python中基于descriptor的一些概念（上）

python中基于descriptor的一些概念（上）

1. 前言

python在2.2版本中引入了descriptor功能，也正是基于这个功能实现了新式类(new-styel class)的对象模型，同时解决了之前版本中经典类(classic class)系统中出现的多重继承中的MRO(Method Resolution Order)的问题，同时引入了一些新的概念，比如classmethod, staticmethod, super,Property等，这些新功能都是基于descriptor而实现的。总而言之，通过学习descriptor可以更多地了解python的运行机制。我在这也大概写一个汇总，写一下对这些东西的理解。欢迎大家讨论。

在这里，为文章中使用的词汇做一下说明：
函数：指的是第一个参数不是self的函数，不在类中定义的函数
方法：指是的第一个参数是self的函数
实例：类的对象，instance
对象模型：就是实现对象行为的整个框架，这里分为经典和新的两种

使用的python版本为python 2.7.2

2. 新式类与经典类

首先来了解一下新式类与经典类的区别，从创建方法上可以明显的看出：

```
#新式类
class C(object):
    pass
#经典类
class B:
    pass
```

简单的说，新式类是在创建的时候继承内置object对象（或者是从内置类型，如list,dict等），而经典类是直接声明的。使用dir()方法也可以看出新式类中定义很多新的属性和方法，而经典类好像就2个：

```
>>> class C(object):
...     pass
...
>>> class B:
...     pass
...
>>> dir(C)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribut
e__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_
_weakref__']
>>> dir(B)
['__doc__', '__module__']
>>>
```

这些新的属性和方法都是从object对象中继承过来的。

2.1 内置的object对象

内置的object对象是所有内置，object对象定义了一系列特殊的方法实现所有对象的默认行为。
1. __new__，__init__方法
这两个方法是用来创建object的子类对象，静态方法__new__()用来创建类的实例，然后再调用__init__()来初始化实例。

2. __delattr__, __getattribute__, __setattr__方法
对象使用这些方法来处理属性的访问

3. __hash__, __repr__, __str__方法
print(someobj)会调用someobj.__str__()， 如果__str__没有定义，则会调用someobj.__repr__()，

__str__()和__repr__()的区别：

- 默认的实现是没有任何作用的
- __repr__的目标是对象信息唯一性
- __str__的目标是对象信息的可读性
- 容器对象的__str__一般使用的是对象元素的__repr__
- 如果重新定义了__repr__，而没有定义__str__，则默认调用__str__时，调用的是__repr__
- 也就是说好的编程习惯是每一个类都需要重写一个__repr__方法，用于提供对象的可读信息，
- 而重写__str__方法是可选的。实现__str__方法，一般是需要更加好看的打印效果，比如你要制作
- 一个报表的时候等。

可以允许object的子类重载这些方法，或者添加新的方法。

2.2 类的方法

新的对象模型中提供了两种类级别的方法，静态方法和类方法，在诸多新式类的特性中，也只有类方法这个特性， 和经典对象模型实现的功能一样。

2.2.1 静态方法

静态方法可以被类或者实例调用，它没有常规方法的行为(比如绑定，非绑定，默认的第一个self参数)，当有一

堆函数仅仅是为了一个类写的时候，采用静态方法声明在类的内部，可以提供行为上的一致性。

创建静态方法的代码如下，使用装饰符@staticmethod进行创建：

```
>>> class A(object):
...     @staticmethod
...     def foo():
...         pass
...     def bar(self):
...         pass
...
>>> a = A()
>>> a.foo
<function foo at 0x021F49B0>
>>> A.foo
<function foo at 0x021F49B0>
>>> a.bar
<bound method A.bar of <__main__.A object at 0x0223BD70>>
>>> A.bar
<unbound method A.bar>
>>>
```

可以看出，不管是 类调用，还是实例调用静态方法，都是指向同一个函数对象

2.2.2 类方法

也是可以通过类和它的实例进行调用，不过它是有默认第一个参数，叫做是类对象，一般被命名为cls，当然你也可以命名为其它名字，这样就你可以调用类对象的一些操作，
代码如下，使用装饰符@classmethod创建：

```
>>> class A(object):
...     @classmethod
...     def foo(cls):
...         print 'class name is', cls.__name__
...     @classmethod
...     def bar(mycls):
...         print 'class name is', mycls.__name__
...
>>> a = A()
>>> a.foo
<bound method type.foo of <class '__main__.A'>>
>>> a.bar
<bound method type.bar of <class '__main__.A'>>
>>> A.foo
<bound method type.foo of <class '__main__.A'>>
>>> A.bar
<bound method type.bar of <class '__main__.A'>>
>>> a.foo()
class name is A
>>> A.foo()
class name is A
>>> a.bar()
class name is A
>>> A.bar()
class name is A
```

