python中基于descriptor的一些概念（下）

3. Descriptor介绍

3.1 Descriptor代码示例

```
class RevealAccess(object):
    """创建一个Descriptor类，用来打印出访问它的操作信息
    """
    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print 'Retrieving', self.name
        return self.val

    def __set__(self, obj, val):
        print 'Updating' , self.name
        self.val = val
#使用Descriptor
 class MyClass(object):
       #生成一个Descriptor实例，赋值给类MyClass的x属性
       x = RevealAccess(10, 'var "x"')
       y = 5 #普通类属性
```

运行结果：

```
>>> a = MyClass()
>>> a.x
Retrieving var "x"
10
>>> a.x = 1
Updating var "x"
>>> a.x
Retrieving var "x"
1
>>> a.y
5
>>> MyClass.x
Retrieving var "x"
1
```

3.2 定义

descriptor可以说是一个绑定了特定访问方法的**类属性**，这些访问方法是重写了descriptor protocol中的三个方法，分别是__get__, __set__, __del__方法。如果三个中任一一个方法在对象中定义了，就说这个对象是一个descriptor对象，可以把这个对象赋值给其它属性。descriptor protocol可以看成是一个有三个方法的接口。

通常对一个实例的属性的访问操作，如get, set, delete是通过实例的__dict__字典属性进行的，例如，对于操作a.x，会一个查找链从a.__dict['x']（实例的字典），再到type(a).__dict__['x']（类的字典），再到type(a)的父类的字典等等。代码如下：

```
>>> class A(object):
...     def __init__(self):
...         self.x = 1
...
>>> a = A()
>>> dir(A)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribut
e__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_
_weakref__']
>>> dir(a)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribut
e__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_
_weakref__', 'x']
>>> type(a)
<class '__main__.A'>
>>> dir(type(a))
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribut
e__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_
_weakref__']
```

可以看出类和实例的字典属性的值的内容其它是不一样的，因为实例中有绑定属性的存在。type(a)返回就是实例a的类型，类A。

如果这个需要被查找的属性是一个定义了descriptor协议方法的对象，那么python就不会按照默认的查找方式，而是调用descriptor协议中定义的方法去做处理。descriptor只对新式类和新式实例有效。

3.3 Descriptor Protocol（协议）

有下面这三个方法
get__(self, obj, type=None) --> value
set__(self, obj, value) --> None
delete__(self, obj) --> None

只要对象重写任何上面的一个方法，对象就被看作是descriptor，就可以不去采用默认的查找属性的顺序。

如果一个对象同时定义了__get__,__set__方法，被看作是data descriptor；只定义了__get__，被称为non-data descriptor。如果实例字典中有一个key和data descriptor同名，那么查找时优先采用data descriptor；如果实例字典中有一个key和non-data descriptor同名，那么优先采用实例字典的方法。

创建一个只读data descriptor，只需要在同时定义__get__,__set__方法的同时，让__set__方法抛出异常AttributeError。

3.4 Descriptor调用方法

可以直接使用descriptor实例进行方法调用，如d.__get__(obj)，但我这样尝试会报错。。。

```
>>> a.x.__get__(a)
Retrieving var "x"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute '__get__'
```

一般是在属性访问的时候自动被调用，例如obj.d是在obj实例的字典属性中查找d变量，如果d定义了__get__方法和__set__方法，是一个data descriptor，则根据上面提到的优先级，会自动去调用 d.__get__(obj)。

对于实例来说，对于任意的属性访问实现的内部机制是使用object.__getatrribute__，把b.x转化为type(b).__dict__['x'].__get__(b, type(b))，实现的优先级是按data descriptor > instance variables > non-data descriptor > __getattr__(如果定义了的话)。

对于类来说，是使用type.__getattribute__，把B.x转化为B.__dict__['x'].__get__(None, B)。

用纯python语言来描述的类属性的访问的话，大概是这个样子：

```
def __getattribute__(self, key):
    "模拟type.__getattribute__()实现"
    #通过默认方式查找到目标
    v = object.__getattribute__(self, key)
    #如果目标属性含有__get__方法，则表示它是一个descriptor
    if hasattr(v, '__get__'):
       #优先使用descriptor中定义的方法返回值
       return v.__get__(None, self)
    #如果不是descriptor，就按默认的方式返回
    return v
```

