python面向对象进阶

类的成员

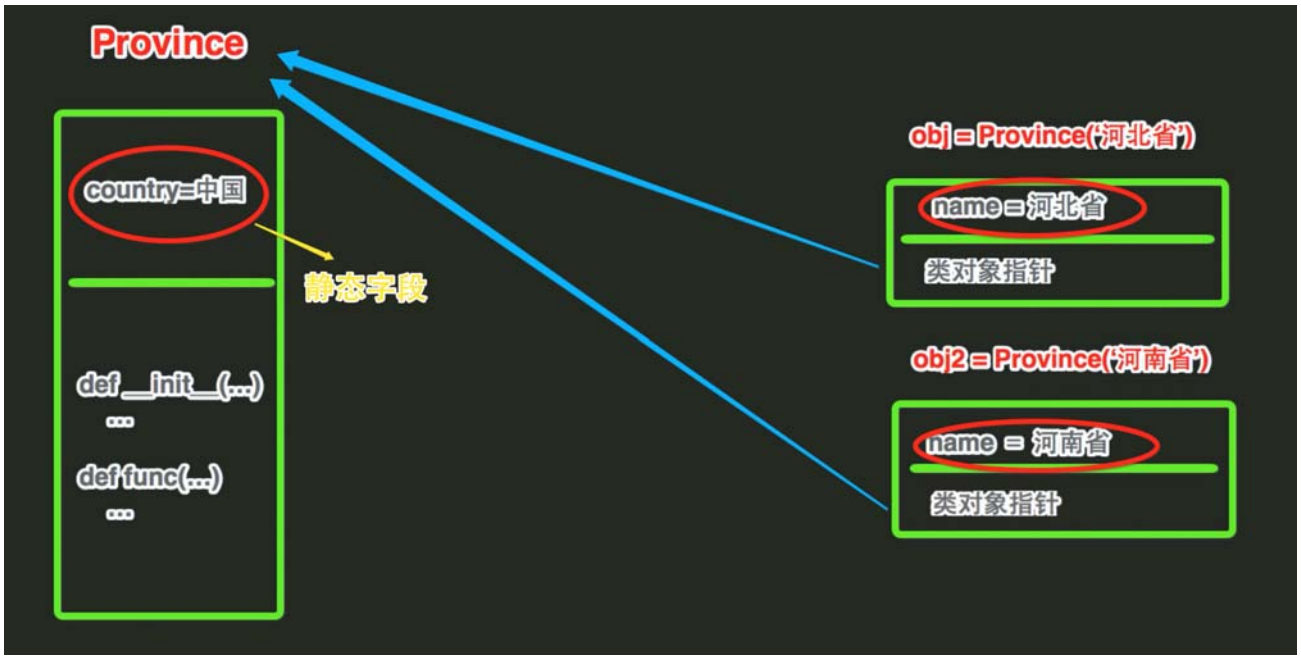
类的成员可以分为三大类：字段，方法，属性

一、字段

字段包括普通字段和静态字段，他们在定义和使用中有所区别，而最本质的区别是在内存中保存的位置不同

- 普通字段属于对象
- 静态字段属于类

```
class Province:

    # 静态字段
    country = '中国'

    def __init__(self, name):

        # 普通字段
        self.name = name


# 直接访问普通字段
obj = Province('河北省')
print obj.name

# 直接访问静态字段
Province.country
```

