9.6 带可选参数的装饰器¶

问题¶

你想写一个装饰器，既可以不传参数给它，比如 `@decorator` ， 也可以传递可选参数给它，比如 `@decorator(x,y,z)` 。

解决方案¶

下面是9.5小节中日志装饰器的一个修改版本：

```
from functools import wraps, partial
import logging

def logged(func=None, *, level=logging.DEBUG, name=None, message=None):
    if func is None:
        return partial(logged, level=level, name=name, message=message)

    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

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

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

可以看到， `@logged` 装饰器可以同时不带参数或带参数。

讨论¶

这里提到的这个问题就是通常所说的编程一致性问题。 当我们使用装饰器的时候，大部分程序员习惯了要么不给它们传递任何参数，要么给它们传递确切参数。 其实从技术上来讲，我们可以定义一个所有参数都是可选的装饰器，就像下面这样：

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

但是，这种写法并不符合我们的习惯，有时候程序员忘记加上后面的括号会导致错误。 这里我们向你展示了如何以一致的编程风格来同时满足没有括号和有括号两种情况。

为了理解代码是如何工作的，你需要非常熟悉装饰器是如何作用到函数上以及它们的调用规则。 对于一个像下面这样的

为了理解代码是如何工作的，你需要先熟悉装饰器是如何作用到函数上以及它们的调用规则。对于一个像下面这样的简单装饰器：

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

这个调用序列跟下面等价：

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

add = logged(add)
```

这时候，被装饰函数会被当做第一个参数直接传递给 `logged` 装饰器。因此，`logged()` 中的第一个参数就是被包装函数本身。所有其他参数都必须有默认值。

而对于一个下面这样有参数的装饰器：

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

调用序列跟下面等价：

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

初始调用 `logged()` 函数时，被包装函数并没有传递进来。因此在装饰器内，它必须是可选的。这个反过来会迫使其他参数必须使用关键字来指定。并且，但这些参数被传递进来后，装饰器要返回一个接受一个函数参数并包装它的函数(参考9.5小节)。为了这样做，我们使用了一个技巧，就是利用 `functools.partial` 。它会返回一个未完全初始化的自身，除了被包装函数外其他参数都已经确定下来了。可以参考7.8小节获取更多 `partial()` 方法的知识。