9.5 可自定义属性的装饰器¶

问题¶

你想写一个装饰器来包装一个函数，并且允许用户提供参数在运行时控制装饰器行为。

解决方案¶

引入一个访问函数，使用 `nonlocal` 来修改内部变量。 然后这个访问函数被作为一个属性赋值给包装函数。

```
from functools import wraps, partial
import logging
# Utility decorator to attach a function as an attribute of obj
def attach_wrapper(obj, func=None):
    if func is None:
        return partial(attach_wrapper, obj)
    setattr(obj, func.__name__, func)
    return func

def logged(level, name=None, message=None):
    '''
    Add logging to a function. level is the logging
    level, name is the logger name, and message is the
    log message. If name and message aren't specified,
    they default to the function's module and name.
    '''
    def decorate(func):
        logname = name if name else func.__module__
        log = logging.getLogger(logname)
        logmsg = message if message else func.__name__

        @wraps(func)
        def wrapper(*args, **kwargs):
            log.log(level, logmsg)
            return func(*args, **kwargs)

        # Attach setter functions
        @attach_wrapper(wrapper)
        def set_level(newlevel):
            nonlocal level
            level = newlevel

        @attach_wrapper(wrapper)
        def set_message(newmsg):
            nonlocal logmsg
            logmsg = newmsg

        return wrapper

    return decorate

# Example use
@logged(logging.DEBUG)
def add(x, y):
    return x + y

@logged(logging.CRITICAL, 'example')
def spam():
    print('Spam!')
```

下面是交互环境下的使用例子：

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> add(2, 3)
DEBUG:__main__:add
```

```
5
>>> # Change the log message
>>> add.set_message('Add called')
>>> add(2, 3)
DEBUG:__main__:Add called
5
>>> # Change the log level
>>> add.set_level(logging.WARNING)
>>> add(2, 3)
WARNING:__main__:Add called
5
>>>
```

讨论¶

这一小节的关键点在于访问函数(如 `set_message()` 和 `set_level()`)，它们被作为属性赋给包装器。 每个访问函数允许使用 `nonlocal` 来修改函数内部的变量。

还有一个令人吃惊的地方是访问函数会在多层装饰器间传播(如果你的装饰器都使用了 `@functools.wraps` 注解)。 例如，假设你引入另外一个装饰器，比如9.2小节中的 `@timethis` ，像下面这样：

```
@timethis
@logged(logging.DEBUG)
def countdown(n):
    while n > 0:
        n -= 1
```

你会发现访问函数依旧有效：

```
>>> countdown(10000000)
DEBUG:__main__:countdown
countdown 0.8198461532592773
>>> countdown.set_level(logging.WARNING)
>>> countdown.set_message("Counting down to zero")
>>> countdown(10000000)
WARNING:__main__:Counting down to zero
countdown 0.8225970268249512
>>>
```

你还会发现即使装饰器像下面这样以相反的方向排放，效果也是一样的：

```
@logged(logging.DEBUG)
@timethis
def countdown(n):
    while n > 0:
        n -= 1
```

还能通过使用lambda表达式代码来让访问函数的返回不同的设定值：

```
@attach_wrapper(wrapper)
def get_level():
    return level

# Alternative
wrapper.get_level = lambda: level
```

一个比较难理解的地方就是对于访问函数的首次使用。例如，你可能会考虑另外一个方法直接访问函数的属性，如下：

```
@wraps(func)
def wrapper(*args, **kwargs):
    wrapper.log.log(wrapper.level, wrapper.logmsg)
    return func(*args, **kwargs)

# Attach adjustable attributes
wrapper.level = level
wrapper.logmsg = logmsg
```

```
wrapper.log = log
```

这个方法也可能正常工作，但前提是它必须是最外层的装饰器才行。 如果它的上面还有另外的装饰器(比如上面提到的 `@timethis` 例子)，那么它会隐藏底层属性，使得修改它们没有任何作用。 而通过使用访问函数就能避免这样的局限性。

最后提一点，这一小节的方案也可以作为9.9小节中装饰器类的另一种实现方法。