逗号相隔的两个整形数字，将其提取后，做后续的操作（注意 `input` 函数会将输入转换为字符串类型）。如果我们输入 `a,b`，程序便会抛出异常 `invalid literal for int() with base 10: 'a'`，然后跳出 `try` 这个 `block`。


由于程序抛出的异常类型是 `ValueError`，和 `except block` 所 `catch` 的异常类型相匹配，所以 `except block` 便会被执行，最终输出 `Value Error: invalid literal for int() with base 10: 'a'`，并打印出 `continue`。

 复制代码

```
1 please enter two numbers separated by comma: a,b
2 Value Error: invalid literal for int() with base 10: 'a'
3 continue
```

我们知道，except block 只接受与它相匹配的异常类型并执行，如果程序抛出的异常并不匹配，那么程序照样会终止并退出。


所以，还是刚刚这个例子，如果我们只输入1，程序抛出的异常就是IndexError: list index out of range，与 ValueError 不匹配，那么 except block 就不会被执行，程序便会终止并退出（continue 不会被打印）。

 复制代码

```
1 please enter two numbers separated by comma: 1
2 IndexError Traceback (most recent call last)
3 IndexError: list index out of range
```


不过，很显然，这样强调一种类型的写法有很大的局限性。那么，该怎么解决这个问题呢？

其中一种解决方案，是在 except block 中加入多种异常的类型，比如下面这样的写法：

 复制代码

```
1 try:
2     s = input('please enter two numbers separated by comma: ')
3     num1 = int(s.split(',')[0].strip())
4     num2 = int(s.split(',')[1].strip())
5     ...
6 except (ValueError, IndexError) as err:
7     print('Error: {}'.format(err))
8
9 print('continue')
10 ...
```

或者第二种写法：


 复制代码

```
1 try:
2     s = input('please enter two numbers separated by comma: ')
3     num1 = int(s.split(',')[0].strip())
4     num2 = int(s.split(',')[1].strip())
```

```
5     ...
6 except ValueError as err:
7     print('Value Error: {}'.format(err))
8 except IndexError as err:
9     print('Index Error: {}'.format(err))
10
11 print('continue')
12 ...
```


这样，每次程序执行时，except block 中只要有一个 exception 类型与实际匹配即可。

不过，很多时候，我们很难保证程序覆盖所有的异常类型，所以，更通常的做法，是在最后一个 except block，声明其处理的异常类型是 Exception。Exception 是其他所有非系统异常的基类，能够匹配任意非系统异常。那么这段代码就可以写成下面这样：

 复制代码

```
1 try:
2     s = input('please enter two numbers separated by comma: ')
3     num1 = int(s.split(',')[0].strip())
4     num2 = int(s.split(',')[1].strip())
5     ...
6 except ValueError as err:
7     print('Value Error: {}'.format(err))
8 except IndexError as err:
9     print('Index Error: {}'.format(err))
10 except Exception as err:
11     print('Other error: {}'.format(err))
12
13 print('continue')
14 ...
```

或者，你也可以在 except 后面省略异常类型，这表示与任意异常相匹配（包括系统异常等）：

 复制代码


```
1 try:
2     s = input('please enter two numbers separated by comma: ')
3     num1 = int(s.split(',')[0].strip())
4     num2 = int(s.split(',')[1].strip())
5     ...
6 except ValueError as err:
```

```
7     print('Value Error: {}'.format(err))
8 except IndexError as err:
9     print('Index Error: {}'.format(err))
10 except:
11     print('Other error')
12
13 print('continue')
14 ...
```

需要注意，当程序中存在多个 `except` block 时，最多只有一个 `except` block 会被执行。换句话说，如果多个 `except` 声明的异常类型都与实际相匹配，那么只有最前面的 `except` block 会被执行，其他则被忽略。

异常处理中，还有一个很常见的用法是 `finally`，经常和 `try`、`except` 放在一起用。无论发生什么情况，`finally` block 中的语句都会被执行，哪怕前面的 `try` 和 `except` block 中使用了 `return` 语句。

一个常见的应用场景，便是文件的读取：

 复制代码

```
1 import sys
2 try:
3     f = open('file.txt', 'r')
4     .... # some data processing
5 except OSError as err:
6     print('OS error: {}'.format(err))
7 except:
8     print('Unexpected error:', sys.exc_info()[0])
9 finally:
10     f.close()
```


这段代码中，`try` block 尝试读取 `file.txt` 这个文件，并对其中的数据进行一系列的处理，到最后，无论是读取成功还是读取失败，程序都会执行 `finally` 中的语句——关闭这个文件流，确保文件的完整性。因此，在 `finally` 中，我们通常会放一些**无论如何都要执行**的语句。

值得一提的是，对于文件的读取，我们也常常使用 `with open`，你也许在前面的例子中已经看到过，`with open` 会在最后自动关闭文件，让语句更加简洁。

