**Python语言学习讲解十六：python之描述符\_\_set\_\_和\_\_get\_\_ 等解释**

2016年11月30日 16:03:56

阅读数：2309

注：每周一到周五都会进行相关[**Python**](http://lib.csdn.net/base/11)基础知识更新，欢迎大家提宝贵的意见

一、方法：

首先说下python中存在的几种方法：对象方法、静态方法、类方法等，归属权分别为obj、cls、cls

其实可以从他们的参数中就可以看的出来

对象方法参数中含有self,这个类似于C++中的this指针。

静态方法使用@staticmethod来修饰,可以通过类或类的实例对象来调用而已.

1. **>>> class** Parent:
3. class\_attr = "class\_attr" #######类似于C++中类成员、方法，所有对象是共享内存,对象和类都是可以进行调用的。
4. **def** \_\_init\_\_(self, value):
5. self.obj\_attr = value
6. **print** self.obj\_attr
8. @staticmethod   #####静态方法
9. **def** static\_fn(str):
10. **print** "static\_fnis call...say %s" % str

13. @classmethod   #####类方法
14. **def** class\_fn(cls, str):
15. **print** "class\_fnis call...say %s" % str
17. **def** obj\_fn(self,str):   #####对象方法
18. **print** "obj\_fn is call...say %s" % str
19. >>> p=Parent('obj\_attr')
20. obj\_attr
21. >>> Parent.\_\_dict\_\_   
    {'obj\_fn': <function obj\_fn at 0x02C44B70>, '\_\_module\_\_': '\_\_main\_\_', 'class\_attr': 'class\_attr', 'class\_fn': <classmethod object at 0x02CB40F0>, 'static\_fn': <staticmethod object at 0x02C36F50>, '\_\_doc\_\_': None, '\_\_init\_\_': <function \_\_init\_\_ at 0x02C44B30>} #####可以看到类中所包含的成员   
    >>> p.\_\_dict\_\_  
    {'obj\_attr': 'obj\_attr'} #####可以看到对象中所包含的成员，类成员对象并没有在对象的dict中
22. >>> id(p.class\_attr)  
    46362560 #####在看用对象调用类成员的地址输出
23. >>> id(Parent.class\_attr)  
    46362560 #####在看用类调用类成员的地址输出，神奇的发现居然地址是一样的。那就是验证了上面所
24. 说的对象其实都是使用类的对象的地址。所有对象在没有修改自己类对象成员的时候都是指向类的初始地址。
25. >>> p.class\_attr = "change\_attr" #####我们使用对象修改了类成员属性，然后查看下具体的情况和地址变化  
    >>> p.class\_attr  
    'change\_attr' #####对象属性变化了，为何那？？？原因在于进行此操作的时候，会再对象中添加属于对象
26. 的成员change\_attr，而且是用新的内存地址，于类的相互不影响  
    >>> Parent.class\_attr  
    'class\_attr' #####类属性无变化了  
    >>> id(p.class\_attr)  
    46212960 #####内存地址变化了   
    >>> id(Parent.class\_attr)  
    46362560
27. >>> p.\_\_dict\_\_  
    {'class\_attr': 'change\_attr', 'obj\_attr': 'obj\_attr'}#####成员新增了：change\_attr
28. >>> Parent.class\_attr = "p\_change\_attr"  
    >>> p.class\_attr  
    'change\_attr' #####可见当对象的成员发生了地址新的变化后，修改类成员引用的地址，不会影响对象的成员。如果
29. 对象此时没有自己的类成员空间，那么类修改类成员空间，则对象的类成员也会发生地址空间变化。

属性可以分为两类，一类是Python自动产生的，如\_\_class\_\_，\_\_hash\_\_等，另一类是我们自定义的，如上面的hello，name。我们只关心自定义属性。  
类和实例对象(实际上，Python中一切都是对象，类是type的实例)都有\_\_dict\_\_属性，里面存放它们的自定义属性(对与类，里面还存放了别的东西)。

有些内建类型，如list和string，它们没有\_\_dict\_\_属性，所以没办法在它们上面附加自定义属性。

对于class\_attr 属性Parent.class\_attr 和Parent.\_\_dict\_\_['class\_attr']是完全一样的。因为查找的地方就是\_\_dict\_\_字典中。默认\_\_dict\_\_省略了。没啥神奇的~~~

**Python代码  [收藏代码](http://onlypython.group.iteye.com/group/wiki/1362-python-39-s-descriptor)**

1. >>> Parent.class\_attr
2. 'p\_change\_attr'
3. >>> Parent.\_\_dict\_\_['class\_attr']
4. 'p\_change\_attr'
5. >>>

但是对于obj\_fn，情形就有些不同了

**Python代码  [收藏代码](http://onlypython.group.iteye.com/group/wiki/1362-python-39-s-descriptor)**

1. >>> Parent.obj\_fn
2. <unbound method Parent.obj\_fn>
3. >>> Parent.\_\_dict\_\_['obj\_fn']
4. <function obj\_fn at 0x00C3AD70>
5. >>>

可以发现，Parent.obj\_fn是个unbound method。而Parent.\_\_dict\_\_['obj\_fn']是个函数(不是方法)。

推断：方法在类的\_\_dict\_\_中是以函数的形式存在的(方法的定义和函数的定义简直一样，除了要把第一个参数设为self)。那么Parent.obj\_fn针对的是对象的调用。

1. >>> p.obj\_fn  
   <bound method Parent.obj\_fn of <\_\_main\_\_.Parent instance at 0x02C2EE90>>

