tech_summary

Release 1.0

Frank

Dec 07, 2020

CONTENTS:

CHAPTER

ONE

PYTHON3 中的特性 PROPERTY 介绍

[TOC]

1.1 特性的引入

1.1.1 特性和属性的区别是什么?

特性与属性的区别是什么?

在 python 中属性这个实例方法, 类变量都是属性. 属性, attribute

在 python 中数据的属性和处理数据的方法都可以叫做属性. 简单来说在一个类中, 方法是属性, 数据也是属性.

```
class Animal:
    name = 'animal'

    def bark(self):
        print('bark')
        pass

    @classmethod
    def sleep(cls):
        print('sleep')
        pass

    @staticmethod
    def add():
        print('add')
```

在命令行里面执行

```
>>> animal = Animal()
>>> animal.add()
```

(continues on next page)

(continued from previous page)

```
add
>>> animal.sleep()
sleep
>>> animal.bark()
bark
>>> hasattr(animal,'add') #1
True
>>> hasattr(animal,'sleep')
True
>>> hasattr(animal,'bark')
True
```

可以看出 #1 animal 中是可以拿到 add ,sleep bark 这些属性的.

特性: property 这个是指什么? 在不改变类接口的前提下使用存取方法 (即读值和取值) 来修改数据的属性.

什么意思呢?

就是通过 obj.property 来读取一个值, obj.property = xxx , 来赋值

还以上面 animal 为例:

```
class Animal:

    @property
    def name(self):
        print('property name ')
        return self._name

    @name.setter
    def name(self, val):
        print('property set name ')
        self._name = val

    @name.deleter
    def name(self):
        del self._name
```

这个时候 name 就是了特性了.

```
>>> animal = Animal()
>>> animal.name='dog'
property set name
>>> animal.name
property name
'dog'
```

(continues on next page)

(continued from previous page)

```
>>>
>>> animal.name='cat'
property set name
>>> animal.name
property name
'cat'
```

肯定有人会疑惑, 写了那么多的代码, 还不如直接写成属性呢, 多方便.

比如这段代码: 直接把 name 变成类属性这样做不是很好吗, 多简单. 这样写看起来也没有太大的问题. 但是如果给 name 赋值成数字这段程序也是不会报错. 这就是比较**大的问题**了.

```
>>> class Animal:
...     name=None
...
>>> animal = Animal()
>>> animal.name
>>> animal.name='frank'
>>> animal.name
'frank'
>>> animal.name='chang'
>>> animal.name
'chang'
>>> animal.name=250
>>> animal
<Animal object at 0x10622b850>
>>> animal.name
250
>>> type(animal.name)
<class 'int'>
```

这里给 `animal.name` 赋值成 250, 程序从逻辑上来说没有问题. 但其实这样赋值是毫无意义的.

我们一般希望不允许这样的赋值, 就希望给出 **报错或者警告**之类的.

```
animal= Animal()
animal.name=100
property set name
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 13, in name
ValueError: expected val is str
```

其实当 name 变成了 property 之后, 我们就可以对 name 赋值进行控制. 防止一些非法值变成对象的属性. 比如说 name 应该是这个字符串, 不应该是数字这个时候就可以在 setter 的时候进行判断, 来控制能否赋值.

要实现上述的效果, 其实也很简单 setter 对 value 进行判断就好了.

```
class Animal:

    @property
    def name(self):
        print('property name ')
        return self._name

    @name.setter
    def name(self, val):
        print('property set name ')
        # 这里 对 value 进行判断
        if not isinstance(val,str):
            raise  ValueError("expected val is str")
        self._name = val
```

感受到特性的魅力了吧, 可以通过赋值的时候, 对值进行校验, 方式不合法的值, 进入到对象的属性中. 下面看下如何设置只读属性, 和如何设置读写特性.

> 假设有这个的一个需求, 某个类的属性一个初始化之后就不允许被更改, 这个就可以用特性这个问题, 比如一个人身高是固定, 一旦初始化后, 就不允许改掉.

1.1.2 设置只读特性

```
class Frank:
    def __init__(self, height):
        self._height = height

    @property
    def height(self):
        return self._height
```

```
>>> frank = Frank(height=100)
>>> frank.height
100
>>> frank.height =150
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: can't set attribute
```

这里初始化 frank 后就不允许就修改这个 height 这个值了. (实际上也是可以修改的) 重新给 height 赋值就会报错, 报错 AttributeError , 这里不实现 setter 就可以了.

1.1.3 设置读写特性

```
class Frank:
    def __init__(self, height):
        self._height = height

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, value):
        """
        给特性赋值
        """
        self._height = value
```

```
>>>
>>> frank = Frank(height=100)
>>> frank.height
100
>>> frank.height=165
>>> frank.height
165
```

比如对人的身高在 1 米到 2 米之间这样的限制

1.1.4 对特性的合法性进行校验

```
class Frank:

    def __init__(self, height):
        self.height = height  # 注意这里写法

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, value):
        """
        判断逻辑 属性的处理逻辑
        定义 了 setter 方法之后就  修改 属性 了.
```

(continues on next page)

(continued from previous page)

```
        判断 属性 是否合理 , 不合理直接报错. 阻止赋值, 直接抛异常

        :param value:
        :return:
        """
        if not isinstance(value, (float,int)):
            raise ValueError("高度应该是 数值类型")
        if value < 100 or value > 200:
            raise ValueError("高度范围是 100cm 到 200cm")
        self._height = value
```

在 python console 里面测试结果:

```
>>> frank = Frank(100)
>>> frank.height
100
>>> frank.height='aaa'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 21, in height
ValueError: 高度应该是 数值类型
>>> frank.height=250
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 23, in height
ValueError: 高度范围是 100cm 到 200cm
```

这样就可以进行严格的控制, 一些特性的方法性, 通过写 setter 方法来保证数据准确性, 防止一些非法的数据进入到实例中.

1.2 property 是什么?

实际是一个类, 然后就是一个装饰器. 让一个方法变成一个特性. 假设某个类的实例方法 bark 被 property 修饰了后, 调用方式就会发生变化.

```
obj.bark() --> obj.bak
```

其实特性模糊了方法和数据的界限.

方法是可调用的属性, 而 property 是可定制化的’属性’. 一般方法的名称是一个动词 (行为). 而特性 property 应该是名词.

如果我们一旦确定了属性不是动作, 我们需要在标准属性和 property 之间做出选择.

一般来说你如果要控制 property 的访问过程, 就要用 property . 否则用标准的属性即可.

attribute 属性和 property 特性的区别在于当 property 被读取, 赋值, 删除时候, 自动会执行某些特定的动作.

peroperty 详解

特性都是类属性，但是特性管理的其实是实例属性的存取。– 摘自 fluent python

下面的例子来自 fluent python

看一下几个例子来说明几个特性和属性区别

```
>>> class Class:
            """
            data 数据属性和 prop 特性。
            """
...     data = 'the class data attr'
...
...     @property
...     def prop(self):
...         return 'the prop value'
...
>>>
>>> obj= Class()
>>> vars(obj)
{}
>>> obj.data
'the class data attr'
>>> Class.data
'the class data attr'
>>> obj.data ='bar'
>>> Class.data
'the class data attr'
```

实例属性遮盖类的数据属性, 就是说如果 obj.data 重新修改了, 类的属性不会被修改.

下面尝试 obj 实例的 prop 特性

```
>>> Class.prop
<property object at 0x110968ef0>
>>> obj.prop
'the prop value'
>>> obj.prop ='foo'
Traceback (most recent call last):
  File "<input>", line 1, in <module>
AttributeError: can't set attribute
>>> obj.__dict__['prop'] ='foo'
```

(continues on next page)

(continued from previous page)

```
>>> vars(obj)
{'data': 'bar', 'prop': 'foo'}
>>> obj.prop  #1
'the prop value'
>>> Class.prop ='frank'
>>> obj.prop
'foo'
```

我尝试修改 obj.prop 会直接报错, 这个容易理解, 因为 property 没有实现 setter 方法. 我直接修改 obj.__dict__

然后在 #1 的地方, 发现还是正常调用了特性, 而没有属性的值.

当我改变 Class.prop 变成一个属性的时候.

再次调用 obj.prop 才调用到了实例属性.

再看一个例子添加特性

```
class Class:
    data = 'the class data attr'

    @property
    def prop(self):
        return 'the prop value'
```

```
>>> obj.data
'bar'
>>> Class.data
'the class data attr'

# 把类的 data 变成 特性
>>> Class.data = property(lambda self:'the "data" prop value')
>>> obj.data
'the "data" prop value'
>>> del Class.data
>>> obj.data
'bar'
>>> vars(obj)
{'data': 'bar', 'prop': 'foo'}
```

改变 data 变成特性后, obj.data 也改变了. 删除这个特性的时候, obj.data 又恢复了.

本节的主要观点是，obj.attr 这样的表达式不会从 obj 开始寻找 attr，而是从 obj.__class__ 开始，而且，仅当类中没有名为 attr 的特性时, Python 才会在 obj 实例中寻找。这条规则适用于 **特性**.

property 实际上是一个类

```
def __init__(self, fget=None, fset=None, fdel=None, doc=None):
    pass
    # known special case of property.__init__
```

完成的要实现一个特性需要这 4 个参数, get , set ,del , doc 这些参数. 但实际上大部分情况下, 只要实现 get ,set 即可.

1.3 特性的两种写法

下面两种写法都是可行的.

1.3.1 第一种写法

使用装饰器 property 来修饰一个方法

```
# 方法 1
class Animal:

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

    @property
    def name(self):
        print('property name ')
        return self._name

    @name.setter
    def name(self, val):
        print('property set name ')

        if not isinstance(val, str):
            raise ValueError("expected val is str")
        self._name = val

    @name.deleter
    def name(self):
        del self._name
```

1.3.2 第二种写法

直接实现 set get delete 方法即可, 通过 property 传入这个参数

```
# 方法二
class Animal2:

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

    def _set_name(self, val):
        if not isinstance(val, str):
            raise ValueError("expected val is str")

        self._name = val

    def _get_name(self):
        return self._name

    def _delete_name(self):
        del self._name

    name = property(fset=_set_name, fget=_get_name,fdel= _delete_name,doc= "name 这是特
性描述")


if __name__ == '__main__':
    animal = Animal2('dog')
```

```
>>> animal = Animal2('dog')
>>>
>>> animal.name
'dog'
>>> animal.name
'dog'

>>> help(Animal2.name)
Help on property:

    name 这是特性描述

>>> animal.name='cat'
>>> animal.name
'cat'
```

1.4 常见的一些例子

1. 缓存某些值
2. 对一些值进行合法性校验.

1.4.1 对一些值进行合法性校验.

对一些特性赋值的时候进行合法性的校验, 前面都有举例子.

在举一个小例子比如有一个货物, 有重量和价格, 需要保证这两个属性是正数不能是 0 , 即 >0 的值.

好了有了刚刚代码的基础, 下面的代码就写好了.

基础版代码

```
class Goods:

    def __init__(self, name, weight, price):
        """

        :param name: 商品名称
        :param weight:  重量
        :param price: 价格
        """
        self.name = name
        self.weight = weight
        self.price = price

    def __repr__(self):

        return f"{self.__class__.__name__}(name={self.name},weight={self.weight},
↪price={self.price})"

    @property
    def weight(self):
        return self._weight

    @weight.setter
    def weight(self, value):
        if value < 0:
            raise ValueError(f"expected value > 0, but now value:{value}")

        self._weight = value
```

(continues on next page)

(continued from previous page)

```
    @property
    def price(self):
        return self._price

    @price.setter
    def price(self, value):
        if value < 0:
            raise ValueError(f"expected value > 0, but now value:{value}")
        self._price = value
```

```
>>> goods = Goods('apple', 10, 30)
...
>>> goods
Goods(name=apple,weight=10,price=30)
>>> goods.weight
10
>>> goods.weight=-10
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 26, in weight
ValueError: expected value > 0, but now value:-10
>>> goods.price
30
>>> goods.price=-3
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 37, in price
ValueError: expected value > 0, but now value:-3
>>> goods
Goods(name=apple,weight=10,price=30)
>>> goods.price=20
>>> goods
Goods(name=apple,weight=10,price=20)
```

代码可以正常的判断出来, 这些非法值了. 这样写有点问题是什么呢? 就是发现 weight ,price 判断值的逻辑几乎是一样的代码.. 都是判断是大于 0 吗? 然而我却写了两遍相同的代码.

