装饰器

廖雪峰

资深软件开发工程师，业余马拉松选手。

由于函数也是一个对象，而且函数对象可以被赋值给变量，所以，通过变量也能调用该函数。

```
>>> def now():
...     print('2024-6-1')
...
>>> f = now
>>> f()
2024-6-1
```

函数对象有一个 `__name__` 属性（注意：是前后各两个下划线），可以拿到函数的名字：

```
>>> now.__name__
'now'
>>> f.__name__
'now'
```

现在，假设我们要增强 `now()` 函数的功能，比如，在函数调用前后自动打印日志，但又不希望修改 `now()` 函数的定义，这种在代码运行期间动态增加功能的方式，称之为"装饰器"（Decorator）。

本质上，decorator就是一个返回函数的高阶函数。所以，我们要定义一个能打印日志的decorator，可以定义如下：

```
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

观察上面的 `log` ，因为它是一个decorator，所以接受一个函数作为参数，并返回一个函数。我们要借助Python的@语法，把decorator置于函数的定义处：

```
@log
def now():
    print('2024-6-1')
```

调用 `now()` 函数，不仅会运行 `now()` 函数本身，还会在运行 `now()` 函数前打印一行日志：

```
>>> now()
call now():
2024-6-1
```

把 `@log` 放到 `now()` 函数的定义处，相当于执行了语句：

```
now = log(now)
```

由于 `log()` 是一个decorator，返回一个函数，所以，原来的 `now()` 函数仍然存在，只是现在同名的 `now` 变量指向了新的函数，于是调用 `now()` 将执行新函数，即在 `log()` 函数中返回的 `wrapper()` 函数。

`wrapper()` 函数的参数定义是 `(*args, **kw)` ，因此， `wrapper()` 函数可以接受任意参数的调用。在 `wrapper()` 函数内，首先打印日志，再紧接着调用原始函数。

如果decorator本身需要传入参数，那就需要编写一个返回decorator的高阶函数，写出来会更复杂。比如，要自定义log的文本：

```
def log(text):
    def decorator(func):
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
```

```
        return wrapper
    return decorator
```

这个3层嵌套的decorator用法如下：

```
@log('execute')
def now():
    print('2024-6-1')
```

执行结果如下：

```
>>> now()
execute now():
2024-6-1
```

和两层嵌套的decorator相比，3层嵌套的效果是这样的：

```
>>> now = log('execute')(now)
```

我们来剖析上面的语句，首先执行 `log('execute')` ，返回的是 `decorator` 函数，再调用返回的函数，参数是 `now` 函数，返回值最终是 `wrapper` 函数。

以上两种decorator的定义都没有问题，但还差最后一步。因为我们讲了函数也是对象，它有 `__name__` 等属性，但你去看经过decorator装饰之后的函数，它们的 `__name__` 已经从原来的 `'now'` 变成了 `'wrapper'` ：

```
>>> now.__name__
'wrapper'
```

因为返回的那个 `wrapper()` 函数名字就是 `'wrapper'` ，所以，需要把原始函数的 `__name__` 等属性复制到 `wrapper()` 函数中，否则，有些依赖函数签名的代码执行就会出错。

不需要编写 `wrapper.__name__ = func.__name__` 这样的代码，Python内置的 `functools.wraps` 就是干这个事的，所以，一个完整的decorator的写法如下：

```
import functools

def log(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

或者针对带参数的decorator：

```
import functools

def log(text):
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print('%s %s():' % (text, func.__name__))
            return func(*args, **kw)
        return wrapper
    return decorator
```

`import functools` 是导入 `functools` 模块。模块的概念稍候讲解。现在，只需记住在定义 `wrapper()` 的前面加上 `@functools.wraps(func)` 即可。

练习

请设计一个decorator，它可作用于任何函数上，并打印该函数的执行时间：

```
import time, functools

def metric(fn):
    print('%s executed in %s ms' % (fn.__name__, 10.24))
    return fn
```

```
# 测试
@metric
def fast(x, y):
    time.sleep(0.0012)
    return x + y;

@metric
def slow(x, y, z):
    time.sleep(0.1234)
    return x * y * z;

f = fast(11, 22)
s = slow(11, 22, 33)
if f != 33:
    print('测试失败!')
elif s != 7986:
    print('测试失败!')
```

