

第十四章类的宿主

一 反射

1. 补充内置方法
2. 反射内置函数

二 面向对象内置方法

1. `__str__`方法
2. `__del__`方法
3. `__call__`方法

三 元类

1. 补充exec的用法
2. 元类介绍（一切皆对象）
3. type自定义类
4. 自定义元类
5. 自定义类的调用

四 单例模式

本文是Python通用编程系列教程，已全部更新完成，实现的目标是从零基础开始到精通Python编程语言。本教程不是对Python的内容进行泛泛而谈，而是精细化，深入化的讲解，共5个阶段，25章内容。所以，需要有耐心的学习，才能真正有所收获。虽不涉及任何框架的使用，但是会对操作系统和网络通信进行全局的讲解，甚至会对一些开源模块和服务器进行重写。学完之后，你所收获的不仅仅是精通一门Python编程语言，而且具备快速学习其他编程语言的能力，无障碍阅读所有Python源码的能力和对计算机与网络的全面认识。对于零基础的小白来说，是入门计算机领域并精通一门编程语言的绝佳教材。对于有一定Python基础的童鞋，相信这套教程会让你的水平更上一层楼。

一 反射

1. 补充内置方法

正式讲反射之前，先来说两个小方法，与反射没有关系，但是有点像

```

class Foo:
    pass

foo = Foo()

# 这是我们今天讲的，以后使用这个方式判断一个对象是不是类的对象
print(isinstance(foo, Foo))

# 这是我们以前判断的方式，type另有用途，以后不使用这种方式判断
print(type(foo) is Foo)

# 同理，一切皆对象
print(isinstance('abc', str))

# Python3 中的一个任何类都是object子类
print(issubclass(Foo, object))

```

2. 反射内置函数

下述四个函数是专门用来操作类与对象属性的，如何操作？通过字符串来操作类与对象的属性，这种操作称为反射

`hasattr`, `getattr`, `setattr`, `delattr`，根据名字猜意思，`attr`是属性

详细代码示例如下：

```

class People:
    country="China"
    def __init__(self, name):
        self.name=name
    def tell(self):
        print('%s is aaa' %self.name)

obj=People('albert')

# 1、hasattr
print(hasattr(People, 'country'))
# 本质原理就是下面这行
print('country' in People.__dict__)
# 同理，一切皆对象
print(hasattr(obj, 'name'))
print(hasattr(obj, 'country'))
print(hasattr(obj, 'tell'))

# 2、getattr
x=getattr(People, 'country1', None)
# x=getattr(People, 'country', None)

```

```

# x=getattr(People,'country1') # 如果没有这个属性，不传None会报错
print(x)
f=getattr(obj,'tell',None) # obj.tell
print(f == obj.tell)
f()
obj.tell()
# 3、setattr
# People.x=111
setattr(People,'x',111)
print(People.x)
# obj.age=18
setattr(obj,"age",18)
print(obj.__dict__)
# 4、delattr
# del People.country
delattr(People,"country")
print(People.__dict__)
# del obj.name
delattr(obj,"name")
print(obj.__dict__)

```

之所以称之为反射，是因为把字符串映射成了一个具体的属性，因为我们平时是根据类或者对象来找属性的，而现在反过来找，所以叫反射。为什么我们要使用反射这种方式来对象的属性呢？从代码上看并没有简洁多少，主要是因为，反射是从字符串触发的，我们在获取用户输入的时候也是拿到的字符串，就在这里用。

现在有一个需求，模拟Linux系统用户输入指令，机器执行相应的操作

```

class Foo:
    def run(self):
        while True:
            cmd = input('cmd>>: ').strip()
            if hasattr(self, cmd):
                func = getattr(self, cmd)
                func()
            elif cmd == 'exit':
                break
    def download(self):
        print('download...')
    def upload(self):
        print('upload...')

obj = Foo()
obj.run()

```

二 面向对象内置方法

1. `__str__`方法

通过前面的学习，我想你已经很清楚一切皆对象的概念了，既然如此，现在有一个问题

```
class Foo:
    pass

obj = Foo()
l = [1, 2, 3, ]
print(obj)
print(l)
```

`obj`和`l`都是对象，为什么`l`能打印出来他的值，而`obj`不是打印他的值而是打印一个内存地址出来。这说明你在打印`l`这个对象的时候一定是触发了某个方法，指定就让他打印对象的值，我们的类能不能也触发这个方法呢？