> 内存中保存的情况：
>
> 
>

- 静态字段在内存中只保存一份
- 普通字段在每个对象中都保存一份

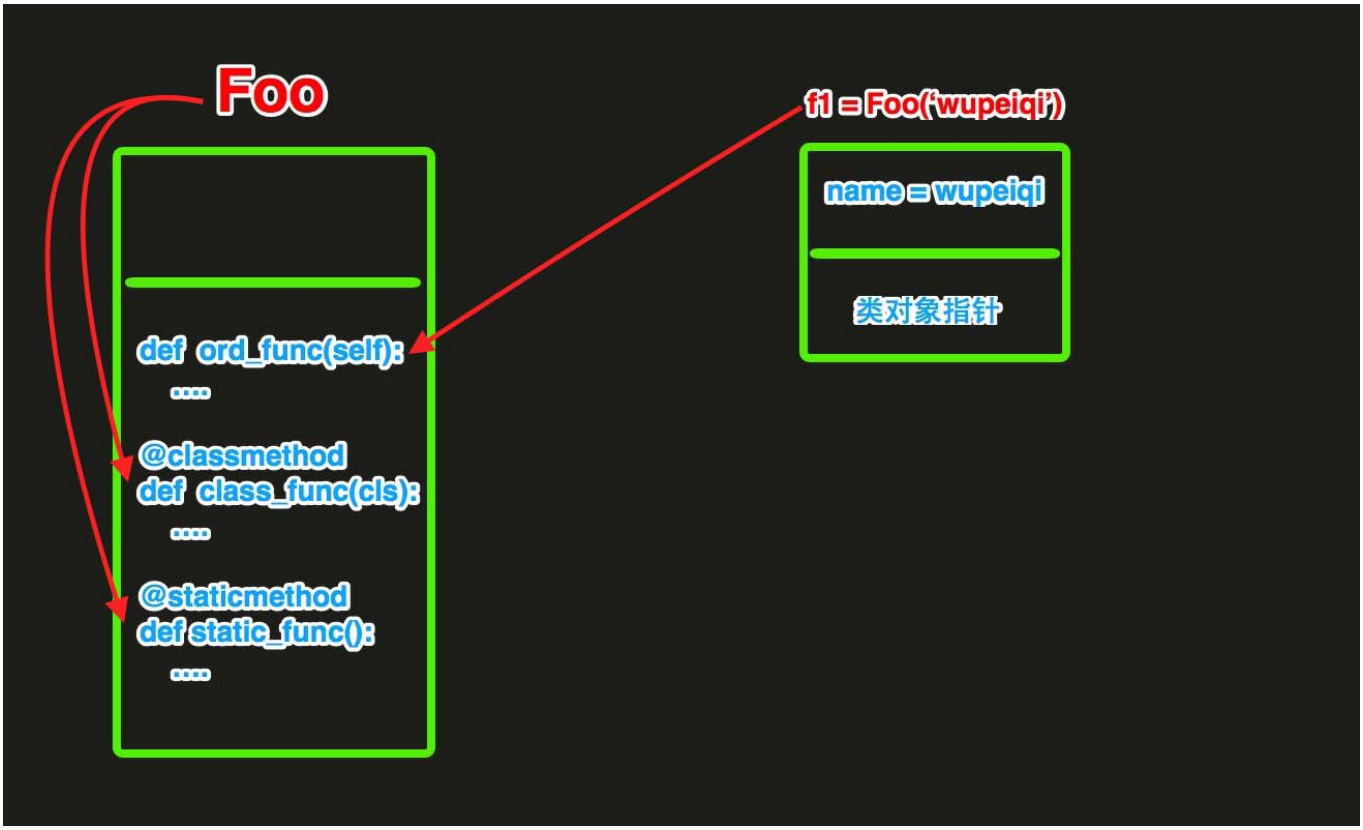
二、方法

方法包括普通方法，静态方法，类方法，三种方法在内存中都归属于类，区别在于调用的方式不同

- 普通方法：由对象调用；至少一个 `self` 参数；执行普通方法时，自动将调用该方法的对象赋值给 `self` ；
- 类方法：由类调用；至少一个 `cls` 参数；执行类方法时，自动将调用该方法的类赋值给 `cls` ；
- 静态方法：由类调用，无默认参数；

```
class Foo:

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

    def ord_func(self):
        """ 定义普通方法，至少有一个self参数 """

        # print self.name
        print '普通方法'

    @classmethod
    def class_func(cls):
        """ 定义类方法，至少有一个cls参数 """

        print '类方法'

    @staticmethod
    def static_func():
        """ 定义静态方法 ，无默认参数"""

        print '静态方法'


# 调用普通方法
f = Foo()
f.ord_func()

# 调用类方法
Foo.class_func()

# 调用静态方法
Foo.static_func()
```