优化版代码

有没有更好的解决方案呢?

是有的, 我们可以写一个工厂函数来返回一个 property , 这实际上是两个 property 而已.

下面就是工厂函数, 用来生成一个 property 的.

```
def validate(storage_name):
    """
    用来验证 storage_name 是否合法性 , weight  , price
    :param storage_name:
    :return:
    """
    pass

    def _getter(instance):
        return instance.__dict__[storage_name]

    def _setter(instance, value):
        if value < 0:
            raise ValueError(f"expected value > 0, but now value:{value}")

        instance.__dict__[storage_name] = value

    return property(fget=_getter, fset=_setter)
```

货物类就可以像下面这样写

```
class Goods:
    weight = validate('weight')
    price = validate('price')

    def __init__(self, name, weight, price):
        """

        :param name: 商品名称
        :param weight:  重量
        :param price: 价格
        """
        self.name = name
        self.weight = weight
        self.price = price

    def __repr__(self):
        return f"{self.__class__.__name__}(name={self.name},weight={self.weight},
↪price={self.price})"
```

这样看起来是不是比较舒服一点了.

```
>>> goods = Goods('apple', 10, 30)
>>> goods.weight
10
>>> goods.weight=-10
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 16, in _setter
ValueError: expected value > 0, but now value:-10
>>> goods
Goods(name=apple,weight=10,price=30)
>>> goods.price=-2
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "<input>", line 16, in _setter
ValueError: expected value > 0, but now value:-2
>>> goods
Goods(name=apple,weight=10,price=30)
```

可以看出代码可以正常的工作了.

1.4.2 缓存一些值

```
>>> from urllib.request import urlopen
...
...
... class WebPage:
...
...     def __init__(self, url):
...         self.url = url
...
...         self._content = None
...
...     @property
...     def content(self):
...         if not self._content:
...             print("Retrieving new page")
...             self._content = urlopen(self.url).read()[0:10]
...
...         return self._content
...
>>>
>>>
>>> url = 'http://www.baidu.com'
```

(continues on next page)

(continued from previous page)

```
>>> page = WebPage(url)
>>>
>>> page.content
Retrieving new page
b'<!DOCTYPE '
>>> page.content
b'<!DOCTYPE '
>>> page.content
b'<!DOCTYPE '
```

可以看出第一次调用了 urlopen 从网页中读取值, 第二次就没有调用 urlopen 而是直接返回 content 的内容.

1.5 总结

python 的特性算是 python 的高级语法, 不要因为到处都要用这个特性的语法. 实际上大部分情况是用不到这个语法的. 如果代码中, 需要对属性进行检查就要考虑用这样的语法了. 希望你看完之后不要认为这种语法非常常见, 事实上不是的. 其实更好的做法对属性检查可以使用描述符来完成. 描述符是一个比较大的话题, 本文章暂未提及, 后续的话, 可能会写一下关于描述的一些用法, 这样就能更好的理解 python, 更加深入的理解 python.

1.6 参考文档

- fluent python
- python3 面向对象编程
- Python 为什么要使用描述符? https://juejin.im/post/5cc4fbc0f265da0380437706

分享快乐, 留住感动. '2019-10-06 15:46:15' –frank

CHAPTER

TWO

PYTHON3 中风格规范

[TOC]

2.1 保持盲目的一致是头脑简单的表现

(A Foolish Consistency Is The Hobgoblin Of Little Minds)

Guido 的一个重要观点是代码被读的次数远多于被写的次数。这篇指南旨在提高代码的可读性，使浩瀚如烟的 Python 代码风格能保持一致。正如PEP 20那首《Zen of Python》的小诗里所说的：“可读性很重要 (Readability counts)”。

这本风格指南是关于一致性的。同风格指南保持一致性是重要的，但是同项目保持一致性更加重要，同一个模块和一个函数保持一致性则最为重要。

然而最最重要的是：要知道何时去违反一致性，因为有时风格指南并不适用。当存有疑虑时，请自行做出最佳判断。请参考别的例子去做出最好的决定。并且不要犹豫，尽管提问。

特别的：千万不要为了遵守这篇 PEP 而破坏向后兼容性！

如果有以下借口，则可以忽略这份风格指南：

1. 当采用风格指南时会让代码更难读，甚至对于习惯阅读遵循这篇 PEP 的代码的人来说也是如此。
2. 需要和周围的代码保持一致性，但这些代码违反了指南中的风格（可是时历史原因造成的）——尽管这可能也是一个收拾别人烂摊子的机会（进入真正的极限编程状态）。
3. 若是有问题的某段代码早于引入指南的时间，那么没有必要去修改这段代码。
4. 代码需要和更旧版本的 Python 保持兼容，而旧版本的 Python 不支持风格指南所推荐的特性。

2.2 Python 风格规范

2.3 分号

不要在行尾加分号, 也不要用分号将两条命令放在同一行.

```
import sys;import os ;

# 不要像下面一样写在一行
import sys, os
```

2.4 空行

顶级定义之间空两行, 方法定义之间空一行

顶级定义之间空两行, 比如函数或者类定义. 方法定义, 类定义与第一个方法之间, 都应该空一行.

函数或方法中, 某些地方要是你觉得合适, 就空一行.

2.5 代码布局 (Code Lay-Out)

2.5.1 缩进 (Indentation)

每个缩进级别采用 4 个空格。

连续行所包装的元素应该要么采用 Python 隐式续行，即垂直对齐于圆括号、方括号和花括号，要么采用悬挂缩进 (hanging indent)。采用悬挂缩进时需考虑以下两点：第一行不应该包括参数，并且在续行中需要再缩进一级以便清楚表示。

正确的例子:

```
# 同开始分界符 (左括号) 对齐
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# 续行多缩进一级以同其他代码区别
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# 悬挂缩进需要多缩进一级
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

错误的例子:

```
# 采用垂直对齐时第一行不应该有参数
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# 续行并没有被区分开，因此需要再缩进一级
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```

对于续行来说，4 空格的规则可以不遵守。

2.5.2 每行最大长度 (Maximum Line Length)

将所有行都限制在 79 个字符长度以内。

一些团队会强烈希望行长度比 79 个字符更长。当代码仅仅只由一个团队维护时，可以达成一致让行长度增加到 80 到 100 字符 (实际上最大行长是 99 字符)，注释和文档字符串仍然是以 72 字符换行。

Python 标准库比较传统，将行长限制在 79 个字符以内（文档字符串/注释为 72 个字符）。

一种推荐的换行方式是利用 Python 圆括号、方括号和花括号中的隐式续行。长行可以通过在括号内换行来分成多行。应该最好加上反斜杠来区别续行。

例外情况:

1. 长的导入模块语句
2. 注释里的 URL

如果一个文本字符串在一行放不下, 可以使用圆括号来实现隐式行连接:

使用括号

```
x = ('This will build a very long long '
     'long long long long long long string')
```

在注释中，如果必要，将长的 URL 放在一行上。

```
Yes:  # See details at
      # http://www.example.com/us/developer/documentation/api/content/v2.0/csv_file_
↪name_extension_full_specification.html

No:  # See details at
     # http://www.example.com/us/developer/documentation/api/content/\
     # v2.0/csv_file_name_extension_full_specification.html
```

```
with open('/path/to/some/file/you/want/to/read') as file_1, \
     open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

2.5.3 二元运算符之前还是之后换行？

(Should a line break before or after a binary operator?)

```
# 错误的例子: 运算符远离操作数
income = (gross_wages +
          taxable_interest +
          (dividends - qualified_dividends) -
          ira_deduction -
          student_loan_interest)
```

建议写法：

```
# 正确的例子: 更容易匹配运算符与操作数
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

2.6 源文件编码 (Source File Encoding)

Python 核心发行版中的代码应该一直使用 UTF-8 . python3 用 utf8 编码

2.7 模块引用 (Imports)

- Imports 应该分行写，而不是都写在一行，例如：

  ```
  # 分开写
  import os
  import sys

  # 不要像下面一样写在一行
  import sys, os
  ```

这样写也是可以的：

```
from subprocess import Popen, PIPE
```

- Imports 应该写在代码文件的开头，位于模块 (module) 注释和文档字符串 (docstring) 之后，模块全局变量 (globals) 和常量 (constants) 声明之前。

  Imports 应该按照下面的顺序分组来写：

  1. 标准库 imports
  2. 相关第三方 imports (从 pypi 下载的)
  3. 本地应用/库的特定 imports (自己写的)

  不同组的 imports 之前用空格隔开。

- 推荐使用绝对 (absolute)imports，因为这样通常更易读，在 import 系统没有正确配置（比如中的路径以 `sys.path` 结束）的情况下，也会有更好的表现（或者至少会给出错误信息）：

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

- 隐式的相对 imports 应该**永不**使用，并且 Python 3 中已经被去掉了。
-
- 当从一个包括类的模块中 import 一个类时，通常可以这样写：

  ```
  from myclass import MyClass
  from foo.bar.yourclass import YourClass
  ```

  如果和本地命名的拼写产生了冲突，应当直接 import 模块：

  ```
  import myclass
  import foo.bar.yourclass
  ```

  然后使用”myclass.MyClass”和”foo.bar.yourclass.YourClass”。

- 避免使用通配符 imports(`from <module> import *`)，因为会造成在当前命名空间出现的命名含义不清晰，给读者和许多自动化工具造成困扰。有一个可以正当使用通配符 import 的情形，即将一个内部接口重新发布成公共 API 的一部分（比如，使用备选的加速模块中的定义去覆盖纯 Python 实现的接口，预先无法知晓具体哪些定义将被覆盖）。

2.8 字符串引用 (String Quotes)

在 Python 中表示字符串时，不管用单引号还是双引号都是一样的。但是不推荐将这两种方式看作一样并且混用。最好选择一种规则并坚持使用。当字符串中包含单引号时，采用双引号来表示字符串，反之也是一样，这样可以避免使用反斜杠，代码也更易读。

对于三引号表示的字符串，使用双引号字符来表示，这样可以和PEP 257的文档字符串（docstring）规则保持一致。

在同一个文件中, 保持使用字符串引号的一致性. 使用单引号' 或者双引号" 之一用以引用字符串, 并在同一文件中沿用.

为多行字符串使用三重双引号"""

文档字符串必须使用三重双引号"""

2.9 文件和 sockets

除文件外, sockets 或其他类似文件的对象在没有必要的情况下打开, 会有许多副作用, 例如:

1. 它们可能会消耗有限的系统资源, 如文件描述符. 如果这些资源在使用后没有及时归还系统, 那么用于处理这些对象的代码会将资源消耗殆尽.
2. 持有文件将会阻止对于文件的其他诸如移动、删除之类的操作.
3. 仅仅是从逻辑上关闭文件和 sockets, 那么它们仍然可能会被其共享的程序在无意中进行读或者写操作. 只有当它们真正被关闭后, 对于它们尝试进行读或者写操作将会抛出异常, 并使得问题快速显现出来.

而且, 幻想当文件对象析构时, 文件和 sockets 会自动关闭, 试图将文件对象的生命周期和文件的状态绑定在一起的想法, 都是不现实的. 因为有如下原因:

1. 没有任何方法可以确保运行环境会真正的执行文件的析构. 不同的 Python 实现采用不同的内存管理技术, 比如延时垃圾处理机制. 延时垃圾处理机制可能会导致对象生命周期被任意无限制的延长.
2. 对于文件意外的引用, 会导致对于文件的持有时间超出预期 (比如对于异常的跟踪, 包含有全局变量等).

对于打开的文件，建议用 with 语句。

推荐使用 “with” 语句 以管理文件:

```
with open("hello.txt") as hello_file:
    for line in hello_file:
        print line
```

对于不支持使用” with” 语句的类似文件的对象, 使用 contextlib.closing():

```
import contextlib

with contextlib.closing(urllib.urlopen("http://www.python.org/")) as front_page:
    for line in front_page:
        print line
```

2.10 命名

module_name, package_name, ClassName, method_name, ExceptionName, function_name, GLOBAL_VAR_NAME, instance_var_name, function_parameter_name, local_var_name.