2.3 新式类(new-style class)

新式类除了拥有经典类的全部特性之外，还有一些新的特性。比如__init__发生了变化，
新增了静态方法__new__

2.3.1 __init__方法

据说在python2.4版本以前，使用新式类时，如果类的初始化方法没有定义，调用的
时候写了多余的参数，编译器不会报错。我现在的python 2.7会报错，还是觉得会报错
比较好点，下面给出新式类和经典类运行这个例子的情况：

```
>>> class A:
...     pass
...
>>> a = A('abc')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: this constructor takes no arguments
>>> class A(object):
...     pass
...
>>> a = A('abc')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object.__new__() takes no parameters
```

2.3.2 __new__静态方法

新式类都有一个__new__的静态方法，它的原型是object.__new__(cls[, ...])
cls是一个类对象，当你调用C(*args, **kargs)来创建一个类C的实例时，python的内部调用是C.__new__(C, *args, **kargs)，然后返回值是类C的实例c，在确认c是C的实例后，python再调用C.__init__(c, *args, **kargs)来初始化实例c。
所以调用一个实例c = C(2)，实际执行的代码为：

```
c = C.__new__(C, 2)
if isinstance(c, C):
    C.__init__(c, 23)#__init__第一个参数要为实例对象
```

object.__new__()创建的是一个新的，没有经过初始化的实例。当你重写__new__方法时，可以不用使用装饰符@staticmethod指明它是静态函数，解释器会自动判断这个方法为静态方法。如果需要重新绑定C.__new__方法时，只要在类外面执行C.__new__ = staticmethod(yourfunc)就可以了。

可以使用__new__来实现Singleton单例模式：

```
class Singleton(object):
    _singletons = {}
    def __new__(cls):
        if not cls._singletons.has_key(cls):            #若还没有任何实例
            cls._singletons[cls] = object.__new__(cls)  #生成一个实例
        return cls._singletons[cls]                             #返回这个实例
```

运行结果如下：

```
>>> class Singleton(object):
...     _singleton = {}
...     def __new__(cls):
...         if not cls._singleton.has_key(cls):
...             cls._singleton[cls] = object.__new__(cls)
...         return cls._singleton[cls]
...
>>> a = Singleton()
>>> id(a)
35918416
>>> b = Singleton()
>>> id(b)
35918416
```

使用id()操作，可以看到两个实例指向同一个内存地址。Singleton的所有子类也有这一特性，只有一个实例对象，如果它的子类定义了__init__()方法，那么必须保证它的__init__方法能够安全的同一个实例进行多次调用。

```
>>> class B(Singleton):
...     pass
...
>>> b = B()
>>> id(b)
35918576
>>> c = B()
>>> id(c)
35918576
```

2.4. 新式类的实例

除了新式类本身具有新的特性外，新式类的实例也具有新的特性。比如它拥有Property功能，该功能会对属性的访问方式产生影响；还有__slots__新属性，该属性会对生成子类实例产生影响；还添加了一个新的方法__getattribute__，比原有的__getattr__更加通用。

2.4.1 Property

在介绍完descriptor会回过头来讲这个。

2.4.2 __slots__属性

通常每一个实例x都会有一个__dict__属性，用来记录实例中所有的属性和方法，也是通过这个字典，可以让实例绑定任意的属性。而__slots__属性作用就是，当类C有比较少的变量，而且拥有__slots__属性时，类C的实例 就没有__dict__属性，而是把变量的值存在一个固定的地方。如果试图访问一个__slots__中没有的属性，实例就会报错。这样操作有什么好处呢？__slots__属性虽然令实例失去了绑定任意属性的便利，但是因为每一个实例没有__dict__属性，却能有效节省每一个实例的内存消耗，有利于生成小而精干的实例。

为什么需要这样的设计呢？
在一个实际的企业级应用中，当一个类生成上百万个实例时，即使一个实例节省几十个字节都可以节省一大笔内存，这种情况就值得使用__slots__属性。

怎么去定义__slots__属性？
__slots__是一个类变量，__slots__属性可以赋值一个包含类属性名的字符串元组，或者是可迭代变量，或者是一个字符串，只要在类定义的时候，使用__slots=aTuple来定义该属性就可以了：

```
>>> class A(object):
...     def __init__(self):
...         self.x = 1
...         self.y = 2
...     __slots__ = 'x', 'y'
...
>>> a = A()
>>> dir(a)
['__class__', '__delattr__', '__doc__', '__format__', '__getattribute__', '__has
h__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__rep
r__', '__setattr__', '__sizeof__', '__slots__', '__str__', '__subclasshook__', '
x', 'y']
>>> a.z = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'z'
>>> a.u = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'A' object has no attribute 'u'
```