这个方法就是`__str__`方法，在对象打印时自动触发

```
class People:
    def __init__(self, name, age, sex): # init对象实例化自动触发
        self.name = name
        self.age = age
        self.sex = sex

    def __str__(self): # str对象打印时自动触发
        # print('=====>')
        return '<名字:%s 年龄:%s 性别:%s>' % (self.name, self.age, self.sex)

obj = People('albert', 18, 'male')
print(obj) # print(obj.__str__())
```

2. `__del__`方法

直接猜一下这个方法什么时候自动触发，你能想到容易，但是我要像个办法验证一下

```
import time
class People:
```

```

def __init__(self, name, age, sex):
    self.name = name
    self.age = age
    self.sex = sex

def __del__(self): # 在对象被删除的条件下，自动执行
    print('__del__')

obj = People('albert', 18, 'male')
# del obj #obj.__del__()
time.sleep(5)

```

运行代码，先创建一个对象，然后睡5秒，这个时候程序就要关了，但是关之前要把在内存中创建的对象删掉，来吧del方法，这个时候就会出打印结果了，这就是你看到的结果的执行过程，手动删除当然也可以。

现在你会发现，其实del方法是与资源回收相关的，Python都已经自动回收了，那还需要我们回收什么呢？其实Python回收的是Python自己的资源，把我们写的代码看作应用程序，他只能回收应用程序的资源，系统资源谁来回收？下面我们先以文件操作为例来讲解

```

class MyOpen:
    def __init__(self, file_path, mode="r", encoding="utf-8"):
        self.file_path = file_path
        self.mode = mode
        self.encoding = encoding
        self.file_obj = open(file_path, mode=mode, encoding=encoding)

    def __str__(self):
        msg = ""
        file_path: %s
        mode: %s
        encoding: %s
        """ % (self.file_path, self.mode, self.encoding)
        return msg

    def __del__(self):
        self.file_obj.close()

f = MyOpen('test.py', mode='r', encoding='utf-8')
print(f.file_path, f.mode, f.encoding)
print(f)
print(f.file_obj)
res = f.file_obj.read()
print(res)

```

打开一个文件，这是从机器的硬盘上读取数据，占用的是操作系统的资源，在程序关闭之前，也要回收，通俗点说就是文件打开了，你要关掉

关闭文件回收的是操作系统的资源，如果连接数据库，当程序关掉的时候，是不是也要关掉连接，以下示例部分代码是伪代码

```
class Mysql:
    def __init__(self, ip, port):
        self.ip = ip
        self.port = port
        self.conn = connect(ip, port) # 申请系统资源, 伪代码
    def __del__(self):
        self.conn.close() # 关闭系统资源, 伪代码
obj = Mysql('1.1.1.1', 3306)
```

del函数就是用来回收系统资源的，Python内部的资源会自动执行del函数，这个函数又叫做析构函数。

3. `__call__`方法

call方法是在调用对象的时候自动执行（等下会用）

```
class Foo:
    def __init__(self):
        pass
    def __str__(self):
        return '123123'
    def __del__(self):
        pass
    # 调用对象，则会自动触发对象下的绑定方法__call__的执行，
    # 然后将对象本身当作第一个参数传给self，将调用对象时括号内的值
    # 传给*args与**kwargs
    def __call__(self, *args, **kwargs):
        print('__call__', args, kwargs)
obj = Foo()
# print(obj)
obj(1, 2, 3, a=1, b=2, c=3)
```

三元类

1. 补充exec的用法

exec()可以用来执行字符串内的代码，需要传三个参数，格式exec(字符串代码，全局名称空间，局部名称空间)

```
# 注意：字符串内的代码没有声明是全局的，默认就是局部的，这和我们常规的代码不同
code = """
x=1
y=2
"""

global_dic = {}
local_dic = {}
exec(code, global_dic, local_dic)
# print(global_dic) # 打印的东西很多，其实就是一个字典，这些都是内置的
print(local_dic)
```

如果我们现在自己定义全局，然后在把它在code里面改了，如下

```
code="""
global x # 先声明全局，再修改
x=111
y=2
"""

global_dic={'x':999}
local_dic={}
exec(code,global_dic,local_dic)
# print(global_dic) # 重点看字典的第一组键值对
print(local_dic)
```