应该避免的名称

1. 单字符名称, 除了计数器和迭代器.
2. 包/模块名中的连字符 (-)
3. 双下划线开头并结尾的名称 (Python 保留, 例如 __init__)

命名约定

1. 所谓”内部 (Internal)”表示仅模块内可用, 或者, 在类内是保护或私有的.
2. 用单下划线 (_) 开头表示模块变量或函数是 protected 的 (使用 from module import * 时不会包含).
3. 用双下划线 (__) 开头的实例变量或方法表示类内私有.
4. 将相关的类和顶级函数放在同一个模块里. 不像 Java, 没必要限制一个类一个模块.
5. 对类名使用大写字母开头的单词 (如 CapWords, 即 Pascal 风格), 但是模块名应该用小写加下划线的方式 (如 lower_with_under.py). 尽管已经有很多现存的模块使用类似于 CapWords.py 这样的命名, 但现在已经不鼓励这样做, 因为如果模块名碰巧和类名一致, 这会让人困扰.

Python 之父 Guido 推荐的规范

Type	Public	Internal
Modules	lower_wit h_under	_lower_with_under
Packages	lower_wit h_under	
Classes	CapWords	_CapWords
Exceptions	CapWords	
Functions	lower_wit h_under()	_lower_with_under()
Global/Class Constants	CAPS_WITH _UNDER	_CAPS_WITH_UNDER
Global/Class Variables	lower_wit h_under	_lower_with_under
Instance Variables	lower_wit h_under	_lower_with_under (protected) or __lower_with_under (private)
Method Names	lower_wit h_under()	_lower_with_under() (protected) or __lower_with_under() (private)
Function/Method Parameters	lower_wit h_under	
Local Variables	lower_wit h_under	

2.11 Main 函数

即使是一个打算被用作脚本的文件, 也应该是可导入的. 并且简单的导入不应该导致这个脚本的主功能 (main functionality) 被执行, 这是一种副作用. 主功能应该放在一个 main() 函数中.

在 Python 中, pydoc 以及单元测试要求模块必须是可导入的.

你的代码应该在执行主程序前总是检查 `if __name__ == '__main__'`, 这样当模块被导入时主程序就不会被执行.

```
def main():
    ...
    pass

if __name__ == '__main__':
    main()
```

所有的顶级代码在模块导入时都会被执行. 要小心不要去调用函数, 创建对象, 或者执行那些不应该在使用 pydoc 时执行的操作.

2.12 最后个人一些小建议

1. 不要隐士导入包

```
from sqlalchemy import *
```

2. 命名尽量不要和内置的库，模块一样

比如下面的命名:

```
math.py    operators.py    heapq.py    copy.py
```

因为你的名字如果和这些内置的一样，有时候你想导入自己的模块，发现导入的是标准库的模块。当然这个 python 解释器如何查找库的顺序有关。

参考文档

上面的这些都来自下面的文档, 几乎没有做过改动。

Python PEP-8 编码风格指南中文版

PEP-8- Style Guide for Python Code

Python 风格指南

分享快乐, 留住感动. '2019-10-13 20:09:18' –frank

CHAPTER

THREE

发布 PYTHON 包到官方 PYPI 上面

[TOC]

举个例子现在要发布 useful_decroation 这个包

发布一般需要 setup.py , LICENSE.txt package ,README.rst 等

3.1 1 首先要写一个 setup.py

主要通过这个脚本来实现发布

```
# -*- coding: utf-8 -*-
"""
@User     : Frank
@File     : setup.py
@DateTime : 2019-09-16 11:24
@Email    : frank.chang@lexisnexis.com
"""
from setuptools import setup,find_packages
import io
import re

with io.open('README.rst', 'r', encoding='utf8') as f:
    long_description = f.read()

with io.open("src/useful_decoration/__init__.py", "rt", encoding="utf8") as f:
    version = re.search(r'__version__ = "(.*?)"', f.read()).group(1)


setup(
    name="useful_decoration",
    license='Apache License 2.0',
    version=version,
```

(continues on next page)

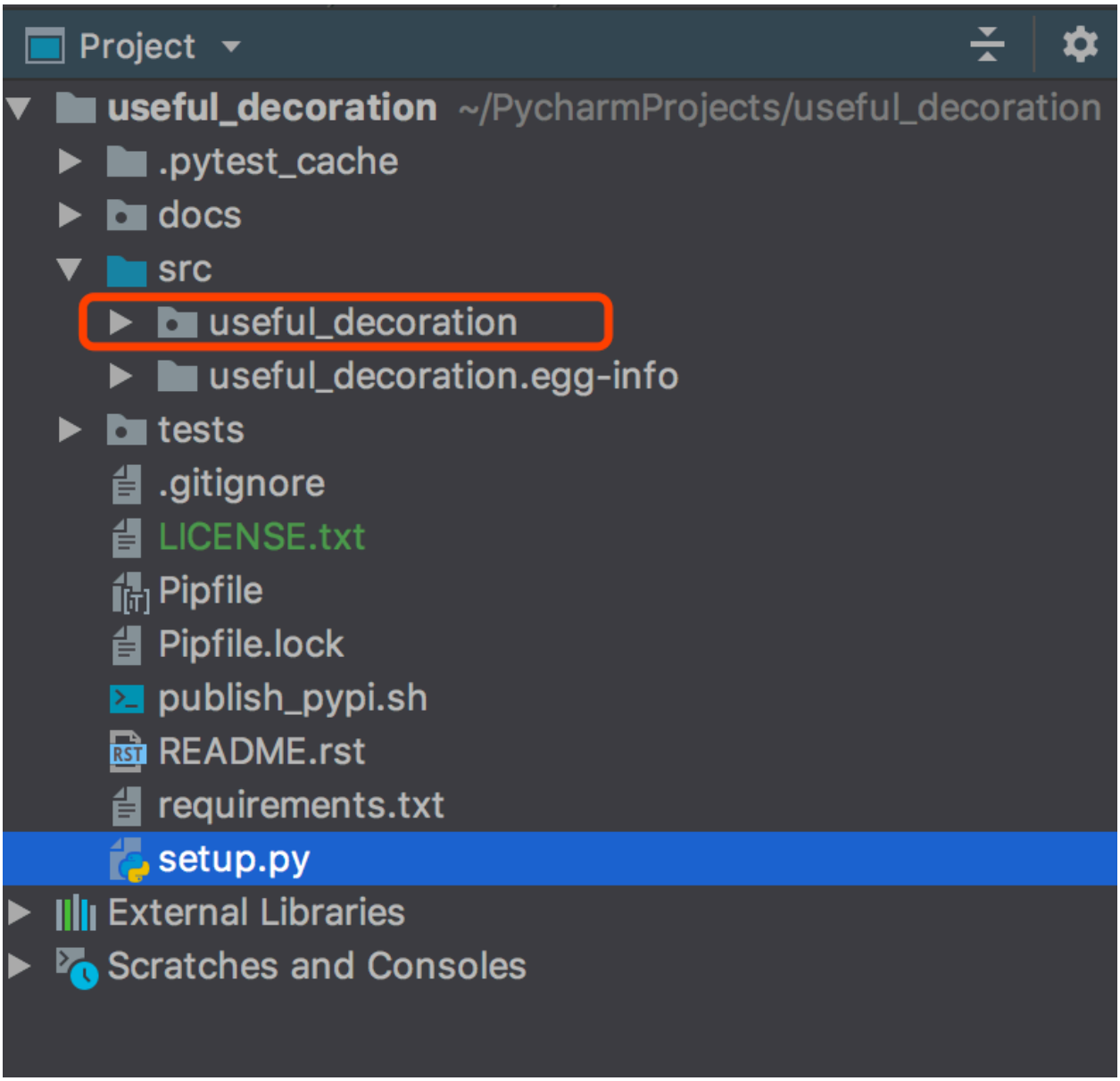

Project
useful_decoration ~/PycharmProjects/useful_decoration
.pytest_cache
docs
src
useful_decoration
useful_decoration.egg-info
tests
.gitignore
LICENSE.txt
Pipfile
Pipfile.lock
publish_pypi.sh
README.rst
requirements.txt
setup.py
External Libraries
Scratches and Consoles

Fig. 1: imag1

(continued from previous page)

```
    packages=find_packages("src"),
    zip_safe=False,
    include_package_data=True,
    package_dir={"": "src"},
    long_description=long_description,
    url='https://github.com/changyubiao/useful_decoration',
    author='frank',
    author_email='frank.chang@lexisnexis.com',
    description='powerful and useful decorations',

    project_urls={
        "Documentation": "https://useful-decoration.readthedocs.io/en/latest/",
        "Code": "https://github.com/changyubiao/useful_decoration",
    },

    python_requires='>=3.6',
    install_requires=[
        "loguru>=0.3.2",
    ],

)
```

有几个参数说一下: package 决定你要发哪个包,

name 就是报名

url 项目地址

version 可以在包里定义一个 __version__ 来制定 version

find_packages 这个是官方提供自动寻找包的一个方法, 他会寻找 package 下面的子包, 如果项目比较大, 用这个比较方便.

setup 参数官方文档

3.2 2 配置 pypirc 文件

在用户家目录.pypirc

用户名, 密码从 pypi 官网注册一个, 写到下面的配置文件里面.

register 注册用户名, 密码 https://pypi.org/account/register/

/c/Users/xxxxx/.pypirc

```
[distutils]
index-servers=pypi

[pypi]
repository = https://upload.pypi.org/legacy/
username: changyubiao
password: xxxxxxxxx



[pypitest]
repository: https://test.pypi.org/legacy/
username:changyubiao
password:xxxxxxxxx
```

3.3 3 尝试本地打包发布

如果不确定发布正不正确, 可以先发到 pypitest 上面进行测试, 没问题在像官方 pypi 里面发布包

```
# 用它来发布先安装 这个
pip install twine
# 打包用到的包 安装 相应依赖
pip install setuptools wheel

# 检查打包文件
python setup.py  check

# 打包
python setup.py sdist bdist_wheel


# 发布包
twine upload dist/*
```

如果执行 check 没有出现错误, 就可以正常打包了, 生成 sdist , bdist_wheel 这两种包.

执行命令会生成一个 dist build 两个目录其中 dist 就是你打包的内容, 有两种格式.tgz , 还有一个.whl

这里面放的就是项目打包后的文件了. 可以解压工具看下 tar.gz 里面是不是你的文件都压缩好了.

如果是就问题不大了.

