

19. 核能来袭-约束和异常处理

本节主要内容:

1. 类的约束
2. 异常处理
3. 自定义异常
4. MD5加密
5. 日志

一. 类的约束

首先,你要清楚.约束是对类的约束.比如.现在.你是一个项目经理.然后呢.你给手下的人分活.张三,你处理一下普通用户登录,李四,你处理一下会员登录,王五,你处理一下管理员登录.那这个时候呢.他们就开始分别取写他们的功能了.但是呢.你要知道,程序员不一定会有那么好的默契.很有可能三个人会写完全三个不同的方法.就比如这样:

```
class Normal:    # 张三, 普通人登录
    def login(self):
        pass

class Member:    # 李四, 会员登录
    def denglu(self):
        pass

class Admin:     # 王五, 管理员登录
    def login(self):
        pass
```

然后呢,他们把这样的代码交给你了.你看了一眼.张三和王五还算OK 这个李四写的是什么鬼? denglu.....难受不.但是好歹能用.还能凑合.但是这时.你这边要使用了.问题就来了.

```
# 项目经理写的总入口
def login(obj):
    print("准备验证码.....")
    obj.login()
    print("进入主页.....")
```

对于张三和王五的代码.没有问题.但是李四的.你是不是调用不了.那如何避免这样的问题呢?我们要约束程序的结构.也就是说.在分配任务之前就应该把功能定义好.然后分别交给底下的程序员来完成相应的功能.

在python中有两种办法来解决这样的问题.

1. 提取父类.然后在父类中定义好方法.在这个方法中什么都不用干.就抛一个异常就可以了.这样所有的子类都必须重写这个方法.否则.访问的时候就会报

错.

2. 使用元类来描述父类. 在元类中给出一个抽象方法. 这样子类就不得不给出抽象方法的具体实现. 也可以起到约束的效果.

首先, 我们先看第一种解决方案: 首先, 提取一个父类. 在父类中给出一个方法. 并且在方法中不给出任何代码. 直接抛异常.

```
class Base:
    def login(self):
        raise Exception("你没有实现login方法()")

class Normal(Base):
    def login(self):
        pass

class Member(Base):
    def denglu(self):
        pass

class Admin(Base):
    def login(self):
        pass

# 项目经理写的总入口
def login(obj):
    print("准备验证码.....")
    obj.login()
    print("进入主页.....")

n = Normal()
m = Member()
a = Admin()
login(n)
login(m)    # 报错.
login(a)
```

在执行到login(m)的时候程序会报错. 原因是, 此时访问的login()是父类中的方法. 但是父类中的方法会抛出一个异常. 所以报错. 这样程序员就不得不写login方法了. 从而对子类进行了相应的约束.

在本示例中. 要注意. 我们抛出的是Exception异常. 而Exception是所有异常的根. 我们无法通过这个异常来判断出程序是因为什么报的错. 所以. 最好是换一个比较专业的错误信息. 最好是换成NotImplementError. 其含义是. "没有实现的错误". 这样程序员或者项目经理可以一目了然的知道是什么错了. 就好比. 你犯错了. 我就告诉你犯错了. 你也不知道哪里错了. 这时我告诉你, 你xxx错了. 你改也好改不是?

第二套方案: 写抽象类和抽象方法. 这种方案相对来说比上一个麻烦一些. 需要给大家先

引入一个抽象的概念. 什么是抽象呢? 想一下. 动物的吃. 你怎么描述? 一个动物到底应该怎么吃? 是不是描述不清楚. 这里动物的吃就是一个抽象的概念. 只是一个动作的概念. 没有具体实现. 这种就是抽象的动作. 换句话说. 我们如果写一个方法. 不知道方法的内部应该到底写什么. 那这个方法其实就应该是一个抽象的方法. 如果一个类中包含抽象方法. 那么这个类一定是一个抽象类. 抽象类是不能有实例的. 比如. 你看看一些抽象派的画作. 在现实中是不存在的. 也就无法建立实例对象与之相对应. 所以抽象类无法创建对象. 创建对象的时候会报错.