方法的定义和使用



> **相同点** —— 对于所有的方法而言，均属于类中，所以在内存中只保存一份；

> **不同点** —— 方法的调用者不同，调用方法时自动传入的参数不同；

三、属性

1、属性的基本使用

```
# ############### 定义 ###############
class Foo:

    def func(self):
        pass

    # 定义属性
    @property
    def prop(self):
        pass
# ############### 调用 ###############
foo_obj = Foo()

foo_obj.func()
foo_obj.prop   #调用属性
```

属性的定义和使用

python的属性的功能是：属性内部进行一系列的逻辑运算，最终将计算结果返回

2.属性的两种定义方式

- 装饰器，在方法上用装饰器

- 静态字段，在类中定义值为property对象的静态字段

装饰器方式

```
# ############### 定义 ###############
class Goods:

    @property
    def price(self):
        return "wupeiqi"
# ############### 调用 ###############
obj = Goods()
result = obj.price  # 自动执行 @property 修饰的 price 方法，并获取方法的返回值
```

具有三种 @property 装饰器

```
# ############### 定义 ###############
class Goods(object):

    @property
    def price(self):
        print '@property'

    @price.setter
    def price(self, value):
        print '@price.setter'

    @price.deleter
    def price(self):
        print '@price.deleter'

# ############### 调用 ###############
obj = Goods()

obj.price          # 自动执行 @property 修饰的 price 方法，并获取方法的返回值

obj.price = 123    # 自动执行 @price.setter 修饰的 price 方法，并将  123 赋值给方法的参数


del obj.price      # 自动执行 @price.deleter 修饰的 price 方法



# 新式类中的属性有三种访问方式，并分别对应了三个被@property,@方法名.setter,@方法名.deleter 修饰的方法
```

由于新式类中具有三种访问属性的方式，我们可以根据他们几个属性的访问特点，分别将三个方法定义为对同一个属性：获取，修改，删除

```
class Goods(object):

    def __init__(self):
        # 原价
        self.original_price = 100
        # 折扣
        self.discount = 0.8
```

```
    @property
    def price(self):
        # 实际价格 = 原价 * 折扣
        new_price = self.original_price * self.discount
        return new_price

    @price.setter
    def price(self, value):
        self.original_price = value

    @price.deltter
    def price(self, value):
        del self.original_price

obj = Goods()
obj.price         # 获取商品价格
obj.price = 200   # 修改商品原价
del obj.price     # 删除商品原价
```

静态字段方式

property的构造方法中有四个参数

- 第一个参数是方法名，调用 对象.属性 时自动触发
- 第二个参数是方法名，调用 对象.属性 = xxx 时自动触发
- 第三个参数是方法名，调用 del 对象.属性 时自动触发
- 第四个参数是字符串，调用 对象.属性.__doc__ ，此参数是该属性的详细信息

```
class Foo:

    def get_bar(self):
        return 'abc'

    # 必须两个参数
    def set_bar(self, value):
        return "set value" , value

    def del_bar(self):
        return 'abc'

    BAR = property(get_bar, set_bar, del_bar, 'description...')

obj = Foo()

obj.BAR          # 自动调用第一个参数中定义的方法：get_bar
obj.BAR = '123' # 自动调用第二个参数中定义的方法：set_bar方法，并将“123”当作参数传入
del obj.BAR      # 自动调用第三个参数中定义的方法：del_bar方法
obj.BAR.__doc__ # 自动获取第四个参数中设置的值：description...
```

类成员的修饰符

- 公有成员，在任何位置都可以访问

- 私有成员，只有在类内部才能访问

*私有成员和公有成员的定义不同：私有成员命名时，前两个字符是下划线。（特殊成员除外，例如__init__，**call, dict**）*

```
class C:

    def __init__(self):
        self.name = '公有字段'
        self.__foo = '私有字段'
```

