魏忠的Space发布 ::-- WeiZhong [2009-08-25 10:46:38] ::-- ZoomQuiet [2006-01-09 06:12:38]

1. python中的 new-style class 及其实例详解

(原文见《Python In a Nutshell(2003)》5.2节)

1.1. 5.2 new-style Class 及其实例

前面我提到 python 2.2 中引入了 new-style 对象模型. new-style class 及其实例与 Classic class 一样, 都是顶级对象。 它们可以拥有任意的属性, 通过调用类对象生成该类的一个实例对象. 在这一小节, 我来向大家揭示新的对象模型及传统对象模型的不同.

从 python 2.2 起，如果一个类继承自 object 对象(或者它是任何内建类型如 list, dict, file 的子类)，那它就是一个 new-style class 。在此之前，Python 不允许通过继承内建类型生成新类，也根本没有 object 这个对象。在本章5.4节的后半部分，我会介绍给大家一个方法，将 Classic class 改造成 new-style class 。

我建议每个人， 从现在开始只使用 new-style class (当然你得用 Python2.2 以上版本)。新的对象模型与传统对象模型相比， 有虽小却非常重要的优势, 可以说接近完美.

1.2. 5.2.1 内建的 object 对象

object 对象是所有内建类型及 new-style class 的祖先。 object 对象定义了一系列特殊方法(参见 5.3 节后半部分)实现所有对象的默认行为。

__new__, __init__ 方法

你可以创建 object 的直接子类，静态方法 __new__()用来创建类的实例， 实例的 __init__() 方法用来初始化自己。 默认的 __init__() 方法会忽略你传递过来的任何参数。

__delattr__, __getattribute__, __setattr__ 方法

对象用这些方法来处理属性引用。本章前半部分已经做了详细介绍。

__hash__, __repr__, __str__ 方法

print(someobj) 会调用 someobj.__str__ 如果 __str__ 没有定义， 则调用 __repr__ repr(someobj) 会调用 someobj.__repr__

允许object的子类重载这些方法，或添加新方法。

1.3. 5.2.2 类方法

新的对象模型提供了两种类方法(传统对象模型没有这些方法)：静态方法和类方法。只有 python2.2 及更新版本才支持类方法. 需要提一下的是,在 python2.2 及更新版本中， Classic class 也实现了类方法。新的对象模型提供的诸多新特性中，有且仅有类方法这一特性被传统对象模型全功能实现。

1.3.1. 5.2.2.1静态方法

静态方法可以直接被类或类实例调用。它没有常规方法那样的特殊行为（绑定、非绑定、默认的第一个参数规则等等）。完全可以将静态方法当成一个用属性引用方式调用的普通函数来看待。任何时候定义静态方法都不是必须的（静态方法能实现的功能都可以通过定义一个普通函数来实现）. 有些程序员认为，当有一堆函数仅仅为某一特定类编写时，采用类方法这种方式能够提供足够的一致性（和一定程度的 namespace 的功能)。

根据python2.4提供的新的语法，你可以象下面这样来创建一个静态方法，

切换行号显示

```
1 class AClass(object):
2     @staticmethod       #静态方法修饰符，表示下面的方法是一个静态方法
3     def astatic( ): print 'a static method'
4 anInstance = AClass( )
5 AClass.astatic( )                   # prints: a static method
6 anInstance.astatic( )               # prints: a static method
```

注:staticmethod是一个内建函数, 用来将一个方法包装成静态方法, 在2.4以前版本, 要用下面的方式定义一个静态方法(不再推荐使用):

切换行号显示

```
class AClass(object):
    def astatic(  ): print 'a static method'
    astatic=staticmethod(astatic)
```

这种方法在函数定义本身比较长时经常会忘记后面这一行.

1.3.2. 5.2.2.2 类方法

一个类方法就是你可以通过类或它的实例来调用的方法, 不管你是用类调用这个方法还是类的实例调用这个方法,python只会将实际的类对象做为该方法的第一个参数.记住:方法的第一个参数都是类对象而不是实例对象. 按照惯例,类方法的第一个形参被命名为 cls. 任何时候定义类方法都不是必须的（静态方法能实现的功能都可以通过定义一个普通函数来实现,只要这个函数接受一个类对象做为参数就可以了）.某些程序员认为这个特性当有一堆函数仅仅为某一特定类编写时会提供使用上的一致性.

