9.4 定义一个带参数的装饰器¶

问题¶

你想定义一个可以接受参数的装饰器

解决方案¶

我们用一个例子详细阐述下接受参数的处理过程。 假设你想写一个装饰器，给函数添加日志功能，同时允许用户指定日志的级别和其他的选项。 下面是这个装饰器的定义和使用示例：

```
from functools import wraps
import logging

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)
        return wrapper
    return decorate

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

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

初看起来，这种实现看上去很复杂，但是核心思想很简单。 最外层的函数 `logged()` 接受参数并将它们作用在内部的装饰器函数上面。 内层的函数 `decorate()` 接受一个函数作为参数，然后在函数上面放置一个包装器。 这里的关键点是包装器是可以使用传递给 `logged()` 的参数的。

讨论¶

定义一个接受参数的包装器看上去比较复杂主要是因为底层的调用序列。特别的，如果你有下面这个代码：

```
@decorator(x, y, z)
def func(a, b):
    pass
```

装饰器处理过程跟下面的调用是等效的;

```
def func(a, b):
    pass
func = decorator(x, y, z)(func)
```

`decorator(x, y, z)` 的返回结果必须是一个可调用对象，它接受一个函数作为参数并包装它， 可以参考9.7小节中另外一个可接受参数的包装器例子。