描述符及一些其他

描述符是对类中的属性进行管理的一种方法. 描述符是实现了特定协议的类，只要一个类实现了 `__get__` , `__set__` , `__delete__` 方法中的任何一个，就可以将其称之为描述符，因为他已经具备了部分属性管理的功能. 大部分描述符只实现了 `__get__` 和 `__set__` 方法. 描述符实例是需要被管理的类的一个属性. 下面用几个例子总结描述符运行的逻辑.

```
class Descriptor:
    def __init__(self, name):
        self.name = name

    def __set__(self, instance, value):
        print("现在调用了描述符的__set__方法！")
        instance.__dict__[self.name] = value


class A:
    quantity = Descriptor("weight")

    def __init__(self, value):
        self.quantity = value


a = A(10)
print(a.__dict__, a.weight, a.quantity, sep="\n")
```

```
现在调用了描述符的__set__方法!
{'weight': 10}
10
<__main__.Descriptor object at 0x0000022261B1E6A0>
```

上面这个运行的顺序是这样的， `quantity` 是类A的一个属性，它是Descriptor类的一个实例，它的name属性的值是weight; `A(10)` 运行了 `__init__` 方法，在 `__init__` 函数中， `self.quantity` 实际上是通过实例a读取类A中的quantity对象，描述符quantity发现了正在对它进行赋值，自动调用了quantity的 `__set__` 方法， `__set__` 方法的第一个参数是描述符实例，在这里就是quantity实例，第二个参数是调用他的对象实例，在这里就是实例a，第三个参数是尝试赋给它的值，在这里就是10. 于是，在这里就给实例a一个新的属性，weight，新属性的值是10.

一些注意

赋值和读取在python中是不等价的，这是理解描述符的关键，看下面的一个例子：

```
class B:
    x = 1


b = B()
print(B.x, b.x, b.__dict__)
b.x = 2
print(B.x, b.x, b.__dict__)
```

```
1 1 {}
1 2 {'x': 2}
```

当实例b没有x属性时，他会自动找到类中的x属性，因此第一个 `b.x` 实际上就是 `B.x` ；但是在尝试为实例的x属性赋值时，python发现这个实例没有x属性，会生成一个新的实例属性x给b，并赋值为2，在这一过程中并不会改变类中的x的值

上面的例子中描述符的 `__set__` 方法为对象生成了一个名字与描述符不同的属性，如果名字相同会发生什么情况呢？

```
class Descriptor:
    def __init__(self, name):
        self.name = name

    def __set__(self, instance, value):
        print("现在调用了描述符的__set__方法！")
        instance.__dict__[self.name] = value


class A:
    weight = Descriptor("weight")

    def __init__(self, value):
        self.weight = value


a = A(10)
print(a.__dict__, a.weight, sep="\n")
```

```
现在调用了描述符的__set__方法！
{'weight': 10}
10
```

现在这个属性的关系是这样的，类A有一个描述符叫weight，实例a又有一个属性叫weight，在尝试读取 a.weight 时，不会返回类A中的描述符，而是会返回a的weight属性的值. 但是在尝试改变 a.weight 的值时，又会调用描述符的 __set__ 方法，理解这一点的关键就是python赋值和读取的不等价性.

```
a.weight = 100
print(a.__dict__, a.weight, sep="\n")
```

```
现在调用了描述符的__set__方法!
{'weight': 100}
100
```

__get__ 方法有什么用呢..其实从上面的讨论就可以看出， __set__ 方法实际上就相当于实现了为类中属性赋值的托管， __get__ 方法就是实现了属性读取的托管~

```
class Descriptor:
    def __get__(self, instance, owner):
        print("现在调用了描述符的__get__方法! ")
        return "不管发生了什么我都会返回这句话.."

class A:
    weight = Descriptor()

a = A()
a.weight
```

```
现在调用了描述符的__get__方法!
```

```
'不管发生了什么我都会返回这句话..'
```

__get__ 方法的第一个参数是描述符实例，第二个是它所在类的实例，第三个是他所在的类，在这里就分别是weight,a,A. 但是单独使用 __get__ 方法时，假如一旦给实例一个与描述符同名的属性时，就会把这个同名描述符属性给覆盖掉：

```
class Descriptor:
    def __get__(self, instance, owner):
        print("现在调用了描述符的__get__方法! ")
```

```
        return "不管发生了什么我都会返回这句话.."


class A:
    weight = Descriptor()

    def __init__(self, value):
        self.weight = value


a = A(100)
a.weight
```

```
100
```

同时实现 `__set__` 方法与 `__get__` 方法时，通过二者的配合，可以实现属性的完全托管. 下面这个例子摘自fluent python，情形是这样的， `LineItem` 是商品类，该类的每一个对象都有 `weight` 和 `price` 两个值，代表不同的商品参数，Quantity是描述符类，实现对这两个量的管理.