下一步就是发布包,

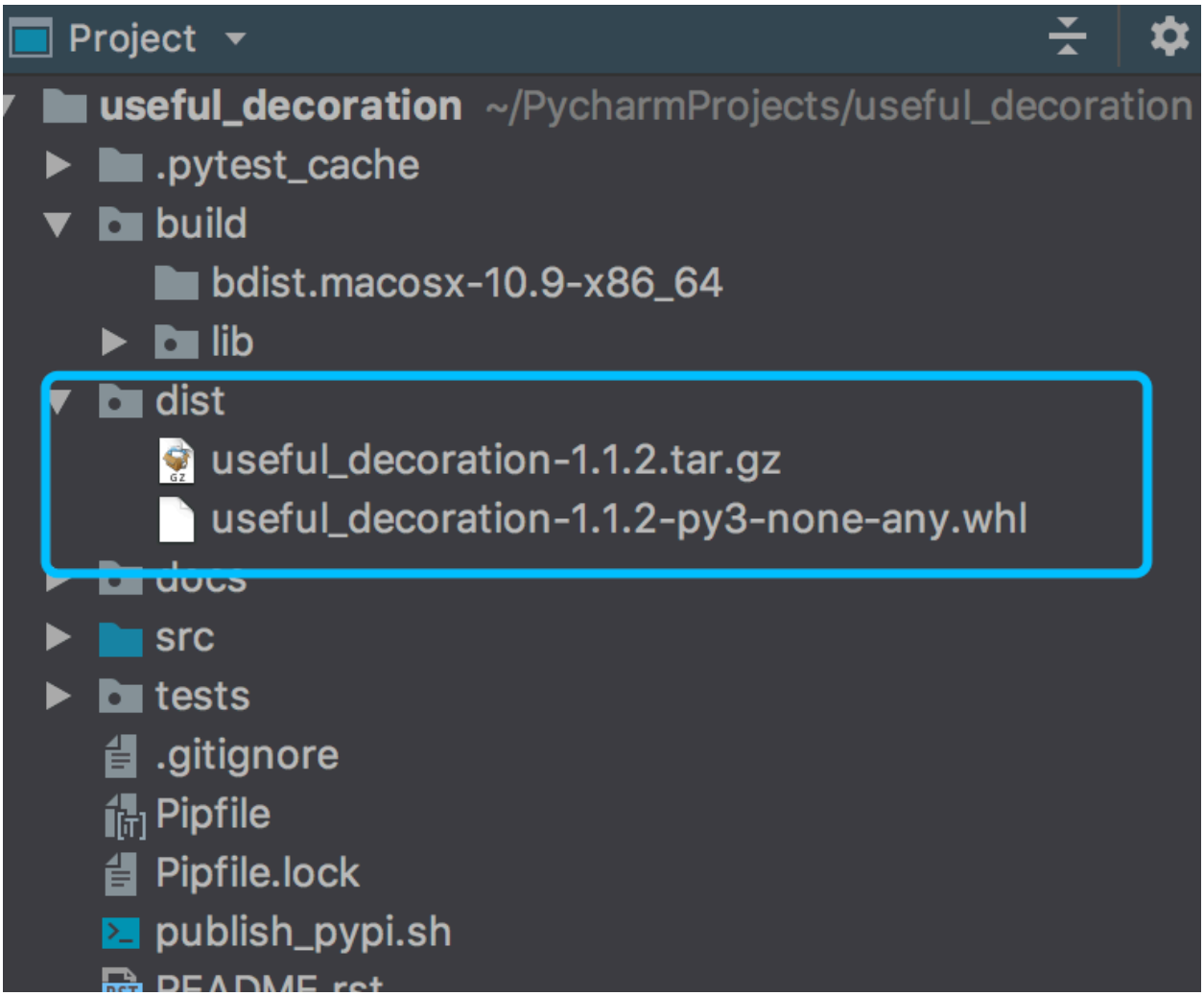

Project
useful_decoration ~/PycharmProjects/useful_decoration
.pytest_cache
build
bdist.macosx-10.9-x86_64
lib
dist
useful_decoration-1.1.2.tar.gz
useful_decoration-1.1.2-py3-none-any.whl
src
tests
.gitignore
Pipfile
Pipfile.lock
publish_pypi.sh

Fig. 2: imag1

```
twine  upload  dist/*
```

如果这里没有报错就说明已经发布上去了, 一切顺利.

进入官网搜一下, 发现就有了 useful-decoration https://pypi.org/project/useful-decoration/

3.4 4 可能遇到的障碍

有可能你的项目有一些数据文件, 不是 xx.py 结尾的默认是不会被打包的.

pypi demo 官方文档

1 Python Packaging User Guide https://packaging.python.org/tutorials/packaging-projects/#semantic-versioning-preferred

2 Packaging and distributing projects

https://packaging.python.org/guides/distributing-packages-using-setuptools/

3 打包数据文件

https://setuptools.readthedocs.io/en/latest/setuptools.html#including-data-files

https://setuptools.readthedocs.io/en/latest/setuptools.html#find-namespace-packages

4 mainfest 是什么可以用来打包吗? 用来控制打包文件

https://docs.python.org/2/distutils/sourcedist.html#manifest-template

打包数据文件, MANIFEST.in 可以使用这个文件来定制化, 需要打包哪些, 需要排除哪些文件等.

5 license.txt 配置, 开源协议

https://packaging.python.org/guides/distributing-packages-using-setuptools/#license-txt

3.5 5 参考文档

package-projects 官方文档

打包数据文件 stackoverflow

useful-decoration 项目地址

分享快乐, 留住感动. '2019-10-31 22:09:18' –frank

[TOC]

CHAPTER

FOUR

PYTHON3 中的上下文管理器

上下文管理器是什么

只要实现了 __exit__ , __enter__ 的类它就是上下文管理器. 它可以用来管理上下文

实现了 __exit__ , __enter__ 魔术方法的对象可以使用 with 语句

上下文表达式返回一个上下文管理器

4.1 上下文管理的作用和目的

上下文管理对象是为了存在的目的是管理 with 语句, 而 with 语句目的是为了简化 tyr/ finally 这种模式

这种模式保证运行一段代码后, , 即便代码里面发生错误, return 语句或者调用终止 sys.exit() , 也会执行特定的代码段, 来做一些最后的处理, 比如释放连接, 还原一些状态, 释放资源等.

4.2 介绍上下文管理器协议

首先要实现一个上下文管理需要实现两个魔术方法 __exit**, __enter** , 即需要在一个类中

不管以哪种方式退出 with 语句, 都会进行 __enter__ 方法里面的代码段, 而不是 __enter__ 返回对象的对象上调用

下面简单实现一个上下文管理器

Resource 类中实现了两个魔术方法, 同时实现了一个 query 方法

```
# -*- coding: utf-8 -*-

class Resource:

    def __enter__(self):
        print("connect to resource")
        return self
```

(continues on next page)

(continued from previous page)

```
    def __exit__(self, exc_type, exc_val, exc_tb):
        print("close resource  connection")

    def query(self):
        print("query  data ...")


with Resource() as r:
    r.query()
```

首先 Resource 实现了这两个魔术方法, 它就是一个上下文管理器, 就可以使用 with 这种语法,with 表达式后面一定要是一个上下文管理器, 才能够这样写.

结果如下:

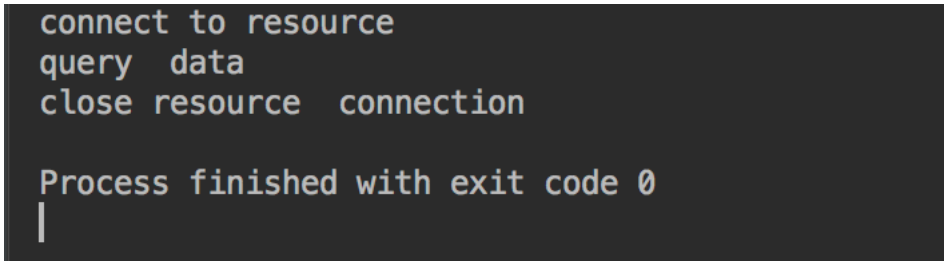


Fig. 1: img1

整个代码的执行流程

可以看出首先执行的 __enter__ 里面的代码段, 后来返回 self , 然后执行 query 方法, 最后执行 __exit__ 魔术方法.

这个就是最简单的上下文管理器了.

4.2.1 enter 方法介绍

你可能会疑惑为啥 __enter__ 方法要返回 self ?

这个例子中返回 self , 即返回 Resource 对象的实例, 然后通过 r.query 来调用实例方法 query , 所以这里是需要返回 self . 这里 __enter__ 方法的返回值,

```
with Resource() as r:pass
```

就是这段代码 as 后对应变量的值, 这里命名为 r 它的值就是 self

下面在 __enter__ 返回 frank , 之后打印 r 来看一下这个值是什么?

```python
# -*- coding: utf-8 -*-

class Resource:

    def __enter__(self):
        print("connect to resource")
        return "frank"

    def __exit__(self, exc_type, exc_val, exc_tb):
        print("close resource  connection")

    def query(self):
        print("query  data ...")


with Resource() as r:
    print(f"r:{r}")
```

结果如下:

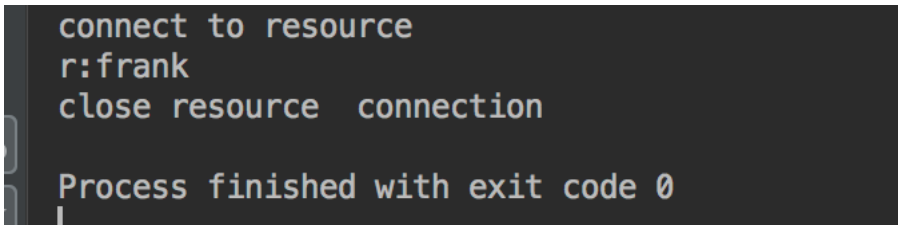


Fig. 2: img2

现在应该理解为啥要返回 self 了吧. 一般情况下 __enter__ 方法会返回 self , 当然也可以不返回.

如果实在不需要返回, 也可以不返回.

只要明白只要执行 with 这种里面的代码段首先是先执行 __enter__ 方法里面的代码即可.

4.2.2 exit 方法介绍

__exit__ 方法实在 with 里面代码段执行完后, 执行的方法, 一般就是资源清理的代码, 会写在这里, 还有一些异常处理的代码, 也可以写在这里.

注意到上面方法有三个参数, 这三个参数只有 with 语句报错后, 这几个参数才会有值, 如果 with 代码段里面没有报错那么这个三个值均为 None

exc_type 异常类

exc_value 异常值

exc_tb traceback 对象

```
# -*- coding: utf-8 -*-

class Resource:

    def __enter__(self):
        print("connect to resource")
        return "frank"
        pass

    def __exit__(self, exc_type, exc_val, exc_tb):
        """

        :param exc_type: 异常类
        :param exc_val: 异常值
        :param exc_tb:  traceback 对象
        :return:
        """
        print(exc_type,exc_val,exc_tb)
        print("close resource  connection")

    def query(self):
        print("query  data ...")


with Resource() as r:

    1/0
    print(f"r:{r}")
```

在 with 语句故意抛出一个异常可以看出这三个值.

可以看出程序就报错了, 并且异常被抛出来了.

刚刚在 __exit__ 方法里面其实是可以处理这种异常的, 保证程序可以正常执行.

可以通过 exc_tb 是否为空来处理这个异常, 然后注意这个时候要返回一个 True , 这里的意思是程序异常已经处理, 不继续抛出到主程序了.

```
# -*- coding: utf-8 -*-

class Resource:

    def __enter__(self):
        print("connect to resource")
        return "frank"
        pass
```

(continues on next page)