可以看出实例a中没有__dict__字典，而且不能随意添加新的属性，不定义__slots__是可以随意添加的：

```
>>> class C(object):
...     pass
...
>>> c = C()
>>> c.x = 1
>>> dir(c)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribut
e__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_
_weakref__', 'x']
```

使用时__slots__时需要注意的几点：
1. 当一个类的父类没有定义__slots__属性，父类中的__dict__属性总是可以访问到的，所以只在子类中定义__slots__属性，而不在父类中定义是没有意义的。

2. 如果定义了__slots属性，还是想在之后添加新的变量，就需要把'__dict__'字符串添加到__slots__的元组里。

3. 定义了__slots__属性，还会消失的一个属性是__weakref__，这样就不支持实例的weak reference，如果还是想用这个功能，同样，可以把'__weakref__'字符串添加到元组里。

4. __slots__功能是通过descriptor实现的，会为每一个变量创建一个descriptor。

5. __slots__的功能只影响定义它的类，因此，子类需要重新定义__slots__才能有它的功能。

2.4.3 __getattribute__方法

对新式类的实例来说，所有属性和方法的访问操作都是通过__getattribute__完成，这是由object基类实现的。如果有特殊的要求，可以重载__getattribute__方法，下面实现一个不能使用append方法的list：

```
>>> class listNoAppend(list):
...     def __getattribute__(self, name):
...         if name == 'append':
...             raise AttributeError, name
...         return list.__getattribute__(self, name)
...
>>> a = listNoAppend()
>>> type(a)
<class '__main__.listNoAppend'>
>>> dir(a)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delsli
ce__', '__dict__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute_
_', '__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
 '__init__', '__iter__', '__le__', '__len__', '__lt__', '__module__', '__mul__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
 '__rmul__', '__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str_
_', '__subclasshook__', '__weakref__', 'append', 'count', 'extend', 'index', 'in
sert', 'pop', 'remove', 'reverse', 'sort']
>>> a.append()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in __getattribute__
AttributeError: append
```

2.4.4 实例的方法

经典的与新的对象模型都允许一个实例拥有私有的属性和方法（可以通过绑定和重绑定）。实例的私有属性会覆盖掉类中定义的同名属性，举例说明：

```
>>> class A(object):
...     def __init__(self):
...         self.x = 2
...     def foo(self):
...         print 'it is foo'
...
>>> a = A()
>>> a.x
2
>>> a.x = 3
>>> a.x
3
>>> def bar():
...         print 'it is bar'
...
>>> a.foo = bar
>>> a.foo()
it is bar
```

然而在python中，隐式调用实例的私有特殊方法时，新的对象模型和经典对象模型表现上不太一样。
在经典对象模型中，无论是显示调用还是隐式调用特殊方法，都会调用实例中后绑定的特殊方法。
而在新的对象模型中，除非显式地调用实例的特殊方法，否则python总是会去调用类中定义的特殊方法，
如果没有定义的话，就报错。代码如下：
经典类：

```
>>> def fakeGetItem(index):
...     return index + 1
...
>>> class Classic:
...     pass
...
>>> c = Classic()
>>> c.__getitem__ = fakeGetItem
>>> print c[1]
2
>>>
```

新式类：

```
>>> class NewStyle(object):
...     pass
...
>>> a = NewStyle()
>>> a.__getitem__ = fakeGetItem
>>> print a[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'NewStyle' object does not support indexing
>>> print a.__getitem__(1)
2
>>>
```

调用a[1]，将产生一个隐式的__getitem__方法的调用，在新式类中，因为类中没有定义这个方法，也不是object基类有的方法，所以报错。需要显示地调用才可以运行。

2.5 新的对象模型

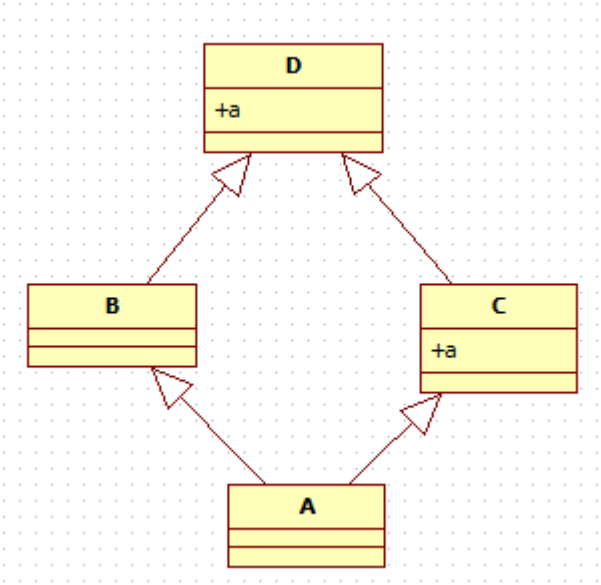
在新的对象模型中，继承方式和经典对象模型大体相同，一个关键的区别就是新式类能够从python的内置类型中继承，而经典类不行。