管理逻辑是这样的，这两个量分别对应着两个描述符实例，如果像本文最上面的描述符那样实现，用name属性来区分不同的描述符实例，则会很麻烦，每实例化一个描述符对象时，就要传给它一个名字. 这里巧妙地通过Quantity类属性 `__counter` 来区分，每实例化一个描述符时，它的值就+1，从而实现了对不同描述符中name值的区分. 每一个描述符实例都具有一个独一无二的 `storage_name`

这里通过 `__set__` 方法和 `__get__` ，使得不需要直接访问这个 `sotrage_name` ，就可以直接获得对象的这两个值.. 相当于对象的 `storrage_name` 被隐藏了，唯一暴露的接口就是描述符方法..

```
class Quantity:
    __counter = 0

    def __init__(self):
        cls = self.__class__
        prefix = cls.__name__
        index = cls.__counter
        self.storage_name = "_{}#{}".format(prefix, index)
        cls.__counter += 1

    def __get__(self, instance, owner):
        return getattr(instance, self.storage_name)

    def __set__(self, instance, value):
        if value > 0:
            setattr(instance, self.storage_name, value)
```

```
            else:
                raise ValueError("value must be > 0")


    class LineItem:
        weight = Quantity()
        price = Quantity()

        def __init__(self, weight, price):
            self.weight = weight
            self.price = price

        def subtotal(self):
            return self.weight * self.price


    apple = LineItem(1, 5)


    print(apple.__dict__, apple.weight, apple.price, sep="\n")
```

```
{'_Quantity#0': 1, '_Quantity#1': 5}
1
5
```

```
    apple.weight = 10
    apple.price = 100
    print(apple.__dict__, apple.weight, apple.price, sep="\n")
```

```
{'_Quantity#0': 10, '_Quantity#1': 100}
10
100
```

这里生成了一个 `apple` 对象， `apple` 对象有两个属性， `_Quantity#0` 和 `_Quantity#1` ，这两个属性是不可见的（这也是为什么要设置成这个名字） `weight` 和 `price` 都是描述符，在通过 `apple` 对象给 `weight` 和 `price` 赋值时，自动调用描述符的 `__set__` 方法，检查数值的同时，给 `apple` 对象赋予 `_Quantity#0` 和 `_Quantity#1` 两个属性；在读取 `weight` 和 `price` 的值时，由于 `apple` 对象没有对应的属性，于是自动调用了 `__get__` 方法，返回了 `_Quantity#0` 和 `_Quantity#1` 的值. 实现了闭环.

其他

1. python中的函数实现了 `__get__` 方法

```
def func():
    pass
func.__get__
```

```
<method-wrapper '__get__' of function object at 0x0000022261B5AAF0>
```

2. 在类中定义的函数属于绑定方法

```
class C:
    def func(self):
        pass


print(C.func)
print(C().func)
```

```
<function C.func at 0x0000022261B5A3A0>
<bound method C.func of <__main__.C object at 0x0000022261B785B0>>
```

从类中直接读取func对象，得到的是函数，从类的实例中读取func对象，得到的是绑定方法对象. 绑定方法对象的意思就是，通过对象来读取这个函数时，默认把对象传给了这个函数的第一个参数，返回了一个所谓的绑定方法对象. 实际上下面这两个的过程是一样的：

```
c = C()
print(c.func())
print(C.func(c))
```

```
None
None
```

那么仔细考察上面这个过程的逻辑，实际上就是实现了属性读取的托管. 也就是：

3. 类中定义的函数是一个描述符

通过类直接访问函数时，函数的 `__get__` 方法返回自身的引用；通过实例访问时，函数的 `__get__` 方法返回的是绑定方法对象（可以通过instance参数的值是否为 `None` 来区分.）