请编写一个decorator，能在函数调用的前后打印出 `'begin call'` 和 `'end call'` 的日志。

再思考一下能否写出一个 `@log` 的decorator，使它既支持：

```
@log
def f():
    pass
```

又支持：

```
@log('execute')
def f():
    pass
```

参考源码

decorator.py

小结

在面向对象（OOP）的设计模式中，decorator被称为装饰模式。OOP的装饰模式需要通过继承和组合来实现，而Python除了能支持OOP的decorator外，直接从语法层次支持decorator。Python的decorator可以用函数实现，也可以用类实现。

decorator可以增强函数的功能，定义起来虽然有点复杂，但使用起来非常灵活和方便。

« 匿名函数　　偏函数 »

Comments

Comments loaded. To post a comment, please Sign In

造轮子 @ 2025/12/26 05:10:10

```
import time, functools
#使用时间函数：https://docs.python.org/3/library/time.html#time.time
```

```
# 练习一
def metric(fn):
    def wap(*args,**kw):
        pre=time.time() #当前时间函数
        r=fn(*args, **kw)
        cost=time.time()-pre
        print('%s executed in %.3f ms' % (fn.__name__, cost*1000))
        return r
    return wap

# 练习二
def log(arg):
```

Read More ▼

杰 @ 2025/12/25 06:53:23

力求注释清楚。

```
# !/usr/bin/python3
# -*-coding:UTF-8-*-

import functools

def log(arg=None):
    """同时支持无参和有参调用的装饰器"""
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            # 这一段是[log]文本
            if callable(arg):  # 无参，arg是被装饰的函数
                print(f"[log] {arg.__name__}: ")
            else:  # 有参，arg是传入的字符串
```

Read More ▼

杰 @ 2025/12/25 09:06:03

最后一段返回代码有点绕。

```
无参调用（ @log ）：等价于  f1 = log(f1)
有参调用（ @log('execute') ）：等价于  f2 = log("execute")(f2)
```

龙戏天 @ 2025/11/24 01:13:52

先判断一下有没有参数

```
def log(val):
    if isinstance(val, str):
        def decorator(func):
            def wrapper(*args, **kw):
                print('%s %s():' % (val, func.__name__))
                return func(*args, **kw)

            return wrapper

        return decorator
    else:
        def wrapper2(*args, **kw):
            print('%s %s():' % (val, val.__name__))
            return val(*args, **kw)

        return wrapper2
```

大芒果 @ 2025/12/3 09:52:10

```
def log(fn):
    if fn == 'execute':
        def decorator(fn2):
            @functools.wraps(fn2)
            def wrapper(*args, **kw):
                print('%s begin call' % fn)
                ret = fn2(*args, **kw)
                print('%s end call' % fn)
                return ret
            return wrapper
        return decorator
    else:
        @functools.wraps(fn)
        def wrapper(*args, **kw):
            print('begin call')
            ret = fn(*args, **kw)
```

Read More ▼

Hypersomnia @ 2025/12/1 02:04:38

```
import time, functools

def metric(fn):
    @functools.wraps(fn)
    def wrapper(*args, **kw):
        print('begin call %s' % fn.__name__)
        start_time = time.time()
        result = fn(*args, **kw)
        end_time = time.time()
        execution_time_ms = (end_time - start_time) * 1000
        print('%s executed in %s ms' % (fn.__name__, execution_time_ms))
        print('end call %s' % fn.__name__)
        return result
    return wrapper
```

Floris @ 2025/11/27 04:10:01

你就说能不能用吧

```
# coding=utf-8
import time, functools


def metric(fn):
    @functools.wraps(fn)
    def wrapper(*args, **kw):
        startTime = time.time()
        result = fn(*args, **kw)
        endTime = time.time()

        print(f"耗时: {endTime - startTime}")
        return result
```