是一个bound method。

关于unbound和bound到还好理解，我们不妨先作如下设想：方法是要从实例调用的嘛(指实例方法，classmethod和staticmethod后面讲)，如果从类中访问，如Parent.obj\_fn，obj\_fn没有和任何实例发生联系，也就是没绑定(unbound)到任何实例上，所以是个unbound，对

1. p.obj\_fn

的访问方式，obj\_fn和p发生了联系，因此是bound。

一切的魔法都源自今天的主角：descriptor

查找属性时，如p.class\_attr，如果Python发现这个属性class\_attr有个\_\_get\_\_方法，Python会调用class\_attr的\_\_get\_\_方法，返回\_\_get\_\_方法的返回值，而不是返回class\_attr(这一句话并不准确，我只是希望你能对descriptor有个初步的概念)。

Python中iterator(怎么扯到Iterator了？)是实现了iterator协议的对象，也就是说它实现了下面两个方法\_\_iter\_\_和next()。类似的，descriptor也是实现了某些特定方法的对象。descriptor的特定方法是\_\_get\_\_,\_\_set\_\_和\_\_delete\_\_，其中\_\_set\_\_和\_\_delete\_\_方法是可选的。iterator必须依附某个对象而存在(由对象的\_\_iter\_\_方法返回)，descriptor也必须依附对象，作为对象的一个属性，它而不能单独存在。还有一点，descriptor必须存在于类的\_\_dict\_\_中，这句话的意思是只有在类的\_\_dict\_\_中找到属性，Python才会去看看它有没有\_\_get\_\_等方法，对一个在实例的\_\_dict\_\_中找到的属性，Python根本不理会它有没有\_\_get\_\_等方法，直接返回属性本身。descriptor到底是什么呢：简单的说，descriptor是对象的一个属性，只不过它存在于类的\_\_dict\_\_中并且有特殊方法\_\_get\_\_(可能还有\_\_set\_\_和\_\_delete)而具有一点特别的功能，为了方便指代这样的属性，我们给它起了个名字叫descriptor属性。

**Python代码  [收藏代码](http://onlypython.group.iteye.com/group/wiki/1362-python-39-s-descriptor)**

1. **class** Descriptor(object):
2. **def** \_\_get\_\_(self, obj, type=None):
3. **return** 'get', self, obj, type
4. **def** \_\_set\_\_(self, obj, val):
5. **print** 'set', self, obj, val
6. **def** \_\_delete\_\_(self, obj):
7. **print** 'delete', self, obj

这里\_\_set\_\_和\_\_delete\_\_其实可以不出现，不过为了后面的说明，暂时把它们全写上。

下面解释一下三个方法的参数：

self当然不用说，指的是当前Descriptor的实例。obj值拥有属性的对象。这应该不难理解，前面已经说了，descriptor是对象的稍微有点特殊的属性，这里的obj就是拥有它的对象，要注意的是，如果是直接用类访问descriptor(别嫌啰嗦，descriptor是个属性，直接用类访问descriptor就是直接用类访问类的属性)，obj的值是None。type是obj的类型，刚才说过，如果直接通过类访问descriptor，obj是None，此时type就是类本身。

三个方法的意义，假设T是一个类，t是它的一个实例，d是T的一个descriptor属性(牛什么啊，不就是有个\_\_get\_\_方法吗！)，value是一个有效值：

读取属性时，如T.d,返回的是d.\_\_get\_\_(None, T)的结果，t.d返回的是d.\_\_get\_\_(t, T)的结果。

设置属性时，t.d = value，实际上调用d.\_\_set\_\_(t, value)，T.d = value，这是真正的赋值，T.d的值从此变成value。删除属性和设置属性类似。

是时候坦白**真正详细的属性查找策略** 了，对于obj.attr（注意：obj可以是一个类）：

1.如果attr是一个Python自动产生的属性，找到！(优先级非常高！)

2.查找obj.\_\_class\_\_.\_\_dict\_\_，如果attr存在并且是data descriptor，返回data descriptor的\_\_get\_\_方法的结果，如果没有继续在obj.\_\_class\_\_的父类以及祖先类中寻找data descriptor

3.在obj.\_\_dict\_\_中查找，这一步分两种情况，第一种情况是obj是一个普通实例，找到就直接返回，找不到进行下一步。第二种情况是obj是一个类，依次在obj和它的父类、祖先类的\_\_dict\_\_中查找，如果找到一个descriptor就返回descriptor的\_\_get\_\_方法的结果，否则直接返回attr。如果没有找到，进行下一步。

4.在obj.\_\_class\_\_.\_\_dict\_\_中查找，如果找到了一个descriptor(插一句：这里的descriptor一定是non-data descriptor，如果它是data descriptor，第二步就找到它了)descriptor的\_\_get\_\_方法的结果。如果找到一个普通属性，直接返回属性值。如果没找到，进行下一步。

5.很不幸，Python终于受不了。在这一步，它raise AttributeError

首先，Python判断name属性是否是个自动产生的属性，如果是自动产生的属性，就按特别的方法找到这个属性，当然，这里的name不是自动产生的属性，而是我们自己定义的，Python于是到t的\_\_dict\_\_中寻找。还是没找到。

接着，Python找到了t所属的类T，搜索T.\_\_dict\_\_，期望找到name，很幸运，直接找到了，于是返回name的值：字符串‘name’。如果在T.\_\_dict\_\_中还没有找到，Python会接着到T的父类(如果T有父类的话)的\_\_dict\_\_中继续查找。

**总结：**

**只要记住类中的成员变量是对象的时候，对象又是个修饰器的时候。那么对成员变量的赋值实际是调用他的get和set方法就O了。参数很简单。**