这样就可以实现自己定制全局作用域和局部作用域，在字符串里面来操纵，那这个有什么用，其实也没什么用，我们只是用他来证明我们下面讲的元类的概念

现在有一个小需求，code里面写一段代码，我们需要把这段代码执行过程中，所产生的名称空全部放到局部名称空间中

```
code="""
x=1
y=2
def f1(self,a,b): # 这个self和面向对象没有鸡毛关系
    pass
"""
```

```
local_dic={}
exec(code, {}, local_dic)  # 我们不需要全局名称空间
print(local_dic)
```

2. 元类介绍（一切皆对象）

对象是对象，那么类也是对象，函数也是对象(这也是我第二阶段第一章标题名字叫做函数对象原因，可能你们一开始并不理解为什么这样命名，因为以前从未见到过)，既然是对象，那么必然是由一个类实例化而来，所以类或者函数的创建过程一定是要实例化这个类或者函数的对象

```
class Chinese:
    country = "China"
    def __init__(self, name, age, gender):
        self.name = name
        self.age = age
        self.gender = gender
    def speak(self):
        print('%s speak Chinese' % self.name)

p = Chinese('albert', 18, 'male')
print(type(p))  # 他的类是Chinese，既然一切皆对象，那么Chinese也应该是对象
print(type(Chinese))

#####代码分割线#####

def abc():
    pass
print(type(abc))
```

对象p的类是Chinese，Chinese的类是type，type的类是谁？这个就不要再问了，到头了。type实例化得到了Chinese，Chinese实例化得到了p，type可以称之为类的类，也就是本章标题类的宿主，当然这是一种比喻的说法（为了凑齐四个字，没办法）

既然如此，你在用class关键字在造一个Chinese类的时候，它的本质就是Chinese这个对象调了type这个类，完成了实例化，伪代码如下：

```
# 一切皆为对象：
# Chinese=type(...)
class Chinese:
    country = "China"

    def __init__(self, name, age, gender):
```


同理

```
# abc = function(...)
```

```
def abc():
    pass
```

```
print(type(abc))
```

结论：

类的类就是元类，我们用class定义的类是用来产生我们自己的对象的，内置元类type是用来专门产生class定义的类的

3. type自定义类

现在我们知道了自定义类其实有两种方式，第一用class，第二用type，接下来我们就来研究用内置的元类type（你所有用class关键字定义的类就是用type产生的）来实例化得到我们的类。

```
222
223 # print(type(Chinese))
224 # 元类：类的类就是元类，
self: type, o: object
self: type, name: str, bases: Tuple[type, ...], dict: Dict[str, Any]
227
228 type()
229
```

用type定义类先来看一下有几个参数：self或者cls请忽略，第一个name，也就是类名，数据类型字符串，第二个bases，类的基类/父类/超类，数据类型元祖，第三个dict，类的名称空间（类内部的名称空间一定是一个局部的名称空间），数据类型字典。这三个也是我们用class定义类的三大重点：类名，类的基类，类体代码（用class定义类会立刻执行类体内的代码，并把代码里面定义的名字放入类的名称空间，最后我们要是类的名称空间）

```
class_name='Chinese'
class_bases=(object,) # 注意这里必须有逗号，否则不是元组
class_body="""
country="China"
def __init__(self,name,age,gender):
    self.name=name
    self.age=age
    self.gender=gender
def speak(self):
    print('%s speak Chinese' %self.name)
```

```

"""
class_dic={}
exec(class_body,{},class_dic)  # 全局名称空间我们不需要
print(class_dic)

```

输出：

```

09 x /Users/albert/anaconda3/bin/python /Users/albert/Desktop/demo_code/09.py
{'country': 'China', '__init__': <function __init__ at 0x10d310ea0>, 'speak': <function speak at 0x10d525268>}

Process finished with exit code 0

```

可以自行与你用class关键字定义类的做比较，接下来我们来实例化对象

```

243 # print(class_dic)
244
245 obj = type(class_name,class_bases,class_dic)
246 print(obj)
247
9 x /Users/albert/anaconda3/bin/python /Users/albert/Desktop/demo_code/09.py
<class '__main__.Chinese'>