2.5.1 多继承

新式类同样支持多继承，但是如果新式类想要从多个内置类型中继承生成一个新类的话，则这些内置类必须是经过精心设计，能够互相兼容的。显然，python也没会让你随意的从多个内置类中进行多继承，想创建一个超级类不是那么容易的。。。通常情况下，至多可以继承一个内置类，比如list, set, dict等。

2.5.2 MRO(Method Resolution Order， 方法解析顺序)

对于下图的多继承关系：



b = A()，当调用b.a的时候会发生什么事呢?
在经典对象模型中，方法和属性的查找链是按照从左到右，深度优先的方式进行查找。所以当A的实例b要使用属性a时，它的查找顺序为:A->B->D->C->A，这样做就会忽略类C的定义a，而先找到的基类D的属性a，这是一个bug，这个问题在新式类中得到修复，新的对象模型采用的是从左到右，广度优先的方式进行查找，所以查找顺序为A->B->C->D，可以正确的返回类C的属性a。
经典类:

```
>>> class D:
...     def __init__(self):
...         self.a = 1
...
>>> class B(D):
...     pass
...
>>> class C(D):
...     def __init__(self):
...         self.a = 2
...
>>> class A(B, C):
...     pass
...
>>> a = A()
>>> a.a
1
```

新式类:

```
>>> class D(object):
...     def __init__(self):
...         self.a = 1
...
>>> class B(D):
...     pass
...
>>> class C(D):
...     def __init__(self):
...         self.a = 2
...
>>> class A(B, C):
...     pass
...
>>> a = A()
>>> a.a
2
>>>
```

这个顺序的实现是通过新式类中特殊的只读属性__mro__，类型是一个元组，保存着解析顺序信息。只能通过类来使用，不能通过实例调用。

```
>>> A.__mro__
(<class '__main__.A'>, <class '__main__.B'>, <class '__main__.C'>, <class '__mai
n__.D'>, <type 'object'>)
>>>
```

顺序还和继承时，括号中写的父类顺序有关：

```
>>> class A(C, B):
...     pass
...
>>> A.__mro__
(<class '__main__.A'>, <class '__main__.C'>, <class '__main__.B'>, <class '__mai
n__.D'>, <type 'object'>)
>>>
```

2.5.3 协作式调用父类方法

当子类重写了父类的一个方法时，通常会调用父类的同名方法做一些工作，这是比较常见的使用方式--使用非绑定语法来调用父类的方法。不过在多继承中，这种方法有缺陷：

```
>>> class A(object):
...     def foo(self):
...         print "A's foo"
...
>>> class B(A):
...     def foo(self):
...         print "B's foo"
...         A.foo(self)
...
>>> class C(A):
...     def foo(self):
...         print "C's foo"
...         A.foo(self)
...
>>> class D(B, C):
...     def foo(self):
...         print "D's foo"
...         B.foo(self)
...         C.foo(self)
...
>>> d = D()
>>> d.foo()
D's foo
B's foo
A's foo
C's foo
A's foo
>>>
```

可以看到，基类A的方法重复运行了两次。怎样才能确保父类中的方法只被顺序的调用一次呢？在新的对象系统中，有一种特殊的方法super(aclass, obj)，可以返回obj实例的一个特殊类型superobject(超对象， 不是简单的父类的对象)，当我们使用超对象调用父类的方法时，就能保证只被运行一次：

```
>>> class A(object):
...     def foo(self):
...         print "A's foo"
...
>>> class B(A):
...     def foo(self):
...         print "B's foo"
...         super(B, self).foo()
...
>>> class C(A):
...     def foo(self):
...         print "C's foo"
...         super(C, self).foo()
...
>>> class D(B, C):
...     def foo(self):
...         print "D's foo"
...         super(D, self).foo()
...
>>> d = D()
>>> d.foo()
D's foo
B's foo
C's foo
A's foo
```

可以看到，D的父类中所有的foo方法都得到执行，并且基类A的foo方法只执行了一次。如果养成了使用super去调用父类方法的习惯，那么你的类就可以适应无论多么复杂的继承调用结构。super()可以看成是更加安全调用父类方法的一种新方式。

通过 为知笔记 发布

分类: python

标签: python