9.10 为类和静态方法提供装饰器¶

问题¶

你想给类或静态方法提供装饰器。

解决方案¶

给类或静态方法提供装饰器是很简单的，不过要确保装饰器在 `@classmethod` 或 `@staticmethod` 之前。例如：

```
import time
from functools import wraps

# A simple decorator
def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        r = func(*args, **kwargs)
        end = time.time()
        print(end-start)
        return r
    return wrapper

# Class illustrating application of the decorator to different kinds of methods
class Spam:
    @timethis
    def instance_method(self, n):
        print(self, n)
        while n > 0:
            n -= 1

    @classmethod
    @timethis
    def class_method(cls, n):
        print(cls, n)
        while n > 0:
            n -= 1

    @staticmethod
    @timethis
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1
```

装饰后的类和静态方法可正常工作，只不过增加了额外的计时功能：

```
>>> s = Spam()
>>> s.instance_method(1000000)
<__main__.Spam object at 0x1006a6050> 1000000
0.11817407608032227
>>> Spam.class_method(1000000)
<class '__main__.Spam'> 1000000
0.11334395408630371
>>> Spam.static_method(1000000)
1000000
0.11740279197692871
>>>
```

讨论¶

如果你把装饰器的顺序写错了就会出错。例如，假设你像下面这样写：

```
class Spam:
```

```
    @timethis
    @staticmethod
    def static_method(n):
        print(n)
        while n > 0:
            n -= 1
```

那么你调用这个静态方法时就会报错：

```
>>> Spam.static_method(1000000)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "timethis.py", line 6, in wrapper
start = time.time()
TypeError: 'staticmethod' object is not callable
>>>
```

问题在于 `@classmethod` 和 `@staticmethod` 实际上并不会创建可直接调用的对象， 而是创建特殊的描述器对象(参考8.9小节)。因此当你试着在其他装饰器中将它们当做函数来使用时就会出错。 确保这种装饰器出现在装饰器链中的第一个位置可以修复这个问题。

当我们在抽象基类中定义类方法和静态方法(参考8.12小节)时，这里讲到的知识就很有用了。 例如，如果你想定义一个抽象类方法，可以使用类似下面的代码：

```
from abc import ABCMeta, abstractmethod
class A(metaclass=ABCMeta):
    @classmethod
    @abstractmethod
    def method(cls):
        pass
```

在这段代码中，`@classmethod` 跟 `@abstractmethod` 两者的顺序是有讲究的，如果你调换它们的顺序就会出错。