Read More ▼

Meow @ 2025/11/24 03:40:14

```
import functools

def log(text=None):
    # 情况1: @log  -> text 是函数
    if callable(text):
        func = text
        @functools.wraps(func)
        def wrapper(*args, **kw):
            print(f"调用 {func.__name__}():")
            return func(*args, **kw)
```

```
        return wrapper

    # 情况2: @log() 或 @log("xxx") -> text 是 None 或字符串
    def decorator(func):
        @functools.wraps(func)
        def wrapper(*args, **kw):
```

Read More ▼

雪国骑士 @ 2025/10/13 22:40:57

```
paste code here>>> import time,functools
>>> def metric(text=None):
...     def decorator(fn):
...         @functools.wraps(fn)
...         def wrapper(*args,**kw):
...             result = text if text is not None else fn.__name__
...             print(f'began call and the text is {result}')
...             start_time = time.time()
...             func_result = fn(*args,**kw)
...             end_time = time.time()
...             time_result = (end_time - start_time) * 1000
...             print(f'end call and the time is {time_result}')
...             return fn(*args,**kw)
...         return wrapper
...     if callable(text):
...         func = text
```

Read More ▼

升华 @ 2025/10/31 02:17:53

第一个答案是不是返回func_result好一些

次日清晨. @ 2025/11/4 08:07:20

你这样执行了两遍方法 应该直接返回func_result而不是返回fn(*args,**kw)

我的八见奈🥰🥰🥰 @ 2025/11/21 02:04:37

既然定义了func_result那返回的时候就用func_result就可以

心灵的守护 @ 2025/11/17 22:01:01

简单实现

```
import time, functools

def metric(fn):
    @functools.wraps(fn)
    def wrapper(*args, **kw):
        start_time = time.time()
        a = fn(*args, **kw)
        end_time = time.time()
        total_time = end_time - start_time
        print('%s executed in %s ms' % (fn.__name__, total_time))
        return a
    return wrapper
```

Read More ▼

王大可 @ 2025/9/5 04:33:41

这节好难理解啊

1037号森林公园 @ 2025/9/24 22:33:29

我也是看不懂

升华 @ 2025/10/30 04:00:32

一样，感觉这节的抽象思维要求好高。。

小尾巴 @ 2025/11/5 04:05:40

同感，看不懂代码是怎么运行的

哈集达 @ 2025/11/15 21:12:07

+1

大陆 @ 2025/11/2 07:30:38

```
# case 1
import time, functools

def metric(fn):
    @functools.wraps(fn)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        ans = fn(*args, **kwargs)
        end_time = time.time()
        print('%s executed in %s ms' % (fn.__name__, (end_time-start_time)*1000))
        return ans
    return wrapper

# case 2
def log(fn):
    @functools.wraps(fn)
```

Read More ▼

YC @ 2025/10/8 00:16:22

装饰器函数的固定写法： def outer(func): def inner(): # 执行原函数前触发 func() # 执行原函数后触发 return inner @outer def x(a,b): print() return a+b 执行步骤分析，当代码执行道@outer时,此时会执行装饰器函数将被装饰函数的函数名传递给装饰器函数，outer(x)，然后将outer函数的 返回值返回给被装饰函数: x=outer(x)，再往下执行x函数时，其实此时执行的就不在时x函数，而是inner函数。 而对于需要执行的原函数需要传入参数或存在返回值的情况。 1、传入参数：装饰器函数inner(*args,**kwargs) 标识传入可变参数与关键字参数，可传入任意多个，并再往下执行调用函数时也需要传入，func(*args,**kwargs)。（可作为固定写法） 2、返回值：若原函数存在返回值，若装饰器中不进行对于的返回，会导致代码执行结果有误。 所以装饰器函数需要声明变量存放函数的执行结果 value=func(*args,**keargs),再返回 return value

raffia @ 2025/9/26 05:52:28

```
def metric(fn):
    @functools.wraps(fn)
    def wrapper(*args, **kwargs):
        time_start = time.time()
        result = fn(*args, **kwargs)
        time_end = time.time()
        print(f"{fn.__name__}程序运行时间:{time_end - time_start:.6f}s")
        return fn(*args,**kwargs)
    return wrapper
```