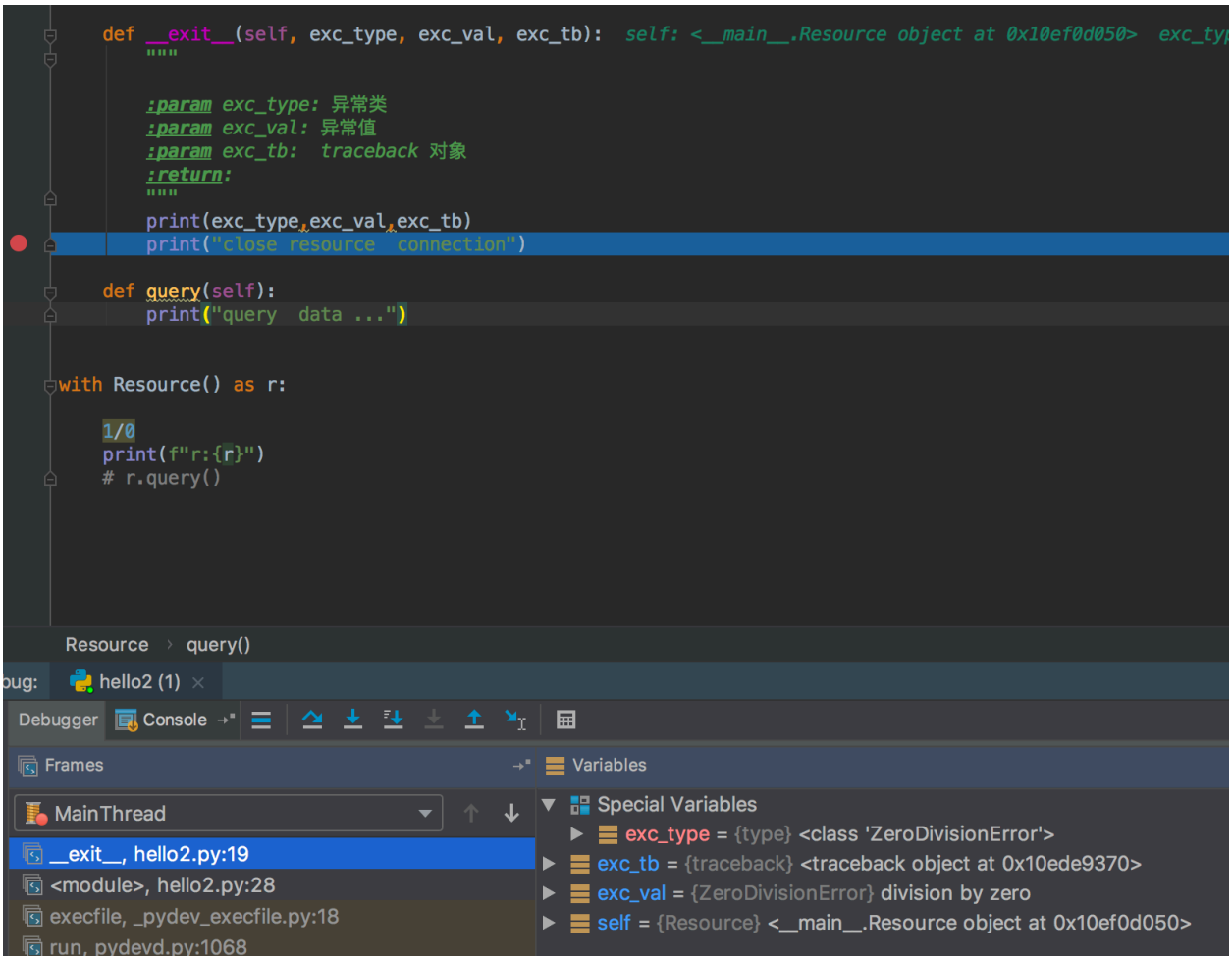


Fig. 3: img3

```
connect to resource
<class 'ZeroDivisionError'> division by zero <traceback object at 0x10ede9370>
close resource  connection
Traceback (most recent call last):
  File "/Applications/PyCharm.app/Contents/helpers/pydev/pydevd.py", line 1664, in <module>
    main()
  File "/Applications/PyCharm.app/Contents/helpers/pydev/pydevd.py", line 1658, in main
    globals = debugger.run(setup['file'], None, None, is_module)
  File "/Applications/PyCharm.app/Contents/helpers/pydev/pydevd.py", line 1068, in run
    pydev_imports.execfile(file, globals, locals)  # execute the script
  File "/Applications/PyCharm.app/Contents/helpers/pydev/_pydev_imps/_pydev_execfile.py", line 18, in execfile
    exec(compile(contents+"\n", file, 'exec'), glob, loc)
  File "/Users/frank/PycharmProjects/new-feature/first_package/hello2.py", line 27, in <module>
    1/0
ZeroDivisionError: division by zero

Process finished with exit code 1
```

Fig. 4: img4

(continued from previous page)

```
    def __exit__(self, exc_type, exc_val, exc_tb):
        """

        :param exc_type: 异常类
        :param exc_val: 异常值
        :param exc_tb:  traceback 对象
        :return: True or False
        """
        if exc_tb:
            print("catch exception .  deal exception")
            return True
        print("close resource  connection")

    def query(self):
        print("query  data ...")


with Resource() as r:

    1/0
    print(f"r:{r}")
```

执行结果如下:

```
connect to resource
catch exception .  deal exception

Process finished with exit code 0
```

Fig. 5: img5

这里 __enter__ 方法返回值, 决定是否要将异常抛出来

__exit__ 这个方法用来上下文管理器退出执行的方法, 如果有异常, 可以在这里处理, 并且返回 True, 则异常就不会被抛出来, 如果返回 false 异常就会被抛出来. 有主程序处理该异常.

4.3 上下文管理用法

1 常用的示例

- 比如数据连接以及关闭的操作

一般连接数据库需要以下步骤

数据库连接, 管理

```
1 连接数据库

2 exectue  sql

3 释放连接 con.close()
```

```
try:
  pass
except:
  pass
finally:
  pass
```

比如可以像下面的的例子

来实现一个上下文管理器

```
# -*- coding: utf-8 -*-

import pymysql


class ConnText:

    def __enter__(self):
        print("begin db connection")
        self.conn = self.get_connection()
        return self.conn

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.close()

    def get_connection(self):
        conn = pymysql.connect()
        return conn

    def close(self):
        self.conn.close()
        print("close db connection")

    def query(self):
        print("query data")
```

(continues on next page)

(continued from previous page)

```
with ConnText() as r:
    r.query()
```

- 文件读写也可以用上下文管理器

  一般打开文件后最后都需要关闭文件句柄, 这个时候就可以使用上下文管理了.

```
with open('/tmp/1.sql') as f
  pass
```

4.4 框架里面使用

其实在很多源码中也可以看到上下文管理器的用法,

比如 celery 核心对象 Celery 也实现了上下文管理器

代码位于:

```
python3.x/site-packages/celery/app/base.py
```

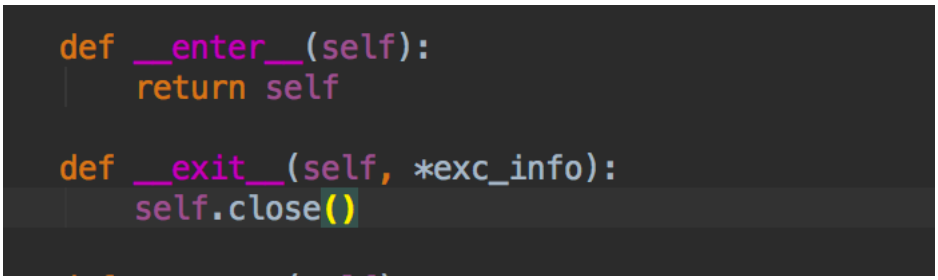


Fig. 6: img7

在 flask 框架里面用了很多上下文管理, 比如这个模块

/python3.x/site-packages/flask/ctx.py

`RequestContext` , `AppContext` 这两个类都实现了上下文的管理器的协议

4.5 总结

本文简单总结了上下文管理器的语法, 使用 python 这种语法可以写代码看起来更优雅一些, 更加的 pythonic. 有时候要学会看源代码来学习.

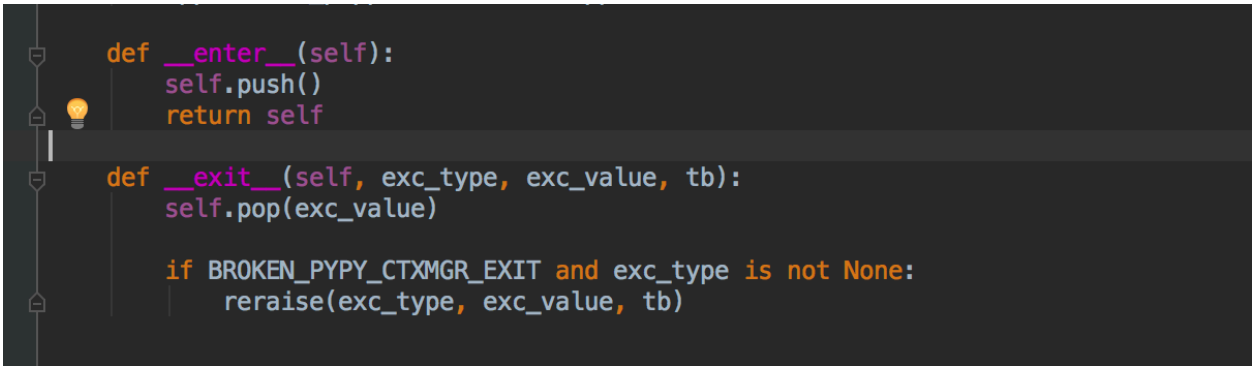

def __enter__(self):
self.push()
return self
def __exit__(self, exc_type, exc_value, tb):
self.pop(exc_value)
if BROKEN_PYPY_CTXMGR_EXIT and exc_type is not None:
reraise(exc_type, exc_value, tb)

Fig. 7: img7

CHAPTER

FIVE

SQLALCHEMY 中 COLUMN 的默认值属性

我们知道使用 sqlalchemy 定义 ORM 对象，需要给一些字段设置一个默认值，default 属性

类似下面的代码.

```
class Person(Base):
  __tablename__ = 'Person'

  id = Column(Integer, autoincrement=True, primary_key=True)
  name = Column(String(length=64), comment='姓名')
  is_delete = Column(Integer,comment="是否删除",default=0)

  def __repr__(self):
      return "<Person(id='%s', name='%s', mobile='%s')>" % \
             (self.id,
              self.name, self.mobile, )
```

这样就可以在 session.add() , session.commit() 的时候，如果没有提供这个字段的值，就会自动设置会 0 写入到数据库里面。

我用这个类创建表的时候发现，其实 sqlalchemy 并没有进行设置，表结构里面的默认值。

通过上面的日志，我可以清晰的发现，实际上 engine 来执行 sql 是上面的建表语句，并没有将 is_deleted 设置成默认值。

后来发现其实 Column 还有一个属性，叫 `server_default` 这个值才是真正可以生成表结构的时候，会设置默认值。

但是我设置 server_default 值的时候

```
class Person(Base):
    __tablename__ = 'Person'

    id = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(String(length=64), comment='姓名')
```

(continues on next page)

```
2020-12-06 10:38:41,860 INFO sqlalchemy.engine.base.Engine ROLLBACK
2020-12-06 10:38:41,862 INFO sqlalchemy.engine.base.Engine
CREATE TABLE `Person` (
    id INTEGER NOT NULL AUTO_INCREMENT,
    name VARCHAR(64) COMMENT '姓名',
    is_delete INTEGER COMMENT '是否删除',
    PRIMARY KEY (id)
)


2020-12-06 10:38:41,862 INFO sqlalchemy.engine.base.Engine {}
2020-12-06 10:38:41,897 INFO sqlalchemy.engine.base.Engine COMMIT
```

Fig. 1: 20201206104137582.png

(continued from previous page)

```
    # 这里设置 server_default 值
    is_deleted = Column(Integer,comment="是否删除",default=0,server_default=0)
    def __repr__(self):
        return "<Person(id='%s', name='%s', mobile='%s')>" % \
               (self.id,
                self.name, self.mobile, )
```

之后生成表结构的时候，发现出现了一个错误如下：

Argument ‘arg’ is expected to be one of type ‘<class ’ str’ >’

```
sqlalchemy.exc.ArgumentError: Argument 'arg' is expected to be one of type '<class
↪'str'>' or '<class 'sqlalchemy.sql.elements.ClauseElement'>' or '<class 'sqlalchemy.
↪sql.elements.TextClause'>', got '<class 'int'>'
```

```
Traceback (most recent call last):
  File "/Users/frank/code/python-study/mysqlalchemy/model.py", line 39, in <module>
    class Person(Base):
  File "/Users/frank/code/python-study/mysqlalchemy/model.py", line 44, in Person
    is_delete = Column(Integer,comment="是否删除",default=0,server_default=0)
  File "/Users/frank/code/python-study/venv/lib/python3.7/site-packages/sqlalchemy/sql/schema.py", line 1449, in __init__
    args.append(DefaultClause(self.server_default))
  File "/Users/frank/code/python-study/venv/lib/python3.7/site-packages/sqlalchemy/sql/schema.py", line 2800, in __init__
    arg, (util.string_types[0], ClauseElement, TextClause), "arg"
  File "/Users/frank/code/python-study/venv/lib/python3.7/site-packages/sqlalchemy/util/langhelpers.py", line 1122, in assert_arg_type
    % (name, " or ".join("'%s'" % a for a in argtype), type(arg))
sqlalchemy.exc.ArgumentError: Argument 'arg' is expected to be one of type '<class 'str'>' or '<class 'sqlalchemy.sql.elements.ClauseElement'>' or '<class 'sqlalc

Process finished with exit code 1
```