## 用户自定义异常


前面的例子里充斥了很多 Python 内置的异常类型，你可能会问，我可以创建自己的异常类型吗？

答案是肯定是，Python 当然允许我们这么做。下面这个例子，我们创建了自定义的异常类型 `MyInputError`，定义并实现了初始化函数和 `str` 函数（直接 `print` 时调用）：

 复制代码

```
1 class MyInputError(Exception):
2     """Exception raised when there're errors in input"""
3     def __init__(self, value): # 自定义异常类型的初始化
4         self.value = value
5     def __str__(self): # 自定义异常类型的 string 表达形式
6         return ("{} is invalid input".format(repr(self.value)))
7
8 try:
9     raise MyInputError(1) # 抛出 MyInputError 这个异常
10 except MyInputError as err:
11     print('error: {}'.format(err))
```

如果你执行上述代码块并输出，便会得到下面的结果：

 复制代码

```
1 error: 1 is invalid input
```

实际工作中，如果内置的异常类型无法满足我们的需求，或者为了让异常更加详细、可读，想增加一些异常类型的其他功能，我们可以自定义所需异常类型。不过，大多数情况下，Python 内置的异常类型就足够好了。


## 异常的使用场景与注意点

学完了前面的基础知识，接下来我们着重谈一下，异常的使用场景与注意点。

通常来说，在程序中，如果我们不确定某段代码能否成功执行，往往这个地方就需要使用异常处理。除了上述文件读取的例子，我可以再举一个例子来说明。

大型社交网站的后台，需要针对用户发送的请求返回相应记录。用户记录往往储存在 key-value 结构的数据库中，每次有请求过来后，我们拿到用户的 ID，并用 ID 查询数据库中此人的记录，就能返回相应的结果。

而数据库返回的原始数据，往往是 json string 的形式，这就需要我们首先对 json string 进行 decode（解码），你可能很容易想到下面的方法：

 复制代码

```
1 import json
2 raw_data = queryDB(uid) # 根据用户的 id，返回相应的信息
3 data = json.loads(raw_data)
```

这样的代码是不是就足够了呢？

要知道，在 json.loads() 函数中，输入的字符串如果不符合其规范，那么便无法解码，就会抛出异常，因此加上异常处理十分必要。

 复制代码

```
1 try:
2     data = json.loads(raw_data)
3     ....
4 except JSONDecodeError as err:
5     print('JSONDecodeError: {}'.format(err))
```

不过，有一点切记，我们不能走向另一个极端——滥用异常处理。

比如，当你想要查找字典中某个键对应的值时，绝不能写成下面这种形式：


 复制代码

```
1 d = {'name': 'jason', 'age': 20}
2 try:
3     value = d['dob']
4     ...
5 except KeyError as err:
6     print('KeyError: {}'.format(err))
```



诚然，这样的代码并没有 bug，但是让人看了摸不着头脑，也显得很冗余。如果你的代码中充斥着这种写法，无疑对阅读、协作来说都是障碍。因此，对于 flow-control（流程控制）的代码逻辑，我们一般不用异常处理。

字典这个例子，写成下面这样就很好。

 复制代码

```
1 if 'dob' in d:
2     value = d['dob']
3     ...
```

## 总结

这节课，我们一起学习了 Python 的异常处理及其使用场景，你需要重点掌握下面几点。

异常，通常是指程序运行的过程中遇到了错误，终止并退出。我们通常使用 try except 语句去处理异常，这样程序就不会被终止，仍能继续执行。


处理异常时，如果有必须执行的语句，比如文件打开后必须关闭等等，则可以放在 finally block 中。

异常处理，通常用在你不确定某段代码能否成功执行，也无法轻易判断的情况下，比如数据库的连接、读取等等。正常的 flow-control 逻辑，不要使用异常处理，直接用条件语句解决就可以了。

## 思考题

最后，给你留一个思考题。在异常处理时，如果 try block 中有多处抛出异常，需要我们使用多个 try except block 吗？以数据库的连接、读取为例，下面两种写法，你觉得哪种更好呢？


第一种：

 复制代码

```
1 try:
2     db = DB.connect('<db path>') # 可能会抛出异常
```

```
3     raw_data = DB.queryData('<viewer_id>') # 可能会抛出异常
4 except (DBConnectionError, DBQueryDataError) err:
5     print('Error: {}'.format(err))
```

第二种：

 复制代码

```
1 try:
2     db = DB.connect('<db path>') # 可能会抛出异常
3     try:
4         raw_data = DB.queryData('<viewer_id>')
5     except DBQueryDataError as err:
6         print('DB query data error: {}'.format(err))
7 except DBConnectionError as err:
8     print('DB connection error: {}'.format(err))
```

欢迎在留言区写下你的答案，还有你今天学习的心得和疑惑，也欢迎你把这篇文章分享给你的同事、朋友。




# Python 核心技术与实战

系统提升你的 Python 能力

景霄

Facebook 资深工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

上一篇 07 | 修炼基本功：条件与循环

下一篇 09 | 不可或缺的自定义函数

## 精选留言 (44)

写留言



Hoo-Ah

2019-05-27

16

第一种写法更加简洁，易于阅读。而且except后面的错误类型先抛出数据库连接错误，之后才抛出查询错误，实现的异常处理和第二种一样。

作者回复: 正解



不瘦到140...