```

这个结果与使用class关键字定义的类实例化出来的结果是一样的，区别在于过程，用class关键字，会自动获取类名，自动默认继承object类，自动执行类体内的代码拿到类的名称空间，并把这个三个当作参数传递给type来实例化类并赋值给一个变量名叫Chinese，如果我们不赋值给obj，而赋值Chinese也就是一样了。测试如下：

```

244
245 Chinese = type(class_name,class_bases,class_dic)
246
247 albert = Chinese('albert',18,'male')
248 # 这样没有对象属性的代码提示，这是Pycharm不完善
249 print(albert.name,albert.age,albert.gender)
250
09 x /Users/albert/anaconda3/bin/python /Users/albert/Desktop/demo_code/09.py
albert 18 male

```

所以你以后定义类就用type定义就好了，这样逼格比较高（你一定会被同事打死的）

4. 自定义元类

每次创建类的时候都是既然都是由type实例化而来，那么如果我们能够控制type理论上讲就可以控制类的创建，也就可以在类创建时添加任意的逻辑，但是type我们是没办法改的，不过我们可以修改创建类的时候默认的元类，默认就是type，如果我们自己写另外一个元类，并且将创建类的默认使用的元类修改为我们自己写的元类，就可以实现

```
295
296
297 class Foo(metaclass=type):
298     pass
299
```

297行type是创建Foo类默认的元类，因为指定默认参数，所以我们平时不写也是一样的，现在我们要修改默认参数，就要在它上面自己定义一个type元类：

```
296 class MyType(type): # 我们只是重写type的部分功能，所以要继承这个类
297     pass
298
299
300 # Foo=MyType('Foo',(object,),class_dic) # 本质就是在用MyType这个元类实例化Foo对象
301 class Foo(metaclass=MyType): # 现在要把Foo看作是一个对象
302     pass
303
```

300行注释中既然是实例化，那么MyType这个类中必然会有一个init方法，我们可以先来验证一下

```

dhfk.docx
ettings.py
st.py
rnal Libraries
atches and Consoles

295
296 class MyType(type): # 我们只是重写type的部分功能，所以
297     def __init__(self):
298         pass
299
300
301 # Foo=MyType('Foo',(object,),class_dic) # 本质就是
302 class Foo(metaclass=MyType): # 现在要把Foo看作是一个
303     pass
304

MyType > __init__()

09 x
/Users/albert/anaconda3/bin/python /Users/albert/Desktop/demo_code/09.py
Traceback (most recent call last):
  File "/Users/albert/Desktop/demo_code/09.py", line 302, in <module>
    class Foo(metaclass=MyType): # 现在要把Foo看作是一个对象
TypeError: __init__() takes 1 positional argument but 4 were given

```

自动传了四个参数，但是init函数中只需要一个参数，另外注意：基类object必须要传，当然你不传他也不会报错（经典类与新式类），修改如下：

```

dhfk.docx
ettings.py
est.py
ernal Libraries
atches and Consoles

295
296 class MyType(type): # 我们只是重写type的部分功能，所以要继承这个类
297     def __init__(self, class_name, class_bases, class_dic):
298         print(class_name)
299         print(class_bases)
300         print(class_dic)
301
302
303 # Foo=MyType('Foo',(object,),class_dic) # 本质就是在用MyType这个元类
304 class Foo(object, metaclass=MyType): # 现在要把Foo看作是一个对象
305     pass
306

09 x
/Users/albert/anaconda3/bin/python /Users/albert/Desktop/demo_code/09.py
Foo
(<class 'object'>,)
{'__module__': '__main__', '__qualname__': 'Foo'}
```

刚才我们已经完成了控制类的创建的行为，接下来我们就来试试：

我们定义类名最好是用大驼峰体，那么必定是首字母大写的，如果不用大驼峰体，那也没人管得了你，现在我就能管了

```

295
296 class MyType(type): # 我们只是重写type的部分功能，所以要继承这个类
297     def __init__(self, class_name, class_bases, class_dic):
298         if not class_name.istitle():
299             raise TypeError('类名的首字母必须大写傻叉')
300
301
302 # Foo=MyType('Foo',(object,),class_dic) # 本质就是在用MyType这个类
303 class foo(object, metaclass=MyType): # 现在要把Foo看作是一个对象
304     pass
305

```

09 x

```

Traceback (most recent call last):
  File "/Users/albert/Desktop/demo_code/09.py", line 303, in <module>
    class foo(object, metaclass=MyType): # 现在要把Foo看作是一个对象
  File "/Users/albert/Desktop/demo_code/09.py", line 299, in __init__
    raise TypeError('类名的首字母必须大写傻叉')
TypeError: 类名的首字母必须大写傻叉