DF11-0045 @ 2025/9/25 22:19:33

import time, functools

def log(text):

```
if callable(text):
    fn = text
    @functools.wraps(fn)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        print('begin call')
        print('call %s():' % fn.__name__)

        result = fn(*args, **kwargs)

        print('end call')
        end_time = time.time()
```

Read More ▼

greensea @ 2025/9/18 22:59:19

```
import time
import functools

def metric(fn):
    @functools.wraps(fn)  # 保持原函数的元信息
    def wrapper(*args, **kwargs):
        start_time = time.time()  # 记录开始时间
        result = fn(*args, **kwargs)  # 执行原函数
        end_time = time.time()  # 记录结束时间
        execution_time = (end_time - start_time) * 1000  # 转换为毫秒
        print('%s executed in %.2f ms' % (fn.__name__, execution_time))
        return result
    return wrapper

# 测试
@metric
```

Read More ▼

k烨 @ 2025/9/16 02:30:07

—

```
import time, functools

def metric(fn):
    @functools.wraps(fn)
    def wrapper(*args,**kw):
        start=time.time()
        fn(*args,**kw)
        end=time.time()
        print('%s executed in %s s' % (fn.__name__, end-start))
        return fn(*args,**kw)
    return wrapper
```

二

Read More ▼

Fredmars @ 2025/9/9 08:45:23

练习1： import time, functools

def metric(fn): @functools.wraps(fn) def wrapper(*args,**kw): print('%s executed in %s ms' % (fn.**name**, time.time())) return fn(*args,**kw) return wrapper

练习2： import time, functools

def metric(fn): @functools.wraps(fn) def wrapper(*args,**kw):

```
    print('begin call %s executed in %s ms' % (fn.__name__, time.time()))
    ret = fn(*args,**kw)
    print('end call %s executed in %s ms' % (fn.__name__, time.time()))
    return ret
```

Read More ▼

2号机 @ 2025/9/7 03:58:02

一

```
def metric(fn):
    @functools.wraps(fn)
    def wrapper(*args, **kw):
        print('%s executed in %s ms' % (fn.__name__, time.time()))
        return fn(*args, **kw)
    return wrapper
```

二

```
def decorator(func):
    @functools.wraps(func)
    def wrapper(*args, **kw):
        print('begin call %s(): %s' % (func.__name__, time.time()))
```

Read More ▼

时光的守望者 @ 2025/9/6 09:45:55

练习 1

```
def timing(fn):
    @functools.wraps(fn)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = fn(*args, **kwargs)
        end = time.time()
        print("%s executed in %.4f s" % (fn.__name__, end - start))
        return result
    return wrapper
```

练习2

```
def log(content=None):
    if callable(content):
```

Read More ▼

---

ss @ 2025/8/21 03:08:49

```
#ex_01
import time, functools

def metric(fn):
    @functools.wraps(fn)  # 保持原函数的名字和文档
    def wrapper(*args, **kwargs):
        start = time.time()  # 记录开始时间
        result = fn(*args, **kwargs)  # 调用原函数
        end = time.time()  # 记录结束时间
        print(f'{fn.__name__} executed in {int((end - start) * 1000)} ms')  # 打印执行时间
        return result
    return wrapper

#ex_02
from functools import wraps

def log_call(func):
    @wraps(func)
```

```
    def wrapper(*args, **kwargs):
        print('begin call')
        result = func(*args, **kwargs)
        print('end call')
        return result
    return wrapper

# 测试
@log_call
def f1():
    print("doing something")

f1()

#ex_03
from functools import wraps

def log(arg=None):
    # 如果装饰器直接用 @log
    if callable(arg):
        func = arg
        @wraps(func)
        def wrapper(*args, **kwargs):
            print('begin call')
            result = func(*args, **kwargs)
            print('end call')
            return result
        return wrapper
    else:
        # 装饰器带参数 @log('execute')
        text = arg
        def decorator(func):
            @wraps(func)
            def wrapper(*args, **kwargs):
                print(f'{text} begin call')
                result = func(*args, **kwargs)
                print(f'{text} end call')
                return result
            return wrapper
        return decorator

# 测试
@log
def f1():
    print("do f1")

@log('execute')
def f2():
    print("do f2")

f1()
f2()
```

Collapse ▲