需要注意的几点：

- descriptor是被__getattribute__方法调用的。
- 重写__getattribute__方法，会阻止自动的descriptor调用，必要时需要你自己加上去。
- __getattribute__方法只在新式类和新式实例中有用。
- object.__getattribute__和class.__getattribute__会用不一样的方式调用__get__
- data descriptors总是覆盖instance dictionary

- non-data descriptors有可能被instance dictionary覆盖

使用super()返回的对象也有一个__getattribute__方法来调用descriptor。对于super(B, obj).m()
是在obj.__class__.__mro__(类属性__mro__)查找路径中找类B的基类A，然后再调用
A.__dict__['m'].__get__(obj, A)。
如果m不是descriptor，则直接返回；如果不在A的字典里，则会使用object.__getattribute__进行查找。

可以看到，descriptor实现的细节被定义在了object, type和super()的__getattribute__方法中。新式类从object继承了
这一特性，或者也可以通过元类的实现去完成类似的用法，同样的，类定义时也可以通过重写__getattribute__方法来关闭
descirptor的调用。

4. 基于Descriptor实现的功能

descriptor协议是简单而又强大的，新式类中的一些新特性就是利用descriptor功能封装成一个独立的函数调用，如:

- Property
- 绑定和非绑定方法
- 静态方法
- 类方法
- super

4.1 property

调用proprety()是一种创建data descriptor的一种简洁的方式，函数结构如下：
property(fget=None, fset=None, fdel=None, doc=None) #返回的是property对象，可以赋值给某属性，
propety方法有四个参数，只要对没有进行赋值的参数进行访问就会报错。

x 是 C 的一个实例, attrib是C中定义的一个property属性：
当你引用 x.attrib 时, python调用 fget 方法取值给你.
当你为x.attrib赋值: x.attrib=value 时, python调用 fset方法, 并且value值做为fset方法的参数,
当你执行del x.attrib 时, python调用fdel方法,
当你传过去的名为 doc 的参数即为该属性的文档字符串.

用法如下：

```
class C(object):
    def getX(self):
        print 'get x'
        return self.__x
    def setX(self, value):
        print 'set x', value
        self.__x = value
    def delX(self):
        print 'del x'
        del self.__x
    x = property(getX, setX, delX, "This is 'x' property.")
```

运行结果如下：

```
>>> c = C()
>>> c.x = 1
set x 1
>>> c.x
get x
1
>>> del c.x
del x
>>> C.x
<property object at 0x0229FA50>
>>>
```

非常方便地就改变了默认的访问属性x的方式。又如，我们定义一个只读property属性：

```
class Rect(object):
    def __init__(self, width, heigth):
        self.width = width
        self.heigth = heigth
    def getArea(self):
        return self.width * self.heigth
    area = property(getArea, doc='area of the rectangle')
```

只需要传入fget参数就可以，运行如下：

```
>>> a = Rect(2, 3)
>>> a.area
6
>>> a.area = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> del a.area
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't delete attribute
>>>
```

属性area为只读，任何重新绑定和删除的操作都会报错。这是因为我们只定义了fget方法。

properties所做的事情与那些特殊方法__getattr__, __setattr__, __delattr__ 等是极其相似的,
不过同样的工作它干起来更简单更快捷. 区别在于: 在经典类中，当你想要改变属性的访问方式时，
只能重载__getattr__，__setattr__方法，不过这样会对所有的属性访问方式进行改动；而使用
property方法就可以在不影响其它属性的前提下，任意地对某个属性的访问方式进行改动，这样做
更加灵活。

如果要用python语言来描述property功能实现的话，可以把property对象定义为这样一个descritptor是：

```
class Property(object):
    "模拟在Objects/descrobject.c文件中的PyProperty_Type()函数"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
```

```
            raise AttributeError, "unreadable attribute"
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError, "can't set attribute"
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError, "can't delete attribute"
        self.fdel(obj)
```

关于property()方法的几点声明：
1. 它不适用于经典类，但你在经典类中使用的时候，也不会报错，表面上好像OK,但实际上是不会调用参数中你设置的访问函数的。比如，当你设置一个新的属性时，经典类只是传统的在__dict__上加上了它，而不去调用fset函数进行设置。也许你可以在__setattr__函数中修复这一问题，但代价太高。

2. property()函数的四个参数，应该为methods（带self参数的那种），而不是function.