定义类方法:

切换行号显示

```
class ABase(object):
    @classmethod        #类方法修饰符
    def aclassmet(cls): print 'a class method for', cls.__name__
class ADeriv(ABase): pass
bInstance = ABase(  )
dInstance = ADeriv(  )
ABase.aclassmet(  )              # prints: a class method for ABase
bInstance.aclassmet(  )          # prints: a class method for ABase
ADeriv.aclassmet(  )             # prints: a class method for ADeriv
dInstance.aclassmet(  )          # prints: a class method for ADeriv
```

注:classmethod是一个内建函数,用来将一个方法封装成类方法,在2.4以前版本,你只能用下面的方式定义一个类方法:

切换行号显示

```
class AClass(object):
    def aclassmethod(cls): print 'a class method'
    aclassmethod=staticmethod(aclassmethod)
```

并没有人要求必须封装后的方法名字必须与封装前一致,但建议你总是这样做(如果你使用python2.4版本以下时). 这种方法在函数定义本身比较长时经常会忘记后面这一行.

1.4. 5.2.3 new-style class

除了拥有 Classic class 的全部特性之外, new-style class 当然还具有一些新特性.__init__特殊方法的行为与 Classic class 相比有了一些变化, 另外还新增了一个名为 __new__ 的静态方法

1.4.1. 5.2.3.1 __init__方法

下面的 C 类(一个 new-style class)中, 从 object 继承来的原始 __init__方法, 可以认为就是一个 pass 语句, 因为它几乎什么都不做, 建议你在所有的 new-style class 中重新实现 __init__ 方法.

切换行号显示

```
class C(object):
    def __init__(self): pass
    # rest of class body omitted
```

示例中的的类只允许无参数调用,硬要传递一个参数给它会产生异常(如用C('xyz')). 如果C没有重载__init__方法, 调用C('xyz')会象 'xyz' 根本不存在一样忽略参数继续执行. 注意: (根据我的试验,2.4版中这点发生了变化,即使没有重载__init__方法,象C('xyz')这样调用一样会引发异常)

1.4.2. 5.2.3.2 __new__方法

每一个 new-style class 都有一个名为__new__的静态方法. 当你调用 C(*args,**kwds)创建一个C实例时,python内部调用的是C.__new__(C,*args,**kwds).

__new__方法的返回值 x 就是该类的实例. 在确认 x 是C的实例以后, python调用C.__init__(x,*args,**kwds)来初始化这个实例. 也就是说,对新类C来讲,语句 x=C(23)等同于:

切换行号显示

```
x = C.__new__(C, 23)
if isinstance(x, C): C.__init__(x, 23)
```

object.__new__创建一个新的,未初始化的类实例,它接收传递过来的第一个参数(也就是类对象本身),忽略其它的参数. 当你重载__new__方法时,你不必使用函数修饰符@staticmethod, python解释器根据上下文会认出__new__()方法是一个静态方法. 如果你需要重绑定 C.__new__方法,你只需要在类外面执行 C.__new__=staticmethod(你想使用的新方法)就可以了.(极少有这样的需求)

__new__方法拥有函数工厂的绝大部分弹性. 根据实际需求,我们可以让__new__返回一个已有的实例或者创建一个新的实例.
下面举一个通过重载__new__方法实现独身对象的设计模式的例子:

切换行号显示

```
1 class Singleton(object):
2     _singletons = {}
3     def __new__(cls, *args, **kwds):
4         if not cls._singletons.has_key(cls):            #若还没有任何实例
5             cls._singletons[cls] = object.__new__(cls)  #生成一个实例
6         return cls._singletons[cls]                           #返回这个实例
```

Singleton的所有子类(当然是没有重载__new__方法的子类)都只可能有一个实例. 如果该类的子类定义了一个__init__方法,那么它必须保证它的__init__方法能够安全的对同一实例进行多次调用.

1.5. 5.2.4 new-style class 实例

> new-style class 实例除了拥有 Classic class 实例的全部特性之外,还拥有一种称为property的新属性及一个叫作__slots__的特殊属性,该属性会对实例其它属性的访问产生重要影响.

新的对象模型同样添加了一个新的方法 __getattribute__ 比原有的 __getattr__ 方法更通用. 不同的实例可以拥有这些特殊方法的不同实现.

