9.9 将装饰器定义为类¶

问题¶

你想使用一个装饰器去包装函数，但是希望返回一个可调用的实例。 你需要让你的装饰器可以同时工作在类定义的内部和外部。

解决方案¶

为了将装饰器定义成一个实例，你需要确保它实现了 `__call__()` 和 `__get__()` 方法。 例如，下面的代码定义了一个类，它在其他函数上放置一个简单的记录层：

```
import types
from functools import wraps

class Profiled:
    def __init__(self, func):
        wraps(func)(self)
        self.ncalls = 0

    def __call__(self, *args, **kwargs):
        self.ncalls += 1
        return self.__wrapped__(*args, **kwargs)

    def __get__(self, instance, cls):
        if instance is None:
            return self
        else:
            return types.MethodType(self, instance)
```

你可以将它当做一个普通的装饰器来使用，在类里面或外面都可以：

```
@Profiled
def add(x, y):
    return x + y

class Spam:
    @Profiled
    def bar(self, x):
        print(self, x)
```

在交互环境中的使用示例：

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls
2
>>> s = Spam()
>>> s.bar(1)
<__main__.Spam object at 0x10069e9d0> 1
>>> s.bar(2)
<__main__.Spam object at 0x10069e9d0> 2
>>> s.bar(3)
<__main__.Spam object at 0x10069e9d0> 3
>>> Spam.bar.ncalls
3
```

讨论¶

将装饰器定义成类通常是很简单的。但是这里还是有一些细节需要解释下，特别是当你想将它作用在实例方法上的时候。

首先，使用 `functools.wraps()` 函数的作用跟之前还是一样，将被包装函数的元信息复制到可调用实例中去。

其次，通常很容易会忽视上面的 `__get__()` 方法。如果你忽略它，保持其他代码不变再次运行， 你会发现当你去调用被装饰实例方法时出现很奇怪的问题。例如：

```
>>> s = Spam()
>>> s.bar(3)
Traceback (most recent call last):
...
TypeError: bar() missing 1 required positional argument: 'x'
```

出错原因是当方法函数在一个类中被查找时，它们的 `__get__()` 方法依据描述器协议被调用， 在8.9小节已经讲述过描述器协议了。在这里， `__get__()` 的目的是创建一个绑定方法对象 (最终会给这个方法传递self参数)。下面是一个例子来演示底层原理：

```
>>> s = Spam()
>>> def grok(self, x):
...     pass
...
>>> grok.__get__(s, Spam)
<bound method Spam.grok of <__main__.Spam object at 0x100671e90>>
>>>
```

`__get__()` 方法是为了确保绑定方法对象能被正确的创建。 `type.MethodType()` 手动创建一个绑定方法来使用。只有当实例被使用的时候绑定方法才会被创建。 如果这个方法是在类上面来访问， 那么 `__get__()` 中的instance参数会被设置成None并直接返回 `Profiled` 实例本身。 这样的话我们就可以提取它的 `ncalls` 属性了。

如果你想避免一些混乱，也可以考虑另外一个使用闭包和 `nonlocal` 变量实现的装饰器，这个在9.5小节有讲到。例如：

```
import types
from functools import wraps

def profiled(func):
    ncalls = 0
    @wraps(func)
    def wrapper(*args, **kwargs):
        nonlocal ncalls
        ncalls += 1
        return func(*args, **kwargs)
    wrapper.ncalls = lambda: ncalls
    return wrapper

# Example
@profiled
def add(x, y):
    return x + y
```

这个方式跟之前的效果几乎一样，除了对于 `ncalls` 的访问现在是通过一个被绑定为属性的函数来实现，例如：

```
>>> add(2, 3)
5
>>> add(4, 5)
9
>>> add.ncalls()
2
>>>
```