Fig. 2: image-20201206104911808

这里很明显说明参数错误，我陷入了沉思？为啥说我参数不对呢？

源码 sqlalchemy.sql.schema.py 查看 server_default 要求传入一个字符串类型的变量。

```
:param server_default: A :class:`.FetchedValue` instance, str, Unicode
    or :func:`~sqlalchemy.sql.expression.text` construct representing
    the DDL DEFAULT value for the column.

    String types will be emitted as-is, surrounded by single quotes::

        Column('x', Text, server_default="val")

        x TEXT DEFAULT 'val'

    A :func:`~sqlalchemy.sql.expression.text` expression will be
    rendered as-is, without quotes::

        Column('y', DateTime, server_default=text('NOW()'))

        y DATETIME DEFAULT NOW()

    Strings and text() will be converted into a
    :class:`.DefaultClause` object upon initialization.

    Use :class:`.FetchedValue` to indicate that an already-existing
    column will generate a default value on the database side which
    will be available to SQLAlchemy for post-fetch after inserts. This
    construct does not specify any DDL and the implementation is left
    to the database, such as via a trigger.

    .. seealso::
```

Fig. 3: image-20201206111853684

修改 orm 类

```
from sqlalchemy import Column, Integer, String, text


class Person(Base):
    __tablename__ = 'Person'

    id = Column(Integer, autoincrement=True, primary_key=True)
```

(continues on next page)

(continued from previous page)

```
    name = Column(String(length=64), comment='姓名')
    # 注意这里 只设置 server_default
    is_deleted = Column(Integer, comment="是否删除", server_default=text('0'))

    def __repr__(self):
        return "<Person(id='%s', name='%s')>" % \
               (self.id,
                self.name)
```

执行生成 table 语句，发现可以正常生成表结构了，并且 default 值也默认设置好了。

```
2020-12-06 11:33:27,531 INFO sqlalchemy.engine.base.Engine DESCRIBE `Person`
2020-12-06 11:33:27,531 INFO sqlalchemy.engine.base.Engine {}
2020-12-06 11:33:27,531 INFO sqlalchemy.engine.base.Engine ROLLBACK
2020-12-06 11:33:27,533 INFO sqlalchemy.engine.base.Engine
CREATE TABLE `Person` (
    id INTEGER NOT NULL AUTO_INCREMENT,
    name VARCHAR(64) COMMENT '姓名',
    is_deleted INTEGER DEFAULT 0 COMMENT '是否删除',
    PRIMARY KEY (id)
)


2020-12-06 11:33:27,533 INFO sqlalchemy.engine.base.Engine {}
2020-12-06 11:33:27,542 INFO sqlalchemy.engine.base.Engine COMMIT
```

Fig. 4: image-20201206113614276

好了，一切看起来完美了。

所以如果要想定义 orm，生成表结构的时候，就自动生成默认值，一定要使用 `server_default` 这个字段，并且要求这个字段为字符串类型，可以使用 `text` 去装饰一下。

5.1 server_default vs. default 的区别

在 sqlalchemy 中定义 Column 字段可以有两个 default 相关的字段，一个是 `default` 另一个是 `server_default`，他们之间的区别呢?

查看源码位置 sqlalchemy.sql.schema.py `Column` 这个类

`default` 这个属性，就是默认生成 orm 对象，如果某个字段没有传值，就使用 default 值，然后写入到数据库中。

server_default 这个属性，要求是一个 str, unicode 类型。用来生成表结构的时候，需要指定字段默认值的时候来指定的。

5.1.1 看一个小例子

下面以一个例子作为演示，下面我创建一个 User 的 model 类，然后有一个字段 password 我设置一个 default 的属性，然后创建一个表。

```
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy import Column, Integer, String, create_engine
from sqlalchemy.orm import sessionmaker

Base = declarative_base()

# 创建连接对象，并使用 pymsql 引擎
conn_str = "mysql+pymysql://{user}:{pwd}@{host}:3306/{db_name}?charset=utf8mb4"
connect_info = conn_str.format(user='root',

                               pwd='123456',
                               host='127.0.0.1',
                               db_name='db1')

engine = create_engine(connect_info, max_overflow=5)

session_factory = sessionmaker()
session_factory.configure(bind=engine)

session = session_factory()


class User(Base):
    __tablename__ = 'User'

    id = Column(Integer, autoincrement=True, primary_key=True)
    name = Column(String(length=64), comment='姓名')
    mobile = Column(String(length=64), comment='手机号')
    password = Column(String(length=64), comment='密码', default='0000')

    def __repr__(self):
        return "<User(id='%s', name='%s', mobile='%s', password='%s')>" % \
               (self.id,
                self.name, self.mobile, self.password)
```

(continues on next page)

(continued from previous page)

```
def create_table():
    # 创建表结构
    Base.metadata.create_all(engine)


if __name__ == '__main__':

    create_table()
    print("create table successfully ")
```

创建完成后，我们到数据库查看表结构，发现并没有给 password 一个默认值。

建表语句如下：

```
CREATE TABLE `User` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `name` varchar(64) DEFAULT NULL COMMENT '姓名',
  `mobile` varchar(64) DEFAULT NULL COMMENT '手机号',
  `password` varchar(64) DEFAULT NULL COMMENT '密码',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=latin1;
```

User 表结构中并没有给 password 生成一个密码的 default 值。

下面我们使用 sqlalchemy 插入一个 user

```
if __name__ == '__main__':
    u = User(name='frank',mobile='123xxxx3456')
    session.add(u)
    session.commit()
    session.close()
```

数据库查看没有任何问题，已经自动把 password 字段填充成 0000 了。

这是在执行 sql 的时候，当 ORM 对象没有给某个字段赋值的时候，sqlalchemy 会查看 Column 属性的 default 是否有值，如果有值，则使用当前值; 如果没有值，则会默认为 default 值。

然后在进行执行 sql , 所以就自动加上了默认值。

因此想要在表结构生成的时候就设置默认值，要使用 `server_default` 这个属性，另外 server_default 的值必须是字符串。

```
# 正确的设置方式是
is_deleted = Column(Integer, default=0, server_default=text('0'))
```

如果没有写 server_default 参数，那么在代码中新建对象往数据库插入的时候是有一个值的，但是在数据库里查看表结构，会发现表上并没有给字段设置默认值。

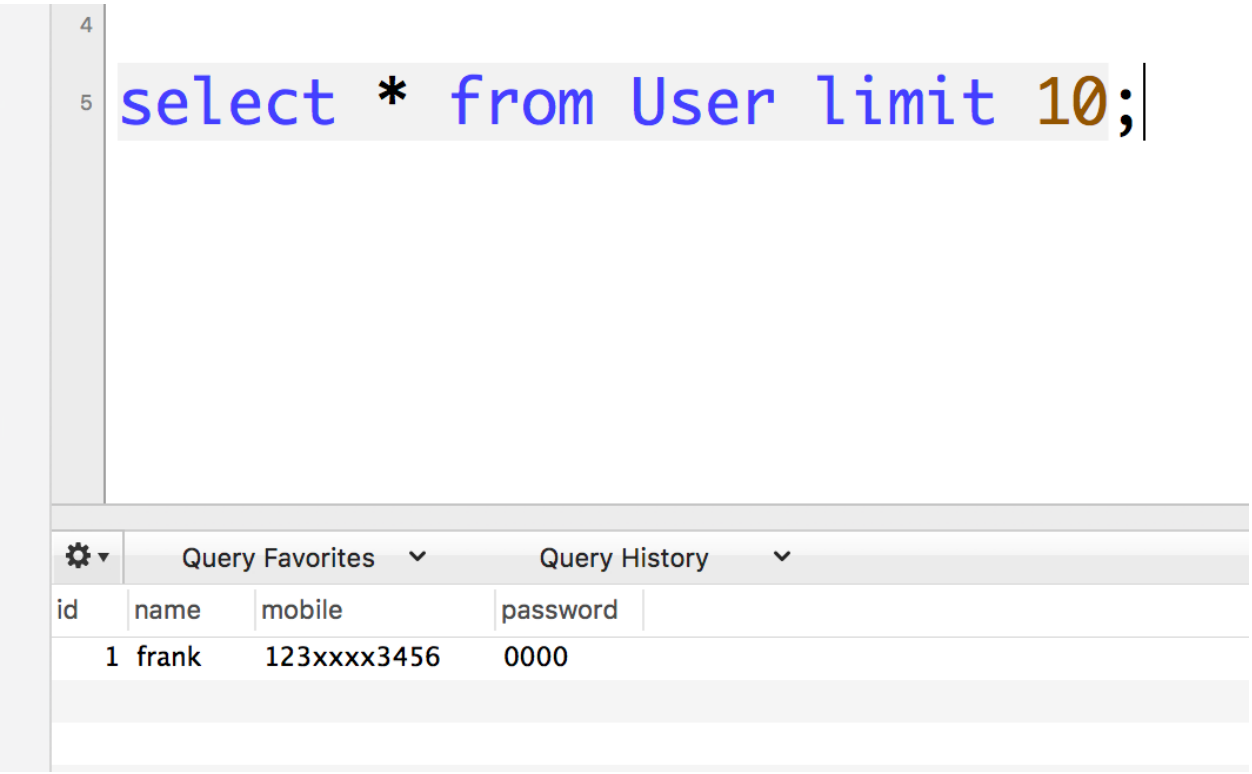


Fig. 5: image-20201205151915773

另外 server_default 的值必须是字符串。

5.2 设置表的默认创建时间和更新时间

有的时候我们希望在表创建的时候，有创建时间和更新时间。所以我们就可以使用 `server_default` 这个属性来生成就好了。

```
from sqlalchemy import TIMESTAMP, Boolean, Column, Float
from sqlalchemy.ext.declarative import declarative_base

base = declarative_base()


class Base(base):
    __abstract__ = True
    __table_args__ = {
        'mysql_engine': 'InnoDB',
        'mysql_charset': 'utf8',
        'extend_existing': True
    }
```

(continues on next page)

(continued from previous page)

```
    id = Column(INT, primary_key=True, autoincrement=True)

    create_time = Column(TIMESTAMP, default=None, nullable=True,
                         server_default=text('CURRENT_TIMESTAMP'))
    update_time = Column(TIMESTAMP, default=None, nullable=True,
                         server_default=text(
                             'CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP'))
```

5.3 参考文档

stackoverflow discussion

官方文档

分享快乐, 留住感动. ‘2020-12-07 21:08:08’ –frank

CHAPTER

SIX

程序员的自我修养-算法递归

[TOC]

6.1 1 递归概念引入

首先, 思考一个小小的问题，计算机是如何解决问题的? 在计算机的世界里面只有简单的

if else , for loop , while ,do while , recursion 等这些简单的指令集，计算机是一个没有感情的机器，计算机更加喜欢做的事情是什么呢?

就是 **重复性的问题**，计算机特别擅长。

所以递归（recursion ）就诞生了，所以递归是什么呢？递归就是自己调用自己的一个过程，是一种编程的技巧。

6.1.1 1.1 来举一个简单的例子

比如现在我要计算 1 + 2 + ···100 , 举一个不是特别恰当的例子，假设我现在有一个函数可以从 50 到 100 计算和，那么我只需要计算 1 到 50 的和，然后调用另外一个函数计算结果，然后把结果加起来。

```
def sum_to_hundred():
    """
    计算 1 + 2 + ... 100
    :return:
    """
    r = 0
    for i in range(1, 50):
        r += i

    r2 = sum_50_100()
    return r + r2
```

(continues on next page)

(continued from previous page)

```
def sum_50_100():
    """
    计算 50 到 100 的和
    :return:
    """
    sum = 0
    for i in range(50, 101):
        sum += i
    return sum

if __name__ == '__main__':

    r = sum_to_hundred()
    print(r)
```

例子应该比较简单，但是我们应该思考一个问题，1 + 2 + 3 …+98+ 99+ 100 这个问题会有一些自相似性。

换个角度想一想，如果我要计算 1 + 2 + 3 …+ 98+ 99 + 100 的和，

如果是可以知道假设 1 + 2 + 3 …+98+ 99 = X ，上面的式子 X + 100

如果是可以知道假设 1 + 2 + 3 …+98 = Y ，上面的式子 Y + 99 + 100

如果是可以知道假设 1 + 2 + 3 …+97 = Z ，上面的式子 Z + 98 + 99 + 100

….