在python中编写一个抽象类比较麻烦. 需要引入abc模块中的ABCMeta和abstractmethod这两个内容. 来看一个例子.

```
from abc import ABCMeta, abstractmethod

# 类中包含了抽象方法. 那此时这个类就是个抽象类. 注意: 抽象类可以有普通方法
class IGame(metaclass=ABCMeta):
    # 一个游戏到底怎么玩儿? 你能形容? 流程能一样么?
    @abstractmethod
    def play(self):
        pass

    def turn_off(self):
        print("破B游戏不玩了, 脱坑了")

class DNFGGame(IGame):
    # 子类必须实现父类中的抽象方法. 否则子类也是抽象类
    def play(self):
        print("dnf的玩儿法")

# g = IGame() # 抽象类不能创建对象

dg = DNFGGame()
dg.play()
```

通过代码我们能发现. 这里的IGame对DNFGGame进行了约束. 换句话说. 父类对子类进行了约束.

接下来. 继续解决我们一开始的问题.

```
from abc import ABCMeta, abstractmethod

class Base(metaclass=ABCMeta):
    @abstractmethod
    def login(self):
        pass

class Normal(Base):
    def login(self):
```

```

        pass

class Member(Base):
    def denglu(self):    # 这个就没用了
        pass

    def login(self):    # 子类对父类进行实现
        pass

class Admin(Base):
    def login(self):
        pass

# 项目经理写的总入口
def login(obj):
    print("准备验证码.....")
    obj.login()
    print("进入主页.....")

n = Normal()
m = Member()
a = Admin()
login(n)
login(m)
login(a)

```

总结: 约束. 其实就是父类对子类进行约束. 子类必须要写xxx方法. 在python中约束的方式和方法有两种:

1. 使用抽象类和抽象方法, 由于该方案来源是java和c#. 所以使用频率还是很少的
2. 使用人为抛出异常的方案. 并且尽量抛出的是NotImplementError. 这样比较专业, 而且错误比较明确.(推荐)

二. 异常处理

首先, 我们先说一下, 什么是异常? 异常是程序在运行过程中产生的错误. 就好比. 你在回家路上突然天塌了. 那这个就属于一个异常. 总之就是不正常. 那如果程序出现了异常. 怎么处理呢? 在之前的学习中我们已经写过类似的代码了.

我们先制造一个错误. 来看看异常长什么样.

```

def chu(a, b):
    return a/b

ret = chu(10, 0)
print(ret)

```

结果：

Traceback (most recent call last):

File "/Users/sylar/PycharmProjects/oldboy/面向对象/day05.py", line 100, in
<module>

ret = chu(10, 0)

File "/Users/sylar/PycharmProjects/oldboy/面向对象/day05.py", line 98, in
chu

return a/b

ZeroDivisionError: division by zero

什么错误呢. 除法中除数不能是0. 那如果真的出了这个错. 你把这一堆信息抛给客户么? 肯定不能. 那如何处理呢?

```
def chu(a, b):  
    return a/b
```

```
try:
```

```
    ret = chu(10, 0)
```

```
    print(ret)
```

```
except Exception as e:
```

```
    print("除数不能是0")
```

结果：

除数不能是0

那try...except是什么意思呢? 尝试着运行xxxxx代码. 出现了错误. 就执行except后面的代码. 在这个过程中. 当代码出现错误的时候. 系统会产生一个异常对象. 然后这个异常会向外抛. 被except拦截. 并把接收到的异常对象赋值给e. 那这里的e就是异常对象. 那这里的Exception是什么? Exception是所有异常的基类, 也就是异常的跟. 换句话说. 所有的错误都是Exception的子类对象. 我们看到的ZeroDivisionError 其实就是Exception的子类. 那这样写好像有点儿问题撒. Exception表示所有的错误. 太笼统了. 所有的错误都会被认为是Exception. 当程序中出现多种错误的时候, 就不好分类了, 最好是出什么异常就用什么来处理. 这样就更加合理了. 所以在try...except语句中. 还可以写更多的except

```
try:
```

```
    print("各种操作....")
```

```
except ZeroDivisionError as e:
```

```
    print("除数不能是0")
```

```
except FileNotFoundError as e:
```

```
    print("文件不存在")
```

```
except Exception as e:
```

```
    print("其他错误")
```