```

299行代码下一章会讲到，现在类名首字母没有大写，直接报错

重写与重用

```

class MyType(type):
    def __init__(self, class_name, class_bases, class_dic):
        if not class_name.istitle():
            raise TypeError('类名的首字母必须大写傻叉')
        # 我们重写了init方法，但是不能全部重写，所以还要重用父类的功能
        super(MyType, self).__init__(class_name, class_bases, class_dic)
class Foo(object, metaclass=MyType):
    pass

```

补充关于文档注释doc方法

```
Final Libraries
atches and Consoles

308
309 class Student:
310     """
311     学生类
312     """
313     pass
314
315
316 print(Student.__doc__)
317
```

09 x /Users/albert/anaconda3/bin/python /Users/albert/Desktop/demo_

学生类

如果不写文档注释就是None，所有的类都有doc这个属性，Student也不例外，这个属性一定是存在于他的名称空间的

```
class Student:

    pass

# print(Student.__doc__)
print(Student.__dict__)

code/09.py
'Student' objects>, '__weakref__': <attribute '__weakref__' of 'Student' objects>, '__doc__': None}
```

你要是再不写文档注释，哼哼，直接上代码吧

```
class MyType(type):
    def __init__(self, class_name, class_bases, class_dic):
        if not class_name.istitle():
            raise TypeError('类名的首字母必须大写傻叉')
        if not class_dic.get('__doc__'):
            raise TypeError('类中必须写好文档注释，大傻叉')
```

```
# 我们重写了init方法，但是不能全部重写，所以还要重用父类的功能
super(MyType, self).__init__(class_name, class_bases, class_dic)
class Foo(object, metaclass=MyType):
    pass
```

5. 自定义类的调用

类也是一个对象，那么我们在调用类这个对象的时候就会自动执行它里面的`call`方法

```
295
296 class MyType(type):
297     def __init__(self, class_name, class_bases, class_dic):
298         if not class_name.istitle():
299             raise TypeError('类名的首字母必须大写傻叉')
300         if not class_dic.get('__doc__'):
301             raise TypeError('类中必须写好文档注释，大傻叉')
302
303         # 我们重写了init方法，但是不能全部重写，所以还要重用父类的功能
304         super(MyType, self).__init__(class_name, class_bases, class_dic)
305
306
307 class Foo(object, metaclass=MyType):
308     pass
309
310
311 Foo()
```

把`Foo`看作是一个对象，311行调用这个对象就会自动去找他的`call`方法，很明显它自己的名称空间中沒有，那就会去他的类`MyType`中找，`MyType`中也没有，那就会去他的父类`type`中找，这回肯定能找到，但是这就失去了我们定制元类的意义了，所以我们要在`MyType`类中自己定义`call`方法。

```

.py
.py
.py
.txt
mo.py
hfk.docx
ttings.py
st.py
nal Libraries
ches and Consoles

296 class MyType(type):
297     def __init__(self, class_name, class_bases, class_dic):
298         super(MyType, self).__init__(class_name, class_bases, class_dic)
299
300     def __call__(self, *args, **kwargs):
301         print('=====>')
302
303
304 class Foo(object, metaclass=MyType):
305     pass
306
307
308 Foo()
309

MyType > __call__()

/Users/albert/anaconda3/bin/python /Users/albert/Desktop/demo_code/09.py
=====>

```

只要执行代码，就会调用这个类，那么就会打印那一行箭头，接下来我们再来改一下

```

.py
s.py
3.py
7.py
3.py
9.py
o.txt
emo.py
hfk.docx
ttings.py
st.py
nal Libraries
ches and Consoles

296 class MyType(type):
297     def __init__(self, class_name, class_bases, class_dic):
298         super(MyType, self).__init__(class_name, class_bases, class_dic)
299
300     def __call__(self, *args, **kwargs):
301         print('=====>')
302
303
304 class Foo(object, metaclass=MyType):
305     def __init__(self, y):
306         self.Y = y
307
308     def f1(self):
309         print('from f1')
310
311
312 obj = Foo(1)
313 print(obj)
314

/Users/albert/anaconda3/bin/python /Users/albert/Desktop/demo_code/09.py
=====>
None