那么我们知道什么，很简单啊，

1 = XXX 等于 1 ，这里就是找了最初的结果。所以我们尝试用递归的方式来实现一下这个函数。

```
def recur(n=100):
        # base case
    if n == 1:
        return 1
    return recur(n - 1) + n
```

这里我定义 recur(n) 表示的从 [1,n] 的和.

recur(5)

recur(4) + 5

(recur(3) + 4) + 5

((recur(2) + 3)+ 4) + 5

(((recur(1) + 2) + 3) + 4) + 5

因为 recur(1) 的结果我们很清楚啊，就是 1 ，所以计算机就把 1 +2 +⋯+ 5 的结果计算出来了。

6.1.2 1.2 递归的一些必要条件

思考一下递归需要哪些必要的条件

- base case 就是基线的条件
- 递归条件

递归的基线条件就是何时结束递归函数进行返回上一个例子中，就是 n==1 ，这个条件就是基线条件

第二点递归的递归条件，递归条件是如何把递归往递归条件上改变的条件，随着递归条件的不断变化，最终可以到达基线条件。上一个求和的例子中 `recur(n) = recur(n-1) + n` 这个就是递归条件。

我们按照算法时间复杂度的角度重新思考这个问题，n 相当于问题规模，递归条件每次把问题规模减低到一个，最后降到我们直接就可以看出结果的条件 (base case).

好，现在练手一下，如果需要算法 `n!` 请使用递归的方式写出来

首先思考：基线条件是什么？第二递归条件是什么？

```
def factorial(n: int) -> int:
    if n == 1:
        return 1
    return factorial(n - 1) * n
```

6.2 2 如何使用递归

刚刚举例子，是比较简单的，这个时候你可能并没有感觉到递归带来的好处是什么，我举几个稍微复杂的例子，

大家一起思考。

6.2.1 爬楼梯问题

假设小明正在爬楼梯。需要 n 阶你才能到达楼顶。

每次你可以爬 1 或 2 个台阶。你有多少种不同的方法可以爬到楼顶呢?

注意：给定 n 是一个正整数。

示例 1:

示例 2:

假设小明现在在楼梯的开始上楼，

思考一下:

Fig. 1: image-20201108113754855

如果小明可以跳到 n 阶台阶，则只有两种方案可以跳上来。

假设 f(n) 表示跳到 n 阶台阶的方法数，那么

f(n) = f(n-1) + f(n-2)

想一想 base case

f(1) = 1

f(2) = 2

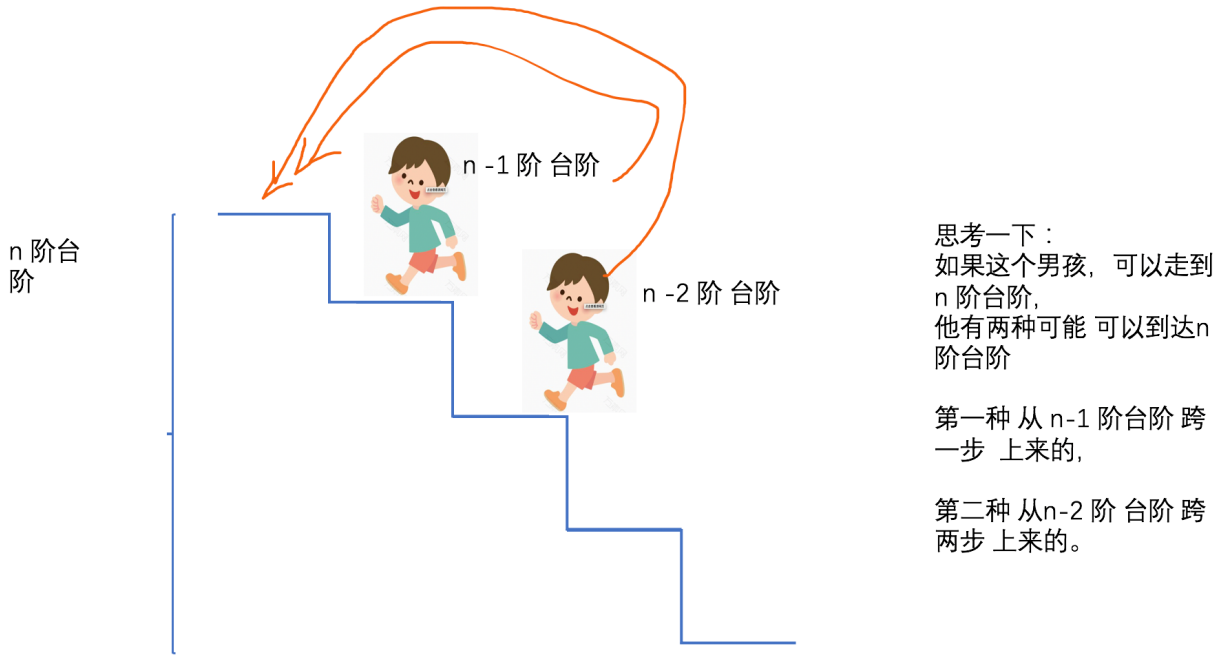


Fig. 2: image-20201108145637215

那么比较容易些出来以下的代码。

```
class Solution:

    def climbStairs(self, n: int) -> int:
        if n <= 2:
            return n
        else:
            return self.climbStairs(n-1) + self.climbStairs(n-2)
```

思考一下这个代码有没有什么问题呢?

画一下递归的状态树，这里大概画了一下，

我们从一下可以看出递归的状态有大量的重复计算的问题。如何解决这个问题呢?

其中红色部分，绿色部分都进行了重复的计算，所以可以把计算的结果先保存起来，如果发现这个值已经计算过了，直接使用之前计算过的值即可。

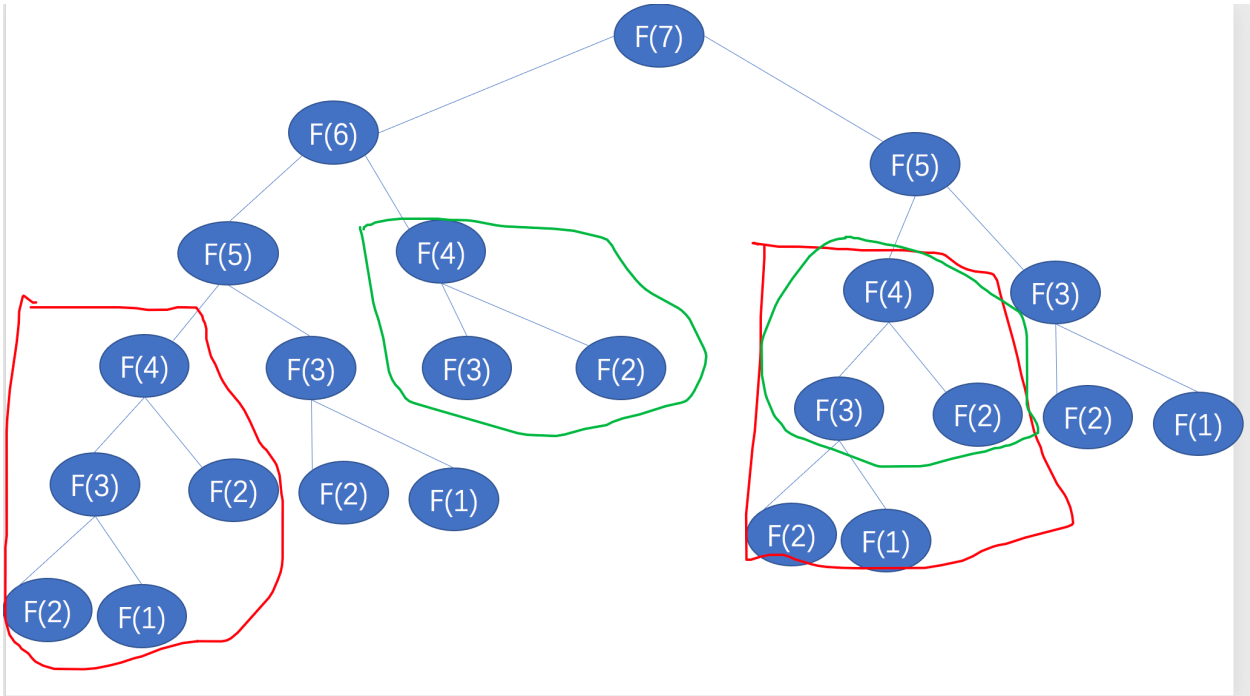


Fig. 3: image-20201108192205304

优化后的代码：

把重复计算的结果保存起来

```
class Solution:

    memo = dict()

    def climbStairs(self, n: int) -> int:
        if n in self.__class__.memo:
            return self.__class__.memo[n]

        if n <= 2:
            self.__class__.memo[n] = n
        else:
            r = self.climbStairs(n - 1) + self.climbStairs(n - 2)
            self.__class__.memo[n] = r
        return self.__class__.memo[n]
```

使用内置的 lru_cache 进行缓存就行，不用自己手写 lru_cache

```
from functools import lru_cache

class Solution:

    @lru_cache(maxsize=128)
    def climbStairs(self, n: int) -> int:
        if n <= 2:
            return n
        else:
            return self.climbStairs(n-1) + self.climbStairs(n-2)
```

更好的解法 dp 法

```
class Solution:

    def climbStairs(self, n: int) -> int:

        if n <= 2:
            return n
        dp = [0] * (n + 1)
        dp[0] = 0
        dp[1] = 1
        dp[2] = 2

        for i in range(3, n + 1):
            dp[i] = dp[i - 1] + dp[i - 2]
        return dp[n]
```

6.2.2 括号生成问题

数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且有效的括号组合。

示例：

有效括号的含义是指是一个成对出现的括号，是一个合法的括号的表达方式，就如上面的例子。

假设 n = 3 这种情况，

先假设没有要求括号合法性的要求

```
from typing import List

class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
```

(continues on next page)

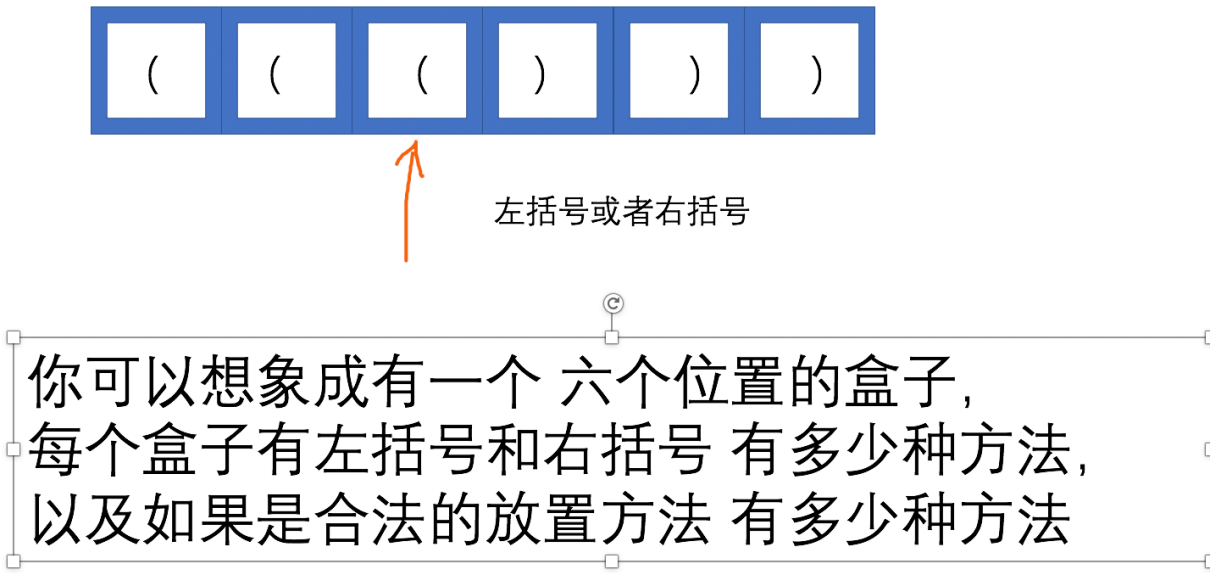


Fig. 4: image-20201110222111289

(continued from previous page)

```
        max_level = 2*n
        level = 0
        cur_result = ""
        self._generate(max_level=max_level,level=level,cur_result=cur_result)


    def _generate(self,max_level,level, cur_result):
        #  terminator
        if level == max_level:
            print(cur_result)
            # notice
            return

        # current logic ,and drill down next  level
        self._generate(max_level,level+1,cur_result+"(")
        self._generate(max_level,level+1,cur_result+")")

        # reverse  current level status
        pass
```

加上如何检查括号的合法性的逻辑

其实递归的过程中，我们可以检查一些不合法的括号，直接停止递归就好了。

对于左括号，如果括号没有用完，就可以直接添加。

对于右括号，要保证左括号的了数量 > 右括号的数量，就可以继续添加了。

```
from typing import List

class Solution:
    def generateParenthesis(self, n: int) -> List[str]:

        max_level = n
        cur_result = ""
        left,right = 0,0
        self.result = []
        self._generate(max_level=max_level, left=left,right=right,cur_result=cur_
↪result)
        return self.result


    def _generate(self,max_level,left,right, cur_result):
        #  terminator
        if left == max_level and right == max_level:
            # print(cur_result)
            self.result.append(cur_result)
            # notice
            return

        # current logic  ,and drill down
        if left < max_level:
            self._generate(max_level,left+1,right,cur_result+"(")
        if left > right:
            self._generate(max_level, left, right + 1, cur_result + ")")

        # reverse  current level status
        pass
```

这里有一个模板尝试找一些题目进行练习。

```
# python
def recursion(level, param1, param2, ...):
    # recursion terminator
    if level > MAX_LEVEL:
       process_result
       return
    # process logic in current level
    process(level, data...)
```

(continues on next page)

(continued from previous page)

```
    # drill down
    self.recursion(level + 1, p1, ...)
    # reverse the current level status if needed
    # pass
```

6.2.3 组合问题

给定两个整数 n 和 k，返回 1 …n 中所有可能的 k 个数的组合。

示例:

Related Topics

回溯算法

思考一下如何求解:

其实只要吧转态树画出来理解一下，就相对简单一点。当从取出一个数后，之后就不能取相同的数字，

所以怎么控制取不到相同的数字呢?