私有成员和公有成员的访问限制不同

- 静态字段
  i. 公有静态字段：类可以访问；类内部可以访问；派生类中可以访问
  ii. 私有静态字段：仅类内部可以访问；

```
class C:

    name = "公有静态字段"

    def func(self):
        print C.name

class D(C):

    def show(self):
        print C.name


C.name         # 类访问

obj = C()
obj.func()     # 类内部可以访问

obj_son = D()
obj_son.show() # 派生类中可以访问
```

公有静态字段

```
class C:

    __name = "私有静态字段"

    def func(self):
        print C.__name

class D(C):

    def show(self):
        print C.__name


C.__name       # 类访问            ==> 错误
```

```
obj = C()
obj.func()      # 类内部可以访问      ==> 正确

obj_son = D()
obj_son.show() # 派生类中可以访问    ==> 错误
```

私有静态字段

- 普通字段
  i. 公有普通字段：对象可以访问；类内部可以访问；派生类中可以访问
  ii. 私有普通字段：仅类内部可以访问；
  iii. *如果想要强制访问私有字段，可以通过* 对象._类名__私有字段 *访问*

```
class C:

    def __init__(self):
        self.foo = "公有字段"

    def func(self):
        print self.foo   #  类内部访问

class D(C):

    def show(self):
        print self.foo  #  派生类中访问

obj = C()

obj.foo     # 通过对象访问
obj.func()  # 类内部访问

obj_son = D();
obj_son.show()  # 派生类中访问
```

```
class C:

    def __init__(self):
        self.__foo = "私有字段"

    def func(self):
        print self.foo   #  类内部访问

class D(C):

    def show(self):
        print self.foo  #  派生类中访问

obj = C()

obj.__foo     # 通过对象访问    ==> 错误
obj.func()  # 类内部访问         ==> 正确

obj_son = D();
obj_son.show()  # 派生类中访问  ==> 错误
```

类的特殊成员

1. __doc__

> 表示类的描述信息

```
class Foo:
    """ 描述类信息，这是用于看片的神奇 """

    def func(self):
        pass

print Foo.__doc__
#输出：类的描述信息
```

2. __module__ 和 __class__

> __module__ 表示当前操作的对象在那个模块

> __class__ 表示当前操作的对象的类是什么

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-

class C:

    def __init__(self):
        self.name = 'wupeiqi'

lib/aa.py



from lib.aa import C

obj = C()
print obj.__module__  # 输出 lib.aa，即：输出模块
print obj.__class__      # 输出 lib.aa.C，即：输出类
```

3. __init__

> 构造方法，通过类来创建对象时，自动触发执行

```
class Foo:

    def __init__(self, name):
        self.name = name
        self.age = 18


obj = Foo('wupeiqi') # 自动执行类中的 __init__ 方法
```

4. __del__

> 析构方法，当对象在内存中被释放时，自动触发执行

```
class Foo:

    def __del__(self):
        pass
```

5. __call__

> 对象后面加括号，自动触发执行

```
class Foo:

    def __init__(self):
        pass

    def __call__(self, *args, **kwargs):

        print '__call__'


obj = Foo() # 执行 __init__
obj()       # 执行 __call__
```

6. __dict__

> 类或对象中的成员字典

```
class Province:

    country = 'China'

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

    def func(self, *args, **kwargs):
        print 'func'

# 获取类的成员，即：静态字段、方法、
print Province.__dict__
# 输出：{'country': 'China', '__module__': '__main__', 'func': <function func at 0x10be30f50>,
'__init__': <function __init__ at 0x10be30ed8>, '__doc__': None}

obj1 = Province('HeBei',10000)
print obj1.__dict__
# 获取 对象obj1 的成员
# 输出：{'count': 10000, 'name': 'HeBei'}

obj2 = Province('HeNan', 3888)
```

```
print obj2.__dict__
# 获取 对象obj1 的成员
# 输出: {'count': 3888, 'name': 'HeNan'}
```

7. __str__

> 如果一个类定义了 __str__ 方法，那么在打印对象时，默认输出该方法的返回值

```
class Foo:

    def __str__(self):
        return 'wupeiqi'


obj = Foo()
print obj
# 输出: wupeiqi
```

8. __getitem__ , __setitem__ , __delitem__

> 用于索引，如字典。以上分别表示获取，设置，删除数据

