异常处理

异常 Exception

错误 Error
逻辑错误：算法写错了，加法写成了减法
笔误：变量名写错了，语法错误
函数或类使用错误，其实这也属于逻辑错误
总之，错误是可以避免的

异常 Exception
本意就是意外情况
这有个前提，没有出现上面说的错误，也就是说程序写的没有问题，但是在某些情况下，会出现一些意外，导致程序无法正常的执行下去。
例如open函数操作一个文件，文件不存在，或者创建一个文件时已经存在了，或者访问一个网络文件，突然断网了，这就是异常，是个意外的情况。
异常不可能避免

错误和异常
在高级编程语言中，一般都有错误和异常的概念，异常是可以捕获，并被处理的，但是错误是不能被捕获的。

举例
对比异常和错误

```
with open('testabc') as f:
    pass

# 异常
Traceback (most recent call last):
  File "test.py", line 4, in <module>
    with open('testabc') as f:
FileNotFoundError: [Errno 2] No such file or directory: 'testabc'

def 0A():
    pass

# 错误
  File "test.py", line 3
```

```
    def 0A():
        ^
SyntaxError: invalid syntax
```

一个健壮的程序
尽可能的避免错误
尽可能的捕获、处理各种异常

产生异常

产生：

- raise 语句显式的抛出异常
- Python解释器自己检测到异常并引发它

```
def foo():
    print('before')
    def bar():
        print(1/0) # 除零异常

    bar()
    print('after')
foo()

def bar():
    print('before')
    raise Exception('my exception')
    print('after')
bar()
```


程序会在异常抛出的地方中断执行，如果不捕获，就会提前结束程序

raise语句
raise后什么都没有，表示抛出最近一个被激活的异常，如果没有被激活的异常，则抛类型异常。
这种方式很少用
raise后要求应该是BaseException类的子类或实例，如果是类，将被无参实例化。

异常的捕获

```
try:
    待捕获异常的代码块
except [异常类型]:
    异常的处理代码块
```

```
try:
    print('begin')
    c = 1/0
    print('end')
except:
    print('catch the exception')
print('outer')
```

上例执行到 `c = 1/0` 时产生异常并抛出，由于使用了try...except语句块则捕捉到了这个异常，异常生成位置之后语句将不再执行，转而执行对应的except部分的语句，最后执行try...except语句块之外的语句。

捕获指定类型的异常

```
try:
    print('begin')
    c = 1/0
    print('end')
except ArithmeticError:
    print('catch the ArithmeticError')
print('outer')
```


异常类及继承层次

```
# Python异常的继承

BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- RuntimeError
      |    +-- RecursionError
```

```
      +-- MemoryError
      +-- NameError
      +-- StopIteration
      +-- StopAsyncIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- SyntaxError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
```


BaseException及子类

BaseException
所有内建异常类的基类是BaseException

SystemExit
sys.exit()函数引发的异常，异常不捕获处理，就直接交给Python解释器，解释器退出。

```
import sys
print('before')
sys.exit(1)
```

```
print('SysExit')
print('outer') # 是否执行?

# 捕获这个异常
import sys
try:
    sys.exit(1)
except SystemExit:
    print('SysExit')
print('outer') # 是否执行?
```

KeyboardInterrupt
对应的捕获用户中断行为Ctrl + C

```
try:
    import time
    while True:
        time.sleep(0.5)
        pass
except KeyboardInterrupt:
    print('ctl + c')
print('outer')
```


Exception及子类

Exception是所有内建的、非系统退出的异常的基类，自定义异常应该继承自它

SyntaxError 语法错误
Python将这种错误也归到异常类下面的Exception下的子类，但是这种错误是不可捕获的

```
def a():
    try:
        0a = 5
    except:
        pass

# 错误
  File "test2.py", line 3
    0a = 5
     ^
```

```
SyntaxError: invalid syntax
```

ArithmeticError 所有算术计算引发的异常，其子类有除零异常等

LookupError
使用映射的键或序列的索引无效时引发的异常的基类：IndexError, KeyError

自定义异常
从Exception继承的类

```
class MyException(Exception):
    pass

try:
    raise MyException()
except MyException: # 捕获自定义异常
    print('catch the exception')
```

异常的捕获

except可以捕获多个异常

```
class MyException(Exception):
    pass

try:
    a = 1/0 #
    raise MyException() # 自定义
    open('a1.txt') #
except MyException:
    print('catch the MyException')
except ZeroDivisionError:
    print('1/0')
except Exception: # 调整except的顺序
    print('Exception')
```

捕获规则
捕获是从上到下依次比较，如果匹配，则执行匹配的except语句块
如果被一个except语句捕获，其他except语句就不会再次捕获了
如果没有任何一个except语句捕获到这个异常，则该异常向外抛出