此时. 程序运行过程中. 如果出现了ZeroDivisionError就会被第一个except捕获. 如果出现了FileNotFountError就会被第二个except捕获. 如果都不是这两个异常. 那就会被最后的Exception捕获. 总之最后的Exception就是我们异常处理的最后一个守门员. 这时我们最常用的一套写法. 接下来. 给出一个完整的异常处理写法(语法):

```
try:
```

```

'''操作'''
except Exception as e:
    '''异常的父类，可以捕获所有的异常'''
else:
    '''保护不抛出异常的代码，当try中无异常的时候执行'''
finally:
    '''最后总是要执行我'''

```

解读：程序先执行操作，然后如果出错了会走except中的代码。如果不出错，执行else中的代码。不论处不出错，最后都要执行finally中的语句。一般我们用try...except就够用了。顶多加上finally，finally一般用来作为收尾工作。

上面是处理异常。我们在执行代码的过程中如果出现了一些条件上的不对等，根本不符合我的代码逻辑。比如，参数。我要求你传递一个数字，你非得传递一个字符串，那对不起，我没办法帮你处理。那如何通知你呢？两个方案。

方案一，直接返回即可。我不管你还不行么？

方案二，抛出一个异常，告诉你，我不好惹，乖乖的听话。

第一种方案是我们之前写代码经常用到的方案，但这种方案并不够好，无法起到警示作用。所以，以后的代码中如果出现了类似的问题，直接抛一个错误出去，那怎么抛呢？我们要用到raise关键字

```

def add(a, b):
    '''
    给我传递两个整数，我帮你计算两个数的和
    :param a:
    :param b:
    :return:
    '''
    if not type(a) == int and not type(b) == int:
        # 当程序运行到这句话的时候，整个函数的调用会被中断，并向外抛出一个异常。
        raise Exception("不是整数，朕不能帮你搞定这么复杂的运算。")
    return a + b

# 如果调用方不处理异常，那产生的错误将会继续向外抛，最后就抛给了用户
# add("你好", "我叫赛利亚")

# 如果调用方处理了异常，那么错误就不会丢给用户，程序也能正常进行
try:
    add("胡辣汤", "滋滋冒油的大腰子")
except Exception as e:
    print("报错了，自己处理去吧")

```

当程序运行到raise，程序会被中断，并实例化后面的异常对象，抛给调用方。如果调用方不处理，则会把错误继续向上抛出，最终抛给用户。如果调用方处理了异常，那程序可以正常的进行执行。

说了这么多，异常也知道如何抛出和处理了。但是我们现在用的都是人家python给的异常。如果某一天，你写的代码中出现了一个无法用现有的异常来解决问题，那怎么办呢？别着

急. python可以自定义异常.

自定义异常: 非常简单. 只要你的类继承了Exception类. 那你的类就是一个异常类. 就这么简单. 比如. 你要写一个男澡堂子程序. 那这时要是来个女的. 你怎么办? 是不是要抛出一个性别异常啊? 好. 我们来完成这个案例:

```
# 继承Exception. 那这个类就是一个异常类
class GenderError(Exception):
    pass

class Person:

    def __init__(self, name, gender):
        self.name = name
        self.gender = gender

def nan_zao_tang_xi_zao(person):
    if person.gender != "男":
        raise GenderError("性别不对. 这里是男澡堂子")

p1 = Person("alex", "男")
p2 = Person("eggon", "蛋")

# nan_zao_tang_xi_zao(p1)
# nan_zao_tang_xi_zao(p2) # 报错. 会抛出一个异常: GenderError

# 处理异常
try:
    nan_zao_tang_xi_zao(p1)
    nan_zao_tang_xi_zao(p2)
except GenderError as e:
    print(e) # 性别不对, 这里是男澡堂子
except Exception as e:
    print("反正报错了")
```