```
class Foo(object):

    def __getitem__(self, key):
        print '__getitem__',key

    def __setitem__(self, key, value):
        print '__setitem__',key,value

    def __delitem__(self, key):
        print '__delitem__',key


obj = Foo()

result = obj['k1']      # 自动触发执行 __getitem__
obj['k2'] = 'wupeiqi'   # 自动触发执行 __setitem__
del obj['k1']           # 自动触发执行 __delitem__
```

9. __getslice__ , __setslice__ , __delslice__

> 该三个方法用于分片操作

```
class Foo(object):

    def __getslice__(self, i, j):
        print '__getslice__',i,j

    def __setslice__(self, i, j, sequence):
        print '__setslice__',i,j

    def __delslice__(self, i, j):
```

```
        print '__delslice__',i,j

obj = Foo()

obj[-1:1]                   # 自动触发执行 __getslice__
obj[0:1] = [11,22,33,44]    # 自动触发执行 __setslice__
del obj[0:2]                # 自动触发执行 __delslice__
```

10. __iter__

> 用于迭代器，之所以字典，列表，元组可以进行for循环，是因为类型内部定义了 __iter__

```
class Foo(object):
    pass


obj = Foo()

for i in obj:
    print i

# 报错: TypeError: 'Foo' object is not iterable
# ============================================================
class Foo(object):

    def __iter__(self):
        pass

obj = Foo()

for i in obj:
    print i

# 报错: TypeError: iter() returned non-iterator of type 'NoneType'
# ============================================================
class Foo(object):

    def __init__(self, sq):
        self.sq = sq

    def __iter__(self):
        return iter(self.sq)

obj = Foo([11,22,33,44])

for i in obj:
    print i
```

> 以上步骤可以看出，for循环迭代其实是 `iter([11, 22, 33, 44])` ，所以执行流程变更为：

```
obj = iter([11,22,33,44])

for i in obj:
    print i
```

11. **__new__ , __metaclass__**

阅读以下代码：

```
class Foo(object):

    def __init__(self):
        pass

obj = Foo()   # obj是通过Foo类实例化的对象
```

上述代码中，obj 是通过 Foo 类实例化的对象，其实，不仅 obj 是一个对象，Foo类本身也是一个对象，因为在Python中一切事物都是对象。

如果按照一切事物都是对象的理论：obj对象是通过执行Foo类的构造方法创建，那么Foo类对象应该也是通过执行某个类的 构造方法 创建。