1.5.1. 5.2.4.1 Properties

property 是实例中具有特殊功能的属性. 你可以使用常规语法对property进行引用,绑定或解除绑定.如:

切换行号显示

```
1 print x.prop
2 x.prop=23
3 del x.prop
```

然而,property如果只有这点功能那就和普通属性没什么两样了,property有它的独到之处,请往下读. 下面介绍如何定义一个只读property:

切换行号显示

```
1 class Rectangle(object):
2     def __init__(self, width, heigth):
3         self.width = width
4         self.heigth = heigth
5     def getArea(self):
6         return self.width * self.heigth
7     area = property(getArea, doc='area of the rectangle')
```

矩形类的每一个实例 r 均拥有一个只读属性 r.area, 该属性由 r.getArea()方法实时计算得来. Rectangle.area.__doc__是'area of the rectangle', 这个属性是只读的(试图对它进行重绑定或解除绑定的企图都注定会失败), 这是因为我们在property定义中指定了该属性的 get 方法.

properties 干的活与那些特殊方法__getattr__, __setattr__, __delattr__ 等是极其相似的, 不过同样的活它干起来更简单更快捷. 内建 property 类别(我倒宁愿把当成一个函数来看)用来生成一个 property, 并将其返回值绑定为一个类属性. 如同绑定类的常规属性, 一般在定义类时就创建property, 当然也有其它选择. 假设在定义 new-style class C 时, 使用以下语法:

attrib = property(fget=None, fset=None, fdel=None, doc=None)

x 是 C 的一个实例, 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, 你传过去的名为 doc 的参数即为该属性的文档字符串. 在矩形类中,因为我们没有为area属性指定 fset 和 fdel 参数, 所以该属性只能读取.

1.5.2. 5.2.4.2 __slots__属性

通常, 每个实例对象 x 都拥有一个字典 x.__dict__. python通过此字典允许你绑定任意属性给 x 实例. 定义一个名为 __slots__ 的类属性可以有效减少每个实例占用的内存数量. __slots__ 是一个字符串序列(通常是一个tuple). 当类 C 拥有 __slots__属性, x 的直接子类就没有 x.__dict__属性. 如果试图绑定一个 __slots__ 中不存在属性给实例的话, 就会引发异常. __slots__属性虽然令你失去绑定任意属性的方便, 却能有效节省每个实例的内存消耗, 有助于生成小而精干的实例对象.
注: 当一个类会生成很多很多实例时(有些类同时拥有数百万而不是几千个实例), 即使一个实例节省几十个字节都可节省一大笔内存时, 就值得使用__slots__属性. 只有在类定义中可以使用 __slots__ = aTuple 语句来为一个类添加__slots__属性, 其它任何位置对一个类或其父类的__slots__属性的修改, 重新绑定或解除绑定都是无效的.
下面介绍如何通过添加 __slots__ 属性给刚才定义的 Rectangle 类, 以得到瘦身的类实例:

切换行号显示

```
1 class OptimizedRectangle(Rectangle):
```

```
__slots__ = 'width', 'heigth'
```

__slots__里不能包含 properties, 只能包含常规实例属性. 我们不需也不允许给area property 也定义一个slot. 若不定义 __slots__属性, 常规属性则保存在实例的__dict__属性中.

__slot__ 只是用来占位，因此对于__slot__ 定义的属性名，你首先要赋值，然后才可以使用。直接使用是会报错的。 -- Limodou

1.5.3. 5.2.4.3 __getattribute__方法

对 new-style class 的实例来说, 所有的属性引用都是通过特殊方法 __getattribute__() 完成的. 该方法由基类对象提供, 负责实现对象属性引用的全部细节. 在本章的前面有该方法详细的文档. 如果有特殊需求,你也可以重载 __getattribute__ 属性(比如你打算在子类实例中隐藏父类的某些属性或方法).下面的例子演示了实现一个没有 append 方法的 list 类:

切换行号显示

```
class listNoAppend(list):
    def __getattribute__(self, name):
        if name == 'append': raise AttributeError, name
        return list.__getattribute__(self, name)
```

除了功能不全以外,该类的实例与内建list对象完全相同. 任何调用该类实例append方法的企图都会引发一个异常.

1.5.4. 5.2.4.4个体实例方法

传统与新的对象模型都允许一个实例拥有私有的属性和方法(通过绑定或重绑定) . 实例的私有属性会屏蔽掉类定义中的同名属性.举例来说:

切换行号显示

```
class abc(object):
        def attrib_a(self):
                print 'aMethod defined in class abc'
b = abc()
def afunc():
        print 'hello,world!'
b.attrib_a=afunc
b.attrib_a()
```

该例子将打印 'hello,world!'