2019-05-27

14

老师，看到异常这一讲，忽然想起了一个问题，一直困扰着我

```
e = 1
```

```
try:
```

```
    1 / 0
```

```
except ZeroDivisionError as e:...
```

展开

作者回复: 你可以阅读官方文档:

[https://docs.python.org/3/reference/compound\\_stmts.html#the-try-statement](https://docs.python.org/3/reference/compound_stmts.html#the-try-statement)

"When an exception has been assigned using as target, it is cleared at the end of the except clause."

比如下面这个code block:

```
except E as N:
```

```
    foo
```

就等于

```
except E as N:
```

```
    try:
```

```
        foo
```

finally:  
del N

因此你例子中的e最后被delete了，所以会抛出NameError

 **Geek\_b6f31...**

2019-05-27

 10

第一种方法简单明了，是不是少了一个as

展开 ▾



**liput**

2019-05-27

 3

想请问老师，在facebook里面开发，对于异常处理有什么规范需要遵循吗？自定义异常、抛异常、捕获异常，粒度一般怎么把控呢？

与此相应的，我对日志输出也有同样的疑问，希望老师能结合您在大公司里的实战经验多讲讲。

作者回复: 我会在最后一章里对大公司开发的规范，流程做一个详细的介绍。通常来说，异常能用内置的exception就用，如果需要自定义就自定义，看实际的需求。一般来说异常抛出，我们都会对其进行Log（一般每1000次log一次），输出到real time的table和dashboard里，这样有利于之后的分析和改进。



**Fergus**

2019-05-28

 2

选择1，原因有2：

1. 从开始学就如①这么写的；
2. ②读起来太难受，太不pythonic；

读了留言后意识到： ...

展开 ▾



**third**

2019-05-27

 2

小白提问

针对06思考的第一问，分次读取文件

读取完成一个很大的文件之后，这些文件存放在哪里？

如果这个文件大到了无法一次读取内存，那么他是否应该存储在磁盘中？在需要用的时候，又从磁盘中调取呢？ ...

展开 ∨



Python高...

2019-05-27

👍 2

第一种方式清晰一点

展开 ∨



John Si

2019-05-27

👍 2

1. 第一種寫法比第二種寫法簡潔

2. 因我對try語法執行流程不太清楚，還是老師跟熟悉該同學多講解一下。但我自己想法是第二種寫法跟巢狀迴圈寫法很像，假設是第二句語法發生錯誤，第二種寫法會多執行一次try 語句，從而增加了程序運行時間。

...

展开 ∨

作者回复: 正解



太湖湖-燃...

2019-05-31

👍 1

什么是flow-control。。

展开 ∨



HelloWorld

2019-05-29

👍 1

老师，对文中的自定义异常不太理解，为什么要主动去raise自定义异常，有啥意义吗



无才不肖生

👍 1



2019-05-28

老师，想问个题外题

对象的属性是个字典，django中要根据属性里字典的某个键值查询该怎么做呢，如下面的，我想直接根据companyType来查询过滤记录

person

{...

展开 ▾



SCAR

2019-05-27

👍 1

应该是第一种情况更优：

1. 从实现的功能来看第一种和第二种都能达到，不过程序里两种异常是线性依次，所以完全可以并行写在一个except里，再者python里的标准异常都\_\_str\_\_处理过，是完全可以区分哪个异常发生的。
2. 但就像the zen of python里强调的那样，简单和清晰是 python代码应该遵从的，显...

展开 ▾



alan

2019-05-27

👍 1

老师好，相比try-catch的错误处理模式，您觉得Golang那样的返回error的错误处理方式怎么样？您更喜欢哪种错误处理方式？



小豹子

2019-05-27

👍 1

老师，系统异常，非系统异常能举个例子说明下吗？

展开 ▾

作者回复: 系统异常比如说keyboardInterrupt



leixin

2019-05-27

👍 1

期待更进阶的。

展开 ▾



王校

2019-05-27

👍 1

第一种对阅读程序的人更友好。

展开 ▾

---



小侠龙旋风

2019-06-03

👍

用with open打开文件就不需要手动close了。

第一种异常处理较好。

调用json.loads时要考虑异常处理，新技能get。

展开 ▾

---



夹心面包

2019-06-03

👍

第一种吧,数据库如果无法建立链接只需要第一个错误即可

展开 ▾

---



yllopy

2019-06-03

👍

在用finally中用f.close()前建议加上一条判断: if 'f' in locals() 用以避免在f被赋值前发生异常而没有被定义导致的二次异常。

---



hlz-123

2019-06-02

👍

第一种写法简洁易懂，但实际编程中，数据库经常只连接一次，读数据往往重复多次，可能分开来写更好一些。