捕获的原则
从小到大，从具体到宽泛

as子句

被抛出的异常，应该是异常的实例，如何获得这个对象呢？使用as子句

```
class MyException(Exception):
    def __init__(self, code, message):
        self.code = code
        self.message = message

try:
    raise MyException
except MyException as e:
    print('MyException = {} {}'.format(e.code, e.message))
except Exception as e:
    print('Exception = {}'.format(e))

#运行结果如下：
Exception = __init__() missing 2 required positional arguments: 'code' and 'message
'
```

raise后跟类名是无参构造实例，因此需要2个参数

```
class MyException(Exception):
    def __init__(self, code, message):
        self.code = code
        self.message = message

try:
    raise MyException(200, 'ok')
except MyException as e:
    print('MyException = {} {}'.format(e.code, e.message))
except Exception as e:
    print('Exception = {}'.format(e))
```

finally子句

finally
最终，即最后一定要执行的，try...finally语句块中，不管是否发生了异常，都要执行finally的部分

```
try:
    f = open('test.txt')
except FileNotFoundError as e:
    print('{} {} {}'.format(e.__class__, e.errno, e.strerror))
finally:
    print('清理工作')
    f.close() #
```

注意上例中的f的作用域，解决的办法是在外部定义f

finally中一般放置资源的清理、释放工作的语句

```
f = None
try:
    f = open('test.txt')
except Exception as e:
    print('{}'.format(e))
finally:
    print('清理工作')
    if f:
        f.close()
```


也可以在finally中再次捕捉异常

```
try:
    f = open('test.txt')
except Exception as e:
    print('{}'.format(e))
finally:
    print('清理工作')
    try:
        f.close()
    except NameError as e:
        print(e)
```

finally 执行时机

```
# 测试1
def foo():
    try:
        return 3
    finally:
        print('finally')
    print('==')
print(foo())
```

```
# 测试2
def foo():
    #return 1
    try:
        return 3
    finally:
        return 5
    print('==')
print(foo())
```

测试1
进入try，执行return 3，虽然函数要返回，但是finally一定还要执行，所以打印了finally后，函数返回。

测试2
进入try，执行return 3，虽然函数要返回，但是finally一定还要执行，所以执行return 5，函数返回。5被压在栈顶，所以返回5。简单说，函数的返回值取决于最后一个执行的return语句，而finally则是try...finally中最后执行的语句块。

异常的传递

```
def foo1():
    return 1/0

def foo2():
    print('foo2 start')
    foo1()
    print('foo2 stop')
```

```
foo2()
```

foo2调用了foo1，foo1产生的异常，传递到了foo2中。
异常总是向外层抛出，如果外层没有处理这个异常，就会继续向外抛出
如果内层捕获并处理了异常，外部就不能捕获到了
如果到了最外层还是没有被处理，就会中断异常所在的线程的执行

```
# 线程中测试异常
import threading
import time


def foo1():
    return 1/0

def foo2():
    time.sleep(3) # 3秒后抛出异常
    print('foo2 start')
    foo1()
    print('foo2 stop')

t = threading.Thread(target=foo2)
t.start()

while True:
    time.sleep(1)
    print('Everything OK')
    if t.is_alive():
        print('alive')
    else:
        print('dead')
```

try嵌套

```
try:
    try:
        ret = 1 / 0
    except KeyError as e:
        print(e)
```

```
    else:
        print('inner OK')
    finally:
        print('inner fin')
except:
    print('outer catch')
finally:
    print('outer fin')
```

内部捕获不到异常，会向外层传递异常
但是如果内层有finally且其中有return、break语句，则异常就不会继续向外抛出

```
def foo():
    try:
        ret = 1 / 0
    except KeyError as e:
        print(e)
    finally:
        print('inner fin')
        return # 异常被丢弃

try:
    foo()
except:
    print('outer catch')
finally:
    print('outer fin')
```


异常的捕获时机

1、立即捕获

需要立即返回一个明确的结果

```
def parse_int(s):
    try:
        return int(s)
    except:
        return 0
```

```
print(parse_int('s'))
```

2、边界捕获

封装产生了边界。

例如，写了一个模块，用户调用这个模块的时候捕获异常，模块内部不需要捕获、处理异常，一旦内部处理了，外部调用者就无法感知了。
例如，open函数，出现的异常交给调用者处理，文件存在了，就不用再创建了，看是否修改还是删除
例如，自己写了一个类，使用了open函数，但是出现了异常不知道如何处理，就继续向外层抛出，一般来说最外层也是边界，必须处理这个异常了，否则线程退出

else子句

```
try:
    ret = 1 * 0
except ArithmeticError as e:
    print(e)
else:
    print('OK')
finally:
    print('fin')
```


else子句
没有任何异常发生，则执行

总结

```
try:
    <语句>      #运行别的代码
except <异常类>:
    <语句>      # 捕获某种类型的异常
except <异常类> as <变量名>:
    <语句>      # 捕获某种类型的异常并获得对象
else:
    <语句>      #如果没有异常发生
```

```
finally:
    <语句>      #退出try时总会执行
```

try的工作原理

1、如果try中语句执行时发生异常，搜索except子句，并执行第一个匹配该异常的except子句

2、如果try中语句执行时发生异常，却没有匹配的except子句，异常将被递交到外层的try，如果外层不处理这个异常，异常将继续向外层传递。如果都不处理该异常，则会传递到最外层，如果还没有处理，就终止异常所在的线程

3、如果在try执行时没有发生异常，将执行else子句中的语句

4、无论try中是否发生异常，finally子句最终都会执行