# 装饰器

## 装饰器

* python中一切皆对象，函数也可以当做参数传递
* 装饰器是接受函数作为参数，添加功能后返回一个新函数的函数（类）
* python中通过@使用装饰器，语法糖

import time
def log\_time(func): # 接受一个函数作为参数
 def \_log(\*args, \*\*kwargs):
 beg = time.time()
 res = func(\*args, \*\*kwargs)
 print('use time: {}'.format(time.time()-beg))
 return res
 return \_log

@log\_time # 装饰器语法糖
def mysleep():
 time.sleep(1)

mysleep()

# 另一种写法

def mysleep2():
 time.sleep(1)

newsleep = log\_time(mysleep2)
newsleep()

使用类编写装饰器

import time
class LogTime:
 def \_\_call\_\_(self, func): # 接受一个函数作为参数
 def \_log(\*args, \*\*kwargs):
 beg = time.time()
 res = func(\*args, \*\*kwargs)
 print('use time: {}'.format(time.time()-beg))
 return res
 return \_log

@LogTime()
def mysleep():
 time.sleep(1)

mysleep()

如何给装饰器增加参数？使用类转时期比较方便实现装饰器参数

import time
class LogTime:
 def \_\_init\_\_(self, use\_int=False):
 self.use\_int = use\_int

 def \_\_call\_\_(self, func): # 接受一个函数作为参数
 def \_log(\*args, \*\*kwargs):
 beg = time.time()
 res = func(\*args, \*\*kwargs)
 if self.use\_int:
 print('use time: {}'.format(int(time.time()-beg)))
 else:
 print('use time: {}'.format(time.time()-beg))
 return res
 return \_log

@LogTime(True)
def mysleep():
 time.sleep(1)

mysleep()

## 再来亿遍

### 简单装饰器

def my\_logging(func):

 def wrapper():
 print("{} is running.".format(func.\_\_name\_\_))
 return func()
 return wrapper

@my\_logging
def foo():
 print("this is foo function.")

foo()

### 带参数的简单装饰器

def my\_logging(func):

 def wrapper(\*args, \*\*kwargs):
 print("{} is running.".format(func.\_\_name\_\_))
 return func(\*args, \*\*kwargs)
 return wrapper

@my\_logging
def foo(x, y):
 print("this is foo function.")
 return x + y

print(foo(1, 2))

### 带参数的装饰器

def my\_logging(level):
 def decorator(func):
 def wrapper(\*args, \*\*kwargs):
 if level == "info":
 print("{} is running. level: ".format(func.\_\_name\_\_), level)
 elif level == "warn":
 print("{} is running. level: ".format(func.\_\_name\_\_), level)
 return func(\*args, \*\*kwargs)
 return wrapper
 return decorator

@my\_logging(level="info")
def foo(name="foo"):
 print("{} is running".format(name))

@my\_logging(level="warn")
def bar(name="bar"):
 print("{} is running".format(name))

foo()
bar()

上面的 my\_logging 是允许带参数的装饰器。它实际上是对原有装饰器的一个函数封装，并返回一个装饰器。我们可以将它理解为一个含有参数的闭包。当使用 @my\_logging(level="info") 调用的时候，Python 能够发现这一层的封装，并把参数传递到装饰器的环境中。

@my\_logging(level="info") 等价于 @decorator

### 类装饰器

装饰器不仅可以是函数，还可以是类，相比函数装饰器，类装饰器具有灵活度大、高内聚、封装性等优点。使用类装饰器主要依靠类的\_\_call\_\_方法，当使用 @ 形式将装饰器附加到函数上时，就会调用此方法。

class MyLogging(object):

 def \_\_init\_\_(self, func):
 self.\_func = func

 def \_\_call\_\_(self, \*args, \*\*kwargs):
 print("class decorator starting.")
 a = self.\_func(\*args, \*\*kwargs)
 print("class decorator end.")
 return a

@MyLogging
def foo(x, y):
 print("foo is running")
 return x + y

print(foo(1, 2))

## 参考资料

装饰器的部分讲的不错

https://gitbook.cn/books/5ca40fd11763103ff10b0e43/index.html