3. 当你使用类去访问属性的时候，property设置的函数是不会被调用的。只有用实例去访问才会调用。

4.2 函数和方法，绑定与非绑定

Python的面向对象特性是基于函数的，函数的实现是需要使用到non-data descriptor的功能。

类字典把方法存放为函数。在类定义中，方法是由def或者lambda声明的。和一般函数不同的是，方法的第一个参数是self对象。

为了支持方法的调用，在访问方法属性的时候，functions使用相应的__get__方法。这就意味着所有的函数都是non-data
descriptor，用于根据类或者对象的调用来返回unbound或者bound的方法。用python语言可以这样描述：

```
class Function(object):
    . . .
    def __get__(self, obj, objtype=None):
        "模拟Objects/funcobject.c文件中的func_descr_get()"
        return types.MethodType(self, obj, objtype)
```

可以在解释器中运行一下：

```
>>> class D(object):
...     def f(self, x):
...         return x
...
>>> d = D()
>>> D.__dict__['f']
<function f at 0x02164A30>
>>> D.f
<unbound method D.f>
>>> d.f
<bound method D.f of <__main__.D object at 0x021B1070>>
>>> D.f(d, 3)
3
>>> d.f(3)
3
>>> D.f(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unbound method f() must be called with D instance as first
got int instance instead)
```

可以看到方法在字典中存放的类型其实是函数对象，bound和unbound方法是两个不同的类型。
内部的实现其实是一个同一个对象，不同的是这个对象的im_self属性是否被赋值，或者是设为None。

4.3 super

在支持多继承的语言中，讨论谁是父类，感觉意义不大，尤其是像之类mro的菱形问题，父类是谁就更说不清了。需要强调的是super不会返回父类，它返回的是代理对象。理对象就是利用委托（delegation）使用别的对象的方法来实现功能的对象。

super返回的是一个定制了__getattribute__方法的对象，是一个代理对象，它可以访问MRO中的方法。形式如下：
super(cls, instance-or-subclass).method(*args, **kw)
可以转化为：
right-method-in-the-MRO-applied-to(instance-or-subclass, *args, **kw)
需要注意的是，第二个参数instnce-or-subclass可以是第一个参数的实例。

```
>>> class B(object):
...     def __repr__(self):
...         return '<instance of %s>' % self.__class__.__name__
...
>>> class C(B):
...     pass
...
>>> class D(C):
...     pass
...
>>> d = D()
>>> print super(C, d).__repr__
<bound method D.__repr__ of <instance of D>>
>>> print super(C, D).__repr__
<unbound method D.__repr__>
>>> print super(C, d).__repr__()
<instance of D>
>>> print super(C, D).__repr__()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unbound method __repr__() must be called with D instance as first arg
ument (got nothing instead)
```

如果返回了非绑定的方法，调用的时候需要加上第一个self参数。

通过descriptor的实现，可以说super也是一个non-data descriptor类。也就是实现了__get__(self, obj, objtyp=None)的类。
假设descr是C类的一个descriptor，C.descr实现上调用的是descr.__get__(None, C)；
如果是实例来调用，c.descr调用的是descr.__get__(c, type(c))。

super功能用python语言来描述的话，可以是这样：

```
class Super(object):
    def __init__(self, type, obj=None):
        self.__type__ = type
        self.__obj__ = obj
    def __get__(self, obj, type=None):
        if self.__obj__ is None and obj is not None:
            return Super(self.__type__, obj)
        else:
            return self
    def __getattr__(self, attr):
        if isinstance(self.__obj__, self.__type__):
            starttype = self.__obj__.__class__
        else:
            starttype = self.__obj__
        mro = iter(starttype.__mro__)
        for cls in mro:
            if cls is self.__type__:
                break
        # Note: mro is an iterator, so the second loop
        # picks up where the first one left off!
        for cls in mro:
            if attr in cls.__dict__:
                x = cls.__dict__[attr]
                if hasattr(x, "__get__"):
                    x = x.__get__(self.__obj__)
                return x
        raise AttributeError, attr
```

5. 结尾

在这里，就介绍完了基于descriptor的新式类的新特性。欢迎大家讨论。

通过 为知笔记 发布

分类: python

标签: python

posted @ 2012-09-18 16:18 btchenguang 阅读(5780) 评论(3) 编辑 收藏