```
print type(obj) # 输出: <class '__main__.Foo'>     表示，obj 对象由Foo类创建
print type(Foo) # 输出: <type 'type'>              表示，Foo类对象由 type 类创建
```

所以，obj对象是Foo类的一个实例，Foo类对象是type类的一个实例，即：Foo类对象是通过type类的构造方法创建的。 那么，创建类就可以有两种方式:

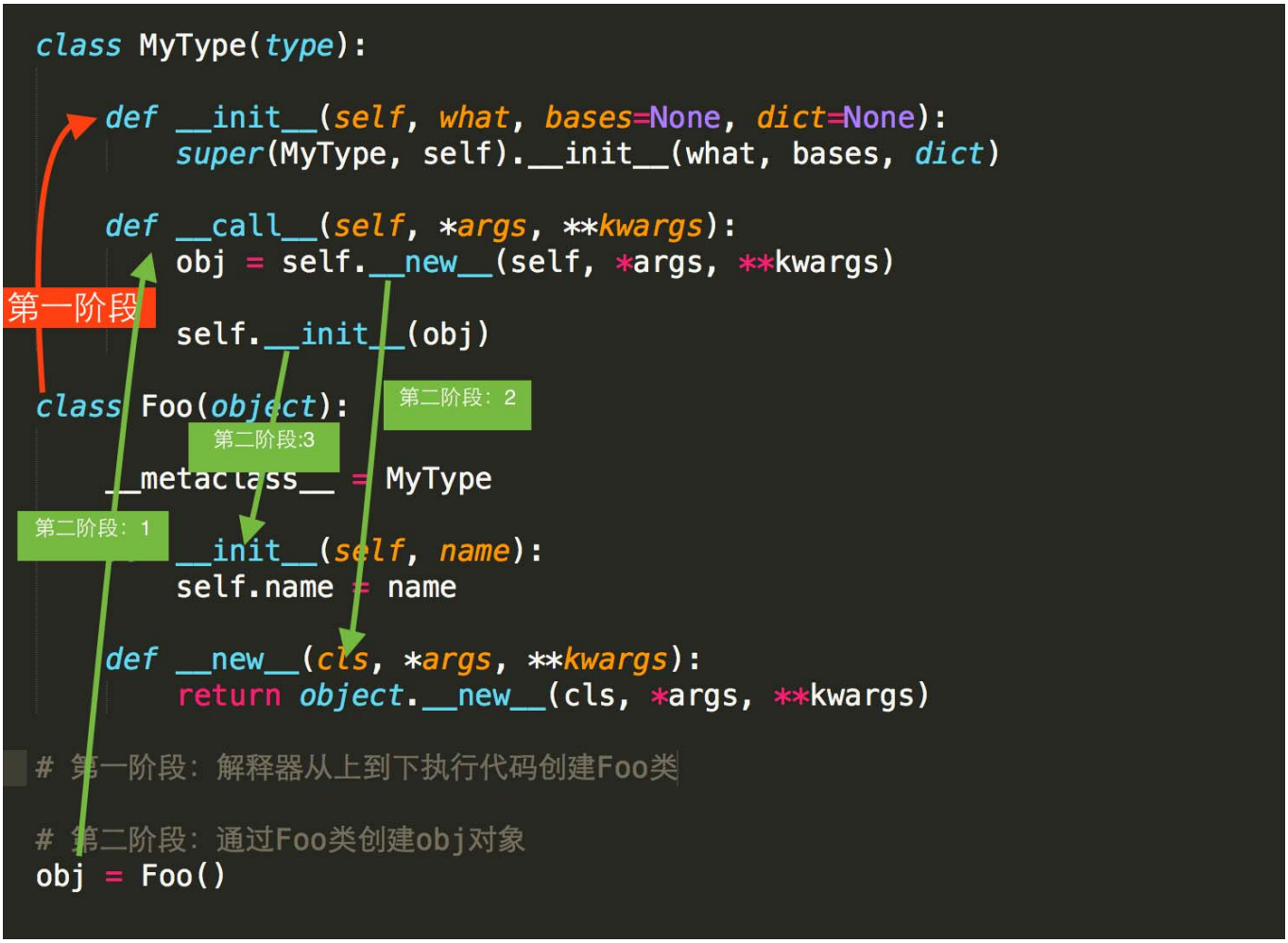
- 普通方式

```
class Foo(object):

    def func(self):
        print 'hello wupeiqi'
```

- 特殊方式（type类的构造函数）

```
def func(self):
    print 'hello wupeiqi'

Foo = type('Foo',(object,), {'func': func})
# type第一个参数：类名
# type第二个参数：当前类的基类
# type第三个参数：类的成员
```

类默认是由 type 类实例化产生的， type 类中如何实现的创建类？类又是如何创建对象？ 答：类中有一个属性 __metaclass__ ，其用来表示该类由谁来实例化创建，所以，我们可以为 __metaclass__ 设置一个 type 类的派生类，从而查看类的创建过程。




```python
class MyType(type):

    def __init__(self, what, bases=None, dict=None):
        super(MyType, self).__init__(what, bases, dict)

    def __call__(self, *args, **kwargs):
        obj = self.__new__(self, *args, **kwargs)

        self.__init__(obj)

class Foo(object):

    __metaclass__ = MyType

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

    def __new__(cls, *args, **kwargs):
        return object.__new__(cls, *args, **kwargs)

# 第一阶段：解释器从上到下执行代码创建Foo类
# 第二阶段：通过Foo类创建obj对象
obj = Foo()
```