8.10 使用延迟计算属性¶

问题¶

你想将一个只读属性定义成一个property，并且只在访问的时候才会计算结果。 但是一旦被访问后，你希望结果值被缓存起来，不用每次都去计算。

解决方案¶

定义一个延迟属性的一种高效方法是通过使用一个描述器类，如下所示：

```
class lazyproperty:
    def __init__(self, func):
        self.func = func

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            value = self.func(instance)
            setattr(instance, self.func.__name__, value)
            return value
```

你需要像下面这样在一个类中使用它：

```
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @lazyproperty
    def area(self):
        print('Computing area')
        return math.pi * self.radius ** 2

    @lazyproperty
    def perimeter(self):
        print('Computing perimeter')
        return 2 * math.pi * self.radius
```

下面在一个交互环境中演示它的使用：

```
>>> c = Circle(4.0)
>>> c.radius
4.0
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.perimeter
Computing perimeter
25.132741228718345
>>> c.perimeter
```

```
>>> c.perimeter
25.132741228718345
>>>
```

仔细观察你会发现消息 `Computing area` 和 `Computing perimeter` 仅仅出现一次。

讨论¶

很多时候，构造一个延迟计算属性的主要目的是为了提升性能。 例如，你可以避免计算这些属性值，除非你真的需要它们。 这里演示的方案就是用来实现这样的效果的， 只不过它是通过以非常高效的方式使用描述器的一个精妙特性来达到这种效果的。

正如在其他小节(如8.9小节)所讲的那样，当一个描述器被放入一个类的定义时， 每次访问属性时它的 `__get__()`、`__set__()` 和 `__delete__()` 方法就会被触发。 不过，如果一个描述器仅仅只定义了一个 `__get__()` 方法的话，它比通常的具有更弱的绑定。 特别地，只有当被访问属性不在实例底层的字典中时 `__get__()` 方法才会被触发。

`lazyproperty` 类利用这一点，使用 `__get__()` 方法在实例中存储计算出来的值， 这个实例使用相同的名字作为它的property。 这样一来，结果值被存储在实例字典中并且以后就不需要再去计算这个property了。 你可以尝试更深入的例子来观察结果：

```
>>> c = Circle(4.0)
>>> # Get instance variables
>>> vars(c)
{'radius': 4.0}

>>> # Compute area and observe variables afterward
>>> c.area
Computing area
50.26548245743669
>>> vars(c)
{'area': 50.26548245743669, 'radius': 4.0}

>>> # Notice access doesn't invoke property anymore
>>> c.area
50.26548245743669

>>> # Delete the variable and see property trigger again
>>> del c.area
>>> vars(c)
{'radius': 4.0}
>>> c.area
Computing area
50.26548245743669
>>>
```

这种方案有一个小缺陷就是计算出的值被创建后是可以被修改的。例如：

```
>>> c.area
Computing area
50.26548245743669
>>> c.area = 25
>>> c.area
25
>>>
```

如果你担心这个问题，那么可以使用一种稍微没那么高效的实现，就像下面这样：

```
def lazyproperty(func):
    name = '_lazy_' + func.__name__
    @property
    def lazy(self):
        if hasattr(self, name):
            return getattr(self, name)
        else:
            value = func(self)
            setattr(self, name, value)
            return value
    return lazy
```

如果你使用这个版本，就会发现现在修改操作已经不被允许了：

```
>>> c = Circle(4.0)
>>> c.area
Computing area
50.26548245743669
>>> c.area
50.26548245743669
>>> c.area = 25
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

然而，这种方案有一个缺点就是所有get操作都必须被定向到属性的 `getter` 函数上去。 这个跟之前简单的在实例字典中查找值的方案相比效率要低一点。 如果想获取更多关于property和可管理属性的信息，可以参考8.6小节。而描述器的相关内容可以在8.9小节找到。