Python代码结构

错误和异常

什么是异常

错误

错误可以分为两种，语法上的和逻辑上的，当检测到一个错误时，Python解释器会指出当前流已经无法继续执行下去。这个时候就出现了异常(Exception)。

异常

对异常的最好描述是：它是因为程序出现了错误而在正常控制流之外采取的行为，这个行为分为两个阶段。

- 引起异常发生的错误
- 检测（和采取可能措施）阶段

当程序运行过程中遇到错误时就会抛出一个异常（异常也可以由程序员主动触发）。其可以被异常控制语句捕捉并予以处理。异常如果不能被捕捉和处理，就会以错误的形式呈现出来。

在类C语言（包括C语言）的程序设计中，程序员必须尽量考虑到各种错误情况，从而通过代码设计增强代码健壮性，然而总是有些特殊的会引发错误的情况无法考虑到，这就导致了错误发生时，只能眼睁睁的看着程序崩溃掉。异常的引入改变了这一情况，错误引发的异常可以被分类捕捉并在程序主流程以外加以处理，异常处理后，程序重新回到主流程中运行。对于像交换机、路由器这样的需要不间断运行的设备，出现错误并挂掉是不允许的，异常处理便显得极为重要。

Python中的几种常见异常

- `NameError`：尝试访问一个未声明的变量

```
foo
```

- `ZeroDivisionError`：除数为0

```
1/0
```

- `SyntaxError`：解释器语法错误

```
for
```

- `IndexError`：请求的索引超出序列范围

```
alist = [0, 1, 2]
alist[4]
```

- `KeyError`：请求一个不存在的字典键

```
adict = {'a': 1, 'b': 2}
adict['c']
```

- `FileNotFoundError`：输入/输出错误

```
afile = open('not_exists.txt')
```

- `AttributeError`：尝试访问未知的对象属性

```
'abc'.good()
```

检测和处理异常

`try-except`语句

语法：

```
try:
    try_suite   # 监控这里的异常
except exceptiontype as name:
    except_suite  # 异常处理代码
```

```
try:
    f = open('not_exists.txt')
except FileNotFoundError as err:
    print('File does not exist.')
```

程序执行时，尝试执行`try`中的代码，如果代码块中没有发现任何异常，则忽略`except`代码块中的内容，反之，如果发生的异常和`except`中要捕捉的异常类型相同。则调用其代码块中的代码对异常进行处理。处理后，返回程序的主流程中。

如果`try`中产生的异常在`except`中无法被捕捉（类型不匹配），异常就会被递交至上一级，也就是由该段代码的调用者去处理。如果最后还是无法解决的话，就会出现错误，导致程序崩溃。

`try-except-except-...`语句（多`except`）

语法：

```
try:
    try_suite   # 监控这里的异常
except exceptiontype1 as name1:
    except_suite1  # 异常处理代码
except exceptiontype2 as name2:
    except_suite2  # 异常处理代码
...
```

有时候`try`代码中可能发生的异常有多种类型，我们可以针对不同类型的异常分类捕捉并予以处理。下面的例子定义了一个浮点类型转换函数，对于各种非法参数可以返回“标准的waring”：

```
def safe_float(obj):
    try:
        retval = float(obj)
    except ValueError:  # 异常：类型正确，值不正确
        retval = 'could not convert non-number to float'
    except TypeError:   # 异常：类型不正确
        retval = 'object type cannot be converted to float'
    return retval

print(safe_float('xyz'))
print(safe_float(()))
print(safe_float(200))
```

处理多个异常的`except`语句

语法：

```
try:
    try_suite
except (Exc1,[, Exc2[, ... ExcN]])[as reason]:
    suite_for_exception1_to_excN
```

这种语法使得我们可以对多种异常类型使用相同的处理方法。使用这种语法重写的`safe_float()`如下：

```
def safe_float(obj):
    try:
        retval = float(obj)
    except (ValueError, TypeError):
        retval = 'arguments must be a number or numeric string'
    return retval

print(safe_float('xyz'))
print(safe_float(()))
print(safe_float(200))
```