```

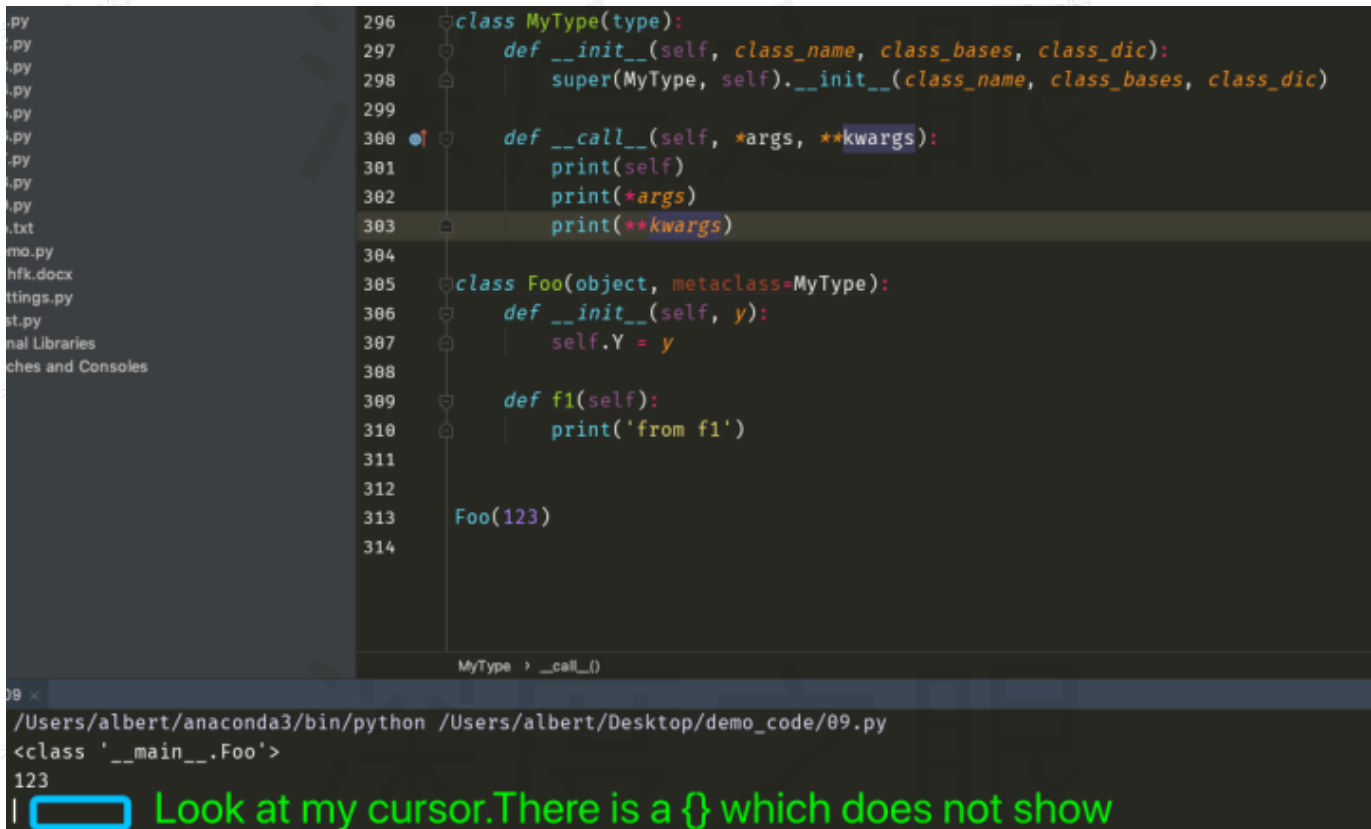
调用类不是应该传参吗，打印对象居然是None？这是什么鬼？

运行结果与我们以前的说法并不一致，只是因为，以前我们说的调类会产生一个空对象，但是他不可能凭空产生，必然是出发触发了它内部的某个方法，现在你知道了，就是call函数，那么call函数没有返回值，obj自然就是None