ok搞定. 但是, 如果是真的报错了. 我们在调试的时候, 最好是能看到错误源自于哪里? 怎么办呢? 需要引入另一个模块traceback. 这个模块可以获取到我们每个方法的调用信息. 又被成为堆栈信息. 这个信息对我们拍错是很有帮助的.

```
import traceback
# 继承Exception. 那这个类就是一个异常类
class GenderError(Exception):
    pass

class Person:

    def __init__(self, name, gender):
        self.name = name
        self.gender = gender
```

```

def nan_zao_tang_xi_zao(person):
    if person.gender != "男":
        raise GenderError("性别不对. 这里是男澡堂子")

p1 = Person("alex", "男")
p2 = Person("eggon", "蛋")

# nan_zao_tang_xi_zao(p1)
# nan_zao_tang_xi_zao(p2) # 报错. 会抛出一个异常: GenderError

# 处理异常
try:
    nan_zao_tang_xi_zao(p1)
    nan_zao_tang_xi_zao(p2)
except GenderError as e:
    val = traceback.format_exc() # 获取到堆栈信息
    print(e) # 性别不对. 这里是男澡堂子
    print(val)
except Exception as e:
    print("反正报错了")

```

结果:

性别不对. 这里是男澡堂子

Traceback (most recent call last):

File "/Users/sylar/PycharmProjects/oldboy/面向对象/day05.py", line 155, in
<module>

nan_zao_tang_xi_zao(p2)

File "/Users/sylar/PycharmProjects/oldboy/面向对象/day05.py", line 144, in
nan_zao_tang_xi_zao

raise GenderError("性别不对. 这里是男澡堂子")

GenderError: 性别不对. 这里是男澡堂子

搞定了. 这样我们就能收放自如了. 当测试代码的时候把堆栈信息打印出来. 但是当到了线上的生产环境的时候把这个堆栈去掉即可.

四. MD5加密

想一个事情. 你在银行取钱或者办卡的时候. 我们都要输入密码. 那这个密码如果就按照我们输入的那样去存储. 是不是很不安全啊. 如果某一个程序员进入到了银行的数据库. 而银行的数据库又存的都是明文(不加密的密码)密码. 这时, 整个银行的账户里的信息都是非常非常不安全的. 那怎么办才安全呢? 给密码加密. 并且是不可逆的加密算法. 这样. 即使获取到了银行的账户和密码信息. 对于黑客而言都无法进行破解. 那我们的账号就相对安全了很多. 那怎么加密呢? 最常见的就是用MD5算法.

MD5是一种不可逆的加密算法. 它是可靠的. 并且安全的. 在python中我们不需要手写这一套算法. 只需要引入一个叫hashlib的模块就能搞定MD5的加密工作

```
import hashlib
```



```
obj = hashlib.md5()
obj.update("alex".encode("utf-8")) # 加密的必须是字节
miwen = obj.hexdigest()
print(miwen) # 534b44a19bf18d20b71ecc4eb77c572f
```

那这样的密文安全么？其实是不安全的. 当我们用这样的密文去找一个所谓的MD5解密工具. 是有可能解密成功的.

密文:	<input type="text" value="534b44a19bf18d20b71ecc4eb77c572f"/>
类型:	<input type="text" value="自动"/> [帮助]
<div>查询 加密</div>	
查询结果: alex [添加备注]	

这就尴尬了. MD5不是不可逆么? 注意. 这里有一个叫撞库的问题. 就是. 由于MD5的原始算法已经存在很久了. 那就有一些人用一些简单的排列组合来计算MD5. 然后当出现相同的MD5密文的时候就很容易反推出原来的数据是什么. 所以并不是MD5可逆, 而是有些别有用心的人把MD5的常见数据已经算完并保留起来了.

那如何应对呢? 加盐就行了. 在使用MD5的时候. 给函数的参数传递一个byte即可.

```
import hashlib

obj = hashlib.md5(b"fjllksajflkjasfsalwer123dfskjf") # 加盐
obj.update("alex".encode("utf-8")) # 加密的必须是字节
miwen = obj.hexdigest()
print(miwen) # 99fca4b872fa901aac30c3e952ca786d
```

此时你再去任何网站去试. 累死他也解不开密.

那MD5如何应用呢?

```
import hashlib

def my_md5(s):
    obj = hashlib.md5(b"fjllksajflkjasfsalwer123dfskjf")
    obj.update(s.encode("utf-8")) # 加密的必须是字节
    miwen = obj.hexdigest()
    return miwen

# alex: 99fca4b872fa901aac30c3e952ca786d
username = input("请输入用户名:")
password = input("请输入密码:")
# 数据存储的时候.
# username: my_md5(password)
```

```
# 假设现在的用户名和密码分别是
# wusir: 99fca4b872fa901aac30c3e952ca786d ==> wusir: alex

# 用户登录
if username == "wusir" and my_md5(password) ==
"99fca4b872fa901aac30c3e952ca786d":
    print("成功")
else:
    print("失败")
```

所以, 以后存密码就不要存明文了. 要存密文. 安全, 并且. 这里加的盐不能改来改去的. 否则, 整套密码就都乱了.