在 python 隐式调用实例的私有(后绑定)特殊方法时, 新的对象模型的行为与传统对象模型不同. 在传统对象模型中, 无论是显式调用, 还是隐式调用, 都会调用这个实例的后绑定特殊方法. 而在新的对象模型中, 除非显式调用实例的特殊方法, 否则python总是去调用在类中定义的特殊方法. 下面这个例子可以说明这一点:

切换行号显示

```
def fakeGetItem(idx): return idx
class Classic: pass
c = Classic( )
c.__getitem__ = fakeGetItem
print c[23]                     # prints: 23

class NewStyle(object): pass
n = NewStyle( )
n.__getitem__ = fakeGetItem
print n[23]                     # 程序执行到这步会出错. 如果将代码改为 print n.__getitem__(23) 则正常运行

# Traceback (most recent call last):
#   File "<stdin>", line 1, in ?
# TypeError: unindexable object
```

调用n[23],将产生一个隐式的__getitem__方法调用,因为 new-style class 对象 n 中并未定义该方法, 所以引发了异常. 不过如果你使用 n.__getitem__(23)这种方式来显式调用特殊方法时, 它还是可以工作的.

1.6. 5.2.5 新的对象模型中的继承

在新的对象模型中, 继承的使用方式与传统模型大致相同. 一个关键的区别就是 new-style class 能从一个内建类型中继承而 Classic class 不能.

> new-style class 仍然支持多继承, 若要从多个内建类型继承生成一个新类, 则这些内建类型必须是经过特殊设计能够相互兼容. python不支持随意的从多个内建类型进行多继承, 通常情况都是通过至多从一个内建类型继承得到新类. 这意味着在多继承时, 除object以外, 至多有一个内建类型可以是其它内建类型和 new-style class 的超类.

1.6.1. 5.2.5.1方法解析顺序:

在传统对象模型中, 方法和属性按 从左至右 深度优先 的顺序查找. 显然, 当多个父类继承自同一个基类时, 这会产生我们不想要的结果. 举例来说, A 是 B 和 C 的子类, 而 B 和 C 继承自 D,传统对象模型的的属性查找方法是 A - B - D - C - D. 由于Python先查找 D 后查找 C, 即使 C 对 D 中的方法进行了重定义, 也只能使用 D 中定义的版本. 由于这个继承模式固有的问题, 在实际应用中会造成一些麻烦.

在新的对象模型中,所有类均直接或间接生成子类对象. python改变了传统对象模型中的解析顺序, 使用上面的例子, 当 D 是一个 new-style class (比如 D 是 object 的直接子类), 新的对象模型的搜索顺序就变为 A - B - C - D.

每个内建类型及 new-style class 均内建一个特殊的只读属性 __mro__ , 这是一个tuple, 保存着方法解析类型. 只允许通过类来引用 __mro__ (不允许通过实例).

1.6.2. 5.2.5.2 协作式调用超类方法

前面我们提到, 当一个子类重载了父类中一个方法, 子类中的方法通常要调用父类中的同名方法来做一些事. 这也是python传统对象模型惯用的方式, 即使用非绑定方法语法调用父类的同名方法. 当多继承时, 这种方法是有缺陷的, 见下例:

切换行号显示

```
 1 class A(object):
 2     def met(self):
 3         print 'A.met'
 4 class B(A):
 5     def met(self):
 6         print 'B.met'
 7         A.met(self)
 8 class C(A):
 9     def met(self):
10         print 'C.met'
11         A.met(self)
12 class D(B,C):
13     def met(self):
14         print 'D.met'
15         B.met(self)
16         C.met(self)
```

在上面的代码中, 当我们调用 D().met()方法时, A.met()方法被调用了两次. 我们怎样才可以保证每个父类的实现均被顺序调用且仅仅调用一次呢? 不采取点特殊措施这个问题很难解决. 从 python2.2 起, 提供了这样一个特殊手段. 那就是 super类型. super(aclass,obj) 返回对象 obj 的一个特殊的超对象 (superobject). 当我们调用该超对象的一个属性或方法时, 就保证了每个父类的实现均被调用且仅仅调用一次了. 改写后的代码如下:

切换行号显示

```
 1 class A(object):
 2     def met(self):
 3         print 'A.met'
 4 class B(A):
 5     def met(self):
 6         print 'B.met'
 7         super(B,self).met( )
 8 class C(A):
 9     def met(self):
10         print 'C.met'
11         super(C,self).met( )
12 class D(B,C):
13     def met(self):
14         print 'D.met'
15         super(D,self).met( )
```

现在就可以得到期望的结果了. 如果你养成了总是使用superclass调用父类方法,你的类就能适应无论多复杂的继承结构.

PyNewStyleClass (2010-11-04 08:40:13由WeiZhong编辑)