现在再把Foo看成是一个对象，在调用这个对象的时候会把Foo这个对象本身给了self，123给了args，kwargs不接收参数，这么看的话，调用Foo不传参数是不会报错的（因为星args和星星kwargs的组合可以不传任何参数，也可以传任何参数）


```
296 class MyType(type):
297     def __init__(self, class_name, class_bases, class_dic):
298         super(MyType, self).__init__(class_name, class_bases, class_dic)
299
300     def __call__(self, *args, **kwargs):
301         print('====>')
302
303
304 class Foo(object, metaclass=MyType):
305     def __init__(self, y):
306         self.Y = y
307
308     def f1(self):
309         print('from f1')
310
311
312 Foo(123)
313
```

口说无凭，我们来验证一下：



```
296 class MyType(type):
297     def __init__(self, class_name, class_bases, class_dic):
298         super(MyType, self).__init__(class_name, class_bases, class_dic)
299
300     def __call__(self, *args, **kwargs):
301         print(self)
302         print(*args)
303         print(**kwargs)
304
305 class Foo(object, metaclass=MyType):
306     def __init__(self, y):
307         self.Y = y
308
309     def f1(self):
310         print('from f1')
311
312
313 Foo(123)
314
```

MyType > __call__()

09 x /Users/albert/anaconda3/bin/python /Users/albert/Desktop/demo_code/09.py

<class '__main__.Foo'>

123

Look at my cursor. There is a {} which does not show

这里面不知道为什么打不了中文，那就炫一下我这蹩脚的英文吧

现在我们已经可以控制类的调用了，我们调用Foo这个类的初衷是什么？是不是让他产生一个obj空对象，也就是要让他返回值是一个空对象，并让这个obj对象本身连同Foo括号内的参数一同传给第306行init，并且调用第306行的init完成foo对象的初始化

```
class MyType(type):
    def __init__(self, class_name, class_bases, class_dic):
        super(MyType, self).__init__(class_name, class_bases, class_dic)
    def __call__(self, *args, **kwargs):  # self = Foo
        # 1 造一个空对象obj
        obj = object.__new__(self)  # self = Foo
        # 2 调用Foo.__init__, 将obj连同调用Foo括号内的参数一同传给__init__
        self.__init__(obj, *args, **kwargs)  # self = Foo
        # 3 返回一个实例化之后的对象
        return obj

class Foo(object, metaclass=MyType):
    def __init__(self, y):
        self.y = y

Foo(123)
```

说明：object有一个__new__方法，是用来造一个空对象的，你想造哪个类的对象，就把那个类传进来，现在我们造的是Foo的对象，所以把Foo传进去

```

296 class MyType(type):
297     def __init__(self, class_name, class_bases, class_dic):
298         super(MyType, self).__init__(class_name, class_bases, class_dic)
299
300     def __call__(self, *args, **kwargs): # self = Foo
301
302         # 1 造一个空对象obj
303         obj = object.__new__(self) # self = Foo
304
305         # 2 调用Foo.__init__, 将obj连同调用Foo括号内的参数一同传给__init__
306         self.__init__(obj, *args, **kwargs) # self = Foo
307
308         # 3 返回一个实例化之后的对象
309         return obj
310
311
312 class Foo(object, metaclass=MyType):
313     def __init__(self, y):
314         self.y = y
315
316
317 Foo()
318

```

parameter passing

```

File "/Users/albert/Desktop/demo_code/09.py", line 317, in <module>
    Foo()
File "/Users/albert/Desktop/demo_code/09.py", line 306, in __call__
    self.__init__(obj, *args, **kwargs) # self = Foo
TypeError: __init__() missing 1 required positional argument: 'y'

```

第300行到第306行参数会自动传递，这种自动传递所有参数的方式在源码里经常会看到，现在已经完成标准的自定义类的调用，再调用Foo这个类，不传参数肯定是不行了

```

class MyType(type):
    # 控制类Foo的创建
    def __init__(self, class_name, class_bases, class_dic):
        super(MyType, self).__init__(class_name, class_bases, class_dic)
    # 控制类Foo的调用，即控制实例化Foo的过程
    def __call__(self, *args, **kwargs): # self = Foo
        # 1 造一个空对象obj
        obj = object.__new__(self) # self = Foo
        # 2 调用Foo.__init__, 将obj连同调用Foo括号内的参数一同传给__init__
        self.__init__(obj, *args, **kwargs) # self = Foo
        # 3 返回一个实例化之后的对象
        return obj
class Foo(object, metaclass=MyType):
    def __init__(self, y):
        self.y = y
    def f1(self):
        print('from f1')
obj = Foo(123)