五. 日志

首先, 你要知道在编写任何一款软件的时候, 都会出现各种各样的问题或者bug. 这些问题或者bug一般都会在测试的时候给处理掉. 但是多多少少的都会出现一些意想不到的异常或者错误. 那这个时候, 我们是不知道哪里出了问题的. 因为很多BUG都不是必现的bug. 如果是必现的. 测试的时候肯定能测出来. 最头疼的就是这种不必现的bug. 我这跑没问题. 客户那一用就出问题. 那怎么办呢? 我们需要给软件准备一套日志系统. 当出现任何错误的时候. 我都可以去日志系统里去查. 看哪里出了问题. 这样在解决问题和bug的时候就多了一个帮手. 那如何在python中创建这个日志系统呢? 很简单.

1. 导入logging模块.
2. 简单配置一下logging
3. 出现异常的时候(exception). 向日志里写错误信息.

```
# filename: 文件名
# format: 数据的格式化输出. 最终在日志文件中的样子
#      时间-名称-级别-模块: 错误信息
# datefmt: 时间的格式
# level: 错误的级别权重, 当错误的级别权重大于等于level的时候才会写入文件
logging.basicConfig(filename='x1.txt',
                    format='%(asctime)s - %(name)s - %(levelname)s - %(
(module)s: %(message)s',
                    datefmt='%Y-%m-%d %H:%M:%S',
                    level=0) # 当前配置表示 10以上的分数会被写入文件

# CRITICAL = 50
# FATAL = CRITICAL
# ERROR = 40
# WARNING = 30
# WARN = WARNING
# INFO = 20
# DEBUG = 10
# NOTSET = 0
logging.critical("我是critical") # 50分. 最贵的
logging.error("我是error") # 40分
```

```

logging.warning("我是警告")    # 警告 30
logging.info("我是基本信息")    # 20
logging.debug("我是调试")    # 10

logging.log(2, "我是自定义")    # 自定义. 看着给分

```

简单做个测试, 应用一下

```

class JackError(Exception):
    pass

for i in range(10):

    try:
        if i % 3 == 0:
            raise FileNotFoundError("文件不在啊")
        elif i % 3 == 1:
            raise KeyError("键错了")
        elif i % 3 == 2:
            raise JackError("杰克Exception")
    except FileNotFoundError:
        val = traceback.format_exc()
        logging.error(val)
    except KeyError:
        val = traceback.format_exc()
        logging.error(val)
    except JackError:
        val = traceback.format_exc()
        logging.error(val)
    except Exception:
        val = traceback.format_exc()
        logging.error(val)

```

最后, 如果你系统中想要把日志文件分开. 比如. 一个大项目, 有两个子系统, 那两个子系统要分开记录日志. 方便调试. 那怎么办呢? 注意. 用上面的basicConfig是搞不定的. 我们要借助文件助手(FileHandler), 来帮我们完成日志的分开记录

```

import logging

# 创建一个操作日志的对象logger (依赖FileHandler)
file_handler = logging.FileHandler('l1.log', 'a', encoding='utf-8')
file_handler.setFormatter(logging.Formatter(fmt="%(asctime)s - %(name)s - %(levelname)s - %(module)s: %(message)s"))

logger1 = logging.Logger('s1', level=logging.ERROR)
logger1.addHandler(file_handler)

logger1.error('我是A系统')

```

```
# 再创建一个操作日志的对象logger (依赖FileHandler)
file_handler2 = logging.FileHandler('l2.log', 'a', encoding='utf-8')
file_handler2.setFormatter(logging.Formatter(fmt="%(asctime)s - %(name)s -
%(levelname)s -%(module)s: %(message)s"))

logger2 = logging.Logger('s2', level=logging.ERROR)
logger2.addHandler(file_handler2)

logger2.error('我是B系统')
```