链滴

我是装饰器

作者：somenzz
原文链接：https://ld246.com/article/1556025650722
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

在 Python 的世界里，我是一名装饰器。

函数或类都是我服务的对象，我不改变他，但我会他更强。

一些朋友在初次接触我时觉得有点复杂，看不懂，因此对我敬而远之。

其实，我并不难理解，而且学会使用我之后，可以让你写代码时偷点懒，少点重复性工作，代码也更雅，更具有 Pythonic。

为了你让更容易理解和使用我，我先自下而上讲讲我的来历，再自上而下讲讲我的用法。

小明是一个程序员，一天，他洋洋洒洒写了好几十个函数，经过测试，虽然有一些问题，但很快就被解决了，很有成就感，顺利交付上线。

过了几天，系统运行变慢了，经理要求小明找出慢在哪里。

小明不得不为每个函数增加计时功能，当初是考虑到了，但就是懒，懒得为统计每个函数的耗时多写几行代码，现在可能要一个一个改函数，把

```
def somefunc(*args, **kwargs):
    do_something()
```

改成：

```
import time
def somefunc(*args, **kwargs):
    start = time.time()
    do_something()
    end = time.time()
    print(somefunc.__name__, end-start)
```

小明不喜欢做重复的事情，一个函数一个函数的修改，也不是他的风格，另外修改了函数代码，还要

部重新测试。

于是我出现了。

自下而上，讲讲我的来历。

在 Python

世界里，万物皆对象，函数亦如此。上述的函数需要增加新功能，但又不想改变原来的代码和调用方，那么我实现一个新函数，将原来的函数做为参数传到这个新函数中，然后在新函数中实现新的功能并返回一个新函数。代码如下所示：

```
import time
def somefunc_new(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end - start)
        return result
    return wrapper
```

这里应该不难理解，新函数 somefunc_new 返回了一个函数 wrapper，wrapper 在使用的时候会在用原函数 func 并加入计时功能。

具体使用时这样子：

```
decorated_func = somefunc_new(somefunc)
somefunc = decorated_func
```

就可以了。为了让你直观的感受下，可以得到下面的代码运行下：

```
import time
def somefunc_new(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end - start)
        return result
    return wrapper

def somefunc(n):
    print("I am some func")
    time.sleep(n)

print("before decorate:")
somefunc(2)
decorated_func = somefunc_new(somefunc)
somefunc = decorated_func
print("after decorate:")
somefunc(2)
```

上述代码运行结果如下：

```
before decorate:
I am some func
after decorate:
I am some func
somefunc 2.0108282566070557
```

有了上面的新函数，我们再也不用每个函数都增加那三行计时的代码了，但仍不有方便之处就是需要每一个函数都执行重新赋值操作，还是有点麻烦。

不过设计我的人已经想好了，直接在定义函数的上面加上 @装饰器名 即可自动实现转换。即：

```
@somefunc_new
def somefunc(n):
    print("I am some func")
    time.sleep(n)
```

这样写就相当于告诉 python 解释器在执行 somefunc(n) 时这样执行：

```
decorated_func = somefunc_new(somefunc)
decorated_func(n)
```

大家可以实验下。

这样，在每个函数头前加上 @somefunc_new，就实现了自动赋值，这一步甚至可以通过查找替换批量操作，完美解决小明的问题。

现在是不是已经理解我的作用了？

还有一点我得告诉你，上述示例代码如果你执行 print(somefunc.**name**) 打印的是新函数 wrapper 名称，而不是原有的 somefunc:

我这里贴出源代码和运行结果，假如：somefunc 长这样：

```
import time

def somefunc_new(func):
    '''
    这是文档字符串
    somefunc_new
    '''
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end-start)
        return result
    return wrapper


@somefunc_new
def somefunc(n):
    '''
    这是文档字符串
    somefunc
```

```
    '''
    time.sleep(n)
    return n


somefunc(1)
print(somefunc.__name__)
print(somefunc.__doc__)
```

运行结果如下：

```
somefunc 1.0156023502349854
wrapper
None
```

注意这里 somefunc.**name** 等价于 somefunc_new(somefunc).**name**，所以返回值为 wrapper。

修正方法也很简单，需要使用标准库中提供的一个 wraps 装饰器，将被装饰函数的信息复制给 wrapp
r 函数：

```
from functools import wraps
import time

def timethis(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(func.__name__, end - start)
        return result

    return wrapper
```

这样，再次执行的结果如下：

```
somefunc 1.0040576457977295
somefunc

    这是文档字符串
    somefunc
```

至此，一个完整的，不带参数的装饰器便写好了。

自上而下，讲讲我的用法。

上面设计比较简单，不带任何参数。我们也会经常看到带参数的，其使用方法大概如下：

```
@logged('debug', name='example', message='message')
def somefunc(*args, **kwargs):
    pass
```

下面我们自上而下实现这样一个装饰器。先分解对被装饰函数 somefunc 的调用过程：

```
>>> somefunc(a, b, c='value')
# 等价于
>>> decorator = logged('debug', name='example', message='message')
>>> decorated_func = decorator(somefunc)
>>> decorated_func(a, b, c='value')
```