```

```
print(obj)
print(obj.y)
print(obj.fl)
```

四 单例模式

单例模式：顾名思义就是单个实例的意思，我们使用单例模式就是为了节省内存空间。

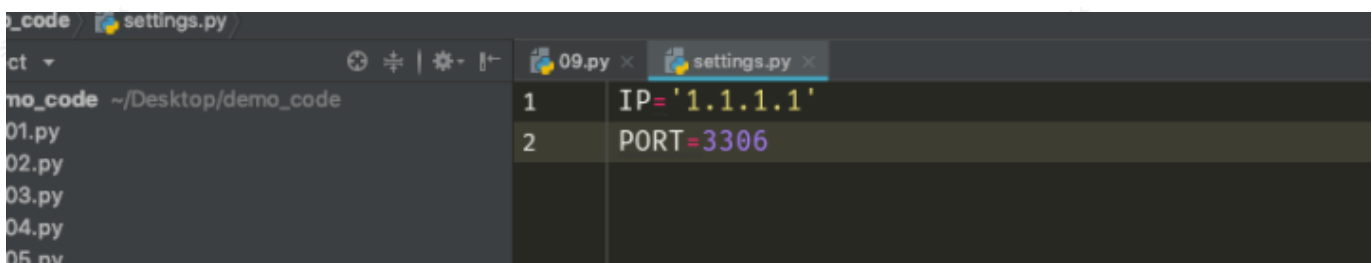
今天我们就来讲单例模式，那么什么场景下会用到单例模式来节省内存空间呢？

```
class MySQL:
    def __init__(self, ip, port):
        self.ip = ip
        self.port = port

obj1 = MySQL('1.1.1.2', 3306)
obj2 = MySQL('1.1.1.3', 3307)
obj3 = MySQL('1.1.1.4', 3308)

print(obj1)
print(obj2)
print(obj3)
```

现在是在手动输入参数实例化对象，但是实际工作当中，我们都是从配置文件来读取配置来实例化的，配置文件信息如下：



```
import settings
class MySQL:
    def __init__(self, ip, port):
        self.ip = ip
        self.port = port
    @classmethod
    def read_from_conf(cls):
        obj = cls(settings.IP, settings.PORT)
```

```

        return obj

obj4 = MySQL.read_from_conf()
obj5 = MySQL.read_from_conf()
obj6 = MySQL.read_from_conf()
print(obj4)
print(obj5)
print(obj6)

```

输出：

```

<__main__.MySQL object at 0x10e4e16a0>
<__main__.MySQL object at 0x10e4e16d8>
<__main__.MySQL object at 0x10e4e1940>

```

三个obj明显不是一个内存地址，因为对实例化了三次就会需要三个对象的名称空间来存他们各自的属性，但是我们存不是目的，目的是为了取，取得时候确实一样的，这样显然浪费了内存空间而毫无意义

```

import settings
class MySQL:
    # 先定义一个空的实例，由于这是给内部用的，所以我们把它隐藏
    __instance = None
    def __init__(self, ip, port):
        self.ip = ip
        self.port = port
    @classmethod
    def create_singleton(cls): # 这里函数名字改了
        if not cls.__instance:

            # 第一次调用__instance是空走这个分支，创建一个obj
            obj = cls(settings.IP, settings.PORT)
            # 把obj赋值给类属性__instance
            cls.__instance = obj
            # 第二次调用类属性__instance有值，走这个分支，直接返回第一次调用的属性值
            return cls.__instance
obj1 = MySQL.create_singleton()
obj2 = MySQL.create_singleton()
obj3 = MySQL.create_singleton()
print(obj1 is obj2 is obj3)

```

至此：单例模式的逻辑就讲完了，虽然我没有使用元类，但是元类也能实现。

总结：元类这一章，看似很高深，但是核心就只有控制类的调用也就是实例化类对象的那3行代码。如果前面的内容你都理解了，那么恭喜你，现在你已经掌握了全世界所有会用Python写代码的人中超过60%的人所不会的技能了。我们学习这一章节的用处非常广泛，比如你们调用的包或者开源组件或者使用的一些框架中几乎所有的源码中都会使用元类，你以后不仅可以看懂，甚至还可以自己写开源组建或者框架。

作者：马一特

深度之眼

作者：马一特

深度之眼

深度之眼