想一想递归的基线条件是什么?

是不是递归的深度 level 等于 k 的时候，

每次递归的下一层的时候，要从没有取到数开始取，不能取到之前的数字，所以需要在递归的时候给一个参数代表当前层的开始的位置在哪里呢? 这里我命名为 start 代表下一层开始的数字。

```
from typing import List


class Solution:
    def combine(self, n: int, k: int) -> List[List[int]]:
        level = 0
        start = 1
        cur_result = []
        self.result = []
        self._generate(n, k, level=level, start=start, cur_result=cur_result)
        return self.result

    def _generate(self, n, k, level, start, cur_result:List):
        #  terminator
        if level == k:
            self.result.append(cur_result.copy())
            return

        #  currrent logic process  and drill down
```

(continues on next page)

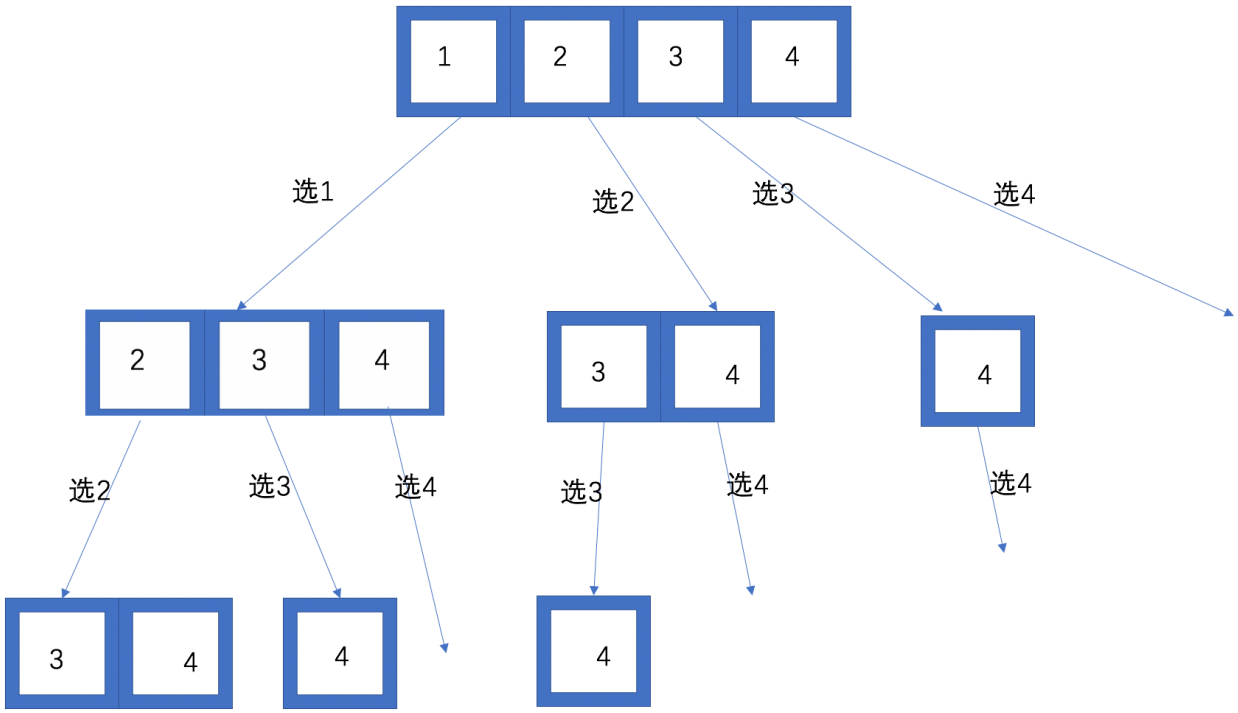


Fig. 5: image-20201110232026262

(continued from previous page)

```
        for i in range(start, n + 1):
            cur_result.append(i)

            self._generate(n, k, level + 1, start=i + 1, cur_result=cur_result)

            # reverse current level states
            cur_result.pop(-1)


if __name__ == '__main__':
    r = Solution().combine(n=4, k=2)

    print(r)
```

6.2.4 路径计数问题

一个机器人位于一个 m x n 网格的左上角（起始点在下图中标记为“Start”）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。

问总共有多少条不同的路径?

例如，上图是一个 7 x 3 的网格。有多少可能的路径?

示例 1:

示例 2:

提示:

1 <= m, n <= 100

题目数据保证答案小于等于 2 * 10 ^ 9

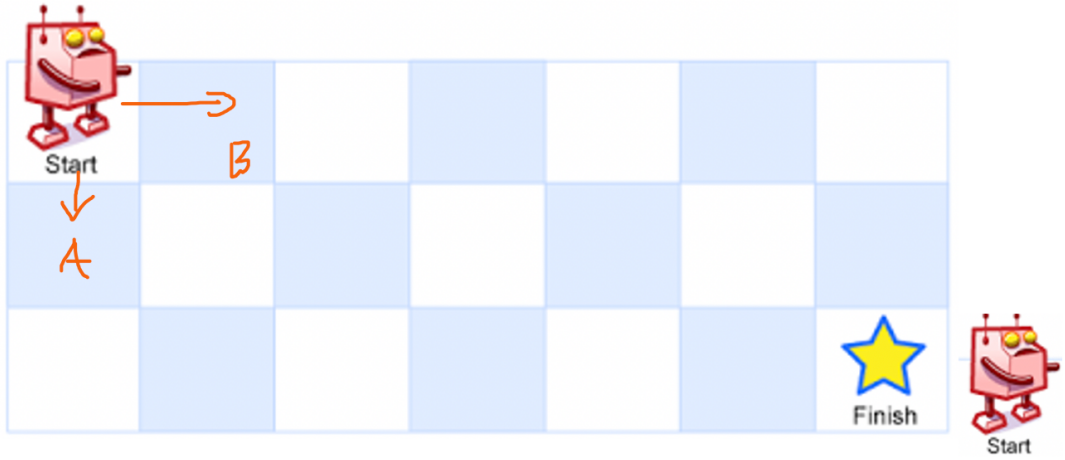
Fig. 6: image-20201108181316963

由于机器人只能向右或者向下走。这样就走到了重复子问题了。

递归的写法

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        i, j = 0, 0
        r = self.count_path(m, n, i, j)
        return r

    def count_path(self, row, col, i, j):
        # terminator
        if i >= row or j >= col:
            return 0
        if i == row-1 and j == col-1:
            # find a  result
            return 1
        return self.count_path(row, col, i + 1, j) + self.count_path(row, col, i, j +␣
↪1)
```



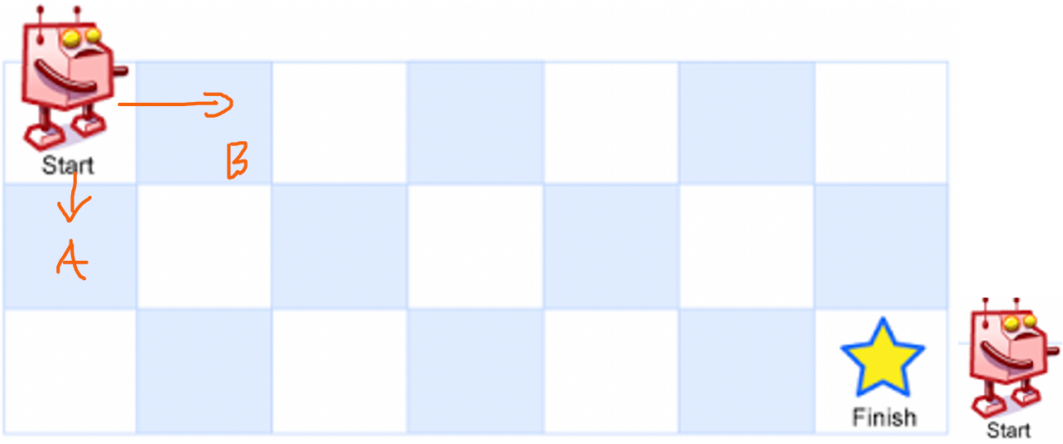
机器人从 start 走到 finish 的 路径数量 F(start,finish)

等于 F(A,finish) + F(B,finish)

这样就是找到了递归条件， start 坐标（i, j) ，A (i+1,j)，B(i,j+1)

Base case i == row-1, 并且 j == col-1 ,就是start 与finish 重合 的位置。

Fig. 7: image-20201108181334938



这样就是找到了递归条件， start 坐标（i，j），A (i+1,j)，B(i,j+1)

Base case i == row-1，并且 j == col-1 ,就是start 与finish 重合 的位置。

F(start(i,j),Finish) = F((i+1,j),Finish) + F((i,j+1),Finish)

Fig. 8: image-20201108181813895

记忆化搜索可以把结果存起来

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        i, j = 0, 0
        self.memo = dict()
        r = self.count_path(m, n, i, j)
        return r

    def count_path(self, row, col, i, j):
        if (i,j) in self.memo:
            return self.memo.get((i,j))

         # terminator
        if i >= row or j >= col:
            self.memo[(i, j)] = 0
            return self.memo[(i, j)]
        if i == row-1 and j == col-1:
            self.memo[(i, j)] = 1
            return self.memo[(i, j)]

        self.memo[(i, j)] = self.count_path(row, col, i + 1, j) + self.count_path(row,
↪ col, i, j + 1)
        return self.memo[(i,j)]
```

有没有更好的办法？可以自行思考一下.

6.3 3 递归的效率问题

递归有哪些问题呢?

效率对比

递归的话，需要额外的栈的空间开销，这个需要一定空间成本的。对于 for 循环就不太需要，直接循环，不需要额外的栈空间。

```
def recur(n=100):
    if n == 1:
        return 1
    return recur(n - 1) + n


def my_sum(n):
    _sum = 0
```

(continues on next page)

(continued from previous page)

```
    for i in range(n):
        _sum += i
    return _sum


if __name__ == '__main__':
    n = 500
    start = time.time()
    print(my_sum(n=n))
    print(f"my_sum totoal time :{time.time() - start}")

    start = time.time()
    print(recur(n=n))
    print(f"recur totoal time :{time.time() - start}")
    pass
```

6.4 总结

递归的关键要点第一要找到最近重复子问题，第二要找 base case 基线条件。然后开始写递归，递归过程中一定不要忘记递归的终止条件。

6.5 参考文档

爬楼梯问题
括号生成问题
不同路径
组合
全排列
全排列 II

分享快乐, 留住感动. '2020-11-17 06:56:28' –frank

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search