现在，对于不同的类型错误返回的是相同的错误提示字符串。

捕获所有异常

语法：

```
try:
    ...
except [Exception] [as reason]:
    ...
```

在异常的继承树结构中，Exception是所有错误引发的异常的基类。当然，这里不包括`SystemExit`（当前应用程序需要退出）、`KeyboardInterupt`（用户按下了Ctrl+C键）这两种异常。

真正的异常基类是`BaseException`，其有3个子类，分别为`Exception`，`SystemExit`，`KeyboardInterrupt`。

所以，如果要真正捕捉“所有”的异常，`except`语句应该写：`except BaseException as e`：

所以，通常的代码框架以如下写法进行：

```
try:
    ...
except (KeyboardInterrupt, SystemExit):
    #用户希望退出
    raise   # 将异常上交给 caller
except Exception:
    # 处理真正的错误
```

异常参数

异常有参数，前面的例子中，在`except`后你经常看到有个`e`，这个`e`就是异常参数，它是指示了异常原因的一个字符串。虽然在前面的很多例子中异常参数都被忽略了，但在实际编码过程中，将参数字符串和自己设定的错误信息一起输出是一个好的习惯。

```
...
except ValueError as e:
    print('valid value', e)
```

`else`子句

```
try:
    ...
except:
    ...
else:
    ...
```

`try`代码块中没有异常检测到时，执行`else`子句。

`finally`子句

```
try:
    ...
except MyException:
    ...
else:
    ...
finally:
    ...
```

无论异常是否发生，无论异常是否被捕捉到，`finally`后的语句块一定会被执行。诸如关闭文件，断开数据库连接之类的语句理所当然应当放在这一部分。

`try-finally`子句

```
try:
    try_suite
finally:
    finally_suite
```

这种用法和`try-except`的区别主要在于它不是用来捕捉异常的，而是用于保证无论异常是否发生，`finally`代码段都会执行。

综合

```
try:
    try_suite
except Exception1:
    suite_for_exception1
except (Exception2, Exc3, Exc4):
    suite_for_exc2_to_4
except (Exc5, Exc6) as argu_for_56:
    suite_for_exc5_to_6
except:
    suite_for_other_exceptions
else:
    no_exception_detected_suite
finally:
    always_excute_suite
```

触发异常

到目前为止，看到的异常都是由Python解释器引起的，由执行期间的错误引发。很多情况下，你编写自己的类或者API，需要在遇到错误输入的时候触发异常，Python提供了`raise`语句来实现这一机制。

语法：

```
raise [expression [from expression]]
```

如果`raise`后面没有跟表达式，`raise`将重新引发当前作用域中的最后一个异常。如果当前作用域中没有异常，将引发一个`RuntimeError`来提示有错误发生。

其他情况下，`raise`将执行第一个表达式作为异常对象。它必须是`BaseException`的子类或者一个实例。如果它是一个类，那么将通过不带任何参数的初始化这个类来获取其实例对象。

`from`语句被用来进行异常链追踪，其后的表达式必须是另外一个异常类或者实例。

```
try:
    print(1 / 0)
except Exception as exc:
    raise RuntimeError("Something bad happened") from exc
```

```
try:
    print(1 / 0)
except:
    raise RuntimeError("Something bad happened")
```

创建异常

当系统的内建异常不能满足你的需求时，就需要动手创建自己的异常。异常的创建实质是异常类的设计。这里涉及到了面向对象编程中类的设计，因此不再这里讲述，在后面会讲到。

```
class UppercaseException(Exception):
    pass

words = ['eenie', 'meenie', 'miny', 'MO']
for word in words:
    if word.isupper():
        raise UppercaseException(word)
```

学习异常就是学习如何解决程序中的各种有意、无意引入的错误，这对于提高程序健壮性，减少bugs是非常重要的。程序设计中，使用异常框架来管理诸如数据库连接，文件操作，GUI响应等极容易发生异常的过程是必须的。