由此可见，logged 是一个函数，它返回一个装饰器，这个返回的装饰器再去装饰 somefunc 函数，此 logged 的模板代码应该像这样：

```
def logged(level, name=None, message=None):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            do_something()
            return func(*args, **kwargs)
        return wrapper
    return decorator
```

wrapper 是最终被调用的函数，我们可以随意丰富完善 decorator 和 wrapper 的逻辑。

假设我们的需求是被装饰函数 somefunc 被调用前打印一行 log 日志，代码如下：

```
from functools import wraps
def logged(level, name=None, message=None):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            logname = name if name else func.__module__
            logmsg = message if message else func.__name__
            print(level, logname, logmsg, sep=' - ')
            return func(*args, **kwargs)
        return wrapper
    return decorator
```

有时候，我们也会看到同一个装饰器有两种使用方法，可以像简单装饰器一样使用，也可以传递参数例如：

```
@logged
def func(*args, **kwargs):
    pass

@logged(level='debug', name='example', message='message')
def fun(*args, **kwargs):
    pass
```

不带参数的装饰器和带参数的装饰器定义是不同的。不带参数的装饰器返回的是被装饰后的函数，带数的装饰器返回的是一个不带参数的装饰器，然后这个返回的不带参数的装饰器再返回被装饰后的函。那么怎么统一呢？先来分析一下两种装饰器用法的调用过程。

```
# 使用 @logged 直接装饰
>>> func(a, b, c='value')
# 等价于
>>> decorated_func = logged(func)
>>> decorated_func(a, b, c='value')
```

```
# 使用 @logged(level='debug', name='example', message='message') 装饰
>>> func(a, b, c='value')
# 等价于
>>> decorator = logged(level='debug', name='example', message='message')
>>> decorated_func = decorator(func)
>>> decorated_func(a, b, c='value')
```

可以看到，第二种装饰器比第一种装饰器多了一步，就是调用装饰器函数再返回一个装饰器，这个返的装饰器和不带参数的装饰器是一样的：接收被装饰的函数作为唯一参数。唯一的区别是返回的装饰携带固定参数，固定函数参数正是 partial 函数的使用场景，因此我们可以定义如下的装饰器：

```
from functools import wraps, partial

def logged(func=None, *, level='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__
    logmsg = message if message else func.__name__

    @wraps(func)
    def wrapper(*args, **kwargs):
        print(level, logname, logmsg, sep=' - ')
        return func(*args, **kwargs)
    return wrapper
```

实现的关键在于，若这个装饰器以带参数的形式使用，这第一个参数 func 的值为 None，此时我们用 partial 返回了一个其它参数固定的装饰器，这个装饰器与不带参数的简装饰器一样，接收被装饰函数对象作为唯一参数，然后返回被装饰后的函数对象。

我能被继承吗？

当然可以继承，目前为止你只看到过用来构建装饰器的函数。幸运的是，类也可以用来构建装饰器。

那我们现在以一个类而不是一个函数的方式来创建一个装饰器。

```
from functools import wraps

class logit(object):
    def __init__(self, logfile='out.log'):
        self.logfile = logfile

    def __call__(self, func):
        @wraps(func)
        def wrapped_function(*args, **kwargs):
            log_string = func.__name__ + " was called"
            print(log_string)
            # 打开logfile并写入
            with open(self.logfile, 'a') as opened_file:
                # 现在将日志打到指定的文件
                opened_file.write(log_string + '\n')
            # 现在，发送一个通知
            self.notify()
            return func(*args, **kwargs)
```

```
        return wrapped_function

    def notify(self):
        # logit只打日志，不做别的
        pass
```

这个实现有一个附加优势，在于比嵌套函数的方式更加整洁，而且包裹一个函数还是使用跟以前一样语法：

```
@logit()
def myfunc1():
    pass
```

现在，我们给 logit 创建子类，来添加 email 的功能(虽然 email 这个话题不会在这里展开)

```
class email_logit(logit):
    '''
    一个logit的实现版本，可以在函数调用时发送email给管理员
    '''
    def __init__(self, email='admin@myproject.com', *args, **kwargs):
        self.email = email
        super(email_logit, self).__init__(*args, **kwargs)

    def notify(self):
        # 发送一封email到self.email
        # 这里就不做实现了
        pass
```

从现在起，@email_logit 将会和 @logit 产生同样的效果，但是在打日志的基础上，还会多发送一封件给管理员。

够强大吧?

装饰一个类怎么写?

由于类的实例化和函数调用非常类似，因此装饰器函数也可以用于装饰类，只是此时装饰器函数的第个参数不再是函数，而是类。基于自顶而下的设计方法，设计一个用于装饰类的装饰器函数就是轻而举的事情，这里给一个示例：

```
def decorator(aClass):
    class newClass:
        def __init__(self, age):
            self.total_display  = 0
            self.wrapped        = aClass(age)
        def display(self):
            self.total_display += 1
            print("total display", self.total_display)
            self.wrapped.display()
    return newClass

@decorator
class Bird:
    def __init__(self, age):
        self.age = age
```

```
    def display(self):
        print("My age is",self.age)

eagleLord = Bird(5)
for i in range(3):
    eagleLord.display()
```

请自行体验效果。

最后

润物细无声。我致力于增强函数或类，却不曾改变过它们，

(完)

如果你觉得文章对你有帮助，请关注公众号 somenzz 获取更多。